NICTA

UNSW
THE UNIVERSITY OF NEW SOUTH WALES

COMP9242 Advanced Operating Systems

S2/2013 Week 4: Microkernel Design

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

Australian National University
UNSW THE UNIVERSITY OF NEW SOUTH WALES
NSW GOVERNMENT
ACT GOVERNMENT
State Government Victoria
THE UNIVERSITY OF MELBOURNE
THE UNIVERSITY OF SYDNEY
Queensland Government
Griffith UNIVERSITY
QUT Queensland University of Technology
THE UNIVERSITY OF QUEENSLAND AUSTRALIA

Copyright Notice

Microkernel Principles: Minimality

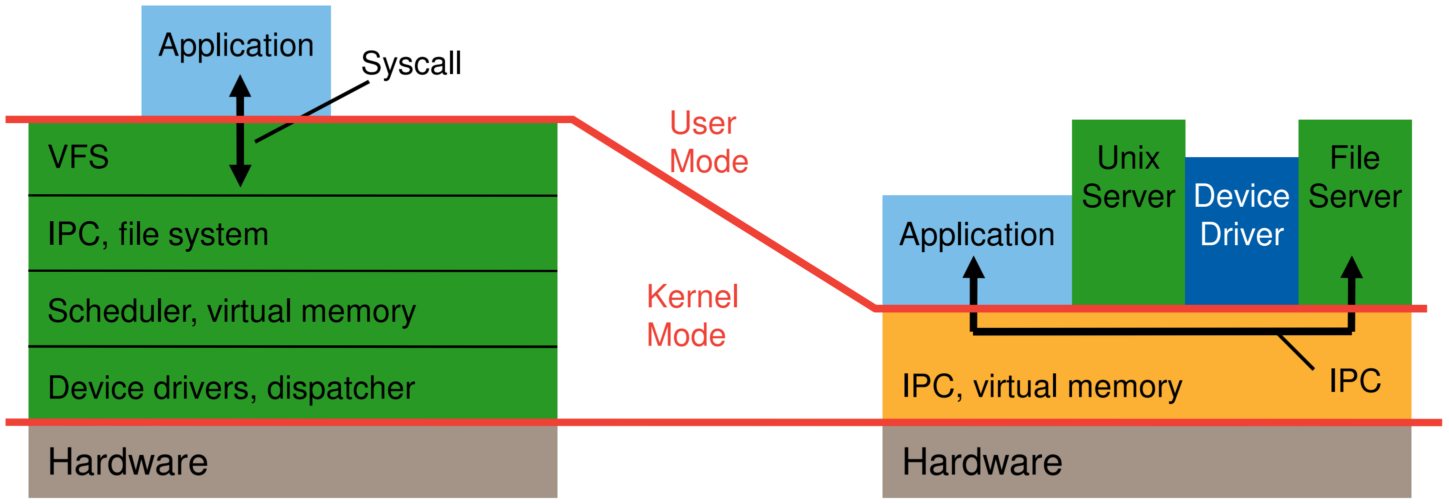
A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

- Advantages of resulting small kernel:
  - Easy to implement, port?
  - Easier to optimise
  - Hopefully enables a minimal *trusted computing base* (TCB)
  - Easier debug, maybe even *prove* correct?
- Challenges:
  - API design: generality despite small code base
  - Kernel design and implementation for high performance

Limited by arch-specific micro-optimisations

Small attack surface, fewer failure modes

Consequence of Minimality: User-level Services



- Kernel provides no services, only mechanisms
- Strongly dependent on fast IPC and exception handling

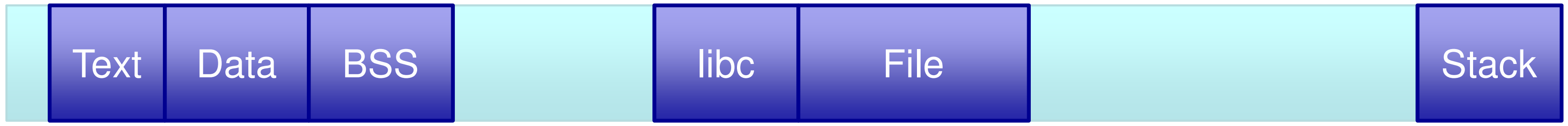
Microkernel Principles: Policy Freedom

Consequence of generality and minimality requirements:

A true microkernel must be free of policy!

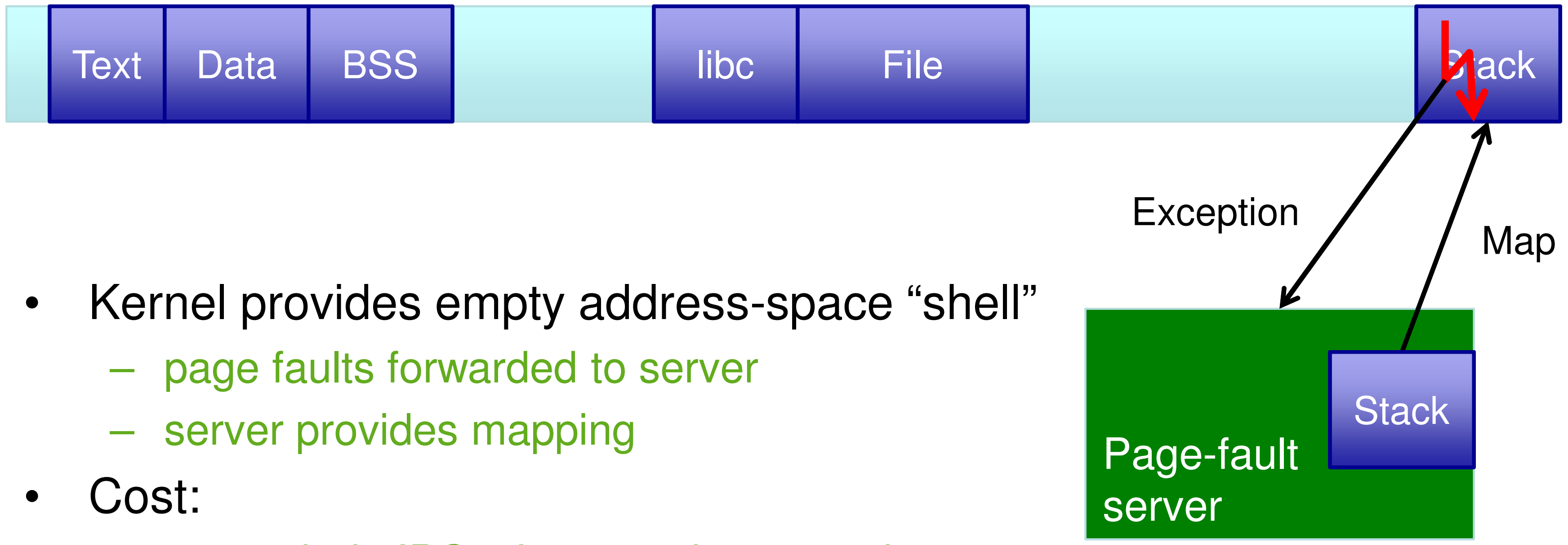
- Policies limit
  - May be good for many cases, but always bad for some
  - Example: disk pre-fetching
- Attempts to make policies general lead to bloat
  - Implementing combination of policies
  - Try to determine most appropriate one at run-time

Policy Example: Address-Space Layout



- Kernel determines layout, knows executable format, allocates stack
  - limits ability to import from other OSes
  - cannot change layout
    - small non-overlapping address spaces beneficial on some archs
  - kernel loads apps, sets up mappings, allocates stack
    - requires file system in kernel or interfaced to kernel
    - bookkeeping for revokation & resource management
    - heavyweight processes
  - memory-mapped file API

Policy-Free Address-Space Management



- Kernel provides empty address-space "shell"
  - page faults forwarded to server
  - server provides mapping
- Cost:
  - 1 round-trip IPC, plus mapping operation
    - mapping may be side effect of IPC
    - kernel may expose data structure
  - kernel mechanism for forwarding page-fault exception
- "External pagers" first appeared in Mach [Rashid et al, '88]
  - … but were optional – in L4 there's no alternative

What Mechanisms?

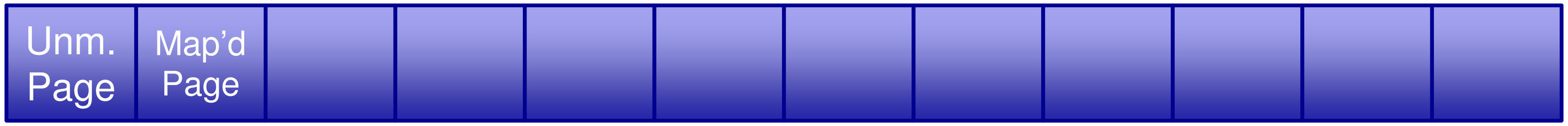
- Fundamentally, the microkernel must abstract
  - *Physical memory*
  - *CPU*
  - *Interrupts/Exceptions*
- Unfettered access to any of these bypasses security
  - No further abstraction needed for devices
    - memory-mapping device registers and interrupt abstraction suffices
    - …but some generalised memory abstraction needed for I/O space
- Above isolates execution units, hence microkernel must also provide
  - *Communication* (traditionally referred to as *IPC*)
  - *Synchronization*

What Mechanisms?

