Virtual Memory

Prof. Kavita Bala and Prof. Hakim Weatherspoon

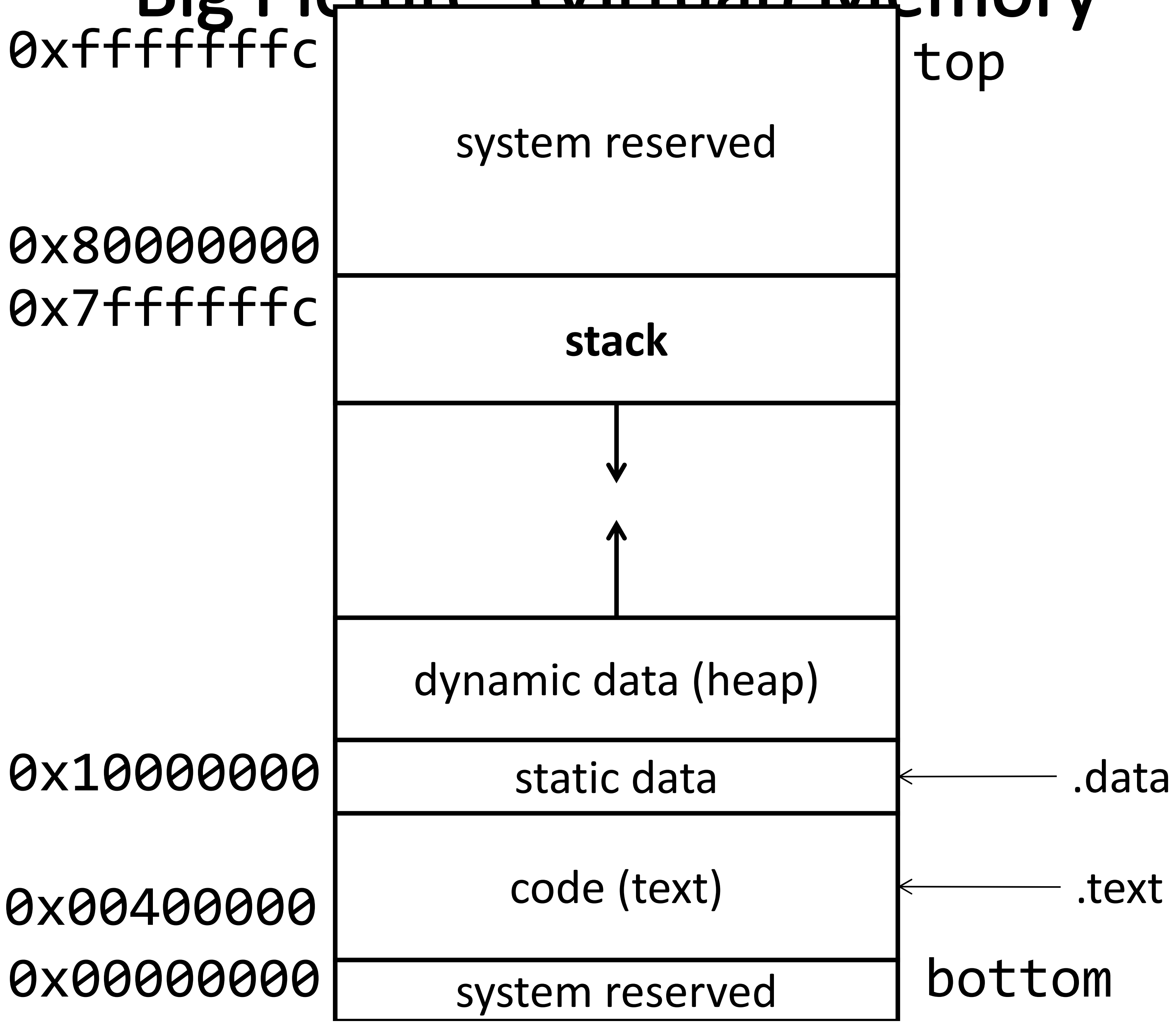
CS 3410, Spring 2014

Computer Science

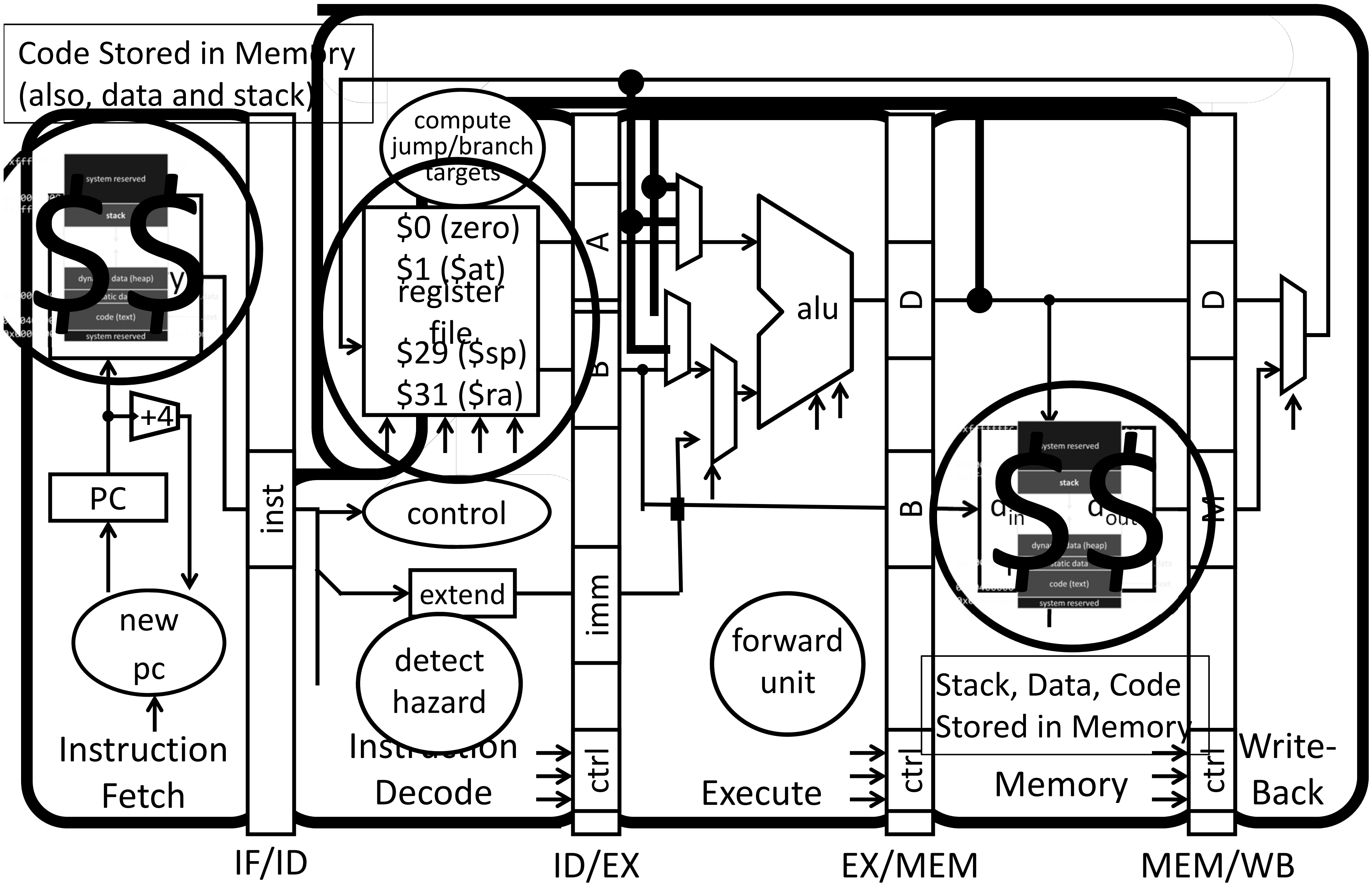
Cornell University

P & H Chapter 5.7 (up to TLBs)

Big Picture: (Virtual) Memory



Big Picture: (Virtual) Memory



Big Picture: (Virtual) Memory

How do we execute ***more than one*** program at a time?

