Microkernel Construction

seL4

Viktor Reusch, Matthias Hille, Till Miemietz, Nils Asmussen

06/11/2026

About this lecture

- First in a series on microkernel case studies
- Explain the basics of the *seL4* microkernel
- Compare seL4’s design to NOVA

About this lecture

- First in a series on microkernel case studies
- Explain the basics of the *seL4* microkernel
- Compare seL4's design to NOVA
- Notational shortcuts:
  - Advantage of a design decision: ⤊
  - Disadvantage of a design decision: ⤋
  - Same as in NOVA: =
  - Different compared to NOVA: ≠

Introduction [1], [2], [3]

- Formally verified microkernel
- Proof completed in 2009
- Research driven by Trustworthy Systems @ UNSW Sydney, Australia
- Capability-based =
- Kernel memory allocation managed by user space ≠

[4]

seL4 Concepts [5]

- Capabilities

Kernel Objects

seL4 Concepts [5]

- Capabilities

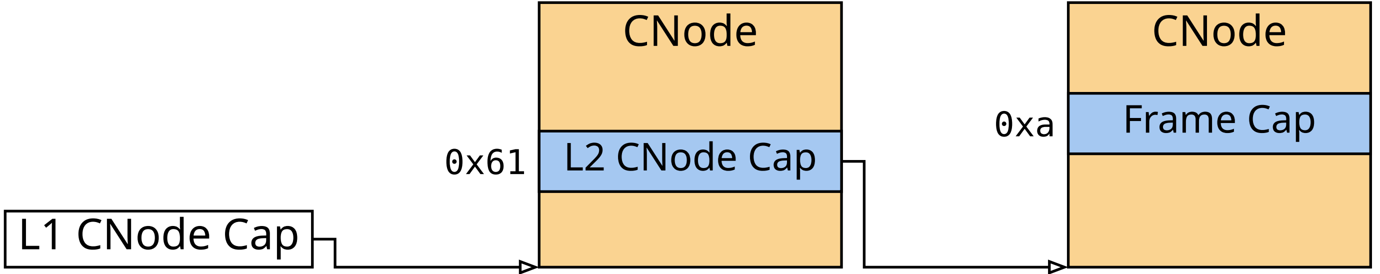
- Threads
- Capability spaces
- Page tables
- Frames
- Scheduling contexts
- Endpoints (synchronous IPC)
- Notifications (asynchronous signaling)
- Untyped memory (basis for memory management)

} Kernel Objects

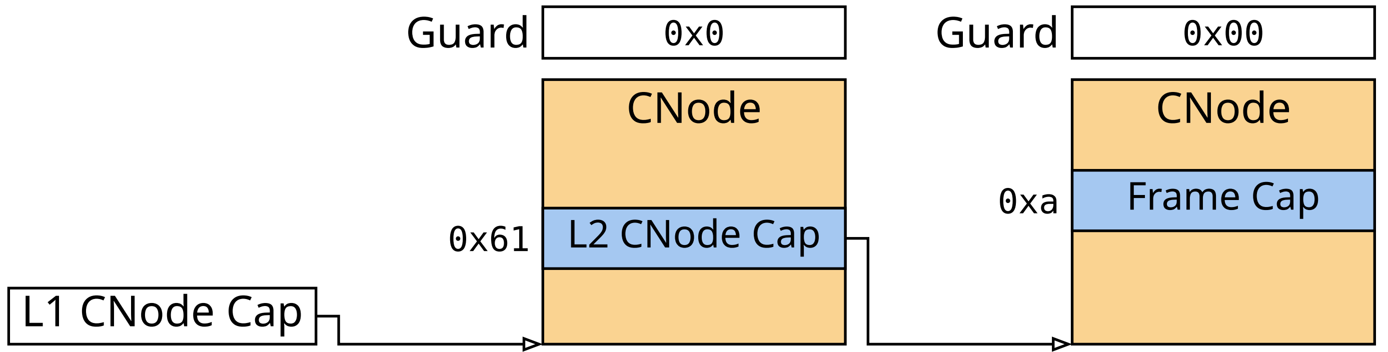
Capability Spaces [6]

- Capabilities stored in *CNodes*
- CNode is array containing capabilities
- A capability can point to another CNode
  $\longrightarrow$ enables construction of a hierarchy comparable to page tables
- Capability space (*CSpace*): all capabilities reachable from a given root CNode capability

Capability Addressing [6]

