Caches IV, System Control Flow

CSE 351 Autumn 2018

Instructor:

Justin Hsia

Teaching Assistants:

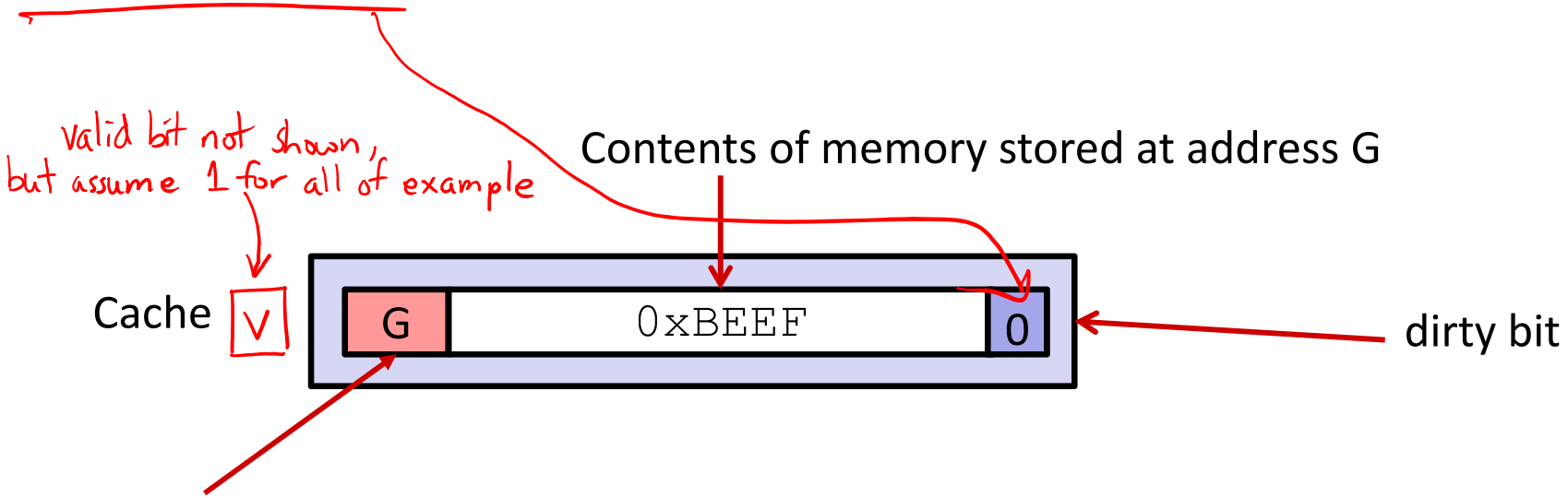
Akshat Aggarwal
Brian Dai
Kevin Bi
Sophie Tian
An Wang
Britt Henderson
Kory Watson
Teagan Horkan
Andrew Hu
James Shin
Riley Germundson

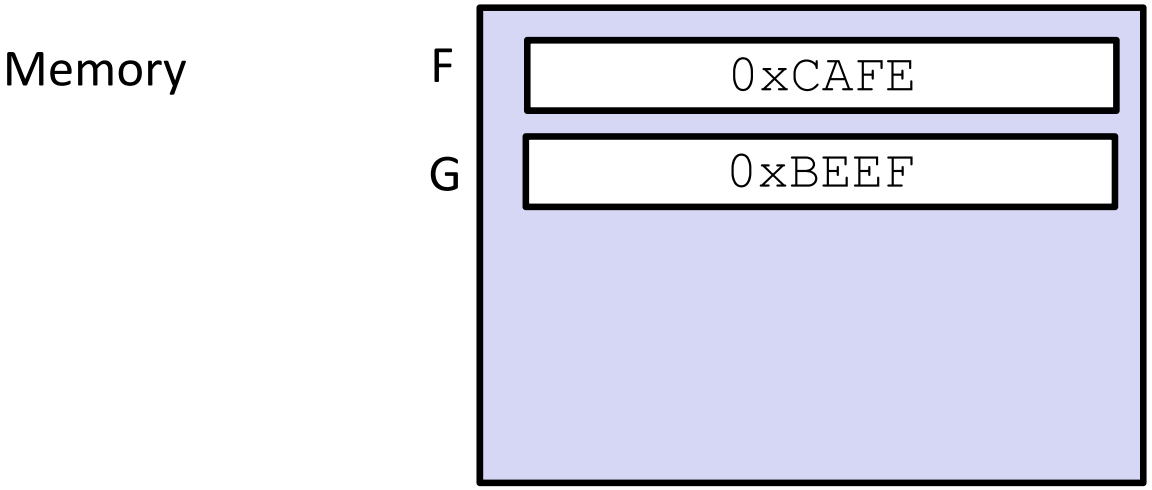
http://xkcd.com/908/

Administrivia

- Homework 4 due Friday (11/16)
- Lab 4 released over the long weekend
  - Cache parameter puzzles and code optimizations

Write-back, write-allocate example





In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.

