barkhausen institut

Microkernel Construction

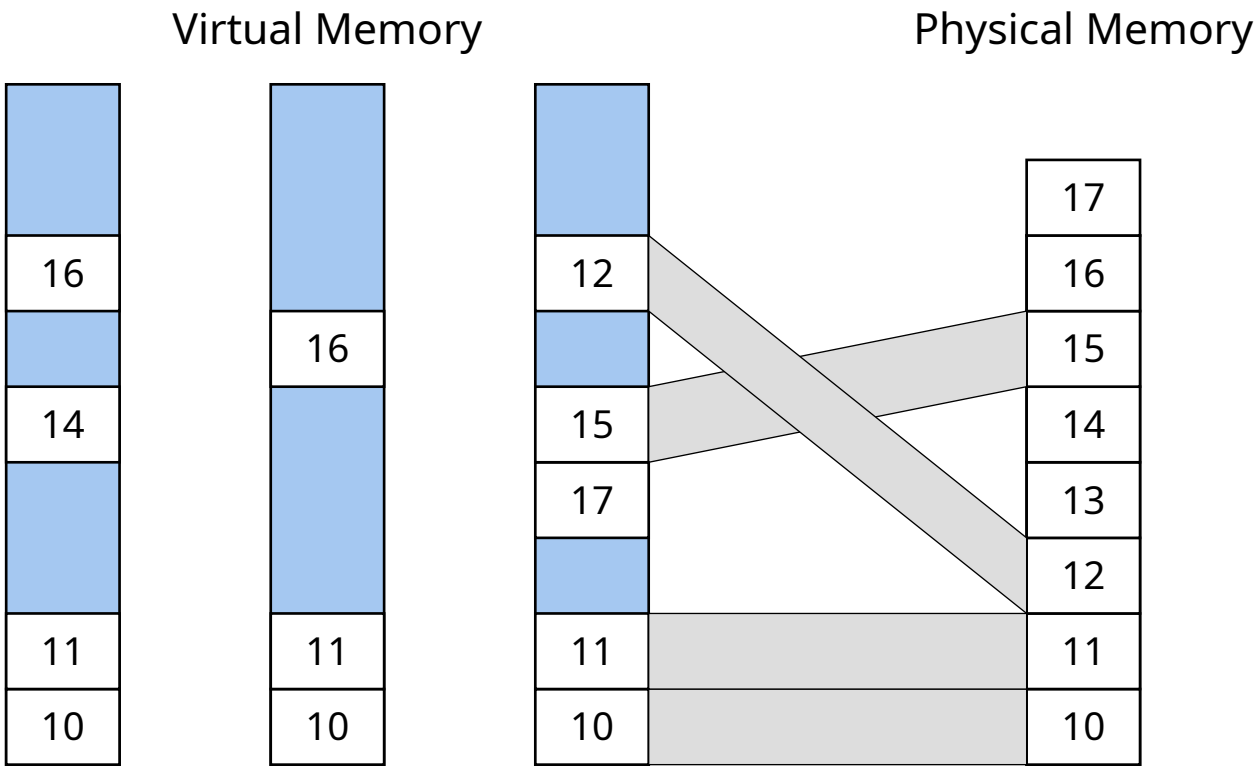
Address Spaces

Nils Asmussen

04/21/2026

- **Virtual Memory Recap**
  - **Page Tables**
  - Translation Lookaside Buffer
- Core-local Mappings
- Manipulating Page Tables

Virtual Memory Basics



Paging (x86-64)

- Translation of linear to physical addresses
- Done by memory management unit (MMU)

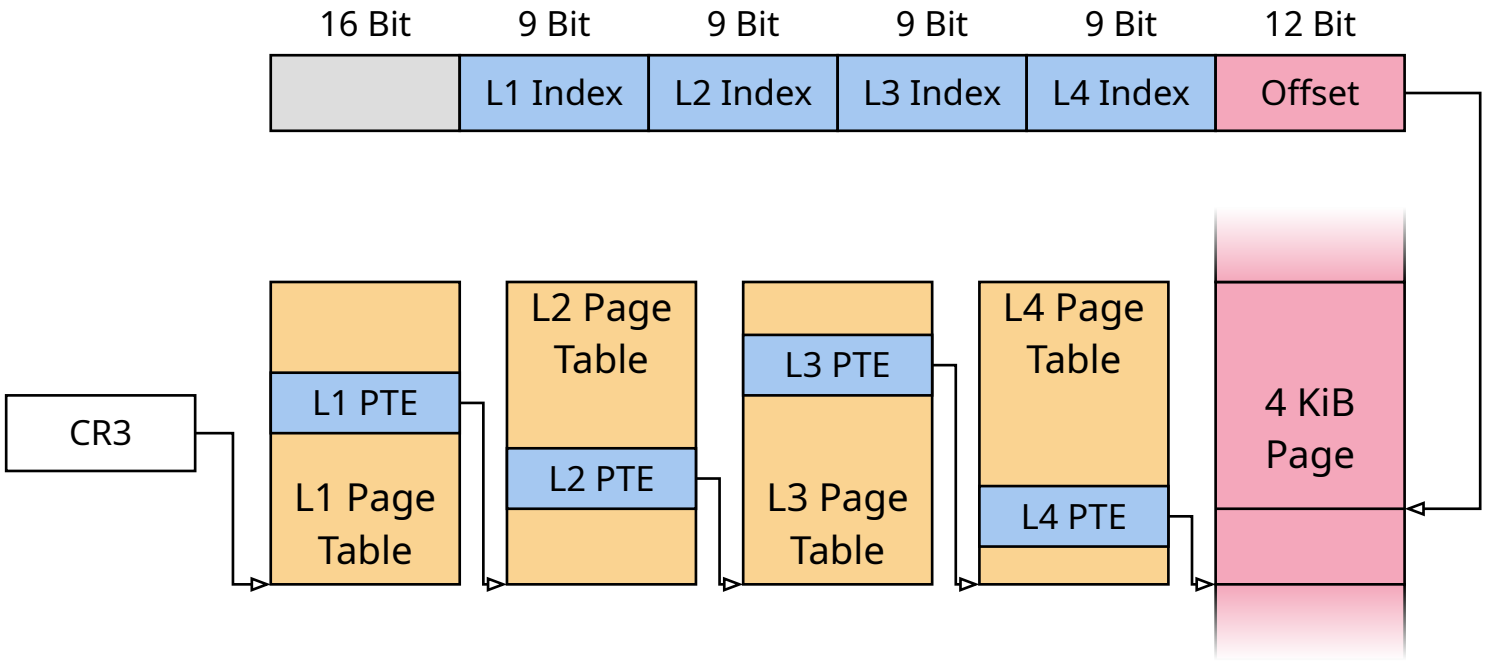
Paging (x86-64)

