CS5460: Operating Systems

Lecture 8: Segmentation, Paging, & TLBs

(Chapters 16, 17, 18, 19)

Slide Credit: Andrea Arpaci-Dusseau

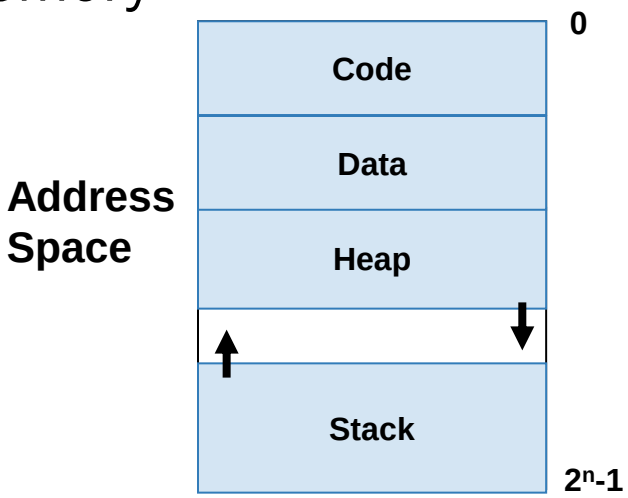
Assignment 2

- Due Tue Feb 16

Segmentation

- Divide address space into logical segments
  - Each corresponds to logical entity in address space
  - Code, Stack, Heap

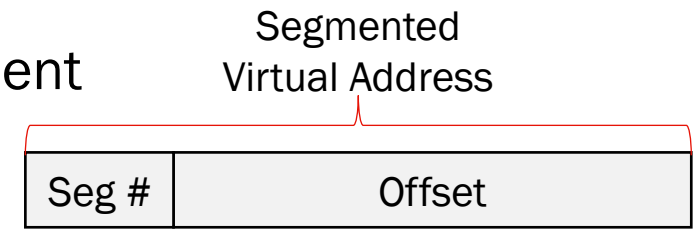
- Each segment can independently:
  - Be placed separately in physical memory
  - Grow and shrink
  - Be protected (separate read/write/execute protection bits)



Segmented Addressing

- Process specifies segment and offset within segment

- How does process designate a particular segment?
  - Explicit: use part of virtual address
  - Top bits select segment
  - Low bits indicate offset within segment



- What if small virtual address size, not enough bits?
  - Implicit: by type of memory reference
  - Special registers (%esp, %ebp – stack access, %eip – code access, %eax and other general purpose – data access)

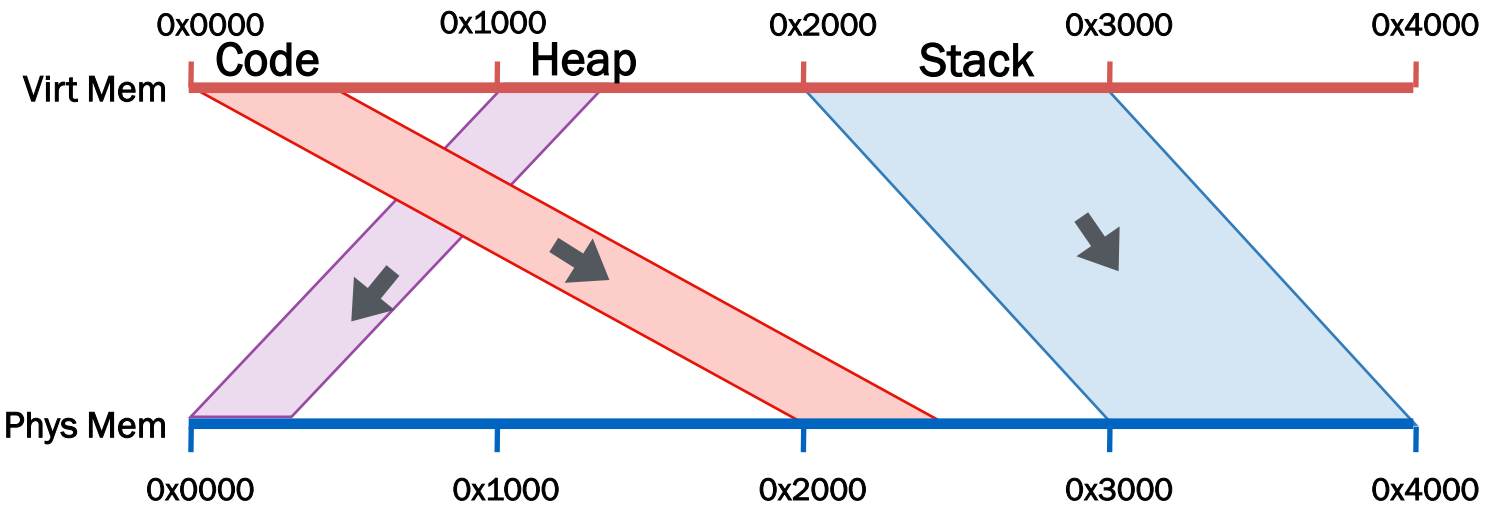
Segmentation Implementation

MMU has base/bound/permissions register **per segment**

- Example: 14 bit logical address, 4 segments
- How many bits for segment?
- How many bits for offset?

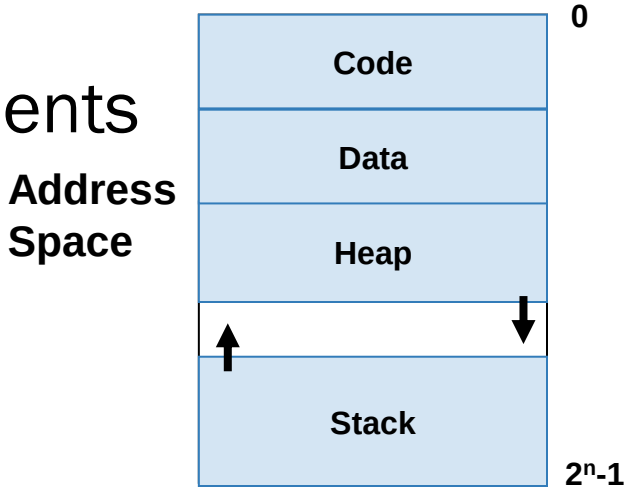
Seg	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

Physical address for these virtual addresses?
0x0240
0x1108
0x265c
0x3002



Advantages of Segmentation

- Enables sparse allocation of address space
  - Stack and heap can grow independently
  - Heap: if no free memory, then sbrk()
  - Stack: kernel recognizes reference just outside legal segment and extends stack implicitly, if possible
- Protections for different segments
  - Read-only status for code
- Enables sharing of selected segments
- Supports dynamic relocation of each segment



