OS Challenges for Modern Memory Systems

Christian Dietrich, Daniel Lohmann

Technische Universität Hamburg, Leibniz Universität Hannover

Winterschool on Operating Systems 2023

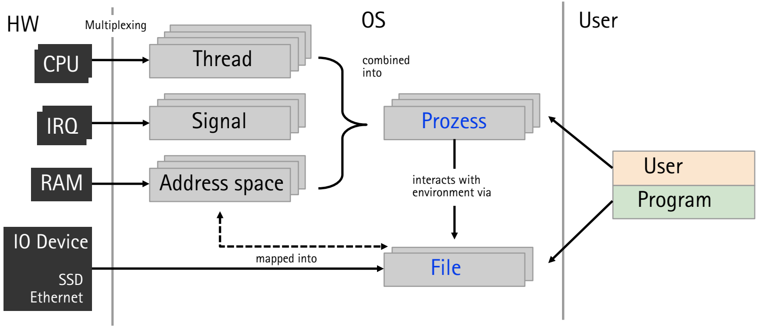
Überblick: Teil

6 OS Challenges for Modern Memory Systems

- 6.1 Virtualizing Memory – A Short Recap
- 6.2 Hardware Developments
- 6.3 ParPerOS – Contention-Avoiding Design
- 6.4 Summary and Conclusion
- 6.5 Referenzen

About This Lecture

- **Operating System** ↦ multiplexing and isolation of hardware
  by means of hardware virtualization
- Virtual hardware is represented by the OS's **basic abstractions**: Example UNIX

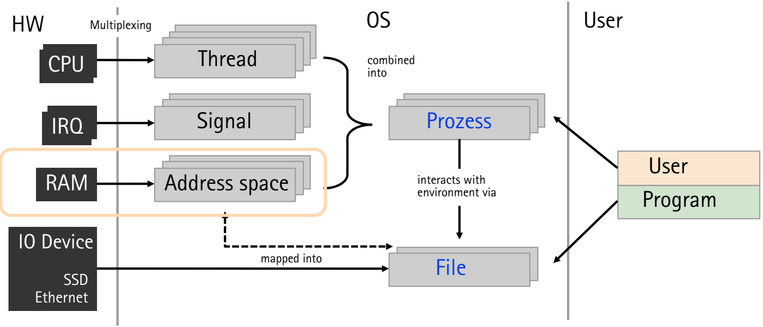


About This Lecture

- **Operating System** ↦ multiplexing and isolation of hardware by means of hardware virtualization
- Virtual hardware is represented by the OS's **basic abstractions**: Example UNIX

This lecture:

We mostly focus on basic **OS-internal memory management.**



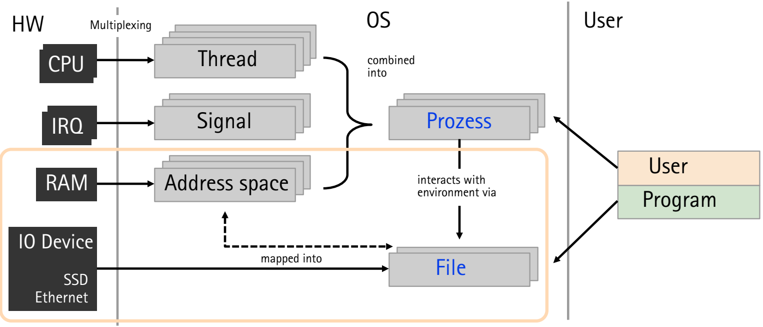
About This Lecture

- **Operating System** ↦ multiplexing and isolation of hardware
  by means of hardware virtualization
- Virtual hardware is represented by the OS's **basic abstractions**: Example UNIX

This lecture:

We mostly focus on basic **OS-internal memory management**.

But of course, the area and problem is **bigger...**



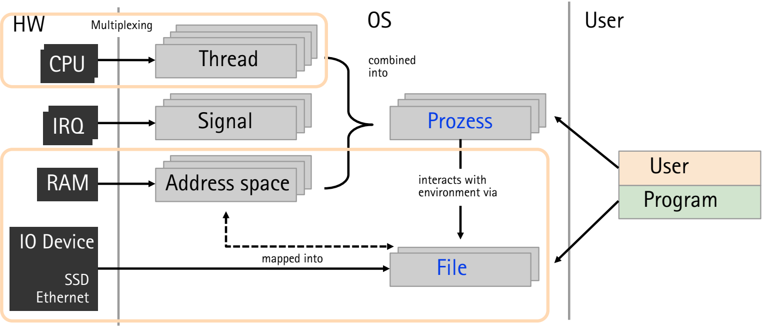
About This Lecture

- **Operating System** ↦ multiplexing and isolation of hardware
  by means of hardware virtualization
- Virtual hardware is represented by the OS's **basic abstractions**: Example UNIX

This lecture:
We mostly focus on basic **OS-internal memory management**.

But of course, the area and problem is **bigger**...

...and in the end, its also a lot about **contention**.



6.1 Virtualizing Memory – A Short Recap

Recap: Physical and Virtual Address Space

- **Physical address space A_p** ⇝ *all hardware addresses*
  - defined by the hardware manufactor (OEM)
  - contains memory-mapped hardware devices and all physical memory (RAM, ROM, NVRAM)

  ⇝ **main memory**

Recap: Physical and Virtual Address Space

- **Physical address space A_p** ⇝ *all hardware addresses*
  - defined by the hardware manufactor (OEM)
  - contains memory-mapped hardware devices and all physical memory (RAM, ROM, NVRAM)

  ⇝ **main memory**

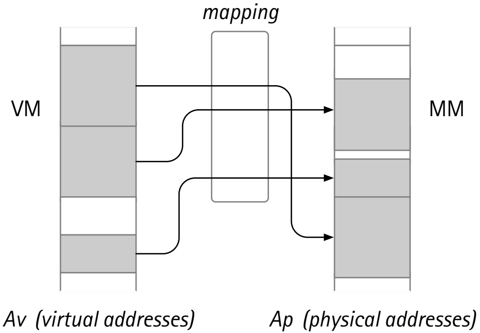
- **Virtual address space A_v** ⇝ *all software addresses (for some process p)*
  - defined by the operating system
  - containes memory-mapped files and all code and data of the programm running in p
  - OS dynamically maps logical addresses to physical addresses and (optionally) also backing store: $p : A_l \mapsto A_p \vee BS$. ⇝ isolation ⇝ multiplexing

  ⇝ **(virtual) working memory** of process p

Recap: Memory Virtualization

- **Virtual addresses** generated by process *p* are translated
  - transparently via an OS-controlled process-specific mapping $p : A_v \mapsto A_p$
  - mapping is done „on access“ by the MMU (memory management unit)

Segmentation: Mapping of objects of arbitrary size, linearly stored in both, A_v and A_p. ⇝ [18]

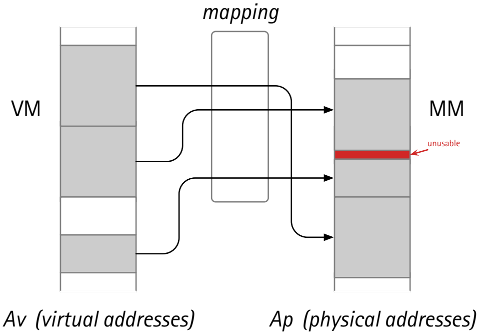


Recap: Memory Virtualization

- **Virtual addresses** generated by process *p* are translated
  - transparently via an OS-controlled process-specific mapping $p : A_v \mapsto A_p$
  - mapping is done „on access“ by the MMU (memory management unit)

Segmentation: Mapping of objects of arbitrary size, linearly stored in both, A_v and A_p. ⇝ [18]

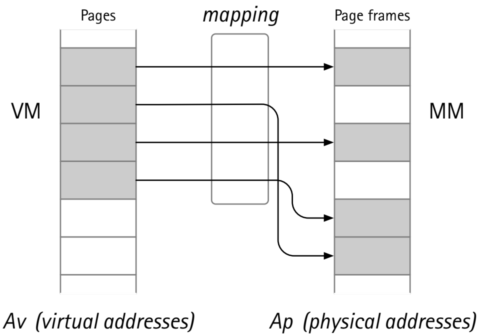
Leads to *external fragmentation*.



Recap: Memory Virtualization

- **Virtual addresses** generated by process *p* are translated
  - transparently via an OS-controlled process-specific mapping $p : A_v \mapsto A_p$
  - mapping is done „on access“ by the MMU (memory management unit)

Paging: Mapping of equal-sized fragments of objects, linearly stored only in A_v. ⇝ [9]

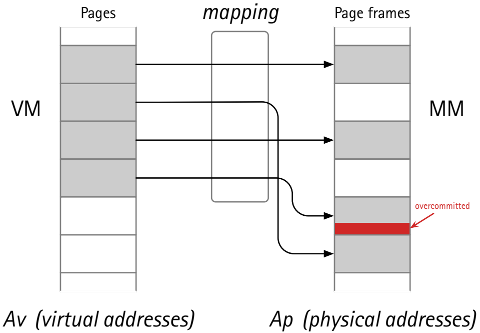


Recap: Memory Virtualization

- **Virtual addresses** generated by process *p* are translated
  - transparently via an OS-controlled process-specific mapping $p : A_v \mapsto A_p$
  - mapping is done „on access" by the MMU (memory management unit)

Paging: Mapping of equal-sized fragments of objects, linearly stored only in A_v. ⇝ [9]

Causes some *internal fragmentation*, but eases memory management a lot.



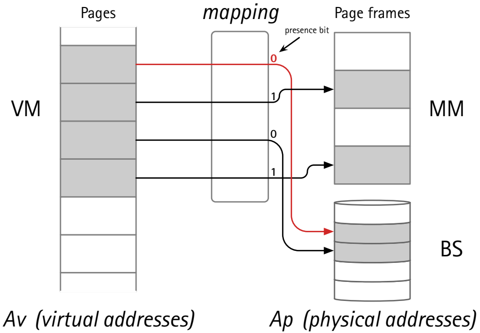
Recap: Memory Virtualization

- **Virtual addresses** generated by process p are translated
  - transparently via an OS-controlled process-specific mapping $p : A_v \mapsto A_p$
  - mapping is done „on access“ by the MMU (memory management unit)

Paging: Mapping of equal-sized fragments of objects, linearly stored only in A_v. ⇝ [9]

Causes some *internal fragmentation*, but eases memory management a lot.

Provides for efficient RAM virtualization, by optionally trapping into the OS on access.



Recap: Memory Virtualization

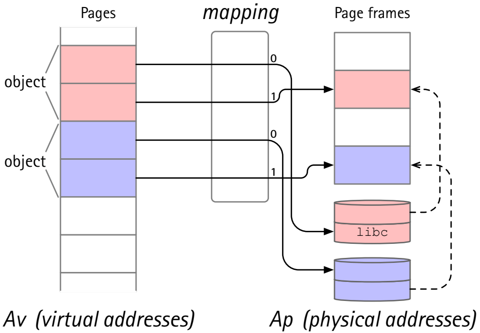
- **Virtual addresses** generated by process *p* are translated
  - transparently via an OS-controlled process-specific mapping $p : A_v \mapsto A_p$
  - mapping is done „on access“ by the MMU (memory management unit)

Paging: Mapping of equal-sized fragments of objects, linearly stored only in A_v. ⇝ [9]

Causes some *internal fragmentation*, but eases memory management a lot.

Provides for efficient RAM virtualization, by optionally trapping into the OS on access.

Provides for virtualization of *anything* by memory objects with object-specific *pagers*. ⇝ [1]



Recap: Memory Virtualization

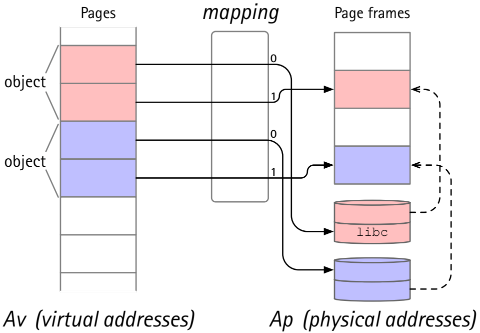
- **Virtual addresses** generated by process *p* are translated
  - transparently via an OS-controlled process-specific mapping $p : A_v \mapsto A_p$
  - mapping is done „on access“ by the MMU (memory management unit)

Paging: Mapping of equal-sized fragments of objects, linearly stored only in A_v. ⇝ [9]

Causes some *internal fragmentation*, but eases memory management a lot.

Provides for efficient RAM virtualization, by optionally trapping into the OS on access.

Provides for virtualization of *anything* by memory objects with object-specific *pagers*. ⇝ [1]



- Fundamental principle of **demand paging** ⇝ OS can do lots of thing *lazily*.

Recap: Memory Virtualization

- Fundamental principle of **demand paging** ⇝ provide RAM only *implicitly*.
  - Delay provision of page frames until actually needed.
  - Implicitly share page frames as long as possible.
  - Transfer page frames instead of data *(zero copy)*.

Unified buffer cache: [11]
The 4 KiB page frame has become the defacto entity for *everything!*

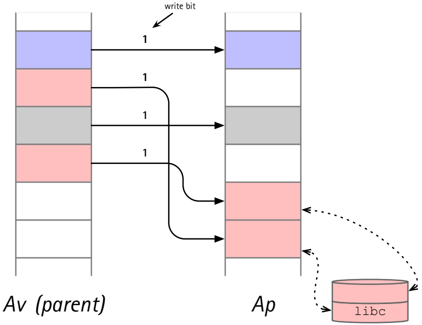
Recap: Memory Virtualization

- Fundamental principle of **demand paging** ⇝ provide RAM only *implicitly*.
  - Delay provision of page frames until actually needed.
  - Implicitly share page frames as long as possible.
  - Transfer page frames instead of data *(zero copy)*.

Unified buffer cache: [11]
The 4 KiB page frame has become the defacto entity for *everything!*

- Textbook examples

`fork()` and the mighty **copy-on-write** (COW)



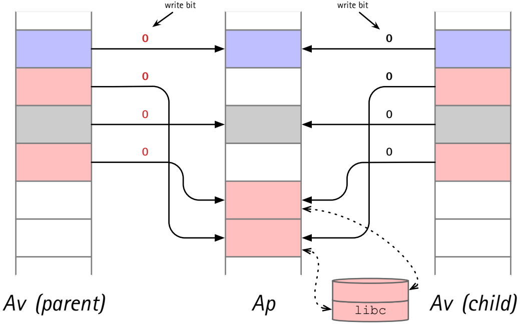
Recap: Memory Virtualization

- Fundamental principle of **demand paging** ⇝ provide RAM only *implicitly*.
  - Delay provision of page frames until actually needed.
  - Implicitly share page frames as long as possible.
  - Transfer page frames instead of data *(zero copy)*.

Unified buffer cache: [11]
The 4 KiB page frame has become the defacto entity for *everything!*

- Textbook examples

`fork()` and the mighty **copy-on-write** (COW)



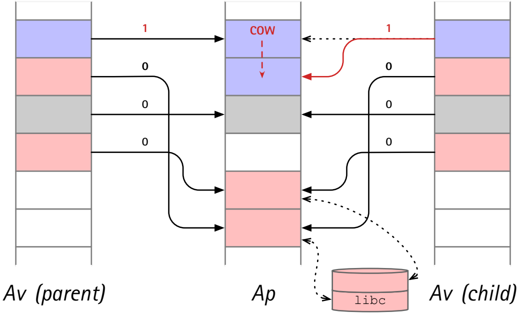
Recap: Memory Virtualization

- Fundamental principle of **demand paging** ⇝ provide RAM only *implicitly*.
  - Delay provision of page frames until actually needed.
  - Implicitly share page frames as long as possible.
  - Transfer page frames instead of data *(zero copy)*.

