School of Computer Science & Engineering

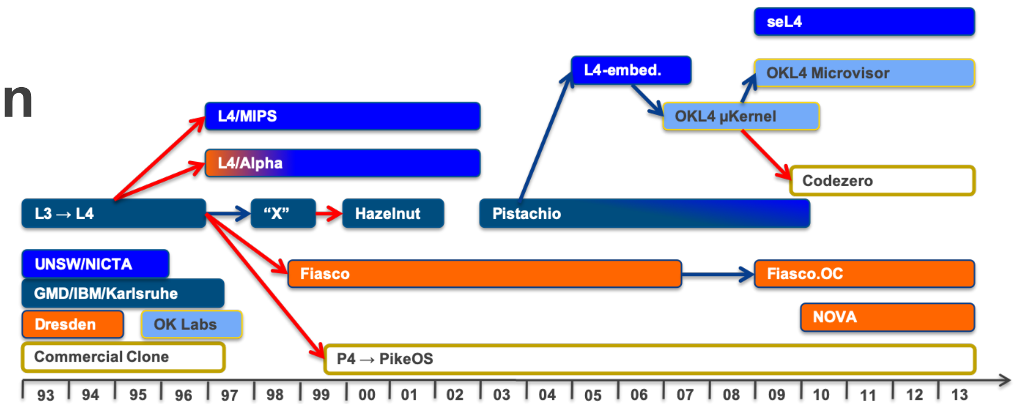
COMP9242 Advanced Operating Systems

2025 T3 Week 08 Part 2

Microkernel Design & Implementation

The 25-year quest for the right API

@GernotHeiser



Copyright Notice

L4 Microkernels – Deployed by the Billions

Today's Lecture

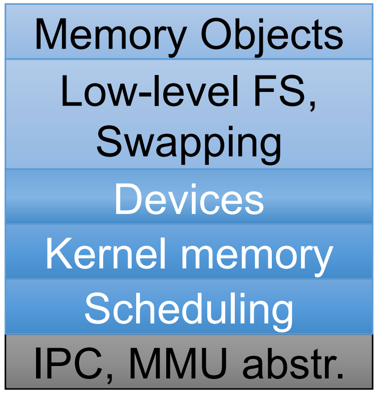
- Towards real microkernels: The history of L4 microkernels
- Implementation highlights
- Virtualisation: Microkernel as hypervisor
- Lessons and principles

L4: The Quest for a Real Microkernel

Microkernel Evolution

First generation

Mach ['87], Chorus

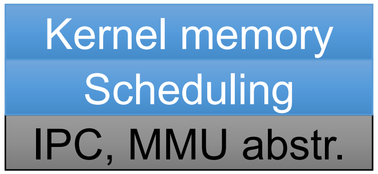


180 syscalls, 100 kSLOC

100 μs IPC

Second generation

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



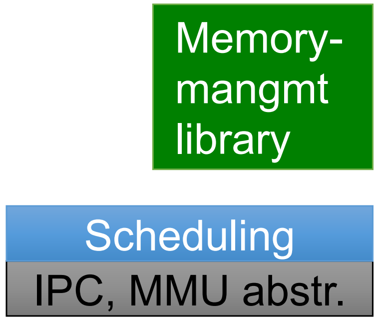
~ 7 syscalls, ~ 10 kSLOC

~ 1 μs IPC (L4)

~ 10 μs IPC (others)

Third generation

seL4 ['09]



~3 syscalls, ~10 kSLOC

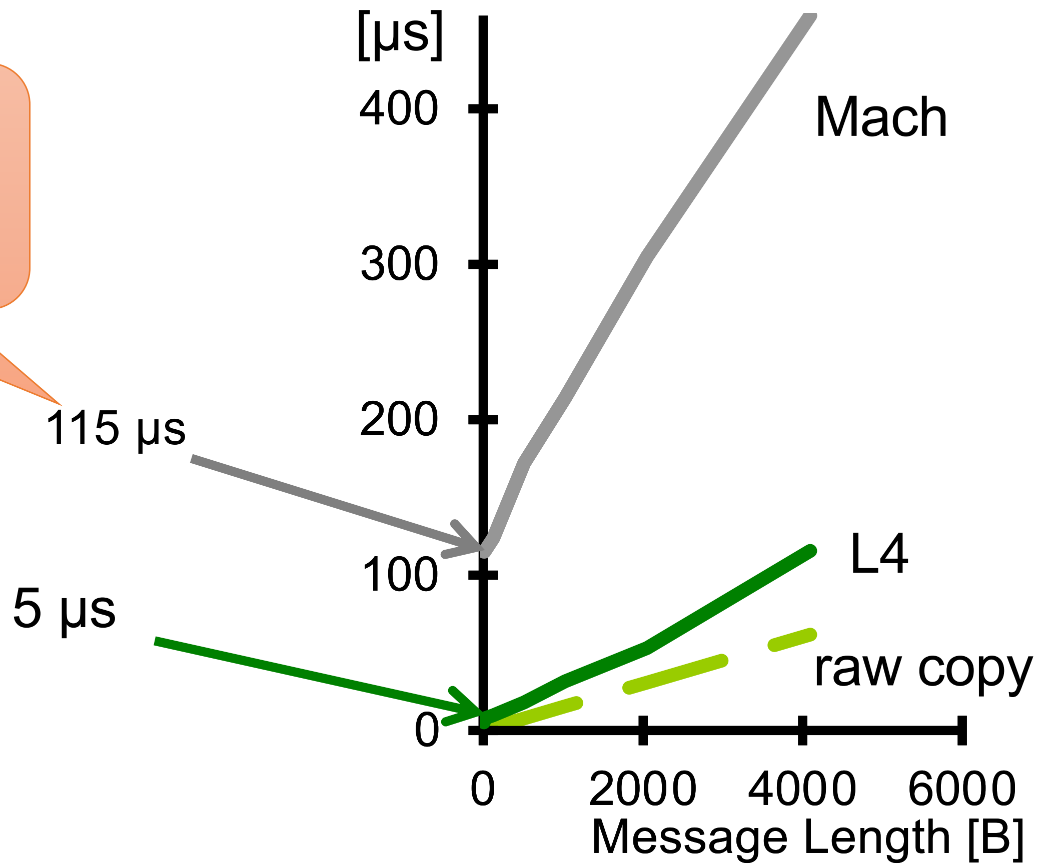
0.1–0.3 μs IPC (faster HW)

