UNSW SYDNEY | Australia's Global University

School of Computer Science & Engineering

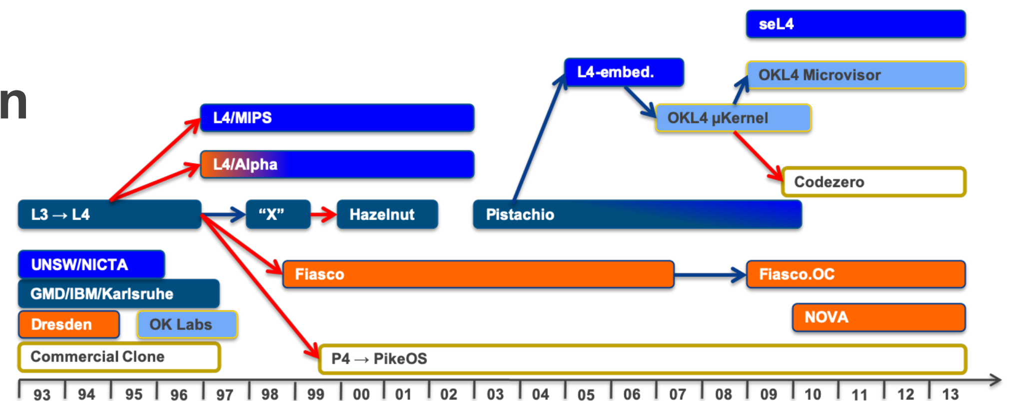
COMP9242 Advanced Operating Systems

2022 T2 Week 07 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

© Gernot Heiser 2019 – CC BY 4.0

Today's Lecture

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

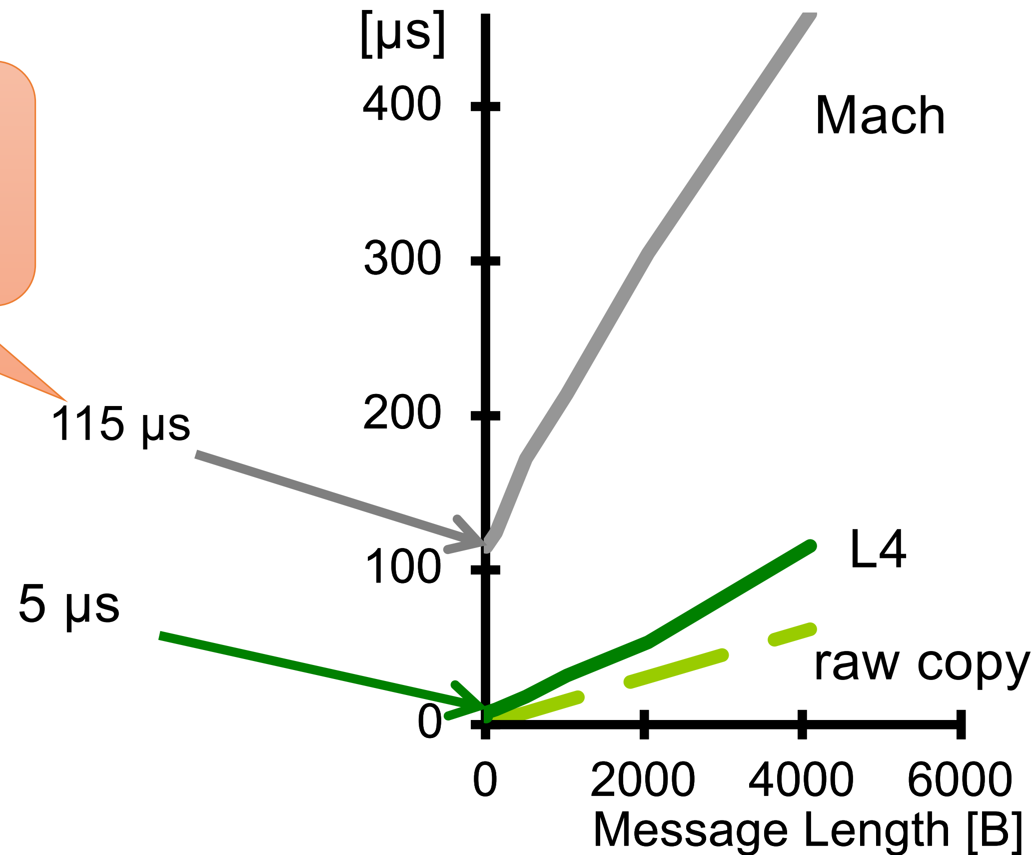
© Gernot Heiser 2019 – CC BY 4.0

L4: The Quest for a Real Microkernel

© Gernot Heiser 2019 – CC BY 4.0

1993 “Microkernel”: IPC Performance

Culprit:
Cache footprint
[Liedtke’95]



