Mixed-Criticality Scheduling and Resource Sharing for High-Assurance Operating Systems

Anna Lyons

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering

University of New South Wales

Sydney, Australia

March 2018

Thesis/Dissertation Sheet

Surname/Family Name	:	**Lyons**
Given Name/s	:	**Anna Mary**
Abbreviation for degree as give in the University calendar	:	**PhD**
Faculty	:	**Engineering**
School	:	**Computer Science**
Thesis Title	:	**Mixed-Criticality Scheduling and Resource Sharing for High-Assurance Operating Systems**

Abstract 350 words maximum: (PLEASE TYPE)

Criticality of a software system refers to the severity of the impact of a failure. In a high-criticality system, failure risks significant loss of life or damage to the environment. In a low-criticality system, failure may risk a downgrade in user-experience. As criticality of a software system increases, so too does the cost and time to develop that software: raising the criticality also raises the assurance level, with the highest levels requiring extensive, expensive, independent certification.

For modern cyber-physical systems, including autonomous aircraft and other vehicles, the traditional approach of isolating systems of different criticality by using completely separate physical hardware, is no longer practical, being both restrictive and inefficient. The result is mixed-criticality systems, where software applications with different criticalities execute on the same hardware. Sufficient mechanisms are required to ascertain that software in mixed-criticality systems is sufficiently isolated, otherwise, all software on that hardware is promoted to the highest criticality level, driving up costs to impractical levels. For mixed-criticality systems to be viable, both spatial and temporal isolation are required.

Current aviation standards allow for mixed-criticality systems where temporal and spatial resources are strictly and statically partitioned in time and space, allowing some improvement over fully isolated hardware. However, further improvements are not only possible, but required for future innovation in cyber-physical systems.

This thesis explores further operating systems mechanisms to allow for mixed-criticality software to share resources in far less restrictive ways, opening further possibilities in cyber-physical system design without sacrificing assurance properties. Two key properties are required: first, time must be managed as a central resource of the system, while allowing for overbooking with asymmetric protection without increasing certification burdens. Second, components of different criticalities should be able to safely share resources without suffering undue utilisation penalties. The implementation platform is the seL4 microkernel, which is built for the safety-critical, high assurance domain.

Originality Statement

‘I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation and linguistic expression is acknowledged.’

Signed .....

Date .....

Abstract

Criticality of a software system refers to the severity of the impact of a failure. In a high-criticality system, failure risks significant loss of life or damage to the environment. In a low-criticality system, failure may risk a downgrade in user-experience. As criticality of a software system increases, so too does the cost and time to develop that software: raising the criticality also raises the assurance level, with the highest levels requiring extensive, expensive, independent certification.

For modern cyber-physical systems, including autonomous aircraft and other vehicles, the traditional approach of isolating systems of different criticality by using completely separate physical hardware, is no longer practical, being both restrictive and inefficient. The result is *mixed-criticality* systems, where software applications with different criticalities execute on the same hardware. Sufficient mechanisms are required to ascertain that software in mixed-criticality systems is sufficiently isolated, otherwise, all software on that hardware is promoted to the highest criticality level, driving up costs to impractical levels. For mixed-criticality systems to be viable, both spatial and temporal isolation are required.

Current aviation standards allow for mixed-criticality systems where temporal and spatial resources are strictly and statically partitioned in time and space, allowing some improvement over fully isolated hardware. However, further improvements are not only possible, but required for future innovation in cyber-physical systems.

This thesis explores further operating systems mechanisms to allow for mixed-criticality software to share resources in far less restrictive ways, opening further possibilities in cyber-physical system design without sacrificing assurance properties. Two key properties are required: first, time must be managed as a central resource of the system, while allowing for overbooking with asymmetric protection without increasing certification burdens. Second, components of different criticalities should be able to safely share resources without suffering undue utilisation penalties. The implementation platform is the seL4 microkernel, which is built for the safety-critical, high assurance domain.

Contents

Acknowledgements

It's a common aphorism to state that research outcomes "stand on the shoulders of giants", referring to all the research and hard work that the outcomes build upon. It's definitely true, however I'd like to personally thank those who are rarely recognised: the carers, the cleaners, the cooks, the friends and family, the physiotherapists, the detectives, therapists, and all of the people who form a support group around a person. It takes a village to raise a child, and the same goes for a PhD. In both cases, there's a lot of thankless, gruelling work, which might sound similar, but you don't get a certificate and floppy hat at the end. You're all giants to me.

I'd like to thank the following people for their help and support: my partner Kara Hatherly, Debbie & Geoff Fear, Biara Webster, Ryan Williams and his team, Patricia Durning and Gary Penhall. I would *never* have finished this project without you.

I thank my supervisor, Gernot Heiser, for encouraging me, giving me the time I needed, and ultimately proving the support that led to me complete the project in the face of extreme external events. My co-supervisor, Kevin Elphinstone, for being drawn into hours of whiteboard discussion, after I waited earnestly at the coffee machine. Additionally all of the students, engineers and researchers who have helped at some point or another: David Greenaway, Thomas Sewell, Adrian Danis, Kent McLeod, Yanyan Shen, Luke Mondy, Hesham Almatary and Qian Ge. Further thanks to everyone in the Trustworthy Systems team, past and present, you're all fantastic.

Matthew Fernandez, Rafal Kolanski, and Joel Beeren get a special thanks not only for help with the work, but for supporting me in the lab, and the pub, and being my friends even at my worst, you were a huge factor in keeping me sane.

Finally I'd like to thank my veritable army of proof readers: Kara Hatherly, Kent McLeod, Peter Chubb, Tim Cerexhe, Alex Legg, Robert Sison, Joel Beeren, Sebastian Holzapfel, Sean Peters, Ihor Kuz and of course, Gernot Heiser.

Publications

This thesis is partially based on work described in the following publications:

- Anna Lyons, Kent Mcleod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM
- Anna Lyons and Gernot Heiser. Mixed-criticality support in a high-assurance, general-purpose microkernel. In *Workshop on Mixed Criticality Systems*, pages 9–14, Rome, Italy, December 2014

Acronyms

AAV autonomous aerial vehicle. 1, 34

AES advanced encryption standard. 92, 129–131, 140, 142

API application programming interface. 95

APIC advanced PIC. 126

ASID address space ID. 61

BMK bare metal kernel. 126

BWI bandwidth inheritance. 54, 90

CA certification authority. 3, 32, 33

CBS constant-bandwidth server. 30, 31, 34, 36, 43–49

CFS completely-fair scheduler. 43

CPU central processing unit. 47, 127, 146

DM deadline monotonic. 14, 15

DROPS Dresden Real-time OPerating system. 47, 49

DS deferrable server. 45, 48, 49

EDF earliest deadline first. 14–17, 19, 21, 28–30, 33, 36, 43–47, 53, 75, 82, 115, 135, 146

FIFO first-in first-out. 21, 41–43, 48, 63, 83

FP fixed priority. 14–17, 19, 21, 28, 33, 34, 44–46, 53, 75

FPRM fixed-priority rate monotonic. 14, 15, 33, 34, 36, 44, 71, 76

FPU floating point unit. 116

GPS global positioning system. 2, 5

HLP highest lockers protocol. 40, 44

HRT hard real-time. 10, 14, 17, 28–30, 33, 36, 44, 49, 73, 78, 82, 86

IPC inter-process communication. 22, 36, 43, 50–52, 55, 58, 59, 63, 65, 69, 70, 74, 76, 80, 83–85, 87, 89, 90, 92, 93, 96, 108, 109, 114, 115, 118–123, 130, 134, 143, 161

IPCP immediate priority ceiling protocol. 18–20, 35, 42, 83, 84, 131

MCAR Mixed Criticality Architecture Requirements. 1

MCP maximum controlled priority. 103, 108

MCS mixed criticality system. 104, 118

MMU memory management unit. 8, 27, 41, 44

MPCP multiprocessor priority ceiling protocol. 21

MPU memory protection unit. 44

MSI message-signalled interrupts. 126

MSR model-specific register. 117

MSRP multiprocessor stack resource policy. 21

NCET nominal-case execution time. 34

NCP non-preemptive critical sections protocol. 18–20

OCET overloaded-case execution time. 34

OPCP original priority ceiling protocol. 19–21, 35, 83, 84, 91

OS operating system. 1, 3, 7, 15, 21–24, 39, 40, 42, 44, 46–48, 50, 53, 55, 61, 110, 113

PIC programmable interrupt controller. 126

PIP priority inheritance protocol. 18–20, 35, 40, 44, 51, 54, 83, 84, 86, 90–92

POSIX portable operating system interface. 39, 40, 43, 44

QoS quality of service. 34

RBED rate-based earliest deadline first. 30, 33, 48

RM rate monotonic. 14–16, 34, 75

RPC remote procedure call. 50, 63, 71, 82, 92, 109, 111

RTA response time analysis. 81

RTOS real-time operating system. 23, 24, 28, 39, 44–46

SC scheduling context. 76–80, 83, 91, 93

SCO scheduling context object. 95, 97–99, 103, 106–110, 121

SMP symmetric multiprocessor. 8

SRP stack resource policy. 19, 21, 35

SRT soft real-time. 10, 14, 27, 28, 30, 33, 36, 42, 44, 49, 73, 80, 132

SS sporadic server. 36, 43, 45, 46, 49

SWaP size, weight and power. 1, 4, 6

TCB thread control block. xxiii, 51, 58–60, 63, 65, 77, 95, 97, 98, 103, 105–108, 110, 113, 114, 118–120, 131, 132

TLB translation lookaside buffer. 119, 121

TSC timestamp counter. 116, 117

UDP User Datagram Protocol. 128, 129

VM virtual machine. 128

WCET worst-case execution-time. 2, 4, 10, 11, 14, 24, 27, 29, 30, 32–34, 36, 44, 45, 54, 55, 71, 82, 86, 92, 111

YCSB Yahoo! Cloud Serving Benchmark. 126

ZS zero slack. 34, 36, 44

List of Figures

List of Tables

List of Listings

1 Introduction

Criticality of a software system refers to the severity of the impact of a failure. In a high-criticality system, failure risks significant loss of life or damage to the environment. In a low-criticality system, failure may risk a downgrade in user-experience. As criticality of a software system increases, so too does the cost and time to develop that software: raising the criticality also raises the assurance level, with the highest levels requiring extensive, expensive, independent certification.

For modern cyber-physical systems, including autonomous aircraft and other vehicles, the traditional approach of isolating systems of different criticality by using completely separate physical hardware is no longer practical, being both restrictive and inefficient. The result is *mixed-criticality* systems, where software applications with different criticalities execute on the same hardware. Sufficient mechanisms are required to ascertain that software in mixed-criticality systems is isolated. Otherwise, all software on shared hardware is promoted to the highest criticality level, driving up costs to impractical levels. For mixed-criticality systems to be viable, both spatial and temporal isolation are required.

However, mixed-criticality systems have conflicting requirements that challenge operating systems (OS) design: they require distrusting components of different criticalities to share resources and must degrade gracefully in the face of failure. For example, an autonomous aerial vehicle (AAV) has multiple inputs to its flight-control algorithm: object detection, to avoid flying into obstacles, and navigation, to get to the desired destination. Clearly the object detection is more critical than navigation, as failure of the former can be catastrophic, while the latter would only result in a non-ideal route. Yet the two subsystems must cooperate; accessing and modifying shared data thus cannot be fully isolated.

The AAV is an example of a mixed-criticality system, a notion that originates in avionics and its need to reduce size, weight and power (SWaP) by consolidating growing functionality onto a smaller number of physical processors. More recently the Mixed Criticality Architecture Requirements (MCAR) [Barhorst et al., 2009] program was launched, which recognises that in order to construct fully autonomous systems, critical and less critical systems must be able to share resources. However, resource sharing between components of mixed-criticality is heavily restricted by current standards.

While MCS are becoming the norm in avionics, this is presently in a very restricted form: in the ARINC standard, the system is orthogonally partitioned spatially and temporally, and partitions are scheduled with fixed time slices [ARINC]. Shared resources are permitted but can only be accessed through fixed-time partitions; in other words, a critical component must not trust a lower criticality one to behave correctly. This limits integration and cross-partition communication, and implies long interrupt latencies and poor resource utilisation. These challenges are not unique to avionics: top-end cars exceeded 70 processors ten years ago [Broy et al., 2007]; with the robust packaging and wiring required for vehicle electronics, the SWaP problem is obvious, and will be magnified by the move to more autonomous operation. Other classes of cyber-physical systems, such as smart factories, will experience similar challenges.

One could reduce the number of separate processors without mixing criticalities by simply consolidating systems of the same criticality level, however, this is does not lead to efficient use of resources. High-criticality systems that are subject to high levels of assurance have very high worst-case execution-time (WCET) estimates, which are often orders of magnitude beyond the average execution time due to the limitations in analysis, and the complexity of modern hardware [Wilhelm et al., 2008]. However, in current systems the full WCET must be statically reserved for the high criticality tasks. By building mixed-criticality systems, where low-criticality tasks—for which temporal interference can lead only to degradation in service—we can leverage the unused capacity from high-criticality tasks.

Fundamental to building mixed-criticality systems is the concept of *asymmetric protection*, which differs from strict temporal isolation in that only the highest criticality systems are guaranteed top levels of service. Less-critical software is not completely isolated, and can be affected by high-criticality software, but not vice versa.

Finally, mixed-criticality systems are about more than basic consolidation and leveraging excess resources; they can achieve more than traditional physically isolated systems by allowing actual resource sharing. Consider the driving software of a self-driving car: it may take inputs from various sensors which detect objects, and also input from a global positioning system (GPS) navigation unit, and traffic data from an internet-connected service providing alternate route information. All of these systems *must* provide input to the shared driving software, which actually makes decisions about what commands to issue to the car. Clearly, the object detection is more critical than the GPS, which in turn is more critical than the route information service. In fact, the route information service—being connected to the internet—is not even a trusted service: it is subject to attacks, such as denial-of-service.

Allowing untrusted, less critical components to share hardware and communicate with more critical components, far more complex software can be introduced to the system. Examples include heuristics that are common in artificial intelligence algorithms, or internet-connected software which by its nature cannot be completely trusted. Both of these use-

Level	*Impact*
Catastrophic	Failure may cause multiple fatalities, usually with loss of the airplane.
Hazardous	Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
Major	Failure significantly reduces the safety margin or significantly increases crew workload. May result in passenger discomfort (or even minor injuries).
Minor	Failure slightly reduces the safety margin or slightly increases crew workload. Examples might include causing passenger inconvenience or a routine flight plan change.
No Effect	Failure has no impact on safety, aircraft operation or crew workload.

Table 1.1: Criticality levels from DO-178C, a safety standard for commercial aircraft.

cases are far too complex to certify to the highest criticality standard, but are essential for emerging cyber-physical systems like self-driving cars.

Our goal is to develop trustworthy, high-assurance OS mechanisms that provide for the unique requirements of mixed-criticality systems, without requiring all systems to be promoted to the highest criticality in the system. The implementation platform will be the seL4 [Klein et al., 2009] microkernel, which is has been designed for the high-assurance, safety-critical domain.

Concisely, the goals of this research are to provide:

G1 A principled approach to processor management, treating time as a fundamental kernel resource, while providing mechanisms for asymmetric protection, a key requirement of mixed-criticality systems;

G2 Safe resource sharing between applications of different criticalities, assurance levels and temporal requirements.

1.1 Motivation

As noted in the introduction, the *criticality* of a system reflects the severity of failure, where higher criticality implies higher severity. Table 1.1 shows criticality levels considered when designing software for commercial aircraft in the United States.

As the criticality level rises, so too do the assurance levels: higher engineering and safety standards are required for higher criticality levels, up to certification by independent certification authorities (CAs) at the highest level, which is a timely, expensive and restrictive process. Additionally, requirements specified by CAs are highly cautious, and

pessimistic, which when combined with safety strategies like redundant resources, leads to large amounts of excess capacity in processing power.

Consequently, highly critical software is incredibly expensive to develop and tends to have low complexity in order to minimise costs. Any software that is not fully isolated from a critical component is promoted to that level of criticality, increasing the production cost, and imposing strict requirements.

Traditionally, systems of different criticality levels were fully isolated with air gaps between physical components. However, given the increased amount of computing in every part of our daily lives, the practice of physical isolation has resulted in unscalable growth in the amount of computing hardware in embedded systems, with some modern cars containing over 100 processors [Hergenhan and Heiser, 2008]. The practice of physical separation is no longer viable for three reasons: SWaP, efficiency, and resource sharing.

1.1.1 SWaP

First, systems with air-gaps require increased physical resources each extra processor comes with extra cabling and power requirements: increasing production costs and environmental impact. For vehicles, especially aircraft, this goes further to reduce their function; the heavier the system, the more fuel it requires, especially when considering the redundancy built into these systems for safety purposes: often each component is double- or triple-redundant.

1.1.2 Efficiency

However, SWaP alone could be reduced by consolidating systems of the same criticality. Ignoring the fact that this also consolidates points of failure (redundant systems by definition, cannot be consolidated), this alone is insufficient to completely address SWaP challenges. The higher assurance required of a system, the more over-provisioned the hardware is: not only in redundancy but in excess capacity. For example, WCET estimates may be orders of magnitude larger than the typical execution time, and computation of safe WCET bounds for non-trivial software tends to be highly pessimistic [Wilhelm et al., 2008]. At the same time, the core *integrity property* of a high-criticality, real-time system is that deadlines must always be met, meaning that there must always be time to let such threads execute their full *WCET*. Consequently, consolidating high-criticality systems only ameliorates the problem slightly: the pessimism inherent in high-assurance systems is also consolidated, leaving unused, excess hardware resources.

High-utilisation of hardware to address SWaP challenges is therefore only possible with the advent of mixed-criticality systems, which can leverage the excess capacity for less critical, less time-sensitive software.

System	*Purpose*	*Criticality*
Obstacle detection	Safety: failure can lead to collision, which could lead to significant damage and loss of life.	Catastrophic
GPS Navigation	Route planning: failure can lead to an incorrect turn and further time travelling.	Minor
Traffic service.	Route planning: failure can lead to a slower trip, by failing to avoid congested paths.	Minor

Table 1.2: Fictional example systems in a self-driving car.

1.1.3 Resource Sharing

Mixed-criticality systems also bring opportunities for new types of systems, and are indeed required for emerging cyber-physical systems like advanced driver assistance systems, autonomous vehicles and aircraft.

Consider the systems required for a self-driving car to determine what commands to issue the vehicle next, as outlined in Table 1.2. The object-detection is the most critical, and is generally comprised of a host of sensors: infra-red, LIDAR, cameras, and RADAR, which themselves are subject to different criticality levels. Object-detection, especially from image data, is a function of the amount and complexity of data available: when driving in desert, compared to driving in a blizzard, far less input is available. It is also the most critical system in our example, with failure leading to potentially disastrous collisions.

However, a GPS navigation service is also required to determine the route to take, and finally a traffic service which allows cars to avoid congestion. Although the same criticality, the GPS service is likely more highly-assured than the traffic service: the traffic service is connected to the internet, requiring complex network drivers, and subject to attacks from remote attackers. In fact, the fastest way to integrate a traffic service is to run a large, open source code base with existing network drivers like Linux, which has no assurance level.

All three systems are important to the car manufacturer: a bad review due to poor navigation, or not avoiding a traffic jam, can have a marked impact on sales. The three systems must access the same shared resource (the driving logic), which requires the system to be mixed-criticality: the traffic service, and to a lesser extent the GPS, are not practical to develop at high assurance levels. In order to gain high utilisation, the resources must be overbooked: the traffic service may have a low-level guarantee to some hardware resources, but the object-detection service may infringe on that resource occasionally. If the traffic service is statically assigned its resource allocation, its utility may decrease, as it can run more frequently when the object-detection does not require that time. Similarly, when stuck in traffic, the GPS service isn't much use: the traffic service is of far more utility to find alternate routes. Conversely, when on a highway at high speed, the GPS service requires more processing time, and the traffic service can run less frequently.

Consequently, mixed-criticality shared resources are integral to future cyber-physical systems, something that is not possible with traditional separation of physical hardware. At the same time, for high system utilisation, asymmetric protection, and over-booking are required to address SWaP challenges. Finally, without sufficient mechanisms, none of this system is possible, as developing such a system to high assurance levels is not practical.

1.2 Definitions

Mixed-criticality In the real-time community, much work has been conducted around a theoretical model introduced by Vestal [2007] which we discuss in Section 3.2. The focus on this thesis is for the broader definition of mixed-criticality, such that software of different levels of criticality can safely share hardware, and not specifically the model presented by Vestal [2007] and beyond [Burns and Davis, 2017], although it is discussed in Chapter 3.

Temporal isolation For a system A to be temporally isolated from system B, B cannot cause temporal failures in A. This does not necessarily mean that B cannot affect A, but that any temporal effect B can have on A is deterministic and bounded, such that A is correct regardless of B.

Asymmetric Protection In order to leverage an overbooked system, high-criticality tasks must be able to cause failures in low-criticality tasks but not vice-versa. Another way to state this is that *criticality inversion* must be bounded, where a low-criticality task must not cause failures in a high-criticality task.

1.3 Objectives

Given their improvements to SWaP, function and efficiency, mixed-criticality systems offer great advantages over the traditional physical isolation approach. Ernst and Di Natale [2016] identify two sets of mechanisms that need to be provided in order to support true mixed-criticality systems:

1. operating system kernels and schedulers that guarantee resource management to provide independence in the functional and time domain; separation kernels are the most notable example;
2. mechanisms to detect timing faults and control them, including monitors, and the scheduling provisions for guaranteeing controllability in the presence of faults.

1.4 Approach

In this thesis we look to systems and real-time theory related to scheduling, resource allocation and sharing, criticality and trust to develop a set of mechanisms required in an

OS to support mixed criticality systems. We implement those mechanisms in seL4 and develop a set of microbenchmarks and case studies to evaluate the implementation.

Chapter 2 establishes the basic terminology required to present the remaining ideas, including an introduction to real-time scheduling and resource sharing theory. We also introduce the concept of a resource kernel, an existing approach to providing principled access to time as a first class resource.

In Chapter 3 we examine in detail the theoretical models and methods for achieving temporal isolation and safe resource sharing in traditional, real-time systems and show that sufficient theory exists for resource-overbooking and asymmetric protection.

Chapter 4 investigates systems support for the same concepts; by surveying existing commercial, open-source and research operating systems we show that in the current state of the art, support for resource overbooking and asymmetric protection is insufficient for modern, cyber-physical systems

Chapter 5 presents the final piece of background, presenting the existing concepts which make our implementation platform, seL4, an excellent target for high-assurance systems with its existing spatial-resource isolation mechanisms. However, we also show that like the other operating systems considered in the chapter before it, current mechanisms are insufficient for temporal isolation, asymmetric protection, and resource overbooking.

We then present in Chapter 6 a model for mechanisms required to build mixed-criticality operating systems, including capability-based access control to processing time which allows for overbooking and does not limit user-level policy. Additionally, we provide new mechanisms for resources that can be shared safely between systems of different criticalities. Our model is designed to integrate with seL4, however has wider applications to OS design for mixed-criticality systems.

Chapter 7 delves deeply into the implementation details, presenting a full, mature implementation of our model along with discussions of design trade-offs and limitations. Finally, Chapter 8 presents a detailed set of benchmarks and case studies on a variety of x86 and ARM platforms, showing the overheads of our implementation, demonstrating temporal isolation in systems with and without shared resources, and showing how resource overbooking and asymmetric protection can be achieved.

1.5 Contributions

We make the following specific contributions:

- We develop a design and model for fine-grained access control to processing time, without imposing strict limitations on system policy or introducing performance overheads, and without requiring sacrifices to integrity or confidentiality;

- an implementation of the model in the high-assurance, non-preemptible seL4 microkernel, and an exploration of the simplicity of the model and its integration with the existing model which provides strong isolation properties;
- an evaluation of the implementation, demonstrating low overheads and temporal isolation between systems sharing a processor and other resources;
- user-level implementations of dynamic schedulers and resource sharing servers which demonstrate that a variety of policies are not only possible to implement, but low overhead.

1.6 Scope

We focus on uniprocessor and symmetric multiprocessor (SMP) systems with memory management units (MMUs) for ARM and x86, where our focus is on safety, through integrity and availability. We leave security concerns such as covert channels to future work. Additionally, we assume a constant processor speed, and leave variable-speed processors and power-saving mechanisms to future work.

2 Core concepts

In this section we provide the background required to motivate and understand our research. We introduce real-time theory, including scheduling algorithms and resource sharing protocols. In addition, we define operating systems, microkernels and introduce the concept of a resource kernel. This thesis draws on all of these fields in order to support real-time and mixed-criticality systems.

2.1 Real-time theory

Real-time systems have timing constraints, where the correctness of the system is dependent not only on the results of computations, but on the time at which those results arrive [Stankovic, 1988]. Essentially all software is real-time: if the software never gets access to processing time, no results will ever be obtained. However, for a system to be considered real-time it must be sensitive to timing behaviour—how much processing time is allocated and when it is allocated. For most software, time is fungible: it does not matter when the software is run, as long as it does get run. For real-time software this is not the case. Consider the case of software that deploys the brakes on a train: the routine must run in time for the train to stop at the platform. Or, in a more critical scenario, software that deploys the landing gear on a plane: it must run and the gear must be fully down before the plane hits the runway.

There are many ways to model real-time systems, which we touch on in the coming chapters.

2.1.1 Types of real-time tasks

The term *task* in real-time theory is used to refer to a single instruction stream, which in operating systems terms is referred to as a thread. How tasks are realised by an operating system—with or without memory protection, for example—is specific to the implementation. Computations by tasks in real-time systems have deadlines which determine the correctness of the system. How those deadlines effect correctness depends on the category

of the system, which is generally referred to as hard real-time (HRT), soft real-time (SRT), or *best-effort*.

Hard Real-Time tasks have absolute deadlines; if a deadline is missed, the task is considered incorrect. HRT tasks are most common in safety-critical systems, where deadline misses lead to catastrophic results. Examples include airbag systems in cars and control systems for autonomous vehicles. In the former, missing a deadline could cause an airbag to be deployed too late. In the latter, the autonomous vehicle could crash. To guarantee that a HRT task can meet deadlines, the code must be subject to WCET analysis. State of the art WCET analysis is generally pessimistic by several orders of magnitude, which means that allocated time will not generally be used completely, although it must be available. WCET is pessimistic for multiple reasons: firstly, much behaviour like interrupt arrival times, cannot be predicted but only bounded, so the WCET includes the worst possible execution case. Secondly, accurate models of modern hardware are few, so often the WCET must assume caches are not primed and possibly conflicting. Further detail about WCET analysis can be found in Lv et al. [2009].

Soft Real-Time tasks, as opposed to HRT, can be considered correct in the presence of some defined level of deadline misses. Although WCET can be known for SRT tasks, less precise but more optimistic time estimates are generally used for practicality. Examples of SRT tasks can be found in multimedia and video game platforms, where deadline misses may result in some non-critical effect such as performance degradation. SRT tasks can also be found in safety-critical systems where the result degrades if not enough time has been allocated by the deadline, but the result remains useful e.g. image recognition and object detection in autonomous vehicles. In such tasks, a minimum allocation of time to the task might be HRT, while further allocations are SRT.

Various models exist to quantify permissible deadline misses in SRT systems. One measure is to consider the upper bound of how much the deadline is missed, referred to as *tardiness* [Devi, 2006]. Another is to express an upper bound on the percentage of deadline misses permitted for the system to be considered correct, which is easier to measure but less meaningful. Some SRT systems allow deadlines to be skipped completely [Koren and Shasha, 1995] while others allow deadlines to be postponed. Systems that allow deadlines to be skipped are often referred to as firm real-time e.g. media processing, where some frames can be dropped.

best-effort tasks do not have temporal requirements, and generally execute in the background of a real-time system. Examples include logging services and standard applications, but may be far more complicated. Of course, any task must be scheduled at some point for system success, so there must be some timing guarantee.

2.1.2 Real-time models

A real-time system is a collection of tasks, traditionally of the same model. If a system is referred to as SRT, then all tasks in that system are SRT. For analysis, each task in a

```
for (;;) {
  // job arrives
  doJob();
  // job completes before deadline
  // sleep until next job is ready
  sleep();
}
```

Listing 2.1: Example of a basic sporadic real-time task.

real-time system is modelled as an infinite series of *jobs*. Each job represents a computation with a deadline. One practical model in common use is the *sporadic task model* [Sprunt et al., 1989], which can model both *periodic* and *aperiodic* tasks and maps well onto typical control systems.

The sporadic task model considers a real-time system as a set of tasks, $A_1, A_2, \ldots, A_n$. Each A_i refers to a specific task in the task set, usually each for a different functionality. The infinite series of jobs is per task, and represented by $A_{i1}, A_{i2}, \ldots, Ain$. Each task has a WCET, C_i, and period, T_i, which represents the minimum inter-arrival time between jobs. When a T_i has passed and a job can run again, it is said to be *released*. When a job is ready to run, it is said to have *arrived* and when a job finishes and blocks until the next arrival, it is said to be *completed*.

For periodic tasks, which are commonly found in control systems, jobs release and arrive at the same time; they are always ready to run once their period has passed. For aperiodic, i.e. interrupt driven tasks, the arrival time can be at or beyond the release time.

Jobs must complete before their period has passed ($C_i \leq T_i$), as the model does not support jobs that run simultaneously. Tasks with interleaving jobs must be modelled as

Notation	*Meaning*
A	A task set.
A_i	A specific task in a task set
A_{ij}	A specific job of a specific task set.
T_i	The minimum period of a task, the minimum time between job releases.
C_i	The WCET of a task ($C_i \leq T_i$).
U_i	The maximum utilisation of task i, which is $\frac{C}{T}$
D_i	The relative deadline of task i
t_{ij}	The release time of the jth job of task i
d_{ij}	The deadline for the jth job of task i.
a_{ij}	The arrival time of the jth job of task i, $a_{ij} \geq t_{ij}$
n	The number of tasks in a task set

Table 2.1: Parameters and notation for the sporadic task model.

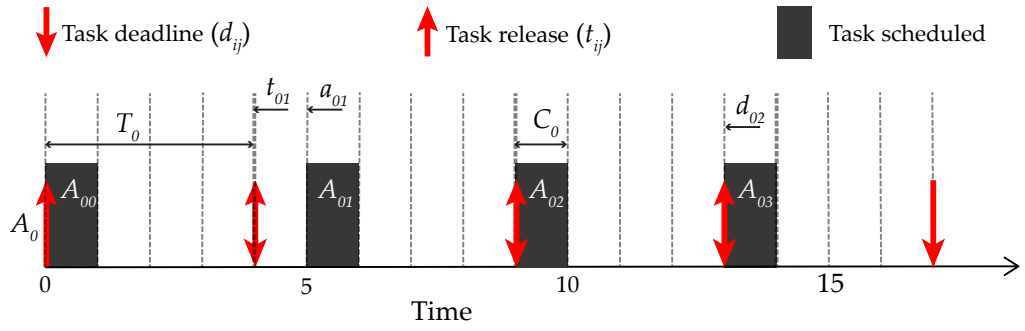


Figure 2.1: Diagram of the sporadic task model notation described in Table 2.1, showing a task set A with a single task A_0 where $T_0 = 4$ and $C_0 = 1$.

separate tasks which do not overlap. Listing 2.1 shows pseudocode for a sporadic task and Table 2.1 summarises the notation which is used throughout this thesis.

Sporadic tasks have deadlines relative to their arrival time, which may be *constrained* or *implicit*. An *implicit* deadline means the job must finish before the period elapses. *Constrained* deadlines are relative to the release time, but before the period elapses. Other task models allow for *arbitrary* deadlines, which can be after the period elapses, however arbitrary deadlines are not compatible with the sporadic task model without adjustment, as they allow multiple jobs from one task to be active at one time. In order to model tasks with arbitrary deadlines with the sporadic task model, they must be split into multiple, different tasks with larger periods and constrained deadlines.

2.2 Scheduling

Simply put, scheduling is deciding which task to run at a specific time. In operating systems terms, this is deciding which thread, or process to run on the processor next. More formally, scheduling is the act of assigning resources to activities or tasks [Baruah et al., 1996], generally discussed in the context of processing time, but is also required for other resources such as communication channels. A correct scheduling algorithm can guarantee that all deadlines are met within their requirements, whilst also maintaining maximum utilisation of the scheduled resource(s). Real-time scheduling algorithms differ from their non-real-time equivalents in that they must guarantee that all tasks not only receive sufficient access to the processor, but that all tasks meet their timing requirements, i.e. deadlines.

Scheduling can be either *static* or *dynamic*. In a static or *off-line* schedule, the set of tasks is fixed and the schedule pre-calculated when the system is built, and does not change once the system is running. Dynamic or *on-line* scheduling occurs while the system is running, and the schedule is calculated whenever a scheduling decision is required. In dynamic systems, the task set can change, allowing task parameters to be adjusted, and

	C_i	T_i	U_i
A_1	1	4	0.25
A_2	1	5	0.20
A_3	3	9	0.33
A_4	3	18	0.17
$U_{sum}(\tau)$			0.95

Table 2.2: A sample task set, adapted from [Brandenburg, 2011]

allowing tasks to be added and removed while the system is running. All schedulers in known operating systems like Linux, are dynamic, online schedulers.

All possible permutations of task parameters are not schedulable, and optimal scheduling is equivalent to a bin-packing problem. Thus, we refer to feasible and non-feasible task sets. A task set is *feasible* if it can be scheduled by at least one scheduling algorithm such that all temporal requirements are satisfied, and *non-feasible* if no scheduling algorithm can be found that satisfies the requirements. An *optimal* scheduling algorithm can schedule every feasible task set. To test if a task set is schedulable by a scheduling algorithm, a *schedulability test* is applied. The complexity of schedulability tests is important for both dynamic and static schedulers. For static schedulers, the test is conducted offline and not repeated, but the algorithm must complete in a reasonable time for the number of tasks in the task set. Dynamic schedulers conduct a schedulability test each time a new task is submitted to the scheduler, or scheduling parameters are altered, which means the test cannot be too complex to reduce overheads. The schedulability test conducted at admission time is referred to as an *admission test*.

There is an absolute limit on task sets that are feasible which can be derived from the total utilisation. The *total utilisation* of a task set is the sum of all the rates and must be less than the total processing capacity of a system for all deadlines to be met. For each processor in a system, this amounts to Eq. (2.1).

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

If the inequality does not hold, the system is considered *overloaded*. Overload can be *constant*, in that it is all the time, or *transient*, where the overload may be temporary due to exceptional circumstances.

Ideally, task sets are scheduled such that the total utilisation is equal to the number of processors. In practice, scheduling algorithms are subject to two different types of *capacity loss* which render 100% utilisation impossible—algorithmic and overhead-related. *Algorithmic* capacity loss refers to processing time that is wasted due to the schedule used, due to a non-optimal scheduling algorithm. *Overhead-related* capacity loss refers to time spent due to hardware effects (such as cache misses, cache contention, and context

switches) and computing scheduling decisions. Accurate schedulability tests should account for overhead-related capacity loss in addition to algorithmic capacity loss.

Tasks often do not use all of their execution requirement, any execution remaining in is referred to as *slack*. For HRT tasks, slack is a consequence of pessimistic WCET values. Slack also occurs for SRT tasks as they may vary in length. Slack also occurs for aperiodic tasks where the actual arrival time varies from the minimum inter-arrival time. Many scheduling algorithms attempt to gain performance by reclaiming or stealing slack.

Scheduling algorithms are classed as either dynamic or fixed priority. A scheduling algorithm is *fixed priority* if all the jobs in each task run at the same priority, which never changes. *Dynamic priority* scheduling algorithms assign priorities to jobs not tasked, based on some criteria. There are two definitive dynamic scheduling algorithms: fixed-priority rate monotonic (FPRM), which has fixed priorities, and earliest deadline first (EDF), which has dynamic priorities. Each is optimal for its respective scheduling class, and both algorithms are defined along with schedulability tests for the periodic task model in the seminal paper by Liu and Layland [1973]. We describe them briefly here, after covering the dominant static scheduling algorithm: cyclic executives.

2.2.1 Cyclic executives

A *cyclic-executive* is an offline, static scheduler that dispatches tasks according to a pre-computed schedule, and is only suitable for closed systems. Each task has one or more entries in the pre-computed schedule, referred to as *frames*, which specify a WCET. Frames are never preempted while running, resulting in a completely deterministic schedule. The cyclic executive completes in a *hyperperiod*, which is the least common multiplier of all task periods. The *minor cycle* is the greatest common divisor of all the task periods.

Cyclic executives can, in theory, schedule task sets where the total utilisation is 100%, as in Eq. (2.1). However, this only holds in a task model where tasks can be split into infinite chunks, as tasks that do not fit into the minor cycle must be split. This assumption is unrealistic as many tasks cannot be split, and task switching is not without overheads. Calculating a cyclic schedule is NP-hard, and must be done every time the task set changes.

Because cyclic executives are non-preemptible and deterministic, they cannot take advantage of over-provisioned WCET estimates, therefore processor utilisation is low for cyclic executives on modern hardware.

2.2.2 Fixed priority scheduling

As the name implies, fixed priority (FP) scheduling involves assigning fixed priorities to each task. The scheduler is invoked when a job is released or a job ends, and the job with the highest priority is always scheduled. Under real-time fixed-priority scheduling, priorities must be assigned such that all deadlines are met. Two well-established priority-assignment techniques are rate monotonic (RM) and deadline monotonic (DM), both of which are

optimal with respect to FP scheduling. RM priority assignment [Liu and Layland, 1973] allocates higher priorities to tasks with higher rates—where *rate* is determined by the period, as shown in Eq. (2.2).

$$\frac{1}{T_i} \tag{2.2}$$

Schedulability analysis for RM priority assignment requires that deadlines are equal to periods. DM priority assignment [Leung and Whitehead, 1982] allocates higher priorities to tasks with shorter deadlines and relaxes this requirement. In both cases, ties are broken arbitrarily. The FP scheduling technique itself is not optimal, as it results in algorithmic capacity loss and may leave up to 30% of the processor idle. Eq. (2.3) shows the schedulability test for FPRM, and Eq. (2.4) shows that the limit as the number of tasks in the task set (n) tends towards infinity. Fig. 2.2 shows an example FPRM schedule.

$$\sum_{i=0}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{2.3}$$

$$\lim_{n\to\infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots \tag{2.4}$$

2.2.3 Earliest Deadline First Scheduling

The EDF algorithm is theoretically optimal for scheduling a single resource, with no algorithmic capacity loss; that is 100% of processing time can be scheduled. This is because EDF uses dynamic priorities rather than fixed priorities. Priorities are assigned by examining the deadlines of each ready job; jobs with more immediate deadlines have higher priorities. Figure 2.3 illustrates how the task set in Table 2.2 is scheduled by EDF, highlighting the places where tasks are scheduled differently from FPRM.

EDF is compatible with fixed-priority scheduling, as EDF can be mapped to priority bands in a fixed-priority system. Whenever an EDF priority is selected, a second-level EDF scheduler dispatches the next task. This "EDF in fixed-priority" approach has been analysed in detail [Harbour and Palencia, 2003] and is deployed in the Ada programming language [Burns and Wellings, 2007], often used to build real-time systems.

2.2.4 Earliest Deadline First versus Fixed Priority Scheduling

EDF is less popular in commercial practice than FP for a number of reasons, some which are misconceptions, such as complexity of implementation and algorithmic capacity loss, and others which represent concrete concerns, such as behaviour on overload.

In terms of misconceptions, EDF is considered more complex to implement and to have higher overhead-related capacity loss. However, both of these points were debunked by Buttazzo [2005]. Although EDF is difficult and inefficient to implement on top of existing, priority-based OSes, both schedulers can be considered equally complex to implement from scratch. FP scheduling has higher overhead-related capacity loss due to an increase in the

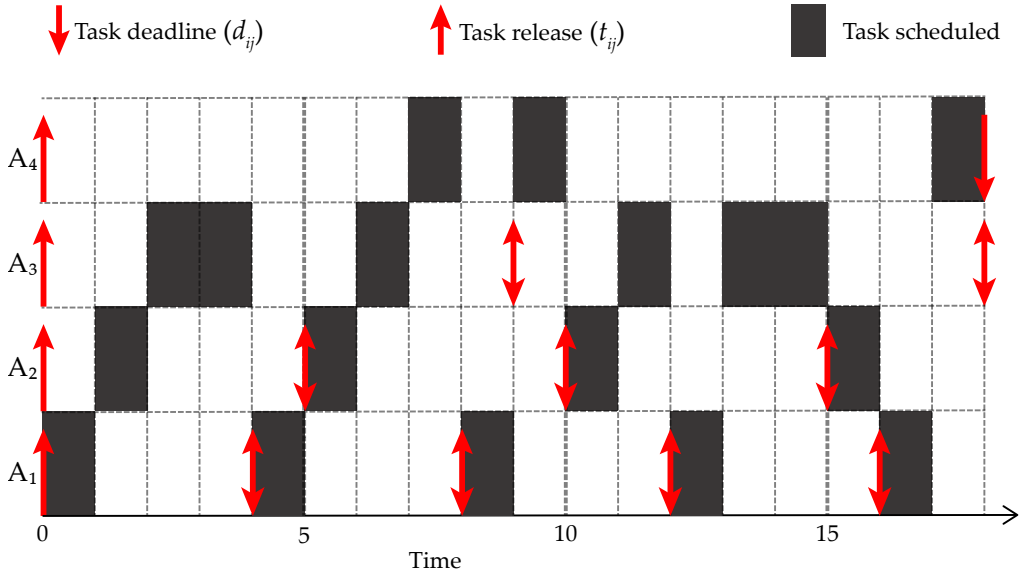


Figure 2.2: An example FPRM schedule using the task set from Table 2.2.

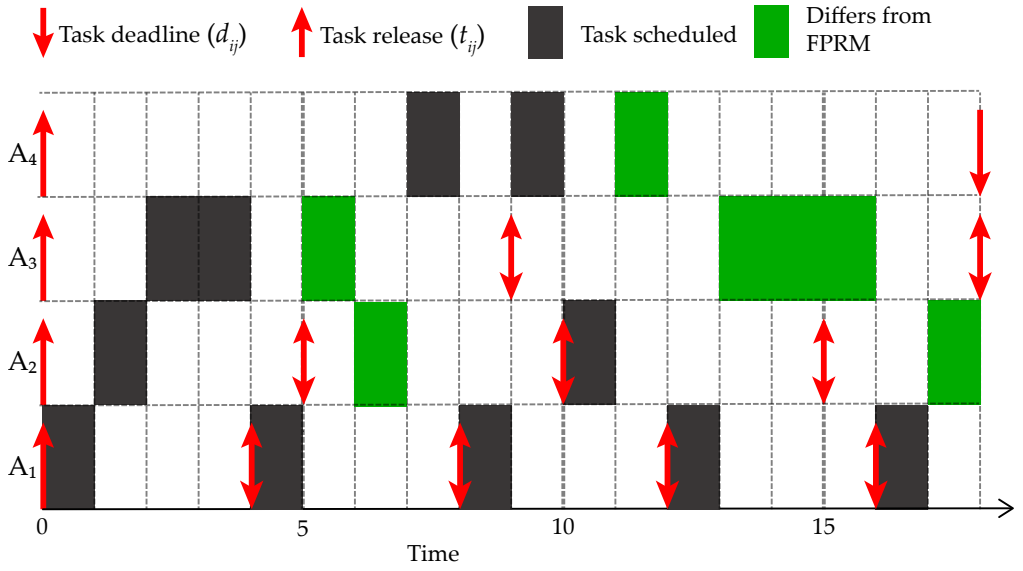


Figure 2.3: An example EDF schedule using the task set from Table 2.2.

amount of preemption. This compounds the algorithmic capacity loss, rendering EDF a clear winner in from-scratch implementations in terms of both properties.

Both algorithms behave differently under constant overload. EDF allows progress for all jobs but at a lower rate, while FP will continue to meet deadlines for jobs with higher RM priorities, completely starving other jobs. Whether these behaviours are desirable is subject to context, under transient overload conditions both algorithms can cause deadline misses. FP overload behaviour is often preferred, as it will drop jobs from lower priority tasks deterministically. In contrast, EDF will drop jobs from all tasks, with no prioritisation.

However, note that the deterministic overload behaviour of FP scheduling comes with a price: optimal priority ordering is determined by the rate of jobs, not their importance to the system.

Other comparisons between EDF and FP are the complexity of their schedulability tests. EDF and FP scheduling both have pseudo-polynomial schedulability tests under the sporadic task model, although EDF under the periodic task model[1] has an $O(n)$ schedulability test. Like all pseudo-polynomial problems, approximations can be made to reduce the complexity, although this comes with an error factor which may not be suitable for HRT systems.

2.2.5 Multiprocessors

Both fixed and dynamic scheduling algorithms scheduling can be used on multiprocessor machines, either *globally* or *partitioned*. Global schedulers share a single scheduling data structure between all processors in the system, whereas partitioned schedulers have a scheduler per processor. Neither is perfect: global approaches suffer from scalability issues such as hardware contention, however partitioned schedulers require load balancing across cores. Partitioning itself is known to be a NP-hard bin-packing problem. On modern hardware, partitioned schedulers outperform global schedulers [Brandenburg, 2011]. For clustered multiprocessors a combination of global and partitioned scheduling can be used; global within a cluster, and partitioned across clusters.

