School of Computer Science & Engineering

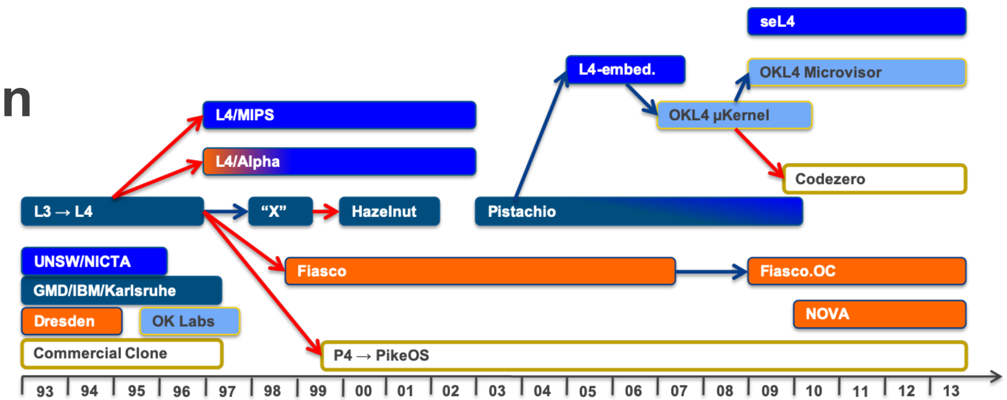
COMP9242 Advanced Operating Systems

2023 T3 Week 05 Part 1

Microkernel Design & Implementation

The 25-year quest for the right API

@GernotHeiser



Copyright Notice

L4 Microkernels – Deployed by the Billions

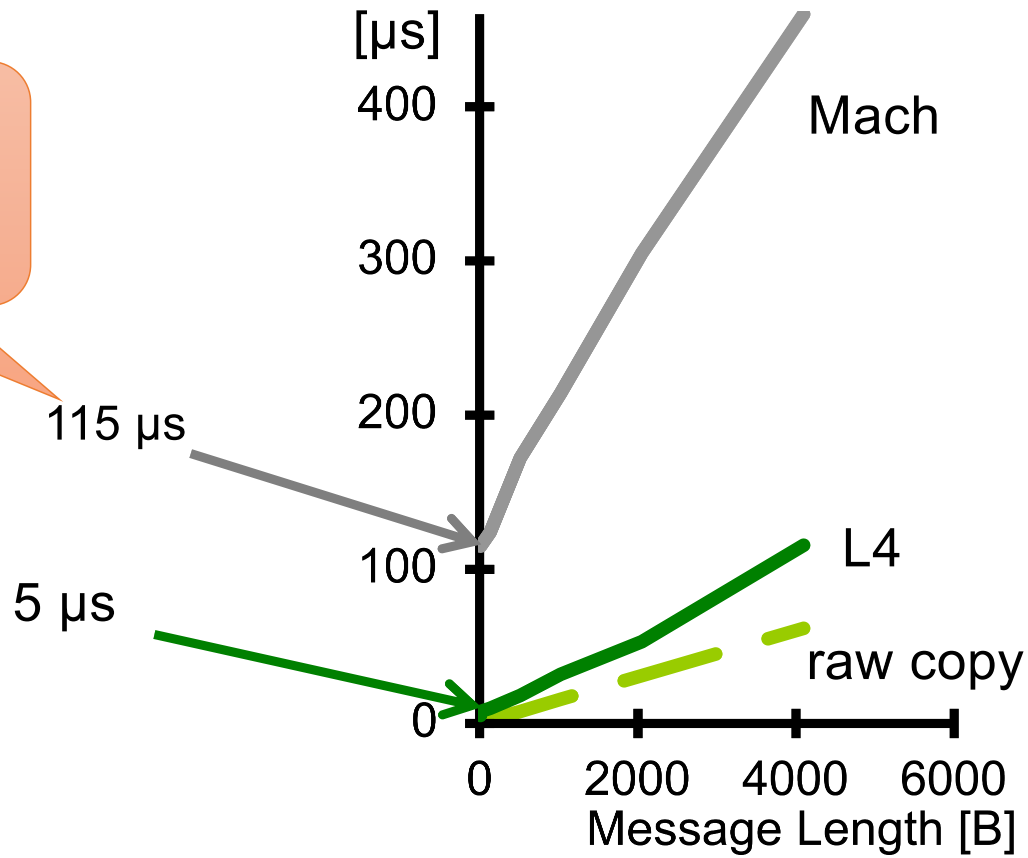
Today's Lecture

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

L4: The Quest for a Real Microkernel

1993 “Microkernel”: IPC Performance

Culprit:
Cache footprint
[Liedtke’95]