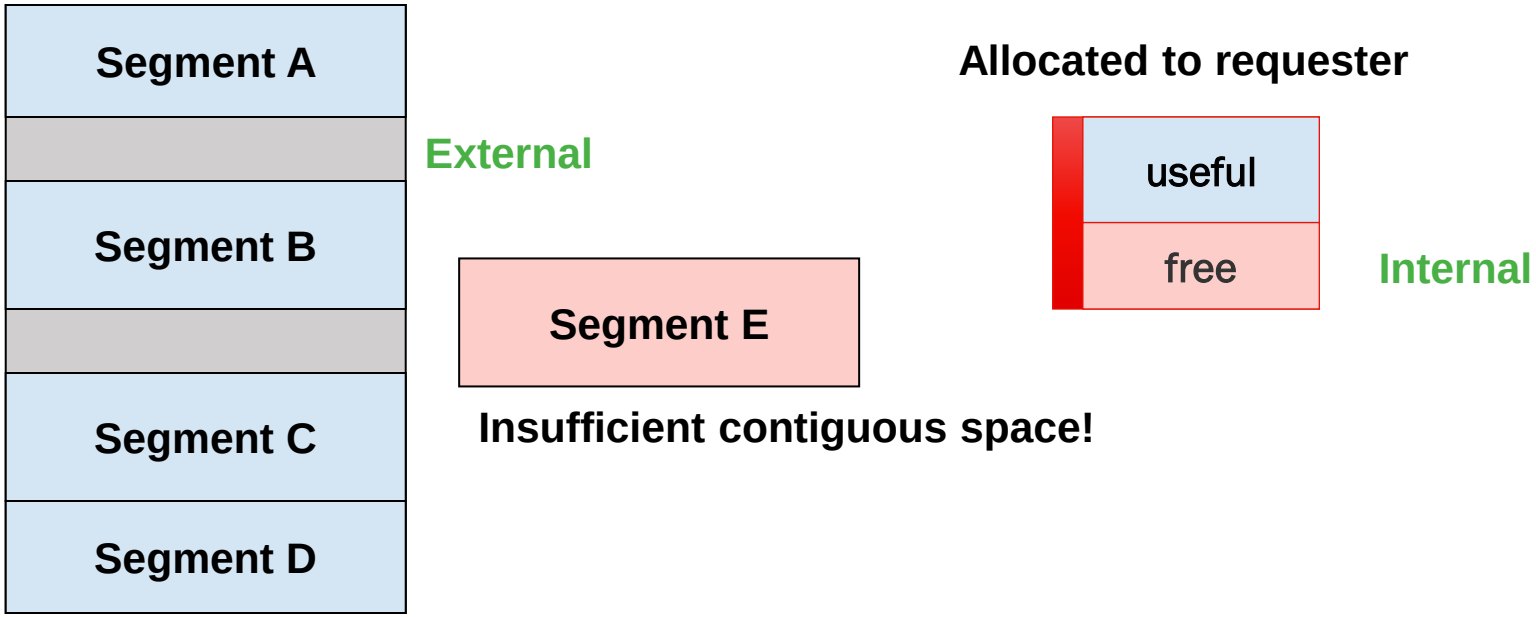
Disadvantages of Segmentation

- Each segment must be allocated contiguously
  - May not have contiguous physical memory for large segments
  - External fragmentation: free memory for object (segment) of size N, but no single size N free region
  - Fix next with paging…

- Lots of segments means lots of registers
  - Would have to resort to an in-memory table

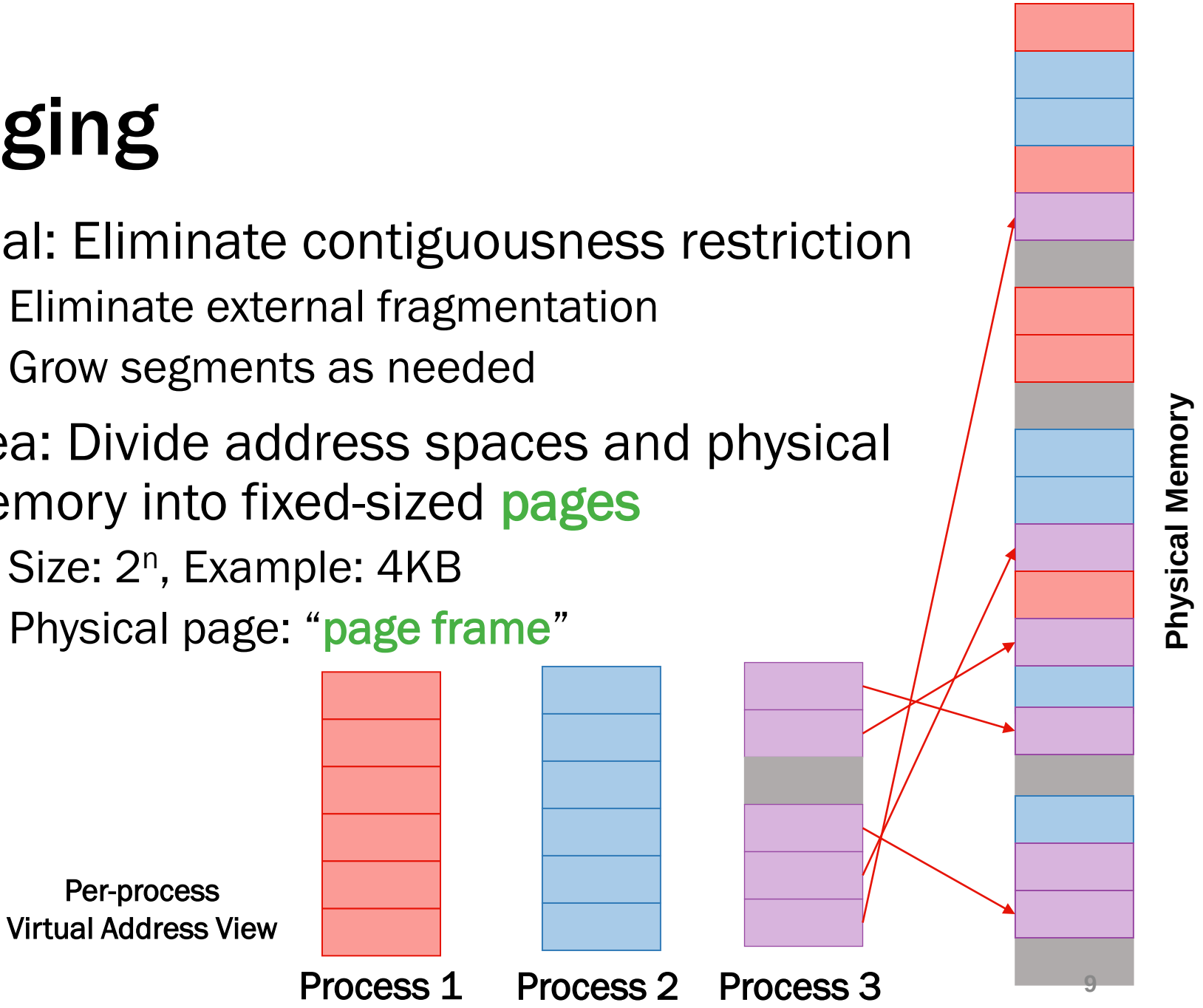
Problem: Fragmentation

- Definition: Free memory that can't be usefully allocated
- Why?
  - Free memory (hole) is too small and scattered (external)
  - Unit of allocation does not match unit of need (internal)
- Types of fragmentation
  - External: Visible to allocator (e.g., OS)
  - Internal: Visible to requester (e.g., must allocate at some granularity)