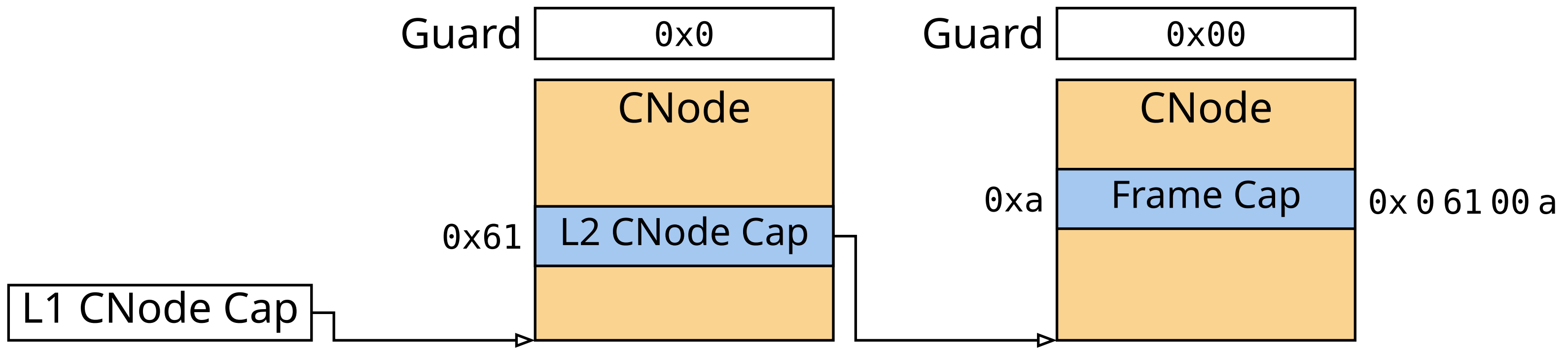


Capability Addressing [6]



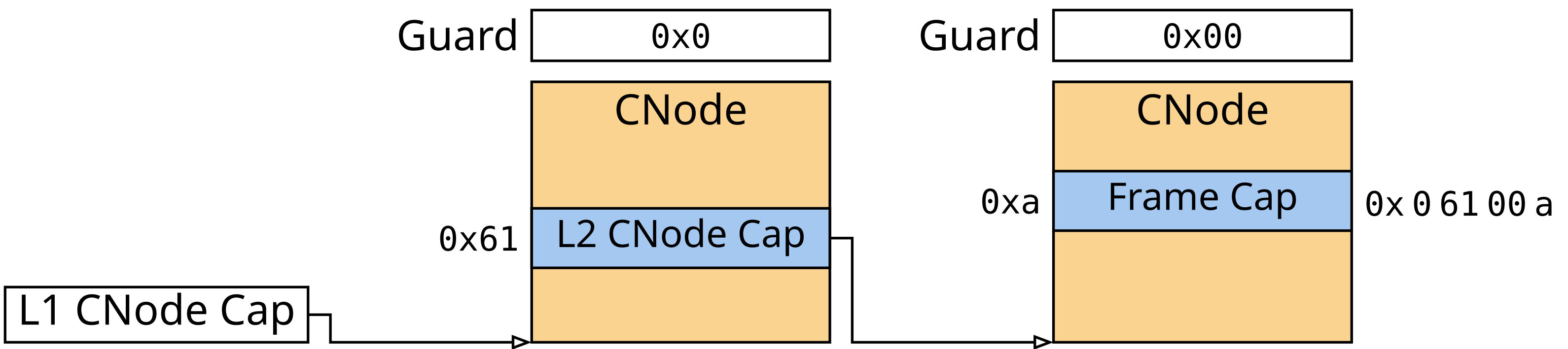
- Address resolution similar to guarded page tables

Capability Addressing [6]



- Address resolution similar to guarded page tables

Capability Addressing [6]



- Address resolution similar to guarded page tables
- To address CNode capabilities themselves: applications can specify depth limit

Capability Spaces / Addressing

- Capability space data structure similar to page tables ≠
- Applications manage capability space themselves ≠

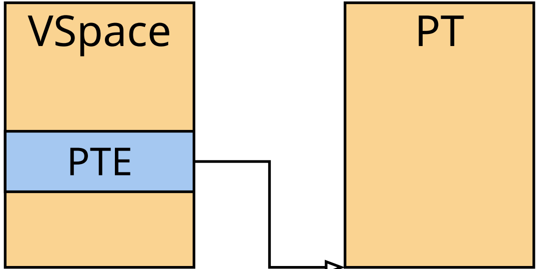
Capability Spaces / Addressing

- Capability space data structure similar to page tables ≠
- Applications manage capability space themselves ≠
- ⤊ No in-kernel management of capability lists
- ⤋ User has to construct the capability space
  - Estimate how many capability slots will be required
  - Create new CNodes when necessary
  - Purge empty CNodes when they are not needed anymore

