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

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. Traditionally, systems of different criticality were isolated by hardware. This approach is no longer practical as it has proven inefficient and restrictive. The result is mixed-criticality systems, where software applications with different criticalities execute on the same hardware. Such systems go beyond standard temporal isolation and require asymmetric protection between applications of different criticalities.

Whilst there has been some momentum in real-time operating system (RTOS) research, the implications of mixed-criticality systems for OSes have not been fully explored. The goal of this project is to develop a mixed-criticality OS. 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 will be the seL4 microkernel, which is built for the safety-critical, high assurance domain.

TODO: Make point that assurance is essential TODO: update with eurosys abstract

Acknowledgements

TODO: THANKS

Publications & Reports

- Anna Lyons, Kent Mcleod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, EuroSys Conference, page 14, Porto, Portugal, April 2018. ACM Sigops
- 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
- Manohar Vanga, Felipe Cerqueira, Björn B. Brandenburg, Anna Lyons, and Gernot Heiser. FlaRe: Efficient capability semantics for timely processor access. Manuscript available from https://mpi-sws.org/~bbb/papers/pdf/preprint-FlaRe.pdf, October 2013

Acronyms

AAV autonomous aerial vehicle. 1, 28

AES advanced encryption standard. 100, 106–108, 114

API application programming interface. 75, 86

ASID address space ID. 50

BWI bandwidth inheritance. 44

CA certification authority. 17, 28

CBS constant-bandwidth server. 25, 26, 28, 34–37, 39, 117

CFS completely-fair scheduler. 34

CPU central processing unit. 38, 105

DM deadline monotonic. 12

DROPS Dresden Real-time OPerating system. 37, 39

DS deferrable server. 36, 39

EDF earliest deadline first. 11–14, 16, 17, 24–26, 28, 29, 34–38, 43, 62, 63, 68, 114, 117

FIFO first-in first-out. 17, 32–34, 38, 54, 97

FP fixed priority. 11–14, 17, 24, 28, 29, 35–37, 43, 62, 63

FPRM fixed-priority rate monotonic. 11–13, 35, 59

FPU floating point unit. 95

HLP highest lockers protocol. 32, 35, 87

HRT hard real-time. 7, 8, 10, 13, 17, 23–28, 35, 39, 61, 65, 71, 76, 90, 118

IPC inter-process communication. 18, 19, 29, 40, 41, 45, 48, 49, 52, 54, 58, 59, 68–70, 72, 73, 86, 92, 97–101, 107, 110, 118

IPCP immediate priority ceiling protocol. 15, 16, 29, 33, 69

MCAR Mixed Criticality Architecture Requirements. 1

MCP maximum controlled priority. 77

MCS mixed criticality system. 97, 99

MMU memory management unit. 5, 23, 33, 35

MPCP multiprocessor priority ceiling protocol. 17

MPU memory protection unit. 35

MSR model specific register. 96

MSRP multiprocessor stack resource policy. 17

NCET nominal-case execution time. 28

NCP non-preemptive critical sections protocol. 15, 16

OCET overloaded-case execution time. 28

OPCP original priority ceiling protocol. 15–17, 29, 69

OS operating system. 1, 4, 13, 17–20, 30–32, 34, 35, 37–40, 43, 45, 50, 80

PIP priority inheritance protocol. 15, 16, 29, 32, 35, 41, 44, 69, 73

POSIX portable operating system interface. 31, 32, 34, 36, 43

QoS quality of service. 17, 28

RBED rate-based earliest deadline first. 25, 26, 28, 39

RM rate monotonic. 11, 12, 14, 28, 29, 63

RPC remote procedure call. 40

RTA response time analysis. 67, 68

RTOS real-time operating system. 20, 24, 31, 35–37

SC scheduling context. 63–66, 69, 72, 75

SCO scheduling context object. 75–78, 97, 98

SMP symmetric multiprocessor. 5, 118

SRP stack resource policy. 16, 17, 29

SRT soft real-time. 7, 8, 11, 17, 23, 25–28, 34, 35, 39, 61, 67, 76, 90, 109, 118

SS sporadic server. 35–37, 39, 117

SWaP size, weight and power. 1, 3, 4

TCB thread control block. 41, 48, 49, 51, 52, 54, 75–77, 108

TSC timestamp counter. 96–98

UDP User Datagram Protocol. 105, 106

VM virtual machine. 105, 106

WCET worst-case execution-time. 3, 4, 7–11, 17, 20, 23–29, 35, 36, 43, 45, 59, 76, 79, 80, 119

YCSB Yahoo! Cloud Serving Benchmark. 101, 104

ZS zero slack. 28, 35

Contents

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. Traditionally, systems of different criticality were isolated by hardware. This approach is no longer practical as it has proven inefficient and restrictive. The result is *mixed-criticality* systems, where software applications with different criticalities execute on the same hardware.

However, mixed-criticality systems have conflicting requirements that challenge operating systems (OS) design: they require mutually 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: the system is orthogonally partitioned spatially and temporally, and partitions are scheduled with fixed time slices [ARINC]. Any components that share resources are promoted to the highest criticality level and must meet the same certification standards; 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.

However, mixed-criticality systems are about more than basic consolidation; they can achieve more than traditional physically isolated systems. Allowing untrusted, less critical components to share hardware and communicate with more critical components, far more

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.

complex software can be introduced to the system. Examples include heuristic algorithms that are common in artificial intelligence algorithms, or internet connected software which by its nature cannot be completely trusted. Both of these use-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 design and implement an OS that provides the right mechanisms for efficiently supporting mixed criticality systems, and to reason about their safety. 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 allowing it to be overbooked, a key requirement of mixed-criticality systems;

G2 safe resource sharing between applications of different criticalities and different 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.

Higher engineering and safety standards are required for higher criticality levels, up to certification by independent certificate authorities at the highest levels. As a result, 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.

System	*Purpose*	*Criticality*
Airbags	Safety	Catastrophic
Anti lock breaks	Safety	Catastrophic
Obstacle detection	Safety	Hazardous
Navigation	Route planning	Minor
Communications	Optimal route planning	No effect

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

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 practise of physical separation is no longer viable for three reasons: SWaP, efficiency, and function.

SWaP First, systems with air-gaps require increased physical resources: 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 to travel. This in turn reduces utility in the form of reducing vehicle range. Therefore the consolidation that comes with combining systems of different criticalities is worthwhile.

Function 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 internet of things devices. For example consider the system for a self driving car as with components as outlined in Table 1.2. The safety systems are the most critical: if air bag or anti-lock brakes fail this could cause great injury or death to the passengers. The communications system is least critical, however useful: it downloads weather and road conditions, status of road works and accidents. This feeds in to the more critical navigation system, requiring resource sharing. This sort of system would not be possible without a mixed criticality system: unless the communications system were certified to the same level as the navigation system, which would greatly increase the cost of development. Consequently, mixed-criticality systems provide increased functionality over physically separated, isolated systems.

Efficiency High system utilisation is essential for addressing SWaP challenges, and high responsiveness is important for much of a system's desirable functionality. But high utilisation is a challenge in critical real-time systems, which are usually over-provisioned, both with redundant hardware and excess capacity; the core *integrity property* is that deadlines must always be met, meaning that there must always be time to let such threads execute their full *worst-case execution-time (WCET)*. This 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].

Consequently, most of the time the highly-critical components leave plenty of slack, which should be usable by less critical components. In terms of schedulability analysis, this constitutes an *overcommitted* system, where not everything is guaranteed to be schedulable.

In case of actual overload, the system must guarantee sufficient time to the critical components at the expense of the less critical ones, which is referred to as *asymmetric protection*.

Additionally, the higher the criticality of a software component, the higher the pessimism in analysis, raising the WCET and reducing the amount of software that can be consolidated onto a single platform. Therefore it is impractical to raise all software to the same criticality level should it share hardware, as the total resource utilisation is lowered.

More complex systems such as those found in aviation allow for highly restricted mixed-criticality systems in the form of separation kernels. ARINC allows for sharing of hardware between software applications of mixed criticality, and outlines the primitives required from an OS built for these systems, mandating full temporal and spatial isolation of the different criticality components and controlled communication. However, this is one standard that may not be suitable for all types of system.

1.2 Contributions

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. 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.

The contributions of our research are to provide the core mechanisms for building whole mixed-criticality systems upon.

1.3 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. In Chapter 3 we examine in detail methods for achieving temporal isolation and safe resource sharing in real-time systems; Chapter 4 investigates the same concepts from the systems perspective, and presents a survey of existing commercial, open-source and research operating systems. We then present the relevant details of our implementation platform, seL4, in Chapter 5.

We then draw on all of the previous chapters to design mechanisms for efficiently supporting mixed criticality systems, and present our findings, along with the trade-offs and consequences of our design in Chapter 6. Chapter 7 delves deeply into the implementation details. Finally Chapter 8 presents our benchmarks, case studies and findings.

1.4 Scope

This thesis focuses on mechanisms for building mixed-criticality systems. We focus on uniprocessor and symmetric multiprocessor (SMP) systems with memory management units (MMUs) for ARM and x86. We do not consider side- or timing-channels as part of this research.

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 much 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.

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 enough time is not 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 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.

Listing 1 Example of a basic sporadic real-time task.

```
1  for (;;) {
2          // job arrives
3          doJob();
4          // job completes before deadline
5      // sleep until next job is ready
6      sleep();
7  }
```

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 an execution time, C_i, and period, T_i, which represents the minimum inter-arrival time between

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 as used in this document.

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 separate tasks which do not overlap. Listing 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 as they allow multiple jobs from one task to be active at one time. However, tasks with arbitrary deadlines can be modelled by the sporadic task model by splitting tasks 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. 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 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).

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

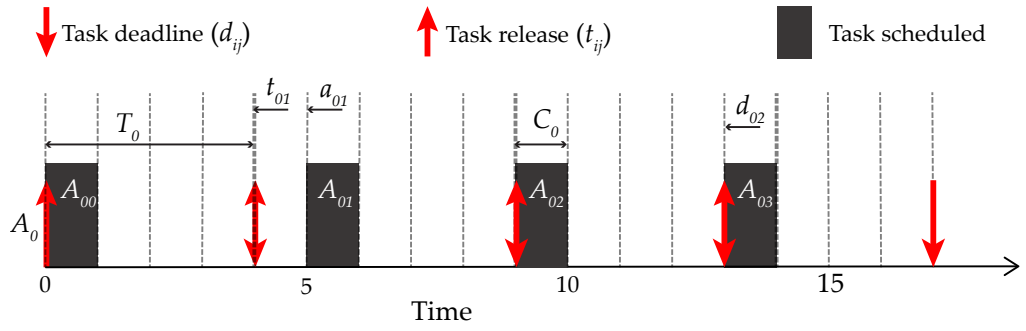


Figure 2.1: Diagram of the sporadic task model notation described in Table 2.1, showing a task set A with a single task A_0 where $T_0 = 4$ and $C_0 = 1$.

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 allowing tasks to be added and removed while the system is running.

A task set is schedulable by a scheduling algorithm if all temporal requirements are satisfied. If at least one scheduling algorithm can schedule a task set, it is said to be *feasible*. 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 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

	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]

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 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.

2.2.1 Cyclic executives

A cyclic-executive scheduler dispatches tasks according to a pre-computed schedule, and is only suitable for static, 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 if 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. The job with the highest priority is always scheduled.

Priority assignment such that all tasks meet their deadlines is the core challenge of FP scheduling. Two well-established techniques are rate monotonic (RM) and deadline monotonic, both of which are optimal with respect to FP scheduling. RM priority assign-

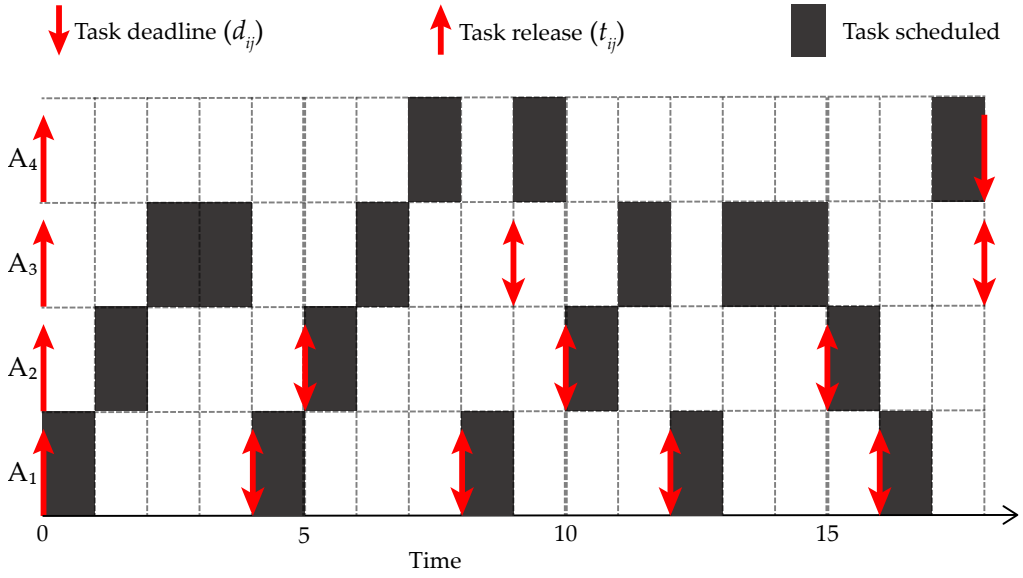


Figure 2.2: An example FPRM schedule using the task set from Table 2.2.

ment [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. Deadline monotonic (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.

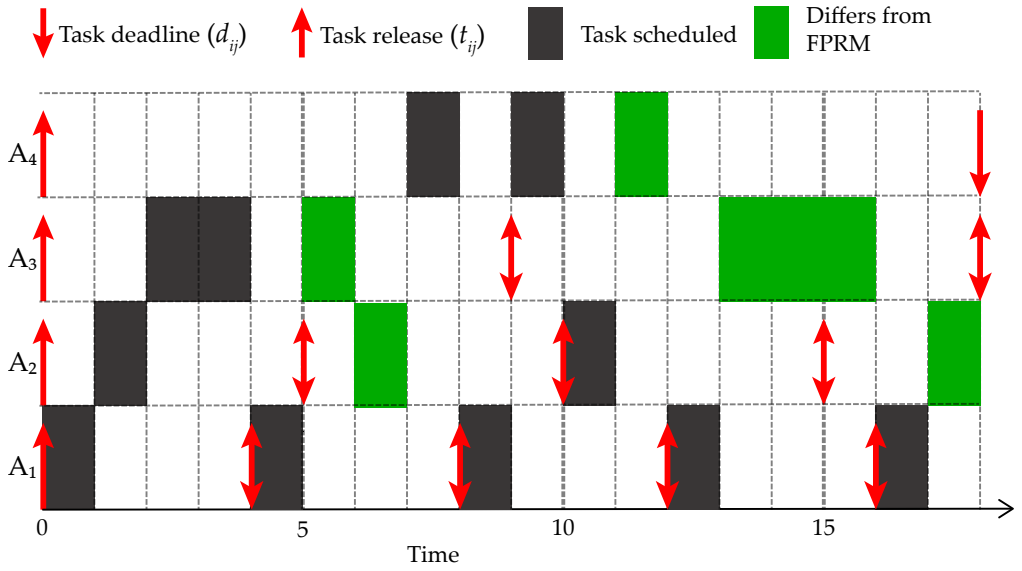


Figure 2.3: An example EDF schedule using the task set from Table 2.2.

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. EDF is considered more complex to implement and to have higher overhead-related capacity loss. Both algorithms have different behaviour under overload, and many consider the FP behaviour preferable. FP is mandated by the POSIX standard, possibly due to these misconceptions. One common argument for FPRM is the idea that the algorithmic capacity loss is mediated by the overhead related capacity loss of EDF, and much easier to implement.

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

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.

[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.

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.

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 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 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.

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 if when 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.

Neither protocol requiring resource ceilings is suitable for dynamic priority scheduling algorithms like EDF, however analogous algorithms exist, namely the stack resource policy (SRP) [Baker, 1991].

2.3.5 Other locking protocols

We do not consider lock-free and wait-free algorithms in detail in this chapter, but note while these may be deployed in real time systems there are always resources that need true blocking in order to access them; in a word, stateful devices. For memory based resources, there is no need for algorithms that avoid blocking, as long as the total blocking time is bounded and fairness is not mandated. Any approach that is transactional with no guarantee of progress is not suitable, additionally, fair algorithms are not suited to a real-time system, which are fundamentally not fair.

2.3.6 Summary

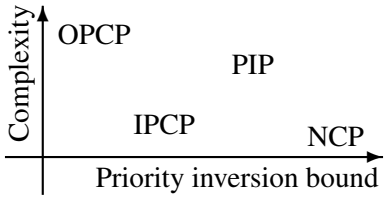


Figure 2.4: Comparison of real-time locking protocols based on implementation complexity and priority inversion bound.

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.

2.3.7 Multiprocessor locking protocols

Resource sharing on multiprocessors is far more complicated than the single processor case and still a topic of active research. 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.

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.

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 Mixed-criticality systems

Broadly, a mixed-criticality system is simply a system where software classified at different levels of criticality, according to a standard such as DO-178C (recall Table 1.1), share the same physical hardware.

TODO: references in this paragraph Varying definitions exist in the literature. Some consider merely the combination of HRT, SRT and best-effort tasks, with criticality observed in that order. Others allow for quality of service (QoS) specifications, which can reflect criticality.

In the real-time community, much work has been conducted around a theoretical 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 $L_i \leq L_{now}$ 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 mode switch increases L_{now}.

This 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 certification authority (CA) disagree on WCET estimates for a system. 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.

However, while this model, and many variations upon it have been subject too 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.

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].

