**Motivation**

- Early operating systems had very little structure
- A strictly layered approach was promoted by Dijkstra
  - THE Operating System (Dijkstra)
- Later OS (more or less) followed this approach (e.g., Unix).
- Such systems are known as *monolithic kernels*.

**Issues of Monolithic Kernels**

**Advantages:**
- Kernel has access to everything:
  - all optimizations possible
  - all techniques/mechanisms/concepts implementable
- Kernel can be extended by adding more code, e.g., for:
  - new services
  - support for new hardware

**Problems:**
- Widening range of services and applications
  - OS bigger, more complex, slower, more error prone.
- Need to support same OS on different hardware.
- Like to support various OS environments.
- Distribution
  - Impossible to provide all services from same (local) kernel.

**Approaches to Tackling Complexity**

- Classical software-engineering approach: modularity
  - modularity, small, mostly self-contained components
  - well-defined interfaces between them
  - enforcement of interfaces
  - contains of faults to few modules

- Doesn’t work with monolithic kernels:
  - all kernel code executes in privileged mode
  - faults aren’t contained
  - interfaces cannot be enforced
  - performance takes priority over structure

**Evolution of the Linux Kernel**

Software-engineering study of Linux kernel [SUN+02]

- Looked at size and interdependencies of kernel “modules”
  - “common coupling”: interdependency via global variables
- Analyzed development over time (linearized version number)
- Result 1: Module size grows linearly with version number
Microkernel Idea: Break Up the OS

Evolution of the Linux Kernel — Part 2

Monolithic vs. Microkernel OS Structure

Monolithic OS vs. Microkernel OS Structure

Microkernel OS

Downcall vs. Upcall

- Kernel:
  - contains code which must run in supervisor mode
  - isolates hardware dependence from higher levels
  - is small and fast, extensible system
  - provides abstractions
- User-level servers:
  - are hardware independent/portable
  - provide "OS environment" ("OS personality"; maybe several)
  - may be invoked:
    - from application (via message passing IPC)
    - from kernel (application)
  - implement utilities (B/HK).

Software-engineering study of Linux kernel [SAYH02]
- Looked at size and interdependencies of kernel "modules"—"common coupling" vs. "dependency via global variables"
- Analyzed development over time (linear/log version number)
- Result 1: Module size grows linearly with version number
- Result 2: Interdependency grows exponentially with version!
- The present Linux model is doomed!
- There is no reason to believe that others are different—e.g., Windows, MacOS ...
- Need better software engineering in operating systems!
Microkernel-Based Systems

<table>
<thead>
<tr>
<th>classic</th>
<th>thin</th>
<th>specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Early Example: Hydra

- Separation of mechanisms from policy
  - e.g., protection vs. security
- No hierarchical layering of kernel
  - Protection, even within OS.
    - Uses (segregated) capabilities
- Objects, encapsulation, units of protection.
- Unique object name, no ownership.
- Object persistence based on reference counting [WCC’74]

Hydra...

- Can be considered the first object-oriented OS.
- Has been called the first microkernel OS
  - by people who ignored Birnh Haver.
- Has had enormous influence on later OS research.
- Was never widely used even at CMU because of
  - poor performance
  - lack of a complete environment

Popular Example: Mach

- Developed at CMU by Raimi and others [RY’80] from 1984
- Successor of Accent [FR’81] and RG [Rao’88]

Goals:

- Tailoring: support different OS interfaces
- Portability: almost all code HW independent
- Real-time capability
- Multiprocessor and distribution support
- Security

Coined term microkernel.

Basic Features of Mach Kernel

- Task and thread management
- Interprocess communication
  - asynchronous message-passing
- Memory object management
- System call redirection
- Device support
- Multiprocessor support

Mach Tasks and Threads

- Thread
  - active entity (basic unit of CPU utilization)
  - own stack, kernel scheduled
  - may run in parallel on multiprocessor
- Task
  - consists of one or more threads
  - provides address space and other environment
  - created from "blueprint"
  - empty or inherited address space
  - similar approach adopted by Linux
  - Activated by creating a thread in it
- *Privileged user-state program* may control scheduling
**Mach IPC: Ports**

- Addressing based on ports:
  - port is a mailbox allocated/deallocated via a system call
  - has a fixed-size message queue associated with it
  - is protected by (small) capabilities
  - has exactly one receiver, but possibly many senders

- Can have "hand-off" capability of a port
  - pass the receive capability for a port to another process
  - give up read access to the port

