Advanced Operating Systems
Summer Semester 2024/2025

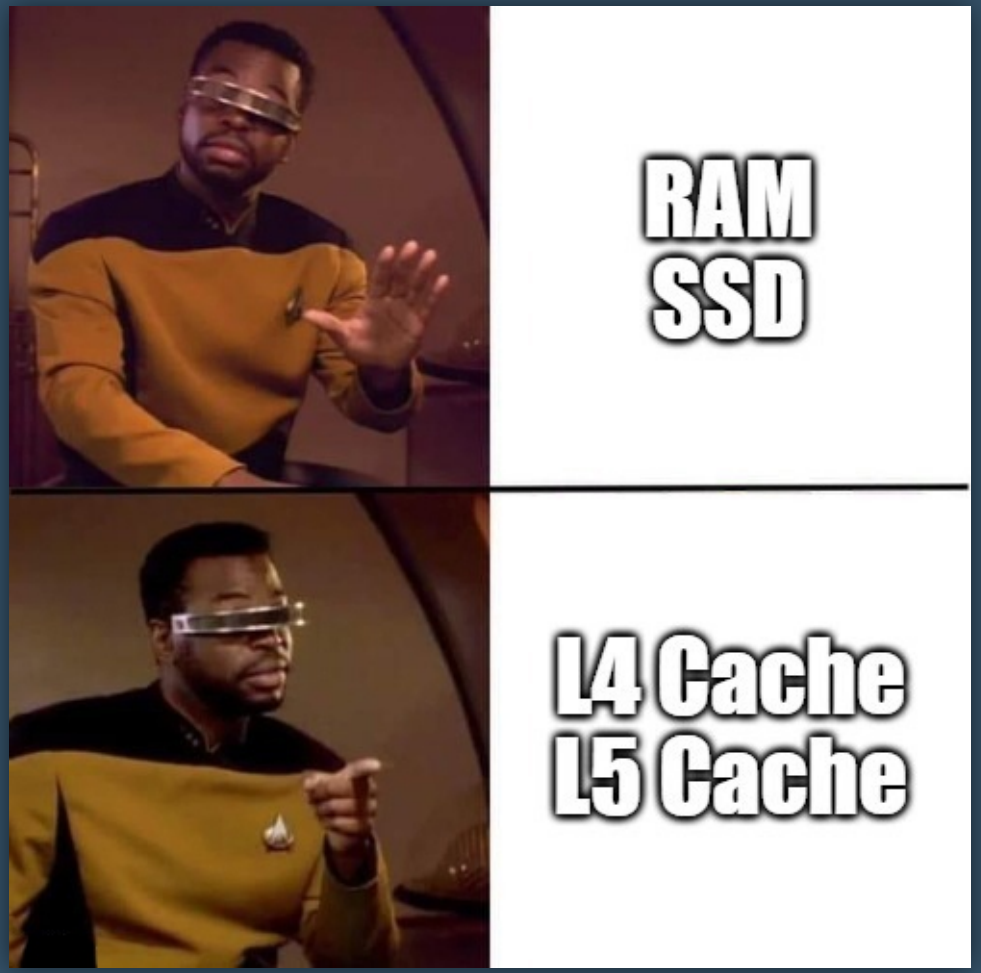
Martin Děcký

3

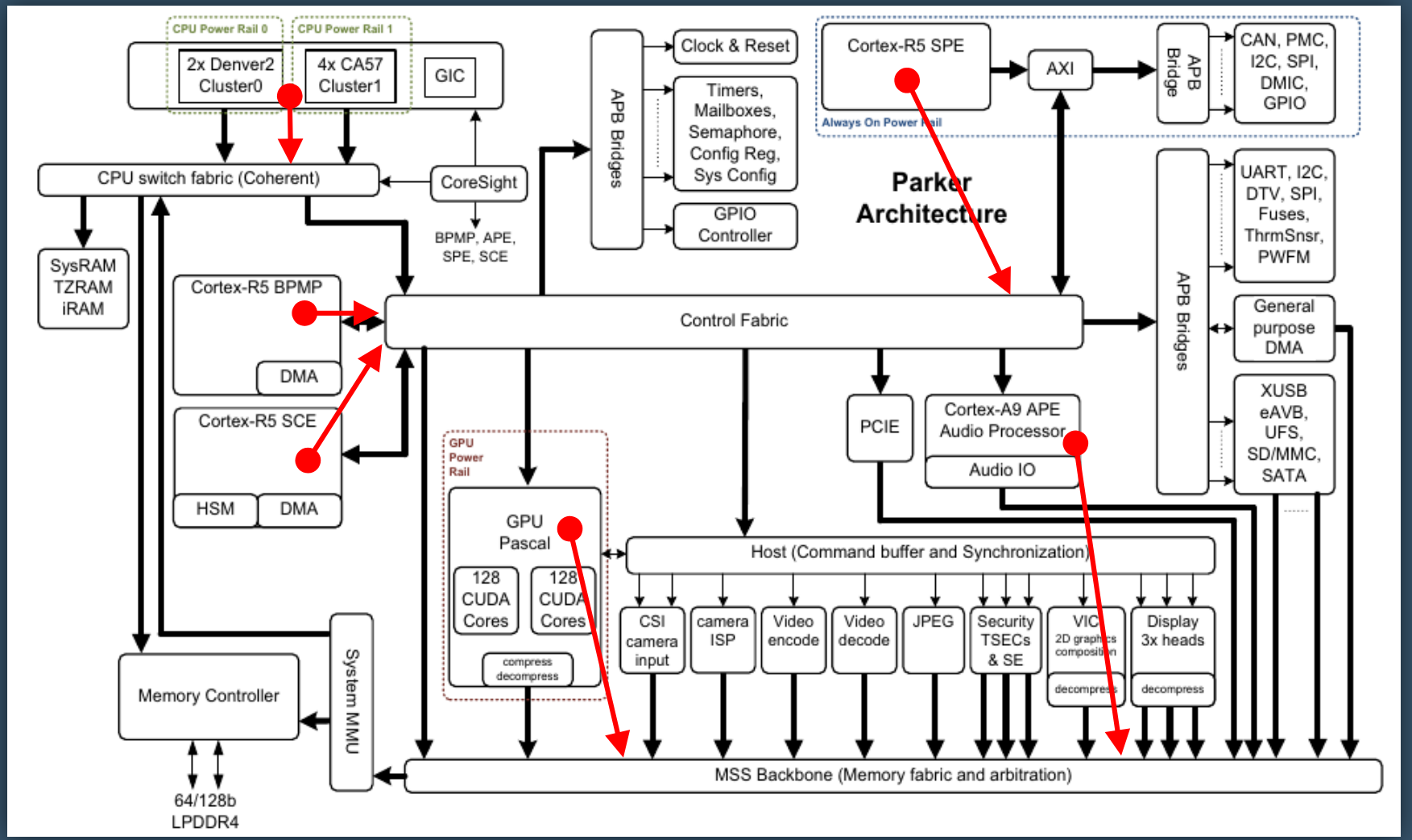
Memory and Resource Management

Physical Memory Myths

- **Random access performance seems to be O(1) in time units**
  - In reality it is closer to $O(\sqrt{n})$
    - Where *n* is the size of the working set
    - Performance effects of the cache hierarchy
- **There is a canonical physical address space**
  - Different views of the physical address space
