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

January 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.

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, which is prevented by ARINC.

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 less-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.

Level	*Impact*
Catastrophic	Failure may cause a crash
Hazardous	Failure has a large negative impact on safety or performance
Major	Failure is significant, but less than hazardous (passenger discomfort or increased crew workload)
Minor	Failure is noticeable (passenger inconvenience or changed route)
No Effect	No impact on safety, aircraft operation or crew workload

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

However, mixed-criticality systems are about more than basic consolidation; they can achieve more than traditional physically isolated systems. By allowing untrusted, less critical components to share hardware and communicate with more critical components, far more complex software can be introduced to the system, for example heuristic software, or internet connected software, which would be far too complex to certify to the highest criticality standard, but is 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 reason about their safety. The implementation platform will be the seL4 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 **??** 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 be a 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.

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

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.

systems, with some modern cars containing over 100 processors [Hergenhan and Heiser, 2008].

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. However, such standards are restrictive in three forms: 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 However, one could argue that it is possible to simply consolidate hardware from systems of the same criticality. Ignoring the fact that this may not be possible, mixed-criticality systems also bring opportunities for new types of systems. This is because, by definition, isolated isolated cannot interact. AAVs and other autonomous vehicles provide an excellent motivations for systems where sub systems of different criticality share resources: meaning they must share hardware. 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 breaks 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*.

1.2 Contributions

Given their improvements to SWaP, function and efficiency, mixed-criticality systems offer great advantages over the traditional physical isolation approach. Such systems have strong requirements, and as such, the contributions of this thesis are operating system design and mechanisms for achieving the following:

- Temporal and spatial isolation are essential to providing the same guarantees as their physically isolated counterparts.
- Asymmetric protection across criticalities is required, which means that higher criticality sub-systems are prioritised over lower ones: if success of a high criticality task may cause a failure in a low criticality task this is permitted, but not vice versa.
- Trusted resource sharing mechanisms, which allow tasks of different criticalities to safely share resources.

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, and as that is entire research field in itself, although we take care not to introduce any.

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. Finally 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 basic software, time is fungible: it does not matter when the software is run, as long as it does get run.

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

2.1.1 Types of real-time task

The term *task* in real-time theory is used to refer to a single instruction stream. How tasks are realised by an operating system — as a processes or threads — 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 can 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. 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 mights 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 on how late a deadline miss may occur, referred to as *tardiness* [Devi, 2006]. Another is to express an upper bound on the percentage of deadline misses, which is easier to measure but less meaningful, as this allows for unbounded execution time. 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.

Best effort tasks do not have temporal requirements that 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.

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

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

The sporadic task model considers a real-time system as a set of tasks, $A_1, A_2, ..., A_n$. 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 $A_{i1}, A_{i2}, ..., Ain$. Each task has an execution time, C_i, and period, T_i, which represents

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 execution requirement 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, $a_{ij} \geq t_{ij}$

Table 2.1: Parameters and notation for the sporadic task model as used in this document.

the minimum inter-arrival time between jobs. When a T_i has passed and a job can run again, it is said to be *released*. When a job is ready to run, it is said to have *arrived*. 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 completed 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 2.1 shows pseudocode for a sporadic task and Table 2.1 summarises the notation which is used throughout this thesis.

Sporadic tasks have deadlines relative to the 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).

	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]

Scheduling can be either *static* or *dynamic*. Static or *off-line* scheduling involves a pre-computed set of tasks and schedule that are executed. No new tasks can be added and schedules cannot change. Dynamic or *on-line* scheduling involves a set of tasks that can change and where scheduling decisions are made according to the current state of the system.

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 requires less complexity. The schedulability test conducted at admission time is referred to as an *admission test*.

There is an absolute limit on the feasibility of task sets. The *total utilisation* of a task set is the sum of all of 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

$$\sum_{i=0}^{n} \frac{e_i}{p_i} \leq 1$$

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

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

Tasks often do not use all of their execution requirement, any execution remaining in is referred to as *slack*. For HRT tasks, slack is a consequence of pessimistic WCET values. Slack also occurs for SRT tasks as they may be variable 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 referred to 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. Of the two definitive scheduling algorithms fixed priority rate monotonic and earliest deadline first, the former is fixed and the latter is dynamic. Both algorithms are defined along with schedulability tests for the periodic task model in the seminal paper by Liu and Layland [1973].

2.2.1 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 challenge present in FP scheduling. Two well established techniques are rate monotonic and deadline monotonic, both of which are optimal with respect to FP scheduling. Rate monotonic (RM) priority assignment [Liu and Layland, 1973] allocates higher priorities to tasks with higher rates.

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. **??** shows an example FPRM schedule.

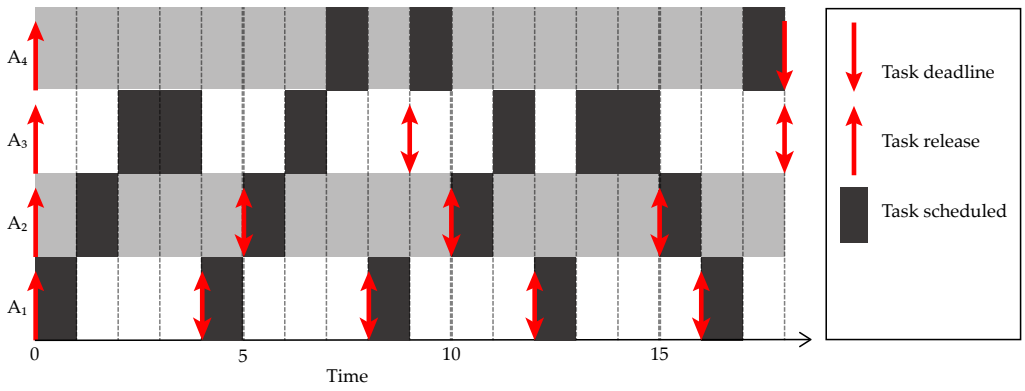

Figure 2.1: An example FPRM schedule using the task set from Table 2.2.

2.2.2 Earliest Deadline First Scheduling

The earliest deadline first (EDF) algorithm is theoretically optimal for scheduling a single resource, with no algorithmic capacity loss; the is 100% of processing time

can be scheduled. This is due to EDF being a dynamic algorithm. Priorities are assigned by examining the deadlines of each ready job; jobs with more immediate deadlines have higher priorities. Figure 2.2 illustrates how the task set in Table 2.2 is scheduled by EDF, highlighting the places where tasks are scheduled differently than FPRM.

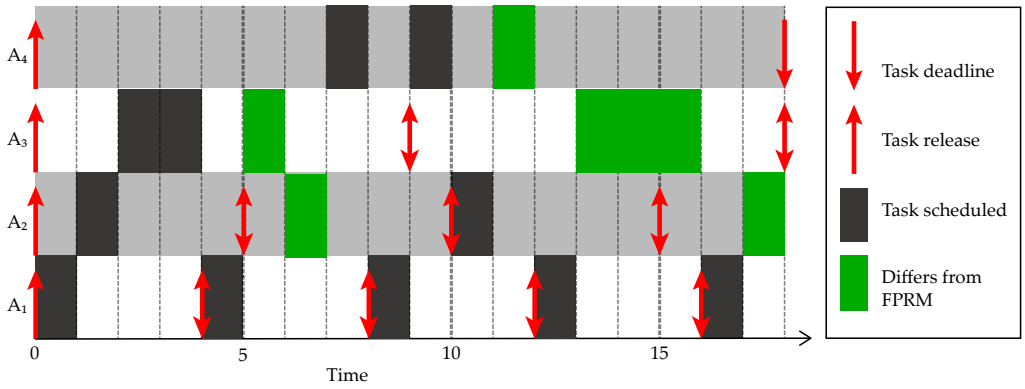

Figure 2.2: An example EDF schedule using the task set from Table 2.2.

2.2.3 Earliest Deadline First vs. 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. FP is mandated by the POSIX standard, possibly due to these misconceptions. One common argument for fixed-priority rate monotonic (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 and how they behave under overload. 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. 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

[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.

behaviours are desirable is subject to context. Under transient overload conditions both algorithms can cause deadline misses.

2.2.4 Multiprocessors

Both fixed and dynamic scheduling algorithms scheduling can be used on multiprocessor machine 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 if 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 access in a mutually exclusive fashion is called a *critical section*. Generally speaking, tasks lock access to a 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 synchronised, 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 accesses 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 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 priority inversion that a task will experience is the sum of all critical section accesses in the system.

2.3.2 Priority Inheritance Protocol

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

To understand this, consider a task set with n tasks, $A_1, ..., A_n$, where each tasks 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 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 Highest lockers' protocol / Immediate ceiling priority protocol

Under highest lockers protocol (HLP), also known as the immediate priority ceiling protocol, resources are assigned a 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 HLP is that all priorities of clients to servers 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 Priority Ceiling Protocol

The priority ceiling protocol (PCP) combines the previous two approaches, and avoids deadlock, excessive blocking and excessive preemption. In addition to considering the priorities of tasks, PCP 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 holders priority is boosted to the resources 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. PCP results in less blocking overall than

HLP, 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, 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

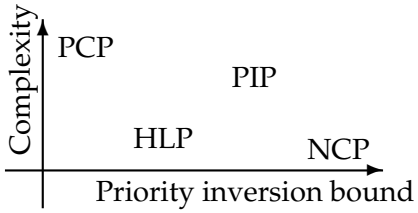

Figure 2.3: Comparison of real-time locking protocols based on implementation complexity and priority inversion bound.

Figure 2.3 compares the different uniprocessor locking protocols, showing that PCP 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 HLP falling between the two. HLP 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* based 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

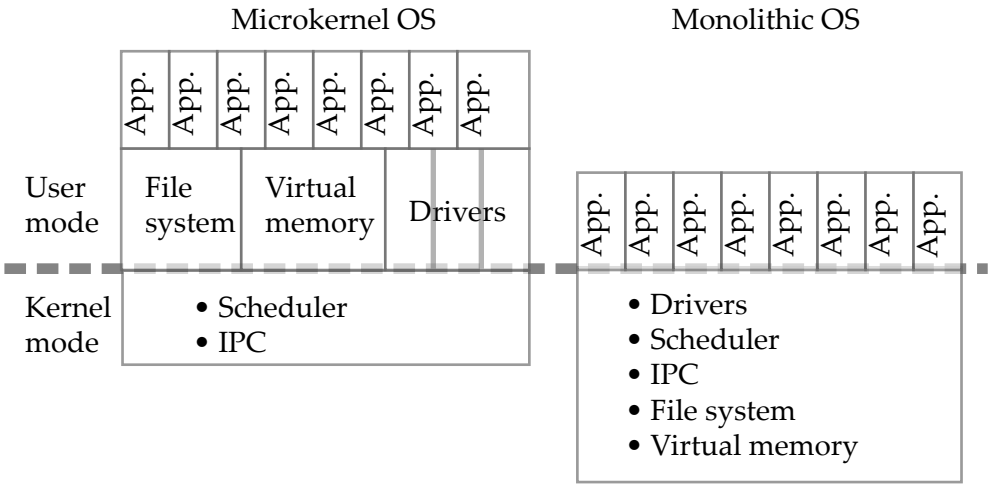

Figure 2.4: Structure of a microkernel vs. monolithic operating system.

examined so far can be used under global scheduling, Brandenburg [2011] showed that partitioned approaches suffer far less cache overheads than global scheduling.

multiprocessor priority ceiling protocol (MPCP) [Rajkumar, 1990] is a modified version of PCP for multiprocessors, which 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. multiprocessor stack resource policy (MSRP) [Gai et al., 2003] is a spin-lock based protocol, which can be used for FP and EDF scheduling and 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 alone, and many more sophisticated locking protocols exist, however we do not survey them here.

2.4 Operating systems

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

Common OSes, such as Windows, Mac OS X 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 2.4 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.

Which services and utilities are included as part of the microkernel varies per implementation. In larger kernels thread scheduling, memory allocation and some

device drivers are included in the kernel. In others, subsets of these functionality are included and the kernel and others delegated to user-mode.

Microkernels are far more amenable to security domains, 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.

2.4.1 Open vs. Closed Systems

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

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

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

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

2.4.2 Real-Time Operating Systems

An 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 effects 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 impossible to guarantee [Blackham et al., 2012]. The overheads of RTOS operations like interrupt handling and context switching must also be considered when determining whether deadlines can be met.

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

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

2.5 Summary

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

3 Temporal Isolation & Asymmetric Protection

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

However, 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 might argue that the method of over-provisioning systems in order to never face such a scenario is sufficient and should not be changed. 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.

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 or 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 an RTOS. 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 different 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 systems where it is not worth conducting WCET analysis or execution time is dependant 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 is has O(n) complexity in the number of tasks, the authors recommend using a dedicated scheduling processor such that overhead will not effect 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 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. 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 fewer 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 run-time. Lamastra et al. [2001] introduce bandwidth inheritance across CBS servers applied to different resources, providing temporal isolation for additional resources other than processing time.

Rate-based 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 [Lehoczky et al., 1987] wake every period, checks if there are any pending tasks and 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 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 tasks 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 $A_{11}, A_{12}, ..., A_{1n}$. 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 is only complete at exactly the implicit deadline. Then a second job may arrive arrives 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 [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 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 [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 [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 Resource sharing

Like the scheduling algorithms, the locking protocols presented in Section 2.3 do not work if tasks are untrusted: in all of 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 2.4.3, 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: PCP, HLP or PIP. Charging the reservation, and which reservation to charge, is more interesting. de Niz et al. [2001] describe the possible mappings between reservations and resources consuming those reservations, and comes down to the following choices:

Bandwidth inheritance Tasks using the resource run on their own reservation. If that reservation expires and their 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 tasks 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.

