Synchronization and Prelim 2 Review

Hakim Weatherspoon
CS 3410, Spring 2011
Computer Science
Cornell University

Announcements

FarmVille pizza Party was fun!

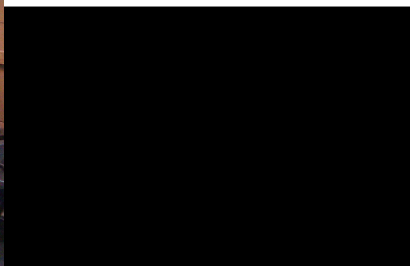
- Winner: Team HakimPeterSpoon

Joseph Mongeluzzi and Jason Zhao

Seed	1st Round	Sweet 16	Elite Eight	Final Four
1	hakimPeterSpoon	hakimPeterSpoon	hakimPeterSpoon	hakimPeterSpoon
32	bossbot			
16	seeder	seeder		
17	ryanbot			
9	tsapanzerhalis	tsapanzerhalis	tsapanzerhalis	
24	superfarmer			
8	drunkbot	drunkbot		
25	housandFlippingVegetables			
4	WALLE	WALLE	WALLE	QihoTheShark
29	etter_than_peters_bots			
13	petey9	enb_lw_farmer		
20	enb_lw_farmer			
12	aen_dpm_farmer	aen_dpm_farmer	QihoTheShark	
21	exploit			
5	QihoTheShark	QihoTheShark		
28	fertile_cresent			

Championship		
hakimPeterSpoon	**hakimPeterSpoon**	gaxlr

Final Four	Elite Eight	Sweet 16	1st Round	Seed
gaxlr	gaxlr	gaxlr	gaxlr	2
			ams_jhj_test	31
		eli_whitney	Sparkles	15
			eli_whitney	18
	vanessaKensington	vanessaKensington	vanessaKensington	10
			our_farmer	23
		cheese	cheese	7
			jhl_ncn_farmer	26
better_greedy	phantom	phantom	phantom	3
			testExploit12	30
		dcv_prd_farmer	dcv_prd_farmer	14
			fastfiller	19
	better_greedy	better_greedy	better_greedy	11
			chomp	22
		frl_hd_farmer	frl_hd_farmer	6
			mjbebe	27



Announcements

HW4 due *today*, Tuesday, April 26th

- Work alone

Next two weeks

- Prelim2 will ***in-class*** *this* Thursday, April 28th
  - 90 minutes in class: 1:25pm – 2:55pm
  - Topics: Caching, Virtual Memory, Paging, TLBs, I/O, Traps, Exceptions, multicore, and synchronization

- PA4 will be final project (no final exam)
  - Available this Friday, April 29th and due Friday, May 13th
  - Need to schedule time for presentation May 16, 17, or 18.
  - ***Will not be able to use slip days***

Goals for Today

Finish Synchronization

- Threads and processes
- Critical sections, race conditions, and mutexes

Prelim 2 review

- Caching,
- Virtual Memory, Paging, TLBs
- I/O and DMA
- Operating System, Traps, Exceptions,
- Multicore and synchronization

Multi-core is a reality…

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

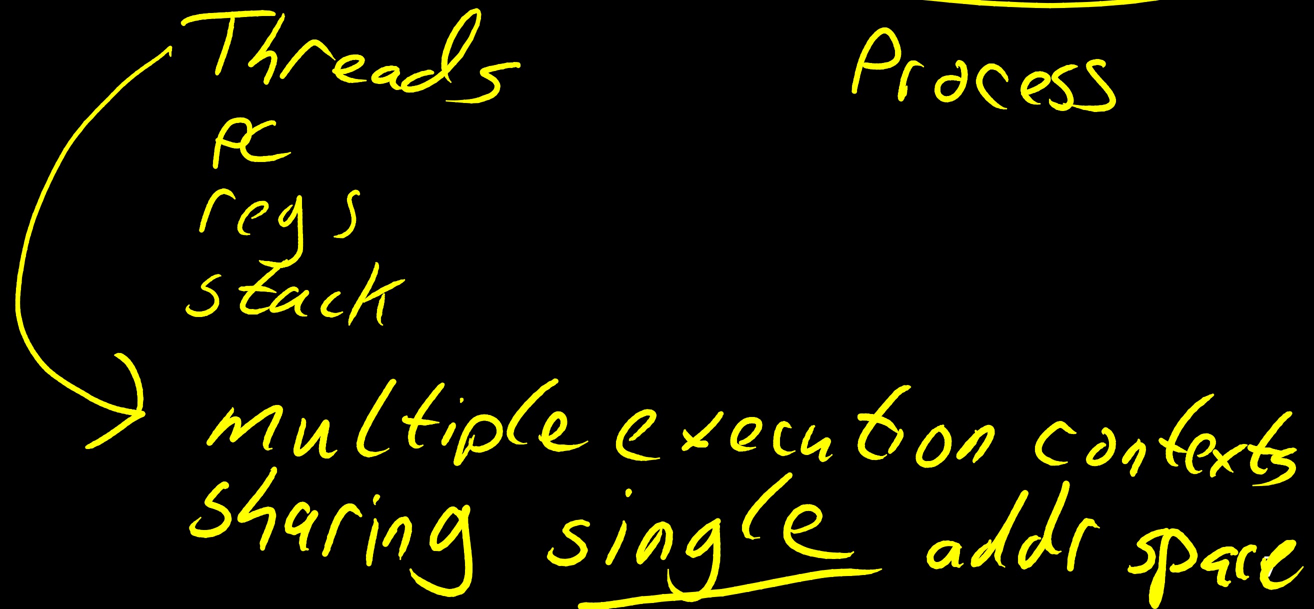
Processes and Threads

Processes are heavyweight

Parallel programming with processes:

- They share almost everything
  code, shared mem, open files, filesystem privileges, …
- Pagetables will be *almost* identical
- Differences: PC, registers, stack

Recall: process = execution context + address space



Processes and Threads

Process

OS abstraction of a running computation

- The unit of execution
- ~~The unit of scheduling~~
- Execution state
  + address space

From process perspective

- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, …

Thread

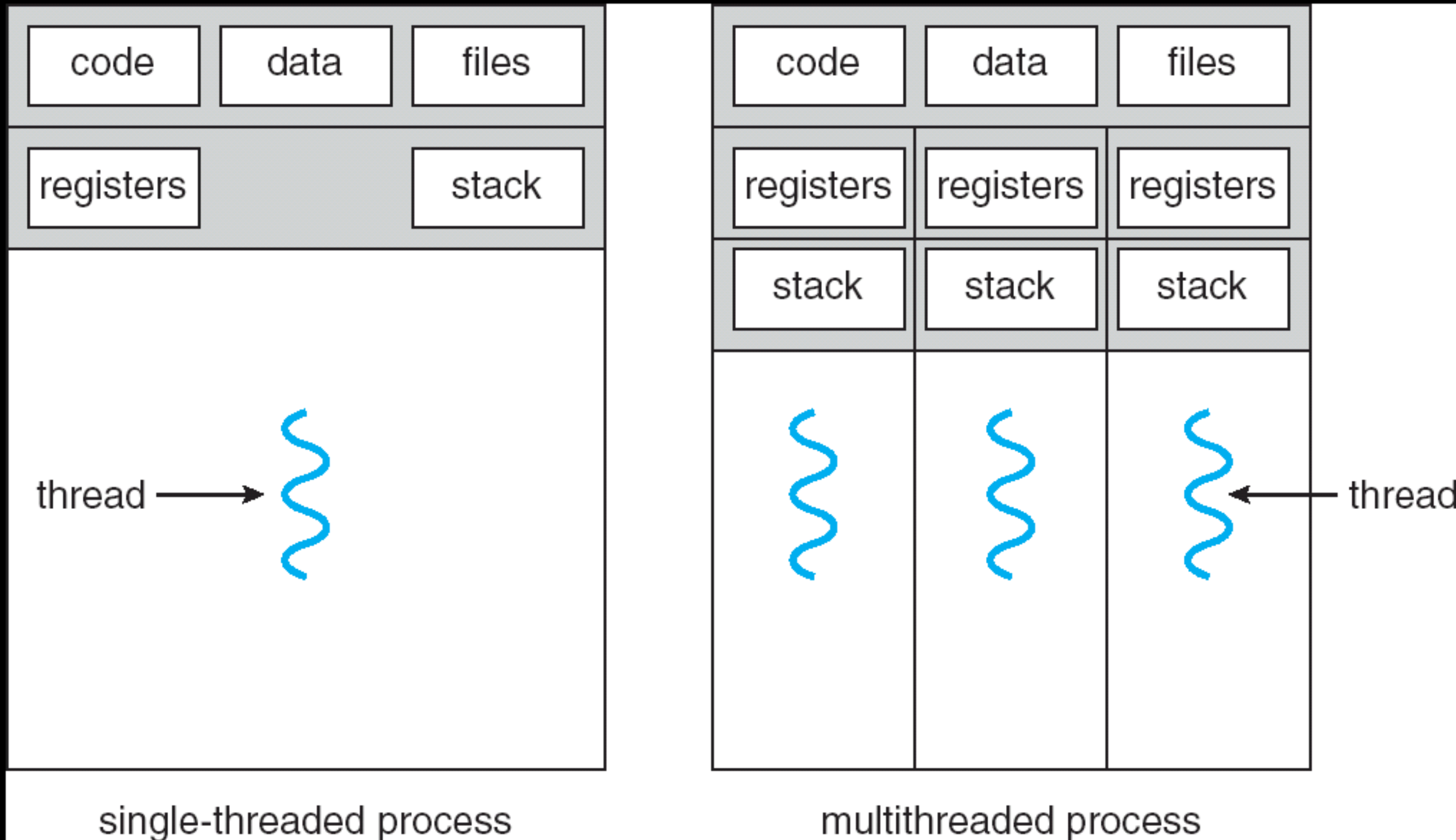
OS abstraction of a single thread of control

- The unit of scheduling
- Lives in one single process

From thread perspective

- one virtual CPU core on a virtual multi-core machine

Multithreaded Processes



Threads

```
#include <pthread.h>
int counter = 0;

void PrintHello(int arg) {
   printf("I'm thread %d, counter is %d\n", arg, counter++);
   ... do some work ...
   pthread_exit(NULL);
}

int main () {
   for (t = 0; t < 4; t++) {
       printf("in main: creating thread %d\n", t);
       pthread_create(NULL, NULL, PrintHello, t);
    }
    pthread_exit(NULL);
}
```

Threads versus Fork

```
in main: creating thread 0
I'm thread 0, counter is 0
in main: creating thread 1
I'm thread 1, counter is 1
in main: creating thread 2
in main: creating thread 3
I'm thread 3, counter is 2
I'm thread 2, counter is 3
```

If processes?

Example Multi-Threaded Program

Example: Apache web server

```
void main() {
  setup();
  while (c = accept_connection()) {

    req = read_request(c);
    hits[req]++;
    send_response(c, req);

  }
  cleanup();
}
```

— shared counter

Race Conditions

Example: Apache web server

Each client request handled by a separate thread (in parallel)

- Some shared state: hit counter, ...

Thread 52

```
read hits
addi
write hits
```

Thread 205

```
read hits
addi
write hits
```

(look familiar?)

Timing-dependent failure ⇒ race condition

- hard to reproduce ⇒ hard to debug

Programming with threads

Within a thread: execution is sequential

Between threads?

- No ordering or timing guarantees
- Might even run on different cores at the same time

Problem: hard to program, hard to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Cache coherency isn't sufficient…

Need explicit synchronization to make sense of concurrency!

Managing Concurrency

Races, Critical Sections, and Mutexes

Goals

Concurrency Goals

Liveness

- Make forward progress

Efficiency

- Make good use of resources

Fairness

- Fair allocation of resources between threads

Correctness

- Threads are isolated (except when they aren't)

Race conditions

Race Condition

Timing-dependent error when accessing shared state

- Depends on scheduling happenstance
  … e.g. who wins "race" to the store instruction?

Concurrent Program Correctness = all possible schedules are safe

- Must consider *every possible* permutation
- In other words…

… the scheduler is your adversary

Critical sections

What if we can designate parts of the execution as critical sections

- Rule: only one thread can be "inside"

Thread 52	Thread 205
CS enter()	CS enter()
read hits	**read hits**
addi	**addi**
write hits	**write hits**
CS exit()	CS exit()

Interrupt Disable

Q: How to implement critical section in code?

A: Lots of approaches....

Disable interrupts?

CSEnter() = disable interrupts (including clock)

CSExit() = re-enable interrupts

```
read hits
addi
write hits
```

works, but limited use

does not work well w/ multi core

Works for some kernel data-structures

Very bad idea for user code

Preemption Disable

Q: How to implement critical section in code?

A: Lots of approaches....

Modify OS scheduler?

CSEnter() = syscall to disable context switches

CSExit() = syscall to re-enable context switches

```
read hits
addi
write hits
```

Doesn't work if interrupts are part of the problem

Usually a bad idea anyway

Mutexes

Q: How to implement critical section in code?

