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
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 $\mu$-kernel systems exhibited poor performance when compared to monolithic UNIX implementations.
  - particularly Mach, the best-known example
MICROKERNEL PERFORMANCE

- First generation $\mu$-kernel systems exhibited poor performance when compared to monolithic UNIX implementations. particularly Mach, the best-known example
- Reasons are investigated by [Chen & Bershad 93]:
  - 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)
INTERPRETATION

Observations:

- Mach memory penalty (i.e. cache misses or write stalls) higher
- Mach VM system executes more instructions than Ultrix (but has more functionality).
**INTERPRETATION**

**Observations:**

- Mach memory penalty (i.e. cache misses 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.
INTERPRETATION

Observations:

• Mach memory penalty (i.e. cache misses 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 [Ber92].
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.
1 OS has less instruction and data locality than user code.
   ➔ System code has higher cache and TLB miss rates.
   ➔ Particularly bad for instructions.

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.


**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.

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.
   - Split system/user caches wouldn’t help.
**SELF INTERFERENCE**

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>sed+U</td>
<td>0.11</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.24</td>
</tr>
<tr>
<td>egrep+U</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.02</td>
</tr>
<tr>
<td>yacc+U</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.06</td>
</tr>
<tr>
<td>gcc+U</td>
<td>0.07</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.23</td>
</tr>
<tr>
<td>compress+U</td>
<td>0.05</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.13</td>
</tr>
<tr>
<td>ab+U</td>
<td>0.10</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.21</td>
</tr>
<tr>
<td>espresso+U</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.02</td>
</tr>
<tr>
<td>lisp+U</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.00</td>
</tr>
<tr>
<td>eqn:ott+U</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.00</td>
</tr>
<tr>
<td>fppp+U</td>
<td>0.14</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.01</td>
</tr>
<tr>
<td>doduc+U</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.00</td>
</tr>
<tr>
<td>liv+U</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.04</td>
</tr>
<tr>
<td>tomcatv+U</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>+M</td>
<td>0.00</td>
</tr>
</tbody>
</table>

- Only examine system cache misses.
- Shaded: System cache misses removed by associativity.
- MCPI for system-only, using R3000 direct-mapped cache.
- Reductions due to associativity were obtained by running system on a simulator and using a two-way associative cache of the same size.
4 Self-interference is a problem in system instruction reference streams.

- High internal conflicts in system code.
- System would benefit from higher cache associativity.
ASSERTIONS...

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.
ASSERTIONS...

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 asynch. writes on cache misses
ASSERTIONS...

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 asynch. 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) are to be avoided.
OTHER EXPERIENCE WITH $\mu$-KERNEL PERFORMANCE

- System call costs are (inherently?) high.
  ➔ Typically hundreds of cycles, 900 for Mach/i486.
- Context (address-space) switching costs are (inherently?) high.
  ➔ Getting worse (in terms of cycles) with increasing CPU/memory speed ratios [Ous90].
  ➔ IPC (involving system calls and context switches) is inherently expensive.
So, what’s wrong?
So, what’s wrong?

- $\mu$-kernels heavily depend on IPC
- IPC is expensive
**So, what’s wrong?**

- $\mu$-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.

⇒ Cannot trust the conclusions [Lie95].
Re-analysis of Chen & Bershad’s Data

Other MCPI
System cache miss MCPI

<table>
<thead>
<tr>
<th>Command</th>
<th>User Time</th>
<th>System Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sed</td>
<td>0.227</td>
<td>0.495</td>
</tr>
<tr>
<td>M</td>
<td>0.427</td>
<td>0.534</td>
</tr>
<tr>
<td>egrep</td>
<td>0.035</td>
<td>0.081</td>
</tr>
<tr>
<td>M</td>
<td>0.067</td>
<td>0.129</td>
</tr>
<tr>
<td>yacc</td>
<td>0.067</td>
<td>0.129</td>
</tr>
<tr>
<td>gcc</td>
<td>0.434</td>
<td>0.690</td>
</tr>
<tr>
<td>compress</td>
<td>0.250</td>
<td>0.418</td>
</tr>
<tr>
<td>ab</td>
<td>0.427</td>
<td>0.534</td>
</tr>
<tr>
<td>espresso</td>
<td>0.041</td>
<td>0.068</td>
</tr>
</tbody>
</table>

MCPI for Ultrix and Mach

MCPI for Ultrix and Mach

Overhead is mostly in the system

1
MCPI caused by cache misses: conflict (black) vs capacity (white)
CONCLUSION

- Mach system (kernel + UNIX server + emulation library) is too big!
CONCLUSION

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

Can we build $\mu$-kernels which avoid these problems?
**Requirements for μ-kernels:**

- 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
Are high system costs essential?

