Programming for Locality

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

The Memory Bottleneck

- The memory is a very common performance bottleneck that programmers sometimes don't think about
  - When you look at code, you often pay more attention to computation
- Example: a[i] = b[j] + c[k]
- I think "I am adding numbers", but in fact the access to the 3 arrays can take much more time than doing an addition
- For this line of code, the memory is the bottleneck!

Why the Memory Bottleneck?

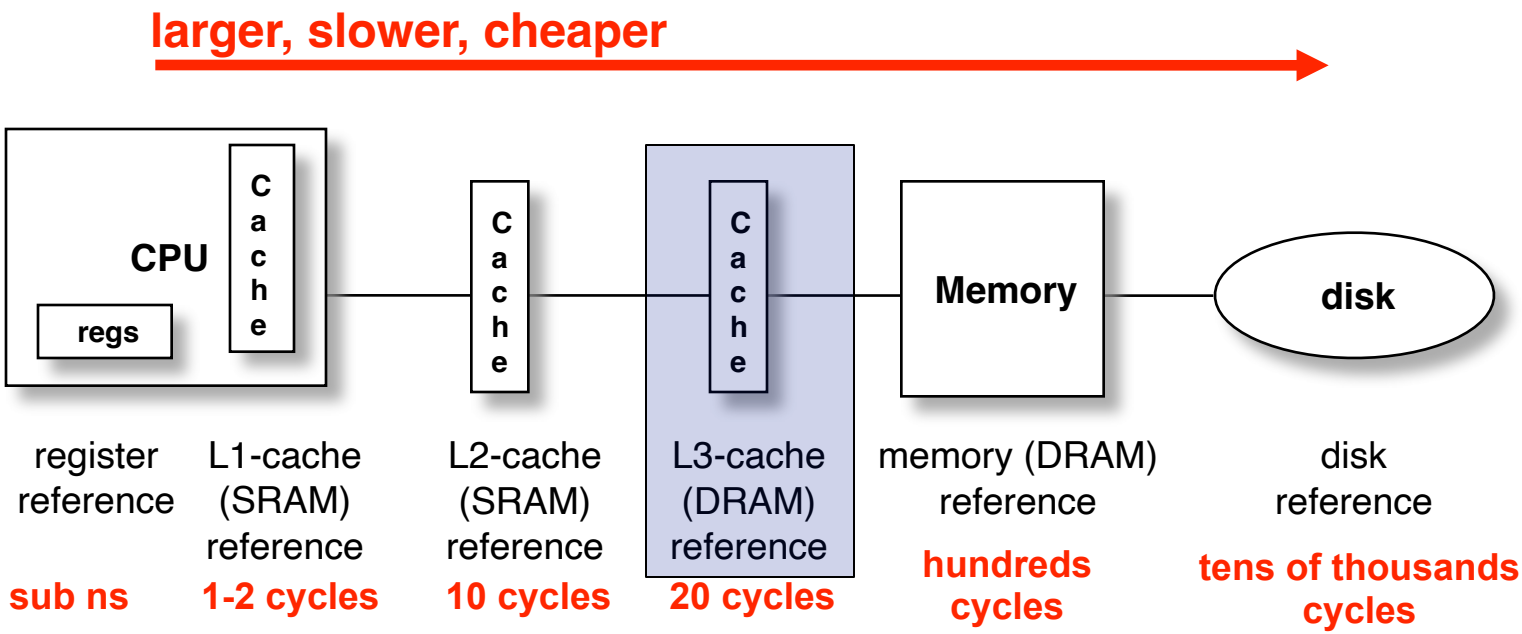
- In the 70's, everything was balanced
  - The memory kept pace with the CPU
    - n cycles to execute an instruction, n cycles to bring in a word from memory
- No longer true
  - CPUs have gotten 1,000x faster
  - Memories have gotten 10x faster and 1,000,000x larger

➔ Flops are free and bandwidth is expensive and processors are STARVED for data

- And we keep adding more starved cores (but at least they're not getting any faster...)

Reducing the Memory Bottleneck

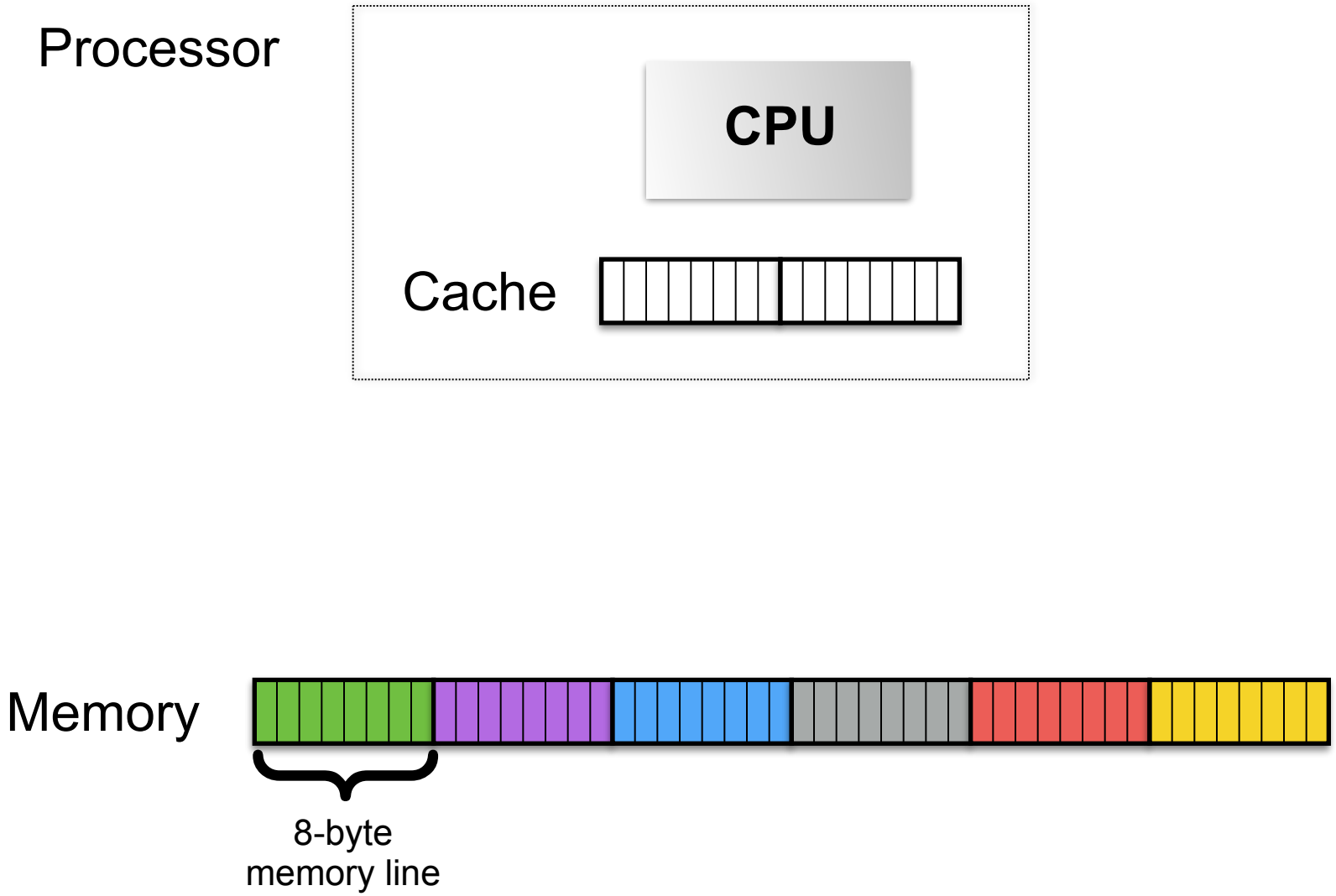
- The way in which computer architects have dealt with the memory bottleneck is via the memory hierarchy (see ICS 332)



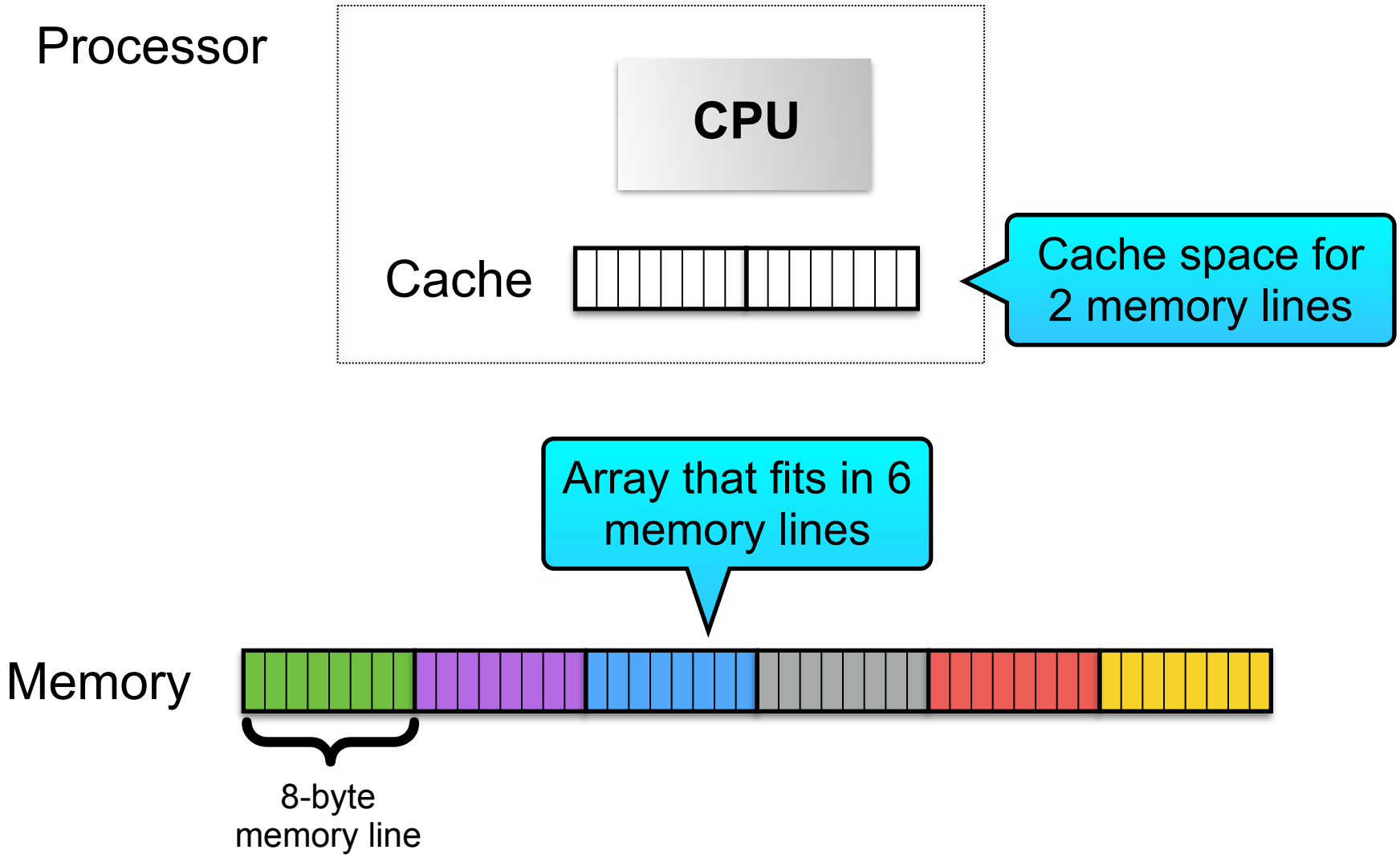
Misses and Hits

- Cache hit: the processor references an address, and the data at that address is in cache
  - The good case
  - You hope for most of your references to be hits
- Cache miss: the processor references an address, and the data at that address is not in cache
  - The bad case, which takes much more time
  - A **memory** line is brought into the cache
    - The bytes you need and some bytes around it
  - So that next time, all those bytes will be in cache
- Let's see this on a picture…

