COMP9242 Advanced OS

S2/2016 W09: **Microkernel Design & Implementation**

@GernotHeiser

Never Stand Still | Engineering | Computer Science and Engineering

Copyright Notice

Microkernel Principles: Minimality

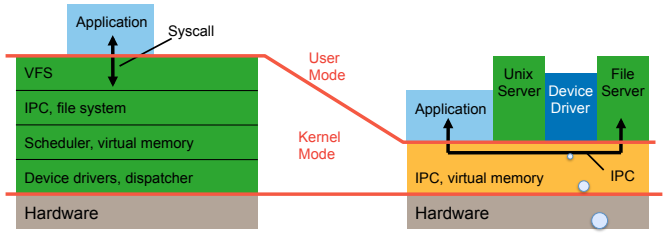
A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality. [SOSP'95]

- 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

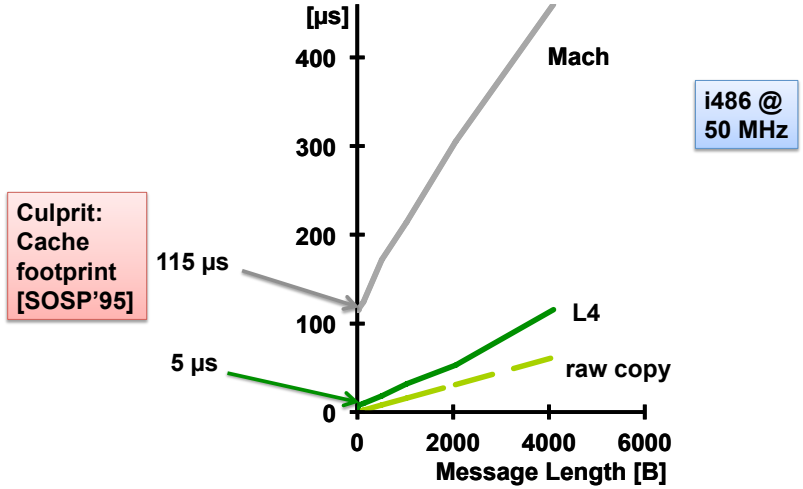
Consequence : User-level Services



- Kernel provides no services, only mechanisms
- Kernel is policy-free
  - Policies limit (good for 90% of cases, disastrous for some)
  - "General" policies lead to bloat, inefficiency

IPC performance is critical!

1993 “Microkernel” IPC Performance



  UNSW

L4 IPC Performance over 20 Years

Name	Year	Processor	MHz	Cycles	µs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	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	XScale 255	400	151	0.64
NOVA	2010	i7 Bloomfield (32-bit)	2,660	288	0.11
seL4	2013	i7 Haswell (32-bit)	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

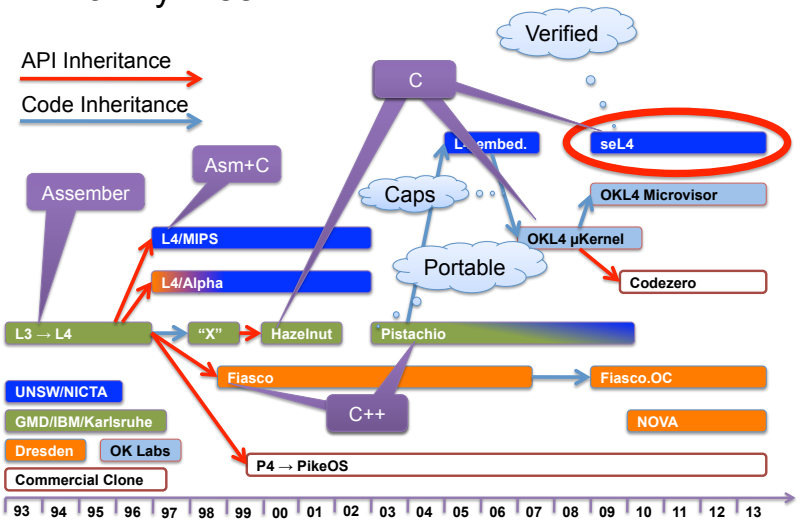
  UNSW

Minimality: Source Code Size

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

  UNSW

L4 Family Tree



  UNSW

L4 Deployments – in the Billions

Original L4 Design and Implementation

Implement. Tricks [SOSP'93]