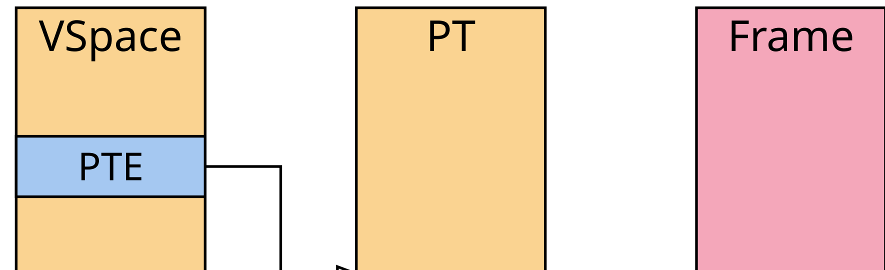
Address Spaces [7]

- *VSpace*: virtual address space of a thread
- Root of hierarchical page table objects
- User-space memory: *Frame* kernel objects ≠
- Applications insert frames into page tables via syscall

Address Space Figure

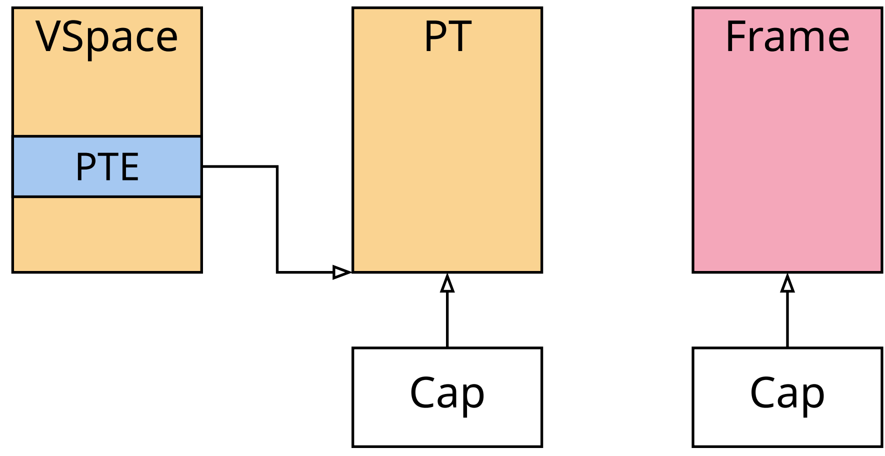


Address Space Figure



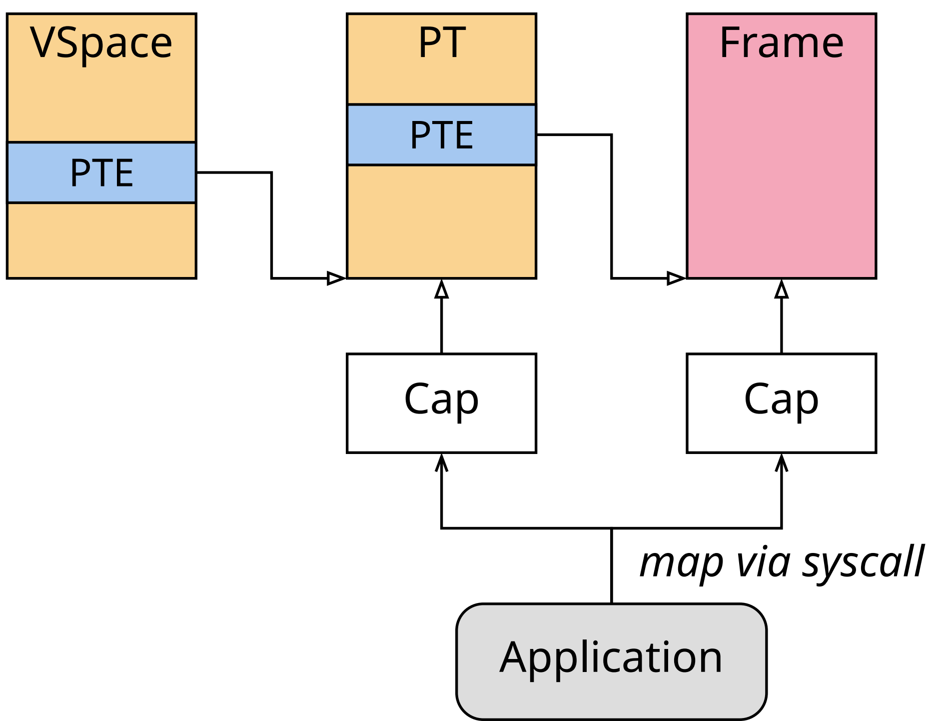
Application

Address Space Figure



Application

Address Space Figure



Address Spaces Continued

- Applications manage page table nesting themselves ≠
  - Individually map intermediate page tables into VSpace