Unified buffer cache: [11]
The 4 KiB page frame has become the defacto entity for *everything!*

- Textbook examples

`fork()` and the mighty **copy-on-write** (COW)



Recap: Memory Virtualization

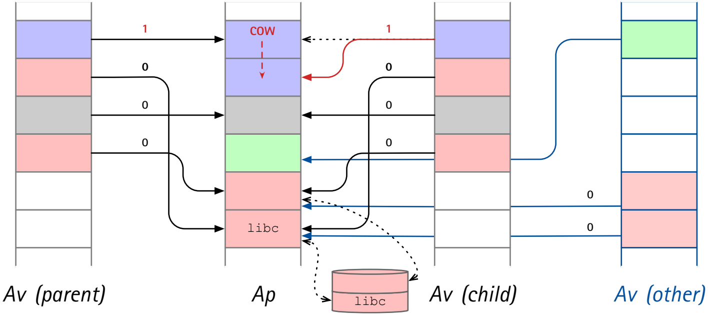
- Fundamental principle of **demand paging** ⇝ provide RAM only *implicitly*.
  - Delay provision of page frames until actually needed.
  - Implicitly share page frames as long as possible.
  - Transfer page frames instead of data *(zero copy)*.

Unified buffer cache: [11]
The 4 KiB page frame has become the defacto entity for *everything!*

- Textbook examples

`fork()` and the mighty **copy-on-write** (COW)

Implicit sharing of other **objects**, like file-mapped binaries.



Recap: Memory Virtualization

- Fundamental principle of **demand paging** ⇝ provide RAM only *implicitly*.
  - Delay provision of page frames until actually needed.
  - Implicitly share page frames as long as possible.
  - Transfer page frames instead of data *(zero copy)*.

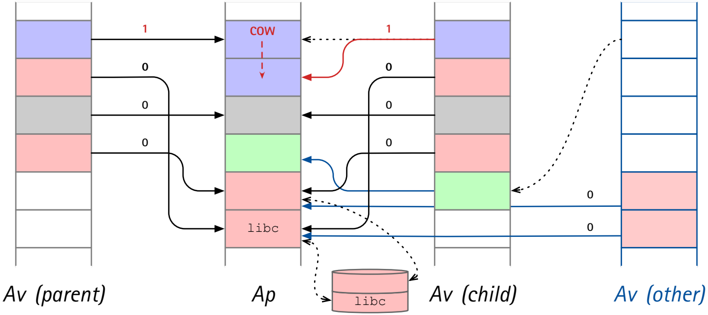
Unified buffer cache: [11]
The 4 KiB page frame has become the defacto entity for *everything!*

- Textbook examples

`fork()` and the mighty **copy-on-write** (COW)

Implicit sharing of other **objects**, like file-mapped binaries.

Data transfer by page-level **zero copy**, also from devices.



Recap: Memory Virtualization

- Fundamental principle of **demand paging** ⇝ provide RAM only *implicitly*.
  - Delay provision of page frames until actually needed.
  - Implicitly share page frames as long as possible.
  - Transfer page frames instead of data *(zero copy)*.

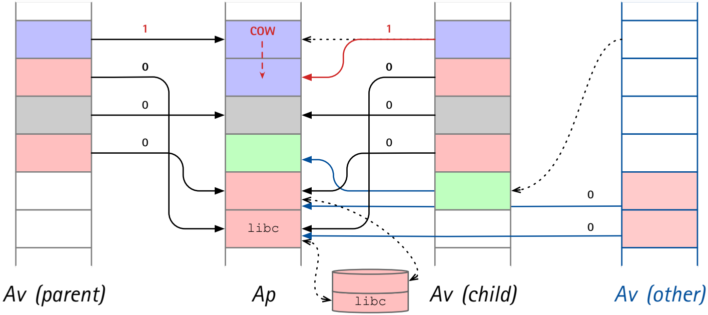
Unified buffer cache: [11]
The 4 KiB page frame has become the defacto entity for *everything!*

- Textbook examples

`fork()` and the mighty **copy-on-write** (COW)

Implicit sharing of other **objects**, like file-mapped binaries.

Data transfer by page-level **zero copy**, also from devices.



↪ Save on the **scarce and expensive** physical memory and avoid/delay **costly** copy operations.

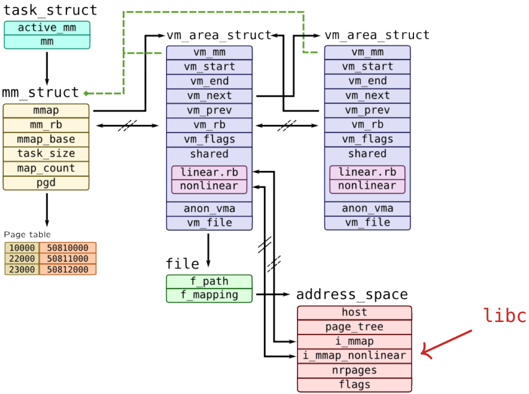
Problem 1: The Cost of Sharing

- Sharing demands **extensive** OS-internal bookkeeping
  - Object-specific virtual ⟷ physical mappings, lots of reference counting.
  - Nested COW-relationships make things even more complicated.

Problem 1: The Cost of Sharing

- Sharing demands **extensive** OS-internal bookkeeping
  - Object-specific virtual ⟷ physical mappings, lots of reference counting.
  - Nested COW-relationships make things even more complicated.
- Example: Linux

Excerpt from Linux's Memory Management



task_struct
active_mm
mm

mm_struct
mmap
mm_rb
mmap_base
task_size
map_count
pgd

Page table

10000	50810000
22000	50811000
23000	50812000

vm_area_struct
vm_mm
vm_start
vm_end
vm_next
vm_prev
vm_rb
vm_flags
shared
linear.rb
nonlinear
anon_vma
vm_file

vm_area_struct
vm_mm
vm_start
vm_end
vm_next
vm_prev
vm_rb
vm_flags
shared
linear.rb
nonlinear
anon_vma
vm_file

file
f_path
f_mapping

address_space
host
page_tree
i_mmap
i_mmap_nonlinear
nrpages
flags

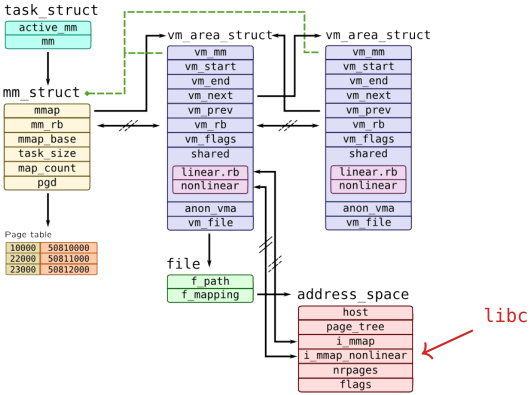
libc

Problem 1: The Cost of Sharing

- Sharing demands **extensive** OS-internal bookkeeping
  - Object-specific virtual ⟷ physical mappings, lots of reference counting.
  - Nested COW-relationships make things even more complicated.
- Example: Linux

Excerpt from Linux's Memory Management

Lots of doubly-linked lists and lookup trees ⇝ locks.



task_struct: active_mm, mm

mm_struct: mmap, mm_rb, mmap_base, task_size, map_count, pgd

Page table

10000	50810000
22000	50811000
23000	50812000

vm_area_struct: vm_mm, vm_start, vm_end, vm_next, vm_prev, vm_rb, vm_flags, shared, linear.rb, nonlinear, anon_vma, vm_file

vm_area_struct: vm_mm, vm_start, vm_end, vm_next, vm_prev, vm_rb, vm_flags, shared, linear.rb, nonlinear, anon_vma, vm_file

file: f_path, f_mapping

address_space: host, page_tree, i_mmap, i_mmap_nonlinear, nrpages, flags

libc

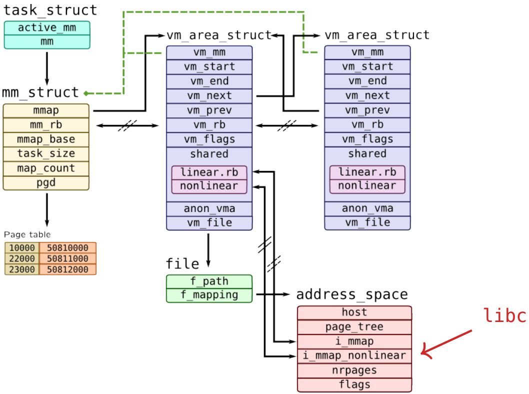
Problem 1: The Cost of Sharing

- Sharing demands **extensive** OS-internal bookkeeping
  - Object-specific virtual ⟷ physical mappings, lots of reference counting.
  - Nested COW-relationships make things even more complicated.
- Example: Linux

Excerpt from Linux's Memory Management

Lots of doubly-linked lists and lookup trees ⇝ locks.

Additional **struct page** (64B) per page frame.
(not shown)



task_struct
active_mm
mm

mm_struct
mmap
mm_rb
mmap_base
task_size
map_count
pgd

Page table

10000	50810000
22000	50811000
23000	50812000

vm_area_struct
vm_mm
vm_start
vm_end
vm_next
vm_prev
vm_rb
vm_flags
shared
linear.rb
nonlinear
anon_vma
vm_file

vm_area_struct
vm_mm
vm_start
vm_end
vm_next
vm_prev
vm_rb
vm_flags
shared
linear.rb
nonlinear
anon_vma
vm_file

file
f_path
f_mapping

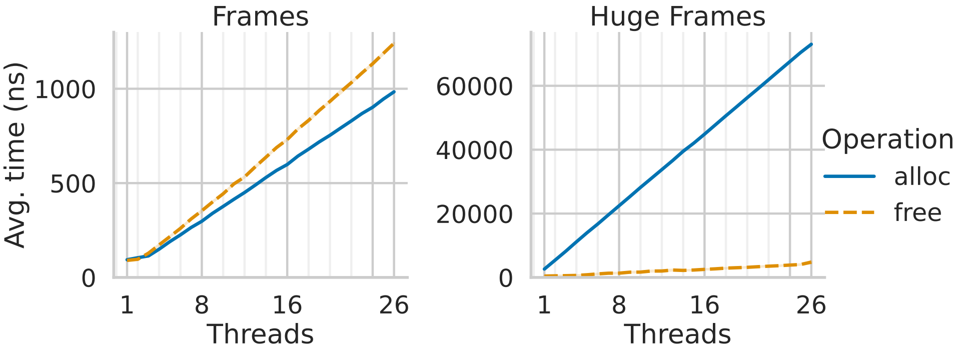
address_space
host
page_tree
i_mmap
i_mmap_nonlinear
nrpages
flags

libc

Problem 1: The Cost of Sharing

- Example: Linux frame allocation

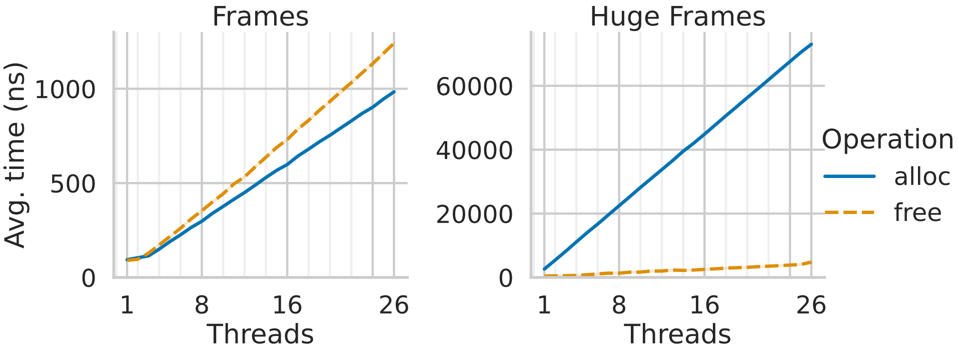
Linux 6.0 on Xeon(R) Gold 5320: 2 × 26 physical cores @ 2.20 GHz, 256/512 GiB DRAM/NVRAM per node



Problem 1: The Cost of Sharing

- Example: Linux frame allocation

Linux 6.0 on Xeon(R) Gold 5320: 2 × 26 physical cores @ 2.20 GHz, 256/512 GiB DRAM/NVRAM per node



Problem: Complex OS-internal bookkeeping hinders scalability

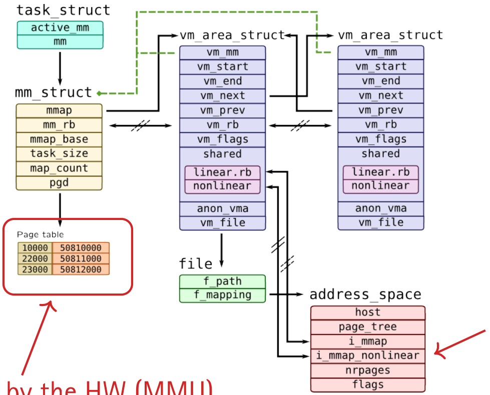
Problem 1: The Cost of Sharing

- Sharing demands **extensive** OS-internal bookkeeping
  - Object-specific virtual ⟷ physical mappings, lots of reference counting.
  - Nested COW-relationships make things even more complicated.
- Example: Linux

Excerpt from Linux's Memory Management

Lots of doubly-linked lists and lookup trees ⇝ locks.