i486 @
50 MHz

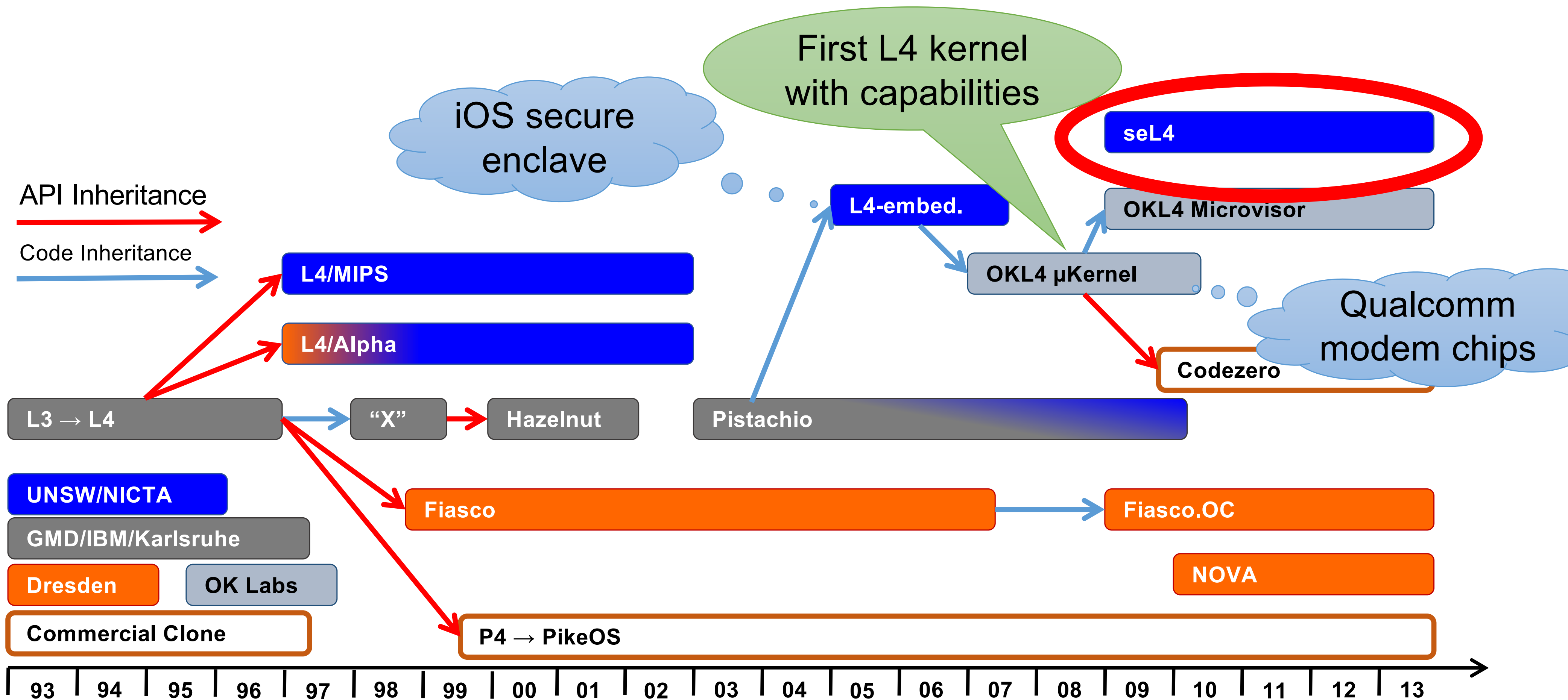
© Gernot Heiser 2019 – CC BY 4.0

The 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]

© Gernot Heiser 2019 – CC BY 4.0

L4: 25 Years High Performance Microkernels

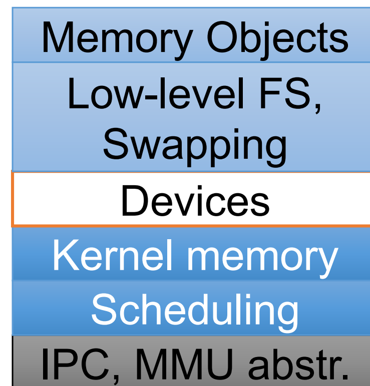


© Gernot Heiser 2019 – CC BY 4.0

Microkernel Evolution

First generation

Mach ['87], QNX, Chorus



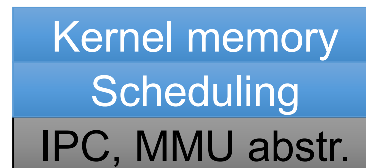
180 syscalls, 100 kSLOC

100 µs IPC

Second generation

L4 ['95], PikeOS, Integrity

Minix 3



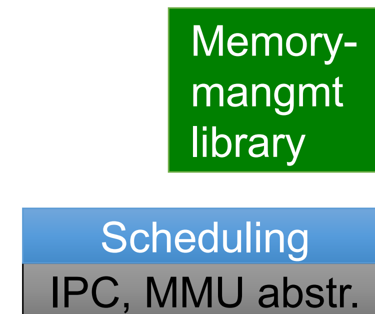
~7 syscalls, ~10 kSLOC

~ 1 µs IPC (L4)

Others much slower!

Third generation

seL4 ['09]



~3 syscalls, ~10 kSLOC

0.1 µs IPC

Capabilities

Design for isolation

© Gernot Heiser 2019 – CC BY 4.0

L4 1-Way IPC Performance Over the Years

Name	Year	Processor	MHz	Cycles	μs
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**

© Gernot Heiser 2019 – CC BY 4.0

Independent Comparison

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

Source	**seL4**	Fiasco.OC	Zircon
Mi et al, 2019, EuroSys'20	**986**	2717	8157
seL4.systems, Jul'22	**763**	– –	– –

SW overheads dominate

Cost dominated by mandatory HW operations

Operation	Cycles	RT Cycles
SYSCALL	**82**	**164**
Switch PT	**186**	**372**
SYSRET	**75**	**150**
Total	**343**	**686**

© Gernot Heiser 2019 – CC BY 4.0

Minimality: Source Lines of Code (SLOC)

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

© Gernot Heiser 2019 – CC BY 4.0
UNSW SYDNEY

Issues With 2G Microkernels

- L4 solved microkernel performance [Härtig et al, SOSP'97]
- Left a number of issues unsolved
- Problem: ad-hoc approach to security and resource management
  - Global thread name space ⇒ covert channels [Shapiro'03]
  - Threads as IPC targets ⇒ insufficient encapsulation
  - No delegation of authority ⇒ impacts flexibility, performance
  - Single kernel memory pool ⇒ DoS attacks
  - Unprincipled management of time

