CSC 252: Computer Organization
Spring 2020: Lecture 18

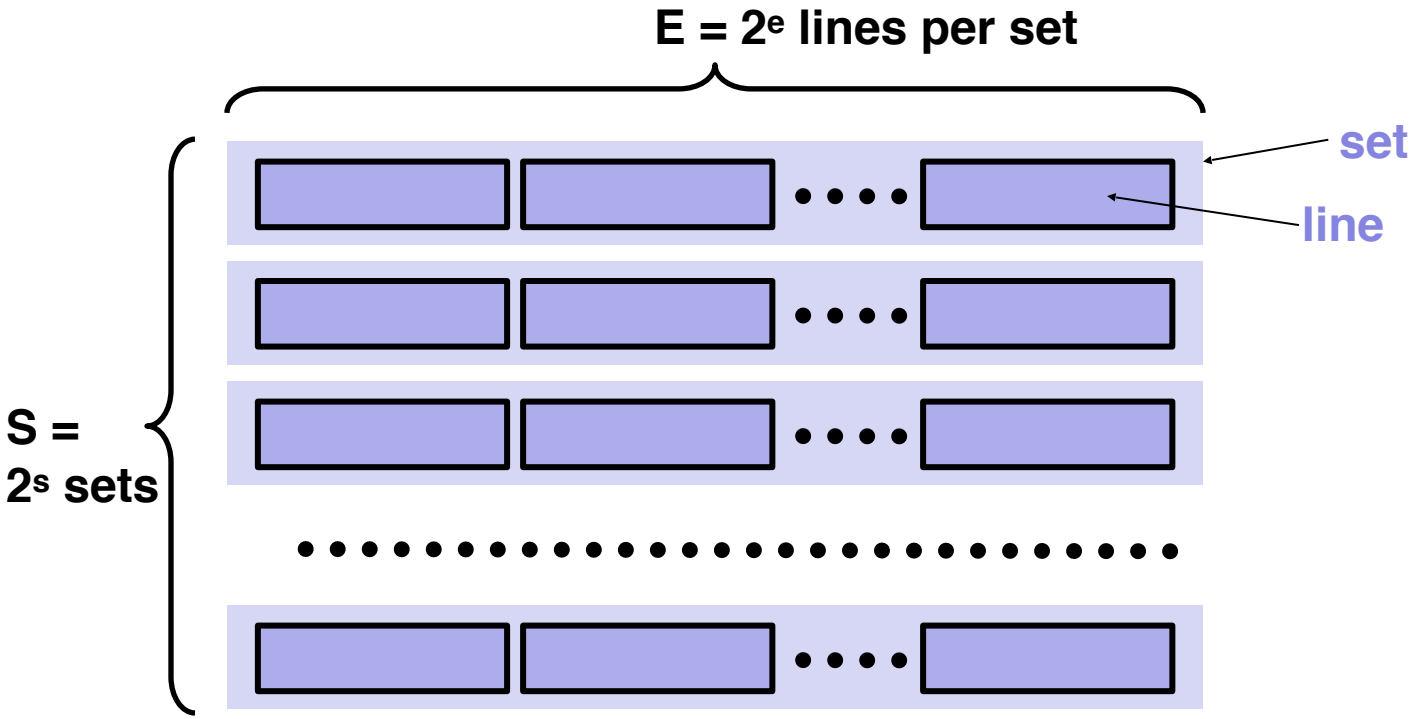
Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

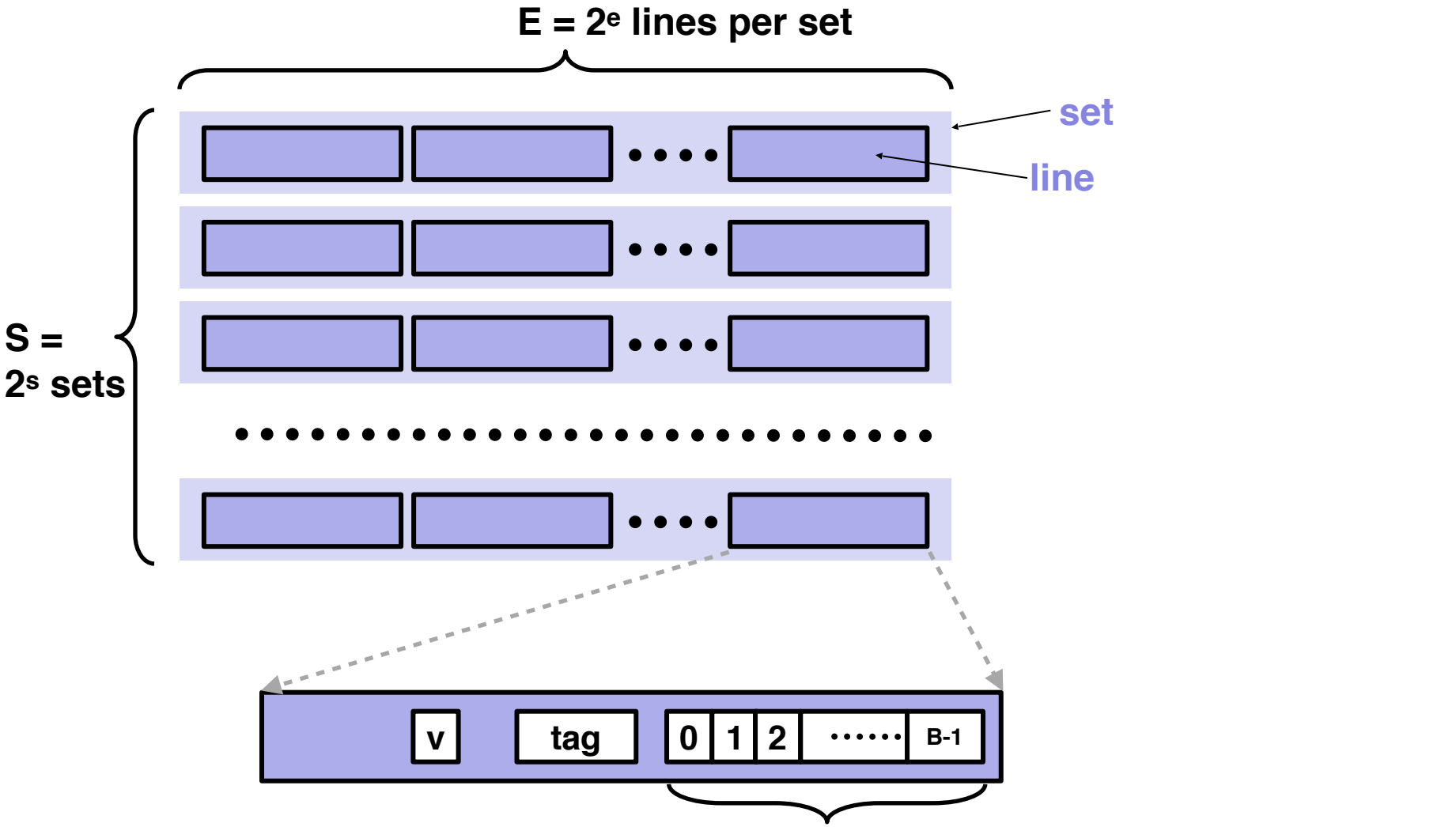
Announcements

- **Cache problem set: https://www.cs.rochester.edu/courses/252/spring2020/handouts.html**
- **Not to be turned in**
- **Assignment 4 soon to be released later today**

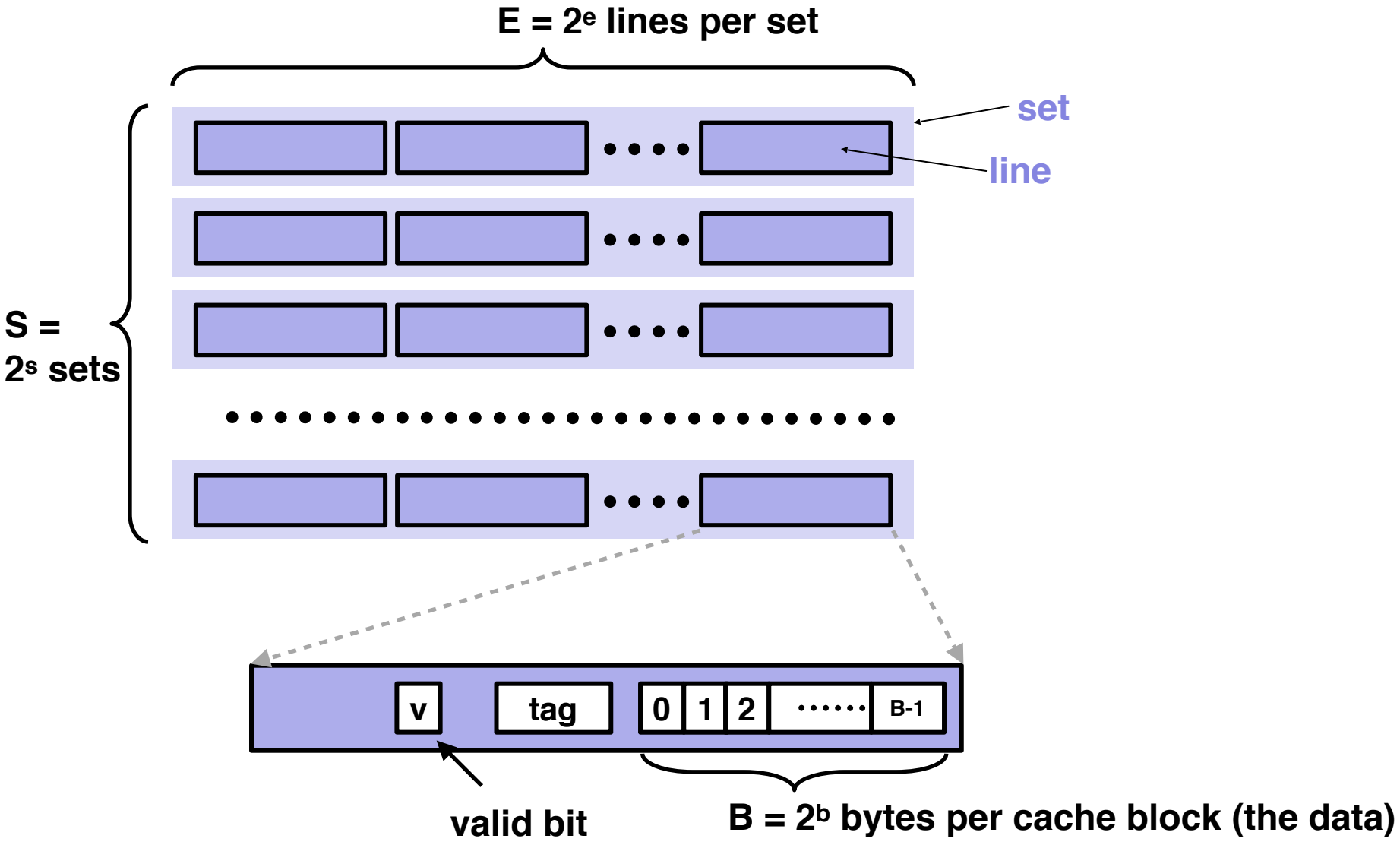
General Cache Organization (S, E, B)



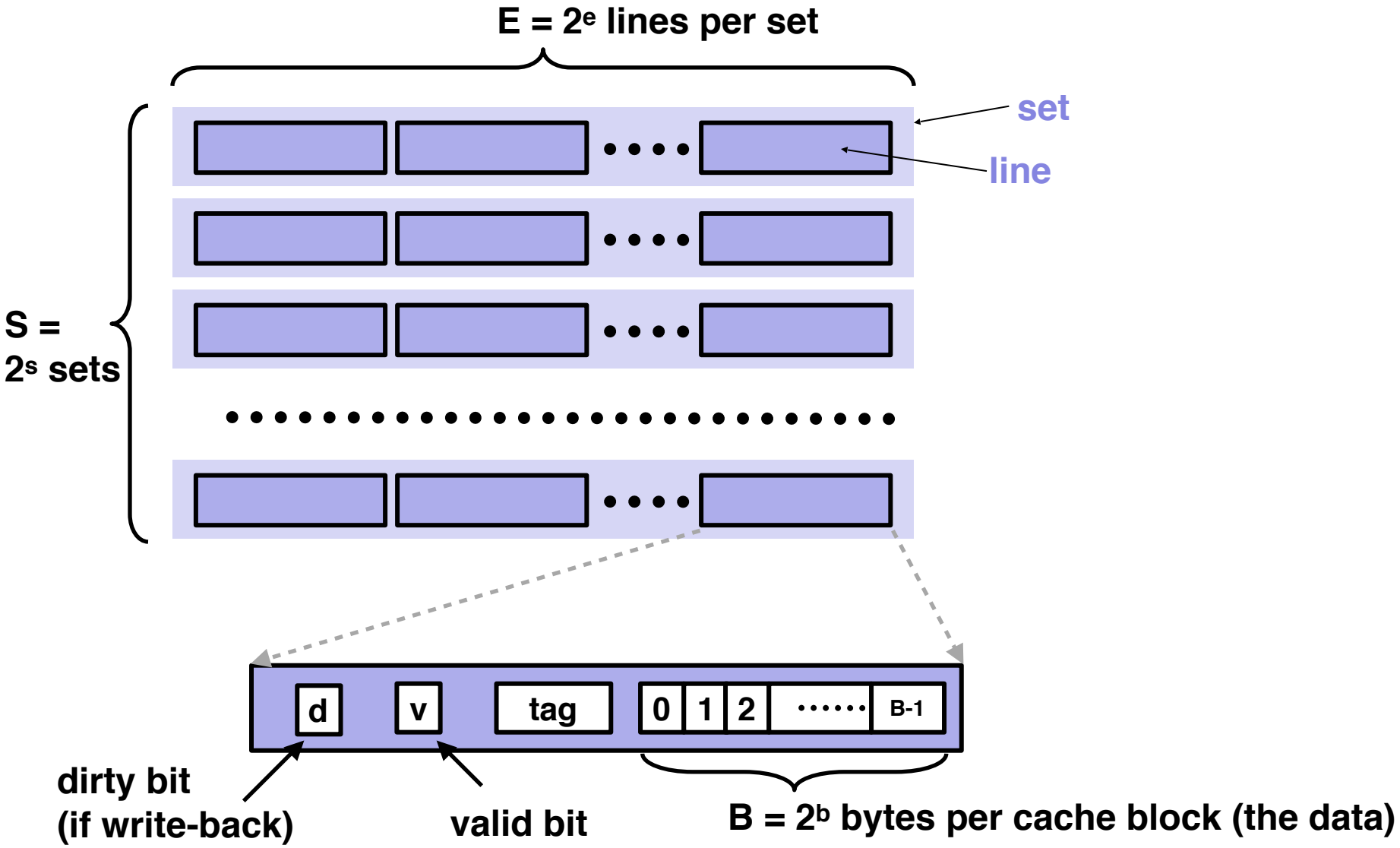
General Cache Organization (S, E, B)



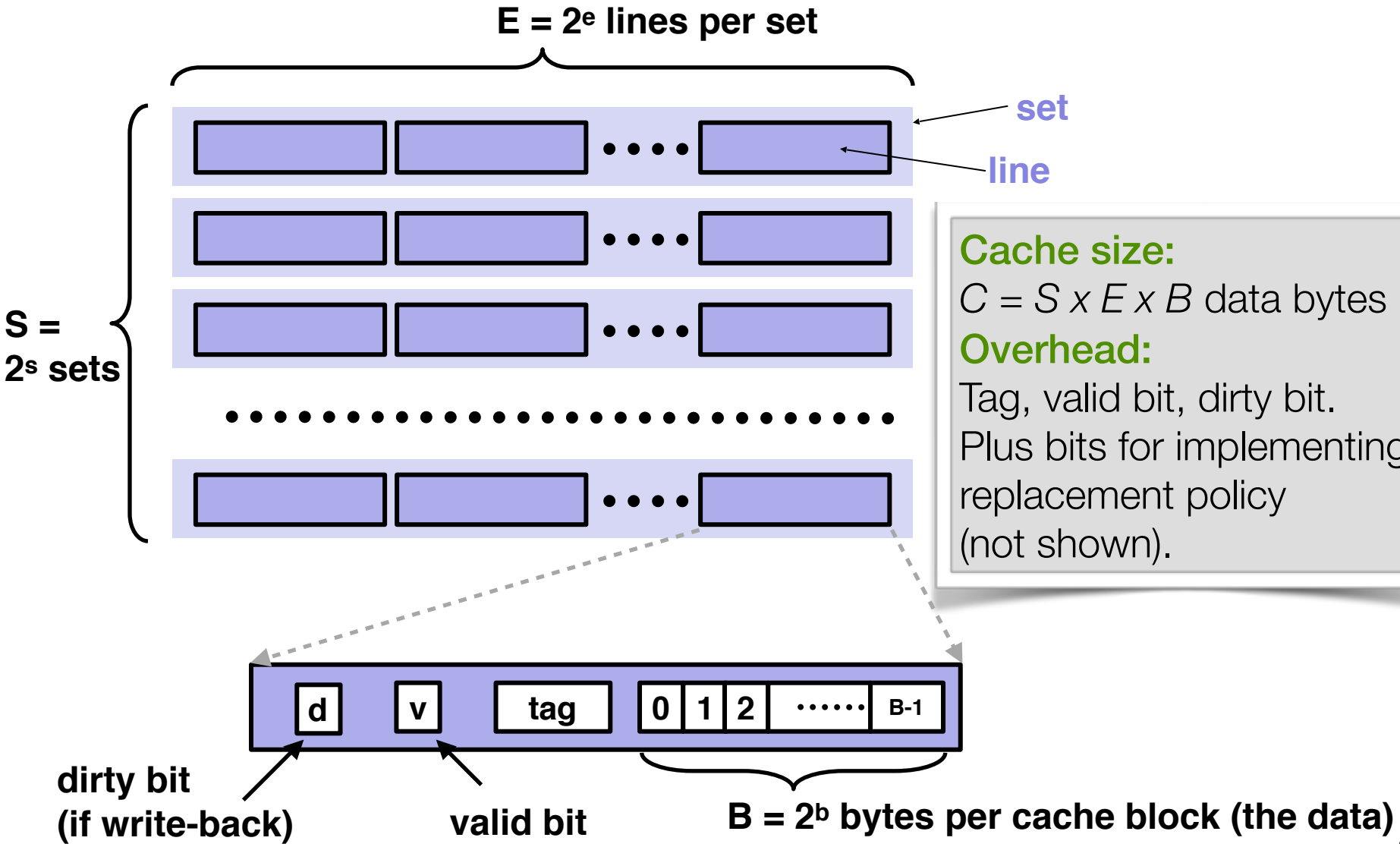
General Cache Organization (S, E, B)



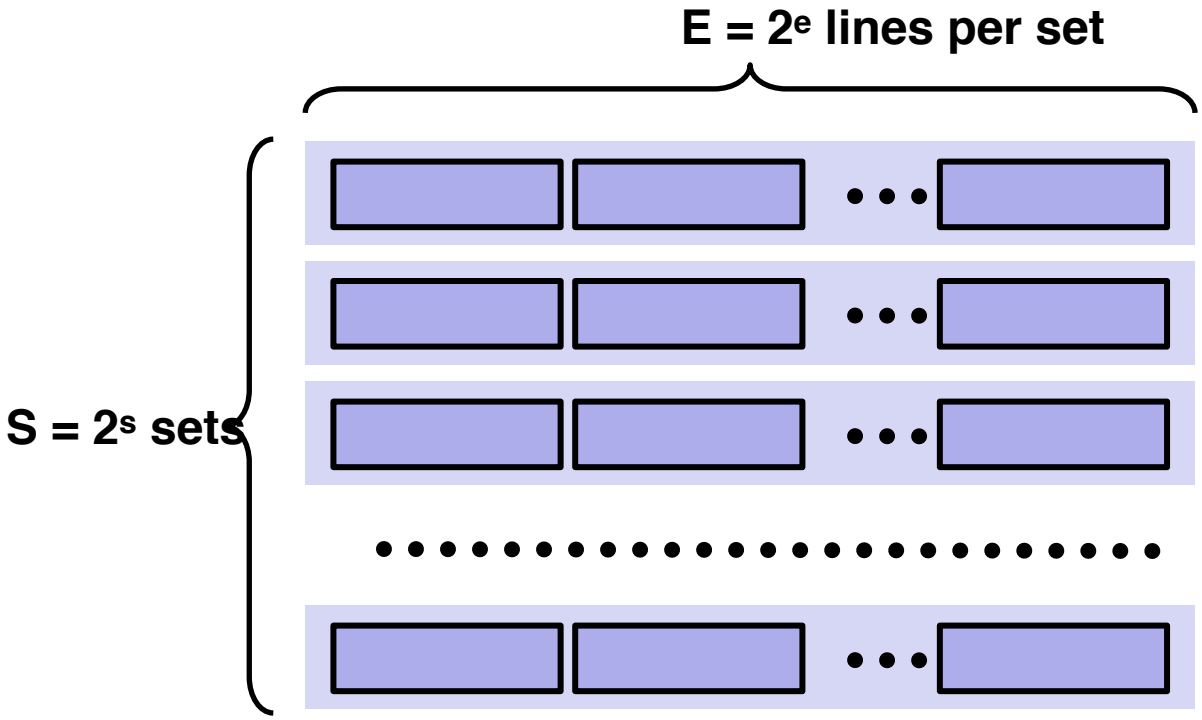
General Cache Organization (S, E, B)

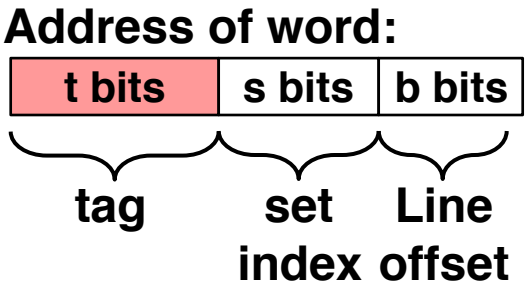


General Cache Organization (S, E, B)