Traditional hypervisor vs microkernel abstractions

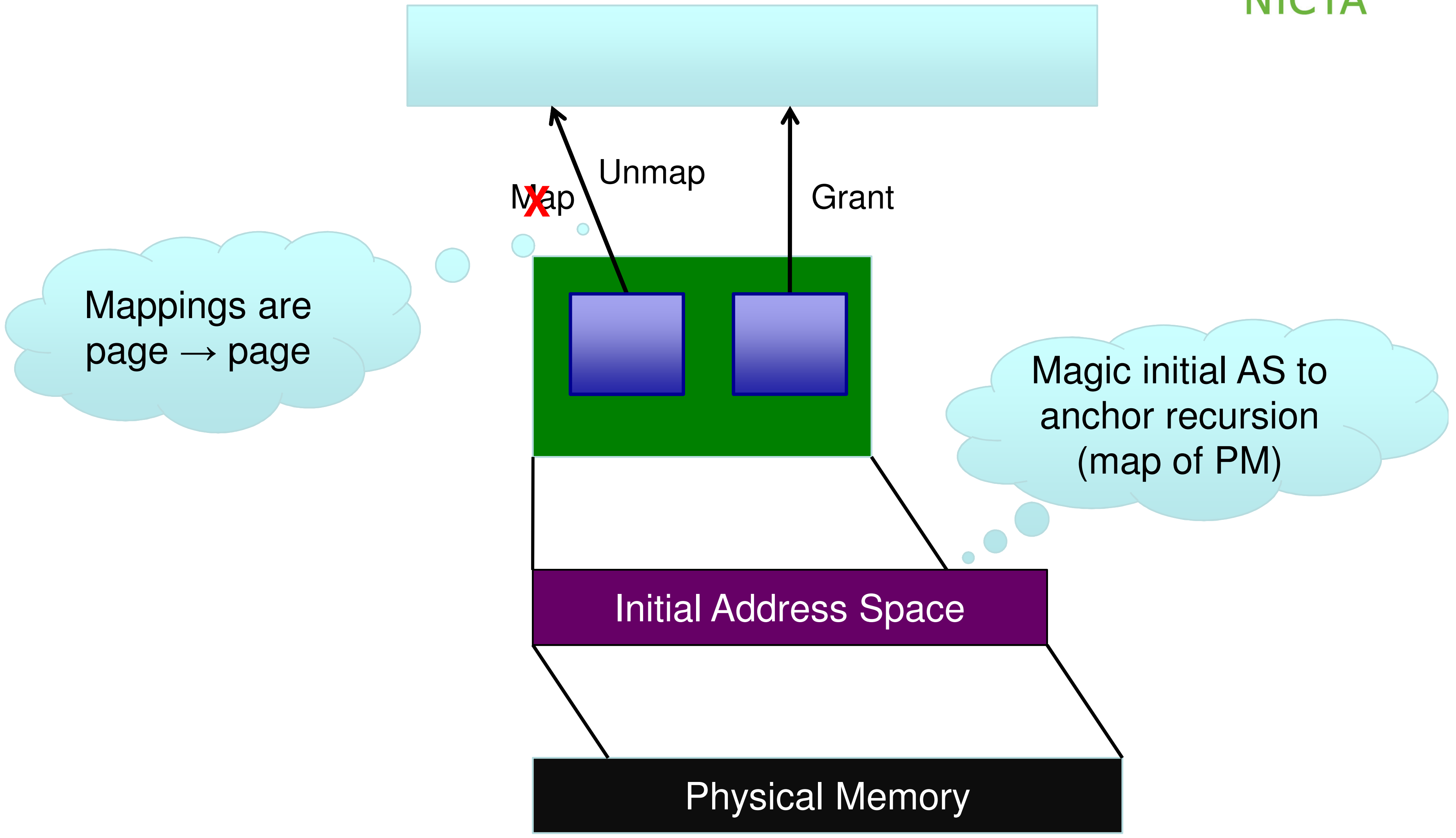
Resource	**Hypervisor**	**Microkernel**
Memory	Virtual MMU (vMMU)	Address space
CPU	Virtual CPU (vCPU)	Thread or scheduler activation
Interrupt	Virtual IRQ (vIRQ)	IPC message or signal
Communication	Virtual NIC	Message-passing IPC
Synchronization	Virtual IRQ	IPC message

Abstracting Memory: Address Spaces



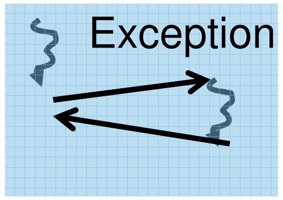
- Minimum address-space abstraction: empty slots for page mappings
  - paging server can fill with mappings
    - virtual address → physical address + permissions
- Can be
  - page-table–like: array under full user control
  - TLB-like: cache for mappings which may vanish
- Main design decision: is source of a mapping a page or a frame?
  - Frame: hardware-like
  - Page: recursive address spaces (original L4 model)

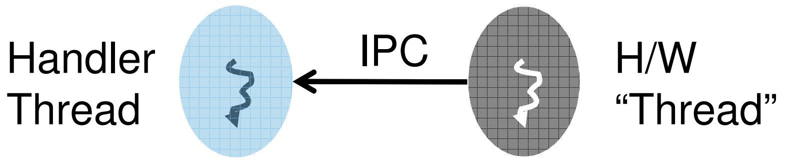
Traditional L4: Recursive Address Spaces



Abstracting Interrupts and Exceptions

- Can abstract as:
  - Upcall to interrupt/exception handler
    - hardware-like diversion of execution
    - need to save enough state to continue interrupted execution
  - IPC message to handler from magic “hardware thread”
    - OS-like
    - needs separate handler thread ready to receive

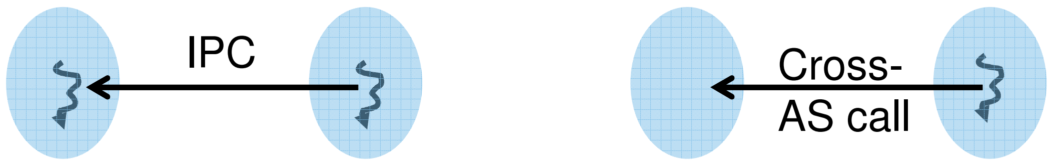




- Page fault tends to be special-cased for practical reason
  - Tends to require handling external to faulter
    - IPC message to page-fault server rather than exception handler
  - But also “self-paging” as in Nemesis [Hand ’99] or Barrelfish

Abstracting Execution

- Can abstract as:
  - kernel-scheduled threads
    - Forces (scheduling) policy into the kernel
  - vCPUs or scheduler activations
    - This essentially virtualizes the timer interrupt through upcall
      - Scheduler activations also upcall for exceptions, blocking etc
    - Multiple vCPUs only for real multiprocessing
- Threads can be tied to address space or "migrating"



- Tight integration/interdependence with IPC model!

Communication Abstraction (IPC)

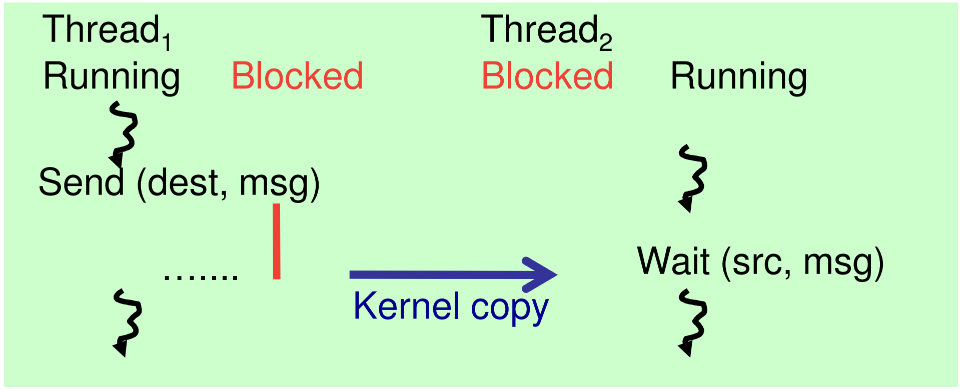
Sender: send (dest, msg)

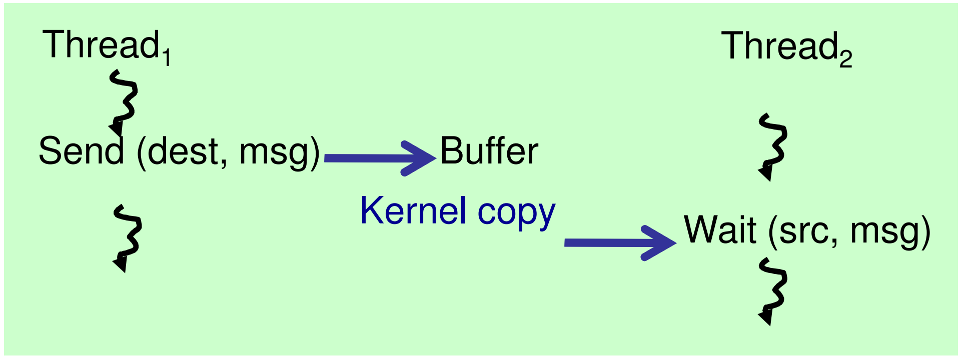
Receiver: receive (src, msg)

- Seems simple, but requires several major design decisions
  - Does the sender block if the receiver isn't ready?
  - Does the receiver block if there is no message
  - Is the message format/size fixed or variable?
  - Do "dest", "src" refer to active (thread) or passive (mailbox) entities?
  - How is the other party identified?

Blocking vs Non-Blocking IPC

- Blocking send:
  - Forces synchronization (rendez vous) with receiver
    - Doubles as synchonization primitive
  - Requires kernel threads or scheduler activations
    - … else block whole app
- Non-blocking send:
  - Requires buffering
  - Data copied twice
  - Can buffer at receiver, but then can only have single message in transit
- Non-blocking receive requires polling or asynchronous upcall
  - Polling is inefficient, upcall forces concurrency on apps
  - Usually have at least an option to block





Message Size and Location

Fixed- vs variable-size messages:

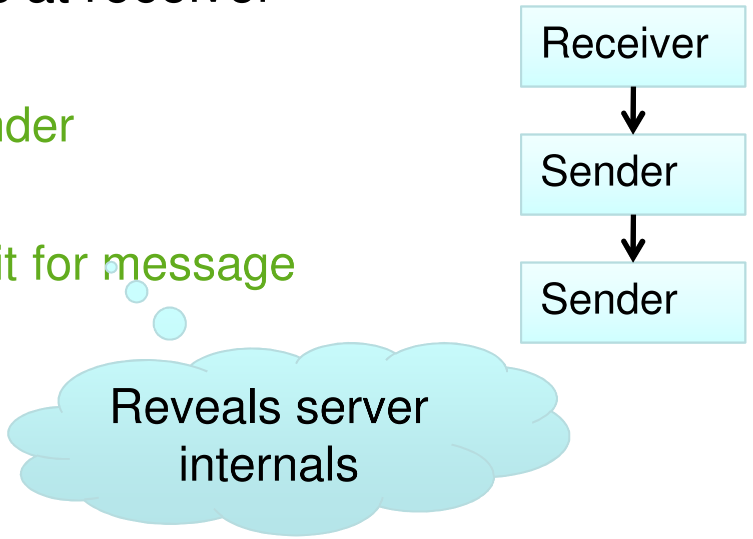
- *Fixed* simplifies buffering and book-keeping
- *Variable* requires receiver to provide big enough buffer
  - Only an issue if messages are very long

Dedicated message buffer vs arbitrary pointer to data:

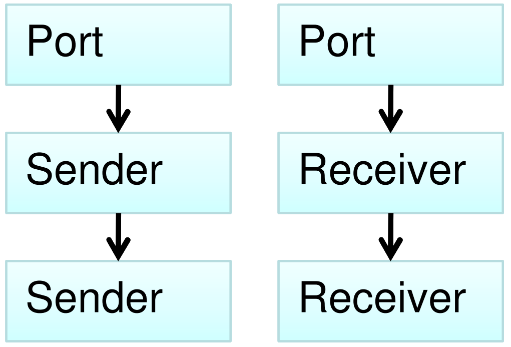
- (Small) dedicated message buffer may be pinned (virtual registers)
- Arbitrary data strings may cause page faults
  - abort IPC?
  - handle fault by invoking pager?

