



School of Computer Science & Engineering

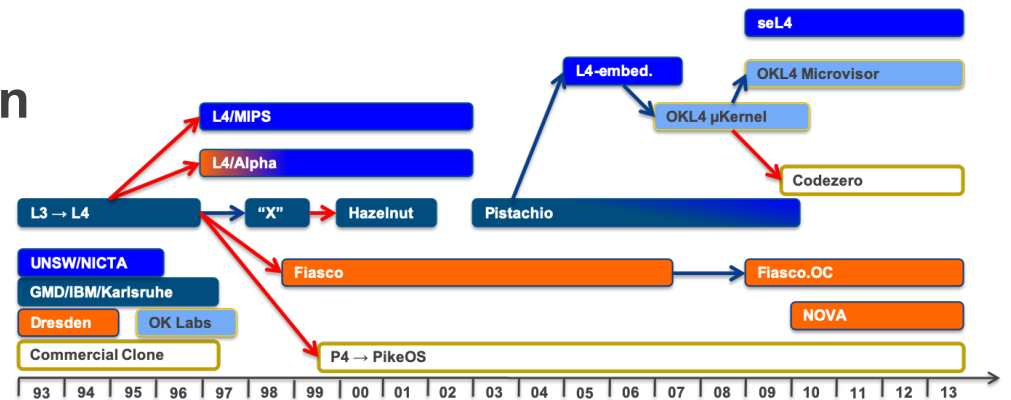
COMP9242 Advanced Operating Systems

2024 T3 Week 05 Part 1

Microkernel Design & Implementation

The 25-year quest for the right API

@GernotHeiser



Copyright Notice

These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) License

- You are free:
 - to share—to copy, distribute and transmit the work
 - to remix—to adapt the work
- under the following conditions:
 - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

“Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at
<http://creativecommons.org/licenses/by/4.0/legalcode>

L4 Microkernels – Deployed by the Billions



Today's Lecture

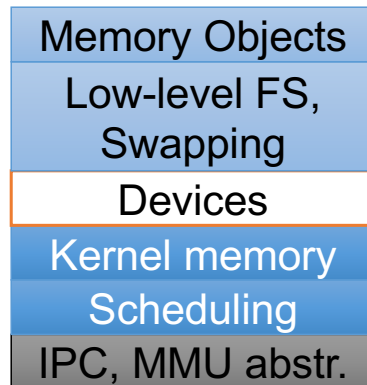
- Towards real microkernels: The history of L4 microkernels
- Implementation highlights
- Virtualisation: Microkernel as hypervisor
- Lessons and principles

L4: The Quest for a Real Microkernel

Microkernel Evolution

First generation

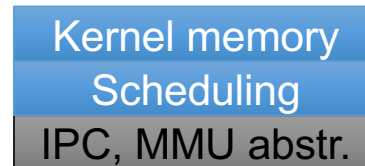
Mach ['87], Chorus



180 syscalls, 100 kSLOC
100 μ s IPC

Second generation

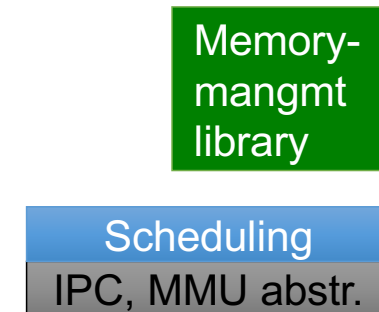
L4 ['95], PikeOS,
INTEGRITY, Minix 3,
QNX



~ 7 syscalls, ~ 10 kSLOC
~ 1 μ s IPC (L4)
~ 10 μ s IPC (others)

Third generation

seL4 ['09]



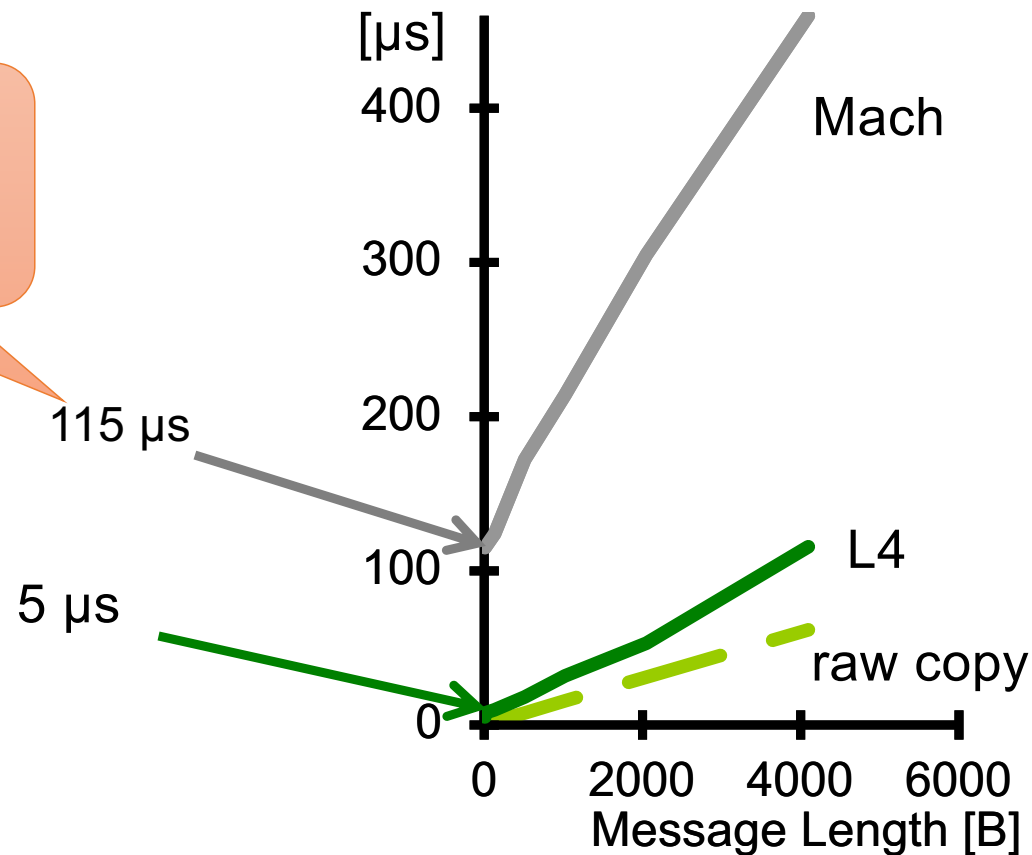
~3 syscalls, ~10 kSLOC
0.1–0.3 μ s IPC (faster HW)

Capabilities

Design for isolation

1993 “Microkernel”: IPC Performance

Culprit:
Cache footprint
[Liedtke'95]