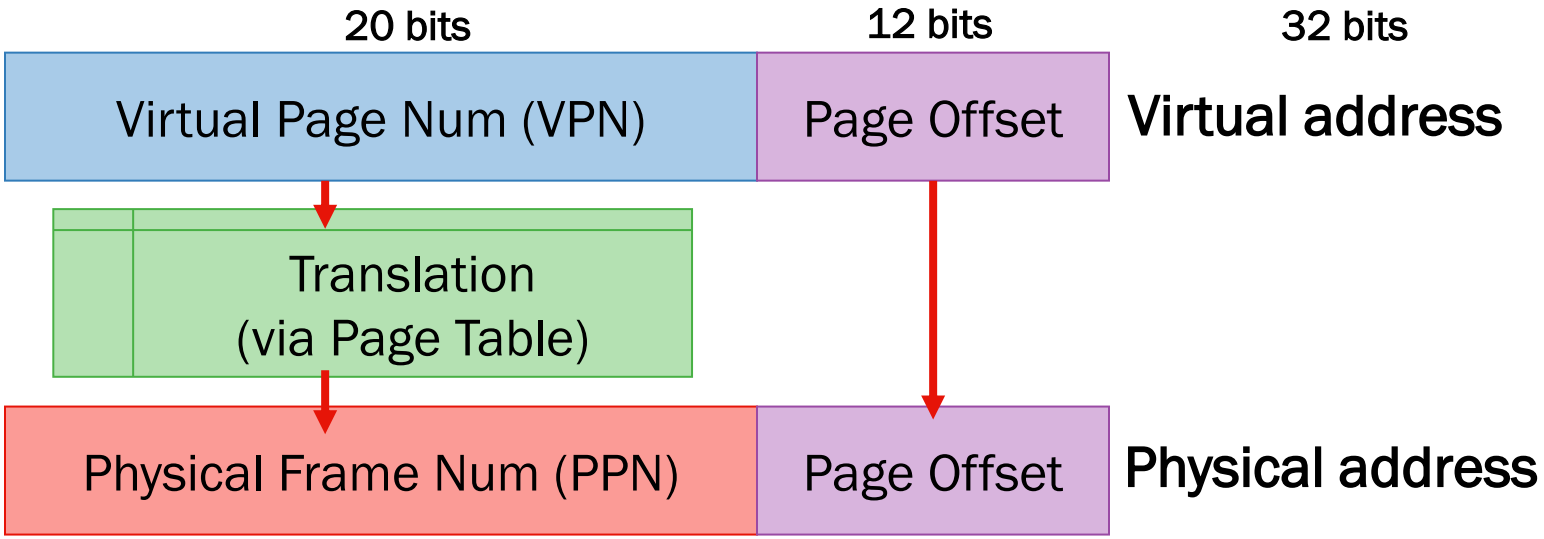
Paging

- Goal: Eliminate contiguousness restriction
  - Eliminate external fragmentation
  - Grow segments as needed
- Idea: Divide address spaces and physical memory into fixed-sized **pages**
  - Size: 2^n, Example: 4KB
  - Physical page: "**page frame**"



Translation of Page Addresses

- Translate virtual address to physical address?
  - High-order bits of address designate page number
  - Low-order bits of address designate offset within page



No addition needed; just append bits correctly…

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

Quiz: Address Format

Given known page size,
how many bits are needed in address to specify offset in page?

Page Size	Low Bits (offset)
16 bytes	
1 KB	
1 MB	
512 bytes	
4 KB	

Quiz: Address Format

Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (VPN)
16 bytes	4	10	
1 KB	10	20	
1 MB	20	32	
512 bytes	9	16	
4 KB	12	32	

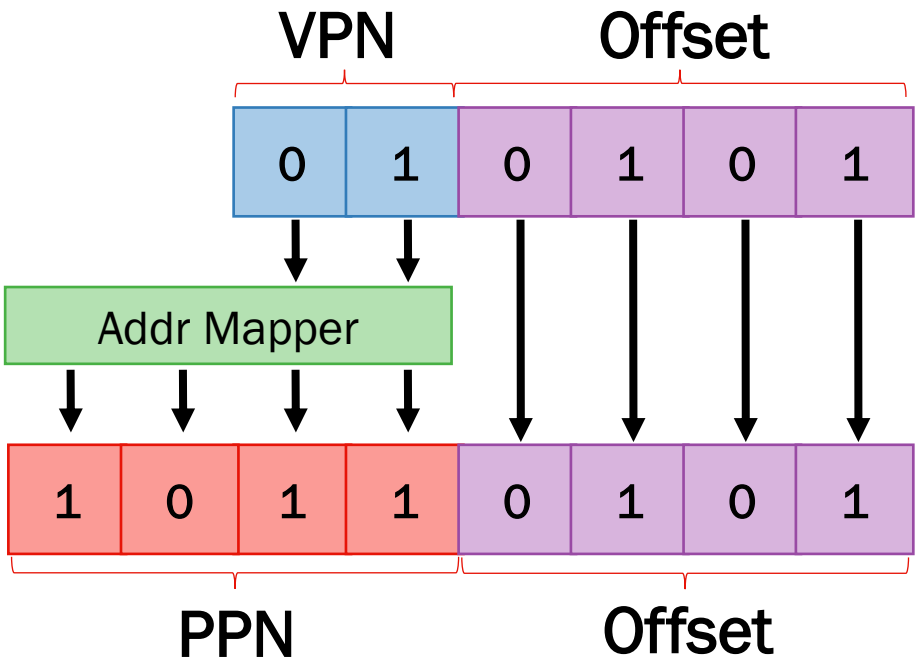
32-bit mode x86

Quiz: Address Format

Given number of bits for VPN,
how many virtual pages can there be in an address space?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (VPN)	Virt Pages
16 bytes	4	10	6	
1 KB	10	20	10	
1 MB	20	32	12	
512 bytes	9	16	7	
4 KB	12	32	20	

Virtual to Phys Page Mapping



Note: Weirdness!
Bits in VA need not equal bits in PA!

What if sizeof(VA) < sizeof(PA)?

What if sizeof(VA) > sizeof(PA)?

Benefits? Drawbacks?

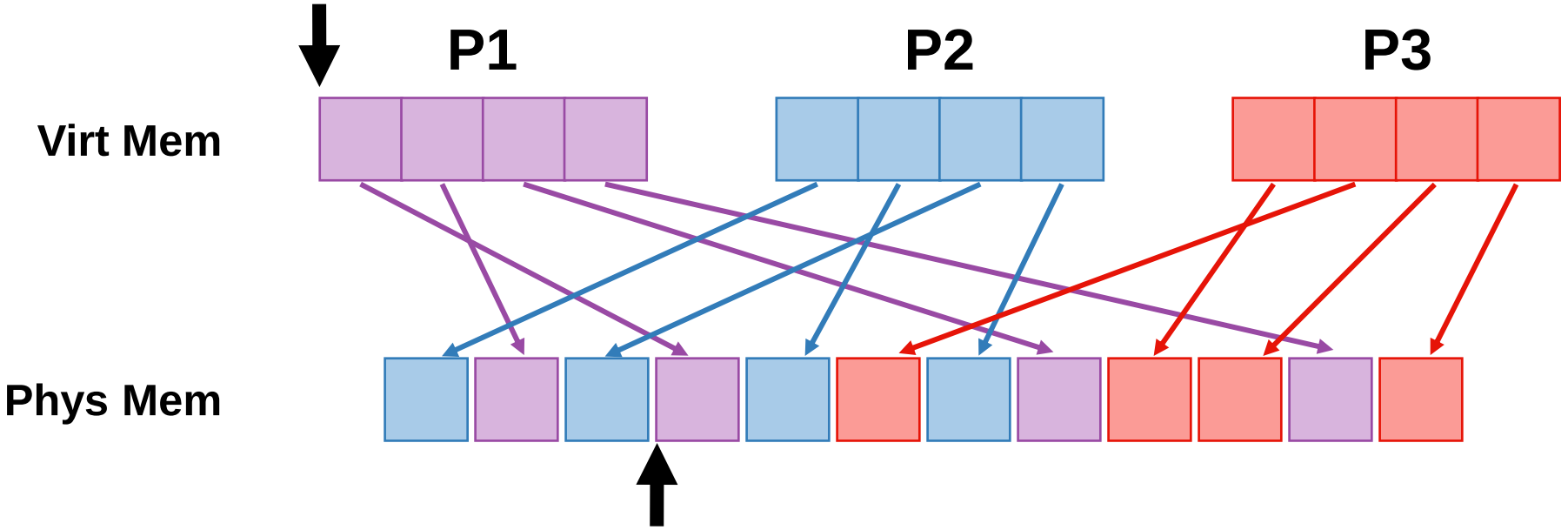
Translate from VPN to PPN?

