Virtual Memory 1

Hakim Weatherspoon
CS 3410, Spring 2011
Computer Science
Cornell University

P & H Chapter 5.4 (up to TLBs)

Announcements

HW3 available due *today* Tuesday

- HW3 has been updated. Use updated version.
- Work with alone
- Be responsible with new knowledge

PA3 available later today or by tomorrow

- Work in pairs

Next five weeks

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

Goals for Today

Title says Virtual Memory, but really finish caches: writes

Introduce idea of Virtual Memory

Cache Design

Need to determine parameters:

- Cache size
- Block size (aka line size)
- Number of ways of set-associativity (1, N, ∞)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy

A Real Example

Dual-core 3.16GHz Intel (purchased in 2009)

```
> dmidecode -t cache
Cache Information
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Back
        Installed Size: 128 KB
        Error Correction Type: None
Cache Information
        Configuration: Enabled, Not Socketed, Level 2
        Operational Mode: Varies With Memory Address
        Installed Size: 6144 KB
        Error Correction Type: Single-bit ECC
> cd /sys/devices/system/cpu/cpu0; grep cache/*/*
cache/index0/level:1
cache/index0/type:Data
cache/index0/ways_of_associativity:8
cache/index0/number_of_sets:64
cache/index0/coherency_line_size:64
cache/index0/size:32K
cache/index1/level:1
cache/index1/type:Instruction
cache/index1/ways_of_associativity:8
cache/index1/number_of_sets:64
cache/index1/coherency_line_size:64
cache/index1/size:32K
cache/index2/level:2
cache/index2/type:Unified
cache/index2/shared_cpu_list:0-1
cache/index2/ways_of_associativity:24
cache/index2/number_of_sets:4096
cache/index2/coherency_line_size:64
cache/index2/size:6144K
```

A Real Example

Dual-core 3.16GHz Intel
(purchased in 2009)

Dual 32K L1 Instruction caches

- 8-way set associative
- 64 sets
- 64 byte line size

Dual 32K L1 Data caches

- Same as above

Single 6M L2 Unified cache

- 24-way set associative (!!!)
- 4096 sets
- 64 byte line size

4GB Main memory

1TB Disk

Basic Cache Organization

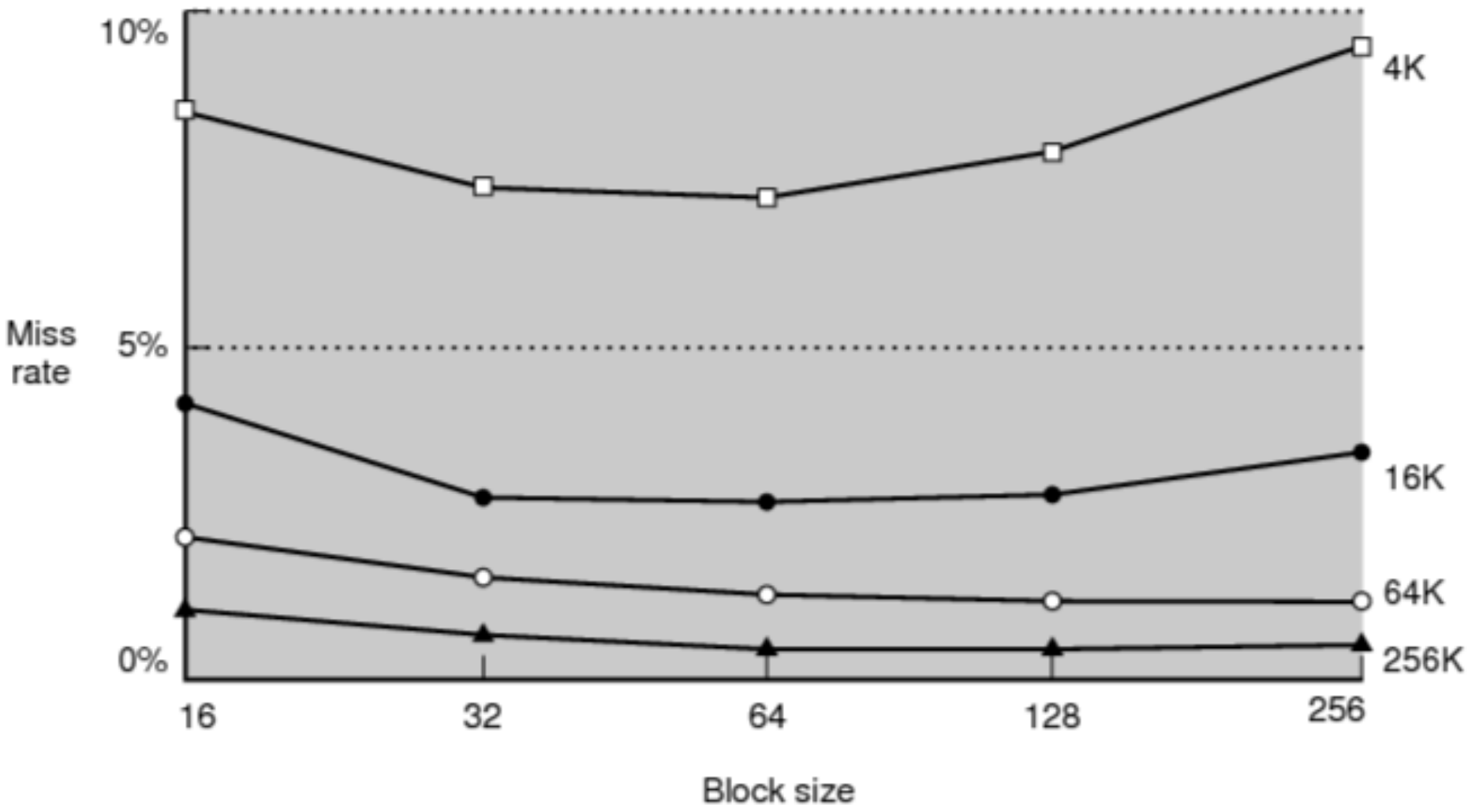
Q: How to decide block size?

A: Try it and see

But: depends on cache size, workload, associativity, …

Experimental approach!

Experimental Results



Tradeoffs

For a given total cache size,

larger block sizes mean….

- fewer lines
- so fewer tags (and smaller tags for associative caches)
- so less overhead
- and fewer cold misses (within-block “prefetching”)

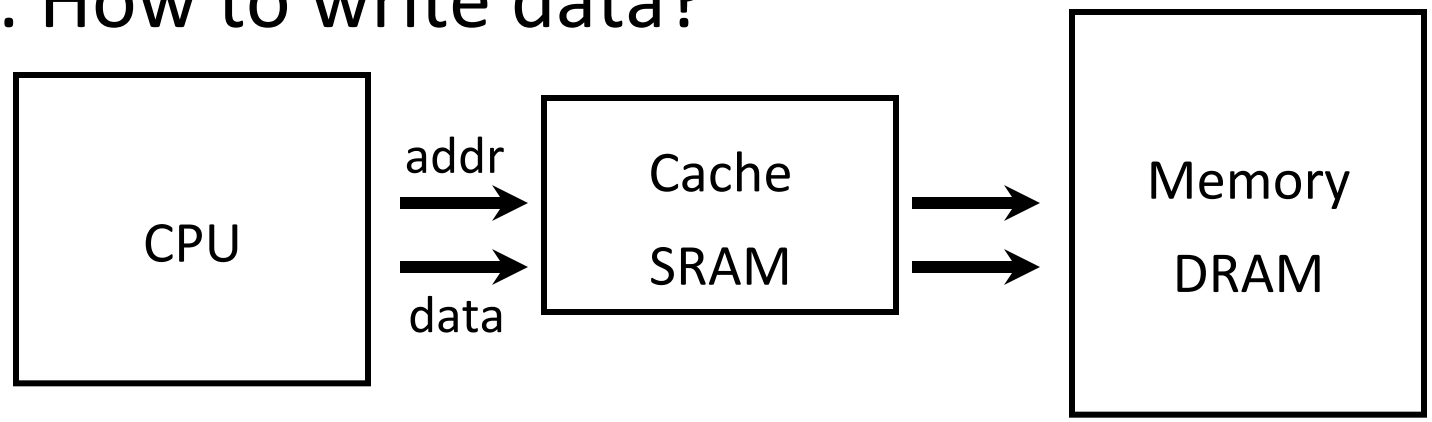
But also…

- fewer blocks available (for scattered accesses!)
- so more conflicts
- and larger miss penalty (time to fetch block)

Writing with Caches

Cached Write Policies

Q: How to write data?



If data is already in the cache…

No-Write

- writes invalidate the cache and go directly to memory

Write-Through

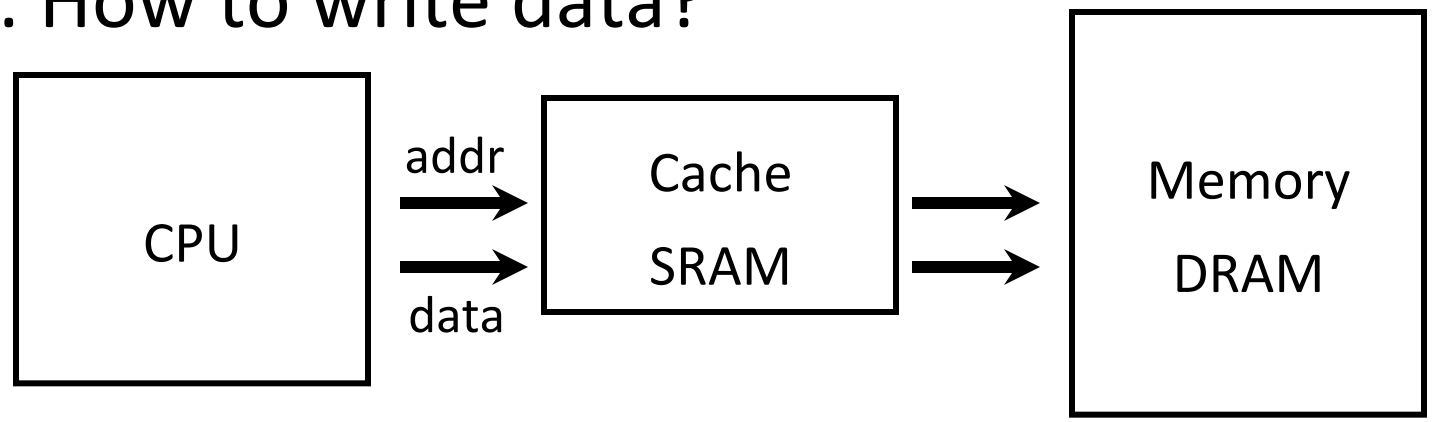
- writes go to main memory and cache

Write-Back

- CPU writes only to cache
- cache writes to main memory later (when block is evicted)

Write Allocation Policies

Q: How to write data?