- ⤊ Avoids implicitly allocating intermediated levels

- ⤋ Complicates user-space code

Untyped Memory [7], [8]

- Generic abstraction for managing memory
- Basis for all kinds of memory

Untyped Memory [7], [8]

- Generic abstraction for managing memory
- Basis for all kinds of memory
- Kernel manages untyped memory via simple bump allocator
- Untyped memory can be *retyped* to any kernel object, including:
  - Frames for user-space memory
  - Smaller portions of untyped memory (to give to child processes)
- Retyped objects available via new capabilities

Untyped Memory Figure

Untyped Memory Figure

Untyped Memory
TCB
Cap
Cap
retype via syscall
Application
Watermark

Untyped Memory Figure

Untyped Memory

Untyped Memory

TCB

Cap

Cap

Cap

retype via syscall

Application

Watermark

Untyped Memory Figure

Untyped Memory
Frame
Frame
Untyped Memory
TCB
Cap
Cap
Cap
Cap
Cap
Watermark
retype via syscall
Application

Untyped Memory Continued

- Memory management is fully controlled by applications ≠
- ⤊ No predetermined kernel memory; avoids resource exhaustion attacks
- ⤋ Complicated user space and more fragmentation

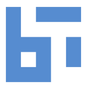
Threads [7], [9]

- *Thread Control Block* (*TCB*) links thread to a VSpace and a CSpace, both may be shared
- TCB contains IPC buffer (used for IPC and most system calls) =
- Thread may generate exceptions during runtime
  $\longrightarrow$ processed by an exception handler associated with a thread =

Scheduling [7]

- Enforcement implemented inside the kernel, strategy in user space
- Threads have priorities ≠
- Scheduler is preemptive and tickless with 256 priority levels
- Additionally possible to set budgets represented by scheduling objects =
- Passing on scheduling objects possible—donating execution time to services =

IPC [7], [10]



- Used for transferring small data and capabilities—“context switch with benefits” =
- Each IPC message has a *tag*:
  - Label (passed on as-is by kernel, used to specify service to call)
  - Message length
  - Number of capabilities to transfer
- Contents of IPC message placed in dedicated memory region (*IPC buffer*) =
- Receive: only a single CNode slot for storing one incoming capability
- Cross-core IPC is possible ≠ but discouraged

IPC Endpoints [7]

- Abstraction for exchanging IPC messages =
- Sending and receiving threads queue at an IPC endpoint
- Capabilities to endpoints can have badges (for identification =)
  - *Minting*: Assign badge to endpoint capability ≠
  - Badges cannot be changed or removed after establishing them

Notifications [7]

- For asynchronous (non-blocking) signals between threads
- Stores a *notification word* used as an array of binary semaphores ≠
- Capabilities to notifications can have badges
- Can be bound to TCB to also receive notifications while waiting/blocking on Endpoint

Notifications [7]

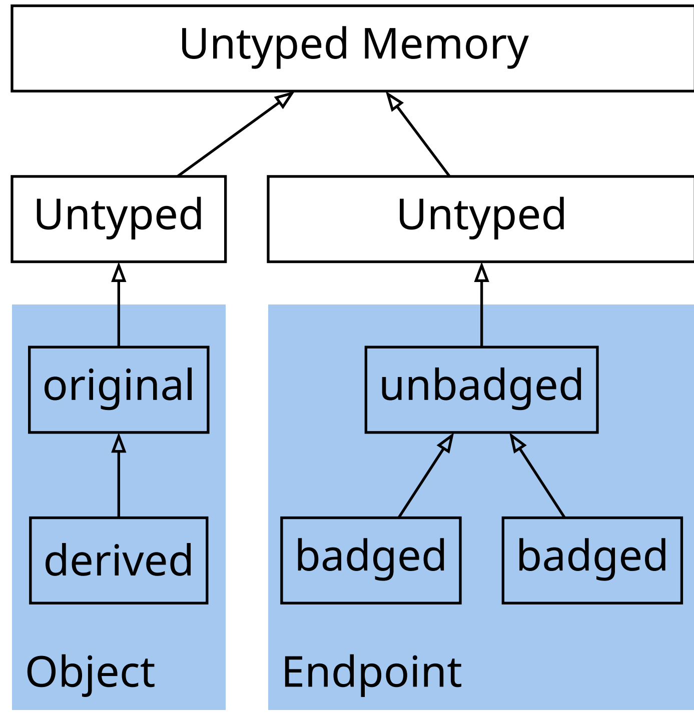
- For asynchronous (non-blocking) signals between threads
- Stores a *notification word* used as an array of binary semaphores ≠
- Capabilities to notifications can have badges
- Can be bound to TCB to also receive notifications while waiting/blocking on Endpoint
- Operations:
  - *Signal*
    - Updates notification word by bit-wise *or*-ing it with badge of signaler's notification capability
    - Unblocks first waiting thread
  - *Poll*: Non-blocking read of notification word
  - *Wait*: Blocking until notification word is non-zero; clears notification word

