Microkernels & Proof-Carrying Code

Prof. Phillip Gibbons

Spring 2020, Lecture 16
“On $\mu$-Kernel Construction”

Jochen Liedtke 1995

- **Jochen Liedtke** (GMD)
  - IBM Research, U. Karlsruhe
  - Worked on micro-kernels most of his professional life
  - SOSP’93, SOSP’95, SOSP’97 papers
  - d. 2001. “We pay tribute to his brilliance”
This paper presented the core design ideas behind the L4 microkernel, especially the minimality principle, which states that functionality must only be implemented inside the kernel if moving it outside would prevent the implementation of required system functionality. This principle was at the heart of L4’s design, and supported a ruthless performance focus, which allowed L4 to outperform other microkernels by an order of magnitude. The core ideas of this paper led to a family of L4 microkernels which were commercially deployed on a large scale, and eventually enabled unprecedented assurance through formal verification.
Conventional Wisdom on μ-Kernels (25 years since introduced)

• Advantages
  – Clear μ-kernel interface enforces modularity
  – Enhanced isolation from server malfunction
  – More flexible/tailorable
  – Simple enough to be formally verified (see Klein09)

• Disadvantages:
  – μ-kernels are inherently inefficient
  – μ-kernels are insufficiently flexible

Inefficiency due to increased user-kernel mode & address-space switches
Minimality Principle

Functionality must only be implemented inside the kernel
IF
moving it outside would prevent the implementation
of required system functionality

• Moving it outside => permits competing implementations

• Assume: Must deal with protection. HW provides page-based protections for virtual pages

Principle of independence
– Subsystem S can give guarantees independent of subsystem S’

Principle of integrity
– S1 can establish communication channel with S2 such that S’ can’t corrupt or eavesdrop the channel
Address Spaces

Why in μ-kernel?
Must hide HW concept of address spaces to implement protection

- Start with one address space representing all physical memory
- Owner of an address space can grant, map, flush pages
- Include I/O ports & memory-mapped I/O

Figure 1: A Granting Example.
Threads, IPC & UIDs

• Why include Thread-concept in μ-kernel?
  – Prevent corruption of address spaces (all changes to a thread’s address space controlled by the kernel)

• Why include IPC in μ-kernel?
  – Enable cross-address-space communication
  – Also enable HW interrupts

• Why include unique-identifier (UID) generation in μ-kernel?
  – Required for naming message sources & destinations
Flexibility

- Memory Managers
- Pagers
- Real-time Application Resource Allocators
- Device Drivers
- L2 Caches & SW TLBs
- Remote IPCs
- Unix Servers

All but: processor architecture, registers, L1 Caches, first-level TLBs
Discussion: Summary Question #1

• **State the 3 most important things the paper says.** These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.
Performance: Kernel-User Switches

• Prior measurements: 900+ cycles
  – Entering kernel mode = 71 cycles
  – Rest (executing instructions, L1/TLB misses) = 800+ cycles
  – Returning to user mode = 36 cycles

• L3 μ-kernel: 123-180 cycles
  – At most 3 TLB misses & 10 cache misses
Performance: Address Space Switches

- Prior measurements: 864 cycles for TLB reload on Pentium
- L4 μ-kernel: < 50 cycles (see table)
  - Idea: use tagged TLB or segment registers

<table>
<thead>
<tr>
<th></th>
<th>TLB entries</th>
<th>TLB miss cycles</th>
<th>Page Table switch cycles</th>
<th>Segment cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>32</td>
<td>9...13</td>
<td>36...364</td>
<td>39</td>
</tr>
<tr>
<td>Pentium</td>
<td>96</td>
<td>9...13</td>
<td>36...1196</td>
<td>15</td>
</tr>
<tr>
<td>PowerPC 601</td>
<td>256</td>
<td>?</td>
<td>?</td>
<td>29</td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>40</td>
<td>20...50&lt;sup&gt;a&lt;/sup&gt;</td>
<td>80...1800</td>
<td>n/a</td>
</tr>
<tr>
<td>Mips R4000</td>
<td>48</td>
<td>20...50&lt;sup&gt;a&lt;/sup&gt;</td>
<td>0&lt;sup&gt;b&lt;/sup&gt;</td>
<td>n/a</td>
</tr>
</tbody>
</table>

<sup>a</sup>Alpha and Mips TLB misses are handled by software.
<sup>b</sup>R4000 has a tagged TLB.