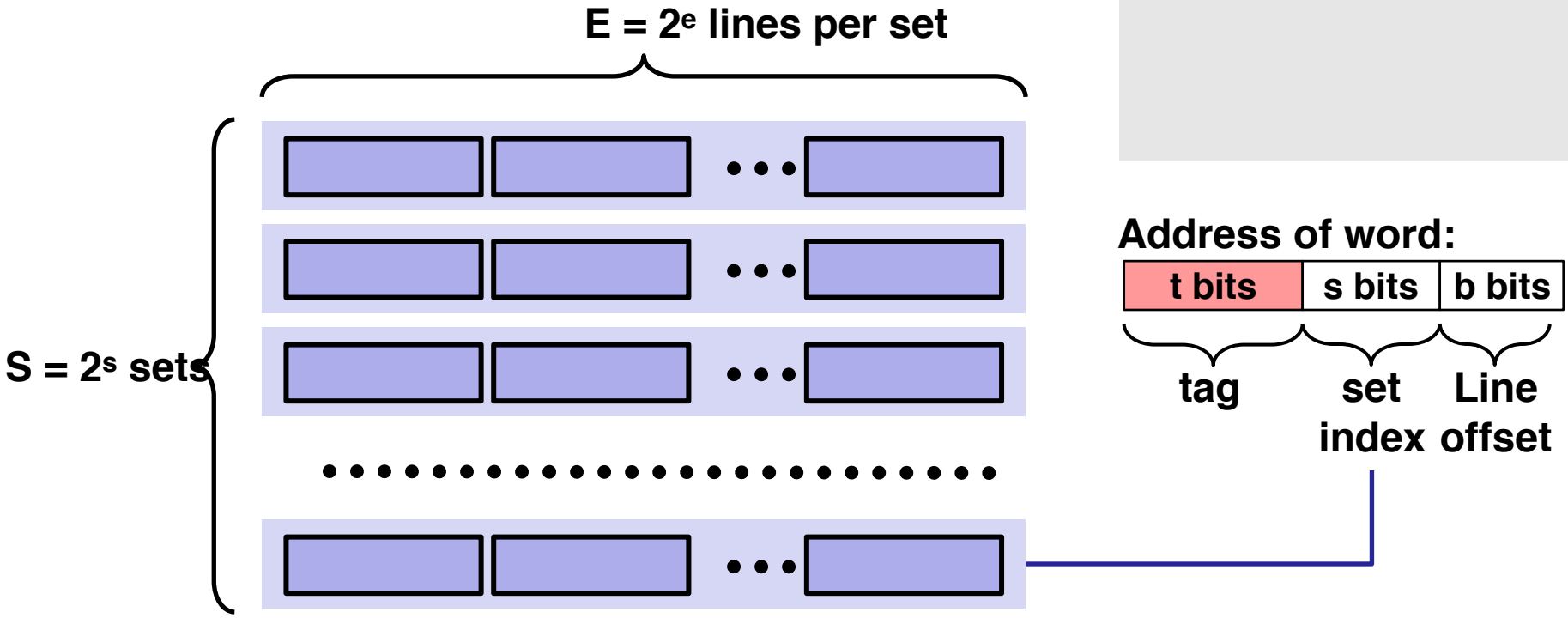
Cache Access





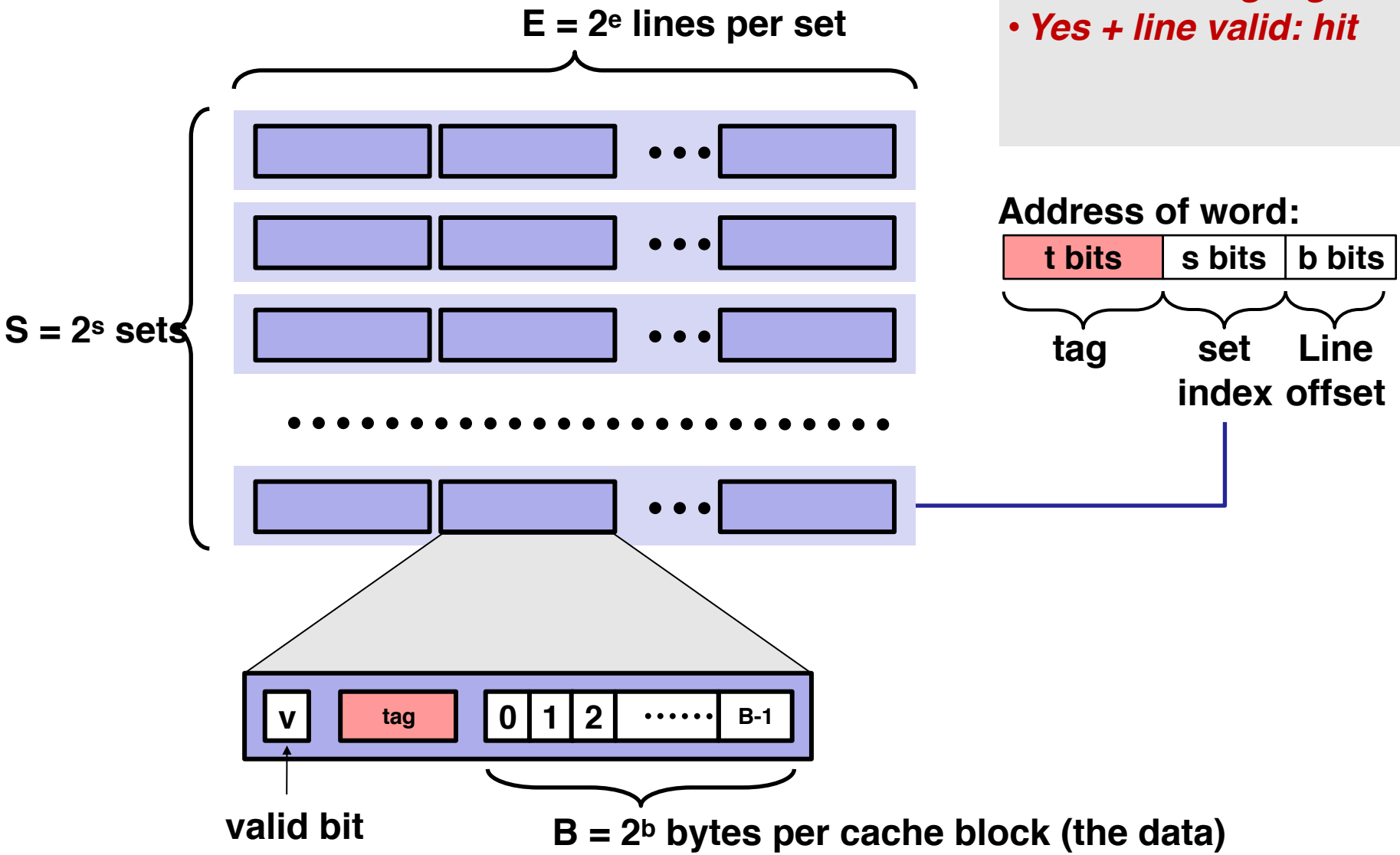
Cache Access

- *Locate set*



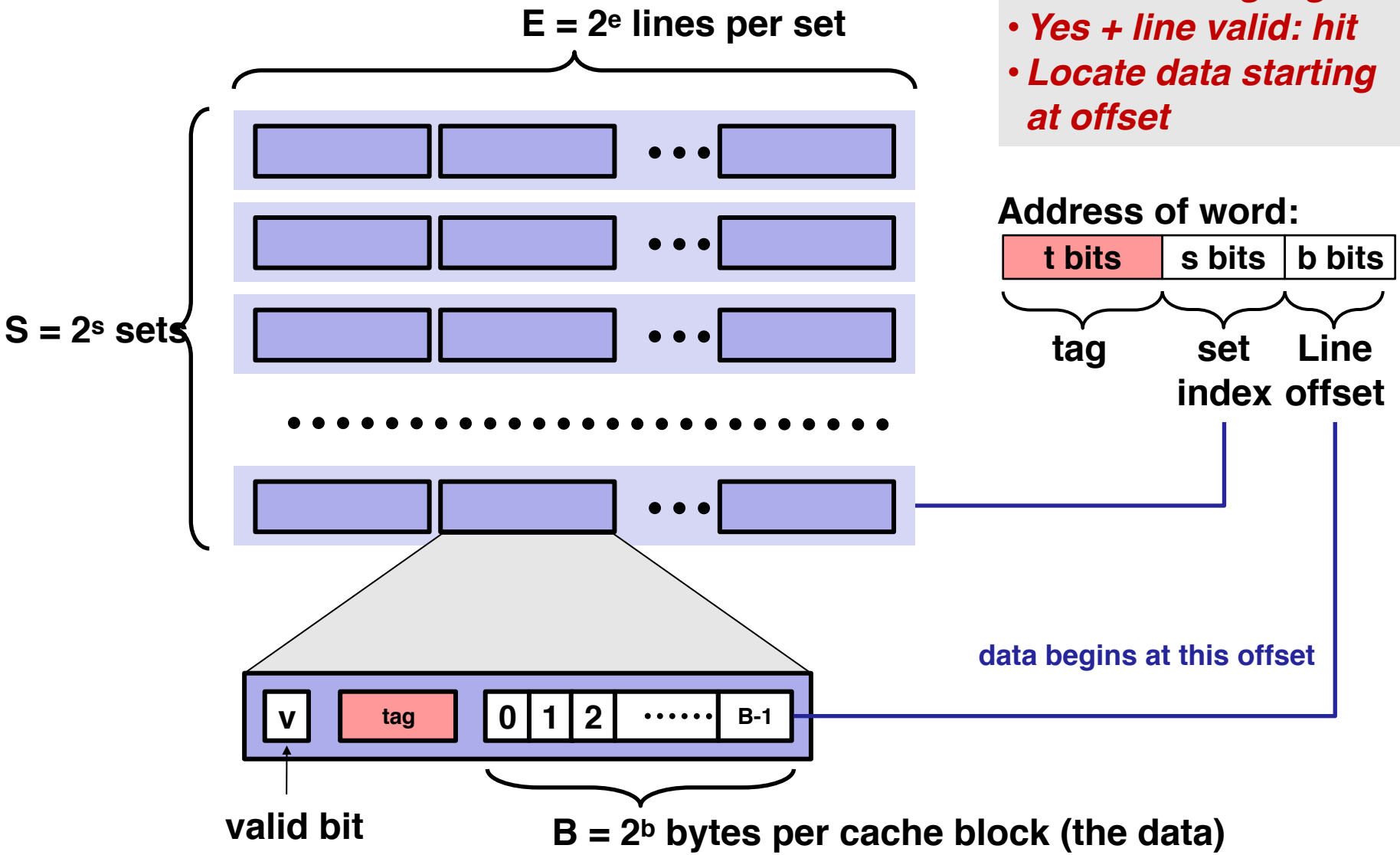
Cache Access

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*



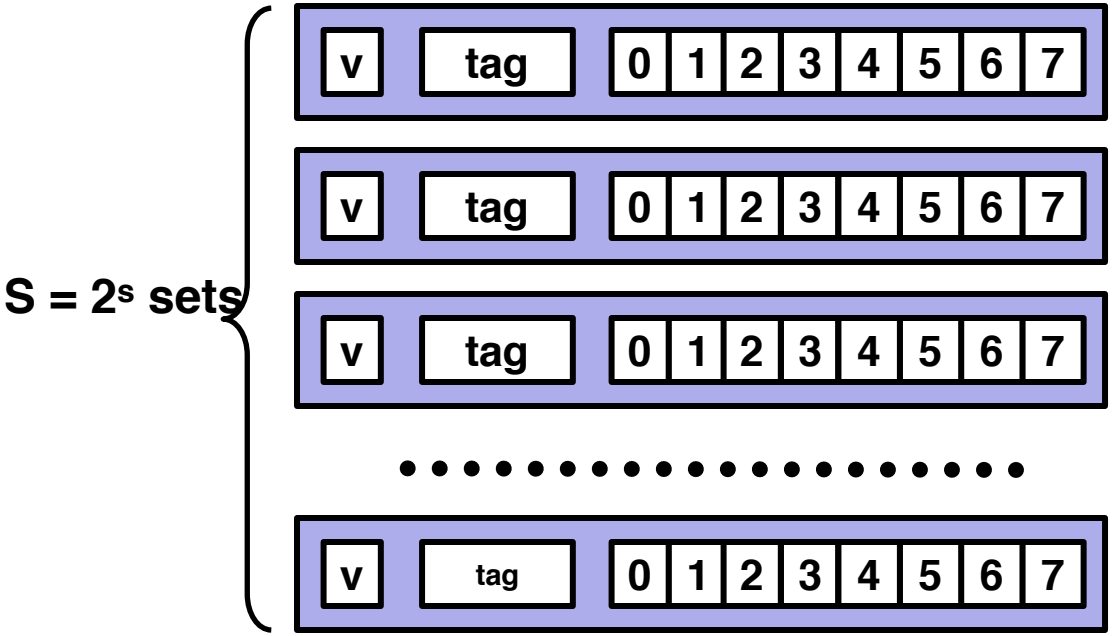
Cache Access

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes

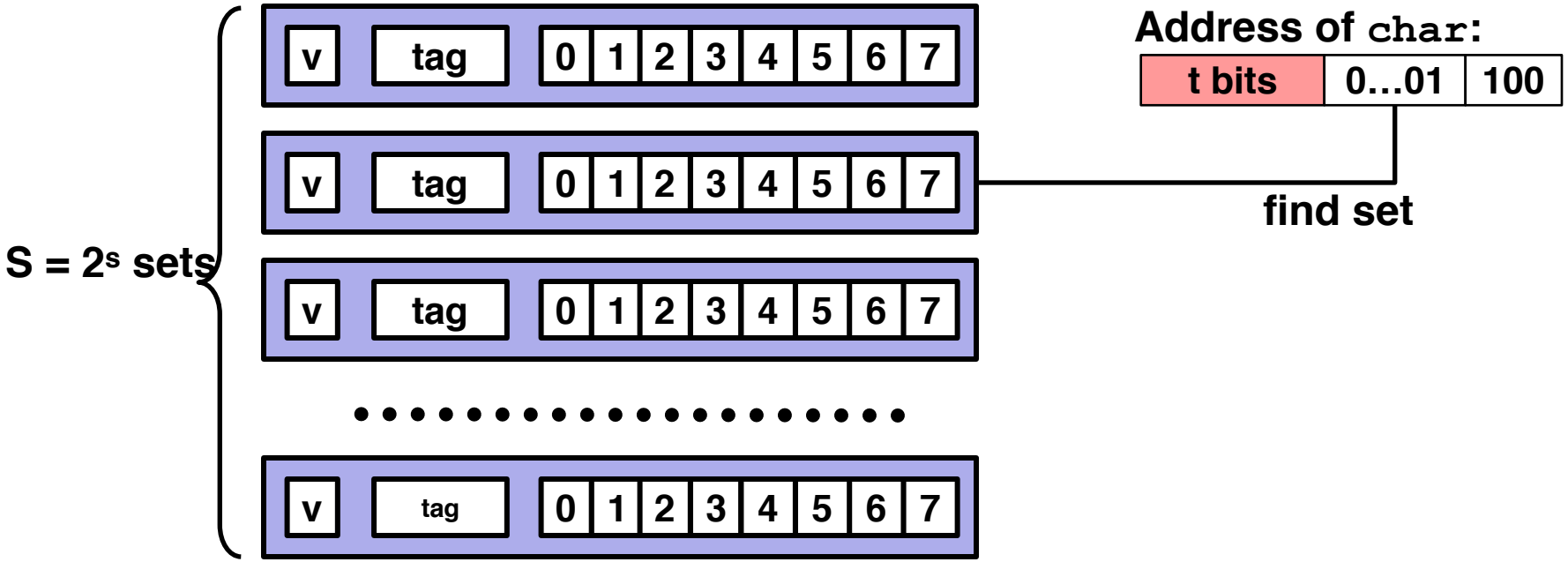


Address of `char`:

t bits	0…01	100

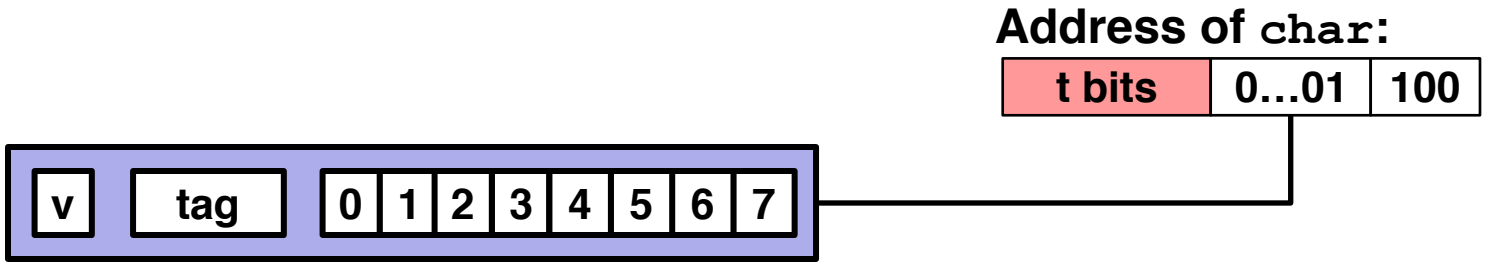
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes



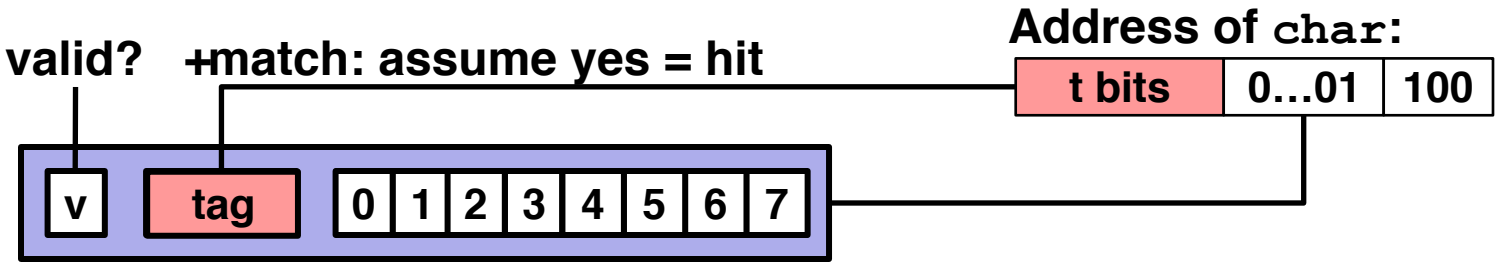
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes



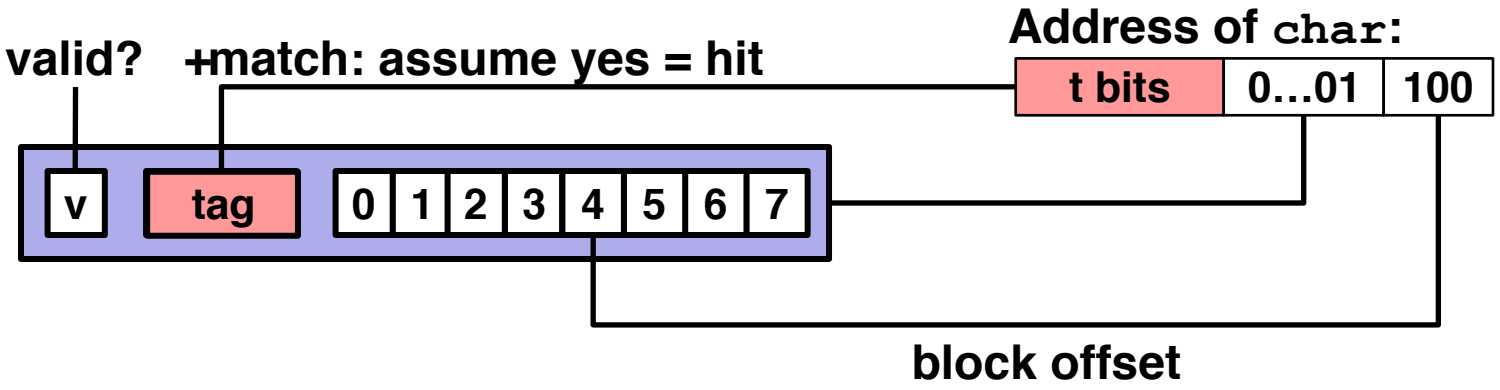
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes



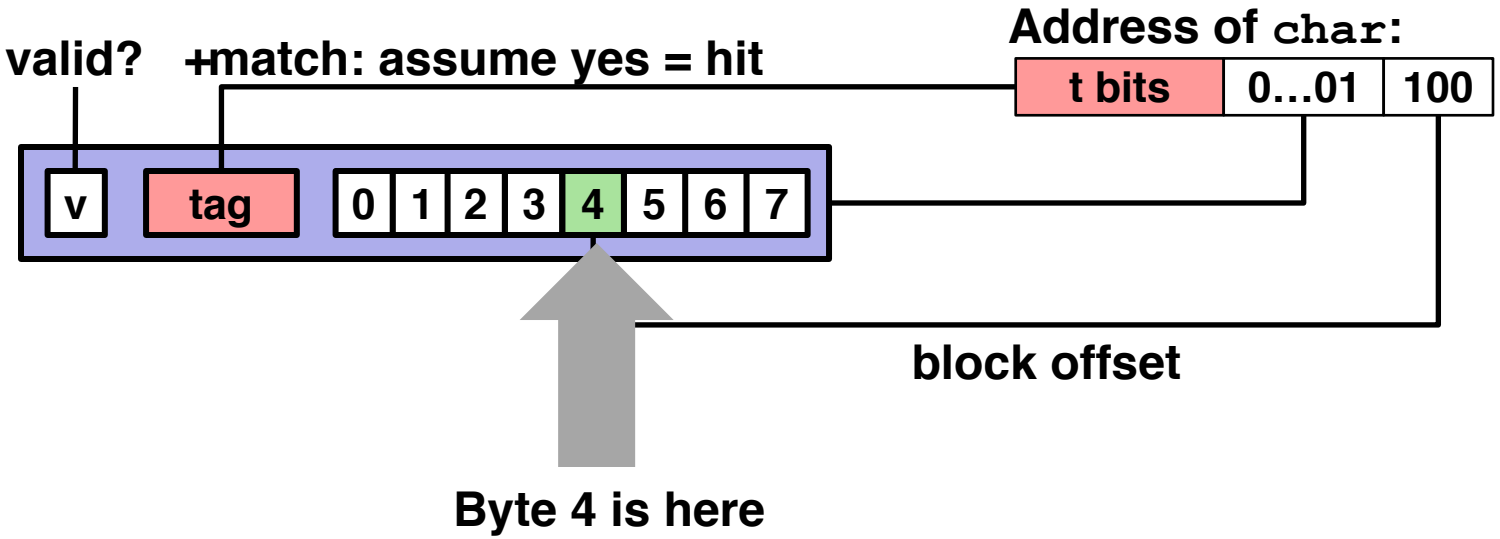
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes



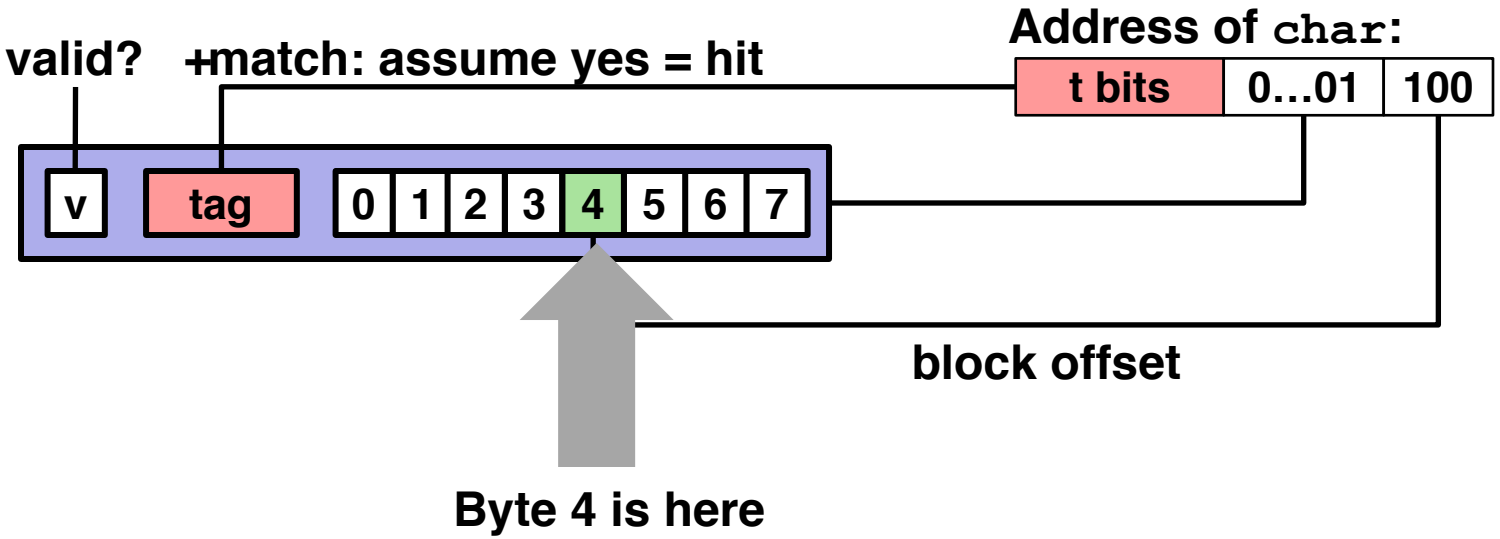
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes



Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache block size 8 bytes



If tag doesn't match: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0 **[0$\underline{00}$0$_2$],**
1 **[0$\underline{00}$1$_2$],**
7 **[0$\underline{11}$1$_2$],**
8 **[1$\underline{00}$0$_2$],**
0 **[0$\underline{00}$0$_2$]**

	v	Tag	Line
Set 0	0	?	?
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	
7	**[0$\underline{11}$1$_2$],**	
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	0	?	?
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	
7	**[0$\underline{11}$1$_2$],**	
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	
7	**[0$\underline{11}$1$_2$],**	
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3			

← The two bytes at memory address 0 and 1

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line	
Set 0	1	0	M[0-1]	← The two bytes at memory address 0 and 1
Set 1				
Set 2				
Set 3				

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3			

← The two bytes at memory address 0 and 1

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line	
Set 0	1	0	M[0-1]	← The two bytes at memory address 0 and 1
Set 1				
Set 2				
Set 3	1	0	M[6-7]	

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line	
Set 0	1	0	M[0-1]	← The two bytes at memory address 0 and 1
Set 1				
Set 2				
Set 3	1	0	M[6-7]	← The two bytes at memory address 6 and 7

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line	
Set 0	1	0	M[0-1]	← The two bytes at memory address 0 and 1
Set 1				
Set 2				
Set 3	1	0	M[6-7]	← The two bytes at memory address 6 and 7

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{000}$0$_2$],**	miss
1	**[0$\underline{000}$1$_2$],**	hit
7	**[0$\underline{111}$1$_2$],**	miss
8	**[1$\underline{000}$0$_2$],**	miss
0	**[0$\underline{000}$0$_2$]**	

	v	Tag	Line	
Set 0	1	1	M[8-9]	← The two bytes at memory address 8 and 9
Set 1				
Set 2				
Set 3	1	0	M[6-7]	← The two bytes at memory address 6 and 7

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	miss

	v	Tag	Line	
Set 0	1	1	M[8-9]	← The two bytes at memory address 8 and 9
Set 1				
Set 2				
Set 3	1	0	M[6-7]	← The two bytes at memory address 6 and 7

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	miss

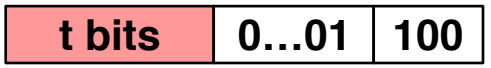
	v	Tag	Line	
Set 0	1	0	M[0-1]	← The two bytes at memory address 0 and 1
Set 1				
Set 2				
Set 3	1	0	M[6-7]	← The two bytes at memory address 6 and 7

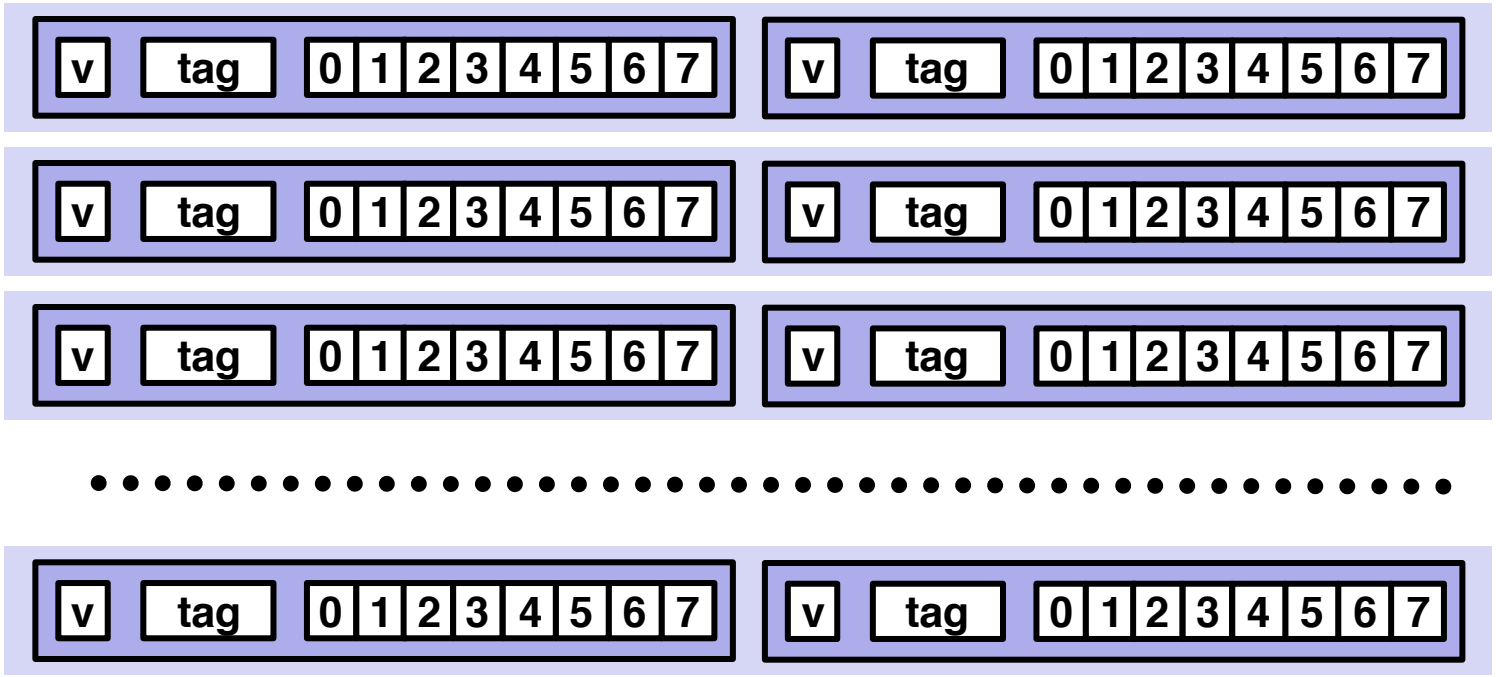
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of `short int`:

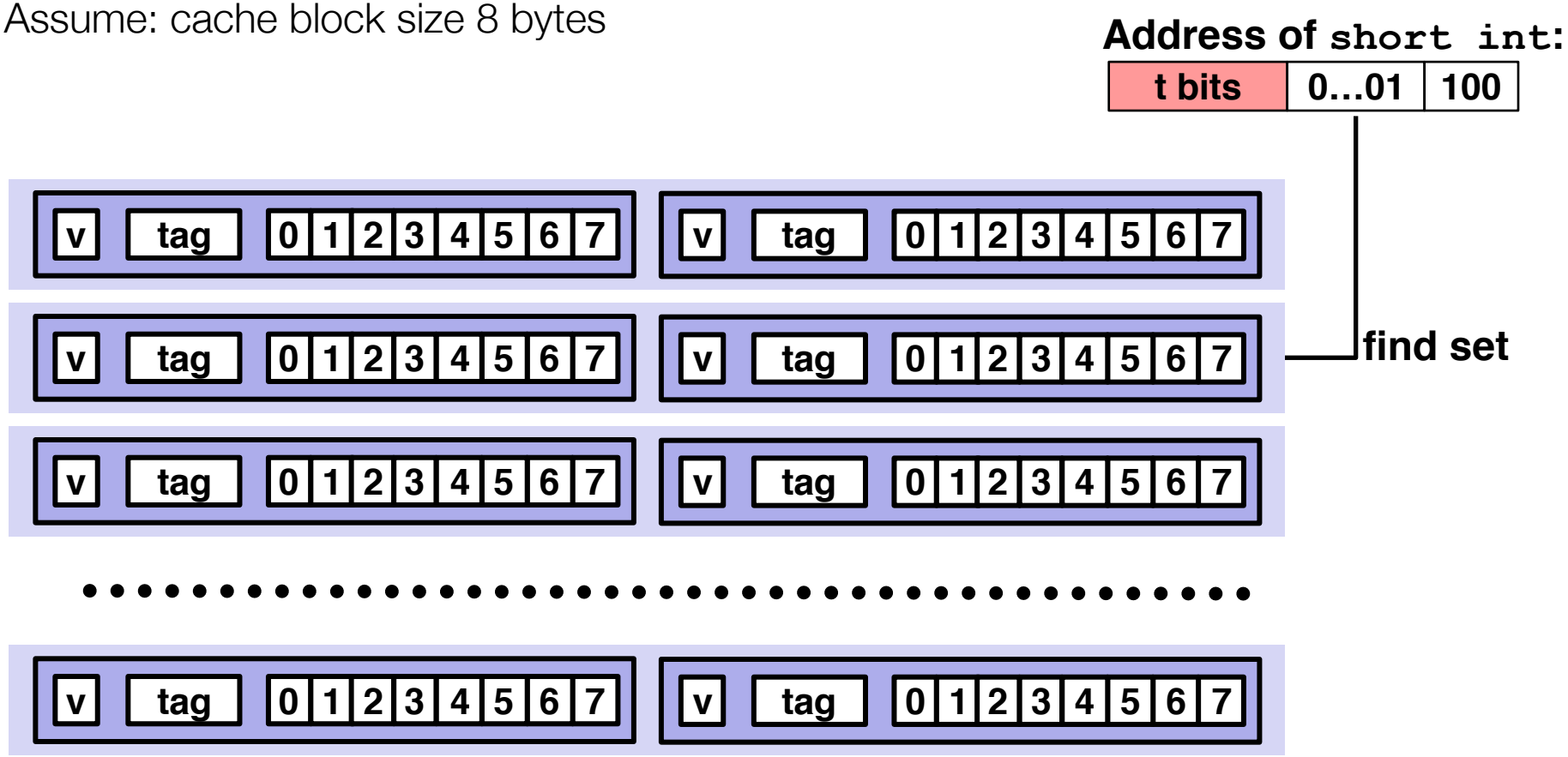




E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

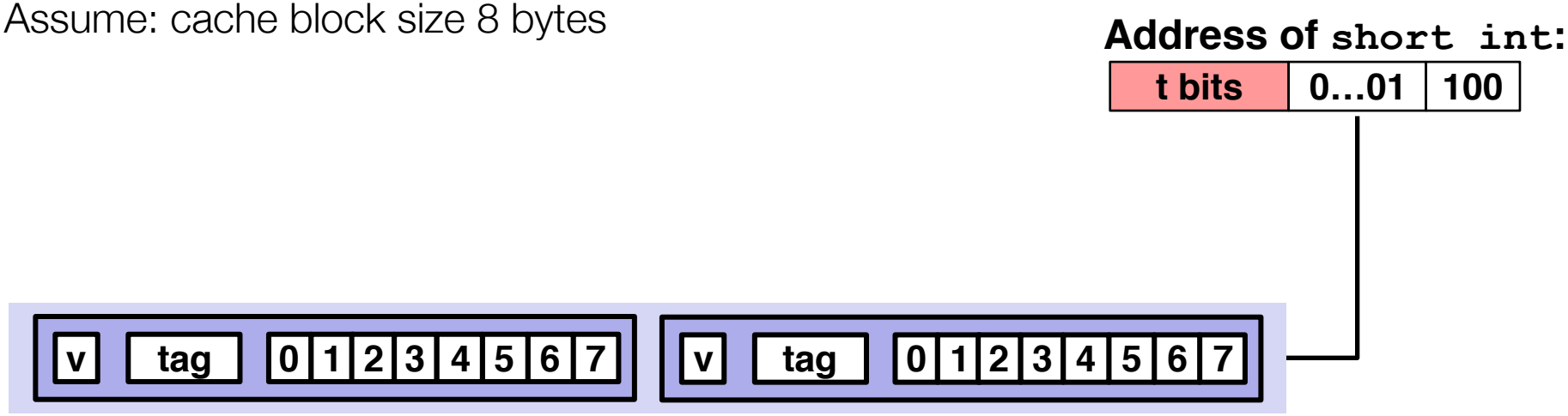
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

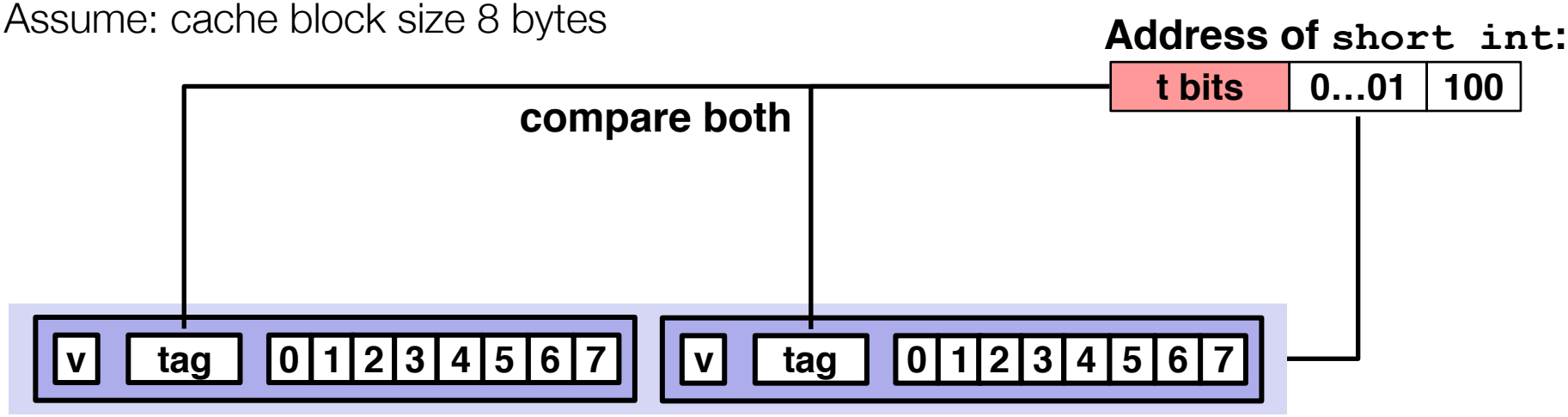
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

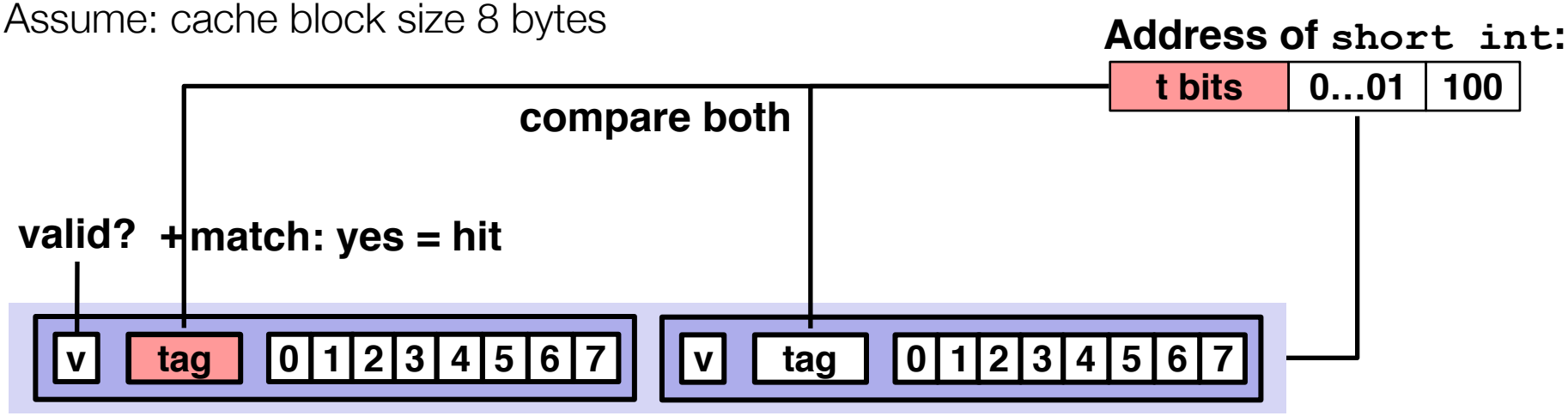
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

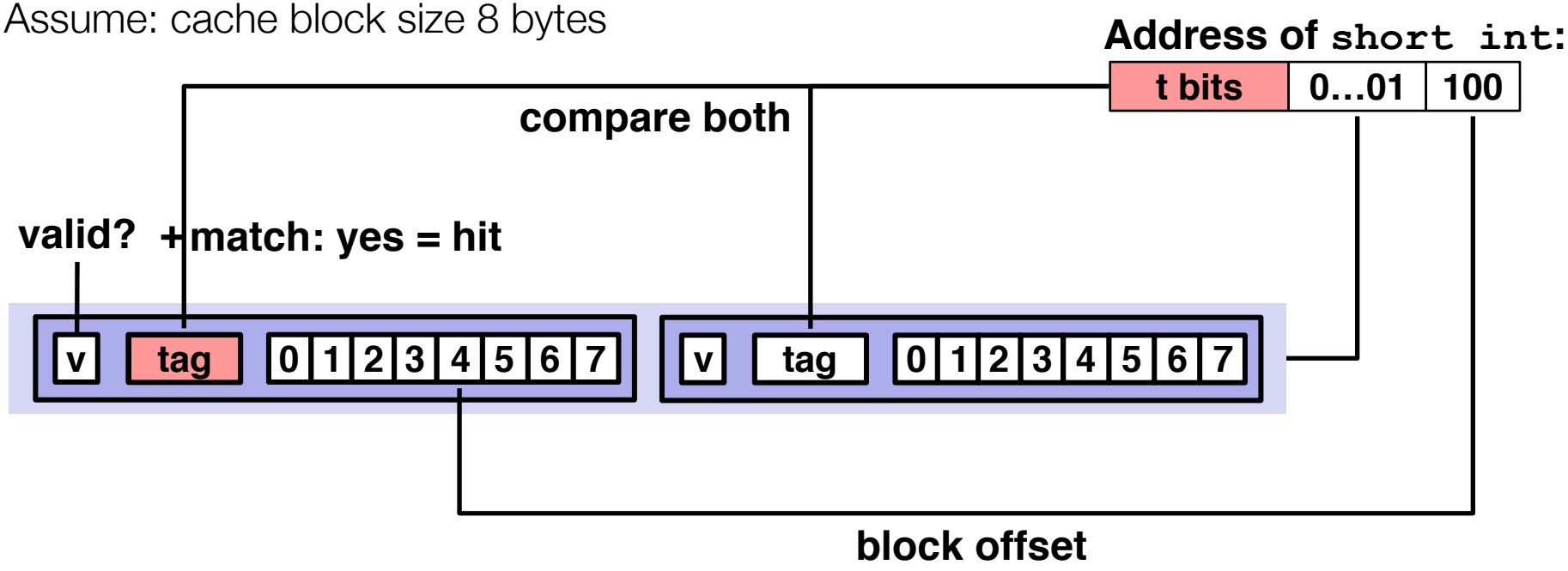
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

