CS5460: Operating Systems

Lecture 9: TLBs & Multi-level Paging

(Chapters 19, 20)

Slide Credit: Andrea Arpaci-Dusseau

Assignments

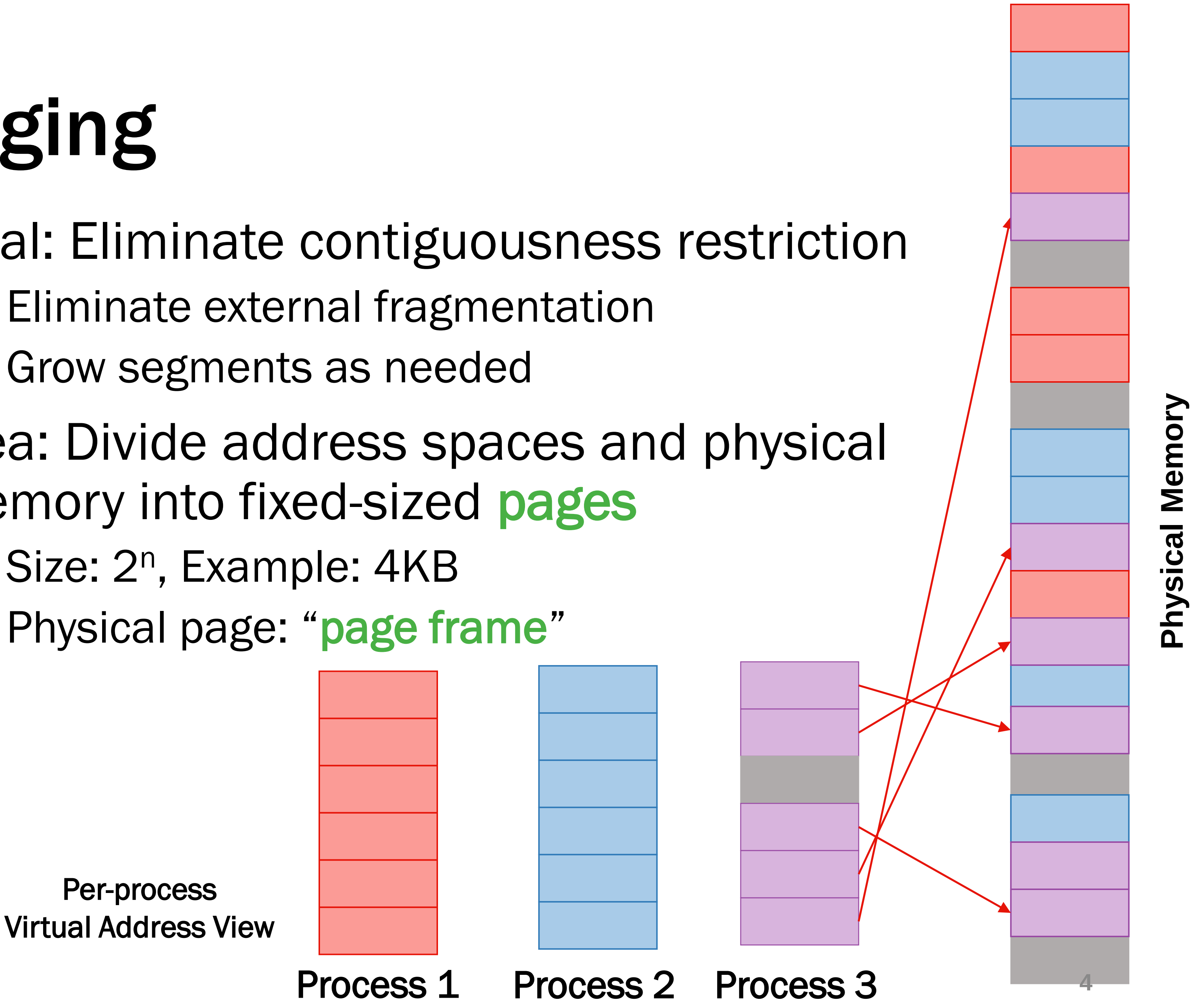
- Assignment 2
    - Due Tonight! 2/16

Virtualizing Memory

- How do we run multiple processes when addresses are “hardcoded” into process binaries?
- Possible solutions
  - Time Sharing
  - Static Relocation
  - Base & Bound
  - Segmentation
  - Paging