If data is not in the cache…

Write-Allocate

- allocate a cache line for new data (and maybe write-through)

No-Write-Allocate

- ignore cache, just go to main memory

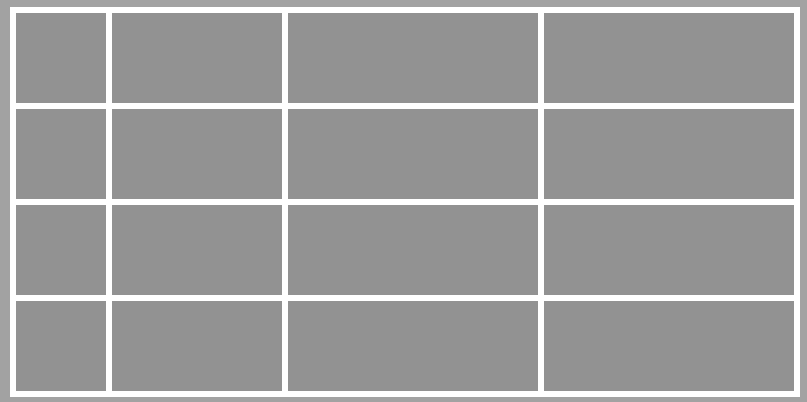
A Simple Direct Mapped Cache

Using **byte addresses** in this example! Addr Bus = 5 bits

Processor

lb $1 ← M[1]
lb $2 ← M[7]
sb $2 → M[0]
sb $1 → M[5]
lb $2 ← M[9]
sb $1 → M[5]
sb $1 → M[0]

Direct Mapped Cache
+ Write-through
+ Write-allocate



Hits: Misses:

Memory

0	**101**
1	**103**
2	**107**
3	**109**
4	**113**
5	**127**
6	**131**
7	**137**
8	**139**
9	**149**
10	**151**
11	**157**
12	**163**
13	**167**
14	**173**
15	**179**
16	**181**

How Many Memory References?

Write-through performance

Each miss (read or write) reads a block from mem

- 5 misses → 10 mem reads

Each store writes an item to mem

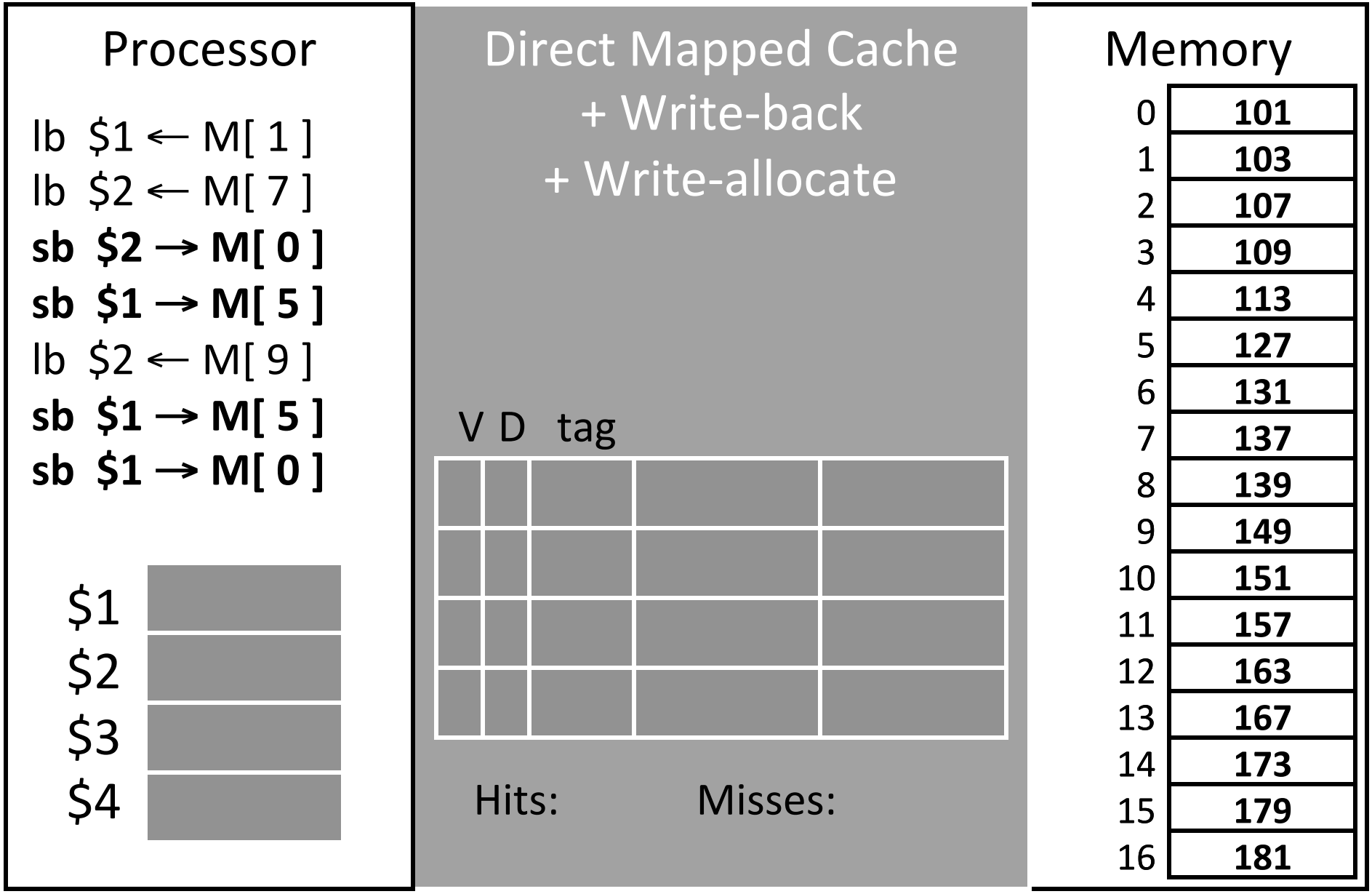
- 4 mem writes

Evictions don't need to write to mem

- no need for dirty bit

A Simple Direct Mapped Cache

Using **byte addresses** in this example! Addr Bus = 5 bits



How Many Memory References?

Write-back performance

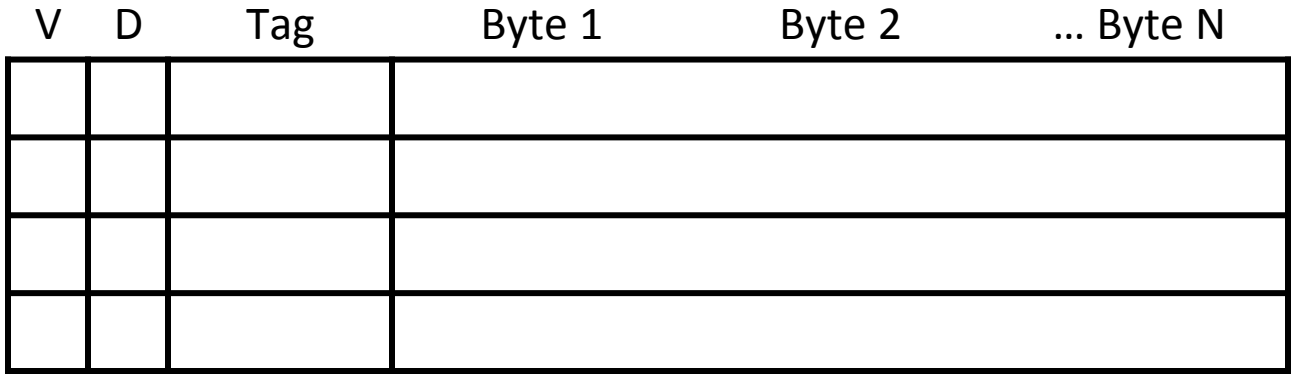
Each miss (read or write) reads a block from mem

- 5 misses → 10 mem reads

Some evictions write a block to mem

- 1 dirty eviction → 2 mem writes
- (+ 2 dirty evictions later → +4 mem writes)
- need a dirty bit

Write-Back Meta-Data



V = 1 means the line has valid data

D = 1 means the bytes are newer than main memory

When allocating line:

- Set V = 1, D = 0, fill in Tag and Data

When writing line:

- Set D = 1

When evicting line:

- If D = 0: just set V = 0
- If D = 1: write-back Data, then set D = 0, V = 0

Performance: An Example

Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

```
for (i=1; i<n; i++)
      A[0] += A[i];



for (i=0; i<n; i++)
      B[i] = A[i]
```

Performance Tradeoffs

Q: Hit time: write-through vs. write-back?

A: Write-through slower on writes.

Q: Miss penalty: write-through vs. write-back?

A: Write-back slower on evictions.

Write Buffering

Q: Writes to main memory are **slow!**

A: Use a write-back buffer

- A small queue holding dirty lines
- Add to end upon eviction
- Remove from front upon completion

Q: What does it help?