Segmentation used a formula (e.g., PhysAddr = Base + Offset)

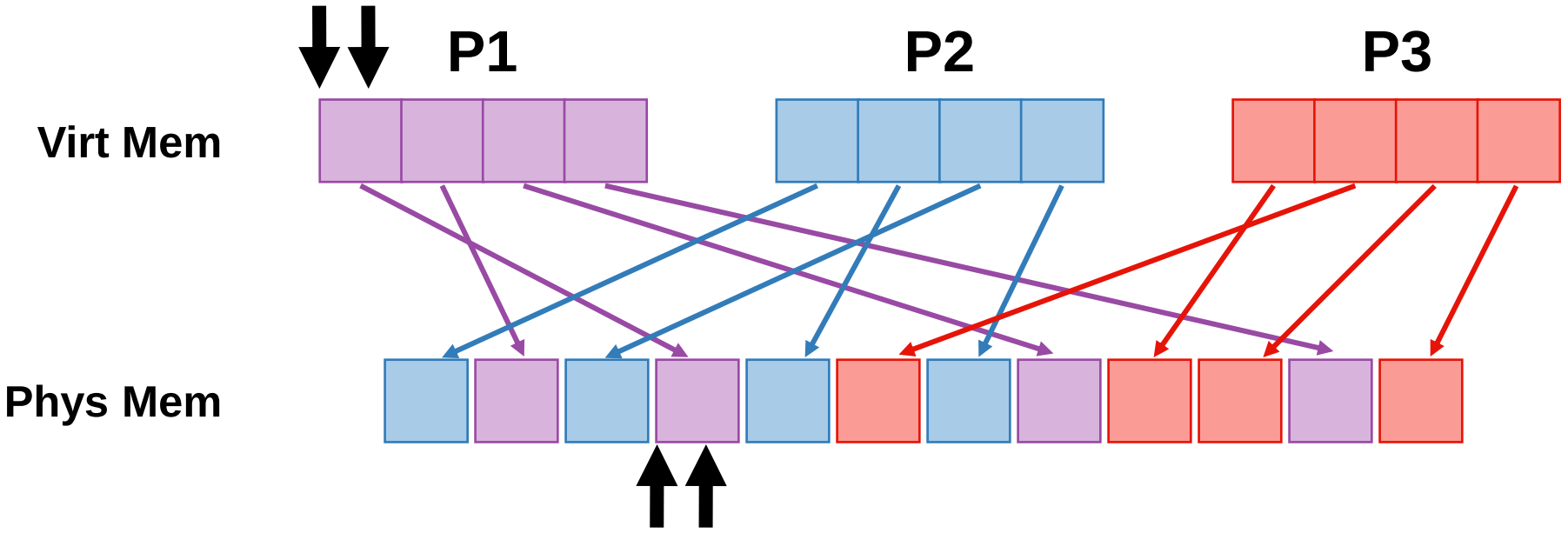
For paging, similar but need **many** more "bases" (no offsets)