- Kernel detects ports without senders or receiver
  - Processes may have many ports (UNIX server has 2000!)
  - can be grouped into port sets
  - supports listening to many (similar to Unix sockets)

- Send blocks if queue is full
  - except with send-one cap (used for server replies)

**Mach IPC: Messages**

- Segregated capabilities:
  - threads refer to them via local indices
  - kernel maintains capabilities in messages
  - message format must identify caps

- Message contents:
  - Send capability to destination port (mandatory)
  - used by kernel to validate operation
  - optional send capability to reply port
  - for use by receiver to send reply
  - possibly other capabilities
  - "in-line" (by-value) data
  - "out-of-line" (by-reference) data, using copy-on-write,
    - may contain whole address spaces

**Mach Virtual Memory Management**

Address space constructed from memory regions

- Initially empty
  - populated by:
    - explicit allocation
    - inheriting from "blueprint" (as in Linux clone())
    - inherited: not shared or copied
    - allocated automatically by kernel during IPC

- when passing by-reference parameters
  - sparse virtual memory use (unlike UNIX)

**Copy-on-Write in Mach**

- When data is copied ("blueprint" or passed by-reference):
  - source and destination share single copy
  - both virtual pages are mapped to the same frame

- Marked as read-only
- When one copy is modified, a fault occurs
- Handling by kernel involves making a physical copy
  - VM mapping is changed to refer to the new copy

- Advantages
  - efficient way of sharing/pausing large amounts of data

- Drawbacks
  - expensive for small amounts of data (page-table manipulations)
  - data must be properly aligned

**Mach Address Maps**

- Address spaces represented as address maps
  - Any part of AS can be mapped to (part of) a memory object
  - Compact representation of sparse address spaces
  - Compare to multi-level page tables
Memory Objects

- Kernel doesn’t support file system
- Memory objects are an abstraction of secondary storage:
  - can be mapped into virtual memory
  - are cached by the kernel in physical memory
  - pager invoked if uncached page is touched
  - used by the system server to provide data
- Support data sharing
  - by mapping objects into several address spaces
- Memory is only cache for memory objects

Handling Page Faults

1. Check protection & locate memory object
   - use address map
2. Check cache, invoke page if cache miss
   - use a hashed page table
3. Check copy-on-write
   - perform physical copy if write fault
4. Enter new mapping into H/V page tables

Remote Communication

- Client A sends message to server B on remote node
  1. A sends message to local proxy port for B’s receive port
  2. User-level network message server receives from proxy port
  3. NIS converts proxy port into (global) network port
  4. NIS sends message to NIS on B’s node
     - may need conversion (byte order…)
  5. Remote NIS converts network port into local port (B’s)
  6. Remote NIS sends message to that port
- Note: networking built into kernel

Mach Unix Emulation

- Emulation library in user address space handles IPC
- Invoked by system call redirection (trampoline mechanisms)
  - supports binary compatibility
  - example of what’s now called pass-virtualization

Mach = Microkernel?

- Most OS services implemented at user level
  - using memory objects and external pages
  - Provides mechanisms, not policies
- Mostly hardware independent
- Big!
  - 140 system calls
  - Size: 200K instructions
- Performance poor
  - tendency to move features into kernel
  - OSF/IX
  - Darwin (base of MacOS X): complete BSD kernel inside Mach
- Further information on Mach: [YTR #87, CD#94, 5in97]
### Other Client-Server Systems

- **Lotus:** Most notable systems:
  - Notes: IBM (1984) [TM1, TM3, MT32]
  - Lotus Notes: Lotus 3 (1987)

- **Chorus:** IBM (France), early 1990s [DA40, RAA+92, RAF+92]
  - Bought by Sun a number of years back, closed down later

- **Quint:** First commercial microkernel (early 90s)
  - Highly successful in automotive

- **Green Hills Integrity**
  - 3T for military, commercial release 1992
  - market leader in aerospace, military

- **Windows NT:** Microsoft (early 1990s) [Core][1]
  - Early versions (NT 3) were microkernel-like
  - Now run main server and most drivers in kernel mode

### Critique of Microkernel Architectures

I’m not interested in making devices look like user-level.

They aren’t, they shouldn’t be, and microkernels are just stupid.

**Linux Torvalds**

Is Linux right?

