Caches IV, System Control Flow
CSE 351 Autumn 2018

Instructor: **Teaching Assistants:**

Justin Hsia   Akshat Aggarwal   An Wang   Andrew Hu

Brian Dai   Britt Henderson   James Shin

Kevin Bi   Kory Watson   Riley Germundson

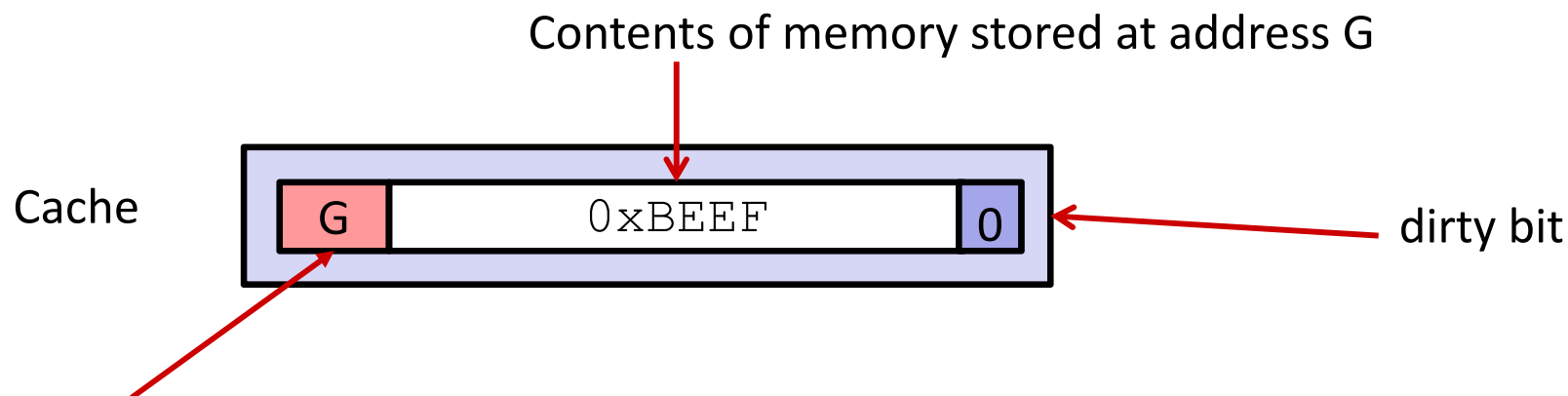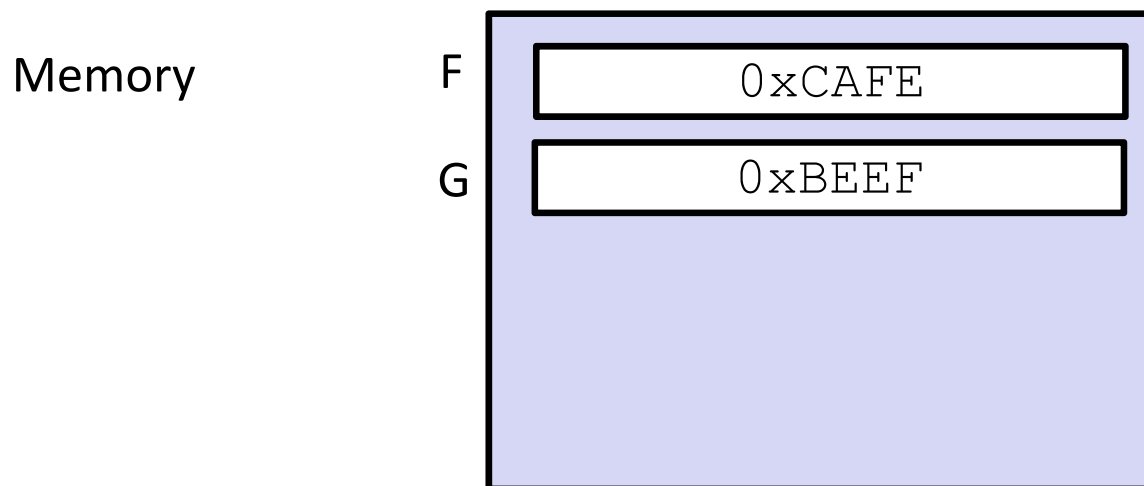Sophie Tian   Teagan Horkan

