

Virtual Memory 2

Hakim Weatherspoon

CS 3410, Spring 2011

Computer Science

Cornell University

Announcements

PA3 available. Due Tuesday, April 19th

- Work with **pairs**
- Be responsible with new knowledge
- **Scheduling a games night, possibly Friday, April 22nd**

Next five weeks

- One homeworks and two projects
- Prelim2 will be Thursday, April 28th
- PA4 will be final project (no final exam)

Goals for Today

Virtual Memory

- Address Translation
 - Pages, page tables, and memory mgmt unit
- Paging
- Role of Operating System
 - Context switches, working set, shared memory
- Performance
 - How slow is it
 - Making virtual memory fast
 - Translation lookaside buffer (TLB)
- Virtual Memory Meets Caching

Address Translation

Pages, Page Tables, and the Memory Management Unit (MMU)

Address Translation

Attempt #1: How does MMU translate addresses?

```
paddr = PageTable[vaddr];
```

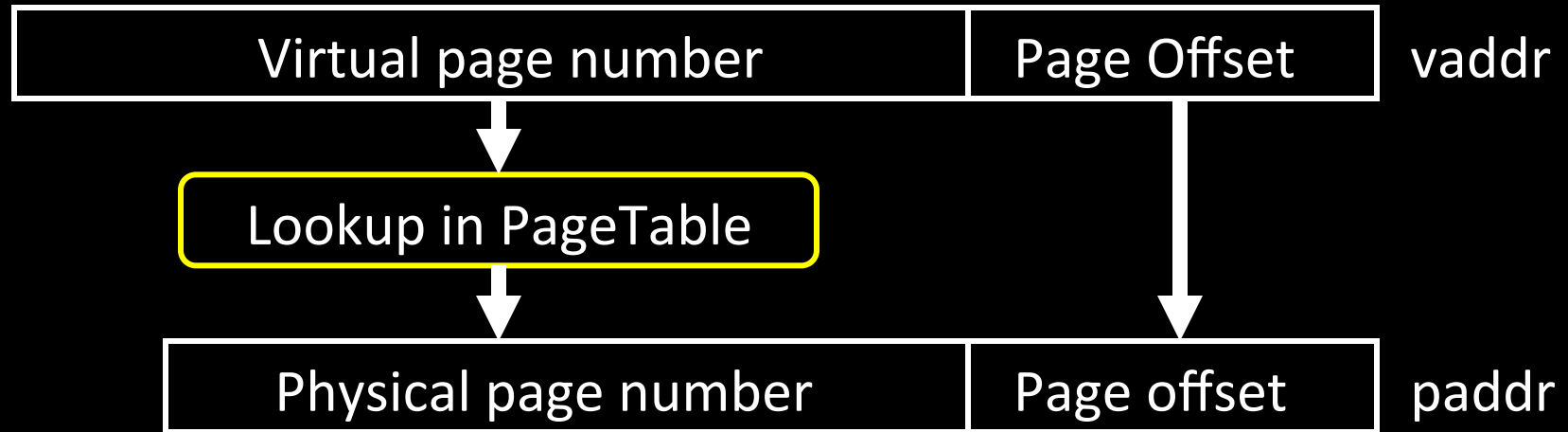
Granularity?

- Per word...
- Per block...
- Variable.....

Typical:

- 4KB – 16KB **pages**
- 4MB – 256MB **jumbo pages**

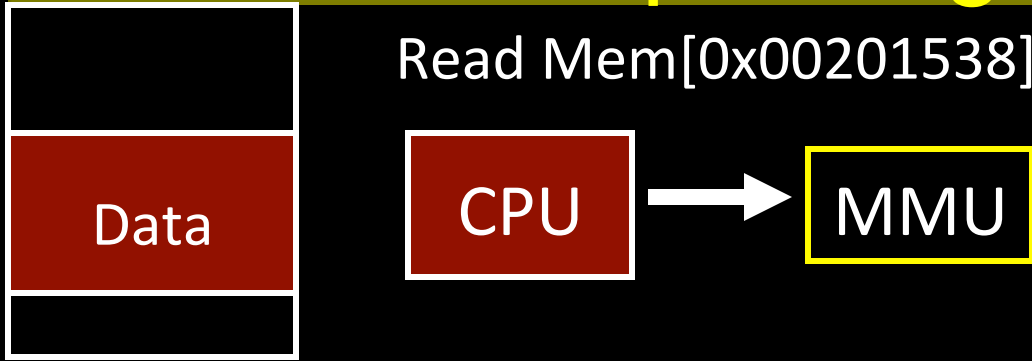
Address Translation



Attempt #1: For any access to virtual address:

- Calculate **virtual page number** and **page offset**
- Lookup **physical page number** at `PageTable[vpn]`
- Calculate physical address as `ppn:offset`

Simple PageTable



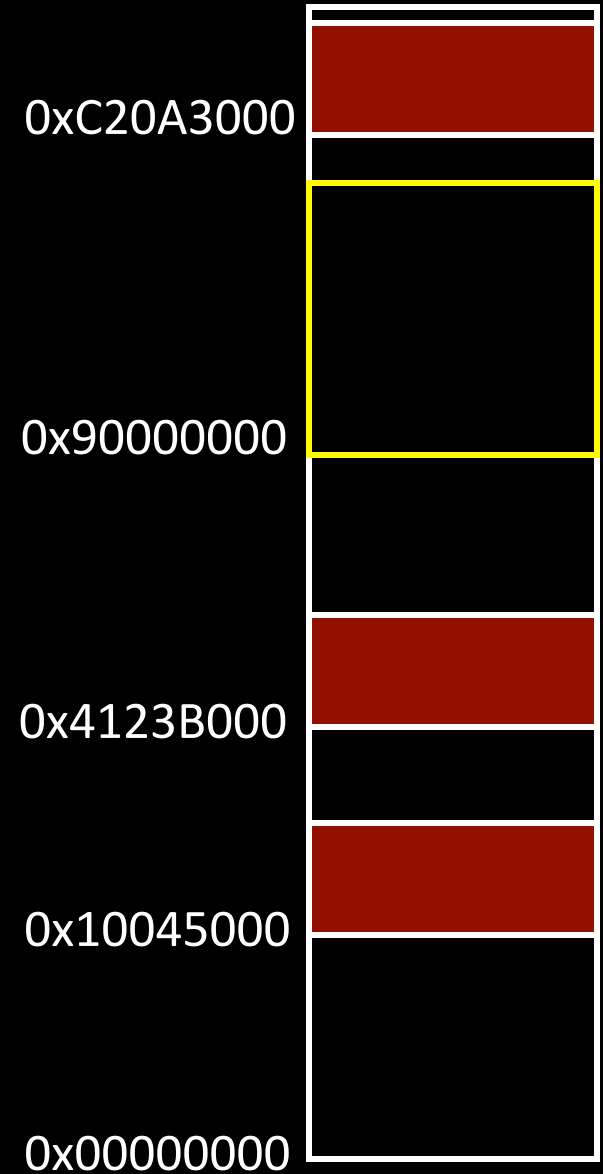
Q: Where to store page tables?

A: In memory, of course...