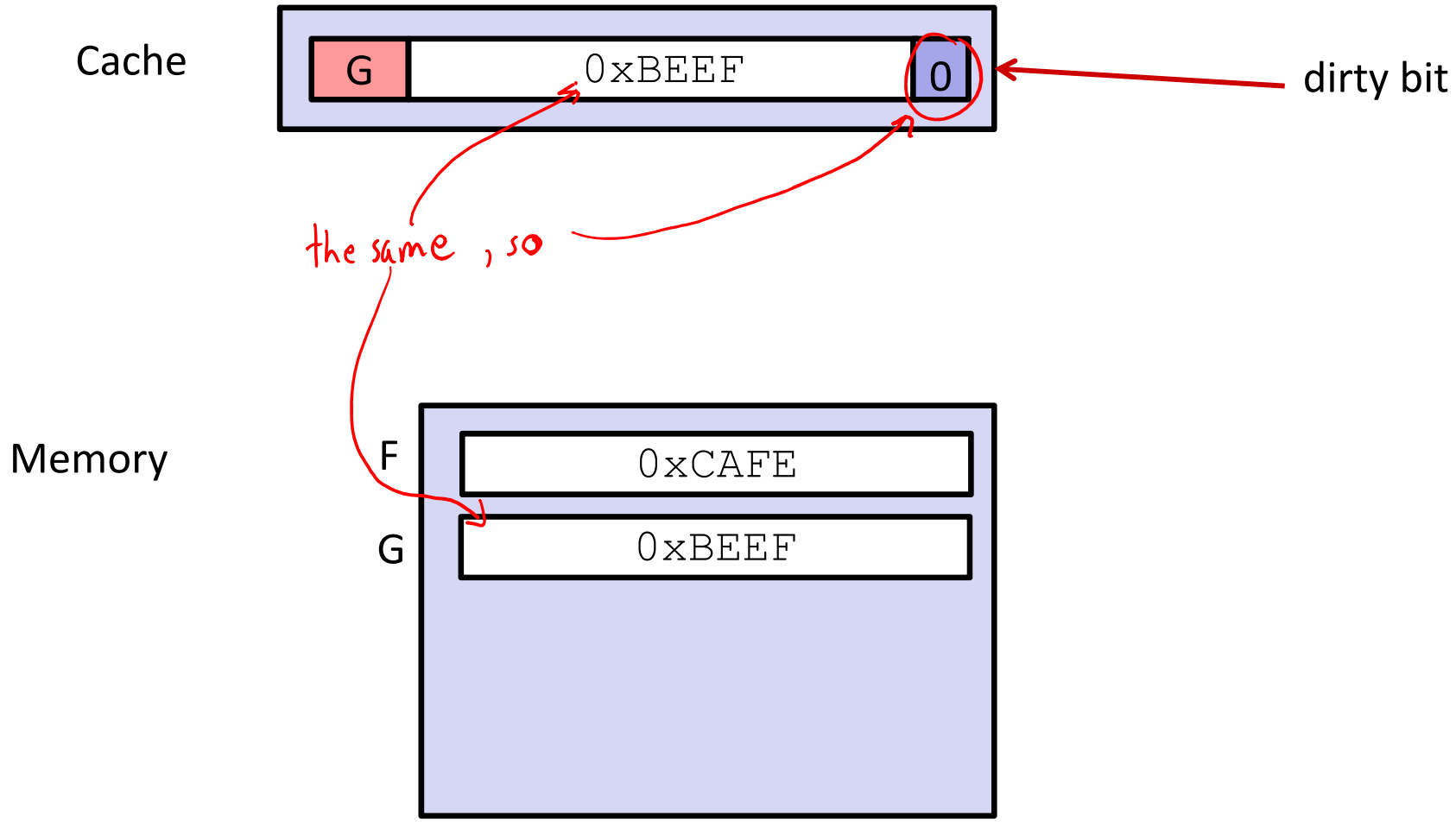
Write-back, write-allocate example

write miss

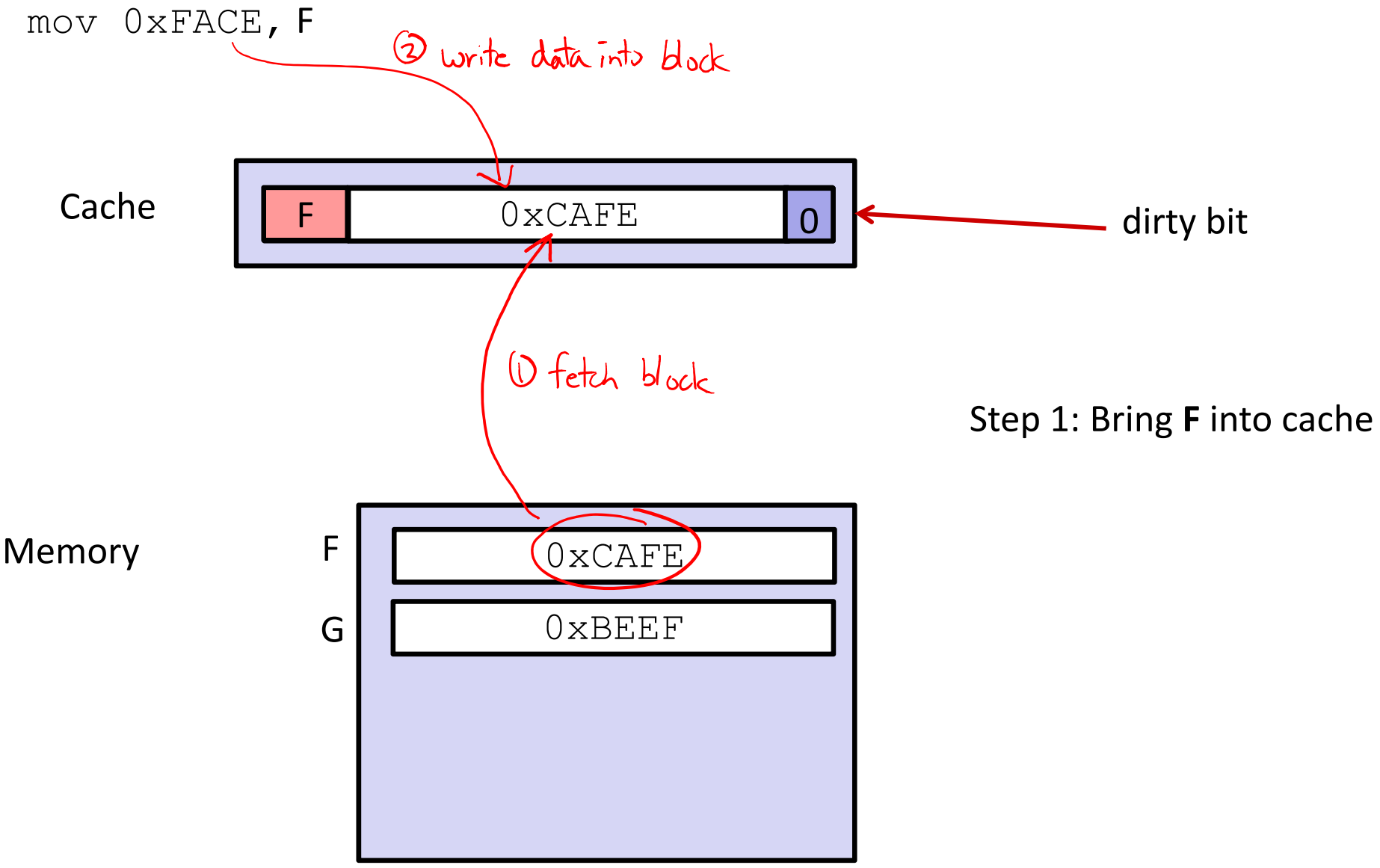
```
mov 0xFACE, F
```

① check cache for F → miss

② pull block into $, then write



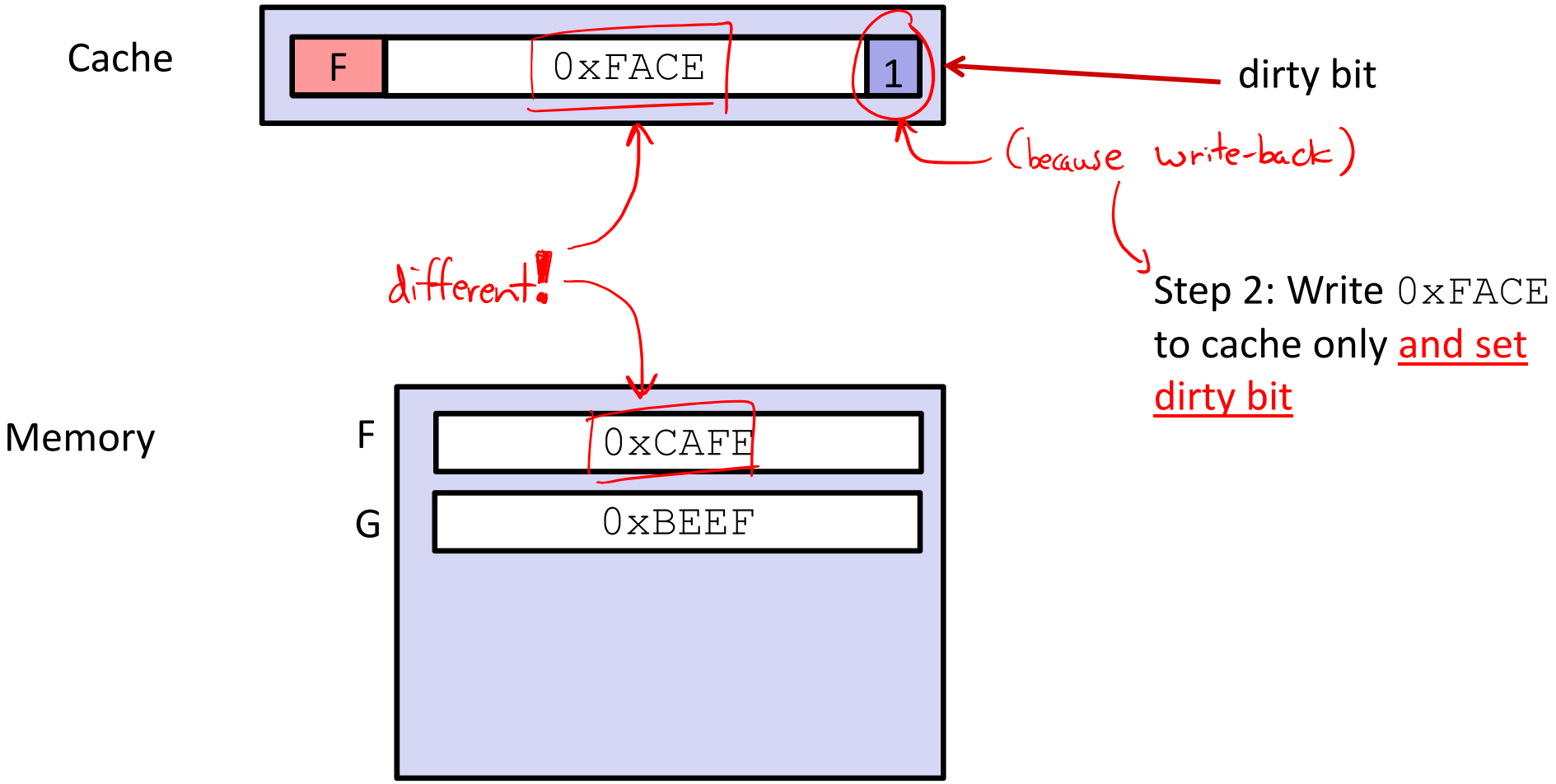
Write-back, write-allocate example



Step 1: Bring **F** into cache

Write-back, write-allocate example

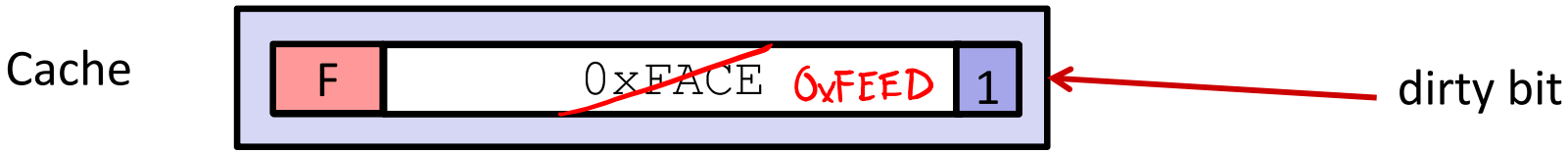
```
mov 0xFACE, F
```



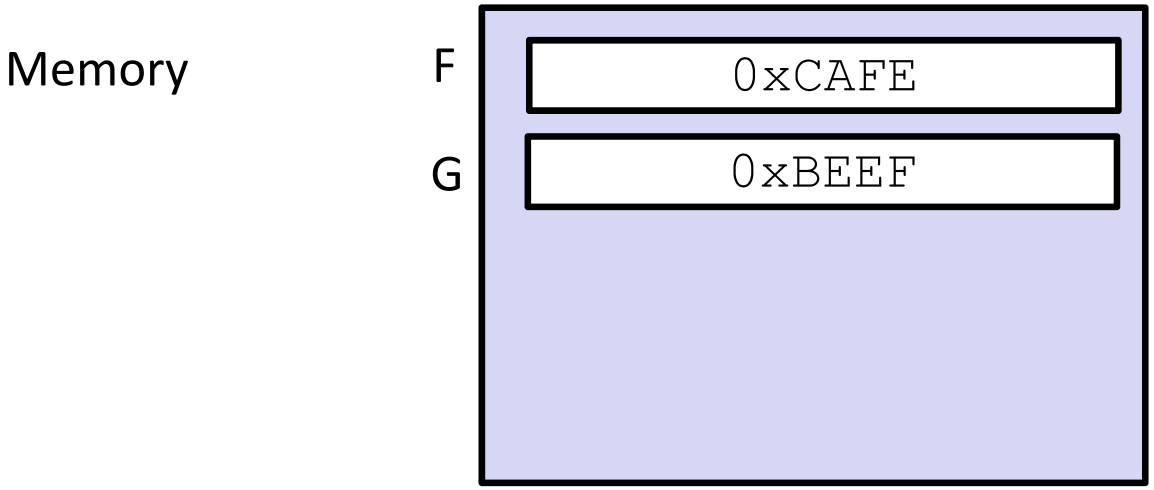
Write-back, write-allocate example

write hit

```
mov 0xFACE, F      mov 0xFEED, F
```



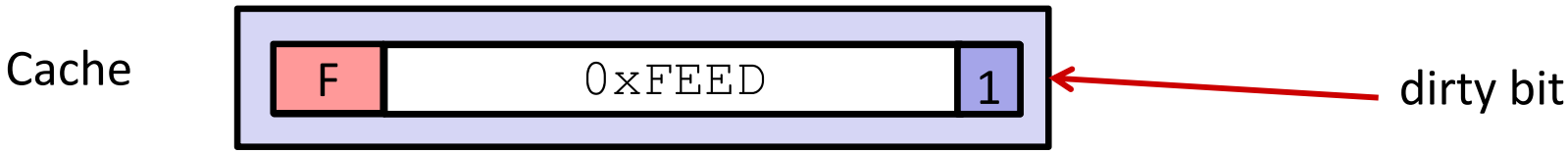
Write hit!
Write 0xFEED to cache only

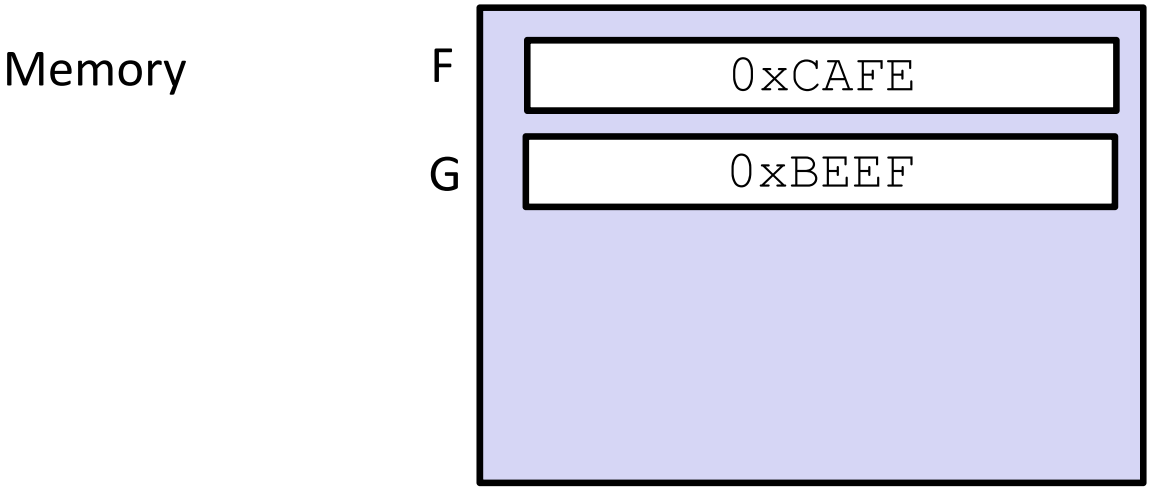


Write-back, write-allocate example

```
mov 0xFACE, F      mov 0xFEED, F      mov G, %rax
```

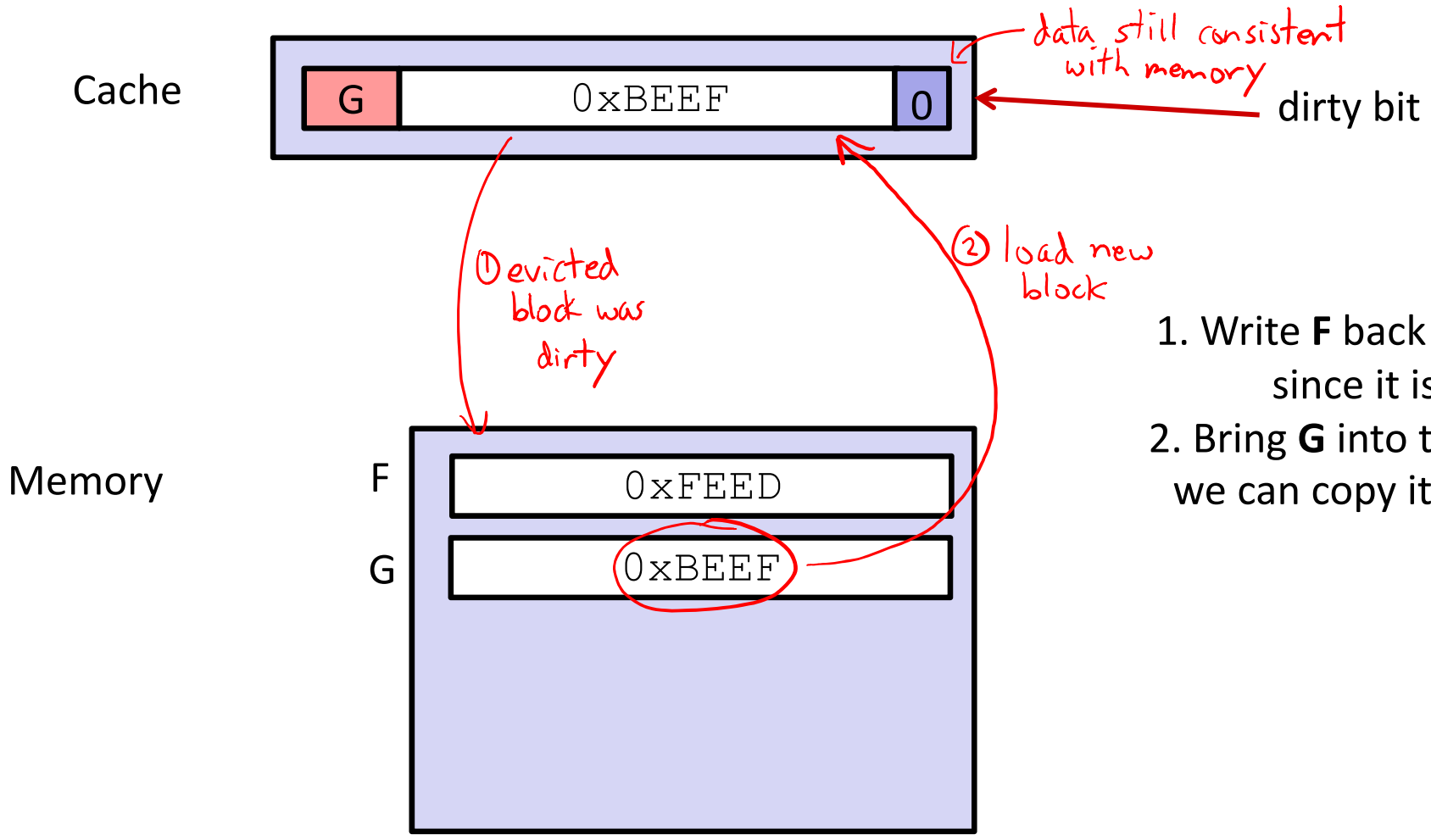
read miss





Write-back, write-allocate example

```
mov 0xFACE, F      mov 0xFEED, F          mov G, %rax
```



1. Write **F** back to memory since it is dirty
2. Bring **G** into the cache so we can copy it into `%rax`

Peer Instruction Question

- Which of the following cache statements is FALSE?
  - Vote at http://PollEv.com/justinh

A. We can reduce compulsory misses by decreasing our block size — smaller block size pulls fewer bytes into $ on a miss

B. We can reduce conflict misses by increasing associativity — more options to place blocks before evictions occur

C. A write-back cache will save time for code with good temporal locality on writes — frequently-used blocks rarely get evicted, so fewer write-backs

D. A write-through cache will always match data with the memory hierarchy level below it — yes, its main goal is data consistency

