High Performance Computing for Science and Engineering I

Exercise 2: Cache Design, Shared Memory

Ivica Kičić
kicici@ethz.ch
Computational Science and Engineering Laboratory

Overview

- Question 1:
  - Memory access speed vs. array size?
  - (demonstrating L1, L2 and L3 **cache size**)
- Question 2:
  - Memory access speed vs. # of simultaneously used arrays?
  - (demonstrating **cache associativity**)
- Question 3:
  - N-thread lock
  - (demonstrating **mutual exclusion**)

Overview

- Question 1:
  - Memory access speed vs. array size?
  - (demonstrating L1, L2 and L3 **cache size**)
- Question 2:
  - Memory access speed vs. # of simultaneously used arrays?
  - (demonstrating **cache associativity**)
- Question 3:
  - N-thread lock
  - (demonstrating **mutual exclusion**)

How you should behave regarding memory access.

Overview

- Question 1:
  - Memory access speed vs. array size?
  - (demonstrating L1, L2 and L3 **cache size**)
- Question 2:
  - Memory access speed vs. # of simultaneously used arrays?
  - (demonstrating **cache associativity**)

How you should behave regarding memory access.

- Question 3:
  - N-thread lock
  - (demonstrating **mutual exclusion**)

A rough idea on how locks work.
Visualization of mutual exclusion.

Overview

- Question 1:
  - Memory access speed vs. array size?
  - (demonstrating L1, L2 and L3 **cache size**)

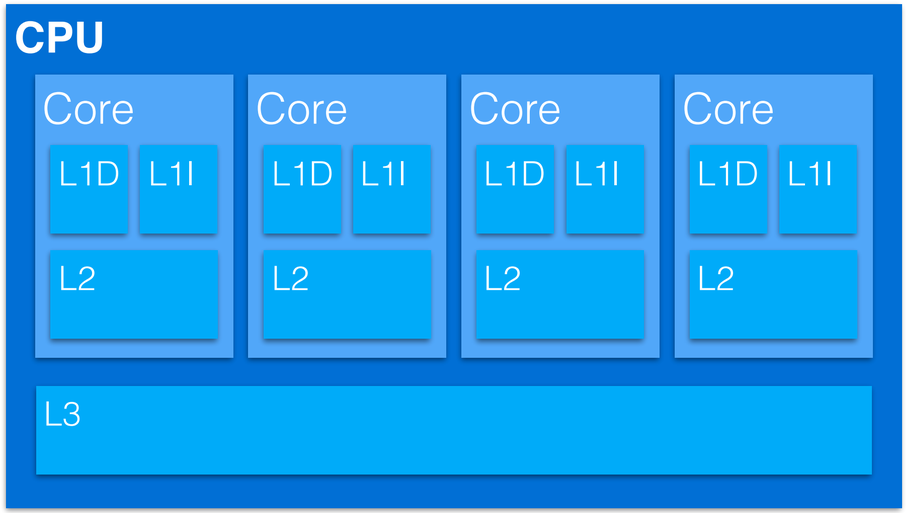
How you should behave regarding memory access.

- Question 2:
  - Memory access speed vs. # of simultaneously used arrays?
  - (demonstrating **cache associativity**)

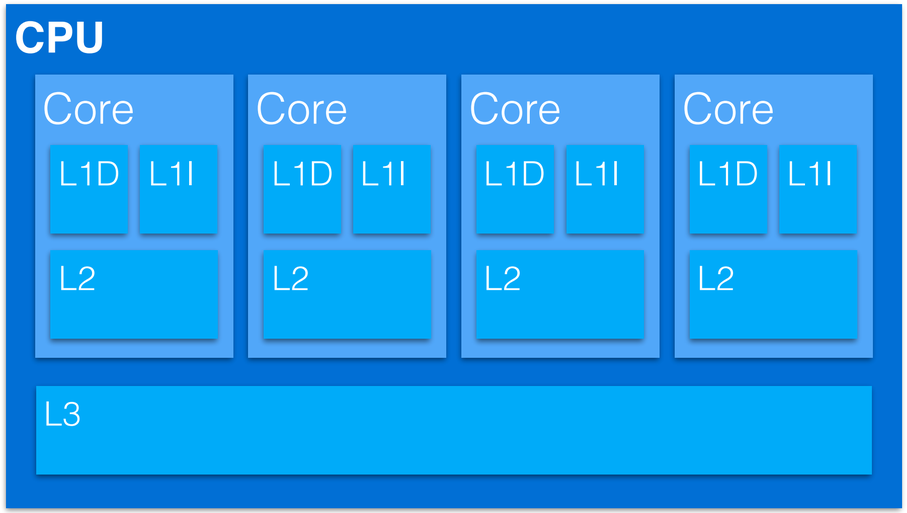
- Question 3:
  - N-thread lock
  - (demonstrating **mutual exclusion**)

A rough idea on how locks work. Visualization of mutual exclusion.

Overview of the Cache Design

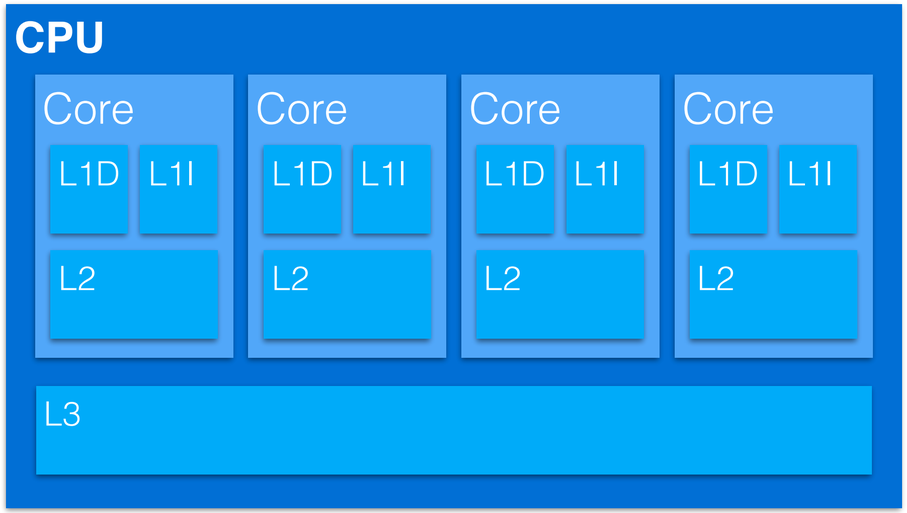


Overview of the Cache Design



- Cache = Temporary local memory
- Closer to CPU = Faster, but smaller

Overview of the Cache Design



- Cache = Temporary local memory
- Closer to CPU = Faster, but smaller
- Example:

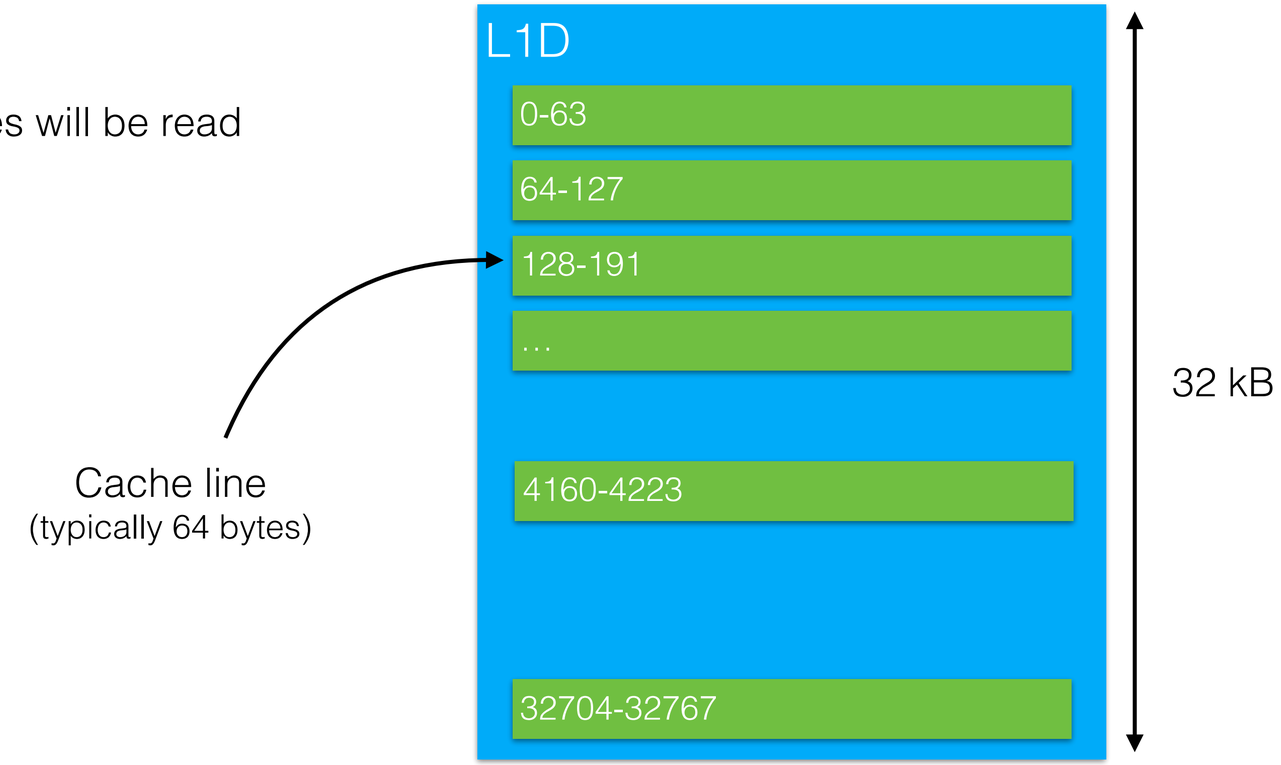
	Size	Latency [cycles]	Associativity
L1	32 kB	4	8
L2	256 kB	14	8
L3	3-24MB	34-85	4-16
DRAM	GB-TB	100+	N/A

} Intel Skylake*

* https://www.agner.org/optimize/microarchitecture.pdf

Cache Line

- Data stored in chunks of 64 bytes
- If a single double (8 bytes) is accessed, 56 more bytes will be read
  - 7 more doubles for free!



Question 1: Memory access speed vs. array size?

Setup:

Question 1: Memory access speed vs. array size?

Setup:

Question 1: Memory access speed vs. array size?

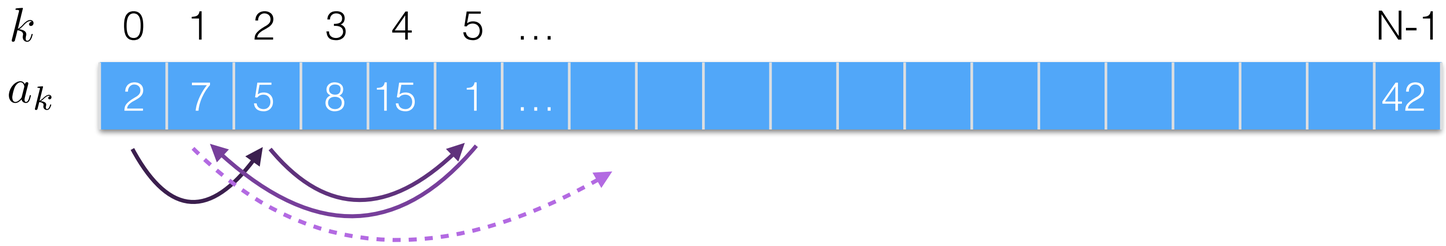
Setup:

Question 1: Memory access speed vs. array size?

Setup:

Question 1: Memory access speed vs. array size?

Setup:



Question 1: Memory access speed vs. array size?

Setup:

Question 1: Memory access speed vs. array size?

Setup:

Repeat M times:

$k \leftarrow a_k$

Question 1: Memory access speed vs. array size?

Setup:

Repeat M times:

$k \leftarrow a_k$

Question:

$$\eta = \frac{M}{\text{total time}} = ?$$

Question 1: Memory access speed vs. array size?

Setup:

$$\text{Repeat } M \text{ times:} \quad k \leftarrow a_k$$

$$\text{Question:} \quad \eta = \frac{M}{\text{total time}} = ?$$

Expectation:

Larger N —> don't fit into (small) caches —> slower than smaller N

Question 1: Memory access speed vs. array size?

Setup:

Repeat M times:
$k \leftarrow a_k$

Question:
$$\eta = \frac{M}{\text{total time}} = ?$$

Expectation:

Larger N —> don't fit into (small) caches —> slower than smaller N

Variant 1: $a_k =$ a random one-cycle permutation (Sattolo's algorithm, see skeleton code)

Variant 2: $a_k = (k+1)\%N$

Variant 3: $a_k = \left(k + \frac{\text{cache line size}}{\text{sizeof(int)}}\right)\%N$

Question 1: Task

Variant 1: $a_k =$ a random one-cycle permutation

Variant 2: $a_k = (k+1)\%N$

Variant 3: $a_k = \left(k + \frac{\text{cache line size}}{\text{sizeof(int)}}\right)\%N$

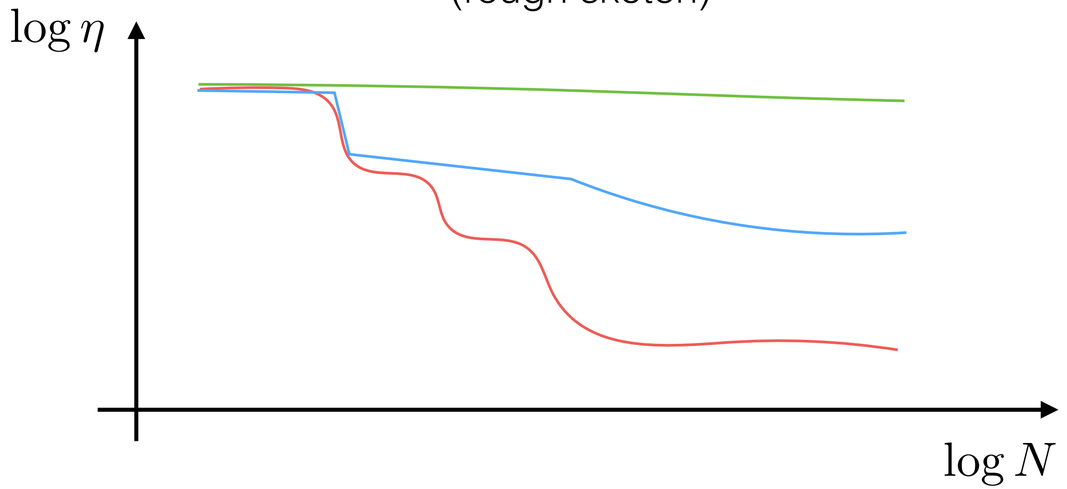
Question 1: Task

Variant 1: $a_k =$ a random one-cycle permutation

Variant 2: $a_k = (k+1)\%N$

Variant 3: $a_k = \left(k + \frac{\text{cache line size}}{\text{sizeof(int)}}\right)\%N$

(rough sketch)

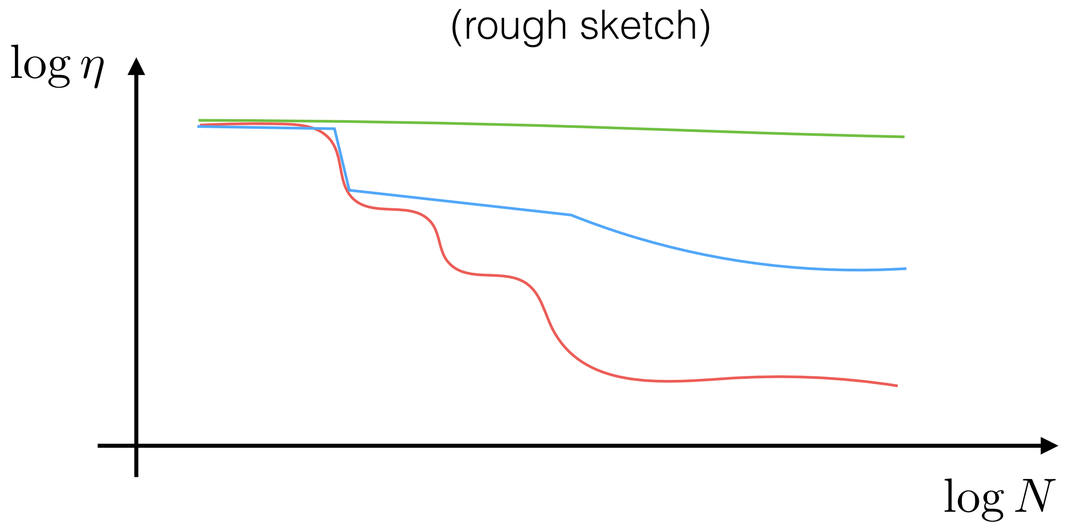


Question 1: Task

Variant 1: $a_k =$ a random one-cycle permutation

Variant 2: $a_k = (k+1)\%N$

Variant 3: $a_k = \left(k + \frac{\text{cache line size}}{\text{sizeof(int)}}\right)\%N$



Old Grandma's saying: *Always access your memory sequentially!*

Overview

- Question 1:
  - Memory access speed vs. array size?
  - (demonstrating L1, L2 and L3 **cache size**)

How you should behave regarding memory access.