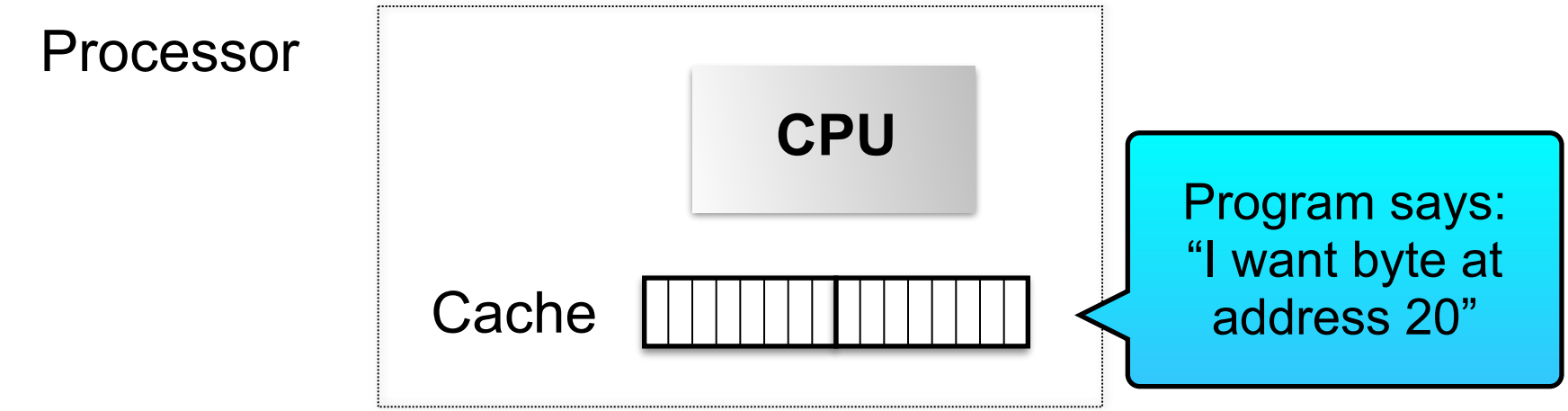
Cache/Memory Lines

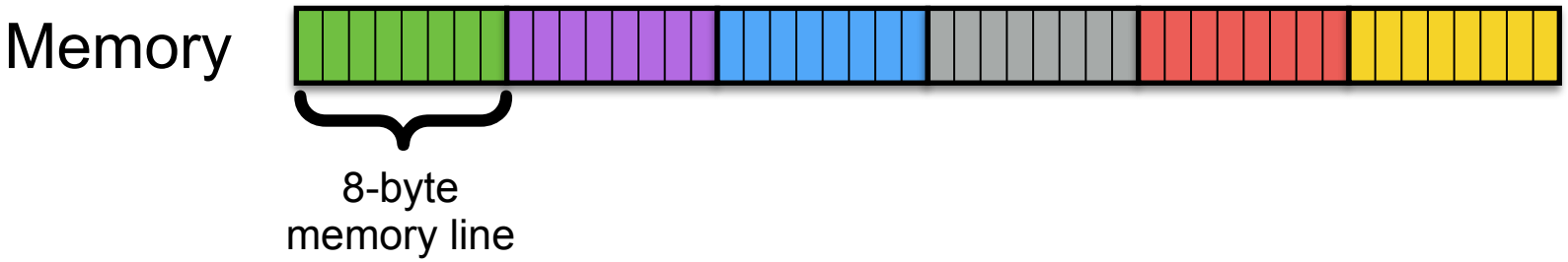


Cache/Memory Lines

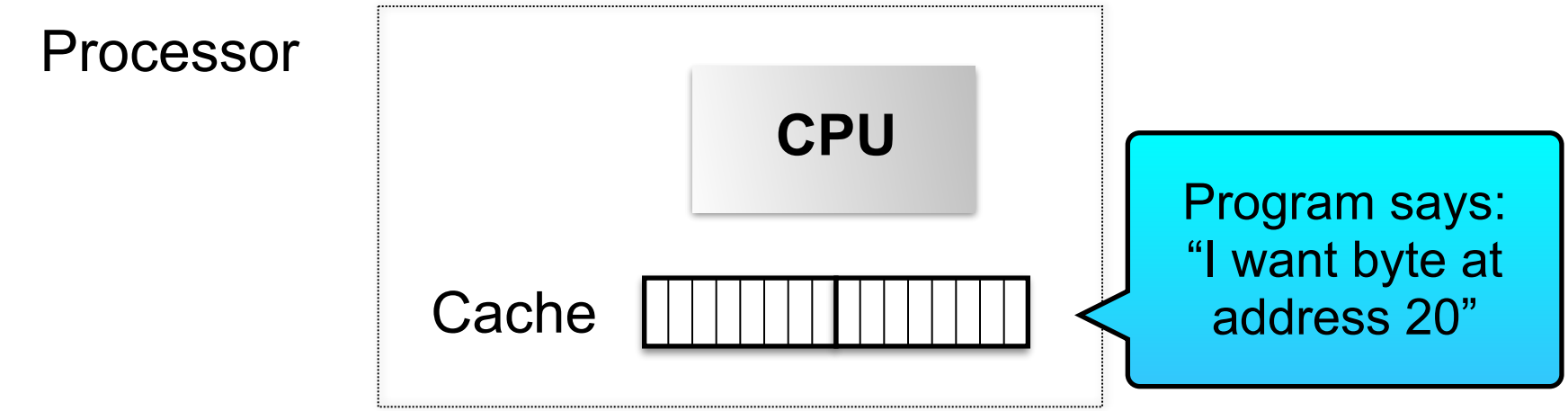


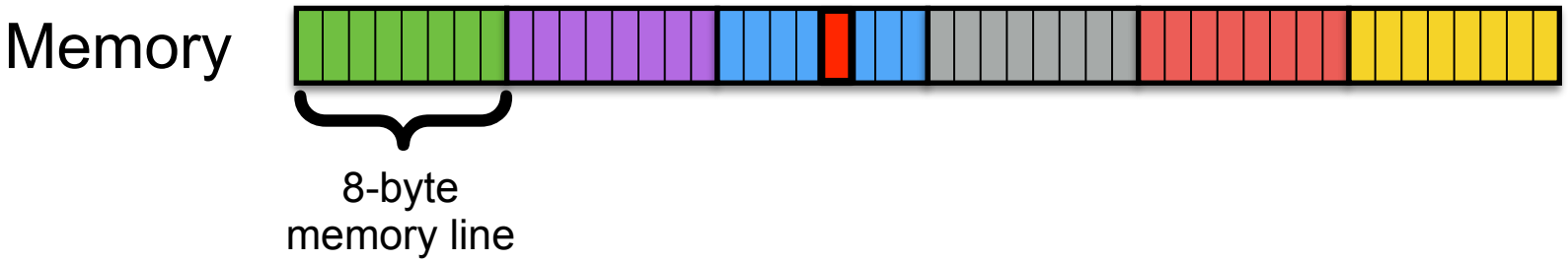
Cache/Memory Lines



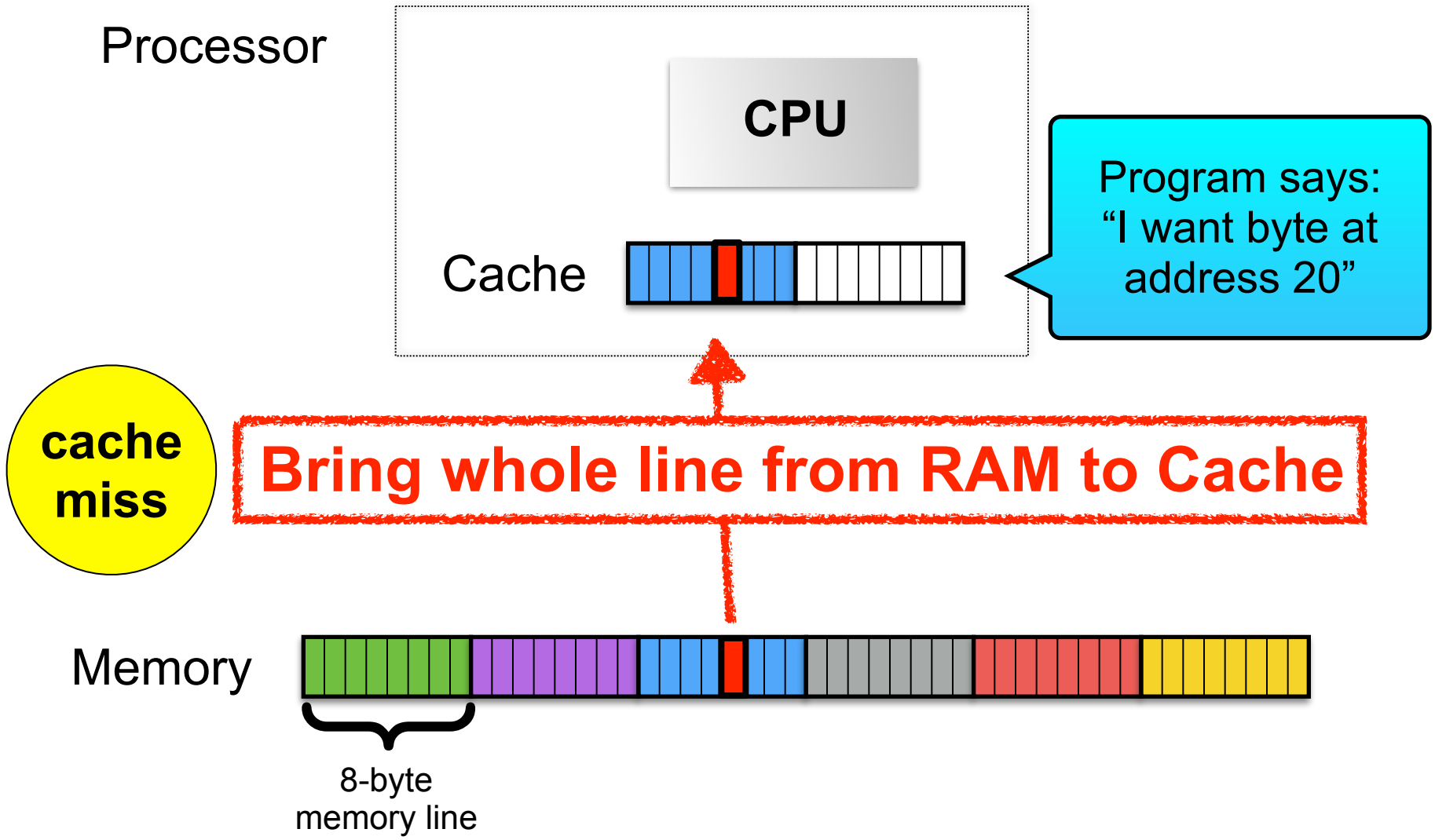


Cache/Memory Lines

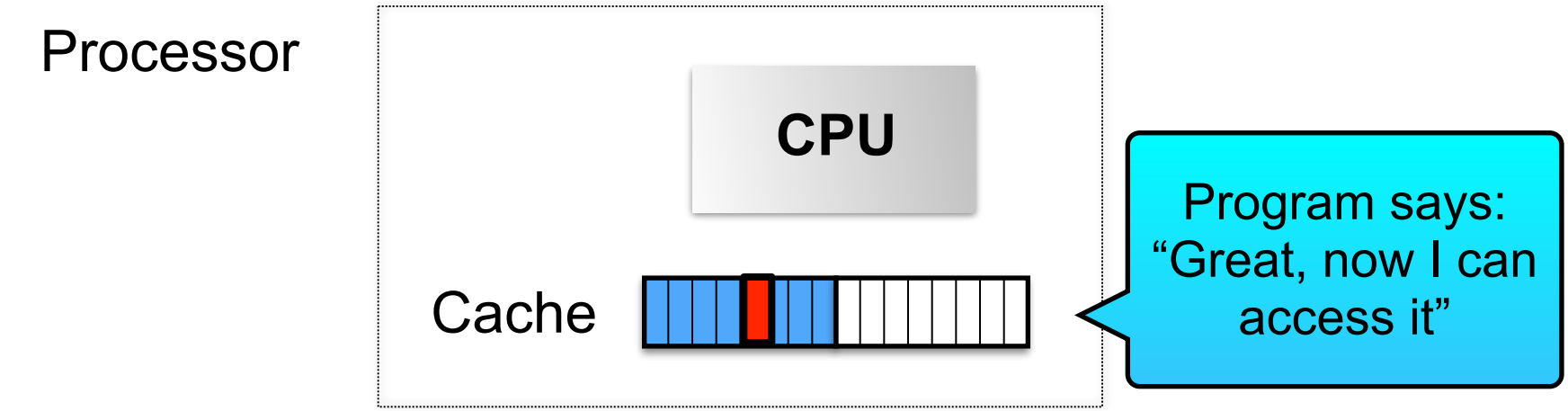


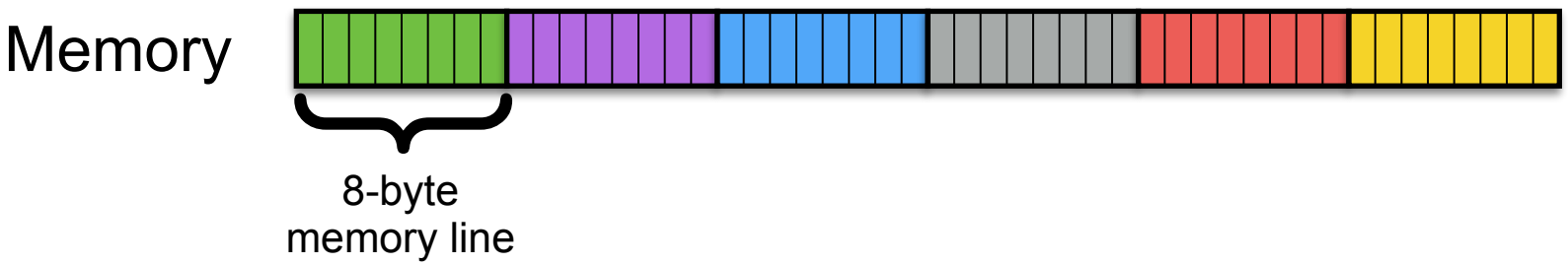


Cache/Memory Lines

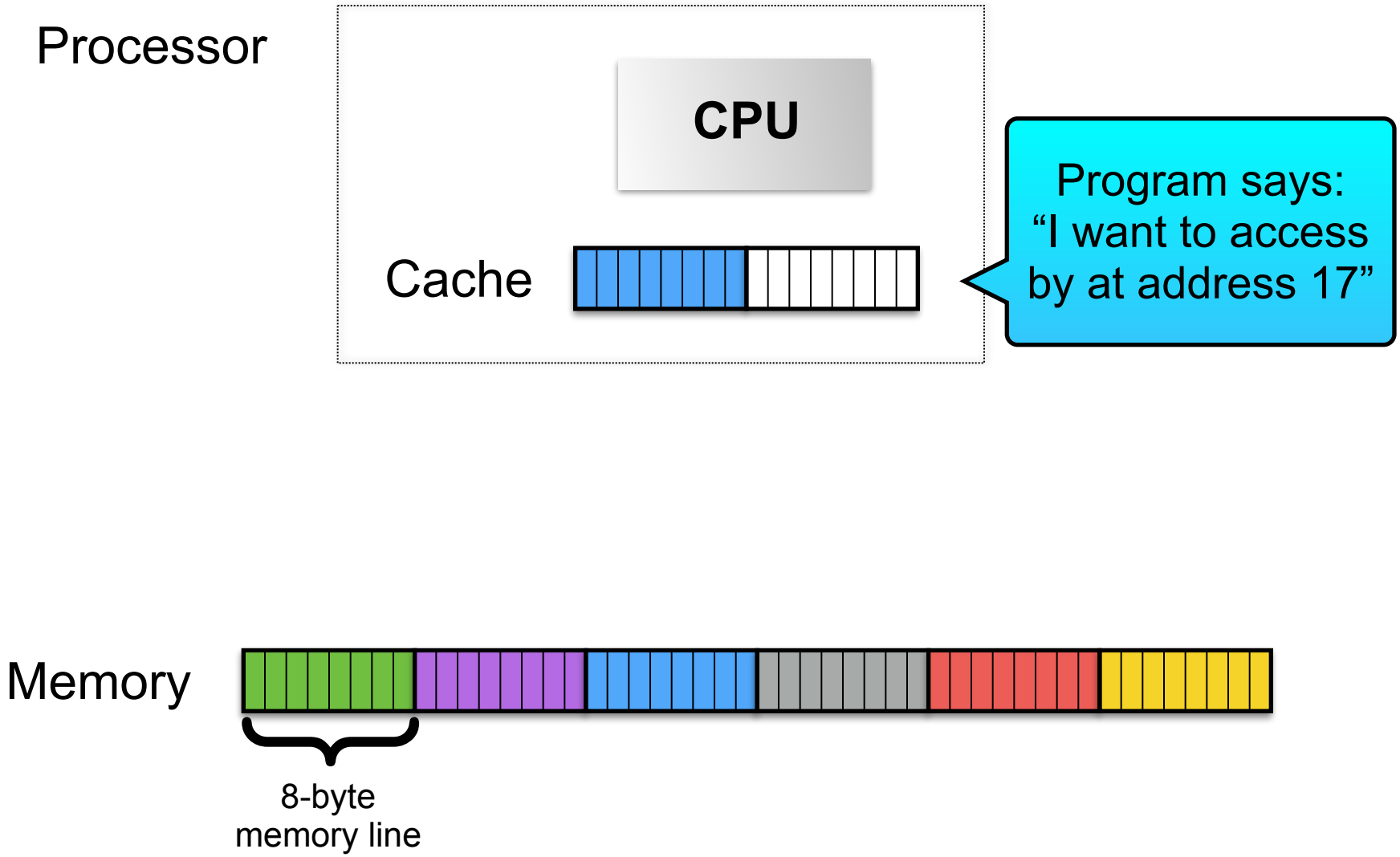


Cache/Memory Lines

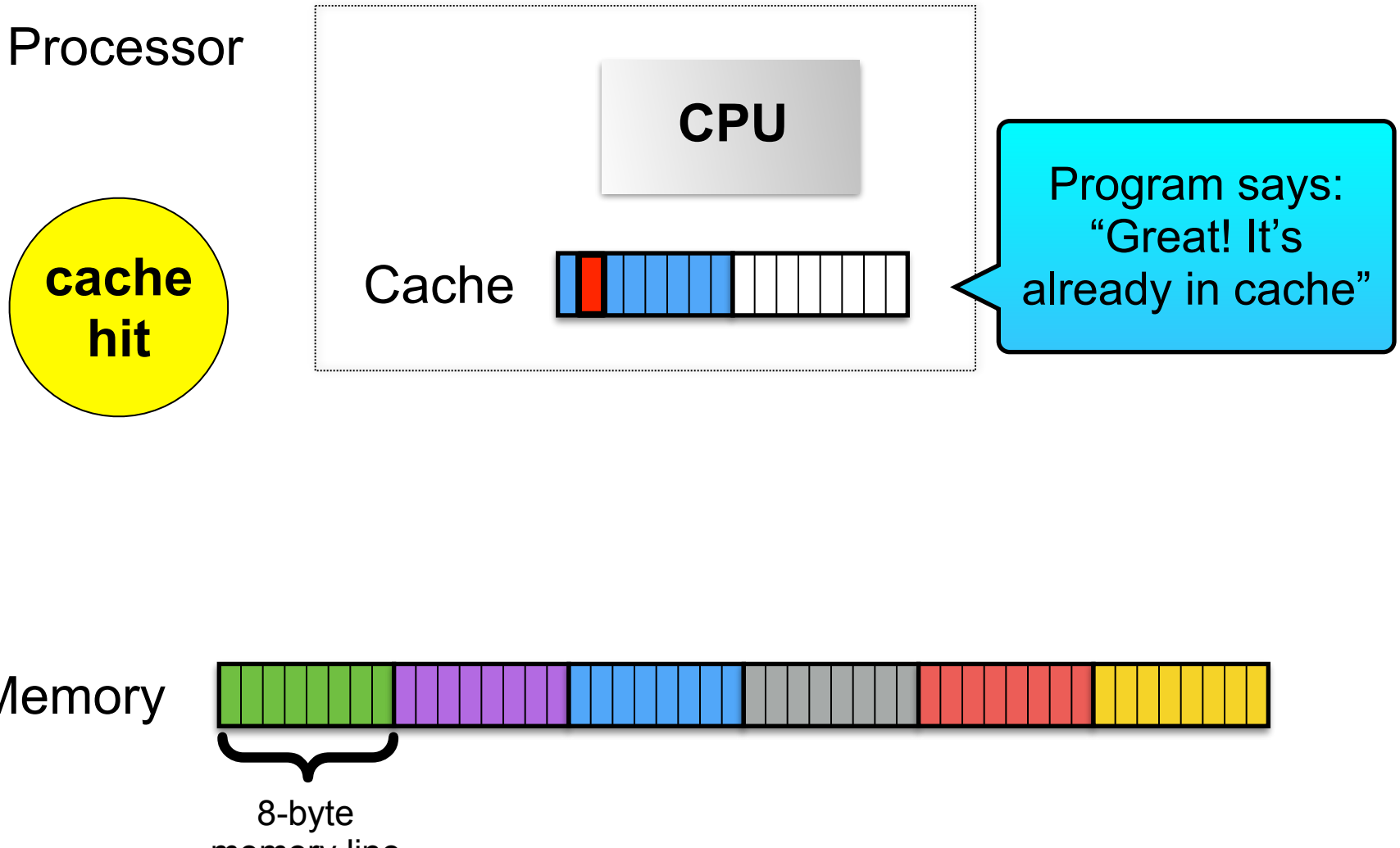