A: Lots of approaches….

Mutual Exclusion Lock (mutex)

acquire(m): wait till it becomes free, then lock it

release(m): unlock it

```
apache_got_hit() {
    pthread_mutex_lock(m);
    hits = hits + 1;
    pthread_mutex_unlock(m)
}
```

critical section

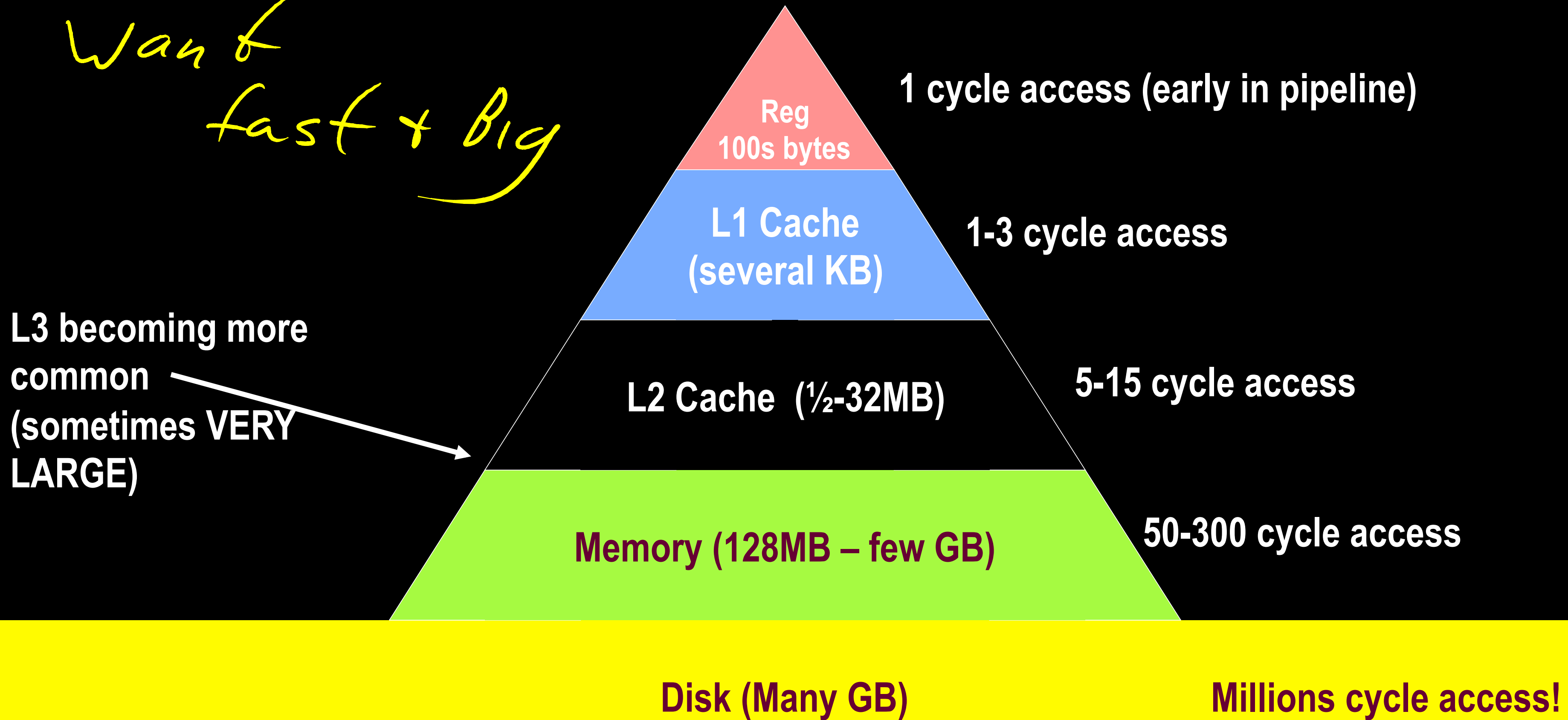
Q: How to implement mutexes?

Next time

Prelim 2 Review

Caches

Cache Design 101



These are rough numbers: mileage may vary for latest/greatest
Caches USUALLY made of SRAM

Cache *Misses*

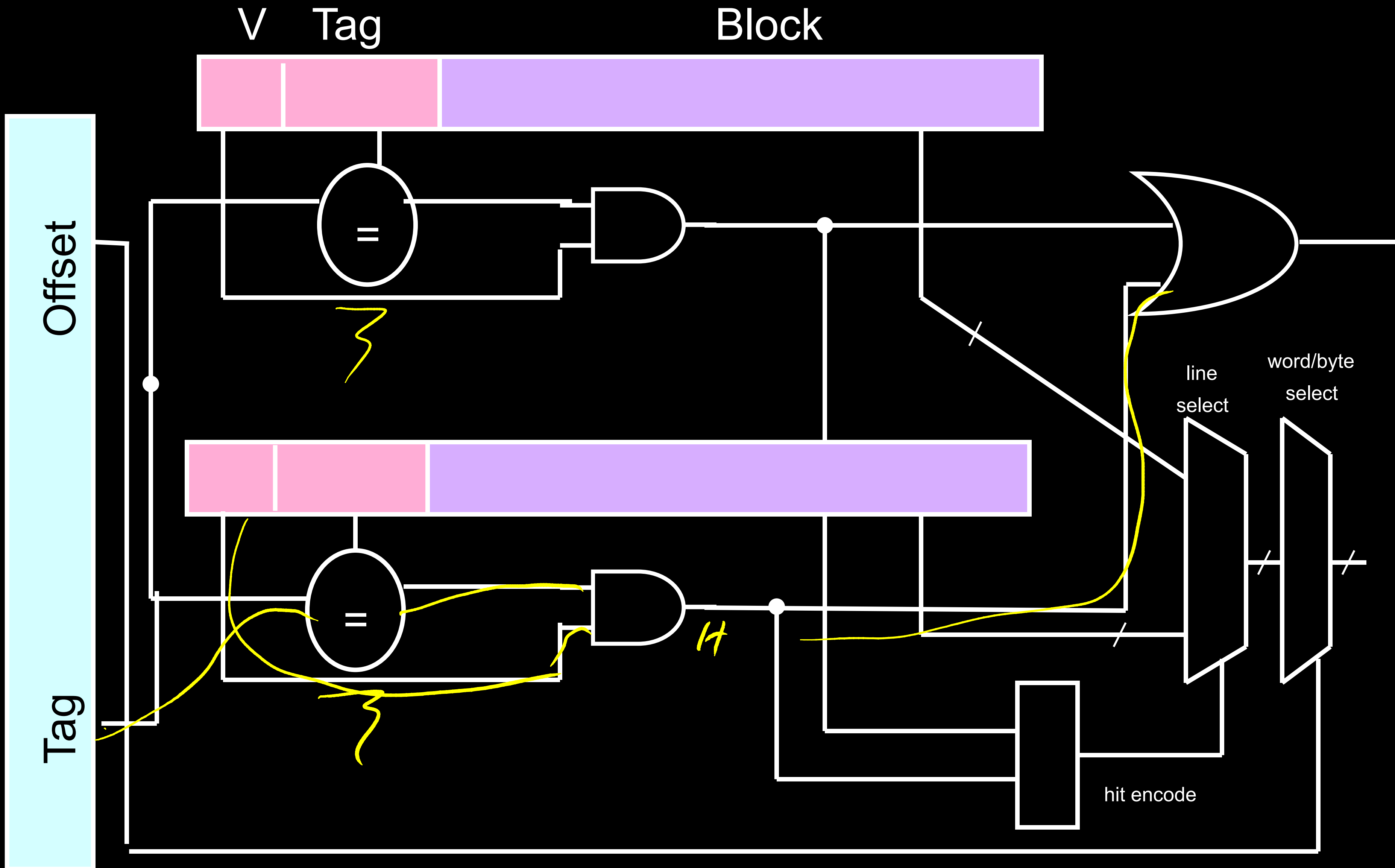
- Three types of misses
  - Cold
    - The line is being referenced for the first time
  - Capacity
    - The line was evicted because the cache was not large enough
  - Conflict
    - The line was evicted because of another access whose index conflicted

Direct Mapped Cache

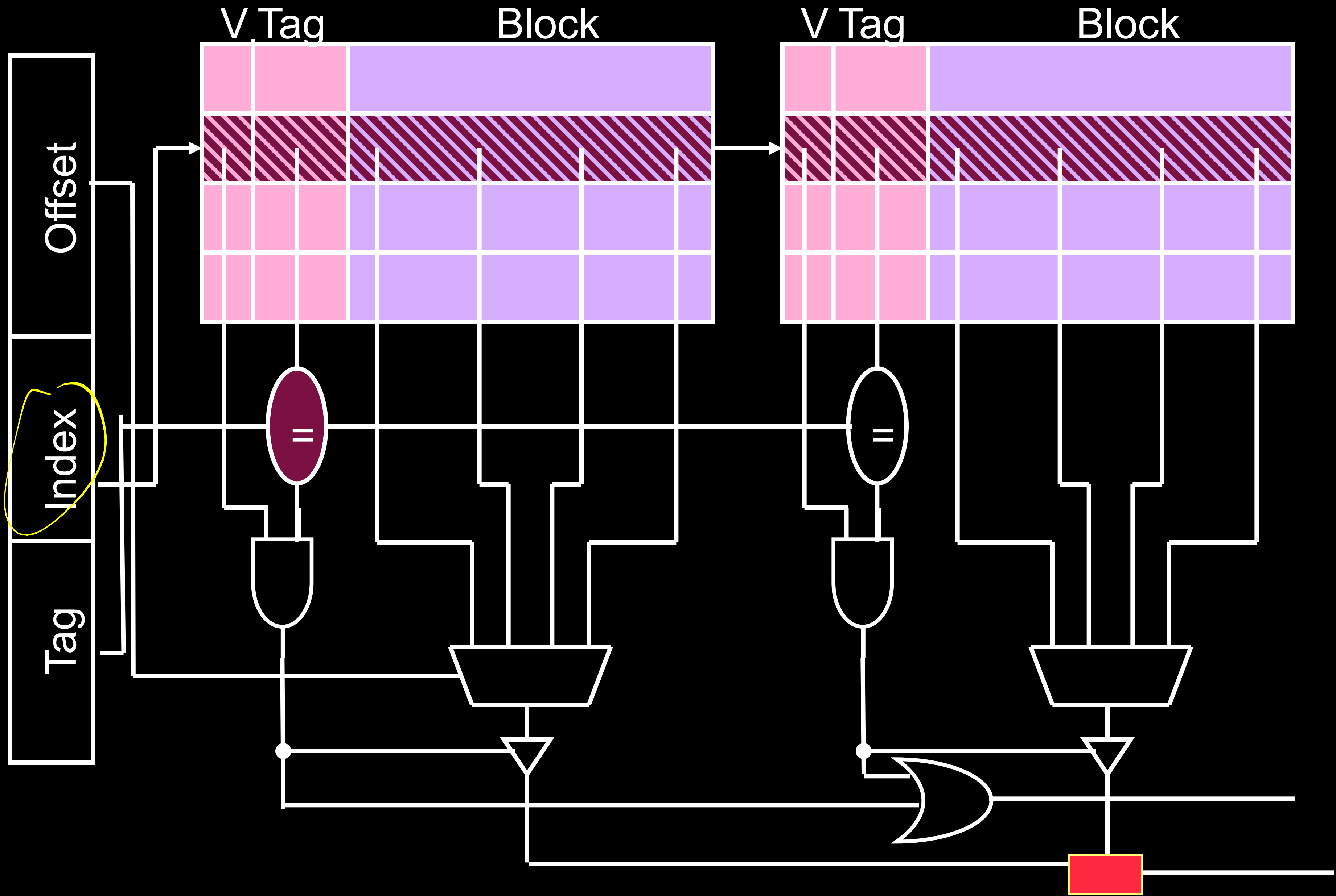
- Simplest
- Block can only be in one line in the cache

- How to determine this location?
  - Use modulo arithmetic
  - (Block address) modulo (# cache blocks)
  - For power of 2, use log (cache size in blocks)

