Microkernel Principles: Minimality

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. [SOSP'95]

- Advantages of resulting small kernel:
  - Easy to implement, port?
  - Easier to optimise
  - Hopefully enables a minimal trusted computing base (TCB)
  - Easier debug, maybe even prove correct?

- Challenges:
  - API design: generality despite small code base
  - Kernel design and implementation for high performance

Consequence of Minimality: User-level Services

- Kernel provides no services, only mechanisms
- Kernel is policy-free
  - Policies limit (good for 90% of cases, disastrous for some)
  - “General” policies lead to bloat, inefficiency

Hardware

VFS
IPC, file system
Scheduler, virtual memory
Device drivers, dispatcher

Kernel Mode

User Mode

Hardware

Application

Syscall

IPC

IPC performance is critical
1993 “Microkernel” IPC Performance

![Graph showing IPC performance over message length.]

L4 IPC Performance over 20 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>5.00</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>1997</td>
<td>Pentium</td>
<td>160</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>L4/MIPS</td>
<td>1997</td>
<td>R4700</td>
<td>100</td>
<td>0.86</td>
<td></td>
</tr>
<tr>
<td>L4/Alpha</td>
<td>1997</td>
<td>21064</td>
<td>433</td>
<td>0.10</td>
<td></td>
</tr>
<tr>
<td>Hazelnut</td>
<td>2002</td>
<td>Pentium 4</td>
<td>1,400</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>Pistachio</td>
<td>2005</td>
<td>Itanium</td>
<td>1,500</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>OKLA4</td>
<td>2007</td>
<td>XScale 255</td>
<td>400</td>
<td>0.64</td>
<td></td>
</tr>
<tr>
<td>NOVA</td>
<td>2010</td>
<td>i7 Bloomfield (32-bit)</td>
<td>2,660</td>
<td>0.11</td>
<td></td>
</tr>
<tr>
<td>seL4</td>
<td>2013</td>
<td>i7 Haswell (32-bit)</td>
<td>3,400</td>
<td>0.09</td>
<td></td>
</tr>
<tr>
<td>seL4</td>
<td>2013</td>
<td>ARM11</td>
<td>532</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>seL4</td>
<td>2013</td>
<td>Cortex A9</td>
<td>1,000</td>
<td>0.32</td>
<td></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-embedded</td>
<td>ARMv5</td>
<td>7.6</td>
<td>1.4</td>
<td>9.0</td>
</tr>
<tr>
<td>OKLA4 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>37.6</td>
</tr>
<tr>
<td>seL4</td>
<td>ARMv6</td>
<td>9.7</td>
<td>0.5</td>
<td>10.2</td>
</tr>
</tbody>
</table>

L4 Family Tree

![Diagram showing the family tree of L4 systems.]
L4 Deployments – in the Billions

Original L4 Design and Implementation

Implement. Tricks [SOSP’93]
- Process kernel
- Virtual TCB array
- Lazy scheduling
- Direct process switch
- Non-preemptible
- Non-portable
- Non-standard calling convention
- Assembler

Design Decisions [SOSP’95]
- Synchronous IPC
- Rich message structure, arbitrary out-of-line messages
- Zero-copy register messages
- User-mode page-fault handlers
- Threads as IPC destinations
- IPC timeouts
- Hierarchical IPC control
- User-mode device drivers
- Process hierarchy
- Recursive address-space construction

Objective: Minimise cache footprint and TLB misses

What Mechanisms?
- Fundamentally, the microkernel must abstract
  - Physical memory: Address spaces
  - CPU: Threads
  - Interrupts/Exceptions
- Unfettered access to any of these bypasses security
  - No further abstraction needed for devices
    - memory-mapping device registers and interrupt abstraction suffices
  - ...but some generalised memory abstraction needed for I/O space
- Above isolates execution units, hence microkernel must also provide
  - Communication (traditionally referred to as IPC)
  - Synchronization
Memory: Policy-Free Address-Space Management