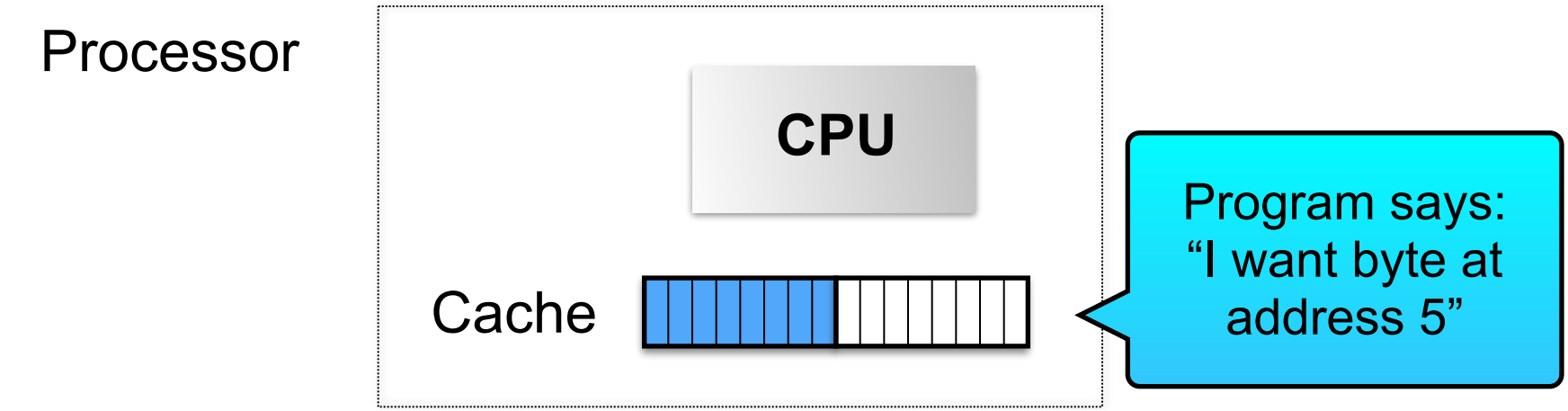
Cache/Memory Lines

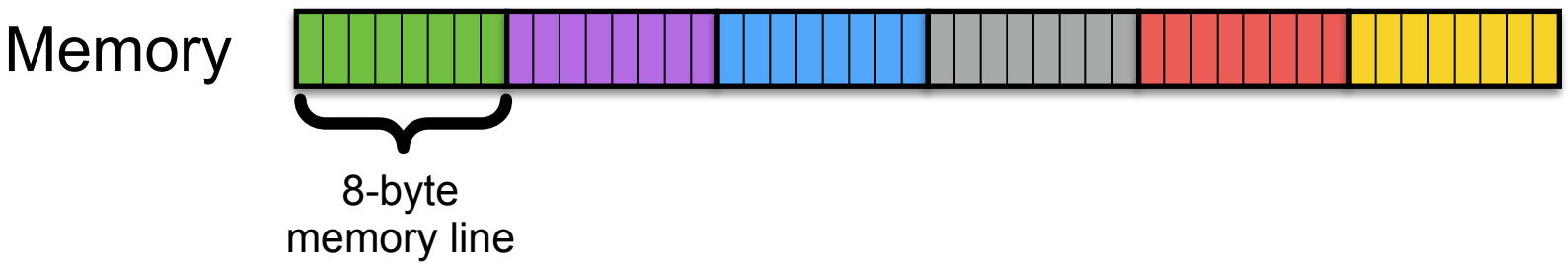


Cache/Memory Lines

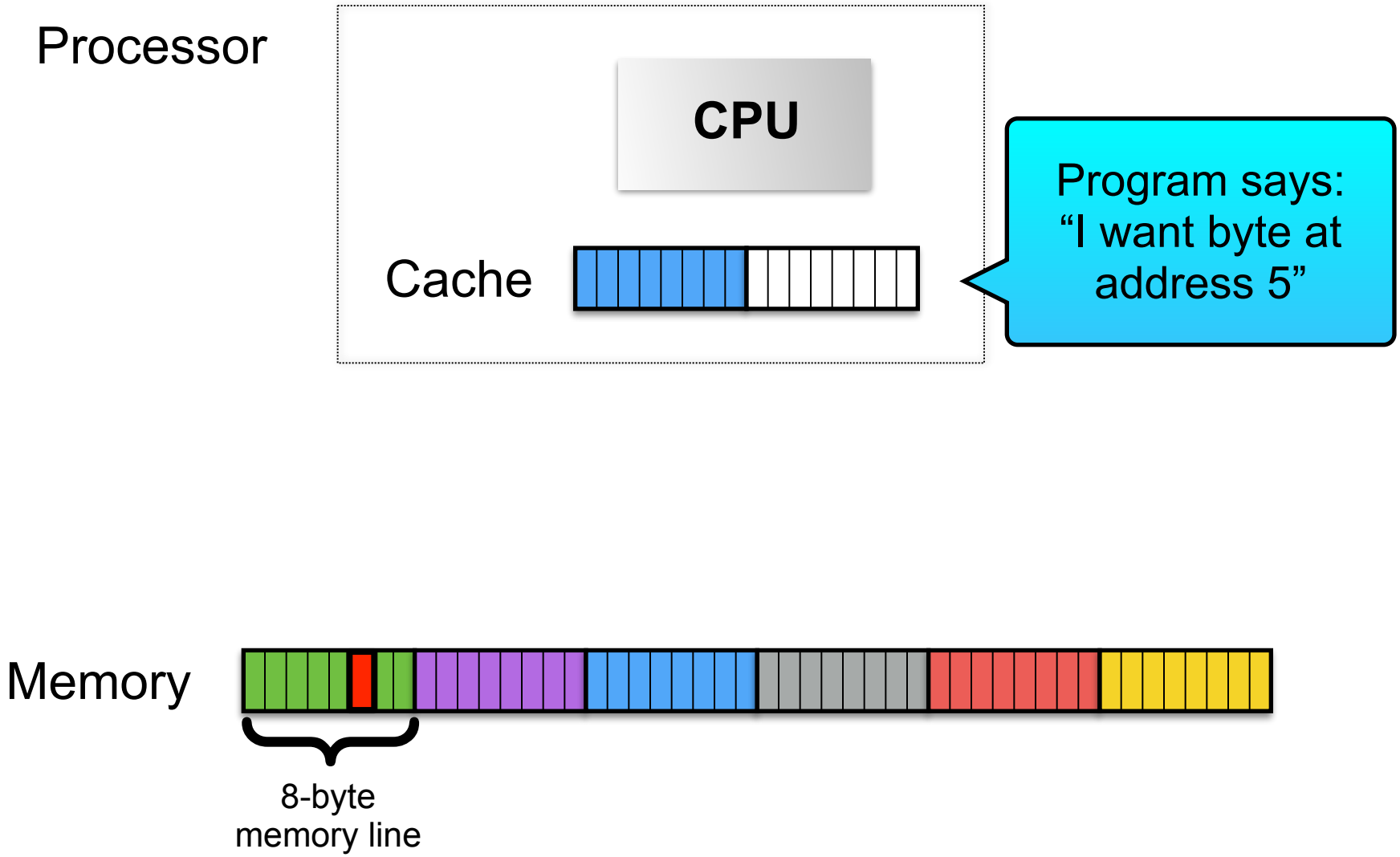


Cache/Memory Lines

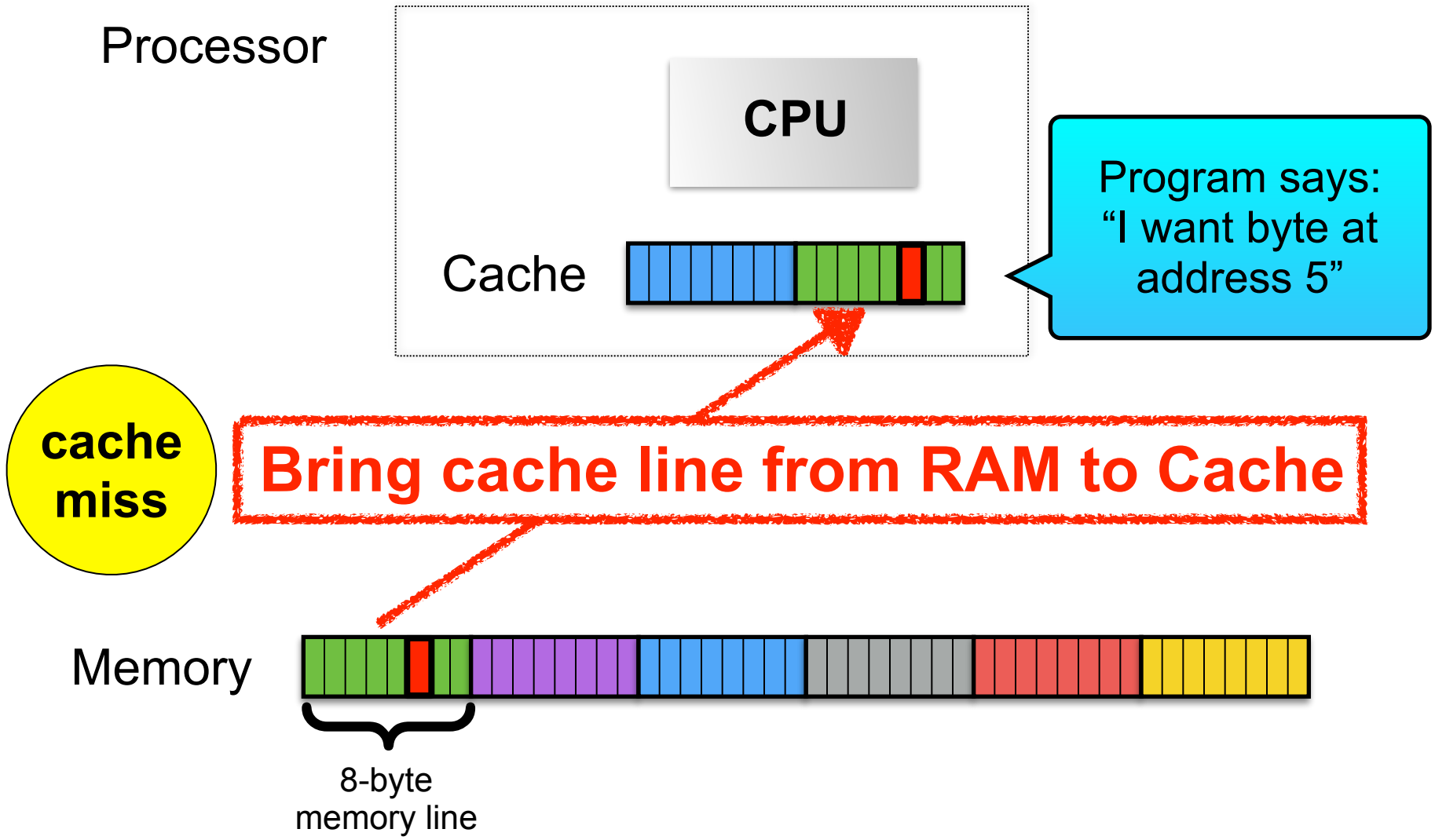




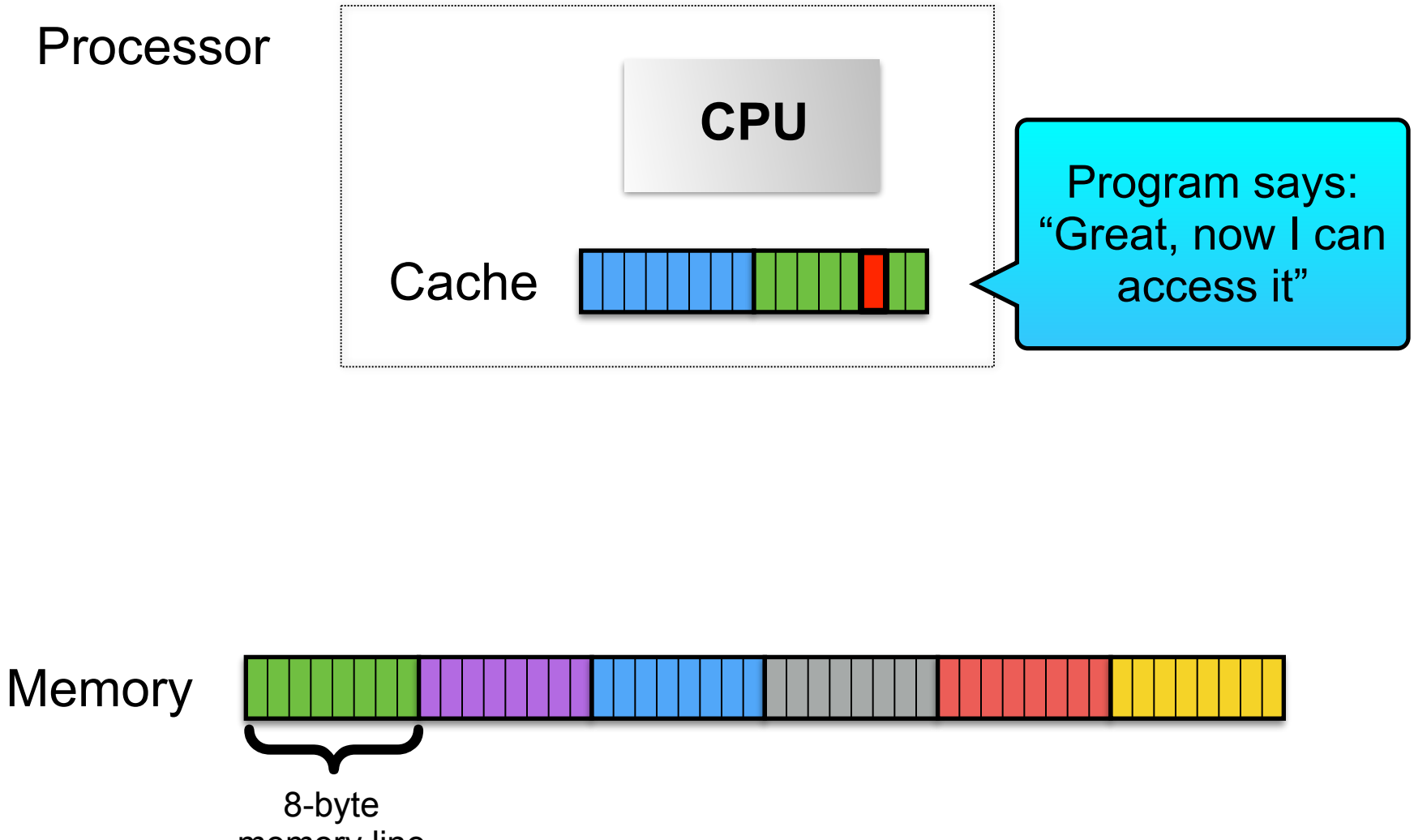
Cache/Memory Lines



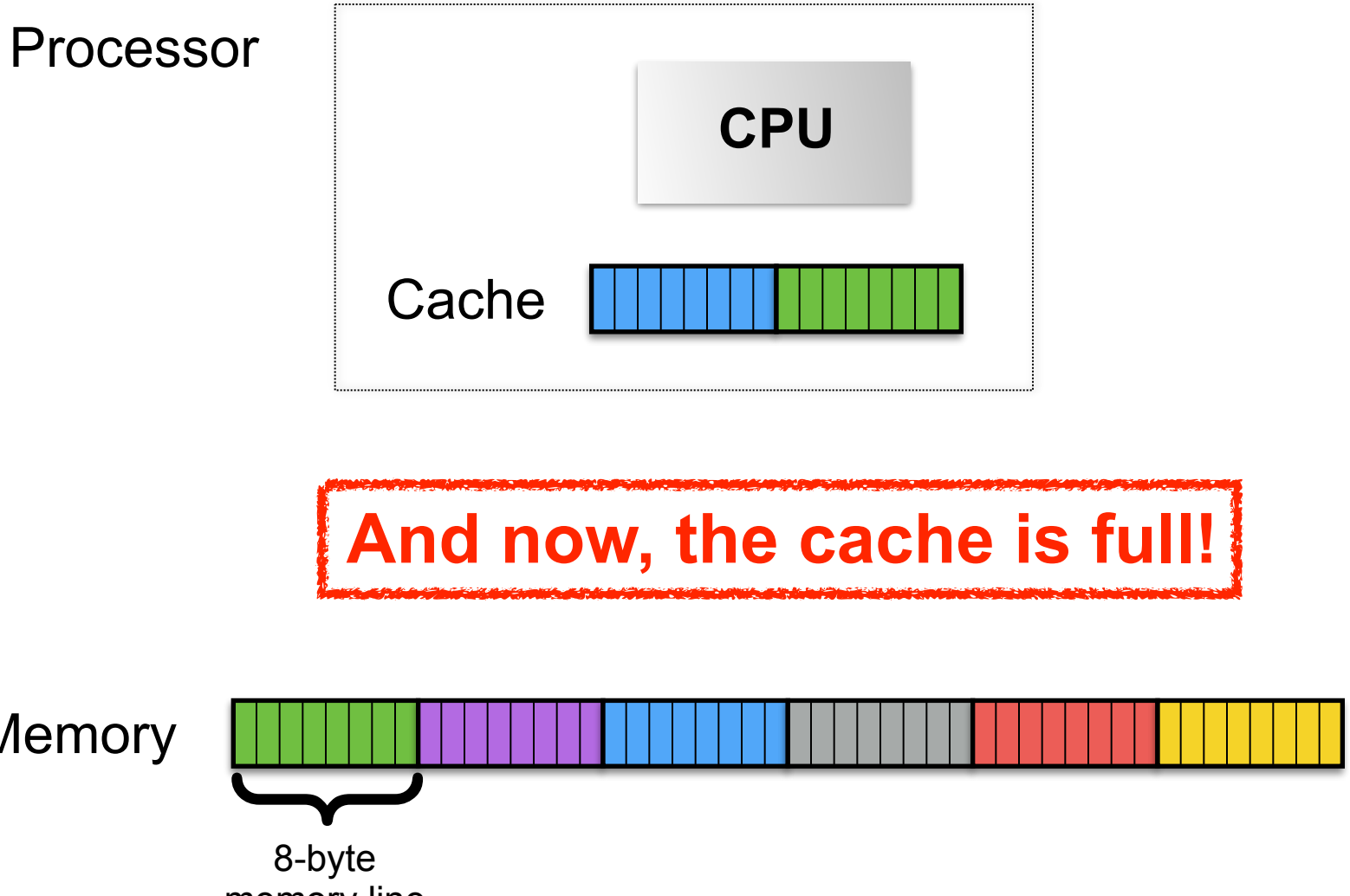
Cache/Memory Lines



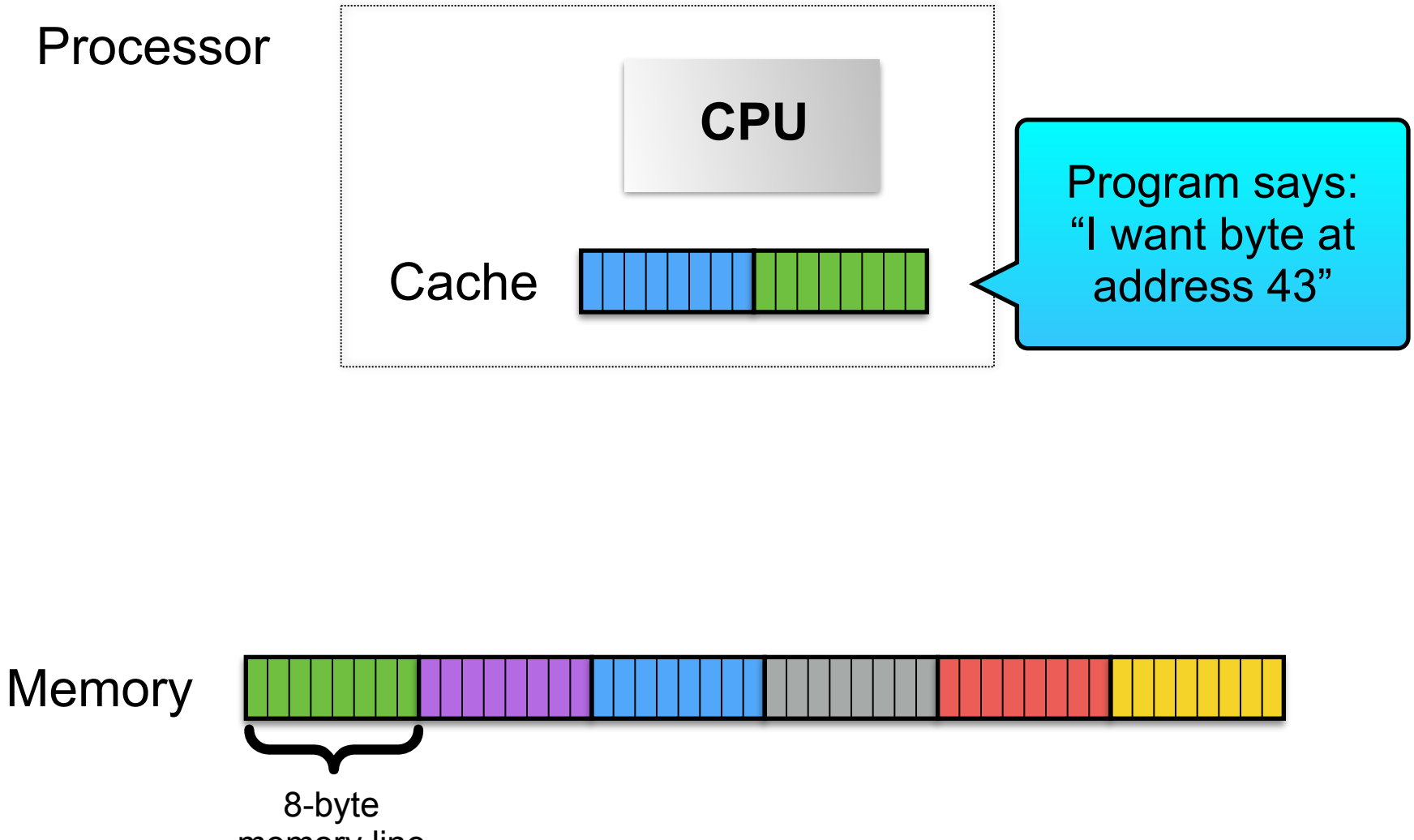
Cache/Memory Lines



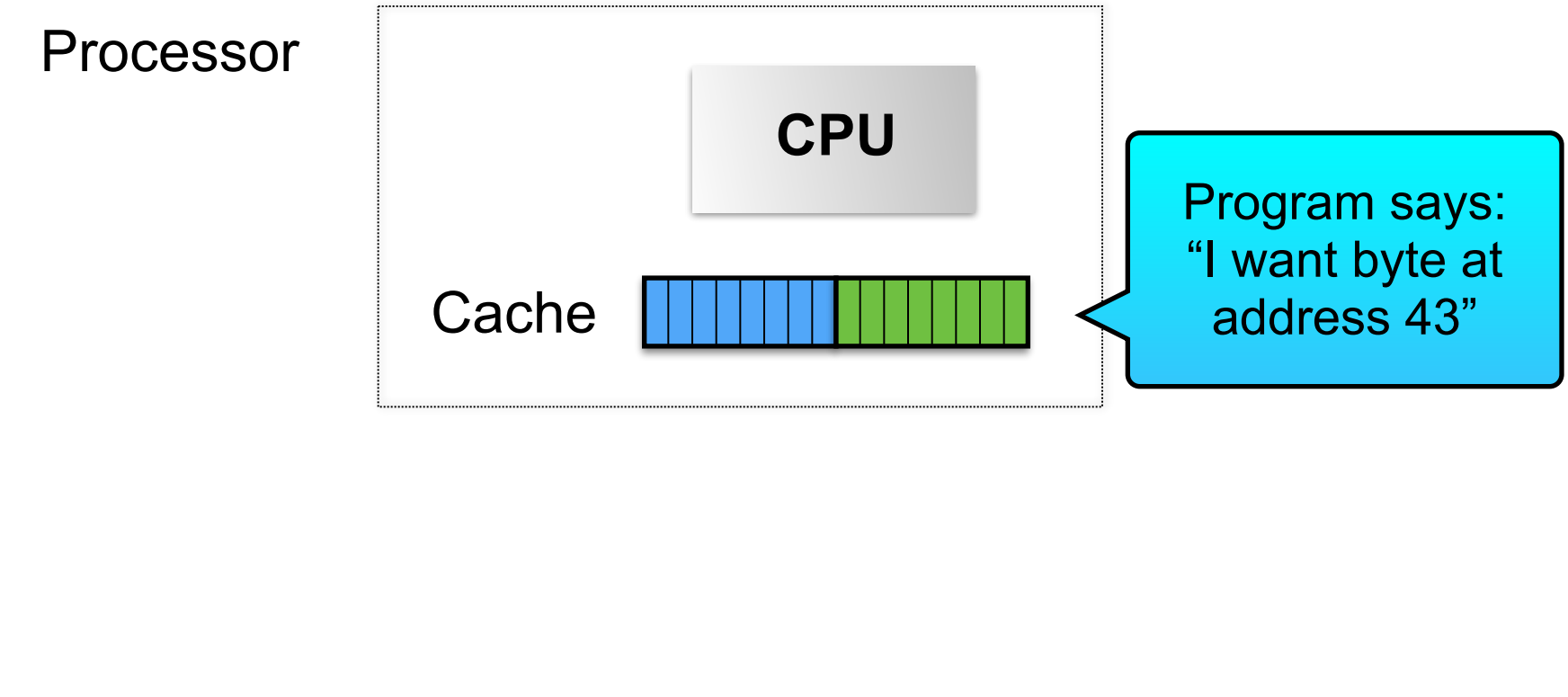
Cache/Memory Lines