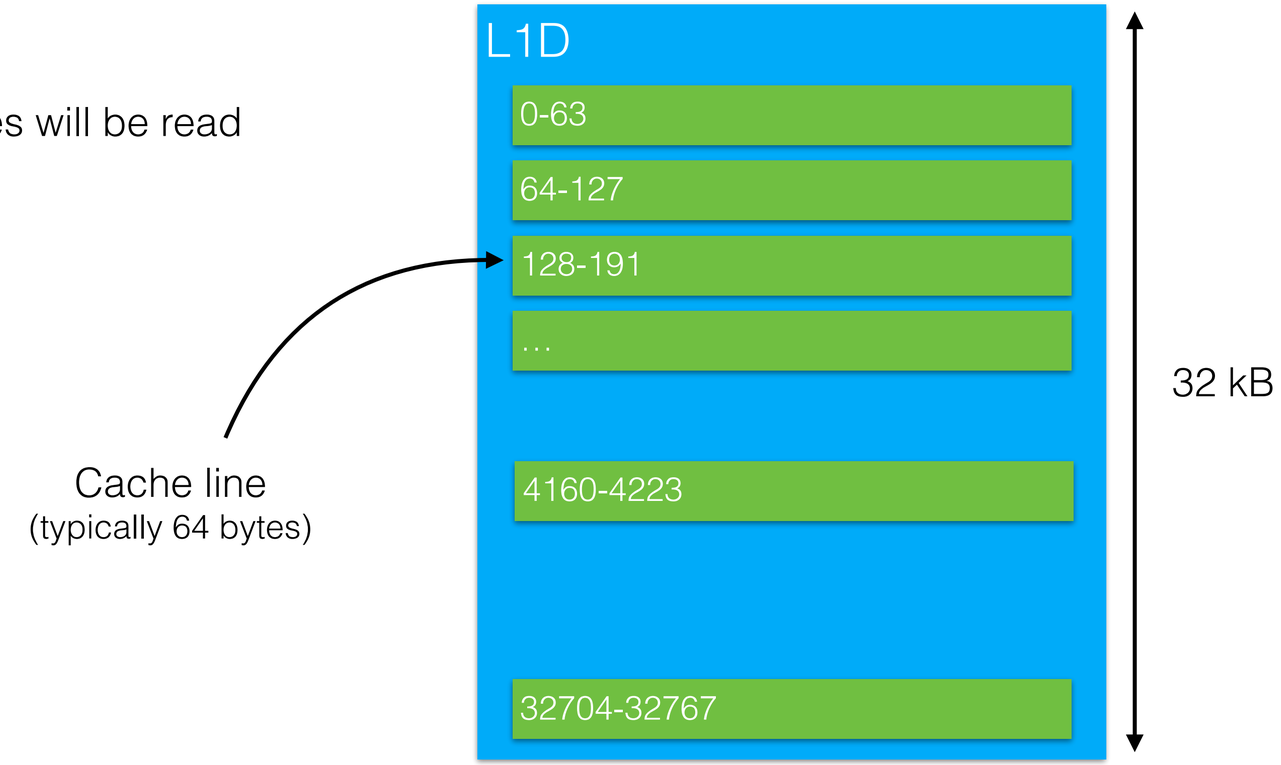
- Question 2:
  - Memory access speed vs. # of simultaneously used arrays?
  - (demonstrating **cache associativity**)

- Question 3:
  - N-thread lock
  - (demonstrating **mutual exclusion**)

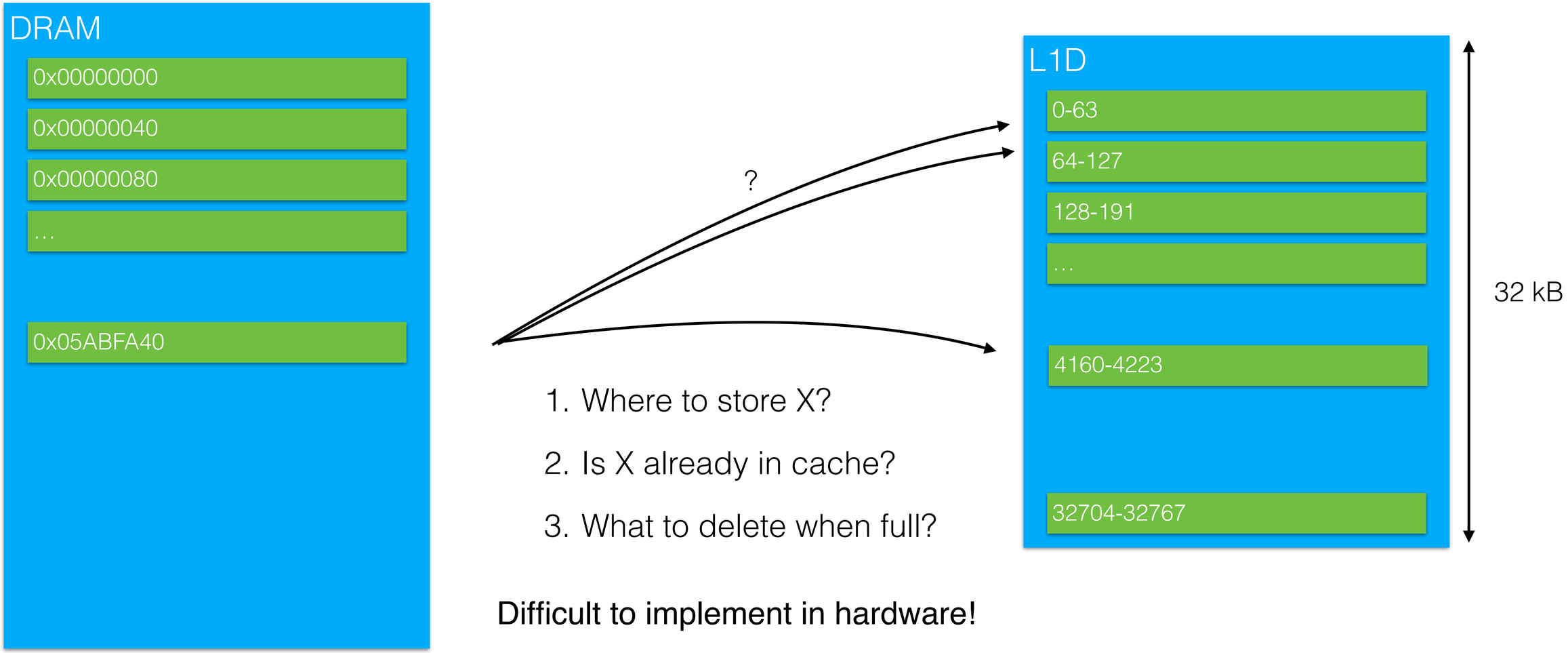
A rough idea on how locks work. Visualization of mutual exclusion.

Cache Line

- Data stored in chunks of 64 bytes
- If a single double (8 bytes) is accessed, 56 more bytes will be read
  - 7 more doubles for free!



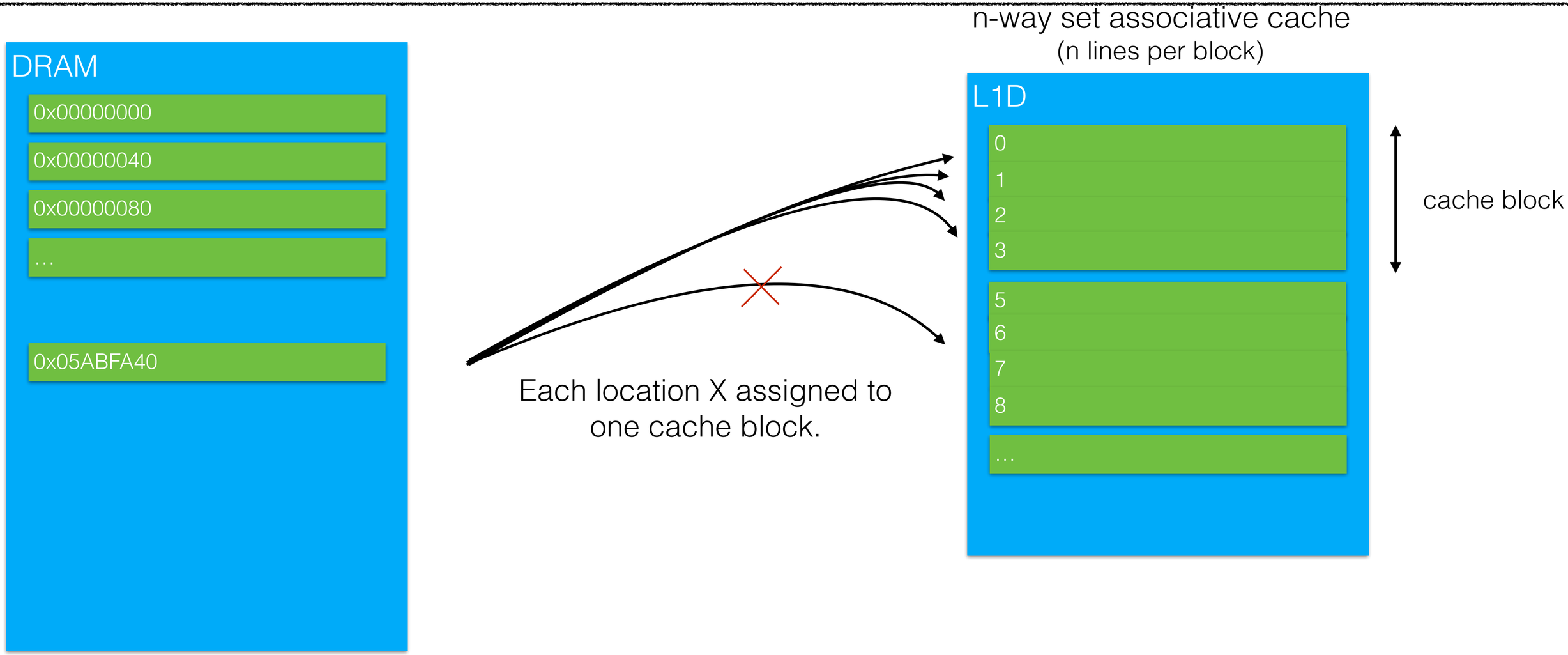
Cache Associativity



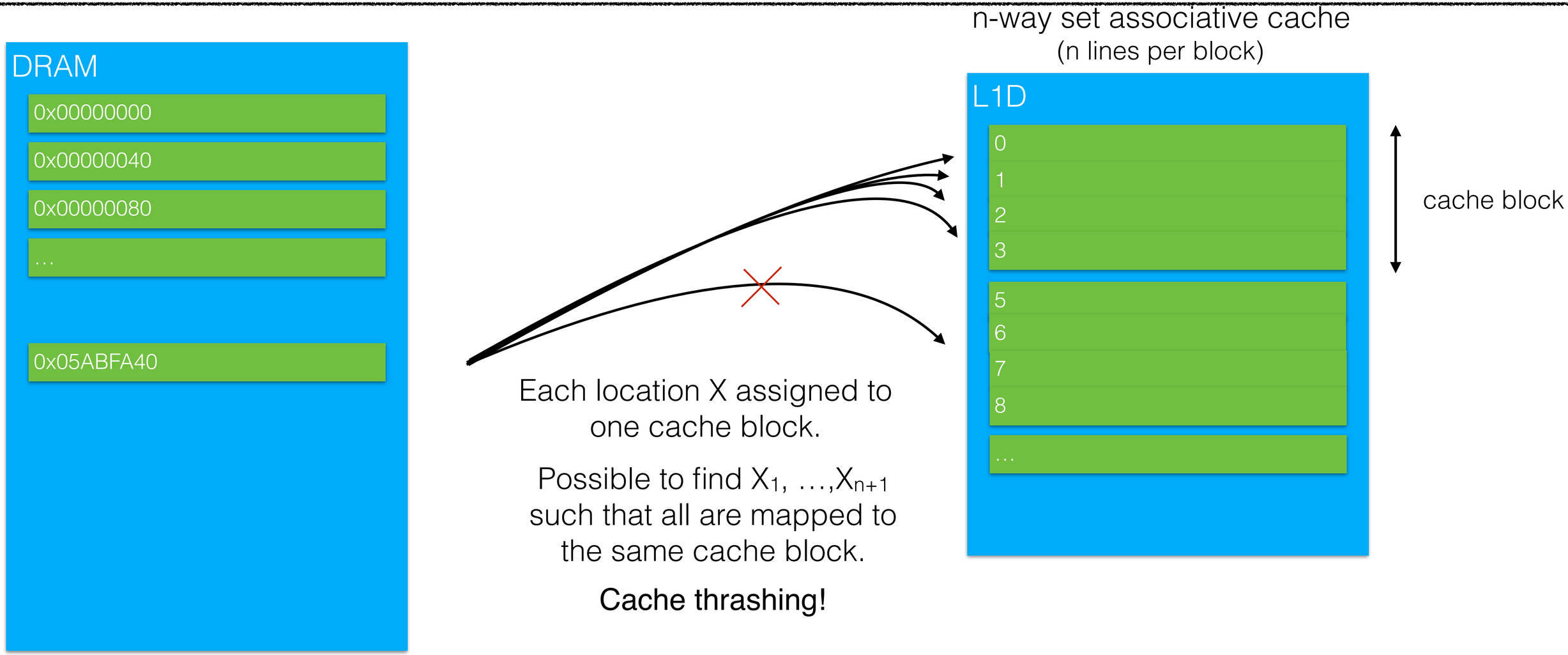
1. Where to store X?
2. Is X already in cache?
3. What to delete when full?

Difficult to implement in hardware!

Cache Associativity (cont.)



Cache Associativity (cont.)

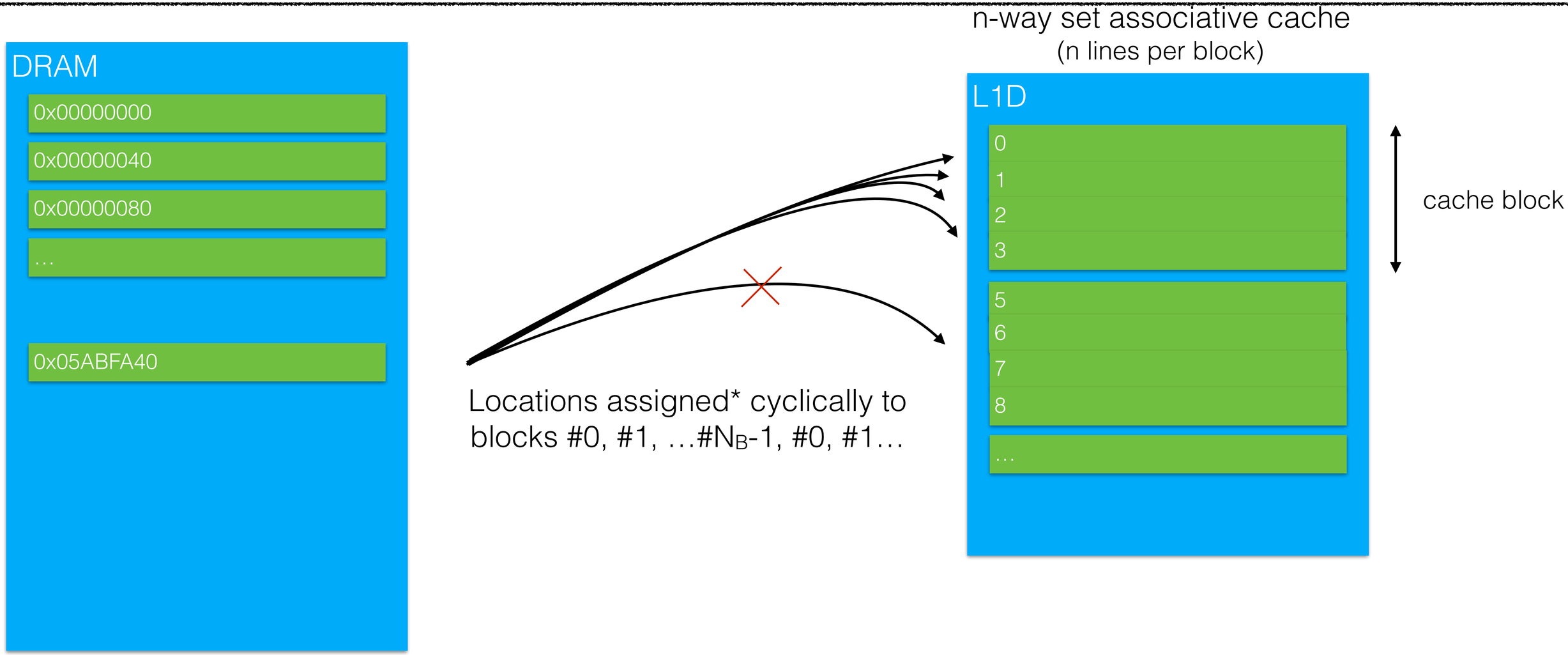


Each location X assigned to one cache block.

Possible to find $X_1, \ldots, X_{n+1}$ such that all are mapped to the same cache block.