Special *page table base register*

(CR3:PTBR on x86)

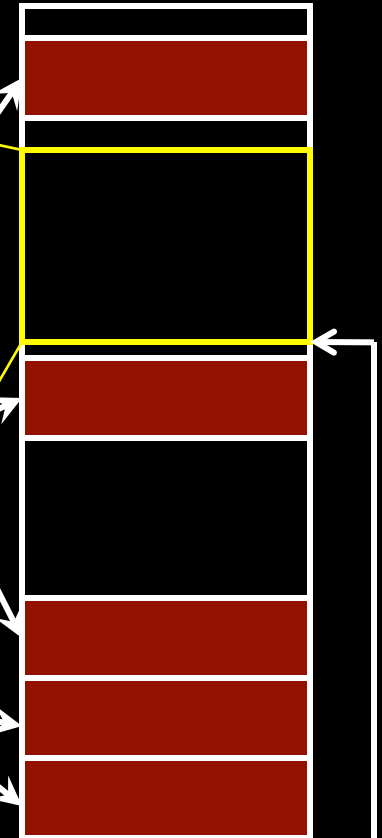
(Cop0:ContextRegister on MIPS)



Summary

Physical Page
Number

	0x10045	●
	0xC20A3	●
	0x4123B	●
	0x00000	●
	0x20340	●



vaddr



Page Size Example

Overhead for VM Attempt #1 (example)

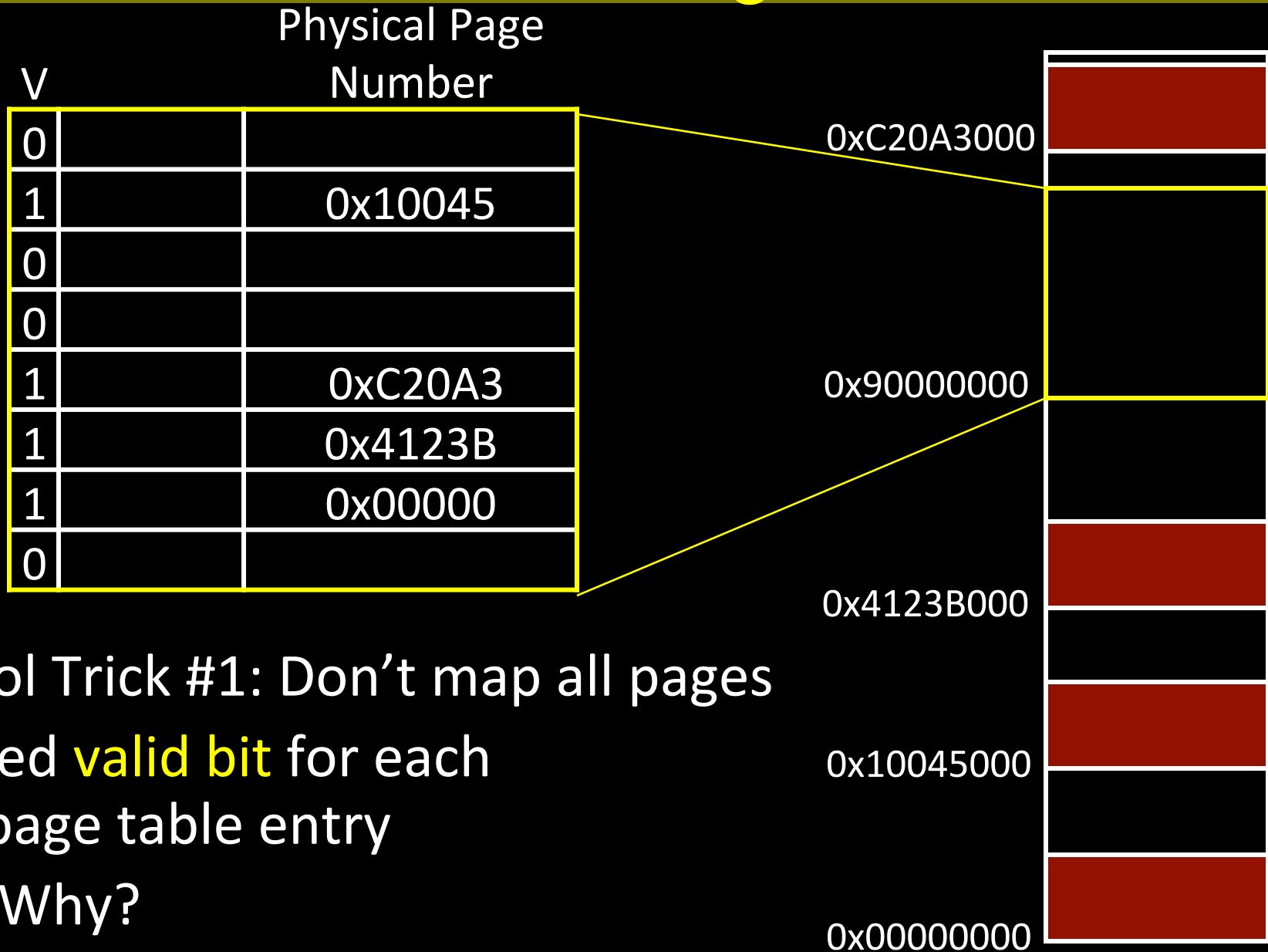
Virtual address space (for each process):

- total memory: 2^{32} bytes = 4GB
- page size: 2^{12} bytes = 4KB
- entries in PageTable?
- size of PageTable?

Physical address space:

- total memory: 2^{29} bytes = 512MB
- overhead for 10 processes?