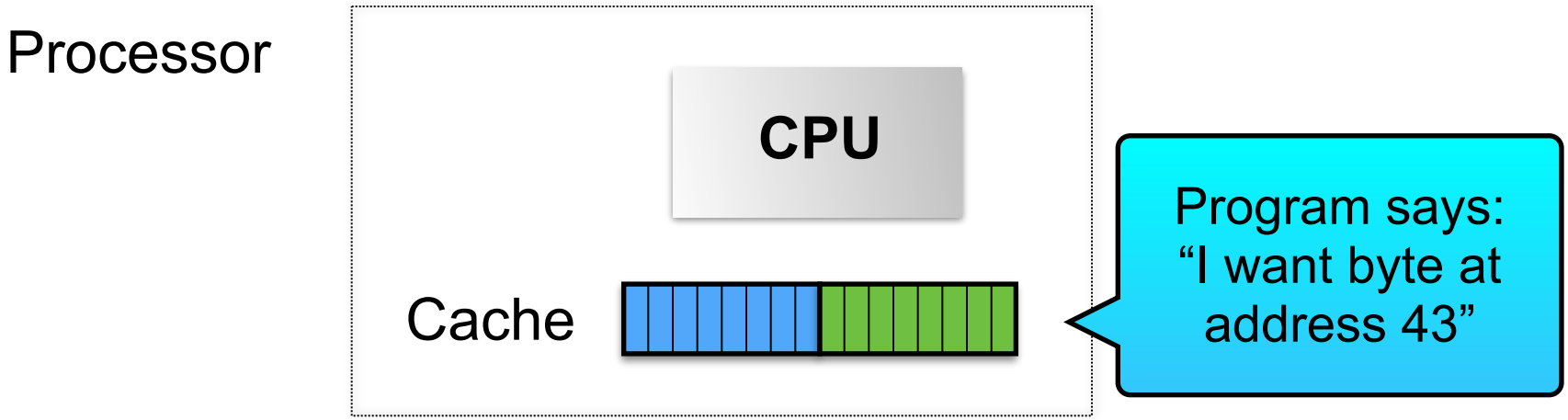
Cache/Memory Lines



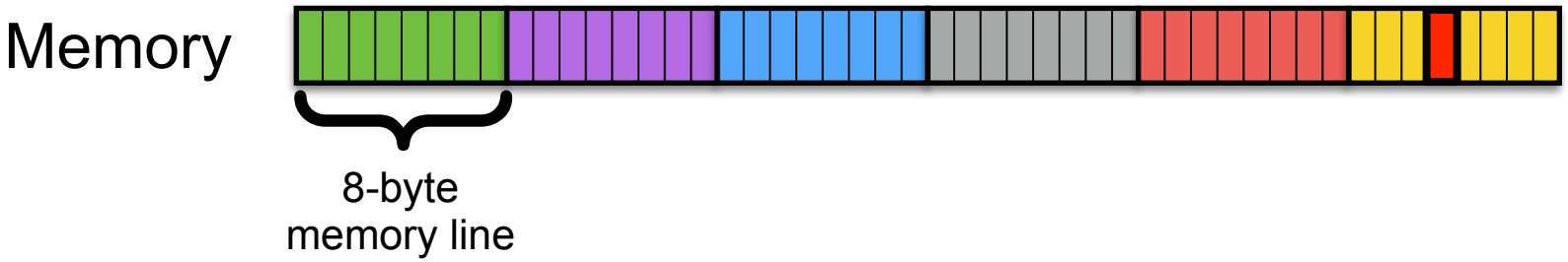
Cache/Memory Lines



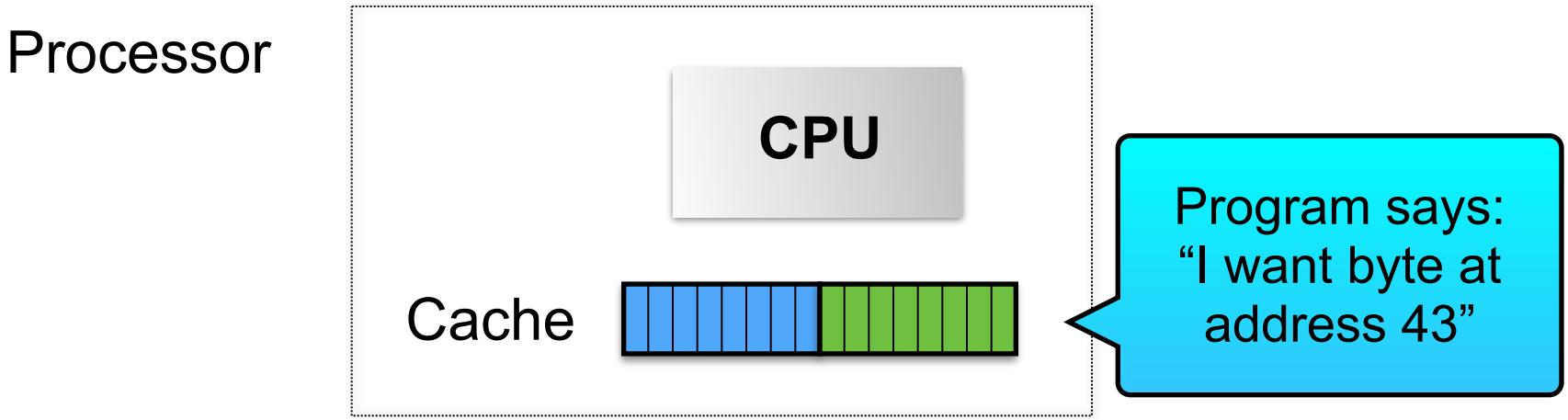
Cache/Memory Lines



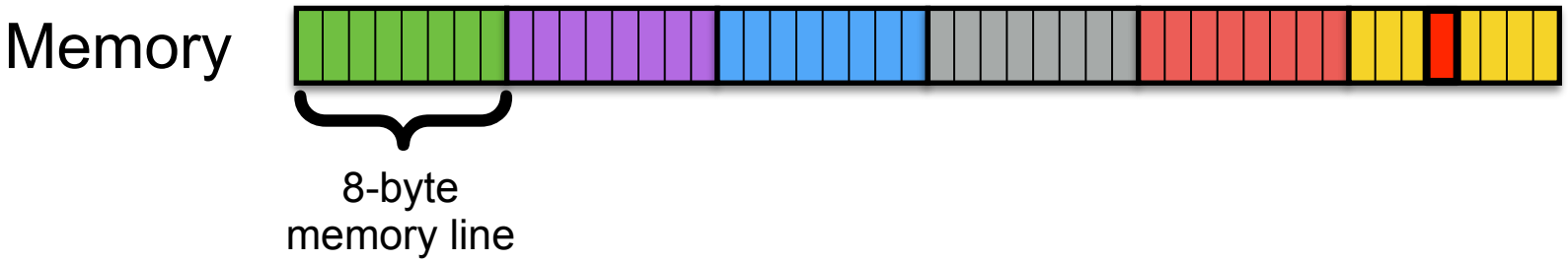
We need to “evict” a memory line from the cache to create space (say the blue one)



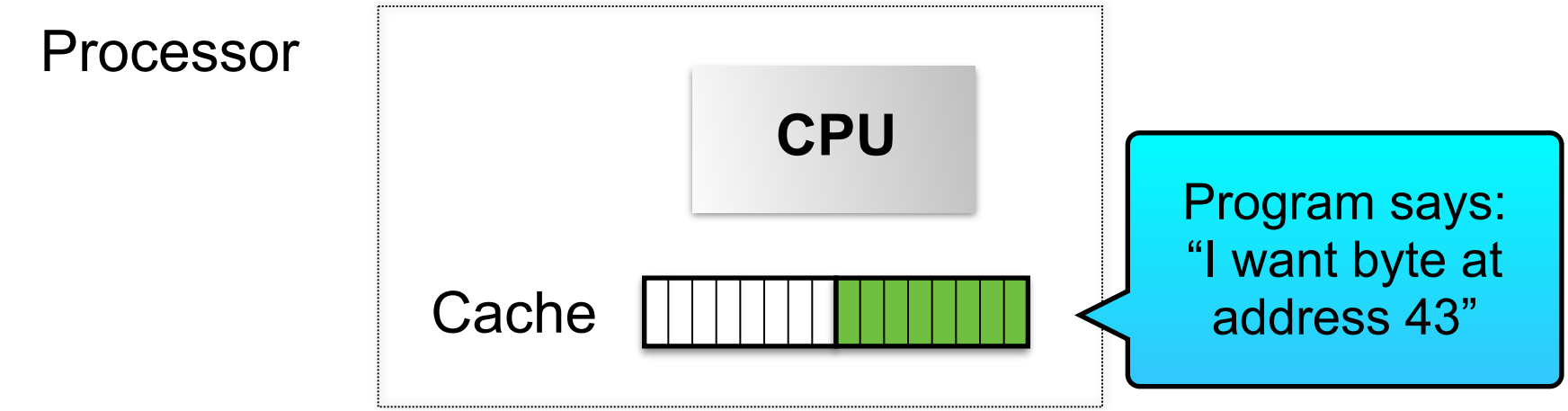
Cache/Memory Lines

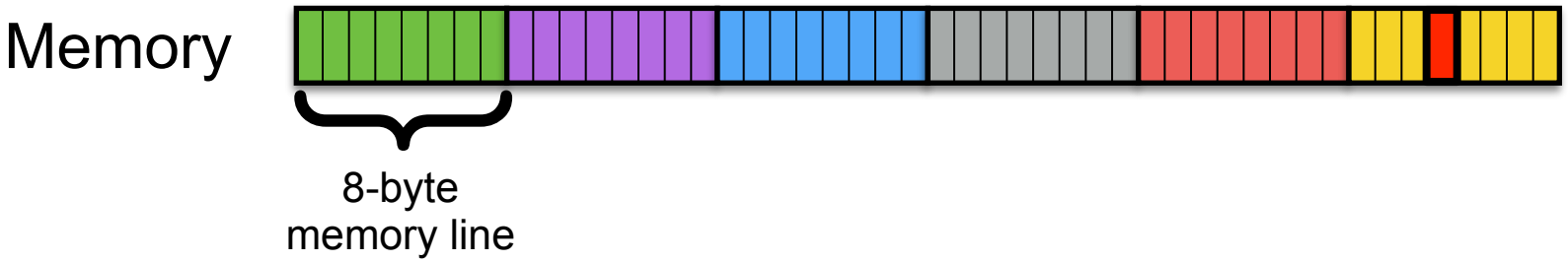


Let’s say we evict the Least Recently Used (LRU) line from the cache (blue one)

