[544] Caching and PyArrow

Tyler Caraza-Harter

Learning Objectives

- write cache-friendly code with Numpy and PyArrow

- use memory mappings via PyArrow to access data that is larger than physical memory

- enable swapping to alleviate memory pressure

- configure Docker memory limits on physical memory used

Outline

CPU: L1-L3

Demos: Numpy+PyArrow...

Background: Virtual Address Spaces

OS (Operating System): Page Cache

Demos: PyArrow+Docker

Granularity

If a process reads 1 byte and misses, *how much data should the CPU bring into the cache?*

- **too little:** we'll have many more misses if we read nearby bytes soon
- **too much:** wasteful to load data to cache that might never be accessed

L1-L3 cache data in units called **cache lines**

- modern CPUs typically 64 bytes (for example, 8 int64 numbers)
- M1/M2 uses 128

Cache Lines and Misses

cacheline

int64
int64
int64
int64
int64
int64
int64
int64

cacheline

int64
int64
int64
int64
int64
int64
int64
int64

how many
misses?

cacheline

int64
int64
int64
int64
int64
int64
int64
int64

cacheline

int64
int64
int64
int64
int64
int64
int64
int64

how many
misses?

cacheline

int64
int64
int64
int64
int64
int64
int64
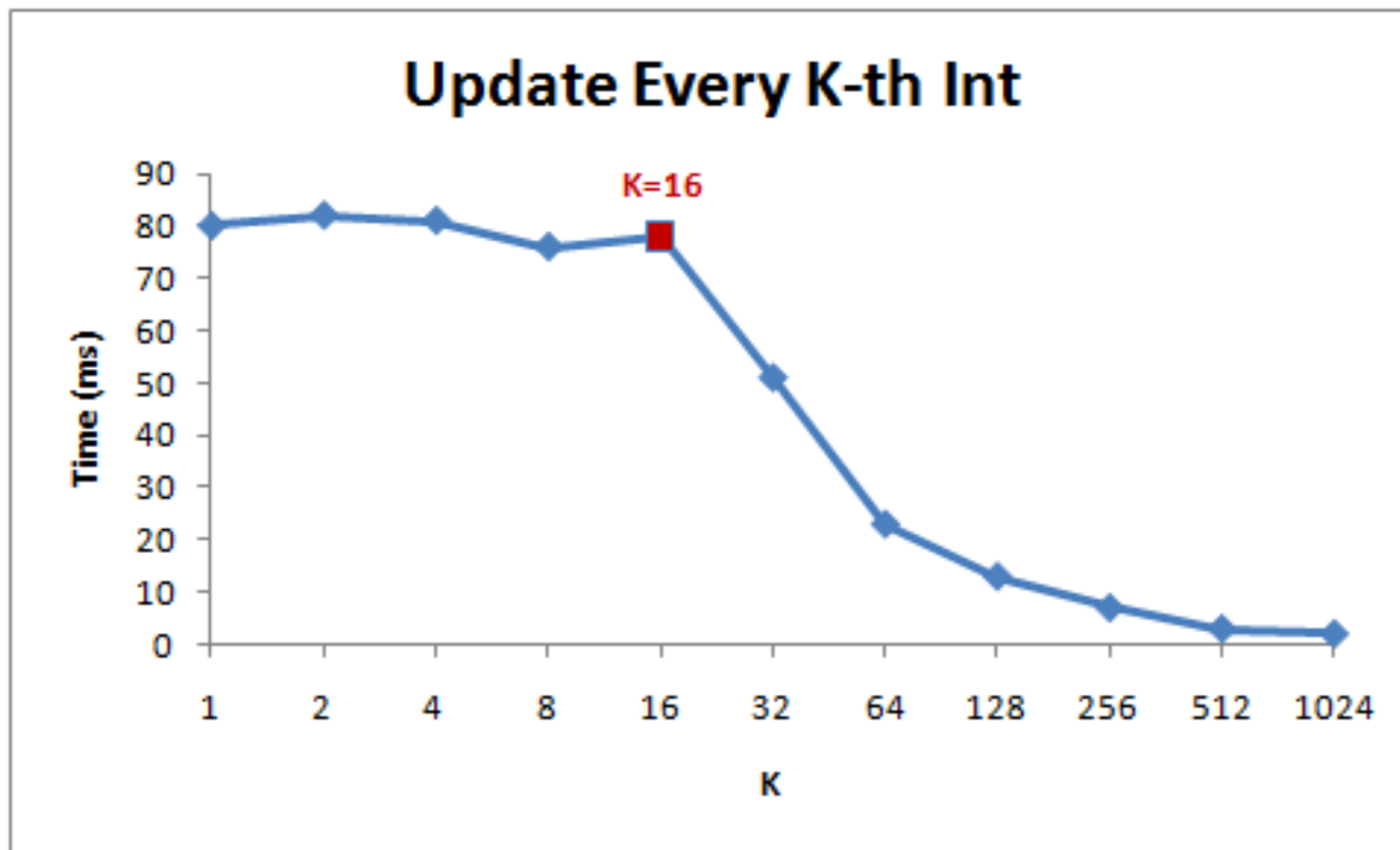int64

cacheline

int64
int64
int64
int64
int64
int64
int64
int64

how many
misses?

Example 1: Step and Multiply

as K gets bigger, we do fewer multiplications. But does it matter?

```
for (int i = 0; i < arr.Length; i += K) arr[i] *= 3;
```
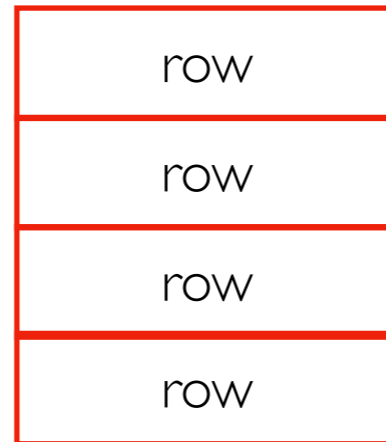
Update Every K-th Int

K=16

Gallery of Processor Cache Effects
http://igoro.com/archive/gallery-of-processor-cache-effects/

Example 2: Matrices

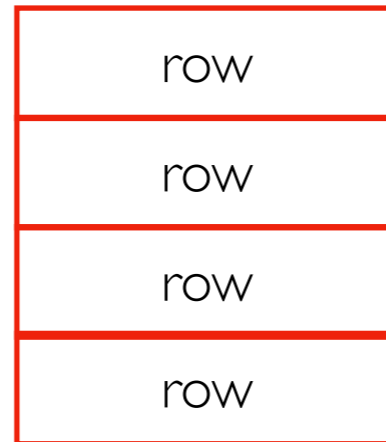matrix of numbers
logically, 2-dimensional

row
row
row
row

physically, those rows are arranged along
1-dimension in the virtual address space

	code		row	row	row	row	...	stack	

0   N

virtual address
spaces

Example 2: Matrices

matrix of numbers
logically, 2-dimensional

row
row
row
row
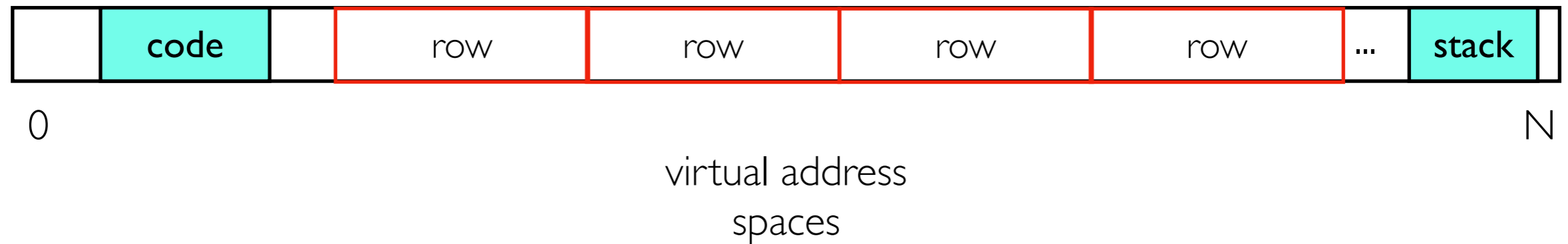
summing over row:
data consolidated over few cache lines

	code		row	row	row	row	...	stack	

0   N
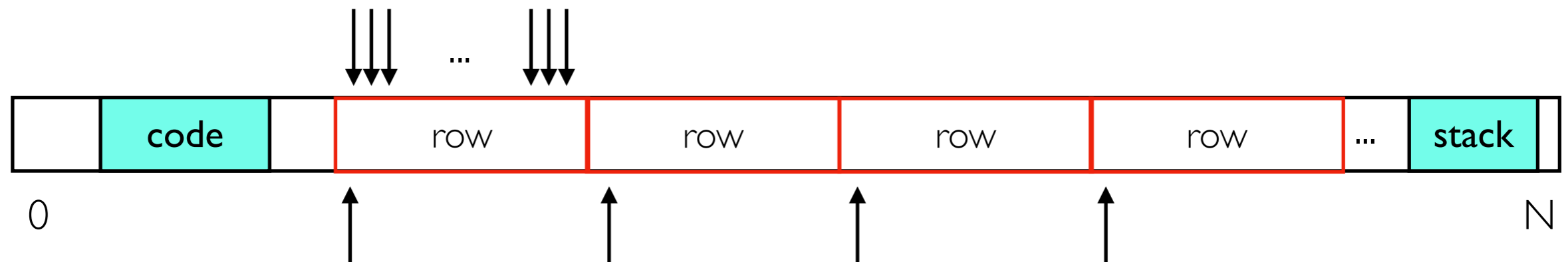
summing over column: each number is in its own cache line and triggers a cache miss
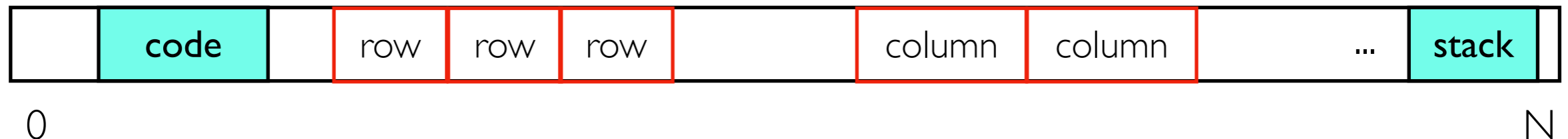
Numpy: Controlling Layout with Transpose

for efficiency, transpose doesn't actually move/copy data,
meaning we can get fast column sum by (a) putting
column data in rows and (b) transposing

```
np.array([[1,2],
          [3,4],
          [5,6]])
```

```
np.array([[1,3,5],
          [2,4,6]]).T
```