We address this topic further in the next chapter, when we evaluate survey existing operating systems.

3.3 Asymmetric Protection

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.

There are many varied definitions of mixed criticality 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. Systems where criticality can be specified explicitly (regardless of SRT/HRT specification) also exist.

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.

Scheduling algorithms for mixed-criticality systems are a hot-topic in the real-time community. In this section we briefly present a few such algorithms.

3.3.1 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 utilization 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 proven 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 unmanned aerial vehicles(UAVs). Many models of real-time systems assume that WCETs for real-time tasks are stable and can be calculated. However, UAVs 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.

3.3.2 Mixed Criticality and Certification

An approach to mixed-criticality scheduling that involves two different WCET estimates arises due to the pessimism of certification authorities (CAs). High criticality tasks in safety-critical systems require certification, conducted by a CA. Note that tasks of lower criticality may not require certification.

CAs provide WCET estimates that must be schedulable, however they are often very pessimistic. This results in a tasks 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. An EDF based algorithm is presented in Baruah et al. [2011], which is generalised to an unlimited amount of criticality levels. An FP based approach is implemented in Pathan [2012], where run-time monitoring is used to switch between criticalities. Some tasks may be dropped in order to better utilise processors without violating certification requirements.

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 in a safe way.

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, with a focus on how each one provides mechanisms for temporal isolation, asymmetric protection, and resource sharing.

4.1 POSIX

The portable operating system interface (POSIX) standard specifies real-time operating systems interfaces [Harbour, 1993] which influence a large amount of OSes. Scheduling policies specified by POSIX are shown in Table 4.1.

Faggioli [2008] provides an implementation of `SCHED_SPORADIC`, which Stanovic et al. [2010] use 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 in 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

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 HLP to prevent priority inversion, recall Section 2.3.3.

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

servers depending on load, where the transition involves reducing the maximum number of replenishments to 1 and merging available refills.

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

Although POSIX provides SCHED_SPORADIC which can be used for temporal isolation (however flawed), the intention of the policy is to contain aperiodic tasks. Since SCHED_SPORADIC allows tasks to run at a lower priority when they have exceeded their allowed budget in any given period, it follows that the locking protocols PRIO_INHERIT and PRIO_PROTECT would continue to operate – although the excess time is not billed to the task. If a task never releases a locked resource, temporal isolation is completed violated. 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.

4.1.1 Linux

While Linux cannot be considered a real-time operating system for high criticality applications, it is frequently used for lower criticality applications with SRT demands. Additionally, Linux is often used as a platform for conducting real-time systems research. Linux has fixed-priority preemtive 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 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 completly 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 to Linux [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 mechansims for assymettric protection.

On the resource sharing side Linux provides real-time locking via the POSIX as per Table 4.2.

4.1.2 Real-time Linux Extensions

A large amount of projects exist that 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. Interrupts are handled by the virtualisation layer, and only directed to Linux if required. This means that 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.

LITMUS^{RT} [Calandrino et al., 2007] is an extension of Linux that allows for pluggable real-time schedulers to be easily developed for testing multiprocessor schedulers.

Linux/RK [Oikawa and Rajkumar, 1998] is a resource kernel implementation Linux that is often used to implement new scheduling algorithms. ZS scheduling [de Niz et al., 2009] was implemented and tested using Linux/RK.

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 virtualising Linux to run SRT and best effort applications.

4.1.3 Commercial RTOSes

There are several widely deployed RTOSes used commercially. The majority implement part or all of POSIX. We present three popular alternatives: VxWorks, QNX Neutrino and RTEMS.

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 priority inheritance or immediate priority ceilings for resource sharing, as well as higher level primitives for these.

QNX Neutrino [Co., 2010] is a commercial, microkernel-based RTOS 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 [River, 2008] is a monolithic RTOS deployed most notably in aircraft and spacecraft. It supports FP scheduling with a native POSIX-compliant scheduler. VxWorks also has a pluggable scheduler framework, allowing developers to implement their own, in-kernel scheduler.

There are many other RTOSes used commercially, but the general pattern is POSIX-compliant, FP scheduling and resource sharing. Deos and PikeOS are two more examples. This brief survey shows that FP scheduling is dominant in industry due to its place in the POSIX standard. Although temporal isolation is sometimes provided with the possibility of bounded bandwidth via CBSs, asymmetric protection is not provided. In addition to scheduling, code-bases are generally large and complex, and beyond the grasp of modern WCET analysis. Although all of these RTOSes are deployed in safety critical systems, their support for mixed-criticality applications is questionable if non-existent.

4.2 Microkernels

Microkernels, as introduced in are small operating systems kernels which contain the minimum amount of software to implement an OS. meal-time concepts are not new microkernels.

Real-Time First we review some common microkernel based concepts, before presenting an overview of relevant microkernel designs.

Many microkernels with varying real-time support have arisen since.

4.2.1 Capabilities

Capabilities [Dennis and Van Horn, 1966] are an established mechanism for fine-grained access control to spatial resources. 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.

4.2.2 IPC

4.2.3 Timeslice Donation

4.2.4 Scheduling contexts

Many kernels distinguish between a threads execution context (the registers, stack etc) and scheduling context (access to processing time) in order to allow the scheduling context to transfer between threads for accounting purposes. This is a more explicit form of timeslice donation: when the scheduling context is passed between client and server, the scheduling context of the client is billed for activity on the servers behalf. Designs differ as to whether scheduling context donation is required or optional. Some designs allow multiple execution contexts to be attached to one scheduling context, such that the scheduling context forms a secondary run queue. In others, multiple scheduling contexts can be bound to one execution context, allowing the execution context to execute on any scheduling context with available execution time.

Steinberg et al. [2005] scheduling context consisted of a time quanta and a priority and were donated across inter-process communication (IPC).

4.2.5 Examples

Real-time Mach [Mercer et al., 1993, 1994] first introduced the concept of processor capacity reserves, which is the first form of scheduling contexts in microkernel design. In Real-time Mach, scheduling context donation was compulsory where processor capacity reserves were used to implement temporal isolation, however threads could be reserved or non-reserved threads. Threads which had exhausted their reservation were able to be scheduled by a second level time-sharing scheduler that scheduled all of the unreserved threads, if no unexhausted reservations were present in the scheduler. Real-time Mach used fixed-priority scheduling and rate-monotonic analysis to conduct an admission test for reservations in the kernel. If a server exausted the donated reservation, it would finish the operation using the time-sharing scheduler. Finally, Real-time Mach provided PIP over IPC to avoid priority inversion [Tokuda et al., 1990].

KeyKOS [Bomberger et al., 1992]

EROS [Shapiro et al., 1999] is a capability based operating system designed to demonstrate the viability of capability-based systems in terms of performance. EROS 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. EROS also used what is known as an *entry* capability, which allowed the holder to invoke the services provided by a program within a particular process. EROS uses the same scheduling and resource sharing policies as Real-time Mach, using capacity reserves, however these are not accessed via the capability system.

DROPS (Dresden Real-time OPerating System) [Härtig et al., 1998] is an L4 based real-time microkernel that also takes the resource reservation route to temporal isolation. The scheduling scheme in DROPS allows a process to reserve a priority for an amount of cycles during a time period. Page colouring is used to reserve parts of the caches for real-time tasks, significantly decreasing upper bounds on WCET.

NOVA [Steinberg and Kauer, 2010] is a capability-based microkernel aimed at virtulisation. NOVA provides fixed-priority round-robin scheduling, with priority inheritance across IPC [Steinberg et al., 2010]. Priorities in NOVA reflect importance: it is not a real-time kernel, and periodic scheduling is not available. NOVA provides bandwidth inheritance (BWI) through the microkernel mechanism of donation is used, so NOVA does not provide concrete reservations, but when a timeslice expires other tasks waiting for the resource *help* the blocked task, where the blocked task excecutes on the pending tasks timeslice. If the task holding the resource does not release it, all pending tasks will block forever.

Fiasco.O.C is an L4 microkernel where scheduling, accounting, enforcement and admission are all implemented in the kernel, with the motivation that these functions would have high overheads if implemented at user-level. Resource reservations are realised as *scheduling contexts* which act as timeslices: a budget paired with a priority, and a replenishment rule. Some scheduling mechanisms are exposed to the user; Lackorzyński et al. [2012] alter the system call interface to support

multiple mixed-criticality guests. They find that the only way to avoid scheduling integrity violations is to export scheduling information to the host virtual machine (VM). Guests can change scheduling contexts on priority switches such that the host schedules guests with enough time to schedule all tasks. Additionally, guests associate scheduling contexts to interrupt service routines(ISRs).

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

seL4 is a microkernel that is particularly suited to safety-critical, real-time systems with one major caveat: the lack of real-time scheduling support. Three main features of seL4 support this claim: it has been formally verified for correctness [Klein et al., 2009] and other properties [Sewell et al., 2011]; All memory management, including kernel memory, is all at user-level [Elkaduwe et al., 2006]; Finally it is the only OS to date with full WCET analysis [Blackham et al., 2011]. The scheduler in seL4 has been left intentionally underspecified [Petters et al., 2012] for later work. The current implementation is a placeholder, and follows the traditional L4 scheduling model [Ruocco, 2006] — a fixed-priority, round-robin scheduler with 256 priorities.

MINIX 3 [Herder et al., 2006] is a traditional microkernel with a focus on reliability rather than performance. MINIX 3 has been adapted for real-time with temporal isolation provided by resource servers [Mancina et al., 2009]. The implementation is designed for bandwidth servers to be implemented at user-level. This is achieved by adding system calls allowing for the kernel's best-effort scheduling policy to be switched to EDF per process. Eight system calls are added, as well as semantics for the kernel to up-call the resource server when a real-time task is blocked, unblocked, exits or exhausts its budget. Real-time tasks execute as the second-highest priority tasks in the system, the highest being reserved for the bandwidth server. The advantage of this implementation is that different types of bandwidth servers can be implemented on top of MINIX 3 without kernel modifications. Although not covered in the paper, this could be extended to a split, user-level RBED implementation, however the overheads of such an approach are unclear. The MINIX 3 implementation is a good example of implementing bandwidth servers with minimal kernel modifications. MINIX 3 is not aimed at hard-real time, supporting only mixed-criticality between SRT and best effort processes.

4.3 Other research kernels

COMPOSITE is a component-based OS with similar with goals to a microkernel, however with a more dominant focus on support for fine-grained components. In COMPOSITE all four mechanisms are implemented at user-level, however unlike a microkernel, COMPOSITE contains device drivers inside the kernel.

A hierarchical scheduling framework, HIRES [Parmer and West, 2011], is used in 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. 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 of the goals of a resource kernel according to their own policy.

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. Like MINIX 3, Tiptoe uses the CBS approach, although it uses variable bandwidth servers which allow for bandwidths to be altered. Tiptoe implements bandwidth servers inside the kernel.

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*. Each CPU-driver is single-threaded and non-preemtible. , although not explicitly real-time, implements a version of RBED for managing distributed SRT processes. Barrelfish schedules dispatchers, which are roughly equivalent to processes. Execution context and scheduling context are not split. When messages are sent between dispatchers on barrelfish the sender can choose to yield to the receiver or not by flags on the message.

TODO Quest first, then quest v. As a separation kernel, Quest does not provide resource sharing mechanisms by definition.

Quest-V [Danish et al., 2011] is a separation kernel / hypervisor with fixed priority scheduling. Temporal isolation is provided by sporadic servers, however I/O and normal processes are ditinguished statically: I/O processes use polling servers and normal processes use sporadic servers, with a maximum replenishment length of 32.

4.3.1 Summary

OS	*Scheduler*	*Isolation*	*Asymmetric protection*	*Resource Sharing*
Linux	FP + EDF	CBS	✗	PIP, HLP
POSIX	FP	sporadic server (SS)	✗	PIP, HLP
Linux/RK	TODO	TODO	✗	TODO
RTEMS	FP + EDF	CBS	✗	PIP, HLP
QNX	FP	SS	✗	TODO
VxWorks	FP	TODO	✗	TODO
Real-Time Mach	FP	deferrable server (DS)	✗	PIP
EROS	FP	DS	✗	PIP
Minix 3	TODO	TODO	✗	TODO
Tiptoe	TODO	TODO	✗	TODO
Barrelfish	EDF	RBED	✗	Timeslice donation
DROPS	FP	TODO	✗	TODO
NOVA	FP	✗	✗	BWI
Quest-V	FP	SS	✗	✗
seL4	FP	✗	✗	✗
Composite	user level	TODO	TODO	TODO

Table 4.3: Summary of mixed criticality system (MCS) support available in existing operating systems.

5 seL4 Basics

So far we have provided a general background on real-time scheduling and resource sharing. Finally we will 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: the lack of real-time scheduling support. Three main features of seL4 support this claim: it has been formally verified for correctness [Klein et al., 2009] and other properties [Sewell et al., 2011]; All memory management, including kernel memory, is all at user-level [Elkaduwe et al., 2006]; Finally it is the only OS to date with full WCET analysis [Blackham et al., 2011]. The scheduler in seL4 has been left intentionally underspecified [Petters et al., 2012] for later work and as a result has a very informal notion of time. The current implementation is a placeholder, and follows the traditional L4 scheduling model [Ruocco, 2006] — a fixed-priority, round-robin scheduler with 256 priorities.

Systems can currently choose between a restrictive implementation of temporal isolation or a system that presents timing behaviour that is impossible to analyse beyond trivial examples[1]. In this section we will present the current state of relevant seL4 features in order to highlight deficiencies and motivate our changes. We will outline the existing scheduler, the API curiosity that is `Yield`, and how IPC interacts with scheduling, followed by an analysis of how the current mechanisms can be used in real-time systems.

5.1 seL4 concepts

First we introduce the basics of seL4: kernel memory management, capabilities, IPC and signalling. Figure 5.1 shows the legend for diagrams in this section.