- Kernel provides empty address-space “shell”
  - page faults forwarded to server
  - server provides mapping
  - AS layout is server policy (not kernel)
- Cost:
  - 1 round-trip IPC, plus mapping operation
    - mapping may be side effect of IPC
    - kernel may expose data structure
- Kernel mechanism: forwarding page-fault exception
- “External pagers” first appeared in Mach [Rashid et al., ’88]
  - but were optional (and slow) – in L4 there’s no alternative

Abstracting Memory: Address Spaces

- Minimum address-space abstraction: empty slots for page mappings
  - paging server can fill with mappings
    - virtual address → physical address + permissions
- Can be
  - page-table-like: array under full user control (traditional L4)
  - TLB-like: cache for mappings which may vanish (OKL4 Microvisor)
    - Less predictable performance – real-time?
- Main design decision: is source of a mapping a page or a frame?
  - Frame: hardware-like
  - Page: recursive address spaces (original L4 model)

Traditional L4: Recursive Address Spaces

- Mappings are page → page
- Magic initial AS to anchor recursion (map of PM)

Recursive Address Space Experience

API complexity: Recursive address-space model

- Conceptually elegant
  - trivially supports virtualization
- Drawback: Complex mapping database
  - kernel needs to track mapping relationship
    - Tear down dependent mappings on unmap
  - Mapping database problems:
    - accounts for 1/4–1/2 of kernel memory use
    - SMP coherence is performance bottleneck
- NICTA’s L4-embedded, OKL4 removed MDB
  - Map frames rather than pages
    - need separate abstraction for frames / physical memory
    - subsystems no longer virtualizable (even in OKL4 cap model)
- Properly addressed by seL4’s capability-based model
  - But have cap derivation tree, subject of on-going research
**Abstracting Execution**

- Can abstract as:
  - kernel-scheduled threads
    - Forces (scheduling) policy into the kernel
  - vCPUs or scheduler activations
    - This essentially virtualizes the timer interrupt through upcall
    - Scheduler activations also upcall for exceptions, blocking etc
    - Multiple vCPUs only for real multiprocessing
  - Threads can be tied to address space or “migrating”

- Implementation-wise not much of a difference
- ... but migrating thread requires kernel to provide/cache stacks
- Tight integration/interdependence with IPC model!

**Abstracting Interrupts and Exceptions**

- Can abstract as:
  - Upcall to interrupt/exception handler
    - hardware-like diversion of execution
    - need to save enough state to continue interrupted execution
  - IPC message to handler from magic “hardware thread”
    - OS-like
    - needs separate handler thread ready to receive

- Page fault tends to be special-cased for practical reason
  - Tends to require handling external to faulter
    - IPC message to page-fault server rather than exception handler
  - But also “self-paging” as in Nemesis [Hand ’99] or Barrelfish

**L4 IPC**

- **Rendezvous model**
  - Thread₁, Running | Thread₂, Blocked
  - Send (dest, msg) → Wait (src, msg)

  - Kernel executes in sender’s context
    - copies memory data directly to receiver (single-copy)
    - leaves message registers unchanged during context switch (zero copy)

**“Long” IPC**

- **LONG IPC ABANDONED**

  - IPC page faults are nested exceptions ⇒ In-kernel concurrency
  - L4 executes with interrupts disabled for performance, no concurrency
  - Must invoke untrusted usermode page-fault handlers
    - potential for DOSing other thread
  - Timeouts to avoid DOS attacks
    - complexity

- Why have long IPC?
  - POSIX-style APIs
    - write (fd, buf, nbytes)
  - Usually prefer shared buffers
Timeouts

- Limit IPC blocking time
- Thread 1: Running → Blocked → Running
- Thread 2: Blocked → Running

IPC Timeouts ABANDONED in sel4, QKL4
- No theory/heuristics for determining timeouts
- Typically server reply with zero TO, else \(\infty\)
- Added complexity
- Can do timed wait with timer syscall

Timed wait

Notifications

- IPC complemented with notifications
  - Delivers few bits (destructively)
  - Logically array of binary semaphores
  - Maps well to interrupts, exceptions multicore, ...

Thread can wait for IPC and notifications concurrently

Synchronous IPC Issues

- Worker_Th: Running → Blocked
- IO_Th: Blocked → Running

- Initiate_IO()...
- IO_Wait()...
- not generally possible

- Sync IPC forces multi-threaded code or select()
- Also poor choice for multi-core

Is IPC Redundant?

