Profiling tools

Operating Systems

PROF. JORGE GONZALEZ REAÑO <JGONZALEZ@UTEC.EDU.PE>

Outline

Performance and I/O

- Perf tool
- IO stat tool

Linux perf tools

- Install perf:

```
$ sudo apt-get install linux-tools-common linux-
tools-4.2.0-27-generic linux-cloud-tools-4.2.0-27-generic
```

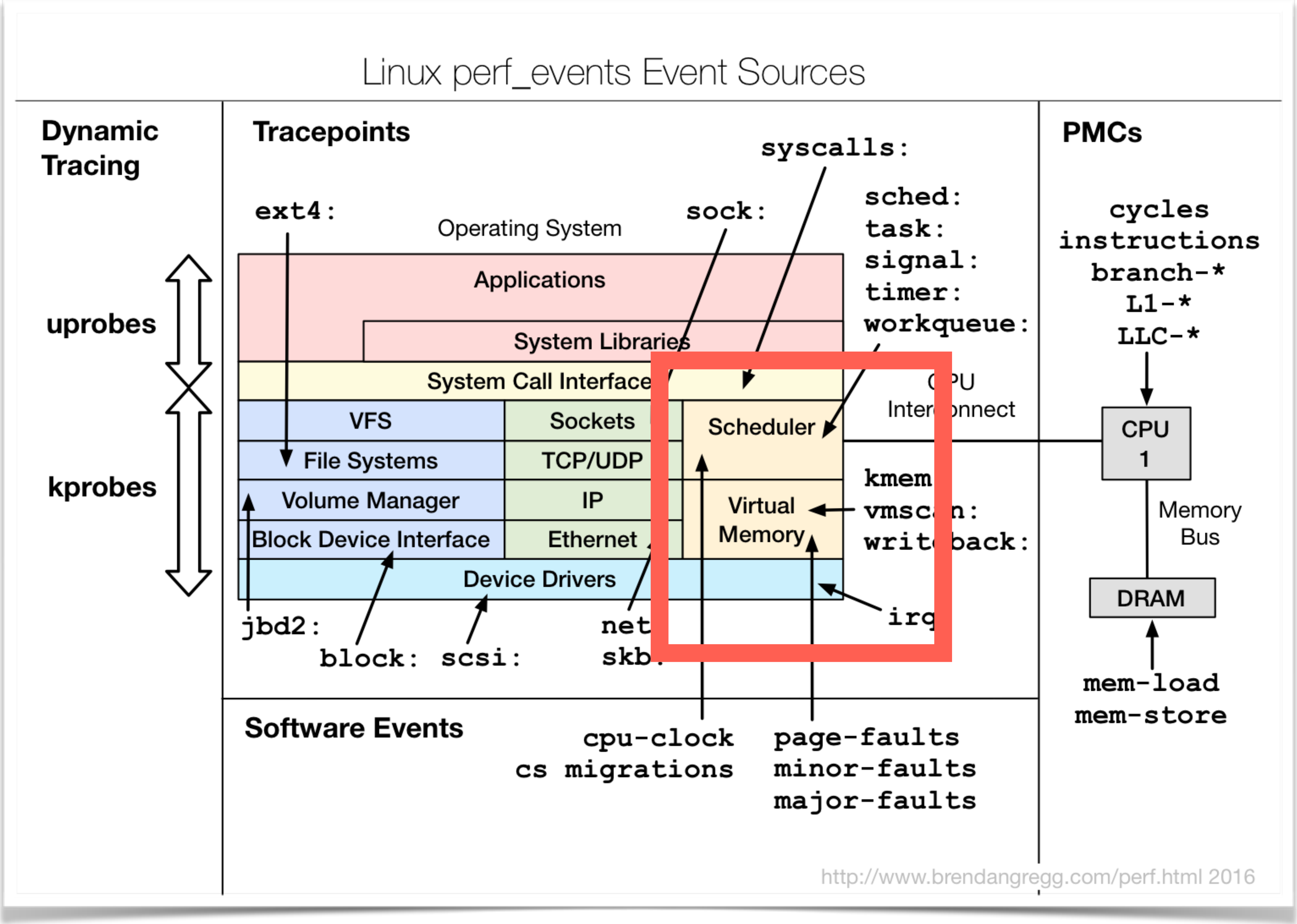
- Analize cache and TLB:

```
$ perf stat -e cache-misses <command>
$ perf stat -e dTLB-load-misses,iTLB-load-misses <command>
```

- Examples:

```
$ perf stat -e cache-misses ls>/dev/null
$ perf stat -e dTLB-load-misses,iTLB-load-misses ls>/dev/
null
```

Perf and virtual memory



Example: Perf counters

HTTP://WWW.BRENDANGREGG.COM/PERF.HTML

```
# CPU counter statistics for the specified command:
perf stat command

# Detailed CPU counter statistics (includes extras) for the specified command:
perf stat -d command

# CPU counter statistics for the specified PID, until Ctrl-C:
perf stat -p PID

# CPU counter statistics for the entire system, for 5 seconds:
perf stat -a sleep 5

# Various basic CPU statistics, system wide, for 10 seconds:
perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles -a sleep 10

# Various CPU level 1 data cache statistics for the specified command:
perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores command

# Various CPU data TLB statistics for the specified command:
perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses command

# Various CPU last level cache statistics for the specified command:
perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches command

# Using raw PMC counters, eg, counting unhalted core cycles:
perf stat -e r003c -a sleep 5

# Count all vmscan events, printing a report every second:
perf stat -e 'vmscan:*' -a -I 1000
```

Example: Perf counters

```
# PMCs: counting cycles and frontend stalls via raw specification:
perf stat -e cycles -e cpu/event=0x0e,umask=0x01,inv,cmask=0x01/ -a sleep 5

# Count syscalls per-second system-wide:
perf stat -e raw_syscalls:sys_enter -I 1000 -a

# Count system calls by type for the specified PID, until Ctrl-C:
perf stat -e 'syscalls:sys_enter_*' -p PID

# Count system calls by type for the entire system, for 5 seconds:
perf stat -e 'syscalls:sys_enter_*' -a sleep 5

# Count scheduler events for the specified PID, until Ctrl-C:
perf stat -e 'sched:*' -p PID

# Count scheduler events for the specified PID, for 10 seconds:
perf stat -e 'sched:*' -p PID sleep 10

# Count ext4 events for the entire system, for 10 seconds:
perf stat -e 'ext4:*' -a sleep 10

# Count block device I/O events for the entire system, for 10 seconds:
perf stat -e 'block:*' -a sleep 10
```