http://xkcd.com/908/
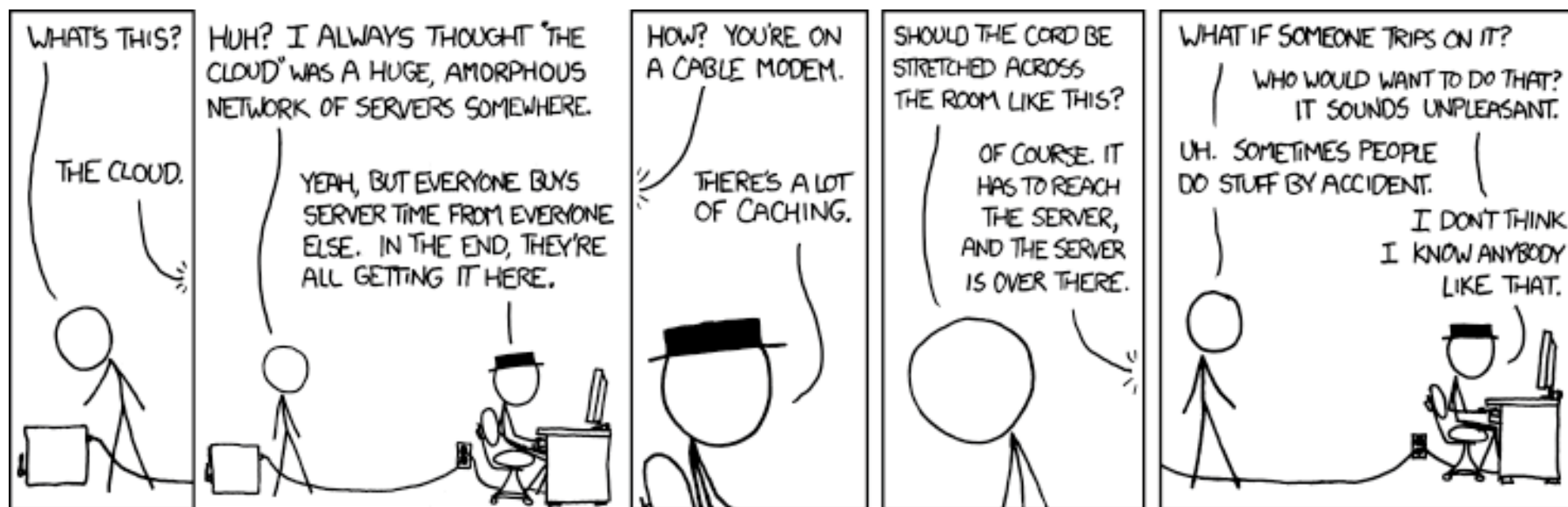
Administrivia

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

Write-back, write-allocate example

Contents of memory stored at address G

Cache
G | 0xBEEF | 0   dirty bit

tag (there is only one set in this tiny cache, so the tag is the entire block address!)
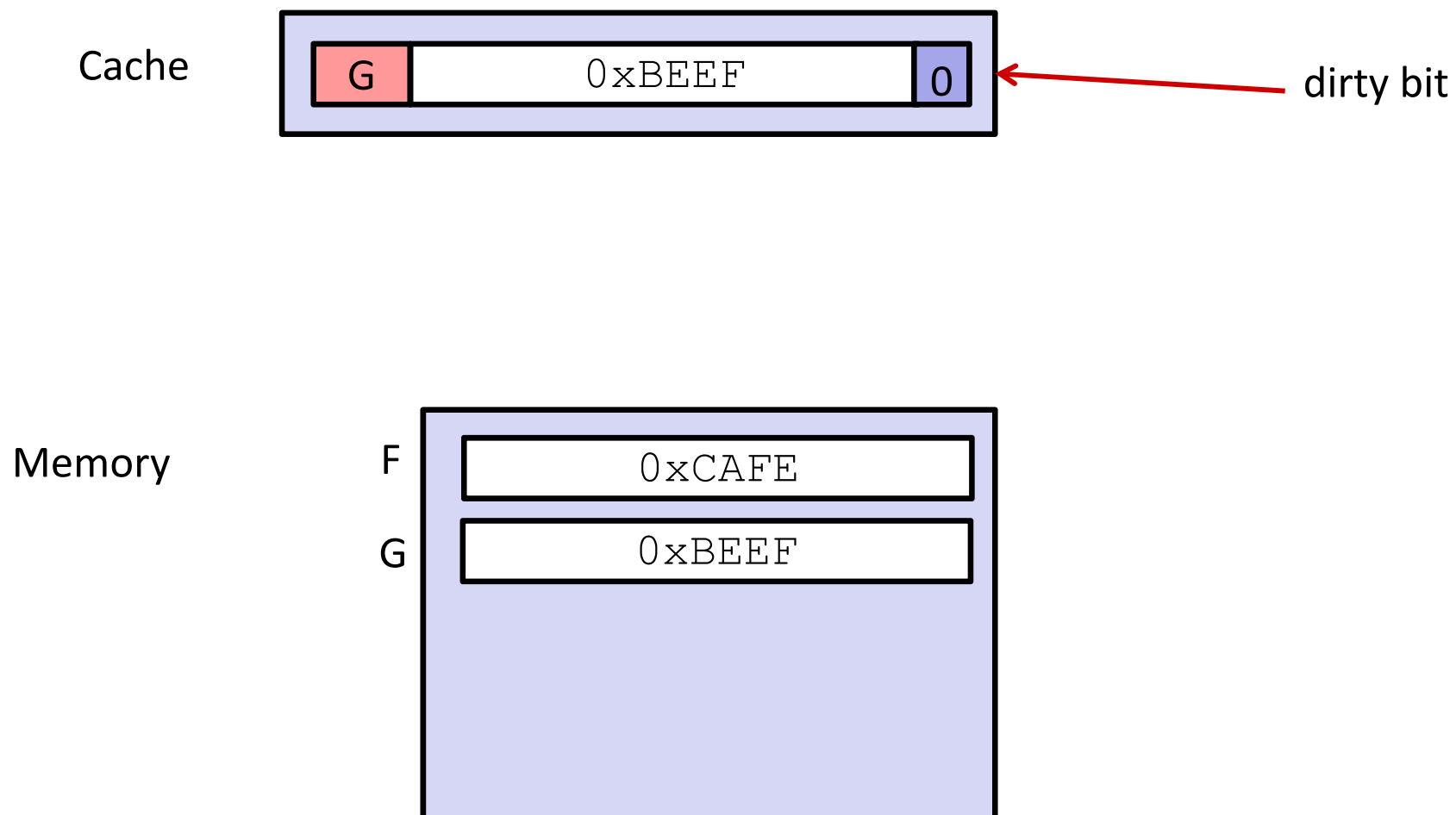
Memory   F | 0xCAFE

G | 0xBEEF

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.

3

Write-back, write-allocate example

`mov 0xFACE, F`
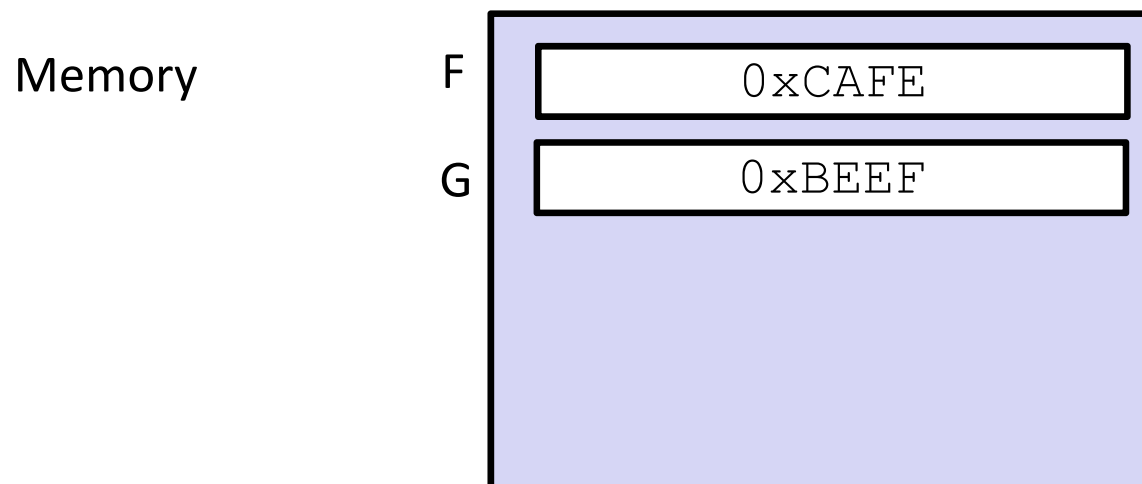
Cache

| G | 0xBEEF | 0 |

← dirty bit

Memory

F 0xCAFE

G 0xBEEF

Write-back, write-allocate example

`mov 0xFACE, F`

Cache

| F | 0xCAFE | 0 |

← dirty bit

Step 1: Bring **F** into cache

Memory

F   0xCAFE

G   0xBEEF

Write-back, write-allocate example

`mov 0xFACE, F`

Cache

F | `0xFACE` | 1

dirty bit

Memory

F | `0xCAFE`

G | `0xBEEF`
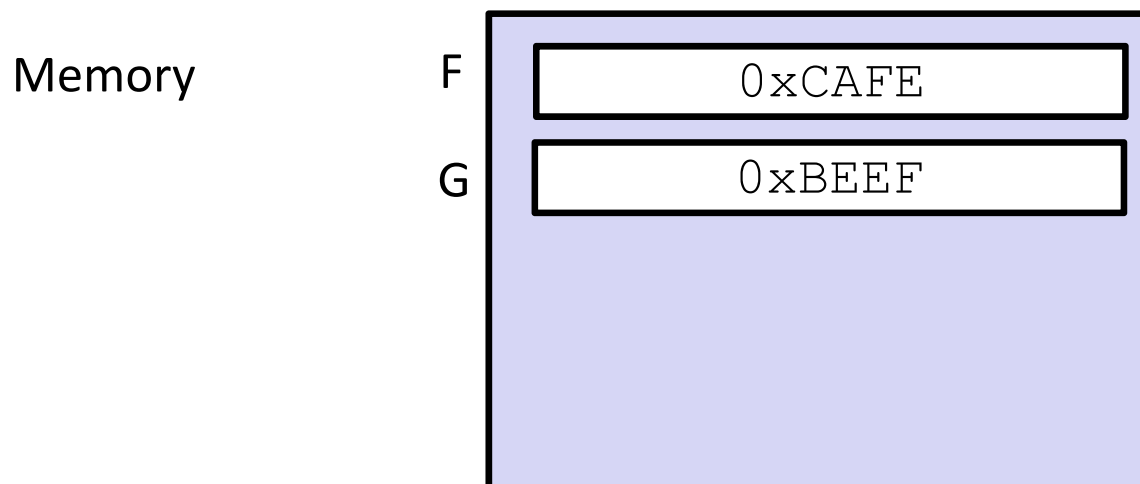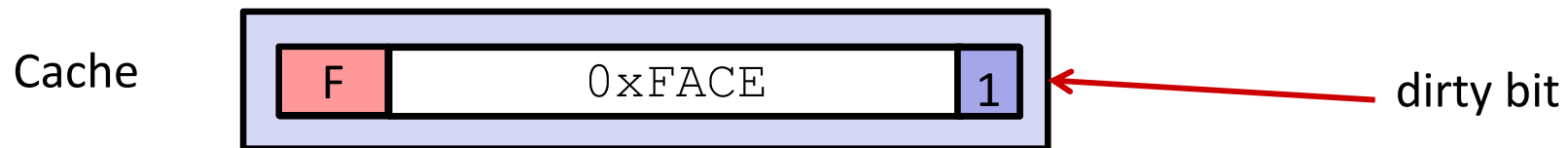
Step 2: Write `0xFACE` to cache only and set dirty bit