Caps & endpoints

seL4 memory management model

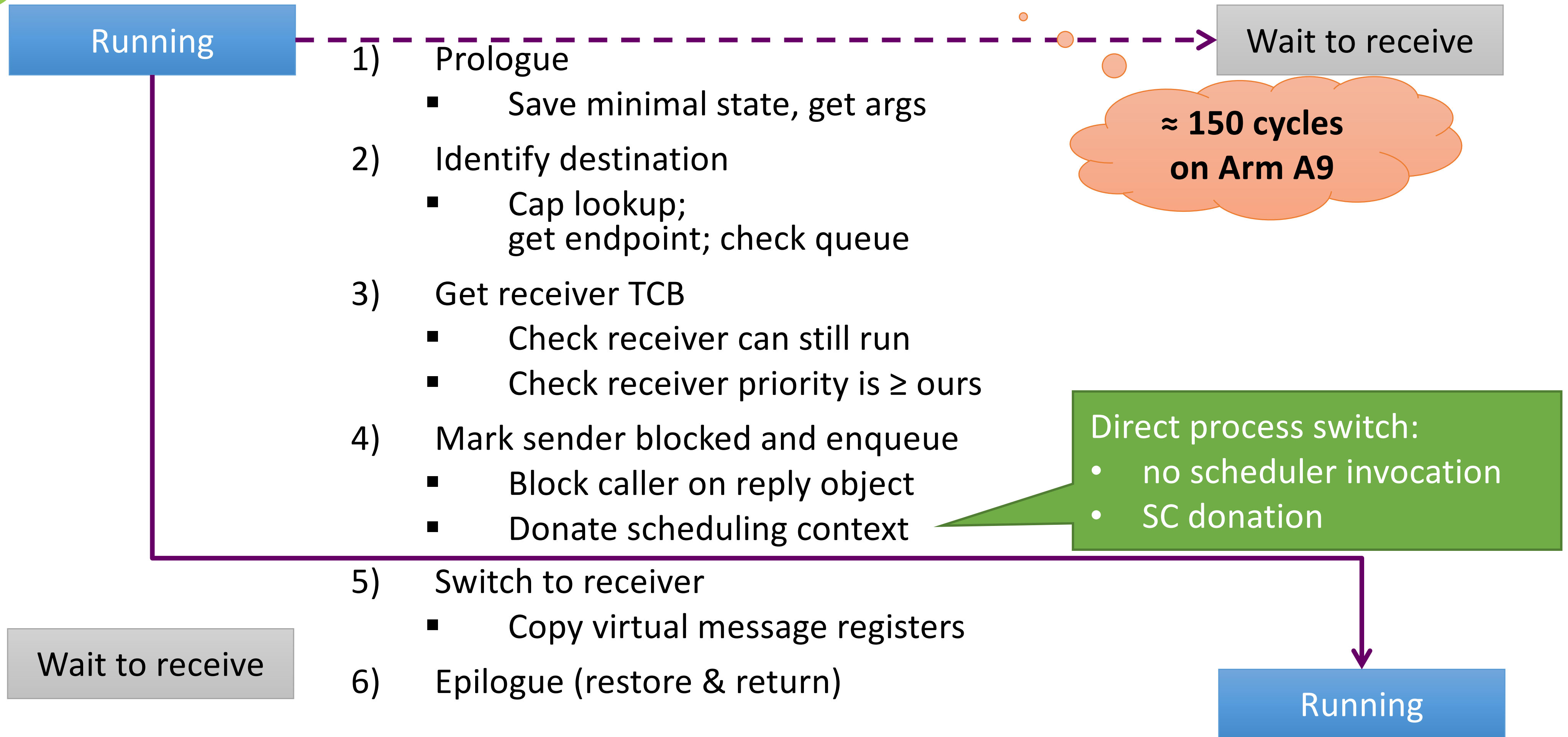
seL4 scheduling contexts

© Gernot Heiser 2019 – CC BY 4.0

Implementation Highlights

© Gernot Heiser 2019 – CC BY 4.0

IPC Fastpath: Send Phase of Call



1) Prologue
   - Save minimal state, get args
2) Identify destination
   - Cap lookup; get endpoint; check queue
3) Get receiver TCB
   - Check receiver can still run
   - Check receiver priority is ≥ ours
4) Mark sender blocked and enqueue
   - Block caller on reply object
   - Donate scheduling context
5) Switch to receiver
   - Copy virtual message registers
6) Epilogue (restore & return)

© Gernot Heiser 2019 – CC BY 4.0

seL4 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 (slightly)
  - insignificant compared to basic slow-path cost

© Gernot Heiser 2019 – CC BY 4.0

How About Real-Time Support?

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

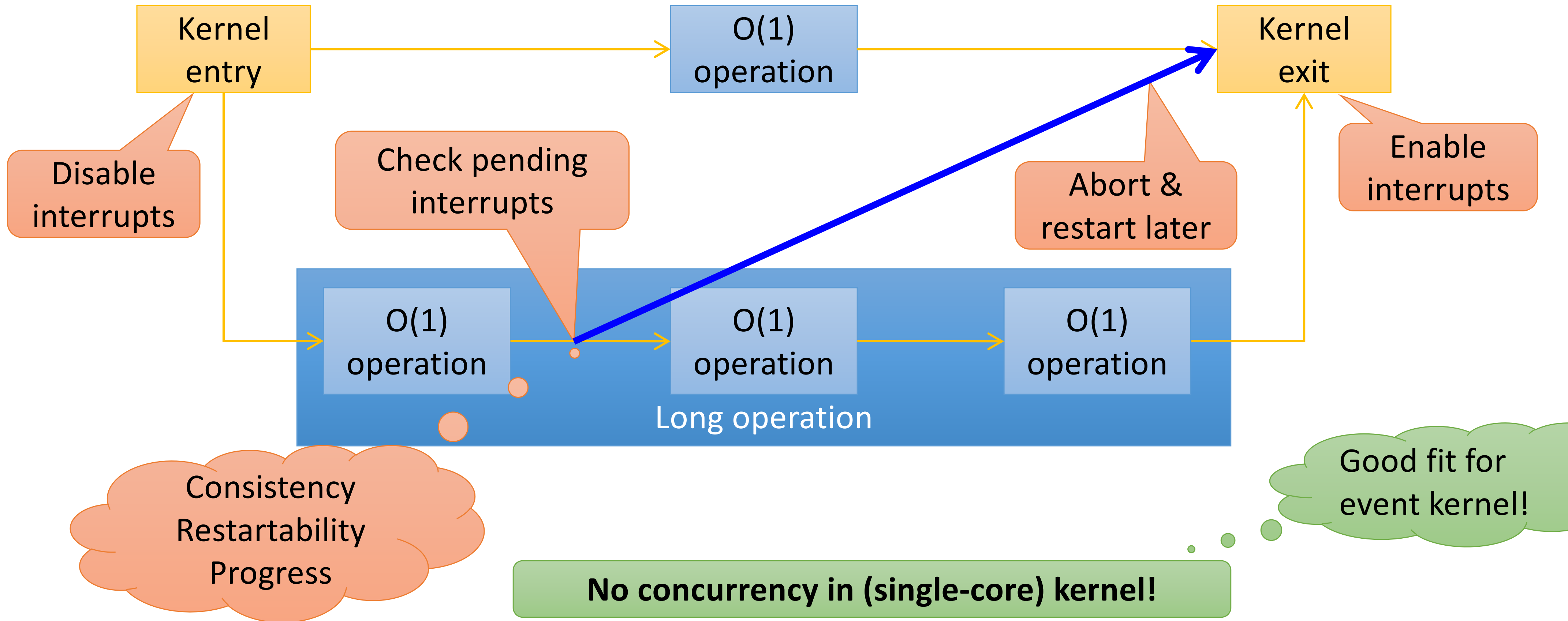
How about long-running system calls?

Most protected-mode RTOSes are fully preemptible

Lots of concurrency in kernel!