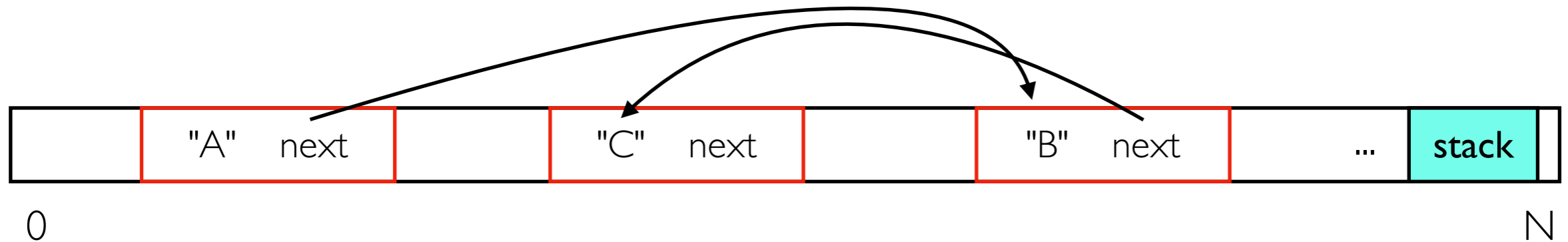| code | | row | row | row | | column | column | ... | stack |

0   N
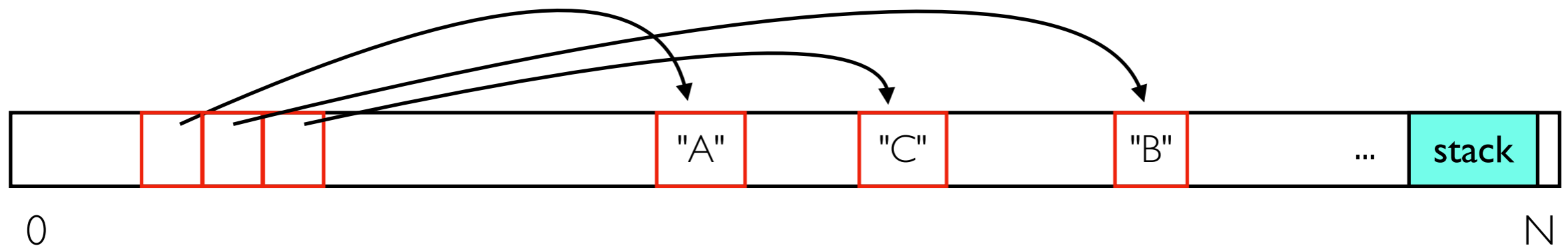
any calculations on the two tensors will produce the same results,
but they'll each be faster for different access patterns!

Example 3: Ordered Collections of Strings

which layout is most cache friendly?

linked list

array of references to strings
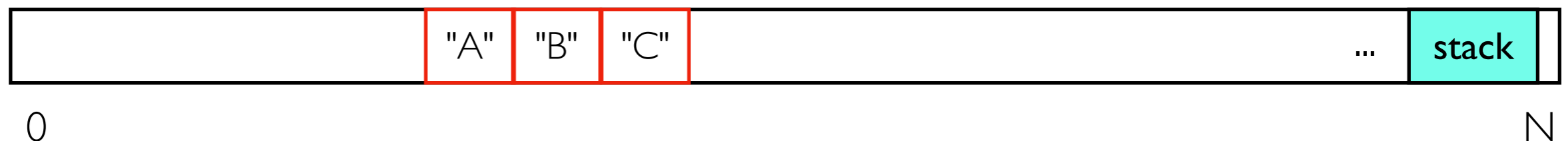
array of inline strings

Example 3: Ordered Collections of Strings

how to tell the end of one string from the start of the next?
how to jump immediately to string at index i?
how support null/None?

		"A"	"B"	"C"		...	stack	

0   N

array of inline strings

PyArrow String Array Data Structure

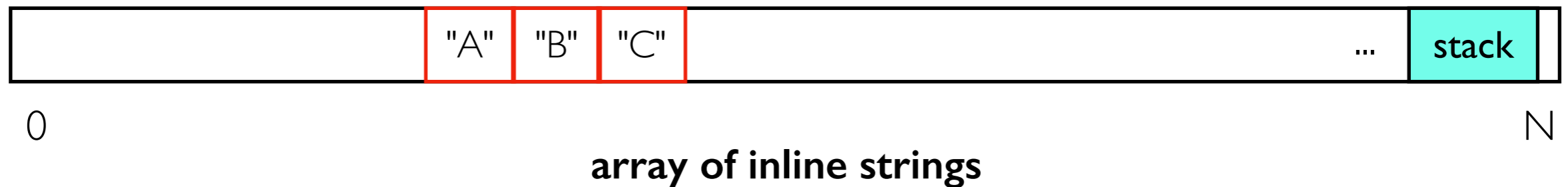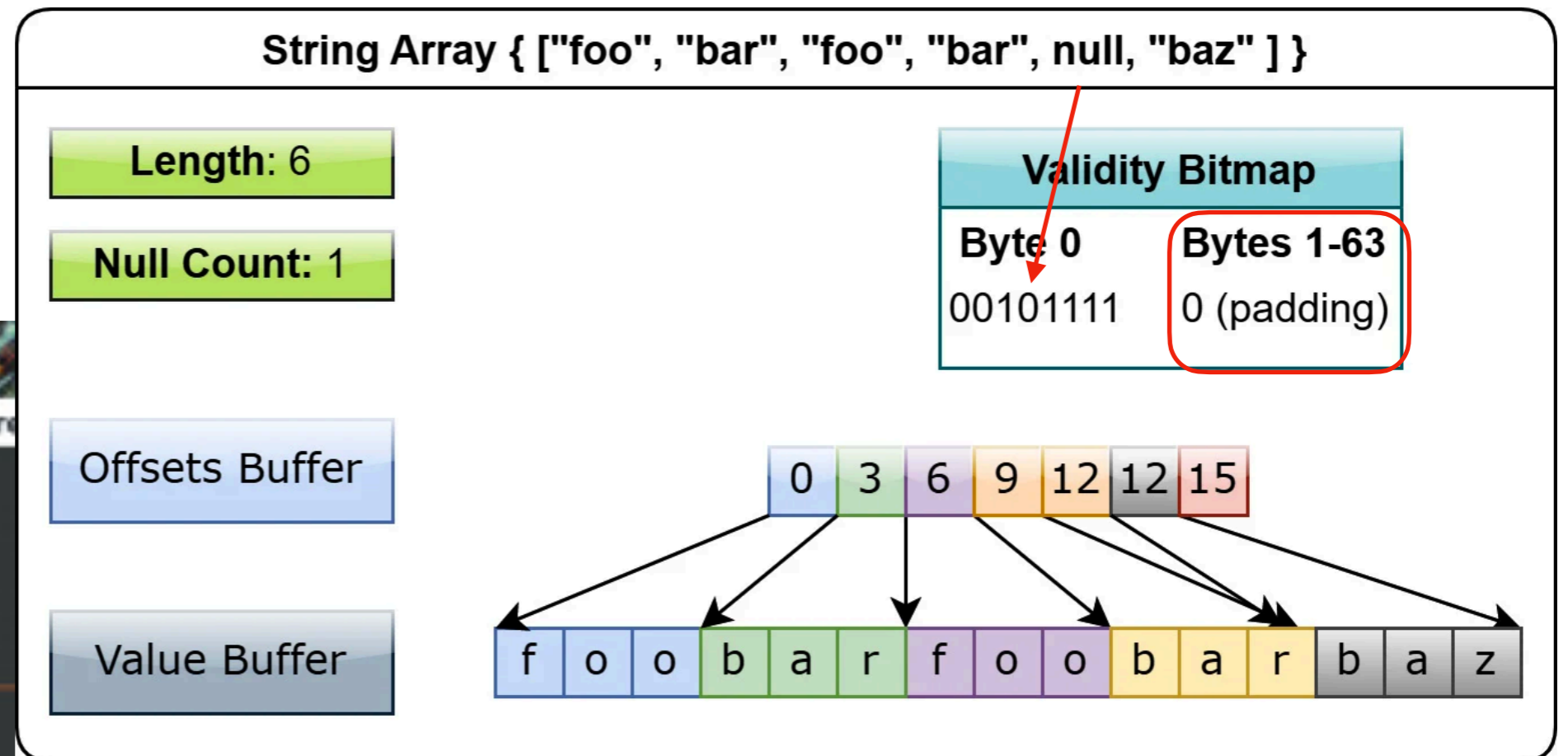**String Array { ["foo", "bar", "foo", "bar", null, "baz"] }**

Length: 6
Null Count: 1

Validity Bitmap	
Byte 0	**Bytes 1-63**
00101111	0 (padding)

