Prelim 3 Review

Hakim Weatherspoon
CS 3410, Spring 2012
Computer Science
Cornell University

Administrivia

Pizza party: PA3 Games Night
- **Tomorrow**, Friday, April 27th, 5:00-7:00pm
- Location: Upson B17

Prelim 3
- **Tonight**, Thursday, April 26th, 7:30pm
- Location: Olin 155

PA4: Final project out next week
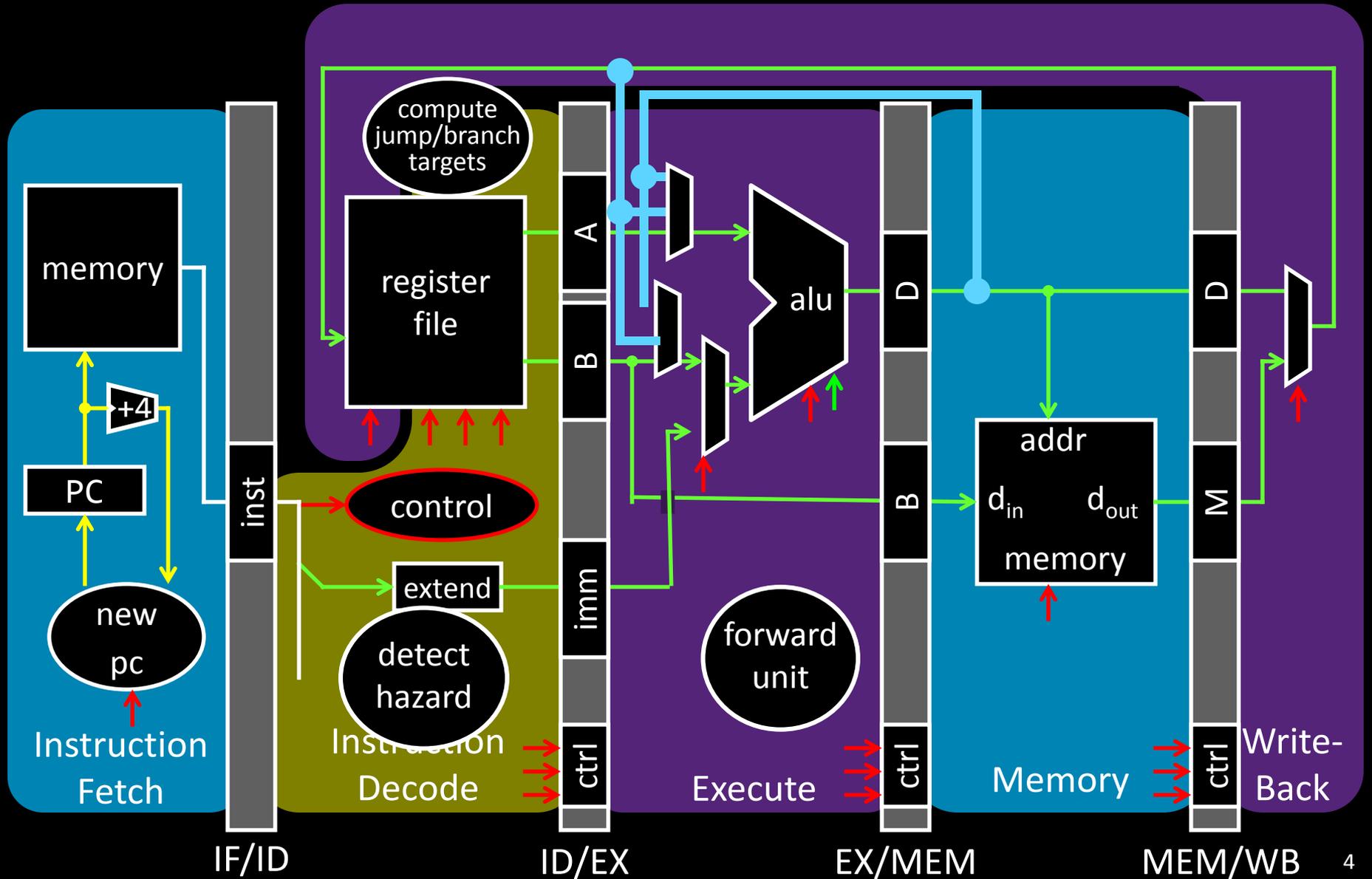- **Demos**: May 14-16
- ***Will not be able to use slip days***

Goals for Today

Prelim 3 review

- Caching,

- Virtual Memory, Paging, TLBs

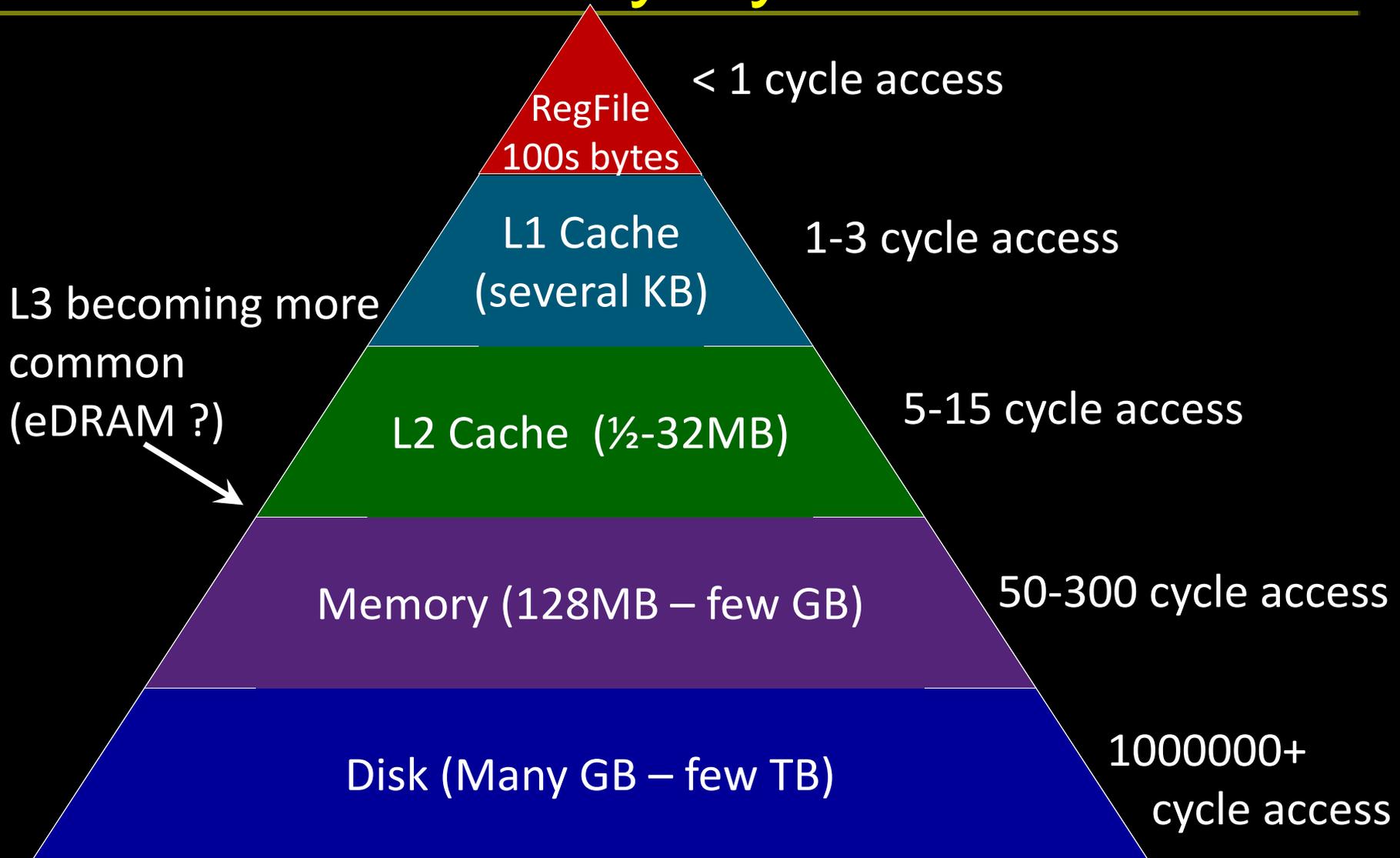- Operating System, Traps, Exceptions,

- Multicore and synchronization

Big Picture

Instruction Fetch · Instruction Decode · Execute · Memory · Write-Back

IF/ID · ID/EX · EX/MEM · MEM/WB

4

Memory Hierarchy and Caches

Memory Pyramid

RegFile
100s bytes

< 1 cycle access

L1 Cache
(several KB)

1-3 cycle access

L3 becoming more
common
(eDRAM ?)

L2 Cache (½-32MB)

5-15 cycle access

Memory (128MB – few GB)

50-300 cycle access

Disk (Many GB – few TB)

1000000+
cycle access

These are rough numbers: mileage may vary for latest/greatest
Caches usually made of SRAM (or eDRAM)

Memory Hierarchy
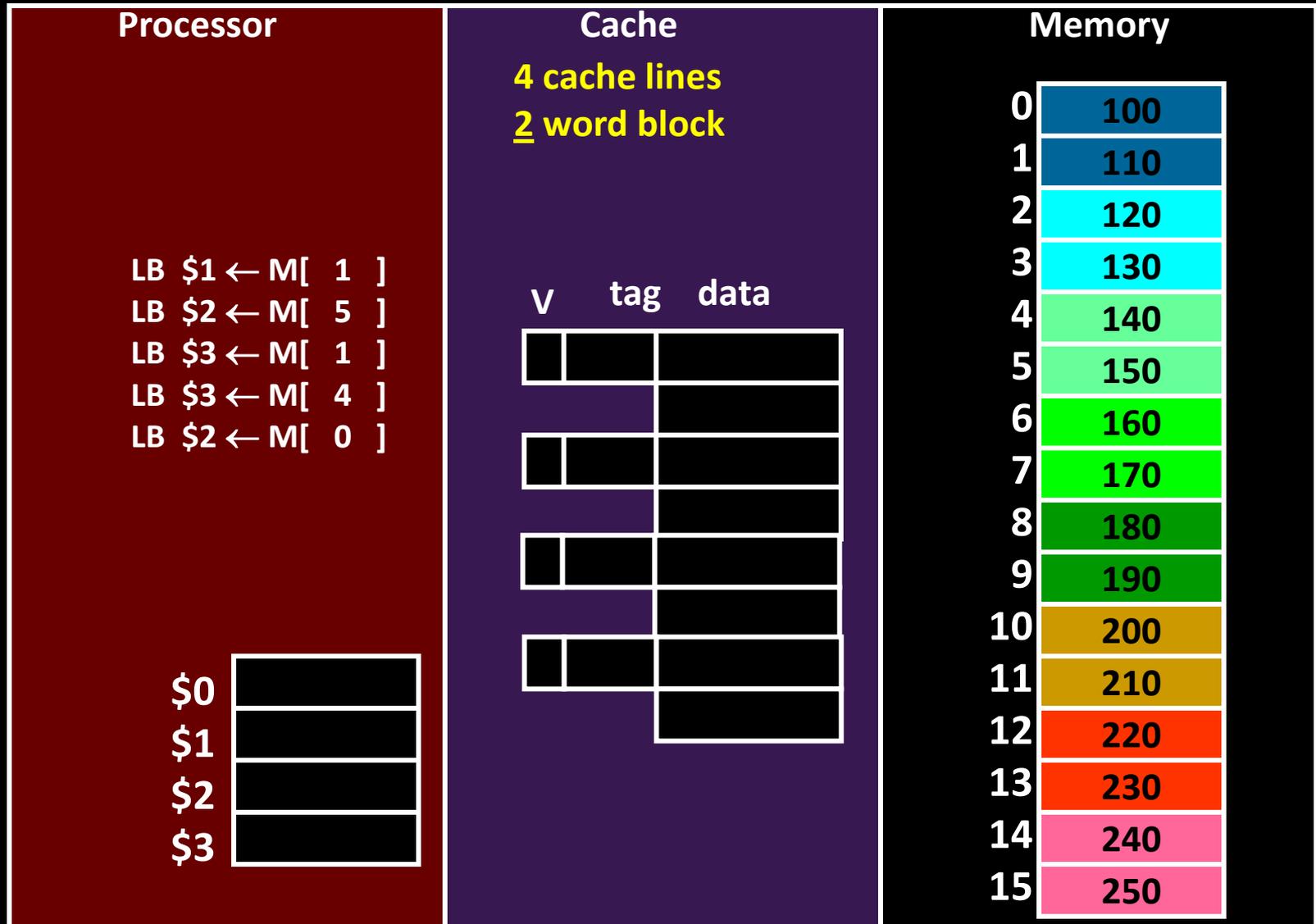
Insight for Caches

If Mem[x] is was accessed *recently*…

… then Mem[x] is likely to be accessed *soon*

- Exploit temporal locality:
  - Put recently accessed Mem[x] <u>higher</u> in memory hierarchy since it will likely be accessed again soon