- Process kernel
- Virtual TCB array
- Lazy scheduling
- Direct process switch
- Non-preemptible
- Non-portable
- Non-standard calling convention
- Assembler

Design Decisions [SOSP'95]

- Synchronous IPC
- Rich message structure, arbitrary out-of-line messages
- Zero-copy register messages
- User-mode page-fault handlers
- Threads as IPC destinations
- IPC timeouts
- Hierarchical IPC control
- User-mode device drivers
- Process hierarchy
- Recursive address-space construction

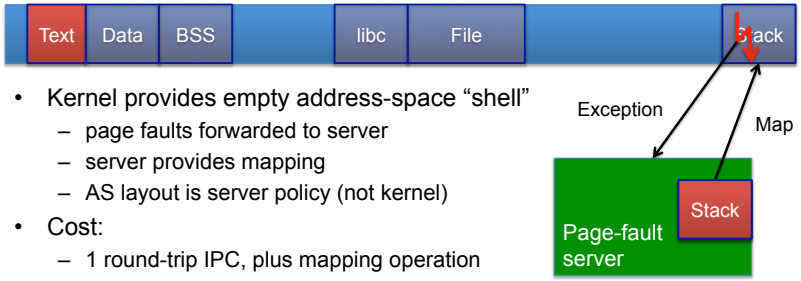
Objective: Minimise cache footprint and TLB misses

Design

What Mechanisms?

- Fundamentally, the microkernel must abstract
  - *Physical memory:* Address spaces
  - *CPU:* Threads
  - *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*

Memory: Policy-Free Address-Space Management

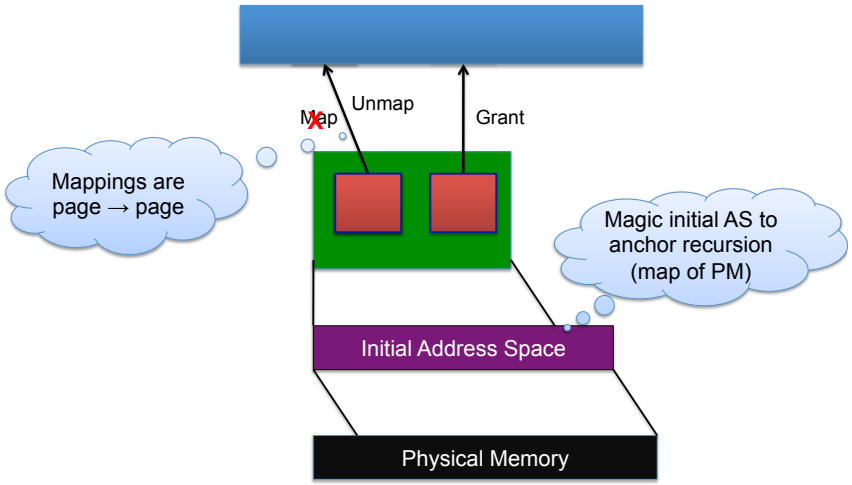


- Kernel provides empty address-space "shell"
  - page faults forwarded to server
  - server provides mapping
  - AS layout is server policy (not kernel)
- Cost:
  - 1 round-trip IPC, plus mapping operation
    - mapping may be side effect of IPC
    - kernel may expose data structure
- *Kernel mechanism:* forwarding page-fault exception
- "External pagers" first appeared in Mach [Rashid et al, '88]
  - … but were optional (and slow) – in L4 there's no alternative

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 (traditional L4)
  - TLB-like: cache for mappings which may vanish (OKL4 Microvisor)
    - Less predictable performance – real-time?
- Main design decision: is source of a mapping a page or a frame?
  - Frame: hardware-like
  - Page: recursive address spaces (original L4 model)

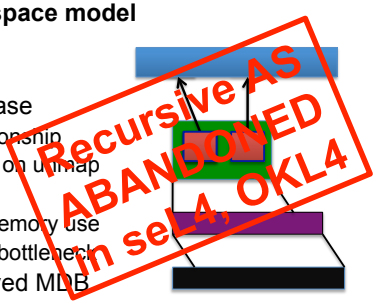
Traditional L4: Recursive Address Spaces



Recursive Address Space Experience

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



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"

  - Implementation-wise not much of a difference
  - Both need a stack in either domain
  - … but migrating thread requires kernel to provide/cache stacks
- Tight integration/interdependence with IPC model!