i486 @
50 MHz

Remember: Microkernel Minimality Principle

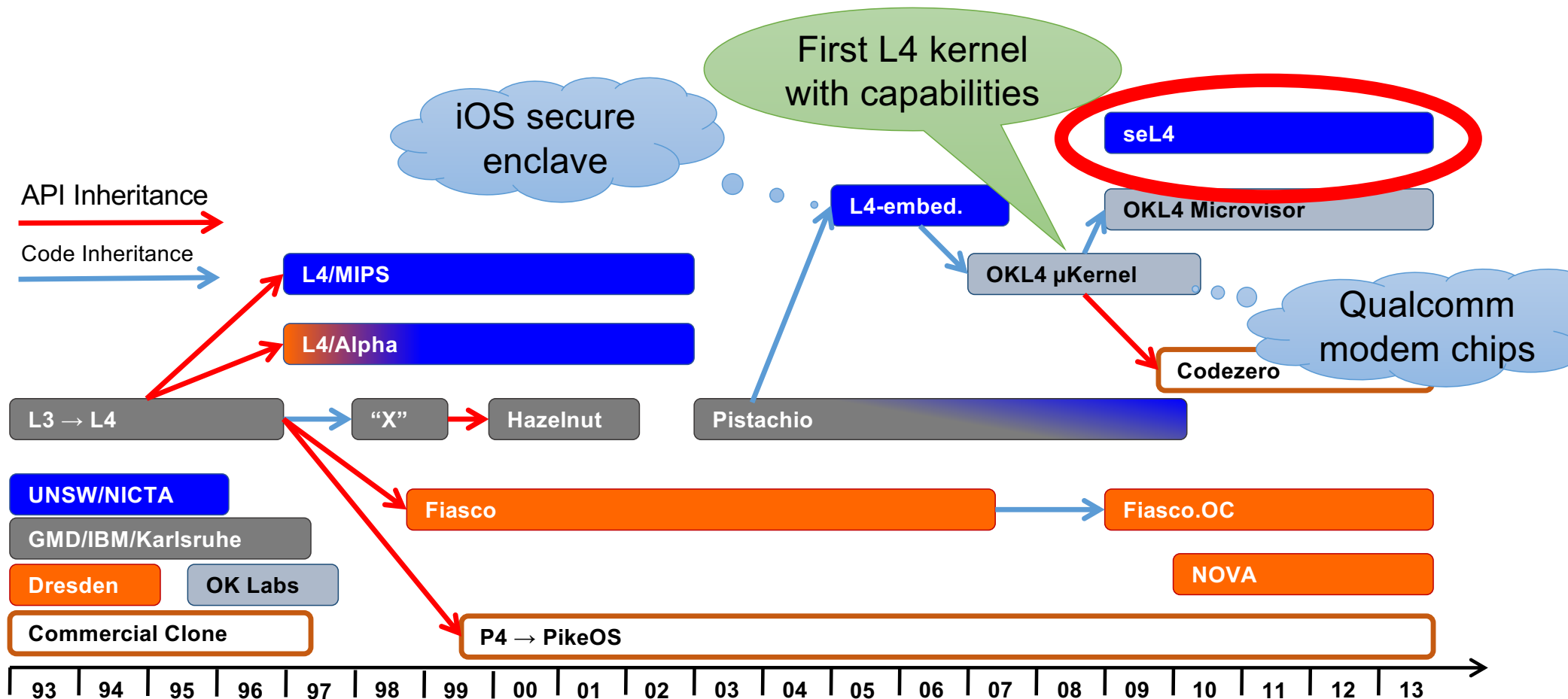


A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality. [Liedtke SOSP'95]

- Small *trusted computing base*
 - Easier to get right
 - Small attack surface
- Challenges:
 - API design: **generality** despite small code base
 - Kernel design and implementation for high performance

Needs policy-freedom!

L4: 25 Years High Performance Microkernels



L4 1-Way IPC Performance Over the Years

Name	Year	Processor	MHz	Cycles	μ s
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	MIPS R4700	100	86	0.86
L4/Alpha	1997	Alpha 21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium	1,500	36	0.02
OKL4	2007	Arm XScale 255	400	151	0.64
NOVA	2010	x86 i7 Bloomfield (32-bit)	2,660	288	0.11
seL4	2013	ARM11	532	188	0.35
seL4	2018	x86 i7 Haswell (64-bit)	3,400	442	0.13
seL4	2018	Arm Cortex A9	1,000	303	0.30
seL4	2020	RISC-V HiFive (64-bit, no ASID)	1,500	500	0.33

Independent Comparison [Mi et al., 2019]

Cost	seL4	Fiasco.OC	Zircon
IPC RT latency (cycles)	986	2717	8157
Mand. HW cost (cycles)	790	790	790
Abs. overhead (cycles)	196	1972	7367
Rel. overhead (%)	25	240	930

Hardware
cost dominates

SW overheads
dominate

Round-trip, cross-address-space IPC on x64 (Intel Skylake)

Operation	1-way	RT
SYSCALL	82	164
SWAPGS	2×26	104
Switch PT	186	372
SYSRET	75	150
Total	395	790

Source: Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, Haibo Chen: "SkyBridge: Fast and Secure Inter-Process Communication for Microkernels", EuroSys, April 2019

Minimality: Source Lines of Code (SLOC)

Name	Architecture	C	C++	asm	total
Original	i486	0 k	0 k	6.4 k	6.4 k
L4/Alpha	Alpha	0 k	0 k	14.2 k	14.2 k
L4/MIPS	MIPS64	6.0 k	0 k	4.5 k	10.5 k
Hazelnut	x86	10.0 k	0 k	0.8 k	10.8 k
Pistachio	x86	0 k	22.4 k	1.4 k	23.0 k
L4-embedded	ARMv5	7.6 k	0 k	1.4 k	9.0 k
OKL4 3.0	ARMv6	15.0 k	0 k	0.0 k	15.0 k
Fiasco.OC	x86	0 k	36.2 k	1.1 k	37.6 k
seL4	ARMv6	9.7 k	0 k	0.5 k	10.2 k

Issues With 2G Microkernels

- L4 solved microkernel performance [Härtig et al, SOSPP'97]
- Left a number of issues unsolved
- Problem: ad-hoc approach to security and resource management

- Global thread name space $\Rightarrow$ covert channels [Shapiro'03]
- Threads as IPC targets $\Rightarrow$ insufficient encapsulation
- No delegation of authority $\Rightarrow$ impacts flexibility, performance

Caps & endpoints

- Single kernel memory pool $\Rightarrow$ DoS attacks

seL4 memory management model

- Unprincipled management of time

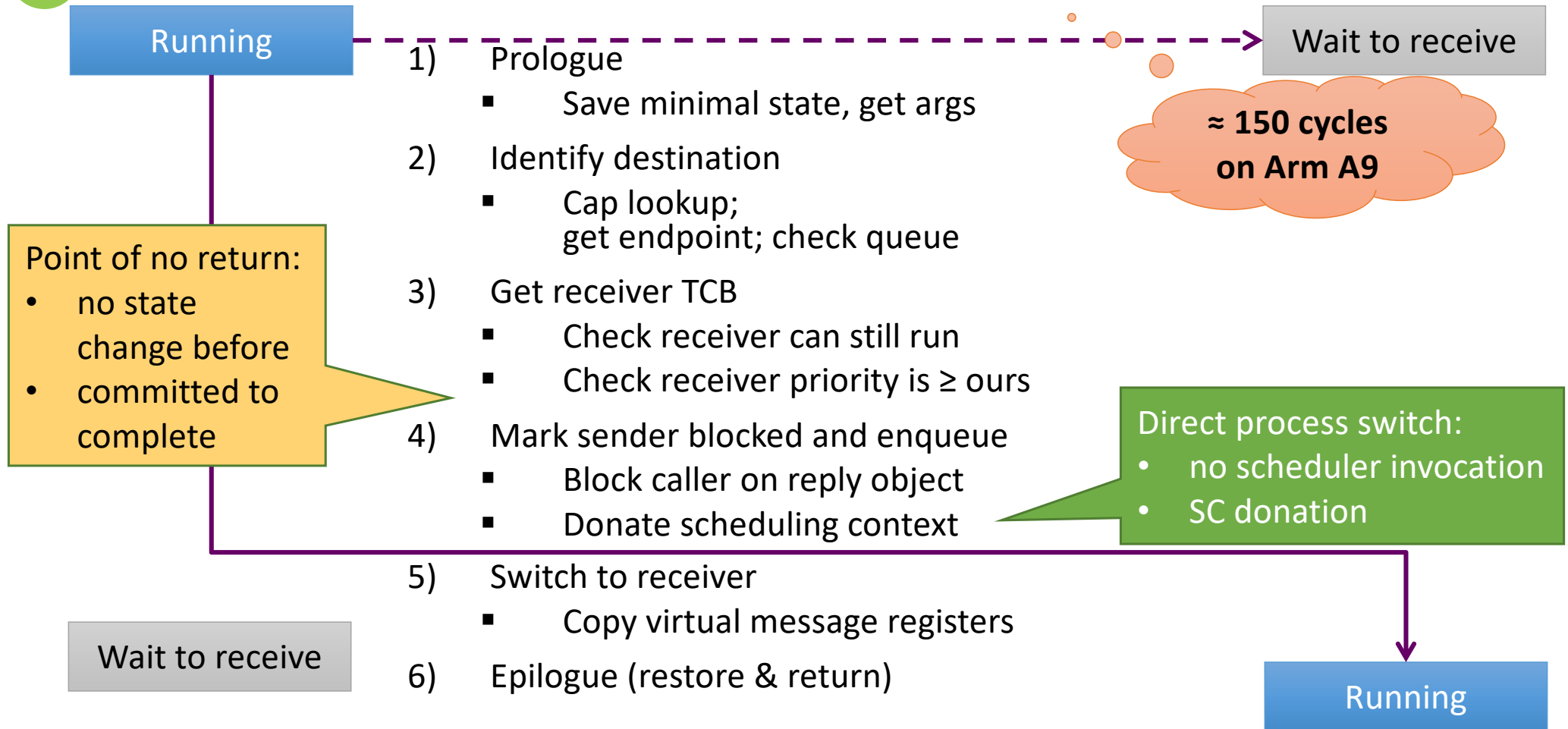
seL4 scheduling contexts (MCS)