Recall: Memory Hierarchy

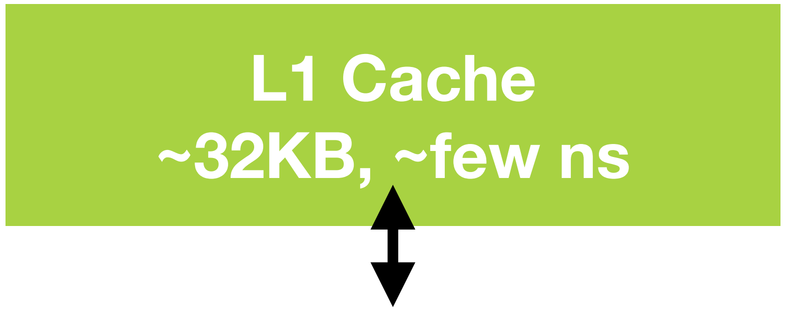
…

Recall: Memory Hierarchy

L1 Cache
~32KB, ~few ns

…

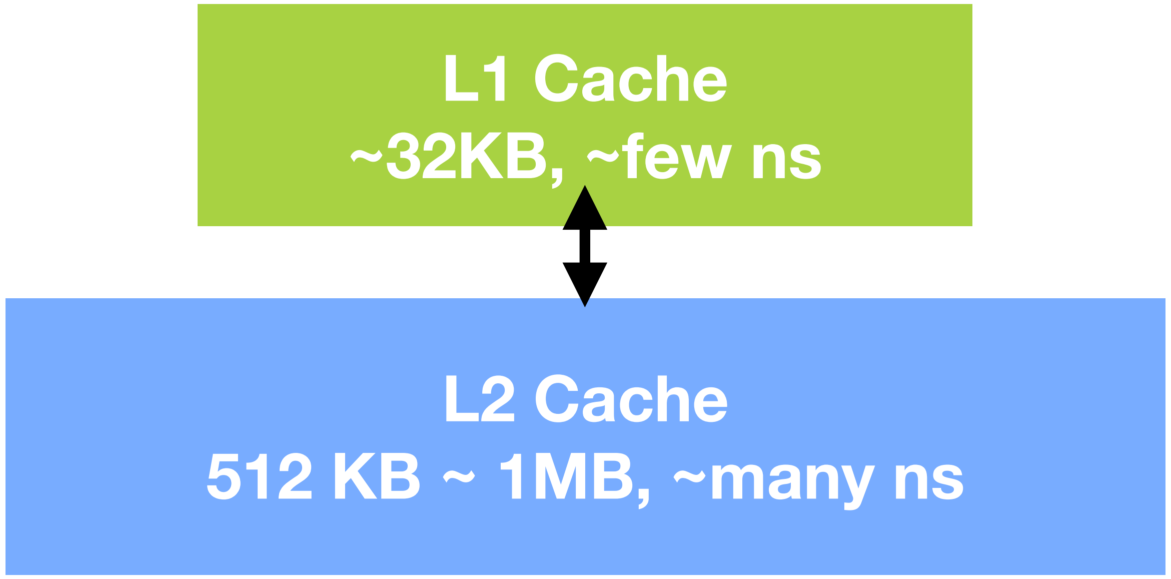
Recall: Memory Hierarchy



L1 Cache
~32KB, ~few ns

…

Recall: Memory Hierarchy