- Translation of linear to physical addresses
- Done by memory management unit (MMU)
- Hardware defines data structures:
  - Page Directory Base Register (CR3)
  - Page Table (PT): contains 512 page table entries (PTEs)
  - Page Directory: contains 512 page directory entries (PDEs)
  - Page Directory Pointer Table (PDPT):
    contains 512 page directory pointer entries (PDPTE)
  - Page Map Level 4 Table (PML4):
    contains 512 page map level 4 entries (PML4E)

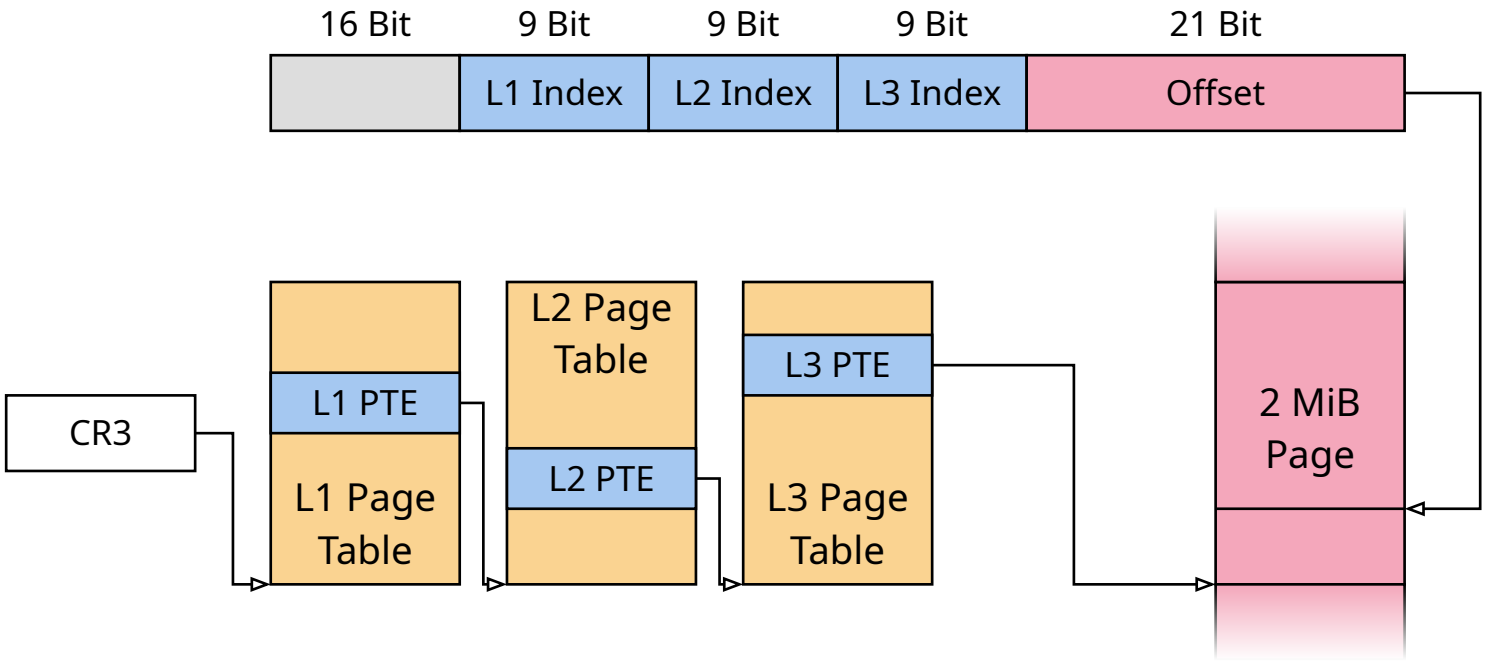
Paging (x86-64)

- Translation of linear to physical addresses
- Done by memory management unit (MMU)
- Hardware defines data structures:
  - Page Directory Base Register (CR3)
  - Page Table (PT): contains 512 page table entries (PTEs)
  - Page Directory: contains 512 page directory entries (PDEs)
  - Page Directory Pointer Table (PDPT):
    contains 512 page directory pointer entries (PDPTE)
  - Page Map Level 4 Table (PML4):
    contains 512 page map level 4 entries (PML4E)
- Paging data structures use physical addresses