Implementation Highlights





IPC Fastpath: Send Phase of Call



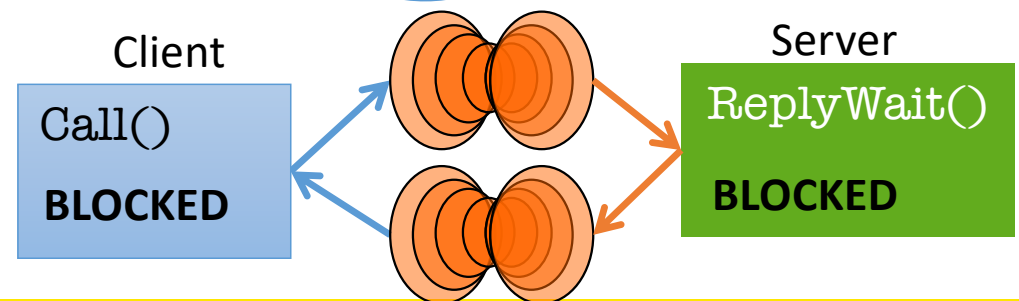
L4 Scheduler Optimisation: Lazy Scheduling

```
thread_t schedule() {  
    foreach (prio in priorities) {  
        foreach (thread in runQueue[prio]) {  
            if (isRunnable(thread))  
                return thread;  
            else  
                schedDequeue(thread);  
        }  
    }  
    return idleThread;  
}
```

Problem: Unbounded
scheduler execution time!

- Frequent blocking/unblocking in IPC-based systems
- Many ready-queue manipulations

Idea: leave blocked
threads in ready
queue, scheduler
cleans up



se14

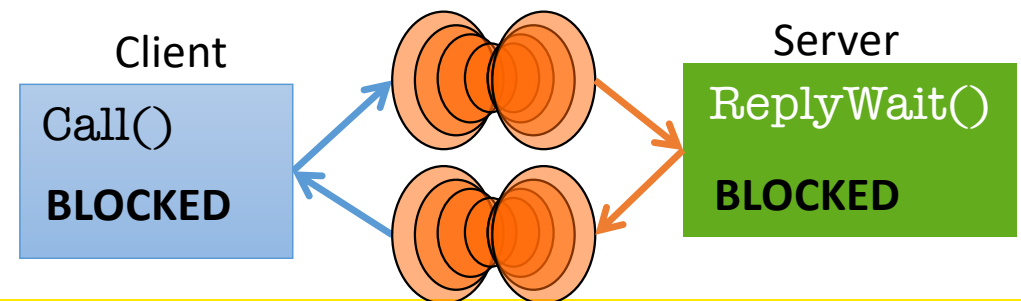
Scheduler: Benno Scheduling

```
thread_t schedule() {
    foreach (prio in priorities) {
        foreach (thread in runQueue[prio]) {
            if (thread=head(runQueue[prio]))
                return thread;
        else
            schedDequeue(thread);
    }
    return idleThread;
}
```

Only current thread
needs fixing up at
preemption time!

Idea: Lazy on
unblocking instead
on *blocking*

- Frequent blocking/unblocking in IPC-based systems
- Many ready-queue manipulations



Scheduler Optimisation: Direct Process Switch

- Sender was running $\Rightarrow$ had highest prio
- If receiver prio $\geq$ sender prio $\Rightarrow$ run receiver

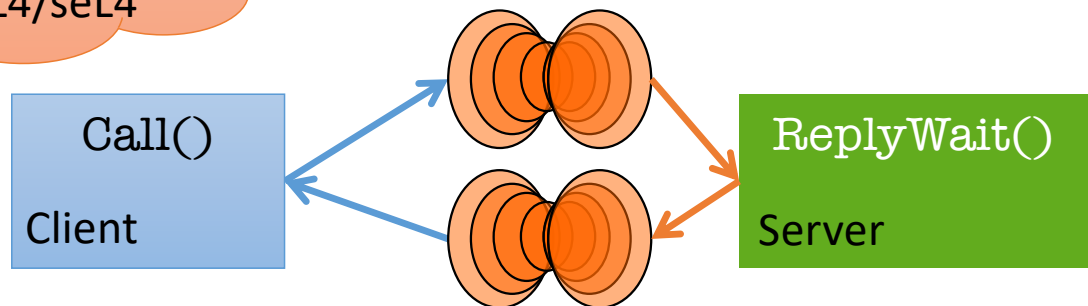
Idea: Don't invoke scheduler if you know who'll be chosen

Unprincipled time-slice donation in earlier L4/seL4

Note:

- Only works if server can run on client's time slice
- MCS passive server with scheduling-context donation
- Donate on `Call()`
- Return on `ReplyWait()`

- Frequent context switches in IPC-based systems
- Many scheduler invocations





Fastpath Coding Tricks

```
slow =    cap_get_capType(en_c) != cap_endpoint_cap ||  
          !cap_endpoint_cap_get_capCanSend(en_c);  
if (slow) enter_slow_path();
```

Common case: 0

Common case: 1

- Reduces branch-prediction footprint
- Avoids mispredicts, stalls & flushes
- Uses ARM instruction predication (pre-v8)
- Slightly increases slow-path latency (very slightly)
 - insignificant compared to basic slow-path cost

How About Real-Time Support?

- Kernel runs with interrupts disabled
 - No concurrency control $\Rightarrow$ simpler kernel
 - Easier reasoning about correctness
 - Better average-case performance

How about long-running system calls?

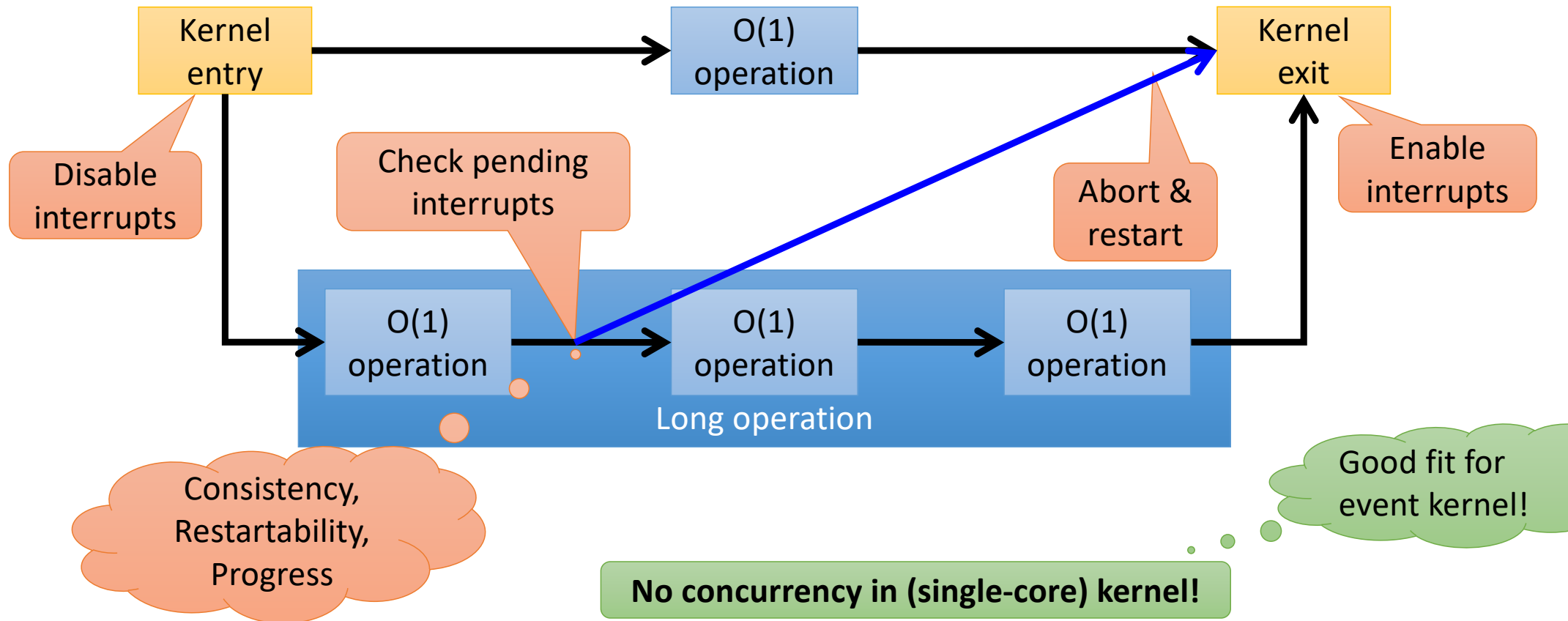
Most protected-mode RTOSes are mostly/fully preemptible