Capabilities

Design for isolation

1993 "Microkernel": IPC Performance

Culprit:
Cache footprint
[Liedtke'95]



i486 @
50 MHz

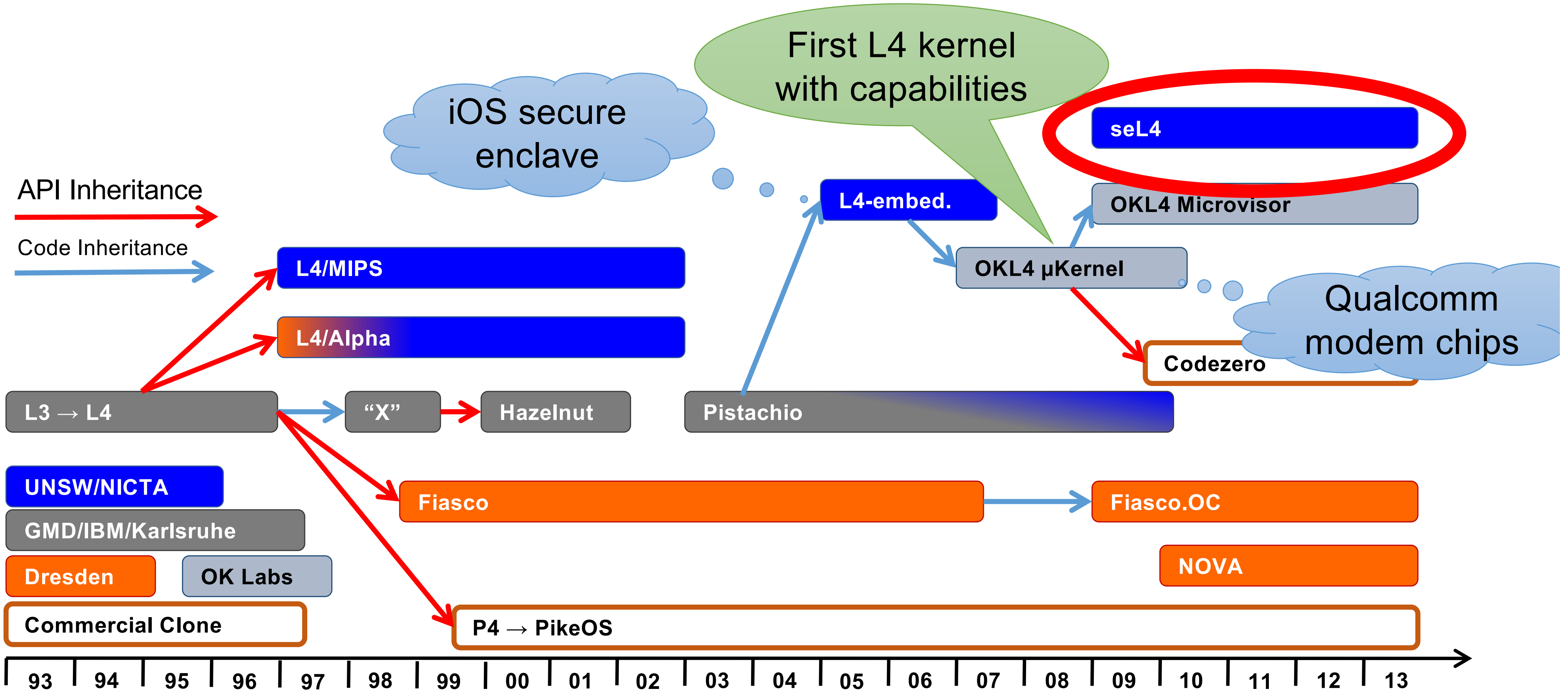
Remember: Microkernel Minimality Principle

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	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**

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, 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 (MCS)

Implementation Highlights

seL4 IPC Fastpath: Send Phase of Call

Running → Wait to receive

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)

≈ **150 cycles on Arm A9**

Point of no return:
- no state change before
- committed to complete

Direct process switch:
- no scheduler invocation
- SC donation

Wait to receive → Running

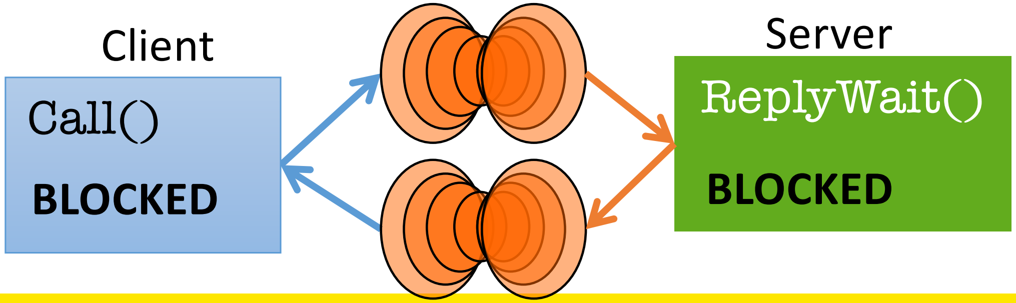
L4 Scheduler Optimisation: Lazy Scheduling

```
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



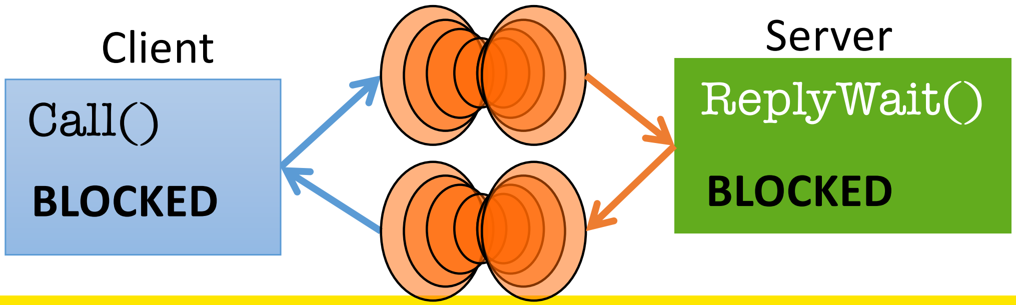
Scheduler: Benno Scheduling

```
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



Scheduler Optimisation: Direct Process Switch

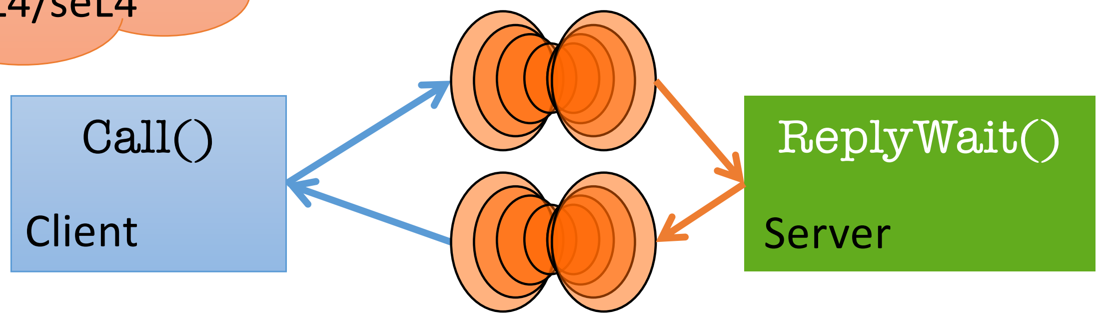
- Sender was running ⇒ had highest prio
- If receiver prio ≥ sender prio ⇒ run receiver

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()

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

Unprincipled time-slice donation in earlier L4/seL4

- 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?