- Example: kernel call cost on i486
  * Mach kernel call: 900 cycles
  * Inherent (hardware-dictated cost): 107 cycles.
  \[\Rightarrow\] 800 cycles kernel overhead.
Are high system costs essential?

- Example: kernel call cost on i486
  - Mach kernel call: 900 cycles
  - Inherent (hardware-dictated cost): 107 cycles.
    \[ \Rightarrow \text{800 cycles kernel overhead.} \]
**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 \( \mu \)-kernel concept!**
μ-kernel Design Principles

Minimality: If it doesn’t **have to be** in the kernel, it **shouldn’t** 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).
**What must a $\mu$-Kernel provide?**

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

**$\mu$-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*
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
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)
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
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
- As many syscall parameters in registers as possible
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
- As many syscall parameters in registers as possible
- One kernel stack (for interrupt handling) per thread (in TCB)
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
- As many syscall parameters in registers as possible
- One kernel stack (for interrupt handling) per thread (in TCB)
- TCBs in (mapped) VM, cache-friendly layout
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
• As many syscall parameters in registers as possible
• One kernel stack (for interrupt handling) per thread (in TCB)
• TCBs in (mapped) VM, cache-friendly layout
• Thread UIDs (containing thread ID)
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
- As many syscall parameters in registers as possible
- One kernel stack (for interrupt handling) per thread (in TCB)
- TCBs in (mapped) VM, cache-friendly layout
- Thread UIDs (containing thread ID)
- “Hottest” kernel code is shortest
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
- As many syscall parameters in registers as possible
- One kernel stack (for interrupt handling) per thread (in TCB)
- TCBs in (mapped) VM, cache-friendly layout
- Thread UIDs (containing thread ID)
- “Hottest” kernel code is shortest
- Kernel IPC code on single page, critical data on single page
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
- As many syscall parameters in registers as possible
- One kernel stack (for interrupt handling) per thread (in TCB)
- TCBs in (mapped) VM, cache-friendly layout
- Thread UIDs (containing thread ID)
- “Hottest” kernel code is shortest
- Kernel IPC code on single page, critical data on single page
- Many H/W specific optimisations
## Performance

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>MHz</th>
<th>RPC $\mu$s</th>
<th>cyc/IPC</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4</td>
<td>R4600</td>
<td>100</td>
<td>1.7 $\mu$s</td>
<td>100</td>
<td>full</td>
</tr>
<tr>
<td>L4</td>
<td>Alpha</td>
<td>433</td>
<td>0.2 $\mu$s</td>
<td>45</td>
<td>full</td>
</tr>
<tr>
<td>L4</td>
<td>Pentium</td>
<td>166</td>
<td>1.5 $\mu$s</td>
<td>121</td>
<td>full</td>
</tr>
<tr>
<td>L4</td>
<td>486</td>
<td>50</td>
<td>10 $\mu$s</td>
<td>250</td>
<td>full</td>
</tr>
<tr>
<td>QNX</td>
<td>486</td>
<td>33</td>
<td>76 $\mu$s</td>
<td>1254</td>
<td>full</td>
</tr>
<tr>
<td>Mach</td>
<td>R2000</td>
<td>16.7</td>
<td>190 $\mu$s</td>
<td>1584</td>
<td>full</td>
</tr>
<tr>
<td>SCR RPC</td>
<td>CVAX</td>
<td>12.5</td>
<td>464 $\mu$s</td>
<td>2900</td>
<td>full</td>
</tr>
<tr>
<td>Mach</td>
<td>486</td>
<td>50</td>
<td>230 $\mu$s</td>
<td>5750</td>
<td>full</td>
</tr>
<tr>
<td>Amoeba</td>
<td>68020</td>
<td>15</td>
<td>800 $\mu$s</td>
<td>6000</td>
<td>full</td>
</tr>
<tr>
<td>Spin</td>
<td>Alpha 21064</td>
<td>133</td>
<td>102 $\mu$s</td>
<td>6783</td>
<td>full</td>
</tr>
<tr>
<td>Mach</td>
<td>Alpha 21064</td>
<td>133</td>
<td>104 $\mu$s</td>
<td>6916</td>
<td>full</td>
</tr>
<tr>
<td>Exo-tlrpc</td>
<td>R2000</td>
<td>116.7</td>
<td>6 $\mu$s</td>
<td>53</td>
<td>restricted</td>
</tr>
<tr>
<td>Spring</td>
<td>SparcV8</td>
<td>40</td>
<td>11 $\mu$s</td>
<td>220</td>
<td>restricted</td>
</tr>
<tr>
<td>DP-Mach</td>
<td>486</td>
<td>66</td>
<td>16 $\mu$s</td>
<td>528</td>
<td>restricted</td>
</tr>
<tr>
<td>LRPC</td>
<td>CVAX</td>
<td>12.5</td>
<td>157 $\mu$s</td>
<td>981</td>
<td>restricted</td>
</tr>
</tbody>
</table>
Case in Point: $L^4$Linux [Härtig et al. 97]

