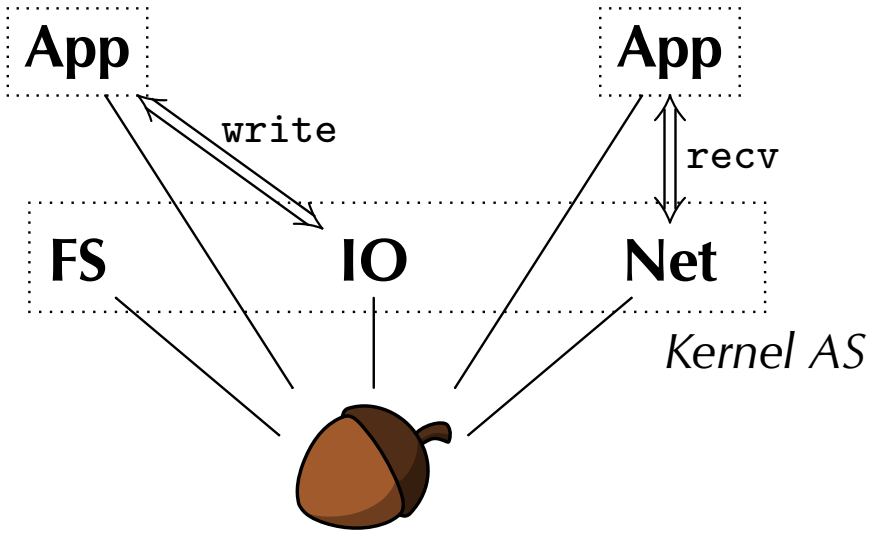
Capabilities in seL4

David Cock

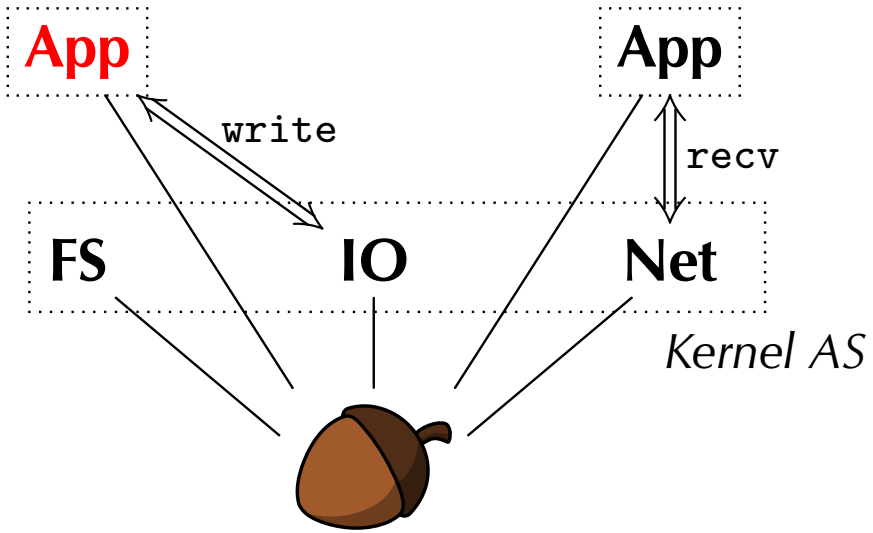
May 13, 2015

Microkernels



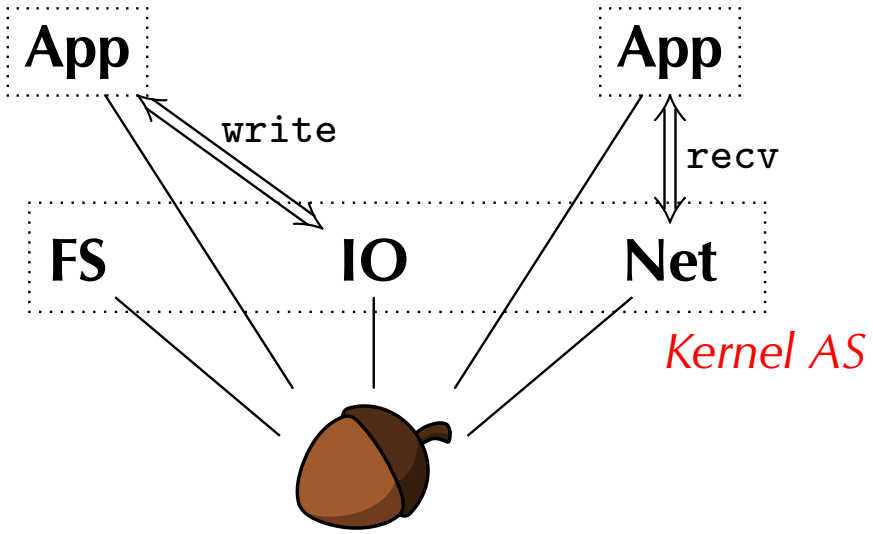
- Partition an OS into servers.
- Small, trusted kernel.
- Core primitives:

Microkernels



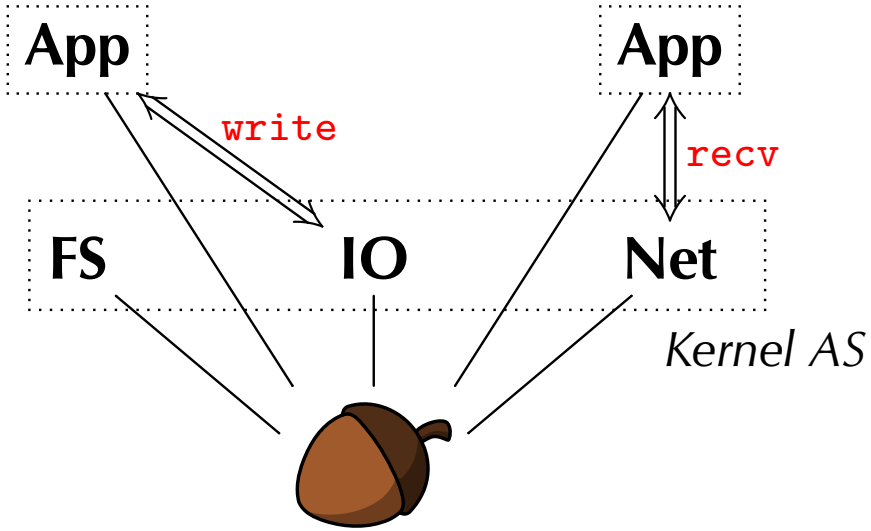
- Partition an OS into servers.
- Small, trusted kernel.
- Core primitives:
  - Threads

Microkernels



- Partition an OS into servers.
- Small, trusted kernel.
- Core primitives:
  - Threads
  - Address spaces

Microkernels



- Partition an OS into servers.
- Small, trusted kernel.
- Core primitives:
  - Threads
  - Address spaces
  - IPC

seL4 & Barrelfish

seL4

- Classical μkernel.
- 1 CPU performance.
- Embedded systems.
- High assurance/verified.

- Multikernel.
- Scalability.
- Large systems.

- The seL4 capability system was adapted to Barrelfish.
- Concurrency means real challenges.

Systems on a Microkernel

An seL4/Barrelfish system is a set of processes, built from:

Kernel Objects

- Execution contexts (Barrelfish) / Threads (seL4).
- Communication endpoints.

Hardware Objects

- Memory regions (frames).
- Address translations (page tables).
- Interrupt routing tables.

Access Control in seL4/Barrelfish

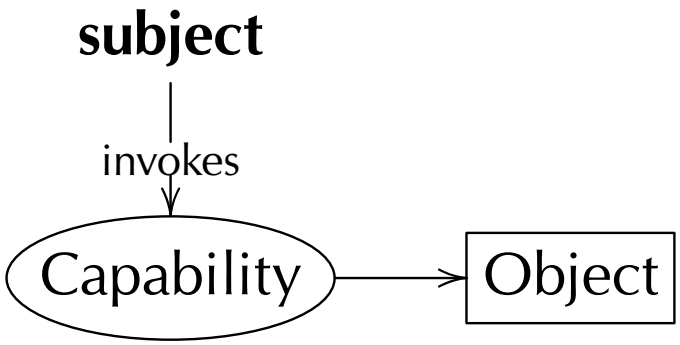
Subjects are user-level processes. *Object* access is kernel (seL4) / CPU driver (BF) -enforced.

Kernel Objects are only accessed during system calls, where the kernel checks permissions.

Hardware Objects are accessed through hardware security mechanisms (e.g. MMU), which are configured by the kernel via system calls.

The kernel and MMU form a *reference monitor*.

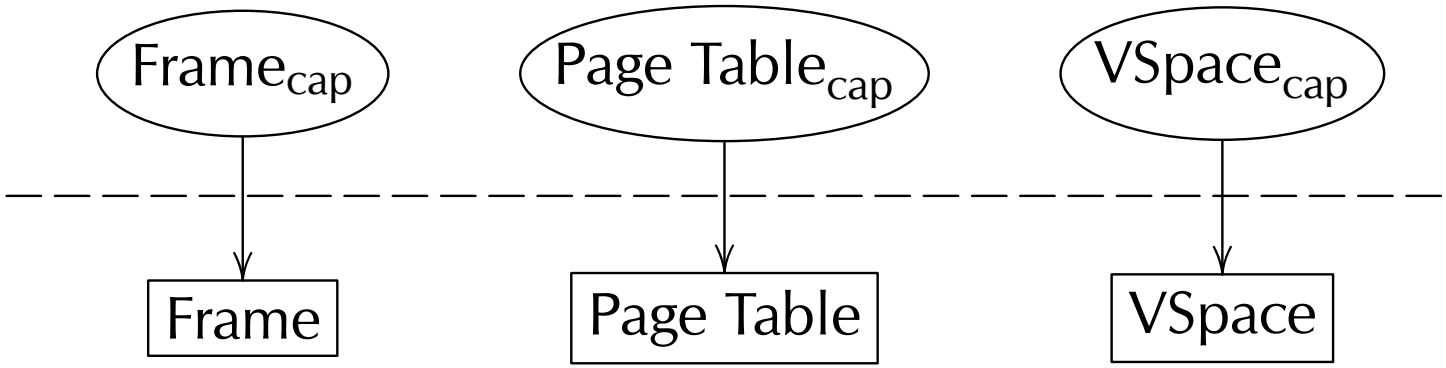
Capabilities



Authority is granted by *capabilities* (caps):

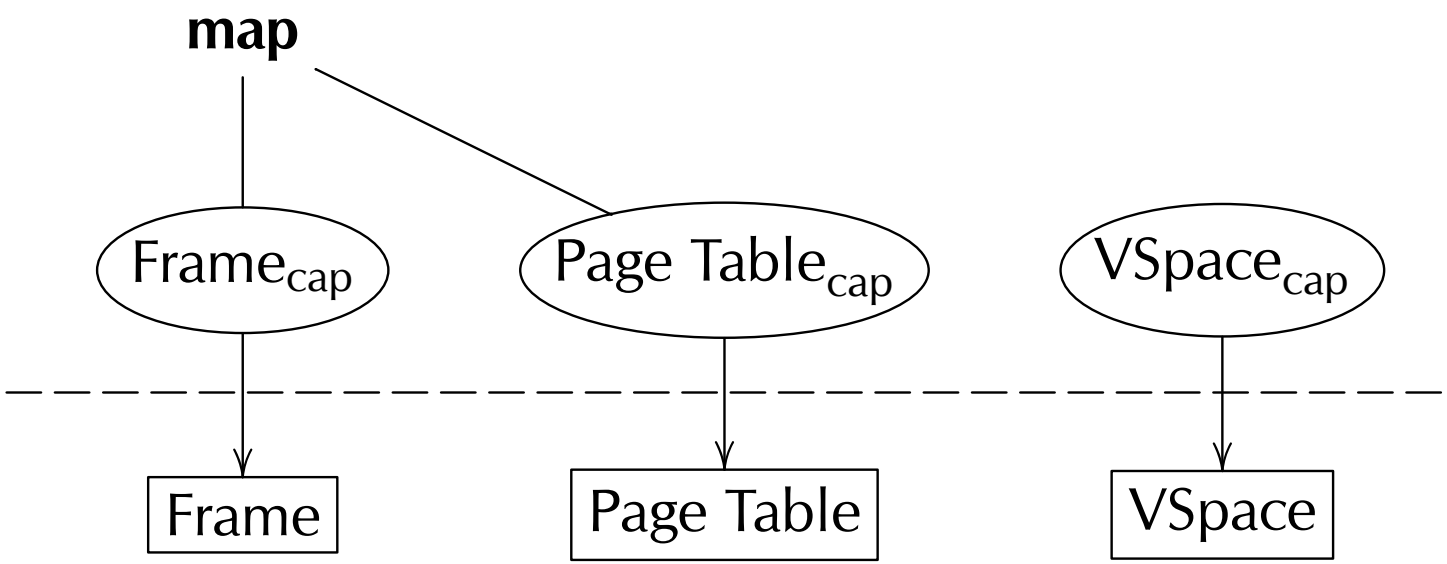
- Unforgeable (kernel/CPU driver checked).
- Transferrable.
- Extensible.

The Capability System



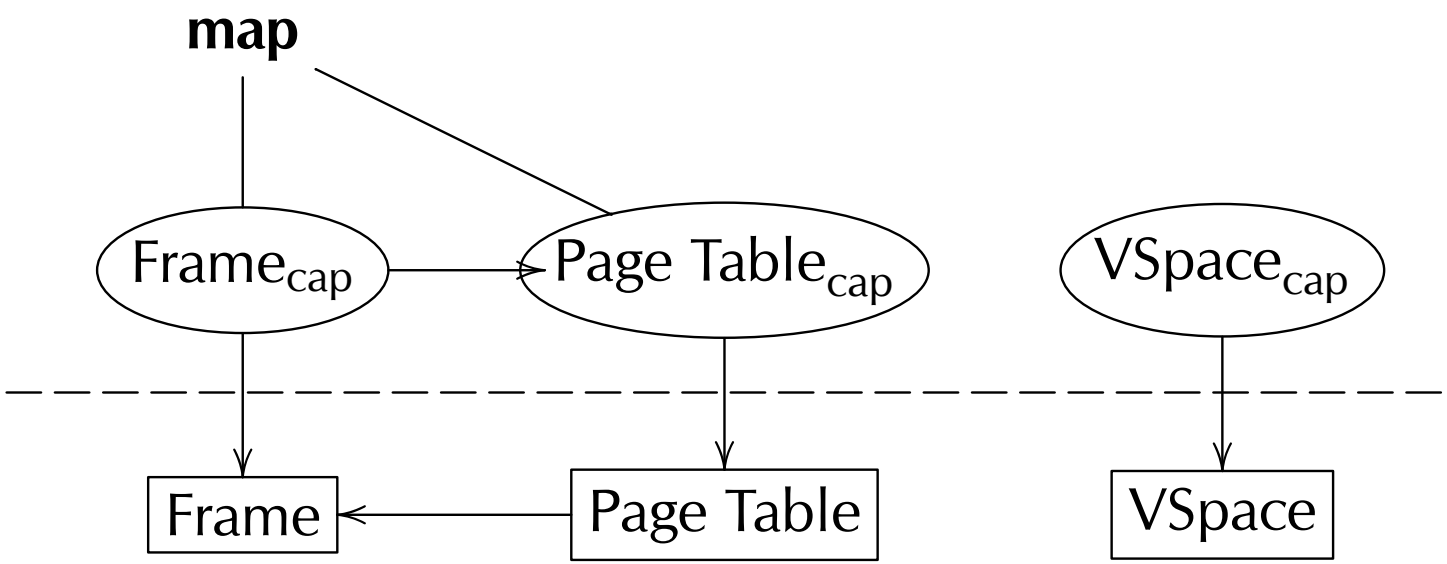
- All objects referred to by caps.
- All system calls are cap invocations.
- Hardware structures mirrored in cap structure.
- Kernel ops are (mostly) *atomic*, also *local* on Barrelfish.

The Capability System



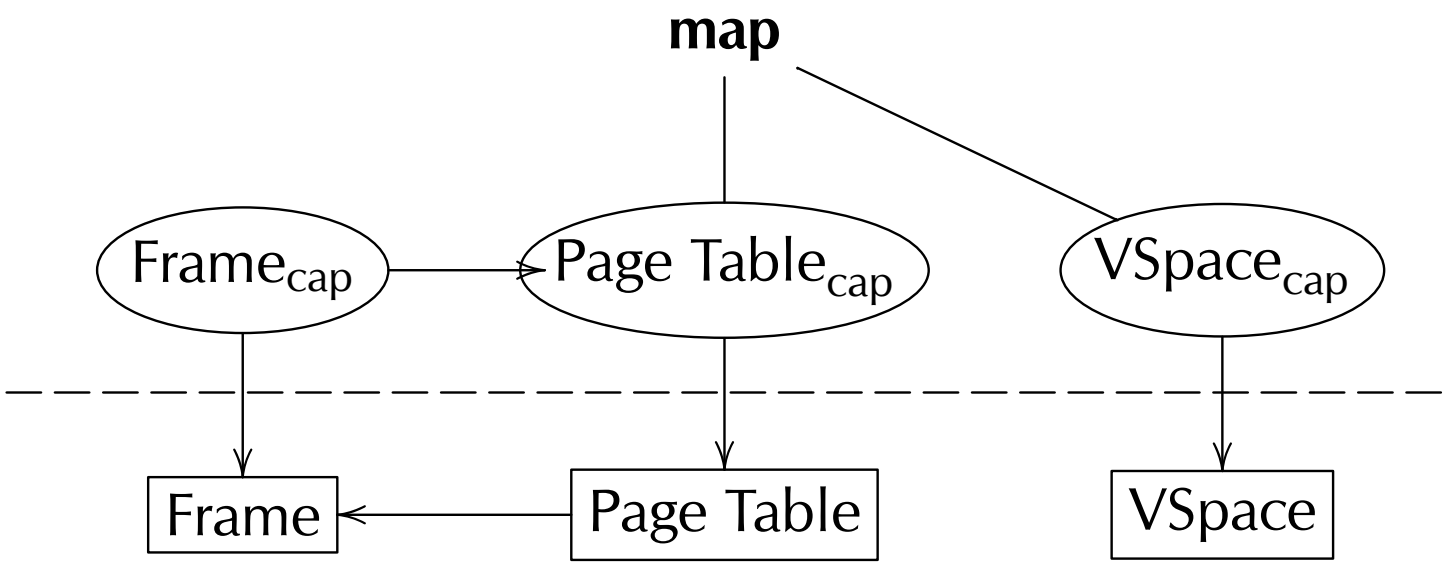
- All objects referred to by caps.
- All system calls are cap invocations.
- Hardware structures mirrored in cap structure.
- Kernel ops are (mostly) *atomic*, also *local* on Barrelfish.

The Capability System



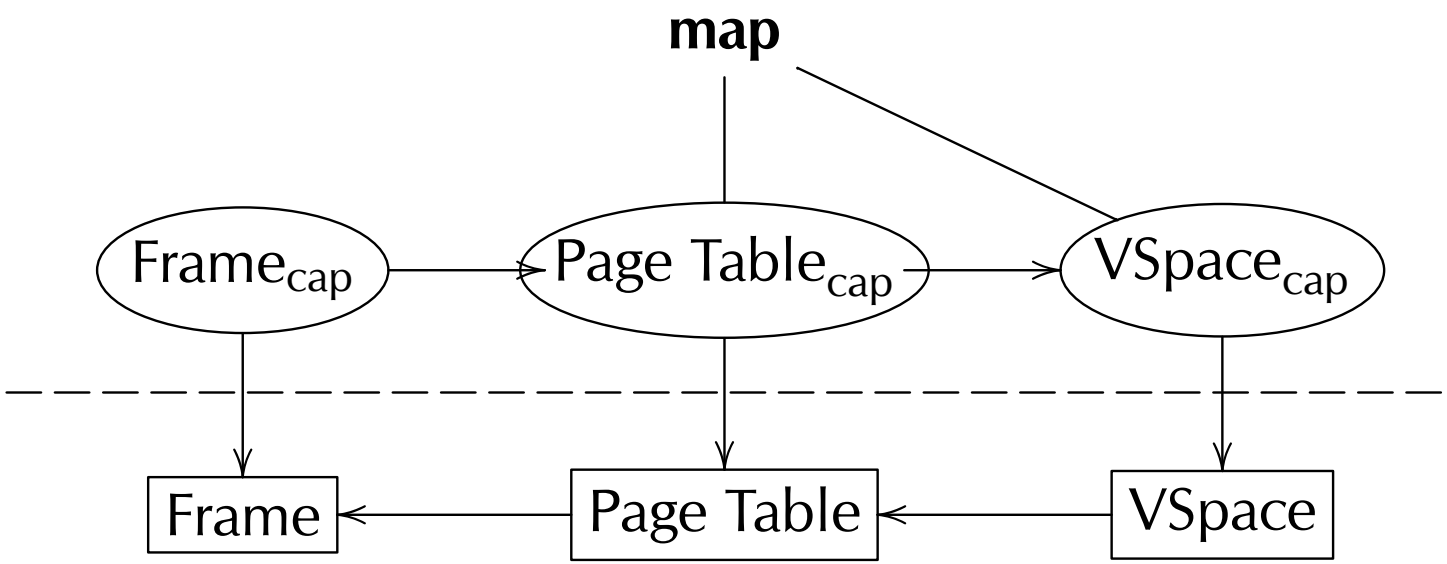
- All objects referred to by caps.
- All system calls are cap invocations.
- Hardware structures mirrored in cap structure.
- Kernel ops are (mostly) *atomic*, also *local* on Barrelfish.

The Capability System



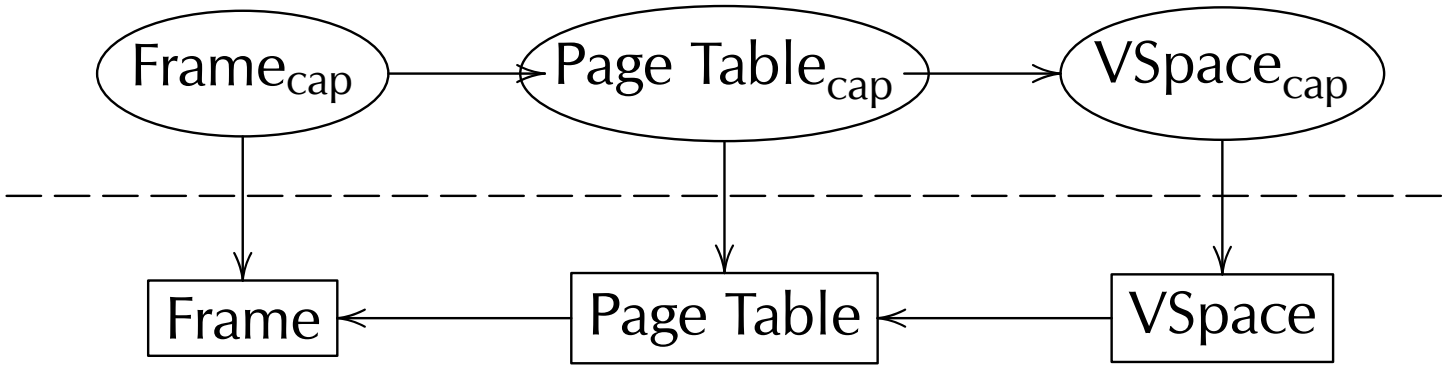
- All objects referred to by caps.
- All system calls are cap invocations.
- Hardware structures mirrored in cap structure.
- Kernel ops are (mostly) *atomic*, also *local* on Barrelfish.

The Capability System



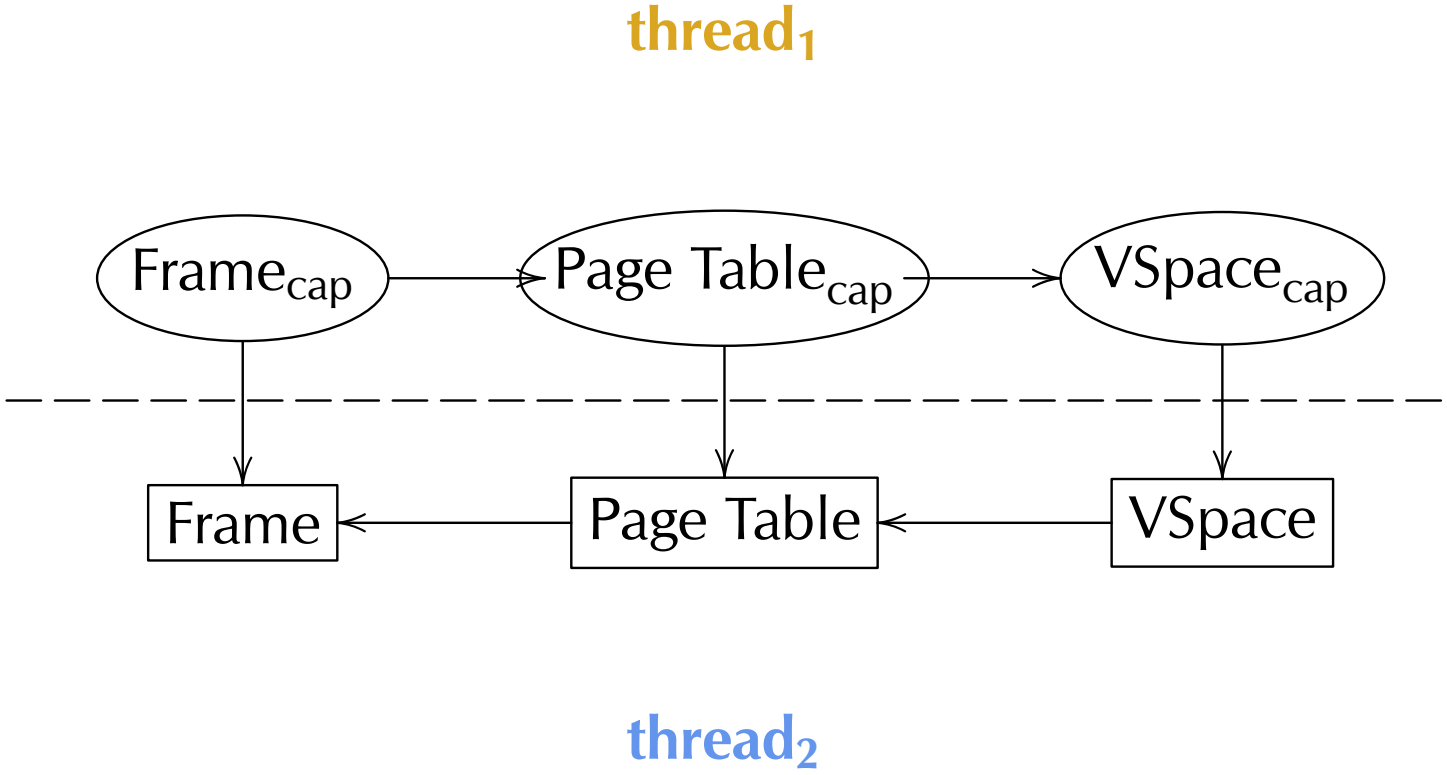
- All objects referred to by caps.
- All system calls are cap invocations.
- Hardware structures mirrored in cap structure.
- Kernel ops are (mostly) *atomic*, also *local* on Barrelfish.

The Capability System



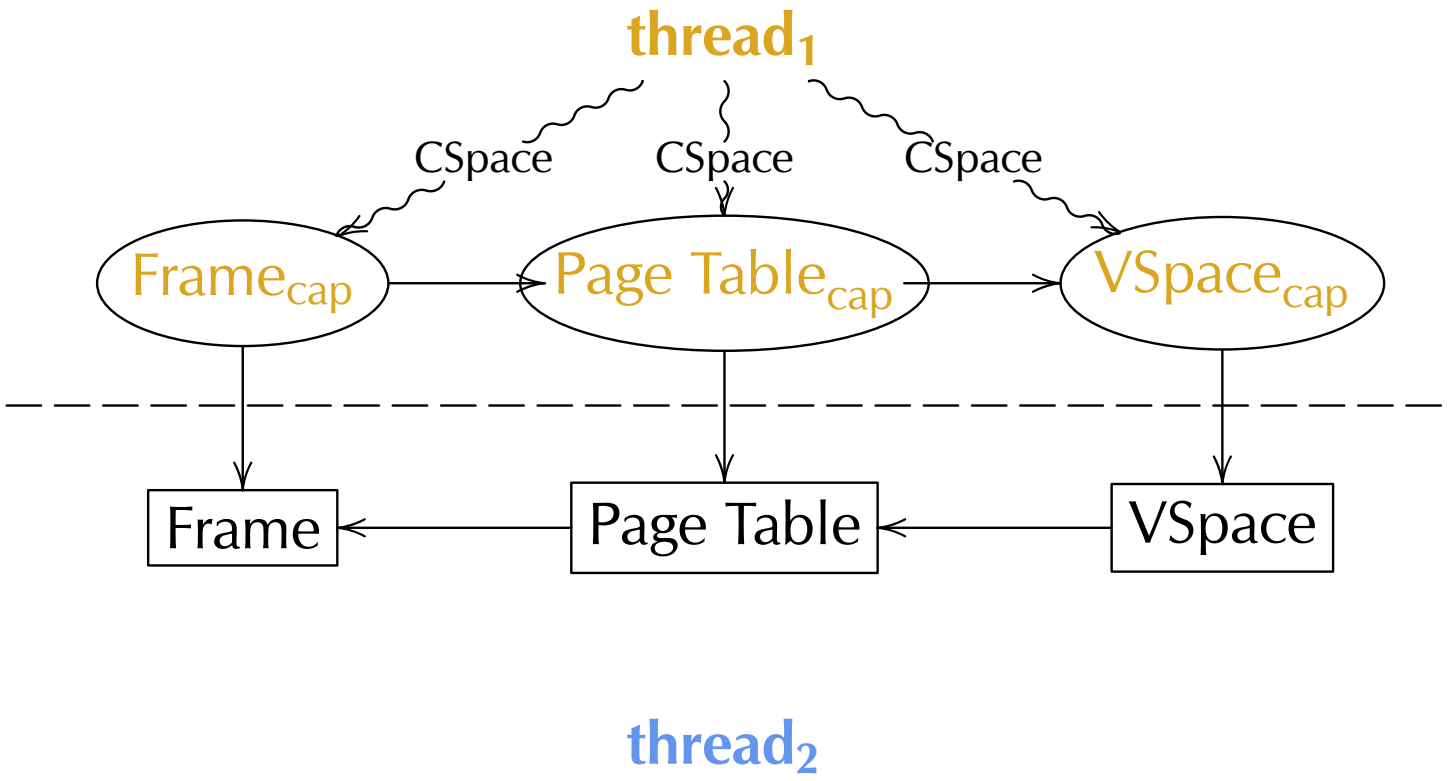
- All objects referred to by caps.
- All system calls are cap invocations.
- Hardware structures mirrored in cap structure.
- Kernel ops are (mostly) *atomic*, also *local* on Barrelfish.