2.3 Resource sharing

In the discussion so far we have assumed all real-time tasks are separate, and do not share resources. Of course, any practical system involves shared resources. In this section we introduce the basics of resource sharing, and the complexities of doing so in a real-time system.

Access to shared resources requires *mutual exclusion*, where only one task is permitted to access a resource at a time, to prevent system corruption. Code that must be accessed in a mutually exclusive fashion is called a *critical section*. Generally speaking, tasks lock access to resources, preventing other tasks from accessing that resource until it is unlocked. However, many variants on locking protocols exist, including locks that permit n tasks to access a section, or locks that behave differently for read and write access.

Resource sharing in a real-time context is more complicated than standard resource sharing and synchronisation, due to the problem of *priority inversion*, which threatens the temporal correctness of a system. Priority inversion occurs when a low priority task prevents a high priority task from running. Consider the following example: if a low priority task locks a resource that a high priority task requires, then the low priority task can cause the high priority task to miss its deadline. Consequently, all synchronised resource access

[1] The periodic task model is the same as the sporadic task model, with the restriction that deadlines must be equal to periods ($d = p$), while periods themselves are considered absolute, not minimum.

in a real-time system must be bounded, and the deadlines of tasks must account for those bounds.

Bounded critical sections alone are not sufficient to guarantee correctness in a real-time system. Consider the scenario outlined earlier, where a low priority thread holds a lock that a high priority thread is blocked on. If other, medium-priority tasks exist in the system, then the low priority task will never run and unlock the lock, leaving the high priority task blocked for an unbounded period. This exact scenario caused the Mars Pathfinder to fault, causing unexpected system resets [Jones, 1997].

In this section we provide a brief overview of real-time synchronisation protocols that avoid unbounded priority inversion, drawn from Sha et al. [1990]. First we consider uniprocessor protocols before canvassing multiprocessor resource sharing.

2.3.1 Non-preemptive critical sections

Using the non-preemptive critical sections protocol (NCP), preemption is totally disabled whilst in a critical section. This approach blocks all threads in the system while any client accesses a critical section. Consequently, the bound on any single priority inversion is the length of the longest critical section in the system. Although functional, this approach results in a lot of unnecessary blocking of higher priority threads. The maximum bound on priority inversion that a task can experience is the sum of the length of all critical sections accessed by that task, as these are the only places that specific task can be blocked while other tasks run.

2.3.2 Priority Inheritance Protocol

In the priority inheritance protocol (PIP), when a high priority task encounters a locked resource, it donates its priority to the task holding the lock and when the lock is released the priority is restored. This approach avoids blocking any higher priority threads that do not access this resource, and works for both fixed and dynamic priority scheduling. However, PIP results in a large preemption overhead and as a result has poor WCET analysis.

To understand this, consider a task set with n tasks, $A_1, \ldots, A_n$, where each task's priority corresponds to its index, such that the priority of $A_i = i$. The highest priority is n and the lowest is 1 and all tasks access the same resource. If A_1 holds the lock to that resource, then the worst preemption overhead occurs if A_2 wakes up, elevating the priority of A_1 to 2. Subsequently, each task wakes up in increasing priority order, each preempting A_1 until its priority reaches n resulting in n total preemptions.

Another disadvantage of PIP is that deadlock can occur if resource ordering is not used.

2.3.3 Immediate Priority Ceiling Protocol

Under immediate priority ceiling protocol (IPCP), also known as the highest lockers' protocol, resources are assigned a *ceiling* priority: the highest priority of all tasks that

access that resource + 1. When tasks lock that resource, they run at the ceiling priority, removing the preemption overhead of PIP.

The disadvantage of IPCP is that all priorities of task that access locked resources must be known *a priori*. Additionally, if priority ceilings are all set to the highest priority, then behaviour degrades to that of NCP. Finally, this protocol allows intermediate priority tasks that do not need the resource to be blocked unnecessarily.

2.3.4 Original Priority Ceiling Protocol

The original priority ceiling protocol (OPCP) combines the previous two approaches, and avoids deadlock, excessive blocking and excessive preemption. In addition to considering the priorities of tasks, OPCP introduces a dynamic global state referred to as the *system ceiling*, which is the highest priority ceiling of any currently locked resource. When a task locks a resource, its priority is not changed until another task attempts to acquire that resource, at which point the resource holder's priority is boosted to the resource's ceiling. By delaying the priority boost the excessive preemption of PIP is avoided. Additionally, tasks can only lock resources when their priority is higher than the system ceiling, otherwise they block until this condition is true, thus avoiding the risk of deadlock. OPCP results in less blocking overall than IPCP, however requires global state to be tracked across all tasks, even those that do not share resources, increasing the complexity of an implementation.

2.3.5 Stack Resource Protocol

Ceiling-based protocols are only appropriate for FP schedulers, however the stack resource policy (SRP) [Baker, 1991] is provides similar functionality for EDF. Under the SRP, all tasks are assigned *preemption levels* and all resources are assigned ceilings, which are derived from the maximum preemption-level of tasks accessing those resources. Similar to OPCP, a system ceiling is also maintained, and is the maximum active preemption level of all tasks currently executing in the system. SRP works by preventing preemption: a task is only allowed to preempt the system when two conditions are met: that tasks absolute deadline must be less than the currently executing task, and its preemption level must be higher than the current system ceiling.

2.3.6 Lock free algorithms

Lock-free data structures can be used to avoid locks and priority inversion altogether, on a uniprocessor this is achieved with atomic compare-and-swap, which is either supported by hardware or provided by disabling preemption. Lock-free data structures allow concurrent access to the same memory, however interleaved access can result in a given operation being unbounded due to the failure of atomic retries. Although this seems incompatible with real-time systems, Anderson et al. [1997] show that lock free approaches can be bounded and occur less overhead than wait-free or lock-based schemes.

However, given our approach uses formal verification, we do not consider lock-free options further, as interleaving of program execution results in a state-space explosion and remains a challenge for verification on a large scale.

2.3.7 Summary

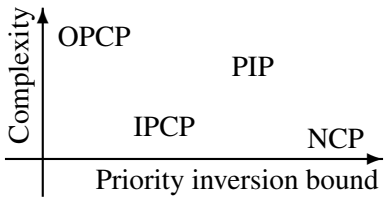


Figure 2.4: Comparison of real-time locking protocols based on implementation complexity and priority inversion bound.

Figure 2.4 compares the different uniprocessor locking protocols, showing that OPCP provides the lowest bound on priority inversion; however is also the most complicated to implement. NCP on the other hand, is the simplest to implement but exhibits the worst priority inversion behaviour, with PIP and IPCP falling between the two. IPCP provides minimal implementation complexity but requires a policy on priority assignment to be in place in the system.

2.3.8 Multiprocessor locking protocols

Resource sharing on multiprocessors is far more complicated than the single processor case and still a topic of active research [Davis and Burns, 2011]. Of course, uniprocessor techniques can be used for resources that are local to a processor, but further protocols are required for resources shared across cores (termed *global* resources).

Protocols for multiprocessor locking are either spinning- or suspension-based; *spinning* protocols spin on shared memory; *suspension* protocols block the task until the resource is available, such that other tasks can use the processor during that time. Spin-lock protocols are effective for short critical sections, but once the critical section exceeds the time taken to switch to another task and back, semaphore protocols are more efficient. Brandenburg et al. [2008] find that for small, simple objects, non-blocking algorithms are preferably, and that wait-free or spin-based approaches are better for large or complex objects, in a real-time context with few cores. However, for systems with many-cores, spin-lock approaches do not scale beyond a small number (~10) cores [Clements et al., 2013].

Multiprocessor locking protocols differ depending on the scheduling policy across cores; in partitioned approaches, priorities on different cores are not comparable, meaning existing protocols do not work. While the protocols we have examined so far can be used under global scheduling, Brandenburg [2011] showed that partitioned approaches suffer far less cache overheads than global scheduling.

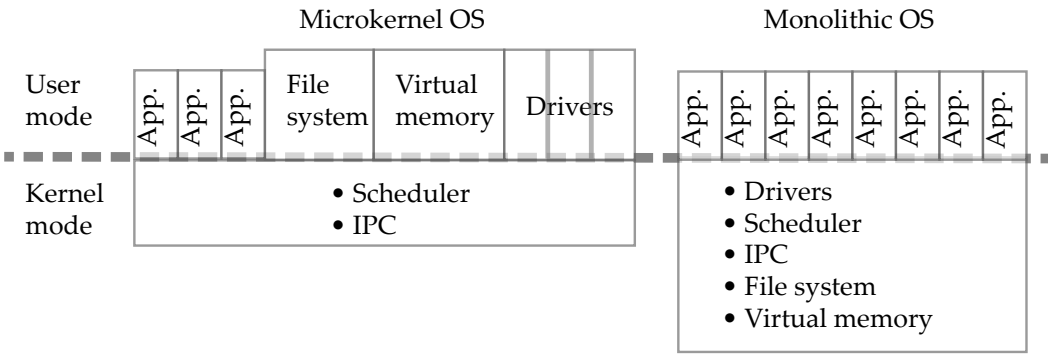


Figure 2.5: Structure of a microkernel versus monolithic operating system.

The multiprocessor priority ceiling protocol (MPCP) [Rajkumar, 1990] is a modified version of OPCP for multiprocessors. It is a suspension-based protocol that works by boosting task priorities. Tasks run at the highest priority of any task that may access a global resource, and waiting tasks block in a priority queue. Nested access to global resources is disallowed. The multiprocessor stack resource policy (MSRP) [Gai et al., 2003] is a spin-lock based protocol, which can be used for FP and EDF scheduling. MSRP uses the SRP locally, combined with first-in first-out (FIFO) spin-locks which guard global resources.

Multiprocessor real-time locking protocols are an extensive field of research, and many more sophisticated locking protocols exist, however we do not survey them here.

2.4 Operating systems

An OS is a software system that interfaces with hardware and devices in order to present a common interface to applications. The *kernel* is the part of the operating system that operates with privileged access to the processor(s) in order to safely perform tasks that allow applications to run independently of each other.

Common OSes, such as Windows, MacOS and Linux, are *monolithic* operating systems, which means that many services required to run applications are inside the kernel. A *microkernel* attempts to minimise the amount of code running in the kernel in order to reduce the amount of trusted code. Figure 2.5 illustrates the difference between monolithic OSes and microkernels. Modern microkernel implementation is guided by the minimality principle [Liedtke, 1995] which aims to provide minimal mechanisms to allow resource servers to function, leaving the rest of the policy up to the software running outside of the kernel. According to the minimality principle, if a service does not need to be in the kernel to achieve its functionality, it should not be in the kernel.

Monolithic operating systems provide scheduling, inter-process communication (IPC), device drivers, memory allocation, file systems and other services in the kernel, resulting in a large *trusted computing base*.

With respect to microkernels, interpretations of the minimality principle varies, consequently which services and utilities are included in the privileged kernel varies. In larger kernels thread scheduling, memory allocation and some device drivers are included in the kernel. For example, seL4 [Klein et al., 2009] contains scheduling and IPC; COMPOSITE [Parmer, 2009] does not provide a scheduler, or any blocking semantics, but does provide IPC.

Microkernels are far more amenable for deployment in areas where security is a primary concern, due to their small trusted computing base. Services on top of the microkernel can be isolated and assigned different levels of trust, unlike the services in a monolithic OS which all run at the same privilege level such that a fault in one service can compromise the entire system.

Operating systems can run on each other in a process called *virtualisation*, where the underlying OS presents an interface imitating hardware. A *hypervisor* is an operating system that can run other operating systems on top of it, and operating systems running on the hypervisor are referred to as *guests*. Guest operating systems can be para- or fully-virtualised, where the former involves modifications to the source of the guest. Modern hardware has virtualisation extensions which improve virtualisation performance and reduce the need for para-virtualisation. Both microkernels and monolithic kernels can also be hypervisors, although monolithic hypervisors are often smaller than full OS counterparts, as they provide less functionality and rely on guest OSes to provide most systems.

2.4.1 IPC

IPC is the microkernel mechanism for synchronous transmission of data and capabilities between processes. Because the microkernel model provides services encapsulated into user-level servers, IPC is key to microkernel performance, as it is used more predominantly than in monolithic OSes. Originally, microkernels were criticised as impractical due to inefficient IPC implementations of first-generation microkernels. However, this was demonstrated to be false [Härtig et al., 1997] due to the high cache footprint and poor design of the original microkernels. Second-generation microkernels were built much leaner, with fewer services in the kernel and fast, optimised code paths for IPC, referred to as *fastpaths*. *Third-generation* microkernels follow the pattern of minimality and speed, whilst also promoting security as a first-class concern, which resulted in the incorporation of capability systems.

2.4.2 Capabilities

Third-generation microkernels make use of *capabilities* [Dennis and Van Horn, 1966], an established mechanism for fine-grained access control to spatial resources which allow

for spatial isolation. A capability is a unique, unforgeable token that gives the possessor permission to access an entity or object in system. Capabilities can have different levels of access rights, e.g. read, write, execute etc.

By combining access rights with object identifiers capabilities avoid the confused deputy problem, a form of privilege escalation where a deputy program acting on behalf of a client is tricked into using its own rights to manipulate a resource that the client would not normally have access to [Hardy, 1988].

2.4.3 Open vs. Closed Systems

Operating systems can be built for open or closed systems. An *open system* is any system where code outside of the control of the system designers can be executed. For example, modern smart phones are open systems, given that users can install third-party applications.

A *closed system* is the opposite; the system designers have complete control over all code that will execute on the system. The majority of closed systems are embedded, including those found in cars, spacecraft and aircraft.

In general, there is a trend toward systems becoming more open; initial mobile phones were closed systems. This trend can be perceived from infotainment units in automobiles to televisions, where the option to install third party applications is becoming more prevalent. Allowing third-party applications to run alongside critical applications on shared hardware increases the security requirements of the system: critical applications must be isolated from third-party applications and secure communications must be used between distributed components. This is currently not the general case, which has led to researchers demonstrating attacks on cars [Checkoway et al., 2011].

Open systems are generally *dynamic*—where resource allocations are configured at run-time and can change—as opposed to closed systems which have *fixed* or *static* resource allocation patterns.

2.4.4 Real-Time Operating Systems

A real-time operating system (RTOS) is an OS that provides temporal guarantees, and can be microkernel-based or monolithic. Whilst some real-time systems run without operating systems at all, this approach is generally limited to small, closed systems and is both inflexible and difficult to maintain [Sha et al., 2004].

In a general purpose OS, time is shared between applications with the aim of providing *fairness*, where applications share the processor equally. This fairness is not divided into equal share, but weighted, such that some applications are awarded more time than others in order to tune overall system performance. The OS itself is not directly aware of the timing needs of applications.

In an RTOS, fairness is replaced by the need to meet deadlines. As a result, time is promoted to a first class resource [Stankovic, 1988].

Time being an integral part of the system affects every other part of the OS. For example, in an RTOS, one application having exclusive access to a resource cannot be allowed to cause a deadline miss. Similarly, the RTOS itself cannot cause a deadline miss. This means that all operations in the RTOS must either be bounded with known WCET or the RTOS must be fully preemptible. However, it must be noted that a fully preemptible OS is completely non-deterministic, making correctness impractical to guarantee [Blackham et al., 2012]. The overheads of RTOS operations like interrupt handling and context switching must also be considered when determining whether deadlines can be met.

Traditional RTOSes, and the applications running on them, require extensive offline analysis to guarantee that all temporal requirements are met. This is done by using scheduling algorithms, WCET analysis, and resource sharing algorithms with known real-time properties.

2.4.5 Resource kernels

Resource kernels are a class of OS that treat time as a first class resource, by providing timely, guaranteed access to system resources. In a resource kernel, a reservation represents a portion of a shared resource, like processor, or disk bandwidth. Unlike traditional real-time operating systems, resource kernels do not trust all applications to stay within their specified resource bounds: resource kernels enforce them, preventing misbehaving applications from interfering with other applications and thus providing temporal isolation.

In the seminal resource kernel paper, Rajkumar et al. [1998] outline four main goals that are integral to resource kernels:

G1: Timeliness of resource usage Applications must be able to specify resource requirements that the kernel will guarantee. Requirements should be dynamic: applications must be able to change them at run-time, however the kernel should ensure that the set of all requirements can be admitted.

G2: Efficient resource utilisation The mechanisms used by the resource kernel utilise available resources efficiently and must not impose high utilisation penalties.

G3: Enforcement and protection The kernel must enforce resource access such that rogue applications cannot interrupt the resource use of other applications.

G4: Access to multiple resource types The kernel must provide access to multiple resource types, including processing cycles, disk bandwidth, network bandwidth and virtual memory.

In another paper, de Niz et al. [2001] outline the four main mechanisms that a resource kernel must provide, in order to implement the above concepts.

Admission check that all resource requests can be scheduled (**G1**).

Scheduling implements the dynamic allocation of resources according to reservations (**G1, G2**).

Enforcement limit the consumption of the resources to that specified by the reservation (**G3**).

Accounting of reservation use, to implement scheduling and enforcement (**G1, G2, G3**).

In order to share resources in a resource kernel, avoiding priority inversion becomes a more complicated problem. de Niz et al. [2001] outline three key policies that must be considered when handling resource sharing in reservation-based systems:

Prioritisation What (relative) priority is used by the task accessing the shared resource (and under what conditions)?

Charging Which reservation(s), if any, gets charged, and when?

Enforcement What happens when the reservations being charged by the charging policy expire?

Resource kernels are a form of monolithic operating system, where all system services and drivers are provided by the kernel. In a microkernel, not all the mechanisms of a resource kernel are suitable for inclusion in the kernel itself: some can be provided by user-level middle-ware. This is because core resource kernel concepts contain both policy and mechanism. We argue that the microkernel should provide resource kernel mechanisms such that a resource kernel can be built with a microkernel, but policy should be left up to the system designer, as long as it does not result in performance or security concessions.

2.5 Summary

In this chapter we have briefly covered the core real-time theory that this thesis draws upon. We have defined operating systems, and introduced the concepts that inform the design of resource kernels. In the next chapter we will survey how these can be combined to achieve isolation and asymmetric protection for mixed-criticality systems.

3 Temporal Isolation & Asymmetric Protection

Mixed-criticality systems at their core require isolation: isolation as strong as that provided by physically isolated systems, meaning if one sub-system fails it cannot affect other sub-systems. Isolation can be divided into two categories of resources: spatial and temporal. *Spatial* resources include devices and memory, where isolation can be achieved using the MMU and I/O MMUs. *Temporal* isolation of resources is more complicated, and forms the focus of this chapter, where we survey the relevant literature. A system is said to provide *temporal isolation* if temporal behaviour of one task cannot cause temporal faults in another, independent task.

What does isolation mean in a fully- or over-committed system, where there is no slack time to schedule? What if there simply is not enough time? One could argue that systems should be over-provisioned to avoid such a scenario. However, in the presence of SRT and best-effort tasks which may be low in criticality, this requirement is too strong. Instead, we must explore mechanisms for *asymmetric protection*, where high criticality tasks can cause a failure in low criticality tasks, but not vice versa.

Much of the background examined in the previous chapter (Section 2.1) made the assumption that tasks would not exceed a declared WCET or critical section bound. Many existing real-time systems run either one application, or multiple applications of the same criticality, meaning each application that is running is certified to the same level. This means that all applications are trusted: trusted to not crash, and trusted to not overrun their deadlines. If one application does overrun its deadline or use more processing time than specified by its WCET, guarantees are no longer met.

Tasks can be untrusted for many reasons including:

- sporadic tasks with inter-arrival times that are event driven will not necessarily have device drivers which guarantee the inter-arrival time;
- the task may have an unknown or unreliable WCET;
- the system may be open and the task from an untrusted source;
- the task may be low criticality and therefore not certified to a high level.

The issue of trust in real-time literature has not been greatly addressed: real-time tasks are often assumed to perform correctly and safely. However, much research has looked into the scheduling of aperiodic tasks, which by definition cannot be trusted to follow a specific schedule, or abide by their estimated minimum inter-arrival time. Further applicable research examines the scheduling of SRT and best-effort tasks along with HRT tasks. Consequently, we examine scheduling methods for these types of systems.

Neither FP nor EDF scheduling approaches discussed so far provide temporal isolation, although both can be adapted to do so. In this chapter we examine the techniques used by the real-time community to achieve temporal isolation.

3.1 Scheduling

3.1.1 Proportional-share schedulers

Proportional share schedulers provide temporal isolation, as long as the system is not overloaded, although this class of schedulers is based on achieving scheduling fairness between tasks, rather than running untrusted tasks which may exceed their execution requirement.

Recall that fairness is not a central property of scheduling in a real-time operating system. However, one approach for real-time scheduling is to specify a set of constraints that attempt to provide fairness and also satisfy temporal constraints. These are referred to as *proportional share* algorithms, which allocate time to tasks in discrete sized quanta. Tasks in proportional share schedulers are assigned weights according to their rate, and those weights determine the share of time for which each task has access to a resource.

While proportional share algorithms are applied to many scheduling problems, they apply well to real-time scheduling on one or more processors. Unlike other approaches to real-time scheduling, proportional share schedulers have the explicit property of guaranteeing a rate of progress for all tasks in the system.

Baruah et al. [1996] introduced the property *proportionate fairness* or *Pfair* as a strong fairness property for proportionate share scheduling algorithms. For a schedule to be Pfair, then at every time t a task T with weight T_w must have been scheduled either $\lceil T_w.t \rceil$ or $\lfloor T_w.t \rfloor$ times. *Early-Release fair* or *ERfair* [Anderson and Srinivasan, 2004] is an extension of the Pfair property that allows tasks to execute before their Pfair window, which can allow for better response times.

Pfair scheduling algorithms break jobs into sub-jobs that match the length of a *quantum*, which is a fixed, discrete length of time defined by the system. Real-time and non-real time tasks are treated similarly. When overload conditions exist, the rate is slowed for all tasks.

Pfair scheduling algorithms are good theoretically but do not perform well in practice; they incur large overhead related capacity loss due to an increased number of context switches [Abeni and Buttazzo, 2004]. Additionally, since scheduling decisions can only be made at quantised intervals, scheduling is less precise in proportionate fair systems.

This problem can be exacerbated by critical sections, which may last longer than a single quantum. Stoica et al. [1996] propose defining arbitrary quanta sizes based on maximum critical section size, however quanta size decreases the accuracy of the scheduler. Additionally, it may not be possible to have *a priori* knowledge of critical section size, especially in a soft real-time system where it is not worth conducting WCET analysis or execution time is dependent on exterior factors, such as network behaviour.

One early uniprocessor Pfair scheduling algorithm is earliest-eligible deadline first, presented in Stoica et al. [1996]. PD^2 [Srinivasan and Anderson, 2006] is a more recent Pfair/ERfair scheduling algorithm that is theoretically optimal for multiprocessors under HRT constraints, although only under the assumption that process preemption and migration are free.

Recall that temporal isolation means that tasks should not be able to interfere with the temporal behaviour of other tasks in the system. Proportionate fair systems provide temporal isolation as part of their fairness property, unless the system is overloaded, at which point the rate of all tasks will degrade. Pfair schedulers by definition do not support asymmetric protection.

3.1.2 Isolation with EDF schedulers

Temporal isolation in EDF scheduling has been explored thoroughly in the real-time discipline. We outline the most dominant approaches in this section.

Robust earliest deadline scheduling

One early approach to temporal isolation with EDF scheduling attempts to extend the algorithm to allow overload conditions to be handled with respect to a value. Robust earliest deadline scheduling [Buttazzo and Stankovic, 1993] assigns a value to each task set, and will drop jobs from low-value tasks under overload. If the system returns to non-overload conditions, those tasks are scheduled again. This is a very early version of asymmetric protection. The algorithm is optimal, however this is only the case if scheduler overhead is excluded. Since the algorithm has $O(n)$ complexity in the number of tasks, the authors recommend using a dedicated scheduling processor such that overhead will not affect the timing behaviour – but this is not suitable for embedded systems, where the goal is to minimise the number of processors, not increase them.

Constant-bandwidth servers

In the real-time community, the term *server* is used to describe virtual time sources, where an intermediate algorithm monitors task execution and, using that information, prevents task(s) guarded by the server from exceeding specified temporal behaviour. These algorithms are integrated into the scheduler.

Constant-bandwidth server (CBS) is a technique for scheduling HRT and SRT tasks and providing temporal isolation [Abeni and Buttazzo, 2004]. Under CBS, HRT tasks are scheduled using an EDF scheduler, but SRT tasks are treated differently as EDF does not handle overload reasonably. Instead, a CBS is assigned to each SRT task. Each CBS has a bandwidth assigned to it, and breaks down SRT jobs into sub-jobs such that the utilisation rate of the task does not exceed the assigned bandwidth. Any sub-job that will cause the bandwidth to be exceeded is postponed, but still executed.

CBS stands out from previous server-based approaches like the total bandwidth server[Spuri and Buttazzo, 1994] as it does not require a WCET or a minimum bound on job inter-arrival time, making it much more suitable for SRT tasks. Implementation wise, CBS has less hardware overheads than Pfair schedulers.

Many extensions exist for CBS to improve functionality. Kato et al. [2011] extend CBS to implement *slack donation*, where any unused bandwidth is given to other jobs. In [Craciunas et al., 2012], CBS is extended such that bandwidths are variable at run-time. Lamastra et al. [2001] introduce bandwidth inheritance across CBS servers applied to different resources, providing temporal isolation for additional resources other than processing time.

Rate-based EDF

Rate-based earliest deadline first (RBED) schedulers explicitly separate the resource allocation and dispatching (choosing which thread to run) in order to provide flexibility in timeliness requirements supported by the scheduler. RBED [Brandt et al., 2003] is an algorithm that implements such a scheduler. In RBED, tasks are considered as either HRT, SRT, best-effort or rate-based. Tasks are modelled using an extension of the periodic task model, allowing any job of a task to have a different period. If rate-based or HRT tasks cannot be scheduled at their desired rate they are rejected. SRT tasks are given their rate if possible with the option to provide a quality of service specification. Processor time reservations can be used to make sure best-effort tasks are allowed some execution time. Otherwise, they are allocated slack time unused by SRT and HRT tasks. Either way, best-effort tasks are scheduled by assigning them a rate that reflects how they would be scheduled in a standard, fair, quantum-based scheduler. Based on the rates used, RBED breaks tasks down and feeds them to an EDF scheduler to manage processing time. Rates are enforced using a one-shot timer to stop tasks that exceed their WCET. As tasks enter and leave the system, the rates of SRT tasks will change. Slack time that occurs as a result of tasks completing before their deadlines is only donated to best-effort tasks, although the authors note that extensions should be able to donate slack to SRT tasks as well. RBED is similar to the concept of CBS, however it deals with separate types of real-time tasks more explicitly.

3.1.3 Isolation with FP schedulers

We will now examine methods for temporal isolation in fixed-priority systems. Like CBS, tasks are constrained by being encapsulating in a server, an algorithm which rations out processing time, preventing specific scheduling parameters from being exceeded. Approaches include polling servers, deferrable servers, sporadic servers, priority exchange servers and slack stealing servers.

Polling servers

Polling servers [Lehoczky et al., 1987] wake every period, to check if there are any pending jobs, then allows jobs to run until their budget is exhausted, at most. If there is no task to run, the polling server will go back to sleep. That is, at time t_i, if there are no tasks ready to execute, the server will sleep until t_{i+1}. This has the limitation that task latency is just under the period T – if an event triggers a job just after the polling server wakes, finds there is no work and returns to sleep, the pending job is delayed until the polling server wakes again.

Deferrable Servers

Unlike polling servers, *deferrable servers* [Lehoczky et al., 1987; Strosnider et al., 1995] preserve any unused budget across periods, although the budget can never be exceeded. This removes latency problems with polling servers, as deferrable servers can wake any time a job is available. Unfortunately, deferrable servers do not provide true temporal isolation and cannot guarantee that tasks will not exceed a maximum bandwidth. This is due to the fact that deferrable servers permit jobs to execute back-to-back, thus exceeding their allocated scheduling bandwidth for any specific occurrence of the period. This occurs as deferrable servers replenish the budget to full at the start of each period, and the budget can be used at any point during a task's execution.

We demonstrate the problem with deferrable servers using the notation introduced in Table 2.1. Consider a sporadic task with implicit deadlines in a task set, A_1, with jobs $A_{11}, A_{12}, \ldots, A_{1n}$. Each job in that task set has a deadline once the period has passed: $d_{1j} = t_{1j} + T_1$. The problem occurs if the first job arrives at $a_{11} = d_{11} - C_1$, such that it only completes at exactly the implicit deadline. Then a second job may arrive at the release time d_{11} such that it runs back-to-back with the first task, from $a_1 1$ to $d_{11} + C_1$, then the task has exceeded its permitted utilisation ($\frac{C_1}{T_1}$). As a result deadline misses can be caused in other tasks, violating temporal isolation.

Sporadic servers

Sporadic servers [Sprunt et al., 1989] address the problems of deferrable servers by scheduling multiple replenishment times, in order to preserve the property that for all possible points in time $U_i \leq \frac{C_i}{T_i}$, known as the *sliding window* constraint, which is the condition that deferrable servers violate. Each time a task is preempted, or blocks, a replenishment is set

for current time + T_i, for the amount consumed. When no replenishments are available, sporadic servers have their priority decreased below any real-time task. The priority is restored once a replenishment is available. While this approach addresses the problems of deferrable servers, the implementation is problematic as the number of times a thread is preempted or blocked is potentially unbounded. It is also subject to capacity loss as tasks that use very small chunks of budget at a time increase the interrupt load. The bigger the bound on replenishments the less accurate the sporadic server, but the more memory used resulting in degraded performance.

Priority exchange servers

Priority exchange servers [Sprunt et al., 1989; Spuri and Buttazzo, 1994] allow inactive, high-priority tasks to swap their priorities with active, low-priority tasks, such that server capacity is not lost but used at a lower priority. Implementations of priority exchange require control and access to priorities across an entire system.

Slack stealing

Slack stealing [Ramos-Thuel and Lehoczky, 1993] is an approach that runs a scheduling task at the lowest priority and tracks the amount of slack per task in the system. As aperiodic tasks arrive, the slack stealer calculates whether they can be scheduled or not based on the slack in the system and current load of periodic tasks. This method does not provide guarantees at all for the aperiodic tasks, unless a certain bound is placed on the execution of periodic tasks.

3.2 Mixed-criticality schedulers

In the real-time community, much work has been conducted around a theoretical mixed-criticality task model introduced by Vestal [2007]. In this model, a system has a specific range of criticality levels, $L_{min} - L_{max}$ and a current criticality level L_{now}. Each task in the system is assigned a criticality level, L_i, and has a vector of execution times of size $L_i - L_{min}$. When the system undergoes a mode change, should $L_i \geq L_{now}$ the execution time required by that task is set to the value in the vector corresponding to the new L_{now}. If a task's criticality is less than L_{now}, it may be dropped or scheduled with much lower priority until a criticality mode-switch increases L_{now}.

Vestal [2007]'s model can be interpreted in several ways; as a form of graceful degradation; or as an optimisation in the case where a system designer and CA disagree on WCET estimates for a system, which results in tasks with two WCET estimates, one very pessimistic one from the CA and a less pessimistic one from the system designers or automated tools. Theoretically, if the system designer could show that the higher estimates of the CA can be met by such a mode change, they could schedule less criticality tasks when in a low criticality mode.

None of the scheduling algorithms so far directly support mixed-criticality systems. RBED is the closest, although it assumes a direct relationship between criticality and real-time model, with the assumption that HRT tasks are more critical than SRT tasks which are more critical than best-effort tasks.

As a result of this, a family of mixed-criticality schedulers exists that handle high criticality tasks with two WCET estimates, and low-criticality tasks. The scheduling algorithm will always schedule high-criticality tasks. If high-criticality tasks finish before the lower WCET estimate, lower criticality tasks are also scheduled. Otherwise, tasks of lower criticality may not be scheduled at all.

3.2.1 Static mixed-criticality

Static mixed-criticality schedulers are built for variants on Vestal [2007]'s model. Scheduling algorithms in this class are distinguished by a *criticality mode-switch* between two or more criticality levels, which may result in low criticality tasks being dropped or deprioritised in some way. Schedulers for this model of mixed-criticality have been developed and extensively studied for FP [Pathan, 2012; Vestal, 2007] and EDF [Baruah et al., 2011a], and further. As this has been a very active topic, we refer the reader to Burns and Davis [2017] for an extensive background.

However, while this model, and many variations upon it have been subject to much research in the real-time community, questions have been raised as to its practicality in industry. Ernst and Di Natale [2016] claim that a CA would be unlikely to accept multiple WCET definitions and state that the focus of mixed-criticality research should be on providing systems for error handling and proof of isolation in order to make mixed-criticality systems practical.

3.2.2 MC^2

MC^2 [Herman et al., 2012] is a mixed-criticality two-level hierarchical scheduler for multiprocessors with low core counts (2–8) which defines levels of task, which are scheduled differently. *Level-A* tasks are high-criticality tasks which are scheduled with a cyclic executive (recall Section 2.2.1), with parameters computed offline. If no level-A tasks are eligible to run, including in slack left in cyclic, level-A frames, *level-B* tasks are considered. Level-B tasks are trusted, HRT tasks scheduled with partitioned EDF, where one EDF scheduler exists per processor. The authors note that the scheduler at level-B could be swapped with FPRM. In the slack left by level-A and level-B tasks, level-C tasks can run. Level-C tasks are globally scheduled with EDF, using one scheduler queue for all cores. The intuition is that level-C tasks are SRT tasks which run in the slack time of the more critical level-A and level-B tasks. Level-D tasks are considered to be best-effort tasks, and are scheduled in the remaining slack from A,B and C. MC^2 also has optional *budget*

enforcement that can be applied to all levels of task, which prevents tasks from exceeding a specified bandwidth.

The MC^2 scheduler is an example of a combination of policies, where criticality and real-time model are aligned, in a way that may not be appropriate for all systems.

3.2.3 Zero Slack Scheduling

De Niz et al. [2009] propose a scheduling approach that can handle multiple levels of criticality, called zero slack (ZS) scheduling. ZS scheduling is based on the fact that tasks rarely use their WCET. This means that resource reservation techniques like CBS without slack donation result in low effective utilisation. ZS scheduling takes the reverse approach: high criticality tasks steal utilisation from lower criticality tasks. This involves calculating a ZS instant —the last point at which a task can be scheduled without missing its deadline. Under overload, the ZS scheduler makes sure that high criticality tasks are scheduled by their ZS instant, such that they cannot be preempted by lower criticality tasks.

Implementations of ZS scheduling can be built using any priority-based scheduling technique, however in the initial work, FP with RM priority assignment is used. The ZSRM scheduler is proved to be able to schedule anything that standard RM scheduling can, whilst maintaining the asymmetric protection property. ZS scheduling can be combined with temporal isolation via bandwidth servers.

ZS scheduling has been adapted to use a quality of service (QoS) based resource allocation model [de Niz et al., 2012], in the context of AAVs. Many models of real-time systems assume that WCETs for real-time tasks are stable and can be calculated. However, AAVs have complicated visual object tracking algorithms where WCET is difficult to calculate, and execution time varies with the number of objects to track. In practice, De Niz et al. [2012] found that ZSRM scheduling resulted in *utility inversion* — where lower utility tasks prevent the execution of higher utility tasks. Although assuring no criticality inversion occurred with a criticality-based approach, under overload, some tasks offer more utility than others with increased execution time. As a result, the authors replace criticality in the algorithm with a utility function. Two execution time estimates are used for real-time tasks — nominal-case execution time (NCET) and overloaded-case execution time (OCET), each having their own utility. The goal of the scheduler is to maximise utility, under normal operation and overload. In practice, utility and criticality are system specific values that allow for further prioritisation than simply the rate of the task, required by FPRM.

3.3 Resource sharing

One of our goals is to allow tasks of different criticality to share resources. While the resource itself must be at the highest criticality of any task using it, this relationship is not necessarily symmetric; low criticality systems should be able to use high criticality resources, as long as their resource access is bounded. In this section we explore how

resource reservations and real-time locking protocols interact, and assess their suitability for mixed criticality systems.

So far in this chapter we have looked at real-time theory for providing temporal isolation: usually in the form of isolation algorithms (termed servers in real-time theory) which ration out execution time, guaranteeing a maximum bandwidth or allowing aperiodic tasks to run in the slack time of other tasks. Similarly, the resource sharing locking protocols of priority inheritance and ceiling priorities are not designed to work if tasks misbehave: in all the protocols, if a task does not voluntarily release a resource, all other tasks sharing that resource will be blocked. One cannot blindly apply a technique like polling servers directly to resource sharing, as if the bandwidth expires while the resource is held no other task can access that resource.

Resource kernels, as introduced in Section 2.4.5, outline the policy decisions that must be made when combining locking protocols and reservations: prioritisation, charging and enforcement.

Prioritisation, or what priority a task uses while accessing a resource, can be decided by any of the existing protocols: OPCP, IPCP, PIP or SRP. Charging is more complex. Notionally, when a thread is temporally contained by an isolation algorithm, that algorithm corresponds to a reservation. Under normal execution, the reservation is charged when that thread executes. However, if a several threads are vying for a resource, which thread's reservation should be charged for the time executed accessing that resource>

de Niz et al. [2001] describe the possible mappings between reservations and resources consuming those reservations, which comes down to the following choices:

Bandwidth inheritance Tasks using the resource run on their own reservation. If that reservation expires and there are other pending tasks, the task runs on the reservations of the pending tasks.

Reservation for the resource Shared resources have their own reservation, which tasks use. This reservation must be enough for all tasks to complete their request. Once again, if tasks are untrusted no temporal isolation is provided.

Multi-reserve resources Shared resources have multiple reservations, and the resource actively switches between them depending on which task it is servicing.

Finally, the most relevant to mixed-criticality systems, is enforcement: if the reservation being charged for a resource access is fully consumed, what should happen? Without an enforcement mechanism, all tasks waiting to access that resource are blocked until more execution bandwidth is rationed by the isolation algorithm. In the majority of cases, the bandwidth inheritance is used, relying on the idea that any operations involving shared resources in real-time systems are bounded. Then, depending on the prioritisation protocol, tasks must have enough budget to account for that bound. This imposes a major restriction

on shared resources: even in the case where soft-real time access is required of those resources, bounds must be known.

3.3.1 MC-IPC

Brandenburg [2014] implements a multiprocessor IPC-based protocol referred to as MC-IPC, where shared resources are placed in resource servers accessed. In this scheme, the resources themselves must be at the ceiling criticality of any task accessing those resources, but all tasks do not have to be at that criticality level. The protocol works by channelling all IPC requests through a three-level, multi-ended queue where high-criticality tasks are prioritised over best-effort tasks. The protocol relies on bounded resource access if a task exhausts its allocation while using the resource, combined with correct queue ordering such that high-criticality tasks are not blocked by low-criticality ones.

3.4 Summary

In traditional scheduling algorithms (EDF and FPRM), temporal isolation only exists as a convention: tasks are expected to specify their parameters *a priori*, and trusted to not exceed those parameters, an approach which is not suitable for mixed-criticality systems, where tasks of different levels of assurance need to share resources.

In this chapter we have covered theoretical techniques for temporal isolation, mostly derived from theoretical approaches to contain aperiodic tasks, which are unpredictable workloads. One approach is to specify a processor share, or bandwidth, and use an algorithm such as CBS or sporadic server (SS) to temporally contain a task. The majority of these algorithms can be used to implement processor reservations.

We also examined several mixed-criticality schedulers, however all incorporated a great deal of policy: schedulers that provide a criticality switch assume multiple WCET estimates, and that low-criticality tasks can be de-prioritised or dropped. MC^2 defines four levels of scheduling and assumes that the highest criticality task are also hard real time, and that best-effort tasks are less critical than SRT tasks, while ZS-scheduling is optimised for a particular measure of utility.

The mixed-criticality schedulers studied all assume that the criticality of a task is directly related to the real-time strictness: for example, MC^2 prioritises HRT over SRT. While in general critical tasks are HRT, it is possible to have critical tasks that are SRT, for instance, object tracking algorithms whose WCET depends on factors external to the software system. Therefore, although the scheduling policies discussed can solve a specific problem, they are not appropriate for all systems, and should not be mandated.

Finally, we looked at resource sharing, and how this interacts with processor reservations, We saw that while there are existing policies for prioritisation and charging, the majority of enforcement mechanisms are based on bandwidth inheritance.

In the next chapter, we survey existing operating systems and systems techniques with respect to temporal isolation capability, resource sharing, and asymmetric protection.

4 Operating Systems

In this chapter we provide a survey of existing operating systems and mechanisms for building mixed-criticality systems. We evaluate the scheduling and resource sharing policies and mechanisms available, with a focus on temporal isolation, asymmetric protection, policy freedom, and resource sharing.

First we present a number of industry standards, before examining Linux and other open-source operating systems, in addition to commercial offerings, in order to establish current industry standards for temporal isolation. Finally, we survey existing, relevant operating systems from systems research, deeply examining techniques that can be leveraged to build mixed-criticality systems, including isolation and resource sharing techniques.

4.1 Standards

In order to establish standard industry practices, we first present three standards industry standards which provide real-time scheduling and resource sharing (POSIX, ARINC 653, and AUTOSAR) and examine their mechanisms for temporal isolation and resource sharing.

4.1.1 POSIX

First, we look at the *portable operating system interface (POSIX)* standard which underlies many commercial and open-source operating systems. POSIX is a family of standards, which includes specifications of RTOS interfaces [Harbour, 1993] for scheduling and resource sharing, which influence much OS design. Scheduling policies specified by POSIX are shown in Table 4.1.

Faggioli [2008] provides an implementation of `SCHED_SPORADIC`, which Stanovic et al. [2010] used to show that the POSIX definition of the sporadic server is incorrect and can allow tasks to exceed their utilisation bound. The authors provide a modified algorithm for merging and abandoning replenishments which fixes these problems, of which corrections to the pseudo code were published by Danish et al. [2011]. In further work Stanovic et al. [2011] show that while sporadic servers provide better response times than polling servers under average load, under high load the overhead of preemptions due to fine-

Policy	*Description*
SCHED_FIFO	Real-time tasks can run at a minimum of 32 fixed-priorities until they are preempted or yield.
SCHED_RR	As per SCHED_FIFO but with an added timeslice. If the timeslice for a thread expires, it is added to the tail of the scheduling queue for its priority.
SCHED_SPORADIC	Specifies sporadic servers as described in Section 3.1.3 and can be used for temporal isolation. For practical requirements, the POSIX specification of SCHED_SPORADIC specifies a maximum number of replenishments which is implementation defined.

Table 4.1: POSIX real-time scheduling policies.

Policy	*Description*
NO_PRIO_INHERIT	Standard mutexes that do not protect against priority inversion.
PRIO_INHERIT	Mutexes with PIP to prevent priority inversion, recall Section 2.3.2.
PRIO_PROTECT	Mutexes with highest lockers protocol (HLP) to prevent priority inversion, recall Section 2.3.3.

Table 4.2: POSIX real-time mutex policies for resource sharing.

grained replenishments causes worse response times when compared to polling servers. Consequently, they evaluate an approach where servers alternate between sporadic and polling servers depending on load, where the transition involves reducing the maximum number of replenishments to one and merging available refills.

Resource sharing in the POSIX OS interface is permitted through mutexes, which can be used to build higher synchronisation protocols. Table 4.2 shows the specified protocols.

Although POSIX provides SCHED_SPORADIC which can be used for temporal isolation (however flawed), the intention of the policy is to contain aperiodic tasks. However, temporal isolation of shared resources is not possible with POSIX. This is because SCHED_SPORADIC allows threads to run at a lower priority if they have exhausted their sporadic allocation, meaning those threads can still access resources even when running at lower priorities. In fact, running at lower priorities ensures that threads contained by sporadic servers can unlock locked resources: however, it also does not bound locking, or provide ways to pre-empt locked resources. As a result, POSIX is insufficient for mixed-criticality systems where tasks of different criticalities share resources. Few OSes implement the full POSIX standard, however many incorporate features of it, including Linux.

4.1.2 ARINC653

ARINC 653 (Avionics Application Standard Software Interface) is a software specification for avionics which allows for the construction of a limited form of mixed-criticality systems. Under ARINC 653, software of different criticality levels can share hardware under strict temporal and spatial partitioning, where CPU, memory and I/O are all partitioned.

Software of different criticality levels are assigned to separate, non-preemptive partitions, which are scheduled according to a fixed-time window, in the fashion of a cyclic executive (recall Section 2.2.1). When a partition switch occurs, all CPU pipeline state and cache state is flushed to avoid data leakage between partitions [VanderLeest, 2010]. Within partitions, a second-level, preemptive, fixed-priority scheduler is used to dispatch threads, with FIFO ordering for equal priorities. At the end of each partition, all CPU pipeline state and cache state is flushed and a partition switch occurs. Temporal isolation under ARINC is therefore completely fixed between partitions, and non-existent within partitions.

The ARINC 653 specification does not consider multiprocessor hardware, although it is possible to schedule specific partitions on fixed processors.

The standard does permit resource sharing between partitions: resources are either exclusive, and accessible to one partition only, or shared between two or more partitions. Synchronisation mechanisms are specified both inter- and intra-partition.

Inter-partition sharing between partitions is provided by sending and receiving ports, configured to be either sampling ports, or queuing ports, depending on the access required [Kinnan and Wlad, 2004]. Importantly, ports have no impact on the scheduling order of partitions, all operations must complete in the duration of a partition's fixed-time window.