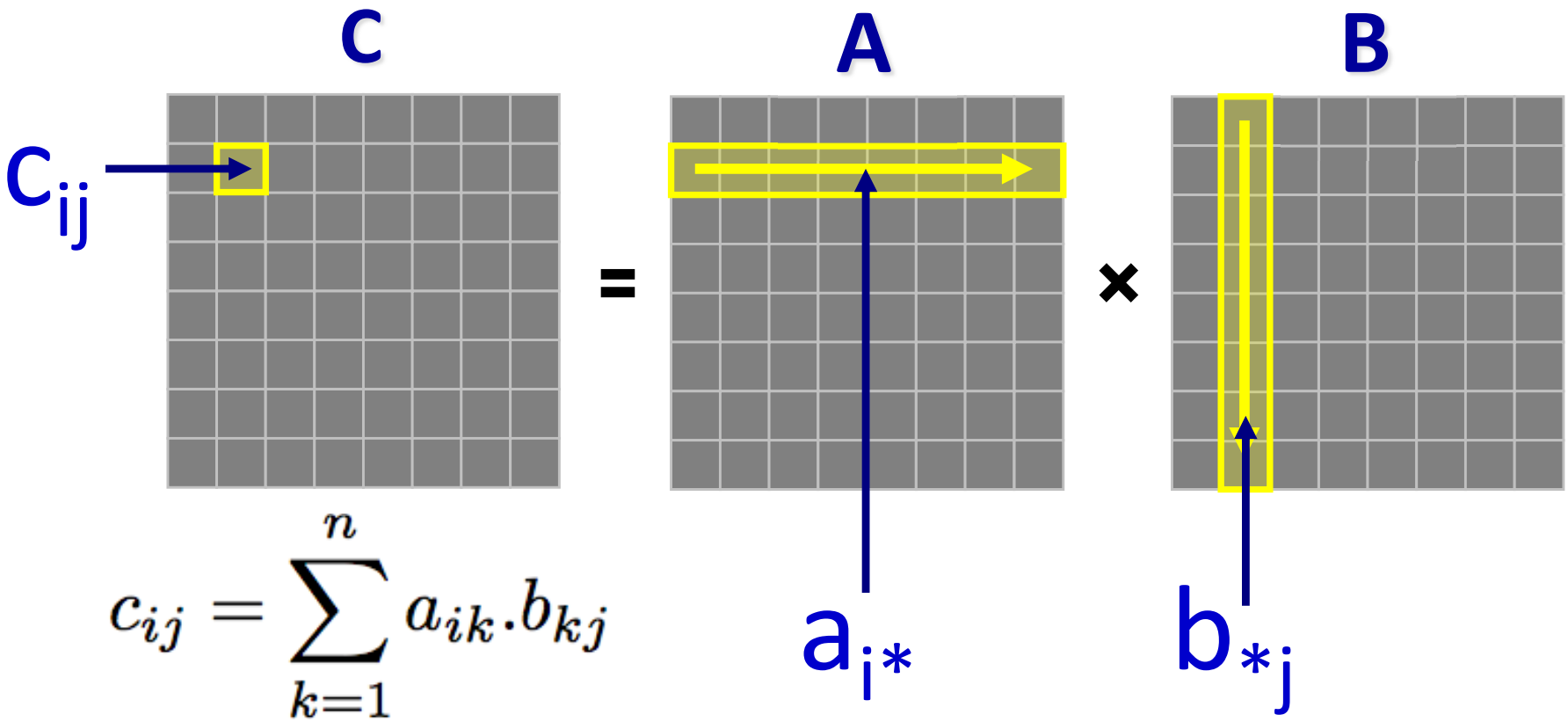
E. We're lost…

Optimizations for the Memory Hierarchy

- Write code that has locality!
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- How can you achieve locality?
  - Adjust memory accesses in *code* (software) to improve miss rate (MR)
    - Requires knowledge of *both* how caches work as well as your system's parameters
  - Proper choice of algorithm
  - Loop transformations

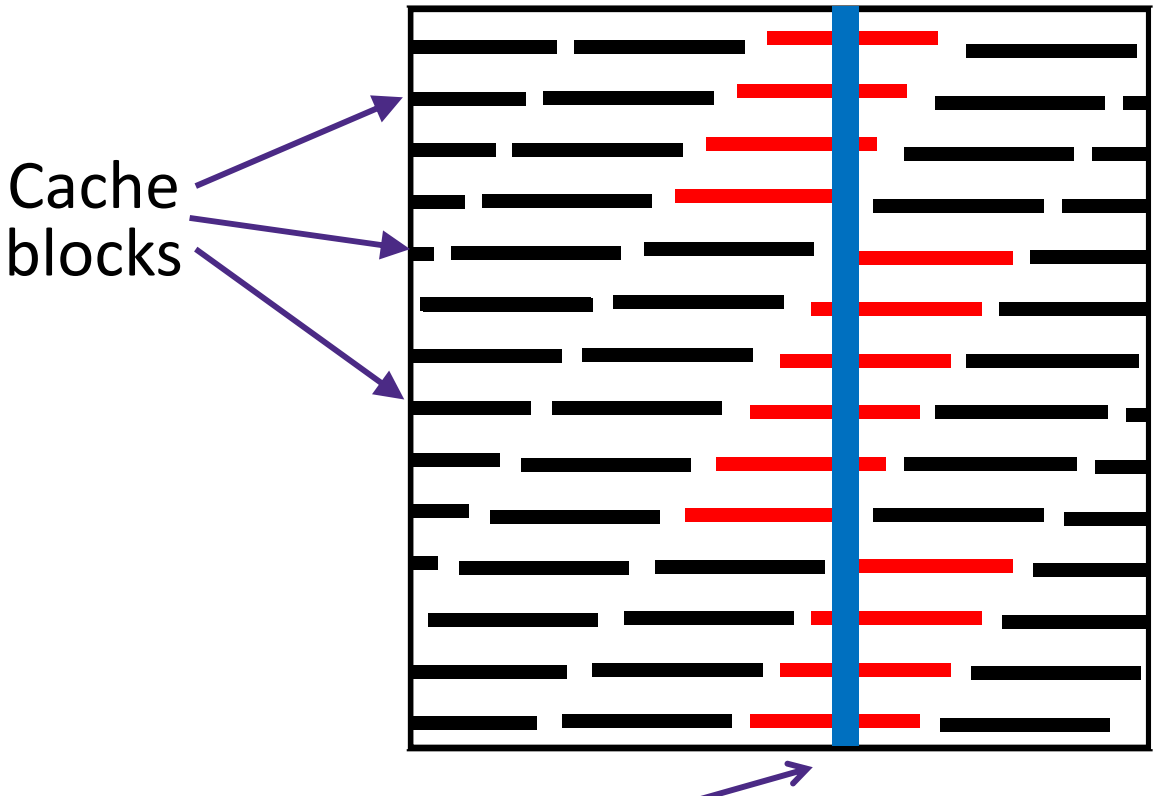
Example: Matrix Multiplication



$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$$

Matrices in Memory

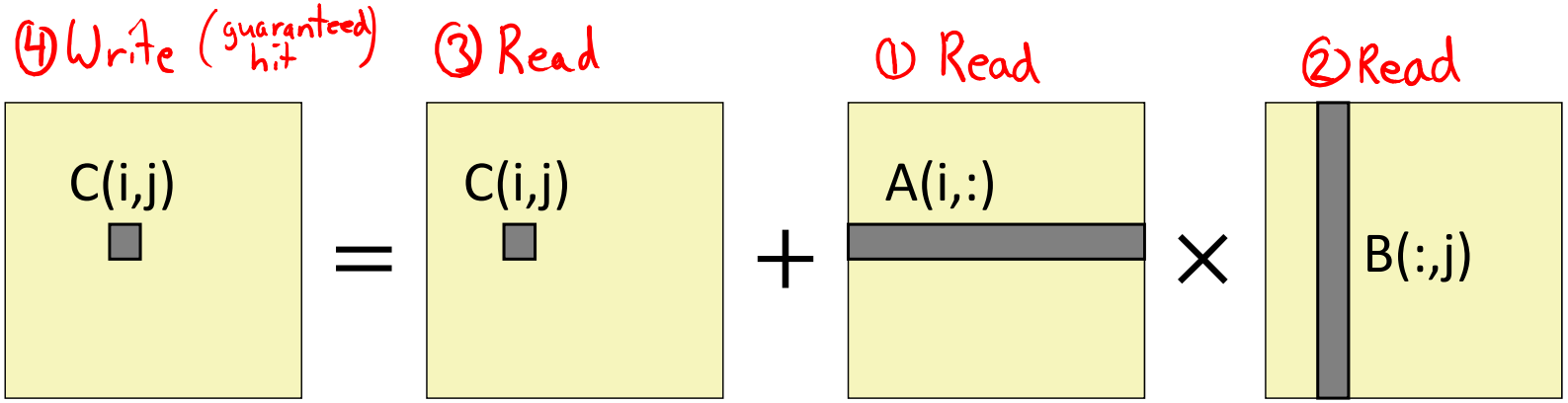
- How do cache blocks fit into this scheme?
  - Row major matrix in memory:



Naïve Matrix Multiply

```
# move along rows of A
for (i = 0; i < n; i++)
  # move along columns of B
  for (j = 0; j < n; j++)
    # EACH k loop reads row of A, col of B
    # Also read & write c(i,j) n times
    for (k = 0; k < n; k++)
      c[i*n+j] += a[i*n+k] * b[k*n+j];
```

check mem access pattern



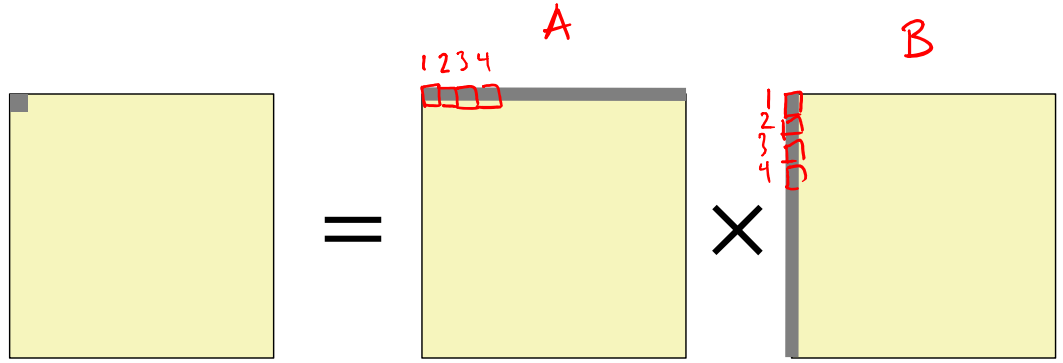
Cache Miss Analysis (Naïve)

Ignoring matrix C

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 doubles ← 8 matrix elements per cache block
  - Cache size $C \ll n$ (much smaller than n)

key assumption!