Offsets Buffer: 0 3 6 9 12 12 15

Value Buffer: f o o b a r f o o b a r b a z

data is packed into fewest possible cache lines

- collection of named arrays is a Table
- arrays for different types, each cache friendly
- null support for types like int (not forced into floats)
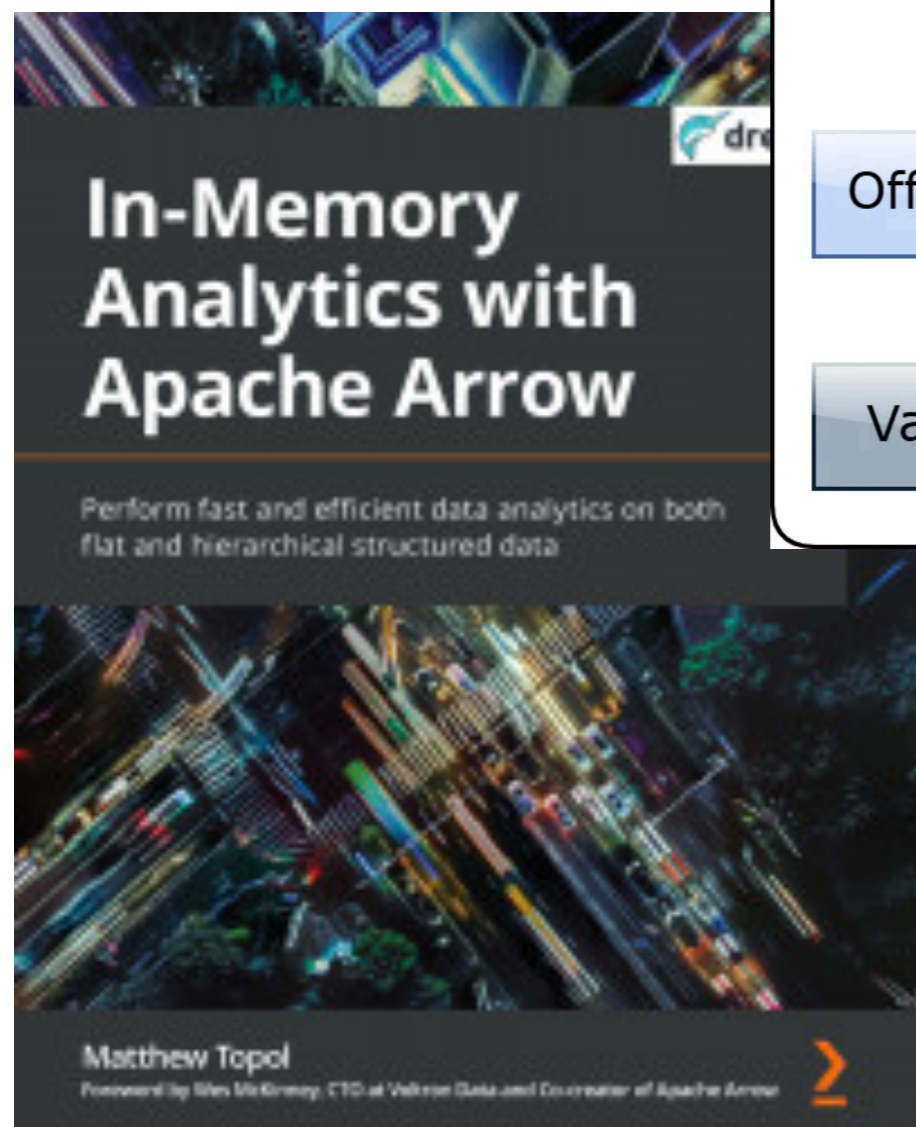
In-Memory Analytics with Apache Arrow

Perform fast and efficient data analytics on both flat and hierarchical structured data

Matthew Topol

Foreword by Wes McKinney, CTO at Voltron Data and Co-creator of Apache Arrow

https://www.packtpub.com/product/in-memory-analytics-with-apache-arrow/9781801071031

Outline

CPU: L1-L3

Demos: Numpy+PyArrow...

Background: Virtual Address Spaces

OS (Operating System): Page Cache

Demos: PyArrow+Docker

Outline

CPU: L1-L3

Demos: Numpy+PyArrow...

Background: Virtual Address Spaces

OS (Operating System): Page Cache

Demos: PyArrow+Docker

Processes and Address Spaces

Address spaces

- A process is a running program
- Each process has it's own virtual address space
- The same virtual address generally refers to different memory in different processes
- Regular processes cannot directly access physical memory or other addr spaces

virtual address spaces

0   N

0   N

"pages" (usually 4 KB) map to physical memory

physical memory

0

M

physical addresses
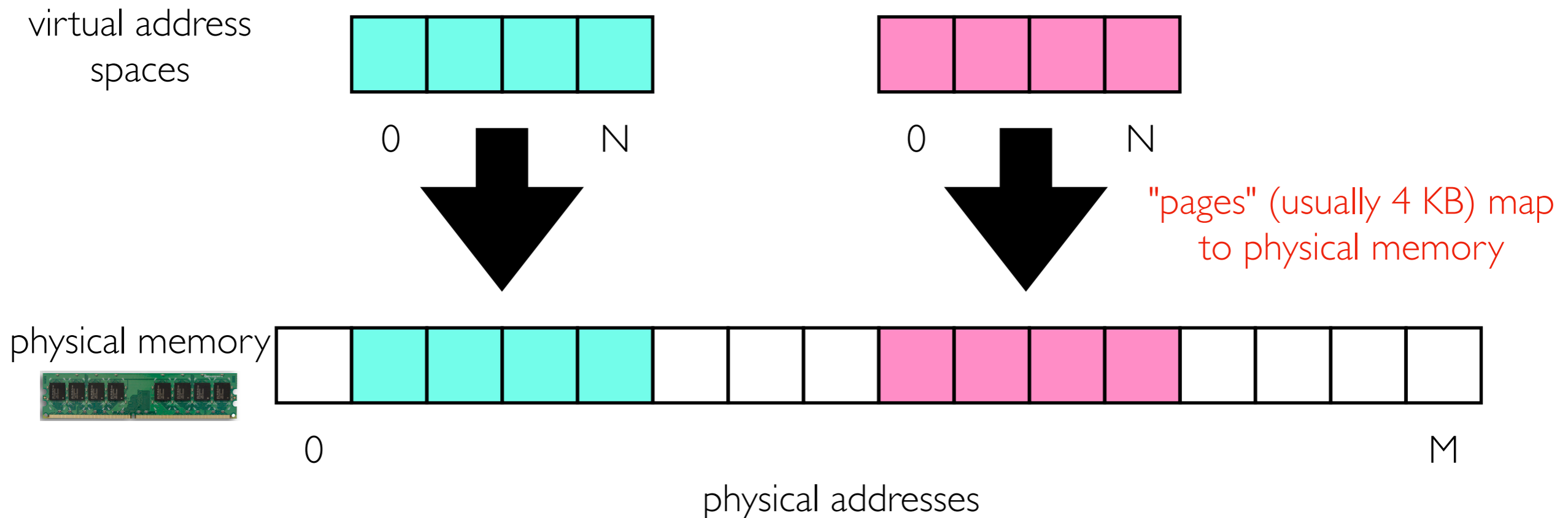
Processes and Address Spaces

Address spaces
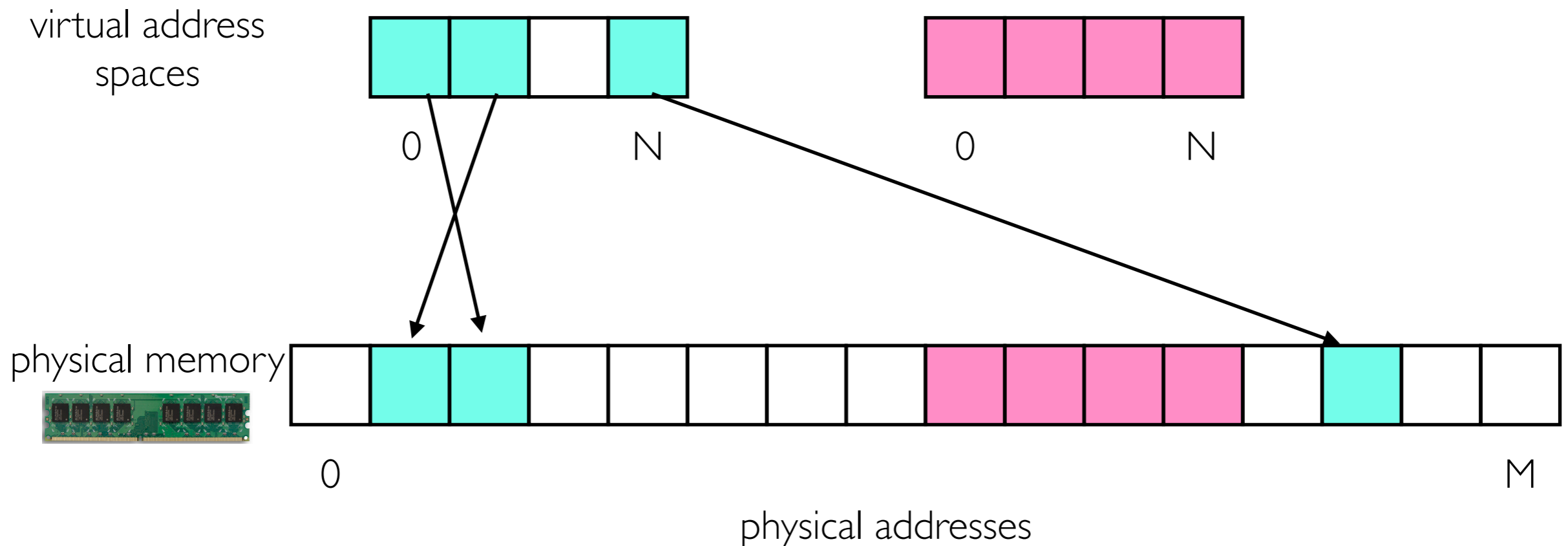
- A process is a running program
- Each process has it's own virtual address space
- The same virtual address generally refers to different memory in different processes
- Regular processes cannot directly access physical memory or other addr spaces
- Address spaces can have holes (N is usually MUCH bigger than M)
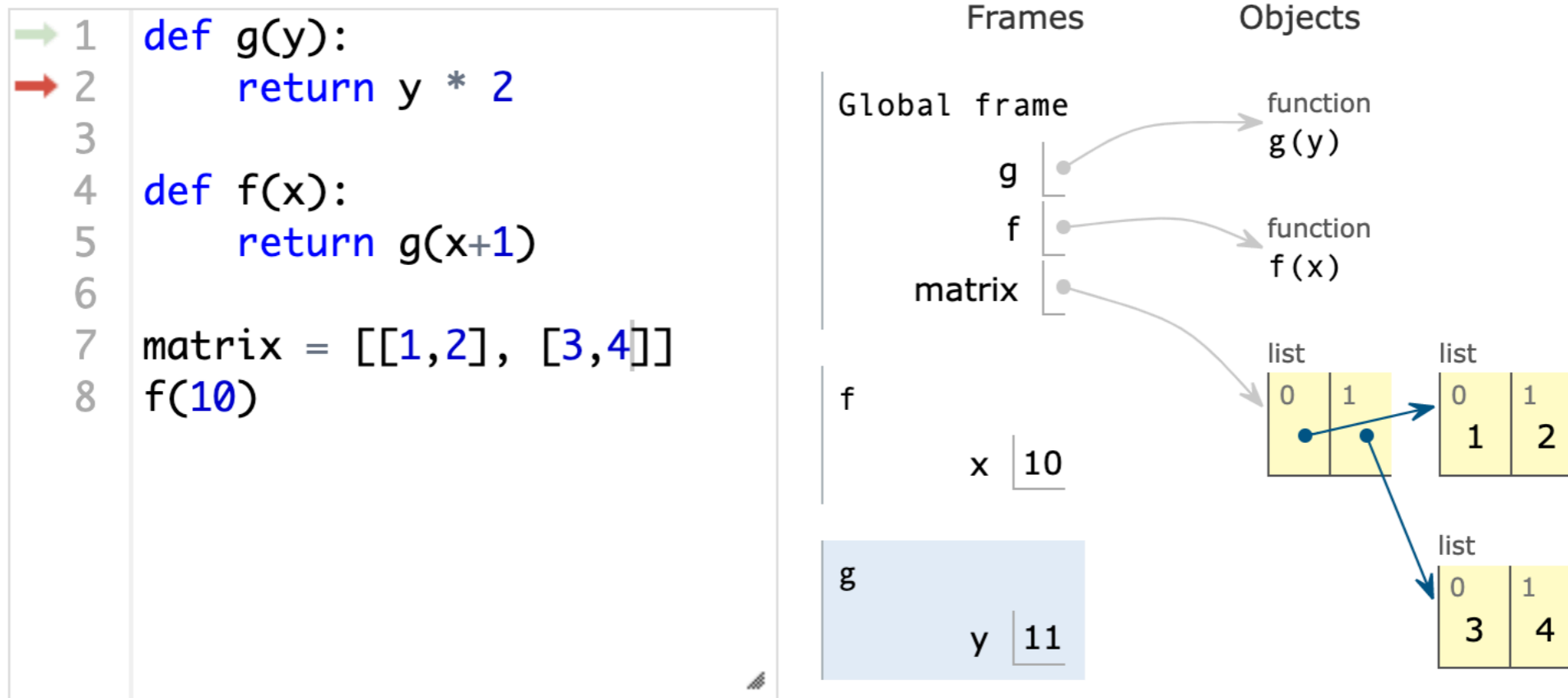- Physical memory for a process need not be contiguous

