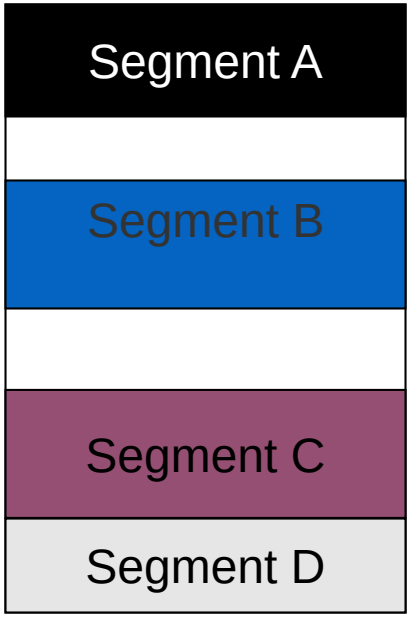
Paging in Virtual Memory

Nima Honarmand

(Based on slides by Prof. Andrea Arpaci-Dusseau)

Problem: Fragmentation

- Definition: Free memory that can't be usefully allocated
- Why?
  - Free memory (hole) is too small and scattered
  - Rules for allocating memory prohibit using this free space
- Types of fragmentation
  - **External**: Visible to allocator (e.g., OS)
  - **Internal**: Visible to requester (e.g., if must allocate at some granularity)

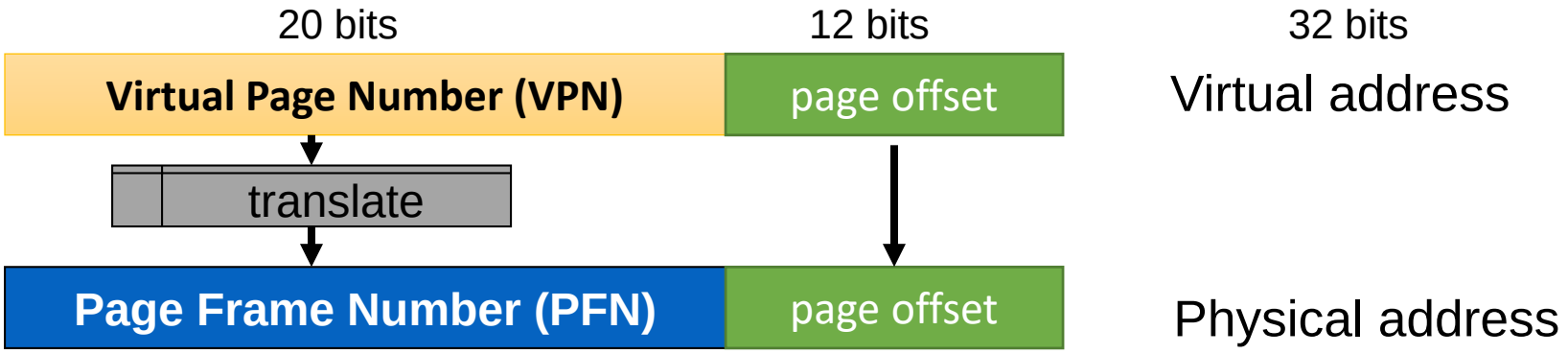


Paging

- Goal: mitigate fragmentation by
  - Eliminating the requirement for segments to be contiguous in physical memory
  - Allocating physical memory in fixed-size fine-grained chunks
- Idea: divide both address space and physical memory into **pages**
  - For address space, we refer to it as a **Virtual Page**
  - For physical memory, we refer to it as a **Page Frame**
- Allow each Virtual Page to be mapped to a Page Frame independently

Translation of Page Addresses

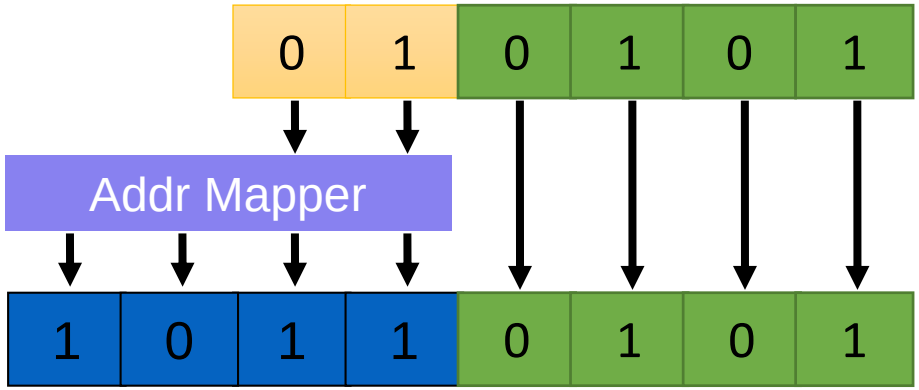
- How to translate virtual address to physical address?
  - High-order bits of address designate page number
    - In a virtual address, it is called ***Virtual Page Number*** (**VPN**)
    - In a physical address, it is called ***Page Frame Number*** (**PFN**) or ***Physical Page Number*** (**PPN**)
  - Low-order bits of address designate offset within page



- How does format of address space determine number of pages and size of pages?

