Page Table Management

Advanced Operating Systems

Overview

- Page tables
- Page table management
- Page table optimizations
- Page tables and sanity/security

Memory Access

- What happens on pointer dereference?
- Where in memory does this code read?

```
int *ptr = (int *) 0x1000000;
return *ptr;
```

```
movq  $0x1000000, %rax
movl  (%rax), %eax
```

- Answer: it depends
  - Address 0x1000000 in *virtual* memory
  - May not by 0x1000000th byte in RAM

Virtual Memory

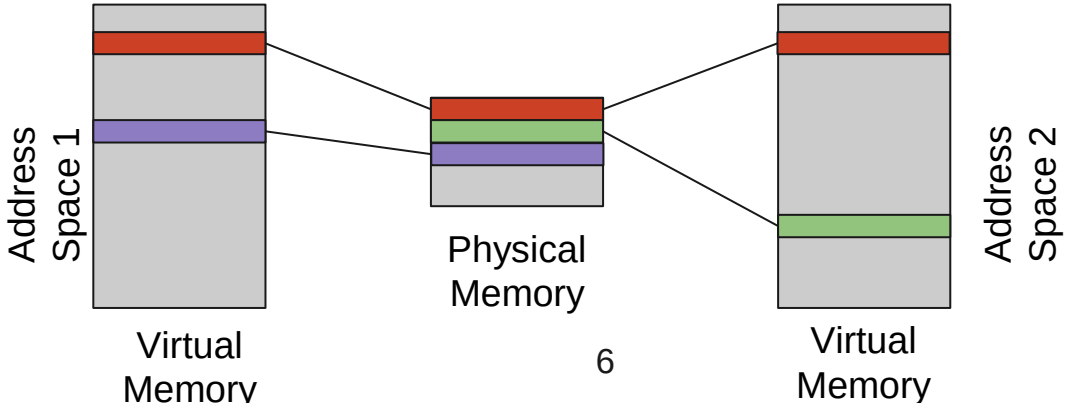
- OS tells CPU how to map *virtual* memory addresses to *physical* memory addresses
  - OS creates *page table*, storing physical addresses and indexed with virtual addresses
  - OS stores physical address of page table in special register (let’s call it cr3)
  - On memory access, CPU looks up virtual address to find physical address

Address Spaces

- OS can create multiple page tables
  - Each can define different virt → phys mapping
  - Only one is active at a time, selected by cr3
- Each page table defines an *address space*
- Why is this useful?

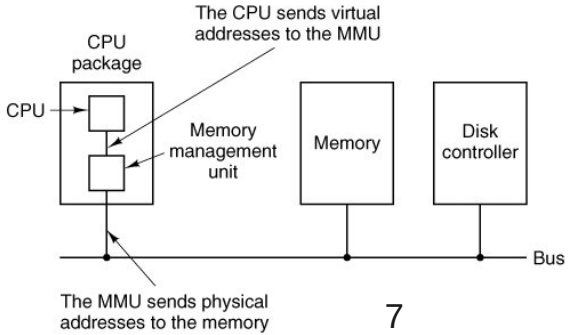
Address Spaces

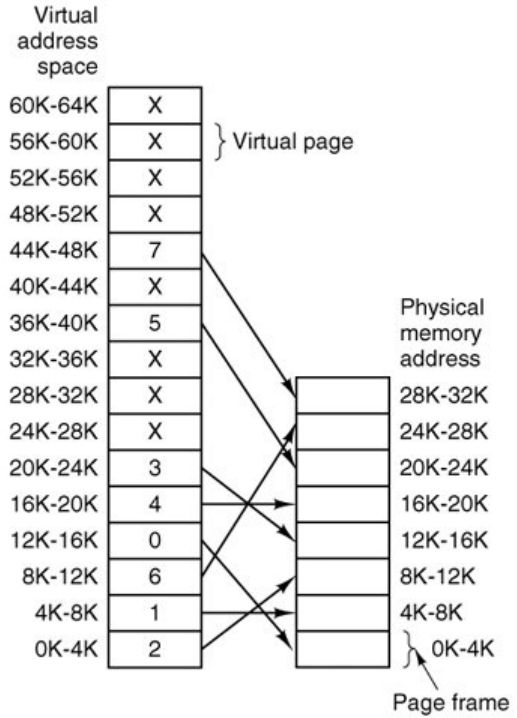
- Benefits of address spaces
  - Virtualize physical memory
  - Flexible memory management
  - Isolation and protection

