L4 Microkernels – Deployed by the Billions

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 IPC Performance Over the Years

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Processor</th>
<th>MHz</th>
<th>Cycles</th>
<th>µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1993</td>
<td>i486</td>
<td>50</td>
<td>250</td>
<td>5.00</td>
</tr>
<tr>
<td>Original</td>
<td>1997</td>
<td>Pentium</td>
<td>160</td>
<td>121</td>
<td>0.75</td>
</tr>
<tr>
<td>L4/MIPS</td>
<td>1997</td>
<td>R4700</td>
<td>100</td>
<td>86</td>
<td>0.86</td>
</tr>
<tr>
<td>L4/Alpha</td>
<td>1997</td>
<td>21064</td>
<td>433</td>
<td>45</td>
<td>0.10</td>
</tr>
<tr>
<td>Hazelnut</td>
<td>2002</td>
<td>Pentium 4</td>
<td>1,400</td>
<td>2,000</td>
<td>1.38</td>
</tr>
<tr>
<td>Pistachio</td>
<td>2005</td>
<td>Itanium</td>
<td>1,500</td>
<td>36</td>
<td>0.02</td>
</tr>
<tr>
<td>OKL4</td>
<td>2007</td>
<td>XScale 255</td>
<td>400</td>
<td>151</td>
<td>0.64</td>
</tr>
<tr>
<td>NOVA</td>
<td>2010</td>
<td>It Blomfield (32-bit)</td>
<td>2,660</td>
<td>288</td>
<td>0.11</td>
</tr>
<tr>
<td>sel4</td>
<td>2013</td>
<td>ARM11</td>
<td>532</td>
<td>188</td>
<td>0.35</td>
</tr>
<tr>
<td>sel4</td>
<td>2018</td>
<td>i7 Haswell (64-bit)</td>
<td>3,400</td>
<td>442</td>
<td>0.13</td>
</tr>
<tr>
<td>sel4</td>
<td>2018</td>
<td>Cortex A9</td>
<td>1,000</td>
<td>382</td>
<td>0.30</td>
</tr>
</tbody>
</table>

Minimality: Source-Code Size

<table>
<thead>
<tr>
<th>Name</th>
<th>Architecture</th>
<th>C/C++</th>
<th>asm</th>
<th>total kSLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>i486</td>
<td>0</td>
<td>6.4</td>
<td>6.4</td>
</tr>
<tr>
<td>L4/Alpha</td>
<td>Alpha</td>
<td>0</td>
<td>14.2</td>
<td>14.2</td>
</tr>
<tr>
<td>L4/MIPS</td>
<td>MIPS64</td>
<td>6.0</td>
<td>4.5</td>
<td>10.5</td>
</tr>
<tr>
<td>Hazelnut</td>
<td>x86</td>
<td>10.0</td>
<td>0.8</td>
<td>10.8</td>
</tr>
<tr>
<td>Pistachio</td>
<td>x86</td>
<td>22.4</td>
<td>1.4</td>
<td>23.0</td>
</tr>
<tr>
<td>L4/ARMv5</td>
<td>ARMv5</td>
<td>7.6</td>
<td>1.4</td>
<td>9.0</td>
</tr>
<tr>
<td>OKL4 3.0</td>
<td>ARMv6</td>
<td>15.0</td>
<td>0.0</td>
<td>15.0</td>
</tr>
<tr>
<td>Fiasco.OC</td>
<td>x86</td>
<td>36.2</td>
<td>1.1</td>
<td>37.3</td>
</tr>
<tr>
<td>sel4</td>
<td>ARMv6</td>
<td>9.7</td>
<td>0.5</td>
<td>10.2</td>
</tr>
</tbody>
</table>

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

Issues:
- Complex mapping DB
- Exhaustion of kernel memory

Replaced by magic-free sel4 resource model

Mappings are page → page

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

12

Other Issues with L4 IPC Addressing

13

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

14

Other Design & Implementation Issues

15

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

Sender address space

Receiver address space

Kernel copy

Page fault!

16

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

Thread (src, msg)

Rcv(NIL_THRD, delay)

server)

send)

17
IPC Fastpath: Send Phase of Call

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
   - Create reply cap & insert in slot
5) Switch to receiver
   - Leave message registers untouched
   - Nuke reply cap
6) Epilogue (restore & return)

301 cycles on Arm A9

Fastpath Coding Tricks

- slow = cap_get_capType(en_cap) != cap_endpoint_cap ||
  cap_endpoint_cap_get_capCanSend(en_cap);
- if (slow) enter_slow_path();

- Common case: 0
  • 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

- Common case: 1
  • Changes to real-time
  • Kernel entry
  • O(1) operation
  • Kernel exit
  • No concurrency in (single-core) kernel!

How About Real-Time Support?

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

Lots of concurrency in kernel!

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

How about long-running system calls?

Inconsistent data structures?

Incremental Consistency Paradigm

Consistency Restartability

Progress

Good fit for event kernel!

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

Program binary → Control-flow graph → Micro-architecture model → Analysis tool → Integer linear equations → WCET

Accurate & sound model of pipeline, caches → Feasible path info → WCET

Pessimism!

Scalability!

WCET Analysis on ARM11

Pessimism mostly due to under-specified hardware

WCET presently limited by verification practicalities

- without regard to verification achieved 50 µs
- 10 µs seem achievable
- BCET ~ 1 µs

[Blackham'11, '12] [Sewell'16]

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;
}
```

- Frequent blocking/unblocking in IPC-based systems
- Many ready-queue manipulations

Idea: leave blocked threads in ready queue, scheduler cleans up

L4 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

Remember: Delegation of Critical Sections

Client may frequently invoke server without using much of its own time!

Client may frequently invoke server without using much of its own time!

Client may frequently invoke server without using much of its own time!
New Model: Scheduling Contexts

Classical thread attributes
- Priority
- Time slice

New thread attributes
- Priority
- Scheduling context capability

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

Limits CPU access!

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

C = 2
T = 3

C = 250
T = 1000

Not runnable if null

Delegation with Scheduling Contexts

Client is charged for server’s time

Passive server runs on client’s scheduling context

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]

Delegation with Scheduling Contexts

Client

Running

Passive Server

Running

Mixed-Criticality Support

For supporting mixed-criticality systems (MCS), OS must provide:
- Temporal isolation, to force jobs to adhere to declared WCET
- Mechanisms for safely sharing resources across criticalities

What if budget expires while shared server executing on Low’s scheduling context?

Mixed-Criticality Support

Crit: Low

Crit: High

Client

Passive Server

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…)

Timeout Exceptions

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

Issues With 2G Microkernels

Lessons & Principles

Solved by scheduling contexts & time-out exceptions
**Original L4 Design and Implementation**

**Implement. Tricks [SOSP’93]**
- Pre-emption
- Virtual TCB evil
- Lazy scheduling
- Direct process switch
- Non-preemptible
- Non-portable
- Non-standard calling convention

**Design Decisions [SOSP’95]**
- Synchronous IPC
- Rich message structure
- Zero-copy register messages
- User-mode page-fault handlers
- Throttled IPC timeouts
- Hierarchical IPC control
- User-mode device drivers
- Process hierarchy
- Selective 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

**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

**A Thirty-Year Dream!**

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 readset protection policy permits it. The two major components...