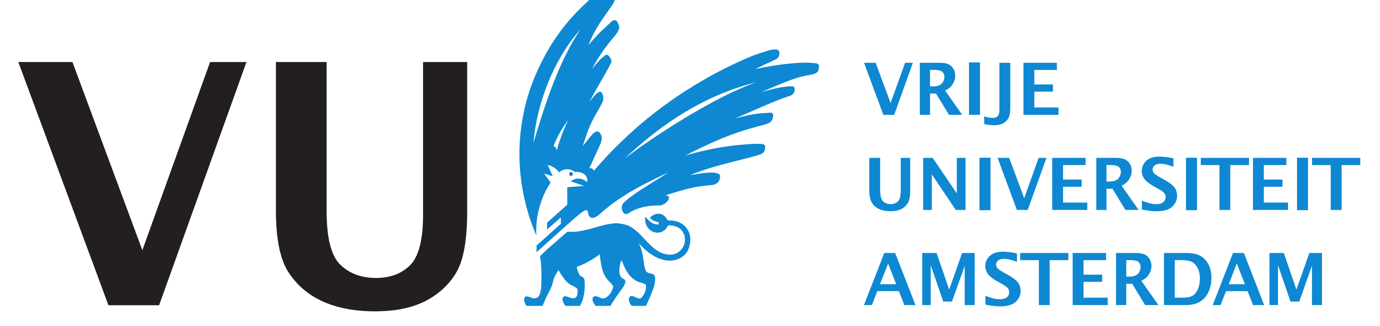
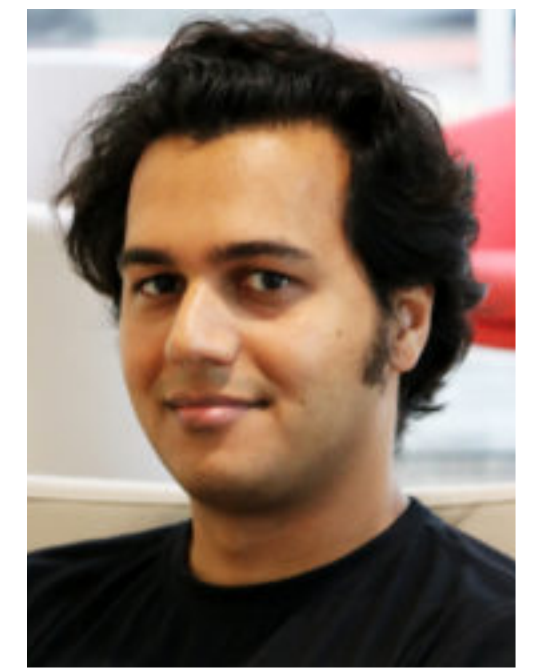
# ASLR on the Line

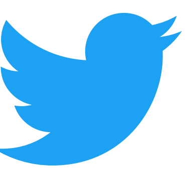
Ben Gras, Kaveh Razavi, **Erik Bosman**, Herbert Bos, Cristiano Giuffrida



# Erik Bosman



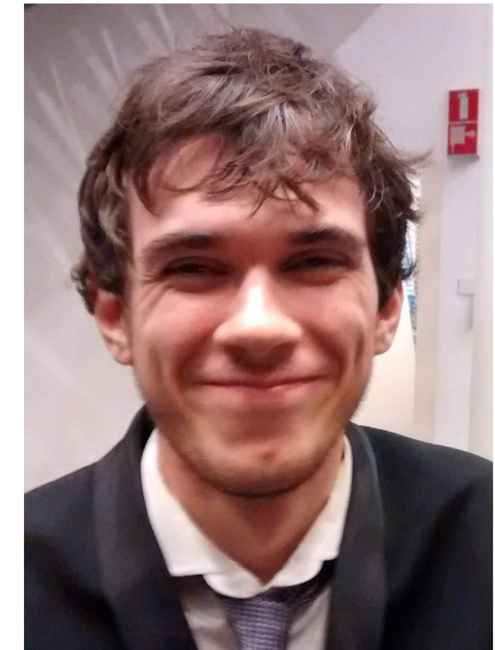
**Kaveh Razavi**



**@gober**

**Ben Gras**

**@bjg**



**Stephan van Schaik**

# WARNING

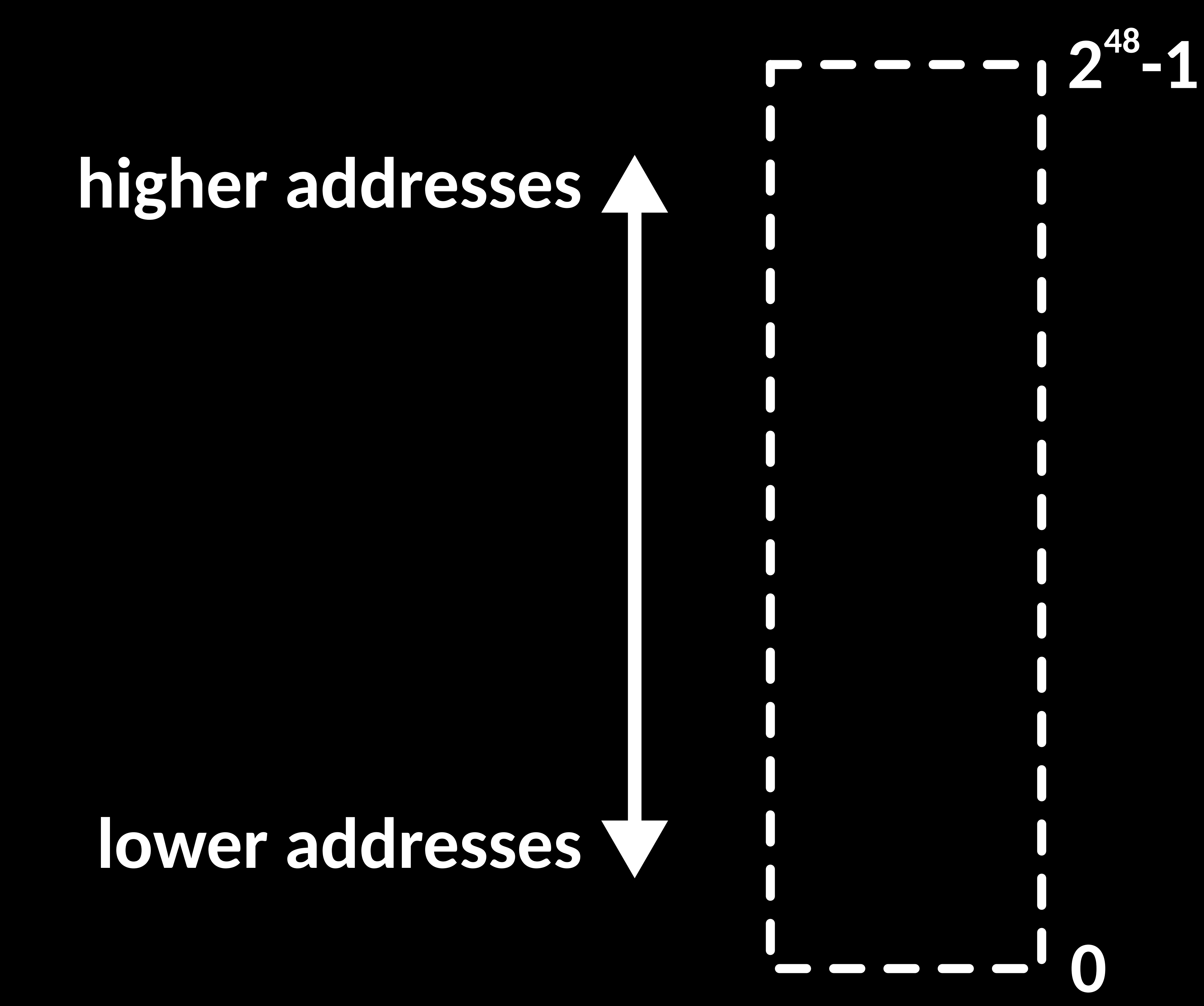
THIS PRESENTATION
MAY CONTAIN POINTERS

# ASLR

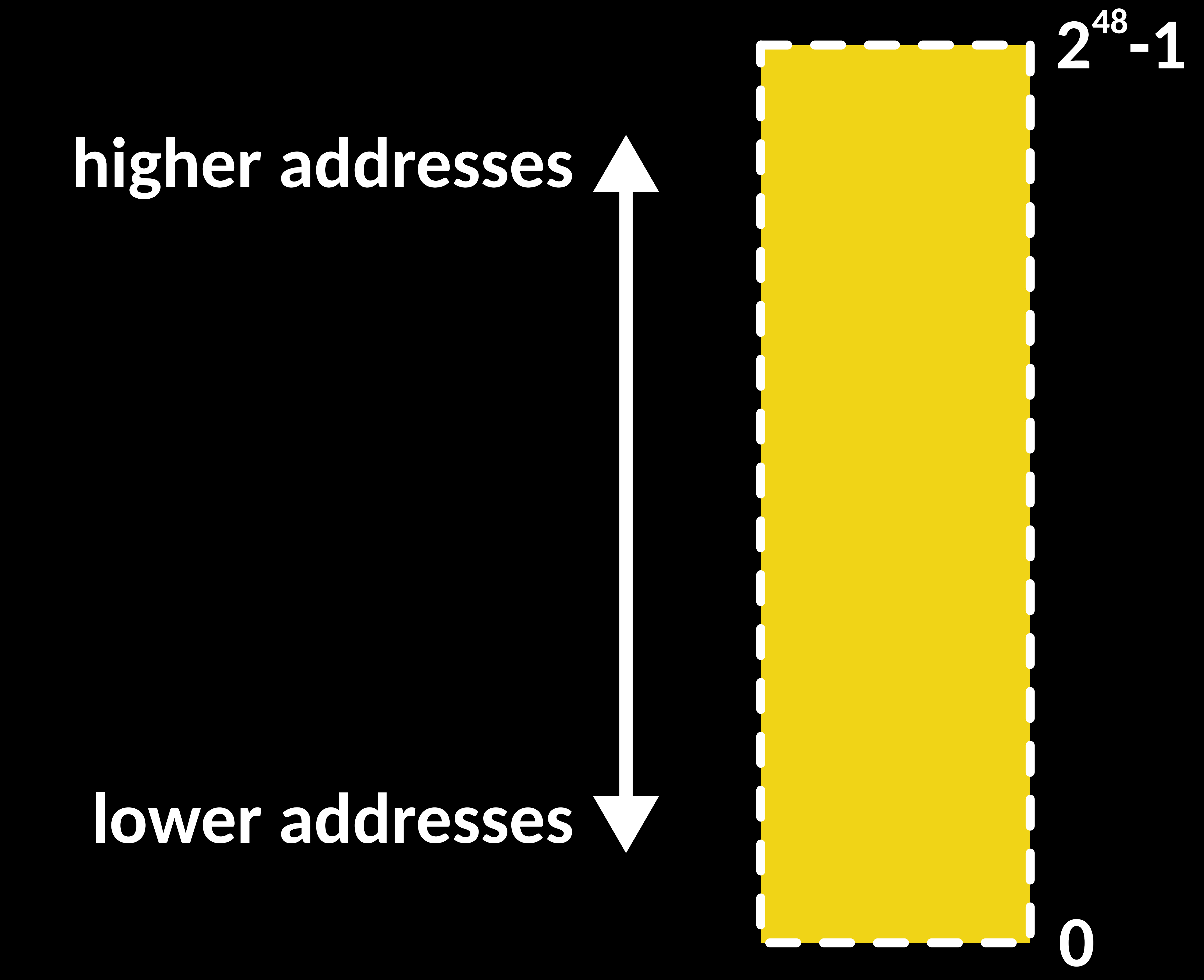
# Address Space Layout Randomization

**Widely deployed exploit mitigation strategy:**

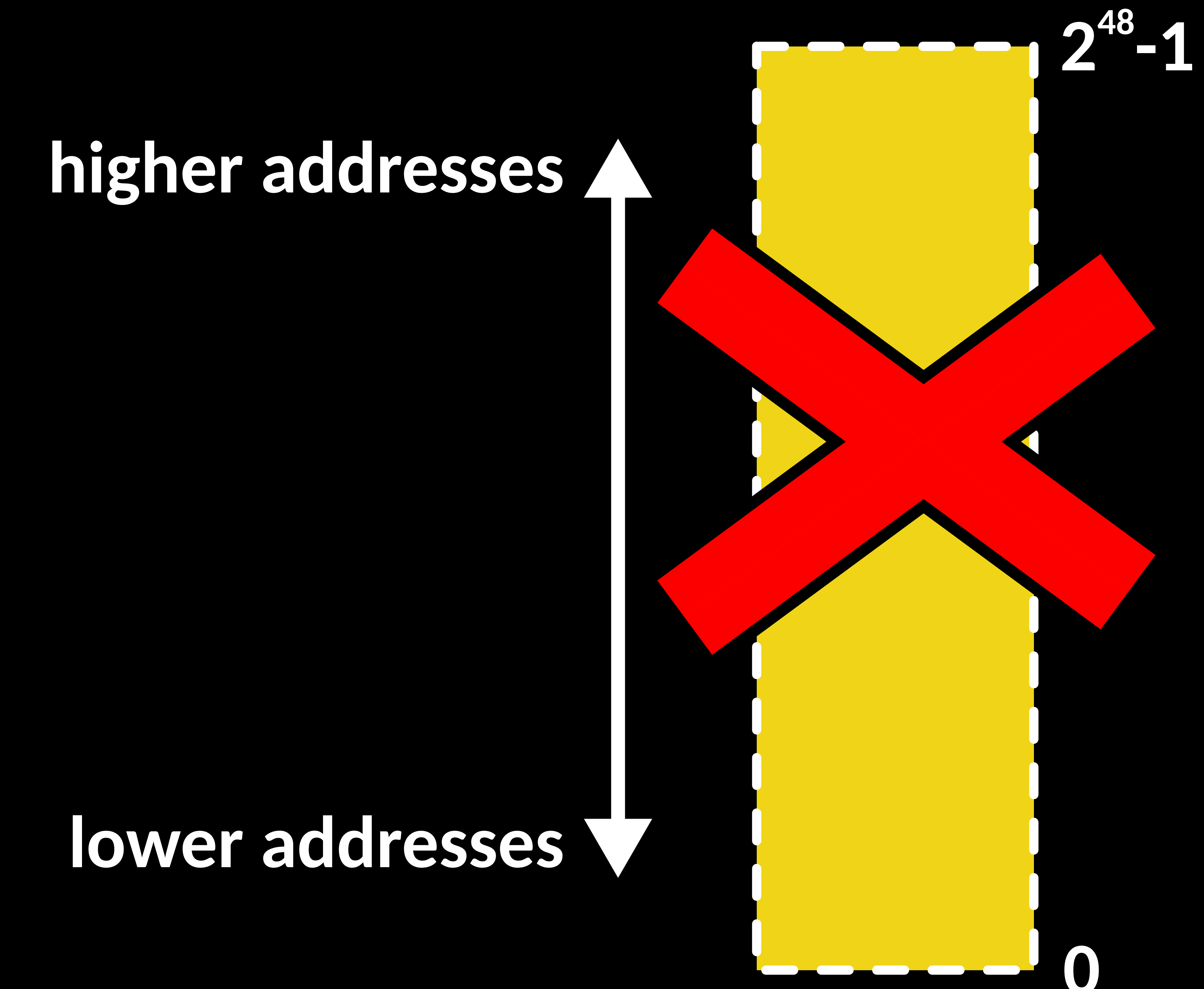
**Choose a different location for code and data every time a process is run.**

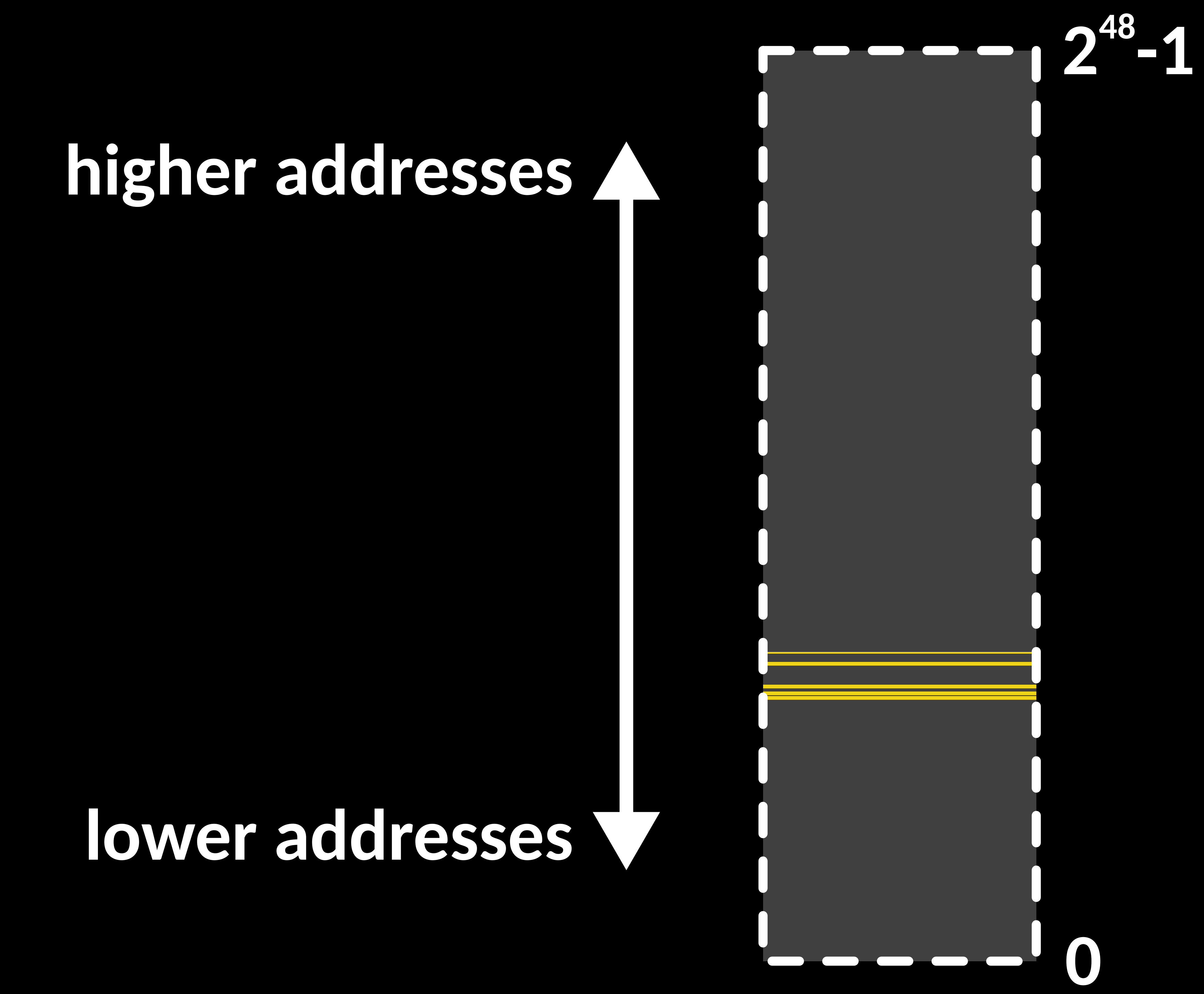


$2^{48}$-1
higher addresses
lower addresses
0

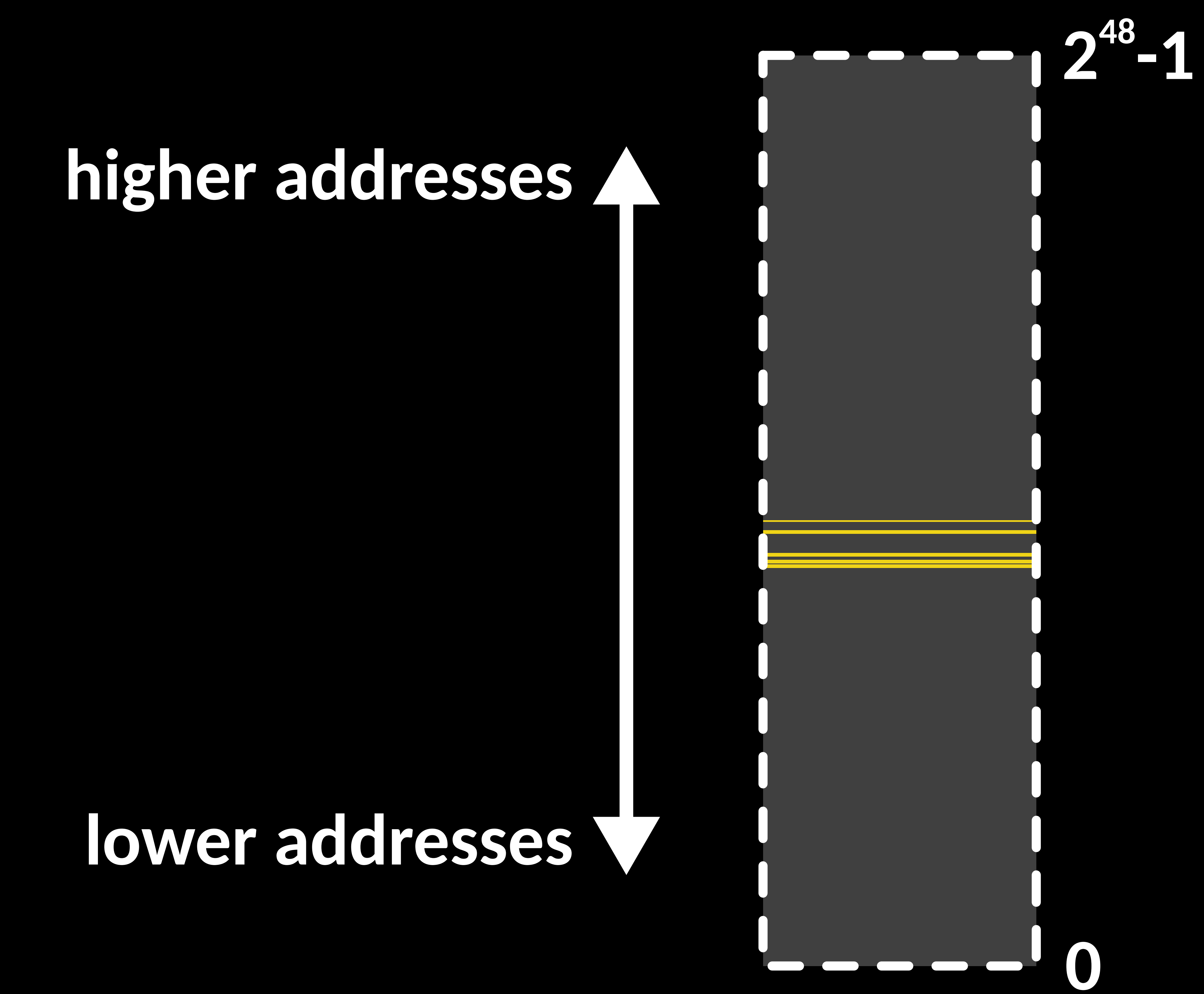


2^48-1
higher addresses
lower addresses
0

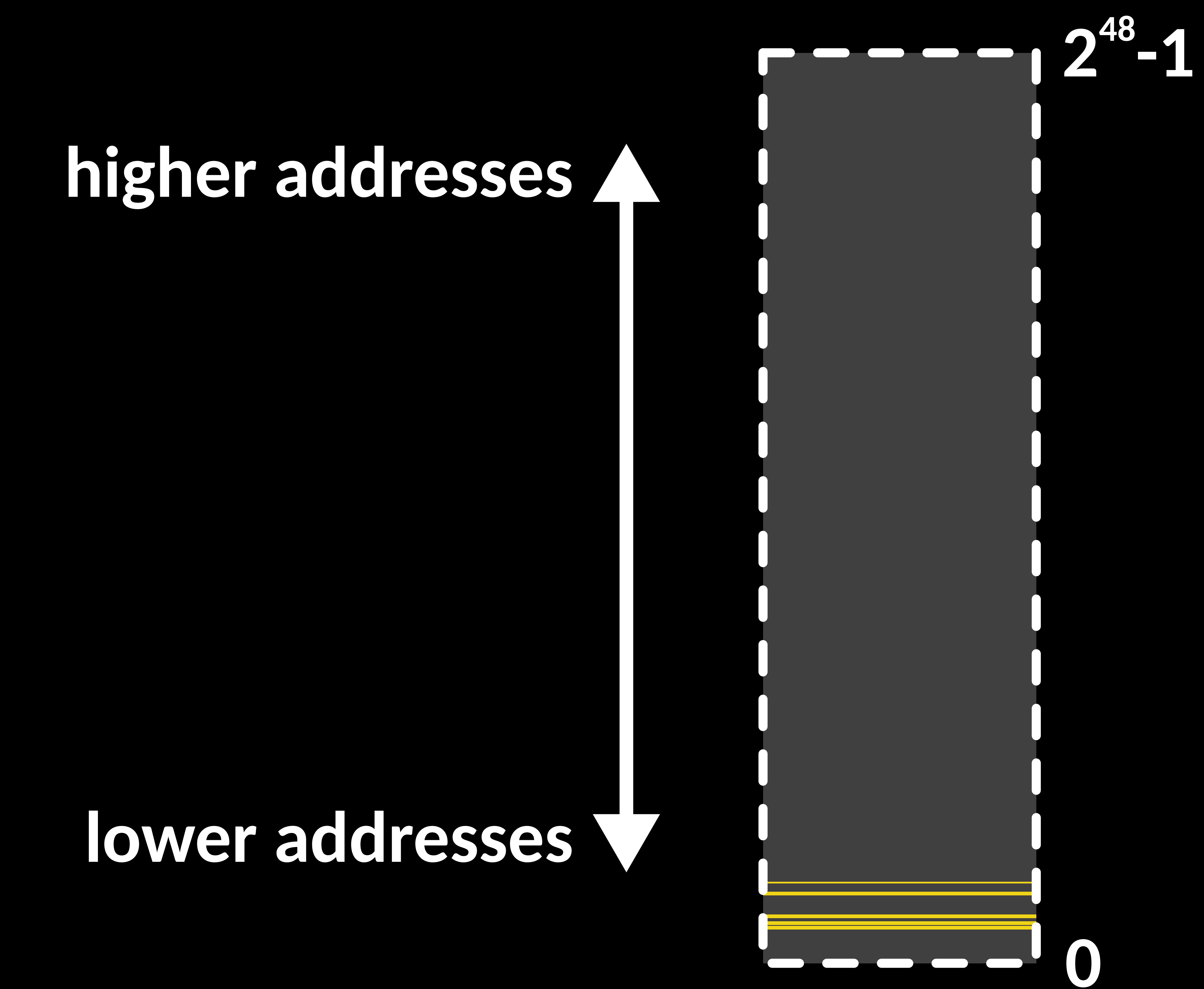

$2^{48}$-1
higher addresses
lower addresses
0



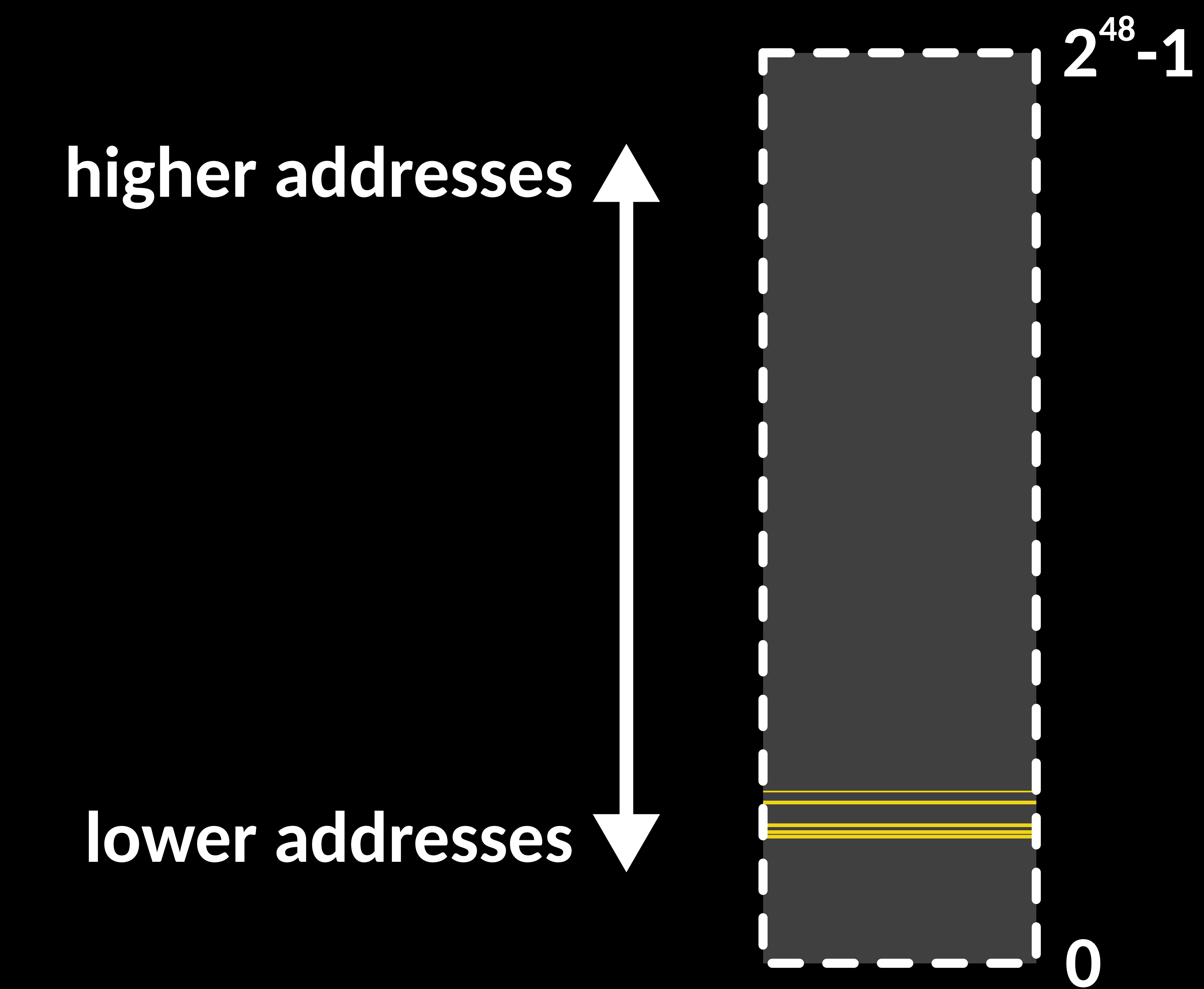
higher addresses
lower addresses
$2^{48}$-1
0



$2^{48}$-1
higher addresses
lower addresses
0



$2^{48}$-1
higher addresses
lower addresses
0


higher addresses
lower addresses
$2^{48}$-1
0

# Address Space Layout Randomization

Makes life for exploit writers a bit more difficult.

Usually exploits need to know the location of certain data in memory.

# A Single Leak Reveals

-- Joshua Drake

# Address Space Layout Randomization

Exploit writers need to find a bug which leaks addresses without crashing the program.

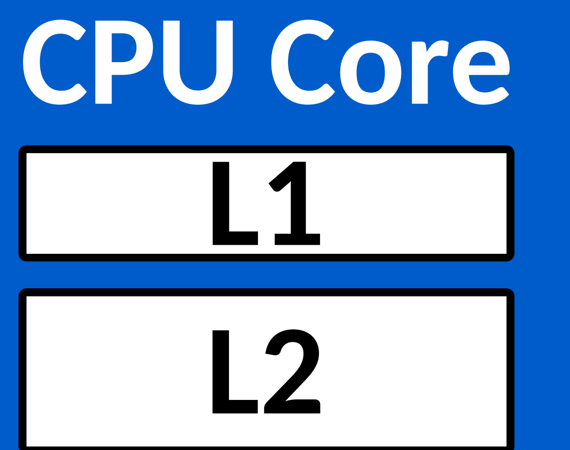
... or do they?

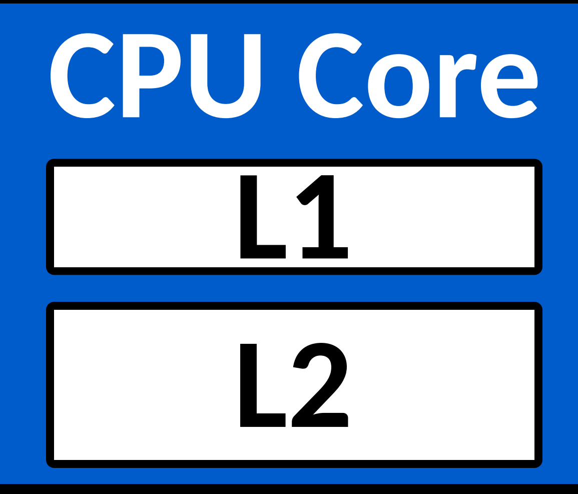
# This Presentation:

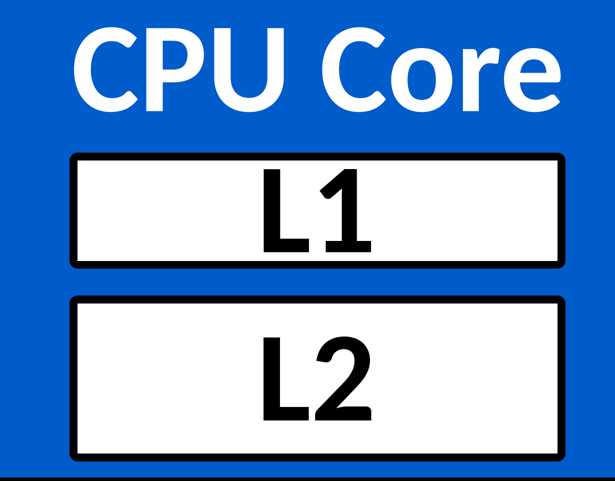
**ASLR ⊕ Cache (AnC)**

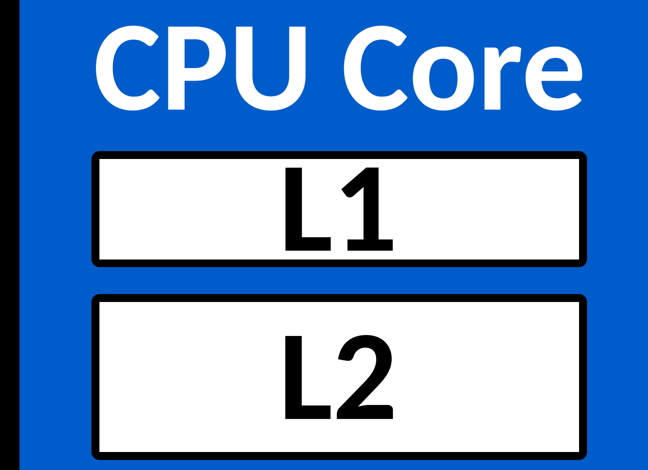
**A side-channel attack on processes baked into the *hardware* to discover ASLR information from Javascript in the browser.**