… then Mem[$x \pm \varepsilon$] is likely to be accessed *soon*

- Exploit spatial locality:
  - Put entire block containing Mem[x] and surrounding addresses higher in memory hierarchy since nearby address will likely be accessed
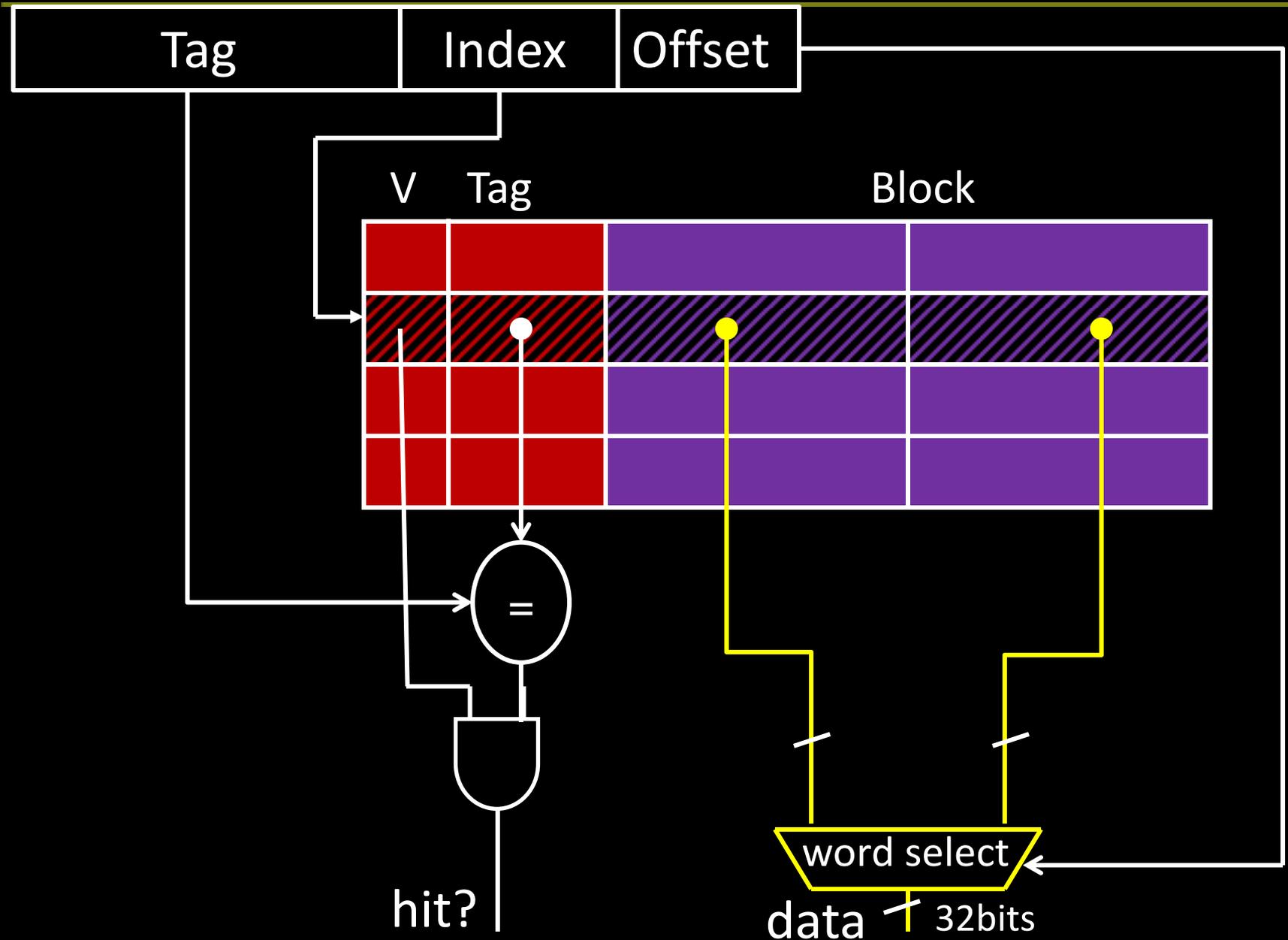
Memory Hierarchy

Processor

LB $1 ← M[1]
LB $2 ← M[5]
LB $3 ← M[1]
LB $3 ← M[4]
LB $2 ← M[0]

$0
$1
$2
$3

Cache

4 cache lines
2 word block

V   tag   data

Memory

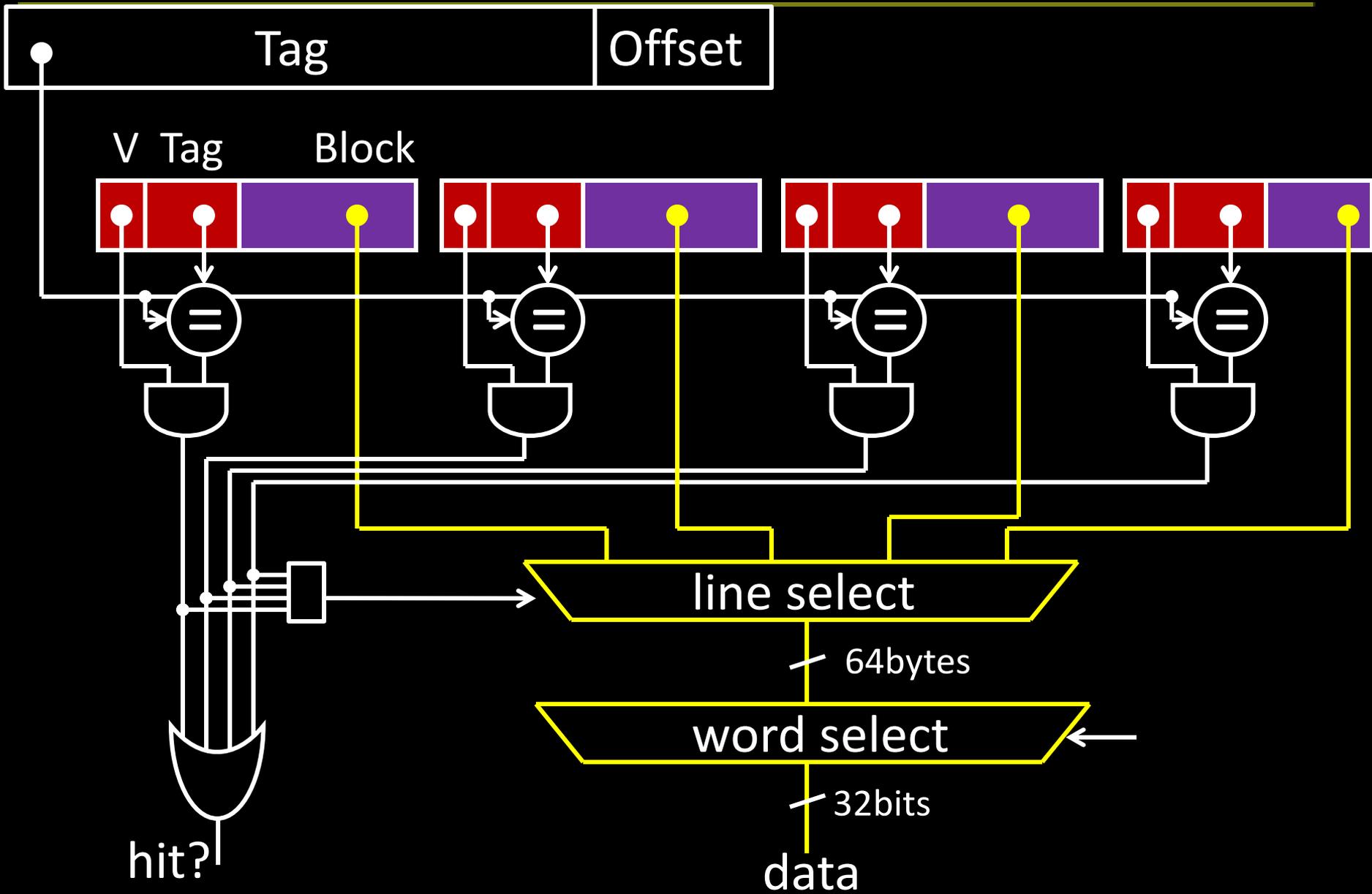|	
0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Three Common Cache Designs