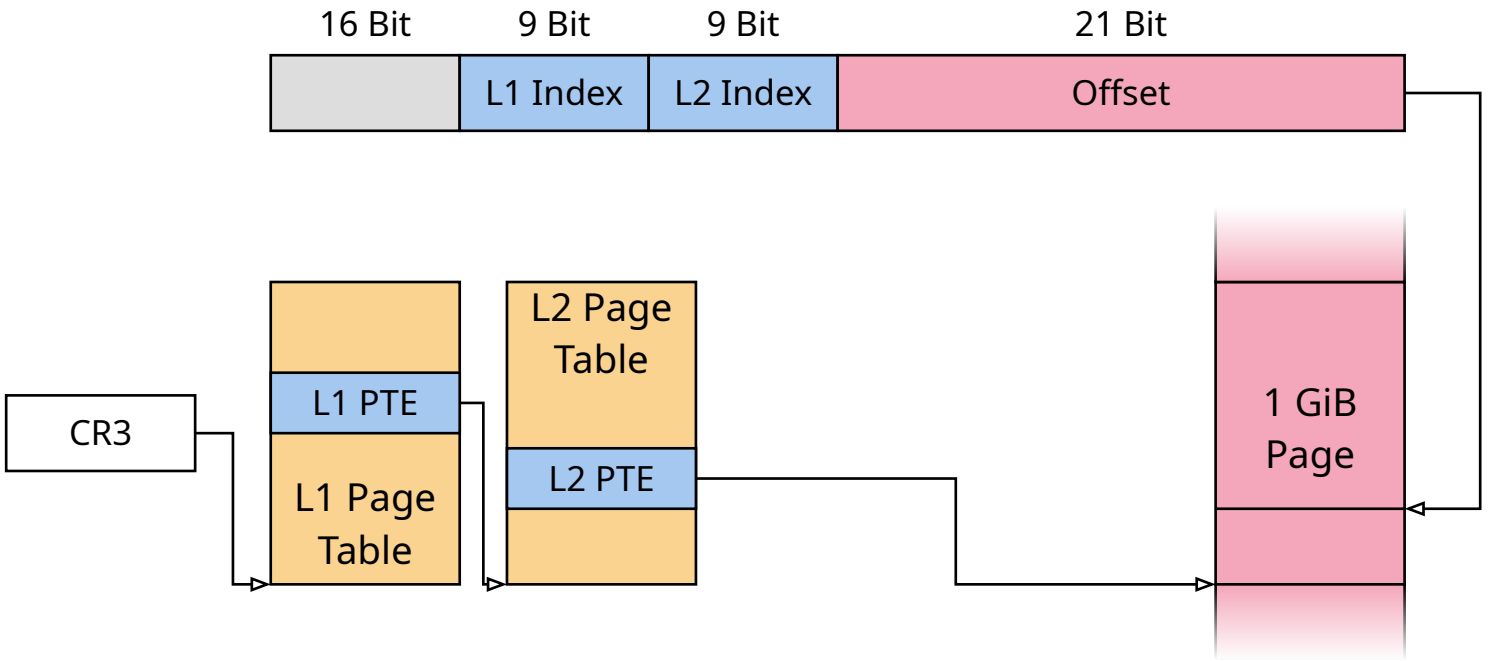
Address Translation (x86-64)



Address Translation (x86-64)



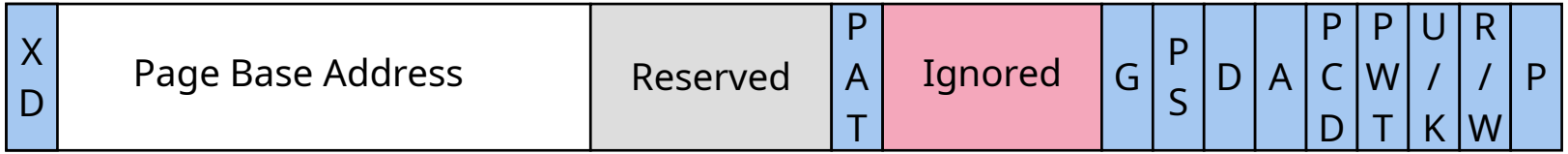
Address Translation (x86-64)



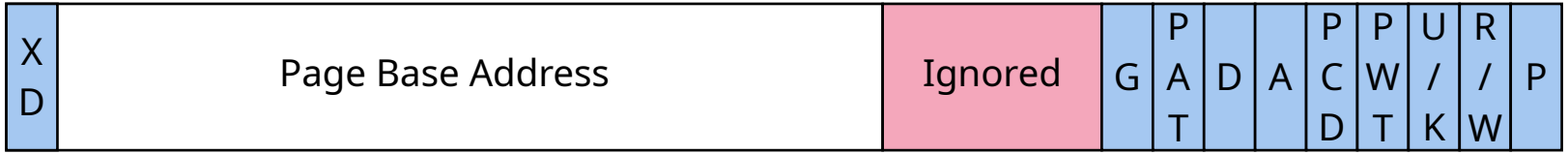
PDEs and PTEs (x86-64)

PDE (4 KiB Page):

PDE (2 MiB Page):



PTE (4 KiB Page):



- **Virtual Memory Recap**
  - Page Tables
  - **Translation Lookaside Buffer**
- Core-local Mappings
- Manipulating Page Tables

Translation Lookaside Buffer

- Caches recent virtual-to-physical translations
- Avoids expensive page-table walk
- Must be kept consistent with the page tables by the OS
- No TLB coherency protocol
- On modifications, OS must flush relevant TLB entries

TLB – Flushes and Shootdowns

- TLB flush triggered by `CR3` reload or `INVLPG` instruction
- No TLB flush required when upgrading page attributes
- `CR3` reload does not flush global pages
- TLB shootdowns for page tables active on other cores
  - Expensive signaling and synchronization
  - Inter-Processor-Interrupt (IPI)

TLB – Performance

- TLB pressure can impact performance
  - TLB misses require expensive page-table walks
  - Larger pages reduce TLB misses
  - Large pages lead to higher internal fragmentation

TLB – Performance

- TLB pressure can impact performance
  - TLB misses require expensive page-table walks
  - Larger pages reduce TLB misses
  - Large pages lead to higher internal fragmentation
- TLB flushes are costly
  - entries can be *tagged* with address space number (ASN)
  - avoid flushes unless processes need to share ASNs

TLB – Performance

- TLB pressure can impact performance
  - TLB misses require expensive page-table walks
  - Larger pages reduce TLB misses
  - Large pages lead to higher internal fragmentation
- TLB flushes are costly
  - entries can be *tagged* with address space number (ASN)
  - avoid flushes unless processes need to share ASNs
- Power hungry
- Multi-Level TLBs
  - fast and expensive first layer
  - slower and larger second layer

- Virtual Memory Recap
- **Core-local Mappings**
  - **Per-Core Data Structures**
  - Implementation in NOVA
- Manipulating Page Tables

Per-Core Data Structures

Static array (example: L4Re)

- additional indirection
- statically limited to certain core count

Per-Core Data Structures

Static array (example: L4Re)