Big Picture: (Virtual) Memory

How do we execute ***more than one*** program at a time?

A: Abstraction – Virtual Memory

- Memory that ***appears*** to exist as main memory (although most of it is supported by data held in secondary storage, transfer between the two being made automatically as required—i.e. ”paging”)
- Abstraction that supports multi-tasking---the ability to run more than one process at a time

Goals for Today: Virtual Memory

What is Virtual Memory?

How does Virtual memory Work?

- 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

Virtual Memory

Big Picture: Multiple Processes

How to Run multiple processes?

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

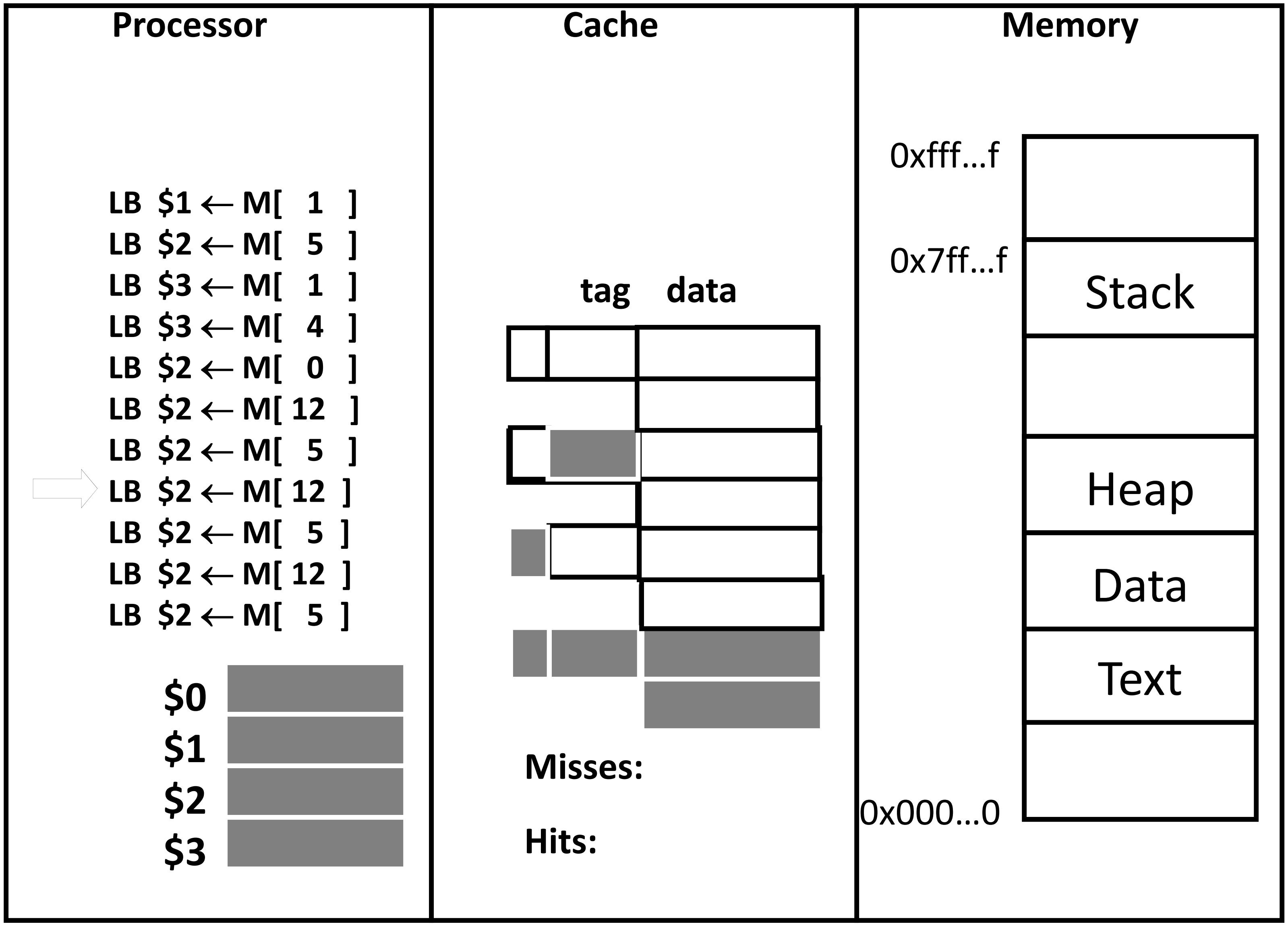
- Web browser, skype, office, … all must co-exist

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

- Multiple programs run *simultaneously*

Big Picture: (Virtual) Memory

Memory: big & slow vs Caches: small & fast



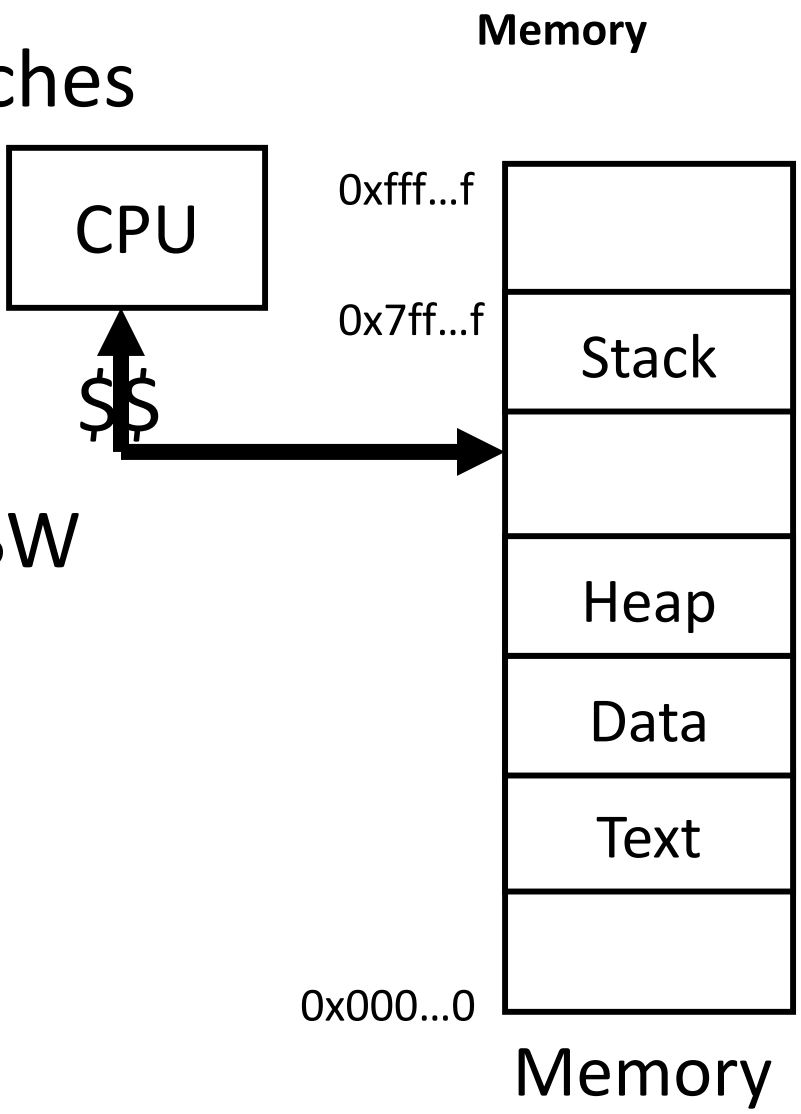
Processor & Memory

CPU address/data bus...