A given data block can be placed…

- … in exactly one cache line → Direct Mapped

- … in any cache line → Fully Associative
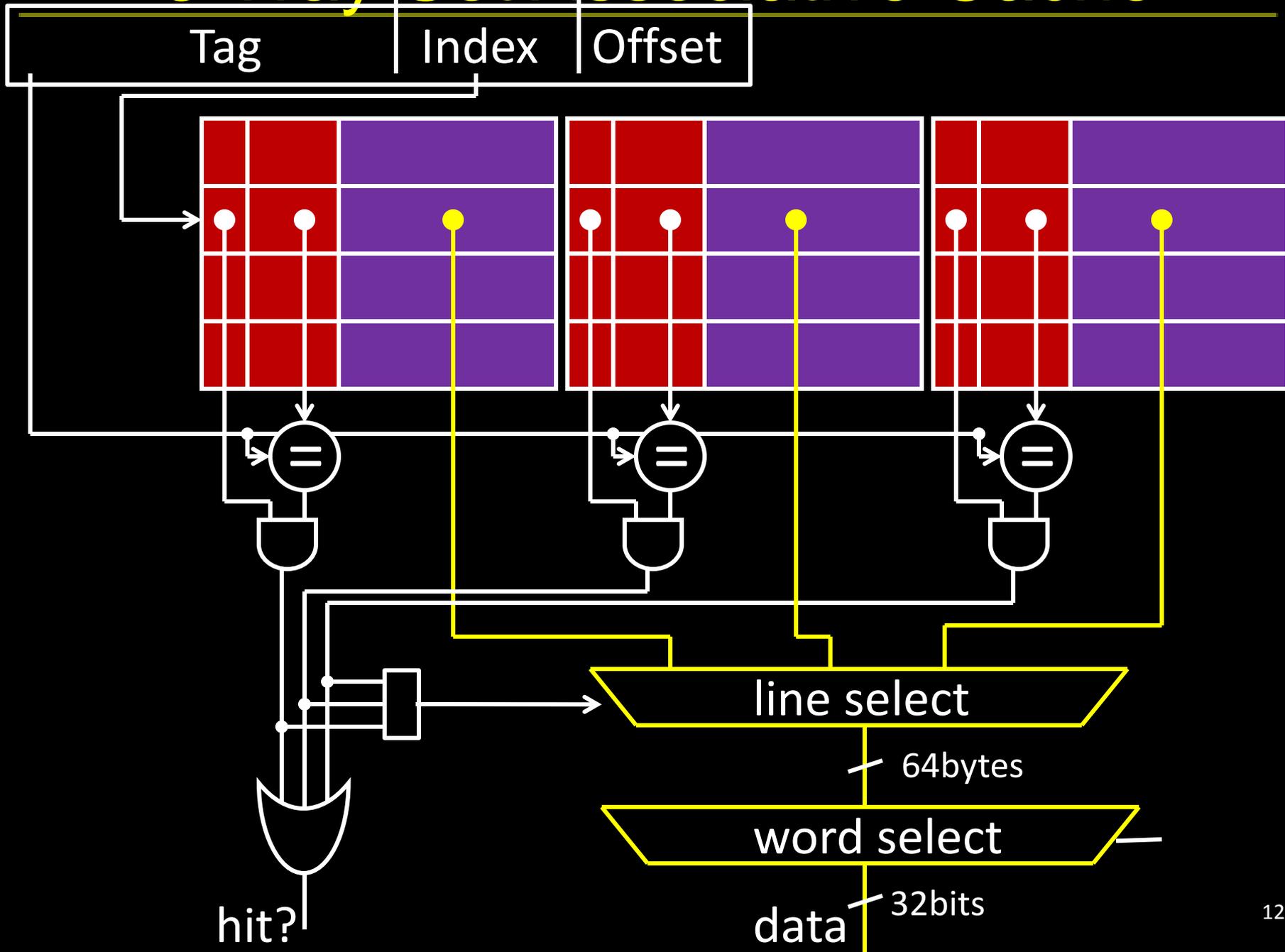
- … in a small set of cache lines → Set Associative

Direct Mapped Cache

Tag	Index	Offset

V   Tag   Block

=

hit?

word select

data 32bits

Fully Associative Cache

Tag | Offset

V Tag   Block

line select

64bytes

word select

32bits

hit?

data

3-Way Set Associative Cache

Tag | Index | Offset

= = =

hit?

line select

64bytes

word select

data 32bits

Cache *Misses*

- Three types of misses
  - Cold (aka Compulsory)
    - The line is being referenced for the first time
  - Capacity
    - The line was evicted because the cache was not large enough
  - Conflict
    - The line was evicted because of another access whose index conflicted

Writing with Caches

Eviction

Which cache line should be evicted from the cache to make room for a new line?

- Direct-mapped
  - no choice, must evict line selected by index
- Associative caches
  - random: select one of the lines at random
  - round-robin: similar to random
  - FIFO: replace oldest line
  - LRU: replace line that has not been used in the longest time

Cached Write Policies

Q: How to write data?
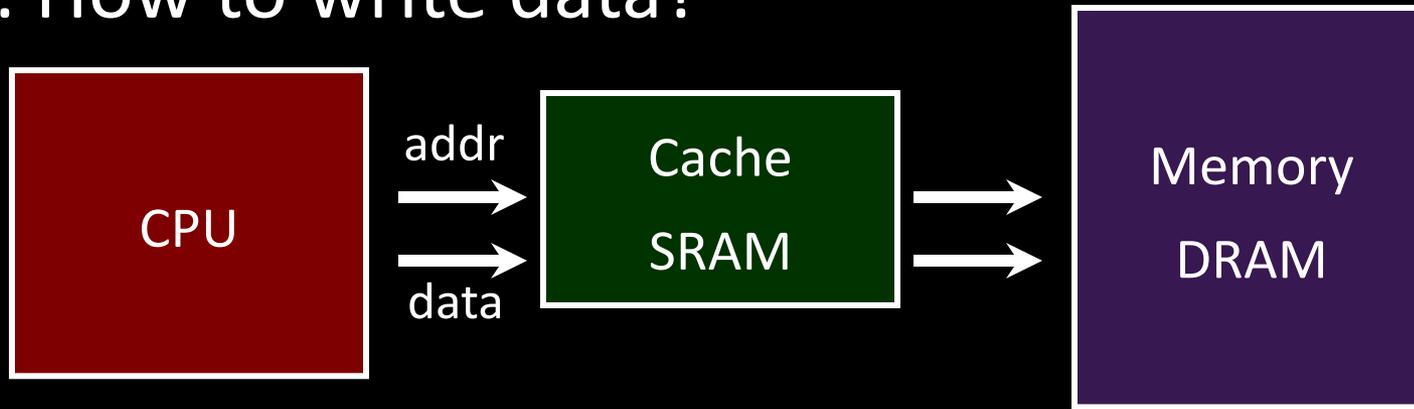
If data is already in the cache…

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)

What about Stores?

Where should you write the result of a store?

- If that memory location is in the cache?
    - Send it to the cache
    - Should we also send it to memory right away? (write-through policy)
    - Wait until we kick the block out (write-back policy)
- If it is not in the cache?
    - Allocate the line (put it in the cache)? (write allocate policy)
    - Write it directly to memory without allocation? (no write allocate policy)

Cache Performance

Cache Performance

- Consider hit (H) and miss ratio (M)
- $H \times AT_{cache} + M \times AT_{memory}$
- Hit rate = 1 – Miss rate
- Access Time is given in cycles
- Ratio of Access times, 1:50

- 90%   : .90   + .1 x 50   = 5.9
- 95%   : .95   + .05 x 50   = .95+2.5=3.45
- 99%   : .99   + .01 x 50   = 1.49
- 99.9%: .999 + .001 x 50 = 0.999 + 0.05 = 1.049

Cache Conscious Programming

Cache Conscious Programming

```
// H = 12, NCOL = 10

int A[NROW][NCOL];


for(col=0; col < NCOL; col++)

    for(row=0; row < NROW; row++)

        sum += A[row][col];
```

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

```
// NROW = 12, NCOL = 10
int A[NROW][NCOL];


for(row=0; row < NROW; row++)
    for(col=0; col < NCOL; col++)
        sum += A[row][col];
```

1	2	3	4	5	6	7	8	9	10
11	12	13	…						

Block size = 4 → 75% hit rate

Block size = 8 → 87.5% hit rate

Block size = 16 → 93.75% hit rate

And you can easily prefetch to warm the cache.

- MMU, Virtual Memory, Paging, and TLB's

Multiple Processes

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

CPU

CPU

0xfff...f

0x7ff...f

Stack

Heap

Data

Text

0x000...0
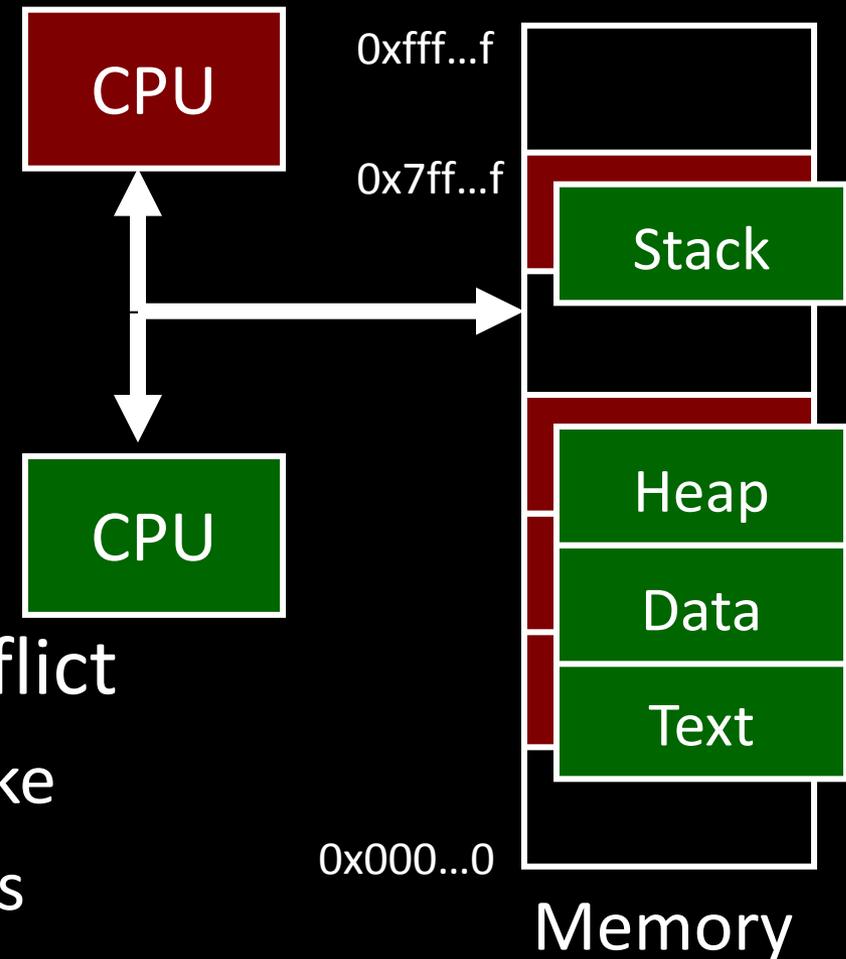
Memory

A: The addresses will conflict

- Even though, CPUs may take turns using memory bus

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 *indirect* through a virtual address
- translate fake virtual address to a real physical address
- redirect load/store to the physical address

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
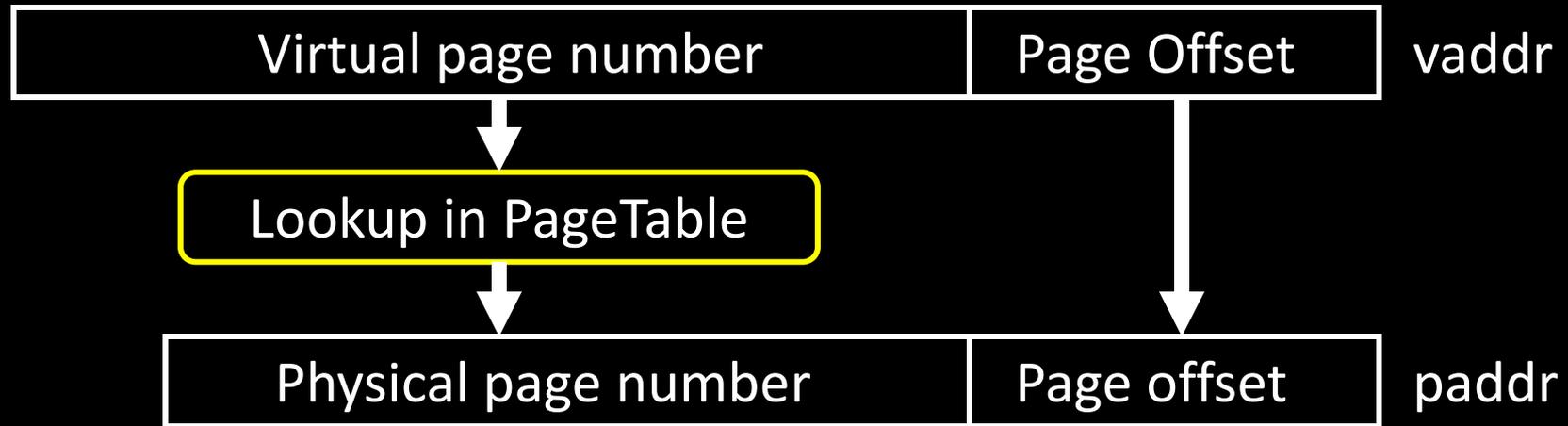
Different Permissions bits