Additional **struct page** (64B) per page frame.
(not shown)

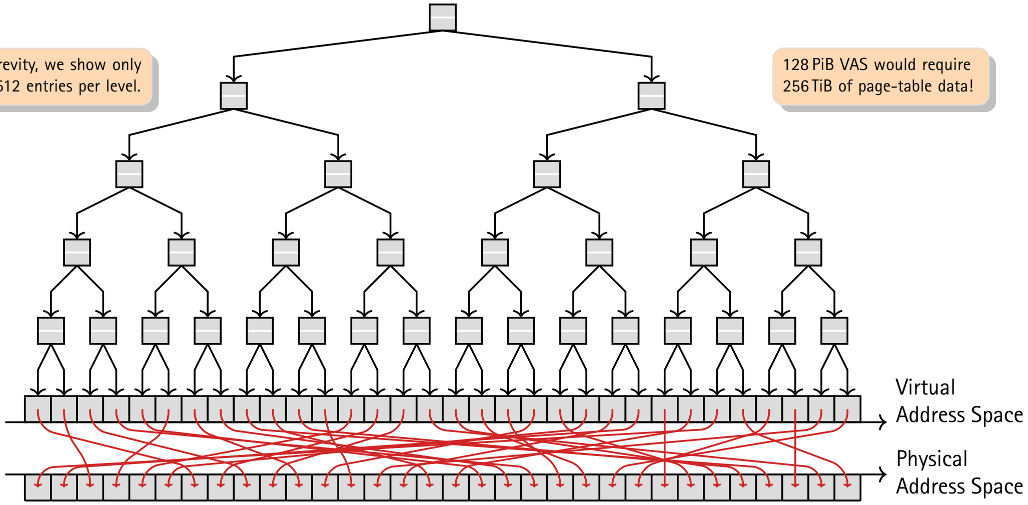


10000	50810000
22000	50811000
23000	50812000

libc

Actually needed by the HW (MMU)

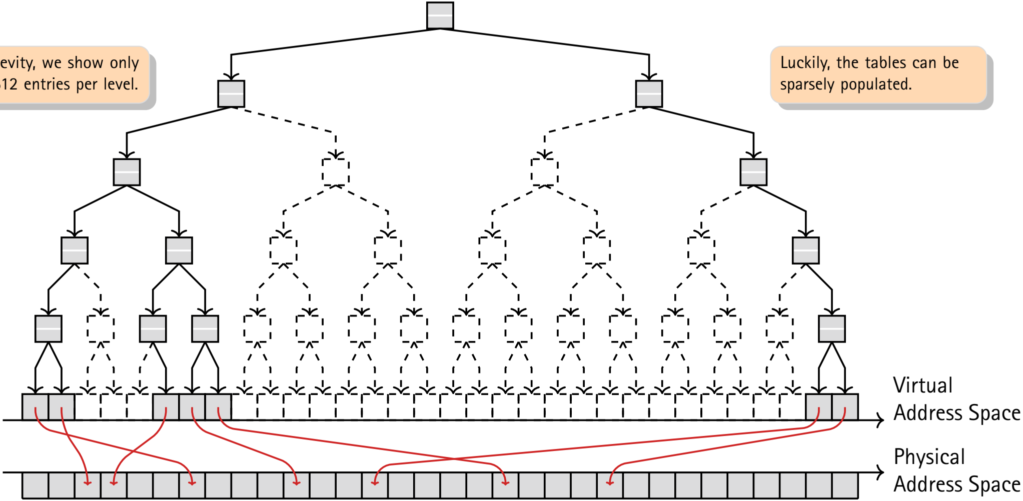
What is Needed by the HW – Intel 5-Level Paging



For brevity, we show only 2 of 512 entries per level.

128 PiB VAS would require 256 TiB of page-table data!

What is Needed by the HW – Intel 5-Level Paging



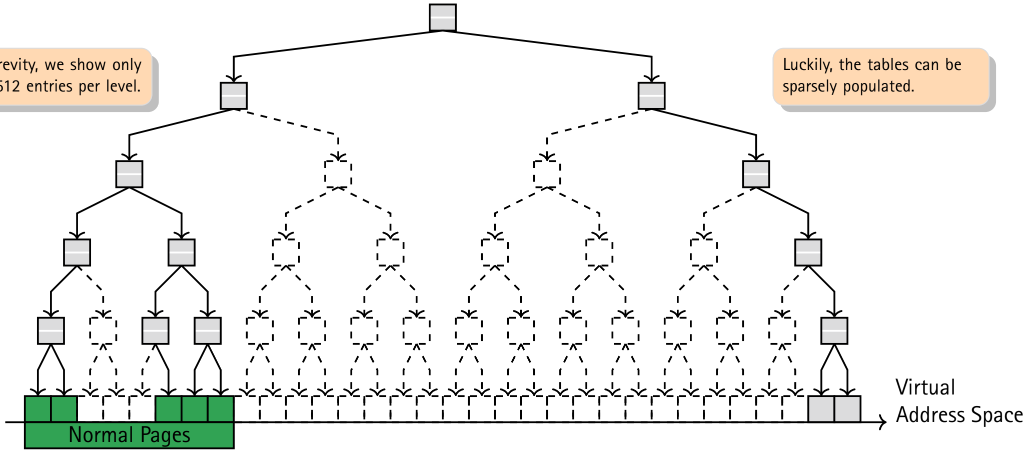
For brevity, we show only 2 of 512 entries per level.

Luckily, the tables can be sparsely populated.

What is Needed by the HW – Intel 5-Level Paging

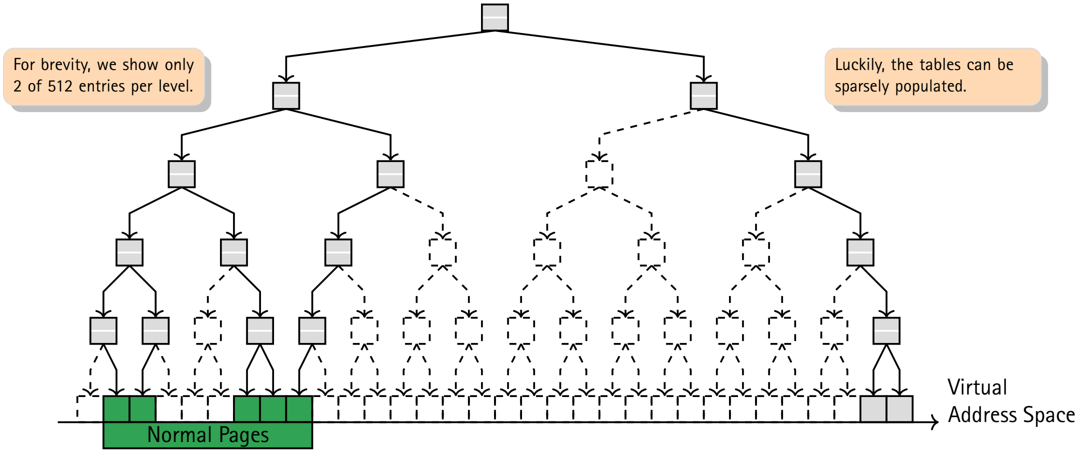
For brevity, we show only 2 of 512 entries per level.

Luckily, the tables can be sparsely populated.



- **Normal pages:** can be placed everywhere

What is Needed by the HW – Intel 5-Level Paging

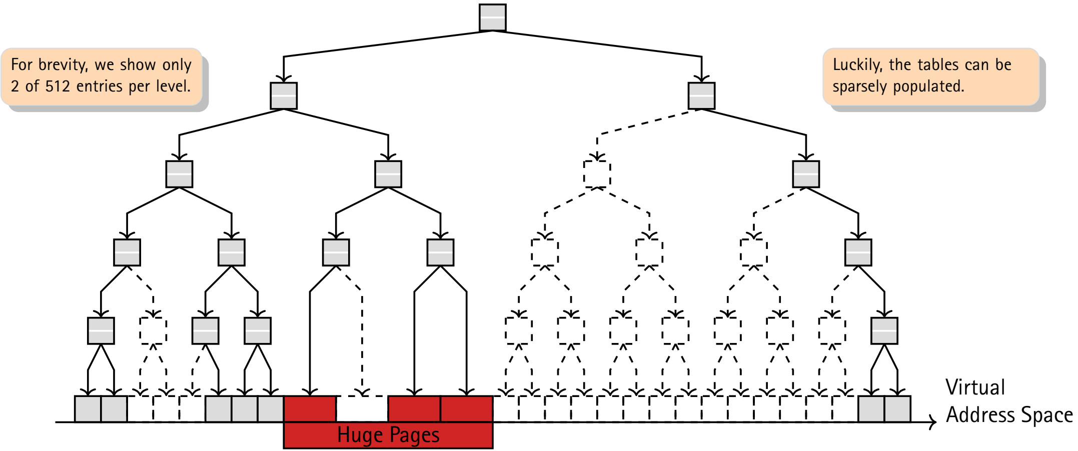


For brevity, we show only 2 of 512 entries per level.

Luckily, the tables can be sparsely populated.

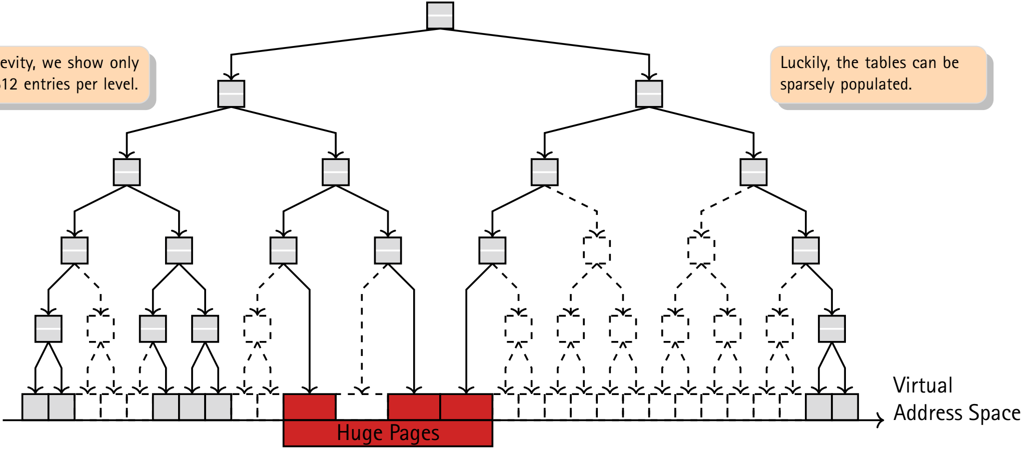
- **Normal pages:** can be placed everywhere, but the management overhead might differ.

What is Needed by the HW – Intel 5-Level Paging


For brevity, we show only 2 of 512 entries per level.
Luckily, the tables can be sparsely populated.
Huge Pages
Virtual Address Space

- **Normal pages:** can be placed everywhere, but the management overhead might differ.
- **Huge pages:** reduce table overhead and TLB pressure

What is Needed by the HW – Intel 5-Level Paging

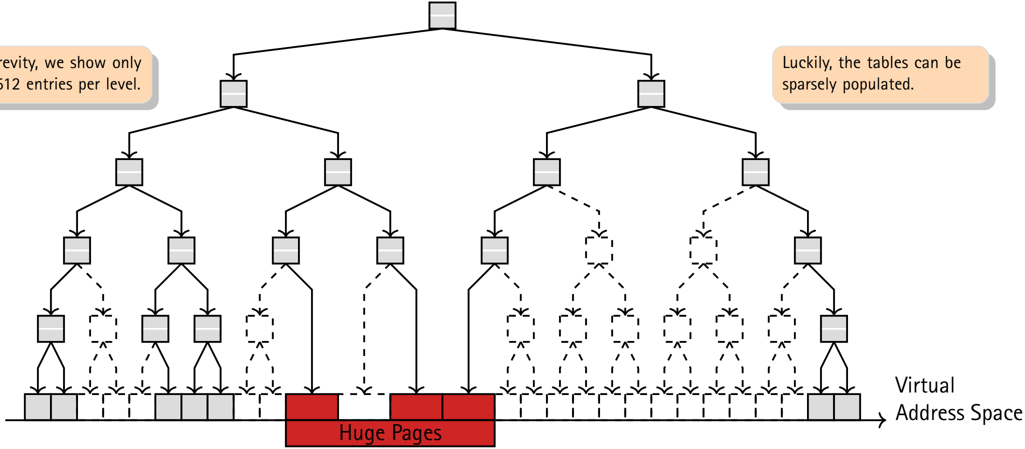


For brevity, we show only 2 of 512 entries per level.

Luckily, the tables can be sparsely populated.

- **Normal pages:** can be placed everywhere, but the management overhead might differ.
- **Huge pages:** reduce table overhead and TLB pressure, but require alignment.

Problem 2: External Fragmentation (Again)



For brevity, we show only 2 of 512 entries per level.

Luckily, the tables can be sparsely populated.

- **Normal pages:**
- **Huge pages:**

Problem: **External fragmentation is back :-(**

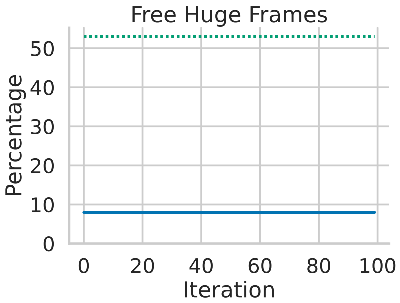
Problem 2: External Fragmentation (Again)

- Example: Linux frame allocation

Linux 6.0 on Xeon(R) Gold 5320: 2 × 26 physical cores @ 2.20 GHz, 256/512 GiB DRAM/NVRAM per node

55% of normal frames free (RND distribution).

10% are freed and reallocated per iteration.



Problem: **External fragmentation is back :-(**

Problem 2: External Fragmentation (Again)

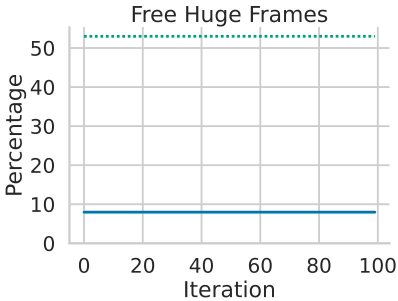
- Example: Linux frame allocation

Linux 6.0 on Xeon(R) Gold 5320: 2 × 26 physical cores @ 2.20 GHz, 256/512 GiB DRAM/NVRAM per node

55% of normal frames free (RND distribution).

10% are freed and reallocated per iteration.

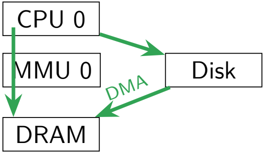
Fragmentation remains **really, really bad**.



Problem: **External fragmentation is back :-(**

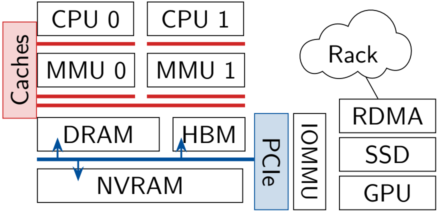
6.2 Hardware Developments

Hardware Developments: Overview



- PDP-11 Model: Single CPU is Queen of the system
  - One virtual address space, secondary storage is separate
  - Direct memory access is an optimization for I/O

Hardware Developments: Overview



- PDP-11 Model: Single CPU is Queen of the system
  - One virtual address space, secondary storage is separate
  - Direct memory access is an optimization for I/O
- Current Reality: More Memory / Users / Interconnects
  - Deep **cache hierarchy** of shared coherent CPU caches
  - **Multiple cores** with separate MMUs with non-coherent TLBs
  - **Memory Types** with different latencies and properties
  - PCIe is *the* **interconnect** standard
  - SSDs provide fast random block access
  - Remote DMA provides access to the PCIe bus
  - peripheral memory access via **IOMMU** but w/o coherency
  - Accelerators are more efficient than the CPU

Hardware Developments: Cache Hierarchy, NUMA

- The Memory hierarchy becomes deeper

  Example: Intel Xeon Gold 5320 (Random Read Access)

  – Caches 1.6 ns (L1), 6 ns (L2), 21 ns (L3)
  – RAM Local: 95 ns, Remote: 155 ns
  – Other Optane: 170-305 ns[23]
  RDMA: 600 ns (@200G)

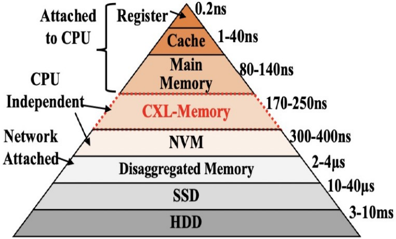
Hardware Developments: Cache Hierarchy, NUMA, and CXL!