... routed through caches

... to main memory

- Simple, fast, but...



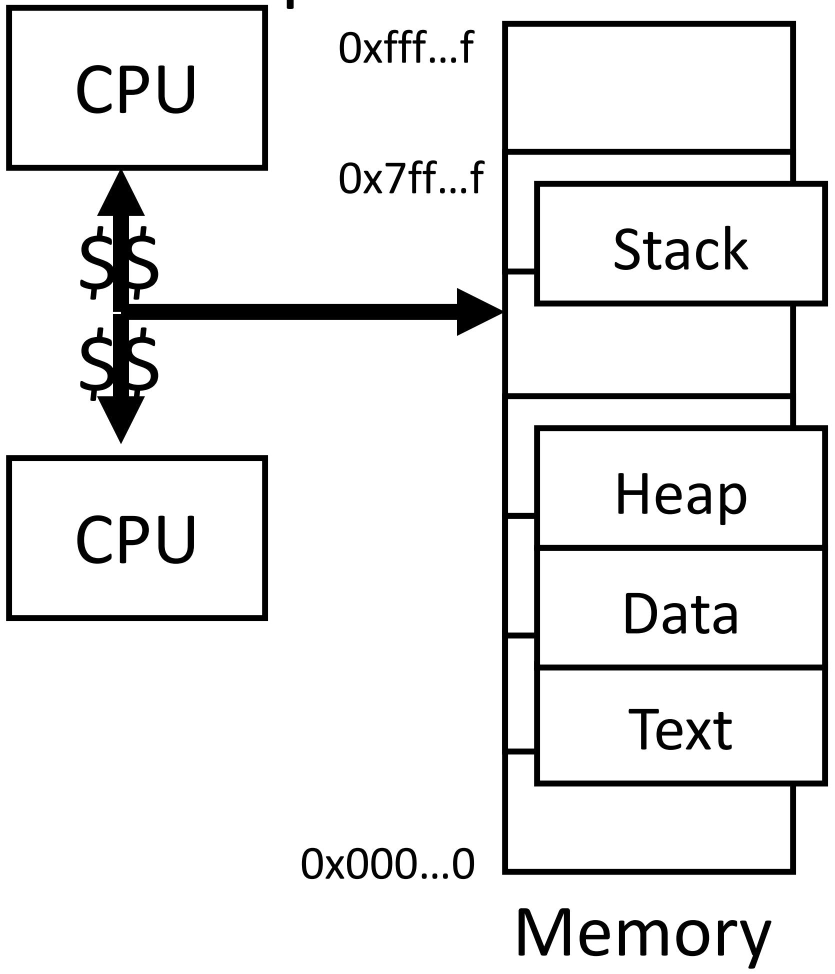
Q: What happens for LW/SW to an invalid location?

- 0x000000000 (NULL)
- uninitialized pointer

.

Multiple Processes

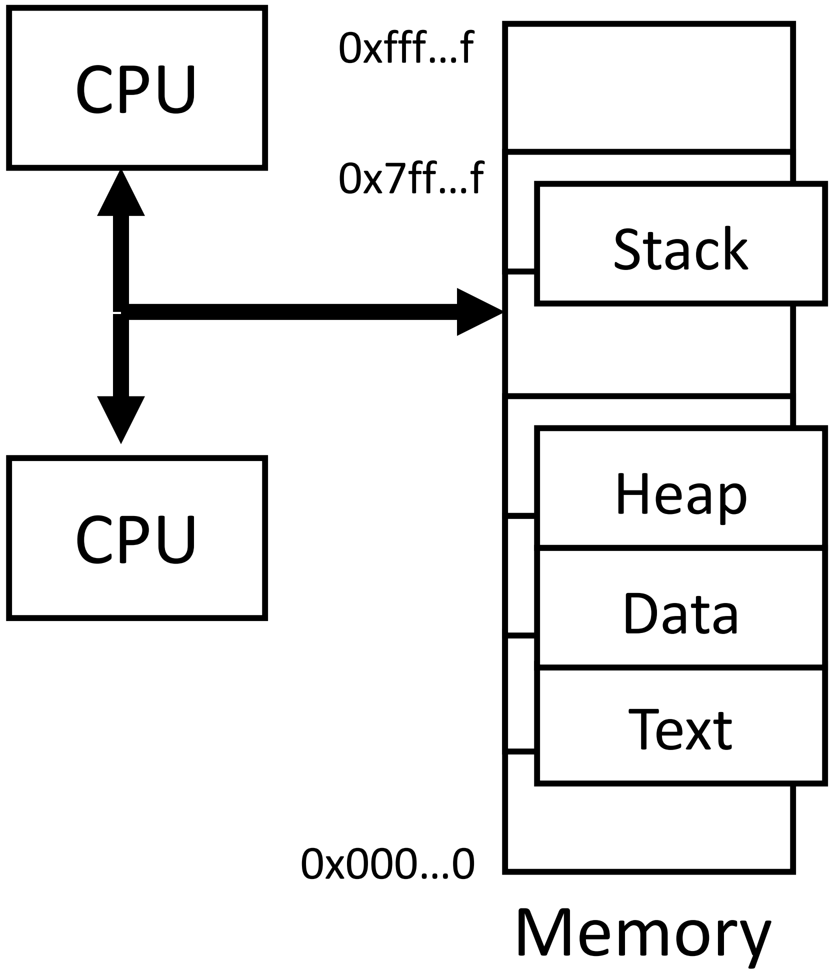
Q: What happens when another program is executed concurrently on another processor?