Address Space

		X		
A	CPU	C	CPU	X
B		B		Y
C		Z		Z
	MMU	Y	MMU	
		A		
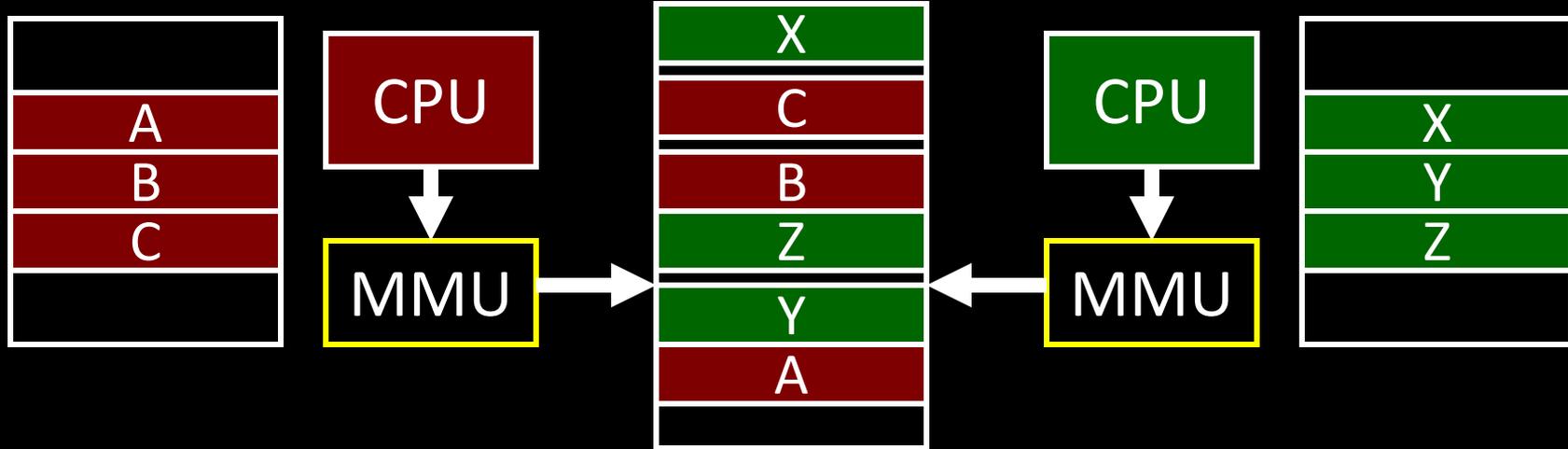
Programs load/store to virtual addresses

Actual memory uses physical addresses

Memory Management Unit (MMU)

- Responsible for translating on the fly

- Essentially, just a big array of integers:
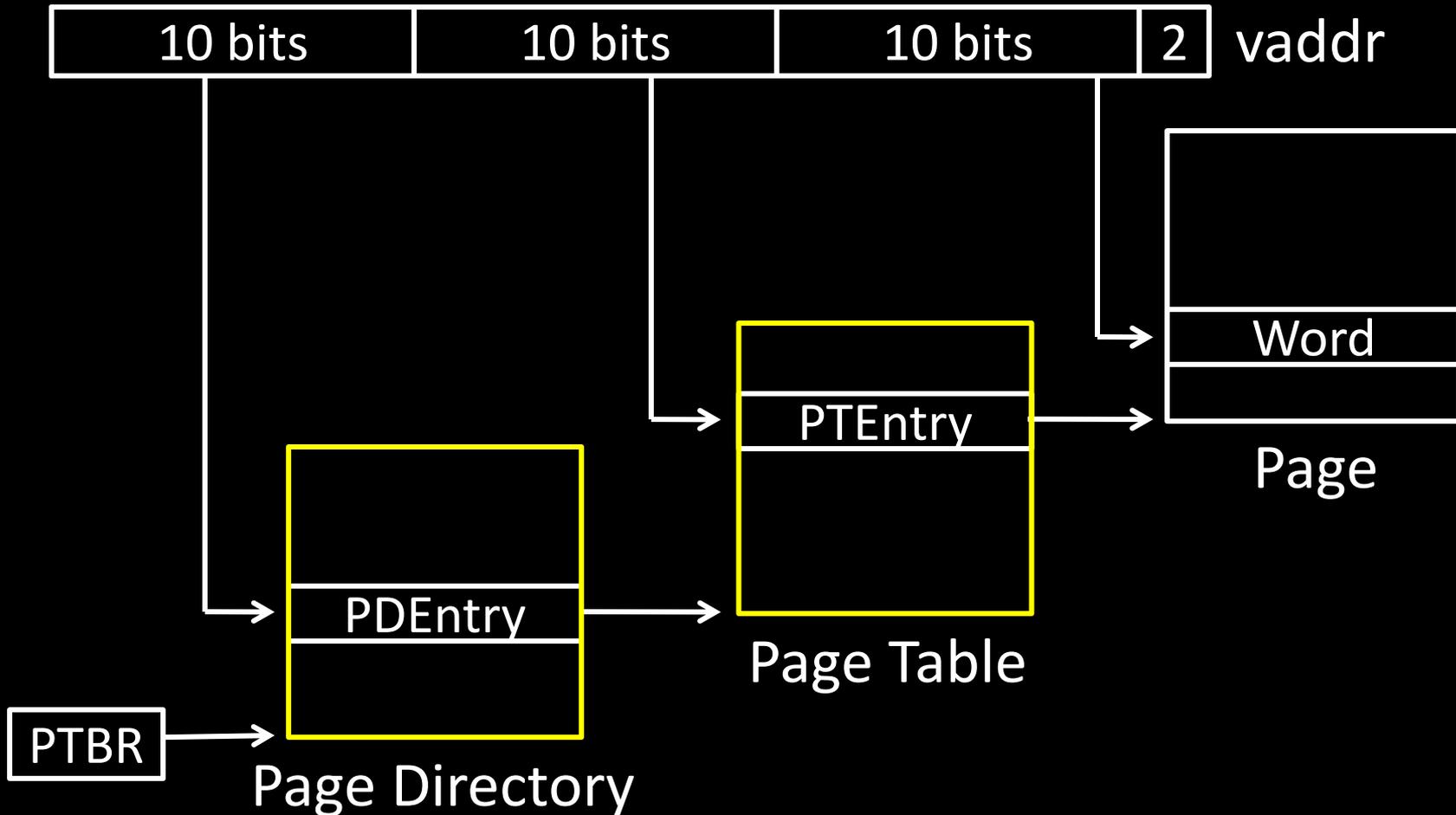 paddr = PageTable[vaddr];

Address Translation

Virtual page number	Page Offset	vaddr

Lookup in PageTable

Physical page number	Page offset	paddr

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

Beyond Flat Page Tables

Assume most of PageTable is empty
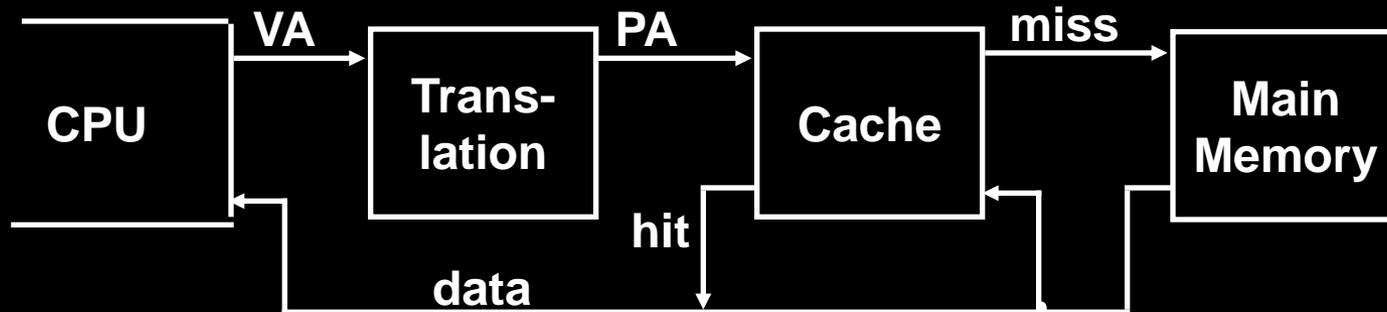
How to translate addresses? Multi-level PageTable

| 10 bits | 10 bits | 10 bits | 2 | vaddr |

Word

Page

PTEntry

Page Table

PDEntry

Page Directory

PTBR

* x86 does exactly this

Virtual Addressing with a Cache

- Thus it takes an *extra* memory access to translate a VA to a PA

```
        VA              PA             miss
┌──────┐   ┌────────┐      ┌──────┐        ┌────────┐
│      │──▶│ Trans- │─────▶│      │───────▶│  Main  │
│ CPU  │   │ lation │      │ Cache│        │ Memory │
│      │◀─┐└────────┘      └──────┘◀───┐   └────────┘
└──────┘  │          hit  │            │
          │               │            │
          │   data        ▼            │
          └───────────────────────────┘
```

- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)

A TLB in the Memory Hierarchy
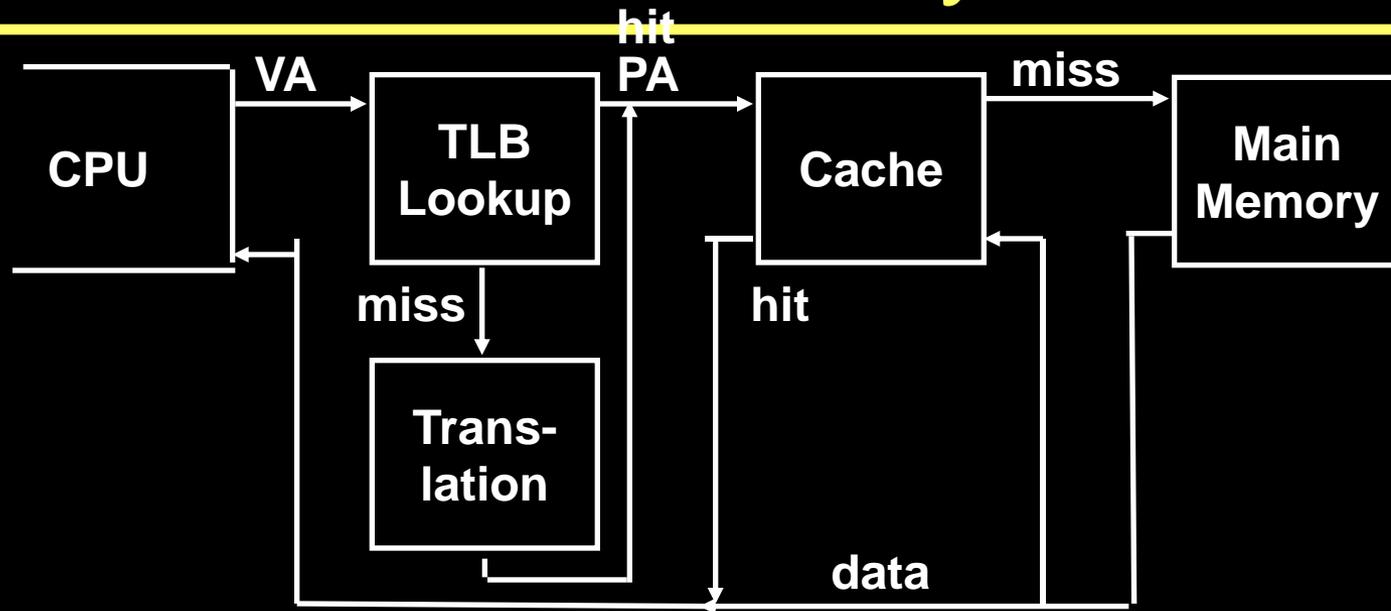
- A TLB miss:
  - If the page is not in main memory, then it's a true page fault
    - Takes 1,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults

Virtual vs. Physical Caches

Cache works on physical addresses

Cache works on virtual addresses
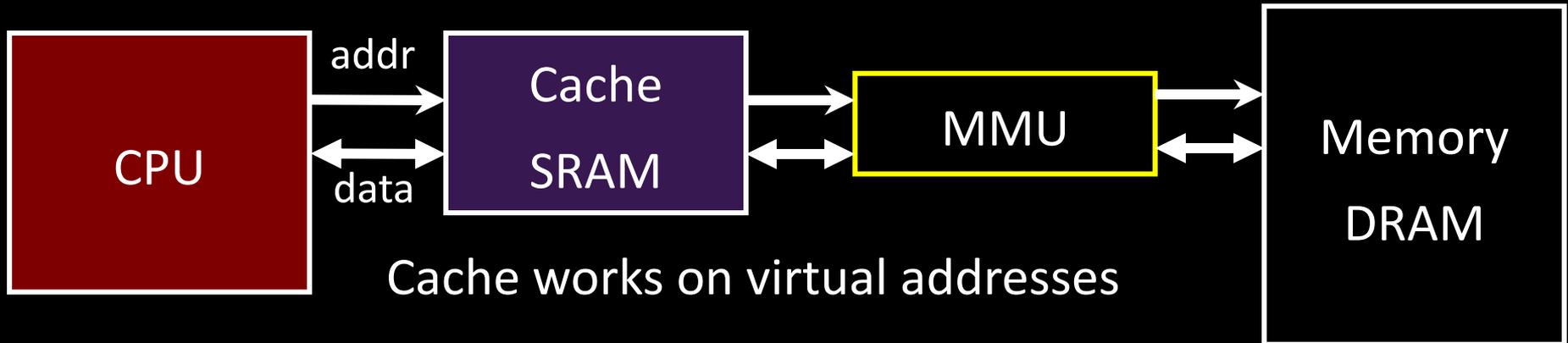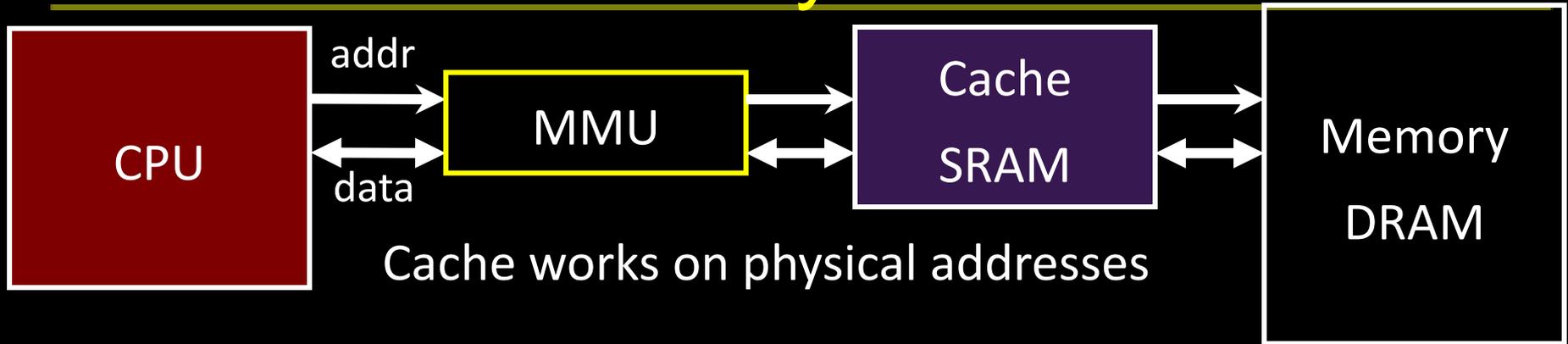
Q: What happens on context switch?
Q: What about virtual memory aliasing?
Q: So what's wrong with physically addressed caches?

Indexing vs. Tagging

Physically-Addressed Cache

- slow: requires TLB (and maybe PageTable) lookup first

Virtually-Addressed 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)
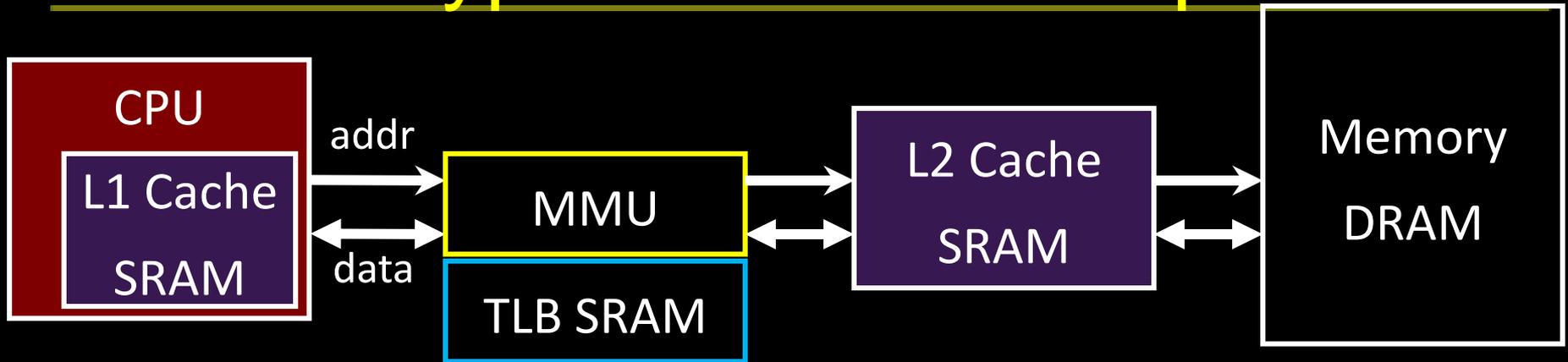   - → could end up in cache twice (very bad!)

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

Indexing vs. Tagging
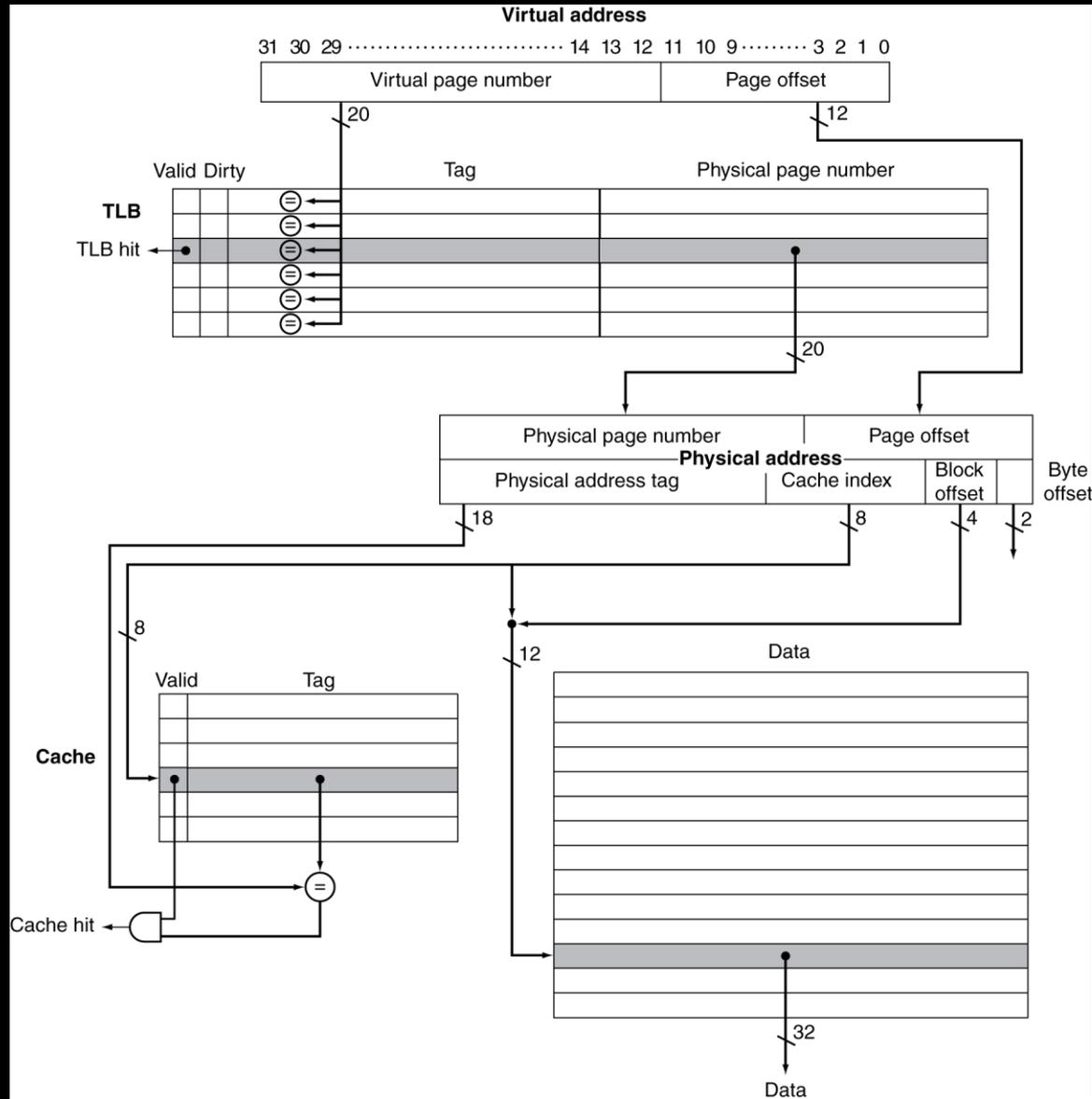
Typical Cache Setup