### Microkernel Performance

- First generation microkernel systems (80s, early 90s)
  - exhibited poor performance when
    - compared to monolithic UNIX implementations
    - particularly Mach, the best-known example
    - but others weren’t better

- Reasons are investigated by (Chen & Bershad 93):
  - Instrumented user and system code to collect execution traces
  - ran on DECstation 5000/200 (25MHz, R3000)
  - run under Ultron and Mach with Unix server
    - traces fed to memory system simulator
    - analyze MCR (memory cycles per instruction)
    - baseline MCR (i.e., excluding idle loops)

### Ultron vs. Mach-Unix MCPI

![Ultron vs. Mach-Unix MCPI](image)

### Interpretation

**Observations:**

- Mach memory penalty higher
  - i.e. cache misses or write data

- Mach VM system executes more instructions than Ultron
  - but has more functionality

**Claim:**

- Degraded performance is intrinsic if result of OS structure

- IPC cost is not a major factor (Berthier)
  - IPC cost known to be high in Mach

### Assertions

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

2. System execution is more dependent on instruction cache behaviour than user execution
   - MCRs dominated by system ip instruction misses
   - No benchmark was small, i.e. user code fits in cache

3. Competition between user & system code is no problem
   - Few conflicts between user and system caching
   - TLB misses are not a relevant factor
   - Note: hardware used has direct-mapped physical caches
   - Split system/user caches wouldn’t help
Self-Interference

- Only examine system cache misses.
- Should system cache misses removed by associativity.
- MCPI for sparsely used, using RO exclusion cache.
- Reductions due to associativity were obtained by running system on a simulator and using a two-way associative cache of the same size.

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 asynchronous 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 Microkernel Performance

- System call costs are (inherently?) high.
  - Typical hundreds of cycles, 560 for Mach 4.0.
- Context (address-space) switching costs (inherently?) high.
  - Getting worse (in terms of cycles) with increasing CPU/memory speed ratios [Sekedu].
  - IPC (including system calls and context switches) is inherently expensive.
- Microkernels heavily depend on IPC.
- IPC is expensive.
  - Is the microkernel 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 generalized without thorough analysis.
  - no such analysis had been done.
  - Cannot trust the conclusions [Law93].

Re-Analysis of Chen & Bershad’s Data

MCPI for Ultras and Mach

Re-Analysis of Chen & Bershad’s Data

MCPI caused by cache misses: conflict (black) vs capacity (white).
Conclusion

- Mach system is too big
  - kernel + UNIX server = emulation library
- UNIX server is essentially same
- Emulation library is irrelevant (according to Chen & Bensheim)
- Mach kernel working set is too big

Can we build microkernels which avoid these problems?

Requirements for Microkernels

- Fast (system call costs, IPC costs)
- Small (big \( \rightarrow \) slow)
- Must be well designed
- Must provide a minimal set of operations

Can this be done?

- Example: kernel call cost on i486
  - Mach kernel call: 905 cycles
  - Inherent hardware-directed cost: 167 cycles
    - 905 cycles kernel overhead
  - 4-k kernel call: 123-180 cycles (15-73 cycle overhead)
- Mach's performance is a result of design and implementation
  - It is the result of the microkernel concept

Microkernel Design Principles [Lie96]

- 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: Linus reports significant overhead of memory management when porting from 486 to Pentium
  - Hardware abstraction 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

With the benefit of hindsight:

- Non-portability is not essential
  - Patching is proof
    - 30% architecture-independent code: all C/C++
    - performance worse than that of original 8086 assembler kernel

What Must a Microkernel Provide?

- Virtual memory/address spaces
  - required for protection
- Threads (or equivalent, eg scheduler activations)
  - as execution abstraction
- Fast IPC
- Unique identifiers (for IPC addressing)
  - actually, no: can use local names
  - as with shared memory:
    - "physical" identifiers only known to kernel
    - "mapped" into local name space

Microkernel Should Not Provide

- File system
  - user-level server (as in Mach)
- Device drivers
  - user-level driver invoked via interrupt (\( \rightarrow \) IPC)
- Page-fault handler
  - use user-level pager
L4 Implementation Techniques [Lieblke]

* Appropriate system calls to minimize # kernel invocations
  - e.g., reply & receive read
  - as many syscall args as possible in registers
* Efficient IPC
  - rich message structure
  - value and reference parameters in message
  - copy message only once (i.e. not user->kernel->user)
* Fast thread access
  - Thread LDTs (containing thread ID)
  - TCBs in (mapped) VM, cache-friendly layout
  - Separate kernel stack for each thread (red interrupt handling)
* General optimizations
  - "Hotspot" kernel code is shortened
  - Kernel IPC code on single page, critical data on single page
  - Many HW-specific optimizations