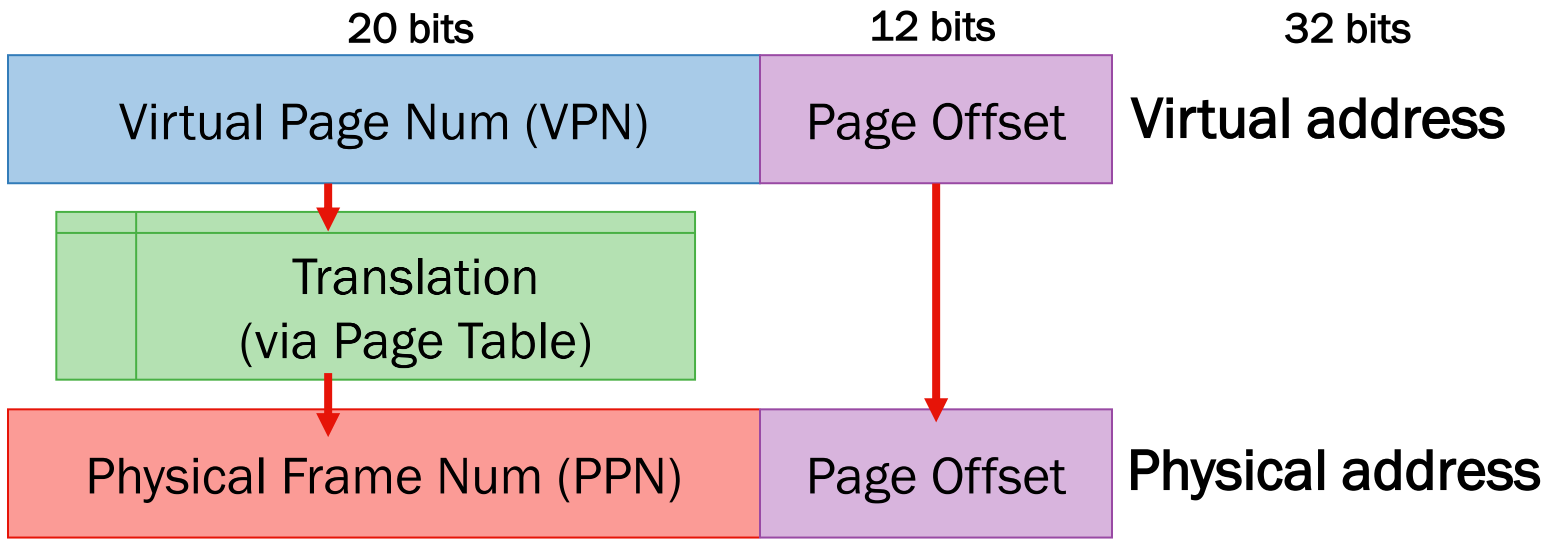
Paging

- 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

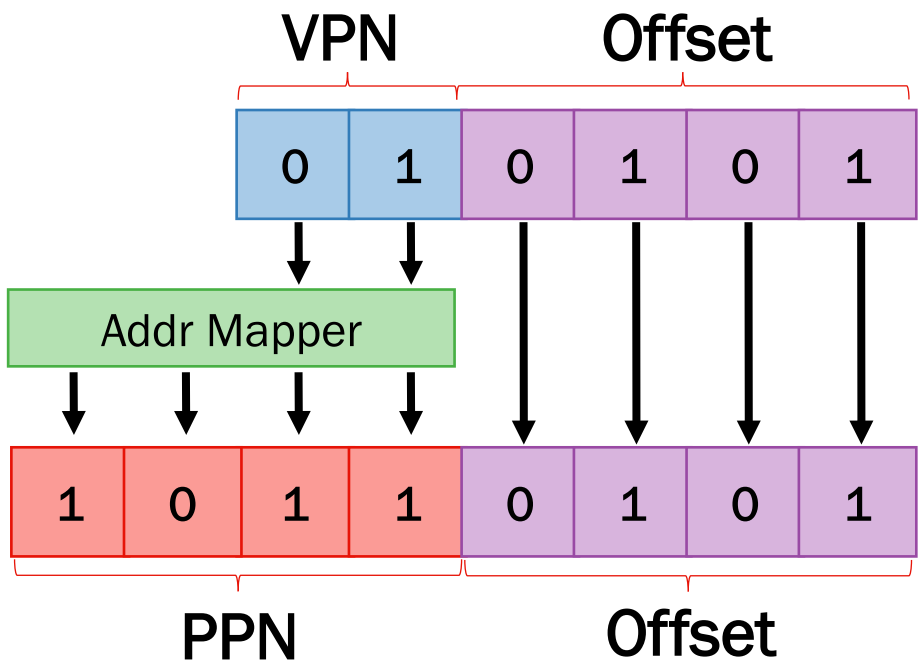
- Translate virtual address to physical address?
  - High-order bits of address designate page number
  - Low-order bits of address designate offset within page



No addition needed; just append bits correctly…

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

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?

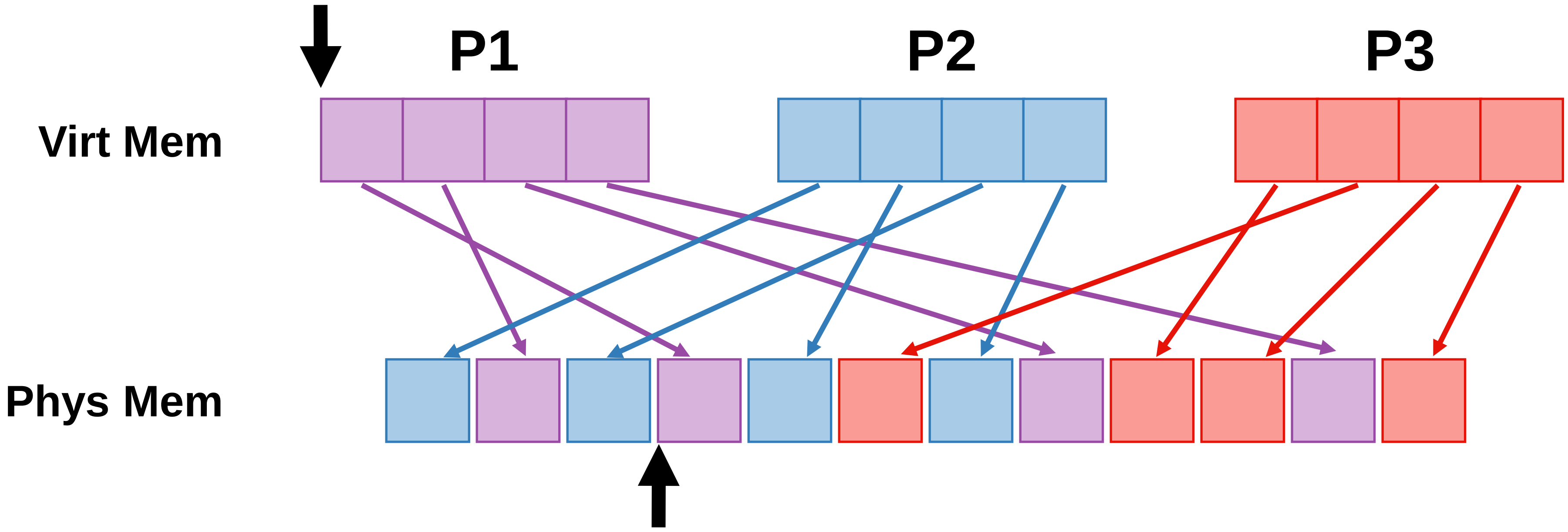
Translate from VPN to PPN?

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

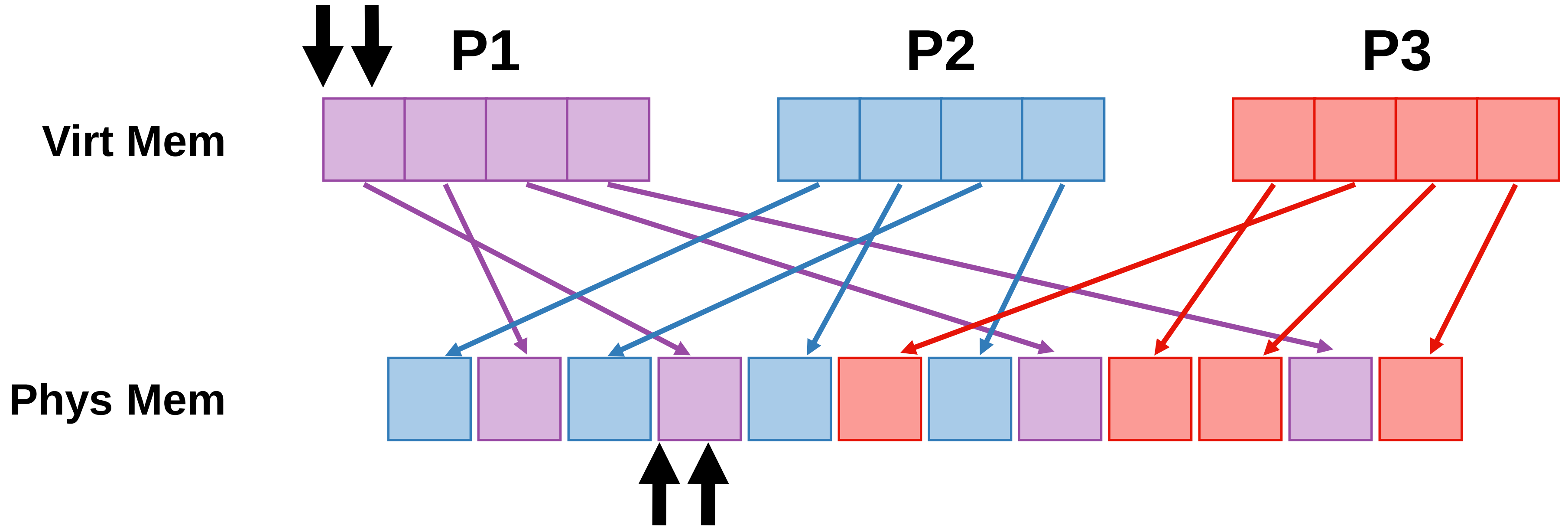
For paging, similar but need **many** more “bases” (no offsets)