Typical L1: On-chip virtually addressed, physically tagged

Typical L2: On-chip physically addressed

Typical L3: On-chip …

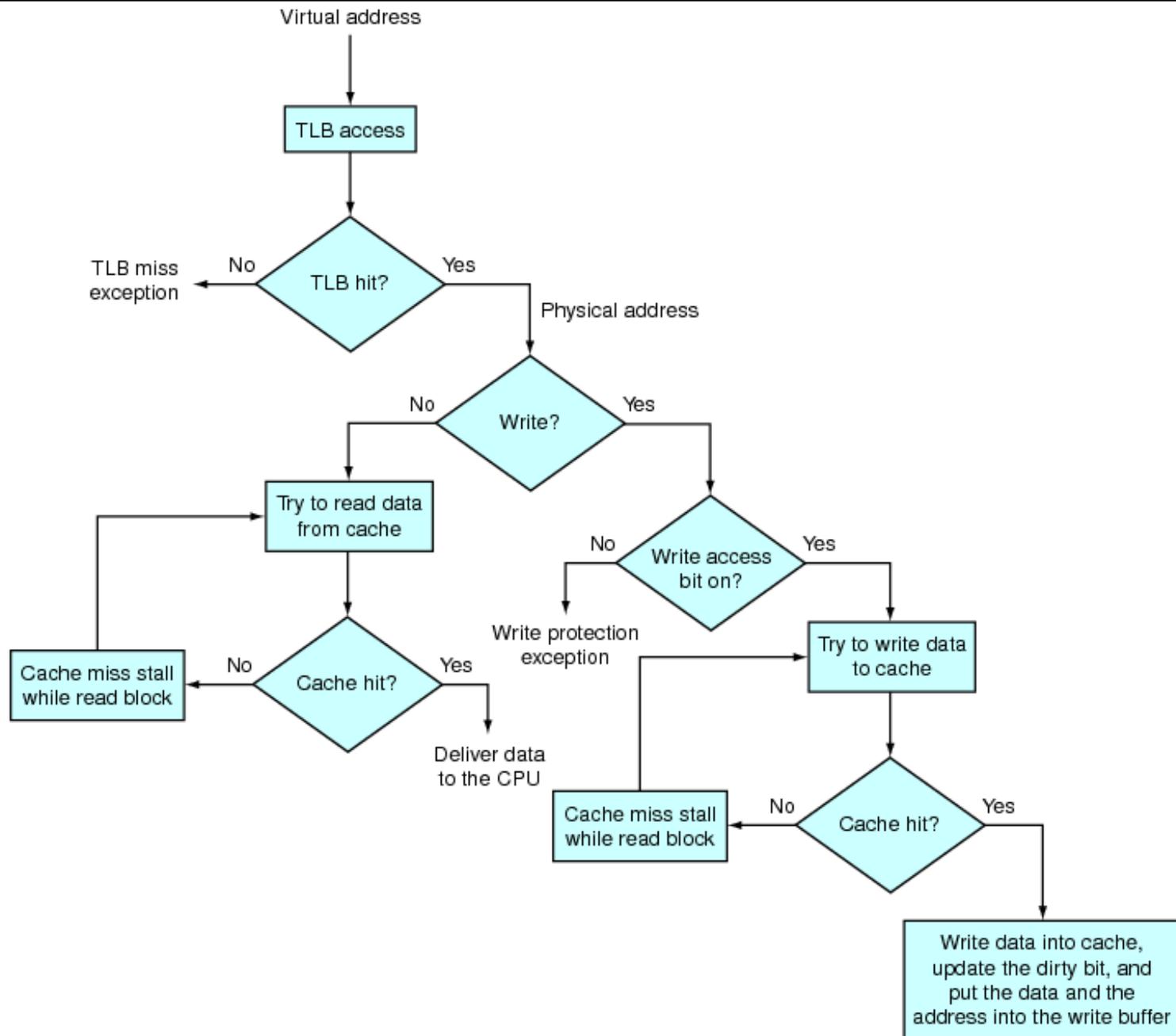# Hardware/Software Boundary

Hardware/Software Boundary

- Virtual to physical address translation is assisted by hardware?
  - Translation Lookaside Buffer (TLB) that caches the recent translations
    - TLB access time is part of the cache hit time
    - May allot an extra stage in the pipeline for TLB access
  - TLB miss
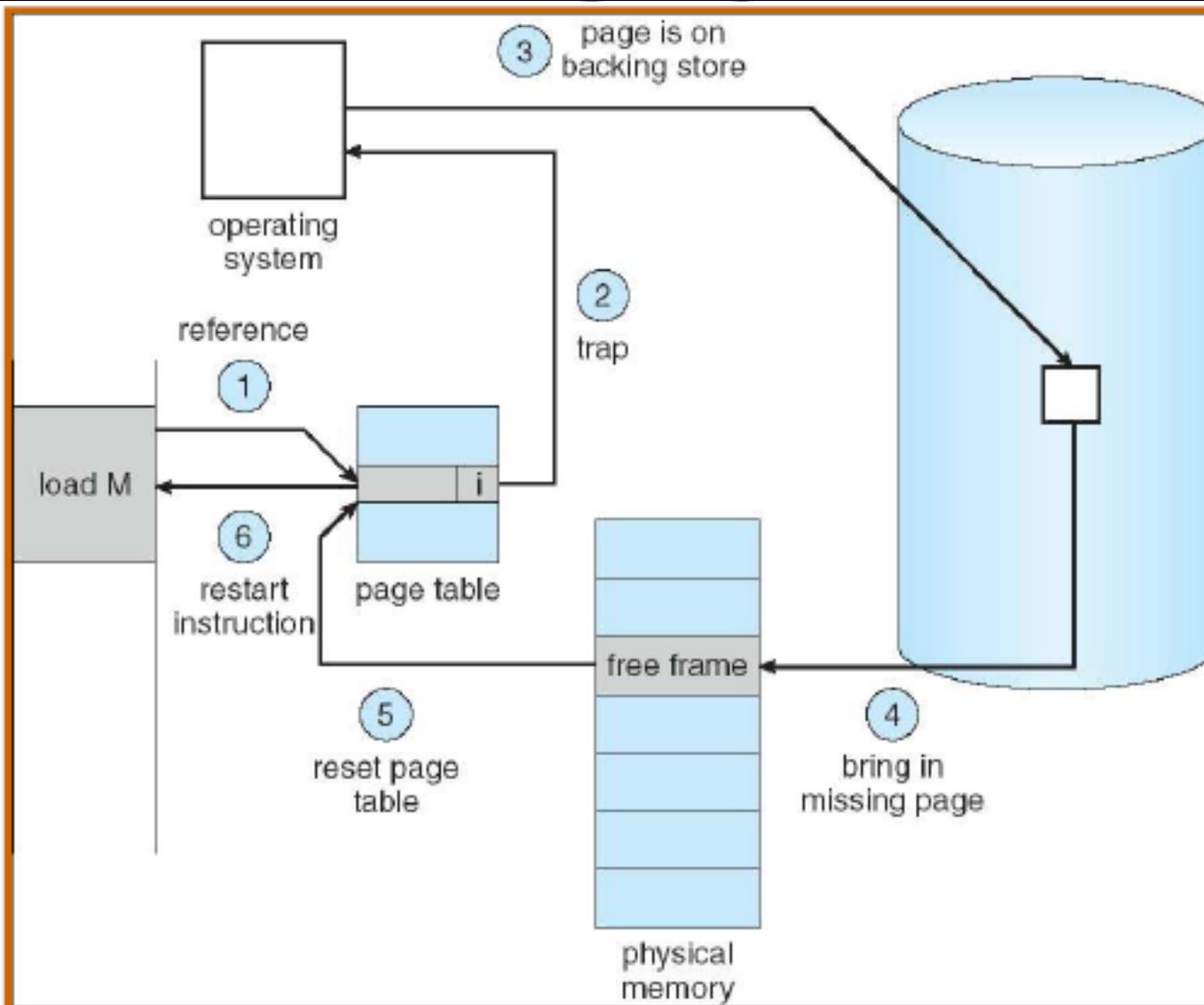   - Can be in software (kernel handler) or hardware

Hardware/Software Boundary

- Virtual to physical address translation is assisted by hardware?
  - Page table storage, fault detection and updating
    - Page faults result in interrupts (precise) that are then handled by the OS
    - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the Page Tables

Virtual address

TLB access

TLB hit?
No → TLB miss exception
Yes → Physical address

Write?
No → Try to read data from cache
Yes → Write access bit on?

Try to read data from cache

Cache hit?
No → Cache miss stall while read block
Yes → Deliver data to the CPU

Write access bit on?
No → Write protection exception
Yes → Try to write data to cache

Try to write data to cache

Cache hit?
No → Cache miss stall while read block
Yes → Write data into cache, update the dirty bit, and put the data and the address into the write buffer

Paging

- Traps, exceptions, and operating system

Operating System

Some things not available to untrusted programs:

- Exception registers, HALT instruction, MMU instructions, talk to I/O devices, OS memory, ...

Need trusted mediator: Operating System (OS)

- *Safe control transfer*
- *Data isolation*

P1	P2	P3	P4

VM   filesystem   net
driver   driver

MMU   disk   eth
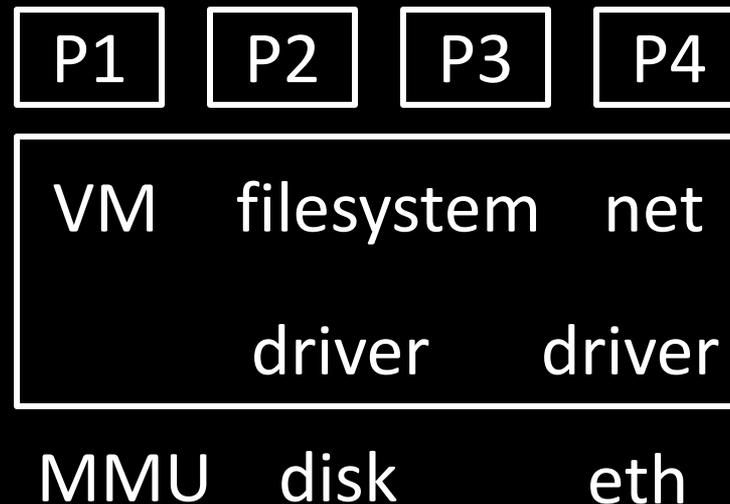
Terminology

Trap: Any kind of a control transfer to the OS

Syscall: Synchronous (planned), program-to-kernel transfer
- SYSCALL instruction in MIPS (various on x86)

Exception: Synchronous, program-to-kernel transfer
- exceptional events: div by zero, page fault, page protection err, …

