Advanced Operating Systems
Summer Semester 2022/2023

Martin Děcký

3

Memory and I/O

Address Space

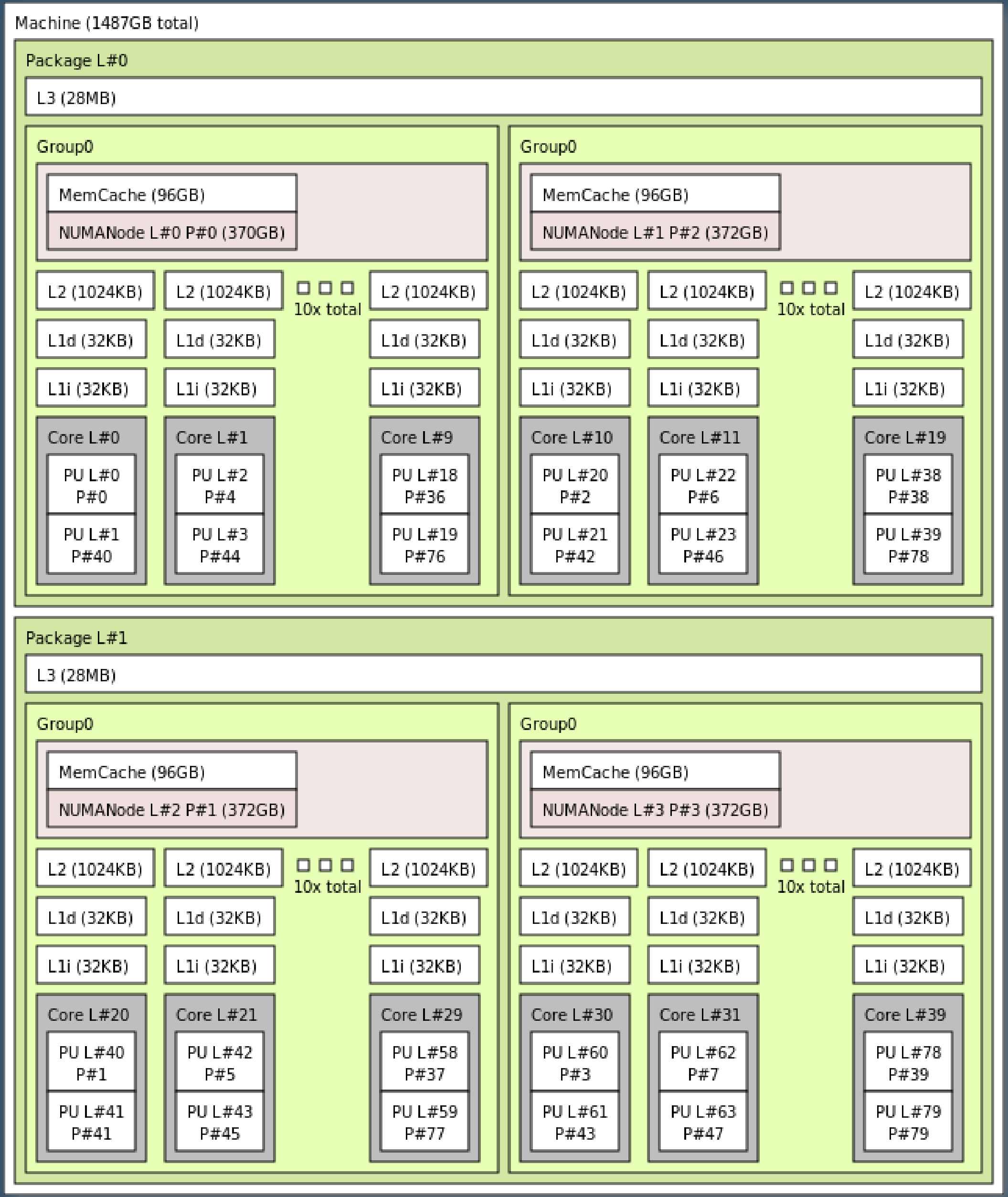
- **Universal abstraction for accessing data (code is just a form of data)**
  - Physical memory
    - Bytes, words, instructions (or similar)
    - Software virtual memory / Device virtual memory
      - Pages (or similar)
  - I/O memory
    - Bytes, words, ports (or similar)
    - Can be embedded in physical memory (memory-mapped I/O)
  - Permanent memory
    - Blocks, pages (or similar)
    - Can be combined with physical memory
  - Object space
    - Keys, capabilities (or similar)

Physical Memory Myths

- **Random access performance**
  - Seems to be O(1) in time units, but in reality it is closer to O(√*n*)
    - Where *n* is the size of the working set
    - Performance effects of the cache hierarchy
- **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

Non-Uniform Memory Access

- **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.