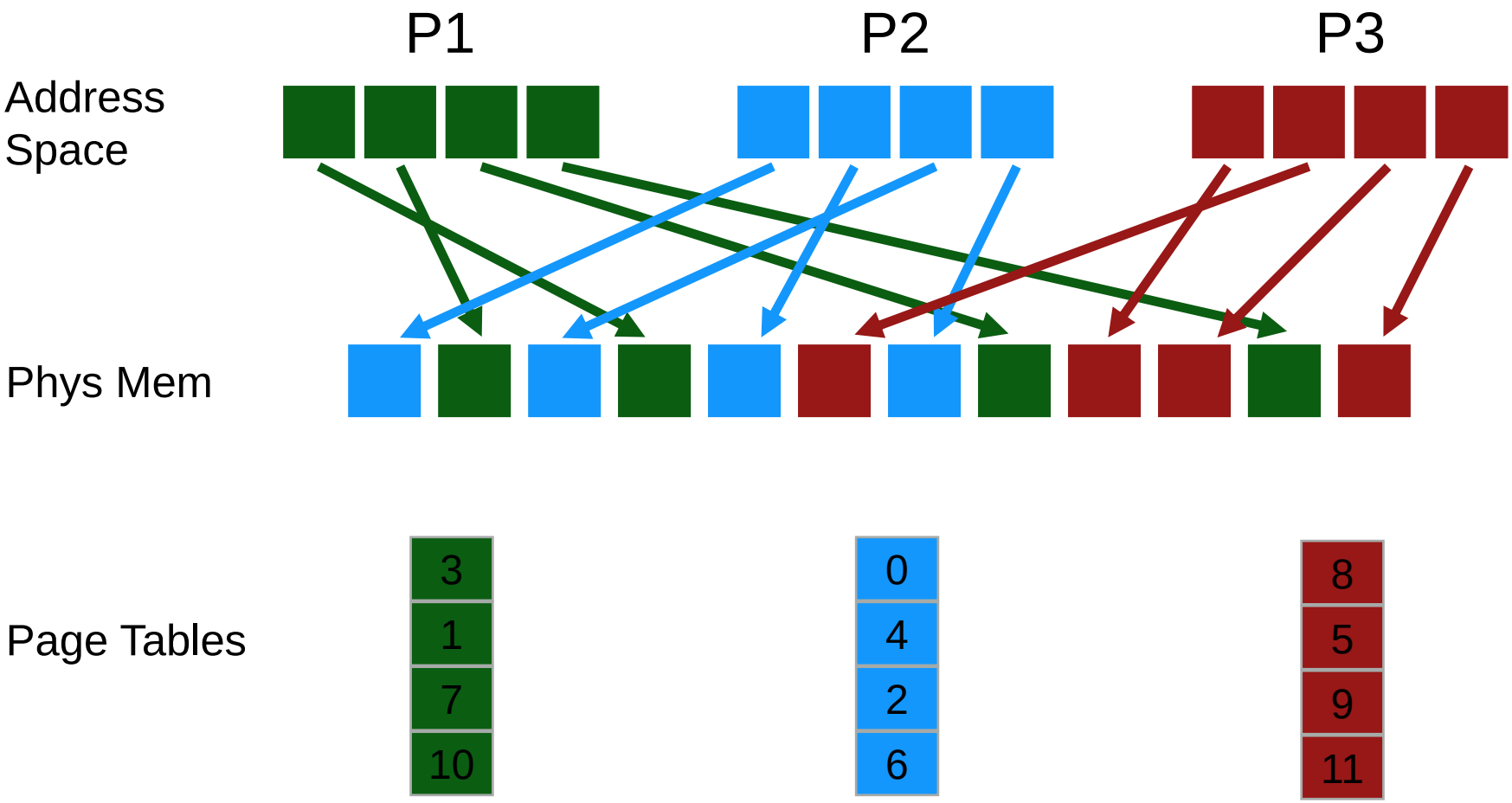
How to Translate?

Note: number of bits in virtual address does not need to equal number of bits in physical address



- How should OS translate VPN to PFN?
  - For segmentation, OS used a formula (e.g., phys_addr = virt_offset + base_reg)
  - For paging, OS needs more general mapping mechanism
- What data structure is good?
  - Old answer: a simple array — called a **Page Table**
  - One entry per virtual page in the address space
  - VPN is the entry index; entry stores PFN
  - Each entry called a **Page Table Entry** (**PTE**)

Example: Fill in the Page Tables



Where Are Page Tables Stored?

- How big is a typical page table?
  - Assume 32-bit address space, 4KB pages and 4 byte PTEs

- Answer: 2 ^ (32 - log(4KB)) * 4 = 4 MB
  - Page table size = Num entries * size of each entry
  - Num entries = Num virtual pages = 2^(bits for VPN)
  - Bits for VPN = 32 – number of bits for page offset = 32 – log(4KB) = 32 – 12 = 20
  - Num entries = 2^20 = 1 MB
  - Page table size = Num entries * 4 bytes = 4 MB

- Implication: Too big to store on processor chip → Store each page table in memory
- Hardware finds page table base using a special-purpose register (e.g., CR3 on x86)

- What happens on a context-switch?
  - PCB contains the address of the process's PT
  - OS changes contents of page table base register to the PT of the newly scheduled process

Other PTE Info

- What other info is in PTE besides PFN?
  - **Valid** bit
  - **Protection** bit
  - **Present** bit (needed later)
  - **Referenced** bit (needed later)
  - **Dirty** bit (needed later)
- Page table entries are just bits stored in memory
  - Agreement between HW and OS about interpretation

Example: Mem Access w/ Segments

%rip = 0x0010

```
0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100
```

Seg	Base	Bounds
0	0x4000	0xfff
1	0x5800	0xfff
2	0x6800	0x7ff

Assume segment selected by 2 virtual addr MSBs

Physical Memory Accesses?