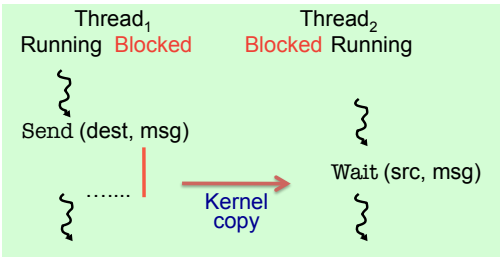
Abstracting Interrupts and Exceptions

- 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 traditionally special-cased (separate handler)
  - IPC message to page-fault server rather than exception handler
  - seL4 only has one exception handler endpoint

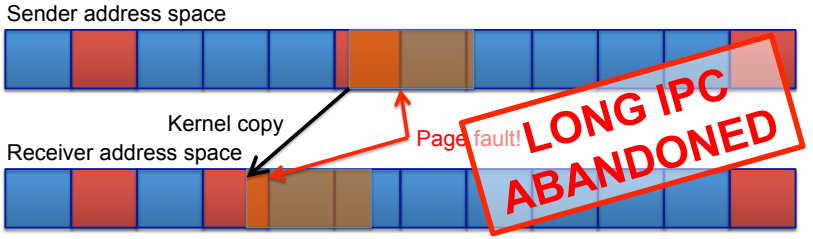
L4 IPC

Rendezvous model



Kernel executes in sender's context
- copies memory data directly to receiver (single-copy)
- leaves message registers unchanged during context switch (zero copy)

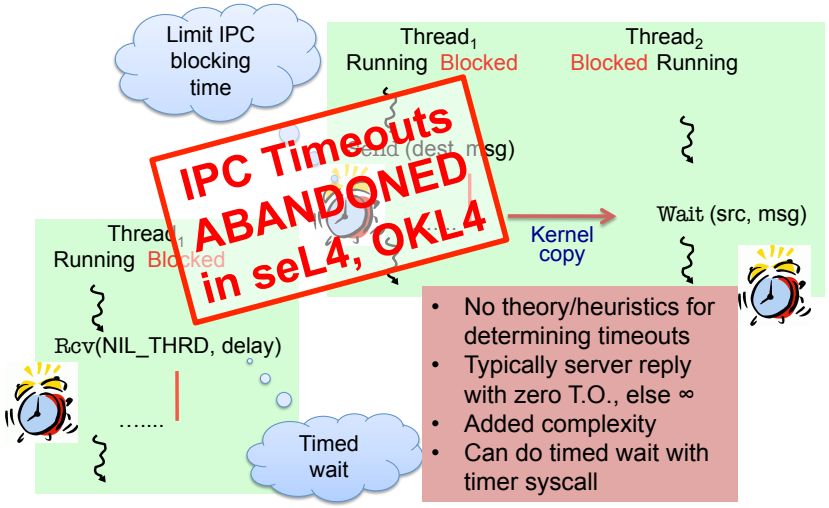
"Long" IPC



- IPC page faults are nested exceptions ⇒ In-kernel concurrency
  - L4 executes with interrupts disabled for performance, no concurrency
- Must invoke untrusted usermode page-fault handlers
  - potential for DOSing other thread
- Timeouts to avoid DOS attacks
  - complexity

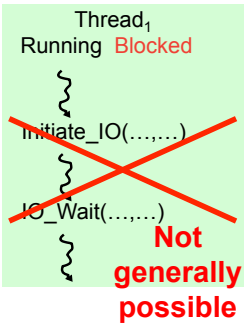
Why have long IPC?
- POSIX-style APIs
  write (fd, buf, nbytes)
- Usually prefer shared buffers