# Modern CPU architectures

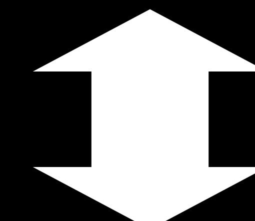









L3 (Last Level Cache), shared between cores



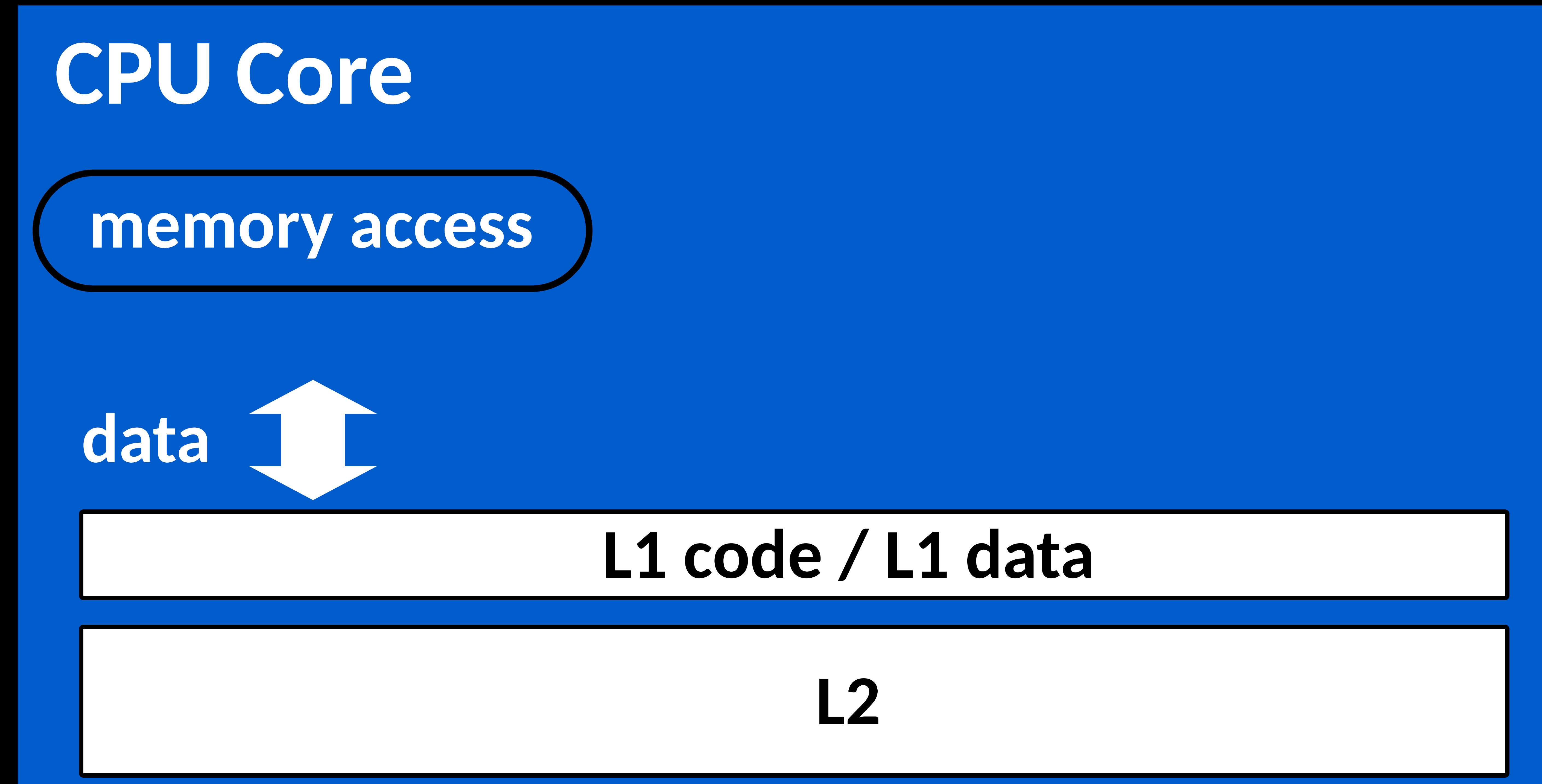
DDR Memory

CPU Core

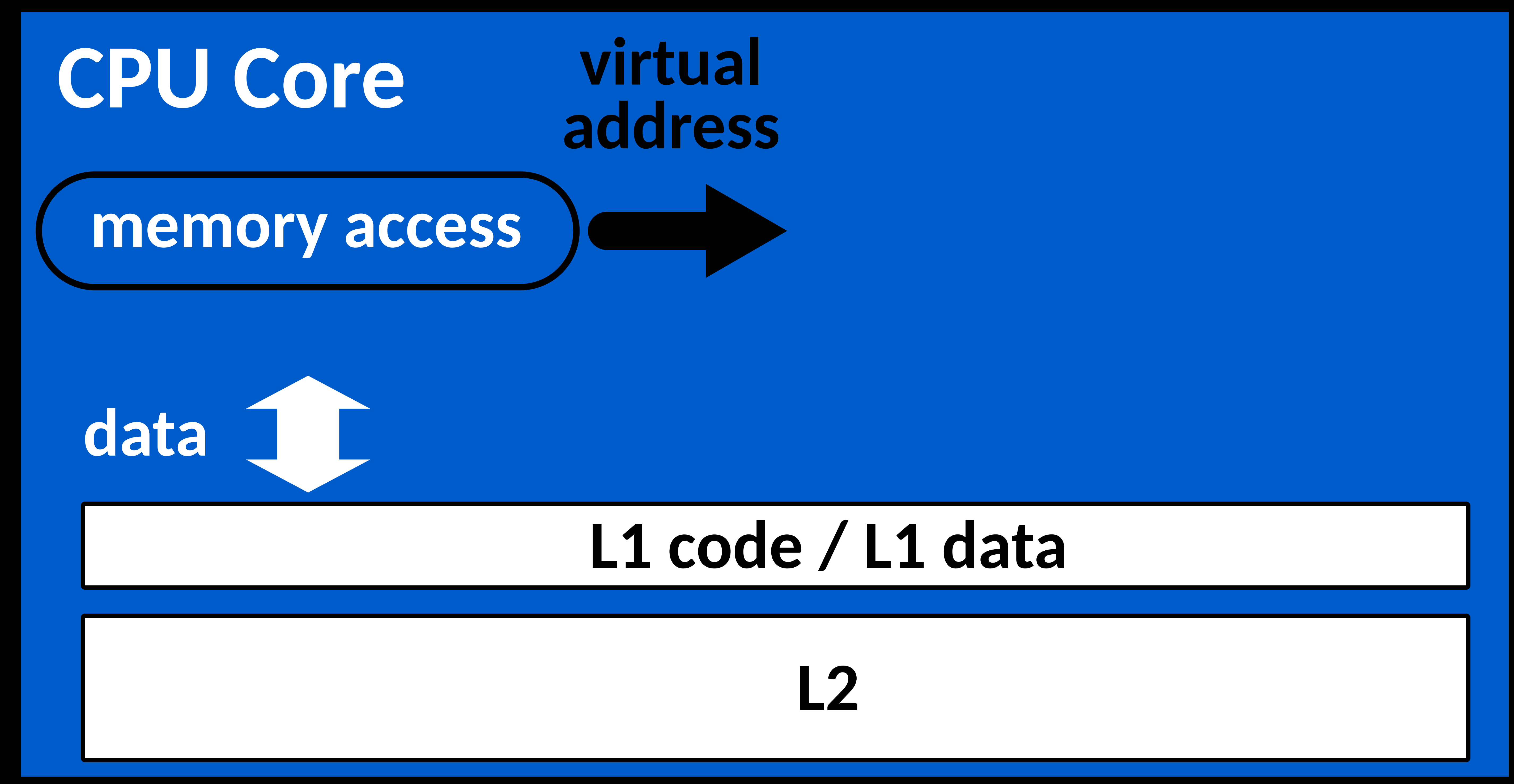
L1 code / L1 data

L2

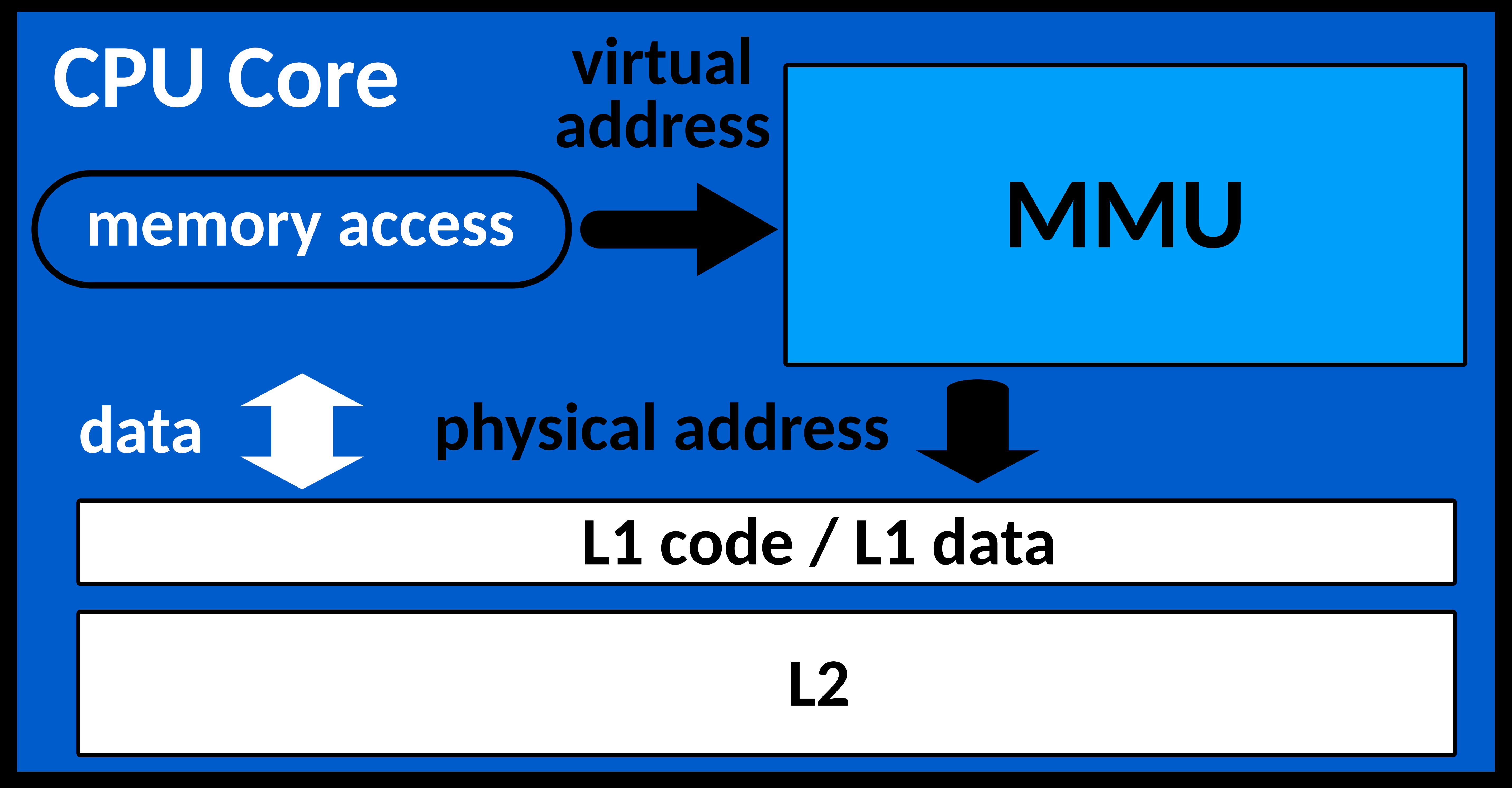
… L3 (Last Level Cache), shared between cores …



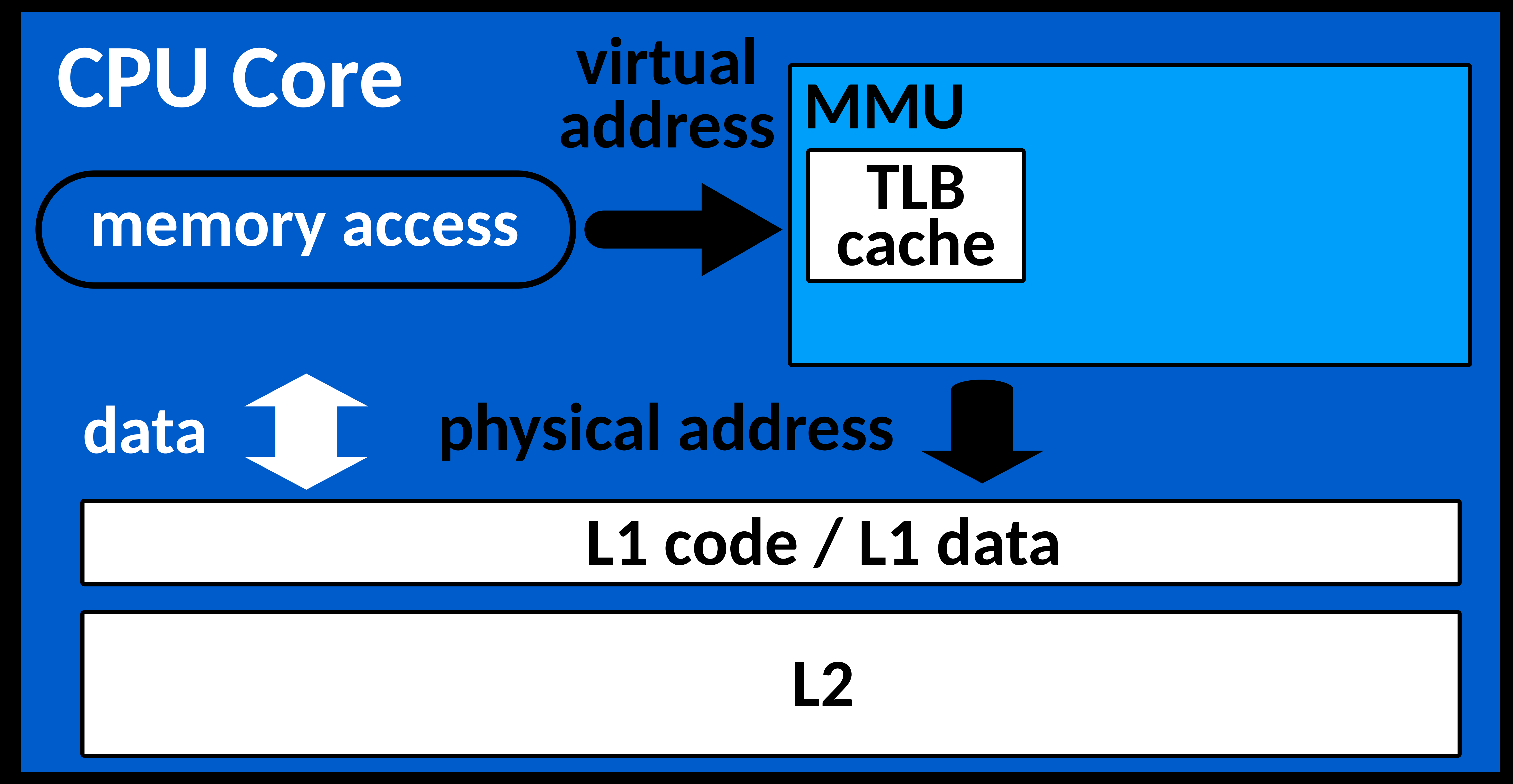

L3 (Last Level Cache), shared between cores



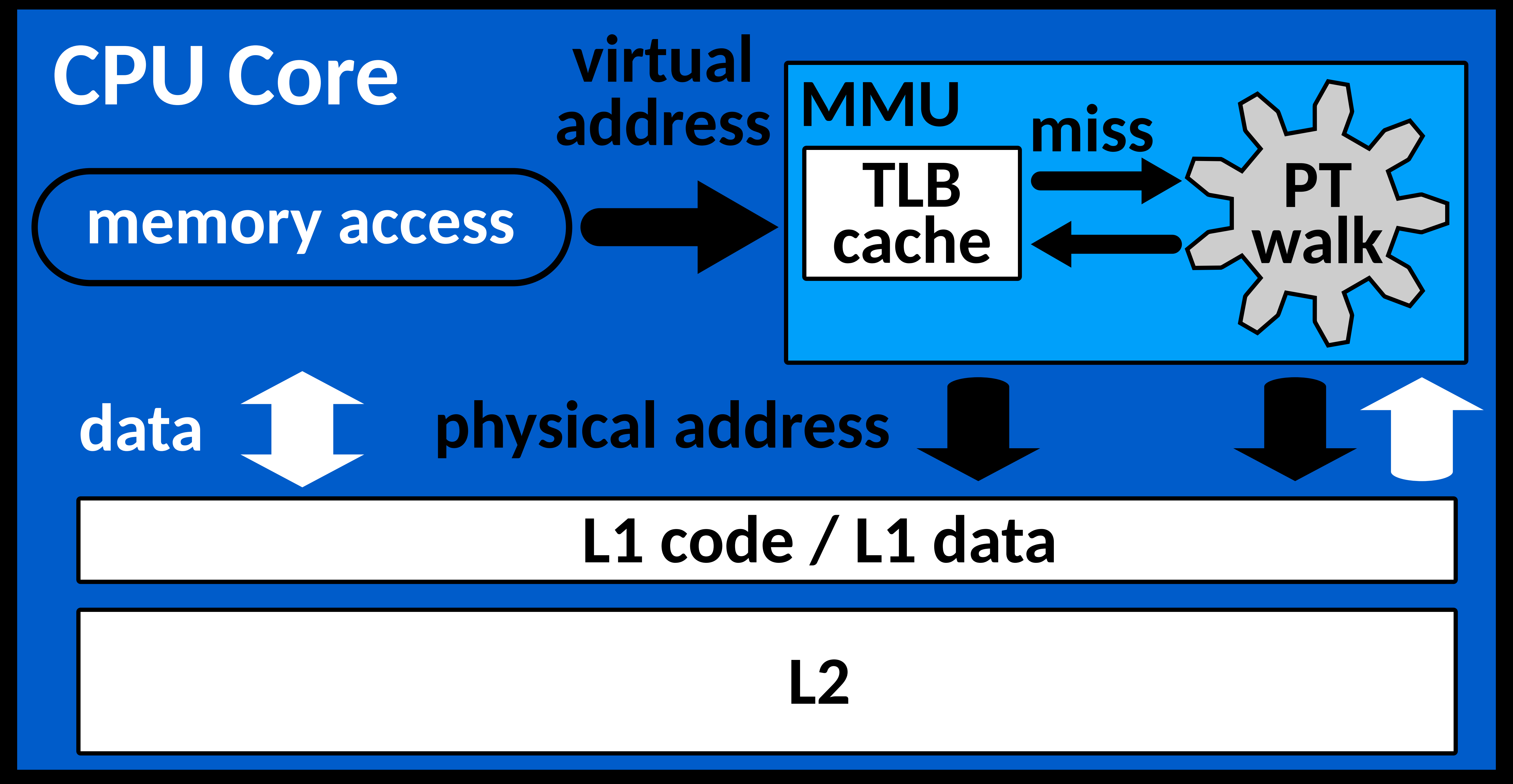

L3 (Last Level Cache), shared between cores



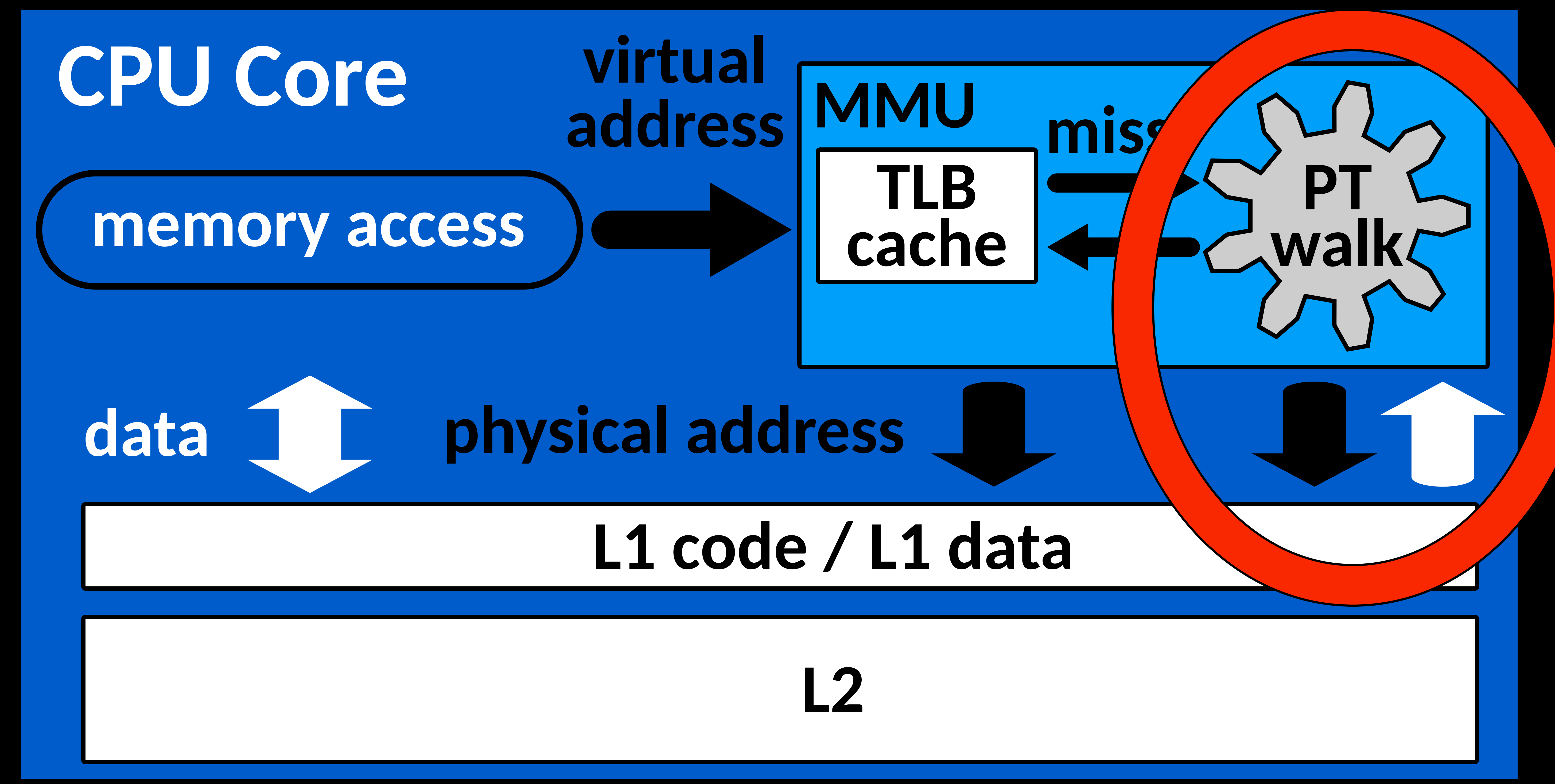

L3 (Last Level Cache), shared between cores




L3 (Last Level Cache), shared between cores




L3 (Last Level Cache), shared between cores

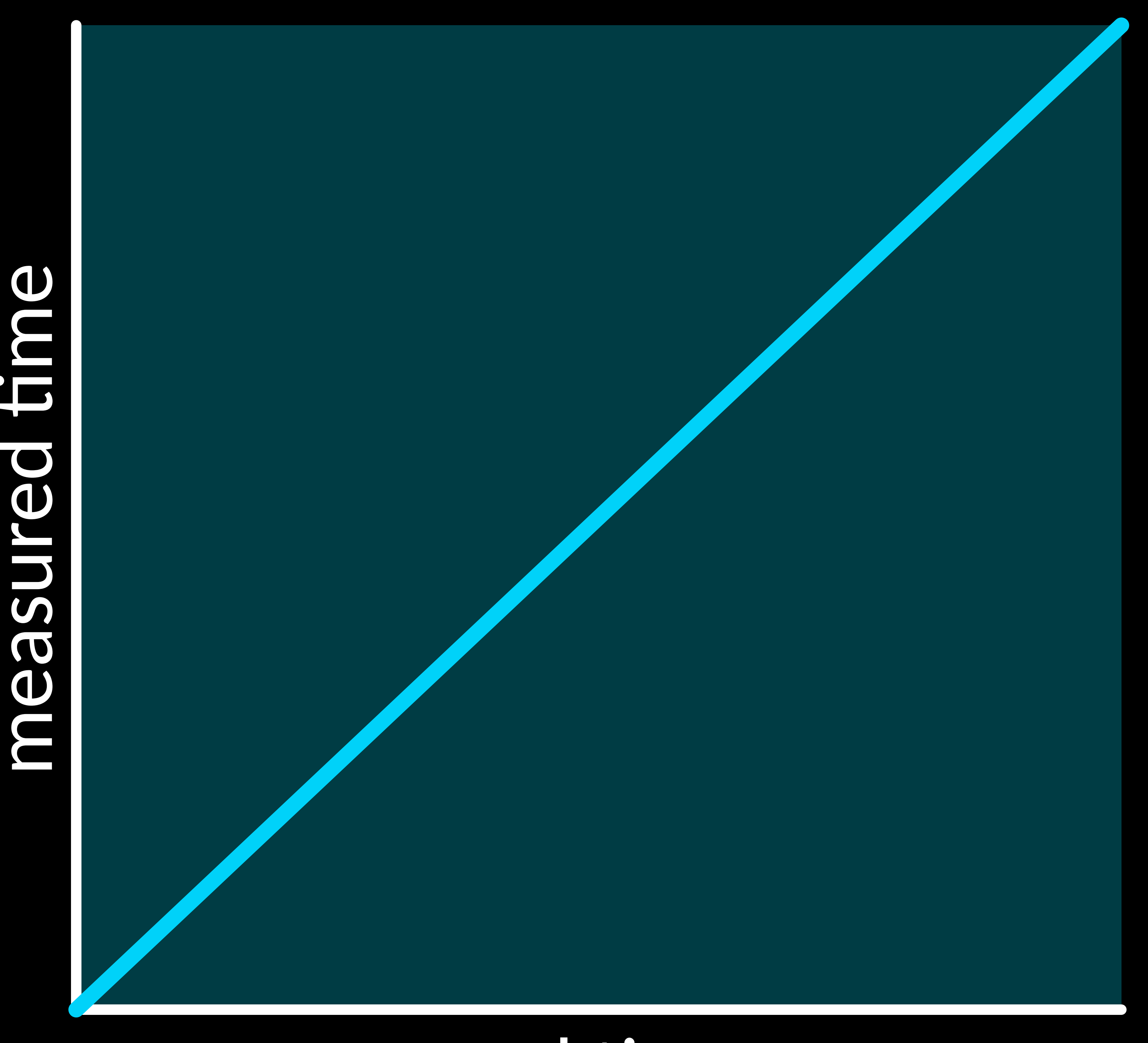



L3 (Last Level Cache), shared between cores

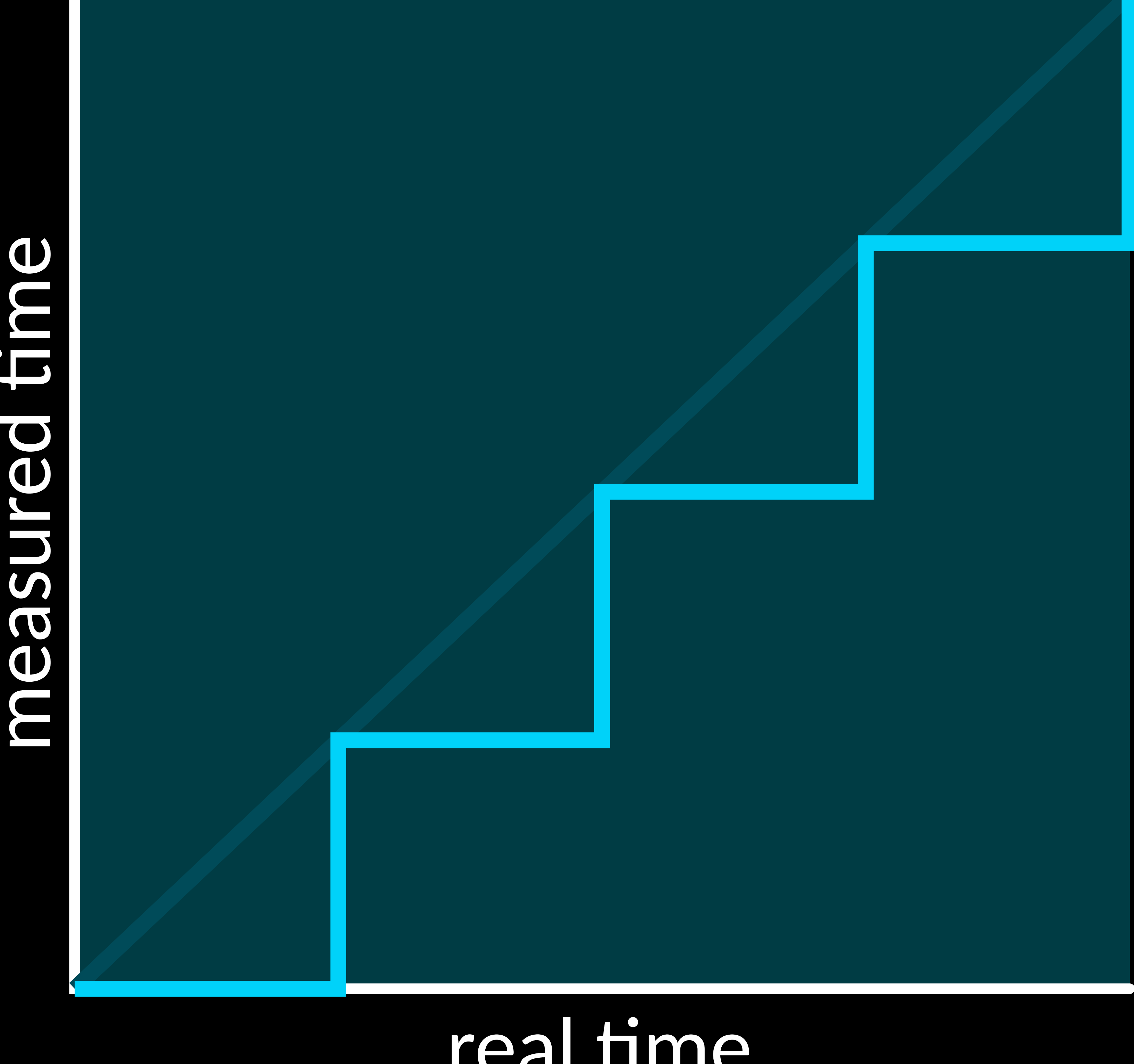
# Timers in Javascript

```
t0=performance.now();

operation();

t1=performance.now();
t = t1-t0;
```

```
t0=performance.now();

operation();

t1=performance.now();
t = t1-t0;
```

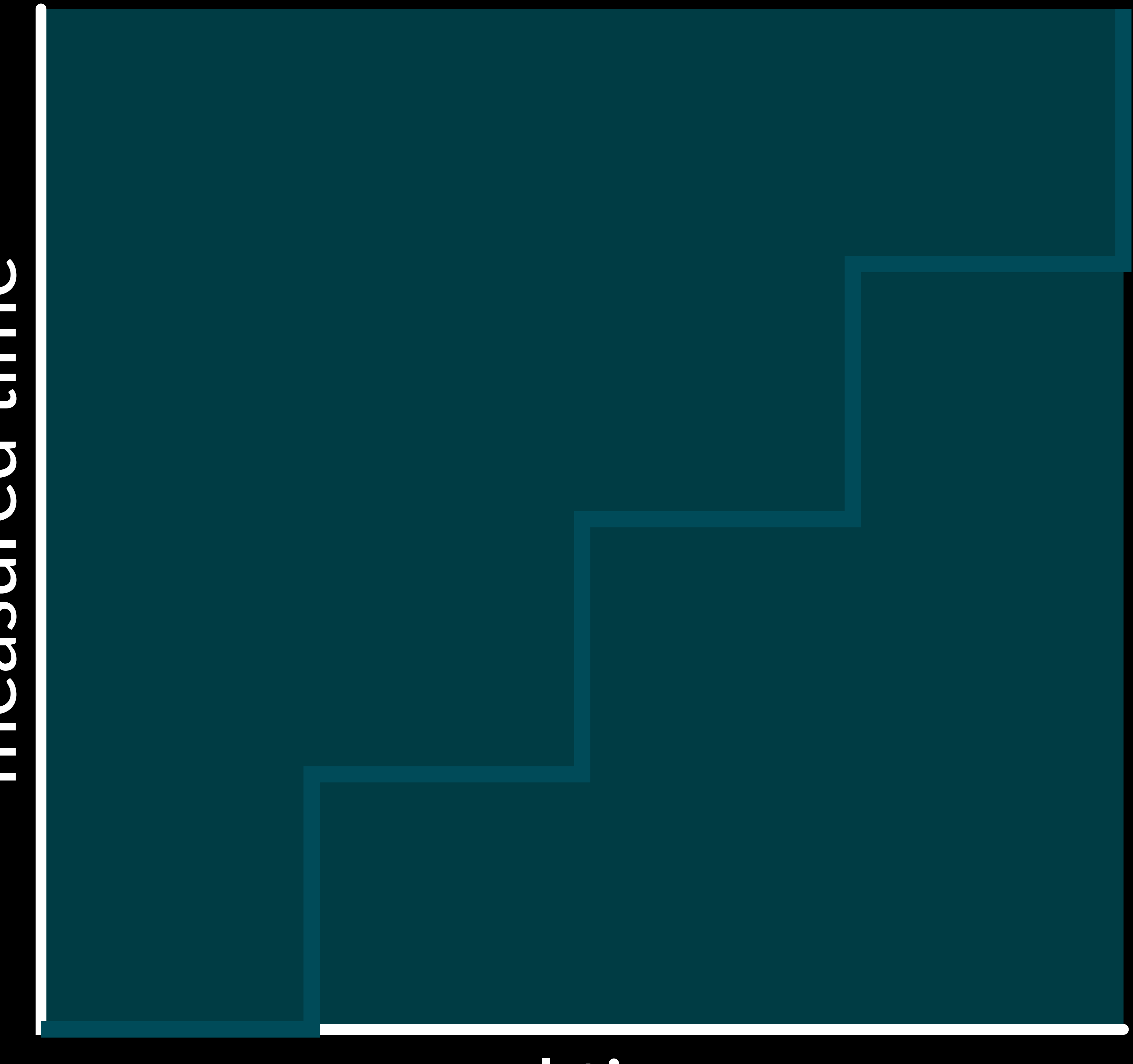


```
t0=performance.now();

operation();

t1=performance.now();
t = t1-t0;
```



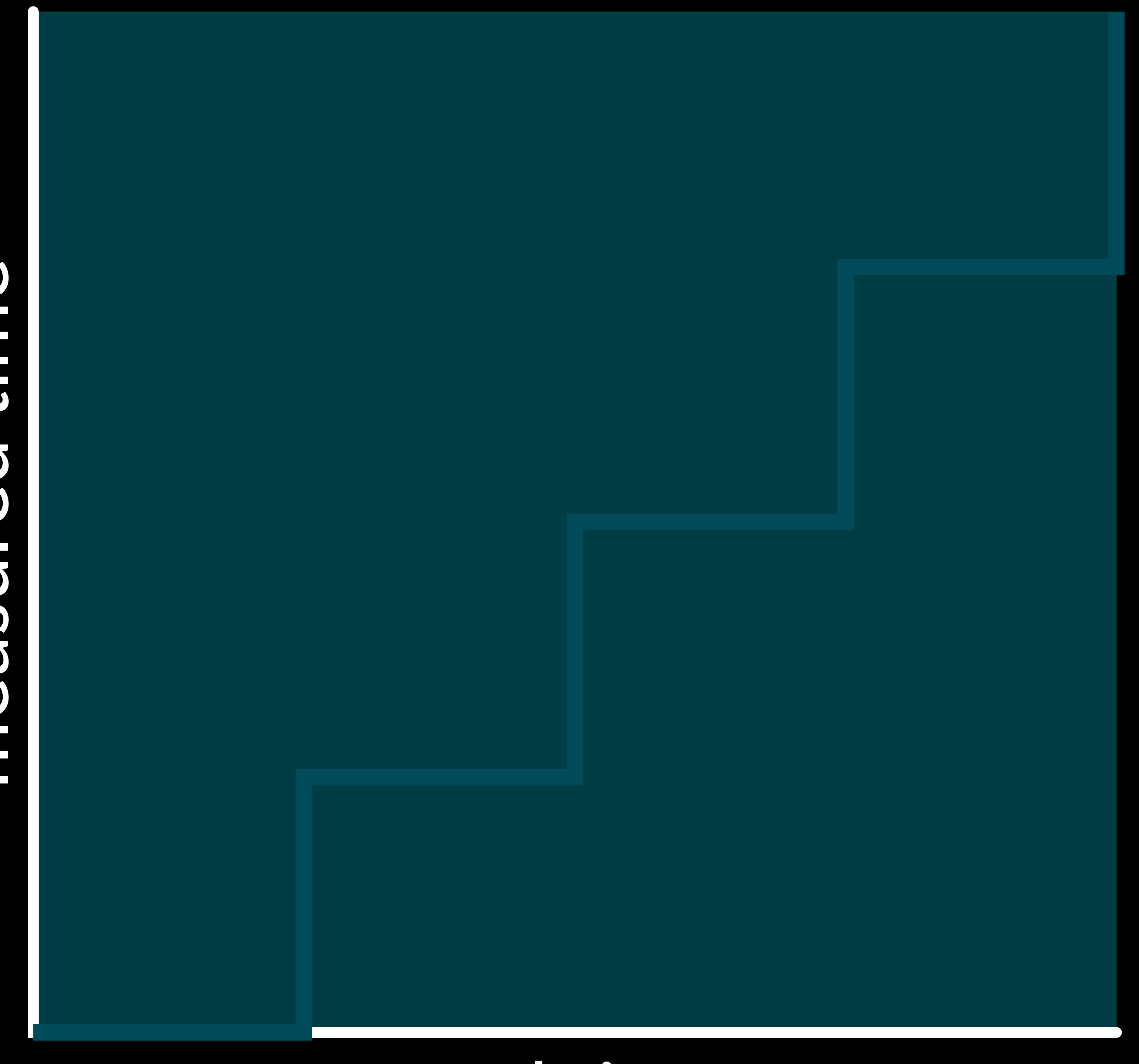

after anti- side-channel mitigations (firefox)

```
c = 0;
t0 = p.now();
while(t0 == p.now());
t1 = p.now();

operation();

while(t1 == p.now())
{ c++; }
```



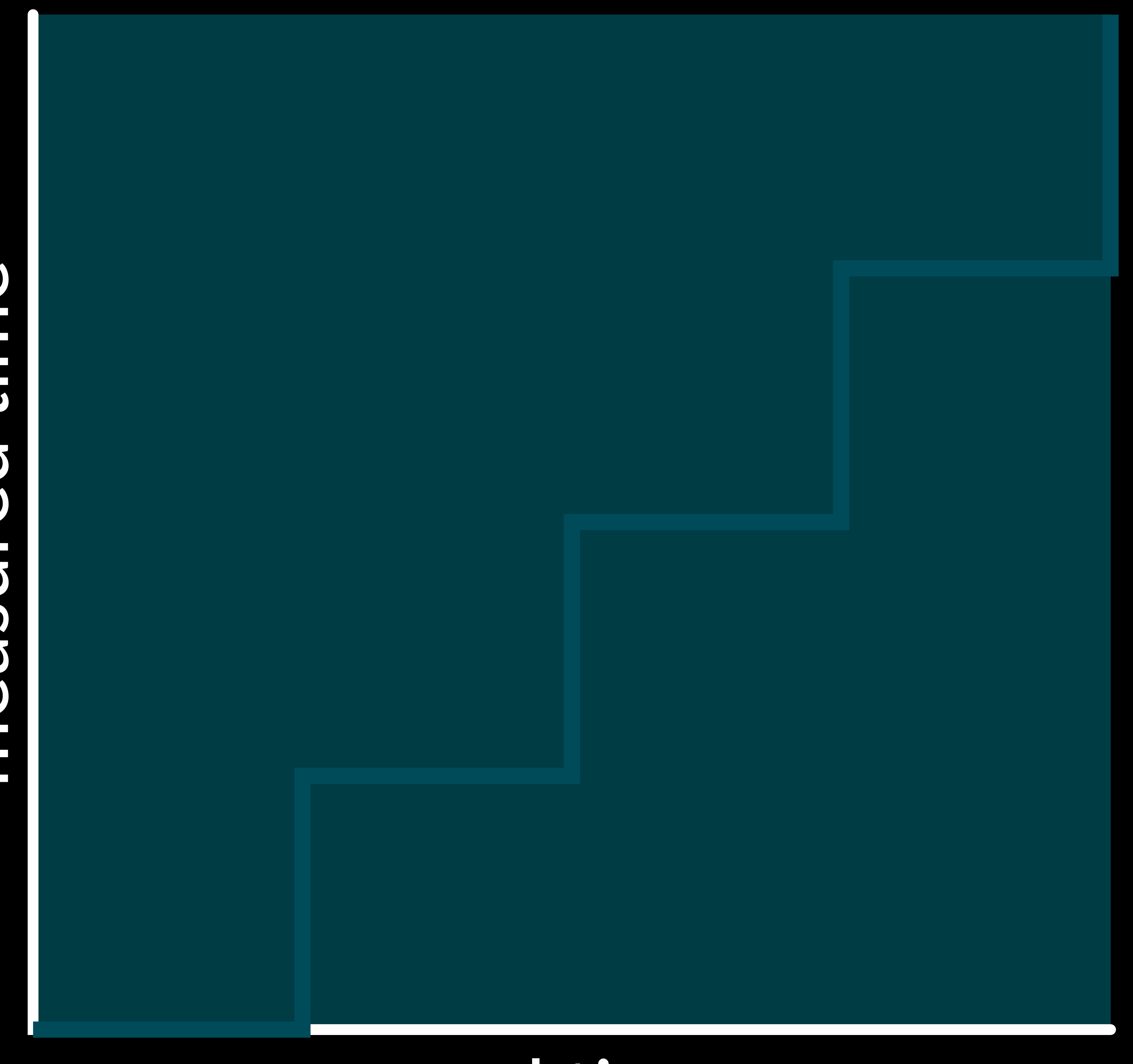

after anti- side-channel mitigations (firefox)

```
c = 0;
t0 = p.now();
●while(t0 == p.now());
t1 = p.now();

operation();

while(t1 == p.now())
{ c++; }
```