    - Local APIC and SMM on x86, secure/non-secure TrustZone on ARM
    - Embedding of the I/O address space into the MMIO address space on x86
  - Completely disjoint address spaces
    - No central interconnect, but a network of nodes and address translations



Source: imgflip.com



Source: Roscoe T.: *It's Time for Operating Systems to Rediscover Hardware*, Joint Keynote Address at USENIX ATC '21 / OSDI '21, https://people.inf.ethz.ch/troscoe/pubs/2021-07-16-OSDIKeyNote-Handout.pdf

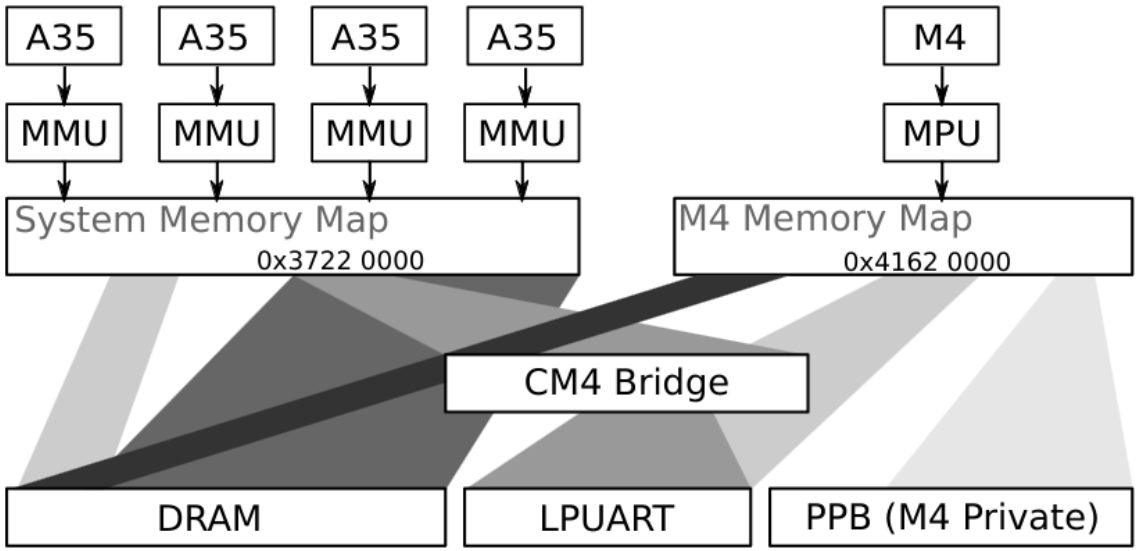
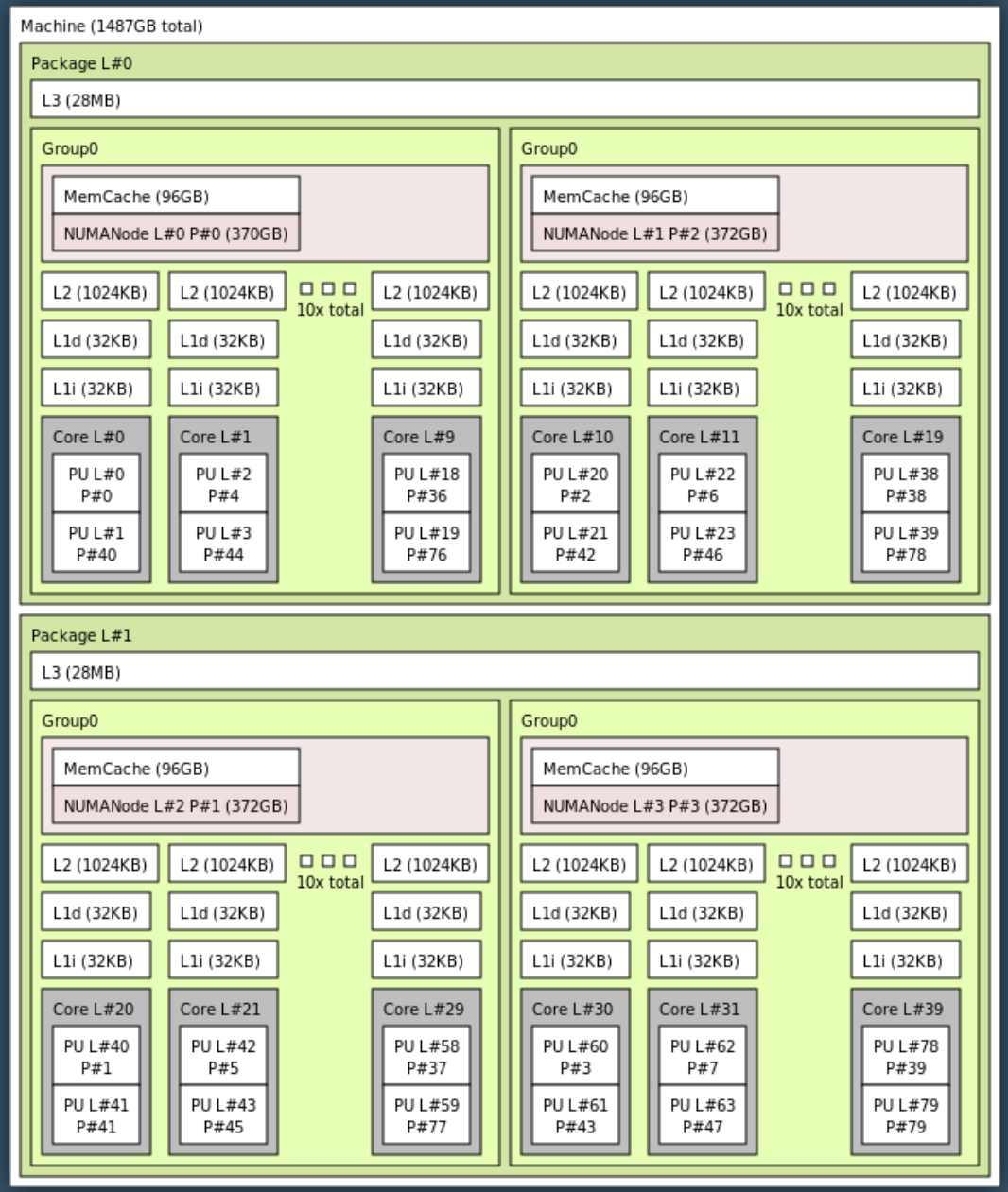