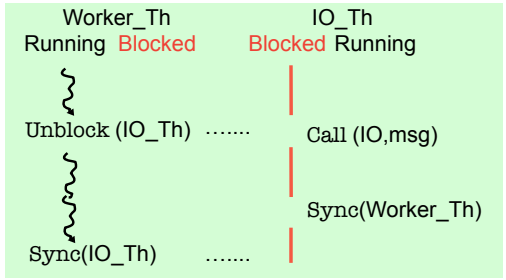
Timeouts



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

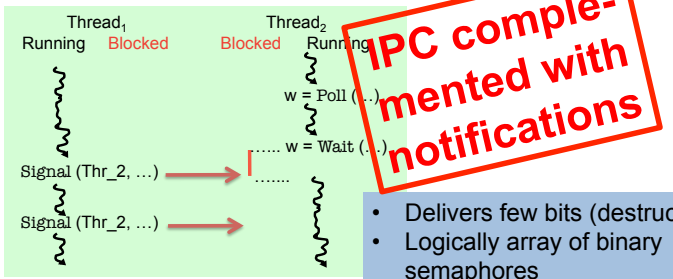
Synchronous IPC Issues





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

Notifications

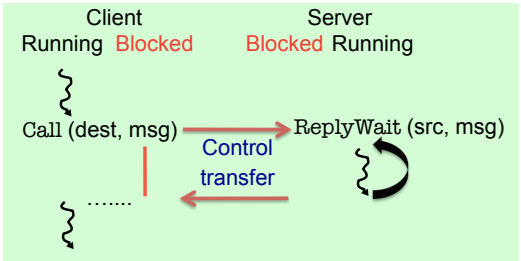


- Delivers few bits (destructively)
- Logically array of binary semaphores
- Maps well to interrupts, exceptions multicore, …

Thread can wait for IPC and notifications concurrently

Is IPC Redundant?

2 communication mechanisms: Minimality violation?

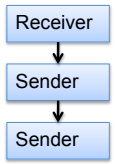


IPC is a user-controlled context switch

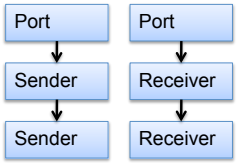
- only makes sense *intra-core*
- fast control transfer
- mimics migrating threads
- enables scheduling context donation
  - ➢ useful for real-time systems

Direct vs Indirect IPC Adressing

- Direct: Queue senders/messages at receiver
  - Need unique thread IDs
  - Kernel guarantees identity of sender
    - useful for authentication

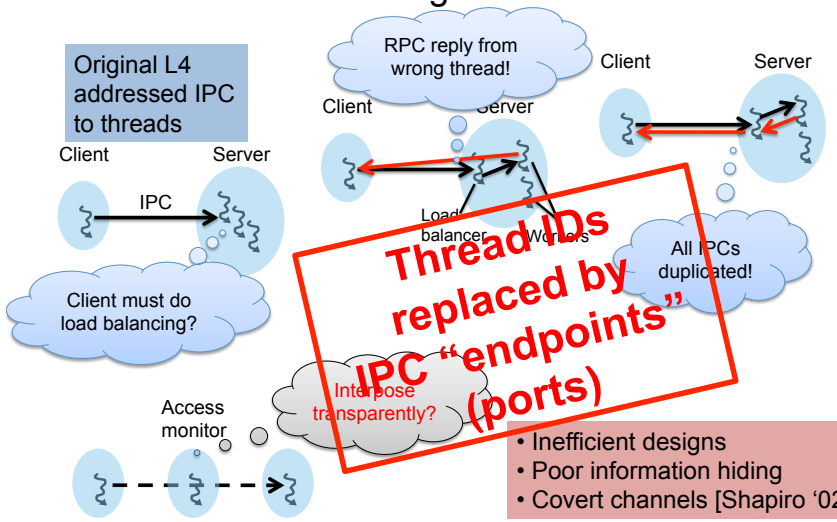


- Indirect: Mailbox/port object
  - Just a user-level handle for the kernel-level queue
  - Extra object type – extra weight?
  - Communication partners are anonymous
    - Need separate mechanism for authentication