virtual address spaces

0   N   0   N

physical memory

0   M

physical addresses

What goes in an address space?

```
1  def g(y):
2      return y * 2
3
4  def f(x):
5      return g(x+1)
6
7  matrix = [[1,2], [3,4]]
8  f(10)
```

Frames   Objects

Global frame   function
                                g(y)
                g

                f   function
                                f(x)
            matrix

                                list   list
    f   0 | 1   0 | 1
                                              1 | 2
                x | 10

    g   list
                                            0 | 1
                y | 11   3 | 4
```

https://pythontutor.com/

virtual address spaces

| what goes here? |

0                                                    N

# What goes in an address space?

```
1  def g(y):
2      return y * 2
3
4  def f(x):
5      return g(x+1)
6
7  matrix = [[1,2], [3,4]]
8  f(10)
```

Frames

Objects

Global frame

g → function g(y)

f → function f(x)

matrix →

list
0	1

list
0	1
1	2

list
0	1
3	4

f

x 10

g

y 11

virtual address spaces

	code (Python)		stack		heap	

0

N

Note: code and heap generally not contiguous

What goes in an address space?

```
1  def g(y):
2      return y * 2
3
4  def f(x):
5      return g(x+1)
6
7  matrix = [[1,2], [3,4]]
8  f(10)
```

Frames

Global frame
g
f
matrix

f
x 10

g
y 11

Objects

function
g(y)

function
f(x)

list
| 0 | 1 |

list
| 0 | 1 |
| 1 | 2 |

list
| 0 | 1 |
| 3 | 4 |

some packages
(like numpy)

virtual address
spaces

| | code (Python) | | code (C) | | stack | | heap | |

0   N

How does code execute?

```
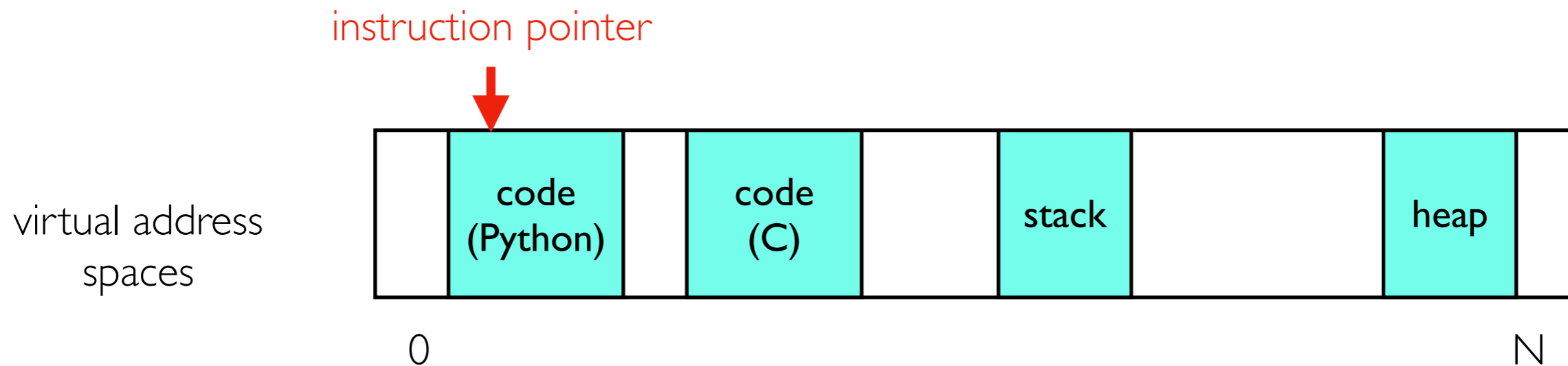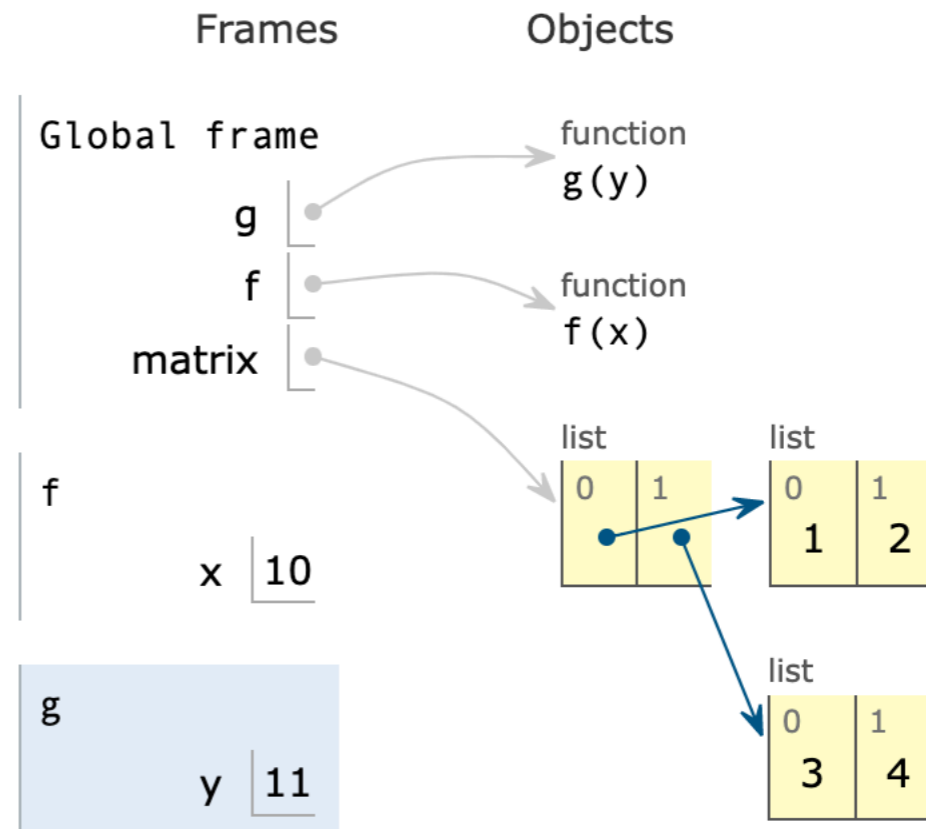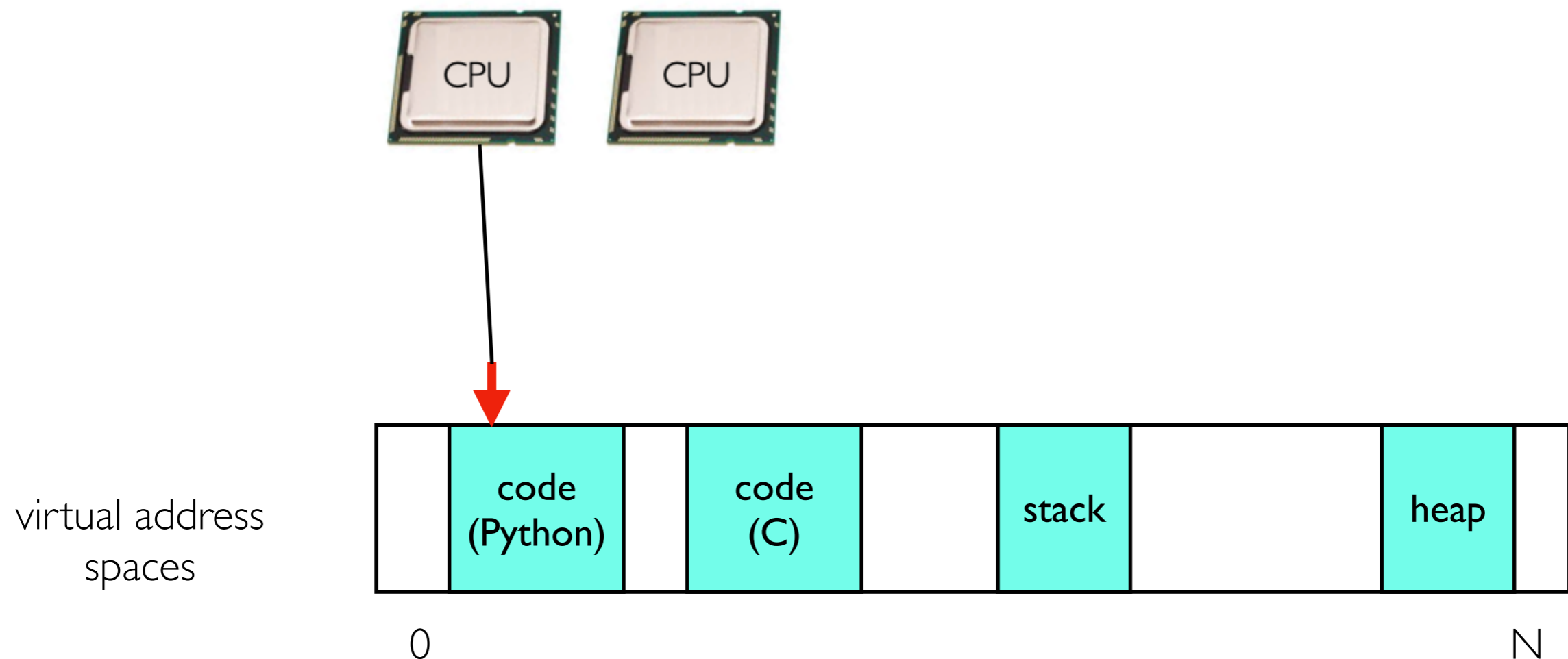1  def g(y):
2      return y * 2
3
4  def f(x):
5      return g(x+1)
6
7  matrix = [[1,2], [3,4]]
8  f(10)
```

Frames

Objects

Global frame

g → function g(y)

f → function f(x)

matrix → list

list
| 0 | 1 |

list
| 0 | 1 |
| 1 | 2 |

f
x | 10

list
| 0 | 1 |
| 3 | 4 |

g
y | 11

instruction pointer

virtual address spaces

| | code (Python) | | code (C) | | stack | | | heap | |

0   N

How does code execute?

CPUs

- CPUs are attached to at most one instruction pointer at any given time
- they run code by executing instructions and advancing the instruction pointer
- **Note**: interpreter left out for simplicity (CPU points to interpreter code, which points to Python bytecode)

virtual address spaces

| | code (Python) | | code (C) | | stack | | heap | |

How does code execute?

CPUs

- CPUs are attached to at most one instruction pointer at any given time
- they run code by executing instructions and advancing the instruction pointer
- **Note**: interpreter left out for simplicity (CPU points to interpreter code, which points to Python bytecode)

virtual address spaces

	code (Python)		code (C)		stack		heap	

0   N

How does code execute?

CPUs

- CPUs are attached to at most one instruction pointer at any given time
- they run code by executing instructions and advancing the instruction pointer
- **Note**: interpreter left out for simplicity (CPU points to interpreter code, which points to Python bytecode)

virtual address spaces

	code (Python)		code (C)		stack		heap	

0   N

call numpy function

Outline

CPU: L1-L3

Demos: Numpy+PyArrow...

Background: Virtual Address Spaces

OS (Operating System): Page Cache

Demos: PyArrow+Docker

mmap (Memory Map)

An mmap call can add new regions to a virtual address space. Two varities:
- anonymous
- backed by a file

virtual address
spaces

0   N

physical memory

0   physical addresses   M

Anonymous mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

anonymous   3 pages