Figure 2. Subset of the NXP i.MX8 memory layout. The LPUART is accessed using a different addresses from the M4 cores, the >2GiB memory is only accessible from the A35 cores, the PPB is only reachable from the M4 cores.

Source: Achermann R., Cock D., Haecki R., Hossle N., Humbel L., Roscoe T., Schwyn D.:
Generating Correct Initial Page Tables from Formal Hardware Descriptions,
In the Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS),
ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP), 2021,
https://retoachermann.ch/static/papers/achermann-2021-gcip.pdf

Non-Uniform Memory Access (NUMA)

- **Explicitly exposed hardware topology**
  - Processing units, cores, packages
  - NUMA nodes (directly byte-addressable memory)
  - Caches
    - Transparent cache coherency (ccNUMA)
      - MSI, MESI, MESIF, MOSI, MOESI, Dragon, Firefly protocols
      - Directory-based cache coherency
  - Buses and I/O devices
- **Guiding heuristics for placing execution near its working set**
  - `numactl`, `libnuma`


Machine (1487GB total)
Package L#0
L3 (28MB)
Group0
MemCache (96GB)
NUMANode L#0 P#0 (370GB)
L2 (1024KB)
L2 (1024KB)
10x total
L2 (1024KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#0
PU L#0 P#0
PU L#1 P#40
Core L#1
PU L#2 P#4
PU L#3 P#44
Core L#9
PU L#18 P#36
PU L#19 P#76
Group0
MemCache (96GB)
NUMANode L#1 P#2 (372GB)
L2 (1024KB)
L2 (1024KB)
10x total
L2 (1024KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#10
PU L#20 P#2
PU L#21 P#42
Core L#11
PU L#22 P#6
PU L#23 P#46
Core L#19
PU L#38 P#38
PU L#39 P#78
Package L#1
L3 (28MB)
Group0
MemCache (96GB)
NUMANode L#2 P#1 (372GB)
L2 (1024KB)
L2 (1024KB)
10x total
L2 (1024KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#20
PU L#40 P#1
PU L#41 P#41
Core L#21
PU L#42 P#5
PU L#43 P#45
Core L#29
PU L#58 P#37
PU L#59 P#77
Group0
MemCache (96GB)
NUMANode L#3 P#3 (372GB)
L2 (1024KB)
L2 (1024KB)
10x total
L2 (1024KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#30
PU L#60 P#3
PU L#61 P#43
Core L#31
PU L#62 P#7
PU L#63 P#47
Core L#39
PU L#78 P#39
PU L#79 P#79

Device Virtual Memory

- **Mapping of device-visible addresses to bus-visible addresses**
  - Similar purpose to software virtual memory
    - Isolation (i.e. safety, security)
    - Mitigating fragmentation (i.e. scatter-gather functionality)
    - Mitigating address range issues
  - Integrated in the device DMA engine
    - Graphics Address/Aperture Remapping Table
  - Separate IOMMU
    - Device memory paging
    - Usually also implementing interrupt remapping

IOMMU

- **AMD-Vi, ARM SMMU**
- **Intel VT-d**
  - Usually located in the peripheral interconnect (a.k.a. north bridge)
  - Address space is usually associated with a protection domain
    - Endpoint is usually associated with a source ID
    - Data structure that maps source IDs to protection domains
    - Memory mapping using hierarchical page tables
      - First-stage translation page tables essentially equivalent to the CPU page tables
      - Second-stage translation for hypervisor, with nested first & second-stage translation
  - Device TLB for translation caching, other caches
  - ACPI DMAR (DMA Remapping Reporting) table

Physical Memory Management

- **Zones**
  - Continuous address ranges with specific properties
    - Available, reserved, firmware, kernel code/data, etc.
    - Logical properties
      - E.g. < 1 MiB, < 16 MiB, < 4 GiB on x86
  - Allocations
    - Tracking of used frames and their owner
    - Bitmaps, free lists, buddy allocation, etc.