Invalid Pages



Cool Trick #1: Don't map all pages

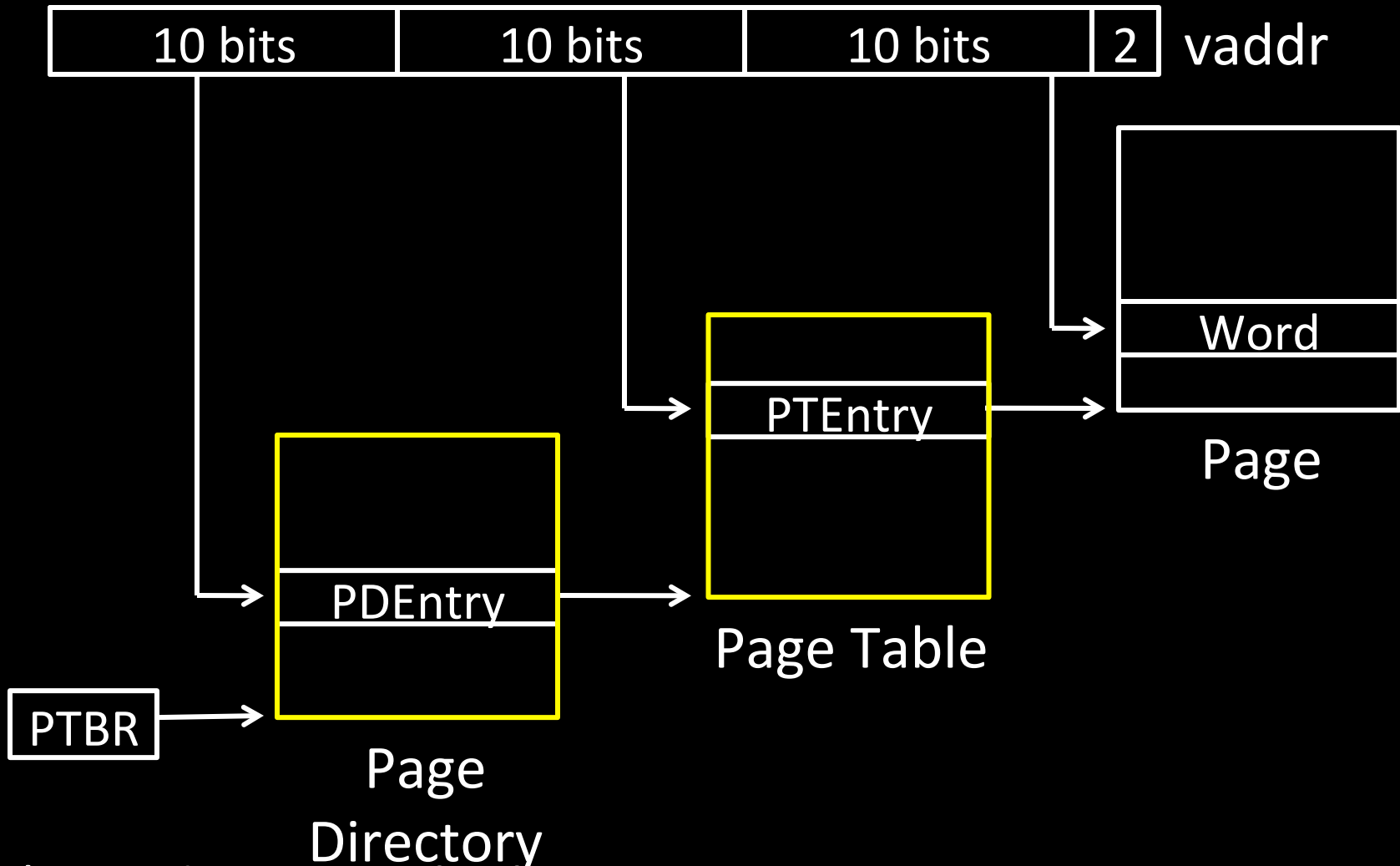
Need **valid bit** for each page table entry

Q: Why?

Beyond Flat Page Tables

Assume most of PageTable is empty

How to translate addresses? Multi-level PageTable



* x86 does exactly this

Page Permissions

Physical Page

V	R	W	X	Physical Page Number
0				
1				0x10045
0				
0				
1				0xC20A3
1				0x4123B
1				0x00000
0				

0xC20A3000

0x90000000

0x4123B000

0x10045000

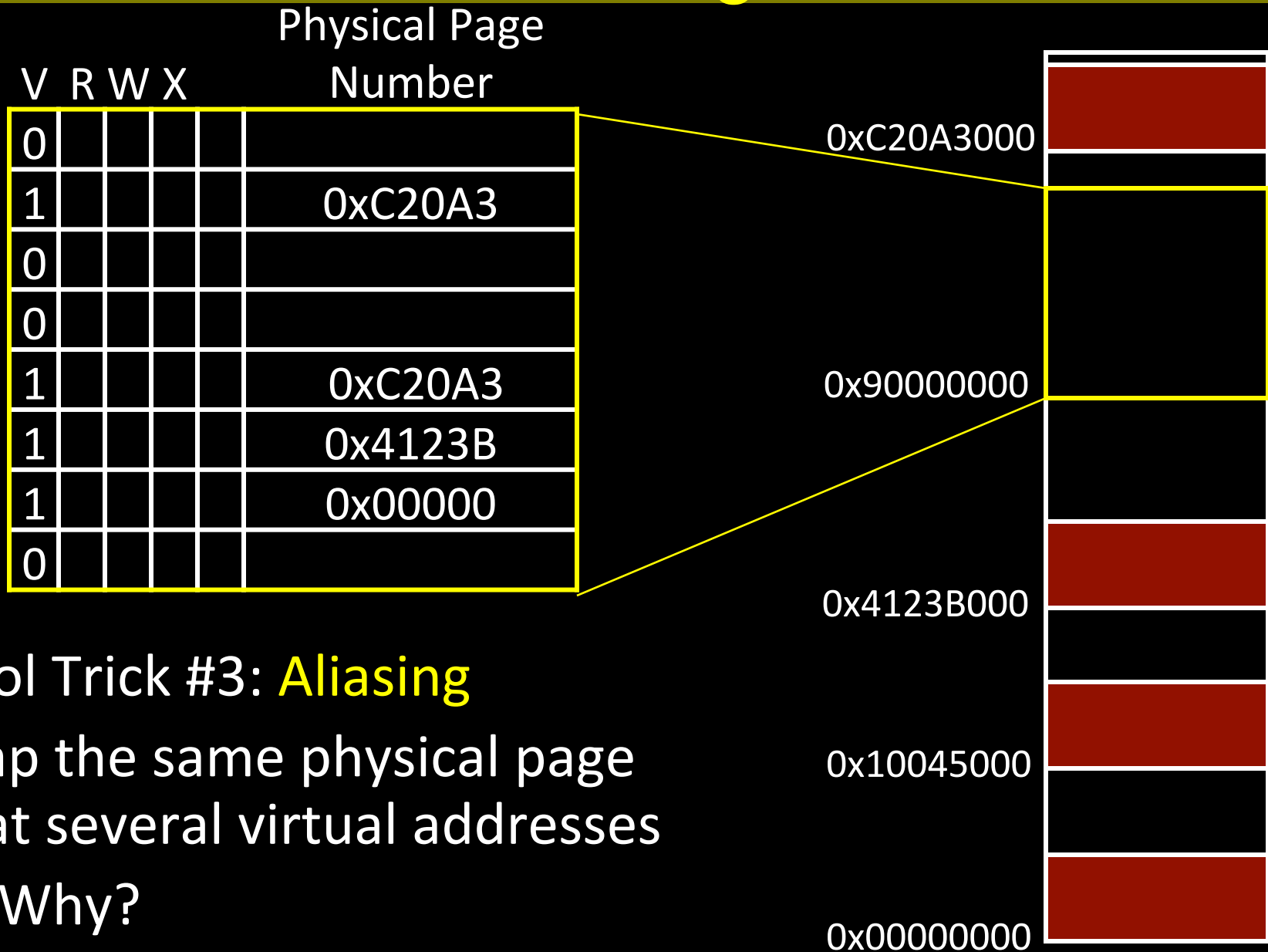
0x00000000

Cool Trick #2: Page permissions!

Keep **R, W, X permission bits** for each page table entry

Q: Why?

Aliasing



Cool Trick #3: **Aliasing**

Map the same physical page
at several virtual addresses

Q: Why?

Paging

Paging

Can we run process larger than physical memory?