```
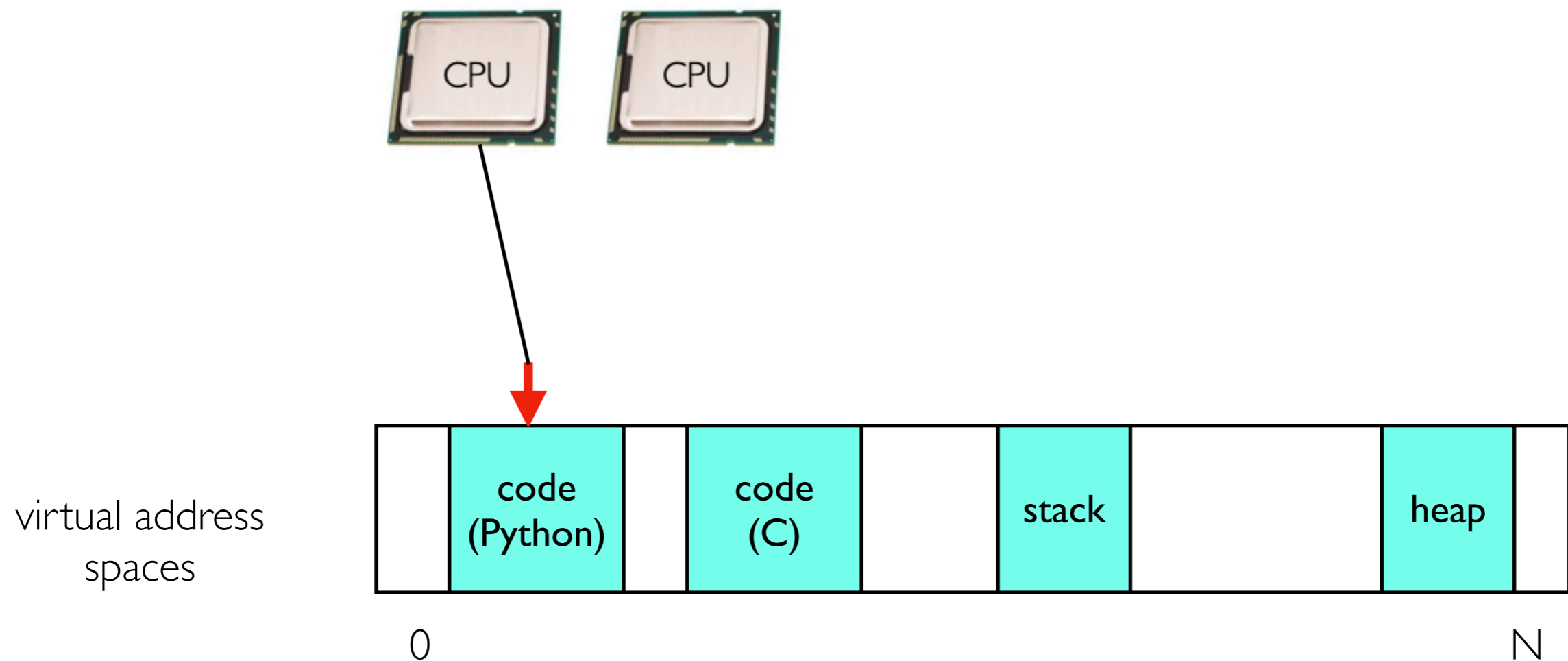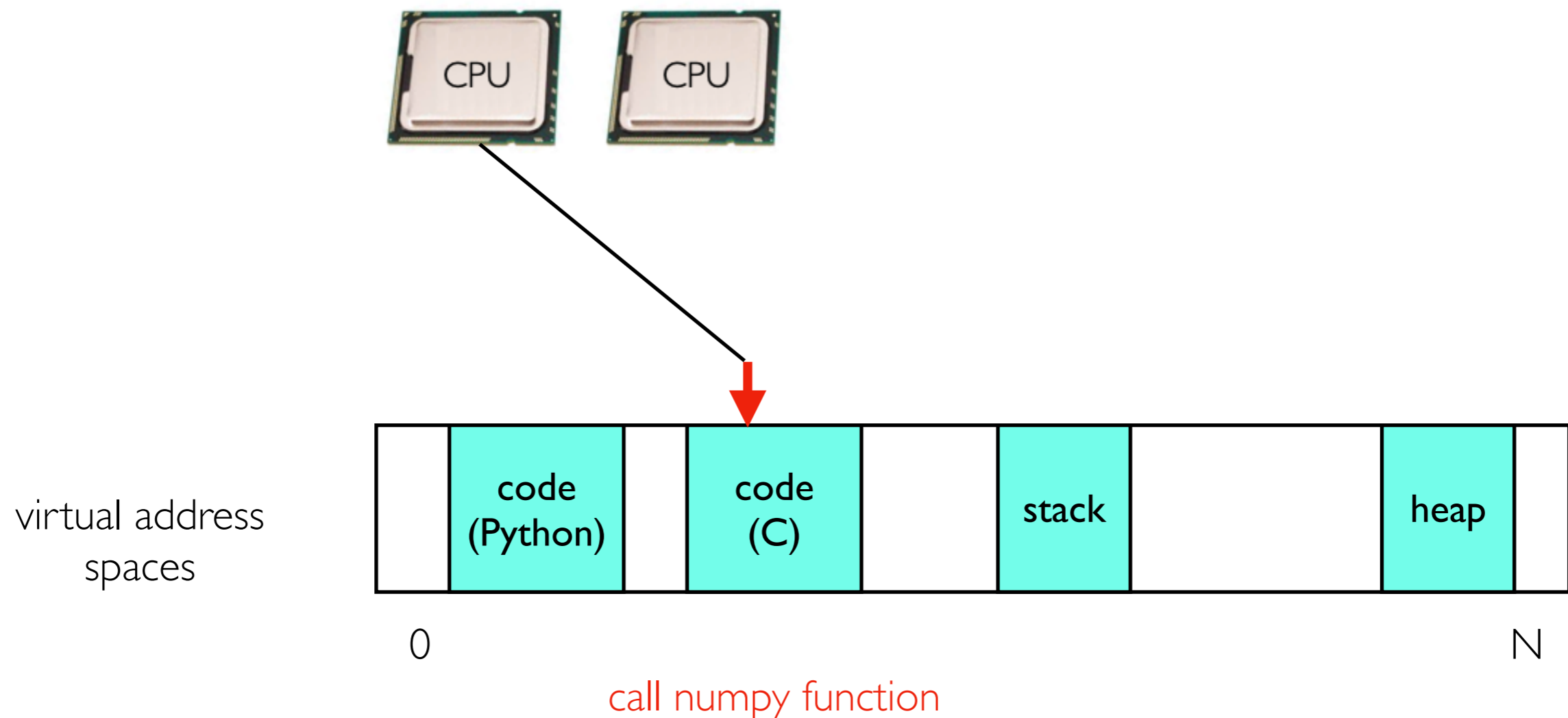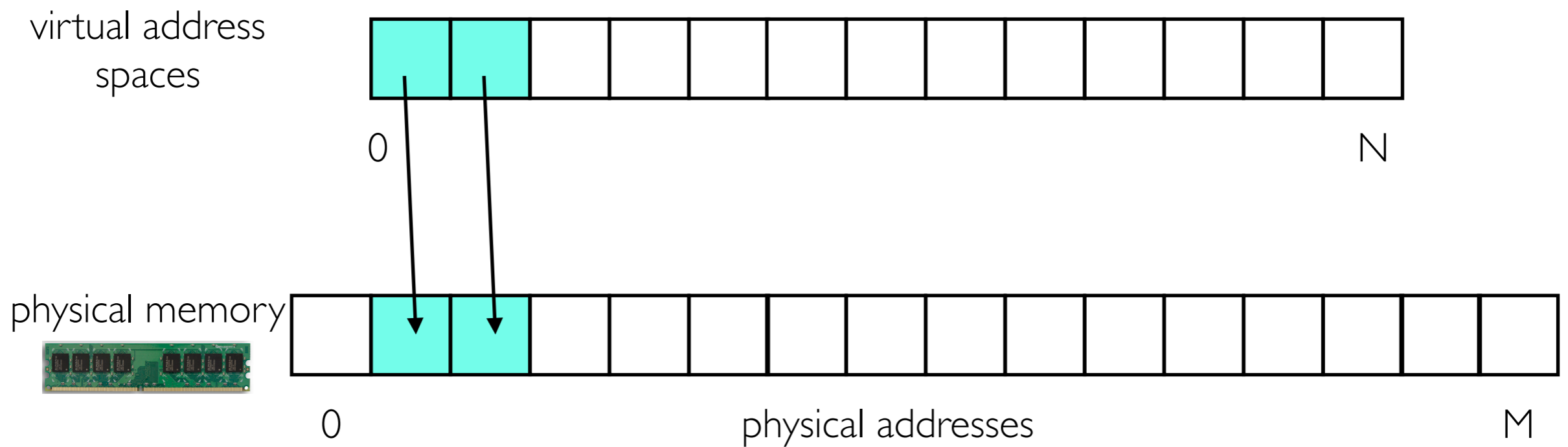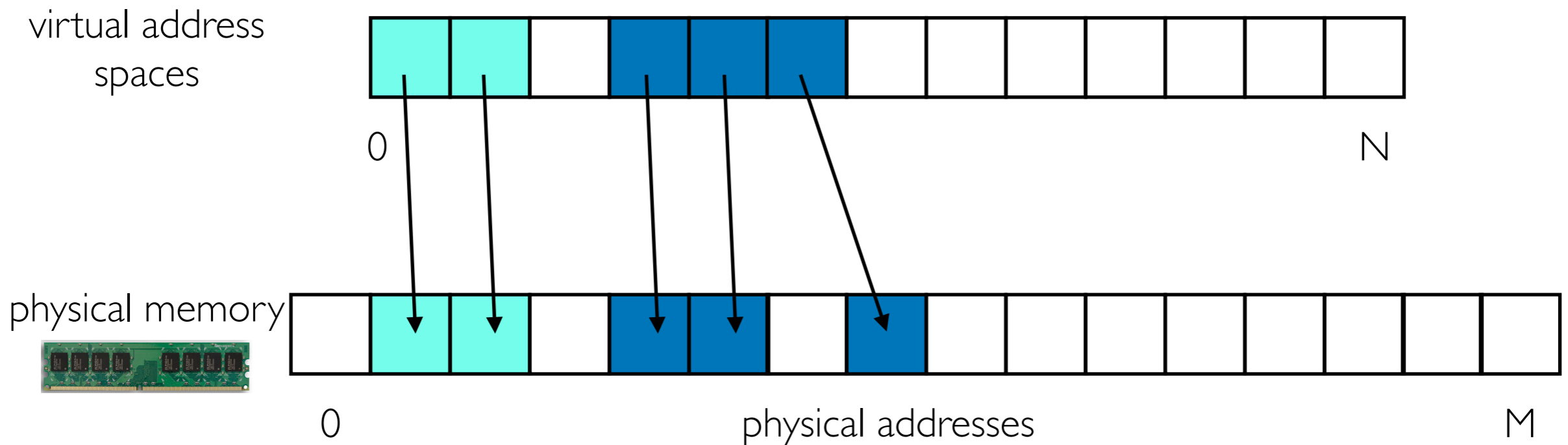import mmap
mm = mmap.mmap(-1, 4096*3)
```


virtual address
spaces

0   N

physical memory

0   physical addresses   M

- Python (and other language runtimes) will mmap some anonymous memory when they need more heap space
- this will be used for Python objects (ints, lists, dicts, DataFrames, etc.)

File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