Intra-partition synchronisation is via the low-level events and semaphores, or high-level blackboards and buffers. The latter are both uni-directional message passing interfaces, *buffers* provide a statically-sized producer-consumer queue while *blackboards* provide asynchronous multicast behaviour where multiple tasks can read the latest message until it is cleared [Zuepke et al., 2015]. Finally, ARINC 653 specifies a health monitoring system, which can detect deadline misses and run preconfigured exception handlers.

4.1.3 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a set of specifications for developing real-time software in the automotive domain, which like ARINC653, has a focus on safety. Unlike ARINC, AUTOSAR does not specifically provide for mixed-criticality.

Software in AUTOSAR is either trusted or untrusted, where trusted indicates it can be run in privileged mode. This is effectively an all-or-nothing mechanism for spatial isolation, derived from the fact that much of the hardware AUTOSAR runs on is embedded without an MMU, which would allow for more fine-grained spatial isolation. In terms of temporal isolation, AUTOSAR provides a mechanism referred to as *timing protection*,

where tasks' execution times, resource access durations and inter-arrival times are specified and monitored, such that exceptions can be raised on temporal violations [Zuepke et al., 2015].

Scheduling in AUTOSAR is partitioned per processor, with a fixed-priority, preemptive scheduler per core, with FIFO for equal priorities. Unlike ARINC, AUTOSAR allows for inter-processor synchronisation, facilitated by spinlocks.

For resource sharing, AUTOSAR specifies synchronisation through IPCP, supported by allowing tasks to raise their priority to the resources' configured priority. Additionally, AUTOSAR supports nesting of resources. For synchronisation, events are provided, which threads can block on.

4.1.4 Summary

We survey three representative software standards for developing real-time systems: POSIX, ARINC and AUTOSAR. All mandate the use of fixed-priority schedulers and provide protocols and mechanisms for resource sharing. Under POSIX, the focus of the resource sharing mechanisms is synchronisation of critical sections for correctness, not temporal isolation. AUTOSAR provides optional monitoring which can raise an exception if a task holds a resource for too long, although this requires exact scheduling details to be provided about the task and resource. ARINC alone provides real temporal isolation for tasks sharing resources however, the support is limited to fixed, static partitions with no flexibility.

4.2 Existing operating systems

There is a significant gap between real-time theory, as surveyed in the last chapter, and real-time practice, which we have partially highlighted in the previous section. Now we survey existing open-source and commercial operating systems, which are used in practice, to demonstrate the status quo, and the impact of the discussed specifications and standards on their development.

4.2.1 Open source

Many open-source OSes are used in real-time settings, although generally not in high-assurance, safety-critical systems. Regardless, temporal isolation and resource sharing mechanisms remain important to guarantee the function of the software.

Linux

Due to its collaborative development and a massive code base, Linux is not amenable to certification and cannot be considered an OS for high-criticality applications. However, Linux is frequently used for low-criticality applications with SRT demands, and can be used to provide low-criticality services in a mixed-criticality setting, as long as it is sufficiently

isolated. Additionally, Linux is often used as a platform for conducting real-time systems research.

Linux has a fixed-priority preemptive scheduler which is split into scheduling classes. Real-time threads can be scheduled with POSIX `SCHED_FIFO` and `SCHED_SPORADIC`. Best-effort threads are scheduled with the time-sharing completely-fair scheduler (CFS), and real-time threads are scheduled either FIFO or round-robin, and are prioritised over the best-effort tasks. Fixed priority threads in Linux are completely trusted: apart from a bound on total execution time for real-time threads which guarantees that best-effort threads are scheduled (referred to as real-time throttling [Corbet, 2008]), individual temporal isolation is not possible.

Linux version 3.14 saw the introduction of an EDF scheduling class [Corbet, 2009], which is between the fair and the fixed priority scheduling classes. The EDF implementation allows threads to be temporally isolated using CBS.

Scheduling in Linux promotes the false correlation we see in many systems: real-time tasks are automatically trusted (unless scheduled with EDF and CBS) and assumed to be more important—or more critical—than best-effort tasks. In reality criticality and real-time strictness are orthogonal. Linux does not provide any mechanisms for asymmetric protection beyond priority.

On the resource sharing side Linux provides real-time locking via the POSIX API as per Table 4.2, which is unsuitable for mixed-criticality shared resources.

Numerous projects attempt to retrofit more extensive real-time features onto Linux. We briefly summarise major and relevant works here.

One of the original works [Yodaiken and Barabanov, 1997] runs Linux as a fully-preemptable task via virtualisation and kernel modifications, and runs real-time threads in privileged mode. Interrupts are virtualised and sent to real-time threads, and only directed to Linux if required. Consequently, real-time tasks do not have to suffer from long interrupt latencies, however it also means that devices drivers need to be rewritten from scratch for real-time. This approach is clearly untenable in a mixed-criticality system, given all real-time threads are trusted.

LITMUS^{RT} [Calandrino et al., 2006] is an extension of Linux that allows for pluggable real-time schedulers to be easily developed for testing multiprocessor schedulers which schedule kernel threads. Real-time schedulers run at a higher priority than best-effort threads, and schedulers can be dynamically switched at run time. LITMUS^{RT} is not intended for practical use, but for developing and benchmarking scheduling and resource sharing algorithms. Implementations of global- and partitioned-, EDF and FP schedulers exist for LITMUS^{RT}, in addition to *PFair* schedulers. A SS implementation exists, as well as various multicore, real-time locking protocols. Both MC^2 (Section 3.2.2) and MC-IPC (Section 3.3.1) are implemented and evaluated using LITMUS^{RT}. However both contain strict policies, not mechanisms, that are only suitable for a monolithic kernel. Additionally, MC-IPC is a very heavy-weight IPC.

Linux/RK [Oikawa and Rajkumar, 1998] is a resource kernel implementation of Linux with scheduling, admission control, and enforcement in the kernel. Every resource, including memory, CPU time and devices, is time-multiplexed using a recurrence period, processing time and deadline. Reservations in Linux/RK can be hard, firm of soft, which alters resource scheduling after exhaustion. Hard reservations are not scheduled again until replenishment, firm are only scheduled if no other undepleted reserve or unreserved resource use is scheduled, while soft allows resource usage to be scheduled at a background priority. Linux/RK is additionally often used to implement and test other schedulers, such as ZS scheduling [de Niz et al., 2009], which was presented in Section 3.2.3.

Whilst Linux implementations are suitable for implementing algorithms, being used as test-beds and even being deployed for non-critical SRT applications, ultimately Linux is not a suitable RTOS for running safety-critical HRT applications. The large amount of source code results in a colossal trusted computing base, where it is impossible to guarantee correctness through formal verification or timeliness through WCET analysis. Major reasons for adapting Linux to real-time are the existing applications and wide array of device and platform support. For mixed-criticality systems these advantages can be leveraged by running Linux as a virtualised, guest OS to run SRT and best-effort applications.

RTEMS

RTEMS is an open-source RTOS that operates with or without memory protection, although in either case it is statically configured. Although it is an open-source project, RTEMS is used widely in industry and research. The main scheduling policy is FPRM, however EDF is also available with temporal isolation an option using CBS. No temporal isolation mechanisms are present for fixed-priority scheduling. RTEMS provides semaphores with PIP or HLP for resource sharing, as well as higher level primitives for these. RTEMS does not provide mechanisms for shared resources, as target threads are trusted to complete critical sections within a determined WCET, and provides no mechanism for isolation through shared resources.

FreeRTOS

FreeRTOS is another open-source RTOS, however it only supports systems with memory protection units (MPUs), not MMUs. The scheduler is preemptive FP and PIP is provided to avoid priority inversion.

4.2.2 Commercial RTOSes

Several widely deployed RTOSes are used commercially, the majority providing support for part or all of POSIX.

QNX Neutrino

QNX [2010] was one of the first commercial microkernels, widely used in the transport industry. QNX is a separation-based, first-generation microkernel that provides FP scheduling and resource sharing with POSIX semantics. QNX satisfies many industry certification standards, although these in practice do not require WCET analysis or formal verification of correctness.

VxWorks

VxWorks [Win, 2008] is a monolithic RTOS deployed most notably in aircraft and spacecraft. It supports FP scheduling with a native POSIX-compliant scheduler, and implements ARINC 653. VxWorks also has a pluggable scheduler framework, allowing developers to implement their own, in-kernel scheduler.

PikeOS

PikeOS [SysGo, 2012] is a second-generation microkernel which implements ARINC 653 and runs RTOSes as paravirtualised guests in different partitions. Partitions are scheduled statically in a cyclic fashion, and each partition has its own scheduling structure supporting 256 priorities. An alternative design has been implemented for PikeOS [Vanga et al., 2017], where reservations are used to schedule low-latency, low-criticality tasks. This is achieved by using "EDF within fixed priorities" [Harbour and Palencia, 2003], which schedules using EDF at specific priority bands, combined with a pluggable interface for using a reservation algorithm (e.g. CBS, SS, deferrable server (DS)) to temporally contain threads. In order to achieve low-latency, these tasks are run in the special partition of PikeOS, known as the system partition, which is scheduled at the same time as the currently active partition and provides essential system services. However, these tasks are intended to run without sharing resources or interfering with high-criticality tasks, which run in their own partitions.

Deos

Deos is another RTOS which provides fixed-priority scheduling, with the addition of slack scheduling, where threads can register to receive slack and are scheduled according to their priority when there is slack in the system. Like PikeOS, Deos also implements ARINC 653 and a defined subset of POSIX.

4.2.3 Summary

There are many other RTOSes used commercially, but the general pattern is POSIX-compliant, FP scheduling and resource sharing. This brief survey shows that FP scheduling is dominant in industry due to its predictable behaviour on overload, specification in the POSIX standard, and compatibility with existing, priority-based, best-effort systems. If

EDF is incorporated, it is provided at priority bands in the fixed-priority system. Without a principled way to treat time as a first-class resource, the reliance on fixed-priority conflates the importance of a task and its scheduling priority, often based on rate, resulting in low utilisation in these systems.

OS	*Scheduler*	*Temporal Isolation*
Linux	FP + EDF	CBS
RTEMS	FP + EDF	CBS
FreeRTOS	FP	✗
QNX	FP	ARINC 653, SS
VxWorks	FP	ARINC 653
PikeOS	FP	ARINC 653
Deos	FP	ARINC 653

Table 4.3: Summary of scheduling and temporal isolation mechanisms in surveyed open-source and commercial OSes.

Table 4.3 summarises the commercial and open-source OSes surveyed. Although temporal isolation is sometimes provided with the possibility of bounded bandwidth via CBS or SS, support for temporal isolation in shared resources is non-existent beyond the strict partitioning of ARINC 653. Although many of these RTOSes are deployed in safety critical systems, their support for mixed-criticality applications is limited to the ARINC653 approach discussed in Section 4.1.2. Clearly, more flexible mechanisms for temporal isolation and resource sharing are required.

4.3 Isolation mechanisms

We now look to systems research and explore mechanisms for temporal isolation and resource sharing in research operating systems, exploring their history and the state of the art. First, we briefly introduce each operating system that is surveyed, before exploring in detail specific mechanisms that can be used to support mixed-criticality systems. We investigate how different OSes address resource kernel concepts required to treat time as a first class resource; scheduling, accounting, enforcement, admission. Additionally, we look at how prioritisation, charging and enforcement are achieved, if at all, to achieve temporal isolation across shared resources.

The majority of kernels surveyed here are microkernels, as introduced in Section 2.4 and are as follows:

- KeyKOS [Bomberger et al., 1992], a persistent, capability based microkernel.
- Real-time Mach [Mercer et al., 1993, 1994], a first-generation microkernel.

- The Dresden Real-time OPerating system (DROPS) [Härtig et al., 1998], a second generation, L4 microkernel.
- EROS [Shapiro et al., 1999], the first third-generation microkernel.
- Fiasco [Hohmuth, 2002] is a second-generation L4 microkernel, with fixed-priority scheduling.
- MINIX 3 [Herder et al., 2006], a traditional microkernel with a focus on reliability rather than performance.
- Tiptoe [Craciunas et al., 2009] is a now-defunct research microkernel that also aims at temporal isolation between user-level processes and the operating system.
- NOVA [Steinberg and Kauer, 2010] a third-generation microkernel and hypervisor.
- COMPOSITE [Parmer, 2009] is a component-based OS microkernel, with a dominant focus on support for fine-grained components, and massive scalability.
- Barrelfish [Peter et al., 2010] is a capability-based multi-kernel OS, where a separate kernel runs on each processing core and kernels themselves share no memory and are essentially *central processing unit (CPU)-drivers*.
- Fiasco.OC [Lackorzyński et al., 2012], a third-generation iteration of Fiasco, both microkernel and hypervisor.
- Quest-V [Danish et al., 2011] is a separation kernel / hypervisor.
- seL4 [Klein et al., 2014], a third-generation microkernel with hypervisor support and a proof of functional correctness. We present seL4 and its mechanisms in more detail in Chapter 5.
- AUTOBEST [Zuepke et al., 2015] is a separation kernel where the authors demonstrate implementations of AUTOSAR and ARINC653 in separate partitions.

We do not consider Nemesis [Leslie et al., 1996], as although it treated time as a first-class resource, the focus was on multimedia performance. Nemesis was as single-address-space operating system, which rules out any possibility of isolation.

4.3.1 Scheduling

As in commercial and open-source OSes, fixed-priority scheduling dominates in research, with only Tiptoe and Barrelfish providing EDF schedulers, although MINIX 3 has been adapted for real-time [Mancina et al., 2009], by allowing the kernel's best-effort scheduling to be set to EDF on a per-process basis, with kernel scheduler activations that allow for a user-level CBS implementation.

COMPOSITE stands out, as it does not provide a scheduler or blocking semantics in the kernel at all, requiring user-level components to make scheduling decisions. HIRES [Parmer and West, 2011] is a hierarchical scheduling framework built on top of COMPOSITE.

HIRES delivers timer interrupts to a root, user-level scheduling component, which are then forwarded through the hierarchy to child schedulers. Consequently, scheduling overhead increases as the hierarchy deepens. Child schedulers with adequate permissions use a dedicated system call to tell the kernel to switch threads, while the kernel itself does not provide blocking semantics, which are also provided by user-level schedulers. This design offers total scheduling policy freedom, as user-level scheduling components can implement all the goals of a resource kernel according to their own policy.

4.3.2 Timeslices and meters

Meters were one of the first mechanisms used to treat time as a resource, originating in KeyKOS. A *meter* represented a defined length of processing time, and threads required meters to execute. Once a meter was depleted a higher authority was required to replenish it, requiring another thread to run.

L4 kernels [Elphinstone and Heiser, 2013] extended this concept with timeslices, which like meters are consumed as threads execute, but no authority is invoked to replenish the meter: the timeslice simply represents an amount of time that a thread could execute at a priority before preemption. On preemption, the thread is placed at the end of the FIFO scheduling queue for the appropriate priority, with a refilled timeslice, in a round-robin fashion.

Meters and timeslices provide a unit of time, but do not restrict *when* that time must be consumed by. Although simpler, timeslices have insufficient policy freedom in that they are recharged immediately, providing no mechanism for an upper bound on execution. Although meters allowed a user-level policy to define replenishment, this proved expensive on the systems and hardware at the time, as every preemption resulted in a switch to a user-level scheduler and back. Neither concept provides anything resembling a bandwidth or share of a CPU, and only threads at the highest priority level have any semblance of temporal isolation as they cannot be preempted by other threads, sharing their time only with threads at the same priority.

As a result, in a system where time is treated as a first-class resource, the timeslice/meter is not an appropriate, policy-free mechanism alone for building mixed-criticality systems with temporal isolation guarantees.

4.3.3 Reservations

Many research OSes have provided mechanisms for processor reservations, a concept with roots in resource kernels, as introduced in Section 2.4.5. Like meters and timeslices, reservations are consumed when threads execute upon them, however, they represent a bandwidth or share of a processor, and provide a mechanism for temporal isolation.

Reservation schemes differ in the algorithm used to ensure the specified bandwidth was not exceeded: Mach and EROS used DS, Tiptoe CBS, while Barrelfish provides RBED.

SSs are provided by Quest-V [Li et al., 2014], which address the back-to-back problem of DS however require a more complex implementation. Quest-V provides reservations through SS, however I/O and normal processes are distinguished statically: I/O processes use polling servers and normal processes use sporadic servers.

All the research OSes that implement reservations have an admission test in the kernel which is run on new reservations, and checks that the set of reservations are schedulable. Although admission testing is dynamic, providing the admission test in the kernel makes the scheduler used, and the schedulability test, an inflexible policy mandated by the kernel.

Enforcement policy, which determines what occurs when a reserve is exhausted, varies between two extremes: either threads cannot be scheduled until the reservation is replenished, or it is scheduled in slack time. CBS approaches enforce an upper bound on execution, not permitting threads to run until replenishment. Real-time Mach, with EROS to follow, allowed threads with exhausted reservations to run in a second-level, time-sharing scheduler. For RBED-based approaches, enforcement is coded into the classification of the task: rate-based tasks are not scheduled until replenishment, but best-effort and SRT tasks can run in slack time. DROPS allowed processes to reserve a higher priority for a certain amount of cycles, before returning to a lower priority, essentially running expired reservations in slack time.

The importance of reservations also varies. In the case of RBED, the highest criticality threads are trusted, and their execution is not monitored at all, while threads with reservations are considered second tier and best-effort threads the least important. This allows for a strict form of asymmetric protection where HRT threads can temporally affect SRT threads and best-effort threads, but not vice-versa. The Mach approach, on the other hand, only guarantees time to threads with reservations, which are scheduled ahead of any threads in the time-sharing scheduler. We consider both models to be policy, as is the original resource kernel concept where all threads must have a reservation in order to execute at all.

Some implementations allow for multiple threads to run on one reservation, which effectively creates a hierarchical scheduler, but allows for convenient temporal isolation of a set of tasks. In Quest-V, reservations are actually bound to virtual processors, not threads scheduled by that processor.

Reservations are a mechanism that can be used to provide isolation, however, all the kernels surveyed combine this mechanism with a good deal of policy for enforcement, how important reservations are, and admission. To build mixed-criticality systems, we need high-performance mechanisms for temporal isolation that are decoupled from policy. To achieve this, systems must allow for *overcommitting* of processing resources: schedulability is a system policy, not a mechanism.

4.3.4 Timeslice donation

Reservations alone can be used to provide temporal isolation, however are insufficient without further mechanisms for resource sharing. Recall from Section 2.4.5 that prioritisa-

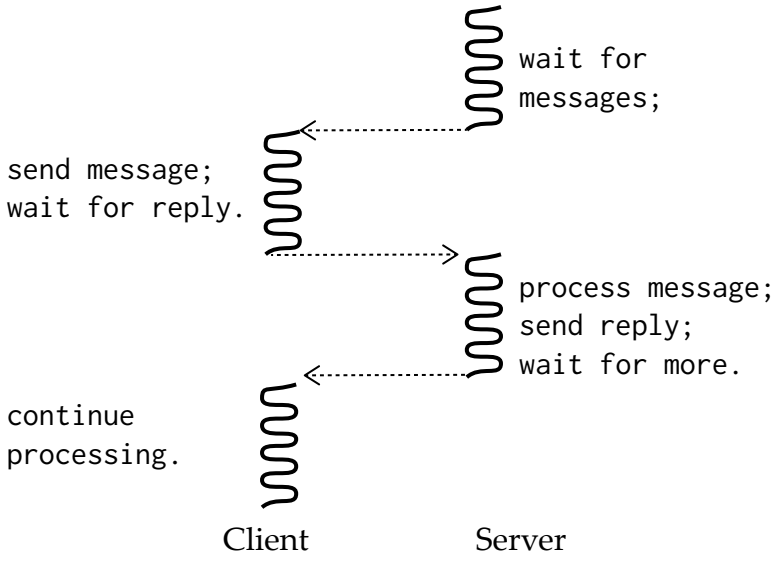


Figure 4.1: Thread execution during IPC between client and server.

tion (the priority threads use when accessing resources), charging (which reservation to charge during resource access) and enforcement (what action to take when a reservation expires while accessing a resource) are key to avoiding priority inversion in a resource kernel.

We first examine mechanisms for charging in the context of microkernels. Recall from Section 2.4.1 that in a microkernel, OS services are implemented at user-level where clients use IPC to communicate with servers, using remote procedure calls (RPCs), as illustrated in Fig. 4.1. These servers logically map to shared resources, where clients are the threads accessing those resources.

Early implementations of IPC had clients send messages directly to servers by referencing the thread ID. Later IPC message *ports* were introduced to provide isolation; clients send messages and wait for replies on ports, and servers receive messages on ports and reply to the client's message on that port, removing the need for threads to know details about each other. Servers effectively provide resources shared with multiple clients, via IPC through ports.

The heavy optimisation of second- and third-generation microkernels resulted in *timeslice donation*, and optimisation which avoided the scheduler [Heiser and Elphinstone, 2016]. Timeslice donation works as follows: the client calls the server, if the server is higher or equal priority it is switched to directly without invoking the scheduler at all, effectively a yield. The intuition is that in a fast IPC system, the request should be finished before the timeslice expires. In reality, longer requests do occur, so while the client's timeslice is used for the start of the request, the server's timeslice is used beyond that. This results in no proper accounting of the server's execution, and no temporal isolation.

Other kernels, like Barrelfish, allowed the sender to set a flag on a message specifying if timeslice donation should occur. However, this approach still suffers the problem of undisciplined charging, rendering timeslice donation an inappropriate mechanism for temporal isolation over shared resources.

Scheduling contexts

The mechanism of scheduling contexts is a more principled extension of timeslice donation. Thread structures in the majority of kernels contain the execution context (registers) and scheduling information, such as priority and accounting information, in a single structure known as a TCB. Other kernels, including real-time Mach, NOVA, and Fiasco.OC, divide the TCB into an execution context and *scheduling context* in order to allow the scheduling context to transfer between threads over IPC for accounting and/or priority inheritance purposes, and for performance [Steinberg et al., 2005].

The contents of a scheduling context vary per implementation. Real-time Mach's scheduling contexts contained parameters for the deferrable server, while NOVA's contained a timeslice. In Fiasco.OC, scheduling contexts contain a replenishment rule, and a budget. All three implementations include a priority in the scheduling context.

Scheduling context donation refers to a scheduling context transferring between threads over IPC, and is implemented in both Real-time Mach and NOVA, although the implementations are quite different. We look at both in terms of prioritisation, charging, and enforcement.

In Real-time Mach, scheduling donation would always occur and the scheduling context of the client always charged. In terms of prioritisation, a flag on the message port indicated if the server should run at the priority of the scheduling context, or a statically set priority. A further flag indicated if PIP should be implemented, where the server's priority would be increased to the priority of scheduling contexts from further pending requests [Kitayama et al., 1993]. Although this approach provided a fair amount of policy freedom, it introduces performance costs on the critical IPC path, on a kernel already notorious for its poor IPC performance [Härtig et al., 1997]. The enforcement policy of Real-time Mach derived directly from its two-level scheduler, and if a scheduling context expired the server would run in the second-level time-sharing scheduler, blocking any pending requests.

NOVA [Steinberg and Kauer, 2010] also provided scheduling contexts with donation over IPC, although the prioritisation policy was strictly PIP, which has high preemption overhead, as described in Section 2.3.2, and conflicts with the policy-freedom goal of a microkernel. Enforcement and charging in NOVA are both provided through the mechanism of helping [Steinberg et al., 2010]: a form of bandwidth inheritance, where pending clients not only boost the priority of the server, but the server can charge the current clients request to the pending client in order to finish the initial request.

In both implementations of scheduling context donation, the charging mechanism becomes policy: the scheduling context of the client is always charged. Although Mach

provided a mechanism to allow for system-specific prioritisation, this came at a considerable IPC cost. Finally, both provide different enforcement mechanisms but both are hard-coded policy which rely on either charging the next client, or running in slack. While scheduling contexts are a good mechanism for passing reservations across IPC, and thereby implementing temporal isolation over shared resources, the state of the art is insufficient.

4.3.5 Capabilities to time

Recall from Section 2.4.2 that capabilities [Dennis and Van Horn, 1966] are an established mechanism for fine-grained access control to system resources. Third-generation microkernels use capabilities for principled access to system resources, including KeyKOS, EROS, Fiasco.OC, NOVA, seL4, COMPOSITE and Barrelfish. Of those, only KeyKOS, EROS and COMPOSITE apply the capability system to processing time, we explore these systems after differentiating temporal and spatial capabilities.

The major challenge of applying capabilities to time is the fact that time cannot be treated as fungible. This is very different to spatial resources like memory, which is rendered fungible by the flexibility of virtual memory systems: a page of memory can be swapped for another page, as long as it has the correct contents. One window of time is not replaceable for another window, even in the case of a best-effort task: a minor change in schedule can force a deadline miss somewhere else. Consider real estate: like time, it is (arbitrarily) divisible but not fungible: If a block is too small to build a house, then having a second, disconnected block of the same size is of no help (unlike spatial resources in a kernel, which can be mapped side-by-side). The implication is that capabilities for time have a different flavour from those for spatial resources—they cannot support hierarchical delegation without loss, and cannot be recursively virtualised. While delegation is an attractive property of spatial capabilities, this delegation is not their defining characteristic, which is actually *prima facie evidence of access*; in the case of time capabilities, the access is to processor time. Previous implementations all have caveats and limitations, which we now detail.

KeyKOS provided capabilities to *meters*, which granted the holder the right to execute for the unit of time held by the meter, however as established in Section 4.3.2, meters are not a suitable mechanism for temporal isolation.

EROS [Shapiro et al., 1999] combined processor capacity reserves with capabilities rather than the meters of KeyKOS. Additionally, However, processor capacity reserves were *optional*: a two level scheduler first scheduled the reserves with available capacity, then threads without reserves, or with exhausted reserves, were scheduled. Like any hierarchical scheduling model, this enforces a policy that reduces flexibility. Furthermore, hierarchical delegation has the significant disadvantage of algorithmic utilisation loss [Lackorzyński et al., 2012]; this is a direct result of the unfungible nature of time.

COMPOSITE recently introduced temporal capabilities [Gadepalli et al., 2017] reminiscent of the meters of KeyKOS in that they represent a scalar unit of time. Unlike KeyKOS, temporal capabilities integrate with user-level scheduling, decoupling access control from

scheduling. An initial capability *Chronos* provides authority to all time, and is used to provide an initial allocation of time and for further replenishment. The kernel activates user-level schedulers which must then provide not only a thread to run, but a temporal capability to drain time from. On expiry, the user-level scheduler is also invoked, however this is a rare occasion as a well-built scheduler will ensure threads have sufficient capabilities on each scheduling decision. A notion of *time quality* supports delegation across hierarchically-scheduled subsystems without explicitly mapping all local priorities onto a single, global priority space, although for performance reasons, the number of supported delegations is statically limited. Additionally, time capabilities cannot be revoked unless empty. COMPOSITE's mechanism of temporal capabilities is the most promising, and decoupling access control from scheduling is surely key to providing mechanisms for temporal isolation without heavily restricting policy freedom. However, the static limit on delegations and lack of revoke makes the design incomplete for a production system, especially as revoke is generally the hardest part to implement in a capability system.

4.4 Summary

In this chapter we have reviewed standards and specifications for real-time operating systems, commercial and open-source real-time operating systems, and finally, surveyed the state-of-the-art research into mechanisms for temporal isolation, resource sharing, and access control to processing time.

While real-time literature is divided between which real-time scheduling algorithm should be deployed as the core of real-time systems (EDF or FP), no such divide exists in industry where all OSes provide FP. Except in a pure research sense, kernels that provide EDF do so in addition to FP. Resource reservations and EDF are more common in research OSes, whilst commercial and open-source products are heavily influenced by ARINC 653 and provide static partitioning.

As we have seen, many existing systems conflate criticality and time sensitivity in a single value: priority. A further assumption is that high criticality, time-sensitive tasks are always trusted, one that falls apart in the mixed-criticality context.

The *criticality* of a component reflects its importance to the overall system mission. Criticality may reflect the impact of failure [ARINC] or the utility of a function. An MCS should degrade gracefully, with components of lower criticality (which we will for simplicity refer to as LOW components) suffering degradation before higher criticality (HIGH) components.

Time sensitivity refers to how important it is to a thread to get access to the processor at a particular time. For best-effort activities, time is fungible, in that only the amount of time allocated is of relevance. In contrast, for a hard real-time component, time is largely unfungible, in that the allocation has no value if it occurs after the deadline; soft real-time components are in between.

Finally, *trust* refers to the degree of reliance in the correct behaviour of a component. Untrusted components may fail completely without affecting the core system mission, while a component which must be assumed to operate correctly for achieving the overall mission is trusted. A component is *trustworthy* if it has undergone a process that establishes that it can be trusted, the degree of trustworthiness being a reflection of the rigour of this process (testing, certification, formal verification) [Veríssimo et al., 2003].

In practice, criticality and trust are closely aligned, as the most critical parts should be the most trustworthy. However, criticality must be decoupled from time sensitivity in MCS. Referring back to the example in the introduction, interrupts from networks or buses have high time sensitivity, but low criticality (i.e. deadline misses are tolerable), while the opposite is true for the flight control component. Similarly, threads (other than the most critical ones which should have undergone extensive assurance) cannot be trusted to honour their declared WCET.

Our claim is that how these attributes are conflated is policy that is specific to a system. We need a mechanism that allows enforcing time limits, and thus isolate the timeliness of critical threads from those of untrusted, less critical ones. Reservation-based kernels often allow for a form of over-committing where best-effort threads are run in the slack-time left by unused reservations or unreserved CPU. However, this also aligns criticality and time-sensitivity, and enforces a two-level scheduling model.

If trustworthiness and real-time sensitivity are not conflated, many assumptions about real-time scheduling fail. Much real-time analysis rely on threads having a known WCET, which implies that those threads are predictable, which implies trust. If a real-time thread is not expected to behave correctly, one cannot assume it will surrender access to the processor voluntarily. Consequently, the PIP/bandwidth inheritance (BWI) based scheduling-context donation mechanisms seen in this chapter are insufficient, forcing not only a protocol which causes extensive preemption overhead, but requiring shared servers to have known, bounded execution time on all requests.

5 seL4 Basics

So far we have provided a general background on real-time scheduling and resource sharing. As the final piece of background we now present an overview of the concepts relevant to the temporal behaviour of our implementation platform, seL4.

seL4 is a microkernel that is particularly suited to safety-critical, real-time systems with one major caveat: time is not treated as a first-class resource, and as a result support for temporal isolation is deficient. Three main features of seL4 support this claim: it has been formally verified for correctness [Klein et al., 2009, 2014], integrity [Sewell et al., 2011], confidentiality [Murray et al., 2013]. All memory management, including kernel memory, is at user-level [Elkaduwe et al., 2006]; Finally it is the only OS to date with full WCET analysis [Sewell et al., 2016].

In this section we will present the current state of relevant seL4 features in order to highlight deficiencies and motivate our changes. We present the capability system, resource management, communication including IPC and the scheduler, followed by an analysis of how the current mechanisms can be used in real-time systems.

First we introduce the powerful seL4 capability system, used to access all resources in the system—with the exception of *time*. The scheduler in seL4 has been left intentionally underspecified [Petters et al., 2012] for later work. The current implementation is a place holder, and follows the traditional L4 scheduling model [Ruocco, 2006]—a fixed-priority, round-robin scheduler with 256 priorities.

5.1 Capabilities

As a capability-based OS, access to any resource in seL4 is via capabilities (recall Section 4.3.5). Capabilities to all system resources are available to the initial task—the first user-level thread started in the system—which can then allocate resources as appropriate. Capabilities exist in a *capability space* that can be configured per thread or shared between threads.

Capability spaces (`cspaces`) are analogous to address spaces for virtual memory: where address spaces map virtual addresses to physical addresses, capability spaces map object identifiers to access rights. `cspaces` are formed of *capability nodes* (`cnodes`) which contain

capabilities, analogous to page tables in virtual memory, and can contain capabilities to further cnodes, which allows for multi-level cspace structure. A cspace address refers to an individual entry in some cnode in the capability space, and may be empty or contain a capability to a specific kernel resource. For brevity, a cspace address is referred to as a *slot*.

Each capability has three potential access rights: read, write and grant. How those rights affect the resource the capability provides access to depends on the type of resource, and is explained in the next section.

Various operations can be done on capabilities, which are summarised in Table 5.1. When a capability is copied or minted, it is said to be *derived* from the original capability. All derived capabilities can be deleted by using seL4_CNode_Revoke. There are restrictions on which capabilities can be derived and under what conditions, depending on what the capability provides access to. *Badging* is a special type of derivation which allows specific capability types to be copied with an unforgeable identifier. We discuss derivation restrictions and the use of badges further in this chapter. Any individual capability can be deleted, or revoked. The former simply removes a specific capability from a capability space, the latter removes all child capabilities.

5.2 System calls and invocations

seL4 has a two-level system call structure, based on capabilities. The first level of system calls, listed in Table 5.2, are distinguishable by system call number. The majority of system calls are for communication; seL4_Send, seL4_NBSend, seL4_Call, seL4_Reply are *sending*

Operation	*Description*
seL4_CNode_Copy	Create a new capability in a specified cnode slot, which is an exact copy of the other capability and refers to the same resource.
seL4_CNode_Mint	Like copy, except the new capability may have diminished rights and/or be badged.
seL4_CNode_Move	Move a capability from one slot to another slot, leaving the previous slot empty.
seL4_CNode_Mutate	Like move, except the new capability may have diminished rights and/or be badged.
seL4_CNode_Rotate	Atomically move two capabilities between three specified slots.
seL4_CNode_Delete	Remove a capability from a slot.
seL4_CNode_Revoke	Delete any capabilities derived from this capability.
seL4_CNode_SaveCaller	Saves the kernel generated resume capability into the designated slot.

Table 5.1: Summary of operations on capabilities provided by baseline seL4 [Trustworthy Systems Team, 2017]

.

System call	*Class*	*Description*
seL4_Send	sending	Invoke a capability.
seL4_NBSend	sending	As above, but the invocation cannot result in the caller blocking.
seL4_Recv	receiving	Block on a capability.
seL4_NBRecv	receiving	Poll a capability.
seL4_Reply	sending	Invoke the capability in the current thread's reply. capability slot.
seL4_Call	sending, receiving	Invoke a capability and wait for a response, can generate a reply capability.
seL4_ReplyRecv	sending, receiving	seL4_Reply, followed by seL4_Recv.
seL4_Yield	scheduling	Trigger the round-robin scheduler.

Table 5.2: seL4 system call summary. All system calls except seL4_Yield are based on sending and/or receiving messages.

system calls for sending messages; seL4_Recv, seL4_NBRecv are *receiving system calls*, for receiving messages. Finally seL4_Yield is a *scheduling system call* for interacting with the scheduler. An NB prefix indicates that this system call will not block.

Second-level system calls are called *invocations* and are modelled as sending a message to the kernel. All invocations are conducted by a sending system call. The kernel is modelled as if it is waiting for a message and receives one every time a system call is made, and sends a message as a reply. To determine the operation, the rest of the arguments to an invocation are encoded as a message to the kernel. Each capability type has a different set of invocations available, and on kernel entry the invoked capability is decoded to determine the action the kernel should take.

All the operations on capabilities that are listed in Table 5.1 are invocations on cnode capability addresses. For example, to copy a capability, one uses seL4_Call on a cnode, and provides the invocation code for seL4_CNode_Copy, as well as the arguments. In the case of seL4_CNode_Copy, one provides the slot of the capability being copied, in addition to the destination cnode and slot.

5.3 Physical memory management

All kernel memory in seL4 is managed at user-level and accessed via capabilities, which is key to seL4's isolation and security properties, but also essential for understanding the complexity and limitations of integrating new models into the kernel. Additionally, this allows for the ultimate in policy freedom: all resource allocation is done from user-level by those holding the appropriate capabilities. Capabilities to kernel memory contain a physical address and a type which indicates what type of memory is at that physical address. Options for different types are shown in Table 5.3.

Object	*Description*
Untyped	Memory that can be retyped into other types of memory, including untyped.
CNode	A fixed size table of capabilities.
TCB	A thread of execution in seL4.
Endpoint	Ports which facilitate IPC.
Notification object	Arrays of binary semaphores.
Page Directory	Top level paging structure.
Page Table	Intermediate paging structures.
Frame	Mappable physical memory.

Table 5.3: Major memory object types in seL4, excluding platform specific objects. For further detail consult the seL4 manual [Trustworthy Systems Team, 2017]

.

In the initial system state, capabilities to all resources are given to the first task started by the system, the *root task*. Then according to system policy the root task can divide up and delegate system resources. This includes capabilities to all memory, apart from the small section of static state (e.g. a pointer to the current thread) used by the kernel. The kernel itself has a large, static *kernel window* initialised at boot time, which consists of memory mapped such that it is directly writeable by the kernel. The kernel window size is platform specific, but is 500MiB on all 32-bit platforms.

5.3.1 Untyped

All memory starts as *untyped* memory, and capabilities to all available untyped memory are placed in the `cspace` of the root task on boot. Each untyped consists of a start address, a size, and a flag indicating whether the untyped is writeable by the kernel or not. Memory reserved for devices and memory outside the kernel window is not readable or writeable by the kernel: the rest is untyped memory, free for use by the system.

Untyped objects have only one invocation: *retype*, which allows for large untyped objects to split into smaller objects of a different size and type, including frames, page tables, cnodes, etc. While the majority of objects in seL4 have a platform-dependent size fixed at compile time, some are sized dynamically at runtime, e.g. untyped and `cnodes`, which can be any power of two size.

Any capability to memory—untyped or not—is a capability to a specific object in memory, containing a pointer to that object. When retype is used to create sub-objects from an untyped object, those subsets of memory will not become available for retyping again until every capability to that object has been deleted, somewhat like reference counting pointers.

Start physical address	*End physical address*	*Kernel virtual address*
0x100000	0x2c00000	✗
0x10000000	0x10010000	0xe0000000
0x105c3000	0x2f000000	0xe105c3000
0x2f106400	0x2fdfc200	0xff106400

Table 5.4: Initial memory ranges on at boot time on the SABRE platform.

Table 5.4 shows an example initial memory layout for the SABRE platform[1], which has a 500MiB kernel window. Physical memory on this platform starts at 0x10000000, which is mapped into the kernel address space at 0xe0000000. Physical addresses outside of this range are devices and are not writeable by the kernel. Capabilities to all available memory and devices are set up as untyped in the initial root task's cnode.

5.3.2 Virtual Memory

Page tables, intermediate paging structures and physical frames are all created by retyping untyped objects. Page tables and frames have a set of architectural invocations including mapping, unmapping, and cache flushing operations.

5.3.3 Thread control blocks

TCBs represent an execution context and manage processor time in seL4, and consist of a base TCB structure and a cnode. The base TCB structure contains accounting information for scheduling and IPC in addition to the register set and floating-point context. The cnode contains capabilities that should not be deleted while a thread is running leveraging the fact that an object cannot be truly deleted until all capabilities to it are removed. These capabilities include the top-level cnode, top-level page directory, and three capabilities for IPC. Table 5.5 shows the main invocations possible on TCB capabilities.

5.3.4 Endpoints

Endpoints are the general communication port used by seL4 for IPC. Any thread with a capability to an endpoint can send and receive messages on that endpoint, subject to the access rights. Endpoints are small, and consist of the endpoint badge, some state information and a queue of threads blocked on the endpoint. We cover how endpoints interact with IPC in the upcoming Section 5.5.1.

[1] SABRE a 32-bit ARM system-on-chip, and the verification platform for seL4. Further details are provided in the evaluation.

5.3.5 Notifications

Notification objects are an array of binary semaphores used to facilitate asynchronous communication in seL4, either from other threads via `seL4_Send` or from interrupts. Notification objects consist of a queue of blocked threads and a word of information, which contains the semaphore state. Further information on notifications is presented in Section 5.5.4.

5.3.6 Consequences

User-level management of kernel memory provides true policy freedom, as there is no policy required by the kernel on memory layout, but this design is not without trade-offs. Ultimately, user-level management of kernel memory is key to seL4's isolation guarantees as system designers can fully partition systems by using specific memory layouts and avoid shared structures determined by the kernel itself. However, there are two major impacts on kernel design: back-pointers, and dynamic data structures.

The fact that any capability may be deleted at any time means that any pointer between two memory objects must be doubled, with pointers from each object to each other, as any object must be able to be traversed in order to reach any other linked object. This is analogous to a doubly-linked list, where for $O(1)$ deletion from any node in the list, the list must have pointers in both directions. This increases performance of deletion but also doubles the memory required for each list node.

Secondly, because of the policy freedom, the kernel has no limits on resources, meaning no memory resource can be statically sized. Consequently, simple, static data structures, like arrays, cannot be used in the kernel as no assumptions on memory layout can be made, meaning dynamic data structures are mandated. Because the kernel cannot make assumptions about the location of memory objects, specific optimisations are also not

Operation	*Description*
`seL4_TCB_Resume`	Place a thread in the scheduler.
`seL4_TCB_Suspend`	Remove a thread from the scheduler.
`seL4_TCB_WriteRegisters`	Configure a thread's execution context.
`seL4_TCB_ReadRegisters`	Read a thread's execution context.
`seL4_TCB_SetAffinity`	Set the CPU on which this thread should run.
`seL4_TCB_SetIPCBuffer`	Set the page to use for the IPC buffer.
`seL4_TCB_SetPriority`	Set the priority of this TCB.
`seL4_TCB_BindNotification`	Bind this TCB with a notification object (see Section 5.5.4).

Table 5.5: Summary of operations on TCBs. Further operations are available that batch several setters to reduce thread configuration overheads.

.

Slot(s)	*Contents*	*Slot(s)*	*Contents*
0	empty	8	`seL4_IOSpaceControl`
1	initial thread's TCB	9	initial thread start-up information frame
2	this `cnode`	10	initial thread's IPC buffer frame
3	initial thread's page directory	11	`seL4_DomainControl`
4	`seL4_IRQControl`	12—18	initial thread's paging structures
5	`seL4_ASIDControl`	19—1431	initial thread's frames
6	initial thread's ASID pool	1431—1591	untyped
7	`seL4_IOPortControl`	1592—$(2^{16}-1)$	empty

Table 5.6: Slot layout in the initial `cnode` set up by the kernel for the root task on the SABRE platform.

possible: it is up to the system designer to decide a trade-off between isolation and efficiency in the memory layout of the system.

Both of these consequences make for restrictions on kernel design, which can be demonstrated through the list of TCBs maintained by the kernel scheduler. Each priority in the scheduler has a doubly-linked list of TCBs: although the scheduler itself only ever removes the head of the list, which is $O(1)$ on a singly-linked list, a doubly-linked list is required as TCB objects can be deleted at any time. Memory placement of TCBs impacts scheduler performance, as depending on allocation patterns, different list nodes may trigger cache misses or worse, cache conflicts. As a result, the scheduler will perform far worse than a static, array-based scheduler using a fixed maximum number of TCBs with known ids for indexing. However, such an approach provides no policy freedom, and is more suitable at a middle-ware level in the OS implemented on top of the microkernel.

5.4 Control capabilities

Not all capabilities refer to memory-based resources, such as interrupts and I/O ports. In order to obtain capabilities to specific interrupts or ranges of I/O ports, the root task is provided with non-derivable control capabilities which can be invoked to place specific hardware resource capabilities in empty slots.

Table 5.6 shows the root task's initial `cspace` layout as set up by the kernel for the SABRE platform. Apart from capabilities to memory objects for the root task and the remaining untyped, the initial `cnode` contains five control capabilities for managing interrupts, address space IDs (ASIDs), I/O Ports, I/O Spaces and domains.

We take interrupts as an example to explain how control capabilities function. The `seL4_IRQControl` capability is the control capability for obtaining capabilities to specific

System call	*Action*
`seL4_Send`	Send a message, blocking until it is consumed.
`seL4_NBSend`	Send a message, but only if it is consumed immediately (i.e. a thread is already waiting on this endpoint for a message).
`seL4_Recv`	Block until a message is available to be consumed from this endpoint.
`seL4_NBRecv`	Poll for a message—consume a message from this endpoint, but only if it is available immediately.
`seL4_Call`	Send a message, and block until a reply message is received.
`seL4_Reply`	Send a reply message to a `seL4_Call`.
`seL4_ReplyRecv`	Send a reply message to a `seL4_Call` and then `seL4_Recv`.

Table 5.7: System calls and their effects when used on endpoint capabilities.

interrupt numbers. By invoking the `seL4_IRQControl` capability, users can obtain IRQHandler capabilities to specific interrupt numbers, thereby granting them authority to invoke that handler. Once an IRQHandler capability is obtained, users can invoke it to manage that specific interrupt. On x86, `seL4_IRQControl` is also used to provide capabilities to model specific registers and I/O interrupts.

5.5 Communication

seL4 provides communication through synchronous IPC via endpoints, or asynchronous notifications via notification objects. All the communication system calls introduced in Table 5.2, when used on endpoints and notifications, are used to communicate between TCBs. There are no invocations that can take place via endpoints or notification capabilities.

5.5.1 IPC

IPC in seL4 consists of threads sending and receiving messages in a blocking or non-blocking fashion over endpoints, which act as message ports. Each thread has a buffer (referred to as the *IPC buffer*), which contains the payload of the message, consisting of data and capabilities. Senders specify a message length and the kernel copies this (bounded) amount between the sender and receiver IPC buffer. Small messages are sent in registers and do not require a copy operation. Along with the message the kernel additionally delivers the badge of the endpoint capability that the sender invoked to send the message.