Capability Derivation Tree—CDT [6], [11]

- CDT contains child capabilities generated by *Copy*, *Mint*, *Retype*

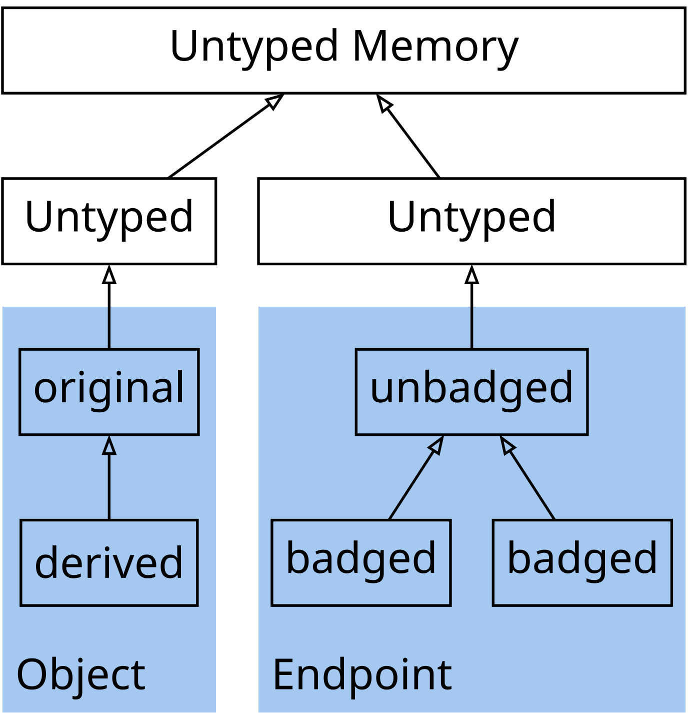
Capability Derivation Tree—CDT [6], [11]

- CDT contains child capabilities generated by *Copy*, *Mint*, *Retype*



Capability Derivation Tree—CDT [6], [11]

- CDT contains child capabilities generated by *Copy*, *Mint*, *Retype*
- Metadata required to implement revocation
- Internally: inheritance stored as doubly-linked list in capabilities



About seL4's Big Kernel Lock [12], [13], [14]

- seL4 uses a single big kernel lock for synchronization on SMP systems ≠

About seL4's Big Kernel Lock [12], [13], [14]

- seL4 uses a single big kernel lock for synchronization on SMP systems ≠
- ⤊ Friendly for future formal verification—currently only verified for uniprocessor
- ⤊ Implementation of kernel becomes much easier
- ⤋ Severely limits scalability and performance on large platforms

About seL4's Big Kernel Lock [12], [13], [14]

- seL4 uses a single big kernel lock for synchronization on SMP systems ≠
- ⤊ Friendly for future formal verification—currently only verified for uniprocessor
- ⤊ Implementation of kernel becomes much easier
- ⤋ Severely limits scalability and performance on large platforms
- Possible solution: use multikernel approach—one seL4 kernel instance per core

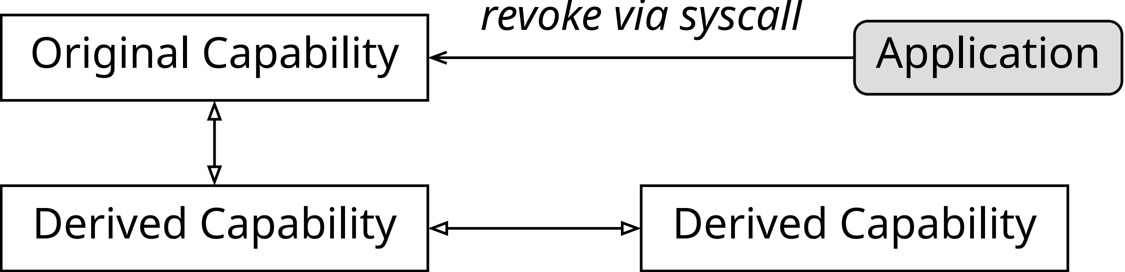
Interrupting Kernel Operations [9], [15]

- Real-time $\Leftarrow$ seL4 needs to handle interrupts promptly
  - Hardware interrupts disabled while in kernel ≠
  - Some system calls (e.g., capability revocation) can take long =