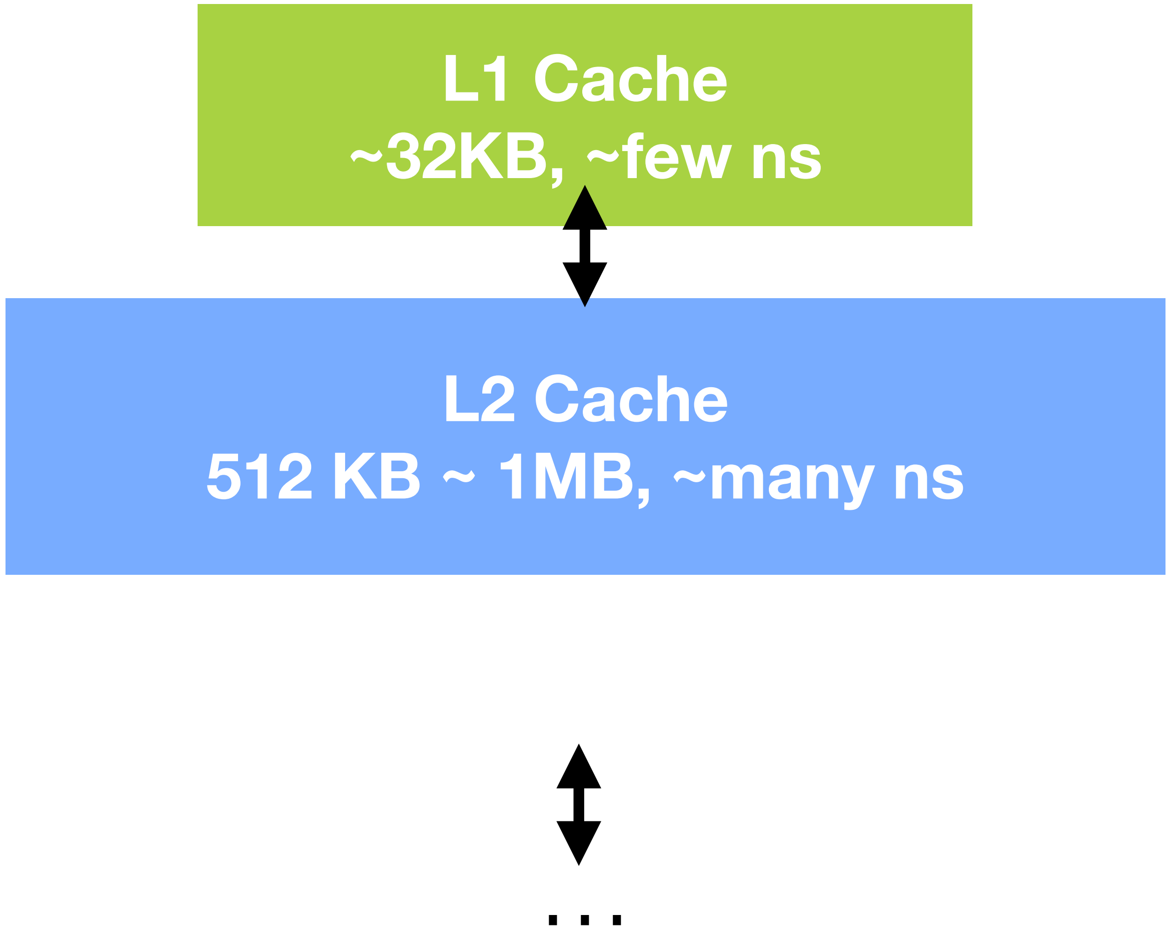


L1 Cache
~32KB, ~few ns

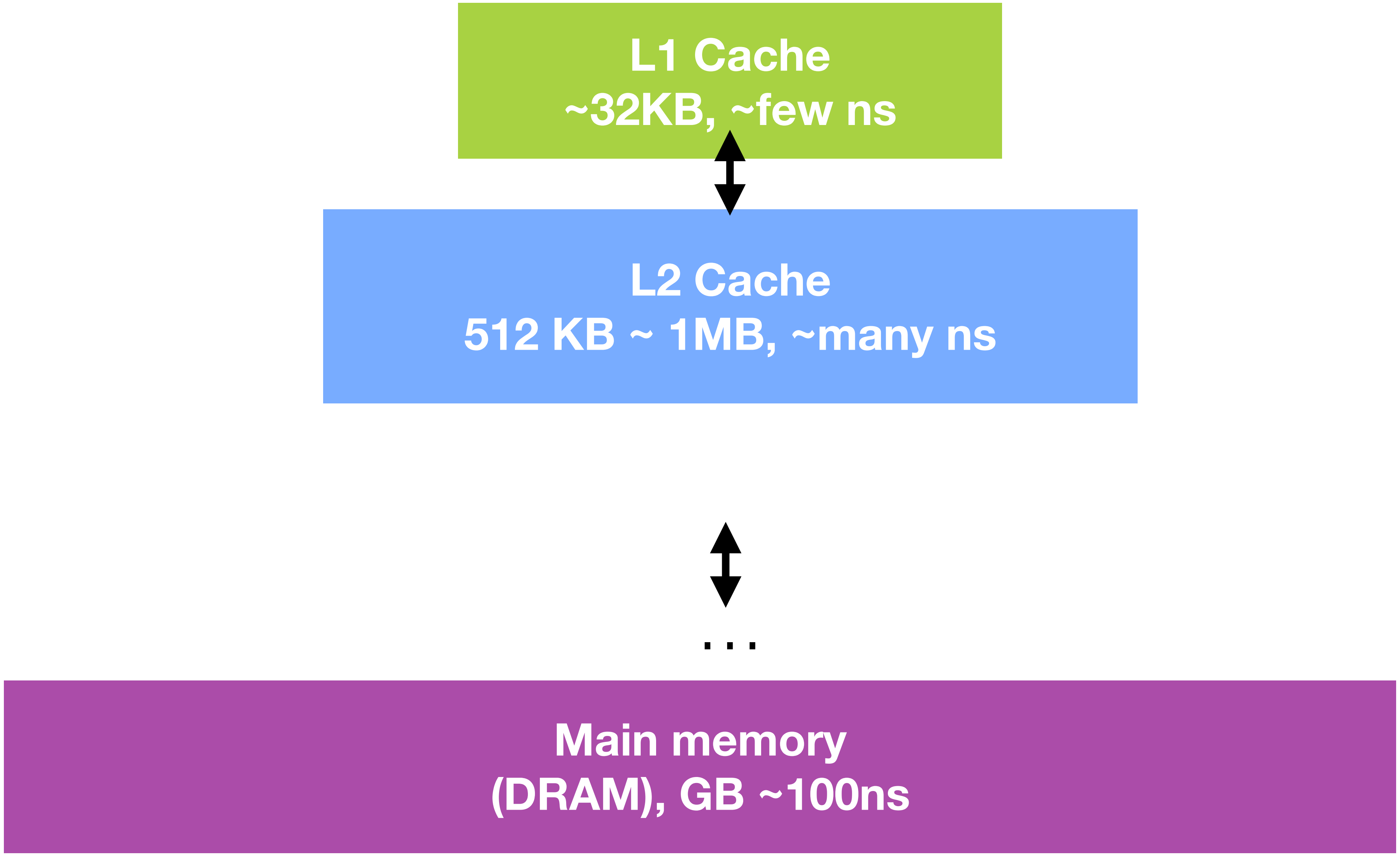
L2 Cache
512 KB ~ 1MB, ~many ns

…

Recall: Memory Hierarchy



Recall: Memory Hierarchy



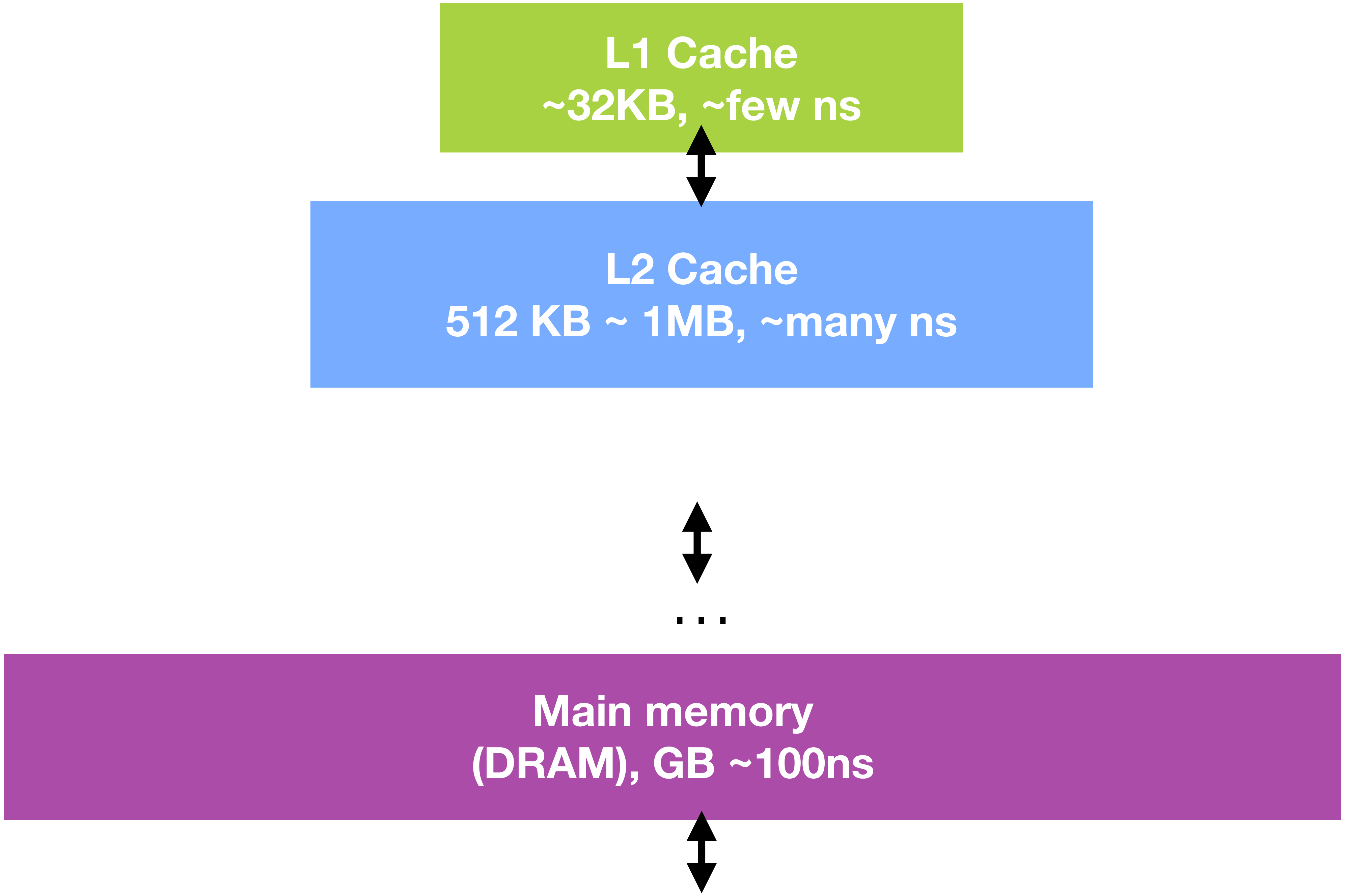