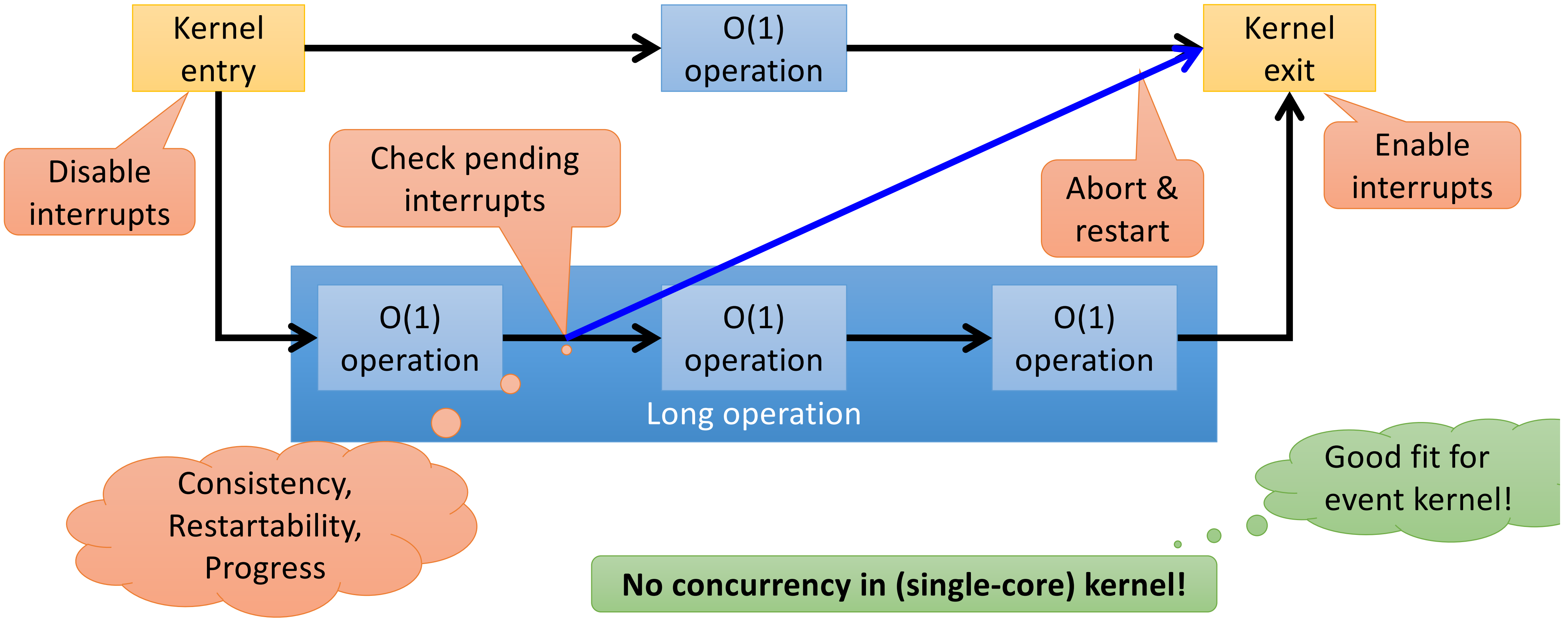
- Kernel runs with interrupts disabled
  - No concurrency control ⇒ simpler kernel
    - Easier reasoning about correctness
    - Better average-case performance

How about long-running system calls?

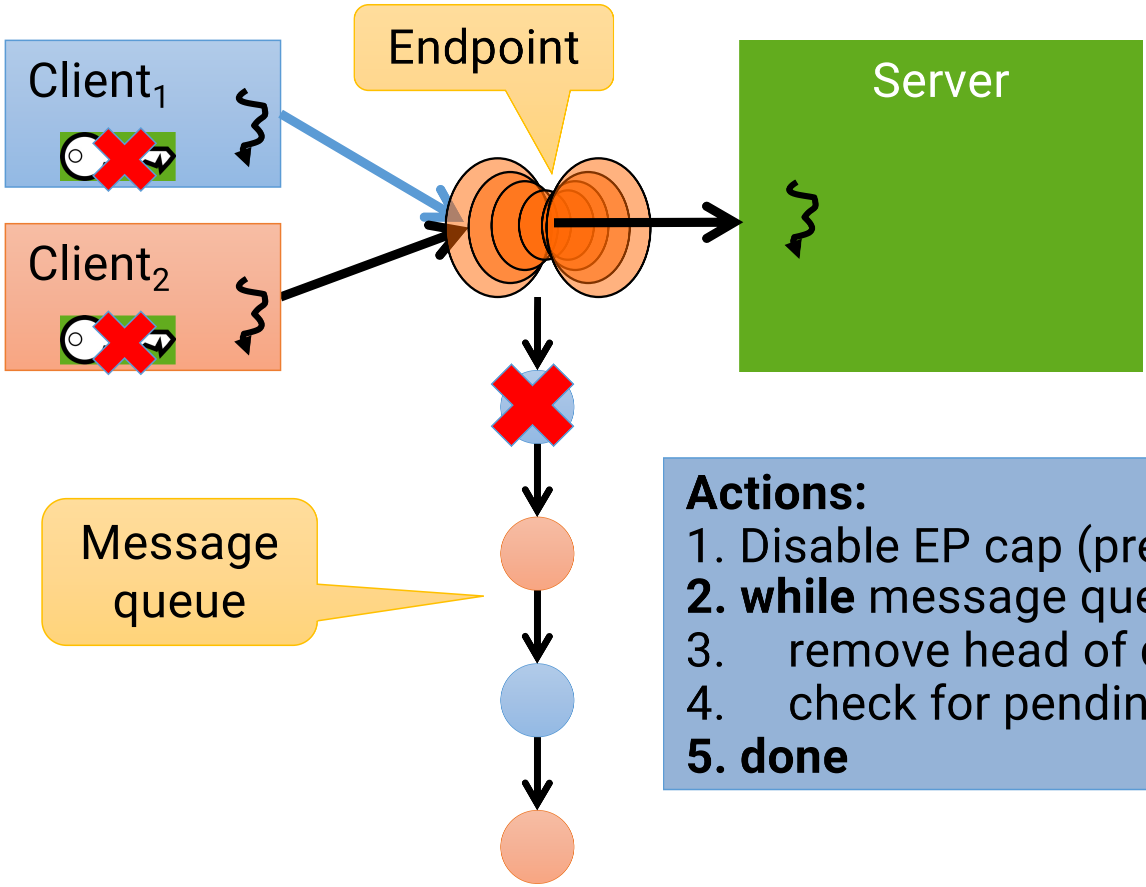
Most protected-mode RTOSes are mostly/fully preemptible

Lots of concurrency in kernel!