IPC can be one-way, where a single message is sent between a sender and receiver, or two-way in an RPC fashion where the sender sends a message and expects a reply. *IPC rendezvous* refers to when the IPC takes place, specifically when the kernel transfers data and capabilities between two threads.

Table 5.7 summarises seL4 system calls when used on endpoint capabilities. Essentially, `seL4_Send` is used to send a message and `seL4_Recv` used to receive one, both having blocking

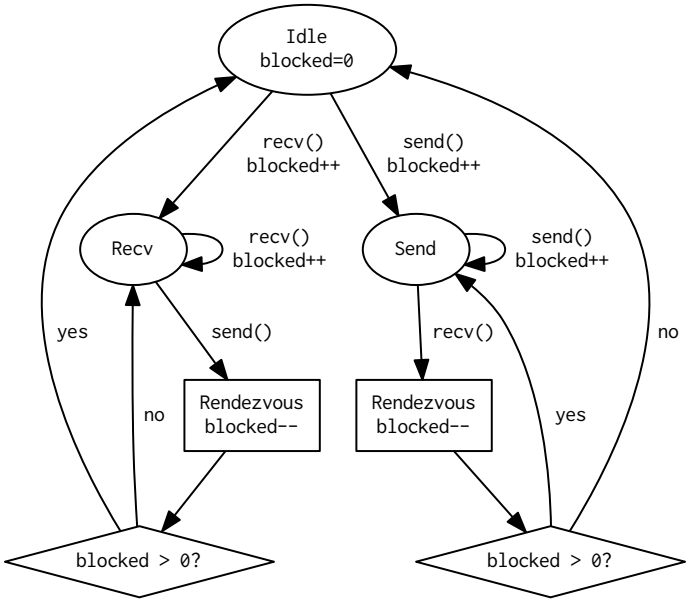


Figure 5.1: State diagram of a single endpoint, where *blocked* tracks the number of threads waiting to send or receive. Note that only one list of threads is maintained by the endpoint: senders and receivers cannot be queued at the same time.

and non-blocking variants. `seL4_Call` is of particular interest as it represents the caller side of an RPC operation critical to microkernel performance, distinguished from a basic pair of `seL4_Send` and `seL4_Recv` by *resume* capabilities. Pioneered in KeyKOS [Bomberger et al., 1992] and further appearing in EROS [Shapiro et al., 1999], resume capabilities are single-use capabilities, generated when a message is sent by `seL4_Call` and consumed when the reply message is received. Resume capabilities grant access to a blocked thread waiting for a reply message, and allow holders to resume that thread. In seL4, the resume capability is stored in the TCB `cnode`, and `seL4_Reply` is the operation used to invoke this capability and send the reply message back.

Figure 5.3a demonstrates the rendezvous phase, where regardless of the order of operations, when one thread blocks (`seL4_Recv`) on the endpoint and another thread sends on that endpoint then the message is consumed by the receiver. This occurs for both one-way and two-way IPC. Receivers can save the *resume* capability into their `cspace` to send a reply to after receiving other messages that would override the resume slot, but otherwise the resume capability is installed in the TCB `cnode`. The *reply* system call directly invokes the resume capability in this slot.

Multiple senders and receivers can use the same endpoint, which act as FIFO queues. In order to distinguish senders, receivers can use endpoint badges, which are unforgeable as they are copied by the kernel directly.

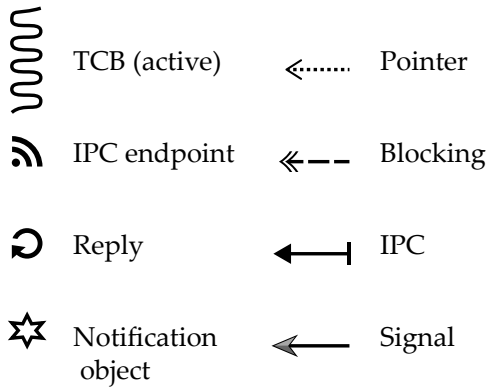
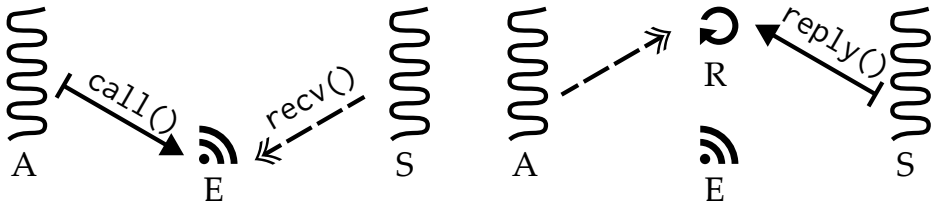


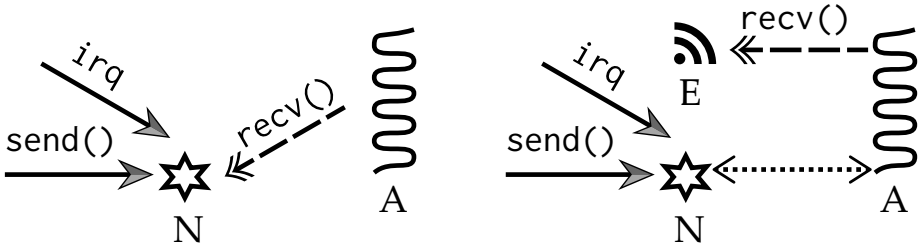
Figure 5.2: Legend for diagrams in this section.



(a) Initial IPC rendezvous.

(b) Reply phase.

Figure 5.3: IPC phases: a TCB, *A* sends a message to endpoint *E* using `seL4_Call`. Another TCB, *S*, blocks on *E* using `seL4_Recv`. At this point the message is transferred from *A* to *S* and *A* is blocked on the reply capability. (b) shows the reply phase, where *S* uses `seL4_Reply` to send a reply message to *A*, waking *A*. See Figure 5.2 for the legend.



(a) Notifications.

(b) Notification binding.

Figure 5.4: Example of a thread, TCB *A*, receiving notifications, by blocking on the notification and by notification binding. In Fig. 5.4a *A* blocks waiting on notification object *N*, and wakes when any notifications or interrupts are sent to *N*. In Fig. 5.4b, *A* blocks on endpoint *E*, however since *N* is bound to *A*, if *N* receives an interrupt or notification, *A* is woken and the data word delivered to *A*. See Figure 5.2 for the legend.

5.5.2 Fastpath

Recall from Section 2.4.1 that IPC performance is critical to a microkernel, which is achieved through optimised fastpaths. A fastpath is an optimised code path for the most performance critical operations. seL4 contains two IPC fastpaths which are used when the following, common-case conditions are satisfied:

1. the sender is using `seL4_Call` or the receiver is using `seL4_ReplyRecv`,
2. there are no threads of higher priority than the receiver in the scheduler,
3. the receiver has a valid address space and has not faulted,
4. and the message fits into registers, and thus does not require access to the IPC buffer.

5.5.3 Fault handling

Fault handling in seL4 is modelled as IPC messages between the kernel and receivers. TCBs can have a specific fault endpoint registered, on which the kernel can send simulated IPC messages containing information about the fault. Fault handling threads receive messages on this endpoint as if the faulting thread had sent a message to that thread with `seL4_Call`. Of course, this message is actually constructed by the kernel, and the message contains information about the fault, generally the faulting thread's registers. The faulting thread is blocked on the resume capability, which is generated, just as if the faulting thread had conducted a `seL4_Call`. The fault handling thread can subsequently reply to this message to resume the thread, although the reply message is also special: it can be used to reply with a new set of registers for the faulting thread to be resumed with, and tell the kernel if it should restart the thread or leave it suspended after the reply message is processed. If no fault endpoint is present, the thread is rendered inactive and no fault message is sent.

5.5.4 Notifications

Notification objects provide the mechanism for semaphores in seL4, and consist of a word of data. Sending on a notification either sends the badge of the invoked notification capability to a thread waiting on that notification object or—if no threads are waiting—stores the badge in the data word of the notifications. Unlike IPC, where messages being sent are queued, notifications accumulate messages in the data word.

The data word is set when the first notification arrives, and further invocations continue to bitwise OR the badge and data word until a thread receives the signal and clears the word. This is illustrated in the notification state diagram depicted in Fig. 5.5, and in Fig. 5.4a which shows the notification object and TCB interaction.

Table 5.8 shows the operations that occur when notification objects are invoked with relevant system calls. Fig. 5.5 depicts changes in the notification object state that occur when threads notify and block on notifications.

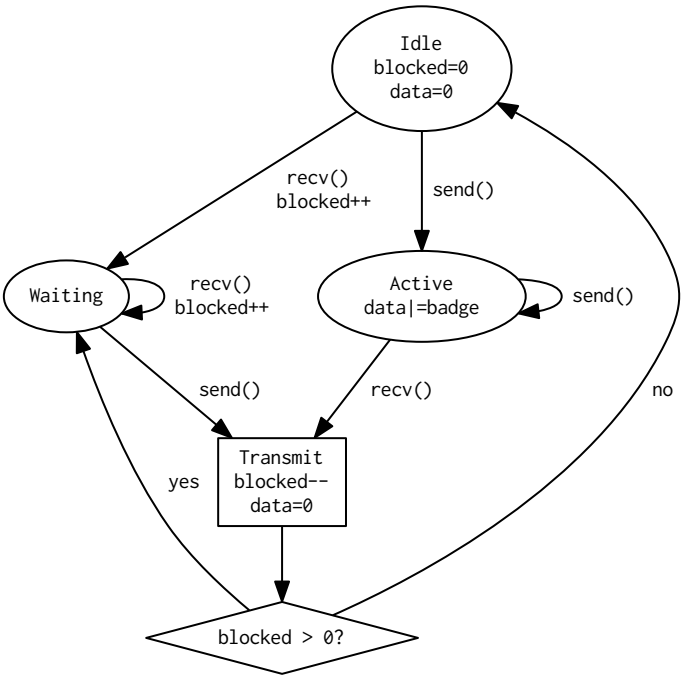


Figure 5.5: Notification object state transitions based on invocations. `seL4_Send` and `seL4_NBSend` correspond to notify in the diagram, while wait corresponds to a `seL4_Recv`.

Interrupts

In addition to providing a mechanism for threads to notify each other, notification objects also allow threads to synchronise with devices via polling and/or blocking for interrupts. IRQHandler capabilities can be associated with a single notification object, via the invocation `seL4_IRQHandler_SetNotification`, which results in the kernel issuing a notification when an interrupt occurs. The badge of the notification capability provided to the invocation is bitwise ORed with the data word when an interrupt is triggered. Further notifications are not issued by the kernel until the interrupt is acknowledged, using the `seL4_IRQHandler_Ack` invocation on `IRQHandler` capabilities.

System call	*Action*
`seL4_Send`	Send a notification, transmitting the badge; do not block.
`seL4_NBSend`	As above.
`seL4_Recv`	Wait until a notification is available then receive the data word.
`seL4_NBRecv`	Poll for a notification, do not block, receive the data word if available.

Table 5.8: System calls and their effects when used on notification objects.

Notification binding

Some systems require threads that can receive both notifications and IPC while blocked, in order to prevent the requirement that services which receive both IPC messages and notifications be multi-threaded. The mechanism for this is *notification binding* where threads can register a specific notification capability to receive notifications from while blocked on an endpoint waiting for IPC. This is done by invoking the TCB with the `seL4_TCB_BindNotification` invocation, which establishes a link between a TCB and notification object. Subsequently, if a notification is sent on that notification object and the TCB receives on any endpoint, that TCB will receive the notification. Without notification binding, services require a thread for blocking on a notification and another thread for blocking on an endpoint, both threads must then synchronise carefully on any shared data. Notification binding is illustrated in Fig. 5.4b.

5.6 Scheduling

The scheduler in seL4 is used to pick which runnable thread should run next on a specific processing core, and is a priority-based round-robin scheduler with 256 priorities (0—255). At a scheduling decision, the kernel chooses the head of the highest-priority, non-empty list.

Implementation wise, the scheduler consists of an array of lists: one list of ready threads for each priority level. A two-level bit field is used to track which priority lists contain threads, in order to achieve a scheduling decision complexity of $O(1)$. In fact, the $O(1)$ scheduler is a recent addition to seL4: scheduling used to be far more expensive before the bit field was introduced.

The bit field data structure works as follows: the top-level consists of one word, each bit representing N priorities, where N is 32 or 64 depending on the word size. Bit 0 in the top level bit field, on a 32-bit system, represents the first 0–31 priorities, bit 1 the next 32-63, etc.: if a bit in the top level is set, it indicates that at least one priority in that range is active. The second level distinguishes which priorities in the range indicated by the first level are set, and consists of $256/N$ words, which are indexed by the first bit set in the top-level bit field. To construct a priority we use the hardware instruction count leading zeros (`CLZ`) to find the highest bit set in the top-level index. We take that index and use `CLZ` on the bottom level bit field corresponding to that index, then construct the priority from these two values, allowing the highest priority to be found by reading only two words. For example, on a 32-bit system if bit 1 is the highest bit set in the top-level bit field, we index entry 1 in the bottom level. If bit 5 is the highest bit set in the first entry of the bottom level, we then obtain $1 * 32 + 5 = 37$ as the highest priority. Listing 5.1 shows the logic for the scheduler bit field on 32-bit systems.

```
uint32_t top_level_bitmap = 0;
uint32_t bottom_level_bitmap[256 / (1u << 5u)];

void addToBitmap(word_t prio) {
  uint32_t l1index = prio >> 5u;
  top_level_bitmap |= (1u << l1index);
  bottom_level_bitmap[l1index] |= (prio & ((1u << 5) - 1u));
}

word_t getHighestPrio(void) {
  uint32_t l1index = 31 - CLZ(top_level_bitmap);
  l2index = 31 - CLZ(bottom_level_bitmap[l1index]));
  return (l1index << 5 | l2index);
}
```

Listing 5.1: Example algorithms for adding a priority to the scheduler bitmap and extracting the highest active priority, on a 32-bit system. Both operations are $O(1)$ and involve two memory accesses. `CLZ` is the hardware instruction for count leading zeros.

5.6.1 Scheduler optimisations

A scheduling decision needs to be made whenever a thread transitions from or to a runnable state. The majority of thread state transitions occur on IPC, which is critical for performance, as can be seen in the simplified thread state diagram shown in Fig. 5.6. The `BlockedOnObject`

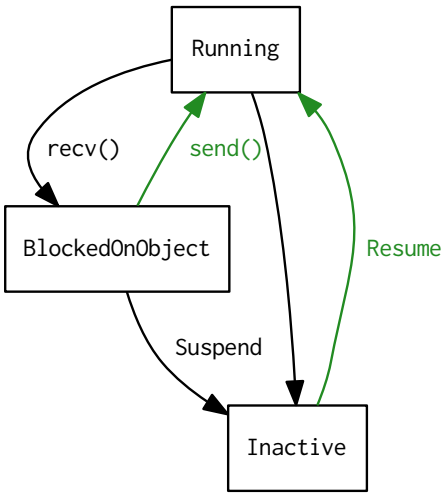


Figure 5.6: Thread state transitions in seL4.

state in the diagram maps to `blockedOn<N>` in the following thread state definitions:

- `Running`: This thread is eligible to be picked by the scheduler, and should be in the scheduling queues or be the currently running thread.

- `Inactive` This thread is not runnable, and is not in the scheduler. It has been suspended, or possibly never resumed.
- `BlockedOnRecv`: This thread is waiting to receive an IPC (or bound notification).
- `BlockedOnSend`: This thread is waiting to send an IPC.
- `BlockedOnReply`: This thread is blocked on a resume capability, waiting for a reply from a `seL4_Call` or fault.
- `BlockedOnNotification`: This thread is waiting to receive a notification.

Several optimisations to the scheduler exist in the kernel for performance, including lazy and benno scheduling. *Lazy* scheduling is used whenever a thread blocks and wakes another thread at the same time, for example, in the `seL4_Call` system call where the sender is blocked on the resume capability and the receiver is woken. Lazy scheduling observes that the current thread must be the highest-priority, runnable thread, and checks if the woken thread is higher than any other runnable thread by querying the scheduler bit field. If so, the receiver is directly switched to and the scheduler is avoided all together. Otherwise, the receiver is moved from the endpoint queue into the scheduling queue and the scheduler called.

Benno scheduling, named for its inventor Ben Leslie, removes the current thread from the scheduler, changing the invariant that all runnable threads are in the scheduler, so all runnable threads *except the current thread* are in the scheduler. Consequently, we avoid manipulating the scheduling queues when doing a direct switch, reducing the cache footprint and resulting in a performance improvement.

The result of these optimisations is that seL4, like L4 kernels before it, has very fast IPC, compared to non-L4 kernels e.g. at least a factor of four in CertiKOS [Gu et al., 2016].

Round-robin

Kernel time is accounted for in fixed-time quanta referred to as ticks, and each TCB has a timeslice field which represents the number of ticks that TCB is eligible to execute until preempted. The kernel timer driver is configured to fire a periodic interrupt which marks each tick, and when the timeslice is exhausted the thread is appended to the relevant scheduler queue, with a replenished timeslice. Threads can surrender their current tick using the `seL4_Yield` system call, which simulates timeslice exhaustion.

seL4 implements timeslice donation, as discussed in Section 4.3.4. When threads communicate with a service over IPC, the TCB of the thread providing the service is charged for any ticks incurred while that service is active. As a result isolation in shared servers is not possible: clients are not charged for their execution time.

Priorities

Like any priority-based kernel without temporal isolation mechanisms, time is only guaranteed to the highest priority threads. Priorities in seL4 act as informal capabilities: threads cannot create threads at priorities higher than their current priority, but can create threads at the same or lower priorities. If threads at higher priority levels never block, lower priority threads in the system will not run. As a result, a single thread running at the highest priority has access to 100% of processing time.

5.6.2 Domain scheduler

In order to provide confidentiality [Murray et al., 2013] seL4 provides a top-level hierarchical scheduler which provides static, cyclical scheduling of scheduling partitions known as *domains*. Domains are statically configured at compile time with a cyclic schedule, and are non-preemptible resulting in completely deterministic scheduling of domains. Each domain has its own priority scheduler, with lists per priority and a bit field structure, which is switched deterministically when a domain's ticks expire. Threads are assigned to domains, and threads are only scheduled when their domain is active. Cross-domain IPC is delayed until a domain switch, and `seL4_Yield` between domains is not possible. When there are no threads to run while a domain is scheduled, a domain-specific idle thread will run until a switch occurs.

The domain scheduler is consistent with that specified by the ARINC standard, and can be leveraged to achieve temporal isolation. However, since domains cannot be preempted, it is only useful for cyclic, non-preemptive scheduling with scheduling order and parameters computed-offline. In such a scenario each real-time task could be mapped to its own domain, and each task would run for its specified time before the domain scheduler switched to the next thread. Any unused time in a domain would be wasted, and spent in the idle thread.

Such a scheduler is only suitable for closed systems and results in low system utilisation. Dynamic real-time applications with shared resources and high system utilisation are not compatible with the domain scheduler, as they require preemption.

5.6.3 Real-time support

We introduced basic seL4 concepts and terminology, and investigated mechanisms that affect timing behaviour in the kernel: the scheduler, domains, priorities, `seL4_Yield` and IPC. In this section we will look at how real-time scheduling could be implemented with those mechanisms.

There are several options for implementing a real-time scheduler in the current version of seL4: leveraging the domain scheduler, using the priority scheduler or implementing a scheduling component to run at user-level. We explore these by explaining how a fixed-priority scheduler can be implemented with each technique, and explore the limitations.

	Temporal isolation	*Utilisation*	*Low kernel overheads*	*Dynamic*
Domain scheduler	✓	✗	✓	✗
Priority scheduler	✗	✓	✓	✓
Scheduling component	✓	✓	✗	✓

Table 5.9: Comparison of existing seL4 scheduling options.

A basic cyclic executive, as introduced in Section 2.2.1 can be implemented using the domain scheduler by mapping domains directly to task frames. This approach only works for closed systems with offline scheduling. An optimal scheduler like FPRM is not possible to implement with domains as they are not preemptible.

The priority scheduler can be leveraged to implement FPRM, by mapping seL4 priorities to rate-monotonic priorities. However, this requires complete trust in every thread in the system, as there is no mechanism for temporal isolation: if one thread executes for too long, other threads will miss their deadlines. Essentially the only thread with a guaranteed CPU allocation is the highest priority thread, which under rate-monotonic priority assignment is not the most critical thread in the system, but the thread with the highest rate. Additionally, periodic threads driven by timer interrupts rather than events would need to share a user-level timer.

To build a dynamic system with temporal isolation and high CPU utilisation, one could provide a high-priority scheduling component which implements FPRM at user-level. This task would manage a time-service for timeouts and monitor task execution, preempting tasks before they exceed their declared WCET. On task completion, tasks would RPC the scheduler in order to dispatch a new task. However, since the timer component would need to maintain accounting information and track the currently running thread, it would need to be invoked for every single scheduling operation. This is prohibitively expensive, as it results in doubled context switching time and increased number of system calls for thread management.

Table 5.9 shows a comparison of all the available scheduling options in the current version of seL4—no option provides all the qualities we require.

5.7 Summary

In this chapter we have provided an overview of seL4 concepts, including capabilities, system calls, resource management, communication and scheduling. We conclude that current real-time scheduling support is deficient, and temporal isolation is not possible over shared server IPC.

In the next section we will outline our model for a more principled approach to managing time by extending the baseline seL4 model presented in this chapter. We appeal to resource

kernel principles where time is treated as a first-class resource, with the aim of supporting diverse task sets, including those for mixed-criticality systems.

6 Design & Model

We now present our design and model for mixed-criticality scheduling support in a high-assurance system such as seL4.

Our goal is to provide support in the kernel for mixed-criticality workloads. This involves supporting tasks of different time sensitivities, (HRT, SRT, best-effort), different criticalities, and different levels of trust. Such tasks should not be forced into total isolation, but be permitted to share resources without violating their temporal correctness properties through mechanisms provided by the kernel.

To achieve this we require temporal isolation: a feature of resource kernels, the mechanisms of which we apply to general-purpose, high-assurance microkernels, and specifically our research platform, seL4. However, temporal isolation alone is insufficient: mixed-criticality systems require asymmetric protection rather than temporal isolation. As a result we leverage traditional resource kernel reservations but decouple them from priority, allowing the processor to be overcommitted while providing guarantees for the highest priority tasks.

In this chapter we first address how we integrate resource kernel concepts with the seL4 model, in order to provide principled access control to CPU processing time. We then describe our mechanisms for temporal isolation and asymmetric protection of resources shared between clients of different levels of criticality, time sensitivity, and trust. Finally, we show how our mechanisms can be used to build several existing user-level policies.

Our design goals are as follows:

Capability-controlled enforcement of time limits: In general, capabilities help to reason about access rights. Furthermore, they allow a seamless integration with the existing capability-based spatial access control of security-oriented systems such as seL4.

Policy freedom: In line with microkernel philosophy [Heiser and Elphinstone, 2016], the model should not force systems into a particular resource-management policy. In particular, it should support a wide range of scheduling policies and resource-sharing models, such as locking protocols.

Efficient: The model should add minimal overhead to the best existing implementations. In particular, it should be compatible with fast IPC implementations in high-performance microkernels.

Temporal isolation: The model must allow system designers to create systems in which a temporal failure in one component cannot cause a temporal failure in another part of the system.

Overcommitment: The model must allow systems to be specified that are not necessarily schedulable: as the degree and nature of overcommitment is a core policy of a particular system, as established in Section 4.3.3. Overcommitment is also key to providing asymmetric protection, where high criticality tasks can disrupt low criticality tasks, and policy freedom, as schedulability is system-specific policy.

Safe resource sharing: Temporal isolation should persist even in the case of shared-resources, to provide mechanisms for sharing between applications with different levels of time-sensitivity, criticality and trust.

The model provides temporal isolation mechanisms from the kernel, while allowing for more complex scheduling policies to be implemented at user level.

6.1 Scheduling

In Section 2.4.5, we outlined four core resource kernel mechanisms—admission, scheduling, enforcement and accounting—that are essential to resource kernels for implementing temporal isolation. However, such kernels are monolithic, where all policy, drivers and mechanisms are provided by the kernel.

Microkernels like seL4 offer a different design philosophy, based on the principle of minimality [Liedtke, 1995], where mechanisms are only included in the kernel if they would otherwise prevent the implementation of required functionality. Previous resource kernels are all monolithic, meaning that all resource policies are implemented in the kernel itself.

The goals of resource kernels do not directly align with that of microkernels in general. This is because microkernels do not directly manage all resources in the system, but provide mechanisms for the system designer to implement custom resource management policies.

In seL4, mechanisms for physical memory, device memory, interrupt and I/O port management are exposed to the user via the capability system, as outlined in Chapter 5. As a result, the only resource that the kernel needs to provide reservations for is processing time. We now present our mechanisms, and discuss how each of the four resource kernel policies can be implemented within our model.

6.1.1 Scheduling

There are two design choices relevant to scheduling:

- Should a scheduler be provided in the kernel at all?
- Should the scheduling algorithm be fixed (FP) or dynamic (EDF) priority?

Kernel scheduling

We retain the scheduler in the kernel, as per previous iterations of L4 microkernels, unlike COMPOSITE, for two reasons: to maintain a small trusted computing base, and for performance. Any system with multiple threads of execution, a requirement of mixed-criticality systems, must have a scheduler, which for the purposes of temporal isolation is part of the trusted computing base. Mixed-criticality systems require a scheduler: cooperative scheduling is not an option when combining components of different assurance levels. The scheduler itself is trusted, and must be at the highest assurance level, in a separate protection domain to components of lower criticality. As a result, a user-level scheduling decision will require at least two context switches for any component that is not at the highest criticality level: from the preempted thread to the scheduler, and from the scheduler to the dispatched thread. Given a mixed-criticality system requires a trusted scheduler, providing a basic scheduler in the kernel obviates this overhead as a requirement, and forms part of a complete trusted computing base.

Additionally, as the scheduler is a core component of the system it must be verified: by keeping the scheduler in the kernel we maintain the current verification. Verification of a user-level scheduler and its interaction with the kernel is a far more complex task, especially as verification of concurrent systems is very much an open research challenge.

One might claim that maintaining a scheduler in the kernel violates our goal of policy freedom. However, we maintain this is not the case, which comes down to our choice of fixed priorities over EDF.

Fixed priorities

Our model uses FP scheduling as a core part of the kernel, with the addition of mechanisms that allow for the efficient implementation of user-level schedulers. We choose FP over EDF for three reasons: fixed-priority is dominant in industry as shown in Chapter 4 and maps well to FP with RM priority assignment; and dynamic scheduling policies like EDF can be implemented at user level; and FP has defined behaviour on overcommitted systems.

EDF scheduling can be implemented by using a single priority for the dynamic priorities of EDF, as we will demonstrate in Section 8.4.2. However, the opposite is not true: mapping the dynamic priorities of EDF to a fixed-priority is non-trivial and would come with high overheads. Our approach is consistent with existing designs in Linux and ADA [Burns and Wellings, 2007], which support both scheduling algorithms, usually with EDF at a specific priority. Additionally, schedulability analysis of EDF-within-priorities is well understood [Harbour and Palencia, 2003].

We do not consider Pfair scheduling (recall Section 3.1.1) an option, as its high interrupt overheads and fairness properties are not suitable for hard real-time systems. Again however, it is possible to implement a Pfair scheduler at user level.

The final reason to base the approach on fixed priorities is the ability to reason about the behaviour of an overcommitted system. Overcommitting is important for achieving high actual system utilisation, given the large time buffers required by critical hard real-time threads. It is also essential to keeping the kernel policy-free: the degree and nature of overcommitment is a core policy of a particular system. For example, the policy might require that the total utilisation of all HIGH threads is below the FPRM schedulability limit of 69%, while LOW threads can overcommit, and the degree of overcommitment may depend on the mix of hard RT, soft RT and best-effort threads. Such policy should be defined and implemented at user level rather than in the kernel.

As discussed in Section 2.2.4, the result of overload in an EDF-based system is hard to predict, and such a system is hard to analyse. In contrast, overloading under fixed-priority scheduling is easy to understand: if the sum of utilisations of threads at priority $\geq P$ is below the utilisation bound, then all those threads will meet their deadlines, while any thread with priority $< P$ may miss. This allows easy analysis of schedulability.

6.1.2 Scheduling contexts

At the core of the model is the *scheduling context (SC)* as the fundamental abstraction for time allocation, and the basis of the enforcement and accounting mechanisms in our model. An SC is a representation of a reservation in the object-capability system, which means that they are first-class objects, like threads, address spaces, or IPC endpoints. An SC is represented by a capability to a scheduling context object.

In order to run, a thread needs an SC, which represents the maximum CPU bandwidth the thread can consume. Threads obtain SCs through *binding*, and only one thread can be bound to an SC at a time. In a multicore system, an SC represents the right to access a particular core. Core migration, e.g. for load balancing, is policy that should not be imposed by the kernel but implemented at user level. A thread is migrated by replacing its SC with one tied to a different core. This renders the kernel scheduler a partitioned scheduler, as opposed to a global scheduler, which aligns with our efficiency goal, since partitioned schedulers outperform global schedulers [Brandenburg, 2011].

The unfungible nature of time in real-time systems requires that the bandwidth limit must be enforced within a certain time window. We achieve this by representing an SC by sporadic task parameters, *period*, T, and a *budget*, C, where $C \leq T$ is the maximum amount of time the SC allows to be consumed in the period. $U = \frac{C}{T}$ represents the maximum *CPU utilisation* the SC allows. The SC can be viewed as a generalisation of the concept of a time slice, discussed in Section 4.3.2.

In order to support mixed-criticality systems, we do not change the meaning of priority, but what it means for a thread to be *runnable*: We associate each thread with an SC, and make it non-runnable if it has exhausted its budget.

SCs can be gracefully integrated into the existing model used in seL4 by replacing the time slice attribute with the scheduling context.

We retain thread priorities as attributes of the TCB, rather than the scheduling context. The advantage of keeping the two orthogonal allows us to avoid mandating a specific prioritisation protocol for resource-sharing, which we expand on in Section 6.2.

6.1.3 Accounting

Accounting must be a mechanism implemented by the kernel, as the kernel is the only entity in the system that can monitor all threads—regardless of preemption in the system—since the kernel facilitates all preemption. The accounting mechanism is provided by a new system invariant that the currently running thread must have a scheduling context with available budget, as all time consumed is billed to the current scheduling context. This includes time spent executing in the kernel, including preemptions. The rule for accounting kernel time is that all time, from kernel entry, is billed to the scheduling context active on kernel exit. This means that if a thread is preempted and that preemption does not trigger a switch to a different scheduling context, all time is accounted to the current scheduling context. Otherwise, time from kernel entry is accounted to the next scheduling context.

In order to facilitate user-level schedulers, the kernel tracks the amount of time consumed by a scheduling context, which can be retrieved by an invocation on that specific scheduling context object. We specifically do not cater for dynamic frequency scaling and ensure that it is turned off during testing—this is out of scope and a topic for future work.

6.1.4 Admission control

Unlike any previous kernel that supports reservations, our model delegates admission control to user level, as it is deeply tied to policy, which a microkernel should not limit. A consequence of this choice is that the design naturally supports over-commitment, as this is part of the admission test.

The mechanisms for admission control consist of two parts: scheduling contexts are created without any budget at all, and a new control capability must be invoked to populate scheduling context parameters. `seL4_SchedControl` is the new control capability, one of which is provided per processing core, and grants authority to 100% of time on that core, thus providing the admission-control privilege. This is analogous to how seL4 controls the right to receive interrupts, which is controlled by the IRQ_control capability as introduced in Section 5.4. Like time, IRQ sources are non-fungible.

Unlike the reservations in resource kernels, the scheduling context does not act as a lower-bound on CPU bandwidth that a thread can consume. This, combined with user-level admission control, is also key to allowing system designers to over-commit the system.

By designating admission tests as user-level policy, we allow system designers complete freedom in determining which admission test to use, if at all, and when that test should be done. Consequently, they can be run dynamically at run-time, or offline, as per user-level policy.

Thus, the kernel places no restriction on the creation of reservations apart from minor integrity checks (i.e. $C \leq T$). For example, some high-assurance systems may sacrifice utilisation for safety with a very basic but easily verifiable, online, admission test. Other implementations may conduct complex admission tests offline in order to obtain the highest possible utilisation, using algorithms that are not feasible at run time. Some systems may require dynamic admission tests that sacrifice utilisation or have increased risk. Basic systems may require a simple break up of the processing time into rate-limited reservations. By taking the admission test out of the kernel, all of these extremes (and hybrids of) are optional policy for the user.

A consequence of this design is that more reservations can be made than processing time available. This is a desirable feature: it allows system designers to overcommit the system, while features of the scheduling mechanisms provided by the kernel guarantee that the most important tasks get their allocations, if the priorities of the system are set correctly.

However, allowing any thread in the system to create reservations could result in overload behaviour and violation of temporal isolation. To prevent this, admission control is currently restricted to the task that holds the scheduling control capability for each processing core.

6.1.5 Replenishment

Scheduling contexts can be *full* or *partial*. A thread with a *full* SC, where $C = T$, may monopolise whatever CPU bandwidth is left over from higher-priority threads, while high-priority threads with full SCs may monopolise the entire system. Partial SCs, where $C < T$, are not runnable once they have used up their budget, until it is replenished, and form our mechanism for temporal isolation.

Full SCs are key to maintaining policy freedom and performance; while systems must be able to use our mechanisms to enforce upper bounds on execution, the usage of those mechanisms is policy. If a task is truly trusted, no enforcement is required, as in standard HRT systems. They can also be used for best-effort threads which run in slack time. The C of a full SC represents the timeslice, which once expired results in round robin scheduling at that priority. Additionally, full SCs provide legacy compatibility: setting $C = T$ to the previous timeslice value results in the same behaviour as the master kernel.

From a performance perspective full SCs prevent mandatory preemption overheads that derive from forcing all threads in a system to have a reservation. Threads with a full budget incur no inherent overhead other than the preemption rate $1/T$.

Threads with *partial* SCs have a limited budget, which forms an upper bound on their execution. For replenishment, we use the sporadic servers model as described in Section 3.1.3 with an implementation based on the algorithms presented by Stanovic et al. [2010].

The combination of full and partial SCs and the ability to overcommit distinguishes our model from that of resource kernels.

Sporadic servers

We use sporadic servers as they provide a mechanism for isolation without requiring the kernel to have access to all threads in the system, unlike other approaches discussed in Section 3.1.3, such as priority exchange servers and slack stealing. Deferrable servers do not require global state but are ruled out due to the back-to-back problem, which violates temporal isolation. Avoiding global shared-state is required for confidentiality, because access to shared state has cache effects, which leak information via cache-based timing-channels. In order to maintain these properties, timing channels must be mitigated through partitions. It must be possible to partition a system completely, such that operations in one partition do not alter the cache state in the other partitions.

Recall that sporadic servers work by preserving the *sliding window* constraint, meaning that during any time interval not exceeding the period, no more than C can be consumed. This stops a thread from saving budget until near the end of its period, and then running uninterrupted for more than C. It is achieved by tracking any leftover budget when a thread is preempted at time t, and scheduling a replenishment for time $t+T$.

In practice, we cannot track an infinite number of replenishments, so in a real implementation, once the number of queued replenishments exceeds a threshold, any excess budget is discarded. If the threshold is one, the behaviour degrades to polling servers [Sprunt et al., 1989] where any unused budget is lost and the thread cannot run until the start of the next period.

Replenishment fragmentation resulting from preemptions has an obvious cost and an arbitrarily high threshold makes little sense. Additionally, polling servers polling servers are more efficient in the case of frequent preemption [Li et al., 2014]. The optimal value depends on implementation details of the system, as well as the characteristics of the underlying hardware. We therefore make the threshold an attribute of the SC. SCs are variably sized, such that system designers can set this bound per SC. This is a generalisation of the approach used in Quest-V [Danish et al., 2011], where I/O reservations are polling servers and other reservations are sporadic servers. This policy can easily be implemented at user-level with variably sized SCs.

6.1.6 Enforcement

Threads may exhaust their budgets for different reasons. A budget may be used to rate-limit a best-effort thread, in which case budget overrun is not different to normal time-slice preemption of best-effort systems. A budget can be used to force an untrusted thread to adhere to its declared WCET. Such an overrun is a contract violation, which may be reason to suspend the thread or restart its subsystem. Finally, an overrun by a critical thread can indicate an emergency situation; for example, critical threads may be scheduled with an optimistic budget to provide better service to less critical threads, and overrun may require provision of an emergency budget or specific exception handler.

Clearly, the handling of overrun is a system-specific policy, and the kernel should only provide appropriate mechanisms for implementing the desired policy. Our core mechanism is the *timeout exception*, which is raised when a thread is preempted due to budget expiry. To allow the system to handle the exceptions, each thread is optionally associated with a timeout exception handler, which is the temporal equivalent to a (spatial) protection exception. When a thread is preempted, the kernel notifies its handler via an IPC message. The exception is ignored when the thread has no timeout exception handler, and the thread can continue to run once its budget is replenished.

The timeout fault endpoint is separate from the existing thread fault endpoint, as the semantics are different: non-timeout faults are not recoverable without the action of another thread. While threads are not required to have a fault endpoint, when a fault occurs a thread is always blocked, as it is no longer in a runnable state. Timeout faults on the other hand are recoverable: the thread must simply wait until budget replenishment.

Timeout fault handling threads should have their own SC to run on, with enough budget to handle the fault, otherwise they will not be eligible for selection by the kernel scheduler.

Similar to a page fault handler, timeout fault handlers can be used to adjust thread parameters dynamically as may be required for an SRT system, or raise an error. The handler has a choice of a range of overrun policies, including:

- providing a one-off (emergency) budget to the thread and letting it continue,
- permanently increasing the budget in the thread's SC, and
- killing/suspending the thread.

Obviously, these are all subject to the handler having sufficient authority (e.g. `seL4_SchedControl`).

If a replenishment is ready at the time the budget expires, the thread is immediately runnable. It is inserted at the end of the ready queue for its priority, meaning that within a priority, scheduling of runnable threads is round-robin.

The timeout fault handler is similar to the original KeyKOS [Bomberger et al., 1992] concept of meters, which was carried through to earlier L4 microkernels as a *preempter*. The preempter was invoked every time the timeslice/meter expired to make a scheduling

decision. This was abandoned due to performance reasons. Our approach differs in that timeout fault handlers are optional, so that they can be used in exceptional scenarios only. However, timeout fault handlers can also be used to facilitate user-level scheduling, which is feasible now that modern hardware has much faster context switch times.

6.1.7 Priority assignment

Assignment of priorities to threads is user-level policy. One approach is to simply use rate-monotonic scheduling: where priorities are assigned to threads based on their period, and threads use scheduling contexts that match their sporadic task parameters. Each thread in the system will be temporally isolated as the kernel will not permit it to exceed the processing time reservation that the scheduling context represents.

However, the system we have designed offers far more options than simple rate-monotonic fixed-priority scheduling. Policy freedom is retained since reservations simply grant a potential right to processing time, at a particular priority. What reservations actually represent is an upper bound on processing time for a particular thread. Low priority threads are *not* guaranteed to run if reservations at higher priorities use all available CPU. However, threads with reservations at low priorities will run in the system slack time, which occurs when threads do not use their entire reservation.

The implication is that a system could use a high range of priorities for rate-monotonic threads, while best-effort and rate-limited threads run at lower priorities. Another alternative is to map user-lever schedulers to specific priorities, for instance an EDF scheduler.

6.1.8 Asymmetric Protection

Recall that in a mixed-criticality system, asymmetric protection means that tasks with higher criticality can cause deadline misses in lower criticality tasks. Two approaches to mixed-criticality scheduling that support asymmetric-protection are slack scheduling and response time analysis (RTA) [Burns and Davis, 2017].

Under slack scheduling, low-criticality tasks run in slack time of high-criticality tasks. Our model supports this easily: high-criticality tasks are given reservations to all processing time at high priorities, and low-criticality tasks are given reservations at a lower priority band.

RTA relies on suspending or de-prioritising low-criticality tasks if a high-criticality task runs for longer than expected. Simple RTA schemes involve two system modes: a HI mode and a LO mode, although they have been generalised to more [Fleming and Burns, 2013]. In LO mode, high-criticality tasks run with smaller reservations, and the remaining CPU time is used for low-criticality tasks. If a high-criticality task does not complete before its LO-mode reservation is exhausted, the system switches to HI mode: all low criticality tasks are suspended. Such a criticality switch is also supported by our model, at user-level: a high-priority scheduling thread can be set up to receive temporal

faults when a task does not complete before its budget expires. On a temporal fault, the scheduling thread can modify the scheduling parameters of tasks appropriately, for example by boosting the high-criticality task's priority and budget.

6.1.9 Summary

The scheduling, accounting and enforcement mechanisms presented are sufficient to support temporally isolated, fixed-priority or EDF scheduled real-time threads. By keeping admission control out of the kernel we preserve policy freedom, while the mechanisms provided allow for a full resource kernel to be built, or other types of system which require the ability to over-commit. Additionally, we have maintained support for best-effort threads, and have added the ability to provide asymmetric protection for mixed-criticality workloads. In the next section, we show how our model provides mechanisms for resource sharing.

6.2 Resource sharing

Reservations through the mechanism of scheduling contexts are not sufficient alone to provide resource sharing between tasks of different time-sensitivity, criticality, and trust. We now present how the scheduling context mechanism facilitates resource sharing policies in terms of the three core mechanisms —prioritisation, charging and enforcement—presented in Section 2.4.5.

In this section we consider only resources shared by resource servers in an RPC style, in which servers are scheduled directly as a result of the RPC. Other forms of resource sharing, such as the asynchronous multicast provided by ARINC 653 (Section 4.1.2), are possible but not relevant, as time is not directly exchanged, and we have already shown how temporal partitioning is possible with reservations.

RPC-style shared servers are fundamentally a trusted entity: the client does not resume until the server replies to a request, and the server may not ever reply. Additionally, the client requires the server to carry out the operation requested. Consequently, the server takes on the ceiling of the requirements of all the clients. If a server is shared between trusted and untrusted clients, the server must be trusted as much as the most trusted client. If a server is shared between HRT and best-effort clients, the server must have the same time sensitivity as the HRT clients: a defined WCET and bounded blocking behaviour. Finally, the server must be at the highest criticality level of all clients, however the opposite is not true. If a server has sufficient isolation properties, then all the clients of that server are not also promoted to the highest criticality level, which as we know greatly increases the certification burden.

This asymmetry also holds for trust: a trusted server need not trust its clients, and the clients need not trust each other. Trusted scenarios do not require this level of encapsulation: user-level threads packages can implement locking protocols with libraries such as

`pthreads`. Therefore, our focus is on temporal isolation in shared servers beyond strict partitioning, where clients do not trust each other, and the server does not trust the client.

The proposed model supports shared servers, including scheduler bypass, through *scheduling context donation*: a client invoking a server can pass its SC along, so the server executes on the client's SC, until it replies to the request. This ensures that the time consumed by the server is billed to the client on whose behalf the server's work is done. If the scheduling context expires, the enforcement mechanism of timeout exceptions can be used to recover the server according to server specific policy.

6.2.1 Prioritisation

We indicated in Section 6.1.2 that scheduling contexts are separate from thread control blocks in order to support resource sharing. Additionally, unlike previous microkernel designs for SCs, priority remains an attribute of the execution context. Decoupling priority and scheduling context avoids prior patterns where priority-inheritance is enforced by the kernel.

The IPCP maps neatly to this model: if server threads are assigned the ceiling protocol of all of their clients, then when the IPC rendezvous occurs and we switch from client to server, the server's priority is used, a consequence of decoupling priority from the scheduling context. IPCP is also practical: it is mandated by the AUTOSAR standard (recall Section 4.1.3) and is therefore understood well by parts of the industry.

The main drawback of the IPCP, namely the requirement that all lockers' priorities are known *a priori*, is easy to enforce in a capability-based system: The server can only be accessed through an appropriate invocation capability, and it is up to the system designer to ensure that such a capability can only go to a thread whose priority is known or appropriately controlled.

The choice of the IPCP over other fixed-priority protocols, the PIP and the OPCP, is intentional, and we now present the case against both protocols.

In order to properly support real-time systems, our model orders endpoint queues by priority, rather than the traditional FIFO of L4 kernels. Threads of the same priority remain FIFO.

PIP

The major factor in ruling the PIP out as a mechanism provided by the microkernel is performance. Introducing PIP would drastically increase scheduler operations on the critical IPC fastpath, as priorities would need to be returned on the reply path. One could take after Real-time Mach, and provide PIP as an optional flag on a message port, thereby retaining policy freedom, but at a performance cost. Mandating the PIP, which incurs a high-preemption overhead compared to other protocols, would violate both our performance and policy freedom requirements.