What data structure is good? Big array → Page table

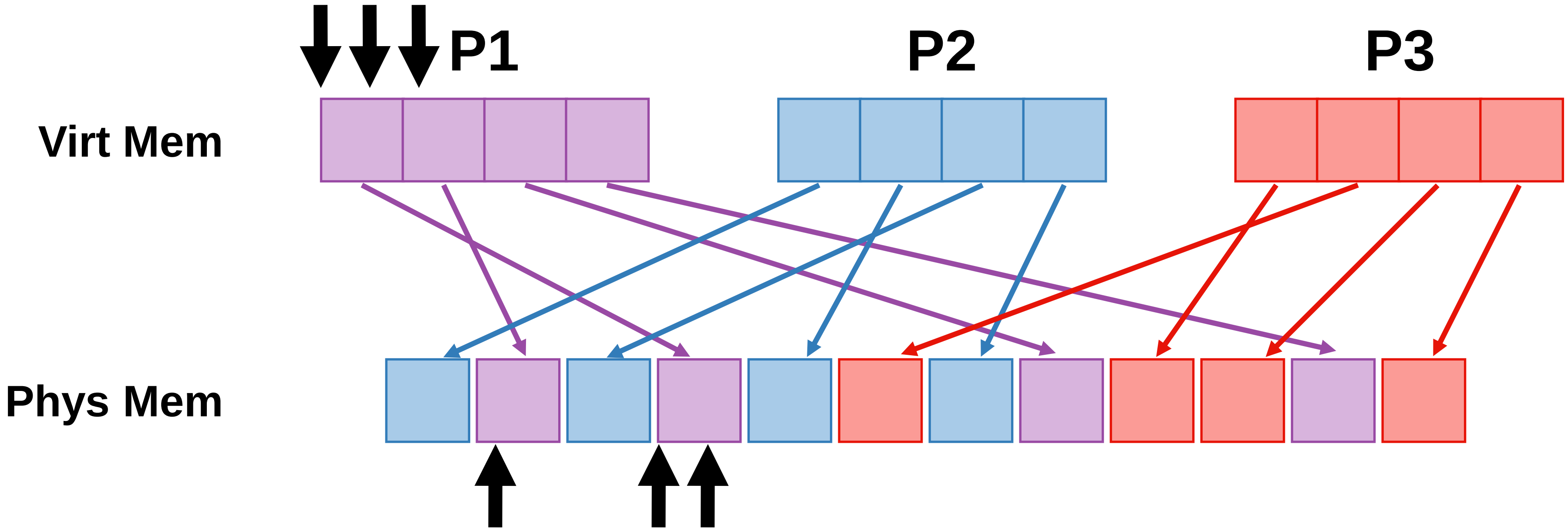
The Mapping



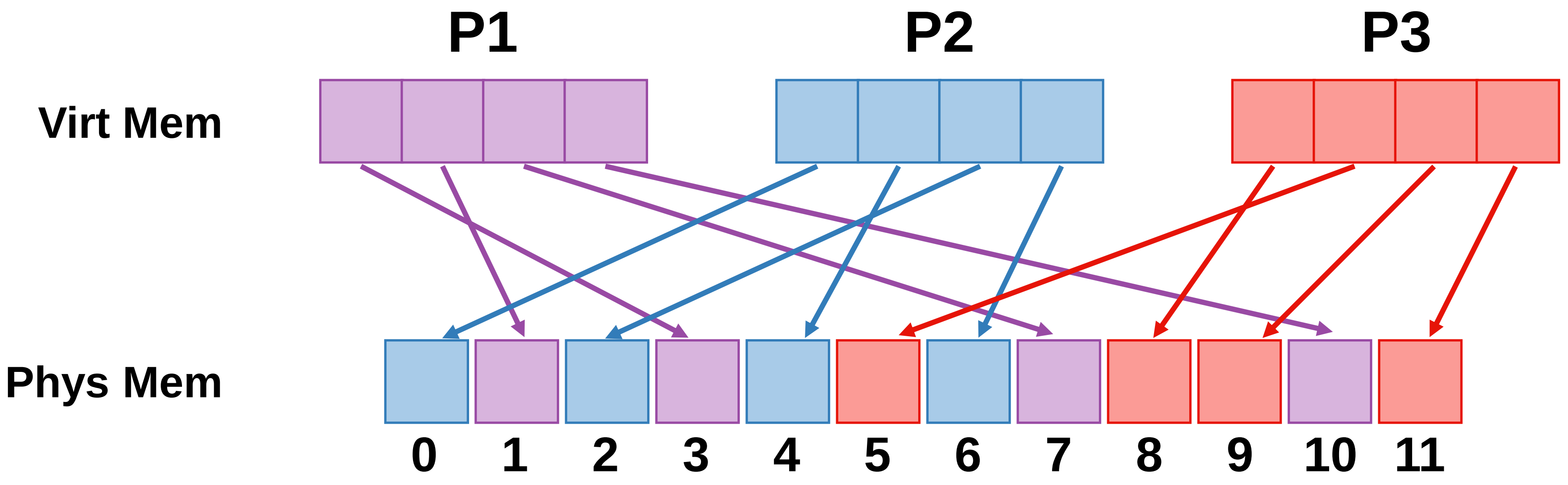
The Mapping



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

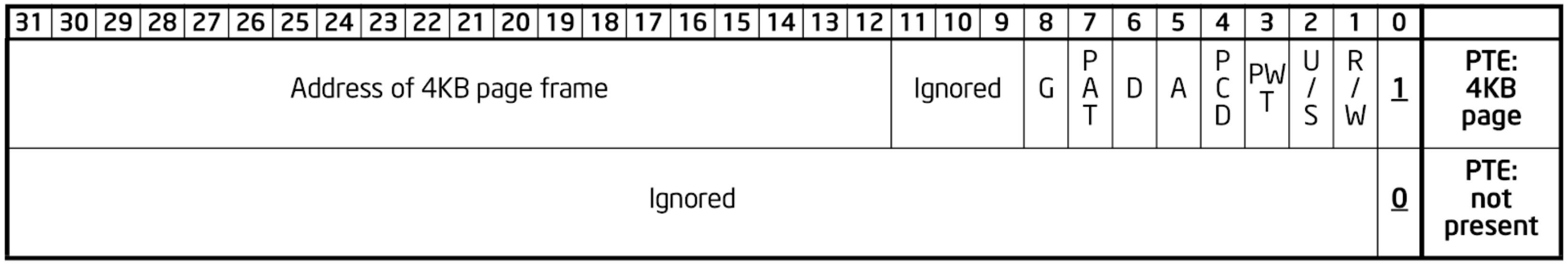
- protection bits (read/write/execute)
- present bit (trap if access not present VPN)
- accessed bit (hw sets when page is used)
- dirty bit (hw sets when page is modified)