2.5 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

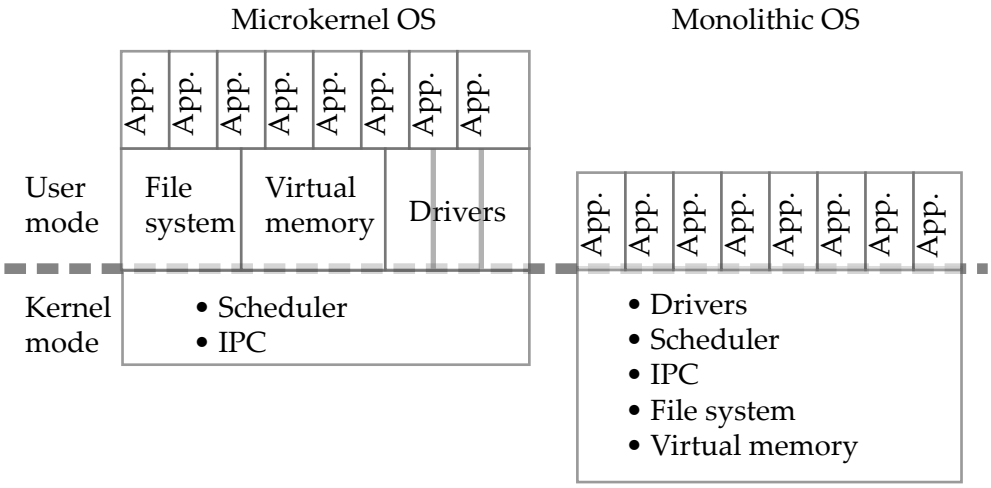


Figure 2.5: Structure of a microkernel versus monolithic operating system.

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*. 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 a scheduling, IPC; COMPOSITE [Parmer, 2009] does not provide a scheduler.

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.

2.5.1 Capabilities

Many security-focused 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].

2.5.2 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 for 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 less 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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.6 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 that 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 do 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 provides 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. 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 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) [Abeni and Buttazzo, 2004] introduce a technique for scheduling HRT and SRT tasks and providing temporal isolation. 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 [Deng and Liu, 1997; Ghazalie and Baker, 1995; Spuri and Buttazzo, 1994, 1996] 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 to 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 runtime. 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

While CBS and RBED provide temporal isolation for EDF scheduling, we will now examine methods for temporal isolation in fixed-priority systems, whilst maintaining compatibility with rate-monotonic schedulability tests. Like CBS, tasks are constrained by encapsulating one or more tasks in a server which prevents the task(s) from overrunning their assigned scheduling parameters.

Polling servers

Polling servers [Lehoczky et al., 1987] wake every period, to check if there are any pending tasks, then runs them for maximum their budget time. 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 a function of the period T.

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, but unfortunately breaks rate-monotonic schedulability analysis, as this policy can result in servers executing back-to-back and 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] swap the priority of an inactive aperiodic task with a periodic task, 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 this section, we briefly consider schedulers designed specifically for mixed-criticality systems. However, as this has been a very active topic, we refer the reader to Burns and Davis [2017] for an extensive background.

Temporal isolation in mixed-criticality systems can result in *criticality inversion*, where high criticality tasks miss their deadlines due to lower criticality tasks. Rather than temporal isolation, mixed-criticality systems require *asymmetric protection*, where deadline misses of low-criticality tasks caused by high-criticality tasks are permitted, but not vice-versa. This can be realised as a system mode-switch or form of graceful degradation.

A key observation about mixed-criticality systems is neither the strictness of the real-time model, nor rate-monotonic priorities have any direct correlation with the criticality of a task. 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.

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.

Recall that in this model CAs provide WCET estimates that must be schedulable, however they are often very pessimistic. This results in a task with two WCET estimates, one very pessimistic one from the CA and a less pessimistic one from the system designers or automated tools. 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.

Scheduling algorithms in this class are distinguished by a mode-switch between two or more criticality levels, which may result in low criticality tasks being dropped or de-prioritised 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., 2011],

3.2.1 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 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.

3.3 Resource sharing

Like the scheduling algorithms, the locking protocols presented in Section 2.3 do not work if tasks are untrusted: in all the protocols, if a task does not voluntarily release the resource, all other tasks sharing that resource will be blocked.

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 systems using it, this relationship need not be symmetric; low criticality systems should be able to use high criticality resources.

In this section we explore how resource reservations and real-time locking protocols interact, and asses their suitability for mixed criticality systems. As introduced in Section 4.3.3, when combining locking protocols and reservations one must consider 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 or PIP, SRP. Which reservation to charge for processing time consumed when accessing a shared resource, and when to charge, are more interesting. 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.

Of most relevant to mixed-criticality systems, where tasks cannot be guaranteed to unlock a resource, is the enforcement mechanism. Many protocols rely on tasks being trusted with *a priori* knowledge of a task's resource usage. However, in systems where tasks may not be trusted (either due to security, certification level, or potential bugs) such *a priori* knowledge is unavailable.

Brandenburg [2014] outlines a multiprocessor IPC based protocol 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.

3.4 Summary

Traditional scheduling algorithms, like EDF and FP-RM schedule processor time but do not consider criticality differences between tasks, and also trust all tasks to stay within their WCET. Due to pessimism in WCET estimates, this results in low utilisation in such systems and also prevents systems of mixed-criticality being constructed safely and securely.

Mixed-criticality systems require at minimum temporal isolation, however the asymmetric protection property allows for higher utilisation in systems. EDF and FP scheduling

can be adapted to have temporal isolation, or another approach, like PFair scheduling can be used to provide temporal isolation. Much research has been done into scheduling algorithms that provide asymmetric protection, but the consequences of practical implementations in OSes have yet to be explored.

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 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 (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-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.

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.

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 have surveyed three standards: 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 an exact scheduling details to be provided about the task and resource. ARINC alone provides

real temporal isolation for tasks sharing resources, however like its scheduler, 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 of 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 safety-criticality systems, 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 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 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, in addition to *PFair* schedulers. A sporadic server (SS) implementation exists, as well as various multicore, real-time locking protocols.

Linux/RK [Oikawa and Rajkumar, 1998] is a resource kernel implementation of Linux with scheduling, admission control, and enforcement in the kernel. Every resource, memory, CPU time and devices was time-multiplexed using a recurrence period, processing time and deadline. Reservations in Linux/RK could be hard, firm of soft, which altered resource scheduling after a resource was exhausted. Hard reservations were not scheduled again until replenishment, firm would only be scheduled if no other undepleted reserve or unreserved resource use was scheduled, while soft allowed 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 3.2.1.

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 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 [ARINC] 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, with 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.

Although temporal isolation is sometimes provided with the possibility of bounded bandwidth via CBS and 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 2.5 and are as follows:

- 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.
- 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 with similar with goals to a microkernel, however with a more dominant focus on support for fine-grained components, and massive scalability.
- FiascoOC, a third-generation iteration of Fiasco, both microkernel and hypervisor.
- 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*.
- 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.

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.

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 [Bomberger et al., 1992]. 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 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 . 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 [Danish et al., 2011] 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 of 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 the testing is done dynamically, 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 or 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 effect 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 of 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.

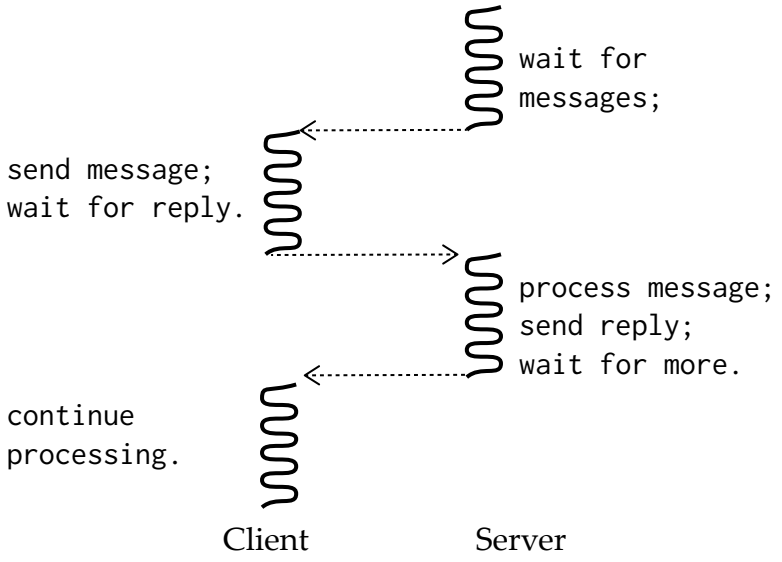


Figure 4.1: Thread execution during IPC between client and server

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 2.5.5 that prioritisation (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 2.5.2 that in a microkernel, OS services are implemented at user-level where clients use IPC to communicate with servers, in an remote procedure call (RPC)-like fashion, 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 clients timeslice is used for the start of the request, the servers 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 thread control block (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 and Kauer, 2010].

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 servers 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 et al., 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: 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 2.5.1 that capabilities [Dennis and Van Horn, 1966] are an established mechanism for fine-grained access control to system resources. Third-generation microker-

nels 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: it where a page of memory is, 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 up calls 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, 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 operation systems, commercial and open-source real-time operation 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 the vast majority of OSes provide FP. FP is far more dominant due to its inclusion in POSIX and 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 the 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.

In the next chapter we present a model for mixed-criticality scheduling that is suitable for high assurance systems such as seL4.

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 (`cspace` s) 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* `cnode` s which contain capabilities, analogous to page tables in virtual memory, and can contain capabilities to further `cnode` s, 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*.

Operation	*Description*
`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.
`Mint`	Like copy, except the new capability may have diminished rights and/or be badged.
`Move`	Move a capability from one slot to another slot, leaving the previous slot empty.
`Mutate`	Like move, except the new capability may have diminished rights and/or be badged.
`Rotate`	Atomically move two capabilities between three specified slots.
`Delete`	Remove a capability from a slot.
`Revoke`	Delete any capabilities derived from this capability.
`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]

.

Each capability has three potential access rights: read, right 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 `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 7.2, are distinguishable by system call number. The majority of system calls are for communication; `send`, `nbsend`, `call`, `reply` are for sending messages; `recv`, `nbrecv` for receiving messages; and `yield` 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.

System call	*Parameter*	*Description*	*Return?*
`send, nbsend`	`dest`	Capability to invoke.	✗
	`info`	Description of message.	
`call`	`dest`	Capability to invoke.	
	`info`	Description of message.	✓
`recv, nbrecv`	`src`	Capability to block on.	✓
	`badge`	Destination for badge.	
`reply`	`info`	Description of reply message.	✗
`replyrecv`	`info`	Description of reply message.	✓
	`src`	Capability to block on.	
	`info`	Description of reply message.	
	`badge`	Destination for badge.	
`yield`	✗	✗	

Table 5.2: seL4 system call summary. All system calls except `yield` are based on sending and/or receiving messages. The *return* column indicates if a system call returns a message or not.

All the operations on capabilities are that are listed in Table 5.1 are invocations on `cnode` capability addresses. For example, to copy a capability, one uses `call` on a `cnode`, and provides the invocation code for copy, as well as the arguments. In the case of 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, but also essential for understanding the complexity of kernel design. 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.

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 memory 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

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]

.

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 in 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.

Table 5.4 shows an example initial memory layout for the SABRE platform, 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.

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.

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.

5.3.5 Notifications

Notification objects are an array of binary semaphores used to facilitate asynchronous communication in seL4, either from other threads via `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.

Operation	*Description*
`Resume`	Place a thread in the scheduler.
`Suspend`	Remove a thread from the scheduler.
`WriteRegisters`	Configure a thread's execution context.
`ReadRegisters`	Read a thread's execution context.
`SetAffinity`	Set the CPU on which this thread should run.
`SetIPCBuffer`	Set the page to use for the IPC buffer.
`SetPriority`	Set the priority of this 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.

.

5.3.6 Consequences

User-level management of kernel memory provides true policy freedom—there is no policy required by the kernel on memory layout—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 on 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 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 tasks 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 `irq_control` capability is the control capability for obtaining capabilities to specific interrupt numbers. By invoking the `irq_control` 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, `irq_control` is also used to provide capabilities to model specific registers and I/O interrupts.

Slot(s)	*Contents*	*Slot(s)*	*Contents*
0	empty	8	`IO_space_control`
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	`domain_control`
4	`irq_control`	12—18	initial thread's paging structures
5	`ASID_control`	19—1431	initial thread's frames
6	initial thread's ASID pool	1431—1591	untyped
7	`IO_port_control`	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.

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 7.2, 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 Table 5.7 summarises seL4 system calls when used on endpoint capabilities. Essentially, `send()` is used to send a message and `recv()` used to receive one, both having blocking and non-blocking variants. `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 `send()` and `recv()` by *resume* capabilities. Pioneered in EROS [Shapiro et al., 1999], resume capabilities are generated once the message sent by `call()` is consumed, and consist of the thread waiting for a reply message. In seL4, the resume capability is stored in the TCB `cnode`, and `reply()` is the operation used to invoke this capability and send the reply message back.

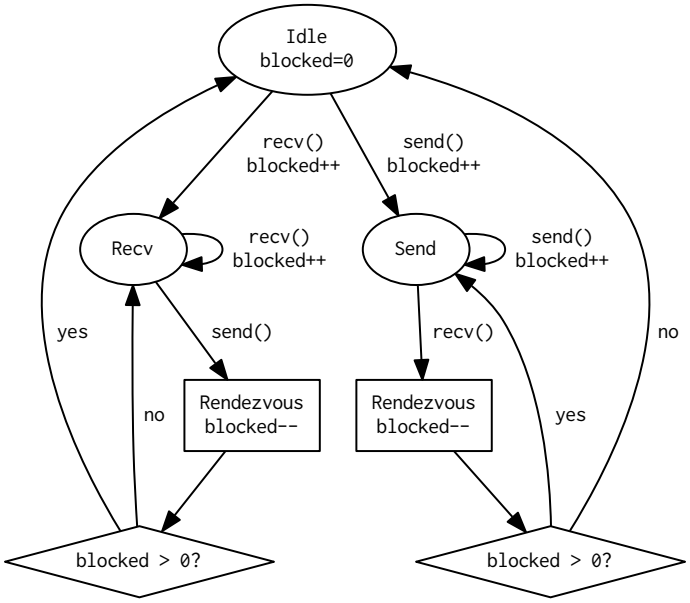


Figure 5.1: State diagram of a single endpoint, where *blocked* tracks the number of threads waiting to send or receive. Note that only one list of threads is maintained by the endpoint: senders and receivers cannot be queued at the same time.

System call	*Action*
`send`	Send a message, blocking until it is consumed.
`nbsend`	Send a message, but only if it is consumed immediately (i.e. a thread is already waiting on this endpoint for a message).
`recv`	Block until a message is available to be consumed from this endpoint.
`nbrecv`	Poll for a message—consume a message from this endpoint, but only if it is available immediately.
`call`	Send a message, and block until a reply message is received.
`reply`	Send a reply message to a `call`.
`replyrecv`	Send a reply message to a `call` and then `recv`.

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

Figure 5.3a demonstrates the rendezvous phase, where regardless of the order of operations, when one thread blocks (`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 later, but otherwise the resume capability is installed in the TCB CNode. The *reply* system call directly invokes the resume capability in this slot.

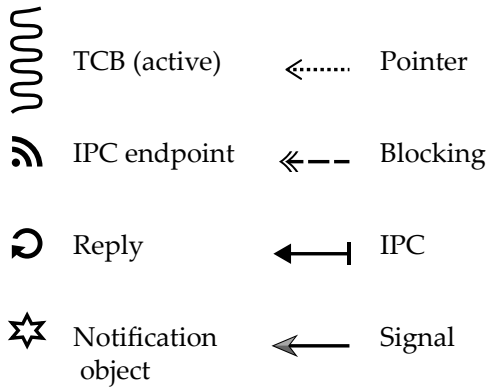
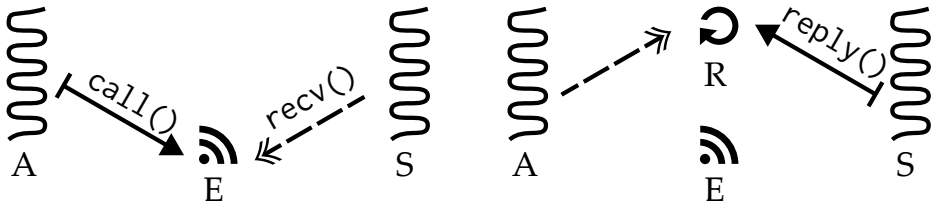


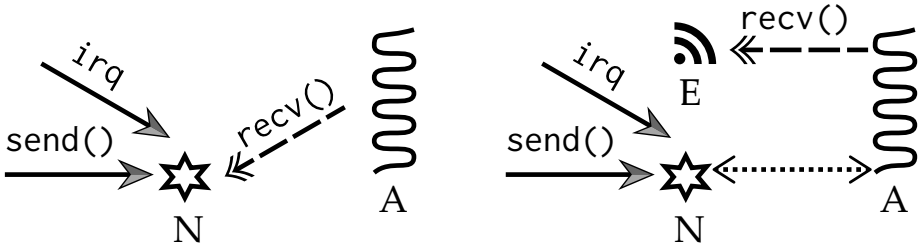
Figure 5.2: Legend for diagrams in this section



(a) Initial IPC rendezvous

(b) Reply phase

Figure 5.3: IPC phases: a TCB, *A* sends a message to endpoint *E* using `call()`. Another TCB, *S*, blocks on *E* using `recv()`. At this point the message is transferred from *A* to *S* and *A* is blocked the reply capability. (b) shows the reply phase, where *S* uses `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 the *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.

Multiple senders and receivers can use the same endpoint, and 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.

5.5.2 Fastpath

Recall from Section 2.5.2 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 is used when the following, common-case conditions are satisfied:

1. the sender is using `call()` or the receiver is using `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 in registers.

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 `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 `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.