- The Memory hierarchy becomes deeper

  Example: Intel Xeon Gold 5320 (Random Read Access)

  - Caches 1.6 ns (L1), 6 ns (L2), 21 ns (L3)
  - RAM Local: 95 ns, Remote: 155 ns
  - Other Optane: 170-305 ns[23]
    RDMA: 600 ns (@200G)

- **C**ompute e**X**press **L**ink is a PCIe Protocol
  - Use PCIe as inter-machine interconnect
  - CXL.mem: NUMA-like latencies for remote memory
  - CXL.cache: devices with cache coherency



Problem 3: The Memory Hierarchy is a Network

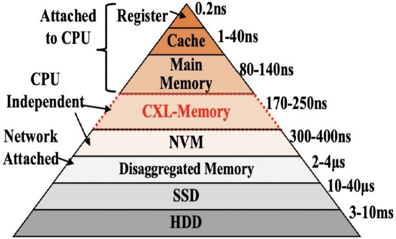
- The Memory hierarchy becomes deeper

  Example: Intel Xeon Gold 5320 (Random Read Access)

  - Caches 1.6 ns (L1), 6 ns (L2), 21 ns (L3)
  - RAM Local: 95 ns, Remote: 155 ns
  - Other Optane: 170-305 ns[23]
    RDMA: 600 ns (@200G)

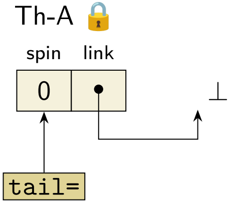
Problem: **“Your computer is already a distributed system” [3]**

- **C**ompute e**X**press **L**ink is a PCIe Protocol
  - Use PCIe as inter-machine interconnect
  - CXL.mem: NUMA-like latencies for remote memory
  - CXL.cache: devices with cache coherency



Multi-Core: NUMA-aware Spinlocks (1)

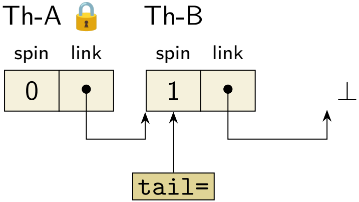
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line



  - Shared State: `tail`-pointer

Multi-Core: NUMA-aware Spinlocks (1)

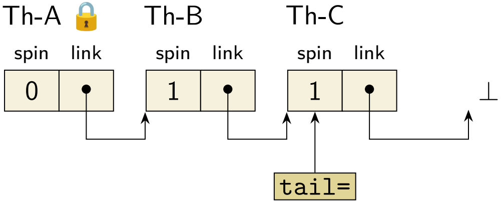
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line



  - Shared State: `tail`-pointer
  - lock(): Enqueue themselves via CAS-operation

Multi-Core: NUMA-aware Spinlocks (1)

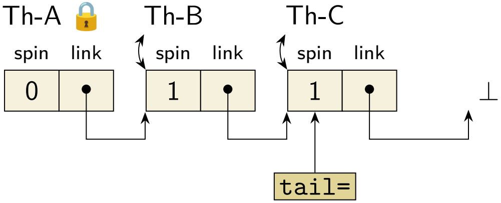
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line



  - Shared State: `tail`-pointer
  - lock(): Enqueue themselves via CAS-operation

Multi-Core: NUMA-aware Spinlocks (1)

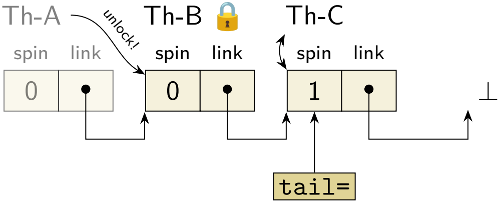
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line



- Shared State: `tail`-pointer
- lock(): Enqueue themselves via CAS-operation
- Wait: Threads poll **local** cache line

Multi-Core: NUMA-aware Spinlocks (1)

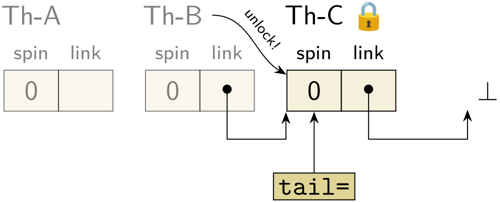
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line



- Shared State: `tail`-pointer
- lock(): Enqueue themselves via CAS-operation
- Wait: Threads poll **local** cache line
- `unlock()`: `next->spin=0`, 1 cache-line transfer

Multi-Core: NUMA-aware Spinlocks (1)

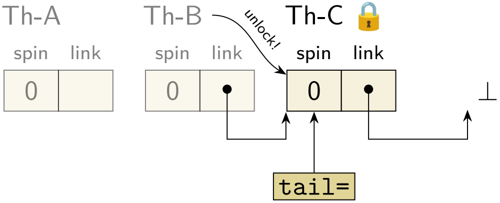
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line



  - Shared State: `tail`-pointer
  - lock(): Enqueue themselves via CAS-operation
  - Wait: Threads poll **local** cache line
  - `unlock()`: `next->spin=0`, 1 cache-line transfer

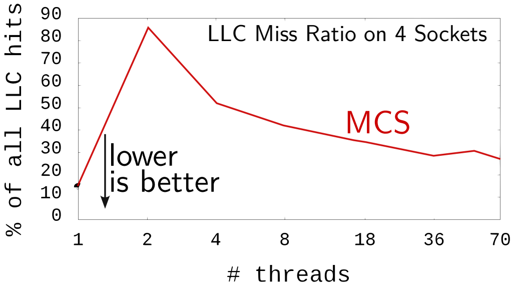
Multi-Core: NUMA-aware Spinlocks (1)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line



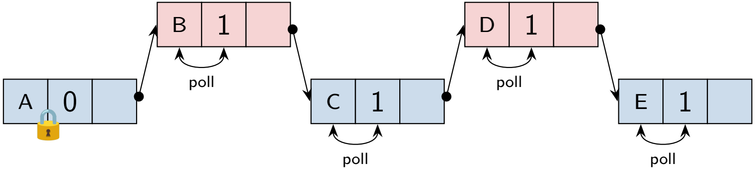
  - Shared State: `tail`-pointer
  - lock(): Enqueue themselves via CAS-operation
  - Wait: Threads poll **local** cache line
  - `unlock()`: `next->spin=0`, 1 cache-line transfer

- Traditional spinlocks are problematic on NUMA
  - Common Wisdom: „Locks should be FIFO!"
  - FIFO ensures fairness and avoids starvation
  - But: Lock-holder bounces between NUMA sockets



Multi-Core: NUMA-aware Spinlocks (2)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea**: waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line
- **CNA-Lock**: Compact NUMA-aware Spinlock [7]
  - **Idea**: prefer waiters on local NUMA node
  - Lock-holder has a queue of non-local waiters
  - Become **unfair** in favor of performance

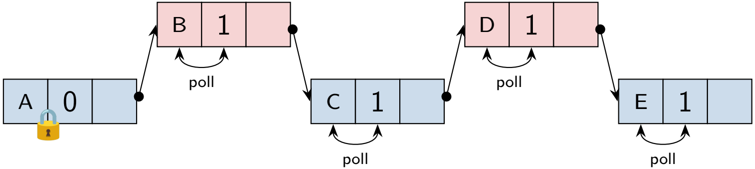


06-MemoryChallenges 2023-04-04

Multi-Core: NUMA-aware Spinlocks (2)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea**: waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line

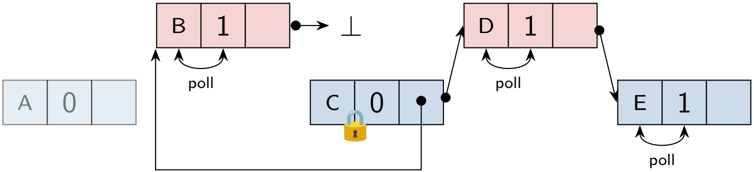
- **CNA-Lock**: Compact NUMA-aware Spinlock [7]
  - **Idea**: prefer waiters on local NUMA node
  - Lock-holder has a queue of non-local waiters
  - Become **unfair** in favor of performance



- Enqueue works like MCS lock

Multi-Core: NUMA-aware Spinlocks (2)

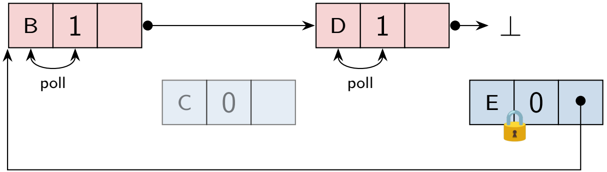
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea**: waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line
- **CNA-Lock**: Compact NUMA-aware Spinlock [7]
  - **Idea**: prefer waiters on local NUMA node
  - Lock-holder has a queue of non-local waiters
  - Become **unfair** in favor of performance



- Enqueue works like MCS lock
- `unlock()` move remote waiters into 2nd queue

Multi-Core: NUMA-aware Spinlocks (2)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea**: waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line
- **CNA-Lock**: Compact NUMA-aware Spinlock [7]
  - **Idea**: prefer waiters on local NUMA node
  - Lock-holder has a queue of non-local waiters
  - Become **unfair** in favor of performance

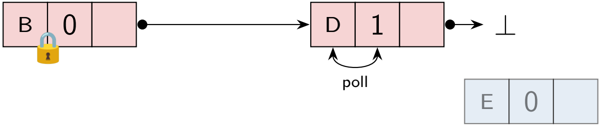


- Enqueue works like MCS lock
- `unlock()` move remote waiters into 2nd queue
- Secondary queue is passed on

Multi-Core: NUMA-aware Spinlocks (2)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea**: waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line

- **CNA-Lock**: Compact NUMA-aware Spinlock [7]
  - **Idea**: prefer waiters on local NUMA node
  - Lock-holder has a queue of non-local waiters
  - Become **unfair** in favor of performance



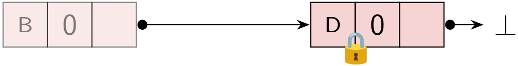
- Enqueue works like MCS lock
- `unlock()` move remote waiters into 2nd queue

- Secondary queue is passed on
- No local waiters, switch NUMA node

Multi-Core: NUMA-aware Spinlocks (2)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea**: waiter queue, local spinning
  - Standard lock for Linux (replaced test-and-test)
  - Everybody spins on its own cache line

- **CNA-Lock**: Compact NUMA-aware Spinlock [7]
  - **Idea**: prefer waiters on local NUMA node
  - Lock-holder has a queue of non-local waiters
  - Become **unfair** in favor of performance

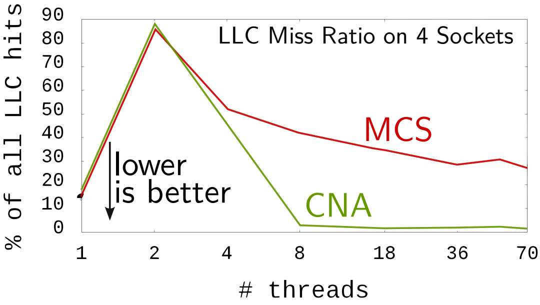


- Enqueue works like MCS lock
- `unlock()` move remote waiters into 2nd queue

- Secondary queue is passed on
- No local waiters, switch NUMA node

Multi-Core: NUMA-aware Spinlocks (3)

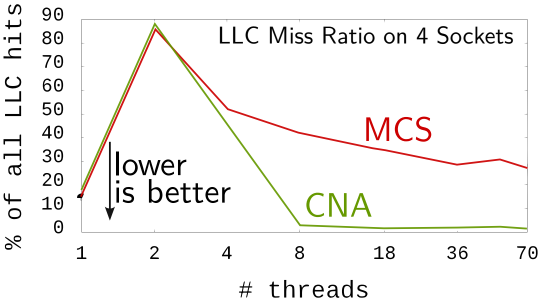
- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
- **CNA-Lock**: Compact NUMA-aware Spinlock
  - **Idea**: prefer waiters on local NUMA node



Multi-Core: NUMA-aware Spinlocks (3)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning
- **CNA-Lock**: Compact NUMA-aware Spinlock
  - **Idea**: prefer waiters on local NUMA node

Both Locks solve Memory Problems!

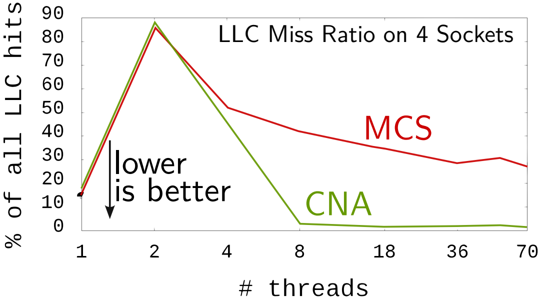


Multi-Core: NUMA-aware Spinlocks (3)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning

- **CNA-Lock**: Compact NUMA-aware Spinlock
  - **Idea**: prefer waiters on local NUMA node

Both Locks solve Memory Problems!



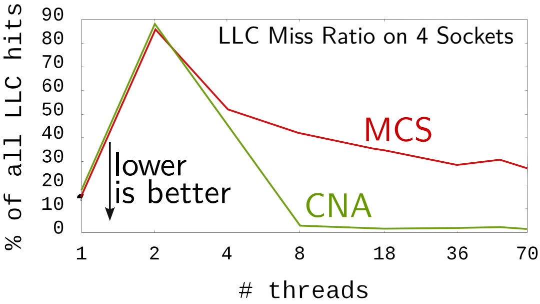
- „Thundering-Herd Problem"
  - TAS: Invalidate shared cache line ⇒ (n-1) misses
  - MCS: Unlock provokes exactly one cache miss
  - *Principle:* Shared memory is 1-to-*N* communication
    Keep *N* small!

Multi-Core: NUMA-aware Spinlocks (3)

- **MCS-Lock**: A Fair, NUMA-oblivious Spinlock
  - **Idea:** waiter queue, local spinning

- **CNA-Lock**: Compact NUMA-aware Spinlock
  - **Idea**: prefer waiters on local NUMA node

Both Locks solve Memory Problems!



- „Thundering-Herd Problem“
  - TAS: Invalidate shared cache line ⇒ (n-1) misses
  - MCS: Unlock provokes exactly one cache miss
  - *Principle:* Shared memory is 1-to-*N* communication
    Keep *N* small!
- NUMA-Aware Programming
  - MCS: Protected state bounces between sockets
  - CNA: Lock sticks to NUMA socket
  - *Principle*: Keep control flow where the **cached** data

Hardware Developments (2)

	Sun 33 (1990)	Xeon 5320 (2022)	Factor
CPU	1× @ 33 Mhz, 10–11 MIPS	2×28 @ 2–3 GHz, 100k MIPS	1000
TLB/Thr.	64 Entries	132 L1 + 1500 L2	25
L1D: Size	256 B	64 KiB	256
Latency[1]	180ns	1 ns	180
RAM: Size	≤ 128 MiB	≤ 3 TiB	25000
Latency[1]	210 ns	100 ns	2
Read (1 MiB) [1]	3200 us	3 us	1000
Bandwidth	200 MiB/s	120 GiB/s [20]	600
Network (Read 2 KiB)[1]	1448 us	16 ns	90500
Disk (Read 1 MiB)[1]	640 ms	825 us / 125 us (SSD)	775 / 5000

[1] Typical from `https://colin-scott.github.io/personal_website/research/interactive_latency.html`

Problem 4: Designed for Scarcity, Not for Latency

	Sun 33 (1990)	Xeon 5320 (2022)	Factor
CPU	1× @ 33 Mhz, 10–11 MIPS	2×28 @ 2–3 GHz, 100k MIPS	1000
TLB/Thr.	64 Entries	132 L1 + 1500 L2	25
L1D: Size	256 B	64 KiB	256
Latency[1]	180ns	1 ns	180
RAM: Size	≤ 128 MiB	≤ 3 TiB	25000
Latency[1]	210 ns	100 ns	2
Read (1 MiB) [1]	3200 us	3 us	1000
Bandwidth	200 MiB/s	120 GiB/s [20]	600
Network (Read 2 KiB)[1]	1448 us	16 ns	90500
Disk (Read 1 MiB)[1]	640 ms	825 us / 125 us (SSD)	775 / 5000

[1] Typical from `https://colin-scott.github.io/personal_website/research/interactive_latency.html`

Problem: Memory has become abundant, but latencies and TLB are killers!