What data structure is good? Big array → Page table

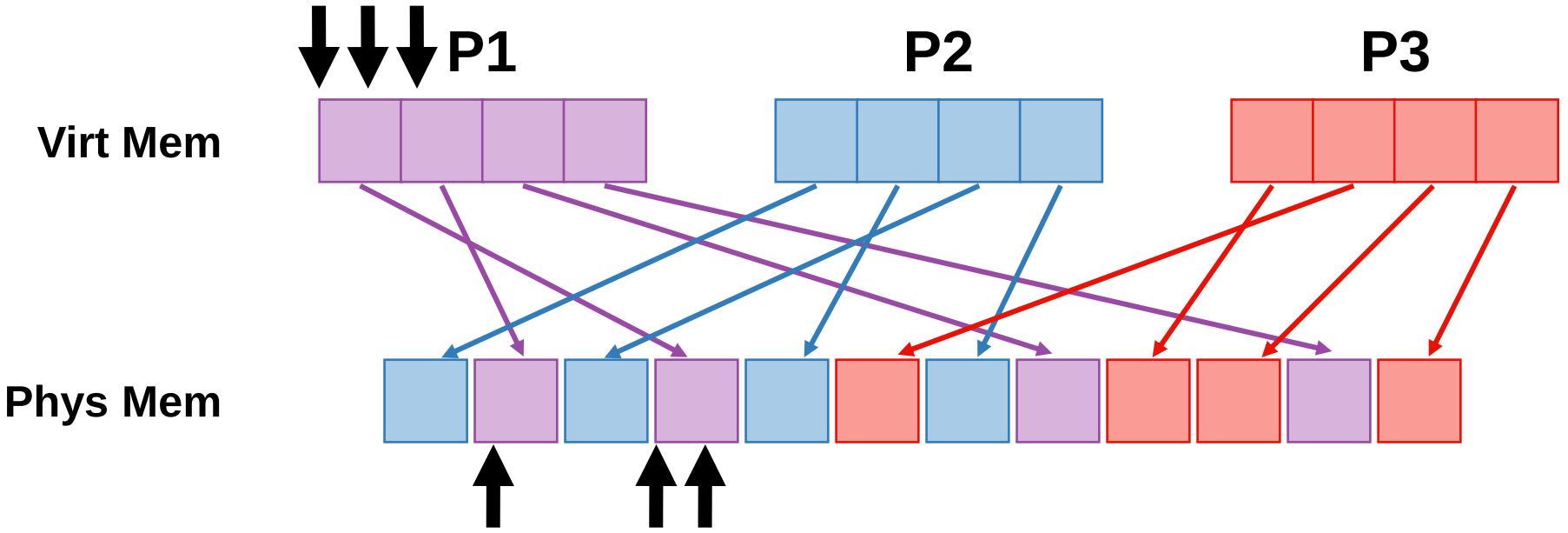
The Mapping



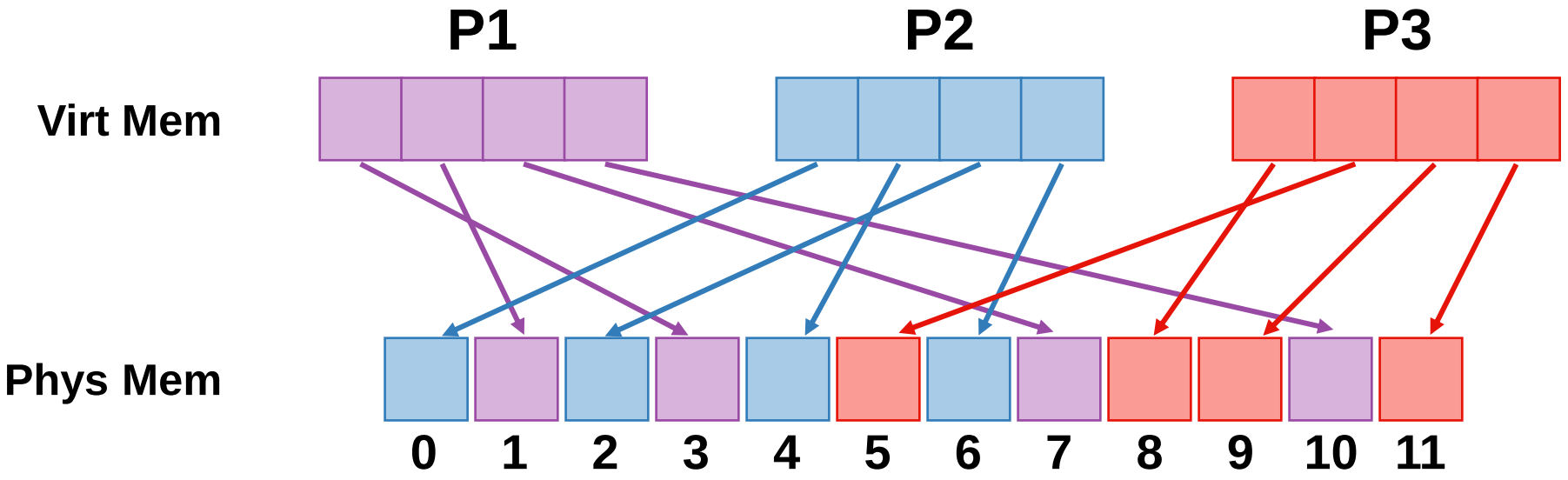
The Mapping



The Mapping



Fill in the Page Table



Page Tables:

P1	P2	P3
3	0	
1	4	
7	2	
10	6	

Where Are Page Tables Stored?

How big is a typical page table? Assuming,

32-bit VAs, 4 KB pages, and 4 byte page table entries (PTEs)

Answer: $2^{(32 - \log(4\ KB))} * 4 =$ **4 MB**

- Page table size = num entries * size of each entry
- Num entries = num virtual pages = $2^{(\text{bits for VPN})}$
- Bits for VPN = 32 – number of bits for page offset
  = 32 – log(4 KB) = 32 – 12 = 20
- Num entries = 2^{20} = ~1 million
- Page table size = Num entries * 4 bytes = 4 MB

Implication: Store each page table in memory

- Hardware finds page table base with register (%cr3 on x86)

What happens on a context-switch?

- Repoint page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

Other PT info

What other info is in pagetable entries besides translation?

- protection bits (read/write/execute)
- present bit (trap if access not present VPN)
- accessed bit (hw sets when page is used)
- dirty bit (hw sets when page is modified)

We'll discuss later when we cover swapping

Page table entries are just bits stored in memory

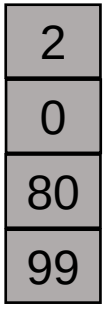
- Agreement between hw and OS about interpretation

Memory Accesses with Pages

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

Assume PT is at PA 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset?

Simplified view of page table



Old: How many mem refs with segmentation?

5 (3 instrs, 2 movl)

Physical Memory Accesses with Paging?

Fetch instruction at VA 0x0010; VPN?

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

Exec, load from VA 0x1100; VPN?

- Access page table to get PPN for VPN 1
- **Mem ref 3: 0x5004**
- Learn VPN 1 is at PPN 0
- **Movl from 0x0100 into reg (Mem ref 4)**

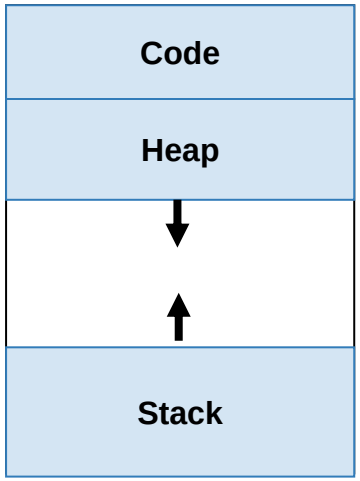
Page Table is slow! Doubles memory references

Advantages of Paging

- No external fragmentation
  - Any page can be placed in any frame in physical memory
- Fast to allocate and free physical memory
  - Alloc: no searching for suitable free space
  - Free: doesn't have to coalesce with adjacent free space
  - Just keep all free page frames on a linked list
- Simple to "swap-out" pages to disk (later lecture)
  - Page size good match for disk I/O granularity
  - Can run process when some pages are on disk
  - Use "present" bit in PTE

Disadvantages of Paging

- Int. fragmentation: page size may not match process need
  - Tension: large pages → small tables but more int. fragmentation
- Additional memory reference to page table, inefficient
  - Page table must be stored in memory
  - MMU stores only base address of page table
  - Solution: Add TLBs (coming up next!)
- Storage for page tables may be substantial
  - Table is large
    (4 GB / 4 KB = 1 mil, 4 B PTE so 4 MB)
    - Even if some entries aren't needed
  - External fragmentation issues again:
    page tables must be contiguous in physical memory
  - Solution: multi-level page tables (page the page tables!)



Translation Steps

H/W: for each mem reference:

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

Which steps are expensive?
How do we avoid them?

Example: Iterating an Array

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

Assume ‘a’ starts at 0x3000

Ignore instruction fetches

VAs Loaded?	PAs Loaded?
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
	load 0x700C
…	