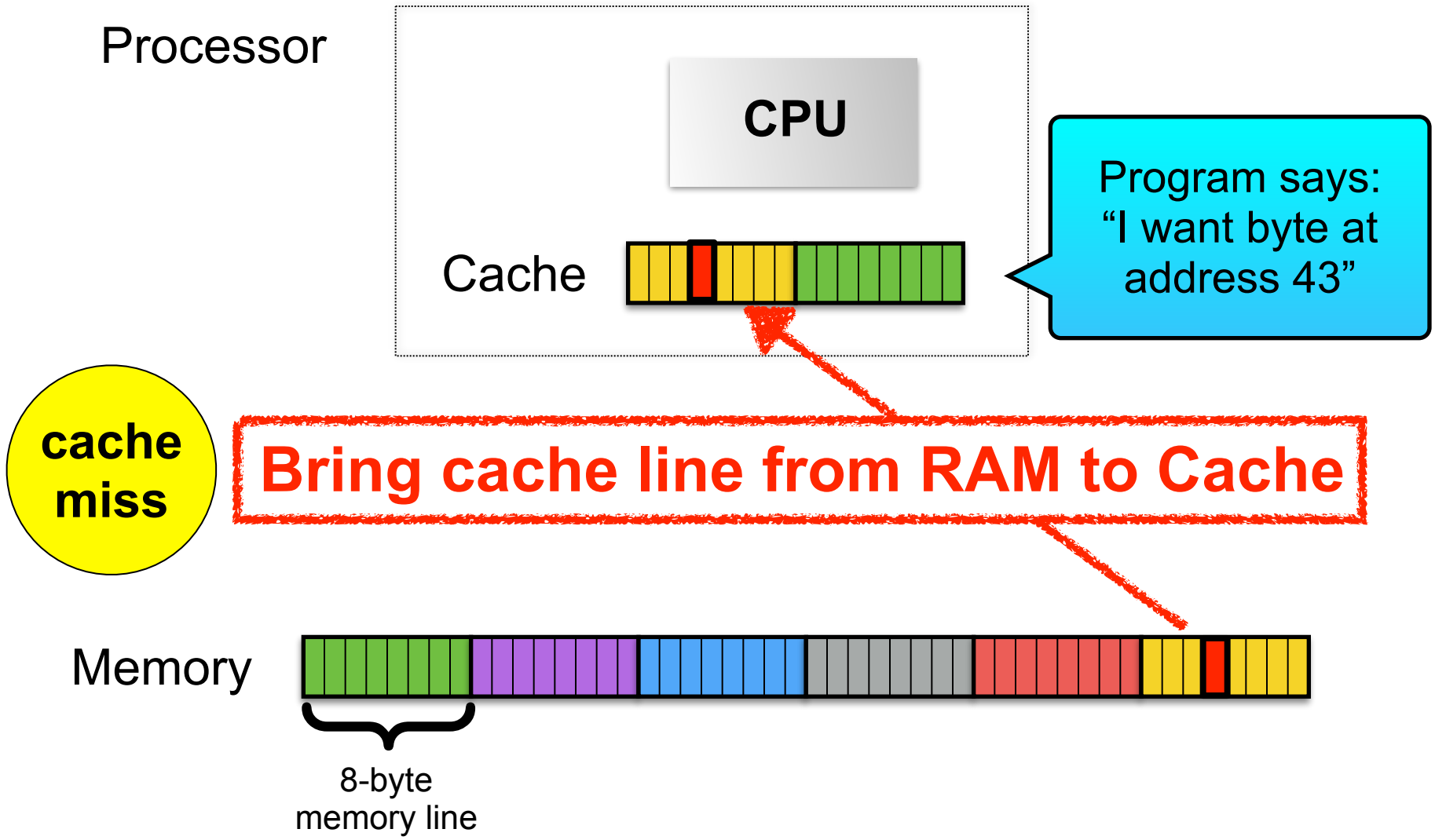


Cache/Memory Lines

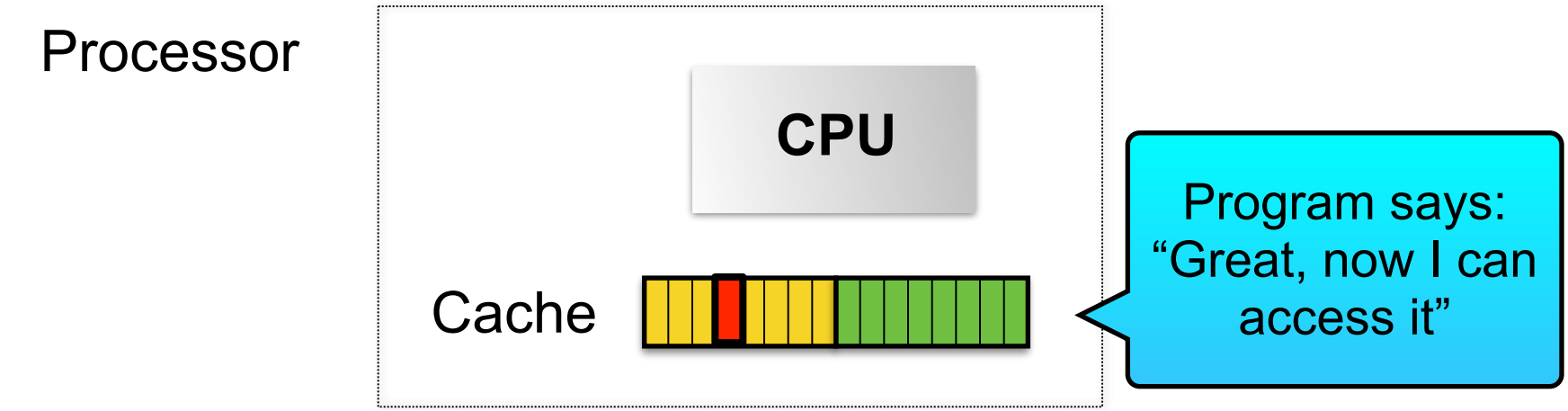


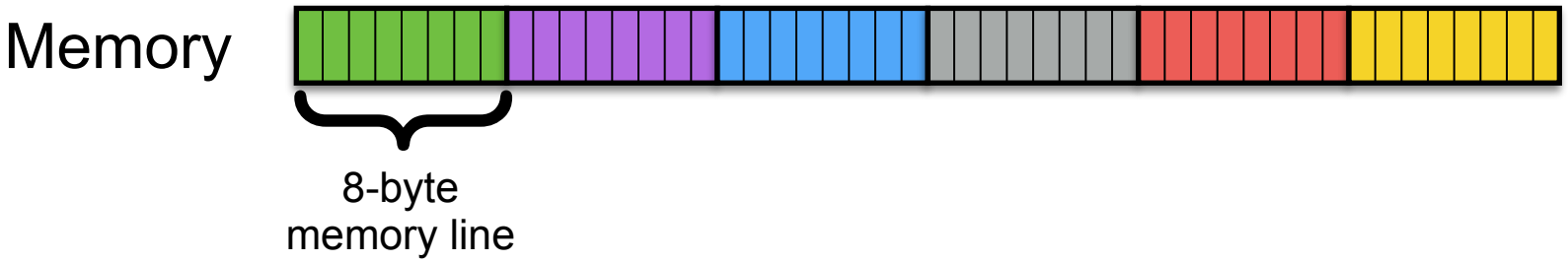


Cache/Memory Lines

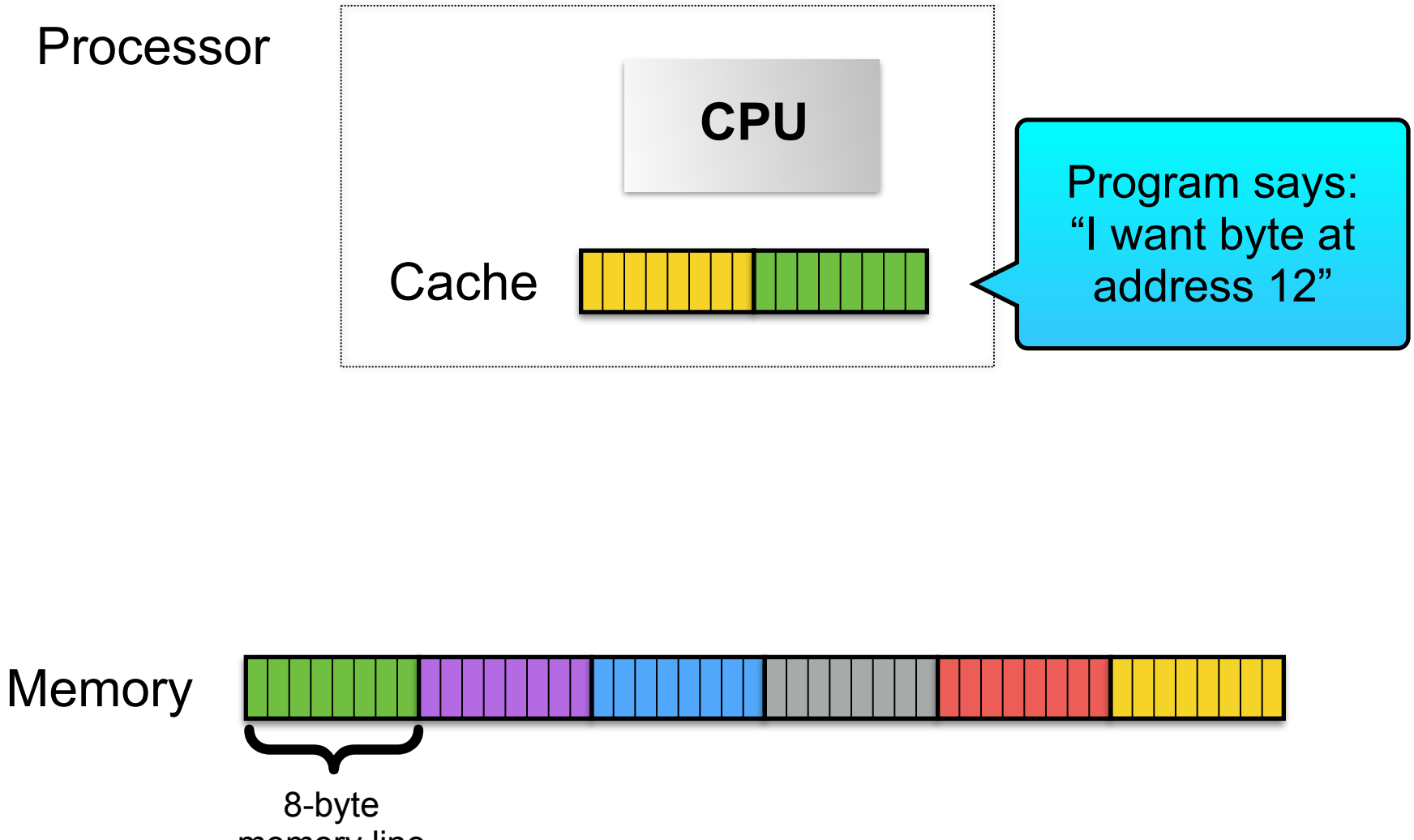


Cache/Memory Lines

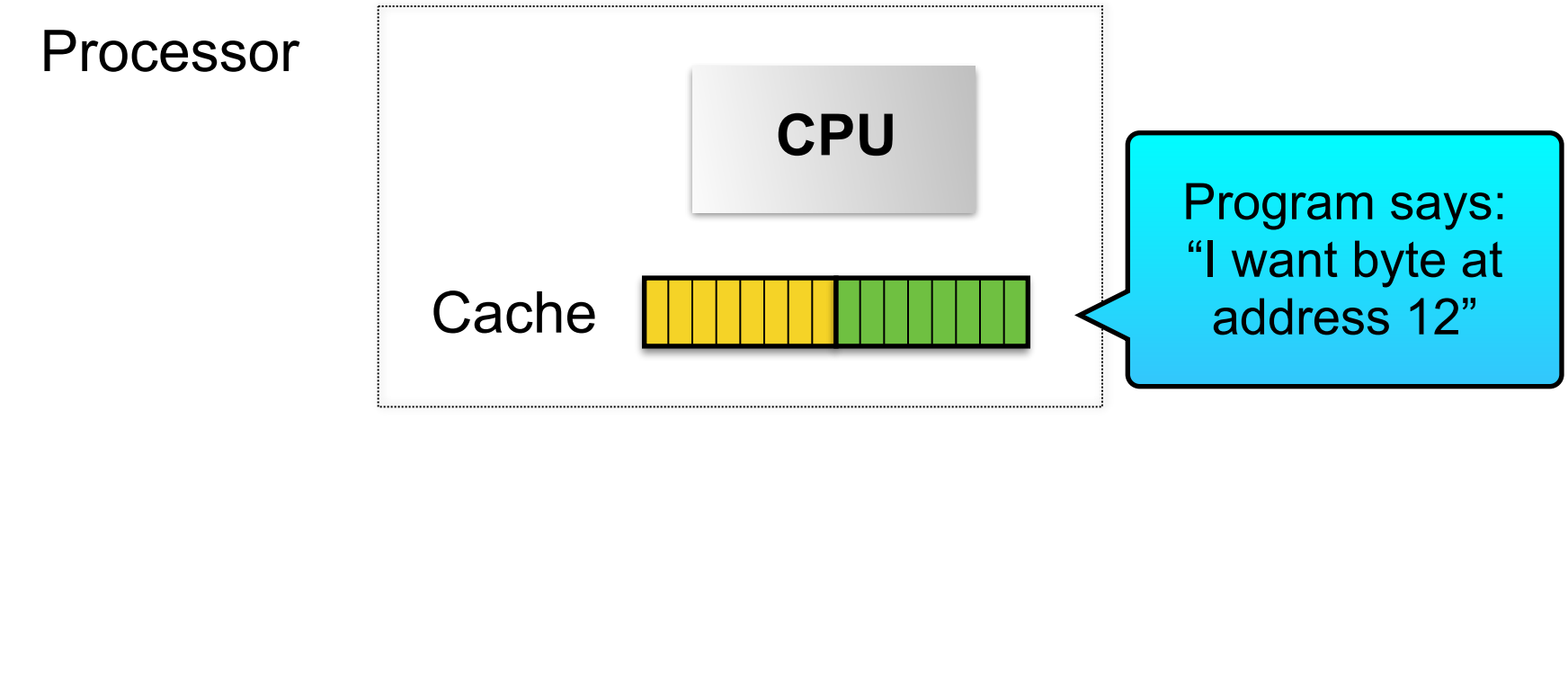




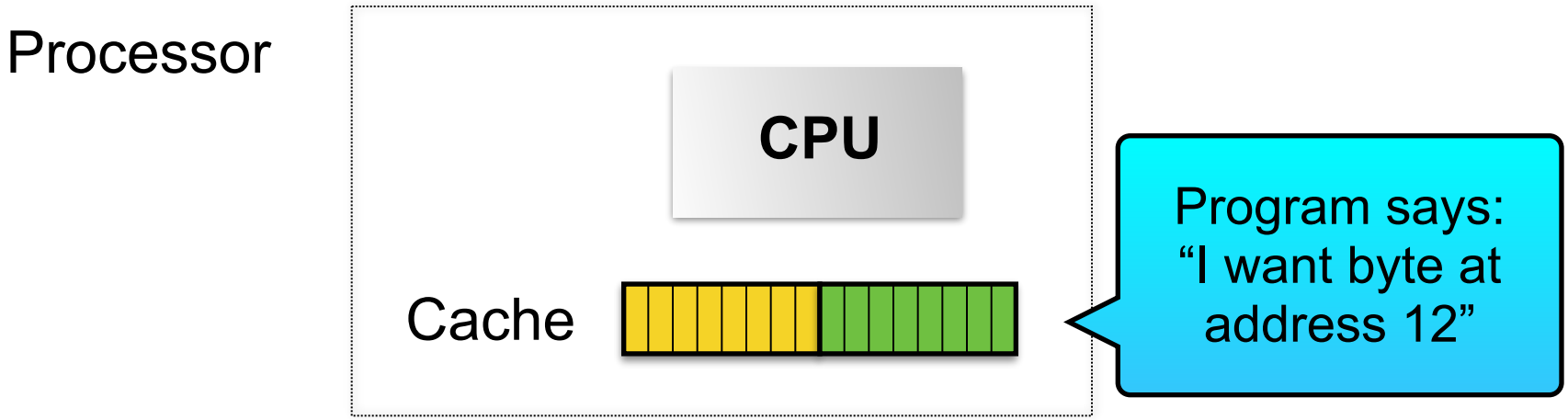
Cache/Memory Lines



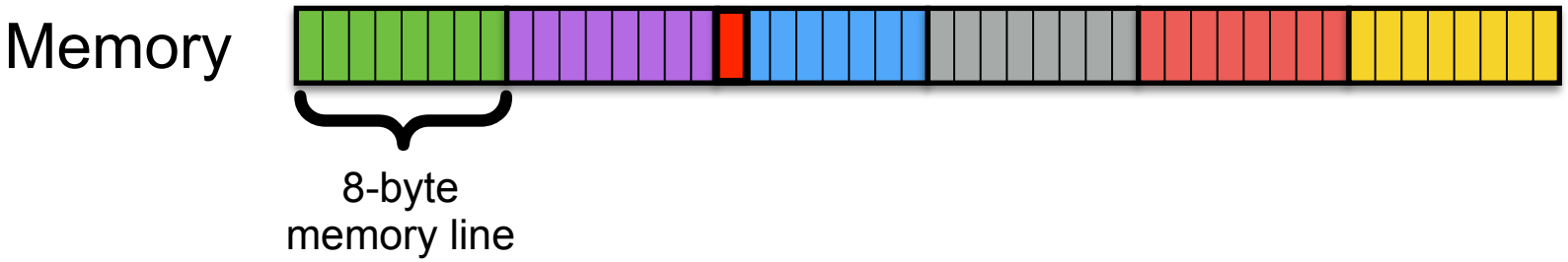
Cache/Memory Lines



Cache/Memory Lines



We had the blue line in cache, but evicted it, so now we’ll incur another cache miss…



All this Happens in Hardware

- All cache management is done in hardware
- The OS and the programmer doesn't have to do anything special, and in fact can't influence how the cache works
- Real hardware is more complex than what we saw in our animation
  - Several levels of cache
  - What happens on a write? (update only the cache or both the catch and the memory?)
  - Which cache lines should be evicted?
  - What happens with multiple cores?
  - See a Computer Architecture course
- But regardless, why does it all work?

Locality in your Programs

- The memory hierarchy is useful because of "locality"
- **Temporal locality:** a memory location that was referenced in the past is likely to be referenced again
  - If you reference a byte, you'll reference it again soon (think of updating a counter)
- **Spatial locality:** a memory location next to one that was referenced in the past is likely to be referenced in the near future
  - If you reference a byte, you'll soon reference a byte close to it (think of going through an array)

How Much Does Locality Help?

- Let's look at the locality_no_locality.c program on the course Web site and run it…

- This program does a "linear scan" of an array, which leads to the largest possible number of cache hits
  - After loading a memory line, one references all its bytes
- The program then does a "strided scan" of the array
  - After loading a memory line, one references only one of its byte, and then the line is evicted before another one of its bytes is referenced
- Let's look at run results on my laptop

Results on My Laptop

compiler \ flag	-O0	-Ofast
clang	Linear: 2.52 s Strided: 10.51 s **4.17x**	Linear: 0.93 s Strided: 14.0 s **15.01x**
gcc	Linear: 3.11 s Strided: 10.1 s **3.25x**	Linear: 1.05 s Strided: 13.70 s **13.05x**

- Weirdly, both compilers make the strided code slower when optimizing!

Locality in your Programs

- It turns out that most (useful) programs have fairly high temporal/spatial locality
  - Even if the programmer doesn't know what locality is
- But when we strive for high performance we want our code to have the maximum amount of locality
  - And the compiler isn't always good at this!
- A programmer should keep a mental picture of the memory layout of the application data, and reason about locality
  - "Whenever I know that an instruction will bring some data from memory into cache, I should try to reuse that data as much as possible"
- This can be extremely complex, but luckily there are a few well-known techniques and cases
- The first "textbook example" is with 2-D arrays…

Example: 2-D Array Initialization

```
int a[200][200];
for (i=0;i<200;i++) {
  for (j=0;j<200;j++) {
    a[i][j] += 1;
  }
}
```

```
int a[200][200];
for (j=0;j<200;j++) {
  for (i=0;i<200;i++) {
    a[i][j] += 1;
  }
}
```

- Show of hands: which alternative is fastest?
  - i-j order is faster
  - j-i order is faster
  - they are the same

2-D Array Accesses

- I wrote a simple program that initializes a two-dimensional array either along rows or along columns
- It comes with a Makefile that compiles the two version of the code with different compiler optimization flags
- It's on the course Web site (locality_example.zip)
- Let's look at results obtained on my Linux server…
- Let's see if:
  - One loop order is better than the other…
  - What compiler optimization does to performance…

Running Locality Example (clang)

- Results with the clang compiler!

flag / i-j order	-O0	-Ofast
for i; for j	35.48 s	5.25 s
for j; for i	59.97 s	49.34 s

the compiler is not able to optimize for locality at all!

~70% performance loss due to wrong loop order

Running Locality Example (gcc)

- Results with the gcc compiler!

flag / i-j order	-O0	-Ofast
for i; for j	33.56 s	5.29 s
for j; for i	60.56 s	49.30 s

the compiler is not able to optimize for locality at all!

~80% performance loss due to wrong loop order

These numbers are a bit old, we can re-run this right now...

Take Away

- Even on this textbook example, if as a programmer I write the loops in the j-i order, then my program will go slower regardless of what I do with these two compilers!
- So, sadly, as a programmer, I should think about data locality
  - Which is known to be difficult
- First let's understand why the i-j order goes faster than the j-i order….

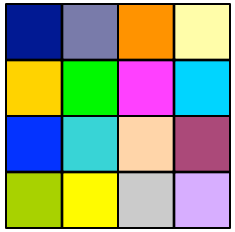
2-D Arrays in Memory

- A static 2-D array is declared as

  ```
  <type> <name>[<size>][<size>]
  ```

- For instance: `int myarray[10][30];`
- The elements of a 2-D array are stored in contiguous memory cells
  - This true in C/C++, not in Java though
- But we now have a problem:
  - The array is 2-D (conceptually)
  - Computer memory is 1-D (just a sequence of addresses)
- Therefore, we need a mapping from 2-D to 1-D
  - From a 2-D abstraction to a 1-D implementation
  - The 2-D abstraction is provided to us by programming languages for convenience
    - Because as humans we like multi-dimensional arrays

Mapping from 2-D to 1-D?



nxn 2-D array

1-D computer memory

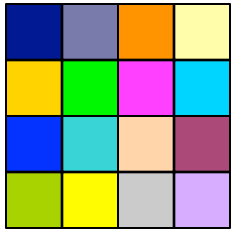
A 2-D to 1-D mapping

Another 2-D to 1-D mapping

$n^2!$ possible mappings

Row-Major, Column-Major

- Luckily, only 2 of the $n^2!$ mappings are implemented in common languages

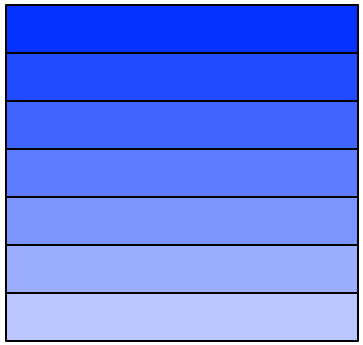


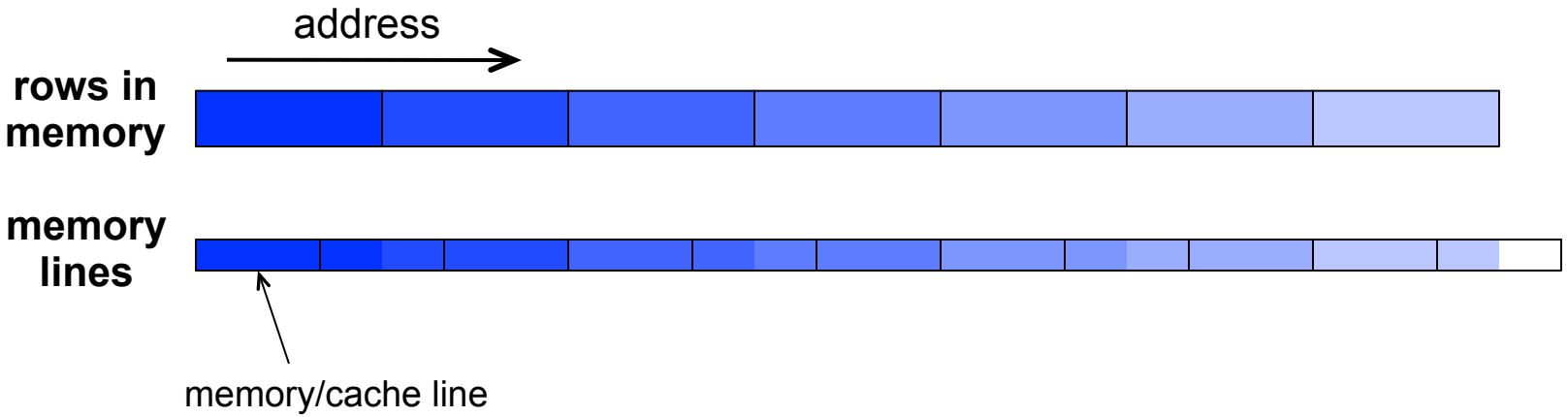
- Row-Major:
  - **Rows are stored contiguously**

- Column-Major:
  - **Columns are stored contiguously**

Row-Major

- C uses Row-Major



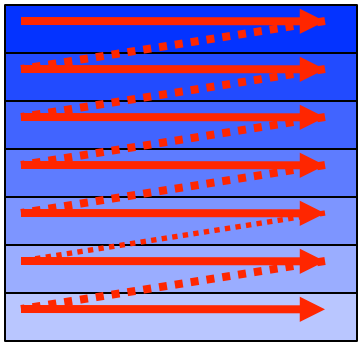


- Array elements are stored in contiguous memory lines

Row-Major

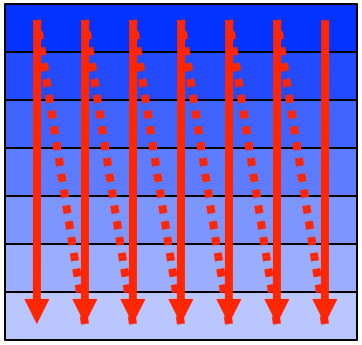
- C uses Row-Major
- First option

```
int a[200][200];
for (i=0;i<200;i++)
  for (j=0;j<200;j++)
    a[i][j] += 1;
```



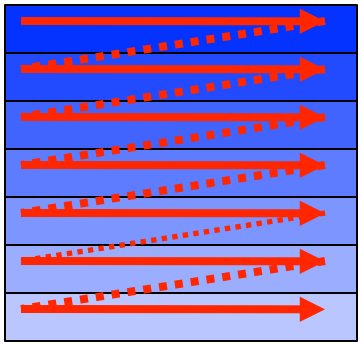
- Second option

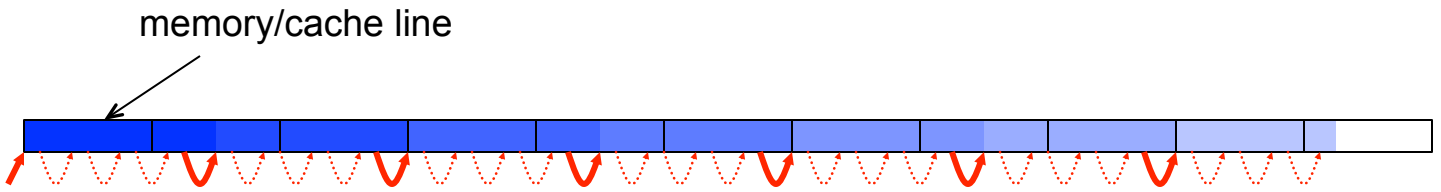
```
int a[200][200];
for (j=0;j<200;j++)
  for (i=0;i<200;i++)
    a[i][j] += 1;
```



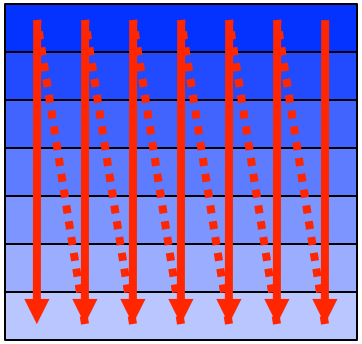
Counting cache misses

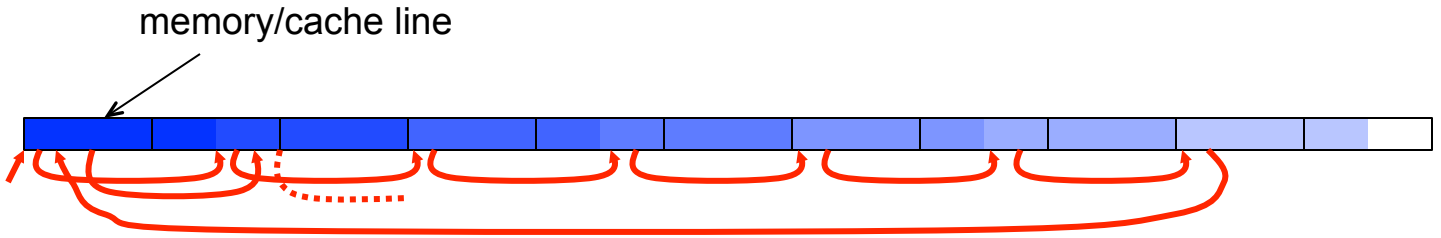
- nxn 2-D array, element size = e bytes, cache line size = b bytes





- One cache miss for every cache line: $n^2 \times e / b$
- Total number of memory accesses: n^2
- Miss rate: e/b
- Example: Miss rate = 4 bytes / 64 bytes = **6.25%**
  - Unless the array is very small



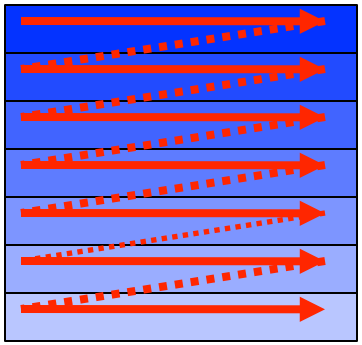


- One cache miss for every access
- Example: Miss rate = **100%**
  - Unless the array is very small

Array Initialization in C

- First option

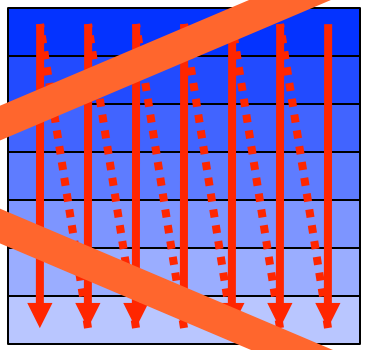
```
int a[200][200];
for (i=0;i<200;i++)
  for (j=0;j<200;j++)
    a[i][j]=2;
```



Great Locality

- Second option

```
int a[200][200];
for (j=0;j<200;j++)
  for (i=0;i<200;i++)
    a[i][j]=2;
```



Awful Locality

Counting Cache Misses

- It would be interesting to count cache misses to see that the differences in performance are really due to the memory bottleneck
- We can reason about the code and the hardware, but that can get really difficult
- There are tools to measure this
- On Linux: perf
  - sudo apt install linux-tools-generic
- Can be used to count Lowest Level Cache (LLC) misses
  - perf stat -e LLC-misses <command>
- For our locality example program, the j-i order leads to about 30x more LLC cache misses than the i-j order

Loop Fusion

- Consider the following code:

```
double a[N], b[N];
for (i=0;i<N;i++) {
  a[i] = i*i;
}
for (i=0;i<N;i++) {
  b[i] = a[i] + (double)i;
}
```

- In this code, the second loop experiences cache misses when accessing array a
- Although array a was loaded into RAM entirely, if N is large, it is no longer in cache
- If we fuse the two loops we get better data locality
  - And less loop overhead!

Loop Fusion

- Consider the following code:

```
double a[N], b[N];
for (i=0;i<N;i++) {
  a[i] = 2.0;
}
for (i=0;i<N;i++) {
  b[i] = a[i] + (double)i;
}
```

```
double a[N], b[N];
for (i=0;i<N;i++) {
  a[i] = I * i;
  b[i] = a[i] + (double)i;
}
```

- In this code, the second loop experiences cache misses when accessing array a
- Although array a was loaded into RAM entirely, if N is large, it is no longer in cache
- If we fuse the two loops we get better data locality
  - And less loop overhead!