The Physical Memory (PM) is Huge

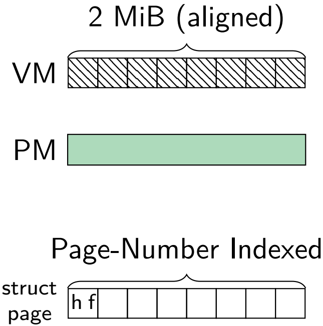
- The physical memory is 25k× larger!
  - 1 Gib $\triangleq$ 512 huge frames $\triangleq$ 262K frames
  - The Sun 33 (1990) had 32K frames
- **Challenge:** Meta-Data Overhead
  - `struct page` stores 64 B metadata per frame
    1 GiB $\triangleq$ 16 MiB of meta-data
  - Linux spends 1.56 % of its DRAM for this!

The Physical Memory (PM) is Huge

- The physical memory is 25k× larger!
  - 1 Gib $\triangleq$ 512 huge frames $\triangleq$ 262K frames
  - The Sun 33 (1990) had 32K frames
- **Challenge:** Meta-Data Overhead
  - `struct page` stores 64 B metadata per frame
    1 GiB $\triangleq$ 16 MiB of meta-data
  - Linux spends 1.56 % of its DRAM for this!

- Multiple Frame Sizes
  - Huge frames extend the **TLB reach**.
  - Using huge frames save on page tables.
- **Challenge**: Allocation Policy
  - When to allocate which granularity?
  - Huge frames are worse for Copy-on-Write
  - Support for existing software!

Transparent Huge Pages



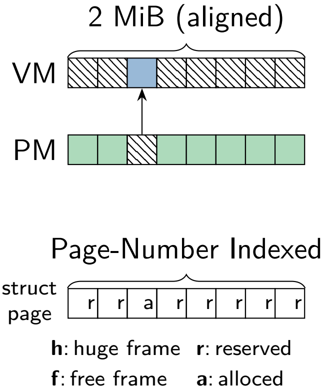
h: huge frame

f: free frame

- Should the OS map 4 KiB Frame or 2 MiB Frame?
  - + 4 KiB: Less memory, faster copy (CoW)
  - + 2 MiB: TLB pressure, less faults

Break even: 70 4 KiB pages

Transparent Huge Pages

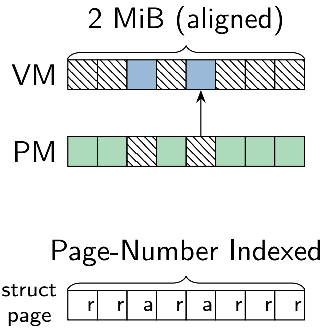


- Should the OS map 4 KiB Frame or 2 MiB Frame?
  - + 4 KiB: Less memory, faster copy (CoW)
  - + 2 MiB: TLB pressure, less faults

Break even:
70 4 KiB pages

- **Transparent Huge Pages** [15, 17]: Why not both?
  - Idea: Start with 4 KiB and upgrade to 2 MiB lateron.
  - First fault: reserve 2 MiB but map only 4 KiB

Transparent Huge Pages



r	r	a	r	a	r	r	r

h: huge frame **r**: reserved

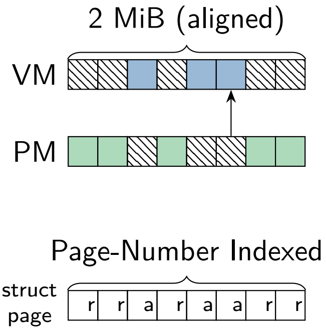
f: free frame **a**: alloced

- Should the OS map 4 KiB Frame or 2 MiB Frame?
  - + 4 KiB: Less memory, faster copy (CoW)
  - + 2 MiB: TLB pressure, less faults

Break even: 70 4 KiB pages

- **Transparent Huge Pages** [15, 17]: Why not both?
  - Idea: Start with 4 KiB and upgrade to 2 MiB lateron.
  - First fault: reserve 2 MiB but map only 4 KiB
  - Individual faults up to a threshold (e.g., $<50\,\%$)

Transparent Huge Pages



2 MiB (aligned)

VM

PM

Page-Number Indexed

struct page

r	r	a	r	a	a	r	r

h: huge frame **r**: reserved

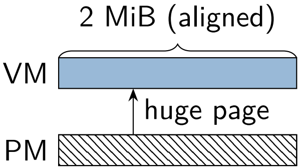
f: free frame **a**: alloced

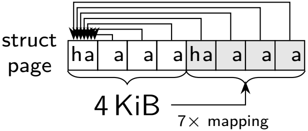
- Should the OS map 4 KiB Frame or 2 MiB Frame?
  - + 4 KiB: Less memory, faster copy (CoW)
  - + 2 MiB: TLB pressure, less faults

Break even: 70 4 KiB pages

- **Transparent Huge Pages** [15, 17]: Why not both?
  - Idea: Start with 4 KiB and upgrade to 2 MiB lateron.
  - First fault: reserve 2 MiB but map only 4 KiB
  - Individual faults up to a threshold (e.g., $<50\,\%$)

Transparent Huge Pages



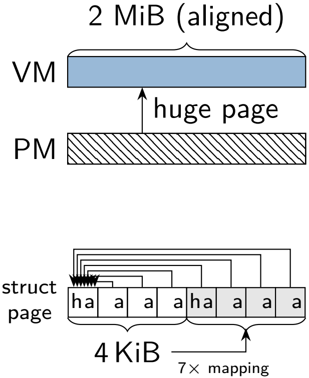


- Should the OS map 4 KiB Frame or 2 MiB Frame?
  - + 4 KiB: Less memory, faster copy (CoW)
  - + 2 MiB: TLB pressure, less faults

Break even: 70 4 KiB pages

- **Transparent Huge Pages** [15, 17]: Why not both?
  - Idea: Start with 4 KiB and upgrade to 2 MiB lateron.
  - First fault: reserve 2 MiB but map only 4 KiB
  - Individual faults up to a threshold (e.g., $<$50 %)
  - Upgrade to 2 MiB Mapping

Transparent Huge Pages



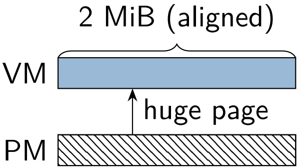
- Should the OS map 4 KiB Frame or 2 MiB Frame?
  - + 4 KiB: Less memory, faster copy (CoW)
  - + 2 MiB: TLB pressure, less faults

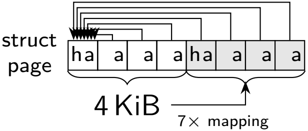
Break even: 70 4 KiB pages

- **Transparent Huge Pages** [15, 17]: Why not both?
  - Idea: Start with 4 KiB and upgrade to 2 MiB lateron.
  - First fault: reserve 2 MiB but map only 4 KiB
  - Individual faults up to a threshold (e.g., $<$50 %)
  - Upgrade to 2 MiB Mapping
- **Linux:** `struct page`-Array for 2 MiB Mappings
  - 512 × `struct page` (64b) $\triangleq$ 32 KiB $\triangleq$ 8 frames 😱
  - Idea: Map the first frame 7 more times
  - Save 28 KiB per 2 MiB mapping (1.36% of all DRAM!)

Transparent Huge Pages

Problem: This is a Memory-Scarce Design!





- Should the OS map 4 KiB Frame or 2 MiB Frame?
  - + 4 KiB: Less memory, faster copy (CoW)
  - + 2 MiB: TLB pressure, less faults

Break even: 70 4 KiB pages

- **Transparent Huge Pages** [15, 17]: Why not both?
  - Idea: Start with 4 KiB and upgrade to 2 MiB lateron.
  - First fault: reserve 2 MiB but map only 4 KiB
  - Individual faults up to a threshold (e.g., $<$50 %)
  - Upgrade to 2 MiB Mapping

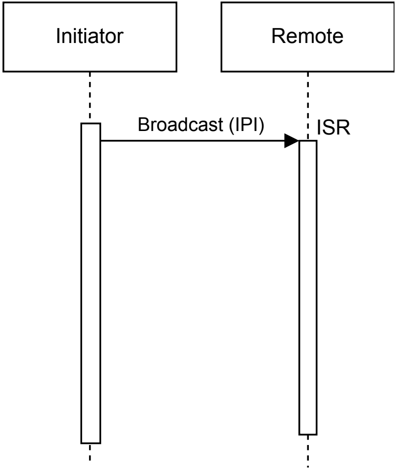
- **Linux:** `struct page`-Array for 2 MiB Mappings
  - 512 × `struct page` (64b) $\triangleq$ 32 KiB $\triangleq$ 8 frames 😱
  - Idea: Map the first frame 7 more times
  - Save 28 KiB per 2 MiB mapping (1.36% of all DRAM!)

Translation Look-Aside Buffer

- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores!
    Optimized variant [2]: 3400 – 4300 cycles

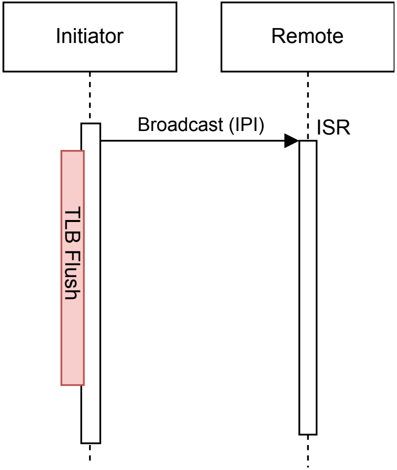
Translation Look-Aside Buffer

- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores!
    Optimized variant [2]: 3400 – 4300 cycles



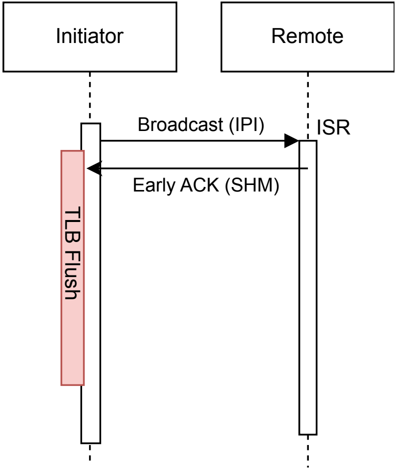
Translation Look-Aside Buffer

- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores!
    Optimized variant [2]: 3400 – 4300 cycles



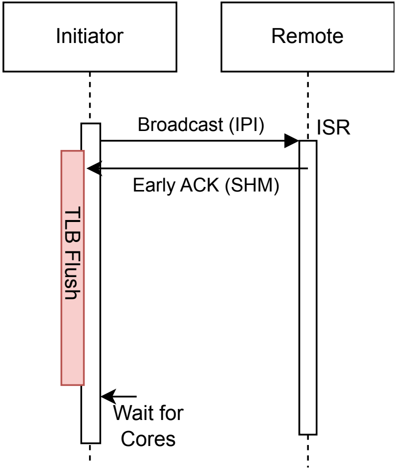
Translation Look-Aside Buffer

- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores!
    Optimized variant [2]: 3400 – 4300 cycles



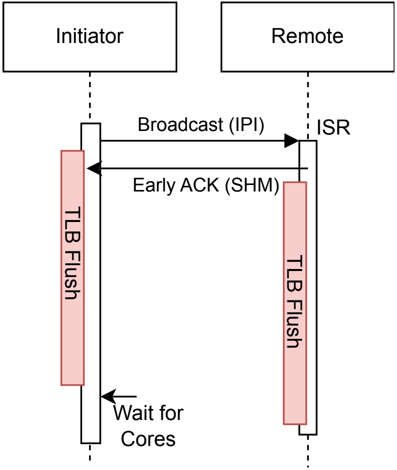
Translation Look-Aside Buffer

- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores!
    Optimized variant [2]: 3400 – 4300 cycles



Translation Look-Aside Buffer

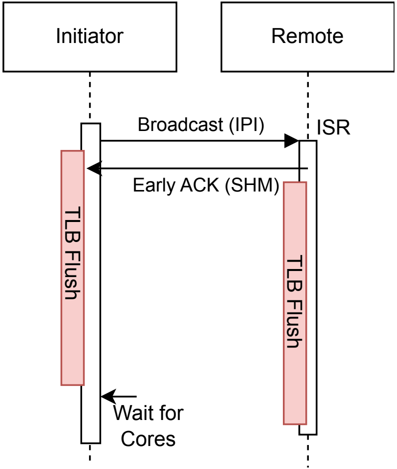
- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores!
    Optimized variant [2]: 3400 – 4300 cycles



Translation Look-Aside Buffer

- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores! Optimized variant [2]: 3400 – 4300 cycles

- *Principle*: Shootdown should be a rare event
  - *Batching*: Combine multiple independent shootdowns
  - *Semantics*: Avoid shootdowns by weakening guarantees
  - Both are problematic with existing software
  - Hard to implement them correct

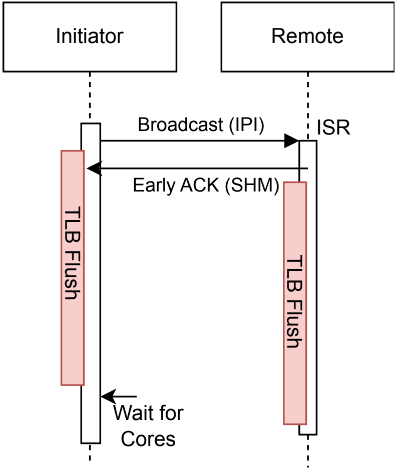


Problem 5: Software must fix Broken Hardware

- TLB: The Last Non-Coherent Cache
  - Each CPU caches the slow page-table walk
  - **Huge Impact** (5-Levels): 600 ns vs 1 ns
  - The OS must invalidate entries on remote cores!
    Optimized variant [2]: 3400 – 4300 cycles

- *Principle*: Shootdown should be a rare event
  - *Batching*: Combine multiple independent shootdowns
  - *Semantics*: Avoid shootdowns by weakening guarantees
  - Both are problematic with existing software
  - Hard to implement them correct

Problem: Fixing Hardware in Software



Hardware Development: Secondary Storage

Medium	Capacity	Sequential Read	4K IOP/S	€ / 1 TB
Seagate Savvio 15K.2 (HDD, 2009)	146 GB	160 MB/s	204	2 000 €
Seagate Exos 2x14 (HDD, 2021)	14 TB	524 MB/s	304	27 €
Intel X25-E (SSD, 2009)	32 GB	250 MB/s	35 000	21 800 €
Samsung PM1735 (SSD, 2019)	12.8 TB	8000 MB/s	1 500 000	340 €

- SSDs will replace HDDs
  - SSDs are large and cheap (enough).
  - Small penalty for random (PM1735: 6 GiB/s)
  - Multi-million IOP/s **if** queues are deep enough

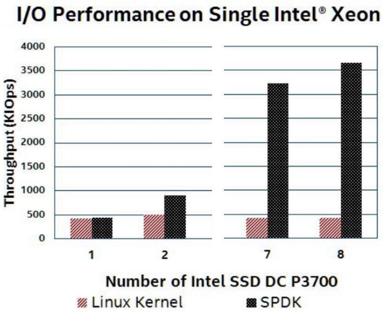
Problem 6: Designed for Slow and Sequential I/O

Medium	Capacity	Sequential Read	4K IOP/S	€ / 1 TB
Seagate Savvio 15K.2 (HDD, 2009)	146 GB	160 MB/s	204	2 000 €
Seagate Exos 2x14 (HDD, 2021)	14 TB	524 MB/s	304	27 €
Intel X25-E (SSD, 2009)	32 GB	250 MB/s	35 000	21 800 €
Samsung PM1735 (SSD, 2019)	12.8 TB	8000 MB/s	1 500 000	340 €

- SSDs will replace HDDs
  - SSDs are large and cheap (enough).
  - Small penalty for random (PM1735: 6 GiB/s)
  - Multi-million IOP/s **if** queues are deep enough

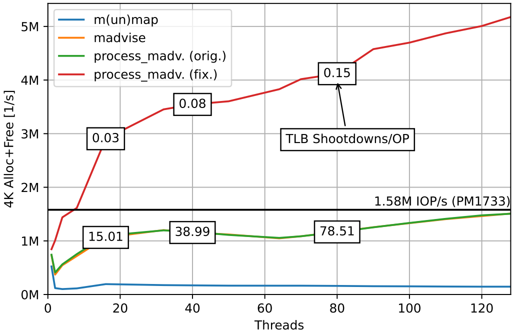
Problem: Designed for Slow I/O

- „Nothing matters if you have to query the disk.“
- A page fault provokes only one small disk read.