•

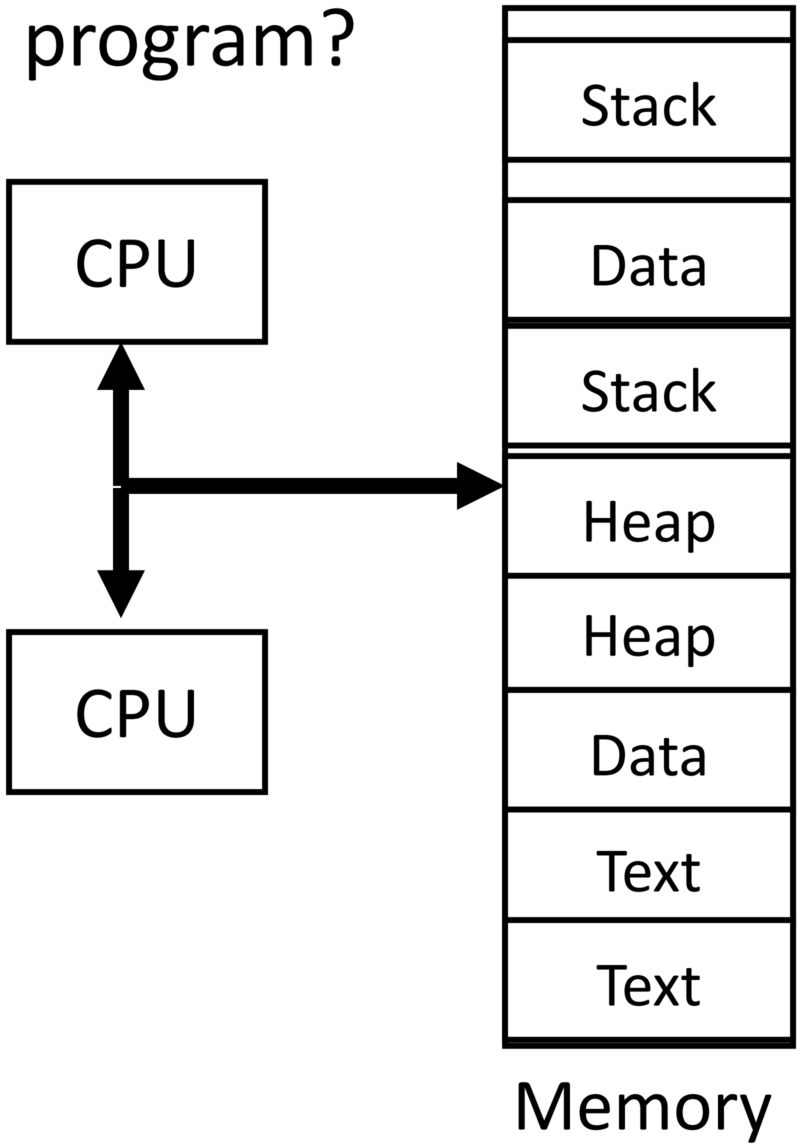
Multiple Processes

Q: Can we relocate second program?



Solution? Multiple processes/processors

Q: Can we relocate second program?



Takeaway

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

– *David Wheeler*

– *or, Butler Lampson*

– *or, Leslie Lamport*

– *or, Steve Bellovin*

Takeaway

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

– *David Wheeler*

– *or, Butler Lampson*

– *or, Leslie Lamport*

– *or, Steve Bellovin*

Solution: Need a ***MAP***
To map a ***Virtual Address (generated by CPU)***
to a ***Physical Address (in memory)***

Next Goal

How does Virtual Memory work?

i.e. How do we create that “map” that maps a virtual address generated by the CPU to a physical address used by main memory?

Virtual Memory

Virtual Memory: A Solution for All Problems

- Program/CPU can access any address from 0…2^N
  (e.g. N=32)

Each process has its own virtual address space

- A process is a program being executed
- Programmer can code as if they own all of memory

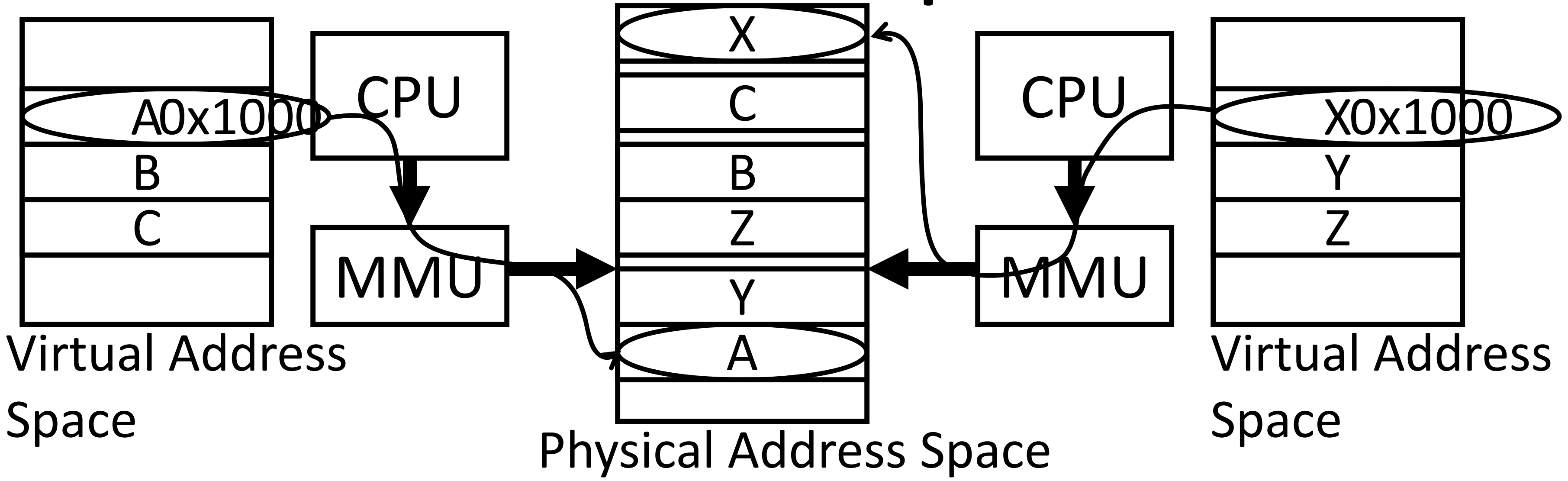
On-the-fly at runtime, for each memory access

- all access is *indirect* through a virtual address

map

- $\boxed{\text{translate fake virtual address to a real physical address}}$
- redirect load/store to the physical address

Address Space



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…

Takeaway

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

Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

Virtual memory is implemented via a “Map”, a ***PageTage,*** that maps a ***vaddr*** (a virtual address) to a ***paddr*** (physical address):

paddr = PageTable[vaddr]

Next Goal

How do we implement that translation from a virtual address (vaddr) to a physical address (paddr)?

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

i.e. How do we implement the PageTable??

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

Attempt#1: Address Translation

How large should a PageTable be for a MMU?

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

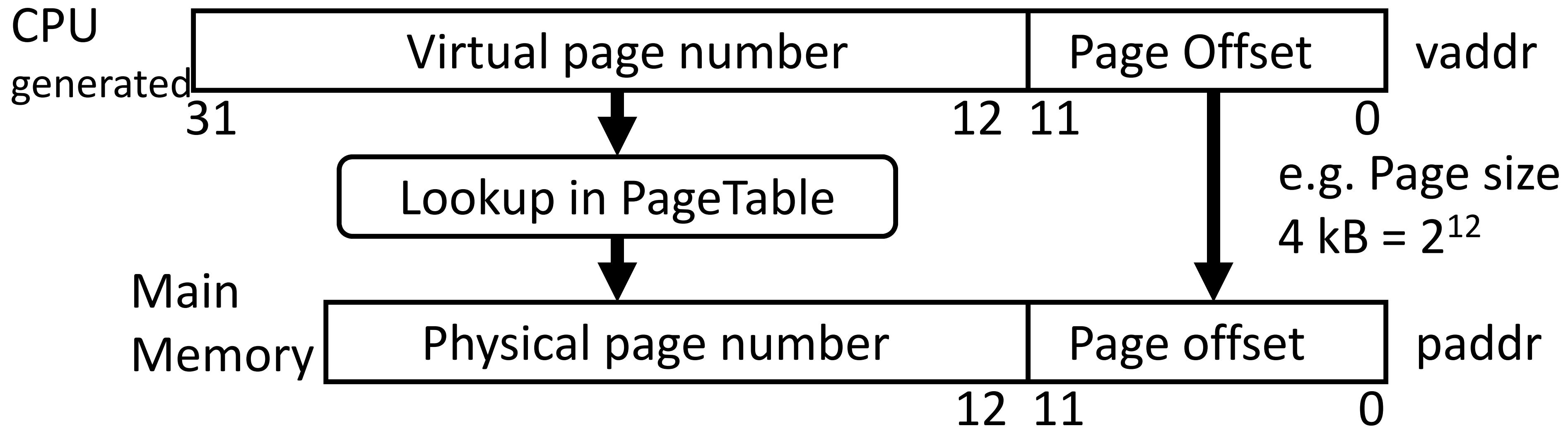
Granularity?