CSpaces and Authority



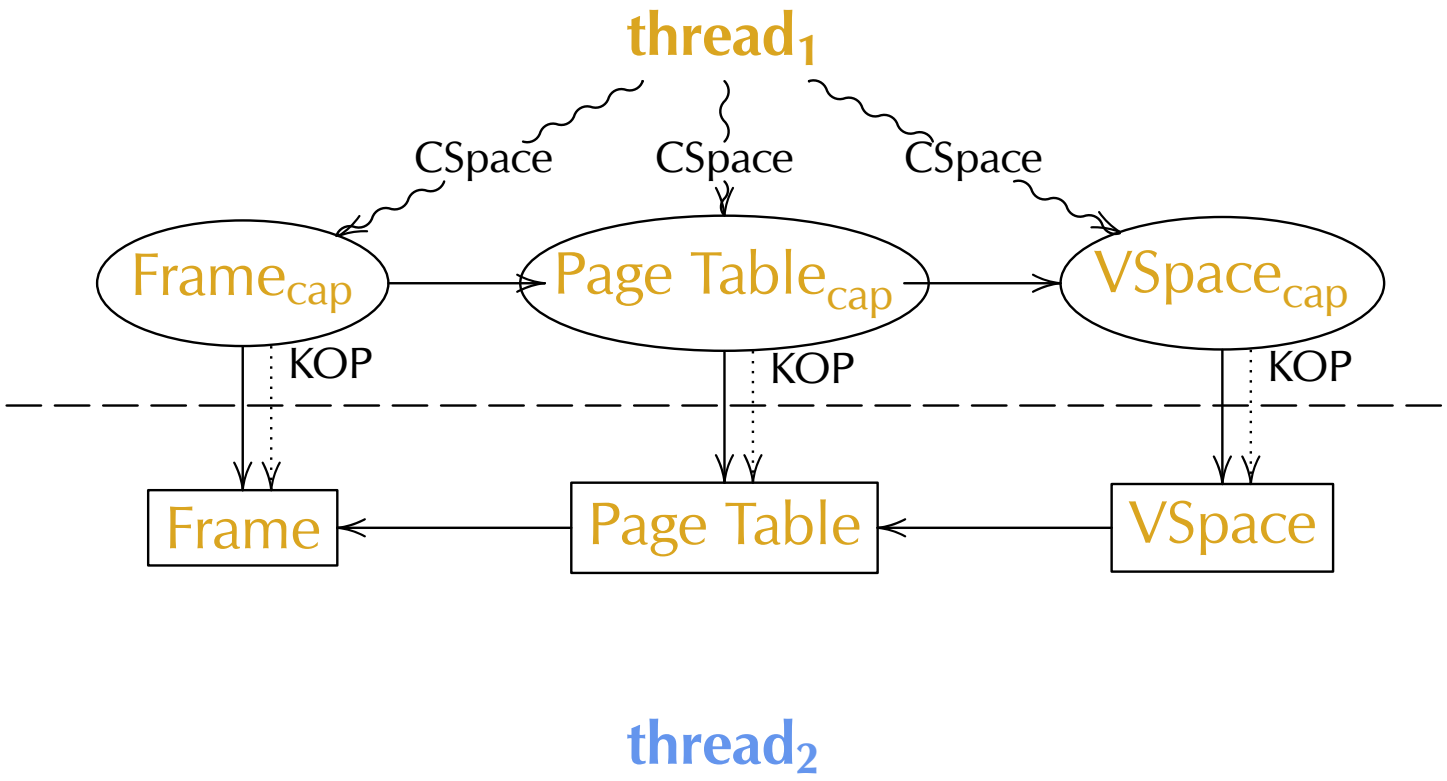
- *CSpaces* hold caps: *explicit authority*.

CSpaces and Authority



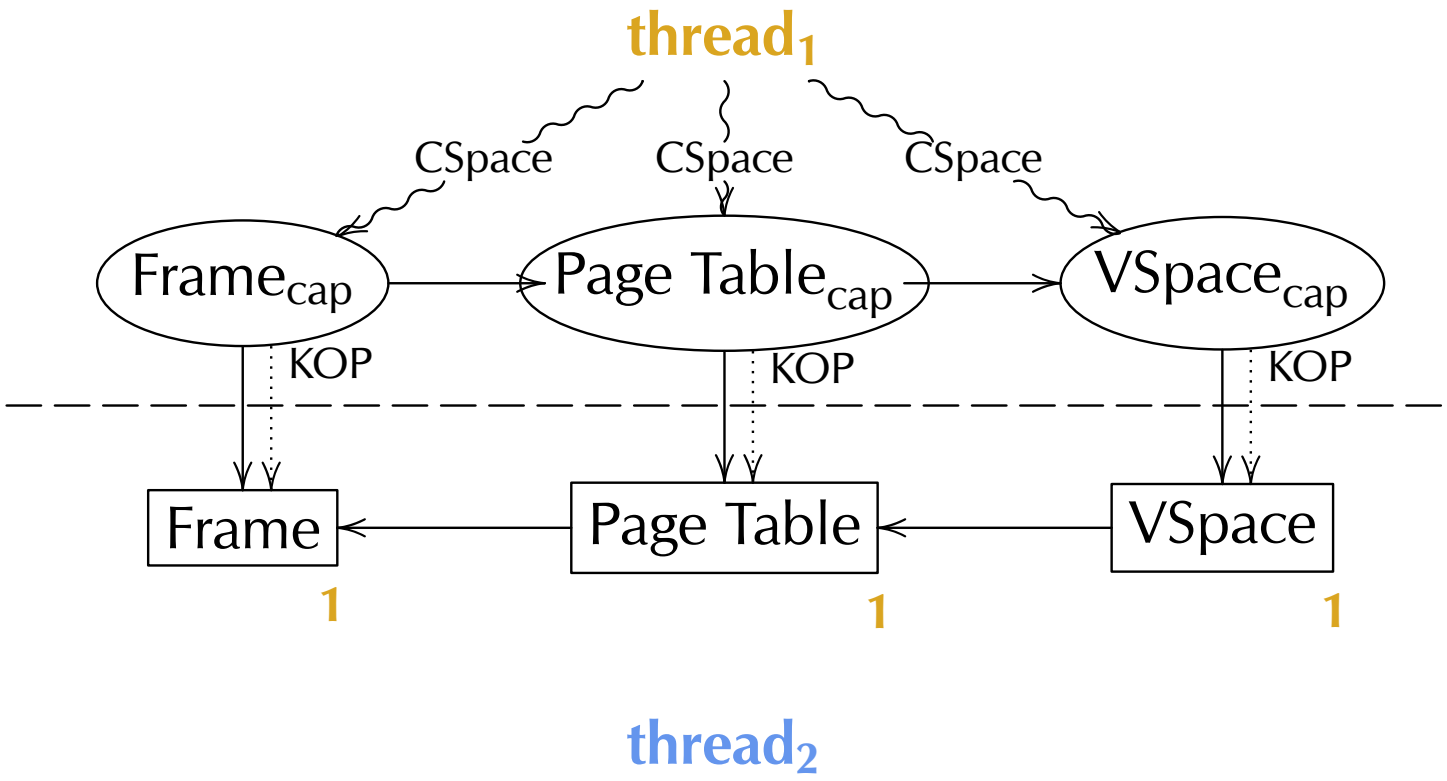
- *CSpaces* hold caps: *explicit authority*.

CSpaces and Authority



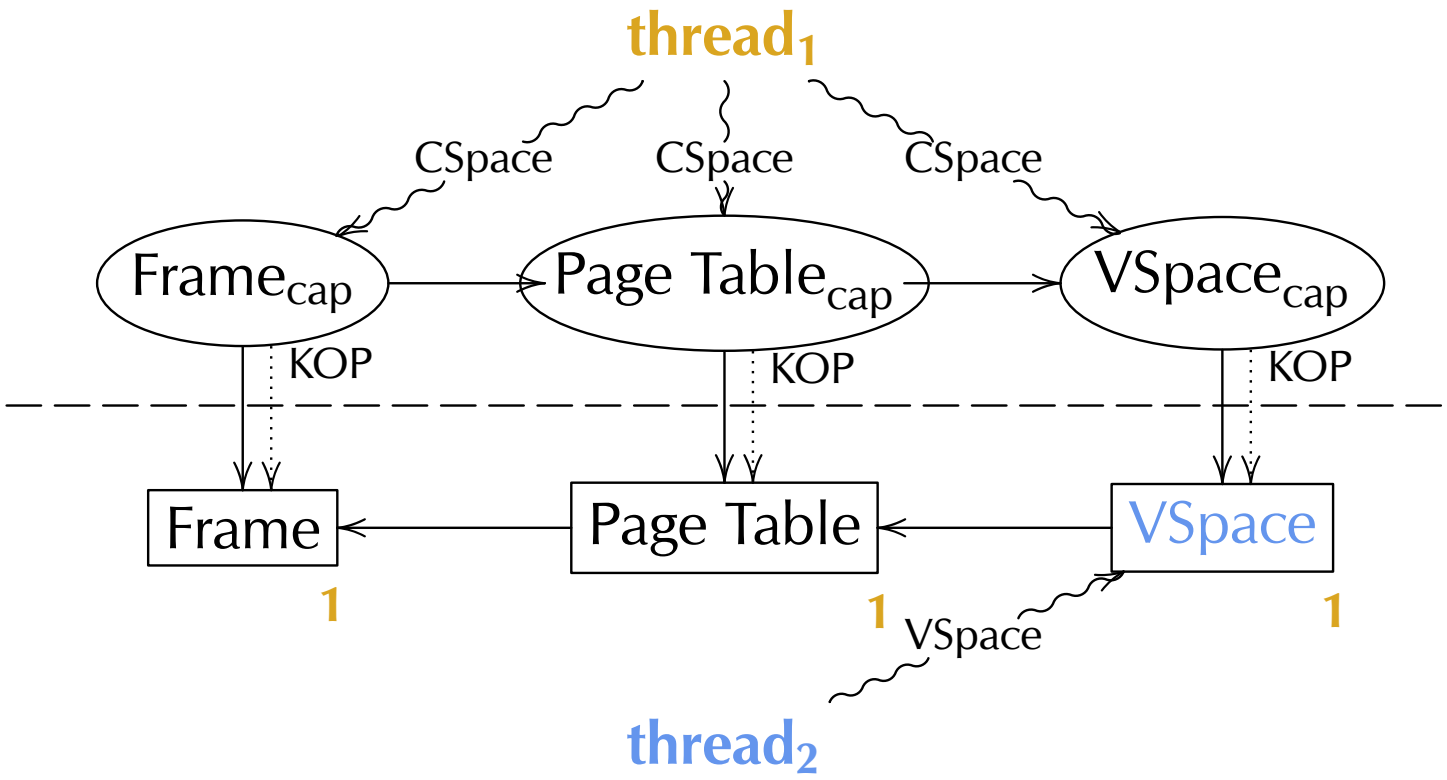
- *CSpaces* hold caps: *explicit authority*.

CSpaces and Authority



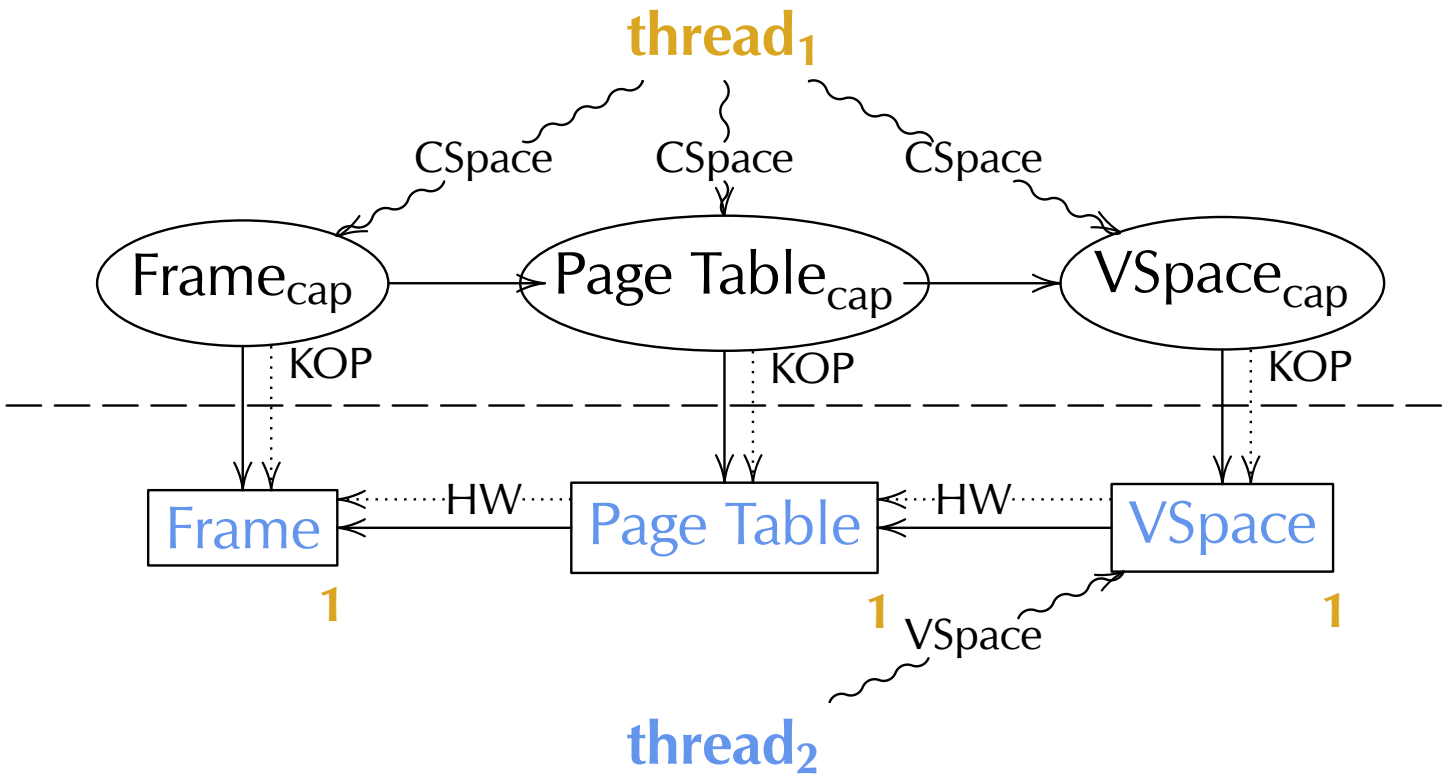
- *CSpaces* hold caps: *explicit authority*.

CSpaces and Authority



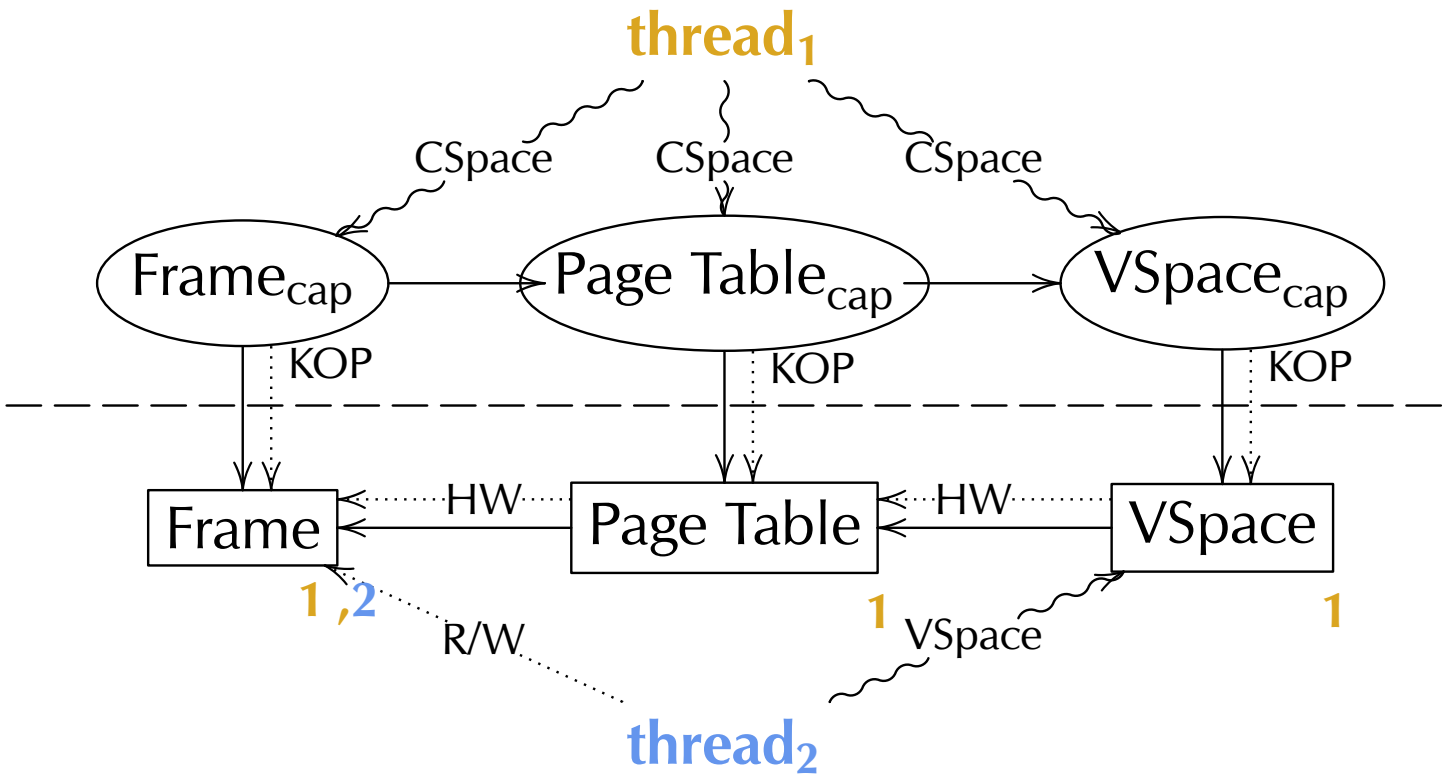
- *CSpaces* hold caps: *explicit authority*.

CSpaces and Authority



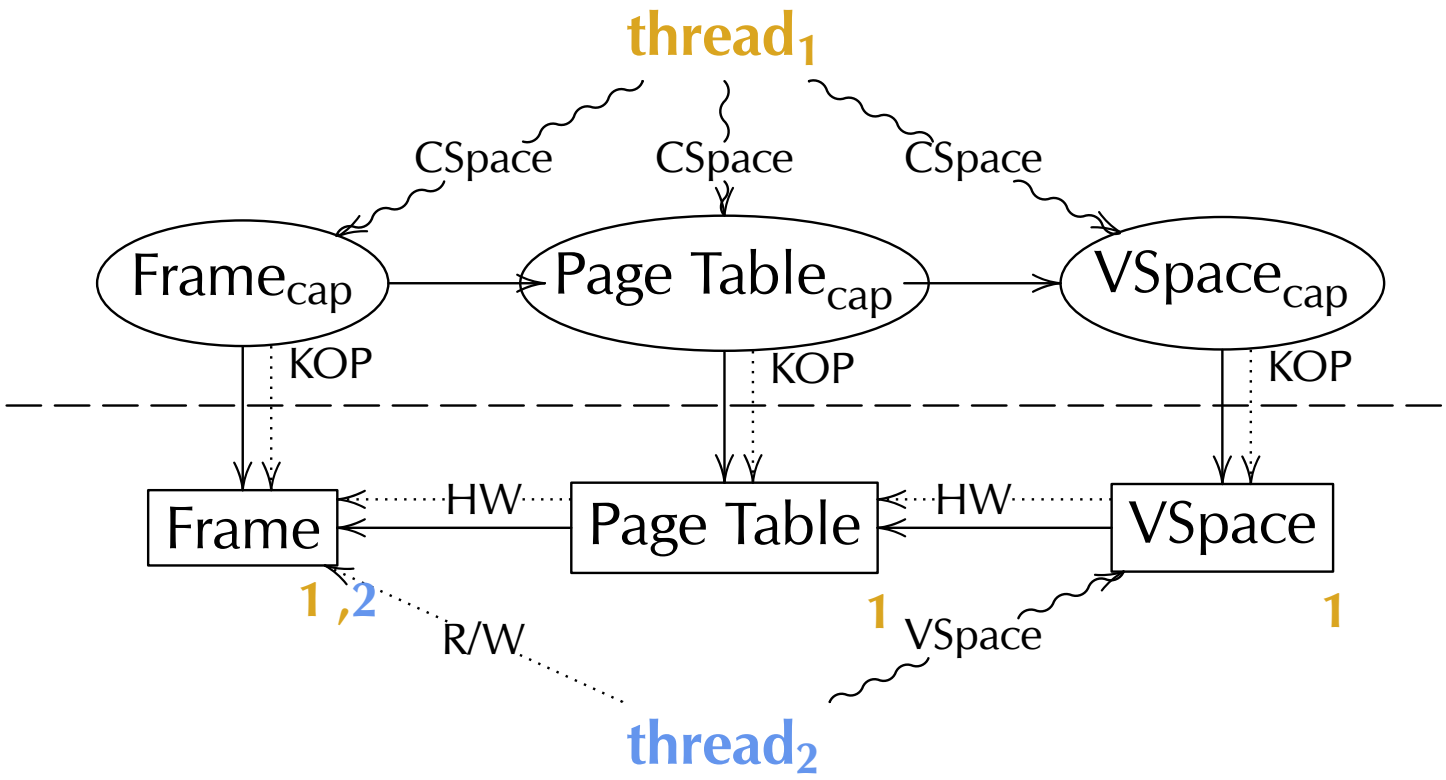
- *CSpaces* hold caps: *explicit authority*.

CSpaces and Authority



- *CSpaces* hold caps: *explicit authority*.
- HW gives *implicit authority* e.g. read/write.

CSpaces and Authority



- *CSpaces* hold caps: *explicit authority*.
- HW gives *implicit authority* e.g. read/write.
- implicit authority $\not\rightarrow$ explicit authority.

Kernel Resource Allocation

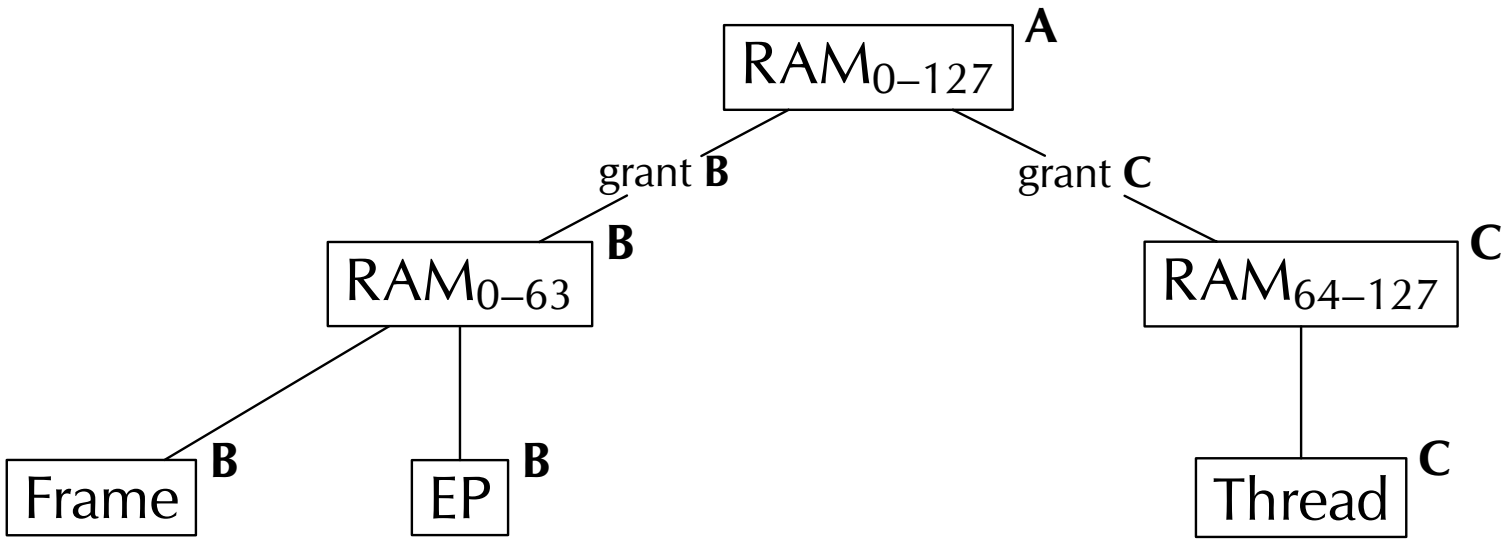
Traditional kernels, including L4, allocate resources for clients: Scheduling queues, IPC queues, ...

Kernel Resource Allocation

Traditional kernels, including L4, allocate resources for clients: Scheduling queues, IPC queues, ...

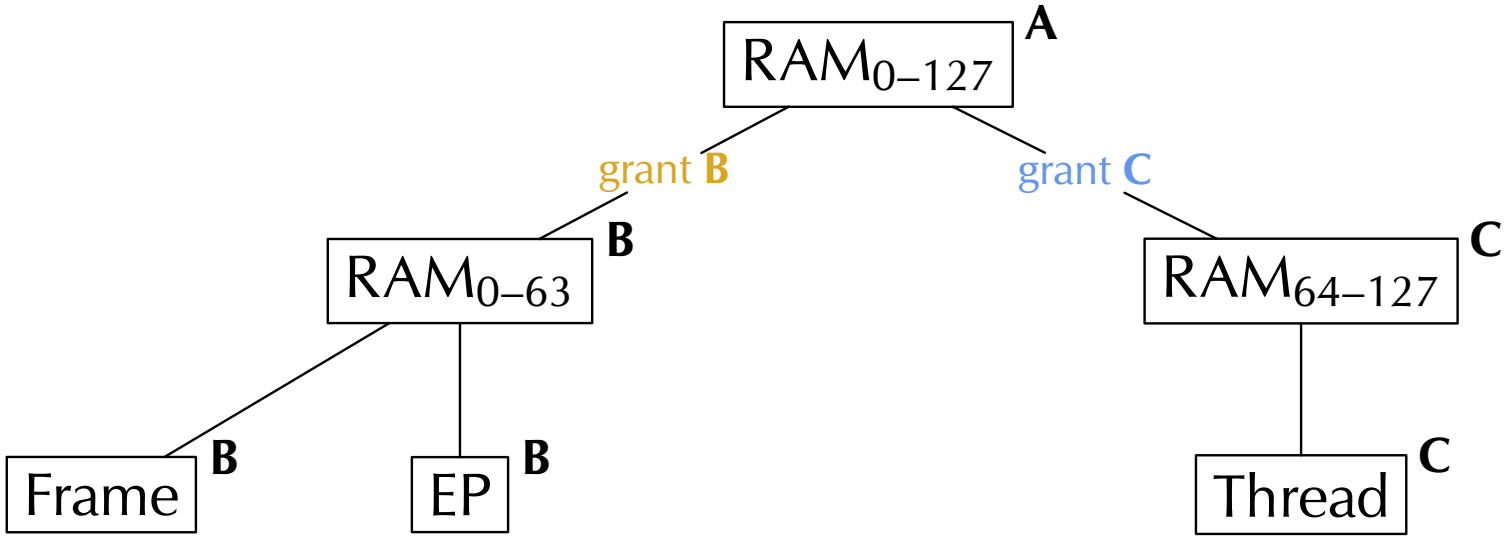
- Threads compete for shared resources.
- Hard to account to threads.
- Allocation policy in the kernel.

Retyping



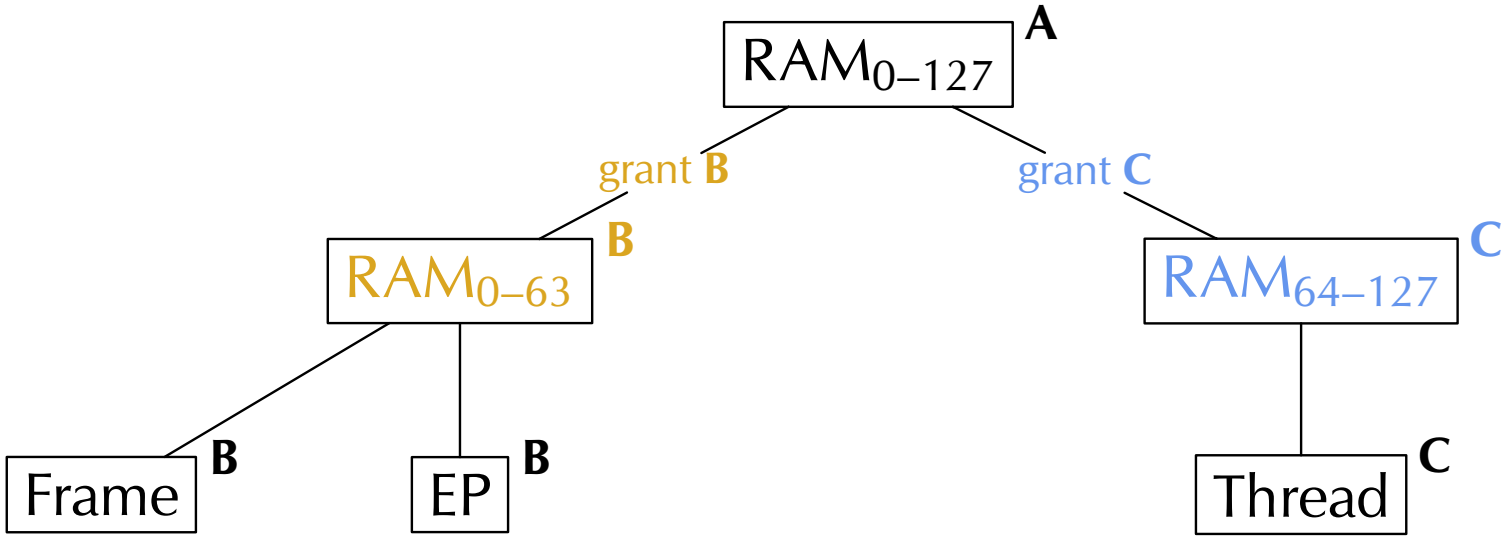
- Resource manager **A** *retypes* (splits) a RAM object.