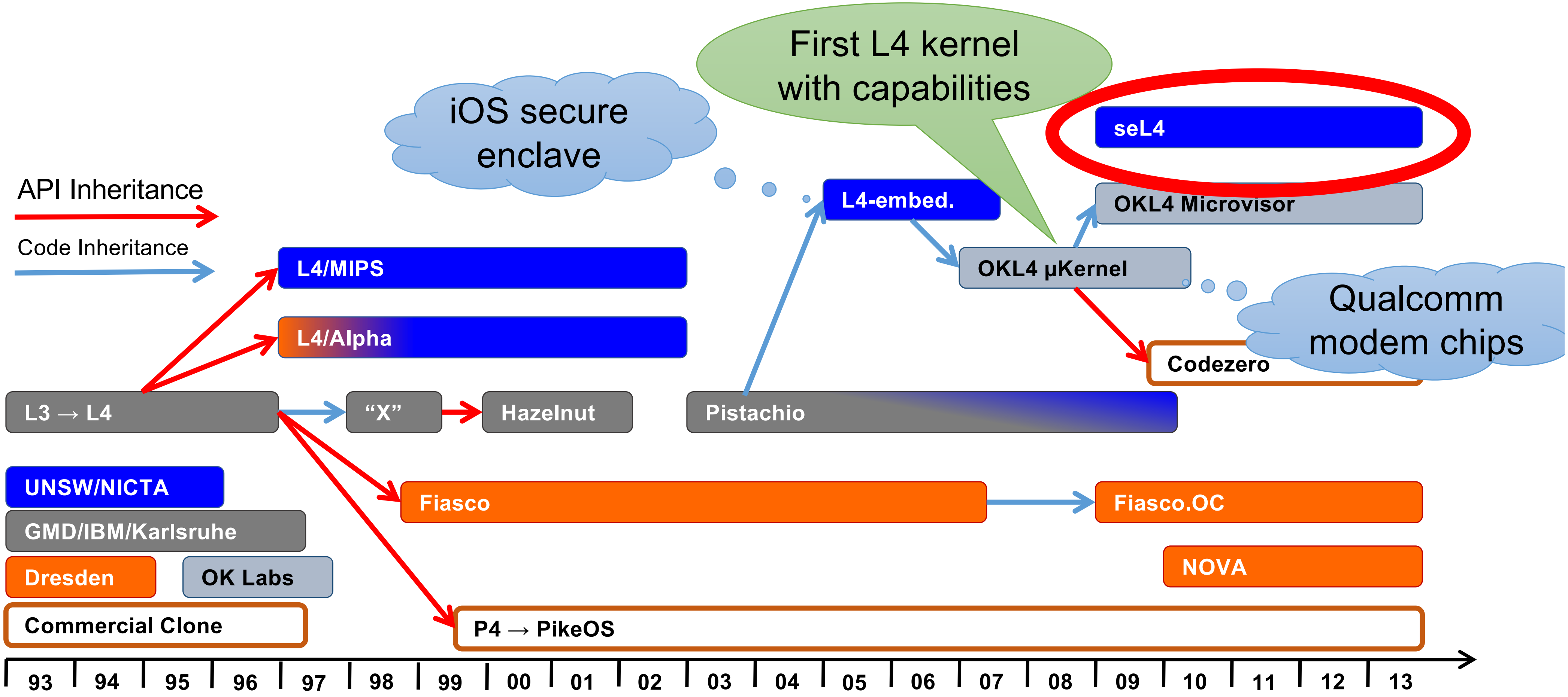


i486 @
50 MHz

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]

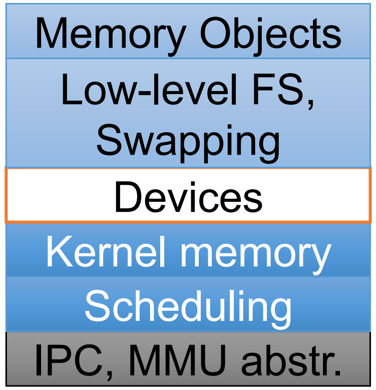
L4: 25 Years High Performance Microkernels



Microkernel Evolution

First generation

Mach ['87], Chorus

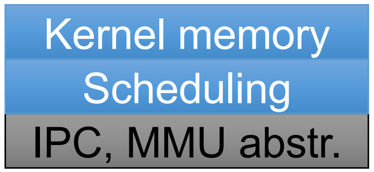


180 syscalls, 100 kSLOC

100 μs IPC

Second generation

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



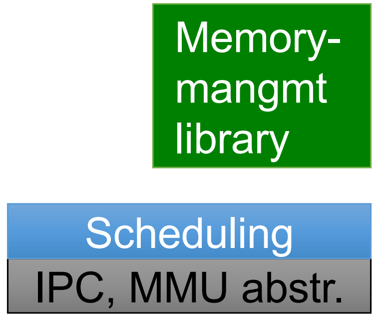
~ 7 syscalls, ~ 10 kSLOC

~ 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

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

Implementation Highlights

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

Direct process switch:
- no scheduler invocation
- SC donation

Wait to receive

Running

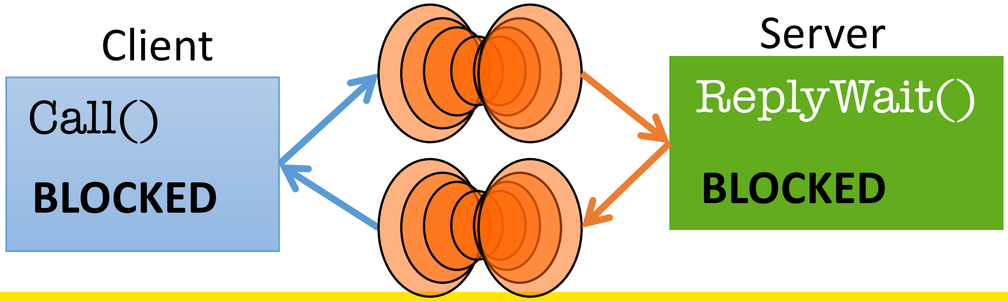
L4 Scheduler Optimisation: Lazy Scheduling

```
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



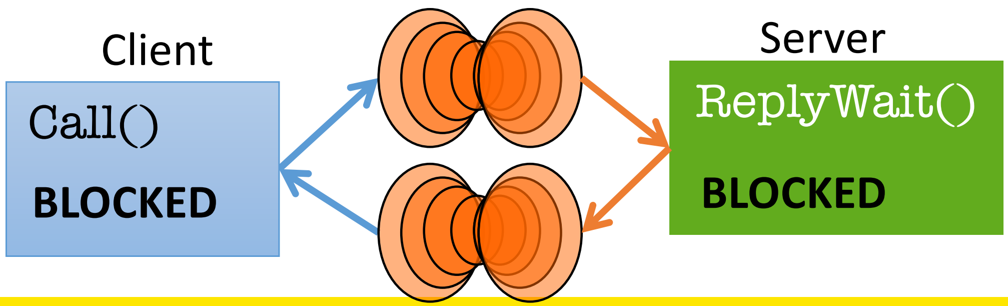
Scheduler: Benno Scheduling

```
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

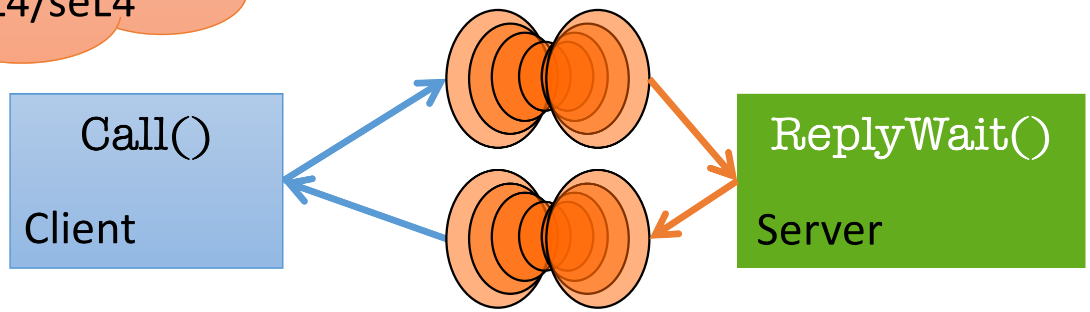
Idea: Don’t invoke scheduler if you know who’ll be chosen

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

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()



Fastpath Coding Tricks

Common case: 0

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

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

UNSW SYDNEY

How About Real-Time Support?