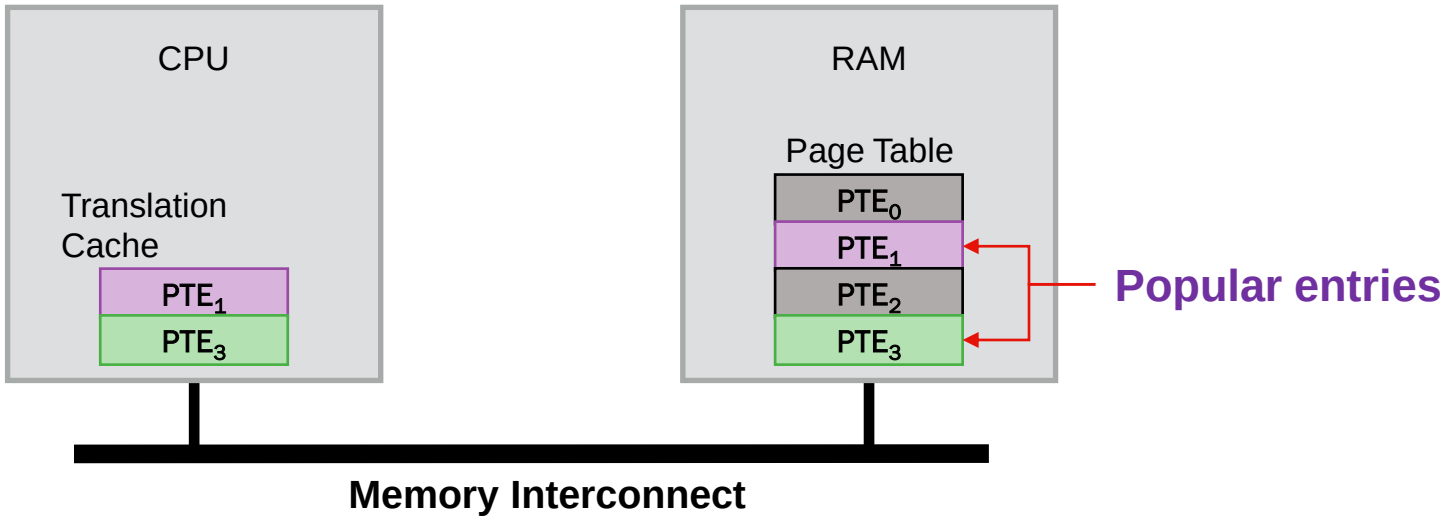
Aside: What can you infer?

- %cr3: 0x1000;
  PTE 4 bytes each
- VPN 3 → PPN 7

Observation:

Spatial locality: repeated access to same PTE because program repeatedly accesses same virtual page

Caching Page Translations



TLB: Translation Lookaside Buffer

Interposes on every memory access

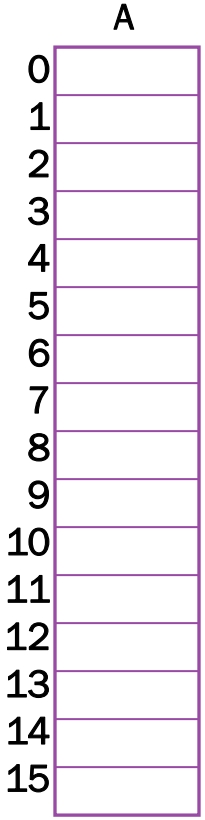
Caches PTEs, each describe VPN to PPN mapping

On access, if PTE in TLB skip page table, else look at page table

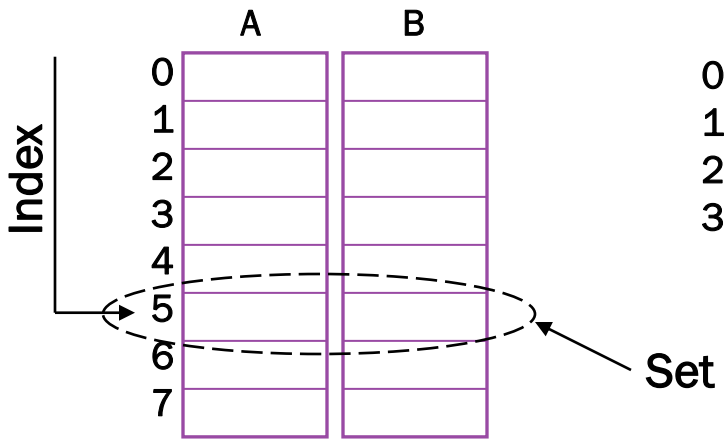
If page table entry present, then cache in TLB

TLB Organization

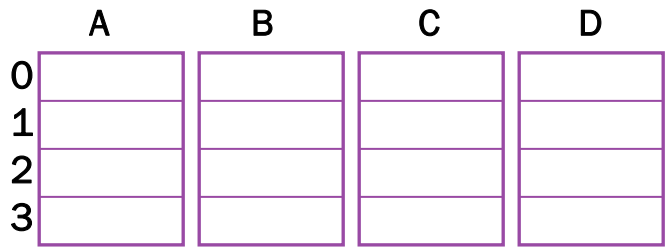
Various ways to organize a 16-entry TLB (artificially small)



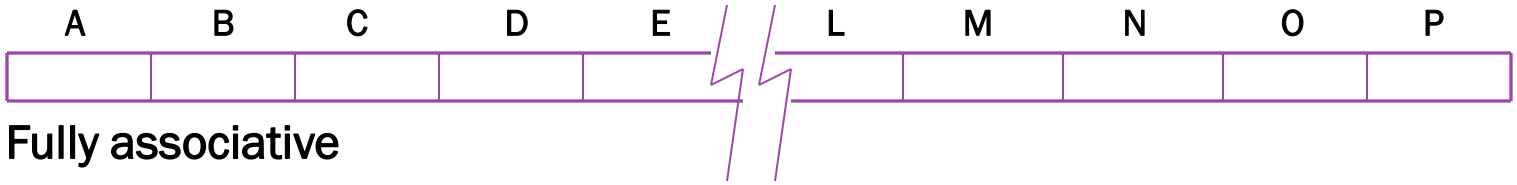
Direct mapped



Two-way set associative



Four-way set associative



Fully associative

Lookup

- Calculate set (tag % num_sets)
- Search for tag within resulting set

TLB Associativity Trade-offs

Higher associativity
- + Better utilization, fewer collisions
- – Slower
- – More hardware
- – Parallel search for all tags doesn't scale, so size of TLB is limited by propagation delay
  - – Generally need to know PA before CPU can use data in caches (3-5 ns)
  - – L2 and lower caches often physically indexed, physically tagged

Lower associativity
- + Fast
- + Simple, less hardware
- – Greater chance of collisions, lower TLB hit rate

TLBs used to be fully associative, but now multi-level TLBs:
32-entry 4-way L1 TLB, 1536-entry 12-way L2 TLB