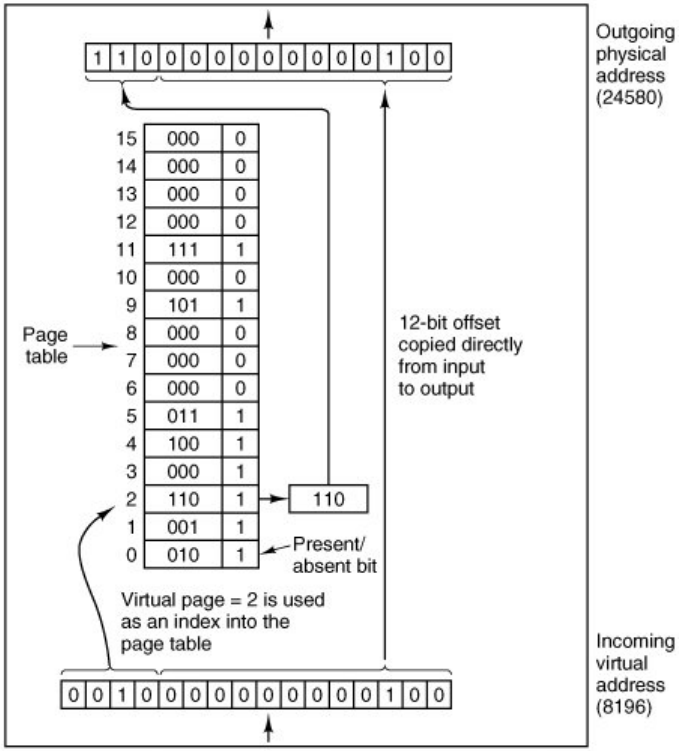


Page Tables

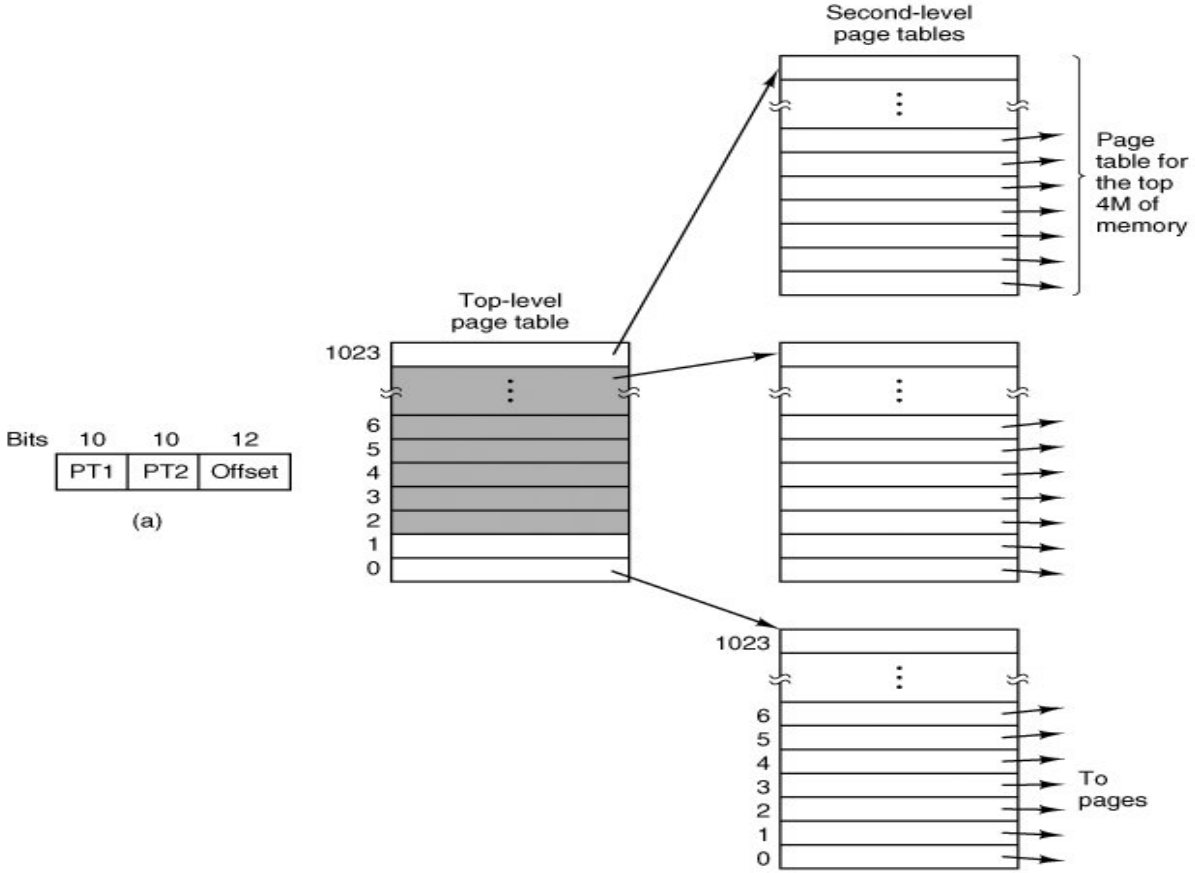
- Address translated at page granularity
  - Page: chunk of memory
  - Architecture defines possible page size(s)
- Memory Management Unit (MMU) performs translation



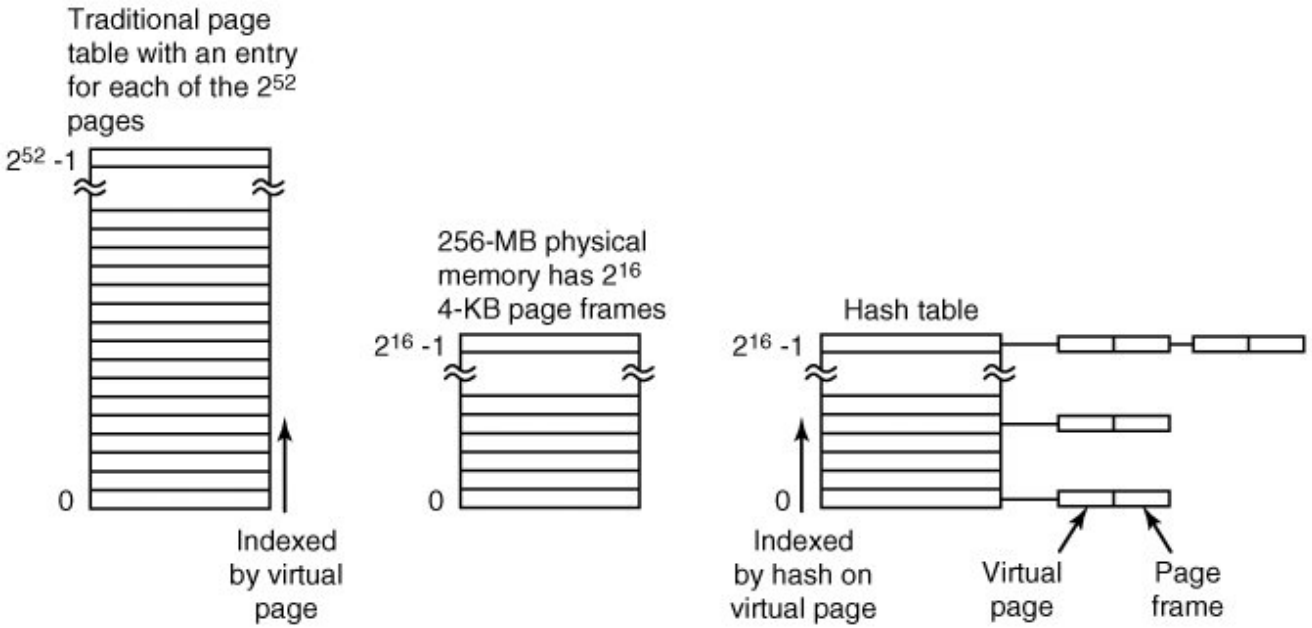




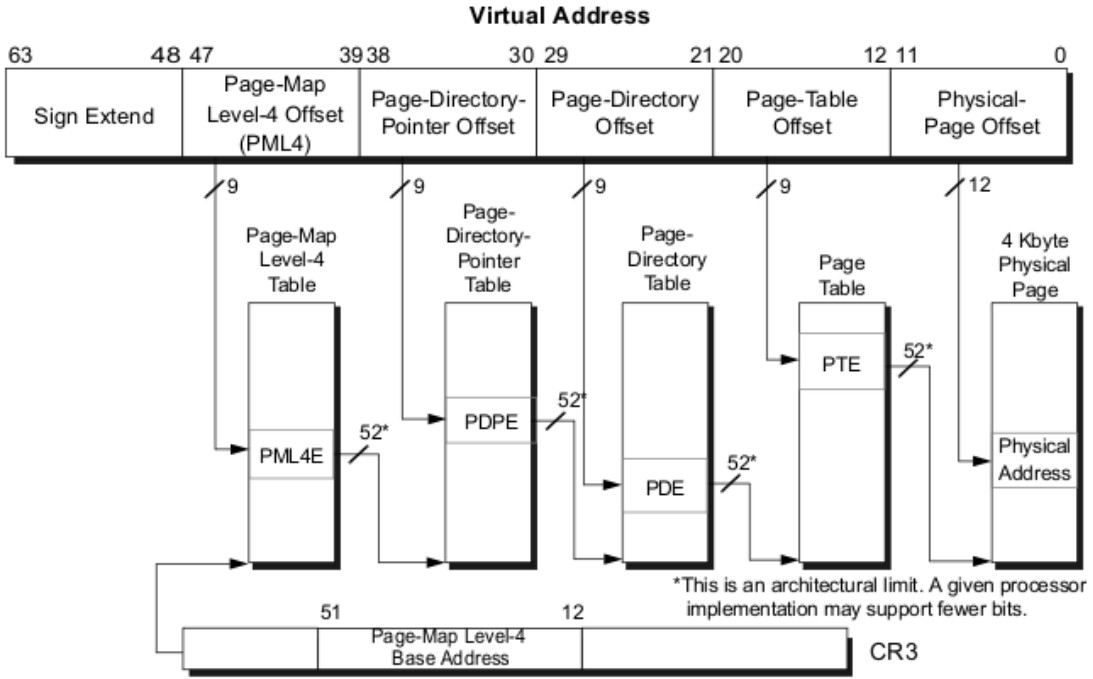
The most basic (linear) page-table (similar to PDP-11)



Hierarchical Page Tables (x86_32)



Inverted Page Tables (IA64)



Four-level Page Tables (x86_64)