A: short bursts of writes (but not sustained writes)

A: fast eviction reduces miss penalty

Write-through vs. Write-back

Write-through is slower

- But simpler (memory always consistent)

Write-back is almost always faster

- write-back buffer hides large eviction cost
- But what about multiple cores with separate caches but sharing memory?

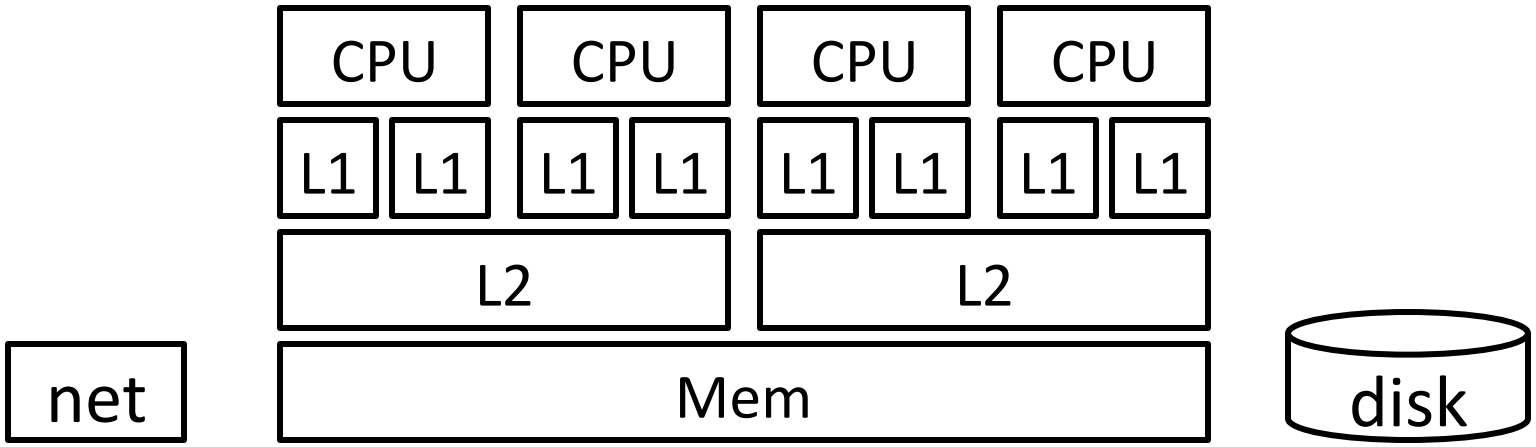
Write-back requires a cache coherency protocol

- Inconsistent views of memory
- Need to “snoop” in each other’s caches
- Extremely complex protocols, very hard to get right

Cache-coherency

Q: Multiple readers and writers?

A: Potentially inconsistent views of memory



Cache coherency protocol

- May need to snoop on other CPU's cache activity
- Invalidate cache line when other CPU writes
- Flush write-back caches before other CPU reads
- Or the reverse: Before writing/reading…
- Extremely complex protocols, very hard to get right

Cache Conscious Programming

Cache Conscious Programming

```
// H = 12, W = 10
int A[H][W];

for(x=0; x < W; x++)
   for(y=0; y < H; y++)
      sum += A[y][x];
```

1	11	21							
		2	12	22					
				3	13	23			
						4	14	24	
								5	15
25									
6	16	26							
		7	17	…					
				8	18				
						9	19		
								10	20

Every access is a cache miss!

(unless *entire* matrix can fit in cache)

Cache Conscious Programming

```
// H = 12, W = 10
int A[H][W];


for(y=0; y < H; y++)
   for(x=0; x < W; x++)
      sum += A[y][x];
```

1	2	3	4	5	6	7	8	9	10
11	12	13	…						

Block size = 4 →

Block size = 8 → 87.5% hit rate

Block size = 16 → 93.75% hit rate

And you can easily prefetch to warm the cache.

Summary

Caching assumptions

- small working set: 90/10 rule
- can predict future: spatial & temporal locality

Benefits

- (big & fast) built from (big & slow) + (small & fast)

Tradeoffs:

associativity, line size, hit cost, miss penalty, hit rate

Summary

Memory performance matters!

- often more than CPU performance
- … because it is the bottleneck, and not improving much
- … because most programs move a LOT of data

Design space is huge

- Gambling against program behavior
- Cuts across all layers:
  users → programs → os → hardware

Multi-core / Multi-Processor is complicated

- Inconsistent views of memory
- Extremely complex protocols, very hard to get right

Virtual Memory

Processor & Memory

CPU address/data bus...

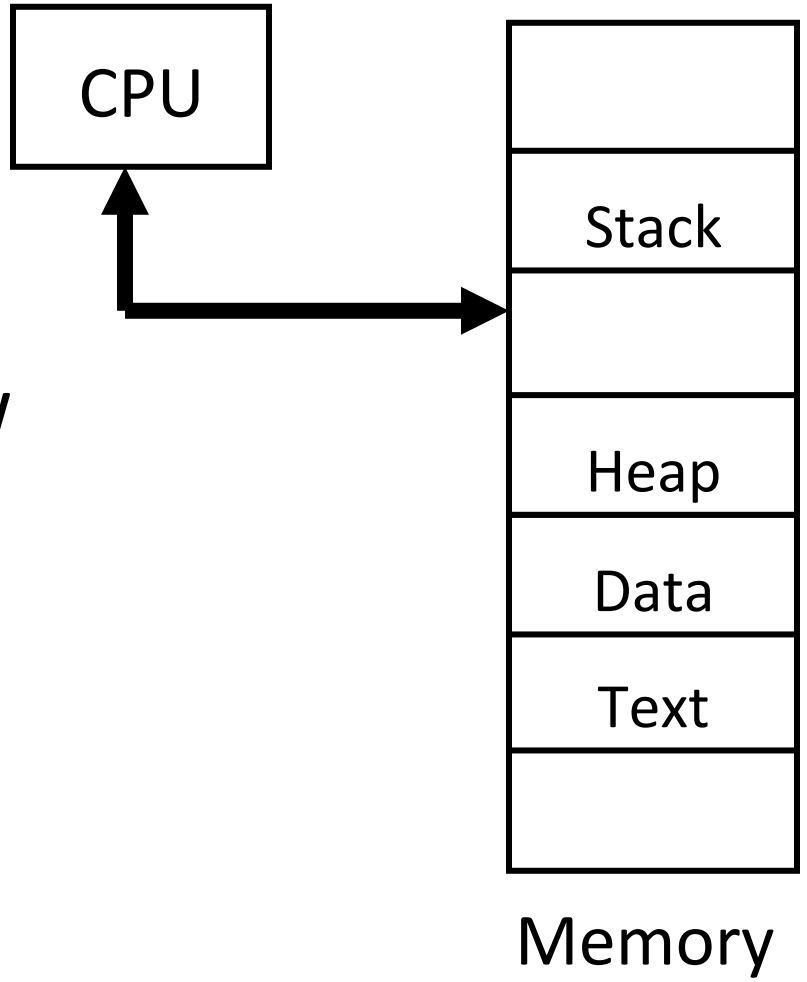
… routed through caches

… to main memory

- Simple, fast, but…

Q: What happens for LW/SW to an invalid location?

- 0x000000000 (NULL)
- uninitialized pointer



Multiple Processes

Running multiple processes…

Time-multiplex a single CPU core (multi-tasking)

- Web browser, skype, office, … all must co-exist

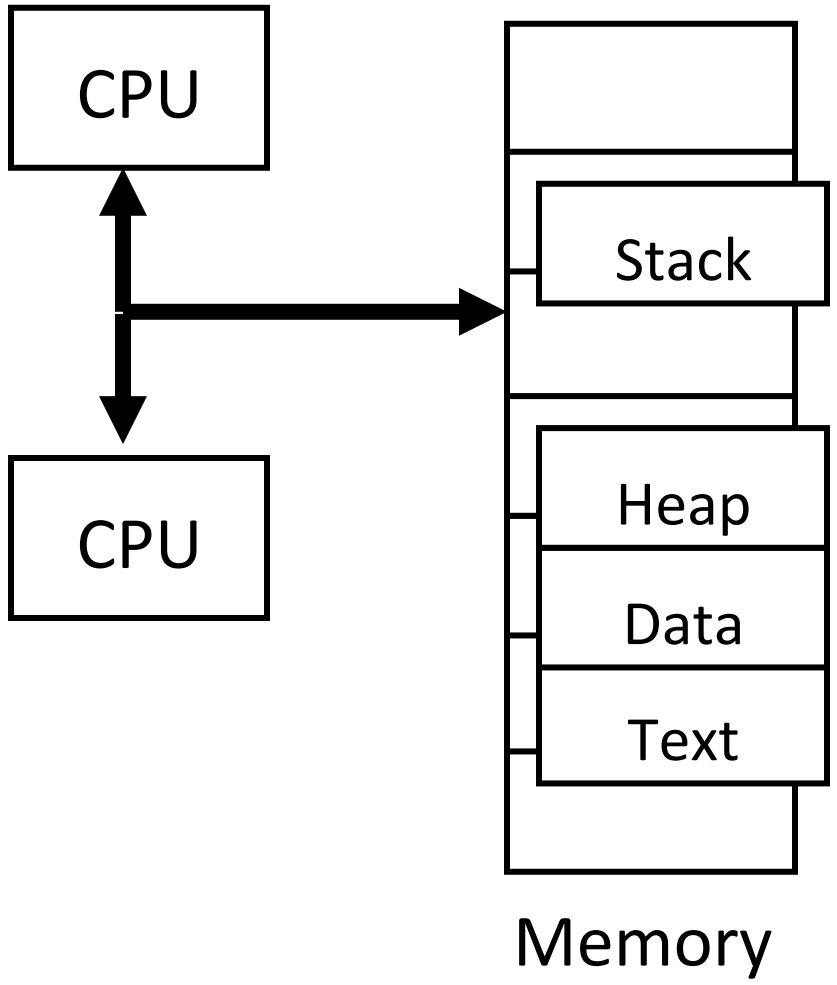
Many cores per processor (multi-core)
or many processors (multi-processor)

- Multiple programs run

Multiple Processes

Q: What happens when another program is executed concurrently on another processor?