Incremental Consistency Paradigm

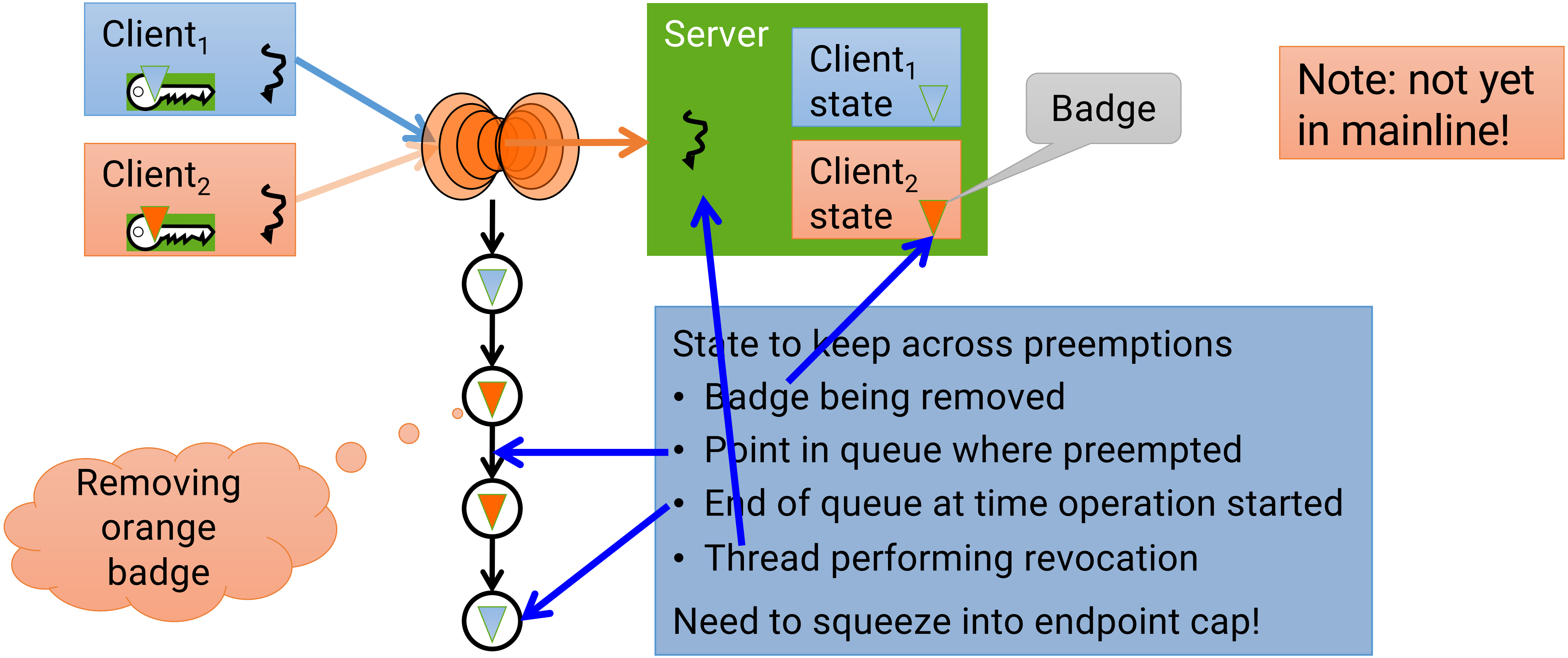


Example: Destroying IPC Endpoint



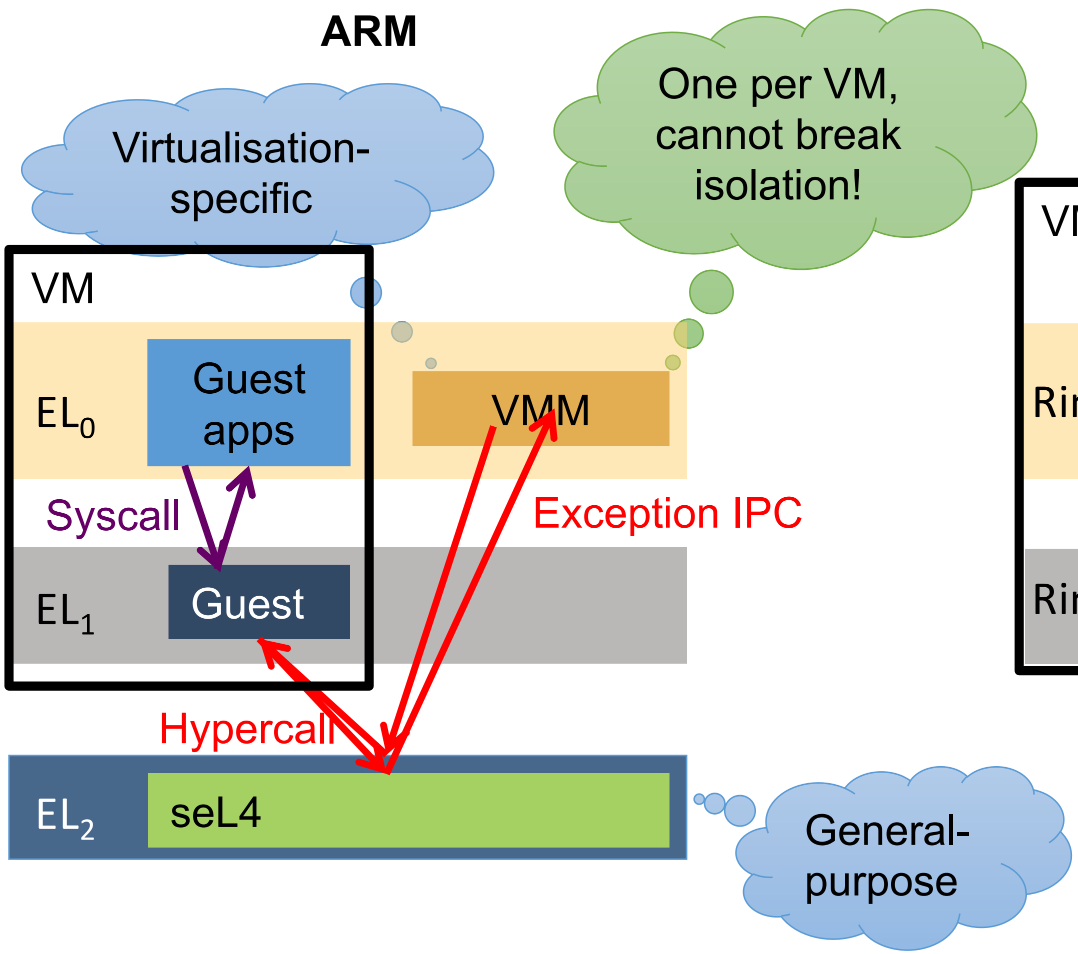
Actions:
1. Disable EP cap (prevent new messages)
2. while message queue not empty **do**
3. remove head of queue (abort message)
4. check for pending interrupts
5. done