Direct vs Indirect IPC Adressing

- Direct: Queue senders/messages at receiver
  - Need unique thread IDs
  - Kernel guarantees identity of sender
    - useful for authentication
  - Can't have multiple receivers wait for message
    - eg pools of worker threads



- Indirect: Mailbox/port object
  - Just a user-level handle for the kernel-level queue
  - Extra object type – extra weight?
  - Communication partners are anonymous
    - Need separate mechanism for authentication



Typical Approaches

- Asynchronous send plus synchronous receive
  - most convenient for programmers
    - minimises explicit concurrency control at user level
    - generally possible to get away with single-threaded processes
  - main drawback is need for kernel to buffer
    - violates minimality, adds complexity
  - typical for 1st generation microkernels
- Traditional L4 model is totally synchronous
  - Allows very tight implementation
  - Not suitable for manycores
  - *Requires (kernel-scheduled) multi-threaded apps!*
    - Kernel policy on intra-process scheduling!
- OKL4 microvisor IPC is totally asynchronous
  - … but forces one partner to supply buffer
  - synchronization via virtual IRQs

Long communication delays

Avoid double copy!

Brief History of Microkernels

0th Generation: 1970s

- Nucleus [Brinch Hansen ’70]
  - most of the microkernel ideas
  - ahead of its time, not feasible on 1970 hardware
- Hydra [Wulf et al ‘74]
  - policy – mechanism separation
  - hardware-implemented capabilities
  - “object oriented” (before that term existed)
  - too slow for practical use

Brief History of Microkernels

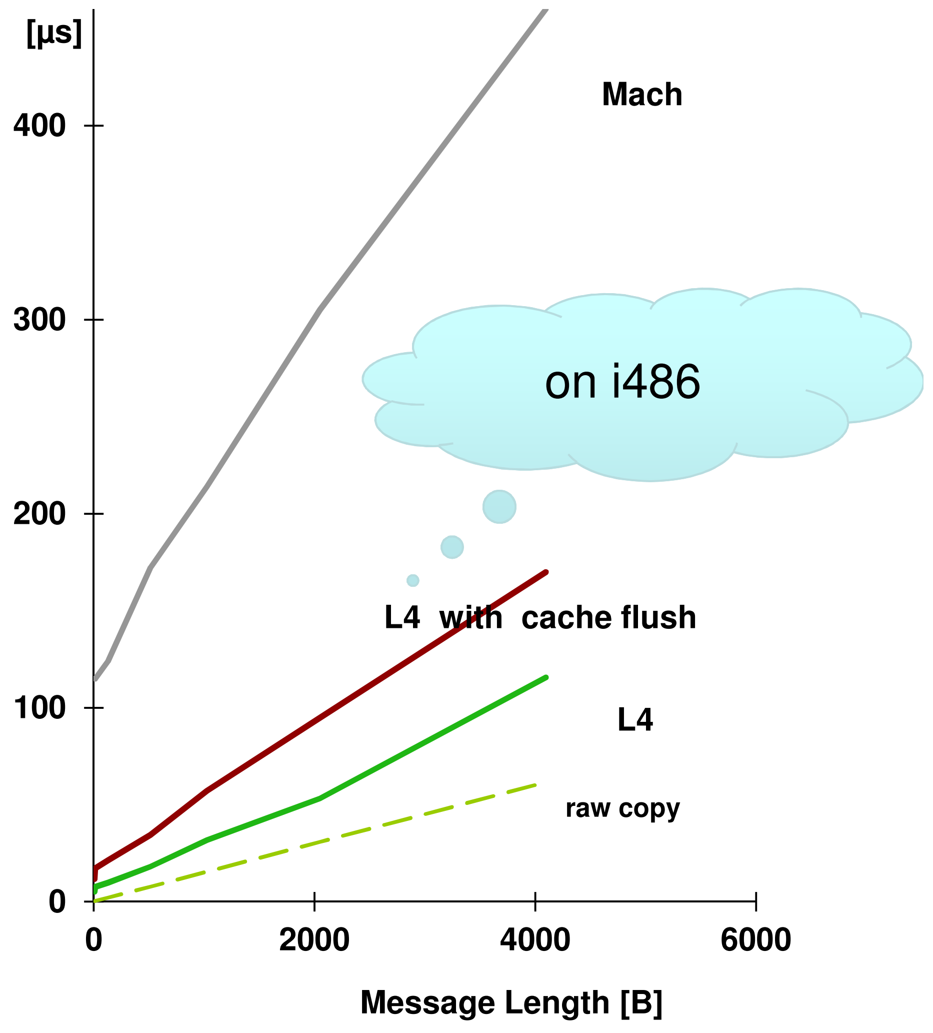
1st Generation: mid-1980 (Mach, Chorus etc)

- Stripped-down monolithic OSes
- Lots of functionality and policy
  - device drivers, low-level file systems, swapping
  - very general, rich and complex IPC
- Big
  - Mach had about 300 kernel APIs, 100s kLOC C
- *Slow: 100 µs IPC*
  - cache footprint shown a major factor in poor performance [Liedtke 95]
  - consequence of IPC complexity, poor design and implementation
    - stripping out stuff from a big blob doesn't produce a good microblob!

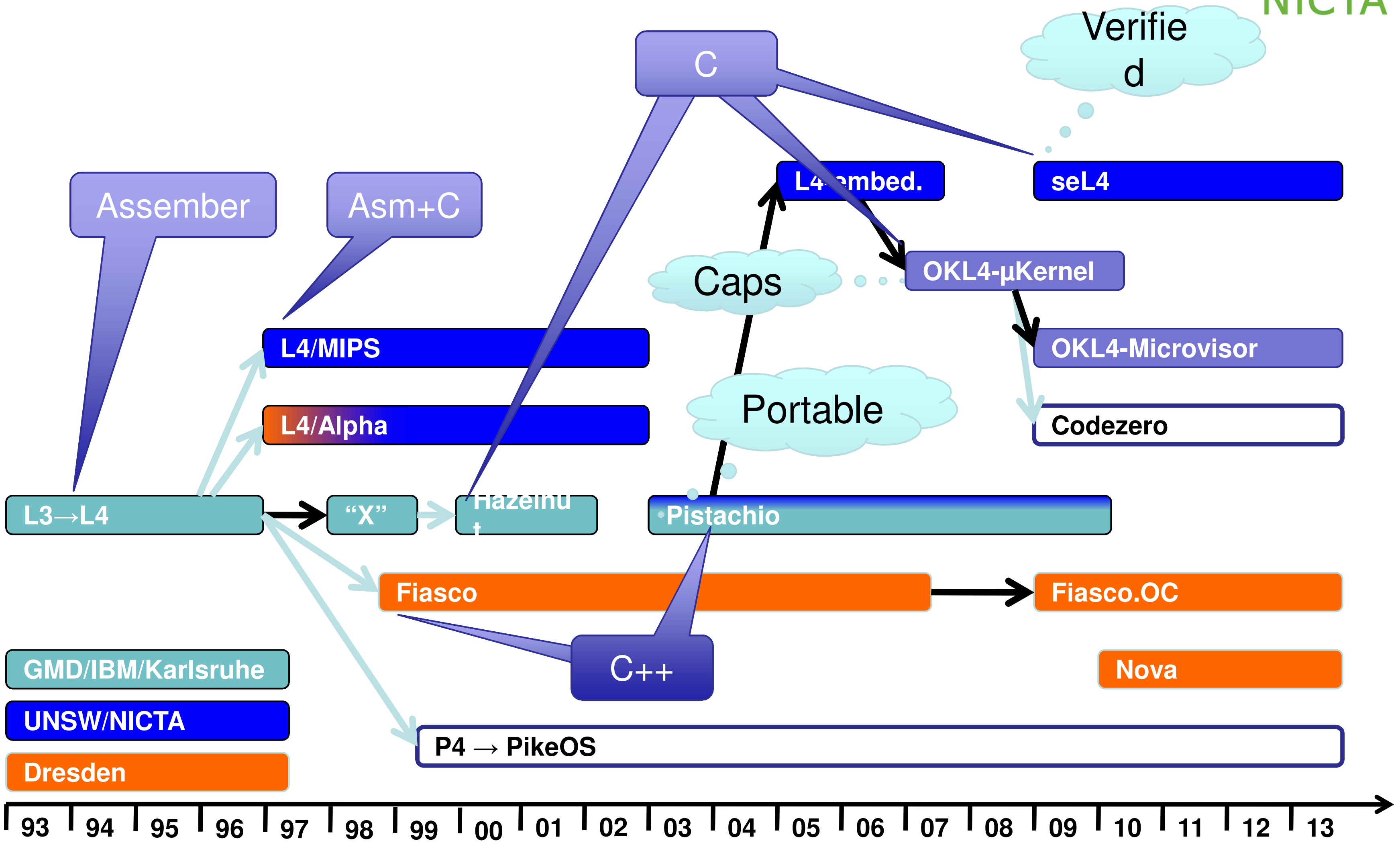
Brief History of Microkernels

2nd Generation: mid-1990s – L4

- "Radical" approach [Liedtke'93, Liedtke '95]:
  - Strict minimality
  - From-scratch design and implementation
- Fast!
- Minimal and orthogonal mechanisms
  - L4 V2 API: 7 system calls plus a few kernel protocols
  - reduced IPC complexity
  - 15 kLOC(?) x86 assembler