5.1.1 Capabilities

As a capability-based OS, access to any functionality in seL4 is via capabilities (recall Section 4.2.1. Capabilities to all system resources are passed 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. Capabilities spaces are analogous to address spaces for virtual memory, although capabilities can be moved and copied between different spaces.

[1] Whether analysis is possible in a trivial example beyond one thread is questionable.

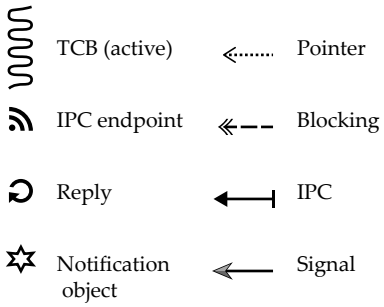

Figure 5.1: Legend for diagrams in this section

All capabilities can be copied, potentially changing their access rights, except *control* capabilities, which can only be moved and not copied. Control capabilities provide the rights to do specific operations, e.g. the *interrupt control* capability is used to obtain a capability to a specific interrupt number.

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.

5.1.2 Kernel memory management

All kernel memory in seL4 is managed at user-level and accessed via capabilities (recall Section 4.2.1, which is key to seL4's isolation and security, but also essential for understanding the complexity of kernel design.

Capabilities to objects can be copied, which creates another capability to the same object, potentially with different access rights, but not a copy of the object itself. Similarly capabilities to objects can also be deleted, but only when the last capability to an object is deleted will the object itself be deleted. Object capabilities contain a pointer to the actual kernel object.

All memory in seL4 starts as *untyped* memory, capabilities to which is granted to the initial task in the system as part of the boot process. The initial task can then create all types of memory, including further untyped, with an operation known as *retyping*. As a result, any object with a pointer to another object must have a back pointer to update should that object be deleted, in order to avoid stray pointers in the kernel.

Table 5.1 gives an overview of basic object types in seL4. While the majority of objects in seL4 have a platform-dependant size fixed at compile time, some are variably sized, e.g. untyped and CNodes, which can be any power of two size, as custom

5.1.3 Thread control blocks

TCBs represent a unit of execution in seL4 and deviate from other object in that they consist of the core TCB, and a CNode which contains a capability to that threads CNode, vspace and IPC buffer.

Object	*Description*
CNode	A fixed size table of capabilities, analogous to a page table in a virtual memory system.
Thread control block (TCB)	A thread of execution in seL4.
Endpoint	Objects which facilitate IPC between threads.
Notification object	Objects which provide a signalling mechanism.
VSpace	A top level page table structure
Interrupt	Allow access to specific hardware interrupts.
Untyped	Memory that can be retyped into other types of memory, including untyped.

Table 5.1: Major object types in seL4, excluding platform specific objects. For further detail consult the seL4 manual [Trustworthy Systems Team, 2017]

.

(a) Phase 1 (b) Phase 2

Figure 5.2: IPC phases: (a) shows the initial IPC rendezvous, (b) shows the reply phase. See Figure 5.1 for the legend.

5.1.4 IPC

IPC is the microkernel mechanism for synchronous transmission of data and capabilities. In the seL4 model, IPC takes place via endpoints, which represent a general communication port. Any thread with a capability to an endpoint can send and receive messages on that endpoint, subject to the access rights. Most messages sent fit into registers, however messages exceeding this platform dependent size are placed in the Messages sent via IPC are stored in the IPC buffer, a kernel-writeable frame per thread.

IPC can be one-way, where the sender blocks until the message is sent then may continue (`send()` , or two-way, where the sender blocks until a reply message is sent (`call()`. One-way IPC may also be non-blocking, where the message is only sent of a receiver is already present. Multiple senders and receivers can use the same endpoint, and which act as FIFO queues.

Two-way IPC has two phases, as Section 5.1.4 illustrates: rendezvous and reply.

Figure 5.2a demonstrates the rendezvous phase, where regardless of the order of operations, when one thread blocks (**??**) on the endpoint and another thread sends on that endpoint then message is sent. This occurs for both one-way and two-way IPC.

The reply phase, illustrated in Figure 5.2b, only occurs if the sender used `call` to send the IPC. In this case, a one-off *resume* capability is generated, which the caller (*A*) blocks on, and the receiver (*S*) sends a reply message to, waking the caller.

(a) Notification (b) Bound notification

Figure 5.3: Signals via a notification object. See Figure 5.1 for the legend.

Receivers can save the *resume* capability 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.

Given IPC performance is critical to overall system performance in a microkernel, 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 thread being woken in the scheduler,
3. the thread to be switch to has a valid address space and has not faulted,
4. and the message fits in registers.

Endpoint capabilities can be minted with unique, unforgeable badges which are delivered to the receiver with the rest of the IPC message. This provides a mechanism for identifying senders.

5.1.5 Signalling

Notification objects facilitate signalling in seL4, either from other threads via `send()` or from interrupts. The signals are non-blocking, as the badge of the notification capability is simply bitwise ORed with a data word stored in the notification object itself. When a receiver blocks on a notification object (`recv()`), if the badge has already been set it the receiver will receive it immediately, otherwise the receiver will block immediately. Receivers can also poll a notification object with `nbrecv()`.

Figure 5.3a shows signal handling, where a thread A receives a signal and an interrupt at the same time. In this case, the interrupt handler has a badged capability to the notification object, *N*, as does the signaller. Both badges will be combined with a bitwise OR and delivered to the receiver.

Signals can also be received by threads waiting on IPC, as shown in Figure 5.3b. A TCB object is bound to a notification object, establishing a link between them. Badges can be used to determine whether a message was received from the endpoint or a signal on the notification object.

5.1.6 System calls

Over the last few sections we have indirectly presented several system calls for IPC and signalling. Table 5.2 shows all the system calls available in seL4. Communication with the kernel itself is modelled as IPC by using `call()` on kernel object

System call	*Description*
`send`	One-way blocking IPC or non-blocking signal.
`nbsend`	One-way non-blocking IPC or non-blocking signal.
`recv`	Block waiting for a message or signal.
`nbrecv`	Poll for a message or signal.
`call`	Send an IPC and wait for a reply, or invoke the kernel.
`reply`	Send a message to the resume capability in the TCB CNode.
`replyrecv`	`reply` and `recv` combined in one system call.
`yield`	Surrender current timeslice.

Table 5.2: seL4 system call summary.

capabilities. Each capability has a set of *invocations* which allow for methods on that object to be used e.g. binding a TCB and notification object.

5.2 Scheduler

The scheduler in seL4 is priority-based round-robin with 256 priorities (0 — 255), implemented as an array of lists: one list of ready threads for each priority level. At a scheduling decision, the kernel chooses the head of the highest-priority, non-empty list, and removes it from the relevant scheduler queue, which is referred to as *Benno Scheduling*. The kernel previously used *lazy scheduling*, leaving the current thread in its run queue, however this was replaced in favour of Benno scheduling to reduce the WCET of the kernel. This is different to a previous kernel implementation which left the currently running thread in its run queue, however this increased the kernels WCET and was removed.

Scheduling decision complexity is $O(1)$ as a two-level bit field tracks the highest priority with a runnable thread.

Kernel time is accounted for in ticks, with a static tick length defined at compile time (CONFIG_TIMER_TICK_MS). Threads have 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. The reload value of the timeslice is also defined at compile time (CONFIG_TIME_SLICE).

In an unrealistically simple system, where threads run at the same priority and do not communicate, it is possible to analyse temporal behavior on seL4: threads will run for the timeslice value in round-robin order. However, the allocation of ticks to threads is not actually that simple due to yield behaviour, preemption and IPC.

5.2.1 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 a 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. However, even this becomes unclear once there is more than one thread at a priority: if two threads are running, they can both access 50% of processing time. If one of two threads blocks, the other gets 100% of processing time and vice versa. There is no way to enforce a certain processor allocation and how CPU time a thread receives is up to priorities and the behavior of other threads in the system, which is impossible to guarantee.

5.2.2 Accounting

Due to the tick-based nature of the seL4 scheduler, when ticks are accounted complicates the scheduling as the precision is reduced to the tick granularity. Small tick intervals lead to an accurate scheduler but increase the base interrupt load, while larger intervals result in imprecise scheduling. For example, signals and other preemptions bill the partially consumed tick to the preempting thread, which will suffer a shorter timeslice.

Similarly, the `yield` system call will not alter the timeslice of the current thread, and only donates a portion of the current tick to the next thread in the round-robin scheduler.

IPC, as introduced in Section 5.1.4, complicates the timing behaviour of the kernel further. When a message is sent to a target thread, and that thread is a higher priority than any other thread in the scheduler that thread is switched to immediately, by passing the round-robin scheduler. Note that other threads in the scheduling queue are not starved, as when one of the rendezvous partners exhausts its timeslice the scheduler will choose the next round-robin thread. However, *when* this can be hard to determine due to the timing of ticks. Consider the case where threads A and B, both at the same priority, and are exchanging IPC messages constantly. Another thread C is running at the same priority, and all threads have full timeslice fields. C will be scheduled when one threads timeslice is exhausted, i.e when either A or B has been preempted `CONFIG_TIME_SLICE` times. As a result C could be scheduled after at minimum `CONFIG_TIME_SLICE` ticks (if one of the IPC partners is preempted for every tick, or at maximum after `CONFIG_TIME_SLICE * 2 - 1` ticks (if both partners are preempted for ticks). This example is illustrated in Fig. 5.4.

Ultimately, the current tick-based scheduler is insufficient for accurately billing threads for time consumed.

5.2.3 Domain scheduler

A recent addition to the seL4 kernel adds the ability to guarantee complete temporal isolation and deterministic scheduling between sets of threads, using the concept of scheduling *domains*. Threads are assigned to domains, each of which has a separate array of lists of threads over the priority range. Each domain runs for a fixed

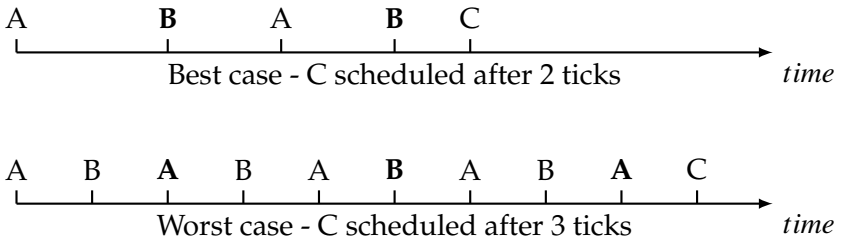

Figure 5.4: Scheduling in the case of IPC and variable execution times. A and B are IPC partners, C is a third thread. A, B and C run at the same priority. `CONFIG_TIME_SLICE` is 2. Bold indicates the thread that is preempted by the timer tick. In the best case, B takes both timer ticks and C is scheduled after 2 ticks. In the worst case, A takes 2 ticks and B takes 1, such that C is scheduled after 3 ticks.

amount of ticks, and domains are scheduled sequentially and deterministically. Cross-domain IPC is delayed until a domain switch, and `yield` between domains is not possible. When there 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 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.

However, in 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.

5.3 RT support

We introduced basic seL4 concepts and terminology, and investigated mechanisms that effect timing behaviour in the kernel: the scheduler, 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. As the previous section established, the domain scheduler offers low utilisation for strictly closed systems with strict temporal partitioning and no preemption, so is clearly insufficient.

The priority scheduler could be leveraged to implement a rate-monotonic scheduler. 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.

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

Table 5.3: Comparison of existing seL4 scheduling options – nothing ticks all of the boxes.

To build a dynamic system with temporal isolation and high CPU utilisation, one could implement a trusted timer component at user-level. This component would be the highest priority thread in the system, and could pause threads to prevent them from overrunning their assigned rate. 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 5.3 shows a comparison of all of the available scheduling options in the current version of seL4– no option provides all of the qualities we require. Clearly, the kernel needs something more. In the next section we will our model for a more principled approach to time by extending the baseline seL4 model presented in this chapter. We incorporate using the principles of resource kernels and with the aim of support diverse task sets, including those for mixed-criticality systems.

6 Design and Model

6.1 Design goals

Our goal is to design microkernel mechanisms for the support of mixed-criticality workloads. As we have seen, many existing systems conflate criticality, temporal sensitivity and trustworthiness in a single value: priority. Our claim is that how these attributes are conflated is policy that is specific to a system.

Temporal sensitivity refers to how sensitive an task is to when it gets access to the processor. Every task is sensitive to time: if no processor time is given to an application, it will never produce a result. However, For best-effort activities, time is fungible, in that only the amount of time allocated is of relevance. In contrast, for a HRT tasks, time is largely non-fungible in that is has no value if the allocation is after the deadline; SRT tasks are in between.

Criticality reflects the importance of an process to the overall system mission. In some systems it may reflect the impact of failure [ARINC] or the utility of a function. An MCS system should degrade gracefully, with tasks of lower criticality suffering degradation before higher criticality tasks.

Trustworthiness whether a task is trustworthy or not. This could be due to the level or certification, the correctness of a task, or where the code for the task has come from. For example, in many open systems tasks using the processor are downloaded from an untrusted source.

Figure 6.1 shows these attributes with a scale for each. As discussed in TODO the majority of existing operating systems use a singe value, *priority* to represent all of these values. With respect to real-time scheduling, priority is used for another thing entirely simply reflects scheduling order, used to achieve optimal processor use.

The model we present defines operating systems primitives for building systems that leverage these properties in a way that is appropriate to that system, which can be implemented in middle ware or an OS personality on top of the trusted computing base.