L1 Cache
~32KB, ~few ns

L2 Cache
512 KB ~ 1MB, ~many ns

…

Main memory
(DRAM), GB ~100ns

Recall: Memory Hierarchy



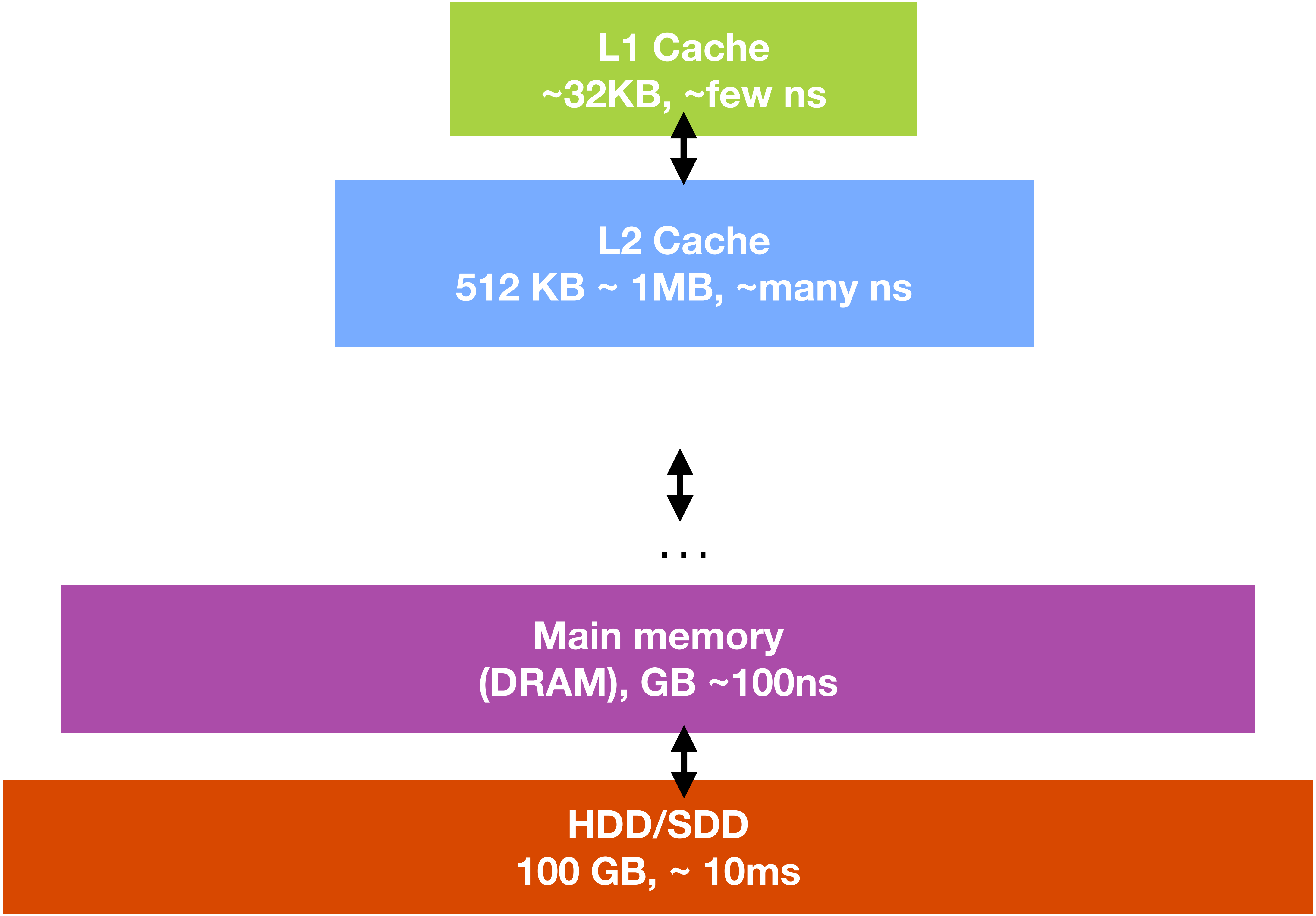
L1 Cache
~32KB, ~few ns

L2 Cache
512 KB ~ 1MB, ~many ns

. . .