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

Typical:

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

Attempt #1: 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

Takeaway

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

Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

Virtual memory is implemented via a “Map”, a ***PageTage,*** that maps a ***vaddr*** (a virtual address) to a ***paddr*** (physical address):

paddr = PageTable[vaddr]

A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

Next Goal

Example

How to translate a vaddr (virtual address) generated by the CPU to a paddr (physical address) used by main memory using the PageTable managed by the memory management unit (MMU).

Next Goal

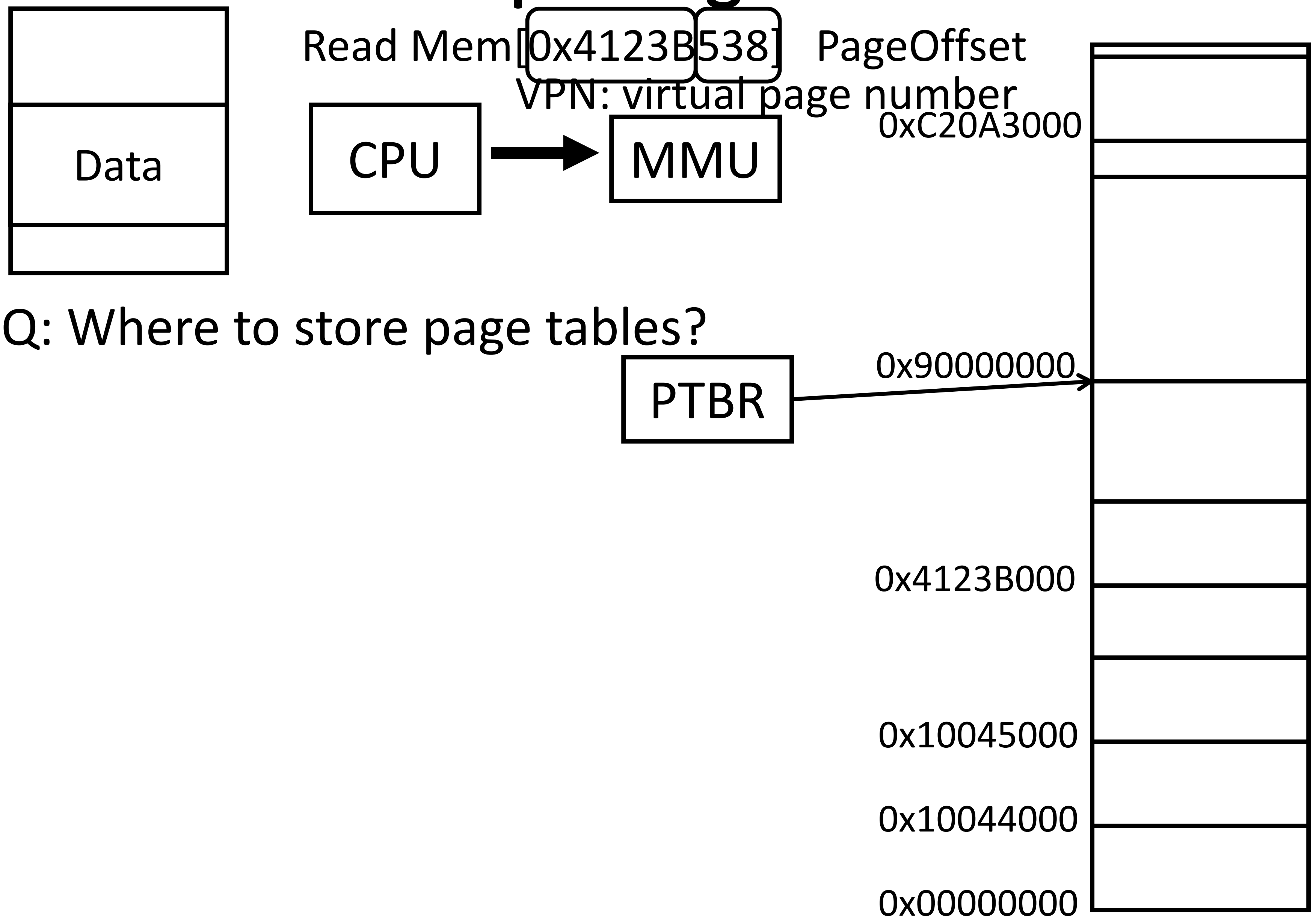
Example

How to translate a vaddr (virtual address) generated by the CPU to a paddr (physical address) used by main memory using the PageTable managed by the memory management unit (MMU).