We'll discuss later when we cover swapping

Page table entries are just bits stored in memory

- Agreement between hw and OS about interpretation

Intel 32-bit PTE Format



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	**1**	**PTE: 4KB page**
Ignored																															**0**	**PTE: not present**

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

- Important: PFN, Dirty (D), Accessed (A), User/Kernel (U/S), Writable (R/W), Present (Bit 0)
- Other stuff:
  - Global (G): PTE need not be flushed on TLB flush;
    - e.g. kernel PTEs which are the same for all processes
  - Caching Control (PAT, PCD, PWT) : Disable & Write-through
    - e.g. disable caching for addresses mapped to devices

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

2
0
80
99

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)**

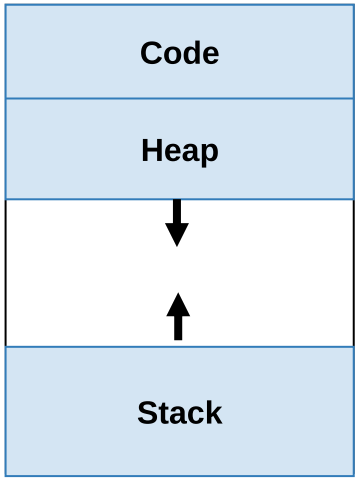
Page Table is slow! Doubles memory references

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

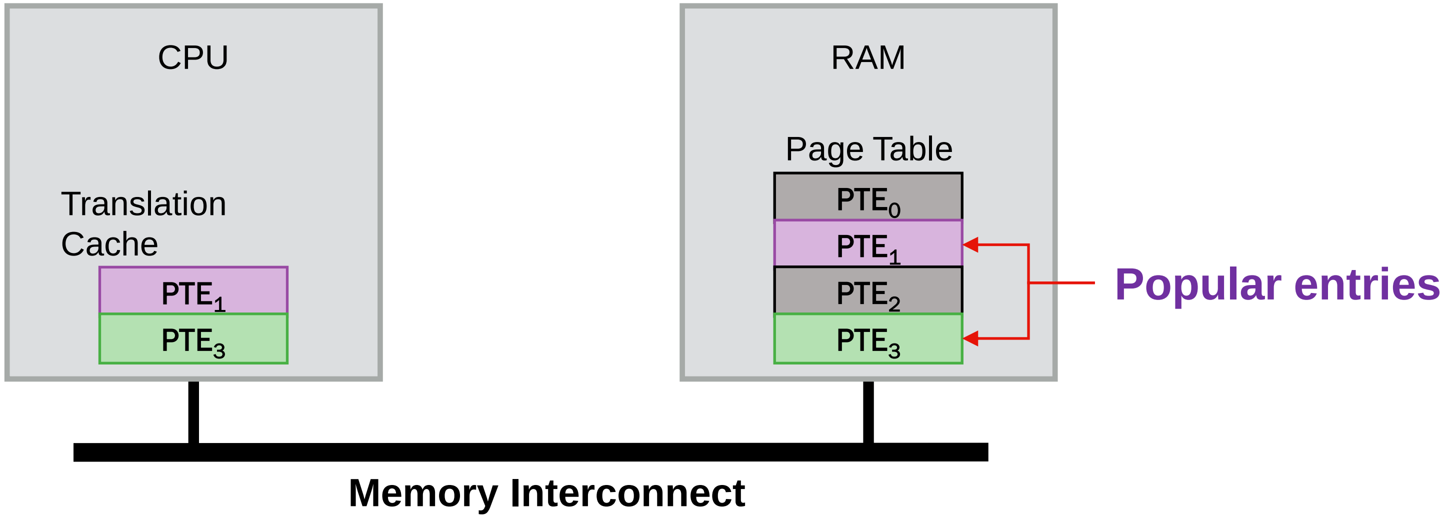
Ignore instruction fetches

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

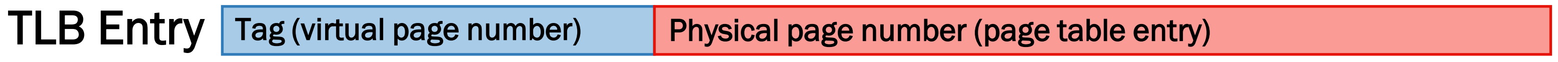
Interposes on every memory access

Caches PTEs, each describe VPN to PPN mapping

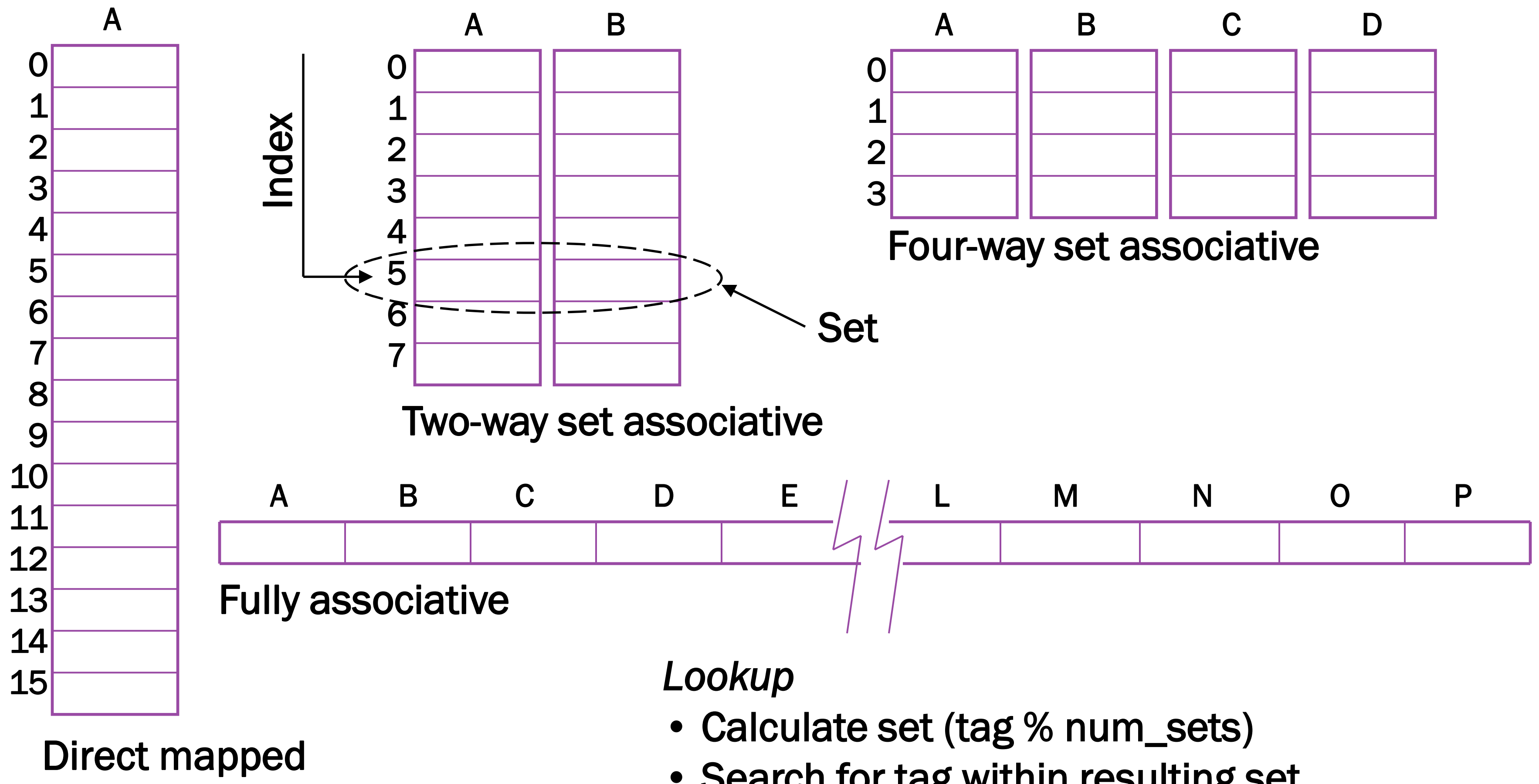
On access, if PTE in TLB skip page table, else look at page table