Interrupt: Aysnchronous, device-initiated transfer
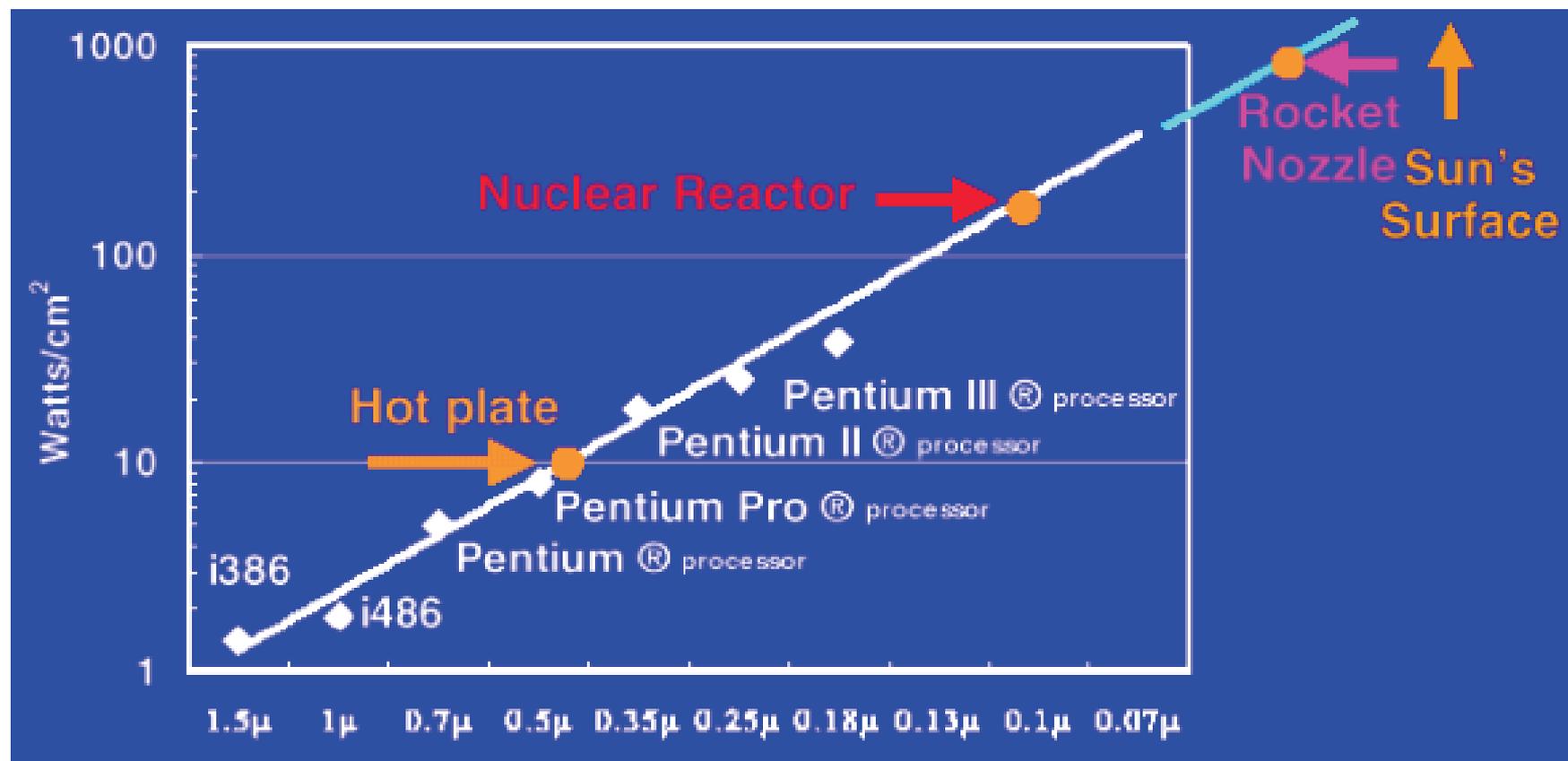- e.g. Network packet arrived, keyboard event, timer ticks

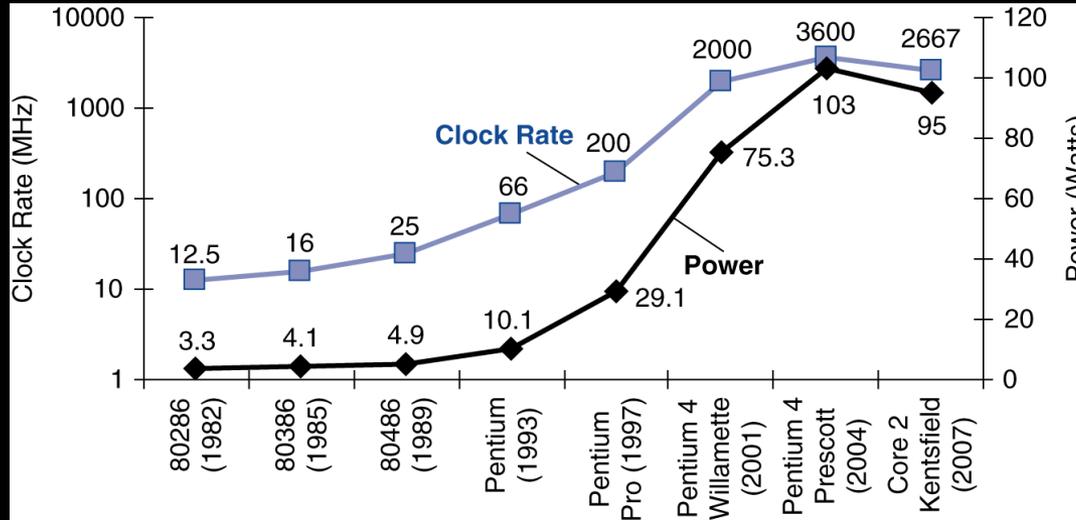* real mechanisms, but nobody agrees on these terms

- Multicore and Synchronization

- Multi-core is a reality…

- … but how do we write multi-core safe code?

Why Multicore?

- ## Moore's law
  - A law about transistors
    (Not speed)
  - Smaller means faster transistors

- Power consumption growing with transistors

Power Limits Performance

Power Trends

- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30   5V → 1V   ×1000

Uniprocessor Performance
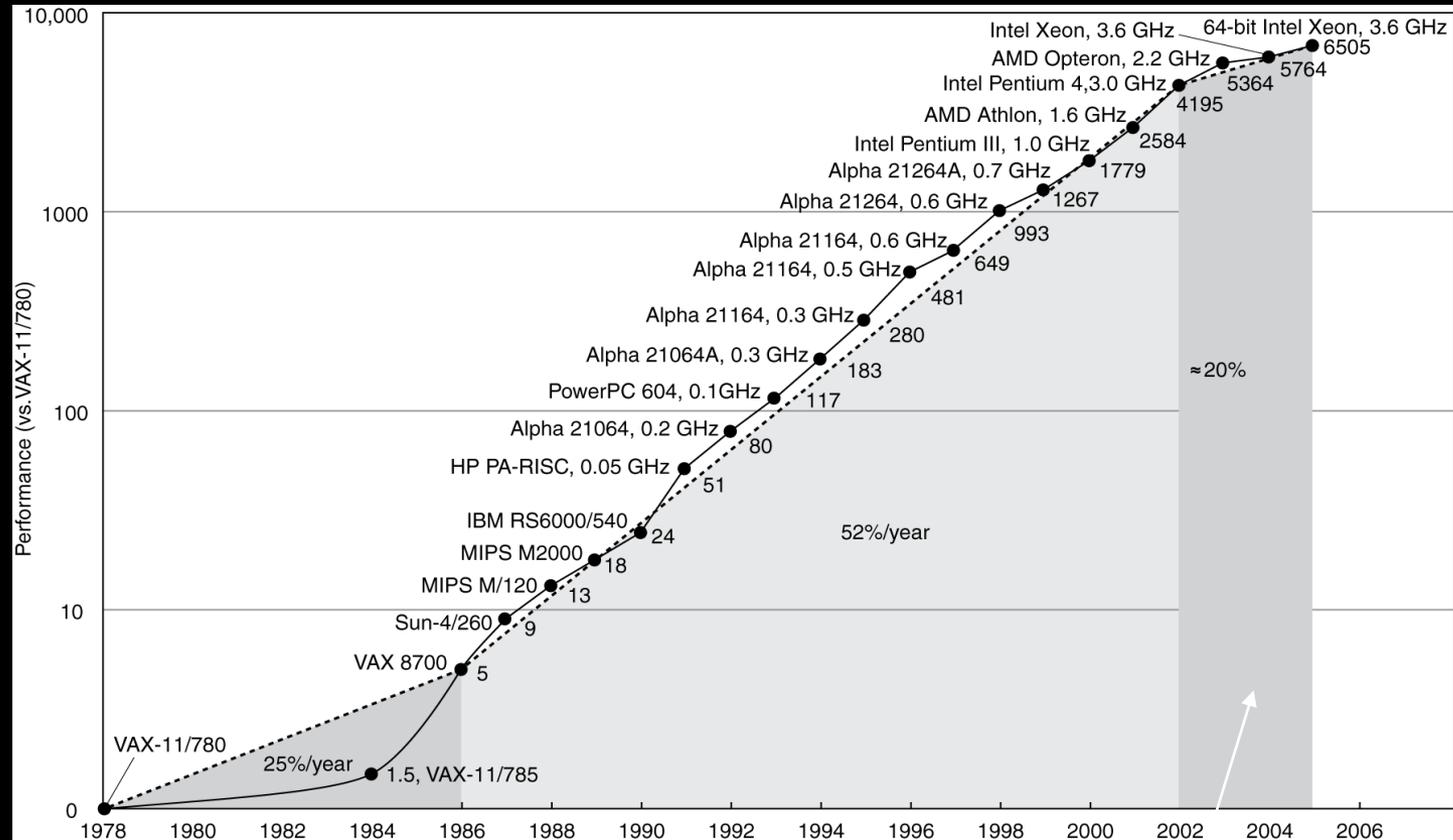
Constrained by power, instruction-level parallelism, memory latency

Why Multicore?

- Moore's law
  - A law about transistors
  - Smaller means faster transistors

- Power consumption growing with transistors

- The power wall
  - We can't reduce voltage further
  - We can't remove more heat
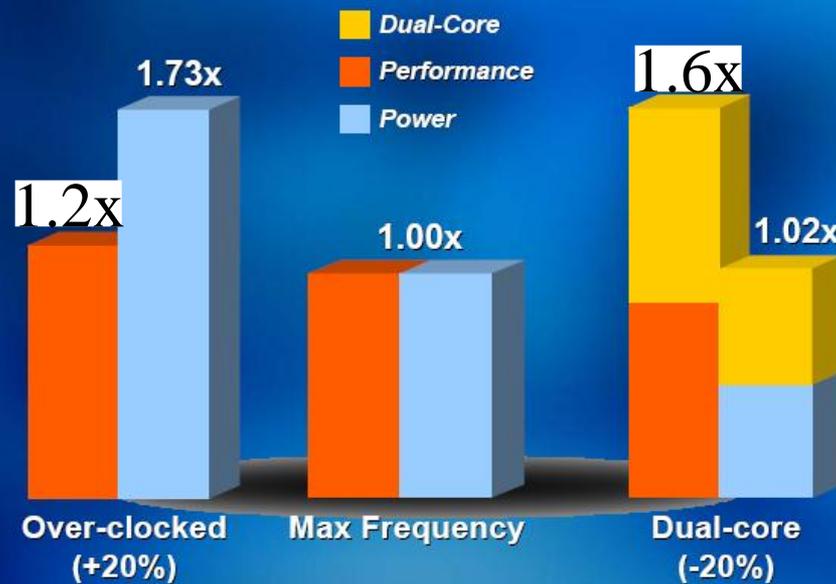- How else can we improve performance?