**Lots of
concurrency in
kernel!**



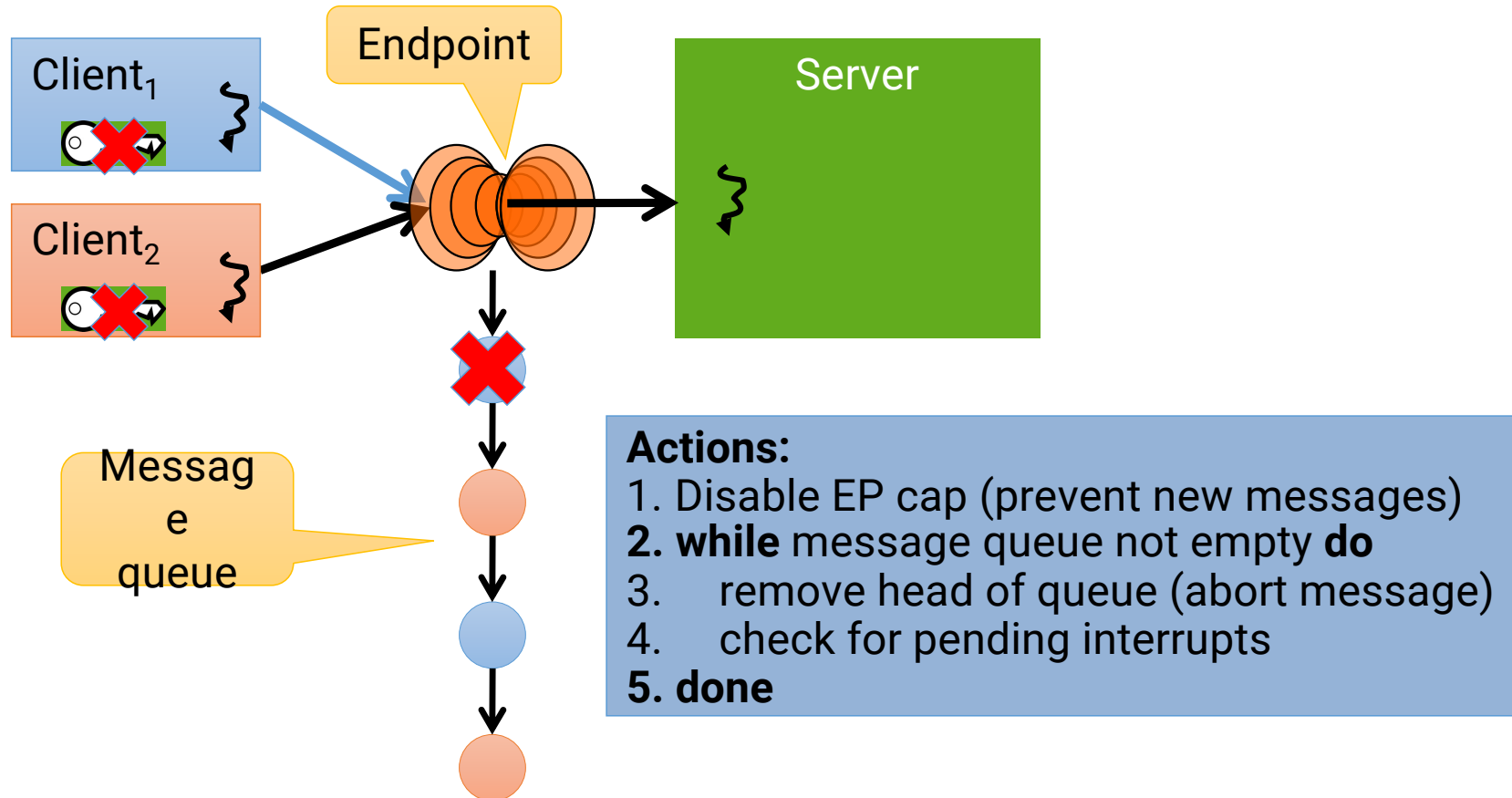


Incremental Consistency Paradigm



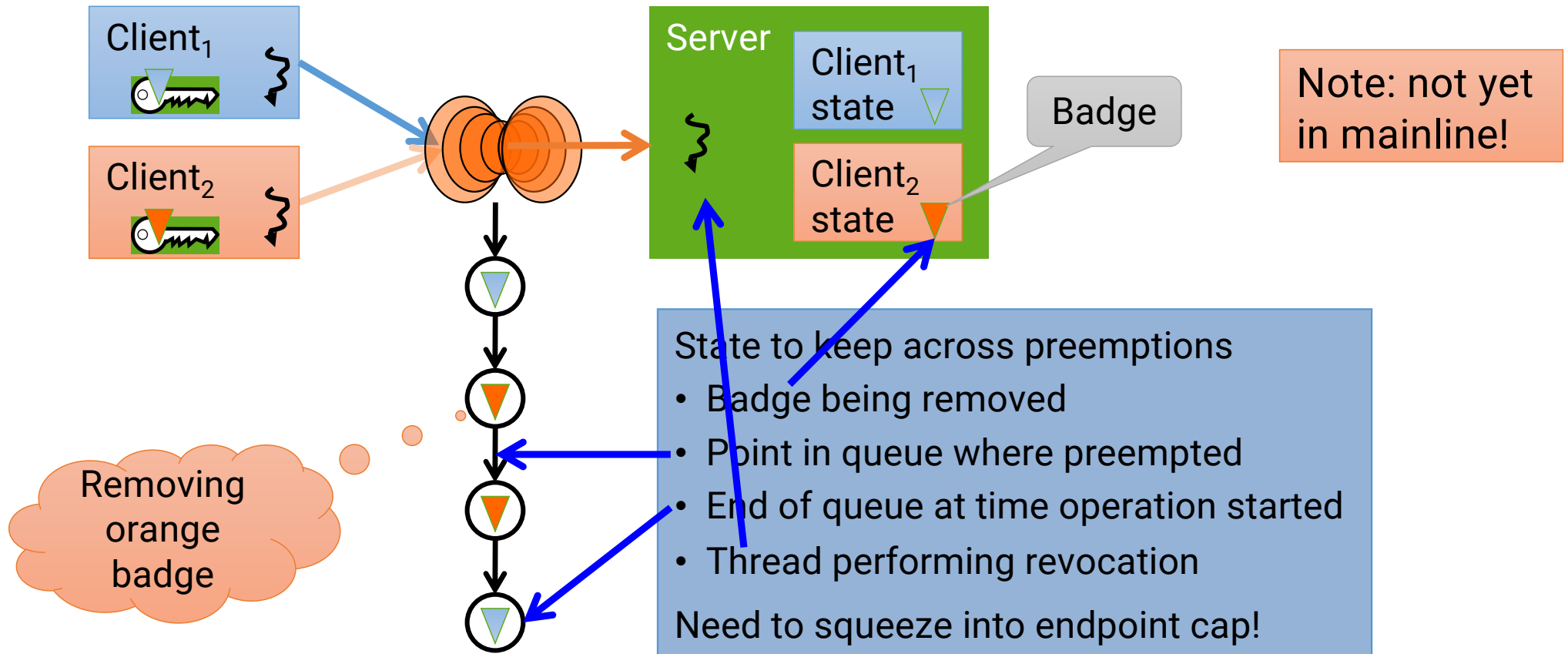


Example: Destroying IPC Endpoint



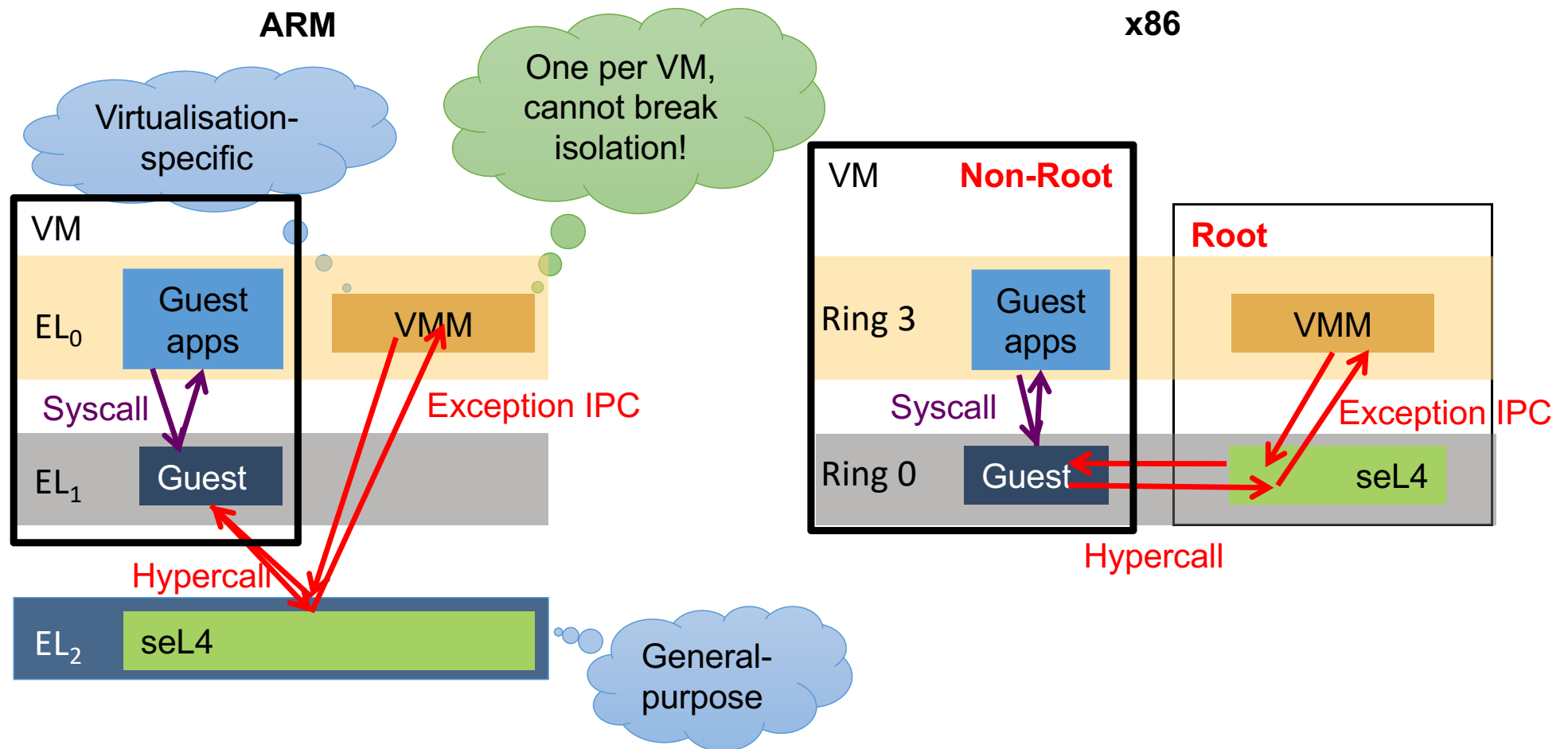


Difficult Example: Revoking Badge



Virtualisation: Microkernel as a Hypervisor

Microkernel as Hypervisor (NOVA, seL4)



Hypervisors vs Microkernels

- Both contain all code executing at highest privilege level
 - Although hypervisor may contain user-mode code as well
 - privileged part usually called “hypervisor”
 - user-mode part often called “VMM”
- Both need to abstract hardware resources
 - Hypervisor: abstraction closely models hardware
 - Microkernel: abstraction designed to support wide range of systems

Difference to traditional terminology!

To abstract:

- CPU
- Memory
- Communication