If page table entry present, then cache in TLB

TLB Organization


TLB Entry
Tag (virtual page number)
Physical page number (page table entry)

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



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

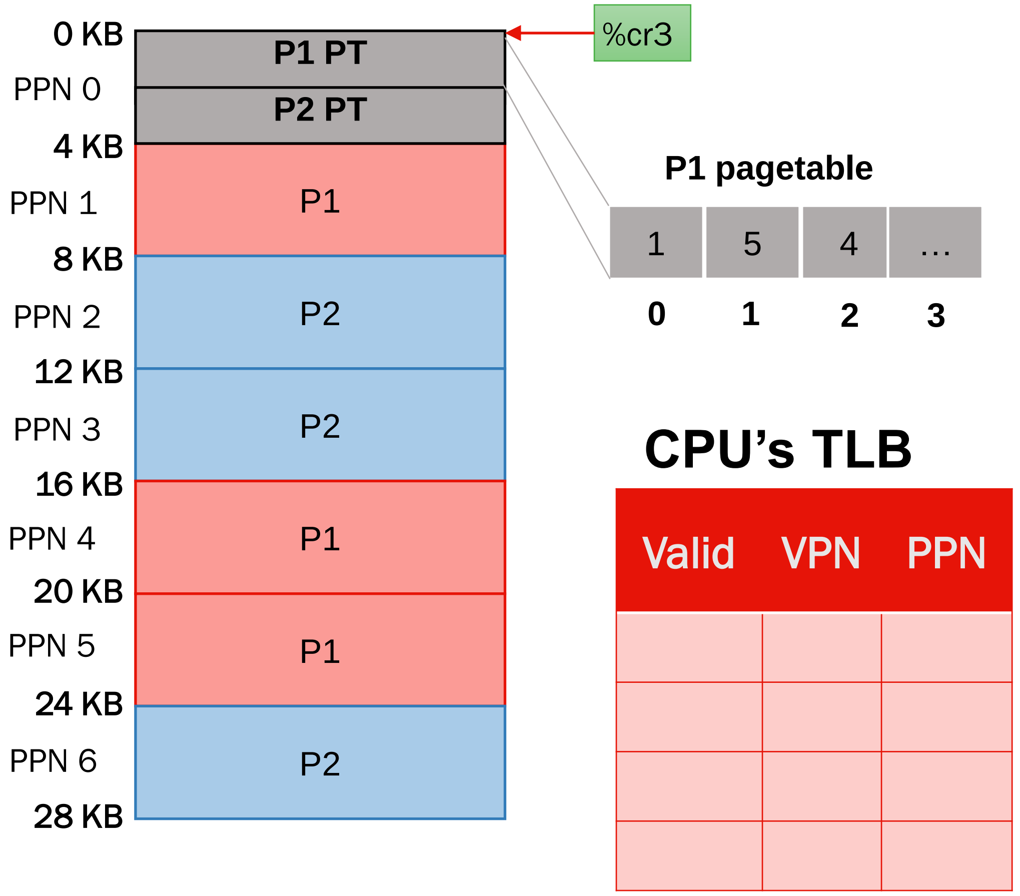
load 0x1008

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];
}
```

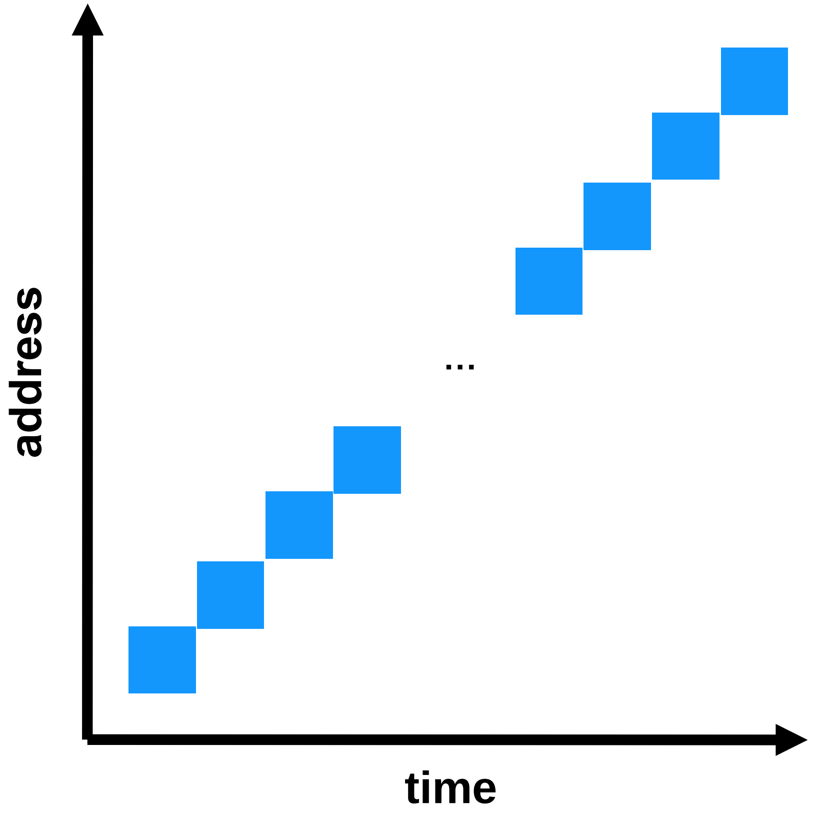
Workload B