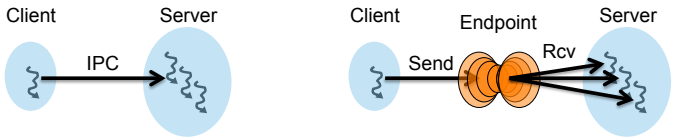
  UNSW

IPC Destination Naming

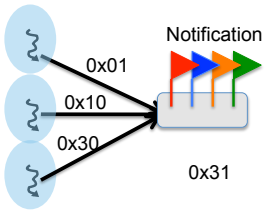


  UNSW

Endpoints and Notifications



- Endpoint queues senders/receivers
- Does not buffer messages

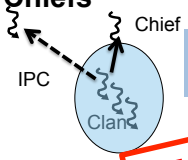


- Notification accumulates bits
- Does not buffer

  UNSW

Other Design Issues

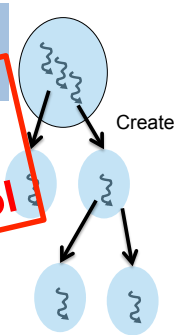
IPC Control: "Clans & Chiefs"



IPC outside clan re-directs to chief

Process Hierarchy

Hierarchical resource management



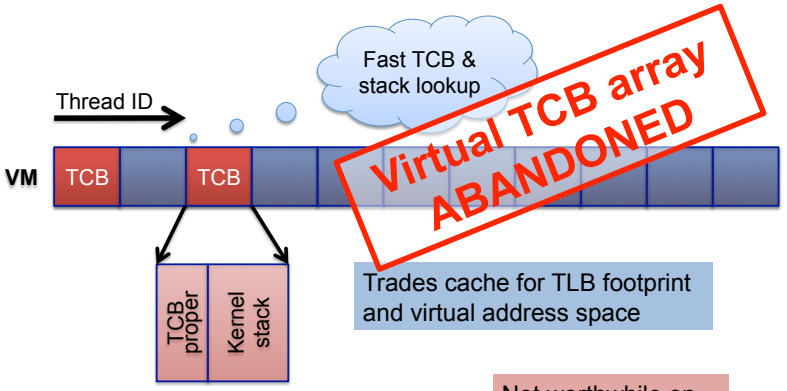
Hierarchies replaced by delegatable cap-based access control

- Inflexible, clumsy, inefficient hierarchies!
- Fundamental problem: no rights delegation

  UNSW

Implementation

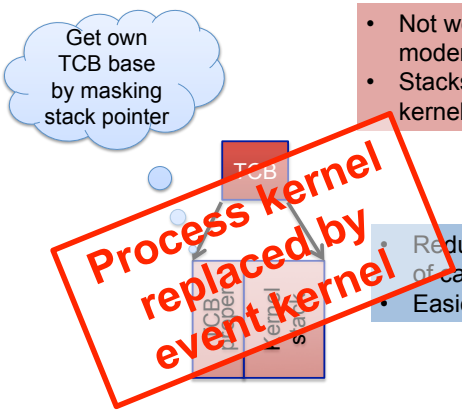
Virtual TCB Array



Trades cache for TLB footprint and virtual address space

Not worthwhile on modern processors!

Process Kernel: Per-Thread Kernel Stack



- Not worthwhile on modern processors!
- Stacks can dominate kernel memory use!

- Reduces TLB footprint at cost of cache and kernel memory
- Easier to deal with blocking