- additional indirection
- statically limited to certain core count

Hash table (example: tasks in Linux)

- bigger performance hit on lookup than static array
- supports any number of cores

Per-Core Data Structures

Static array (example: L4Re)

- additional indirection
- statically limited to certain core count

Hash table (example: tasks in Linux)

- bigger performance hit on lookup than static array
- supports any number of cores

Per-core mappings (example: NOVA)

- no indirection
- can be expensive with many cores if mappings change

Determining Core ID

Local Advanced Programmable Interrupt Controller (LAPIC)

- Per-core interrupt controller
- Sends interrupts received from I/O APIC to CPU core
- Can be used to send IPIs
- Determine LAPIC ID:

```C++
uint32_t apic_id = *(volatile uint32_t*)(lapic_base + 0x20);
apic_id >>= 24;
```

Determining Core ID

Local Advanced Programmable Interrupt Controller (LAPIC)

- Per-core interrupt controller
- Sends interrupts received from I/O APIC to CPU core
- Can be used to send IPIs
- Determine LAPIC ID:

```cpp
uint32_t apic_id = *(volatile uint32_t*)(lapic_base + 0x20);
apic_id >>= 24;
```

`cpuid` Instruction

- Set `EAX` to 1 and execute `cpuid`
- `EBX[31:24]` will contain the LAPIC ID

Determining Core ID

- Reading LAPIC register and executing `cpuid` is rather slow
- Can degrade performance if done on every syscall/interrupt

Determining Core ID

- Reading LAPIC register and executing `cpuid` is rather slow
- Can degrade performance if done on every syscall/interrupt
- Remember it:
  - store on top of the stack
  - thread-local storage via segment registers

Determining Core ID

- Reading LAPIC register and executing `cpuid` is rather slow
- Can degrade performance if done on every syscall/interrupt
- Remember it:
  - store on top of the stack
  - thread-local storage via segment registers
- Avoid needing it: core-local mappings

- Virtual Memory Recap
- **Core-local Mappings**
  - Per-Core Data Structures
  - **Implementation in NOVA**
- Manipulating Page Tables

Protection Domain

- Resource container
- Consists of three *spaces*:
  1. Object capability space (`Space_obj`)
  2. Memory capability space (`Space_mem`)
  3. Port I/O capability space (`Space_pio`)
- Created via `create_pd` syscall
  - All spaces initially empty