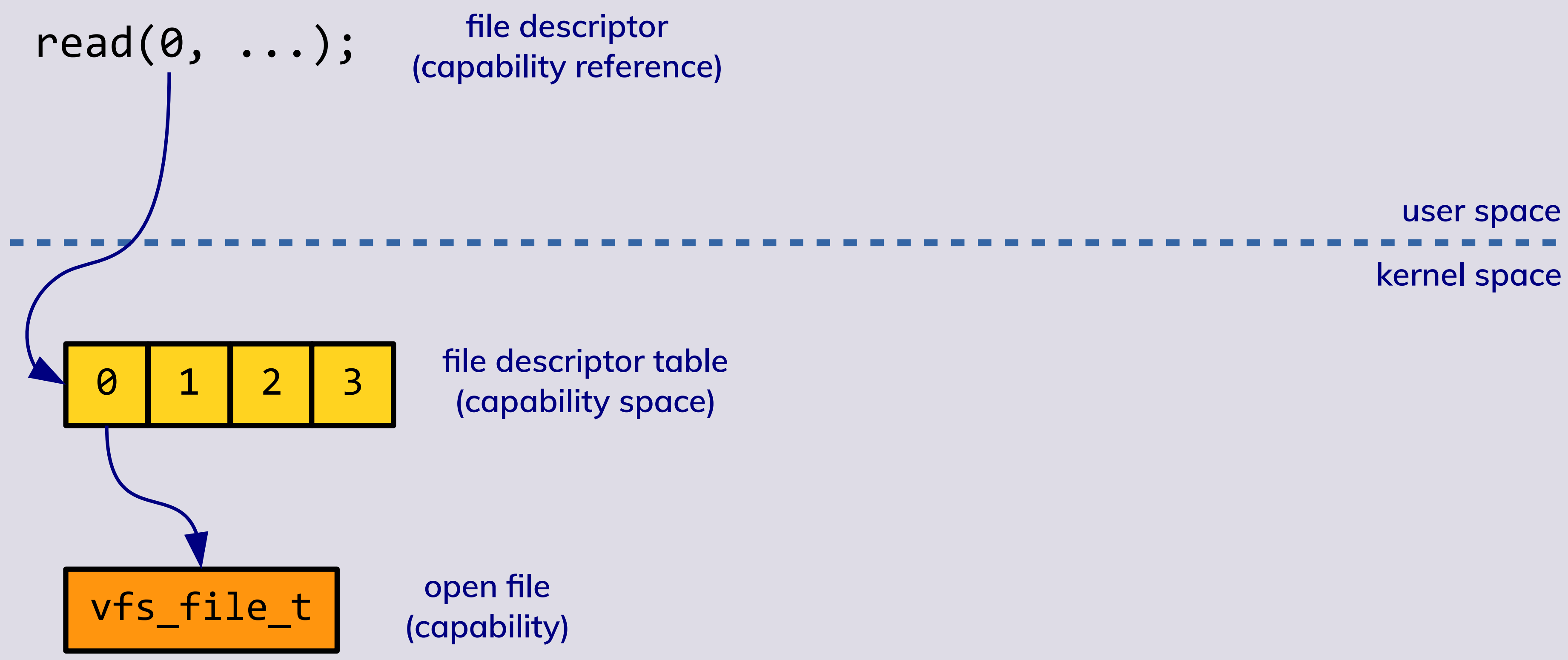
Capabilities

- **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

- **Typical 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

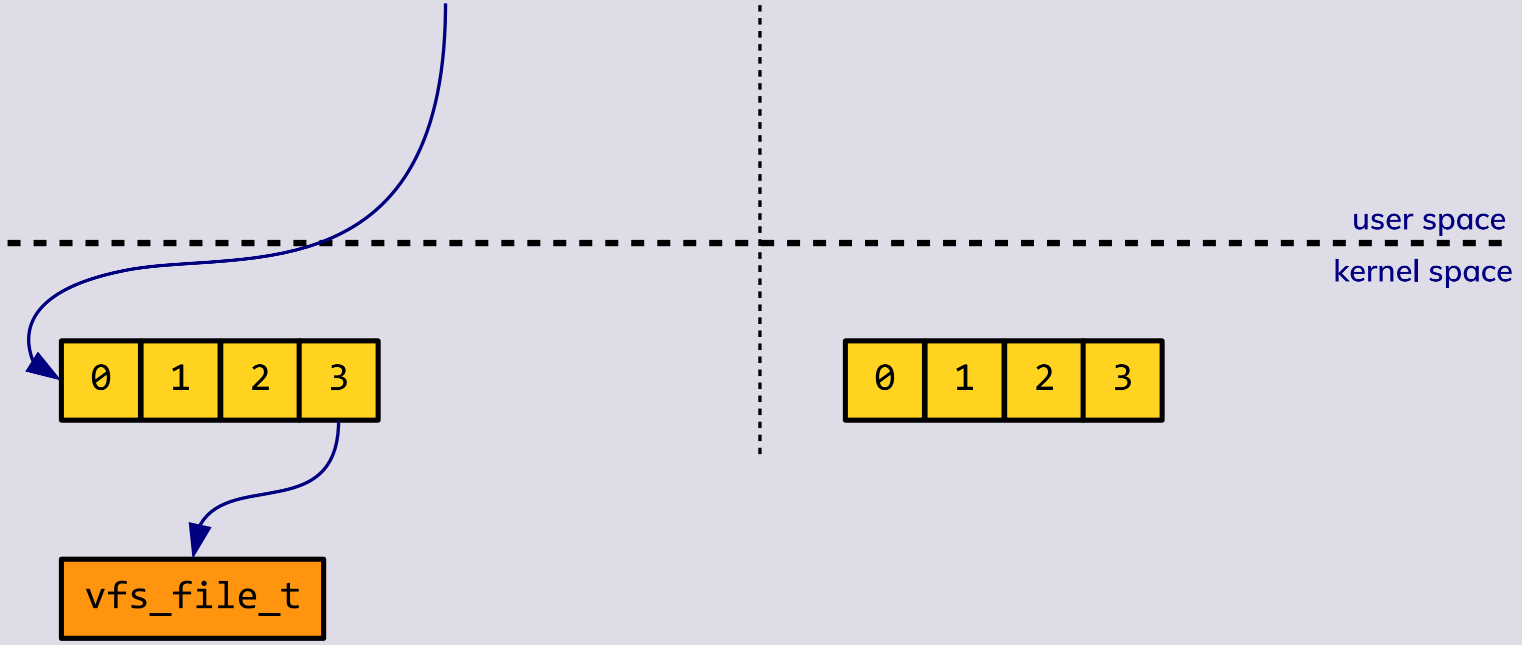
- **Actions performed with capabilities**
  - Can be restricted by the capability reference
    - Multiple capability references can point to the same capability
  - **Invoke**
    - Execute a “business logic” method on the target object
  - **Clone / Mint**
    - Create a duplicate capability reference (possibly with restricted 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:



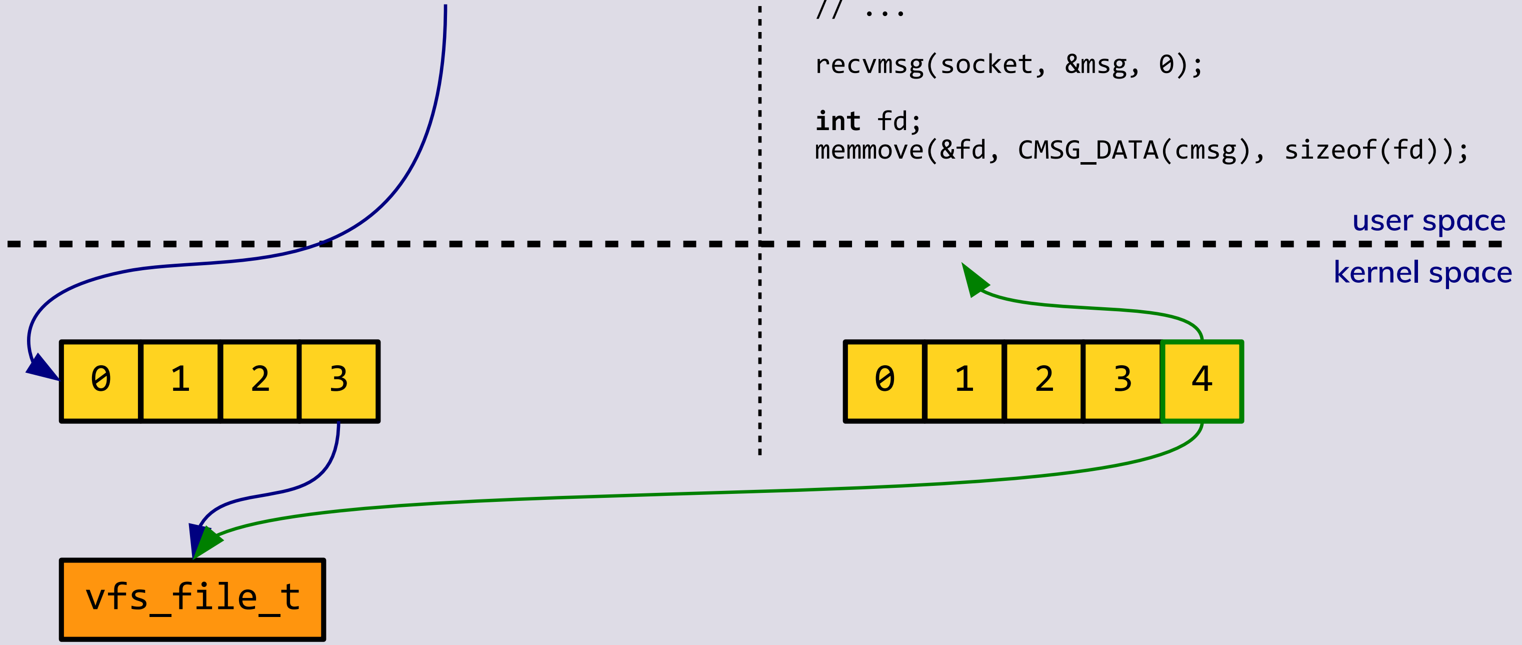
Capability Delegation

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:

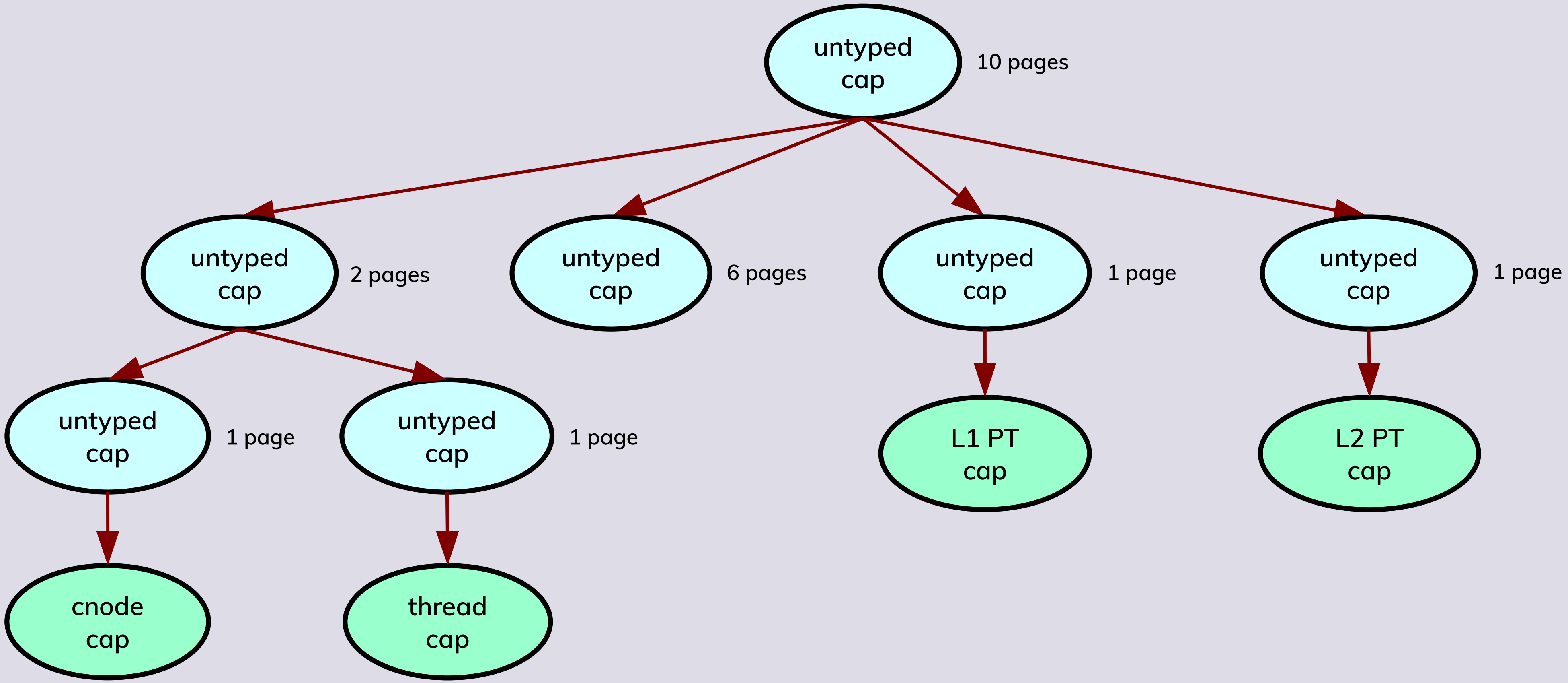
```
struct msghdr msg;
struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
// ...

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



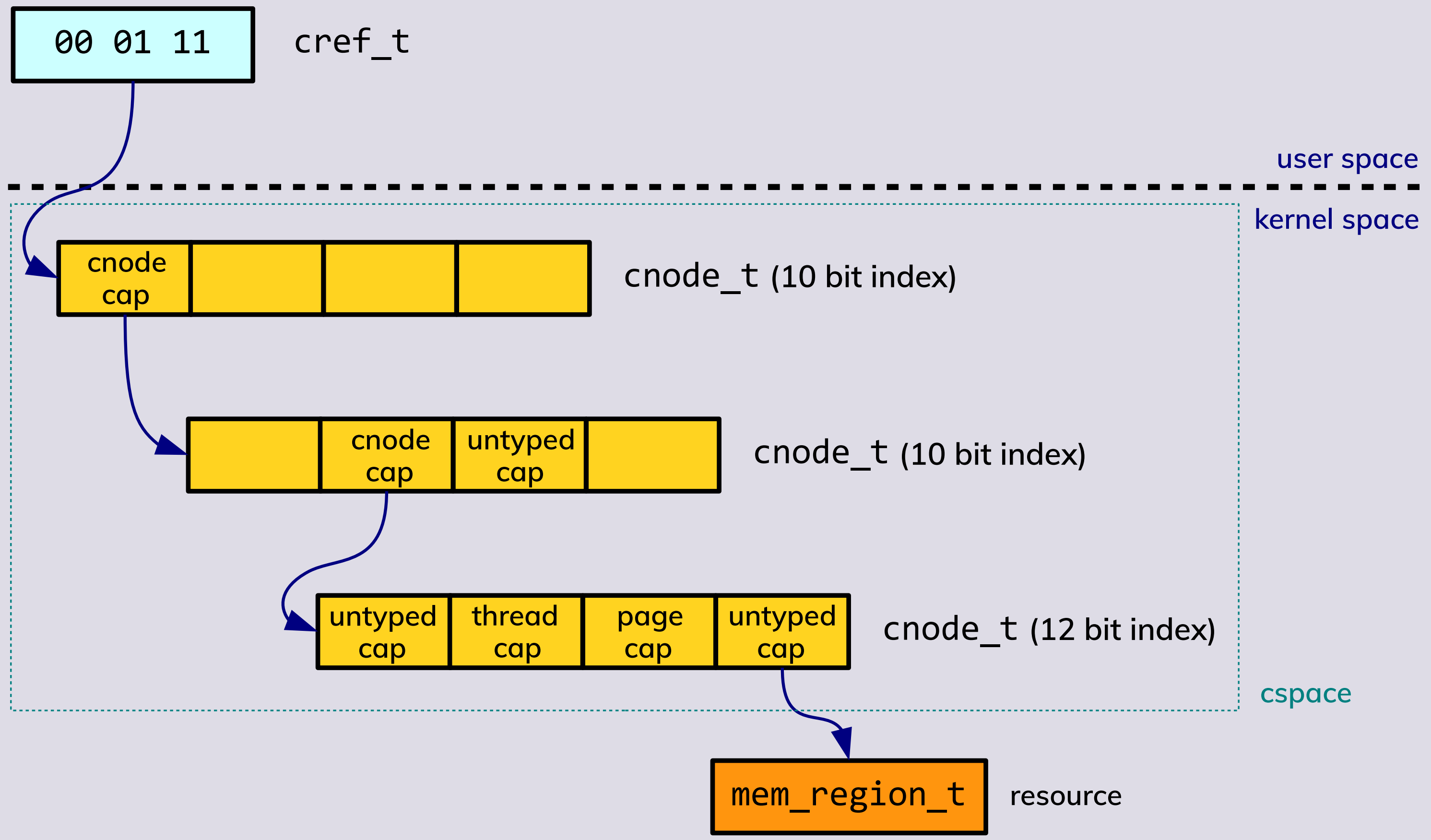
Capability References and Spaces

- **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

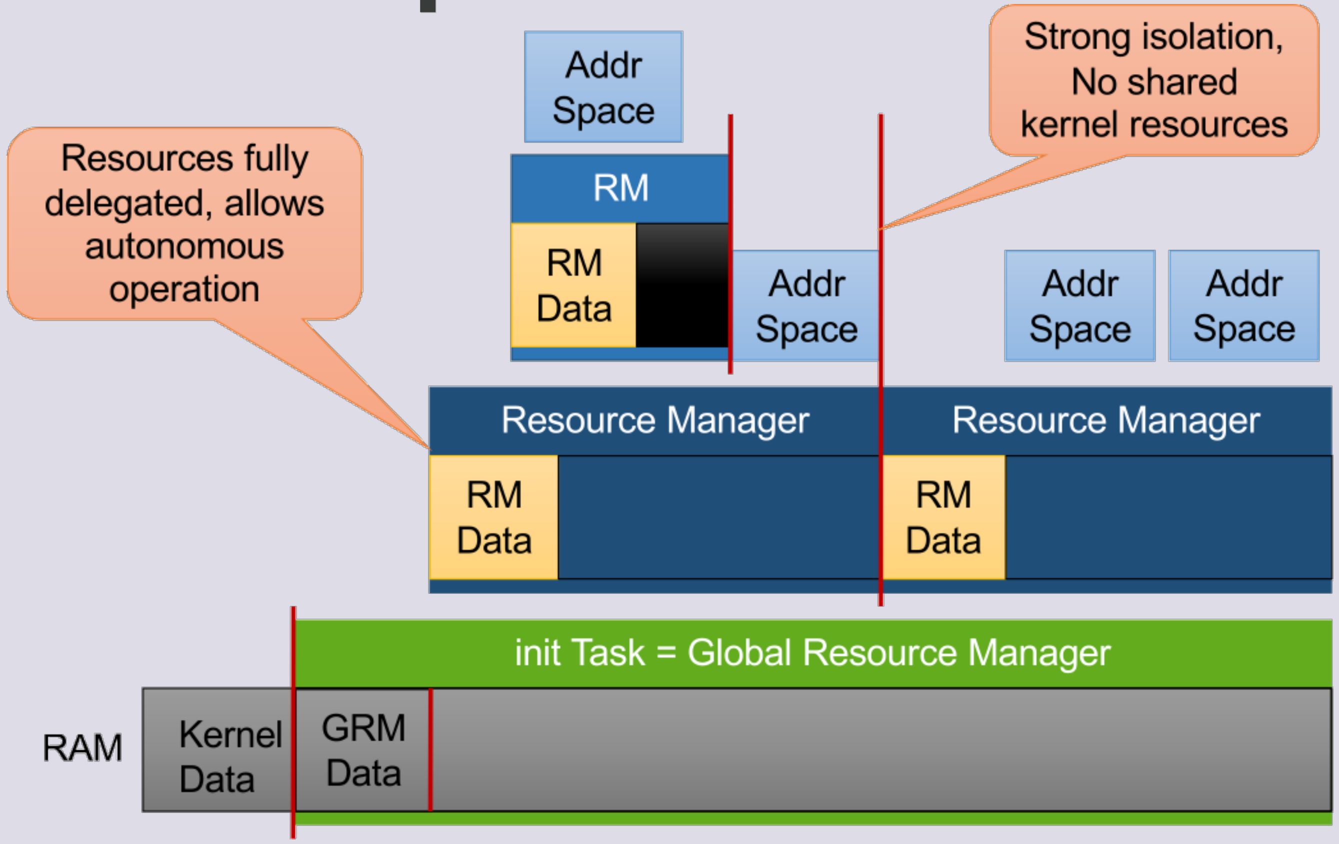
- **Kernel objects**
  - `UntypedObject` (physical memory range)
  - `TCBObject` (thread)
  - `EndpointObject` (IPC calls destination)
  - `AsyncEndpointObject` (signal recipient)
  - `CapTableObject` (array of capabilities)
  - `X86_4K` (4 KiB frame)
  - `X86_4M` (4 MiB frame)
  - `X86_PageTableObject`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

Capabilities Example: seL4

- **Capability derivation**