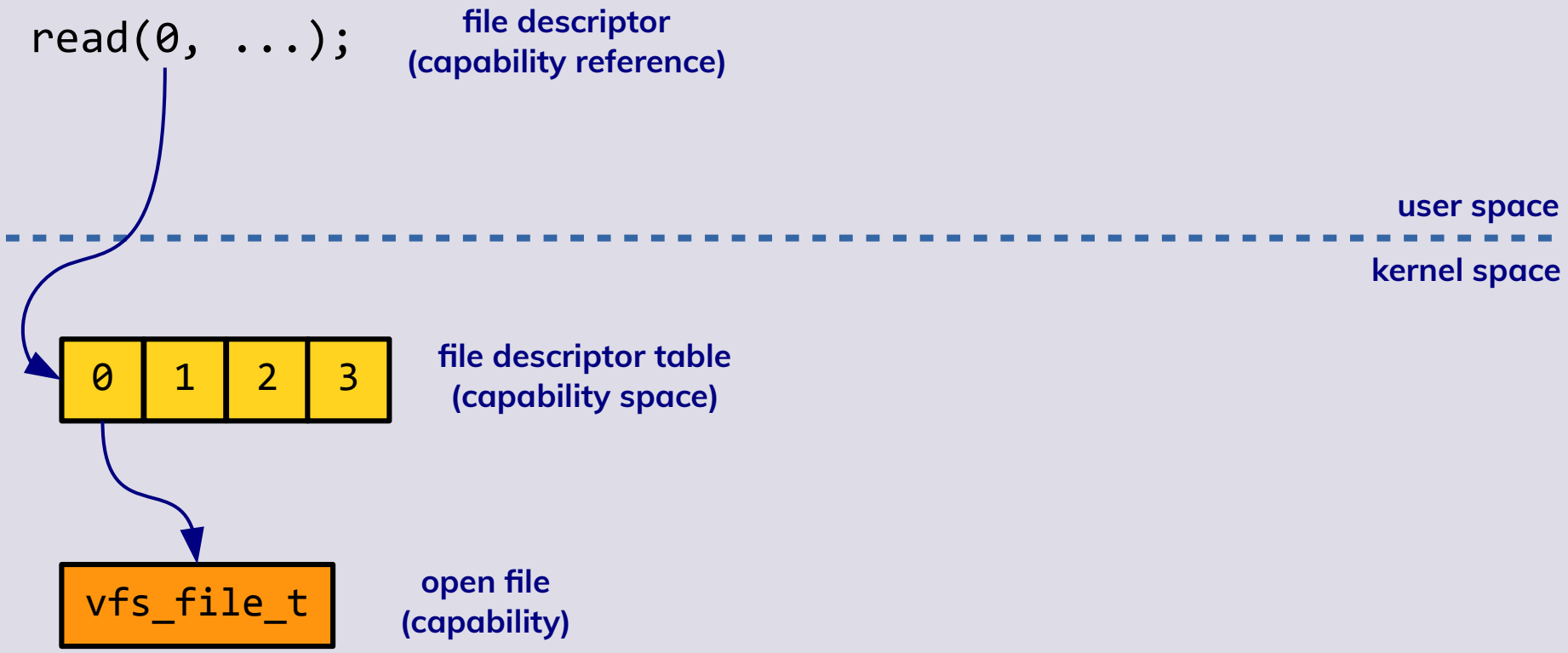
Capabilities

- **Motivation**
  - Universal and **pure** kernel mechanism for resource management
    - No specific management policy in the kernel
    - Policy decision delegated to user space
    - Delegation (granting) of authority over resources from the original owner to other parties
      - Including granting revocation

Capabilities Terminology

- **Capability**
  - Object instance representing (identifying) a specific resource
  - Kernel object representing a kernel-managed resource
  - Kernel proxy object identifying a user-managed resource
  - User space object representing a user space resource
- **Capability reference**
  - Unforgeable identifier (handle) to a capability
    - Possibility to restrict permissions (e.g. permissible operations) and identify ownership
- **Capability space**
  - Address space of capability references
    - Typically associated with a task
  - Capabilities as local identifiers within their namespace