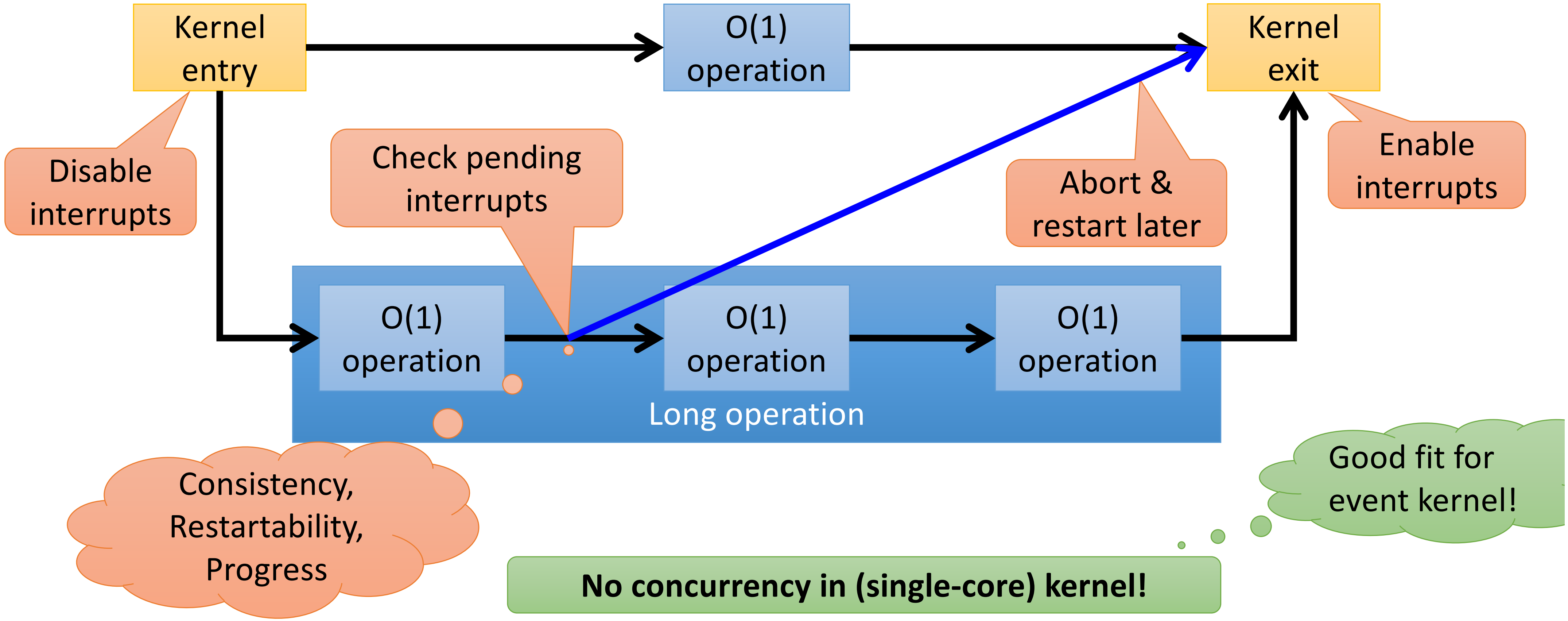
- Kernel runs with interrupts disabled
  - No concurrency control ⇒ simpler kernel
    - Easier reasoning about correctness
    - Better average-case performance

How about long-running system calls?

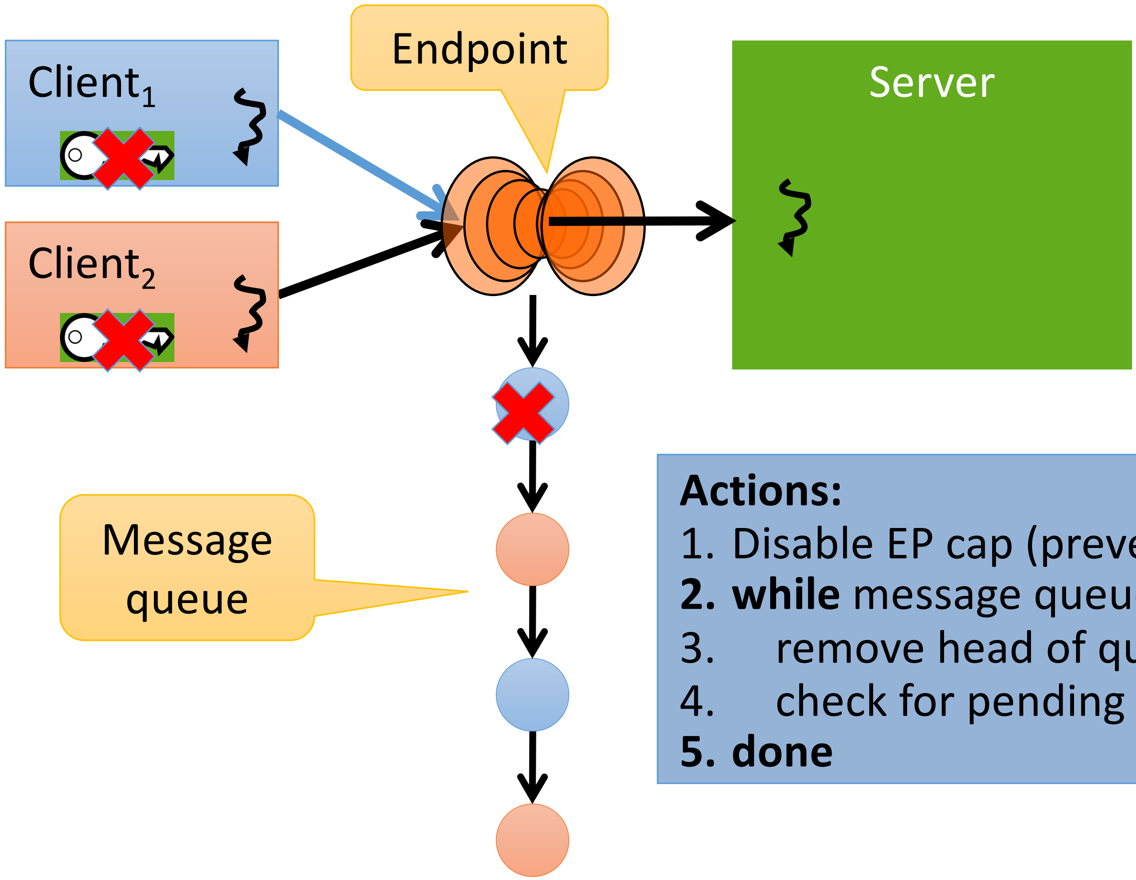
Most protected-mode RTOSes are mostly/fully preemptible