Matrix Multiplication

- A classic example for locality-aware programming is matrix multiplication

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] += a[i][k] * b[k][j];
```

- There are 6 possible orders for the three loops
  - i-j-k, i-k-j, j-i-k, j-k-i, k-i-j, k-j-i
- Each order corresponds to a different access patterns of the matrices
- Let's focus on the inner loop, as it is the one that's executed most often

Matrix Multiplication

- To determine the best i-j-k order, we have two options
- Option #1: pragmatic
  - Implement the 6 options
  - Run them on large matrices see which one's faster
  - Use perf to count cache misses and support our findings
- Option #2: "theory"
  - Reason about locality in our program

- We can all do Option #1 easily, so let's do Option #2

Inner Loop Memory Accesses

```
for (i=0;i<N;i++)
      for (j=0;j<N;j++)
            for (k=0;k<N;k++)
                  c[i][j] += a[i][k] * b[k][j];
```

- Reasoning about the whole code above it too complicated
- We note that the inner loop is executed n^2 times
- Se a common technique is to simply think of the inner loop
- Each matrix element can be accessed in three modes in the inner loop
  - Constant: doesn't depend on the inner loop's index
  - Sequential: contiguous addresses
  - Strided: non-contiguous addresses (N elements apart)

Inner Loop Memory Accesses

- Each matrix element can be accessed in three modes in the inner loop
  - Constant: doesn't depend on the inner loop's index
  - Sequential: contiguous addresses
  - Strided: non-contiguous addresses (N elements apart)

c[i][j]	+=	a[i][k]	*	b[k][j];
i-j-k: Constant		Sequential		Strided
i-k-j: Sequential		Constant		Sequential
j-i-k: Constant		Sequential		Strided
j-k-i: Strided		Strided		Constant
k-i-j: Sequential		Constant		Sequential
k-j-i: Strided		Strided		Constant

Loop order and Performance

- Constant access is better than sequential access
  - it's always good to have constants in loops because they can be put in registers (as we've seen in our very first optimization)
- Sequential access is better than strided access
  - sequential access is better than strided because it utilizes the cache better
- Now we can rank all 6 options

Best Loop Ordering?

c[i][j]	+=	a[i][k]	*	b[k][j];
i-j-k: Constant		Sequential		Strided
i-k-j: Sequential		Constant		Sequential
j-i-k: Constant		Sequential		Strided
j-k-i: Strided		Strided		Constant
k-i-j: Sequential		Constant		Sequential
k-j-i: Strided		Strided		Constant

- k-i-j and i-k-j have the *best* performance
- i-j-k and j-i-k have *worse* performance
- j-k-i and k-j-i have the *worst* performance
- Let's run this and see... (mm_locality_example.zip)

What about Java?

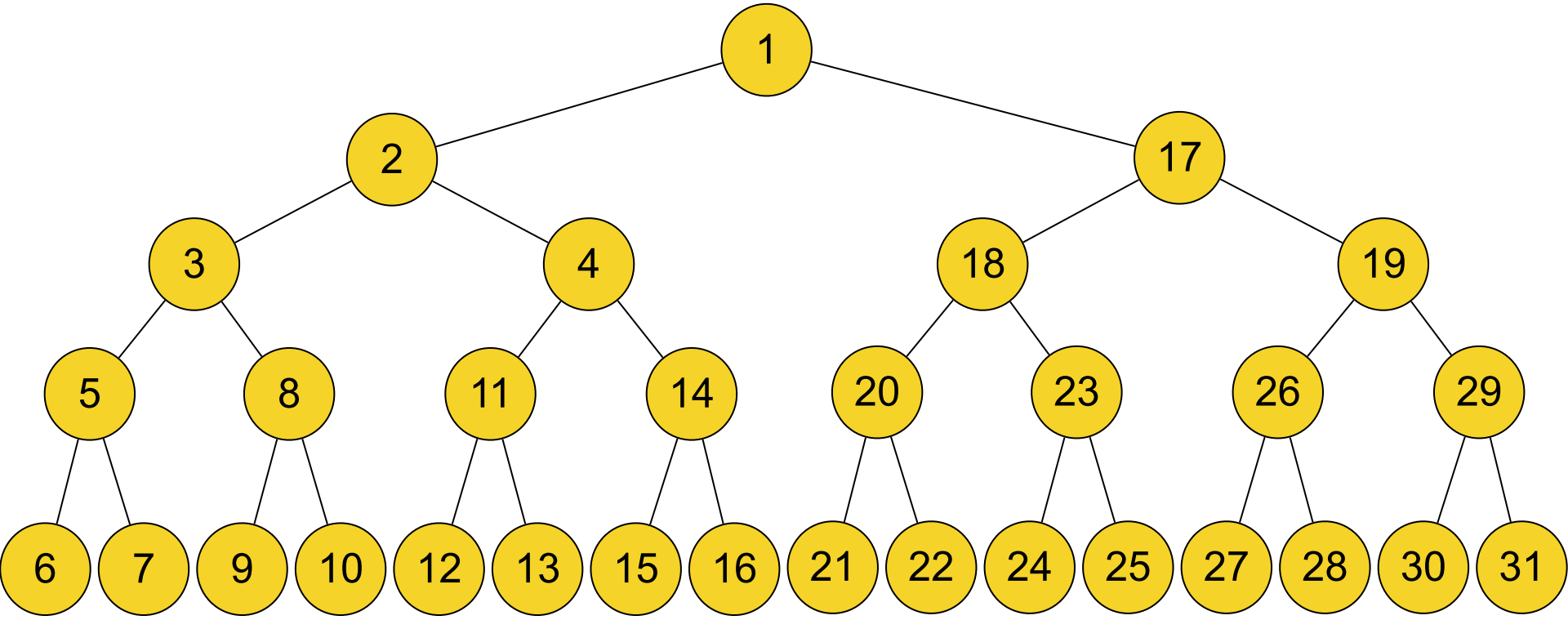
- In Java a 2-D array is not a single contiguous zone of memory, but an array of pointers to row arrays
- To each row of a 2-D array could be stored in a completely different zone of RAM
- Regardless, like in C, locality is good when accessing arrays along rows, and not good when accessing arrays along columns
- Easy to check with a simple program (let's run RowColMajor.java on course Web site)

Programming for Locality

- When designing data structures, and when designing programs that operate on data structures, performance can be gained by increasing data locality
  - e.g., Java's ArrayList vs. LinkedList
- But it can be a lot of work and make the code less readable
- Classic situation: a code with data structures full of pointers everywhere
  - Great for convenience/expressivity
  - Not great for locality ("pointer chasing")
- Developers have to make calls regarding the trade-off between "clean/convenient" and "fast"
  - Sometimes one hits a "best of both worlds jackpot"

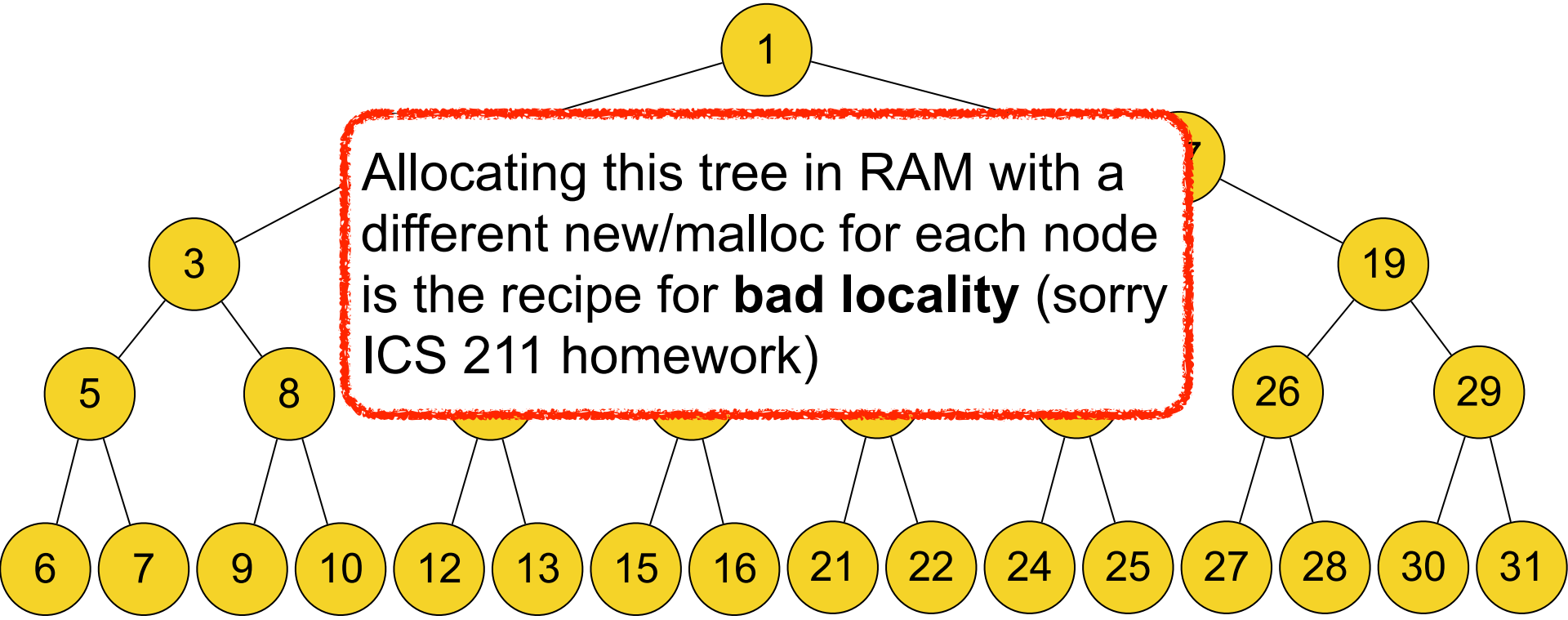
Data Structures and Locality

- One difficult problem is picking/implementing data structures that will improve locality
- Let's use a guiding example of a binary search tree



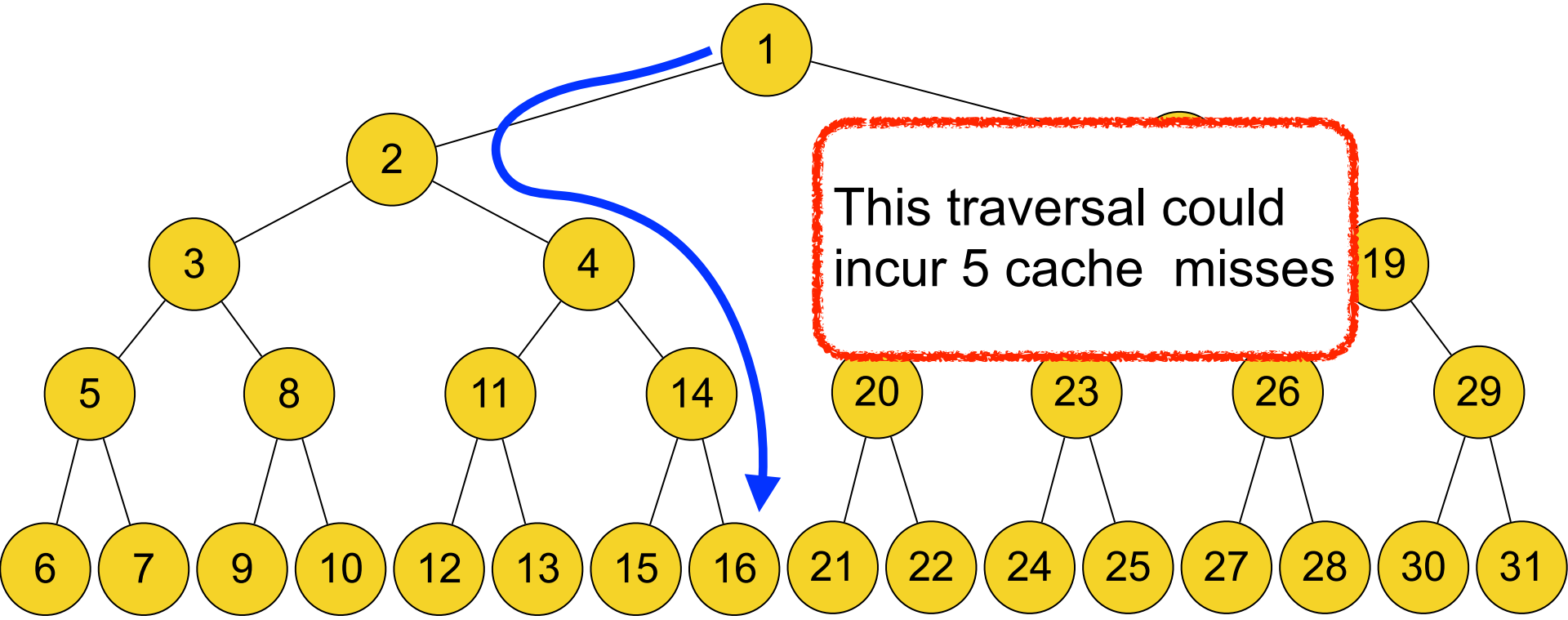
Data Structures and Locality

- One difficult problem is picking/implementing data structures that will improve locality
- Let's use a guiding example of a binary search tree



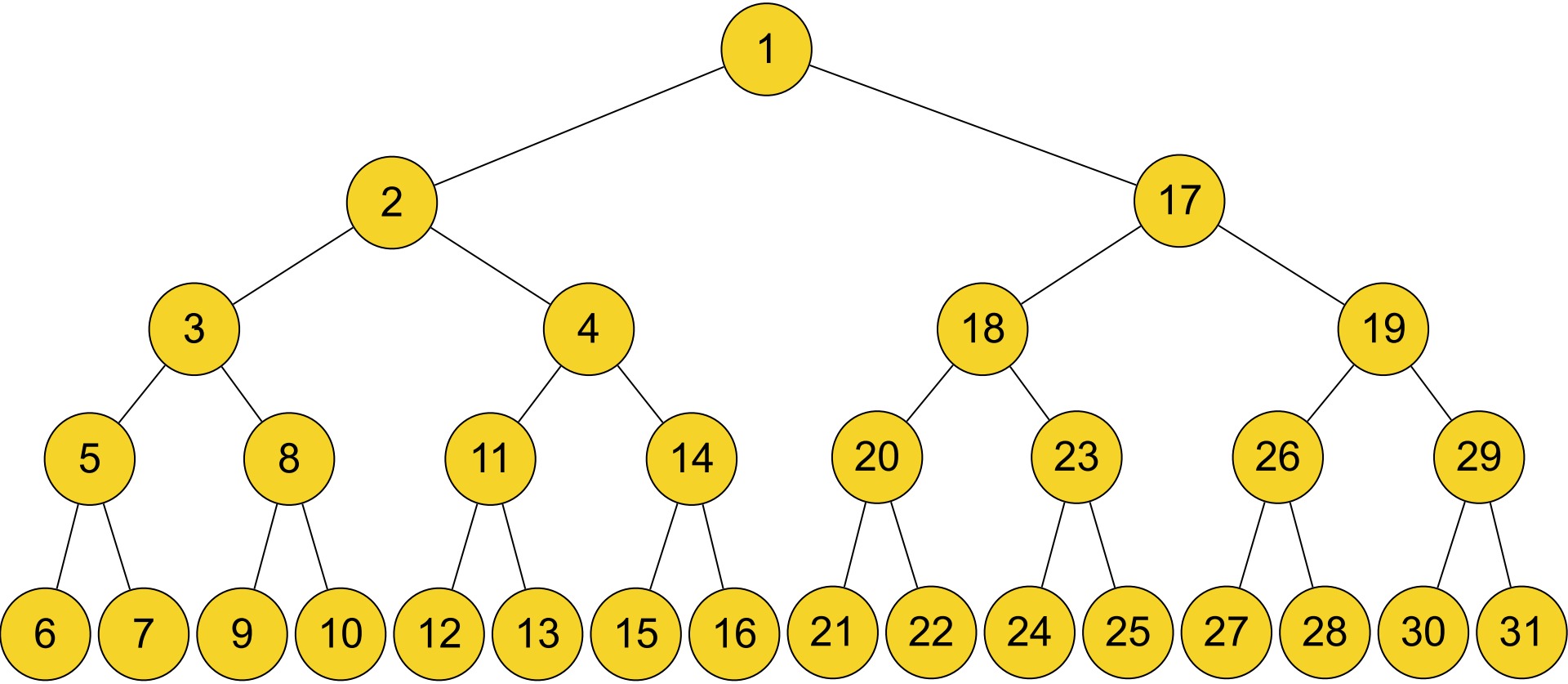
Data Structures and Locality

- One difficult problem is picking/implementing data structures that will improve locality
- Let's use a guiding example of a binary search tree



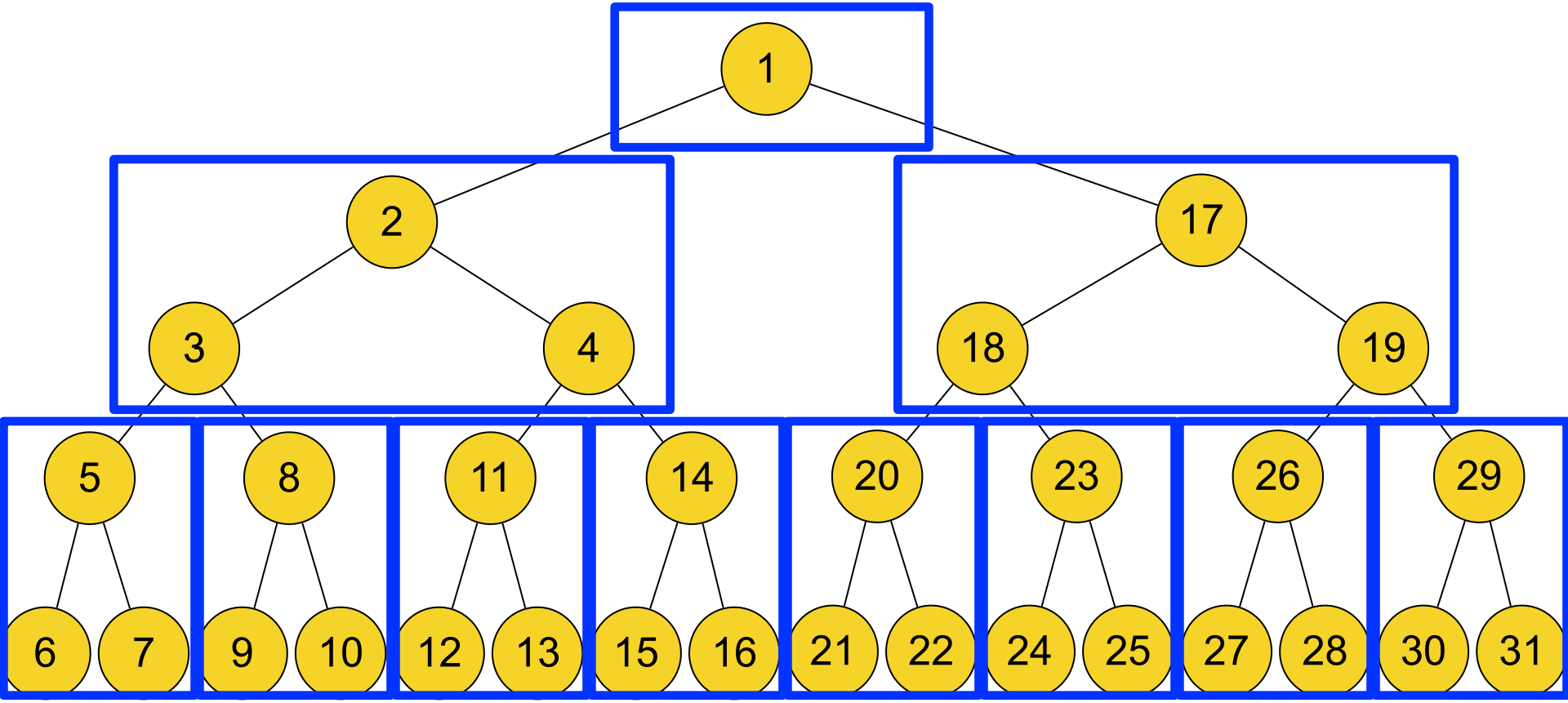
Data Structure Memory Layout

- We need to come up with a good memory layout for our data structure



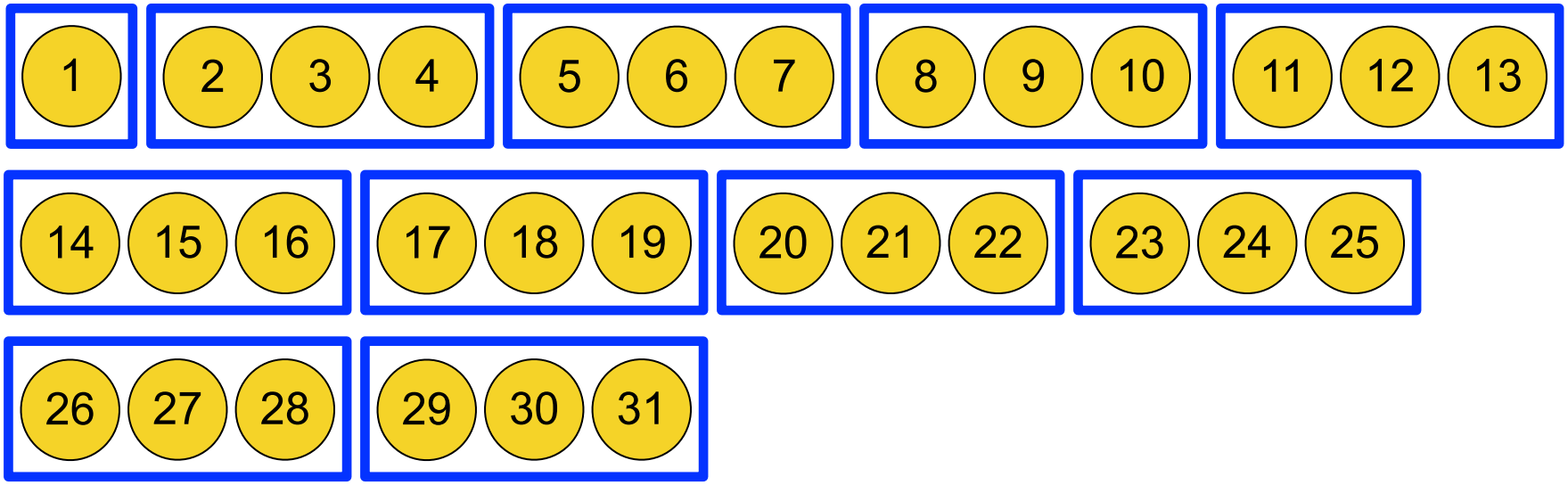
Data Structure Memory Layout

- Let's make sure we allocate particular nodes next to each other in RAM (i.e., in arrays)



Data Structure Memory Layout

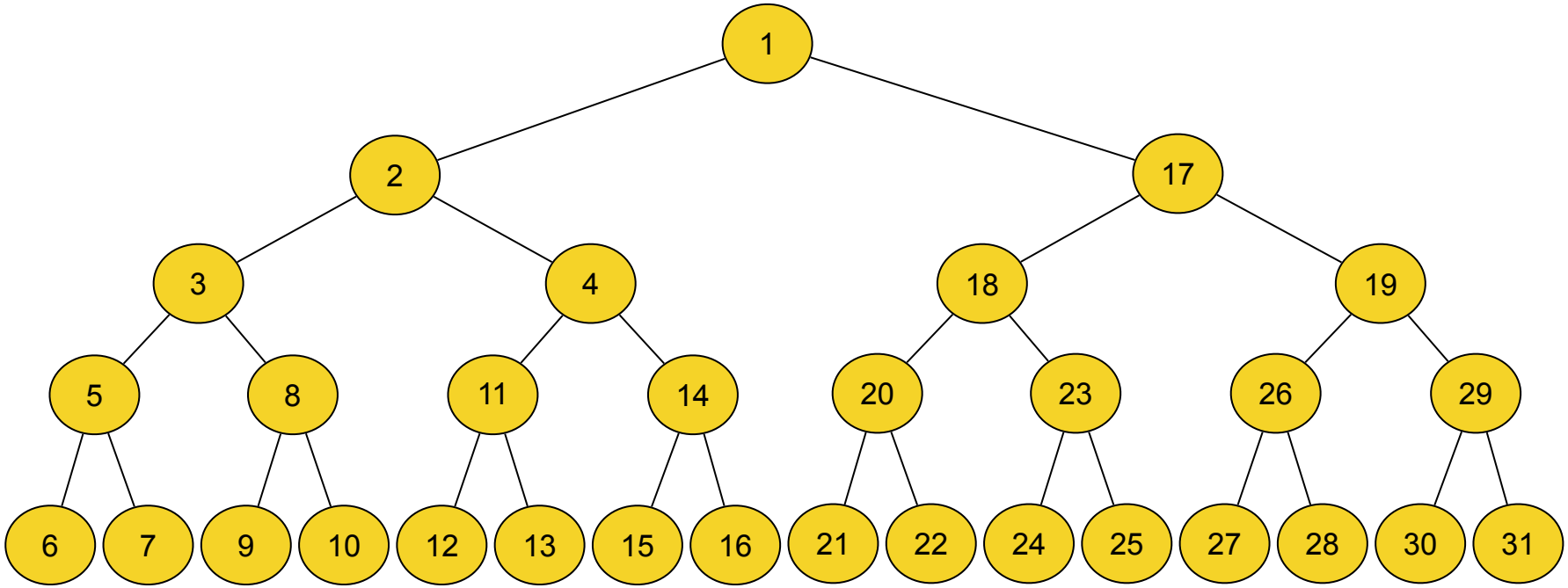
- We think of is as a tree, but it's really a big array

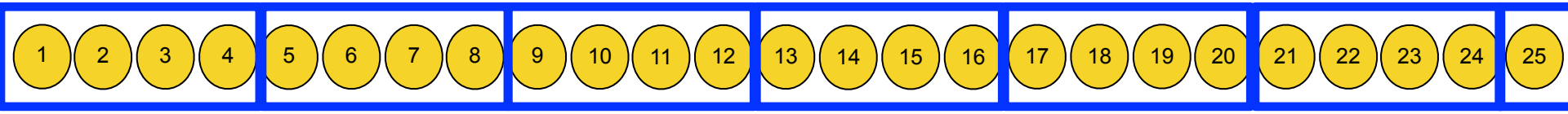


- Determining a node children/parent is now based on simple-ish discrete math based on array indices
  - We made the implementation much less convenient, but that the price we pay for better locality
  - Arrays are just good for locality :(

Data Structure Memory Layout

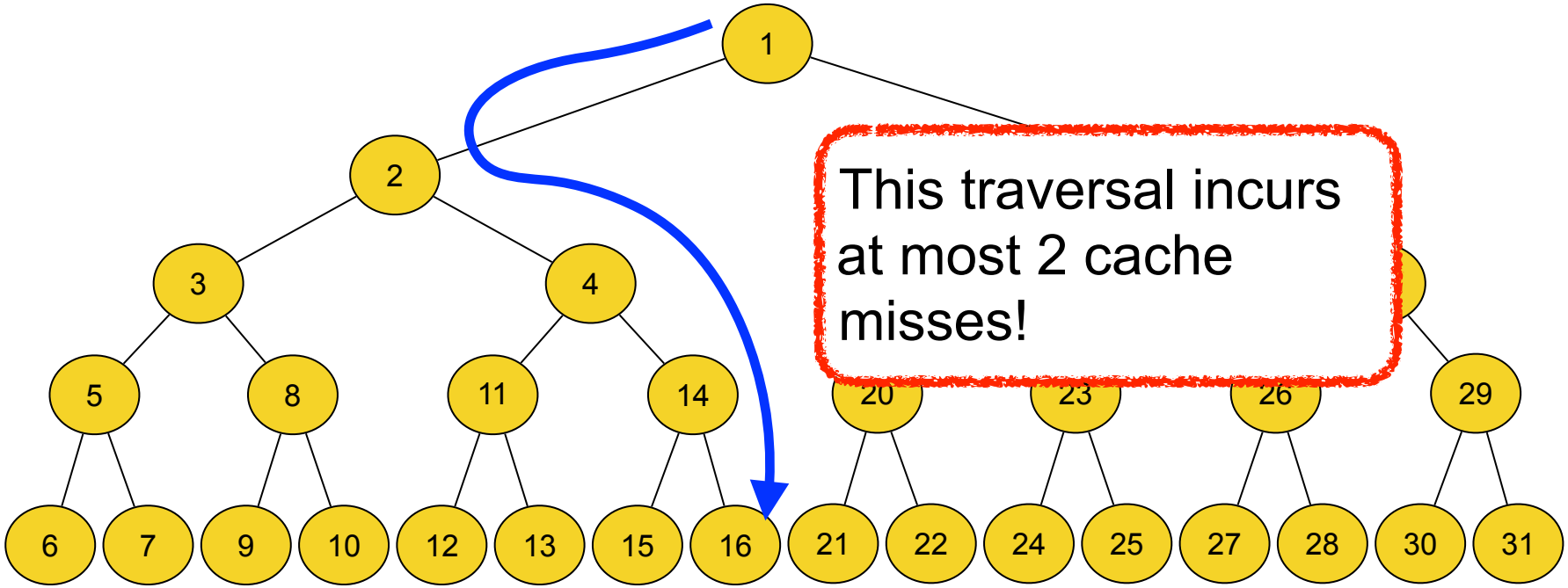
- Say that 4 nodes fit in a cache line

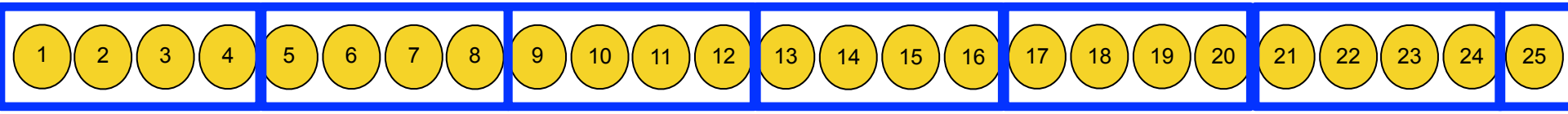




Data Structure Memory Layout

- Say that 4 nodes fit in a cache line





Cache-Aware

- If in your program you explicitly use the size of the cache and/or of the cache line as a parameter to make decisions, one say that the program is cache-aware
- In out previous example, our program could determine at compile/run time the cache line size, which then tells use the best size of our blue boxes, which then defines the in-memory layout
- And now, we have improved locality
- You can see how this gets complicated, especially because there are multiple levels of cache (in a few slides)
- Given a bunch of caches, each with their own cache line sizes, figuring out the best memory layout for a useful data structure is very difficult
  - But a lot of smart people have done it

Cache-Oblivious

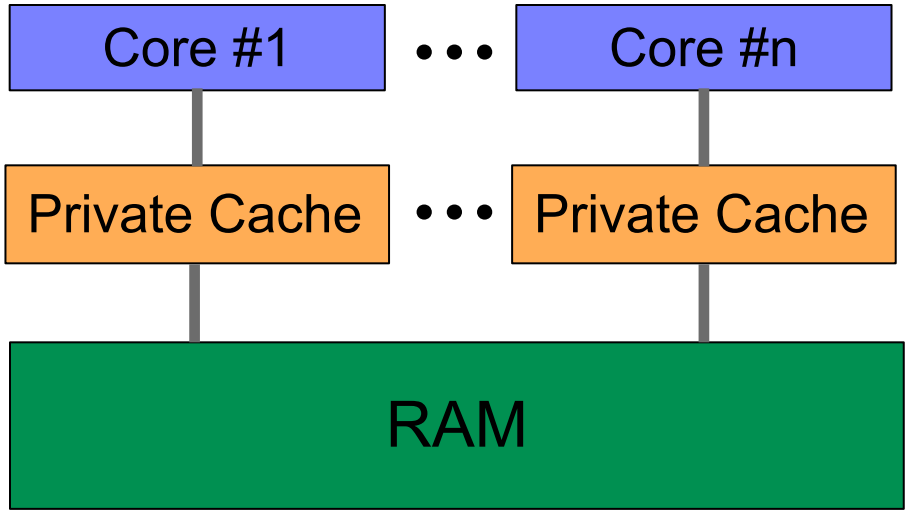
- Wouldn't it be great if your data structure layout promoted locality for any cache configuration?
- This is called cache-obliviousness: the program does not explicitly use the size of the cache or cache lines as a parameter, and yet achieves good locality
- This has been a very active field of research and development and there are cache-oblivious layouts
- For our binary search tree example, the van Emde Boas layout is a cache-oblivious solution
  - It's a somewhat complicated recursive layout
- See an advanced data-structure course for more on this kind of data-structures
  - Basically, take all the good stuff in ICS311 and then say "what about locality?" e.g., a binary search on a sorted array has terrible locality!

Multi-Threading

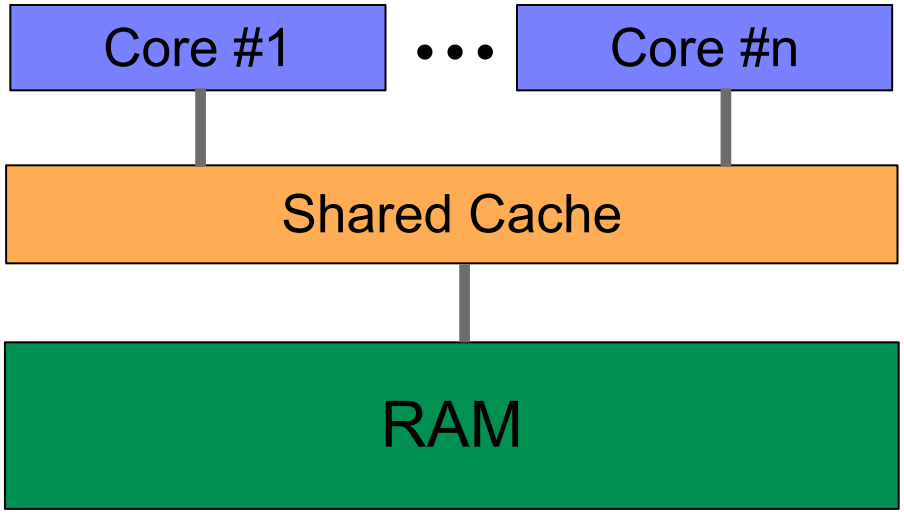
- Reasoning about locality for single-threaded programs is complicated
- When one throws multi-threading into the mix, it's even more complicated
- We want to avoid threads competing for the cache (i.e., evicting each other's data from cache)
- Ideally, threads would cooperate: a thread loads in cache something that other threads happen to need
- See prof. Sitchinava's Parallel Algorithms course for more on such topics
- Let's look at the hardware deals with caches in multi-core machines…

Multi-Core and Caches

- Where are the caches in a multi-proc/core machine?
- Two options:



private caches



shared caches

Shared Caches: Good and Bad

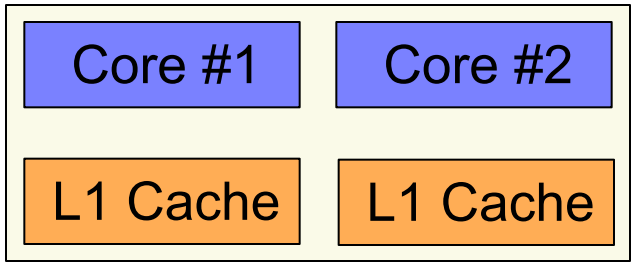
- Shared is good:
  - Cache placement identical to single cache
    - Only one copy of any cached block
    - Can't have different values for the same memory location
  - Good interference
    - One processor may prefetch data for another
    - Two processors can each access data within the same cache block, enabling fine-grain sharing
- Shared is bad:
  - Bandwidth limitation
    - Difficult to scale to a large number of processors/cores
    - Keeping all processors working in cache requires a lot of bandwidth