Main memory
(DRAM), GB ~100ns

Recall: Memory Hierarchy



L1 Cache
~32KB, ~few ns

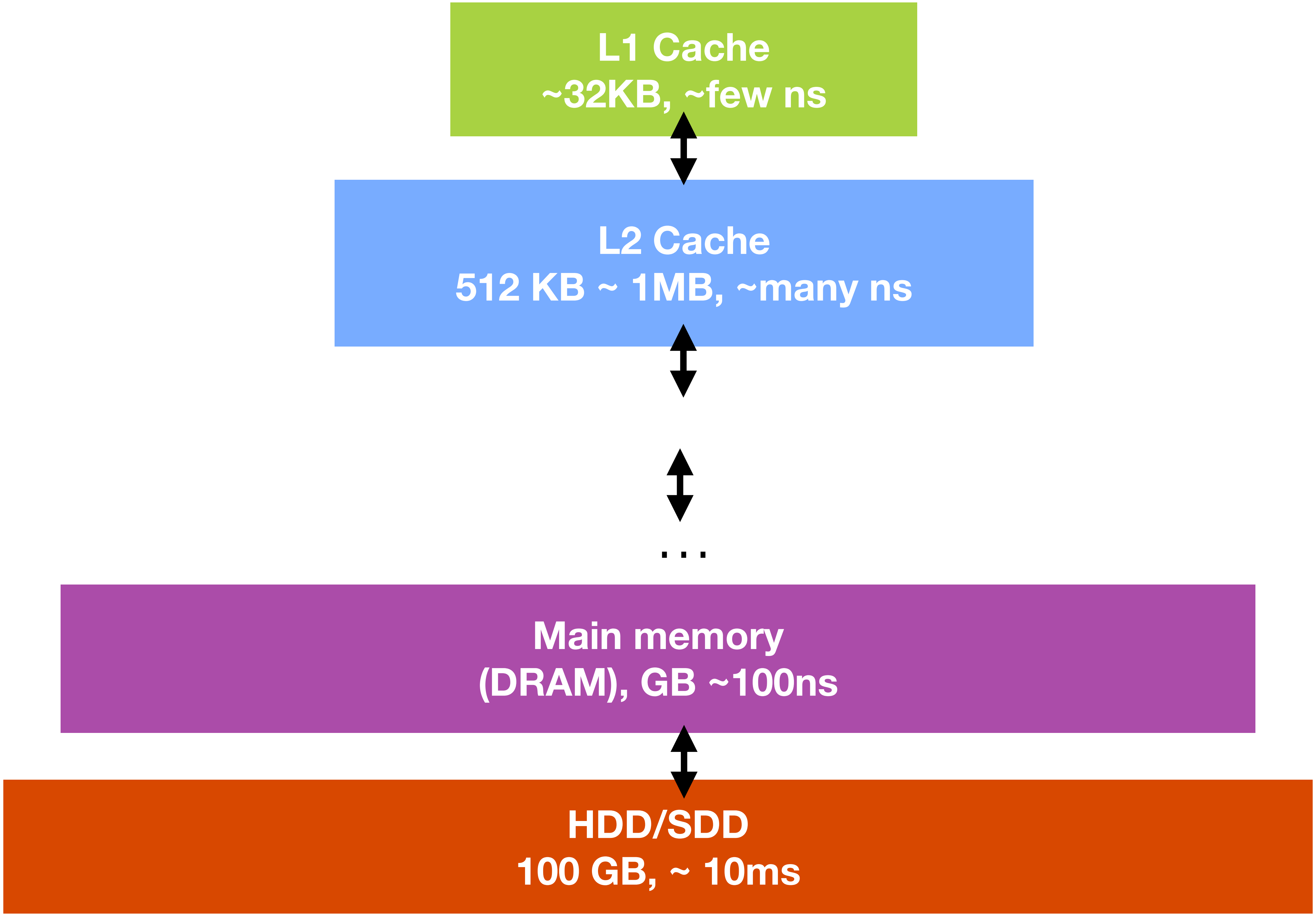
L2 Cache
512 KB ~ 1MB, ~many ns

. . .

Main memory
(DRAM), GB ~100ns

HDD/SDD
100 GB, ~ 10ms

Recall: Memory Hierarchy



L1 Cache
~32KB, ~few ns

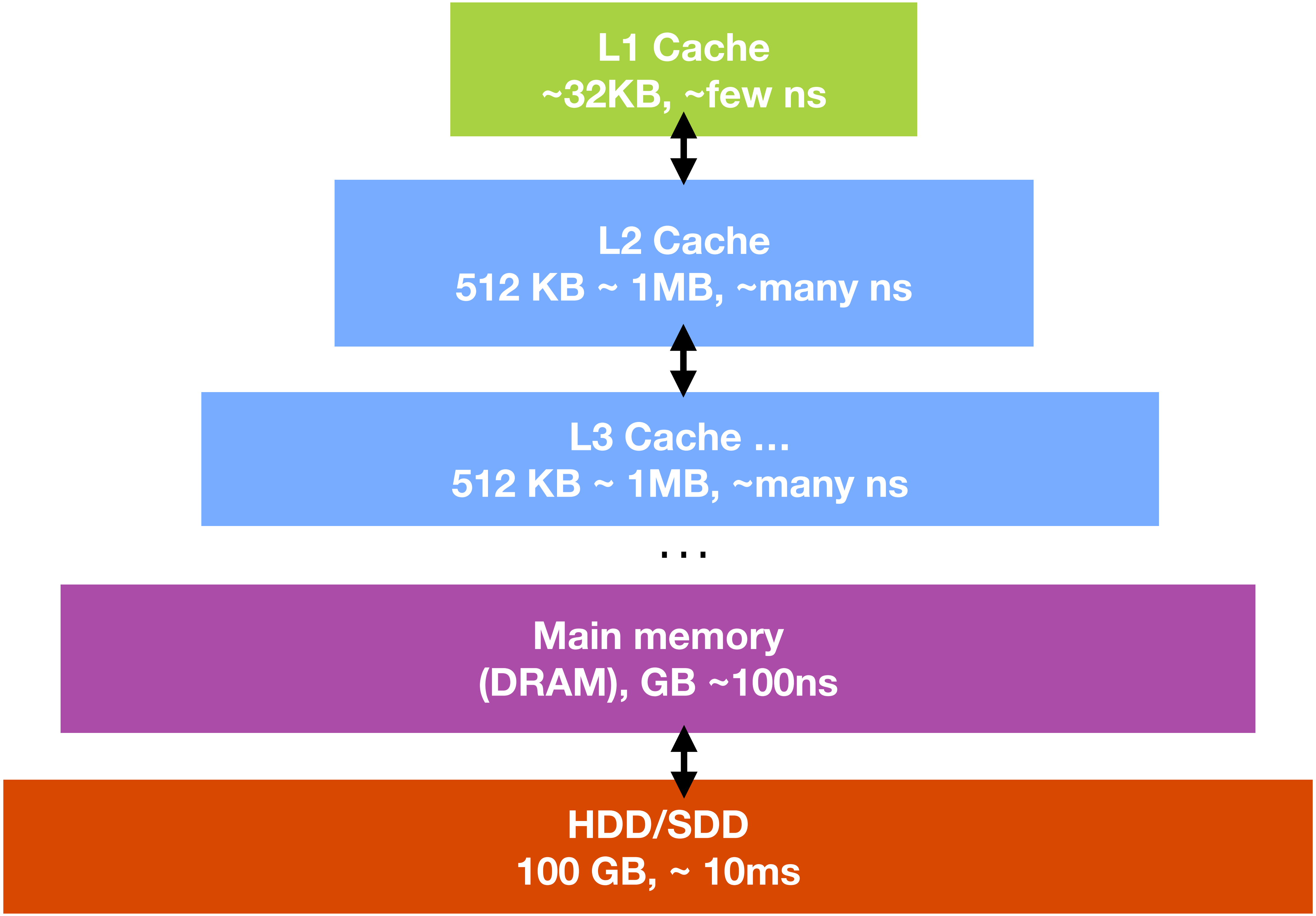
L2 Cache
512 KB ~ 1MB, ~many ns

. . .