1) Fetch instruction at virtual addr 0x0010
   - Physical addr: 0x4010

   Exec, load from virtual addr 0x1100
   - Physical addr: 0x5900

2) Fetch instruction at virtual addr 0x0013
   - Physical addr: 0x4013

   Exec, no mem access

3) Fetch instruction at virtual addr 0x0019
   - Physical addr: 0x4019

   Exec, store to virtual addr 0x1100
   - Physical addr: 0x5900

Total of 5 memory references (3 instruction fetches, 2 movl)

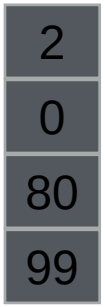
Example: Mem Access w/ Pages

%rip = 0x0010

```
0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100
```

Assume PT is at phys addr **0x5000**
Assume PTE's are **4 bytes**
Assume **4KB pages**

Simplified view of page table



Physical Memory Accesses with Paging?

1) Fetch instruction at virtual addr 0x0010; VPN?

- Access page table to get PFN for VPN 0
- Mem ref 1: 0x5000
- Learn VPN 0 is at PFN 2
- Fetch instruction at 0x2010 (Mem ref 2)

Exec, load from virtual addr 0x1100; VPN?

- Access page table to get PFN for VPN 1
- Mem ref 3: 0x5004
- Learn VPN 1 is at PFN 0
- movl from 0x0100 into %edi (Mem ref 4)

Page Table is Slow!!! Doubles # mem accesses (10 vs. 5)

Advantages of Paging

- Easily accommodates transparency, isolation, protection and sharing
- No external fragmentation
- Fast to allocate and free page frames
  - Alloc: No searching for suitable free space; pick the first free page frame
  - Free: Doesn't have to coallesce with adjacent free space; just add to the list of free page frames
  - Simple data structure (bitmap, linked list, etc.) to track free/allocated page frames

Disadvantages of Paging

- Internal fragmentation: Page size may not match size needed by process
  - Wasted memory grows with larger pages
  - Tension?
- **Additional memory reference to page table** → Very inefficient; high performance overhead
  - Page table must be stored in memory
  - MMU stores only base address of page table
  - Solution: TLBs
- **Storage for page tables may be substantial**
  - Simple page table: Requires PTE for all pages in address space
    - Entry needed even if page not allocated
  - Page tables must be allocated contiguously in memory
  - Solution: alternative page table structures

Mitigating Performance Problem Using TLBs

Translation Steps

H/W: for each mem reference:

(cheap) 1. extract VPN (virt page num) from VA (virt addr)
(cheap) 2. calculate addr of PTE (page table entry)
(expensive) 3. read PTE from memory
(cheap) 4. extract PFN (page frame num)
(cheap) 5. build PA (phys addr)
(expensive) 6. read contents of PA from memory into register

Which Steps are expensive?

Which expensive step will we avoid in today? Step (3)

Example: Array Iterator

```
int sum = 0;

for (i=0; i<N; i++){
      sum += a[i];

}
```

Assume ‘a’ starts at 0x3000

Ignore instruction fetches

What virtual addresses?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

…

What physical addresses?

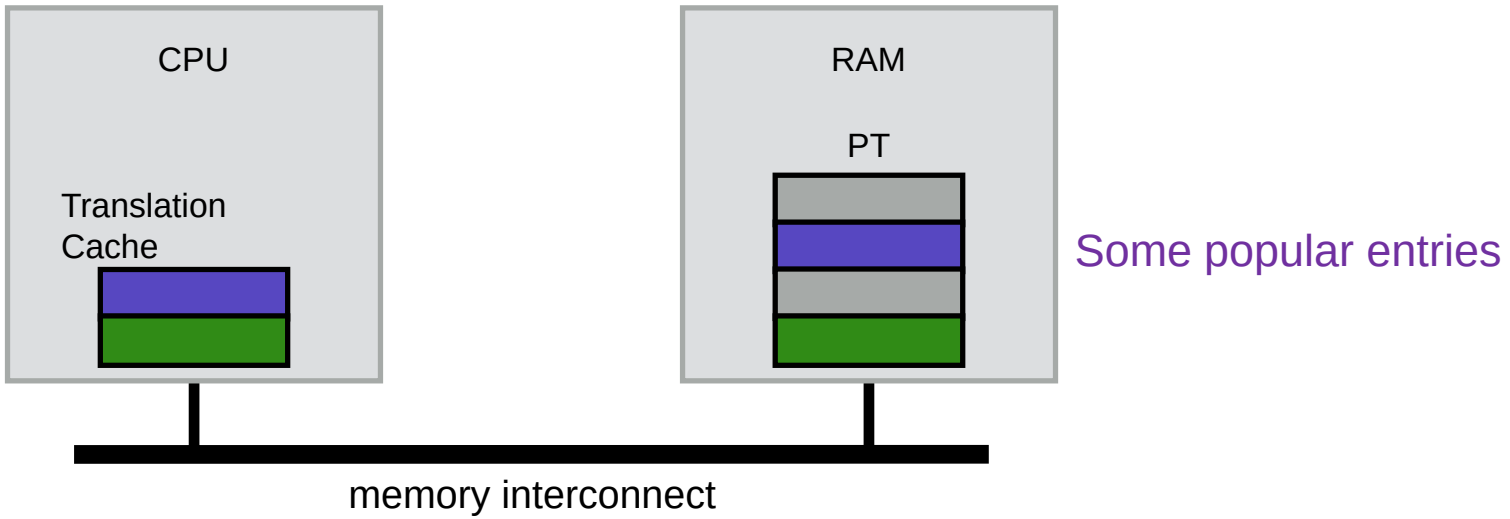
load 0x100C
load 0x7000
load 0x100C
load 0x7004
load 0x100C
load 0x7008
load 0x100C
load 0x700C