  - Size limitation
    - Building a fast large cache is expensive
  - Bad interference
    - One processor may flush another processor's data from cache

Shared Caches

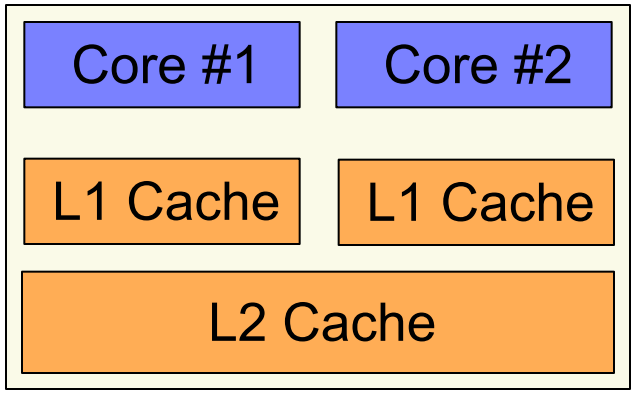
- Shared caches have known a strange evolution
- Early 1980s
  - Alliant FX-8
    - 8 processors with crossbar to interleaved 512KB cache
  - Encore & Sequent
    - first 32-bit microprocessors
    - two procs per board with a shared cache
- Then disappeared
- Only to reappear in recent MPPs
  - Cray X1: shared L3 cache
  - IBM Power 4 and Power 5: shared L2 cache
- Typical multi-proc systems do not use shared caches
- But they are now common in multi-core systems

Caches and multi-core

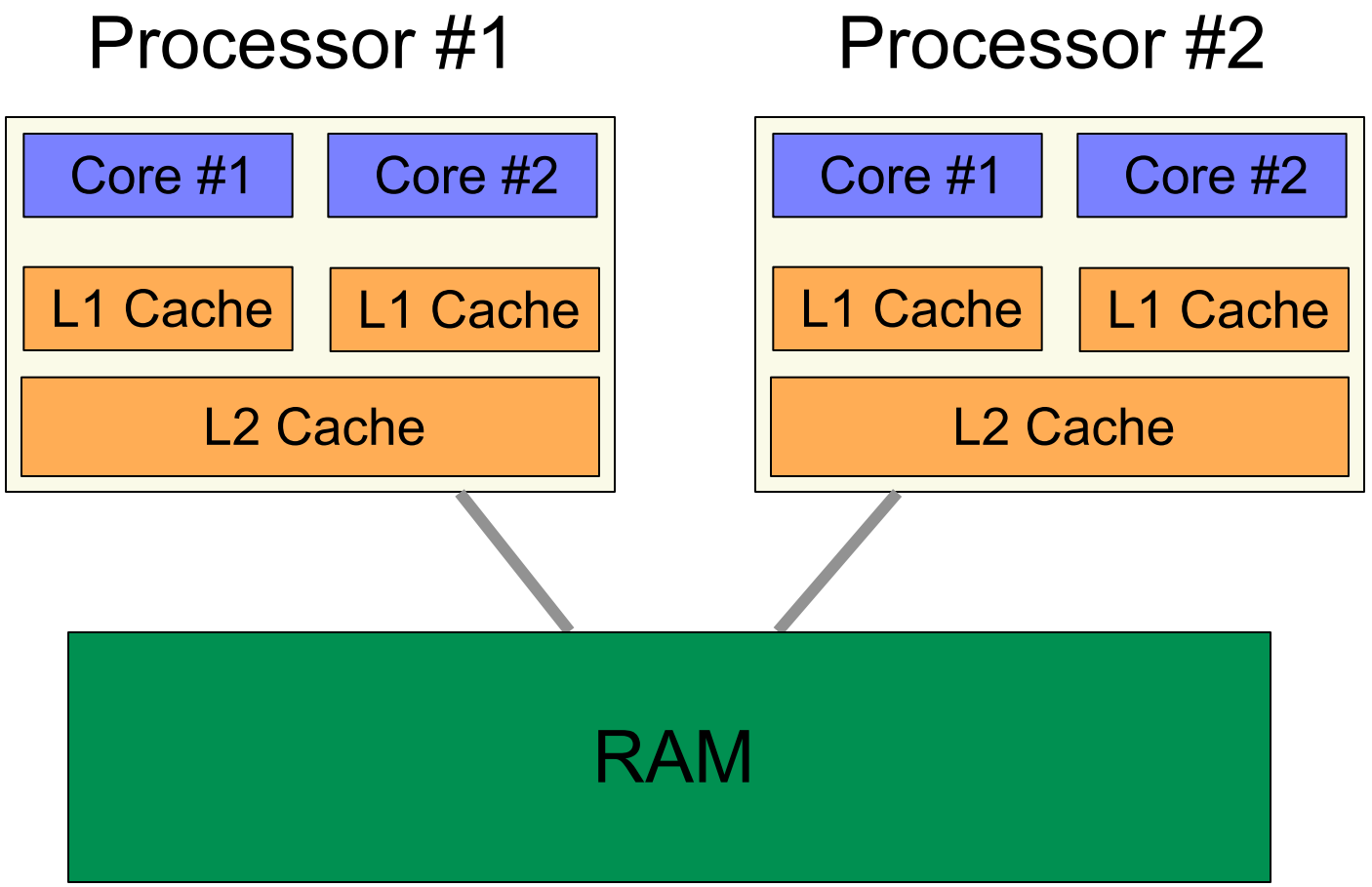
- Typical multi-core architectures use distributed L1 caches



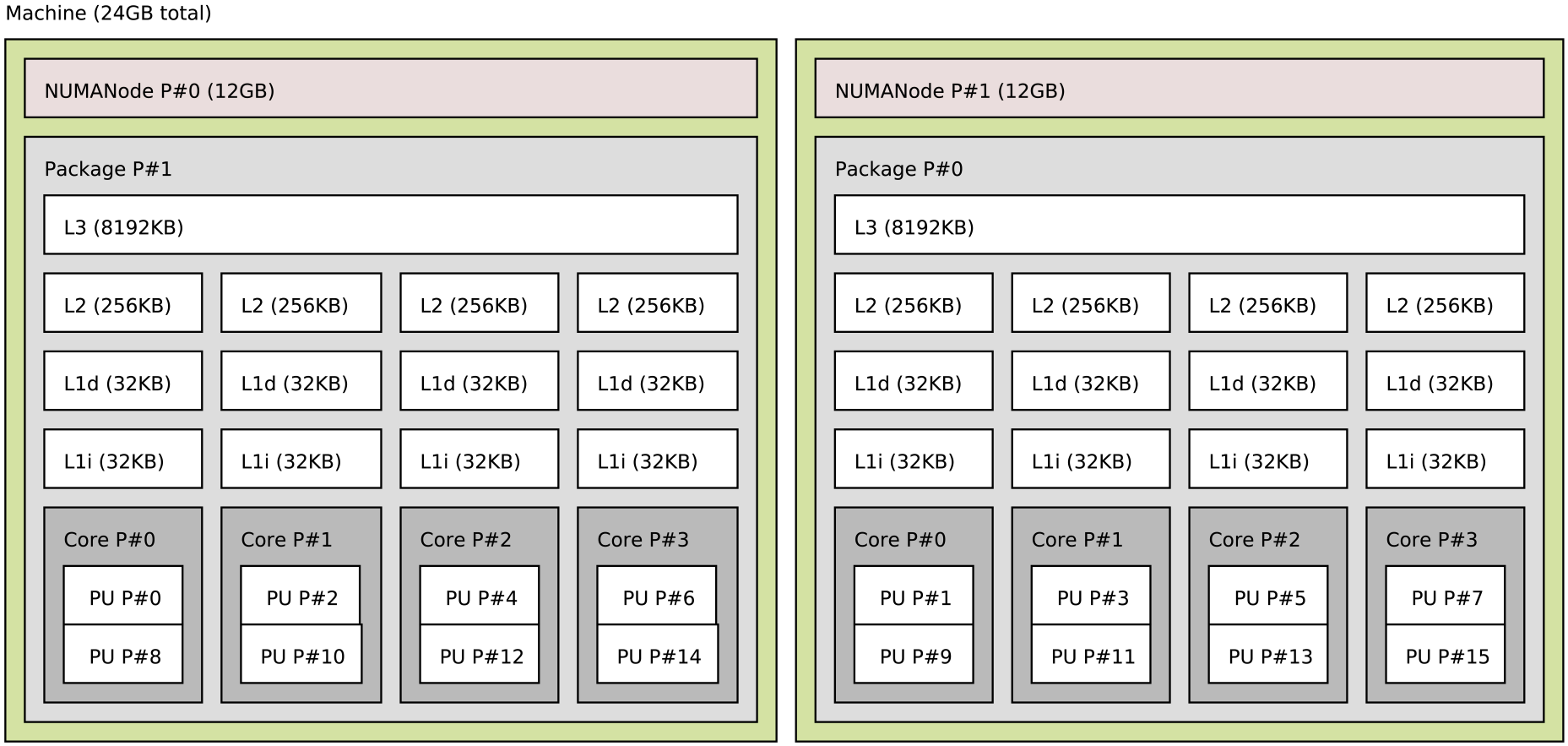
- But lower levels of caches are shared



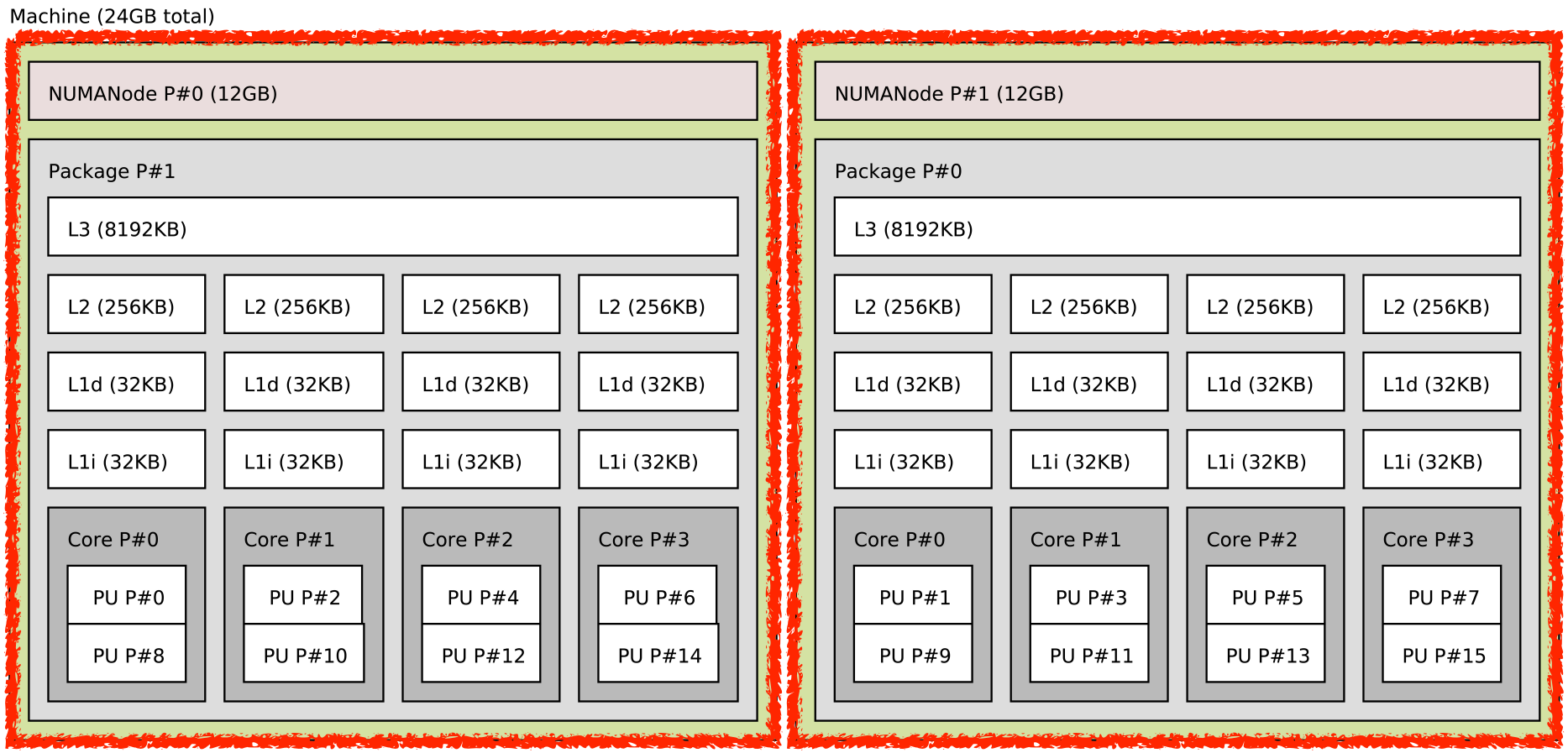
Multi-proc & multi-core systems



My Linux Server (lstopo)

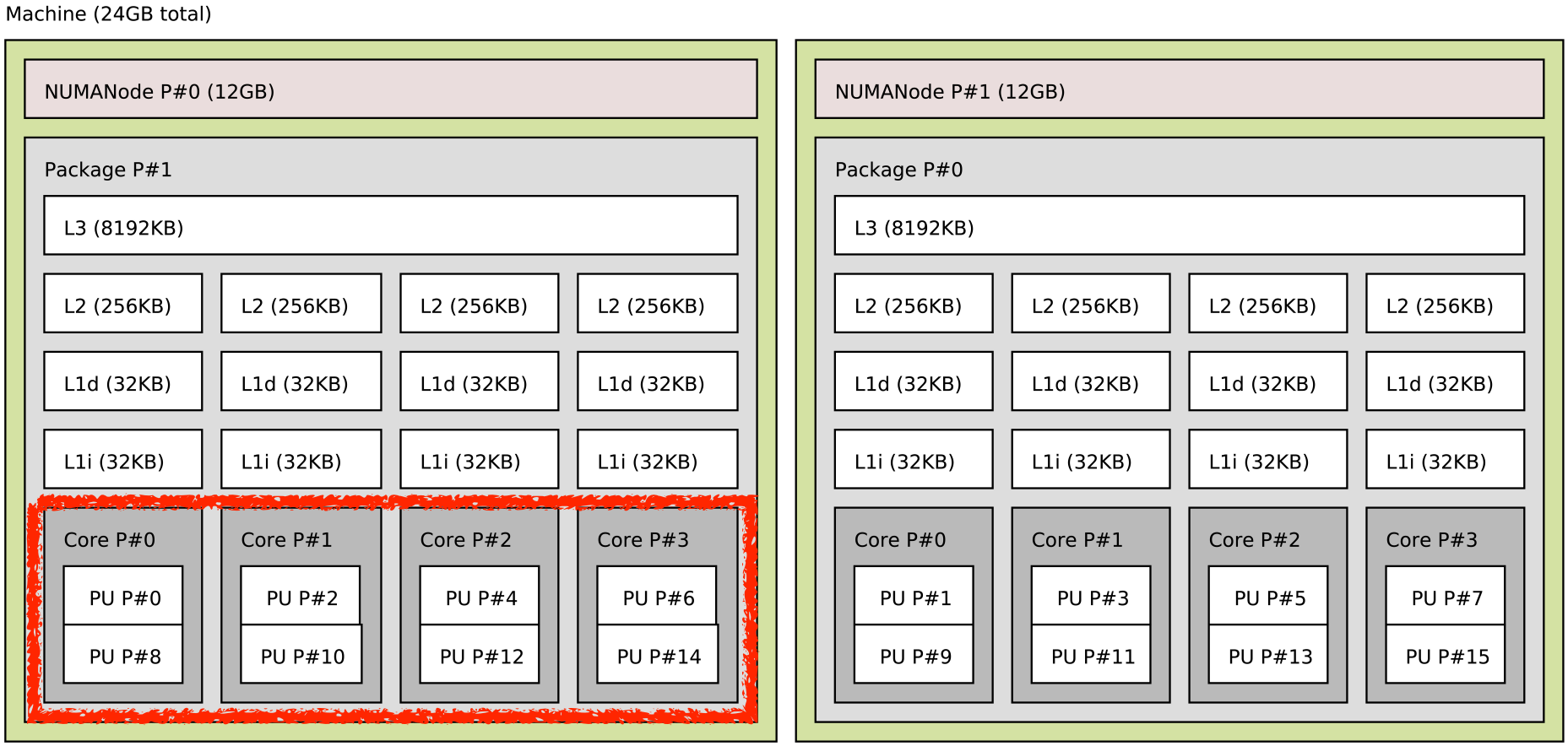


My Linux Server



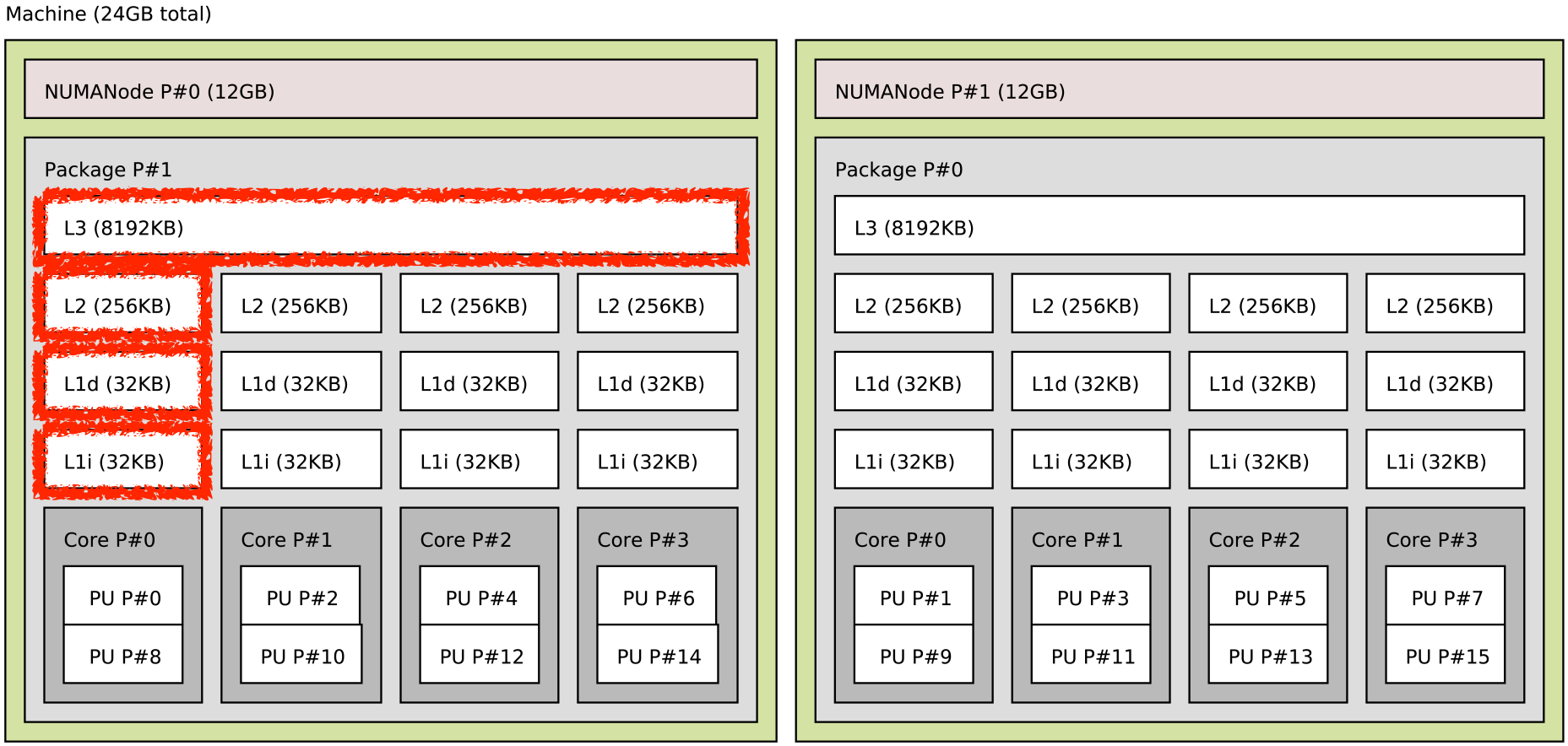
Two processors (one per “socket”)

My Linux Server



4 cores per processor
(or is it 8? stay tuned)

My Linux Server



Shared L3 cache

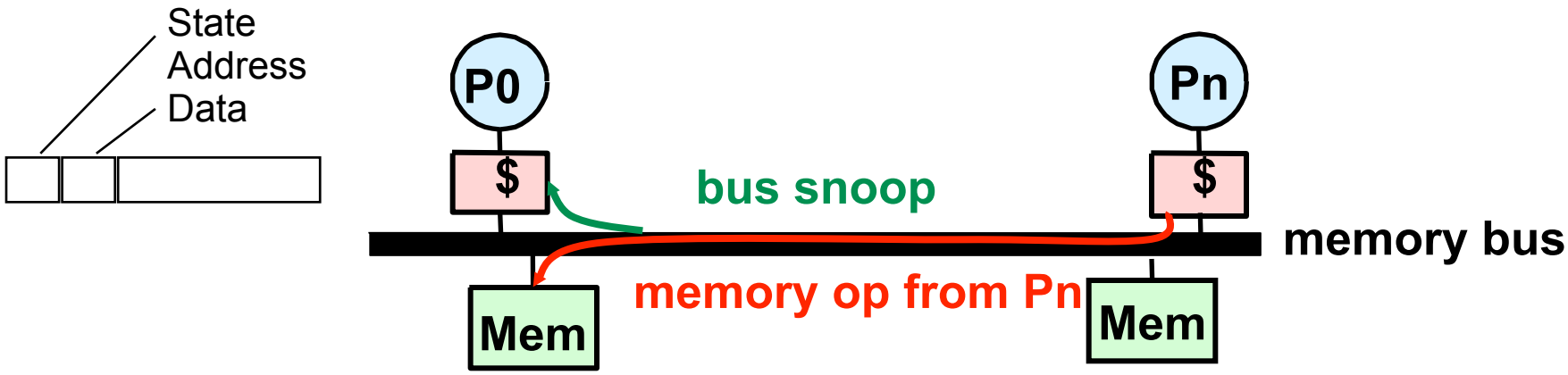
Private L2 and L1 caches

(separate L1 data and instruction caches)

Private Caches

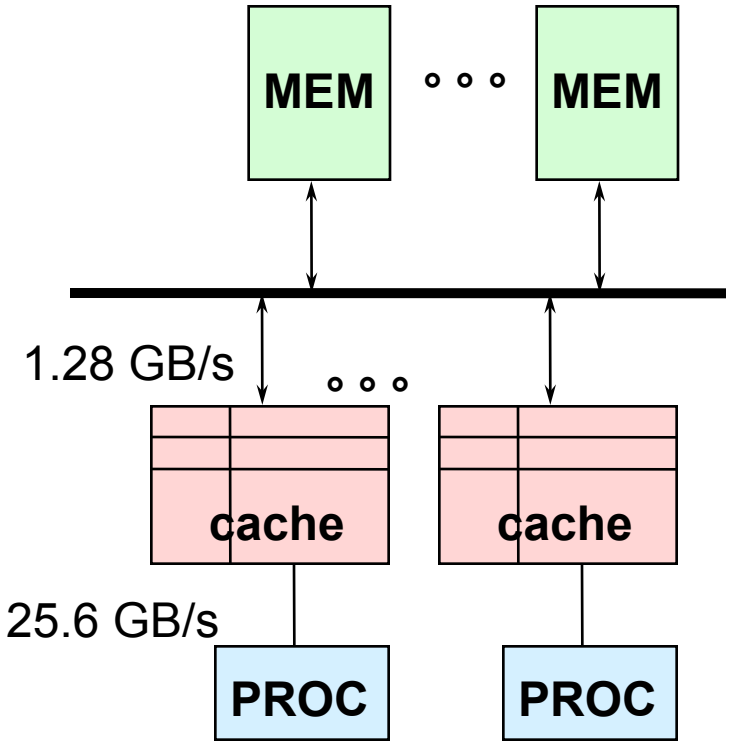
- The main problem with private caches is that of memory consistency
- Memory consistency is jeopardized by having multiple caches
  - P1 and P2 both have a cached copy of a data item
  - P1 writes to it, possibly write-through to memory
  - At this point P2 owns a stale copy
- When designing a multi-processor system, one must ensure that this cannot happen
  - By defining protocols for cache coherence

Snoopy Cache-Coherence



- The memory bus is a broadcast medium
- Caches contain information on which addresses they store
- Cache Controller "snoops" all transactions on the bus
  - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
  - Take action to ensure coherence
    - invalidate, update, or supply value
- This is the kind of thing hapenning, for instance, in the Intel i7 processor (with many bells and whistles)

Limits of Snoopy Coherence



Assume:

4 GHz processor

=> 16 GB/s inst BW per processor (32-bit)

=> 9.6 GB/s data BW at 30% load-store of 8-byte elements

Suppose 98% inst hit rate and 90% data hit rate

=> 320 MB/s inst BW per processor

=> 960 MB/s data BW per processor

=> 1.28 GB/s combined BW

Assuming 10 GB/s bus bandwidth

8 processors will saturate the bus

Directory-based Coherence

- Idea: Implement a “directory” that keeps track of where each copy of a data item is stored
- The directory acts as a filter
  - processors must ask permission for loading data from memory to cache
  - when an entry is changed the directory either update or invalidate cached copies
- Eliminates the overhead of broadcasting/ snooping, thus bandwidth consumption is reduced
- But is slower in terms of latency
- Used to scale up to numbers of processors that would saturate the memory bus

Example machine

- SGI Altix UV
- 2,560 cores
- 16TB of Shared Memory
- Uses a mixture of snoopy and directory-based coherence
- Global address space is possible for multiple such nodes connected over a switch...
- Costs a lot of money!
  - But then you don't have to take ICS632 :)

Conclusion

- Locality is important due to caches and is thus a constant concern for performance
- Compilers are not great at dealing with it
- Bottom-line: locality makes the programmer's life difficult but has high payoff if done right
  - A lot of algorithmic and implementation complexity/difficulty
- This leads to a lot of interesting research on locality issues
  - Awesome if you're a computer science graduate student / researcher in needs of a research topic
  - Not so great if you're an engineer and just want some code to go fast
- Let's look at Homework Assignment #12… (last, and short, one!)