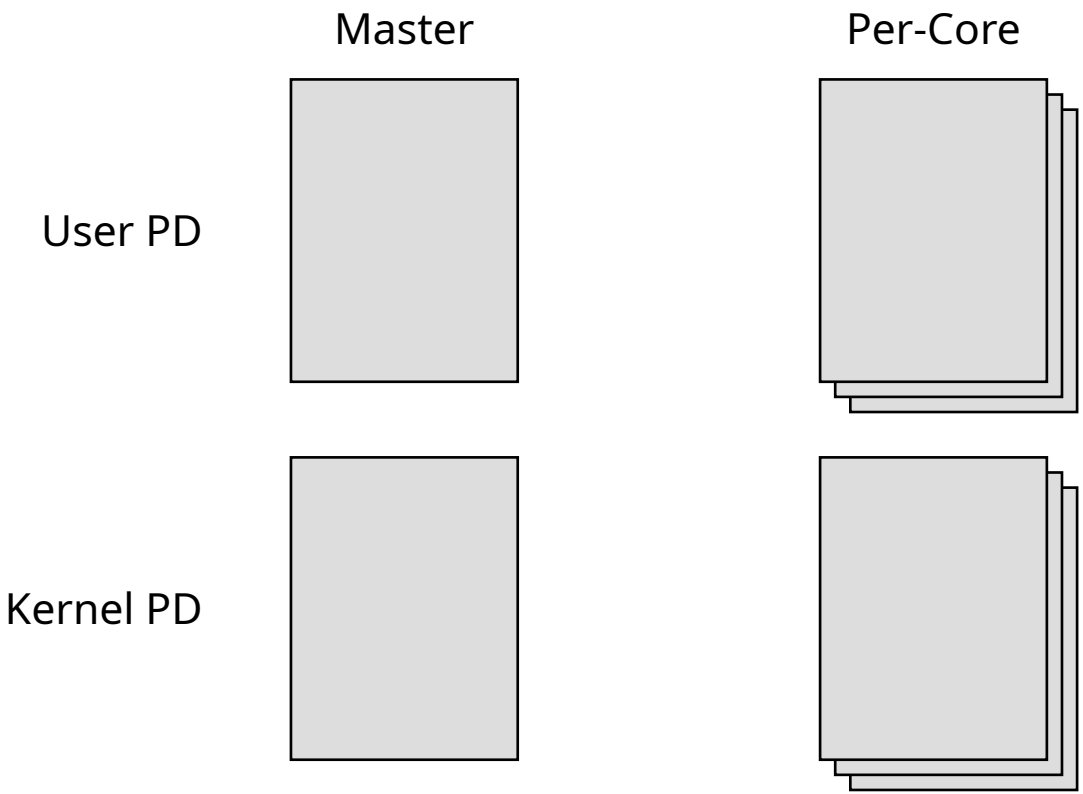
Memory Space

Memory space of a protection domain

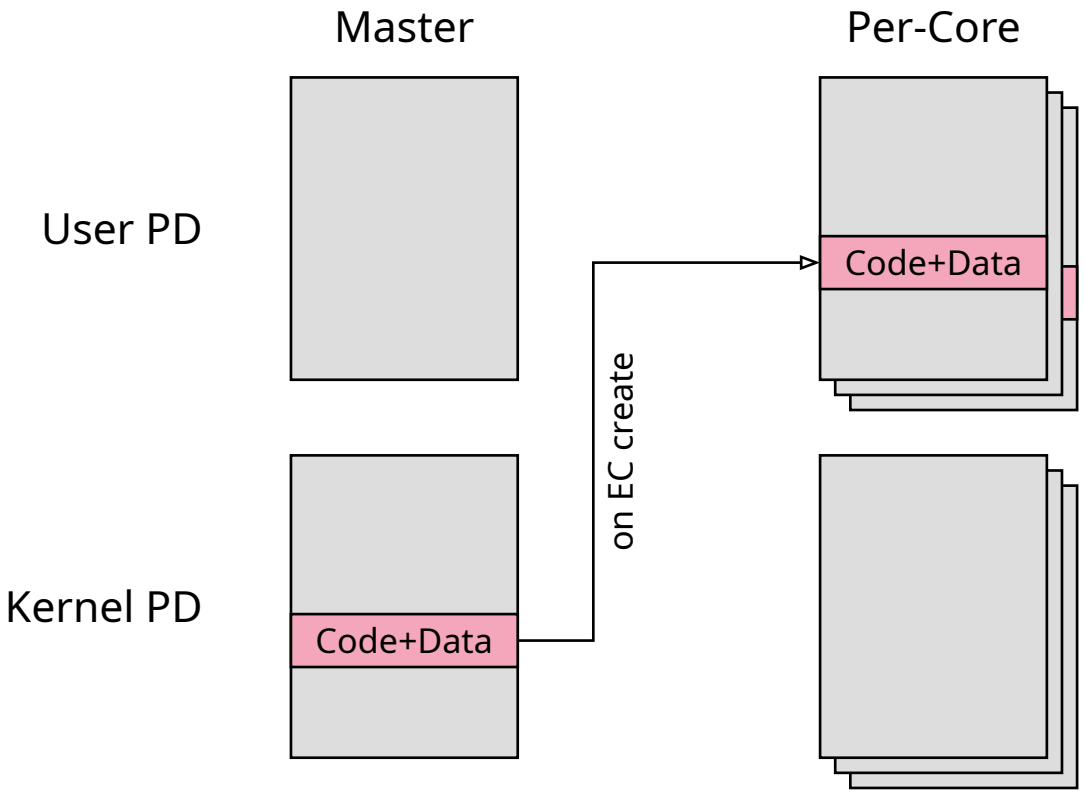
C++

```
1   class Space_mem : public Space {
2   public:
3       Hpt hpt;              // master page table
4       Hpt loc[NUM_CPU];     // per-core PTs; synced from master
5       Dpt dpt;              // DMA PT for IOMMU
6       union {
7           Ept ept;          // nested PT for Intel (VMX)
8           Hpt npt;          // nested PT for AMD (SVM)
9       };
10      Cpuset htlb;          // CPUs using this address space
11  };
```