- Each iteration:
  - $\frac{n}{8}$ (A) + n (B) = $\frac{9n}{8}$ misses



compulsory

A has good spatial locality

M H H H H H H H

stride-1

B has bad spatial locality

stride-n

by the time we get to n+1, block has been kicked out of $

Cache Miss Analysis (Naïve)

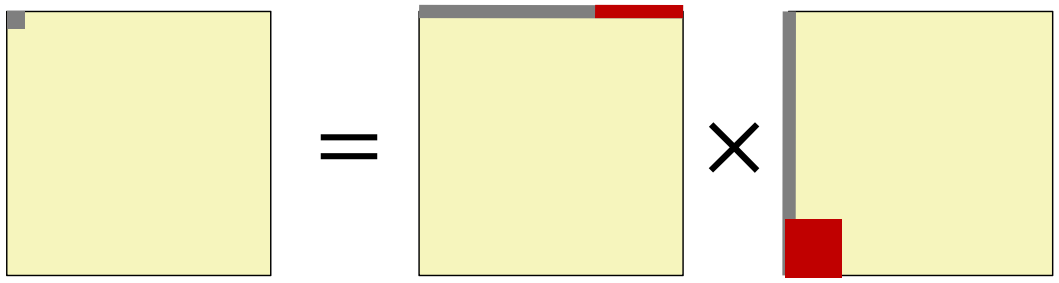
Ignoring matrix `C`

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

- Each iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses

- Afterwards in cache: (schematic)

red showing blocks remaining in the $



8 doubles wide

Cache Miss Analysis (Naïve)

Ignoring matrix `c`

- Scenario Parameters:
  - Square matrix ($n \times n$), elements are `doubles`
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

- Each iteration:
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses

- Total misses: $\frac{9n}{8} \times n^2 = \frac{9}{8}n^3$

once per element

Linear Algebra to the Rescue (1)

This is extra (non-testable) material

- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix "blocks")

- For example, multiply two 4×4 matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ with } B \text{ defined similarly.}$$

$$AB = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\ (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22}) \end{bmatrix}$$

Linear Algebra to the Rescue (2)

This is extra (non-testable) material

Matrices of size $n \times n$, split into 4 blocks of size r (n=4r)

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} * B_{k2}$$

- Multiplication operates on small "block" matrices
  - Choose size so that they fit in the cache!
  - This technique called "*cache blocking*" ★

Blocked Matrix Multiply

6 nested loops may seem less efficient, but leads to much faster code!

- Blocked version of the naïve algorithm:

```
# move by rxr BLOCKS now
for (i = 0; i < n; i += r)
  for (j = 0; j < n; j += r)
    for (k = 0; k < n; k += r)
      # block matrix multiplication
      for (ib = i; ib < i+r; ib++)
        for (jb = j; jb < j+r; jb++)
          for (kb = k; kb < k+r; kb++)
            c[ib*n+jb] += a[ib*n+kb]*b[kb*n+jb];
```

loop over block matrices

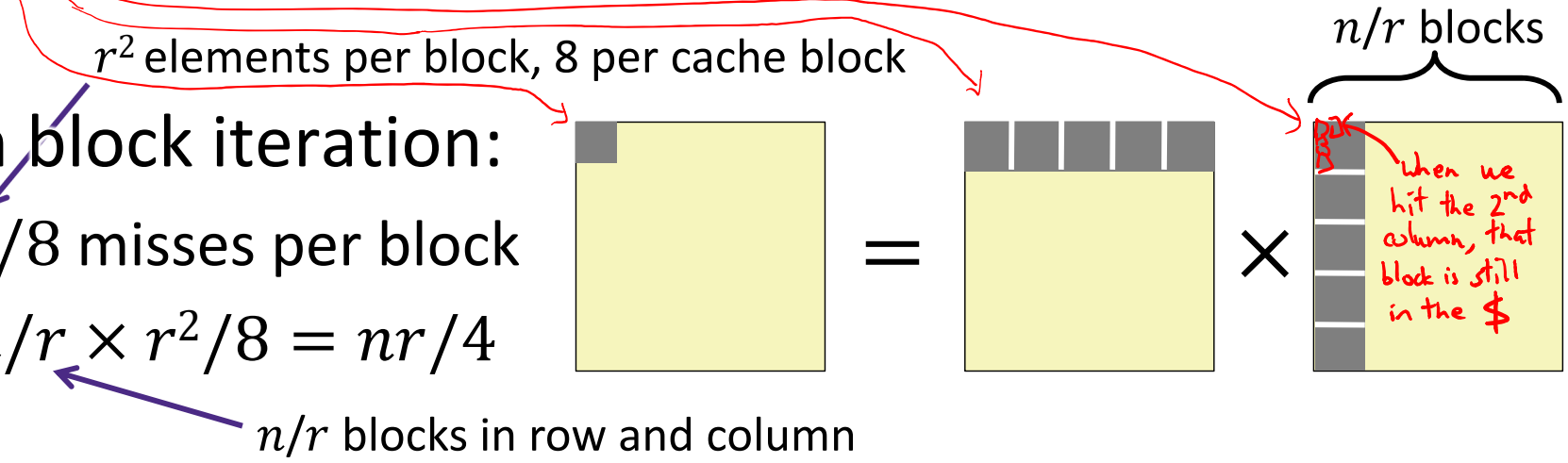
loop within block matrices

  - r = block matrix size (assume r divides n evenly)

Cache Miss Analysis (Blocked)

Ignoring matrix C

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ ($r \times r$) fit into cache: $3r^2 < C$
- Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$



n/r blocks in row and column

Cache Miss Analysis (Blocked)

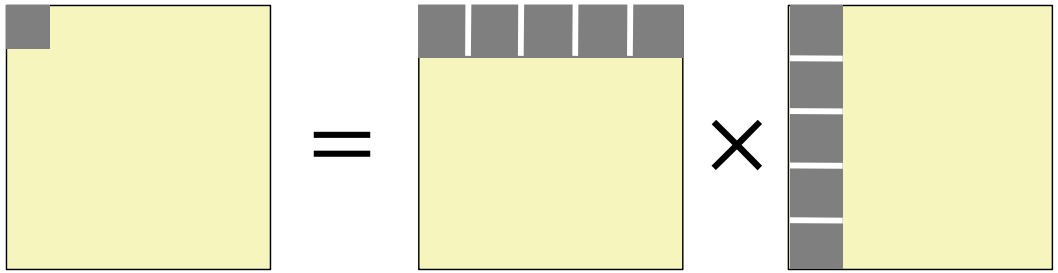
Ignoring matrix C

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ ($r \times r$) fit into cache: $3r^2 < C$

- Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$

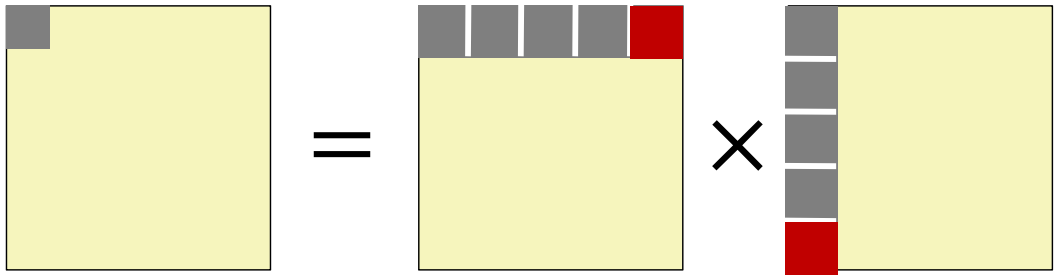
r^2 elements per block, 8 per cache block

n/r blocks



n/r blocks in row and column

  - Afterwards in cache (schematic)

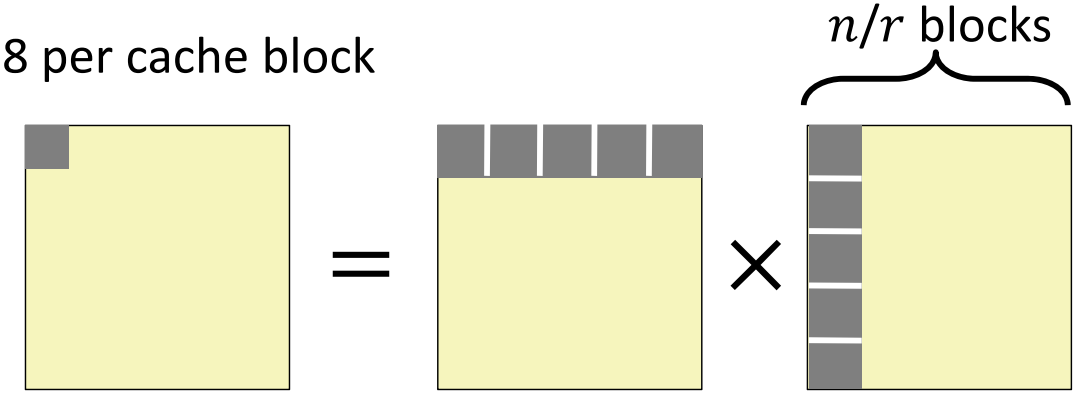


Cache Miss Analysis (Blocked)