```
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];
}
```

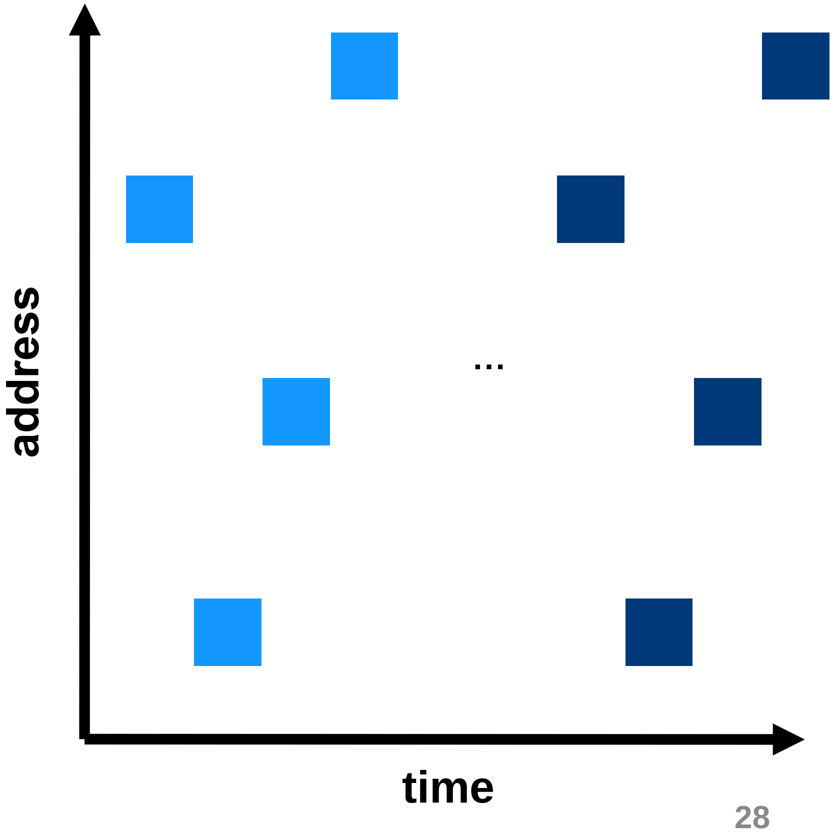
Does the TLB help
  Workload A?
  Workload B?
What does it depend on?

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?

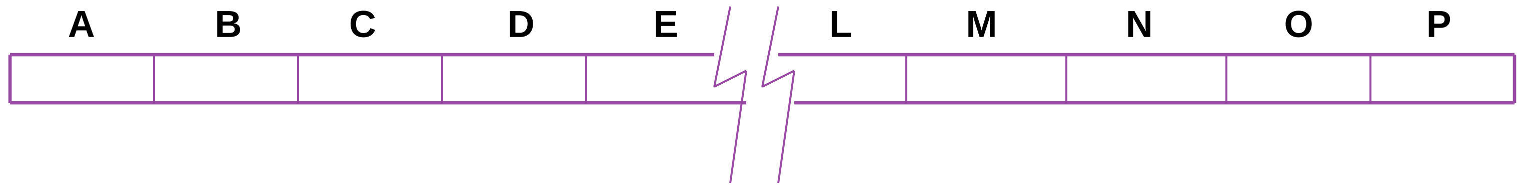
TLB Replacement Policies

LRU: evict Least-Recently Used TLB slot when needed

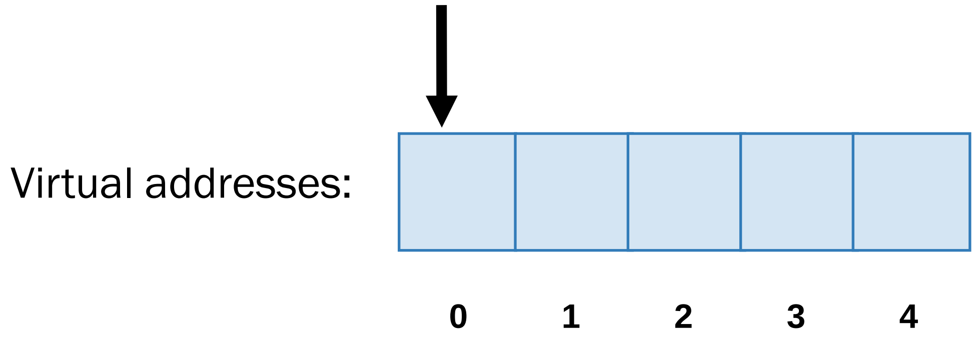
(More on LRU later in policies soon)

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

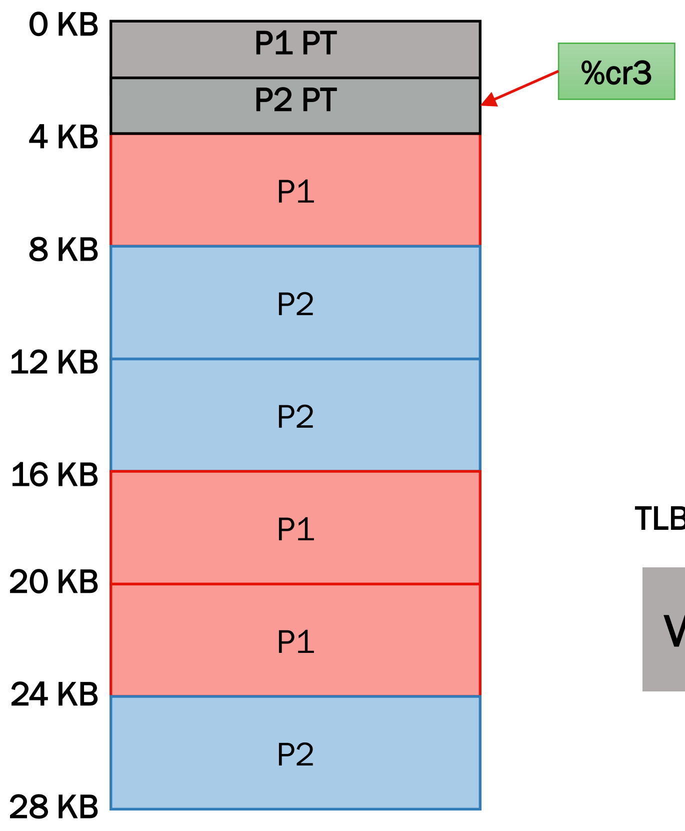
Context Switches

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



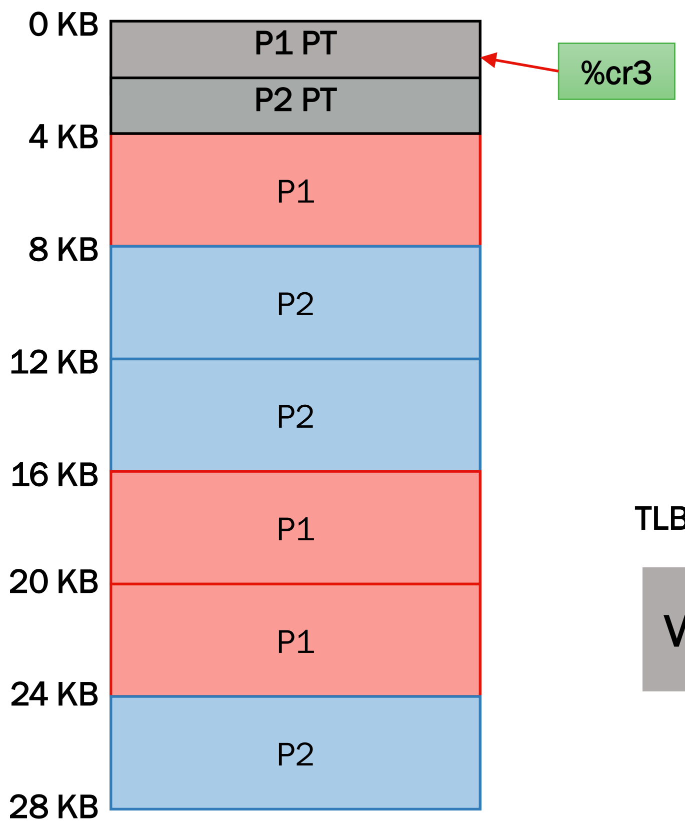
1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

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

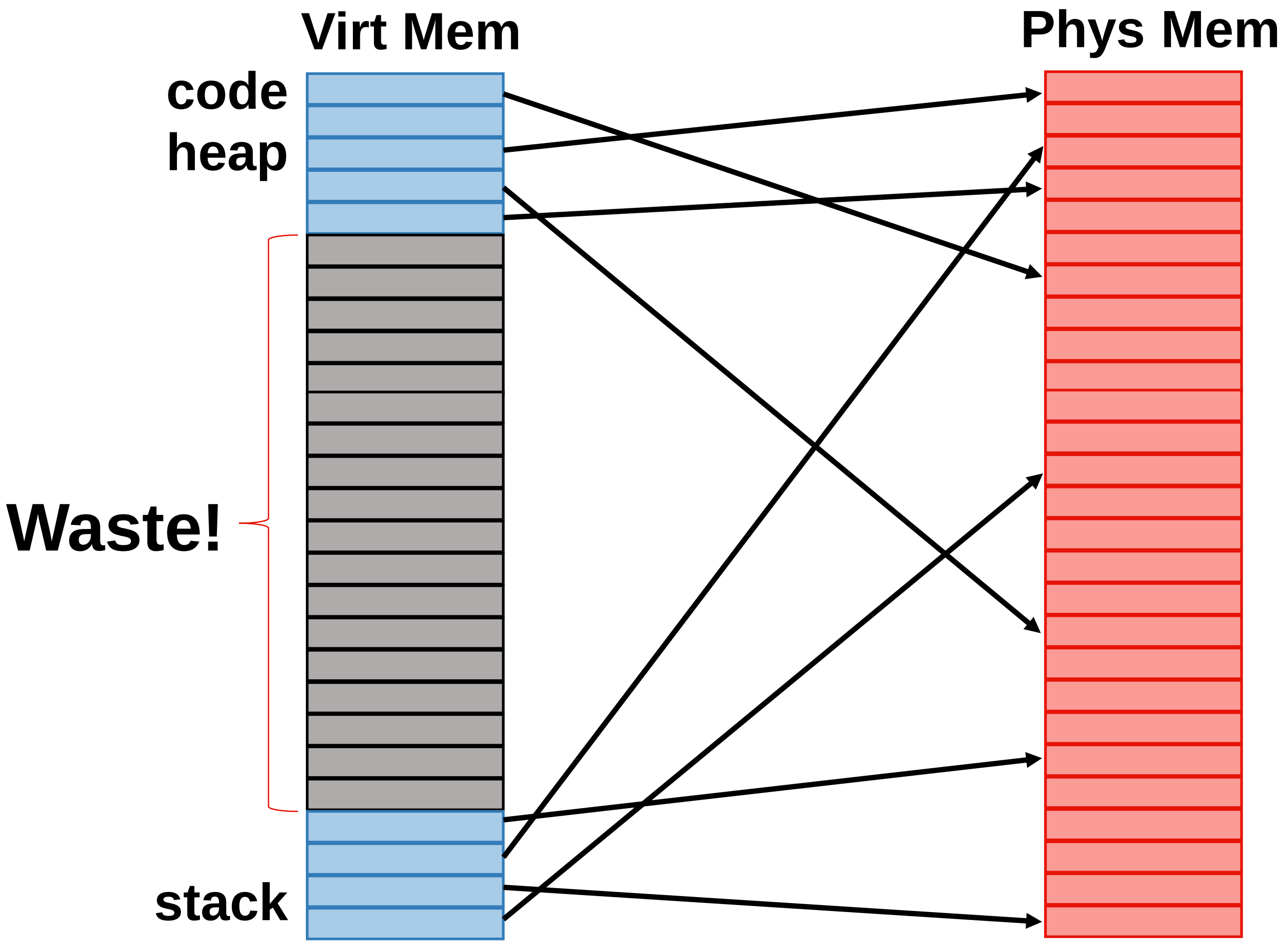
- “Software-managed TLB”
- OS interprets page tables as it chooses
- Modifying TLB entries is privileged
  - otherwise what could process do?

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

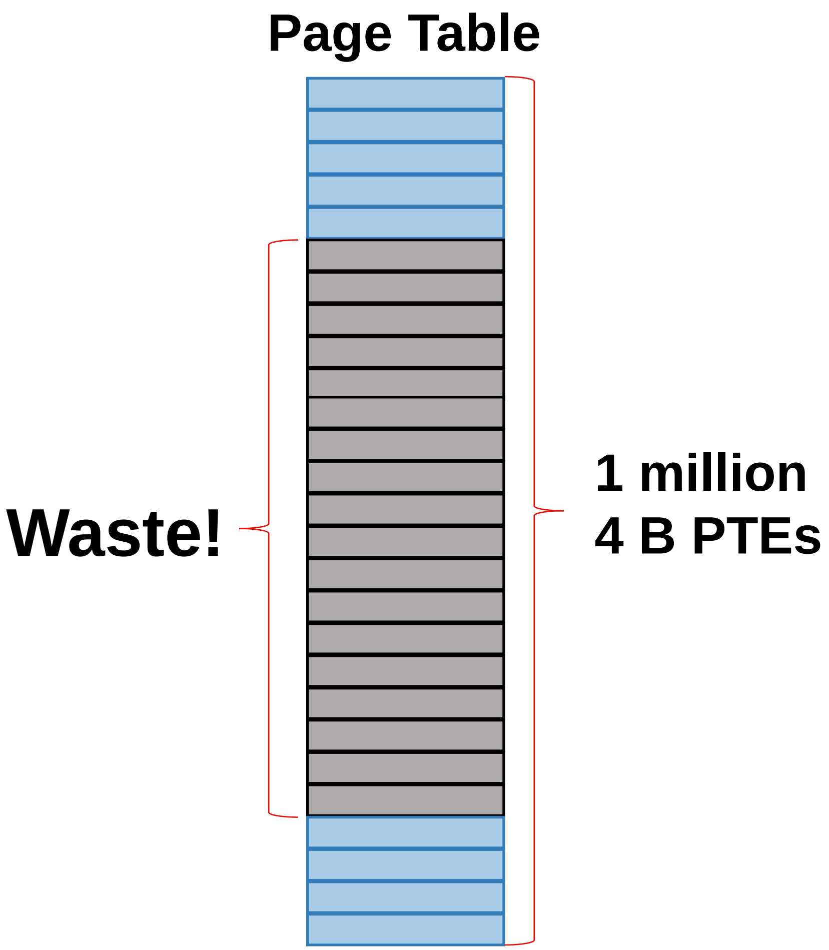
TLBs Summary

- 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

Big Tables due to Hole!



Most PTEs are Invalid (no PPN)



Avoid Simple Linear Page Table

- Use more complex page tables instead of just big array
- Any data structure is possible with software-managed TLB
    - Hardware looks for VPN in TLB on every memory access
    - If TLB does not contain VPN, TLB miss
        - Trap into OS and let OS find VPN to PPN translation