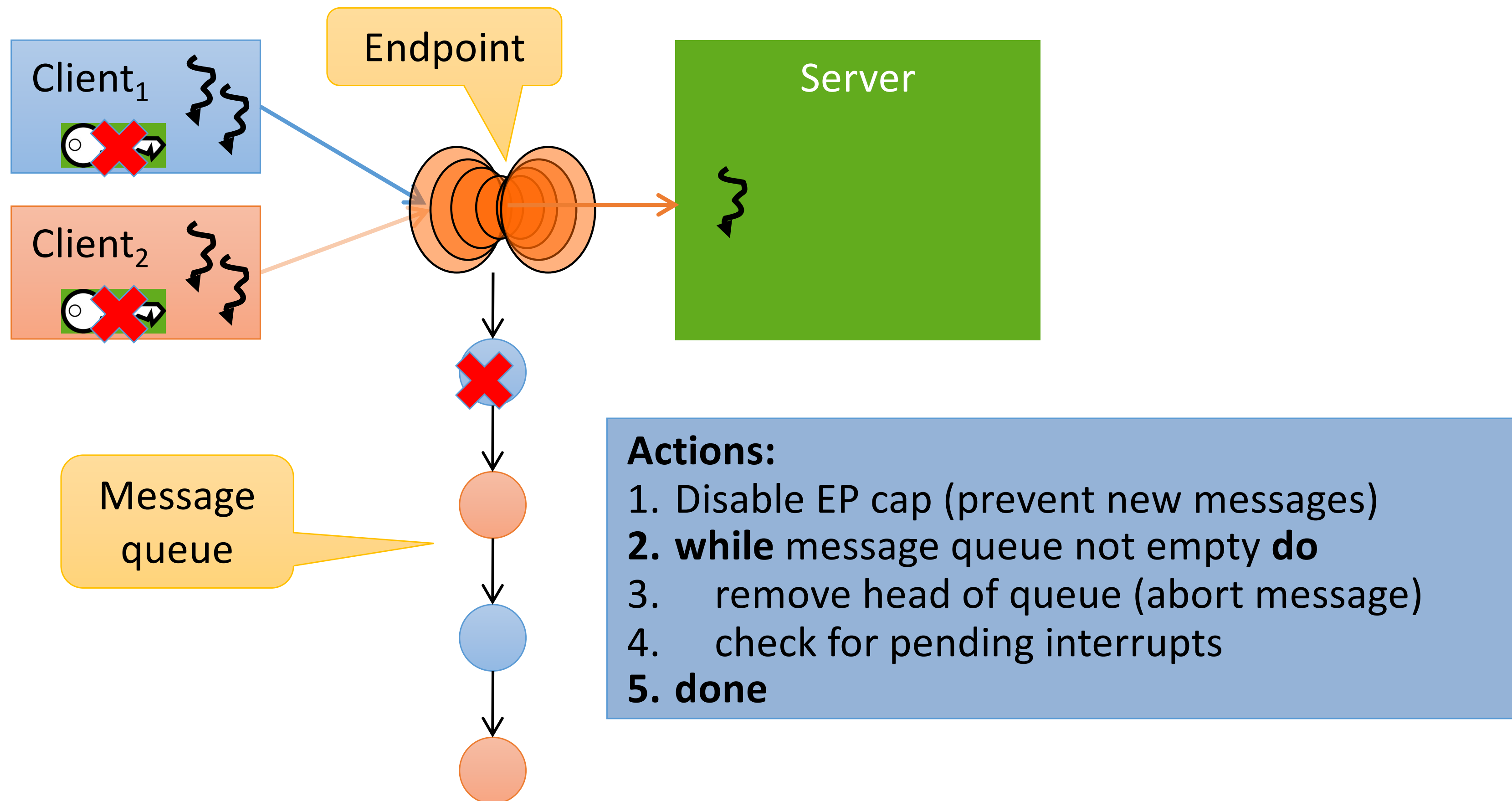
© Gernot Heiser 2019 – CC BY 4.0

Incremental Consistency Paradigm



© Gernot Heiser 2019 – CC BY 4.0

Example: Destroying IPC Endpoint

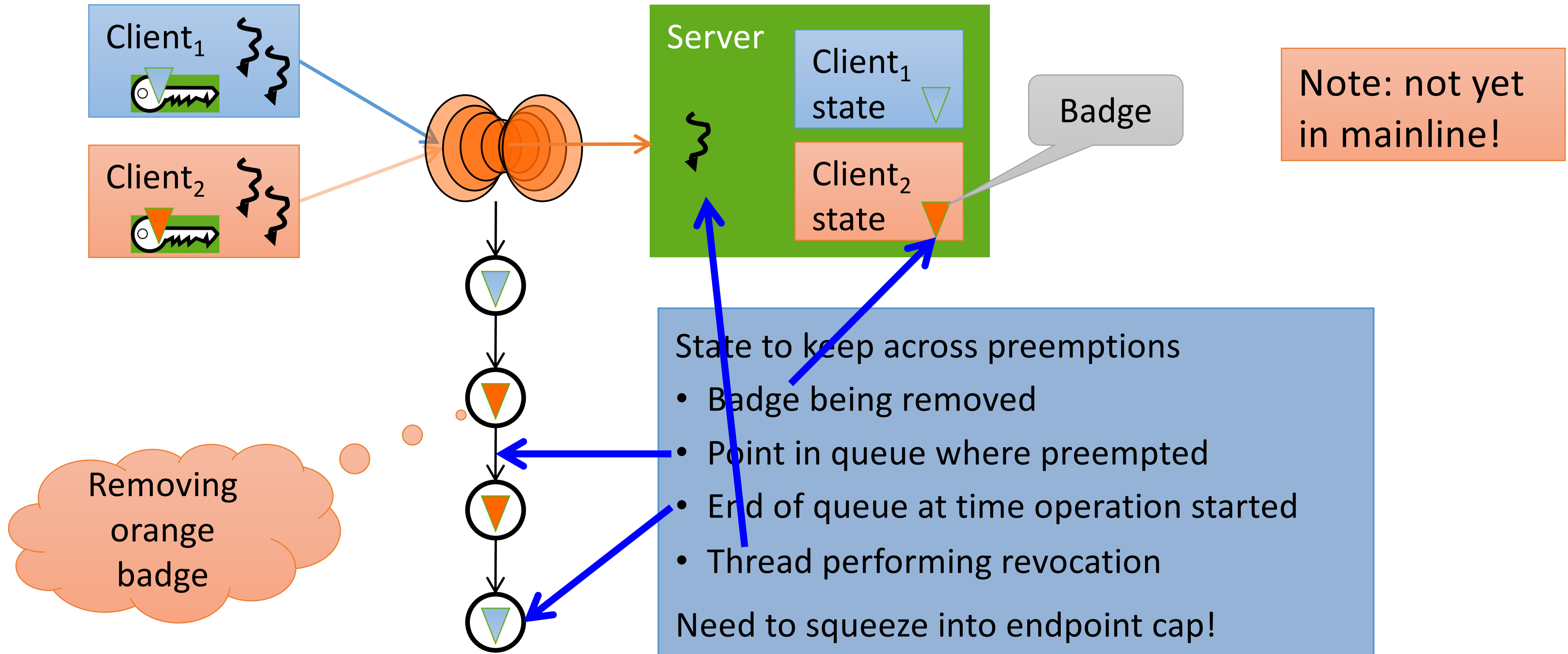


Actions:

1. Disable EP cap (prevent new messages)
2. **while** message queue not empty **do**
3.   remove head of queue (abort message)
4.   check for pending interrupts
5. **done**