Retyping



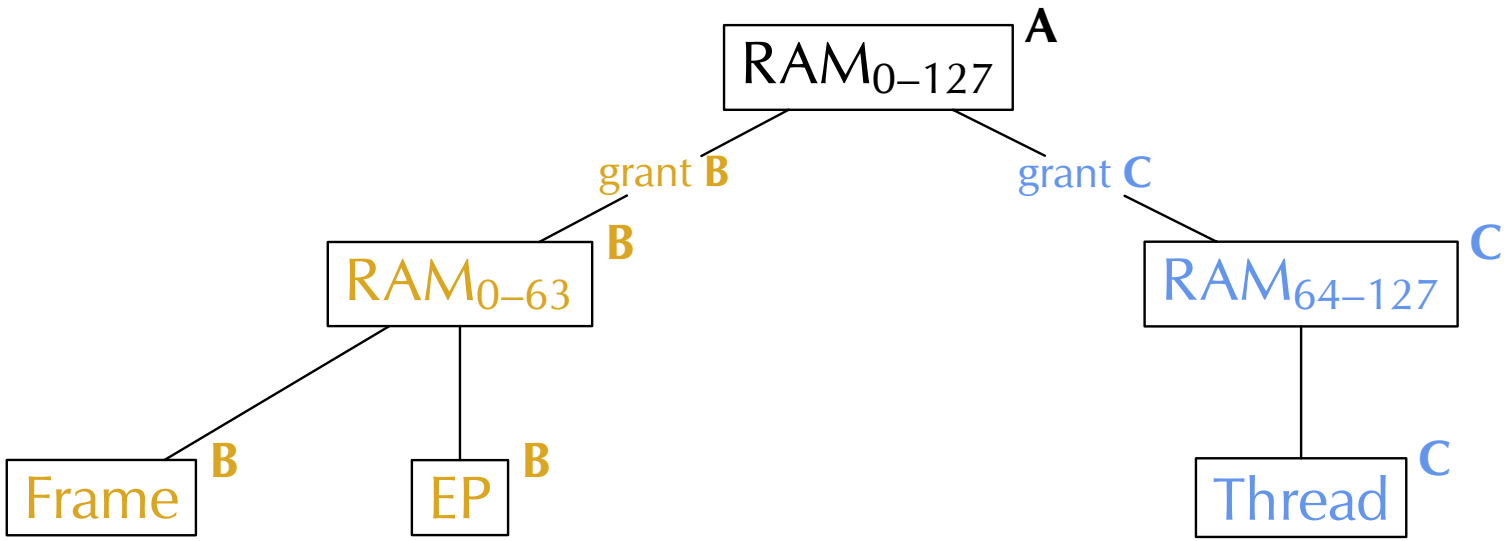
- Resource manager **A** *retypes* (splits) a RAM object.
- **A** *grants* new caps to *mutually untrusting* **B** & **C**.

Retyping



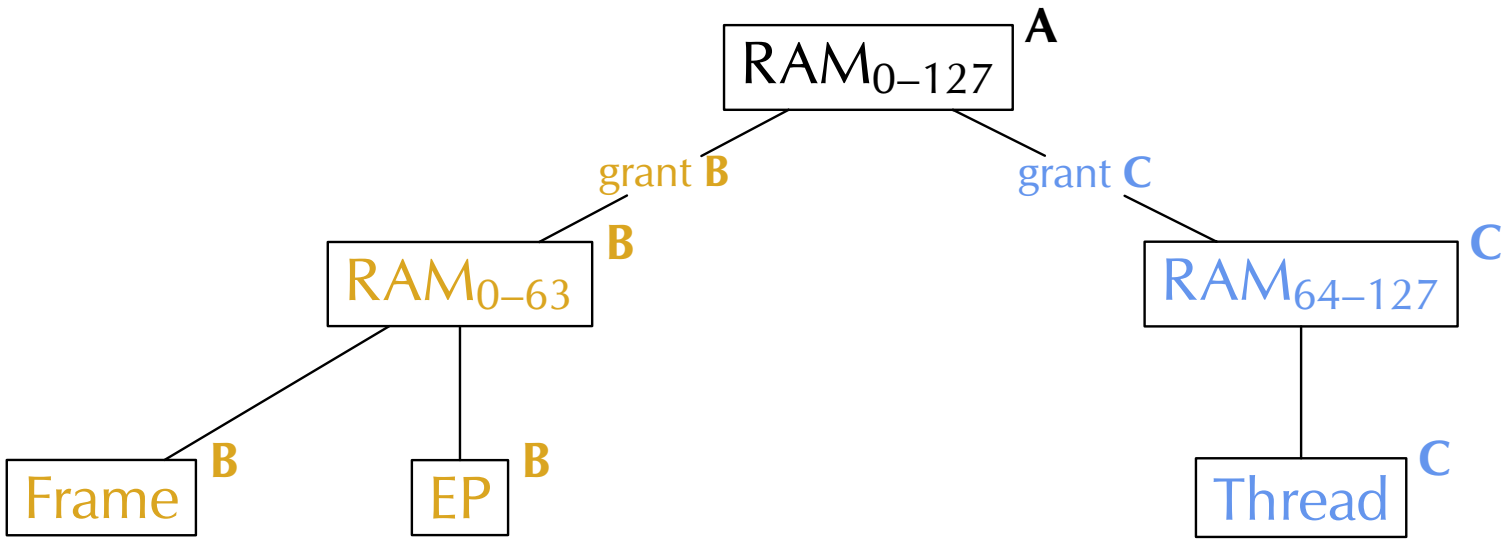
- Resource manager **A** *retypes* (splits) a RAM object.
- **A** *grants* new caps to *mutually untrusting* **B** & **C**.
- **B** & **C** now have partioned resources.

Retyping



- Resource manager **A** *retypes* (splits) a RAM object.
- **A** *grants* new caps to *mutually untrusting* **B** & **C**.
- **B** & **C** now have partioned resources.
- They can perform further retyping themselves.

Retyping



- Resource manager **A** *retypes* (splits) a RAM object.
- **A** *grants* new caps to *mutually untrusting* **B** & **C**.
- **B** & **C** now have partioned resources.
- They can perform further retyping themselves.
- All kernel & user resources are allocated thusly.

The Authority Database Model

The kernel maintains a database of valid capabilities, with requirements:

Atomicity Users (subjects) always see a consistent state.

Performance Cap lookup is on the critical path.

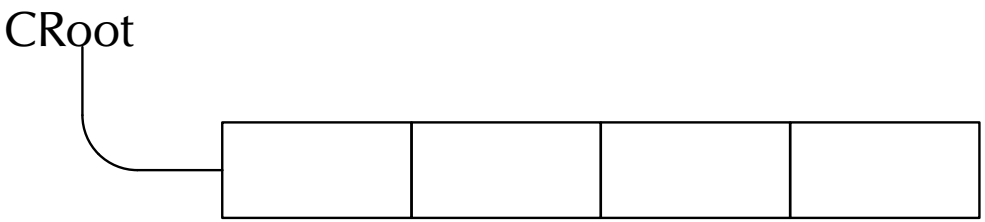
No Allocation Bookkeeping must be stored somewhere.

I will describe the seL4/sequential case. Simon will discuss the Barrelfish/concurrent case.

CNodes

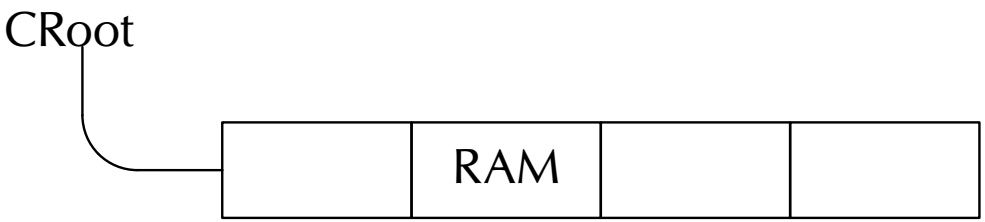
- CNodes objects store caps and bookkeeping.

CNodes



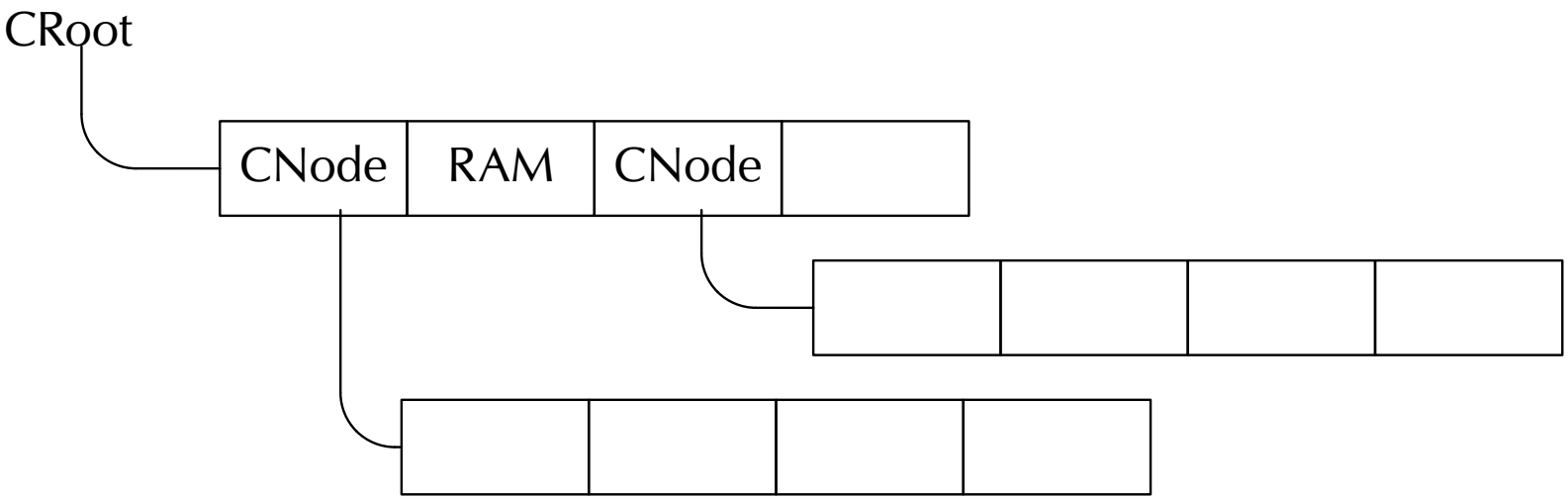
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.

CNodes



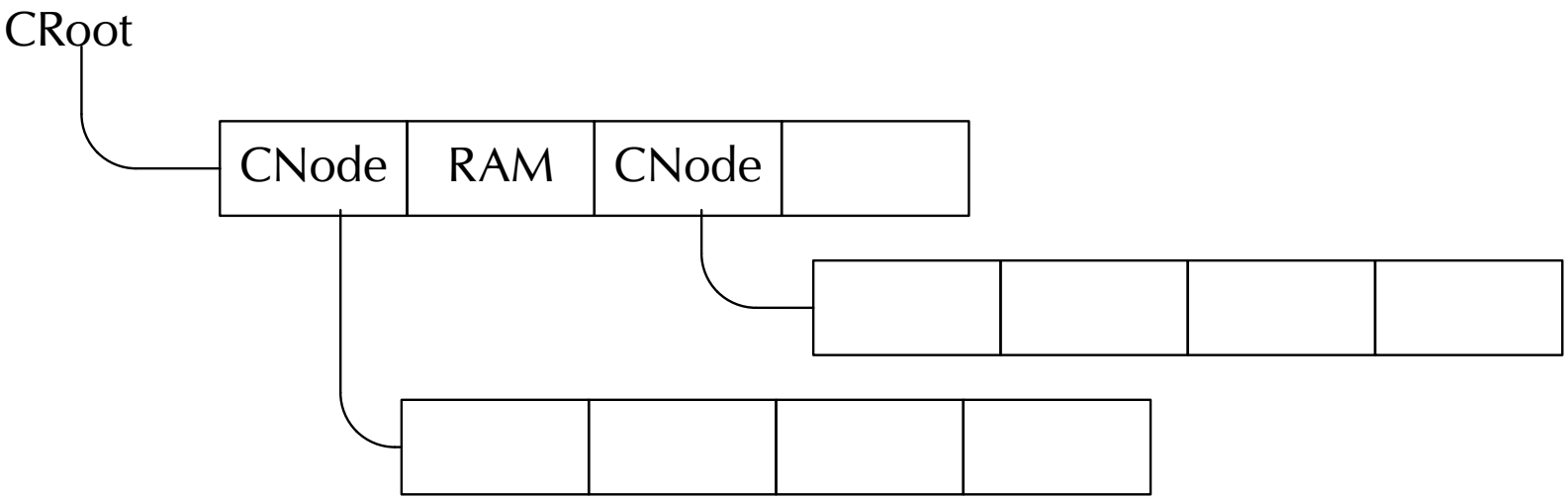
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.

CNodes



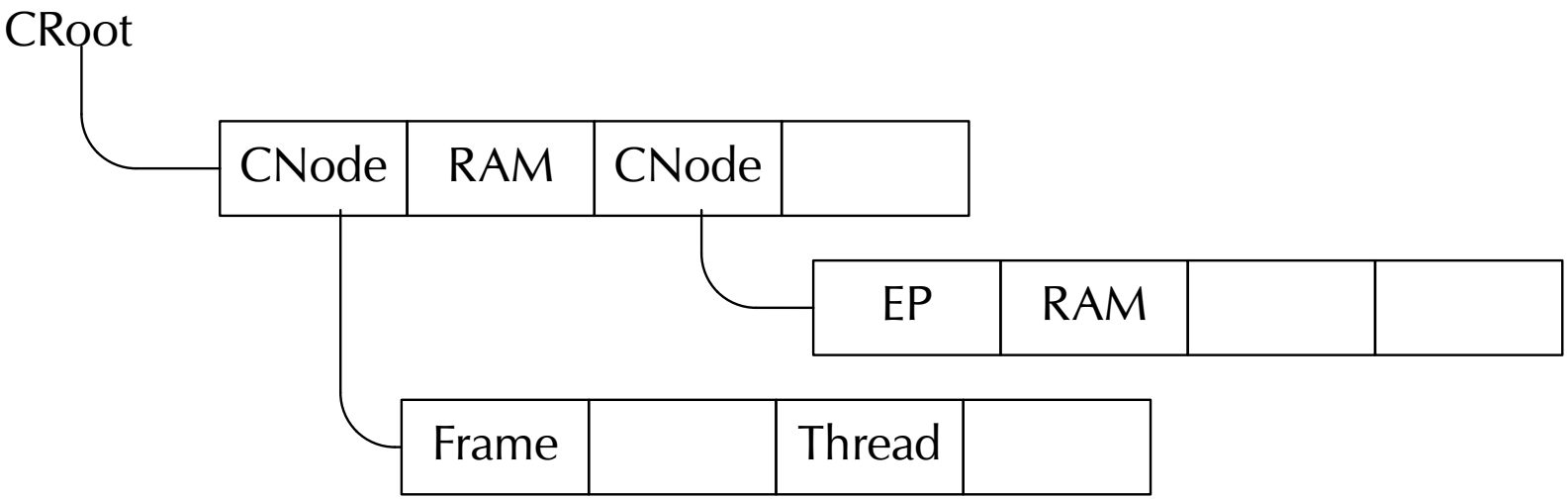
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.

CNodes



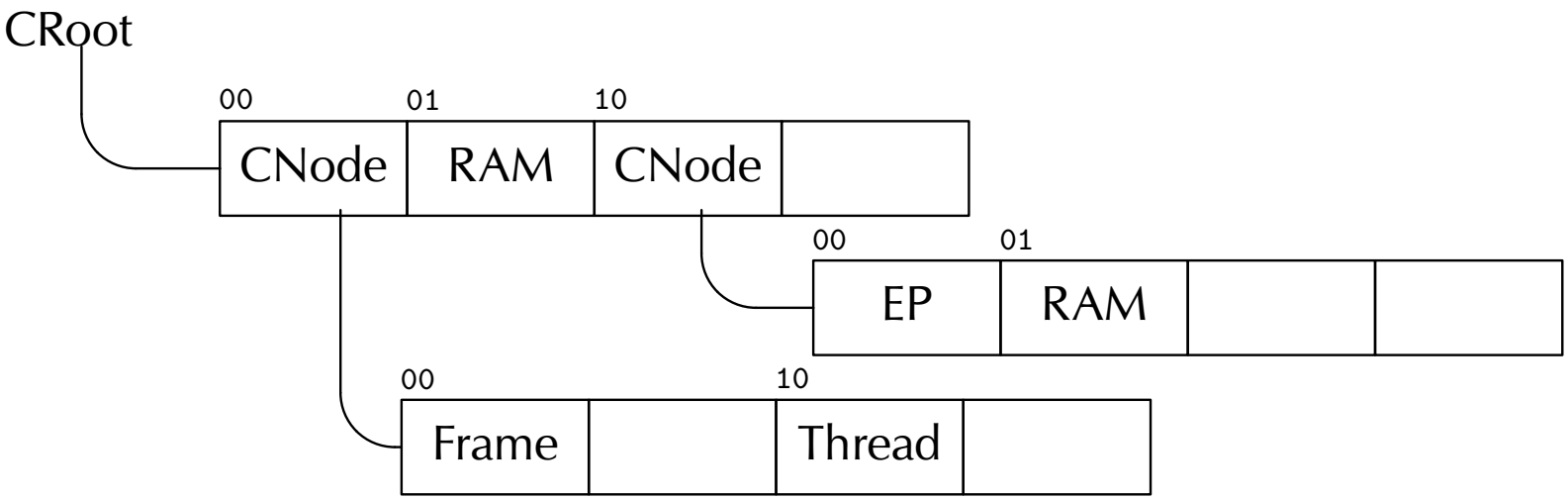
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.

CNodes



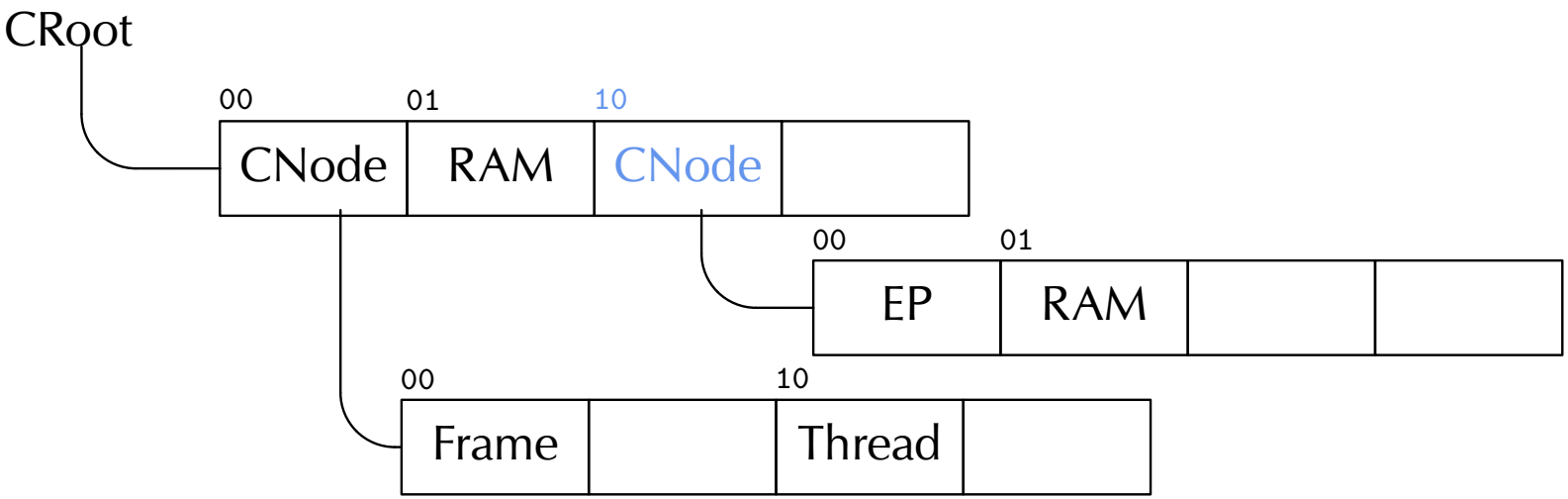
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.

CNodes



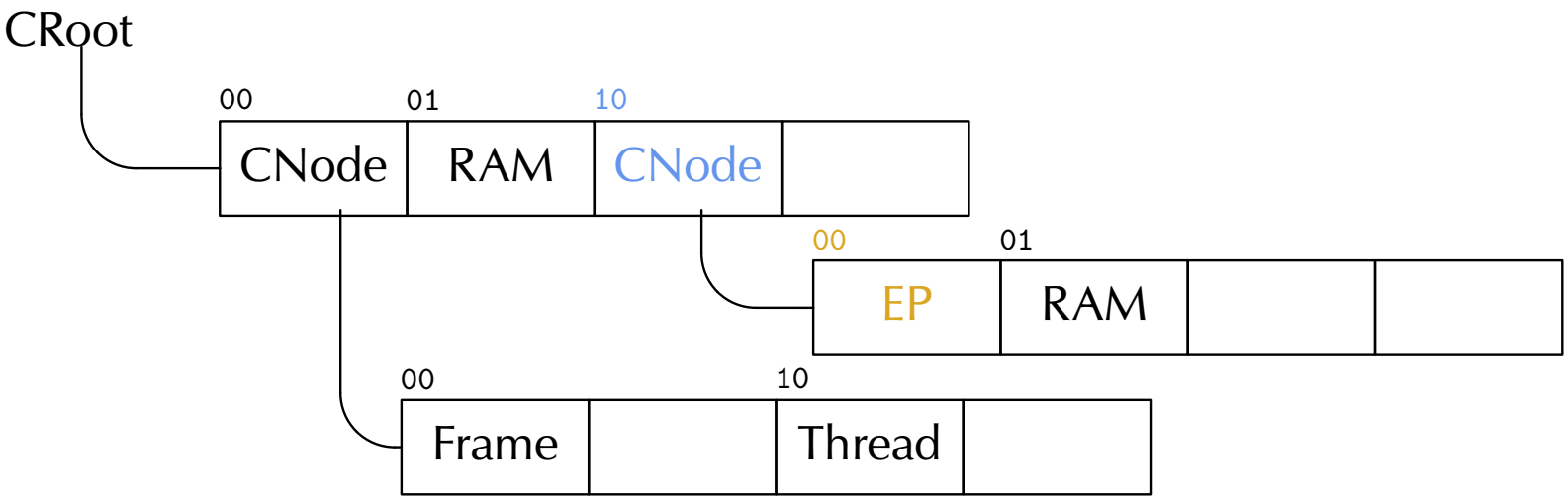
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.
- What's at 1000?

CNodes



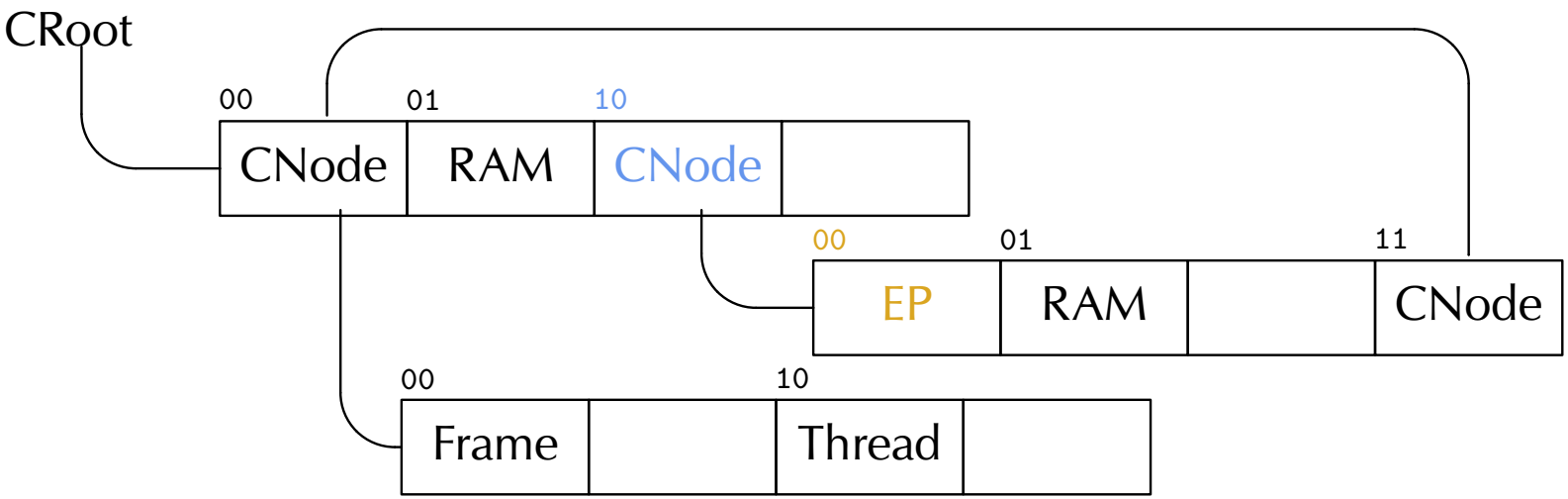
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.
- What's at 1000?

CNodes



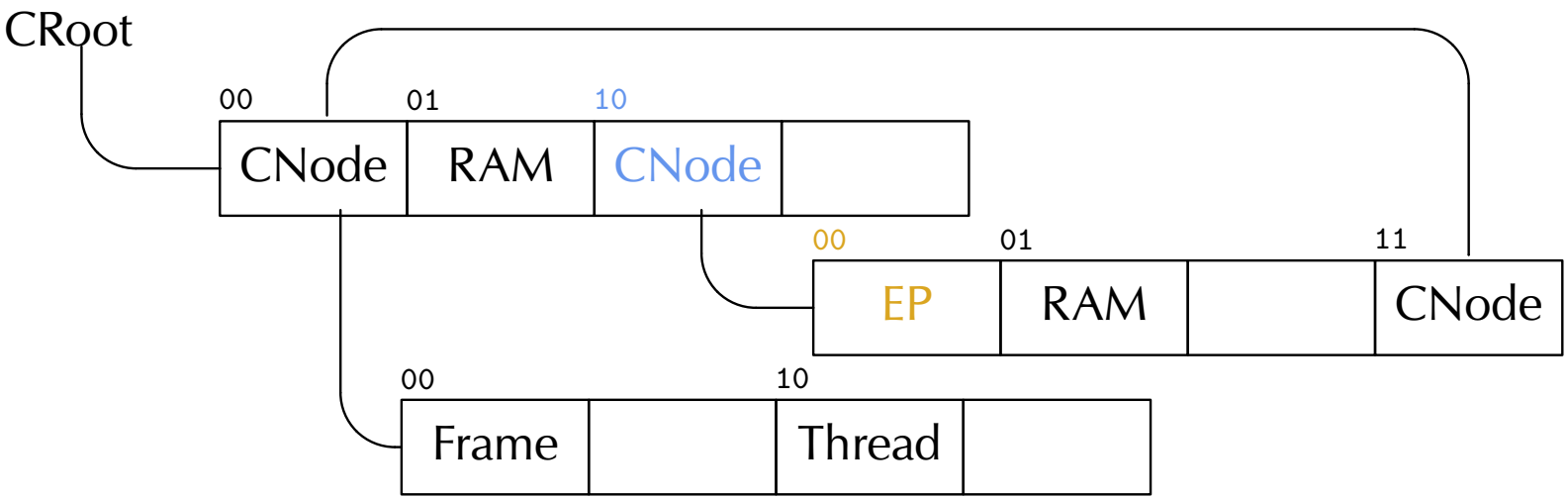
- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.
- What's at 1000? An endpoint.

CNodes



- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.
- What's at 1000? An endpoint.
- CSpaces may have cycles, but finite effective depth.

CNodes



- CNodes objects store caps and bookkeeping.
- A CSpace is all caps reachable from a CRoot.
- CNodes are themselves managed with caps.
- What's at 1000? An endpoint.
- CSpaces may have cycles, but finite effective depth.
- Every invocation is an authority DB *query*.

Cap Operations

These *mutate* the authority DB:

Cap Operations

These *mutate* the authority DB:

Mint/Retype Derive new sub-objects, and caps to them.

Cap Operations

These *mutate* the authority DB:

Mint/Retype Derive new sub-objects, and caps to them.

Copy Create a new cap to an object. The old and new caps are (mostly) indistinguishable.

Cap Operations

These *mutate* the authority DB:

Mint/Retype Derive new sub-objects, and caps to them.

Copy Create a new cap to an object. The old and new caps are (mostly) indistinguishable.

Move Move caps within or between CNodes.

Cap Operations

These *mutate* the authority DB:

Mint/Retype Derive new sub-objects, and caps to them.

Copy Create a new cap to an object. The old and new caps are (mostly) indistinguishable.

Move Move caps within or between CNodes.

Delete Remove the cap. Destroy the object once the last cap is gone.

Cap Operations

These *mutate* the authority DB:

Mint/Retype Derive new sub-objects, and caps to them.

Copy Create a new cap to an object. The old and new caps are (mostly) indistinguishable.

Move Move caps within or between CNodes.

Delete Remove the cap. Destroy the object once the last cap is gone.

Revoke Destroy all objects derived (via retype) from this one.

Cap Operations

These *mutate* the authority DB:

Mint/Retype Derive new sub-objects, and caps to them.

Copy Create a new cap to an object. The old and new caps are (mostly) indistinguishable.

Move Move caps within or between CNodes.

Delete Remove the cap. Destroy the object once the last cap is gone.

Revoke Destroy all objects derived (via retype) from this one.

Delete and Revoke call each other, and are long-running.

Cap Operations

These *mutate* the authority DB:

Mint/Retype Derive new sub-objects, and caps to them.

Copy Create a new cap to an object. The old and new caps are (mostly) indistinguishable.

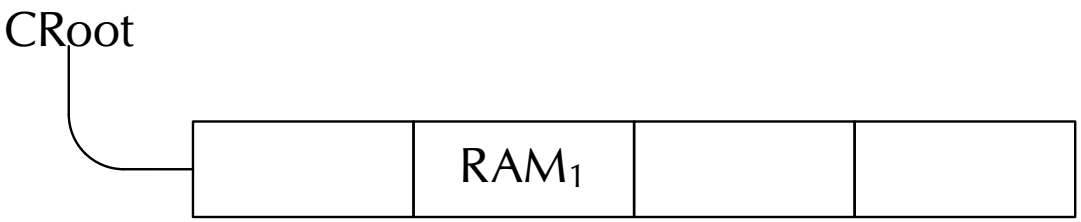
Move Move caps within or between CNodes.

Delete Remove the cap. Destroy the object once the last cap is gone.

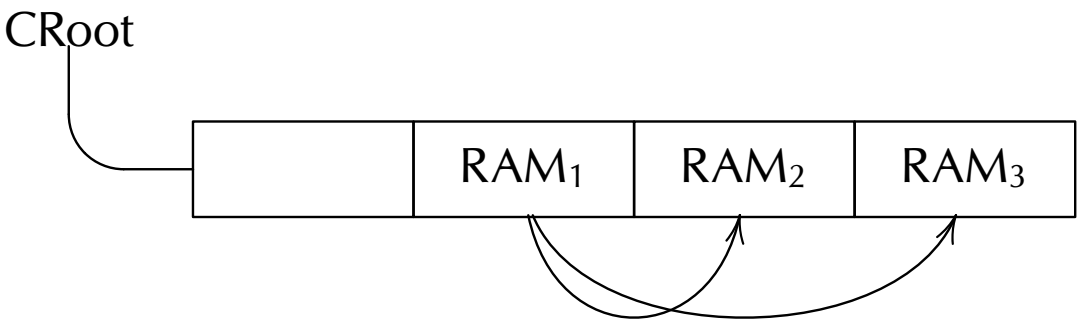
Revoke Destroy all objects derived (via retype) from this one.

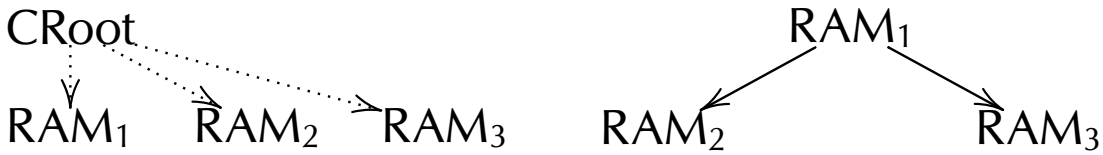
Delete and Revoke call each other, and are long-running. The recursion is not atomic — *Preemptible* on seL4, done in a *user-level monitor* on Barrelfish.

Retype



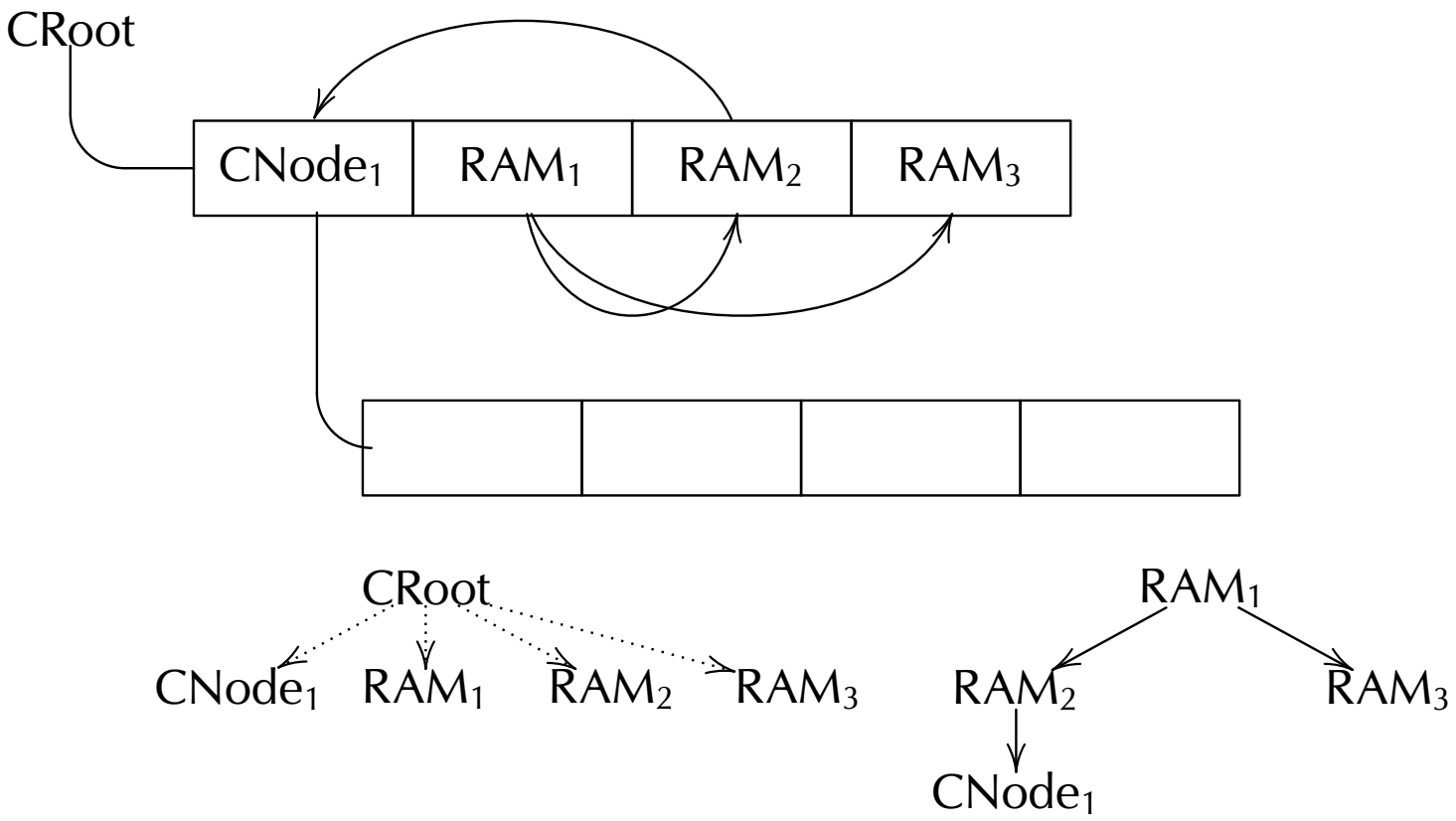
Retype





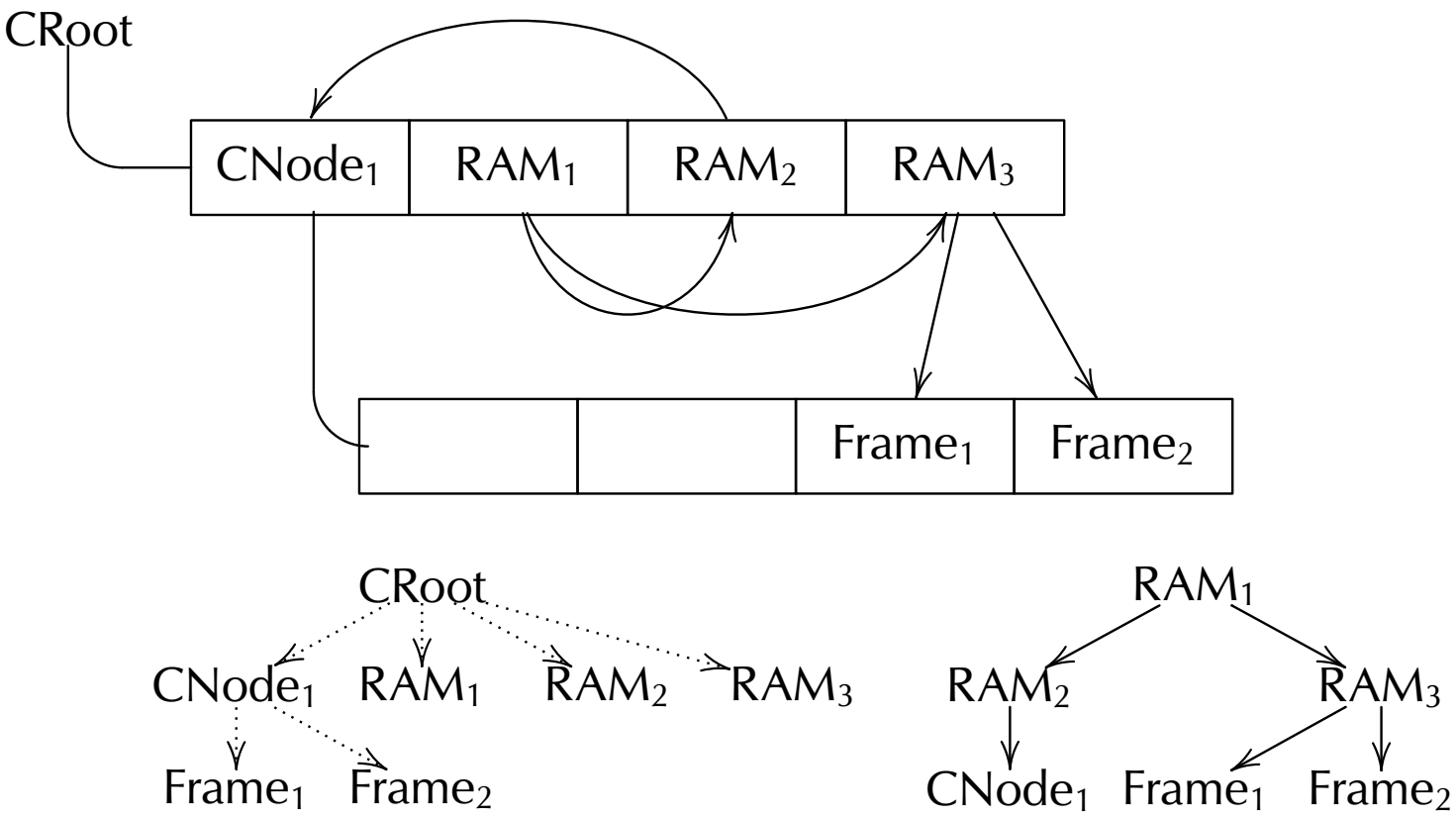
RAM caps may be split.

Retype



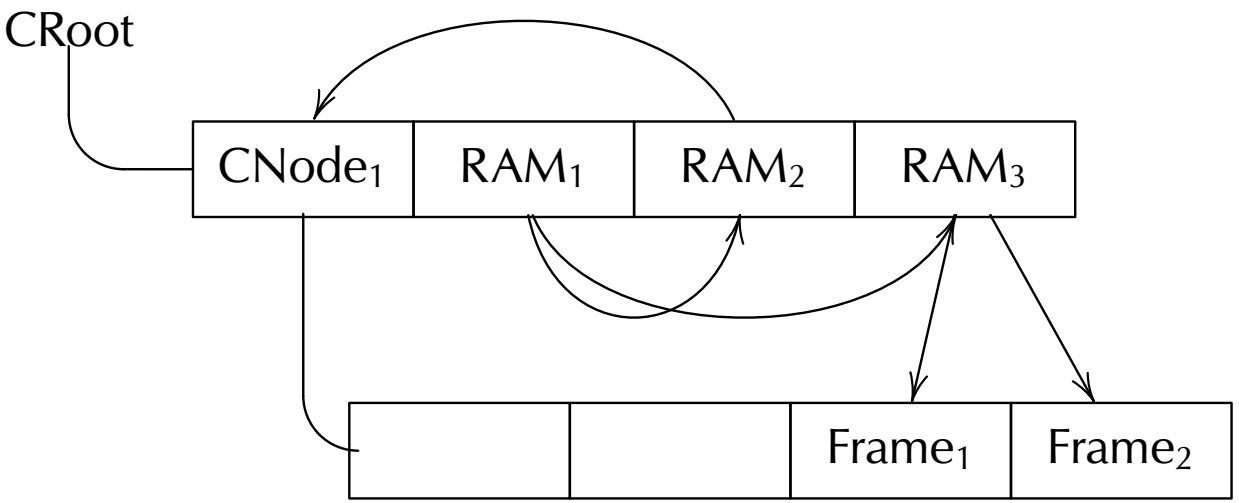
CNodes are created like other objects.

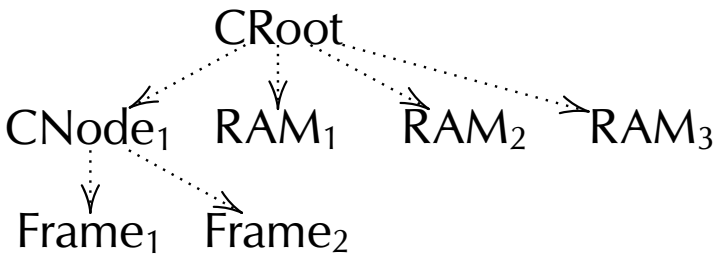
Retype

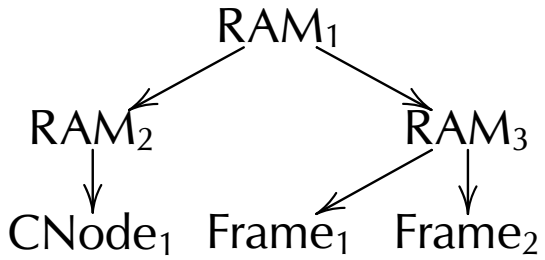


RAM must become Frames before being mapped.

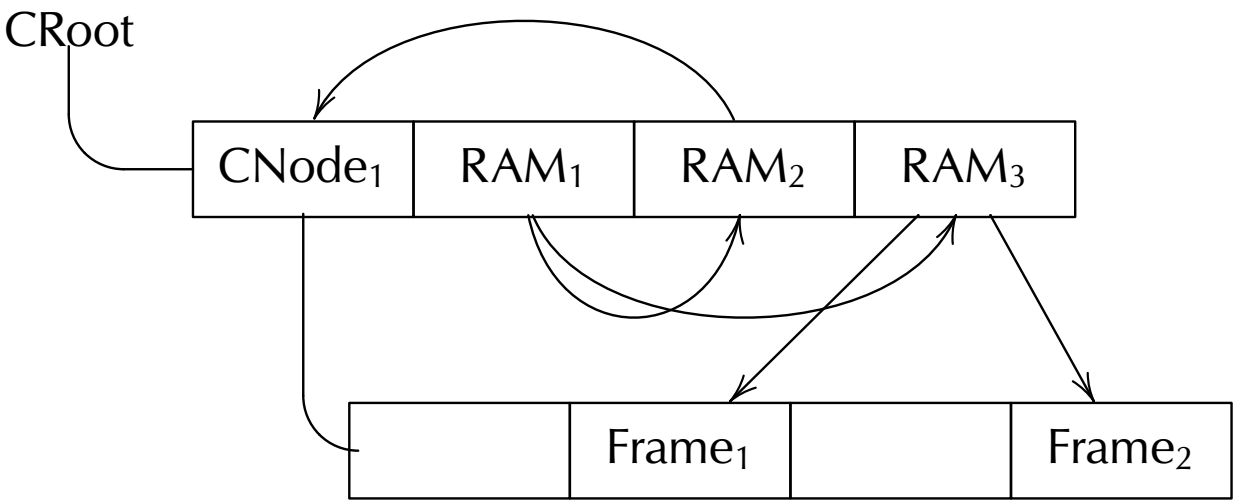
Move

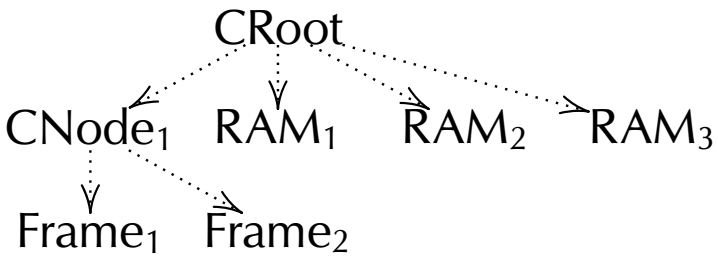

CRoot
CNode1
RAM1
RAM2
RAM3
Frame1
Frame2

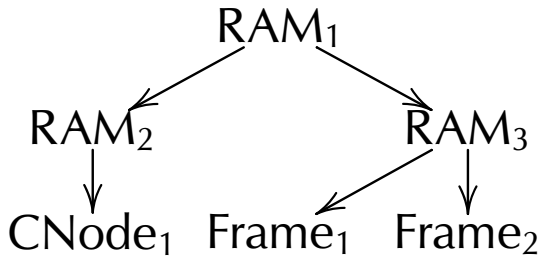

CRoot
CNode1
RAM1
RAM2
RAM3
Frame1
Frame2


RAM1
RAM2
RAM3
CNode1
Frame1
Frame2

Move

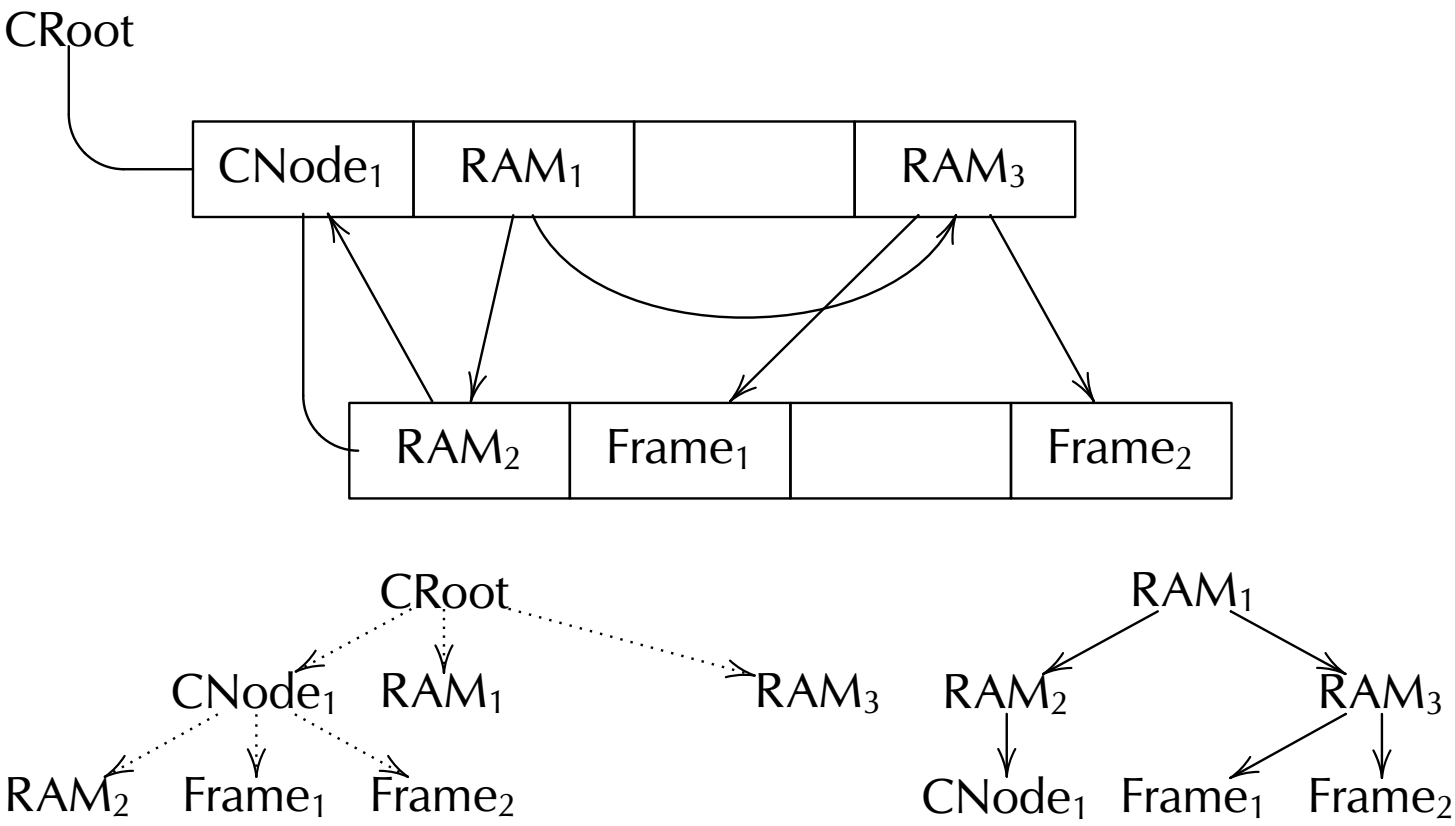






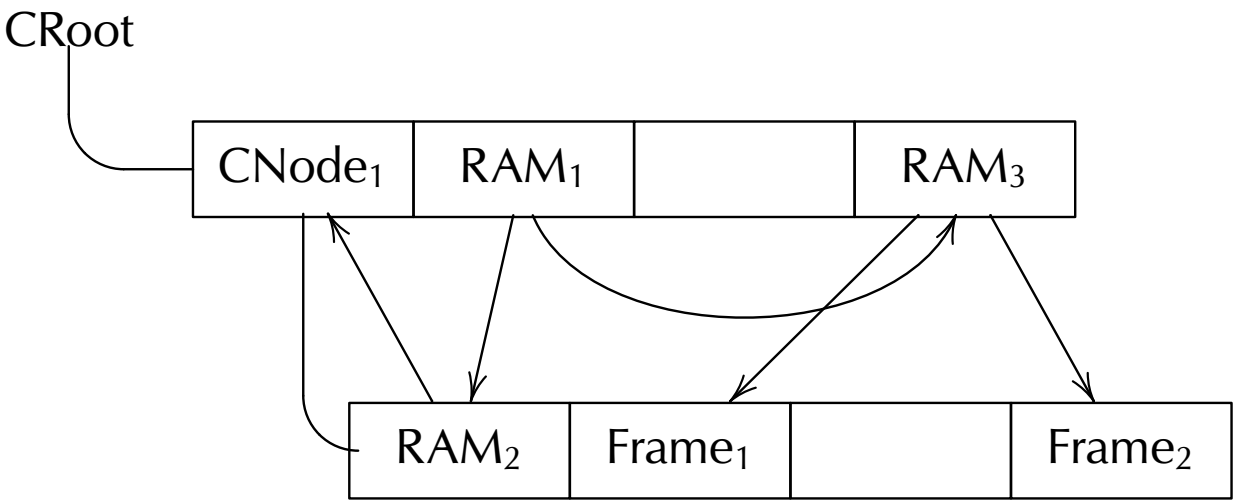
Moving within CNode doesn't affect trees.

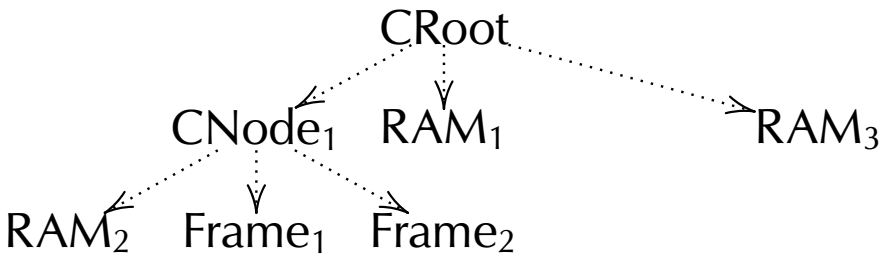
Move

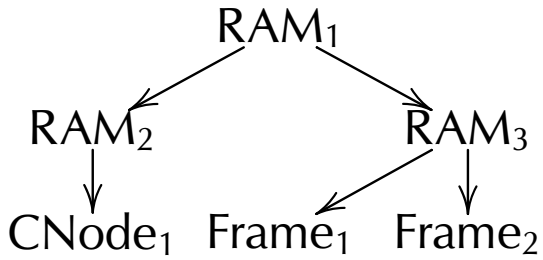


Moving between affects CSpace but not ancestry.

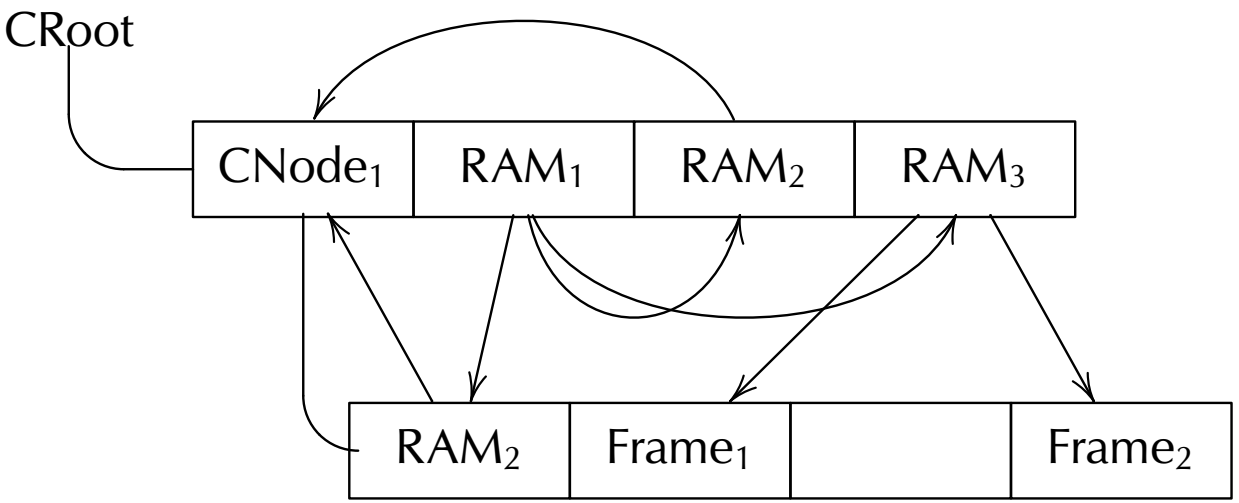
Copy

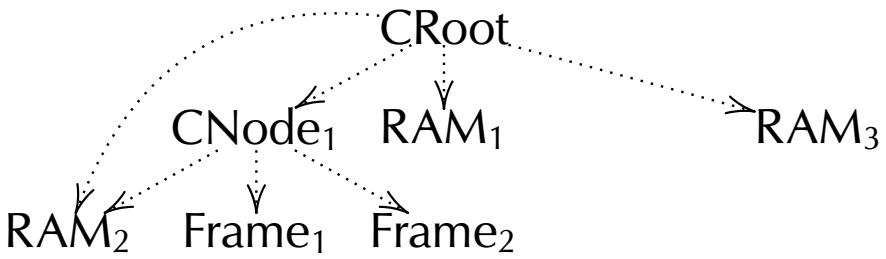

CRoot
CNode1
RAM1
RAM3
RAM2
Frame1
Frame2

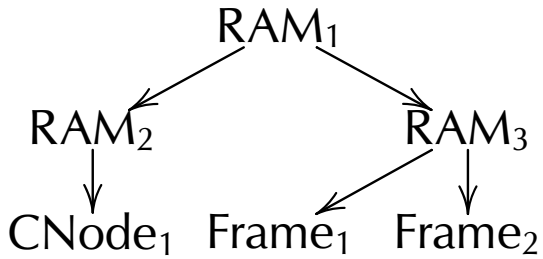

CRoot
CNode1
RAM1
RAM3
RAM2
Frame1
Frame2


RAM1
RAM2
RAM3
CNode1
Frame1
Frame2

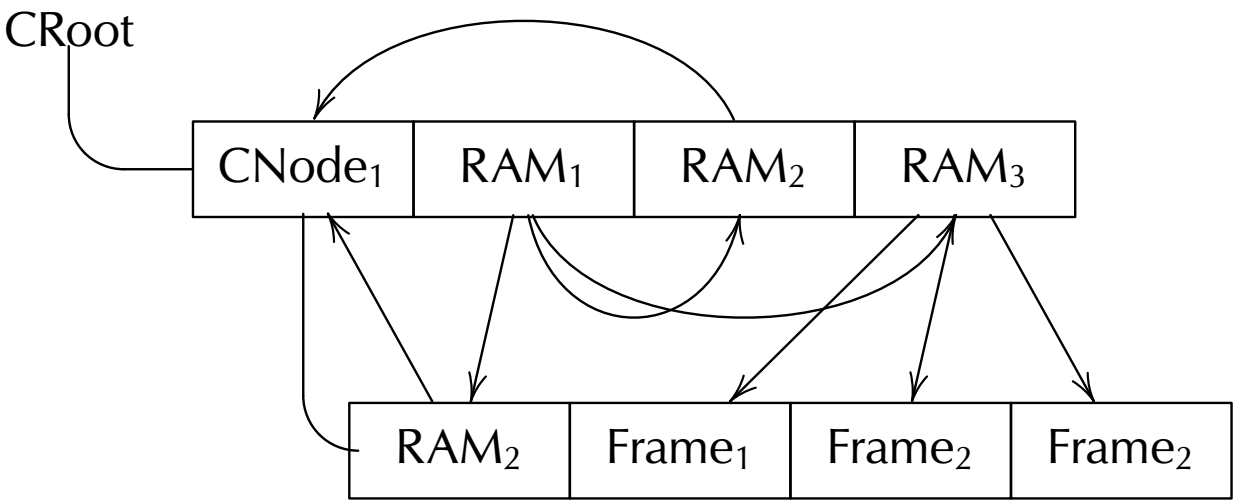
Copy

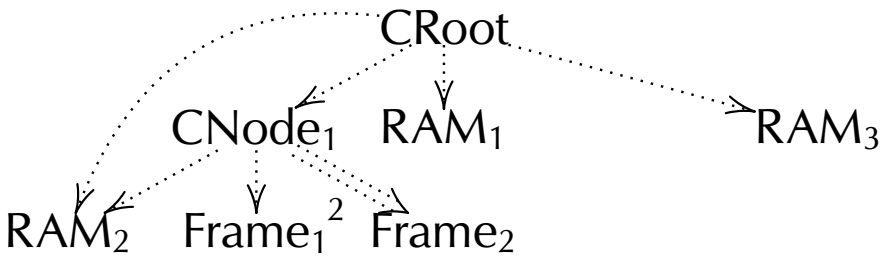

CRoot
CNode1
RAM1
RAM2
RAM3
RAM2
Frame1
Frame2

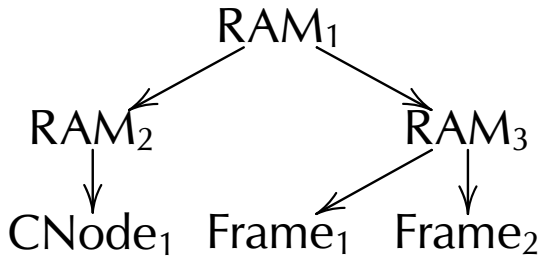

CRoot
CNode1
RAM1
RAM3
RAM2
Frame1
Frame2


RAM1
RAM2
RAM3
CNode1
Frame1
Frame2

Copy

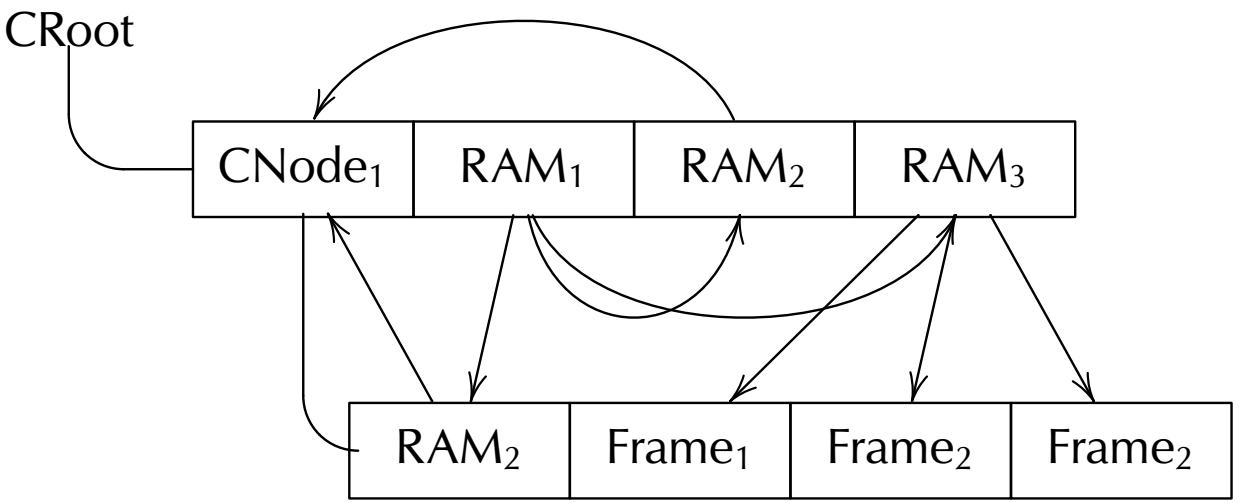


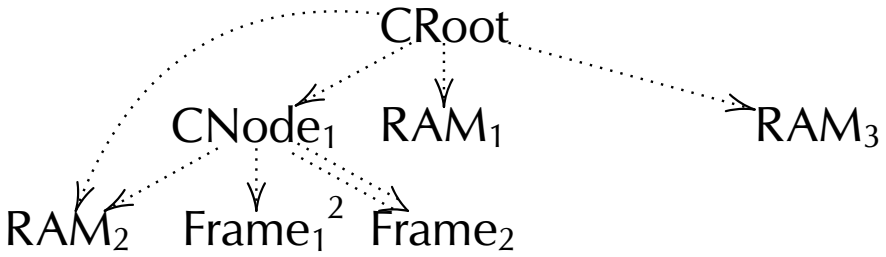


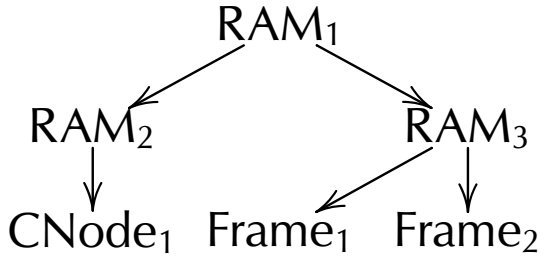


Copies make the CSpace a proper DAG.

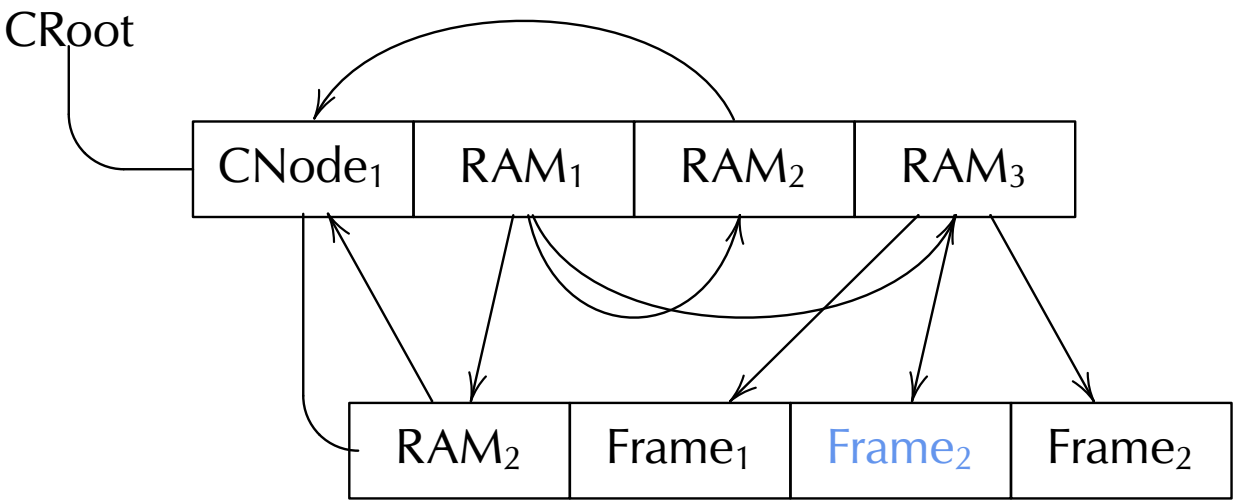
Delete

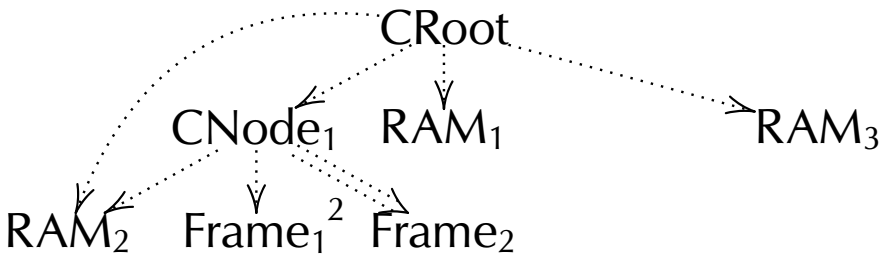


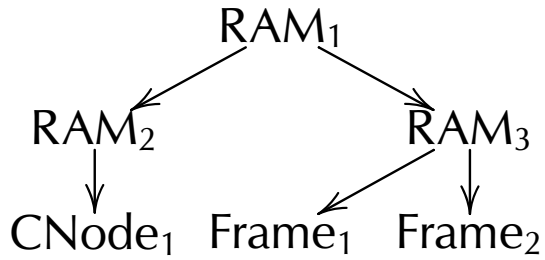




Delete

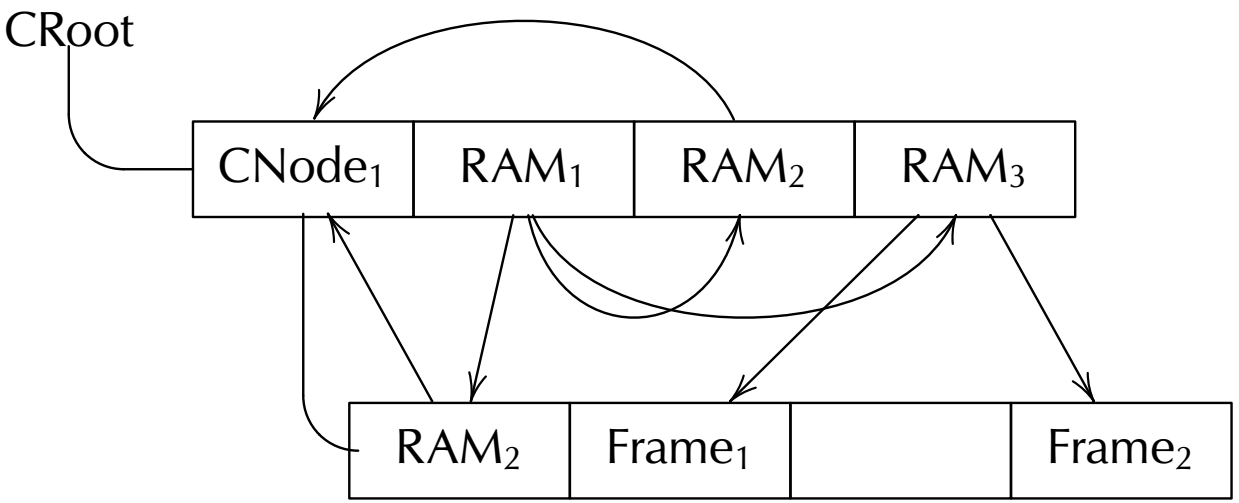


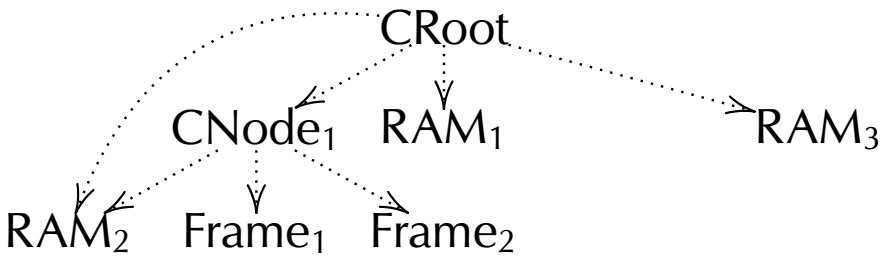


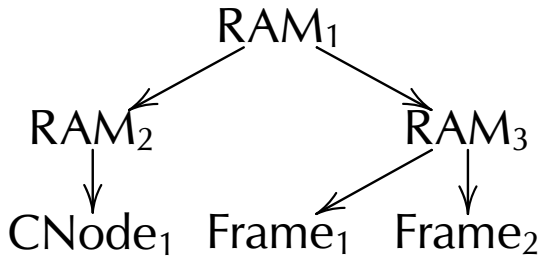


Deleting non-final leaf caps is easy.

Delete

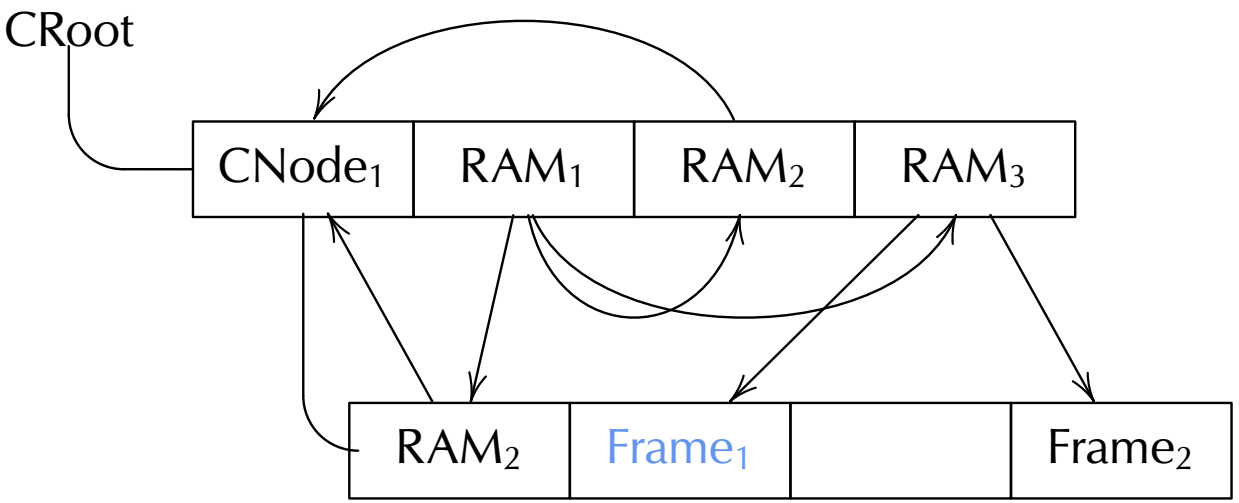


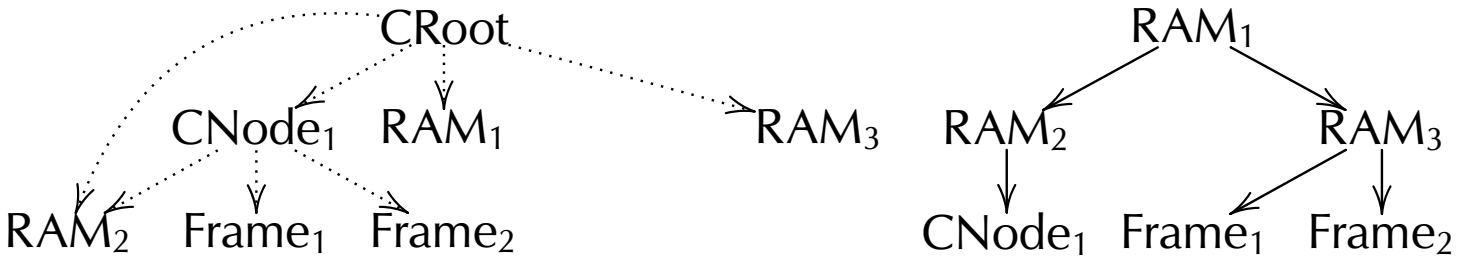




Deleting non-final leaf caps is easy.

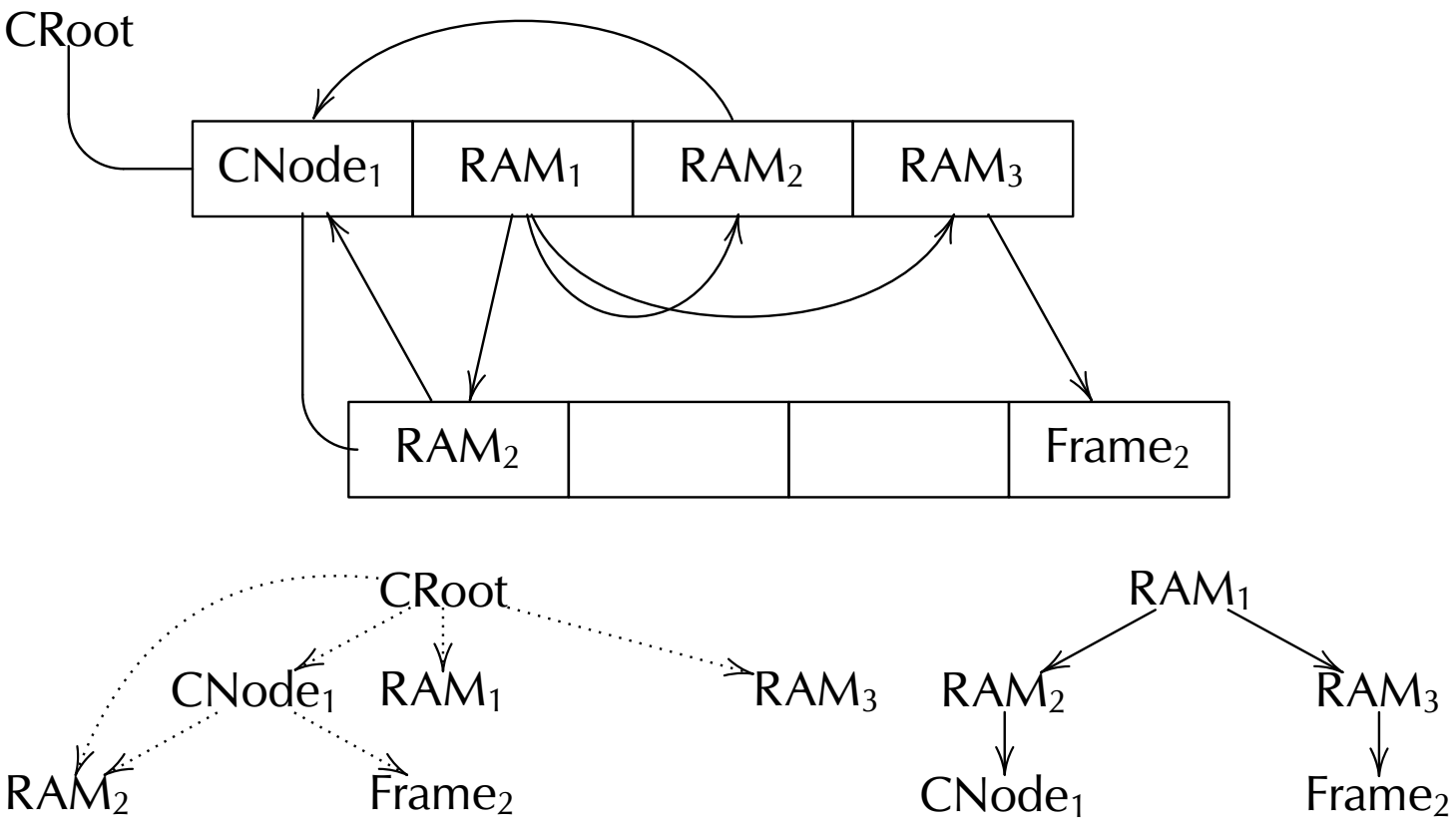
Delete





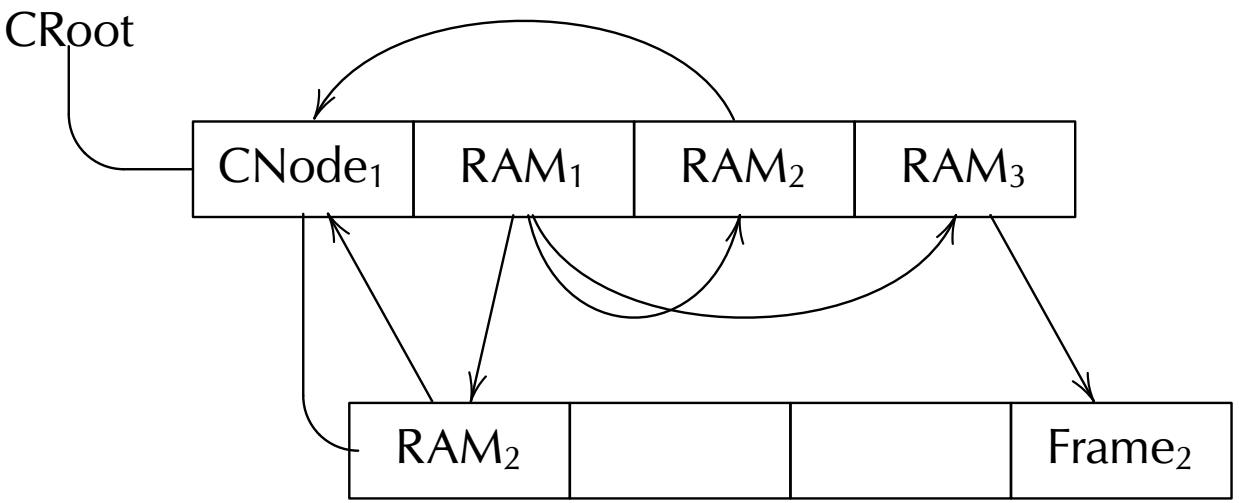
Deleting the last cap deletes the object.

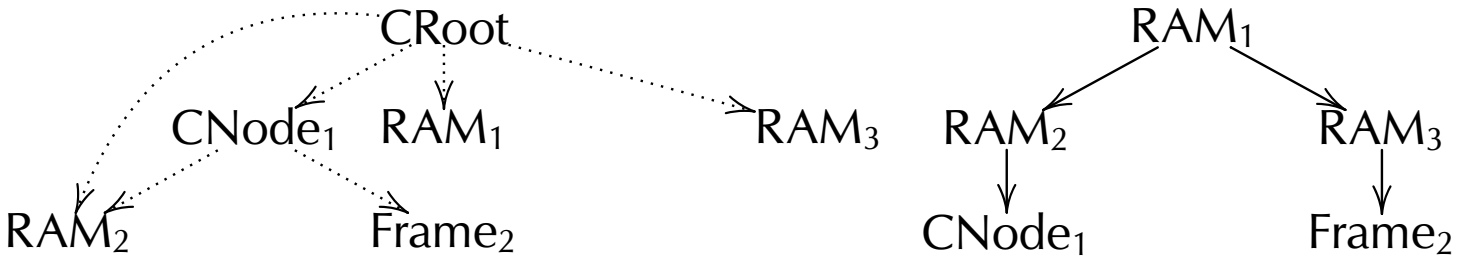
Delete



Deleting the last cap deletes the object.

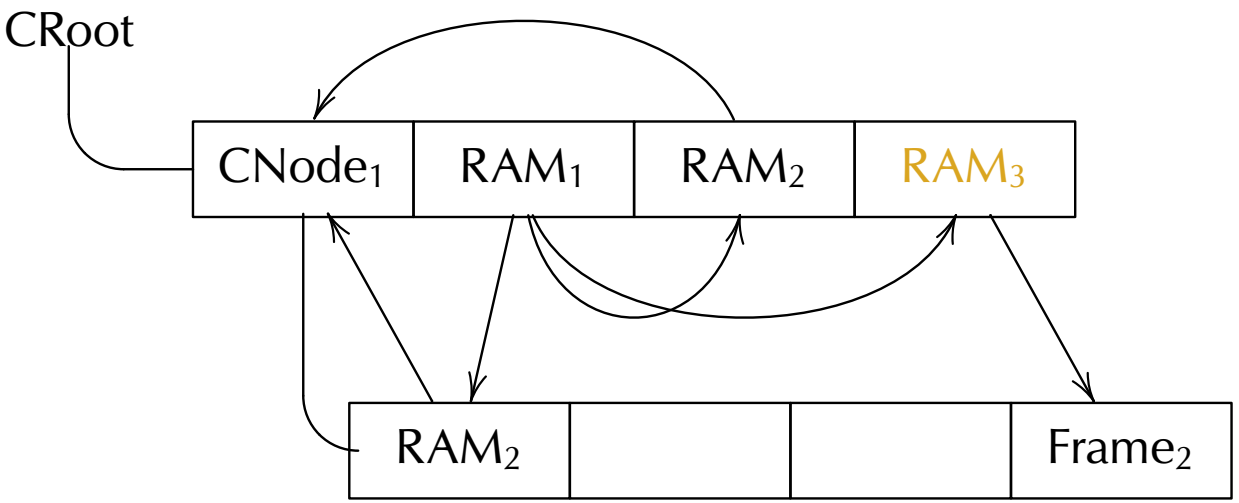
Revoke

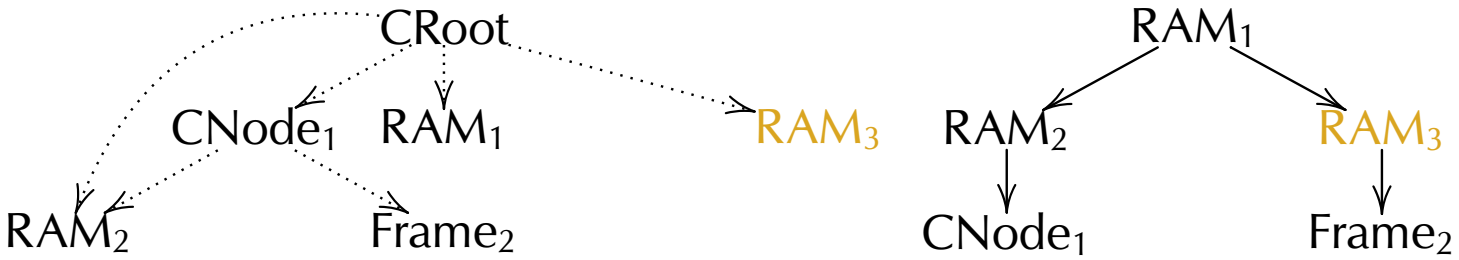




Revoke walks the ancestry tree.

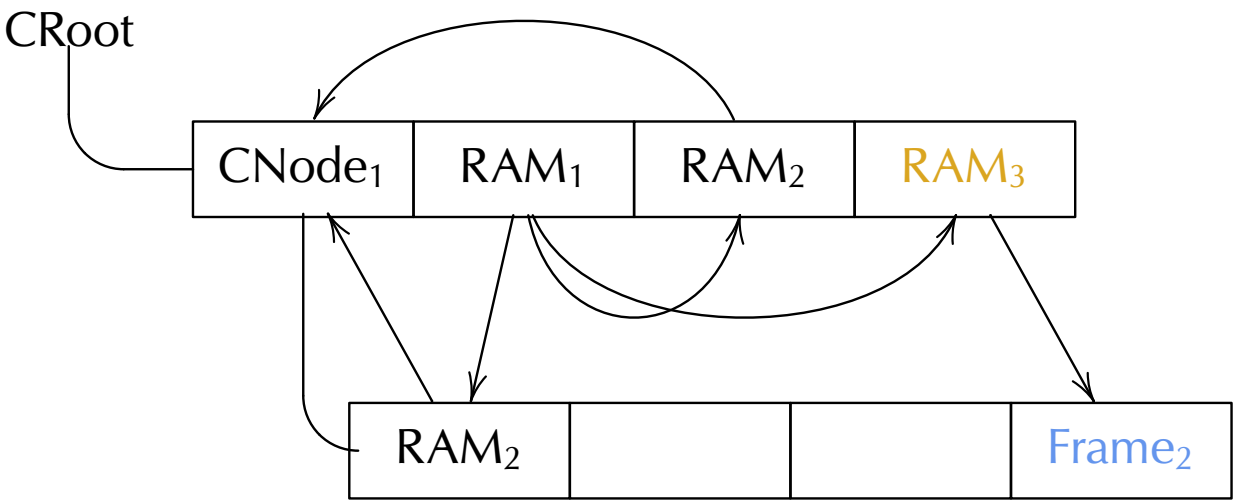
Revoke

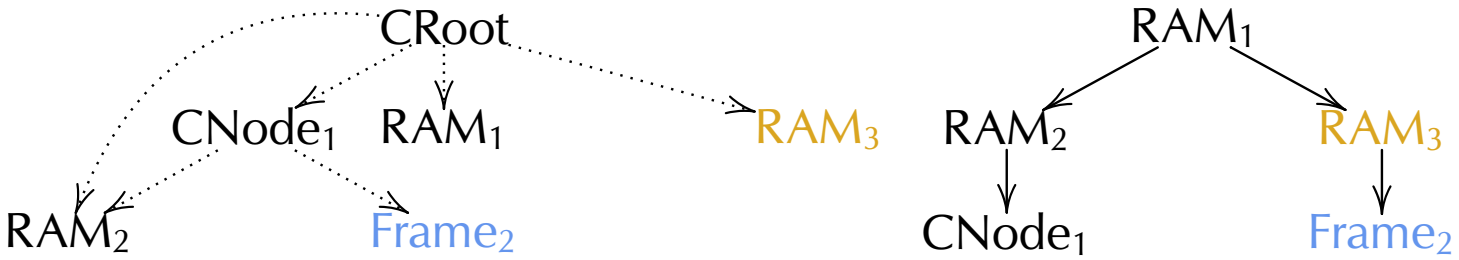




Mark RAM_3 for revocation.

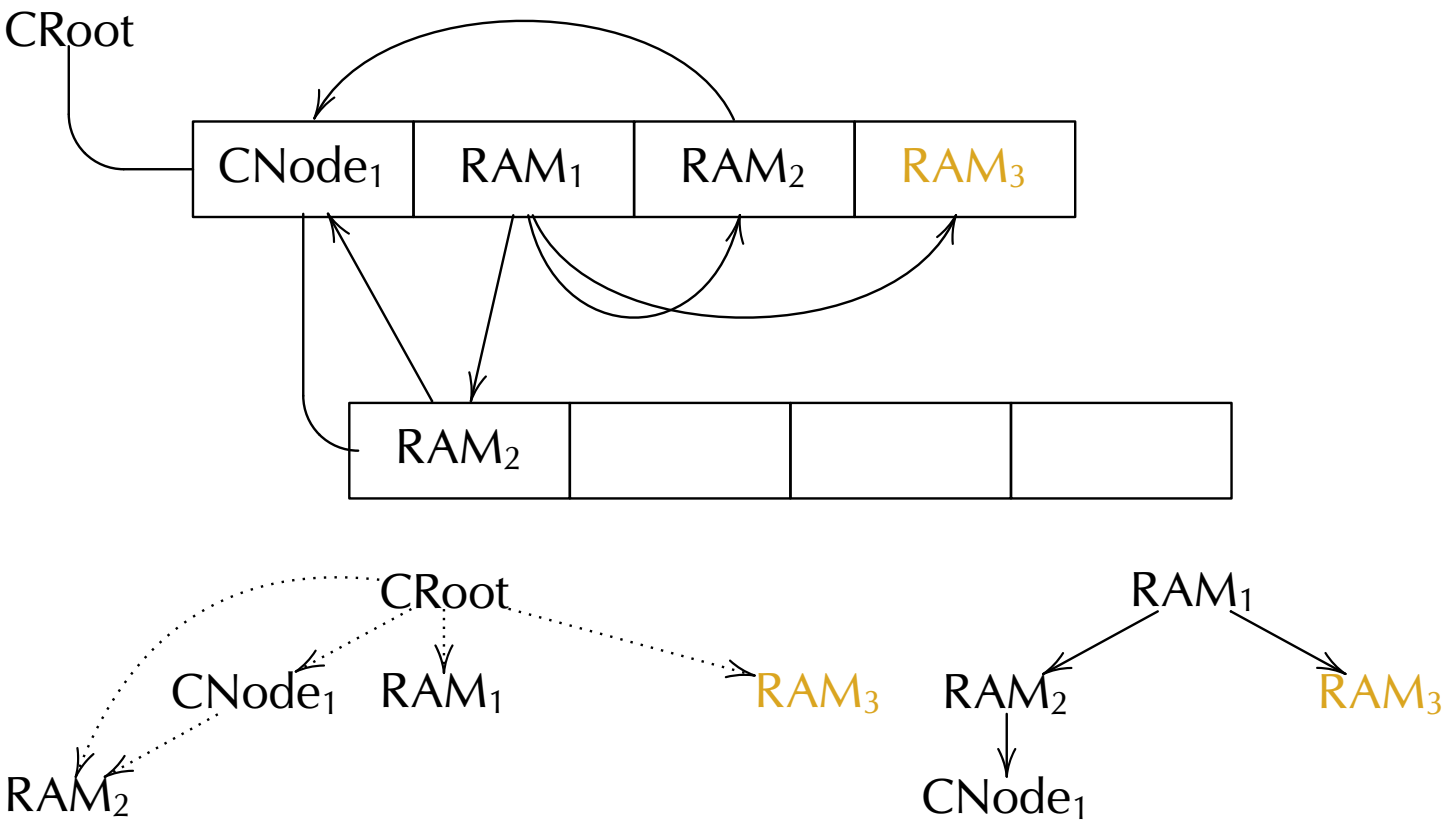
Revoke





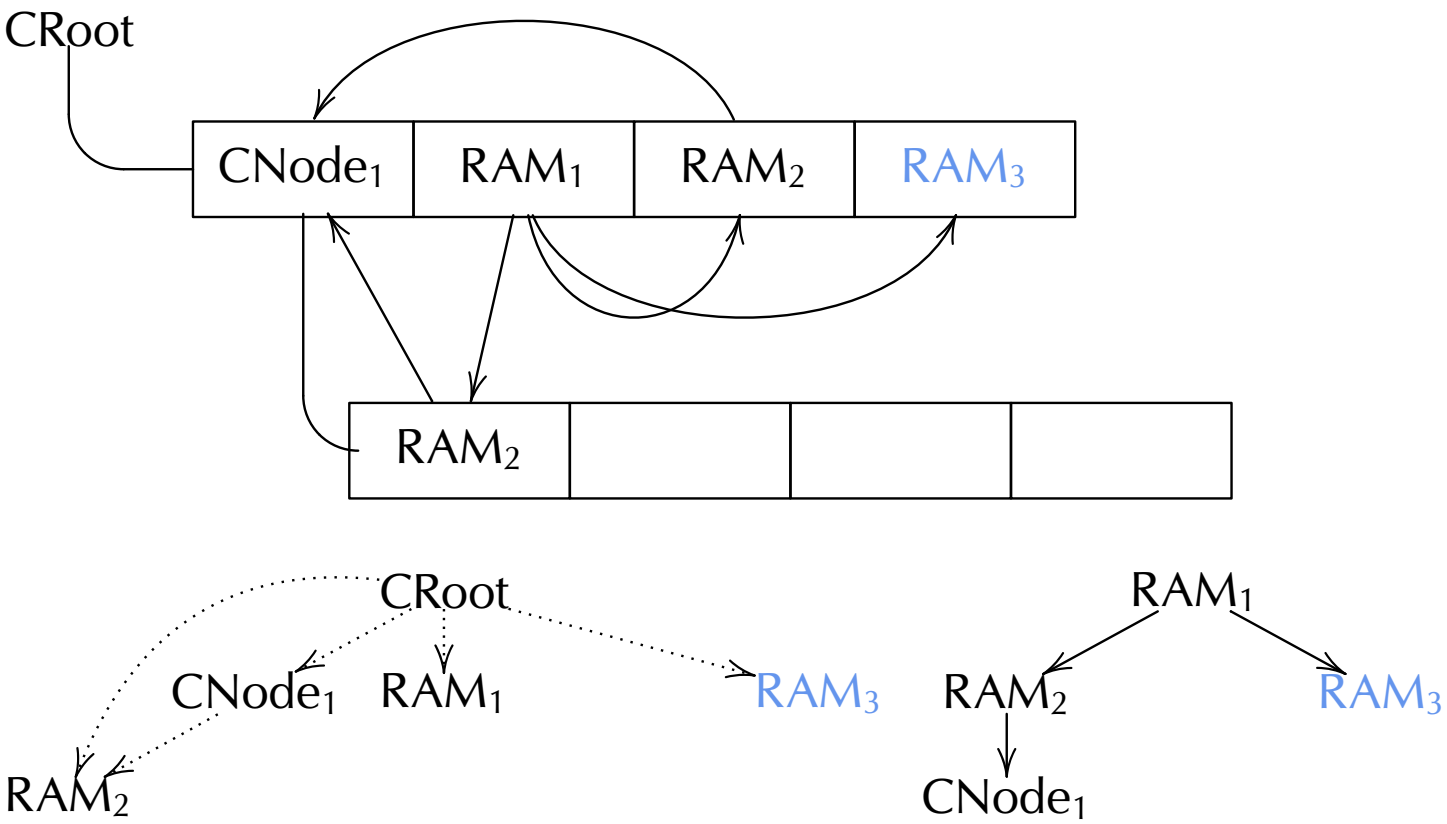
Mark its descendents for deletion.

Revoke



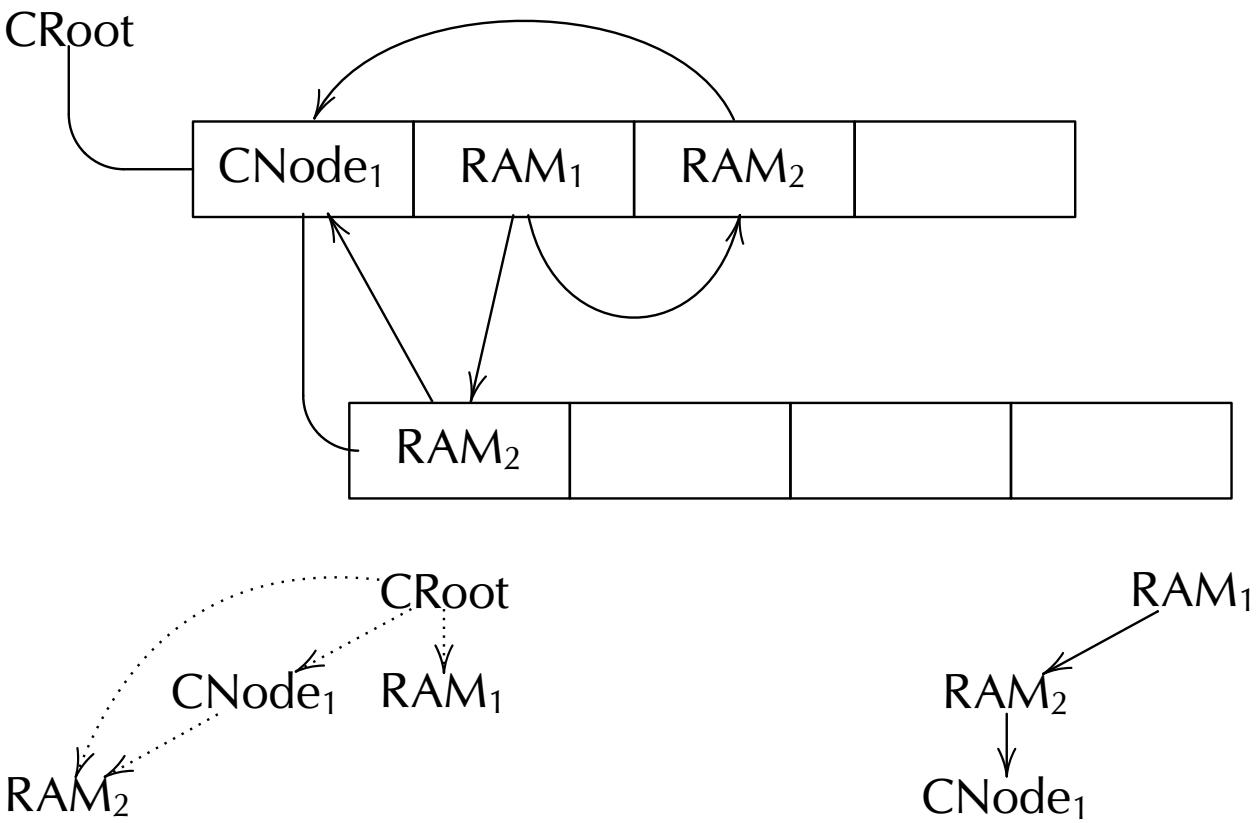
Delete them.

Revoke

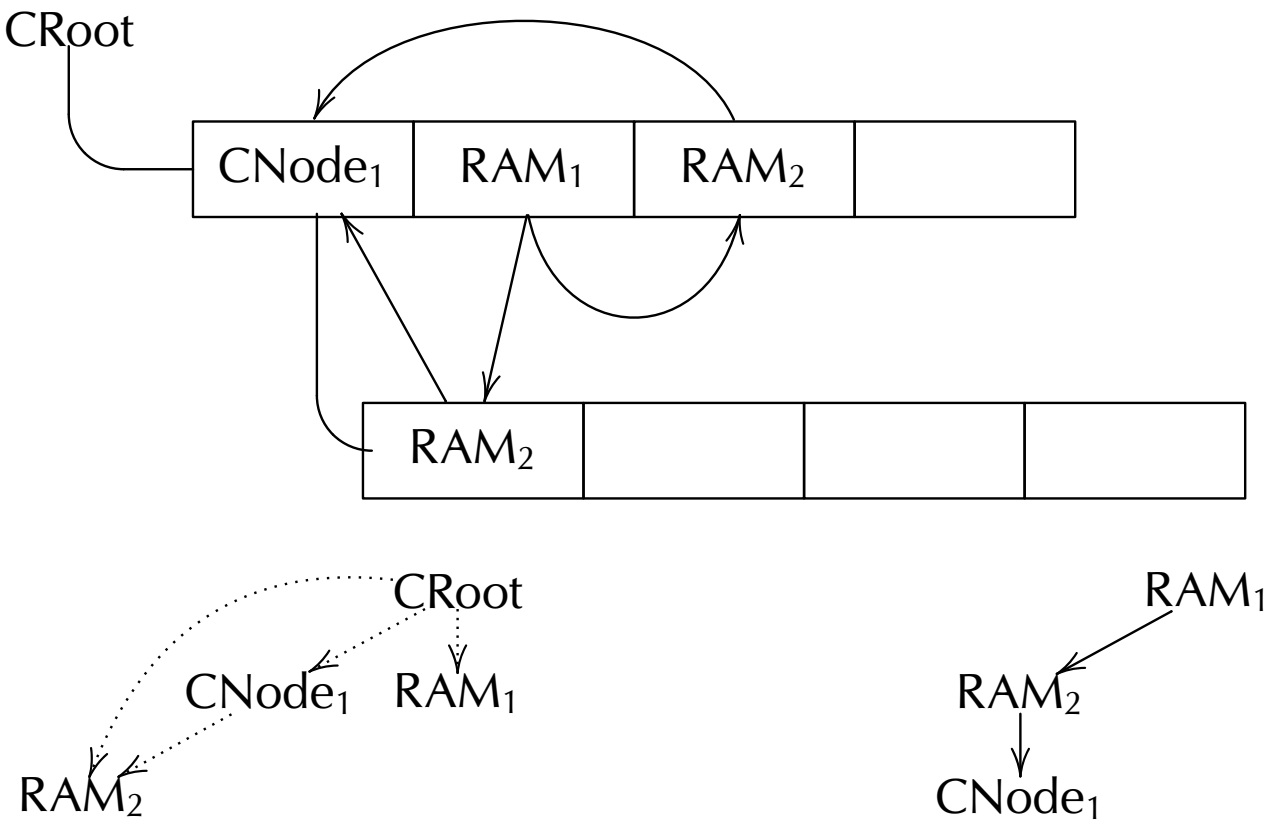


The root can now be deleted, if required.

Revoke

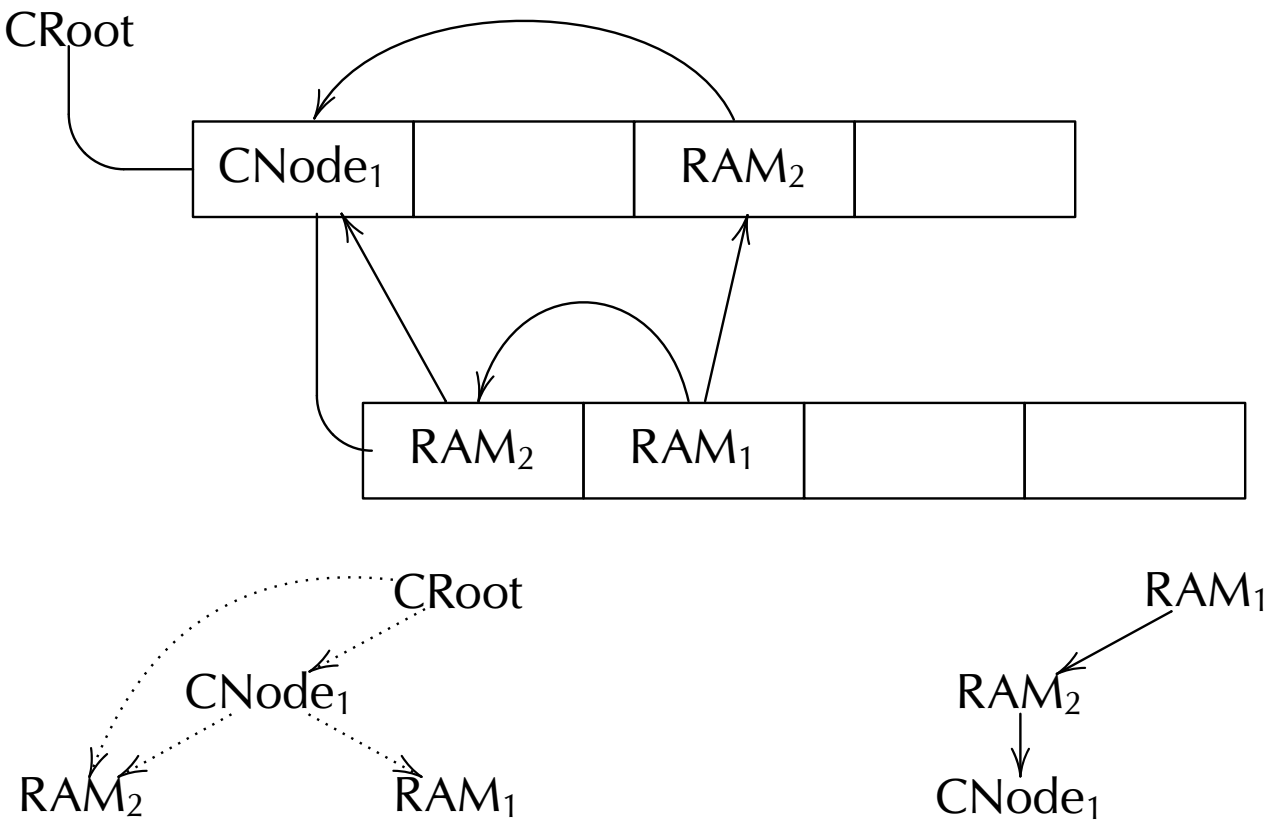

CRoot
CNode1
RAM1
RAM2
RAM2
CRoot
CNode1
RAM1
RAM2
RAM1
RAM2
CNode1

Recursive Revoke & Delete



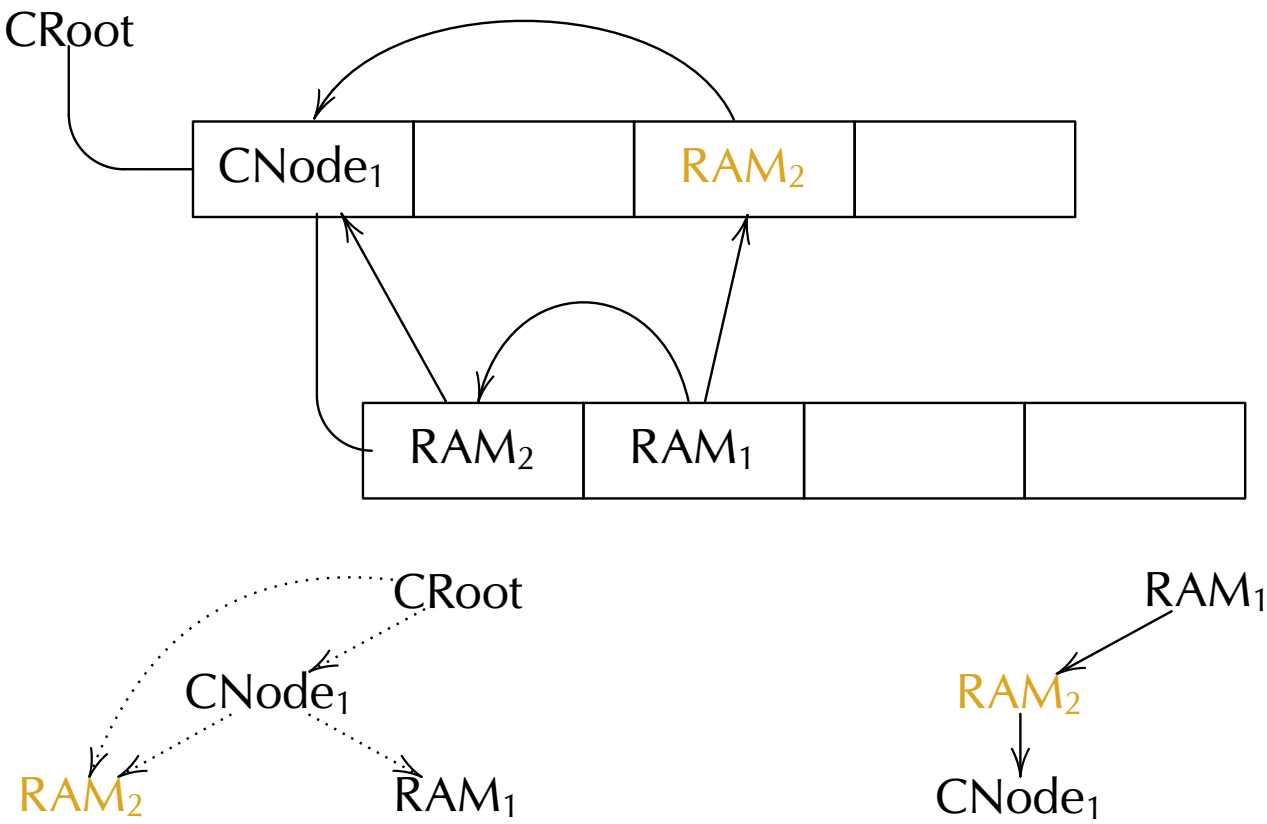
Move RAM$_1$.

Recursive Revoke & Delete



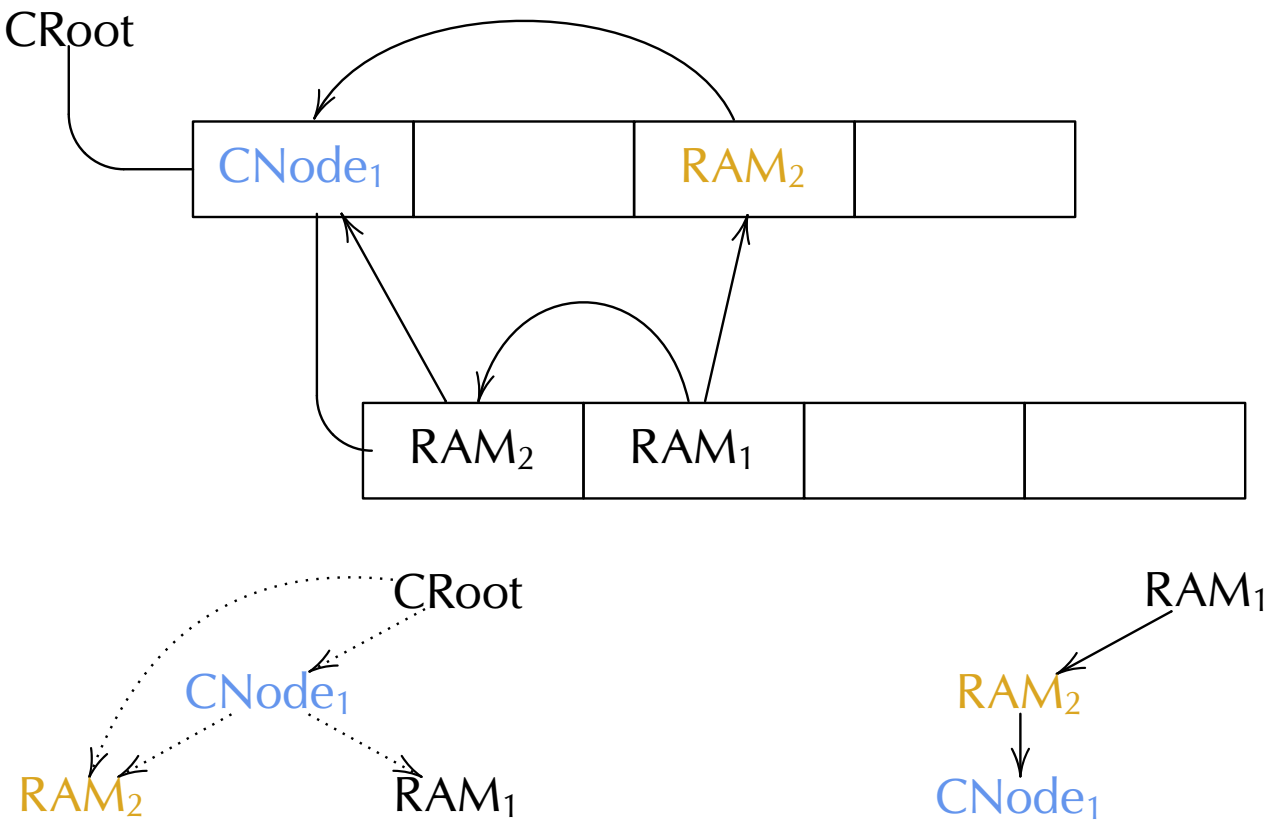
There's now a loop, with links in *both* trees.

Recursive Revoke & Delete



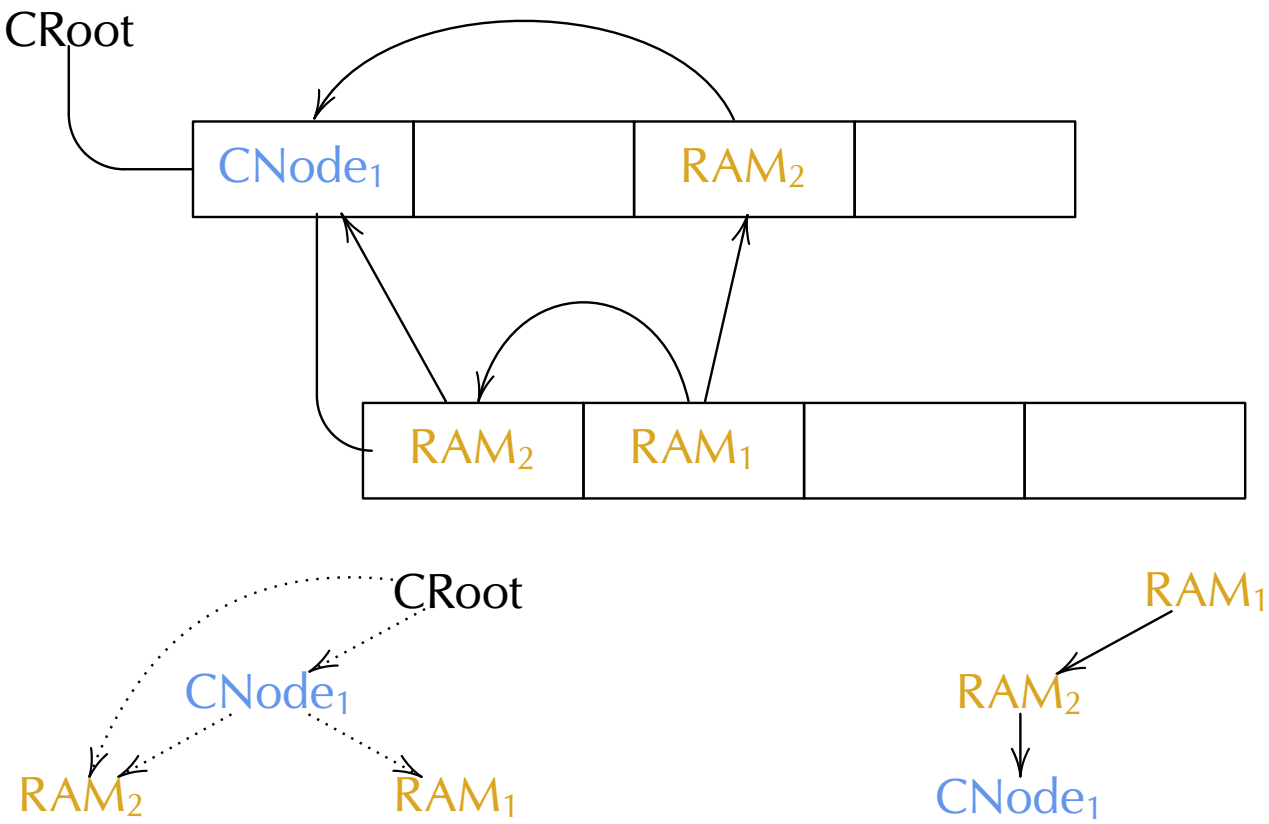
Let's revoke RAM_2, a *child*.

Recursive Revoke & Delete



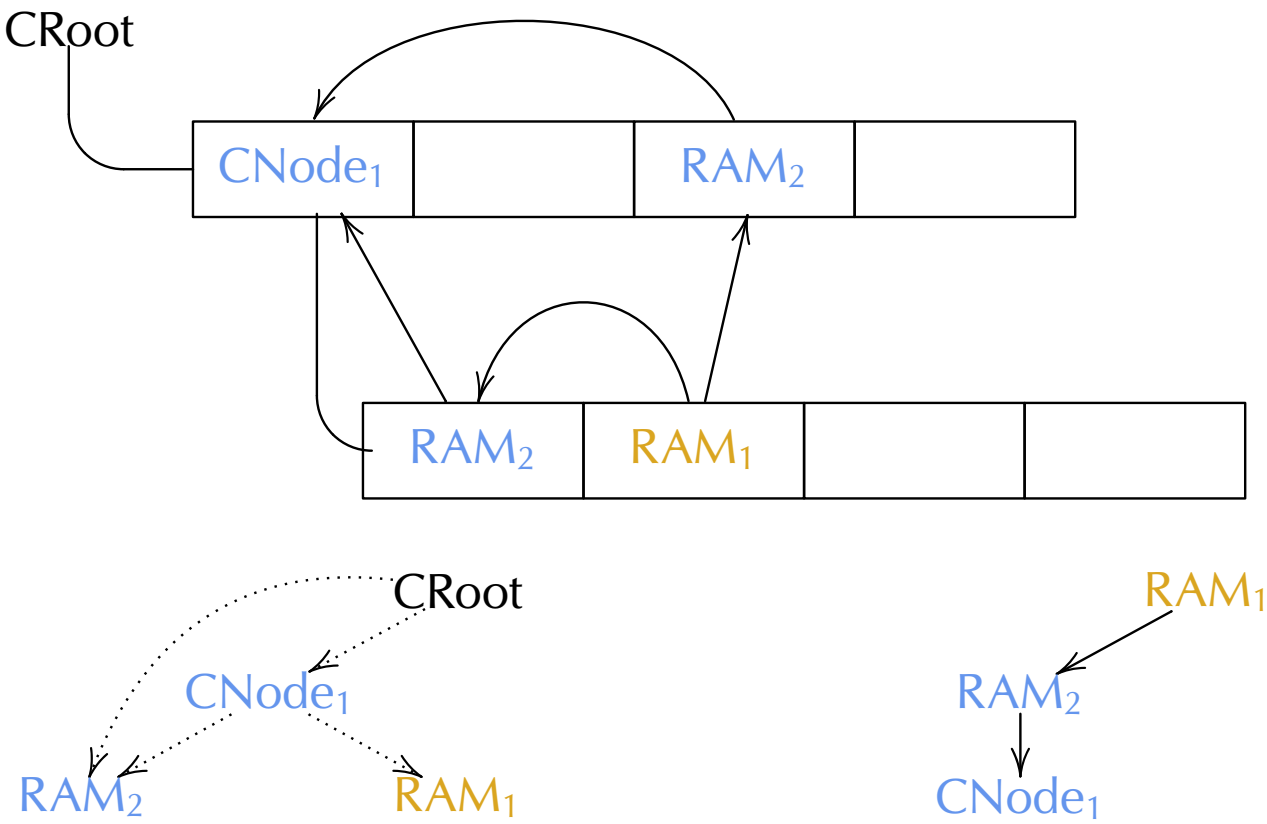
Mark its descendents for deletion.

Recursive Revoke & Delete



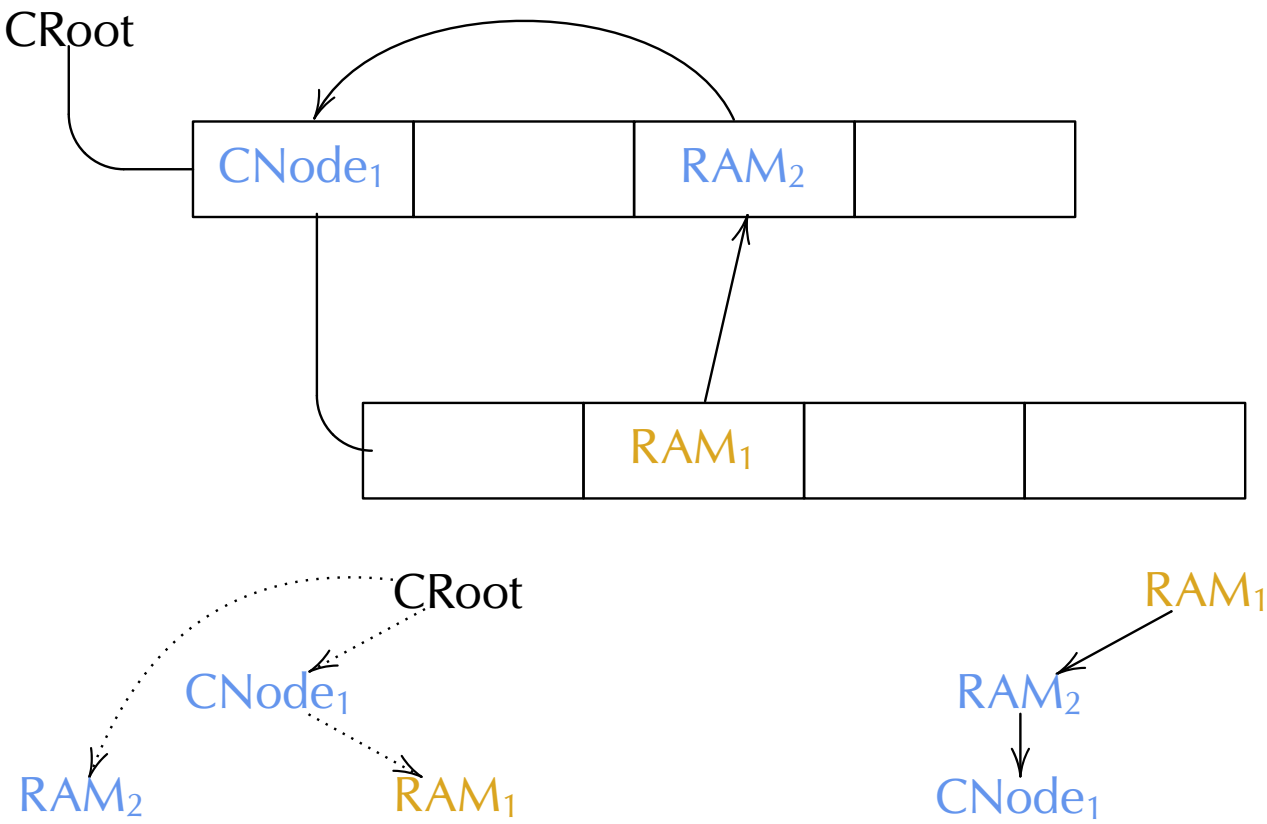
Deleting a CNode first deletes (revokes) its contents.

Recursive Revoke & Delete



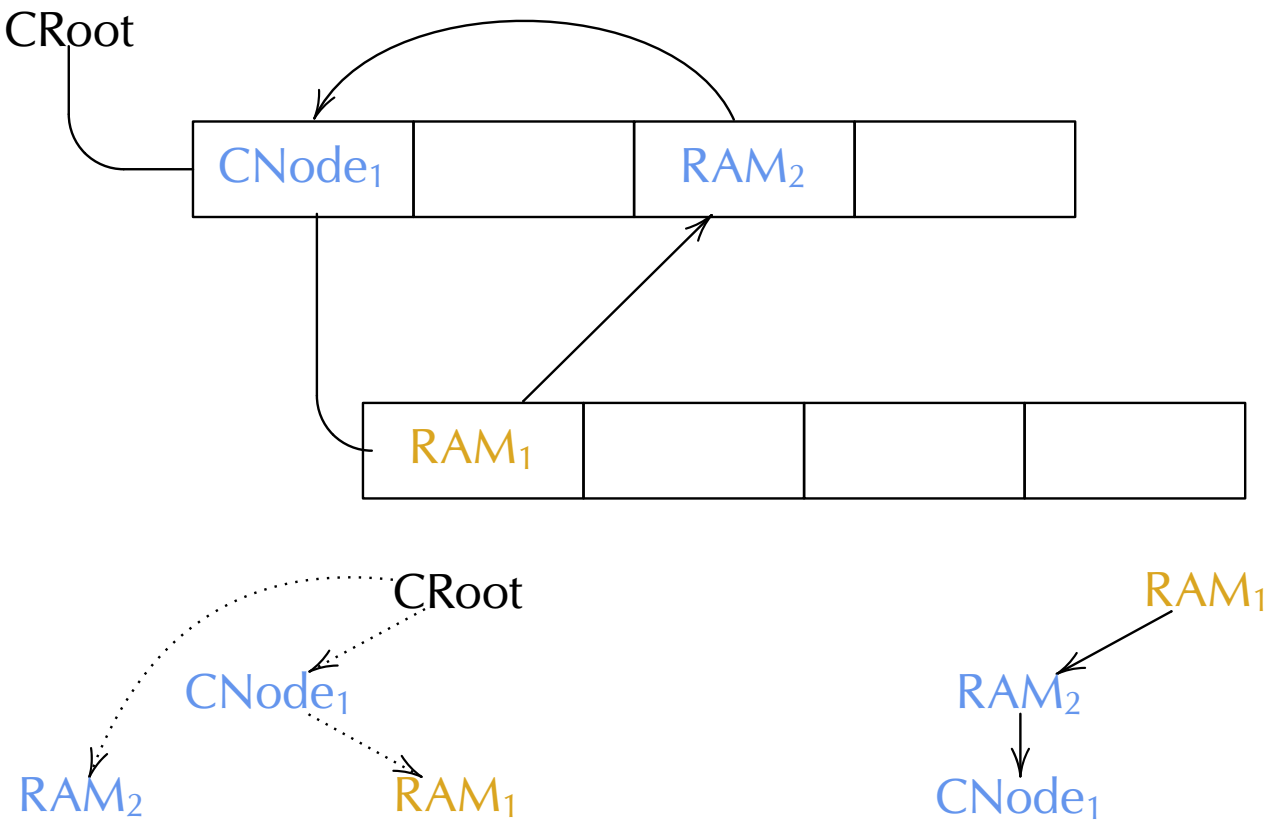
Revoking RAM_2 *deletes* RAM_1.

Recursive Revoke & Delete

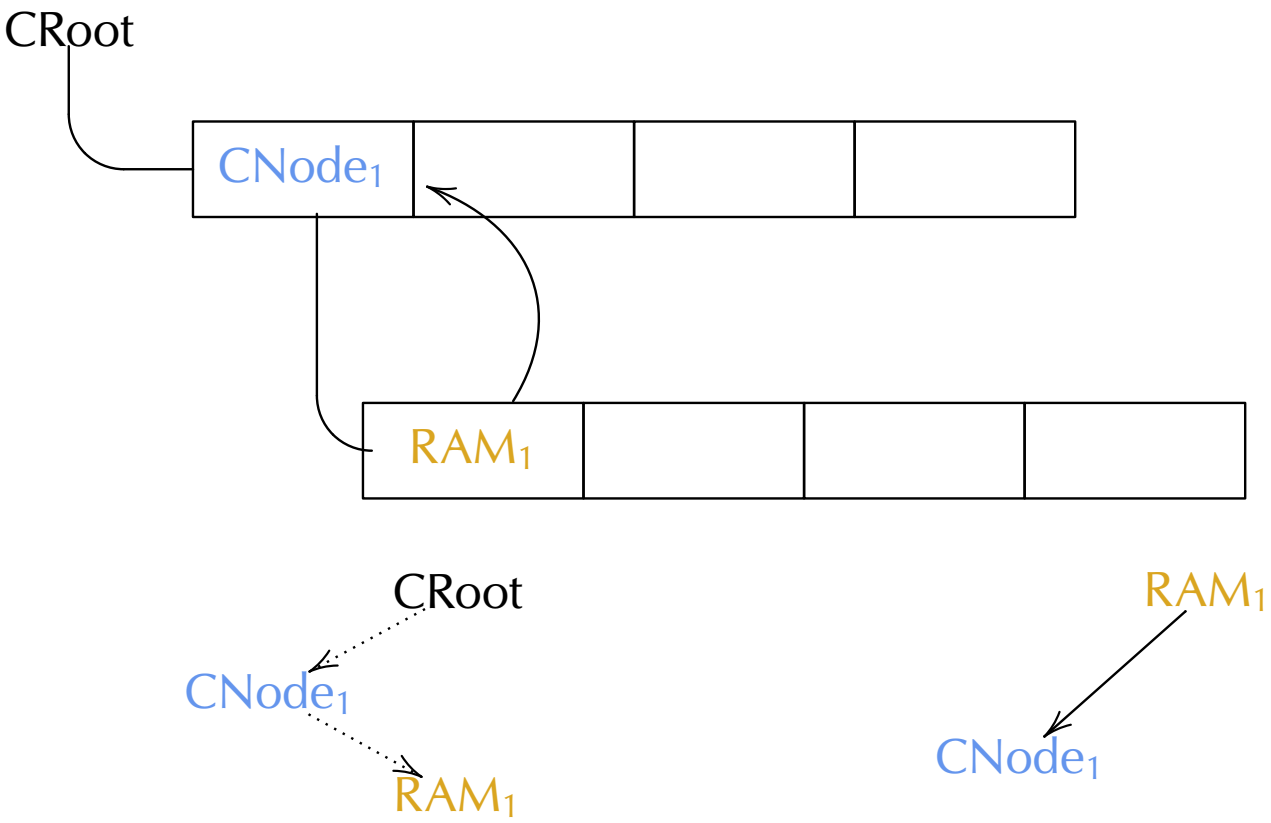


Delete starts bottom up.
This RAM_2 cap is safe to delete.

Recursive Revoke & Delete

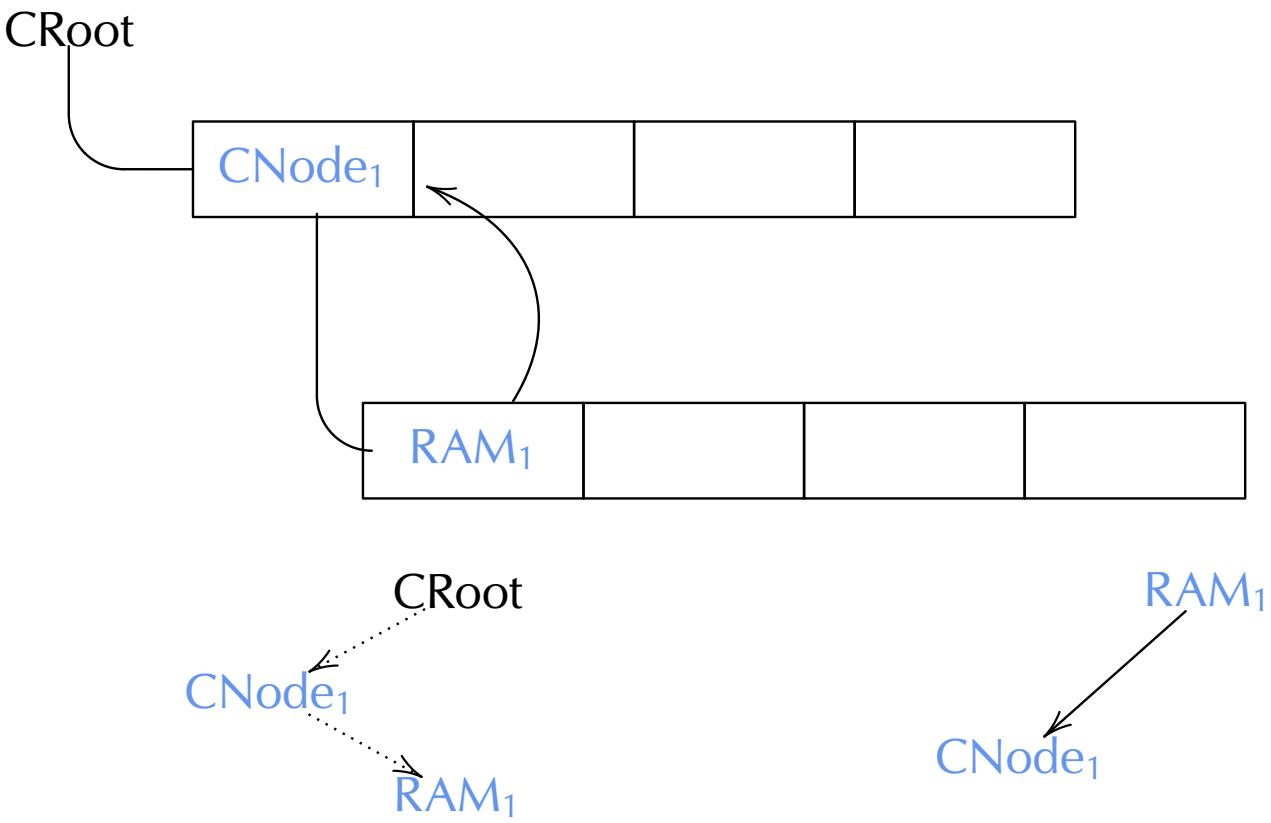


Recursive Revoke & Delete



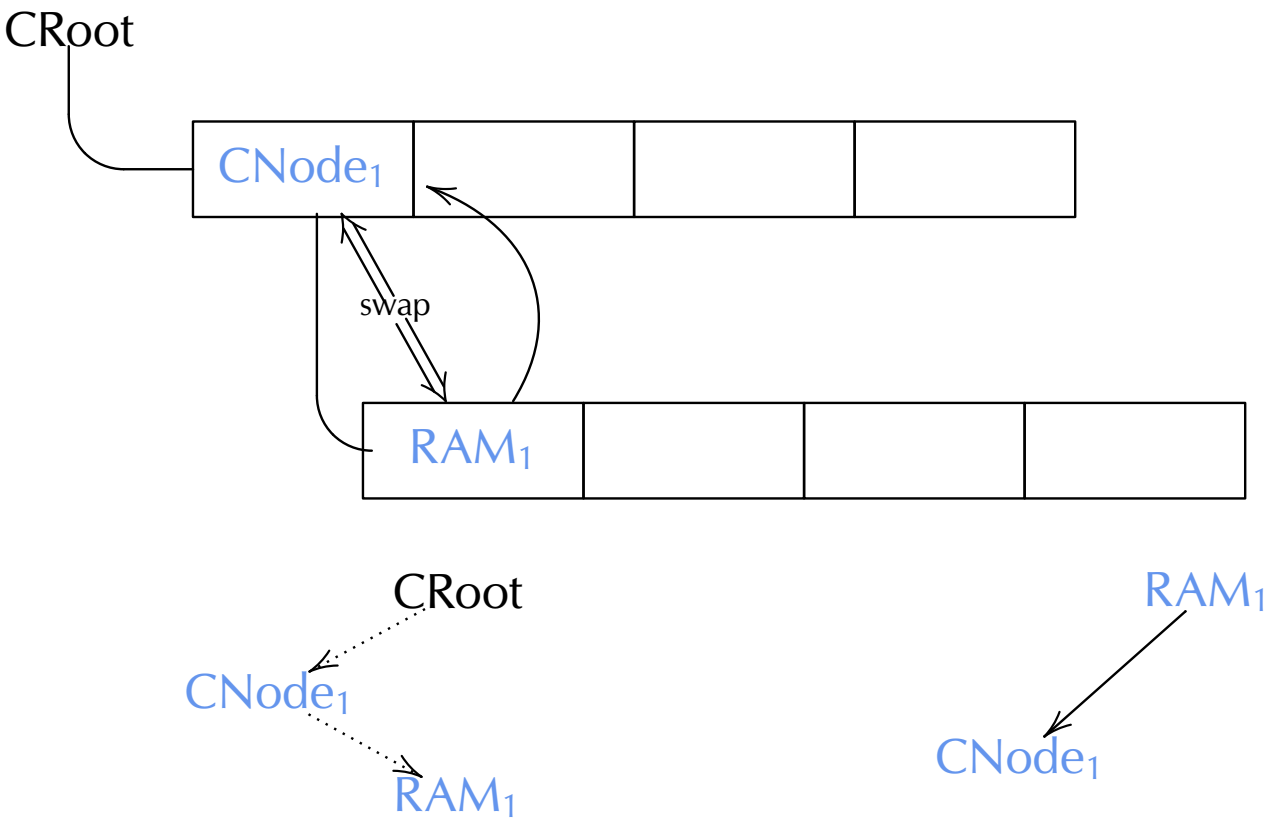
When RAM_2 is destroyed, RAM_1 adopts children.
Now we've got an irreducible cycle.

Recursive Revoke & Delete



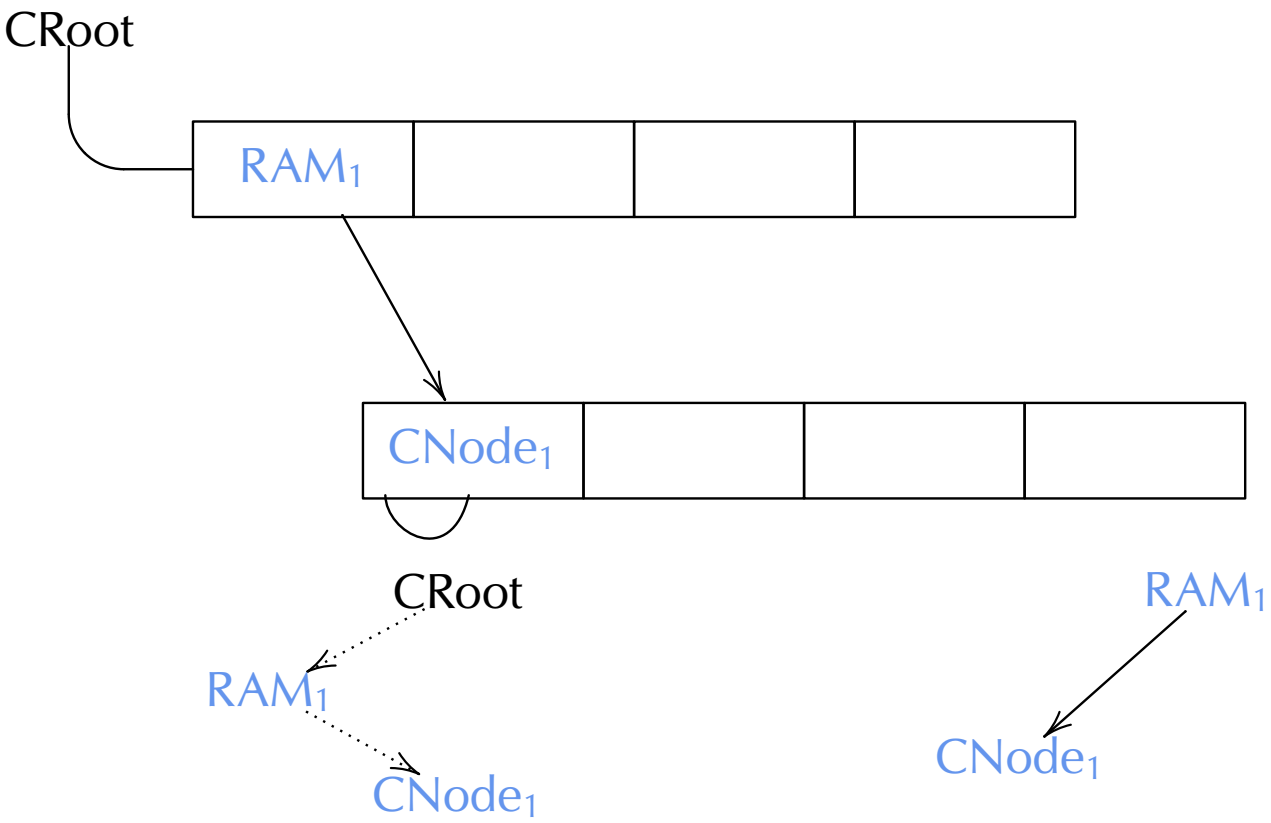
RAM_1's revoke is finished, now delete it, but how?

Recursive Revoke & Delete



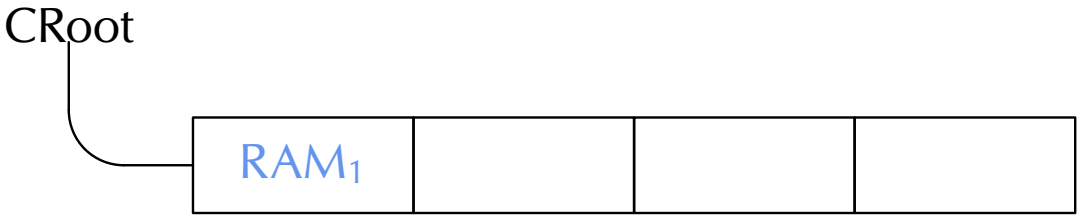
In seL4, we swap the last two caps.

Recursive Revoke & Delete



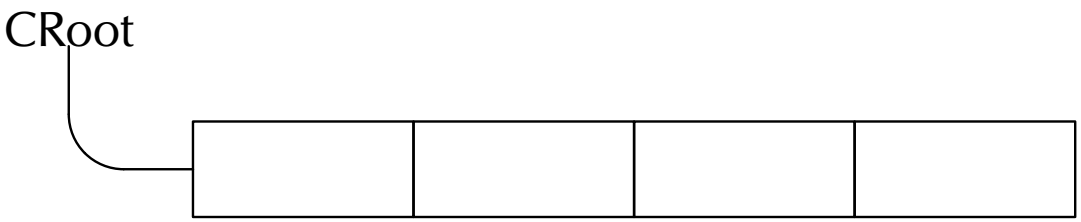
$CNode_1$ can now safely be deleted.

Recursive Revoke & Delete



Finally, RAM_1 goes too.

Recursive Revoke & Delete

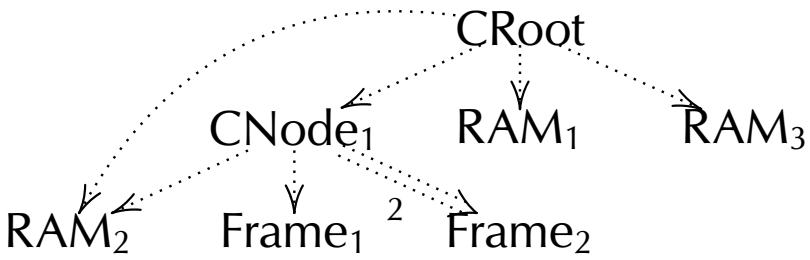


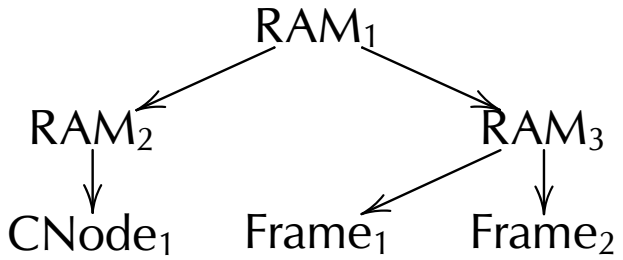
CRoot

This process accidentally destroyed its whole world.

Invariants

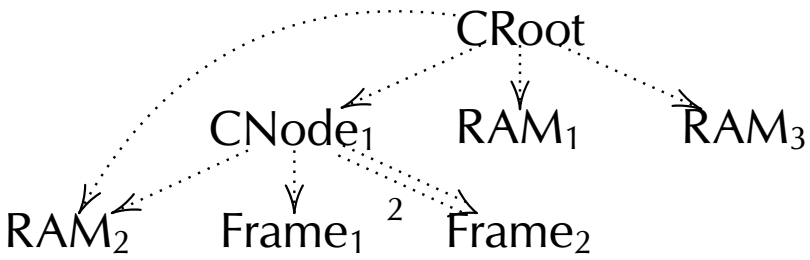
In seL4

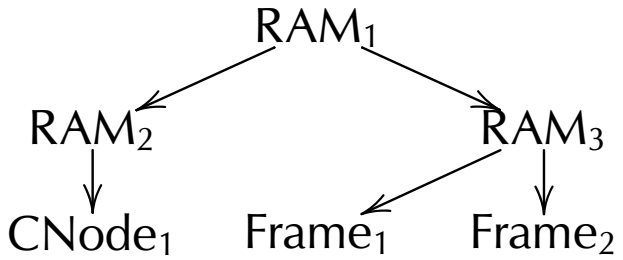




Invariants

In seL4

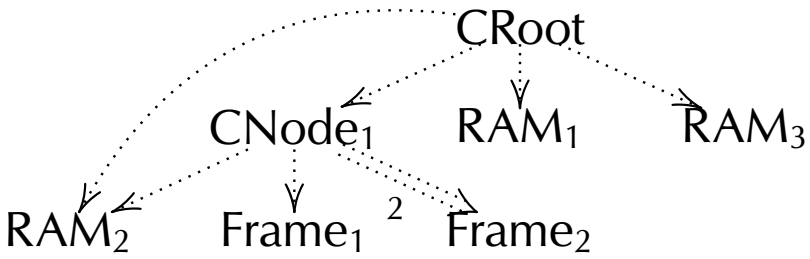


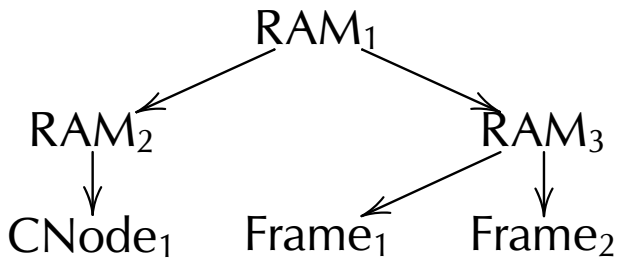


- Ancestry is a tree (forest).

Invariants

In seL4

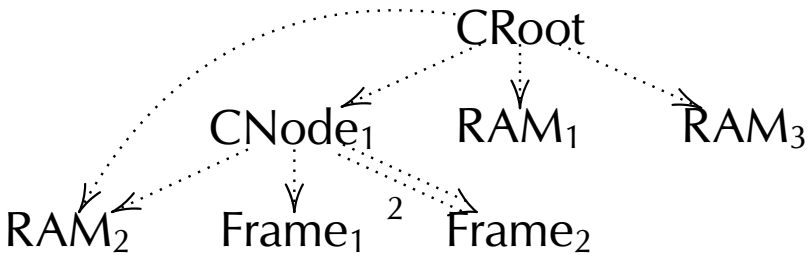


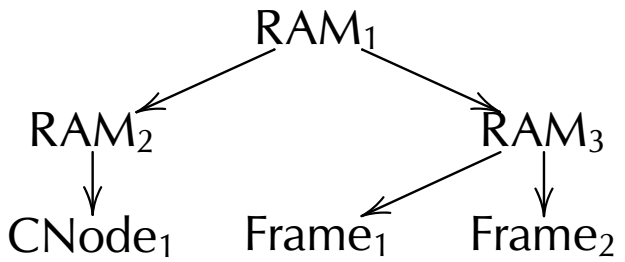


- Ancestry is a tree (forest).
- $\exists$Object $\rightarrow \exists$Cap.

Invariants

In seL4

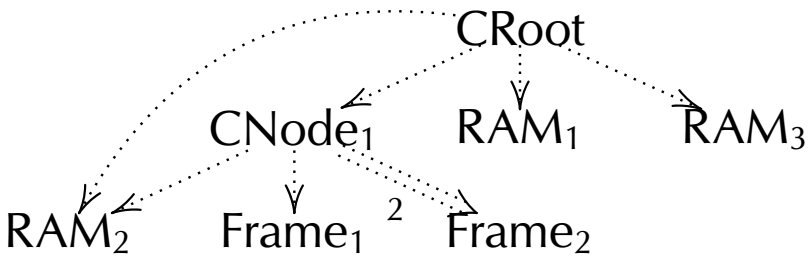


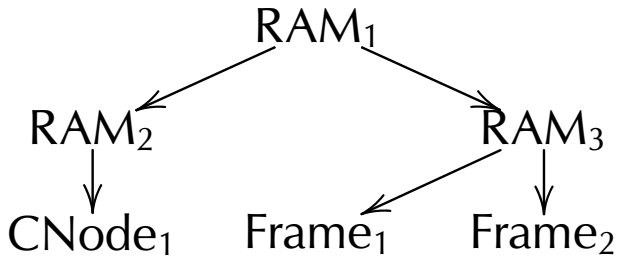


- Ancestry is a tree (forest).
- $\exists$Object $\rightarrow$ $\exists$Cap.
- Barrelfish is not identical.

Invariants

In seL4

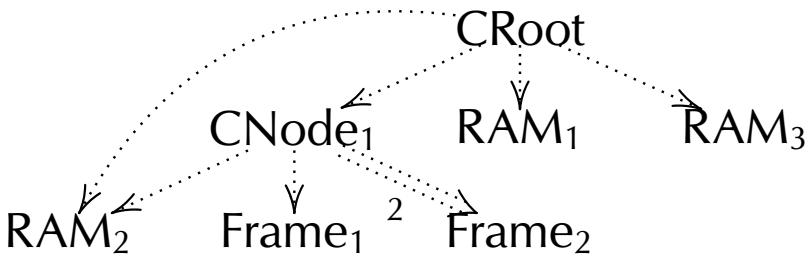


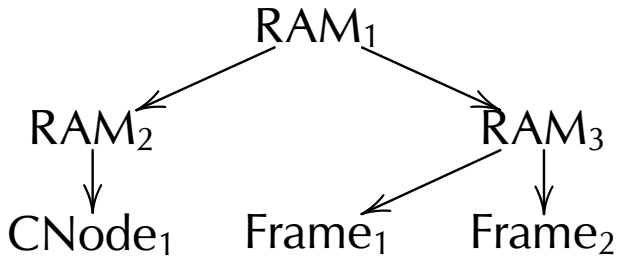


- Ancestry is a tree (forest).
- $\exists$Object $\rightarrow$ $\exists$Cap.
- Barrelfish is not identical.
  We're not sure *exactly* how yet.

Invariants

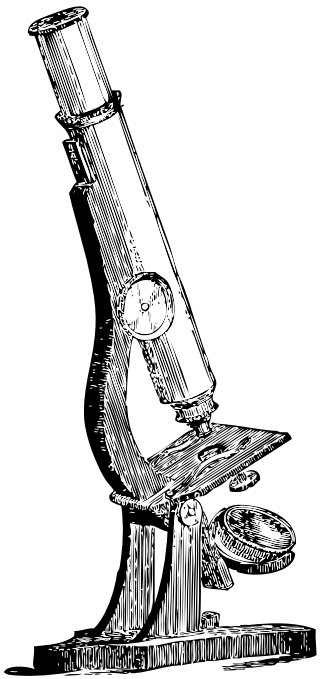
In seL4





- Ancestry is a tree (forest).
- $\exists$Object $\rightarrow$ $\exists$Cap.
- Barrelfish is not identical.
  We're not sure *exactly* how yet.
  We'd really like to.

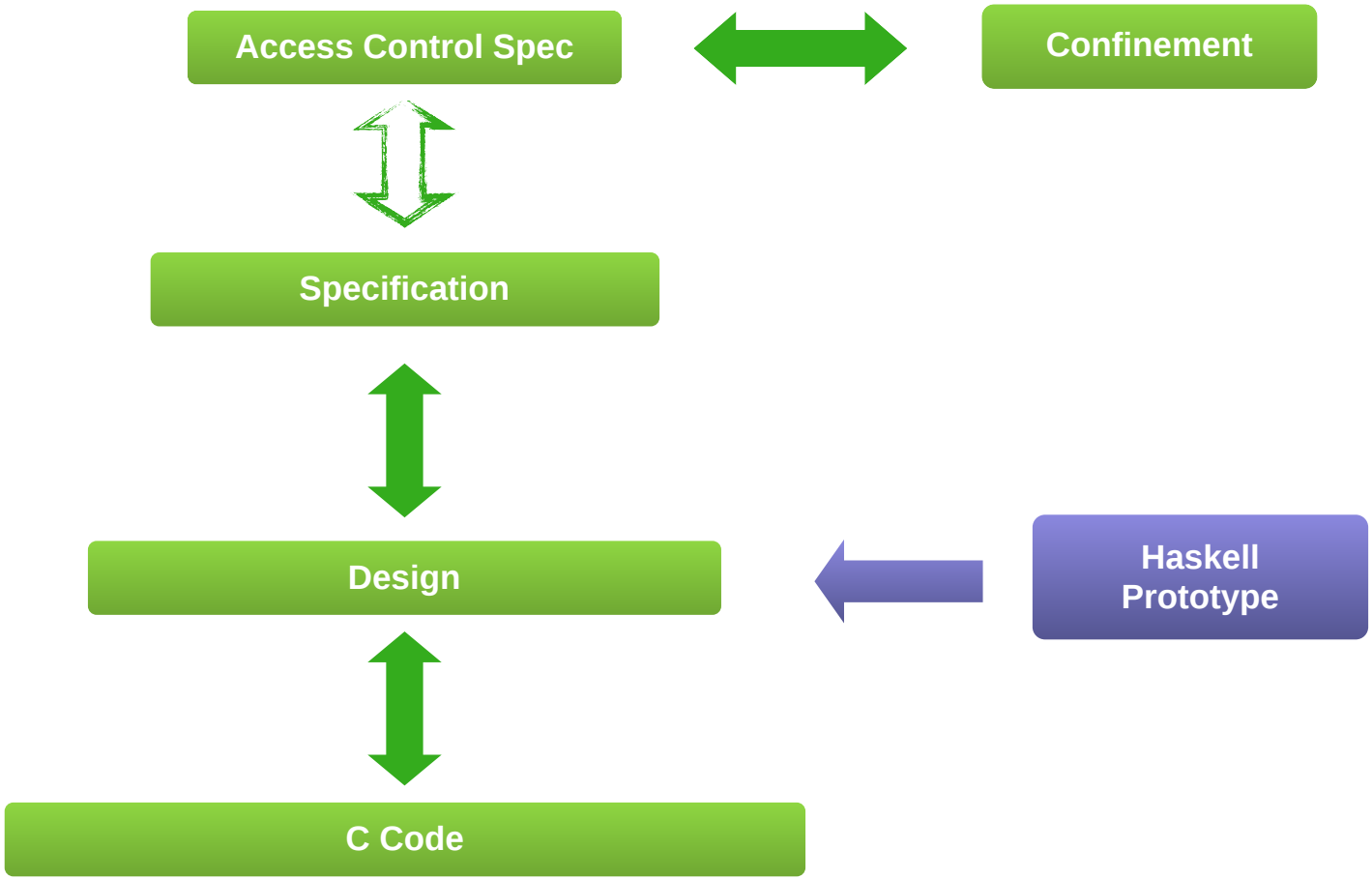
Results



We know quite a bit already (in the context of seL4).

- Implementation proof.
- Integrity proof.
- Confidentiality proof.
- Applications of user-level allocation.

The System is Correctly Implemented



The System is Correctly Implemented

The abstract spec is all that matters now!

Authority Confinement

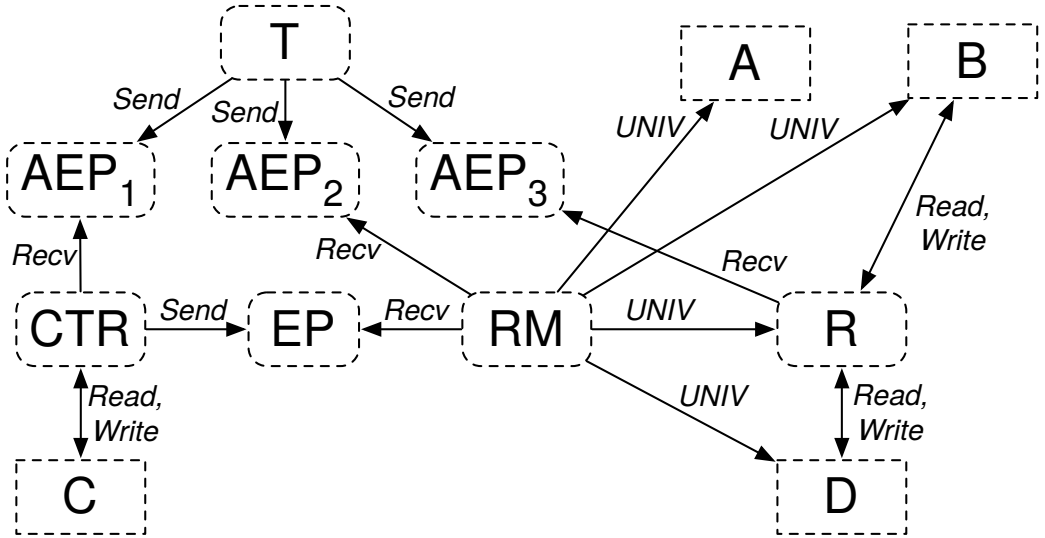


Figure: The Secure Access Controller

seL4 implements the take-grant model:

Authority Confinement

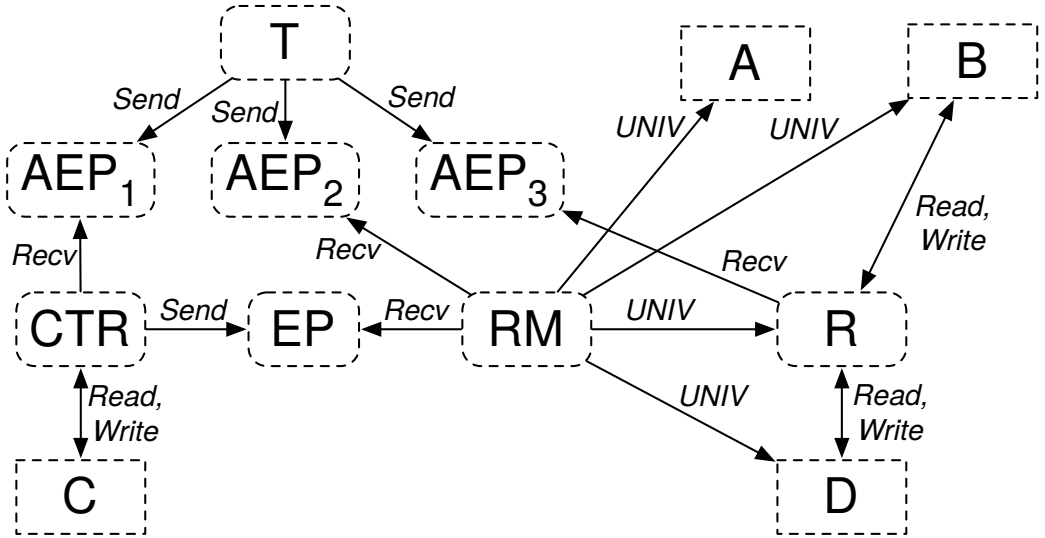


Figure: The Secure Access Controller

seL4 implements the take-grant model:

Confinement Authority (caps) only flows along edges.

Authority Confinement

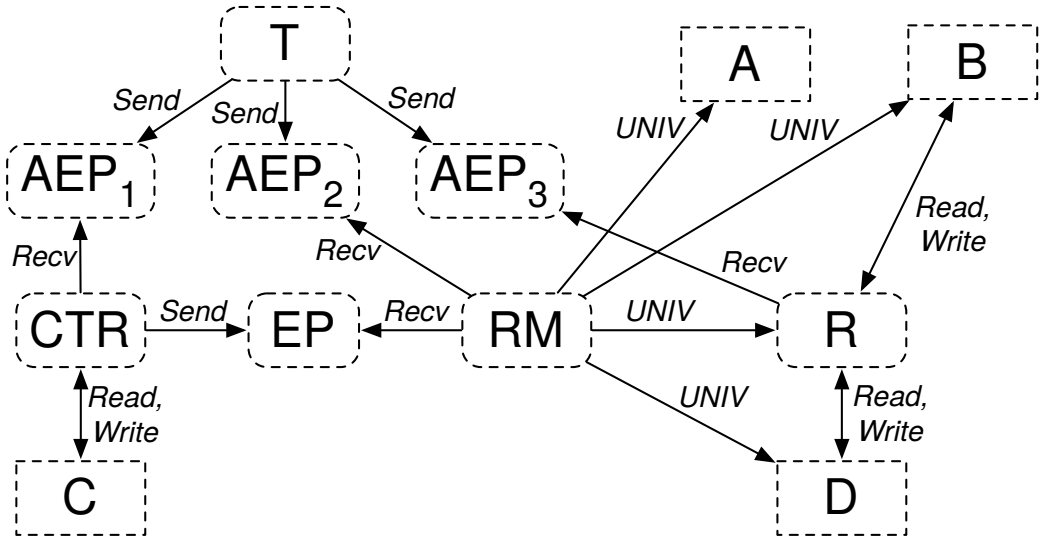


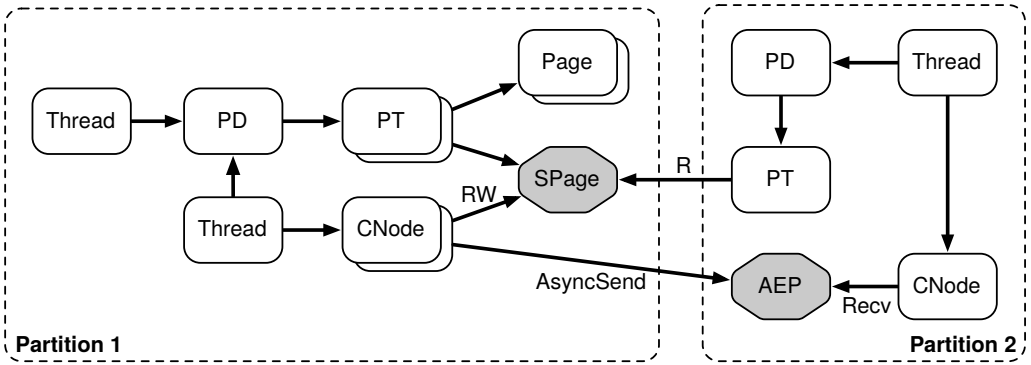
Figure: The Secure Access Controller

seL4 implements the take-grant model:

Confinement Authority (caps) only flows along edges.

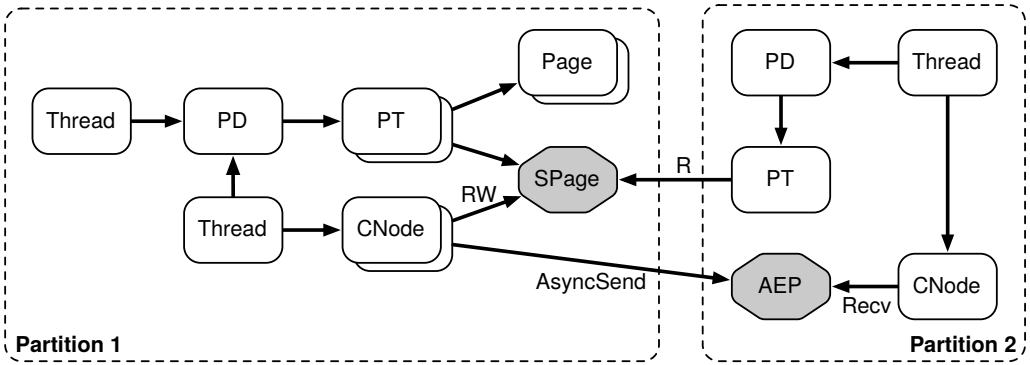
Integrity Objects only modified via (transient) authority.

Information Flow



seL4 enforces information flow policy:

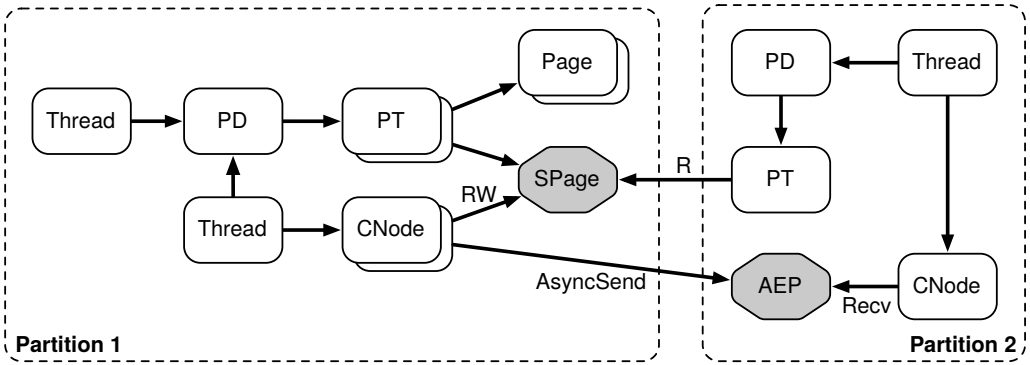
Information Flow



seL4 enforces information flow policy:

- Builds on integrity proof.

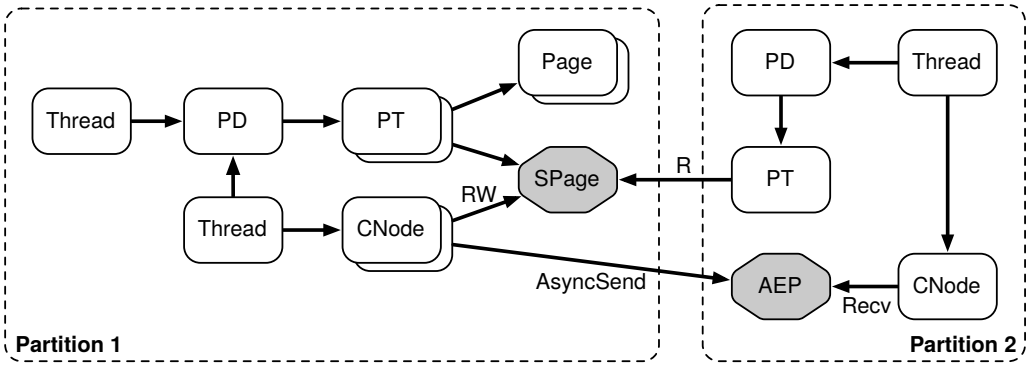
Information Flow



seL4 enforces information flow policy:

- Builds on integrity proof.
- No flow via kernel mechanisms e.g. scheduler.

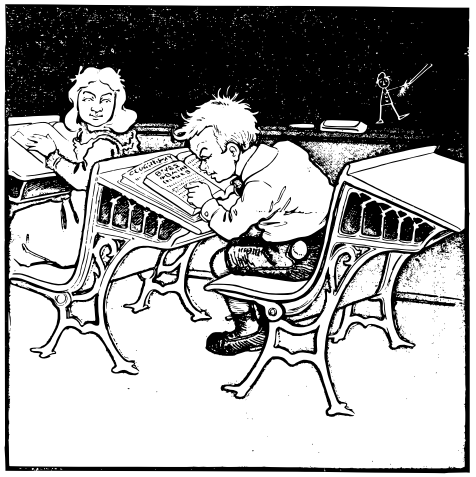
Information Flow



seL4 enforces information flow policy:

- Builds on integrity proof.
- No flow via kernel mechanisms e.g. scheduler.
- No IPC back channel (data diode).

Lessons

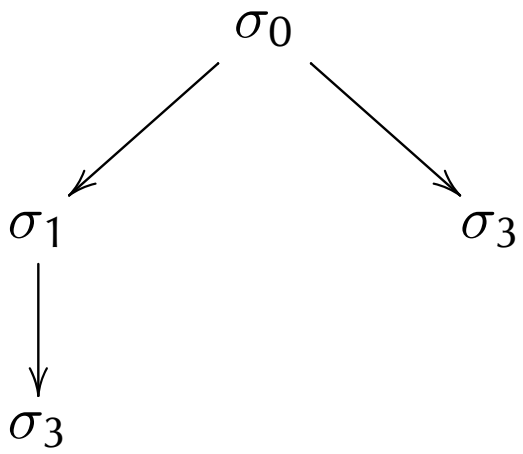


- Caps aren't slow.
- Strong security results are possible.
- Interposability has seldom been used.

Questions?

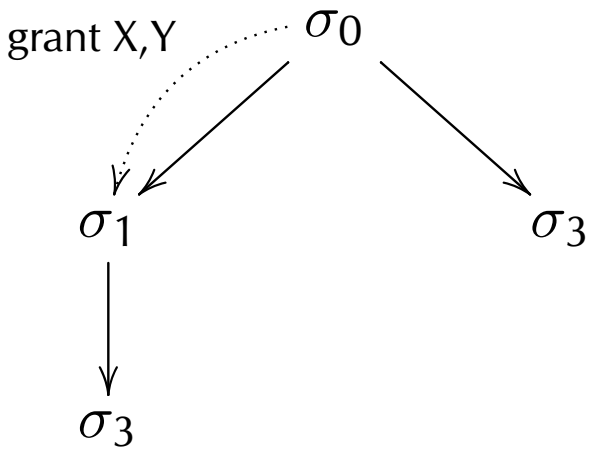
Address Spaces in L4

L4 used hierarchical virtual address spaces, and regions were *granted* to descendents.



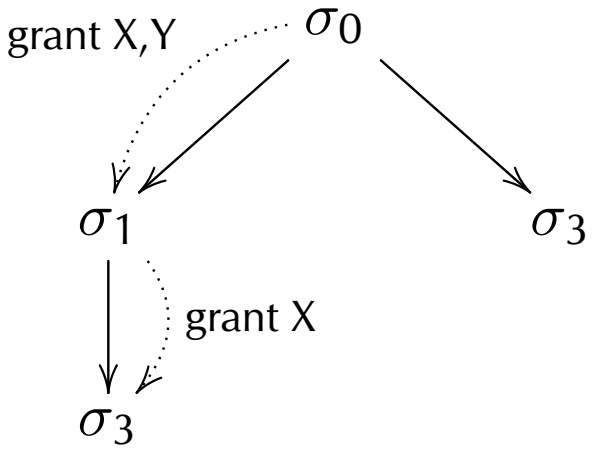
Address Spaces in L4

L4 used hierarchical virtual address spaces, and regions were *granted* to descendents.



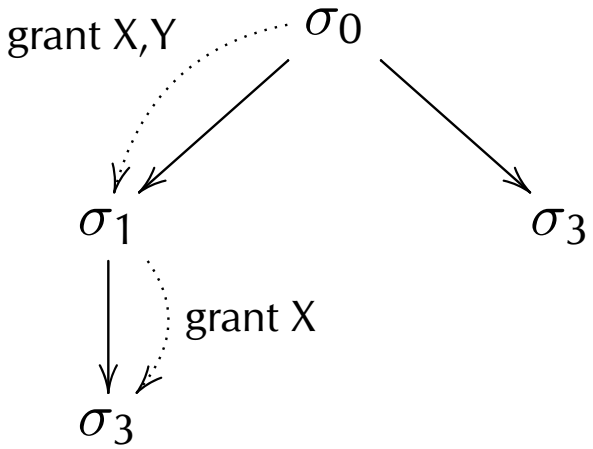
Address Spaces in L4

L4 used hierarchical virtual address spaces, and regions were *granted* to descendents.



Address Spaces in L4

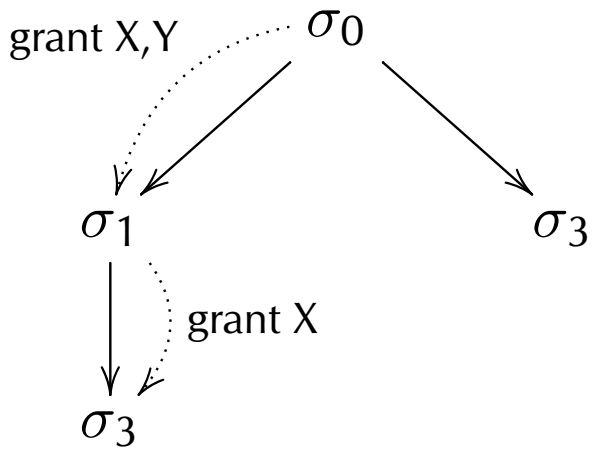
L4 used hierarchical virtual address spaces, and regions were *granted* to descendents.



\+ Allowed user paging & delegation.

Address Spaces in L4

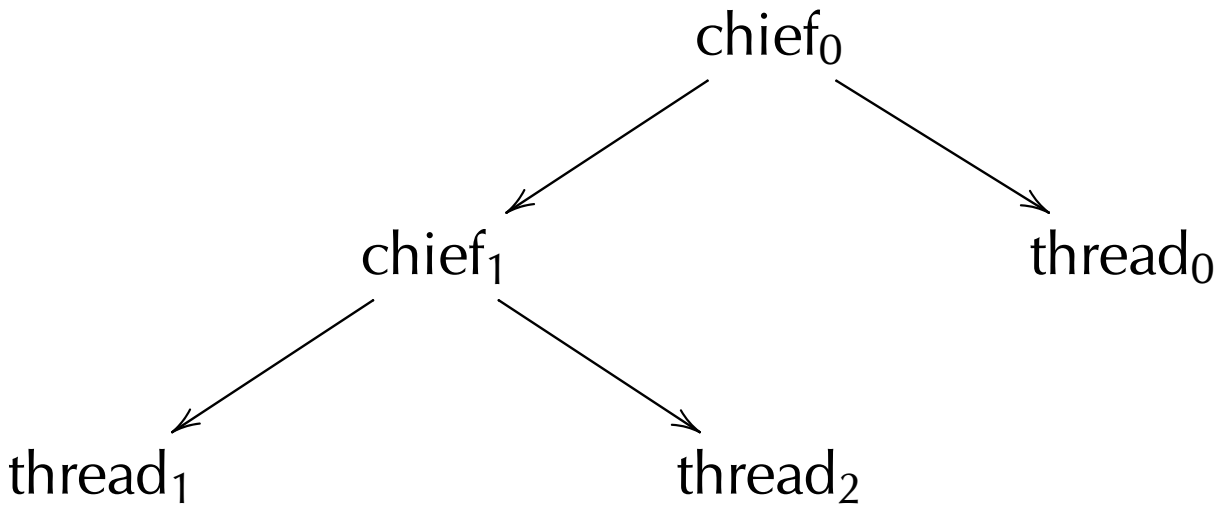
L4 used hierarchical virtual address spaces, and regions were *granted* to descendents.



+ Allowed user paging & delegation.
– Only exposed *virtual* addresses.
– Kernel memory not covered.

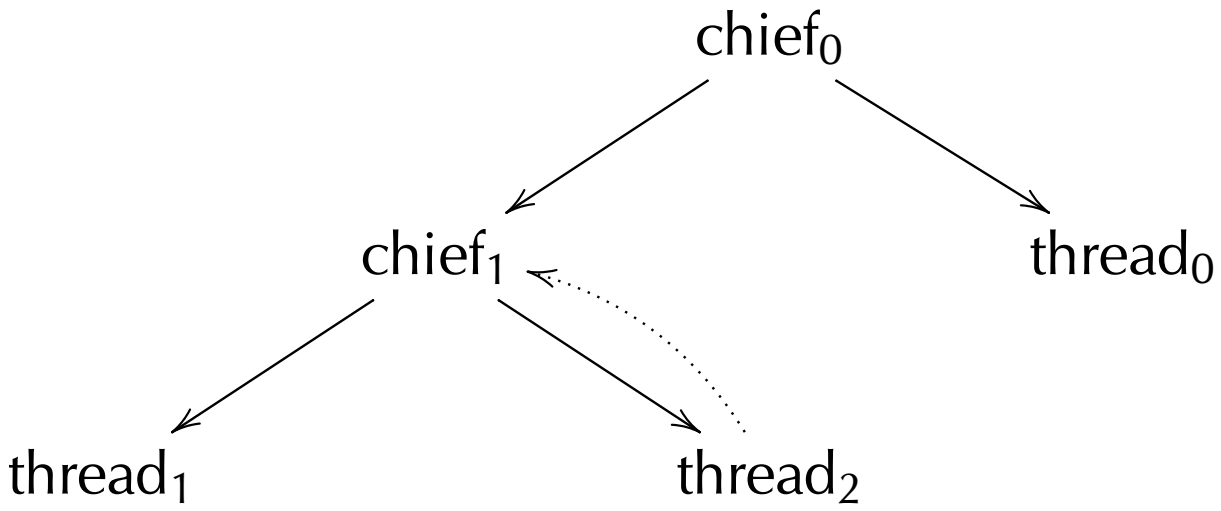
Clans and Chiefs

Threads belong to *clans*. Messages between clans go via *chiefs*.



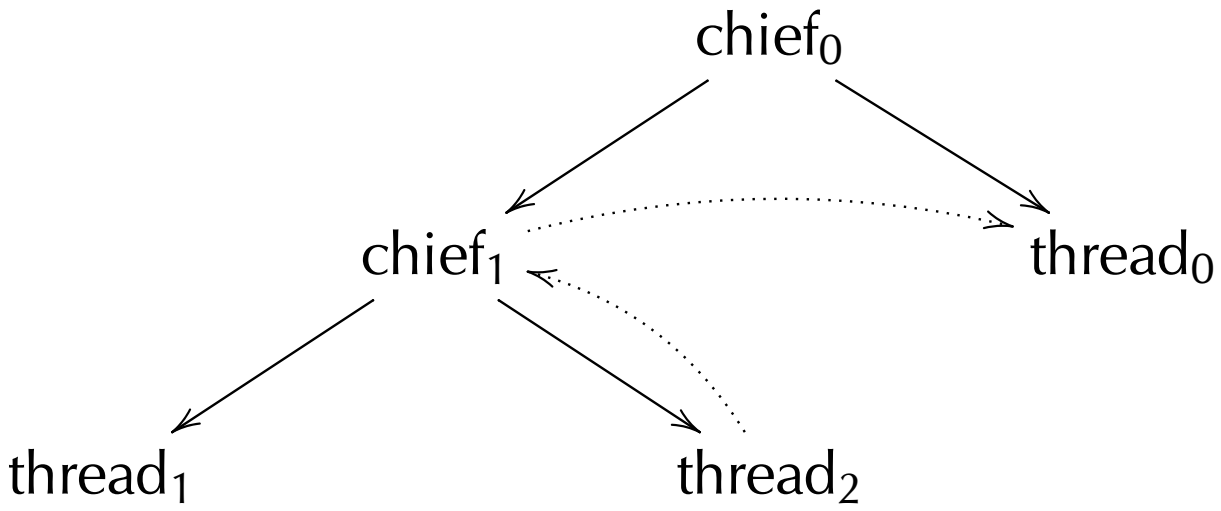
Clans and Chiefs

Threads belong to *clans*. Messages between clans go via *chiefs*.



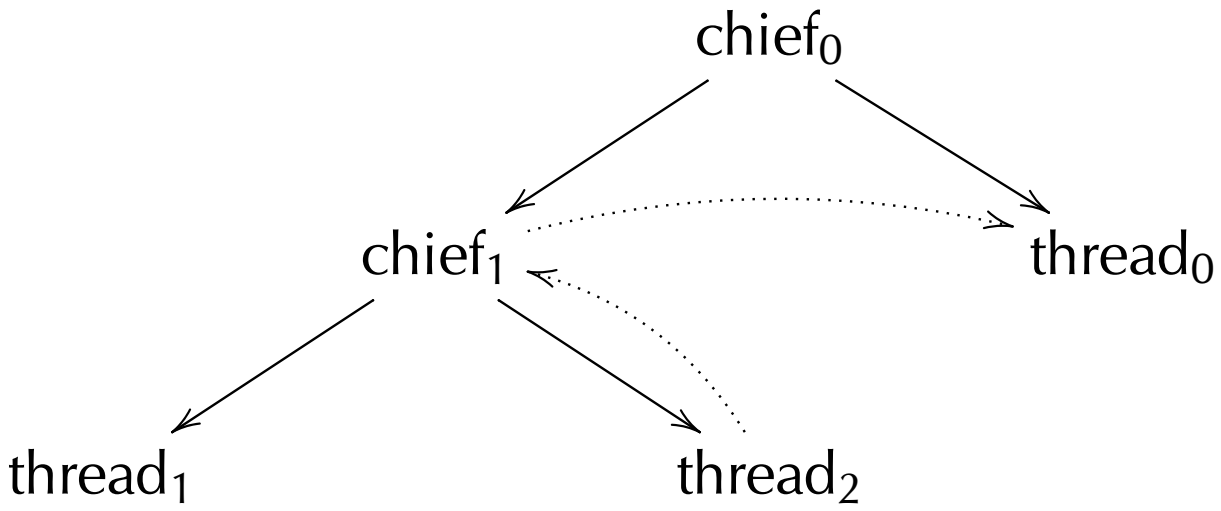
Clans and Chiefs

Threads belong to *clans*. Messages between clans go via *chiefs*.



Clans and Chiefs

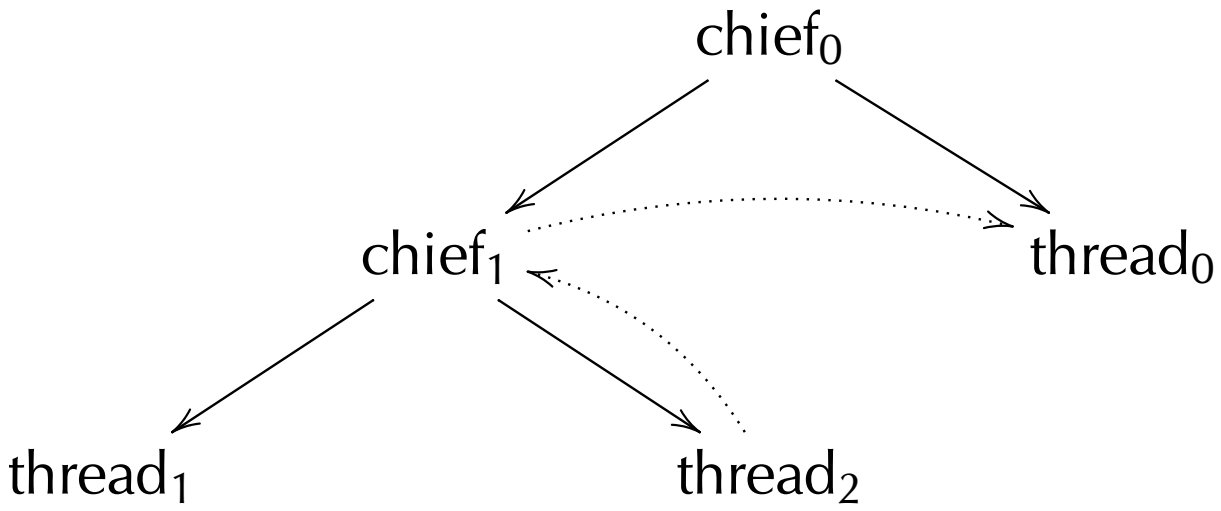
Threads belong to *clans*. Messages between clans go via *chiefs*.



+ Allows communication control.

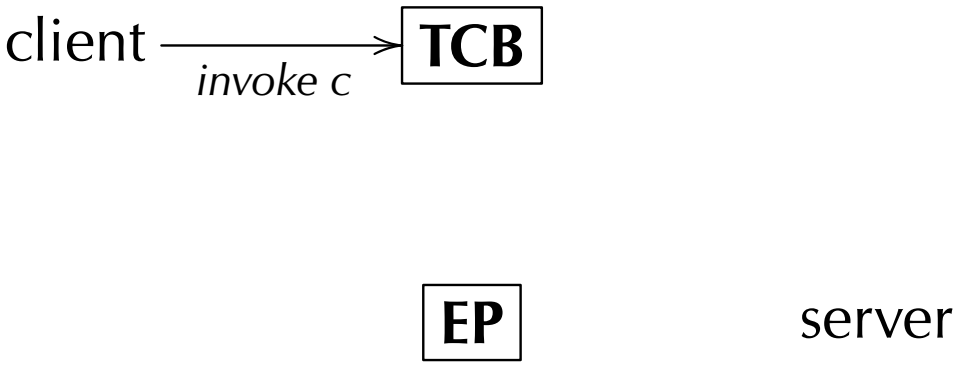
Clans and Chiefs

Threads belong to *clans*. Messages between clans go via *chiefs*.



+ Allows communication control.
– Static and inflexible.
– Introduces latency.
– Addresses still global.

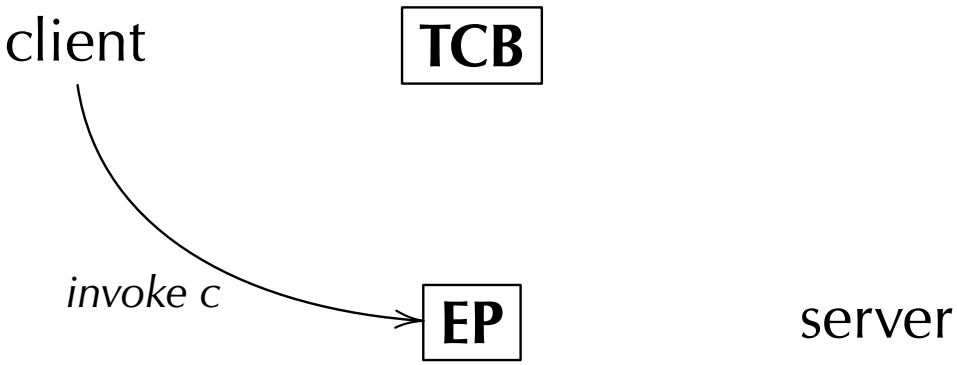
Interposability



Extend system w/o modifying kernel:

- Syscalls are *messages* to *objects*.
- Send messages by invoking *caps*.

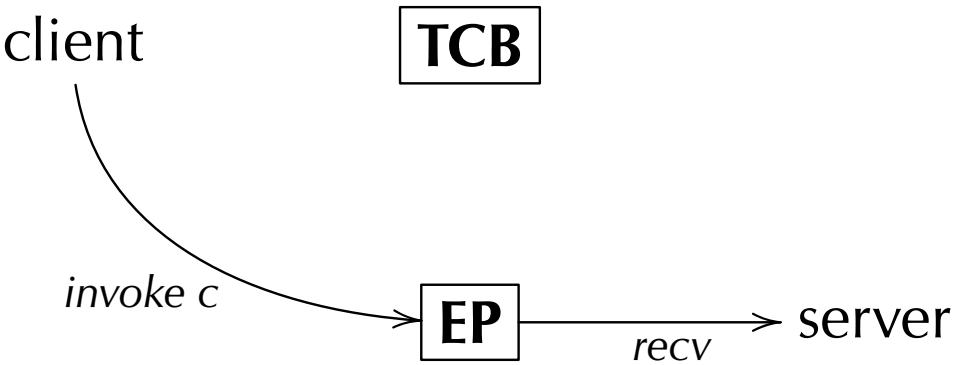
Interposability



Extend system w/o modifying kernel:

- Syscalls are *messages* to *objects*.
- Send messages by invoking *caps*.
- Transparently replace object cap with *endpoint* cap.

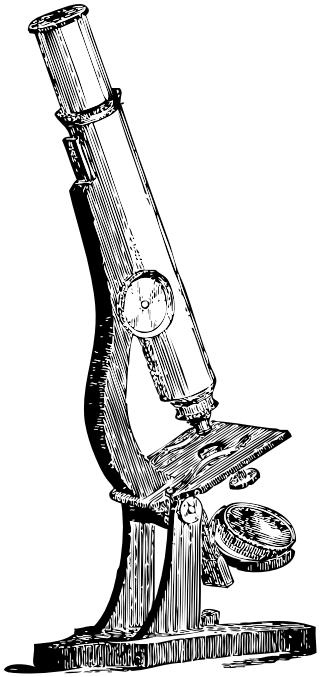
Interposability



Extend system w/o modifying kernel:

- Syscalls are *messages* to *objects*.
- Send messages by invoking *caps*.
- Transparently replace object cap with *endpoint* cap.
- Server implements object semantics.

Design for Verification



The cost of verification is high, so avoid kernel changes.

- Mechanisms as general as possible.
- Only one primitive to reason about: *cap invocation*.
- Amenable to analysis: take-grant model.
- Highly flexible resolution/sharing model: GPT.

Cache Colouring

Example of delegated allocation:

- Isolate subsystems in cache for performance or security.
- Requires control of physical allocation.
- Also partitions kernel memory, with no kernel changes!