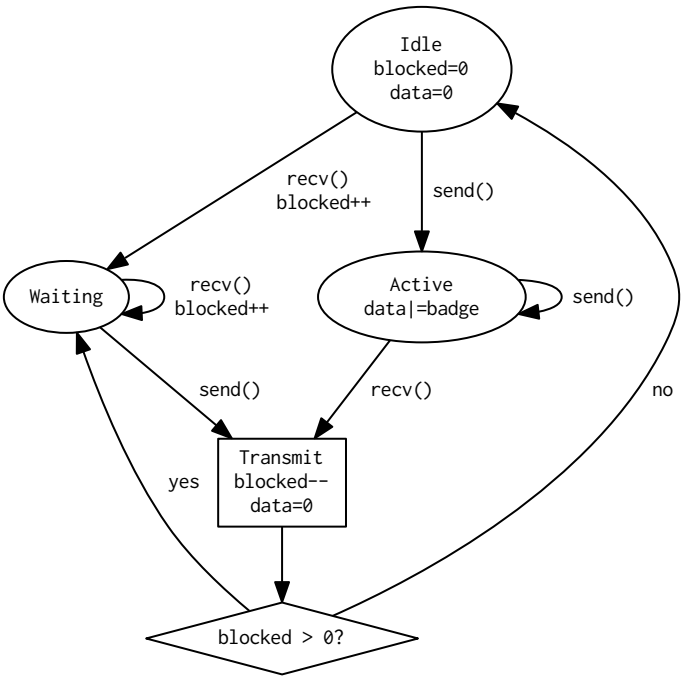


Figure 5.5: State diagram Notification object state transitions based on invocations. send and nbsend correspond to notify in the diagram, want wait corresponds to a recv.

System call	*Action*
send	Send a notification, transmitting the badge; do not block.
nbsend	As above.
recv	Wait until a notification is available then receive the data word.
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 endpoints.

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 `set_notification`, 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 `acknowledge` invocation on IRQHandler capabilities.

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 message 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 `bind_notification` 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 $O(1)$.

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. The second level 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 2 shows the logic for the scheduler bit field on 32-bit systems.

Listing 2 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.

```
uint32_t top_level_index = 0;
uint32_t bottom_level_index[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);
}
```

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

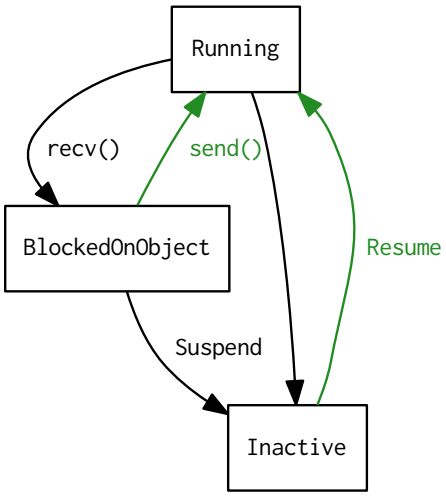


Figure 5.6: Thread state transitions in seL4

`BlockedOnObject` 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 `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 `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 thread are in the scheduler, to 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.

Round-robin

Kernel time is accounted for in ticks, and each TCB has a timeslice field which represents the number of ticks they are eligible to run. This value is decremented every time a timer tick is handled, 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 `yield` system call, which places the thread at the end of the scheduler queue.

When threads communicate with a service over IPC, the TCB of the thread providing the service is charged for any ticks that occur while that service is active. As a result isolation in shared server 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 `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 effect timing behaviour in the kernel: the scheduler, domains, priorities, `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.

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.

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

Table 5.9: Comparison of existing seL4 scheduling options.

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 support 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 real-time strictness (HRT, SRT, best effort), different criticalities, and different levels of security. 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, whose mechanisms we apply to our research platform, seL4. However temporal isolation is not enough: 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 section we first address how we integrate resource kernel mechanisms with seL4 with mechanisms for temporal isolation. We then explain priority decoupling and our model for temporal isolation in shared resources, with clients of different criticalities, temporal sensitivity, and trust.

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 over 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 where a temporal failure in one component cannot cause a temporal failure in another part of the system, even in the case of shared resources.

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. Overcommitment is also key to providing asymmetric protection, where high criticality tasks can disrupt low criticality tasks.

Safe resource sharing: mechanisms for sharing between applications of time sensitivity and criticality.

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

We outlined four core resource kernel mechanisms—admission, scheduling, enforcement and accounting—that are essential to resource kernels for implementing temporal isolation. However, as noted in Section 4.3.3, 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 a systems required functionality. Existing literature is divided as to whether resource kernel mechanisms should be provided by the kernel or at user-level.

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 examine how each of the four resource kernel mechanisms are implemented in 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, unlike COMPOSITE, for two reasons: to maintain a small trusted computing base, and for performance. Any system with multiple threads of execution, which is required in a mixed-criticality system, must have a scheduler, which for the purposes of temporal isolation is part of the trusted computing base. If a system must have a scheduler, and in a mixed-criticality system, with untrusted components, that scheduler must be in a separate protection domain to those untrusted components, then a scheduling decision will require at least two context switches: from the preempted thread to the scheduler, and from the scheduler to the chosen thread. Placing a basic scheduler in the kernel reduces this overhead.

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 maintaining a scheduler in the kernel is a violation of 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 EDF's dynamic priorities, as we will demonstrate in Section 8.5.1. 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.

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 core 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 RMS 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 subsection 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, with fixed priority the result 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, including when the system criticality changes.

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. A 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 communication endpoints (ports). An SC is represented by a capability to a scheduling context object (scCap).

In order to run, a thread needs an scCap, which represents the maximum CPU bandwidth the thread can consume. 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, which aligns with our efficiency goal; 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 a *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 that is used on many systems (including present mainline seL4).

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

SCs can be gracefully integrated into the existing model used in seL4, logically replacing the time slice attribute with the scCap.

Unlike Fiasco's scheduling contexts [Lackorzyński et al., 2012], which are superficially similar to ours, we retain priorities as a thread attribute rather than making them part of SCs.[1] The advantage of keeping the two orthogonal will become evident in Section 6.2.

6.1.3 Accounting

Accounting must be done by the kernel – it's the only one that can do it for performance reasons.

We introduce a new 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. Any time executed on a processor is billed to the current threads scheduling context. 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 Enforcement

Our model offers a single mechanism for enforcement: threads can have an optional timeout fault endpoint. If a timeout fault endpoint is provided, a fault message is sent to that endpoint should that thread exhaust its budget, allowing the system to enact an enforcement policy. Threads without a timeout fault endpoint are not schedulable until their budget is replenished.

The timeout fault endpoint is separate from the existing thread fault endpoint, as the semantics are different. For the existing fault handler threads block if a fault occurs if no fault handling endpoint exists. In the case of a timeout fault handler, the thread remains runnable and is only blocked if a fault message is delivered to the timeout fault endpoint. Otherwise, the kernel will not block the thread but as it can run again once its budget is replenished.

Timeout fault handling threads must have their own SC to run on, with enough budget to handle the fault.

[1] Fiasco's scheduling contexts were developed for an old API version that was not capability-based, and it is not obvious how they would integrate with the capability system of present Fiasco.OC.

6.1.5 Admission control

Setting budgets is admission control and requires appropriate privilege. We introduce a new control capability `sched_control` per core, which gives authority to 100% of time on a processing core. Access to this capability provides 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 the 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.

6.1.6 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 $C = T =$ 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. However, a full resource kernel can be constructed on top of the mechanisms our model provides. ***TODO: describe how to implement this***

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, e.g. priority exchange servers and slack stealing. Deferrable servers do not require global state but are avoided due to the back-to-back problem.

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.

There is an obvious cost to replenishment fragmentation that will arise from preemptions, and polling servers are more efficient in the case of frequent preemption [Li et al., 2014]; an arbitrarily high threshold makes little sense. 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, as policy which can easily be implemented at user-level with variably sized SCs.

6.1.7 Budget overrun

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 a system-criticality change.

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. 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.

Similar to a page fault handler, timeout fault handlers can be used to adjust thread parameters dynamically as may be required for a 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,
- killing/suspending the thread, and
- changing the system criticality level.

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

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.

6.1.8 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 as 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 have real-time threads running at the EDF-priority, where they will be temporally contained, with non-real time threads running at lower priorities. Many other combinations are possible.

6.1.9 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 provide asymmetric 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 of processing time at high priorities, and low criticality tasks given reservations at a lower priority band.

RTA relies on suspending low criticality tasks if a high criticality task runs for longer than expected. In general, RTA schemes involve two system modes: a HI mode and a LO mode. 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. This is also supported by our model: 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 extend the reservation of the high priority task and suspend all low priority tasks.

6.1.10 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

Bringing mixed-criticality to resource sharing is important for future cyber-physical systems. We present our model for resource sharing in terms of the three core mechanisms taken from resource kernels—prioritisation, charging and enforcement—presented in Section 4.3.3.

We consider shared resources as separate threads accessed via synchronous IPC, as this mechanism ultimately means changing which thread is currently running on the processor, and thus changing where processing time is being spent. Note that we focus our analysis on shared servers where the server doesn't trust its clients, and the clients do 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`. Clients on the other hand must trust the server if they share resources over synchronous IPC, as the server can choose to never reply to the client.

Access to shared servers encapsulating shared resources requires cross-address-space IPC. For minimising invocation cost, it is essential that scheduler invocations are avoided during IPC. This has historically been done by L4 microkernels [Heiser and Elphinstone, 2016]. Not bypassing the scheduler is the main reason why IPC on most other systems is significantly slower than that of L4 kernel, e.g. at least a factor of four in CertiKOS [Gu et al., 2016].

Past L4 kernels avoided the scheduler by time-slice donation, where a server could execute on the client's time slice. While fast, this model is unprincipled and hard, if not impossible, to analyse. For example, the time slice may expire during the server's execution, after which the server will run on its own time slice. This results in no proper accounting of the server's execution, and no temporal isolation, and relates to the precision problems discuss in **??**.

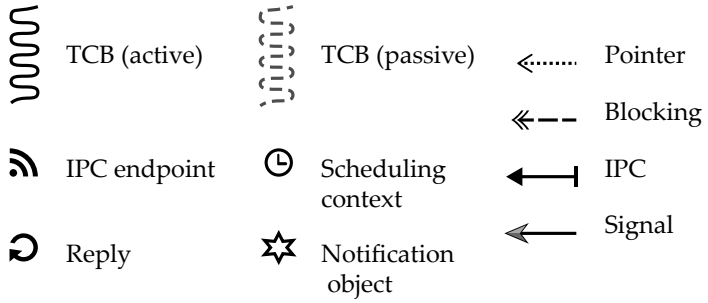


Figure 6.1: Legend for diagrams in this section, an expanded version of Fig. 5.2

The proposed model supports shared servers, including scheduler bypass, in a principled way 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 the PIP is enforced by the kernel.

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 servers priority is used.

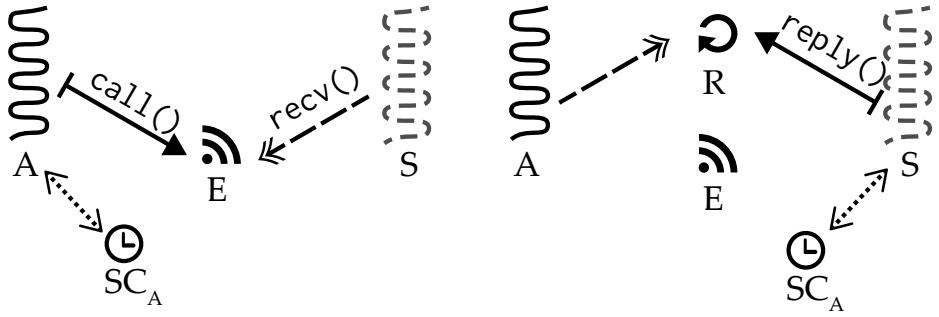
The main drawback of IPCP, namely the requirement that all lockers' priorities are known *a priori*, is easy to enforce in a capability-based system: The server can only be accessed through an appropriate invocation capability, and it is up to the system designer to ensure that such a capability can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

The choice of IPCP is intentional: if the scheduling context holds the priority, then PIP is forced by the mechanism. Since PIP has a known high preemption overhead, this would violate our criteria of policy freedom and efficiency. Although OPCP offers greater processor utilisation than IPCP, we consider its use impractical as it requires too much system state to be drawn into the kernel: the kernel must track a system ceiling and implement priority inheritance.

However PIP and OPCP can used with this mechanism albeit in a more complex fashion, by proxying requests to shared resources via a scheduling server which manipulates the priorities appropriately. For PIP, only one server 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.

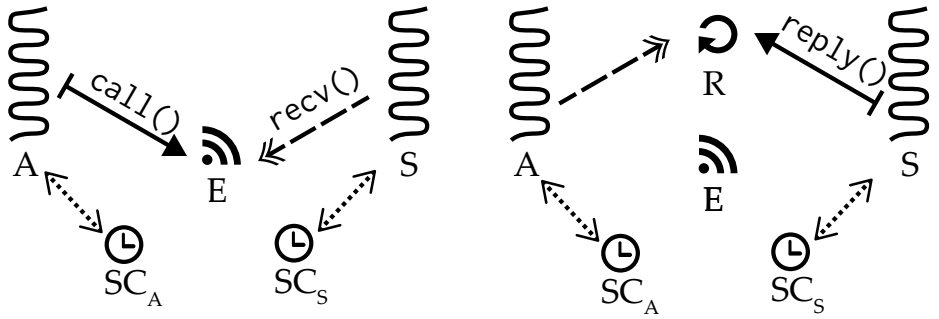
6.2.2 Charging

Unlike prior implementations of scheduling contexts discussed in Section 4.3.4, donation is not compulsory, which provides more policy freedom than prior models. However it is



(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.

also not optional: instead we infer whether donation is required by testing 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 Section 6.2.2, effectively provide a migrating-thread model [Ford and Lepreau, 1994; Gabber et al., 1999], but without requiring the kernel to manage stacks. Section 6.2.2 shows active servers, which allow system designers to build systems without temporal isolation, as it is not suitable for all systems.

Charging for time executed in a server becomes simple with this model: always charge the scheduling context the server is executing on. The mechanism does not change regardless of the server being passive or active.

6.2.3 Enforcement

If a passive server exhausts its budget, it and any waiting clients are blocked until the budget is replenished. 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.

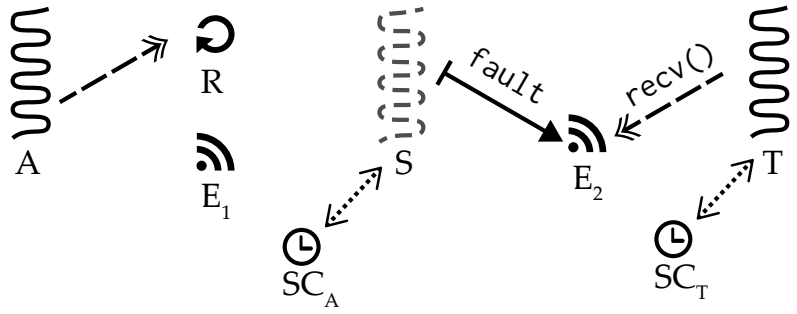


Figure 6.4: Example of a timeout exception handler. The passive server S performs a request on client A's scheduling context, SC_A. A timeout fault is generated when SC_A is exhausted and send to the servers timeout fault endpoint E_2, and the timeout handler receives it. See legend Fig. 5.2

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 for the largest budget of the other clients (plus the short cleanup time).

6.2.4 Cross-core IPC

TODO: diagram of cross core IPC

Scheduling-context donation extends naturally to IPC and allows users to specify active servers on remote cores, or passive servers which migrate to the local core. The mechanism is simple: if a server is passive, it migrates to the core that the donated scheduling context is on.

6.2.5 Resource sharing policies

We now describe three policies for resource sharing over IPC — best-effort, helping and isolation — and describe how they can be implemented with our mechanisms of scheduling contexts, scheduling-context donation and timeout exceptions.

TODO: A diagram of each of these policies TODO: Finish this section

Best effort

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. Server threads are all active, and at 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 servers 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 lax temporal requirements.

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 servers 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 clients budget expiring while the server is in a critical section, which is solved by providing atomic primitives that call the kernel.

While our model does not require this, it is possible to create a server which allocates execution contexts should the existing server threads all be busy. Server threads would wait on an endpoint, and a stack spawning thread would wait on the same endpoint with low-priority. The stack spawning thread is only woken if all the server threads are busy, and otherwise does not run. An alternative policy is to build systems with a fixed number of server threads.

Like the best-effort policy, our model supports this approach but it is not required.

Server isolation

TODO: Address peters comment: This is essentially an illustration for the preceding section, so title it as such

In a mixed-criticality system, clients of different criticalities may share a server such as an encryption service. Consider such a service which encrypts data in blocks. Clients can request arbitrary length blocks to be encrypted. Using a passive server, the server can work in blocks, and use a transaction-based approach to alter between clean and dirty state. If a timeout exception occurs, because a clients scheduling context has expired, the timeout exception handler can rollback to the last clean state and reply to the client with the amount of data encrypted. The rollback has a known complexity, so any pending requests need

only to be able to cope with the cost of the rollback, assuming the scheduling contexts are configured with a schedulable time allocation.

6.3 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 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 the requirement of PIP, and by differentiating between passive and active servers we 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 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

TODO: Finish this chapter

In the previous chapter we presented our model for for a microkernel. This section provides the details of the implementation in seL4.

Kernel changes include additions to the scheduler to support temporal isolation, scheduling contexts objects, system call API changes and a new mechanism—scheduling context donation—which supports bandwidth inheritance.

These changes allow user-level to construct systems with combinations of best-effort and real-time threads. We illustrate through example how threads can achieve resource sharing through reservation-per-thread, migrating threads or scheduling context donation policies.

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. All three objects interact to implement our mechanisms.