- I/O

What /s the Difference?

Resource	Hypervisor	Microkernel
Memory	Virtual MMU (vMMU)	Address space
CPU	Virtual CPU (vCPU)	Thread or scheduler activation
Communication	Virtual NIC, with device driver and network stack	High-performance message-passing IPC
I/O	• Simplified virtual device • Driver in hypervisor • Virtual IRQ (vIRQ)	• IPC interface to user-mode driver • Interrupt IPC

Just page tables in disguise

Just kernel-scheduled activities

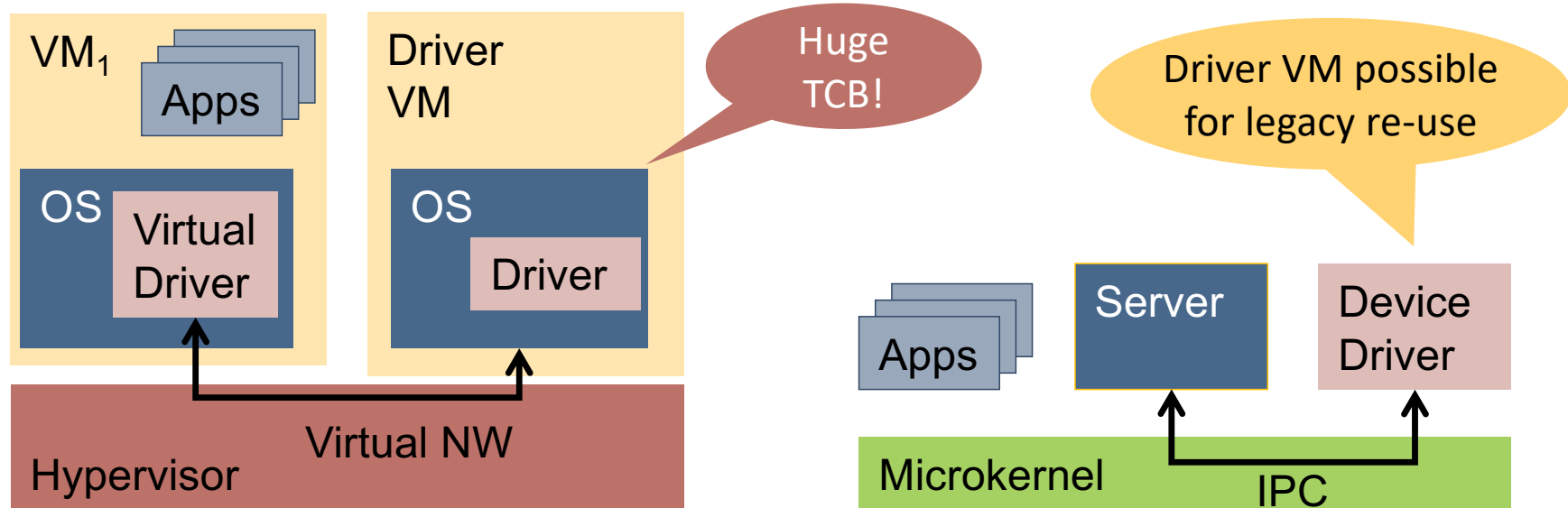
Modelled on HW, Re-uses SW

Minimal overhead, Custom API

Real Difference?

- Similar abstractions
- Optimised for different use cases

Closer Look at Communication and I/O

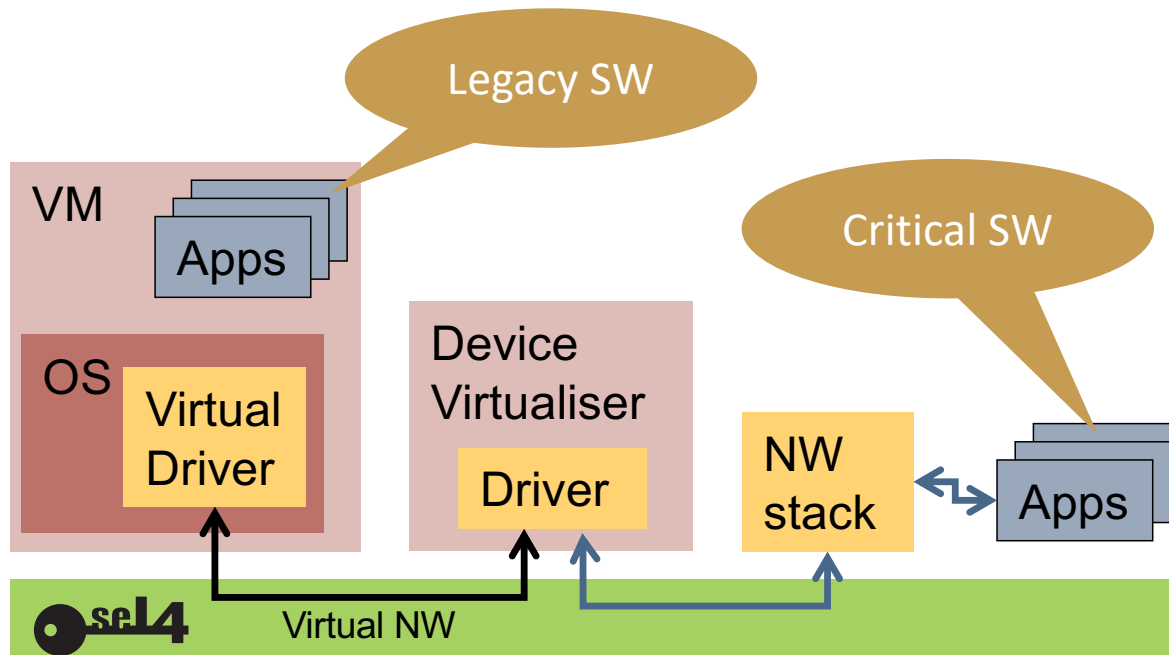


Communication is critical for I/O

- Highly-optimised microkernel IPC
- Inter-VM communication is frequently a bottleneck in hypervisors