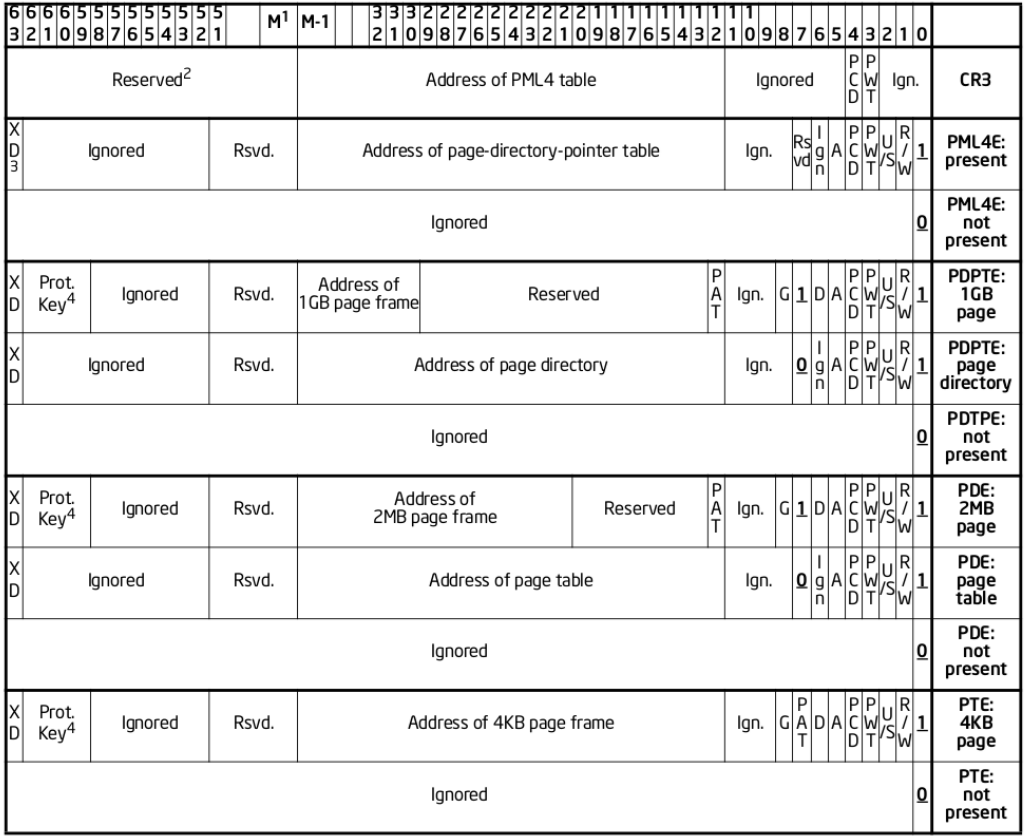
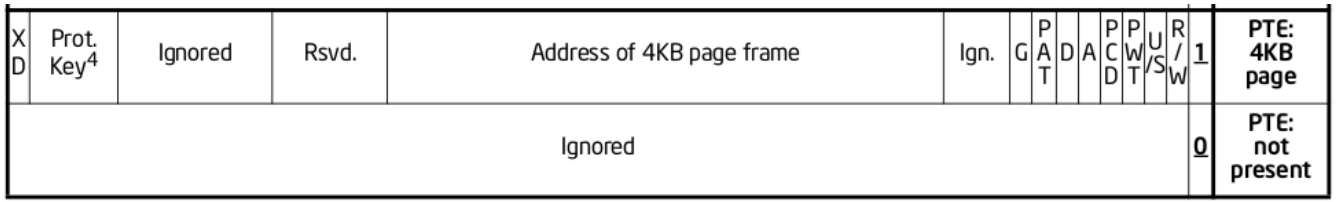


Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging

Inside Page Table Pages

Page Table Entries

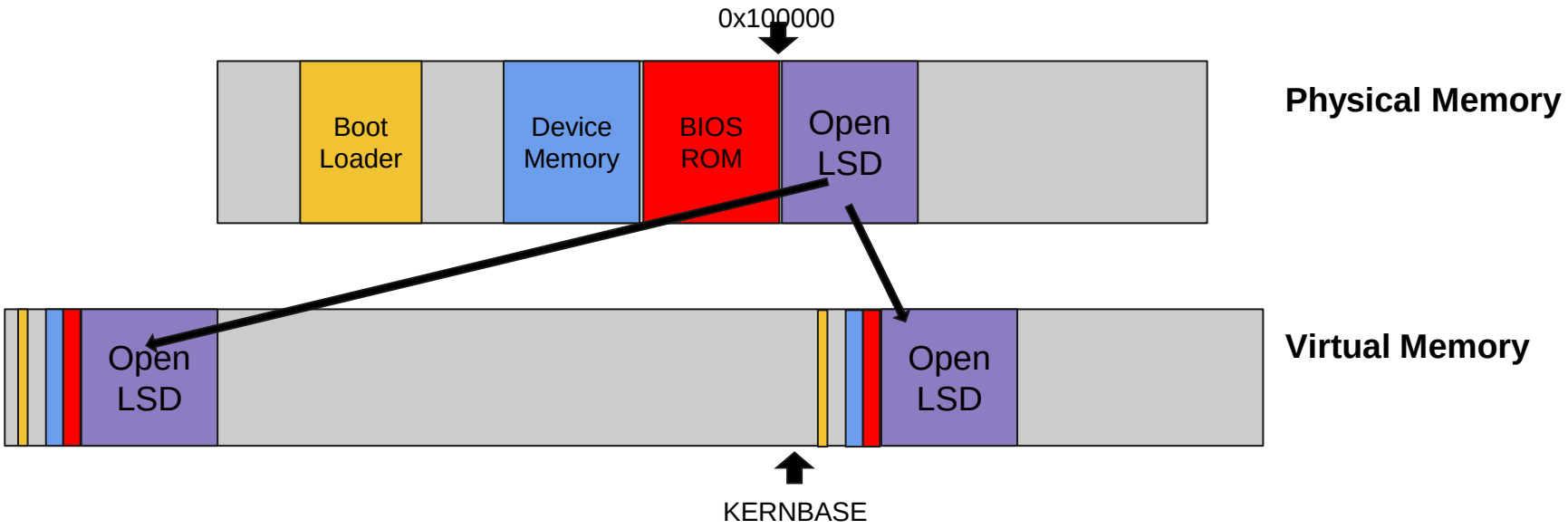


- 4KB (page), 2MB (huge page), 1GB (super page)
- 48-bit physical and virtual address space
- 64-bit page table entries (PTEs), including attributes
- **Questions**
  - What is in the address field of a PTE?
  - How much memory can you map in x86_64?
  - How much memory does one page table page map?