Fully Associative Cache



2-Way Set-Associative Cache

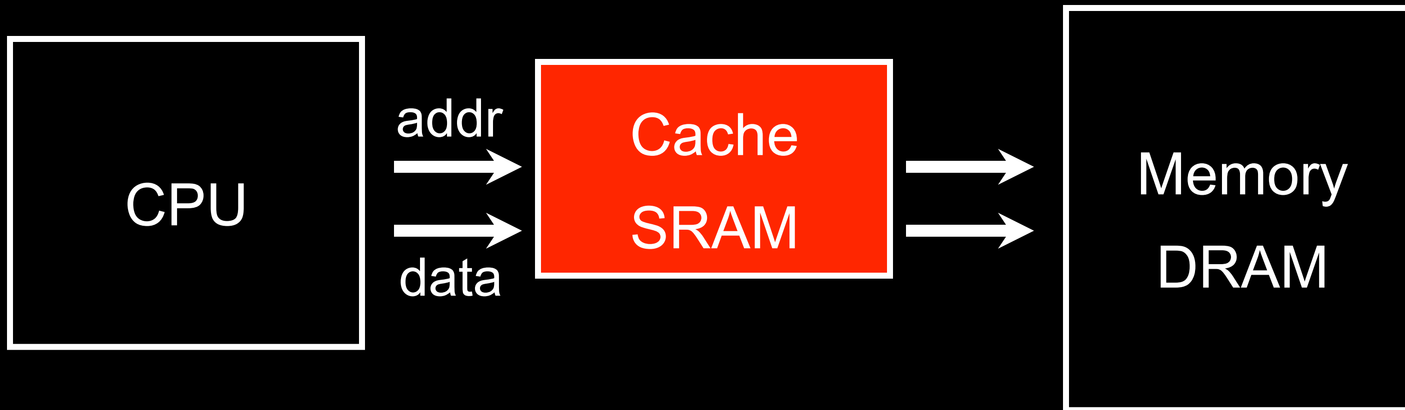


Eviction

- 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

OPT
MRU

Cache Writes

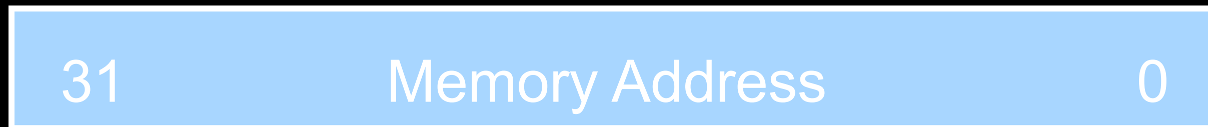


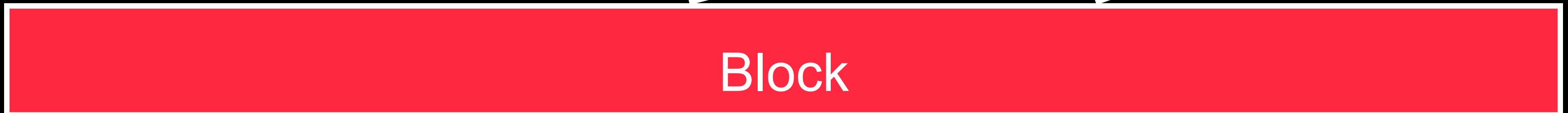
- No-Write
  - writes invalidate the cache and go to memory
- Write-Through
  - writes go to cache and to main memory
- Write-Back
  - write cache, write main memory only when block is evicted

write-buffer

Tags and Offsets

- Tag: matching
- Offset: within block
- Valid bit: is the data valid?





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

H : M

- 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 Hit/Miss Rate

- Consider processor that is 2x times faster
  - But memory is same speed
- Since AT is access time in terms of cycle time: it doubles 2x
- $H \times AT_{cache} + M \times AT_{memory}$
- Ratio of Access times, 1:100
- 99% : .99 + .01 x 100 = 1.99

Cache Hit/Miss Rate

- Original is 1GHz, 1ns is cycle time
- CPI (cycles per instruction): 1.49
- Therefore, 1.49 ns for each instruction

1.49 CPI x 1ns = 1.49 ns per inst

- New is 2GHz, 0.5 ns is cycle time.
- CPI: 1.99, 0.5ns. 0.995 ns for each instruction.

- So it doesn't go to 0.745 ns for each instruction.
- Speedup is 1.5x (not 2x)

Perf = CPI x time per inst x # inst

Cache Conscious Programming

```
int a[NCOL][NROW];
int sum = 0;

for(j = 0; j < NCOL; ++j)
        for(i = 0; i < NROW; ++i)
                sum += a[j][i];
```

1	11								
2	12								
3	13								
4	14								
5	15								
6									
7									
8									
9									
10									

- Every access is a cache miss!

Cache Conscious Programming

```
int a[NCOL][NROW];
int sum = 0;

for(i = 0; i < NROW; ++i)
        for(j = 0; j < NCOL; ++j)
                sum += a[j][i];
```

- Same program, trivial transformation, 3 out of four accesses hit in the cache

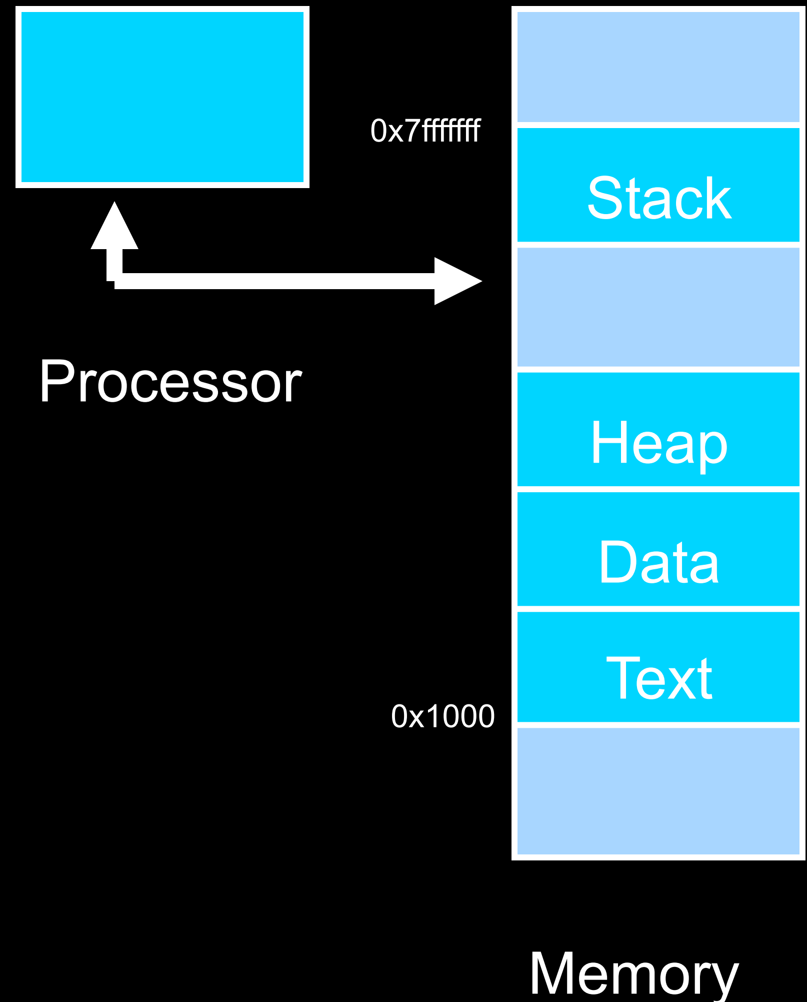
Can answer the question…..

- A: for i = 0 to 99
  - for j = 0 to 999
    - A[i][j] = complexComputation ()
- B: for j = 0 to 999
  - for i = 0 to 99
    - A[i][j] = complexComputation ()
- Why is B 15 times slower than A?

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

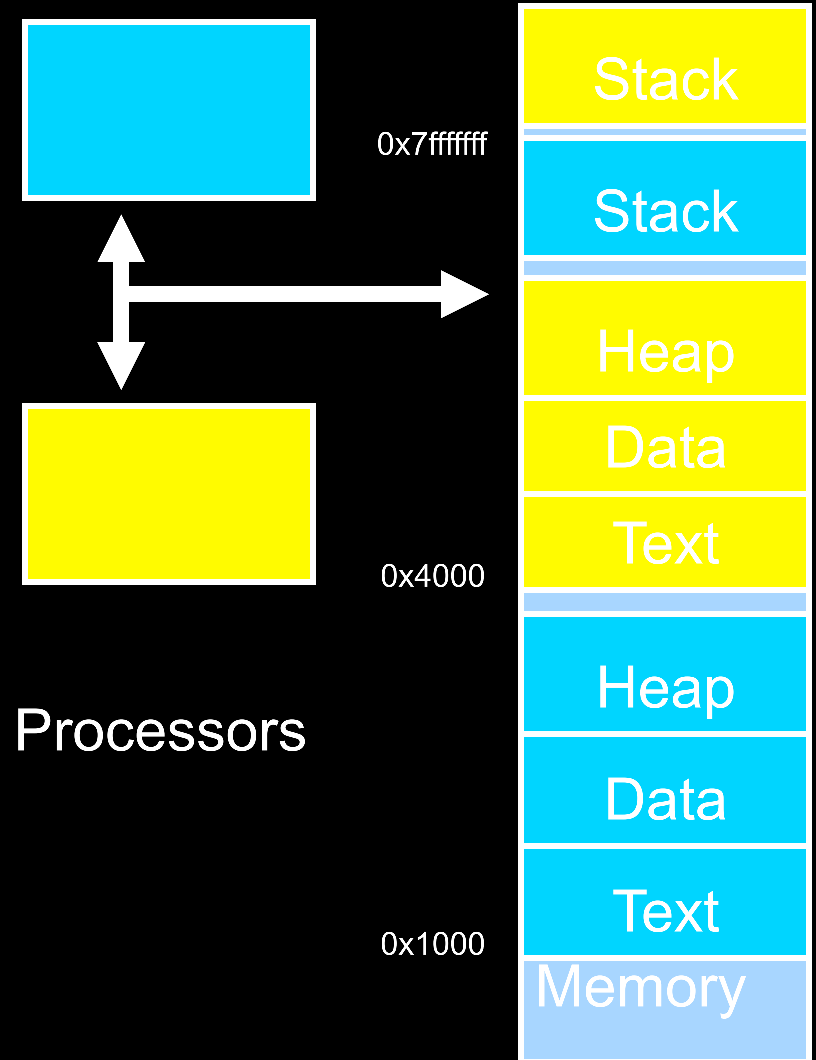
Processor & Memory

- Currently, the processor's address lines are directly routed via the system bus to the memory banks
  - Simple, fast
- What happens when the program issues a load or store to an invalid location?
  - e.g. 0x000000000 ?
  - uninitialized pointer



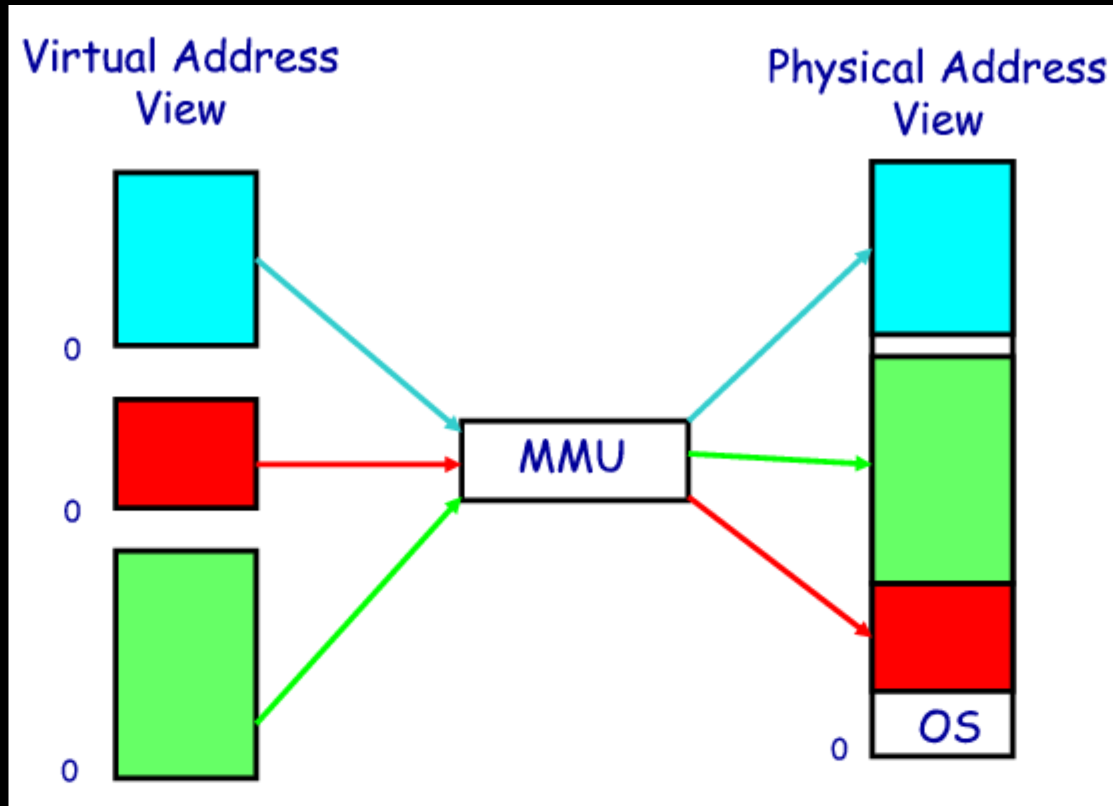
Physical Addressing Problems

- What happens when another program is executed concurrently on another processor?
  - The addresses will conflict
- We could try to relocate the second program to another location
  - Assuming there is one
  - Introduces more problems!