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

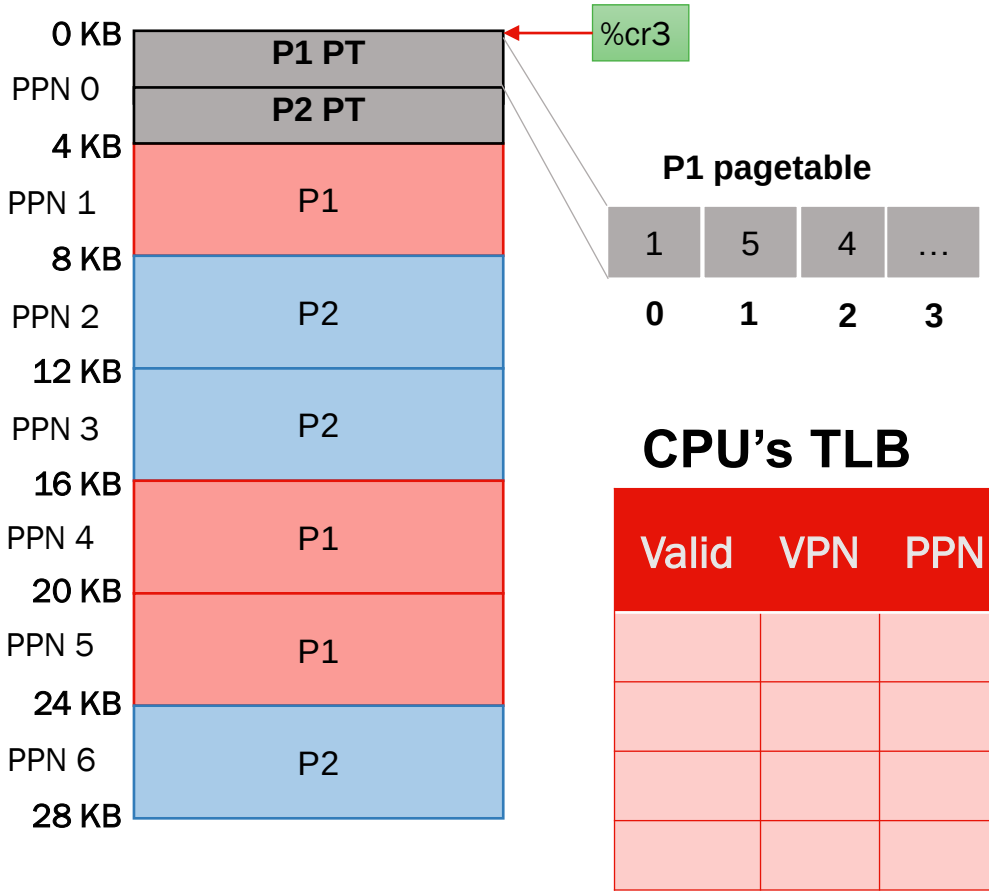
load 0x1008

load 0x100C

…

What will TLB behavior look like?

TLB Accesses: Sequential



VAs	PAs
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

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

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)
= 2048 / (4096 / sizeof(int))
= 4096 / 1024
= 2

Miss rate?

2/2048 = 0.1%

Hit rate? (1 – miss rate)

99.9%

Hit rate better or worse with smaller pages?

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

- TLB “reach”:
  - Number of TLB entries * Page Size

TLB Performance

- Sequential array accesses almost always hit in TLB
  - Very fast!

- What access pattern will be slow?
  - Highly random, with no repeat accesses

Workload Access Patterns

Workload A

```
int sum = 0;

for (i=0; i<2048; 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];
}
```

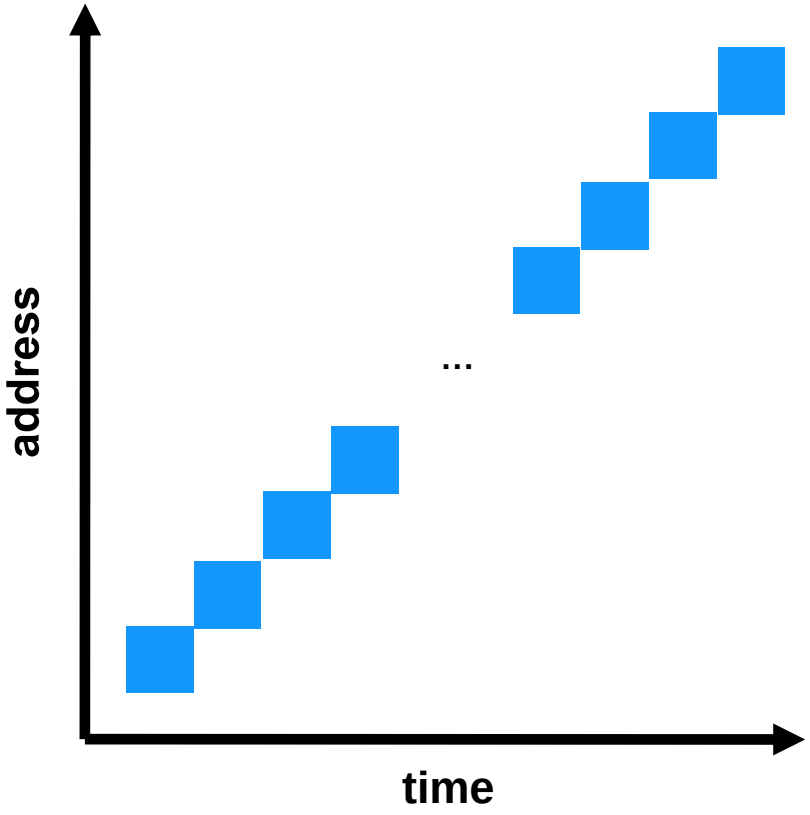
Does the TLB help
- Workload A?
- Workload B?

What does it depend on?

Workload Access Patterns

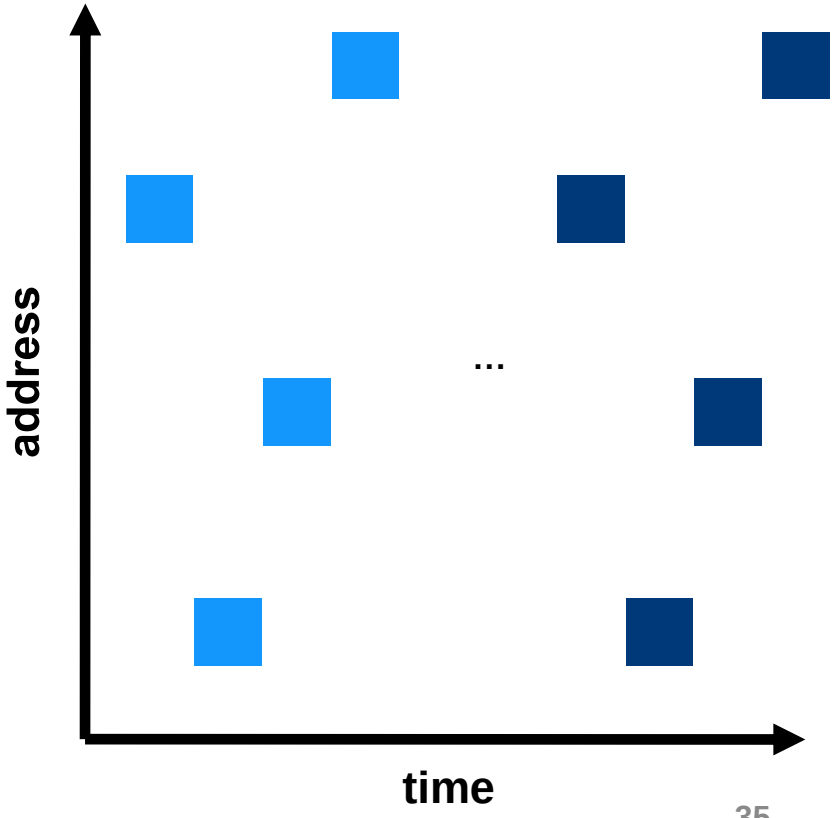
Workload A
Spatial Locality

Sequential Accesses



Workload B
Temporal Locality

Repeated Random Accesses



Workload Locality

Spatial Locality: future access will be to nearby addresses

Temporal Locality: future access repeats to the same data

What TLB characteristics are best for each type?

Spatial:

- Access same page repeatedly; need same VPN to PPN translation
- Same TLB entry reused

Temporal:

- Access same address near in future
- Same TLB entry reused in near future
- How near in future? How many TLB entries are there?