Write-back, write-allocate example
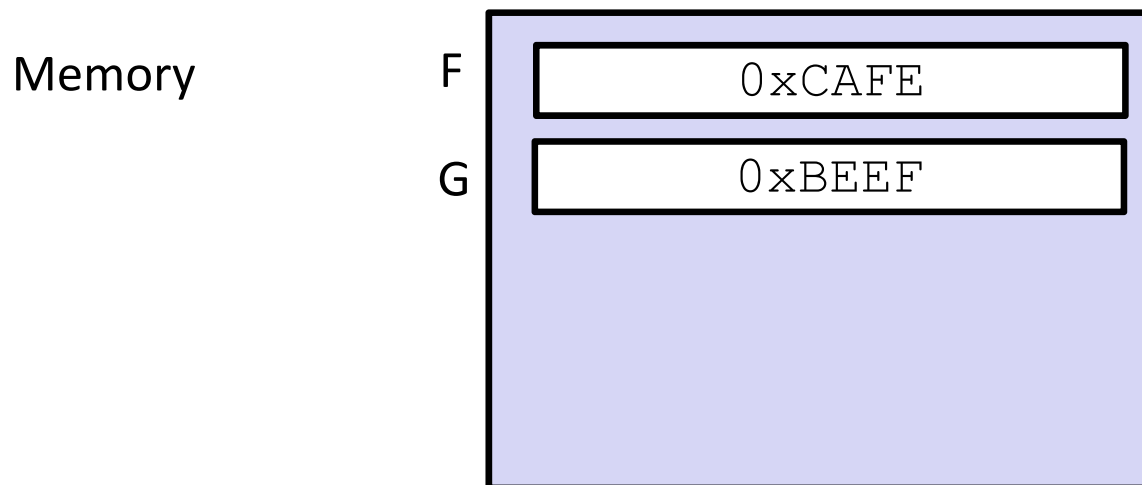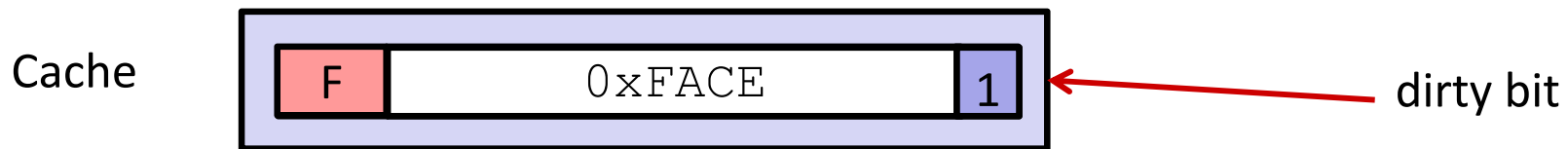
`mov 0xFACE, F`   `mov 0xFEED, F`

Cache

| F | 0xFACE | 1 |

dirty bit

Write hit!
Write `0xFEED` to
cache only

Memory

F | 0xCAFE

G | 0xBEEF

Write-back, write-allocate example

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

Cache

| F | 0xFEED | 1 |

← dirty bit

Memory

F 0xCAFE

G 0xBEEF

Write-back, write-allocate example

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

Cache

dirty bit

Memory

F   `0xFEED`

G   `0xBEEF`
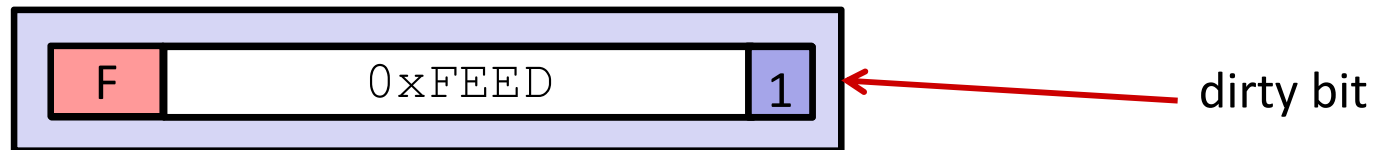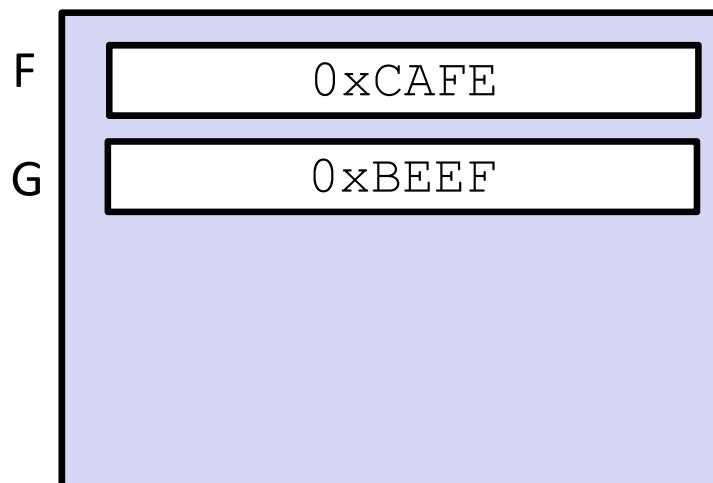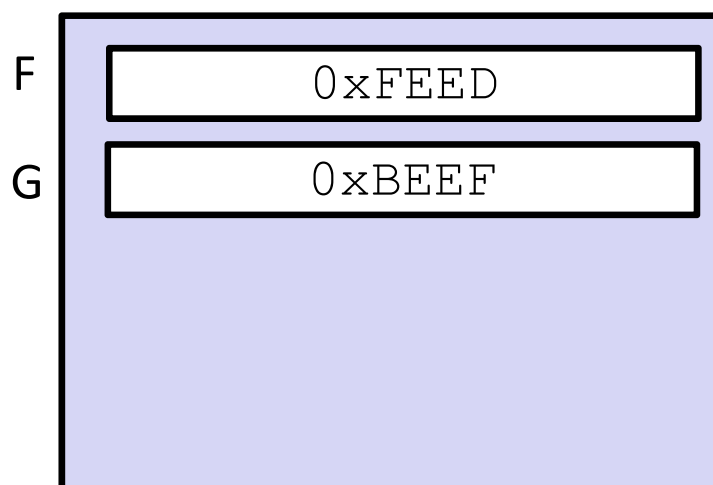
1. Write **F** back to memory since it is dirty
2. Bring **G** into the cache so we can copy it into `%rax`

9

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**

  B. **We can reduce conflict misses by increasing associativity**

  C. **A write-back cache will save time for code with good temporal locality on writes**

  D. **A write-through cache will always match data with the memory hierarchy level below it**

  E. **We're lost…**

Optimizations for the Memory Hierarchy

- ❖ Write code that has locality!
  - ▪ <u>Spatial</u>: access data contiguously
  - ▪ <u>Temporal</u>: 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   **A**   **B**