Aside: What can you infer?

- ptbr: 0x1000; PTE 4 bytes each
- VPN 3 -> PFN 7

Observation:
Repeatedly access same PTE because program repeatedly accesses same virtual page

Strategy: Cache Page Translations



TLB: Translation Lookaside Buffer

(yes, a poor name!)

TLB Entry

- TLB is a cache of page table
- Each TLB entry should cache all the information in a PTE
- It also needs to store the VPN as a tag
  - To be used when the hardware searches TLB for a particular VPN

TLB Entry

Tag (Virtual Page Number)	**Page Table Entry** (PFN, Permission Bits, Other flags)

Array Iterator w/ TLB

```
int sum = 0;
for (i = 0; i < 2048; i++){
        sum += a[i];
}
```

Assume following virtual address stream:

load 0x1000

load 0x1004

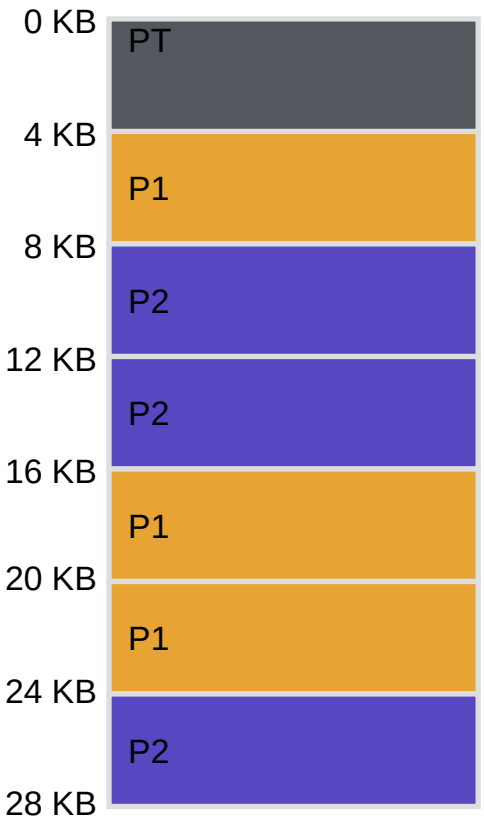
load 0x1008

load 0x100C

…

What will TLB behavior look like?

Array Iterator w/ TLB



P1 Page Table

1	5	4	…
0	1	2	3

CPU's TLB

Valid	VPN	PFN
1	1	5
1	2	4

Virt	Phys
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB hit) load 0x5004
load 0x1008	(TLB hit) load 0x5008
load 0x100c	(TLB hit) load 0x500C
…	…
load 0x2000	load 0x0008 load 0x4000
load 0x2004	(TLB hit) load 0x4004

TLB Performance

Calculate miss rate of TLB for data:
TLB misses / # TLB lookups

TLB lookups?
= number of accesses to `a` = 2048

TLB misses?
= number of unique pages accessed
= 2048 / (elements of 'a' per 4K page)
= 2K / (4K / sizeof(int)) = 2K / 1K
= 2

Miss rate?
2/2048 = 0.1%

Hit rate? (1 – miss rate)
99.9%

```
int sum = 0;
for (i = 0; i < 2048; i++) {
        sum += a[i];
}
```

Would hit rate get better or worse with smaller pages?
Answer: Worse

TLB & Workload Access Patterns

- Sequential array accesses almost always hit in TLB
  - Very fast!
- What access pattern will be slow?
  - Highly random, with no repeat accesses