Ignoring matrix C

- Scenario Parameters:
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ ($r \times r$) fit into cache: $3r^2 < C$

- Each block iteration:
  - r^2 elements per block, 8 per cache block
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$
  - n/r blocks in row and column



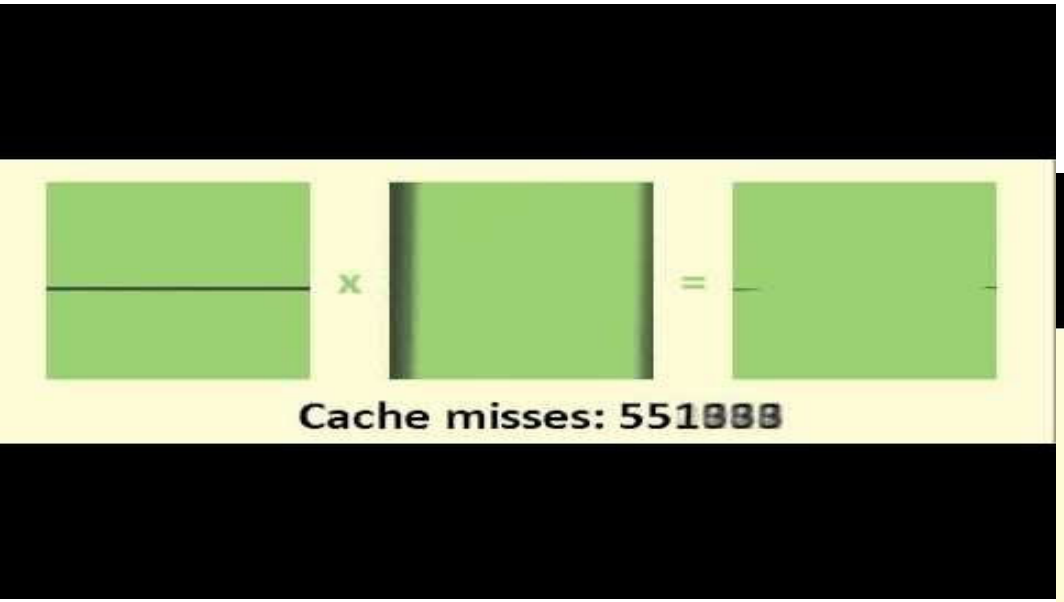
- Total misses:
  - $nr/4 \times (n/r)2 = n^3/(4r)$ vs. $9n^3/8$

Matrix Multiply Visualization

- Here n = 100, C = 32 KiB, r = 30

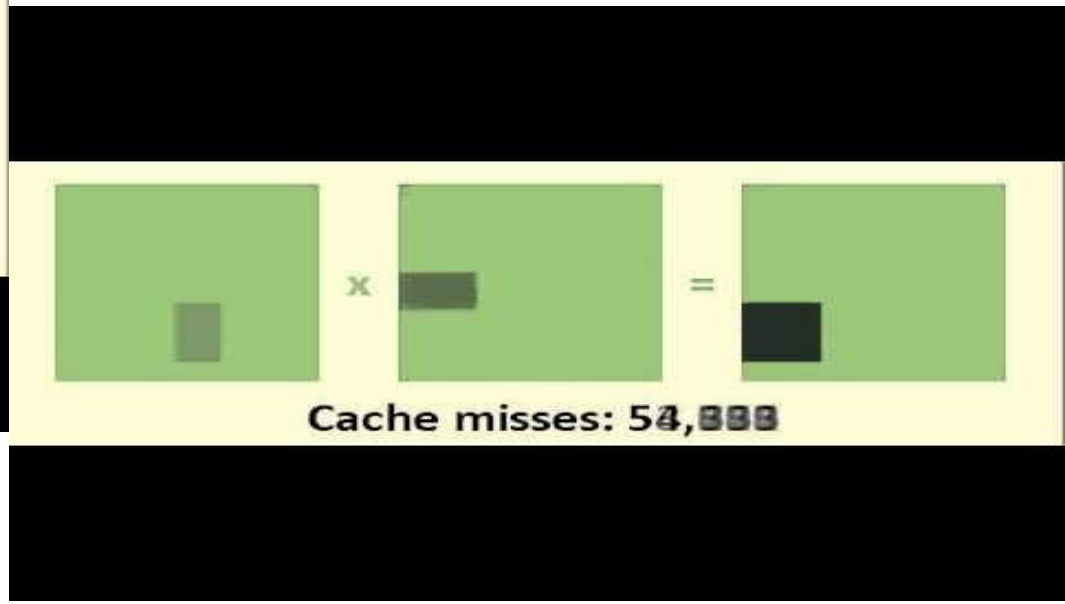
shaded areas show blocks stored in the $

Naïve:



≈ 1,020,000
cache misses

Blocked:



≈ 90,000
cache misses

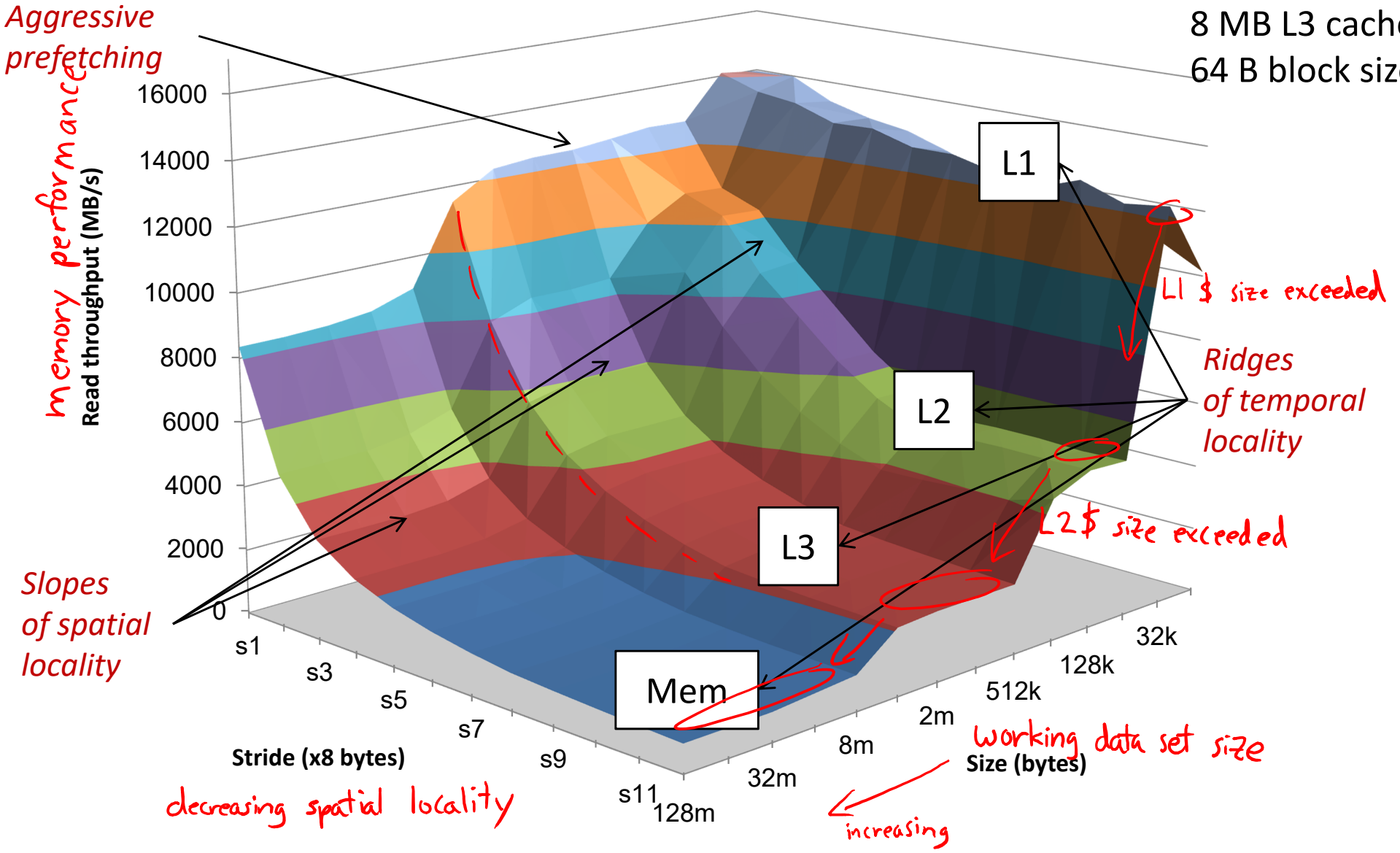
Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor "cache-friendly code"
  - Getting absolute optimum performance is very platform specific
    - Cache size, cache block size, associativity, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code

great general rules of thumb!