Integration: VMs and Native



- Typical configuration in embedded systems
- Supports “incremental cyber retrofit”

Lessons & Principles

Reflecting on Lessons of 2nd Generation

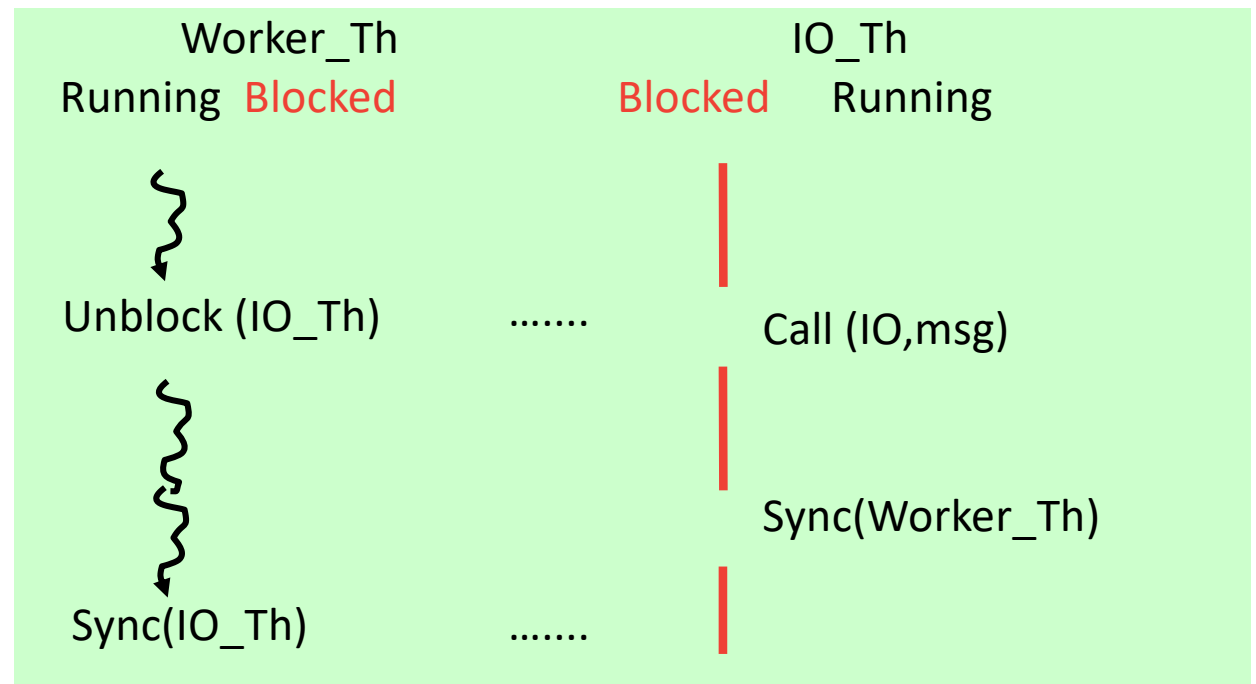
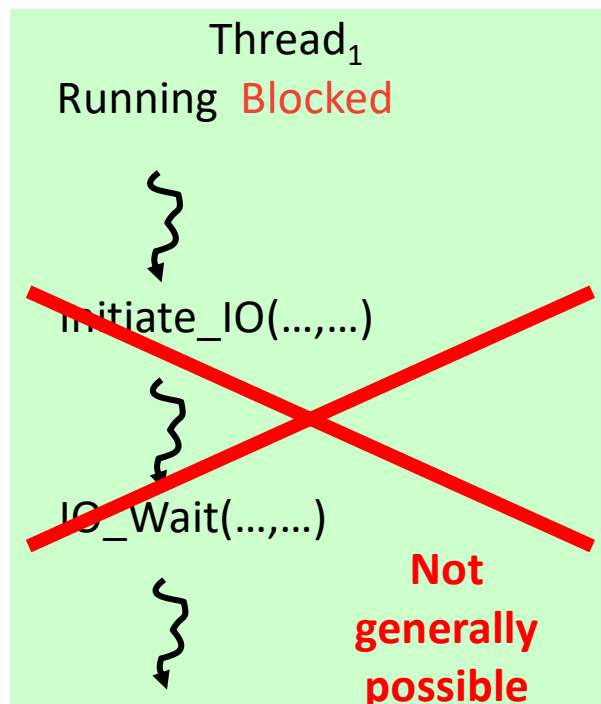
Original L4 design had two major shortcomings:

1. Insufficient/impractical resource control
 - Poor/non-existent control over kernel memory use
 - Inflexible & costly process hierarchies (policy!)
 - Arbitrary limits on number of address spaces and threads (policy!)
 - Poor information hiding (IPC addressed to threads)
 - Insufficient mechanisms for authority delegation
2. Over-optimised IPC abstraction, mangles:
 - Communication, incl bulk data copy
 - Synchronisation
 - Timed wait
 - Memory management – sending mappings
 - Scheduling – time-slice donation

Synchronous IPC issues

- Sync IPC forces multi-threaded code or select()!
- Also poor choice for multi-core

seL4: Notification binding



L4 “Long” IPC

**seL4:
Removed**

- Not minimal
- Source of kernel complexity:
 - nested exceptions
 - concurrency in kernel
 - must upcall PF handlers during IPC
 - timeouts to prevent DOS attacks

