Critique Of Microkernel Architectures

I’m not interested in making devices look like user-level. They aren’t, they shouldn’t, and microkernels are just stupid.  
Linus Torvalds

Is Linus right?

Microkernel Performance

• First generation µ-kernel systems exhibited poor performance when compared to monolithic UNIX implementations.  
  – particularly Mach, the best-known example
• Reasons are investigated by [Chen & Bershad, 1993]:
  – instrumented user and system code to collect execution traces
  – run on DECstation 5000/200 (25MHz R3000)
  – run under Ultrix and Mach with Unix server
  – traces fed to memory system simulator
  – analyse MCPI (memory cycles per instruction)
  – baseline MCPI (i.e. excluding idle loops)

ULTRIX VS. MACH MCPI

Interpretation

Observations:
• Mach memory penalty (i.e. cache missess or write stalls) higher
• Mach VM system executes more instructions than Ultrix (but has more functionality).

Claim:
• Degraded performance is (intrinsic?) result of OS structure.
• IPC cost (known to be high in Mach) is not a major factor [Bershad, 1992].

Assertions

1. OS has less instruction and data locality than user code.
   – System code has higher cache and TLB miss rates.
   – Particularly bad for instructions.

Assertions

2. System execution is more dependent on instruction cache behaviour than is user execution
   – MCPIs dominated by system i-cache misses.
   – Note: most benchmarks were small, i.e. user code fits in cache.
3. Competition between user and system code is not a problem
   - Few conflicts between user and system caching.
   - TLB misses are not a relevant factor.
   - Note: the hardware used has direct-mapped physical caches.

   $\Rightarrow$ Split system/user caches wouldn’t help.

4. Self-interference is a problem in system instruction reference streams.
   - High internal conflicts in system code.
   - System would benefit from higher cache associativity.

5. System block memory operations are responsible for a large percentage of memory system reference costs.
   - Particularly true for I/O system calls.

6. Write buffers are less effective for system references.
   - Write buffer allows limited async. writes on cache misses.

7. Virtual to physical mapping strategy can have significant impact on cache performance.
   - Unfortunate mapping may increase conflict misses.
   - "Random" mappings (Mach) less likely to exhibit consistently poor performance.

4. assertions...

So, What’s Wrong?

- The MCPI for Mach is significantly higher than Ultrix.
- $\mu$-kernels heavily depend on IPC.
- IPC is expensive.
  - Is the $\mu$-kernel idea flawed?
  - Should some code never leave the kernel?
  - Do we have to buy flexibility with performance?

A Critique Of The Critique

- Data presented earlier:
  - are specific to one (or a few) system.
  - results cannot be generalised without thorough analysis.
  - no such analysis has been done.

  $\Rightarrow$ Cannot trust the conclusions [Liedkte, 1995].
Re-analysis Of Chen & Bershad’s Data

Conclusion
- Mach system (kernel + UNIX server + emulation library) is too big!
- UNIX server is essentially same.
- Emulation library is irrelevant (according to Chan & Bershad).
  ⇒ Mach μ-kernel working set is too big

Can we build μ-kernels which avoid these problems?

Requirements For Microkernels:
- Fast (system call costs, IPC costs)
- Small (big ⇒ slow)
  ⇒ Must be well designed, providing a minimal set of operations.

Can this be done?

Are High System Costs Essential?
- Example: kernel call cost on i486
  - Mach kernel call: 900 cycles
  - Inherent (hardware-dictated cost): 107 cycles.
    ⇒ 800 cycles kernel overhead.
  ⇒ Mach’s performance is a result of design and implementation not the μ-kernel concept!

Microkernel Design Principles (Liedtke)
- Minimality: If it doesn’t have to be in the kernel, it shouldn’t be in the kernel
  - Security is the only case for must be in the kernel
- Appropriate abstractions which can be made fast and allow efficient implementation of services
- Well written: It pays to shave a few cycles off TLB refill handler or the IPC path
- Unportable: must be targeted to specific hardware
  - no problem if it’s small, and higher layers are portable
  - Example: Liedtke reports significant rewrite of memory management when porting from 486 to Pentium
    ⇒ “abstract hardware layer” is too costly
Non-Portability Example: 
I486 vs Pentium:

- Size and associativity of TLB
- Size and organisation of cache (larger line size - restructured IPC)
- Segment regs in Pentium used to simulate tagged TLB

⇒ different trade-offs

What must a µ-Kernel provide?

- Virtual memory/address spaces
- threads,
- fast IPC,
- unique identifiers (for IPC addressing).