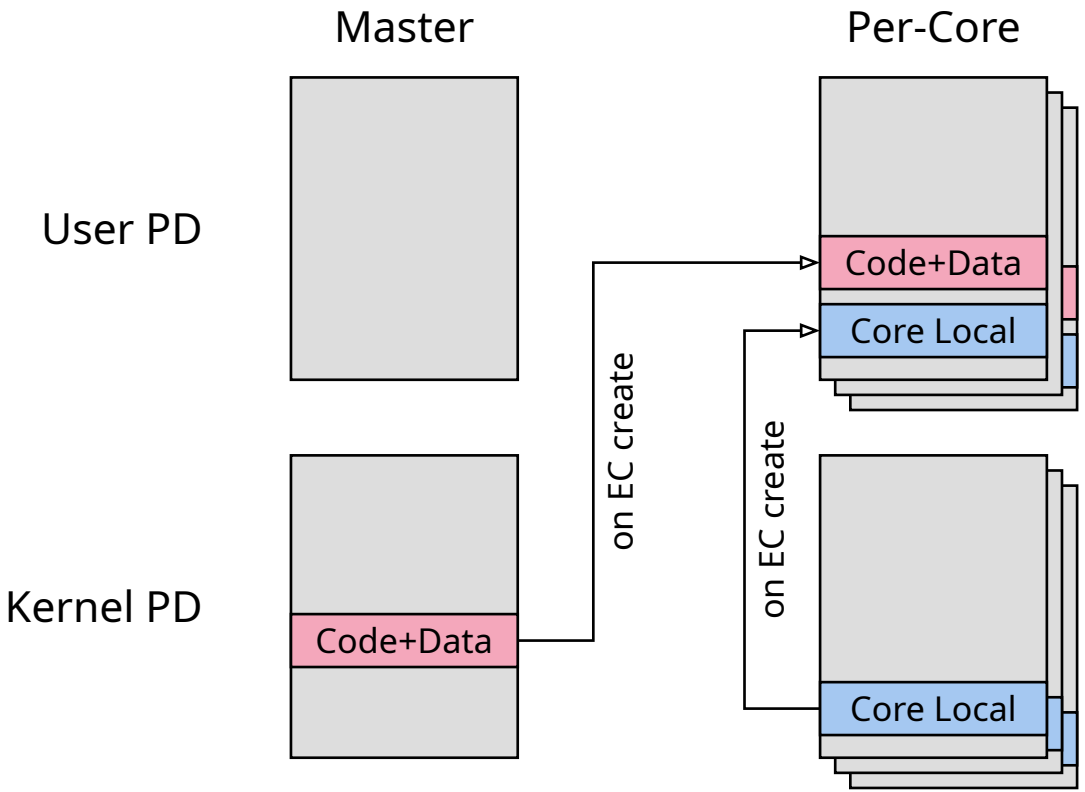
Mapping Concept in NOVA



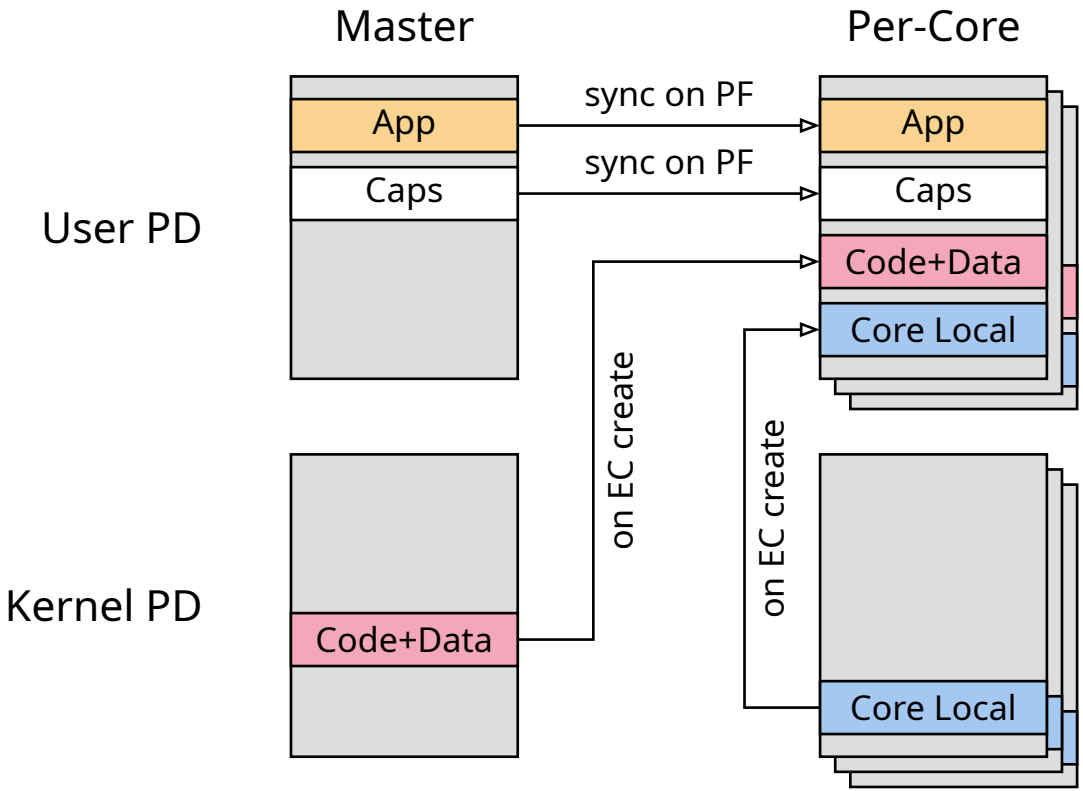
Mapping Concept in NOVA



Mapping Concept in NOVA



Mapping Concept in NOVA



Virtual Memory Layout

Start	End	Size	Purpose
`0x0000_0000_0000_0000`	`0x0000_7FFF_FFFF_F000`	≈128 TiB	User
`0xFFFF_FFFF_8100_0000`	`0xFFFF_FFFF_BFDF_FFFF`	≈990 MiB	Kernel code+data+heap
`0xFFFF_FFFF_BFFF_D000`	`0xFFFF_FFFF_BFFF_DFFF`	4 KiB	Stack
`0xFFFF_FFFF_BFFF_E000`	`0xFFFF_FFFF_BFFF_EFFF`	4 KiB	LAPIC
`0xFFFF_FFFF_BFFF_F000`	`0xFFFF_FFFF_BFFF_FFFF`	4 KiB	Core-local data
`0xFFFF_FFFF_C000_0000`	`0xFFFF_FFFF_C000_1FFF`	8 KiB	I/O bitmap
`0xFFFF_FFFF_DF00_0000`	`0xFFFF_FFFF_DFFF_FFFF`	16 MiB	Temporal mappings
`0xFFFF_FFFF_E000_0000`	`0xFFFF_FFFF_FFFF_FFFF`	512 MiB	Capabilities

Syncing Page Tables

Handler for page faults

C++

```
1  bool Ec::handle_exc_pf (Exc_regs *r) {
2    mword addr = r->cr2;
3
4    if (r->err & Hpt::ERR_U)
5      return addr < USER_ADDR &&
6             Pd::current->loc[Cpu::id].sync_from (Pd::current->hpt, addr, USER_ADDR);
7
8    if (addr >= SPC_LOCAL_IOP && addr <= SPC_LOCAL_IOP_E)
9      return Space_pio::page_fault (addr, r->err);
10   if (addr >= SPC_LOCAL_OBJ)
11     return Space_obj::page_fault (addr, r->err);
12
13   die ("#PF (kernel)", r);
14 }
```

- Virtual Memory Recap
- Core-local Mappings
- **Manipulating Page Tables**
  - **Approaches**
  - Implementation in NOVA

Manipulating Page Tables

- Most architectures translate all addresses if paging is enabled
- How can you change the structures that define this translation?
- Bootstrapping:
  - Prepare page tables in physical memory
  - Enable paging afterwards
- Map the kernel eagerly into every address space
- However, typically page tables contain physical addresses

Approaches for Page-Table Manipulation

Software Page-Table Walk

- Start at the root page table and walk through levels
- Following the physical address for the next page table
- Access this page table in virtual memory