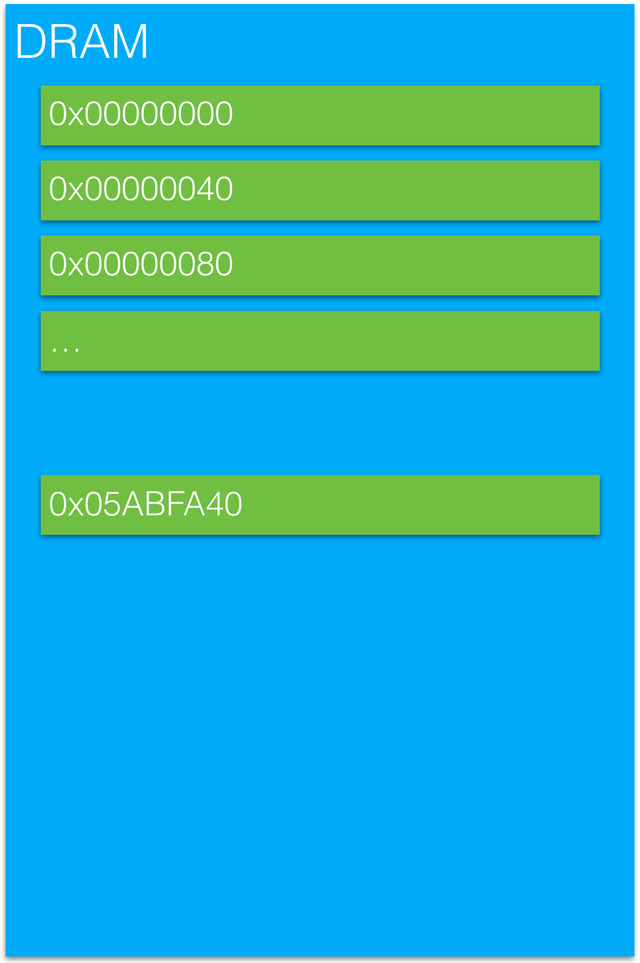
Cache thrashing!

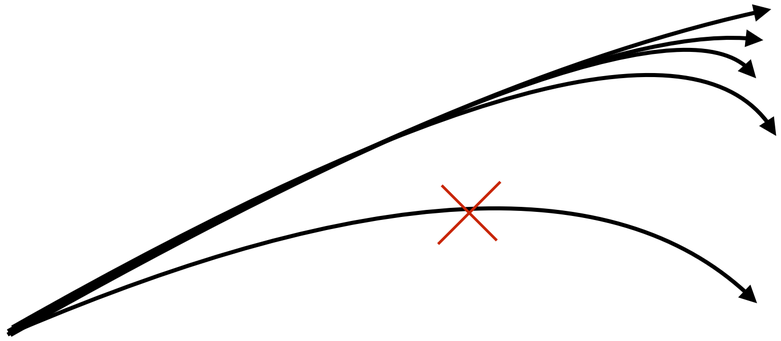
Cache Associativity (cont.)



*Depends on the architecture

Cache Associativity (cont.)

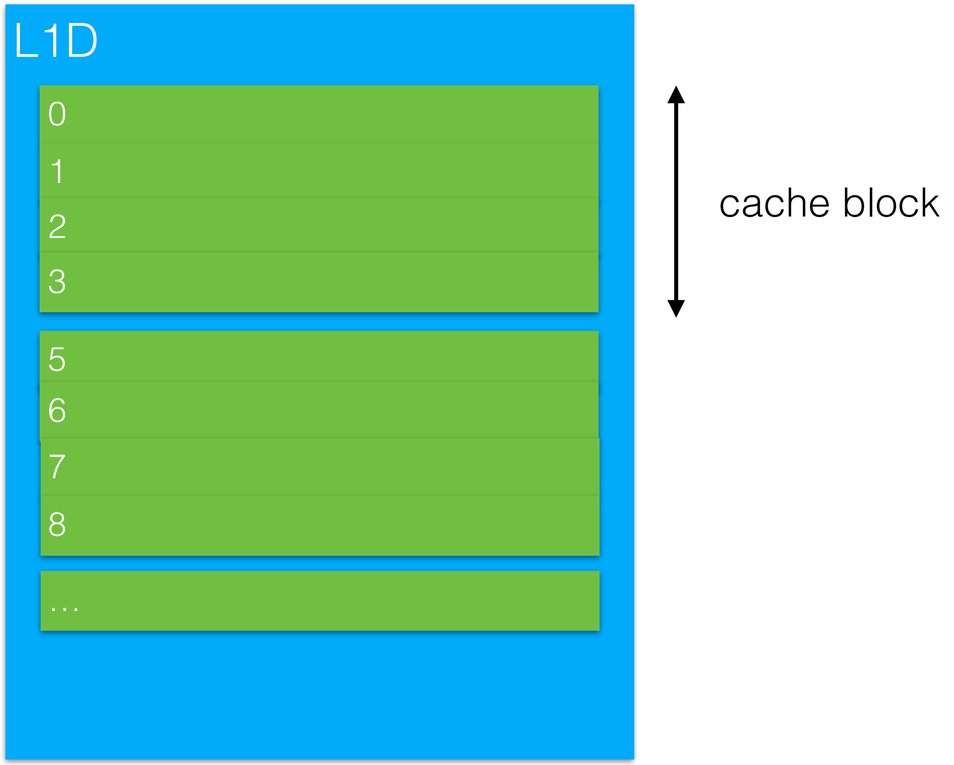




Locations assigned* cyclically to blocks #0, #1, …#N_B-1, #0, #1…

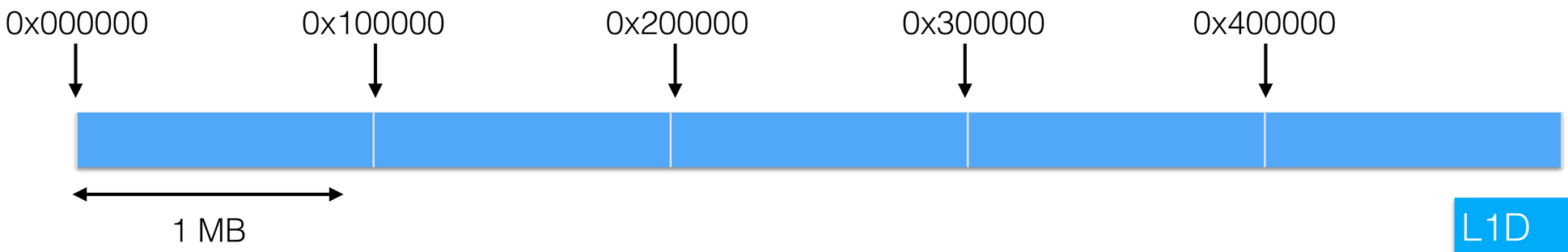
Assume X_1 and X_2 are in the same block if $(X_1 - X_2)$ % 2^L == 0 for some large L.

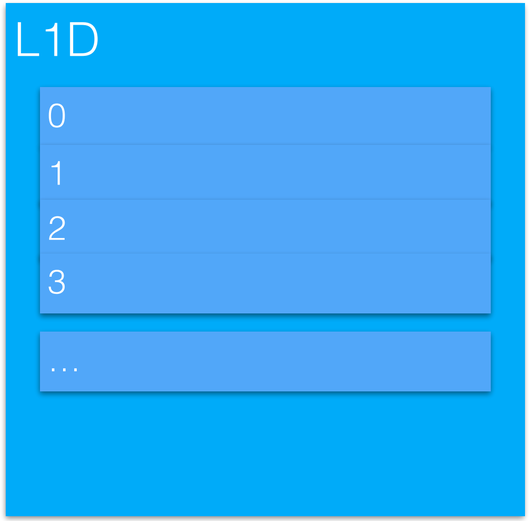
n-way set associative cache
(n lines per block)



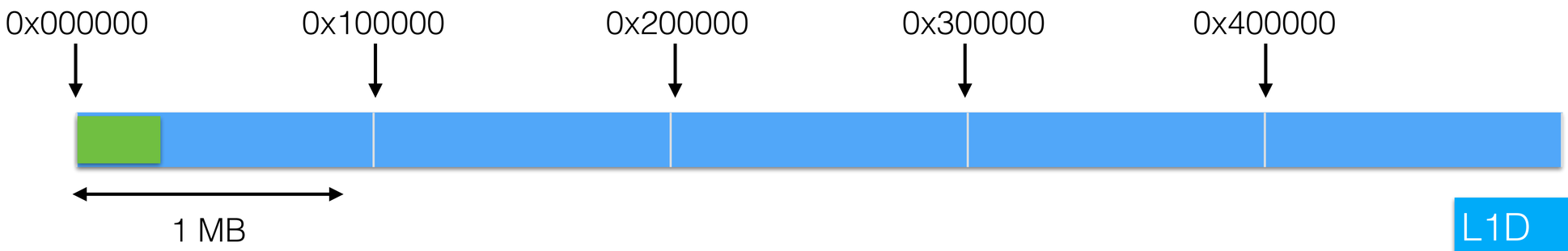
*Depends on the architecture

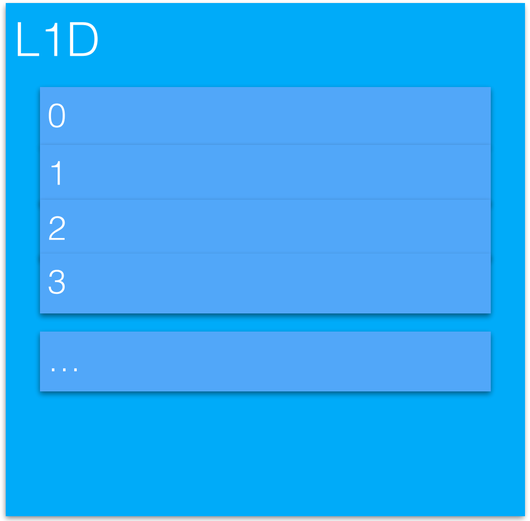
Question 2: Memory access speed vs. # of simultaneously used arrays?



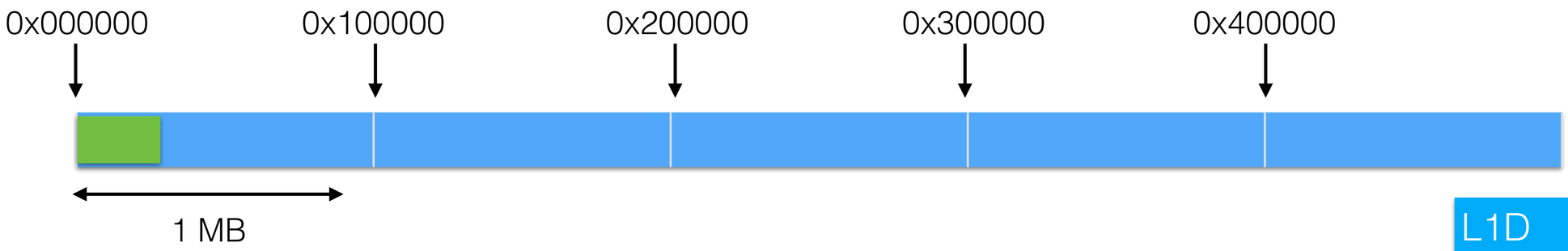


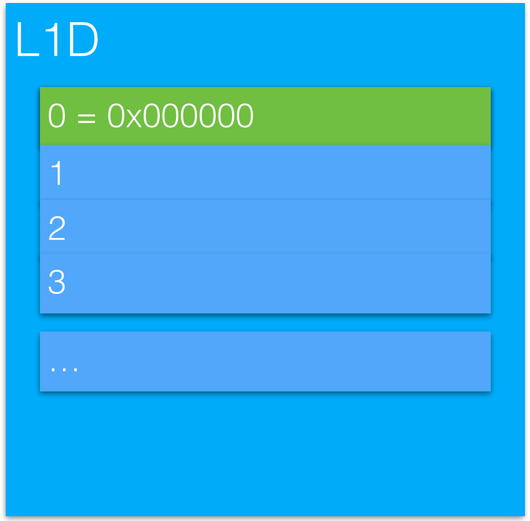
Question 2: Memory access speed vs. # of simultaneously used arrays?



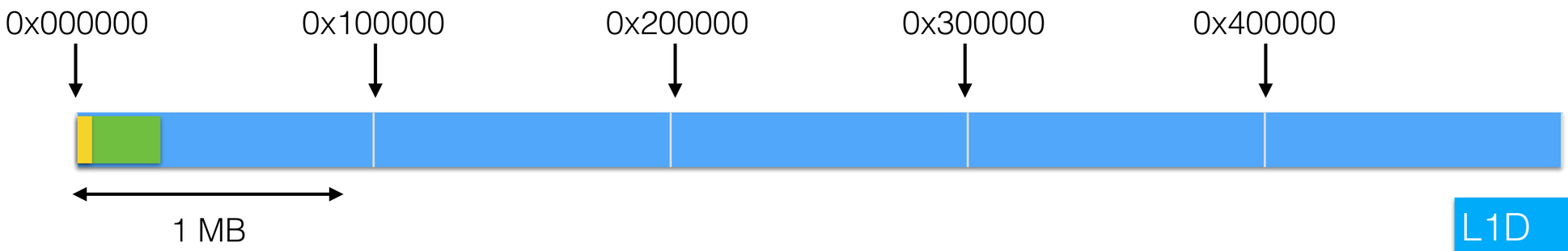


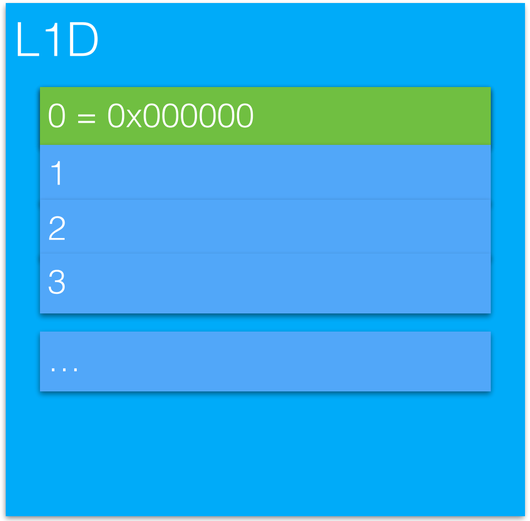
Question 2: Memory access speed vs. # of simultaneously used arrays?



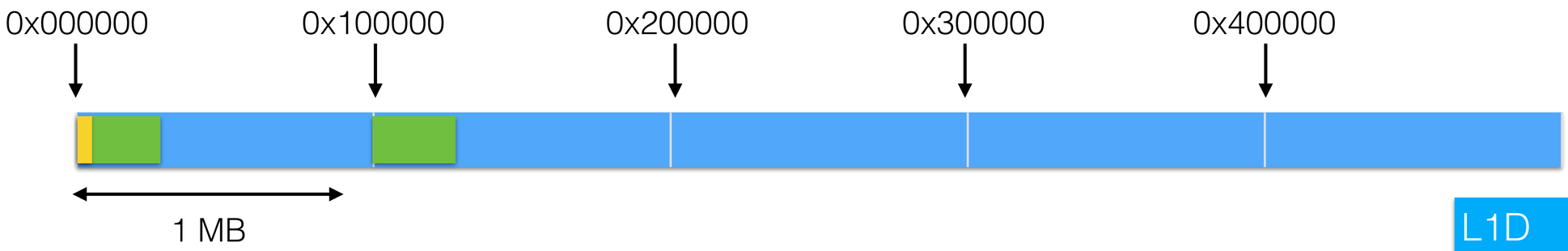


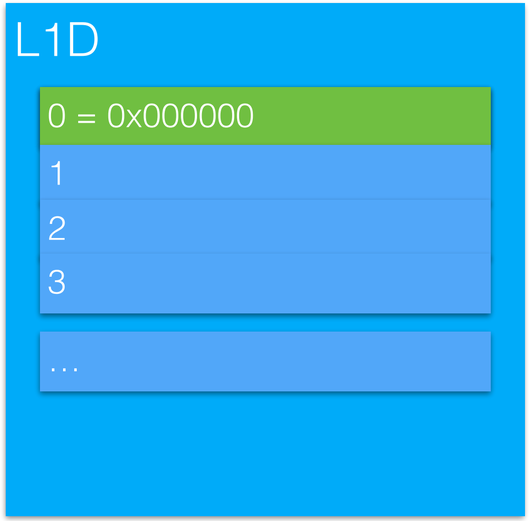
Question 2: Memory access speed vs. # of simultaneously used arrays?