```python
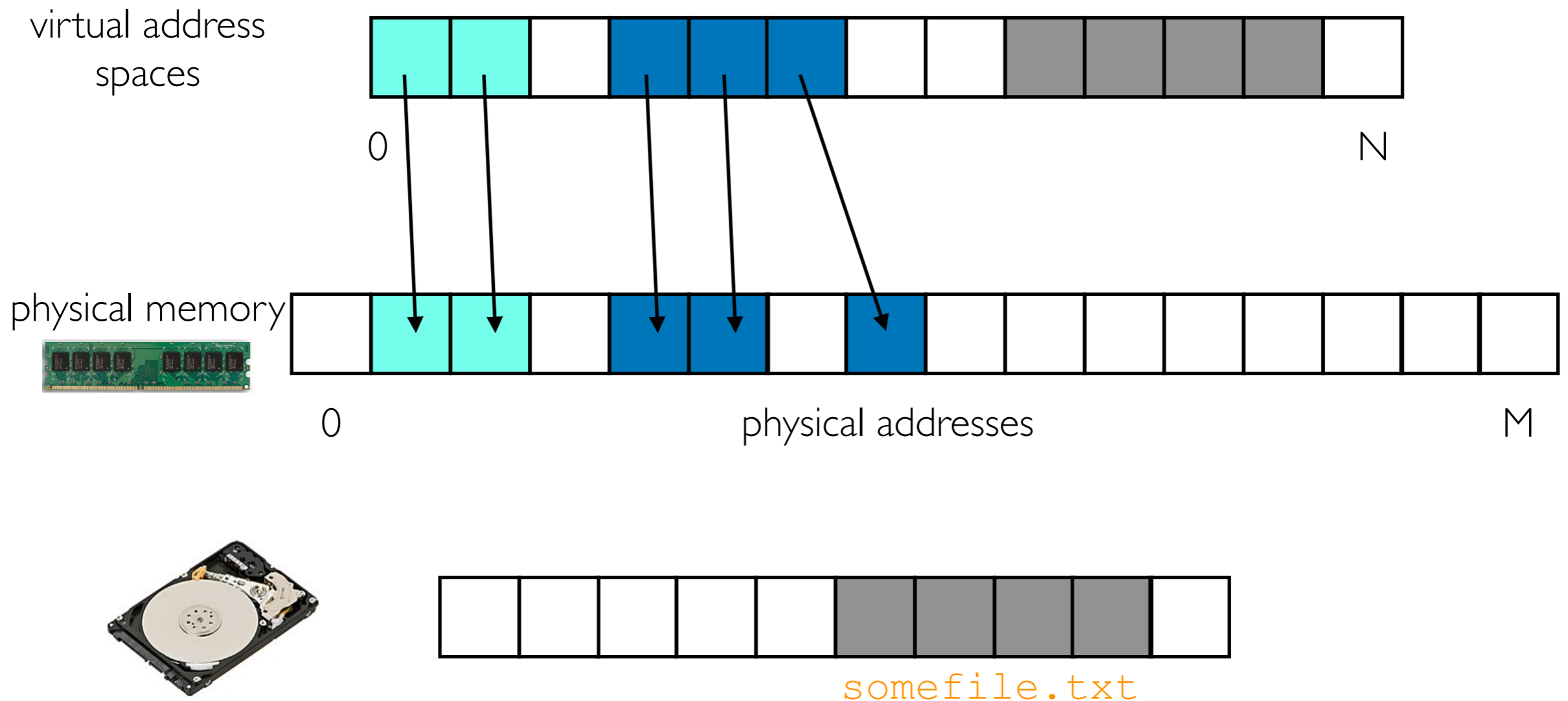import mmap
f = open("somefile.txt", mode="rb")
mm = mmap.mmap(f.fileno(), 0,    # 0 means all
               access=mmap.ACCESS_READ)
```


virtual address spaces

0   N

physical memory

0   physical addresses   M

somefile.txt

File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:
- anonymous
- backed by a file

read data

virtual address
spaces

0   N

physical memory

0   M

read from disk

somefile.txt

File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

read data

virtual address spaces

0   N

physical memory

0   M

read from disk

somefile.txt
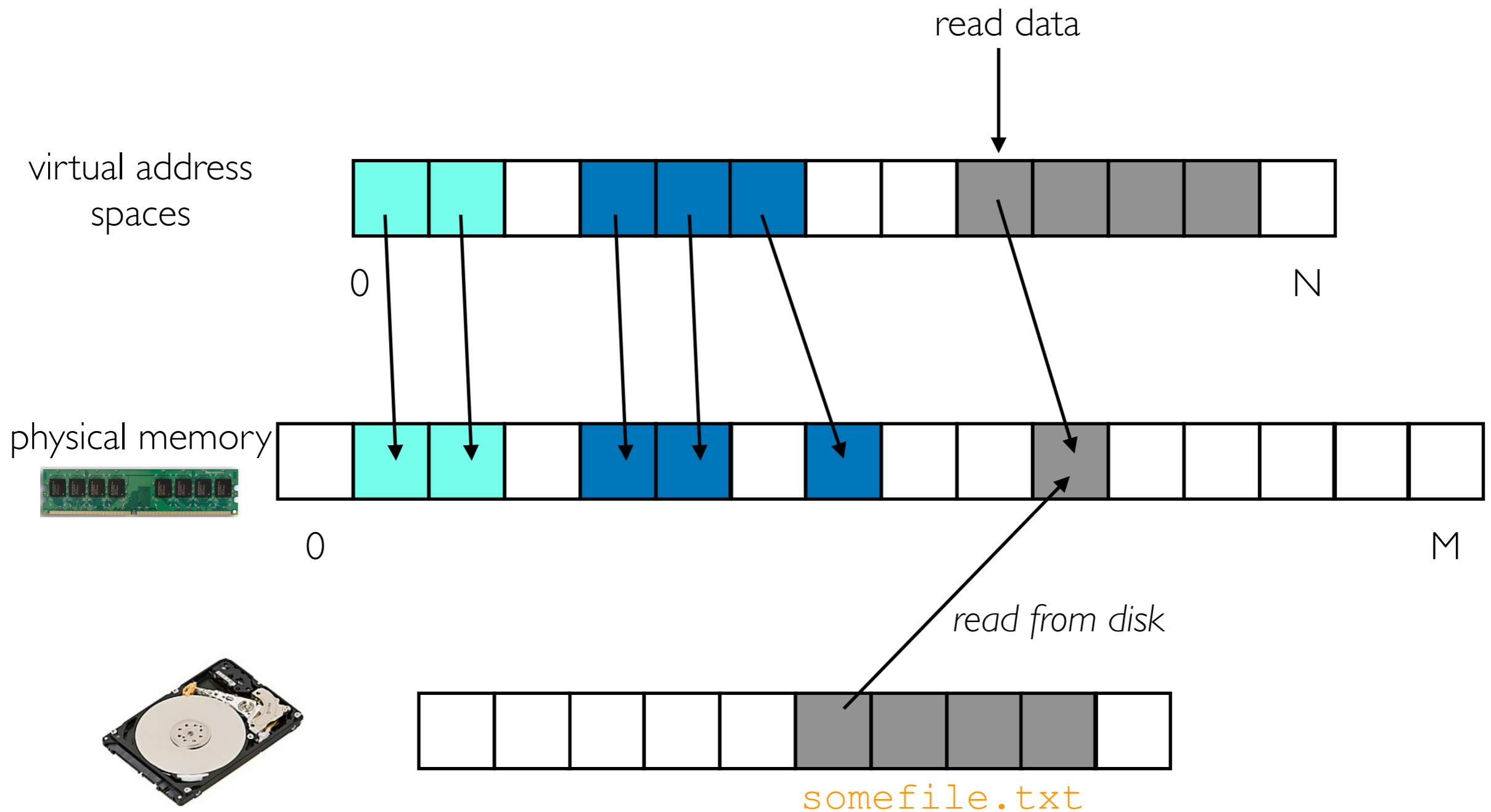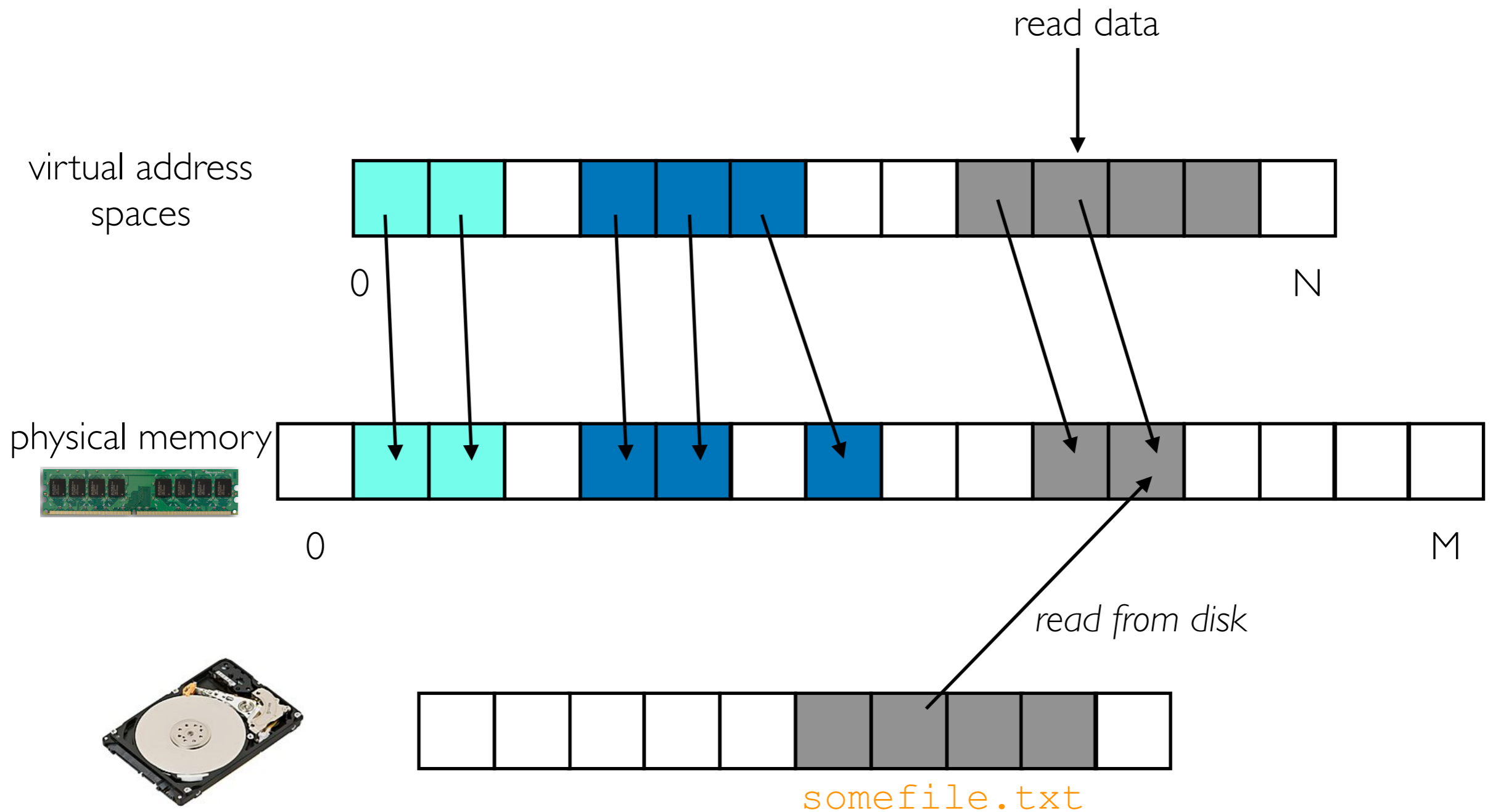
File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

- **virtual** memory used: 9*pagesize = 36 KB
- **physical** memory used: 7*pagesize = 28 KB