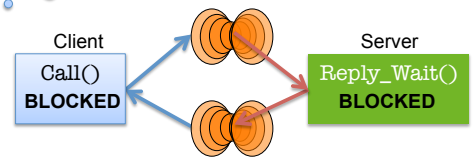
Scheduler Optimisation Tricks: Lazy Scheduling

```
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 Optimisation Tricks: Lazy Scheduling

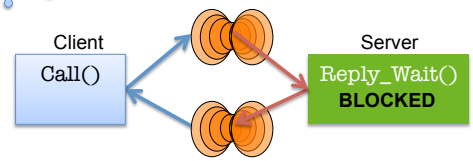
```
thread_t schedule() {
   foreach (prio in priorities) {
      foreach (thread in runQueue[prio]) {
         if (thread=head(runQueue[prio]))
            return thread;
         else
            unlinkedDequeue(thread);
      }
   }
   return idleThread;
}
```

Lazy scheduling REPLACED

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

Replaced by principled model of scheduling-context transfer

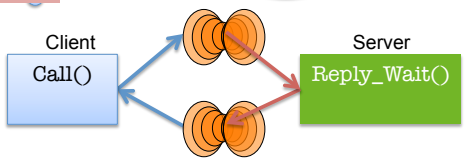
- Arguably, server should donate back if it's a server replying to a call()
- Hence, always donate on Reply_Wait()

Problems:
- Accounting (RT systems)
- Policy

Implication: Time slice donation – receiver runs on sender's time slice

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

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



Speaking of Real Time…

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

Limited concurrency in kernel!

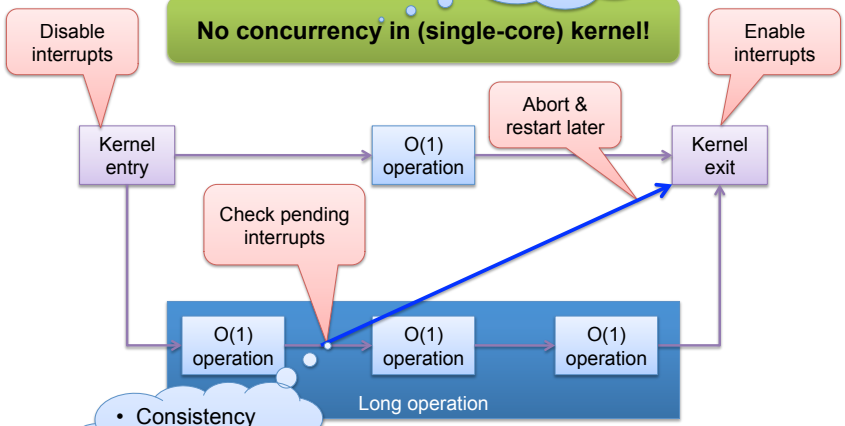
```
while (!done) {
   process_stuff();
   PSW.IRQ_disable=1;
   PSW.IRQ_disable=0;
}
```

Lots of concurrency in kernel!

Incremental Consistency

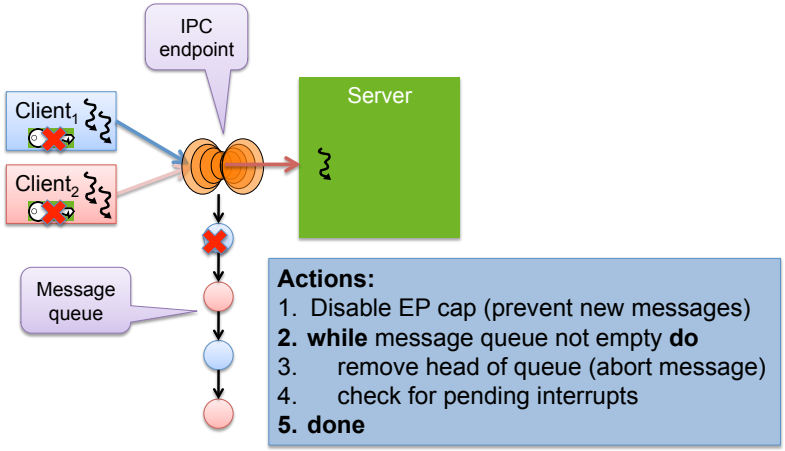
Good fit for event kernel!

No concurrency in (single-core) kernel!



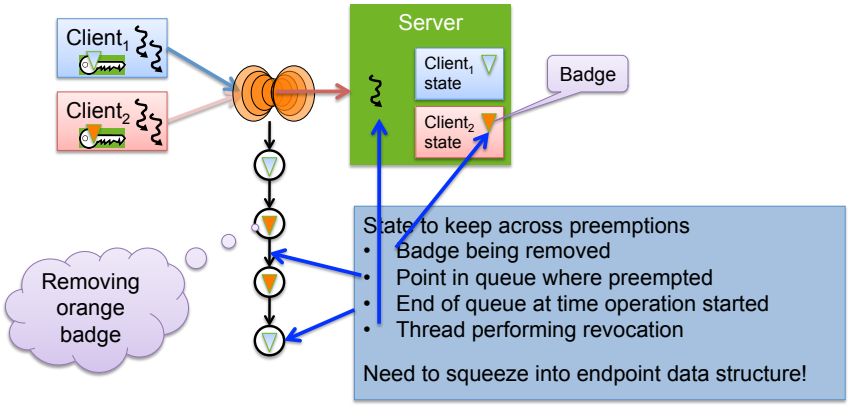
- Consistency
- Restartability
- Progress

Example: Destroying IPC Endpoint



  UNSW

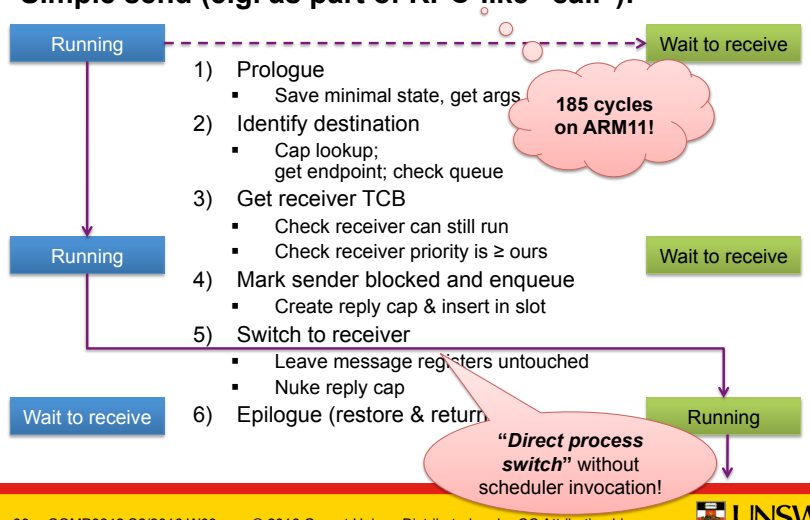
Difficult Example: Revoking IPC "Badge"



  UNSW

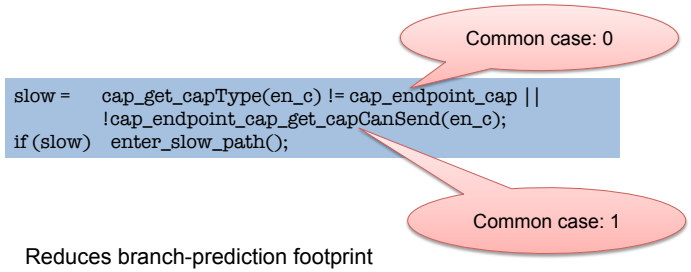
IPC Implementation

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



  UNSW

Fastpath Coding Tricks

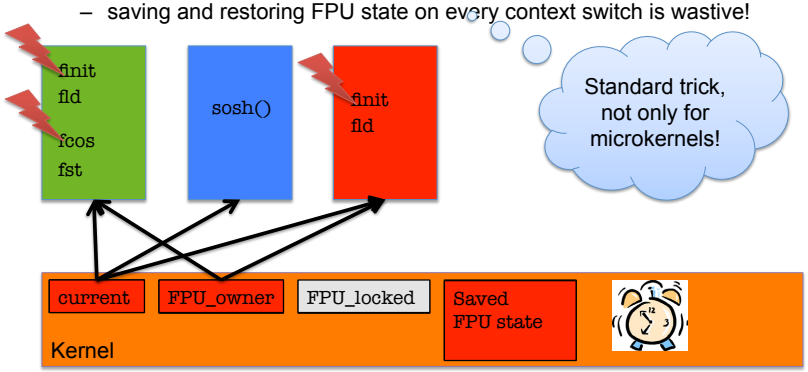


- Reduces branch-prediction footprint
- Avoids mispredicts, stalls & flushes
- Uses ARM instruction predication
- But: increases slow-path latency
  - should be minimal compared to basic slow-path cost