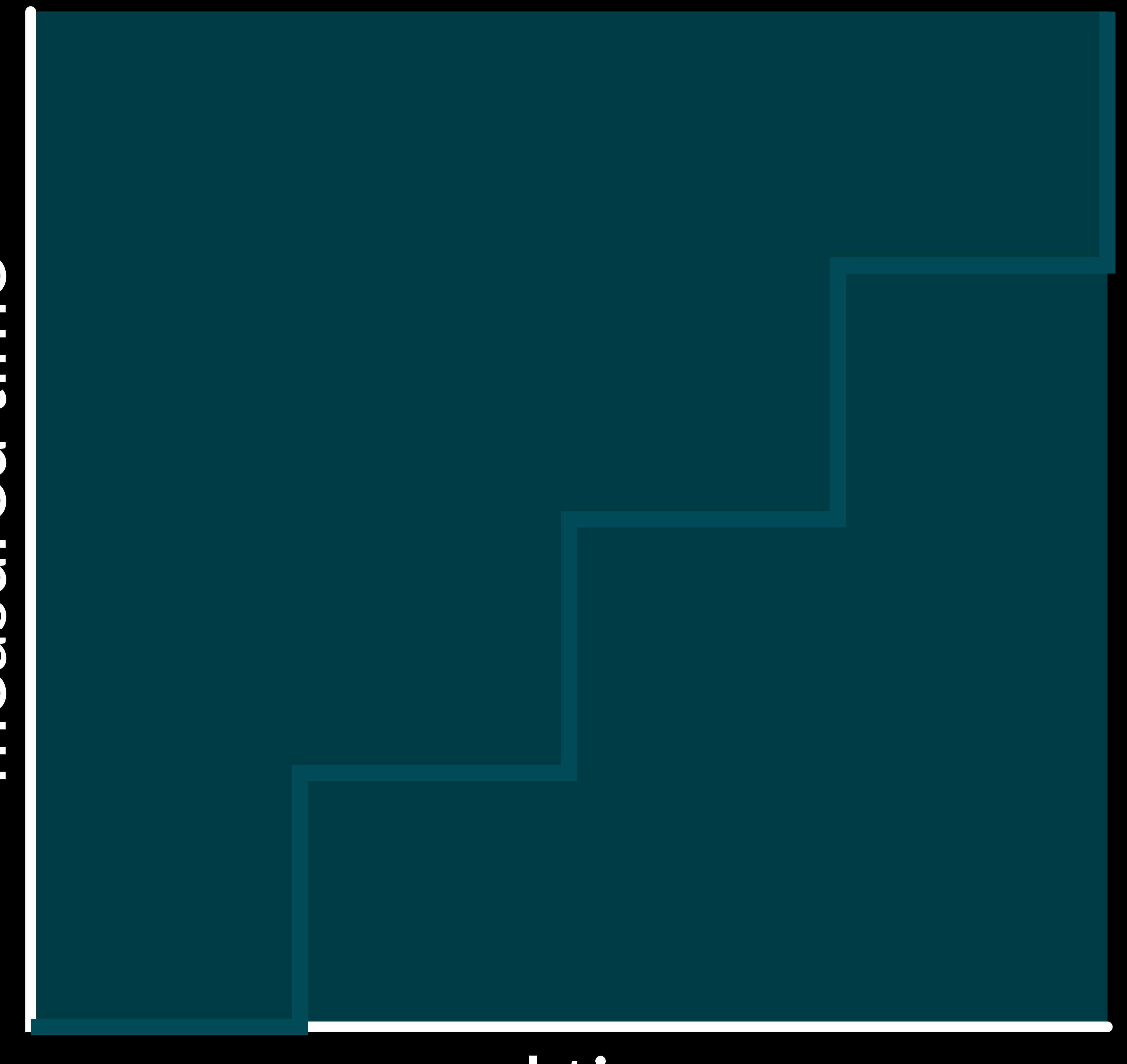

after anti- side-channel mitigations (firefox)

```
c = 0;
t0 = p.now();
while(t0 == p.now());
t1 = p.now();

operation();

while(t1 == p.now())
{ c++; }
```



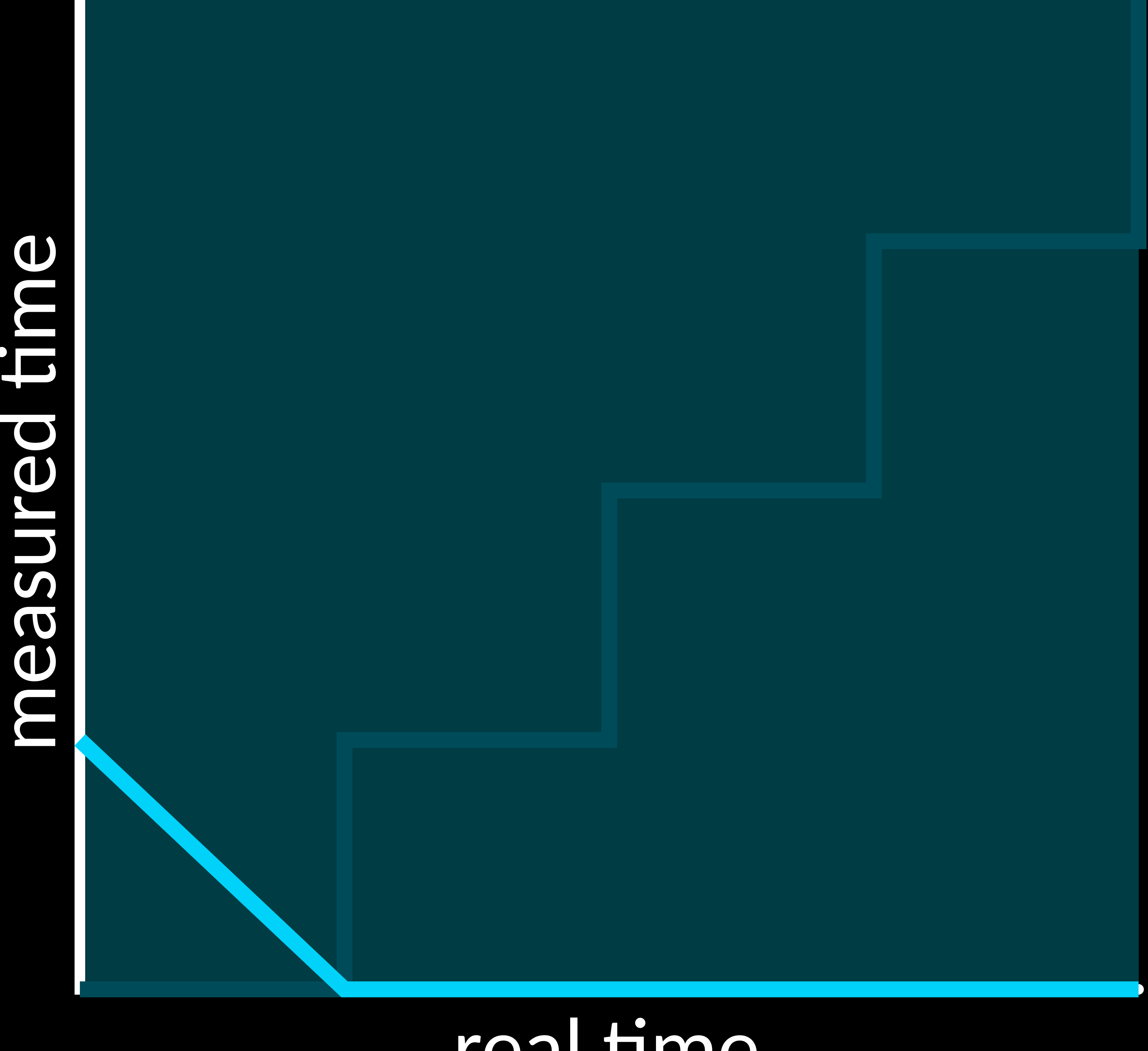

after anti- side-channel mitigations (firefox)

```
c = 0;
t0 = p.now();
while(t0 == p.now());
t1 = p.now();

operation();

while(t1 == p.now())
{ c++; }
```



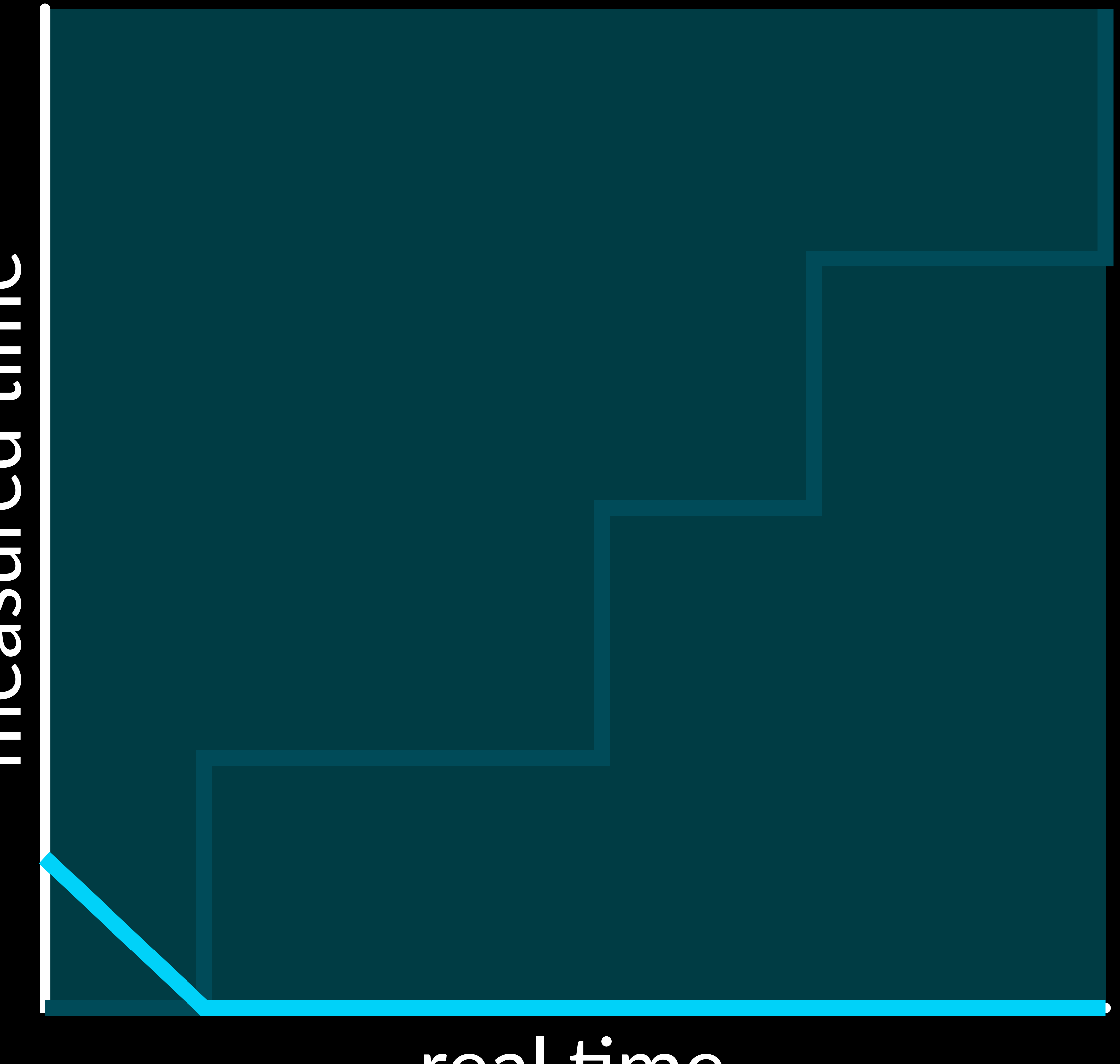

after anti- side-channel mitigations (firefox)

```
c = 0;
t0 = p.now();
while(t0 == p.now());
t1 = p.now();

operation();

while(t1 == p.now())
{ c++; }
```




after anti- side-channel mitigations (firefox)

```
c = 0;
t0 = p.now();
while(t0 == p.now());
t1 = p.now();

operation();

while(t1 == p.now())
{ c++; }
```




after anti- side-channel mitigations (chrome)

```
new SharedArrayBuffer()
```

# new SharedArrayBuffer()

memory which may be shared between multiple worker threads.

# new SharedArrayBuffer()

memory which may be shared between multiple worker threads.

enabled by default by Firefox, Chrome and Edge since 2017

```
let SharedRowhammerBuffer =
      SharedArrayBuffer;
```

```
c=0;
while (buf[0] == 0);

while (buf[0] == 1)
{ c++; }
```
1

```
buf[0]=1;
operation();
buf[0]=0;
```
2

using SharedArrayBuffer and worker threads

```
c=0;
● while (buf[0] == 0);

while (buf[0] == 1)
{ c++; }
```
1

```
buf[0]=1;
operation();
buf[0]=0;
```
2

using SharedArrayBuffer and worker threads

```
c=0;
● while (buf[0] == 0);

while (buf[0] == 1)
{ c++; }
```
1

```
● buf[0]=1;
operation();
buf[0]=0;
```
2

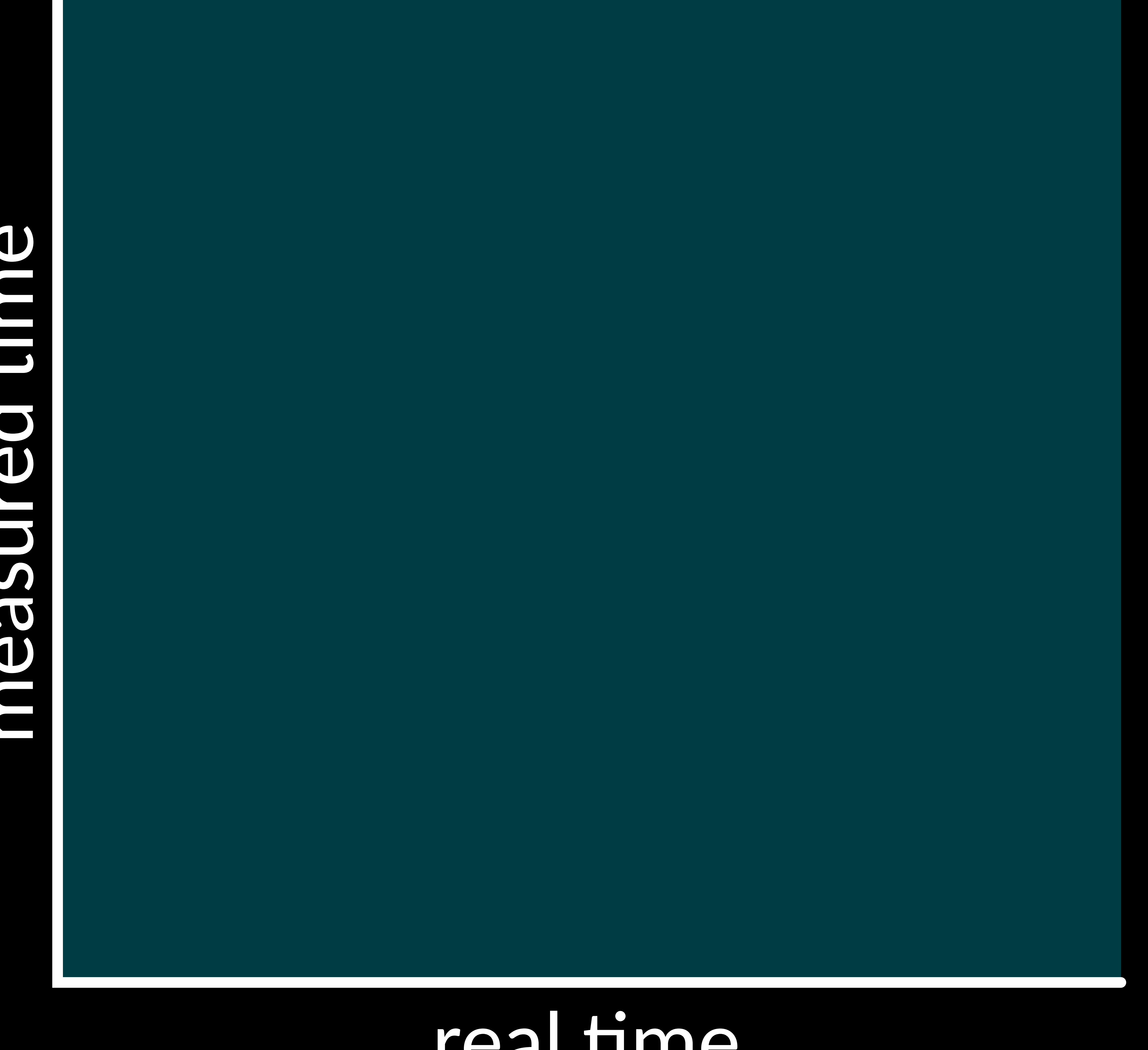
using SharedArrayBuffer and worker threads

```
c=0;
● while (buf[0] == 0);

while (buf[0] == 1)
{ c++; }
```

1

```
buf[0]=1;
● operation();
buf[0]=0;
```

2




using SharedArrayBuffer and worker threads

```
c=0;                                    1
while (buf[0] == 0);

● while (buf[0] == 1)
{ c++; }
```

```
buf[0]=1;                               2
● operation();
buf[0]=0;
```

using SharedArrayBuffer and worker threads

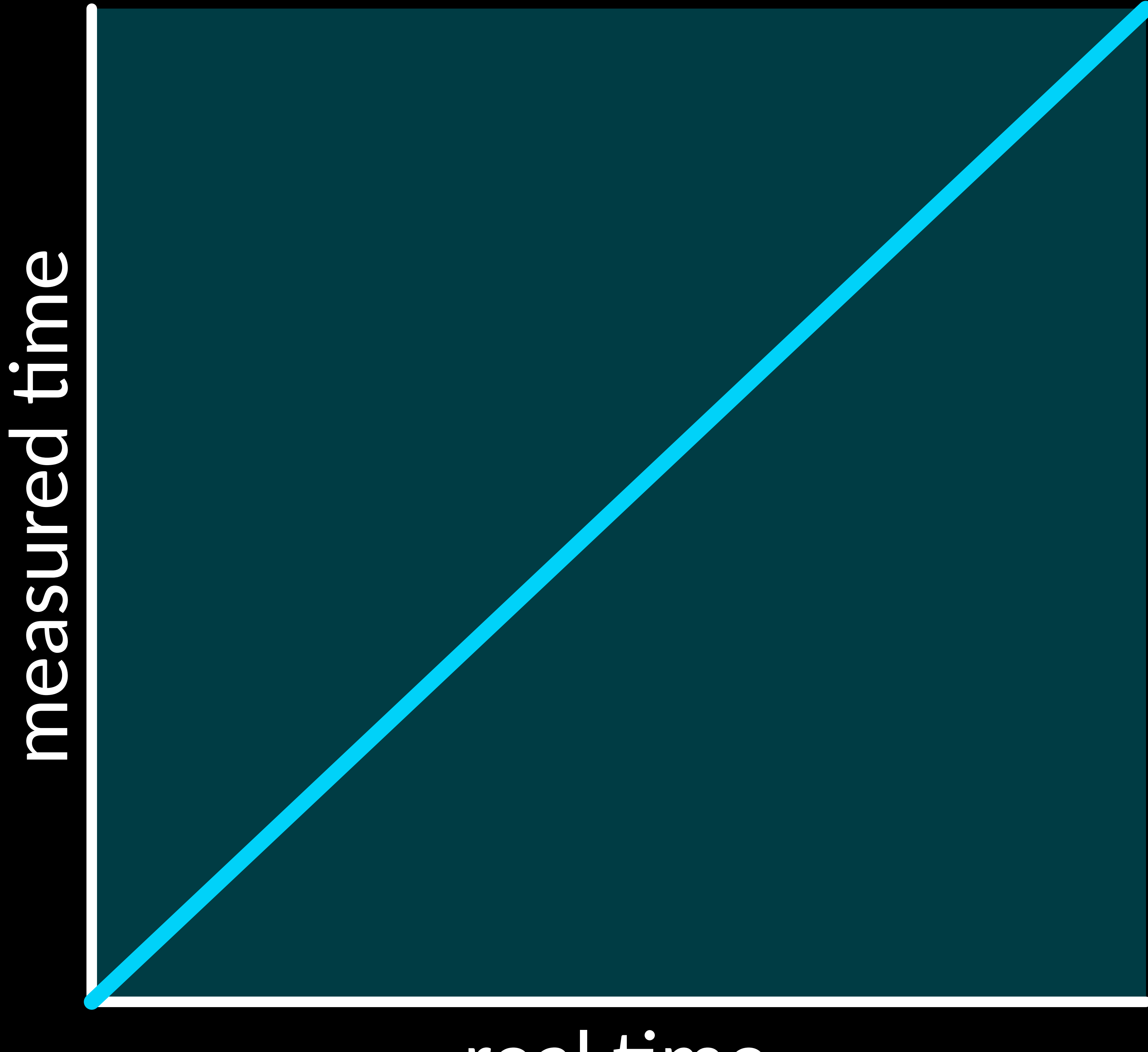
```
1
c=0;
while (buf[0] == 0);

● while (buf[0] == 1)
{ c++; }
```

```
2
buf[0]=1;
operation();
● buf[0]=0;
```

using SharedArrayBuffer and worker threads

```
c=0;                                  1
while (buf[0] == 0);

● while (buf[0] == 1)
{ c++; }
```

```
buf[0]=1;                             2
operation();
● buf[0]=0;
```




using SharedArrayBuffer and worker threads

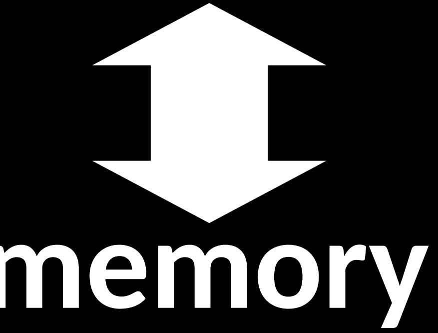
# Cache Side-Channels

memory access

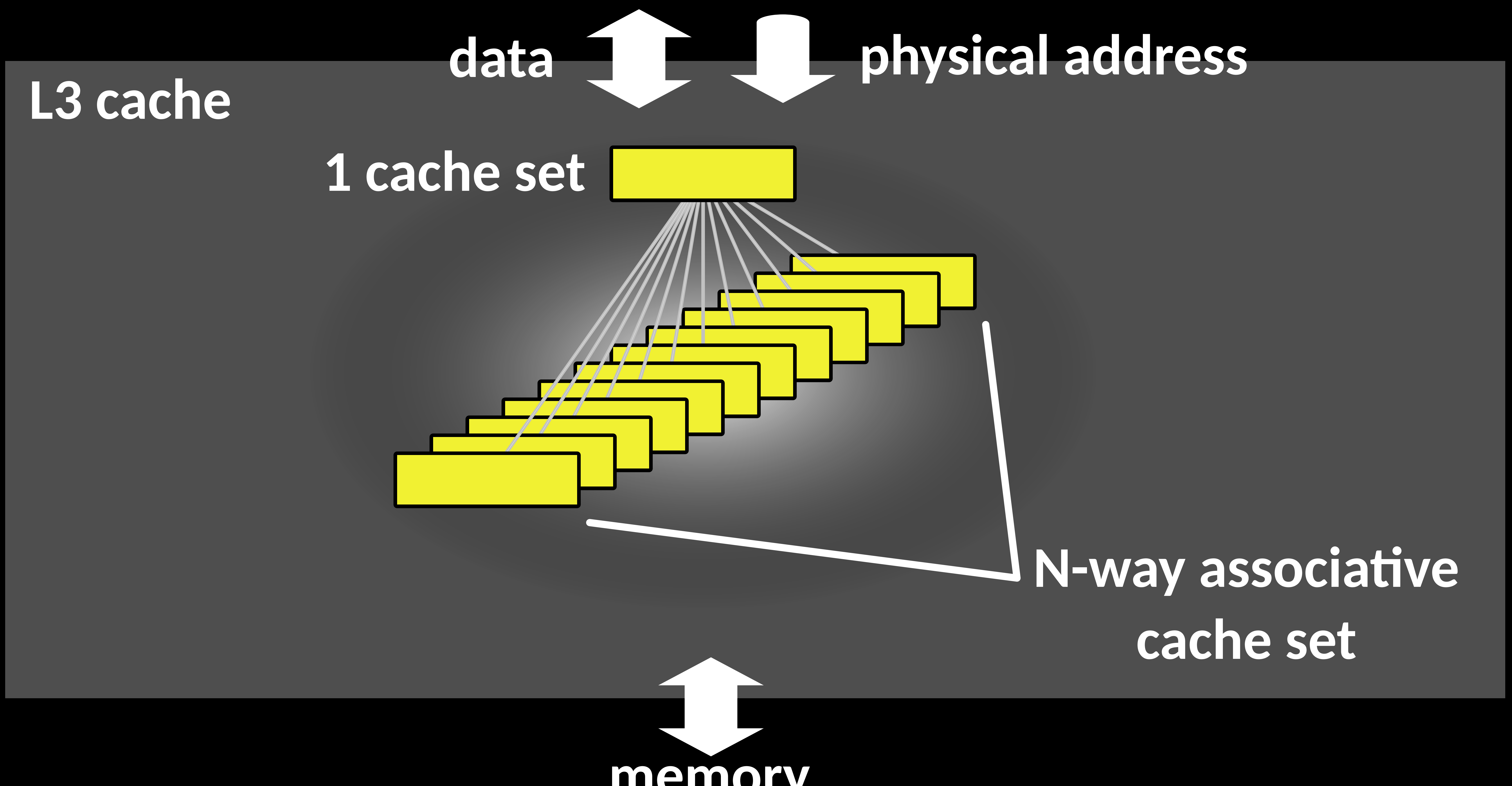
data

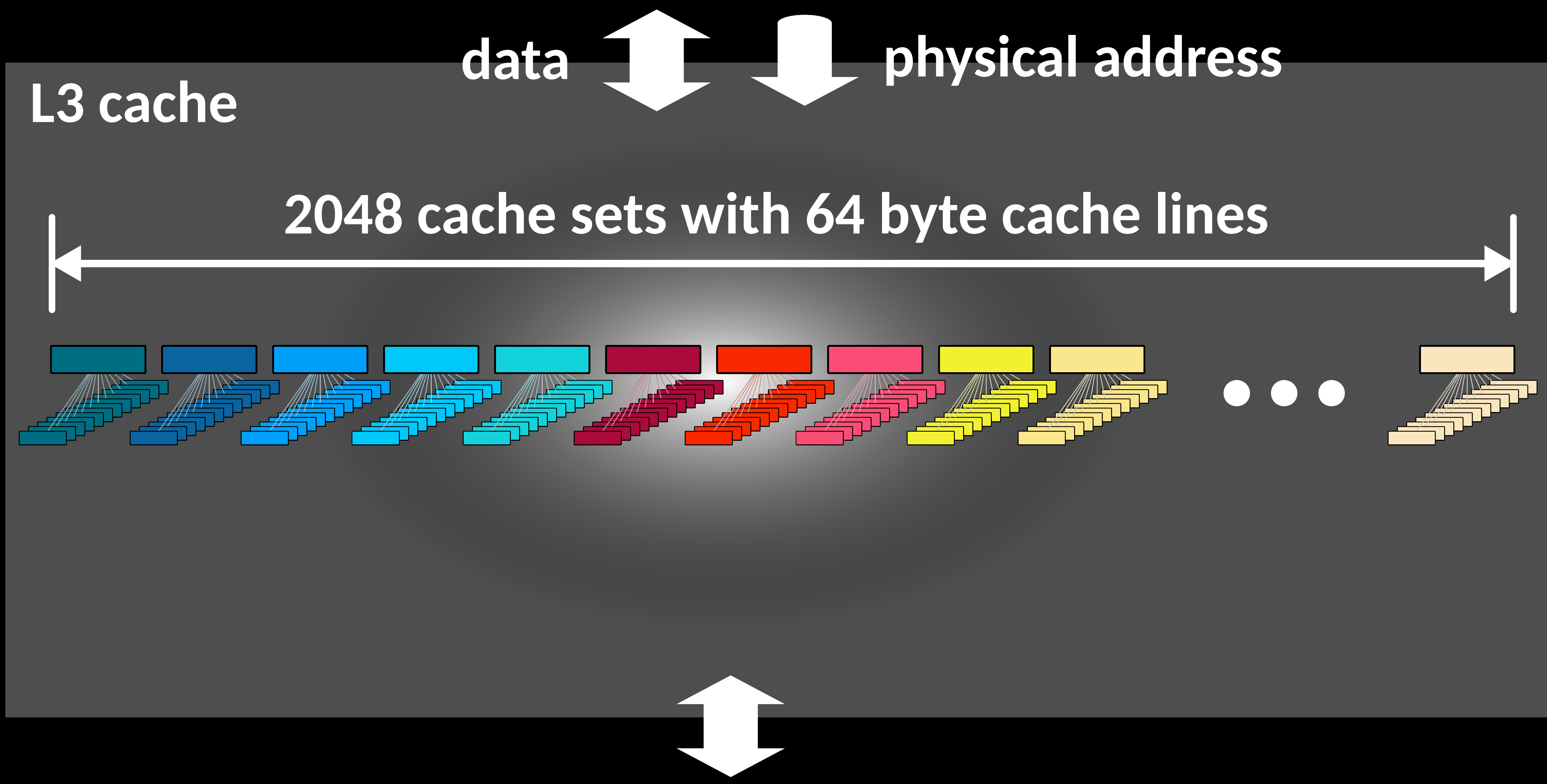
physical address

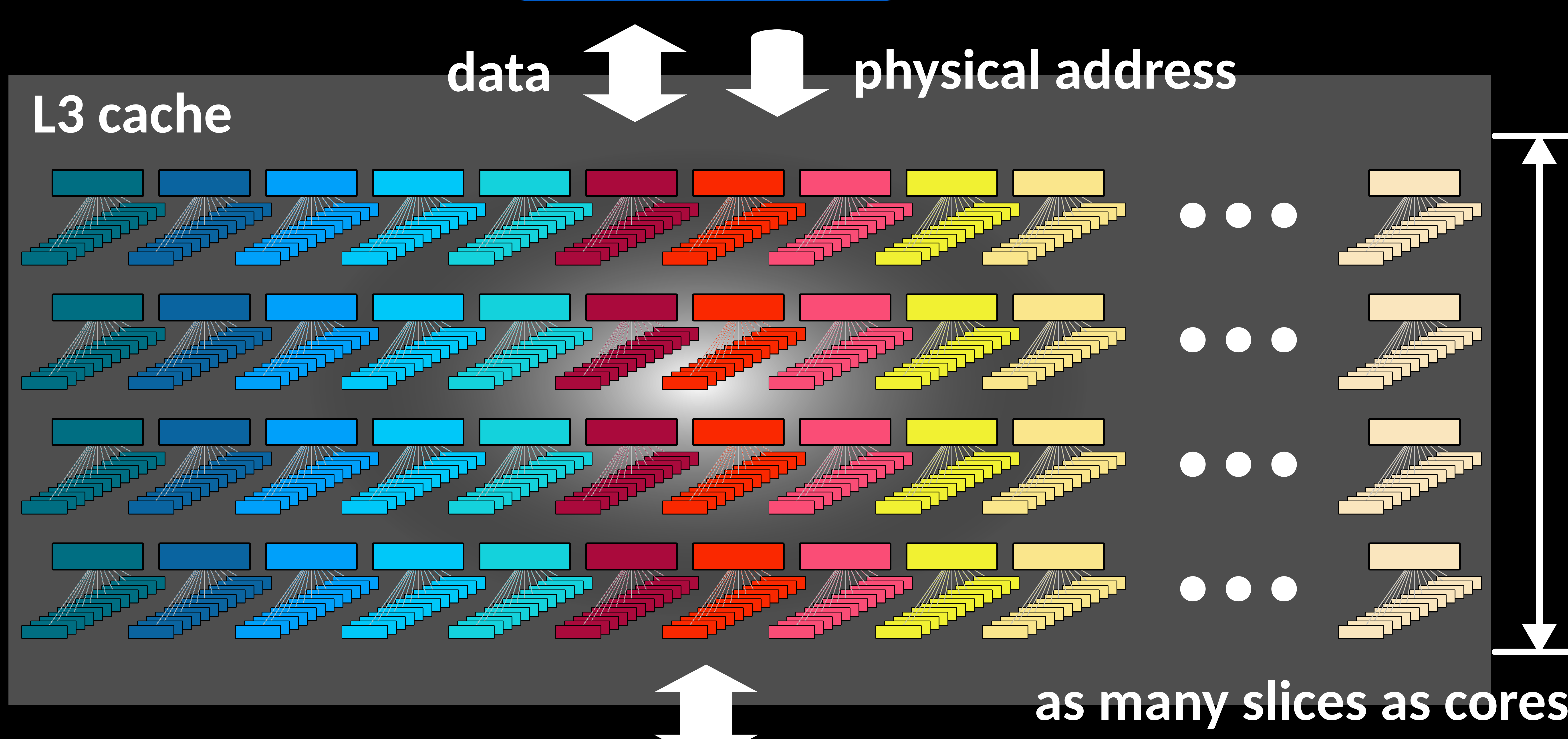
cache line (64 bytes)



memory


memory access
data
physical address
L3 cache
1 cache set
N-way associative
cache set
memory


memory access
data
physical address
L3 cache
2048 cache sets with 64 byte cache lines
memory

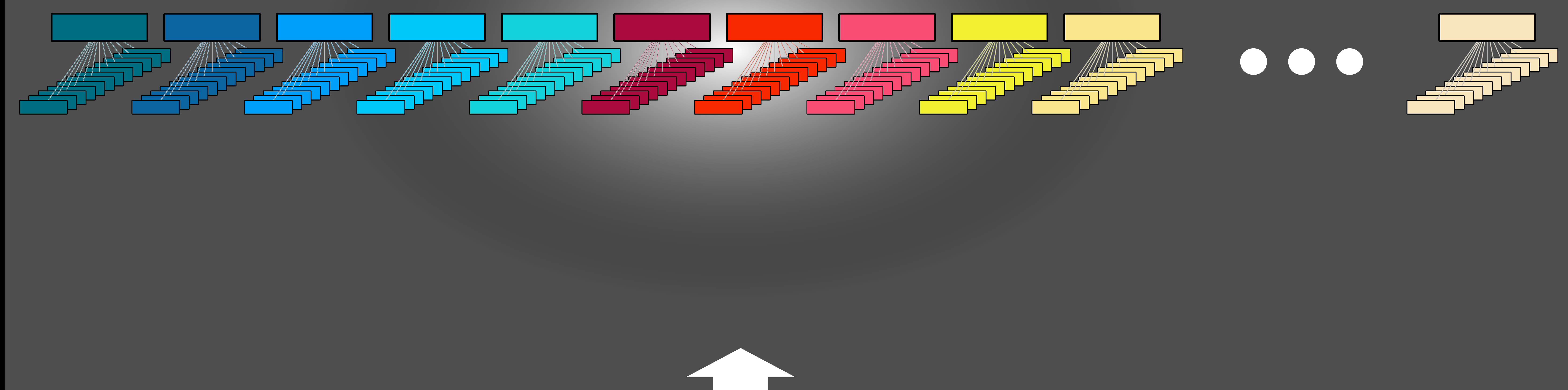

memory access
data
physical address
L3 cache
as many slices as cores
memory

memory access

data

physical address

L3 cache



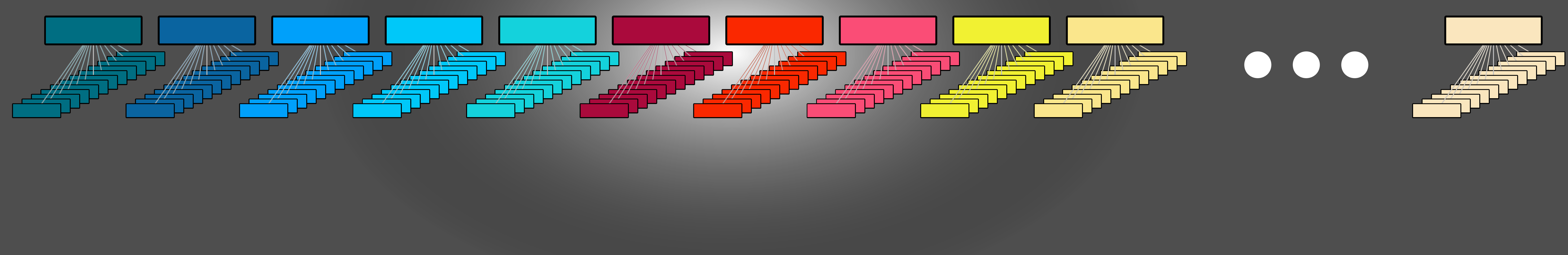
memory

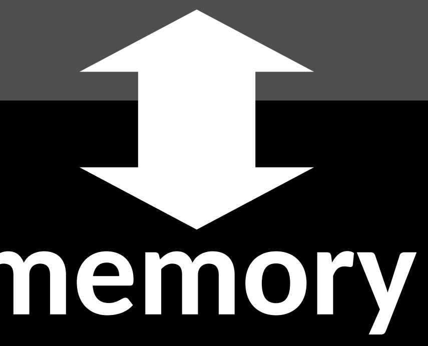
memory access

L3 cache

cache_set = (addr >> 6) % 2048,

direct mapping,
repeated every 128KB



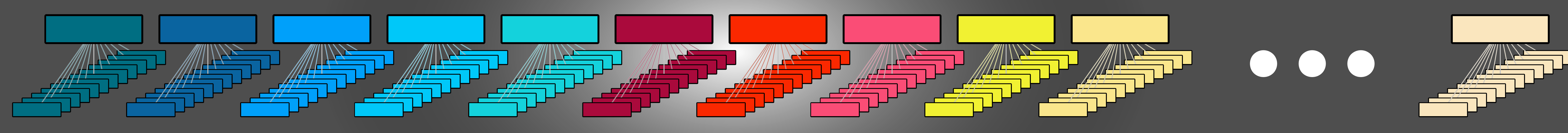


memory access

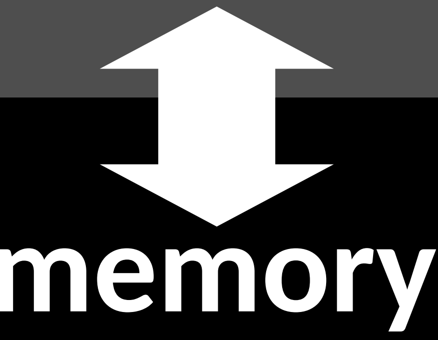
L3 cache

cache_set = (addr >> 6) % 2048,

direct mapping,
repeated every 128KB



cache_slice = xor_hash(addr)