6.3 ParPerOS – Contention-Avoiding Design

Linux: Virtual Memory Page Allocation

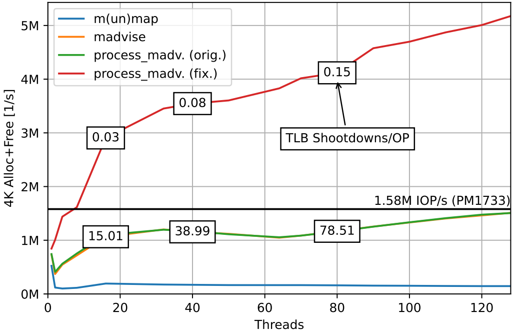


Linux 5.16, AMD EPYC 7713 processor (64 cores, 128 hardware threads), 512 GB RAM

- **Benchmark:** Alloc/Free 4 KiB Pages Randomly
  - Random I/O requires random VM operations
  - Allocate page via page fault
  - Free page via `MADV_DONTNEED` or `munmap()`

- `munmap(2)`
  - Modifies the **global** memory-object list.
  - Memory objects are split and merged
- `madvise(2)`
  - Modify only the page tables
  - One TLB Shootdown per eviction!
- `process_madvise(2)`
  - Vectorized `madvise(2)`
  - One TLB shootdown per 512 pages.

Linux: Virtual Memory Page Allocation



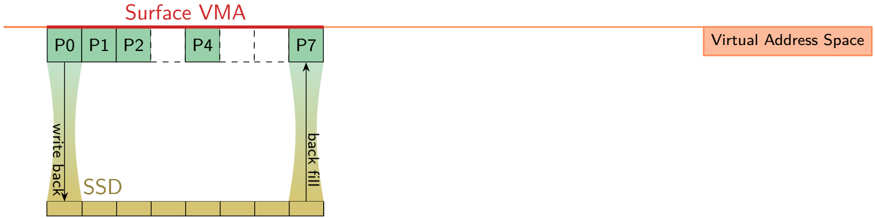
Linux 5.16, AMD EPYC 7713 processor (64 cores, 128 hardware threads), 512 GB RAM

Problem: Designed for Slow I/O

- **Benchmark:** Alloc/Free 4 KiB Pages Randomly
  - Random I/O requires random VM operations
  - Allocate page via page fault
  - Free page via `MADV_DONTNEED` or `munmap()`
- `munmap(2)`
  - Modifies the **global** memory-object list.
  - Memory objects are split and merged
- `madvise(2)`
  - Modify only the page tables
  - One TLB Shootdown per eviction!
- `process_madvise(2)`
  - Vectorized `madvise(2)`
  - One TLB shootdown per 512 pages.

ExMap: File-Mapped IO for High-Performance SSDs

[16]



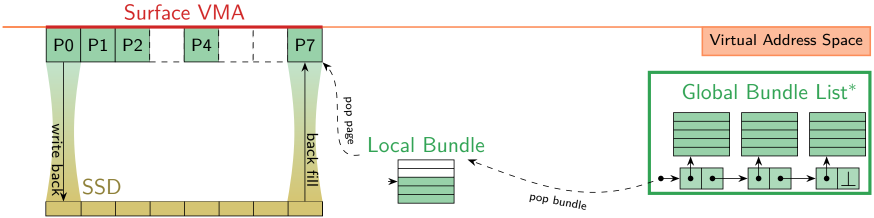
Explicit File-Mapped I/O

- No page-faults, no automatic write-back
- **Vectorized** surface operations (alloc/free)
- Lock-free page-table modifications

Principles

- Forbid slow-I/O paths
- Vectorized operations
- Use CPU atomics

ExMap: File-Mapped IO for High-Performance SSDs [16]



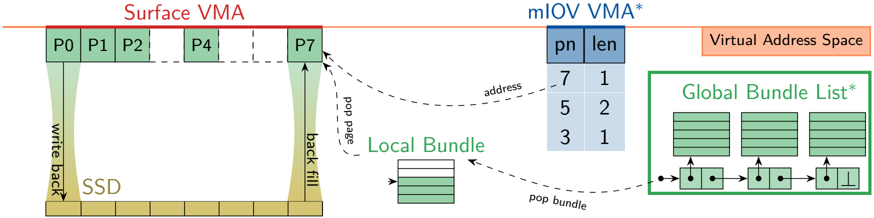
Process-Local Frame Pool

- Avoids zeroing without leak
- Lock-free global bundle list
- CPU-local bundles (513 frames)

Principles

- Memory-abundant design!
- Limited global communication
- Cache-friendly data structures

ExMap: File-Mapped IO for High-Performance SSDs [16]



pn	len
7	1
5	2
3	1

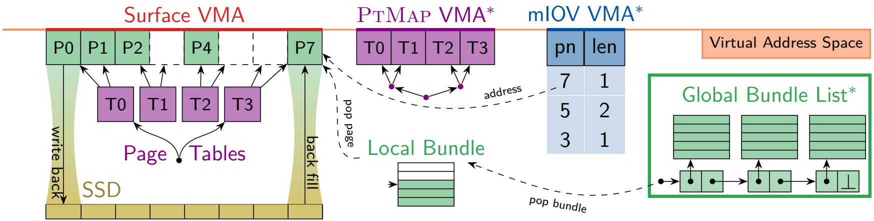
Memory-Mapped IO Vector

- Pre-mapped parameter vector
- Page number (52 bits), length (12 bits)
- Avoids `copy_from_user()` checks

Principles

- Re-use loaded cache lines
- Dense special-purpose encoding
- Memory as communication interface

ExMap: File-Mapped IO for High-Performance SSDs [16]



pn	len
7	1
5	2
3	1

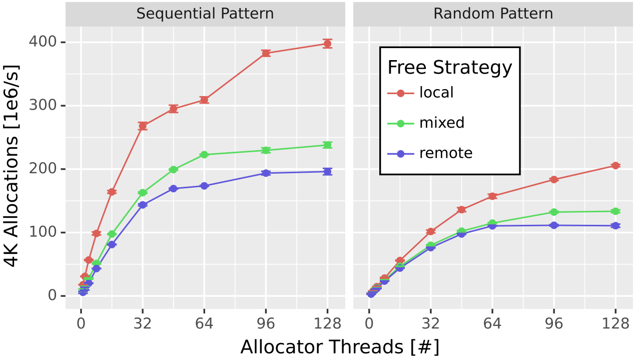
Exported Page Tables

- Read-only mapping
- In-core information
- Cache line is also used by MMU

Principles

- Expose hardware specifics
- Controlled isolation violations
- Re-use loaded cache lines

ExMap: Virtual Memory Allocation Performance

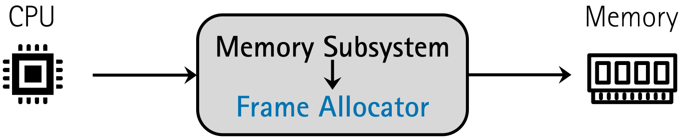


ExMap: 100M – 200M Random allocations per second

Linux: 5M Random allocations per second (with fixes)

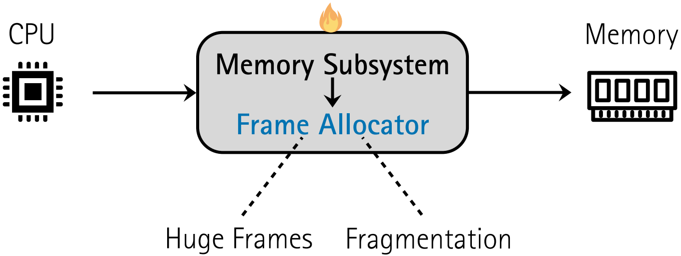
Challenge: Contention in the Memory Subsystem

Contention

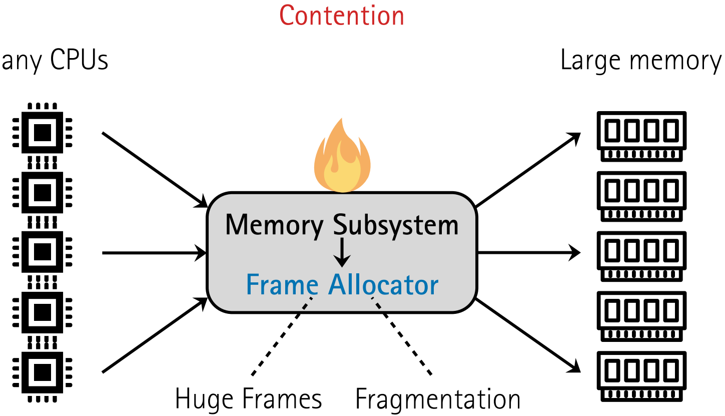


Challenge: Contention in the Memory Subsystem

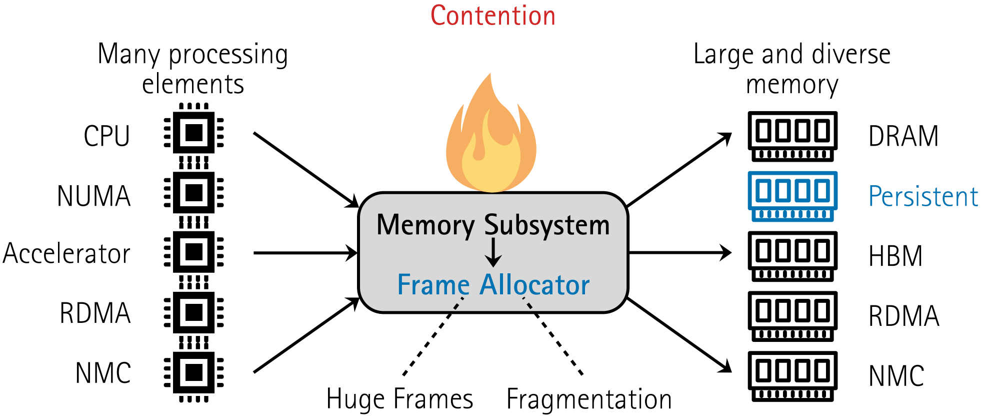
Contention



Challenge: Contention in the Memory Subsystem

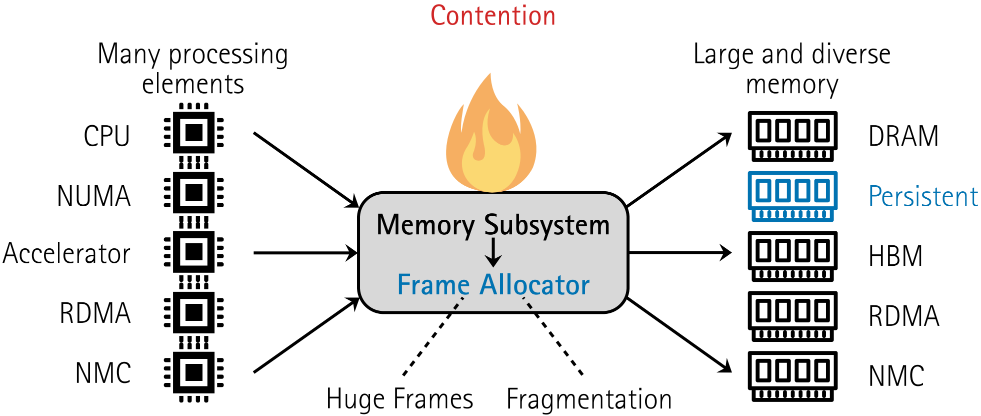


Challenge: Contention in the Memory Subsystem



06-MemoryChallenges 2023-04-04

Challenge: Contention in the Memory Subsystem



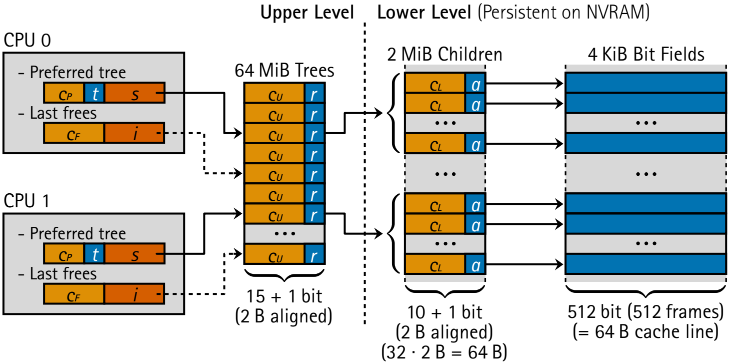
↪ Crucial to **avoid contention** from the very beginning!

How to *Avoid* Contention?

Principles

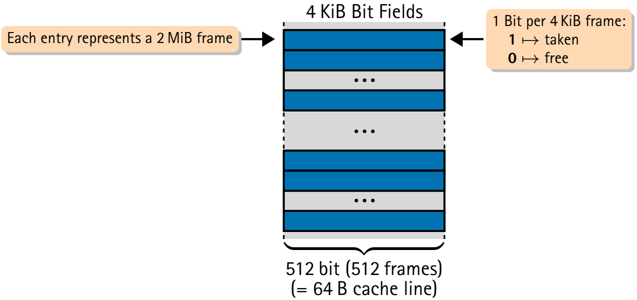
- Do not use locks. Use atomics.
  - CAS, FAA, LL/SC, ...
  - This has *a lot* of implications on data structures.
  - And even more on NVM. [5, 10, 13, 14, 21]
- Respect your hardware. Especially the cache.
  - Well known, but still ignored. [8, 12, 22]
  - Performance is dominated by the number *n* of **cache lines accessed**: $\mathbf{cla} = \mathbf{n}$
  - And even more, if cache lines are shared!
- Avoid true and false sharing. Partition your ressources.
  - Cache trashing is a major bottleneck. [4]
  - Global resource pools require locks.