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 *Is* 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

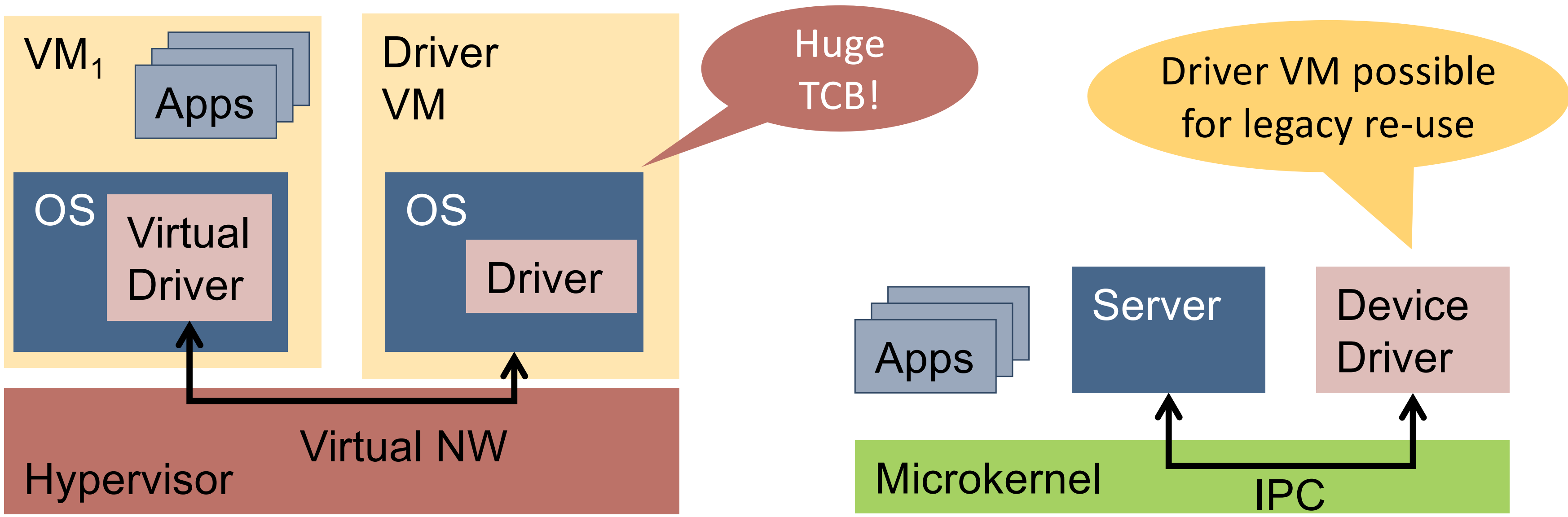
Modelled on HW, Re-uses SW

Minimal overhead, Custom API

Real Difference?