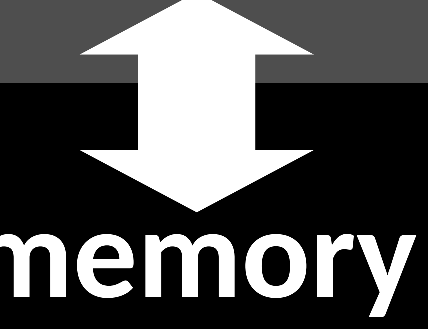
memory

memory access

L3 cache

cache_set = (addr >> 6) % 2048,

direct mapping,
repeated every 128KB



memory

memory access

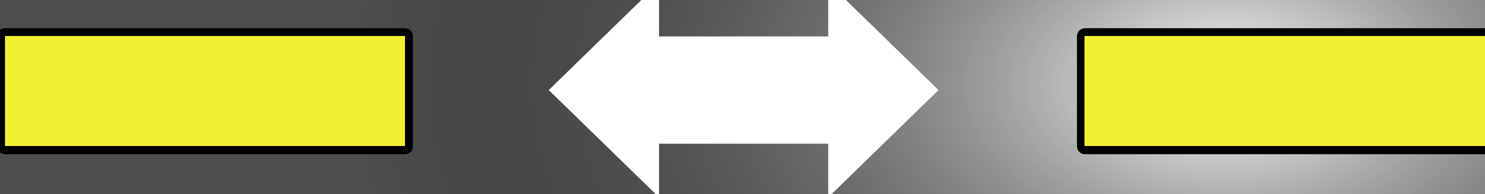
data

physical address

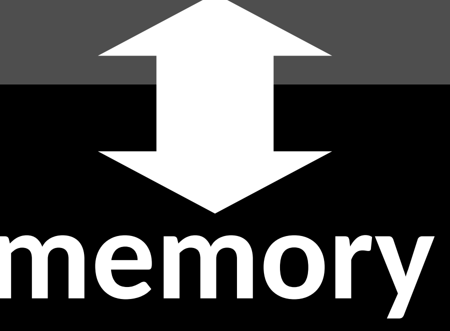
L3 cache

cache_set = (addr >> 6) % 2048,

direct mapping,
repeated every 128KB



two cache lines mapping to the same cache set
have the same physical address modulo 128KB



memory

memory access

data

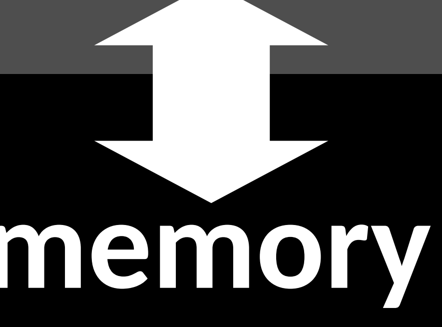
physical address

L3 cache

cache_set = (addr >> 6) % 2048,

direct mapping,
repeated every 128KB

two cache lines mapping to the same cache set
have the same physical address modulo 4KB



memory

memory access

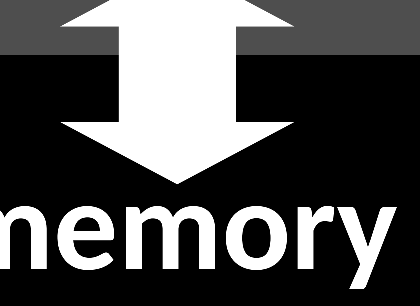
data

physical address

L3 cache

two cache lines mapping to the same cache set have the same offset into their memory page

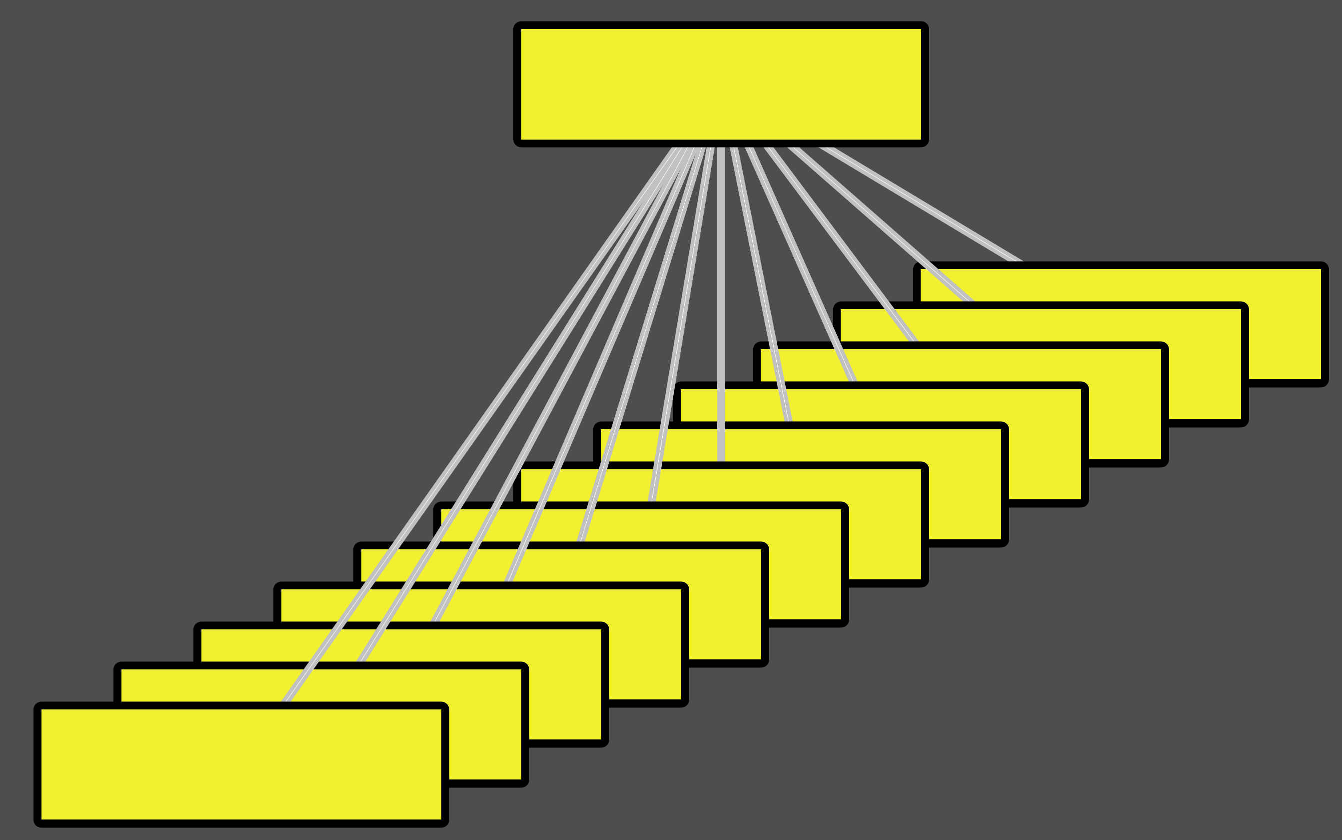
1 page =
64 cache lines



memory

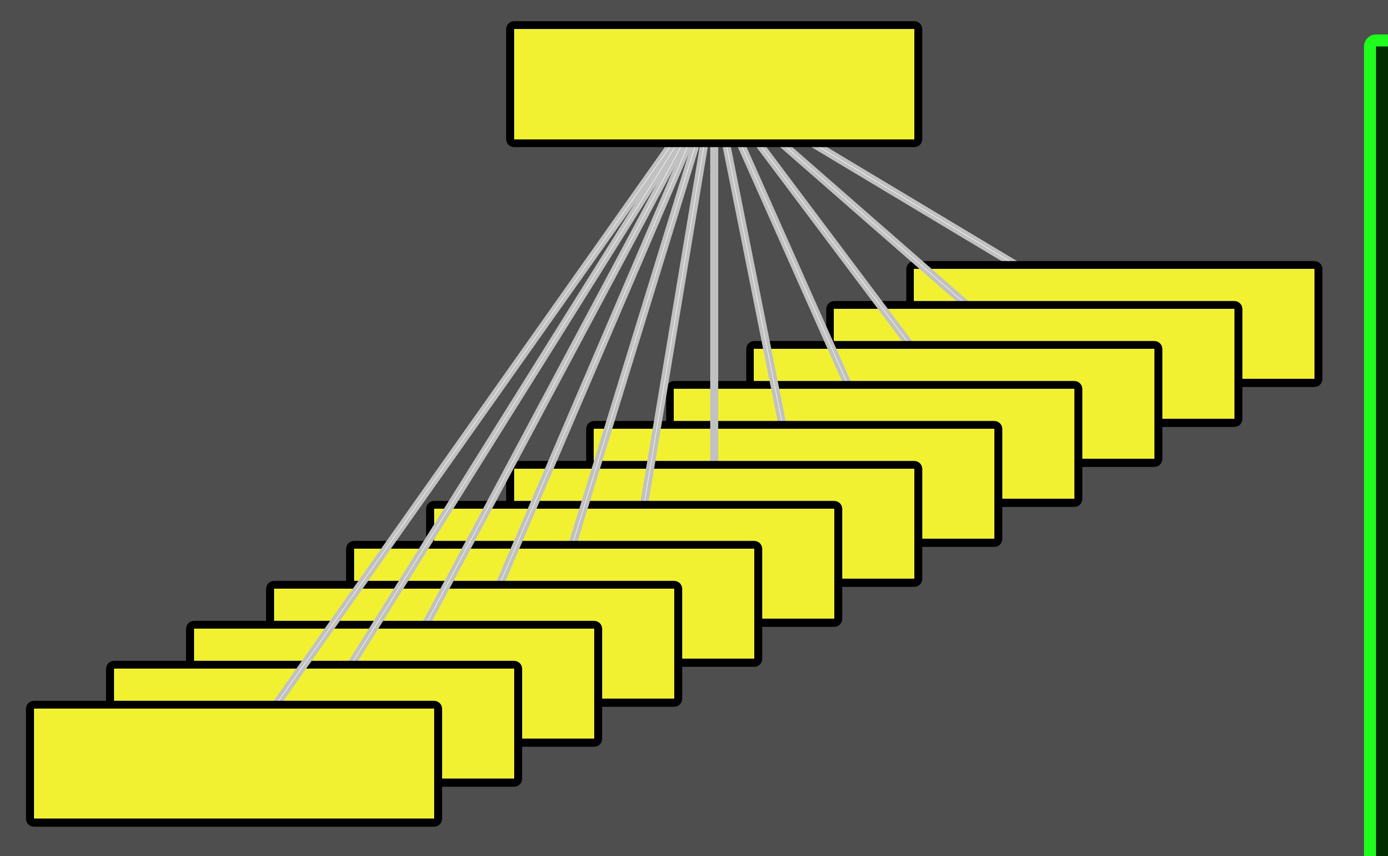
L3 cache

# EVICT + TIME
(does an operation use a specific cache line?)



L3 cache

## EVICT + TIME
## (does an operation use a specific cache line?)

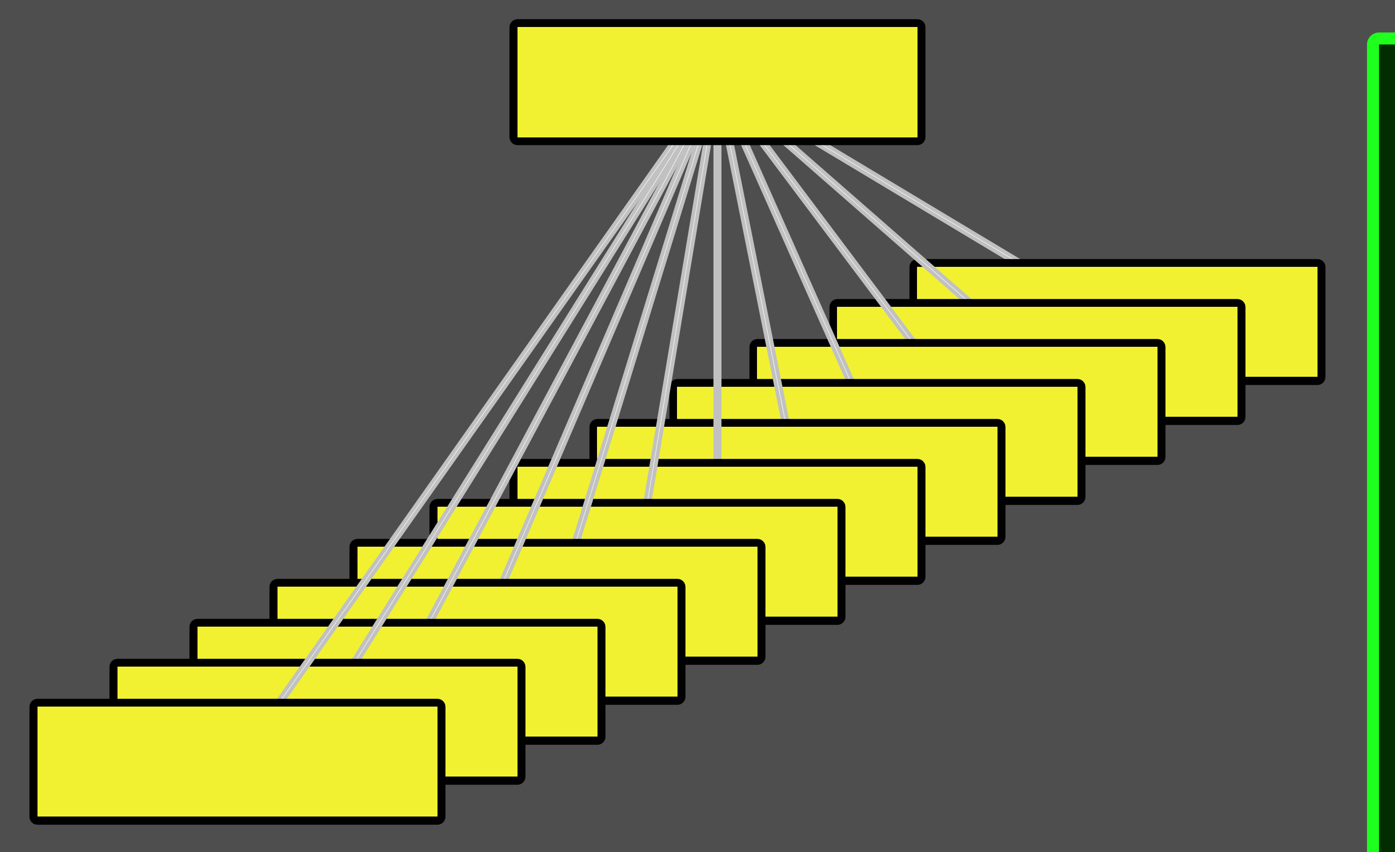


```
evict(line_x);
time();
t0 = time();
operation();
t = time()-t0;
```

L3 cache

## EVICT + TIME

(does an operation use a specific cache line?)



```
• evict(line_x);
  time();
  t0 = time();
  operation();
  t = time()-t0;
```

L3 cache

## EVICT + TIME
(does an operation use a specific cache line?)

mybuf

X

X

X

⋮

X

```
evict(line_x);
time();
t0 = time();
operation();
t = time()-t0;
```

L3 cache

# EVICT + TIME

(does an operation use a specific cache line?)

mybuf

X

X

X

⋮

X

```
• evict(line_x);
  time();
  t0 = time();
  operation();
  t = time()-t0;
```

L3 cache

## EVICT + TIME
(does an operation use a specific cache line?)

mybuf

X

X

X

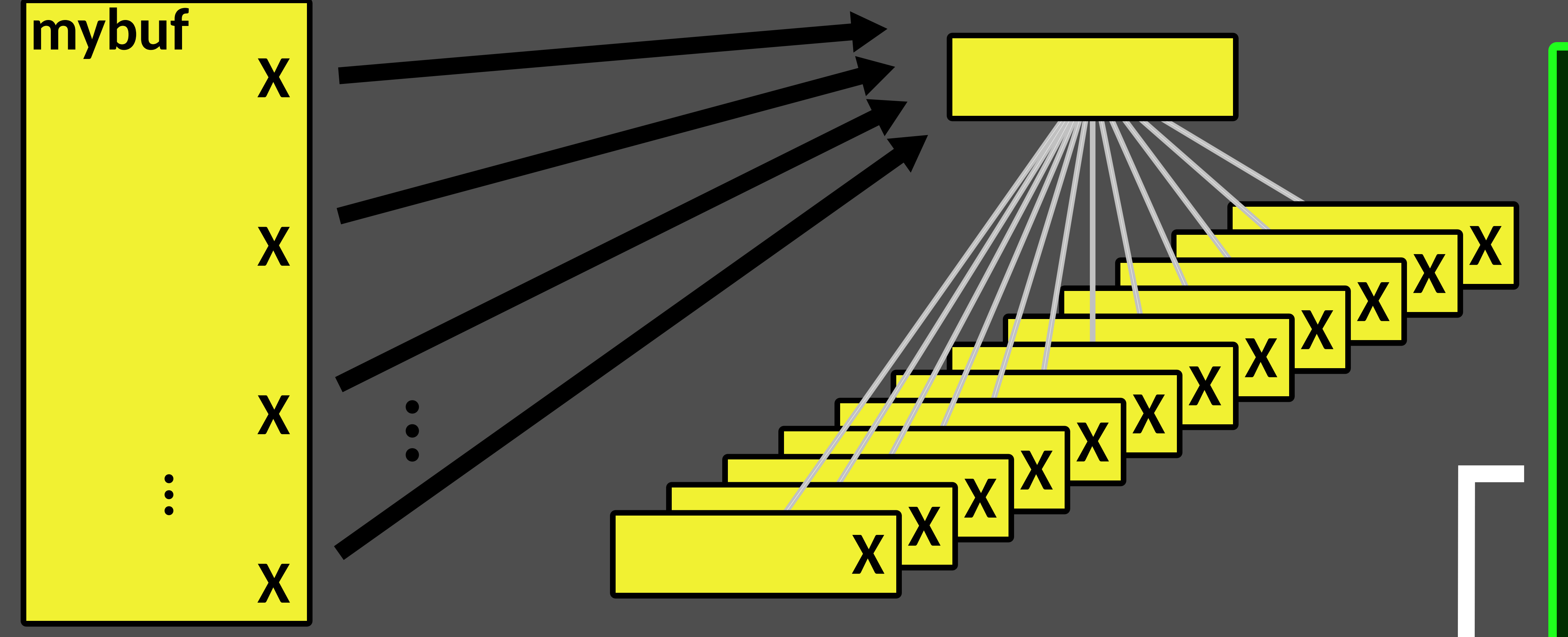
⋮

X

```
 evict(line_x);
 time();
 t0 = time();
•operation();
 t = time()-t0;
```

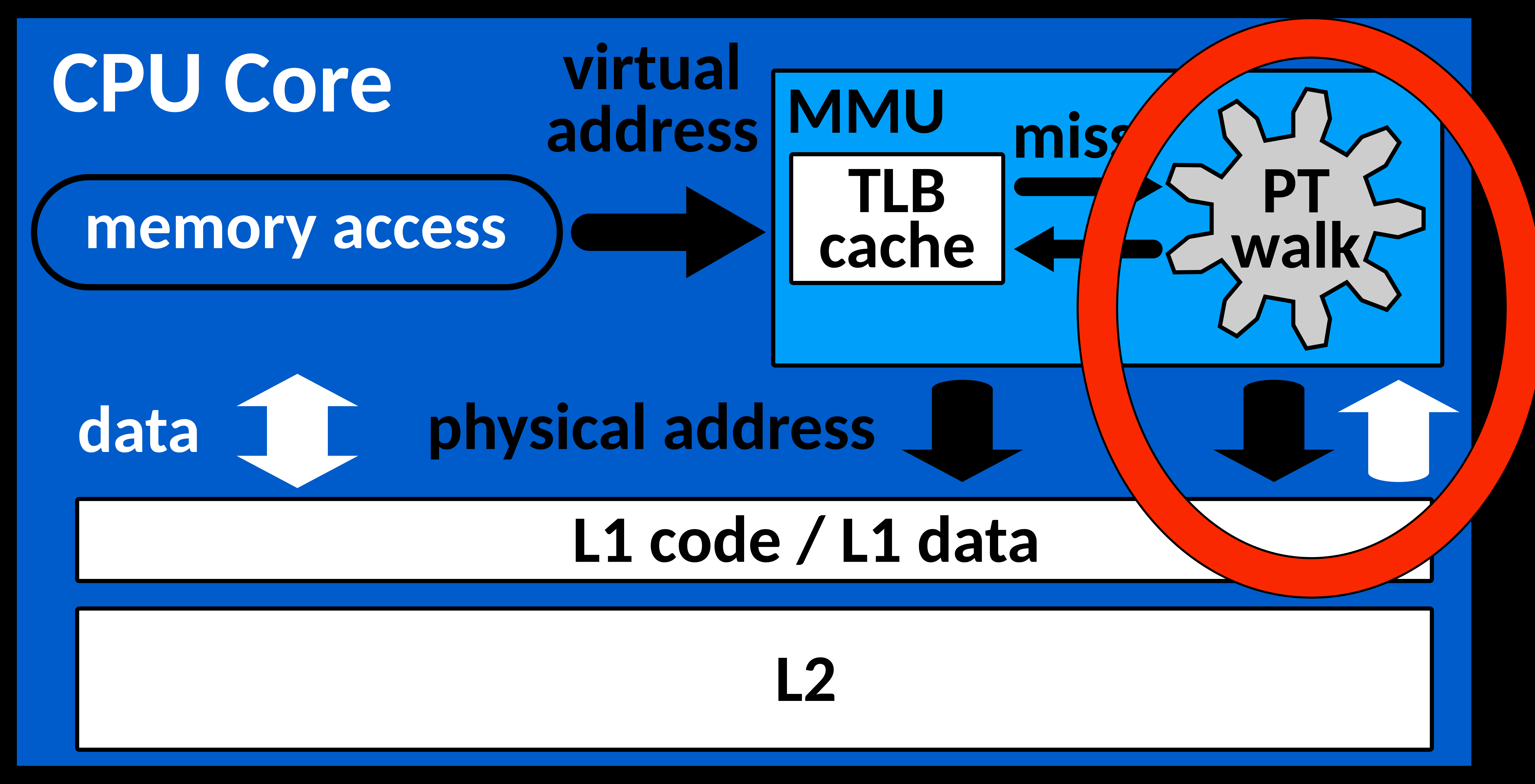
L3 cache

# EVICT + TIME
(does an operation use a specific cache line?)



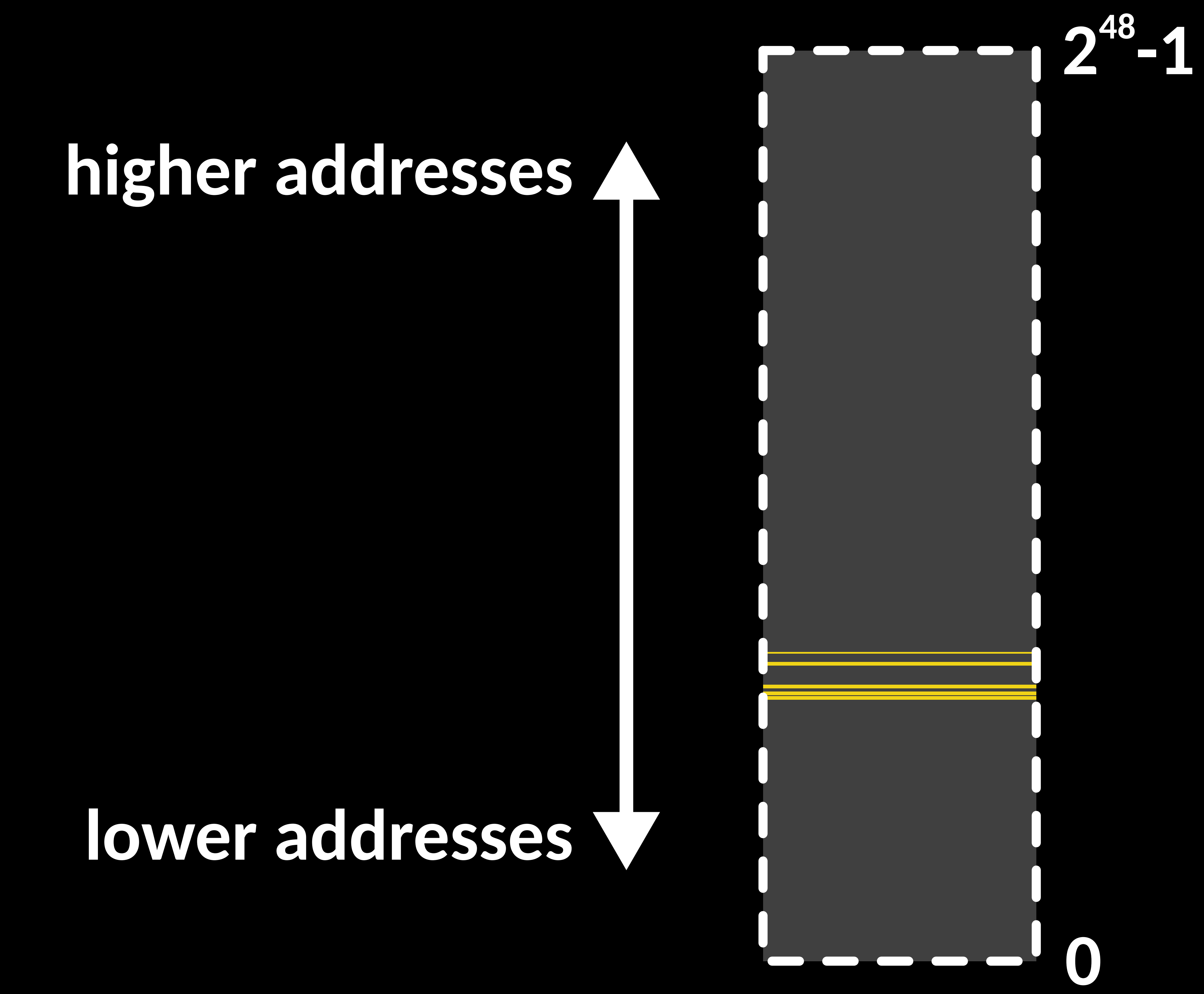

```
evict(line_x);
time();
t0 = time();
operation();
t = time()-t0;
```

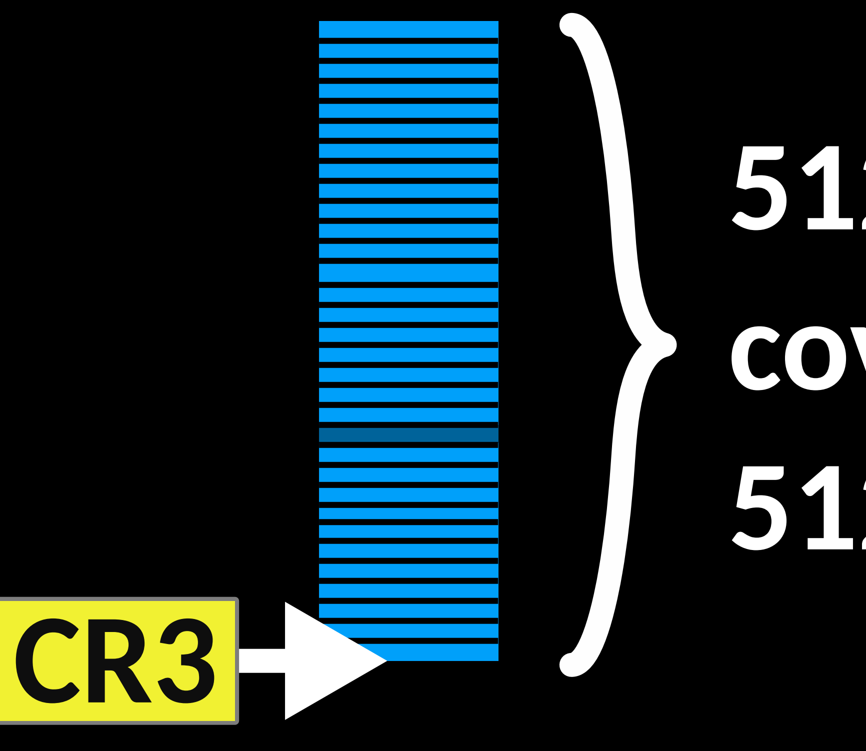
trigger memory access (or not)