© Gernot Heiser 2019 – CC BY 4.0

Difficult Example: Revoking Badge



© Gernot Heiser 2019 – CC BY 4.0

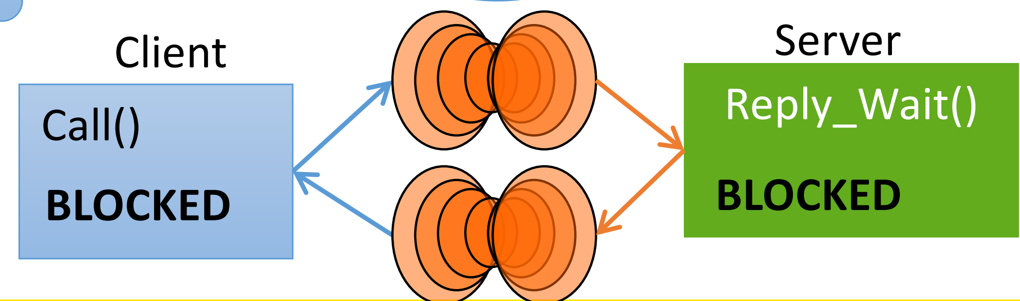
L4 Scheduler Optimisation: Lazy Scheduling

```
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!

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

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



© Gernot Heiser 2019 – CC BY 4.0

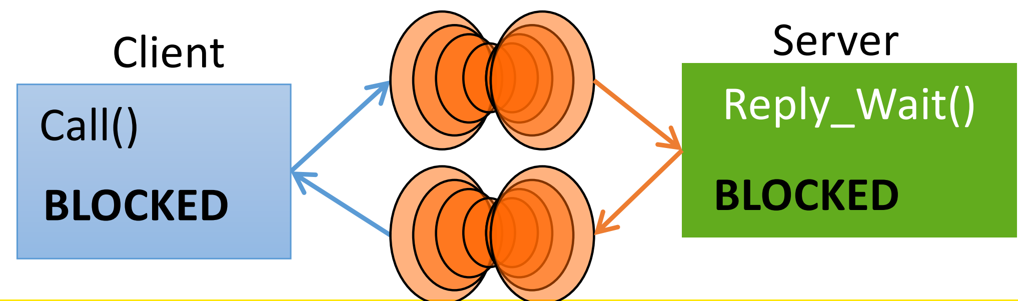
Scheduler: Benno Scheduling

```
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 preemtion time!

Idea: Lazy on *unblocking* instead on *blocking*

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



© Gernot Heiser 2019 – CC BY 4.0

Scheduler Optimisation: Direct Process Switch

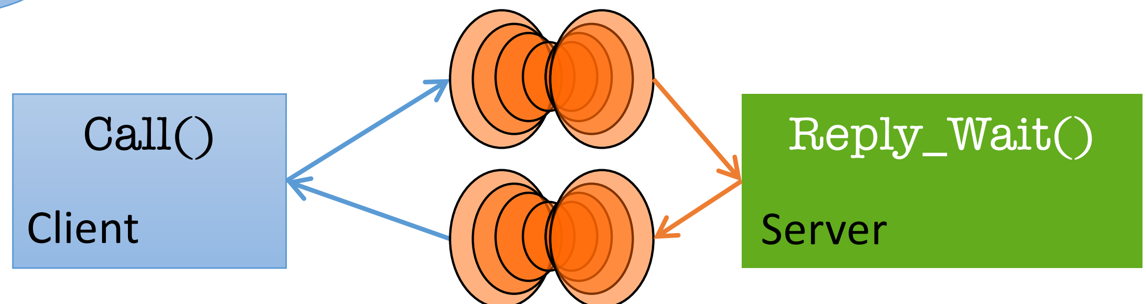
- Sender was running ⇒ had highest prio
- If receiver prio ≥ sender prio ⇒ run receiver

- Arguably, sender should donate back if it's a server replying to a Call()
- Hence, always donate on Reply_Wait()

Implication: Time slice donation – receiver runs on sender's time slice – *how long?*

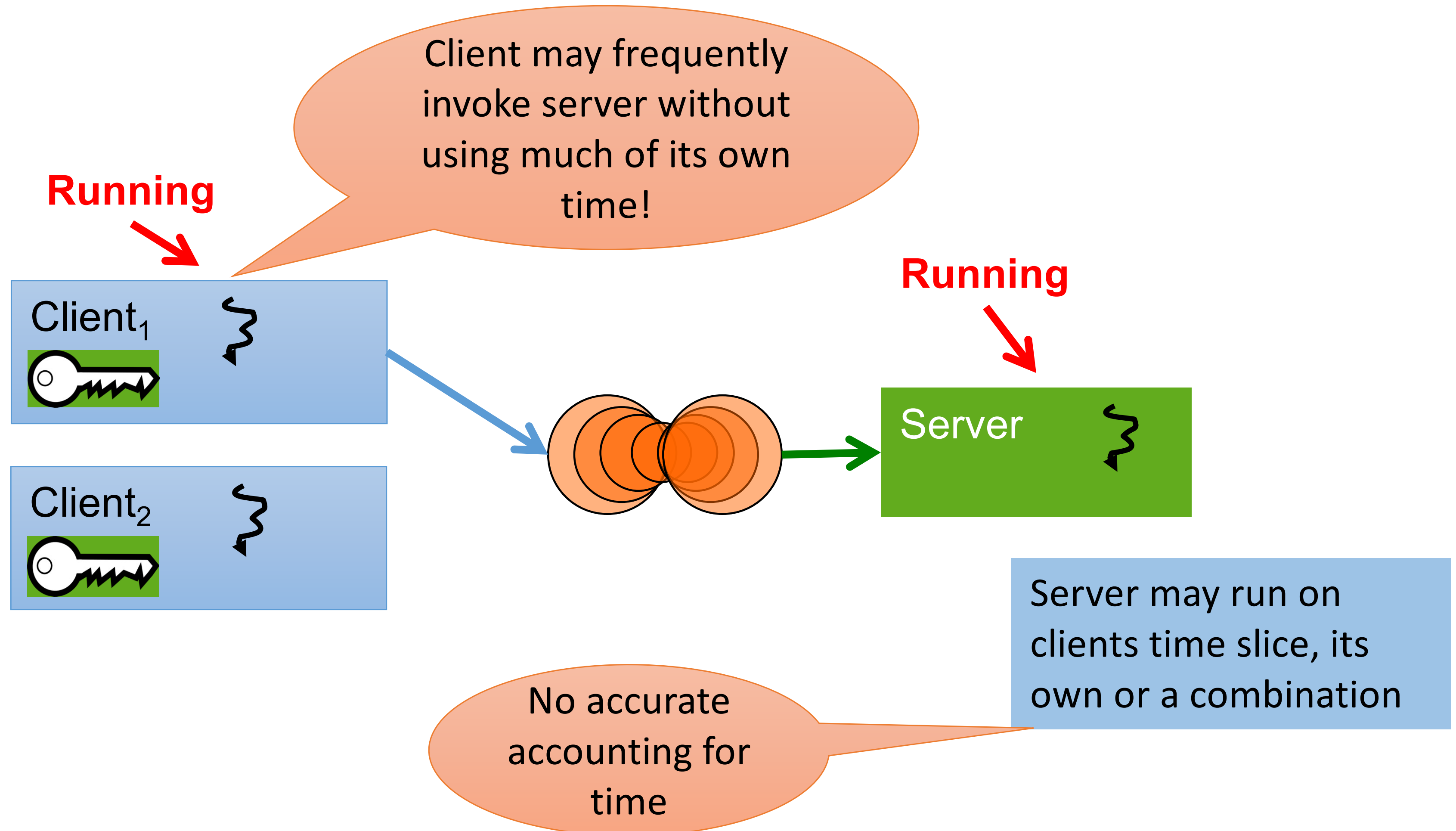
Idea: Don't invoke scheduler if you know who'll be chosen