LLFree – A Fast and Optionally Persistent Frame Allocator



- **Goal:** Efficient management of physical memory.
  - De/allocation of normal (4 KiB) and huge (2 MiB) frames.
- **Goal:** Optional crash consistency on NVRAM.
  - Allocation state survives sudden power loss.

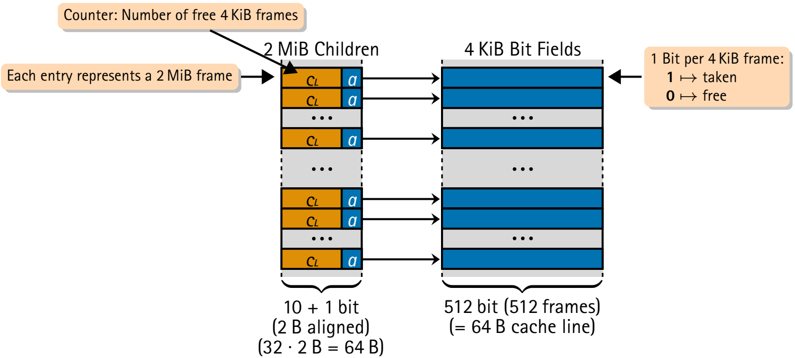
LLFree – Architecture



- **Cache-friendly design:** 512 normal frames are managed within a single cache line.
  - 4 KiB alloc: find first 0-bit in entry, set it to 1
    ⇝ very fast, if there is a 0-bit

$cla = 1$

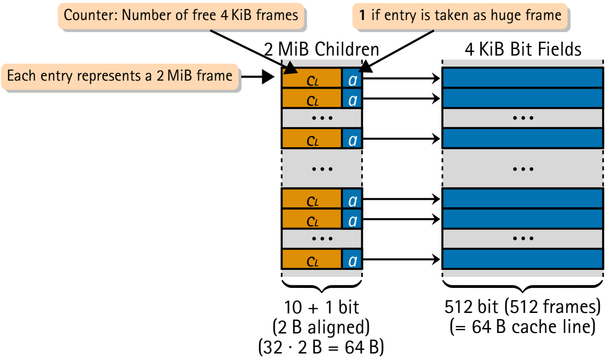
LLFree – Architecture



- **Cache-friendly design:** 512 normal frames are managed within a single cache line.
  - 4 KiB alloc: find entry with $c_L > 0$, decrement c_L, find first 0-bit in entry, set it to 1
    $\leadsto$ there is a 0 bit

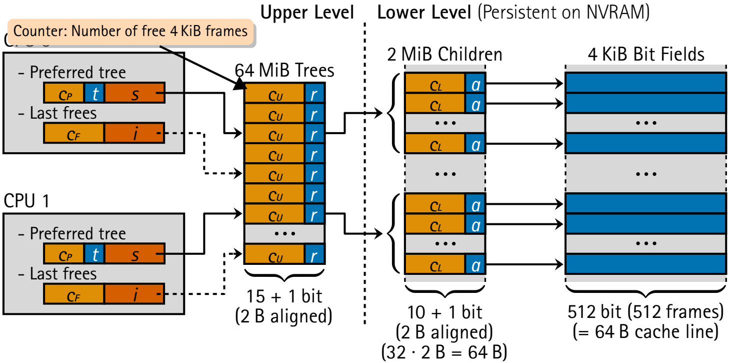
$cla = 2$

LLFree – Architecture



- **Cache-friendly design:** 512 normal frames are managed within a single cache line.
  - 4 KiB alloc: find entry with $c_L > 0$, decrement c_L, find first 0-bit in entry, set it to 1 $\rightsquigarrow$ there is a 0 bit $cla = 2$
  - 2 MiB alloc: find entry with 512 free frames, set $c_L = 0$ and $a = 1$ $\rightsquigarrow$ ignore bit field $cla = 1$

LLFree – Architecture



Counter: Number of free 4 KiB frames

CPU 0

- Preferred tree

c_P t s

- Last frees

c_F i

CPU 1

- Preferred tree

c_P t s

- Last frees

c_F i

Upper Level

64 MiB Trees

c_U r

15 + 1 bit
(2 B aligned)

Lower Level (Persistent on NVRAM)

2 MiB Children

c_L a

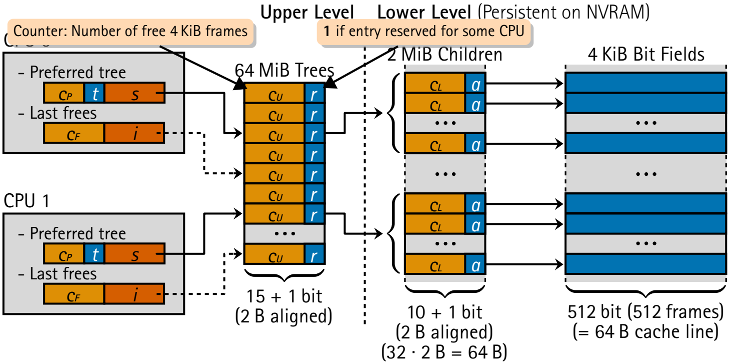
10 + 1 bit
(2 B aligned)
(32 · 2 B = 64 B)

4 KiB Bit Fields

512 bit (512 frames)
(= 64 B cache line)

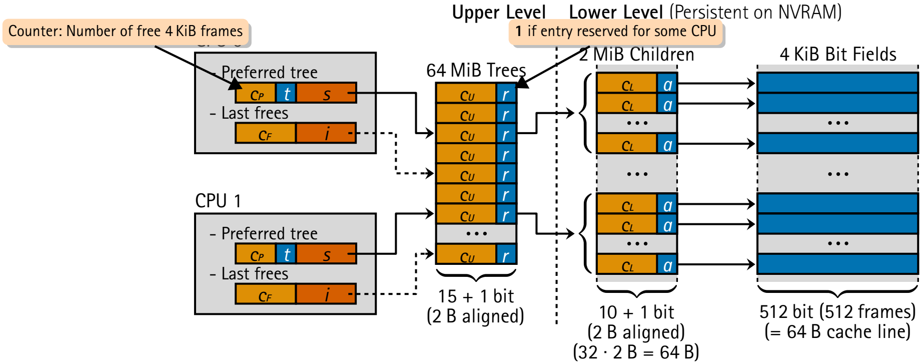
- **Cache-friendly design:** 512 normal frames are managed within a single cache line.
  - 4 KiB alloc: find entry with $c_L > 0$, decrement c_L and c_U, find first 0-bit in entry, set it to 1
    $\rightsquigarrow$ there is a 0 bit $cla = 3$
  - 2 MiB alloc: find entry with 512 free frames, set $c_L = 0$ and $a = 1$, decrement c_U
    $\rightsquigarrow$ ignore bit field $cla = 2$

LLFree – Architecture



- **Avoid false sharing:** per-CPU partitioning into 64 MiB chunks (*Trees*).
  - Lower level: No contention on cache managing children array entries (32 fit into one cache line)
  - Upper Level: Contention on cache managing trees array entries (32 fit into one cache line)

LLFree – Architecture


Counter: Number of free 4 KiB frames

1 if entry reserved for some CPU

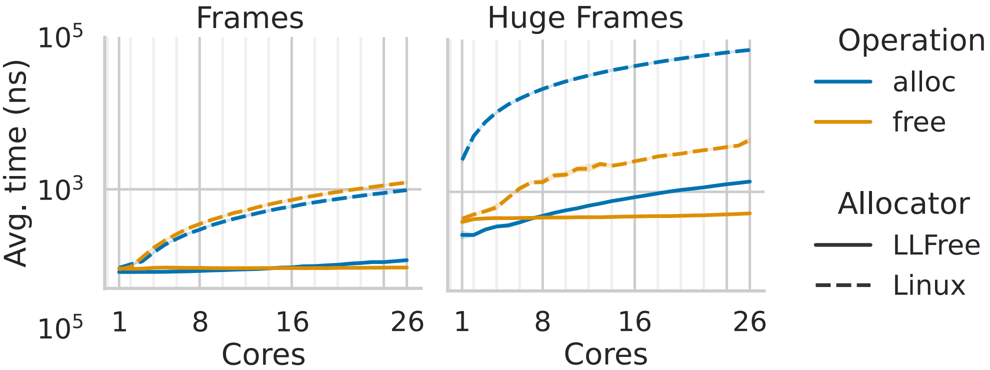
- **Avoid false sharing:** per-CPU partitioning into 64 MiB chunks (*Trees*).
  - Lower level: No contention on cache managing children array entries (32 fit into one cache line)
  - Upper Level: Contention on cache managing trees array entries (32 fit into one cache line)

    ⇝ Split counter to maintain free-frame count mostly locally: ($c_P + c_U \leq 512 \cdot 32 = 16384$)

LLFree – Scalability (DRAM)

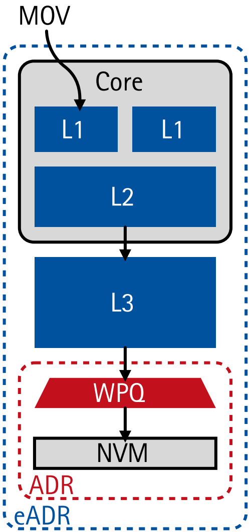
- Linux frame allocation

Linux 6.0 on Xeon(R) Gold 5320: 2 × 26 physical cores @ 2.20 GHz, 256/512 GiB DRAM/NVRAM per node



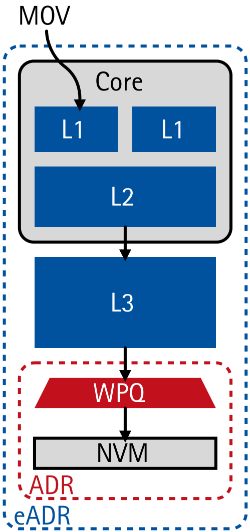
Visibility ≠ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!



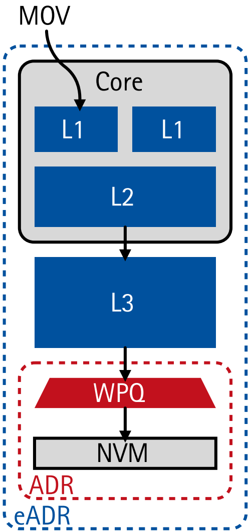
Visibility ≠ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when ...



Visibility ≠ Persistency

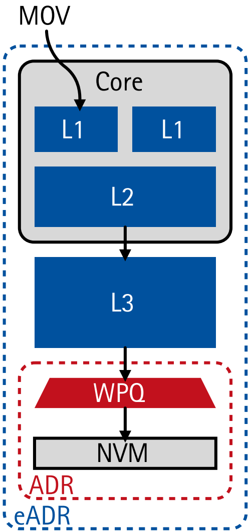
- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when ... it depends



Visibility $\neq$ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when … it depends

$\hookrightarrow$ **eADR:** Change has reached the L1 $\mapsto$ **Visibility $=$ Persistency** 😊

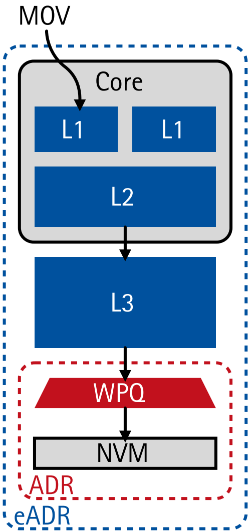


Visibility $\neq$ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when ... it depends

↪ **eADR:** Change has reached the L1 ↦ **Visibility = Persistency** 😊

- **Great concept!**

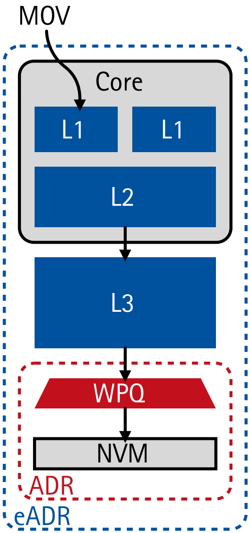


Visibility $\neq$ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when … it depends

$\hookrightarrow$ **eADR:** Change has reached the L1 $\mapsto$ **Visibility** $=$ **Persistency** ☹

- **Great concept!** … that unfortunately did not made it to market



Visibility ≠ Persistency

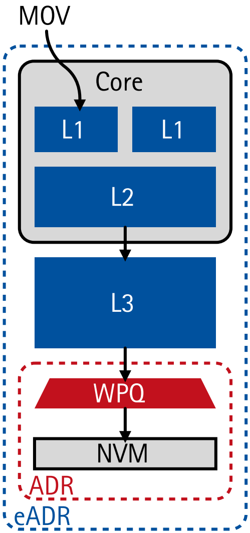
- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when ... it depends

↪ **eADR:** Change has reached the L1 ↦ **Visibility = Persistency** ☹

- **Great concept!** ... that unfortunately did not made it to market

↪ **ADR:** Changed cache line has *eventually* reached the WPQ ☹

- Ensuring durability requires expensive explicit flushes
- **Truely awfull** programming model, especially on multi-core!



Visibility ≠ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when ... it depends

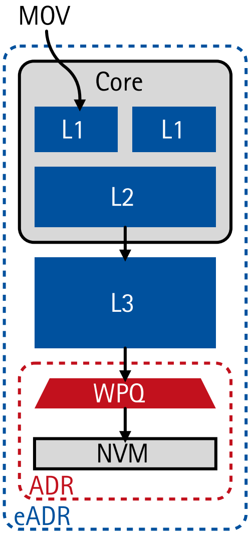
↪ **eADR:** Change has reached the L1 ↦ **Visibility = Persistency** ☹

- **Great concept!** ... that unfortunately did not made it to market

↪ **ADR:** Changed cache line has *eventually* reached the WPQ ☹

- Ensuring durability requires expensive explicit flushes
- **Truely awfull** programming model, especially on multi-core!

↪ **CXL:** We don't know yet, but most probably like ADR 😱



Visibility ≠ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when ... it depends

↪ **eADR:** Change has reached the L1 ↦ **Visibility = Persistency** ☹

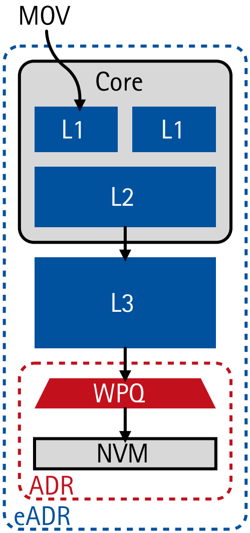
- **Great concept!** ... that unfortunately did not made it to market

↪ **ADR:** Changed cache line has *eventually* reached the WPQ ☹

- Ensuring durability requires expensive explicit flushes
- **Truely awfull** programming model, especially on multi-core!

↪ **CXL:** We don't know yet, but most probably like ADR 😱

↪ **General:** Assume *persist granularity* of a single cache line [6, 19]



Visibility ≠ Persistency

- A change becomes **visible** to other cores, when it reaches the L1 Cache
  - We can order multiple changes by memory barriers.
  - All our multi-core algorithms rely on this!
- A change becomes **persistent** on NVRAM, when ... it depends

↪ **eADR:** Change has reached the L1 ↦ **Visibility = Persistency** ☹️

- **Great concept!** ... that unfortunately did not made it to market

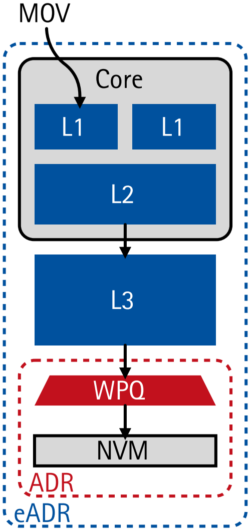
↪ **ADR:** Changed cache line has *eventually* reached the WPQ ☹️

- Ensuring durability requires expensive explicit flushes
- **Truely awfull** programming model, especially on multi-core!

