[544] Advanced Concurrency (part 1)

[544] Block Devices (part 2)

Tyler Caraza-Harter

Learning Objectives

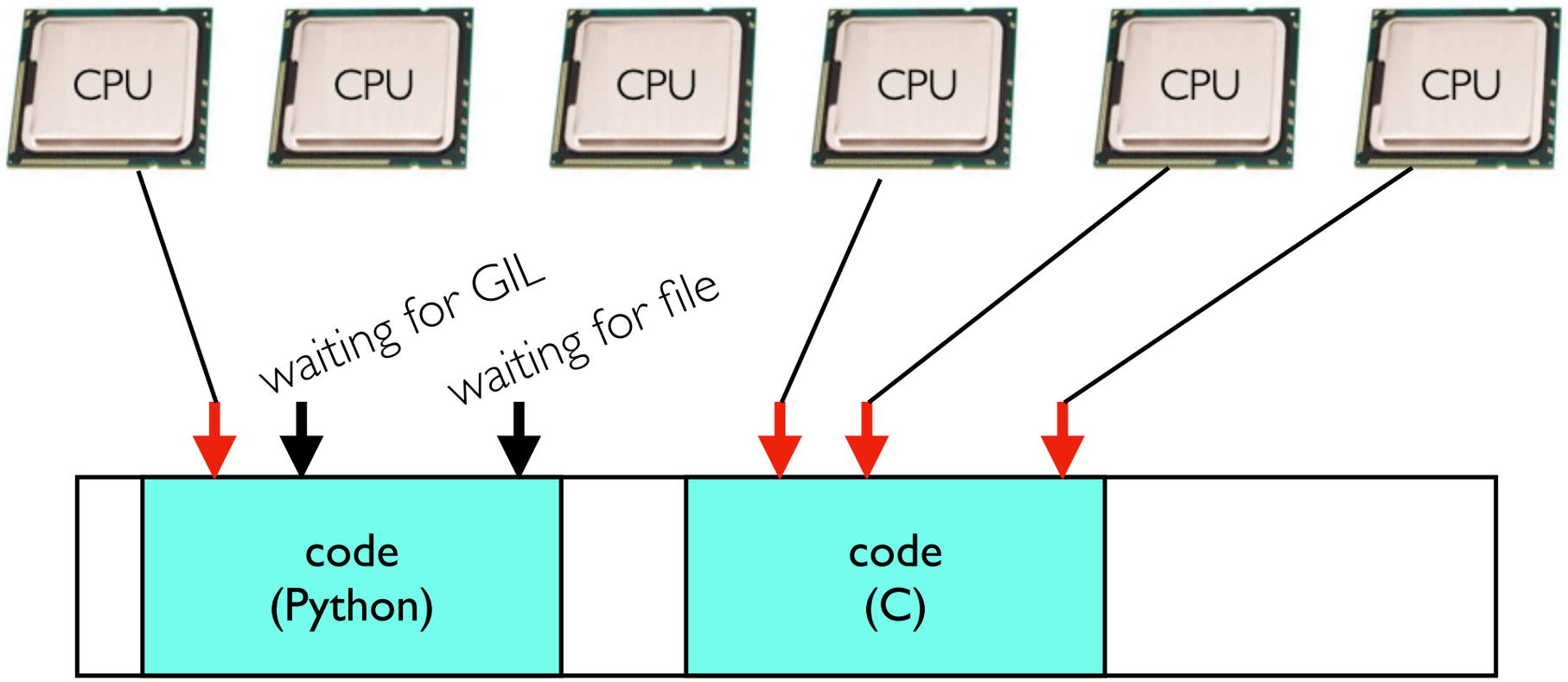
- explain the purpose of the global interpreter lock (GIL)
- avoid GIL-related performance problems
- describe reasons for using locks beyond avoiding certain scheduler interleavings (particularly, instruction reordering and caching)

Outline

Global Interpreter Lock

Instruction Reordering and Caching

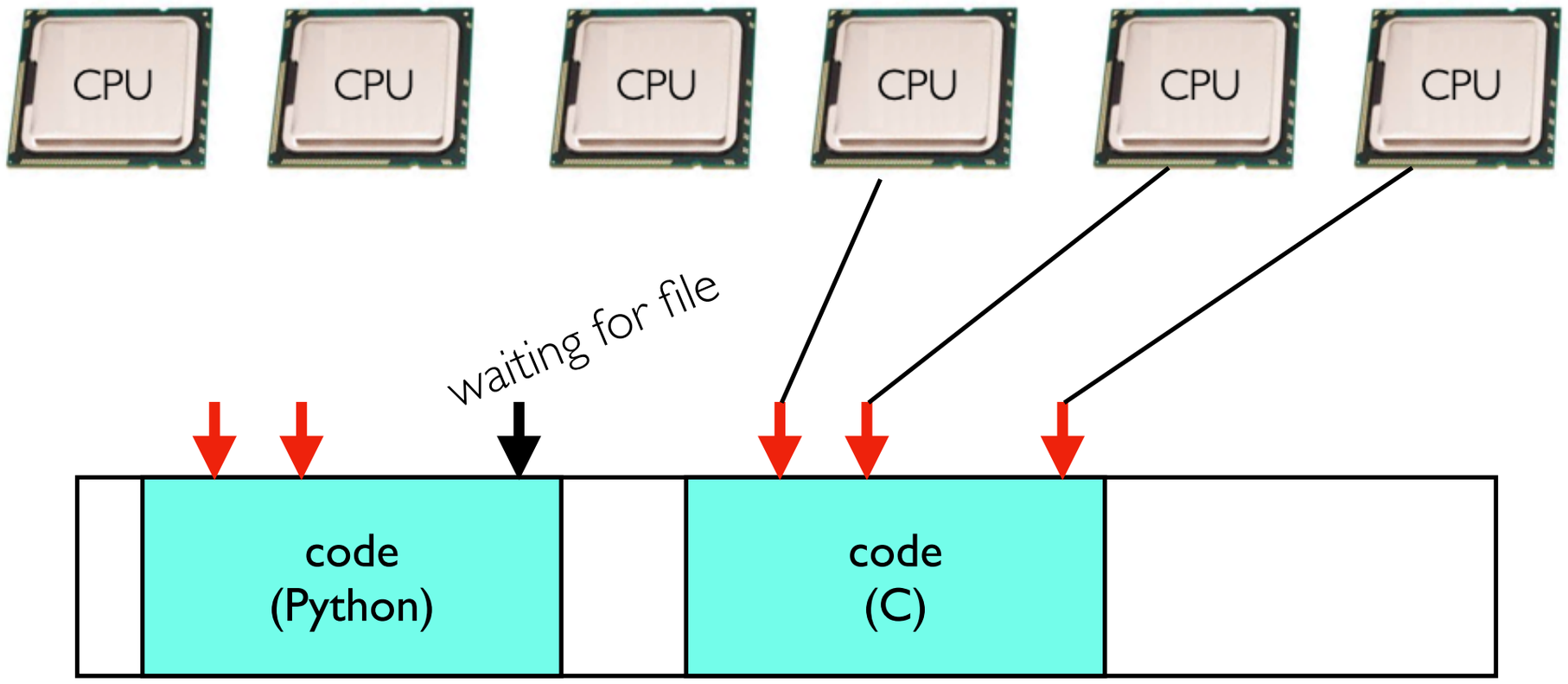
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python bytecode in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

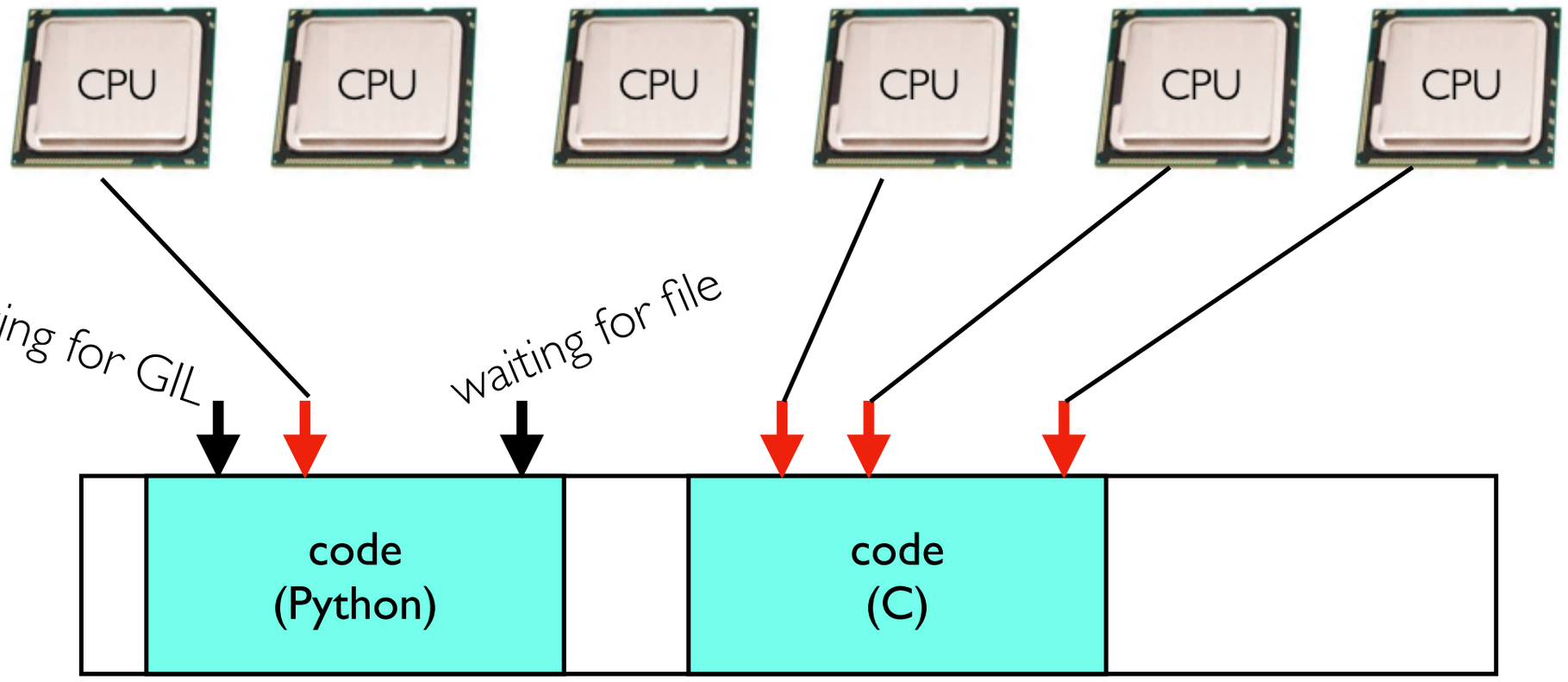
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python bytecode in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

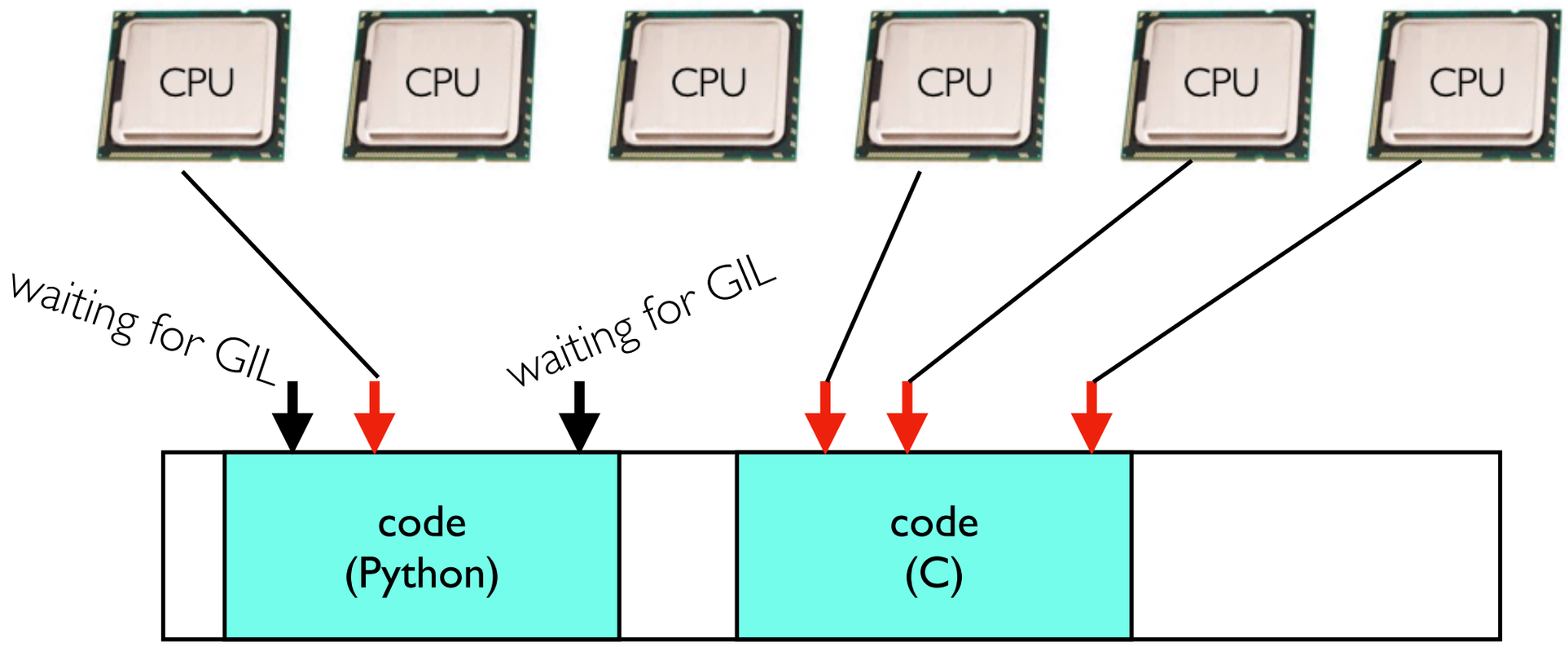
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

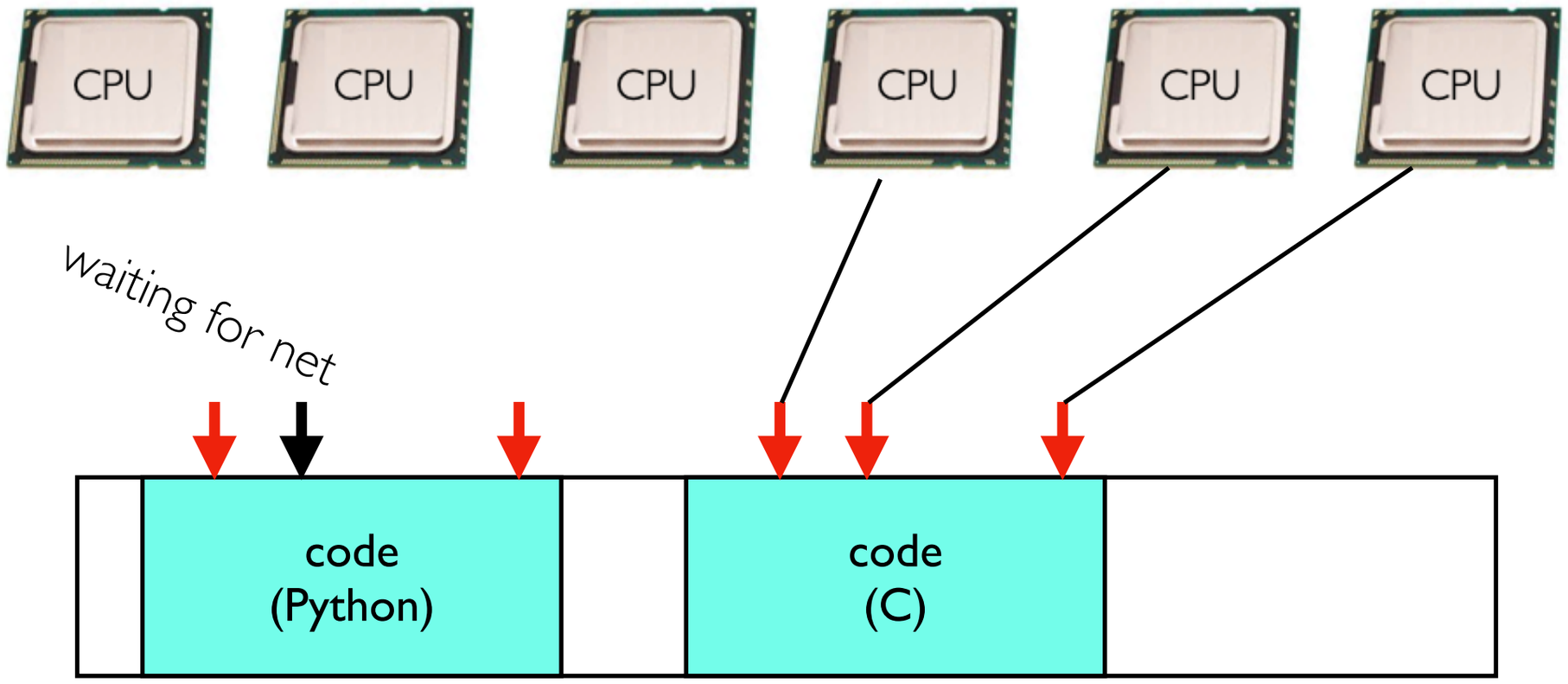
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

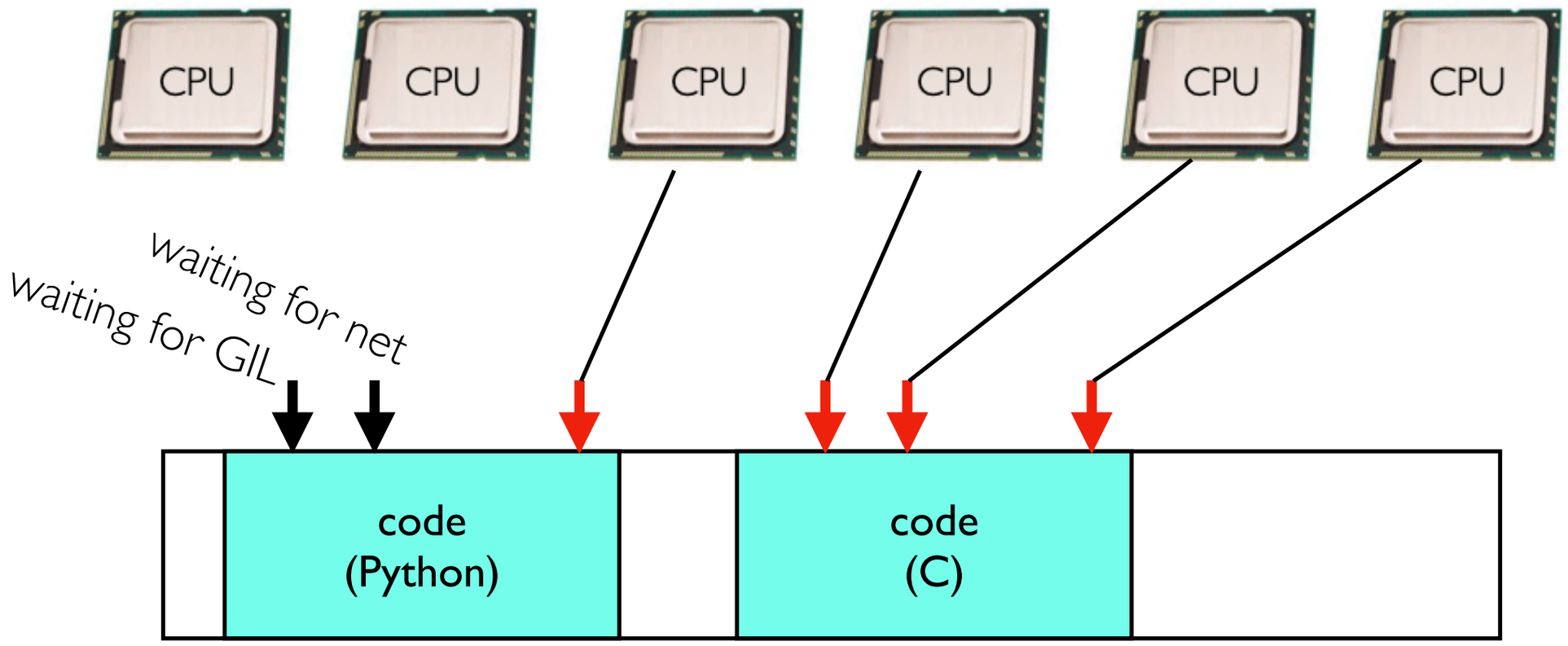
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

Python's GIL (Global Interpreter Lock)

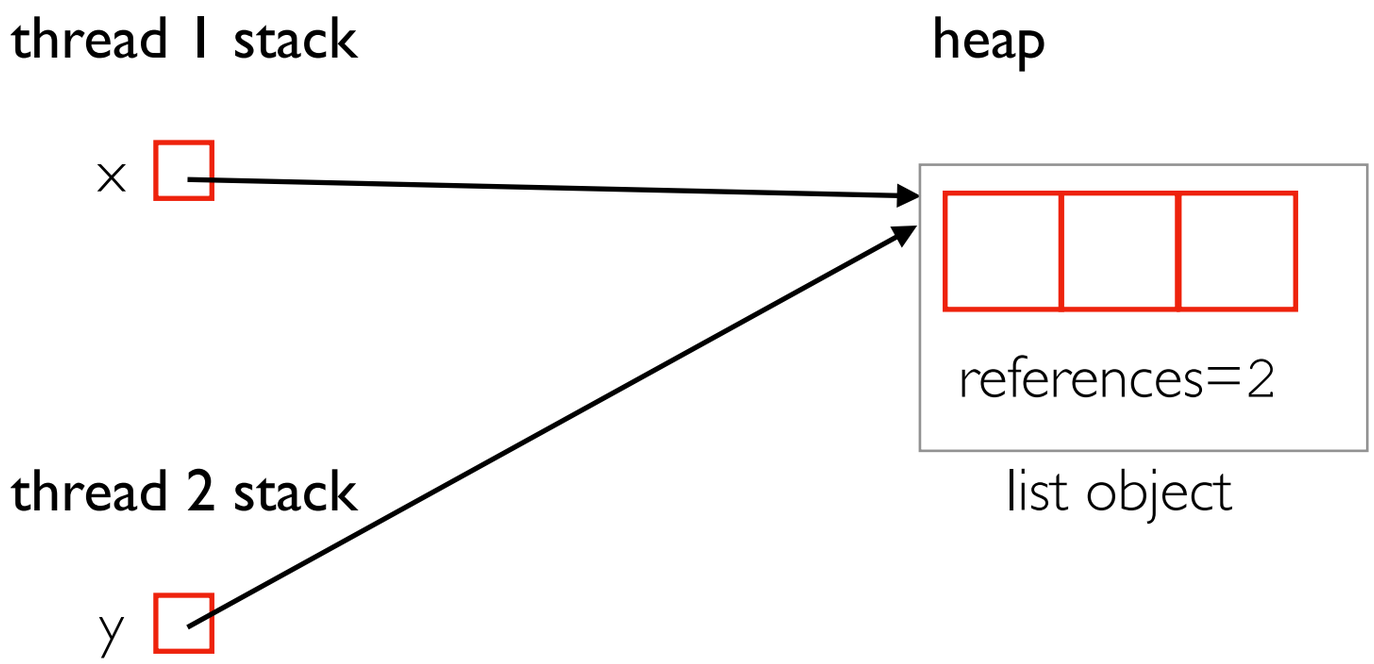


Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

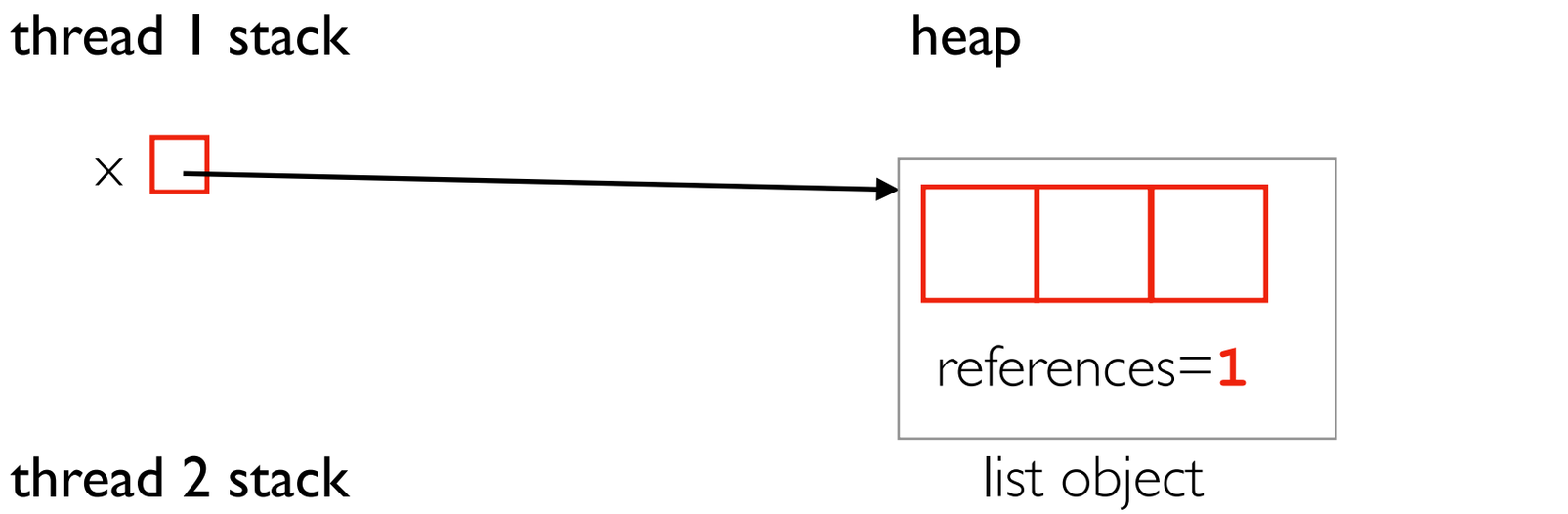
Why the GIL? Garbage collection!

```
# thread 1
x = some list
x = None

# thread 2
y = that same list
y = None
```



Why the GIL? Garbage collection!

```
# thread 1
x = some list
x = None

# thread 2
y = that same list
y = None
```



object will be freed when references is 0

Why the GIL? Garbage collection!

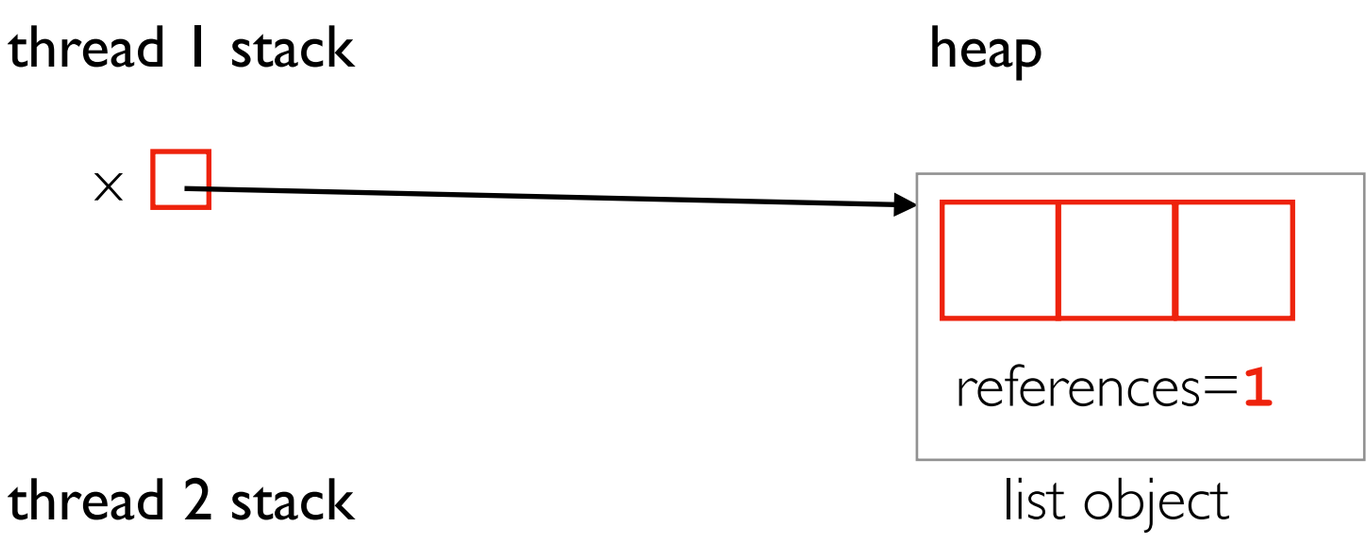
```
# thread 1
x = some list
x = None

# thread 2
y = that same list
y = None
```

situation

- cpython (main Python interpreter) uses reference counting internally to know when it can free objects
- implication: multiple threads modifying same integer

solutions

- run one thread at a time (Python's approach)
- lots of locking (slower for single-threaded code)
- *other?*



Why the GIL? Garbage collection!

What's New In Python 3.13

Editors: Adam Turner and Thomas Wouters

This article explains the new features in Python 3.13, compared to 3.12. Python 3.13 will be released on October 1, 2024. For full details, see the changelog.

See also: PEP 719 – Python 3.13 Release Schedule

Summary – Release Highlights

...

Free-threaded CPython

CPython now has experimental support for running in a free-threaded mode, with the global interpreter lock (GIL) disabled. This is an experimental feature and therefore is not enabled by default. The free-threaded mode requires a different executable, usually called `python3.13t` or `python3.13t.exe`. Pre-built binaries marked as *free-threaded* can be installed as part of the official Windows and macOS installers, or CPython can be built from source with the `--disable-gil` option.

Free-threaded execution allows for full utilization of the available processing power by running threads in parallel on available CPU cores. While not all software will benefit from this automatically, programs designed with threading in mind will run faster on multi-core hardware. **The free-threaded mode is experimental** and work is ongoing to improve it: expect some bugs and a substantial single-threaded performance hit. Free-threaded builds of CPython support optionally running with the GIL enabled at runtime using the environment variable `PYTHON_GIL` or the command-line option `-X gil=1`.

https://docs.python.org/3.13/whatsnew/3.13.html

Outline

Global Interpreter Lock

Instruction Reordering and Caching

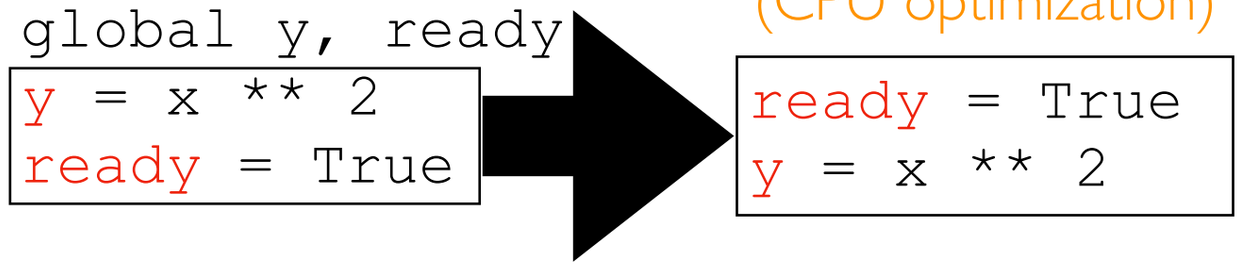
Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True

t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```

no interleaving is problematic, but it's still not correct on a modern CPU!

Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True
t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```



no interleaving is problematic, but it's still not correct on a modern CPU!

Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True

t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```

main

core 1 (running task)

L1 cache:
y = 25
ready = True

core 2 (running main)

L1 cache:
y = 0 (stale)
ready = False (stale)

no interleaving is problematic, but it's still not correct on a modern CPU!

Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True
```

core 1 (running task)

L1 cache:
y = 25
ready = True

core 2 (running main)

L1 cache:
y = 0 (stale)
ready = True

```
t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```

main

no interleaving is problematic, but it's still not correct on a modern CPU!

Concluding Advice

Use provided primitives (like locks+joins) to control isolation+ordering

- these calls control interleaving AND memory barriers (topic beyond 544)
- it's easy to get lockless approaches wrong

Correctness tips (keep it simple to avoid bugs!):

- can you use multiple processes instead of threads?
- is one big lock good enough for protecting all your data?
- is it OK to hold the lock through a whole function call?

Performance tips:

- avoid holding a lock while blocking on I/O (network, disk, user input, etc)
- if you have multiple updates, can you hold the lock for more than one of them?
- use performant packages like numpy
  - ➡ the code in C/C++/Fortran/Rust can often run without the GIL
  - ➡ these will often create threads for you

[544] Advanced Concurrency (part 1)

[544] Block Devices (part 2)

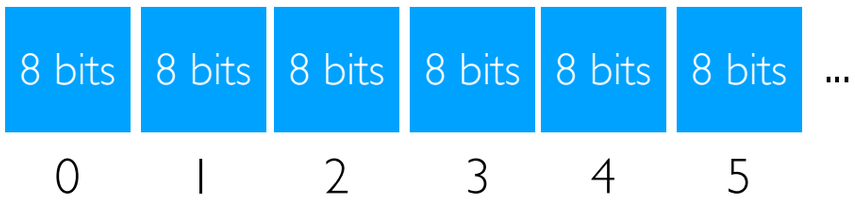
Tyler Caraza-Harter

Learning Objectives

- compare the performance characteristics of different kinds of block devices (HDDs and SSDs)
- describe different kinds of file systems
- interpret the output of tools like "mount" and "df" to understand the structure of a mount namespace

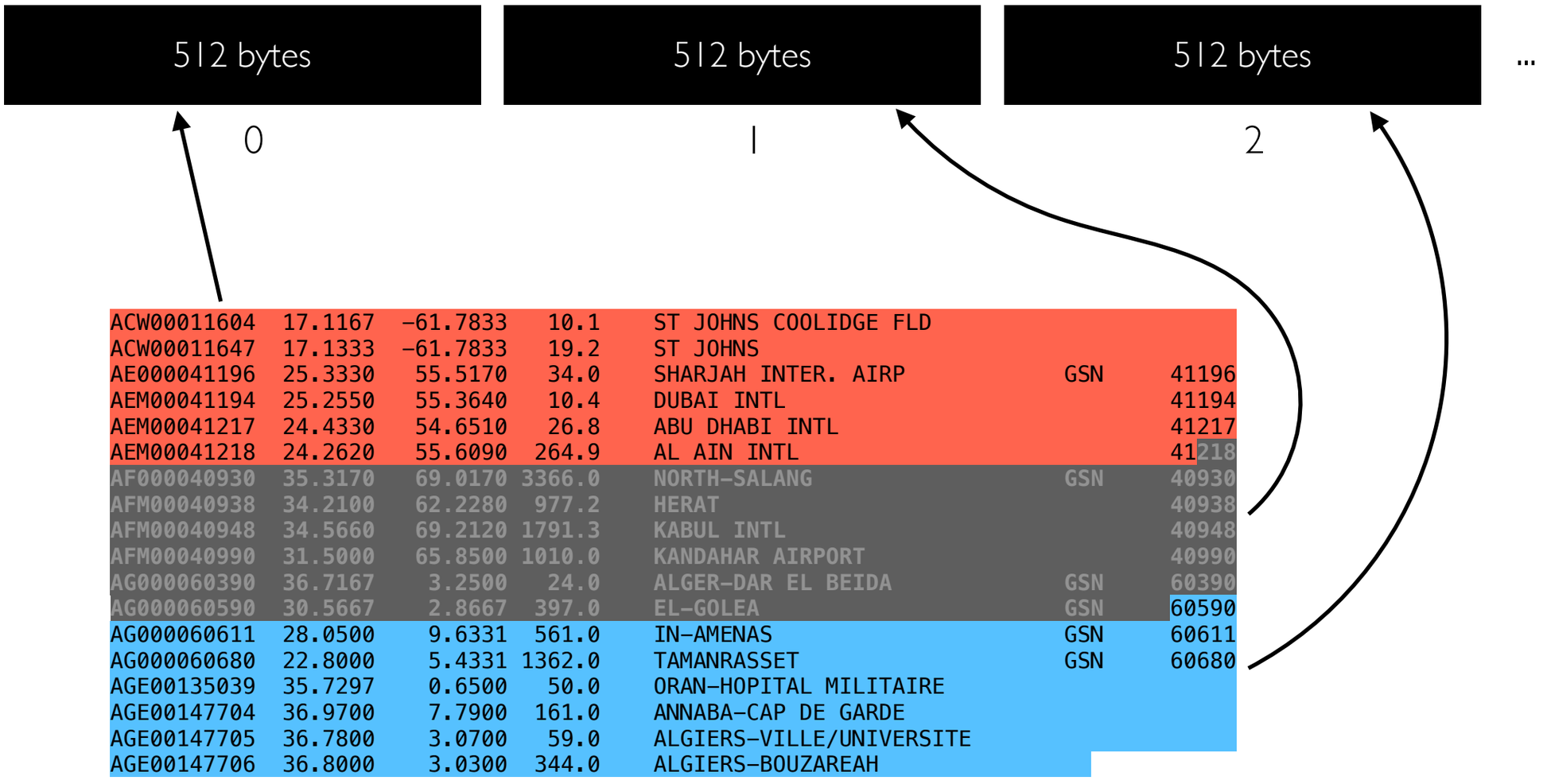
Block Devices

Memory is byte addressable



Block storage devices are accessed in units of blocks (512 bytes, few KBs, etc)

Optimizing Disk I/O with Memory: Caching and Buffering




```
ACW00011604  17.1167  -61.7833   10.1    ST JOHNS COOLIDGE FLD
ACW00011647  17.1333  -61.7833   19.2    ST JOHNS
AE000041196  25.3330   55.5170   34.0    SHARJAH INTER. AIRP           GSN     41196
AEM00041194  25.2550   55.3640   10.4    DUBAI INTL                            41194
AEM00041217  24.4330   54.6510   26.8    ABU DHABI INTL                        41217
AEM00041218  24.2620   55.6090  264.9    AL AIN INTL                           41218
AF000040930  35.3170   69.0170 3366.0    NORTH-SALANG                  GSN     40930
AFM00040938  34.2100   62.2280  977.2    HERAT                                 40938
AFM00040948  34.5660   69.2120 1791.3    KABUL INTL                            40948
AFM00040990  31.5000   65.8500 1010.0    KANDAHAR AIRPORT                      40990
AG000060390  36.7167    3.2500   24.0    ALGER-DAR EL BEIDA            GSN     60390
AG000060590  30.5667    2.8667  397.0    EL-GOLEA                      GSN     60590
AG000060611  28.0500    9.6331  561.0    IN-AMENAS                     GSN     60611
AG000060680  22.8000    5.4331 1362.0    TAMANRASSET                   GSN     60680
AGE00135039  35.7297    0.6500   50.0    ORAN-HOPITAL MILITAIRE
AGE00147704  36.9700    7.7900  161.0    ANNABA-CAP DE GARDE
AGE00147705  36.7800    3.0700   59.0    ALGIERS-VILLE/UNIVERSITE
AGE00147706  36.8000    3.0300  344.0    ALGIERS-BOUZAREAH
```

ghcnd-stations.txt

We might want to process one line a time, but it would be wasteful to repeatedly read the same block from the device

- the Linux page cache stores pages from files in RAM (usually 4KB pages, often larger than device blocks)
- Python (and other) programs might buffer chunks of data to avoid asking Linux too many times for small pieces of data

Optimizing Disk I/O with Memory: Caching and Buffering

htop

page cache, could evict if needed

only counts non-cache memory

We might want to process one line a time, but it would be wasteful to repeatedly read the same block from the device

- the Linux page cache stores pages from files in RAM (usually 4KB pages, often larger than device blocks)
- Python (and other) programs might buffer chunks of data to avoid asking Linux too many times for small pieces of data

Small Reads (<512 bytes): Performance

goal: collect all station IDs

```
ACW00011604  17.1167  -61.7833    10.1    ST JOHNS COOLIDGE FLD
ACW00011647  17.1333  -61.7833    19.2    ST JOHNS
AE000041196  25.3330   55.5170    34.0    SHARJAH INTER. AIRP           GSN     41196
AEM00041194  25.2550   55.3640    10.4    DUBAI INTL                            41194
AEM00041217  24.4330   54.6510    26.8    ABU DHABI INTL                        41217
AEM00041218  24.2620   55.6090   264.9    AL AIN INTL                           41218
AF000040930  35.3170   69.0170 3366.0    NORTH-SALANG                  GSN     40930
AFM00040938  34.2100   62.2280   977.2    HERAT                                 40938
AFM00040948  34.5660   69.2120  1791.3    KABUL INTL                            40948
AFM00040990  31.5000   65.8500  1010.0    KANDAHAR AIRPORT                      40990
AG000060390  36.7167    3.2500    24.0    ALGER-DAR EL BEIDA            GSN     60390
AG000060590  30.5667    2.8667   397.0    EL-GOLEA                      GSN     60590
AG000060611  28.0500    9.6331   561.0    IN-AMENAS                     GSN     60611
AG000060680  22.8000    5.4331  1362.0    TAMANRASSET                   GSN     60680
AGE00135039  35.7297    0.6500    50.0    ORAN-HOPITAL MILITAIRE
AGE00147704  36.9700    7.7900   161.0    ANNABA-CAP DE GARDE
AGE00147705  36.7800    3.0700    59.0    ALGIERS-VILLE/UNIVERSITE
AGE00147706  36.8000    3.0300   344.0    ALGIERS-BOUZAREAH
```

ghcnd-stations.txt

```
start = time.time()
with open("ghcnd-stations.txt") as f:
    for line in f:
        stations.append(line[:11])
print(time.time() - start)
```

simple version that reads everything: **66 ms**

format issue: no good way to read one column without everything else

(similar to issues with bad cache line usage)

```
stations = []
line_len = 86

start = time.time()
with open("ghcnd-stations.txt",
          "rb", buffering=0) as f:
    offset = 0
    while True:
        f.seek(offset)
        station = str(f.read(11), "utf-8")
        offset += line_len

        if station:
            stations.append(station)
        else:
            break
print(time.time() - start)
```

"optimized" version that only reads stations: **171 ms**

Hard Disk Drives (HDDs)

Steps to read/write

1. move head to correct track
2. wait for spinning disk to rotate until data is under head
3. transfer the data

these steps dominate unless transferring lots of data (few MBs)

Layout

- assign block numbers to platter locations so sequential (like 5,6,7,8, ...) reads/writes will be fast
- programmers should assume random accesses (like 2, 9, 5, 1, ...) will be slow

Capacity vs. I/O and Short Stroking

Storage resources

1. capacity
2. I/O (input/output often more limited when using HDDs)

Short Stroking

- head moves over platter faster near outside track
- smaller block addrs correspond to outside tracks
- strategy: only use outside tracks
- pros: faster I/O
- cons: less space

Solid State Drives (SSDs) - Flash

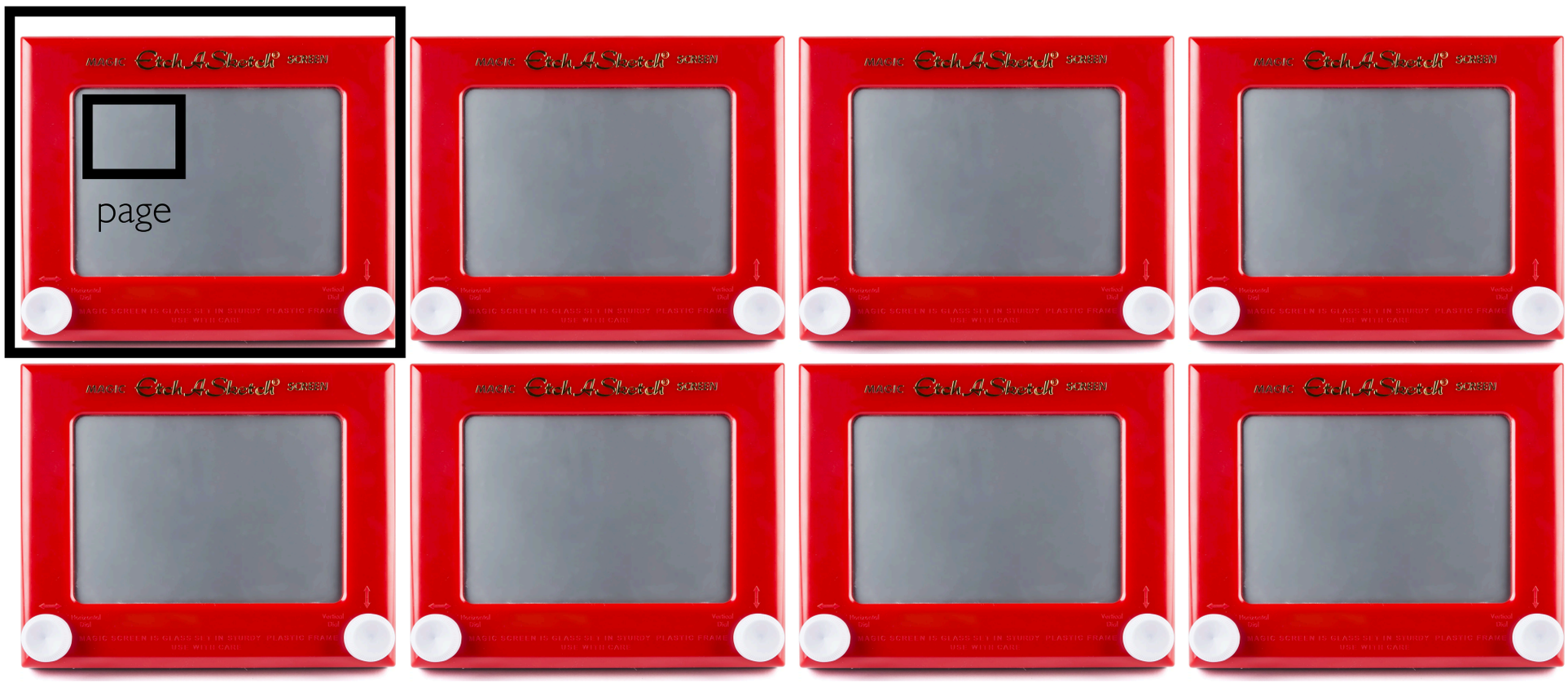
Reading and writing

- no moving parts
- inherently parallel

SSD internals:

- "block" and "page" have different meanings in this context
- "page" => unit that we can read or write (couple KBs)
- pages cannot be individually re-written
- "block" => unit that is erased together (maybe 100s of KBs)

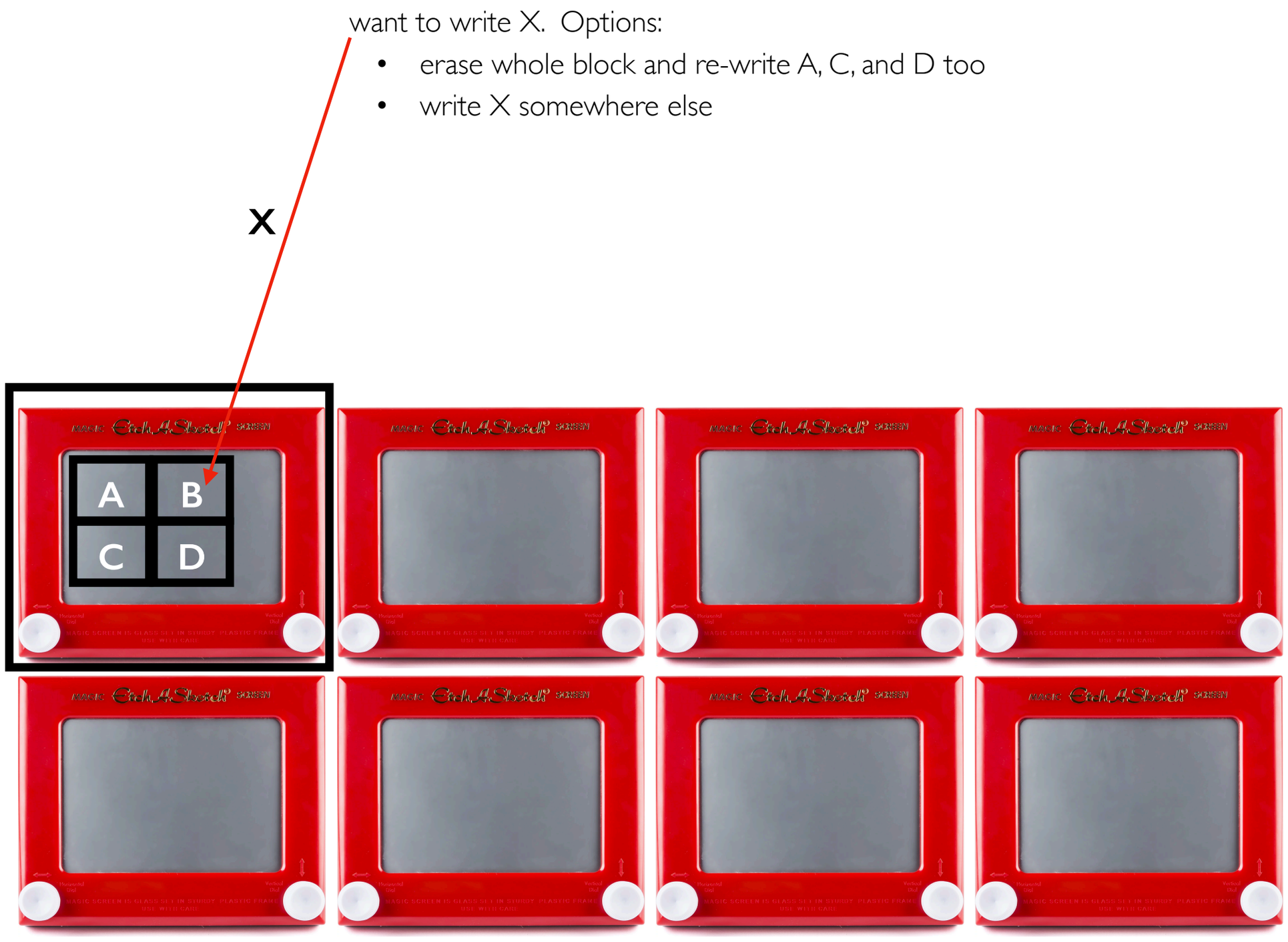
block



Solid State Drives (SSDs) - Flash

want to write X. Options:

- erase whole block and re-write A, C, and D too
- write X somewhere else



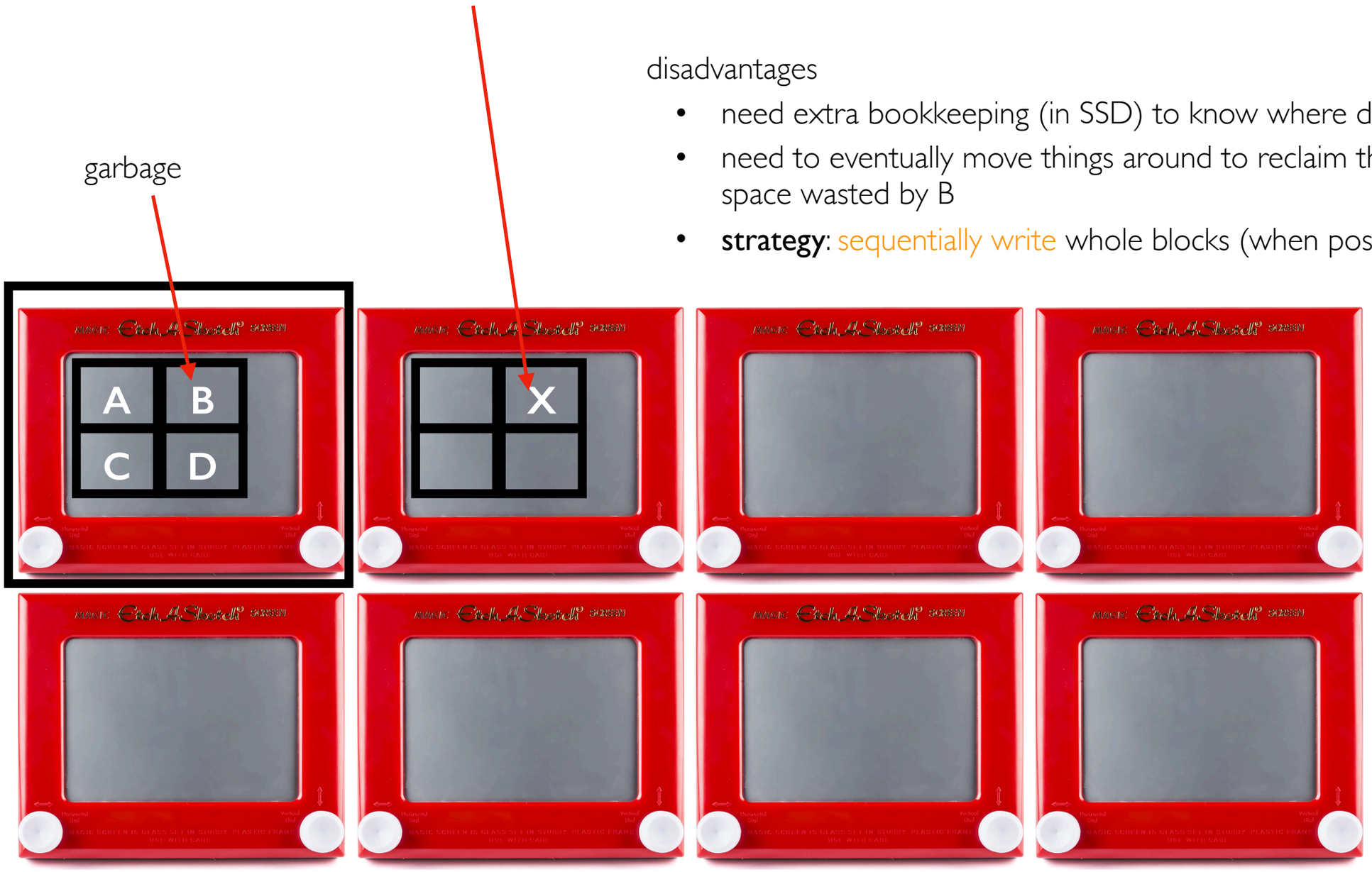
Solid State Drives (SSDs) - Flash

want to write X. Options:

- erase whole block and re-write A, C, and D too
- write X somewhere else

disadvantages

- need extra bookkeeping (in SSD) to know where data is
- need to eventually move things around to reclaim the space wasted by B
- **strategy**: sequentially write whole blocks (when possible)



HDDs vs. SSDs

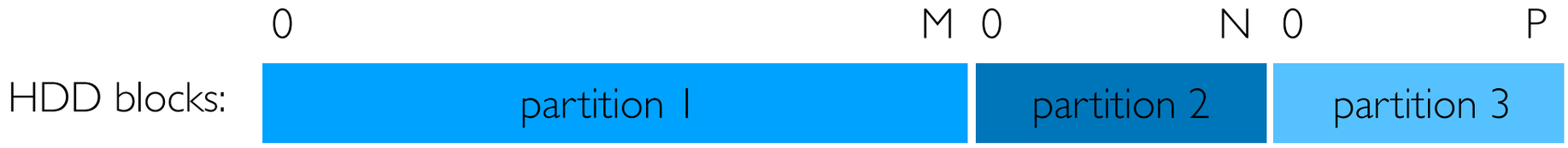
Metrics

- capacity: how many bytes can we store?
- latency: how long does it take to start transferring data
- IOPS (I/O operations, of some max size, per second): how many small/random transfers can we do per second
- throughput: how many bytes can we transfer per second

Metric:	Relative to HDDS, SSDS are:
capacity	*worse*
latency	*much better (no moving parts)*
random IOPS	*even better -- low latency AND in parallel*
throughput (sequential)	*little better*
throughput (random writes)	*better (but block erase is a concern)*
throughput (random reads)	*much better*

Partitions and RAID

Block devices can be divided into **partitions**:

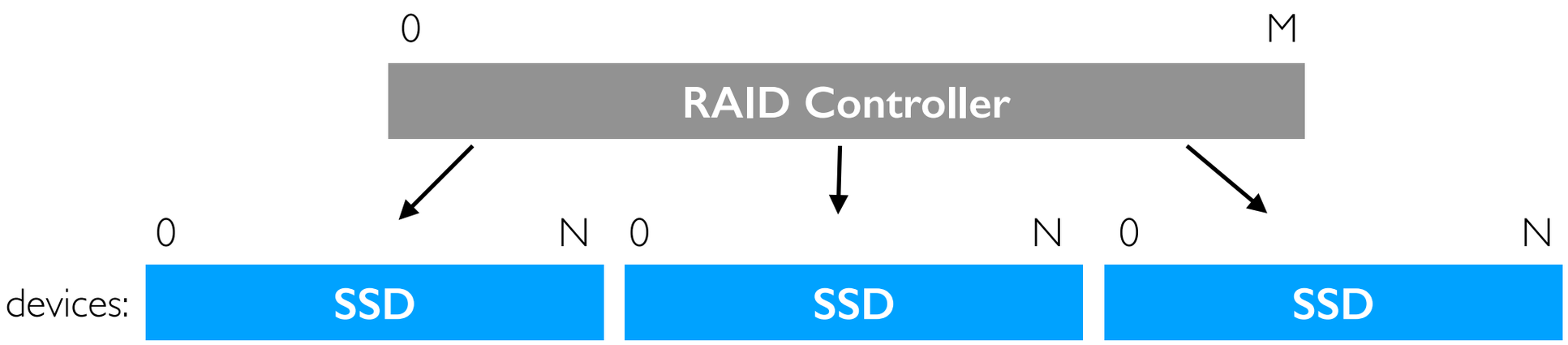



```
root@instance-1:/home/trh# ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda14  /dev/sda15
/dev/sdb  /dev/sdb1
```

2 devices

4 partitions

RAID controllers (Redundant Array of Inexpensive Disks) can make multiple devices appear as one:



Many configs use **redundancy** (e.g., same data on >1 disk) to avoid data loss when one device dies.