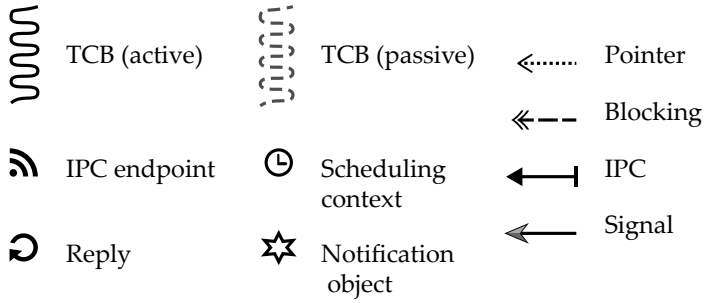


Figure 6.1: Legend for diagrams in this chapter, an expanded version of Fig. 5.2

There is an additional, practical concern, in that the PIP does not map well to the existing capability system. Recall from Section 5.5.1 that endpoints have very minimal state[1], and once an IPC rendezvous has occurred, the caller is blocked on the resume capability. Consequently, there is *no way to reach the caller* after an IPC message has been sent. Significant changes would be required to the IPC model to ensure threads blocked on resume capabilities could be tracked from the endpoint that the message was sent over.

6.2.2 OPCP

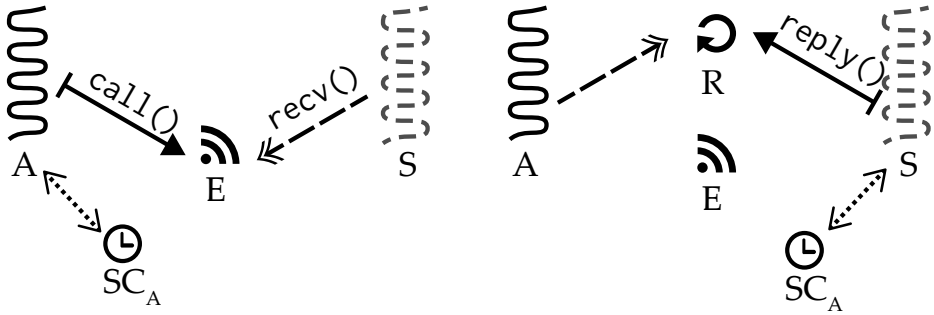
Although the OPCP offers greater processor utilisation than the IPCP, it would require the kernel to track a system ceiling and also implement priority inheritance, in a form where unrelated threads may inherit priorities due to a resource access pattern. Both of these mechanisms are required to implement the OPCP, and both would violate confidentiality by altering state across partitioned resources. This is the same reason we exclude priority-exchange and slack-stealing servers, as discussed in Section 6.1.5.

Although not provided by the kernel, PIP and OPCP can be implemented at user-level if required, by proxying requests to shared resources via an endpoint and thread which manipulates the priorities appropriately, before forwarding on the request. For PIP, only one extra thread per resource would be required and could exist in the same address space as the resource. Since OPCP requires global state, an implementation would require a shared service for all threads sharing a set of resources. We describe such an implementation in Section 6.3 after introducing further mechanisms in Section 6.4.

6.2.3 Charging

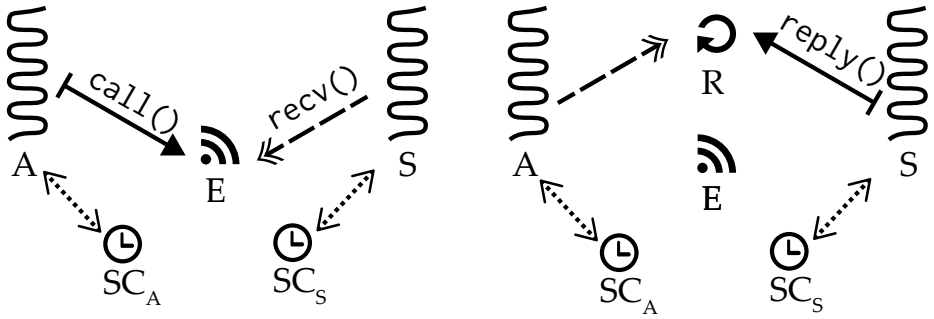
Charging for time executed in a server is simple in our model: charge the scheduling context that the resource server is running on. The question is, *which* scheduling context is the server running on? Unlike previous implementations of donation-like semantics, our model does not mandate scheduling context donation. Resource servers are free to run on their own scheduling context, according to the policy of the system.

[1] Which again contributes to high performance.



(a) Phase 1 (b) Phase 2

Figure 6.2: IPC phases between an active client and passive server: (a) shows the initial IPC rendezvous, (b) shows the reply phase. The client's SC_A is donated between client and server. See Figure 6.1 for the legend.



(a) Phase 1 (b) Phase 2

Figure 6.3: IPC phases between an active client and active server: (a) shows the initial IPC rendezvous, (b) shows the reply phase. Both client and server have their own SC. See Figure 6.1 for the legend.

Whether donation occurs or not is inferred at the time of the IPC rendezvous: we test if the server is passive or active. *Passive* servers do not have a scheduling context and receive them over IPC whereas *active* servers have their own scheduling context. Passive servers, as illustrated in Fig. 6.2, effectively provide a migrating-thread model [Ford and Lepreau, 1994; Gabber et al., 1999], but without requiring the kernel to manage stacks. Figure 6.3 shows active servers, which allow system designers to build systems without temporal isolation, as it is not suitable for all systems.

6.2.4 Enforcement

If a passive resource server exhausts the scheduling context it is running on, it and any waiting clients are blocked until replenishment. On its own, this means that a client not only has to trust its server, but all the server's other clients. This would rule out sharing a server between clients of different criticality.

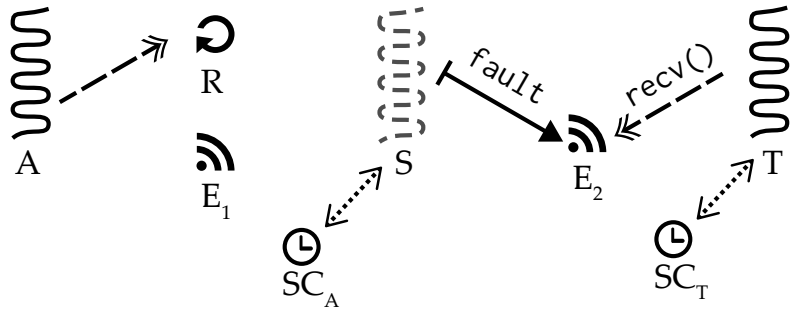


Figure 6.4: Example of a timeout exception handler. The passive server S performs a request on client A's scheduling context, SC_A. A timeout fault is generated when SC_A is exhausted and sent to the server's timeout fault endpoint E_2, and the timeout handler receives it. See legend Fig. 6.1

In a HRT system, we can assume that a client's reservations are sufficient to complete requests to servers. However, in systems with best-effort and soft real-time tasks, no such assumption can be made and client budgets may expire during a server request. This leaves the server in a state where it cannot take new requests as it is stuck without an active reservation to complete the previous request. Without a mechanism to handle this event the server, and any potential clients, would be blocked until the client's budget is replenished.

Timeout exceptions can be used to remove this need for trust, and allow a server to be shared across criticalities, as depicted in Fig. 6.4. Timeout fault endpoints are specific to the execution context of a thread, not the scheduling context. Consequently, servers may have timeout fault handlers while clients do not. The server's timeout handler can, for example, provide an emergency budget to continue the operation (useful for HI clients) or abort the operation and reset or roll back the server. The latter option is attractive for minimising the amount of budget that must be reserved for such cases.

A server running out of budget constitutes a protocol violation by the client, and it makes sense to penalise that client by aborting. Helping schemes, such as PIP or bandwidth inheritance, make the waiting client pay for another client's contract violation. This not only weakens temporal isolation, it also implies that the size of the required reserve budget must be a server's full WCET. This places a restriction on the server that no client request exceed a blocking time that can be tolerated by all clients, or that all clients must budget for the full server WCET in addition to the time the server needs for serving their own request. Our model provides more flexibility: a server can use a timeout exception to finish a request early (e.g. by aborting), meaning that clients can only be blocked by the server's WCET (plus the short clean up time).

6.2.5 Multiprocessors

The mechanism of scheduling-context donation can also be used across processing cores, and allows users to easily specify if servers should migrate to the core of the client, or

process the request on a remote core. Specifically, active servers process requests on the core of their own scheduling context, and passive servers migrate to the scheduling context of the client. The mechanism is simple: if a server is passive, it migrates to the core that the donated scheduling context provides time for.

6.3 Mechanisms

Before exploring how various user-level policies can be implemented at user-level using our model, we describe four new mechanisms introduced for high-performance implementation of those policies.

6.3.1 Directed yield

The first is simple: in order to implement user-level schedulers, the kernel's scheduling queues must be able to be manipulated. For this, we add an operation on scheduling contexts: a directed yield to a specified scheduling context. By invoking the `seL4_SchedContext_YieldTo` invocation on a specific scheduling context, the thread running on that scheduling context is placed at the head of the scheduler queue for its priority. This may change the currently running thread if it is at the same priority, and only effects the scheduler if there is a runnable thread bound to the invoked scheduling context. Threads cannot yield to threads of higher priorities than their own, as they are by definition not currently running.

6.3.2 IPC Forwarding

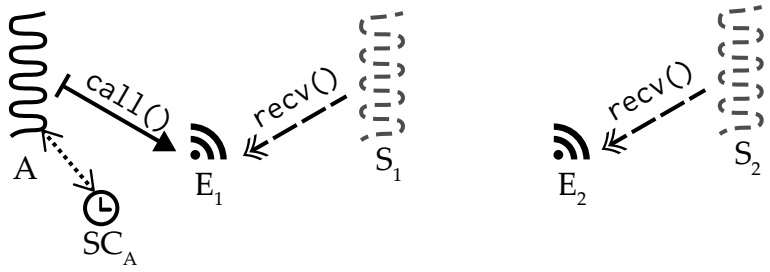
We introduce another concept called *forwarding*, which allows callers to invoke one capability and block on another capability in the same system call, via a new system call `seL4_NBSendRecv`. This is required to facilitate a forwarding IPC, and allows passive proxies to seamlessly forward messages, capabilities and scheduling contexts, as shown in Fig. 6.5.

6.3.3 Flexible Resume Capabilities

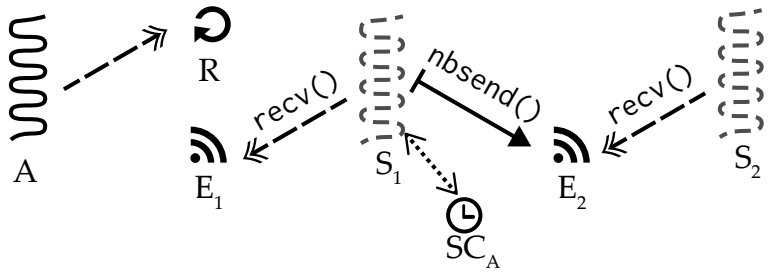
The final mechanism we introduce is the ability to transfer resume capabilities between servers. This means that a blocked thread can be forwarded to other threads, thus allowing the resume capability to be moved. Note that this is explicitly not delegation: resume capabilities cannot be copied.

6.3.4 Notification Binding

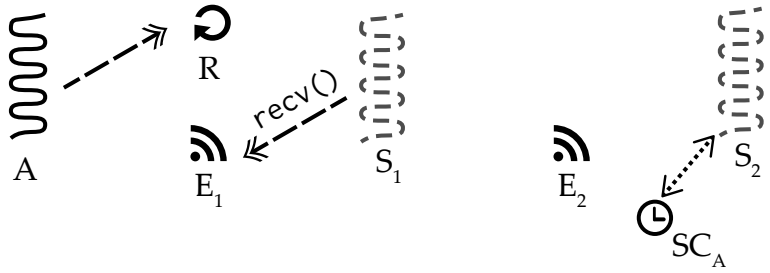
Recall from Section 5.5.4 that notification objects can be bound to threads such that single-threaded servers can be constructed which receive IPC messages and signals. We extend this mechanism with passive server support by allowing scheduling contexts to be bound to notifications, such that the passive server executes using the notification's scheduling context when processing interrupts or signals from other threads.



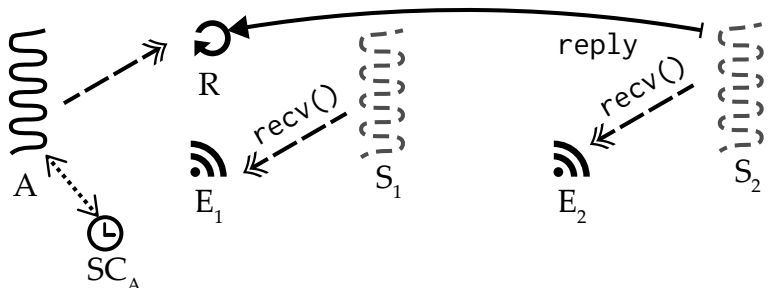
(a) A `seL4_Call` () S_1 over E_1.



(b) S_1 uses `seL4_NBSendRecv` () to forward the request, and A's resume capability, to E_2, and wait for further messages on E_1.



(c) S_2 processes the request.



(d) S_2 uses the `seL4_ReplyRecv` () system call to reply to A and block on E_2 again.

Figure 6.5: IPC forwarding: client A sends a request to passive server S_2 via a passive proxy S_1, and the scheduling context passes from $A \rightarrow S_1 \rightarrow S_2 \rightarrow A$. See Figure 6.1 for the legend.

6.4 Policies

We now describe how different policies can be implemented by user-level systems using the existing mechanisms in seL4, combined with the new mechanisms of scheduling contexts, scheduling context donation, directed yield, IPC forwarding, and flexible resume capabilities. First, we explain how different best-effort, rate-based and real-time tasks are compatible with our model and describe their implementation. We subsequently explain how various different resource sharing policies and servers, with and without temporal isolation, can be constructed.

6.4.1 Scheduling policies

Best-effort threads

Best-effort threads, scheduled round-robin, are compatible with our model, which maintains compatibility with the previous, baseline seL4 scheduler. Full scheduling contexts are the mechanism to support best-effort threads, and by setting the budget equal to the period, system designers can set a timeslice, which determines how long a specific scheduling context can be executed upon before preemption.

More complicated, time-sharing schedulers can be built by a user level scheduler, using the directed yield and consumed mechanisms, which allow user-level reorder the kernel's priority queues and request the amount of time consumed by a specific scheduling context.

Rate-limited threads

Rate-limited threads simply have their scheduling contexts configured with parameters that express the desired upper-bound on rate. No other work is required by the user: if there are no higher priority threads and the rate-limited thread does not block, it will be runnable at the rate expressed in the scheduling context. Otherwise, the thread will be capped at the rate specified, but cannot be guaranteed to get the entire rate allocation if there are higher priority threads in the system, or if the priority of the rate-limited thread is overloaded.

Periodic, polling threads

Threads that need to wake, poll for an event, and sleep again can be implemented using the sporadic server mechanism, by setting threads' scheduling parameters appropriately. As long as the budget is not full ($C \neq T$), the `seL4_Yield` system call can be used to block until the next replenishment is available, as shown in Listing 6.1. However, this approach will only work if the amount of thread preemptions is known and the amount of extra sporadic replenishments is set correctly, as `seL4_Yield` sleeps until the next available replenishment. If fragmentation occurs due to preemption and incorrect sporadic replenishments, the polling thread will not wake at the start of every period. This could be ameliorated by checking the time on wakeup, or using a different task model.

```
for (;;) {
  seL4_Word badge;
  seL4_Poll(notification_object, &badge);
  doJob();
  // sleep until next job is ready
  seL4_Yield();
}
```

Listing 6.1: Example of a polling task on seL4.

```
for (;;) {
  doJob();
  // sleep until next job is ready
  seL4_Word badge;
  seL4_Wait(notification_object, &badge);
}
```

Listing 6.2: Example of a basic sporadic task on seL4.

Sporadic threads

Recall that sporadic threads wake due to events such as an interrupts, then process the event and go back to sleep. To constrain sporadic thread's execution times, they are given a maximum inter-arrival time. To build sporadic threads notifications can be used with scheduling contexts, as displayed in Listing 6.2. In this case, if the number of sporadic replenishments is not sufficient, the processing bandwidth may be artificially limited by excessive preemptions, otherwise it will be $\frac{C}{T}$.

Time triggered threads

Time-triggered threads, which wake up, do some processing, and go to sleep again until the next period can be set up with a user-level timer driver, in order to wake at exactly the right time. Scheduling contexts can be combined with this approach to rate-limit time-triggered threads, but the exact amount of preemptions does not need to be known. Listing 6.2 also applies to this policy, where the notification object is configured to receive periodic notifications from a timer service.

6.4.2 Resource sharing policies

PIP/BWI

Fig. 6.6 shows an example of a resource server (S_2) which proxies IPC messages via another server (S_1) to manipulate priorities according to some protocol before and after requests. The policy works as follows: clients are given a badged capability to an endpoint, E_1, on which to make requests on the resource server. S_1 receives requests, identifies the client via the badge, and manipulates S_2's priority before forwarding the request to S_2 over endpoint E_2. S_1 returns to wait on E_1, while S_2 processes the request. If while S_2 is processing a

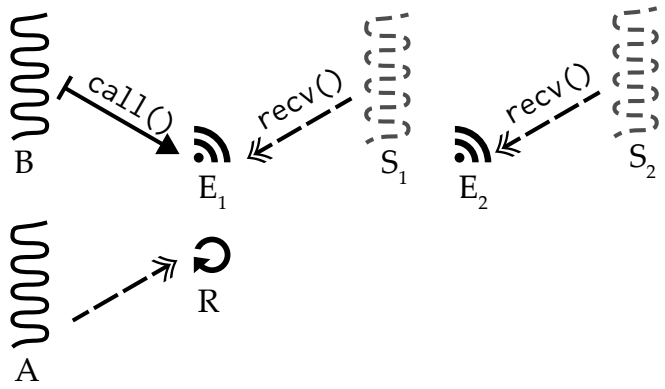


Figure 6.6: Example of proxying requests via a server that manipulates priorities before resource access, see Fig. 6.1 for the Legend.

request from A, and a higher priority thread B preempts it and sends a request on E_1, S_2 can bump S_1's priority again, and forward B's request to S_2, once again returning to wait on E_1 for further messages. When A's request is complete, S_2 enqueues a request on E_1, such that S_1 can readjust the priority of S_2 according to the protocol, and reply to the original caller. Note that for this method to work, S_1 must be the highest priority, and acts as a critical section for manipulating priorities in the system.

Our description so far can be used to implement the PIP or the OPCP, depending on the logic used by the proxy-server S_1. This functions whether S_1 and S_2 are active or not. In order to extend this protocol to bandwidth inheritance, a timeout fault handler would need to be specified for both server threads, which could then bind the pending client's scheduling context to the server to finish the request.

Best-effort Shared Server

Best-effort systems have no timing requirements, so each thread in the system has a full SC with a budget and period the length of the timeslice. Threads are scheduled round-robin. To implement such a system with our mechanisms, all server threads would be configured as active, and set to the ceiling priority of the clients.

This policy does not provide temporal isolation, as the server executes on its own budget. If one client launches a denial of service attack on the server, depleting the server's budget, then other clients are starved.

While our resource sharing mechanisms do not rule out this policy, it is only suitable for systems where all clients are trusted and the amount of requests each client makes is known *a priori*, or systems that have low temporal sensitivity.

Migrating threads

COMPOSITE [Parmer, 2010] solves the resource sharing model by using migrating threads (also termed stack-migrating IPC). On every IPC, client execution contexts (and CPU reservation) transfer to a new stack running in the server's protection domain, resulting in multi-threaded servers.

To implement migrating threads, COMPOSITE requires that every server have a mechanism for allocating stacks. If no memory is available to allocate stacks, then the request is blocked. This solution forces servers to be multi-threaded, and does not solve the problem of a client's budget expiring while the server is in a critical section, which is solved by providing atomic primitives that call the kernel.

A stack-spawning policy can be implemented using the mechanisms we have provided, similar to the PIP implementation described above (Section 6.4.2), except the proxy server spawns new worker threads instead of manipulating priorities. The stack-spawning proxy would also be at a lower priority than the worker threads, such that it would only be activated when all worker threads are busy.

An alternative policy is to build systems with a fixed number of server threads, all waiting on the same endpoint. Like the best-effort policy, our model supports this approach but it is not required.

Blocking Servers

Some servers need to block in order to function: for example, servers providing I/O may need to wait until an operation completes before replying to the client. Consider a disk server, where clients make requests of the server to write data to the disk. The server makes the request to the disk, but does not reply to the client until the device has signalled that I/O is complete. Rather than blocking until the driver responds, preventing the server from taking on further requests, the server can block on the endpoint and wait for the interrupt or further RPC requests. If an interrupt comes in, the server runs the bound notification object's scheduling context to complete the request.

Another option is to use multiple threads, with a passive thread for handling requests and an active thread servicing interrupts, however this requires synchronisation between the threads within the server.

Server isolation

We now describe how timeout handlers can be used to implement server isolation and other policies, as previously introduced in Section 6.1.6.

For our example consider a server shared between clients of different criticality, time sensitivity, and trust, which provides an encryption service via advanced encryption standard (AES). Such a server processes data in blocks, and the WCET of a request is a function of the amount of data to be encrypted. Such a server can be constructed transactionally in

order to render it preemptable: each block processed is considered a transaction. The server atomically switches between two states each time it completes a block, effectively finishing one transaction and starting another. The state the server is working on is always dirty, the previous is clean.

By constructing the server in this way, a timeout handler can reset the server to the last clean state if the passive server raises a timeout exception due to the client's SC expiring. The timeout handler saves the server at a known checkpoint, and flips the server back to the known checkpoint. The handler can then reply to the client on the server's behalf, and return how far through the request the server completed by reading it out of the last clean state. Invoking the reply returns the scheduling context to the client, and the timeout handler blocks waiting for the next request.

Other options are also available to the timeout handler, which is important as not all server operations can be reset midway. The timeout handler can reply with an error to the client or not reply to the client at all, thereby preventing the client from making further requests. Alternatively, the timeout handler could bind a temporary scheduling context to the server to complete the request.

6.5 Summary

In this chapter we have outlined our model for providing temporal isolation via scheduling and resource sharing mechanisms. We introduce support for user-level admission tests via a new control capability, and add processor reservations to the kernel in the form of scheduling contexts. Scheduling contexts differ from prior work in that they are decoupled from priority, thereby avoiding priority inheritance on scheduling context donation. In addition, by providing the active versus passive server definition, we also maintain policy freedom.

Finally, we outlined existing models for integrating real-time resource policies over IPC and show how scheduling-context donation combined with passive servers can be used to create trusted servers with untrusted clients. Our model supports best-effort, migrating threads, or temporal isolation policies for resource sharing.

Timeout exceptions are provided which allow for user-defined enforcement policies, while the default mechanism maintains isolation by preventing threads from executing for more than the share of processing time represented by the scheduling contexts.

In the next section we will outline the implementation of our model in seL4.

7 Implementation in seL4

In the previous chapter we presented our mechanisms for temporal isolation and safe resource sharing in a high-assurance microkernel. Now we delve into the implementation details of scheduling contexts, scheduling context donation, passive servers, timeout fault handlers, and IPC forwarding in seL4. First we explore new kernel objects, followed by new and altered API system calls and invocations, and finally look at structural changes.

7.1 Objects

We add two new objects to the kernel, *scheduling context objects (SCOs)* and *resume objects*. Additionally, we modify the TCB object, although do not increase its total size. Finally, we modify the notification object to allow single-threaded, passive servers to receive signals in addition to IPC messages.

7.1.1 Resume objects

Resume objects, modelled after KeyKOS [Bomberger et al., 1992], are a new object type that generalises the "reply capabilities" of baseline seL4 introduced in Section 5.5.1. Recall that in baseline seL4, the receiver of the message (i.e. the server) receives the reply capability in a magic "reply slot" in its capability space. The server replies by invoking that capability. Resume objects remove the magic by explicitly representing the reply channel, and servers with multiple clients can dedicate a resume object per client. Instead of generating a one-shot reply capability in a reply slot on a `seL4_Call`, the operation populates a resume object, while `seL4_Recv` de-populates the resume object. Additionally, resume objects also provide more efficient support for stateful servers that handle concurrent client sessions, which we expand on further when we introduce the changed system-call application programming interface (API) in Section 7.2.

A resume object can be in the following three states:

- *idle* meaning it is not currently being used,
- *waiting* meaning it is being used by a thread blocked and waiting for a message,

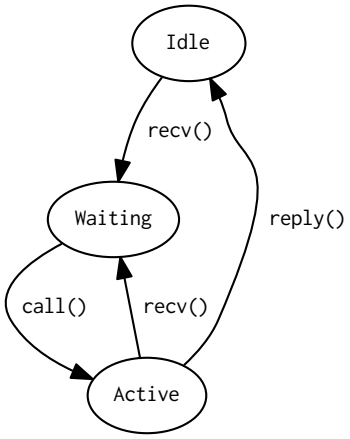


Figure 7.1: State diagram of resume objects.

- or *active*, which means the resume object currently has a thread blocked waiting for a reply associated with it.

Valid state transitions are shown in Fig. 7.1. Resume objects are now required as arguments to receiving system calls along with endpoint capabilities, which transitions them from idle to waiting. If the endpoint is invoked with `seL4_Call`, the caller is blocked on the resume capability, and it transitions to active. If a resume object is directly invoked, using a sending system call (recall from Section 5.2 that sending system calls are `seL4_Send`, `seL4_NBSend`, `seL4_Call`, `seL4_Reply`) then a reply message is delivered to the thread blocked on the object. Finally, if a resume object is in an active state, and provided to `seL4_Recv`, the object is first invoked, and removed from the call stack, which we now examine in detail.

The call stack

Active resume objects track the *call stack* that is built as nested `seL4_Call` operations take place. `seL4_Call` triggers a push operation, adding to the top of the stack, while a reply message triggers a pop, removing the top of the stack. The call stack allows us to track the path of a donated scheduling context, from caller to callee, so that it can be returned to the previous caller, regardless of which thread sends the reply message. This is a direct consequence of flexible resume capabilities: a resume capability can be moved between threads, and *any* thread can execute the reply: usually the server, but occasionally a timeout fault handler, or a nested server which received the resume capability via IPC forwarding. It is therefore impossible to derive the original callee from the thread which sent the reply message. When a reply message is sent via an active resume object at the head of the call

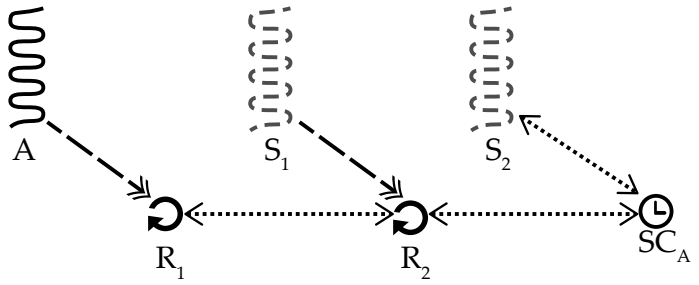


Figure 7.2: The state of the reply stack when a nested server S_2 is performing a request on behalf of an initial server S_1 for client A.

stack, that resume object is popped, and the SCO is linked to the thread that the resume object pointed to, overriding the previous SCO to TCB link.

The call stack is structured as a doubly-linked list, with one minor difference: the head of the call stack is the scheduling context that was donated along the stack, which itself contains a pointer to the thread currently executing on it. Each resume object then forms a node in the stack, going back to the original caller at the base. The SCO remains the head of the stack until the SCO returns to the initial caller and the stack is fully dismantled. When a reply message is sent, the scheduling context travels back along the call stack and the head resume object is popped. Reply objects also point to the thread which donated the SCO along the stack, allowing the SCO to be returned to that thread when a reply message is sent. This process is illustrated in Fig. 7.2, where A has called S_1 which has called S_2.

In a capability system, one of the biggest challenges is that capabilities can be revoked, and the object they grant access to deleted, at any time, regardless of the state of the system. Therefore we provide the following deletion semantics for resume objects:

- If the SCO is deleted, the head of the call stack becomes the first resume object. To avoid a long-running operation, the call stack of resume objects remains linked, and is dismantled as each resume object is deleted.
- If the head resume object (the object that points to the SCO) is deleted, the SCO is returned along the stack to the caller.
- If a resume object in the middle of the stack is deleted, we use the standard operation for removing a node from a doubly-linked list: the previous node is connected to the next node, and the deleted object is no longer pointed to by any member of the stack.
- If the resume object is at the start of the stack, i.e. the node corresponding to the initial caller, it is simply removed. The SCO cannot return to the initial caller by means of a reply message.

Resume objects are small (16 bytes on 32-bit platforms, and 32 bytes on 64-bit platforms), and contain the fields shown in Table 7.1.

Field	*Description*
`tcb`	The calling or waiting thread that is blocked on this reply object.
`prev`	`NULL` if this is the start of the call stack, otherwise points to the previous reply object in the call stack.
`next`	Either a pointer to the scheduling context that was last donated using this reply object, if this reply object is the head of a call stack (the last caller before the server) or a pointer to the next reply object in the stack. 0 if no scheduling context was passed along the stack.

Table 7.1: Fields in a resume object.

7.1.2 Scheduling context objects

We introduce a new kernel object type, SCOs, which all processing time is accounted against, and a new scheduler invariant: any thread in the scheduler queues must have an SCO. SCOs are variable-sized objects that represent access to a certain amount of time and consist of a core amount of fields, and a circular buffer to track the consumption of time. Scheduling contexts encapsulate processor time reservations, derived from sporadic task parameters: minimum inter-arrival time (T) and a set of replenishments which is populated from an original execution budget (C), representing the reserved rate ($U = \frac{C}{T}$). Fields in an SCOs are shown in Table 7.2.

Replenishments

In addition to the core fields, SCOs contain a variable amount of *replenishments*, which consist of a `timestamp` and `amount`. These are used for both round-robin and sporadic threads.

Field	*Description*
`Period`	The replenishment period, T.
`Consumed`	The amount of cycles consumed since the last reset.
`Core`	The ID of the processor this scheduling context grants time on.
`TCB`	A pointer to the thread (if any) that this scheduling context is currently providing time to.
`Reply`	A pointer to the head resume object (if any) in a call stack.
`Notification`	A pointer to the notification object (if any) to which this SCO is bound.
`Badge`	An unforgeable identifier for this scheduling context, which is delivered as part of a timeout fault message to identify the faulting client.
`YieldFrom`	Pointer to a TCB that has performed a directed yield to this SCO.
`Head,Tail`	Indexes to the circular buffer of sporadic replenishments.
`Max`	Size of the circular buffer of replenishments for this scheduling context.

Table 7.2: Fields of a scheduling context object.

0x00	period		consumed	
0x10	core	TCB	reply	ntfn
0x20	badge	yieldFrom	head	tail
0x30	max			
0x40	$\texttt{replenishment}_0$			
0x50	$\texttt{replenishment}_1$			
...	...			
0xF0	$\texttt{replenishment}_n$			

Table 7.3: Layout of a scheduling context object on a 32-bit system.

For round-robin threads we simply use the head replenishment to track how much time is left in that SCO's timeslice.

Sporadic threads are more complex; the replenishments form a circular buffer used to track how much time a thread can execute for (`amount`) and when that time can be used (`timestamp`). The size of the circular buffer is limited by both a variable provided on configuration of the SCO, and the size of the SCO (where the former must be *leq* the latter). This allows system designers to control the preemption and fragmentation of sporadic replenishments as discussed in Section 6.1.5.

Each SCO is minimum 2^8 bytes in size, which can hold eight or ten replenishments for 32 and 64-bit processors respectively. This is sufficient for most uses, but more replenishments can be supported by larger sizes, allowing system designers to trade-off latency for fragmentation overhead in the sporadic server implementation. SCOs can be created as larger objects, 2^n bytes, then the rest of the object can be filled with replenishments, as shown in Table 7.3. System designers can then use the `max` replenishment field to specify the exact number of replenishment to use, up to the object size.

Admission

Like any seL4 object, scheduling contexts are created from zeroed, untyped memory. Consequently, new scheduling objects do not grant authority to any processing time at all, as the parameters are all set to zero. To configure a scheduling context, the new `seL4_SchedControl` capability must be invoked, which allows the period, initial sporadic replenishment, maximum number of refills, and badge fields to be set. The processing core is derived from the `seL4_SchedControl` capability that is invoked: only one exists per core. The `seL4_SchedControl_Configure` operation behaves differently depending on the state of the target scheduling context, and can be used not only to configure SCOs but also to migrate threads across cores, and change the available bandwidth of a currently runnable thread.

```
sc->period = period;
sc->head = 0;
sc->tail = 0;
sc->max = max_refills;
HEAD(sc).amount = budget;
HEAD(sc).time = current_time;
```

Listing 7.1: `refill_new` routine to initialise a scheduling context that is not active or is migrating cores. `HEAD` is short-hand for access the head of the circular buffer of replenishments.

```
/* truncate to size 1 */
INDEX(sc, 0) = HEAD(sc);
sc->head = 0;
sc->tail = sc->head;
sc->max = new_max_refills;
sc->period = new_period;

if (HEAD(sc).time <= (current_time + kernel_wcet))
  HEAD(sc).time = current_time;

if (HEAD(sc).amount >= new_budget) {
  HEAD(sc).amount = new_budget;
} else {
  /* schedule excess amount */
  sc->tail = NEXT(sc, new);
  TAIL(sc).amount = (new_budget - HEAD(sc).amount);
  TAIL(sc).time = HEAD(sc).rTime + new_period;
}
```

Listing 7.2: `refill_update` routine to change the parameters of an active scheduling context. `INDEX`, `TAIL`, and `NEXT` are operations on the circular buffer.

Semantics are as follows:

- If the scheduling context is not bound to a currently runnable thread, or is empty—in that the parameters are set to 0—the operation is a basic configure: the fields are simply set and the first replenishment is configured with the budget provided and the timestamp of the kernel entry (see Listing 7.1).
- If the scheduling context is bound to a currently runnable thread, but that scheduling context is for a different processing core than the `seL4_SchedControl` capability, the fields are set and the thread is migrated to the new core.
- Finally, if the scheduling context is bound to a currently running thread and the `seL4_SchedControl` capability is for the same core, we update the replenishment list without allowing the sliding window constraint to be violated. Listing 7.2 shows the algorithm used.

Sporadic Servers

Recall that polling servers (Section 3.1.3) provide a specified bandwidth to a thread, but once a thread wakes, if it blocks or is preempted, the budget is abandoned until the next

```
if (new.amount < MIN_BUDGET && sc->head != sc->tail) {
  /* used amount is to small - merge with last and delay */
  TAIL(sc).amount += new.amount;
  TAIL(sc).time = MAX(new.time, TAIL(sc).time);
} else if (new.time <= TAIL(sc).time) {
  TAIL(sc).amount += new.amount;
} else {
  sc->tail = NEXT(sc, sc->tail);
  TAIL(sc) = new;
}
```

Listing 7.3: `schedule_used` routine, used below.

```
if (capacity == 0) {
  while (HEAD(sc).amount <= usage) {
    /* exhaust and schedule replenishment */
    usage -= HEAD(sc).amount;
    if (sc->head == sc->tail) {
      /* update in place */
      HEAD(sc).time += sc->period;
    } else {
      refill_t old_head = POP(sc);
      old_head.time = old_head.time + sc->period;
      schedule_used(sc, old_head);
    }
  }

  /* budget overrun */
  if (usage > 0) {
    /* budget reduced when calculating capacity */
    /* due to overrun delay next replenishment */
    HEAD(sc).time += usage;
    /* merge front two replenishments if times overlap */
    if (sc->head != sc->tail &&
        HEAD(sc).time + HEAD(sc).amount >=
        INDEX(sc, NEXT(sc, sc->head)).time) {
      refill_t refill = POP(sc);
      HEAD(sc).amount += refill.amount;
      HEAD(sc).time = refill.time;
    }
  }
}
capacity = MAX(HEAD.amount - usage, 0);
if (capacity > 0 && refill_ready(sc))
  split_check(sc, usage);
/* ensure the refill head is sufficient to run */
while (HEAD(sc).amount < MIN_BUDGET || sc.head == sc.tail) {
  HEAD(sc).amount += POP(sc).amount;
}
```

Listing 7.4: `check_budget` routine used to implement sporadic servers.

```
/* first deal with the remaining budget of the current replenishment */
ticks_t remnant = HEAD(sc).amount - usage;
refill_t new = (refill_t) {
  .amount = usage, .time = HEAD(sc).time + sc->scPeriod
};
if (SIZE(sc) == sc->max || remnant < MIN_BUDGET) {
  /* merge remnant with next replenishment - either it's too small
   * or we're out of space */
  if (sc->head == sc->tail) {
    /* update inplace */
    new.amount += remnant;
    HEAD(sc) = new;
  } else {
    POP(sc);
    HEAD(sc).amount += remnant;
    schedule_used(sc, new);
  }
} else {
  /* split the head refill  */
  HEAD(sc).amount = remnant;
  HEAD(sc).time += usage;
  schedule_used(sc, new);
}
```

Listing 7.5: `split_check` routine used to implement sporadic servers.

period. Sporadic servers(Section 3.1.3) also limit threads to a specified execution bandwidth, but do so by tracking a buffer of replenishments, which is appended to each time a thread is preempted or blocks. Sporadic servers with only one replenishment act like polling servers. As a result, configured scheduling contexts with zero extra replenishments behave like polling servers, otherwise they behave as sporadic servers, allowing application developers to tune the behaviour of threads depending on their preemption levels and execution durations.

The algorithms for managing replenishments are taken from Danish et al. [2011], with adjustments to support periods of 0 (for round robin threads) and to implement a minimum budget. Whenever the current scheduling context is changed, `check_budget` (Listing 7.4) is called to bill the amount of time consumed since the last scheduling context change. If the budget is not completely consumed by `check_budget`, `split_budget` (Listing 7.5) is called to schedule the subsequent refill for the chunk of time just consumed. If the replenishment buffer is full, or the amount consumed is less than the minimum budget, the amount used is merged into the next replenishment. When a new scheduling context is switched to, `unblock_check` (Listing 7.6) is used, which merges any replenishments that are already available, avoiding unnecessary preemptions.

The full code for sporadic servers, with less brevity than the simplified samples included here, is available in Appendix B.1.

```
/* first deal with the remaining budget of the current replenishment */
ticks_t remnant = HEAD(sc).amount - usage;
refill_t new = (refill_t) {
  .amount = usage, .time = HEAD(sc).time + sc->scPeriod
};
if (SIZE(sc) == sc->max || remnant < MIN_BUDGET) {
  /* merge remnant with next replenishment - either it's too small
   * or we're out of space */
  if (sc->head == sc->tail) {
    /* update inplace */
    new.amount += remnant;
    HEAD(sc) = new;
  } else {
    POP(sc);
    HEAD(sc).amount += remnant;
    schedule_used(sc, new);
  }
} else {
  /* split the head refill  */
  HEAD(sc).amount = remnant;
  HEAD(sc).time += usage;
  schedule_used(sc, new);
}
```

Listing 7.6: `unblock_check` routine used to implement sporadic servers.

Field	*Description*
~~`timeslice`~~	Used to track the timeslice, replaced by the scheduling context.
`scheduling context`	The scheduling context (if any) this TCB consumes time from.
`MCP`	The MCP of this TCB.
`reply`	A pointer to the resume object this TCB is blocked on, if the TCB is `BlockedOnReply` or `BlockedOnRecv`.
`yieldTo`	Pointer to the SCO this TCB has performed a directed yield to, if any.
~~`faultEndpoint`~~	Moved to the TCB `cnode`.

Table 7.4: Added and removed fields in a TCB object.

7.1.3 Thread control blocks

Recall from Section 5.3.3 that TCBs are the abstraction of an execution context in seL4, which are formed from a TCB data structure and a special `cnode` containing capabilities specific to that thread, which is only accessible to the kernel. We make several alterations to this structure, but do not impact the TCB size, as there was enough space available. The altered fields are shown in Table 7.4.

Fault endpoints

We also change the contents of the TCB `cnode`, removing two slots previously required by the reply capability implementation, as resume objects are now provided to receiving

system calls, and no slot in the TCB cnode is required. In addition, we add two new slots by installing fault handler capabilities, for faults and timeouts, into the TCB cnode. This is an optimisation for fault handling capabilities: in baseline seL4, the fault-handling capability is looked up on every single fault, increasing the overhead of fault handling. To minimise this overhead, we look up the fault endpoint when it is configured, and copy the capability into the TCB cnode. Note that capabilities in the TCB cnode cannot be changed by user-level: the TCB itself must be deleted for the endpoint capability to be fully revoked and the object deleted, which avoids a time-of-check time-of-use vulnerability.

Maximum controlled priorities

The MCP, or *maximum controlled priority*, resurrects a concept from early L4 kernels [Liedtke, 1996]. It supports lightweight, limited manipulation of thread priorities, useful e.g. for implementing user-level thread packages. When setting the priority or MCP of a TCB, A, the caller must provide the capability to a TCB, B, (which could be the caller's TCB). The caller is allowed to set the priority or MCP of A up to the value of B's MCP.[1] In a typical system, most threads will run with an MCP of zero and have no access to TCB capabilities with a higher MCP, meaning they cannot raise any thread's priority. The MCP is taken from an explicitly-provided TCB, rather than the caller's, to avoid the confused deputy problem [Hardy, 1988].

7.1.4 Notification objects

A scheduling context object can be associated with a notification object, which allows a passive server with bound notifications to receive signals and run on the notification's scheduling context to process those signals. We add a pointer to the scheduling context from the notification context, as well as vice-versa, to facilitate this mechanism, which increases the size of the notification object from 2^4 to 2^5 on 32-bit, and 2^5 to 2^6 on 64-bit.

7.2 MCS API

We now present the new mixed criticality system (MCS) API for seL4, which alters the core system call API, as well as adding new invocations and modifying existing ones. In this section we also present the semantics of scheduling context donation, which are directly linked to the new API.

7.2.1 Waiting system calls

In Section 5.2 we divided the seL4 system call API into two classes: sending system calls (seL4_Send, seL4_NBSend, seL4_Call, seL4_Reply), receiving system calls (seL4_Recv, seL4_NBRecv), plus combinations of both for fast RPC (seL4_Call, seL4_ReplyRecv).

[1] Obviously, this operation also requires a capability to A's TCB.

Our implementation alters the meaning of a receiving system call, and adds a new class of system call: *waiting* system calls. The difference is simple: receiving system calls are expected to provide a capability to a resume object, which can be used if a `seL4_Call` is received over the endpoint being blocked on. Waiting system calls do not, and cannot be paired with a `seL4_Call` successfully. Only receiving system calls can be used with scheduling context donation, as the reply object is used to track the call stack. Such a distinction existed in the original L4 model, where and were referred to as open and closed receive.

Additionally, because the TCB reply capability slot is dropped the TCB `cnode`, we remove the `seL4_Reply` system call, as its only purpose was to invoke the capability in the reply slot, which no longer exists. `seL4_ReplyRecv` remains, and invokes the resume capability, sending the reply, before using the reply in the `seL4_Recv` phase.

By making the reply channel explicit, we significantly increase the practicality of the IPC fastpath. In baseline seL4, any server that served multiple clients and did not reply immediately—because it needed to block on I/O, for example—would save the reply capability, moving it from the special TCB slot to a standard slot in the `cspace`. Later, when the server replied to the client, it would invoke the saved reply capability with `seL4_Send`, then block for new messages with `seL4_Recv`, avoiding the IPC fastpath for `seL4_ReplyRecv`. No system call to move the resume capability back to the TCB `cnode` was provided, however this would require yet another system call. Resume objects avoid the save slowpath, and increase the probability of using the fastpath, should the conditions detailed in Section 5.5.2 be satisfied.

7.2.2 IPC Forwarding

For sending system calls, only the system calls that combine a send- and receive-phase can donate a scheduling context, as a thread must be blocked in order to receive a scheduling context. Both `seL4_Call` and `seL4_ReplyRecv` combine a `seL4_Send` and `seL4_Recv` phase, although with slightly different semantics with respect to resume objects. However, `seL4_Call` conducts both phases on the same capability, while the send phase of `seL4_ReplyRecv` can only act on a resume object. We introduce a new combined system call, `seL4_NBSendRecv`, which is the mechanism for IPC forwarding (Section 6.3.2). IPC forwarding allows a `seL4_NBSend` followed by a `seL4_Recv` on two distinct capabilities, without the restriction that the send-phase must act on a resume capability. Additionally, we provide a variant which combines a send- and a waiting-phase, `seL4_NBSendWait`. Both variants allow for IPC forwarding, and can donate a scheduling context along with the IPC, by-passing the call chain.

The first phase of `seL4_NBSendRecv` and `seL4_NBSendWait` must use an `seL4_NBSend` rather than a `seL4_Send` to avoid introducing a long-running operation in the kernel in the form of a chain reaction triggered by a single thread blocking. We demonstrate by example why the send-phase must be non-blocking.

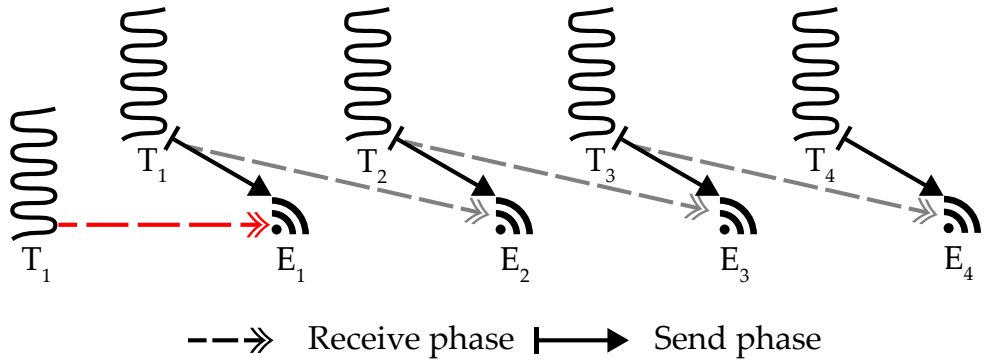


Figure 7.3: The problem with `seL4_SendRecv`.