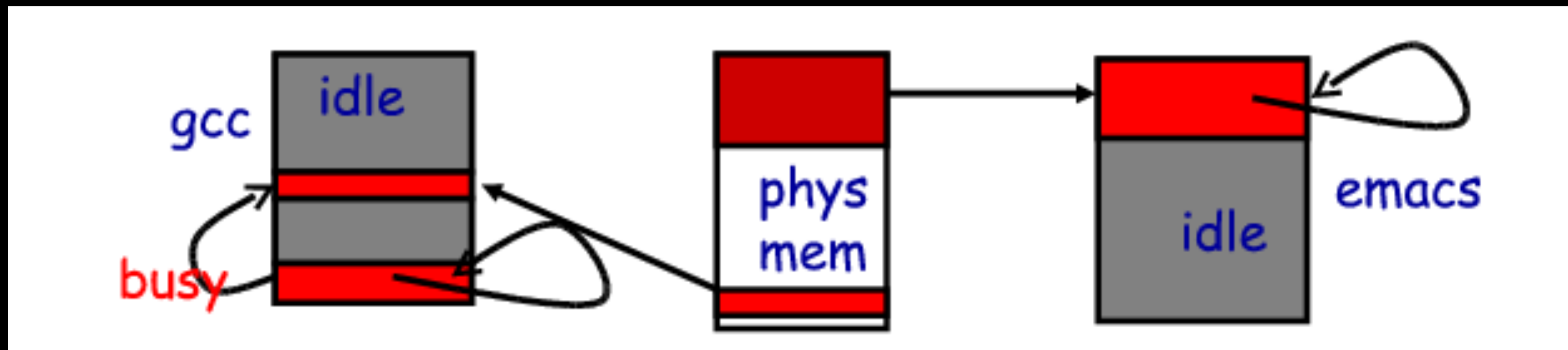
Address Space

- Memory Management Unit (MMU)
  - Combination of hardware and software



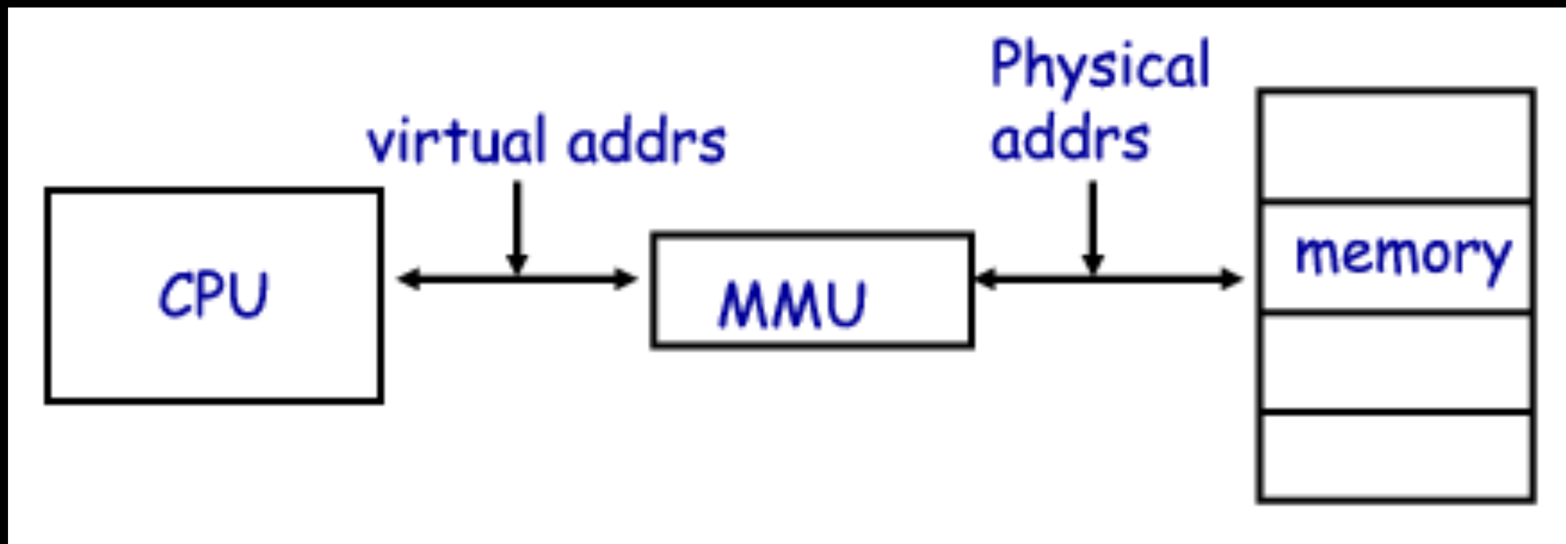
Virtual Memory Advantages

- Can relocate program while running
- Virtualization
  - In CPU: if process is not doing anything, switch
  - In memory: when not using it, somebody else can use it



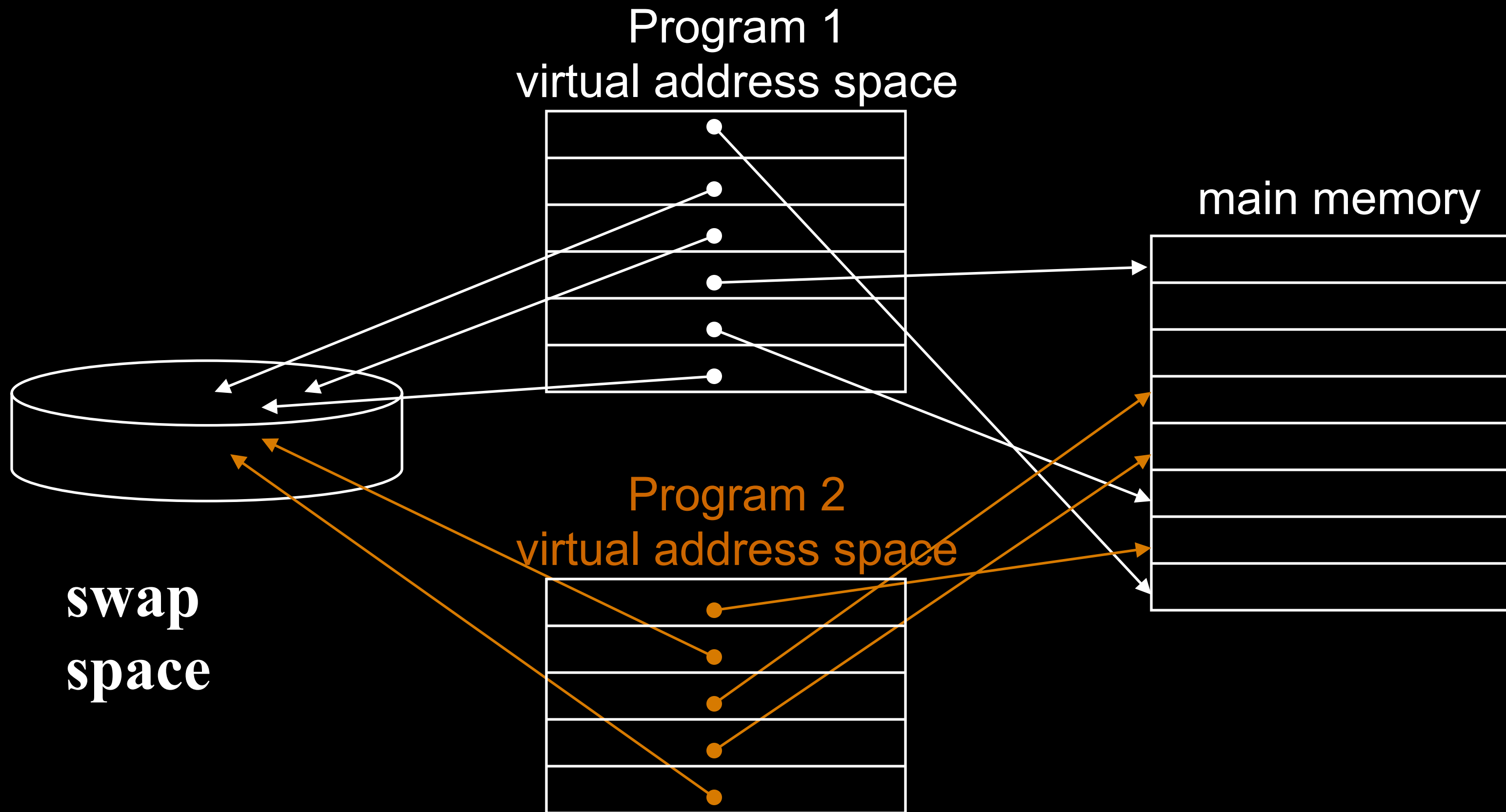
How to make it work?

- Challenge: Virtual Memory can be slow!
- At run-time: virtual address must be *translated* to a physical address
- MMU (combination of hardware and software)

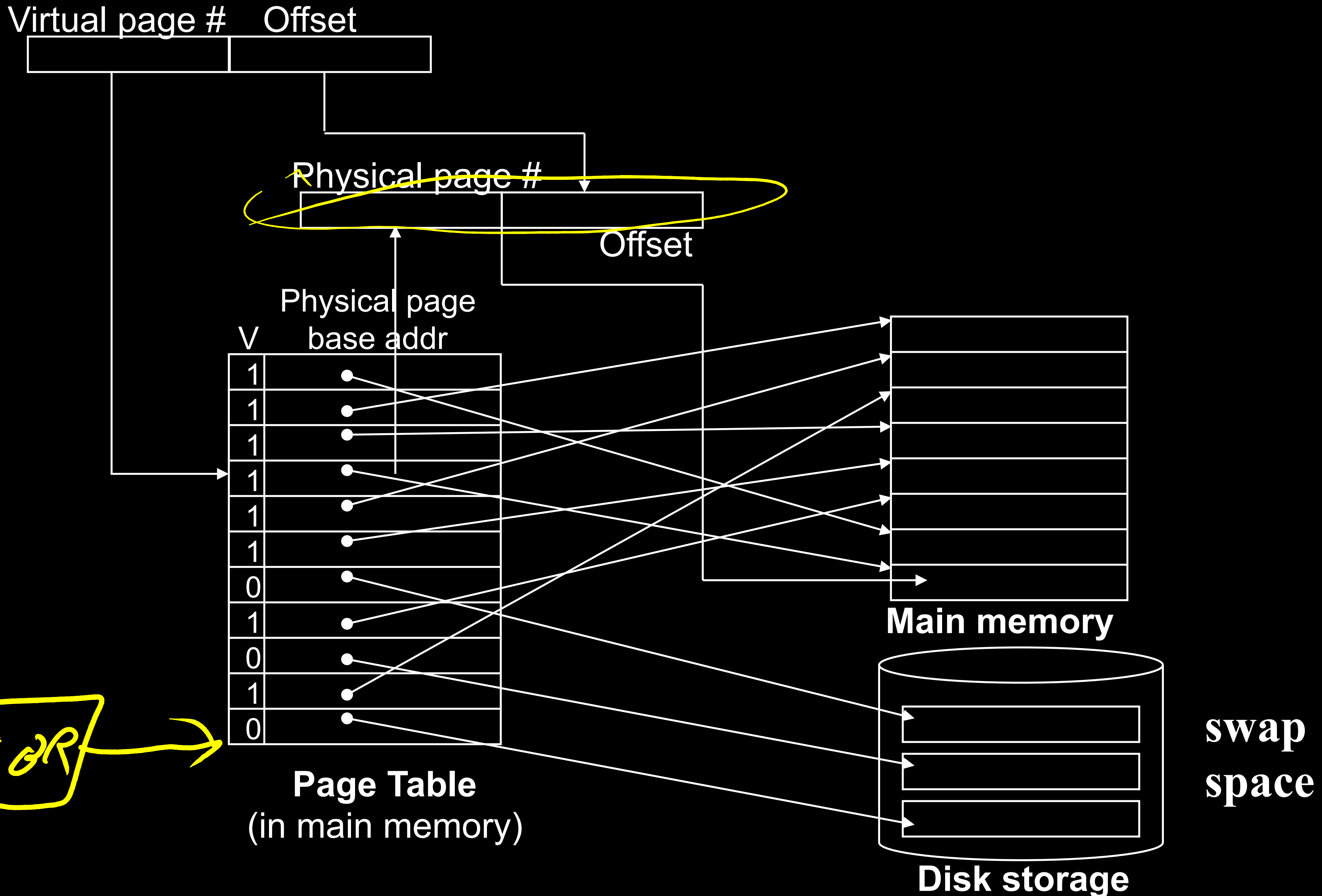


Two Programs Sharing Physical Memory

- The starting location of each page (either in main memory or in secondary memory) is contained in the program's page table