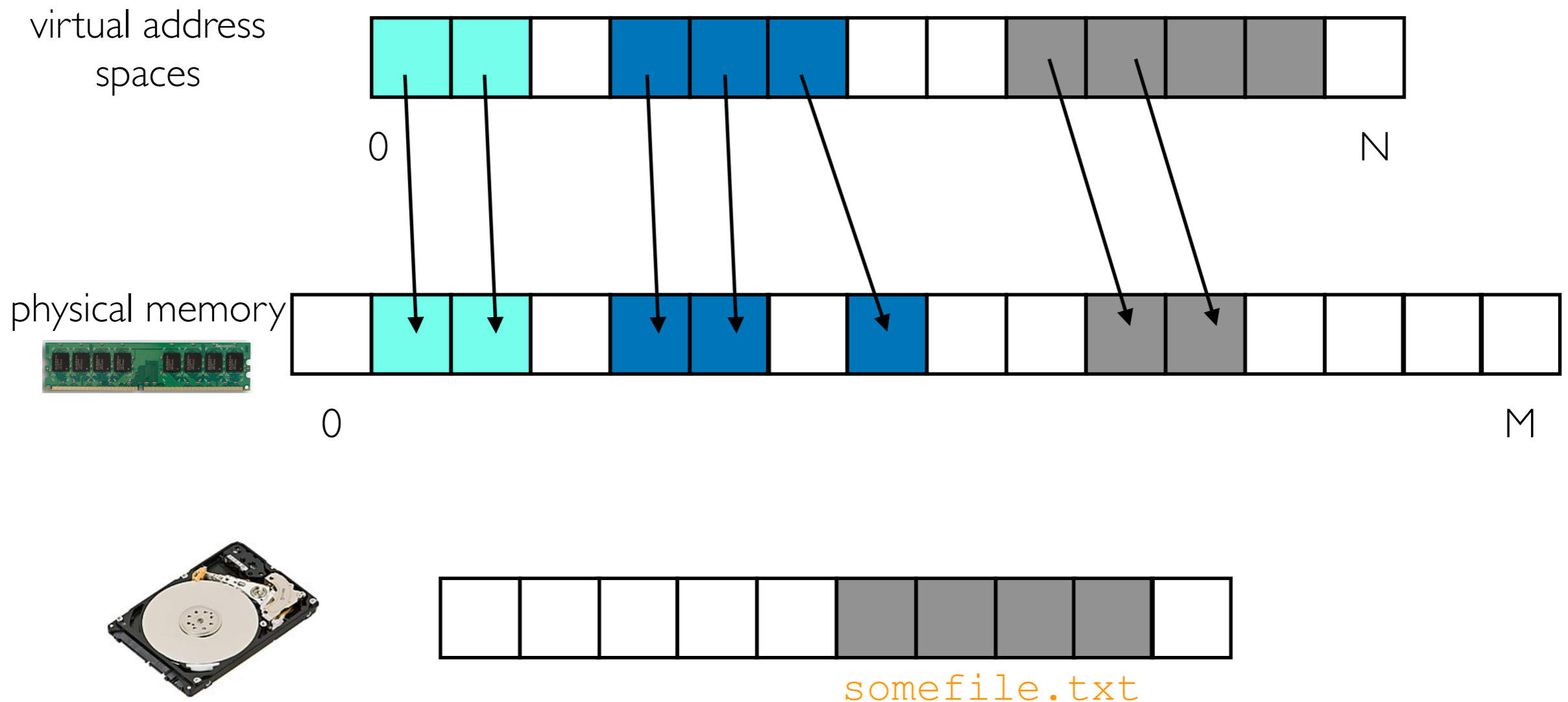
virtual address spaces

0   N

physical memory

0   M

somefile.txt

File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

- data loaded for accesses to file-backed mmap regions are part of the "page cache"
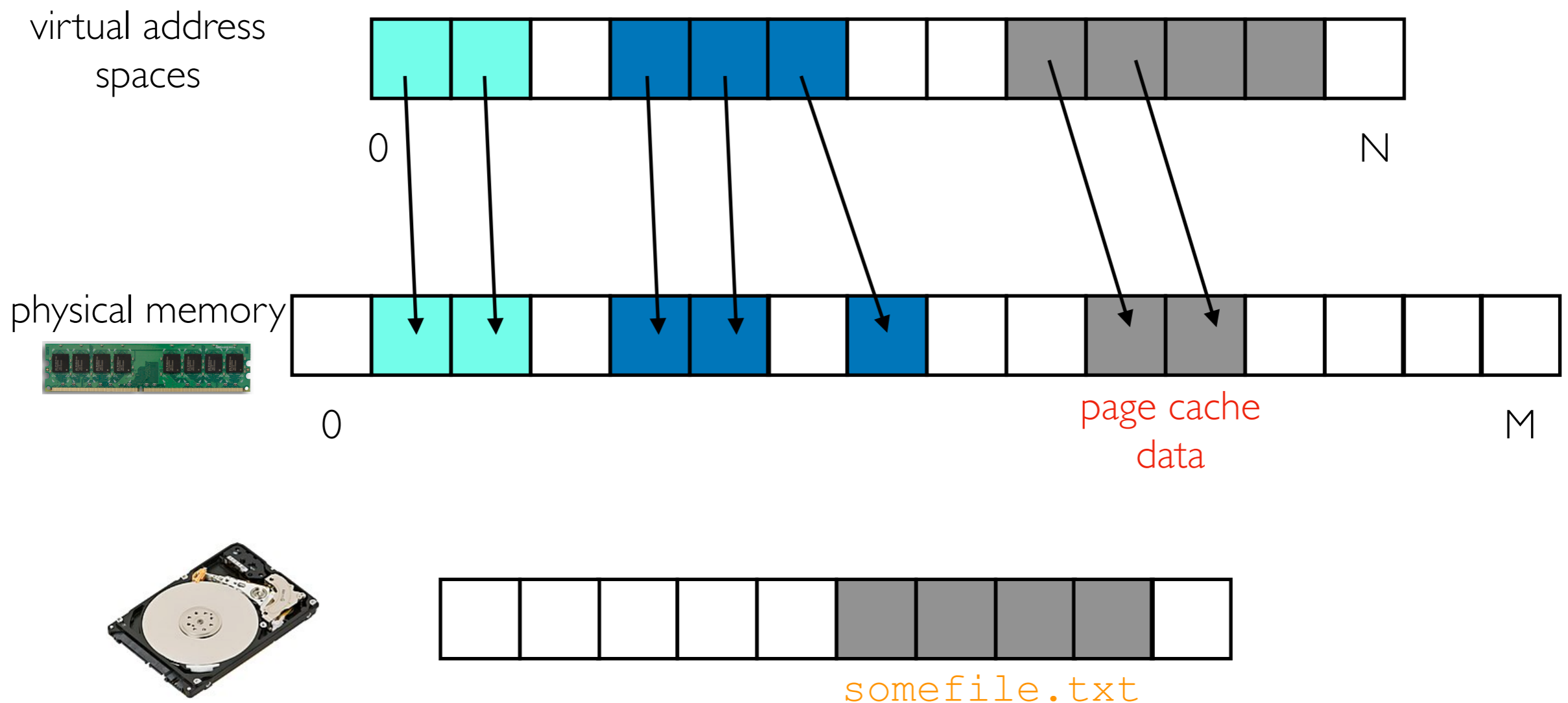
File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

- data loaded for accesses to file-backed mmap regions are part of the "page cache"
- it works like a cache because there is another copy on disk, so we can evict under memory pressure
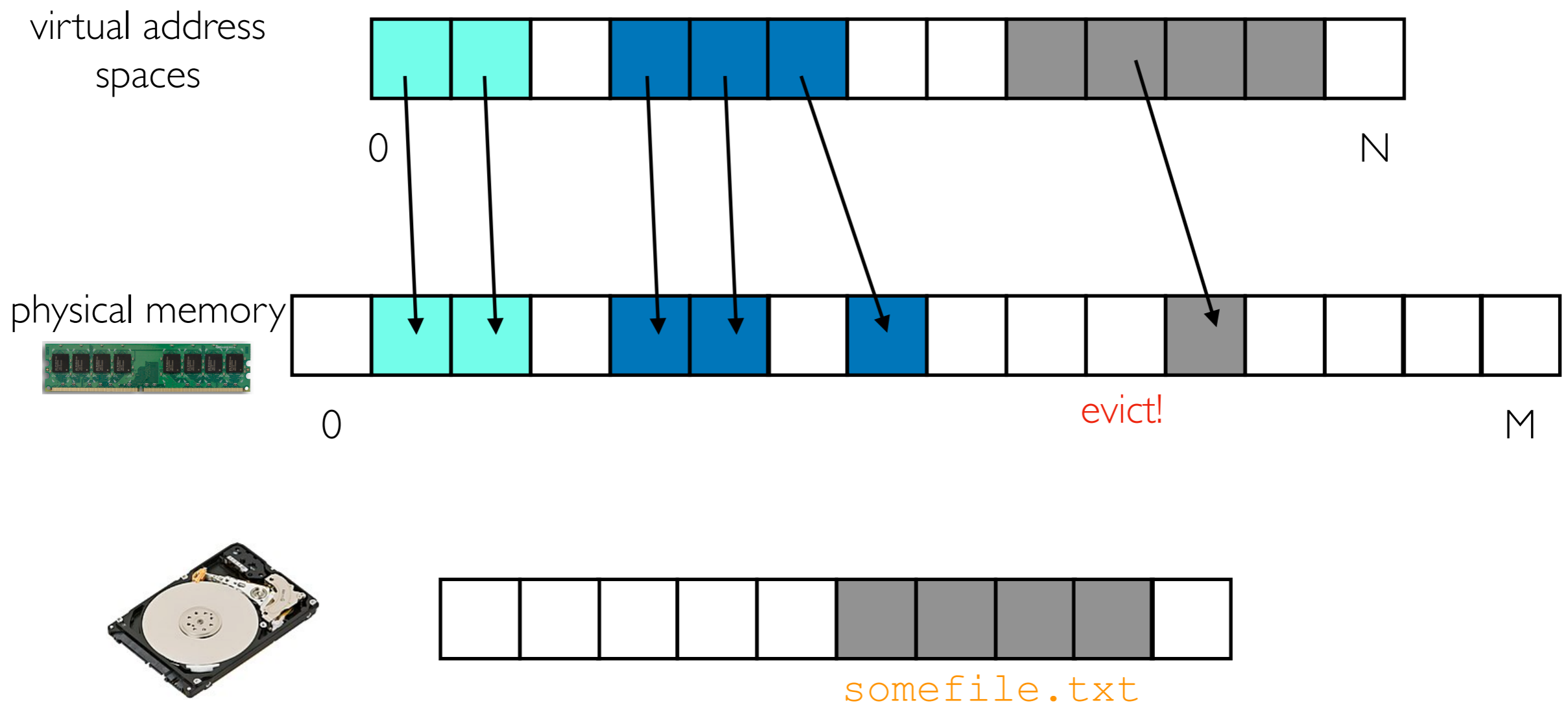
virtual address spaces

0   N

physical memory

0   evict!   M

somefile.txt

Swap Space

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

virtual address spaces

0   N

physical memory

0

can we every evict data for
anonymous mmap data?

M