L4 Family Tree



NICTA

UNSW THE UNIVERSITY OF NEW SOUTH WALES

Brief History of Microkernels

3rd Generation: seL4 [Elphinstone et al 2007, Klein et al 2009]

- Security-oriented design
  - capability-based access control
  - strong isolation by design
- Hardware resources subject to user-defined policies
  - including kernel memory (no kernel heap)
  - except time ☹
- Designed for *formal verification*

Lessons Learned from 2nd Generation

Micro-optimisation: core feature of L4

- Programming languages:
  - original i496 kernel ['95]: all assembler
  - UNSW MIPS and Alpha kernels ['96,'98]: half assembler, half C
  - Fiasco [TUD '98], Pistachio ['02]: C++ with assembler "fast path"
  - seL4 ['07], OKL4 ['09]: all C

C++ ABANDONED

- Lessons:
  - C++ sux: code bloat, no real benefit
  - Changing calling conventions not worthwhile
    - Conversion cost in library stubs and when entering C in kernel
    - Reduced compiler optimization
  - Assembler unnecessary for performance
    - Can write C so compiler will produce near-optimal code
    - C entry from assembler cheap if calling conventions maintained
    - seL4 performance with C-only pastpath as good as other L4 kernels [Blackham & Heiser '12]

Assembler coding ABANDONED

Lessons Learned from 2nd Generation

Micro-optimisation: core feature of L4

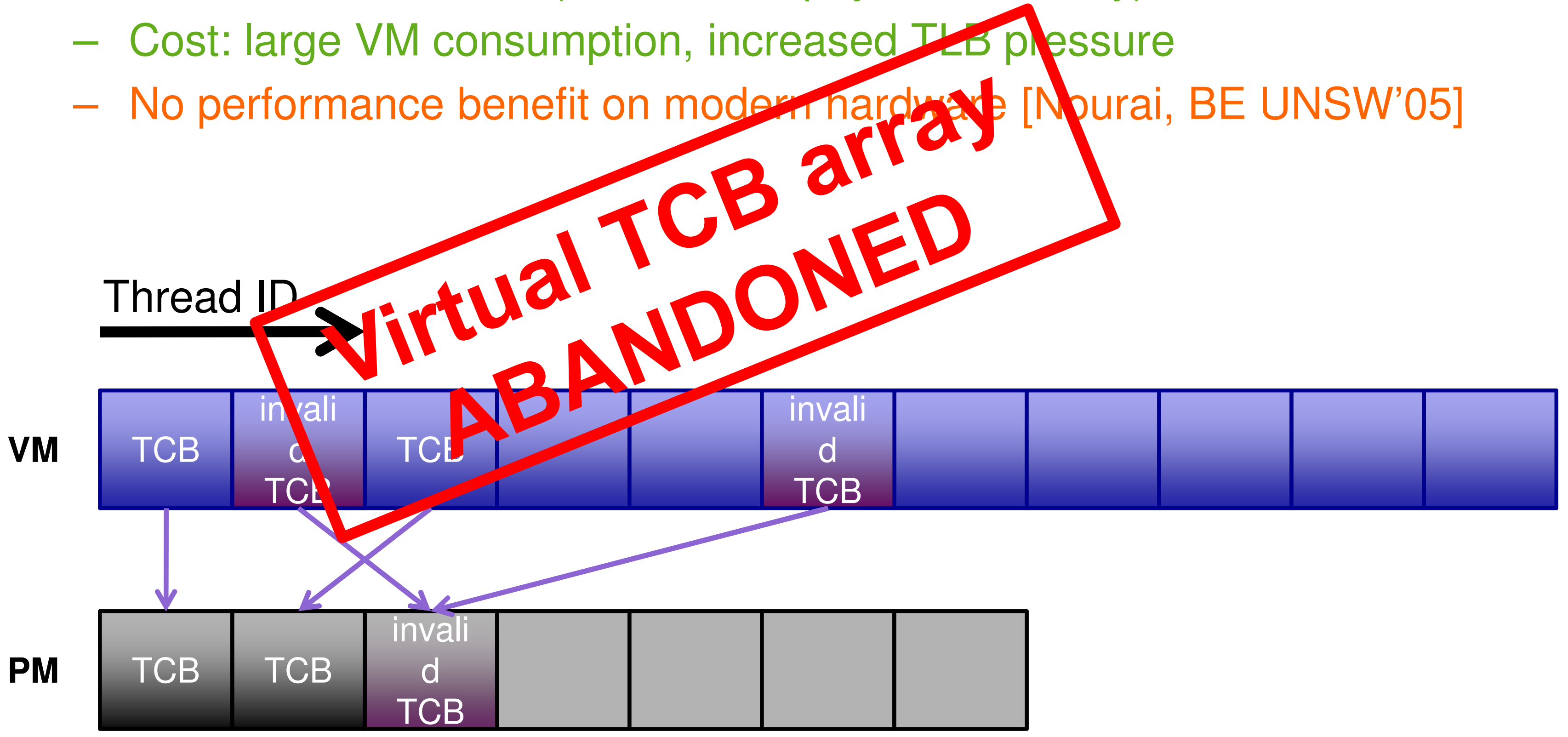
- Liedtke: process-oriented kernel for simplicity and efficiency
  - Per-thread kernel stack, co-located with TCB
    - reduced TLB footprint (i486 had no large pages!)
    - easier to deal with blocking in kernel
  - Cost: high memory overhead
    - about 1/4–1/2 of kernel memory
  - Effectively needed continuations anyway for nested faults
    - page-fault during "long" IPC
  - No performance benefit on modern hardware [Warton, BE UNSW'05]

Per-thread stacks ABANDONED

Lessons Learned from 2nd Generation

Micro-optimisation: core feature of L4

- Liedtke: virtual TCB array for fast lookup from thread ID
  - allocated on demand (no waste of physical memory)
  - Cost: large VM consumption, increased TLB pressure
  - No performance benefit on modern hardware [Nourai, BE UNSW'05]



Lessons Learned from 2nd Generation

NICTA

API complexity still too high

- IPC semantics:
  - In-register, in-line and by-reference message
  - Timeouts on each IPC
  - Mappings created as a side-effect of IPC
- Timeouts: need way to avoid DOS-attacks by blocking partner
  - Timeouts too general: no systematic approach to determine them
  - Significant source of kernel complexity
  - Replaced (in NICTA version) by fail-if-not-ready flag
- Various "long" message forms: complex and rarely used
  - Require handling of in-kernel page faults (during copying)
    - massive source of kernel complexity
  - Replaced (in Pistachio) by pinned message buffers ("virtual registers")
    - essentially retained by seL4

In practice:
zero or infinity!

IPC Timeouts ABANDONED

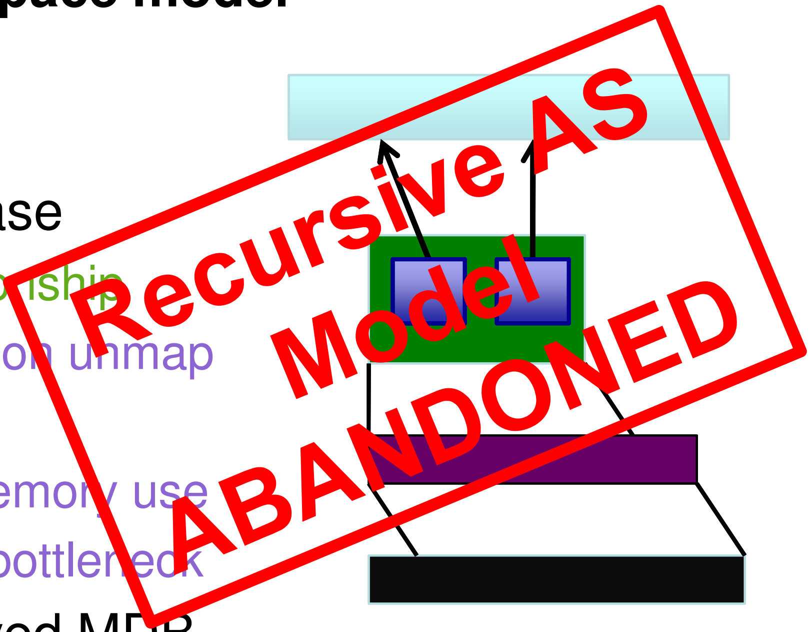
LONG IPC ABANDONED

UNSW THE UNIVERSITY OF NEW SOUTH WALES

Lessons Learned from 2nd Generation

API complexity: Recursive address-space model

- Conceptually elegant
  - trivially supports virtualization
- Drawback: Complex mapping database
  - Kernel needs to track mapping relationship
    - Tear down dependent mappings on unmap
  - Mapping database problems:
    - accounts for 1/4–1/2 of kernel memory use
    - SMP coherence is performance bottleneck
- NICTA's L4-embedded, OKL4 removed MDB
  - Map frames rather than pages
    - need separate abstraction for frames / physical memory
    - subsystems no longer virtualizable (even in OKL4 cap model)
- Properly addressed by seL4's capability-based model
  - But have cap derivation tree, subject of on-going research