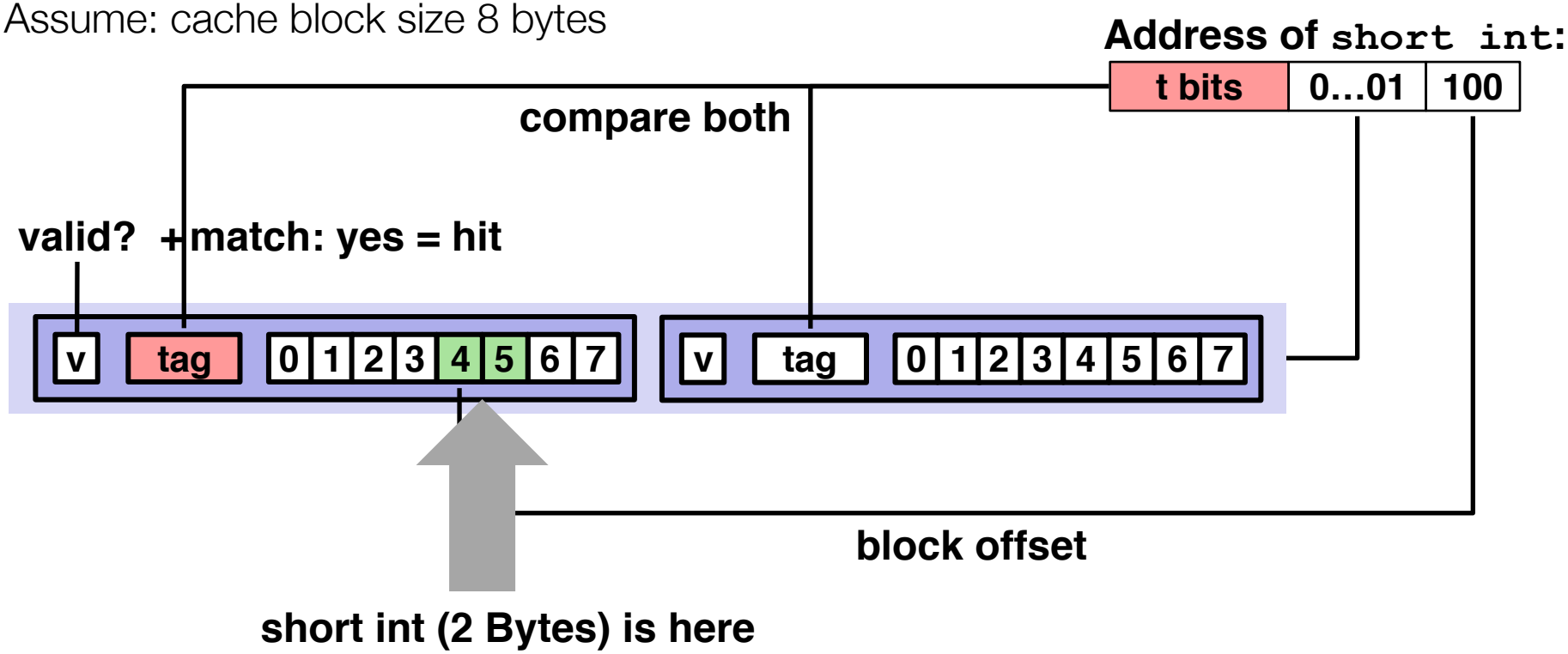
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

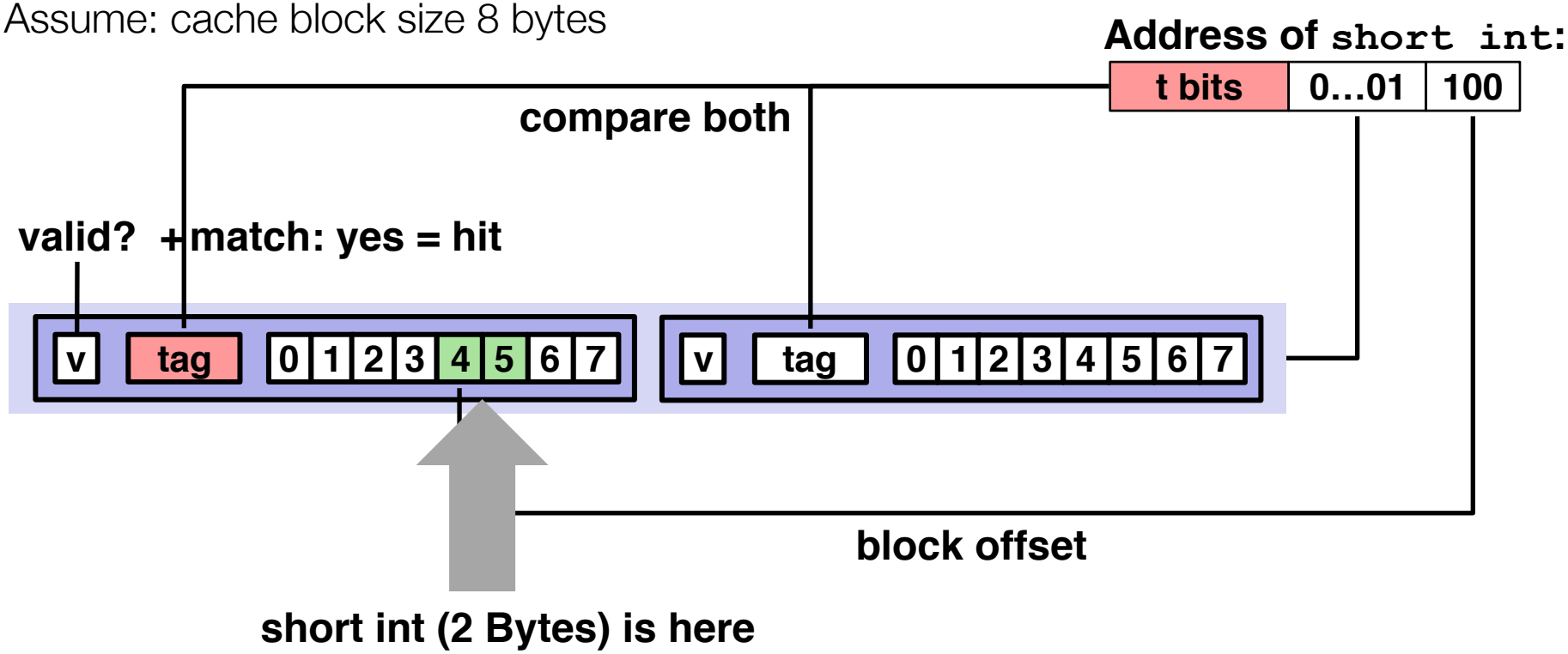
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



No match:

- **One line in set is selected for eviction and replacement**
- **Replacement policies: random, least recently used (LRU),**

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0 **[0000$_2$],**
1 **[0001$_2$],**
7 **[0111$_2$],**
8 **[1000$_2$],**
0 **[0000$_2$]**

	v	Tag	Block
Set 0	0	?	?
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	
7	**[01$\underline{1}$1$_2$],**	
8	**[10$\underline{0}$0$_2$],**	
0	**[00$\underline{0}$0$_2$]**	

	v	Tag	Block
Set 0	0	?	?
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	
7	**[01$\underline{1}$1$_2$],**	
8	**[10$\underline{0}$0$_2$],**	
0	**[00$\underline{0}$0$_2$]**	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	hit
7	**[01$\underline{1}$1$_2$],**	
8	**[10$\underline{0}$0$_2$],**	
0	**[00$\underline{0}$0$_2$]**	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	hit
7	**[01$\underline{1}$1$_2$],**	miss
8	**[10$\underline{0}$0$_2$],**	
0	**[00$\underline{0}$0$_2$]**	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	hit
7	**[01$\underline{1}$1$_2$],**	miss
8	**[10$\underline{0}$0$_2$],**	
0	**[00$\underline{0}$0$_2$]**	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	1	01	M[6-7]
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	hit
7	**[01$\underline{1}$1$_2$],**	miss
8	**[10$\underline{0}$0$_2$],**	miss
0	**[00$\underline{0}$0$_2$]**	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	1	01	M[6-7]
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	hit
7	**[01$\underline{1}$1$_2$],**	miss
8	**[10$\underline{0}$0$_2$],**	miss
0	**[00$\underline{0}$0$_2$]**	

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

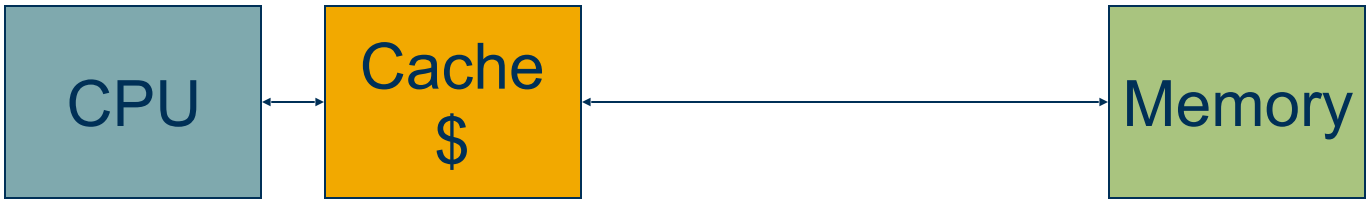
4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	hit
7	**[01$\underline{1}$1$_2$],**	miss
8	**[10$\underline{0}$0$_2$],**	miss
0	**[00$\underline{0}$0$_2$]**	hit

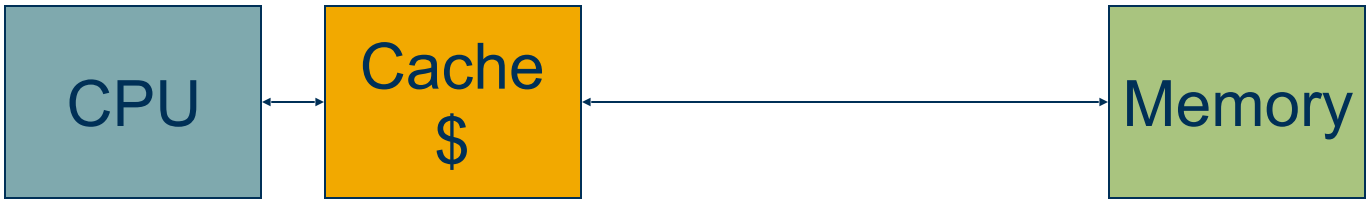
	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

General Rule: Bigger == Slower



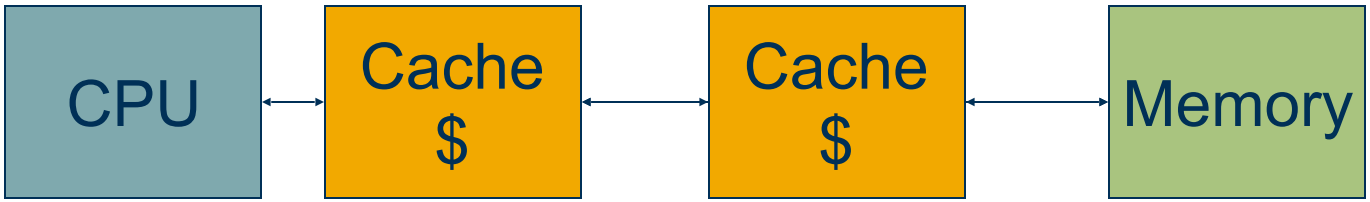
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)

General Rule: Bigger == Slower



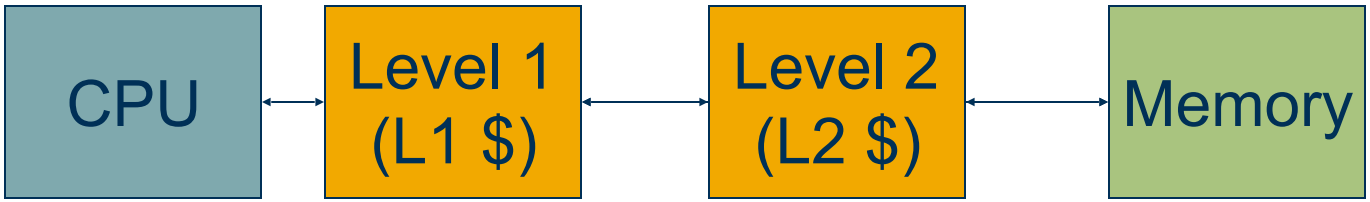
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)
- Make multiple levels of cache
  - Small L1 backed up by larger L2
  - Today's processors typically have 3 cache levels

General Rule: Bigger == Slower



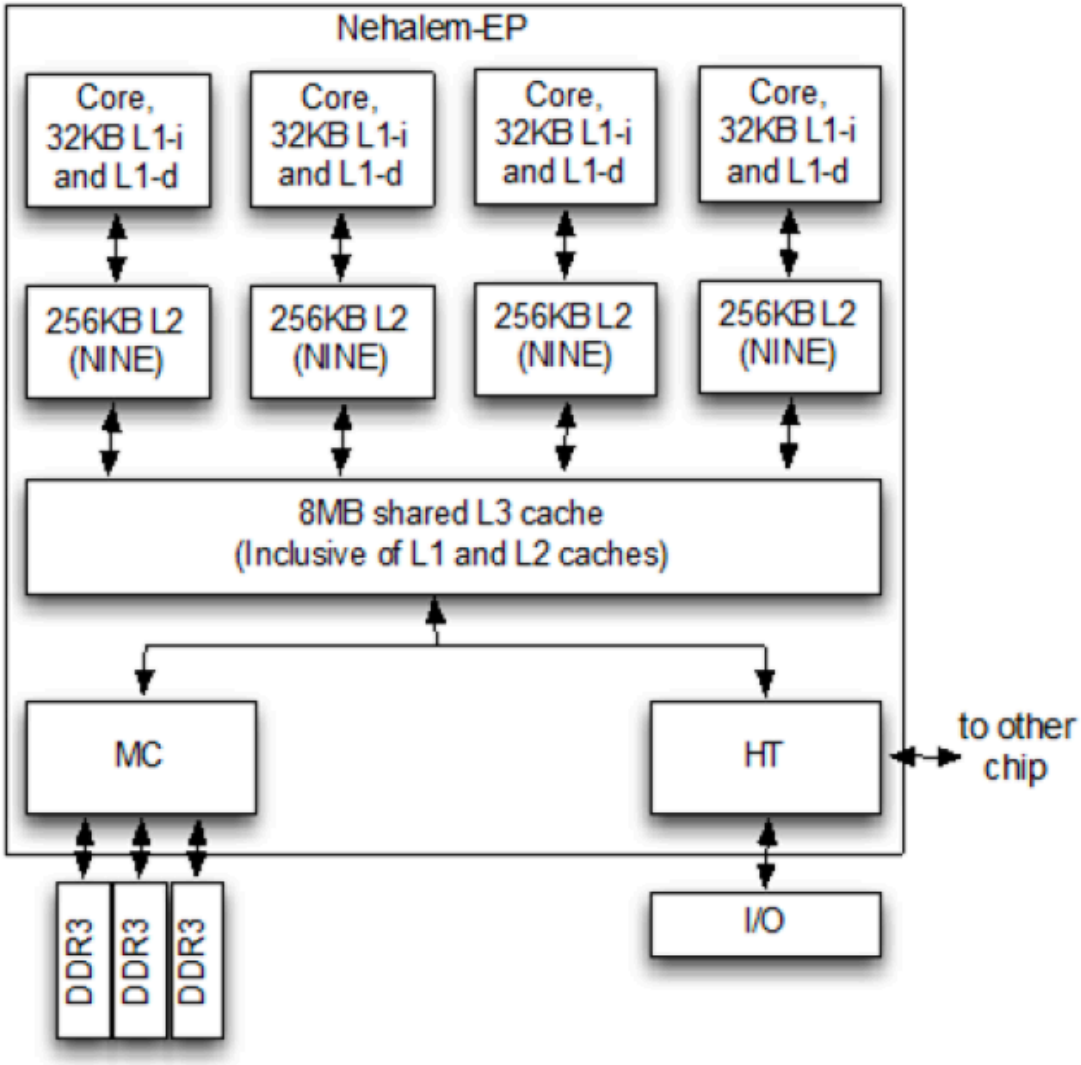
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)
- Make multiple levels of cache
  - Small L1 backed up by larger L2
  - Today's processors typically have 3 cache levels

General Rule: Bigger == Slower



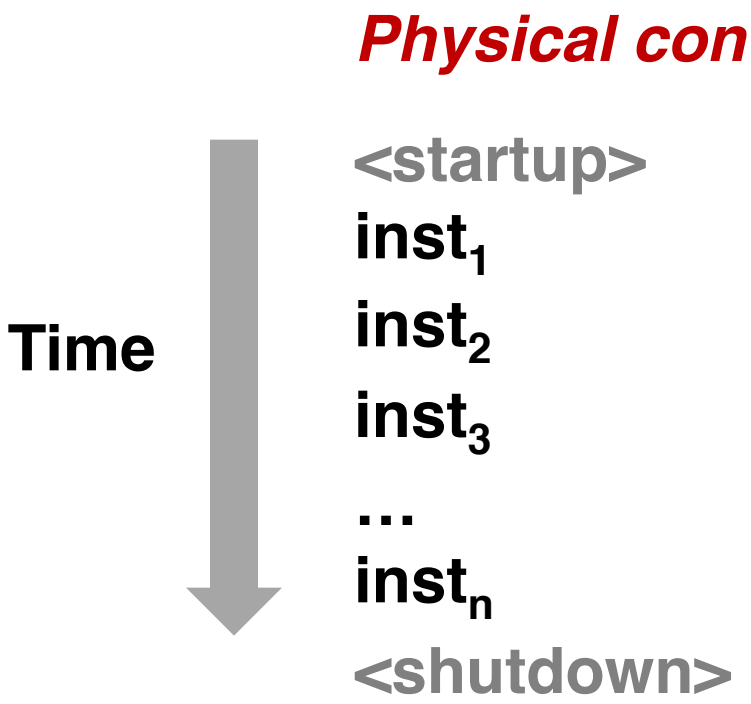
- How big should the cache be?
  - Too small and too much memory traffic
  - Too large and cache slows down execution (high latency)
- Make multiple levels of cache
  - Small L1 backed up by larger L2
  - Today's processors typically have 3 cache levels

Summary


Nehalem-EP
Core, 32KB L1-i and L1-d
Core, 32KB L1-i and L1-d
Core, 32KB L1-i and L1-d
Core, 32KB L1-i and L1-d
256KB L2 (NINE)
256KB L2 (NINE)
256KB L2 (NINE)
256KB L2 (NINE)
8MB shared L3 cache (Inclusive of L1 and L2 caches)
MC
HT
to other chip
DDR3
DDR3
DDR3
I/O

So Far in CSC252…

- 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 mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in ***program state***
- Insufficient for a useful system: Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  React to changes in ***program state***
- Insufficient for a useful system: Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires
- System needs mechanisms for “exceptional control flow”

Exceptional Control Flow

- Exists at all levels of a computer system

Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software

Exceptional Control Flow

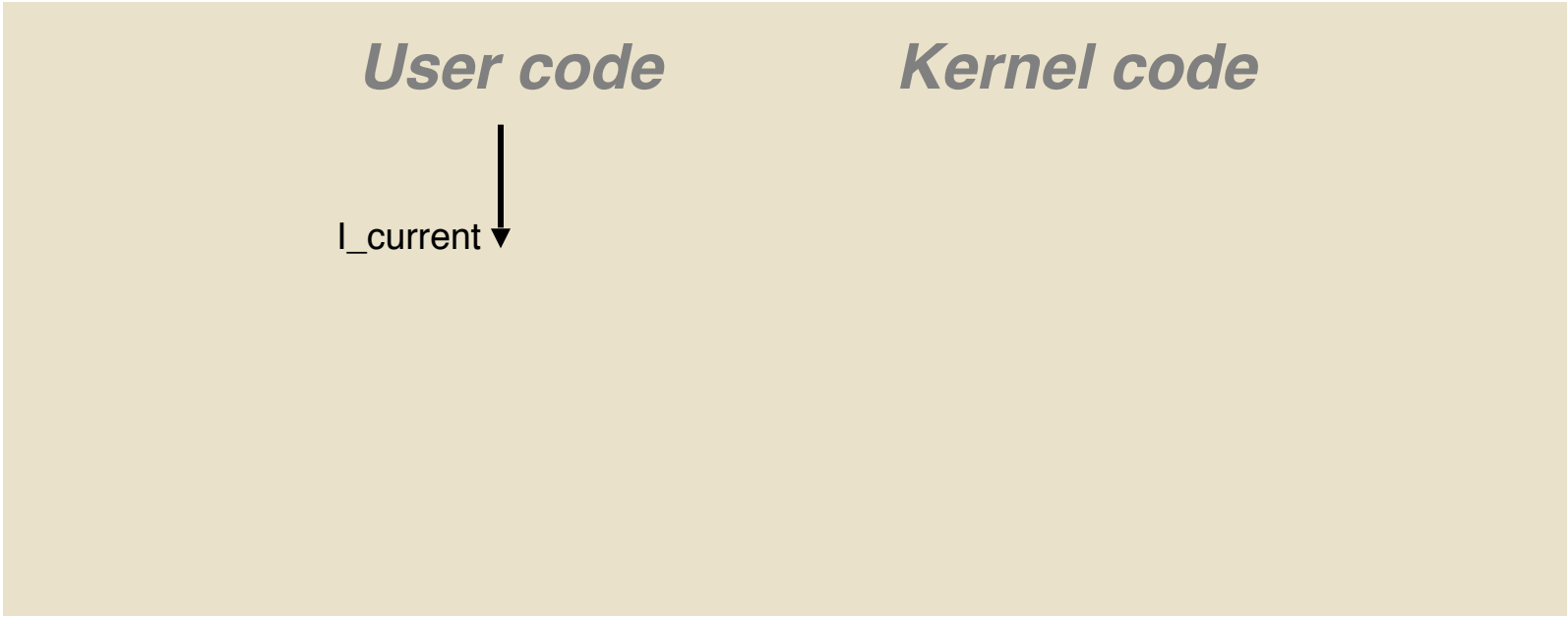
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- Higher level mechanisms
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - Implemented by C runtime library