Workload A

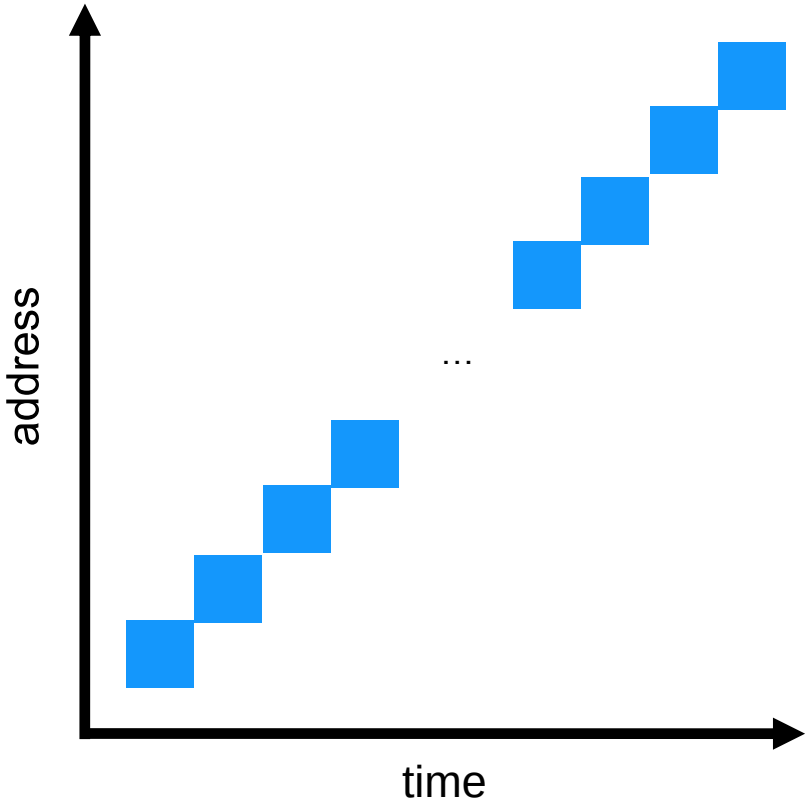
```
int sum = 0;

for (i=0; i<2000; i++) {
        sum += a[i];

}
```

Workload B

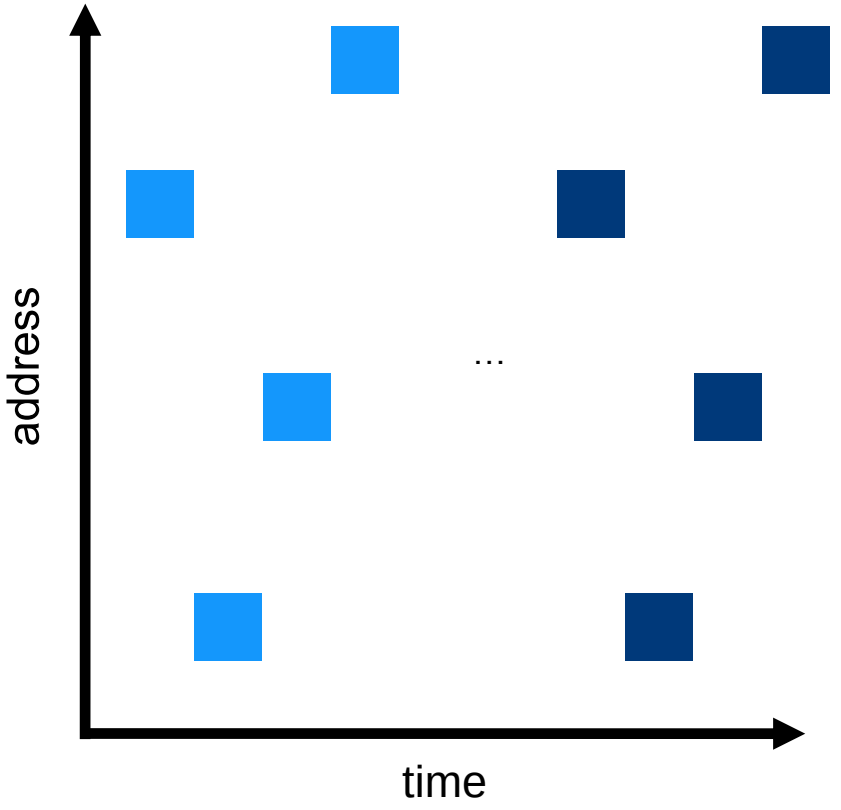
```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
```

Workload Access Patterns

Sequential Accesses
(Good for TLB)



Random Accesses
(Bad for TLB)



TLB Performance

- How can system improve TLB performance (hit rate) given fixed number of TLB entries?

- Increase page size
  - Fewer unique page translations needed to access same amount of memory

- What is the drawback of large pages?
  - Increased internal fragmentation
  - Tradeoffs, tradeoffs

- Most processors support multiple different page sizes
  - In 32-bit x86, 4KB and 2MB
  - In 64-bit x86, 4KB, 2MB and 1GB
  - Programmer (user) makes the choice since they know their program

- **TLB Reach**: Number of TLB entries * Page Size

TLBs and Context Switches

- What happens if a process uses cached TLB entries from another process?

Solutions?

1) Flush TLB on each switch
   - Could be costly; lose all recently cached translations

2) Track which entries are for which process
   - Address Space IDentifier (ASID)
   - When loading a TLB entry, tag it with the ASID of the current process (in addition to the VPN)
   - When reading TLB, check both VPN and ASID

TLB Performance

- Context switches are expensive
- Even with ASID, other processes “pollute” TLB
  - Discard process A’s TLB entries for process B’s entries
- Architectures often have multiple TLBs and multi-level TLBs
  - Level-1: 1 TLB for data, 1 TLB for instructions (smaller, say 64 entries; fast)
  - Level 2: combined or separate inst or data TLBs (larger, say 512 entries; slower)
  - On a Level-1 TLB miss, first check Level-2 TLB before checking the page table

HW and OS Roles

- Who Handles TLB MISS? **H/W** or **OS**?
  - Answer: both are possible
- **H/W**: CPU must know where the current page table is
  - CR3 register on x86
  - Page table structure fixed and determined by processor designer
  - HW "walks" the page table and fills TLB
- **OS**: CPU traps into OS upon TLB miss
  - "Software-managed" TLB
  - OS interprets page tables as it chooses
  - Modifying TLB entries is privileged
    - otherwise what could process do?