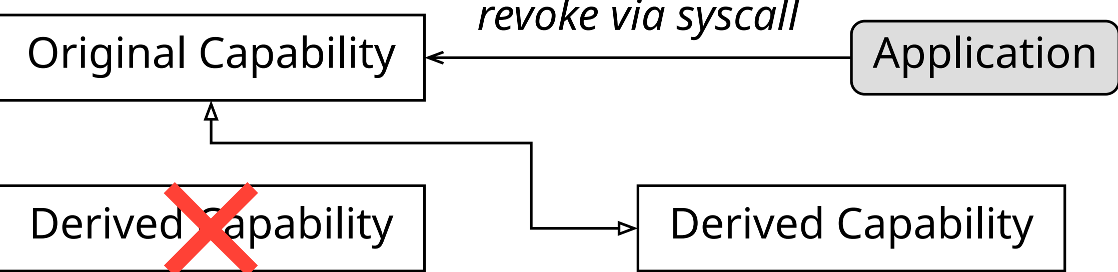
Interrupting Kernel Operations [9], [15]

- Real-time $\Leftarrow$ seL4 needs to handle interrupts promptly
  - Hardware interrupts disabled while in kernel ≠
  - Some system calls (e.g., capability revocation) can take long =
- *Stateless kernel*: incrementally complete system calls
  - On interrupt, abort system call at current increment ≠
  - Prepare thread to re-enter system call and complete remaining increments

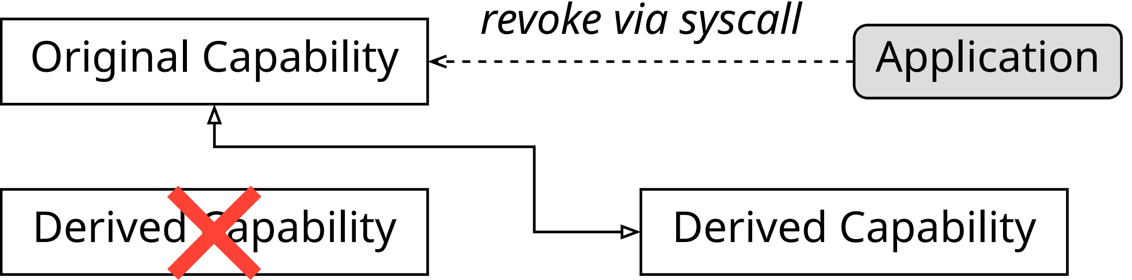
Incremental Consistency [7], [15]



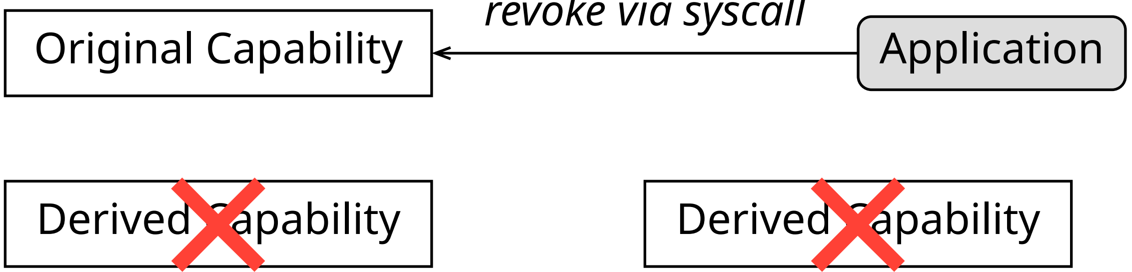
Incremental Consistency [7], [15]



Incremental Consistency [7], [15]



Incremental Consistency [7], [15]



Incremental Consistency [7], [15]

Original Capability

Derived Capability

Derived Capability

- ⤊ Timely interrupt and single kernel stack
- ⤋ Requires careful kernel design with incremental operations

Formal Verification [16]

- First verified general purpose OS (in 2009) ≠
- Functional correctness proof: C implementation correctly refines abstract specification

Formal Verification [16]

- First verified general purpose OS (in 2009) ≠
- Functional correctness proof: C implementation correctly refines abstract specification
- ≈8,300 lines of C code proven against ≈7,500 lines of Isabelle/HOL spec
- Initial proof: ≈200 000 lines of Isabelle script

Formal Verification [16]

- First verified general purpose OS (in 2009) ≠
- Functional correctness proof: C implementation correctly refines abstract specification
- ≈8,300 lines of C code proven against ≈7,500 lines of Isabelle/HOL spec
- Initial proof: ≈200 000 lines of Isabelle script
- ⤊ Proven absence of implementation bugs: crashes, privilege escalation, data corruption
- Proof assumes: hardware behaves correctly
  → does not shield against hardware vulnerabilities like Spectre / Meltdown

Machine-Checked Correctness [17], [18]

Steps of the Verification Process

Machine-Checked Correctness [17], [18]