Today

- Exceptions/Interrupts
- Processes and Signals: Special kinds of exception

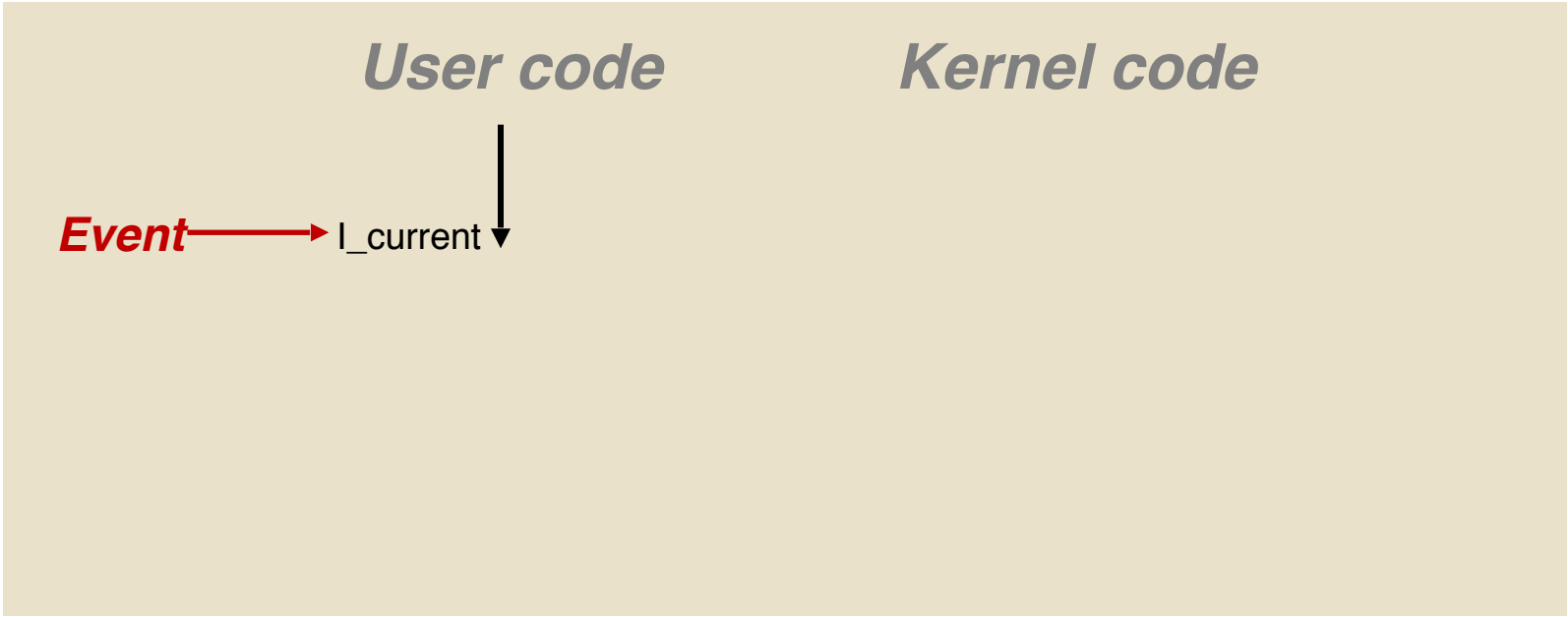
Exceptions

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



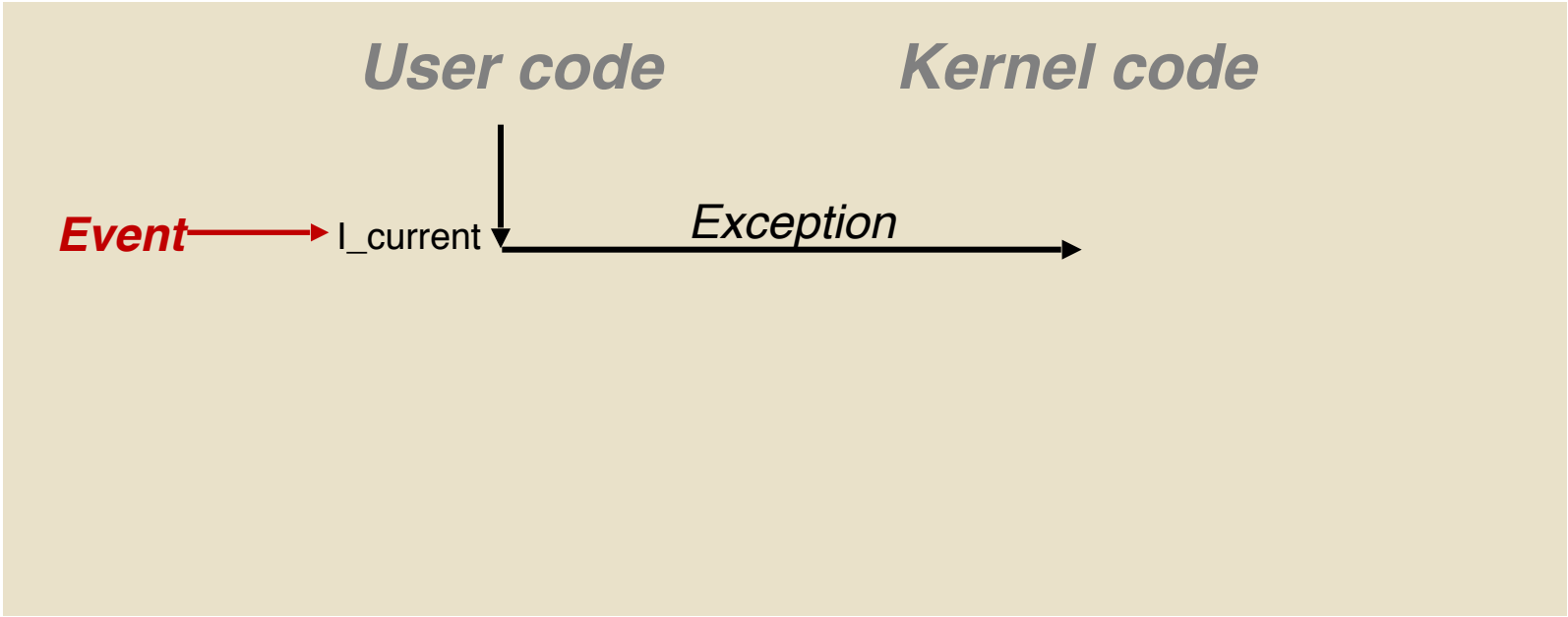
Exceptions

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



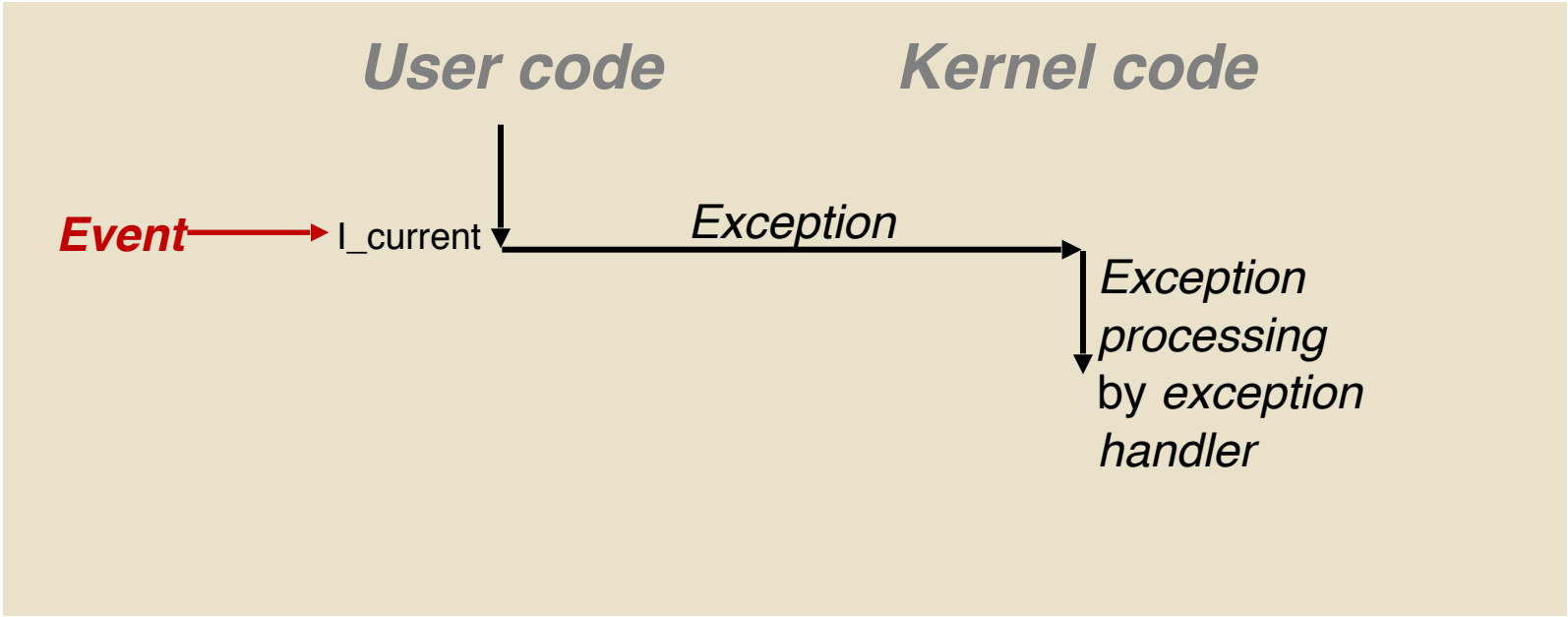
Exceptions

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



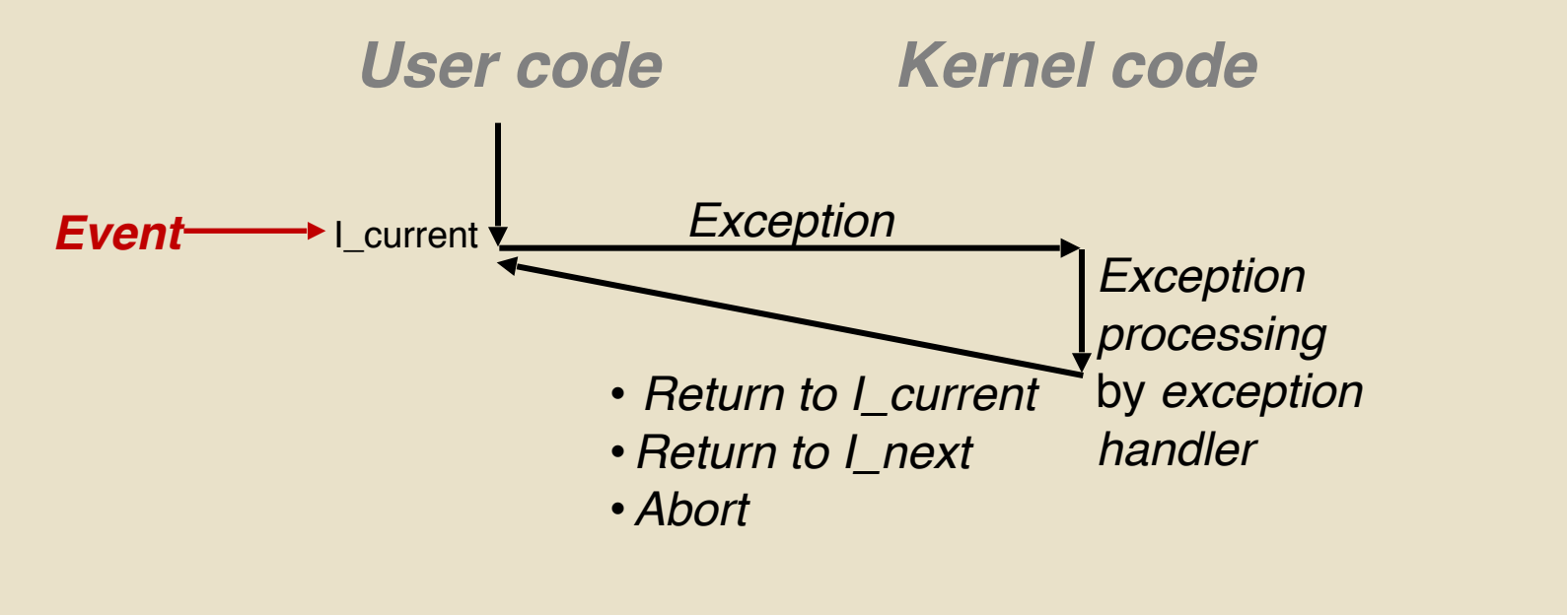
Exceptions

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



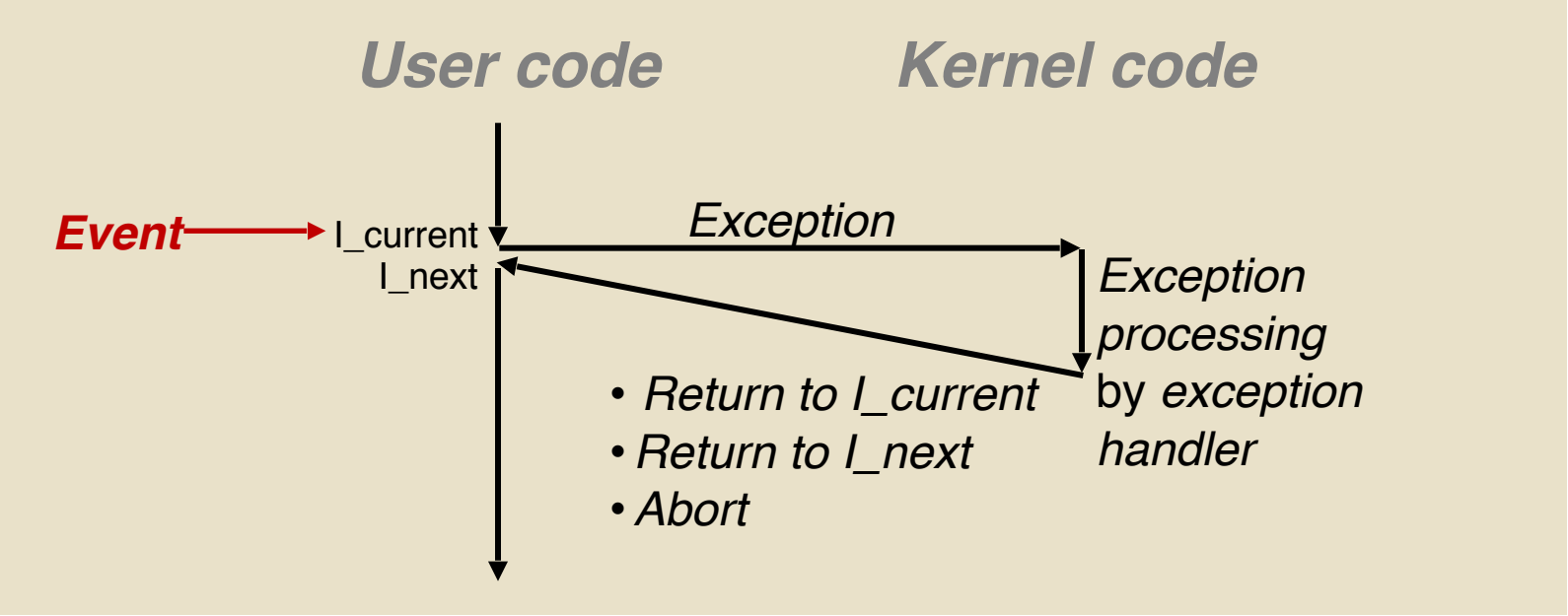
Exceptions

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



Exceptions

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



Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Events that can happen at any time. Computers have little control.
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to “next” instruction

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), ***protection faults (the infamous Segmentation Fault!)*** (unrecoverable in Linux), floating point exceptions (unrecoverable in Linux)
    - Either re-executes faulting (“current”) instruction or aborts

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), ***protection faults (the infamous Segmentation Fault!)*** (unrecoverable in Linux), floating point exceptions (unrecoverable in Linux)
    - Either re-executes faulting (“current”) instruction or aborts
  - *Aborts*
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

Fault Example: Page Fault

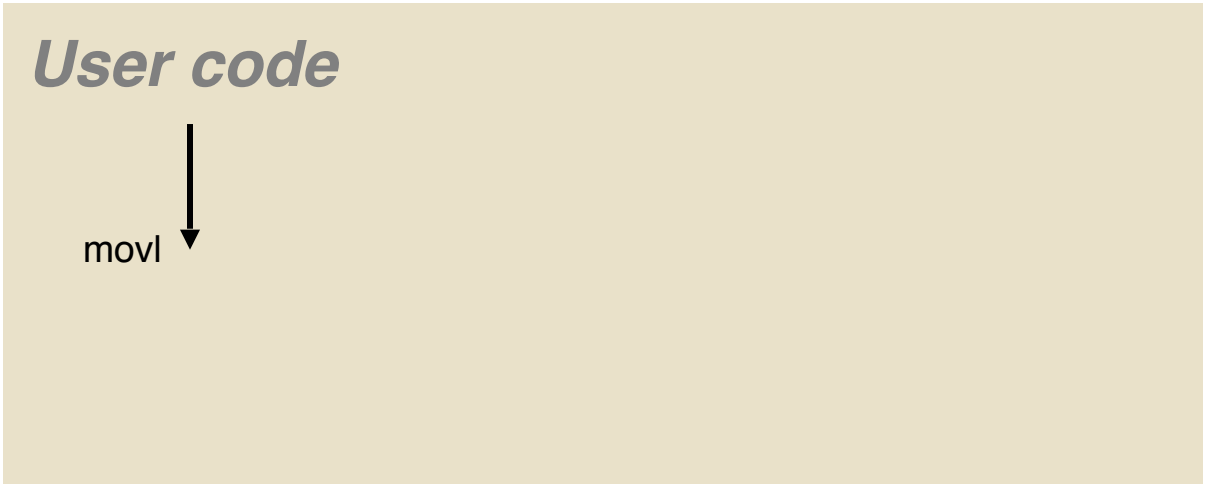
- User writes to memory location
- That memory location is not found in memory because it is currently on disk
- Trigger a Page Fault (recoverable), the exception handler loads the data from disk to memory (will discuss in detail later in the class)

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

Fault Example: Page Fault

- User writes to memory location
- That memory location is not found in memory because it is currently on disk
- Trigger a Page Fault (recoverable), the exception handler loads the data from disk to memory (will discuss in detail later in the class)

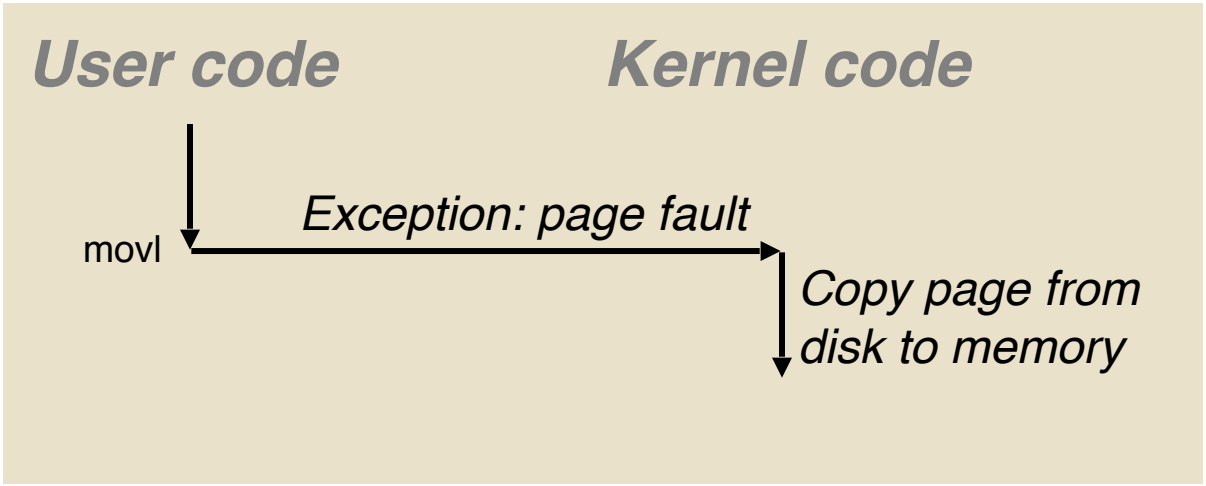
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example: Page Fault