To illustrate the problem, we refer to Fig. 7.3, which shows four threads, T_z and T_1, T_2, T_3 and four endpoints, $E_1, \ldots, E_4$. Black arrows have already occurred, blocking each thread on an endpoint, and grey are pending once each. T_1 has used `seL4_SendRecv` on E_1 and E_2, T_2 has used `seL4_SendRecv` on E_2 and E_3, T_3 has used `seL4_SendRecv` on E_3 and E_4. Because the send-phase is blocking, each thread T_n is blocked waiting for the send to proceed on E_n. T_z then uses `seL4_Recv` on E_1 (the red arrow), which triggers an unbounded chain of message sending: T_1 finishes its send phase and begins the receive phase on E_2, which allows T_2 to finish its send phase and being the receive phase on E_3, which allows T_3 to finish its send phase and start the receive phase on E_4. This chain reaction is an unbounded, long-running operation. By making the send-phase non-blocking such a chain reaction is not possible: the send-phase is aborted as the endpoint has no threads waiting for a message on it, and the receive phase then proceeds.

Yield

In baseline seL4, `seL4_Yield` triggers a reschedule, and appends the calling thread to the end of the appropriate scheduling queue. We retain these semantics for full scheduling contexts, however for partial, sporadic scheduling contexts `seL4_Yield` depletes the head sporadic replenishment immediately. The caller is then blocked until the next replenishment becomes available, and then placed at the end of the appropriate scheduling queue.

We also provide a new invocation on scheduling context capabilities, `seL4_SchedContext_YieldTo`, which allows for a directed yield to a specific SCO. When called on a SCO with a TCB running at the same priority as the caller, `seL4_SchedContext_YieldTo` results in that thread being switched to by the kernel scheduler. Otherwise the TCB is moved to the head of the scheduling queue for its priority.

Table 7.5 summarises the new MCS system call API, while more detailed descriptions of each system call can be found in Appendix A.

System Call	*Class*	*Donation?*	*New?*	*Definition*
seL4_Call	sending, receiving	✓	✗	Appendix A.1.1
seL4_Send	sending	✗	✗	Appendix A.1.2
seL4_NBSend	sending	✗	✗	Appendix A.1.3
seL4_Recv	receiving	✓	✗	Appendix A.1.4
seL4_NBRecv	receiving	✓	✗	Appendix A.1.5
seL4_Wait	waiting	✗	✓	Appendix A.1.6
seL4_NBWait	waiting	✗	✓	Appendix A.1.7
seL4_ReplyRecv	sending, receiving	✓	✗	Appendix A.1.8
seL4_NBSendRecv	sending, receiving	✓	✓	Appendix A.1.9
seL4_NBSendWait	sending, waiting	✗	✓	Appendix A.1.10
seL4_Yield	scheduling	✗	✗	Appendix A.1.11
~~seL4_Reply~~	~~sending~~	-	-	Removed

Table 7.5: New seL4 system call summary, indicating which system calls can trigger donation. and/or receiving messages.

7.2.3 Invocations

Scheduling contexts have five invocations, listed in Table 7.6. Three of those invocations are for binding SCOs to TCB and notification objects, while seL4_SchedContext_YieldTo and seL4_SchedContext_Consumed facilitate user-level scheduling, allowing the user to manipulate the kernel scheduling queues and to query the amount of time consumed by a specific SCO. Both functions return the amount of CPU time the scheduling context has consumed since it was last cleared: reading the consumed value by either system call clears it, as does a timeout exception.

The new control capability, seL4_SchedControl, has only one invocation, seL4_SchedControl_Configure, for setting the parameters of a scheduling context (the function definition can be found in Appendix A.2.8).

As baseline seL4 only supports sending three capabilities in a single message (including to the kernel), we add a second method for configuring multiple fields on a TCB in one system call (seL4_TCB_SetSchedParams), in addition to the existing method (seL4_TCB_Configure). We avoid altering the existing limit to avoid extensive re-verification of a non-critical path. Full configuration of a TCB requires up to six capabilities to be set, not including timeout handlers, which are not included in a combined configuration call as this is a less common operation. In total, we alter two invocations on TCB objects and add three new invocations, shown in Table 7.7.

Finally, we remove the seL4_CNode_SaveCaller invocation, which was previously used to move a reply capability from the TCB cnode to a slot in the TCB cspace, and is no longer required as the resume object capability is now provided to the receiving system call.

Invocation	*Description*	*Definition*
`seL4_SchedContext_Bind`	Bind an object (TCB or Notification) to an SCO.	Appendix A.2.2
`seL4_SchedContext_Unbind`	Unbind all objects from an SCO.	Appendix A.2.3
`seL4_SchedContext_UnbindObject`	Unbind a specific object from an SCO.	Appendix A.2.4
`seL4_SchedContext_Consumed`	Return the amount of time since the last timeout fault, `seL4_SchedContext_Consumed` or `seL4_SchedContext_YieldTo` was called.	Appendix A.2.5
`seL4_SchedContext_YieldTo`	Place the thread bound to this SCO at the front of its priority queue and return any time consumed.	Appendix A.2.6

Table 7.6: Scheduling context capability invocations.

Invocation	*Description*	*New?*	*Definition*
`seL4_TCB_Configure`	Set the cnode, vspace, and IPC buffer.	✗	Appendix A.2.10
`seL4_TCB_SetMCPriority`	Set the MCP.	✓	Appendix A.2.11
`seL4_TCB_SetPriority`	Set the priority.	✗	Appendix A.2.12
`seL4_TCB_SetSchedParams`	Set SCO, MCP, priority and fault endpoint.	✓	Appendix A.2.13
`seL4_TCB_SetTimeoutEP`	Set the timeout endpoint.	✓	Appendix A.2.14
~~`seL4_TCB_SetAffinity`~~	Removed, now derived from SCO.	✗	N/A

Table 7.7: New and altered TCB capability invocations.

7.2.4 Scheduling context donation

Conceptually there are three types of scheduling context donation: lending, returning, and forwarding a scheduling context. The type of donation that occurs, and if it can occur, depends on the system call used.

Lending a scheduling context occurs between caller and callee on a `seL4_Call`, and the semantics are simple: if a thread blocked on an endpoint does not have a scheduling context, the scheduling context is transferred from sender (caller) to receiver (the callee), and the resume object is pushed onto the call-stack, such that the SCO can be returned. If the callee already has a scheduling context, scheduling context donation does not occur. Note that lending also works for handling faults, so thread fault handlers can be passive. Unlike previous versions of timeslice donation in L4 kernels, there is no explicit flag for

permitting donation: if the callee does not have a scheduling context, the caller would block indefinitely, as the callee has no other way to access processing time. As discussed in Section 6.2, the caller in an RPC-style operation must trust the callee: regardless of the scheduling context being executed upon, the callee can always choose to not reply, blocking the caller indefinitely.

Lent scheduling contexts are returned when the resume object is invoked, or the head resume object is deleted. Resume objects can be invoked directly by the send phase of a system call, or indirectly, by using the resume object in the receive phase of a system call, thereby clearing it for another client to block on, or deleting the resume object. Regardless of the state of the callee, the SCO is returned, which if done incorrectly (via a `seL4_Send`), can leave the callee not blocked on an endpoint, and consequently unable to receive further scheduling contexts via donation or lending. Correct usage by a passive thread is with `seL4_ReplyRecv`, `seL4_NBSendRecv`, blocking the passive thread such that it is ready to receive another scheduling context. Like lending, scheduling contexts can also be returned by a fault handler replying to a fault message.

Forwarding a scheduling context does not establish a call-chain relationship, and allows for scheduling contexts to be transferred between threads in non-RPC patterns. The semantics are as follows: if IPC forwarding is used via `seL4_NBSendRecv` (Section 7.2.2) is used, and the receiver does not have a scheduling context, the scheduling context is forwarded.

Scheduling contexts can only be donated and received on specific system calls, indicated in Table 7.5.

Passive servers

Passive servers do not have scheduling contexts, yet they must be waiting on an endpoint in order to receive SCOs over IPC, and possibly initialise some server-specific state first. The protocol for initialising passive servers is to start them as active, then wait for the server to signal to the initialising thread that they are ready to be converted to passive. IPC forwarding can be used to do this, so the passive server can send a message on one endpoint or notification object, then wait on another to receive SCOs over IPC. Example user-level code for this process is provided in Listing 7.7.

Notification Binding

Passive servers can receive scheduling contexts from their bound notification object, allowing for the construction of single-threaded servers which receive both notifications and IPC messages. The semantics are as follows: if a TCB receives a notification from its bound notification object, and that TCB does not have a scheduling context, the TCB receives the scheduling context. If a TCB blocks on an endpoint, and is running on its bound notification object, the TCB is rendered passive again.

```c
void initialiser(seL4_CPtr server_tcb, seL4_CPtr init_sc, seL4_CPtr init_endpoint) {
  /* start server */
  seL4_TCB_Resume(server_tcb);
  /* wait for the server to tell us it is initialised */
  seL4_Wait(init_endpoint, NULL);
  /* convert the server to passive */
  seL4_SchedContext_Unbind(init_sc);
}

void passive_server(seL4_CPtr init_endpoint, seL4_CPtr endpoint, seL4_CPtr reply) {
  seL4_MessageInfo_t info = init_server_state();
  seL4_Word badge;

  /* signal to the initialiser that this server is ready to be converted to passive, and
  ↪  block on the endpoint with resume object reply */
  info = seL4_NBSendRecv(init_endpoint, info, endpoint, &badge, reply);
  while (true) {
    /* when the server wakes, it is running on a client scheduling context */
    info = process_request(sender);
    /* reply to the client and block on endpoint, with resume object reply */
    seL4_ReplyRecv(endpoint, info, &badge, reply);
  }
}

void client(seL4_CPtr endpoint, seL4_MessageInfo_t message) {
  /* send a message to the passive server */
  seL4_Call(endpoint, message);
}
```

Listing 7.7: Example initialiser, passive server, and client.

7.3 Data Structures and Algorithms

Now we discuss changes to the data structures and algorithms added and changed in the kernel to provide accounting, which charges CPU time to the correct SCO, and the enforcement mechanisms, which requires a new scheduling data structure and fault type.

7.3.1 Accounting

All processing time is accounted to the scheduling context of the currently running TCB. In this section we first establish *how* that time is accounted, and then *when* it is accounted.

There are two ways to account for time in an OS kernel:

- in fixed time quanta, referred to as *ticks*,
- in actual time passed between events, referred to as *tickless*.

Using ticks, timer interrupts are set for a periodic tick and are handled even if no kernel operation is required, incurring a preemption overhead. This approach has the advantage of simplicity: the timer implementation in the kernel is stateless and the timer driver can be set once, periodically, never requiring reprogramming. In older-generation hardware, especially x86, reprogramming the timer device incurred a significant cost, enough to ameliorate the preemption overhead. However, this design is not without limitations: the precision of the

scheduler is reduced to the length of the tick. Precision must be traded for preemption overhead: reducing the tick length increases precision, but also increases preemption overhead. Preemption overhead is particularly problematic for real-time systems, as the WCET each real-time task is inflated by the WCET of the kernel for every preemption.

Tickless kernels remove this trade-off by setting timer interrupts for the exact time of the next event. As we will show in Section 8.2.1, the cost of reprogramming the timer device on modern hardware is smaller, rendering the tickless design feasible. Consequently, we convert seL4 to a tickless kernel, a non-trivial change due to the fact that the kernel is non-preemptible[2], which means timer interrupts for the scheduler can only be serviced when the kernel is not running. Allowing a thread to complete an operation in the kernel when it does not have sufficient time to do so would violate temporal isolation, as the bandwidth allocated to the scheduling context could then be exceeded. As a result, on kernel entry, the calling thread must have sufficient budget to complete the operation. Without further decoding a pending system call, the kernel cannot tell which operation a thread is attempting, so sufficient budget becomes the WCET of the kernel.

Threads are not permitted in the scheduler if they have insufficient budget, as if the scheduling algorithm must iterate until a thread with sufficient budget is discovered, the complexity of the algorithm becomes $O(n)$ in the number of threads. The same condition holds for IPC endpoint queues: we take the time on kernel entry in order to charge a preempting thread, and any time from kernel entry is accounted to the scheduling context active at kernel exit. Because a passive scheduling context then must be able to be charged not only for the entry path into the kernel, but the exit path, the sufficient budget is actually *twice* the WCET of the kernel.

We check budget sufficiency on kernel entry, by reading the current timestamp and storing the amount of time consumed since the last kernel entry, allowing us to detect budget expiry in a single point, and carry on other kernel operations with this assumption intact. This means that timeout faults can only be raised at once point in the kernel, greatly simplifying the implementation by avoiding excessive checks in other paths.

Fastpath

We alter the IPC fastpath conditions introduced in Section 5.5.2 to include one more condition to have a very low overhead on fastpath performance:

- The receiver must be passive. For `seL4_Call`, this means the callee must be passive, and for `seL4_ReplyRecv`, the TCB blocked on the resume object must be passive.

This is because although timer device access is cheaper on modern hardware, it is not free. It also allows us to avoid checking budgets and modifying sporadic replenishments on the fastpath, operations which have many conditional branches, which would also increase overheads. Note that we expect most servers in real systems to use passive servers for RPC,

[2]Save for explicit preemption points in a few long-running operations.

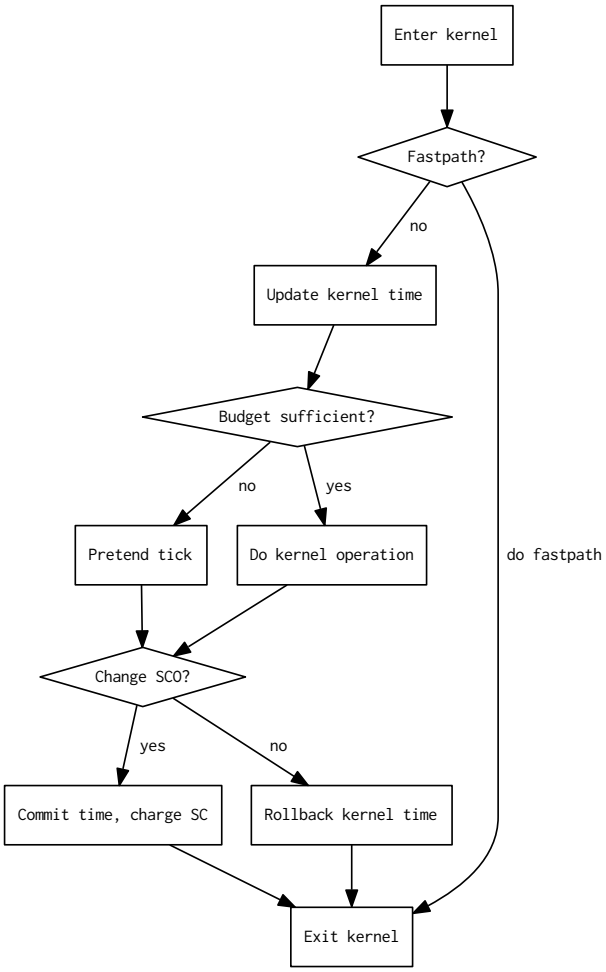


Figure 7.4: New tickless kernel structure.

even for non real-time systems, to assure fairness of resource access between round-robin threads.

Even on the slowpath, we avoid reprogramming the timer unless it is required. Like reading the time, reprogramming the timer is cheaper, but not free. Consequently although we update the current time on every entry, we only charge threads at specific points:

- when the timer interrupt has fired, so clearly the timer needs reprogramming,
- when the current scheduling context changes, so a different interrupt is required,
- when the kernel timestamp has been acted upon: for instance, the used to schedule a replenishment.

If the recorded timestamp is not acted upon, the time is rolled back to the previously recorded value, thus avoiding reprogramming the timer unnecessarily.

Fig. 7.4 illustrates the new control flow of the tickless kernel, and shows how and when processing time is charged to scheduling contexts. On kernel entry, if the available budget is insufficient, the kernel pretends the timer has already fired, resets the budget and adds the thread to the release queue. If the entry was due to a system call, the thread will retry that call once it wakes with further budget. Once the thread is awoken, it will retry the system call.

Domain scheduler

We retain the domain scheduler which, as discussed in Section 5.6.2, is required for the proof of confidentiality [Murray et al., 2013]. However, we convert the implementation to tickless, such that domains are configured with CPU cycles to execute, rather than fixed ticks, which no longer exist.

7.3.2 Enforcement

The enforcement mechanisms require the most significant changes to the kernel, which we now detail. The main change required to the existing scheduler is the addition of a *release queue* per processing core which contains all runnable threads waiting for a replenishment. If a preempted thread does not have any available replenishments, the kernel removes the thread from the ready queue.

Priority queues

In addition to the release queue, we also change endpoint- and notification-object queues to priority queues, ordered by TCB priority. All queues are implemented as ordered, doubly-linked lists, where all operations complete in $O(1)$ except for insertion, which is $O(n)$, where n is the number of threads in the queue. We chose the list data structure over a heap for increased performance and reduced verification burden.

A list-based priority queue out-performs a heap-based priority queue for small n (in our implementation up to around $n = 100$). This n is larger than one would expect in a traditional OS, where heap implementations are array-based in contiguous memory with layouts optimised for cache usage. However, because seL4 kernel memory is managed at user-level (as discussed in Section 5.3), data structures must be dynamic, resulting in much higher code complexity, and poorer performance than a static heap. For endpoint queues, it is reasonable to expect a low number of threads, as many threads queuing on an endpoint at once indicates a poor system design. For the release queue, the restriction becomes a function of the user-level admission test: only scheduling contexts with partial budgets are placed into the release queue, which limits the size of the queue. Consequently we do not expect any of the priority queues to contain large numbers of threads, however if scalability becomes a problem in practice this implementation could be reconsidered. Brandenburg

[2011] used binomial heaps for schedulers in LITMUS^{RT}, however Linux does not provide the guarantees, or implementation requirements of seL4.

One security concern is sub-systems with access to a single scheduling context and a large amount of memory can increase the kernel's run-time by queuing up a large amount of passive threads on an endpoint. To prevent this, system designers must not hand out large untyped capabilities to untrusted sub-systems.

Timeout Exception Handlers

Timeout exceptions are only triggered for threads with non-empty timeout handling slots in the TCB `cnode`, configured with `seL4_TCB_SetTimeoutEP`. The implementation is the same as for fault handlers, with one exception: while fault IPC messages can trigger scheduling context donation, allowing fault-handling threads to be passive, timeout messages cannot. The timeout message contains the badge of the scheduling context which triggered the fault, and the amount of time consumed by that scheduling context since the last reset.

7.4 Summary

In this chapter we have presented the new kernel objects, system call API, invocations and structural changes made to provide mechanisms for temporal isolation. The implementation adds roughly 2,000 lines of code to seL4 (14% increase), as measured by source lines of code (SLOC) [Wheeler, 2001] on the pre-processed code for the SABRE, the verified platform. In the next chapter, we evaluate our implementation with a series of microbenchmarks, system benchmarks and case studies.

8 Evaluation

We now evaluate our model through a series of microbenchmarks, system benchmarks and case studies. First, we present the ARM and x86 hardware platforms used in addition to the cost of timer operations on those platforms. Then we demonstrate the low performance overhead of our mechanisms using a set of microbenchmarks which measure the overheads of the MCS kernel versus a baseline, which is the seL4 kernel, at development version 7.0.0 with some added experimental fastpaths for interrupt and signal handling. Both branches have the same merge-base, i.e. both branches divert from the master branch at the same point. The exact revisions used are available online at:

Baseline: https://github.com/pingerino/seL4/releases/tag/phd-baseline,

MCS: https://github.com/pingerino/seL4/releases/tag/phd-MCS.

We then run several system benchmarks, first comparing a Redis [RedisLabs, 2009] implementation to baseline seL4, Linux and NetBSD to show that our kernel is competitive. We then demonstrate temporal isolation between threads sharing the same CPU in two scenarios, using Redis again, followed by ipbench [Wienand and Macpherson, 2004]. Finally, we show isolation in a shared encryption server, and measure the cost of various timeout fault-handling policies.

In addition, we evaluate two user-level scheduler implementations: one which implements a static criticality switch, and a user-level EDF implementation which we show is competitive with an in-kernel EDF scheduling from LITMUS^{RT}. Lastly, we use an IPC throughput benchmark to demonstrate low, multicore-scalability overhead, and we show how resource server threads can migrate between processing cores.

8.1 Hardware

We run microbenchmarks on a variety of hardware to show the overheads of the model compared to baseline seL4. Table 8.1 summarises the hardware platforms. SABRE is the verified platform for seL4, in addition to x64, which at the time of writing verification is ongoing. Currently, the only platforms that support more than one core are SABRE, and X64.

Platform *Arch*	*Microarch.* *CPU*	*Clock* GHz	L1 (KiB)	L2 (KiB)	L3 (MiB)	TLB (entries)
KZM (32-bit)	ARM1136JF-S	1.0	16+16	128	✗	32
ARMv6	i.MX31		4×	8×	✗	2×
SABRE (32-bit)	Cortex-A9	1.0	32+32	1024	✗	64
ARMv7	i.MX6		4×	16 ×	✗	2×
HIKEY (64-bit)	Cortex-A53	1.2	32+32	512	✗	128
ARMv8	Kirin 620		4×+2×	16×	✗	4×
TX1 (64-bit)	Cortex-A57	1.9	32+48	2,048	✗	256
ARMv8	Jetson TX1		2×+3×	16×	✗	4×
X64 (64-bit)	i7-4770	3.1	32+32	256	8	128,8
x86	Haswell		8×	8×	16×	8×

Table 8.1: Hardware platform details, "×" is associativity, and + indicates I-cache+D-cache.

Two of our platforms, X64 and HIKEY support both 32- and 64-bit execution modes. We use both, referring to them as IA32 and HIKEY32 when in 32-bit mode and X64 and HIKEY64 otherwise. All platforms have out-of-order execution, except HIKEY, which is in-order, and KZM, which has in order execution and out-of-order from some operations (e.g. stores).

Additionally, we use several load generators running Linux on an isolated network for benchmarks that require a network.

8.2 Overheads

We first present a suite of microbenchmarks to evaluate any performance overheads against baseline seL4. Because the kernel uses *lazy switching* for the floating point unit (FPU), where the FPU context is only switched if it is actively being used, we ensure FPU context switching is off. This is achieved by performing the required number of system calls without activating the FPU. We present overheads on IPC operations, signalling and interrupts, and finally scheduling.

For each of the benchmarks in this section, we measure the cost of measurement, which is reading the cycle counter on ARM platforms, and the timestamp counter (TSC) on x86, and subtract the value obtained from the final result.

8.2.1 Timer

Two of the main sources of overhead introduced by our model are related to the need to read and reprogram the timer on non-fastpath kernel entries, and when performing a scheduling

Platform	*Timer*	*Read time*	*Set timeout*	*Sum*
KZM	General purpose timer	83 (0)	203(0)	286
SABRE	ARM MPCore global timer	23 (0)	36 (0)	59
HIKEY32/64	ARM generic timers	6 (0)	6 (0)	12
TX1	ARM generic timers	8 (0)	1 (0)	9
IA32	TSC deadline mode	12 (2.2)	220 (1.0)	232
X64	TSC deadline mode	11 (2.3)	217 (2.0)	228

Table 8.2: Latency of timer operations per platform in cycles. Standard deviations shown in parentheses.

context switch. We show the results of microbenchmarks of both of these operations in Table 8.2, and note the timer hardware used on the specific platform.

For both microbenchmarks, we read the timestamp before and after the operation, and do this 102 times, discarding the first two results to prime the cache. We take the difference of the cycle counts, and subtract the cost of measuring the cycle counter itself. The results show the cost of both operations separately, and then their sum, which is the total measured overhead introduced by timers on scheduling context switch.

All platforms excluding KZM have a 64-bit timer available, making KZM the only platform requiring timer overflow interrupts, which are not measured as KZM is a deprecated platform provided for comparison with modern ARM versions.

KZM and SABRE both use memory-mapped timers, the 32-bit general purpose timer for the former and 64-bit ARM global timer for the latter. SABRE has four cores and while the timer device is shared, the timer registers are per-core, making access fast. Timer access on SABRE is significantly faster than the KZM.

For all other ARM platforms, the ARM generic timers are available, which are accessed via the coprocessor. The majority of new ARM platforms support the ARM generic timers.

On X64 we use the TSC with TSC-deadline mode [Intel 64 & IA-32 ASDM], an architectural model-specific register (MSR) available since Intel SandyBridge by which a local-APIC timer interrupt is triggered when the TSC matches the programmed value.

Table 8.2 shows the instruction latency of each timer operation. In practice, especially for X64, these operations are subject to pipeline parallelism and out-of-order execution, which reduces the overhead.

Results on both architectures show that the overhead of a tickless kernel, which requires the timers to be frequently read and reprogrammed, is tolerable on modern hardware. On ARM, timer costs have reduced by an order of magnitude from ARMv6 through to ARMv8.

Platform	*Baseline*		*MCS*		*Overhead*	
KZM	263	(0)	290	(0)	27	10%
SABRE	288	(0)	371	(58)	83	28%
HIKEY32	235	(2)	251	(3)	16	6%
HIKEY64	251	(3)	277	(4)	26	10%
TX1	398	(7)	424	(1)	26	6%
IA32	440	(2)	422	(2)	-18	-4%
X64	448	(1)	456	(2)	8	1%

Table 8.3: Time in cycles for the `seL4_Call` fastpath. Standard deviations shown in parentheses.

8.2.2 IPC performance

IPC performance is a critical measure of the practicality and efficiency of a microkernel [Liedtke, 1995]. We benchmark our IPC operations against base system, seL4, which has an established efficient IPC fastpath [Elphinstone and Heiser, 2013].

Fastpath

To evaluate IPC fastpath performance, we set up a client (the caller) and server (the callee) in different address spaces. We take timestamps on either side of the IPC operation being benchmarked and record the difference. This is done 16 times for each result value to prime the cache, then record the next value. Results presented are for performing this a total of 16 times. Additionally, we measure the overhead of system calls stubs in the same way and subtract this from the measurement, to obtain only the kernel cost of the operation[1]. The message sent is zero length, so neither the caller nor callee's IPC buffer is accessed.

As discussed in Section 7.3.1, timer operations are avoided on the fastpath. However, we do add significantly to the `seL4_Call` fastpath, by accessing two further objects (the scheduling context and resume object) and two validation checks. In detail, the `seL4_Call` fastpath is altered as follows:

- We remove the reply capability lookup, so the sending TCB's `cnode` is no longer accessed.
- We read the resume object pointer from the thread state, however the thread state is already accessed by the `seL4_Call` fastpath.
- We check that the resume object is valid.
- We read from and write to the resume object to push it onto the call stack.
- We check the destination TCB is passive.

[1]The IPC benchmarks already existed for seL4, but we modify them to support the MCS kernel as part of this thesis.

Platform	*Baseline*		*MCS*		*Overhead*	
KZM	304	(0)	350	(1)	46	15%
SABRE	312	(1)	334	(2)	22	7%
HIKEY32	252	(4)	275	(0)	23	9%
HIKEY64	266	(5)	303	(5)	37	13%
TX1	392	(2)	424	(6)	32	8%
IA32	412	(1)	447	(2)	35	8%
X64	432	(1)	449	(2)	17	3%

Table 8.4: Time in cycles for the `seL4_ReplyRecv` fastpath. Standard deviations shown in parentheses.

- We write scheduling context that is lent over the `seL4_Call` to link it to the receiver, and update the back pointer to the resume object.

Table 8.3 shows the results. Overheads on the `seL4_Call` fastpath are low, with the worst effected platform being KZM with a 10% overhead, which is the oldest platform with the smallest caches. On call, KZM shows L1 data cache miss and 1 memory access, which is likely an unlucky cache-conflict. SABRE suffers a 7% overhead, due to a translation lookaside buffer (TLB) miss likely due to the relatively low associativity on Cortex-A9 TLBs. In general, as hardware becomes newer the overhead is lower, with the exception of the HIKEY, which is the only platform we use that has in-order execution. Additionally HIKEY has a smaller L2 cache than the older SABRE, but a larger TLB wit higher associativity. We posit that for the Hikey, further hand optimisation of the fastpath would result in better performance. The newest ARM hardware is the TX1, which only shows a 5% overhead. x86 with its large caches and highly optimised pipeline only incurs a 1% overhead. For further detail on the numbers read from the performance counters cited here, please see Appendix C.

The fastpath for `seL4_ReplyRecv` has a higher impact, for two reasons: first, we add a new capability lookup, being the resume object provided to receive, which must be looked up by the `seL4_ReplyRecv` fastpath. Although we remove of reply capability from the TCB `cnode`, that `cnode` is still accessed to validate the caller's root `cnode` capability and top-level page directory capability, so the cache-foot print is not reduced. Additionally, because IPC is now priority ordered, the fastpath must check if the callee's TCB will be appended to the endpoint queue. Otherwise, the fastpath is aborted for an ordered insert.

As a result, the overheads shown in Table 8.4 for `seL4_ReplyRecv` are generally higher than `seL4_Call`, except for SABRE, which already suffered some TLB misses on the baseline kernel, again due to low TLB associativity.

	KZM	SABRE	HIKEY32	HIKEY64	TX1	IA32	X64
Baseline	2908	1808	1938	1645	1554	910	769
Overhead (%)	81%	41%	61%	43%	17%	31%	31%
Cycle count	+2346	+745	+1186	+705	+257	+283	+240
L1 I-miss	-10	+0	+15	+11	+0	+0	+0
L1 D-miss	+10	+0	+0	-3	-4	+0	+0
TLB I-miss	+0	+1	+0	+0	+0	+0	+0
TLB D-miss	+0	+11	+0	+0	+0	+0	+0
Instructions	+1193	+688	+1122	+719	+726	+809	+721
Branch mispredict	+43	+5	-1	+1	+0	+0	+0
Memory access	+14	+0	+462	+307	+312	+0	+0

Table 8.5: Passive slowpath round-trip IPC overhead.

Slowpath

Now we evaluate the IPC slowpath, for both active and passive threads in the same address space. Unlike the fastpath, the slowpath generally does not fit into the cache. Consequently, precise measurements like those done for the fastpath show erratic results with high standard deviations on our hardware. Instead, we run average benchmarks for the slowpath, where for 110 runs, we measure the time taken for 10,000 IPC pairs (`seL4_Call`, `seL4_ReplyRecv`) and divide by 10,000 to get a result, abandoning the first 10 results. In order to hit the slowpath we make the message length 10 words, which exceeds the fastpath condition that the message should fit into registers.

We run the benchmark for both active and passive threads, which shows the overhead of changing a scheduling context versus not. Table 8.5 shows the results for the passive slowpath and Table 8.6 for active, where the top row in the table is the baseline number, the second row is the overhead when comparing the number of cycles taken for the benchmark. Every other row shows difference in value of the listed performance counter between baseline and MCS. We do not show standard deviations, however they are measured and are very small (either 0 or a few percent of the mean).

Passive slowpath overhead, as shown in Table 8.5 is not small; this is due to the fact that the seL4 scheduler has far more functionality than previously, increasing the code size, and the cache foot print is higher, given that the resume object and scheduling context must be accessed in addition to the TCBs of the caller and callee. Looking at ARM platforms, the overhead reduces drastically as hardware becomes more modern: for the most part, this means that the caches are bigger, and timer access costs are smaller. Except for the HIKEY platform, with has an in-order execution pipeline and a smaller L2 cache than the SABRE. We suspect that the overheads could be reduced on Hikey with more careful hand optimisation. 64-bit Hikey has nearly half the excess instructions when compared

	KZM	SABRE	HIKEY32	HIKEY64	TX1	IA32	X64
Baseline	2908	1808	1938	1644	1553	910	769
Overhead (%)	92%	80%	53%	36%	8%	75%	83%
Cycle count	+2689	+1455	+1030	+597	+122	+678	+639
L1 I-miss	+0	+0	+9	+10	+0	+0	+0
L1 D-miss	+11	+0	+0	-3	-4	+0	+0
TLB I-miss	+0	+0	+0	+0	+1	+0	+0
TLB D-miss	+0	+9	+0	+0	+0	+0	+0
Instructions	+1112	+599	+1022	+586	+593	+670	+593
Branch mispredict	+48	+7	-2	+0	-1	+0	+0
Memory access	+26	+0	+400	+268	+270	+0	+0

Table 8.6: Active slowpath round-trip IPC overhead.

to 32-bit, and far 50% less memory accesses, resulting in far less overhead on HIKEY64 than HIKEY32. SABRE suffers from the low-associativity of its TLB with misses in both the instruction and data TLBs. On the x86 platform, both IA32 and X64 show the same overhead in terms of ratio, both due to the increase in instructions.

Another factor that complicates the slowpath is the compiler. We use GCC version 5.5.0 with optimisation level O2 for all benchmarks, with the `mtune` parameter set as appropriate. All kernel code is pre-processed and concatenated into to single file with the `whole-program` attribute set which allows for greater compiler optimisations. However, the level of optimisation for a specific platform, especially ARM platforms, is not known and contributes to the variance between platforms, and between arm versions.

Additionally, it should be noted that the slowpath is avoided in most cases, as long IPC is discouraged in favour of establishing shared memory protocols to transfer large amounts of data. The majority of fastpath checks make sure the thread being switched to is valid and runnable. Only two cases hit the slowpath frequently: if a medium priority thread has woken, and is higher priority than the IPC target, and if the target is active, not passive. Additionally, many improvements can be made to the IPC slowpath to improve performance, but it has not been examined extensively.

Table 8.6 shows results for the active slowpath, and has higher overheads again. In this case, we change scheduling contexts, so not only does another object (the second SCO) need to be accessed, but the timer needs to be reprogrammed, and sporadic replenishment logic applied. TX1 and HIKEY64 have the lowest overhead, although the overhead looks to be ameliorated by lucky data placement with a reduction in L1 D-cache misses on both platforms. The vast majority of overhead comes from far more instructions executed, and on platforms with small caches or low-associativity, the increase in cache footprint.

	SABRE	HIKEY32	HIKEY64	TX1	IA32	X64
Baseline	1545	1603	1323	1338	1221	1226
Overhead (%)	52%	67%	44%	21%	25%	24%
Cycle count	+798	+1078	+575	+288	+303	+290
L1 I-miss	+0	+15	+2	+0	+0	+0
L1 D-miss	+0	+0	-1	-4	+0	+0
TLB I-miss	+0	+0	+0	+0	+0	+0
TLB D-miss	+13	+0	+0	+0	+0	+0
Instructions	+696	+1045	+651	+652	+704	+643
Branch mispredict	+8	-3	-1	+0	+0	+0
Memory access	+0	+394	+238	+252	+0	+0

Table 8.7: Passive fault round-trip IPC overhead.

	SABRE	HIKEY32	HIKEY64	TX1	IA32	X64
Baseline	1544	1604	1322	1339	1221	1226
Overhead (%)	70%	59%	34%	11%	57%	55%
Cycle count	+1085	+941	+451	+150	+693	+674
L1 I-miss	+0	+7	+2	+0	+0	+0
L1 D-miss	+0	+0	-1	-4	+0	+0
TLB I-miss	+0	+0	+0	-1	+0	+0
TLB D-miss	+11	+0	+0	+0	+0	+0
Instructions	+603	+942	+513	+516	+561	+513
Branch mispredict	+8	-1	-2	-1	+0	+0
Memory access	+0	+331	+196	+203	+0	+0

Table 8.8: Active fault round-trip IPC overhead.

8.2.3 Faults

Recall that fault handling in seL4 occurs via an IPC, simulated by the kernel, to a fault endpoint, which a fault handling-thread can wait upon for messages (Section 5.5.3). To measure the fault-handling cost, we run two threads in the same address space: a fault handler and a faulting thread, with the same priority. We trigger a fault by executing an undefined instruction in a loop on the faulting thread's side. The fault handler then increments the instruction pointer past the undefined instruction, and the benchmark continues. As this is also a slowpath, we use the same method as above, and measure the amount of time it takes for 10,000 faults then divide the result. We do this for 110 runs, abandoning the first 10, to calculate the standard deviation and average.

Platform	*Baseline*		*MCS*		*Overhead*	
KZM	485	(13)	1251	(24)	766	157%
SABRE	1616	(99)	1794	(82)	178	11%
HIKEY32	528	(6)	763	(2)	235	44%
HIKEY64	509	(30)	657	(21)	148	29%
TX1	767	(8)	815	(12)	48	6%
IA32	1062	(53)	1147	(55)	85	8%
X64	1108	(55)	1239	(53)	131	11%

Table 8.9: Fastpath IRQ overhead in cycles, baseline seL4 versus MCS. Standard deviations shown in parentheses.

Platform	*Baseline*		*MCS*		*Overhead*	
KZM	150	(0)	149	(0)	-1	0%
SABRE	134	(10)	140	(13)	6	4%
HIKEY32	118	(0)	118	(0)	0	0%
HIKEY64	198	(14)	197	(14)	-1	0%
TX1	267	(20)	269	(14)	2	0%
IA32	176	(1)	175	(6)	-1	0%
X64	129	(2)	132	(2)	3	2%

Table 8.10: Fastpath signal overhead in cycles, baseline seL4 versus MCS. Standard deviations shown in parentheses.

We measure both active and passive fault handling, and the results, shown in Table 8.7, are similar to slowpath IPC, being slower for active, and improving as cache-size and timer operation cost reduces. Note that although there is currently no fastpath for fault handling, it is merely a matter of engineering effort to add one should this become a performance issue.

We do not show results for the KZM platform, as it is ARMv6 and the performance monitor unit, including the cycle counter, cannot be read from user mode. As a result we use the undefined instruction to read the cycle counter efficiently on this platform, so the fault benchmark does not work.

8.2.4 Signalling and interrupts

As noted in the previous section, mainly engineering effort is required to add new fastpaths. For this reason, we add two new (non-verified) experimental fastpaths to the baseline and MCS kernels: one for interrupt delivery, and the other for signalling a low-priority thread, which we now examine.

	KZM	SABRE	HIKEY32	HIKEY64	TX1	IA32	X64
Baseline	628	473	435	421	481	254	182
Overhead (%)	187%	95%	89%	30%	9%	113%	145%
Cycle count	+1172	+451	+385	+128	+44	+286	+266
L1 I-miss	+1	+0	+0	+1	+0	+0	+0
L1 D-miss	+0	+0	+0	+0	+0	+0	+0
TLB I-miss	+0	+0	+0	+0	+0	+0	+0
TLB D-miss	+0	+5	+0	+0	+0	+0	+0
Instructions	+580	+215	+426	+129	+132	+207	+134
Branch mispredict	+25	+5	+1	-2	-1	+0	+0
Memory access	+7	+0	+121	+61	+61	+0	+0

Table 8.11: Yield to self costs. Baseline seL4 versus MCS. Standard deviations shown in parentheses.

We measure interrupt latency using two threads, one spinning in a loop updating a volatile cycle counter, the other, higher priority thread waiting for an interrupt. On delivery, the handler thread determines the interrupt latency by subtracting the looped timestamp from the current time. Table 8.9 shows the results. The overhead is higher here as we must switch scheduling contexts, which requires reprogramming the timer, however the scheduler is by-passed as the switch is to a higher priority thread. KZM shows the highest overhead, where once again we exceed the small cache. The HIKEY pays for its in-order execution pipeline, which could be improved with further profiling of the fastpath. The TX1 shows the least impact, with only a 5% increase on the interrupt fastpath.

`signal()`, on the other hand shows little overhead as it does not alter the scheduling context or access to the timer device. In this benchmark, a high priority thread signals a low priority thread, a common operation for interrupt service routines. As Table 8.9 shows, in cases where slowpath performance is an issue, further fastpaths can be added to avoid accessing the timer, in the case where older hardware is required.

8.2.5 Scheduling

In our final microbenchmark we look at the cost of the scheduler, and the `seL4_Yield` system call, both of which are slowpaths.

Table 8.11 shows the results of measuring the average cost for the current thread to `seL4_Yield` to itself, which in the current kernel code always triggers a reschedule. The overhead is large as `seL4_Yield` is nearly a completely different system call: previously `seL4_Yield` was incomplete, and would simply dequeue and enqueue the current thread from the scheduling queues, before returning to user level. `seL4_Yield` in the new model charges the remaining budget in the head replenishment to the thread, and triggers a timer reprogram. Although semantically similar for round robin threads, `seL4_Yield` does a lot

	KZM	SABRE	HIKEY32	HIKEY64	TX1	IA32	X64
Baseline	1056	703	707	653	637	406	349
Overhead (%)	91%	66%	58%	46%	27%	127%	155%
Cycle count	+957	+466	+409	+302	+169	+518	+543
L1 I-miss	+5	+0	+0	+6	+0	+0	+0
L1 D-miss	+1	+0	+0	+0	+0	+0	+0
TLB I-miss	+0	+2	+0	+0	+0	+0	+0
TLB D-miss	+0	+10	+0	+0	+0	+0	+0
Instructions	+340	+208	+317	+179	+184	+234	+168
Branch mispredict	+24	-3	+4	+2	+1	+0	-1
Memory access	+8	+0	+132	+88	+105	+0	+0

Table 8.12: Scheduler costs in cycles, baseline seL4 versus MCS. Standard deviations shown in parentheses.

more, hence the drastic overheads. ARM platforms show an large increase in instructions executed, and older ARM platforms experience some cache-misses. x86 platforms show a smaller increase in instructions executed, but recall from Table 8.2 that when run in a tight loop, the overhead of reading and reprogramming the timer is over 200 cycles.

The `schedule` benchmark measures the cost of two threads in different address spaces switching to each other in a loop by changing each other's priority. A high-priority thread increases the priority of a low-priority thread, subsequently the low-priority thread then sets its own priority back down, resulting in a reschedule to the high-priority thread. Both threads do this 50,000 times, and we measure the cost for 110 runs, abandoning the first 10, so the caches are primed. This results in 10,000 invocations of the scheduler, as both threads do the switching. Of course, setting the priority of a thread higher does not need to invoke the scheduler: but this is the current state of the source code[2], and allows us to run this benchmark.

Table 8.12 shows the results. On x86, the overhead can be attributed half to the increase in instructions and the rest to reading and reprogramming the timer. The overhead looks large on both IA32 and X64, however note the very small initial value. On ARM, we see a larger overhead that decreases in newer platforms.

The scheduler is by definition a slowpath activity, as it is completely avoided on the fastpath, and much of the slowpath, by the lazy scheduling mechanism (Section 5.6.1). Table 8.12 shows that scheduling cost increases noticeably, however note that seL4 IPC, particularly scheduler-context donation (and its predecessor, the undisciplined timeslice donation), is designed to minimise the need for invoking the scheduler, therefore this increase is unlikely to have a noticeable effect in practice.

[2]We have a patch to fix this which is waiting in the queue for verification.

8.2.6 Full system benchmark

To demonstrate the impact of the overheads in a real system scenario, we measure the performance of the Redis key value store [RedisLabs, 2009] using Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al., 2010] on baseline and MCS seL4, and compare this against Linux, the Rump unikernel [Kantee and Cormack, 2014] and NetBSD [The NetBSD Foundation, 2018] all on the x64 machine. Note that the Rump unikernel only currently supports x86 platforms, consequently experiments requiring Rump are only carried out on the x64 platform.

For seL4, we use a single-core Rump library OS [Kantee and Cormack, 2014] to provide NetBSD network drivers at user level, by leveraging an existing port of this infrastructure to seL4 [McLeod, 2016]. The system consists of Redis/Rump running on three active seL4 threads: two for servicing interrupts (network, timer) and one for Rump, as shown in Fig. 8.1. Interrupt threads run at the highest priority, followed by Redis and a low-priority idle thread (not shown) for measuring CPU utilisation; this setup forces frequent invocations of the scheduler and interrupt path.

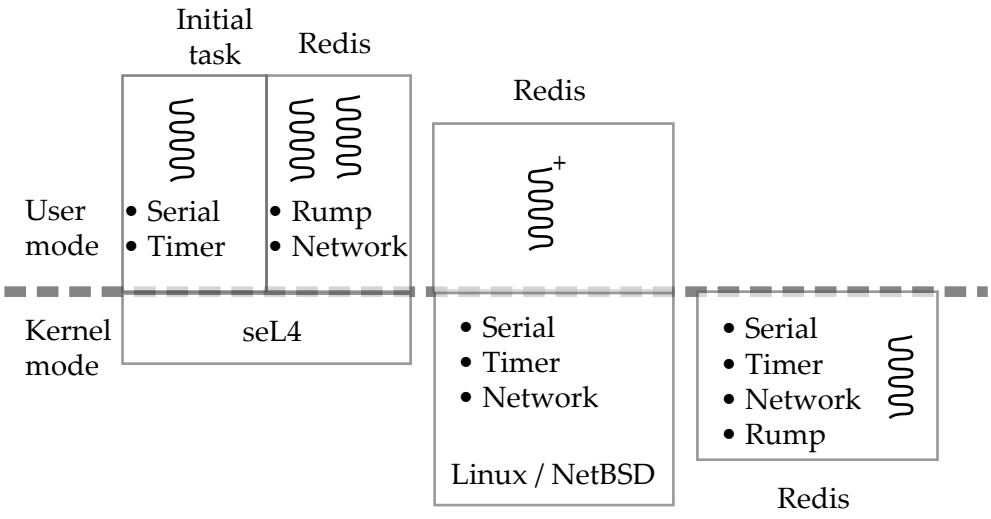


Figure 8.1: System architecture of the Redis / YCSB benchmark on seL4, Linux, NetBSD and Rump unikernel.