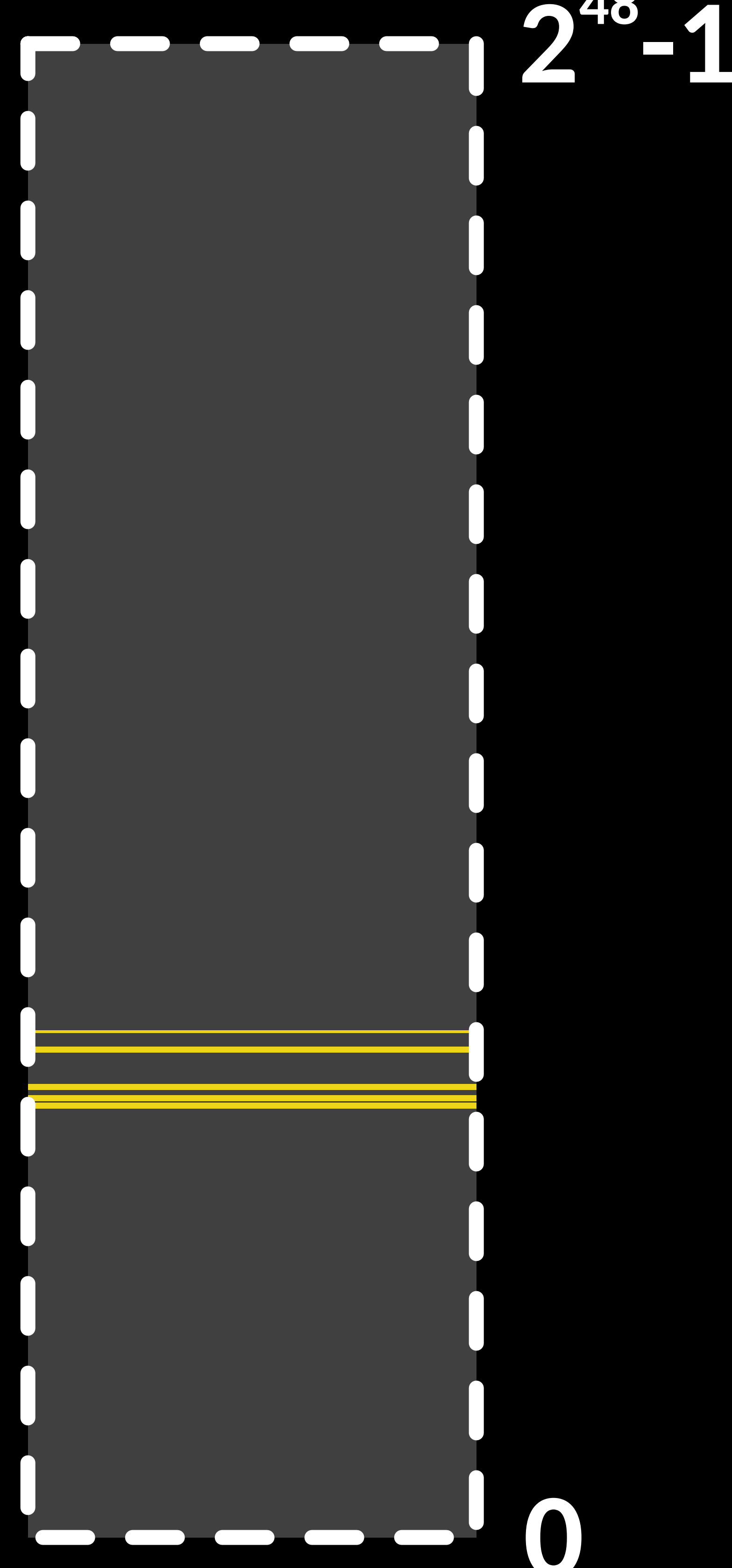

L3 (Last Level Cache), shared between cores

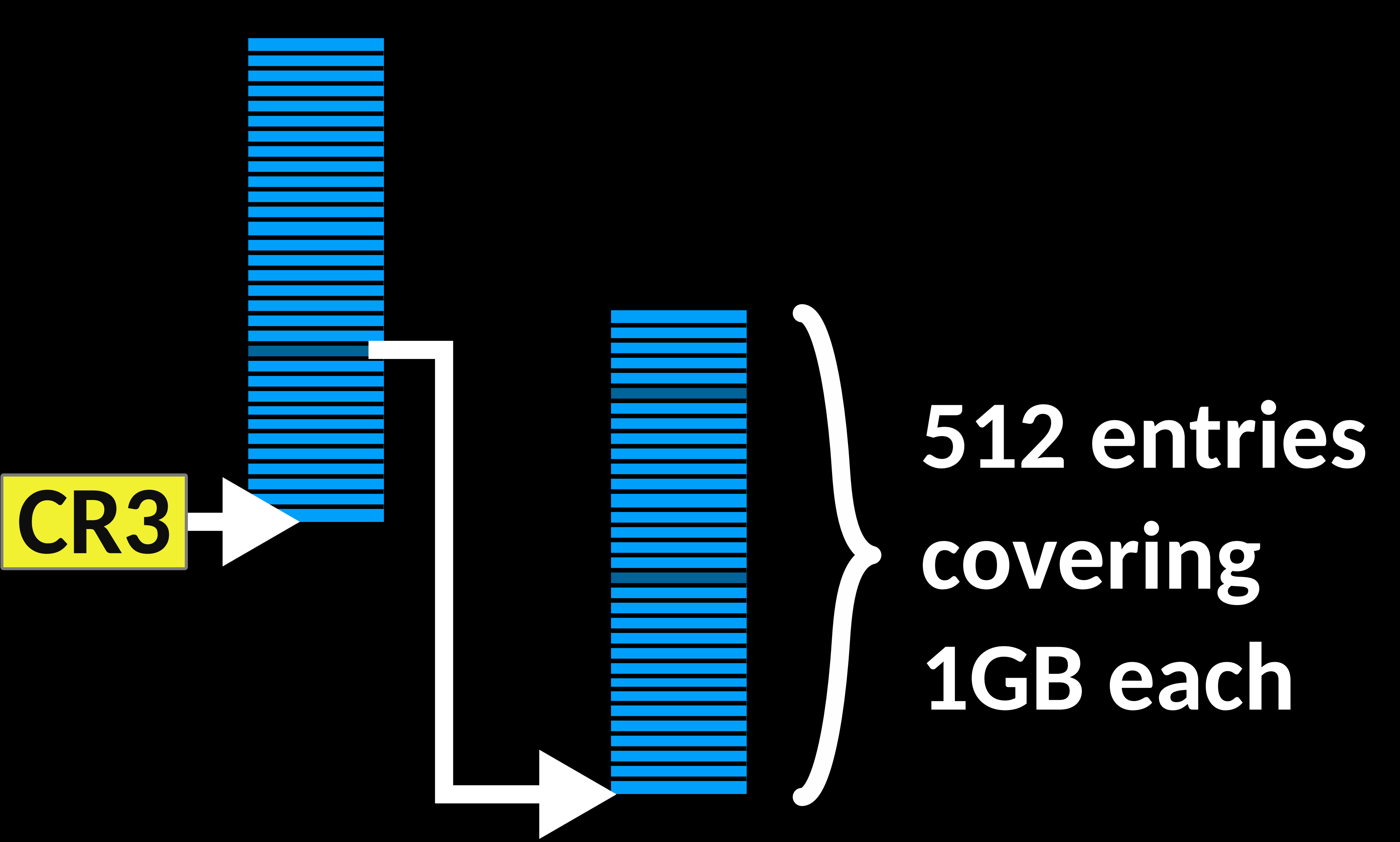
# Page Tables

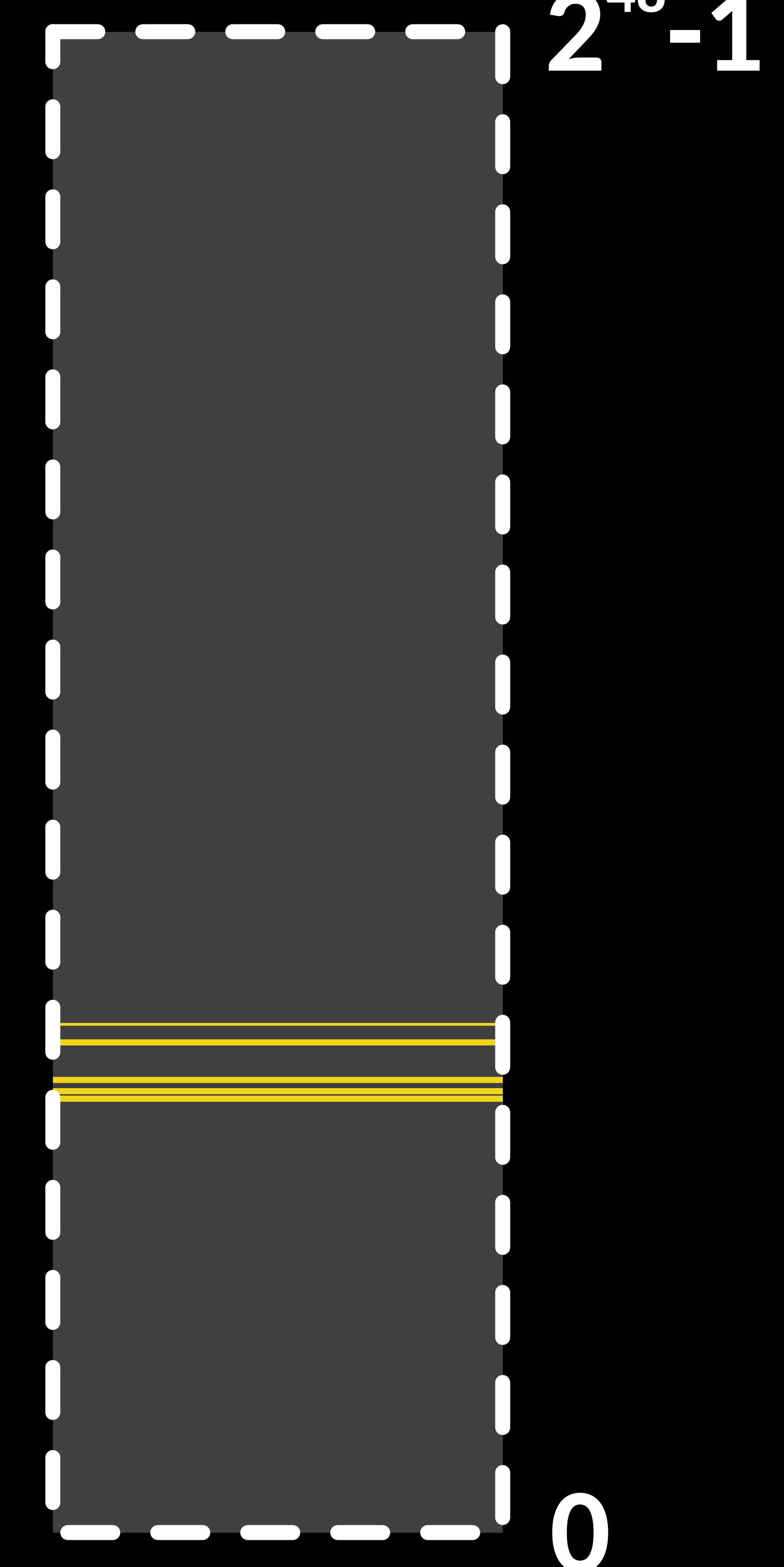

higher addresses
lower addresses
$2^{48}$-1
0

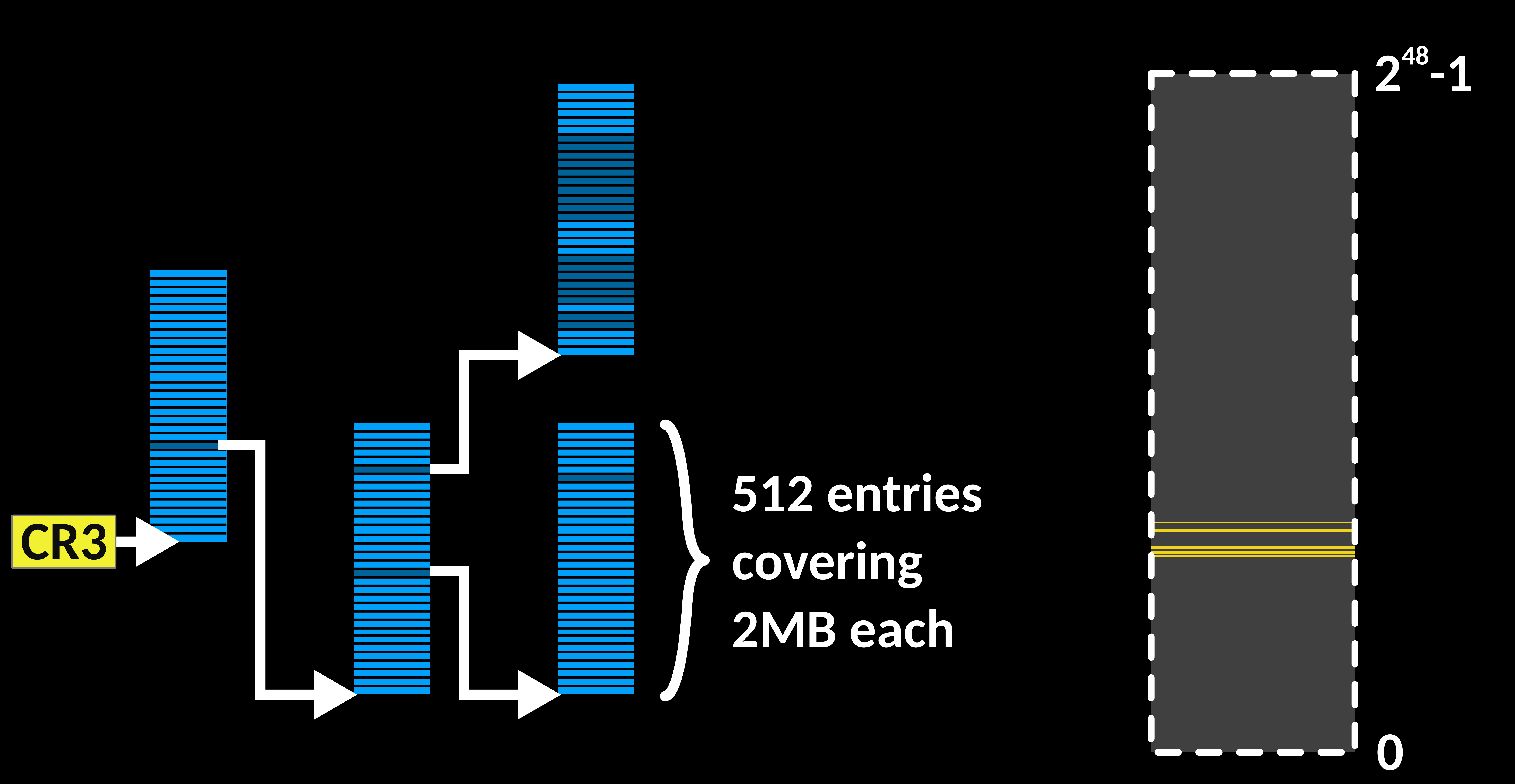


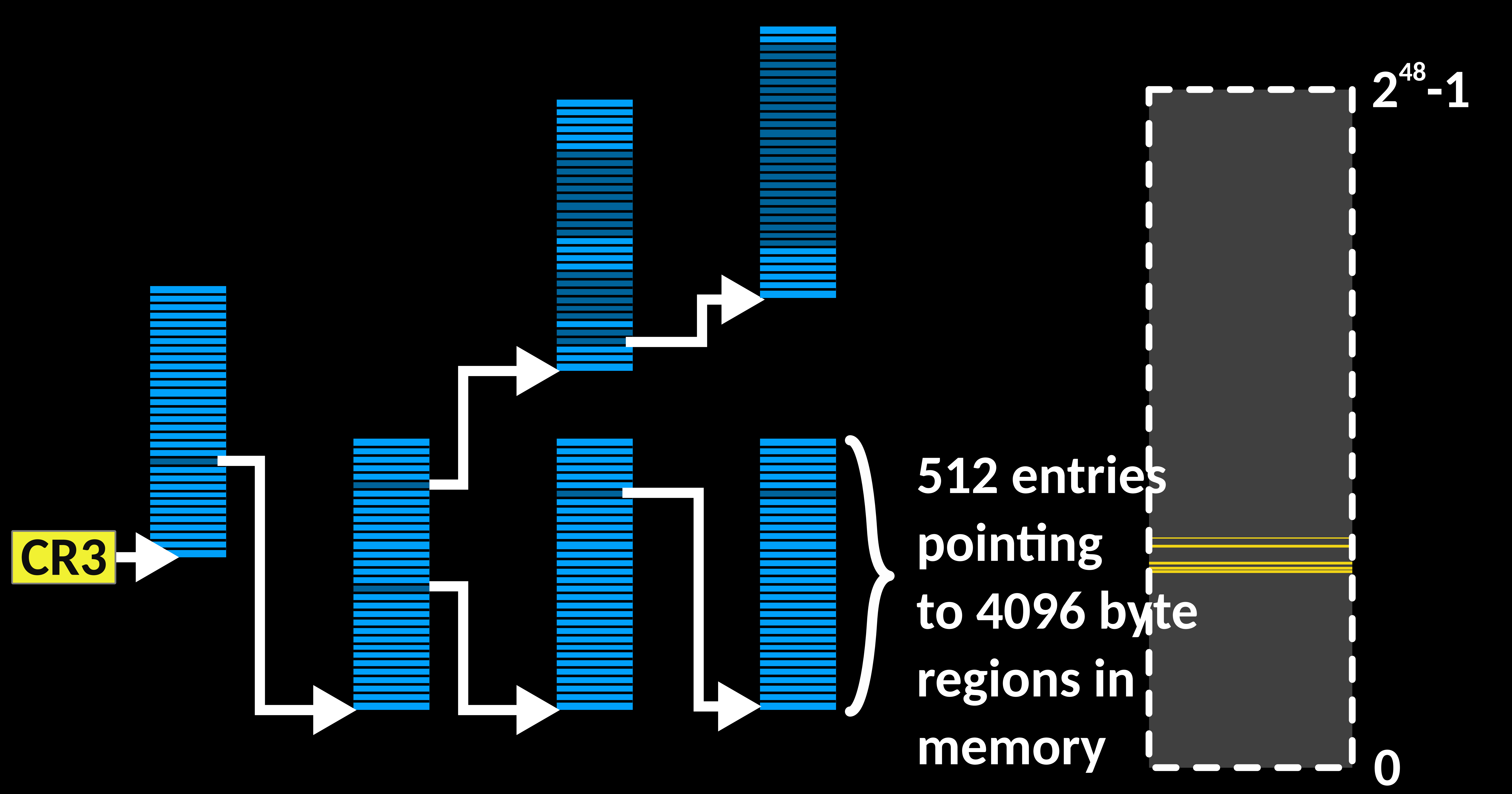

512 entries covering 512GB each

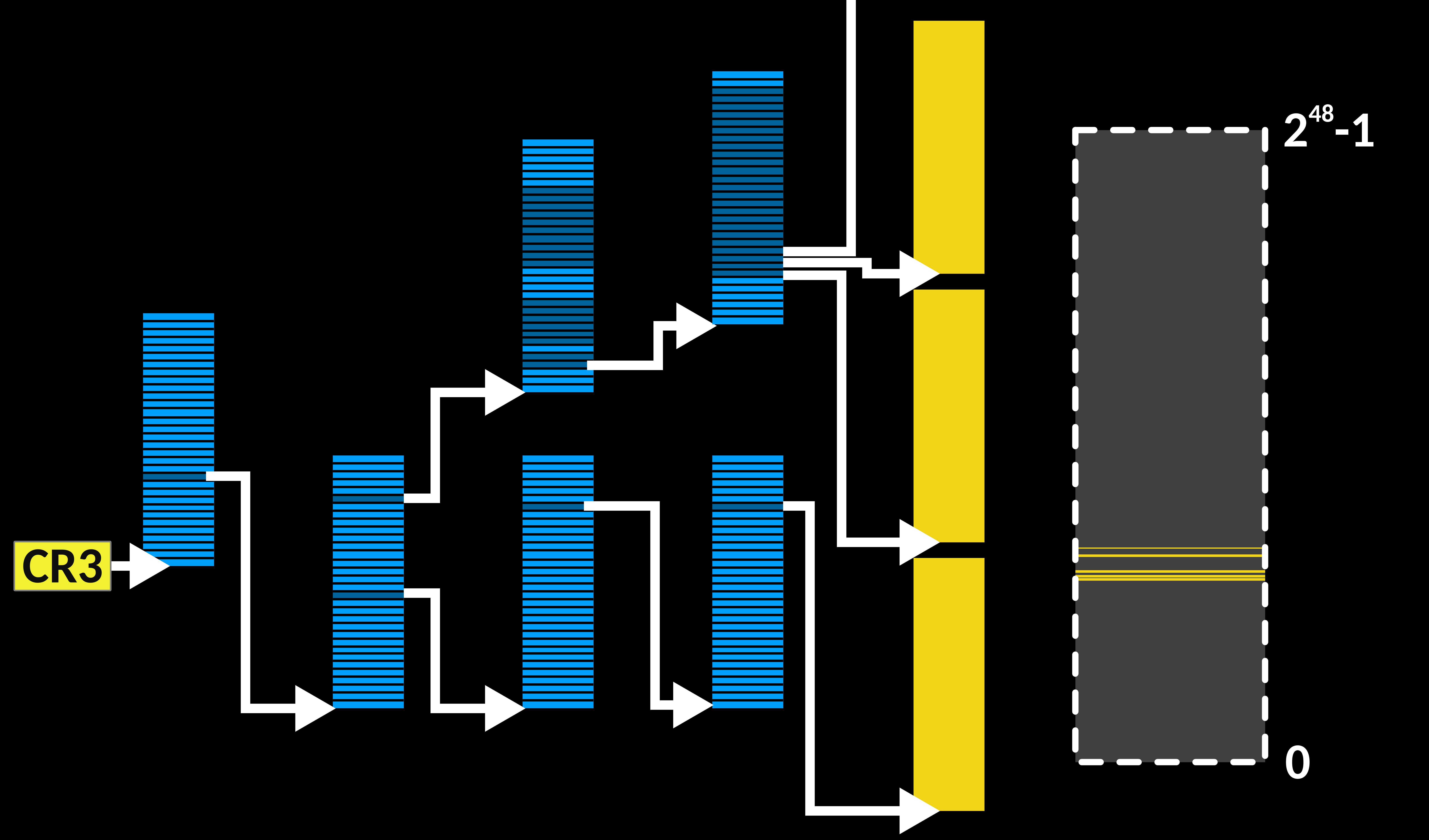



CR3
512 entries
covering
1GB each


$2^{48}$-1
0


CR3
512 entries
covering
2MB each
2^48-1
0

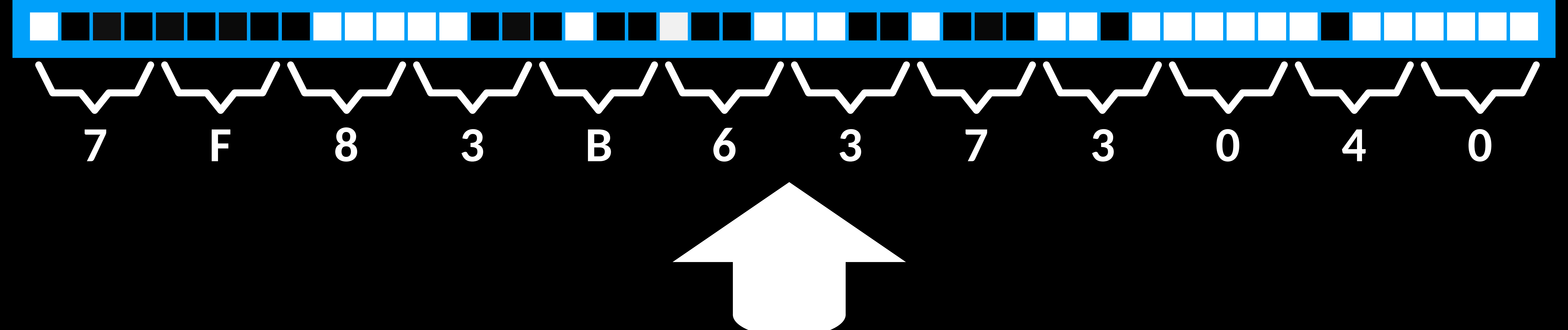

CR3
512 entries pointing to 4096 byte regions in memory
$2^{48}$-1
0



CR3
$2^{48}$-1
0

| 7 | F | 8 | 3 | B | 6 | 3 | 7 | 2 | 0 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|

virtual address lookup (x86_64)

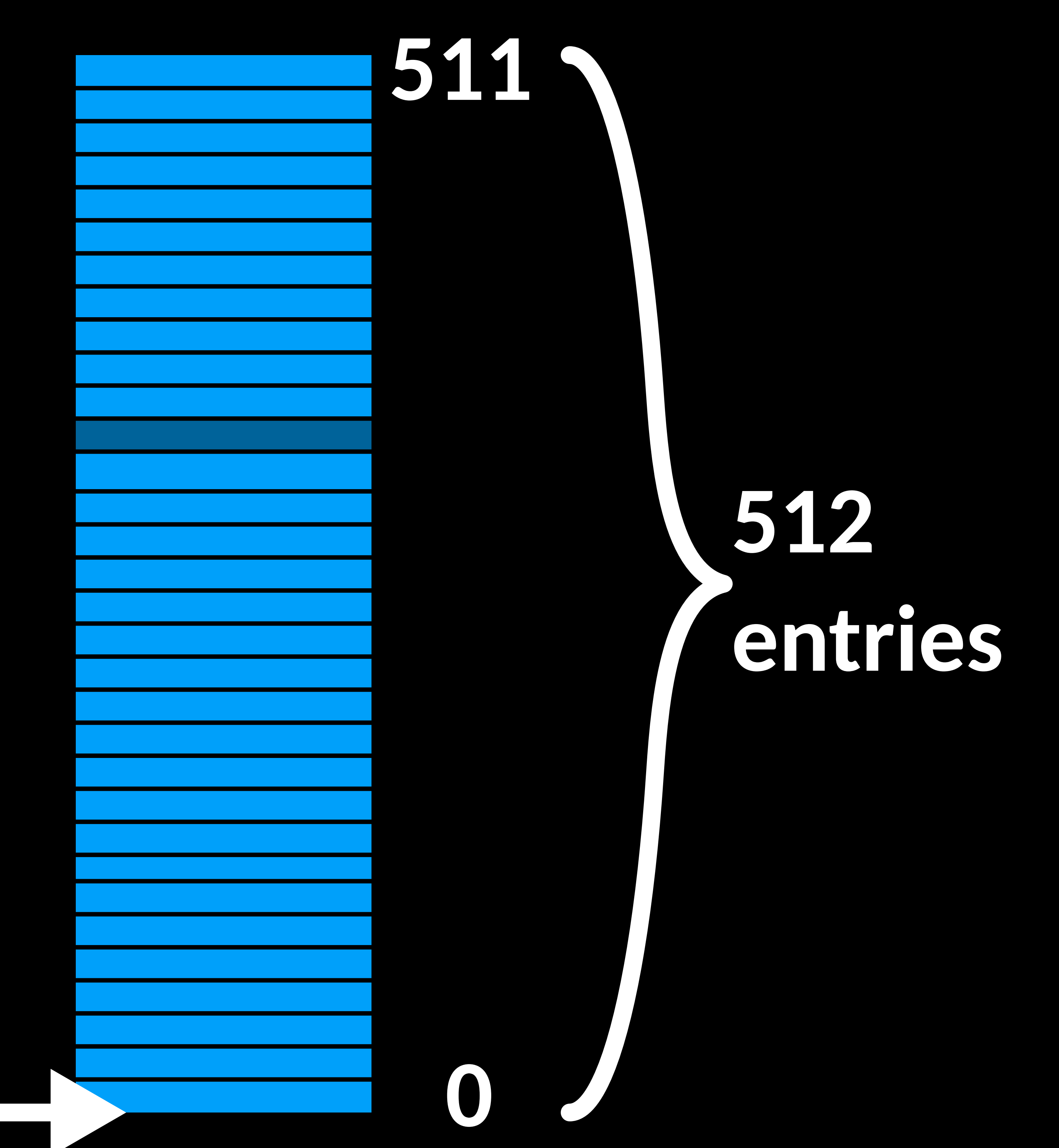



| 7 | F | 8 | 3 | B | 6 | 3 | 7 | 2 | 0 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|

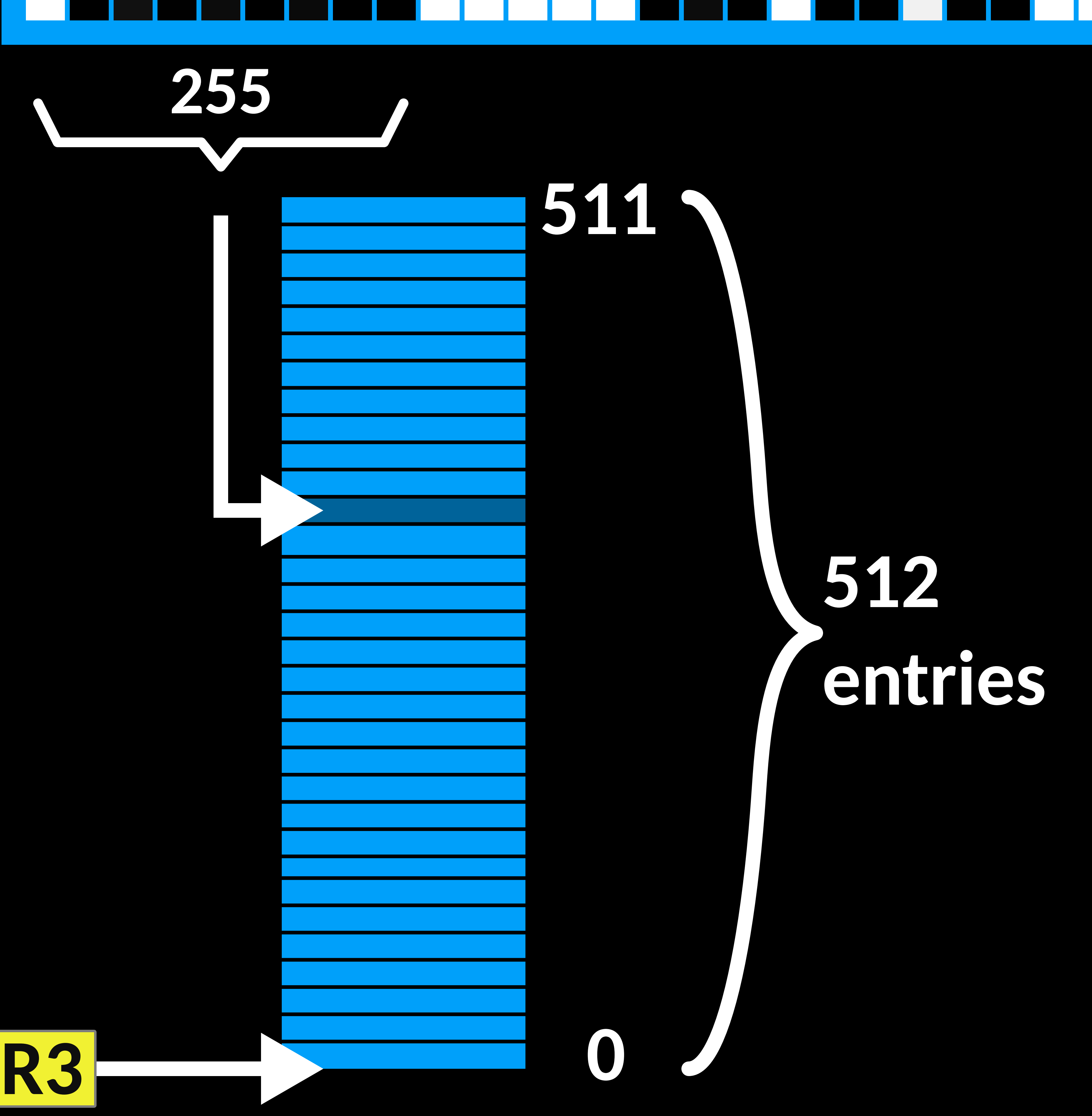
virtual address lookup (x86_64)

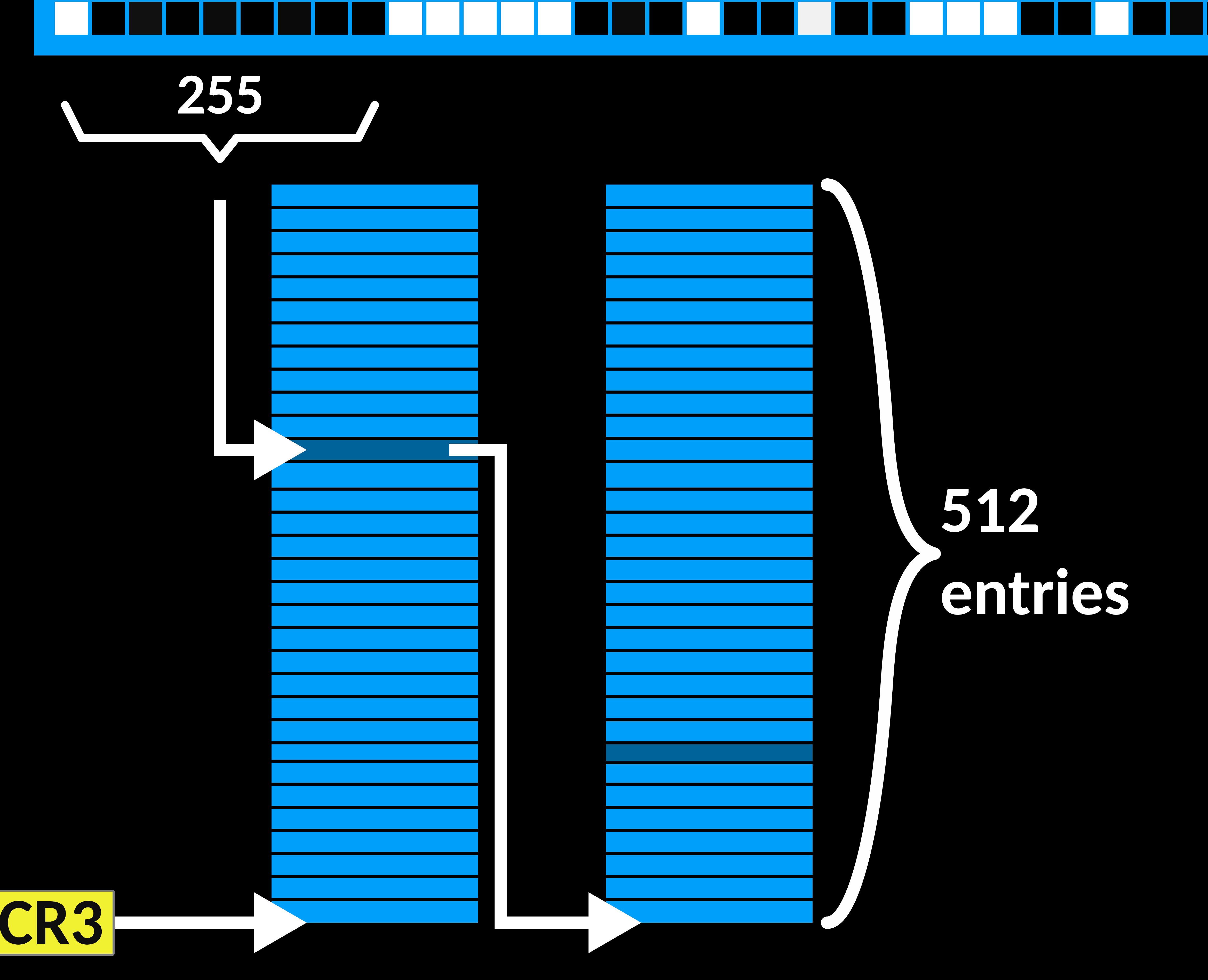
TLB miss!