```
seL4_X86_Untyped_Retype(cnode_selector(phys_addr), seL4_X86_4K, ..., ..., ..., ...,
  phys_addr >> FRAME_WIDTH, 1);

seL4_X86_Untyped_Retype(cnode_selector(pt_phys_addr), seL4_X86_PageTableObject, ..., ..., ..., ...,
  pt_phys_addr >> FRAME_WIDTH, 1);

seL4_X86_Untyped_Retype(cnode_selector(pd_phys_addr), seL4_X86_PageDirectoryObject, ..., ..., ..., ...,
  pd_phys_addr >> FRAME_WIDTH, 1);


seL4_X86_PageTable_Map(cnode_selector(pt_phys_addr), cnode_selector(pd_phys_addr), virt_addr,
  seL4_X86_Default_VMAttributes);

seL4_X86_Page_Map(cnode_selector(phys_addr), cnode_selector(pd_phys_addr), virt_addr, seL4_AllRights,
  seL4_X86_Default_VMAttributes);
```

Capabilities Example: seL4


Addr Space
Strong isolation, No shared kernel resources
Resources fully delegated, allows autonomous operation
RM
RM Data
Addr Space
Addr Space
Addr Space
Resource Manager
Resource Manager
RM Data
RM Data
init Task = Global Resource Manager
RAM
Kernel Data
GRM Data

Source: Gernot Heiser: *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

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 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

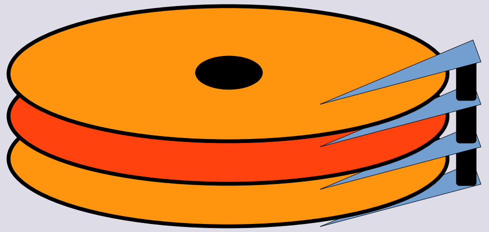
Non-Volatile Memory

- **Historically biased towards rotational media**
  - Cylinder / Head / Sector → Linear (Logical) Block Addressing
    - Originally interface abstraction not very high
      - Hard sectored → Soft sectored (with remapping)
    - 512 B blocks → 4096 B blocks (floppy/hard drives)
    - 2048 B blocks (optical drives), 2353 B blocks (raw optical drives)
  - Latency several orders of magnitude larger than volatile memory
    - Originally interface I/O efficiency not very important
      - Single tenant
      - Single request stream

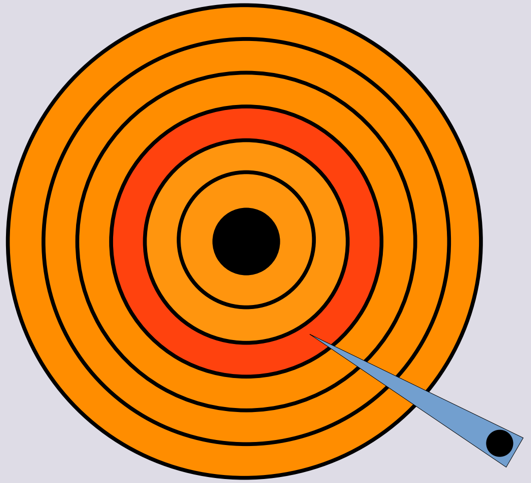
Non-Volatile Memory

- **Historically biased towards rotational media**