c_{ij}

=   ×

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

a_{i*} b_{*j}
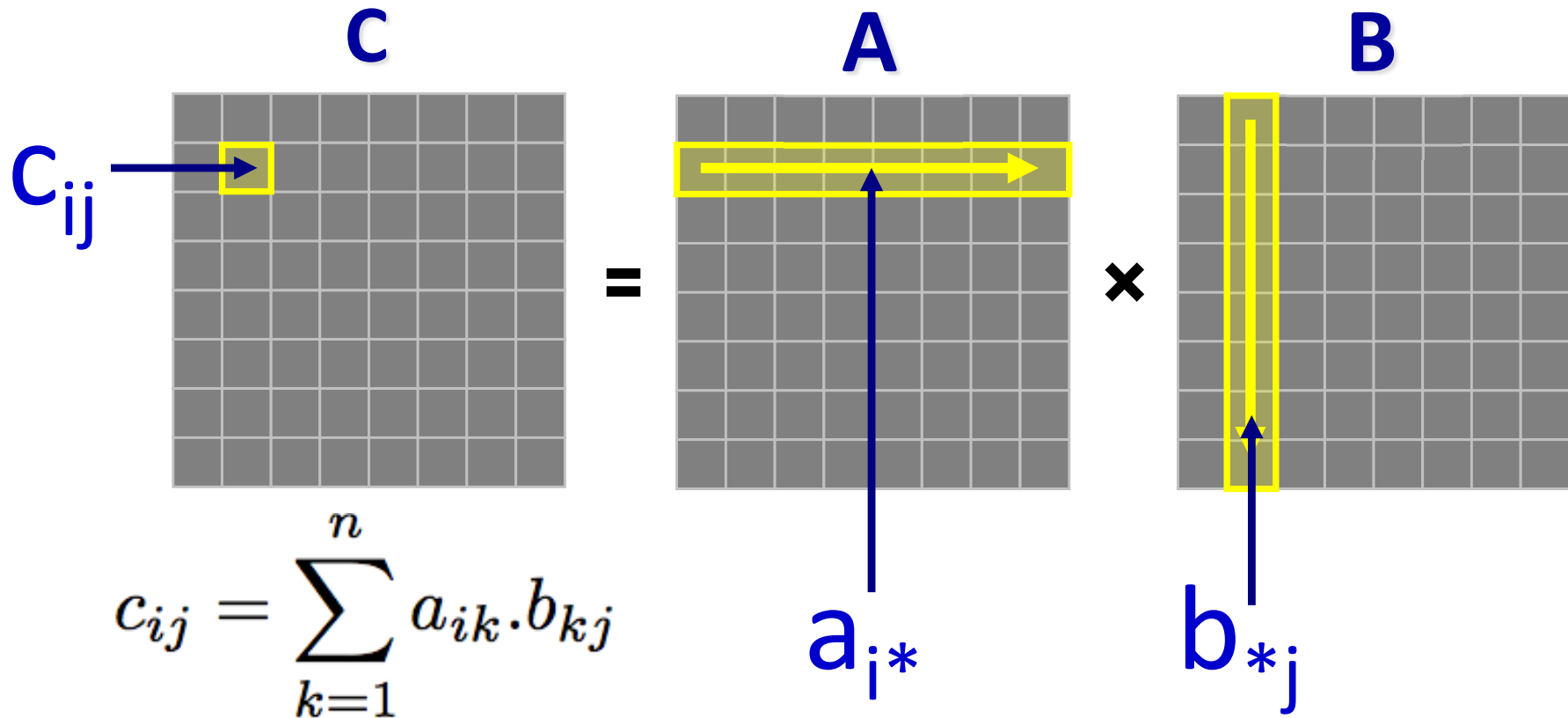
Matrices in Memory

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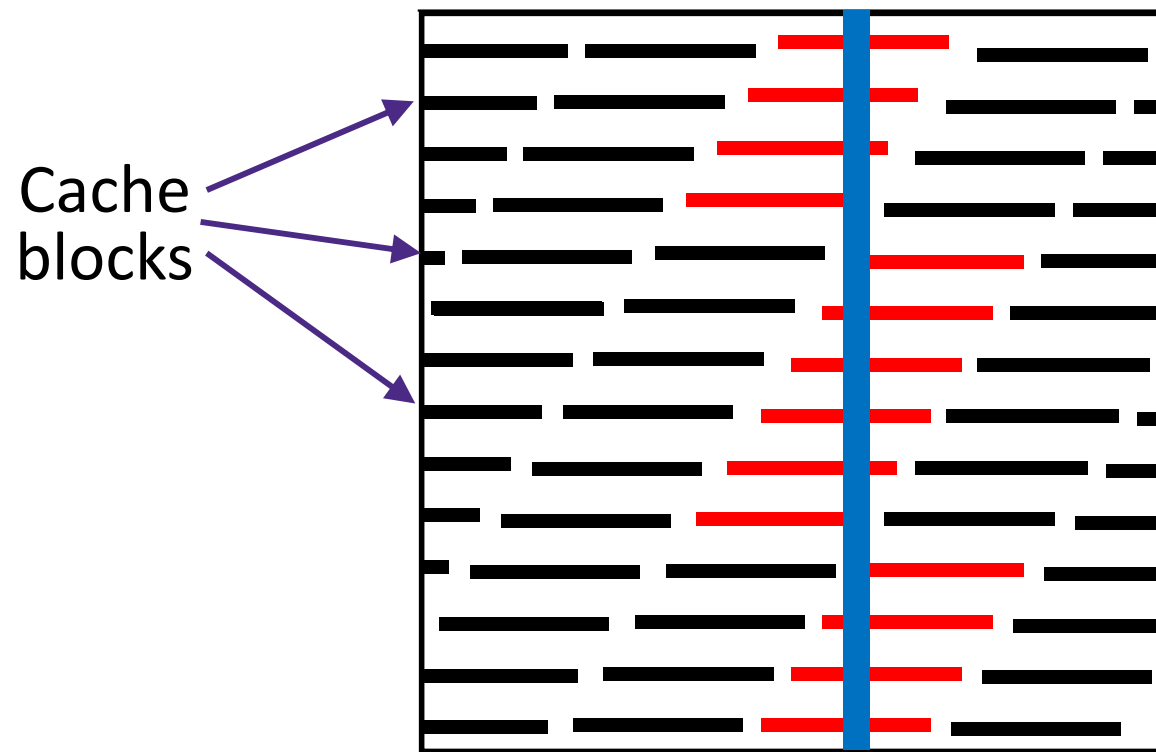
Cache blocks

COLUMN of matrix (blue) is spread among cache blocks shown in red

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];
```
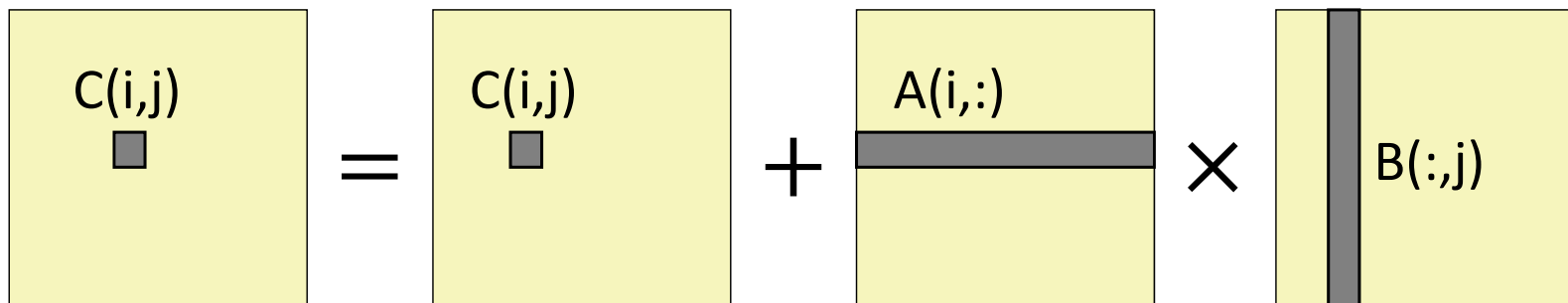
C(i,j)   =   C(i,j)   +   A(i,:)   ×   B(:,j)

Cache Miss Analysis (Naïve)

Ignoring matrix c

❖ Scenario Parameters:

  ▪ Square matrix ($n \times n$), elements are `double`s

  ▪ 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

Cache Miss Analysis (Naïve)

Ignoring matrix c

❖ Scenario Parameters:

■ Square matrix ($n \times n$), elements are `double`s

■ 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)

8 doubles wide

16

Cache Miss Analysis (Naïve)

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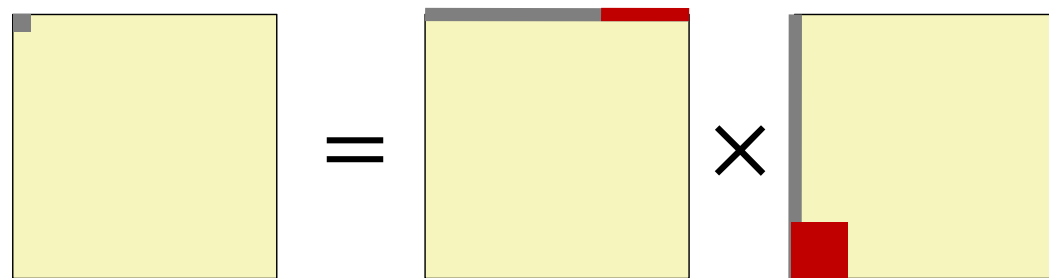
- ❖ Scenario Parameters:
  - Square matrix ($n \times n$), elements are `double`s
  - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)

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

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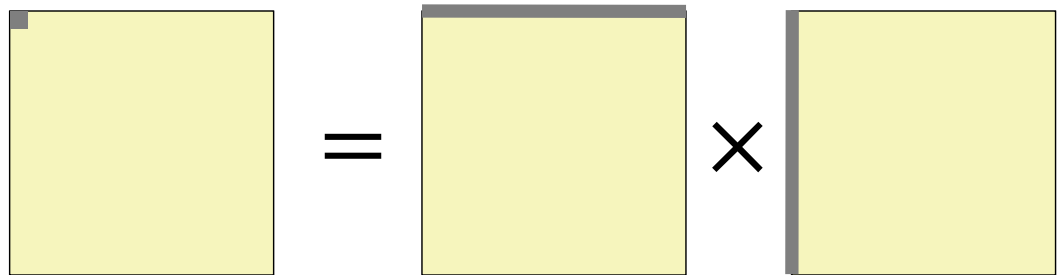
- ❖ Total misses: $\frac{9n}{8} \times n^2 = \frac{9}{8}n^3$

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

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

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_{144}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

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

❖ 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];
```