Main memory
(DRAM), GB ~100ns

HDD/SDD
100 GB, ~ 10ms

Recall: Memory Hierarchy



L1 Cache
~32KB, ~few ns

L2 Cache
512 KB ~ 1MB, ~many ns

L3 Cache …
512 KB ~ 1MB, ~many ns

. . .

Main memory
(DRAM), GB ~100ns

HDD/SDD
100 GB, ~ 10ms

Recall: Cache

- **Definition:** Structure to **"store"** the recently and/or frequently used data and results to avoid high-latency operations on other structures to generate new data and results.
- This concept extends to OS: **Demand Paging.**
- **Cache basics:**
  - **Block (line):** Cache storage unit.
  - **Hit:** if in cache, the cached data is used instead of accessing the next level.
  - **Miss:** if not in cache, access the next level and stored the data inside (new cached data).

Recall: Memory Hierarchy

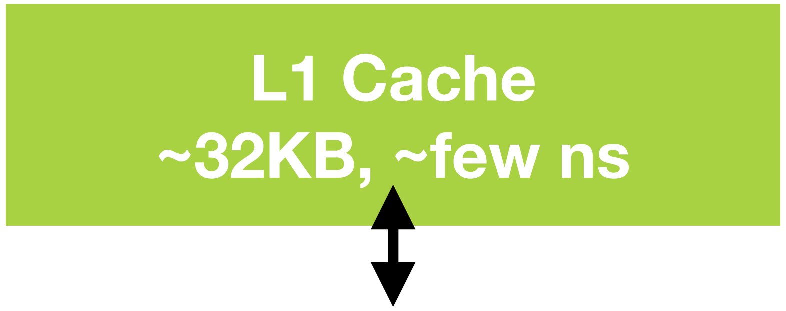
...

Recall: Memory Hierarchy

L1 Cache
~32KB, ~few ns

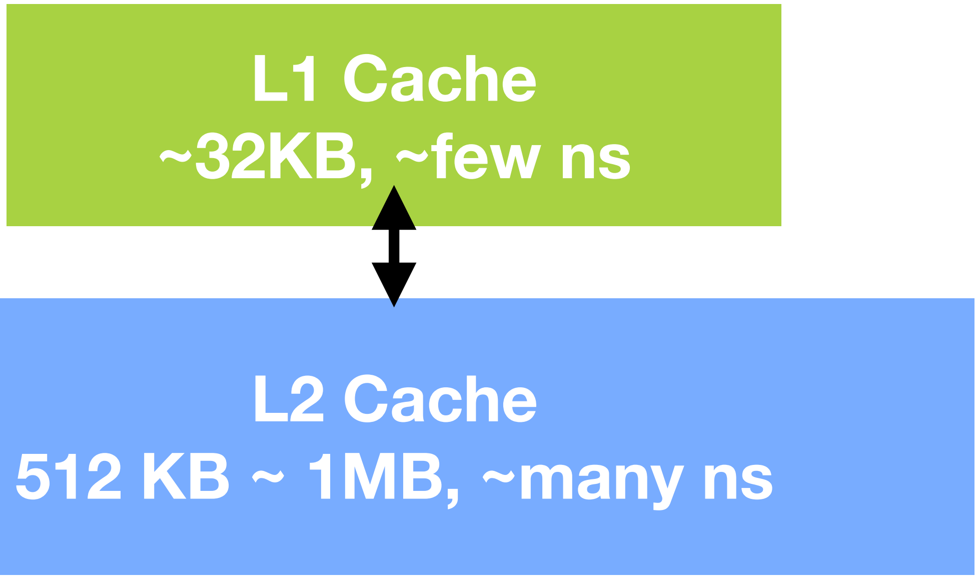
…

Recall: Memory Hierarchy



. . .