6.1.1 Temporal sensitivity

Tasks that are temporally sensitive require control over exactly when they are scheduled, and for what duration. For systems where tasks are independent this

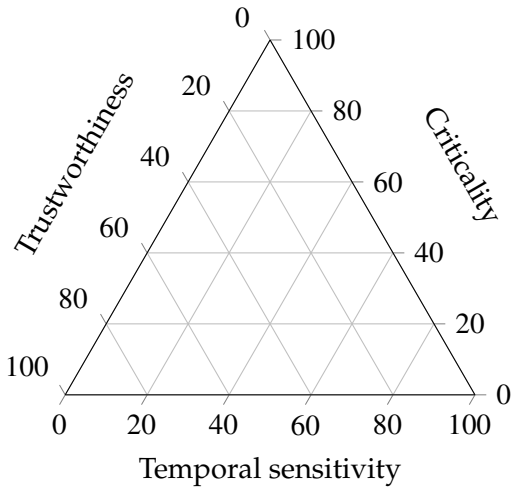

Figure 6.1: Commonly conflated attributes of systems software

applies only to processor time. However, in real systems where resources are shared this applies to every shared resource.

In contrast, tasks that are not temporally sensitive should not be restricted by the model: all tasks should not be required to participate in complex scheduling

6.1.2 Criticality

Support for tasks of different criticalities requires mechanisms for graceful degradation and asymmetric protection, even when sharing resources.

6.1.3 Trustworthiness

If tasks do not trust each other they need to be able to to be isolated. Further, they should be able to share resources and maintain that isolation in order to facilitate mixed-criticality systems. Therefore we need mechanisms for processor reservations which allow for temporal isolation, and mechanisms for resource sharing where trust is not assumed between tasks that access those resources.

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 overbooked while providing guarantees for the highest priority tasks.

In this section we first address how we integrate resource kernel mechanisms with seL4 and explain priority decoupling and our model for temporal isolation. We then discuss alternatives for resource sharing and present mechanisms to facilitate sharing between tasks of different criticalities and different real-time models.

Our design goals are as follows:

- Verifiable mechanisms
- Policy freedom - only in the kernel if it has to be
- Performance
- Security - trust

6.2 Design alternatives

6.2.1 Resource Reservations

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 2.4.3, 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 fact, in the context of a microvisor, the resource managers may actually be OS guests.

In seL4, mechanisms for physical memory, device memory, interrupt and I/O port management are exposed to the user via the capability system. As a result, the only resource that the kernel needs to provide reservations for is processing time.

To define processor time reservations, we rely on the sporadic task model as presented in **??**. A reservation is realised as new kernel object, a scheduling context, which contains scheduling related task parameters. One scheduling context can be bound to a thread at a time, allowing that thread to use the reservation. Threads without reservations are not considered runnable, as they have no rights to the processor. A scheduling context is considered *active* if it currently has budget available, and *inactive* otherwise.

Scheduling contexts have been used in kernels before – Fiasco introduced the concept in Steinberg et al. [2005], where the contexts contain a priority, budget and period. They are also used in NOVA [Steinberg and Kauer, 2010], where they contain a timeslice and a priority.

In our model, scheduling contexts are decoupled from priority, and contain sporadic task parameters – a budget and a period. Instead, the priority remains as a property of a thread which allows for a very efficient implementation of HLP, which will be discussed in Section 6.3. Decoupling the priority also allows us to overbook the processor: since only threads with highest priorities and active reservations will run, reservations can be made that exceed 100% of the processor.

6.2.2 Admission

Admission tests in our system are intended to be conducted dynamically at run-time, or offline, as per user-level policy. In our design, we consider admission tests to be the domain of the user, as the type of schedulability test used is subject to the policy of the system. Thus, the kernel places no restriction on the creation of reservations apart from minor validity checks (i.e $b \leq p$). 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 overload 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 a single process. Delegatable reservations that would allow other processes to change their parameters can be implemented at user-level.

6.2.3 Scheduling

There are two questions with regard to scheduling policy:

- Should the scheduler be in the kernel at all?
- Which base scheduling algorithm should be used, EDF or FP?

First, in the context of a high assurance system, we believe scheduling, accounting and enforcement should all be included in the kernel. This reduces the overheads of increased context switching inherent in hierarchically scheduled systems. Additionally, it allows for a solid foundation of temporal isolation to be provided by the kernel: without this, we cannot verify important properties like information flow.

Real-time literature is utterly divided between which real-time scheduling algorithm should be deployed as the core of real-time systems, EDF or FP. In the systems community, FP is far more dominant due to its inclusion in POSIX and its compatibility with existing, priority-based best effort systems. Support for FP is important as it is most prevalent in industry. It is quite common for kernels to provide EDF scheduling at a specific priority level on top of an FP based scheduler. However it is not emulate FP scheduling on top of EDF.

Therefore our model uses FP scheduling as remain a core part of the kernel. EDF scheduling can be implemented by user-level using green threads or a user-level scheduler for kernel threads. We will demonstrate this in a specific priority with

little overheads. This 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 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.

6.2.4 Accounting

There are two potential ways to account for time in a kernel: in discrete ticks, or in cycles. The former implies that the kernel is tick-based while the latter implies a tick-less kernel.

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 higher the preemption overhead, and vice versa.

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. Since real-time tasks require greater precision and suffer WCET penalties for every preemption, we have chosen to convert seL4 to a tickless kernel, although the impact of timer reprogramming has yet to be quantified. We track accounting details, such as the last time scheduled and the amount of unused budget, in the current threads scheduling context alongside sporadic task parameters.

6.2.5 Enforcement

While systems must be able to use kernel mechanisms to enforce scheduling reservations, we do not want to force all tasks in a system to be subject to enforcement, as enforcement does imply accounting and tracking overhead. If a task is truly trusted, no enforcement is required, as in standard HRT systems. Therefore, our model must allow for enforcement to be optional.

Then the question is how enforcement should be undertaken: given seL4's capability based design, and user-level management of kernel memory, we must rule out isolation policies that require the kernel to have full knowledge of all threads. Of the policies discussed in Section 3.1.3 this rules out priority exchange servers and slack steal.Given that deferrable servers do not enforce temporal isolation, we choose a combination of polling and sporadic servers similar to Quest-V [Danish et al., 2011] with a significant difference: we allow threads to specify the amount of replenishments available up to a statically configurable max, rather than denoting kernel threads as either I/O threads (1 replenishment - a polling server) or not (MAX replenishments), users can then tune threads that are subject to enforcement as per their system policy.

Replenishments are tracked in scheduling conterxts, and threads are only runnable while they are bound to an active scheduling context. If the replenishment of the current thread expires, that thread is removed from its run queue and placed in a release queue of threads waiting for their next replenishment. When a replen-

ishment is eligible, threads are placed back at the end of the run queue for their priority.

Tasks can opt to have *timeout faults* delivered if their budget is exhausted before the current job is completed, or if their deadline is not met. 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. Unlike page fault handlers, if a timeout fault is not handled, the thread can continue once the next refill is available, which is why timeout faults are optional.

6.2.6 Priority assignment

The mechanisms we have presented can be leveraged to implement standard 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 them to exceed the processing time reservation that the scheduling context represents.

However, the system we have designed offers far 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 priorities 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.

6.2.7 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 [2014].

Under slack scheduling, low-criticality tasks run in slack time of high criticality tasks. Our system supports this easily: hi 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.

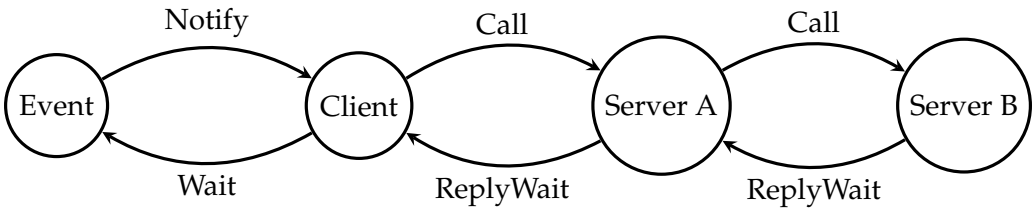

Figure 6.2: Client-Server scenario

On a temporal fault, the scheduling thread can extend the reservation of the high priority task and suspend all low priority tasks.

6.2.8 Summary

The scheduling, accounting and enforcement mechanisms presented are sufficient to support temporally isolated, fixed-priority or EDF scheduled real-time threads. Additionally, we have maintained support for best-effort threads, and have added the ability to provide asymmetric protection for mixed-criticality workloads. Implementation details of how each type of task can be supported is explained further in Chapter 7.

In the next section, we discuss the options available for resource-sharing in the system we have designed and outline the mechanisms we have designed.

6.3 Resource Sharing

For resource sharing, we outlined a further three core mechanisms taken from resource kernels – prioritisation, charging and enforcement. We indicated in Section 6.2.1 that scheduling contexts are separate from thread control blocks in order to support resource sharing. In this section we explore the design choices for supporting resource sharing, and describe our model.

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.

6.3.1 Scenarios

Our analysis of is guided by two models for the flow of time between communicating threads in a system: a client-server or remote procedure call (RPC) scenario, and a data-flow scenario.

Client-Server

The client-server scenario involves clients contacting servers in an RPC style fashion: the client sends a message to the server, the server executes an operation on the clients behalf and replies. Servers can be clients too: a server may make a nested request to another server on behalf of a client. There can be multiple servers, and multiple clients in operation at the same time.

An example of this scenario is illustrated in Figure 6.2, albeit with just one client and nested servers.

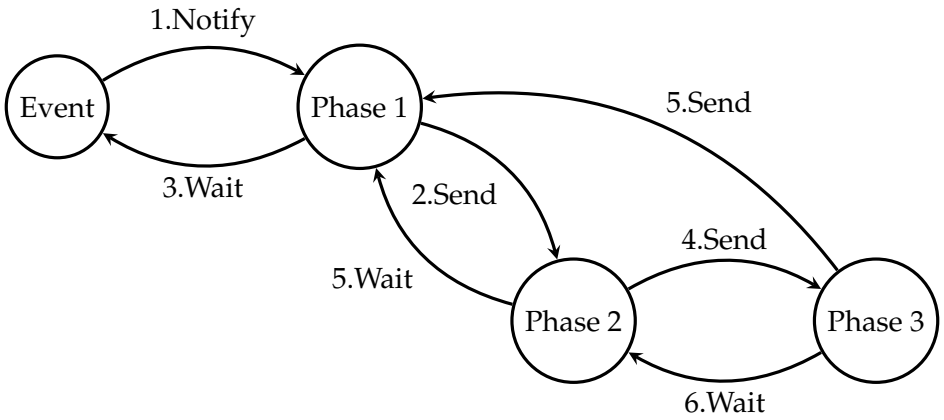

Figure 6.3: Data-flow scenario. Numbering indicates the ordering of events.

Data-flow

In this scenario, as illustrated in Figure 6.3, an external event triggers the release of a job. The job passes through a chain of activities, all doing different forms of processing. Only one job can be active in an activity at a time, and the next job cannot be released until the previous has finished (consistent with the sporadic task model). Forks are permitted in the chain, although the forks must be choices: one job cannot trigger two jobs, as a processor reservation cannot be transferred to two jobs at once. Cycles and sequences are permitted in the chain.

Both scenarios can be combined to form a larger system.

6.3.2 Resource sharing models

We consider three existing models for resource sharing over IPC: reservation-per-thread, timeslice donation, and migrating threads. In all models, the shared server or phase is assumed to have a known WCET, which should match that of the strictness real-time task using the server.

Reservation-per-thread

In the reservation-per-thread model, reservations are assigned to every thread in the system, clients and servers alike. Server budgets must be sufficient to serve all clients or the system will be bottle-necked on the server. In order to calculate sufficient budgets, the number of client requests must be known *a priori*.

This design 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. Servers could be placed at a higher priority with a 100% CPU reservation, but this is also problematic as the server can then monopolise the entire system in the case of rogue clients.

While our resource sharing mechanisms should 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.

Timeslice donation

Fiasco.O.C [Steinberg et al., 2010] implements timeslice donation, where servers run on the reservation of the client, at the priority of the client. Such a design allows for temporal isolation as clients can only make requests while their reservation is active, and servers deplete the clients reservation.

As the server inherits the clients priority, this design has the same preemption overheads PIP, discussed in **??**. Implementing priority inheritance requires the kernel to track dependencies between servers and clients, which requires a large amount of book keeping.

Further challenges arise if the clients reservation expires while the server is still executing. Without further mechanisms, the resulting scenario leaves the server unable to service other clients, as it is in the middle of serving another clients request.

In the context of a hard real-time system, expiry will not occur, as client reservations are based on WCET. However, systems that support other real-time models – soft real time and rate based tasks – must solve the issue. In Fiasco.O.C expiry is ameliorated by further applying priority inheritance, which the authors refer to as *helping*. If a server is stuck servicing a request, and a second client attempts to make a request of the server, the server will finish the previous clients request on the second clients budget and priority.

Dependency tracking in this model quickly becomes very complex when clients and servers use chained requests. Additionally, because priorities must be returned to clients, this model does not support the data-flow scenario outlined earlier.

Migrating threads

COMPOSITE [Parmer, 2010] takes a different approach to solving 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.

We perceive the migrating thread model as valid for systems suited to multi-threaded servers, but do not think it is suitable for microkernel mechanisms to require multi-threaded servers. Like the reservation-per-thread model, we choose to support this approach, but not enforce it, unlike COMPOSITE.

Summary

We have outlined reservation-per-thread, timeslice donation and migrating threads. Our model supports reservations-per-thread and migrating threads, while introducing a design based on timeslice donation without the shortcomings: we remove the need for dependency tracking trees and add support for the data-flow scenario.

6.3.3 Scheduling Context Donation

Finally we present *scheduling context donation*, the new mechanism we have added to the kernel to support resource sharing with temporal isolation. This model is inspired by timeslice donation, however we utilise HLP rather than priority inheritance to reduce preemption and remove the need for a dependency tracking tree. First we present the basic model, then in the next section we will describe design alternatives for handling budget expiry, which complicates the scenario.