somefile.txt

Swap Space

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

- we can create same space (a swap file) to which the OS can evict data from anonymous mappings

virtual address spaces

0   N

physical memory

0   M

can we every evict data for anonymous mmap data?

swap file   somefile.txt
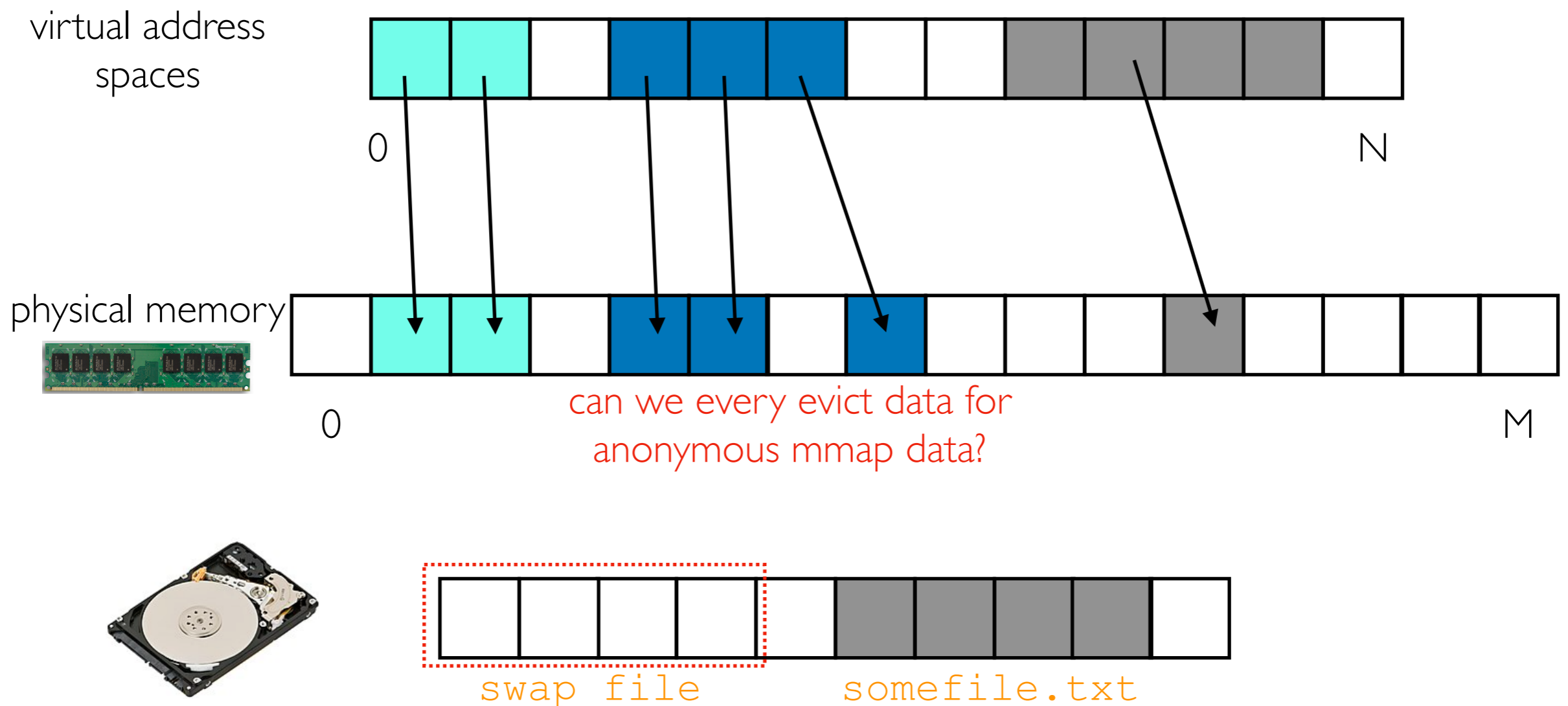
Swap Space

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file

- we can create same space (a swap file) to which the OS can evict data from anonymous mappings
- of course, if we access these virtual addresses again, it will be slow to bring the data back

virtual address spaces

0   N

physical memory

0   M

evict!   evict!

swap file   somefile.txt

Outline

CPU: L1-L3

Demos: Numpy+PyArrow...

Background: Virtual Address Spaces

OS (Operating System): Page Cache

Demos: PyArrow+Docker