Page Table Management

- Static page tables for bootstrapping
- Dynamic page tables

OpenLSD:
Bootstrapping Page Tables



OpenLSD: kernel/boot.S

```
/* Set up paging to map both 0x0000000000000000 and 0xFFFF800000000000 to
 * the first 8 MiB. Since huge page is guaranteed to be available on
 * x86-64, we will be using 2 MiB pages (huge pages) to simplify the
 * mapping.
 */
movl $(PAGE_PRESENT | PAGE_WRITE | PAGE_HUGE), %eax
movl $page_dir, %edi
movl $4, %ecx

.map_pages:
movl %eax, (%edi)
addl $0x200000, %eax
addl $8, %edi
dec %ecx
jnz .map_pages
```

```
movl $page_dir, %eax
orl $(PAGE_PRESENT | PAGE_WRITE), %eax
movl %eax, pdpt

movl $pdpt, %eax
orl $(PAGE_PRESENT | PAGE_WRITE), %eax
movl %eax, pml4
movl %eax, pml4 + 256 * 8

/* Load the root of the page table hierarchy. */
movl $pml4, %eax
movl %eax, %cr3
```

Linux: arch/x86/boot/compressed/head_64.S

```
    /* Initialize Page tables to 0 */
    leal    pgtable(%ebx), %edi
    xorl    %eax, %eax
    movl    $(BOOT_INIT_PGT_SIZE/4), %ecx
    rep     stosl

    /* Build Level 4 */
    leal    pgtable + 0(%ebx), %edi
    leal    0x1007 (%edi), %eax
    movl    %eax, 0(%edi)
    addl    %edx, 4(%edi)

    /* Build Level 3 */
    leal    pgtable + 0x1000(%ebx), %edi
    leal    0x1007(%edi), %eax
    movl    $4, %ecx
1:  movl    %eax, 0x00(%edi)
    addl    %edx, 0x04(%edi)
    addl    $0x00001000, %eax
    addl    $8, %edi
    decl    %ecx
    jnz     1b
```

```
    /* Build Level 2 */
    leal    pgtable + 0x2000(%ebx), %edi
    movl    $0x00000183, %eax
    movl    $2048, %ecx
1:  movl    %eax, 0(%edi)
    addl    %edx, 4(%edi)
    addl    $0x00200000, %eax
    addl    $8, %edi
    decl    %ecx
    jnz     1b

    /* Enable the boot page tables */
    leal    pgtable(%ebx), %eax
    movl    %eax, %cr3
```

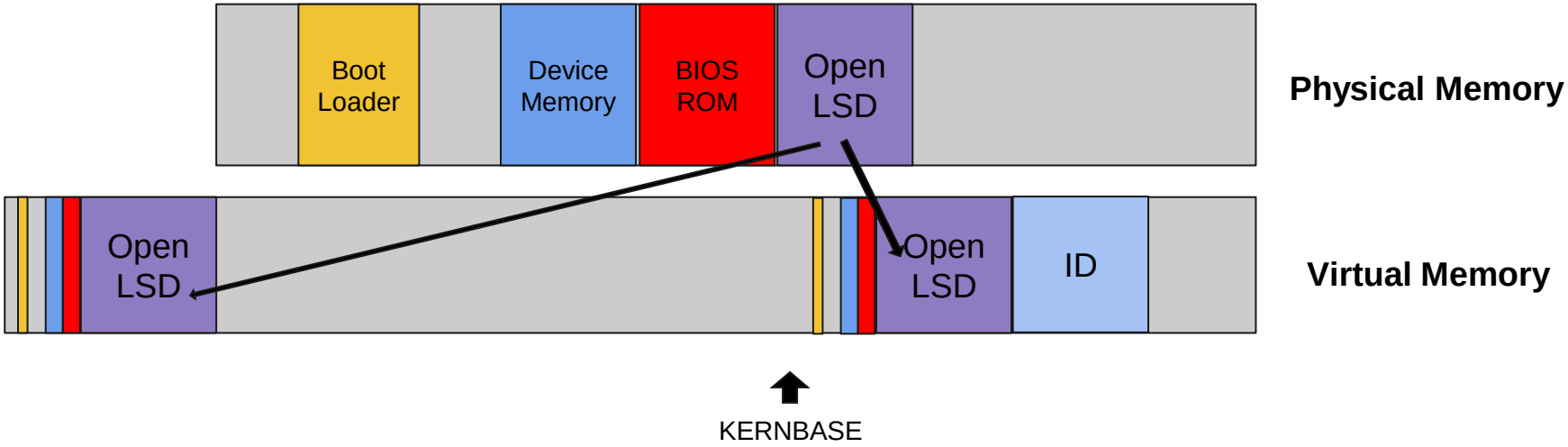
Dynamic Page Table Management (Post Boot)

- Address spaces are flexible
  - Need more memory? Map new page
  - Don’t need memory anymore? Unmap page
- What should your virtual address spaces look like? (policies)
- How would you do these things? (mechanisms)

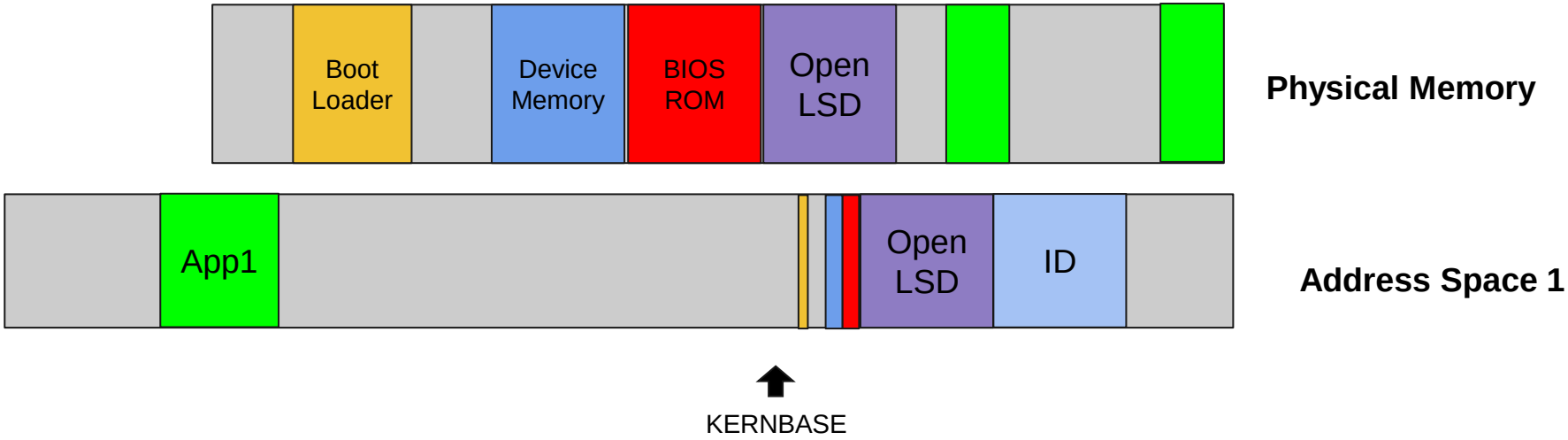
Dynamic Page Table Management (Post Boot)

- Address spaces are flexible
  - Need more memory? Map new page
  - Don’t need memory anymore? Unmap page
- What should your virtual address spaces look like? (policies)
- How would you do these things? (mechanisms)