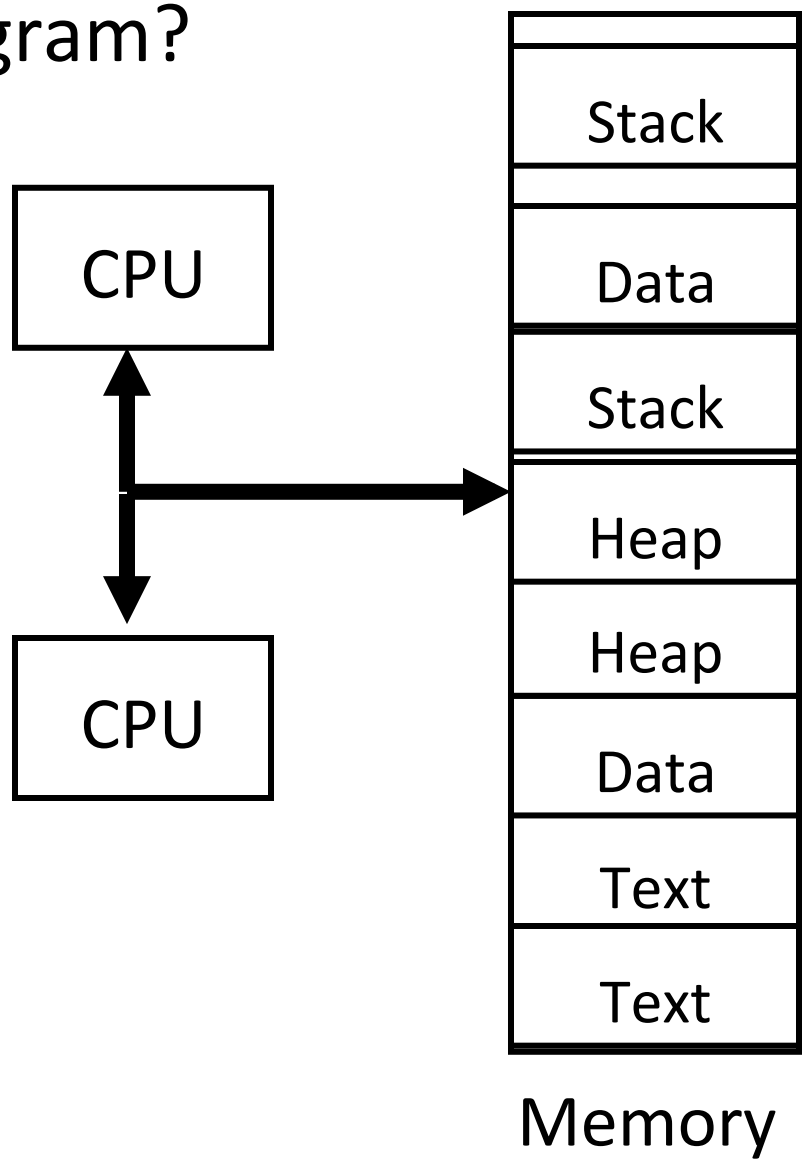
- Take turns using memory?



Solution? Multiple processes/processors

Can we relocate second program?

- What if they don't fit?
- What if not contiguous?
- Need to recompile/relink?
- …



All problems in computer science can be solved by another level of indirection.

– David Wheeler

– or, Butler Lampson

– or, Leslie Lamport

– or, Steve Bellovin

Virtual Memory

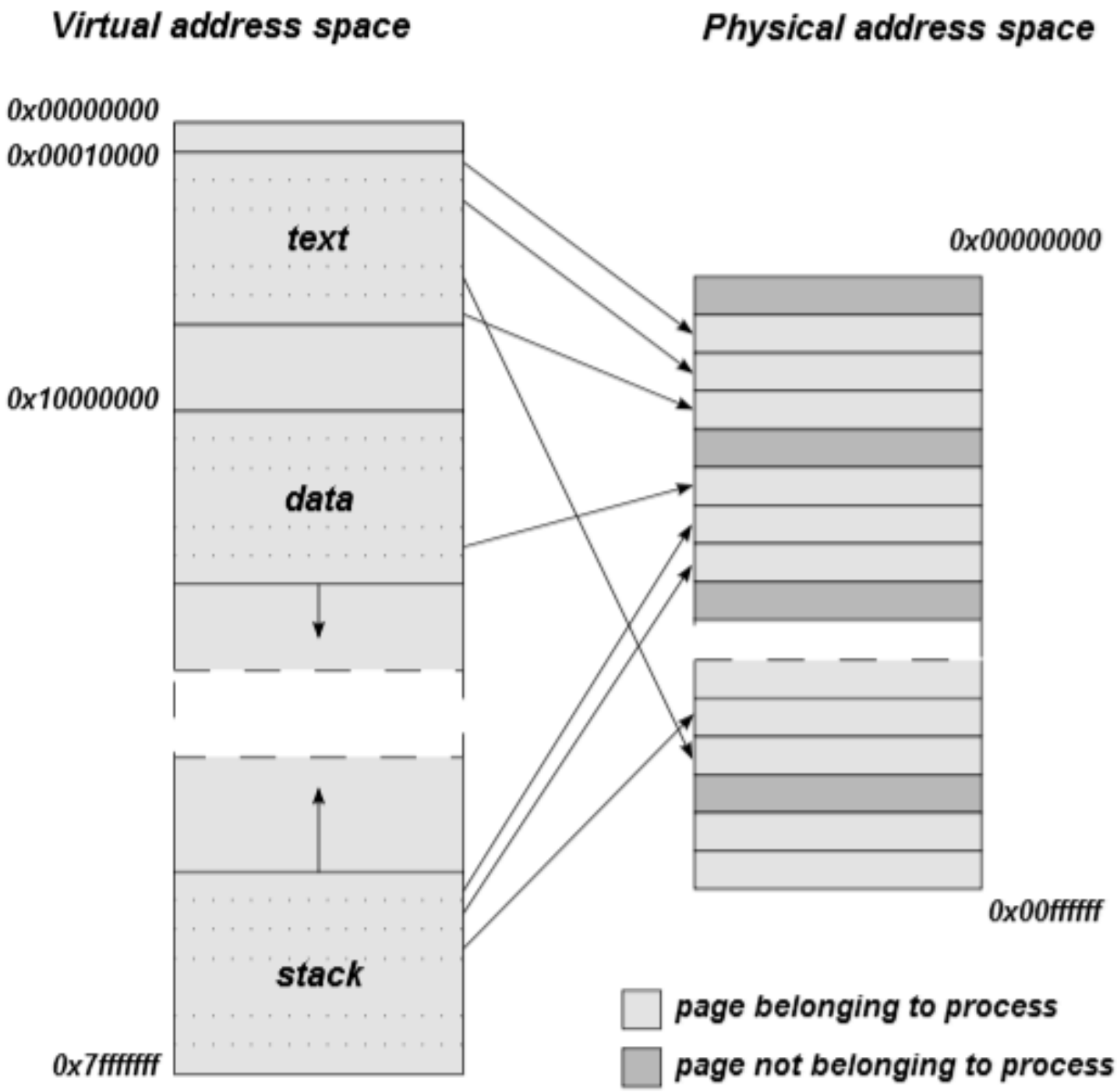
Virtual Memory: A Solution for All Problems

Each process has its own virtual address space

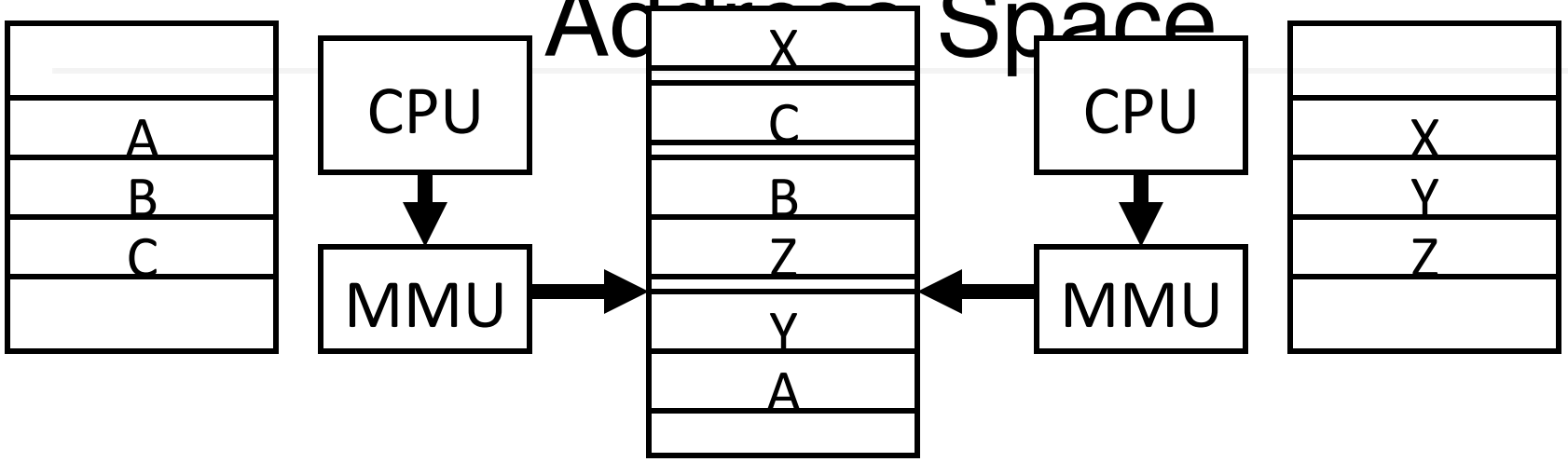
- Programmer can code as if they own all of memory

On-the-fly at runtime, for each memory access

- all access is through a virtual address
- translate fake virtual address to a real physical address
- redirect load/store to the physical address


Virtual address space
Physical address space
0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff
0x00000000
0x00ffffff
page belonging to process
page not belonging to process

wikipedia



Programs load/store to virtual addresses

Actual memory uses physical addresses

Memory Management Unit (MMU)

- Responsible for translating on the fly
- Essentially, just a big array of integers:

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

Virtual Memory Advantages

Advantages

Easy relocation

- Loader puts code anywhere in physical memory
- Creates virtual mappings to give illusion of correct layout

Higher memory utilization

- Provide illusion of contiguous memory
- Use all physical memory, even physical address 0x0

Easy sharing

- Different mappings for different programs / cores

And more to come…

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

Address Translation

Attempt #1: How does MMU translate addresses?