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



© Gernot Heiser 2019 – CC BY 4.0

Remember: Delegation of Critical Sections



© Gernot Heiser 2019 – CC BY 4.0

MCS Model: Scheduling Contexts

Classical thread attributes

- Priority
- Time slice

Not runnable if null

MCS thread attributes

- Priority
- Scheduling context capability

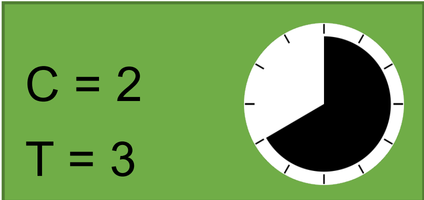
Capability for time

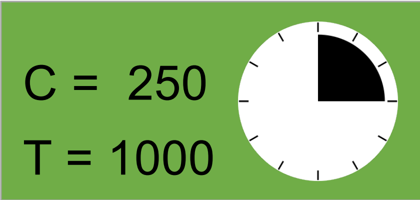
Scheduling context object

- T: period
- C: budget (≤ T)

Limits CPU access!

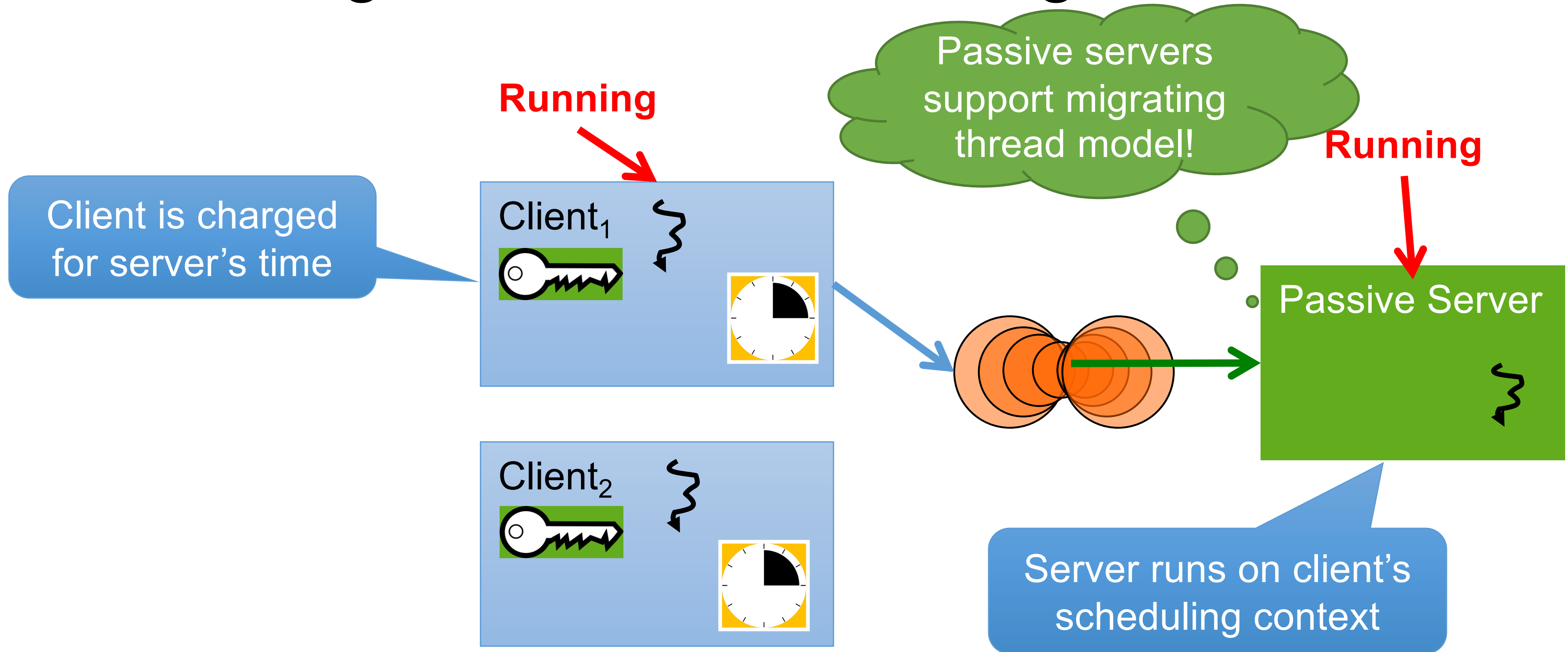
Per-core SchedControl capability conveys right to assign budgets (i.e. perform admission control)





© Gernot Heiser 2019 – CC BY 4.0

Delegation with Scheduling Contexts



Scheduling-context capabilities: a principled, light-weight OS mechanism for managing time [Lyons et al, EuroSys'18]

© Gernot Heiser 2019 – CC BY 4.0

Mixed-Criticality Support

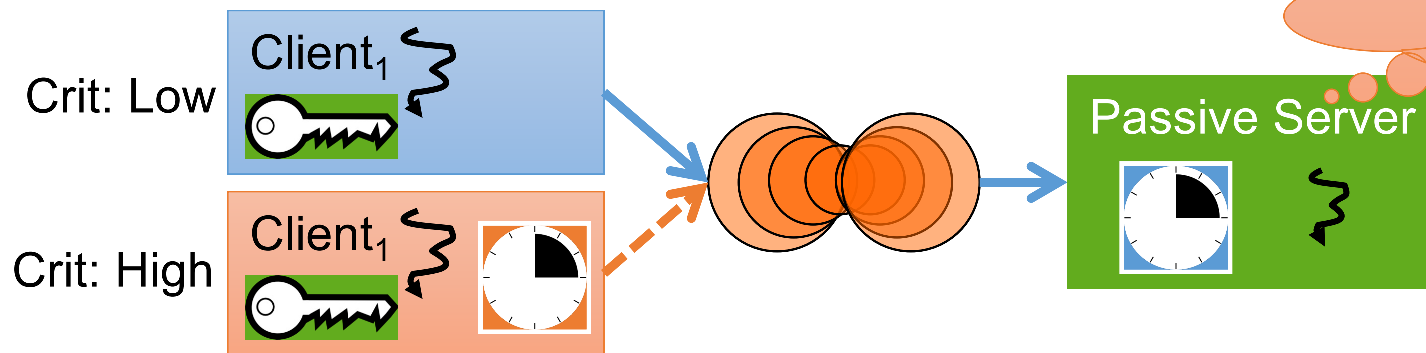
For *mixed-criticality systems* (MCS), OS must provide:

- *Temporal isolation*, to force jobs to adhere to declared WCET

Solved by scheduling contexts

- Mechanisms for *safely sharing resources* across criticalities

What if budget expires while shared server executing on Low's scheduling context?