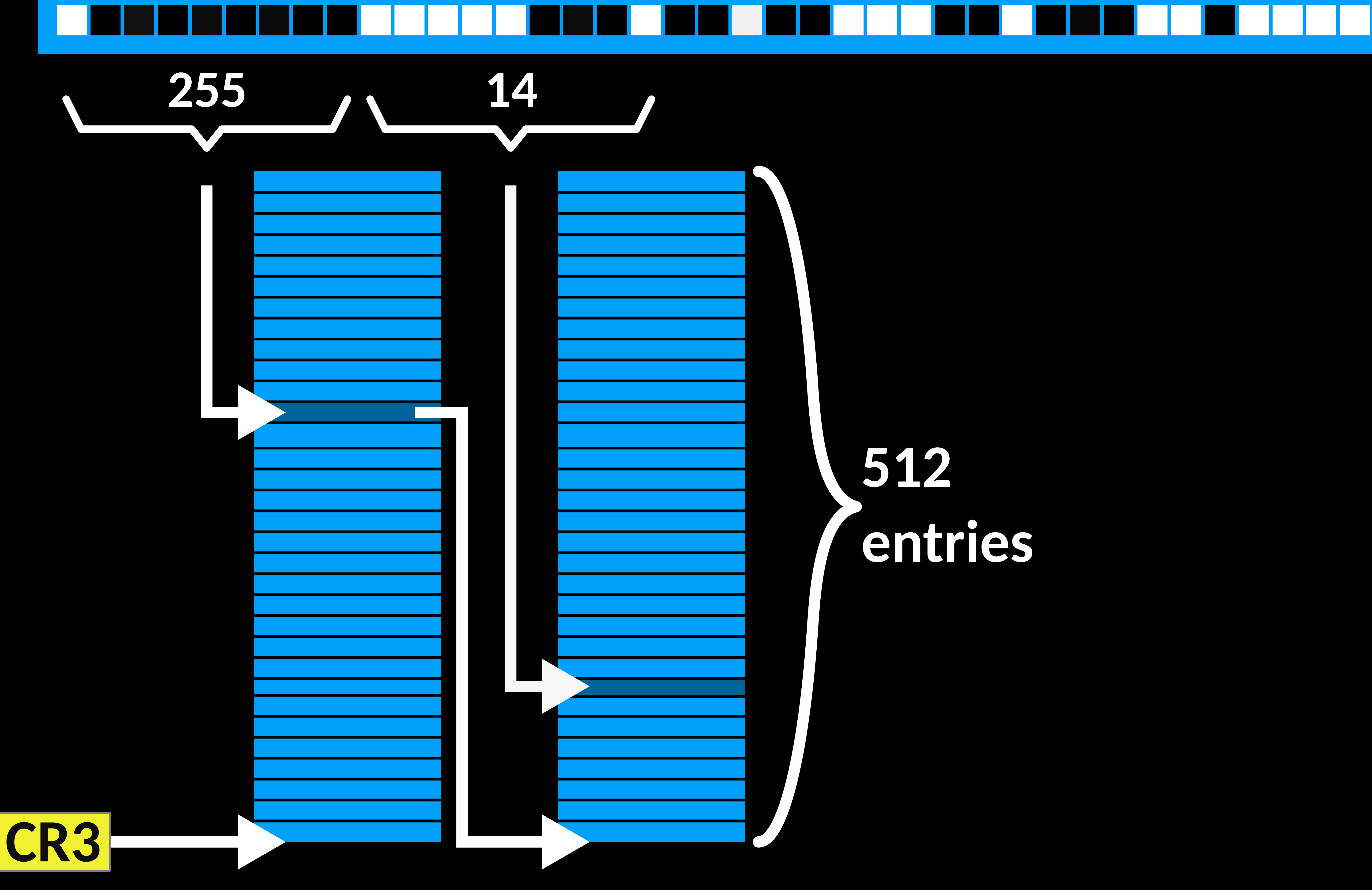
CR3

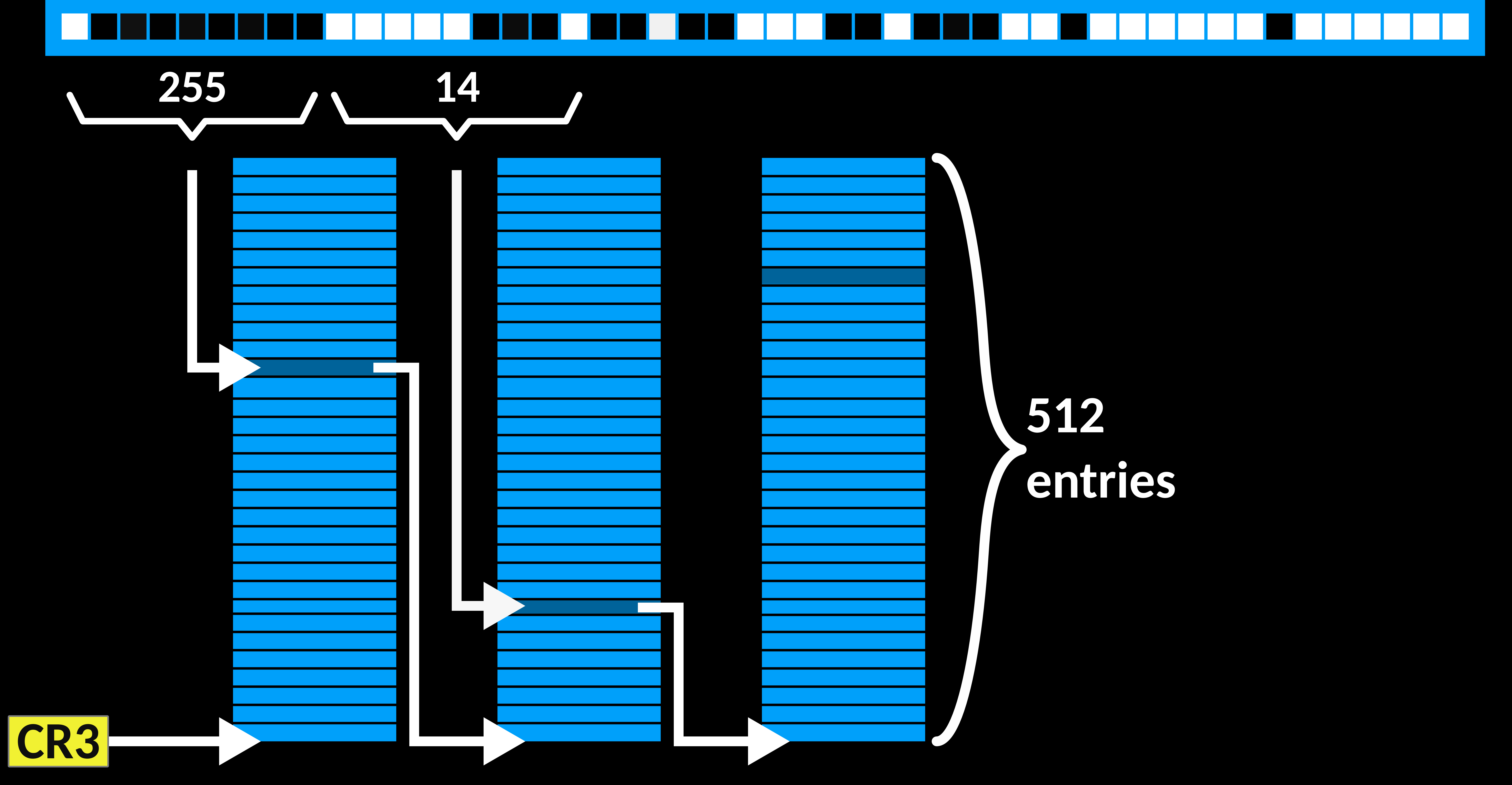


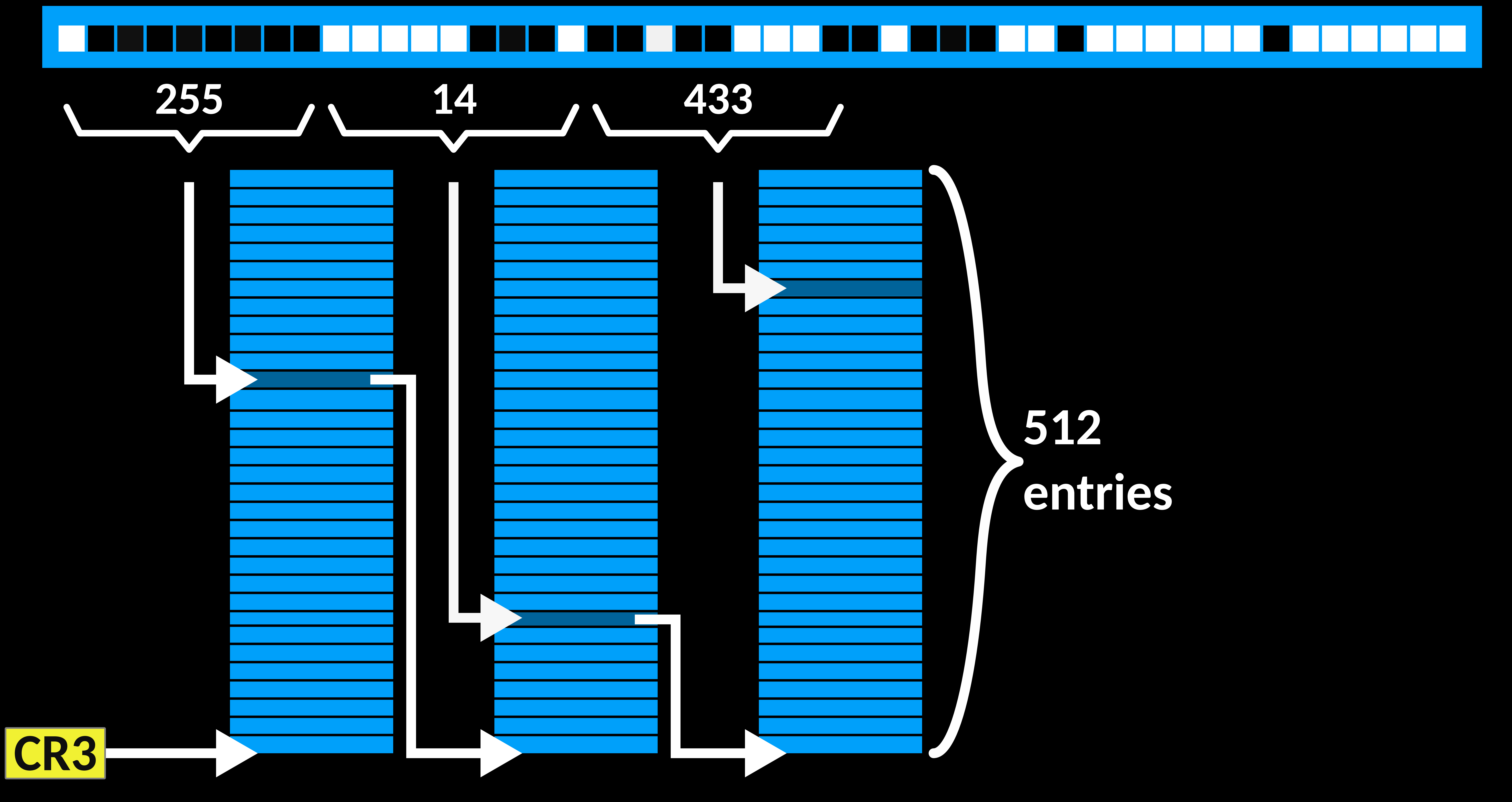

512
entries

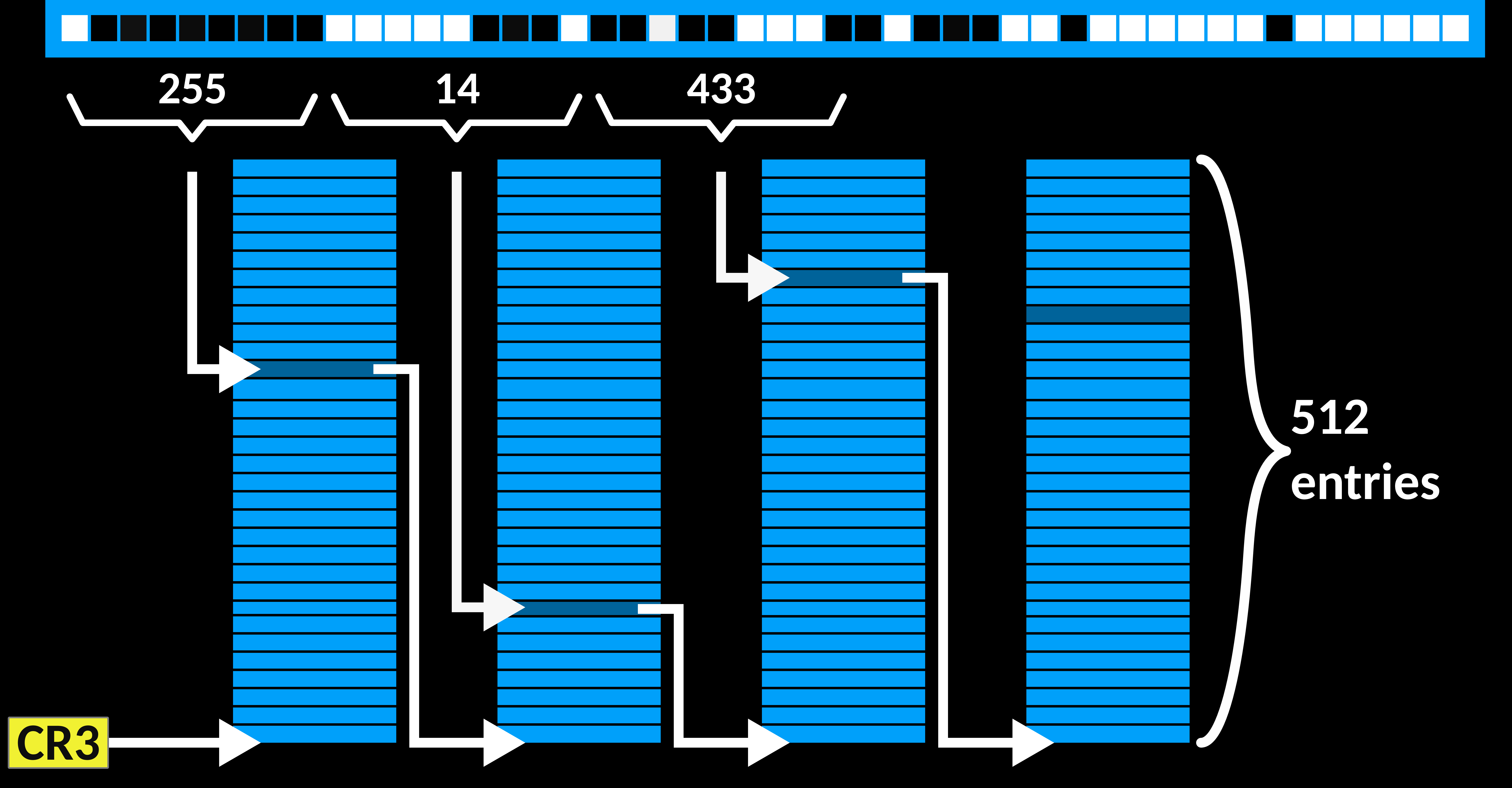

255
511
512
entries
0
CR3

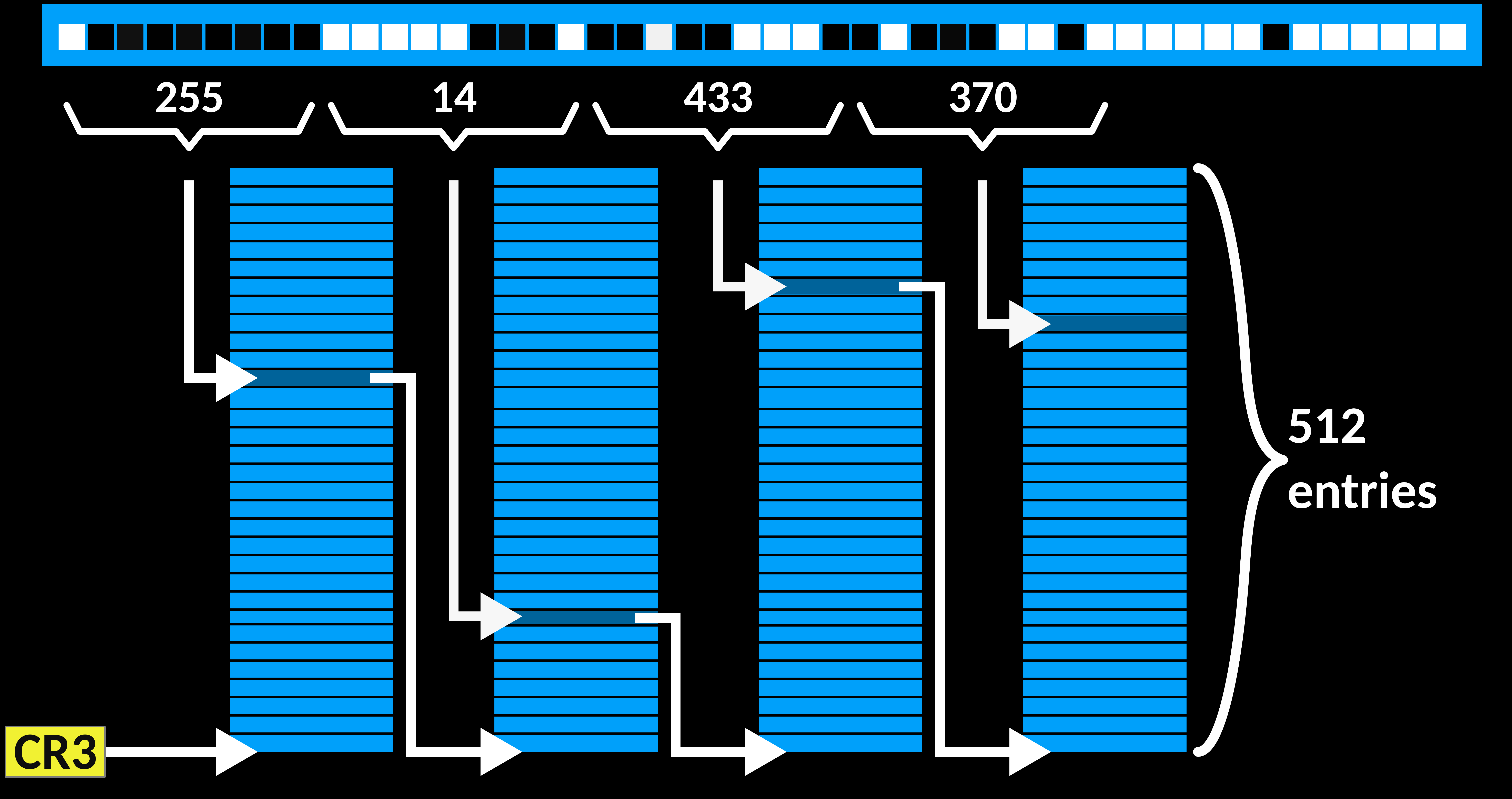

255
CR3
512
entries


255
14
512
entries
CR3

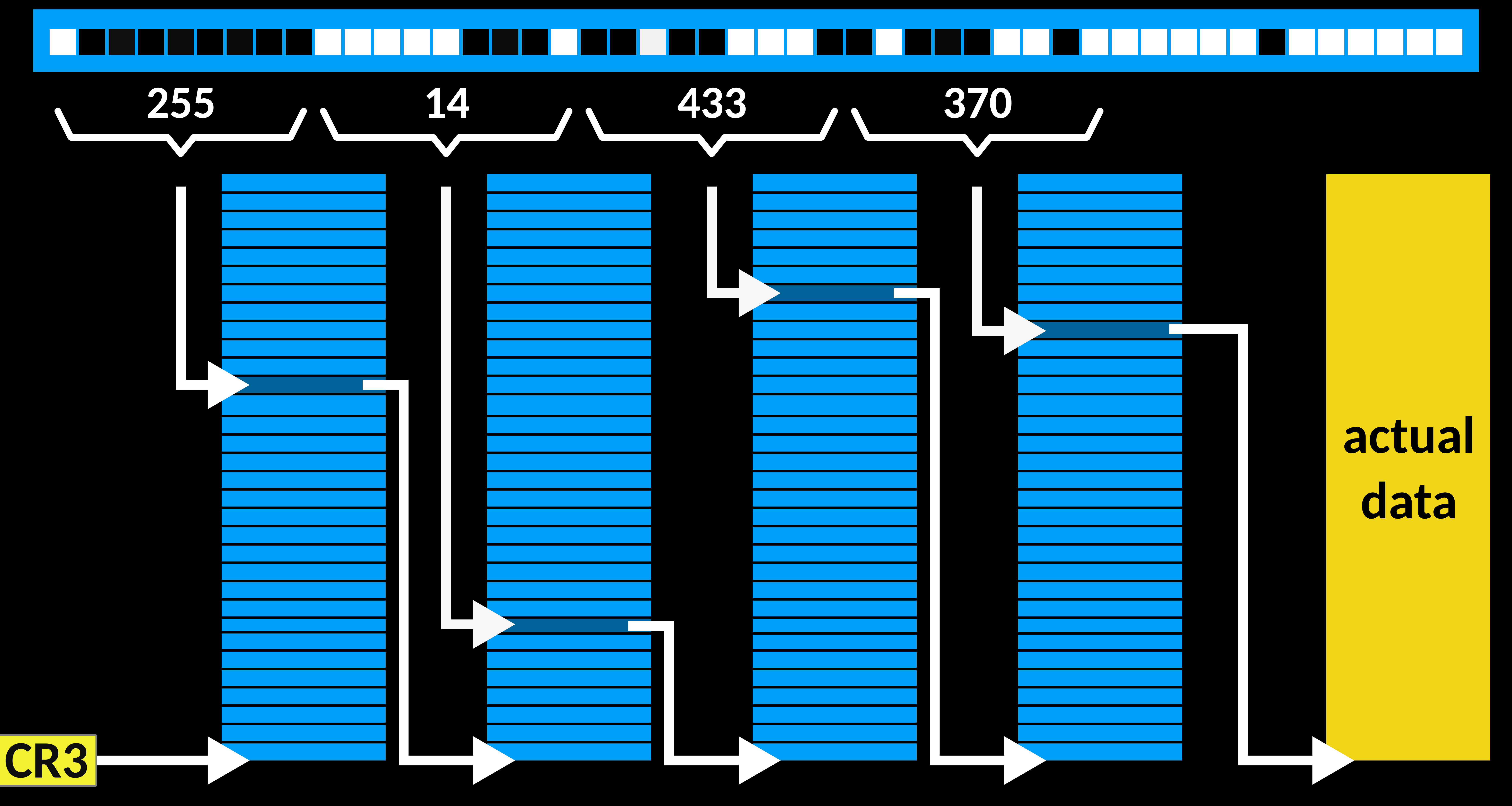

255
14
512
entries
CR3

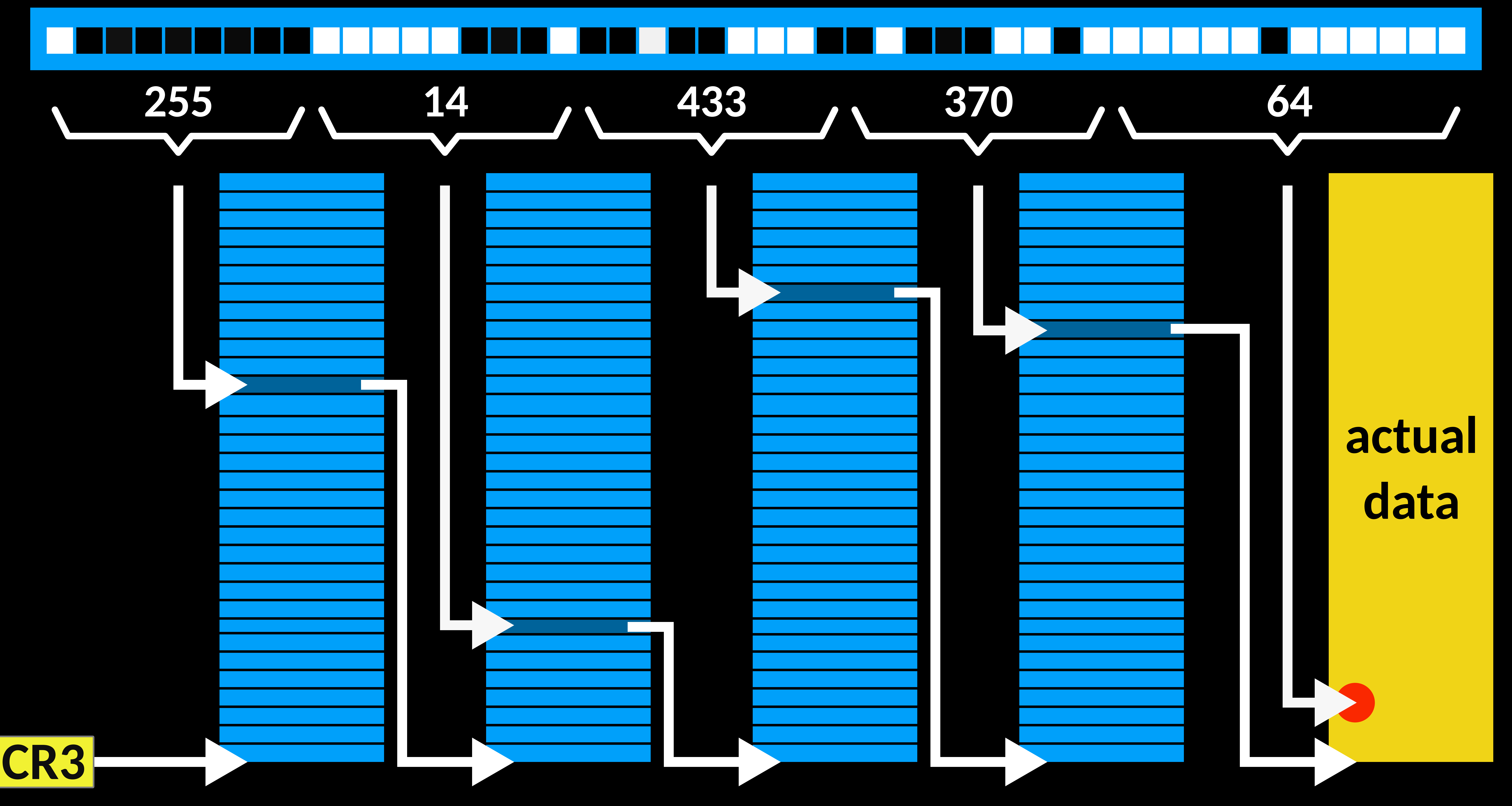

255
14
433
512
entries
CR3

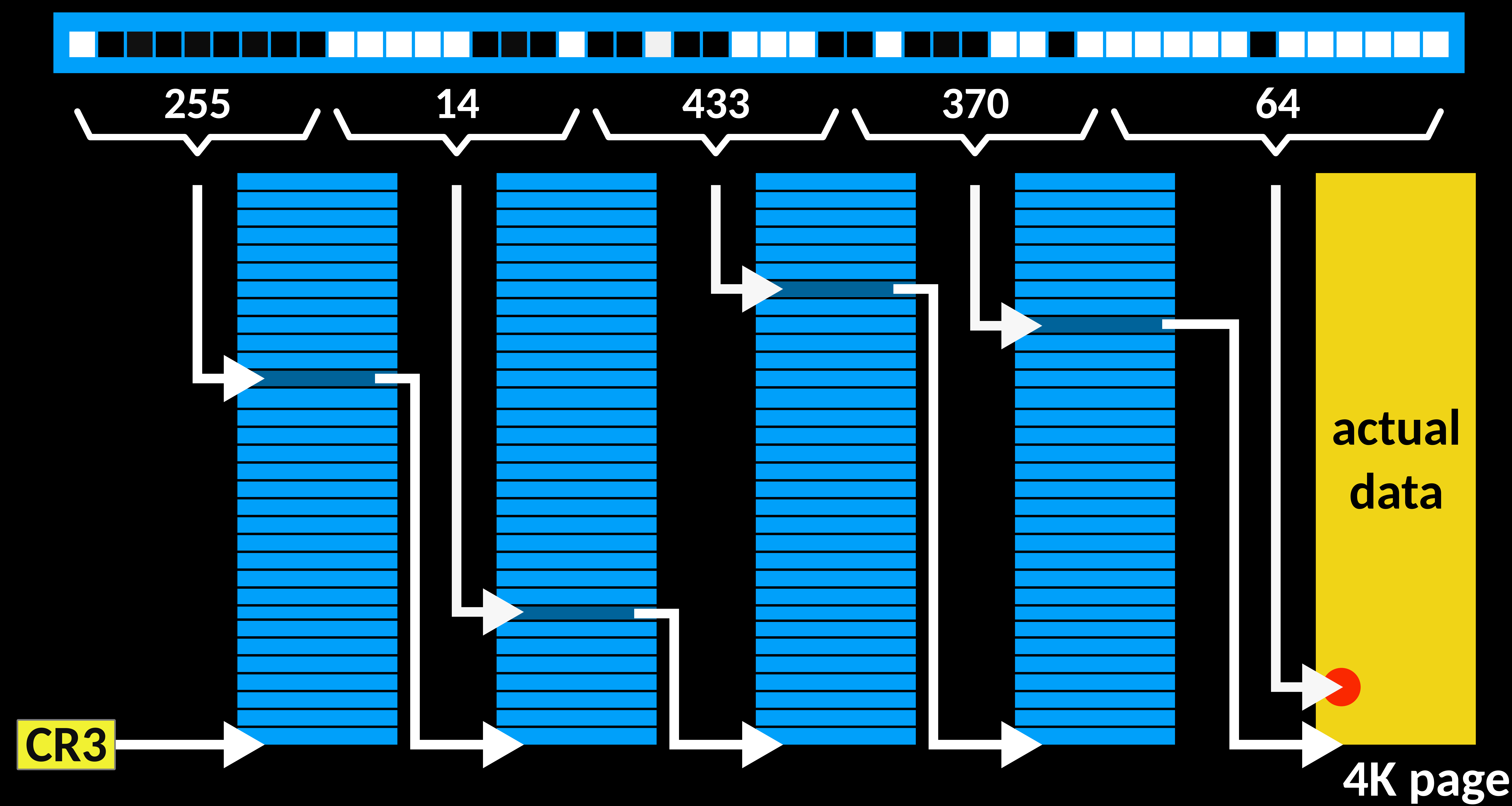

255
14
433
512
entries
CR3

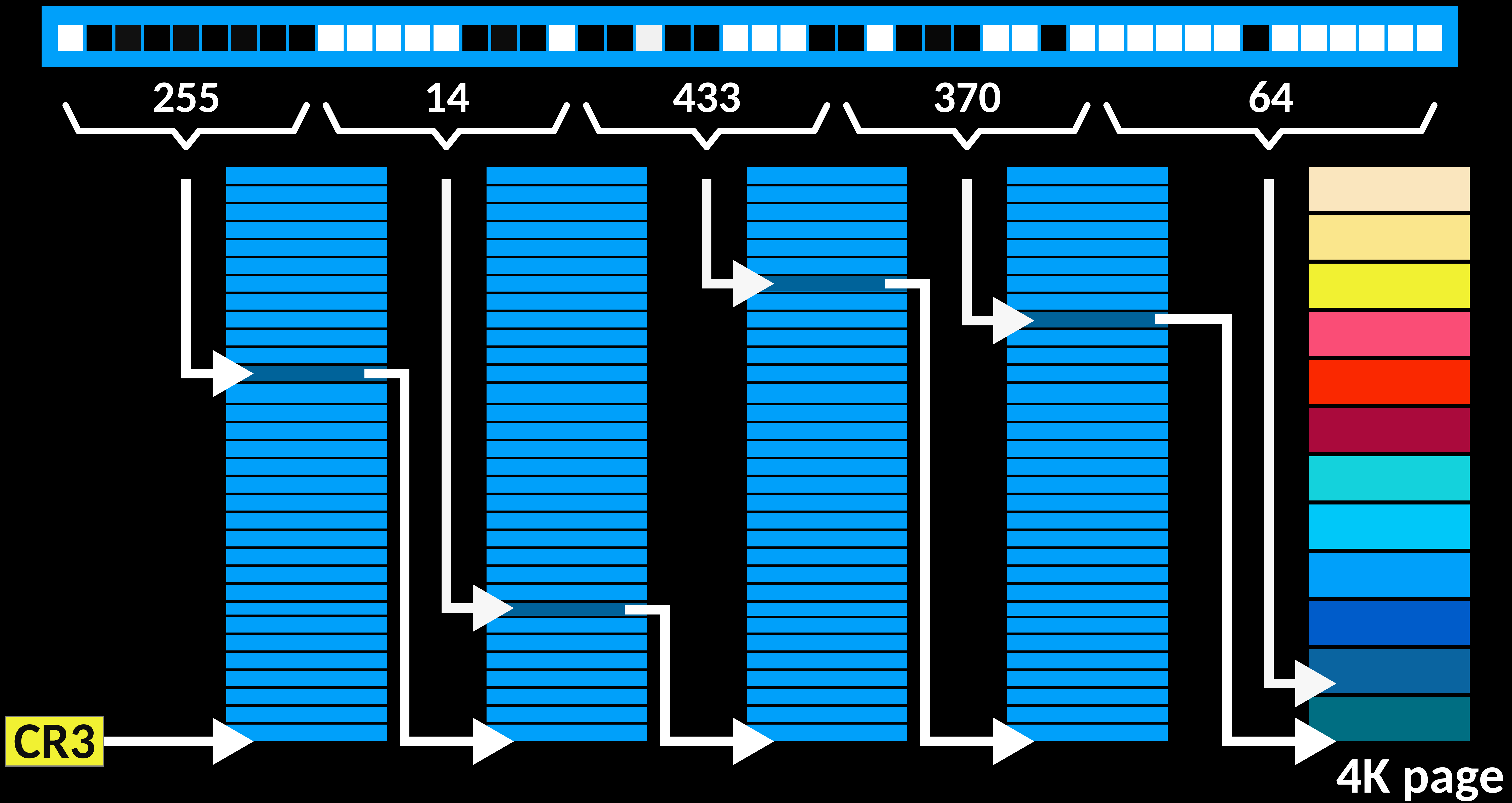


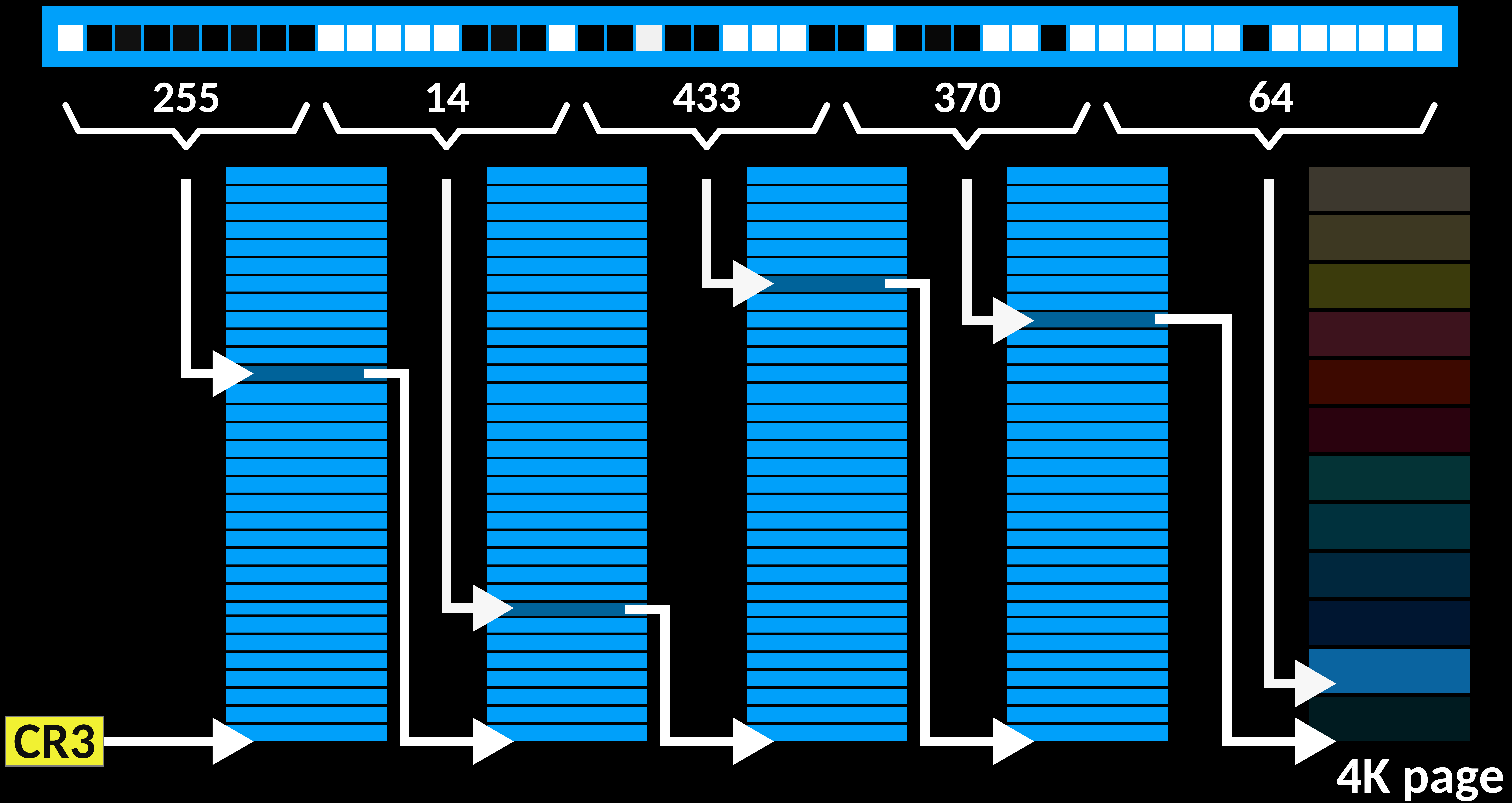
255
14
433
370
512
entries
CR3


255
14
433
370
actual data
CR3


255
14
433
370
64
actual
data
CR3

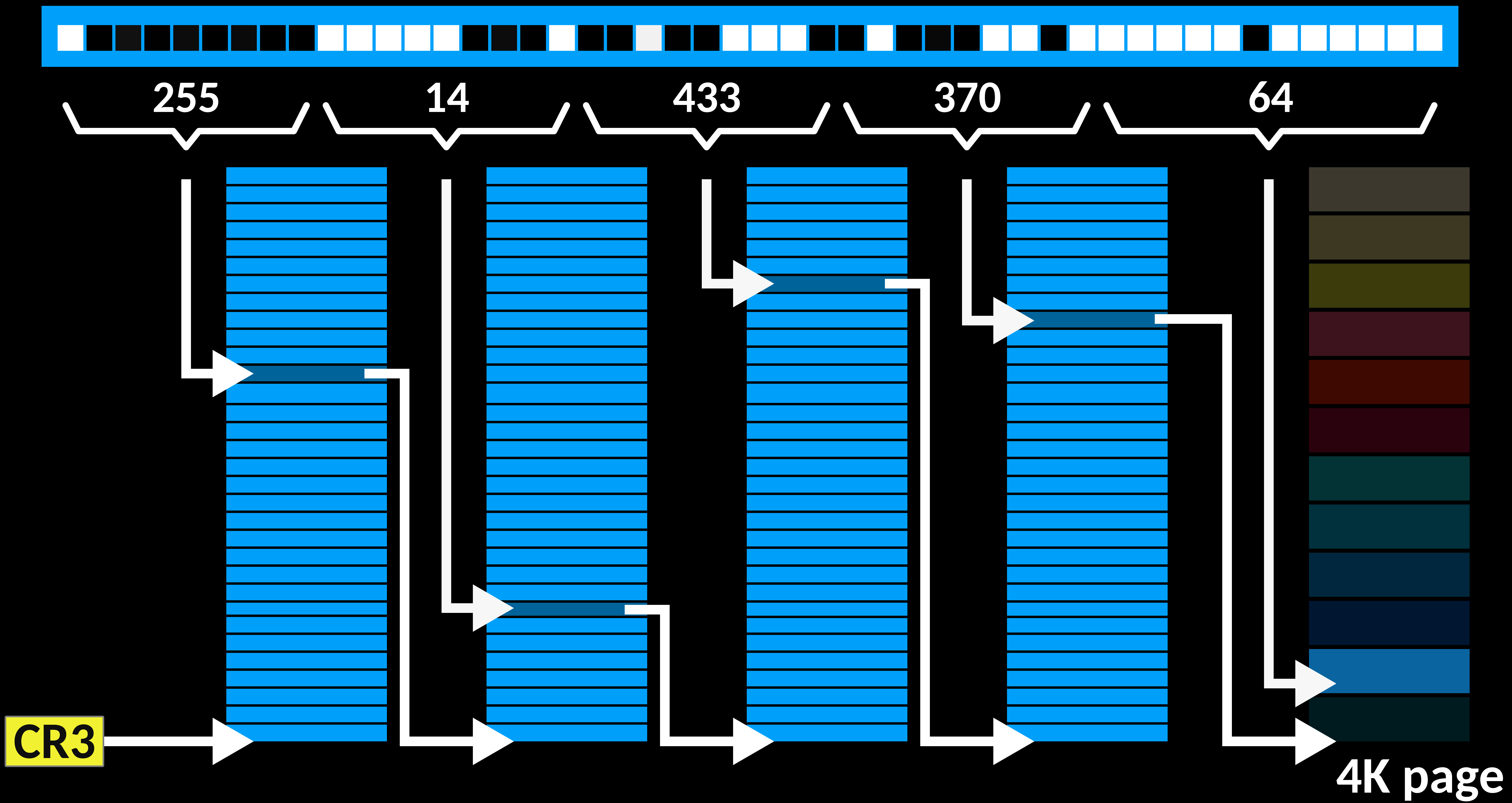

255
14
433
370
64
actual data
CR3
4K page

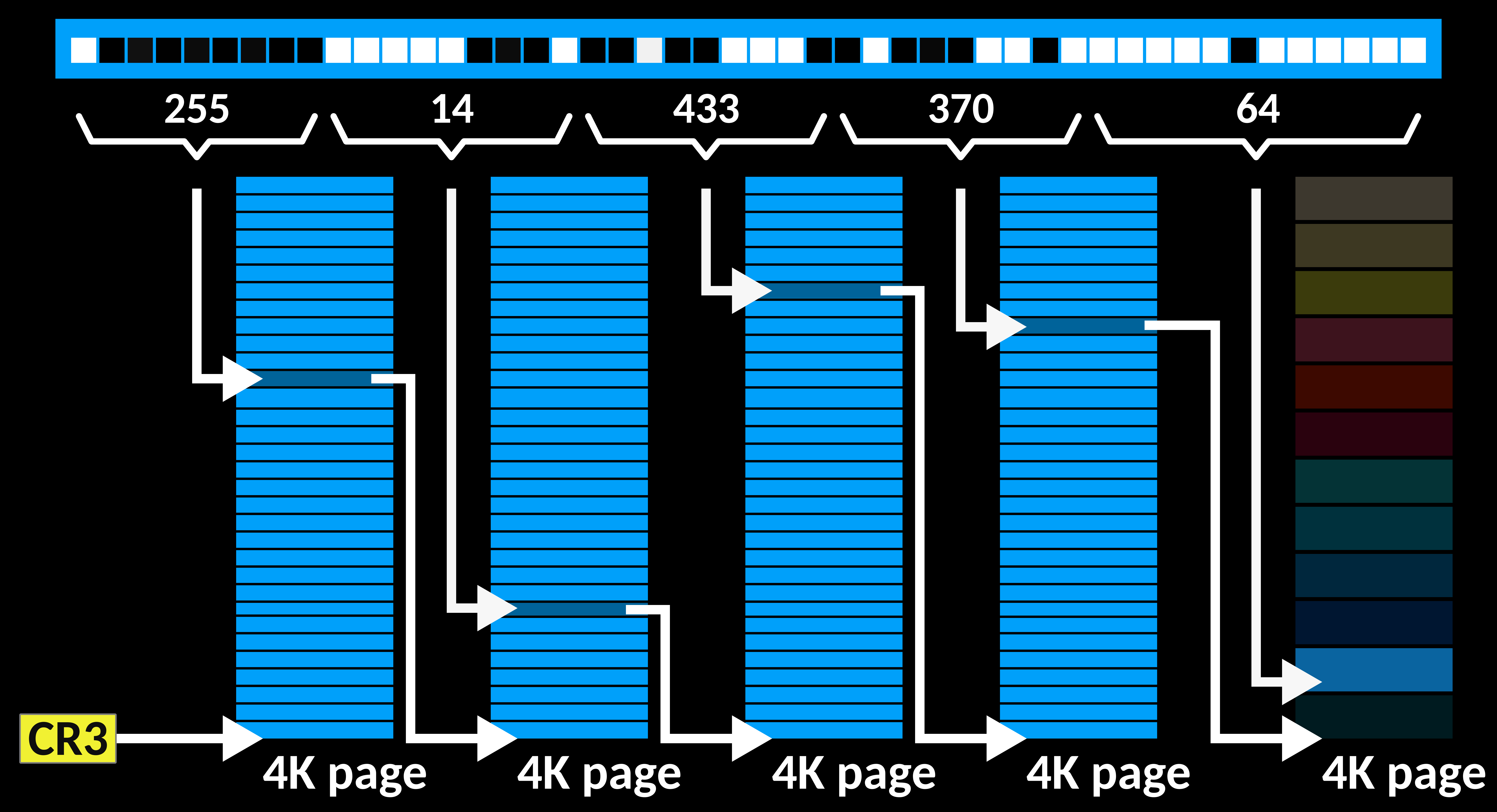

255
14
433
370
64
CR3
4K page

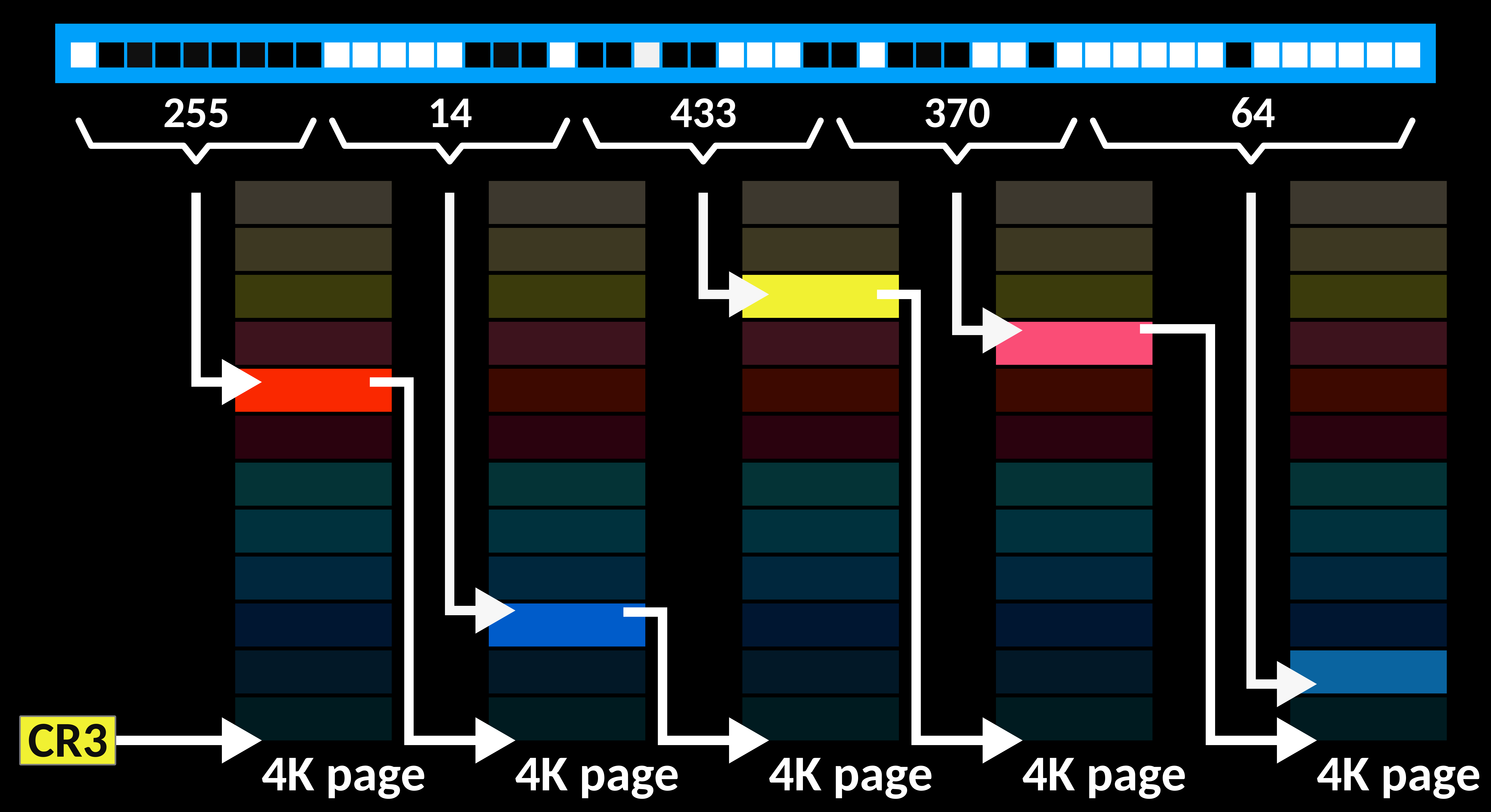

255
14
433
370
64
CR3
4K page

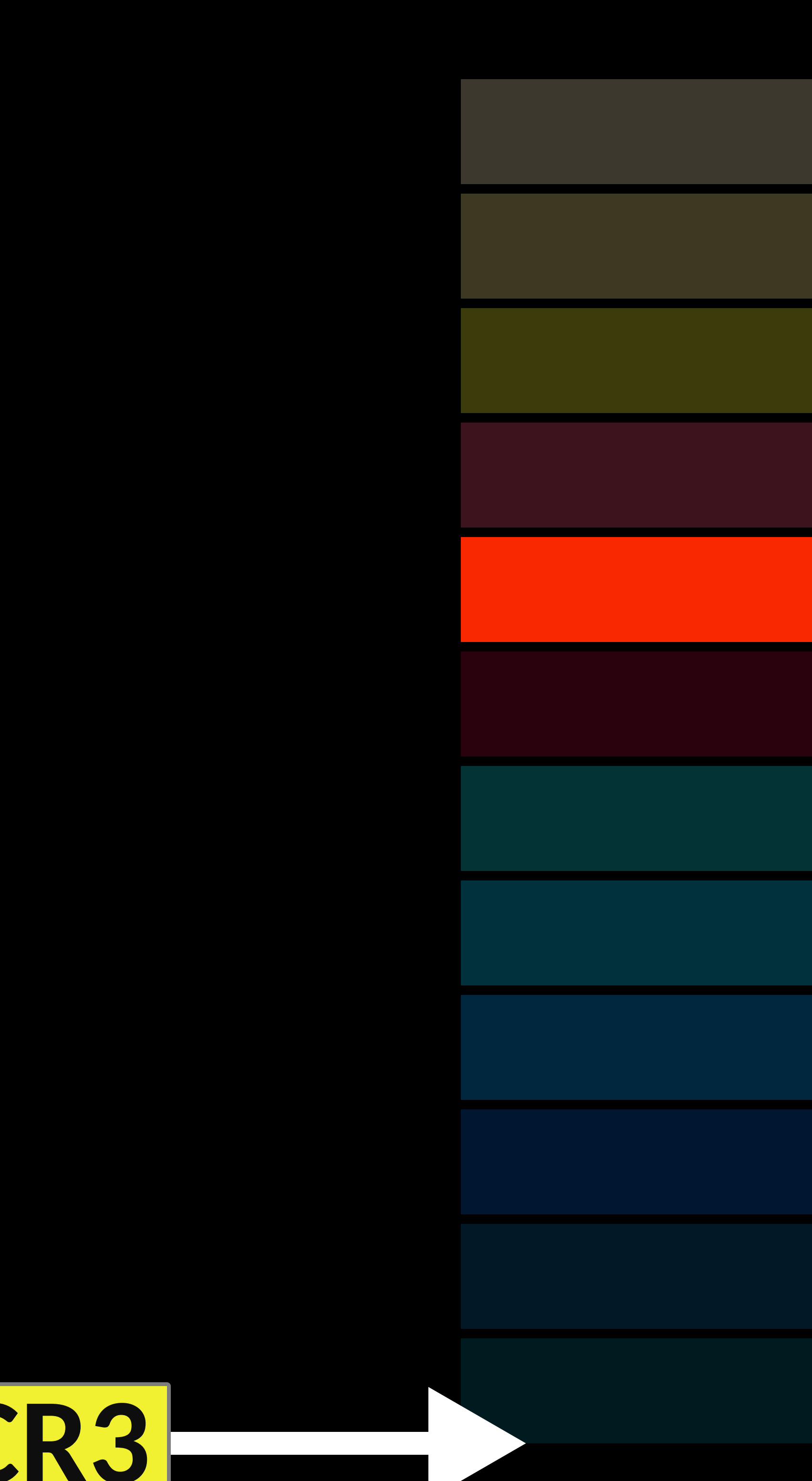
# Observation:

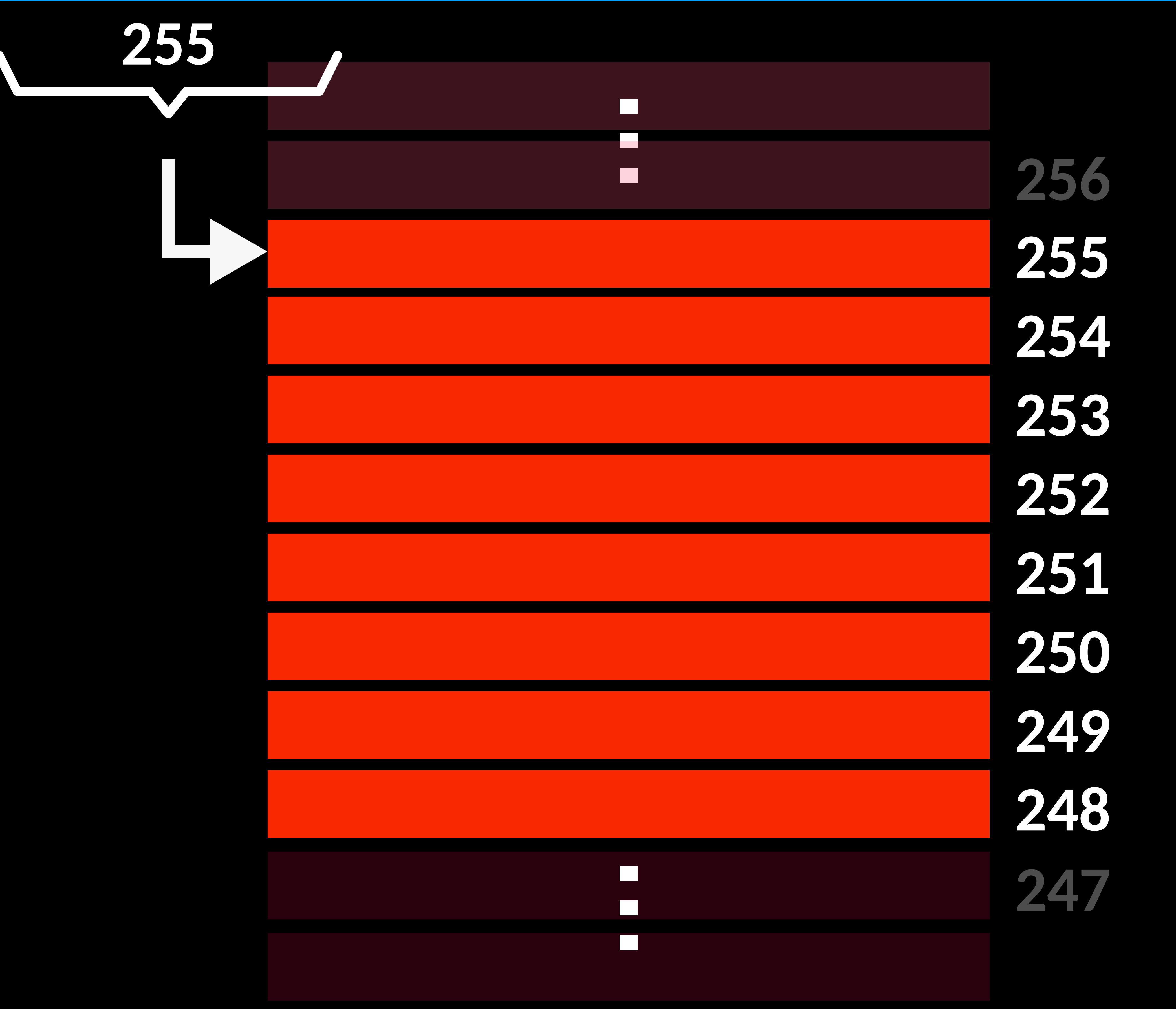
address information is directly encoded into the page table lookups, and page tables are pages themselves.


255
14
433
370
64
CR3
4K page

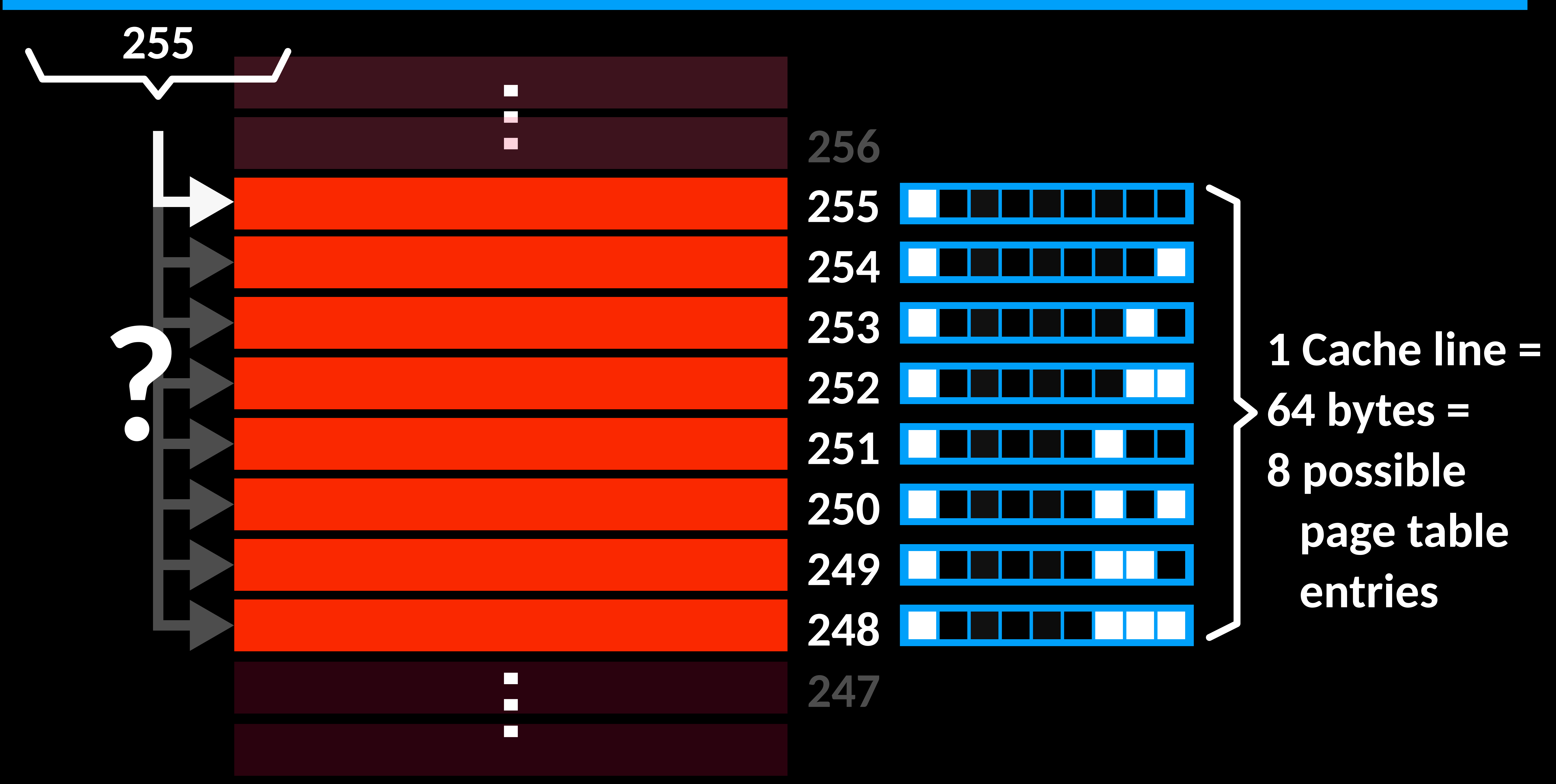

255
14
433
370
64
CR3
4K page
4K page
4K page
4K page
4K page

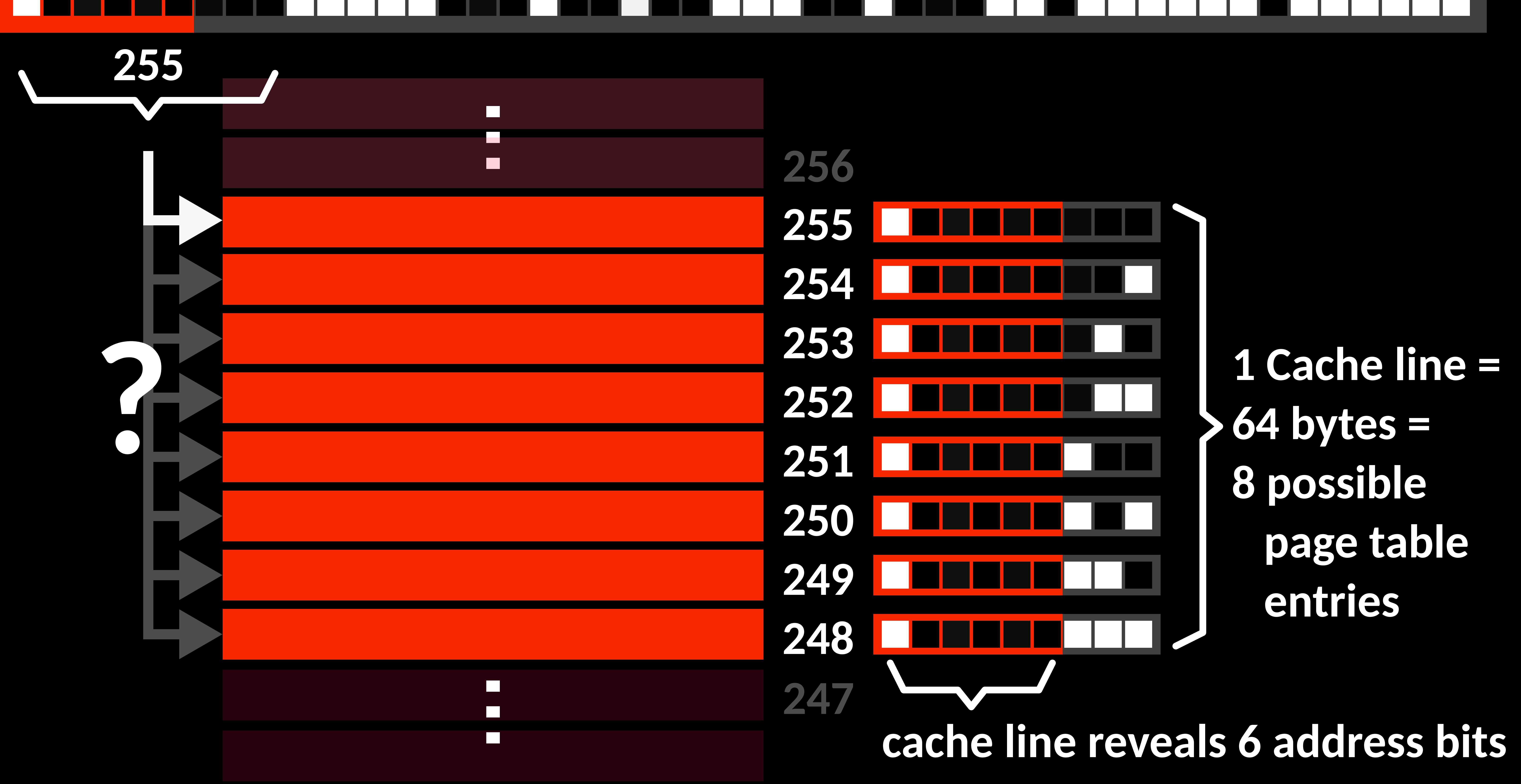

255
14
433
370
64
CR3
4K page
4K page
4K page
4K page
4K page


CR3


255
256
255
254
253
252
251
250
249
248
247

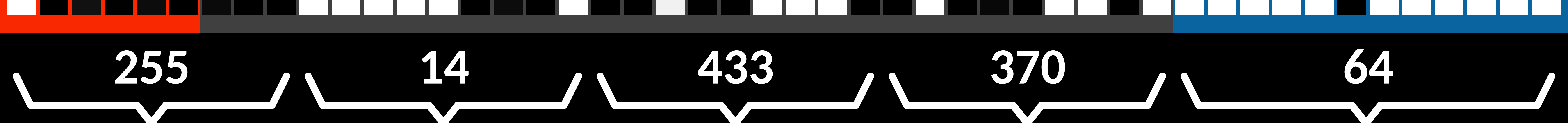
255
256
255
254
253
252
251
250
249
248
247
?
1 Cache line =
64 bytes =
8 possible
page table
entries


255
?
256
255
254
253
252
251
250
249
248
247
1 Cache line =
64 bytes =
8 possible
page table
entries


255
256
255
254
253
252
251
250
249
248
247
?
1 Cache line =
64 bytes =
8 possible
page table
entries
cache line reveals 6 address bits

255 14 433 370 64

255 14 433 370 64




location within
the page known
by studying
browser
memory allocator

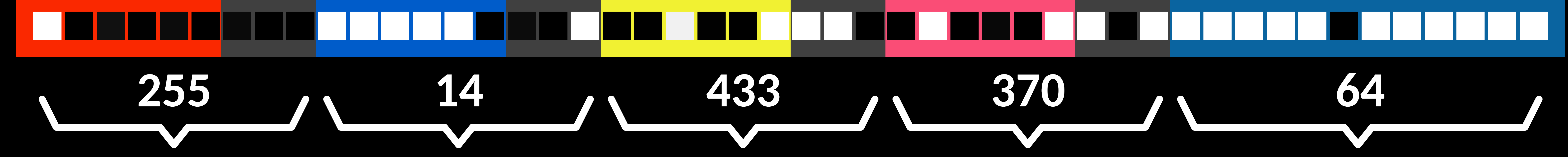
255

14

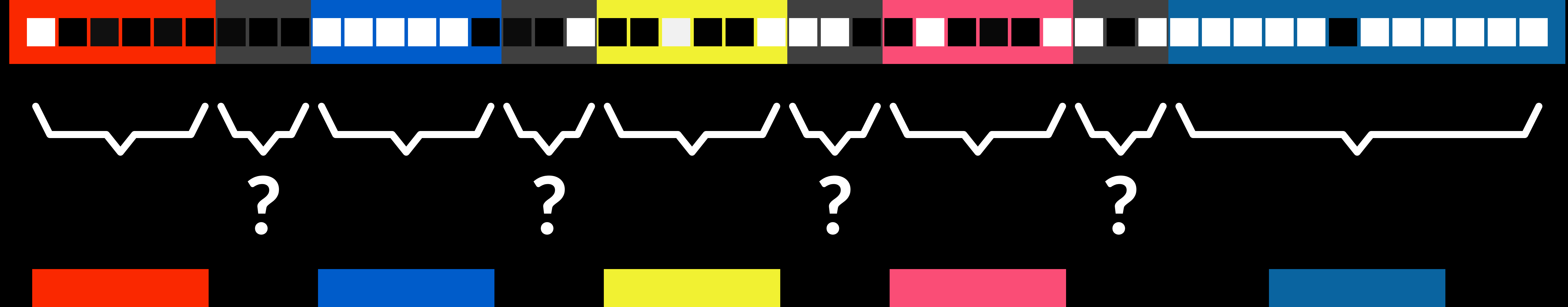
433

370

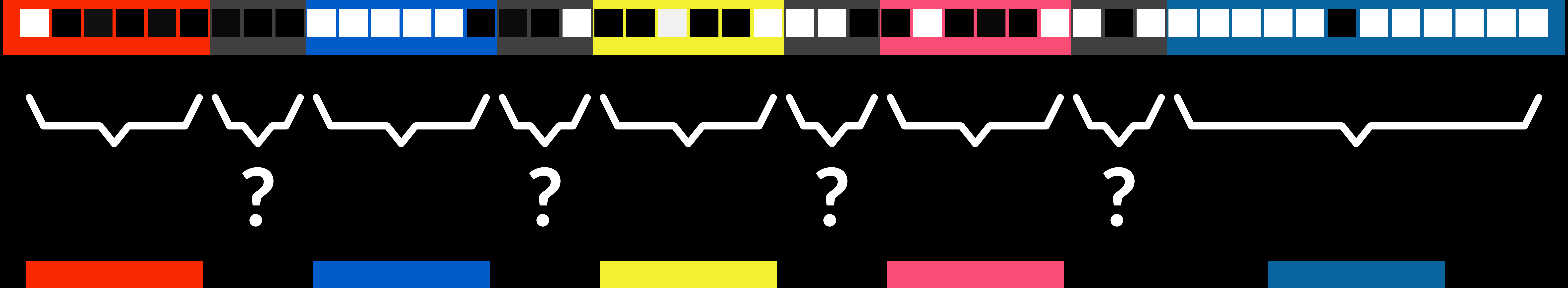
64




max entropy left:



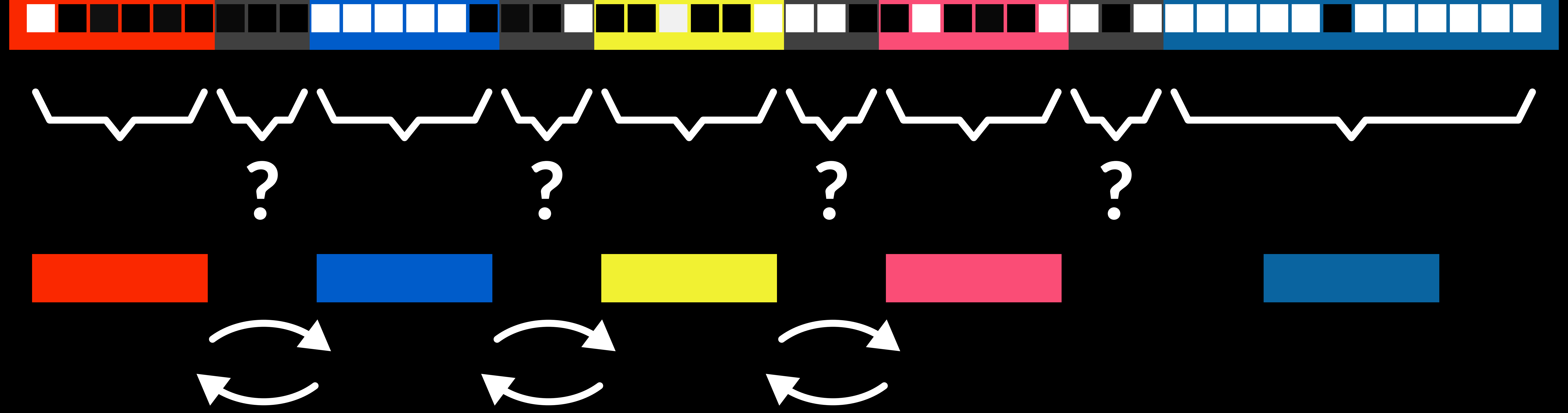

**max entropy left:**




max entropy left: 4*3 bits + ...

which hit belongs to which cache line?

max entropy left: 4*3 bits + ...




**which hit belongs to which cache line?**

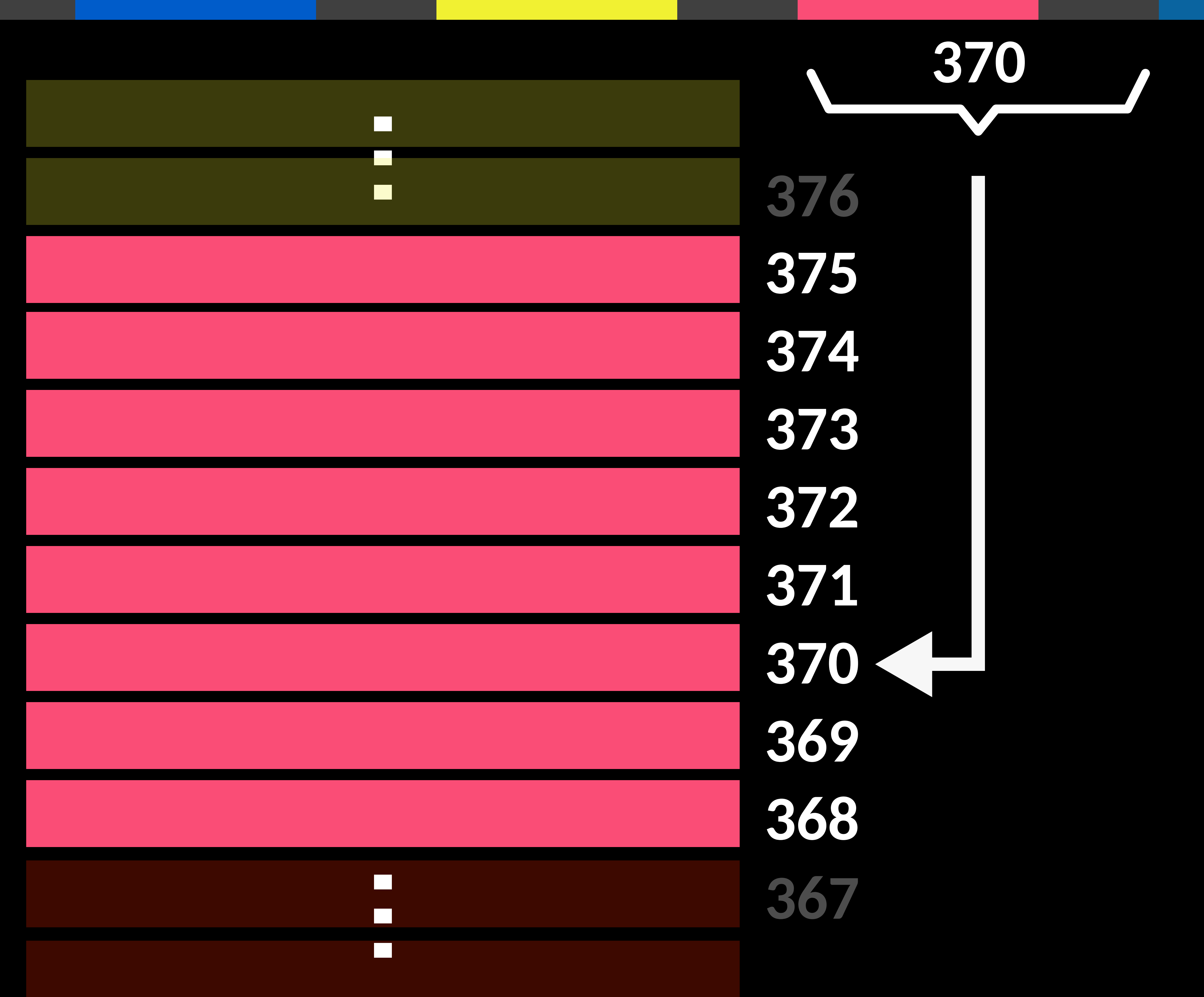
**max entropy left: $4*3$ bits + $\log^2( 4 * 3 * 2 * 1 )$**

which hit belongs to which cache line?

max entropy left: ~ 16.6 bits

# Sliding

- allocate a buffer
- perform this side-channel attack on buffer entries 4096 bytes apart
- measure when the page table lookup crosses a cache line boundary


370
376
375
374
373
372
371
370
369
368
367

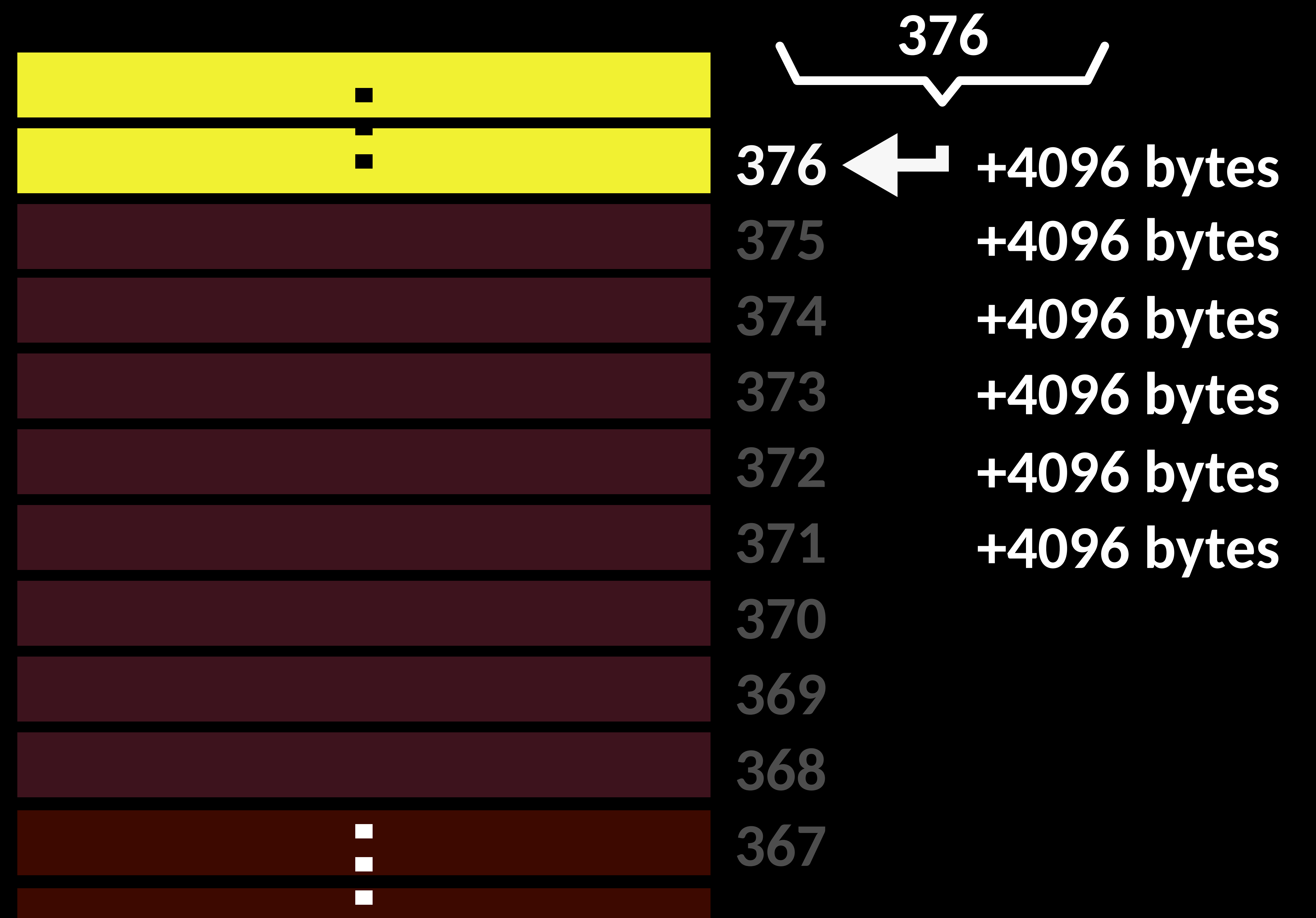
371
376
375
374
373
372
371
+4096 bytes
370
369
368
367

372
376
375
374
373
372
371
370
369
368
367
+4096 bytes
+4096 bytes

373
376
375
374
373
372
371
370
369
368
367
+4096 bytes
+4096 bytes
+4096 bytes

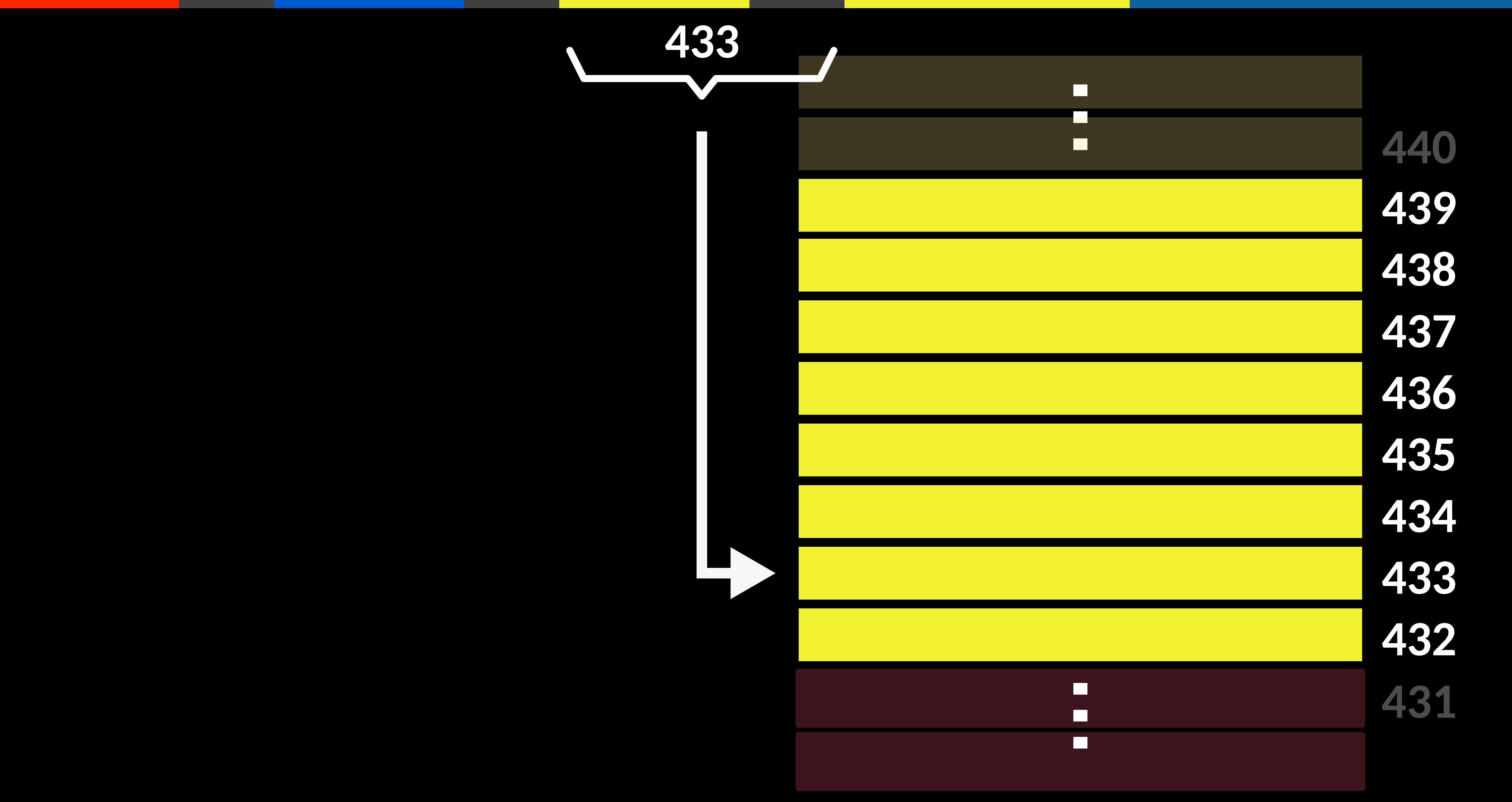
374
376
375
374
373
372
371
370
369
368
367
+4096 bytes
+4096 bytes
+4096 bytes
+4096 bytes

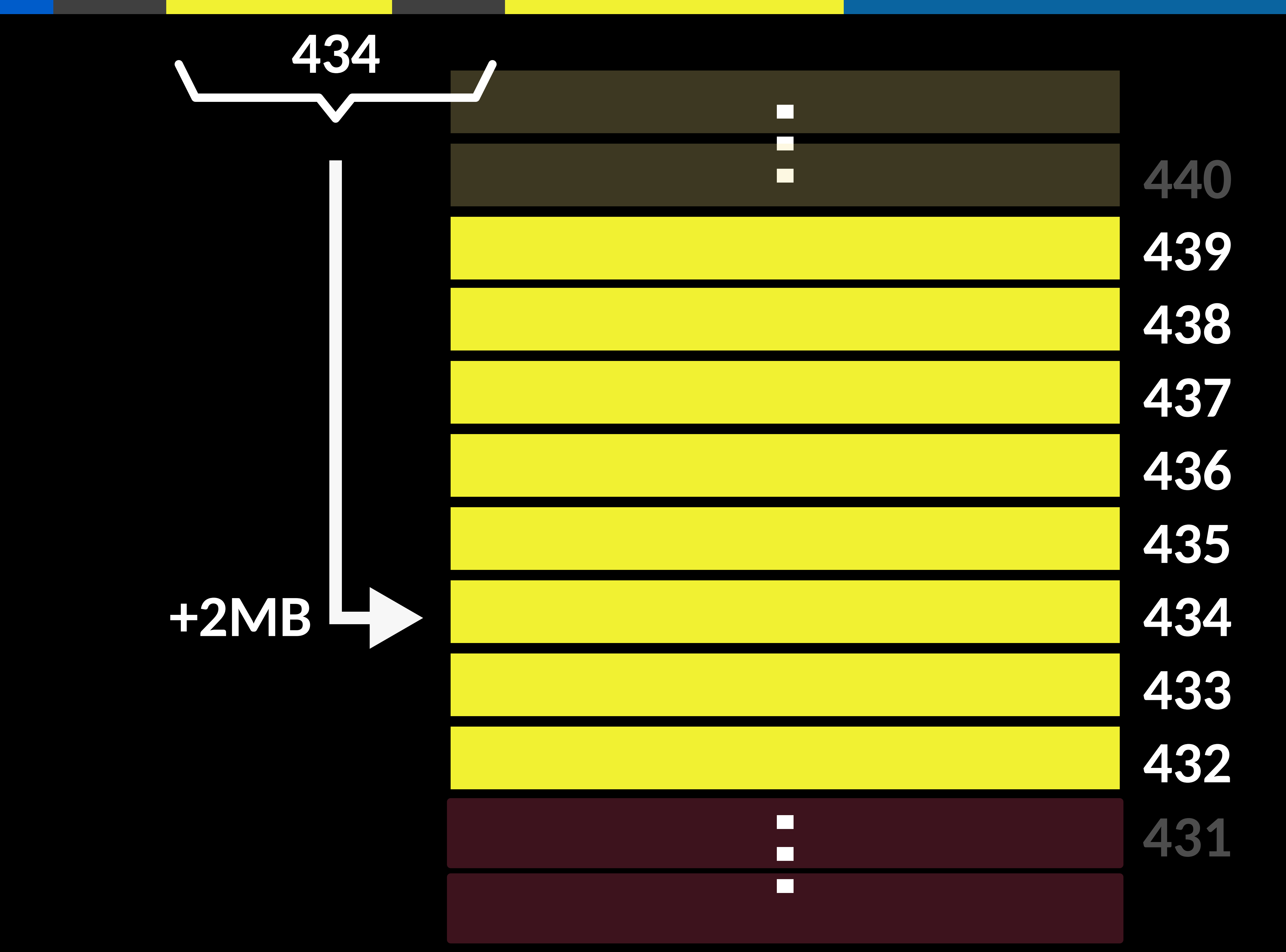
375
376
375
374
373
372
371
370
369
368
367
+4096 bytes
+4096 bytes
+4096 bytes
+4096 bytes
+4096 bytes