Table 1: Address Space Switch Overhead
Performance: Thread Switches & IPC

- Prior measurements: 400-1450 μsecs per RPC
- L4 μ-kernel: 3.2-3.6 μsecs per RPC

<table>
<thead>
<tr>
<th>System</th>
<th>CPU, MHz</th>
<th>RPC time (round trip)</th>
<th>cycles/IPC (oneway)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3</td>
<td>486, 50</td>
<td>10 μs</td>
<td>250</td>
</tr>
<tr>
<td>QNX</td>
<td>486, 33</td>
<td>76 μs</td>
<td>1254</td>
</tr>
<tr>
<td>Mach</td>
<td>R2000, 16.7</td>
<td>190 μs</td>
<td>1584</td>
</tr>
<tr>
<td>SRC RPC</td>
<td>CVAX, 12.5</td>
<td>464 μs</td>
<td>2900</td>
</tr>
<tr>
<td>Mach</td>
<td>486, 50</td>
<td>230 μs</td>
<td>5750</td>
</tr>
<tr>
<td>Amoeba</td>
<td>68020, 15</td>
<td>800 μs</td>
<td>6000</td>
</tr>
<tr>
<td>Spin</td>
<td>Alpha 21064, 133</td>
<td>102 μs</td>
<td>6783</td>
</tr>
<tr>
<td>Mach</td>
<td>Alpha 21064, 133</td>
<td>104 μs</td>
<td>6916</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>CPU, MHz</th>
<th>RPC time</th>
<th>cycles/IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exo-tlrmach</td>
<td>R2000, 16.7</td>
<td>6 μs</td>
<td>53</td>
</tr>
<tr>
<td>Spring</td>
<td>SparcV8, 40</td>
<td>11 μs</td>
<td>220</td>
</tr>
<tr>
<td>DP-Mach</td>
<td>486, 66</td>
<td>16 μs</td>
<td>528</td>
</tr>
<tr>
<td>LRPC</td>
<td>CVAX, 12.5</td>
<td>157 μs</td>
<td>981</td>
</tr>
</tbody>
</table>

Table 2: 1-byte-RPC performance
Performance: Memory Effects

• Prior measurements: Mach μ-kernel 1.25-2.31x slower than Ultrix
  – On sed, egrep, yacc, gcc, compress, espresso, Andrew benchmark

• L4 μ-kernel:
  – Comparable when ignore system cache misses
  – Comparable on conflict misses
  – Small working sets of L4 => few capacity misses

Figure 3: Baseline MCPI for Ultrix and Mach.

Figure 4: MCPI Caused by Cache Misses.
• **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.
Take Aways

- With the right abstraction, \( \mu \)-kernel can be efficient & flexible
- \( \mu \)-kernel must be tailored to processor characteristics
- L4 achieves good performance via processor-specific implementations of processor-independent abstractions

L4 kernels have been deployed on billions of Qualcomm mobile devices & in safety-critical systems
Aside: Clans & Chiefs

• L4’s mandatory access control policy: requires a mechanism for mediating and authorizing communication

• Processes are organized in a hierarchy of “clans”, each with a designated “chief”
  – Inside the clan, all messages are transferred freely & the kernel guarantees message integrity
  – Messages crossing a clan boundary (incoming/outgoing) are redirected to the clan’s chief, who controls the flow of messages

• Later replaced by capability-mediated access control to endpoint
“Safe Kernel Extensions Without Run-Time Checking”

George C. Necula, Peter Lee 1996

- George Necula (CMU PhD)
  - Retired Professor UC Berkeley, working at Google
  - ACM Grace Hopper Award
  - “Test of Time” award for POPL’97 & POPL’02 papers

- Peter Lee (CMU)
  - Corporate Vice President of Microsoft Healthcare
  - Former CMU CSD Dept Head
  - “Test of Time” award for POPL’96 & POPL’97 papers
  - ACM Fellow, National Academy of Medicine
This paper introduced the notion of proof carrying code (PCC) and showed how it could be used for ensuring safe execution by kernel extensions without incurring run-time overhead. PCC turns out to be a general approach for relocating trust in a system; trust is gained in a component by trusting a proof checker (and using it to check a proof the component behaves as expected) rather than trusting the component per se. PCC has become one of the cornerstones of language-based security.