Recall: Memory Hierarchy

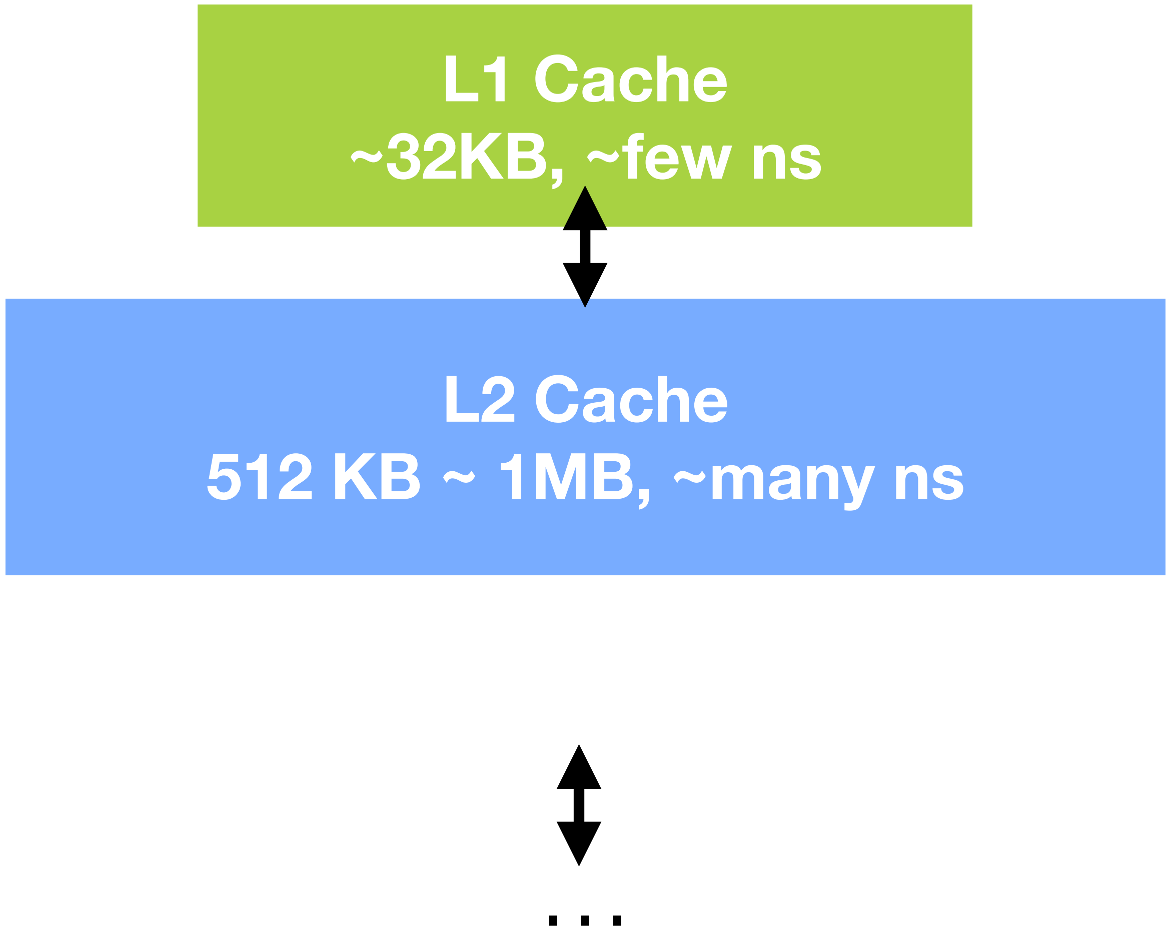


L1 Cache
~32KB, ~few ns

L2 Cache
512 KB ~ 1MB, ~many ns

…

Recall: Memory Hierarchy

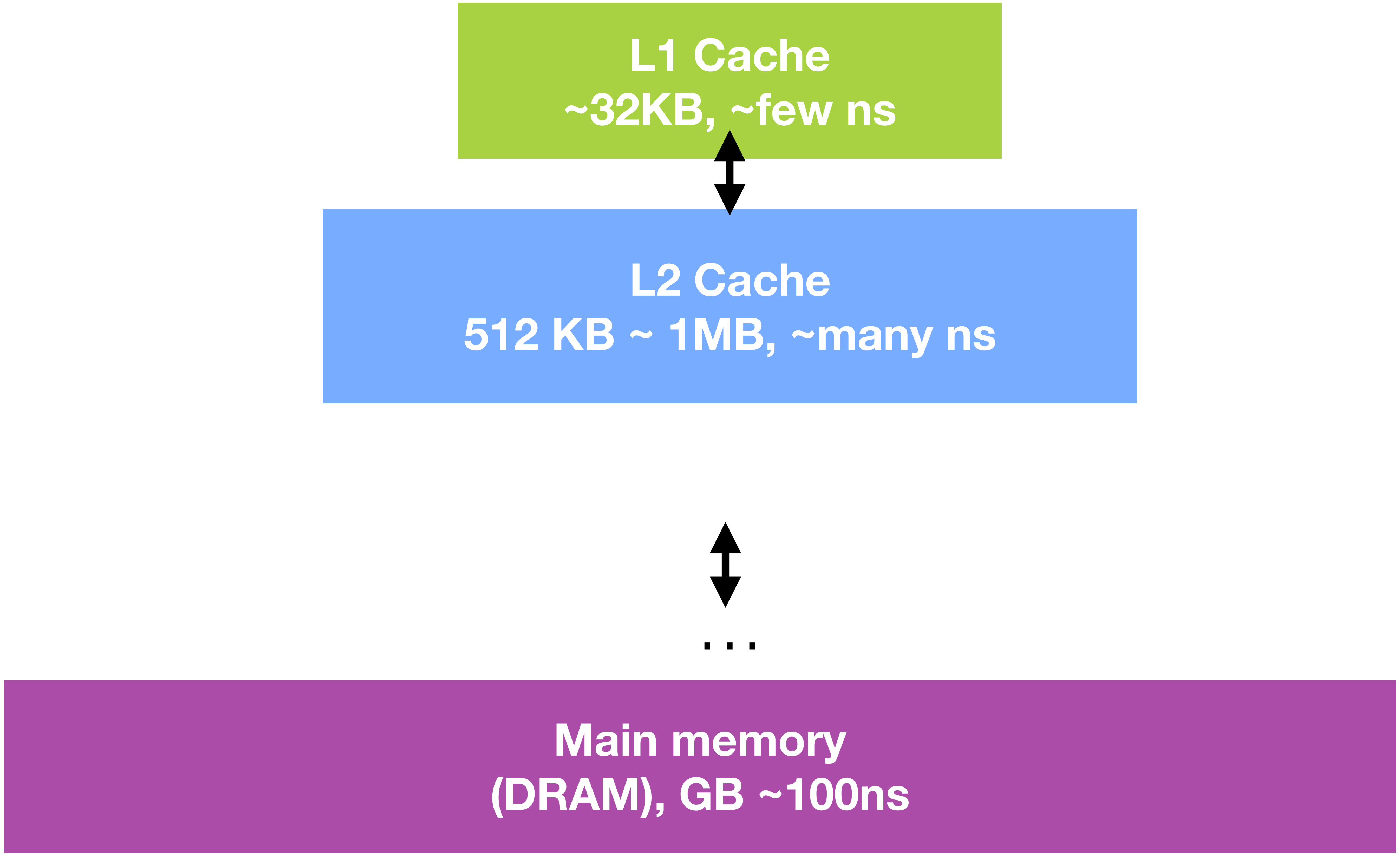


L1 Cache
~32KB, ~few ns

L2 Cache
512 KB ~ 1MB, ~many ns

…