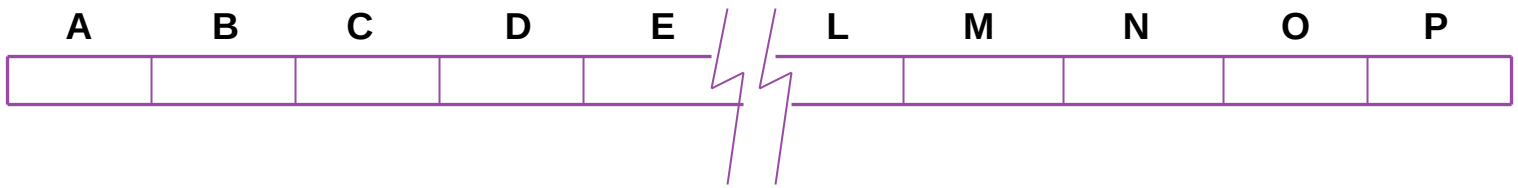
TLB Replacement Policies

LRU: evict Least-Recently Used TLB slot when needed

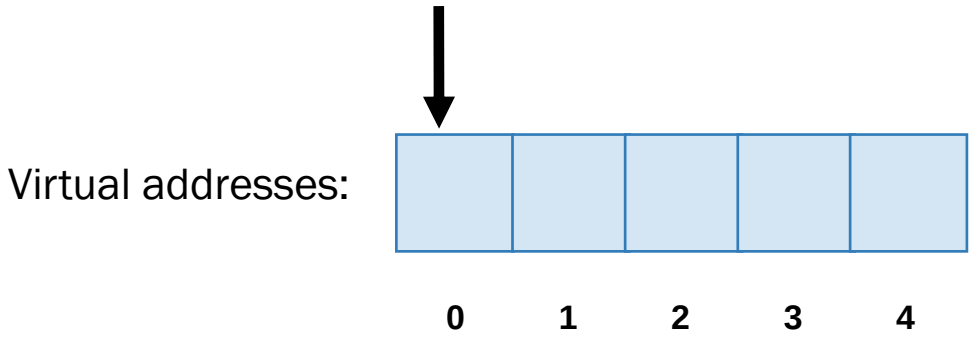
(More on LRU later in policies soon)

Random: Evict randomly choosen entry

Which is better?



LRU Troubles



Valid	Virt	Phys
0	?	?
0	?	?
0	?	?
0	?	?

Workload repeatedly accesses same offset across 5 pages (strided access), but only 4 TLB entries

What will TLB contents be over time?
How will TLB perform?

Sometimes random is better than “smarter” policy

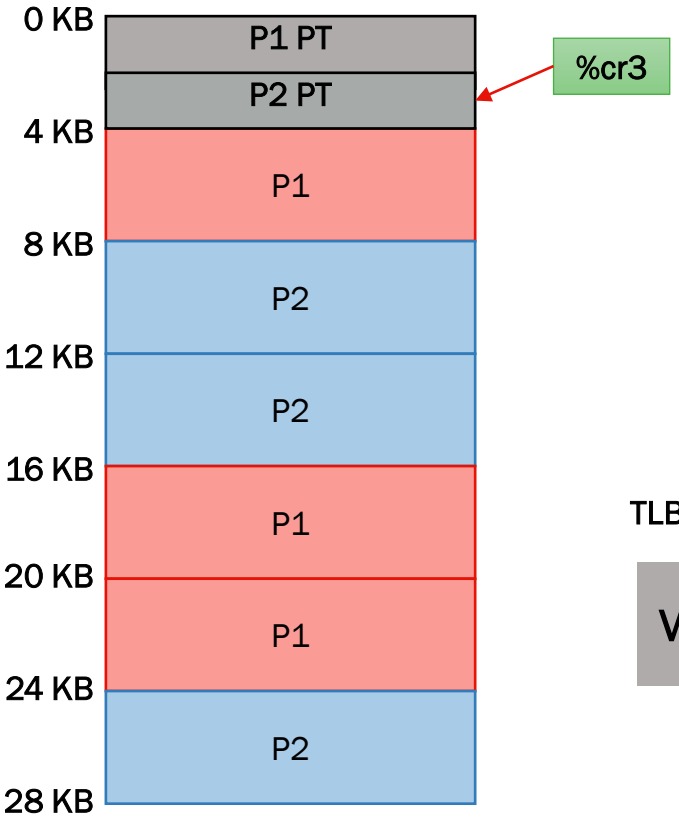
Context Switches

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

Solutions?

1. Flush TLB on each switch
   - Costly; lose all recently cached translations
2. Track which entries are for which process
   - Address Space Identifier (called PCIDs on Intel)
   - Tag each TLB entry with an 8-bit ASID
     - How many ASIDs do we get? (Intel has 4096)
     - Why not use PIDs?

TLB Example with ASID

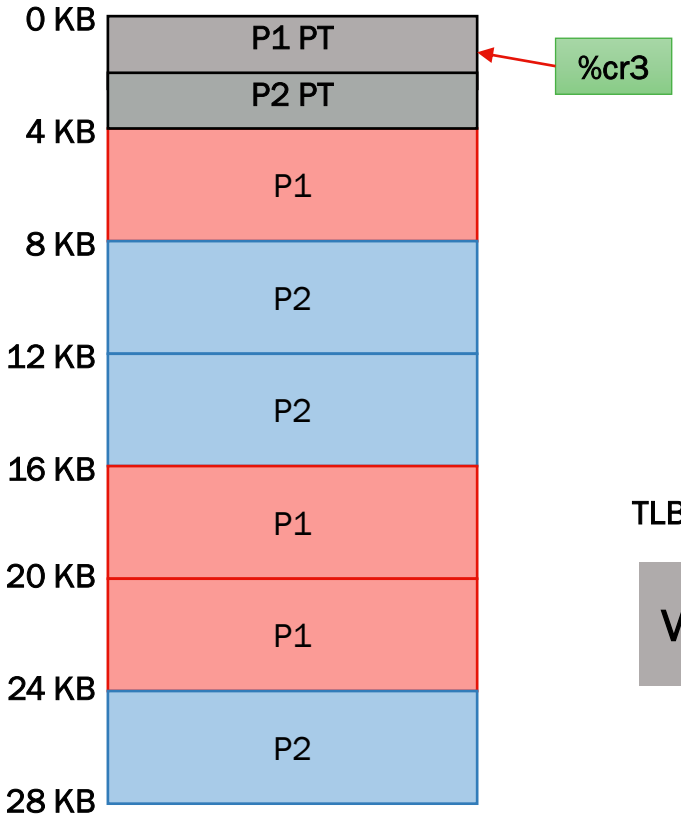


Virtual	Physical
load 0x1444 ASID: 12	load 0x2444

TLB:

Valid	Virt	Phys	ASID
1	1	2	12

TLB Example with ASID



1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual		Physical
load 0x1444	ASID: 12	load 0x2444
load 0x1444	ASID: 11	load 0x5444

TLB:

Valid	Virt	Phys	ASID
1	1	2	12
1	1	5	11

No need to flush TLB on context switch; TLB hardware ensures cached entries from different processes don't interfere

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 can have multiple TLBs

- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 TLB for “super pages”

HW and OS Roles

Who handles TLB miss? **Hardware or OS?** Both have been used

If Hardware: CPU must know where pagetables are

- %cr3 register on x86
- Page table structure fixed and agreed upon between HW and OS
- HW "walks" the page table and fills TLB

If 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?

Need same protection bits in TLB as page table
(read/write/execute and kernel/user mode access)

Summary

- 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