Lessons Learned from 2nd Generation

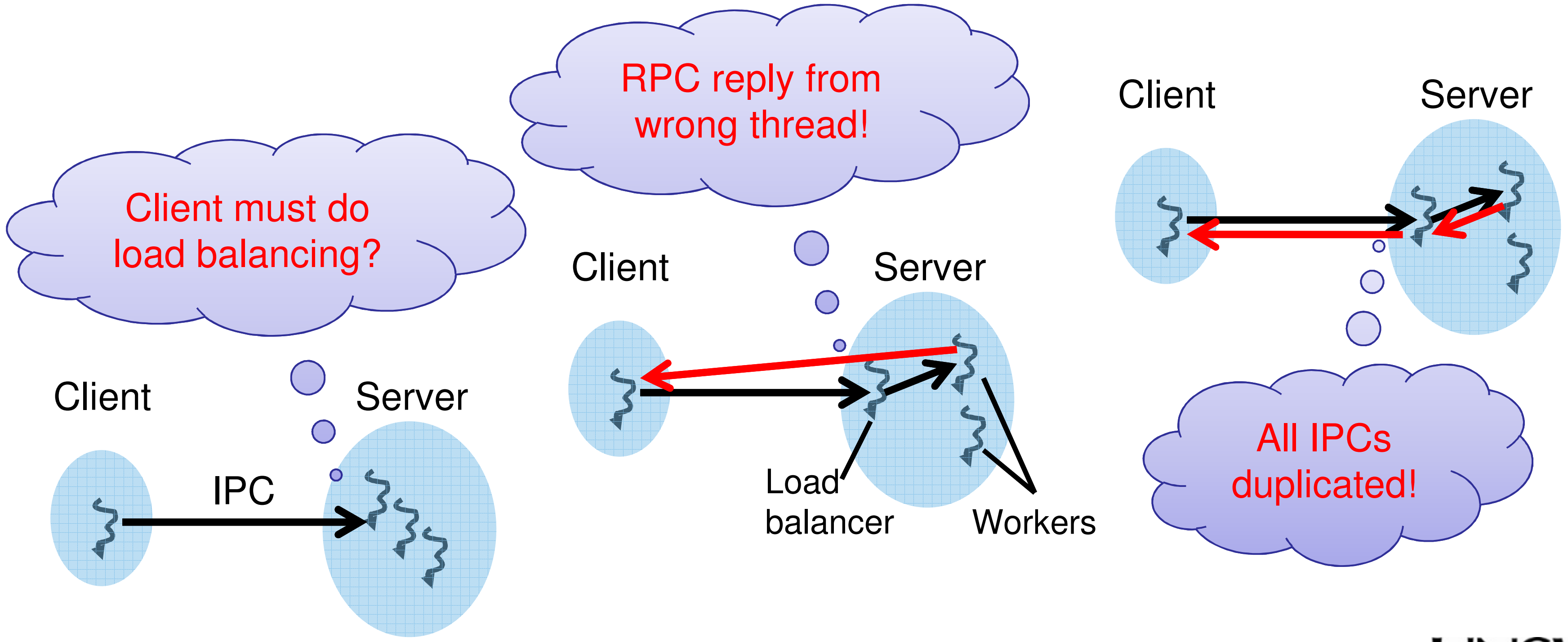
Blocking IPC is not sufficient in practice

- Does not map well to hardware-generated events (interrupts)
  - Many real-world systems are event-driven (especially RT)
  - Mapping to synchronous IPC model requires proliferation of threads
    - Forces explicit concurrency control on user code
    - Made worse by IPC being too expensive for synchronization
- Attempt by Liedtke to address with “user-level” IPC [Liedtke ’01]
  - intra-address-space only
  - thread manipulates partner’s TCB
    - part of thread state kept in user-level TCB (UCTB)
    - caller executes kernel IPC code in user mode
    - inconsistencies fixed up on next kernel entry
  - too messy & limiting in practice
- Introduction of asynchronous notify (L4-embedded) [NICTA ’04]
  - much closer to hardware interrupts
  - OKL4 Microvisor completely discards synchronous IPC

Lessons Learned from 2nd Generation

Access control, naming and resource management

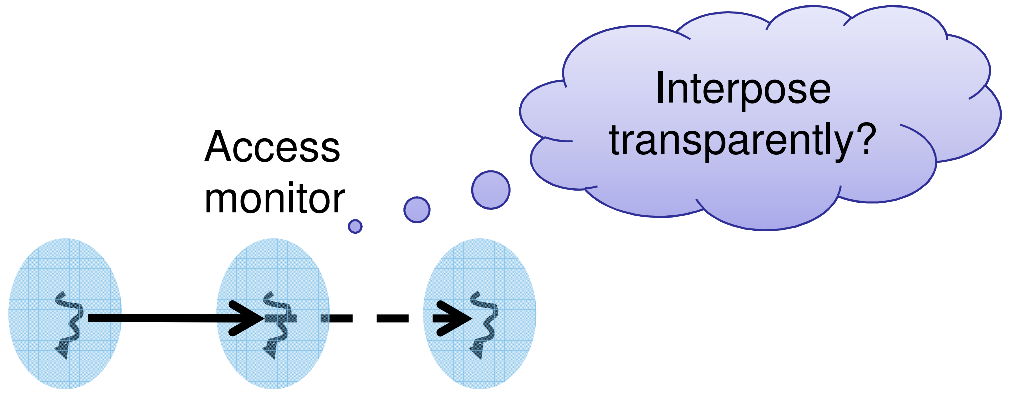
- L4 used global thread IDs to address IPC
  - fast as it avoids indirection via ports or mailboxes
  - inflexible, as server threads need to be externalised (thread pools!)
    - … or messages duplicated
    - various hacks around this were tried, none convinced



Lessons Learned from 2nd Generation

Access control, naming and resource management

- L4 used global thread IDs to address IPC
  - fast as it avoids indirection via ports or mailboxes
  - inflexible, as server threads need to be externalised (thread pools!)
    - … or messages duplicated
    - various hacks around this were tried, none convinced
  - expensive to virtualize, monitor
    - “clans and chiefs” hack doubles message, too expensive in practice
  - global names are a covert channel [Shapiro ’03]
- Need anonymising intermediate message target (endpoints)

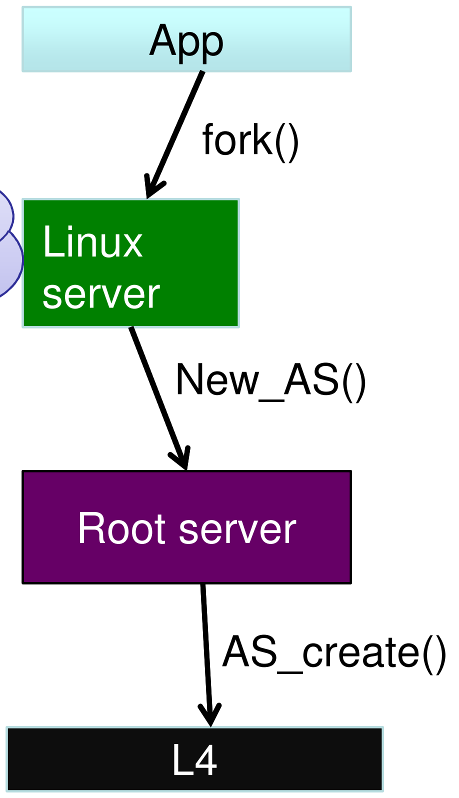


Lessons Learned from 2nd Generation

Access control, naming and resource management

- L4 had no proper model for *rights delegation*
  - Partially due to ad-hoc resource protection approach
- Subsystem could DOS kernel
  - Create mappings until kernel out of memory
  - In V4 addressed by restricting resource management to root server
  - Requires subsystem asking root server to perform operations
    - *expensive!*
- Properly addressed by seL4's caps and resource-management

Security – performance tradeoff!

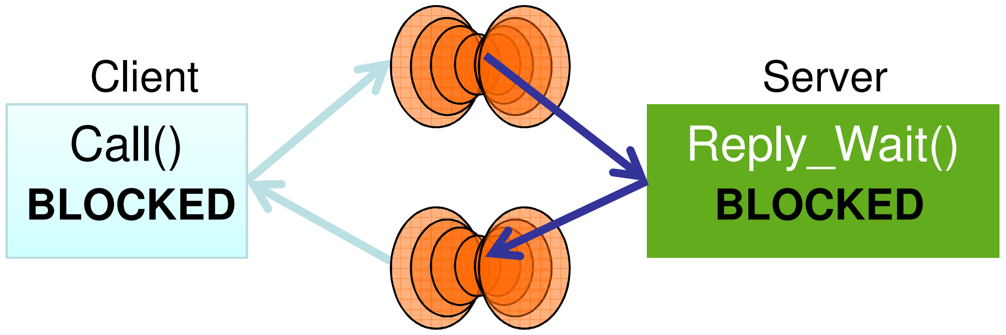


Lessons Learned from 2nd Generation

Suitability for real-time systems

- Basic idea was there: hard-prio round-robin scheduling, but…
  RT properties undermined by a number of implementation tricks!
  - "Lazy scheduling" to avoid frequent updates of scheduling queuess
    - Excellent average-case performance
    - How about worst case?

Idea:
Leave blocked threads in ready queue!



Lazy Scheduling

Scheduler must clean up the mess:

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

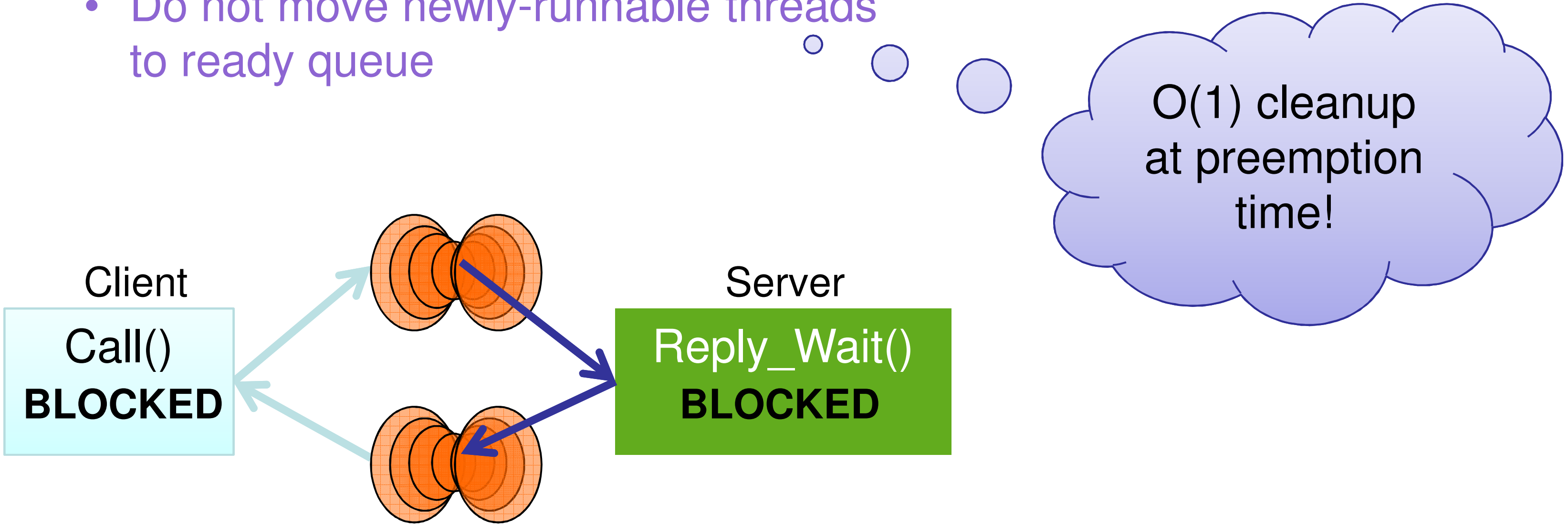
Scheduling becomes unbounded!