"Safe Kernel Extensions Without Run-Time Checking"
George C. Necula, Peter Lee 1996
Overview

• Mechanism for an OS kernel to determine with certainty that it is safe to execute a binary from an untrusted source

• “Proof-carrying code”
  – Binary contains a (tamper-proof) formal proof that the code obeys the kernel’s published safety policy
  – Kernel validates proof w/o using cryptography or consulting any external trusted entity

• Main advantage: No run-time checking

• Main practical difficulty: Generating the safety proofs
Overview of Proof-Carrying Code

- Source Program
  - Compilation & Certification
    - PCC
      - Native Code
      - Safety Proof
    - Code Producer
      - User Process
      - Untrusted Client
  - Code Consumer
    - OS Kernel
    - Network Server
- Proof Validation
  - Enable
  - CPU
Application: Network Packet Filters

Safety Policy

1. Memory reads restricted to packet & statically-allocated scratch memory
2. Memory writes limited to scratch memory
3. All branches are forward
4. Reserved & callee-saved registers are not modified

Precondition Rules

1. Packet length is $\geq 64$ bytes and $< 2^{32}$ bytes
2. OK to read each byte in packet
3. OK to write each byte in 16-byte (aligned) scratch memory
4. No aliasing between packet and scratch memory
Average Per-Packet Run Time

- **BPF**: BSD Packet Filter, encode filter in Domain-specific language & run in BPF interpreter that does static checks
- **M3-VIEW**: Use type-safe Modula-3, some run-time checks
- **SFI**: Software Fault Isolation, run-time mem safety checks
Startup Cost Amortization

<table>
<thead>
<tr>
<th>Packet Filter</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>8</td>
<td>15</td>
<td>47</td>
<td>28</td>
</tr>
<tr>
<td>Binary Size (bytes)</td>
<td>385</td>
<td>516</td>
<td>1024</td>
<td>814</td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (μs)</td>
<td>780</td>
<td>1070</td>
<td>2350</td>
<td>1710</td>
</tr>
<tr>
<td>Cost</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space (KB)</td>
<td>5.5</td>
<td>8.7</td>
<td>24.6</td>
<td>15.1</td>
</tr>
</tbody>
</table>
Discussion: Summary Question #1

• State the 3 most important things the paper says. These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.
Defining a (Kernel) Safety Policy

“The impatient reader may want to skip ahead to Section 3”

• Floyd-style verification-condition generator
  – Procedure that computes a “safety predicate” in first-order logic based on the code to be certified
  – Obtained by first specifying an “abstract machine” (operational semantics) that simulates the execution of safe programs

• Set of axioms used to validate the safety predicate

• Precondition: “calling convention” rules for kernel to invoke PCC binary
Abstract Machine for Memory-Safe DEC Alpha Machine Code

\[
\begin{align*}
\text{op} & ::= n \mid r_i \quad i \in 0 \ldots 10 \\
\text{al} & ::= \text{ADDQ} \mid \text{SUBQ} \mid \text{AND} \mid \text{OR} \mid \text{SLL} \mid \text{SRL} \\
\text{br} & ::= \text{BEQ} \mid \text{BNE} \mid \text{BGE} \mid \text{BLT} \\
\text{instr} & ::= \text{LDQ} \ r_d, n(r_s) \mid \text{STQ} \ r_s, n(r_d) \mid \text{al} \ r_s, op, r_d \mid \text{br} \ r_s, n \mid \text{RET}
\end{align*}
\]