  UNSW

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

Other Lessons Learned from 2nd Generation

- 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 ['09], OKL4 ['09]: all C

- 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
    - Easily lost from assembler unless calling conventions maintained
    - seL4 performance with C-only pastpath as good as other L4 kernels [Blackham & Heiser '12]

C++ ABANDONED

Assembler coding ABANDONED

Lessons and Principles

Original L4 Design and Implementation

Implement. Tricks [SOSP'93]

- ~~Process kernel~~
- ~~Virtual TCB array~~
- Lazy scheduling
- Direct process switch
- Non-preemptible
- ~~Non-portable~~
- ~~Non-standard calling convention~~
- ~~Assembler~~

Design Decisions [SOSP'95]

- Synchronous IPC
- ~~Rich message structure, arbitrary out-of-line messages~~
- Zero-copy register messages
- User-mode page-fault handlers
- ~~Threads as IPC destinations~~
- ~~IPC timeouts~~
- ~~Hierarchical IPC control~~
- User-mode device drivers
- ~~Process hierarchy~~
- ~~Recursive address-space construction~~

Reflecting on Changes

Original L4 design had two major shortcomings

1. Insufficient/impractical resource control
   - Poor/non-existent control over kernel memory use
   - Inflexible 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

   IPC mangles:
   - Communication
   - Synchronisation
   - Memory management – sending mappings
   - Scheduling – time-slice donation

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
  - Important for safety-critical systems

(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!

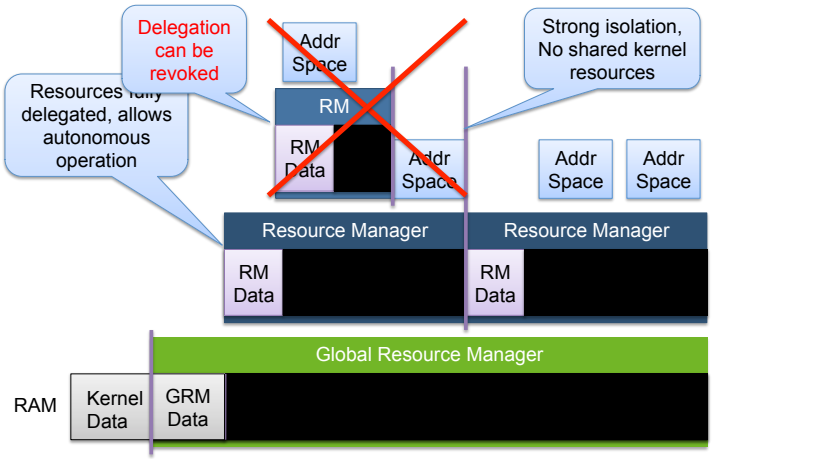
Main restriction presently is not passing pointers to stack variables

seL4 Fundamental Abstractions

- Capabilities as opaque names and access tokens
  - All kernel operations are cap invokations (except `Yield()`)
- IPC:
  - Synchronous (blocking) message passing
  - 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
- Notifications:
  - Arrays of binary semaphores for lightweight synchronisation
- 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!

seL4 Remember: Memory Management



seL4 Remaining Conceptual Issues

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

SOLVED!

- Scheduling contexts de-couple scheduling from IPC cleanly
  - Passive servers transfer scheduling-context
    - No scheduling needed
    - Migrating-threads model (without the stack allocation policy)
    - No artificial concurrency
  - Active servers have own scheduling context
    - Independently scheduled with well-defined effect

seL4 Remaining Conceptual Issues

Multicore Model:

- What is the right kernel design that scales up and down
- What is the role of IPC in multicore
  - Does cross-core IPC make any sense?
  - How does the RT scheduling model work on multicore?

In progress – details Week 12

seL4 Other Open Questions

- Time and space overhead of mapping operations
  - Model is not really tested on truly dynamic systems
  - Presently no support for superpages or batching mappings
  - Needs thorough, in-depth evaluation and playing with tradeoffs
- Interrupt handling model
  - Presently handler needs two syscalls
    - Acknowledging interrupt
    - Waiting for next interrupt
  - Keeps wait() implementation fast and simple, but may not be optimal
  - Needs thorough, in-depth evaluation and playing with tradeoffs

Lessons From 20 Years of L4

- Minimality is excellent driver of design decisions
  - L4 kernels have become simpler over time
  - Policy-mechanism separation (user-mode page-fault handlers)
  - Device drivers really belong to user level
  - Minimality is key enabler for formal verification!
- IPC speed still matters
  - But not everywhere, premature optimisation is wasteful
  - Compilers have got so much better
  - Verification does not compromise performance
  - Verification invariants can help improve speed! [Shi, OOPSLA'13]
- Capabilities are the way to go

- Details changed, but principles remained
- Microkernels rock! (If done right!)