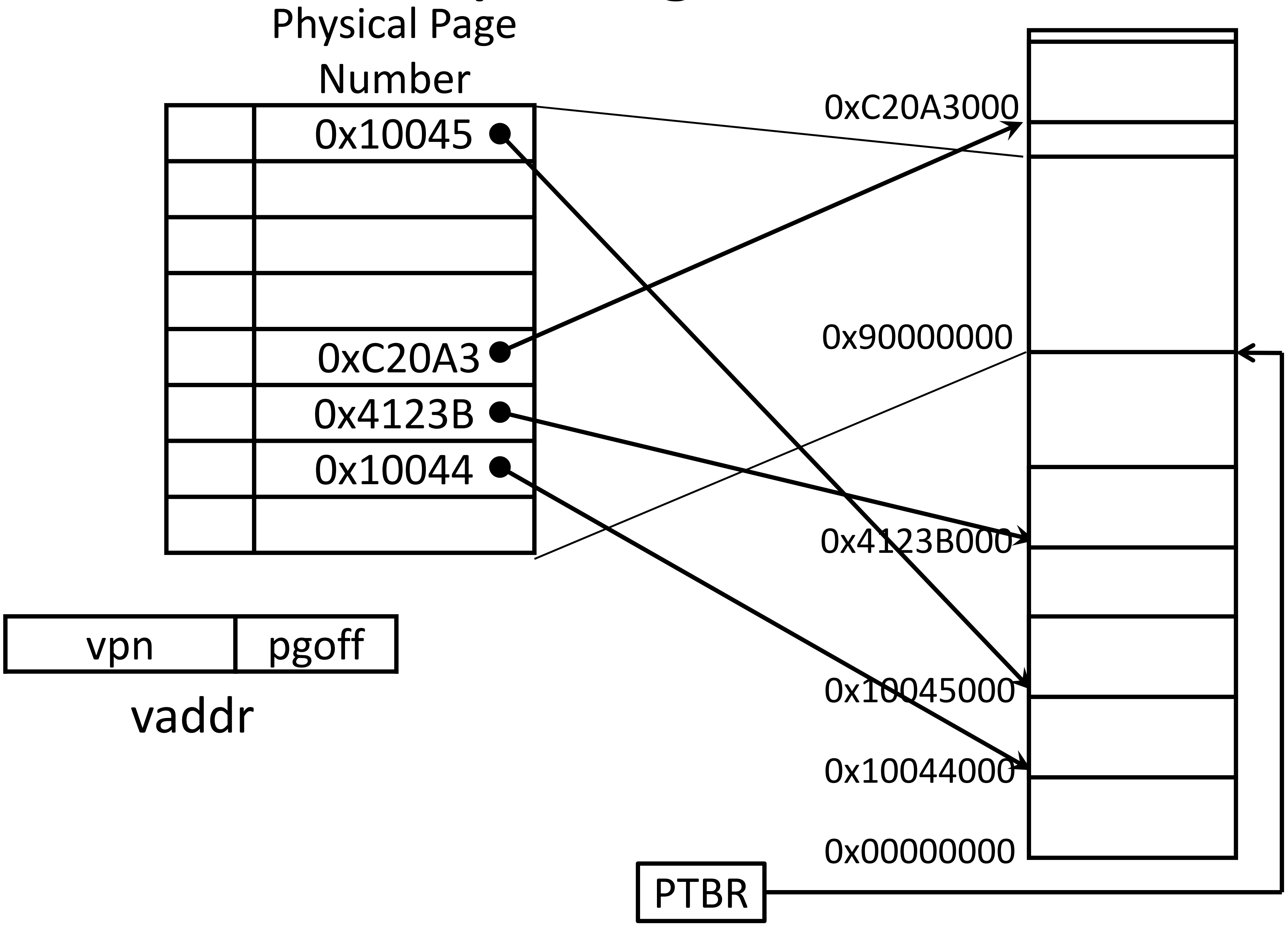
Q: Where is the PageTable stored??

Simple PageTable

Q: Where to store page tables?

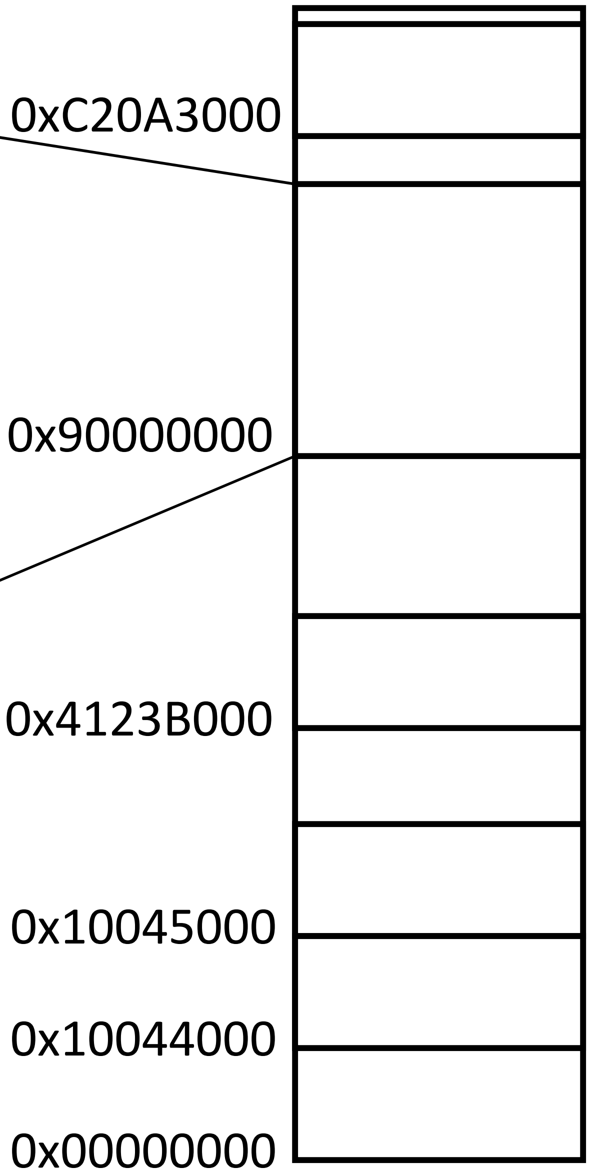


Simple PageTable



Invalid Pages

V		Physical Page Number
0		
1		0x10045
0		
0		
1		0xC20A3
1		0x4123B
1		0x10044
0		