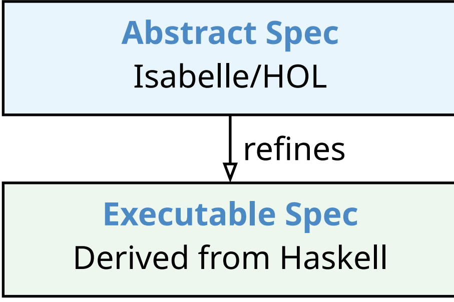
Abstract Spec
Isabelle/HOL

Isabelle
Proof

Steps of the Verification Process

1. *Abstract Spec*: High-level behavior in HOL—higher-order logic

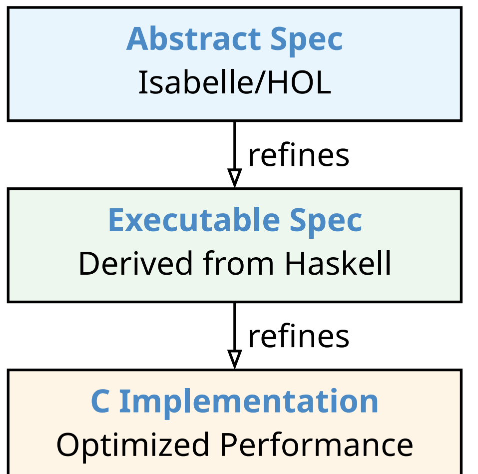
Machine-Checked Correctness [17], [18]



Steps of the Verification Process

1. *Abstract Spec*: High-level behavior in HOL—higher-order logic
2. *Executable Spec*: Functional prototype

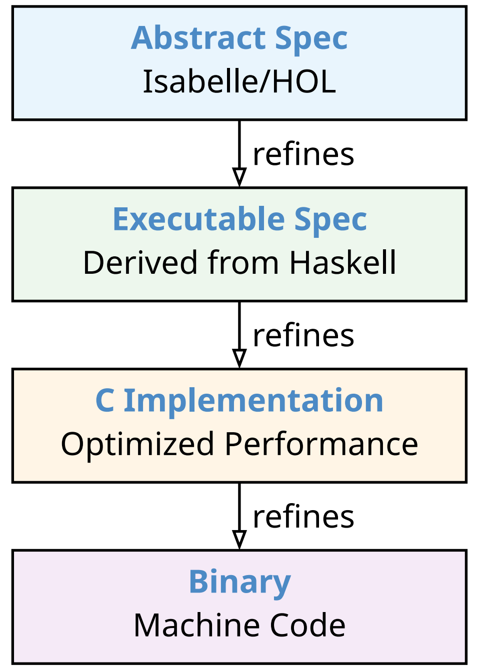
Machine-Checked Correctness [17], [18]



Steps of the Verification Process

1. *Abstract Spec*: High-level behavior in HOL—higher-order logic
2. *Executable Spec*: Functional prototype
3. *C Implementation*: For performance and reduced complexity

Machine-Checked Correctness [17], [18]



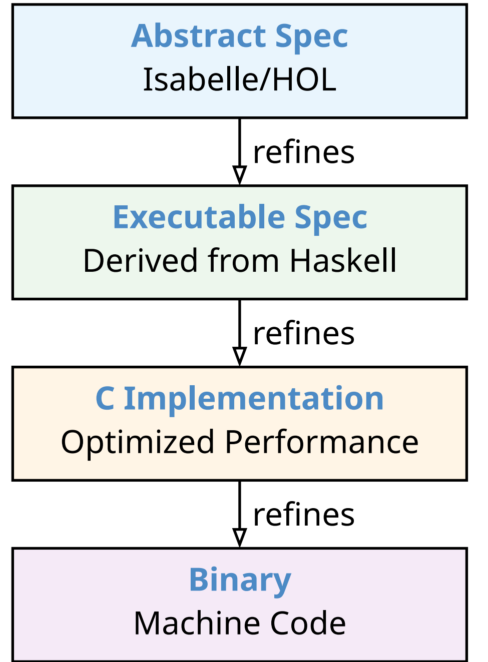
Isabelle
Proof

SMT
(Binary Verif.)

Steps of the Verification Process

1. *Abstract Spec*: High-level behavior in HOL—higher-order logic
2. *Executable Spec*: Functional prototype
3. *C Implementation*: For performance and reduced complexity
4. *Binary Verification*: Machine-code proof using SMT solvers

Machine-Checked Correctness [17], [18]



Isabelle
Proof

SMT
(Binary Verif.)