Lots of concurrency in kernel!

Incremental Consistency Paradigm

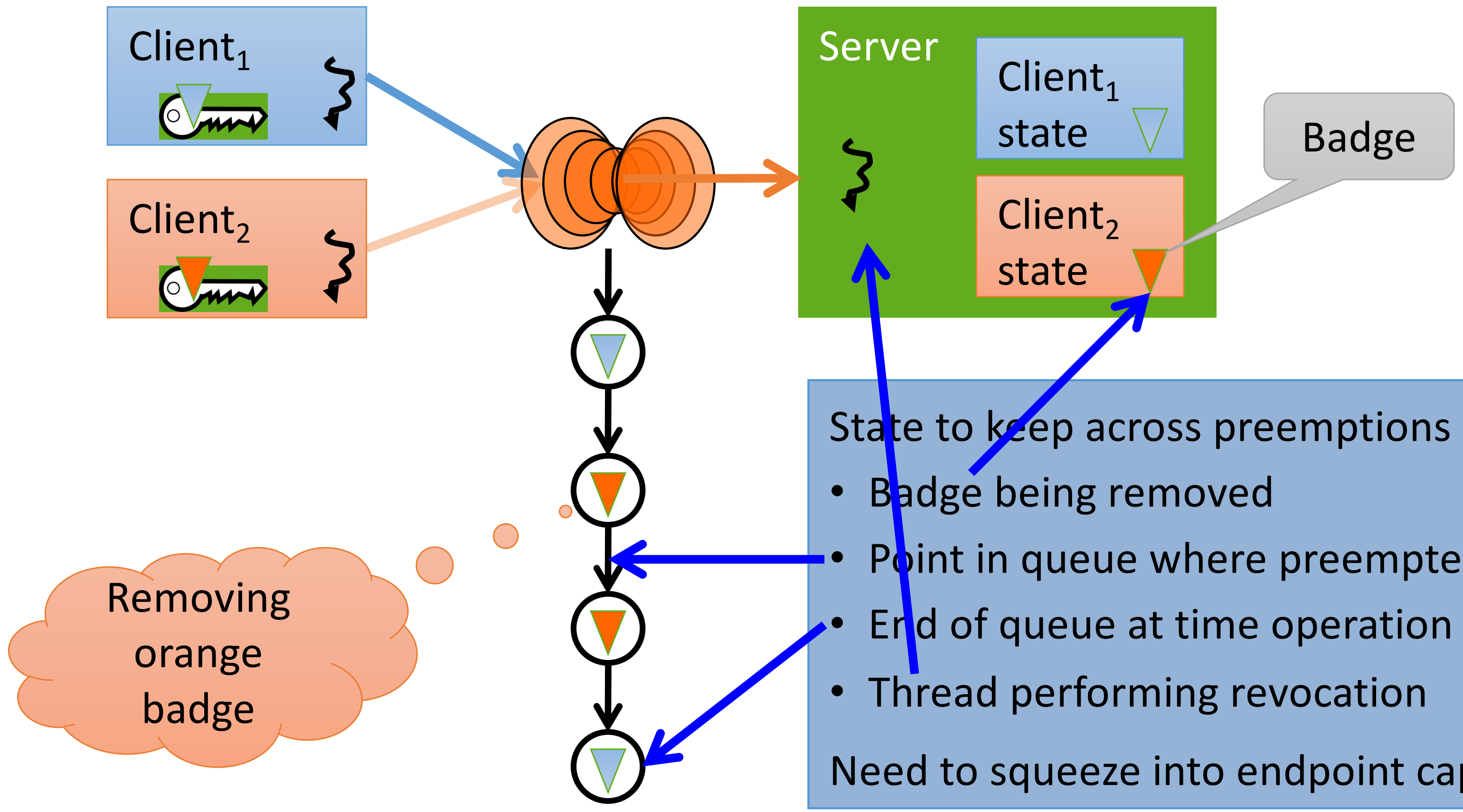


Example: Destroying IPC Endpoint



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



State to keep across preemptions

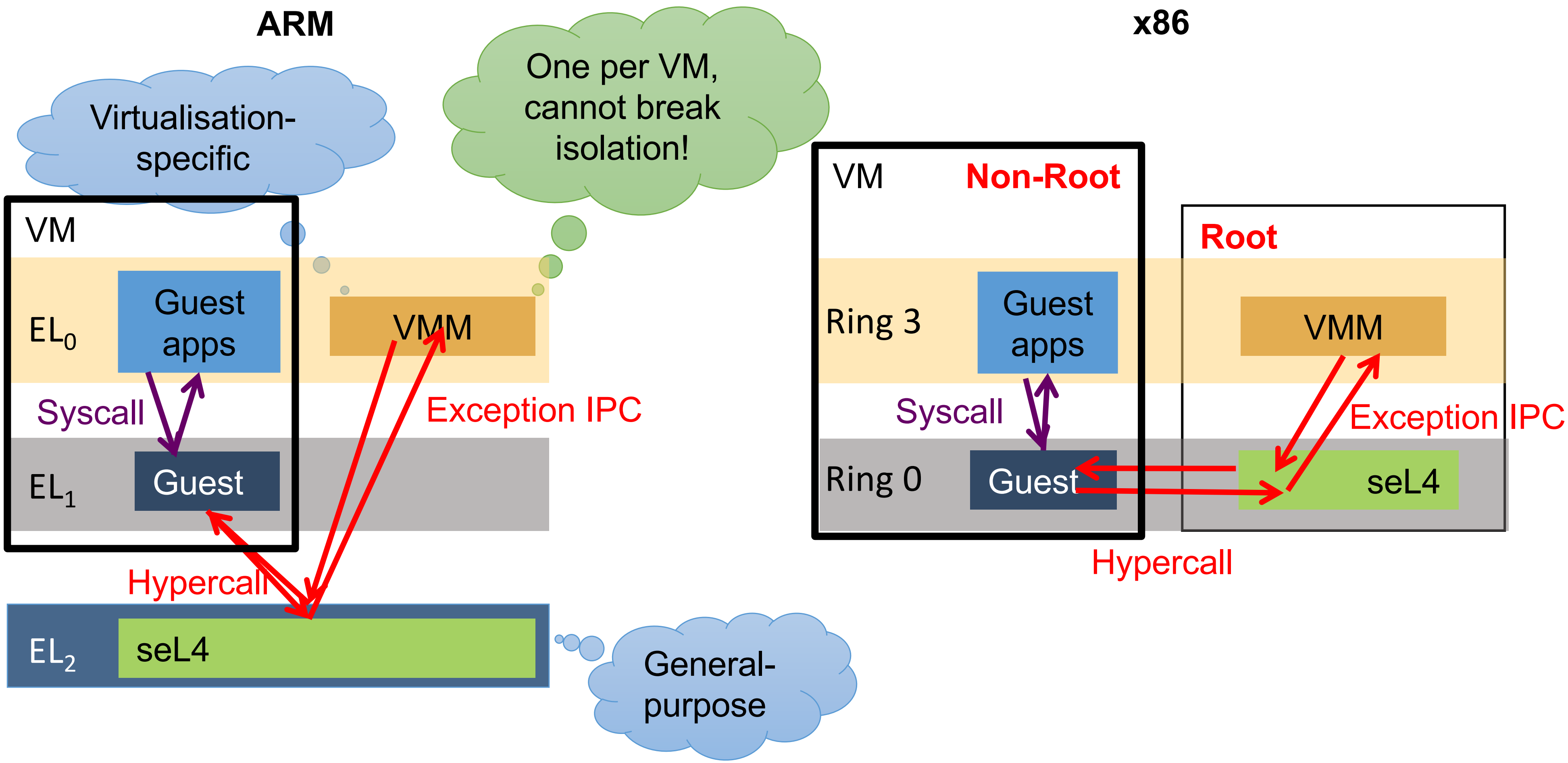
- Badge being removed
- Point in queue where preempted
