

CSC 252/452: Computer Organization

Fall 2024: Lecture 17

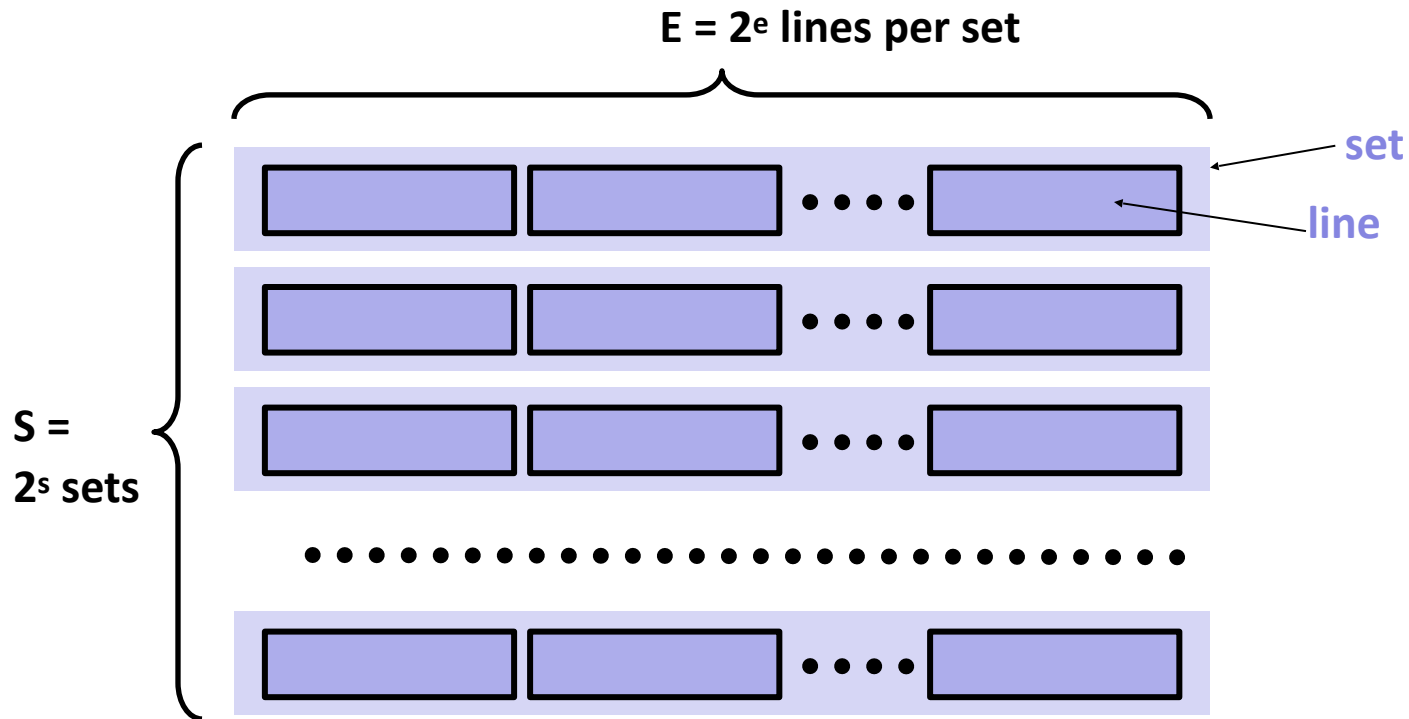
Instructor: Yanan Guo

Department of Computer Science
University of Rochester

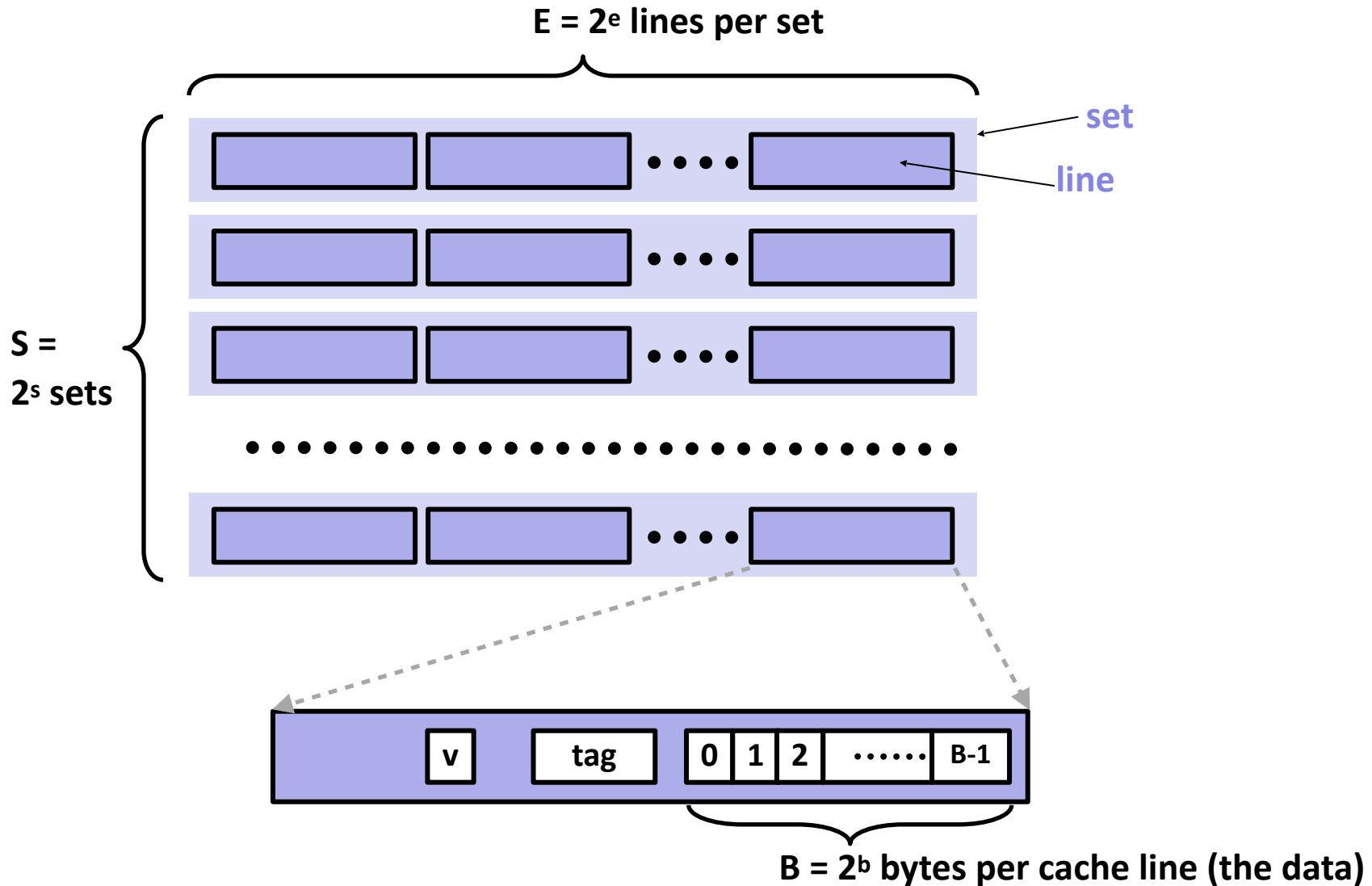
Announcements

- Programming Assignment 4 released
 - Due on Nov. 15th
 - Cache Simulator + Matrix Transpose
- For midterm
 - Come to my office hours tomorrow if TAs cannot solve your problems of midterm grades
- For Programming Assignment 3
 - Contact Mengqi if you did not generate your cookie using NetID
- For final
 - December 14th at 8:30am

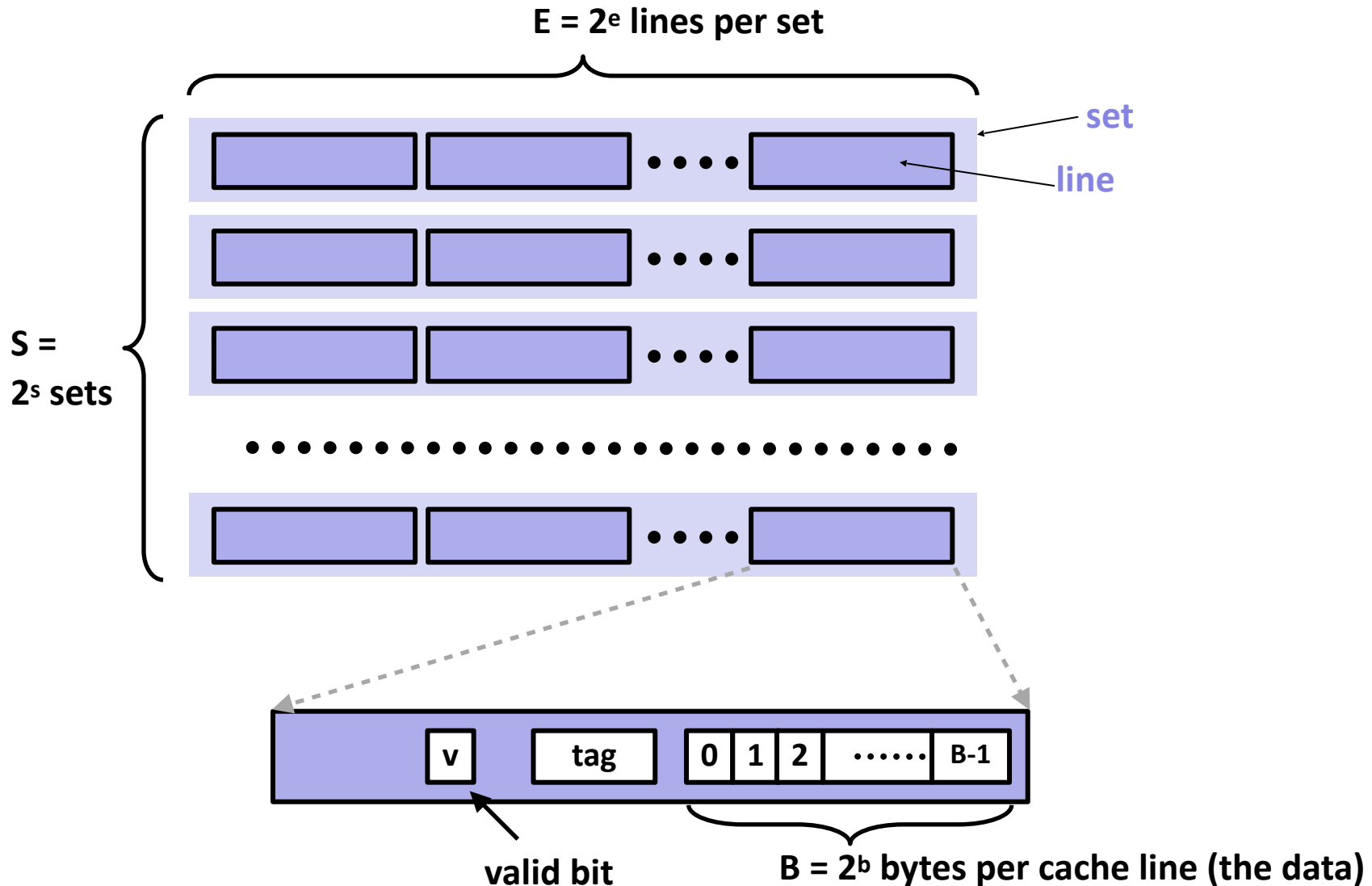
General Cache Organization (S, E, B)



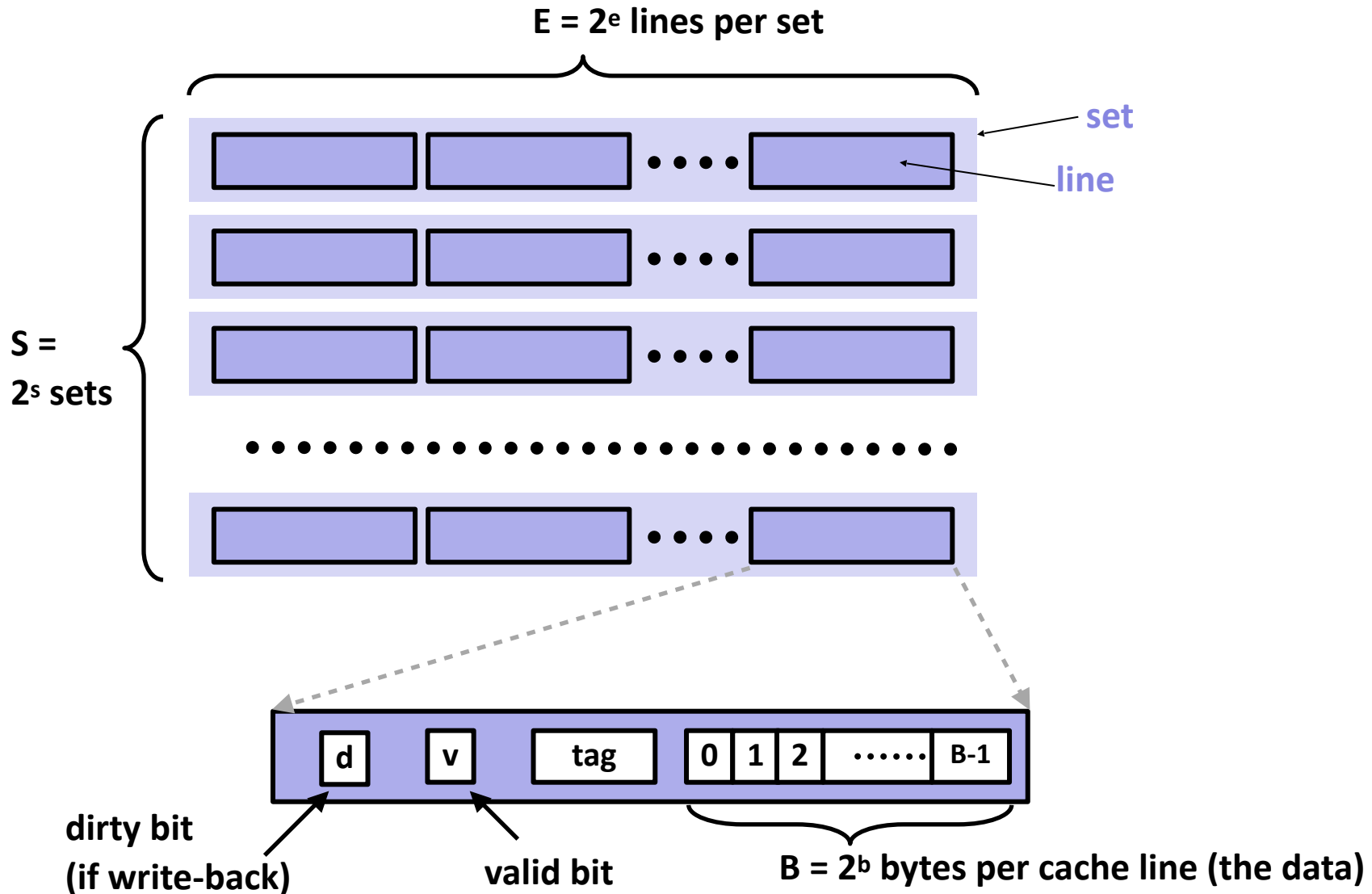
General Cache Organization (S, E, B)



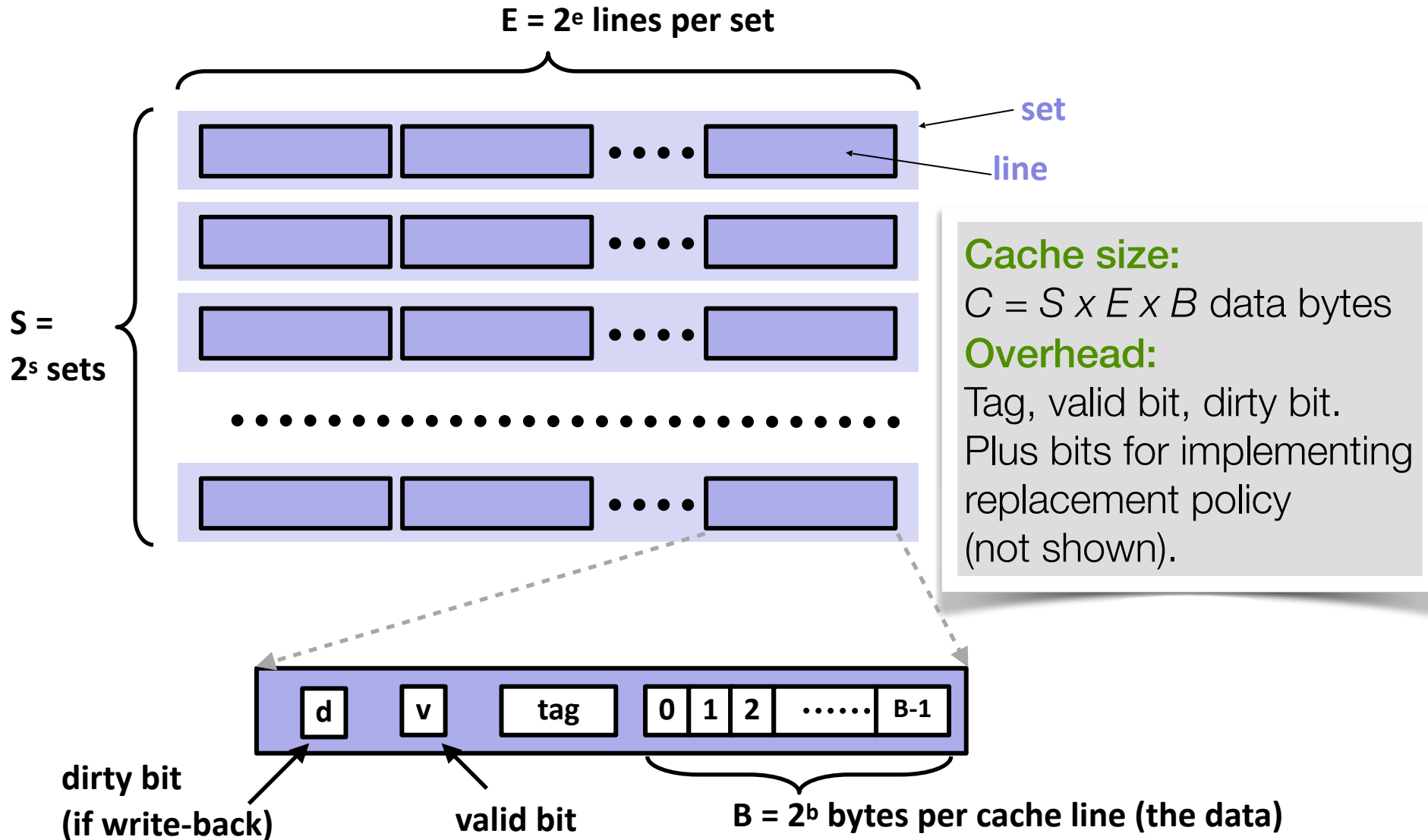
General Cache Organization (S, E, B)



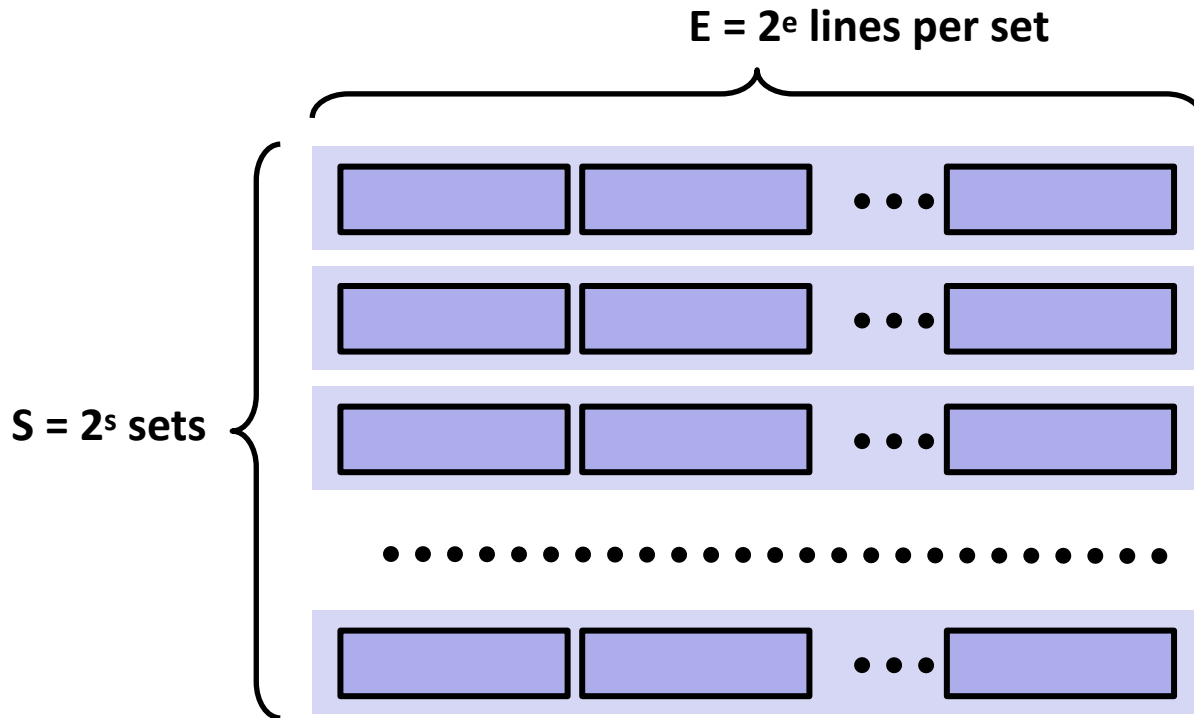
General Cache Organization (S, E, B)



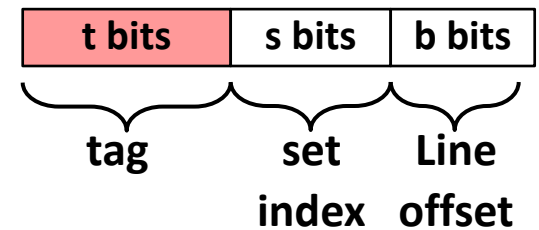
General Cache Organization (S, E, B)



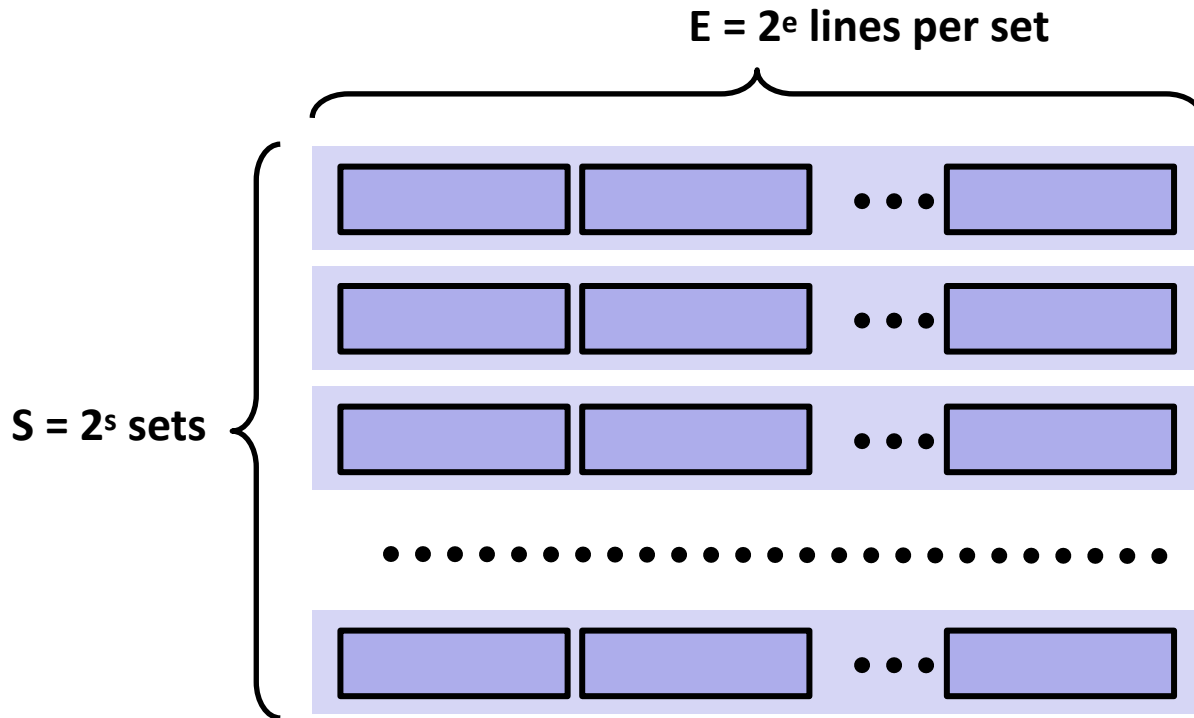
Cache Access



Address of word:

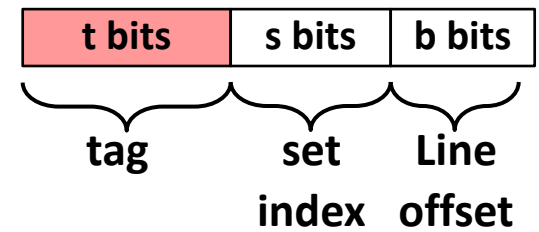


Cache Access



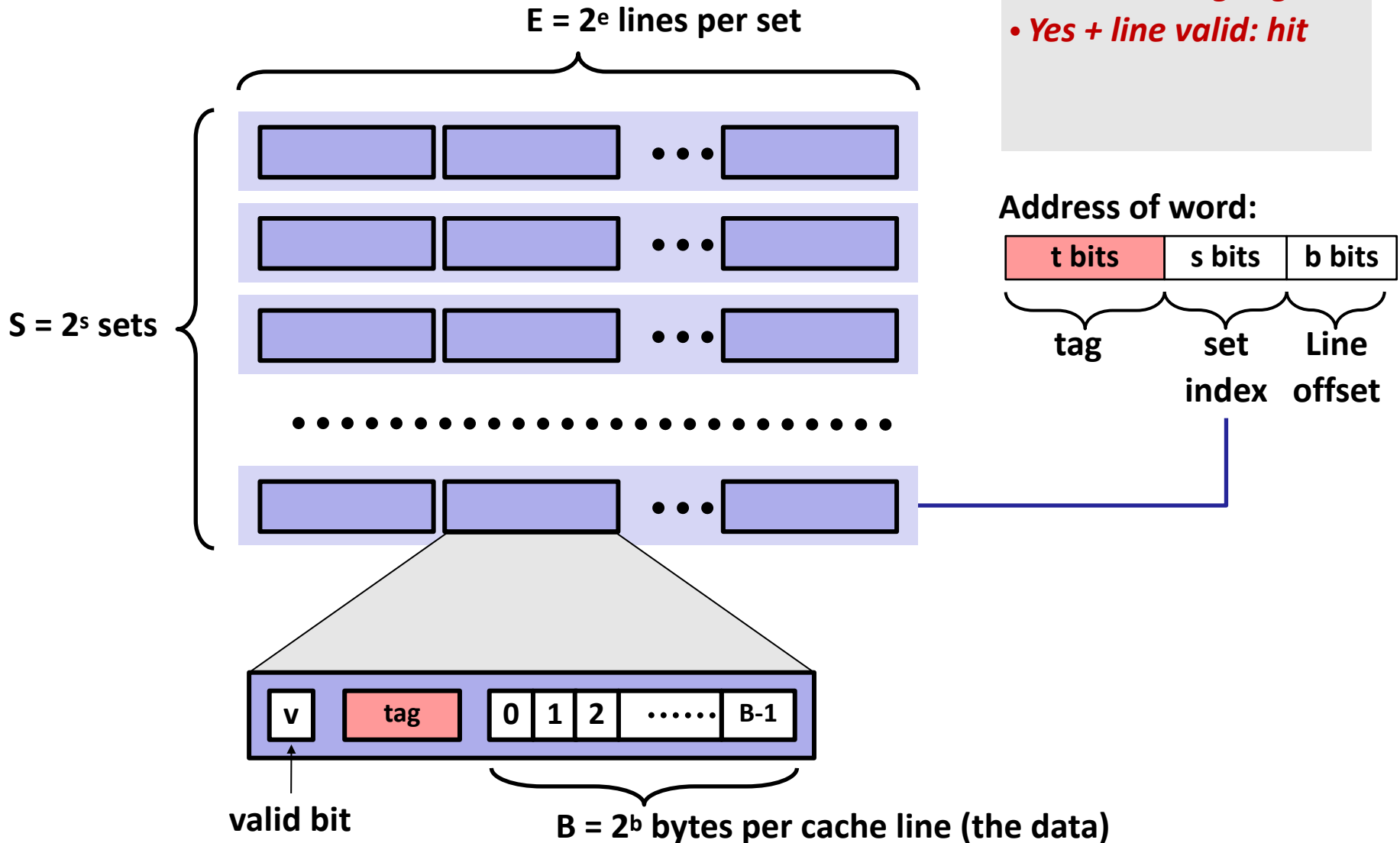
• *Locate set*

Address of word:



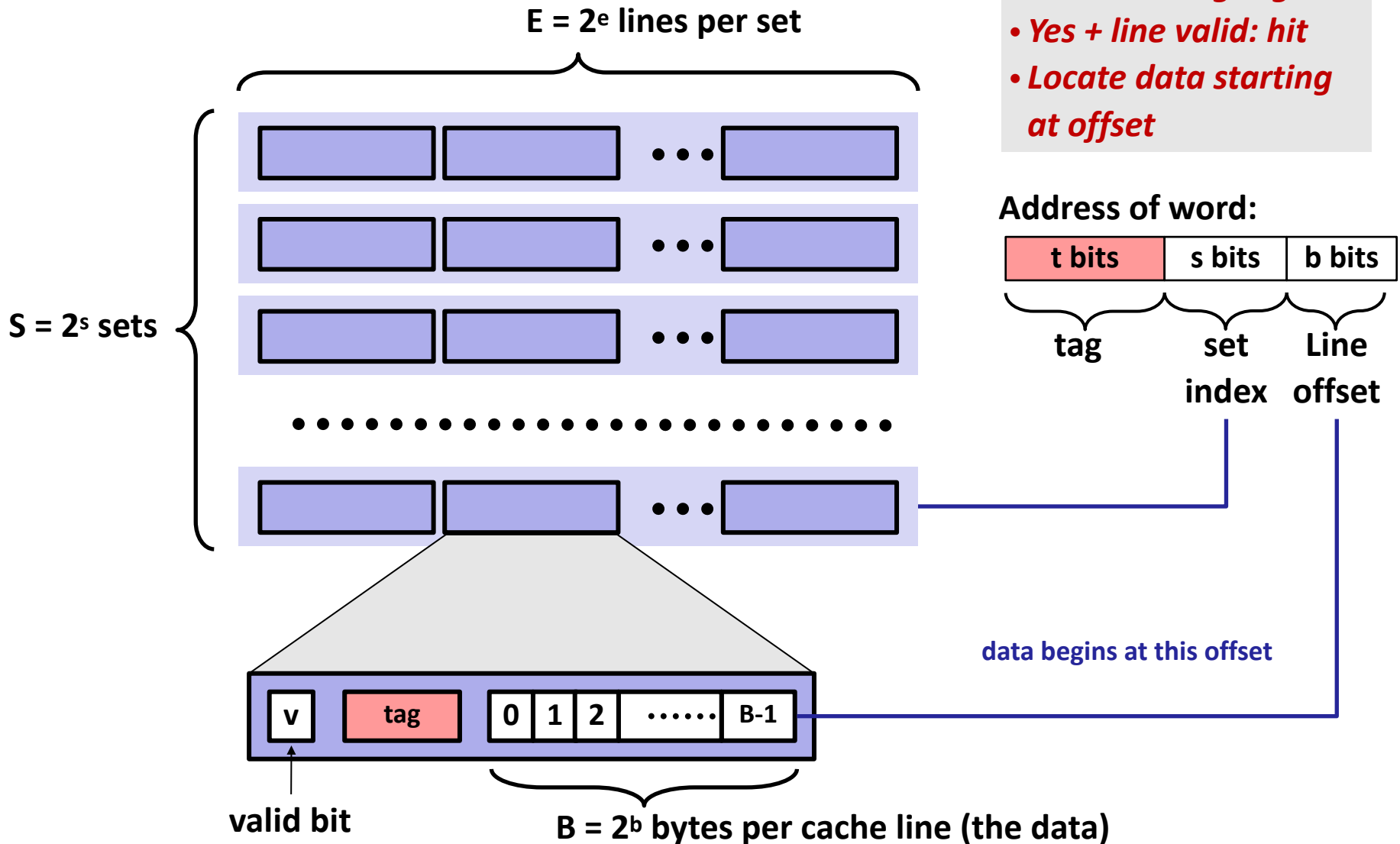
Cache Access

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



Cache Access

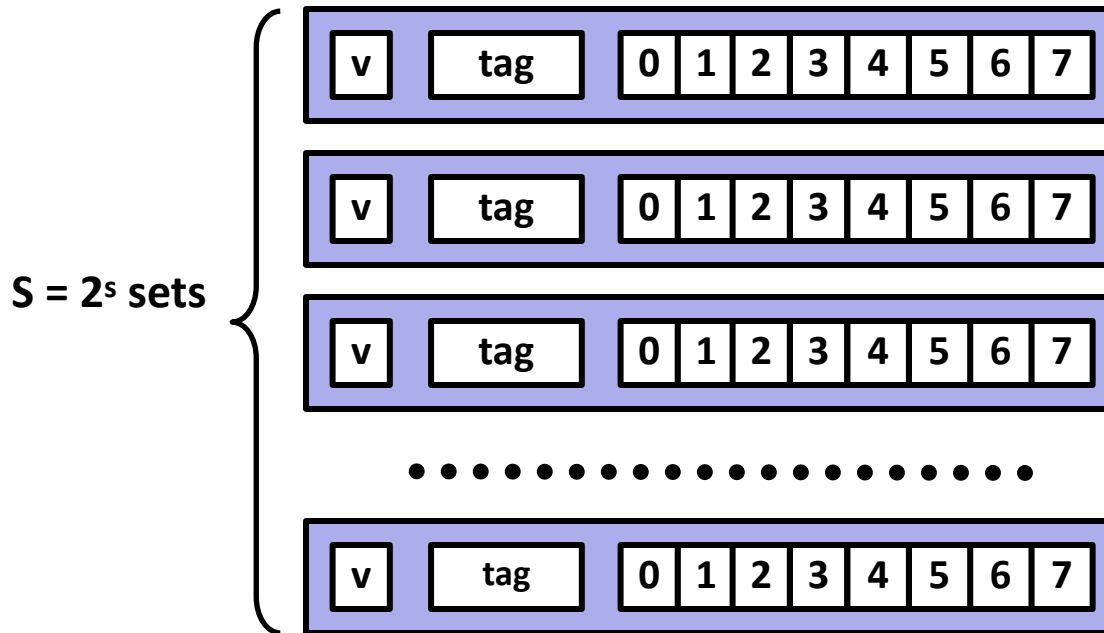
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



Example: Direct Mapped Cache

Direct mapped: One line per set

Assume: cache line size 8 bytes



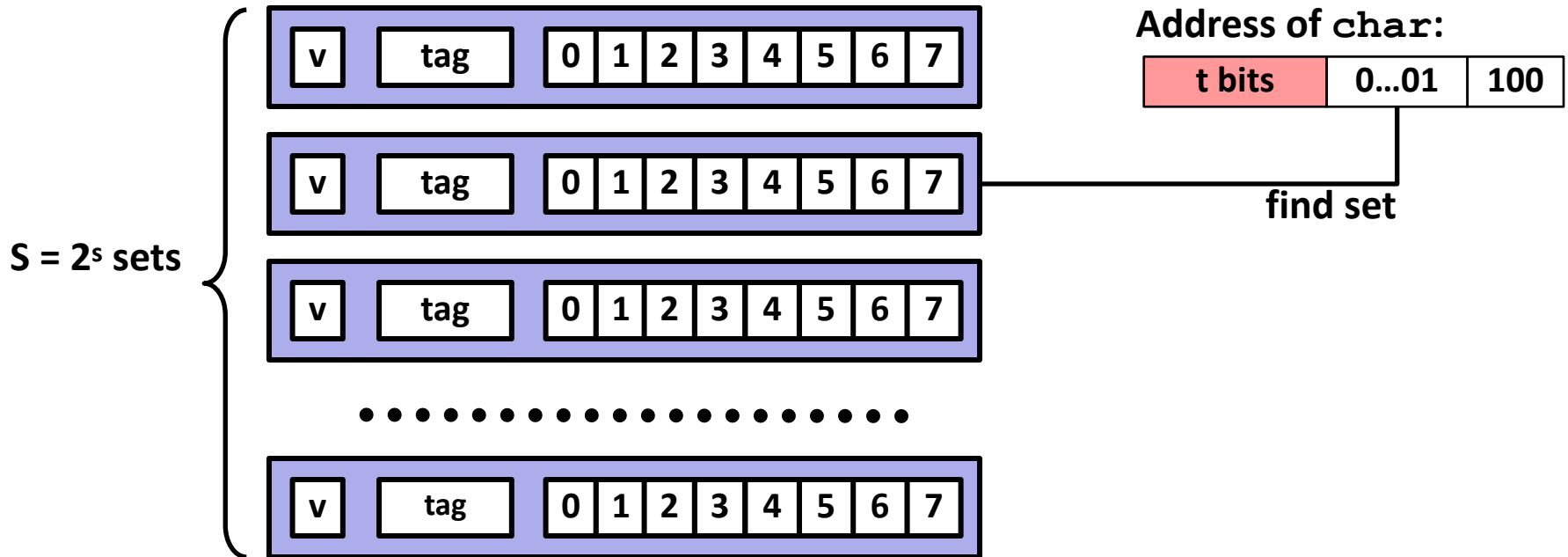
Address of char:

t bits	0...01	100
--------	--------	-----

Example: Direct Mapped Cache

Direct mapped: One line per set

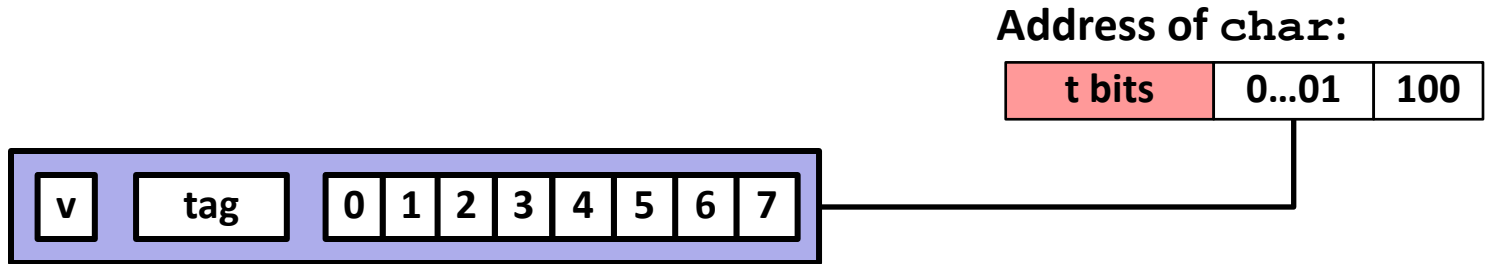
Assume: cache line size 8 bytes



Example: Direct Mapped Cache

Direct mapped: One line per set

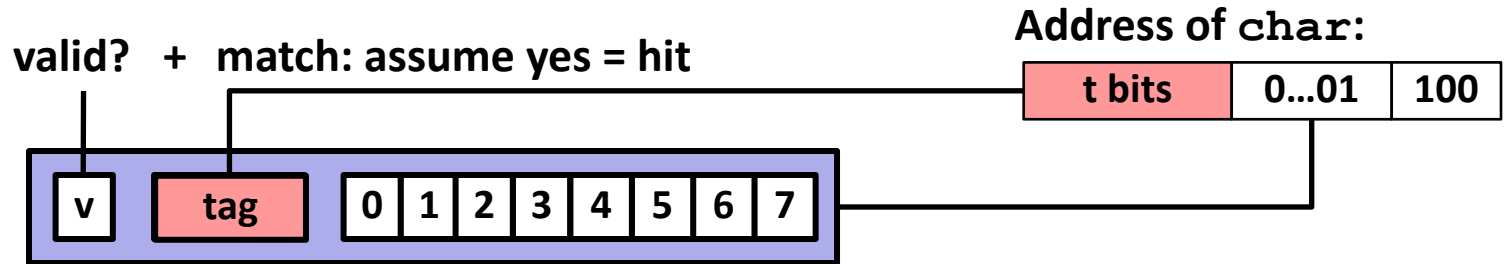
Assume: cache line size 8 bytes



Example: Direct Mapped Cache

Direct mapped: One line per set

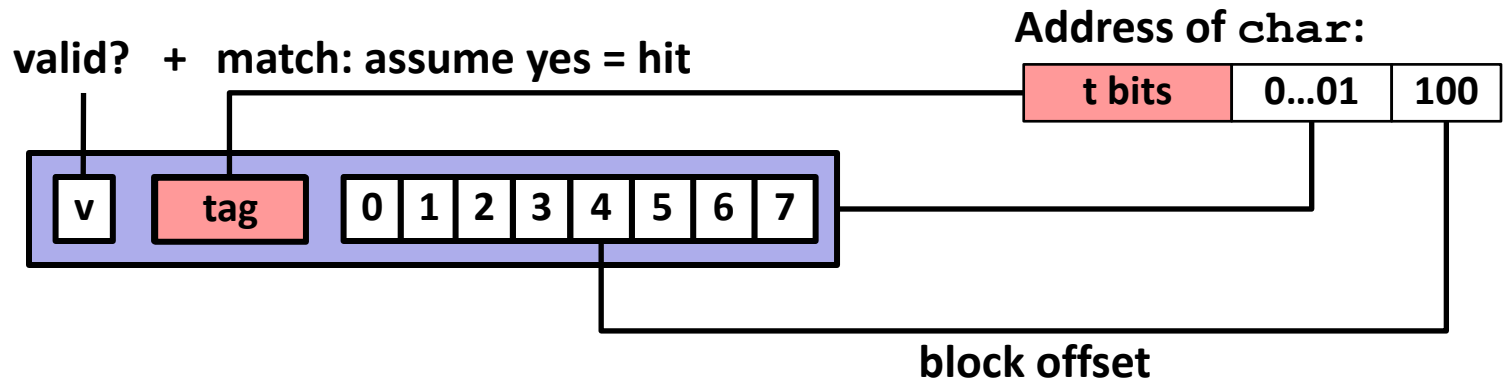
Assume: cache line size 8 bytes



Example: Direct Mapped Cache

Direct mapped: One line per set

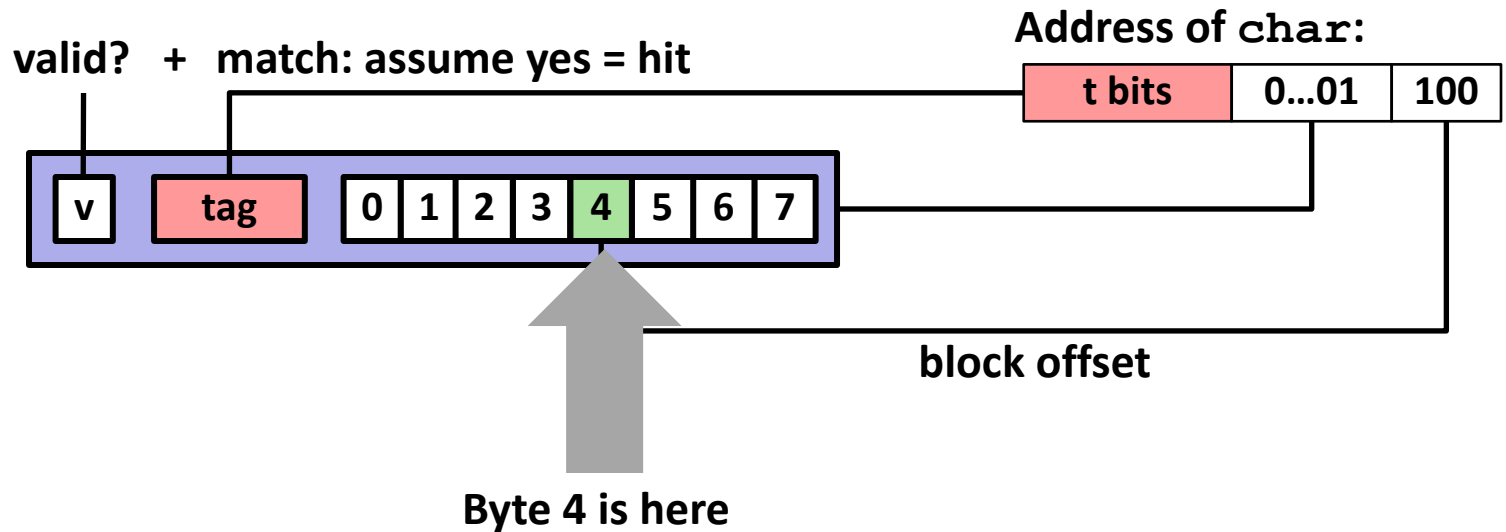
Assume: cache line size 8 bytes



Example: Direct Mapped Cache

Direct mapped: One line per set

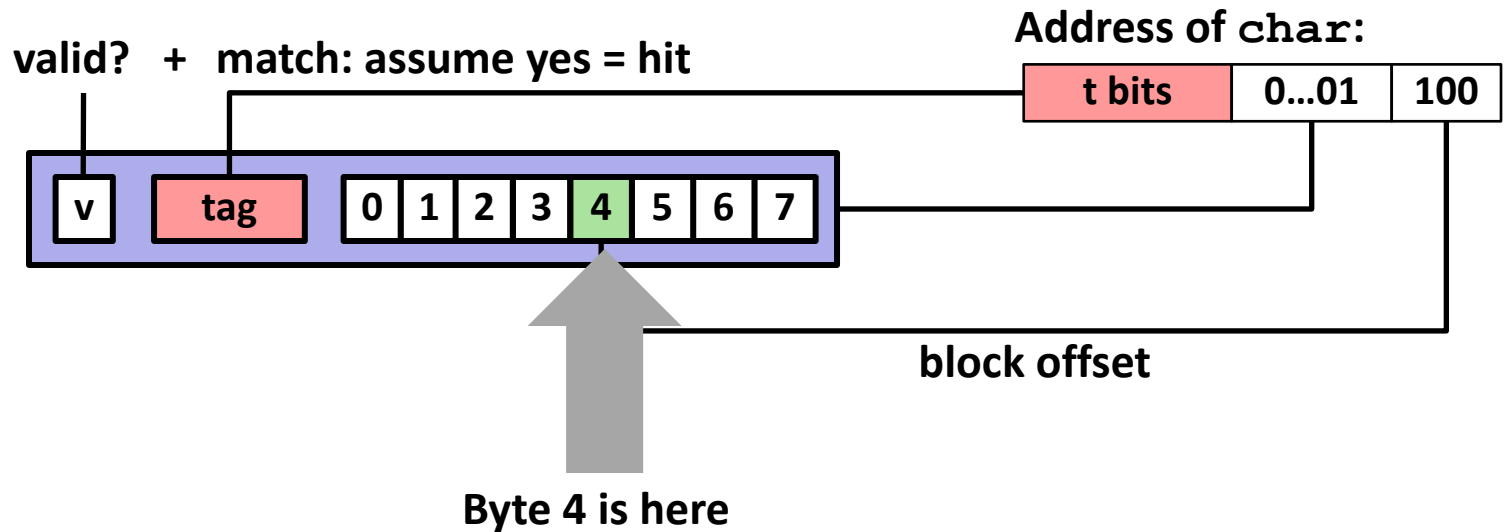
Assume: cache line size 8 bytes



Example: Direct Mapped Cache

Direct mapped: One line per set

Assume: cache line 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 [0000₂],
 1 [0001₂],
 7 [0111₂],
 8 [1000₂],
 0 [0000₂]

	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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	
7	[<u>0111</u> ₂],	
8	[<u>1000</u> ₂],	
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	
7	[<u>0111</u> ₂],	
8	[<u>1000</u> ₂],	
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	
7	[<u>0111</u> ₂],	
8	[<u>1000</u> ₂],	
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	
8	[<u>1000</u> ₂],	
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	
0	[<u>0000</u> ₂]	

	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

← 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> 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	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> 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

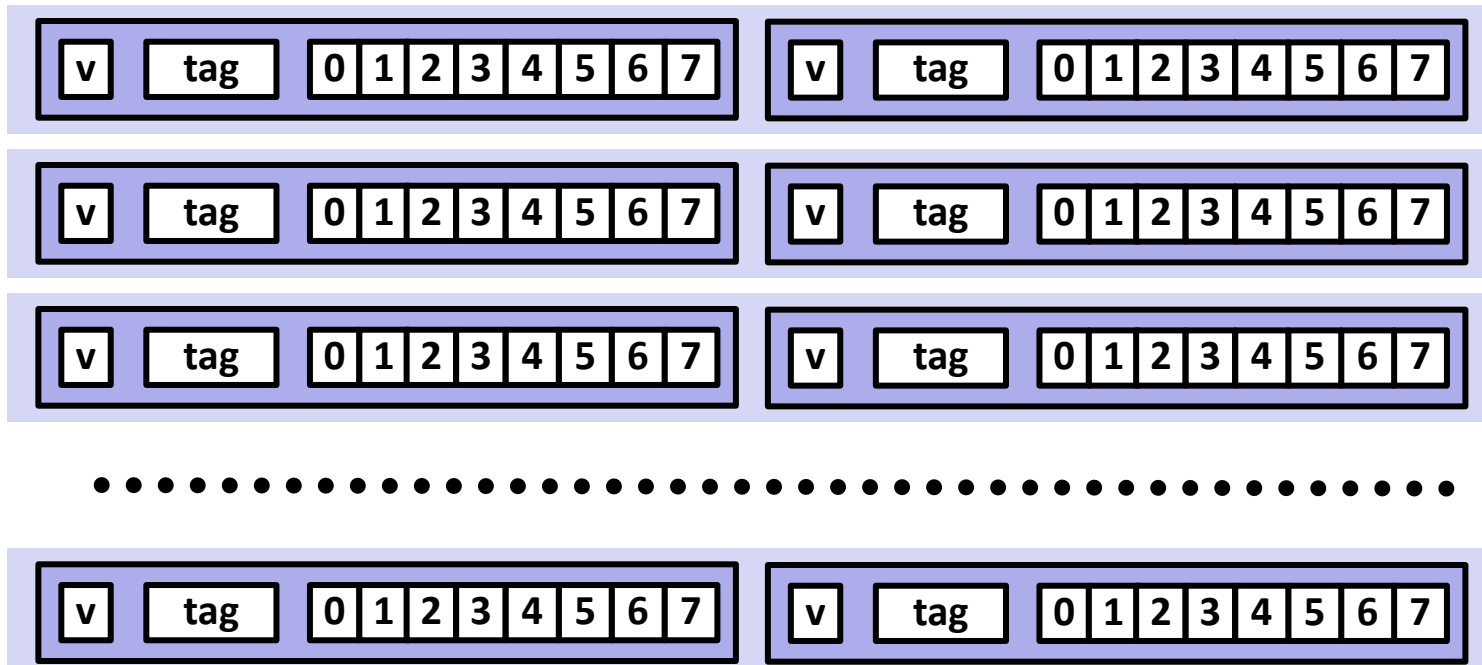
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache line size 8 bytes

Address of short int:

t bits	0...01	100
--------	--------	-----



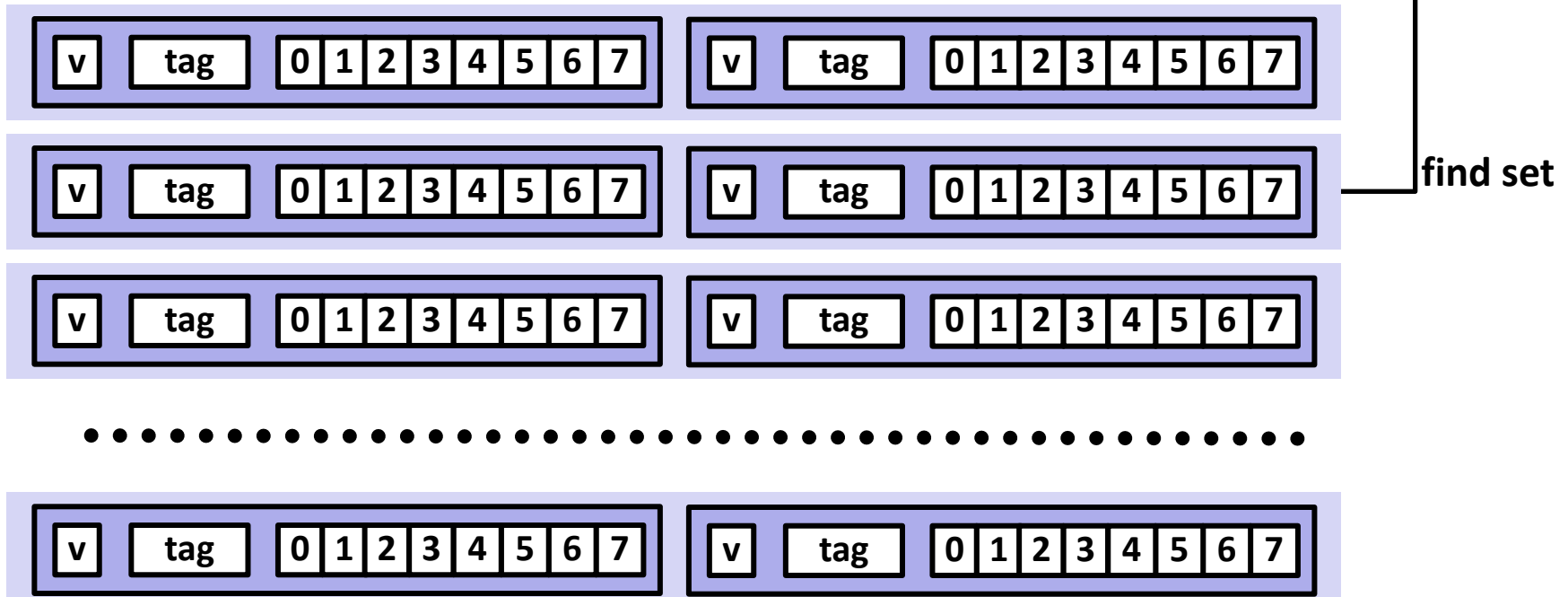
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache line size 8 bytes

Address of short int:

t bits	0...01	100
--------	--------	-----



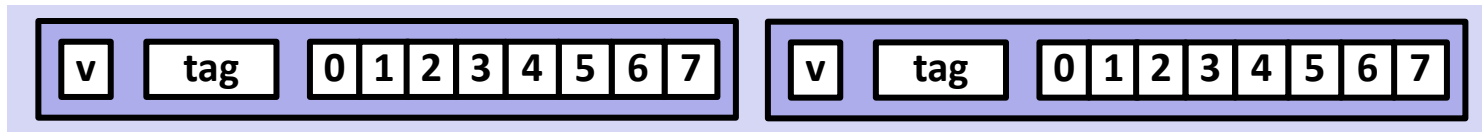
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache line size 8 bytes

Address of short int:

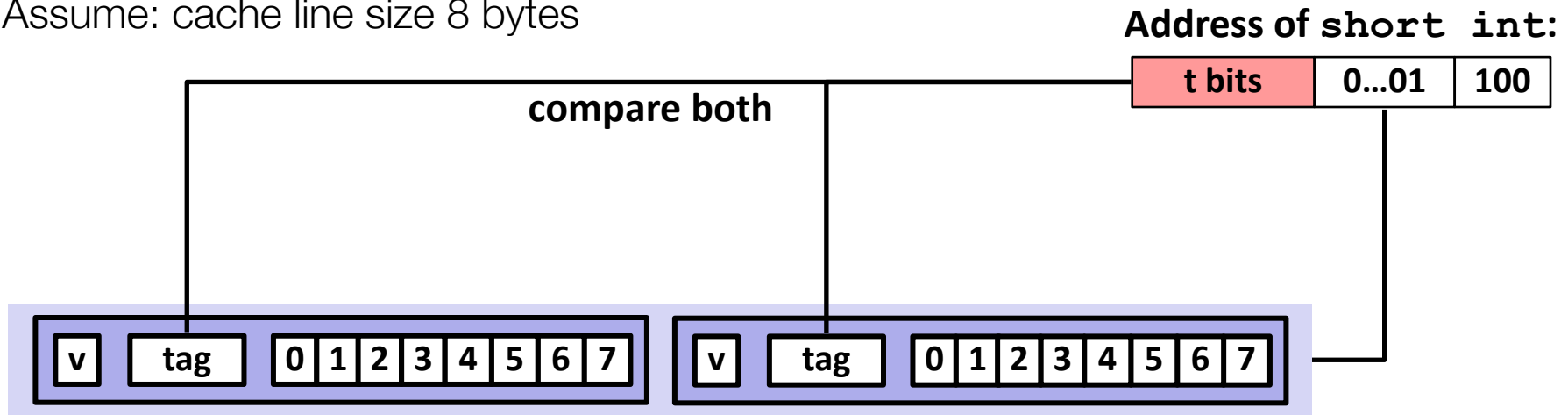
t bits	0...01	100
--------	--------	-----



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

E = 2: Two lines per set

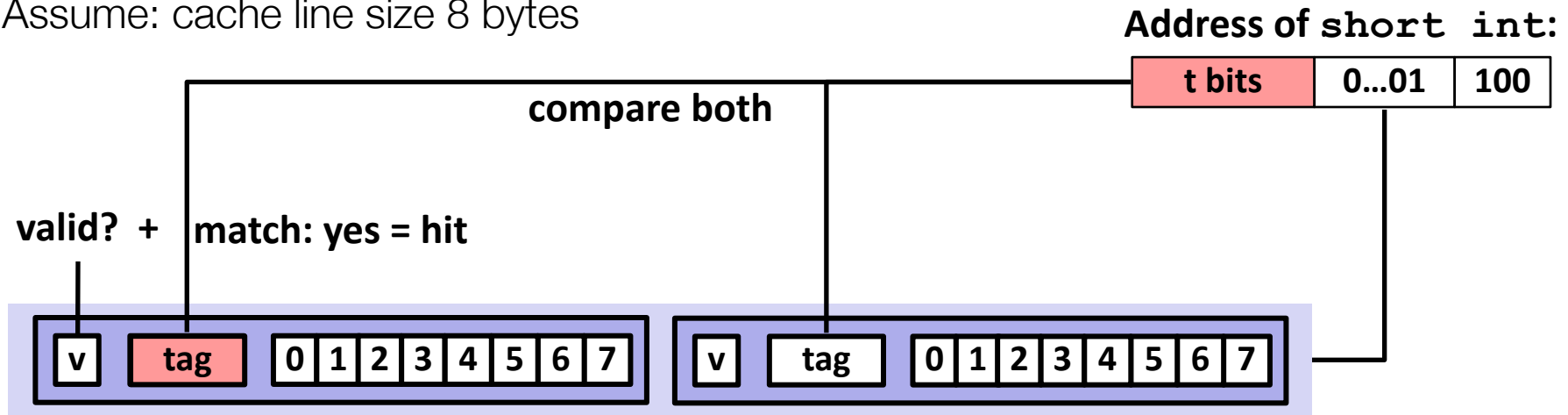
Assume: cache line size 8 bytes



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

E = 2: Two lines per set

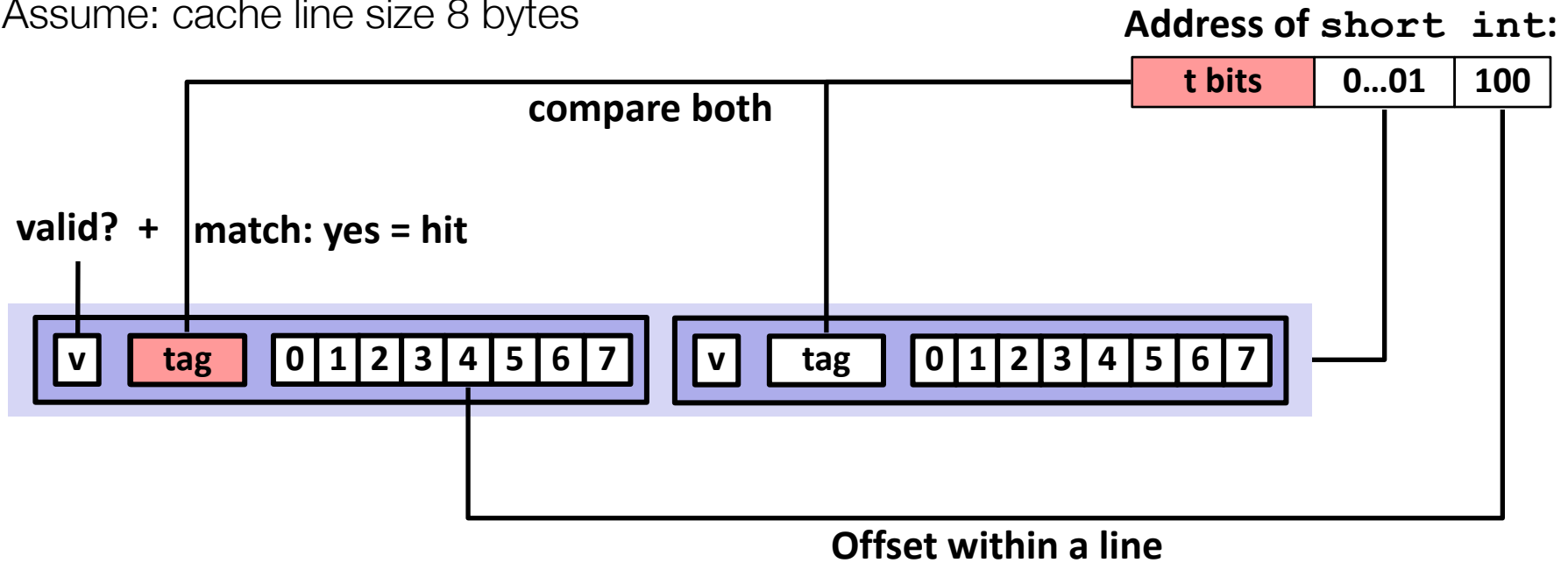
Assume: cache line size 8 bytes



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

E = 2: Two lines per set

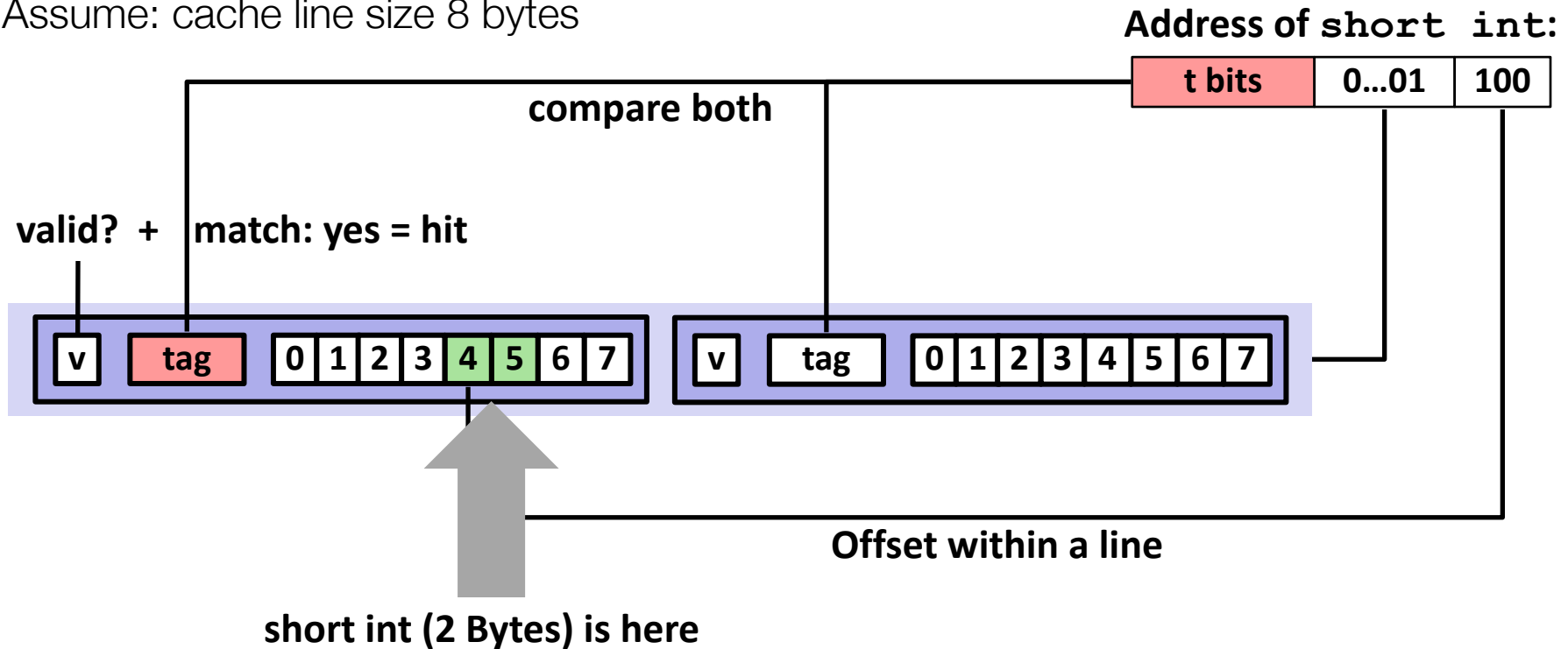
Assume: cache line size 8 bytes



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

E = 2: Two lines per set

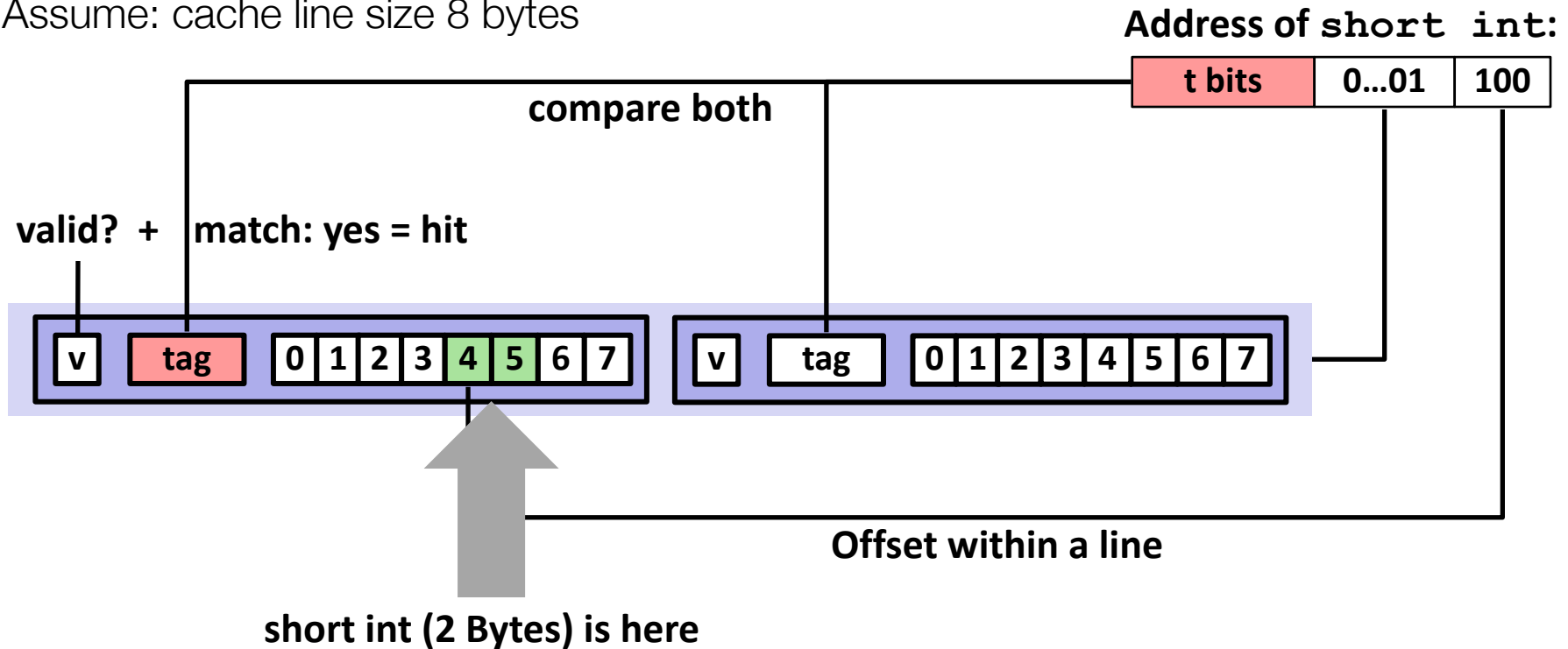
Assume: cache line size 8 bytes



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

E = 2: Two lines per set

Assume: cache line 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 cache lines/set

Address trace (reads, one byte per read):

0 [0000₂],
 1 [0001₂],
 7 [0111₂],
 8 [1000₂],
 0 [0000₂]

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	
7	[01 <u>1</u> 1 ₂],	
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	
7	[01 <u>1</u> 1 ₂],	
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	miss
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	miss
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Line			
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 cache lines/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	miss
0	[00 <u>0</u> 0 ₂]	hit

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

Today

- Processes and Signals: running multiple programs concurrently
 - Processes
 - Process Control
 - Signals

Processes

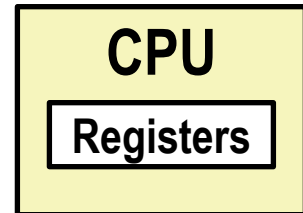
- Definition: A *process* is an instance of a program running in a computer..
 - One of the most profound ideas in computer science
 - Not the same as “program” or “processor”

Processes

- Definition: A *process* is an instance of a program running in a computer..
 - 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
 - Done by the OS kernel through “context switching”
 - *Private address space*
 - Each program seems to have exclusive use of main memory.
 - Provided by OS through “virtual memory”

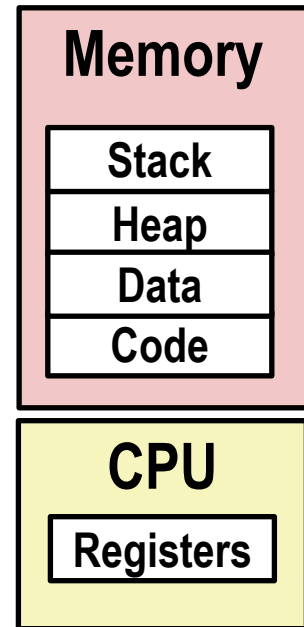
Processes

- Definition: A *process* is an instance of a program running in a computer..
 - 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
 - Done by the OS kernel through “context switching”
 - *Private address space*
 - Each program seems to have exclusive use of main memory.
 - Provided by OS through “virtual memory”

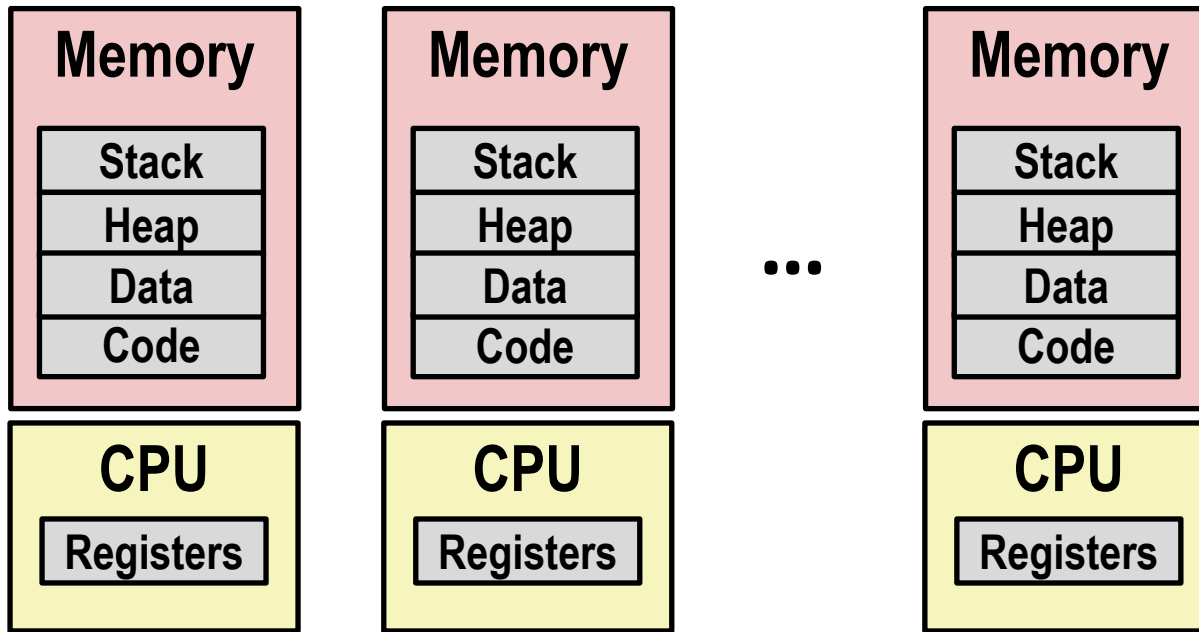


Processes

- Definition: A *process* is an instance of a program running in a computer..
 - 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
 - Done by the OS kernel through “context switching”
 - *Private address space*
 - Each program seems to have exclusive use of main memory.
 - Provided by OS through “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

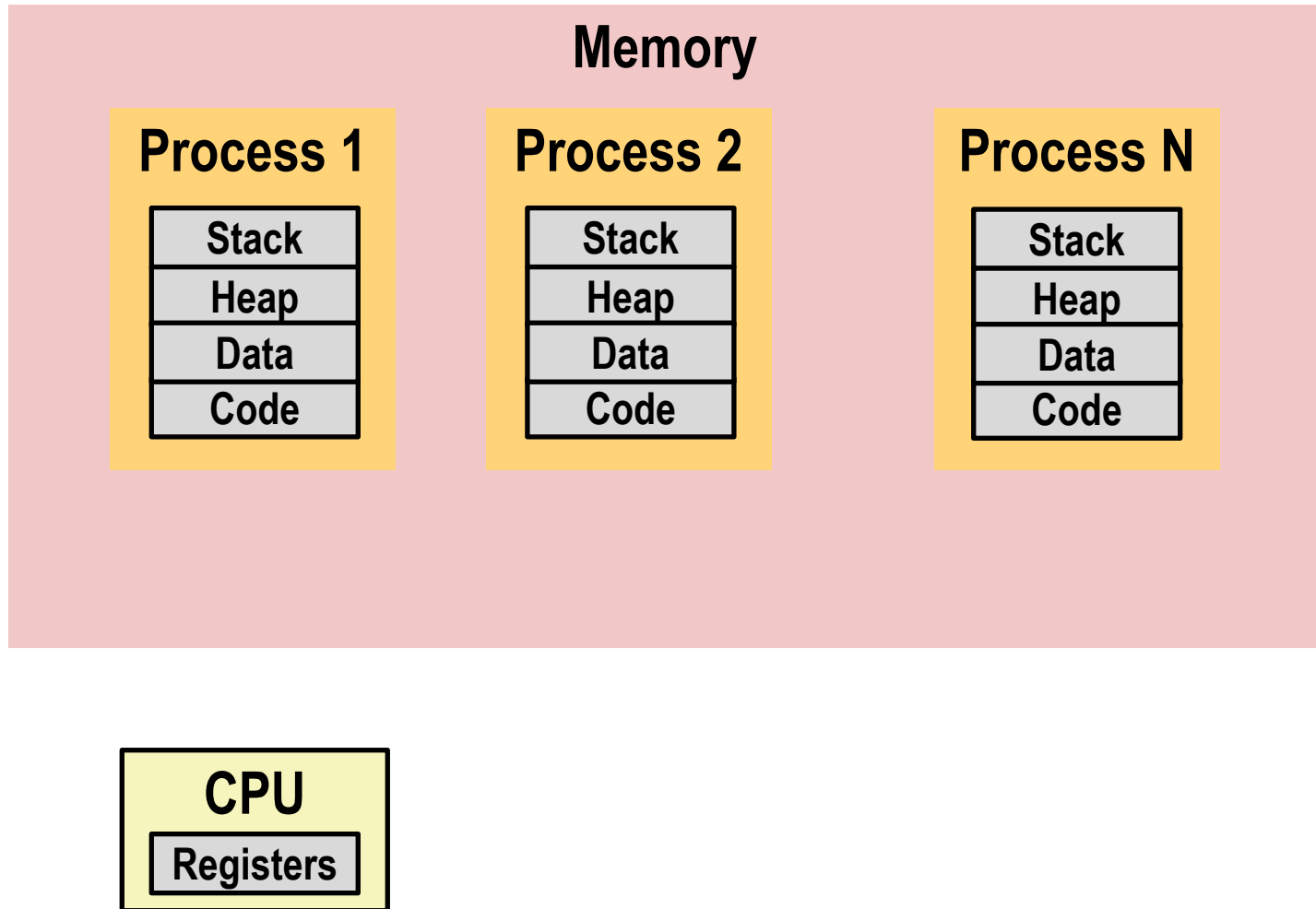
Multiprocessing Example

```
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads
Load Avg: 1.03, 1.13, 1.14 CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

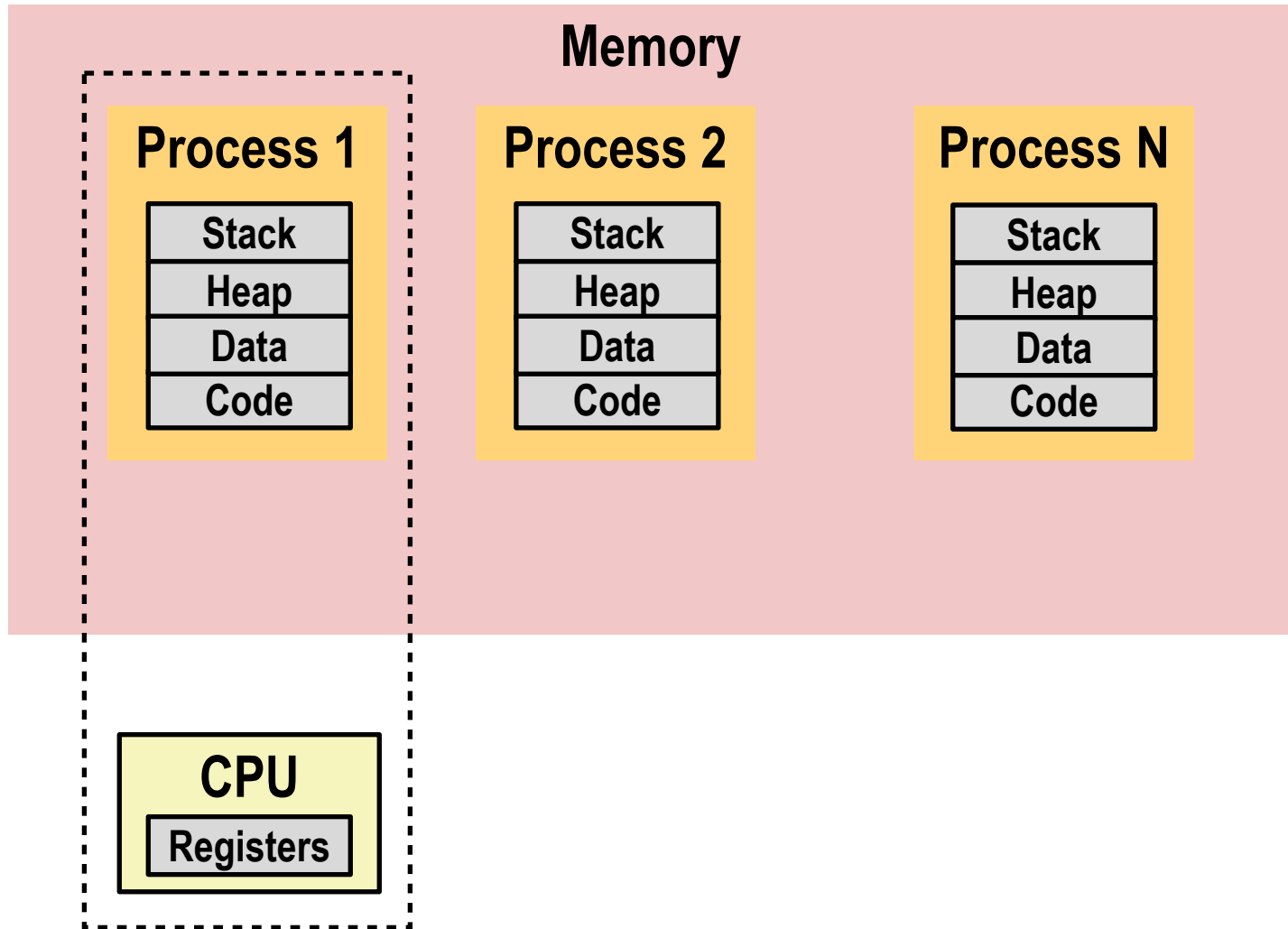
PID    COMMAND    %CPU TIME    #TH    #WQ    #PORT    #MREG    RPRVT    RSHRD    RSIZE    VPRVT    VSIZE
99217-  Microsoft Of 0.0 02:28.34 4      1      202     418     21M     24M     21M     66M     763M
99051   usbmuxd     0.0 00:04.10 3      1      47      66      436K    216K    480K    60M     2422M
99006   iTunesHelper 0.0 00:01.23 2      1      55      78      728K    3124K   1124K   43M     2429M
84286   bash        0.0 00:00.11 1      0      20      24      224K    732K    484K    17M     2378M
84285   xterm       0.0 00:00.83 1      0      32      73      656K    872K    692K    9728K   2382M
55939-  Microsoft Ex 0.3 21:58.97 10     3      360     954     16M     65M     46M     114M    1057M
54751   sleep       0.0 00:00.00 1      0      17      20      92K     212K    360K    9632K   2370M
54739   launchdadd  0.0 00:00.00 2      1      33      50      488K    220K    1736K   48M     2409M
54737   top         6.5 00:02.53 1/1    0      30      29      1416K   216K    2124K   17M     2378M
54719   automountd  0.0 00:00.02 7      1      53      64      860K    216K    2184K   53M     2413M
54701   ocspd       0.0 00:00.05 4      1      61      54      1268K   2644K   3132K   50M     2426M
54661   Grab        0.6 00:02.75 6      3      222+    389+    15M+    26M+    40M+    75M+    2556M+
54659   cookied     0.0 00:00.15 2      1      40      61      3316K   224K    4088K   42M     2411M
53919   ...         0.0 00:00.07 1      1      50      81      7699K   7449K   16M     49M     2470M
```

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

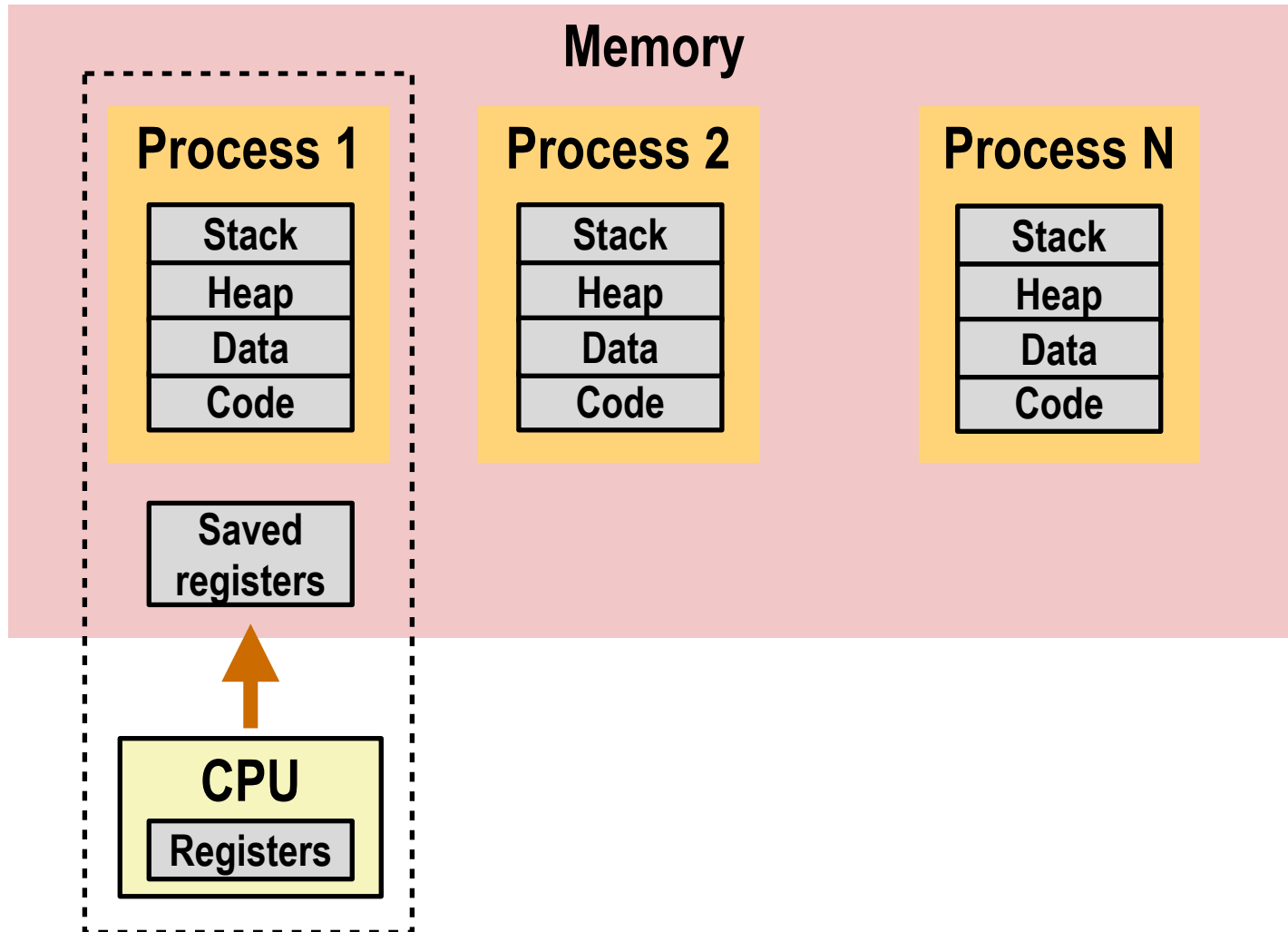
Multiprocessing Illustration



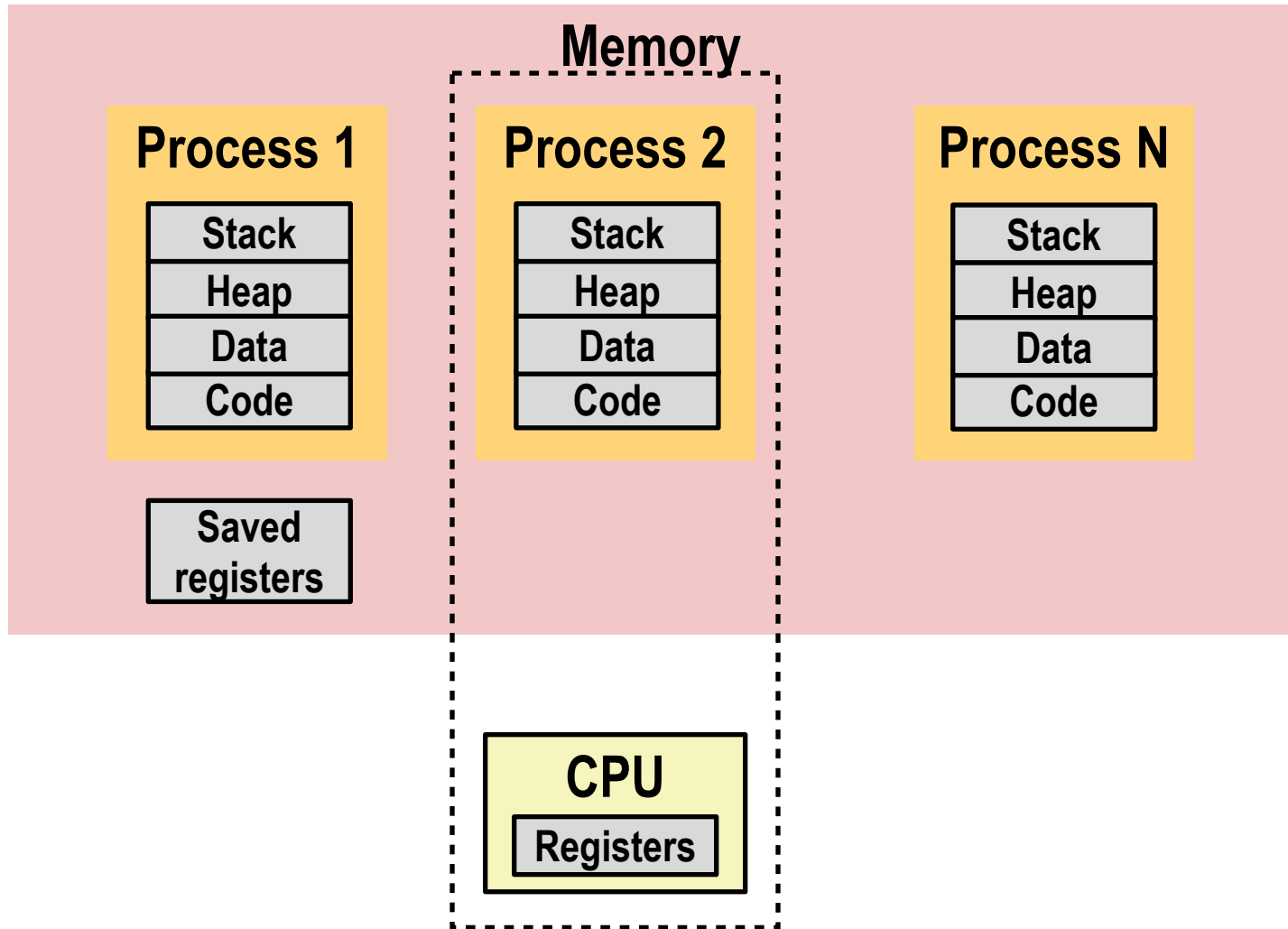
Multiprocessing Illustration



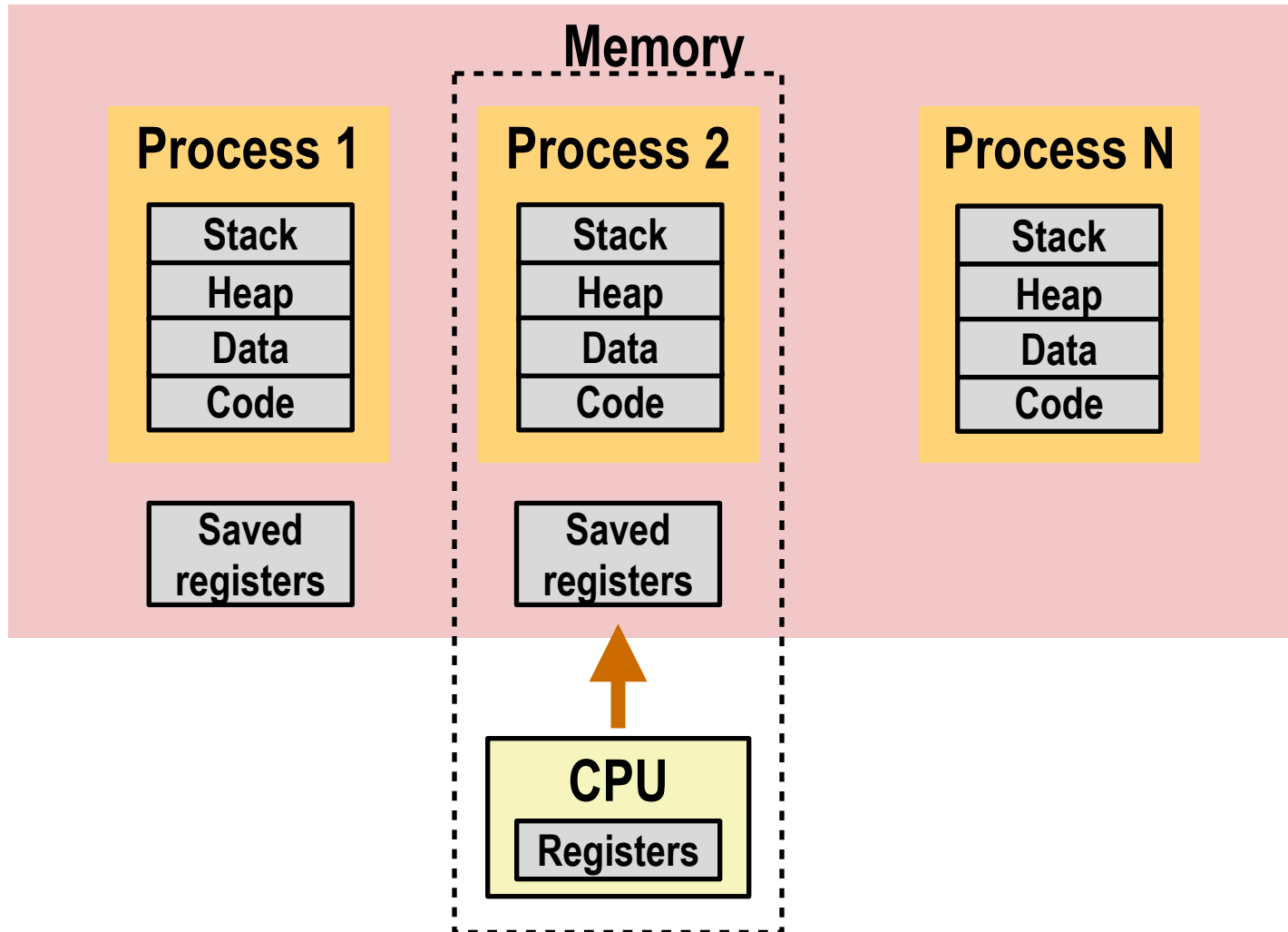
Multiprocessing Illustration



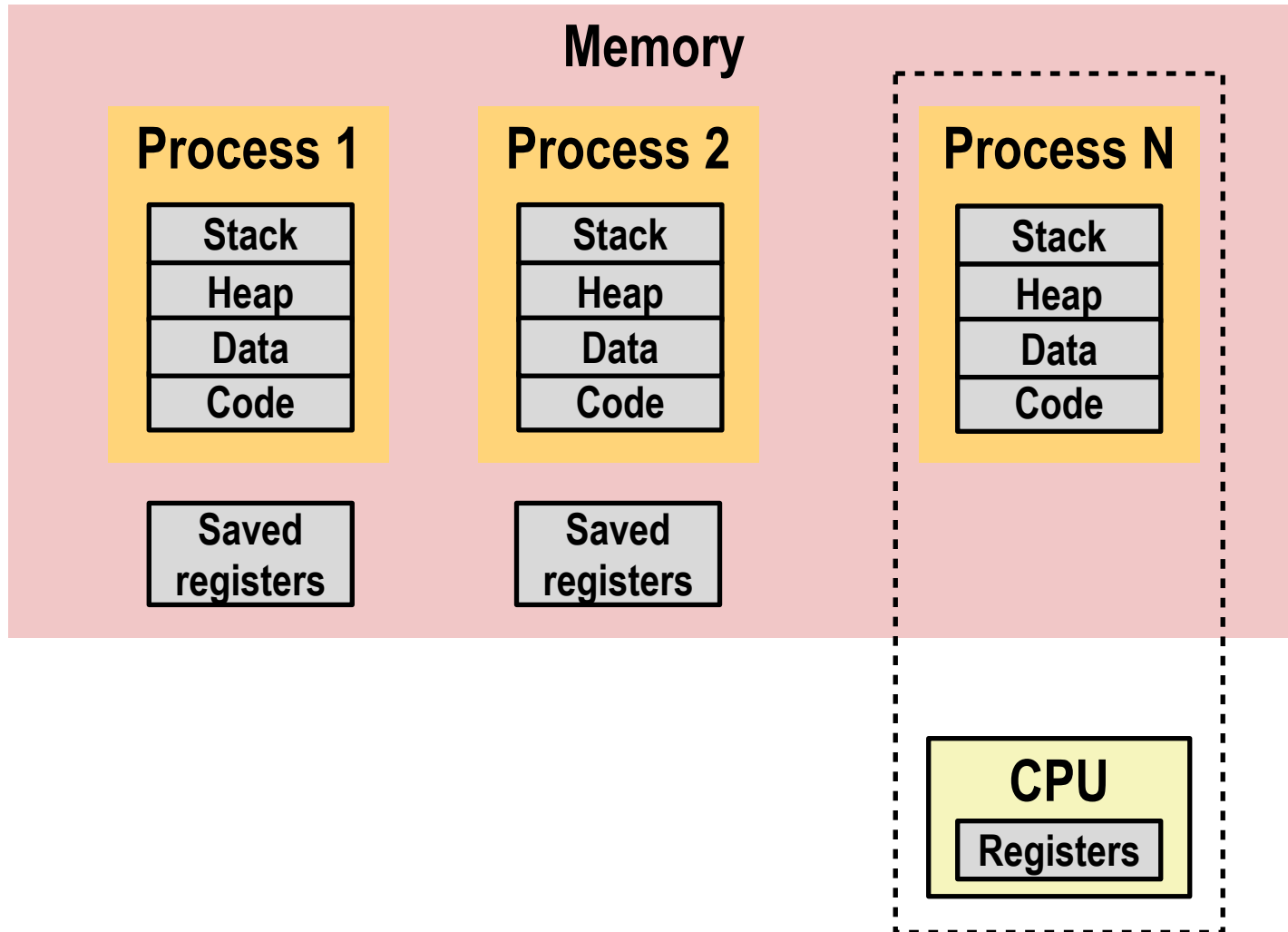
Multiprocessing Illustration



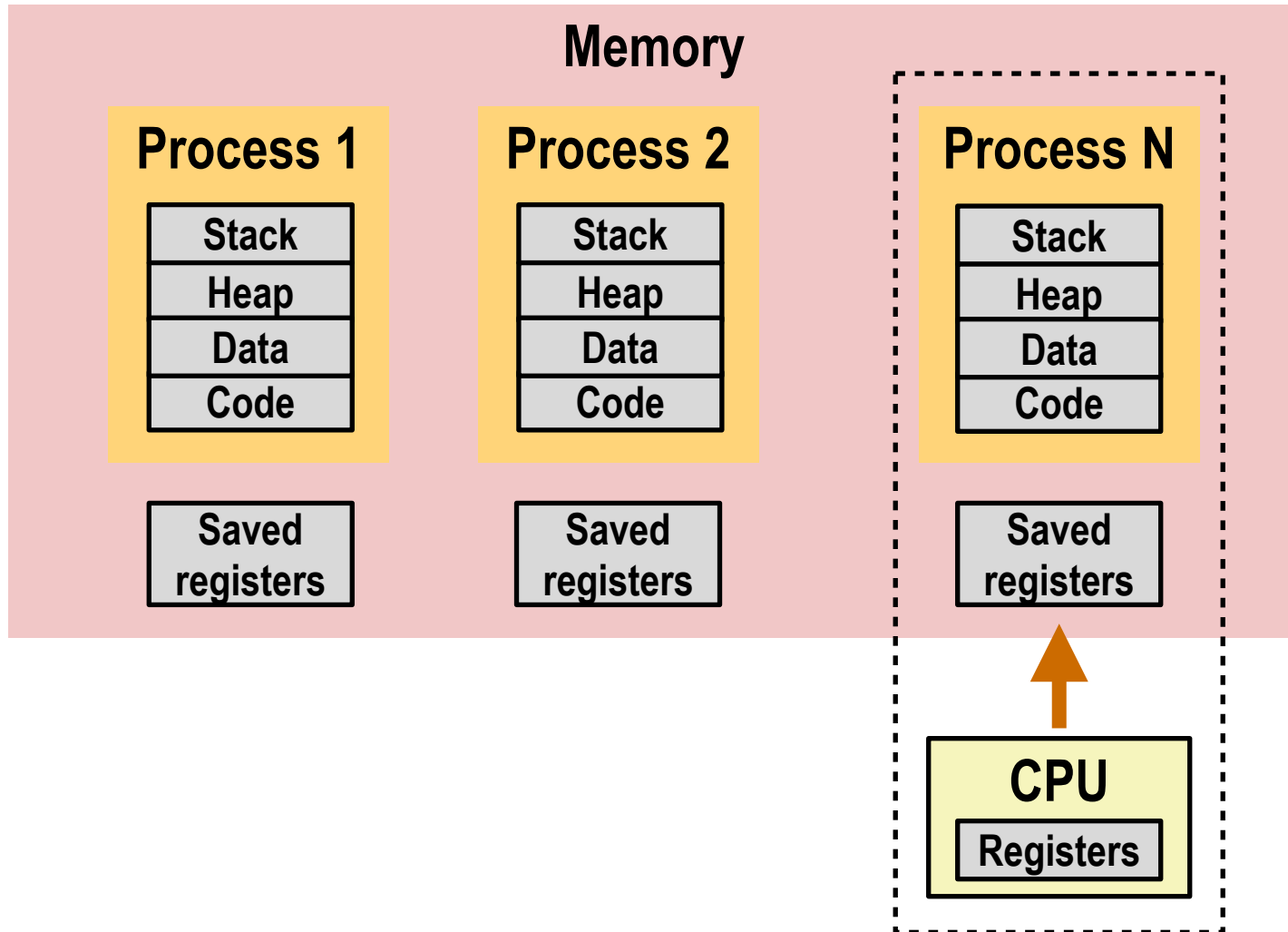
Multiprocessing Illustration



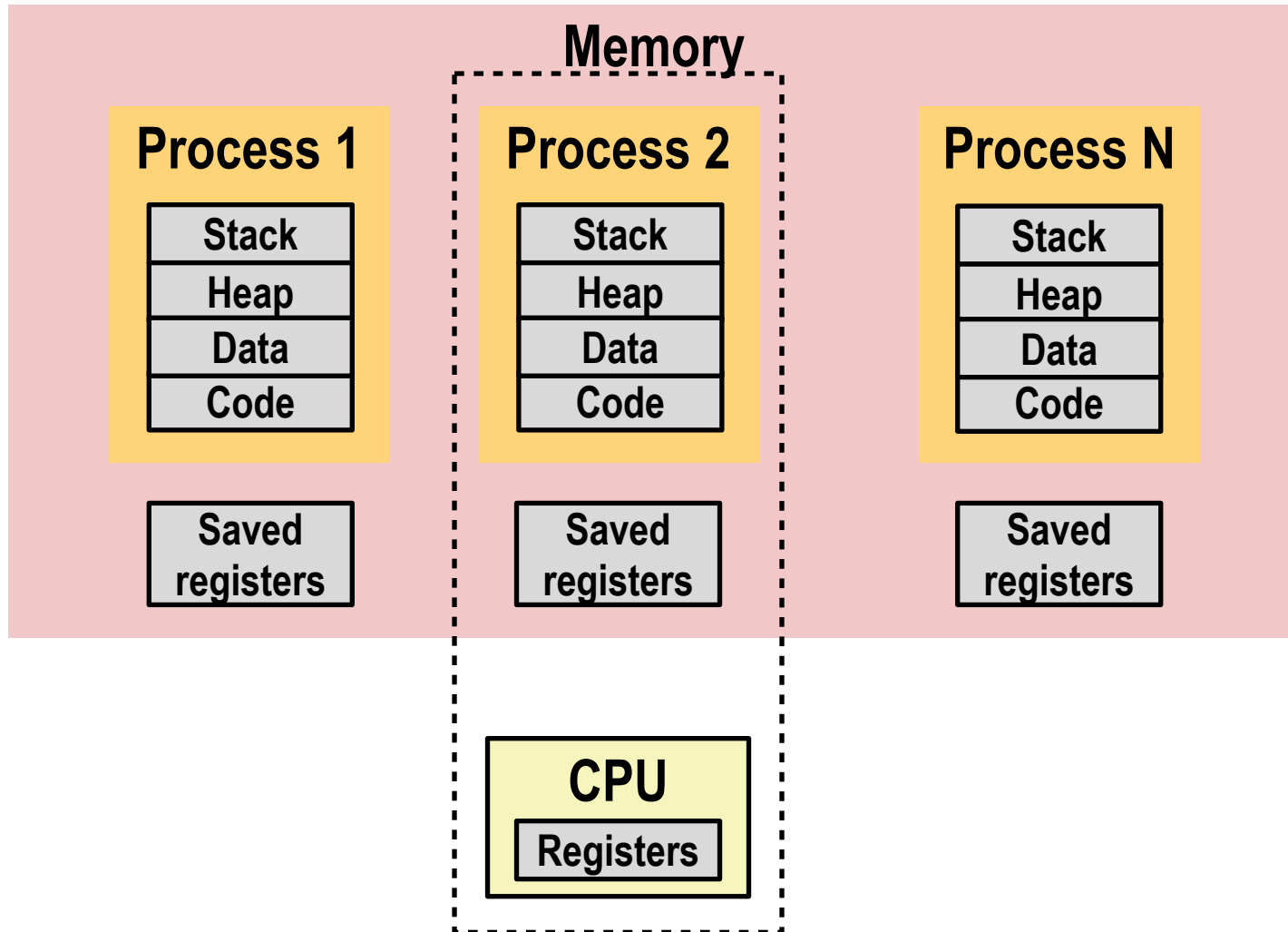
Multiprocessing Illustration



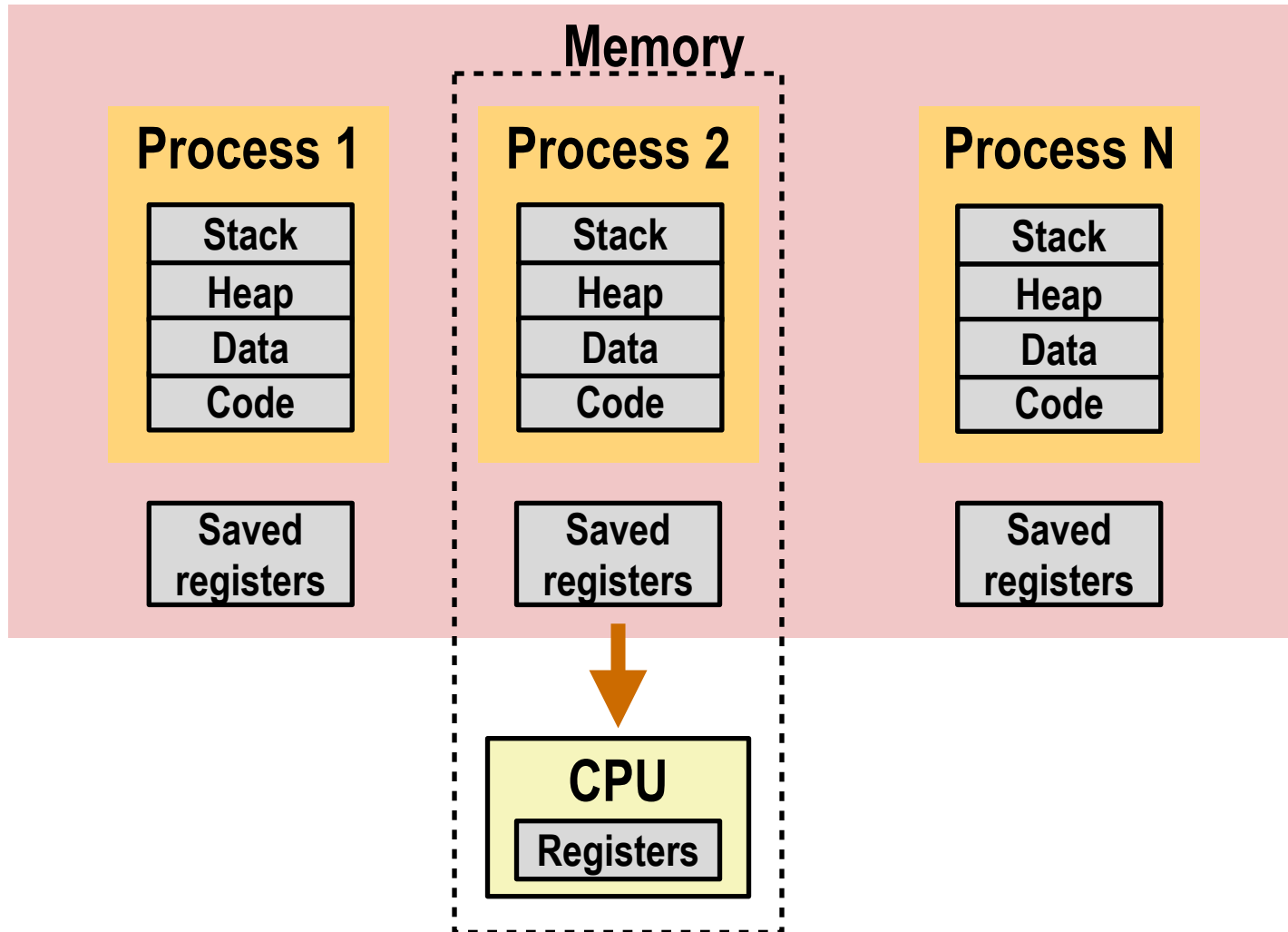
Multiprocessing Illustration



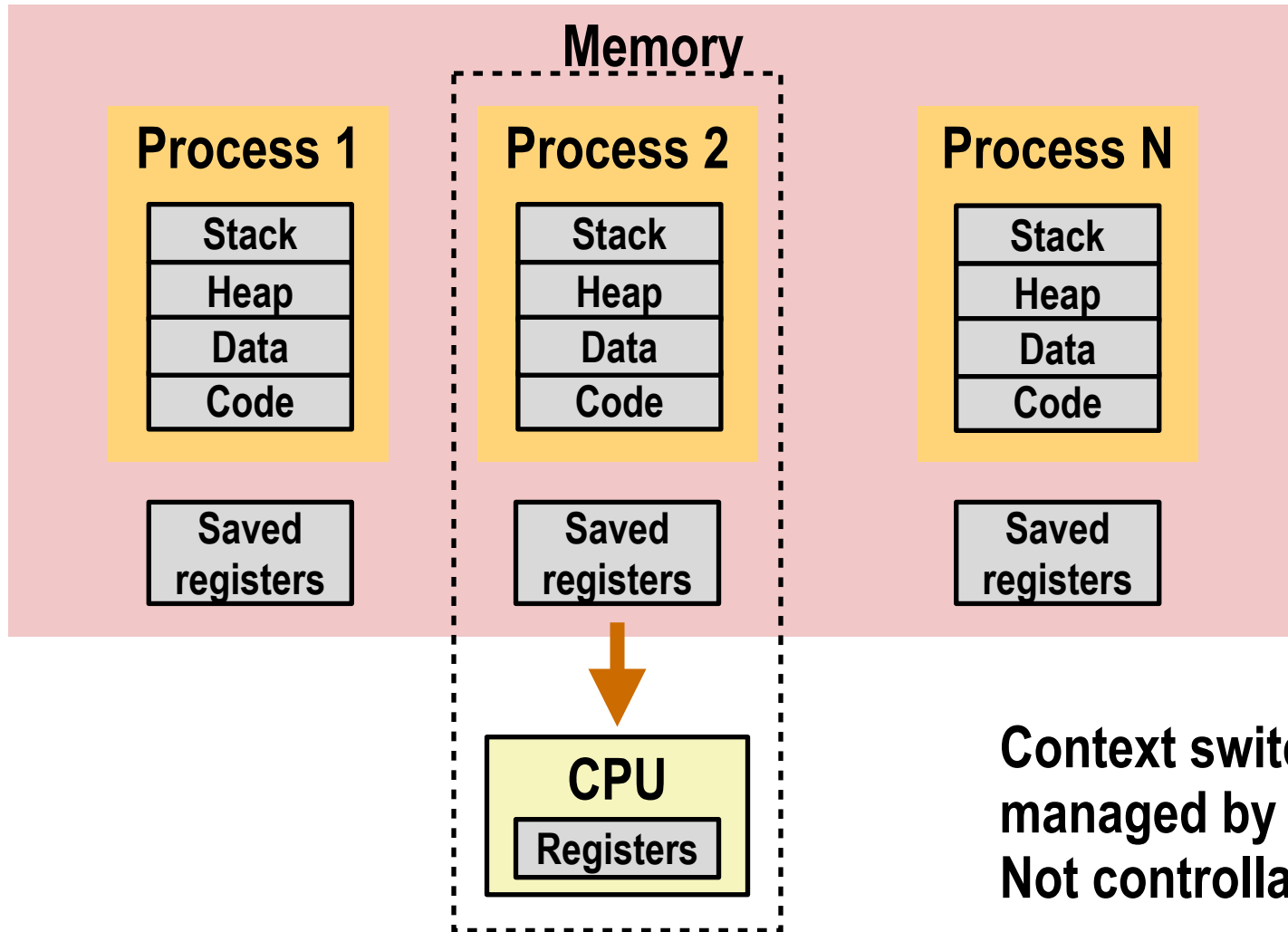
Multiprocessing Illustration



Multiprocessing Illustration

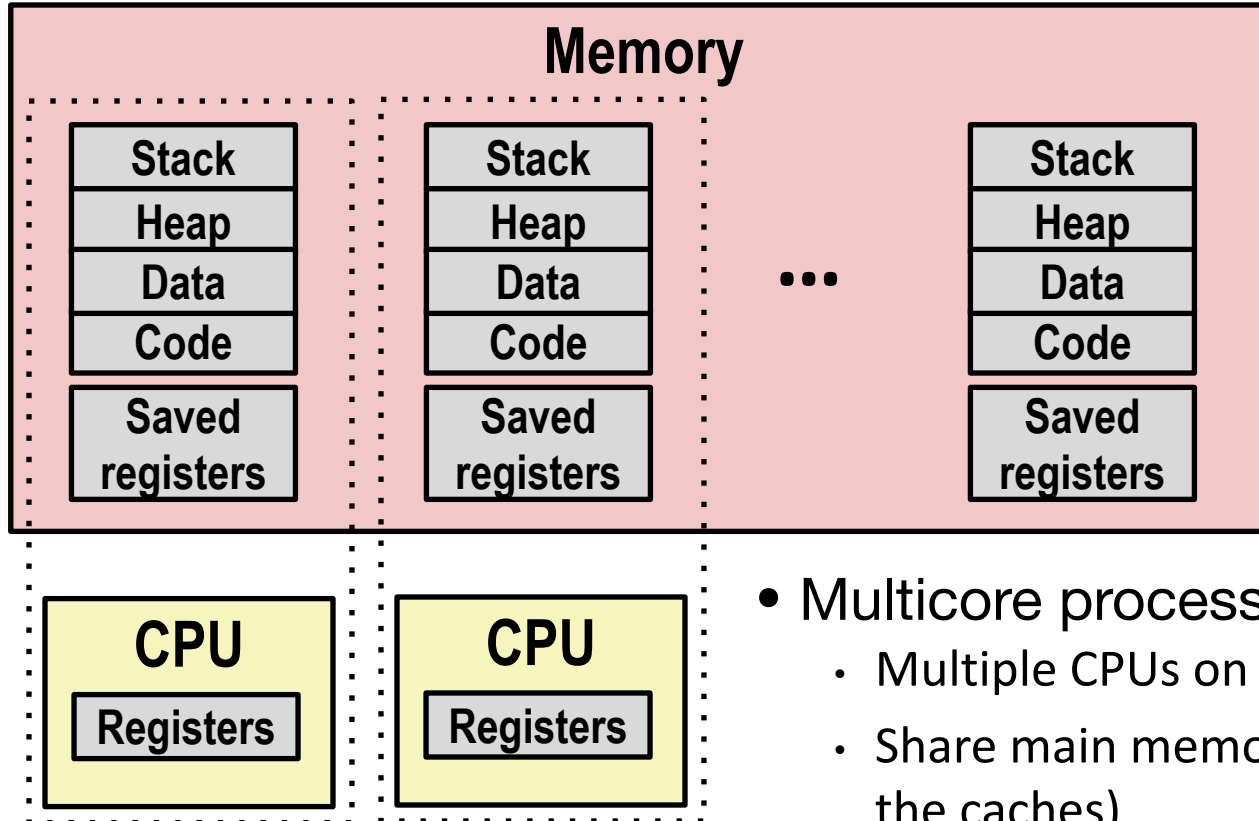


Multiprocessing Illustration



**Context switch
managed by the OS.
Not controllable by
programmers.**

Multiprocessing: The Multi-Core Case



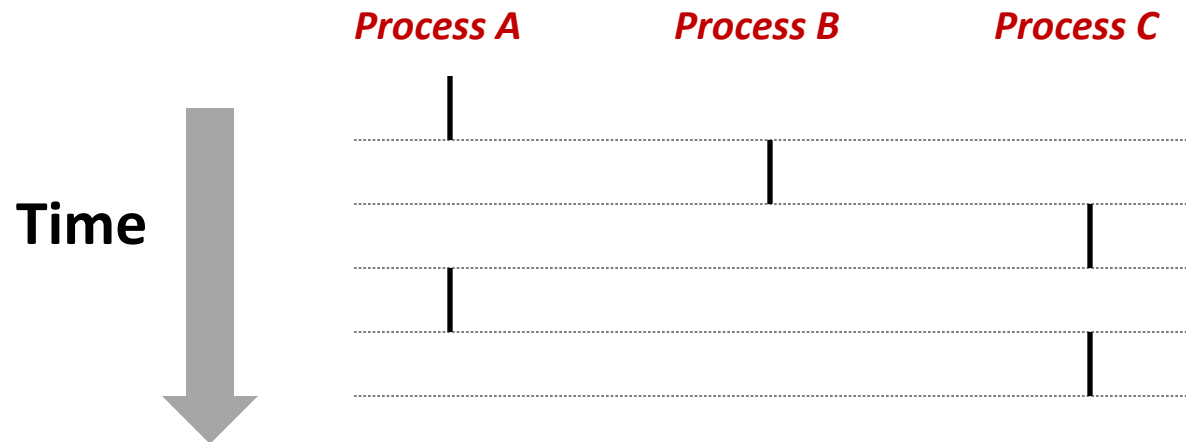
- Multicore processors
 - Multiple CPUs on single chip
 - Share main memory (and some of the caches)
 - Each can execute a separate process
 - Scheduling of processes onto cores done by kernel

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*

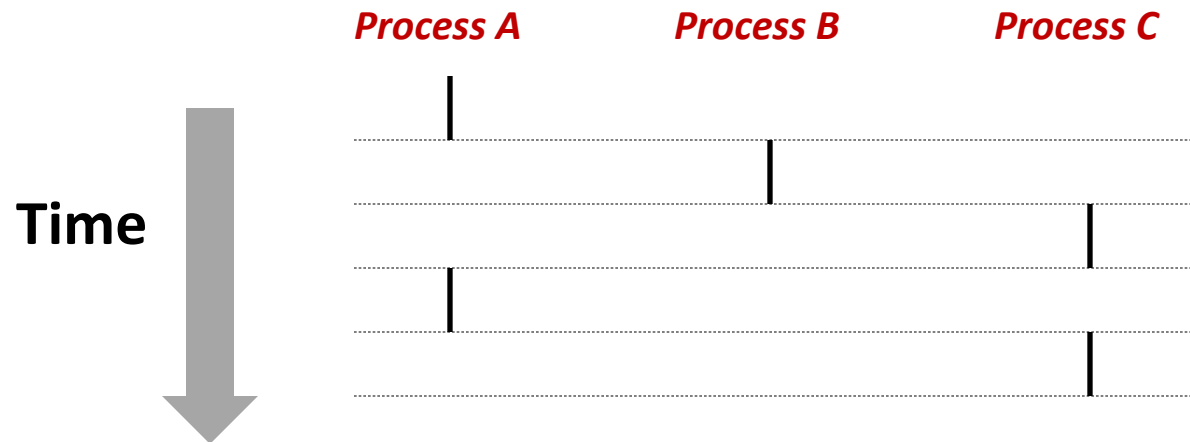
Concurrent Processes

- 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):



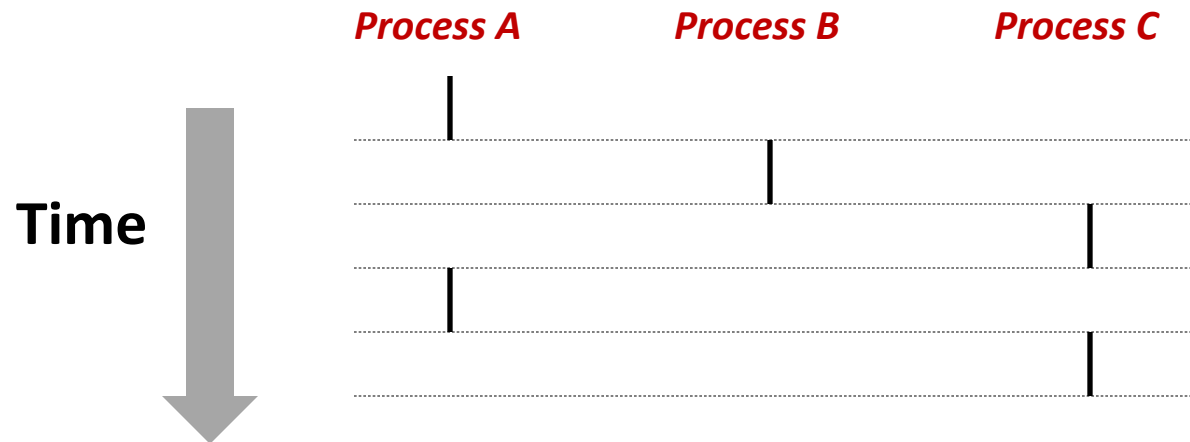
Concurrent Processes

- 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



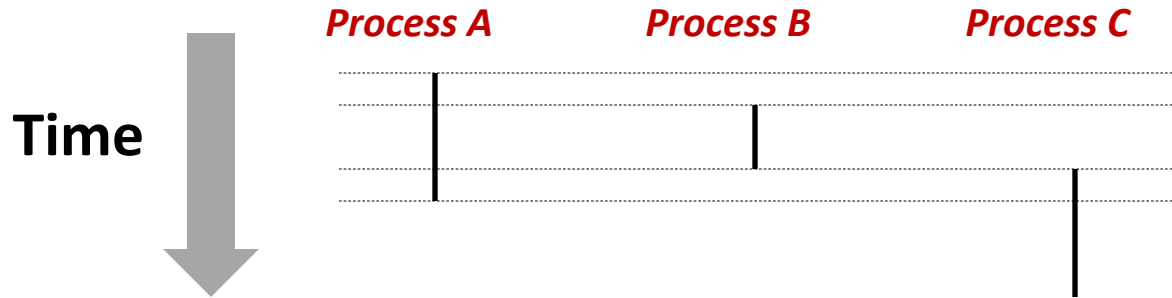
Concurrent Processes

- 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



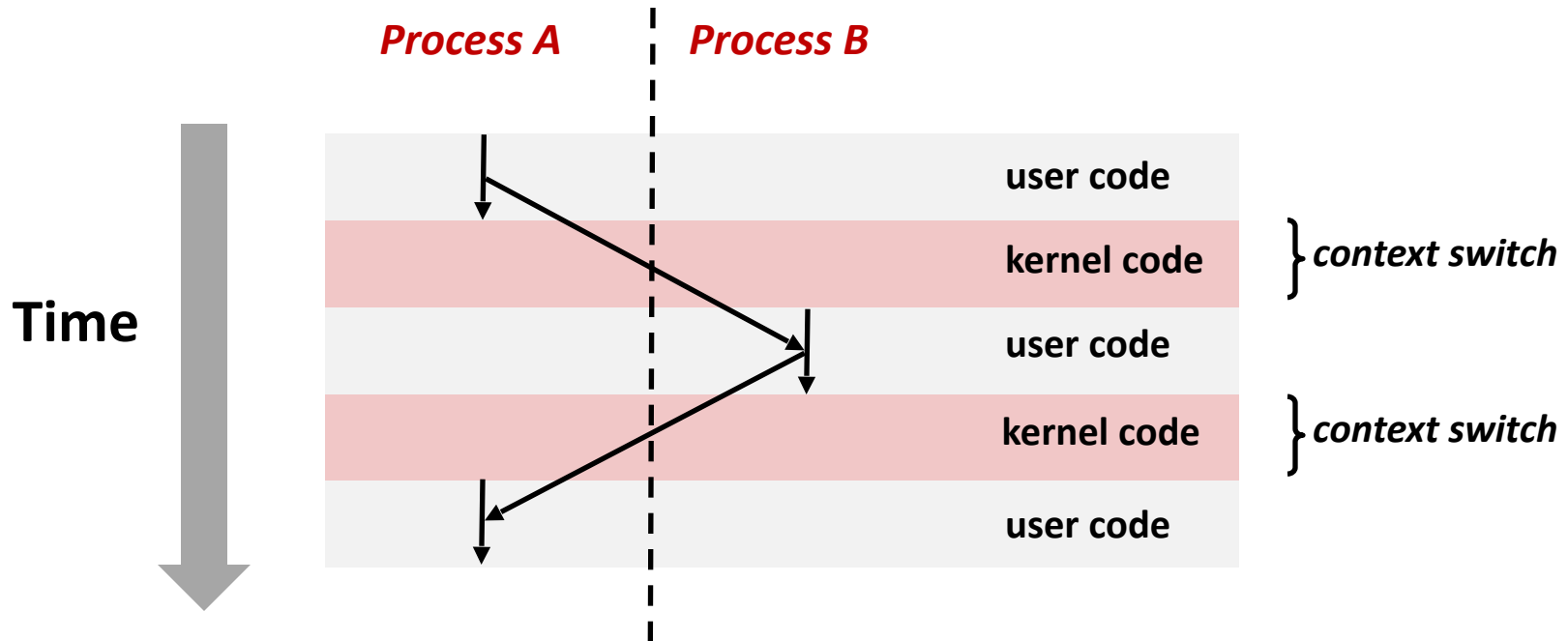
User View of Concurrent Processes

- 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

- Processes and Signals: running multiple programs concurrently
 - 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.

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
 - Returns 0 to the child process, child's PID to parent process
 - Child is almost identical to parent:
 - Child get an identical (but separate) copy of the parent's (virtual) address space (i.e., same stack copies, code, etc.)
 - Child gets identical copies of the parent's open file descriptors
 - Child has a different PID than the parent
- `fork` is interesting (and often confusing) because it is called **once** but returns **twice**

fork Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

```
linux> ./fork
parent: x=0
child : x=2
```

fork Example

- Call once, return twice

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

```
linux> ./fork
parent: x=0
child : x=2
```

fork Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child

```
linux> ./fork
parent: x=0
child : x=2
```

fork Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child
- Duplicate but separate address space
 - x has a value of 1 when fork returns in parent and child
 - Subsequent changes to x are independent

```
linux> ./fork
parent: x=0
child : x=2
```


fork Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

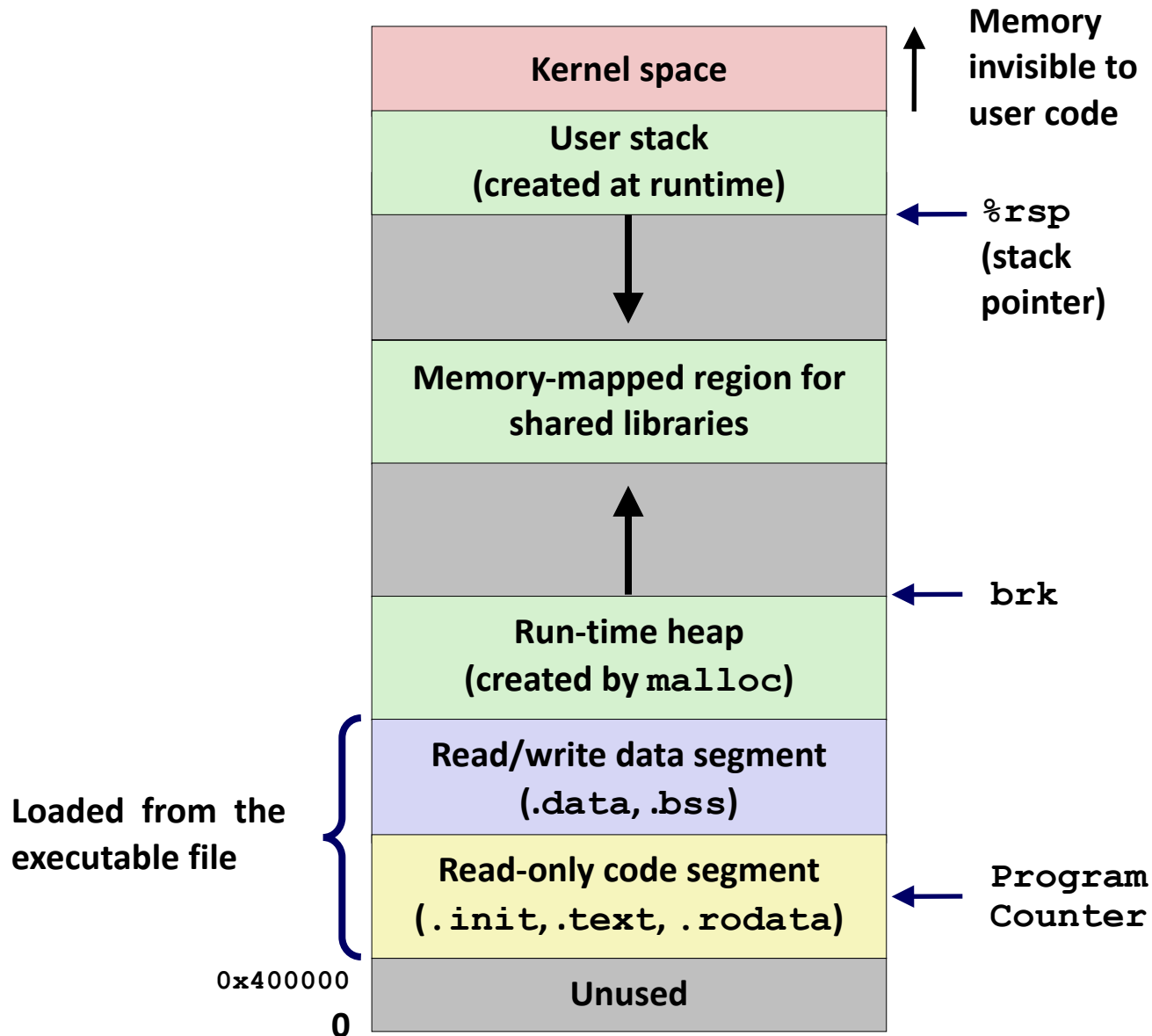
    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

```
linux> ./fork
parent: x=0
child : x=2
```

- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child
- Duplicate but separate address space
 - x has a value of 1 when fork returns in parent and child
 - Subsequent changes to x are independent
- Shared open files
 - stdout is the same in both parent and child

Process Address Space



Process Address Space