- The “virtual” in “virtual memory”

View memory as a “cache” for secondary storage

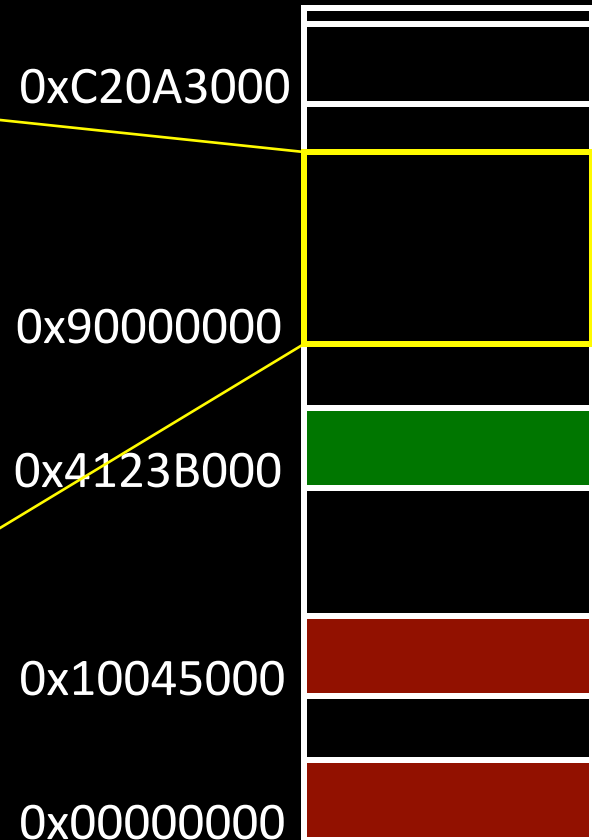
- **Swap** memory pages out to disk when not in use
- **Page** them back in when needed

Assumes Temporal/Spatial Locality

- Pages used recently most likely to be used again soon

Paging

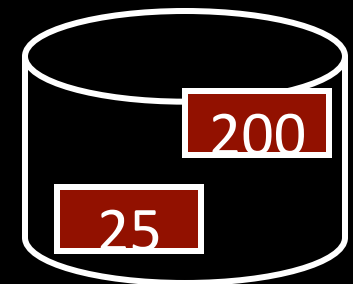
Physical Page					
V	R	W	X	D	Number
0					invalid
1				0	0x10045
0					invalid
0					invalid
0				0	disk sector 200
0				0	disk sector 25
1				1	0x00000
0					invalid



Cool Trick #4: **Paging/Swapping**

Need more bits:

Dirty, RecentlyUsed, ...



Role of the Operating System

Context switches, working set,
shared memory

sbrk

Suppose Firefox needs a new page of memory

(1) Invoke the Operating System

```
void *sbrk(int nbytes);
```

(2) OS finds a free page of physical memory

- clear the page (fill with zeros)
- add a new entry to Firefox's PageTable

Context Switch

Suppose Firefox is idle, but Skype wants to run

(1) Firefox invokes the Operating System

```
int sleep(int nseconds);
```

(2) OS saves Firefox's registers, load skype's

- (more on this later)

(3) OS changes the CPU's Page Table Base Register

- Cop0:ContextRegister / CR3:PDBR

(4) OS returns to Skype

Shared Memory

Suppose Firefox and Skype want to share data

(1) OS finds a free page of physical memory

- clear the page (fill with zeros)
- add a new entry to Firefox's PageTable
- add a new entry to Skype's PageTable
 - can be same or different vaddr
 - can be same or different page permissions

Multiplexing

Suppose Skype needs a new page of memory, but Firefox is hogging it all

(1) Invoke the Operating System

```
void *sbrk(int nbytes);
```

(2) OS can't find a free page of physical memory

- Pick a page from Firefox instead (or other process)

(3) If page table entry has dirty bit set...

- Copy the page contents to disk

(4) Mark Firefox's page table entry as "on disk"

- Firefox will fault if it tries to access the page

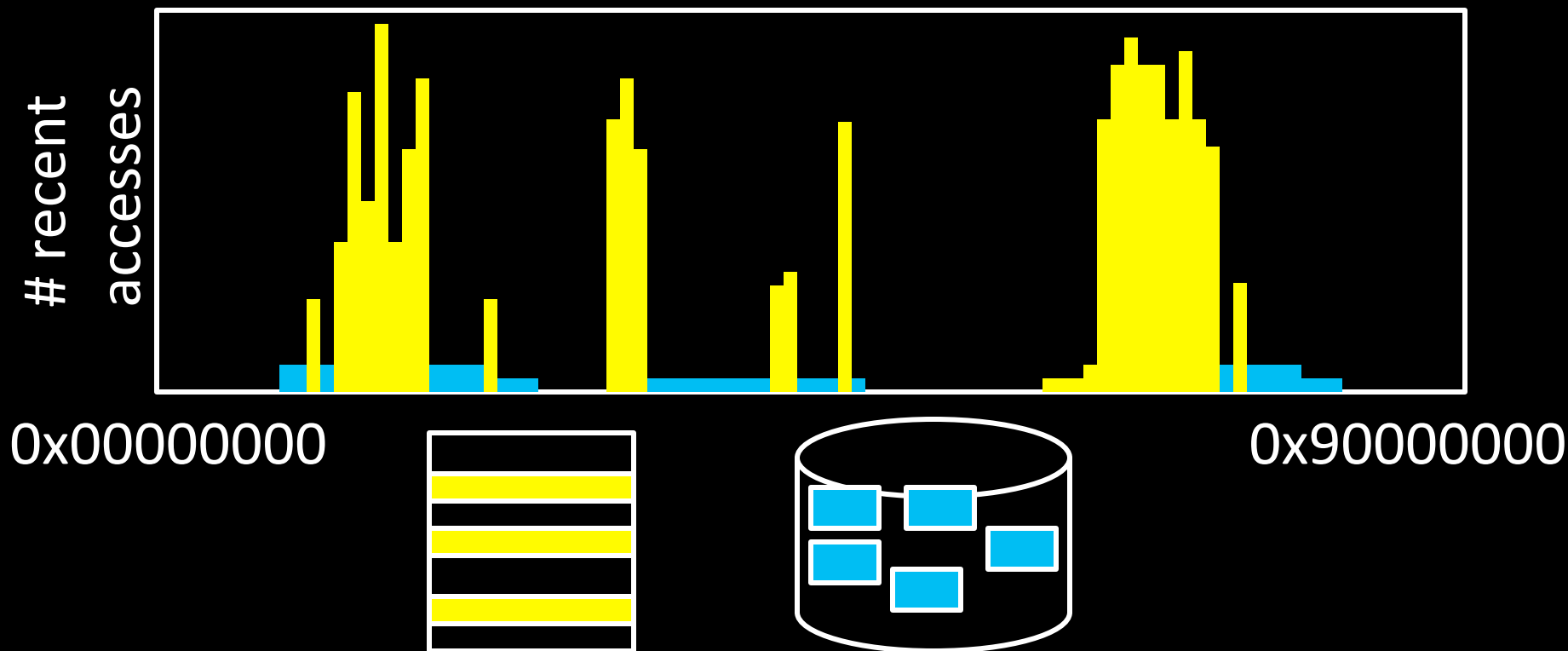
(5) Give the newly freed physical page to Skype

- clear the page (fill with zeros)
- add a new entry to Skyp's PageTable

Paging Assumption 1

OS **multiplexes** physical memory among processes

- assumption # 1:
processes use only a few pages at a time
- **working set** = set of process's recently actively pages



Reasons for Thrashing

P1

working set

swapped

mem

disk

Q: What if working set is too large?