Intel's argument
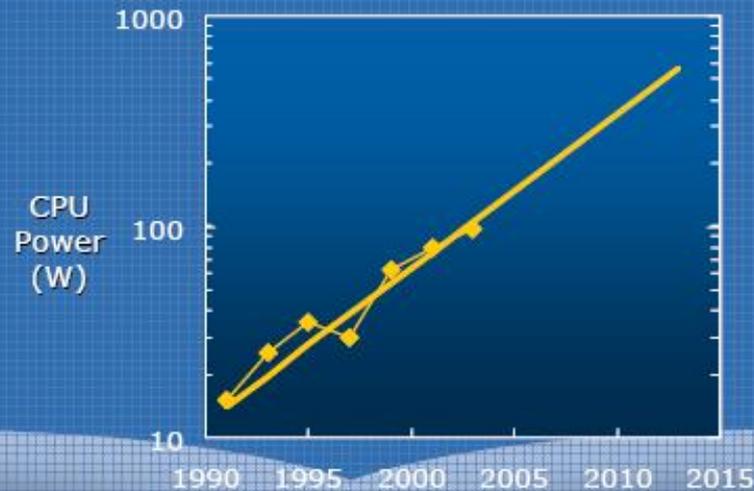
Amdahl's Law

- Task: serial part, parallel part
- As number of processors increases,
  - time to execute parallel part goes to zero
  - time to execute serial part remains the same
- *Serial part eventually dominates*
- Must parallelize ALL parts of task

$$\text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E}$$
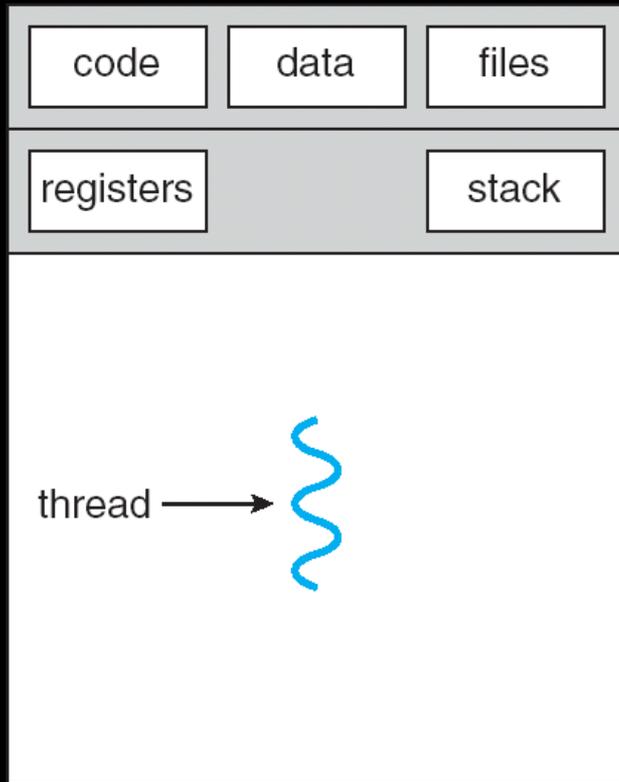
Amdahl's Law

- Consider an improvement E
- F of the execution time is affected
- S is the speedup

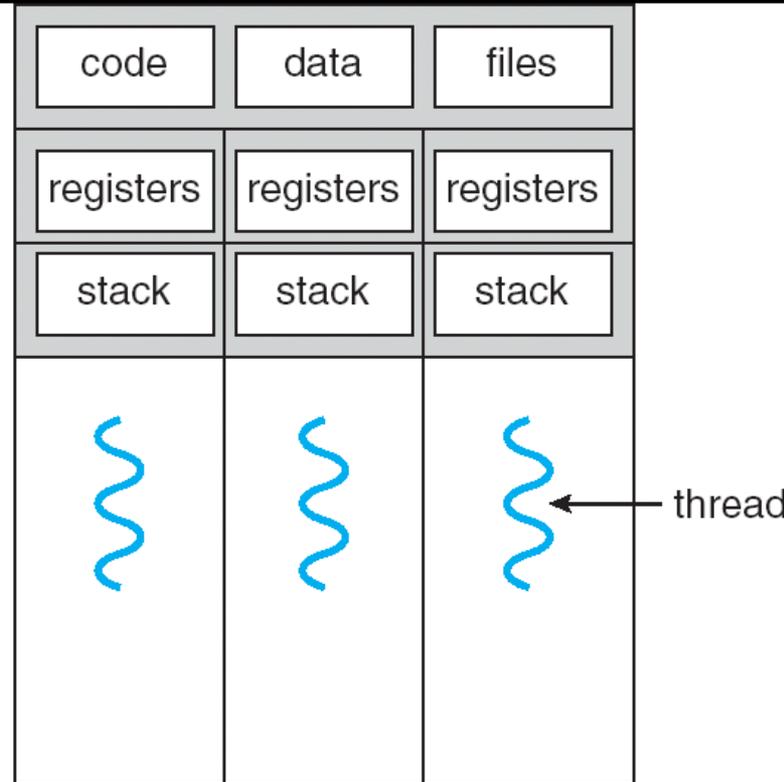$$\text{Execution time (with } E) = ((1 - F) + F/S) \cdot \text{Execution time (without } E)$$

$$\text{Speedup (with } E) = \frac{1}{(1 - F) + F/S}$$

Multithreaded Processes

Shared counters
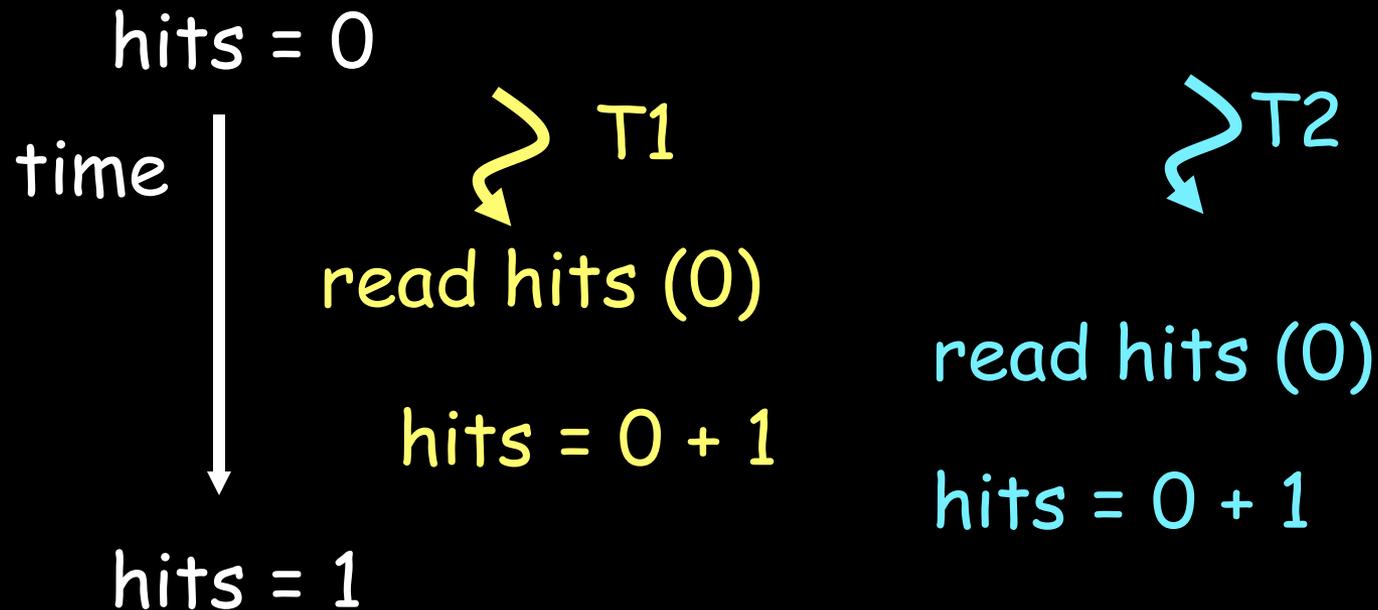
- Usual result: works fine.
- Possible result: lost update!

hits = 0

time

T1

read hits (0)

hits = 0 + 1

hits = 1

T2

read hits (0)

hits = 0 + 1

- Occasional timing-dependent failure $\Rightarrow$ Difficult to debug
- Called a *race condition*

Race conditions

- Def: a timing dependent error involving shared state
  - Whether it happens depends on how threads scheduled: who wins "races" to instructions that update state
  - Races are intermittent, may occur rarely
    - Timing dependent = small changes can hide bug
  - A program is correct *only* if *all possible* schedules are safe
    - Number of possible schedule permutations is huge
    - Need to imagine an adversary who switches contexts at the worst possible time

Critical Sections

- Basic way to eliminate races: use *critical sections* that only one thread can be in
  - Contending threads must wait to enter

time

T1

```
CSEnter();
    Critical section
CSExit();
```

T1

T2

```
CSEnter();
    Critical section
CSExit();
```

T2
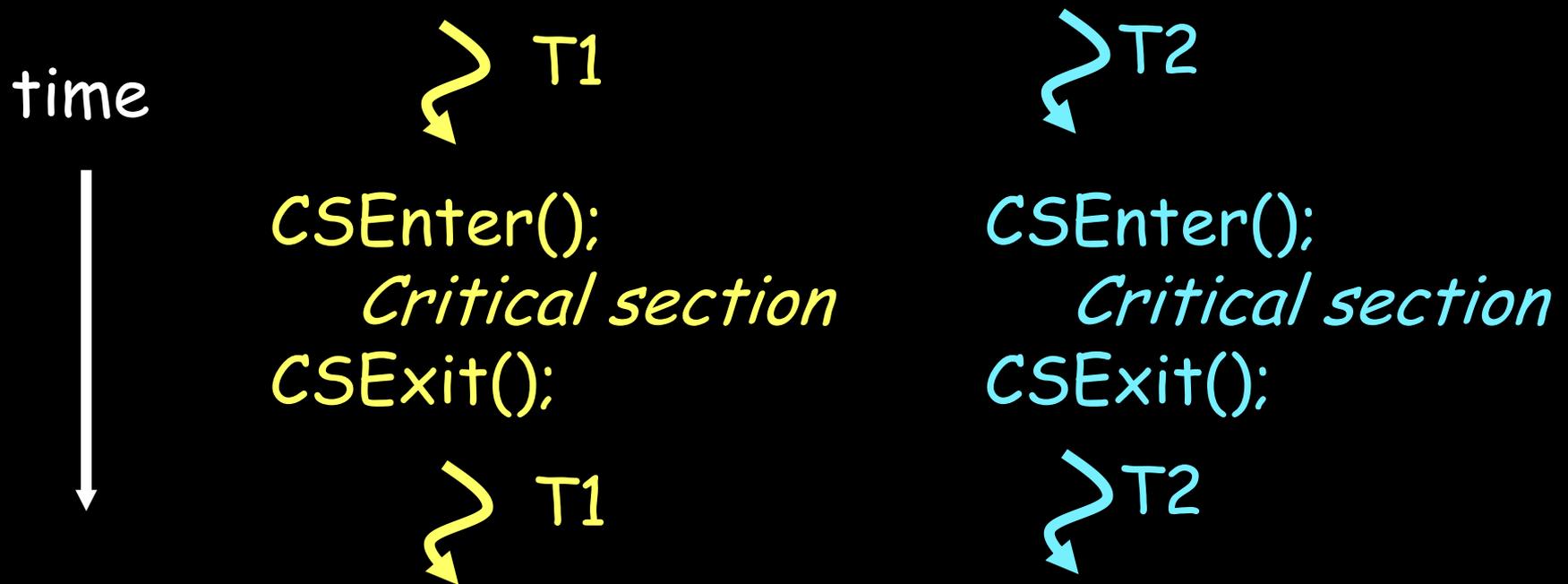
Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire (lock) mutex on entry to critical section
  - Or block if another thread already holds it
- Release (unlock) mutex on exit
  - Allow one waiting thread (if any) to acquire & proceed

```
                pthread_mutex_init(m);

pthread_mutex_lock(m);      pthread_mutex_lock(m);
    hits = hits+1;              hits = hits+1;
pthread_mutex_unlock(m);    pthread_mutex_unlock(m);
```

T1   T2

Protecting an invariant

```
// invariant: data is in buffer[first..last-1]. Protected by m.
pthread_mutex_t *m;
char buffer[1000];
int first = 0, last = 0;

void put(char c) {
    pthread_mutex_lock(m);
    buffer[last] = c;
    last++;
    pthread_mutex_unlock(m);
}

char get() {
    pthread_mutex_lock(m);
    char c = buffer[first];
    first++;          X what if first==last?
    pthread_mutex_unlock(m);
}
```

- Rule of thumb: all updates that can affect invariant become critical sections.

See you Tonight
Good Luck!