Using HLP is not without drawbacks. Recall that under HLP servers run at the ceiling of the priorities of all clients that will access them. Many claim that such a policy is inappropriate for open systems, as *a priori* knowledge is required of all clients, servers and their priorities in a system. However, this claim is somewhat unsubstantiated: a system should have a general policy on priorities, especially a system running real-time components. For example, if a mobile phone operating system allowed all apps to run at the highest priority levels in the system then the phone could be rendered inoperable by apps running at this priority. Therefore, we believe that is is not unreasonable to deploy HLP in our scenario.

Although PCP offers greater processor utilisation than HLP, 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. Using ceilings instead of priority inheritance results in a very clean model when budget expiry is excluded. Servers inherit the scheduling context of their clients, basically inheriting their budget, while executing on their own priority, resulting in reduced preemption overheads. When the server replies to clients, the budget is returned. The kernel has no need to track the dependency relationship, as the reply IPC is sufficient to return the scheduling context. By removing dependency tracking, we can support the data-flow model: instead of replying, a dataflow component simply sends the scheduling context along to the next component.

Unfortunately, scheduling context donation is complicated by budget expiry. However, since we aim to support rate-based, best-effort and soft real-time threads, budget expiry is required. The next section will canvas the design options and outline our proposed solutions and mechanisms.

6.3.4 Budget Expiry

In a hard real-time system, we can assume that a client 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.

Figure 7.5 illustrates the problem, and Table 6.1 outlines the notation used. A makes a request to S, such that A_{sc} is donated to S_{tcb}. A_{sc} expires while S is executing A's request. Then B makes a request to S, however S cannot take new requests as it is still still servicing A's request and will not be available until A_{sc} is replenished.

Our goal is to support systems that run best-effort, soft real-time and hard real-time tasks where all tasks can share resources while preserving temporal isolation. Resource sharing may occur between tasks sharing a real-time model (i.e where all

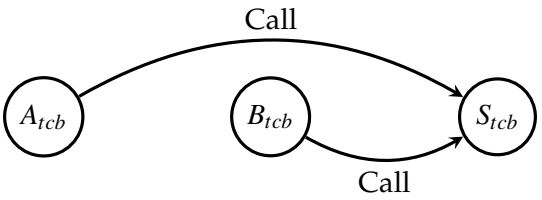

Figure 6.4: **Budget expiry:** client A makes a request of server S, but runs out of budget during the request and is blocked. Another client, B, attempts to make a request but finds S blocked.

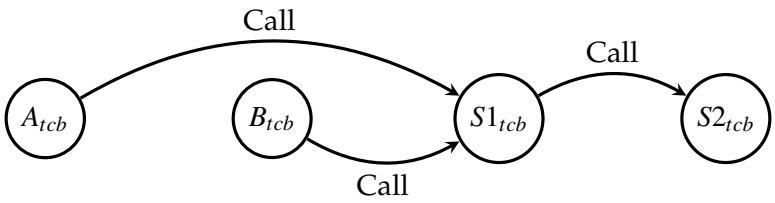

Figure 6.5: **Nested budget expiry:** client A makes a request of server S1, which makes a nested request to S1, but runs out of budget during the request and is blocked. Another client, B, attempts to make a request but finds S1 blocked on S2 who has no budget to complete the request.

Notation	**Meaning**
A, B, C, ...	A task, or user application
$S1$, $S2$, $S3$ ...	Resource servers.
A_{sc}	Scheduling context of task A.
A_{tcb}	Thread control block of task A
A_b	The budget of task A
A_d	Deadline of task A
A_p	Period of task A.
A_e	The WCET of task A.

Table 6.1: Summary of notation used.

tasks are soft real-time), or between tasks with different real-time models. In the latter case the resource server must comply with the strictest real-time model of all clients. This means that for a server that serves hard real-time and soft real-time tasks, the server must have a known WCET even though the soft real-time tasks do not.

We outline several potential mechanisms for handling budget expiry: blocking, denying unfeasible requests, emergency reservations for completing requests, rolling back unfinished requests, raising an exception on budget expiry, bandwidth inheritance, and leveraging migrating threads. Each is analysed according to the following qualities:

Schedulable utilisation The longer the maximum blocking time of each server request, the larger client reservations must be to handle potential blocking.

This reduces overall system utilisation, impacting schedulability. What is the impact on schedulability?

Temporal containment Tasks sharing a resource cannot be truly temporally isolated, as one task can block the other by simply using the resource. Temporal containment places a limit on the amount of time one task can block another, by bounding the maximum blocking time due to a resource contention. Does the policy offer temporal containment?

Nested budget expiry Nested requests (see Figure 7.6) is required in both the client-server and data-flow scenarios. Can the policy support expiry in the case of nested requests?

User-level implications What changes to user-level applications are required by the mechanism?

Kernel implementation implications What changes to the kernel are required by the mechanism.

Blocking

Blocking is the default case, where budget expiry is not handled at all and the server is blocked until the clients reservation is replenished. In the example in Figure 7.5, S_{tcb} is blocked until A_{sc} is replenished, and therefore so is B_{tcb}. While clearly not a viable solution we present analysis for the blocking technique as a baseline.

In the best case, if A_{tcb}'s request completes when A_{sc} is recharged, then B is only delayed by $A_p + S_e$. However, in a pathologically bad (or malicious) case, where $A_b \leq S_e$, A_{sc} will need to be recharged multiple times before the server request completes. An upper bound on the delay to other clients caused by one budget expiry is $\frac{S_e}{A_b} * A_p$, which would result in an incredibly pessimistic schedulability test. If the delay is not factored into the schedulability test, then temporal isolation is violated as A can cause B to miss deadlines.

As far as implementation considerations go, neither the kernel or user-level see any impact as this policy requires no implementation. Chained invocations only make the schedulability test worse.

Deny requests

A simple approach to solving this problem is to deny requests if a client does not have sufficient budget. The kernel could be configured with the WCET of each server and simply deny or post-pone client requests where the budget is insufficient.

This would have a 0 schedulability penalty, but would starve clients and deny requests even if there was no contention for the resource. Forcing clients to have available budget that matches a servers WCET could also see requests denied that would have been possible to complete, since WCET estimates are generally orders of magnitude beyond average execution times. Additionally, nested requests would not work, so although simple and effective, this policy is impractical.

Emergency reservation

Under this policy, servers are assigned an emergency reservation (ER), which the kernel switches the server to when a clients reservation expires.

Server reservations need to be sufficient: if the servers emergency reservation expires, the schedulability impact will degrade to that of the blocking case. Sufficient reservations are derived by [de Niz et al., 2001], although their work assumed servers always execute on their own reservation (never billing a clients reservation), the bounds still apply. A sufficient reservation is the sum of all non hard-real time client requests, replenished at the highest rate of all of those clients, in order to guarantee that no hard real time task is blocked by the server for more than the length of a single server request. This large, pessimistic reservation for the server must then be incorporated into the schedulability test, which then suffers a large utilisation penalty.

De Niz et al. [2001] reduce this penalty by observing that the main source of pessimism in this estimate is due to the use of a single reservation, whose period must be the highest period of all non hard real-time clients. The schedulability penalty can be reduced by assigning the server one reservation per client, such that each reservation can have a period matching the client. However, for a dynamic system this requires dynamic allocations of scheduling contexts, and one wonders if such effort were to go into creating and assigning reservations for servers, then what was preventing clients reservations from being specified sufficiently initially? For this reason we do not consider the multiple reservation alternative.

As long as the server reservation is sufficient, the maximum blocking time encountered by a task is S_e, proving temporal containment. However the schedulability test is impacted by the size of the server reservation. Additionally, this mechanism results in client requests always being finished immediately after budget expiry, even if the resource is not contended. This results in unrelated threads lower than the servers priority being blocked, and will impact total utilisation and restrict deadline constraints.

Nested requests in this model can be handled, when $S2$ finishes its request, the expired A_{sc} is passed back to $S1$, which switches to its emergency reservation since the received reservation is depleted.

Using a single emergency reservation, the kernel must be able to track two reservations per thread, and switch between them on budget expiry. The kernel will need extra logic to track replenishment of emergency reservations. Multiple reservations make the kernel even more complicated.

User-level implications are minimal, except that a user must specify the parameters for emergency reservations for each server.

Rollback

Under the rollback mechanism, clients whose budget runs out during a server request have their request cancelled and the server rolls back to a previous state, as if the request did not occur. The kernel sends a notification to the server in the form of an upcall. The rollback itself could either run on the clients reservation or a dedicated server reservation for rollback.

Using a reservation to rollback faces the same schedulability penalty as the emergency reservation: the rollback budget must be enough to service all potential

client requests. The penalty is less severe assuming that the rollback time is less than the time for the server to execute the request. Sending a notification to the server requires the kernel to know the WCET value for the server rollback process, such that a notification can be sent in time. If the reservation, or notification, is not sufficient then behaviour will degrade to that of the blocking case.

Nested requests make the kernel notification option complicated. If a nested request fails, in the client-server scenario, then kernel must send a notification to the nested server with enough time in the clients budget for both the nested server and the first server to rollback, including the IPC time. Emergency reservations allow nested requests to be more easily handled: the kernel can detect that a server is running on an expired reservation and switch to the emergency reservation. However, if the server needs to rollback previous nested requests by recontacting servers this model is also insufficient.

User level implications are as follows:

- Client reservations are returned to them and the request fails, meaning clients must be able to handle failed requests.
- Starvation is possible: if clients don't have enough budget to complete a single server request.
- Rollback is compulsory even if the resource is not contended. A possible fix is to only rollback the request if the server is contended, switching to the rollback reservation when the resource is contended.
- All servers must implement rollback, or degrade to the blocking case.
- Servers must be preemptible for the rollback, requiring servers to be lock-free or to use atomic primitives provided by the kernel.

The kernel would require a mechanism for notifying the server that a rollback is required, and the server needs to be able to execute atomic updates in the face of that.

Exception

When a clients budget expires, an exception is delivered to a temporal fault endpoint, and the thread waiting on that endpoint handles the exception with its own reservation. The temporal fault endpoint is separate to the standard fault endpoint for a thread.

The thread waiting on the endpoint could then choose a strategy: it could reset the server (doing a rollback) or extend the current budget. In some cases, this would require the server to be preemptible but it would be specific to the design.

Schedulability impact and temporal isolation would depend on the user-level fault handler.

Nested requests would need to be handled by the exception handler, or the kernel could deliver an exception when an expired scheduling context was sent back via a reply, but only if the reply is not to the client who owns the scheduling context.

Bandwidth inheritance

The bandwidth inheritance (BI) strategy takes the helping concept from Fiasco, without the priority inheritance. When a B_{tcb} makes a request to blocked S_{tcb}, it donates B_{sc} to S_{tcb} and A_{tcb}'s request is completed on B_{sc}. The difference to priority inheritance is that the server still runs at the ceiling priority.

This provides temporal isolation, with the maximum blocking time for a single resource access being S_e. Schedulability-wise, a pessimistic schedulability test requires that all hard real-time jobs contain enough budget to help every resource that they access. This is effectively the same penalty as under the emergency reservation scheme, except that the budget is included in the hard real-time tasks instead of in a separate reservation. This should result in a less pessimistic schedulability test, as it can take into account the different periods tasks, rather than being the maximum time that clients could need the emergency reservation at the highest rate.

The main difference is that client requests finish on their own budgets in cases with no contention, rather than finishing immediately.

Nested requests require inheritance chains to be followed in this model.

Migrating threads

The migrating thread (MT) model used by COMPOSITE inherently solves the problem of budget expiry by requiring multi-threaded servers: since servers spawn a new thread for each request, budget expiry for one client does not block the server for other clients. This option offers nearly perfect temporal isolation, although this is achieved mostly by pushing real-time synchronisation problems to user level. Not only are servers forced to be multi-threaded, but they must either be lock-free or use a kernel-provided atomic primitives to implement server critical sections.

Summary

Table 6.2 summarises all of the analysis presented so far. Immediately we rule out Block, as it provides no temporal isolation and Deny, as it cannot handle nested resources. To further understand the trade-offs between ER, Rollback, Except., BI, and MT we present Table 6.3 which summarises the implementation issues for each policy.

We exclude the emergency reservation mechanism as it requires *a priori* knowledge of all non-hard real-time resource access patterns in order to specify a sufficient reservation. In this case a system should be able to use a full reservation for the server: resource access patterns for hard real-time tasks should also be known.

Rollback as a policy can be implemented using the exception mechanism, and as a result we provide no kernel mechanisms to require rollback. Exceptions for budget expiry are introduced, in the form of temporal faults. Since not all tasks require temporal faults, we allow a temporal fault endpoint to be specified per thread – if a thread does not have a temporal fault endpoint then no fault will be delivered. This is in contrast to how normal fault endpoints work in seL4 – if there is no fault endpoint, the kernel will fault trying to deliver the fault message and then kill the target thread.

While we do not rule out the migrating thread approach, but we do not force this through kernel mechanisms as in COMPOSITE. In the next section we will outline how such a system can be built on the current primitives. However, we the

	Block	**ER**	**Rollback**	**Deny**	**BI**	**MT**
Temporally contained	✗	✓	✓	✓	✓	✓
Bound on single request PI	$\frac{S_e}{A_b} * A_p$	S_e*	S_r*	0	S_e	0*
Schedulable utilisation	Worst	Medium	Better	Best	Better	Best
Nested Requests	✓	✓	✓	✗	Complex	✓
Non-preemptible servers	✓	✓	✗	✓	✓	✗
Requests succeeds without contention	✓	✓	✗	✗	✓	✓
Requests completes on clients budget without contention	✓	✗	✗	✗	✓	✓
Isolation strength	Worst	Medium	High	Best	High	Best

Table 6.2: Comparison of options for handling budget expiry. The 'exception' option is omitted, as the properties of this policy depend on user-level exception handling.

leave atomic operations provided by the kernel in order to make this approach viable to future work.