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

Cache Miss Analysis (Blocked)

Ignoring matrix $\mathbb{C}$

❖ 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: $3r^2 < C$

r^2 elements per block, 8 per cache block

n/r blocks

❖ 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 $\mathbb{C}$

- ❖ 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: $3r^2 < C$

r^2 elements per block, 8 per cache block

n/r blocks

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

n/r blocks in row and column

  - Afterwards in cache (schematic)
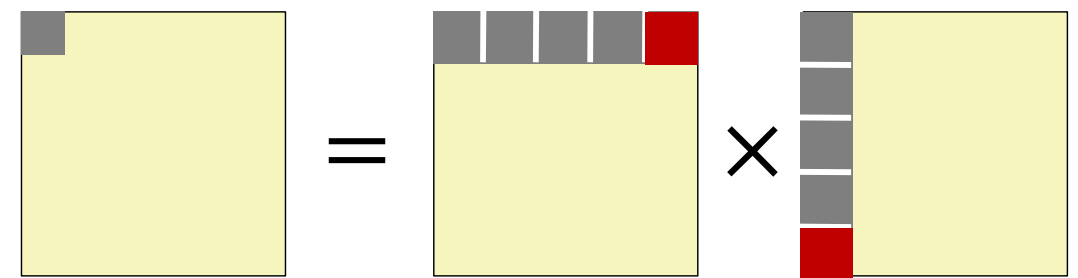
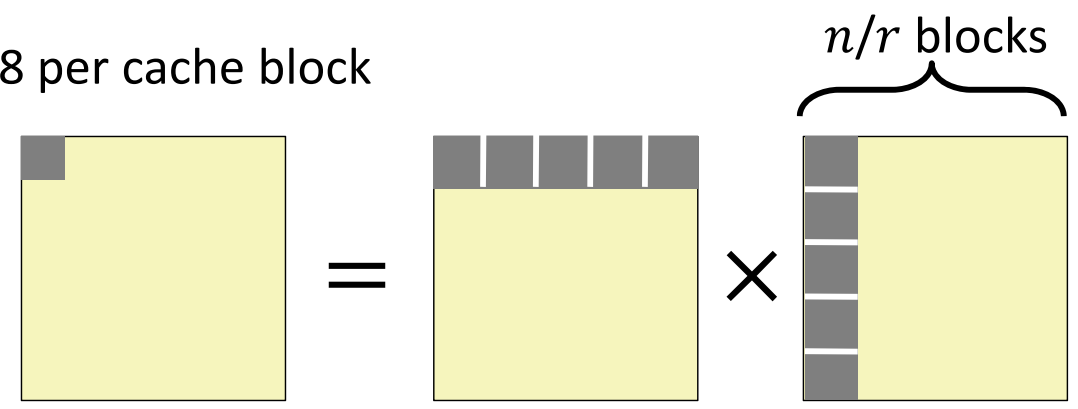# Cache Miss Analysis (Blocked)

Ignoring matrix $\subset$

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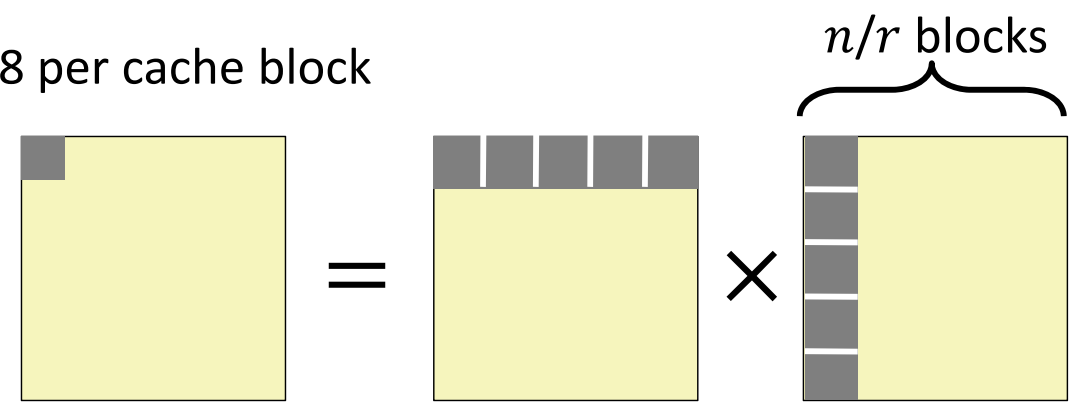 - Cache block size K = 64 B = 8 doubles
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ ($r \times r$) fit into cache: $3r^2 < C$

r^2 elements per block, 8 per cache block

n/r blocks

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

n/r blocks in row and column
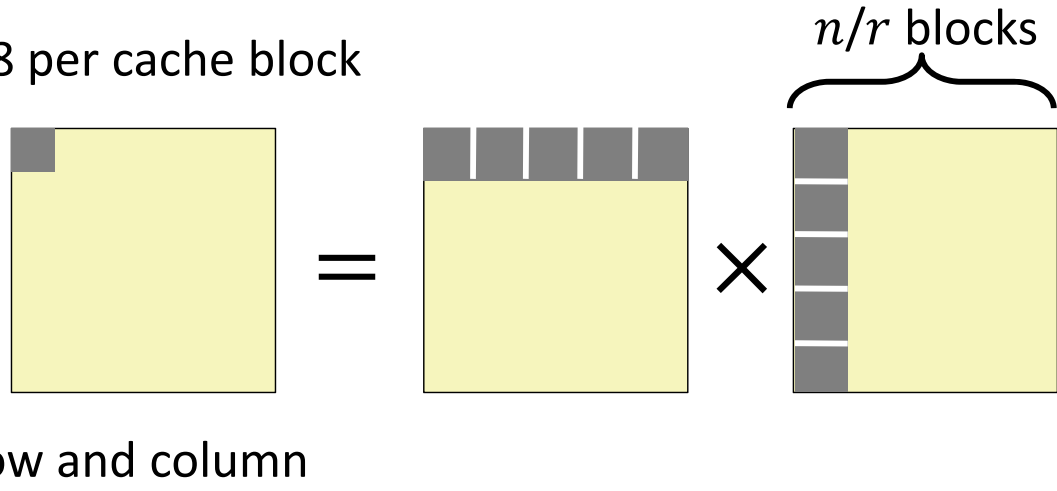
- ❖ Total misses:
  - $nr/4 \times (n/r)2 = n^3/(4r)$

Matrix Multiply Visualization

❖ Here n = 100, C = 32 KiB, r = 30

Naïve:

Cache misses: 551888
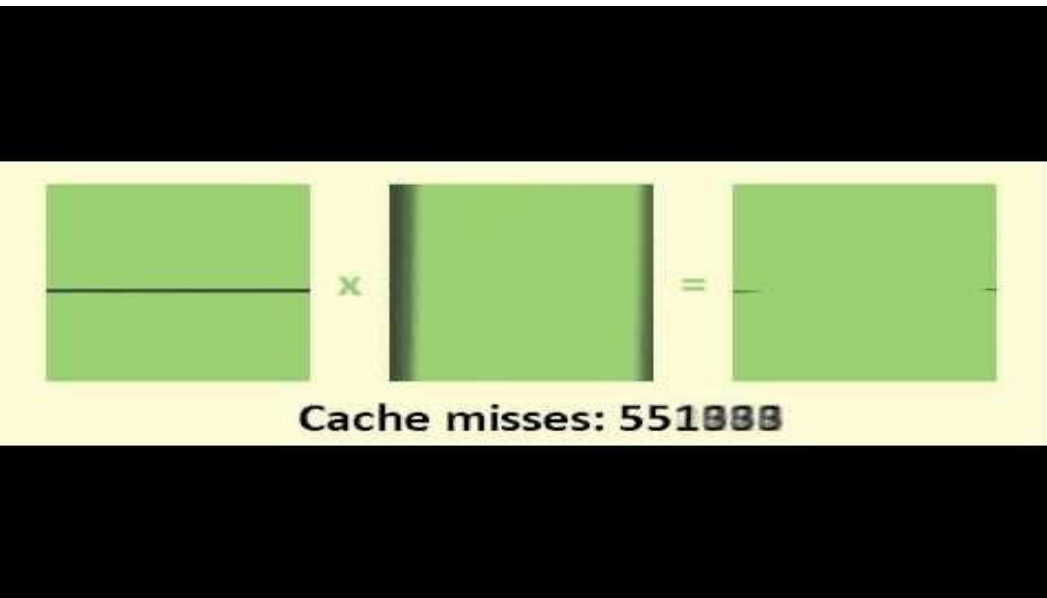
≈ 1,020,000
cache misses

Blocked:
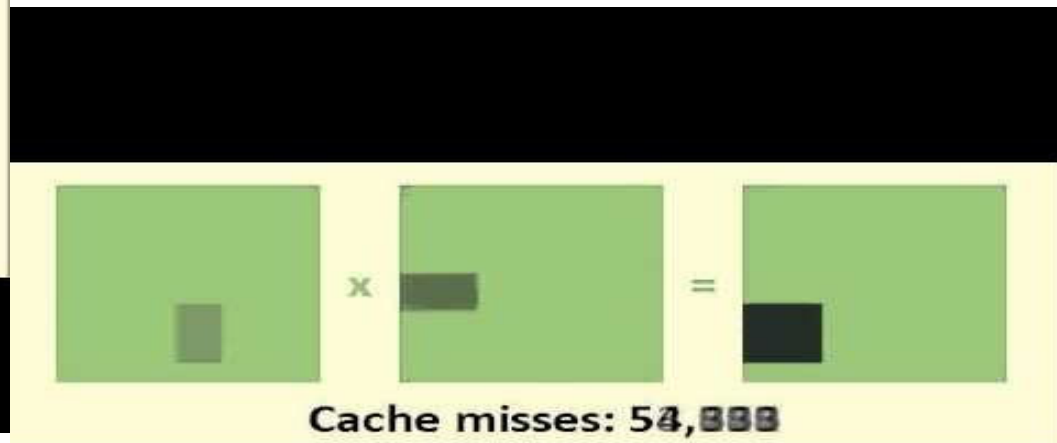
Cache misses: 54,888

≈ 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

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

Aggressive prefetching
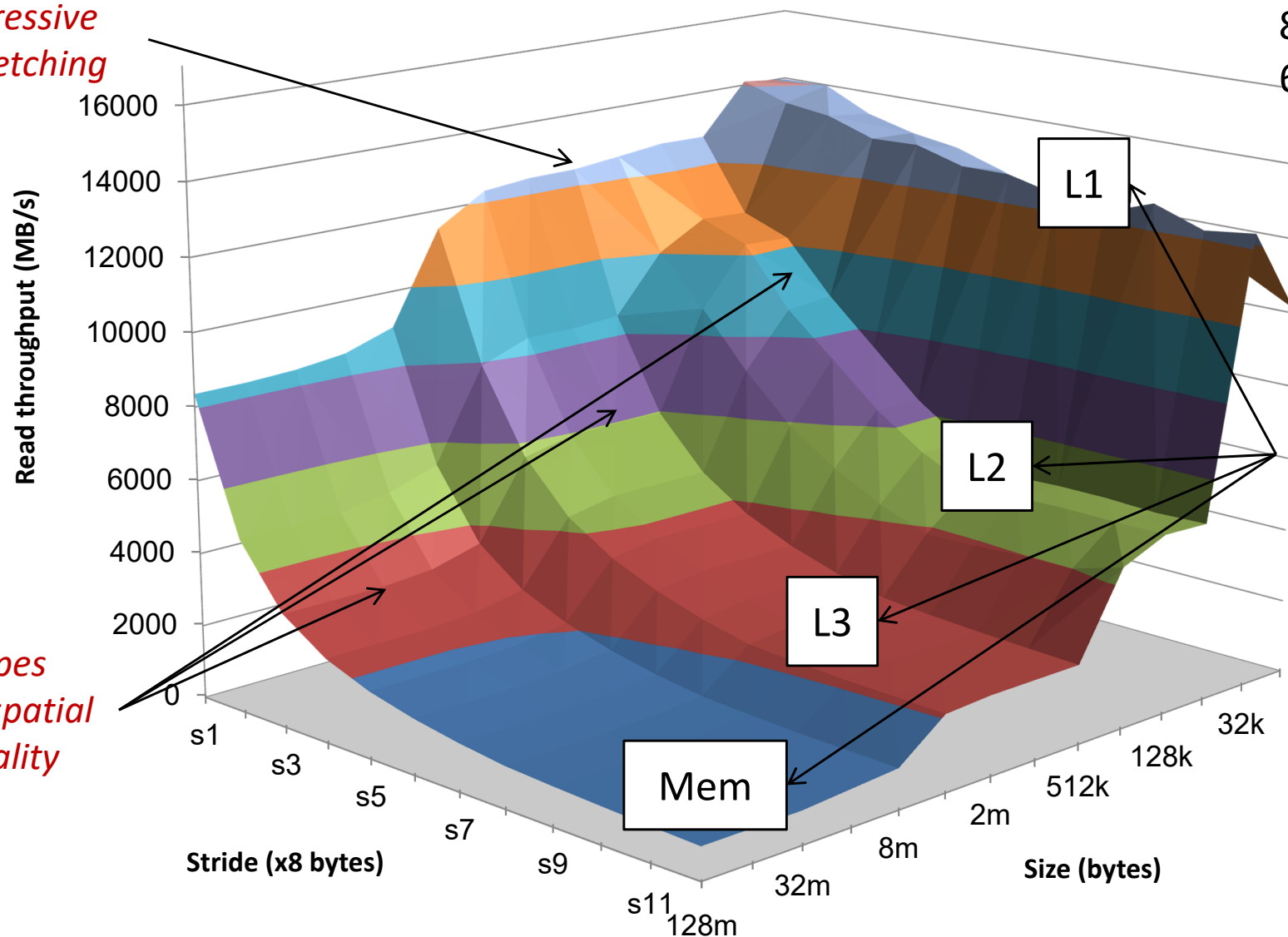
Ridges of temporal locality

Slopes of spatial locality

L1

L2

L3

Mem

Read throughput (MB/s)

16000
14000
12000
10000
8000
6000
4000
2000
0

Stride (x8 bytes)

s1
s3
s5
s7
s9
s11

Size (bytes)