7.1.1 Resume objects

EROS also introduced the idea of a single-use reply capability, termed a *resume* capability, which when invoked would be consumed and allows the receiver to reply to the sender. A resume capability is like a temporal capability, in that it grants authority to run a thread on the processor, however without an upper bound on execution.

Resume objects, modelled after KeyKOS [Bomberger et al., 1992], are a new object type that generalise "reply capabilities" of baseline seL4 introduced in **??**. 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 representing the SC donation chain). They 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 7.5.

With SC donation we need slightly more bookkeeping to guarantee that a donated SC eventually returns to its rightful owner, even if the server invokes another passive server, or the server operation is long-running and the server handles multiple requests concurrently (as a file server would). Resume objects are used to keep track of the donation chain, details are in **??**.

Resume objects are small (16 bytes on 32-bit platforms, and 32 bytes on 64-bit platforms) and contain the following:

TCB a pointer to the TCB that is blocked on this reply object. Depending on the thread state this is either:

BlockedOnReply a pointer to the caller that is blocked on this resume object after performing a `Call`.

BlockedOnRecv a pointer to the receiver that is blocked on an endpoint with this resume object.

Prev,Next pointers which track the call chain. The head of the call chain is always the SCO which itself contains a pointer to the TCB using the SCO. Each reply object then points to the TCB which donated the SCO along the chain.

TODO: Diagram of reply objects and call chain.

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 must have a 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.

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

contain sporadic task parameters. Scheduling contexts encapsulate processor time reservations, derived from sporadic task parameters: min-period (T) and a set of replenishments which is populated from an original execution budget (C), representing the reserved rate reserved rate (U) = $\frac{C}{T}$. The execution budget will be populated with the WCET for HRT tasks, and a max-rate for SRT and rate-based tasks.

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 last resume object (see Section 7.1.1) in a call chain where this scheduling context was donated.

Notification A pointer (if any) to the notification object this SCO is bound to.

Badge A unique 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 yielded to this SCO.

Head,Tail Indexes to the circular buffer of sporadic replenishments.

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 7.1: Layout of a scheduling context object on a 34-bit system.

Max Maximum amount of replenishments for this scheduling context.

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

Sporadic threads are more complex; the amount of replenishments is limited by both a variable provided on configuration and the size of the SCO, configured on creation. This allows system designers to control the preemption and fragmentation of sporadic replenishments as discussed in Section 6.1.6.

Scheduling contexts are connected to one thread at a time. A thread is not permitted to execute for more time than that represented in the scheduling context. Threads without scheduling contexts are not runnable.

Scheduling contexts also form part of the accounting mechanism: the amount of budget a thread has remaining, and when the budget is due to be recharged, are stored in the scheduling contexts. **??** displays a summary of the main fields stored in a scheduling context object.

The method `seL4_SchedControl_Configure` is used to set the parameters on a specific scheduling context. It taskes a `budget`, `period`, `num_extra_refills` and `badge`.

7.1.3 Thread control blocks

TCBs are the abstraction of an execution context in seL4. We add the following additional fields to the TCB object for our implementation:

Scheduling context The scheduling context (if any) this TCB consumes time from.

maximum controlled priority (MCP) The MCP of this TCB.

Reply A pointer to the reply object this TCB is blocked on, if blocked.

TODO: And we changed how fault handlers work

System call	*Parameter*	*Description*	*Return?*	*Donation?*
`send, nbsend`	`dest`	Capability to invoke.	✗	✗
	`info`	Description of message.		
`call`	`dest`	Capability to invoke.		
	`info`	Description of message.	✓	✓
`recv, nbrecv`	`src`	Capability to wait on.	✓	✓
	`badge`	Destination for badge.		
	`reply`	Reply object to block callers on.		
`wait, nbwait`	`src`	Capability to wait on.	✓	✗
	`badge`	Destination for badge.		
`replyrecv`	`src`	Capability to wait on.	✓	✓
	`info`	Description of reply message.		
	`badge`	Destination for badge.		
	`reply`	Reply object to invoke, then block callers on.		
`nbsendrecv`	`dest`	Capability to invoke.	✓	✓
	`info`	Description of reply message.		
	`src`	Capability to wait on.		
	`badge`	Destination for badge.		
	`reply`	Reply object to block callers on.		
`nbsendwait`	`dest`	Capability to invoke.	✓	✗
	`info`	Description of message.		
	`src`	Capability to wait on.		
	`badge`	Destination for badge.		
`yield`	✗	✗	✗	✗

Table 7.2: seL4 system call summary. All system calls except `yield` are based on sending and/or receiving messages. The *return* column indicates if a system call returns a message or not.

7.2 System calls

7.3 Scheduling

Now we discuss changes to the kernel scheduler and how the new SCOs interact which the scheduling algorithm to provide temporal isolation and asymmetric protection. We show how best-effort and real-time threads are supported by these changes, and consider *rate-based* threads—which are essentially best-effort threads with a kernel-enforced bound on maximum rate.

We make changes to the scheduler and structure of the scheduler to support temporal isolation: we convert the scheduler to tickless, add a release queue of threads waiting for

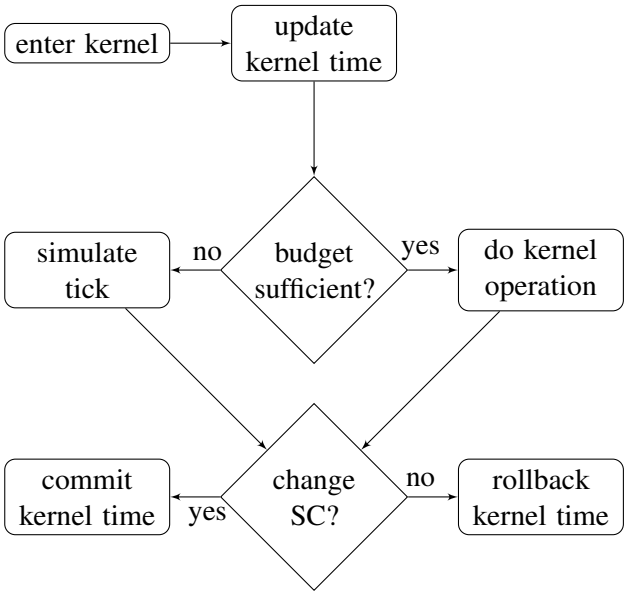


Figure 7.1: Kernel structure

budget replenishment and introduce the invariant that threads without scheduling contexts are not eligible for scheduling.

7.3.1 Tickless

There are two ways to account for time in a kernel:

tick-based in fixed time quanta, referred to as *ticks*,

tickless in cycles between specific events.

In a tick-based kernel, timer interrupts are set for a periodic tick and are handled even if no kernel operation is required. This approach has advantages: the timer in the kernel is stateless and never has to be reprogrammed, rendering it very simple, fast and easy to verify. However, such simplicity comes with a cost of reduced precision and greater preemption overhead. The greater the desired precision in a tick-based kernel, the smaller the size of the tick quanta, and the higher the preemption overhead, and vice-versa. Additionally,

Tickless kernels set timer interrupts for the exact time of the next event. This approach affords greater precision and results in no more preemption overhead than is required by the workload. A tickless model offers less overheads and more precision to real-time tasks, which suffer WCET penalties for every preemption. Additionally, we show in show in Section 8.2.1, that the cost of reprogramming a timer has significantly decreased on modern hardware.

Therefore, to increase precision and reduce interrupt overhead, we convert seL4 from a tick-based to a tickless kernel in order to reduce preemptions and improve scheduling precision, as discussed in **??**. As seL4 is non-preemptible (save for explicit preemption points in a few long-running operations), tickless kernel design is non-trivial, as preemption interrupts cannot interrupt the kernel itself.

7.3.2 Algorithm

The main change required to the existing scheduler is the addition of a *release queue* per processing core. If a preempted thread does not have any available replenishments, the kernel removes the thread from the ready queue to the release queue, retaining the invariant for the ready queue, which the release queue is charaterised as holding all threads that would be runnable but are presentingly out of budget. The queue is ordered by the timer when the next replenishment is available.

On kernel entry (except on the IPC fastpath, which never leads to an SC change or scheduler invocation) the kernel updates the current timestamp and stores the time since the last entry. This is required as when preemption occurs, the preemptor is charged for the time since the kernel entry. Without knowing if the kernel entry will trigger a thread switch in advance, the kernel must record the time for each entry. However, if the recorded timestamp is not acted upon, the time is rolled back to the previously recorded value.

After recording the timestamp, the kernel then checks whether the thread has sufficient budget to complete the kernel operation, using a fixed estimate of double the kernels WCET. 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

This adds a new invariant, that any thread in the scheduling queues must have enough budget to exit the kernel. It makes the scheduler precision equal twice the kernel's WCET, which for seL4 is known (unlike any other protected-mode OS we are aware of). This invariant is required as it simplifies the kernel design and actually minimises the WCET: when a thread runs out of time it may need to raise a timeout exception resulting in delivering an IPC to a timeout handler.By requiring that anything in the scheduler queue, or any endpoint queue, must have enough budget to wake up we avoid needing to potentially raise timeout exceptions on many wakeup paths in the kernel.

Threads are only charged when the scheduling context changes, in order to avoid reprogramming the timer which can be expensive on many platforms. If there is no SC change, the timestamp update is rolled back by subtracting the stored consumed value from the timestamp. Figure 7.1 illustrates the structure of this kernel design.

7.3.3 Priority queues

Priority queues for both scheduling algorithms are implemented as ordered lists, with $O(n)$ insertion and removal complexity. The most frequent operation on the lists is to remove the head, which is $O(1)$. We choose a list 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, in order to provide isolation and confidentiality, seL4 kernel memory is managed at user-level, as discussed in Consequently, to put a seL4 kernel object into a heap, the pointers for the heap implementation must be contained within the object, which could be anywhere in memory as chosen by the user. This means that seL4 heap implementations are non-contiguous, and must be pointer based, resulting in a much larger cache footprint with worse performance than an array-based approach.

verification. Of course, even if the heap implementation is slower, given a sufficient number of tasks a heap will scale better than a list. However, we do not expect systems to run a large amount of real-time tasks, as seL4 target applications generally run virtualised Linux along-side a few critical real-time tasks. Additionally, the reduced complexity of a list compared to a heap will result in faster verification, so consequently a list implementation looks favourable.

7.3.4 Admission

As established in section Section 6.1.5, admission tests for reservations are considered policy to be implemented by user-level.

In the current design, we control the creation of scheduling contexts by conveying the authority to populate their parameters into a single capability per processor. Each processor has a scheduling control capability, all of which are given to the root task on initialisation. Scheduling contexts can be created by any task, using standard seL4 conventions to create an object, which result in a scheduling context with all parameters are set to 0. The implication is that tasks in the system with access to memory can only create empty reservations. While an empty reservation can be bound to a thread, since the reservation contains no budget that thread will never be eligible to run.

In order to populate the parameters of a scheduling context, one must invoke a new capability: the scheduling control capability for the core that the thread is intended to run on. Only a single copy of this capability is available per processor in the system, so the population of scheduling contexts with parameters is restricted to processes with access to those capabilities, which can conduct admission tests as per user-level policy.

7.3.5 Scheduling Contexts

7.3.6 Replenishments

Scheduling contexts contain a circular buffer for sporadic task replenishments. Each replenishment has an amount of time that stands to be replenished, and the absolute time from when that replenishment can be used, as shown in **??**. When `seL4_SchedControl_Configure` is called on an inactive scheduling context, the amount is set to the budget and the replenishment time to the current time. At all times, the sum of the amounts in the replenishment buffer is equal to the configured budget. The maximum size of this buffer is statically configurable, and the minimum size is one. Users can specify the amount of extra refills a scheduling context can have, up to the static maximum. This extends the approach used by Quest-V [Danish et al., 2011], which has as static limit of 32 replenishments

Scheduling contexts with zero extra refills behave like polling servers (Section 3.1.3), otherwise they behave as sporadic servers (Section 3.1.3), allowing application developers to tune the behaviour of threads depending on their preemption levels and execution durations.

The algorithms to manage 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` as shown in Listing 3 is called to bill the amount of time consumed since the last scheduling context change. ‘check_budget‘ If the budget is not completely consumed by `check_budget`, `split_budget` as shown in Listing 4 is called to schedule the subseqeunt 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. The

scheduling context being switched to has `unblock_check` Listing (5) called on it, which merges an replenishments that are already available, avoiding unneccessary preemptions.

When `seL4_SchedControl_Configure` is called on an active scheduling context, the refills are adjusted to reflect the new budget and period but respect the sliding window constraint.

Round-robin threads have full reservations: $T = C$, entitling them to 100% of the processor but subject to preemption every time C is used. Clearly we don't want to track replenishments for round-robin threads, so the kernel detects if round robin scheduling contexts when `seL4_SchedControl_Configure` is called, and sets the period to 0. This avoids much special casing in the replenishment code, as round-robin threads are always ready (we always add 0 to the replenishment time).

TODO: Say what head refill does

7.3.7 Task model

We extend the seL4 API with support for real-time and rate-based tasks, while maintaining support for round-robin, best-effort tasks. In this section we present how our mechanisms support each type of task, and how user-level should configure them.

Best-effort threads

Best effort threads, scheduled round-robin, are compatible with our model. In the original seL4 kernel, best effort threads would be scheduled for CONFIG_TIME_SLICE ticks, before being taken from the start of their run queue and placed at the end of the run queue. This was sufficient for round-robin scheduling.

Listing 3 Check budget routine.

```
1  uint_64_t check_budget(sched_context_t *sc, uint64_t usage) {
2    while (head_refill(sc).amount <= usage) {
3      // exhaust and schedule replenishment
4      old_head = pop_head(sc);
5      usage -= old_head.amount;
6      old_head.time += sc->period;
7      add_tail(sc, old_head)
8    }
9
10   /* handle budget overrun */
11   if (usage > 0 && sc->period > 0) {
12     // delay refill by overrun
13     head_refill(sc).time += usage;
14     // merge replenishments if time overlaps
15     if (refill_size(sc) > 1 &&
16         head_refill(sc).time + head_refill(sc).amount
17         >= refill_next(sc).time) {
18
19       refill_t old_head = pop_head(sc);
20       head_refill(sc).amount += old_head.amount;
21     }
22   }
23 }
```

Listing 4 Split check routine.

```
void split_check(sched_context_t *sc, uint64_t usage) {
  uint64_t remnant = head_refill(sc).amount - usage;
  if (remnant < MIN_BUDGET && refill_size(sc) == 1) {
    // delay entire replenishment
    // refill too small to use and nothing to merge with */
    head_refill(sc).time += sc->period;
    return;
  }

  if (refill_size(sc) == sc->refill_max || remnant < MIN_BUDGET) {
    // merge remnant - out of space or its too small
    pop_head(sc);
    head_refill(sc).amount += remant;
  } else {
    // split the head refill
    head_refill(sc).amount = remnant;
  }

  // schedule the used amount
  refill_t split;
  split.amount = usage;
  split.time = head_refill(sc).time + sc->period;
  add_tail(sc, split);
}
```

Scheduling contexts for best-effort threads are assigned an equal budget and period, with the value set to the desired round-robin time slice for the thread. Since the budget and period of threads configured in this fashion are equal, when the budget expires it will be immediately replenished as the period has passed already. However, replenishment places a thread on the end of its run queue, thereby implementing round-robin scheduling, as any other thread in the queue will be scheduled first. This approach minimises the amount of code required and results in a fast scheduler, with no special cases for best-effort versus real-time threads. Of course, populating scheduling context parameters as such entitles best-effort threads to an effective 100% share of the CPU, so it should only be used at low priorities. Additionally, real-time threads and best-effort threads should not run at the same priority.

Rate-limited threads

Rate-limited threads simply have their scheduling contexts configured with parameters that express their desired 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.

Listing 5 Unblock check routine.

```
1  void unblock_check(sched_context_t *sc) {
2    if (!head_refill(sc).time > now)) {
3      return;
4    }
5
6    head_refill(sc).time = now;
7    // merge available replenishments
8    while (refill_size(sc) > 1) {
9      if (refill_next(sc).time < now + head_refill(sc).amount) {
10       refill_t old_head = pop_head(sc);
11       head_refill(sc).amount += old_head.amount;
12       head_refill(sc).time = now;
13     } else {
14       break;
15     }
16
17     if (head_refill(sc).amount < MIN_BUDGET) {
18       // second part of split_check can leave refills
19       // with less than MIN_BUDGET amount.
20       // detect them here and merge.
21       refill_t old_head = pop_head(sc);
22       head_refill(sc).amount += old_head.amount;
23     }
24 }
```

Real-time threads

The scheduling of real-time threads is more complicated than that of rate-limited or best-effort threads, as the concept of a sporadic job, including job release time and job completion, need to be supported by the kernel API.

We define the initial job release as when a thread is resumed: the available budget is set to the total budget in the reservation for that thread.

Job completion is more complicated. As described in section **??**, job completion occurs when a job blocks waiting for the next job to be released. Any budget left by the current job is released as slack into the system, which means the available reservation budget drops to 0. If a job is time-triggered, such that it only relies on time for job release, then next job will be released once the period has passed. If a job is event-triggered, then the next job should be released once the period has passed *and* an external event occurs, such as an interrupt.

Simply defining job completion as when a task blocks is not sufficient as jobs can also block for other reasons, like polling I/O, or waiting for an asynchronous server. Unintentionally completing a job by blocking is incorrect, as it would result in real-time threads receiving less processing time than they have reserved.

One design option we considered was to use the yield system call to complete the current job. However this approach would not enable a thread to receive a notification on job release, requiring more system calls to retrieve notifications.