Since bandwidth inheritance is the only mechanism that allows threads without known resource access patterns to share resources while avoiding the requirement for preemptive servers, we believe this mechanism to be worth implementing, although systems are not required to use it.

6.3.5 Summary

In this chapter we have outlined our model for introducing resource kernel concepts and resource sharing to seL4, presenting a more principled approach to time. We introduce support for user-level admission tests and add processor reservations to the kernel in the form of scheduling contexts. We have also altered the scheduler, providing optional EDF scheduling and provided accounting and enforcement mechanisms to implement temporal isolation. By decoupling priorities from reservations, asymmetric protection is also supported, a key requirement of mixed-criticality systems.

Finally, we outlined existing models for integrating real-time resource sharing and IPC and present a new one: scheduling context donation, which unlike previous donation over IPC uses HLP rather than PIP, eliminating many overheads while adding the restriction that a policy on priorities be present in the system. Our model supports reservations-per-thread, migrating threads, or scheduling context donation policies for resource sharing.

Budget expiry is the largest concern for systems using scheduling context donation. We solve this using temporal exceptions or bandwidth inheritance, as it is the only mechanism which provides temporal containment while allowing for non-thread-safe, single-threaded resources.

Mechanism	Kernel impact	User-level impact
ER	• Track two reservations per thread. • Doubles length of release queue.	• Must specify emergency reservation parameters.
Rollback	• Track two reservations per thread. • Doubles length of release queue. • Provide rollback upcall mechanism. • Provide atomic operation mechanism.	• Provide rollback functionality. • Servers must be thread safe.
Except.	• Requires significantly more transparency between kernel and user-level	• Must choose and implement a policy compatible with kernel transparency.
BI	• Kernel must follow forwarding chains to handle nested budget expiry. • Kernel must track inheritance links.	• None.
MT	• Provide atomic operation mechanism.	• Servers must be thread safe.

Table 6.3: Kernel and user-level impact of different budget-expiry handling policies.

In the next section we will outline the implementation of our model, covering changes to the kernel, and how user-level can utilise the mechanisms to implement a variety of systems with policy freedom.

6.4 Model

6.5 Summary

7 Implementation

In the previous chapter we presented our model for MCS mechanisms for a microkernel. This section provides the deatils 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

Resume objects are introduced to track scheduling context donation over IPC path when `Call` is used. Futher detail is provided in TODO.

Caller a pointer to the TCB that is blocked on this reply object.

Prev,Next pointers which track the call chain.

7.1.2 Scheduling context objects

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. The core fields in a SCO are as follows:

Period The replenishment period.

Refills A circular buffer of sporadic replenishments.

Consumed The amount of cycles consumed since the last reset.

Core The id of the processor this scheduling context grants time on.

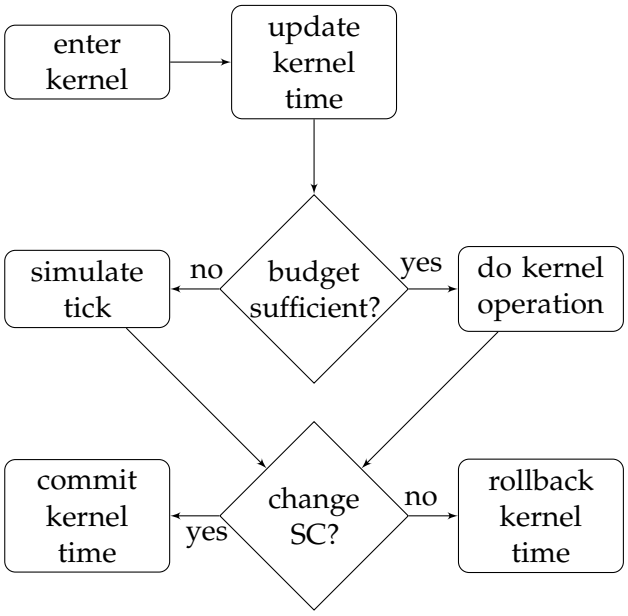

Figure 7.1: Kernel structure

TCB A pointer to the thread 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 to the notification object

Badge A unique identifier for this scheduling context.

YieldFrom Pointer to a TCB that has yielded to this SCO.

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

Max Maximum amount of replenishments for this scheduling context.

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

7.1.3 Thread control block objects

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

7.2.1 Tickless

We convert seL4 from a tick-based to a tickless kernel in order to reduce preemptions and improve scheduling precision, as discussed in Section 6.2.4. As seL4 is non-preemtible (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.

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.

7.2.2 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.

7.2.3 Admission

As established in section Section 6.2.2, 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.

7.2.4 Scheduling Contexts

We introduce a new kernel object type, scheduling contexts, which act as processor reservations and 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.

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`.

7.2.5 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.

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 7.1 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 7.2 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 (7.3) 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).

7.2.6 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

```
uint_64_t check_budget(sched_context_t *sc, uint64_t usage) {
  while (head_refill(sc).amount <= usage) {
    // exhaust and schedule replenishment
    old_head = pop_head(sc);
    usage -= old_head.amount;
    old_head.time += sc->period;
    add_tail(sc, old_head)
  }

  /* handle budget overrun */
  if (usage > 0 && sc->period > 0) {
    // delay refill by overrun
    head_refill(sc).time += usage;
    // merge replenishments if time overlaps
    if (refill_size(sc) > 1 &&
        head_refill(sc).time + head_refill(sc).amount
        >= refill_next(sc).time) {

      refill_t old_head = pop_head(sc);
      head_refill(sc).amount += old_head.amount;
    }
  }
}
```

Listing 7.1: Check budget 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);
}
```

Listing 7.2: Split check routine.

```
void unblock_check(sched_context_t *sc) {
  if (!head_refill(sc).time > now)) {
    return;
  }

  head_refill(sc).time = now;
  // merge available replenishments
  while (refill_size(sc) > 1) {
    if (refill_next(sc).time < now + head_refill(sc).amount) {
      refill_t old_head = pop_head(sc);
      head_refill(sc).amount += old_head.amount;
      head_refill(sc).time = now;
    } else {
      break;
    }

    if (head_refill(sc).amount < MIN_BUDGET) {
      // second part of split_check can leave refills
      // with less than MIN_BUDGET amount.
      // detect them here and merge.
      refill_t old_head = pop_head(sc);
      head_refill(sc).amount += old_head.amount;
    }
}
```

Listing 7.3: Unblock check routine.

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.

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 should result 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.

Real-time threads

Real-time threads are more complicated than 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.