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

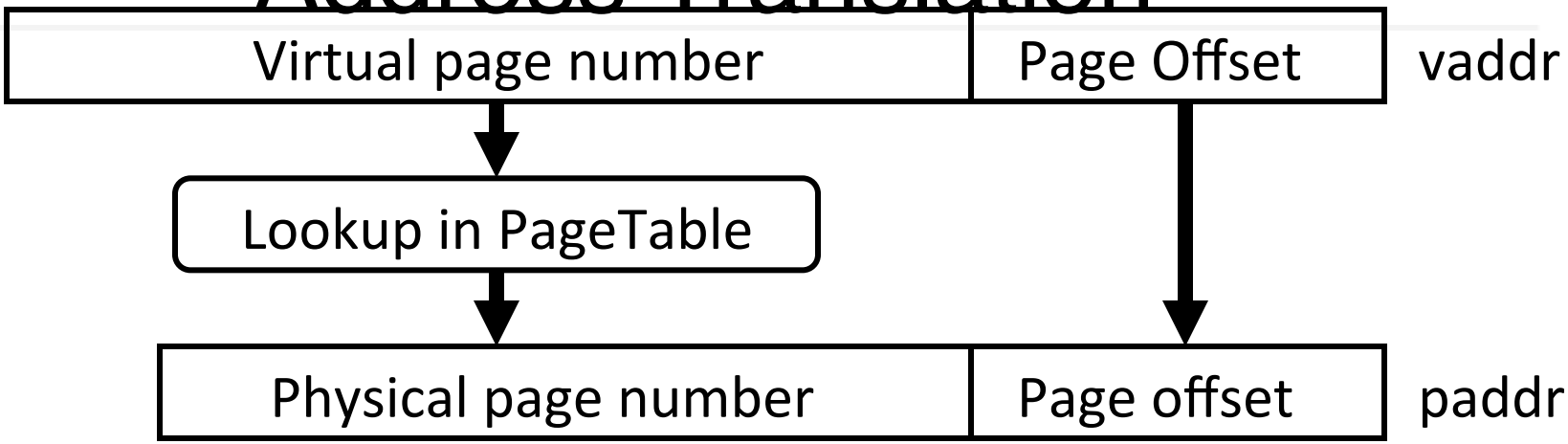
Granularity?

- Per word…
- Per block…
- Variable…

Typical:

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

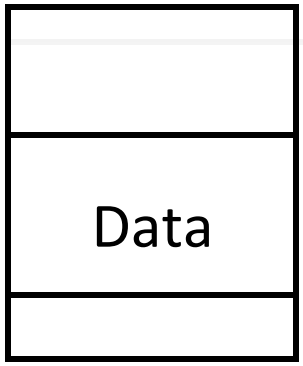
Address Translation



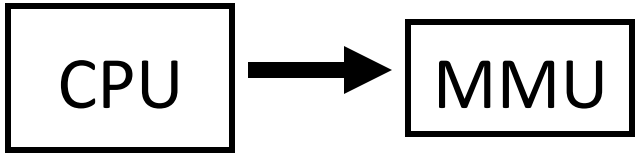
Attempt #1: For any access to virtual address:

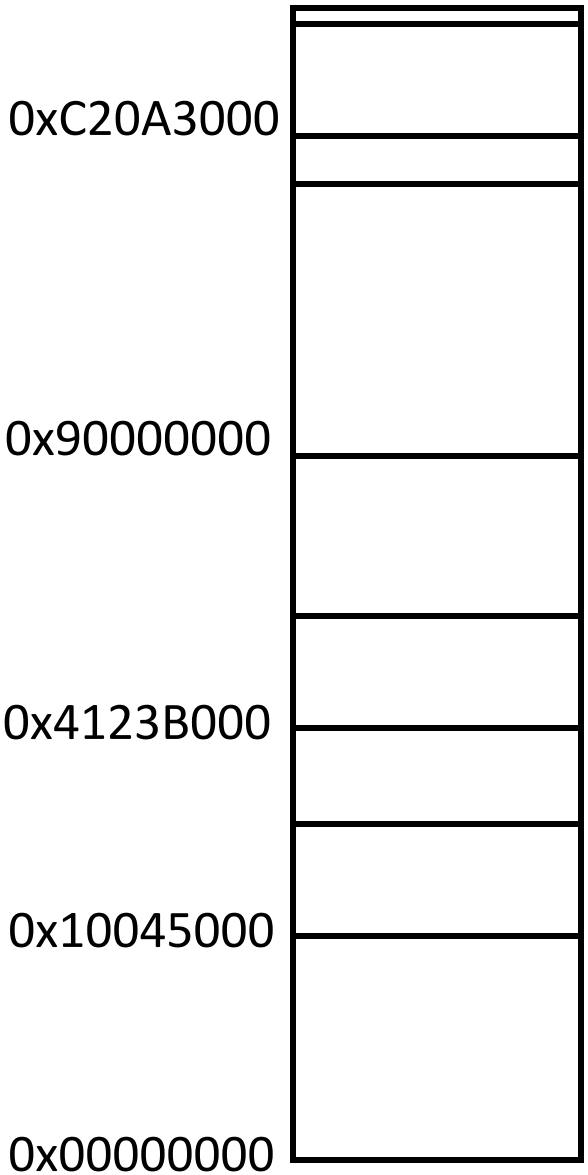
- Calculate virtual page number and page offset
- Lookup physical page number at PageTable[vpn]
- Calculate physical address as ppn:offset

Simple PageTable



Read Mem[0x00201538]





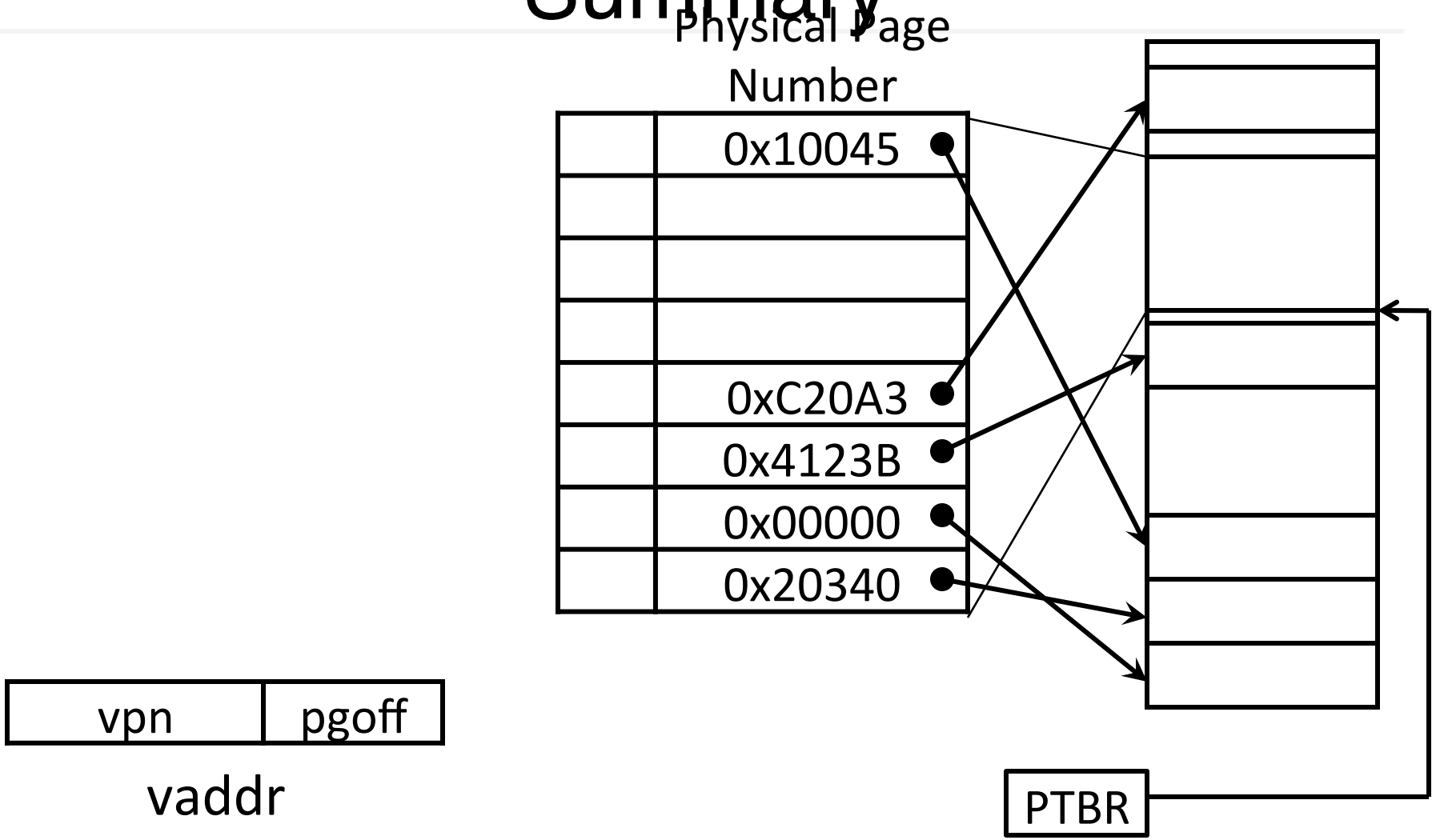
Q: Where to store page tables?