Case 1: Single process using too many pages

working set

swapped

mem

disk

Case 2: Too many processes

WS

WS

WS

WS

WS

WS

mem

disk

Thrashing

Thrashing b/c working set of process (or processes) greater than physical memory available

- Firefox steals page from Skype
- Skype steals page from Firefox
- I/O (disk activity) at 100% utilization
 - But no useful work is getting done

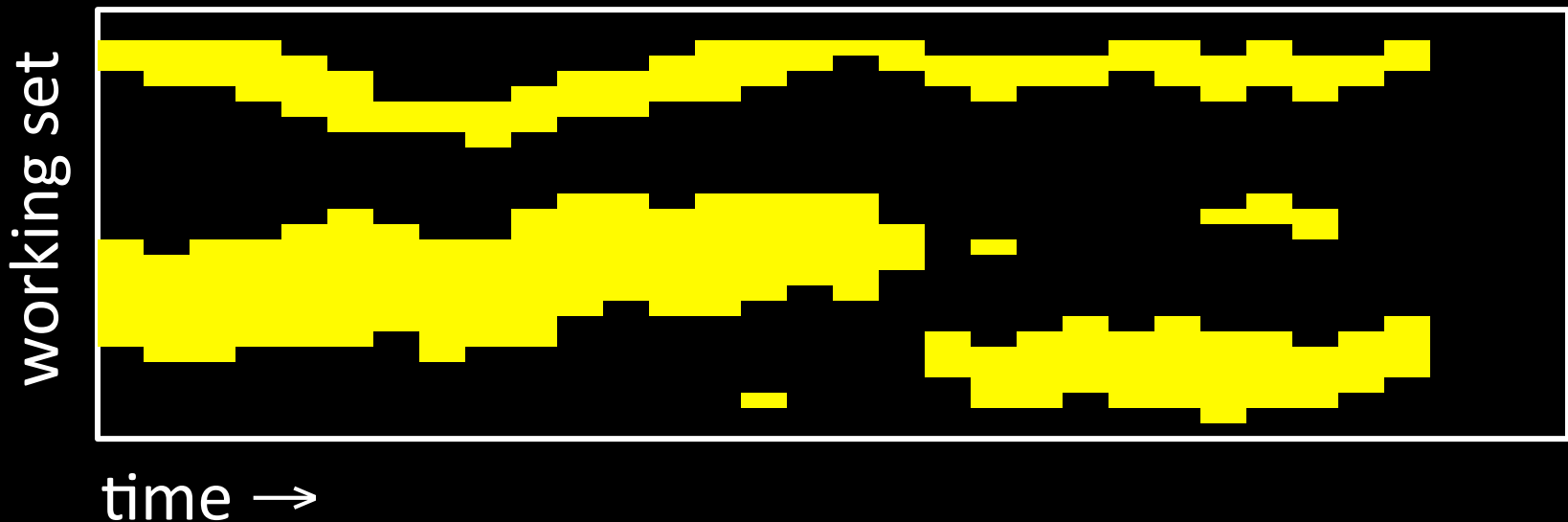
Ideal: Size of disk, speed of memory (or cache)

Non-ideal: Speed of disk

Paging Assumption 2

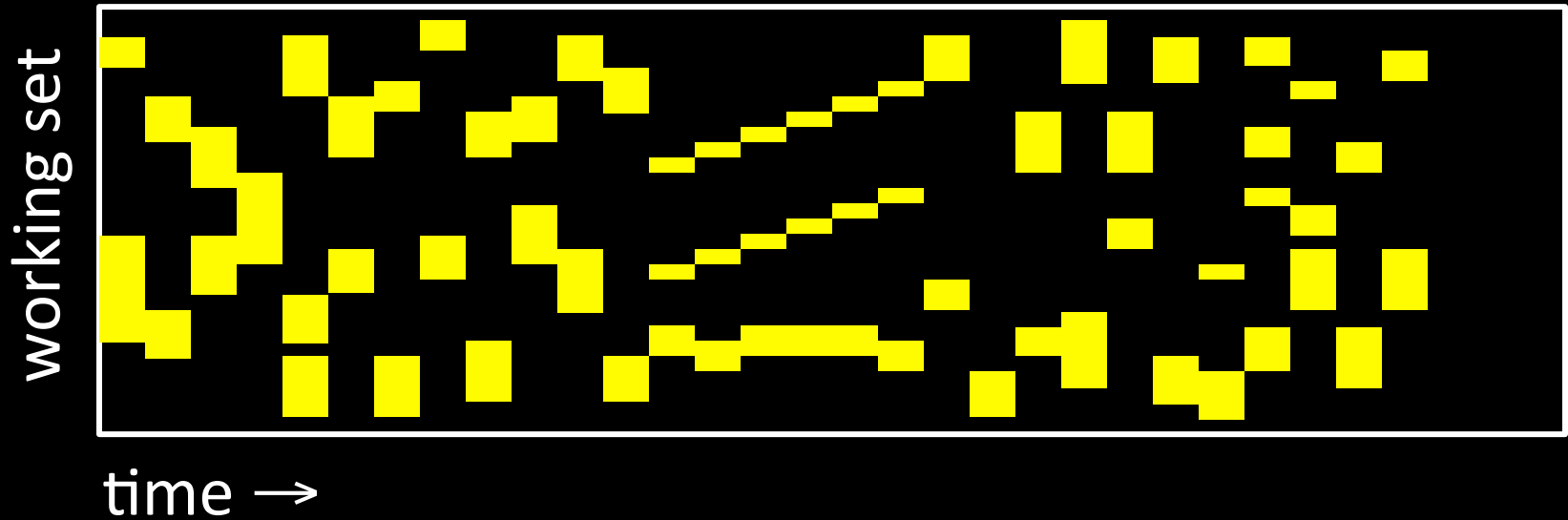
OS **multiplexes** physical memory among processes

- assumption # 2:
recent accesses predict future accesses
- working set usually **changes slowly** over time



More Thrashing

Q: What if working set changes rapidly or unpredictably?



A: Thrashing b/c recent accesses don't predict future accesses

Preventing Thrashing

How to prevent thrashing?

- User: Don't run too many apps
- Process: efficient and predictable mem usage
- OS: Don't over-commit memory, memory-aware scheduling policies, etc.

Performance

Performance

Virtual Memory Summary

PageTable for each process:

- 4MB contiguous in physical memory, or multi-level, ...
- every load/store translated to physical addresses
- page table miss = *page fault*
load the swapped-out page and retry instruction,
or kill program if the page really doesn't exist,
or tell the program it made a mistake

Page Table Review

x86 Example: 2 level page tables, assume...

32 bit vaddr, 32 bit paddr

4k PDir, 4k PTables, 4k Pages

PTBR

PDE
PDE
PDE
PDE

PTE
PTE
PTE
PTE

Q: How many bits for a page number?