- 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
Case in Point: L⁴Linux [Härtig et al. 97]

- 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
**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
## Linux 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>
</tbody>
</table>

getpid() on 133MHz Pentium
### L$^4$Linux Performance

**MICROBENCHMARKS:**

<table>
<thead>
<tr>
<th>System</th>
<th>Time [$\mu$s]</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>1.68</td>
<td>223</td>
</tr>
<tr>
<td>L$^4$Linux</td>
<td>3.95</td>
<td>526</td>
</tr>
<tr>
<td>L$^4$Linux (trampoline)</td>
<td>5.66</td>
<td>753</td>
</tr>
</tbody>
</table>

`getpid()` on 133MHz Pentium
# 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>2050</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>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>enter emulation lib</td>
<td>20</td>
<td>wait for msg</td>
</tr>
<tr>
<td>send syscall message</td>
<td>168</td>
<td>Linux kernel</td>
</tr>
<tr>
<td>receive reply</td>
<td>131</td>
<td>send reply</td>
</tr>
<tr>
<td>leave emulation lib</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

Hardware cost: 82 cycles
MACROBENCHMARKS: lmbench

write /dev/null [lat]
null process [lat]
simple process [lat]
/bin/sh process [lat]
mmap [lat]
2-proc context switch [lat]
8-proc context switch [lat]
pipe [lat]
UDP [lat]
RPC/UDP [lat]
TCP [lat]
RPC/TCP [lat]
pipe [bw⁻¹]
TCP [bw⁻¹]
file reread [bw⁻¹]
mmap reread [bw⁻¹]
MACROBENCHMARKS: LMBENCH

write /dev/null [lat]  
null process [lat]  
simple process [lat]  
/bin/sh process [lat]  
mmap [lat]  
2-proc context switch [lat]  
8-proc context switch [lat]  
pipe [lat]  
UDP [lat]  
RPC/UDP [lat]  
TCP [lat]  
RPC/TCP [lat]  
pipe [bw⁻¹]  
TCP [bw⁻¹]  
file reread [bw⁻¹]  
mmap reread [bw⁻¹]  

AIM Multiuser Benchmark Suite VII. Jobs completed per minute depending on AIM load units. (133 MHz Pentium)

MACROBENCHMARKS: KERNEL COMPILe

Linux  476 s
L₄Linux  506 s (+6.3%)
L₄Linux (trampo)  509 s (+6.9%)
MkLinux (kernel)  555 s (+16.6%)
MkLinux (user)  605 s (+27.1%)
Conclusions

• Mach sux $\nRightarrow$ microkernels suck
Conclusions

- Mach sux $\not\Rightarrow$ microkernels suck
- L4 shows that performance *might* be deliverable
  - $L^4$Linux gets close to monolithic kernel performance
  - need real multi-server system to evaluate $\mu$-kernel potential
Conclusions

• Mach sux $\not\Rightarrow$ microkernels suck
• L4 shows that performance *might* be deliverable
  $\Rightarrow$ L$^4$Linux gets close to monolithic kernel performance
  $\Rightarrow$ need real multi-server system to evaluate $\mu$-kernel potential
• Jury is still out!
• Mach has prejudiced community (see Linus...)
  $\Rightarrow$ It’ll be an uphill battle!
References