Capabilities Put Simply



Capability Operations

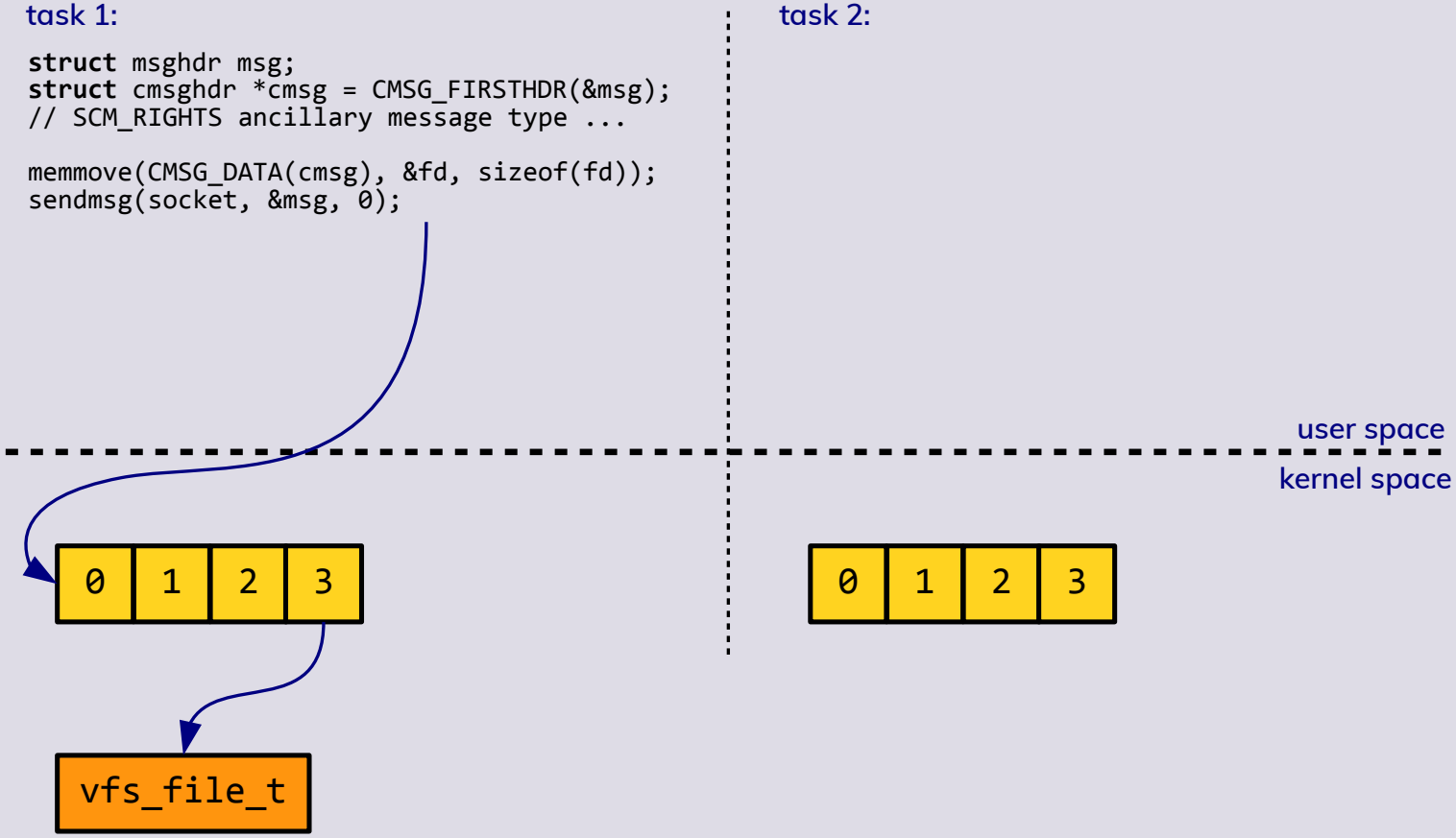
- **Invoke**
  - Execute a “business logic” method on the target object
- **Clone / Mint**
  - Create a duplicate capability reference (possibly with restricted permissions)
  - Multiple capability references can point to the same capability, but with different permissions
- **Delegate / Grant**
  - Pass a duplicate capability reference (possibly with restricted permissions) to a different capability space
  - In case of granting, the original ownership is kept
  - Only once or recursively
- **Revoke**
  - Forcefully removing and granted capability reference from other capability spaces

Capability Delegation

task 1:

```
struct msghdr msg;
struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
// SCM_RIGHTS ancillary message type ...

memmove(CMSG_DATA(cmsg), &fd, sizeof(fd));
sendmsg(socket, &msg, 0);
```

task 2:



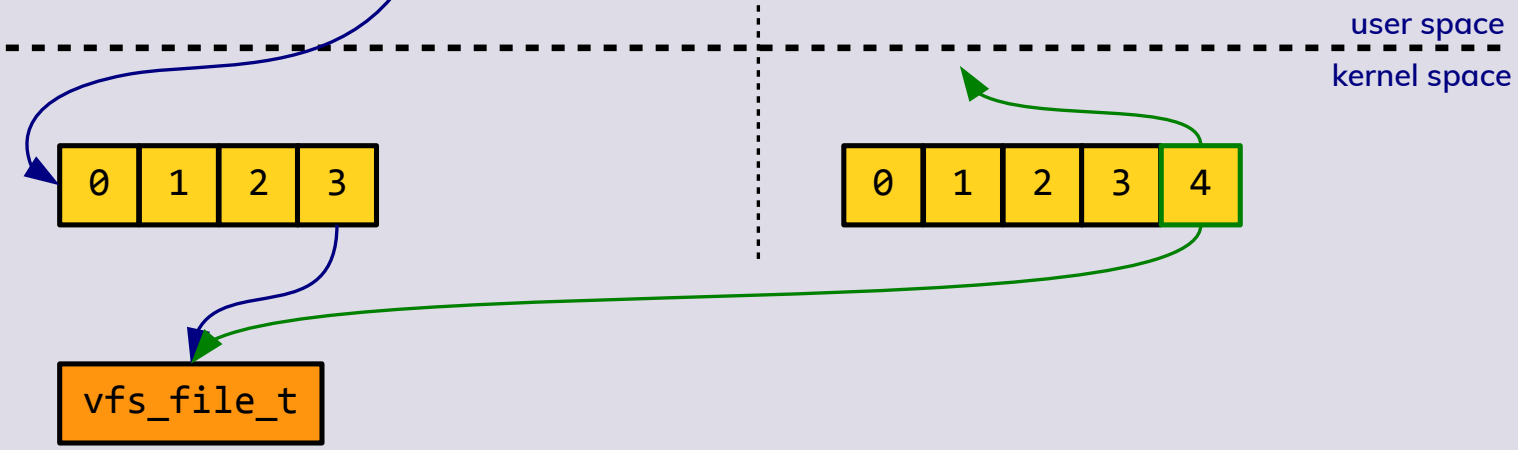
Capability Delegation

task 1:

```
struct msghdr msg;
struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
// SCM_RIGHTS ancillary message type ...

memmove(CMSG_DATA(cmsg), &fd, sizeof(fd));
sendmsg(socket, &msg, 0);
```

task 2:

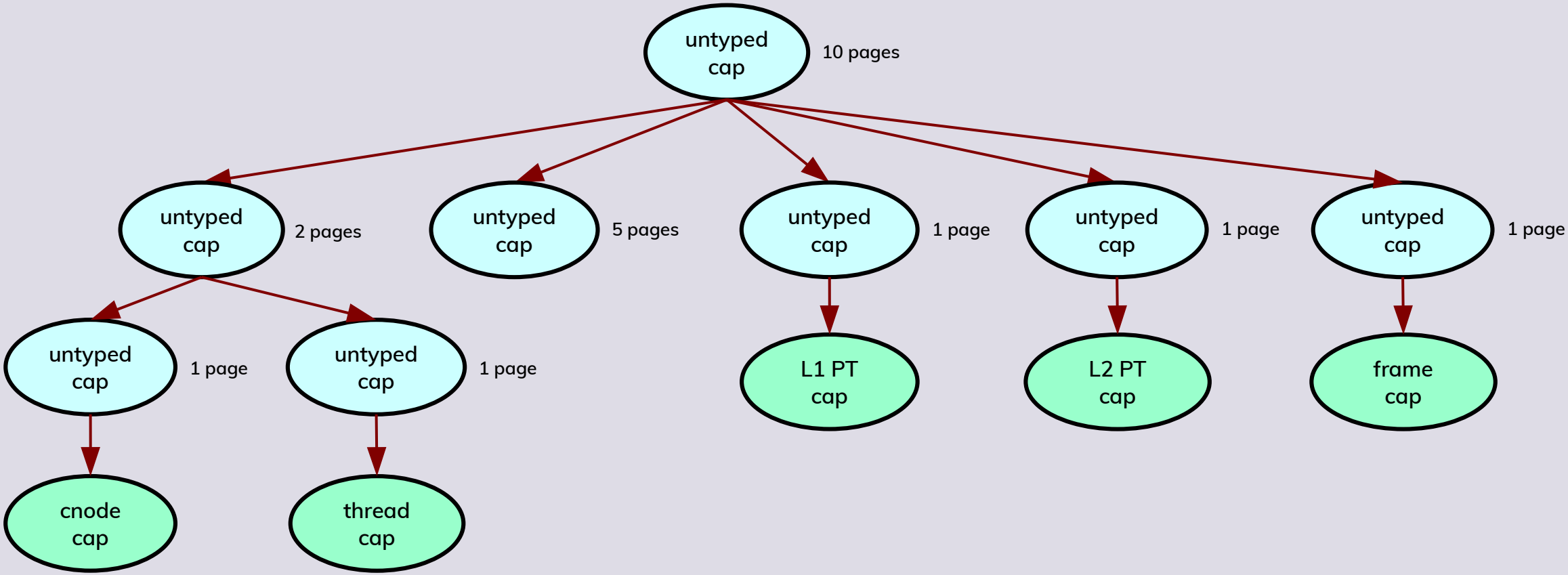
```
struct msghdr msg;
struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
// ...

recvmsg(socket, &msg, 0);

int fd;
memmove(&fd, CMSG_DATA(cmsg), sizeof(fd));
```



Physical Memory Management

- **Representing physical memory as capabilities**
  - Chicken & egg problem: Capabilities, capability spaces, page tables and other bookkeeping structures require memory for storage (i.e. capabilities)
  - Recursive solution: Type hierarchy of capabilities
    - *Untyped memory* capability type
      - Representing a range of physical memory
      - Initially a single capability representing the entire physical memory
      - Untyped capabilities be **derived** …
        - … into multiple untyped capabilities (recursively splitting the physical memory)
        - … into capabilities of other types
          - Providing the memory for capability storage and bookkeeping
          - Providing memory for other kernel objects

Capability Derivation Tree



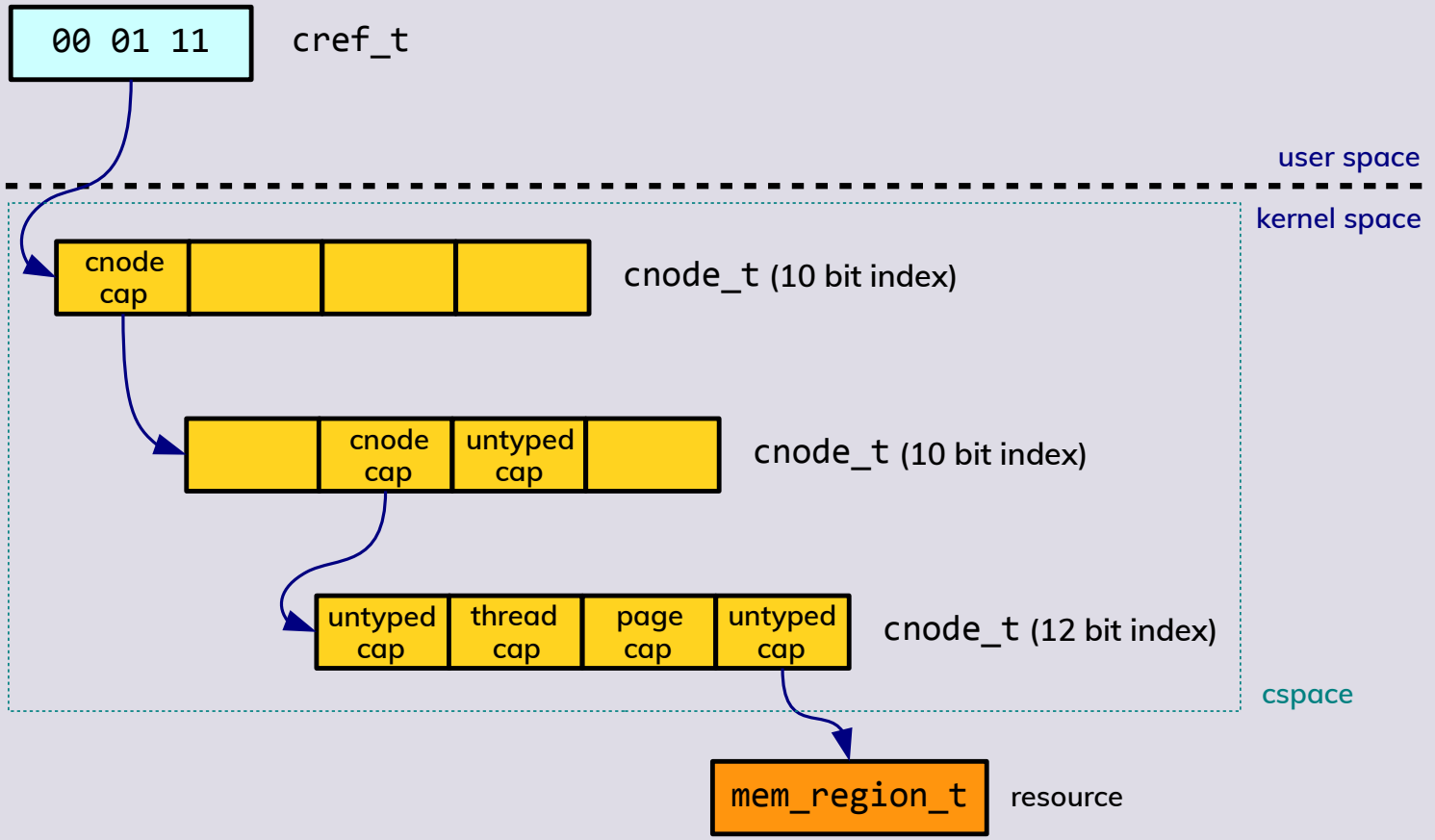
Capability References and Spaces

- **Naked capabilities**
  - Capability references identify capabilities directly
    - E.g. physical memory addresses identifying untyped memory capabilities
- **Encapsulated capabilities**
  - Capability references need to be mapped to capabilities
  - Mapping database of capability space
    - Fast lookup of capability references (most frequent operation)
    - Reasonably fast creation / removal of capability references
    - Low memory overhead and fragmentation (sparse capability space)
    - Additional metadata (permissions, delegation, granting)
    - Possibility for in-line storage of actual kernel objects (up to a certain size)

Capability References and Spaces

- **Capability space (cspace)**
  - Directed graph of capability nodes
    - Can be implicit (no explicit object representation)
- **Capability node (cnode)**
  - Array of capability slots
    - Empty slot
    - Slot pointing to a specific capability
    - Slot pointing to a cnode
      - Hierarchical organization of capability nodes
      - Radix tree indexing

Hierarchical Capability Mapping Database



Capabilities Example: seL4

- **Kernel objects**
  - `UntypedObject` (physical memory range)
  - `TCBObject` (thread)
  - `EndpointObject` (IPC calls destination)
  - `AsyncEndpointObject` (signal recipient)
  - `CapTableObject` (array of capabilities)
  - `X86_4K` (4 KiB frame)
  - `X86_4M` (4 MiB frame)
  - `X86_PageTableObject` (2nd level page table)
  - `X86_PageDirectoryObject` (1st level page table)

Capabilities Example: seL4

- **Initial thread cnode content (4096 slots)**
  - TCB
  - cnode
  - vspace (1st level page table)
  - Global IRQ controller
  - Global ASID controller
  - ASID pool
  - Global I/O port capability
  - Global I/O space capability
  - BootInfo frame
  - IPC buffer
  - Security domain capability
  - Untyped capabilities

Capabilities Example: seL4

- **Page mapping**

```
// Create a new 4 KiB frame object
seL4_Untyped_Retype(untypedCap0, seL4_X86_4K, 12, cnodeCap, cspaceIndex0, cspaceIndexDepth0,
     cspaceSlot0, 1);

// Create a new page table (2nd level) object
seL4_Untyped_Retype(untypedCap1, seL4_X86_PageTableObject, 12, cnodeCap, cspaceIndex1,
     cspaceIndexDepth1, cspaceSlot1, 1);

// Link the page table from the page directory (i.e. 1st level) object (i.e. Vspace)
seL4_X86_PageTable_Map(cspaceSlot1, vspaceCap, virtAddr & ~0x003FFFFF, seL4_X86_Default_VMAttributes);

// Map the frame to virtAddr in VSpace
seL4_X86_Page_Map(cspaceSlot0, vspaceCap, virtAddr, seL4_AllRights, seL4_X86_Default_VMAttributes);
```