        - OS notifies TLB of VPN to PPN for future accesses

Other Approaches

1. Inverted Page Tables
2. Multi-level Page Tables
   - Page the page tables
   - Page the page tables of page tables…

Inverted Page Tables

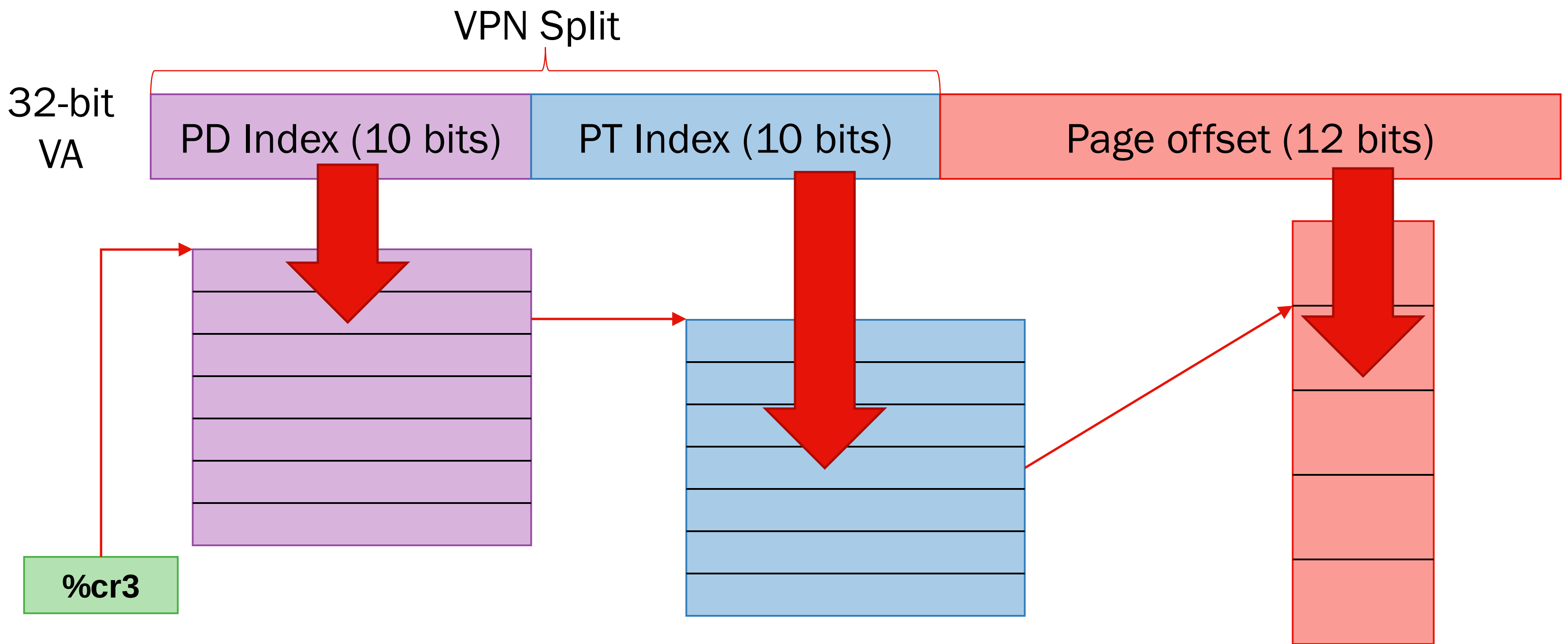
- Hash table indexed by VPN+ASID containing PPN
  - Only one table for whole system rather than per process
  - Only contains entries equal to allocated page frames (size is independent of number of VPNs)
- Complex data structure
  - Only done under software-controlled TLB
- On TLB miss
  - OS handles trap
  - Looks up hash(VPN+ASID) to find PPN
  - Populates TLB entry for VPN, ASID, PPN combination
  - Returns from trap
- For hw-controlled TLB, need well-defined, simple approach

Multi-level Page Tables

Goal: Let page tables be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)



Multi-level Paging Example

Page Directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

Page Table @ PPN 0x3

PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

Page Table @PPN 0x92

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

translate 0x01ABC

translate 0x00000

translate 0xFEED0

20-bit address:

PD Index (4 bits)	PT Index (4 bits)	Page offset (12 bits)

Address Format for MLP

30-bit address:

PD Index	**PT Index**	**Page offset (12 bits)**

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
  - → number PTE per page = 2^{10}
- → # bits for selecting inner page = 10

This is key!

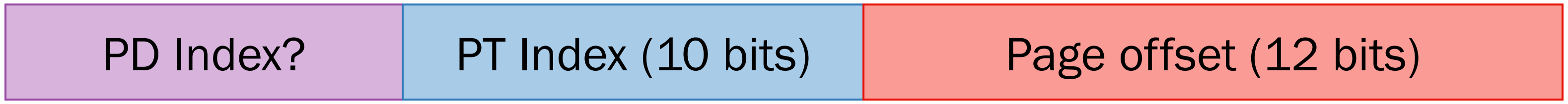