➡️ Requires to map all page tables

Approaches for Page-Table Manipulation

Software Page-Table Walk

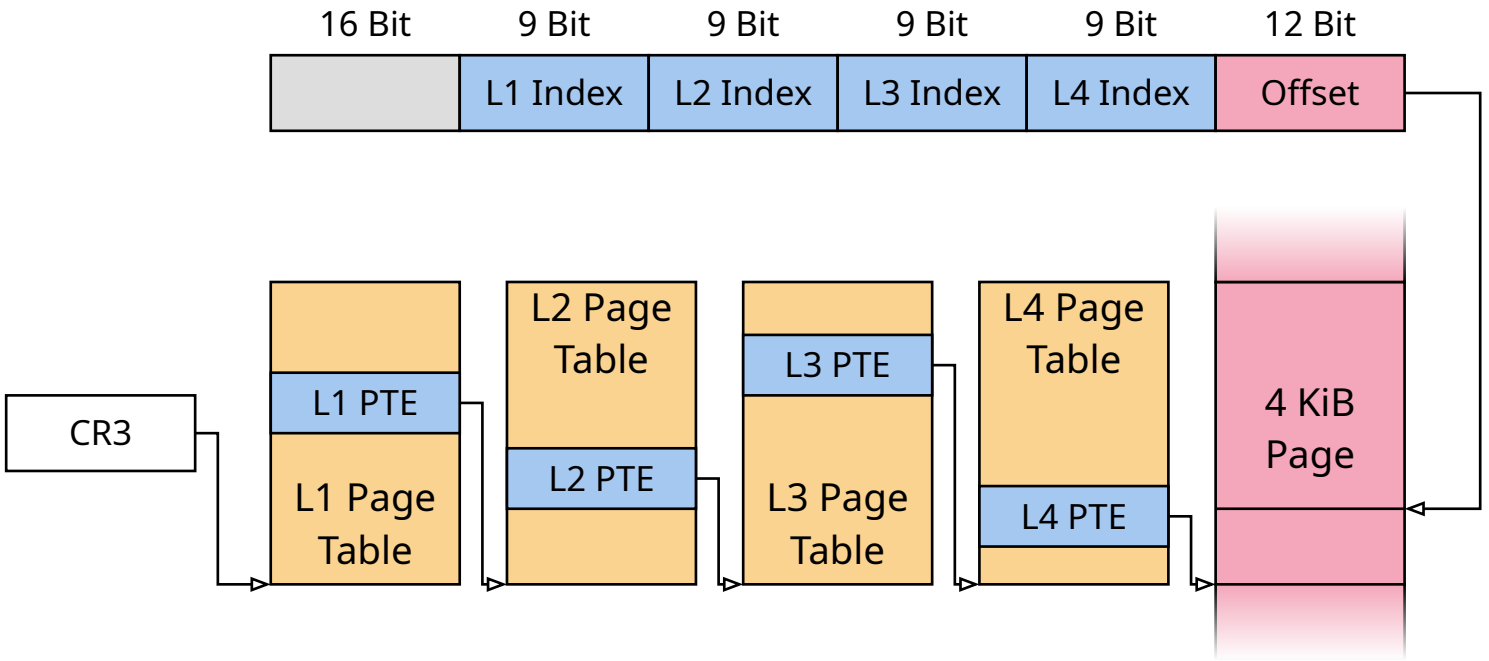
- Start at the root page table and walk through levels
- Following the physical address for the next page table
- Access this page table in virtual memory

➡️ Requires to map all page tables

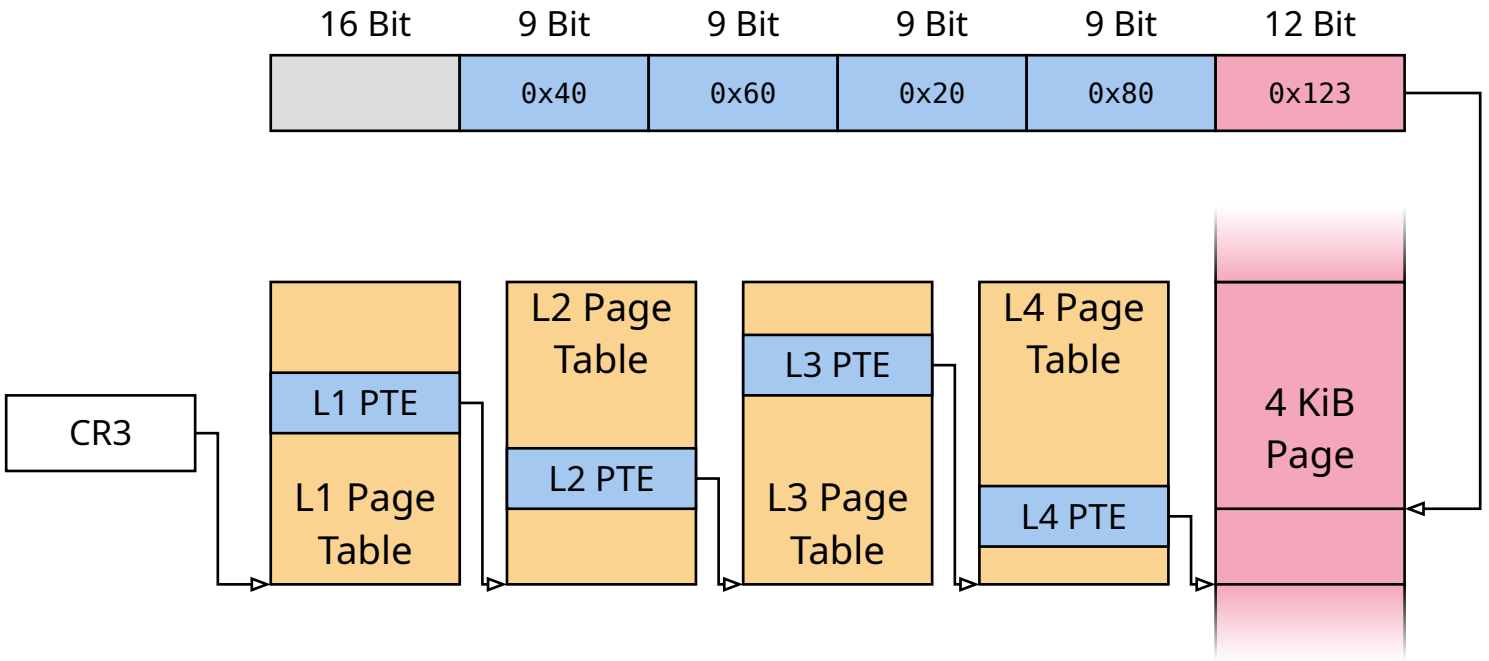
Recursive Mapping

- Map the root page table in itself
- Use this mapping to access all page-table entries