A: 20

Q: What is stored in each PageTableEntry?

A: ppn, valid/dirty/r/w/x/...

Q: What is stored in each PageDirEntry?

A: ppn, valid/?/...

Q: How many entries in a PageDirectory?

A: 1024 four-byte PDEs

Q: How many entries in each PageTable?

A: 1024 four-byte PTEs

Page Table Example

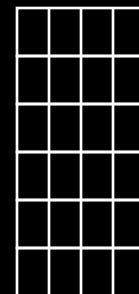
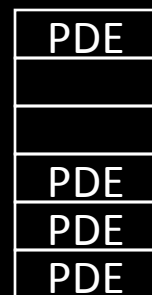
x86 Example: 2 level page tables, assume...

32 bit vaddr, 32 bit paddr

4k PDir, 4k PTables, 4k Pages

PTBR = 0x10005000 (physical)

PTBR



Write to virtual address **0x7192a44c**...

Q: Byte offset in page?

PT Index?

PD Index?

(1) PageDir is at 0x10005000, so...

Fetch PDE from physical address $0x10005000 + 4 * PD I$

- suppose we get {0x12345, v=1, ...}

(2) PageTable is at 0x12345000, so...

Fetch PTE from physical address $0x12345000 + 4 * PT I$

- suppose we get {0x14817, v=1, d=0, r=1, w=1, x=0, ...}

(3) Page is at 0x14817000, so...

Write data to physical address 0x1481744c

Also: update PTE with d=1

Performance

Virtual Memory Summary

PageTable for each process:

- 4MB contiguous in physical memory, or multi-level, ...
- every load/store translated to physical addresses
- page table miss: load a swapped-out page and retry instruction, or kill program

Performance?

- terrible: memory is already slow
translation makes it slower

Solution?

- A cache, of course

Making Virtual Memory Fast

The Translation Lookaside Buffer (TLB)

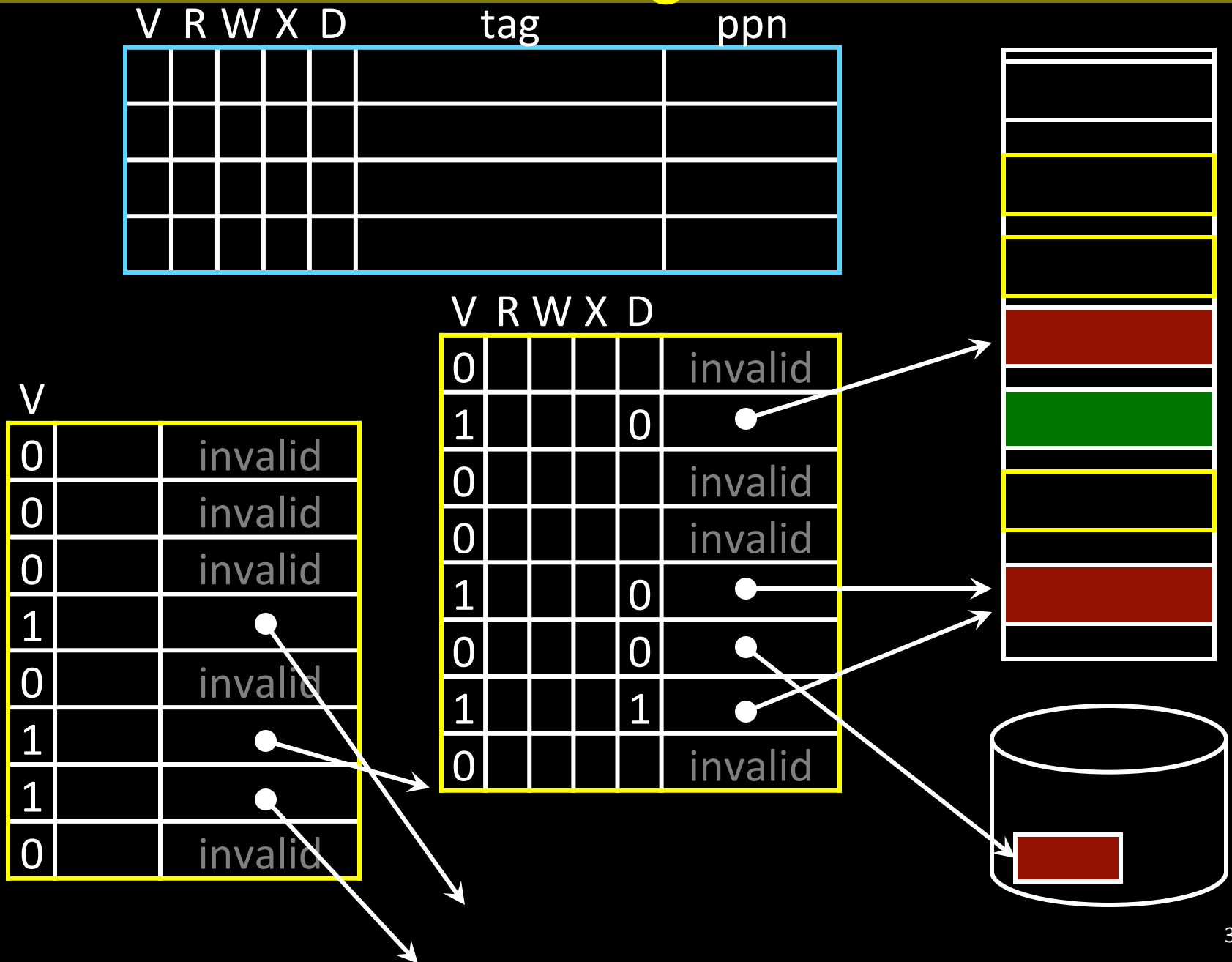
Translation Lookaside Buffer (TLB)

Hardware Translation Lookaside Buffer (TLB)

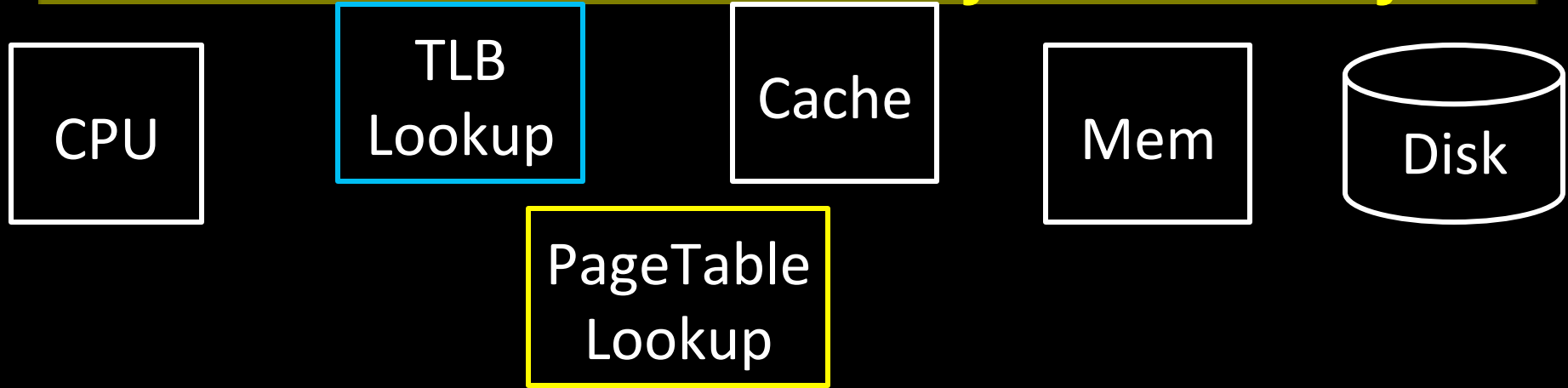
A small, very fast cache of recent address mappings