But scheduling cannot be preempted!

Lessons Learned from 2nd Generation

Suitability for real-time systems

- Basic idea was there: hard-prio round-robin scheduling, but…
  RT properties undermined by a number of implementation tricks!
  - "Lazy scheduling"
    - Excellent average-case performance
    - How about worst case?
  - "Benno scheduling":
    - Do not move newly-runnable threads to ready queue



Lessons Learned from 2nd Generation

Suitability for real-time systems

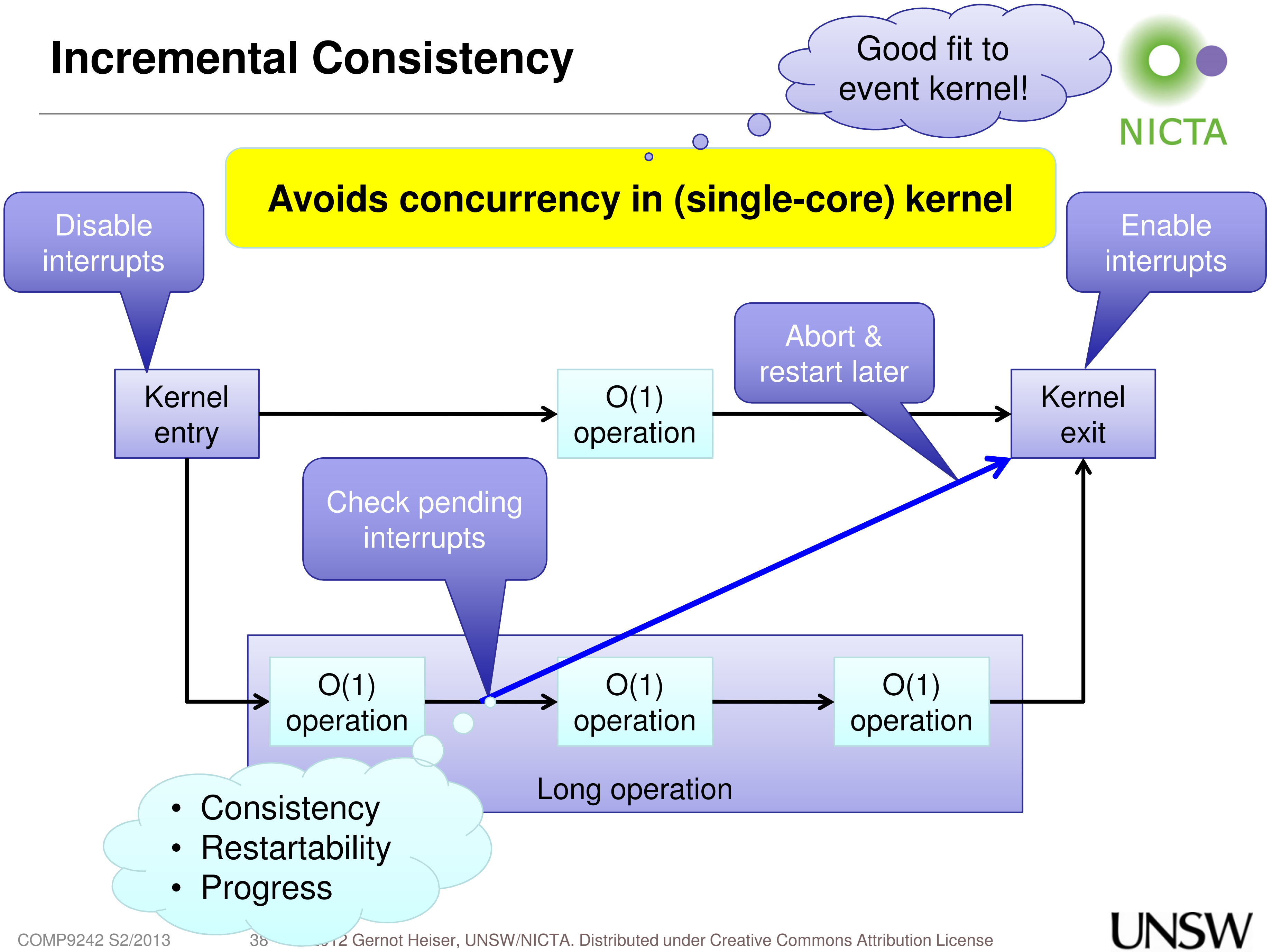
- Kernel runs with interrupts disabled
  - No concurrency control ⇒ simpler kernel
    - Easier reasoning about correctness
    - Better average-case performance
- How about long-running system calls?
  - V2 kernels use premption points
  - (Original) Fiasco has fully preemptible kernel
    - Like commercial microkernels (QNX, Green Hills INTEGRITY)

```
while (!done) {
    process_stuff();
    PSW.IRQ_disable=1;
    PSW.IRQ_disable=0;
}
```

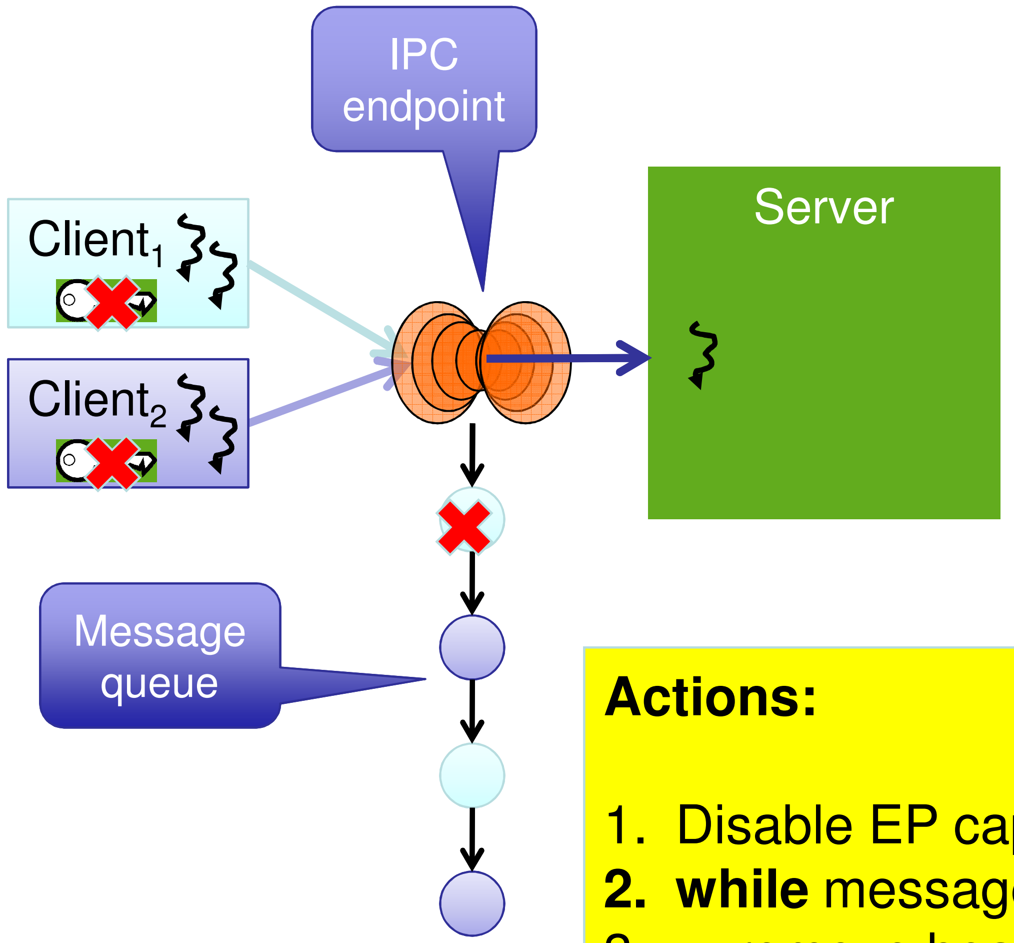
Some concurrency in kernel!

Lots of concurrency in kernel!