- Server: Blocked → Running
- Client: Blocked → Running

- 2 communication mechanisms: Minimality violation?

IPC is user-controlled context switch
- only makes sense intra-core
- fast control transfer
- mimics migrating threads
- enables scheduling context donation
  ✓ useful for real-time systems
Direct vs Indirect IPC Adressing

- **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

Endpoints and Notifications

- **Endpoint queues senders/receivers**
- **Does not buffer messages**

- **Notification accumulates bits**
- **Does not buffer**

IPC Destination Naming

- Original L4 addressed IPC to threads
- Client must do load balancing?
- Interpose transparently?
- Thread IDs replaced by IPC “endpoints” (ports)
- **Inefficient designs**
- **Poor information hiding**
- **Covert channels [Shapiro ‘02]**

Other Design Issues

- **IPC Control: “Clans & Chiefs”**
- **Process Hierarchy**
  - **Hierarchies replaced by delegatable cap-based access control**
  - **Inflexible, clumsy, inefficient hierarchies!**
  - **Fundamental problem: no rights delegation**

Implementation

Process Kernel: Per-Thread Kernel Stack

- Not worthwhile on modern processors!
- Stacks can dominate kernel memory use!
- Not worthwile on modern processors!
- Stacks can dominate kernel memory use!
- Easier to deal with blocking

Scheduler Optimisation Tricks: “Lazy Scheduling”

Problem: Unbounded scheduler execution time!

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

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

Virtual TCB Array

Fast TCB & stack lookup

Trades cache for TLB footprint and virtual address space

Not worthwhile on modern processors!
Scheduler Optimisation Tricks: “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: Lazy on unblocking instead on blocking

Speaking of Real Time...

- Kernel runs with interrupts disabled
  - No concurrency control ⇒ simpler kernel
  - Easier reasoning about correctness
  - Better average-case performance
- How about long-running system calls?
  - Use strategic preemption points
  - (Original) Fiasco has fully preemtible kernel
    - Like commercial microkernels (QNX, Green Hills INTEGRITY)

Incremental Consistency

Limited concurrency in kernel!

```
while (!done) {
    if (process_stuff();) {
        PSW.IRQ_disable=1;
        PSW.IRQ_disable=0;
    }
}
```

Scheduler Optimisation: “Direct Process Switch”

- Sender was running ⇒ had highest prio
  - If receiver prio ≥ sender then receive
    - Arguably, sender should donate back if it’s a sender replying to a server
    - Decrease always results in next Slice on Reply_Wait

Idea: Don’t invoke scheduler if you know who’ll be chosen

Problem:
- Accounting (RT systems)
- Policy

Implication: Time slice donation – receiver runs on sender’s time slice

- Frequent context switches in IPC-based systems
- Many scheduler invocations

```
while (!done) {
    if (process_stuff();) {
        PSW.IRQ_disable=1;
        PSW.IRQ_disable=0;
    }
}
```

Good fit to event kernel!

```
no concurrency in (single-core) kernel!
```

Limited concurrency in kernel!

```
while (!done) {
    process_stuff();
}
```

Consistency
Restartability
Progress

```
while (!done) {
    process_stuff();
}
```

Wrong way! Go back!
**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

**IPC Implementation**

**Simple send (e.g. as part of RPC-like “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)

**Difficult Example: Revoking IPC “Badge”**

**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!**

**Fastpath Coding Tricks**

- Reduces branch-prediction footprint
- Avoids mispredicts, stalls & flushes
- Uses ARM instruction predication
- But: increases slow-path latency
  - should be minimal compared to basic slow-path cost
Lazy FPU Switch

- FPU context tends to be heavyweight
  - e.g. 512 bytes FPU state on x86
- Only few apps use FPU (and those don’t do many syscalls)
  - saving and restoring FPU state on every context switch is wasteful!