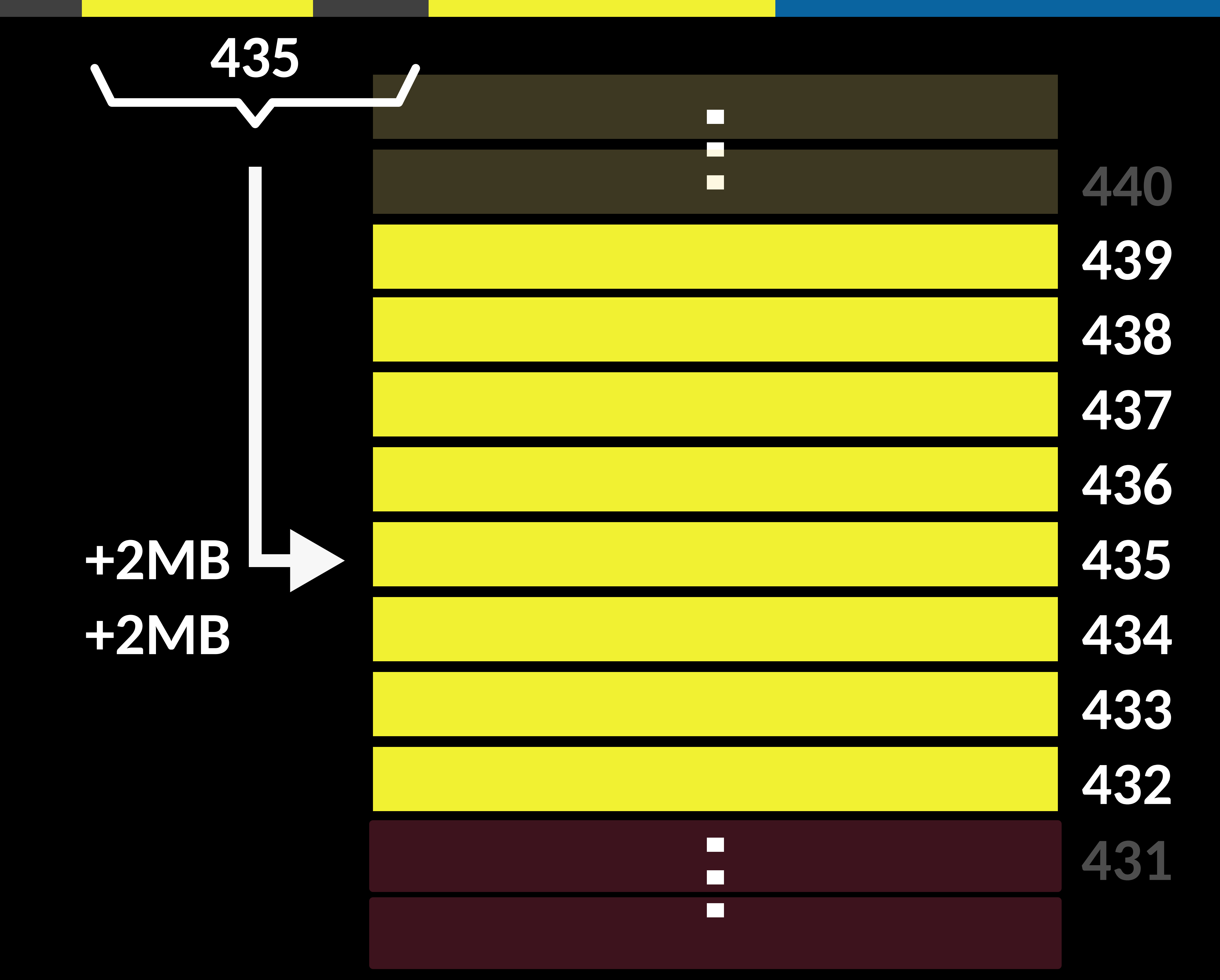

376
376
+4096 bytes
375
+4096 bytes
374
+4096 bytes
373
+4096 bytes
372
+4096 bytes
371
+4096 bytes
370
369
368
367

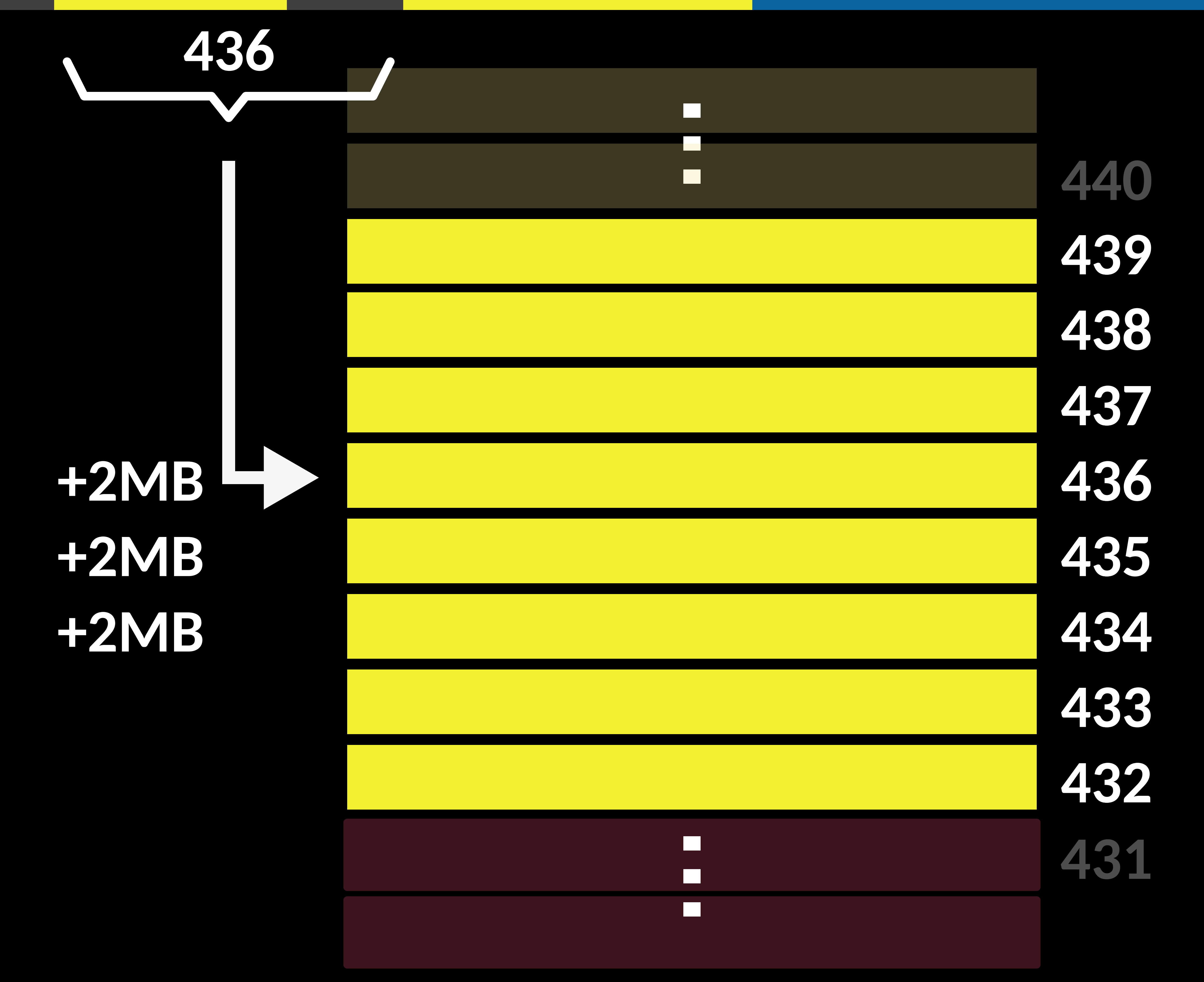
# Sliding

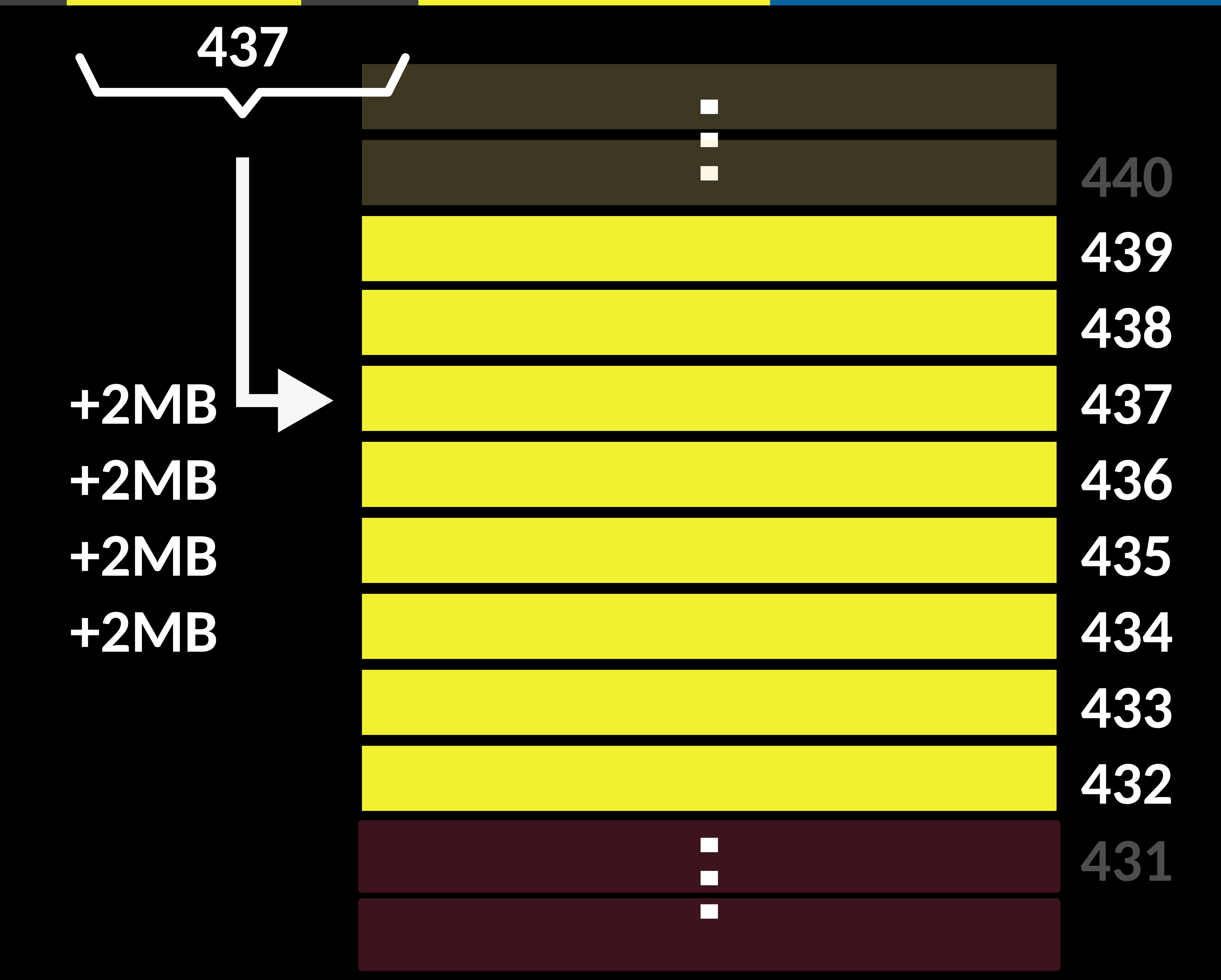
we can do the same thing for the 2nd level page table

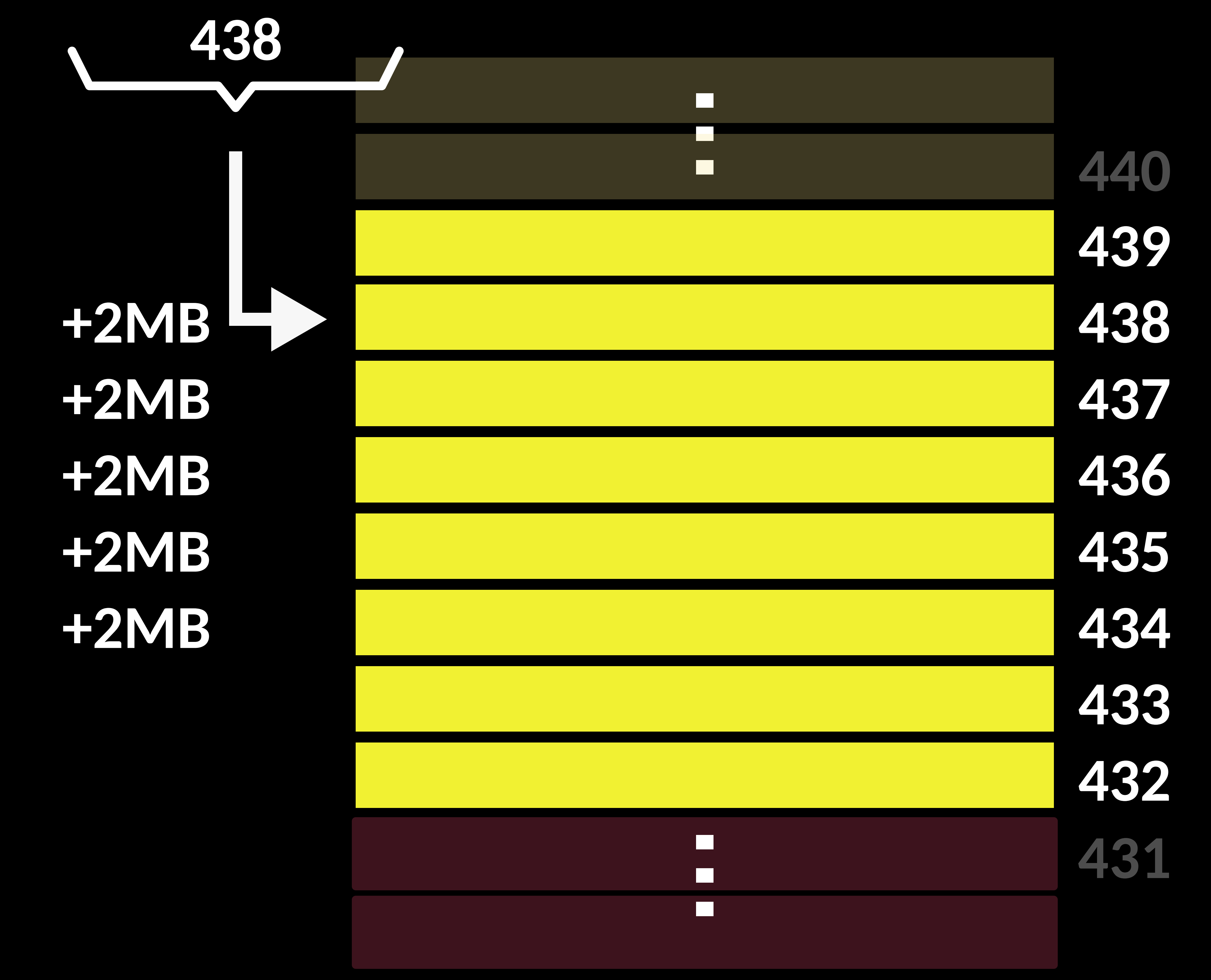

433
440
439
438
437
436
435
434
433
432
431


434
440
439
438
437
436
435
+2MB
434
433
432
431

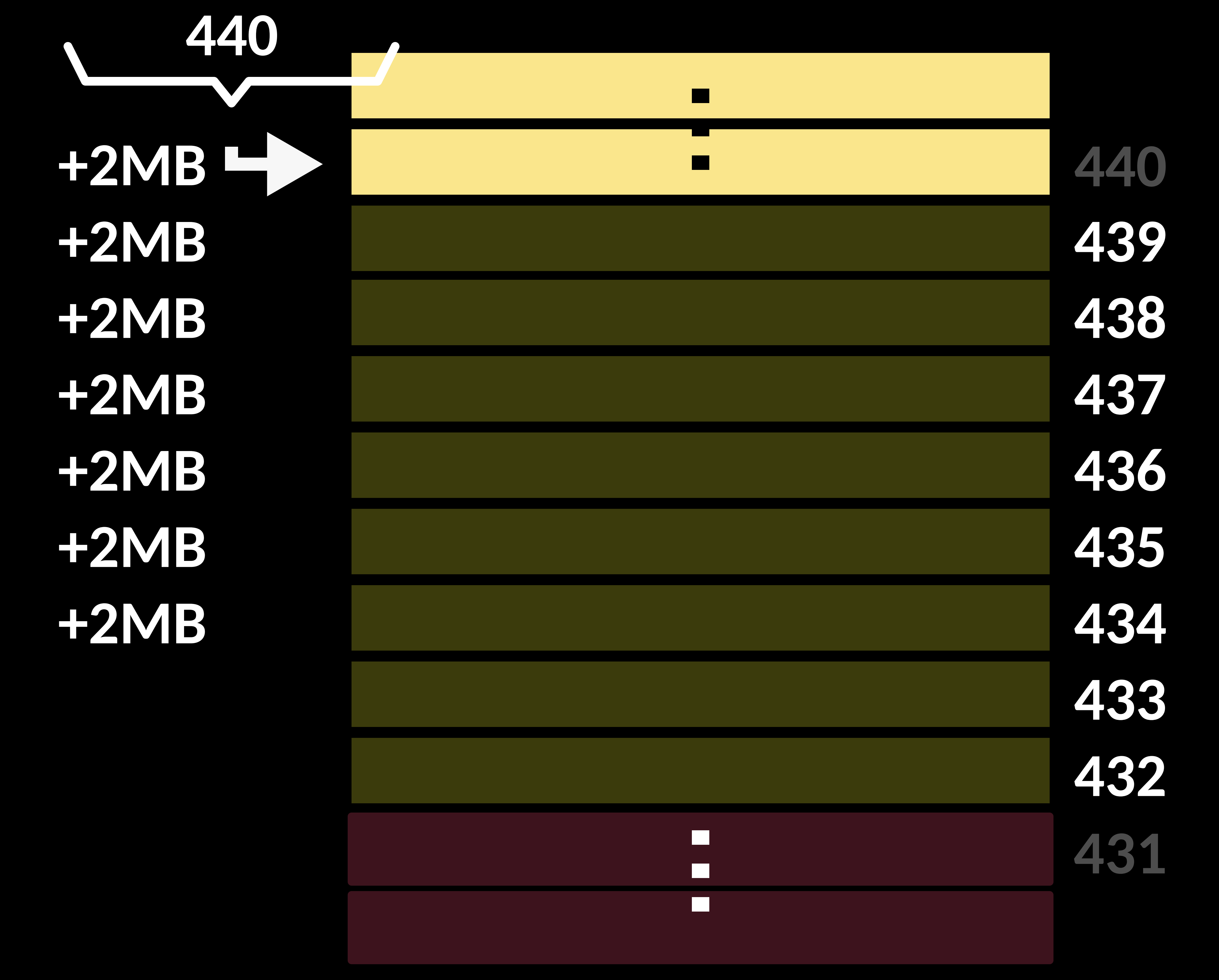

435
440
439
438
437
436
+2MB
+2MB
435
434
433
432
431

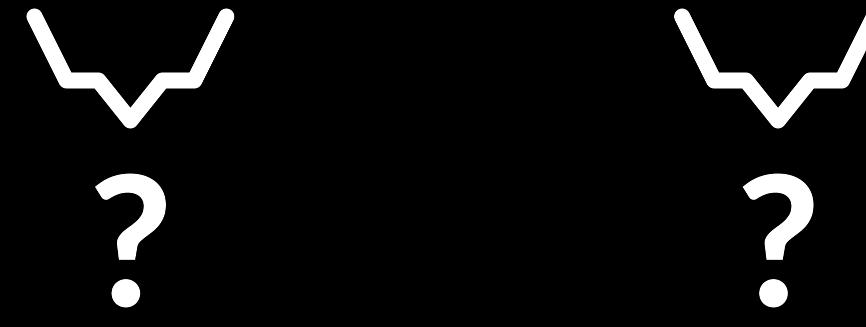

436
440
439
438
437
+2MB
436
+2MB
435
+2MB
434
433
432
431

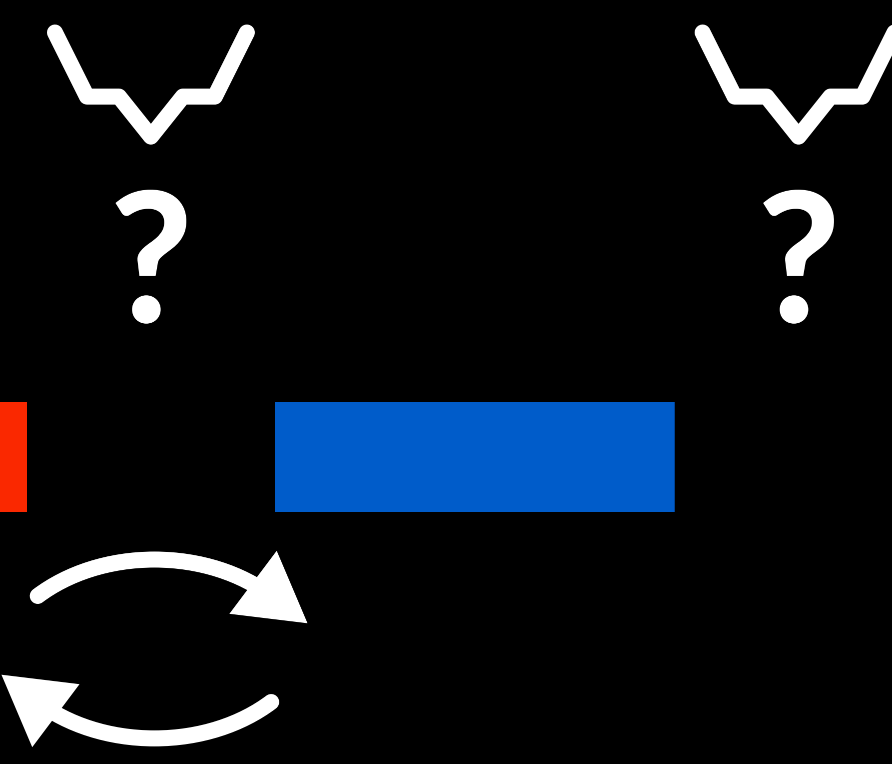

437
+2MB
+2MB
+2MB
+2MB
440
439
438
437
436
435
434
433
432
431

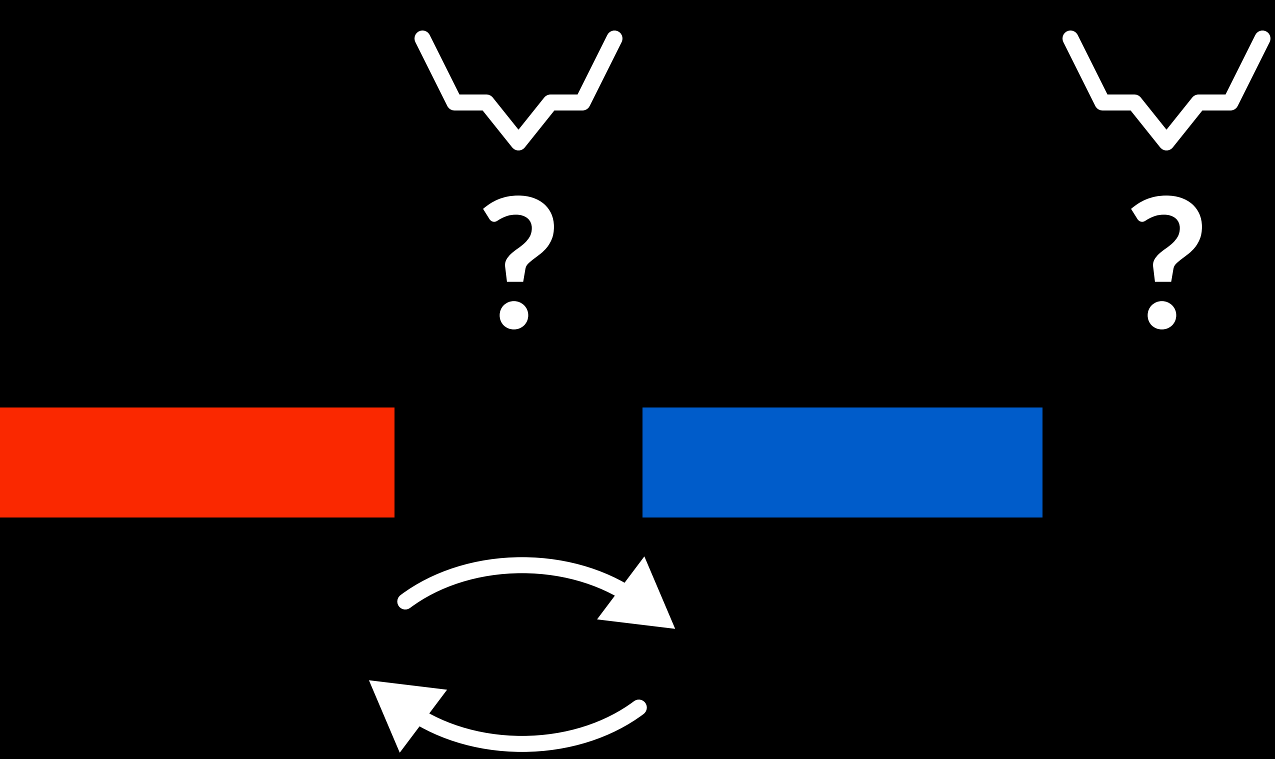

438
440
439
+2MB
438
+2MB
437
+2MB
436
+2MB
435
+2MB
434
433
432
431

439
+2MB
+2MB
+2MB
+2MB
+2MB
+2MB
440
439
438
437
436
435
434
433
432
431

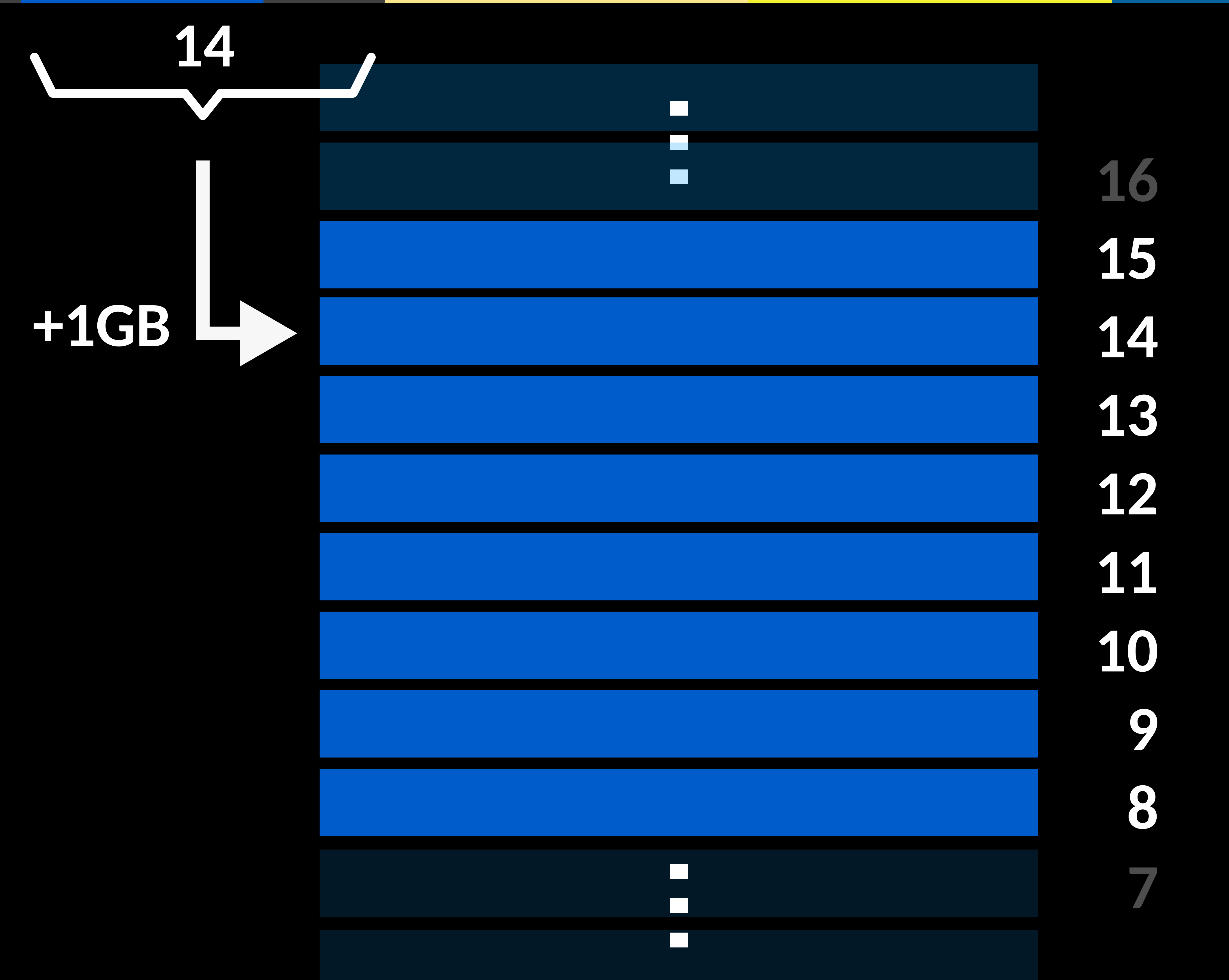

440
+2MB
+2MB
+2MB
+2MB
+2MB
+2MB
+2MB
440
439
438
437
436
435
434
433
432
431

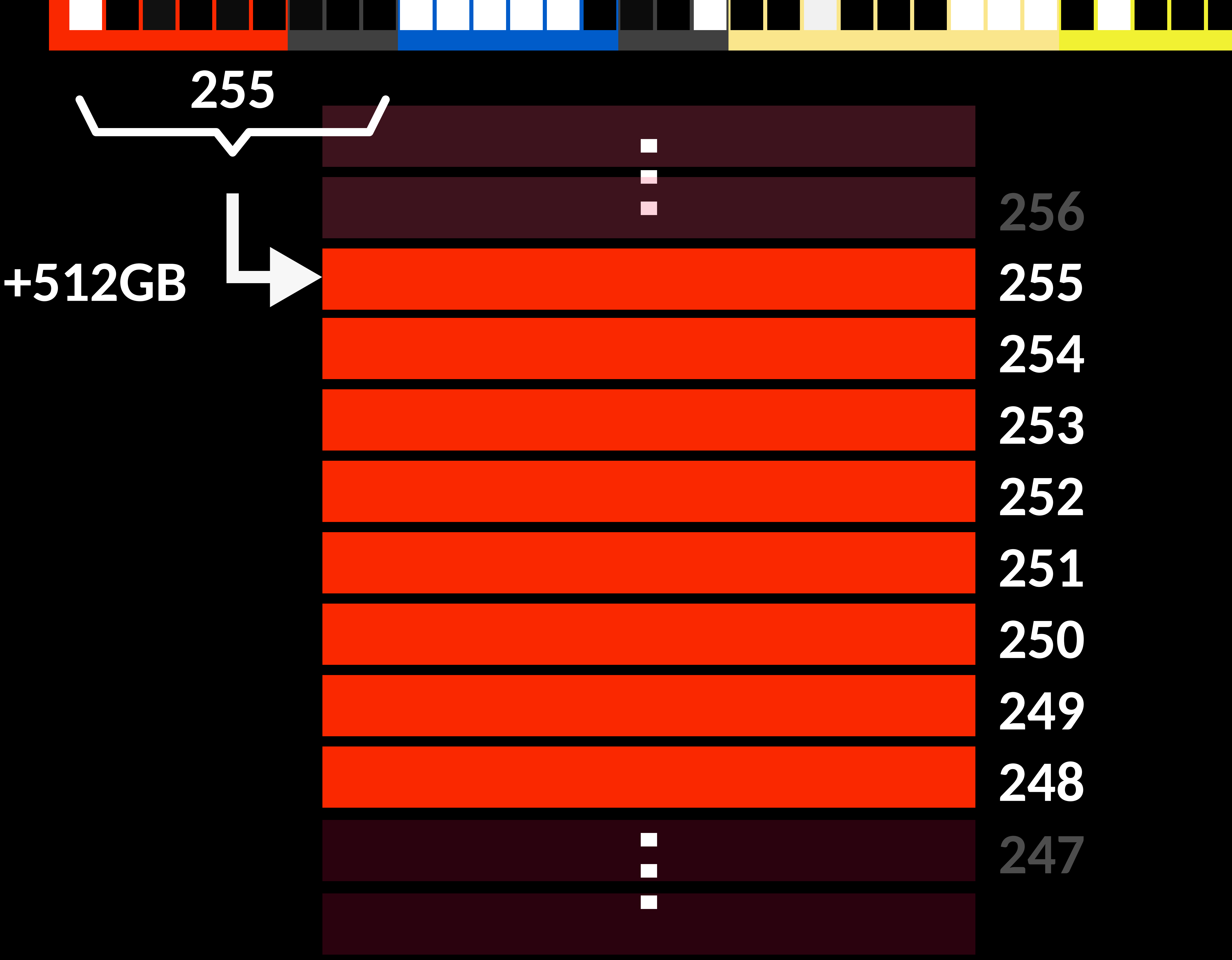







**max entropy left: 2*3 + log2(2 * 1) = 7 bits**


14
+1GB
16
15
14
13
12
11
10
9
8
7



255
+512GB
256
255
254
253
252
251
250
249
248
247

# Allocating large chunks of memory

**Firefox** (on Linux) does not initialize ArrayBuffers, so linux does not allocate space for the actual pages

We can allocate huge chunks and use sliding to recover the whole address

# Allocating large chunks of memory

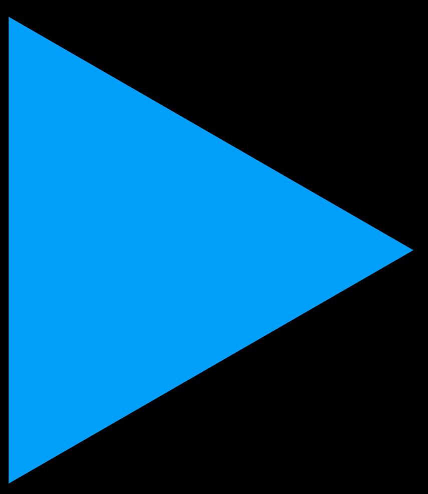
**Chrome** does initialize memory, but jumps ahead in the address space every time it creates a new heap

3rd level address bits can be recovered, 4'th level bits needs chrome to initialize/free up to 4TB :-)

| CPU Model | Microarchitecture | Year |
|---|---|---|
| Intel Xeon E3-1240 v5 | Skylake | 2015 |
| Intel Core i7-6700K | Skylake | 2015 |
| Intel Celeron N2840 | Silvermont | 2014 |
| Intel Xeon E5-2658 v2 | Ivy Bridge EP | 2013 |
| Intel Atom C2750 | Silvermont | 2013 |
| Intel Core i7-4500U | Haswell | 2013 |
| Intel Core i7-3632QM | Ivy Bridge | 2012 |
| Intel Core i7-2620QM | Sandy Bridge | 2011 |
| Intel Core i5 M480 | Westmere | 2010 |
| Intel Core i7 920 | Nehalem | 2008 |
| AMD FX-8350 8-Core | Piledriver | 2012 |
| AMD FX-8320 8-Core | Piledriver | 2012 |
| AMD FX-8120 8-Core | Bulldozer | 2011 |
| AMD Athlon II 640 X4 | K10 | 2010 |
| AMD E-350 | Bobcat | 2010 |
| AMD Phenom 9550 4-Core | K10 | 2008 |
| Allwinner A64 | ARM Cortex A53 | 2016 |
| Samsung Exynos 5800 | ARM Cortex A15 | 2014 |
| Samsung Exynos 5800 | ARM Cortex A7 | 2014 |
| Nvidia Tegra K1 CD580M-A1 | ARM Cortex A15 | 2014 |
| Nvidia Tegra K1 CD570M-A1 | ARM Cortex A15; LPAE | 2014 |

# This side-channel was detected on 22 out of 22 tested architectures!

# Demo video



# Conclusions

- It's possible to perform cache side-channel attacks from Javascript on the Memory Managment Unit to recover ASLR information

- Browser vendors seem to have given up on protecting against side-channel attacks in favor of adding features :,-(

# Any Questions?

**project page:**

https://vusec.net/projects/anc