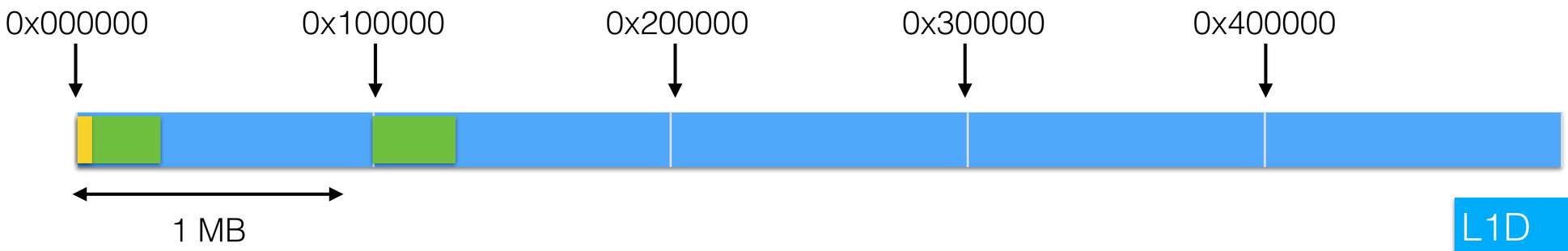


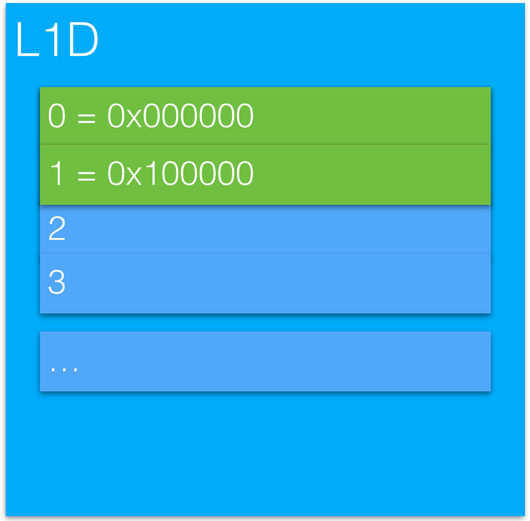
Question 2: Memory access speed vs. # of simultaneously used arrays?



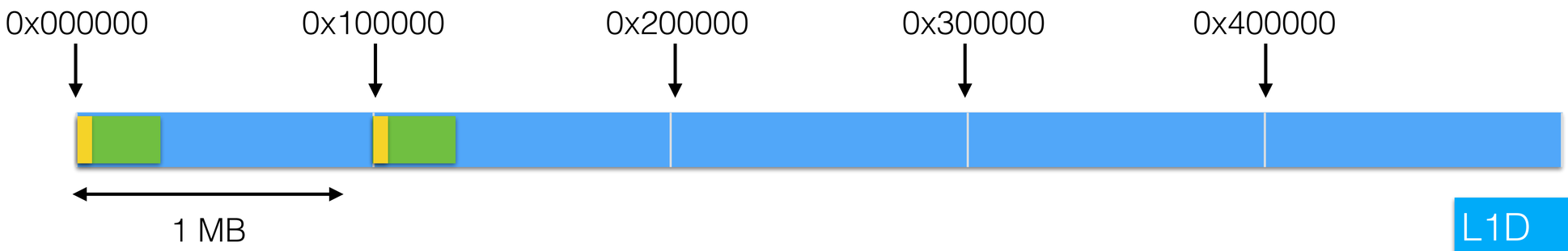


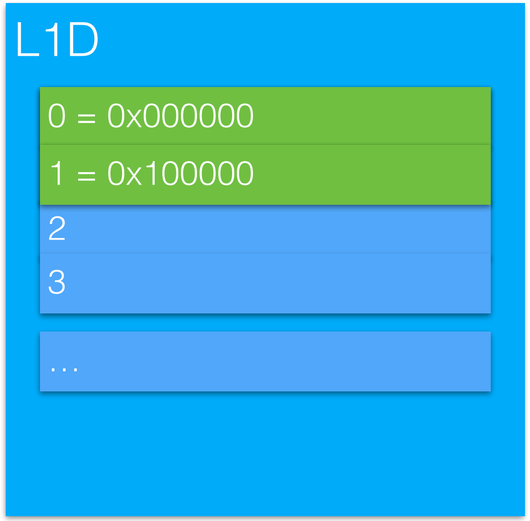
Question 2: Memory access speed vs. # of simultaneously used arrays?



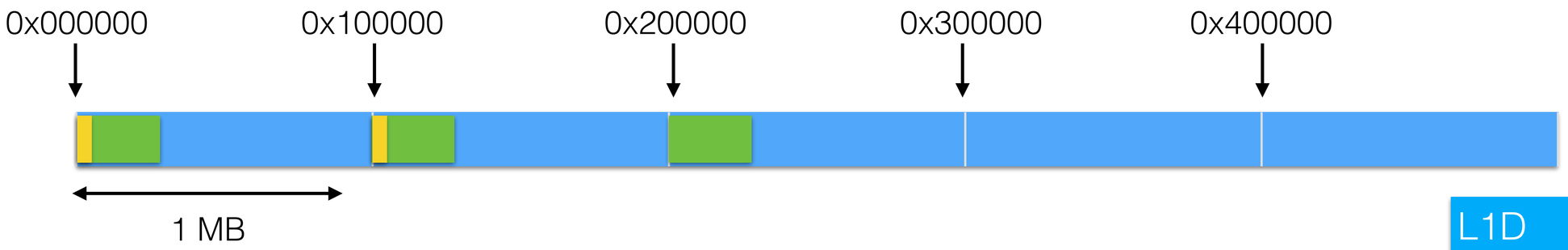


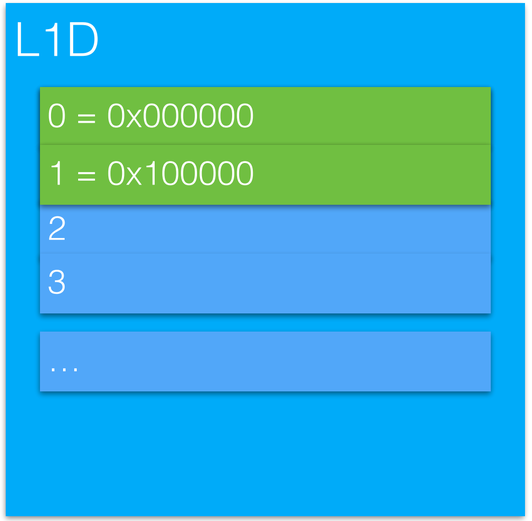
Question 2: Memory access speed vs. # of simultaneously used arrays?



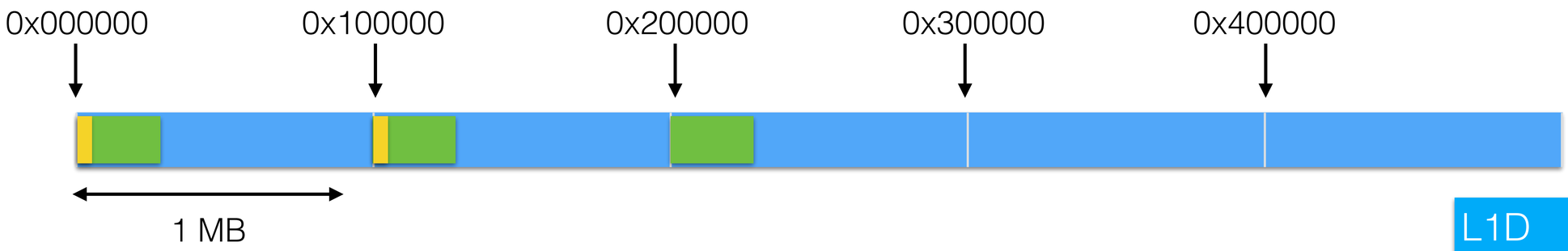


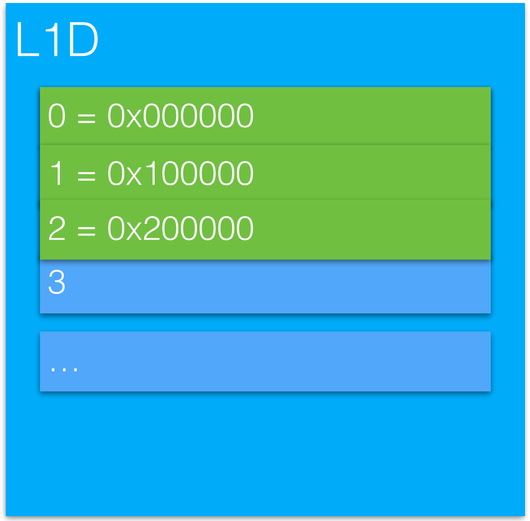
Question 2: Memory access speed vs. # of simultaneously used arrays?



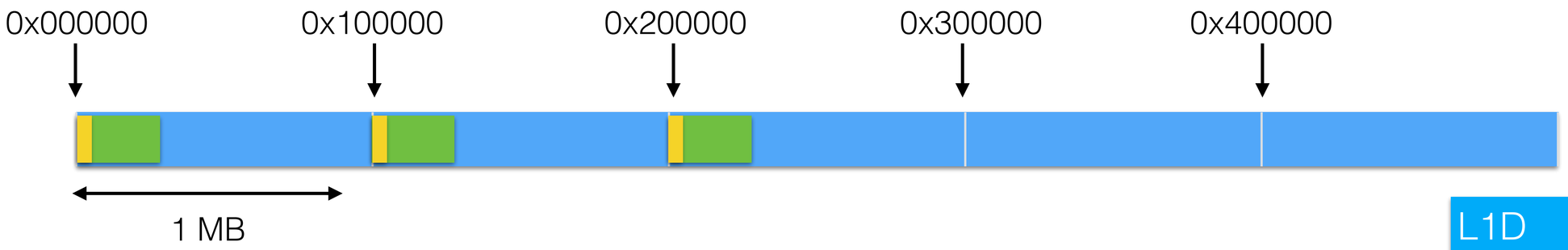


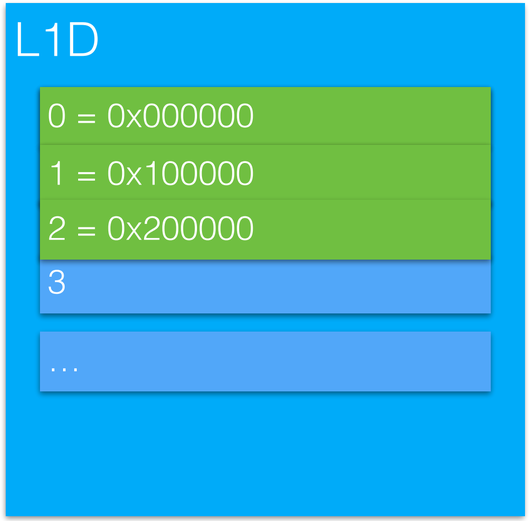
Question 2: Memory access speed vs. # of simultaneously used arrays?



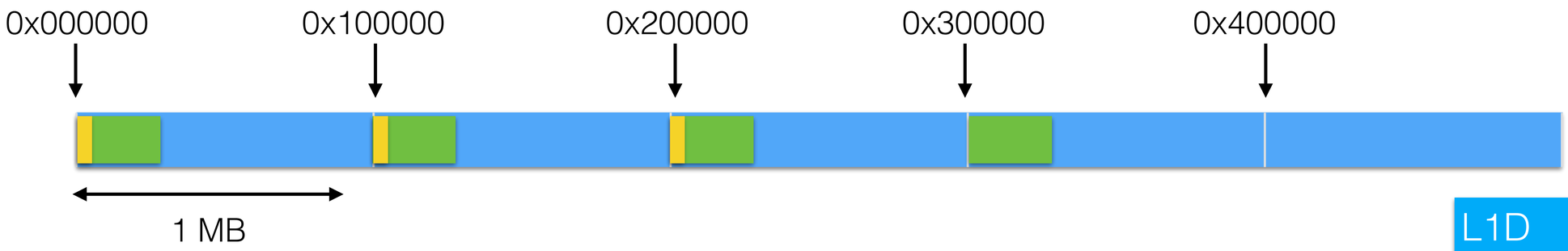


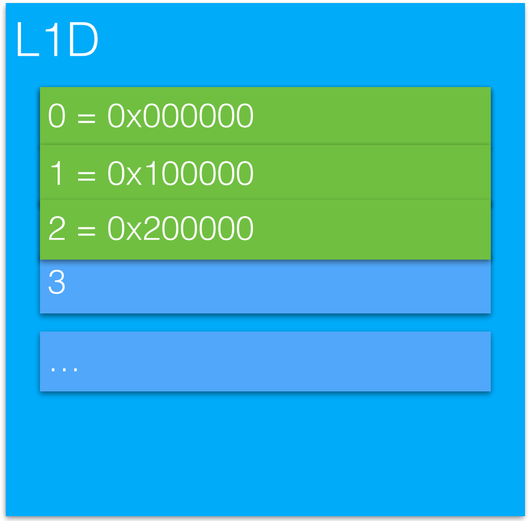
Question 2: Memory access speed vs. # of simultaneously used arrays?



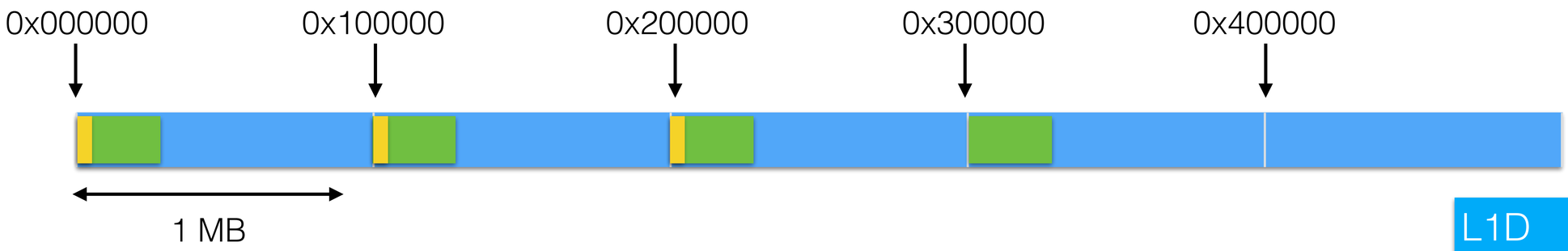


Question 2: Memory access speed vs. # of simultaneously used arrays?





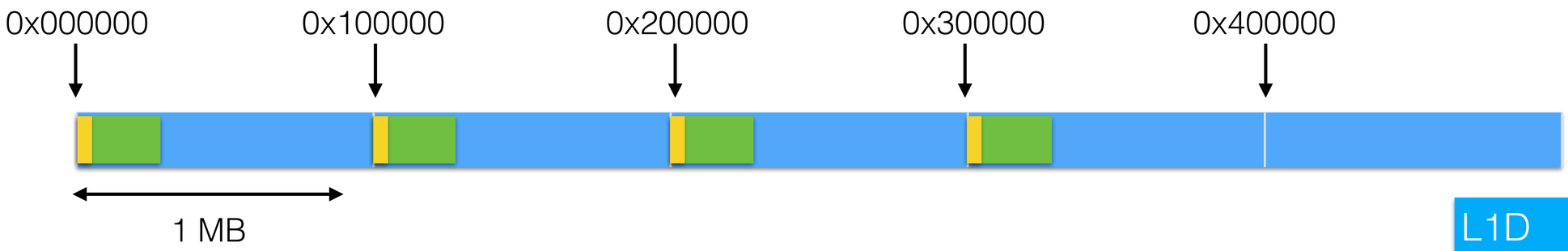
Question 2: Memory access speed vs. # of simultaneously used arrays?



L1D

- 0 = 0x000000
- 1 = 0x100000
- 2 = 0x200000
- 3 = 0x300000
- …

Question 2: Memory access speed vs. # of simultaneously used arrays?