- End of queue at time operation started
- Thread performing revocation

Need to squeeze into endpoint cap!

Note: not yet in mainline!

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
- I/O
- Communication

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

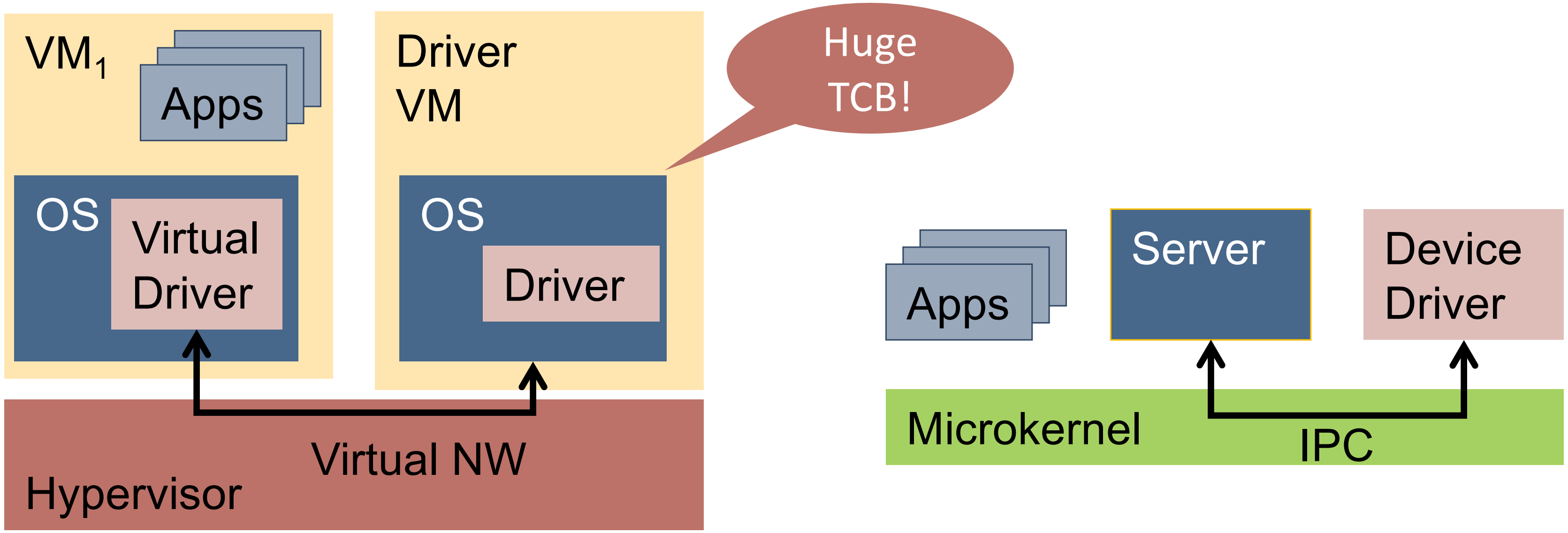
Real Difference?