A: In memory, of course…

Special *page table base register*
(CR3:PTBR on x86)
(Cop0:ContextRegister on MIPS)

* lies to children

Summary



* lies to children

Page Size Example

Overhead for VM Attempt #1 (example)

Virtual address space (for each process):

- total memory: 2 bytes = 4GB
- page size: 2 bytes = 4KB
- entries in PageTable?
- size of PageTable?

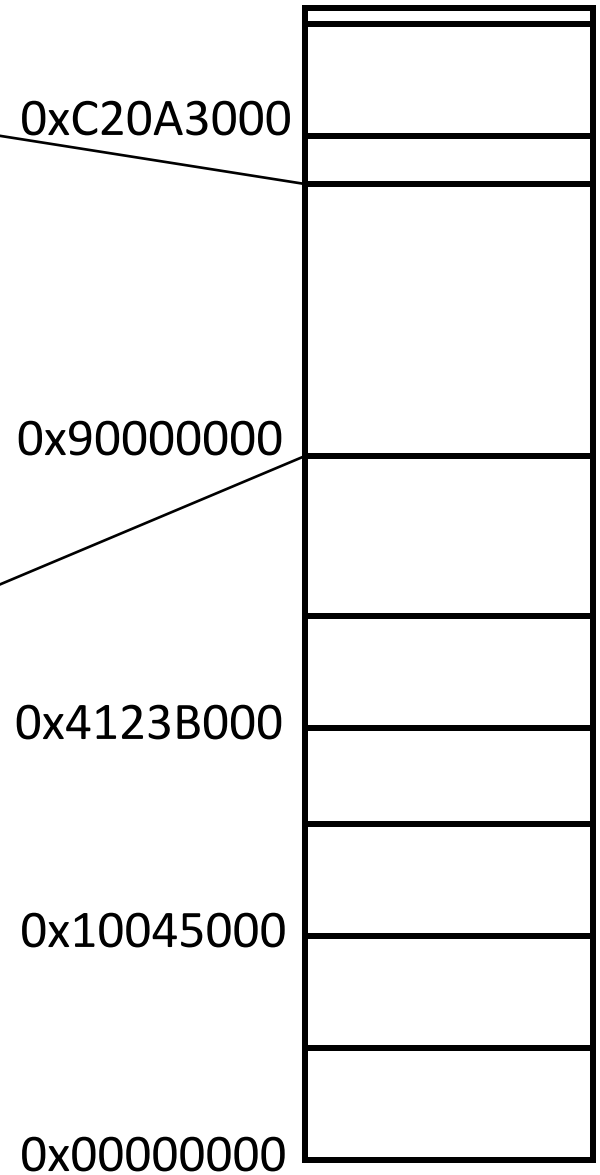
Physical address space:

- total memory: 2 bytes = 512MB
- overhead for 10 processes?

* lies to children

Invalid Pages

V		Physical Page Number
0		
1		0x10045
0		
0		
1		0xC20A3
1		0x4123B
1		0x00000
0		



Cool Trick #1: Don't map all pages

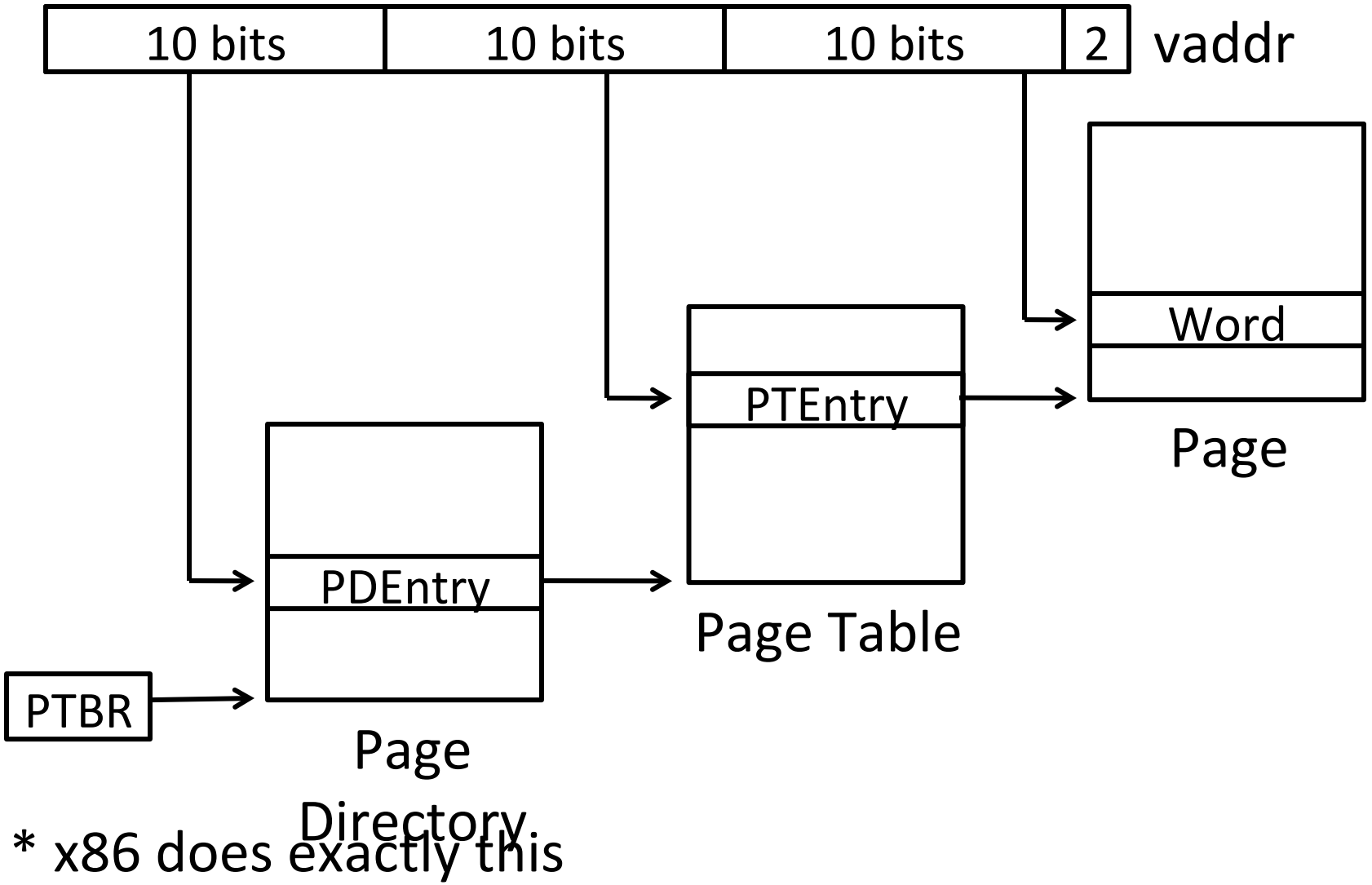
Need valid bit for each page table entry

Q: Why?

Beyond Flat Page Tables

Assume most of PageTable is empty

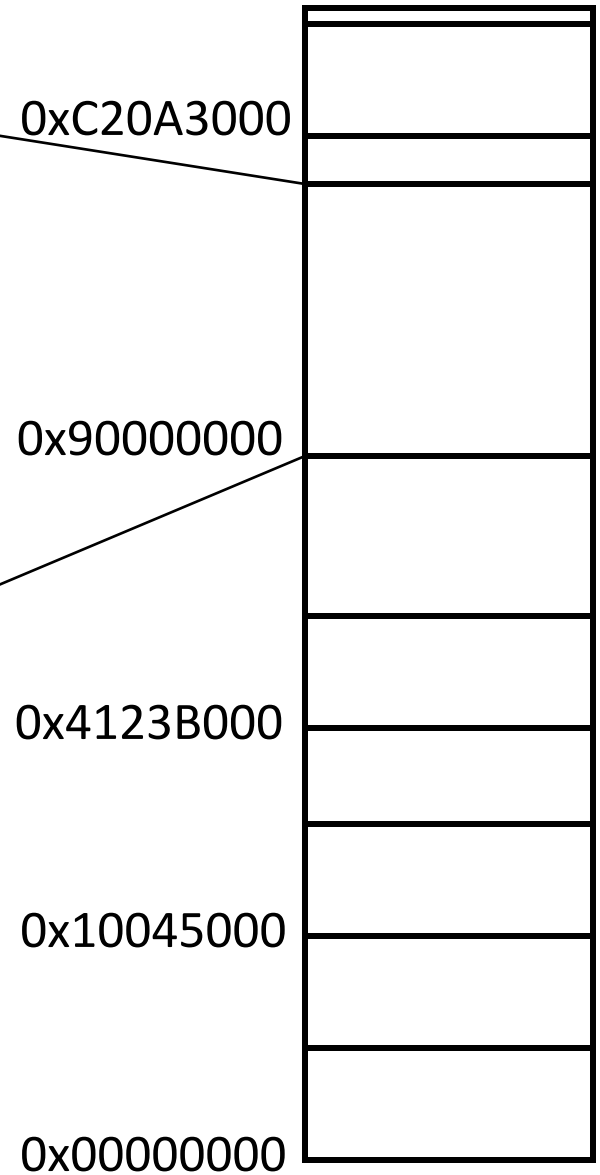
How to translate addresses? Multi-level PageTable