Instead, we use asynchronous endpoints to implement sporadic jobs on seL4, which unifies jobs to be completed and release with notifications, reducing the amount of system calls required. Currently, asynchronous endpoints can be bound to a thread, which enforces a one-to-one relationship between the bound thread and the bound endpoint. The current

Listing 6 Example of a basic sporadic real-time task on sel4

```
1       for (;;) {
2           // job is released
3           doJob();
4           // job completes before deadline or is postponed by CBS
5           // sleep until next job is ready
6           seL4_Word badge;
7           seL4_Wait(bound_async_endpoint, &badge);
8       }
```

semantics of async endpoint binding allow threads to wait on a synchronous endpoint and receive notifications from their bound async endpoint at the same time. To simplify implementation, no thread but the bound thread is permitted to wait on the bound endpoint. However, in practice, threads only ever wait on a separate synchronous endpoint, not the bound endpoint. As a result, we use this operation to implement job completion.

The new semantics are simple: waiting on the bound asynchronous endpoint completes the current job. The next job is released when a notification arrives on the endpoint. If the budget has not been replenished at this point, the thread will not be runnable until it is. This design allows a simple distinction between standard blocking, and blocking to complete the current job. A code example of what real-time threads look like on seL4 is shown in Listing 6.

Time-triggered jobs have an additional semantic, as the kernel will send a notification to the asynchronous endpoint when the period passes to release the next job. This is a conscious design choice: it would be possible for users to implement time-triggered jobs using a user-level timer driver. In that case the kernel could treat all real-time jobs as event-triggered jobs. However, since the kernel at this point already contains a trusted timer driver to support the enforcement mechanism, it is impractical to require users to provide a second trusted timer driver for periodic, real-time threads. This also reduces overhead for time-triggered jobs, as less context switches are required for job release.

Best-effort and rate-limited threads in practise run one real-time job, as they never complete their current job. If a real-time job does not complete before its budget expires, then it will be rate-limited, preserving temporal isolation.

7.3.8 Compatibility with the domain scheduler

While the real-time amendments to the scheduler are compatible with the domain scheduler, either the domain scheduler or the real-time amendments should be used for real-time scheduling. This is because domains are non-preemptive and as a result can only be used for non-preemptive real-time scheduling, where domain parameters exactly match real-time scheduling parameters. Using more than one domain when preemptive real-time scheduling is will result in missed deadlines.

7.4 Resource Sharing

Thread communications in seL4 take place via the IPC mechanism, which we alter to support scheduling context donation. In this section we will address changes to the system calls used to send and receive IPC messages to implement scheduling context donation.

Field	*Description*
`tcb_t *tcb`	The calling thread that is blocked on this reply object.
`void *prev`	0 if this is the start of the call stack, otherwise points to the previous reply object in the call stack.
`void *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 chain. 0 if no scheduling context was passed along the chain.

Table 7.3: Fields in a reply object.

We then illustrate through example how reservation-per-thread, thread-migrating IPC and scheduling context donation can be built with the new API.

7.4.1 Endpoints

TODO: Partially repeats section 5.1.4 IPC in seL4 is conducted through endpoints, which do not denote a specific receiving or sending thread, but act as arbitrary communication endpoints. Any thread with an endpoint capability can send or receive messages on that capability: if two threads send and receive messages on the same endpoint, then a communication rendezvous occurs and a message is send.

Endpoints are either synchronous: sending a message on a synchronous endpoint blocks the sender until the message is received, and in the case of multiple messages a queue of messages to be processed forms on the endpoint. As a result, IPC over synchronous endpoints triggers a thread switch. Messages on asynchronous endpoints do not block the sender, and are combined instead of forming a queue. We use IPC on synchronous endpoints to donate scheduling contexts, while asynchronous communication is expected to occur between threads with their own reservations.

7.4.2 Reply objects

7.4.3 Donation semantics

A synchronous message in seL4 can be sent by using the following system calls on a synchronous endpoint capability:

- `Send` blocks until the message is received by another thread, then the sender continues execution.
- `NBSend` only performs the send if the receiver is already waiting, otherwise it fails (although the client cannot tell which behaviour occurred).
- `Call` sends a message to another thread and blocks until a reply is received back.

`Call` is a special case: when `Call` is executed the kernel manufactures a special, single-use capability (referred to as the reply cap) which the callee invokes to reply to the message. The presence of the reply capability guarantees that a blocking thread is present to receive a reply. The reply capability is actually a thread object, not the endpoint, so reply messages

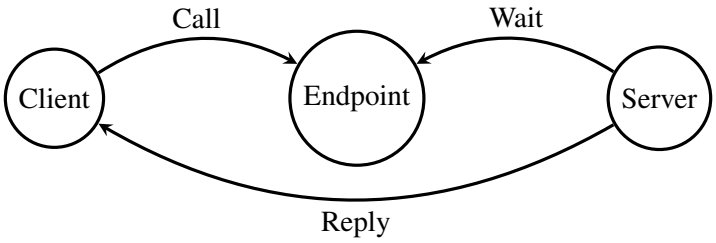


Figure 7.2: Client-server scenario with `Call` and `ReplyWait` and a synchronous endpoint.

are not conducted through an endpoint. Replies can be sent with the following two system calls:

- `Reply` sends a reply message and blocks the replying thread until the message is received.
- `ReplyWait` sends a reply and then blocks on an endpoint argument to the system call until another message is received.

To implement scheduling context donation, we augment the system call API as follows:

- `Call` (altered) between a caller that has a scheduling context and a callee that does not have a scheduling context donates the caller's scheduling context to the callee. If the callee has a scheduling context then donation does not occur. The callee runs at its own priority, as per HLP.
- `ReplyWait` (altered) to a thread without a scheduling context donates the scheduling context to subject of the reply.
- `SendWait` (new), which allows a thread to send a message and donate a scheduling context to an endpoint or reply cap, then wait on another endpoint.

Scheduling context donation does not occur on `Send`, `NBSend` or `Reply`, as the sender cannot continue to execute without a scheduling context, and without receiving one from another thread the sender is blocked. These system calls are not unusable, but should be used between threads who have their own scheduling contexts. If a thread attempts to use `Send`, `Reply` or `NBSend` to communicate to a thread without a scheduling context, the communication will block.

Figure 7.2 and Figure 7.3 illustrate the client-server and data-flow scenarios with endpoints, using `Call`, `ReplyWait` and `SendWait`. The cycle in the data-flow scenario is required in order to donate the scheduling context back to the source of the event: otherwise, after one iteration, phase 1 has no reservation budget to execute the next event on.

SendWait

Introducing `SendWait` into the kernel can lead to a long-running operation if the send part of the system call, when called on a synchronous endpoint, is allowed to block. Figure 7.4 depicts this scenario. This situation can be avoided if `SendWait` is only allowed to occur if the send stage is non-blocking: the rendezvous partner must already be blocking on the endpoint that the send is called on. This is not a problem when the `SendWait` is called on a reply cap: the presence of the reply cap guarantees a thread is blocked waiting for a reply. As a result, `SendWait` when called on a synchronous endpoint will succeed if the

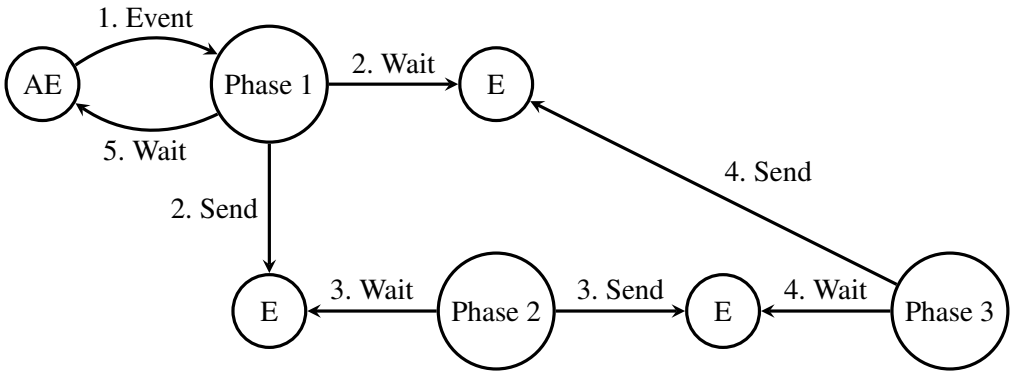


Figure 7.3: Data-flow scenario with endpoints: AE indicates an asynchronous endpoint through which an event occurs, E indicates a synchronous endpoint. Numbering indicates event ordering: a Send and Wait with the same number occur in one system call.

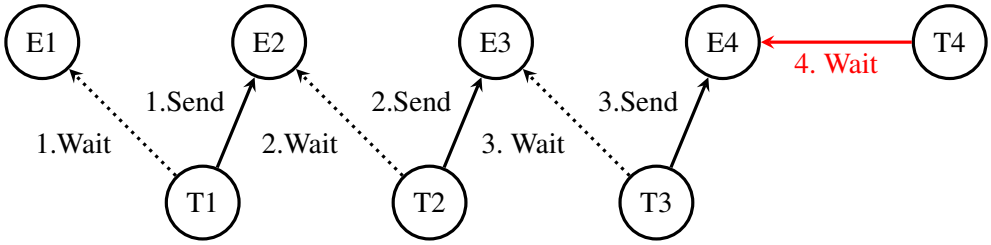


Figure 7.4: A `SendWait` triggering a long-running kernel operation. Threads T1, T2 and T3 call `SendWait` on endpoint pairs (E1, E2), (E2, E3) and (E3, E4) respectively, however each thread blocks on the send. Finally T4 waits on E4, triggering a chain of messages that the kernel would have to process if this pattern were permitted.

receiver is ready and waiting. Note that this long-running operation is not a problem with the existing `Call`, as the presence of the reply cap guarantees that earlier threads in the chain are blocked, preventing the long-running operation.

Server/Activity blocking

If a phase or server blocks waiting on input from another endpoint, messages can accumulate on the endpoint that donation occurs through. This could be input from a device, or input from another task with its own reservation. In this case, the request from the client with the highest priority should be serviced first to maintain real-time guarantees. This requires endpoint queues to be ordered, which increases the complexity of a non-fastpath IPC to O(number of threads)[1]. IPC ordering only needs to be applied to one side of the rendezvous process: either when the message is enqueued or when it is dequeued.

In some models, a server may wish to avoid blocking on input, but instead to issue an asynchronous request and continue to service other clients requests. In the verified seL4 kernel, a server would achieve this using the asynchronous endpoint binding mechanism or using a multi-threaded approach with one thread per client. The latter implementation is compatible with the scheduling context design, and will be explained in **??**. However the former approach is not compatible with passive servers running on clients scheduling

[1]By using a heap this could be reduced to $O(\log_2(\text{number of threads}))$, however we have conducted measurements previously establishing that a pointer-based list implemented in seL4 beats a pointer-based heap for up to and beyond 100 threads.

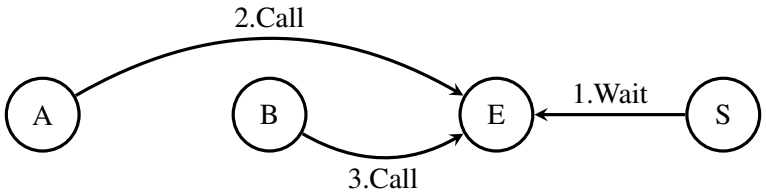


Figure 7.5: Client A makes a request from server S on endpoint E. A's scheduling context is donated from A to S. The budget expires while S is executing the request, blocking S from servicing other requests. Another Client, B, attempts to make a request but finds S blocked.

contexts. If scheduling context donation is not used, and the server has its own scheduling context, then the asynchronous endpoint binding approach remains viable.

Revoking a server

What happens if a server is revoked while executing on a client's scheduling context? Since the kernel does not track the 'home' of a scheduling context, it is not possible to return the scheduling context to the calling thread. Currently the scheduling context will be disconnected from the server, and not returned to the caller.

In real scenarios, this revocation only occurs if the sub-system is being torn down so this is not a problem. Otherwise, the scheduling context object is still valid: but it won't have any threads associated with it. A supervisor thread could be set up to reset the scheduling contexts of any threads that have lost theirs if a server goes down. Generally, if a server goes down in the middle of a request, action must be taken to restart the server and fix the client anyway so restoring the scheduling context must be incorporated in the protocol. Note that if the server goes down the reply cap is also lost: so in the current seL4 design restarting the server and replying to the client is already impossible.

7.4.4 Temporal Exceptions

Threads can register a synchronous endpoints for two different types of temporal faults: deadline faults and budget faults. In the former cause, a temporal fault is generated if the budget of a scheduling context expires while it is not home. In the latter case, the current job is not completed before the deadline passes. Both faults are optional and no fault message will be delivered if the respective endpoint is not set.

7.4.5 Helping

We introduce a new mechanism to the kernel—helping—which implements bandwidth inheritance as discussed in **??**. Helping only occurs if a scheduling context runs out of budget, no temporal fault handler is registered for the current thread, and another thread has attempted to contact the stuck server. Bandwidth inheritance is supported to allow shared servers and phases to remain single threaded.

We explain the helping mechanism with reference to Figure 7.5. A sends an IPC to S via endpoint E, and scheduling context donation occurs: S is now running on A_{sc}. As part of the donation process, a pointer to S_{tcb} is stored in E, as the help-target. A_{sc} expires, and B becomes runnable, issuing a request to S via E. Since no other threads are waiting to receive messages on E, B inspects the state of the help-target, and finds that S is running on expired

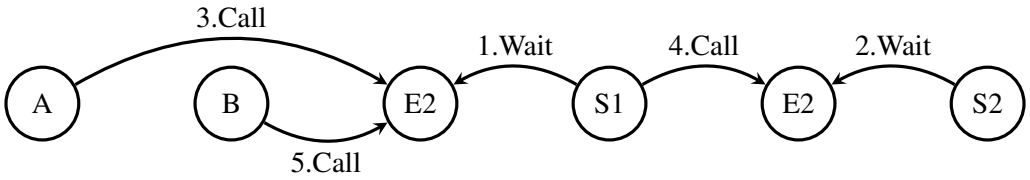


Figure 7.6: Clients A makes a request to server S1 on endpoint E1. S1 receives A's request first, and makes a nested request to another server, S2. A's scheduling context is donated from A to S1 to S2, but the budget expires while S2 is executing. Client B now makes a request, and finds S1 blocked.

scheduling context, A_{sc}. S inherits B_{sc} in order to finish its request. When S replies to A, A_{sc} is returned to A, and S picks up B's request from the IPC message queue, inheriting B_{sc}.

If B_{sc} also expires, then it will be returned to B and B is removed from the endpoint queue. When B_{sc} is recharged, it can restart the operation. This approach avoids large dependency chains in the system, and is acceptable as helping will never be triggered by HRT threads, and is intended for use by rate-based threads and SRT threads.

Nested helping

With the presence of nested servers, budget expiry can also occur and is more complicated. Note that in the data-flow scenario, nested budget expiry is not possible as once a phase is executed and the scheduling context passed on, that phase is ready to execute another request immediately. Nested budget expiry is illustrated in Figure 7.6.

This problem can also be solved through helping by following the chain of help-target TCBs and passing the scheduling context through the chain. This requires the addition of preemption points as it introduces a long-running operation. If the operation is preempted, the chain search will be abandoned until the thread is scheduled again, at which point the `Call` will be restarted (since anything could happen during the preemption—the scheduling context for the server may be replenished).

7.4.6 Server and activity initialisation

Except for systems using the reservation-per-thread model, where a scheduling context is assigned for each thread in the system, shared servers and activities are passive: they do not have their own scheduling context. However, without a scheduling context these threads are not runnable, so how can one initialise a passive thread?

Two options are available using the new mechanisms: dedicated initialisation reservations, or helping. Dedicated reservations involve resuming a thread with a reservation set purely for its initialisation phase. The thread transfers the reservation back to the initialiser with a `SendWait` call once it is ready to receive messages.

The helping approach operates by leveraging the helping mechanism and removes the requirement that a dedicated reservation be created purely for initialisation: instead, the initialiser passes its own reservation directly to the passive component. To achieve this, a new kernel invocation is provided, `seL4_Endpoint_SetHelpTarget`, which allows the help target of an endpoint to be set. To initialise a thread the initialiser can `Call` the server, thus donating its scheduling context to that thread via the helping mechanism. The server can then `ReplyWait` to the initialiser, and it is now blocked waiting for the first client.

7.4.7 Fastpath

seL4 has an optimised fastpath that is executed for `Call` and `ReplyWait`. As the semantics of these system calls have been altered, the impact on the fastpath has to be assessed. The conditions to hit the fastpath currently are:

1. the message arguments must fit into the scratch registers of the calling convention ($\leq$ 2 for x86, $\leq$ 4 for arm),
2. the message must be sent on a synchronous endpoint,
3. the receiver must be higher or equal priority,
4. the receiver must be blocking on the endpoint,
5. the message must be sent via a `Call` or `ReplyWait`.

All of these conditions are performance optimisations: if we `Call` to a lower priority thread the scheduler needs to be invoked to check there are no other runnable threads at a higher priority than the receiver.

The addition of scheduling contexts raises adds a new condition to the fastpath:

6. the scheduling context must not change.

When the current scheduling context changes, the time needs to be read to bill the previous scheduling context and an interrupt set for the budget expiry of the new scheduling context. Adding these operations to the fastpath would slow it down greatly, as access to the timer device is expensive. Consequently, the fastpath is aborted if the current scheduling context would change.

Scheduling context donation *does* occur on the fastpath, as it only adds to writes. Checking if the scheduling context will change adds 2 reads, and the donation takes 5 writes.

In future we will add a fastpath for `SendWait`.

7.4.8 Examples

This chapter so far has outlined the various mechanisms added to the kernel to support resource sharing. In this section we demonstrate how these mechanisms can be used to support different system policies for resource sharing.

Reservation-per-thread

Recall from **??** that in this model, clients and servers have their own reservations. The reservation-per-thread model is the most simple to implement: all components in a system are assigned their own scheduling context with sufficient parameters. If a shared resource runs out of budget, any clients will be blocked until the budget is recharged. While independent threads in this example are temporally isolated from each other, threads sharing resources servers using reservation-per-thread are not temporally isolated.