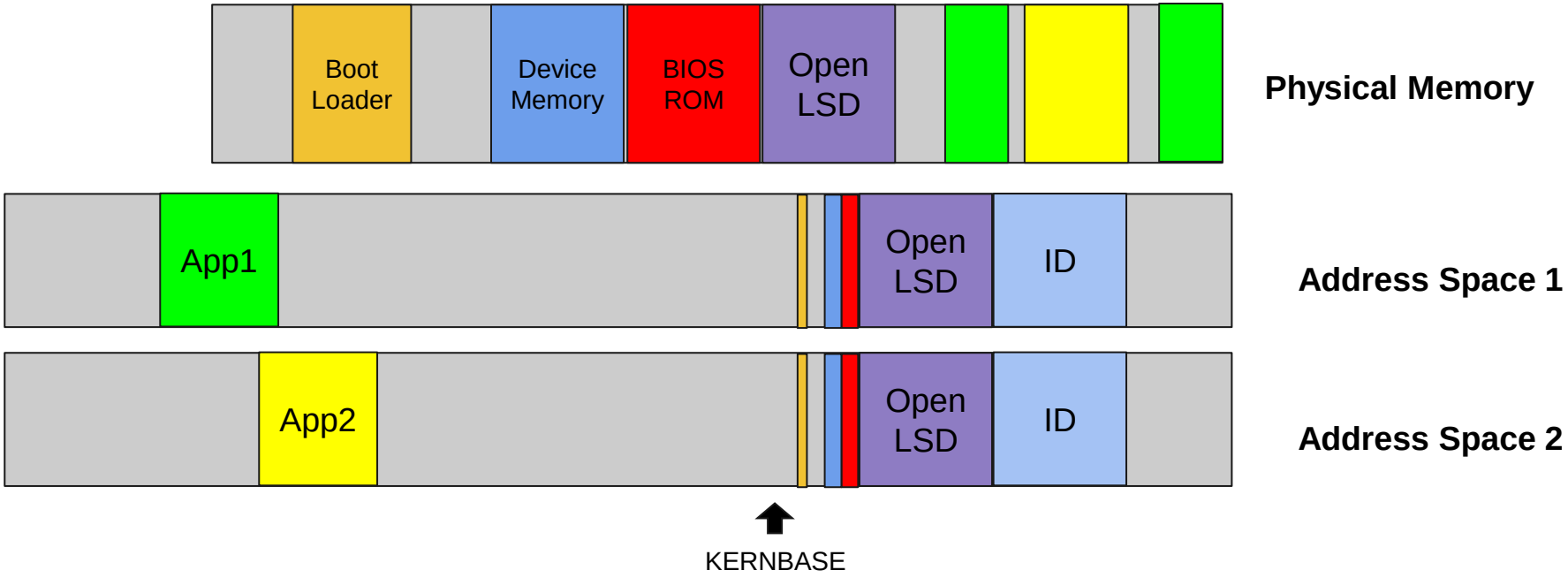
OpenLSD Right After Boot



OpenLSD
Basic Virtual Address Space

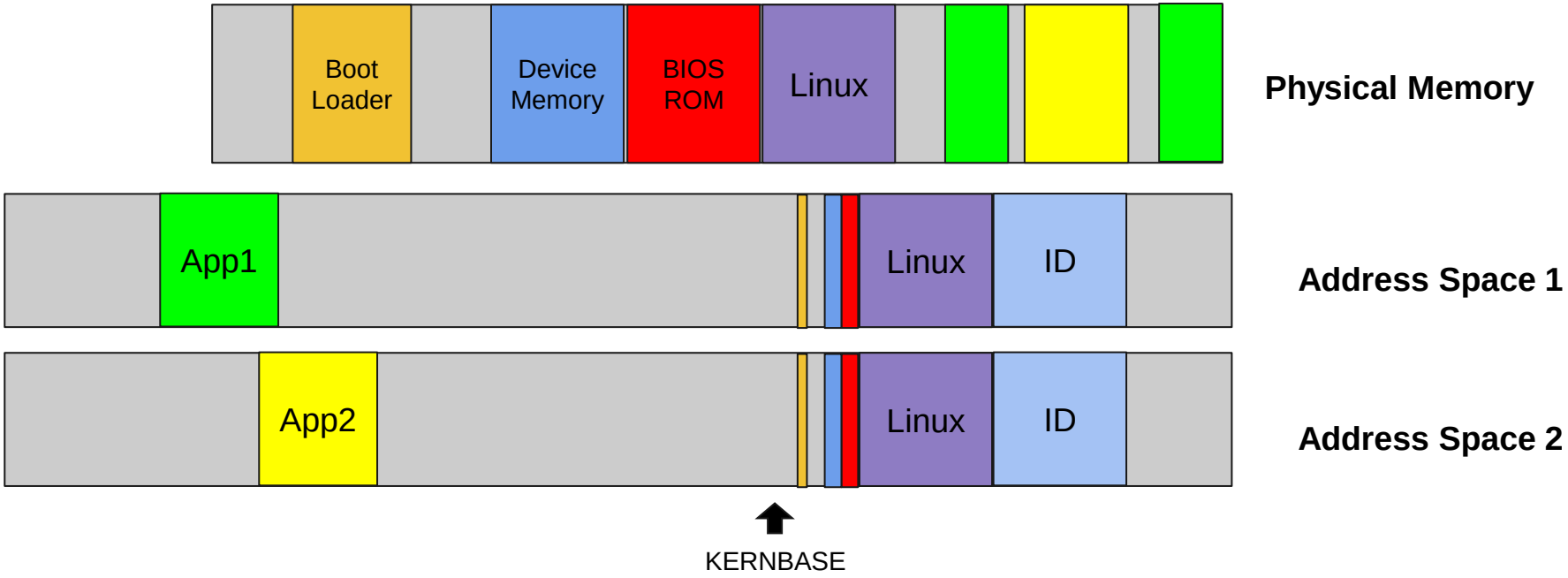


OpenLSD
Basic Virtual Address Space



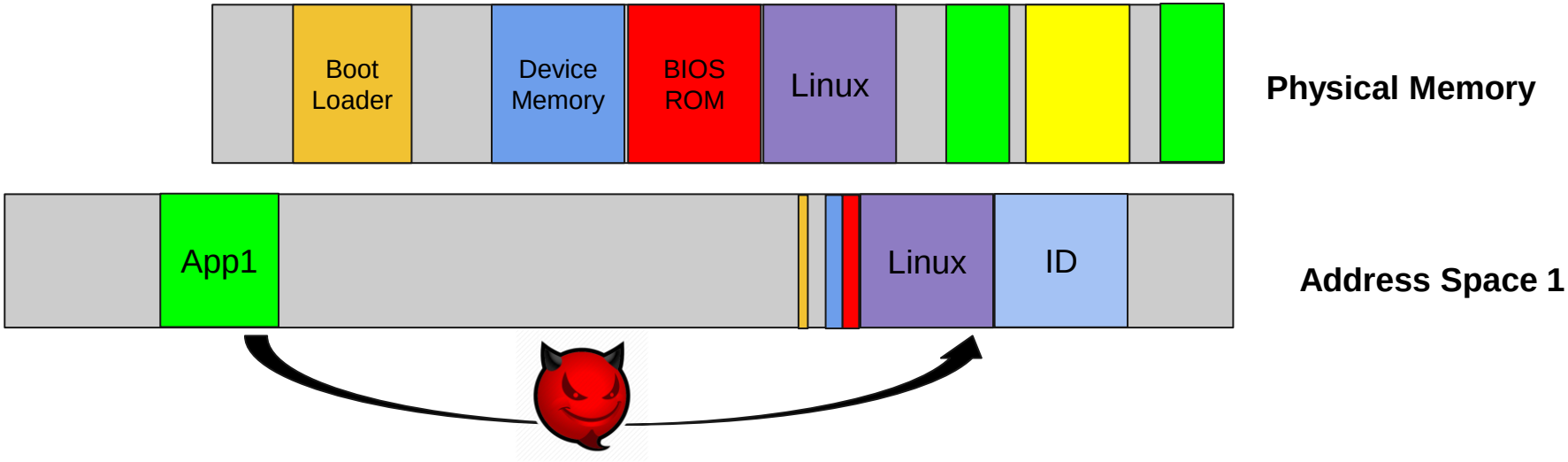
How can we make sure apps cannot modify the kernel data?

Linux Virtual Address Space (until 2018-1)



What happened in January 2018?