Minimal overhead, Custom API

Modelled on HW, Re-uses SW

- Similar abstractions
- Optimised for different use cases

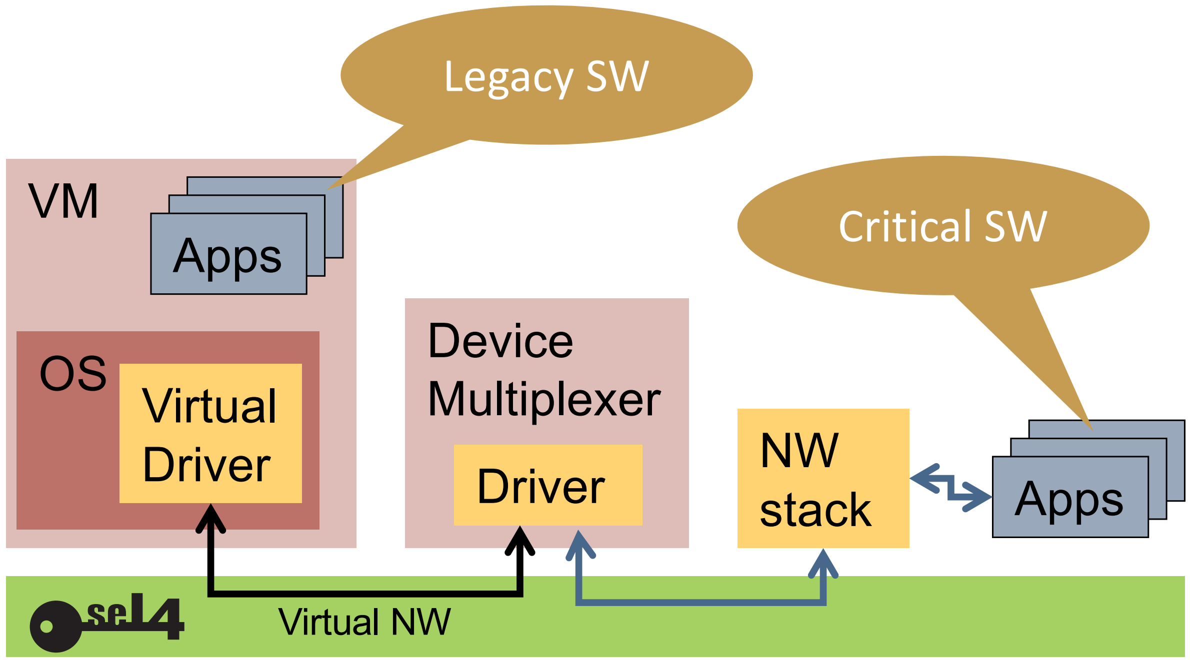
Closer Look at I/O and Communication



Communication is critical for I/O

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

Integration: VMs and Native



- Typical configuration in embedded systems
- Supports "incremental cyber retrofit"

Lessons & Principles

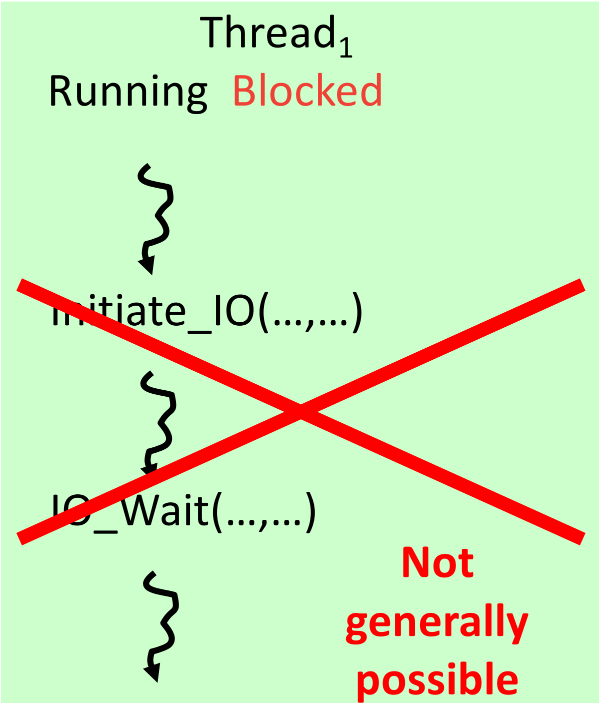
Reflecting on Lessons of 2nd Generation

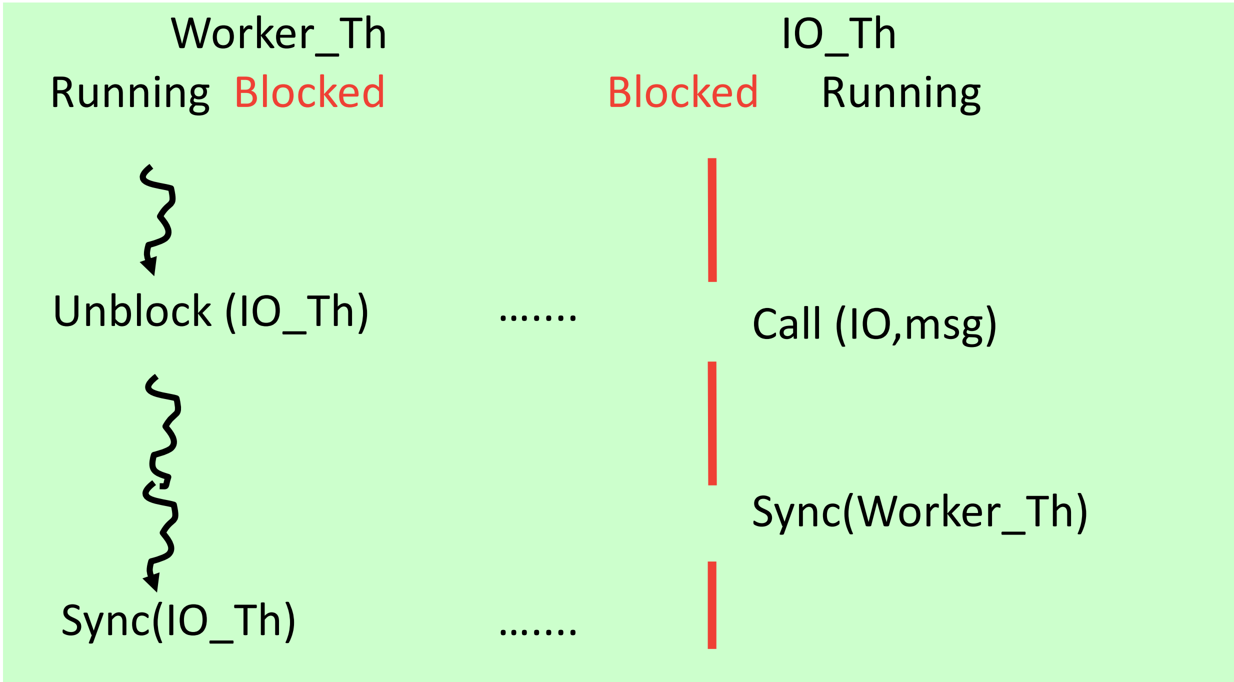
Original L4 design had two major shortcomings:

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

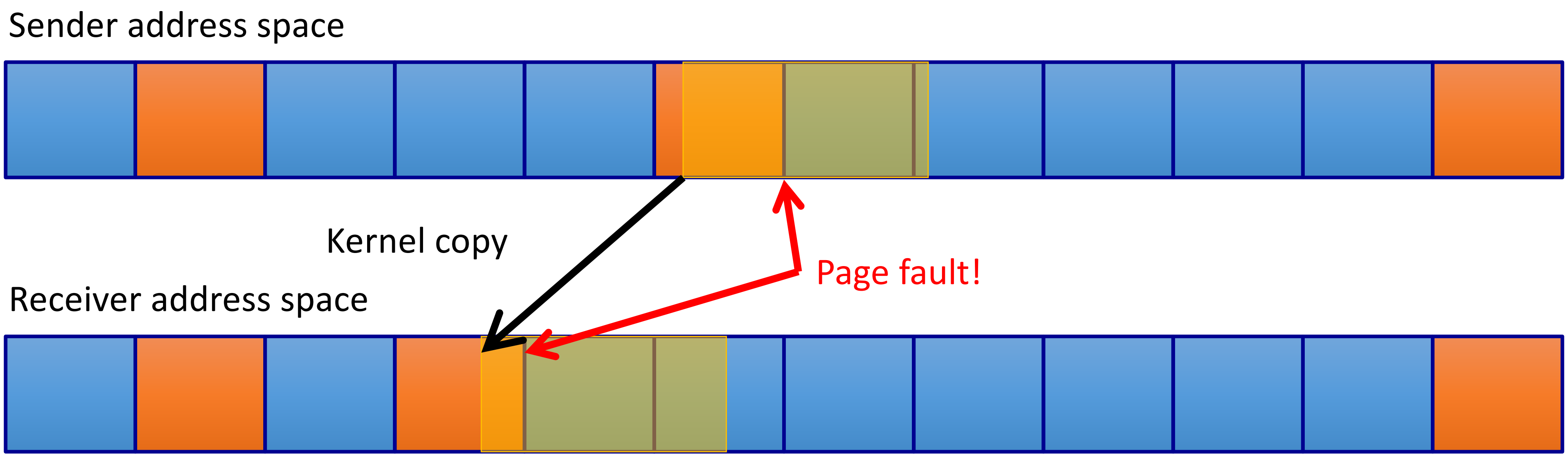
- Sync IPC forces multi-threaded code or select()!
- Also poor choice for multi-core





L4 “Long” IPC

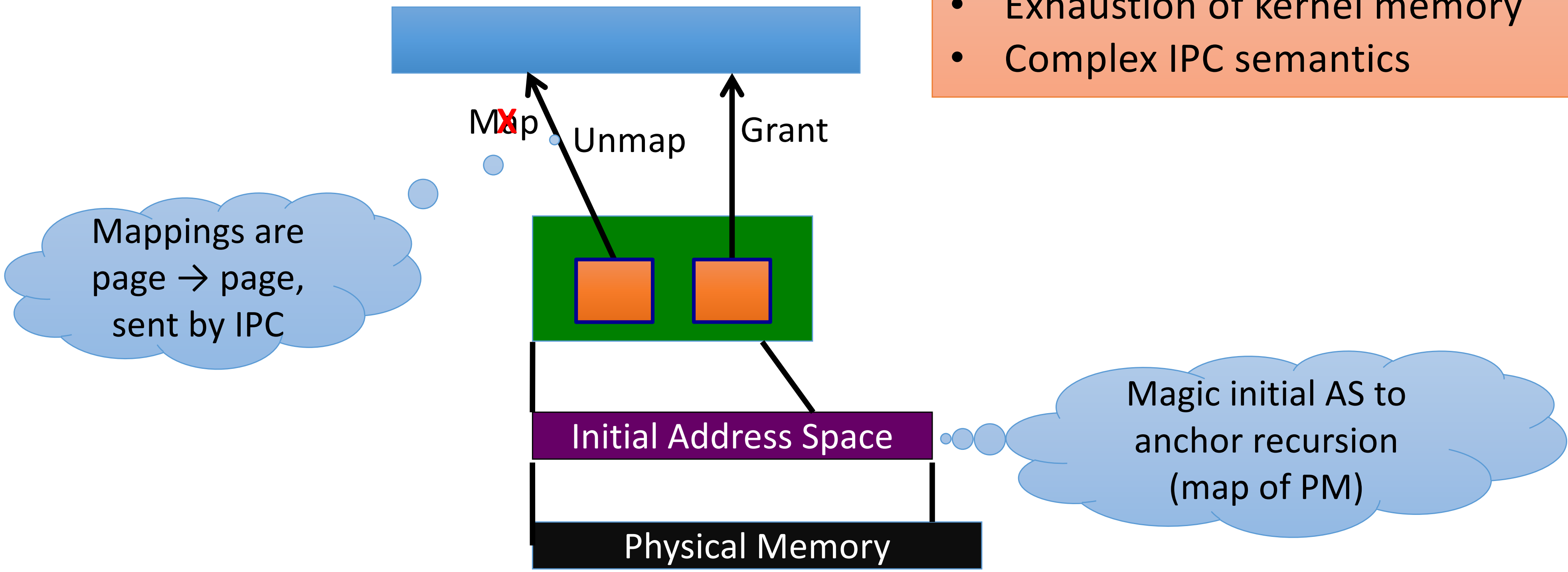
- Not minimal
- Source of kernel complexity:
  - nested exceptions
  - concurrency in kernel
  - must upcall PF handlers during IPC
  - timeouts to prevent DOS attacks