Incremental Consistency



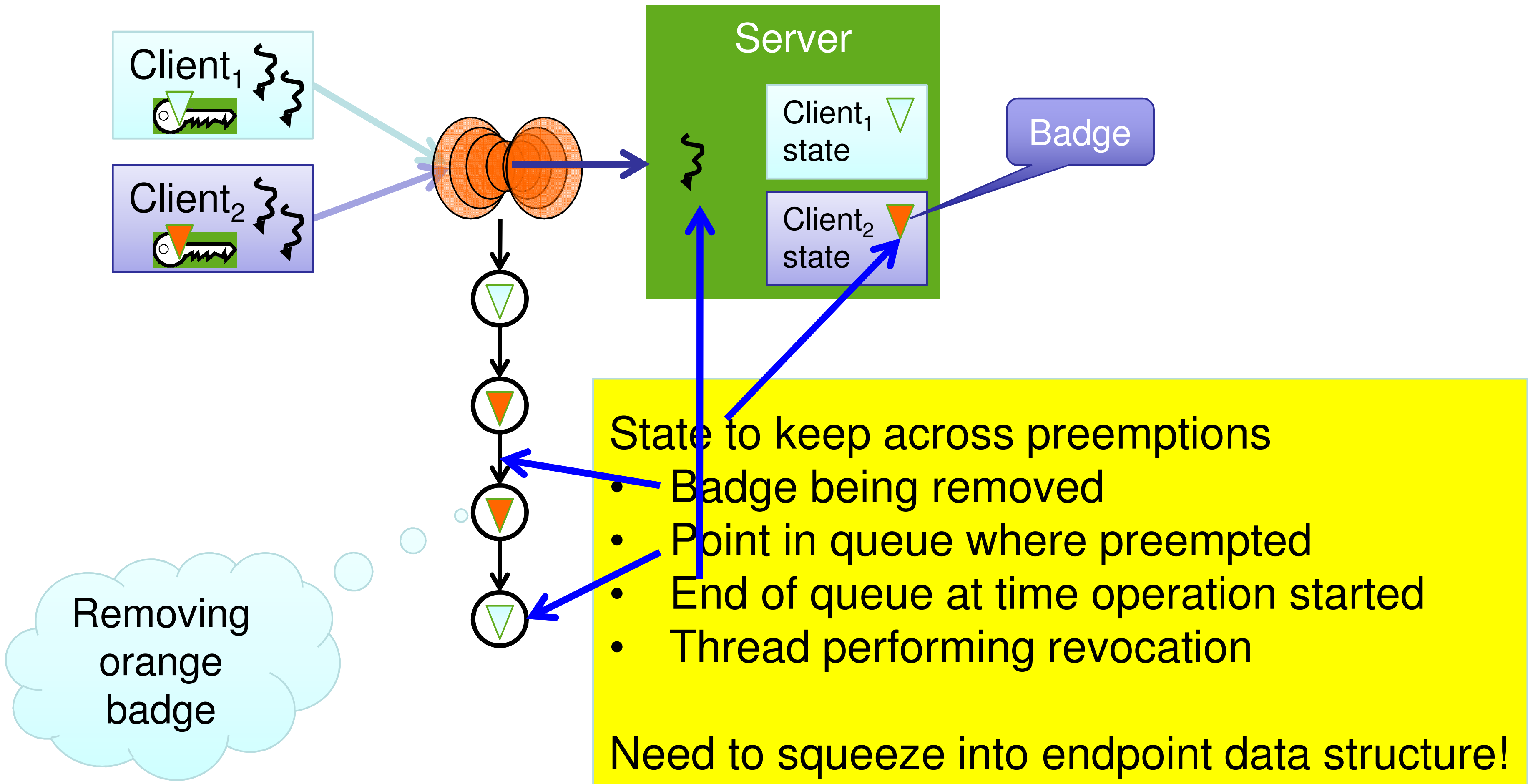
Example: Destroying IPC Endpoint



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 IPC “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 data structure!

seL4 Design Principles

- Fully delegatable access control
- All resource management is subject to user-defined policies
  - Applies to kernel resources too!
- Suitable for *formal verification*
  - Requires small size, avoid complex constructs
- Performance on par with best-performing L4 kernels
  - Prerequisite for real-world deployment!
- Suitability for real-time use
  - Only partially achieved to date ☹
    - on-going work…

(Informal) Requirements for Formal Verification

- Verification scales poorly ⇒ small size (LOC and API)
- Conceptual complexity hurts ⇒ KISS
- Global invariants are expensive ⇒ KISS
- Concurrency difficult to reason about ⇒ single-threaded kernel

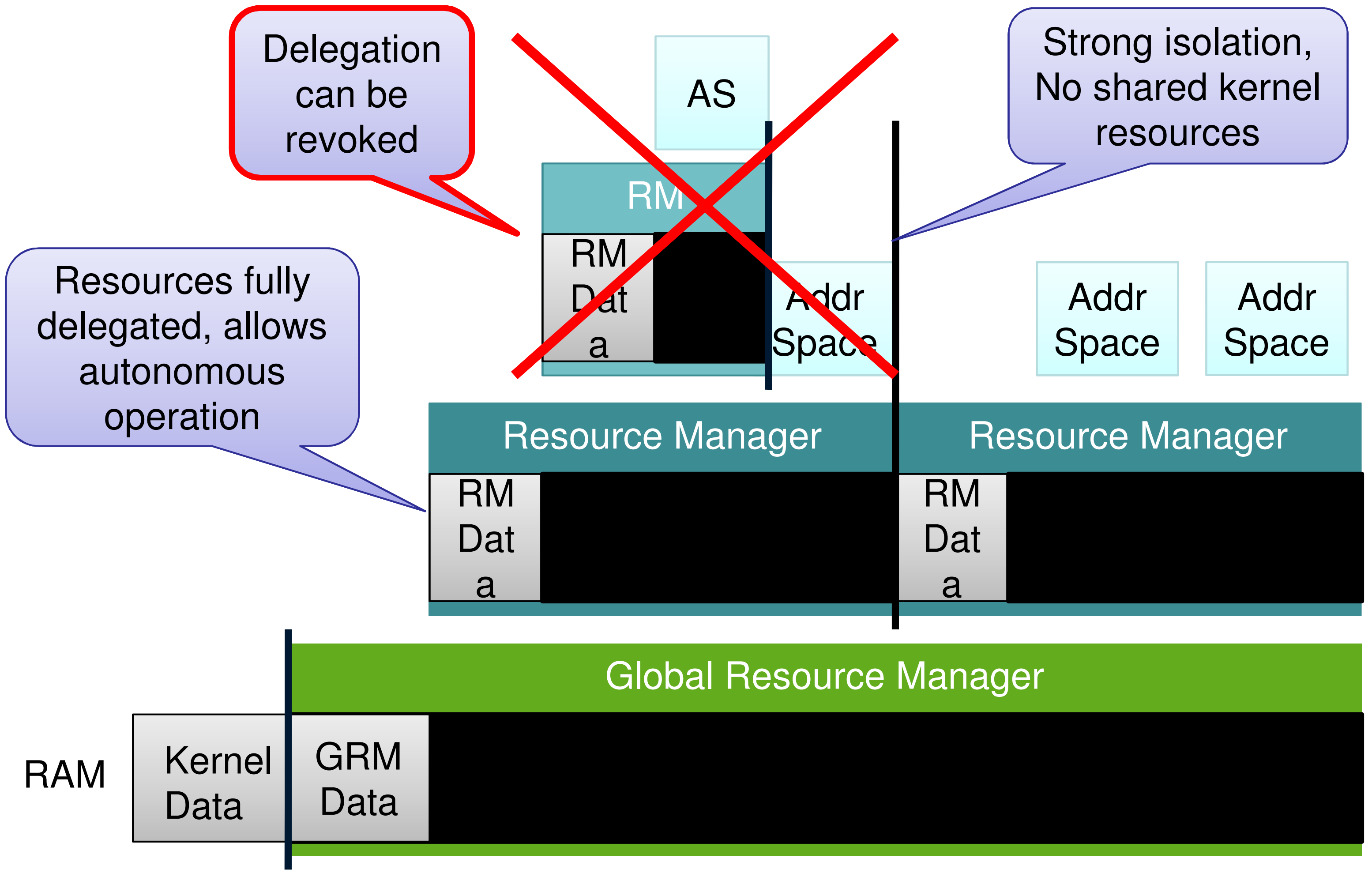
Largely in line with traditional L4 approach!

Fundamental Abstractions

- Capabilities as opaque names and access tokens
  - All kernel operations are cap invokations (except Yield())
- IPC:
  - Synchonous (blocking) message passing plus asynchous notification
  - Endpoint objects implemented as message queues
    - Send: get receiver TCB from endpoint or enqueue self
    - Receive: obtain sender's TCB from endpoint or enqueue self
- Other APIs:
  - Send()/Receive() to/from virtual kernel endpoint
  - Can interpose operations by substituting actual endpoint
- Fully user-controlled memory management

seL4's main conceptual novelty!

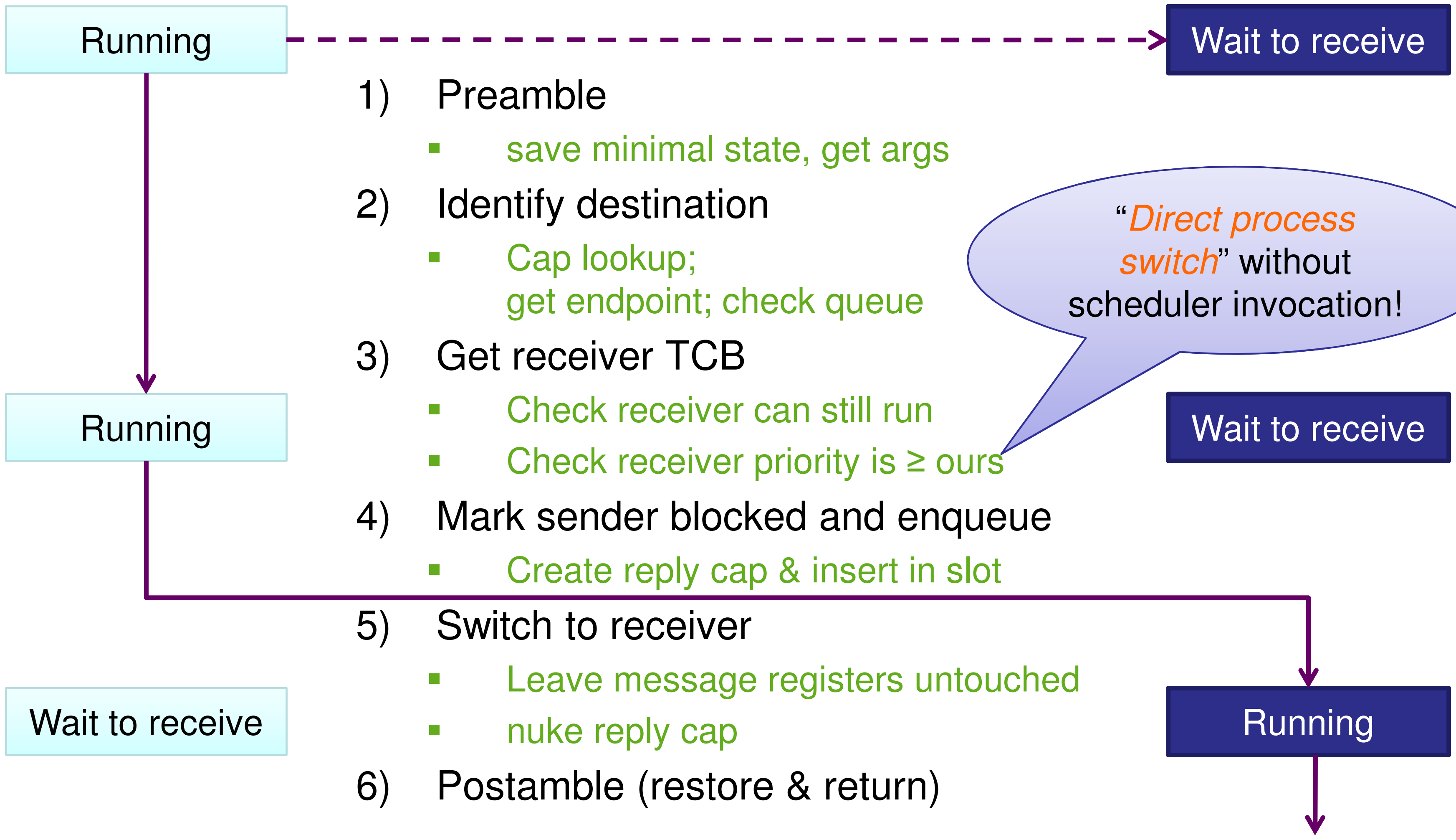
Remember: seL4 User-Level Memory Management



Synchronous IPC Implementation

185 cycles on ARM11!