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Learning About Your Machine

- **Linux:**
  - `lscpu`
  - ls /sys/devices/system/cpu/cpu0/cache/index0/
    - Ex: cat /sys/devices/system/cpu/cpu0/cache/index*/size
- **Windows:**
  - `wmic memcache get <query>` (all values in KB)
  - Ex: `wmic memcache get MaxCacheSize`

- Modern processor specs: http://www.7-cpu.com/

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

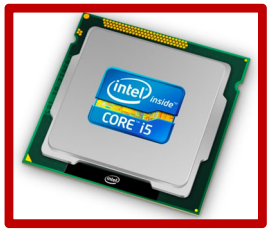
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

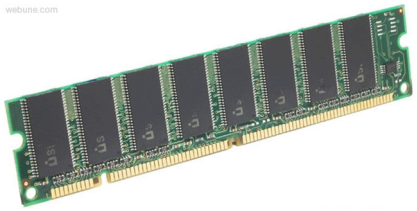
Machine code:

```
01110100000110000
10001101000001000000010
10001001110000010
11000001111110100011111
```

OS:

Computer system:





- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Leading Up to Processes

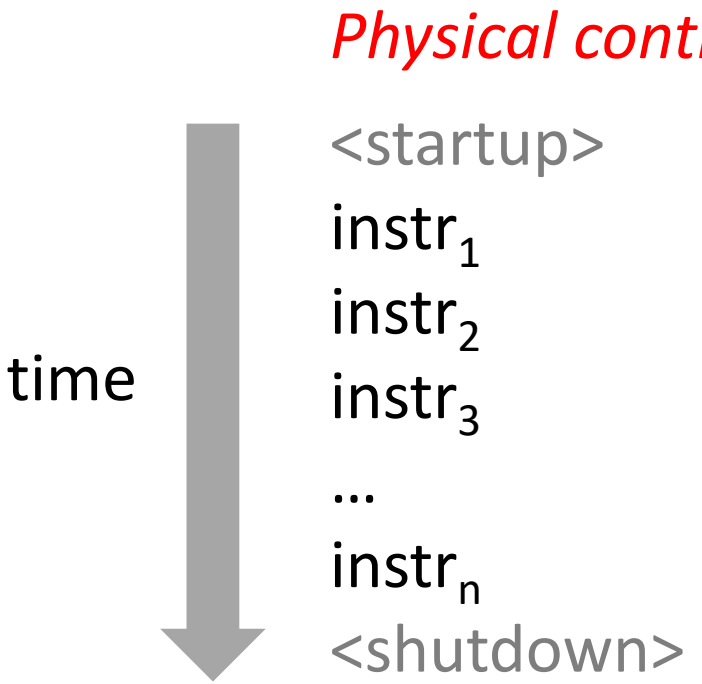
- System Control Flow
  - **Control flow**
  - **Exceptional control flow**
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)

Control Flow

- **So far:** we've seen how the flow of control changes as a *single program* executes
- **Reality:** multiple programs running *concurrently*
  - How does control flow across the many components of the system?
  - In particular: More programs running than CPUs
- *Exceptional* control flow is basic mechanism used for:
  - Transferring control between *processes* and OS
  - Handling *I/O* and *virtual memory* within the OS
  - Implementing multi-process apps like shells and web servers
  - Implementing concurrency

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)



Altering the Control Flow

- Up to now, two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  - Both react to changes in *program state*
- Processor also needs to react to changes in *system state*
  - Unix/Linux user hits "Ctrl-C" at the keyboard
  - User clicks on a different application's window on the screen
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - System timer expires
- Can jumps and procedure calls achieve this?
  - No – the system needs mechanisms for *"exceptional"* control flow!

Java Digression

This is extra (non-testable) material

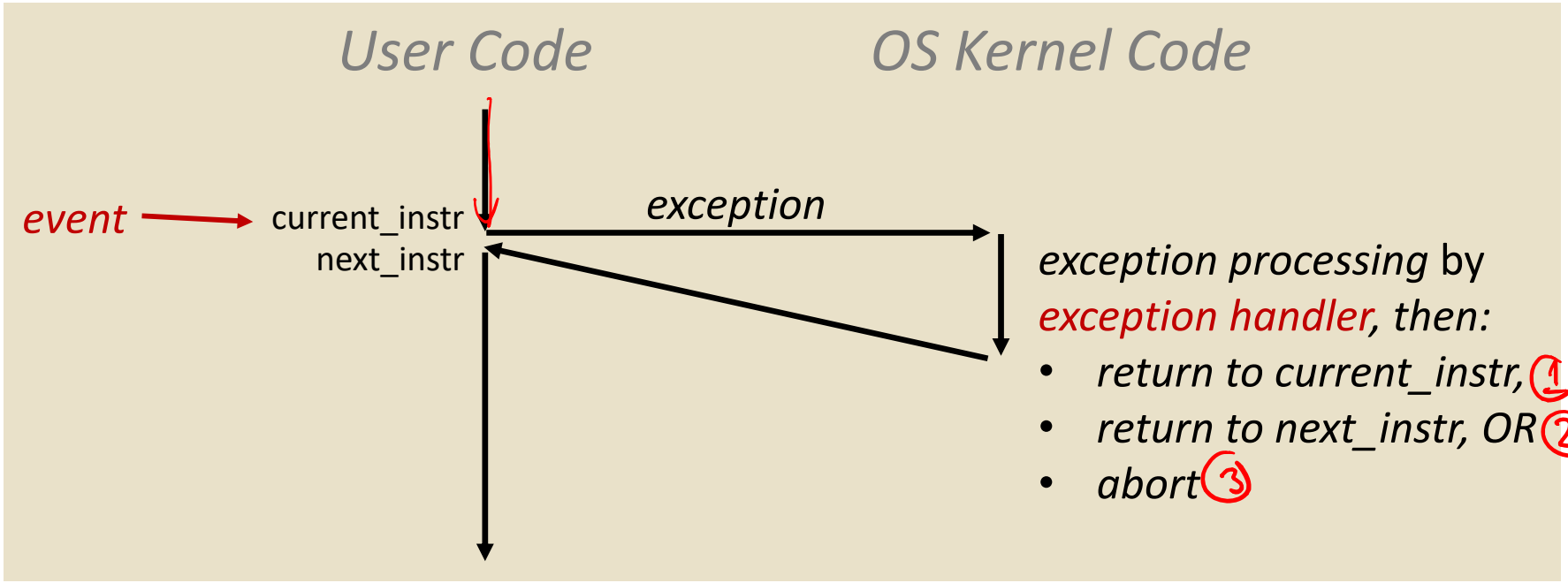
- Java has exceptions, but they're *something different*
  - Examples: NullPointerException, MyBadThingHappenedException, …
  - `throw` statements
  - `try`/`catch` statements ("throw to youngest matching catch on the call-stack, or exit-with-stack-trace if none")
- Java exceptions are for reacting to (unexpected) program state
  - Can be implemented with stack operations and conditional jumps
  - A mechanism for "many call-stack returns at once"
  - Requires additions to the calling convention, but we already have the CPU features we need
- System-state changes on previous slide are mostly of a different sort (asynchronous/external except for divide-by-zero) and implemented very differently

Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - **Exceptions**
    - Change in processor's control flow in response to a system event (*i.e.* change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software
- Higher level mechanisms
  - **Process context switch**
    - Implemented by OS software and hardware timer
  - **Signals**
    - Implemented by OS software
    - We won't cover these – see CSE451 and CSE/EE474

Exceptions

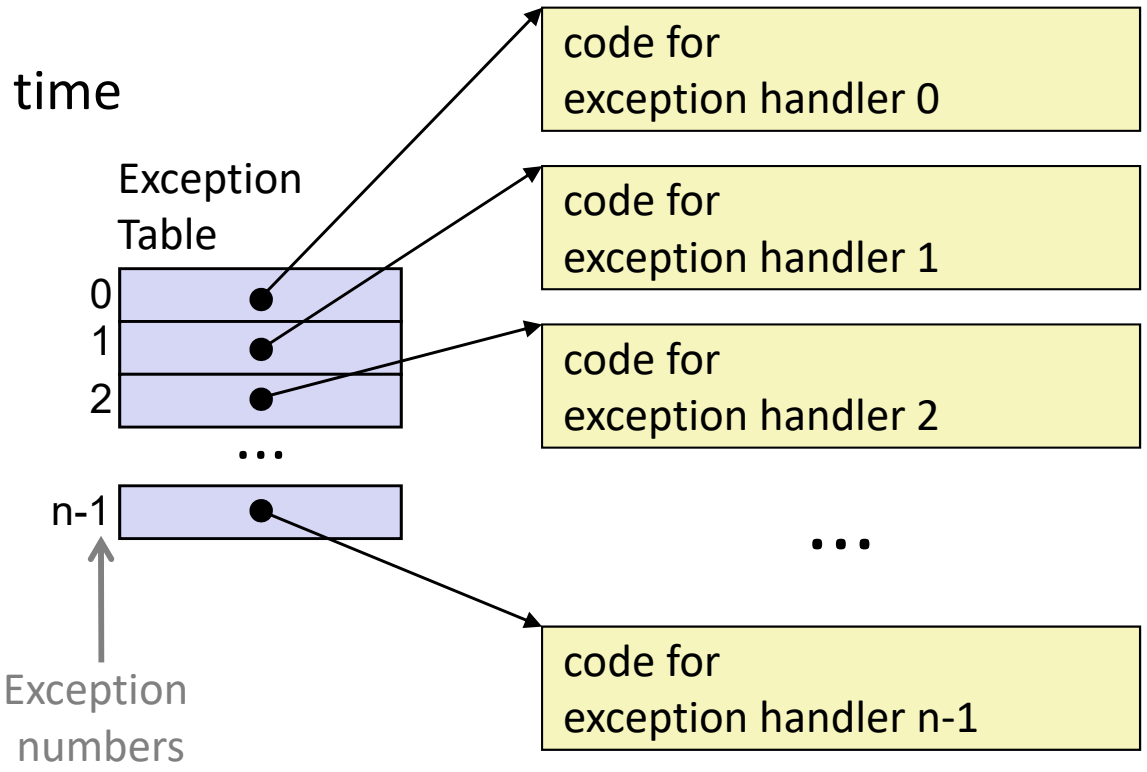
- An *exception* is transfer of control to the operating system (OS) kernel in response to some *event* (*i.e.* change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C