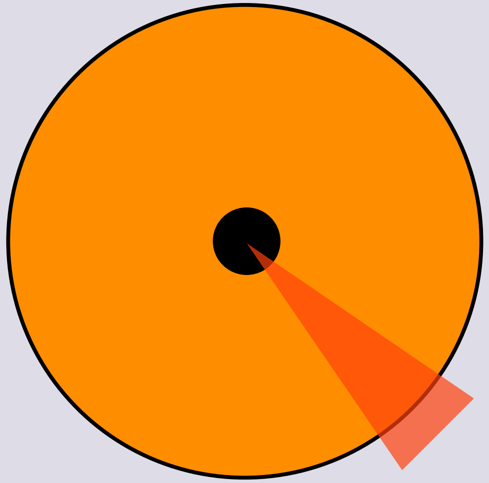
heads



cylinders
(tracks)



sectors & interleaving



Non-Volatile Memory

- **Historically biased towards rotational media**
  - Multi-tenant performance dominated by physical seek time
  - Still mostly via single request stream
    - Software I/O scheduling (shortest seek first, elevator/sweep, shortest deadline first, etc.)
      - Might not have the most accurate physical storage information (i.e. remapping)
    - I/O command batching (queuing)
      - Leaving the optimal I/O order (within the batch) to hardware
      - Incorporates interrupt coalescing

Non-Volatile Memory

- **Solid-state drives**
  - Differing characteristics from rotational drives
    - Physical characteristics mostly unimportant
    - Addressing characteristics
      - Different native read/write and erase blocks
        - Write amplification
      - Physical addressing more like volatile memory
    - Latency much closer to volatile memory
      - Performance dominated by interface I/O efficiency
    - High degree of internal parallelism
    - Unique wear characteristics

Non-Volatile Memory

- **Solid-state drives**
  - Reflection in the I/O interface (e.g. NVMe)
    - Generally provides the common LBA abstraction
      - Wear leveling, block remapping and garbage collection in hardware Flash Transition Layer (FTL)
        - Frequently implemented as multi-level log-based storage
        - Software *trim* hint to indicate unused (erased) blocks
        - Trade-offs between write amplification, performance, idle characteristics
    - Low latency and parallel access
      - “Unlimited” request queues with lock-less access
      - “Unlimited” command queuing
      - Interrupt coalescing & multiple interrupt groups
      - Full-duplex scatter-gather DMA

Non-Volatile Memory

- **Solid-state drives**
  - Exposing more of the hardware architecture to software
    - Addressing
      - Open-channel SSD
      - NVMe Zoned Namespace
        - Note: Zones also useful for Shingled Magnetic Recording (SMR)
    - Compute off-loading
      - Basic NVMe I/O commands: *Compare*, *Write Zeroes*, *Copy*
      - NVMe Key Value command set
      - Near data computing (proposed)

Storage Near Data Computing

- **Off-loading computation to storage controller**
  - Decrease latency, improve throughput, decrease energy consumption
  - Improve performance
    - Trade-off: Lower performance of embedded cores
      - Still a performance boost when compute cores are already loaded

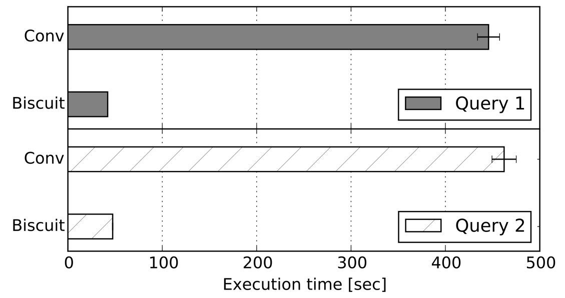
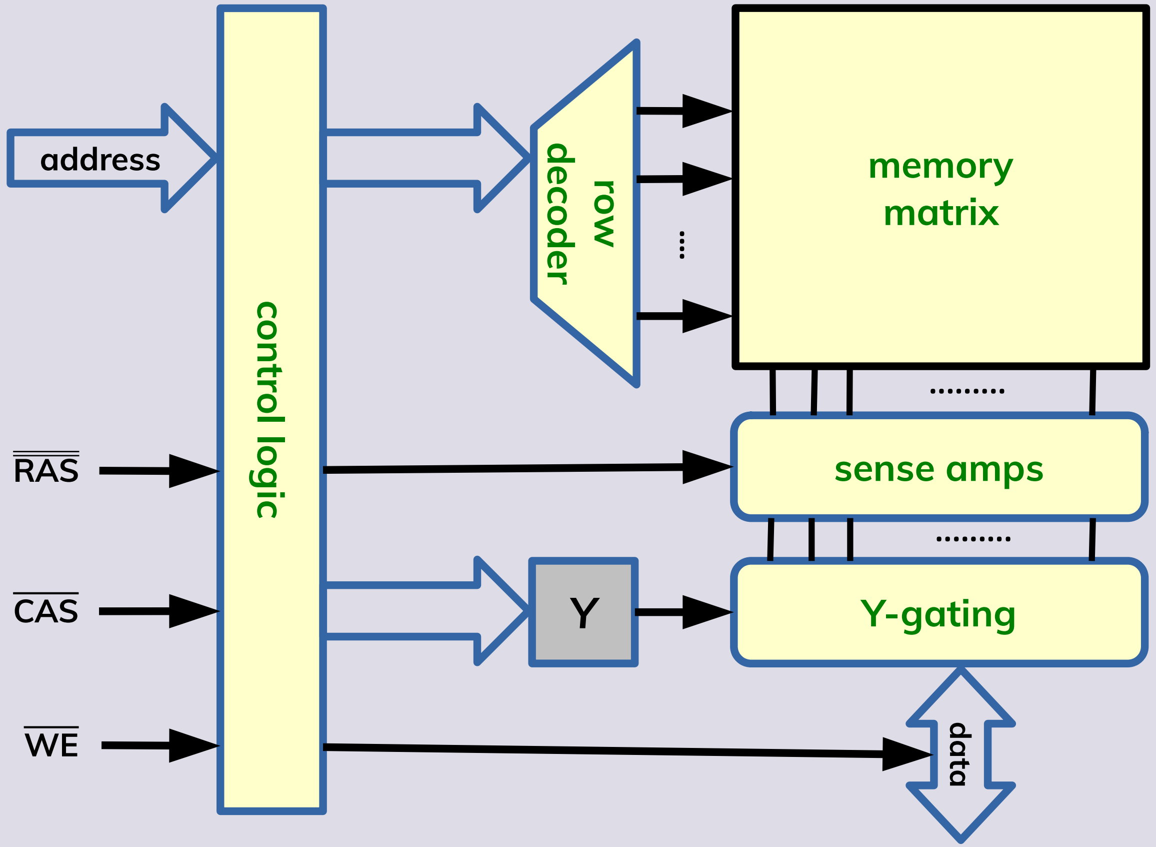


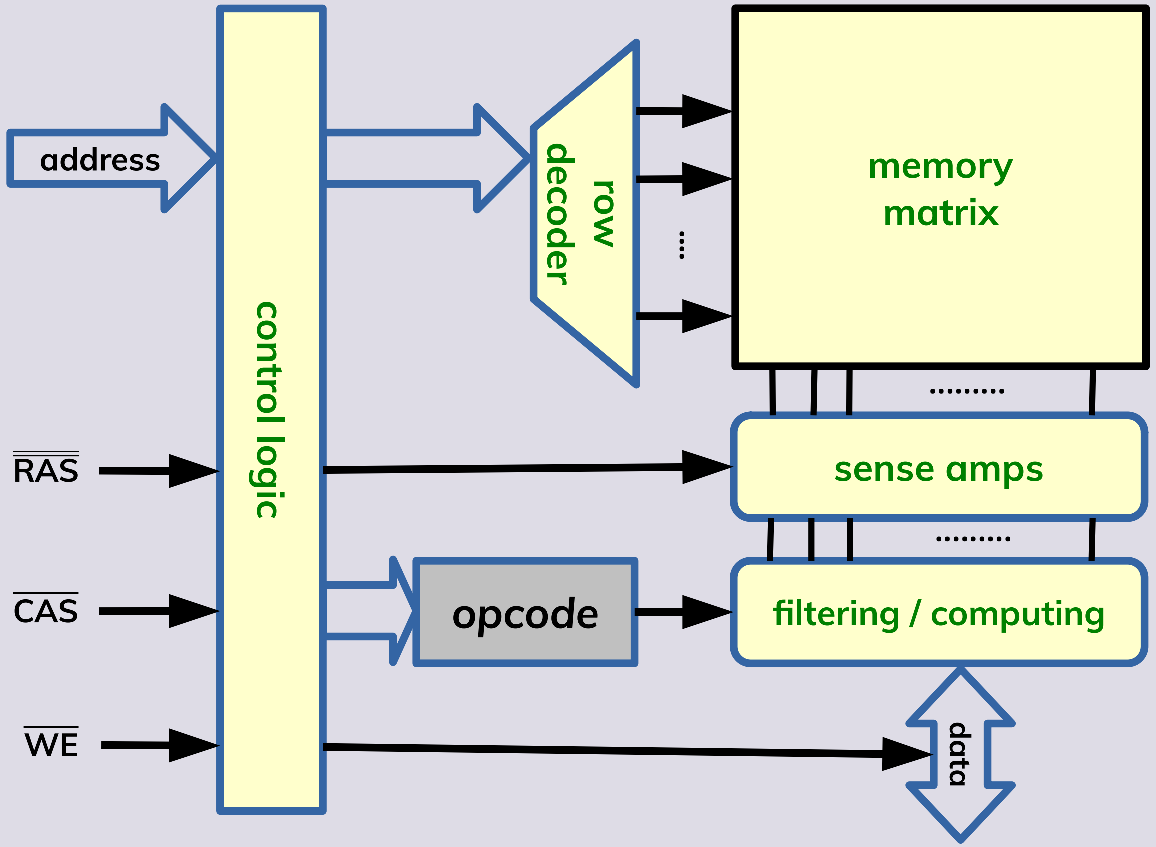
Figure 8. Performance of SQL queries on lineitem table.

Source: Gu B., Yoon A. S., Bae D.-H., Jo I., Lee J., Yoon J., Kang J.-U., Kwon M., Yoon C., Cho S., Jeong J., Chang D.: *Biscuit: A Framework for Near-Data Processing of Big Data Workloads*, in Proceedings of 43rd Annual International Symposium on Computer Architecture, ACM/IEEE, 2016

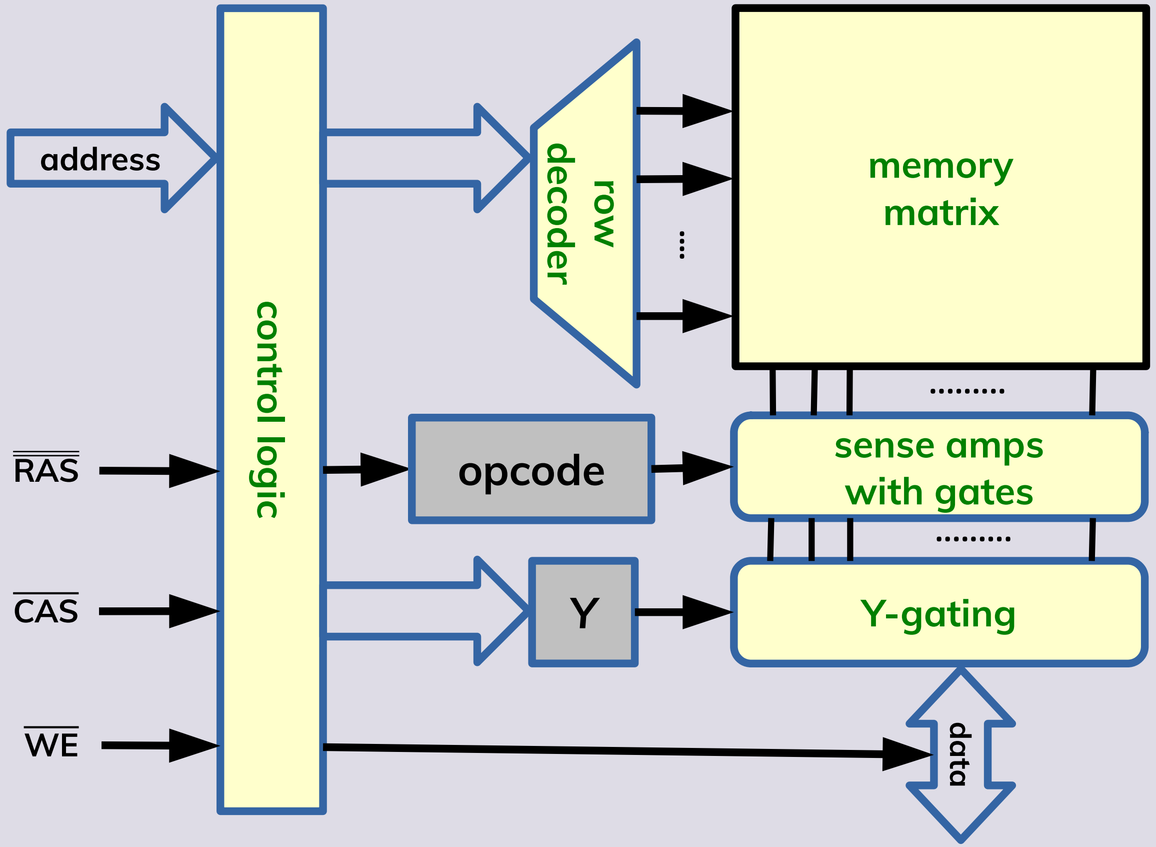
Memory Near Data Computing



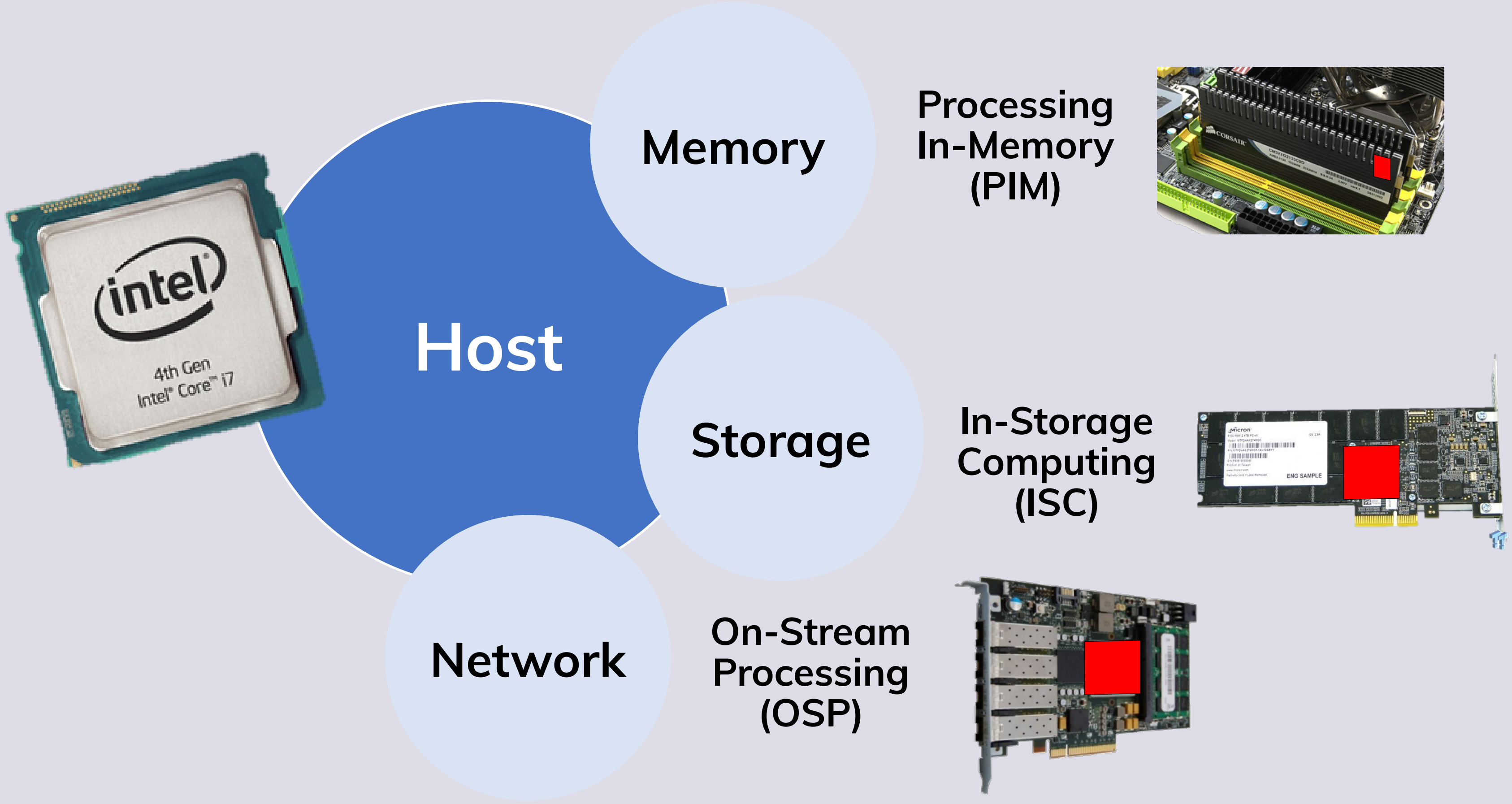
Memory Near Data Computing



Memory Near Data Computing



Generic Near Data Computing


intel
4th Gen
Intel® Core™ i7
Host
Memory
Storage
Network
Processing In-Memory (PIM)
In-Storage Computing (ISC)
On-Stream Processing (OSP)

Generic Near Data Computing

- **Challenges**
  - Universal open interface standard
    - Currently extensions of existing I/O interfaces
  - Universal programming model
    - Stream / flow processing
    - Association of compute units with data
  - Universal compute model
    - ISA
    - Safety, security considerations
  - Off-loading vs. distributed computing

D3S

Thank you!

Questions?