© Gernot Heiser 2019 – CC BY 4.0

Timeout Exceptions

Policy-free mechanism for dealing with budget depletion

Possible actions:

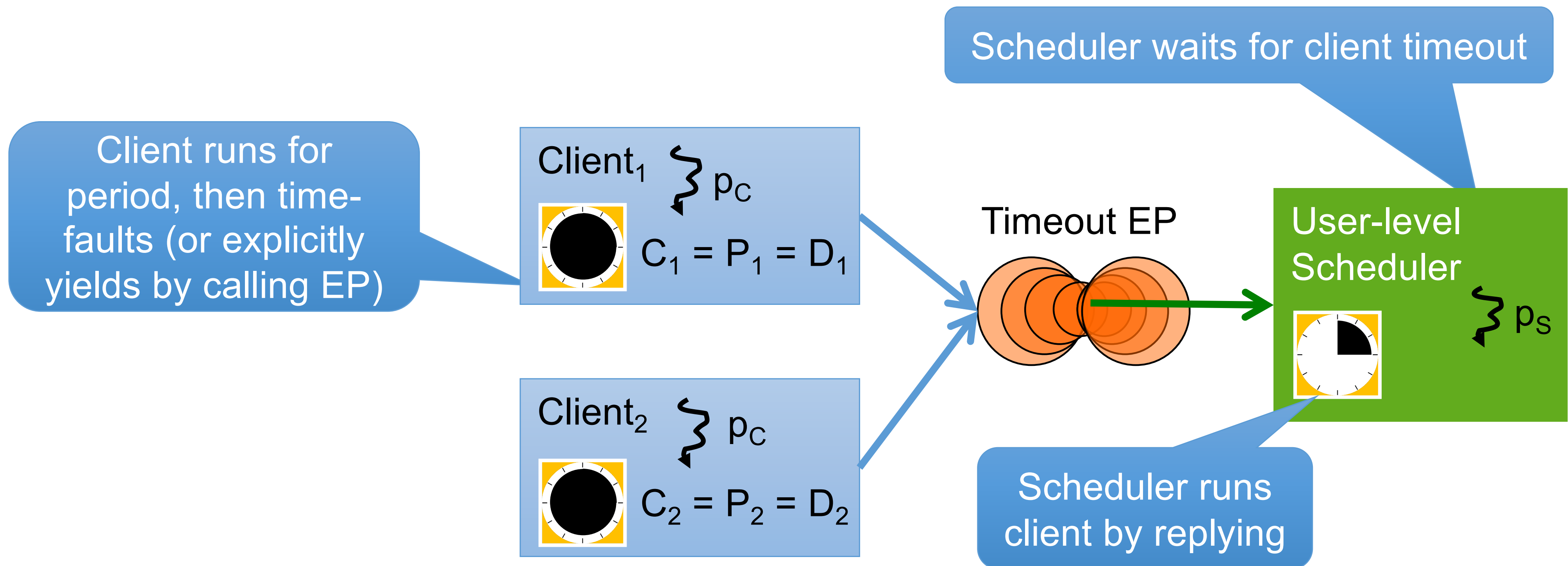
- Provide emergency budget to leave critical section
- Cancel operation & roll-back server
- Reduce priority of low-crit client (with one of the above)
- Implement priority inheritance (if you must…)

Arguable not ideal: better prevent timeout completely
Under investigation (honours thesis Mitch Johnston)

© Gernot Heiser 2019 – CC BY 4.0

Isn't a Fixed-Prio Scheduler Policy?

Implementing scheduling policy at user level



© Gernot Heiser 2019 – CC BY 4.0

User-Level EDF Scheduler Performance

preemptive
coop
LITMUS

Linux in-kernel

Time (µs)

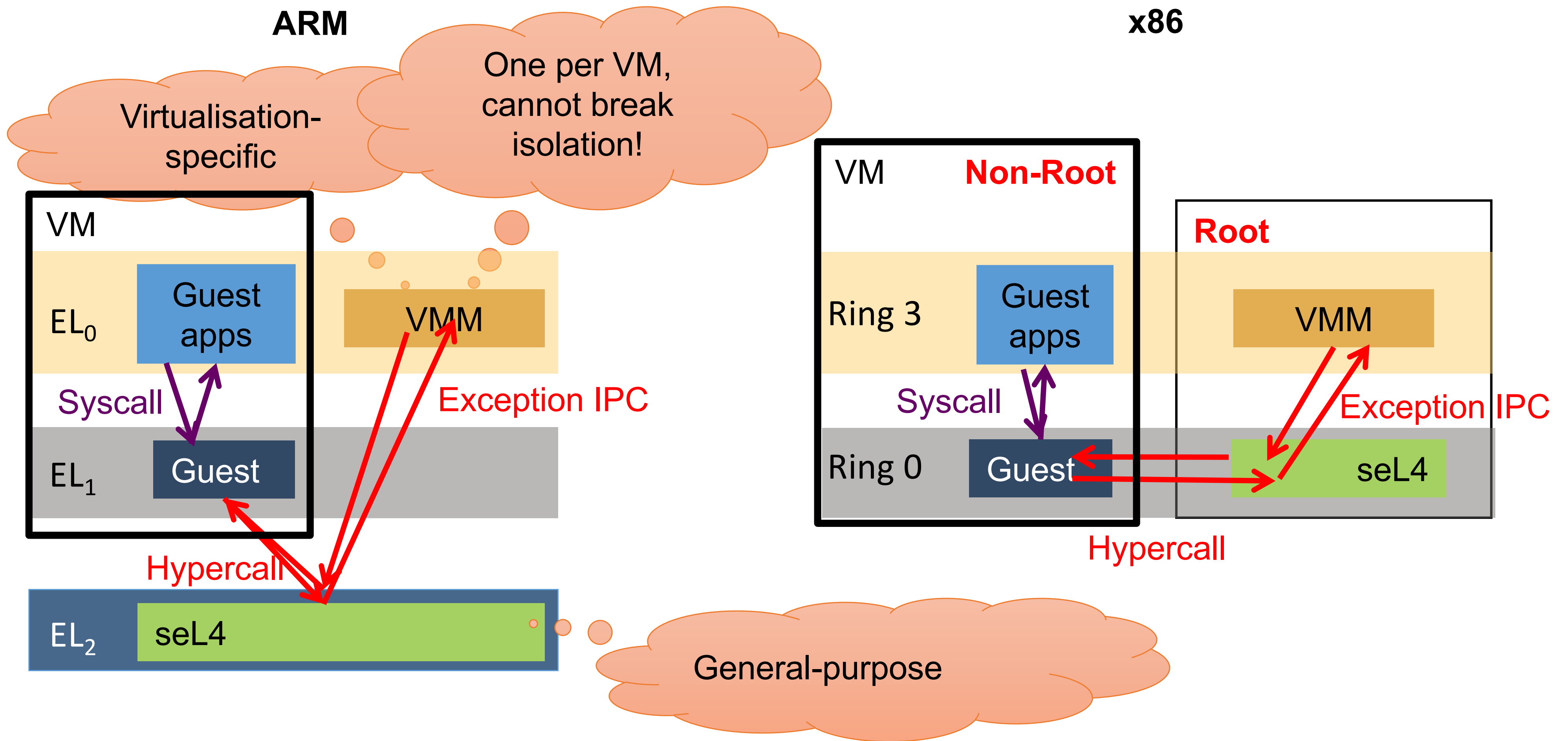
Number of threads

© Gernot Heiser 2019 – CC BY 4.0

Virtualisation: Microkernel as a Hypervisor

© Gernot Heiser 2019 – CC BY 4.0

Microkernel as Hypervisor (NOVA, seL4)



© Gernot Heiser 2019 – CC BY 4.0

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
- I/O
- Communication

© Gernot Heiser 2019 – CC BY 4.0

What *Is* the Difference?