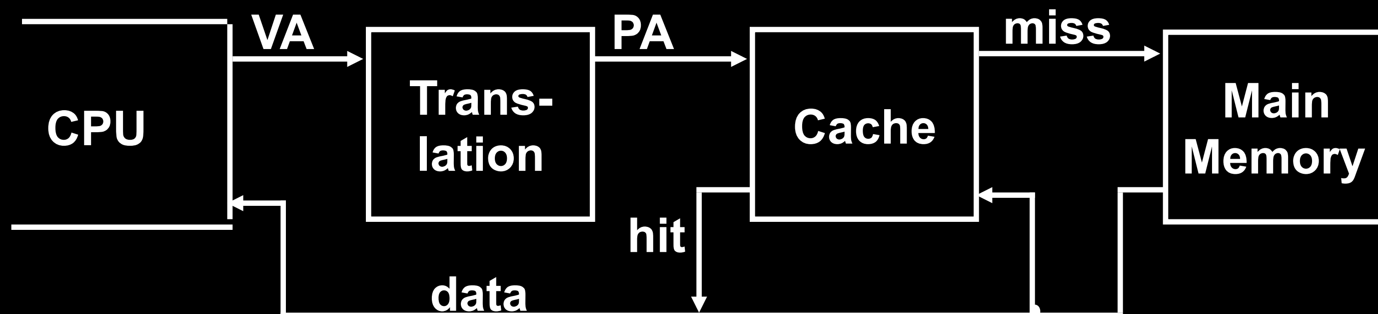


Page Table for Translation



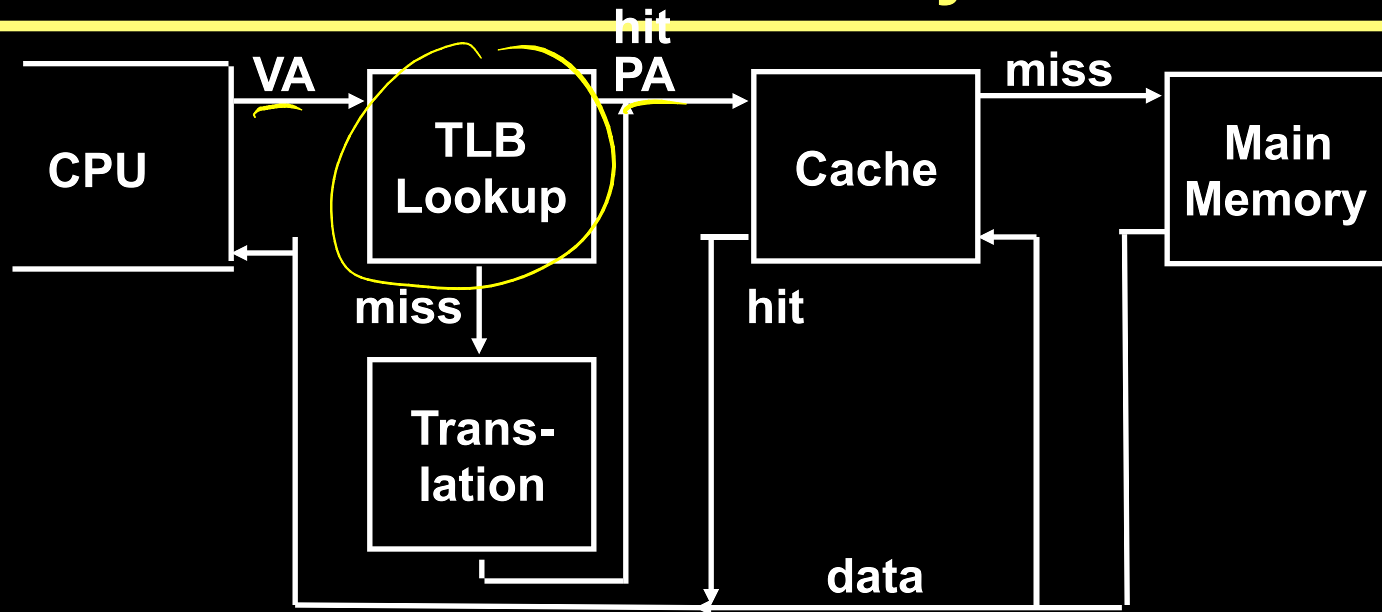
Virtual Addressing with a Cache

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



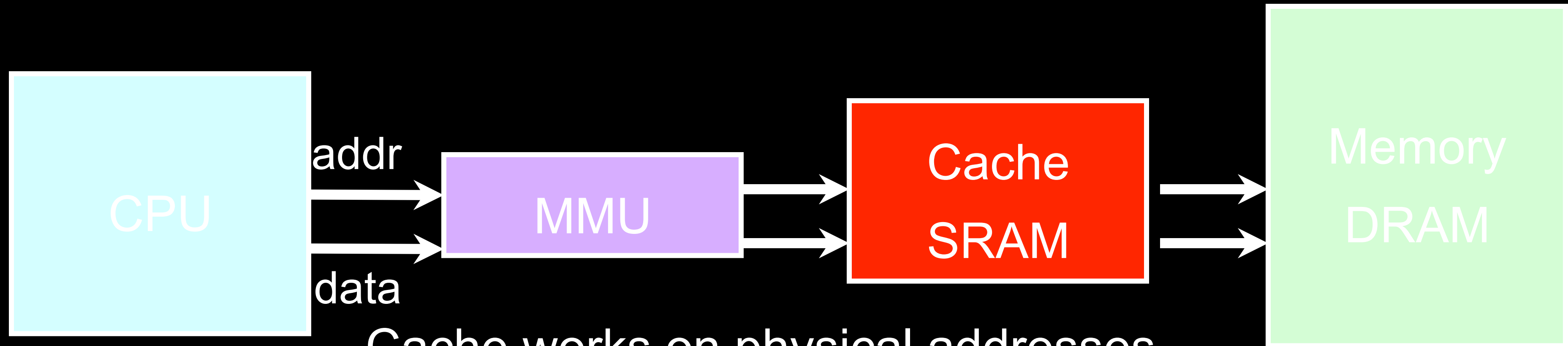
- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)

A TLB in the Memory Hierarchy

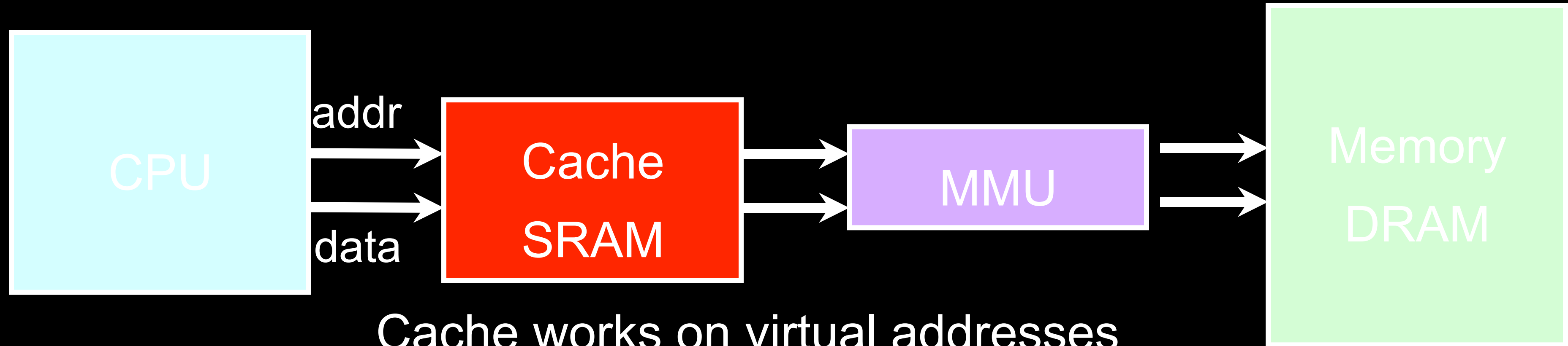


- A TLB miss:
  – If the page is not in main memory, then it's a true page fault
    - Takes 1,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults

Virtual vs. Physical Caches



Cache works on physical addresses

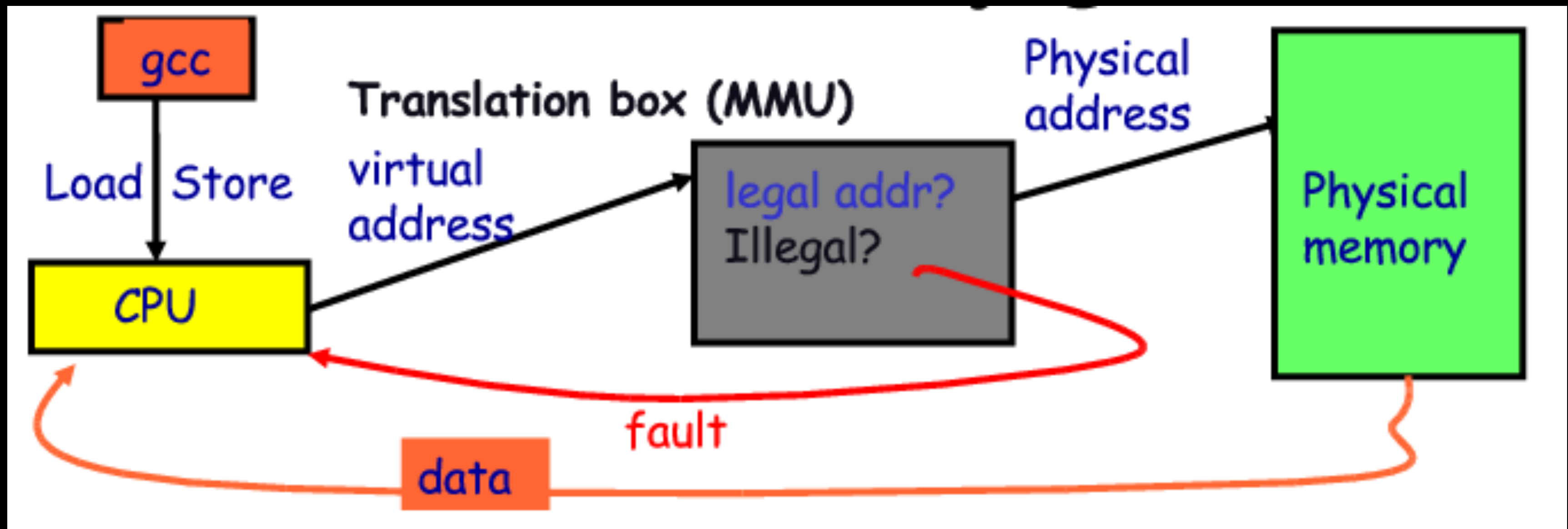


Cache works on virtual addresses

- L1 (on-chip) caches are typically virtual *addr'd (but phys tag)*
- L2 (off-chip) caches are typically physical

Address Translation

- Translation is done through the page table
  – A virtual memory miss (i.e., when the page is not in physical memory) is called a page fault