- TLB hit: avoids PageTable lookup
- TLB miss: do PageTable lookup, cache result for later

TLB Diagram



A TLB in the Memory Hierarchy



(1) Check TLB for vaddr (~ 1 cycle)

(2) TLB Hit

- compute paddr, send to cache

(2) TLB Miss: traverse PageTables for vaddr

(3a) PageTable has valid entry for in-memory page

- Load PageTable entry into TLB; try again (tens of cycles)

(3b) PageTable has entry for swapped-out (on-disk) page

- Page Fault: load from disk, fix PageTable, try again (millions of cycles)

(3c) PageTable has invalid entry

- Page Fault: kill process

TLB Coherency

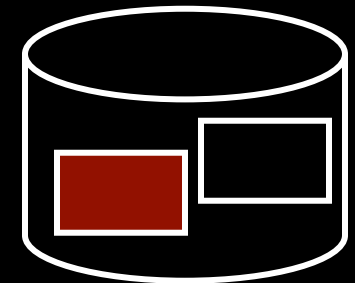
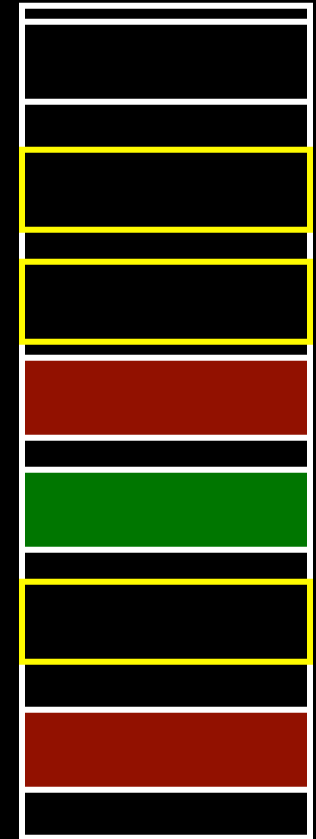
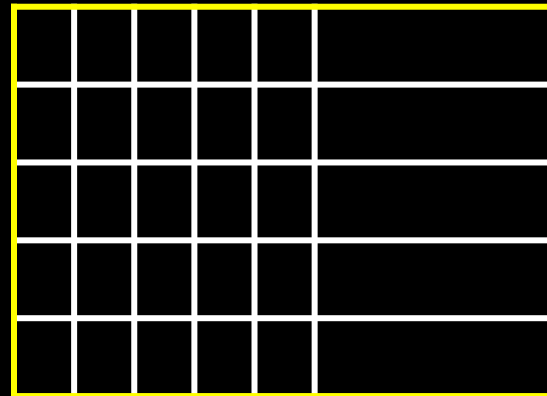
TLB Coherency: What can go wrong?

A: PageTable or PageDir contents change

- swapping/paging activity, new shared pages, ...

A: Page Table Base Register changes

- context switch between processes



Translation Lookaside Buffers (TLBs)

When PTE changes, PDE changes, PTBR changes....

Full Transparency: **TLB coherency in hardware**

- Flush TLB whenever PTBR register changes
[easy – why?]
- Invalidate entries whenever PTE or PDE changes
[hard – why?]

TLB coherency in software

If TLB has a no-write policy...

- OS invalidates entry after OS modifies page tables
- OS flushes TLB whenever OS does context switch

TLB Parameters

TLB parameters (typical)

- very small (64 – 256 entries), so very fast
- fully associative, or at least set associative
- tiny block size: why?

Intel Nehalem TLB (example)

- 128-entry L1 Instruction TLB, 4-way LRU
- 64-entry L1 Data TLB, 4-way LRU
- 512-entry L2 Unified TLB, 4-way LRU

Virtual Memory meets Caching

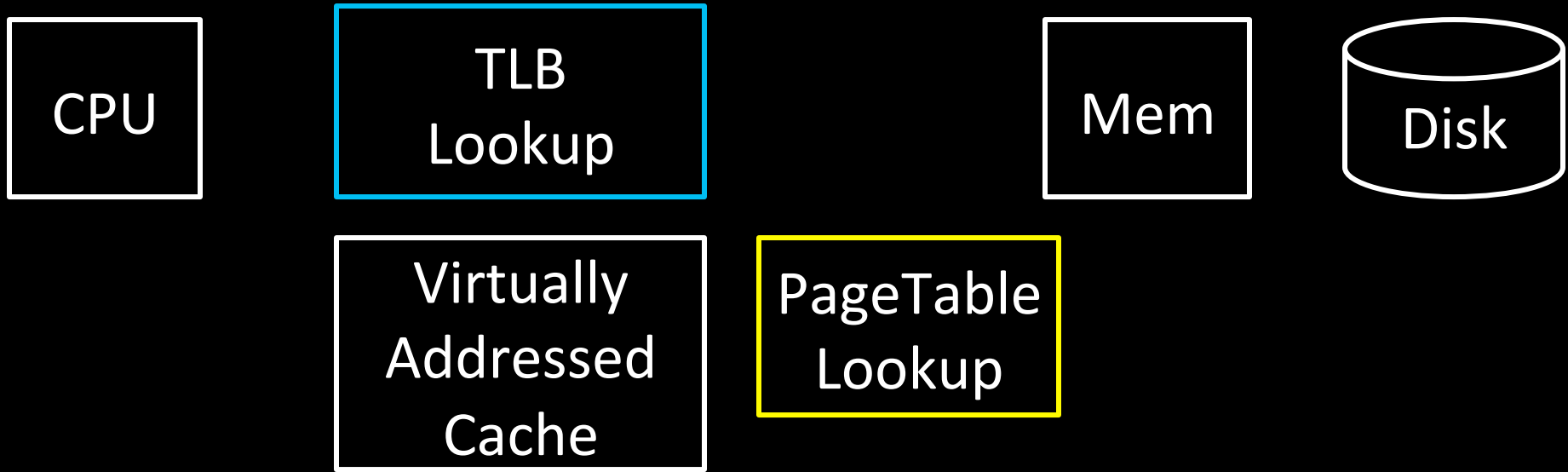
Virtually vs. physically addressed caches