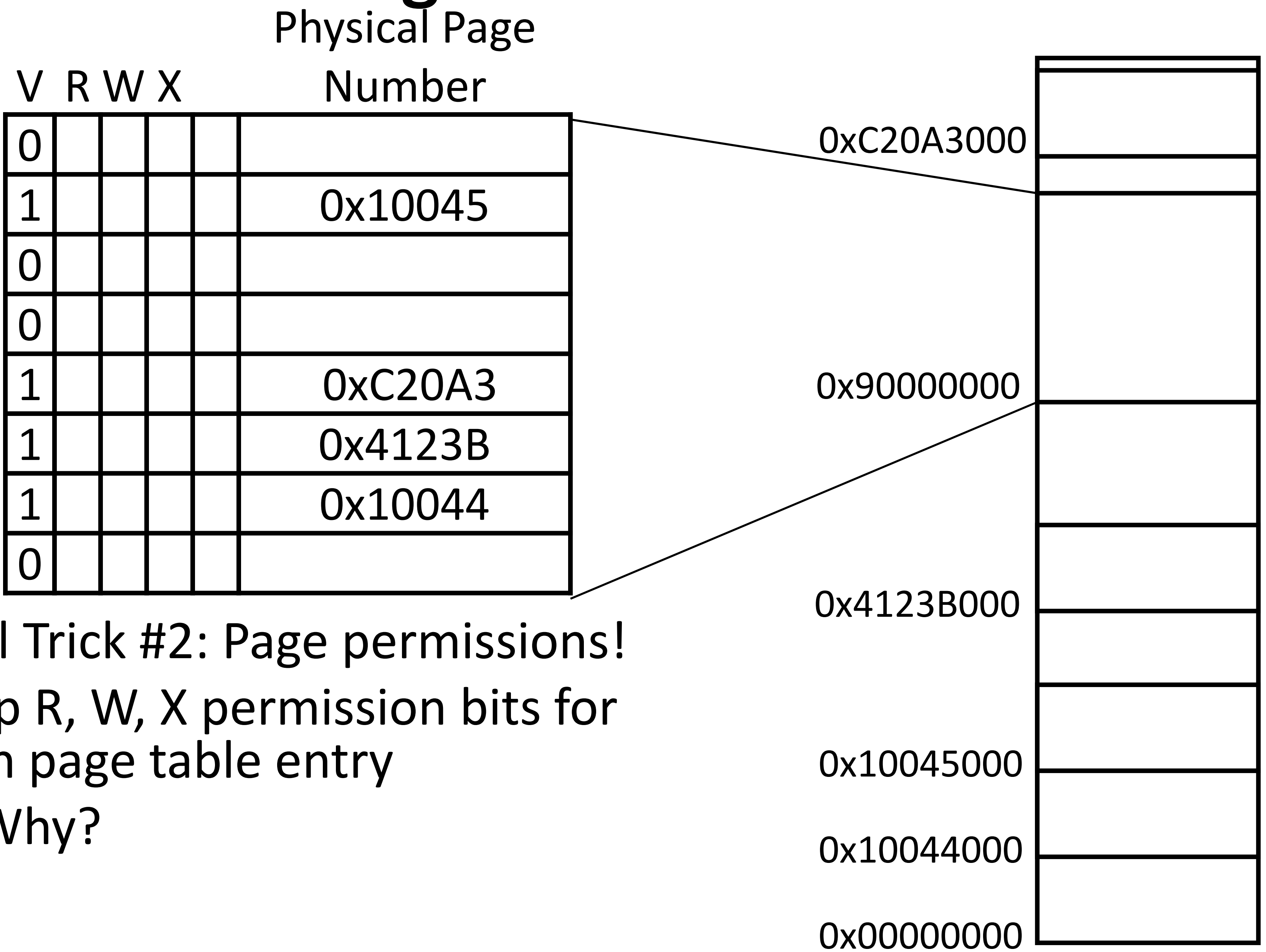
Cool Trick #1: Don’t map all pages

Need valid bit for each page table entry

Q: Why?

.

Page Permissions



V	R	W	X		Physical Page Number
0					
1					0x10045
0					
0					
1					0xC20A3
1					0x4123B
1					0x10044
0					

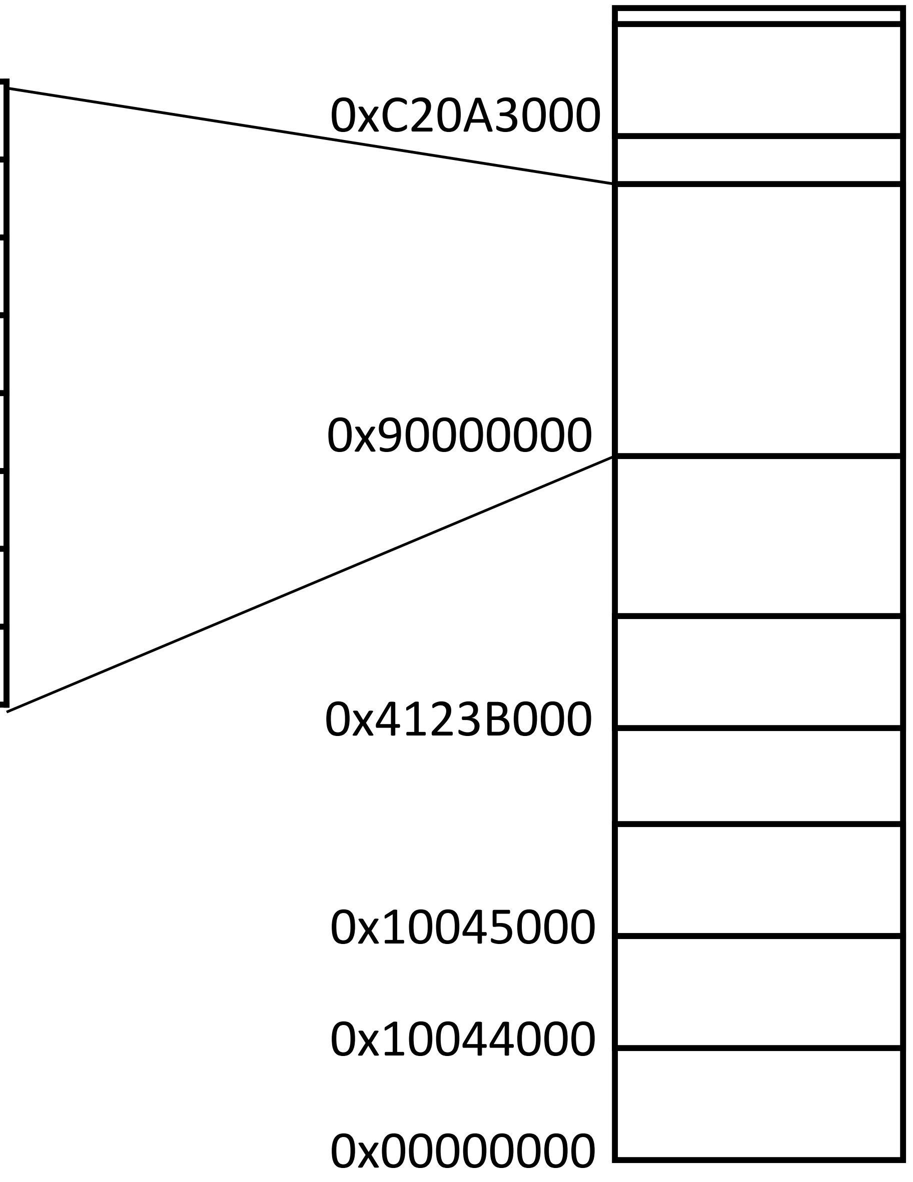
Cool Trick #2: Page permissions!

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

Q: Why?

Aliasing

V	R	W	X		Physical Page Number
0					
1					0xC20A3
0					
0					
1					0xC20A3
1					0x4123B
1					0x10044
0					



Cool Trick #3: Aliasing

Map the same physical page at several virtual addresses

Q: Why?

.

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?

Takeaway

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

Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

Virtual memory is implemented via a “Map”, a ***PageTage,*** that maps a ***vaddr*** (a virtual address) to a ***paddr*** (physical address):

paddr = PageTable[vaddr]

A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits.

But, overhead due to PageTable is significant.

Next Goal

How do we reduce the size (overhead) of the PageTable?