```
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    }
```

Listing 7.4: Example of a basic sporadic real-time task on sel4.

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 be tween the bound thread and the bound endpoint. The current 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. Due to implementation complexities, 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 7.4.

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.

7.2.7 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.

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

7.3.1 Endpoints

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,

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 7.1: Fields in a reply object.

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.

7.3.2 Reply objects

7.3.3 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 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:

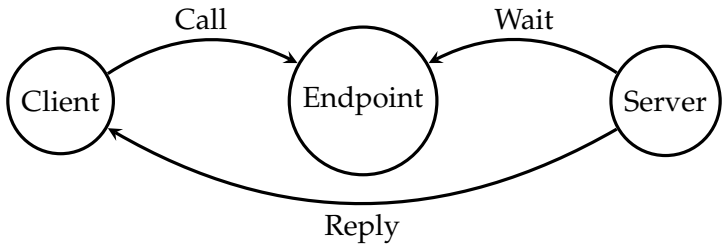

Figure 7.2: Client-server scenario with `Call` and `ReplyWait` and a synchronous endpoint.

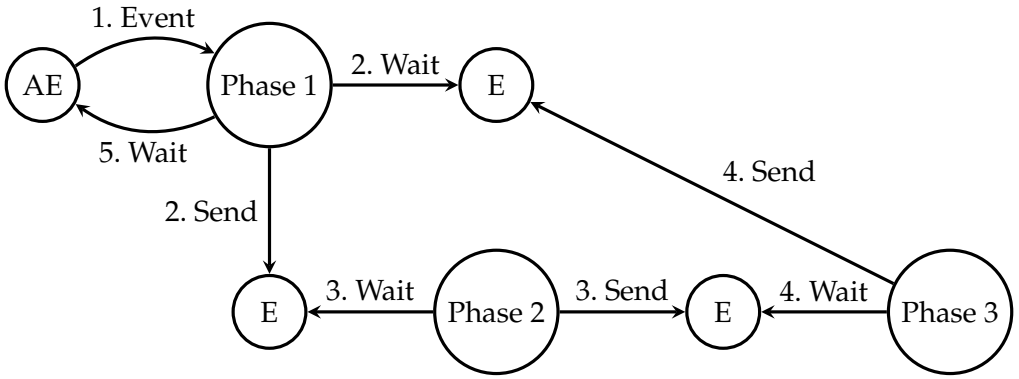

Figure 7.3: Data-flow scenario with endpoints: AE indicates an asynchronous endpoint through which an event occurs, E indicates a synchronous endpoint. Numbering indicates event ordering: a Send and Wait with the same number occur in one system call.

- `Call` (altered) between a caller that has a scheduling context and a callee that does not have a scheduling context donates the callers 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 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.

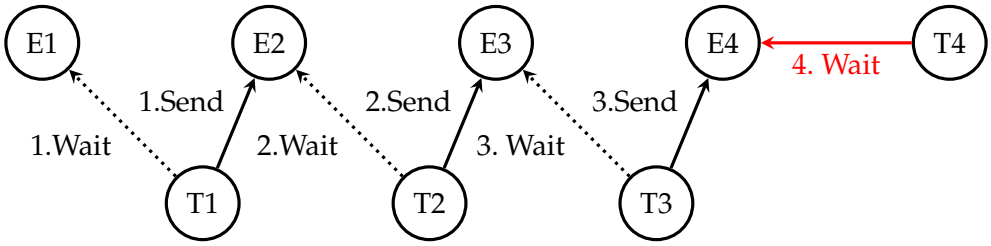

Figure 7.4: A `SendWait` triggering a long-running kernel operation. Threads T1, T2 and T3 call `SendWait` on endpoint pairs (E1, E2), (E2, E3) and (E3, E4) respectively, however each thread blocks on the send. Finally T4 waits on E4, triggering a chain of messages that the kernel would have to process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

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

[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.

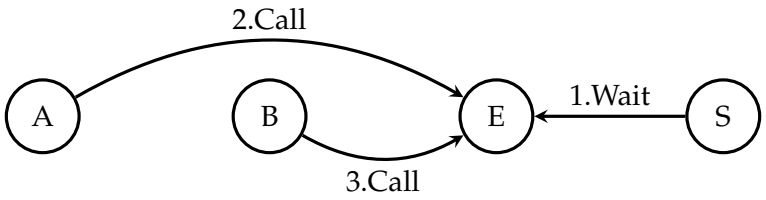

Figure 7.5: Client A makes a request from server S on endpoint E. A's scheduling context is donated from A to S. The budget expires while S is executing the request, blocking S from servicing other requests. Another Client, B, attempts to make a request but finds S blocked.

Revoking a server

What happens if a server is revoked while executing on a clients 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 should only occur 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.

7.3.4 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.

7.3.5 Helping

We introduce a new mechanism to the kernel – helping – which implements bandwidth inheritance as discussed in Section 6.3.4. 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 scheduling context, A_{sc}. S inherits B_{sc} in

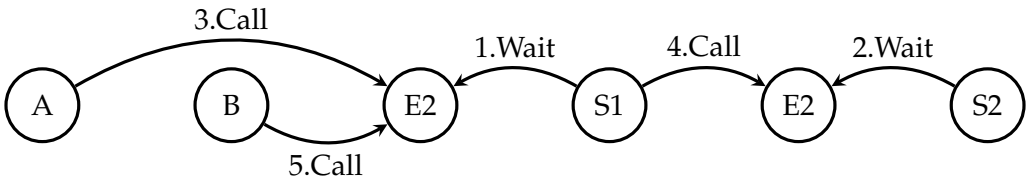

Figure 7.6: Clients A makes a request to server S1 on endpoint E1. S1 receives A's request first, and makes a nested request to another server, S2. A's scheduling context is donated from A to S1 to S2, but the budget expires while S2 is executing. A's budget expires while S is executing on it, blocking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

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).

7.3.6 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.

7.3.7 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 (≤ 2 for x86, ≤ 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 would need 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`.

7.3.8 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 Section 6.3.2 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

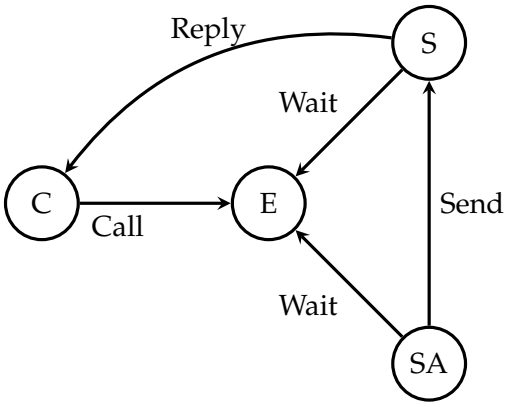

Figure 7.7: Migrating threads with a stack allocating thread, SA.

isolated from each other, threads sharing resources servers using reservation-per-thread are not temporally isolated.

Migrating threads

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 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 clients 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.

7.3.9 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:

Implemented

- Scheduling contexts.
- EDF scheduling.
- User-level admission control.
- Fast-path changes.
- Task completion when a thread waits on its bound asynchronous endpoint.
- Temporal isolation for fixed-priority threads.
- Scheduling context donation.
- Converted kernel to tickless.

To be implemented

- `SendWait`
- Ordered IPC endpoint queues.
- Temporal exceptions.
- Helping.
- Nested Helping.

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 run microbenchmarks on a variety of hardware to show the overheads of the model compared to baseline seL4. Table 8.1 summarises the hardware platforms. SABRE is the verified platform for seL4, 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 which require a network.

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 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

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.

performing a scheduling 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 brackets.

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 time 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 registers (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

[1]The IPC benchmarks already existed for seL4, but were modified to support the 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: Microbenchmarks of seL4 fastpath IPC.

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.

`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 reading the timer with a hot cache as 36 cycles.

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: Microbenchmarks of seL4 slowpath IPC between passive threads.

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 by the side of of 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.

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 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.

8.2.6 SMP

We run an SMP throughput benchmark to show that our MCS model avoids introducing scalability problems on multiple cores compared to the baseline kernel. We modify the existing SMP IPC throughput benchmark for seL4 to run on the

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: Microbenchmarks of 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: Microbenchmarks of seL4 fault IPC between passive threads.

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 SMP 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.

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 [Redis] using Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al., 2010] on baseline and MCS

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: Microbenchmarks of seL4 signal and IRQ latency on seL4 baseline vs. MCS kernels. Standard deviations shown in brackets.

seL4, and compare this against Linux, the Rump unikernel [Kantee and Cormack, 2014] and NetBSD [NetBSD] all on the X64 machine.

For seL4, we use a single-core Rump library OS [Kantee and Cormack, 2014] to provide NetBSD [NetBSD] 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.

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.

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: Microbenchmarks 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.

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.

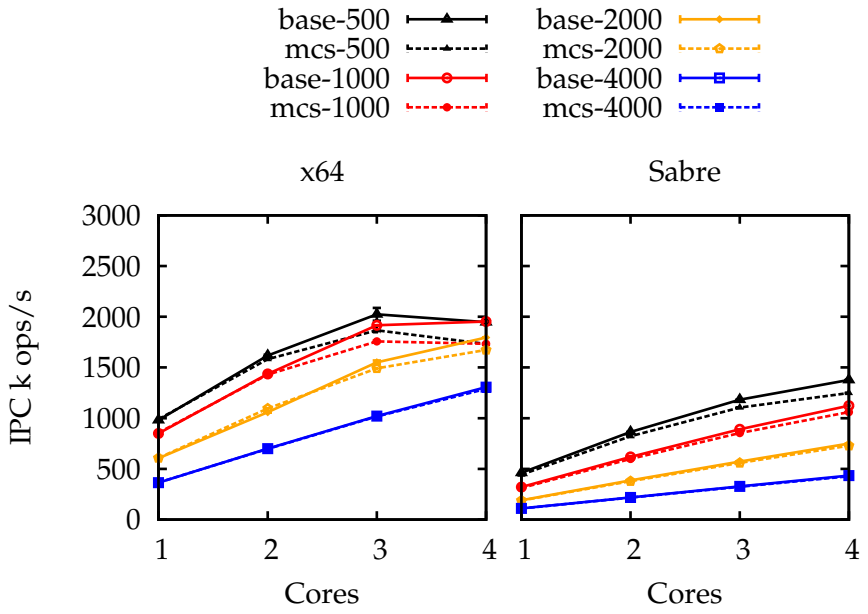

Figure 8.1: Results of the SMP IPC throughput benchmark, baseline seL4 vs MCS. Each series is named *name-N*, where *name* is *base* and *mcs* for the baseline and MCS kernel respectively, and *N* is the upper bound on the number of cycles between each IPC for that series.

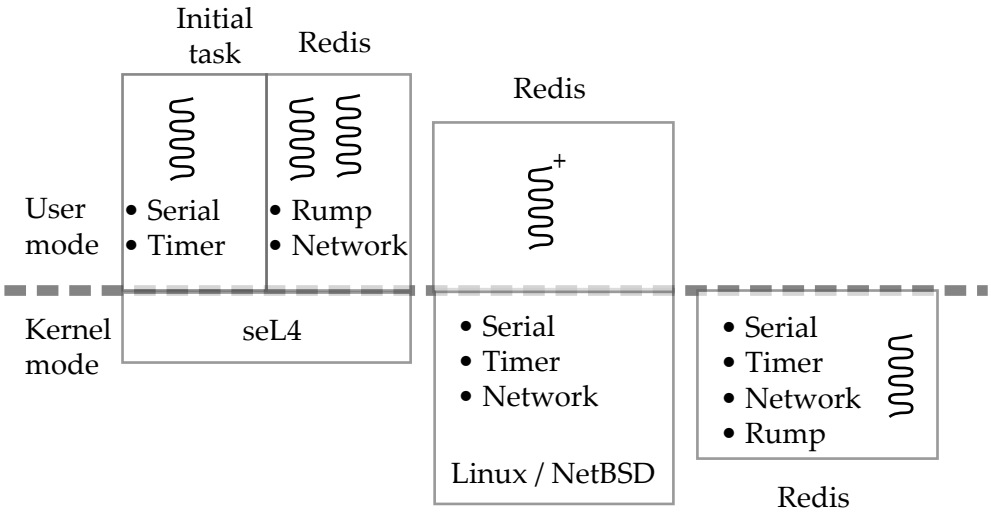

Figure 8.2: System architecture of the Redis / YCSB benchmark on seL4, Linux, NetBSD and Rump unikernel.

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.

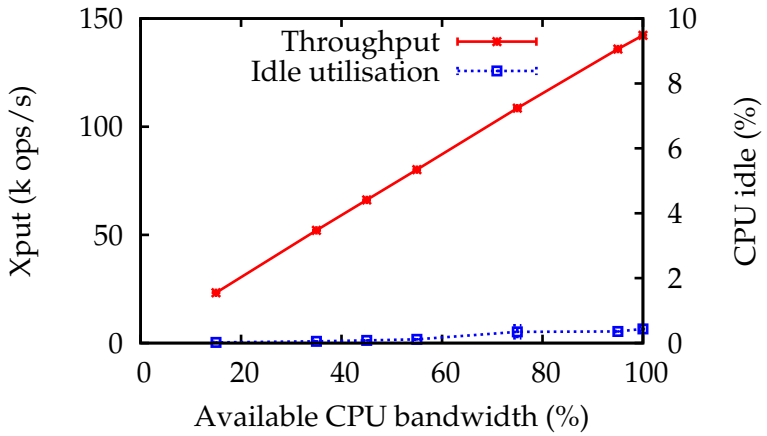

Figure 8.3: Throughput of Redis YCSB workload A and idle time vs available bandwidth.

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 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.

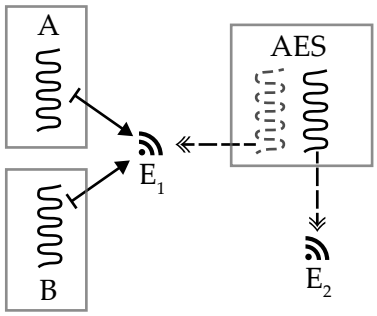

Figure 8.4: System architecture of ipbench benchmark.

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 advanced encryption standard (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.

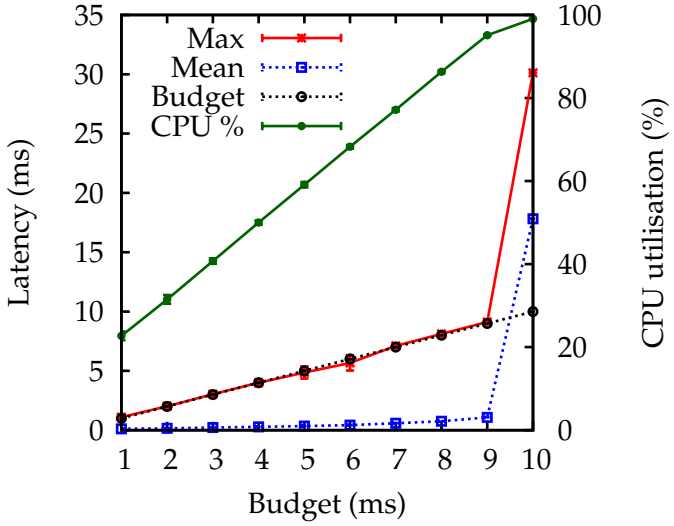

Figure 8.5: Average and maximum latency of UDP packets with a CPU hog running a high priority with a 10 ms budget.

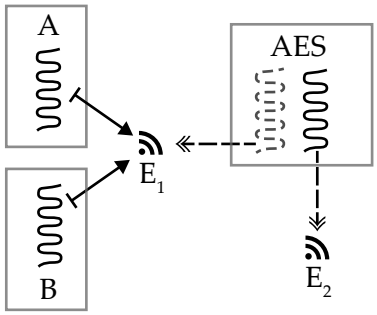

Figure 8.6: Architecture of the AES case study. Client *A* and *B* make IPC requests over endpoint E_1 of passive *AES* which has an active timeout fault handling thread waiting for fault IPC messages on endpoint E_2.

The server itself runs the AES-256 algorithm with a block size of 16 bytes. The server alternates between two buffers using an atomic swap, of which one always contains consistent state, the other is dirty during processing. When a timeout fault occurs only the dirty buffer is lost due to inconsistency.

Both clients request 4 MiB of data to be encrypted, 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.
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.

Platform	*Operation*	*Cache*	*Min*	*Max*	*Mean*	σ
SABRE	Rollback	hot	10.4	12.3	11.3	0.5
		cold	18.9	24.4	23.2	1.3
	Emergency	hot	4.1	5.6	4.6	0.3
		cold	13.0	14.5	13.6	0.4
	Extend	hot	0.8	3.3	1.9	0.4
		cold	6.8	7.6	7.0	0.2
	Kill	hot	9.7	11.6	10.4	0.4
		cold	21.2	22.6	21.9	0.3
X64	Rollback	hot	1.48	3.13	2.11	0.40
		cold	4.25	5.01	4.69	0.15
	Emergency	hot	1.00	1.93	1.32	0.17
		cold	2.29	2.83	2.42	0.10
	Extend	hot	0.16	0.88	0.33	0.16
		cold	0.88	1.32	1.03	0.10
	Kill	hot	1.36	1.45	1.39	0.02
		cold	4.24	5.17	4.50	0.14

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

8.3.4 Results

We measure the pure handling overhead in each case, from when the timeout handler wakes up to when it blocks again. Given the small amount of rollback state, this measures the 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, 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' budget between 0

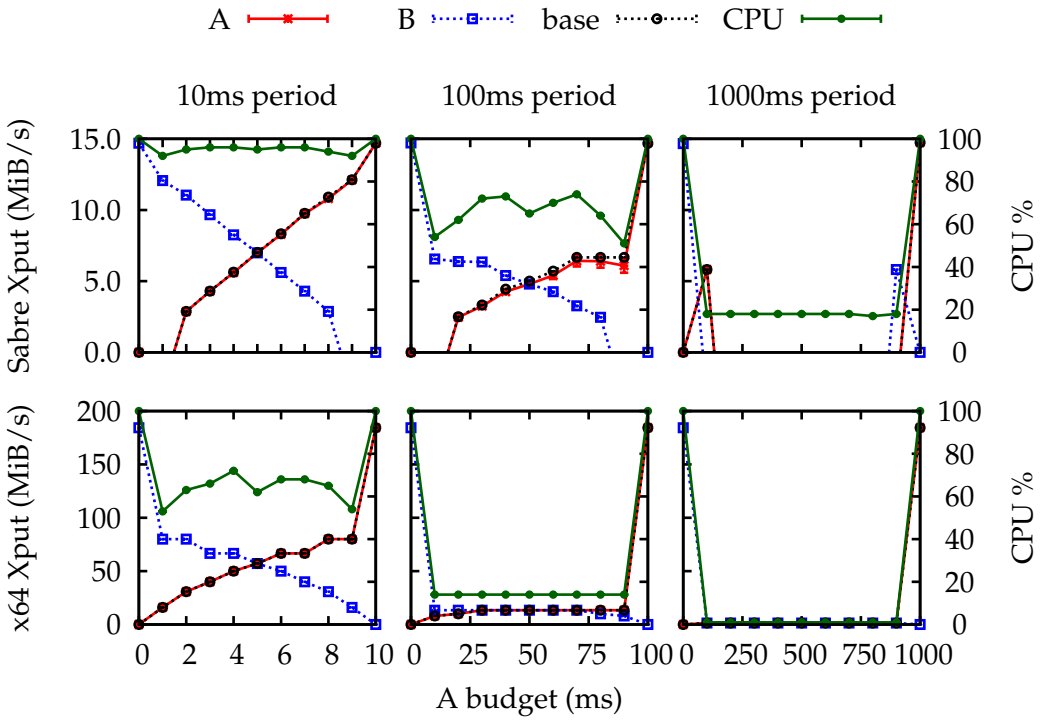

Figure 8.7: Throughput for clients A and B of a passive AES server processing 10 requests of 4 MiB of data with limited budgets on the X64 (top row) and SABRE (bottom row) platforms. The two clients' budgets add up to the period, which is varied between graphs (10, 100, 1000 ms). Clients sleep when they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

and the period. The extreme ends are special, as one of the clients has a full budget and keeps invoking the server without ever getting rolled back, thus monopolising the processor. In all other cases, each client processes at most 4 MiB of data per period, and either succeeds (if the budget is sufficient) or is rolled back after processing less than 4 MiB.

The results show that in the CPU-limited cases (left graphs) we have the expected near perfect proportionality between throughput and budget (with slight wiggles due to the rollbacks), showing isolation between clients. In the cases where there is headspace (centre of the right graphs), both clients achieve their desired throughput.

8.3.6 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.

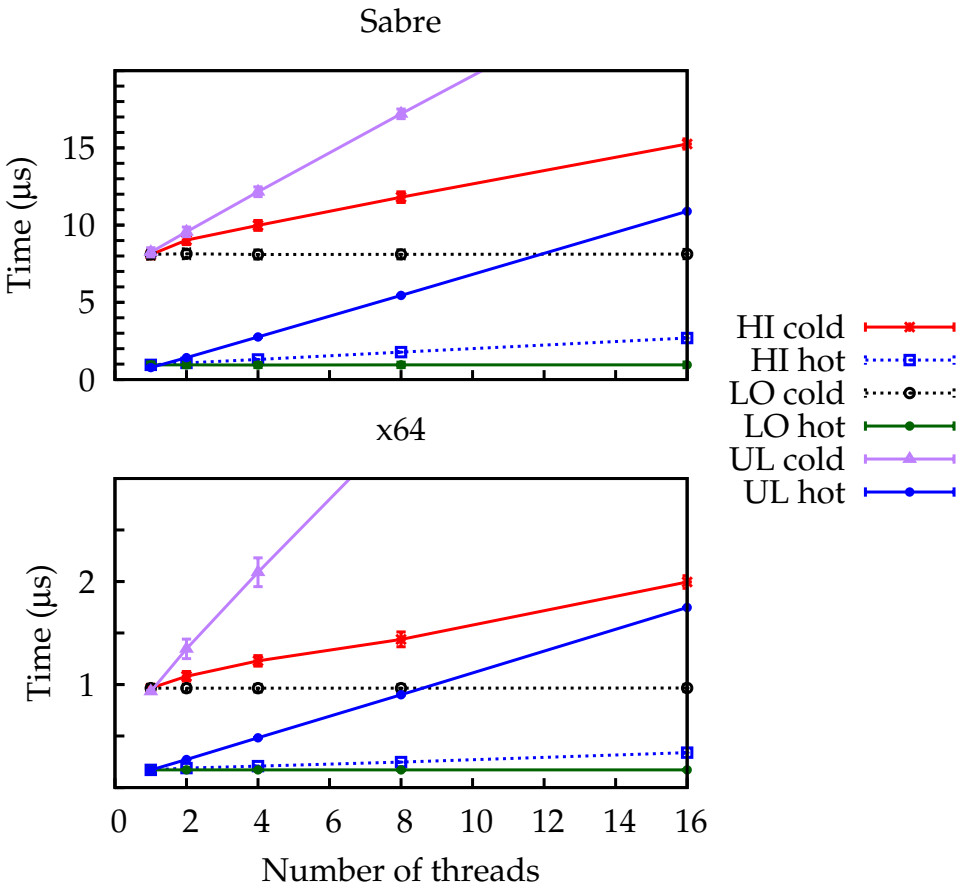

Figure 8.8: Cost of switching the priority of n threads, as well as changing from LOW to HIGH criticality level while increasing the number of HIGH and LOW threads, on SABRE and X64. Each data point is the average of 100 runs, with very small standard deviations.

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;

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.