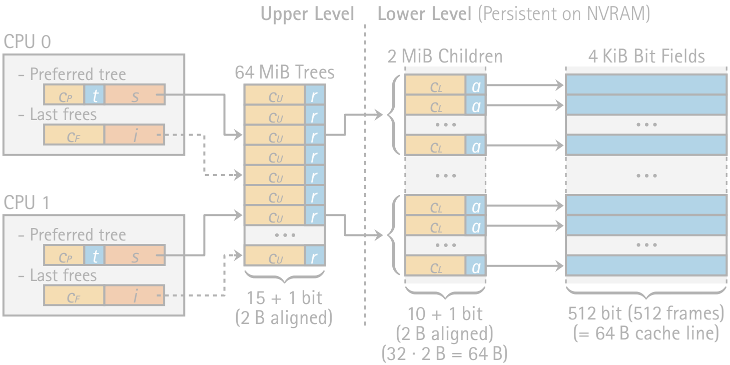
↪ **CXL:** We don't know yet, but most probably like ADR 😱

↪ **General:** Assume *persist granularity* of a single cache line [6, 19]

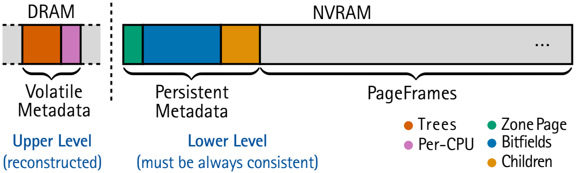
Problem: Fixing Hardware in Software



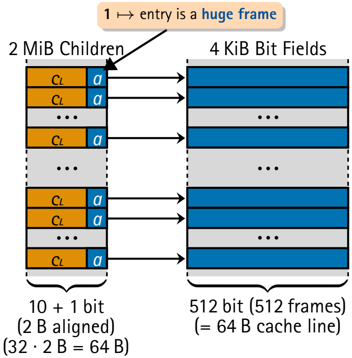
LLFree – Crash Consistency and Recovery on NVRAM



- Only Lower Level kept in NVRAM
  - Plus extra zone page for metadata and crash flag
- Upper Level can be restored

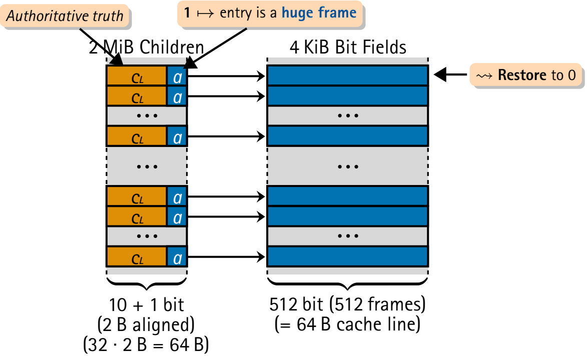


LLFree – Crash Consistency and Recovery on NVRAM



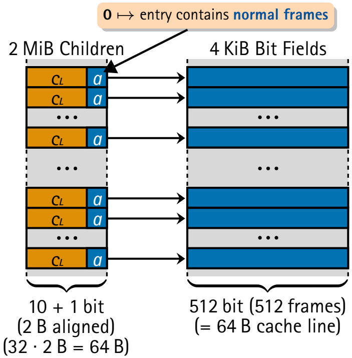
- **Single cache-line rule:** Exactly one cache line (selected by the *a*-flag) is the *authoritative truth*
  - **1:** Entry is allocated as huge frame

LLFree – Crash Consistency and Recovery on NVRAM



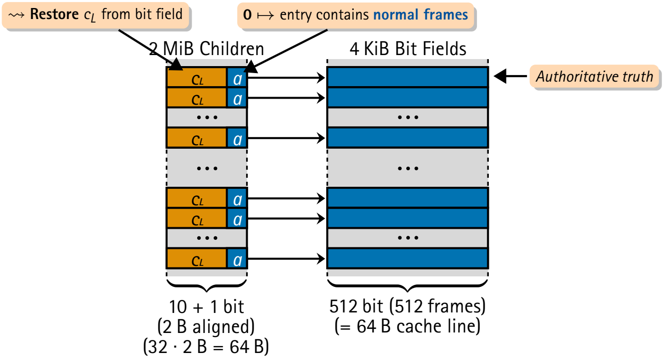
- **Single cache-line rule:** Exactly one cache line (selected by the *a*-flag) is the *authoritative truth*
  - **1:** Entry is allocated as huge frame ⇝ **child entry** defines the *truth*

LLFree – Crash Consistency and Recovery on NVRAM



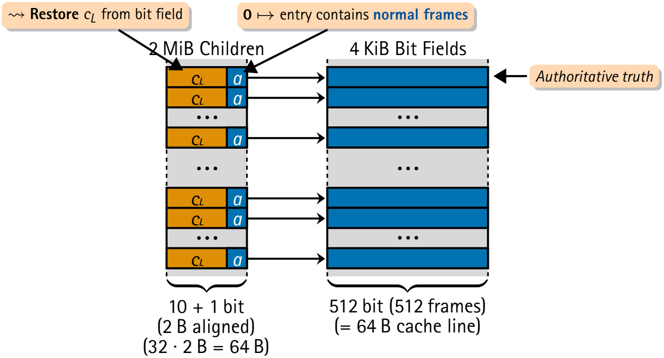
- **Single cache-line rule:** Exactly one cache line (selected by the *a*-flag) is the *authoritative truth*
  - **1:** Entry is allocated as huge frame ⇝ **child entry** defines the *truth*
  - **0:** Entry is free/allocated as normal frames

LLFree – Crash Consistency and Recovery on NVRAM



- **Single cache-line rule:** Exactly one cache line (selected by the *a*-flag) is the *authoritative truth*
  - **1:** Entry is allocated as huge frame ⇝ **child entry** defines the *truth*
  - **0:** Entry is free/allocated as normal frames ⇝ bits in **bit field** define the *truth*

LLFree – Crash Consistency and Recovery on NVRAM



⇝ **Restore** c_L from bit field

0 ↦ entry contains **normal frames**

2 MiB Children

4 KiB Bit Fields

Authoritative truth

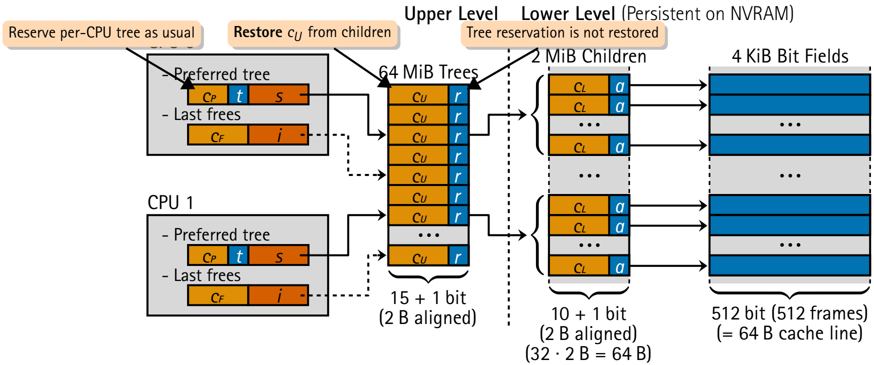
10 + 1 bit
(2 B aligned)
(32 · 2 B = 64 B)

512 bit (512 frames)
(= 64 B cache line)

- **Single cache-line rule:** Exactly one cache line (selected by the a-flag) is the *authoritative truth*
  - **1:** Entry is allocated as huge frame ⇝ **child entry** defines the *truth*
  - **0:** Entry is free/allocated as normal frames ⇝ bits in **bit field** define the *truth*

↪ Works with the minimal *persist granularity* offered by any NVM implementation.

LLFree – Crash Consistency and Recovery on NVRAM



Reserve per-CPU tree as usual

Restore c_U from children

Tree reservation is not restored

- **Upper Level** information is simply recreated at boot time.

↪ **Crash-consistent** page frame allocation and deallocation for normal and huge frames!

6.4 Summary and Conclusion

Summary: OS Challenges for Modern Memory Systems

- In the end, **everything has become a memory problem!**
  - Thread-level paralellism ⇝ memory placement.
  - I/O throughput ⇝ memory allocation.
  - Contention ⇝ memory interaction.

Summary: OS Challenges for Modern Memory Systems

- In the end, **everything has become a memory problem!**
  - Thread-level paralellism ⇝ memory placement.
  - I/O throughput ⇝ memory allocation.
  - Contention ⇝ memory interaction.
- Hardware advances (over 30 years) are **uneven** – and will continue to be!
  - RAM: 25 000x larger L1: 250x larger TLB: 25x larger
  - RAM: 500–1 000x higher througput 2x lower latency
  - I/O: 5 000–90 000x higher throughput.
  - NVRAM: It's a thing, but SSDs still 5-10x cheaper.
- OS memory managment is still dominated by the **„Mach view"**.
  - RAM is scarce. Share it.
  - Memory is an implict resource. Demand paging for everything.
  - I/O is slow. Other overheads neglectible.

Summary: OS Challenges for Modern Memory Systems

- In the end, **everything has become a memory problem!**
  - Thread-level paralellism ⇝ memory placement.
  - I/O throughput ⇝ memory allocation.
  - Contention ⇝ memory interaction.
- Hardware advances (over 30 years) are **uneven** – and will continue to be!
  - RAM: 25 000x larger L1: 250x larger TLB: 25x larger
  - RAM: 500–1 000x higher througput 2x lower latency
  - I/O: 5 000–90 000x higher throughput.
  - NVRAM: It's a thing, but SSDs still 5-10x cheaper.
- OS memory managment is still dominated by the **„Mach view“**.
  - RAM is scarce. Share it.
  - Memory is an implict resource. Demand paging for everything.
  - I/O is slow. Other overheads neglectible.

↪ **Lots of things to do!**

Conclusion: Problems and Principles for Memory Management

Problems

- The Cost of Sharing
- External Fragmentation is back
- The Hierarchy is a Network
- Designed for Scarcity, not Latency
- Software must fix Broken Hardware
- Designed for Slow and Sequential I/O

Principles

- Explicit and Non-Shared Semantics
- Hardware-Specific Granularities
- Constructive Contention Avoidance
- Memory Scarcity is the Exception
- Mitigate Hardware Problems (for Now)
- Parallel and Asynchronous I/O

6.5 Referenzen

Referenzen

[1] Mike Accetta, Robert Baron, David Golub u. a. „MACH: A New Kernel Foundation for UNIX Development". In: *Proceedings of the USENIX Summer Conference*. USENIX Association, Juni 1986, S. 93–113.

[2] Nadav Amit, Amy Tai und Michael Wei. „Don't shoot down TLB shootdowns!" In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, S. 1–14.

[3] Andrew Baumann, Simon Peter, Adrian Schüpbach u. a. „Your computer is already a distributed system. Why isn't your OS?" In: *HotOS*. 2009.

[4] Silas Boyd-Wickizer, Haibo Chen, Rong Chen u. a. „Corey: An Operating System for Many Cores". In: *8th Symposium on Operating System Design and Implementation (OSDI '08)* (San Diego, CA, USA). Berkeley, CA, USA: USENIX Association, 2008, S. 43–57.

[5] Zhangyu Chen, Yu Hua, Bo Ding u. a. „Lock-free Concurrent Level Hashing for Persistent Memory". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Juli 2020, S. 799–812. ISBN: 978-1-939133-14-4. URL: `https://www.usenix.org/conference/atc20/presentation/chen`.

[6] Jeremy Condit, Edmund B. Nightingale, Christopher Frost u. a. „Better I/O Through Byte-addressable, Persistent Memory". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. Big Sky, Montana, USA: ACM, 2009, S. 133–146. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629589`.

[7] Dave Dice und Alex Kogan. „Compact NUMA-aware locks". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, S. 1–15.

Referenzen (Forts.)

[8] Dawson R. Engler, M. Frans Kaashoek und James O'Toole. „Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain, CO, USA). New York, NY, USA: ACM Press, Dez. 1995, S. 251–266. ISBN: 0-89791-715-4. DOI: 10.1145/224057.224076.

[9] John Fotheringham. „Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store". In: *Communications of the ACM* 4.10 (Okt. 1961), S. 435–436.

[10] Michal Friedman, Maurice Herlihy, Virendra Marathe u. a. „A persistent lock-free queue for non-volatile memory". In: *ACM SIGPLAN Notices* 53.1 (2018), S. 28–40.

[11] Robert A. Gingell, J. Moran und William Shannon. „Virtual Memory Architecture in SunOS". In: *Proceedings of Summer '87 USENIX Conference*. Juni 1987.

[12] Hermann Härtig, Michael Hohmuth, Jochen Liedtke u. a. „The Performance of μ-Kernel-Based Systems". In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. New York, NY, USA: ACM Press, Okt. 1997. DOI: 10.1145/269005.266660.

[13] Joseph Izraelevitz, Hammurabi Mendes und Michael L Scott. „Linearizability of persistent memory objects under a full-system-crash failure model". In: *International Symposium on Distributed Computing*. Springer. 2016, S. 313–327.

[14] Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao u. a. „Vorpal: Vector clock ordering for large persistent memory systems". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, S. 435–444.

Referenzen (Forts.)

[15] Youngjin Kwon, Hangchen Yu, Simon Peter u. a. „Coordinated and Efficient Huge Page Management with Ingens“. In: *12th Symposium on Operating Systems Design and Implementation (OSDI '16)*. Savannah, GA, USA: USENIX Association, 2016, S. 705–721. ISBN: 9781931971331.

[16] Viktor Leis, Adnan Alhomssi, Tobias Ziegler u. a. „Virtual-Memory Assisted Buffer Management“. In: *Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD'23)*. Accepted at SIGMOD'23, to appear. Seattle, WA, USA: ACM, Juni 2023.

[17] Juan Navarro, Sitaram Iyer und Alan Cox. „Practical, Transparent Operating System Support for Superpages“. In: *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. Boston, MA: USENIX Association, Dez. 2002.

[18] Elliot I. Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972. ISBN: 0-262-15012-3.

[19] Steven Pelley, Peter M. Chen und Thomas F. Wenisch. „Memory Persistency“. In: *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. Minneapolis, Minnesota, USA: IEEE Press, 2014, S. 265–276. ISBN: 9781479943944.

[20] Markus Velten, Robert Schöne, Thomas Ilsche u. a. „Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors“. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 2022, S. 165–175.

[21] Tianzheng Wang, Justin Levandoski und Per-Ake Larson. „Easy Lock-Free Indexing in Non-Volatile Memory“. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, S. 461–472. DOI: `10.1109/ICDE.2018.00049`.

Referenzen (Forts.)

[22] David Wentzlaff und Anant Agarwal. „Factored operating systems (fos): the case for a scalable operating system for multicores“. In: *ACM SIGOPS Operating Systems Review* 43 (2 Apr. 2009), S. 76–85. ISSN: 0163-5980. DOI: 10.1145/1531793.1531805.

[23] Jian Yang, Juno Kim, Morteza Hoseinzadeh u. a. „An Empirical Guide to the Behavior and Use of Scalable Persistent Memory“. In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, S. 169–182. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/yang.