Virtually vs. physically tagged caches

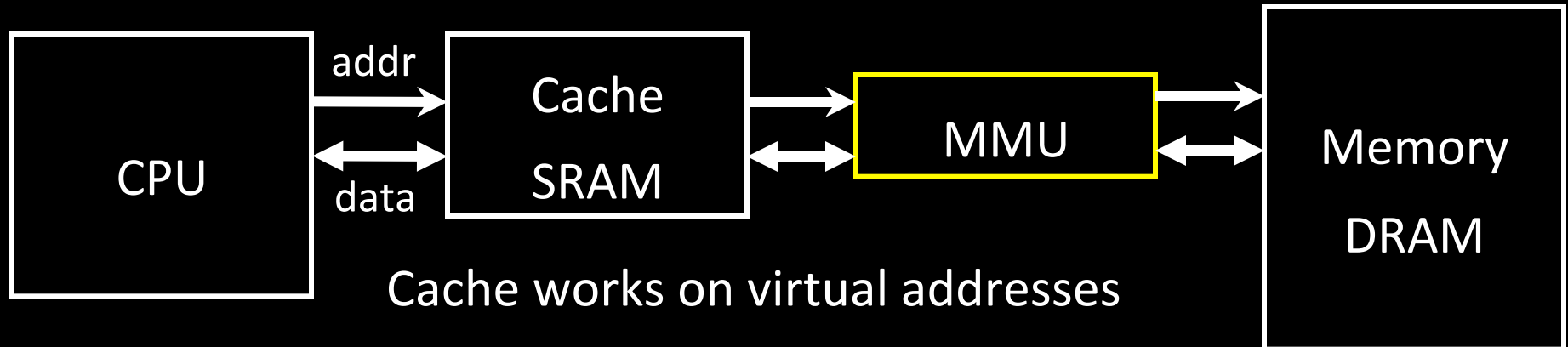
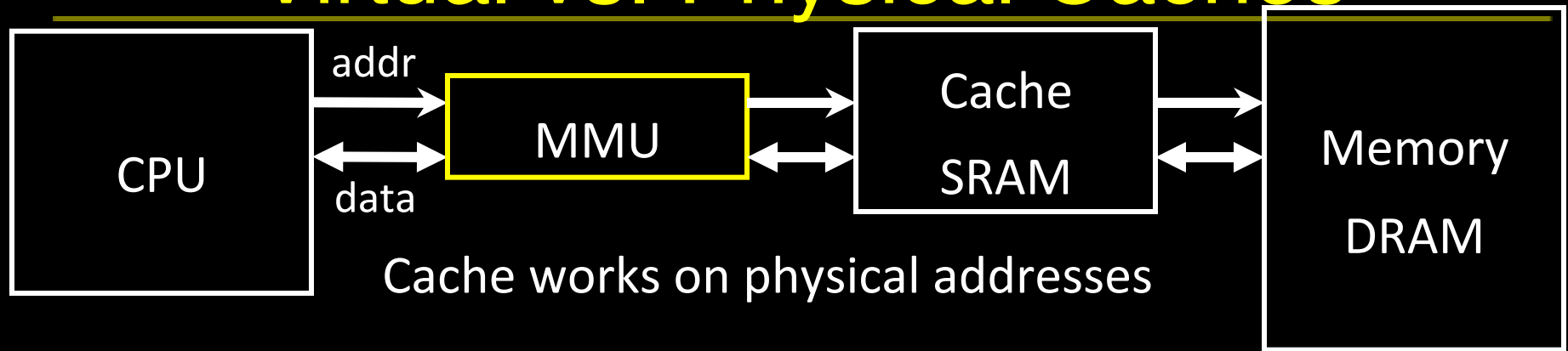
Virtually Addressed Caching

Q: Can we remove the TLB from the critical path?

A: Virtually-Addressed Caches



Virtual vs. Physical Caches



Q: What happens on context switch?

Q: What about virtual memory aliasing?

Q: So what's wrong with physically addressed caches?

Indexing vs. Tagging

Physically-Addressed Cache

Virtually-Addressed Cache

first

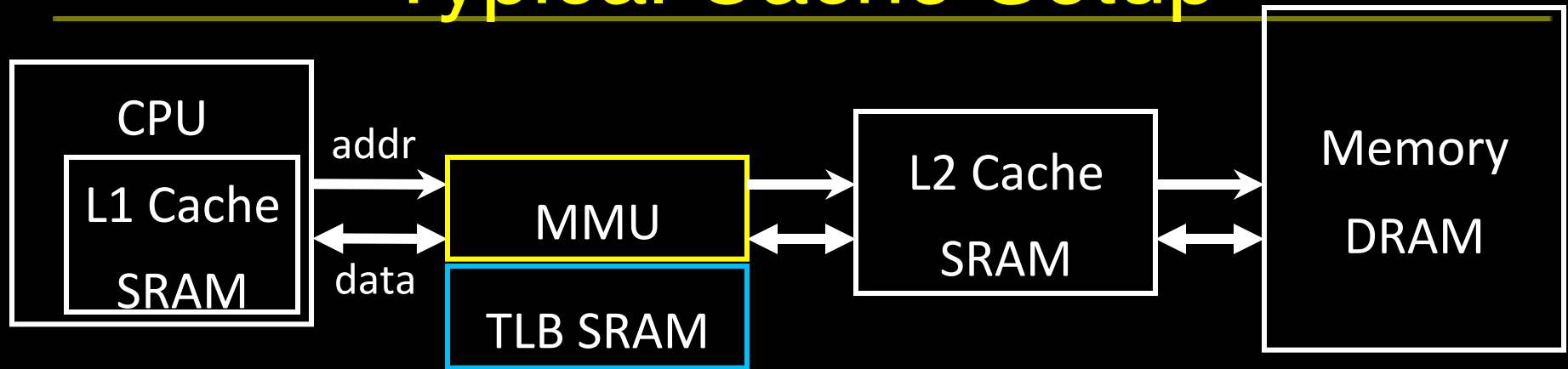
Virtually-Indexed, Virtually Tagged Cache

- fast: start TLB lookup before cache lookup finishes
- PageTable changes (paging, context switch, etc.)
 - need to purge stale cache lines (how?)
- Synonyms (two virtual mappings for one physical page)
 - could end up in cache twice (very bad!)

Virtually-Indexed, Physically Tagged Cache

- ~fast: TLB lookup in parallel with cache lookup
- PageTable changes → no problem: phys. tag mismatch
- Synonyms → search and evict lines with same phys. tag

Typical Cache Setup



Typical L1: On-chip **virtually** addressed, **physically** tagged

Typical L2: On-chip **physically** addressed

Typical L3: On-chip ...

Caches/TLBs/MM

Caches, Virtual Memory, & TLBs

Where can block be placed?

- Direct, n-way, fully associative

What block is replaced on miss?

- LRU, Random, LFU, ...

How are writes handled?

- No-write (w/ or w/o automatic invalidation)
- Write-back (fast, block at time)
- Write-through (simple, reason about consistency)

Summary of Cache Design Parameters

	L1	Paged Memory	TLB
Size (blocks)	1/4k to 4k	16k to 1M	64 to 4k
Size (kB)	16 to 64	1M to 4G	2 to 16
Block size (B)	16-64	4k to 64k	4-32
Miss rates	2%-5%	10^{-4} to $10^{-5}\%$	0.01% to 2%
Miss penalty	10-25	10M-100M	100-1000