Migrating threads

TODO: refer to thread factory design pattern User-level systems can build multi-threaded servers with migrating threads in the following way: server threads run at priority p and wait on the server endpoint. An additional thread runs at priority $p-1$, whose purpose

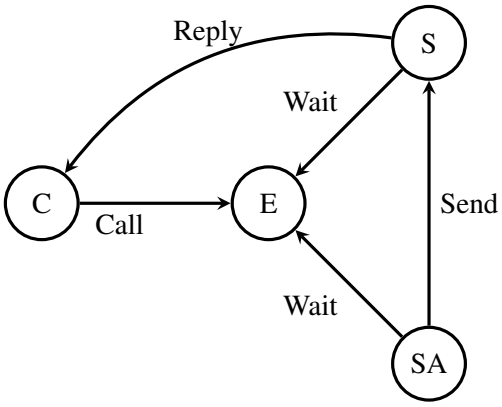


Figure 7.7: Migrating threads with a stack allocating thread, SA.

is to allocate stacks. If a client request comes in and no server thread is available to serve it, then the stack allocating thread will run, allocate a new stack and start a new server thread. The stack allocator will forward the client's scheduling context and request onto the new server thread, and wait on the endpoint again, using the new system call `SendWait`. This is illustrated in Figure 7.7—SA is the stack allocator, which creates server thread S after client C makes the first request.

Because the endpoint always has a thread waiting for messages (the stack allocator), the helping mechanism will not be triggered. This scenario relies on priority ordered IPC, and the correct ceiling priorities assigned to servers.

Exception

Temporal faults allow the user to implement custom handling for budget expiry. A temporal fault endpoint for budget expiry is set per thread, so a server can set up a dedicated thread to wait for temporal faults. The dedicated thread must be assigned its own reservation for handling faults. The thread can be used to complete the request (as an emergency reservation), to rollback the request and return an error from the server, or for other purposes. Of course, servers then must be thread safe such that the fault handling thread does not corrupt state.

Bandwidth inheritance

Bandwidth inheritance will occur if no temporal fault endpoint is registered and a blocked server encounters contention. This allows user systems to have temporally contained servers without the requirement that those servers be thread safe.

7.5 API

7.6 Summary

This section has outlined the details of integrating resource kernel concepts of enforcement, admission and scheduling into seL4. We also outlined our approach to resource sharing, using scheduling context donation over IPC, and showed how this supports servers using reservation-per-thread, migrating-threads, temporal exceptions or bandwidth inheritance.

The status of the implementation so far is as follows:

Invocation	*Description*
`bind`	Bind an object (TCB or Notification) to a SC.
`unbind`	Unbind all objects from a SC.
`unbindObject`	Unbind a specific object from a SC.
`consumed`	Return the amount of time since the last timeout fault, `consumed` or `yieldTo` was called.
`yieldTo`	Place the thread bound to this SC at the front of its priority queue and return any time consumed.

Table 7.4: Scheduling context capability invocations. Further detail is available in Appendix B.1.1.

8 Evaluation

Questions the evaluation needs to answer:

1. Show the implementations do not impose undue performance overheads on standard microkernel operations.
2. Show that criticality mode switch is bounded by high criticality threads.
3. Show the cost of various timeout exception recovery mechanisms.
4. Show that temporal isolation is achieved.
5. Show that deadlines can be met during and after a mode switch.
6. Show the model is practical and consistent with existing frameworks for critical software.

8.1 Hardware

We ran 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, although at the time of writing verification for X64 is ongoing. Currently, the only platforms that support more than one core are SABRE, IA32 and X64.

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

TODO: Appendix listing exact hardware details

8.2 Overheads

We first present a suite of microbenchmarks to evaluate any performance overheads against baseline seL4. For each benchmark we ensure floating point unit (FPU) context switching is off by performing the required number of system calls without activating it such that the kernel will cease switching the FPU context. 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, and subtract that from the final measurement.

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	*Arch.*	*Microarchictecture*	*CPU*	*Clock (GHz)*	*Mode*	*Cores*
KZM	ARMv6	ARM1136JF-S	i.MX31	1.00	32	1
SABRE	ARMv7	Cortex-A9	i.MX6	1.00	32	4
HIKEY32	ARMv8	Cortex-A53	Kirin 620	1.20	32	8
HIKEY64	ARMv8	Cortex-A53	Kirin 620	1.20	64	8
TX1	ARMv8	Cortex-A57	Jetson TX1	1.91	64	4
ATOM	x86	E3825	Atom	1.33	32	2
IA32	x86	i7-4770	Haswell	3.10	32	4
X64	x86	i7-4770	Haswell	3.10	64	4

Table 8.1: Hardware platforms and specifications used in benchmarks. HIKEY32/HIKEY64 and IA32/X64 are the same platform in 32 and 64 bit mode respectively.

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.

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
ATOM	TODO	TODO	TODO	TODO
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 parenthesis.

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.

SABRE and KZM 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 the timer registers are banked, making access fast for each core. 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 vast majority of new ARM platforms support the generic timers.

On X64 we use the timestamp counter (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.

While 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 practical on modern hardware.

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].

To evaluate IPC 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 not neither the caller or callee's IPC buffer accessed. This is done for both directions of IPC, as described in **??**.

We evaluate the IPC fastpath and two slowpath variants: slowpath between passive threads and active threads.

Fastpath

Fastpath results for `Call` and `ReplyRecv` increase by a few percent on each platform, resulting from extra checks on the fastpath to accommodate scheduling contexts, an extra capability lookup for the Resume object, touching two separate new objects (SCO and Resume object) and enforcing priorities on IPC delivery (baseline does FIFO).

We look at each platform in detail to determine the source of the overhead by using the performance monitor unit.

Slowpath

Slowpath results show the greater cost of the model, as shown in Table 8.4, which shows the cost of a slowpath IPC between two passive threads.

The benchmark hits the slowpath as the priorities are arranged such that the task starting the operation is a lower priority, which means not only do we invoke the slowpath but also the scheduler. However, since the server is passive, we do not change scheduling context, meaning the only overhead on `Call` is unnecessarily reading the time on kernel entry, as well as the mechanisms for scheduling context donation. This is quite small, very small on X64 (1%) as the kernel simply does a non-serialised read of the TSC. On ARM the overhead is greater (8%), as we use the ARM Cortex-A9 MPCore global timer (the only 64-bit timer available on the platform), which is memory mapped and shared between all of the cores. We evaluate the cost of reading the timer with a hot cache for ARM as 23 cycles.

[1]The IPC benchmarks already existed for seL4, but were modified to support the mixed criticality system (MCS) kernel during the work for this thesis

Platform	*Operation*	*Baseline*	*MCS*	*Overhead*	
KZM	`Call`	263 (0)	290 (0)	27	10%
	`ReplyRecv`	304 (0)	351 (2)	47	15%
SABRE	`Call`	289 (0)	308 (1)	19	6%
	`ReplyRecv`	313 (1)	348 (43)	35	11%
HIKEY32	`Call`	235 (0)	250 (0)	15	6%
	`ReplyRecv`	253 (4)	279 (11)	26	10%
HIKEY64	`Call`	250 (2)	280 (4)	30	12%
	`ReplyRecv`	265 (3)	304 (4)	39	14%
TX1	`Call`	387 (9)	479 (93)	92	23%
	`ReplyRecv`	396 (7)	439 (14)	43	10%
X64	`Call`	409 (7)	417 (2)	8	1%
	`ReplyRecv`	389 (1)	416 (1)	27	6%
IA32	`Call`	392 (4)	382 (2)	-10	-2%
	`ReplyRecv`	369 (1)	405 (2)	36	9%

Table 8.3: Time in cycles for seL4 fastpath IPC.

`ReplyRecv` shows a higher overhead on both platforms, due to the extra slowpath capability lookup of the resume capability, the ordered IPC checks, and accessing the SCO and resume object.

Active slowpath

Finally we show the cost of slowpath IPC between two active threads, where both caller and callee have a scheduling context. In addition to the overheads of Section 8.2.2, we must bill and change the scheduling context and reprogram the timer. On X64, reprogramming the timer uses TSC-deadline mode, which is programmed via the model specific registers and is also very fast. On ARM, we evaluate the cost of reprogramming the timer with a hot cache as 36 cycles.

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 blocks on, waiting for any fault messages (**??**).

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. The fault handler is passive, so no scheduling context switch occurs.

We measure both directions of the fault, as well as the round trip cost of from the fault handlers side. Similar to standard slowpath-IPC, the largest impact is on the reply path, where ordered IPC and the extra lookup of the resume object add overhead.

Platform	*Operation*	*Base*	*MCS*	*Overhead*	
KZM	`Call`	1471 (19)	2193 (29)	722	49%
	`ReplyRecv`	1433 (10)	3020 (25)	1587	110%
SABRE	`Call`	911 (67)	1222 (55)	311	34%
	`ReplyRecv`	885 (12)	1335 (34)	450	50%
HIKEY32	`Call`	974 (8)	1340 (13)	366	37%
	`ReplyRecv`	924 (9)	1721 (19)	797	86%
HIKEY64	`Call`	805 (3)	1031 (16)	226	28%
	`ReplyRecv`	778 (5)	1205 (15)	427	54%
TX1	`Call`	756 (27)	945 (9)	189	25%
	`ReplyRecv`	747 (15)	1026 (25)	279	37%
X64	`Call`	668 (8)	718 (8)	50	7%
	`ReplyRecv`	652 (8)	822 (13)	170	26%
IA32	`Call`	670 (2)	735 (2)	65	9%
	`ReplyRecv`	702 (31)	885 (32)	183	26%

Table 8.4: Time in cycles seL4 slowpath IPC between passive threads.

8.2.4 Signalling and interrupts

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. 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.

The `signal()` operation signals a Notification object (semaphore). This microbenchmark evaluates the cost of signalling a lower priority thread, a common operation for interrupt service routines. We have added an experimental fastpath to both the base and MCS kernels, which shows that when the scheduler is not used and a thread switch does not occur, there is no overhead at all.

8.2.5 Scheduling

The `schedule` benchmark measures the cost of a signal to a higher priority thread, which forces a reschedule. Scheduling cost increases noticeably due to the need for first reading and then reprogramming the timer for budget enforcement. Furthermore, the sporadic replenishment logic is far more complicated than the previous tick-based logic, and there is some extra code for dealing with scheduling contexts. 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. In fact, the $O(1)$ scheduler is a recent addition to seL4: scheduling used to be far more expensive. ***TODO: mention this in the seL4 chapter***.

8.2.6 Multicore

We run two multicore benchmarks, the first evaluating multicore throughput of the MCS kernel vs. the baseline kernel, the second based on our shared server advanced encryption standard (AES) case study to demonstrate the multicore model.

Throughput

We run an 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 of via `Call` and `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 less calls are made.

We modify the benchmark such that each server thread is passive on the MCS kernel. Results are displayed in Figure 8.1 and show a minor impact on IPC throughput for high 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. This is expected as the MCS model only introduces extra per-core state, with no extra shared state between cores.

Platform	*Operation*	*Base*	*MCS*	*Overhead*	
KZM	`Call`	1471 (19)	2510 (35)	1039	70%
	`ReplyRecv`	1433 (10)	3152 (29)	1719	119%
SABRE	`Call`	911 (67)	1638 (356)	727	79%
	`ReplyRecv`	885 (12)	1448 (97)	563	63%
HIKEY32	`Call`	974 (8)	1359 (21)	385	39%
	`ReplyRecv`	924 (9)	1543 (8)	619	66%
HIKEY64	`Call`	805 (3)	1023 (10)	218	27%
	`ReplyRecv`	778 (5)	1097 (6)	319	41%
TX1	`Call`	756 (27)	914 (3)	158	20%
	`ReplyRecv`	747 (15)	953 (11)	206	27%
X64	`Call`	668 (8)	905 (5)	237	35%
	`ReplyRecv`	652 (8)	967 (7)	315	48%
IA32	`Call`	670 (2)	953 (4)	283	42%
	`ReplyRecv`	702 (31)	1022 (2)	320	45%

Table 8.5: Time in cycles for seL4 slowpath IPC between active threads.

Platform	*Operation*	*Baseline*	*MCS*	*Overhead*	
SABRE	`fault`	758 (67)	1044 (97)	286	37%
	`reply`	839 (10)	1299 (24)	460	54%
	`round trip`	1588 (121)	2353 (87)	765	48%
HIKEY32	`fault`	811 (42)	1035 (58)	224	27%
	`reply`	806 (6)	1642 (23)	836	103%
	`round trip`	1566 (35)	2712 (66)	1146	73%
HIKEY64	`fault`	605 (16)	790 (34)	185	30%
	`reply`	790 (14)	1203 (22)	413	52%
	`round trip`	1323 (12)	1988 (53)	665	50%
TX1	`fault`	664 (23)	667 (22)	3	0%
	`reply`	692 (6)	948 (3)	256	36%
	`round trip`	1348 (12)	1610 (11)	262	19%
X64	`fault`	593 (10)	647 (4)	54	9%
	`reply`	636 (5)	859 (11)	223	35%
	`round trip`	1226 (11)	1502 (10)	276	22%
IA32	`fault`	625 (15)	697 (9)	72	11%
	`reply`	611 (8)	822 (11)	211	34%
	`round trip`	1219 (6)	1505 (17)	286	23%

Table 8.6: Time in cycles of seL4 fault IPC between passive threads.

8.2.7 Full system benchmark

To demonstrate the impact of the overheads measured in this chapter in a real system, 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 [Foundation, 2018] all on the X64 machine.

For seL4, we use a single-core Rump library OS [Kantee and Cormack, 2014] to provide NetBSD [Foundation, 2018] network drivers at user level. The system architecture is shown in Fig. 8.2. 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. 8.2. 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.

We compare against bare-metal Rump,and NetBSD (version 7.0.2) itself, which both use the same drivers. Additionally we compare to Linux in single-user mode. Figure 8.2 shows the architecture of the benchmark for each system.

Table 8.9 shows the achieved throughput of Redis+Rump running bare-metal, and Redis on the seL4 baseline and as well as the MCS branch, plus Linux and NetBSD for comparison.

Platform	*Operation*	*Base*	*MCS*	*Overhead*	
KZM	`IRQ latency`	484 (15)	1287 (24)	803	165%
	`signal`	148 (0)	149 (0)	1	0%
SABRE	`IRQ latency`	1617 (98)	1737 (104)	120	7%
	`signal`	134 (10)	139 (0)	5	3%
HIKEY32	`IRQ latency`	529 (5)	778 (8)	249	47%
	`signal`	118 (0)	118 (0)	0	0%
HIKEY64	`IRQ latency`	505 (19)	633 (13)	128	25%
	`signal`	197 (14)	197 (14)	0	0%
TX1	`IRQ latency`	767 (11)	815 (9)	48	6%
	`signal`	266 (17)	270 (18)	4	1%
X64	`IRQ latency`	1106 (55)	1265 (55)	159	14%
	`signal`	130 (2)	132 (2)	2	1%
IA32	`IRQ latency`	1043 (55)	1146 (56)	103	9%
	`signal`	177 (1)	174 (2)	-3	-1%

Table 8.7: Time in cycles of seL4 signal and IRQ latency on seL4 baseline vs. MCS kernels. Standard deviations shown in brackets.

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 achieve slightly better performance than NetBSD. This indicates that the MCS model comes with low overhead.

8.2.8 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 extend of the provided functionality.

Platform	*Operation*	*Base*	*MCS*	*Overhead*	
KZM	`schedule`	1297 (87)	2436 (228)	1139	87%
	`yield`	580 (0)	1779 (0)	1199	206%
SABRE	`schedule`	879 (91)	1056 (103)	177	20%
	`yield`	565 (288)	820 (256)	255	45%
HIKEY32	`schedule`	823 (68)	1064 (83)	241	29%
	`yield`	421 (54)	830 (52)	409	97%
HIKEY64	`schedule`	742 (54)	968 (94)	226	30%
	`yield`	417 (40)	577 (50)	160	38%
TX1	`schedule`	715 (34)	855 (79)	140	19%
	`yield`	478 (40)	530 (33)	52	10%
X64	`schedule`	365 (33)	713 (62)	348	95%
	`yield`	177 (21)	445 (25)	268	151%
IA32	`schedule`	424 (42)	751 (69)	327	77%
	`yield`	261 (2)	532 (2)	271	103%

Table 8.8: Time in cycles of seL4 IPC scheduling costs.

System (IRQ)	*Throughput* (k ops/s)	*Utilisation* (%)	*Latency* (ms)
seL4 (IOAPIC)	138.7 (0.38)	100 (0.0)	1.4 (.01)
seL4-MCS (IOAPIC)	138.5 (0.34)	100 (0.0)	1.4 (.01)
seL4-MCS (MSI)	127.3 (0.61)	100 (0.2)	1.6 (.01)
NetBSD (MSI)	134.0 (0.21)	99 (0.1)	1.5 (.01)
Linux (MSI)	179.4 (0.36)	95 (0.6)	1.1 (.01)
Linux (IOAPIC)	111.9 (0.36)	100 (0.0)	1.8 (.01)
BMK (PIC)	144.1 (0.23)	100 (0.0)	1.4 (.01)

Table 8.9: 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 brackets.