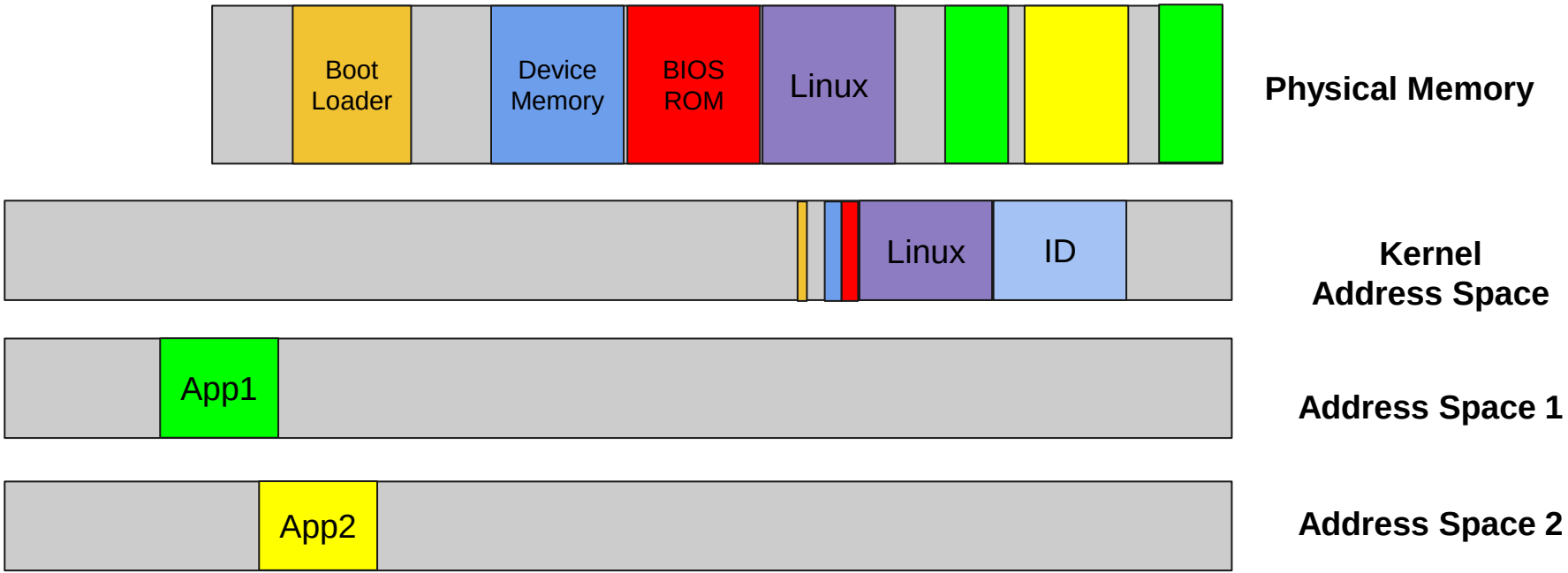
Intel CPU Vulnerability



Rogue Data Cache Load - branded as Meltdown

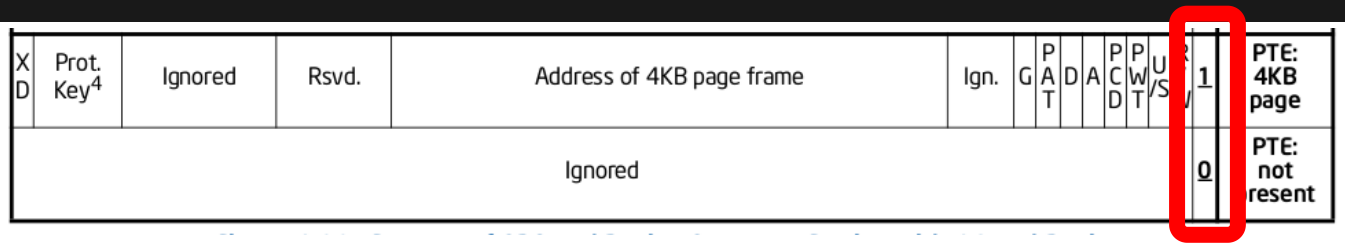
Bypasses the supervisor bit in PTEs → Arbitrary read

Current Linux Virtual Address Space Layout



More about this later

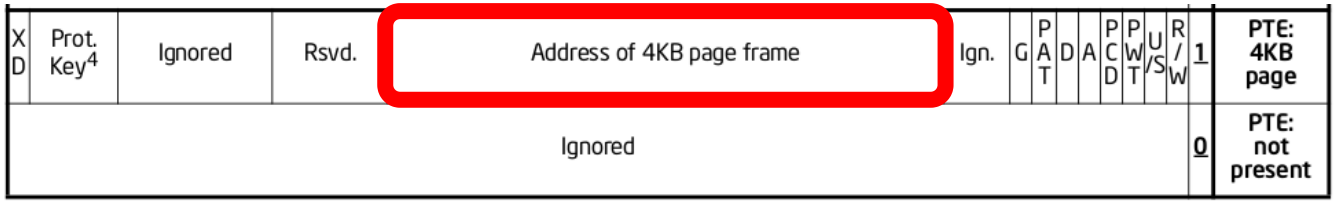
Unmapping and Present Bit



- Flip the present bit and the page is unmapped and not accessible anymore
  - L1TF (dubbed Foreshadow) bypasses the present bit
- Need to clear the page frame address on unmap

More about this later

Page Table Address Bits



- Need physical page address to access a page

MDS (dubbed RIDL) bypasses the address bits

- Need to flush CPU buffers containing recent data

More about this later

Dynamic Page Table Management (Post Boot)

- Address spaces are flexible
  - Need more memory? Map new page
  - Don’t need memory anymore? Unmap page
- What should your virtual address spaces look like? (policies)
- How would you do these things? (mechanisms)

Mechanisms

- Need to dynamically update page tables for each address space
- MMU uses updated page tables to find physical pages for the virtual pages

Page Table Walk

- Need to update page table pages
  - Wait… How can you access page table pages?
- First step: find page table entry
- Assumptions
  - x86_64, 4-level page table, 48-bit virtual address
  - Page table mapped into virtual memory
- Fun fact: no kernel privileges needed if granted access to page table

Page Table Walk

- Locate top-level page table
  - Read cr3 register (or process struct)
- Locate 2nd-level page table
  - Get virtual pointer to top-level page table
  - Use bits 39-47 of virtual address as index
  - Is the page table entry present? If no, then abort
- Locate 3rd-level page table
  - Get virtual pointer to 2nd-level page table
  - Use bits 30-38 of virtual address as index
  - Is the page table entry present? If no, then abort
- …
- Last page table entry has physical address of page

Page Table Mapping

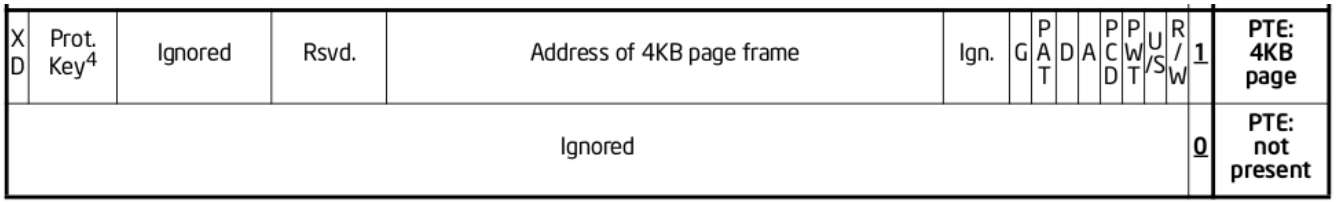
- Locate page table entry for virtual address to be mapped using page table walk
- Not present at any level
  - Allocate new page
  - Store its physical address in non-present entry
  - Continue with next level
- Store physical address to be mapped in final page table entry and mark present