L1D

0 = 0x000000

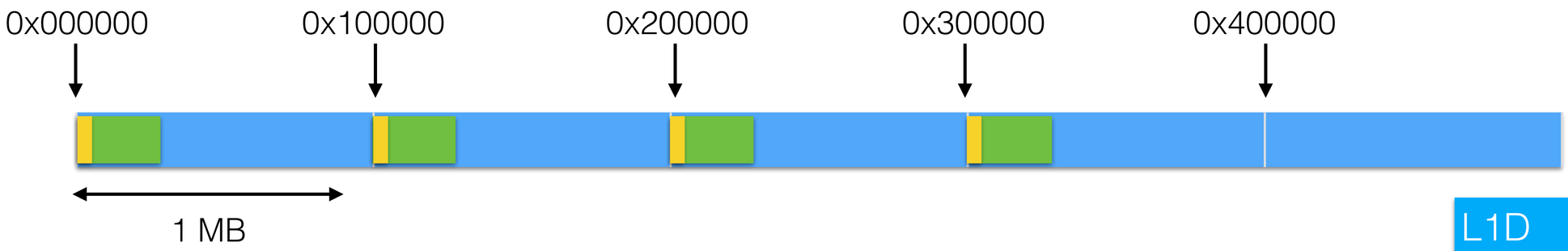
1 = 0x100000

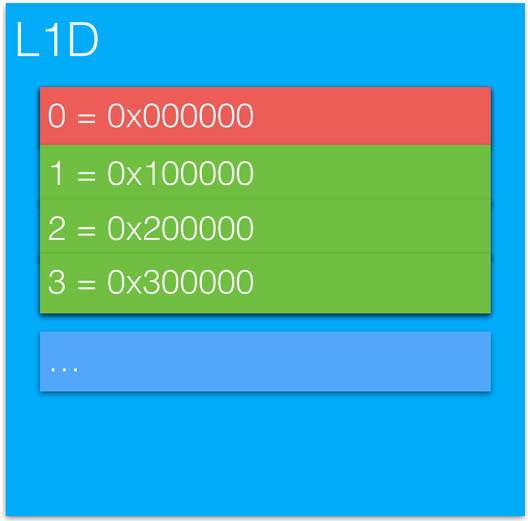
2 = 0x200000

3 = 0x300000

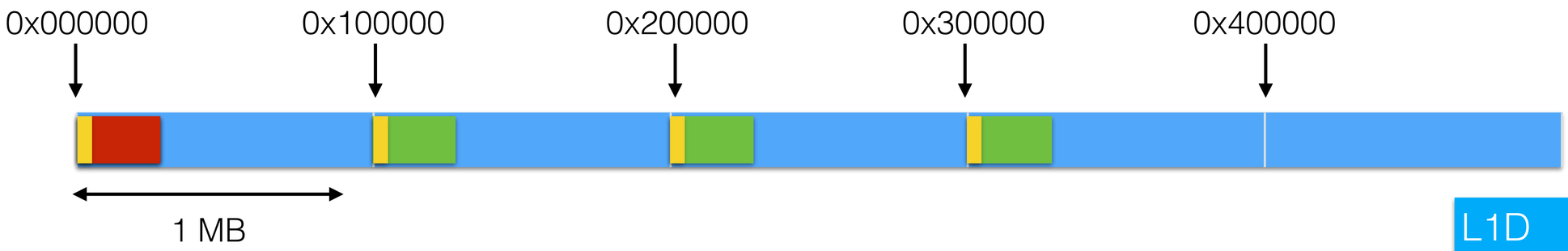
…

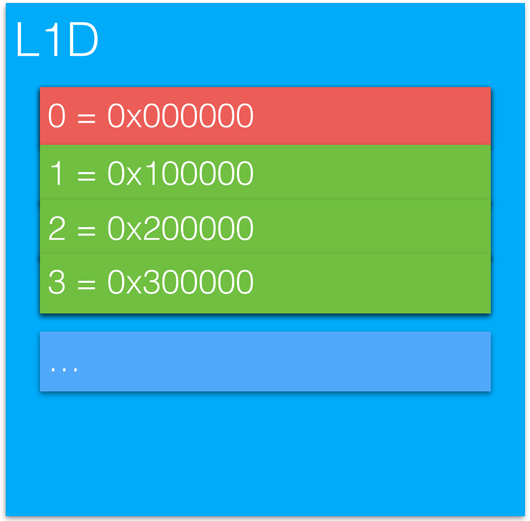
Question 2: Memory access speed vs. # of simultaneously used arrays?



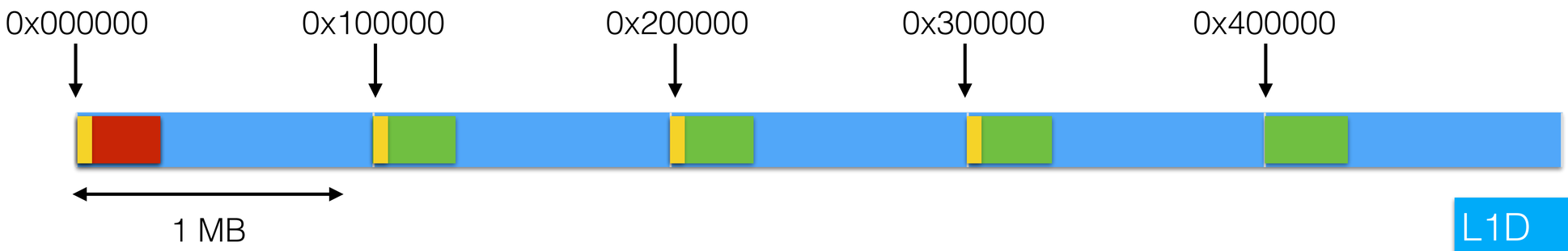


Question 2: Memory access speed vs. # of simultaneously used arrays?





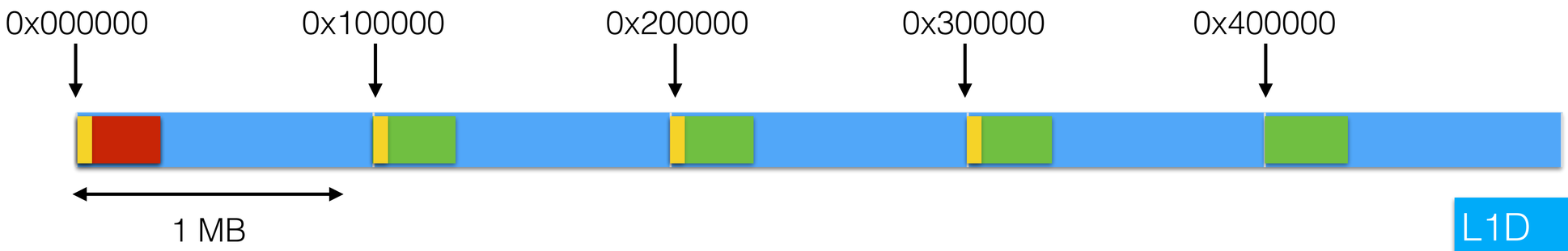
Question 2: Memory access speed vs. # of simultaneously used arrays?



L1D

- 0 = 0x000000
- 1 = 0x100000
- 2 = 0x200000
- 3 = 0x300000
- …

Question 2: Memory access speed vs. # of simultaneously used arrays?



L1D

0 = 0x400000

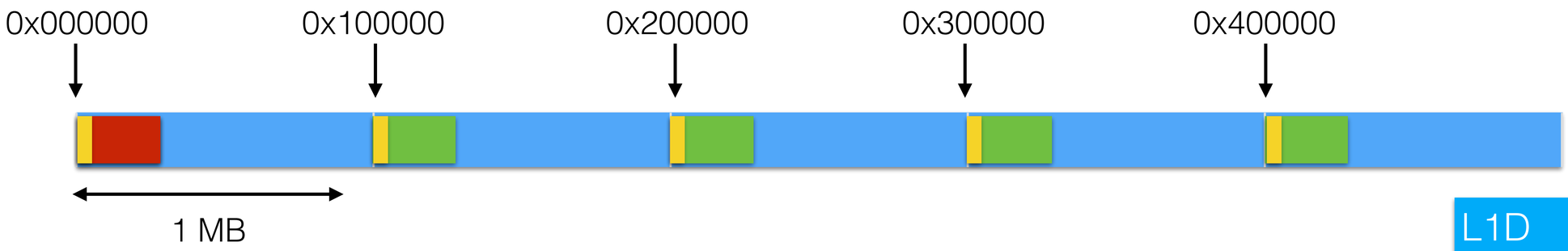
1 = 0x100000

2 = 0x200000

3 = 0x300000

…

Question 2: Memory access speed vs. # of simultaneously used arrays?



L1D

0 = 0x400000

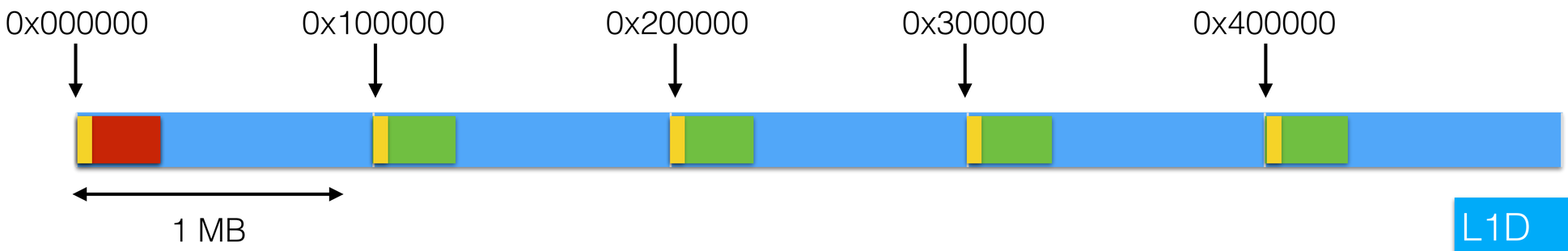
1 = 0x100000

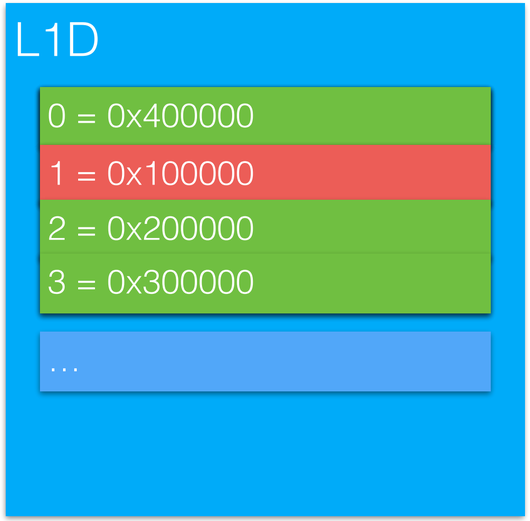
2 = 0x200000

3 = 0x300000

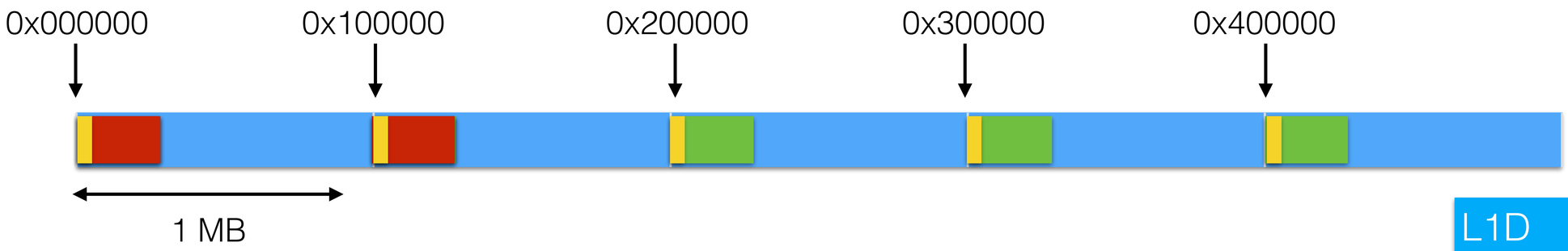
…

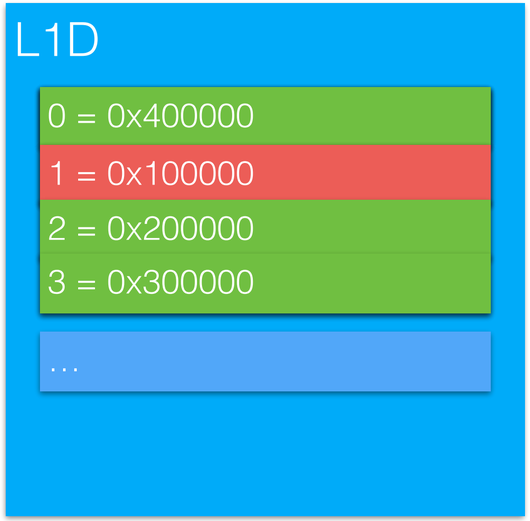
Question 2: Memory access speed vs. # of simultaneously used arrays?



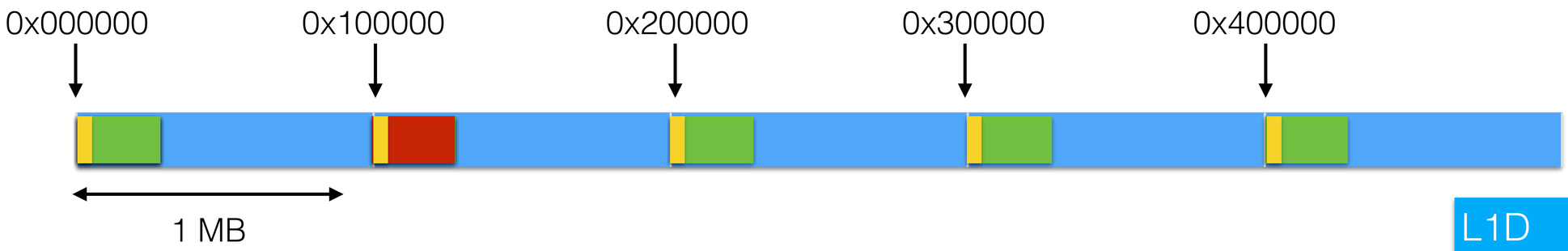


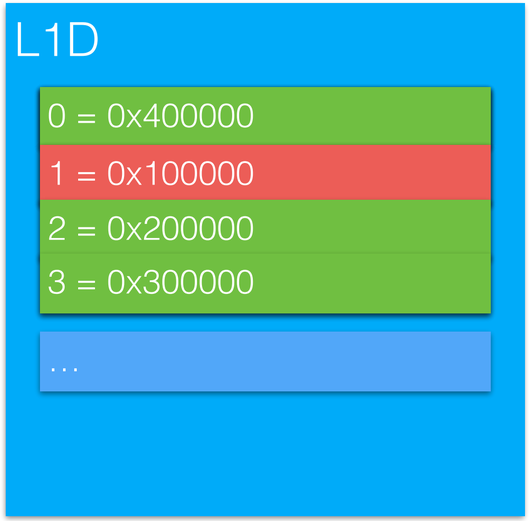
Question 2: Memory access speed vs. # of simultaneously used arrays?



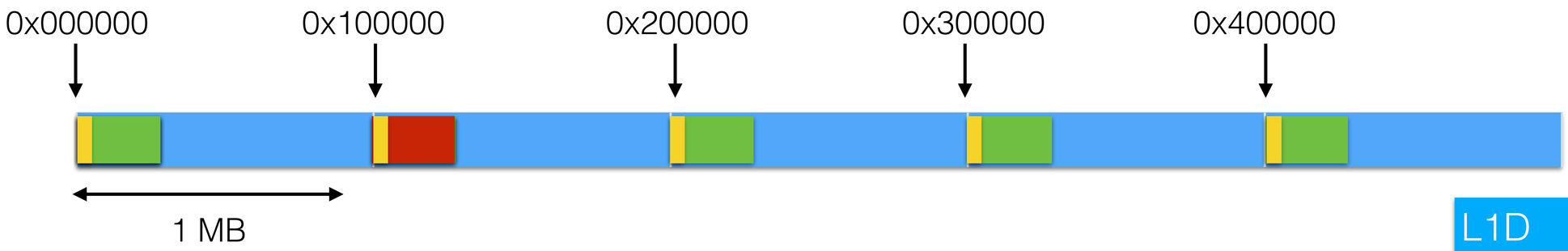


Question 2: Memory access speed vs. # of simultaneously used arrays?





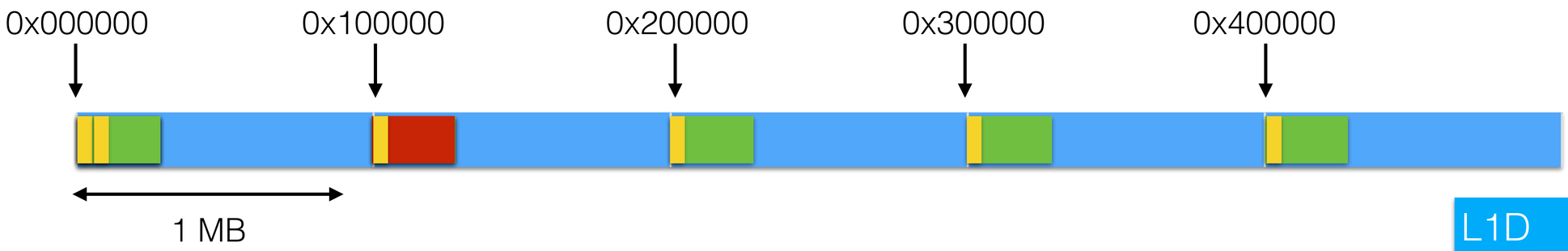
Question 2: Memory access speed vs. # of simultaneously used arrays?



L1D

- 0 = 0x400000
- 0 = 0x000000
- 2 = 0x200000
- 3 = 0x300000
- …

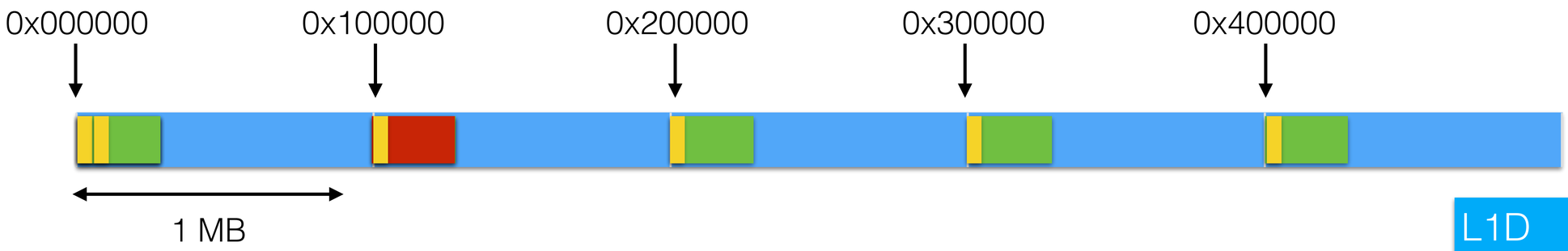
Question 2: Memory access speed vs. # of simultaneously used arrays?