Next Goal

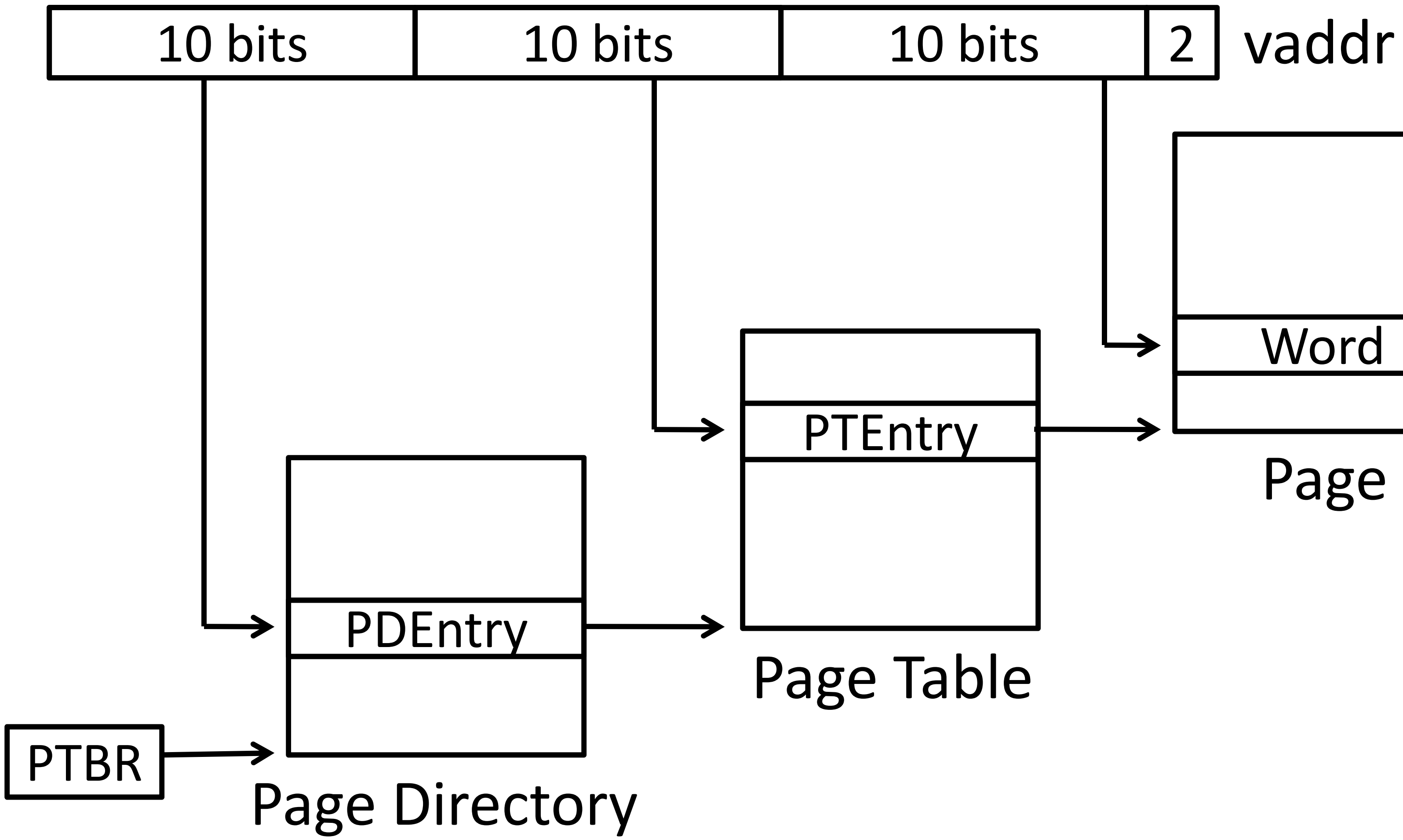
How do we reduce the size (overhead) of the PageTable?

A: Another level of indirection!!

Beyond Flat Page Tables

Assume most of PageTable is empty

How to translate addresses? Multi-level PageTable



* x86 does exactly this

Beyond Flat Page Tables

Assume most of PageTable is empty

How to translate addresses? Multi-level PageTable

Q: Benefits?

Q: Drawbacks

Takeaway

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

Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

Virtual memory is implemented via a “Map”, a ***PageTage,*** that maps a ***vaddr*** (a virtual address) to a ***paddr*** (physical address):

paddr = PageTable[vaddr]

A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits.

But, overhead due to PageTable is significant.

Another level of indirection, two levels of PageTables and significantly reduce the overhead due to PageTables.

Next Goal

Can we run process larger than physical memory?

Paging

Paging

Can we run process larger than physical memory?

- The “virtual” in “virtual memory”

View memory as a “cache” for secondary storage

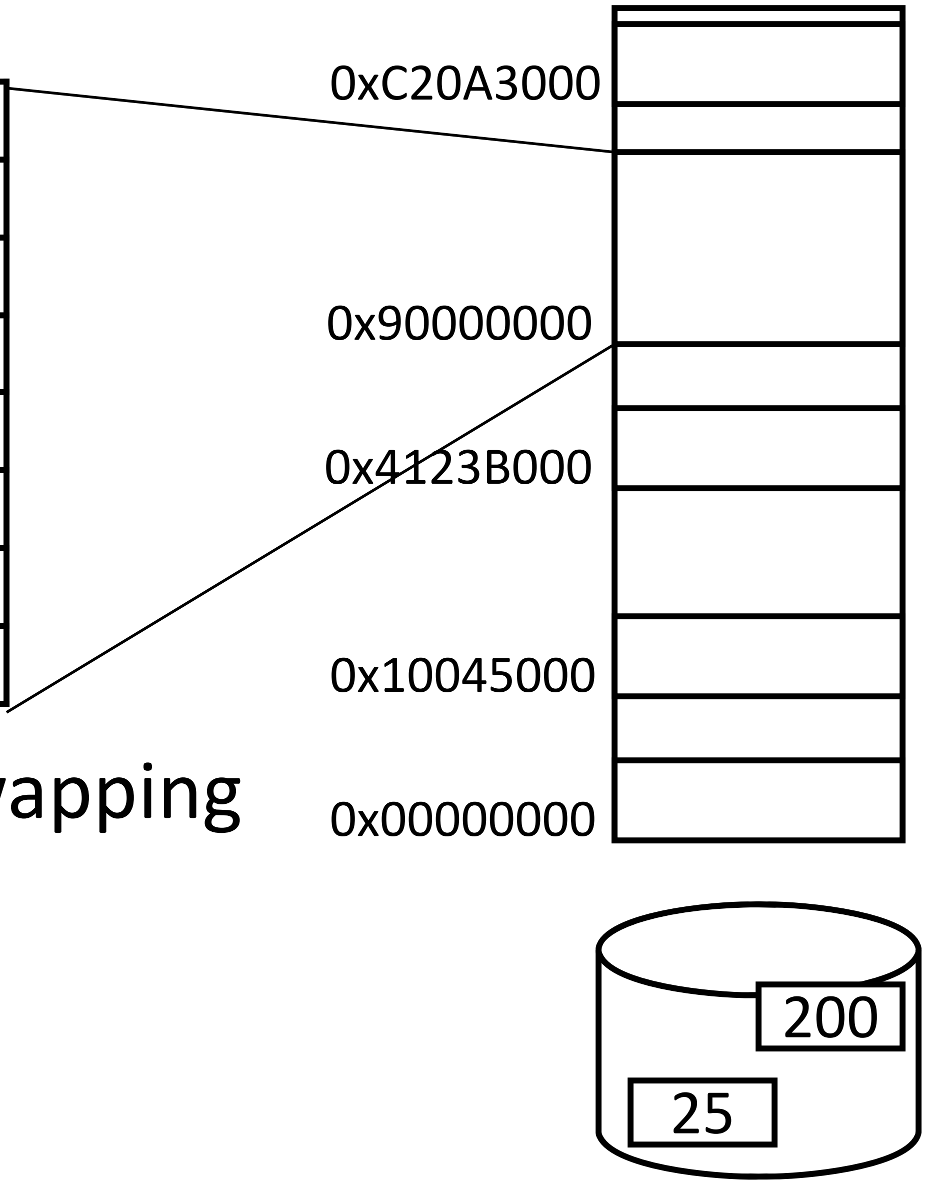
- Swap memory pages out to disk when not in use
- Page them back in when needed

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, …

Summary

Virtual Memory

- Address Translation
  - Pages, page tables, and memory mgmt unit
- Paging

Next time

- 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

Administrivia

Lab3 is out due next Wednesday

Administrivia

Next five weeks

- Week 10 (Apr 8): Lab3 released
- Week 11 (Apr 15): Proj3 release, Lab3 due Wed, HW2 due Fri
- Week 12 (Apr 22): Lab4 release and Proj3 due Fri
- Week 13 (Apr 29): Proj4 release, Lab4 due Tue, Prelim2
- Week 14 (May 6): Proj3 tournament, Proj4 design doc due

Final Project for class

- Week 15 (May 13): Proj4 due