Capabilities Example: seL4

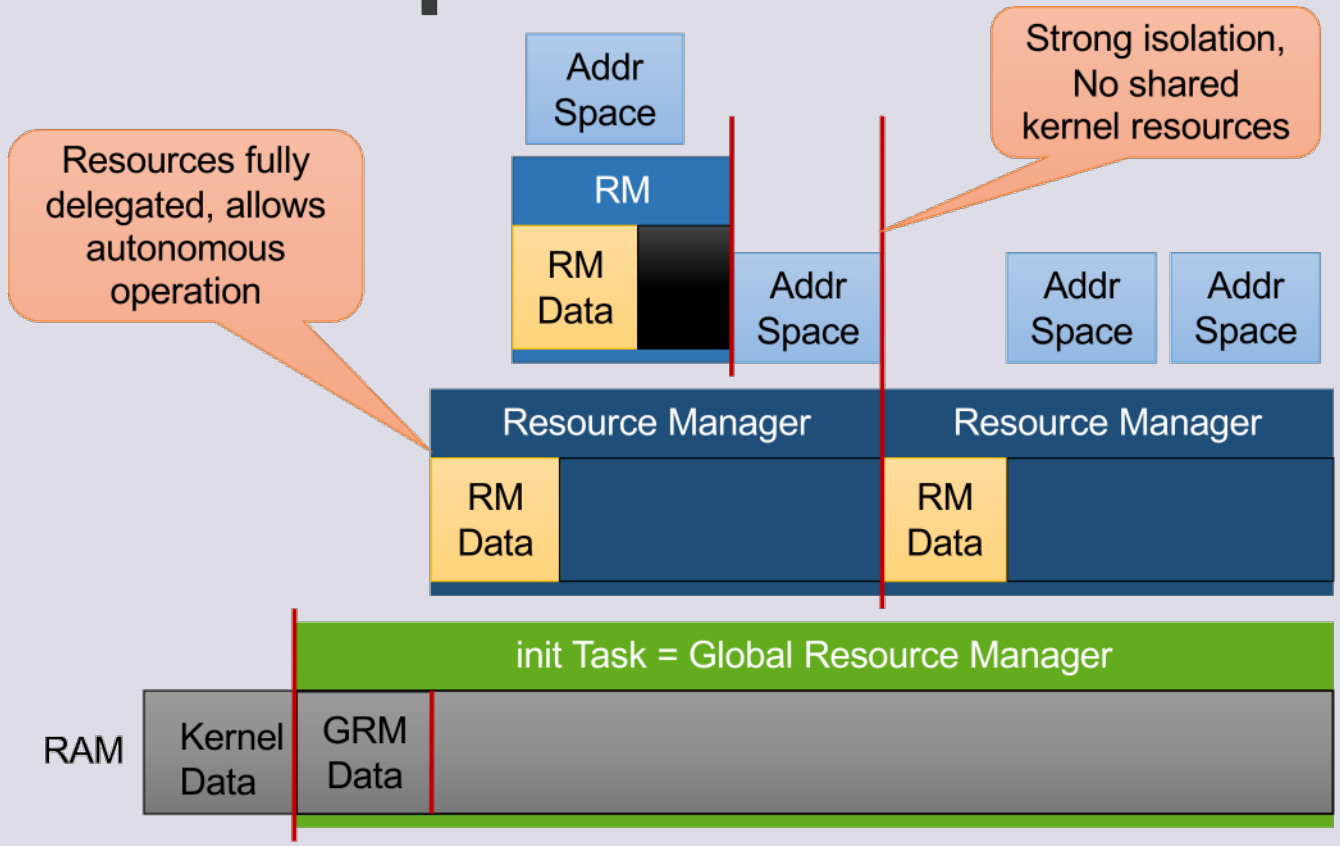
- **Capability node allocation**

```
// Create a new cnode with 256 slots
seL4_Untyped_Retype(untypedCap, seL4_CapTableObject, 8, cnodeCap, cspaceIndex, cspaceIndexDepth,
    cspaceSlot, 1);
```

- **Untyped capability splitting**

```
// Split a 64 KiB region into 8 times 8 KiB regions
seL4_Untyped_Retype(untypedCap, seL4_UntypedObject, 13, cnodeCap, cspaceIndex, cspaceIndexDepth,
    firstCspaceSlot, 8);
```

Capabilities Example: seL4



Source: Heiser G.: *Introduction: Using seL4*
Courtesy of Gernot Heiser, UNSW Sydney, CC BY 4.0,
http://www.cse.unsw.edu.au/~cs9242/22/lectures/01b-sel4.pdf

Physical Memory Management Comparison

- **Traditional**
  - Straightforward API
  - High-level abstraction
  - Portable
  - Implicit policy
  - Accounting out of scope
  - Delegation out of scope
- **Capability-based**
  - No implicit policy (policy set completely by the client)
  - Accounting and delegation within the scope
  - Low-level API
  - Potential abstraction inversion
  - Non-portable

Note on Physical Memory Accounting

- **Strict memory reservation**
  - Sum of virtual memory sizes < Sum of physical memory sizes
    - Swap space counted as physical memory
  - In-bound out-of-memory condition
  - More predictable
  - Potential inefficient resource usage
- **Memory overcommit**
  - Sum of resident memory sizes < Sum of physical memory sizes
    - Decoupling memory mapping from memory allocation
  - Support for large sparse virtual address spaces
    - Potentially more efficient resource usage
  - Out-of-bound out-of-memory condition
    - Victim finding
  - Less predictable

Note on Caches

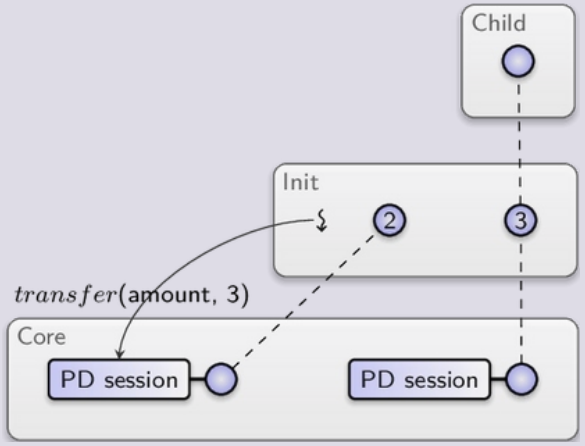
- **Separate instruction and data caches**
  - Self-modifying code (N.B.: including code loading)
- **Virtually-indexed caches**
  - Mostly used for L1 instruction caches nowadays
  - Cache homonyms (same VPN referring to different PFN)
    - Flush on each address space switch costly
    - Distinct virtual addresses unpractical
    - ASID tagging (ASID management by operating system)
  - Cache synonyms (different VPN referring to same PFN)
    - Shared memory or multiple mappings leading to stale data
    - Synonym detection, cache coloring
    - Hardware synonym detection

Resource Limiting

- **Via accounting**
  - Linux cgroups
    - (Hierarchical) groups of processes associated with parameters
      - Children typically belong to the same group as parent
      - Controllers: cpu, cpuacct, cpuset, freezer, hugetlb, io, memory, perf_event, pids, rdma
- **Via visibility**
  - Namespaces (a.k.a. containers, zones, etc.)
    - Non-visible resources are not accessible

Resource Limiting

- **Via delegation**
  - Resource trading in Genode
    - Delegating resources to children
    - Clients paying for server requests (with upgrades)



Thank you!

Questions?