L1D

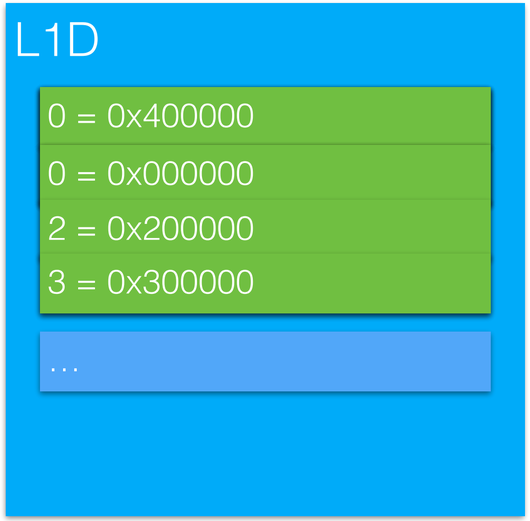
- 0 = 0x400000
- 0 = 0x000000
- 2 = 0x200000
- 3 = 0x300000
- …

Question 2: Memory access speed vs. # of simultaneously used arrays?

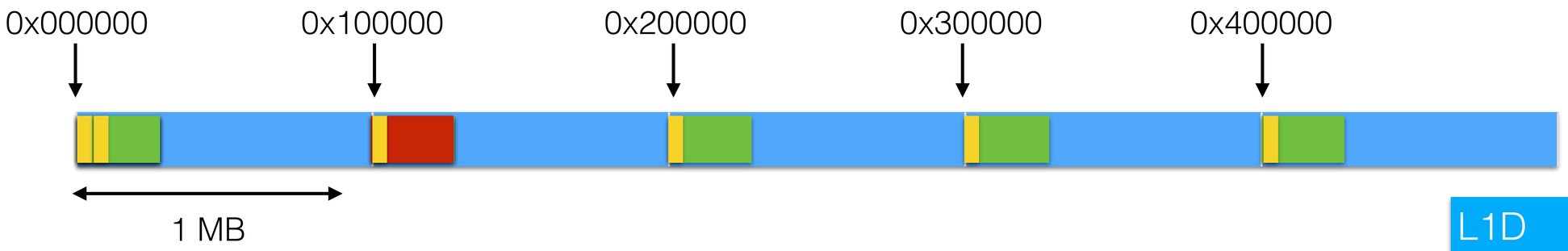


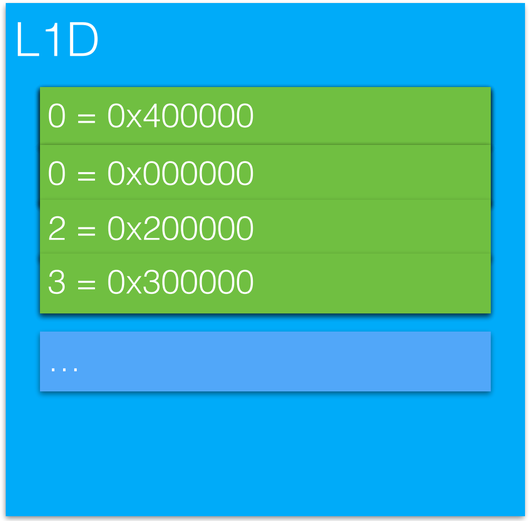
Cache thrashing!

Re-reading the same memory locations all over again.



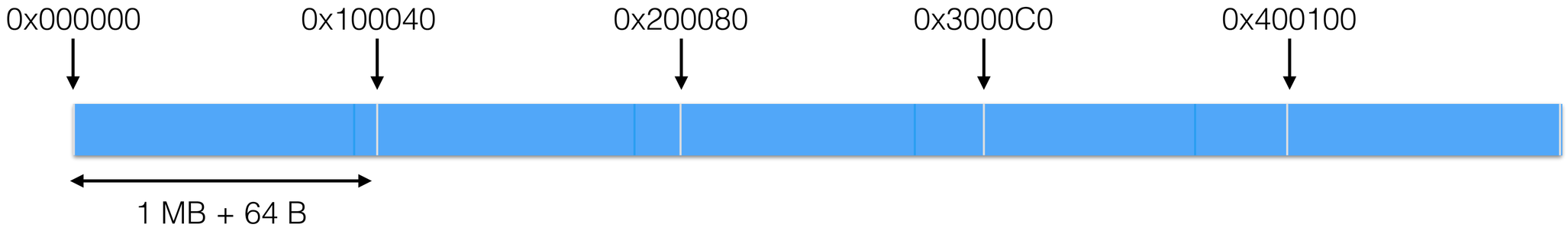
Question 2: Memory access speed vs. # of simultaneously used arrays?





Cache thrashing!

Re-reading the same memory locations all over again.



Question 2: Task

Variant 1: Access K arrays of $N=2^{20}$ doubles (8 MB).

Variant 2: Access K arrays of $N=2^{20}+8$ doubles (8 MB + 64 B).

K = 1, 2, …, 40
access = increase by a constant

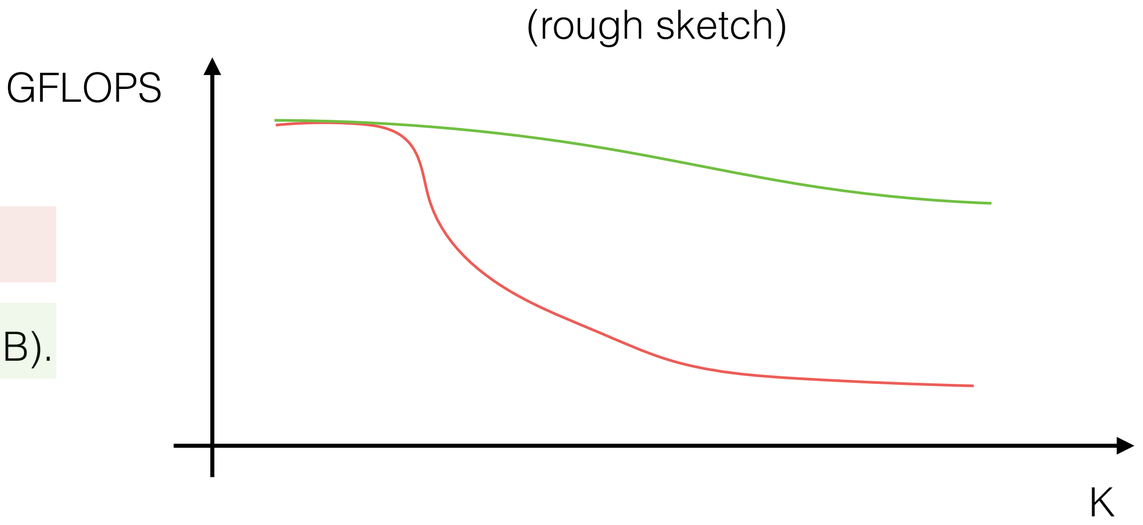
Implementation: Allocate one large array of size N*K
and treat as K different arrays.

Question 2: Task

Variant 1: Access K arrays of $N=2^{20}$ doubles (8 MB).

Variant 2: Access K arrays of $N=2^{20}+8$ doubles (8 MB + 64 B).

K = 1, 2, …, 40
access = increase by a constant



Implementation: Allocate one large array of size N*K and treat as K different arrays.

Overview

- Question 1:
  - Memory access speed vs. array size?
  - (demonstrating L1, L2 and L3 **cache size**)

How you should behave regarding memory access.

- Question 2:
  - Memory access speed vs. # of simultaneously used arrays?
  - (demonstrating **cache associativity**)

- Question 3:
  - N-thread lock
  - (demonstrating **mutual exclusion**)

A rough idea on how locks work.
Visualization of mutual exclusion.

Locks in General

- **Lock**: A mechanism that guarantees that at most one thread is inside a **critical region** at any given point of time

Locks in General

- **Lock**: A mechanism that guarantees that at most one thread is inside a **critical region** at any given point of time

```
void func() {
    FILE *f = fopen(...);

    #pragma omp parallel
    {
        // Many threads running this.
        some_computation();



        // Dangerous!
        fwrite(...);


        some_computation();
    }

    fclose(f);
}
```

Locks in General

- **Lock**: A mechanism that guarantees that at most one thread is inside a **critical region** at any given point of time

```
void func() {
    FILE *f = fopen(...);

    #pragma omp parallel
    {
        // Many threads running this.
        some_computation();


        // Dangerous!
        fwrite(...);


        some_computation();
    }

    fclose(f);
}
```

Critical region!

Locks in General

- **Lock**: A mechanism that guarantees that at most one thread is inside a **critical region** at any given point of time

```
void func() {
    FILE *f = fopen(...);

    #pragma omp parallel
    {
        // Many threads running this.
        some_computation();

        #pragma omp critical
        {
            // Good!
            fwrite(...);
        }

        some_computation();
    }

    fclose(f);
}
```

Critical region!

Locks in General

- **Lock**: A mechanism that guarantees that at most one thread is inside a **critical region** at any given point of time

```
void func() {
    FILE *f = fopen(...);

    #pragma omp parallel
    {
        // Many threads running this.
        some_computation();

        #pragma omp critical
        {
            // Good!
            fwrite(...);
        }

        some_computation();
    }

    fclose(f);
}
```

How does this work?

Critical region!

Locks in General

- **Lock**: A mechanism that guarantees that at most one thread is inside a **critical region** at any given point of time

```
void func() {
    FILE *f = fopen(...);

    #pragma omp parallel
    {
        // Many threads running this.
        some_computation();

        #pragma omp critical
        {
            // Good!
            fwrite(...);
        }

        some_computation();
    }

    fclose(f);
}
```

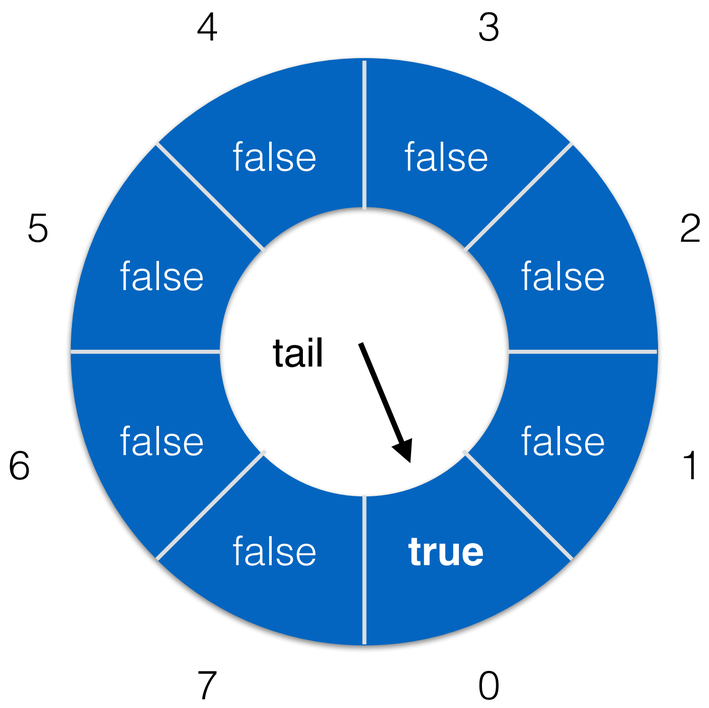
How does this work?

Critical region!

```
Lock L;

#pragma omp parallel
{
    ...

    L.lock(tid);
    fwrite(...);
    L.unlock(tid);

    ...
}
```

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	-1	-1	-1	-1	-1	-1

Lock rule: 1. Remember at which slot is the tail pointing to, and spin it by 1 (fetch_and_add atomically).

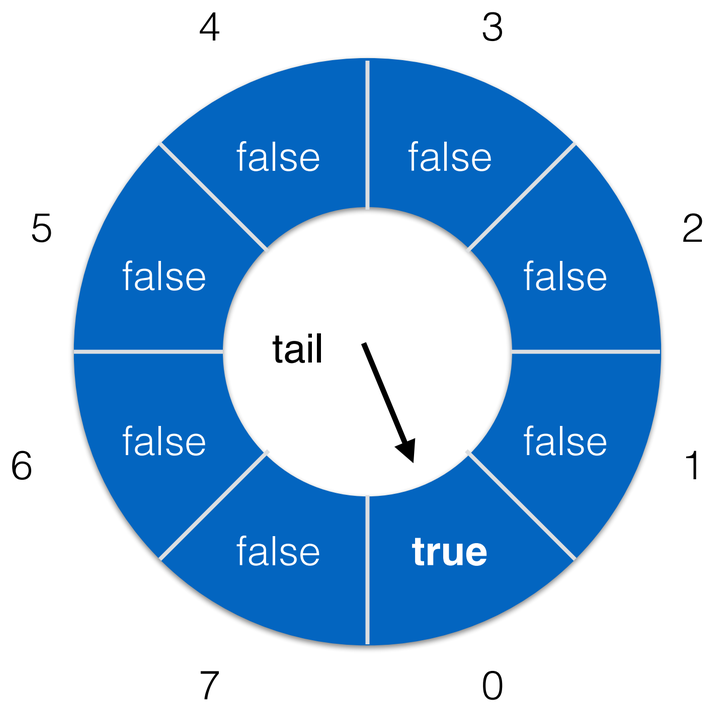
2. Wait until that flag becomes true.

Unlock rule: 1. Set my flag to false.

2. Update next flag to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	-1	-1	-1	-1	-1	-1

Lock rule:

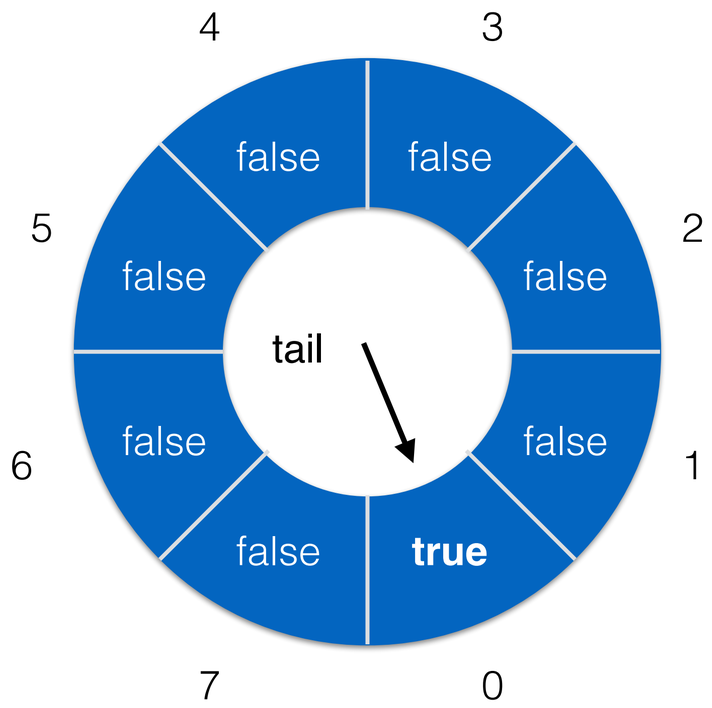
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	-1	-1	-1	-1	-1	-1

1. lock(2)

Lock rule:

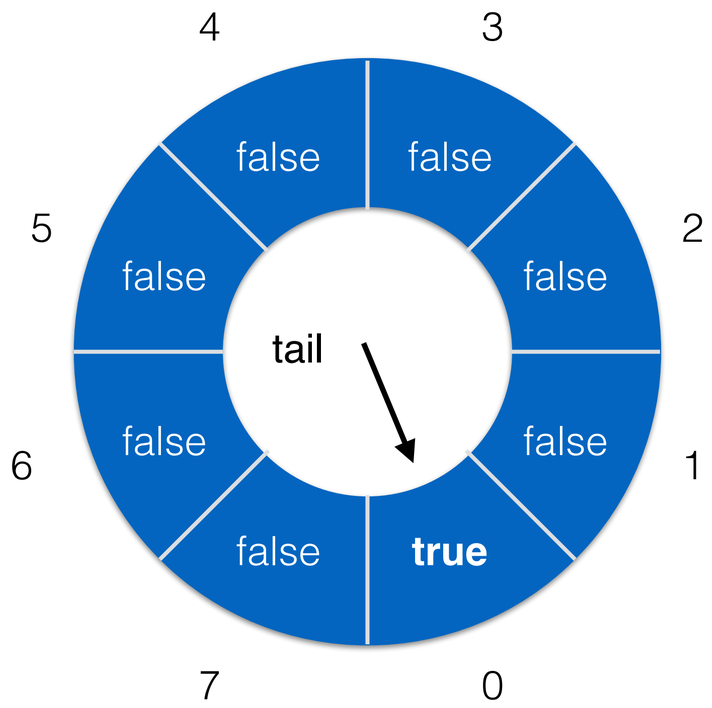
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	-1	-1	-1

1. lock(2)

Lock rule:

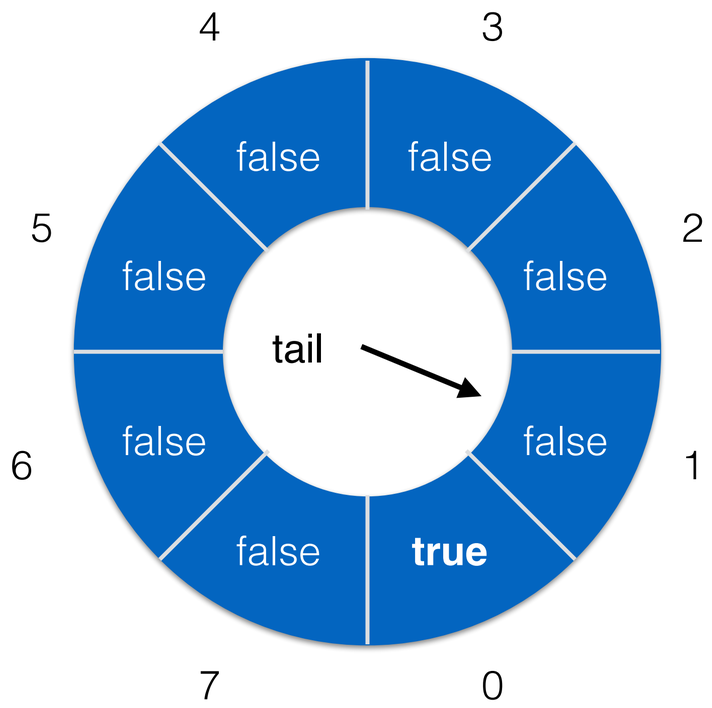
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	-1	-1	-1

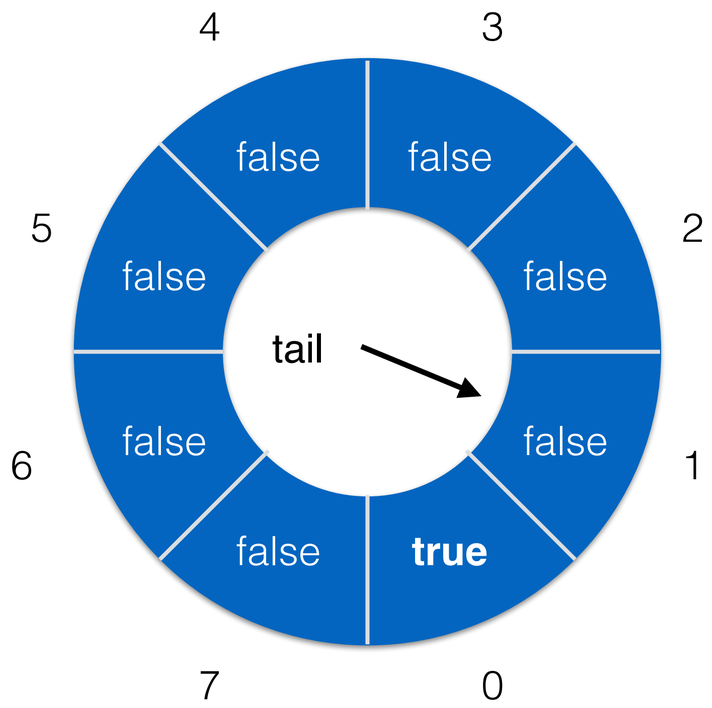
1. lock(2)

Lock rule:
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:
1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	-1	-1	-1

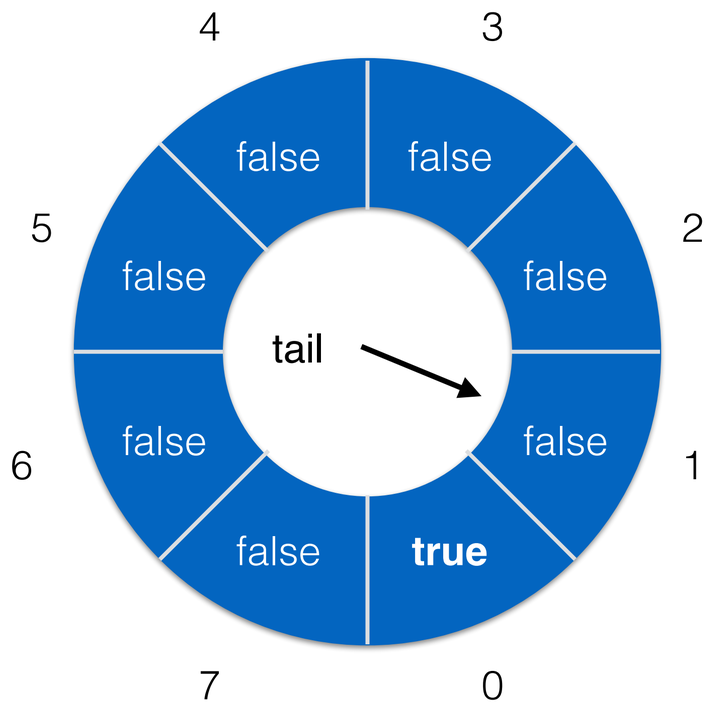
1. **lock**(2)
2. (2 in critical)

Lock rule:
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:
1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	-1	-1	-1

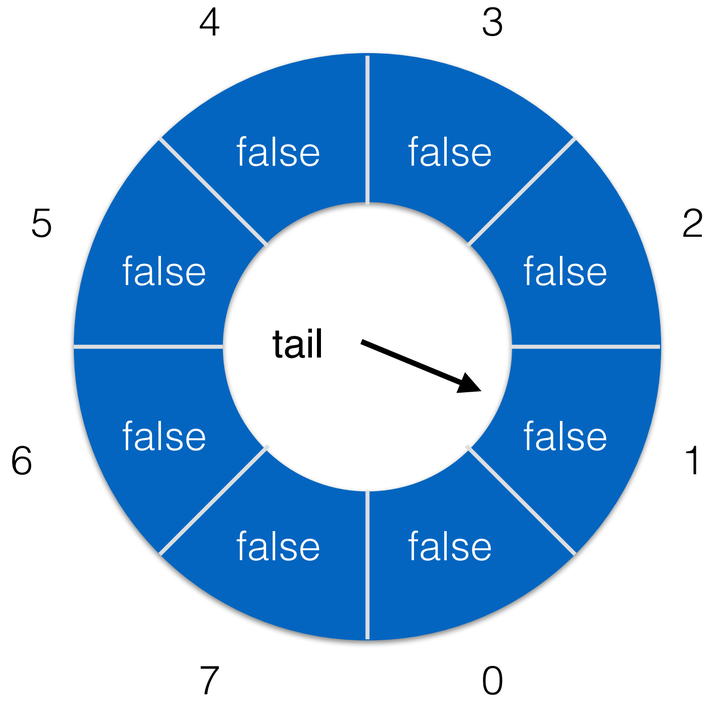
1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)

Lock rule:
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:
1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	-1	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)

Lock rule:

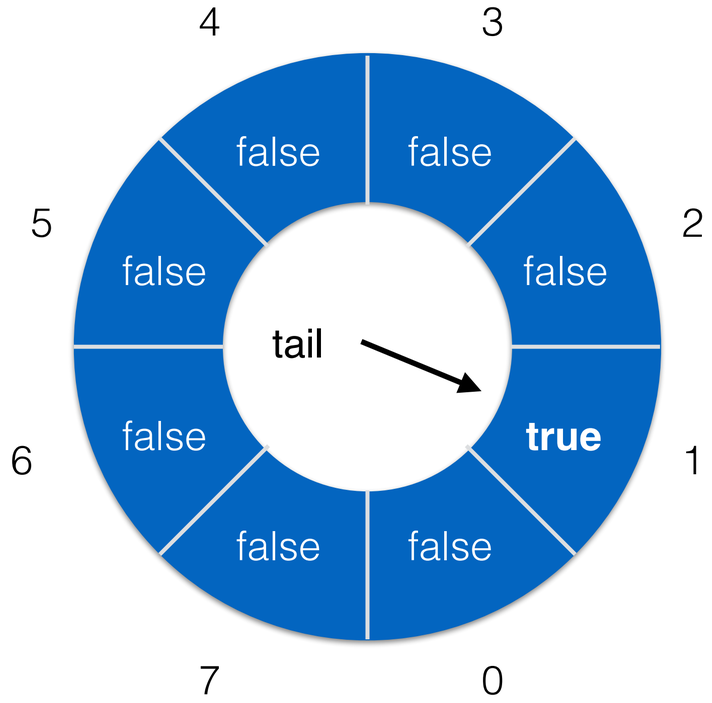
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	-1	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)

Lock rule:

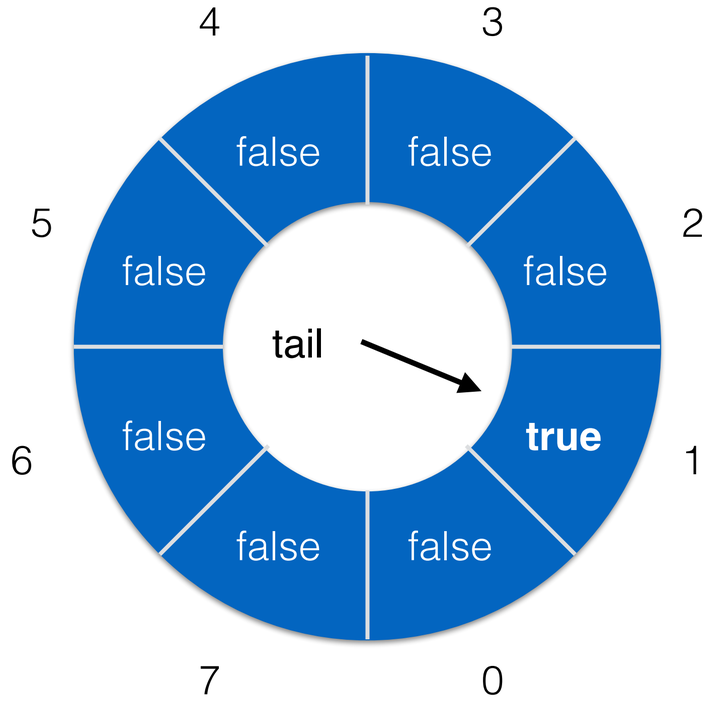
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	-1	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)

Lock rule:

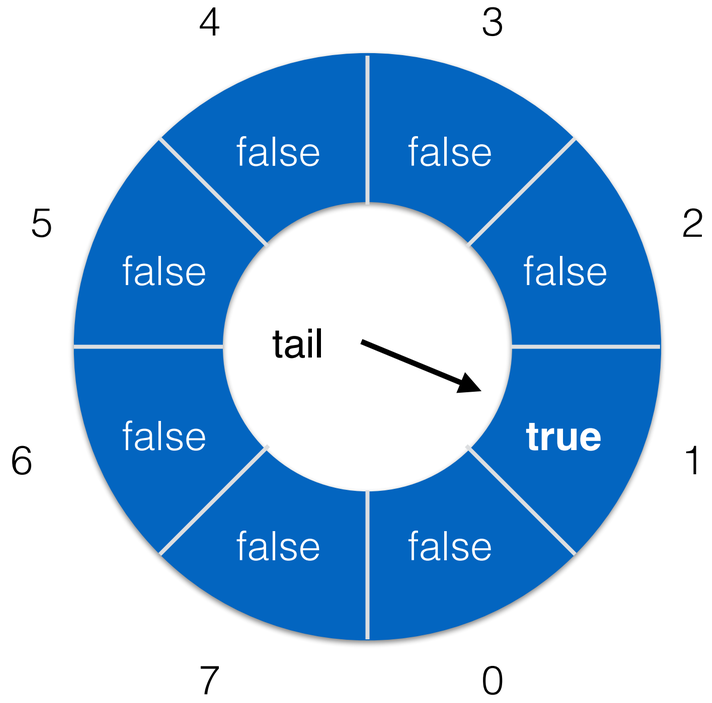
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)

Lock rule:

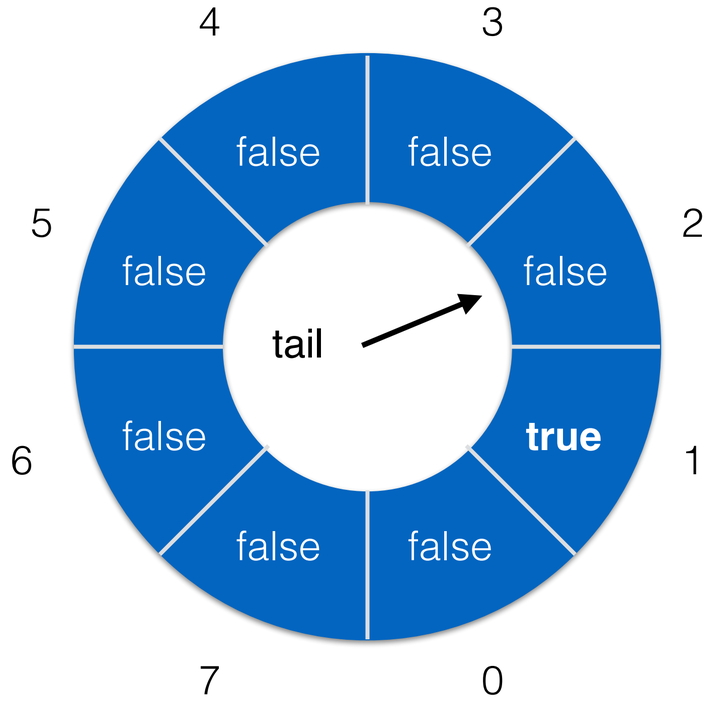
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)

Lock rule:

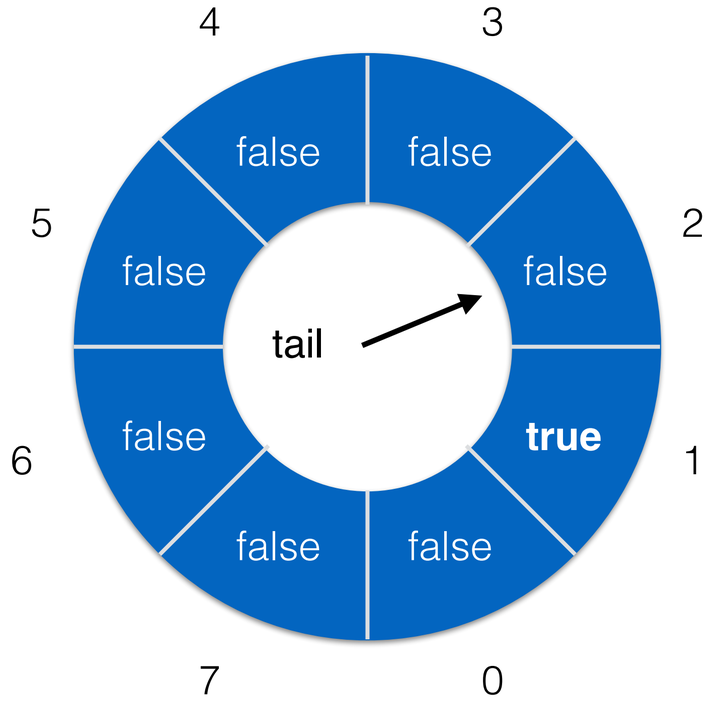
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)

Lock rule:

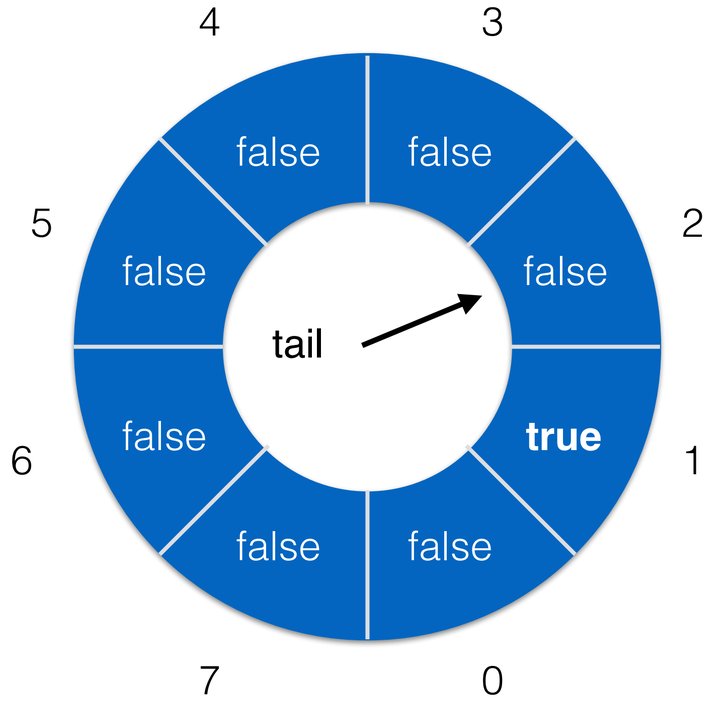
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	-1	**0**	-1	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)

Lock rule:

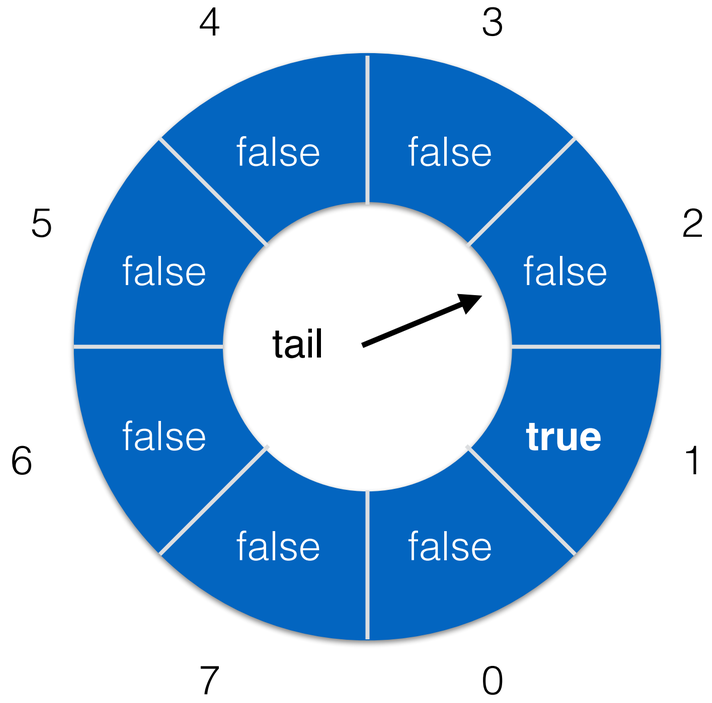
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	-1	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)

Lock rule:

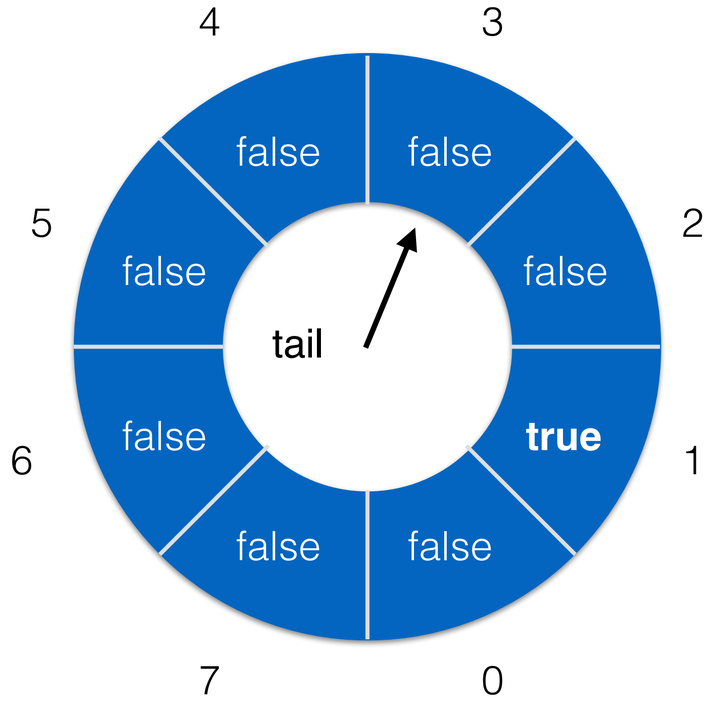
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	-1	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)

Lock rule:

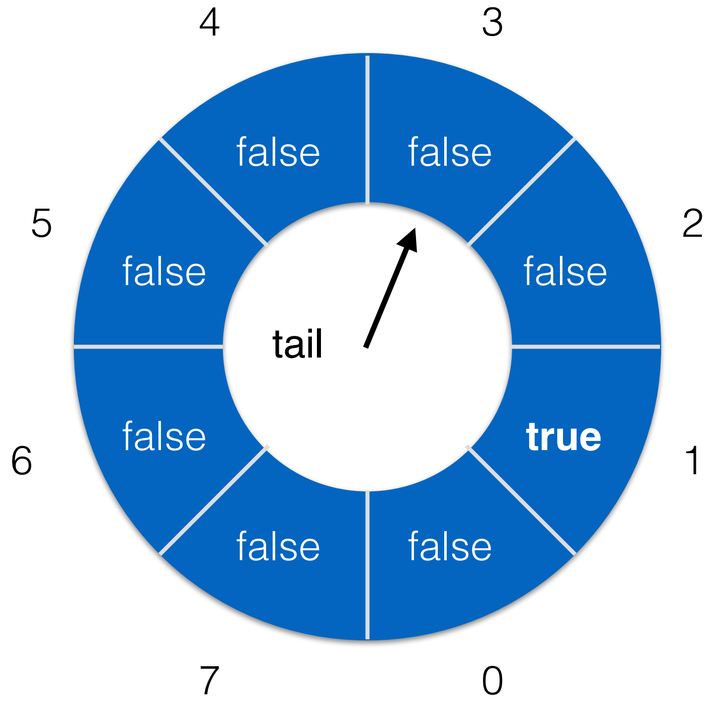
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	-1	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)
7. **lock**(3)

Lock rule:

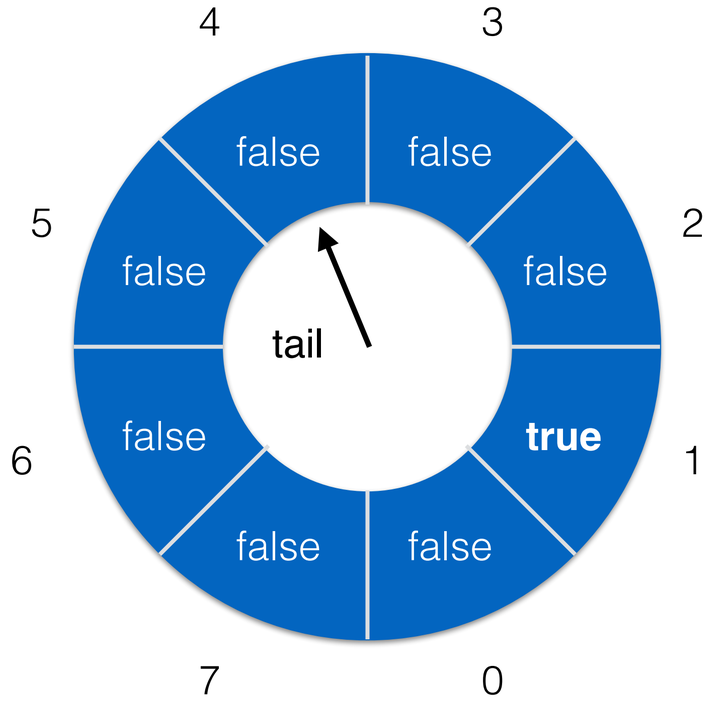
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	**3**	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)
7. **lock**(3)

Lock rule:

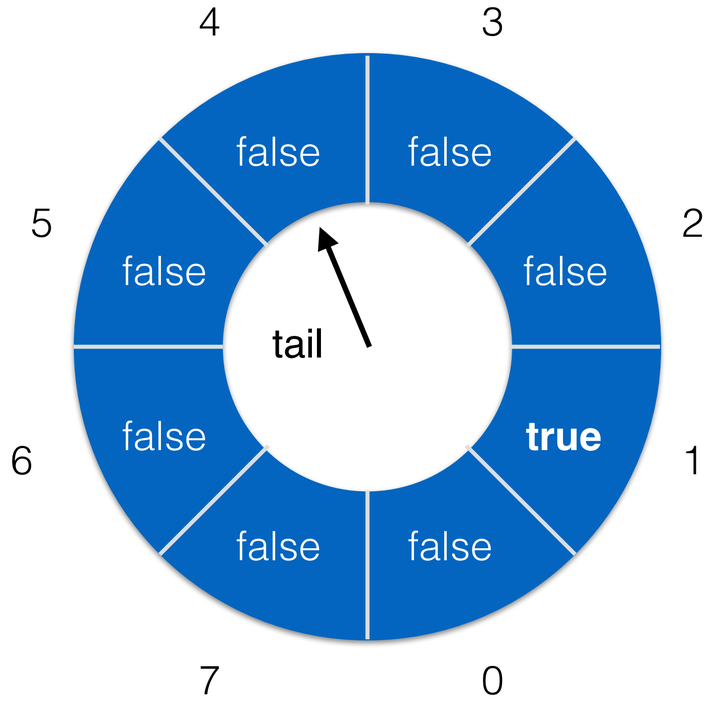
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	**3**	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)
7. **lock**(3)
8. **unlock**(5)

Lock rule:

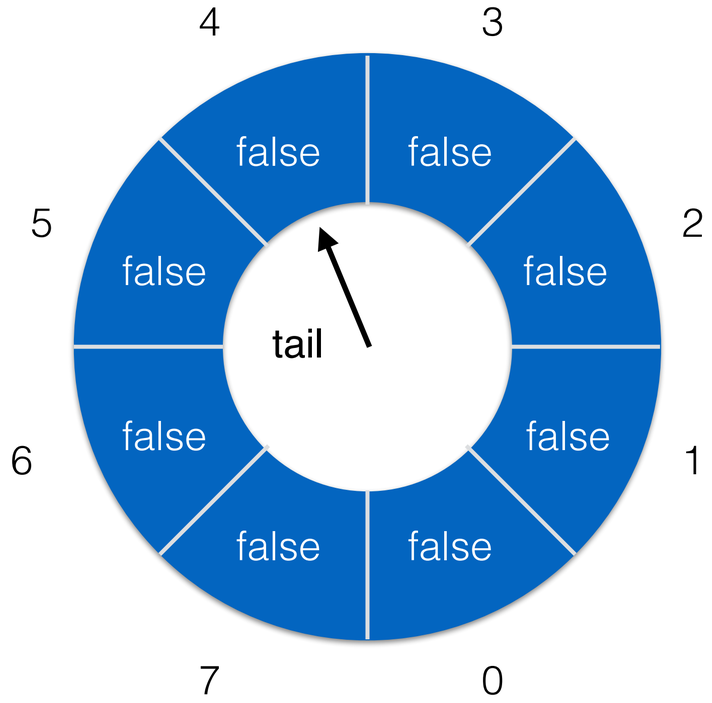
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	**3**	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)
7. **lock**(3)
8. **unlock**(5)

Lock rule:

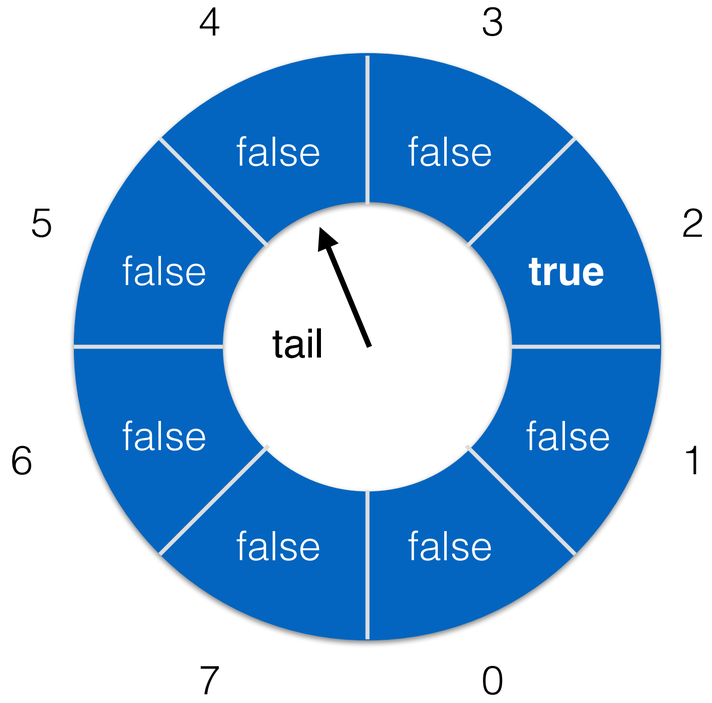
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	**3**	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)
7. **lock**(3)
8. **unlock**(5)

Lock rule:

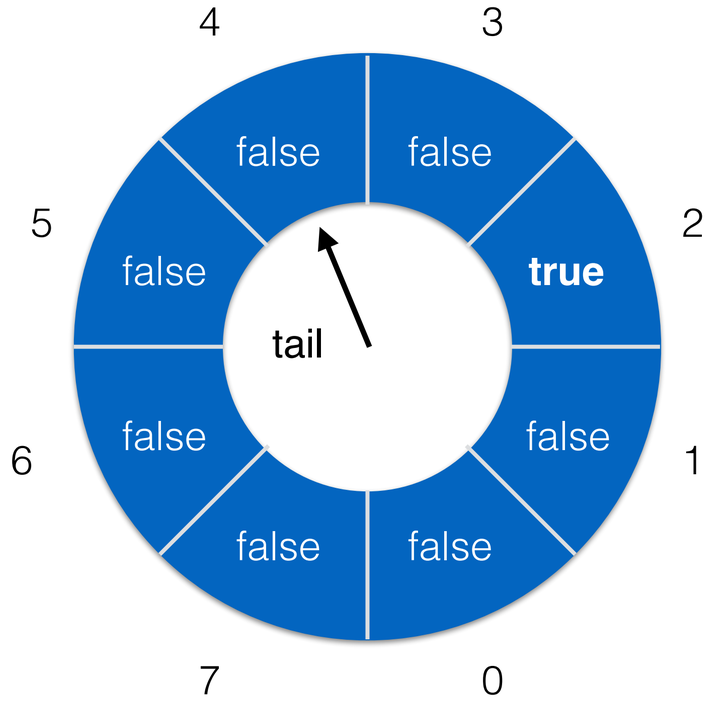
1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Anderson's Lock

flags
(cyclic array)



thread_id	0	1	2	3	4	5	6	7
slot	-1	**2**	**0**	**3**	-1	**1**	-1	-1

1. **lock**(2)
2. (2 in critical)
3. **unlock**(2)
4. **lock**(5)
5. (5 in critical)
6. **lock**(1)
7. **lock**(3)
8. **unlock**(5)
9. (1 in critical)

Lock rule:

1. Remember at which slot is the **tail** pointing to, and spin it by 1 (**fetch_and_add** atomically).
2. Wait until that **flag** becomes true.

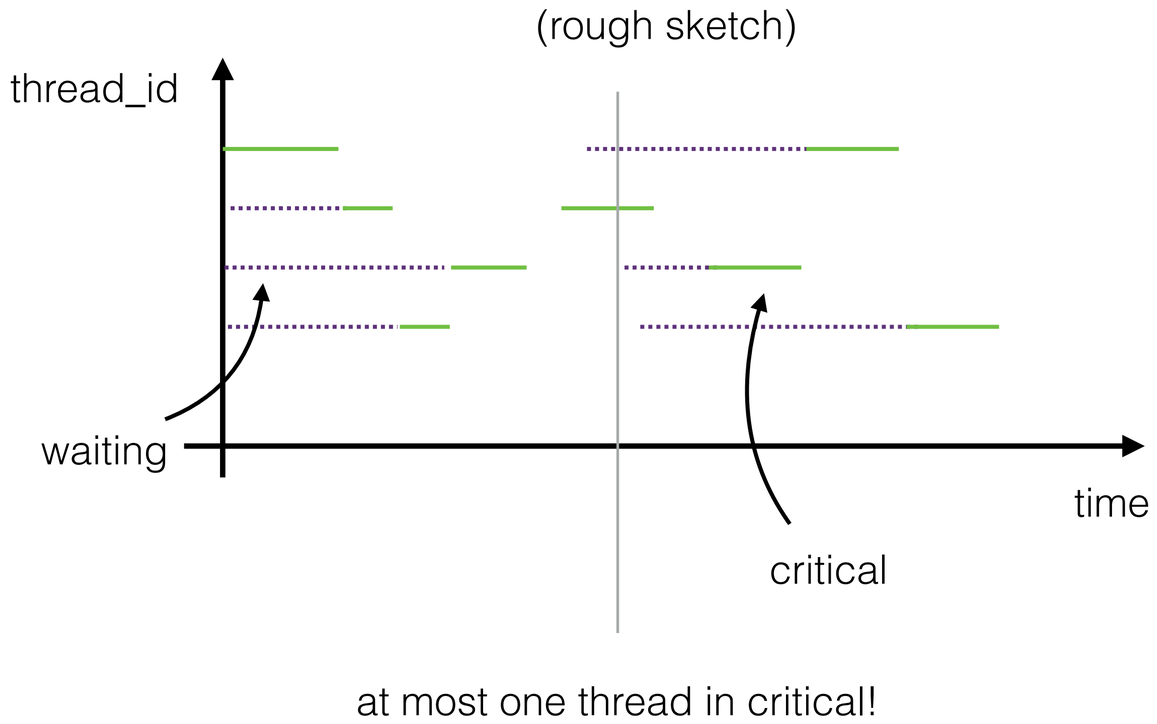
Unlock rule:

1. Set my **flag** to false.
2. Update next **flag** to true.

Question 3: Tasks

- Task 1: Implement Anderson's lock using the skeleton code.

- Task 2:
  1. Emulate the following parallel execution:
     - Each thread **repeats** 5 times:
       - lock()
       - do_work_W_{inside}()
       - unlock()
       - do_work_$W_{outside}$()
  2. Plot waiting time and time spent in the critical region.
  3. (see Exercise sheet for details)



at most one thread in critical!

References

- Cache design:
  https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf

- Agner's optimization guide:
  https://www.agner.org/optimize/microarchitecture.pdf

- Intel documentation:
  https://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

- https://stackoverflow.com/questions/23448528/how-is-an-lru-cache-implemented-in-a-cpu

- https://en.wikipedia.org/wiki/Cache_replacement_policies#Pseudo-LRU_(PLRU)

- Just for fun (see caches section):
  http://csillustrated.berkeley.edu/illustrations.php