* x86 does exactly this

Page Permissions

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

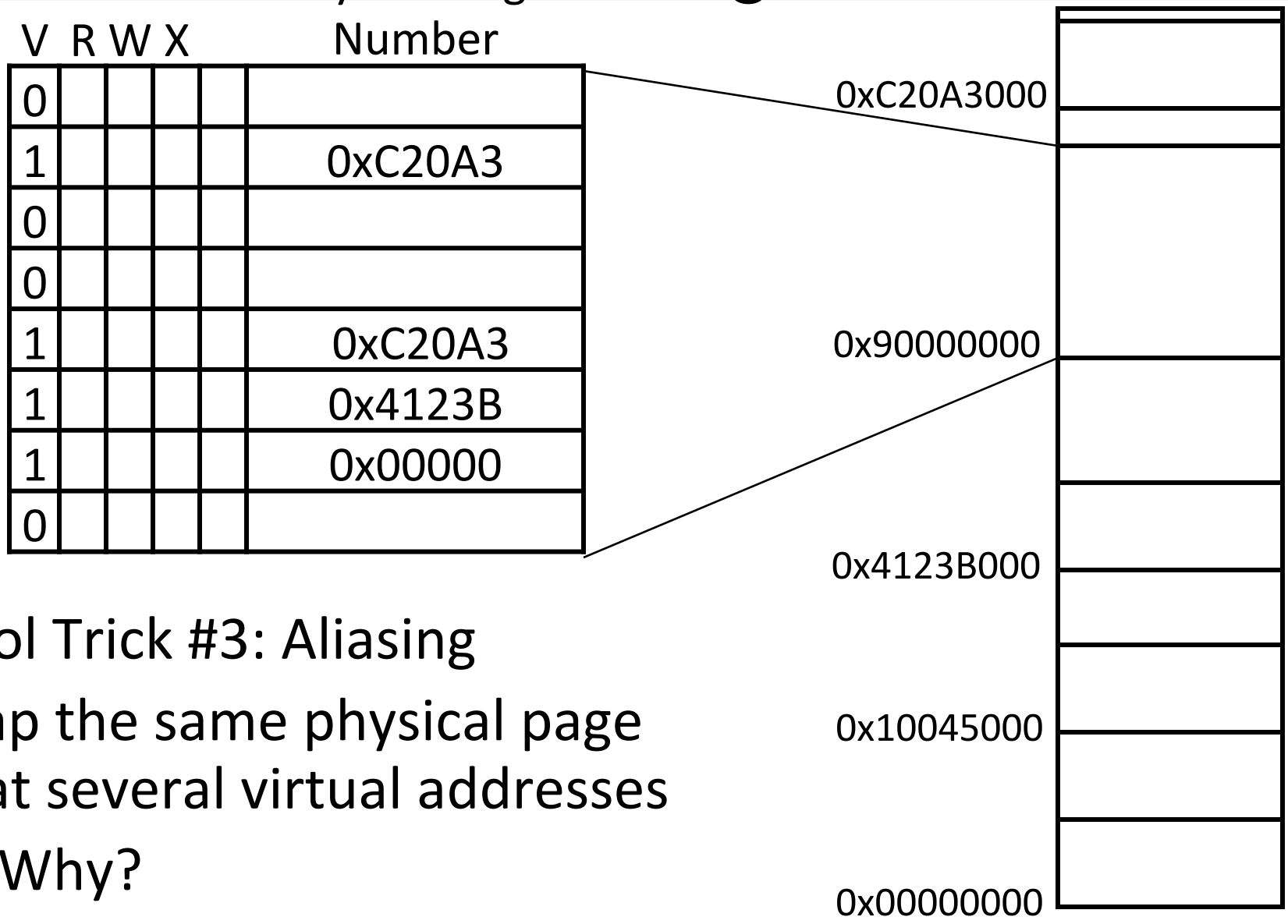


Cool Trick #2: Page permissions!

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

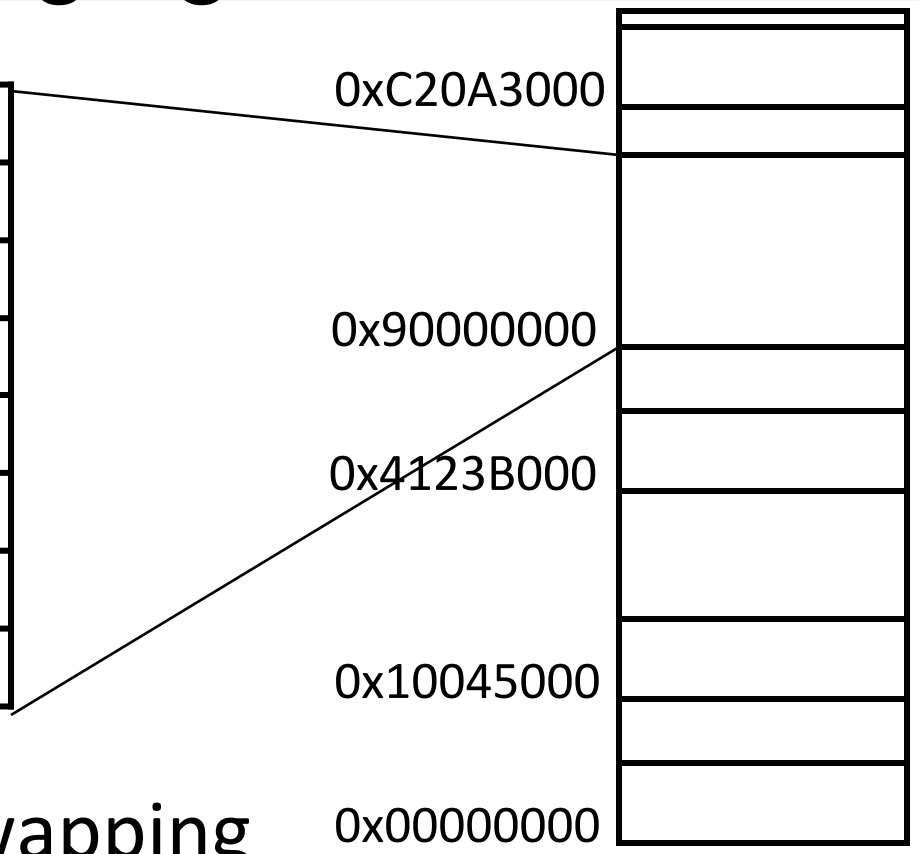
- Swap memory pages out to disk when not in use
- Page them back in when needed

Assumes Temporal/Spatial Locality

- Pages used recently most likely to be used again soon

Paging

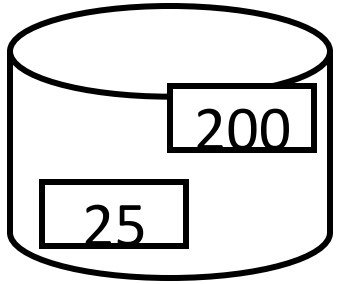
V	R	W	X	D	Physical Page Number
0					invalid
1				0	0x10045
0					invalid
0					invalid
0				0	disk sector 200
0				0	disk sector 25
1				1	0x00000
0					invalid