TLB Shootdown

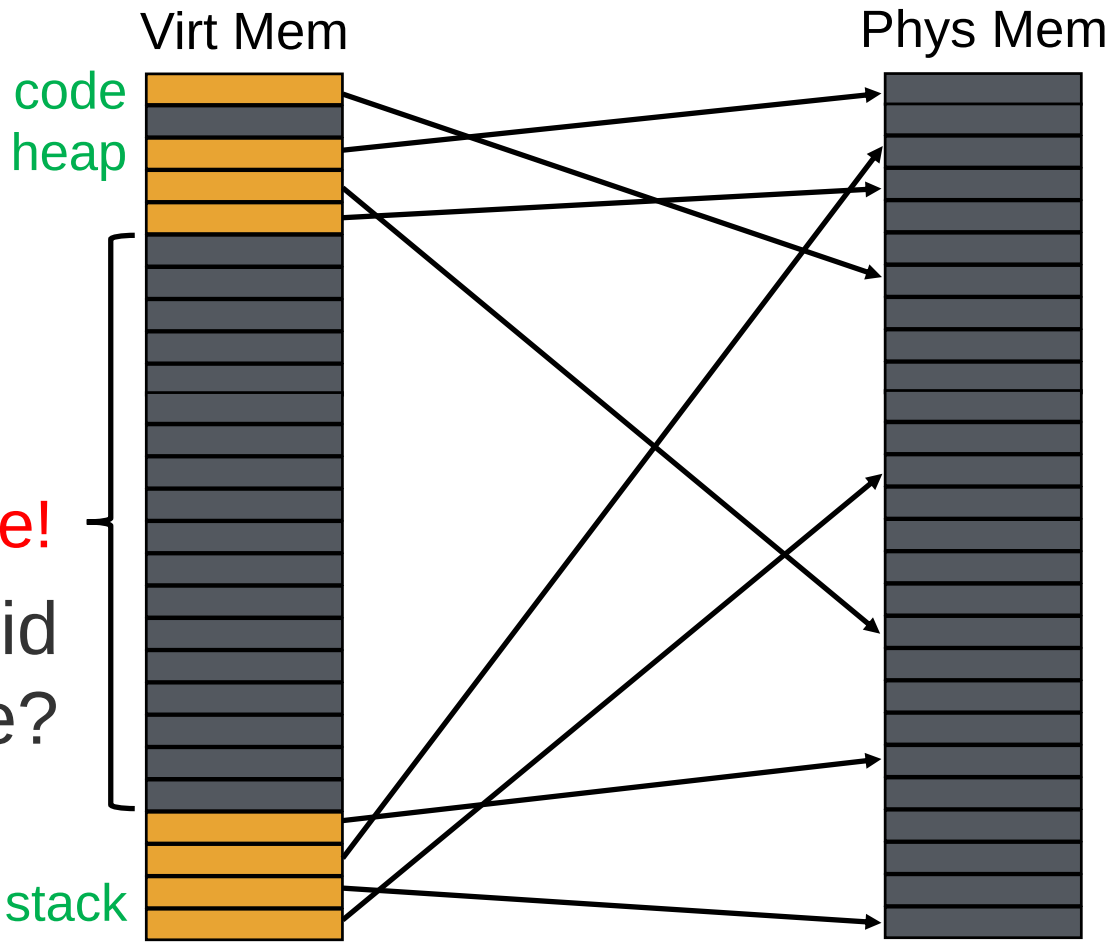
- What happens if the OS changes some virtual → physical mappings in a page table? Or protection flags for a page?
- TLB entries cached on the processor become stale
  - Dangerous if processor keeps using the stale mappings
- The OS should invalidate/update affected TLB entries
  - In software-managed TLB, the OS can change the TLB entry when changing the Page Table
  - In hardware-managed TLB, the OS has to invalidate the TLB entry
    - MMU hardware will reload the TLB entry from the page table upon the next access
    - See `invlpg` instruction in x86

Summary: TLB

- Pages are great, but accessing page tables for every memory access is slow
- Cache recent page translations → TLB
  - Hardware performs TLB lookup on every memory access
- TLB performance depends strongly on workload
  - Sequential workloads perform well
  - Workloads with temporal locality can perform well
  - Increase TLB reach by increasing page size
- In different systems, hardware or OS handles TLB misses
- TLBs increase cost of context switches
  - Flush TLB on every context switch
  - Add ASID to every TLB entry

Better Page Tables

Why Are Page Tables so Large?