Page Table Unmapping

- Locate page table entry for virtual address to be unmapped using page table walk
- Zero out final page table entry
  - Why zero?
  - Optional: also unmap intermediate page tables with no present entries left
- Free the page(s)

Permission Bits

- P: page faults (0) or present (1)
- R/W: read-only (0) or writable (1)
- U/S: supervisor-only (0) or user-accessible (1)
  - SMAP protection: 1 becomes user-only
- XD: execute allowed (0) or disabled (1)



Page Table Management in OpenLSD

Lab 2: On the same page (Page tables)

Page Table Management in Linux

- Done by kernel when necessary, and on demand from user using system calls (e.g., mmap, brk)
- Same code to handle different architectures
- Abstracted in 4 different levels
- The levels are folded if not necessary

Architecture	Bits used			
	PGD	PUD	PMD	PTE
i386	22-31			12-21
i386 (PAE mode)	30-31		21-29	12-20
x86-64	39-46	30-38	21-29	12-20

Page Table Allocation in Linux (mm/memory.c)

```
static pmd_t *walk_to_pmd(struct mm_struct *mm, unsigned long addr)
{
  pgd_t *pgd;
  p4d_t *p4d;
  pud_t *pud;
  pmd_t *pmd;

  pgd = pgd_offset(mm, addr);
  p4d = p4d_alloc(mm, pgd, addr);
  if (!p4d)
    return NULL;
  pud = pud_alloc(mm, p4d, addr);
  if (!pud)
    return NULL;
  pmd = pmd_alloc(mm, pud, addr);
  if (!pmd)
    return NULL;

  VM_BUG_ON(pmd_trans_huge(*pmd));
  return pmd;
}
```

Page Table Manipulation in Linux

```
/* include/linux/pgtable.h */
static inline pgd_t *pgd_offset_pgd(pgd_t *pgd, unsigned long address)
{
  return (pgd + pgd_index(address));
};

/* arch/x86/include/asm/pgtable.h */
static inline int pte_present(pte_t a)
{
  return pte_flags(a) & (_PAGE_PRESENT | _PAGE_PROTNONE);
}

static inline pte_t pte_mkwrite(pte_t pte)
{
  return pte_set_flags(pte, _PAGE_RW);
}
```

Page Table Removal in Linux (mm/memory.c)

```
void unmap_page_range(struct mmu_gather *tlb,
                      struct vm_area_struct *vma,
                      unsigned long addr, unsigned long end,
                      struct zap_details *details)
{
  pgd_t *pgd;
  unsigned long next;

  BUG_ON(addr >= end);
  tlb_start_vma(tlb, vma);
  pgd = pgd_offset(vma->vm_mm, addr);
  do {
    next = pgd_addr_end(addr, end);
    if (pgd_none_or_clear_bad(pgd))
      continue;
    next = zap_p4d_range(tlb, vma, pgd, addr, next, details);
  } while (pgd++, addr = next, addr != end);
  tlb_end_vma(tlb, vma);
}
```

Isn’t All This Freaking Expensive?

- For **every** virtual memory address access the MMU needs to walk page tables
  - E.g., on x86_64, 4 page table walks. We need 5 memory access to resolve one virtual address
- What can we do here?
  - Do caching...

Translation Lookaside Buffer (TLB)

- Result of page table translation:
  virtual address *x* is at physical address *y*
- Approach for memory access
  - Look up *x* in TLB
  - Hit: no need to consult page table
  - Miss: page table walk, cache $x \rightarrow y$ in TLB
- Why will this work?
  - Temporal and spatial locality

TLB Implementation

- TLB affects performance significantly
- Needs to be super fast
- Small number of sets (or fully-associative)
- Complex and fast → Small size

Table 2-5. TLB Parameters of the Skylake Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

TLB Management

- TLB is a cache
- Caches need to remain consistent
- When to flush (part of) the TLB?
  - When switching to a new address space.
    Automatic in x86 when reloading cr3
  - When deleting/updating page table entries.
    Explicit flush (invlpg on x86)
  - Things get fun on multicores
- What if TLB not flushed when necessary?

TLB Scalability

- Memory has become plentiful
  - Your browser uses gigabytes of memory
  - Skylake TLB coverage with 4KB pages:
    6MB = 1536 (L2TLB) × 4KB
  - And this gets even worse with more memory
- How do you fix this?
  - Translation caches
  - Caching page table pages
  - Bigger pages

Translation Caches

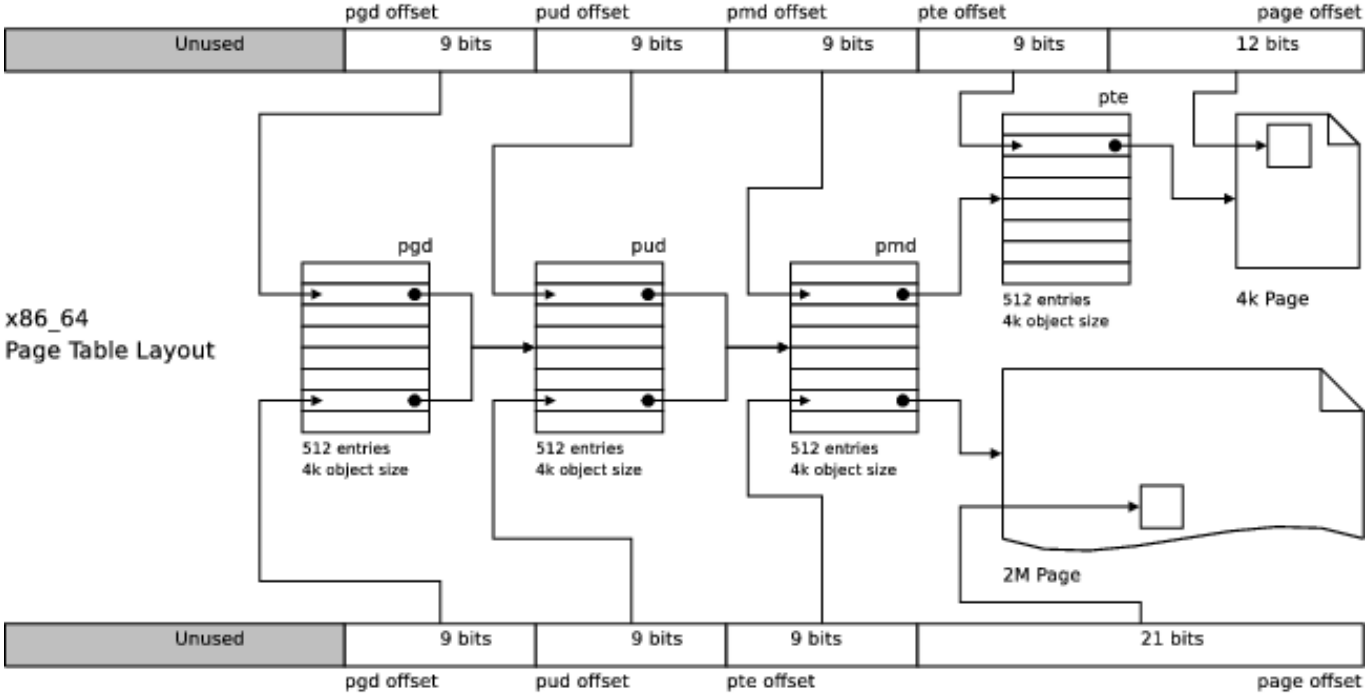
- TLB is tagged with all bits of the virtual addr
- Translation caches tagged with part of virt addr
  - Followed by a partial page walk
- Managed transparently by hardware