- *How does the system know where to jump to in the OS?*

Exception Table

- A jump table for exceptions (also called *Interrupt Vector Table*)
  - Each type of event has a unique exception number k
  - k = index into exception table (a.k.a interrupt vector)
  - Handler k is called each time exception k occurs



like a jump table in a switch statement

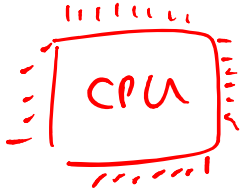
Exception Table (Excerpt)

Exception Number	*Description*	*Exception Class*
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-255	OS-defined	Interrupt or trap

Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - **Asynchronous exceptions (interrupts)**
  - **Synchronous exceptions (traps & faults)**

Asynchronous Exceptions (Interrupts)



- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to "next" instruction

- Examples:
  - I/O interrupts
    - Hitting Ctrl-C on the keyboard
    - Clicking a mouse button or tapping a touchscreen
    - Arrival of a packet from a network
    - Arrival of data from a disk
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - **Intentional**: transfer control to OS to perform some function
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to "next" instruction ("current" instr did what it was supposed to)
  - ***Faults***
    - **Unintentional** but possibly recoverable
    - Examples: *page faults*, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts*** ↑ if recoverable ↑ if not recoverable
    - **Unintentional** and unrecoverable
    - Examples: parity error, machine check (hardware failure detected)
    - Aborts current program

System Calls

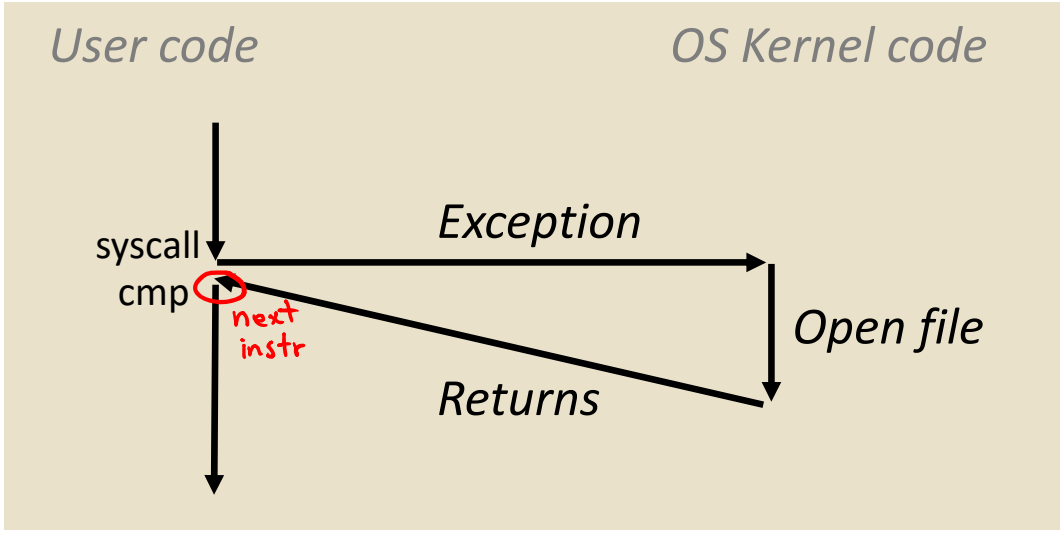
- Each system call has a unique ID number
- Examples for Linux on x86-64:

Number	Name	Description
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Traps Example: Opening File

- User calls `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax   # open is syscall 2
e5d7e:   0f 05               syscall          # return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

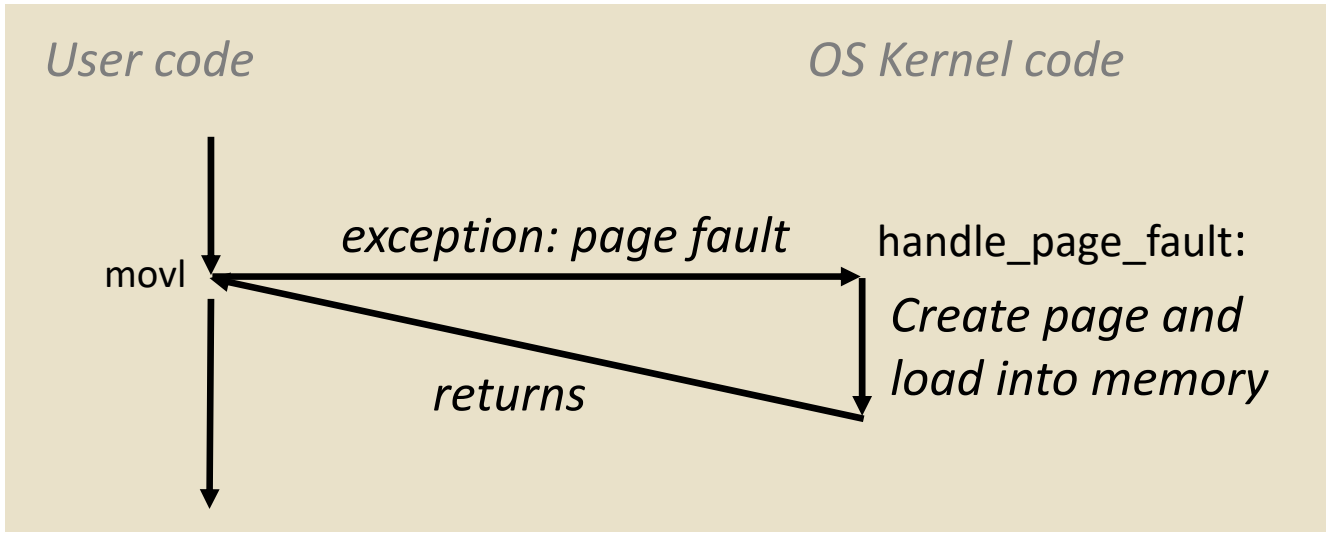
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
int main ()
{
    a[500] = 13;
}
```

```
80483b7:    c7 05 10 9d 04 08 0d    movl   $0xd,0x8049d10
```

normal mov, but address not currently in memory

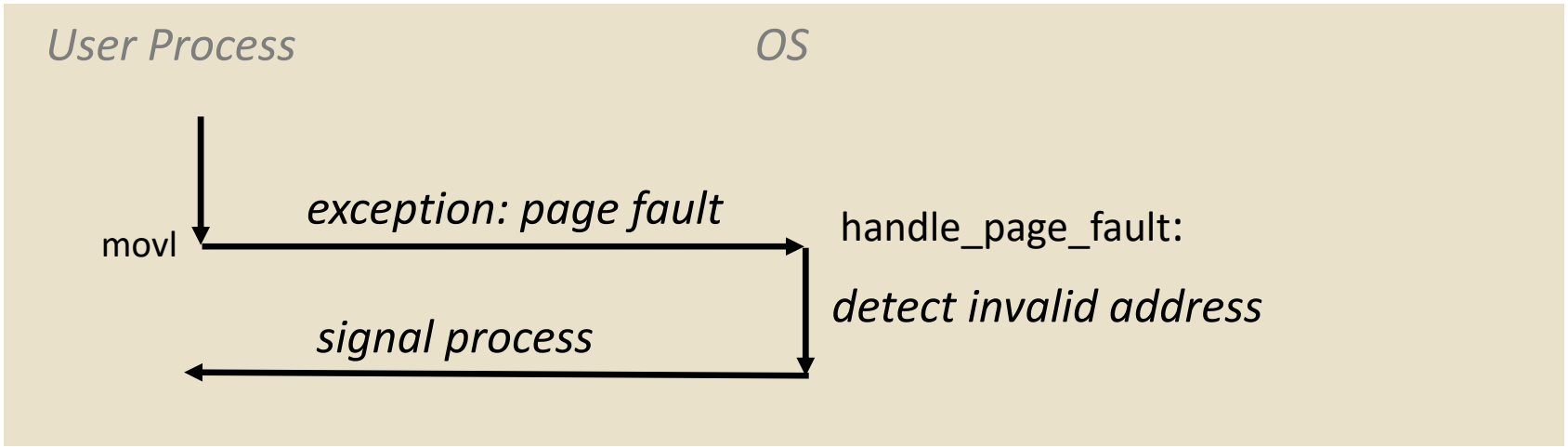


- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try ✓

Fault Example: Invalid Memory Reference

```
int a[1000];
int main()
{
    a[5000] = 13;
}
```

```
80483b7:      c7 05 60 e3 04 08 0d    movl   $0xd,0x804e360
```



- Page fault handler detects invalid address
- Sends `SIGSEGV` signal to user process
- User process exits with "segmentation fault"

Summary

- Exceptions
  - Events that require non-standard control flow
  - Generated externally (interrupts) or internally (traps and faults)
  - After an exception is handled, one of three things may happen:
    - Re-execute the current instruction
    - Resume execution with the next instruction
    - Abort the process that caused the exception