- Similar abstractions
- Optimised for different use cases

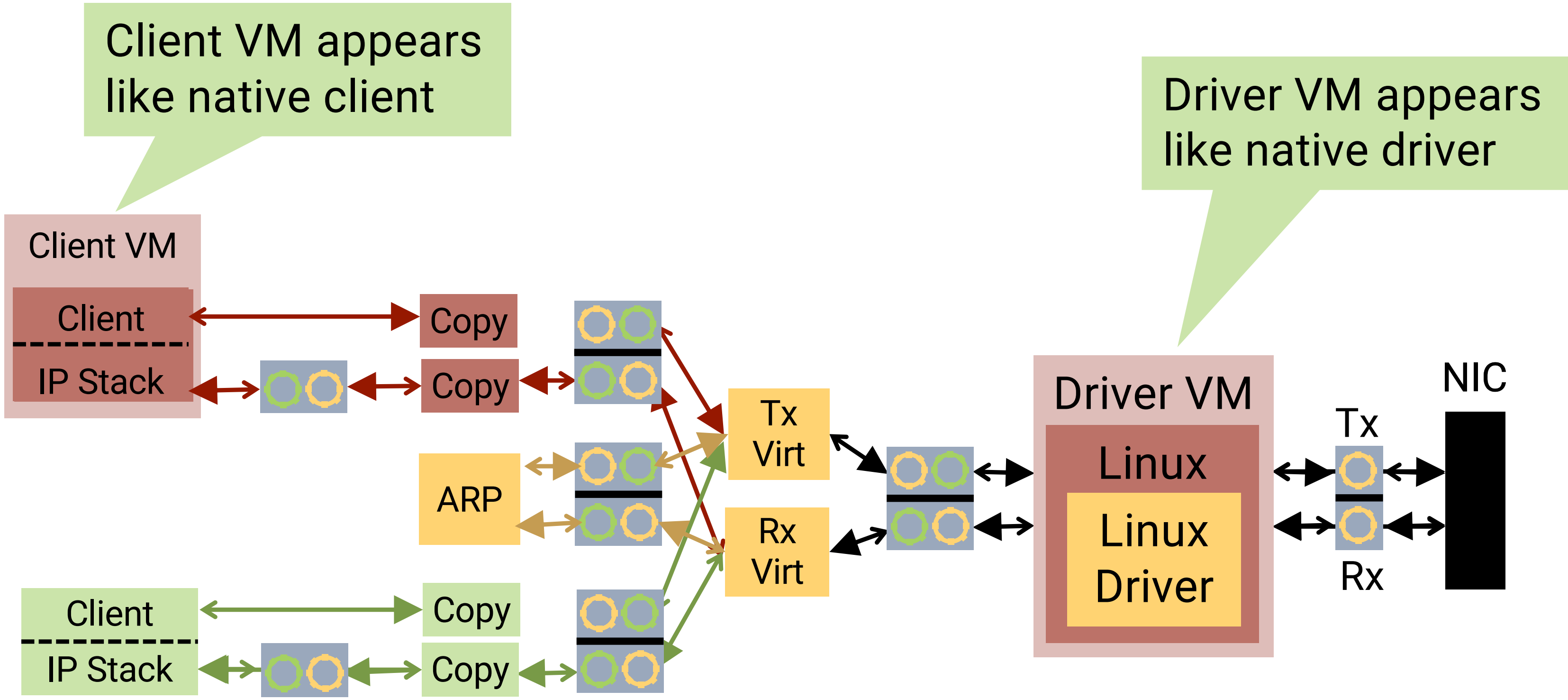
Closer Look at Communication and I/O



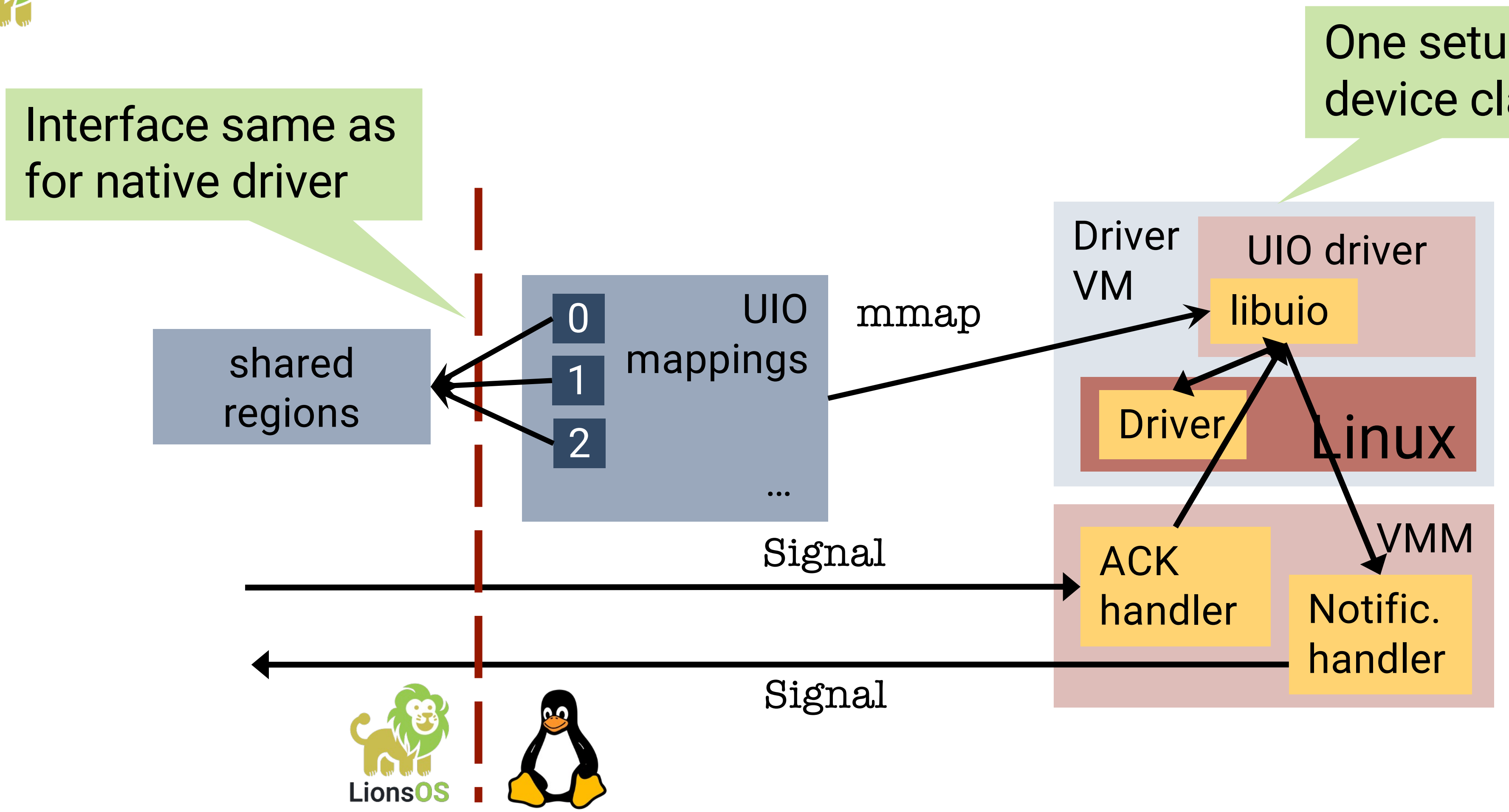
Communication is critical for I/O

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

Integration: VMs & Native



LionsOS Driver VMs



Lessons & Principles

Reflecting on Lessons of 2nd Generation

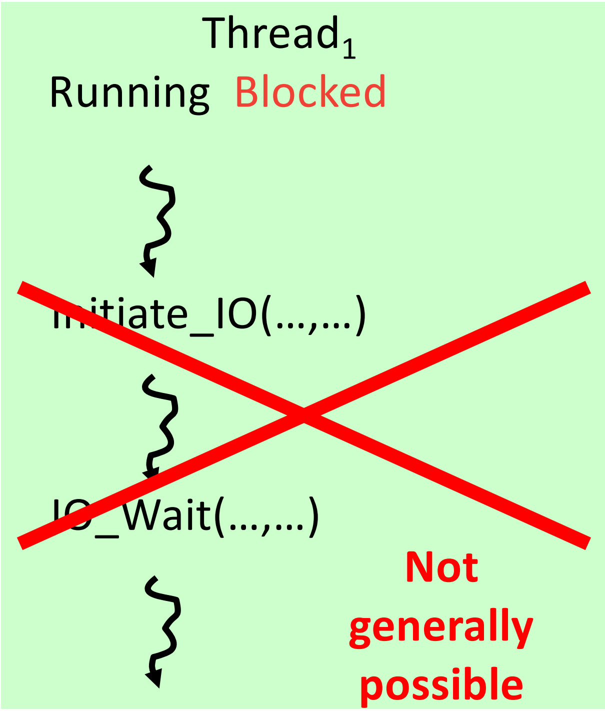
Original L4 design had two major shortcomings:

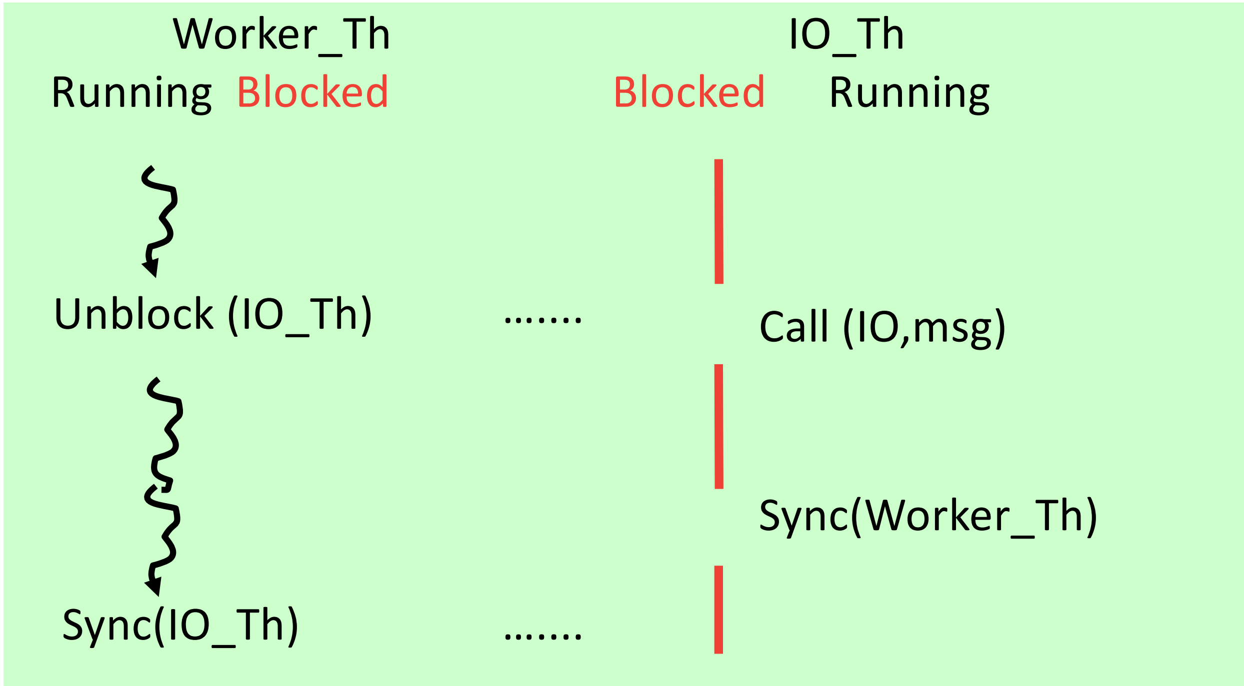
1. Insufficient/impractical resource control
    - Poor/non-existent control over kernel memory use
    - Inflexible & costly process hierarchies (policy!)
    - Arbitrary limits on number of address spaces and threads (policy!)
    - Poor information hiding (IPC addressed to threads)
    - Insufficient 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