Remaining bits for outer page:

- 30 – 10 – 12 = 8 bits

Problem with 2 Levels?

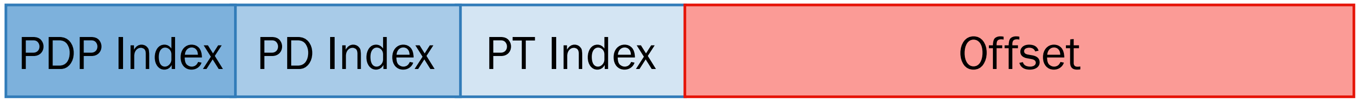
Problem: page directories (outer level) may not fit in a page

64-bit VA:



Solution:

- Split page directories into pieces
- Have a directory of page directories of page tables



How large is virtual address space with 4 KB pages, 4 byte PTEs, each page table fits in page given 1, 2, 3 levels?

4 KB / 4 bytes → 1024 entries per level

1 level: 1024 * 4 KB = 2^{22} bytes = 4 MB

2 levels: 1024 * 4 MB = 2^{32} bytes = 4 GB

3 levels: 1024 * 4 GB = 2^{42} bytes = 4 TB

Effective VA size?
3 levels enough for 64-bit VA?

Why are 4 B PTEs unlikely in this scenario?

Full System with TLBs

On TLB miss: lookups with more levels more expensive

How much does a miss cost?

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211
How many physical accesses for each instruction?

(Ignore previous ops changing TLB)

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

(a) 0xAA10: movl 0x1111, %edi

8 accesses

(b) 0xBB13: addl $0x3, %edi

1 access

(c) 0x0519: movl %edi, 0xFF10

5 accesses

Page Tables Summary

- Linear page tables require too much contiguous memory
  - Wasted space with invalid entries
  - And irregular (non-page) size create external fragmentation
- 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 hw format
  - Multi-level page tables used in x86 architecture
  - Each page table fits within a page
  - Page directory indexes over a process' page tables