Solution

- Use more complex page tables, instead of just big array
- With software-managed TLB, any data structure is possible
  - Hardware looks for VPN in TLB on every memory access
  - If a TLB miss, trap into OS
  - OS finds VPN → PFN translation in its page table, and installs it in the TLB for future accesses
- With hardware-managed TLB, hardware dictates the page-table structure

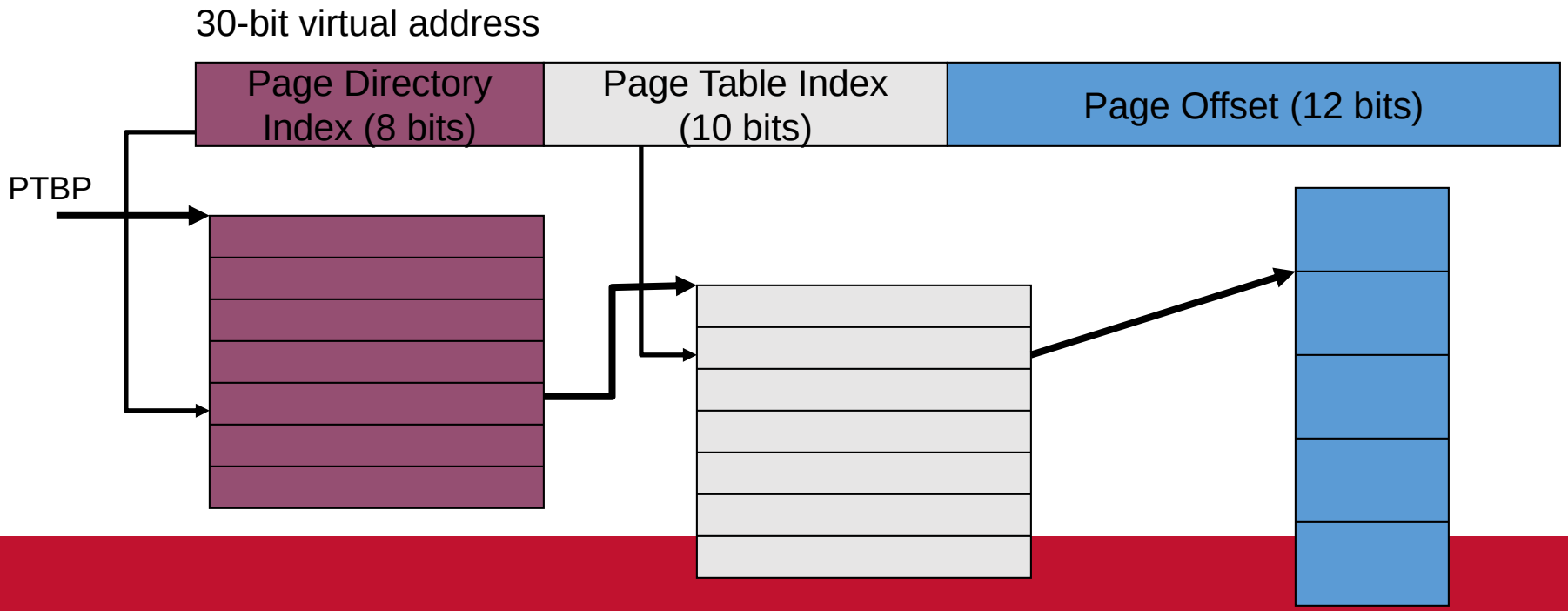
Common Approaches

- Inverted page tables
  - Use a hash-table to hash (PID, VPN) to PFN
  - Read more in the textbook
- Segmented page tables
  - Have a per-segment page table array
  - Read more in the textbook
- Multi-level page tables
  - Tree structure
  - Page the page tables
  - Page the page tables of page tables
  - And so on

Used in x86. We'll talk about this one. Read about the others in the book.

Two-Level Page Table

- Idea: Page the page tables
  - Creates two levels of page tables
    - First level called **Page Directory**
    - Second level called **Page Table**
  - Only allocate page tables for pages in use



Quiz: Two-Level PT

Page Directory

PFN	Valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

PT Page @ PFN 0x3

PFN	Valid
0x10	1
0x23	1
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

PT Page @ PFN 0x92

PFN	Valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

Translate 0x01ABC

0x23ABC

Translate 0x00000

0x10000

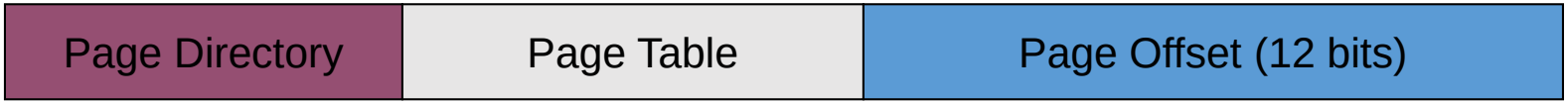
Translate 0xFEED0

0x55ED0

20-bit virtual address

Address Format for Two-Level Paging

30-bit virtual address



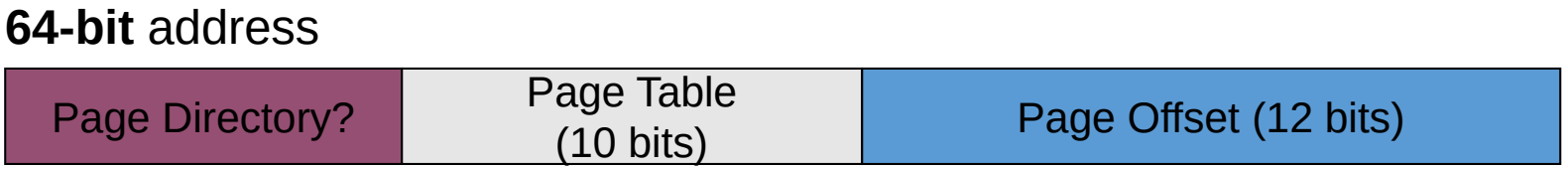
- How should logical address be structured?
  - How many bits for each paging level?
- Goal?
  - Each page table fits within a page
  - PTE size × number PTE = page size
    - Assume PTE size = 4 bytes
    - Page size = 2^{12} bytes = 4KB
    - 2^2 bytes * number PTE = 2^{12} bytes