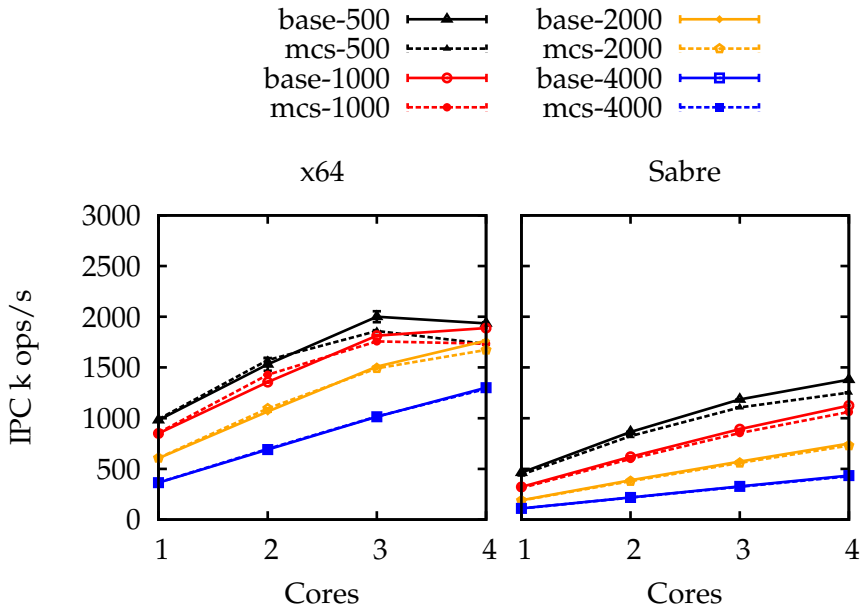


Figure 8.1: Results of the multicore IPC throughput benchmark, baseline seL4 vs MCS. Each series is named *name-N*, where *name* is *base* and *mcs* for the baseline and MCS kernel respectively, and *N* is the upper bound on the number of cycles between each IPC for that series.

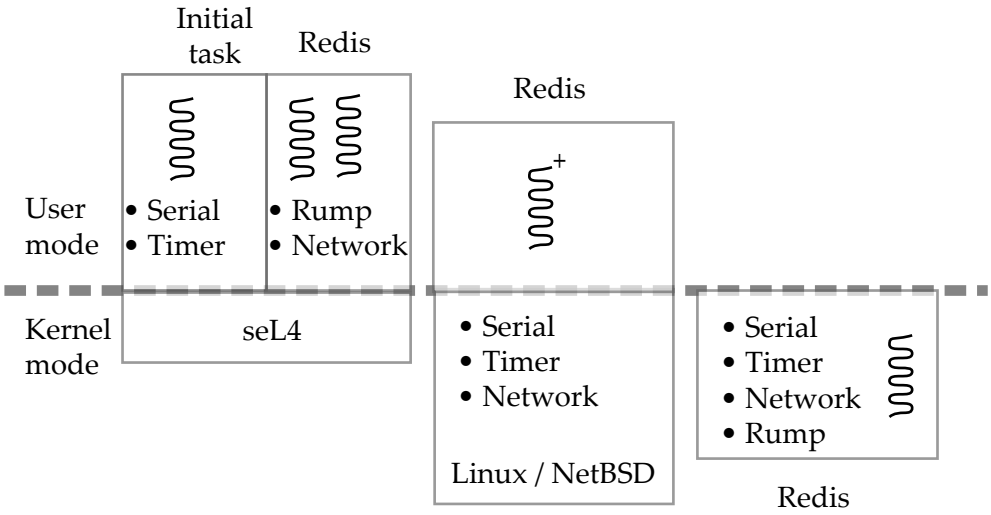


Figure 8.2: System architecture of the Redis / YCSB benchmark on seL4, Linux, NetBSD and Rump unikernel.

8.3 Temporal Isolation

We now evaluate the temporal isolation properties achieved by the model described in TODO. We use two system benchmarks to show that processes attain and do not exceed their CPU allocation provided by their scheduling context. We then evaluate and demonstrate different techniques to restore server state after a timeout exception and finally show temporal isolation between clients in a shared server scenario.

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 setup described in Section 8.2.7 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 hog to control the amount of time left over for the server configuration. Figure 8.3 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 hog thread). Each data point is the average of three benchmark runs.

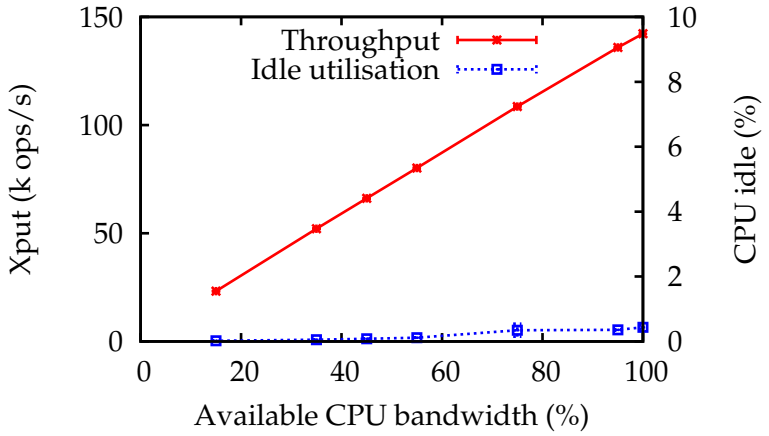


Figure 8.3: Throughput of Redis YCSB workload A and idle time vs available bandwidth.

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. 8.4. 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.

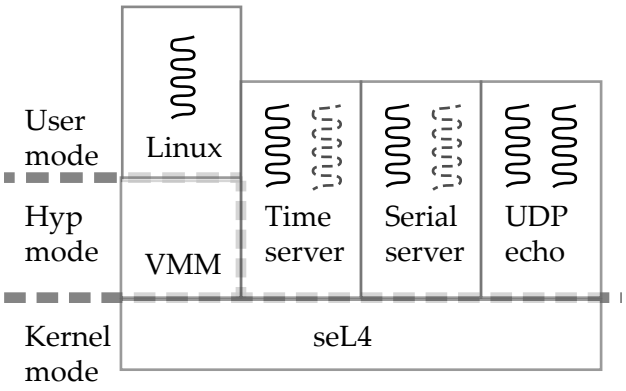


Figure 8.4: System architecture of ipbench benchmark.

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 8.5 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 blowout. This result shows that our sporadic server implementation is effective in bounding interference of a high-priority process.

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.

Figure 8.6 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 request on the clients scheduling context and an active thread 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

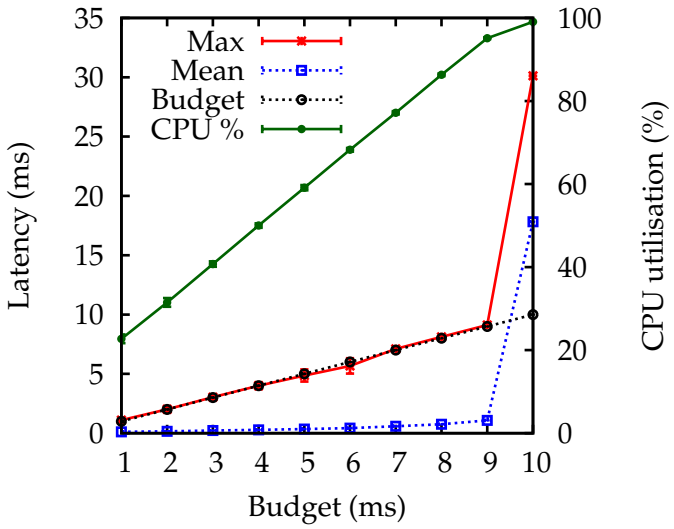


Figure 8.5: Average and maximum latency of UDP packets with a CPU hog running a high priority with a 10 ms budget.

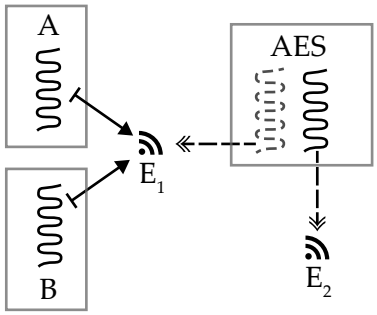


Figure 8.6: Architecture of the AES case study. Client *A* and *B* make IPC requests over endpoint E_1 of passive *AES* which has an active timeout fault handling thread waiting for fault IPC messages on endpoint E_2.

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, and have a budget insufficient to complete the request. When the server is running on the clients 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 client 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 TODO. Additionally, although out 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 servers 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 involves is as follows:

1. Fault is received by timeout handler.
2. Handler replies to the client by sending a message to the client reply object, with an indication of how much progress has been made. In the case of AES this is how much data is left to encrypt. By invoking the reply object, the client's scheduling context is returned from the server to the client.
3. Server is restored to a known state by the timeout handler (restoring registers, stack frames and any global state.
4. Handler binds a scheduling context to the server for it to execute on.
5. Handler blocks waiting for a message from the server.
6. Now the server runs from its checkpoint. In the case study, the server is checkpoint just before it initiates the passive initialisation protocol (TODO ref to passive init figure), so this system call is repeated. The server signals the timeout handler and blocks on E_1, ready for client requests.
7. Handler wakes and converts the server back to passive.
8. Handler 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 servers 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.

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 progress.

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, and 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. Fault is received by timeout handler.
2. Unbind client scheduling context from server. ***TODO: clarify who does what***
3. Bind server with emergency budget.
4. Reply to the timeout fault, which restarts the server.
5. Enqueue the timeout handler itself, with an empty sized request, in the servers endpoint queue, by invoking E_1
6. Server finishes request with extra budget and replies to the client.
7. Timeout handler is highest priority, so it overtakes other clients and wakes up after the server processes its empty request. The handler then converts the server back to passive.
8. Handler blocks on E_2, ready for further timeout faults.

This case requires the timeout handler to have access to client scheduling contexts in order to unbind them from the server.

Extend

The final technique is to simply increase the clients budget on each timeout, which requires the timeout fault handler to have access to the clients scheduling contexts. This could be deployed in SRT systems or for specific threads with unknown budgets up to a limit.

1. Fault is received by timeout handler.
2. Extend client budget by configuring scheduling context.
3. Reply to fault message, which resumes the server.

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 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 8.10 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,

Platform	*Operation*	*Cache*	*Min*	*Max*	*Mean*	σ
SABRE	Rollback	hot	10.7	13.0	11.8	0.5
		cold	19.4	24.7	23.3	1.2
	Emergency	hot	4.2	5.6	4.7	0.3
		cold	13.0	14.4	13.6	0.3
	Extend	hot	0.8	3.1	1.9	0.4
		cold	6.8	7.8	7.1	0.3
	Kill	hot	9.9	12.2	11.0	0.5
		cold	21.4	22.7	22.1	0.3
X64	Rollback	hot	1.47	2.89	1.96	0.39
		cold	4.05	4.96	4.57	0.16
	Emergency	hot	0.97	1.77	1.25	0.18
		cold	2.27	3.04	2.43	0.09
	Extend	hot	0.16	0.86	0.28	0.13
		cold	0.96	1.46	1.13	0.08
	Kill	hot	1.37	1.49	1.40	0.01
		cold	4.03	4.75	4.36	0.16

Table 8.10: Cost of timeout handler operations in μs, as measured by timeout exception handler. σ is the standard deviation.

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 **??**).

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 8.7 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.

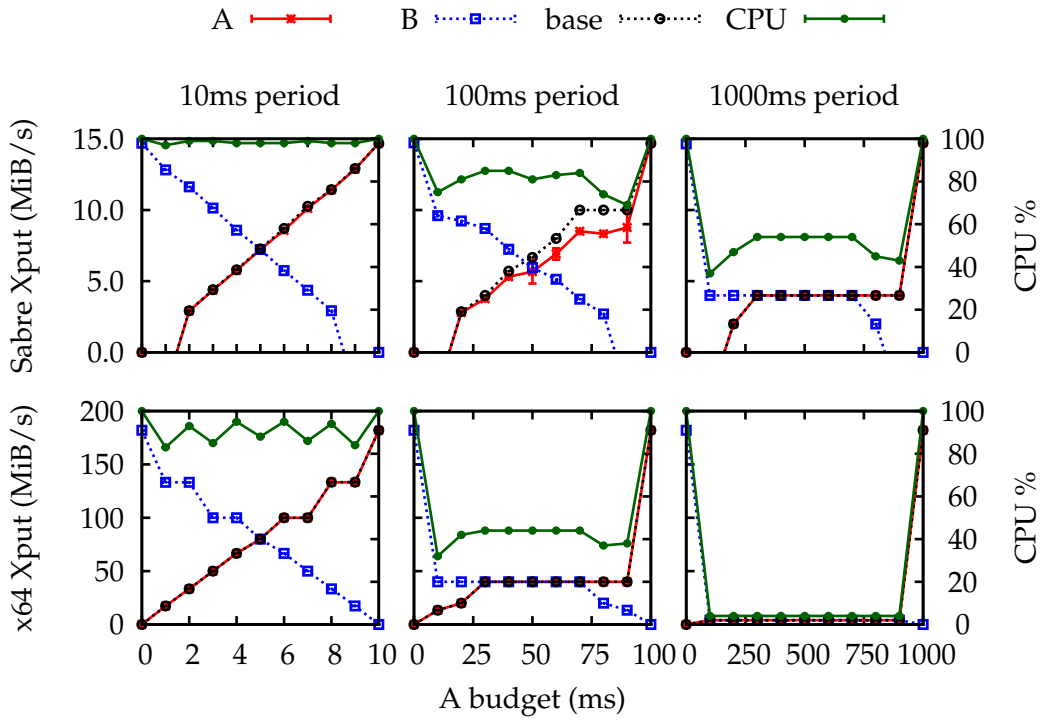


Figure 8.7: Throughput for clients A and B of a passive AES server processing 10 requests of 4 MiB of data with limited budgets on the X64 (top row) and SABRE (bottom row) platforms. The two clients' budgets add up to the period, which is varied between graphs (10, 100, 1000 ms). Clients sleep when they 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.

8.3.6 Summary

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 demonstrated several alternatives for handling such faults, demonstrating the feasibility of the model.

8.4 Asymmetric Protection

We evaluate our kernel mechanism for changing criticality with two benchmarks: A microbenchmark measuring the cost of a mode switch, and another showing deadline misses of threads sets over several mode changes.

To evaluate the cost of changing the system criticality level, we configure the kernel with 128 priorities and 2 criticality levels (0–1). We then run 3 experiments as follows:

UL: measures the time to change the priority of n threads from user-level;

HI: measures the time for raising the system criticality level from 0 to 1 with one LOW thread and n HIGH threads, i.e. the priority of n threads must be boosted;

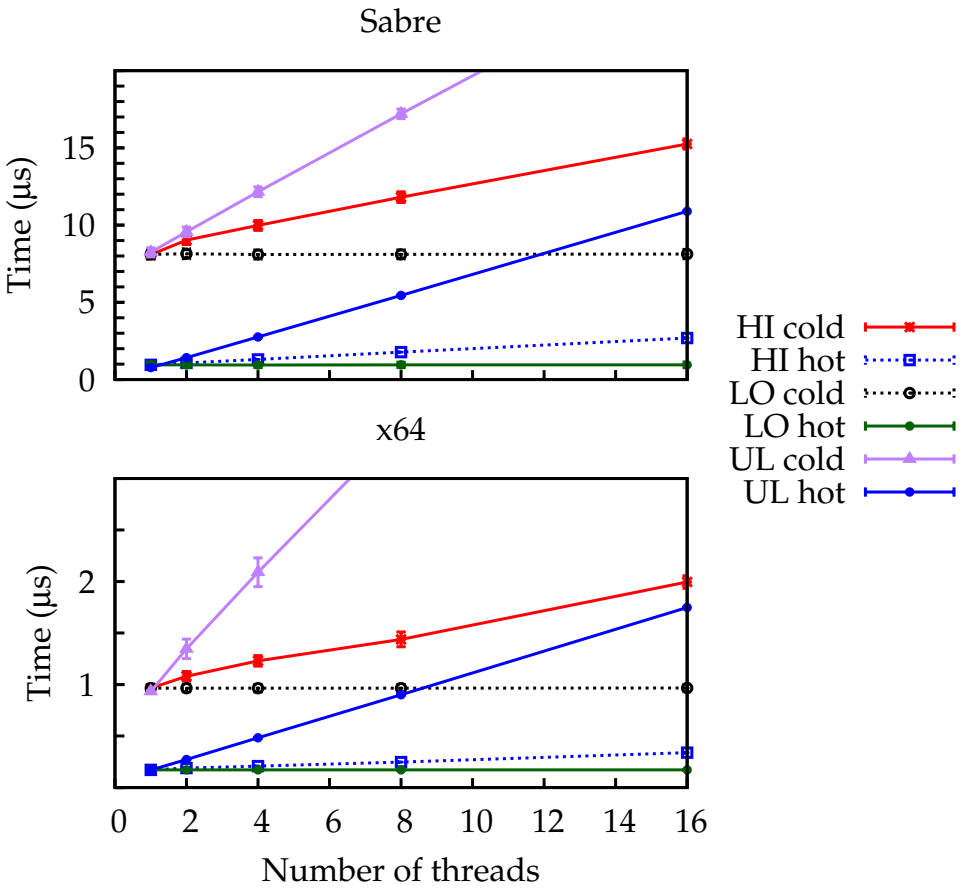


Figure 8.8: Cost of switching the priority of n threads, as well as changing from LOW to HIGH criticality level while increasing the number of HIGH and LOW threads, on SABRE and X64. Each data point is the average of 100 runs, with very small standard deviations.

LO: measures the time for raising the system criticality level from 0 to 1 with one HIGH thread and n LOW threads, i.e. only one thread must be boosted.

Figure 8.8 shows the results, where each data point is the result of 100 measurements. We show results with a primed cache (hot) and flushed cache (cold). As the graph shows, switching is linear in the number of HIGH threads being boosted. Varying the number of LOW threads has no impact on the execution time of a mode switch. This is important, as the schedulability analysis for critical threads must not depend on less-critical threads, and should not make assumptions how many threads less-critical software creates.

In absolute terms, the results show that a mode switch is fairly fast, remaining under 2 μs on X64 and about 12 μs on SABRE for switching 8 threads with a cold cache. Most systems will not have more than a few high criticality threads, and deadlines for critical control loops in cyber-physical systems tend to be in the tens of milliseconds, meaning that budgeting a few microseconds for an emergency mode switch is eminently feasible.