µ-Kernel does not have to provide:

- file system
  - use user-level server (as in Mach)
- device drivers
  - user-level driver invoked via interrupt (= IPC)
- page-fault handler
  - use user-level pager

L4 Implementation Techniques

- Appropriate system calls to reduce number of kernel invocations (e.g., reply & receive next)
- Rich message structure
  - value and reference parameters in message
  - Copy message only once (i.e., not user!kernel!user)
- Short messages in registers
- Copy message only once (i.e., not user!kernel!user)
- One kernel stack (for interrupt handling) per thread (in TCB)
- TCBs in (mapped) VM, cache-friendly layout
- Fast IPC, unique identifiers (for IPC addressing).

- Many HW specific optimisations

Case In Point: L4Linux [Härtig et al., 1997]

- Port of Linux kernel to L4 (like Mach Unix server)
  - single-threaded (for simplicity, not performance)
  - is pager of all Linux user processes
  - maps emulation library and signal-handling code into AS
  - server AS maps physical memory (& Linux runs within)
  - copying between user and server done on physical memory
  - use software lookup of page tables for address translation

- Changes to Linux restricted to architecture-dependent part
- Duplication of page tables (L4 and Linux server)
  - Binary compatible to native Linux via trampoline mechanism
  - but also modified libc with RPC stubs

L4Linux Overview
Server Internals

- L4 threads used to
  - receive device interrupts
  - Emulated Linux's bottom half handling
  - Receive system calls from applications

Signal Delivery In L⁴Linux

- Separate signal-handler thread in each user process
  - server IPCs signal-handler thread
  - handler thread ex regs main user thread to save state
  - user thread IPCs Linux server
  - server does signal processing
  - server IPCs user thread to resume

L⁴Linux Performance

Microbenchmarks:

<table>
<thead>
<tr>
<th>System</th>
<th>Time [µs]</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>1.68</td>
<td>223</td>
</tr>
<tr>
<td>L⁴Linux</td>
<td>3.95</td>
<td>526</td>
</tr>
<tr>
<td>L⁴Linux (trampoline)</td>
<td>5.66</td>
<td>753</td>
</tr>
<tr>
<td>MkLinux in-kernel</td>
<td>15.66</td>
<td>2060</td>
</tr>
<tr>
<td>MkLinux server</td>
<td>110.60</td>
<td>14710</td>
</tr>
</tbody>
</table>

`getpid()` on 133MHz Pentium

Cycle Breakdown

<table>
<thead>
<tr>
<th>Client</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>enter emulation lib</td>
<td>20</td>
</tr>
<tr>
<td>send syscall message</td>
<td>165</td>
</tr>
<tr>
<td>receive reply</td>
<td>188</td>
</tr>
<tr>
<td>leave emulation lib</td>
<td>19</td>
</tr>
</tbody>
</table>

`wait for msg` 131 cycles
`Linux kernel` 555 cycles (+16.6%)
`send reply` 605 cycles (+27.1%)

Macrobenchmarks: LMBENCH

Macrobenchmarks: Kernel Compile
Conclusion

- Mach sux ⇒ microkernels suck
- L4 shows that performance might be deliverable
  - L^4Linux gets close to monolithic kernel performance
  - need real multi-server system to evaluate µ-kernel potential
- Jury is still out!
- Mach has prejudiced community (see Linus...)
  - It'll be an uphill battle!

Implementations

<table>
<thead>
<tr>
<th>API</th>
<th>Kernel</th>
<th>Who</th>
<th>Language</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td>L4/x86</td>
<td>Liedtke</td>
<td>asm</td>
<td>x86</td>
</tr>
<tr>
<td></td>
<td>L4/MIPS</td>
<td>UNSW</td>
<td>asm/C</td>
<td>R4k</td>
</tr>
<tr>
<td></td>
<td>L4/Alpha</td>
<td>UNSW/Dres</td>
<td>PAL/C</td>
<td>21x64</td>
</tr>
<tr>
<td></td>
<td>Fiasco</td>
<td>Dresden</td>
<td>C++</td>
<td>x86</td>
</tr>
<tr>
<td>X.0</td>
<td>L4/x86</td>
<td>Liedtke</td>
<td>asm</td>
<td>x86</td>
</tr>
<tr>
<td></td>
<td>Hazelnut</td>
<td>Karlsruhe</td>
<td>C</td>
<td>x86, ARM</td>
</tr>
<tr>
<td>V4</td>
<td>Pistachio</td>
<td>UNSW</td>
<td>C++</td>
<td>x86, IA-64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PPC-32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MIPS, Alpha</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ARM, PPC-64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SPARC (I.p.)</td>
</tr>
</tbody>
</table>