Simple send (e.g. as part of RPC-like "call"):



1) Preamble
   - 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
   - Create reply cap & insert in slot
5) Switch to receiver
   - Leave message registers untouched
   - nuke reply cap
6) Postamble (restore & return)

"*Direct process switch*" without scheduler invocation!

Fastpath Coding Tricks

Common case: 0

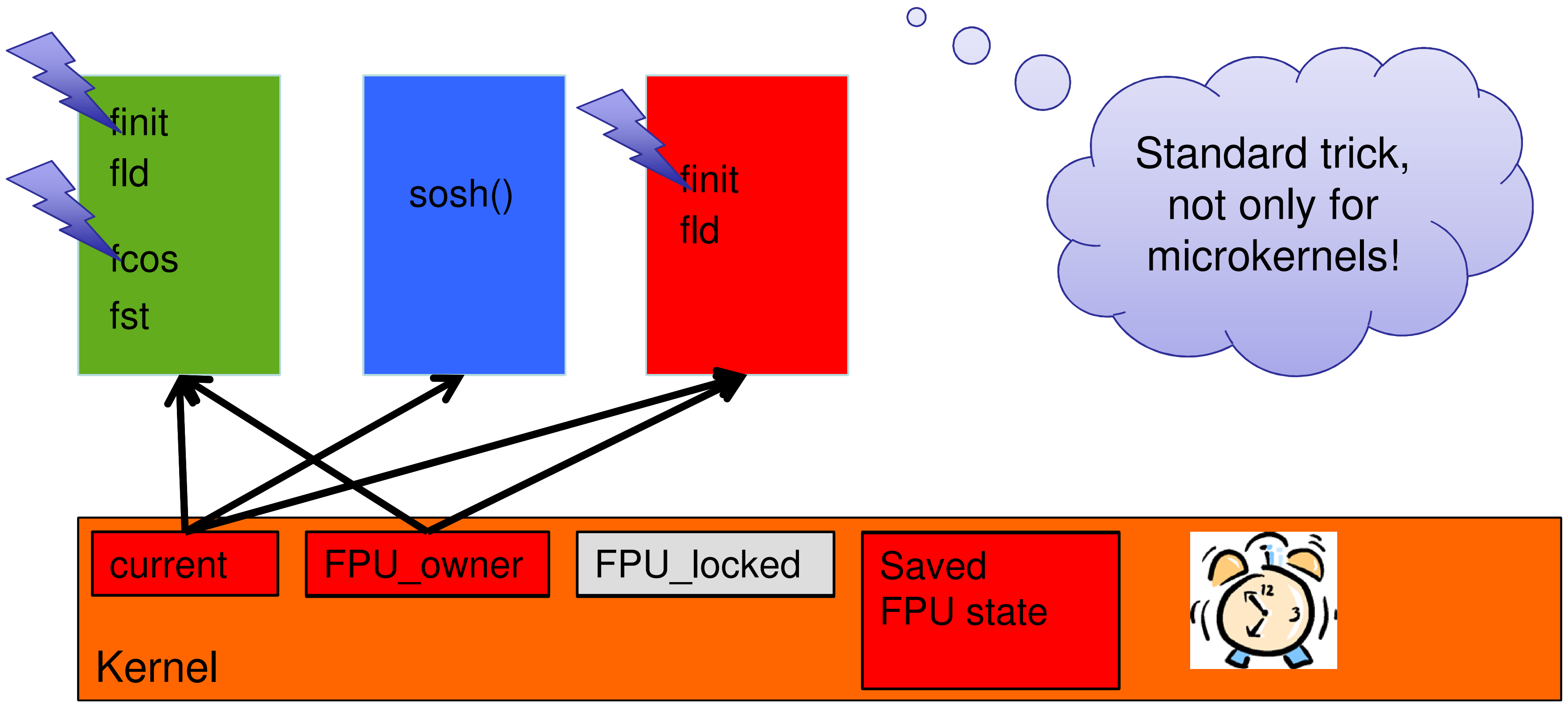
```
slow =      cap_get_capType(en_c) != cap_endpoint_cap ||
            !cap_endpoint_cap_get_capCanSend(en_c);
if (slow)   enter_slow_path();
```

Common case: 1

- Reduces branch-prediction footprint
- Avoids mispredicts, stalls & flushes
- Uses ARM instruction predication
- But: increases slow-path latency
  - should be minimal compared to basic slow-path cost

Lazy FPU Switch

- FPU context tends to be heavyweight
  - eg 512 bytes FPU state on x86
- Only few apps use FPU (and those don't do many syscalls)
  - saving and restoring FPU state on every context switch is wastive!



Other implementation tricks

- Cache-friendly data structure layout, especially TCBs
  - data likely used together is on same cache line
  - helps best-case and worst-case performance

Avoid RAM like the plague!

- Kernel mappings locked in TLB (using superpages)
  - helps worst-case performance
  - helps establish invariants: page table never walked when in kernel

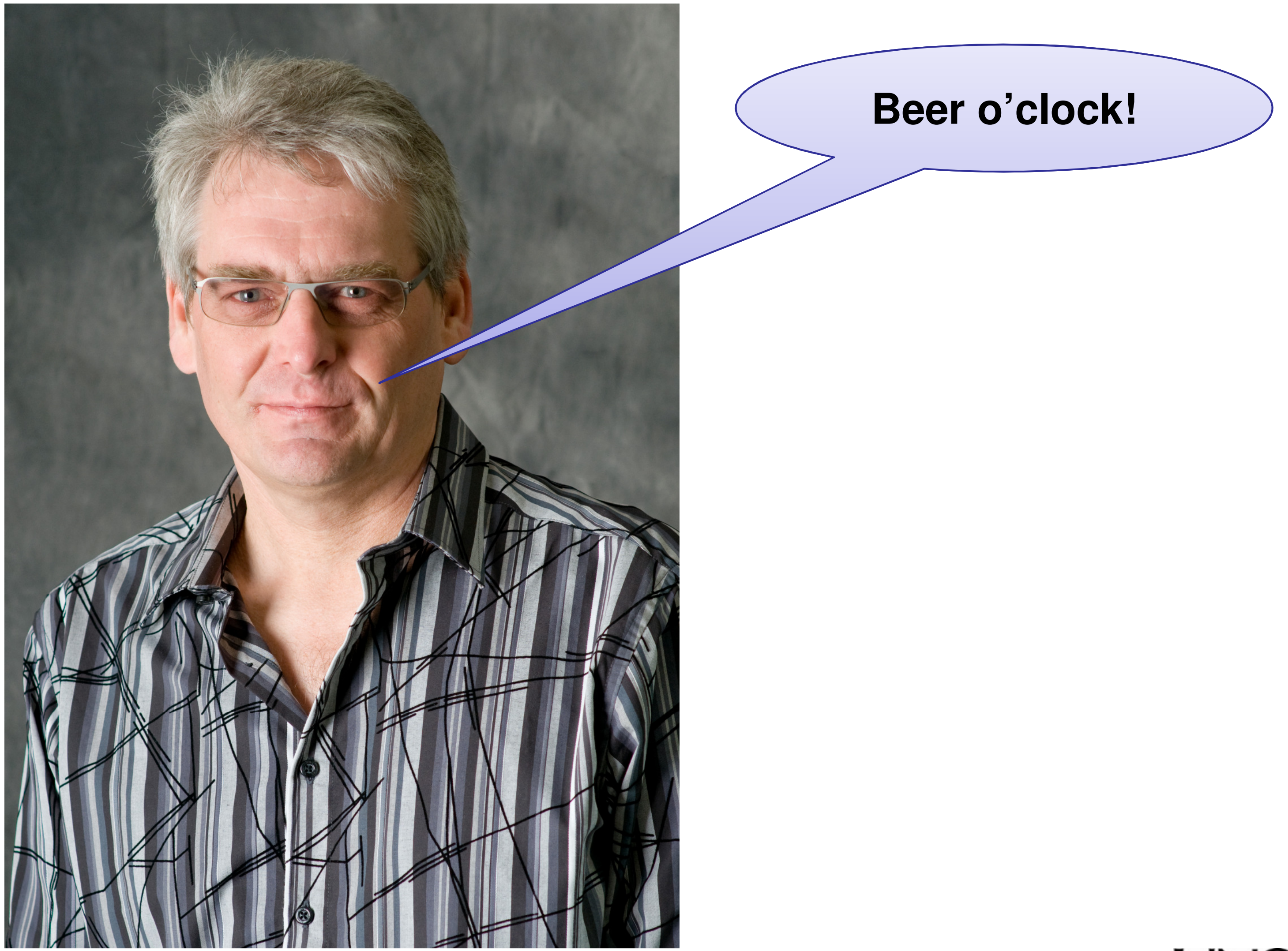
Remaining Conceptual Issues in seL4

IPC & Tread Model:

- Is the “mostly synchronous + a bit of async” model appropriate?
  - forces kernel scheduling of user activities
  - forces multi-threaded userland

Time management:

- Present scheduling model is ad-hoc and insufficient
  - fixed-prio round-robin forces policy
  - not sufficient for some classes of real-time systems (time triggered)
  - no real support for hierarchical real-time scheduling
  - lack of an elegant resource management model for time


Beer o'clock!