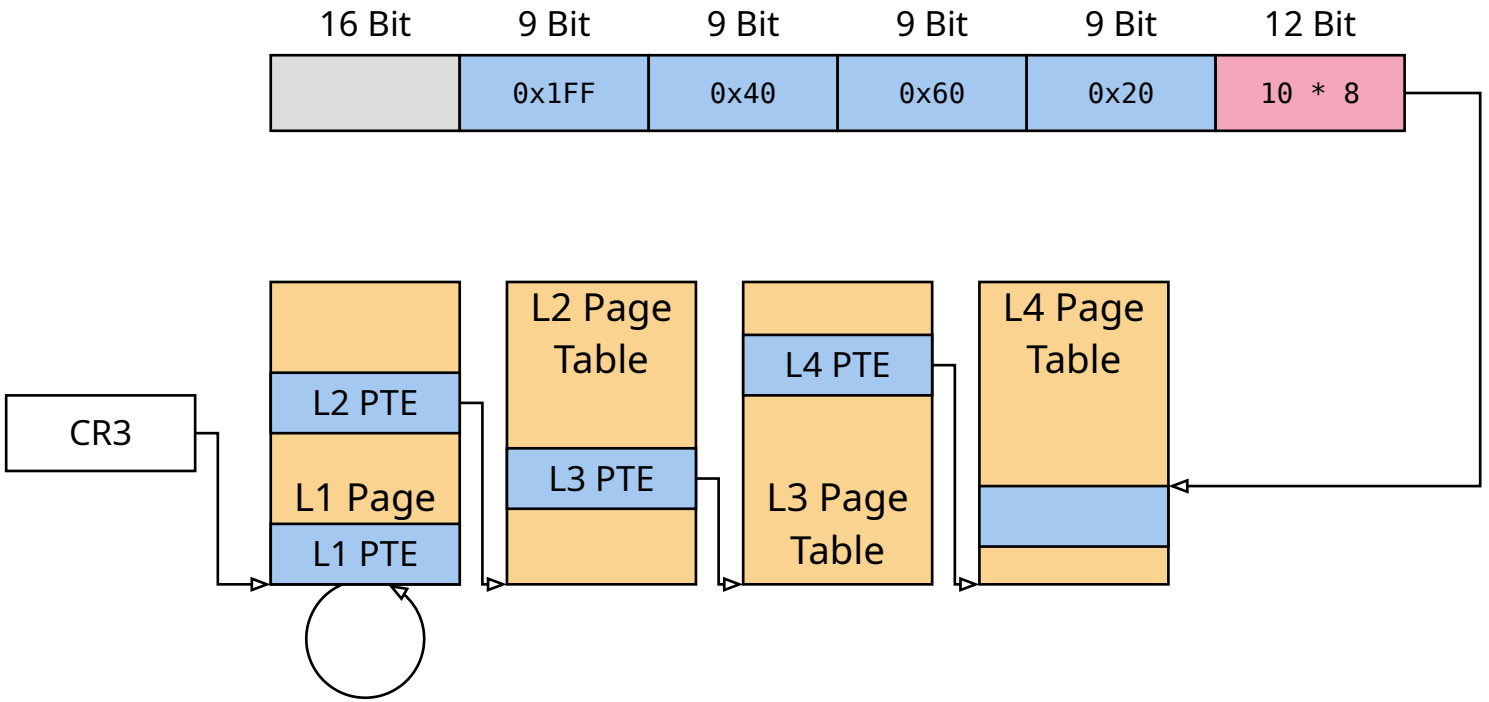
Recursive Mapping



Recursive Mapping



Recursive Mapping



Discussion

Pro Recursive Mapping

- No manual PT walk, use PT walker of CPU
- No need to have all PTs mapped

Discussion

Pro Recursive Mapping

- No manual PT walk, use PT walker of CPU
- No need to have all PTs mapped

Pro Page-Table Walk

- Requires less TLB entries
  - Physical memory for PTs can be mapped with huge pages
  - Recursive mappings touch many different virtual addresses
- Requires no TLB invalidations/shootdowns
  - Mappings for physical memory do not change
- Easy access to other address spaces
  - Recursive mapping only for same address space
  - Another recursive mapping makes downsides even worse

Page-Table Manipulation in Practice

- Most existing OSes use the software page-table walk
- Recursive mappings are used in:
  - FreeBSD
  - NetBSD
  - Windows for 32-bit targets
  - ReactOS
  - Earlier versions of Escape
  - ...

- Virtual Memory Recap
- Core-local Mappings
- **Manipulating Page Tables**
  - Approaches
  - **Implementation in NOVA**

Software Page-Table Walk in NOVA

Generic page table entry handling

C++

```
1  template <typename P, typename E, unsigned L, unsigned B>
2  class Pte {
3      E val;
4      P *walk (E virt, unsigned long level, bool add);
5      size_t lookup (E virt, Paddr &phys, mword &attr);
6      void update (E virt, mword size, E phys,
7                   mword attr, bool add);
8  };
9
10 class Hpt : public Pte<Hpt, uint32, 2, 10>;
11 class Dpt : public Pte<Dpt, uint64, 4, 9>;
12 class Ept : public Pte<Ept, uint64, 4, 9>;
```

TLB Invalidations and Shootdowns

- On revoke of memory capabilities, PTEs are updated
- When permissions are downgraded
  - Collect potentially affected core ids
  - TLB shootdown when revoke is done
  - Sends IPI to other cores, reloads `CR3` for own core
  - Avoids IPIs if possible

TLB Shootdown

TLB Shootdown on revoked memory mappings

C++

```
1   for (unsigned cpu = 0; cpu < NUM_CPU; cpu++) {
2       Pd *pd = Pd::remote (cpu); // current Pd on `cpu`
3       if (!pd->htlb.chk (cpu))   // TLB already updated?
4           continue;
5       if (Cpu::id == cpu) {
6           Cpu::hazard |= HZD_SCHED;
7           continue;
8       }
9       unsigned ctr = Counter::remote (cpu, 1); // increased when IPI handled
10      Lapic::send_ipi (cpu, VEC_IPI_RKE);      // causes scheduling
11      while (Counter::remote (cpu, 1) == ctr)
12          pause();
13  }
```

Address Space Switch

Implementation of `Pd::make_current`

C++

```
1   inline void make_current() {
2       // we are about to update our TLB; no need to send us an IPI later
3       if (EXPECT_FALSE (htlb.chk (Cpu::id)))
4           htlb.clr (Cpu::id);
5       // TLB update to date and same AS?
6       else if (EXPECT_TRUE (current == this))
7           return;
8
9       current = this;
10      // asm volatile ("mov %0, %%cr3" : : "r" (val) : "memory");
11      loc[Cpu::id].make_current();
12  }
```