Steps of the Verification Process

1. *Abstract Spec*: High-level behavior in HOL—higher-order logic
2. *Executable Spec*: Functional prototype
3. *C Implementation*: For performance and reduced complexity
4. *Binary Verification*: Machine-code proof using SMT solvers

- Costly maintenance:
  - Proof updating requires expert knowledge when code changes
  - Proof-to-code ratio at 40:1

LionsOS [19], [20]

- OS of various building blocks on top of seL4 kernel
- Targets static system configurations
- *KISS* principle is paramount
- Specialization of system services to keep them small and simple
- Examples of subsystems working so far: network, storage, I2C, ...
- Other drivers and services can be made available through function donor VMs
- Builds on *Microkit*: Framework for building user-space applications on top of seL4

Bibliography

[1] "Fact Sheet | seL4." Accessed: Apr. 07, 2026. [Online]. Available: https://sel4.systems/About/fact-sheet.html

[2] "History | seL4." Accessed: Apr. 07, 2026. [Online]. Available: https://sel4.systems/About/history.html

[3] "Ongoing seL4 Research | seL4." Accessed: Apr. 07, 2026. [Online]. Available: https://sel4.systems/Research/ongoing.html

[4] *seL4 Logo Pack*. Accessed: Apr. 09, 2026. [Online]. Available: https://sel4.systems/Legal/logo.html

[5] G. Heiser, "Introduction: Microkernels and seL4," *Advanced Operating Systems*. 2025. Accessed: Apr. 01, 2026. [Online]. Available: https://cgi.cse.unsw.edu.au/~cs9242/25/lectures/01a-intro.pdf

[6] "seL4 Reference Manual v15." Apr. 17, 2026. Accessed: Apr. 17, 2026. [Online]. Available: https://sel4.systems/Info/Docs/seL4-manual-15.0.0.pdf

Bibliography

[7] “seL4 Reference Manual v14.” Nov. 25, 2025. Accessed: Jan. 21, 2026. [Online]. Available: https://sel4.systems/Info/Docs/seL4-manual-14.0.0.pdf

[8] G. Heiser, “Introduction: Using seL4,” *Advanced Operating Systems*. 2025. Accessed: Apr. 01, 2026. [Online]. Available: https://cgi.cse.unsw.edu.au/~cs9242/25/lectures/01b-sel4.pdf

[9] G. Heiser, “OS Execution Models: Events, Co-routines, Continuations, Threads,” *Advanced Operating Systems*. 2025. Accessed: Apr. 01, 2026. [Online]. Available: https://cgi.cse.unsw.edu.au/~cs9242/25/lectures/02a-threadsevents.pdf

[10] G. Heiser, “How to (and how not to) use seL4 IPC,” microkerneldude. Accessed: Apr. 29, 2026. [Online]. Available: https://microkerneldude.org/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/

Bibliography

[11] D. Elkaduwe, P. Derrin, and K. Elphinstone, “A Memory Allocation Model for an Embedded Microkernel,” in *International Workshop on Microkernels for Embedded Systems*, NICTA, 2007.

[12] “Multiprocessing on seL4 with verified kernels.” Accessed: Apr. 14, 2026. [Online]. Available: https://sel4.systems/Summit/2022/slides/d1_07_Multiprocessing_on_seL4_with_verified_kernels_Kent_Mcleod.pdf

[13] S. Peters, A. Danis, K. Elphinstone, and G. Heiser, “For a Microkernel, a Big Lock Is Fine,” in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, in APSys '15. Tokyo, Japan: Association for Computing Machinery, 2015. doi: 10.1145/2797022.2797042.

[14] M. von Tessin, “The clustered multikernel: an approach to formal verification of multiprocessor operating-system kernels,” Thesis, 2013. doi: 10.26190/unsworks/16534.

Bibliography

[15] G. Heiser, “The seL4 Microkernel — An Introduction v1.4.” Jan. 08, 2025. Accessed: Apr. 07, 2026. [Online]. Available: https://sel4.org/About/seL4-whitepaper.pdf

[16] G. Klein *et al.*, “seL4: formal verification of an OS kernel,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, in SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. doi: 10.1145/1629575.1629596.

[17] G. Klein *et al.*, “Comprehensive formal verification of an OS microkernel,” *ACM Trans. Comput. Syst.*, vol. 32, no. 1, Feb. 2014, doi: 10.1145/2560537.

[18] T. A. L. Sewell, M. O. Myreen, and G. Klein, “Translation validation for a verified OS kernel,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI '13. Seattle,

Bibliography

Washington, USA: Association for Computing Machinery, 2013, pp. 471–482. doi: 10.1145/2491956.2462183.

[19] G. Heiser *et al.*, “Fast, Secure, Adaptable: LionsOS Design, Implementation and Performance.” [Online]. Available: https://arxiv.org/abs/2501.06234

[20] “The seL4 Microkit | seL4 docs.” Accessed: May 29, 2026. [Online]. Available: https://docs.sel4.systems/projects/microkit/