    → # of PTEs = 2^{10}

  → # of bits for selecting inner page = **10**
- Remaining bits for outer page:
  - 30 – 10 – 12 = **8** bits

Problem w/ Two Levels

- Problem: page directories (outer level) may not fit in a page



- Solution: add more levels
  - Split page directories into pieces
  - Use a higher-level page dir to refer to the page dir pieces.

Quiz: Paging in 64-bit x86

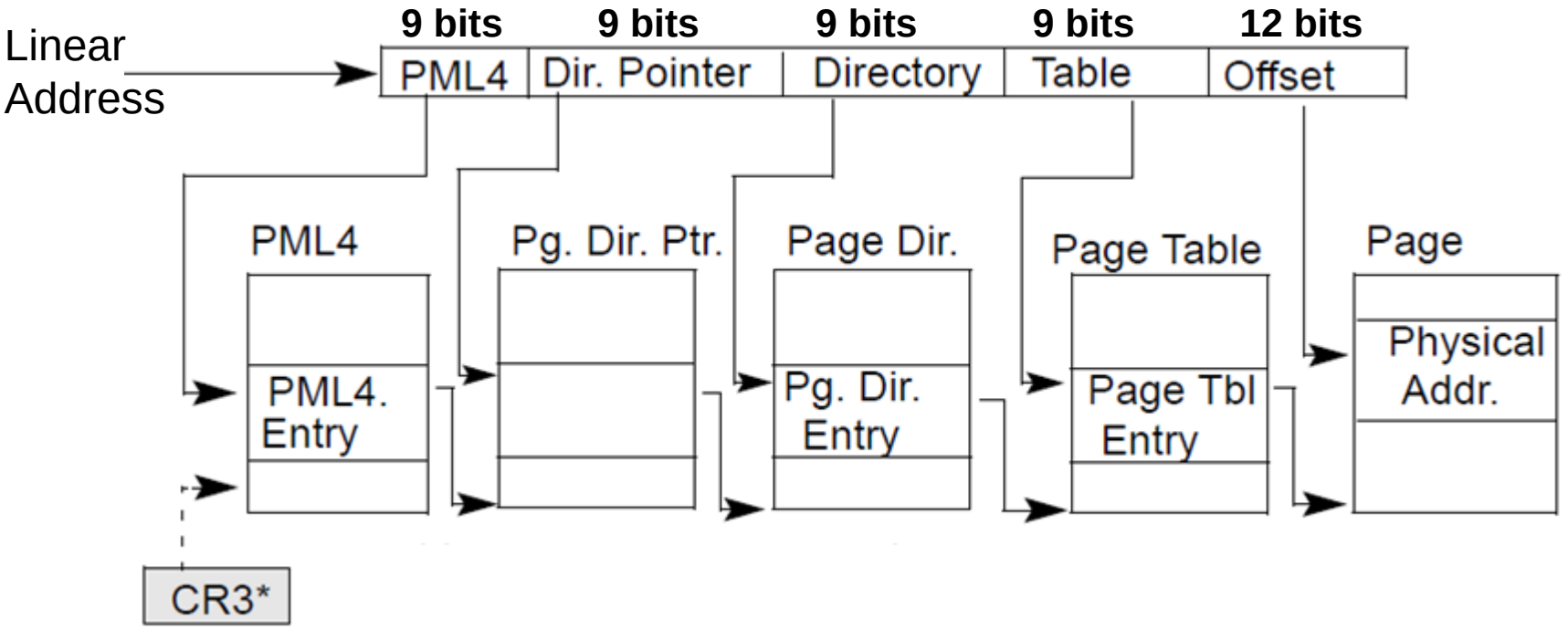
- Virtual addresses are 48 bits
- Physical addresses are 52 bits
- Page size is 4KB

- What is PTE size?
  - PFN is 40 bits (52-12) so we need an 8B PTE

- How many PTEs per page?
  - 4KB/8B = 512 = 2^9 → 9 bits to index each level of page table tree

- How many levels in the page table tree?
  - 4 = (48 -12) / 9
  - Each level of the tree is indexed using 9 bits

64-bit x86 Page Table



- Level names (bottom up)
  - Page Table, Page Directory, Page Directory Pointer, PML4 (Page Map Level 4)
- Enables 4KB, 2MB and 1GB pages

Summary: Better Page Tables

- Problem: Simple linear page tables require too much contiguous memory
- Many options for efficiently organizing page tables
- If OS traps on TLB miss, OS can use any data structure
  - Inverted page tables (hashing)
- If Hardware handles TLB miss, page tables must follow specific format
  - Multi-level page tables used in x86 architecture
  - Each page table fits within a page