- User writes to memory location
- That memory location is not found in memory because it is currently on disk
- Trigger a Page Fault (recoverable), the exception handler loads the data from disk to memory (will discuss in detail later in the class)

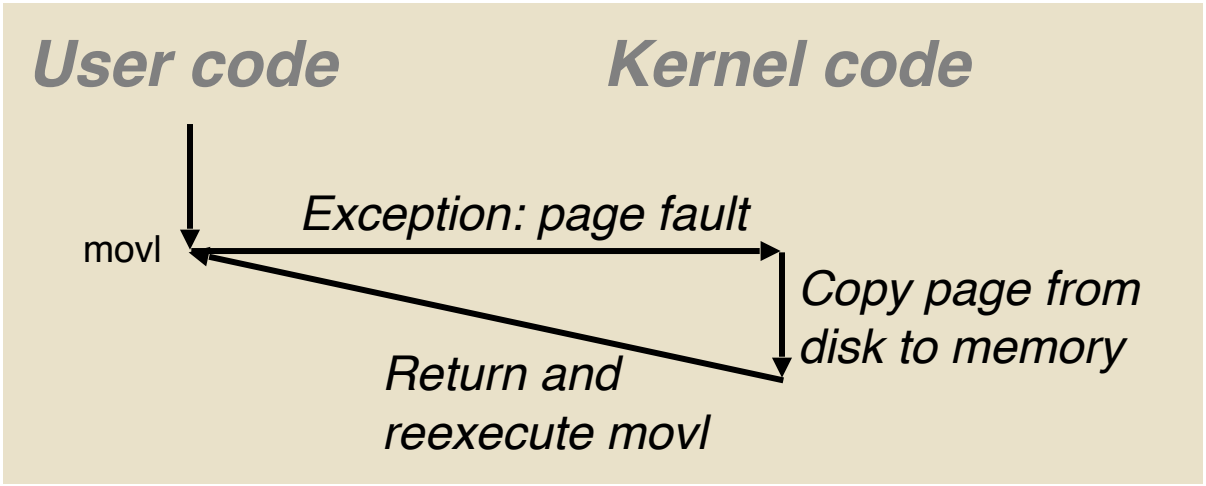
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example: Page Fault

- User writes to memory location
- That memory location is not found in memory because it is currently on disk
- Trigger a Page Fault (recoverable), the exception handler loads the data from disk to memory (will discuss in detail later in the class)

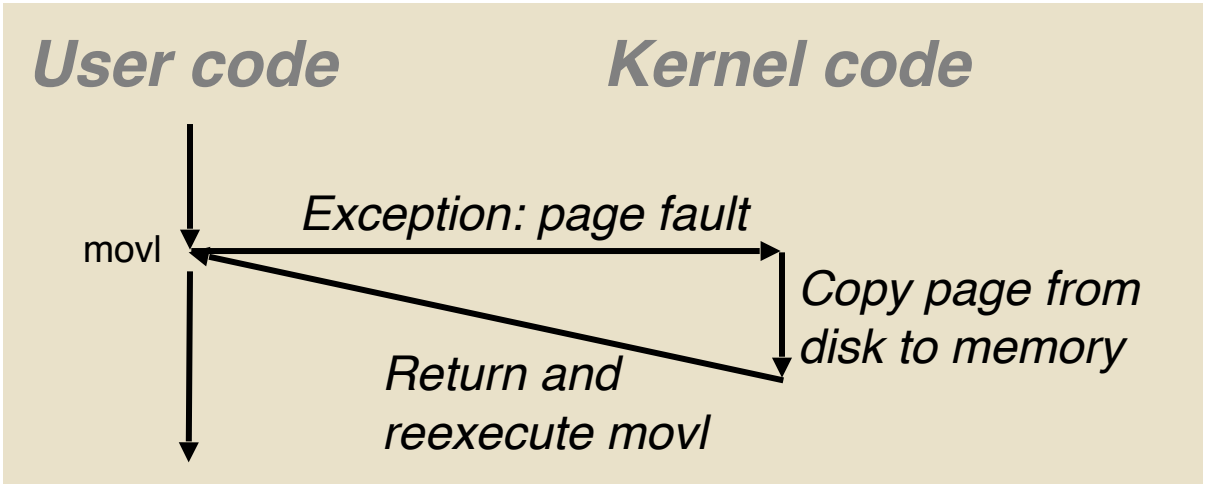
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example: Page Fault

- User writes to memory location
- That memory location is not found in memory because it is currently on disk
- Trigger a Page Fault (recoverable), the exception handler loads the data from disk to memory (will discuss in detail later in the class)

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

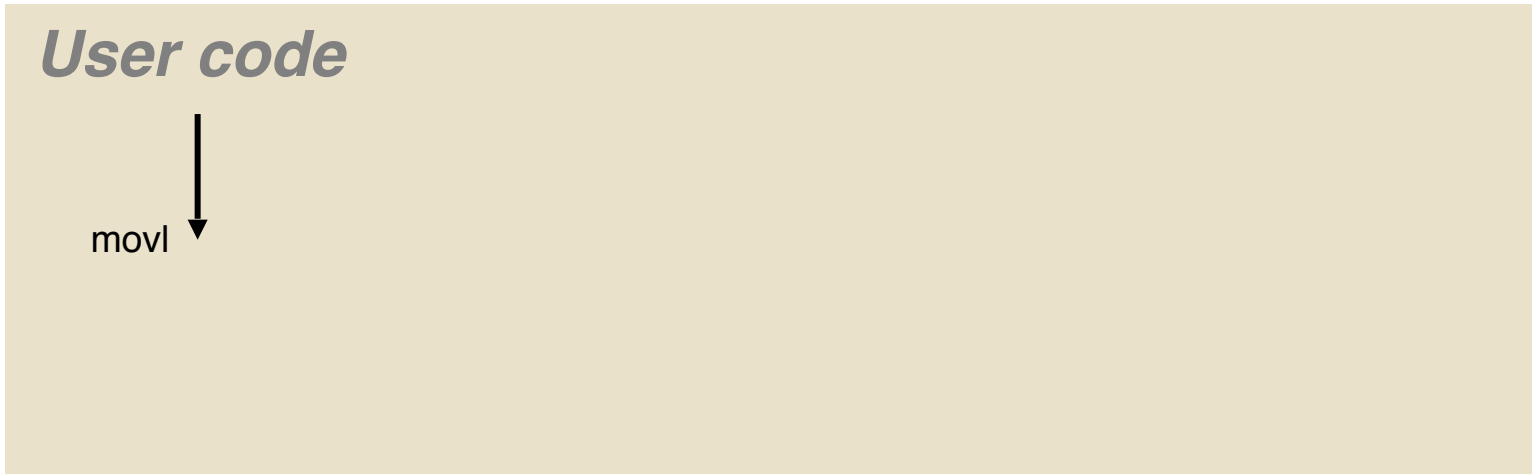


Fault Example: Protection Fault

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

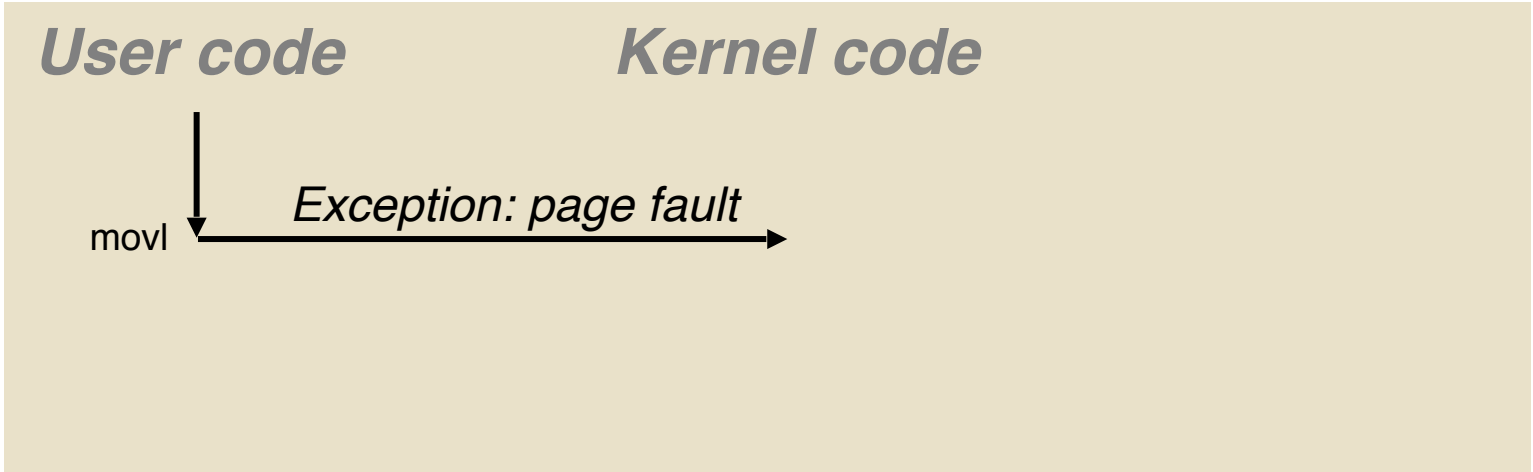
Fault Example: Protection Fault

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



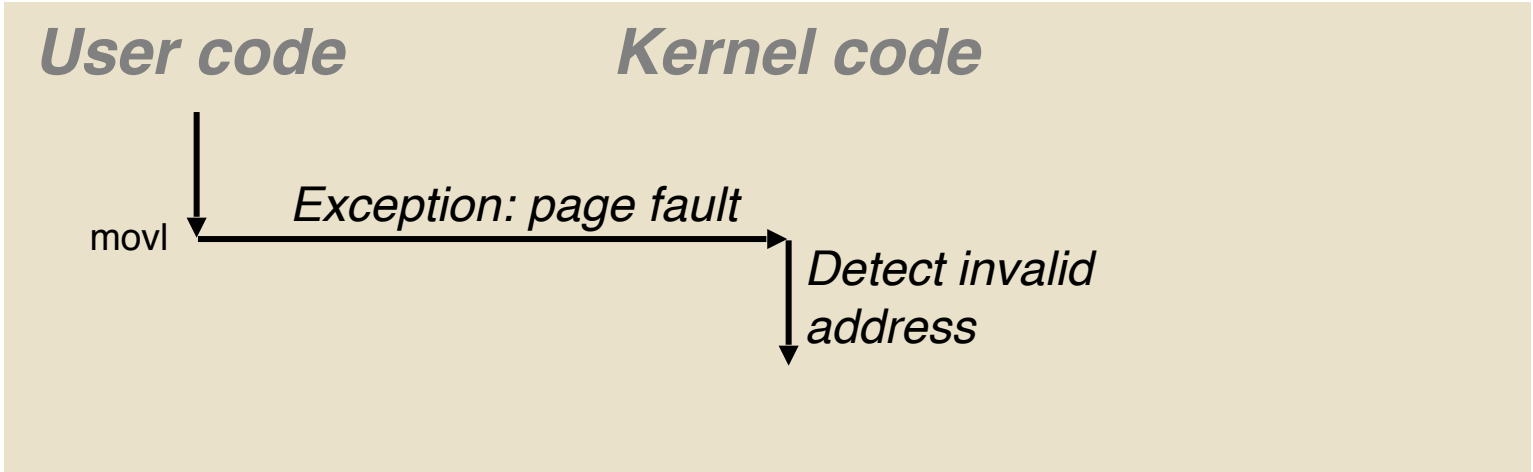
Fault Example: Protection Fault

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



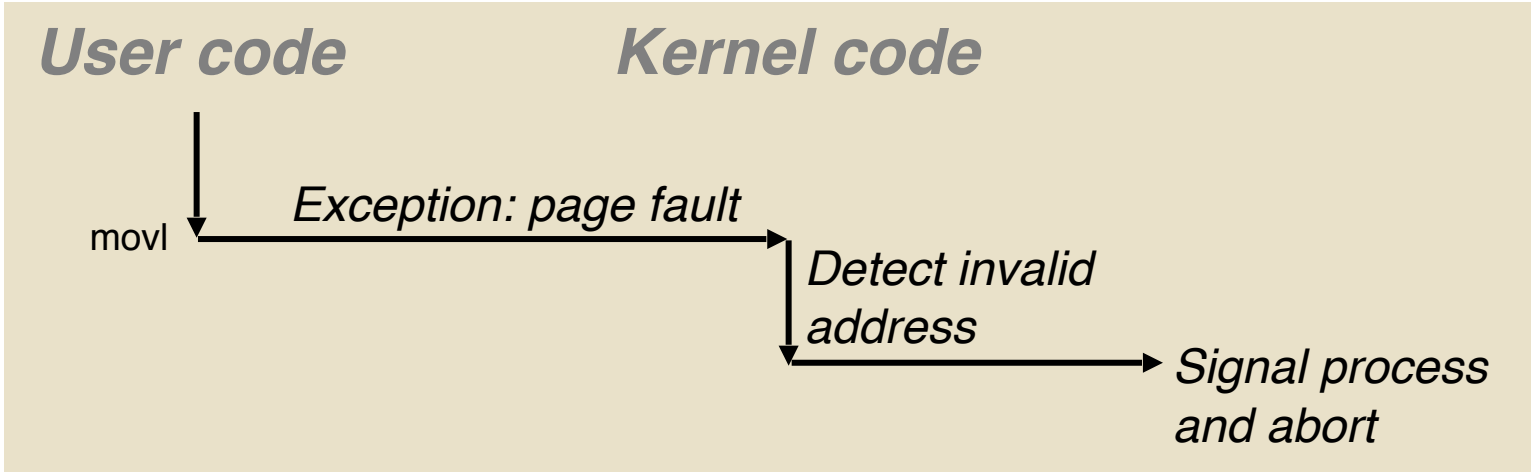
Fault Example: Protection Fault

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



Fault Example: Protection Fault

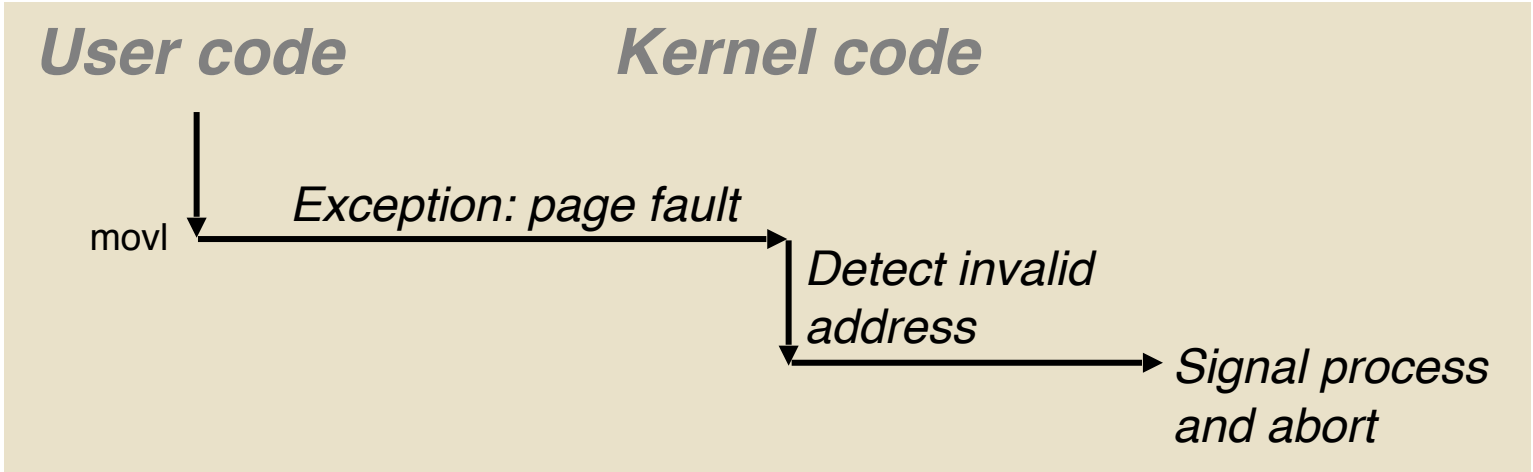
```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Fault Example: Protection Fault

- Access illegal memory location (e.g., dereferencing a null pointer)

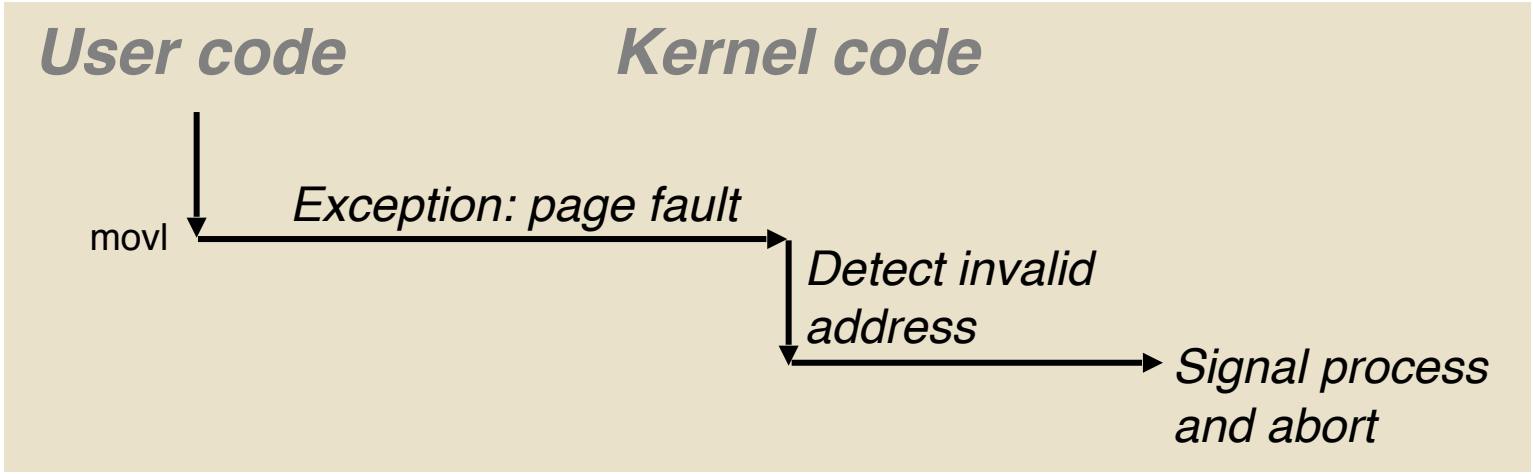
```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Fault Example: Protection Fault

- Access illegal memory location (e.g., dereferencing a null pointer)
- First trigger a Page Fault, the exception handler decides that this is unrecoverable, so simply aborts

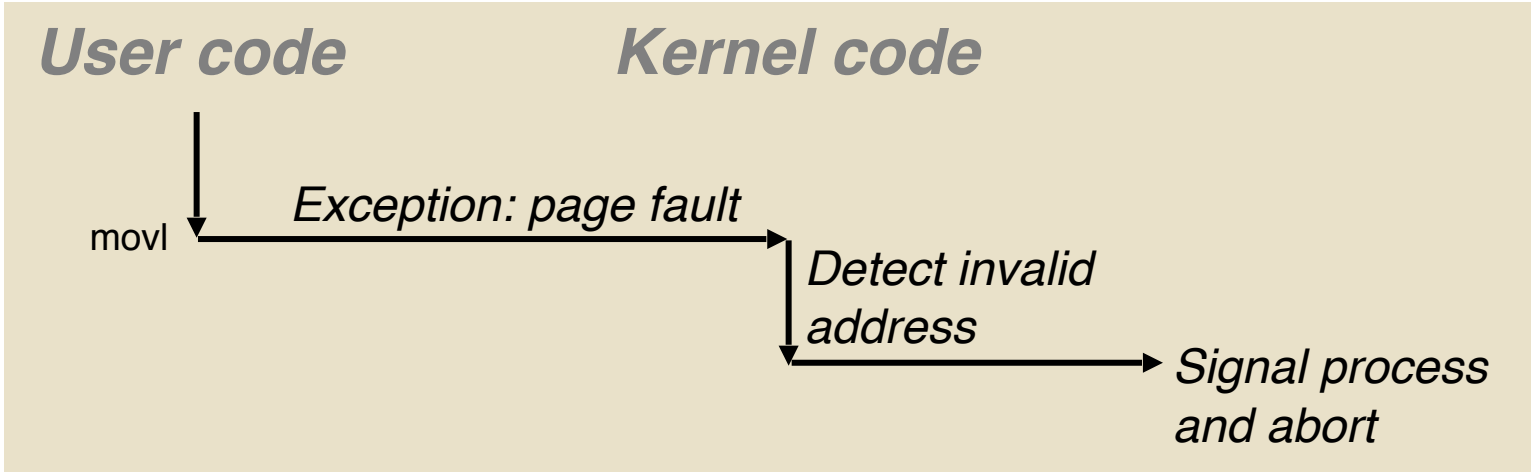
```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Fault Example: Protection Fault

- Access illegal memory location (e.g., dereferencing a null pointer)
- First trigger a Page Fault, the exception handler decides that this is unrecoverable, so simply aborts
- User process exits with “segmentation fault”

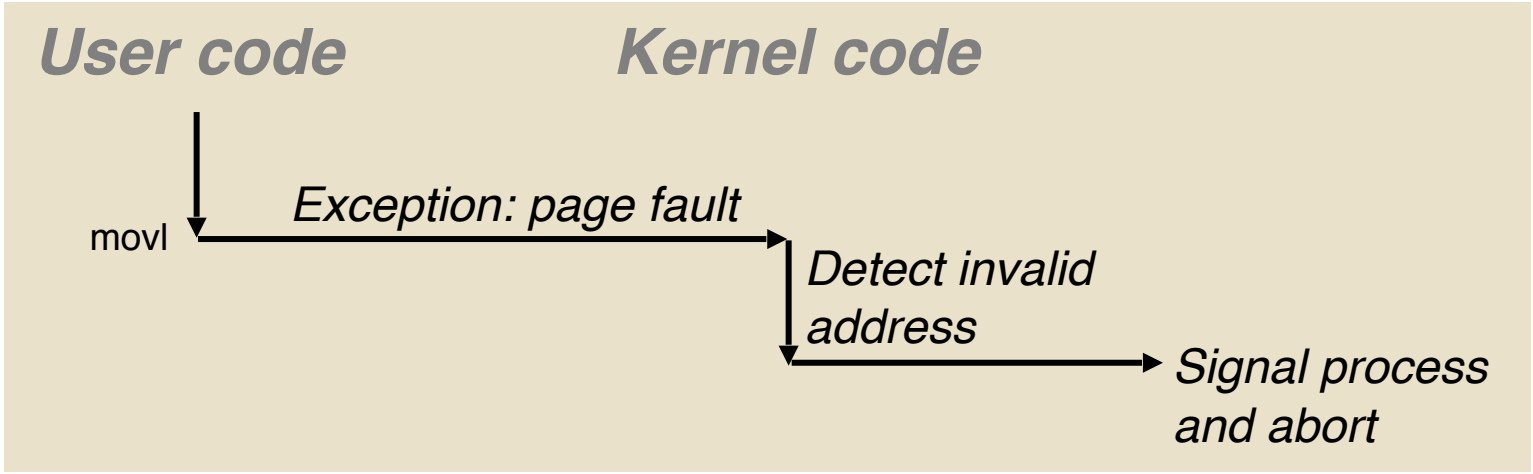
```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Fault Example: Protection Fault

- Access illegal memory location (e.g., dereferencing a null pointer)
- First trigger a Page Fault, the exception handler decides that this is unrecoverable, so simply aborts
- User process exits with “segmentation fault”
- Again, later in the class…

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



Others' Definitions

- The textbook's definitions are not universally accepted
- Intel (http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/ia32/xeon/19250.htm?page=2)
  - **Interrupt**: An exception that comes from outside of the processor. There are two kinds of exceptions: local and external. A local exception is generated from a program. External exceptions are usually generated by external I/O devices and received at exception pins.
- PowerPC Architecture
  - Interrupts "allow the processor to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions"
- PowerPC 604
  - Everything is an exception
- Motorola 68K
  - Everything is an exception
- VAX
  - Interrupts: device, software, urgent
  - Exceptions: faults, traps, aborts

When Do You Call the Handler?

- Interrupts: when convenient. Typically wait until the current instructions in the pipeline are finished
- Exceptions: typically immediately as programs can't continue without resolving the exception (think of page fault)
- Maskable verses Unmaskable
  - Interrupts can be individually masked (i.e., ignored by CPU)
  - Synchronous exceptions are usually unmaskable
- Some interrupts are intentionally unmaskable
  - Called non-maskable interrupts (NMI)
  - Indicating a critical error has occurred, and that the system is probably about to crash

Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction

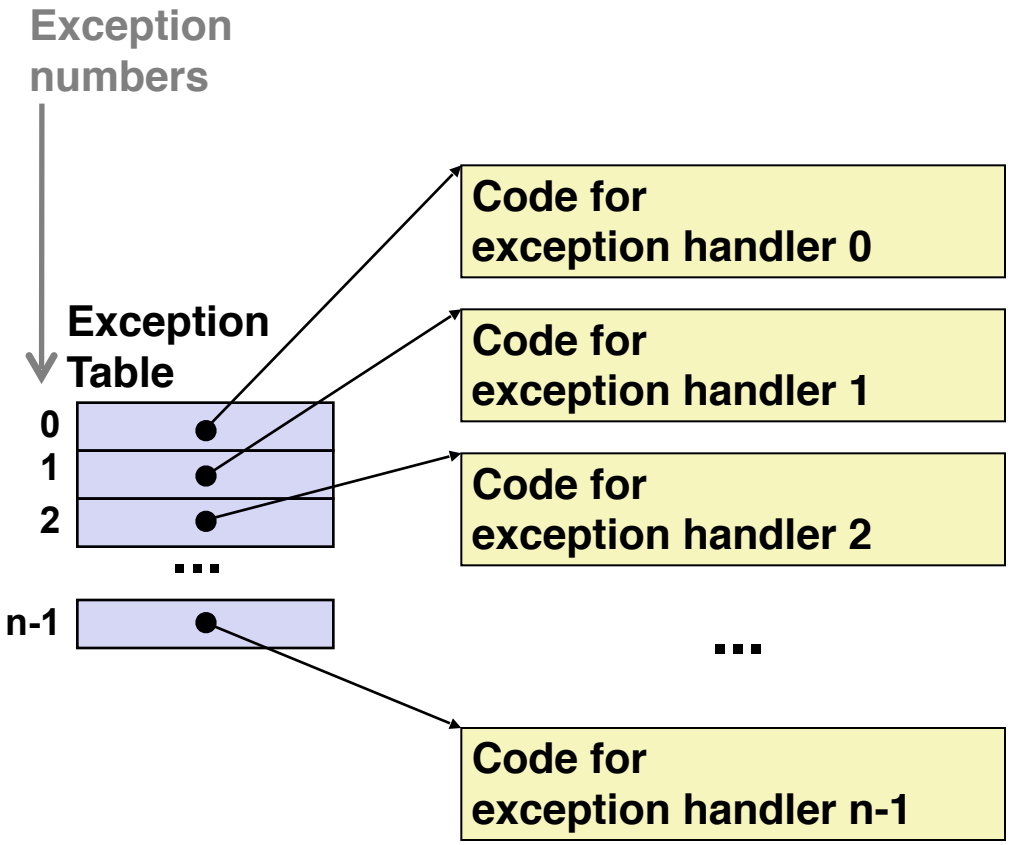
Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction
- Faults
  - Exception handler returns to the instruction that caused the exception, i.e., ***re-execute*** it!

Where Do You Restart?

- Interrupts/Traps
  - Handler returns to the ***following*** instruction
- Faults
  - Exception handler returns to the instruction that caused the exception, i.e., ***re-execute*** it!
- Aborts
  - Never returns to the program

Where to Find Exception Handlers?



- Each type of event has a unique exception number k
- k = index into exception table
- Exception table lives in memory. Its start address is stored in a special register
- Handler k is called each time exception k occurs

Nested Exceptions

- One interrupt/exception occurs when another is already active
- Can fundamentally do it
  - Subroutine calls within subroutine calls
  - Handlers need to save appropriate state

Concurrent Interrupts

- More than one interrupts happen at the same time
- Pre-defined priority
- The chipset arbitrates which one to respond to first

Today

- Exceptions/Interrupts
- Processes and Signals: Special kinds of exception
  - Processes
  - Process Control
  - Signals

Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

Processes

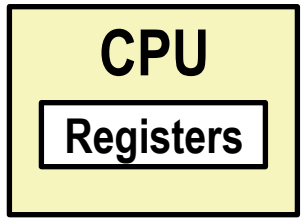
- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:

Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

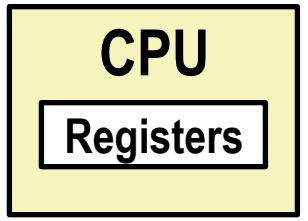
- Process provides each program with two key abstractions:
  - *“Owns” the CPU*



Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

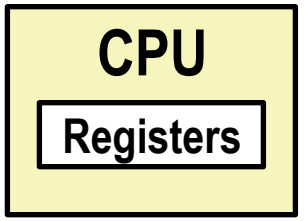
- Process provides each program with two key abstractions:
  - *“Owns” the CPU*
    - Each program seems to have exclusive use of the CPU



Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

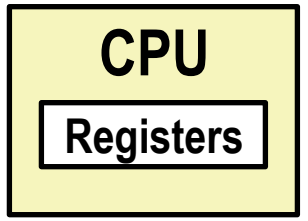
- Process provides each program with two key abstractions:
  - *“Owns” the CPU*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching



Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

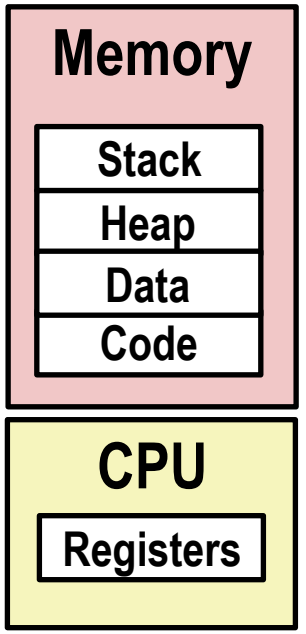
- Process provides each program with two key abstractions:
  - *“Owns” the CPU*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
  - *Private address space*



Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

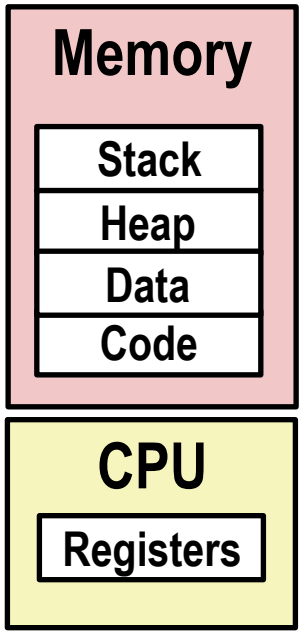
- Process provides each program with two key abstractions:
  - *“Owns” the CPU*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
  - *Private address space*



Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

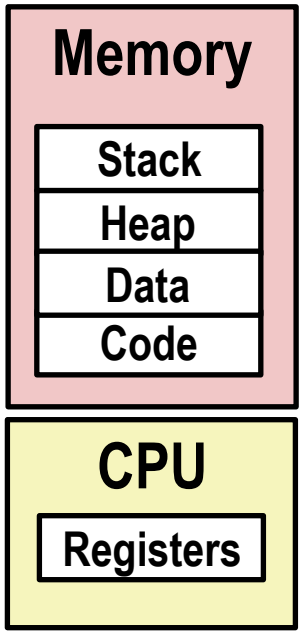
- Process provides each program with two key abstractions:
  - *“Owns” the CPU*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
  - *Private address space*
    - Each program seems to have exclusive use of main memory.



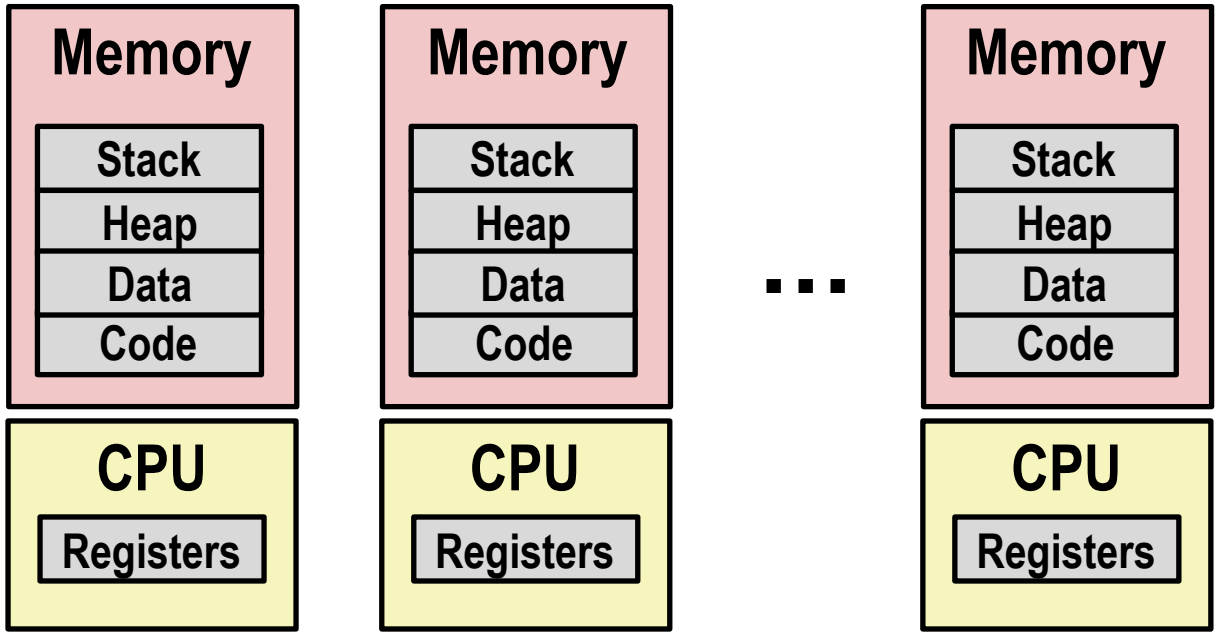
Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - *“Owns” the CPU*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
  - *Private address space*
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called virtual memory

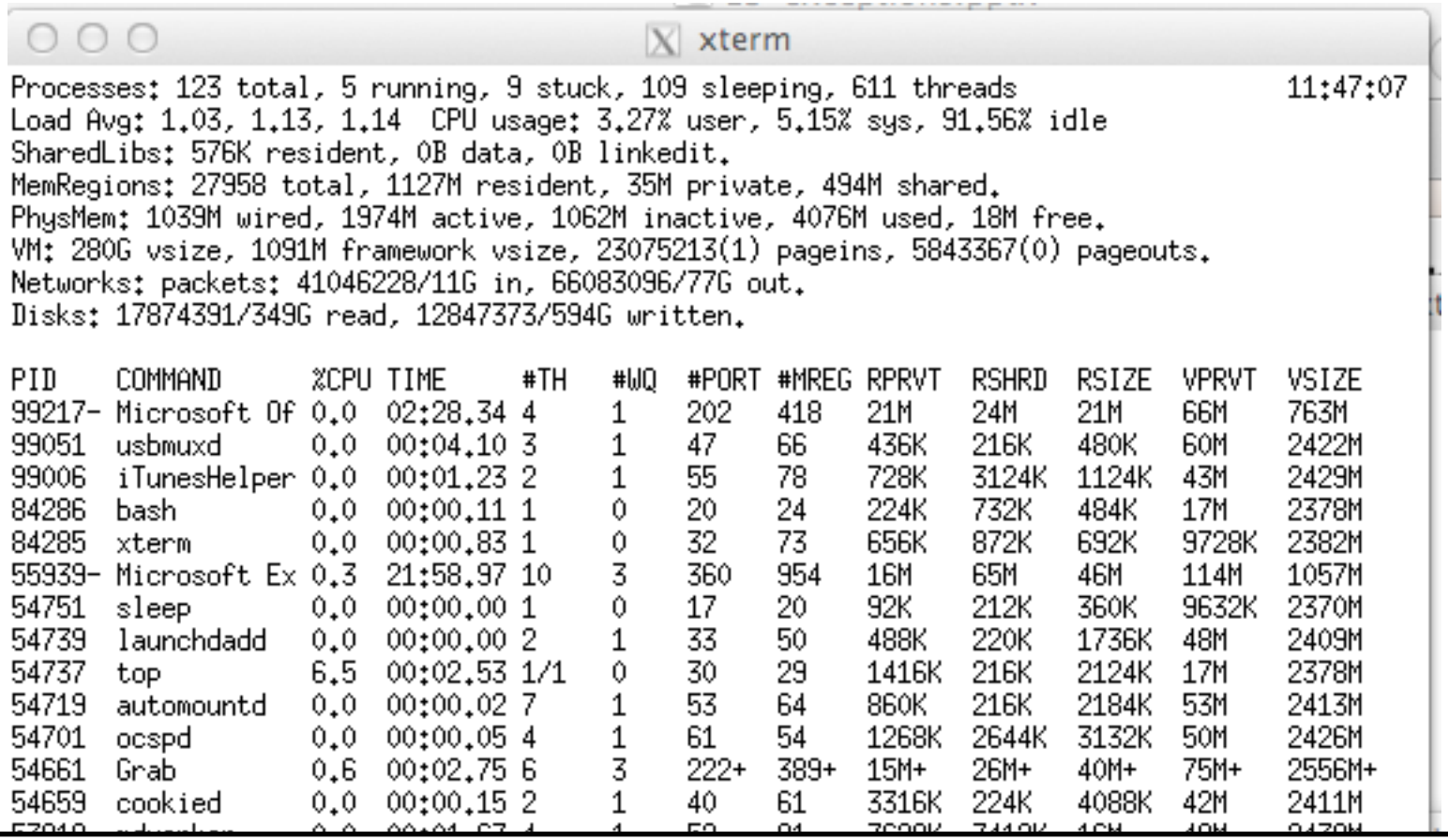


Multiprocessing: The Illusion



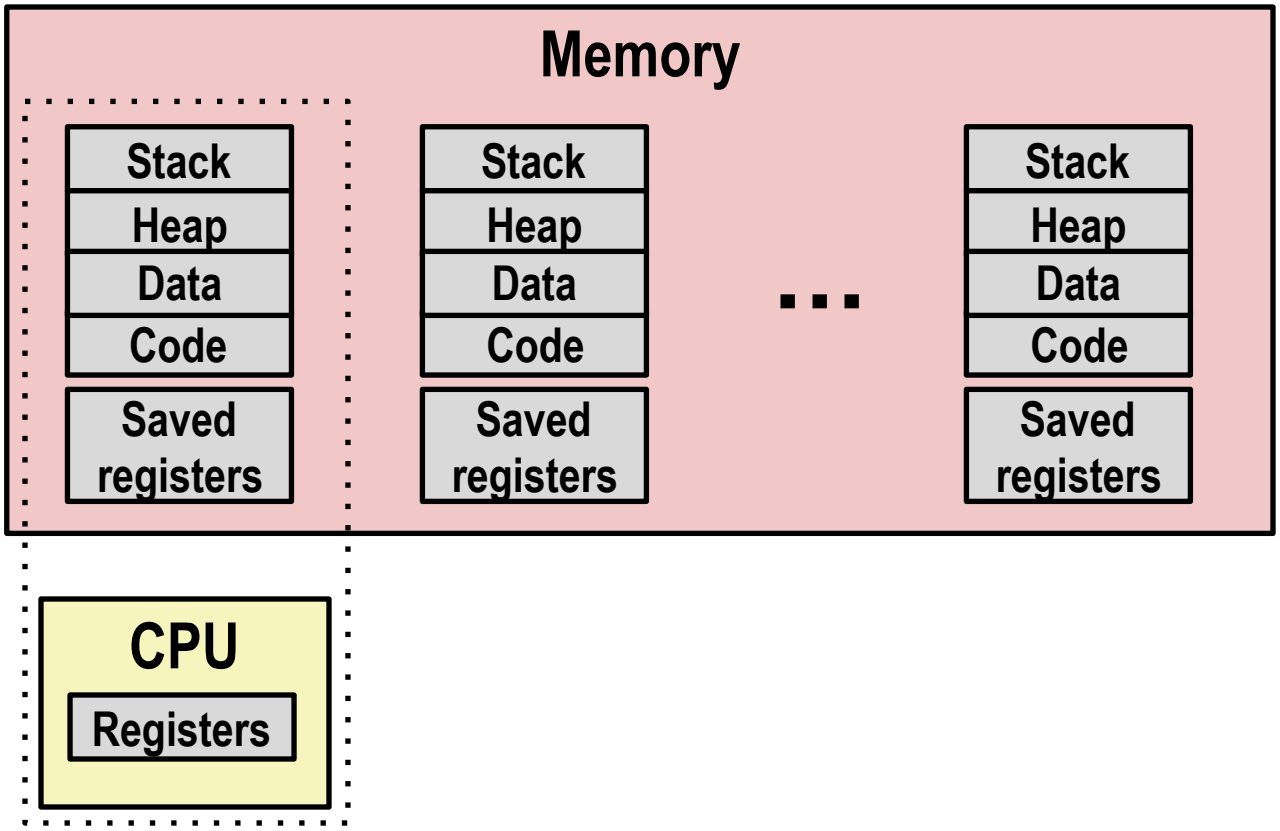
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example



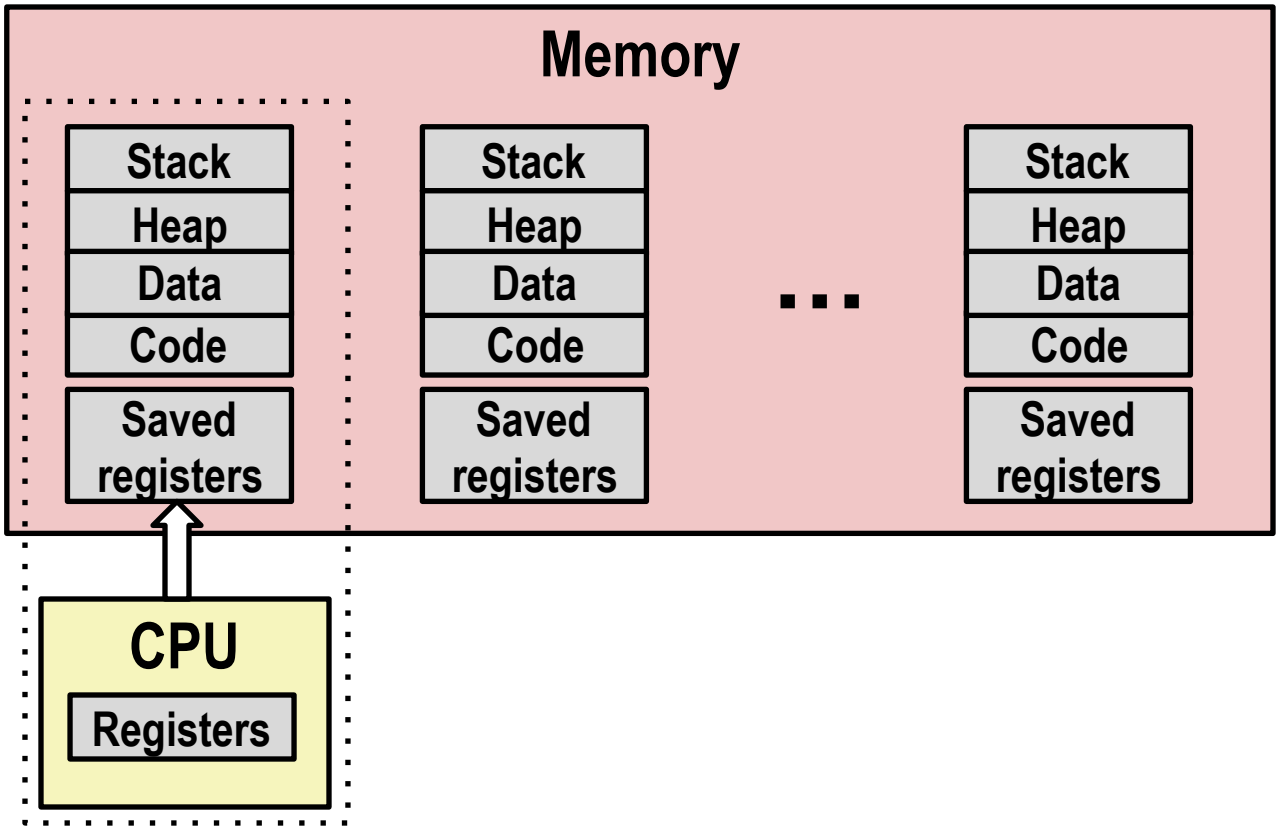
- Running program “top” on Mac
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality



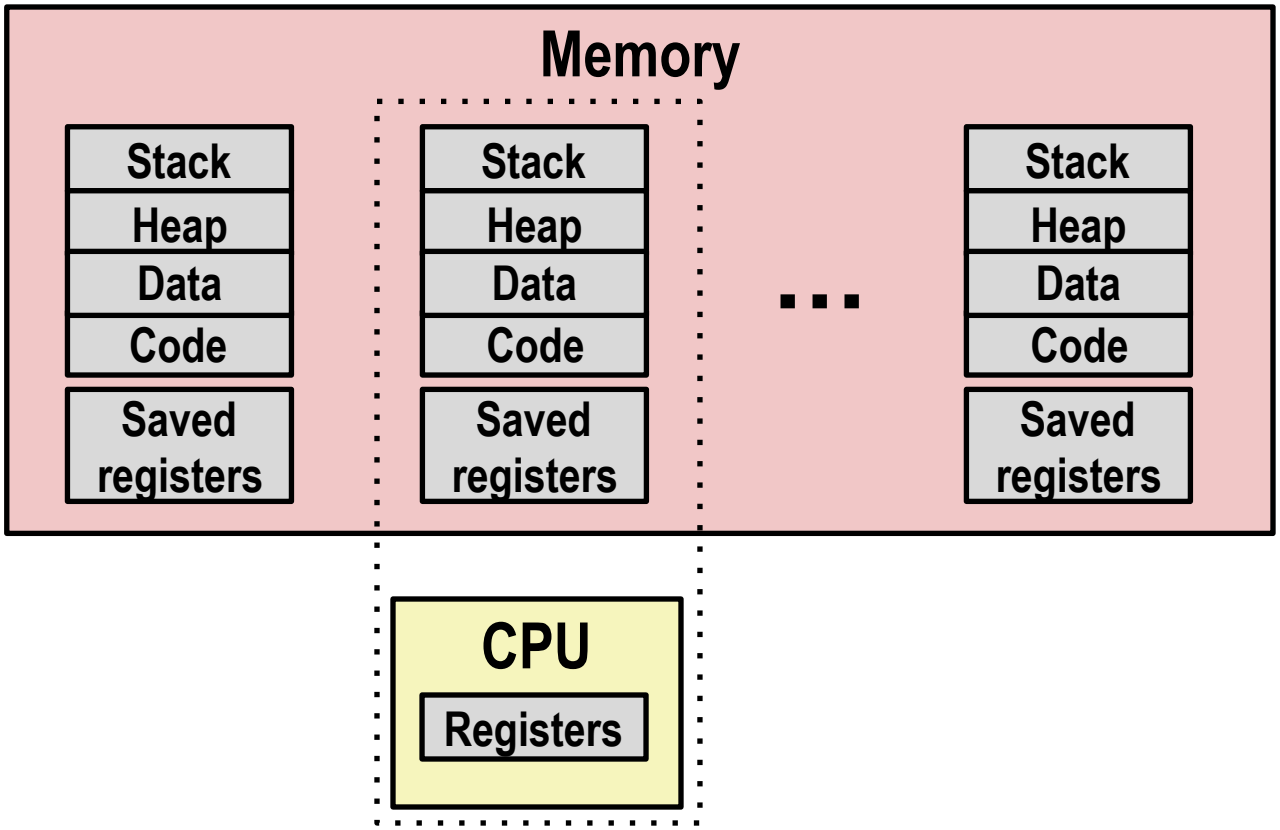
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



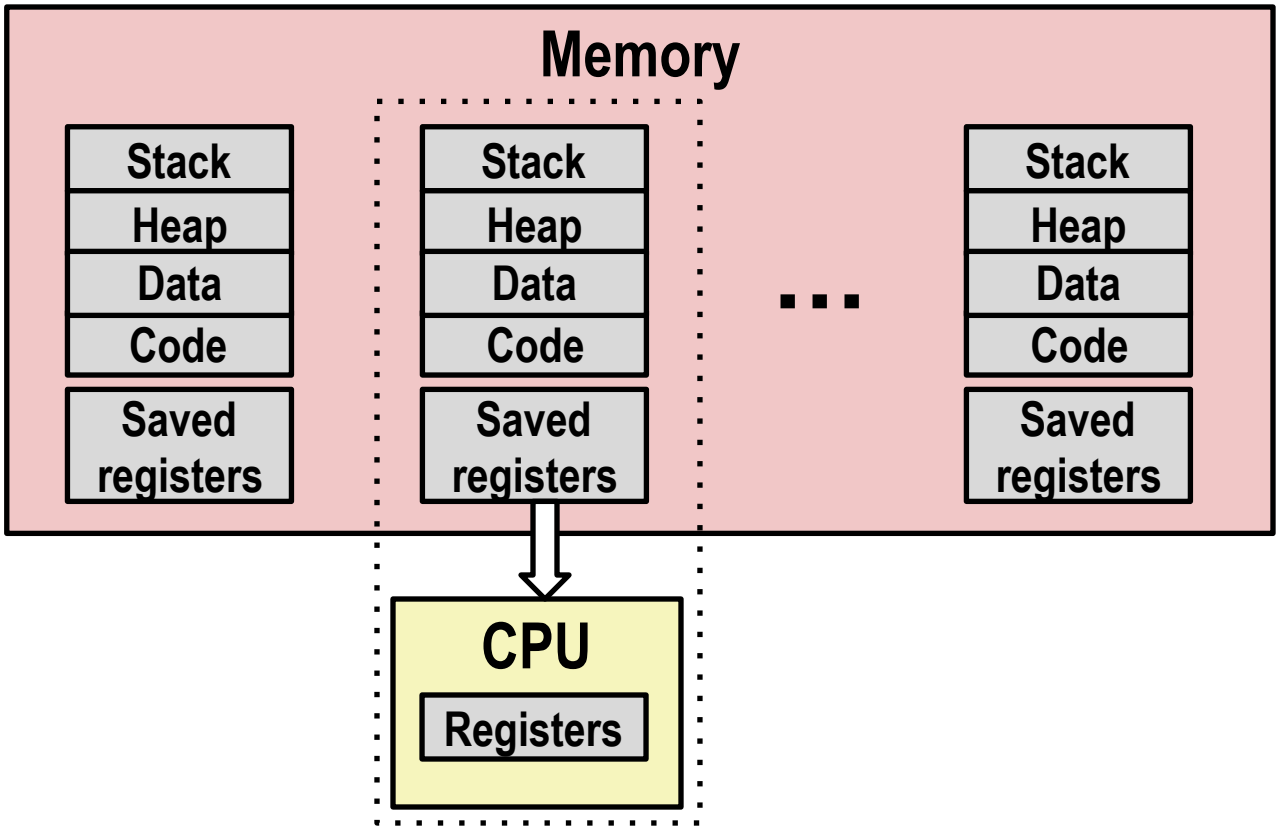
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



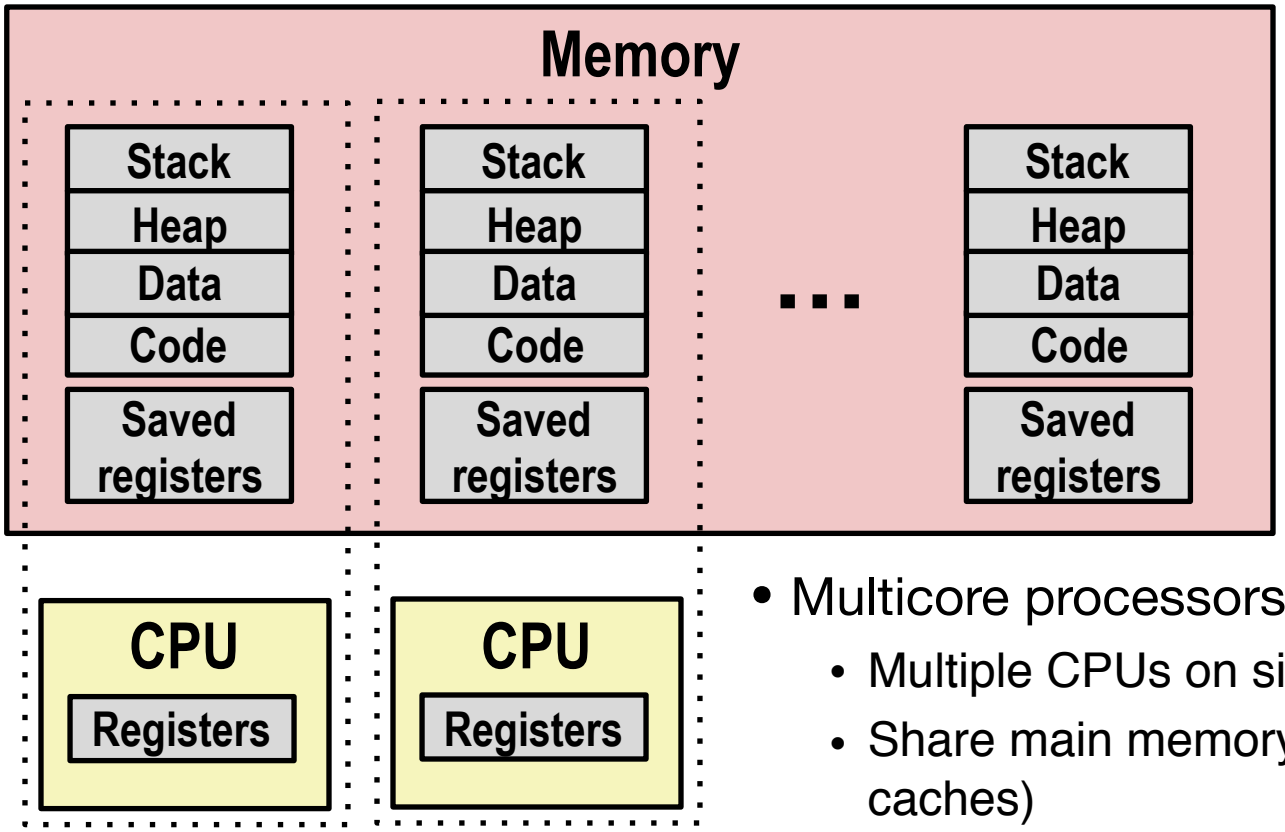
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



- Load saved registers and switch address space (context switch)

Multiprocessing: The (Modern) Reality



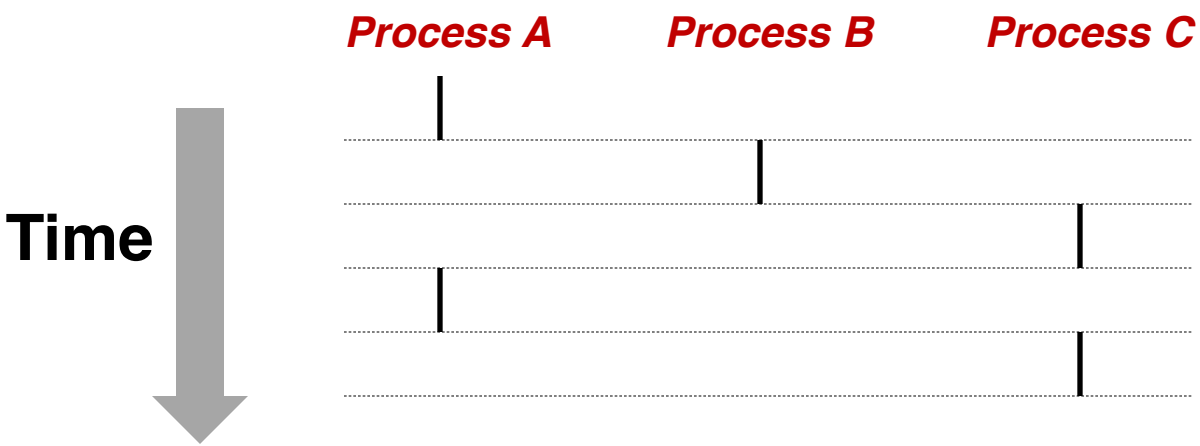
- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*

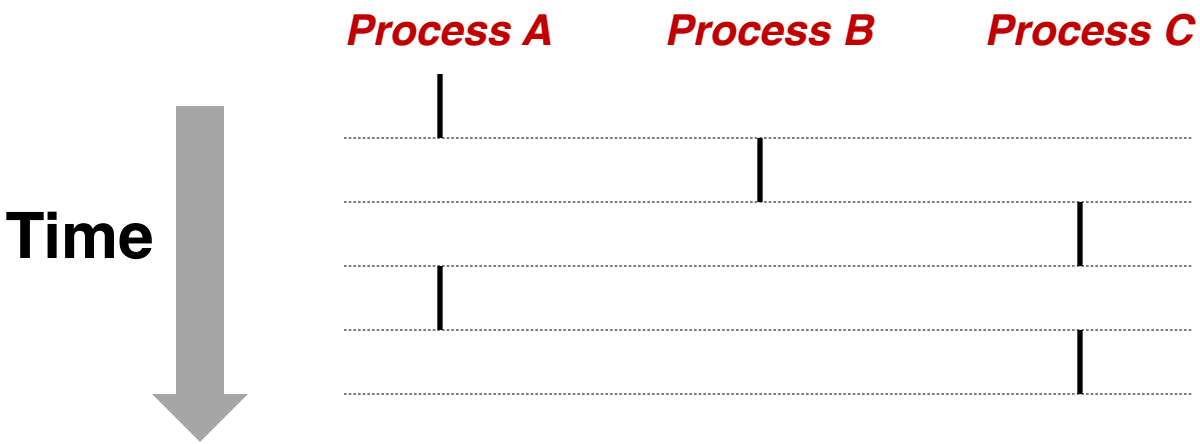
Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):



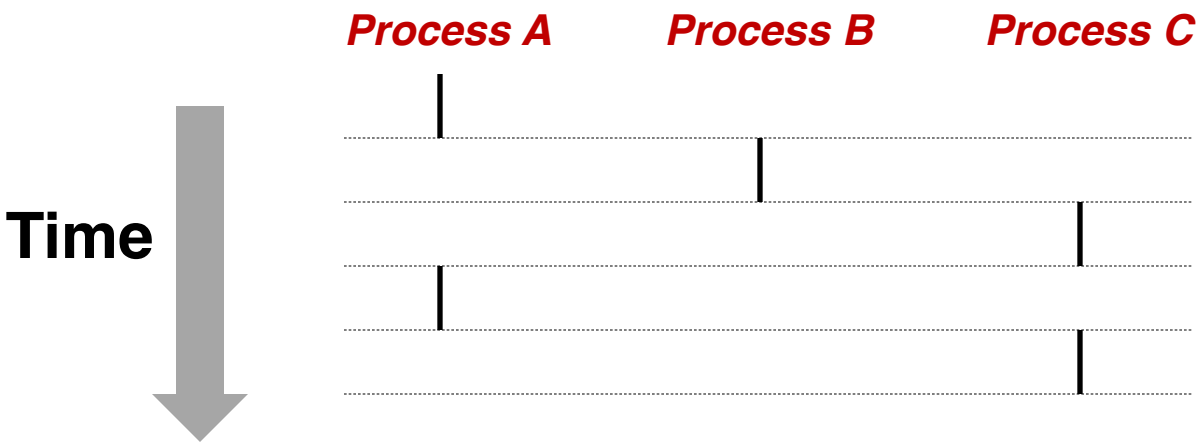
Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C



Concurrent Processes

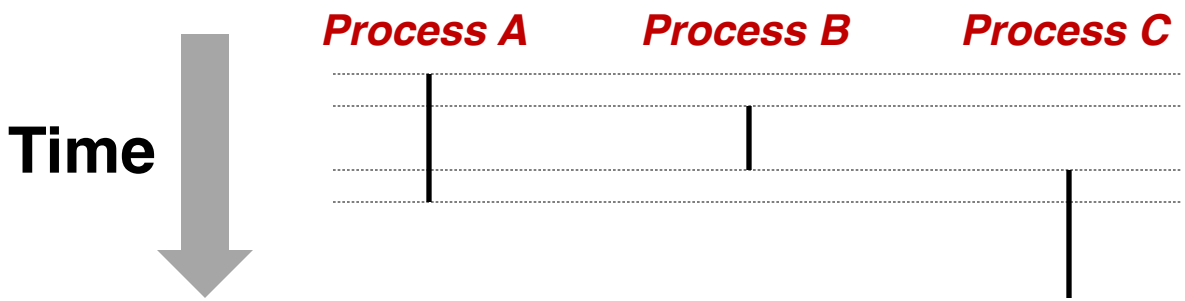
- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

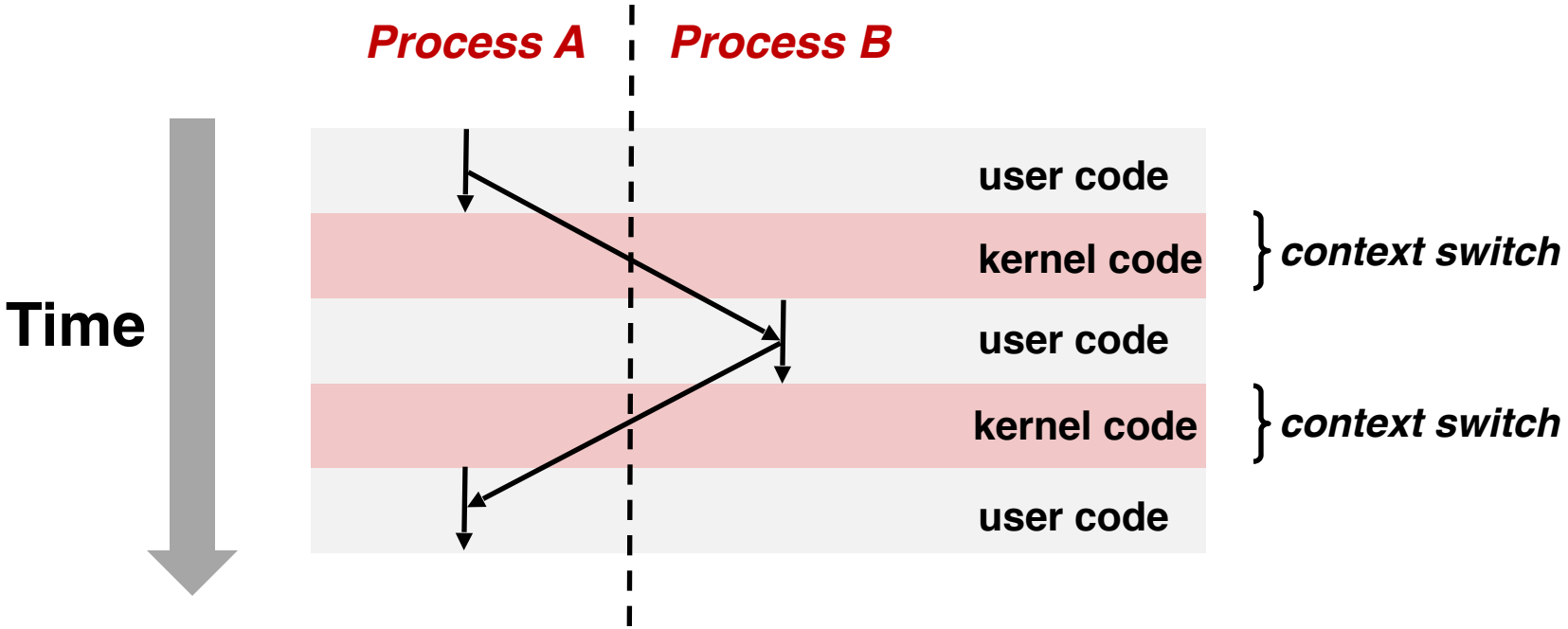
- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other



Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*



Today

- Exceptions/Interrupts
- Processes and Signals: Special kinds of exception
  - Processes
  - Process Control
  - Signals

Obtaining Process IDs

- `pid_t getpid(void)`
  - Returns PID of current process

- `pid_t getppid(void)`
  - Returns PID of parent process

Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states

- Running
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- Stopped
  - Process execution is suspended and will not be scheduled until further notice (through something call **signals**)
- Terminated
  - Process is stopped permanently

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate
  - Returning from the `main` routine
  - Calling the `exit` function

- `void exit(int status)`
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- `exit` is called once but never returns.