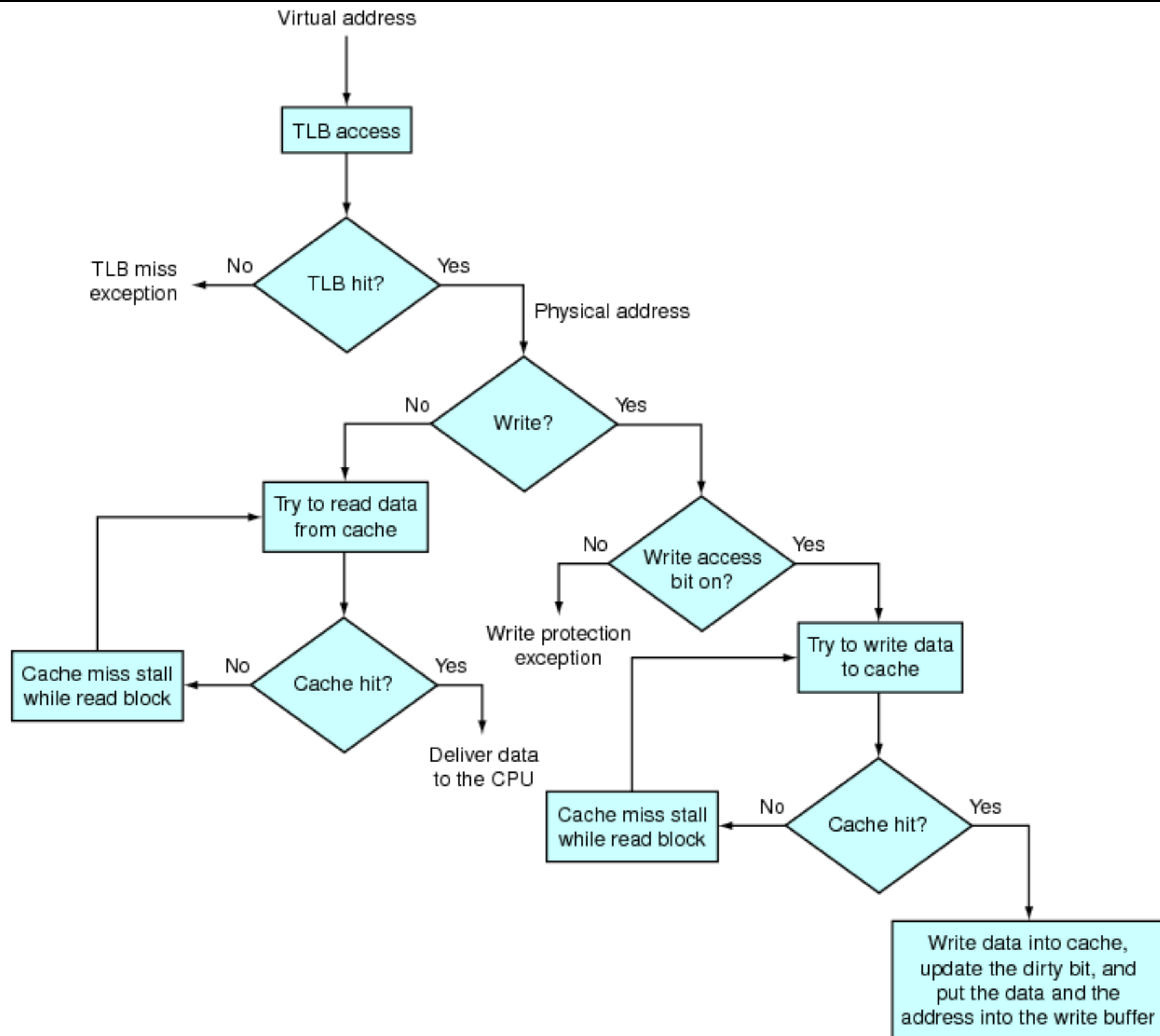


Hardware/Software Boundary

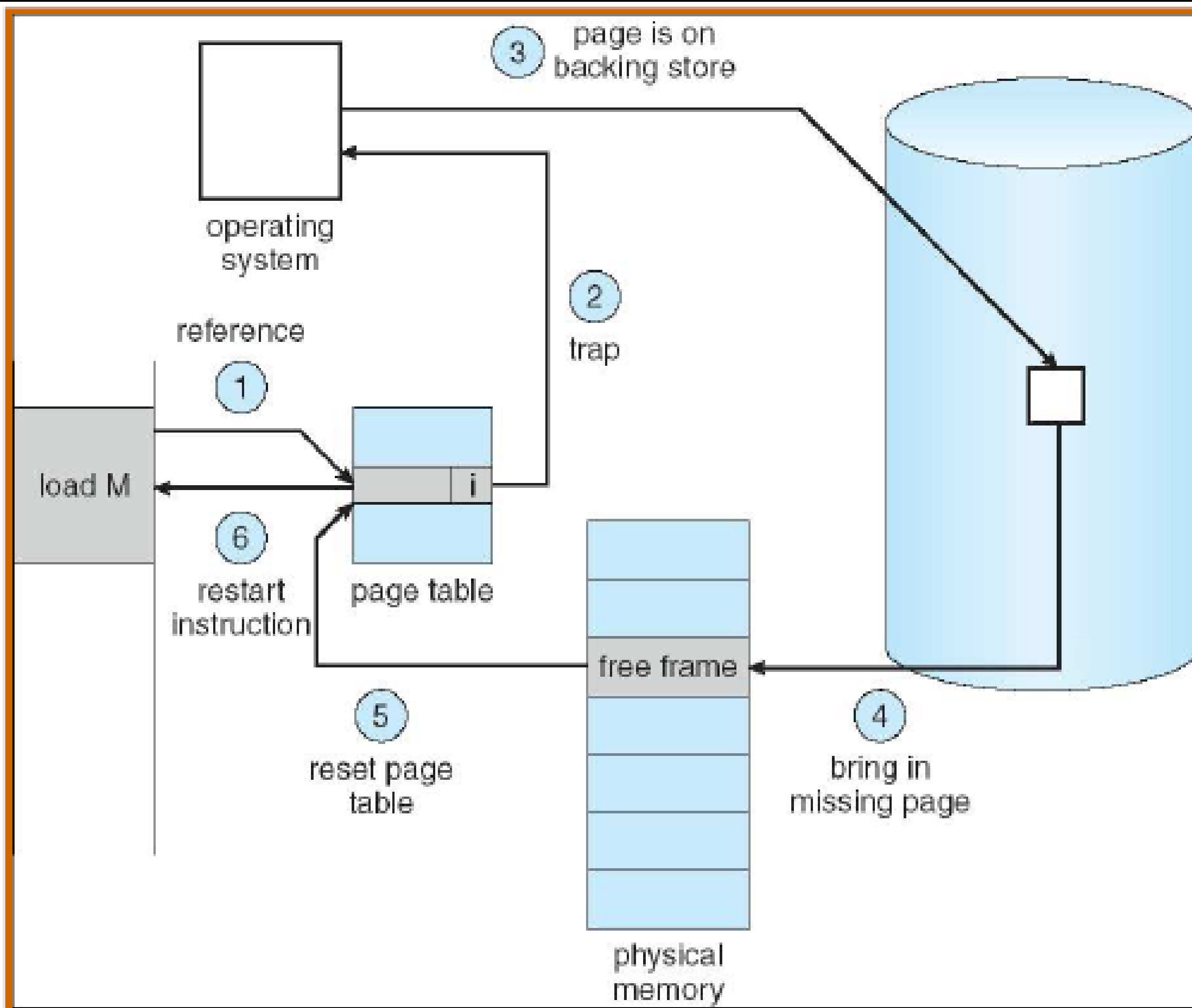
- Virtual to physical address translation is assisted by hardware?
  - Translation Lookaside Buffer (TLB) that caches the recent translations
    - TLB access time is part of the cache hit time
    - May allot an extra stage in the pipeline for TLB access
  - TLB miss
    - Can be in software (kernel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

Hardware/Software Boundary

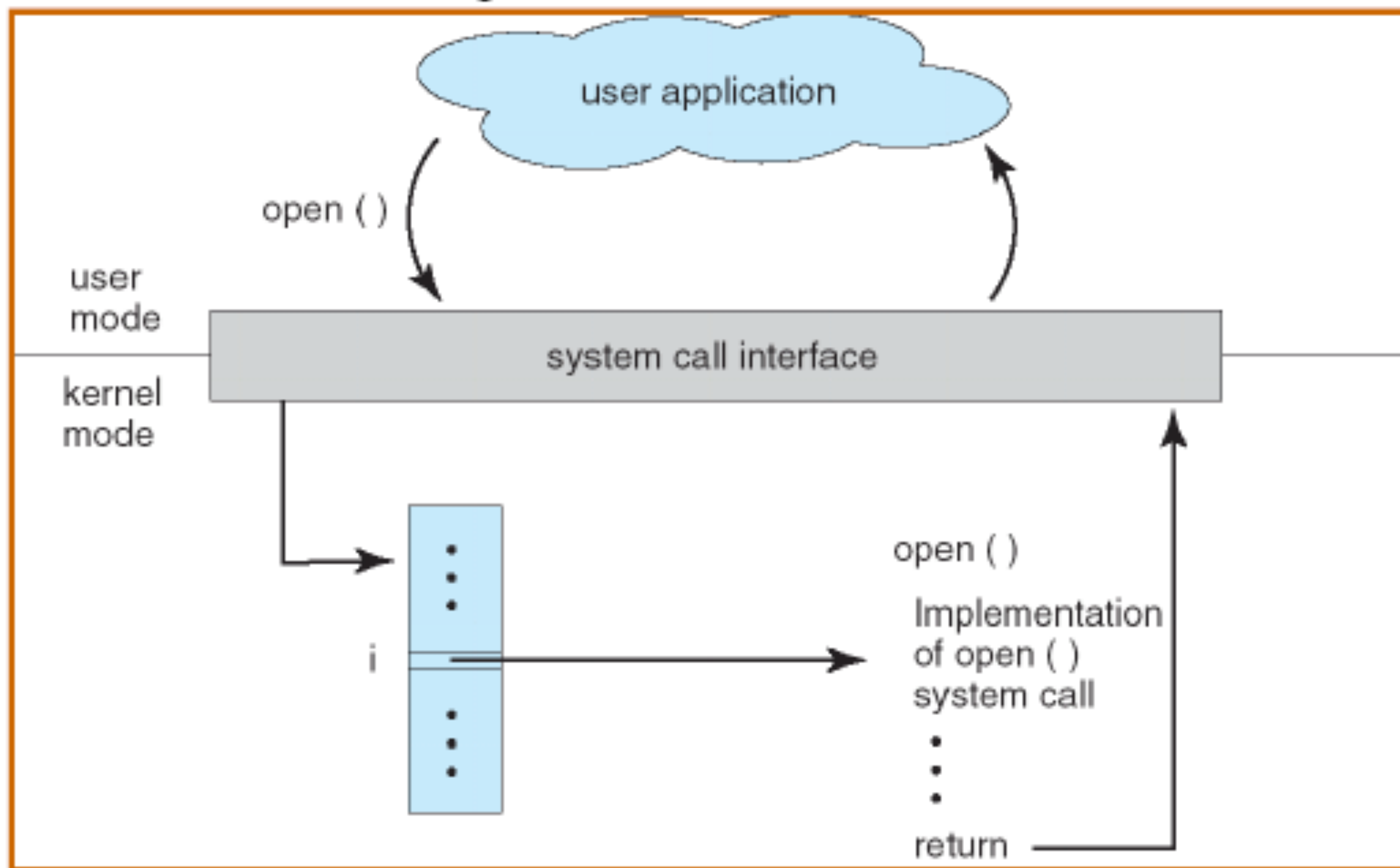
- Virtual to physical address translation is assisted by hardware?
  - Page table storage, fault detection and updating
    - Page faults result in interrupts (precise) that are then handled by the OS
    - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the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


Virtual address
TLB access
TLB hit?
No
Yes
TLB miss
exception
Physical address
Write?
No
Yes
Try to read data
from cache
Write access
bit on?
No
Yes
Write protection
exception
Try to write data
to cache
Cache miss stall
while read block
Cache hit?
No
Yes
Deliver data
to the CPU
Cache miss stall
while read block
Cache hit?
No
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


user application
open ( )
user
mode
system call interface
kernel
mode
i
open ( )
Implementation
of open ( )
system call
return

Exceptions

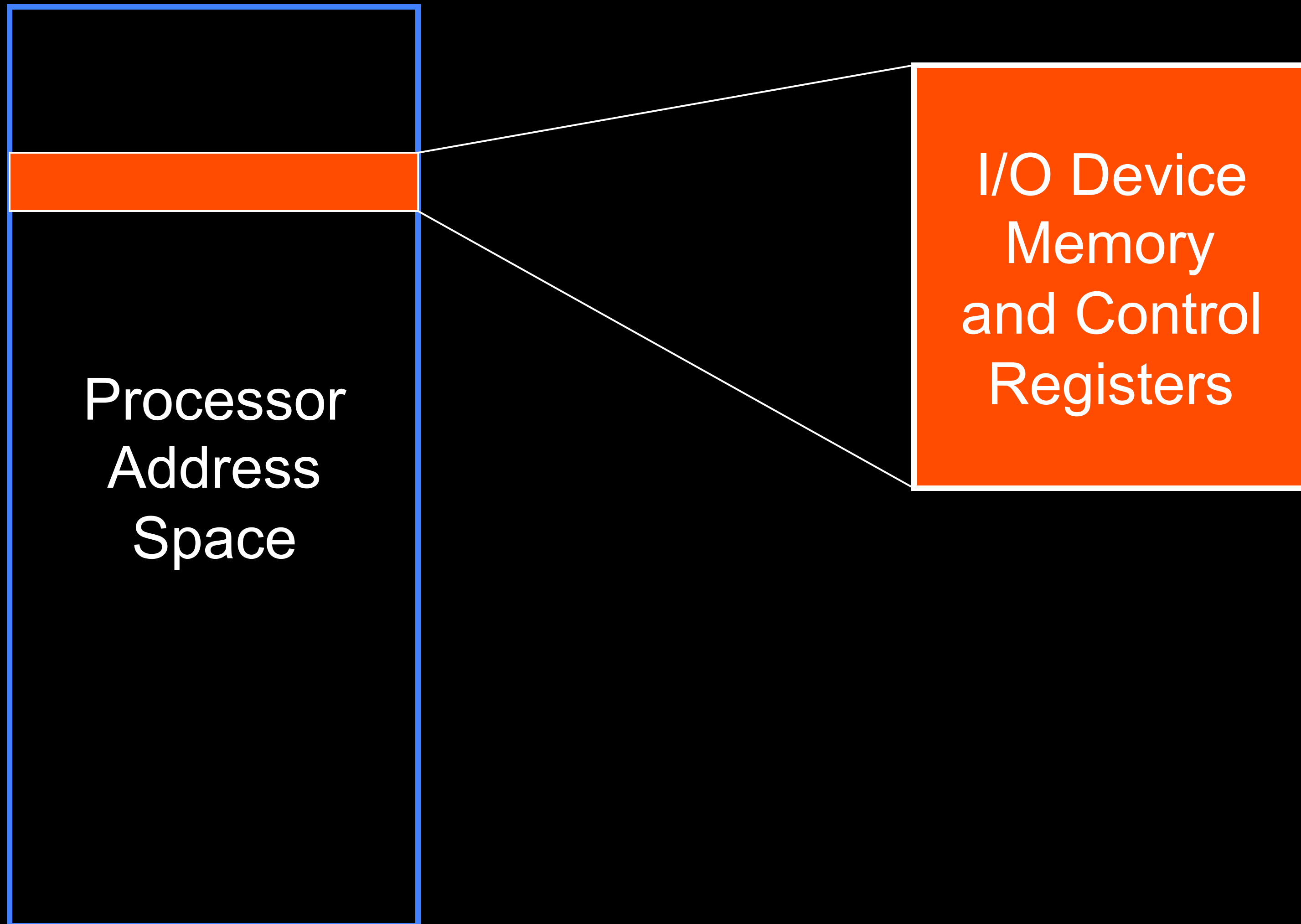
- System calls are control transfers to the OS, performed under the control of the user program
- Sometimes, need to transfer control to the OS at a time when the user program least expects it
  - Division by zero,
  - Alert from power supply that electricity is going out,
  - Alert from network device that a packet just arrived,
  - Clock notifying the processor that clock just ticked
- Some of these causes for interruption of execution have nothing to do with the user application
- Need a (slightly) different mechanism, that allows resuming the user application