8.5 Practicality

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,

Application	T	L	L_S	C	U	j	m
SUSAN	190	2	0	25	0.14	111	0
Image recognition		2	1	51	0.15	111	0
		2	2	127	0.54	111	0
JPEG	100	1	0	15	0.15	200	0
JPEG encode/decode.		1	1	41	0.41	200	0
		1	2	41	0.41	148	88
MAD	112	0	0	28	0.25	179	0
MP3 player		0	1	28	0.25	178	46
		0	2	28	0.25	7	7

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, 91 and 100% CPU utilisation for each stage respectively.

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., 2007], 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

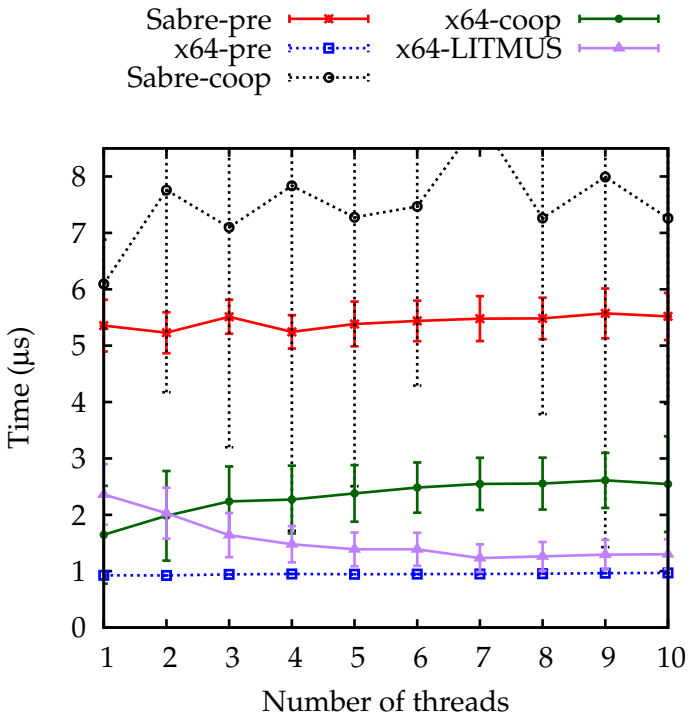

Figure 8.9: Execution time of seL4 user-mode EDF scheduler compared to kernel scheduler in x64 LITMUS^{RT}.

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.

8.6 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.

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.

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

9.2 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.

9.2.1 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.

9.2.2 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.

9.3 Timeline

I have completed 3 years of my PhD project, with 1 year remaining. I expect to safely complete in by the end of 2017. Below is a timeline to completion.

March	Get microbenchmark suite running on latest kernel API, finish write up of all microbenchmarks to date.
April	Design network case study demonstrating temporal isolation. Write up.
May	Finish implementatio and model write up. Start quadcopter/SMACCM case study.
June	Finish quadcopter/SMACCM case study, write up.
July	Work on multicore resource sharing policies.
August	Work on papers and phd.
September	Work on papers and phd.
October	Submit paper on microkernel design to Eurosys, RT multicore resource sharing paper to RTAS.
November	Finalise writing.
December	Send out phd for feedback, submit once collected and integrated.

9.3.1 Community Engagement

I participated in the Symposium on Operating Systems Principles (SOSP) 2013 Diversity Workshop presenting a pitch of my PhD topic centered around a poster. I presented at the Trustworthy Systems Summer School in 2017 and 2015.

9.3.2 Publications

In a collaboration with Manohar Vanga, Felipe Cerqueira and Björn Brandenburg from the Max Planck Institute for Software Systems (MPISS), my supervisor Gernot Heiser and I submitted a paper to the RTSS, during the first year of my PhD. This paper was rejected due to lack of theoretical novelty, which we put down to not being a good fit for that particular conference.

This paper was rejected again, from Real-Time and Embedded Technology and Applications Symposium (RTAS), so we ultimately decided that it was not a good fit for the real time community. We plan a resubmission of this paper to a systems conference once one of the case studies is complete, as to suit the systems community a demonstration of the work in real world scenario is required.

I have submitted papers on the scheduling API to OSDI and RTAS, however these were rejected. I plan two submit two papers in the final year, one to Eurosys on capabilities to time, and one to RTAS demonstrating the API to build mixed-criticality IPC protocols on multicore.

9.3.3 Summary

Over the past year I have developed a design for a principled approach to time in seL4, involving integrating resource sharing into IPC. I have nearly finished a concrete implementation and the next 12 months will involve evaluation and case studies. Although there is a long road ahead, I believe I am on track to a timely completion.

9.4 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.

List of Figures

Acronyms

AAV autonomous aerial vehicle. 1, 3

AES advanced encryption standard. 92–94, 110

API application programming interface. 71

BWI bandwidth inheritance. 33, 36

CA certification authority. 27

CBS constant bandwidth server. 23, 24, 26, 30–32, 35, 36

CFS completely fair scheduler. 30

CPU central processing unit. 35, 91

DM deadline monotonic. 11

DS deferrable server. 36

EDF earliest deadline first. 11, 12, 15, 16, 21, 23, 24, 27, 30, 31, 34, 36, 48, 49, 51, 60, 100

FIFO first-in first-out. 16, 30, 39, 83

FP fixed priority. 11, 12, 16, 21, 27, 28, 31, 32, 36, 48

FPRM fixed-priority rate monotonic. 12, 31

FPU floating point unit. 81

HLP highest lockers protocol. 14, 15, 25, 30, 36, 47, 54, 60, 73

HRT hard real-time. 7, 8, 10, 12, 21–23, 26, 31, 45, 46, 49, 66, 76, 104

IPC inter-process communication. 32, 33, 37–43, 51, 54, 71, 78, 79, 83–87, 93, 96, 104, 106, 110

ISR interrupt service routine. 34

MCAR Mixed Criticality Architecture Requirements. 1

MCS mixed criticality system. 36, 83, 86

MMU memory management unit. 5, 21

MPCP multiprocessor priority ceiling protocol. 16

MPISS Max Planck Institute for Software Systems. 106

MSR model specific registers. 83

MSRP multiprocessor stack resource policy. 16

NCET nominal-case execution time. 27

NCP non-preemptive critical sections protocol. 14, 15

OCET overloaded-case execution time. 27

OS operating system. 1, 3, 4, 12, 16–18, 28–30, 34, 35, 37, 66

PCP priority ceiling protocol. 14–16, 25, 54

PIP priority inheritance protocol. 14, 15, 25, 30, 33, 36, 53, 60

POSIX portable operating system interface. 29–31, 48

QoS quality of service. 26, 27

RBED rate-based earliest deadline first. 23, 24, 26, 34–36

RM rate monotonic. 11, 12, 27

RPC remote procedure call. 51

RTA response time analysis. 50

RTAS Real-Time and Embedded Technology and Applications Symposium. 106

RTOS real-time operating system. 17, 18, 22, 31, 32

SCO scheduling context object. 63, 64, 83, 84

SMP symmetric multiprocessor. 5, 104

SOSP Symposium on Operating Systems Principles. 106

SRP stack resource policy. 15, 16

SRT soft real-time. 7, 8, 10, 21, 23, 24, 26, 30, 31, 34, 35, 45, 46, 50, 66, 76, 95, 104

SS sporadic server. 36

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

TCB thread control block. 38–41, 63, 64, 94, 95

TSC timestamp counter. 83, 84

UAV unmanned aerial vehicle. 27

UDP User Datagram Protocol. 91, 92

VM virtual machine. 34, 91, 92

WCET worst case execution time. 4, 7, 8, 10, 17, 18, 21–24, 26, 27, 31–34, 37, 41, 49, 52, 53, 55, 56, 58, 65, 66, 105

YCSB Yahoo! Cloud Serving Benchmark. 87, 90, 110

ZS zero slack. 26, 27, 31

Bibliography

Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, 2004. URL http://portal.acm.org/citation.cfm?id=990299&dl=ACM&coll=ACM&CFID=50173394&CFTOKEN=46138865.

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*, 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. Available at http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/, April 2009.

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, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 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.

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 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)*, Pisa, IT, July 2007.

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*, Cancun, MX, December 2003.

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. Mixed criticality systems – a review. Technical report, Department of Computer Science, University of York, 2014. http://www-users.cs.york.ac.uk/burns/review.pdf.

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. URL http://portal.acm.org/citation.cfm?id=1035388&dl=ACM&coll=ACM&CFID=50173394&CFTOKEN=46138865.

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, 2007.

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.

QNX Software Systems Co. *QNX Neutrino System Architecture [6.5.0]*, 6.5.0 edition, 2010.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, Indianapolis, IN, US, June 2010.

Johnathan Corbet. Deadline scheduling for linux. http://lkml.net/Articles/356576, October 2009.

Jonathan Corbet. `SCHED_FIFO` and realtime throttling. http:lkml.net/Articles/296419, September 2008.

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.

Deionisio 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.

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.

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. http://www.ddci.com/products_deos.php, 2012.

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*, 2010.

Dario Faggioli. POSIX_SCHED_SPORADIC implementation for tasks and groups. http://lkml.org/lkml/2008/8/11/161, August 2008.

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.

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*, December 2001.

Michael Gonzalez Harbour. Real-time POSIX: An overview. http://www.ctr.unican.es/publications/mgh-1993a.pdf, 1993.

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.

AndrÃľ 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.

Intel 64 & IA-32 ASDM. *Intel 64 and IA-32 Architectures Software DeveloperâĂŹs Manual Volume 3(3A, 3B, 3C & 3D): System Programming Guide*. Intel Corporation, September 2016.

Michael B. Jones. What really happened on Mars?, 1997. URL research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html. [Online; accessed 28-July-2014].

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.

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.

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.

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.

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.

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.

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, 1993.

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*, May 1994.

NetBSD. NetBSD. https://www.netbsd.org.

Shuichi Oikawa and Ragunathan Rajkumar. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symposium*, 1998.

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)*, Brussels, BE, July 2010.

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.

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*, 2010.

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.

PikeOS. PikeOS. http://www.sysgo.com/services-solutions/safety-security-certification, 2012.

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.

Redis. Redis. http://redis.io.

Wind River. *VxWorks Kernel Programmers Guide*, 6.7 edition, 2008.

RTEMS. RTEMS. http://www.rtems.org, 2012.

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.

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. URL http://www.eros-os.org/papers/sosp-eros-preprint.ps.

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.

Mark J. Stanovic, Theodore P. Baker, and An-I Wang. Experience with sporadic server scheduling in linux: Theory vs. practice. In *Proceedings of the 13th 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.

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)*, Brussels, BE, July 2010.

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.

Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, October 1990.

Data61 Trustworthy Systems Team. *seL4 Reference Manual, Version 7.0.0*, September 2017.

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*, 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.

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.

Fastpath Performance Analysis

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 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	21474836

Table 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 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 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 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 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 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 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 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 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 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 12: haswell Reply recv fastpath