32k
128k
512k
2m
8m
32m
128m

26

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/

UNIVERSITY *of* WASHINGTON

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();
```

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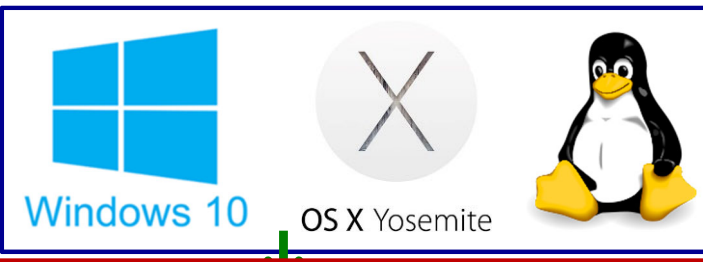
Assembly language:

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

OS:

Windows 10   OS X Yosemite

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

28

W UNIVERSITY *of* WASHINGTON

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*)

Physical control flow

time

<startup>
instr$_1$
instr$_2$
instr$_3$
...
instr$_n$
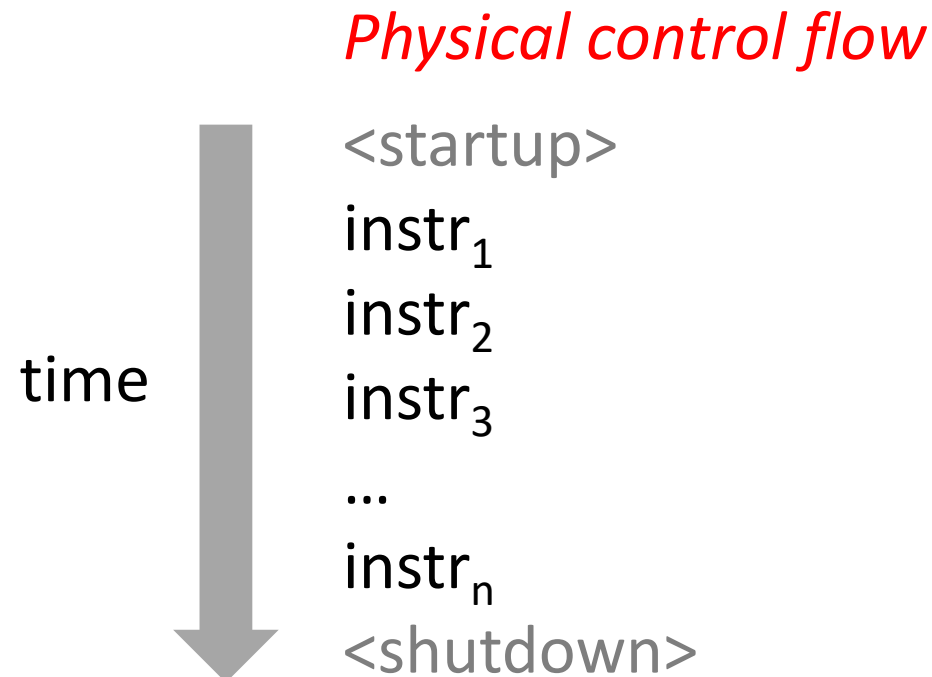
Altering the Control Flow

* 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

▪ <u>Examples</u>: division by 0, page fault, I/O request completes, Ctrl-C

User Code   *OS Kernel Code*

event ⟶ current_instr   *exception*
       next_instr

exception processing by
exception handler, then:
- *return to current_instr,*
- *return to next_instr, OR*
- *abort*
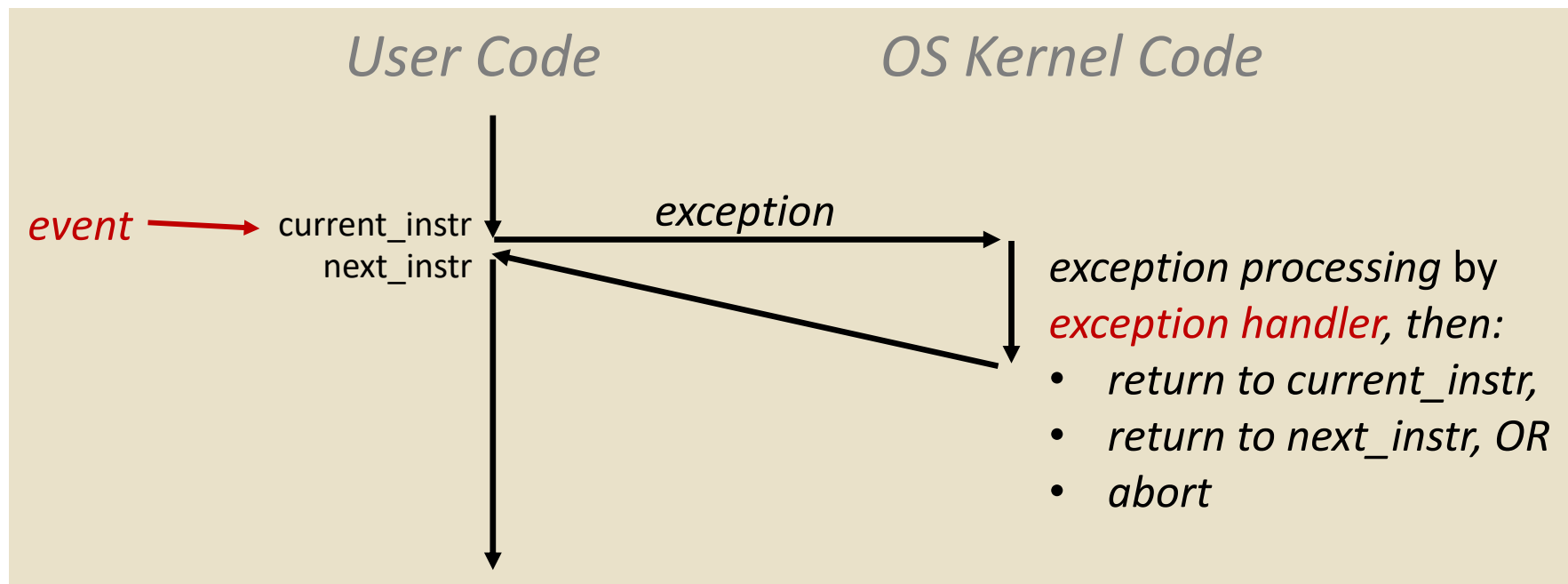
❖ *How does the system know where to jump to in the OS?*

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

Exception Table

0	●
1	●
2	●
...	
n-1	●

Exception numbers

code for exception handler 0

code for exception handler 1

code for exception handler 2

...

code for exception handler n-1
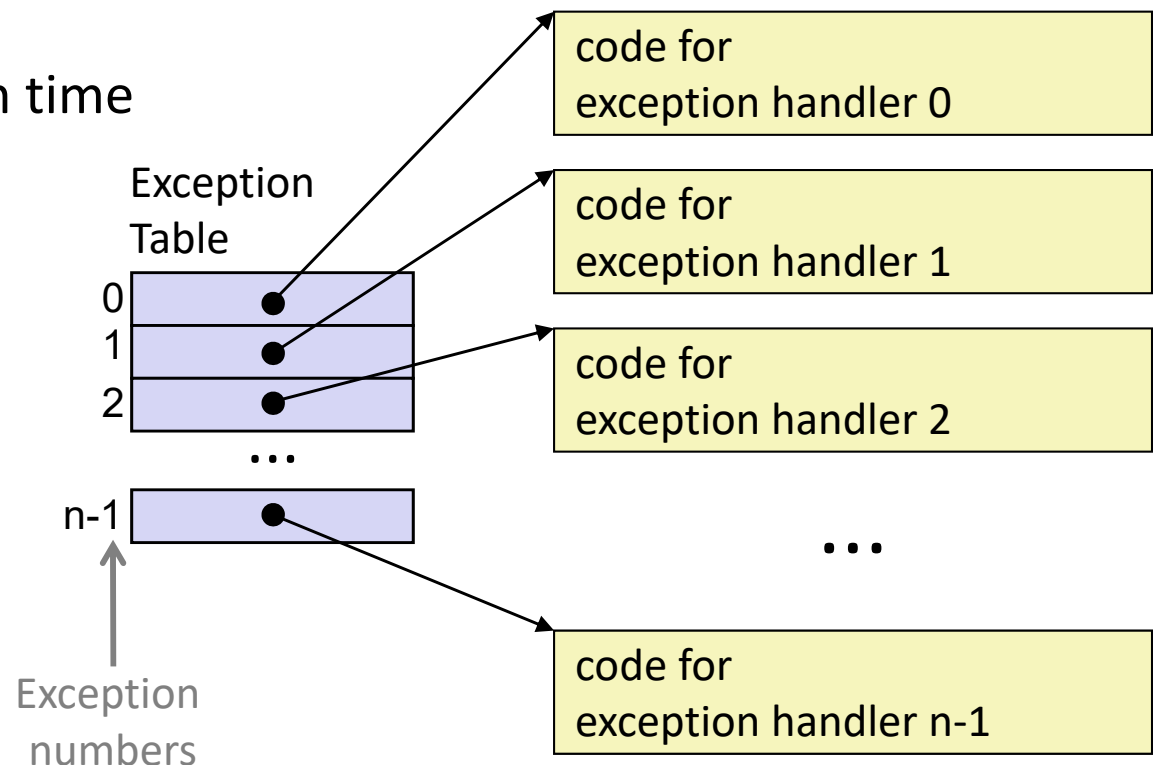
Exception Table (Excerpt)

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

❖ <u>Examples:</u>
  ▪ 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

- *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*
  - **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
```