Other implementation tricks

- Cache-friendly data structure layout, especially TCBs
  - data likely used together is on same cache line
  - helps best-case and worst-case performance
- Kernel mappings locked in TLB (using superpages)
  - helps worst-case performance
  - helps establish invariants: page table never walked when in kernel

Other Lessons Learned from 2nd Generation

- Programming languages:
  - original i4696 kernel ['95]: all assembler
  - UNSW MIPS and Alpha kernels ['96,'98]: half assembler, half C
  - Fiasco [TUD '98], Pistachio ['02]: C++ with assembler “fast path”
    - sel4 ['09], OKL4 ['09]: all C
- Lessons:
  - C++ sucks: code bloat, no real benefit
  - Changing calling conventions not worthwhile
    - Conversion cost in library stubs and when entering C in kernel
    - Reduced compiler optimization
  - Assembler unnecessary for performance
    - Can write C so compiler can produce near-optimal code
    - C entry from assembler changed, calling conventions maintained
    - sel4 performance: C++ path just as good as other L4 kernels
      - [Blackham & Heiser '12]

Lessons and Principles
Implement. Tricks [SOSP’93]

- Process kernel
- Virtual-TCB array
- Lazy scheduling
- Direct process switch
- Non-preemptible
- Non-portable
- Non-standard calling convention
- Assembler

Design Decisions [SOSP’95]

- Synchronous IPC
- Rich message structure, arbitrary out-of-line messages
- Zero-copy register messages
- User-mode page-fault handlers
- Threads as IPC destinations
- IPC timeouts
- Hierarchical IPC control
- User-mode device drivers
- Process hierarchy
- Recursive address space construction

seL4 Design Principles

- Fully delegatable access control
- All resource management is subject to user-defined policies
- Applies to kernel resources too!
- Suitable for formal verification
- Requires small size, avoid complex constructs
- Performance on par with best-performing L4 kernels
- Prerequisite for real-world deployment!
- Suitability for real-time use
  - Only partially achieved to date 😐
  - on-going work…

(Informal) Requirements for Formal Verification

- Verification scales poorly ⇒ small size (LOC and API)
- Conceptual complexity hurts ⇒ KISS
- Global invariants are expensive ⇒ KISS
- Concurrency difficult to reason about ⇒ single-threaded kernel

Largely in line with traditional L4 approach!

seL4 Fundamental Abstractions

- Capabilities as opaque names and access tokens
  - All kernel operations are cap invocations (except Yield())
- IPC:
  - Synchronous (blocking) message passing
  - Endpoint objects implemented as message queues
    - Send: get receiver TCB from endpoint or enqueue self
    - Receive: obtain sender’s TCB from endpoint or enqueue self
- Notifications:
  - Arrays of binary semaphores for lightweight synchronisation
- Other APIs:
  - Send()/Receive() to/from virtual kernel endpoint
  - Can interpose operations by substituting actual endpoint
- Fully user-controlled memory management

seL4’s main conceptual novelty!
Remember: seL4 User-Level Memory Management

Resource Manager | Resource Manager
-----------------|------------------
RM Data          | RM Data          
Addr Space       | Addr Space       
GRM Data         | GRM Data         

Global Resource Manager

Delegation can be revoked
Resources fully delegated, allows autonomous operation
Strong isolation, No shared kernel resources

Remaining Conceptual Issues in seL4

Time management
- Present scheduling model is ad-hoc and insufficient
- Fixed-prio round-robin forces policy
- Not sufficient for some classes of real-time systems (time triggered)
- No real support for most critical real-time scheduling
- Lack of an elegant resource management model for time

Multicore Model:
- What is the right kernel design (big lock, fine-grained locking, multikernel)?
- What is the role of IPC in multicore
- Does cross-core IPC make any sense?
- How does the RT scheduling model work on multicore?

Lessons From 20 Years of L4

- Minimality is excellent driver of design decisions
  - L4 kernels have become simpler over time
  - Policy-mechanism separation (user-mode page-fault handlers)
  - Device drivers really belong to user level
  - Minimality is key enabler for formal verification!

- IPC speed still matters
  - But not everywhere, premature optimisation is wasteful
  - Compilers have got so much better
  - Verification does not compromise performance
  - Verification invariants can help improve speed! [Shi, OOPSLA’13]

- Capabilities are the way to go
  - Details changed, but principles remained
  - Microkernels rock! (If done right!)