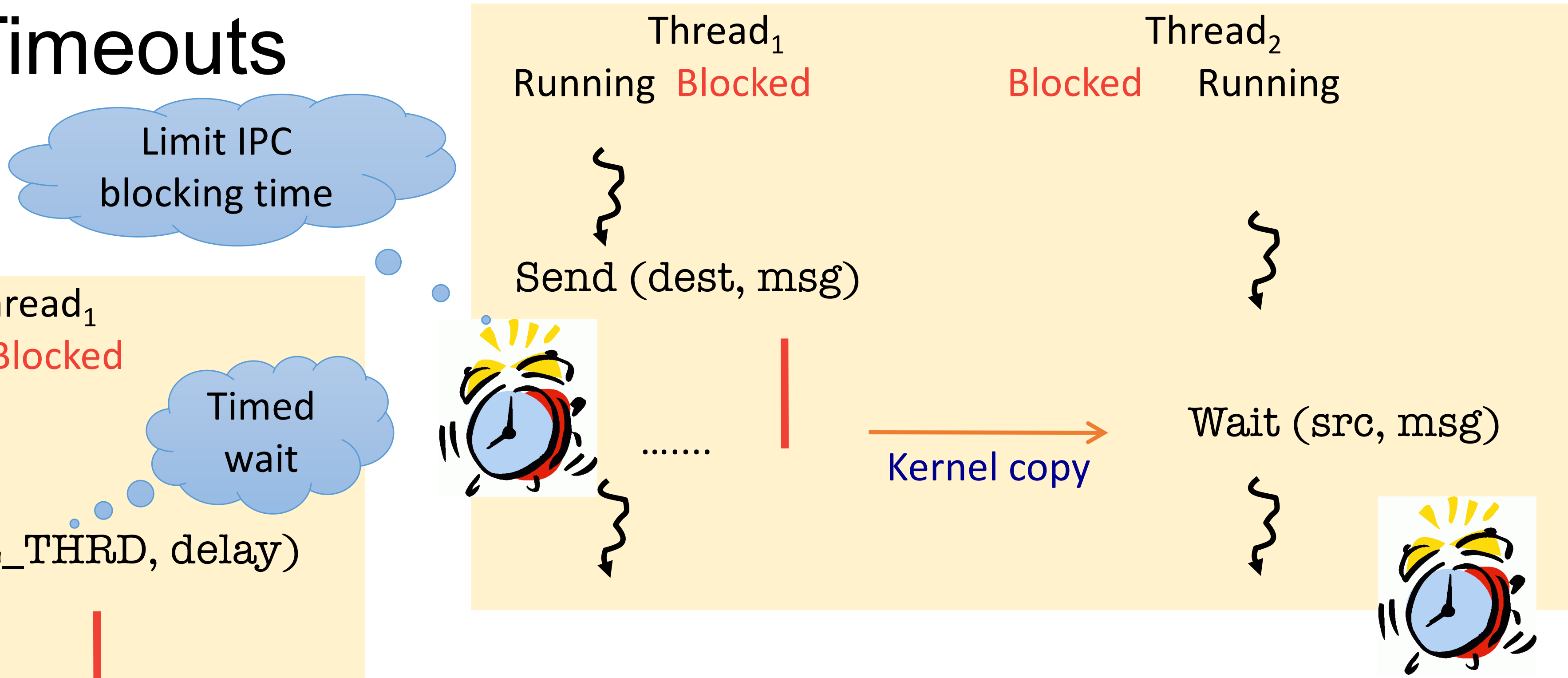
Traditional L4: Recursive Address Spaces

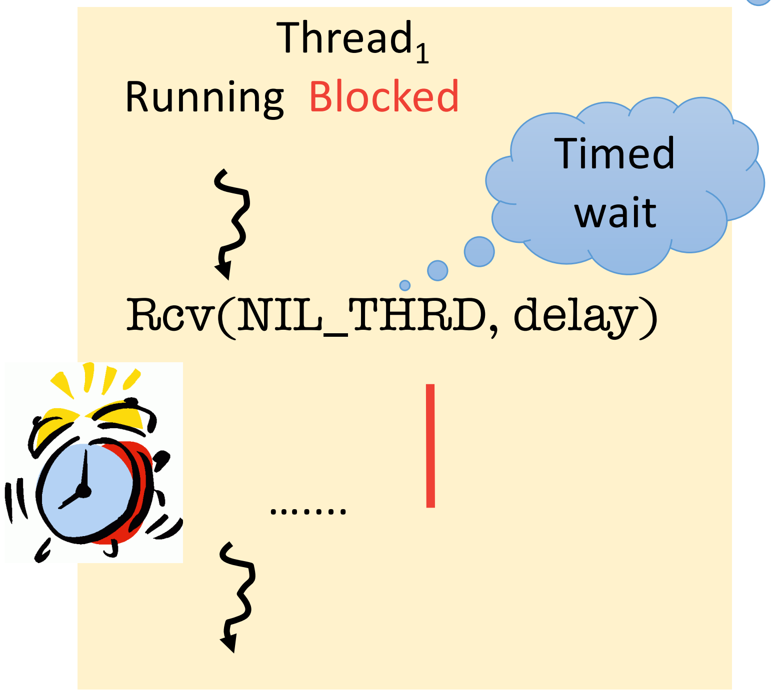
Issues:

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



L4 Timeouts





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

seL4 Design Principles

- Fully delegable 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!

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

August 2009

A NICTA bejelentette a világ első, formális módszerekkel igazolt,

Slashdot

Stories Recent Popular Search

Slashdot is powered by your subm

Technology: World's Firs

Posted by Soulskill on Thursday Aug
from the wait-for-it dept.

An anonymous reader writes

> "Operating systems usually have
> and so forth are known by almos
> to prove that a particular OS ker
> formally verified, and as such it
> researchers used an executable
> the Isabelle theorem prover to ge
> matches the executable and the

Does it run Linux? "We're pleased to say that it does. Presently, we have a para-virtualized ver

eredményeképpen pedig egy olyan megbízhatóságot kapnak a szoftvertől, amely e

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

UNSW SYDNEY