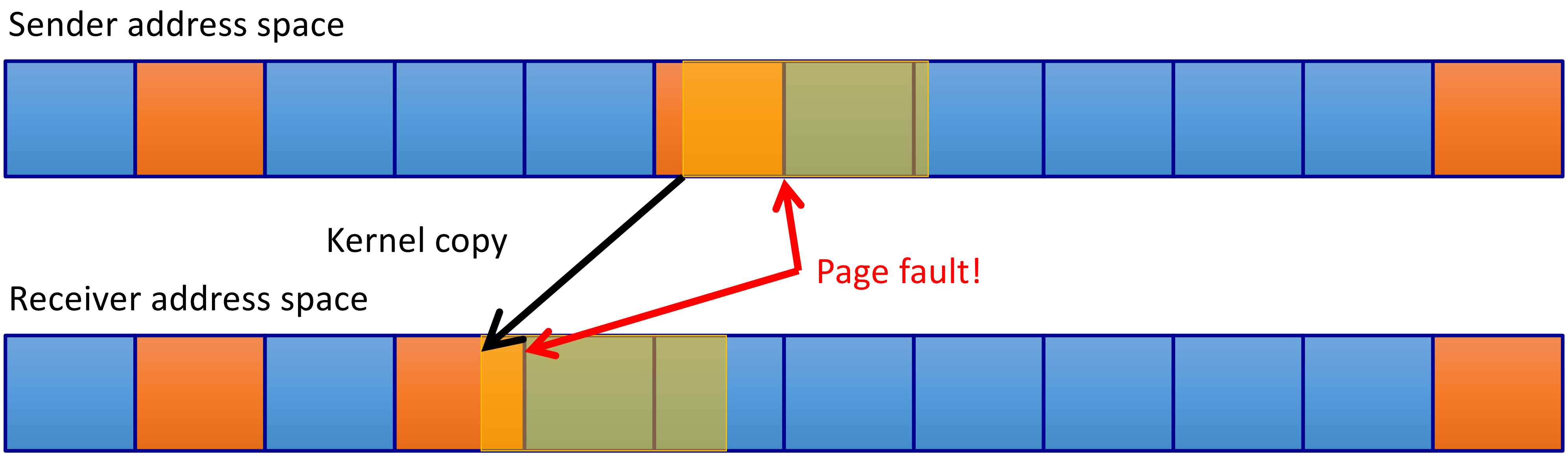




L4 "Long" IPC

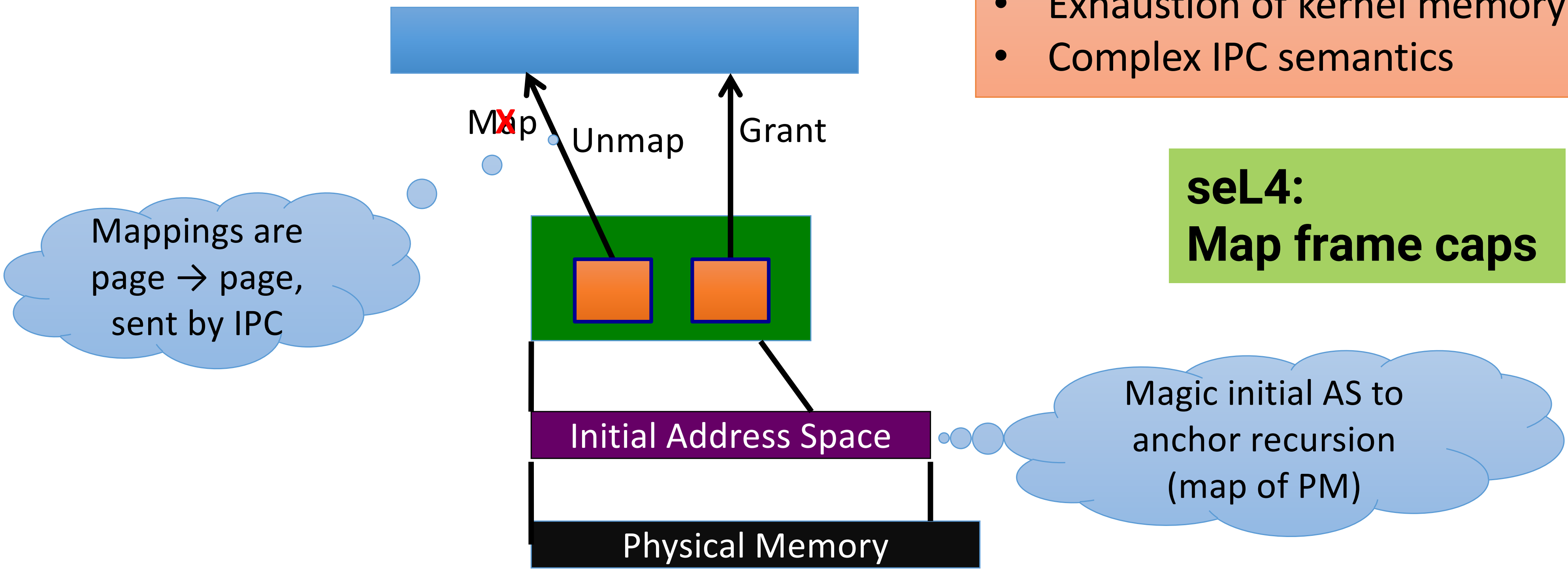
seL4: Removed

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



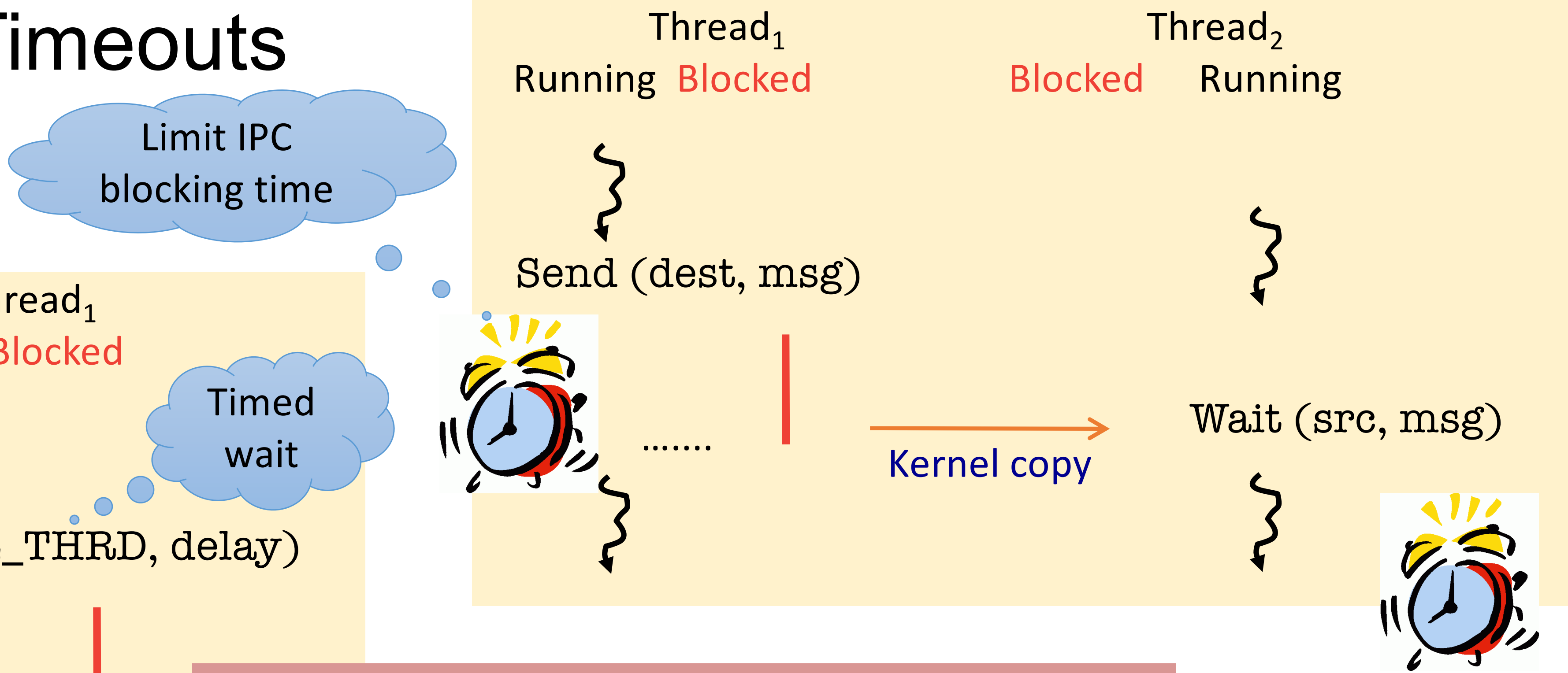
Traditional L4: Recursive Address Spaces

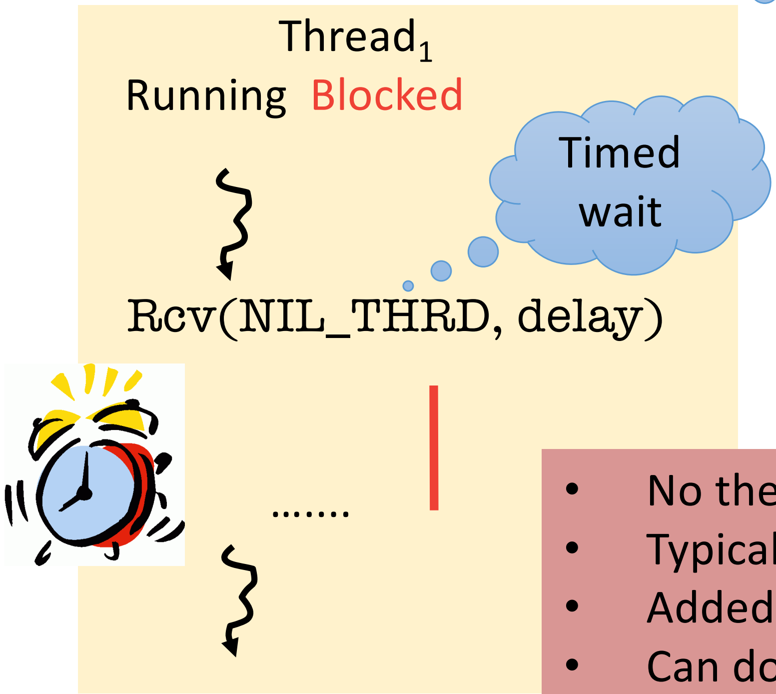
Issues:
- Complex mapping DB
- Exhaustion of kernel memory
- Complex IPC semantics



seL4:
Map frame caps

L4 Timeouts





- No theory/heuristics for determining timeouts
- Typically servers reply with zero T.O., else ∞
- Added complexity
- Can do timed wait with timer syscall

seL4: Removed timeouts