Terminology

- Trap
  - Any kind of a control transfer to the OS
- Syscall
  - Synchronous, program-initiated control transfer from user to the OS to obtain service from the OS
  - e.g. SYSCALL
- Exception
  - Synchronous, program-initiated control transfer from user to the OS in response to an exceptional event
  - e.g. Divide by zero
- Interrupt
  - Asynchronous, device-initiated control transfer from user to the OS
  - e.g. Clock tick, network packet

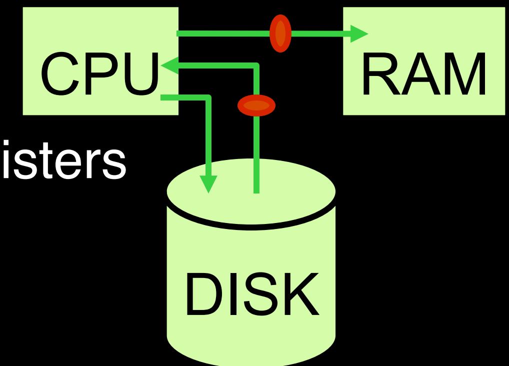
- I/O and DMA

Memory-Mapped I/O

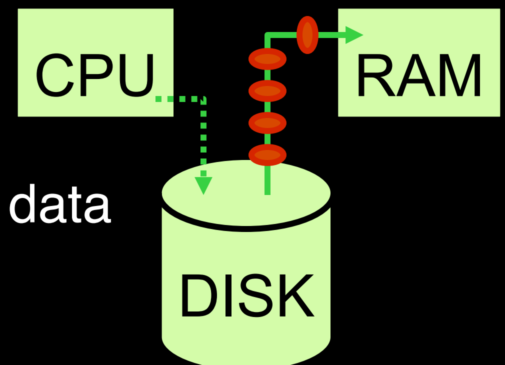


DMA: Direct Memory Access

- Non-DMA transfer: I/O device ←→ CPU ←→ RAM
  - **for (i = 1 .. n)**
    - CPU sends transfer request to device
    - I/O writes data to bus, CPU reads into registers
    - CPU writes data to registers to memory



- DMA transfer: I/O device ←→ RAM
  - CPU sets up DMA request on device
  - for (i = 1 .. n)
    - I/O device writes data to bus, RAM reads data



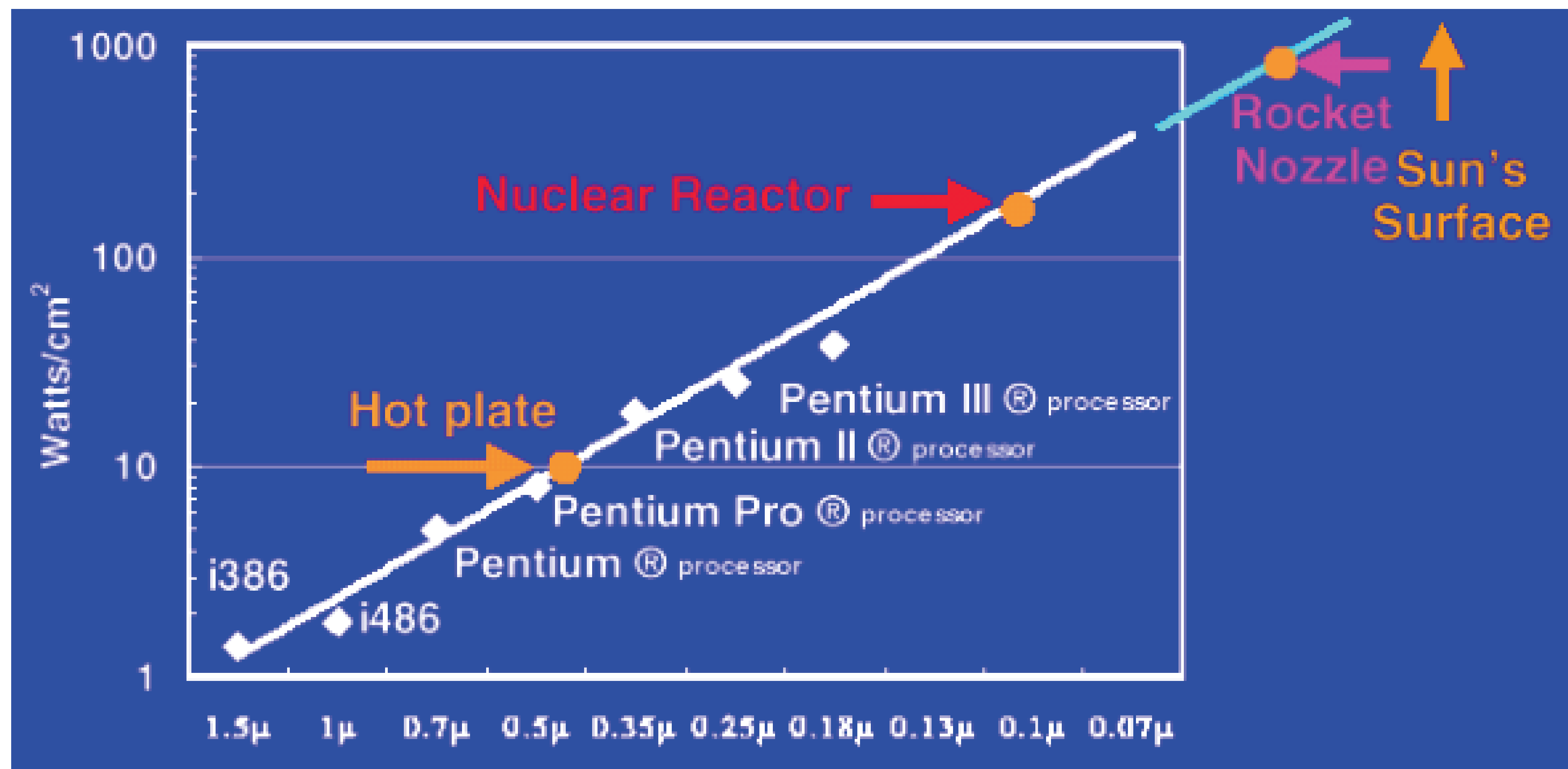
- Multicore and Synchronization

Why Multicore?

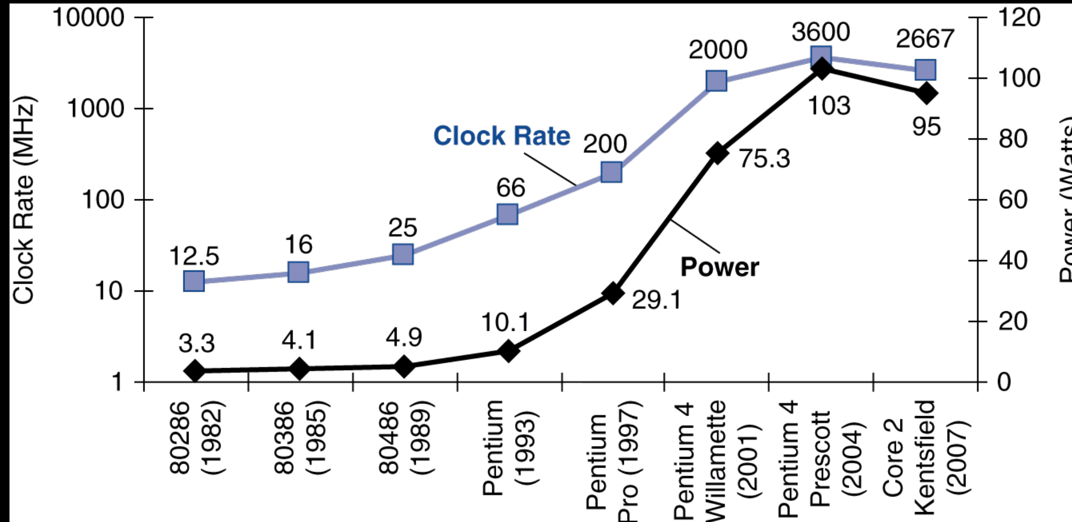
- Moore's law
  - A law about transistors
    (Not speed)
  - Smaller means faster transistors

- Power consumption growing with transistors

Power Limits Performance



Power Trends



- In CMOS IC technology

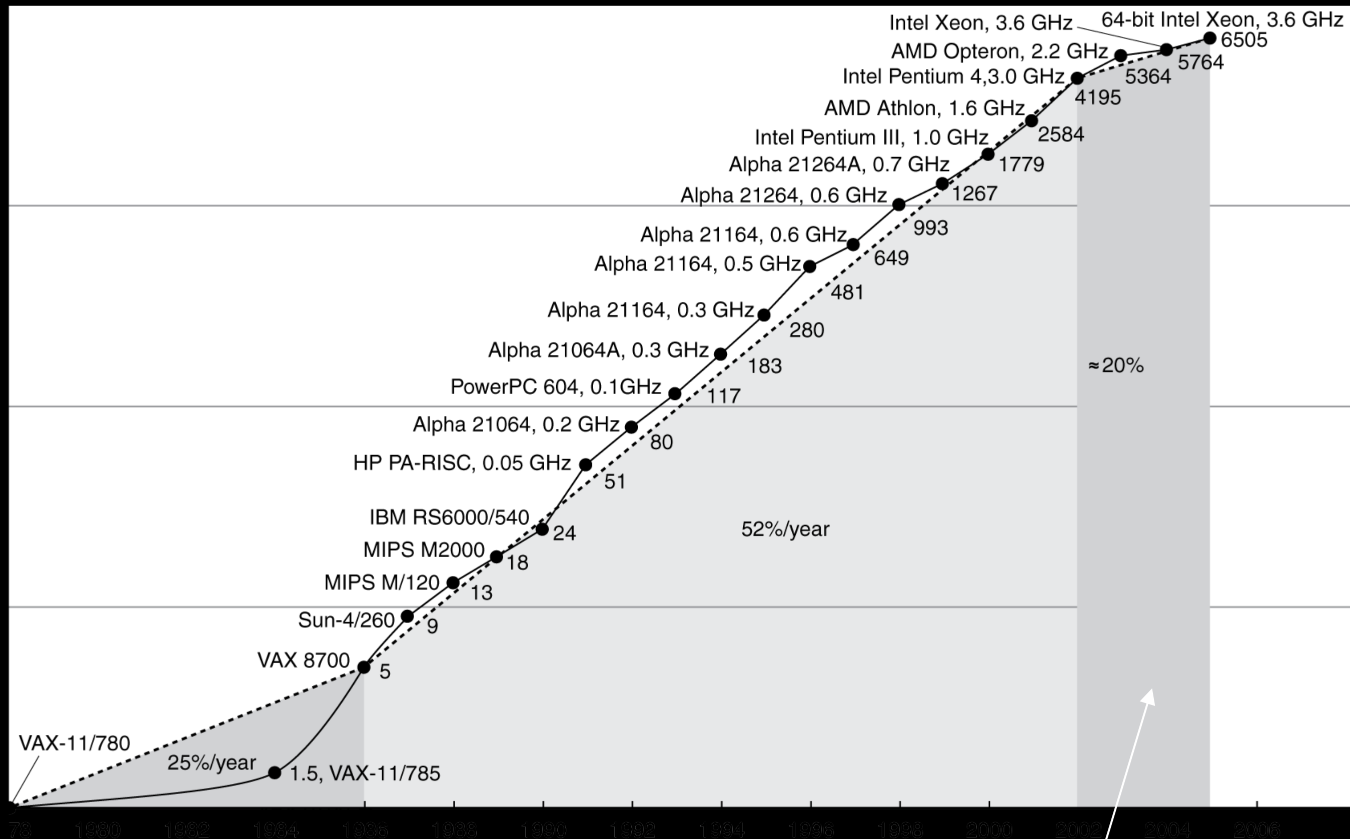
$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30

5V → 1V

×1000

Uniprocessor Performance



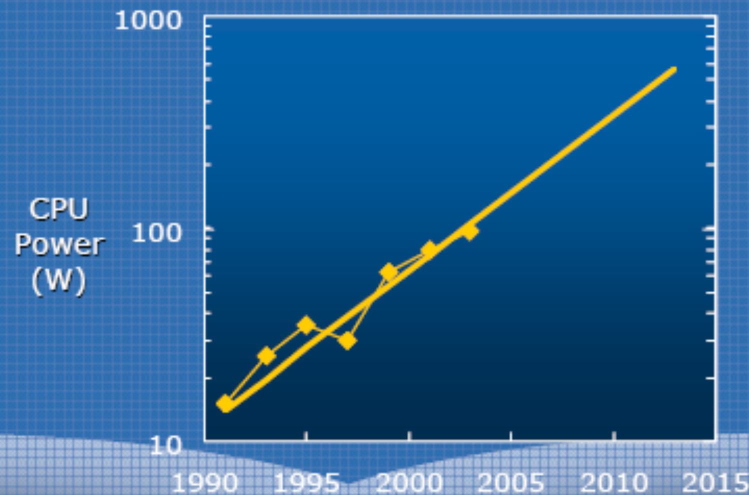
Constrained by power, instruction-level parallelism, memory latency

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?