Sender address space



Kernel copy

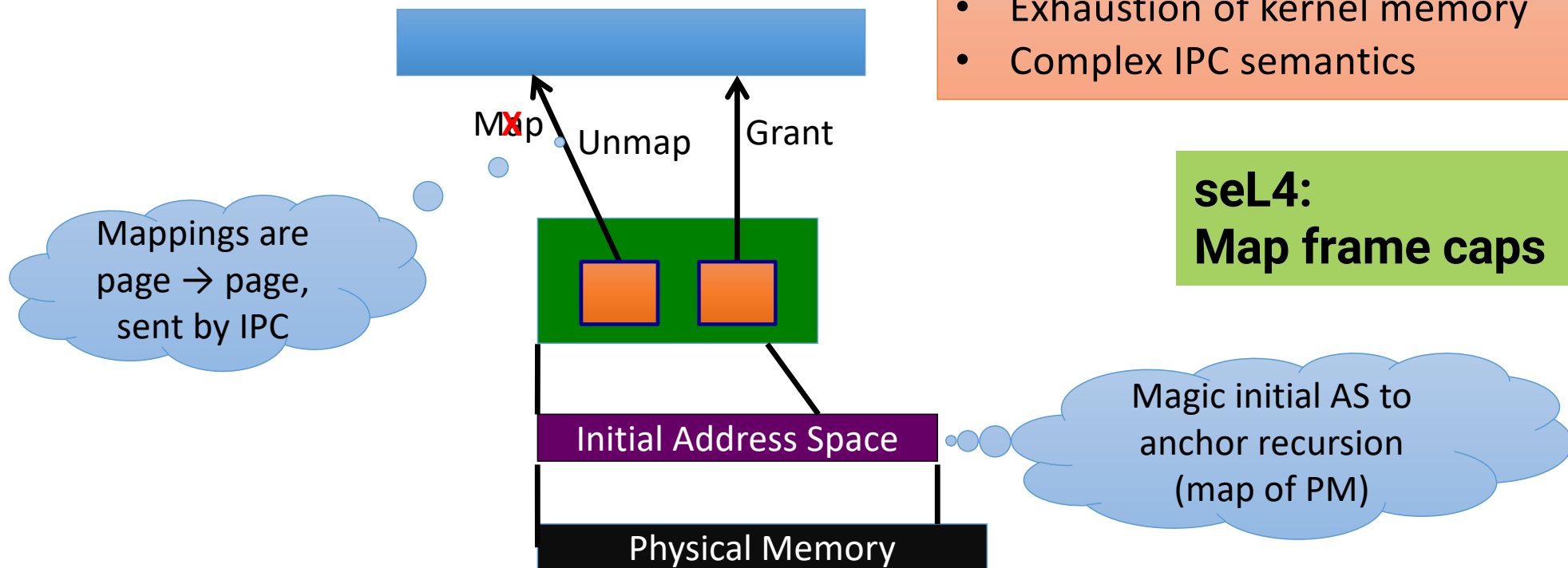
Receiver address space



Traditional L4: Recursive Address Spaces

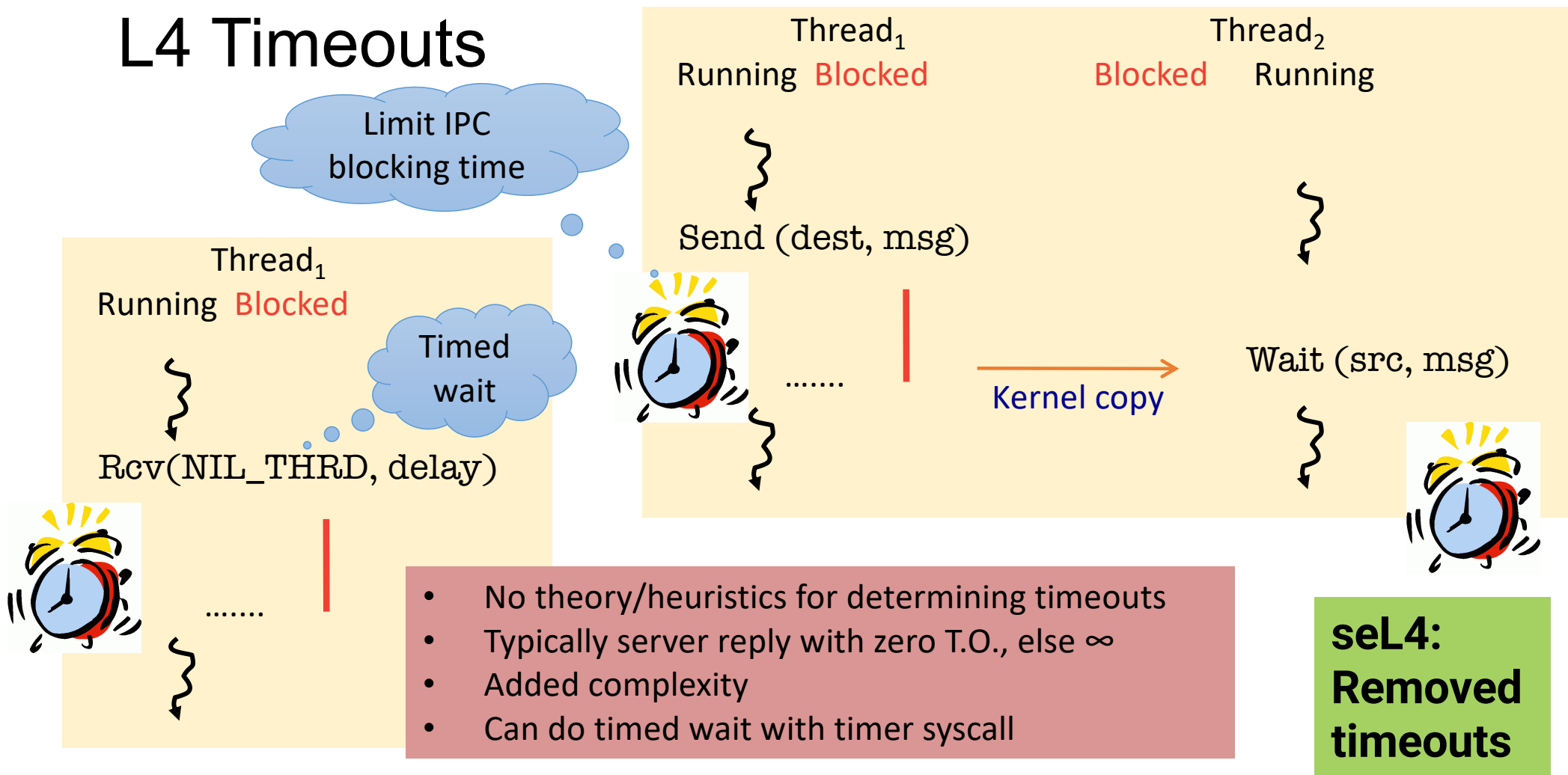
Issues:

- Complex mapping DB
- Exhaustion of kernel memory
- Complex IPC semantics



seL4:
Map frame caps

L4 Timeouts





Unresolved in Original seL4: Time

Simplistic fixed-priority scheduling

- Fixed by scheduling contexts

Priority violation through “lazy scheduling”

- Fixed by Benno scheduling in about 2015

Lack of temporal isolation in server invocations

- Fixed by SC donation to passive servers (MCS kernel)

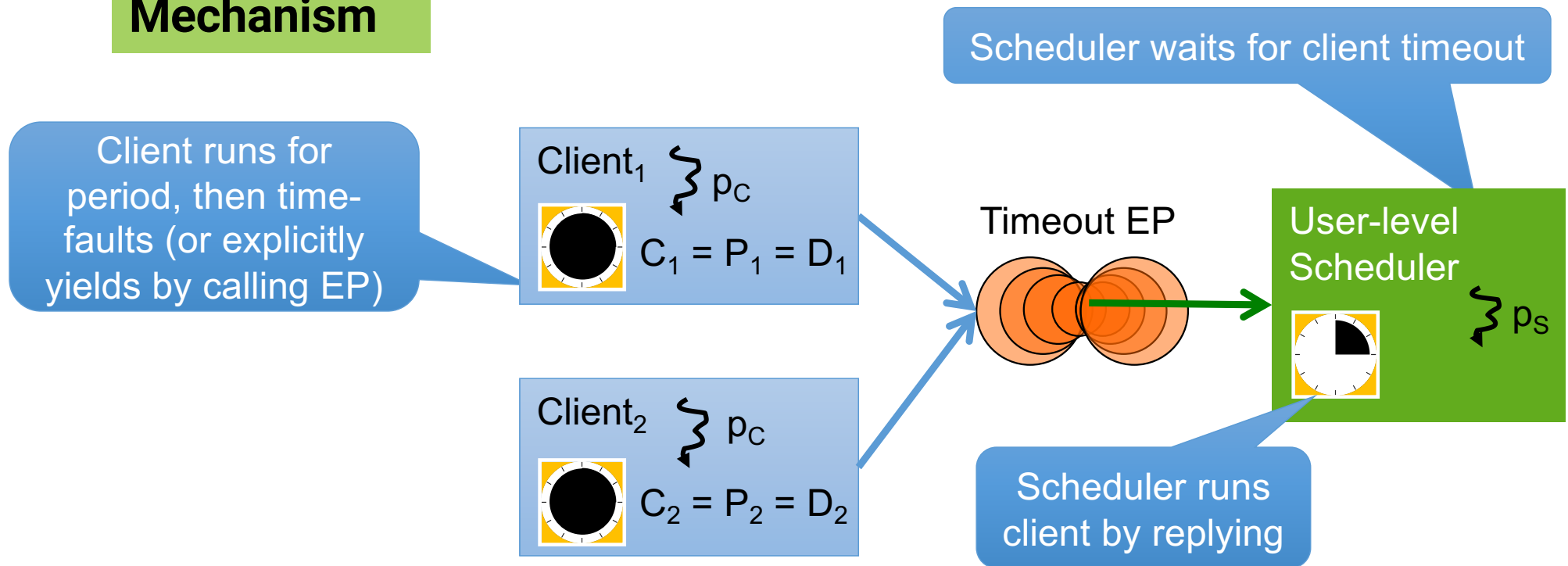
Complexity forced on servers through MCS budget timeouts

- Pending RFC for budget thresholds on endpoints

sel4 Isn't a Fixed-Prio Scheduler Policy?