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

```
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

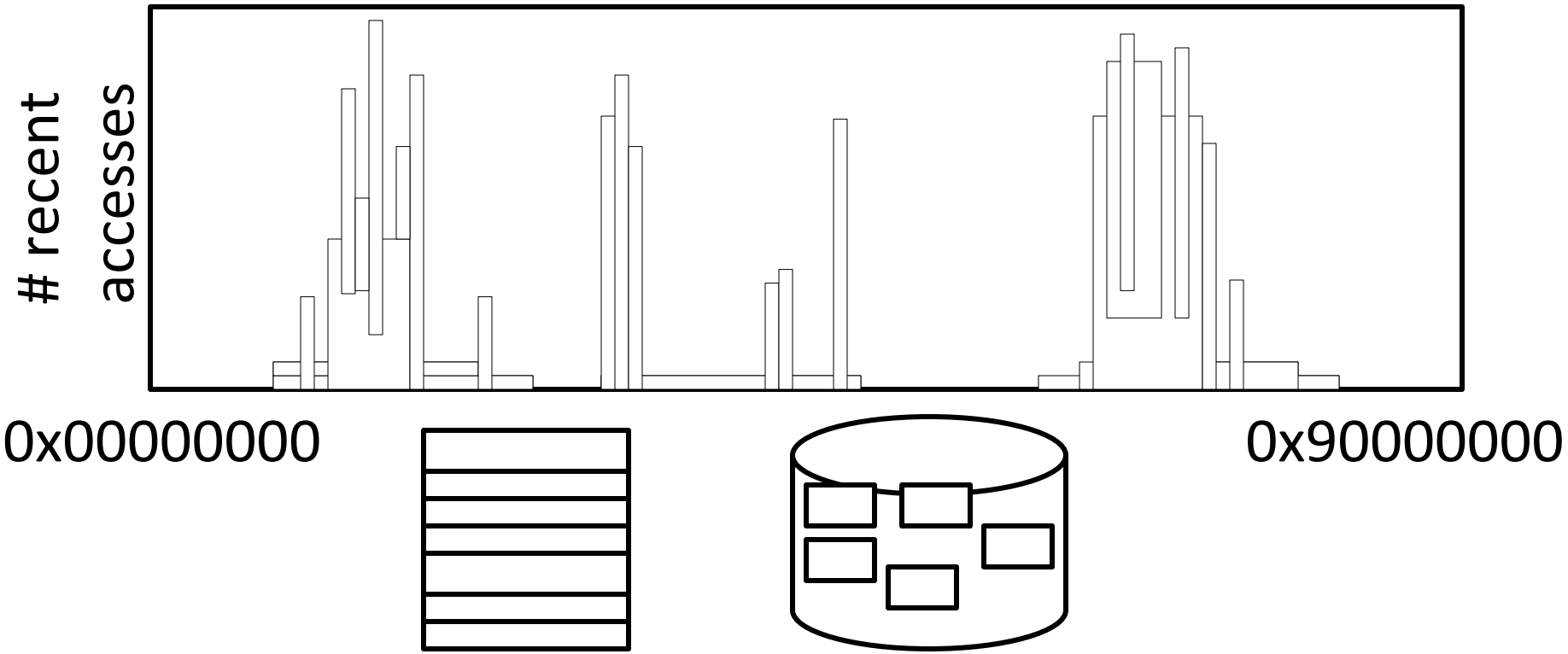
(5) Give the newly freed physical page to Skype

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

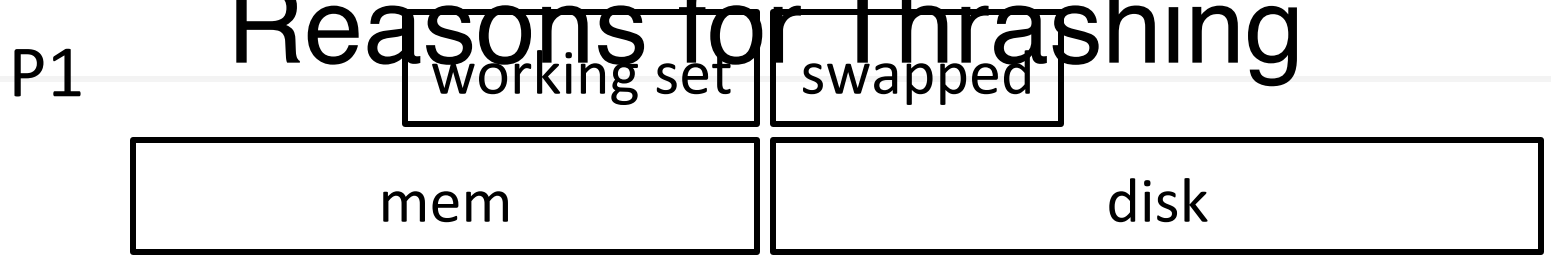
Paging Assumption 1

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

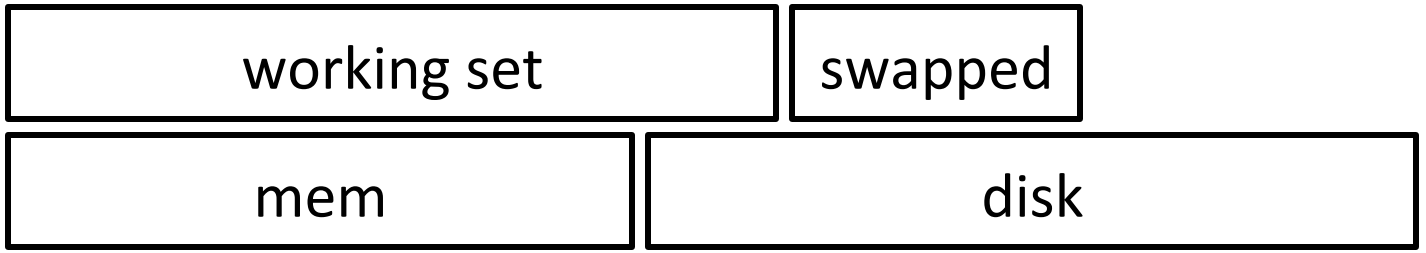


Reasons for Thrashing

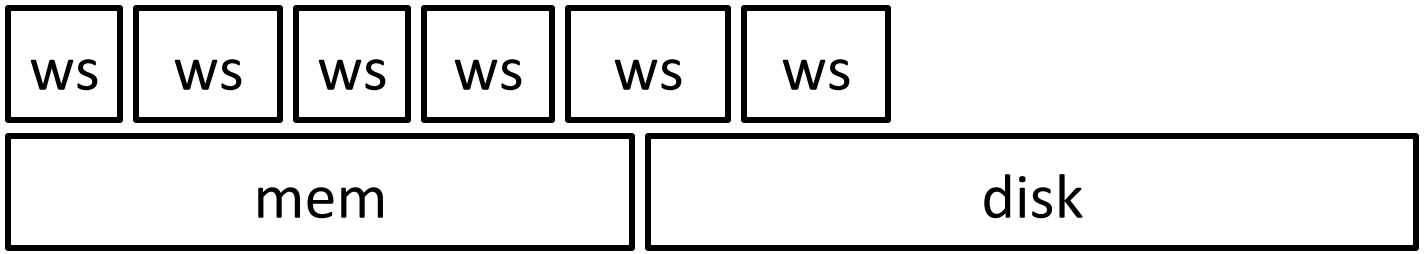


Q: What if working set is too large?

Case 1: Single process using too many pages



Case 2: Too many processes



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

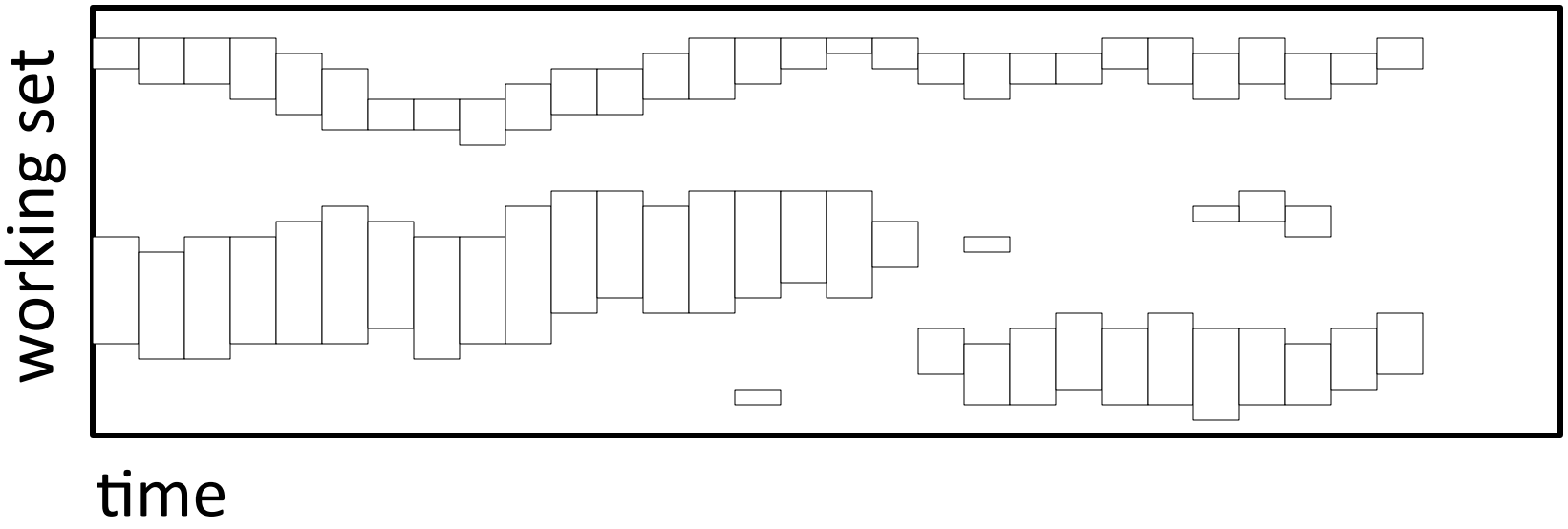
Ideal: Size of disk, speed of memory (or cache)

Non-ideal: Speed of disk

Paging Assumption 2

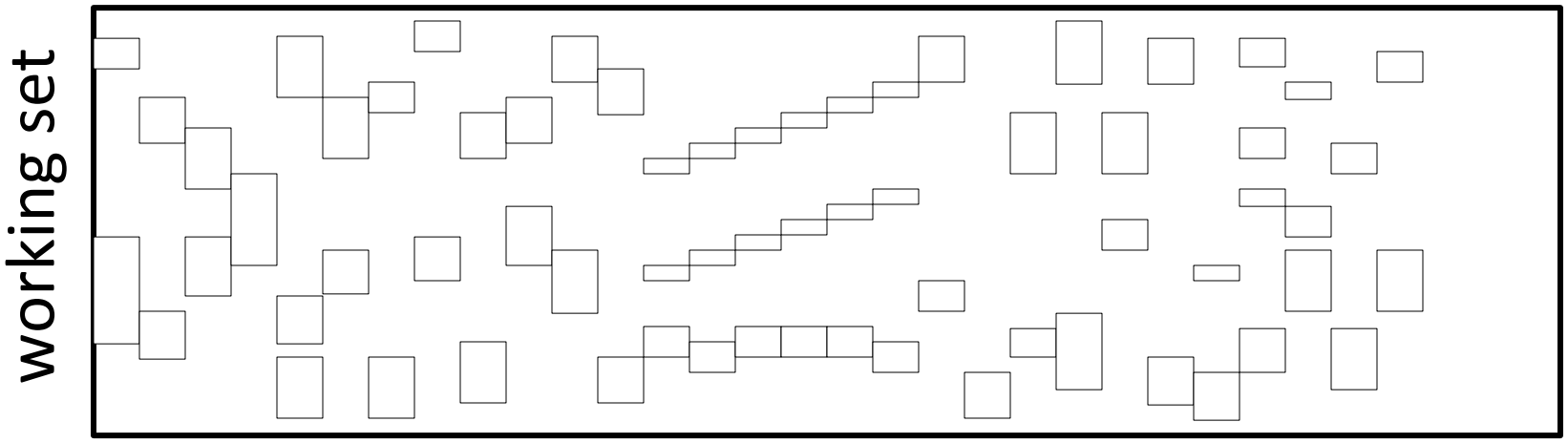
OS multiplexes physical memory among processes

- 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.