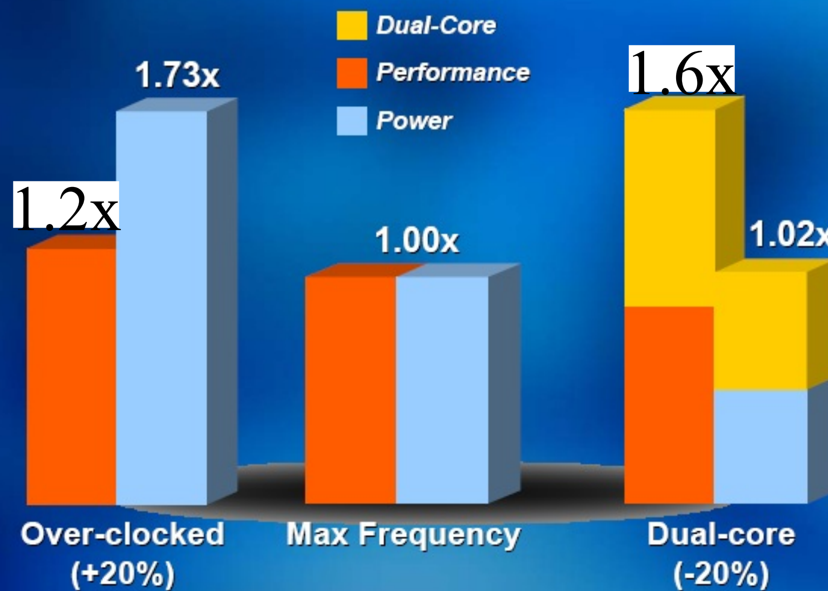
Intel's argument

Power Limitations



Power = Capacitance x Voltage² x Frequency
also
Power ~ Voltage³

Multi-Core
Energy-Efficient Performance



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

$$\text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E}$$

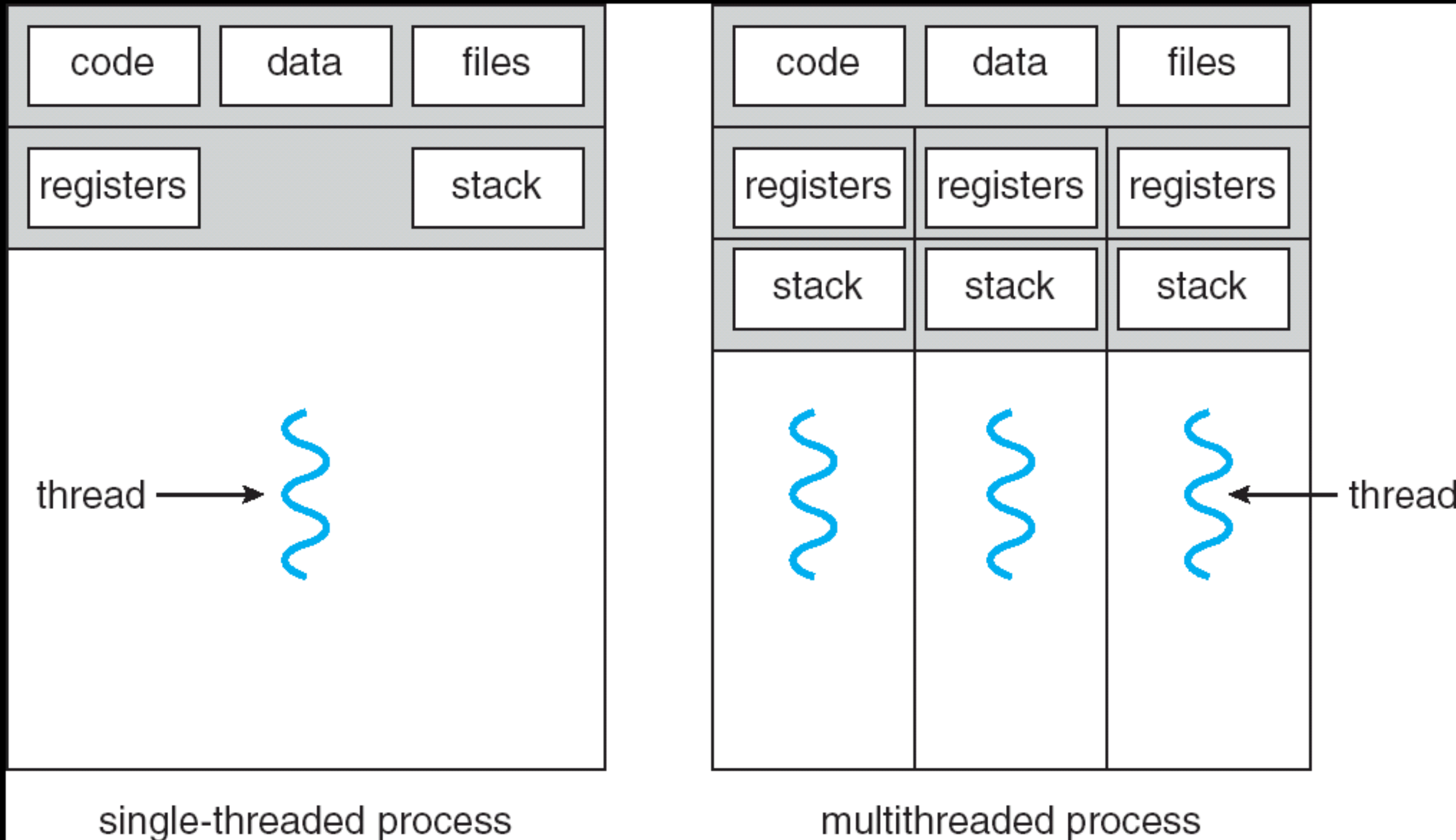
Amdahl’s Law

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

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

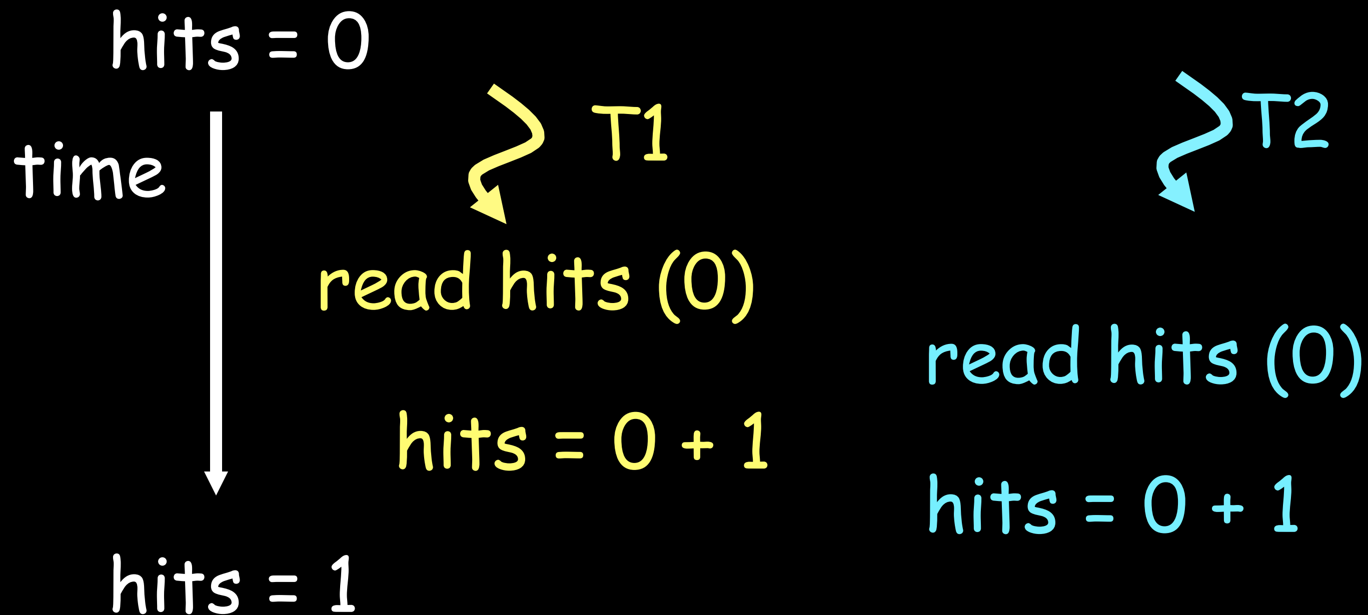
$$\text{Speedup (with } E) = \frac{1}{(1 - F) + F/S}$$

Multithreaded Processes



Shared counters

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



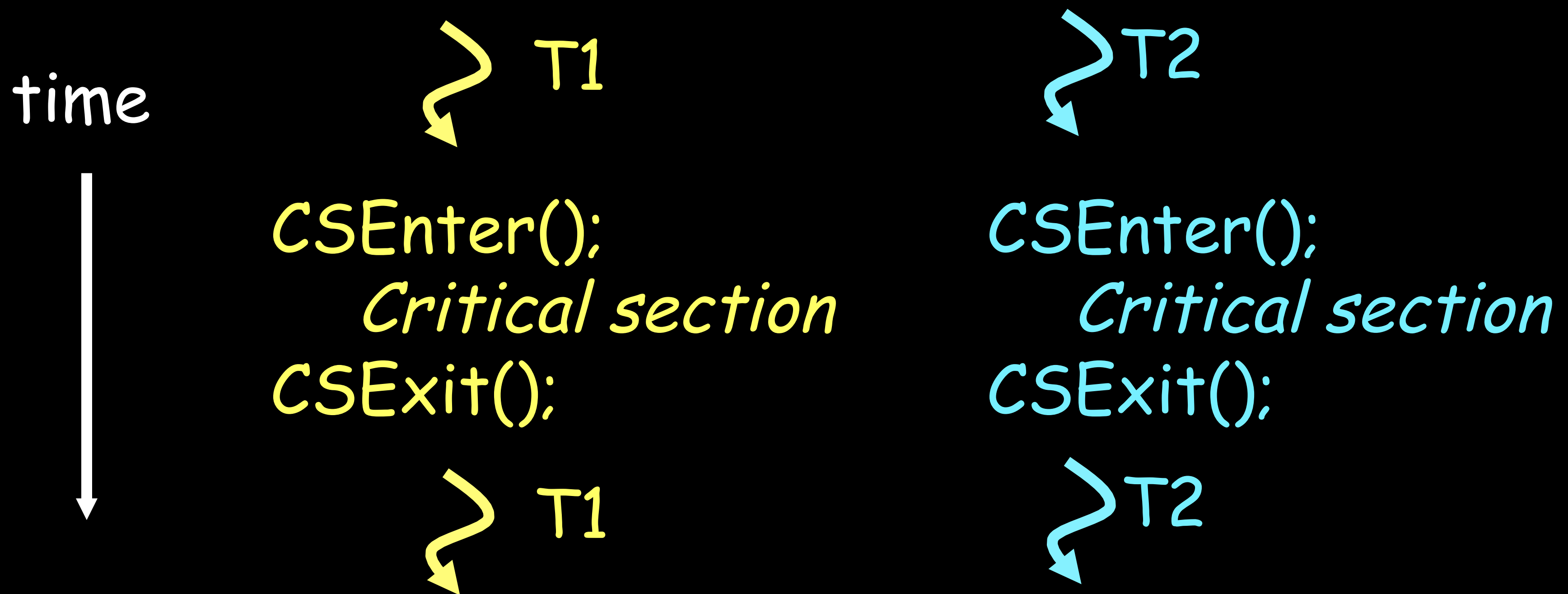
- Occasional timing-dependent failure ⇒ Difficult to debug
- Called a *race condition*

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



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

```
pthread_mutex_lock(m);
    hits = hits+1;
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 on Thursday

Good Luck!