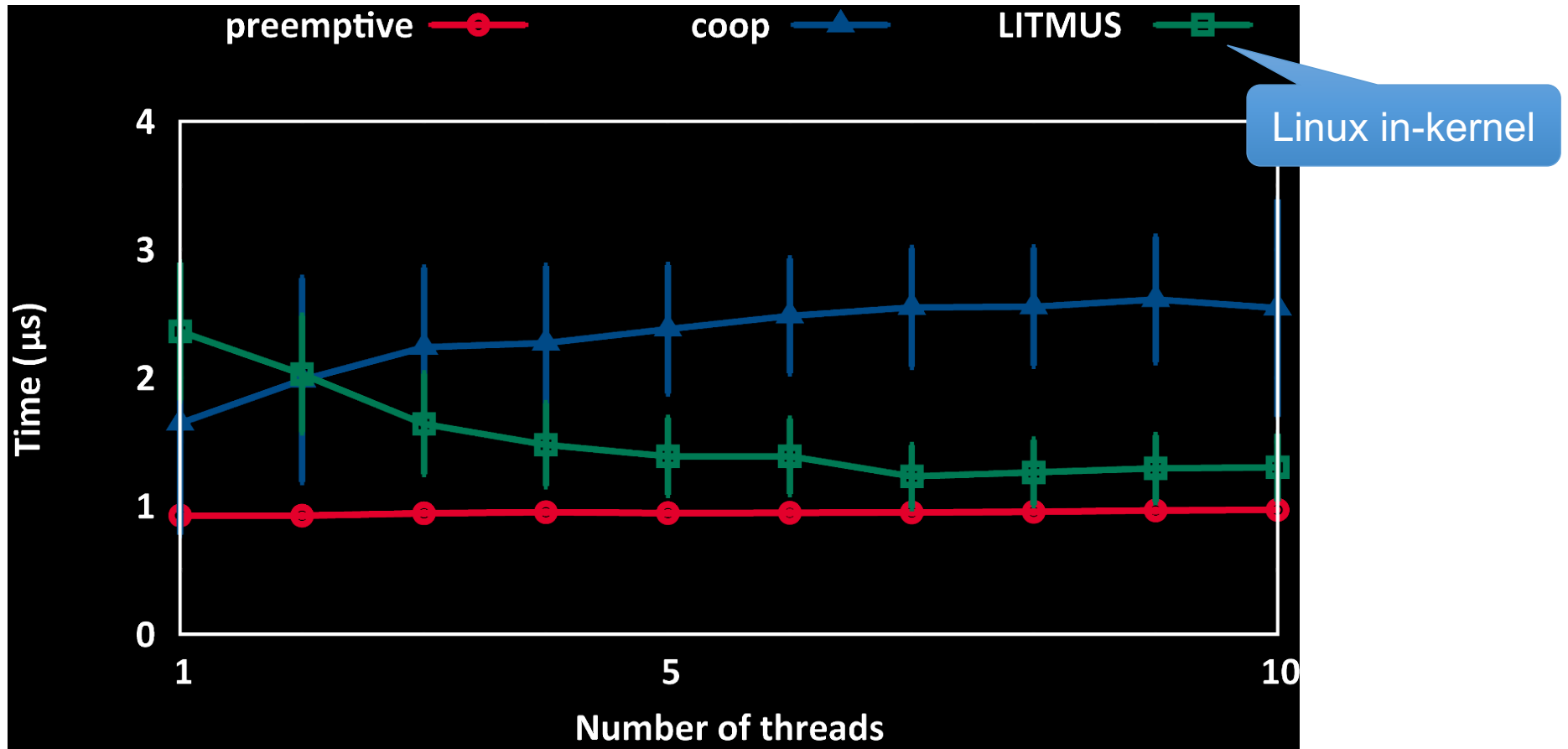
Prios + SCs = Mechanism

Implementing scheduling policy at user level





User-Level EDF Scheduler Performance



seL4 Design Principles

- Fully delegable access control via capabilities
- All resource management is subject to user-defined policies
 - Applies to kernel resources too!
- Performance on par with best-performing L4 kernels
 - Prerequisite for real-world deployment!
- Suitability for real-time use
 - Important for safety-critical systems
- Suitable for *formal verification*
 - Requires small size, avoid complex constructs

Largely in line with traditional L4 approach!

A Thirty-Year Dream!

Operating
Systems

R. Stockton Gaines
Editor

Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. Kemmerer, and
Gerald J. Popek
University of California, Los Angeles

Data Secure Unix, a kernel structured operating system, was constructed as part of an ongoing effort at UCLA to develop procedures by which operating systems can be produced and shown secure. Program verification methods were extensively applied as a constructive means of demonstrating security enforcement.

Here we report the specification and verification experience in producing a secure operating system. The work represents a significant attempt to verify a large-scale, production level software system, including all aspects from initial specification to verification of implemented code.

Key Words and Phrases: verification, security, operating systems, protection, programming methodology, ALPHARD, formal specifications, Unix, security kernel

CR Categories: 4.29, 4.35, 6.35

1. Introduction

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that piecemeal alterations were unlikely ever to succeed [20]. A more systematic method was required, presumably one that controlled the system's design and implementation. Then secure operation could be demonstrated in a stronger sense than an ingenuous claim that the last bug had been eliminated, particularly since production systems are rarely static, and errors easily introduced.

Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components of this task are: (1) developing system architectures that minimize the amount and complexity of software involved in both protection decisions and enforcement, by isolating them into *kernel* modules; and (2) applying extensive verification methods to that kernel software in order to prove that our *data security* criterion is met. This paper reports on the latter part, the verification experience. Those interested in architectural issues should see [23]. Related work includes the PSOS operating system project at SRI [25] which uses the hierarchical design methodology described by Robinson and Levitt in [26], and efforts to prove communications software at the University of Texas [31].

Every verification step, from the development of top-level specifications to machine-aided proof of the Pascal code, was carried out. Although these steps were not completed for all portions of the kernel, most of the job was done for much of the kernel. The remainder is clearly more of the same. We therefore consider the project essentially complete. In this paper, as each verification step is discussed, an estimate of the completed portion of that step is given, together with an indication of the amount of work required for completion. One should realize that it is essential to carry the verification process through the steps of actual code-level proofs because most security flaws in real systems are found at this level [20]. Security flaws were found in our system during verification, despite the fact that the implementation was written carefully and tested extensively. An example of

Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components

Communications
of
the ACM

February 1980
Volume 23
Number 2

August 2009



Slashdot is powered by [your submissions](#)

Technology: World's First

Posted by [Soulskill](#) on Thursday Aug 27, 2009
from the wait-for-it dept.

An anonymous reader writes

"Operating systems usually have bugs and so forth are known by almost everyone. But now we have a way to [prove that a particular OS kernel](#) is formally verified, and as such it is guaranteed to be free of bugs. Researchers used an executable version of the Isabelle theorem prover to generate a proof that matches the executable and the kernel code.

Does it run Linux? ["We're pleased to say that it does."](#)

A NICTA bejelentette a világ első, formális módszerekkel igazolt,



New Scientist

Saturday 29/8/2009

Page: 21

Section: General News

Region: National

Type: Magazines Science / Technology

Size: 196.31 sq.cms.

Published: -----S-

The ultimate way to keep your computer safe from harm

FLAWS in the code, or "kernel", that sits at the heart of modern computers leave them prone to occasional malfunction and vulnerable to attack by worms and viruses. So the development of a secure general-purpose microkernel could pave the

just mathematics, and you can reason about them mathematically," says Klein.

His team formulated a model with more than 200,000 logical steps which allowed them to prove that the program would always behave as its

eredményekeppen pedig egy olyan megbízhatóságot kapnak a szoftvertől, amely e

Reminders

- Arista Advanced Operating Systems Prize for top performer in this course
- OS Hall of Fame for straight HDs in OS, AOS, (Dist Syst,) OS Thesis
- Taste-of-Research opportunities at Trustworthy Systems
<https://trustworthy.systems/students/internships>
Deadline: 27 October!
- Honours theses at Trustworthy Systems
<https://trustworthy.systems/students/theses>
- **NEW: John Lions CS Honours Award for thesis in OS**
<https://www.scholarships.unsw.edu.au/scholarships/id/1757>
Deadline: 25 November for T1/25!