Microkernel Performance

| System | CPU Devices | IPCs | CPU Cycles | System
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L4 Alpha 21264</td>
<td>640</td>
<td>1,592</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>L4 Alpha 21364</td>
<td>640</td>
<td>1,592</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>L4 Alpha 21464</td>
<td>640</td>
<td>1,592</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>L4 Alpha 21564</td>
<td>640</td>
<td>1,592</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>L4 Alpha 21664</td>
<td>640</td>
<td>1,592</td>
<td>96</td>
<td></td>
</tr>
</tbody>
</table>

Pistachio IPC Performance

| Architecture | Optimization | INTRA AS | INTER AS | OPTIMIZED
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>48</td>
<td>29</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Alpha 21364</td>
<td>640</td>
<td>409</td>
<td>180</td>
<td>180</td>
</tr>
<tr>
<td>Alpha 21464</td>
<td>640</td>
<td>409</td>
<td>180</td>
<td>180</td>
</tr>
<tr>
<td>Alpha 21564</td>
<td>640</td>
<td>409</td>
<td>180</td>
<td>180</td>
</tr>
<tr>
<td>Alpha 21664</td>
<td>640</td>
<td>409</td>
<td>180</td>
<td>180</td>
</tr>
</tbody>
</table>

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

* Port of Linux kernel to L4 (like Mach Unix server)
  - single-threaded (for simplicity, not performance)
  - in-page of all/Linux user processes
  - kernel emulation library and signal-handling code into AS
  - server AS maps physical memory (L4 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 exn, main user thread to save state
  - user thread IPCs Linux server
  - does signal processing
  - server IPCs user thread to receive

L4Linux Performance: Microbenchmarks

<table>
<thead>
<tr>
<th>System</th>
<th>Time</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164 (Real Life)</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Alpha 21364 (Real Life)</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Super Linux</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Super Linux</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Super Linux</td>
<td>3.93</td>
<td>534</td>
</tr>
</tbody>
</table>

Cycle breakdown:

* Hardware costs:
  - 52 cycles (Linux Alpha Pentium)

<table>
<thead>
<tr>
<th>Branch</th>
<th>Time</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4Linux (Real Life)</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>Alpha 21164 (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Super Linux</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Super Linux</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Super Linux</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
<tr>
<td>Super Linux</td>
<td>3.93</td>
<td>534</td>
</tr>
<tr>
<td>L4Linux (Real Life)</td>
<td>5.95</td>
<td>116</td>
</tr>
</tbody>
</table>
L4Linux Performance

Microbenchmarks: Imbench

Macrobenchmarks: kernel compile

Conclusions

- Mach size: microkernels suck
- L4 shows that performance might be deliverable
  - L4Linux gets close to monolithic kernel performance
  - Need real multi-server system to evaluate microkernel potential
- Recent work substantially closer to native performance
  - NICTA Kernel, OK Linux
  - Microkernel-based systems can perform
- Mach has prejudiced community (see Linux...)
  - Getting microkernels accepted is still an uphill battle

Present State

- Microkernels deployed for years
  - QNX, Velocity
  - Military, aerospace, automotive
- OKL4 is now being deployed
  - Where performance matters
  - Mobile wireless devices
  - Qualcomm chips
  - Mobile phones
  - Estimated deployment: 12s of millions devices (August '07)
  - Estimated pipeline: 100s of millions devices in '08

Liedtke's Design Principles: What Stands?

- Minimality: definitely
  - Appropriate abstractions: yes
    - But no agreement about some of them
    - L4 API still developing
  - Well-written: absolutely
- Unportable: no
  - Patching is proof
    - But highly optimized IPC bad path (assembler)

How About Implementation Techniques?

- Appropriate system calls: page
  - Not probably less critical than thought
- Efficient IPC, rich message structure: less an
  - OKL4 has abandoned structured messages
  - Passing data in registers beneficial on some architectures
  - Single-copy, definitely wins
- Fast thread access: more (at least as propagated by Liedtke)
  - thread ID's more rise but are a security issue
  - Virtually-mapped base (parent) TCB array: x
    - Performance impact negligible (Kurose '03)
    - Waste address space
  - Per-thread kernel cache: x
    - Performance impact negligible (Watson '05)
  - Waste physical memory
    - Creates multiprocessor scalability issues