Larger Pages

- Large chunks of physical memory that are contiguous and aligned
  - 4KB pages are aligned to 4KB
  - 2MB pages are aligned to 2MB
  - …
- Major performance benefit
  - Reduced page table walks (4 levels → 2 or 3 levels)
  - TLB entries can cover more memory

Larger Pages

- Hardware support at most page table levels
  - On x86_32: 4KB and 4MB
  - On x86_64: 4KB, 2MB, 1GB, (512GB not supported)
- Downsides
  - Wastage
  - Complex management



Four-level Page Tables (x86_64)

Page Tables and Sanity

- Writing kernel code is hard → bugs
- Previous bonus assignment:
  sanity checks with poison values
  - Pros: easy to implement
  - Cons: page frame wastage

Page Tables and Sanity

- Instead, use page tables to do the job
  - E.g., Leave holes in the identity map (i.e., guard pages)
- Pros: no wastage
- Cons: more complex management

OpenLSD
Bonus Assignment

Use page tables to improve sanity checks
in the OpenLSD kernel

Page Tables and Security

Bad guys are exploiting your kernels

Kernel Exploitation

- Find a kernel vulnerability
  - 20M lines of code
  - Highly complex concurrent code
- Find known locations in the kernel space
  - Physmap in Linux is a prime target
    (in OpenLSD the “Remapped Physical Memory” area)
  - Kernel base address is another (for jump targets)
- Trigger vulnerability (potentially remotely)
  - Corrupt data in known locations
    - uid=0(root) gid=0(root) groups=0(root)
  - Diverge control flow

Kernel Address Space Layout Randomization (KASLR)

- Randomizes sections kernel address space
  - Harder for attackers to exploit the kernel
  - Bruteforce → plenty of kernel crashes
  - Leak kernel pointers → secondary vulnerability
  - Side-channel attacks → complicated

OS	Types	Entropy	#Slots	Address Range	Align Base	Align Size
Linux	Kernel	6 bits	64	0xffffffff80000000 – 0xffffffffc0000000	0x1000000	16 MB
	Modules	10 bits	1,024	0xffffffffc0000000 – 0xffffffffc0400000	0x1000	4 KB
Windows	Kernel	*13 bits	8,192	0xfffff80000000000 – 0xfffff80400000000	0x200000	2 MB
	Modules	*13 bits	8,192	0xfffff80000000000 – 0xfffff80400000000	0x200000	2 MB
OS X	Kernel	8 bits	256	0xffffff8000000000 – 0xffffff8020000000	0x200000	2 MB

KASLR Implementation

- Entropy limited to simplify memory management
- Remains the same until reboot
- Random slot chosen early in boot,
  kernel is mapped there
- Random mapping translates to
  different slots in the page table pages

OpenLSD
Bonus Assignment

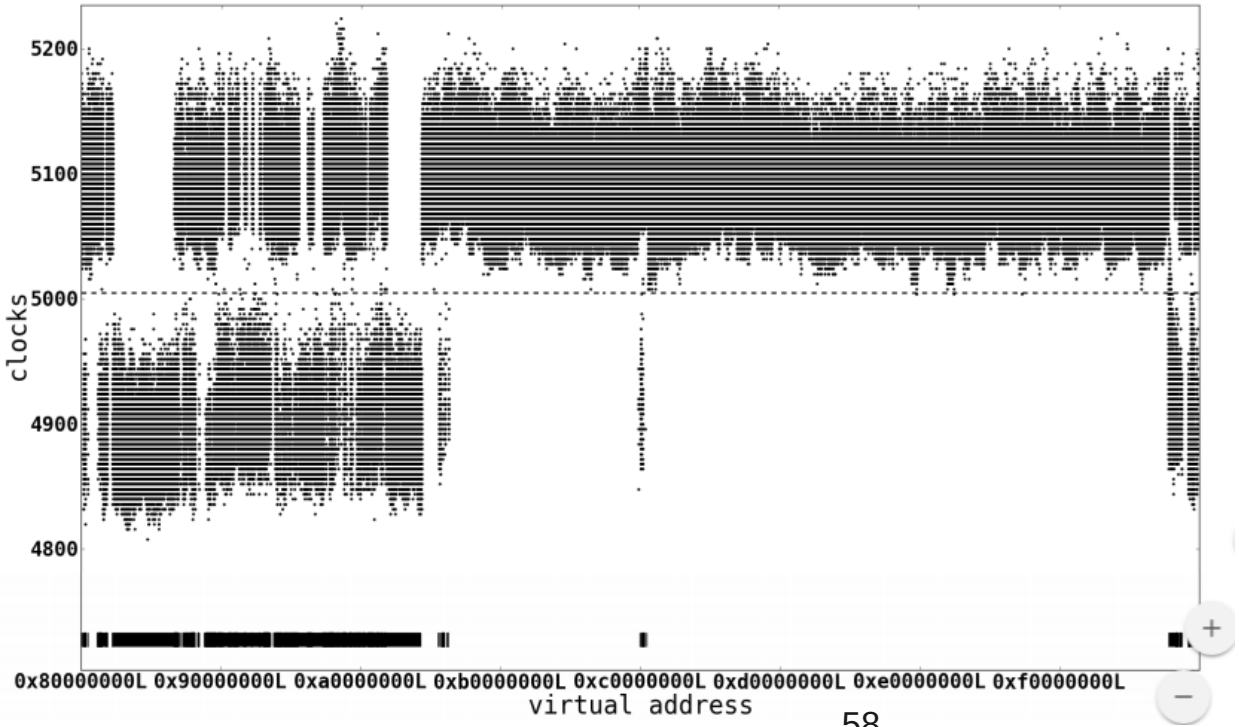
Support KASLR for OpenLSD kernel and its “Remapped Physical Memory”

Breaking KASLR with Side Channels (pre-2018)

- Many examples in academia (TSX, prefetch)
- Attacker wants to know whether address *v* is mapped in the kernel from user space

1. Flush the TLB
2. T1 = Time accessing *v* from user space
   a. TLB miss. Entry is created if *v* is mapped in the kernel
   b. Page fault
3. T2 = Time accessing *v* from user space
   a. TLB miss if *v* is not mapped. Otherwise TLB hit.
   b. Page fault[22]
4. T2 < T1 → *v* is mapped in the kernel

How well does it work?



Something to Think About

How do you break your OpenLSD KASLR implementation?

References

1. Intel® 64 and IA-32 Architectures Optimization Reference Manual
2. Four-level page tables merged, https://lwn.net/Articles/117749/
3. Thomas W. Barr, Alan L. Cox, and Scott Rixner. “Translation Caching: Skip, Don’t Walk (The Page Table)”, In *ISCA*, 2010.
4. Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU.” In *NDSS*, 2017.
5. Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think.”, In USENIX Sec, 2018.
6. Ben Gras, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida, “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”, USENIX Sec 2018.
7. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, “Meltdown: Reading Kernel Memory from User Space”, USENIX Sec, 2018.
8. Gorman, Mel. *Understanding the Linux Virtual Memory Manager*, 2007.
9. Yeongjin Jang, Sangho Lee, and Taesoo Kim, Breaking Kernel Address Space Layout Randomization with Intel TSX, in *CCS*, 2016.
10. Ralf Hund, Carsten Willems, and Thorsten Holz, *Practical Timing Side Channel Attacks against Kernel Space ASLR*, in S&P, 2013.
11. Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms.” In CCS, 2016.
12. Hugetlbpage.txt, https://lwn.net/Articles/375098/