User code   OS Kernel code
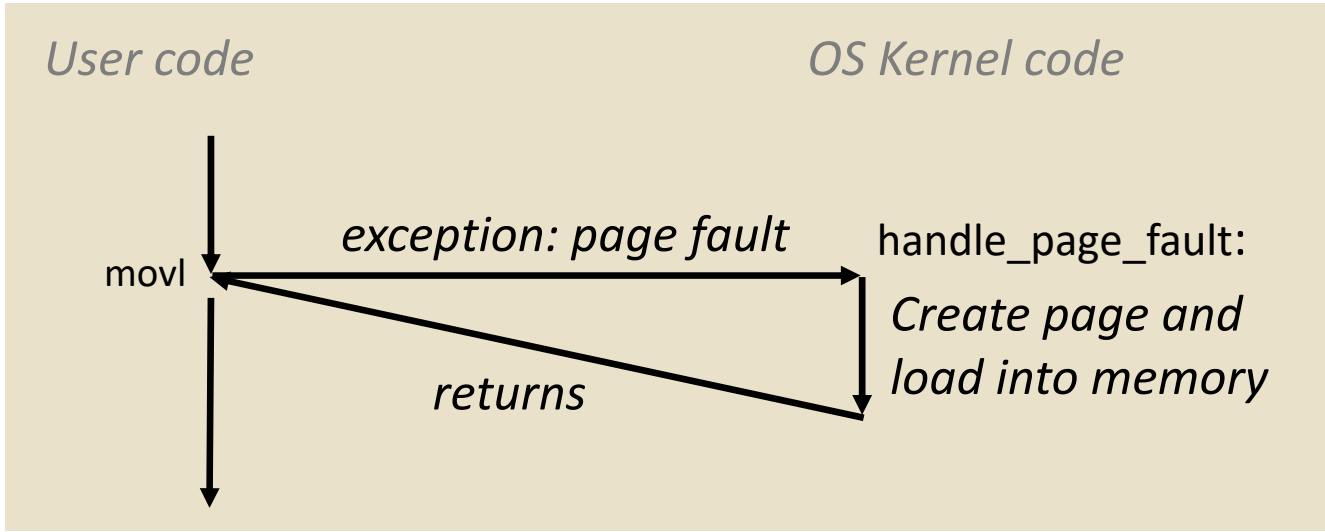
syscall
cmp

Exception

Open file
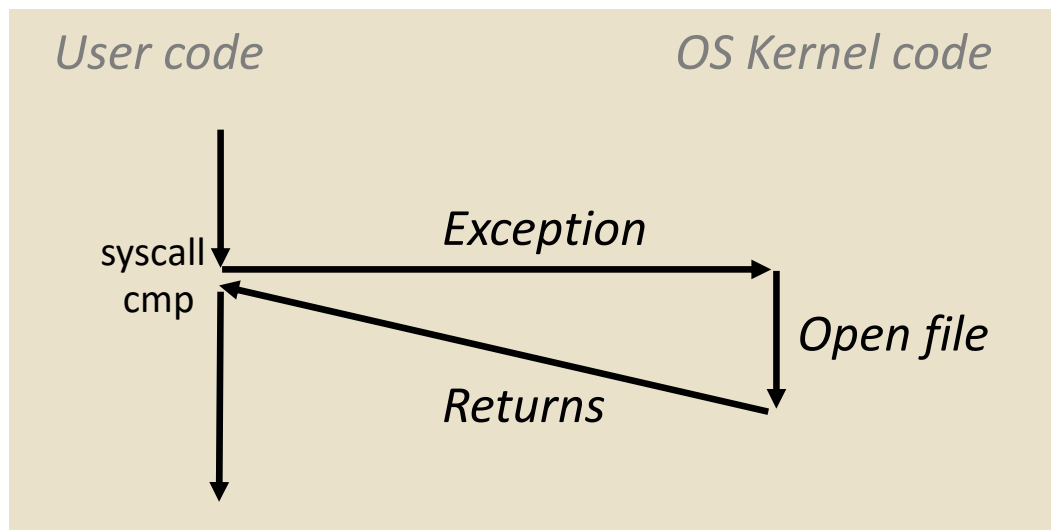
Returns

■ `%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

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

❖ User writes to memory location

❖ That portion (page) of user's memory is currently on disk

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

User code   *OS Kernel code*

movl ◄──── *exception: page fault* ────► handle_page_fault:
         ◄──── *returns* ────►   *Create page and*
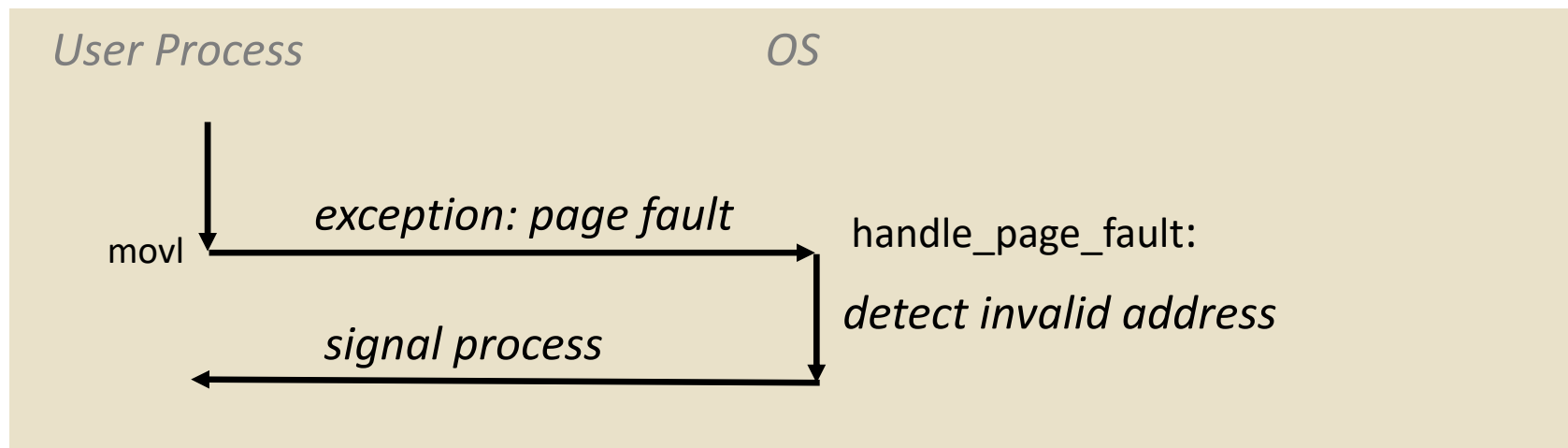   *load into memory*

❖ 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
```

User Process   OS

movl ⟵ *exception: page fault* ⟶ handle_page_fault:

detect invalid address

⟵ *signal process* ⟶

- ❖ 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