\[
(\rho, pc) \rightarrow \begin{cases} 
(\rho[r_d \leftarrow r_s \oplus op], pc + 1), & \text{if } \Pi_{pc} = \text{ADDQ} \ r_s, op, r_d \\
(\rho[r_d \leftarrow \text{sel}(r_m, r_s \oplus n)], pc + 1), & \text{if } \Pi_{pc} = \text{LDQ} \ r_d, n(r_s) \text{ and } \text{rd}(r_s \oplus n) \\
(\rho[r_m \leftarrow \text{upd}(r_m, r_d \oplus n, r_s)], pc + 1), & \text{if } \Pi_{pc} = \text{STQ} \ r_s, n(r_d) \text{ and } \text{wr}(r_d \oplus n) \\
(\rho, pc + n + 1), & \text{if } \Pi_{pc} = \text{BEQ} \ r_s, n \text{ and } r_s = 0 \\
(\rho, pc + 1), & \text{if } \Pi_{pc} = \text{BEQ} \ r_s, n \text{ and } r_s \neq 0
\end{cases}
\]

\text{rd}(a) \text{ true iff } a \text{ aligned on an 8-byte boundary}
Certifying the Safety of Programs

\[
(r, pc) \rightarrow \begin{cases} 
(r_d \leftarrow r_s \oplus op, pc + 1), & \text{if } \Pi_{pc} = \text{ADDQ } r_s, op, r_d \\
(r_d \leftarrow \text{sel}(r_m, r_s \oplus n), pc + 1), & \text{if } \Pi_{pc} = \text{LDQ } r_d, n(r_s) \text{ and } \text{rd}(r_s \oplus n) \\
(r_m \leftarrow \text{upd}(r_m, r_d \oplus n, r_s), pc + 1), & \text{if } \Pi_{pc} = \text{STQ } r_s, n(r_d) \text{ and } \text{wr}(r_d \oplus n) \\
(r, pc + n + 1), & \text{if } \Pi_{pc} = \text{BEQ } r_s, n \text{ and } r_s = 0 \\
(r, pc + 1), & \text{if } \Pi_{pc} = \text{BEQ } r_s, n \text{ and } r_s \neq 0 
\end{cases}
\]

Figure 3: The Abstract Machine.

\[
VC_{pc} = \begin{cases} 
VC_{pc+1}[r_d \leftarrow r_s \oplus op], & \text{if } \Pi_{pc} = \text{ADDQ } r_s, op, r_d \\
\text{rd}(r_s \oplus n) \land VC_{pc+1}[r_d \leftarrow \text{sel}(r_m, r_s \oplus n)], & \text{if } \Pi_{pc} = \text{LDQ } r_d, n(r_s) \\
\text{wr}(r_d \oplus n) \land VC_{pc+1}[r_m \leftarrow \text{upd}(r_m, r_d \oplus n, r_s)], & \text{if } \Pi_{pc} = \text{STQ } r_s, n(r_d) \\
(r_s = 0 \Rightarrow VC_{pc+n+1}) \land (r_s \neq 0 \Rightarrow VC_{pc+1}), & \text{if } \Pi_{pc} = \text{BEQ } r_s, n \\
\text{Post}, & \text{if } \Pi_{pc} = \text{RET}
\end{cases}
\]

Figure 4: The Verification-Condition Generator.

Safety predicate \((\Pi, Pre, Post) = \forall r_0 \ldots \forall r_{10} \forall r_m Pre \Rightarrow VC_0\)

i.e., Executing \(\Pi\) from any initial state satisfying \(Pre\) passes all checks & yields \(Post\)
Validating the Safety Proof

- Use PCC system’s simple theorem prover to generate proof of safety predicate
  - Use Edinburgh Logical Framework (LF), a simple typed lambda calculus

- Kernel computes safety predicate from the assembly code using the VC rules, then checks that the safety proof is valid proof of safety predicate
  - Proof validation = typechecking

Ensures safety even if assembly code or the proof is tampered with
• **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.
Discussion

• Can extend to codes with loops
  – via a table that maps each backward-branch to a loop invariant that is part of proof

• Proof size is a concern

• PCC advantages
  – Eliminates run-time checks
  – Tamper-proof
  – Safety for code written in any language
  – Safety properties beyond memory protection

Final Thought

“Ideas from PL are destined to become increasingly critical for robust & good-performing systems”
Wednesday

Kernels & Virtual Machines (II)

“seL4: Formal Verification of an OS Kernel”

“Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval”