School of Computer Science & Engineering

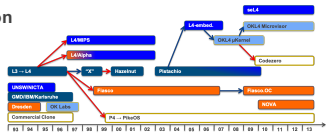
COMP9242 Advanced Operating Systems

2019 T2 Week 05b

Microkernel Design & Implementation

The 25-year quest for the right API

@GernotHeiser



Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 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/3.0/legalcode

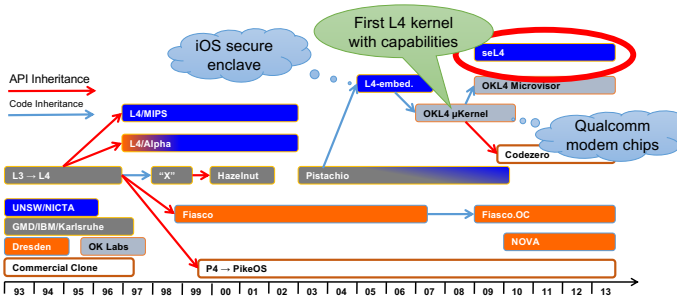
L4 Microkernels – Deployed by the Billions

L4: The Quest for a Real Microkernel

L4: The Quest for a Real Microkernel

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



L4 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**	**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**	**ARM11**	**532**	**188**	**0.35**
seL4	**2018**	**i7 Haswell (64-bit)**	**3,400**	**442**	**0.13**
seL4	**2018**	**Cortex A9**	**1,000**	**303**	**0.30**

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**

What Have We Learnt in 25 Years?

Issues With 2G Microkernels

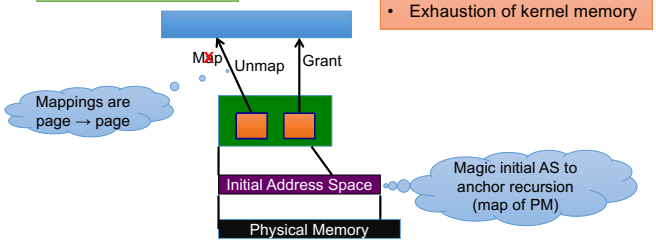
- L4 solved microkernel performance [Härtig et al, SOSP'97] left a number of issues unsolved
- Problem: ad-hoc approach to security and resource management
  - Global thread name space ⇒ covert channels [Shapiro'03]
  - Threads as IPC targets ⇒ insufficient encapsulation
  - Single kernel memory pool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

Solved by capabilities

Traditional L4: Recursive Address Spaces

Replaced by magic-free seL4 resource model

Issues:
- Complex mapping DB
- Exhaustion of kernel memory



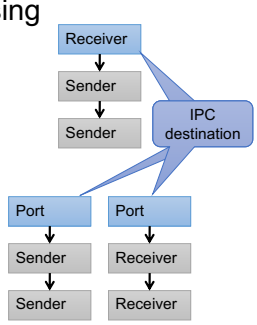
Issues With 2G Microkernels

- 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
  - Single kernel memory pool ⇒ DoS attacks
  - No delegation of authority ⇒ impacts flexibility, performance
  - Unprincipled management of time

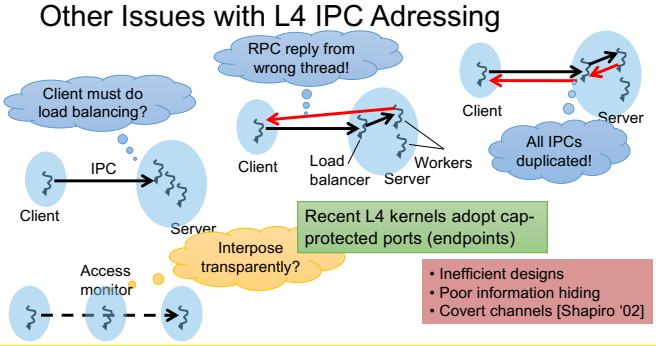
Solved by seL4 memory management model

Direct vs Indirect IPC Addressing

- 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



Other Issues with L4 IPC Adressing



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
  - Single kernel memory pool ⇒ DoS attacks
  - No delegation of authority ⇒ impacts flexibility, performance
  - Unprincipled management of time

Solved by caps & endpoints

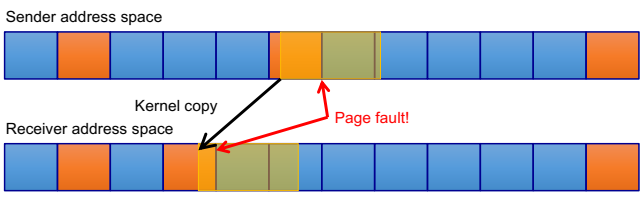
Examine later

Other Design & Implementation Issues

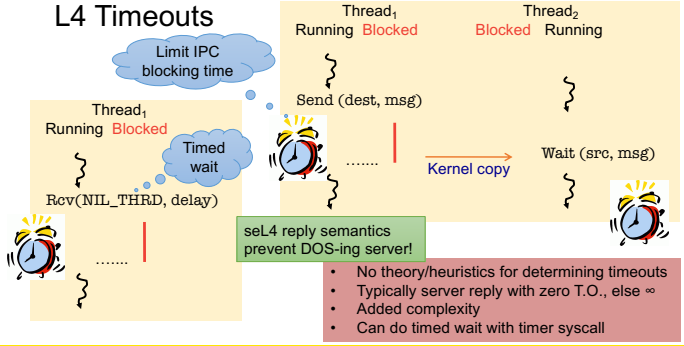
L4 "Long" IPC

Abandoned in seL4

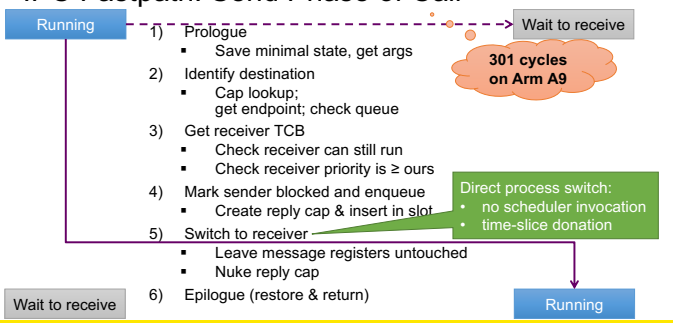
- Not minimal
- Source of kernel complexity:
  - nested exceptions
  - concurrency in kernel
  - must upcall PF handlers during IPC
  - timeouts to prevent DOS attacks



L4 Timeouts



IPC Fastpath: Send Phase of Call



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 (slightly)
  - should be minimal compared to basic slow-path cost

How About Real-Time Support?

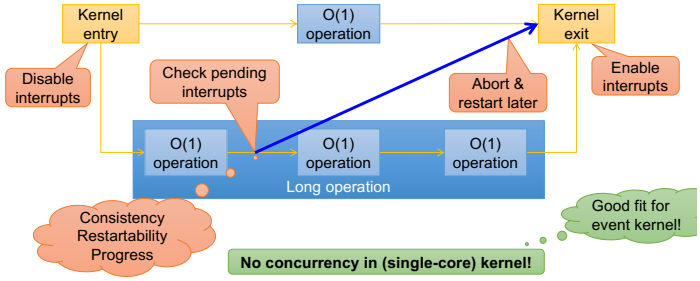
- Kernel runs with interrupts disabled
  - No concurrency control ⇒ simpler kernel
    - Easier reasoning about correctness
    - Better average-case performance

How about long-running system calls?

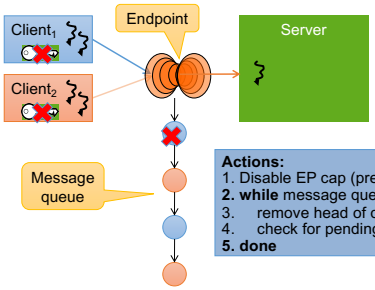
Most protected-mode RTOSes are fully preemptible

Lots of concurrency in kernel!

seL4 Incremental Consistency Paradigm

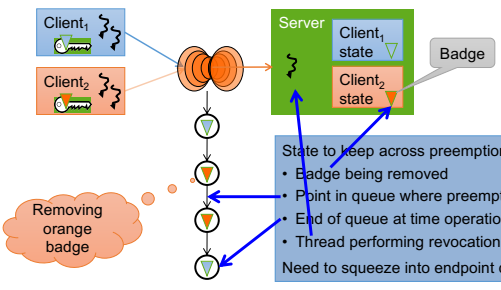


seL4 Example: Destroying IPC Endpoint



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**

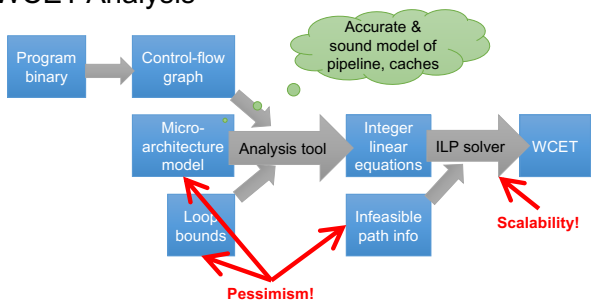
seL4 Difficult Example: Revoking Badge



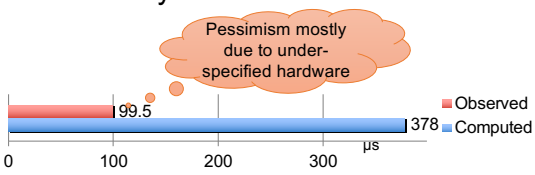
State to keep across preemptions
- Badge being removed
- Point in queue where preempted
- End of queue at time operation started
- Thread performing revocation

Need to squeeze into endpoint data structure!

WCET Analysis



seL4 WCET Analysis on ARM11



WCET presently limited by verification practicalities
- without regard to verification achieved 50 µs
- 10 µs seem achievable
- BCET ~ 1µs
- [Blackham'11, '12] [Sewell'16]

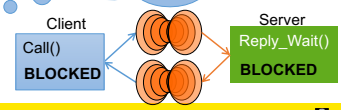
L4 Scheduler Optimisation: Lazy Scheduling

```
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



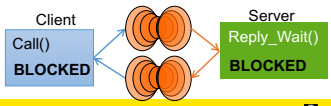
seL4 Scheduler: Benno Scheduling

```
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

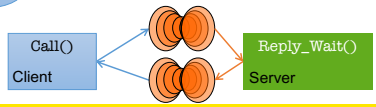
- Sender was running ⇒ had highest prio
- If receiver prio ≥ sender prio ⇒ run receiver

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

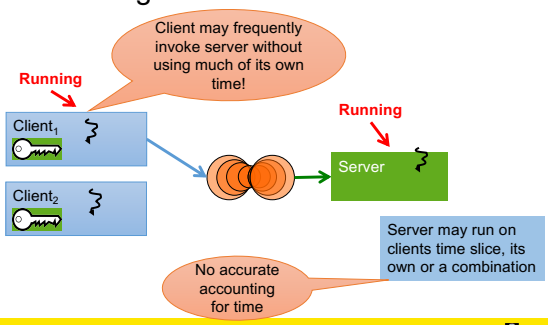
Implication: Time slice donation – receiver runs on sender's time slice

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

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



Remember: Delegation of Critical Sections



New Model: Scheduling Contexts

Classical thread attributes
- Priority
- Time slice

Not runnable if null

New thread attributes
- Priority
- Scheduling context capability

Capability for time

Presently being merged into mainline

Scheduling context object
- T: period
- C: budget (≤ T)

Limits CPU access!

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

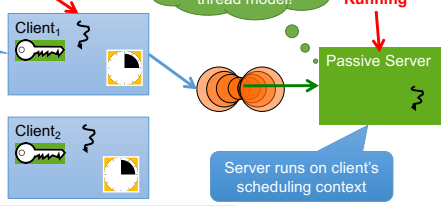
Delegation with Scheduling Contexts

Running

Passive servers support migrating thread model!

Running

Client is charged for server's time



Server runs on client's scheduling context

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

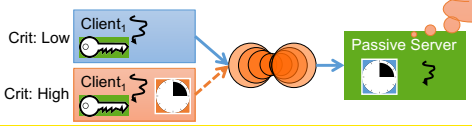
Mixed-Criticality Support

For supporting *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 schedduling context?



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 (together with one of the above)
- Implement priority inheritance (if you must…)

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
  - Single kernel memory pool ⇒ DoS attacks
  - No delegation of authority ⇒ impacts flexibility, performance
  - Unprincipled management of time

Solved by scheduling contexts & time-out exceptions

Lessons & 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~~

Modified

Retained

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:

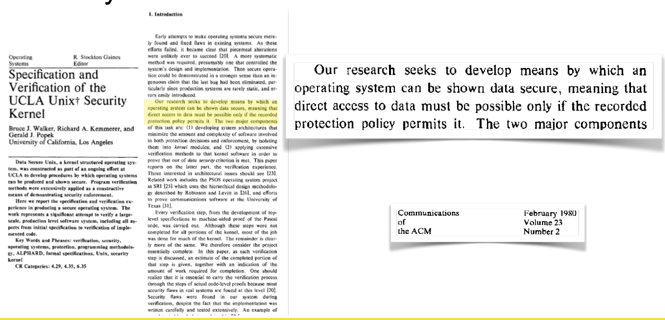
1. Insufficient/impractical resource control
   - Poor/non-existent control over kernel memory use
   - Inflexible & costly process hierarchies (policy!)
   - Arbitrary limits on number of address spaces and threads (policy!)
   - Poor information hiding (IPC addressed to threads)
   - Insufficient mechanisms for authority delegation
2. Over-optimised IPC abstraction, mangles:
   - Communication
   - Synchronisation
   - Memory management – sending mappings
   - Scheduling – time-slice donation

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!

A Thirty-Year Dream!


Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek
University of California, Los Angeles

1. Introduction

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