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

Just page tables in disguise

Just kernel-scheduled activities

Real Difference?

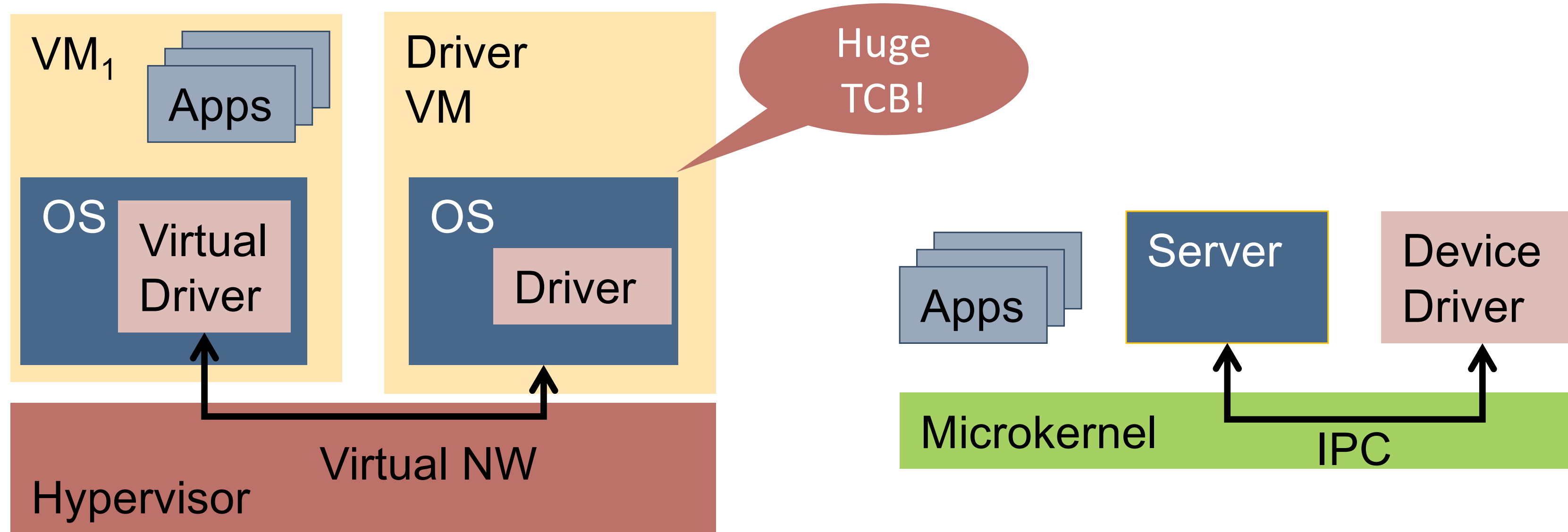
Minimal overhead, Custom API

Modelled on HW, Re-uses SW

- Similar abstractions
- Optimised for different use cases

© Gernot Heiser 2019 – CC BY 4.0

Closer Look at I/O and Communication

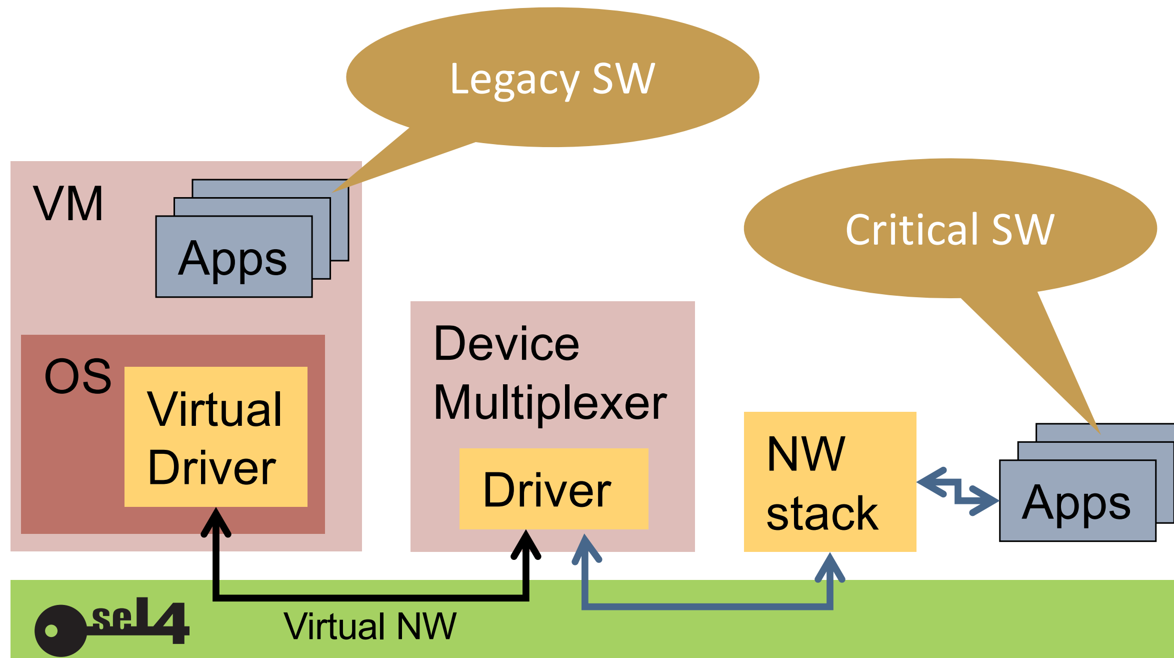


Communication is critical for I/O

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

© Gernot Heiser 2019 – CC BY 4.0

Integration: VMs and Native



© Gernot Heiser 2019 – CC BY 4.0

Lessons & Principles

© Gernot Heiser 2019 – CC BY 4.0

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
   - Synchronisation
   - Memory management – sending mappings
   - Scheduling – time-slice donation

© Gernot Heiser 2019 – CC BY 4.0

seL4 Design Principles

- Fully delegatable access control
- 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!

© Gernot Heiser 2019 – CC BY 4.0

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 of *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

© Gernot Heiser 2019 – CC BY 4.0