Table 8.13 shows the achieved throughput of Redis+Rump running bare metal kernel (BMK), and Redis on the seL4 baseline and as well as the MCS branch, plus Linux and NetBSD (7.0.2) for comparison. Fig. 8.1 shows the seL4 set up, compared with the architecture of Redis on Linux, NetBSD and BMK.

Table 8.13 indicates the interrupt handling method used, as there is no single method supported across all four scenarios. BMK only supports the legacy programmable interrupt controller (PIC), while NetBSD only supports message-signalled interruptss (MSIs). Linux and seL4 both support the advanced PIC (APIC).

System	*IRQ*	*Throughput* (k ops/s)	*Utilisation* (%)	*Cost* per op.	*Latency* (ms)
seL4-base	APIC	138.7 (0.4)	100	0.72	1.4
seL4-MCS	APIC	138.5 (0.3)	100	0.72	1.4
seL4-MCS	MSI	127.3 (0.6)	100	0.79	1.6
NetBSD	MSI	134.0 (0.2)	99	0.74	1.5
Linux	MSI	179.4 (0.4)	95	0.52	1.1
Linux	APIC	111.9 (0.4)	100	0.89	1.8
BMK	PIC	144.1 (0.2)	100	0.69	1.4

Table 8.13: Throughput (k ops/s) achieved by Redis using the YCSB workload A with 2 clients. Latency is the average Read and Update, standard deviations in parentheses and omitted where less than the least significant digit shown.

The utilisation figures show that the system is fully loaded, except in the Linux case, where there is a small amount of idle time. The cost per operation (utilisation over throughput) is best on Linux, a result of its highly optimised drivers and network stack. Our bare-metal and seL4-based setups use Rump's NetBSD drivers, and actually performance within a few percent of native NetBSD. This indicates that the MCS model comes with low overhead.

8.3 Temporal Isolation

We have demonstrated our model has little overhead and is competitive with existing monolithic kernels. Now we evaluate temporal isolation properties, between processes and in a shared-server scenario. In addition, we evaluate and demonstrate different techniques to restore server state after a timeout exception.

8.3.1 Process isolation

We evaluate process isolation, where processes do not share resources, indirectly via network throughput and network latency in two separate benchmarks.

Network throughput

First, we demonstrate our isolation properties with the Redis architecture described in Section 8.2.6, with an additional, high-priority active CPU-hog thread competing for CPU time. All scheduling contexts in the system are configured with a 5 ms period. We use the budget of the CPU-hog to control the amount of time left over for the server configuration. Figure 8.2 shows the throughput achieved by the YCSB-A workload as a function of the available CPU bandwidth (i.e the complement of the bandwidth granted to the CPU-hog thread). All data points are the average of three benchmark runs.

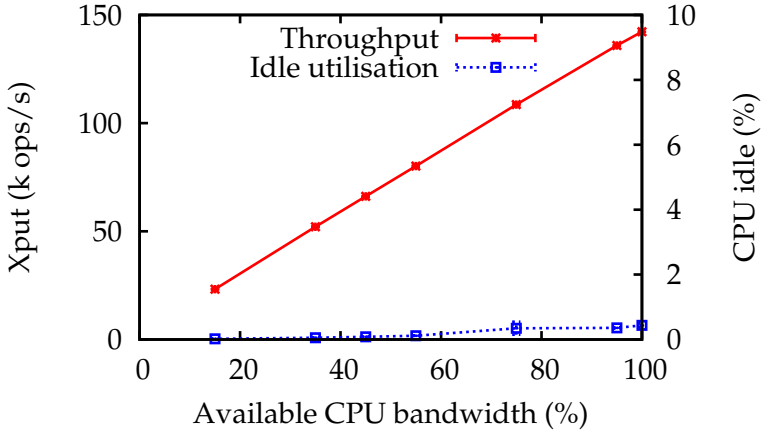


Figure 8.2: Throughput of Redis YCSB workload A and idle time versus available bandwidth.

The graph shows that the server is CPU limited (as indicated by very low idle time) and consequently throughput scales linearly with available CPU bandwidth.

Network latency

Second, we evaluate process isolation via network latency in a system shown in Fig. 8.3. The system consists of a single-core of a Linux virtual machine (VM) which runs at a high priority with a constrained budget and a User Datagram Protocol (UDP) echo server running at a lower priority, representing a lower-rate HIGH thread. We measure the average and maximum UDP latency reported by the ipbench [Wienand and Macpherson, 2004] latency test.

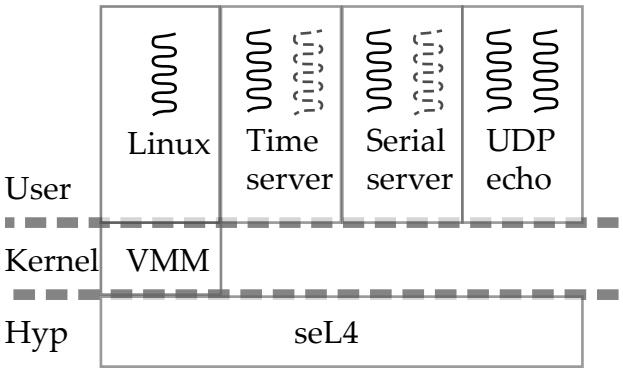


Figure 8.3: System architecture of ipbench benchmark.

Specifically, the Linux VM interacts with timer (PIT) and serial device drivers implemented as passive servers outside the VM; all three components are at a high priority. In the Linux server we run a program (`yes > /dev/null`) which consumes all available CPU bandwidth. The UDP echo server, completely isolated from the Linux instance during the benchmark, but sharing the serial driver, runs at a low priority with its own HPET timer driver.

Two client machines run ipbench daemons to send packets to the UDP-echo server on the target machine (X64). The control machine, one of the load generators, runs ipbench with a UDP socket at 10 Mbps over a 1 Gb/s Ethernet connection with 100-byte packets. The Linux VM has a 10 ms period and we vary the budget between 1 ms and 9 ms. We represent the zero-budget case by an unconstrained Linux that is not running any user code. Any time not consumed by Linux is available to UDP echo for processing 10,000 packets per second, or 100 packets in the time left over from each of Linux's 10 ms period.

Figure 8.4 shows the average and maximum UDP latencies for ten runs at each budget setting. We can see that the maximum latencies follow exactly the budget of the Linux server (black line) up to 9 ms. Only when Linux has a full budget (10 ms), and thus able to monopolise the processor, does the UDP server miss its deadlines, resulting in a latency spike. This result shows that our sporadic server implementation is effective in bounding interference of a high-priority process.

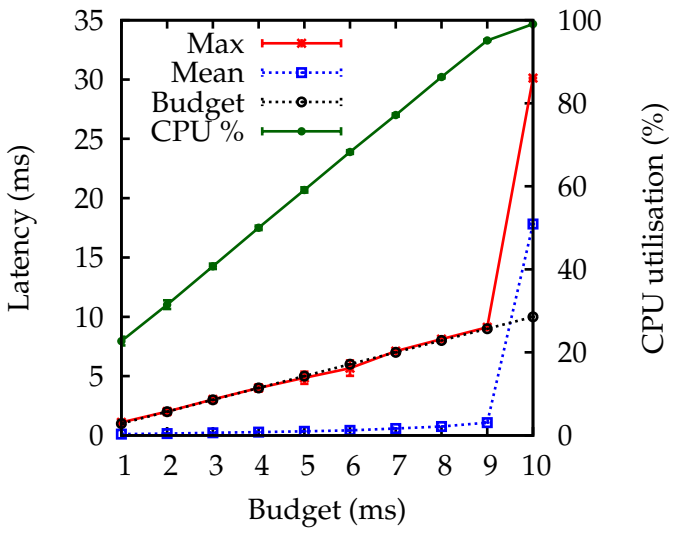


Figure 8.4: Average and maximum latency of UDP packets with a CPU-hog VM running a high priority with a 10 ms budget.

8.3.2 Server isolation

To demonstrate temporal isolation in a shared server, we use a case study of an encryption service using AES to encrypt client data. We measure both the overhead of different recovery techniques, and the throughput achieved when two clients constantly run out of budget in the server. We port an existing, open-source AES implementation to a shared server running on seL4, and run benchmarks on both X86 and SABRE.

Figure 8.5 shows the architecture of the case study. Both clients *A* and *B* are single threaded and exist in separate address spaces to the server. The server has two threads, a passive thread for serving requests on the client's scheduling context and an active thread

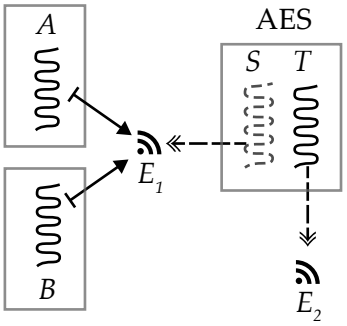


Figure 8.5: Architecture of the AES case study. Client *A* and *B* make IPC requests over endpoint E_1 of passive *AES* which has an active timeout fault handling thread waiting for fault IPC messages on endpoint E_2.

which handles timeout exceptions for the server. The server and timeout exception handler share the same virtual memory and capability spaces.

The server itself runs the AES-256 algorithm with a block size of 16 bytes. The server alternates between two buffers, using an atomic swap, of which one always contains consistent state, the other is dirty during processing. When a timeout fault occurs, only the dirty buffer is lost due to inconsistency. Both clients request 4 MiB of data to be encrypted, provided by a shared buffer between client and server, and have a budget insufficient to complete the request. When the server is running on the client's scheduling context and the budget expires, a timeout exception, which is a fault IPC message from the kernel, is delivered to the timeout handler.

8.3.3 Timeout handling techniques

If a client's scheduling context is depleted while the server is servicing the request, a timeout fault is raised and sent to the timeout handler. The appropriate protocol for handling such faults depends ultimately on the requirements of the system. Consequently, we implement and evaluate four different timeout fault handling techniques: *rollback*, *error*, *emergency* and *extend*.

While each technique is evaluated separately, combinations of such techniques could be used for different clients, depending on their requirements. For example, trusted clients may get special treatment. Note that in all cases of blocking IPC, clients must trust the server as discussed in Section 6.2. Additionally, although our experiment places the timeout fault handler in the server's address space, this is not necessary: for approaches that require access to scheduling contexts and scheduling control capabilities, the timeout handler may be placed in a scheduling server's address space, separate from the server itself.

Rollback

Rollback restores the server to the last known consistent state recorded. In the case of non-thread safe servers, this may require rolling back an entire request. However, algorithms like AES which can easily be batched can make progress. The process for rollback is as follows:

1. A timeout fault is received by timeout handler, T over E_2.
2. T constructs a reply message to the client from the last clean state, and sends this message to the client by invoking the resume capability that the server has used for that client. Because the resume object tracks the donation, the client's scheduling context is returned.
3. T then restores the server, S, to a known state (restoring registers, stack frames and any global state). S is restored to the point before it made its last system call, usually an `seL4_NBSendRecv`, which it used to indicate to the initialiser that S should be made passive, as part of the passive server initialisation protocol presented in Section 7.2.4.
4. Now the server must return to blocking on the IPC endpoint, E_1. T binds a scheduling context to S and waits for a message.
5. Now the S runs from its checkpoint, repeating the `seL4_NBSendRecv`, signalling to T that it can now be converted to passive once more and blocking on E_1.
6. T wakes and converts the server back to passive.
7. Finally, T blocks on E_2, ready for further timeout faults.

The rollback technique requires the server and timeout handler to both have access to the reply object that the server is using, and the server's TCB, meaning the timeout handler must be trusted by the server. In our example the server and timeout handler run at the same priority, in all cases both must run at higher priorities than the clients, to allow the timeout handler to reinitialise the server in the correct order, and to implement the IPCP.

Once the budget of the faulting client is replenished, it can then continue the request based on the content of the reply message sent by the timeout handler. Clients are guaranteed progress as long as their budget is sufficient to complete a single batch of data.

If rollback is not suitable, the server can be similarly reset to the initial state and an error returned to the client. However, this does not guarantee progress for clients with insufficient budgets.

Kill

In cases of non-preemptible servers, potentially due to a lack of thread safety, one option is to kill client threads. Such a scenario would stop untrusted misbehaving clients from constantly monopolising server time. We implement an example were the timeout handler

has access to client TCB capabilities and simply calls suspend, however the server could also switch to a new reply object and leave the client blocked forever, without access to any of the clients capabilities.

The process for suspending the client is the same as that for Section 8.3.3 but for two aspects; the server state does not need to be altered by the timeout handler as the server always restores to the same place, instead of replying to the client it is suspended.

Emergency

Another technique gives the server a one-off emergency budget to finish the client request, after which the exception handler resets the server to being passive. This could be used in low criticality SRT scenarios where isolation is desired but transient overruns are expected. An example emergency protocol follows:

1. A timeout fault is received by timeout handler, T over E_2.
2. T unbinds the client scheduling context from S.
3. T binds a new scheduling context to S, which acts as an emergency budget.
4. T then replies to the timeout fault, resuming S.
5. Now T enqueues itself on E_1, such that when the server finishes the blocked request, T, being a higher priority, is served next.
6. S completes the request and replies to the client.
7. S receives the request from T, and replies immediate as it is an empty request.
8. T wakes, and converts the server back to passive.
9. Finally T blocks on E_2, ready for further timeout faults.

This case requires the timeout handler to have access to clients' scheduling contexts, in order to unbind them from the server.

Extend

The final technique is to simply increase the client's budget on each timeout, which requires the timeout fault handler to have access to the client's scheduling contexts. This could be deployed in SRT systems or for specific threads with unknown budgets up to a limit.

1. A timeout fault is received by timeout handler, T over E_2.
2. T extends the client's budget by configuring scheduling context.
3. T replies to the fault message, which resumes S.

Platform	*Operation*	*Cache*	*Min*	*Max*	*Mean*	σ
X64	Rollback	hot	1.47	2.90	1.96	0.40
		cold	4.20	5.17	4.68	0.18
	Emergency	hot	0.95	1.75	1.20	0.14
		cold	2.34	2.95	2.49	0.11
	Extend	hot	0.16	0.80	0.29	0.14
		cold	0.92	1.42	1.05	0.08
	Kill	hot	1.36	1.45	1.39	0.01
		cold	4.09	4.74	4.32	0.17
SABRE	Rollback	hot	11.2	13.3	12.1	0.5
		cold	19.1	25.2	23.8	1.4
	Emergency	hot	4.4	5.6	4.9	0.2
		cold	13.1	14.5	13.9	0.3
	Extend	hot	0.7	2.9	1.9	0.4
		cold	6.9	8.0	7.3	0.3
	Kill	hot	10.0	12.5	11.1	0.5
		cold	21.9	23.2	22.6	0.3
HIKEY64	Rollback	hot	5.15	6.91	5.77	0.39
		cold	31.75	38.20	36.69	0.71
	Emergency	hot	4.45	7.53	5.55	0.56
		cold	19.31	21.02	20.22	0.35
	Extend	hot	0.65	1.63	0.90	0.21
		cold	10.69	11.90	11.28	0.28
	Kill	hot	5.17	7.07	6.07	0.44
		cold	32.24	34.43	33.14	0.43
TX1	Rollback	hot	2.24	2.66	2.38	0.10
		cold	7.04	8.36	8.07	0.14
	Emergency	hot	1.09	1.36	1.22	0.07
		cold	4.58	4.92	4.75	0.07
	Extend	hot	0.28	0.52	0.31	0.04
		cold	2.50	2.72	2.64	0.05
	Kill	hot	2.20	2.45	2.22	0.03
		cold	7.31	7.64	7.50	0.07

Table 8.14: Cost of timeout handler operations in μs, as measured by timeout exception handler. σ is the standard deviation.

8.3.4 Results

We measure the pure handling overhead in each case, from when the timeout handler wakes up to when it blocks again. Given the small amount of rollback state, this measures the

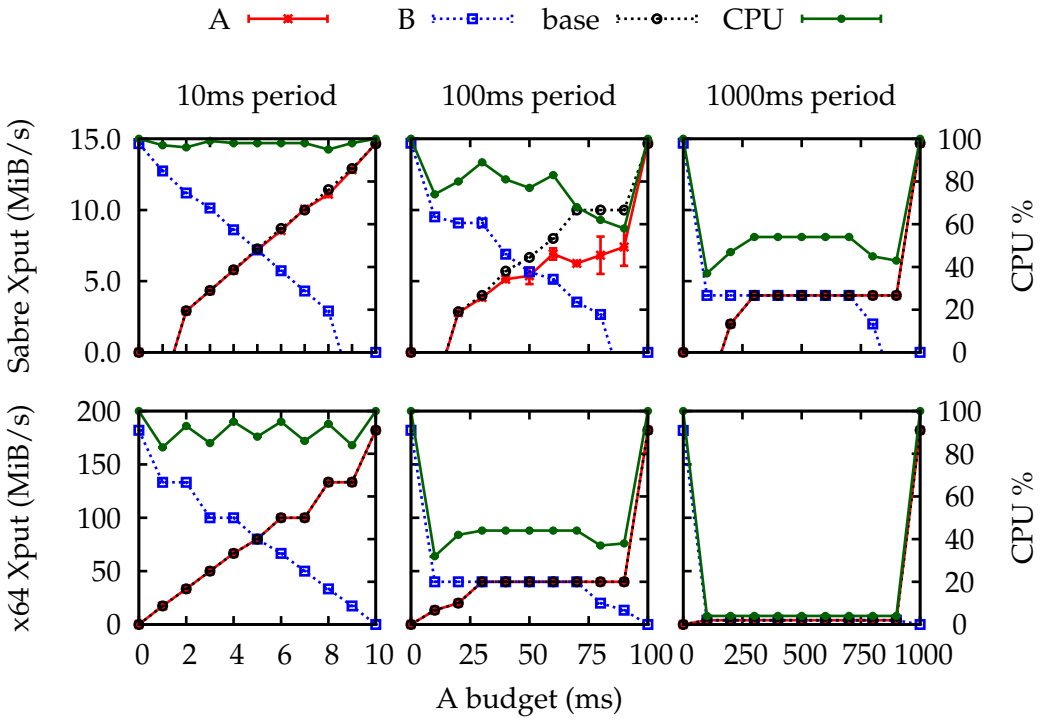


Figure 8.6: Throughput for clients A and B of a passive AES server processing 10 requests of 4 MiB of data with limited budgets on the x64 (top row) and SABRE (bottom row) platforms. The two clients' budgets add up to the period, which is varied between graphs (10, 100, 1000 ms). Clients sleep when they process each 4 MiB, until the next period, except when their budgets are full. Each data point is the average of 10 runs, error bars show the standard deviation.

baseline overhead. For schedulability analysis, the actual cost of the rollback would have to be added, in addition to the duration of the timeout fault IPC.

We run each benchmark with hot caches (primed by some warm-up iterations) as well as cold (flushed) caches and measure the latency of timeout handling, from the time the handler wakes up until it replies to the server.

Table 8.14 shows the results. The maximum cold-cache cost, which is relevant for schedulability analysis, differs by a factor of 3–4 between the different recovery scenarios, indicating that all are about equally feasible. Approaches that restart the server and send IPCs messages on its behalf (rollback, reply) are the most expensive as they must restore the server state from a checkpoint and follow the passive server initialisation protocol (recall Section 7.2.4).

8.3.5 Rollback isolation

We next demonstrate temporal isolation in the server by using the rollback technique and measuring the time taken to encrypt 10 requests of 4 MiB of data. Figure 8.6 shows the

result with both clients having the same period, which we vary between 10 ms and 1000 ms. In each graph we vary the clients' budgets between 0 and the period. The extreme ends are special, as one of the clients has a full budget and keeps invoking the server without ever getting rolled back, thus monopolising the processor. In all other cases, each client processes at most 4 MiB of data per period, and either succeeds (if the budget is sufficient) or is rolled back after processing less than 4 MiB.

The results show that in the CPU-limited cases (left graphs) we have the expected near perfect proportionality between throughput and budget (with slight wiggles due to the rollbacks), showing isolation between clients. In the cases where there is headspace (centre of the right graphs), both clients achieve their desired throughput.

8.4 Practicality

Fundamental to the microkernel philosophy is keeping policy out of the kernel as much as possible, and instead providing general mechanisms that allow the implementation of arbitrary policies [Heiser and Elphinstone, 2016]. As on the face of it, our fixed-priority-based model seems to violate this principle, we demonstrate that the model is general enough to support the efficient implementation of alternate policies at user level. Specifically, we implement two user-level schedulers: first, a static mixed criticality scheduler [Baruah et al., 2011b], which we also compare to an in-kernel implementation, and an EDF scheduler, which we compare to LITMUS^{RT}.

8.4.1 Criticality

Static, mixed-criticality, fixed-priority schedulers are based on *mode switches*, which effectively mean altering the priority of threads in bulk: critical threads that may be of low rate are bumped to higher than their low-criticality counter parts, to ensure deadlines are met in exceptional circumstances.

We implement a kernel mechanism for changing thread priorities in bulk, and compare with a user-level approach which simply changes the priority of threads one at a time. In our prototype implementation, the kernel tracks all threads of specific criticalities and boosts their priority on a criticality switch. However, given threads are kept in per-priority queues, each thread must be removed and reinserted into a new queue, so either way the complexity of the mode-switch is $O(n)$.

Figure 8.7 shows the results measured with a primed cache (hot) and flushed cache (cold). As the graph shows, switching is linear in the number of threads being boosted, in both kernel and user-level implementations. Table 8.15 compares the results of cold, in-kernel switching to user-level in roughly absolute terms. HIKEY64 is the slowest at 30 μs with eight threads to switch, due to the smaller L2 cache and in-order execution, however all platforms show roughly the same factor of two increase when comparing kernel and user-level cold-cache results. However, most systems will not have more than a few

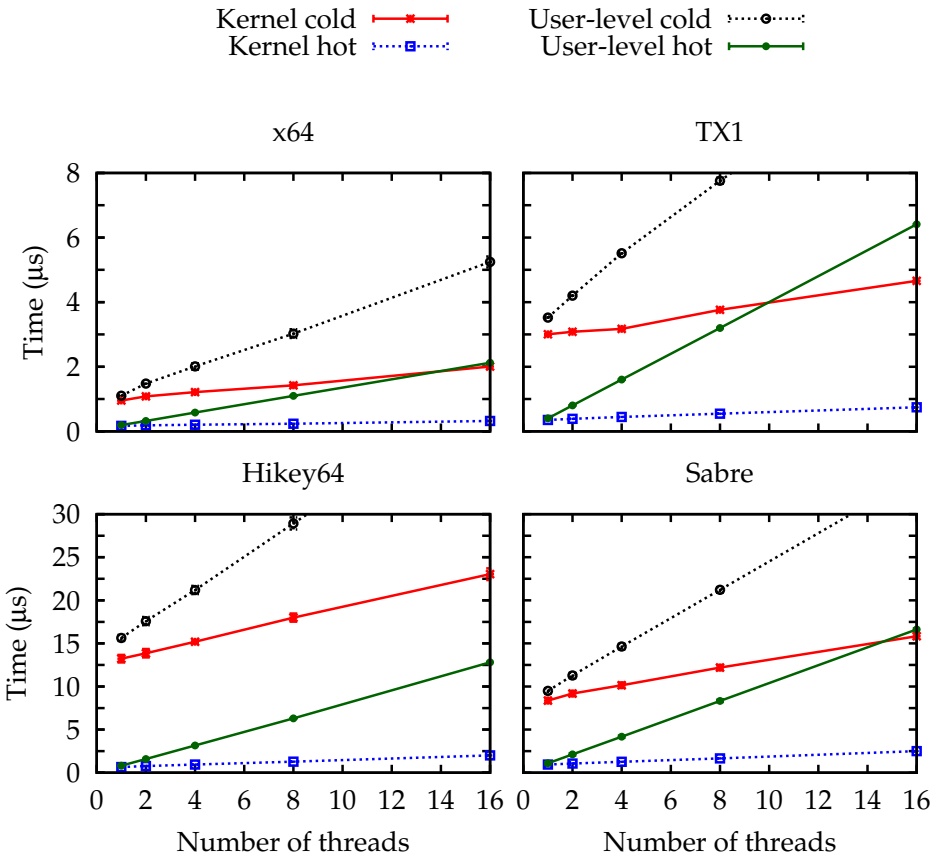


Figure 8.7: Cost of switching the priority of n threads in kernel and user level, with hot and cold caches, on X64 (top-left), TX1 (top-right), HIKEY64 (bottom-left) and SABRE (bottom-right). All data points are the average of 100 runs, with very small standard deviations.

Platform	*Kernel cold (µs)*	*User-level cold (µs)*	*Diff µs*
X64	≤ 2	≤ 4	~2
TX1	≤ 4	≤ 8	~4
HIKEY64	≤ 18	≤ 30	~12
SABRE	≤ 12	≤ 22	~10

Table 8.15: Comparison of cold, in-kernel priority switch to cold, user-level priority switch.

high-criticality threads, and deadlines for critical control loops in cyber-physical systems tend to be in the tens of milliseconds, we conclude that criticality can be implemented at user-level, in line with standard microkernel philosophy.

The higher cost from user-level operation results from multiple switches between kernel and user mode, and the repeated thread-capability look-ups. It could be significantly reduced if seL4 had a way to batch system calls, but to date we have seen no compelling use cases for this.

Application	T	L	L_S	C	U	j		m	
susan	190	2	0	25	0.13	111	(0.0)	0	(0.0)
Image recognition		2	1	51	0.27	111	(0.0)	0	(0.0)
		2	2	127	0.67	111	(0.0)	0	(0.0)
jpeg	100	1	0	15	0.15	200	(0.0)	0	(0.0)
JPEG encode/decode		1	1	41	0.41	200	(0.0)	0	(0.0)
		1	2	41	0.41	155	(0.0 **0.3**)	89	(0.9 **-0.2**)
madplay	112	0	0	28	0.25	179	(0.0)	0	(0.0)
MP3 player		0	1	28	0.25	178	(0.0)	48	(0.0)
		0	2	28	0.25	5 **1**	(0.3 **0.2**)	5 **1**	(0.3 **0.2**)

Table 8.16: Results of criticality switch benchmark for each stage, where the criticality L_S is raised each stage. T = period, C = worst observed execution time (ms), U = allowed utilisation (budget/period), m = deadline misses, j = jobs completed. We record 52 (0.1), 86 (15.2) and 100 (0.0)% CPU utilisation for each stage respectively. Standard deviations are shown in parentheses. Bold values show the difference between the user-level scheduler and kernel.

As a second criticality-switch benchmark, we ported three processor intensive benchmarks from the MiBench [Guthaus et al., 2001] to act as workloads. We use SUSAN, which performs image recognition, JPEG, which does image encoding/decoding, and MAD, which plays an MP3 file. Each benchmark runs in its own Rump process with an in-memory file system, and shares a timer and serial server. We chose these specific benchmarks as they were the easiest to adapt as described below, and use them as workloads, rather than for comparing systems, so there is no issue of bias from sub-setting.

We alter the benchmarks to run periodically in multiple stages. To obtain execution times long enough, some benchmarks iterate a fixed number of times per stage. Each benchmark process executes its workload and then waits for the next period to start. Deadlines are implicit: if a periodic job finishes before the start of the next period, it is considered successful, otherwise the deadline is missed.

We run the benchmarks on both the user-level and in-kernel implementations of static mixed criticality, with 10 runs of each.

Results are shown in Table 8.16. For both schedulers (kernel versus user-level), the results are nearly exactly the same. Bold numbers indicate a difference between the kernel and user-level scheduler, with only the lowest-criticality thread affected by the criticality switch, with an additional missed deadline due to perturbations in run time due to the slightly slower user-level scheduler. For stage one, the entire workload is schedulable and there are no deadline misses. For stage two, the workload is not schedulable, and the criticality switch boosts the priorities of SUSAN and JPEG, such that they meet their deadlines, but MAD does not. In the final stage, only the most critical task meets all deadlines. This shows

```
while(true) {
  uint64_t time = get_time();

  /* release any threads */
  release_top = NULL;
  while (!empty(release_queue) && head(release_queue)->weight < time) {
    /* implict deadlines */
    head->weight = time + head->period;
    push(deadline_queue, release_top);
    release_top = pop(release_queue);
  }

  if (release_top != NULL) {
    /* set preemption timeout */
    set_timeout(release_top->weight);
  }

  /* pick a thread */
  current_thread = head(deadline_queue);
  if (current_thread != NULL) {
    if (!current_thread->resume_cap_set) {
      /* current was preempted - put it at the head of its scheduling queue */
      seL4_SchedContext_YieldTo(current->sc);
      info = seL4_Recv(endpoint, &badge, current->resume_cap);
    } else {
     /* current is waiting for us to reply - it either timeout faulted, or called us to
      * cooperatively schedule */
      current_thread->resume_cap_set = false;
      info = seL4_ReplyRecv(endpoint, info, &badge, current->resume_cap);
    }
  } else {
    /* noone to schedule */
    info = seL4_Wait(data->endpoint, &badge);
  }

  /* here we wake from an IPC or interrupt */
  if (badge >= top_thread_id) {
    /* it's a preemption interrupt */
    handle_interrupt();
  } else {
    /* it's an IPC - must be from current */
    pop(deadline_tree);
    push(release_tree, current);
    prev_thread = current;
    prev_thread->resume_cap_set = true;
  }
}
```

Listing 8.1: Pseudocode for the EDF scheduler.

that it is sufficient to implement criticality at user-level, and our mechanisms operate as intended.

8.4.2 User-level EDF

We implement the EDF scheduler as an active server with active clients which run at an seL4 priority below the user-level scheduler. The scheduler waits on an endpoint on which it receives messages from its clients and the timer, as shown in Listing 8.1.

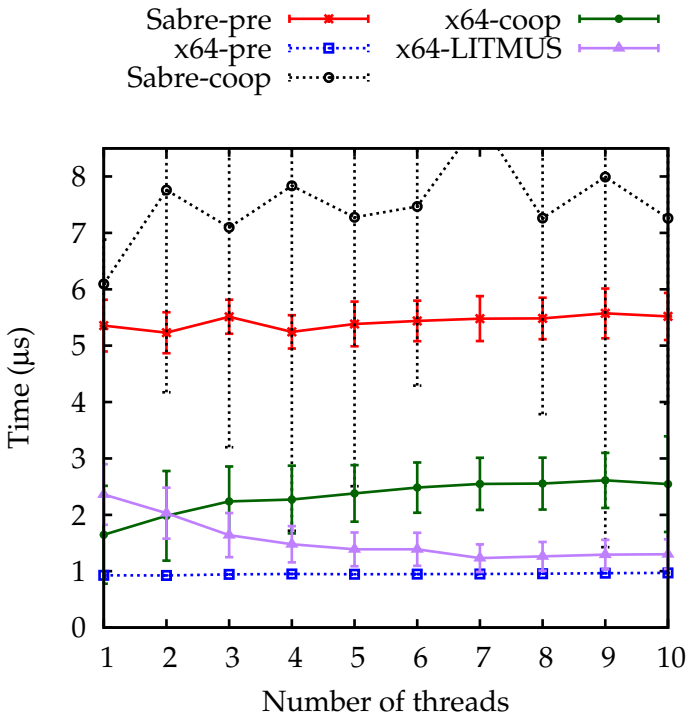


Figure 8.8: Execution time of seL4 user-mode EDF scheduler compared to kernel scheduler in x64 LITMUS^{RT}.

Each client has a period, representing its relative deadline, and a full reservation (equal to the period). Clients either notify the scheduler of completion by an IPC message, or else create a timeout exception on preemption, which is also received by the scheduler. Either is an indication that the next thread should be scheduled.

We use the *randfixedsum* [Emberson et al., 2010] algorithm to generate deadlines between 10 and 1000 ms for a certain number of threads. A set of threads runs until 100 scheduling decisions have been recorded. We repeat this 10 times, resulting in 1,000 scheduler runs for each data point.

We measure the scheduler latency by recording the timestamp when each client thread, and an idle thread, detects a context switch and processing the difference in timestamp pairs offline. We run two schedulers: *pre*-empt where threads never yield and must incur a timeout exception, and *coop*, where threads use IPC to yield to the scheduler. The latter invokes the user-level timer driver more often as the release queue is nearly always full, which involves more kernel invocations to acknowledge the IRQ, in addition to reprogramming the timer.

We compare our latencies to those of LITMUS^{RT} [Calandrino et al., 2006], a widely-used real-time scheduling framework for developing real-time schedulers and locking protocols. As it is embedded in Linux, LITMUS^{RT} is not aimed at high-assurance systems.

We use Feather-Trace [Brandenburg and Anderson, 2007] to gather data. We use the C-EDF scheduler, which is a partitioned (per-core) EDF scheduler, on a single core. We

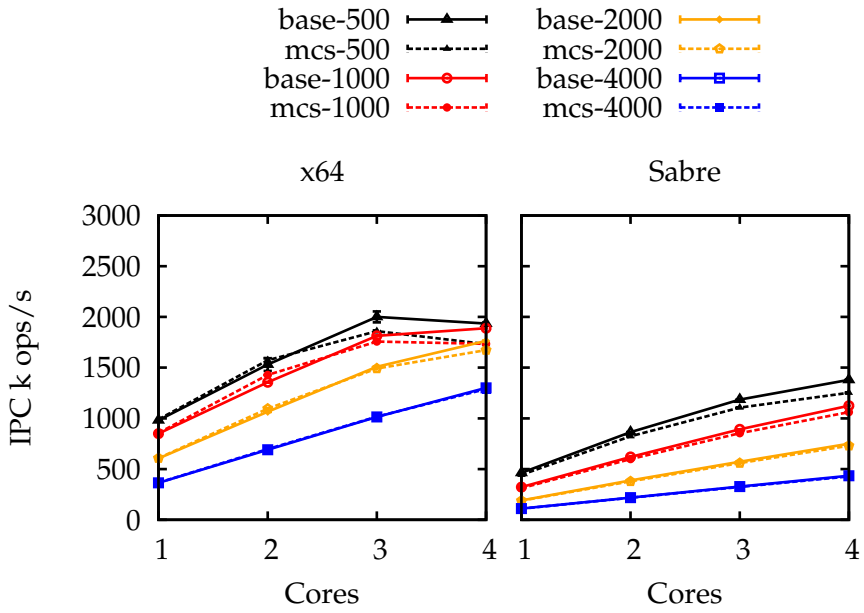


Figure 8.9: Results of the multicore IPC throughput benchmark, baseline seL4 vs MCS. Each series is named *name-N*, where *name* is *base* and *mcs* for the baseline and MCS kernel respectively, and *N* is the upper bound on the number of cycles between each IPC for that series.

use the same parameters and thread sets, running each set for 10 s. The measured overhead considers the in-kernel scheduler, context-switch and user-level code to return to the user.

Figure 8.8 shows that our preemptive user-level EDF scheduler implementation is actually faster than the in-kernel EDF scheduler from LITMUS^{RT}, and that the cost of implementing scheduling policy at user level is of the same order as the in-kernel default scheduler. In other words, implementing different policies on top of the base scheduler is quite feasible.

8.5 Multiprocessor benchmarks

We run two multicore benchmarks, the first evaluating multicore throughput of the MCS kernel versus the baseline kernel, the second based on our shared server AES case study to demonstrate the multicore model.

We run multiprocessor benchmarks on two of our platforms, SABRE and X64. Both are symmetric multiprocessors with four cores.

8.5.1 Throughput

We run a multicore throughput benchmark to show that our MCS model avoids introducing scalability problems on multiple cores compared to the baseline kernel. We modify the

existing multicore IPC throughput benchmark for seL4 to run on the MCS kernel. At time of writing, only X64 and SABRE have seL4 multiprocessor support, consequently these are the platforms used for the benchmark.

The existing multicore benchmark measures IPC throughput of a client and server, both pinned to the same processor, sending fastpath, 0-length IPC messages via `seL4_Call` and `seL4_ReplyRecv`. One pair of client and server is set up per core. Both threads are the same priority and the messages are 0 length. Each thread spins for a random amount with an upper bound N between each subsequent IPC. As N increases so does IPC throughput, as fewer calls are made.

We modify the benchmark such that each server thread is passive on the MCS kernel. Results are displayed in Figure 8.9 and show a minor impact on IPC throughput for low values of N. Scalability is not impacted on SABRE, but is on X64, with the curve flattening slightly more aggressively on the MCS kernel due to the fastpath overhead. The low overhead is expected as the MCS model only introduces extra per-core state, with no extra shared state between cores. The slight scalability impact is due to a higher cache-footprint of the IPC fastpath.

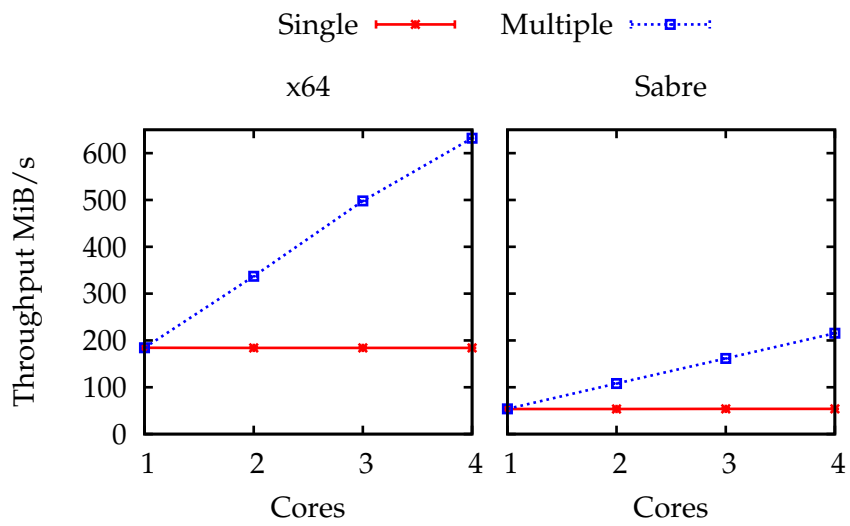


Figure 8.10: Results of the AES shared server multicore case study. *Single* shows results for a passive server thread migrating between cores to service clients, while *Multiple* has one passive server thread per core. For both series, the number of clients is equal to the number of cores and each client requests 1MiB of data encrypted.

8.5.2 Shared server

We adapt our AES case study (Section 8.3.2) to demonstrate how our MCS model applies to multiprocessors. The AES server is configured without a timeout fault handler, and we run two variants.

Single: the AES server has a single passive thread, which waits on a single endpoint and migrates to the core of the active client over IPC, effectively serialising access to the server. Consequently, Fig. 8.10 shows there is no gain in throughput when further cores are added.

Multiple: the AES server has one passive thread per core, and an endpoint is set up for each core, demonstrating a parallel server. Due to the absence of bottlenecks in the stateless AES server, this results in near perfect scalability.

8.6 Summary

All in all, we have demonstrated via micro- and macro- benchmarks that our overheads are reasonable given the speed of the baseline kernel and the extent of the provided functionality.

Through two system benchmarks and one shared-server benchmark, we have shown that our approach guarantees processor isolation and that threads cannot exceed their budget allocation via their scheduling context. Additionally, we have shown that isolation can be

achieved in a shared server via a timeout-fault handler, and implemented several alternatives for handling such faults, demonstrating the feasibility of the model.

Policy freedom is maintained despite providing fixed-priority scheduling in the kernel, as the mechanisms are sufficient to implement low-overhead, user-level schedulers, as demonstrated through the static, mixed-criticality and EDF scheduler implementations.

Finally, we have demonstrated that the model works for multiprocessors, incurring no great scalability penalty over baseline seL4, and show how passive servers migrate across cores on IPC.

9 Conclusion

Emerging cyber-physical systems represent an opportunity for greater safety, security and automation, as they can replace systems where human error prevail. While humans are excellent at innovation and creativity, they cannot compete with machines that do not get tired, drunk or distracted when completing monotonous, repetitive tasks. Car accidents are overwhelmingly caused by human error, measured at 94% in the US [Singh, 2015], something that self-driving cars can overcome. Autonomous aircraft and other fleet vehicles, smart cities and smart factories hold similar promise.

As established in Section 1.1, in order to practically develop such systems, they must be mixed-criticality, as certification of all parts of such a system to high-assurance levels is not feasible. Consequently, require strong temporal isolation, asymmetric protection, and mechanisms to support sharing resources between sub-systems of different criticality levels.

This thesis has drawn on real-time theory and systems practice to develop core-mechanisms required in a high-assurance, trusted computing base for mixed criticality systems. Importantly, the mechanisms we have developed do not violate other requirements, like policy-freedom, integrity, confidentiality, spatial isolation, and security.

In Chapter 6, we introduced our model for treating time as a first-class resource, and defined a minimal set of mechanisms that can be leveraged to implement temporal isolation between threads, even within shared servers. Our primitives also allow for more complex, application specific user-level schedulers to be constructed efficiently, as we showed in the evaluation. Although many attempts at integrating a capability system with the non-fungible resource that is time have existed before, we believe this is the first complete implementation which provides permission to time while not imposing policy restrictions. Unlike many models before it, we do not impose hierarchical scheduling, and our model does not conflate criticality, temporal sensitivity and trust.

9.1 Contributions

Specifically, we make the following contributions:

- Mechanisms for principled management of time through capabilities to scheduling contexts, which is compatible with the fast IPC implementations traditional used in high-performance microkernels, and is compatible with established real-time resource-sharing policies;
- An implementation of those mechanisms in the non-preemptible seL4 microkernel, and an exploration of how the implementation interacts with the existing kernel model;
- An evaluation of the overheads and isolation properties of the implementation, to the point of showing isolation in a shared server through timeout-fault handling;
- Design and implementation of many different, user-level timeout-fault handling policies;
- An implementation of a user-level EDF scheduler which is competitive with the LITMUS^{RT}, in-kernel EDF scheduler, which shows that despite the fixed-priority scheduler present in the kernel, other schedulers remain practical;
- and a design and implementation of a criticality switch at user-level, which shows that criticality is not required to be a kernel provided property;

The implementation is complete, and the verification of this model is currently in-progress. Specifications have already been developed by the verification engineering team at Data61, CSIRO, who are now completing the first-level refinement of the functional correctness proof. The maximum controlled priority feature has already been merged to the master kernel. Verification is beyond the scope of this PhD, although we continue to work closely with the team to assist in verification.

This work is part of the Side-Channel Causal Analysis for Design of Cyber-Physical Security project for the U.S Department of Homeland Security.

Additionally, during the development of this thesis we have made extensive contributions to the seL4 benchmarking suite, testing suite and user-level libraries, all of which provide better support for running experiments and building systems on seL4.

9.2 Future work

Modern CPUs have dynamic frequency scaling and low-power modes in order to save power, which is of high concern in many embedded systems. The implementation as it stands assumes a constant CPU frequency, and all kernel operations are calculated in CPU cycles. Frequency scaling can have undesirable effects on real-time processes: if the period is specified in microseconds, then converted to cycles and the CPU frequency changes, does

the period remain correct? This is a limitation of our model and promising area for future work.

Although we provide a mechanism for cross-core IPC, where passive threads migrate between processing cores and active threads remain fixed, there is much more to consider in terms of multicore scheduling and resource sharing. Our model provides a partitioned, fixed priority scheduler, and load-balancing is up to user level. Further experiments to evaluate the mechanisms for higher-level multicore scheduling and resource sharing protocols is required.

9.3 Summary

Cyber-physical systems are the future, and our work has focused on principled mechanisms for managing time in such systems, which are critical in more ways than the impact of failure. The work we have done is one step of thousands towards of future of trustworthy, safe and secure autonomous systems.

This material is based on research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) BAA HSHQDC-14-R-B00016, and the Government of United Kingdom of Great Britain and the Government of Canada via contract number D15PC00223.

Bibliography

Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, 2004.

James H. Anderson and Anand Srinivasan. Mixed Pfair/ERFair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68:157–204, February 2004.

James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.

ARINC. *Avionics Application Software Standard Interface*. ARINC, November 2012. ARINC Standard 653.

T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A research agenda for mixed-criticality systems, April 2009. URL http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/.

S. K. Baruah, N. K Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *Proceedings of the 19th European Conference on Algorithms*, pages 555–566, Berlin, Heidelberg, 2011a.

Sanroy K. Baruah, Alan Burns, and Rob I. Davis. Response-time analysis for mixed criticality systems. In *IEEE Real-Time Systems Symposium*, pages 34–43, Vienna, Austria, December 2011b. IEEE Computer Society.

Bernard Blackham, Vernon Tang, and Gernot Heiser. To preempt or not to preempt, that is the question. In *Asia-Pacific Workshop on Systems (APSys)*, page 7, Seoul, Korea, July 2012. ACM.

Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 95–112, Seattle, WA, US, April 1992. USENIX Association.

Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011. http://cs.unc.edu/~bbb/diss/brandenburg-diss.pdf.

Björn B. Brandenburg. A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems. In *IEEE Real-Time Systems Symposium*, pages 196–206, Rome, IT, December 2014. IEEE Computer Society.

Björn B. Brandenburg and James H. Anderson. Feather-trace: A light weight event tracing toolkit. In *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*, pages 61–70, Pisa, IT, July 2007. IEEE Computer Society.

Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. Real-time synchronization on multiprocessors: to block or not to block, to suspend or spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 342–353. IEEE Computer Society, April 2008.

Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *IEEE Real-Time Systems Symposium*, pages 396–407, Cancun, MX, December 2003. IEEE Computer Society.

Manfred Broy, Ingolf H. Krüger, Alexander Pretschner, and Christian Salzman. Engineering automotive software. *Proceedings of the IEEE*, 95:356–373, 2007.

Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys*, 50(6):1–37, 2017.

Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, 29(1): 5–26, 2005.

Giorgio C. Buttazzo and John A. Stankovic. RED: Robust earliest deadline scheduling. In *Proceedings of the 3rd International Workshop on Responsive Computing Systems*, pages 100–111, 1993.

John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *IEEE Real-Time Systems Symposium*, pages 111–123, Rio de Janiero, Brazil, December 2006. IEEE Computer Society.

Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys Conference*, Prague, CZ, April 2013.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, pages 143–154, Indianapolis, IN, US, June 2010. ACM.

Johnathan Corbet. Deadline scheduling for Linux, October 2009. URL https://lwn.net/Articles/356576/.

Jonathan Corbet. SCHED_FIFO and realtime throttling, September 2008. URL https://lwn.net/Articles/296419/.

Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Harald Röck, and Ana Sokolova. Programmable temporal isolation in real-time and embedded execution environments. In *2nd Workshop on Isolation and Integration in Embedded Systems*, pages 19–24, Nuremburg, DE, March 2009. ACM.

Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Harald Röck, and Ana Sokolova. Temporal isolation in real-time systems: the VBS approach. *Software Tools for Technology Transfer*, pages 1–21, 2012.

Matthew Danish, Ye Li, and Richard West. Virtual-CPU scheduling in the quest operating system. In *IEEE Real-Time Systems Symposium*, pages 169–179, Chicago, IL, 2011.

Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, October 2011.

Dionisio de Niz, Luca Abeni, Saowanee Saewong, and Ragunathan Rajkumar. Resource sharing in reservation-based systems. In *IEEE Real-Time Systems Symposium*, pages 171–180, London, UK, 2001.

Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *IEEE Real-Time Systems Symposium*, pages 291–300, Washington DC, US, 2009. IEEE Computer Society.

Dionisio de Niz, Lutz Wrage, Nathaniel Storer, Anthony Rowe, and Ragunathan Rajkumar. On resource overbooking in an unmanned aerial vehicle. In *International Conference on Cyber-Physical Systems*, pages 97–106, Washington DC, US, 2012. IEEE Computer Society.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

Deos. Deos, a time & space partitioned, multi-core enabled, DO-178C DAL A certifiable RTOS, 2018. URL https://www.ddci.com/products_deos_do_178c_arinc_653/.

UmaMaheswari C. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, 2006.

Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel data – first class citizens of the system. In *Proceedings of the 2nd International Workshop on Object Systems and Software Architectures*, pages 39–43, Victor Harbor, South Australia, Australia, January 2006.

Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *ACM Symposium on Operating Systems Principles*, pages 133–150, Farmington, PA, USA, November 2013.

Paul Emberson, Roger Stafford, and Robert Davis. Techniques for the synthesis of multiprocessor tasksets. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 6–11, Brussels, Belgium, July 2010. Euromicro.

Rolf Ernst and Marco Di Natale. Mixed criticality systems—a history of misconceptions? *IEEE Design and Test*, 33(5):65–74, October 2016.

Dario Faggioli. POSIX_SCHED_SPORADIC implementation for tasks and groups, August 2008. URL https://lwn.net/Articles/293547/.

Tom Fleming and Alan Burns. Extending mixed criticality scheduling. In *Workshop on Mixed Criticality Systems*, pages 7–12, Vancouver, Canada, December 2013. IEEE Computer Society.

Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 97–114, Berkeley, CA, USA, 1994. USENIX Association.

FreeRTOS. *FreeRTOS*, 2012. https://www.freertos.org.

Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267–282, Monterey, CA, USA, June 1999. USENIX Association.

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, Paris, France, December 2017. IEEE Computer Society.

Paolo Gai, Marco Di Natale, Guiseppe Lipari, Alberto Ferrari, and Claudio Gabellini. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 189–198, Washington, DC, USA, May 2003.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 653–669, Savannah, GA, US, November 2016. USENIX Association.

Mathhew R Guthaus, Jeffrey S. Reingenberg, Dan Ernst, Todd M. Austing, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*, pages 3–14, Washington, DC, USA, December 2001. IEEE Computer Society.

Michael Gonzalez Harbour. Real-time POSIX: An overview. In *VVConex International Conference*, 1993.

Michael Gonzalez Harbour and José Carlos Palencia. Response time analysis for tasks scheduled under EDF within fixed priorities. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4):36–38, 1988.

Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, FR, October 1997.

Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude-Joachim Hamann, Michael Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. DROPS—OS support for distributed multimedia applications. In *Proceedings of the 8th SIGOPS European Workshop*, Sintra, Portugal, September 1998.

Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems*, 34(1):1:1–1:29, April 2016.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89, July 2006.

André Hergenhan and Gernot Heiser. Operating systems technology for converged ECUs. In *6th Embedded Security in Cars Conference (escar)*, page 3 pages, Hamburg, Germany, November 2008.

Jonathan L. Herman, Christopher J. Kenna, Malcolm S. Mollison, James H. Andersony, and Daniel M. Johnson. RTOS support for multicore mixed-criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2012.

Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, DE, July 2002.

Intel 64 & IA-32 ASDM. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3(3A, 3B, 3C & 3D): System Programming Guide*. Intel Corporation, September 2016.

Michael B. Jones. What really happened on Mars?, 1997. URL http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html.

Antti Kantee and Justin Cormack. Rump kernels: No OS? No problem! *USENIX ;login:*, 29(5):11–17, October 2014.

Shinpei Kato, Yutaka Ishikawa, and Ragunathan (Raj) Rajkumar. CPU scheduling and memory management for interactive real-time applications. *Real-Time Systems*, 47(5): 454–488, September 2011.

Larry Kinnan and Joseph Wlad. Porting applications to an ARINC653 compliant IMA platform using VxWorks as an example. In *IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 10.B.1–1–10.B.1–8, Salt Lake City, UT, US, October 2004. IEEE Aerospace and Electronic System Society.

Takuro Kitayama, Tatsuo Nakajima, and Hideyuki Tokuda. RT-IPC: An IPC extension for real-time Mach. In *Proceedings of the 2nd USENIX Workshop on Microkernels and other Kernel Architectures*, pages 91–103, San Diego, CA, US, September 1993.

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. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

Gilard Koren and Dennis Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *IEEE Real-Time Systems Symposium*, pages 110–117, 1995.

Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *International Conference on Embedded Software*, pages 93–102, Tampere, SF, October 2012. USENIX Association.

Gerardo Lamastra, Giuseppe Lipari, and Luca Abeni. A bandwidth inheritance algorithm for real-time task synchronisation in open systems. In *IEEE Real-Time Systems Symposium*, pages 151–160, London, UK, December 2001. IEEE Computer Society Press.

John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard-real-time environments. In *IEEE Real-Time Systems Symposium*, pages 261–270, San Jose, 1987. IEEE Computer Society.

Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, 1996.

J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1982.

Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. Predictable communication and migration in the Quest-V separation kernel. In *IEEE Real-Time Systems Symposium*, pages 272–283, Rome, Italy, December 2014. IEEE Computer Society.

Jochen Liedtke. On μ-kernel construction. In *ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

Jochen Liedtke. *L4 Reference Manual*. GMD/IBM, September 1996.

C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

Mingsong Lv, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, and Jianming Zhang. A survey of WCET analysis of real-time operating systems. In *Proceedings of the 9th IEEE International Conference on Embedded Systems and Software*, pages 65–72, Hangzhou, CN, May 2009.

Anna Lyons and Gernot Heiser. Mixed-criticality support in a high-assurance, general-purpose microkernel. In *Workshop on Mixed Criticality Systems*, pages 9–14, Rome, Italy, December 2014.

Anna Lyons, Kent Mcleod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM.

Antonio Mancina, Dario Faggioli, Guiseppe Lipari, Jorrit N. Herder, Ben Gras, and Andrew S. Tanenbaum. Enhancing a dependable multiserver operating system with temporal protection via resource reservations. *Real-Time Systems*, 48(2):177–210, August 2009.

Kent McLeod. Usermode OS components on seL4 with rump kernels. BE thesis, School of Electrical Engineering and Telecommunication, Sydney, Australia, October 2016.

Cliff Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Workshop on Workstation Operating Systems*, pages 129–134, Napa, CA, US, 1993. IEEE Computer Society.

Cliff Mercer, Raguanthan Rajkumar, and Jim Zelenka. Temporal protection in real-time operating systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 79–83, San Juan, Puerto Rico, May 1994. IEEE Computer Society.

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*, pages 415–429, San Francisco, CA, May 2013.

Shuichi Oikawa and Ragunathan Rajkumar. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symposium*, pages 111–120, Madrid, Spain, 1998. IEEE Computer Society.

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

Gabriel Parmer. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*, pages 91–100, Brussels, BE, July 2010. ACM.

Gabriel Parmer and Richard West. HiRes: A system for predictable hierarchical resource management. In *IEEE Real-Time Systems Symposium*, pages 180–190, Washington, DC, US, April 2011. IEEE Computer Society.

Risat Mahmud Pathan. *Three Aspects of Real-Time Multiprocessor Scheduling: Timeliness, Fault Tolerance, Mixed Criticality*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, SE, 2012.

Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Workshop on Hot Topics in Parallelism*, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

Stefan M Petters, Kevin Elphinstone, and Gernot Heiser. *Trustworthy Real-Time Systems*, pages 191–206. Signals & Communication. Springer, January 2012.

QNX Neutrino System Architecture [6.5.0]. QNX Software Systems Co., 6.5.0 edition, 2010.

Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 116–123, June 1990.

Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. In *Proc. SPIE3310, Multimedia Computing and Networking*, 1998.

Sandra Ramos-Thuel and John P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 160–171, 1993.

RedisLabs. Redis, 2009. URL https://redis.io.

RTEMS. RTEMS, 2012. URL https://www.rtems.org.

Sergio Ruocco. Real-Time Programming and L4 Microkernels. In *Proceedings of the 2nd Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*, Dresden, Germany, July 2006.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer.

Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Vienna, Austria, April 2016.

Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9): 1175–1185, September 1990.

Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real-time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2–3):101–155, November 2004.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *ACM Symposium on Operating Systems Principles*, pages 170–185, Charleston, SC, USA, December 1999. ACM.

Santokh Singh. Critical reasons for crashes investigated in the National Motor Vehicle Causation Survey, February 2015. URL https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115.

Brinkley Sprunt, Lui Sha, and John K. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

Marco Spuri and Giorgio Buttazzo. Efficient aperiodic service under the earliest deadline scheduling. In *IEEE Real-Time Systems Symposium*, December 1994.

Anand Srinivasan and James H. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

John A. Stankovic. Misconceptions about real-time computing: a serious problem for next generation systems. *IEEE Computer*, October 1988.

Mark J. Stanovic, Theodore P. Baker, An-I Wang, and Michael González Harbour. Defects of the POSIX sporadic server and how to correct them. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 35–45, Stockholm, 2010. IEEE Computer Society.

Mark J. Stanovic, Theodore P. Baker, and An-I Wang. Experience with sporadic server scheduling in Linux: Theory vs. practice. In *Real-Time Linux Workshop*, pages 219–230, October 2011.

Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*, pages 209–222, Paris, FR, April 2010. ACM.

Udo Steinberg, Jean Wolter, and Hermann Härtig. Fast component interaction for real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 89–97, Palma de Mallorca, ES, July 2005.

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, Brussels, BE, July 2010. IEEE Computer Society.

Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real-Time Systems Symposium*, 1996.

Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.

SysGo. PikeOS hypervisor, 2012. URL http://www.sysgo.com/products/pikeos-hypervisor.

The NetBSD Foundation. NetBSD, 2018. URL https://www.netbsd.org.

Data61 Trustworthy Systems Team. *seL4 Reference Manual, Version 7.0.0*, September 2017.

Steven H. VanderLeest. ARINC 653 hypervisor. In *Proceedings of the 29th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 5.E.2–1–5.E.2–20, Salt Lake City, UT, US, 2010. IEEE Aerospace and Electronic System Society.

Manohar Vanga, Andrea Bastoni, Henrik Theiling, and Björn B. Brandenburg. Supporting low-latency, low-criticality tasks in a certified mixed-criticality OS. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS)*, pages 227–236, Grenoble, France, October 2017. ACM.

Paulo Esteves Veríssimo, Nuno Ferreira Neves, and Miguel Pupo Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer, Berlin, Heidelberg, 2003.

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.

David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2001.

Ian Wienand and Luke Macpherson. ipbench: A framework for distributed network benchmarking. In *AUUG Winter Conference*, pages 163–170, Melbourne, Australia, September 2004.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

VxWorks Kernel Programmers Guide. Wind River, 6.7 edition, 2008.

Victor Yodaiken and Michael Barabanov. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference*, Anaheim, CA, January 1997. Satellite Workshop of the 1997 USENIX.

Alexander Zuepke, Marc Bommert, and Daniel Lohmann. AUTOBEST: a united AUTOSAR-OS and ARINC 653 kernel. In *IEEE Real-Time Systems Symposium*, pages 133–144, San Antonio, Texas, April 2015. IEEE Computer Society.

Appendices

A MCS API

A.1 System calls

For convenience, in this section we present all of the system calls in the MCS api.

A.1.1 Call

```
seL4_MessageInfo_t seL4_Call(seL4_CPtr cap, seL4_MessageInfo_t info)
```

Invoke the capability provided and block waiting on a reply. Used to communicate with the kernel to invoke objects, or for IPC when used on an endpoint capability. When used for IPC, the scheduling context of the caller may be lent to the receiver. The caller is blocked on a resume object, and wakes once that resume object has been invoked, thus sending a reply and returning a lent scheduling context.

Type	*Parameter*	*Description*
`seL4_CPtr`	`dest`	Capability to invoke
`seL4_CPtr`	`info`	A `seL4_MessageInfo_t` structure encoding details about the message

Return value: an `seL4_MessageInfo_t` structure encoding details about the reply message.

A.1.2 Send

```
void seL4_Send(seL4_CPtr cap, seL4_MessageInfo_t info)
```

Invoke the capability provided. If used on an endpoint capability, block until the message has been sent.

Type	*Parameter*	*Description*
`seL4_CPtr`	`dest`	Capability to invoke
`seL4_CPtr`	`info`	A `seL4_MessageInfo_t` structure encoding details about the message

Return value: none.

A.1.3 NBSend

```
void seL4_NBSend(seL4_CPtr cap, seL4_MessageInfo_t info)
```

Invoke the capability provided. If used on an endpoint capability and no receiver is present, do not send the message and return immediately.

Type	*Parameter*	*Description*
`seL4_CPtr`	`dest`	Capability to invoke
`seL4_CPtr`	`info`	A `seL4_MessageInfo_t` structure encoding details about the message

Return value: none

A.1.4 Recv

```
seL4_MessageInfo_t seL4_Recv(seL4_CPtr cap, seL4_Word *badge, seL4_CPtr reply)
```

Wait for a message on the provided capability. If the provided capability is not an endpoint or notification, raise a capability fault. Passive threads may receive scheduling contexts from this system call.

Type	*Parameter*	*Description*
`seL4_CPtr`	`src`	Capability to wait for a message on.
`seL4_Word*`	`badge`	address for the kernel to write the badge value of the sender to.
`seL4_CPtr`	`reply`	Capability to the resume object to block callers on.

Return value: An `seL4_MessageInfo_t` structure encoding details about the message received.

A.1.5 NBRecv

```
seL4_MessageInfo_t seL4_NBRecv(seL4_CPtr cap, seL4_Word *badge, seL4_CPtr reply)
```

Poll for a message on the provided capability. If the provided capability is not an endpoint or notification, raise a capability fault. Passive threads may receive scheduling contexts from this system call.

Type	*Parameter*	*Description*
`seL4_CPtr`	`src`	Capability to wait for a message on.
`seL4_Word*`	`badge`	address for the kernel to write the badge value of the sender to.
`seL4_CPtr`	`reply`	Capability to the resume object to block callers on.

Return value: An `seL4_MessageInfo_t` structure encoding details about the message received.

A.1.6 Wait

```
seL4_MessageInfo_t seL4_Wait(seL4_CPtr cap, seL4_Word *badge)
```

Wait for a message on the provided capability. If the provided capability is not an endpoint or notification, raise a capability fault.

Type	*Parameter*	*Description*
`seL4_CPtr`	`src`	Capability to wait for a message on.
`seL4_Word*`	`badge`	address for the kernel to write the badge value of the sender to.

Return value: An `seL4_MessageInfo_t` structure encoding details about the message received.

A.1.7 NBWait

```
seL4_MessageInfo_t seL4_NBWait(seL4_CPtr cap, seL4_Word *badge)
```

Poll for a message on the provided capability. If the provided capability is not an endpoint or notification, raise a capability fault.

Type	*Parameter*	*Description*
`seL4_CPtr`	`src`	Capability to wait for a message on.
`seL4_Word*`	`badge`	address for the kernel to write the badge value of the sender to.

Return value: An `seL4_MessageInfo_t` structure encoding details about the message received.

A.1.8 ReplyRecv

```
seL4_MessageInfo_t seL4_ReplyRecv(seL4_CPtr cap, seL4_MessageInfo_t info, seL4_Word
*badge, seL4_CPtr reply)
```

Invoke a resume object, sending a reply message, then block on the provided capability, waiting for a message on the provided capability, with the now cleared resume object. Passive threads may receive scheduling contexts from this system call.

Type	*Parameter*	*Description*
`seL4_CPtr`	`src`	Capability to wait for a message on.
`seL4_CPtr`	`info`	A `seL4_MessageInfo_t` structure encoding details about the message
`seL4_Word*`	`badge`	address for the kernel to write the badge value of the sender to.
`seL4_CPtr`	`reply`	Capability to the resume object to block callers on.

Return value: An `seL4_MessageInfo_t` structure encoding details about the message received.

A.1.9 NBSendRecv

```
seL4_MessageInfo_t seL4_NBSendRecv(seL4_CPtr dest, seL4_MessageInfo_t info, seL4_CPtr
src, seL4_Word *badge, seL4_CPtr reply)
```

Perform a non-blocking send on a capability, then block on the provided capability, waiting for a message on the provided capability, with the resume object. Passive threads may receive scheduling contexts from this system call.

Type	*Parameter*	*Description*
`seL4_CPtr`	`dest`	Capability to invoke
`seL4_CPtr`	`info`	A `seL4_MessageInfo_t` structure encoding details about the message
`seL4_CPtr`	`src`	Capability to wait for a message on.
`seL4_Word*`	`badge`	address for the kernel to write the badge value of the sender to.
`seL4_CPtr`	`reply`	Capability to the resume object to block callers on.

Return value: An `seL4_MessageInfo_t` structure encoding details about the message received.

A.1.10 NBSendWait

```
seL4_MessageInfo_t seL4_NBSendWait(seL4_CPtr dest, seL4_MessageInfo_t info, seL4_CPtr
src, seL4_Word *badge)
```

Perform a non-blocking send on a capability, then block on the provided capability, waiting for a message on the provided capability.

Type	*Parameter*	*Description*
`seL4_CPtr`	`dest`	Capability to invoke
`seL4_CPtr`	`info`	A `seL4_MessageInfo_t` structure encoding details about the message
`seL4_CPtr`	`src`	Capability to wait for a message on.
`seL4_Word*`	`badge`	address for the kernel to write the badge value of the sender to.

Return value: An `seL4_MessageInfo_t` structure encoding details about the message received.

A.1.11 Yield

```
void seL4_Yield(void)
```

Exhaust the head replenishment of the current thread and place the thread at the end of the scheduling queue, or into the release queue if the next replenishment is not yet available.

Type	*Parameter*	*Description*
`void`		

Return value: none

A.2 Invocations

In this section, we only present invocations added or changed by the MCS scheduling API, the rest can be found in the seL4 manual [Trustworthy Systems Team, 2017].

A.2.1 Scheduling context invocations

A.2.2 SchedContext - Bind

```
seL4_Error seL4_SchedContext_Bind(seL4_CPtr sc, seL4_CPtr cap)
```

Bind a scheduling context to a provided TCB or Notification object. None of the objects (SC, TCB or Notification) can be already bound to other objects. If the TCB is in a runnable state and the scheduling context has available budget, this operation will place the TCB in the scheduler at the TCB's priority.

Type	*Parameter*	*Description*
`seL4_CPtr`	`sc`	Capability to the SC to bind an object to.
`seL4_CPtr`	`cap`	Capability to a TCB or Notification object to bind to this SC.

Return value: 0 on success, `seL4_Error` code on error.

A.2.3 SchedContext - Unbind

```
seL4_Error seL4_SchedContext_Unbind(seL4_CPtr sc)
```

Remove any objects bound to a specific scheduling context.

Type	*Parameter*	*Description*
`seL4_CPtr`	`sc`	Capability to the SC to unbind an object to.

Return value: 0 on success, `seL4_Error` code on error.

A.2.4 SchedContext - UnbindObject

```
seL4_Error seL4_SchedContext_UnbindObject(seL4_CPtr sc, seL4_CPtr cap)
```

Remove a specific object bound to a specific scheduling context.

Type	*Parameter*	*Description*
`seL4_CPtr`	`sc`	Capability to the SC to unbind an object to.
`seL4_CPtr`	`cap`	Capability to a TCB or Notification object to unbind from this SC.

Return value: 0 on success, `seL4_Error` code on error.

A.2.5 SchedContext - Consumed

```
seL4_Error seL4_SchedContext_Consumed(seL4_CPtr sc)
```

Return the amount of time used by this scheduling context since this function was last called or a timeout fault triggered.

Type	*Parameter*	*Description*
`seL4_CPtr`	`sc`	Capability to the SC to act on.

Return value: An error code and a **`uint64_t`** consumed value.

A.2.6 SchedContext - YieldTo

```
seL4_Error seL4_SchedContext_YieldTo(seL4_CPtr sc)
```

If a thread is currently runnable and running on this scheduling context and the scheduling context has available budget, place it at the head of the scheduling queue. If the caller is at an equal priority to the thread this will result in the thread being scheduled. If the caller is at a higher priority the thread will not run until the threads priority is the highest priority in the system. The caller must have a maximum control priority greater than or equal to the threads priority.

Type	*Parameter*	*Description*
`seL4_CPtr`	`sc`	Capability to the SC to act on.

Return value: An error code and a **`uint64_t`** consumed value.

A.2.7 Sched_control Invocations

A.2.8 SchedControl - Configure

```
seL4_Error seL4_SchedControl_Configure(seL4_CPtr sched_control, seL4_CPtr sc, uint64_t
budget, uint64_t period, seL4_Word extra_refills, seL4_Word badge)
```

Configure a scheduling context by invoking a `sched_control` capability. The scheduling context will inherit the affinity of the provided `sched_control`.

Type	*Parameter*	*Description*
seL4_CPtr	sched_control	sched_control capability to invoke to configure the SC.
seL4_CPtr	sc	Capability to the SC to configure.
uint64_t	budget	Timeslice in microseconds.
uint64_t	period	Period in microseconds, if equal to budget, this thread will be treated as a round-robin thread. Otherwise, sporadic servers will be used to assure the scheduling context does not exceed the budget over the specified period.
seL4_Word	extra_refills	Number of extra sporadic replenishments this scheduling context should use. Ignored for round-robin threads. The maximum value is determined by the size of the SC that is being configured.
seL4_Word	badge	Badge value that is delivered to timeout fault handlers

Return value: 0 on success, seL4_Error code on error.

A.2.9 TCB Invocations

A.2.10 TCB - Configure

```
seL4_Error seL4_TCB_Configure(seL4_CPtr tcb, seL4_CPtr cnode, seL4_Word guard, seL4_CPtr
vspace, seL4_Word vspace_data, seL4_Word buffer, seL4_CPtr buffer_cap)
```

Set the parameters of a TCB.

Type	*Parameter*	*Description*
seL4_CPtr	tcb	Capability to the TCB to configure.
seL4_CPtr	cnode	Root cnode for this TCB.
seL4_Word	guard	Optionally set the guard and guard size of the new root CNode. If set to zero, this parameter has no effect.
seL4_CPtr	vspace	Top level paging structure for this TCB.
seL4_Word	vspace_data	Has no effect on x86 or ARM processors.
seL4_Word	buffer	Location of the thread's IPC buffer.
seL4_CPtr	buffer_cap	Capability to the frame containing the thread's IPC buffer.

Return value: 0 on success, seL4_Error code on error.

A.2.11 TCB - SetMCPriority

```
seL4_Error seL4_TCB_SetMCPriority(seL4_CPtr tcb, seL4_CPtr auth, seL4_Word mcp)
```

Set the maximum control priority of a TCB.

Type	*Parameter*	*Description*
`seL4_CPtr`	`tcb`	Capability to the TCB to configure.
`seL4_CPtr`	`auth`	Capability to the TCB to use the MCP from when setting the new MCP.
`seL4_Word`	`mcp`	Value for the new MCP.

Return value: 0 on success, `seL4_Error` code on error.

A.2.12 TCB - SetPriority

```
seL4_Error seL4_TCB_SetPriority(seL4_CPtr tcb, seL4_CPtr auth, seL4_Word prio)
```

Set the priority of a TCB.

Type	*Parameter*	*Description*
`seL4_CPtr`	`tcb`	Capability to the TCB to configure.
`seL4_CPtr`	`auth`	Capability to the TCB to use the MCP from when setting the new MCP.
`seL4_Word`	`prio`	Value for the new priority.

Return value: 0 on success, `seL4_Error` code on error.

A.2.13 TCB - SetSchedParams

```
seL4_Error seL4_TCB_SetSchedParams(seL4_CPtr tcb, seL4_CPtr auth, seL4_Word mcp,
seL4_Word prio, seL4_CPtr sc, seL4_CPtr fault_ep)
```

Set the scheduling parameters of a TCB.

Type	*Parameter*	*Description*
`seL4_CPtr`	`tcb`	Capability to the TCB to configure.
`seL4_CPtr`	`auth`	Capability to the TCB to use the MCP from when setting the new MCP.
`seL4_Word`	`mcp`	Value for the new MCP.
`seL4_Word`	`prio`	Value for the new priority.
`seL4_CPtr`	`sc`	Capability of the sc to bind to this TCB.
`seL4_CPtr`	`fault_ep`	Capability to the endpoint to set as the fault endpoint for this TCB.

Return value: 0 on success, `seL4_Error` code on error.

A.2.14 TCB - SetTimeoutEndpoint

```
seL4_Error seL4_TCB_SetTimeoutEndpoint(seL4_CPtr tcb, seL4_CPtr ep)
```

Set the timeout endpoint of a TCB.

Type	*Parameter*	*Description*
`seL4_CPtr`	`tcb`	Capability to the TCB to configure.
`seL4_CPtr`	`ep`	Capability to the endpoint to set as the timeout endpoint for this TCB.

Return value: 0 on success, `seL4_Error` code on error.

B Code

B.1 Sporadic servers

This is the sporadic server implementation, scrubbed of assertions and debugging code for (some) brevity.

```
1  /* This header presents the interface for sporadic servers,
2   * implemented according to Stankcovich et. al in
3   * "Defects of the POSIX Spoardic Server and How to correct them",
4   * although without the priority management and enforcing a minimum budget.
5   */
6  /* functions to manage the circular buffer of
7   * sporadic budget replenishments (refills for short).
8   *
9   * The circular buffer always has at least one item in it.
10  *
11  * Items are appended at the tail (the back) and
12  * removed from the head (the front). Below is
13  * an example of a queue with 4 items (h = head, t = tail, x = item, [] = slot)
14  * and max size 8.
15  *
16  * [][h][x][x][t][][][]
17  *
18  * and another example of a queue with 5 items
19  *
20  * [x][t][][][][h][x][x]
21  *
22  * The queue has a minimum size of 1, so it is possible that h == t.
23  *
24  * The queue is implemented as head + tail rather than head + size as
25  * we cannot use the mod operator on all architectures without accessing
26  * the fpu or implementing divide.
27  */
28
29 /* To do an operation in the kernel, the thread must have
30  * at least this much budget - see comment on refill_sufficient */
31 #define MIN_BUDGET_US (2u * getKernelWcetUs())
32 #define MIN_BUDGET    (2u * getKernelWcetTicks())
33
34 /* Short hand for accessing refill queue items */
35 #define REFILL_INDEX(sc, index) (((refill_t *) ((sched_context_t *)(sc) +
    ↪  sizeof(sched_context_t)))[index])
36 #define REFILL_HEAD(sc) REFILL_INDEX((sc), (sc)->scRefillHead)
37 #define REFILL_TAIL(sc) REFILL_INDEX((sc), (sc)->scRefillTail)
38
39 /* return the amount of refills we can fit in this scheduling context */
40 static inline word_t
41 refill_absolute_max(cap_t sc_cap)
42 {
```

```
43      return (BIT(cap_sched_context_cap_get_capSCSizeBits(sc_cap)) -
        ↪  sizeof(sched_context_t)) / sizeof(refill_t);
44  }
45
46  /* Return the amount of items currently in the refill queue */
47  static inline word_t
48  refill_size(sched_context_t *sc)
49  {
50      if (sc->scRefillHead <= sc->scRefillTail) {
51          return (sc->scRefillTail - sc->scRefillHead + 1u);
52      }
53      return sc->scRefillTail + 1u + (sc->scRefillMax - sc->scRefillHead);
54  }
55
56  static inline bool_t
57  refill_full(sched_context_t *sc)
58  {
59      return refill_size(sc) == sc->scRefillMax;
60  }
61
62  static inline bool_t
63  refill_single(sched_context_t *sc)
64  {
65      return sc->scRefillHead == sc->scRefillTail;
66  }
67
68  /* Return the amount of budget this scheduling context
69   * has available if usage is charged to it. */
70  static inline ticks_t
71  refill_capacity(sched_context_t *sc, ticks_t usage)
72  {
73      if (unlikely(usage > REFILL_HEAD(sc).rAmount)) {
74          return 0;
75      }
76
77      return REFILL_HEAD(sc).rAmount - usage;
78  }
79
80  /*
81   * Return true if the head refill has sufficient capacity
82   * to enter and exit the kernel after usage is charged to it.
83   */
84  static inline bool_t
85  refill_sufficient(sched_context_t *sc, ticks_t usage)
86  {
87      return refill_capacity(sc, usage) >= MIN_BUDGET;
88  }
89
90  /*
91   * Return true if the refill is eligible to be used.
92   * This indicates if the thread bound to the sc can be placed
93   * into the scheduler, otherwise it needs to go into the release queue
94   * to wait.
95   */
96  static inline bool_t
97  refill_ready(sched_context_t *sc)
98  {
99      return REFILL_HEAD(sc).rTime <= (NODE_STATE(ksCurTime) + getKernelWcetTicks());
100 }
101
102 /* return the index of the next item in the refill queue */
103 static inline word_t
104 refill_next(sched_context_t *sc, word_t index)
105 {
106     return (index == sc->scRefillMax - 1u) ? (0) : index + 1u;
107 }
```

```
108
109 /* pop head of refill queue */
110 static inline refill_t
111 refill_pop_head(sched_context_t *sc)
112 {
113     UNUSED word_t prev_size = refill_size(sc);
114     refill_t refill = REFILL_HEAD(sc);
115     sc->scRefillHead = refill_next(sc, sc->scRefillHead);
116     return refill;
117 }
118
119 /* add item to tail of refill queue */
120 static inline void
121 refill_add_tail(sched_context_t *sc, refill_t refill)
122 {
123     word_t new_tail = refill_next(sc, sc->scRefillTail);
124     sc->scRefillTail = new_tail;
125     REFILL_TAIL(sc) = refill;
126 }
127
128 static inline void
129 maybe_add_empty_tail(sched_context_t *sc)
130 {
131     if (isRoundRobin(sc)) {
132         /* add an empty refill - we track the used up time here */
133         refill_t empty_tail = { .rTime = NODE_STATE(ksCurTime)};
134         refill_add_tail(sc, empty_tail);
135     }
136 }
137
138 /* Create a new refill in a non-active sc */
139 void
140 refill_new(sched_context_t *sc, word_t max_refills, ticks_t budget, ticks_t period)
141 {
142     sc->scPeriod = period;
143     sc->scRefillHead = 0;
144     sc->scRefillTail = 0;
145     sc->scRefillMax = max_refills;
146     /* full budget available */
147     REFILL_HEAD(sc).rAmount = budget;
148     /* budget can be used from now */
149     REFILL_HEAD(sc).rTime = NODE_STATE(ksCurTime);
150     maybe_add_empty_tail(sc);
151 }
152
153 /* Update refills in an active sc without violating bandwidth constraints */
154 void
155 refill_update(sched_context_t *sc, ticks_t new_period, ticks_t new_budget, word_t
    ↪  new_max_refills)
156 {
157
158     /* this is called on an active thread. We want to preserve the sliding window
        ↪  constraint -
159      * so over new_period, new_budget should not be exceeded even temporarily */
160
161     /* move the head refill to the start of the list - it's ok as we're going to
        ↪  truncate the
162      * list to size 1 - and this way we can't be in an invalid list position once
    ↪  new_max_refills
163      * is updated */
164     REFILL_INDEX(sc, 0) = REFILL_HEAD(sc);
165     sc->scRefillHead = 0;
166     /* truncate refill list to size 1 */
167     sc->scRefillTail = sc->scRefillHead;
168     /* update max refills */
169     sc->scRefillMax = new_max_refills;
```

```
    /* update period */
    sc->scPeriod = new_period;

    if (refill_ready(sc)) {
        REFILL_HEAD(sc).rTime = NODE_STATE(ksCurTime);
    }

    if (REFILL_HEAD(sc).rAmount >= new_budget) {
        /* if the heads budget exceeds the new budget just trim it */
        REFILL_HEAD(sc).rAmount = new_budget;
        maybe_add_empty_tail(sc);
    } else {
        /* otherwise schedule the rest for the next period */
        refill_t new = { .rAmount = (new_budget - REFILL_HEAD(sc).rAmount),
                         .rTime = REFILL_HEAD(sc).rTime + new_period
                       };
        refill_add_tail(sc, new);
    }
}

static inline void
schedule_used(sched_context_t *sc, refill_t new)
{
    /* schedule the used amount */
    if (new.rAmount < MIN_BUDGET && !refill_single(sc)) {
        /* used amount is to small - merge with last and delay */
        REFILL_TAIL(sc).rAmount += new.rAmount;
        REFILL_TAIL(sc).rTime = MAX(new.rTime, REFILL_TAIL(sc).rTime);
    } else if (new.rTime <= REFILL_TAIL(sc).rTime) {
        REFILL_TAIL(sc).rAmount += new.rAmount;
    } else {
        refill_add_tail(sc, new);
    }
}

/* Charge the head refill its entire amount.
 *
 * `used` amount from its current replenishment without
 * depleting the budget, i.e refill_expired returns false.
 */
void
refill_budget_check(sched_context_t *sc, ticks_t usage, ticks_t capacity)
{
    if (capacity == 0) {
        while (REFILL_HEAD(sc).rAmount <= usage) {
            /* exhaust and schedule replenishment */
            usage -= REFILL_HEAD(sc).rAmount;
            if (refill_single(sc)) {
                /* update in place */
                REFILL_HEAD(sc).rTime += sc->scPeriod;
            } else {
                refill_t old_head = refill_pop_head(sc);
                old_head.rTime = old_head.rTime + sc->scPeriod;
                schedule_used(sc, old_head);
            }
        }

        /* budget overrun */
        if (usage > 0) {
            /* budget reduced when calculating capacity */
            /* due to overrun delay next replenishment */
            REFILL_HEAD(sc).rTime += usage;
            /* merge front two replenishments if times overlap */
            if (!refill_single(sc) &&
                    REFILL_HEAD(sc).rTime + REFILL_HEAD(sc).rAmount >=
                    REFILL_INDEX(sc, refill_next(sc, sc->scRefillHead)).rTime) {
```

```
236
237                 refill_t refill = refill_pop_head(sc);
238                 REFILL_HEAD(sc).rAmount += refill.rAmount;
239                 REFILL_HEAD(sc).rTime = refill.rTime;
240             }
241         }
242     }
243
244     capacity = refill_capacity(sc, usage);
245     if (capacity > 0 && refill_ready(sc)) {
246         refill_split_check(sc, usage);
247     }
248
249     /* ensure the refill head is sufficient, such that when we wake in awaken,
250      * there is enough budget to run */
251     while (REFILL_HEAD(sc).rAmount < MIN_BUDGET || refill_full(sc)) {
252         refill_t refill = refill_pop_head(sc);
253         REFILL_HEAD(sc).rAmount += refill.rAmount;
254         /* this loop is guaranteed to terminate as the sum of
255          * rAmount in a refill must be >= MIN_BUDGET */
256     }
257 }
258
259 /*
260  * Charge a scheduling context `used` amount from its
261  * current refill. This will split the refill, leaving whatever is
262  * left over at the head of the refill.
263  */
264 void
265 refill_split_check(sched_context_t *sc, ticks_t usage)
266 {
267     /* first deal with the remaining budget of the current replenishment */
268     ticks_t remnant = REFILL_HEAD(sc).rAmount - usage;
269
270     /* set up a new replenishment structure */
271     refill_t new = (refill_t) {
272         .rAmount = usage, .rTime = REFILL_HEAD(sc).rTime + sc->scPeriod
273     };
274
275     if (refill_size(sc) == sc->scRefillMax || remnant < MIN_BUDGET) {
276         /* merge remnant with next replenishment - either it's too small
277          * or we're out of space */
278         if (refill_single(sc)) {
279             /* update inplace */
280             new.rAmount += remnant;
281             REFILL_HEAD(sc) = new;
282         } else {
283             refill_pop_head(sc);
284             REFILL_HEAD(sc).rAmount += remnant;
285             schedule_used(sc, new);
286         }
287     } else  {
288         /* split the head refill  */
289         REFILL_HEAD(sc).rAmount = remnant;
290         REFILL_HEAD(sc).rTime += usage;
291         schedule_used(sc, new);
292     }
293 }
294
295 /*
296  * This is called when a thread is eligible to start running: it
297  * iterates through the refills queue and merges any
298  * refills that overlap.
299  */
300 void
301 refill_unblock_check(sched_context_t *sc)
```

```
{
    if (isRoundRobin(sc)) {
        /* nothing to do */
        return;
    }

    /* advance earliest activation time to now */
    if (refill_ready(sc)) {
        REFILL_HEAD(sc).rTime = NODE_STATE(ksCurTime);
        NODE_STATE(ksReprogram) = true;

        /* merge available replenishments */
        while (!refill_single(sc)) {
            ticks_t amount = REFILL_HEAD(sc).rAmount;
            if (REFILL_INDEX(sc, refill_next(sc, sc->scRefillHead)).rTime <=
            ↪  NODE_STATE(ksCurTime) + amount) {
                refill_pop_head(sc);
                REFILL_HEAD(sc).rAmount += amount;
                REFILL_HEAD(sc).rTime = NODE_STATE(ksCurTime);
            } else {
                break;
            }
        }
    }
}
```

C Fastpath Performance

C.1 ARM

C.1.1 KZM

Counter	Baseline	MCS	Diff	Overhead
cycles	263 (0)	290 (0)	27	10.3 %
Cache L1I miss	3.0 (0.6)	4.0 (0.6)	1.0	33.3 %
Cache L1D miss	0 (0)	1.0 (1.0)	1.0	∞
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	132 (0)	149 (0)	17	12.9 %
Branch misspredict	5 (0)	5 (0)	0	0.0 %
memory access	0 (0)	1.0 (1.4)	1.0	∞

Table C.1: KZM `seL4_Call` fastpath.

Counter	Baseline	MCS	Diff	Overhead
cycles	304.0 (0.5)	350.0 (1.1)	46.0	15.1 %
Cache L1I miss	3.0 (0.7)	4 (0.7)	1.0	33.3 %
Cache L1D miss	0 (0)	1.0 (1.3)	1.0	∞
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	154 (0)	191 (0)	37	24.0 %
Branch misspredict	6 (0)	6 (0)	0	0.0 %
memory access	0 (0)	0.0 (0.5)	0.0	0 %

Table C.2: KZM `seL4_ReplyRecv` fastpath.

C.1.2 Sabre

Counter	Baseline	MCS	Diff	Overhead
cycles	288 (0)	372.0 (58.3)	84.0	29.2 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	4 (0)	4 (0)	0	0.0 %
TLB L1D miss	3 (0)	4 (0)	1	33.3 %
Instruction exec.	143 (0)	160 (0)	17	11.9 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	0 (0)	0 (0)	0	0 %

Table C.3: SABRE `seL4_Call` fastpath.

Counter	Baseline	MCS	Diff	Overhead
cycles	313.0 (1.0)	334.0 (2.1)	21.0	6.7 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	4 (0)	5 (0)	1	25.0 %
TLB L1D miss	4 (0)	3 (0)	-1	-25.0 %
Instruction exec.	165 (0)	201 (0)	36	21.8 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	0 (0)	0 (0)	0	0 %

Table C.4: SABRE `seL4_ReplyRecv` fastpath.

C.1.3 Hikey32

Counter	Baseline	MCS	Diff	Overhead
cycles	236.0 (2.8)	251.0 (3.4)	15.0	6.4 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	144 (0)	161 (0)	17	11.8 %
Branch misspredict	4 (0)	4.0 (0.3)	0.0	0.0 %
memory access	50 (0)	60 (0)	10	20.0 %

Table C.5: HIKEY32 `seL4_Call` fastpath.

Counter	Baseline	MCS	Diff	Overhead
cycles	252.0 (4.1)	275 (0)	23.0	9.1 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	166 (0)	202 (0)	36	21.7 %
Branch misspredict	4.0 (0.4)	4 (0)	0.0	0.0 %
memory access	59 (0)	67 (0)	8	13.6 %

Table C.6: HIKEY32 `seL4_ReplyRecv` fastpath.

C.1.4 Hikey64

Counter	Baseline	MCS	Diff	Overhead
cycles	251.0 (3.1)	278.0 (4.1)	27.0	10.8 %
Cache L1I miss	0 (0)	1 (1.0)	1	∞
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	183 (0)	210 (0)	27	14.8 %
Branch misspredict	2.0 (0.4)	3.0 (0.9)	1.0	50.0 %
memory access	81 (0)	93 (0)	12	14.8 %

Table C.7: HIKEY64 `seL4_Call` fastpath.

Counter	Baseline	MCS	Diff	Overhead
cycles	267.0 (5.2)	303 (5.4)	36.0	13.5 %
Cache L1I miss	0 (0)	2.0 (1.3)	2.0	∞
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	201 (0)	254 (0)	53	26.4 %
Branch misspredict	2.0 (0.6)	2.0 (0.4)	0.0	0.0 %
memory access	87 (0)	97 (0)	10	11.5 %

Table C.8: HIKEY64 `seL4_ReplyRecv` fastpath.

C.1.5 TX1

Counter	Baseline	MCS	Diff	Overhead
cycles	398.0 (7.6)	424.0 (1.6)	26.0	6.5 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	1.0 (1)	1.0	∞
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	183 (0)	210 (0)	27	14.8 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	94 (0)	107 (0)	13	13.8 %

Table C.9: TX1 `seL4_Call` fastpath.

Counter	Baseline	MCS	Diff	Overhead
cycles	393.0 (2.8)	424.0 (6.4)	31.0	7.9 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0.0 (0.3)	0 (0)	0.0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	201 (0)	254 (0)	53	26.4 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	102 (0)	112 (0)	10	9.8 %

Table C.10: TX1 `seL4_ReplyRecv` fastpath.

C.2 x86

C.2.1 ia32

Counter	Baseline	MCS	Diff	Overhead
cycles	440 (2.9)	422 (2.1)	-18	-4.1 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	202 (0)	213 (0)	11	5.4 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	0 (0)	0 (0)	0	0 %

Table C.11: IA32 `seL4_Call` fastpath.

Counter	Baseline	MCS	Diff	Overhead
cycles	412.0 (1.5)	448.0 (2.1)	36.0	8.7 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	208 (0)	264 (0)	56	26.9 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	0 (0)	0 (0)	0	0 %

Table C.12: IA32 `seL4_ReplyRecv` fastpath.

C.2.2 x64

Counter	Baseline	MCS	Diff	Overhead
cycles	449.0 (1.8)	456.0 (2.3)	7.0	1.6 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0 (0)	0 (0)	0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	187 (0)	224 (0)	37	19.8 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	0 (0)	0 (0)	0	0 %

Table C.13: X64 `seL4_Call` fastpath.

Counter	Baseline	MCS	Diff	Overhead
cycles	432 (1.5)	450.0 (2.1)	18.0	4.2 %
Cache L1I miss	0 (0)	0 (0)	0	0 %
Cache L1D miss	0.0 (0.5)	0 (0)	0.0	0 %
TLB L1I miss	0 (0)	0 (0)	0	0 %
TLB L1D miss	0 (0)	0 (0)	0	0 %
Instruction exec.	203 (0)	280 (0)	77	37.9 %
Branch misspredict	0 (0)	0 (0)	0	0 %
memory access	0 (0)	0 (0)	0	0 %

Table C.14: X64 `seL4_ReplyRecv` fastpath.