Additionally, the results show that changing the priority at user-level, rather than using the criticality boost, is also linear in the number of threads to be boosted, but is significantly slower than using the kernel mechanism (i.e. mode switch). The higher cost from user-level

operation results from multiple switches between kernel and user mode, and the repeated thread-capability look-ups.

For our second mode-switch benchmark, we ported 3 processor intensive benchmarks from the MiBench [Guthaus et al., 2001] to act as workloads. Each benchmark runs in its own Rump process with an in-memory file system, and shares a timer and serial server.

We altered the benchmarks to run periodically in multiple stages. To obtain execution times long enough, some of the 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.

Table 8.11 lists the benchmarks with periods and criticalities. *susan*, the most critical, has three stages: edge detection, smoothing, and corners. The next critical task, *jpeg*, has two stages: encode, and decode. The least critical task, *mad* has only one stage. We run the benchmark for 20 s for each of the stages (repeating the last phase where threads have no new phase), and increment the system criticality level at stage transition. The parameters are arranged such that rate-monotonic priorities are inverse to the criticalities.

Results are shown in Table 8.11. 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 that our mechanisms operate as intended.

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 8.11: Results of mode 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 recorded 52 (0.1), 86 (15.2) and 100 (0.0)% CPU utilisation for each stage respectively. Standard deviations are shown in parenthesis.

8.5 Practicality

TODO: intro

8.5.1 User-level scheduling

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 show that we can efficiently implement EDF, the popular and optimal dynamic-priority policy (see **??**).

We implement the EDF scheduler as an active server with active clients which run at an seL4 priority below the scheduler. The scheduler waits on an endpoint on which it receives messages from its clients or the timer.

Each client has a period which represents its relative deadline and a full reservation (i.e. equal to the period). Clients either explicitly 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 embedded in Linux, where we use Feather-Trace [Brandenburg and Anderson, 2007] to gather data. We use the C-EDF scheduler, which is a partitioned (per-core), clustered (per-node) EDF scheduler, on a single core. We 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 8.9 shows that our user-level EDF scheduler implementation is competitive with t EDF 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.

8.5.2 multicore

TODO: diagrams

We use the AES shared server from Section 8.3.2 to demonstrate how our MCS mechanisms can be deployed in an multicore scenario with shared servers with each client requesting 1MB of data to be encrypted. We set up the following case studies:

TODO: update to multi and single

1-passive 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.

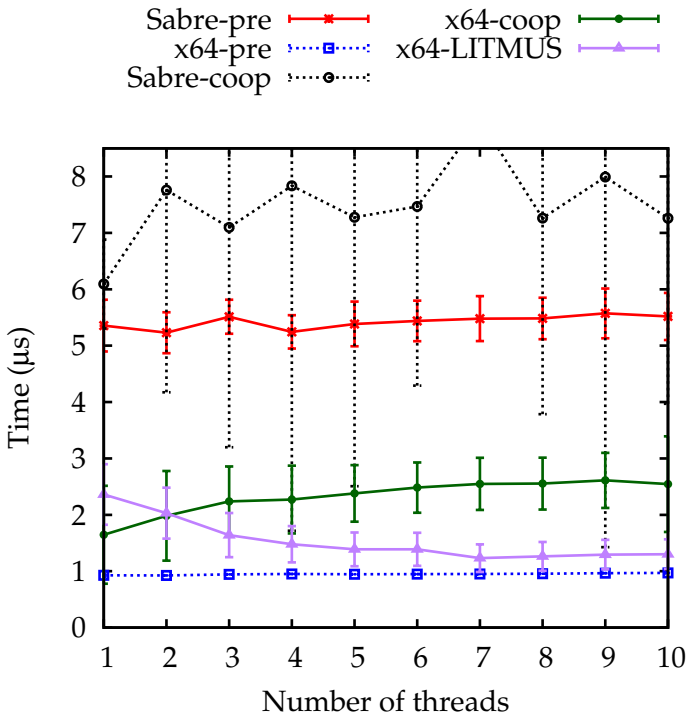


Figure 8.9: Execution time of seL4 user-mode EDF scheduler compared to kernel scheduler in x64 LITMUS^{RT}.

N-passive the AES server has one passive thread per core, and an endpoint is set up for each core, demonstrating a parallel server.

Results are shown in Fig. 8.10. As expected, the serialised server (1-passive) achieves the same throughput regardless of the number of clients. N-passive scales nearly perfectly, as the benchmark does not have any bottlenecks: the server does minimal system calls and the workload does not involve synchronisation.

8.6 Summary

Our evaluation demonstrates our model has minimal overheads, achieves isolation, provides a notion of criticality orthogonal to priority, and allows for efficient user-level scheduling. In total, we add 2,245 lines of SLOC [Wheeler, 2001] to the preprocessed kernel code for the SABRE (the verified platform), representing a 16% increase.

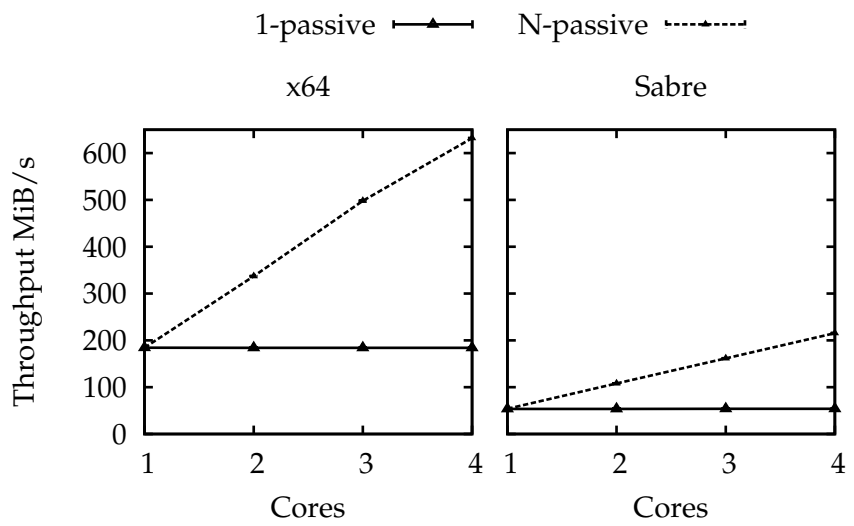


Figure 8.10: Results of the AES shared server multicore case study. 1-passive shows results for a single passive server thread migrating between cores to service clients while N-passive has one passive thread per core. For both series, the number of clients is equal to the number of cores.

9 Conclusion

9.1 Contributions

Specifically, we make the following contributions:

- A capability system for time that has low overhead and does not limit the system to a particular scheduling policy, including implementation of arbitrary scheduling policies at user level;
- a notion of CPU reservations that is compatible with fast IPC implementations traditionally used in high-performance microkernels, and is compatible with established real-time resource-sharing policies;
- the first OS kernel supporting an explicit notion of criticality, orthogonal to priority, in addition to the above;
- an exploration of implementation in the non-preemptible seL4 microkernel and its interaction with the kernel's model of user-level management of kernel memory, which is a critical enabler of strong spatial isolation.

9.2 Related work

AUTOBEST [Zuepke et al., 2015] is a separation kernel where the authors demonstrate implementations of AUTOSAR and ARINC653 in separate partitions.

MINIX 3 is a traditional microkernel with a focus on reliability rather than performance. Herder et al. [2006] implemented temporal isolation in MINIX 3 by allowing threads to be selectively switched to EDF scheduling, and providing hooks that allowed for a CBS implementation at user level.

Quest-V [Danish et al., 2011] 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. In Quest-V, separate partitions are assigned to different priority levels, and communication via shared memory and inter-processor interrupts is permitted between partitions.

Fiasco.OC introduced scheduling contexts in order to allow paravirtualised guests to switch between scheduling context with different parameters in order to provide flattened hierarchical scheduling [Lackorzyński et al., 2012], which prevents scheduling integrity violations when scheduling multiple real-time guests by exporting scheduling information to the hypervisor. Scheduling contexts in Fiasco.OC contained a budget, a replenishment

rule, and a priority, and guests can change scheduling contexts on priority switches and interrupt service routines.

An implementation of RBED has also been completed on OKL4 [Petters et al., 2009].

9.3 Future work

TODO: Frequency scaling TODO: multicore locking and scheduling

This chapter outlines what this PhD project has accomplished so far and what I plan to achieve in the future.

9.4 Progress

This research will investigate the scheduling and resource sharing problems of running application software at different criticality levels with different scheduling requirements in the context of the verified microkernel, seL4. This research will attempt to break new ground by developing a real-time operating system kernel with full system schedulability analysis, including that of the kernel code and scheduler.

The goals of this project are to provide (reproduced from the introduction):

G1 A principled approach to processor management, treating time as a fundamental kernel resource, while allowing it to be overbooked, a key requirement of mixed-criticality systems;

G2 safe resource sharing between applications of different criticalities and different temporal requirements.

This research is motivated by the rise in demand for mixed-criticality systems. It is clear from Chapter 2 that although much work has been conducted on scheduling algorithms for mixed-criticality systems, little work has considered the implications for operating systems. In addition, mixed-criticality systems require more strict access control to time than has typically been required from standard real-time systems. This motivates **G1**, which investigates how trusted applications can control the use of time as a resource, such that all timing requirements are met and low criticality tasks cannot compromise the timing requirements of higher criticality tasks. **G2** covers how time flows through shared resources, which may be shared between tasks of different criticalities and different temporal strictness (HRT, SRT, best effort) without violating the real-time requirements.

This research is split into several clear phases: background research (year 1), design and implementation (year 2) and evaluation (year 3). We are currently at the middle of the evaluation phase, having completed the following: :

- developed primitives in seL4 to allow a diverse range of scheduling options at user-level;
- extended the seL4 capability model to apply to time, such that it can be enforced by the kernel;
- evolved the current seL4 IPC mechanisms to allow for analysable blocking of real-time tasks, as required for mixed-criticality resource sharing.
- extended the model to cover and run on SMP systems.

- built a set of microbenchmarks for evaluating seL4 master against the real-time version.
- built several examples including an AES encryption server, and built a variety of timeout exception handling

Throughout the evaluation, I've discovered problems with the design that have faciliated entire redesigns, tweaks and revaluations.

The next stage is run the benchmark suite on the newest kernel and complete the case studies.

9.4.1 Ongoing Evaluation

Our design and will be evaluated according to the following qualities over the next 12 months:

Policy-Mechanism Separation

The aim of the project is to be as true to the microkernel minimality principle as possible—this means that kernel features should avoid influencing user policy decisions as much as possible. The motivation for using the microkernel minimality policy is to provide widely-applicable, trustworthy scheduling. Our case studies should allow us to evaluate how successful our design is in keeping policy and mechanism separate.

Schedulability analysis

Designs must be conscious of minimising actual and estimated WCET as in order to provide a kernel that allows for schedulability analysis, a full profile of WCET must be available. At the end of the project it should be possible to take a full description of a closed system and irrefutably show that the system is always schedulable. Similarly, for a mixed-criticality system, it must be possible to show that the critical parts of the system are always schedulable.

Measures of verification potential

I have determined that the code follows the verification subset of C and passes through the verification C-Parser. I am actively working with the verification team this year to verify the first set of patches.

9.4.2 Case studies

Case studies will demonstrate that the scheduling and resource sharing mechanisms are practical. So far I have built user level schedulers, an AES server for measuring rollback costs. I need to finish a network case study and the SMACCM case study.

SMACCM

Another ideal case study for mixed-criticality systems is an autonomous helicopter. Our research group is currently working on building a high-assurance system for such a helicopter (www.ssrg.nicta.com.au/projects/TS/SMACCM), which I should be able to test running my scheduler. I have already been involved with the SMACCM project, advising on real-time scheduling, resource sharing and analysis.

9.5 Conclusion

Mixed-criticality real-time systems that provide isolation and hard guarantees at the operating system level are desirable, due to the unscalable nature of hardware isolation. Much research has been conducted into how to schedule workloads for real-time, mixed-criticality systems, however the implications for operating systems are yet to be examined, with most implementations falling short of what is required in a highly critical environment.

As a result, this ongoing research investigates the implications of mixed-criticality, real-time scheduling in the context of a high-assurance operating system, driven by the rise of mixed-criticality systems in safety-critical environments. The research investigates the trade-offs between verification potential, performance and policy-mechanism separation with respect to mixed-criticality scheduling and resource sharing. In addition to a real-time iteration of seL4 that offers full-system schedulability analysis including the kernel and scheduler, the contributions are expected to provide a model for future L4 real-time scheduling primitives and a novel approach to applying fine-grained permissions to time without sacrificing performance.

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.

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, 2011.

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.

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.

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.

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.

Z Deng and Jane W. S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium*, December 1997.

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/.

FiascoOC. Fiasco.OC, 2013. URL http://tudos.org/fiasco.

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.

The NetBSD Foundation. NetBSD, 2018. URL https://www.netbsd.org.

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.

T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995.

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.

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.

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.

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*, EuroSys Conference, page 14, Porto, Portugal, April 2018. ACM Sigops.

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.

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, Martin Lawitzky, Ryan Heffernan, and Kevin Elphinstone. Towards real multi-criticality scheduling. In *IEEE Conference on Embedded and Real-Time Computing and Applications*, pages 155–164, Beijing, China, August 2009.

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.

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.

Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, March 1996.

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.

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, Felipe Cerqueira, Björn B. Brandenburg, Anna Lyons, and Gernot Heiser. FlaRe: Efficient capability semantics for timely processor access. Manuscript available from https://mpi-sws.org/~bbb/papers/pdf/preprint-FlaRe.pdf, October 2013.

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 Fastpath Performance

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	263	290	263	293	263	290	0	0
Cache L1I miss	3	3	5	5	3	4	0	0
Cache L1D miss	0	0	0	2	0	1	0	1
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	132	149	132	149	132	149	0	0
Branch misspredict	5	5	5	5	5	5	0	0
memory access	0	0	0	2	0	0	0	1

Table A.1: kzm Call fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	304	350	304	356	304	351	0	2
Cache L1I miss	3	3	5	5	3	4	0	0
Cache L1D miss	0	0	0	2	0	0	0	1
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	154	191	154	191	154	191	0	0
Branch misspredict	6	6	6	6	6	6	0	0
memory access	0	1	0	-1	0	2684354559	0	2147483647

Table A.2: kzm Reply recv fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	289	307	289	311	289	308	0	1
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	4	4	4	4	4	4	0	0
TLB L1D miss	3	4	3	4	3	4	0	0
Instruction exec.	143	160	143	160	143	160	0	0
Branch misspredict	0	0	0	0	0	0	0	0
memory access	0	0	0	0	0	0	0	0

Table A.3: sabre Call fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	312	331	314	459	313	348	1	43
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	4	5	4	5	4	5	0	0
TLB L1D miss	4	3	4	3	4	3	0	0
Instruction exec.	165	201	165	201	165	201	0	0
Branch misspredict	0	0	0	0	0	0	0	0
memory access	0	0	0	0	0	0	0	0

Table A.4: sabre Reply recv fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	235	250	235	250	235	250	0	0
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	144	161	144	161	144	161	0	0
Branch misspredict	4	4	4	5	4	4	0	0
memory access	50	60	50	60	50	60	0	0

Table A.5: hikey32 Call fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	251	275	263	307	253	279	4	11
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	166	202	166	202	166	202	0	0
Branch misspredict	4	4	5	4	4	4	0	0
memory access	59	67	59	67	59	67	0	0

Table A.6: hikey32 Reply recv fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	250	275	259	284	250	280	2	4
Cache L1I miss	0	0	0	2	0	1	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	183	210	183	210	183	210	0	0
Branch misspredict	2	2	3	3	2	2	0	0
memory access	81	93	81	93	81	93	0	0

Table A.7: hikey64 Call fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	264	303	275	315	265	304	3	4
Cache L1I miss	0	1	0	3	0	1	0	1
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	201	254	201	254	201	254	0	0
Branch misspredict	2	2	3	3	2	2	0	0
memory access	87	97	87	97	87	97	0	0

Table A.8: hikey64 Reply recv fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	383	455	414	829	387	479	9	93
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	1	1	0	0	0	1
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	183	210	183	210	183	210	0	0
Branch misspredict	0	0	0	0	0	0	0	0
memory access	94	107	94	107	94	107	0	0

Table A.9: tx1 Call fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	392	428	415	480	396	439	7	14
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	201	254	201	254	201	254	0	0
Branch misspredict	0	0	0	0	0	0	0	0
memory access	102	112	102	112	102	112	0	0

Table A.10: tx1 Reply recv fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	400	416	416	420	409	417	7	2
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	187	224	187	224	187	224	0	0
Branch misspredict	0	0	0	0	0	0	0	0
memory access	0	0	0	0	0	0	0	0

Table A.11: haswell Call fastpath

counter	min(b)	min(rt)	max(b)	max(rt)	avg(b)	avg(rt)	std(b)	std(rt)
cycles	388	412	392	420	389	416	1	1
Cache L1I miss	0	0	0	0	0	0	0	0
Cache L1D miss	0	0	0	0	0	0	0	0
TLB L1I miss	0	0	0	0	0	0	0	0
TLB L1D miss	0	0	0	0	0	0	0	0
Instruction exec.	203	280	203	280	203	280	0	0
Branch misspredict	0	0	0	0	0	0	0	0
memory access	0	0	0	0	0	0	0	0

Table A.12: haswell Reply recv fastpath

B Modified API

B.1 New Invocations

B.1.1 Scheduling context

SchedContext - Bind

`seL4_Error seL4_SchedContext_Bind`

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.

SchedContext - Unbind

`seL4_Error seL4_SchedContext_Unbind`

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.

SchedContext - UnbindObject

`seL4_Error seL4_SchedContext_UnbindObject`

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.

SchedContext - Consumed

`seL4_Error seL4_SchedContext_Consumed`

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.

SchedContext - YieldTo

`seL4_Error seL4_SchedContext_Consumed`

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.

B.1.2 Sched_control

SchedControl - Configure

`seL4_Error seL4_SchedControl_Configure`

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.