Recall: Memory Hierarchy



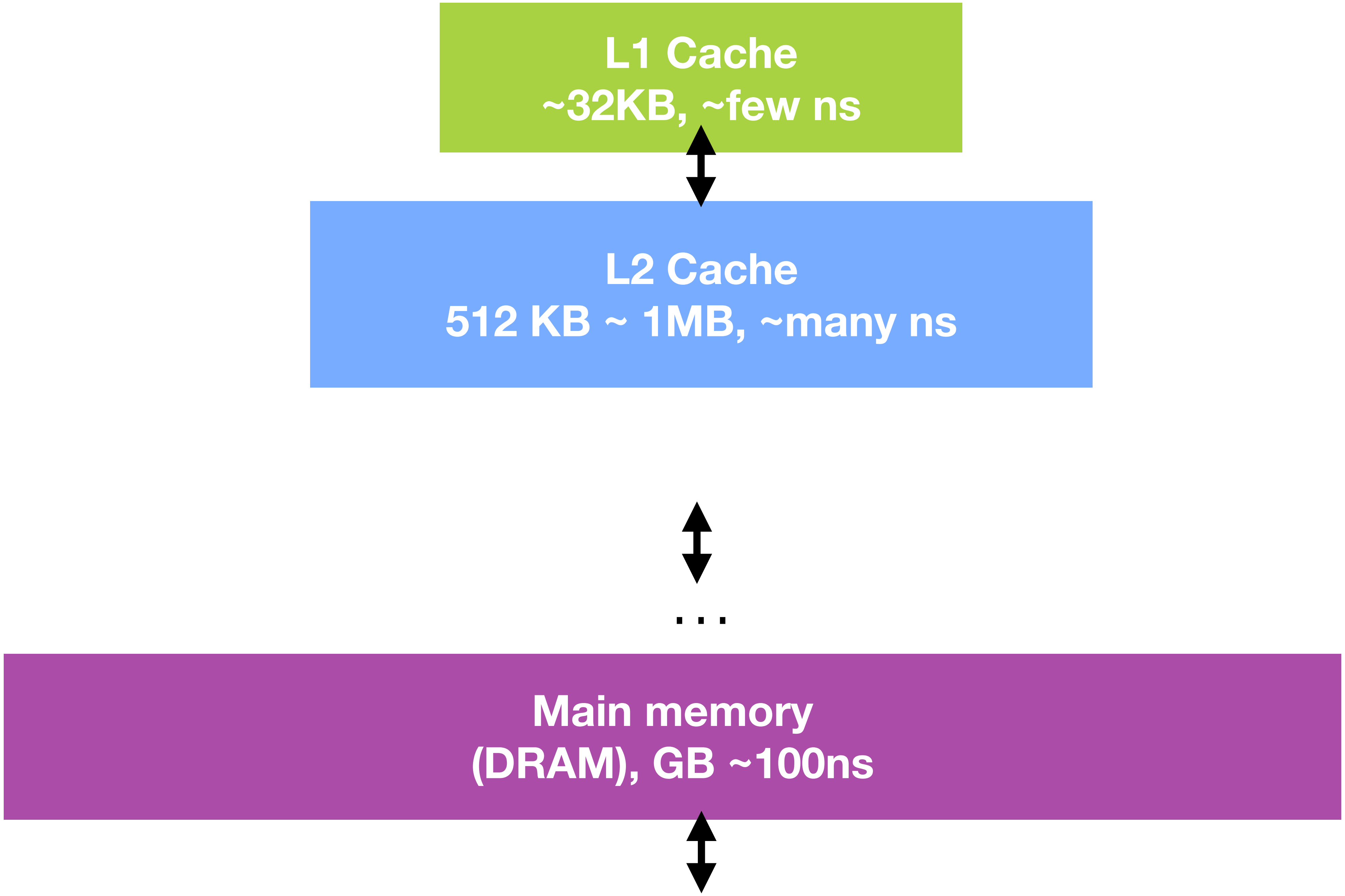
L1 Cache
~32KB, ~few ns

L2 Cache
512 KB ~ 1MB, ~many ns

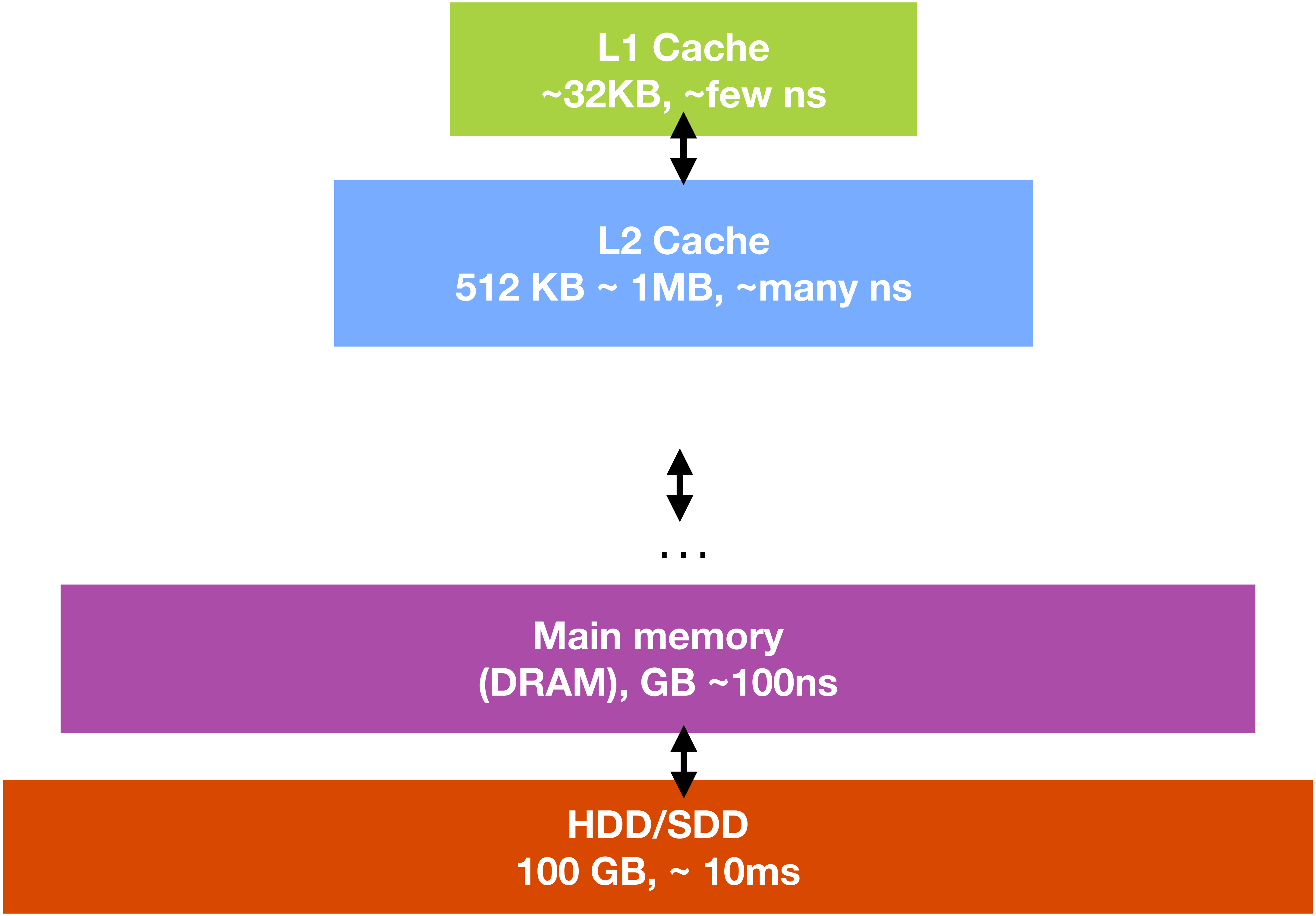
…

Main memory
(DRAM), GB ~100ns

Recall: Memory Hierarchy



Recall: Memory Hierarchy



L1 Cache
~32KB, ~few ns

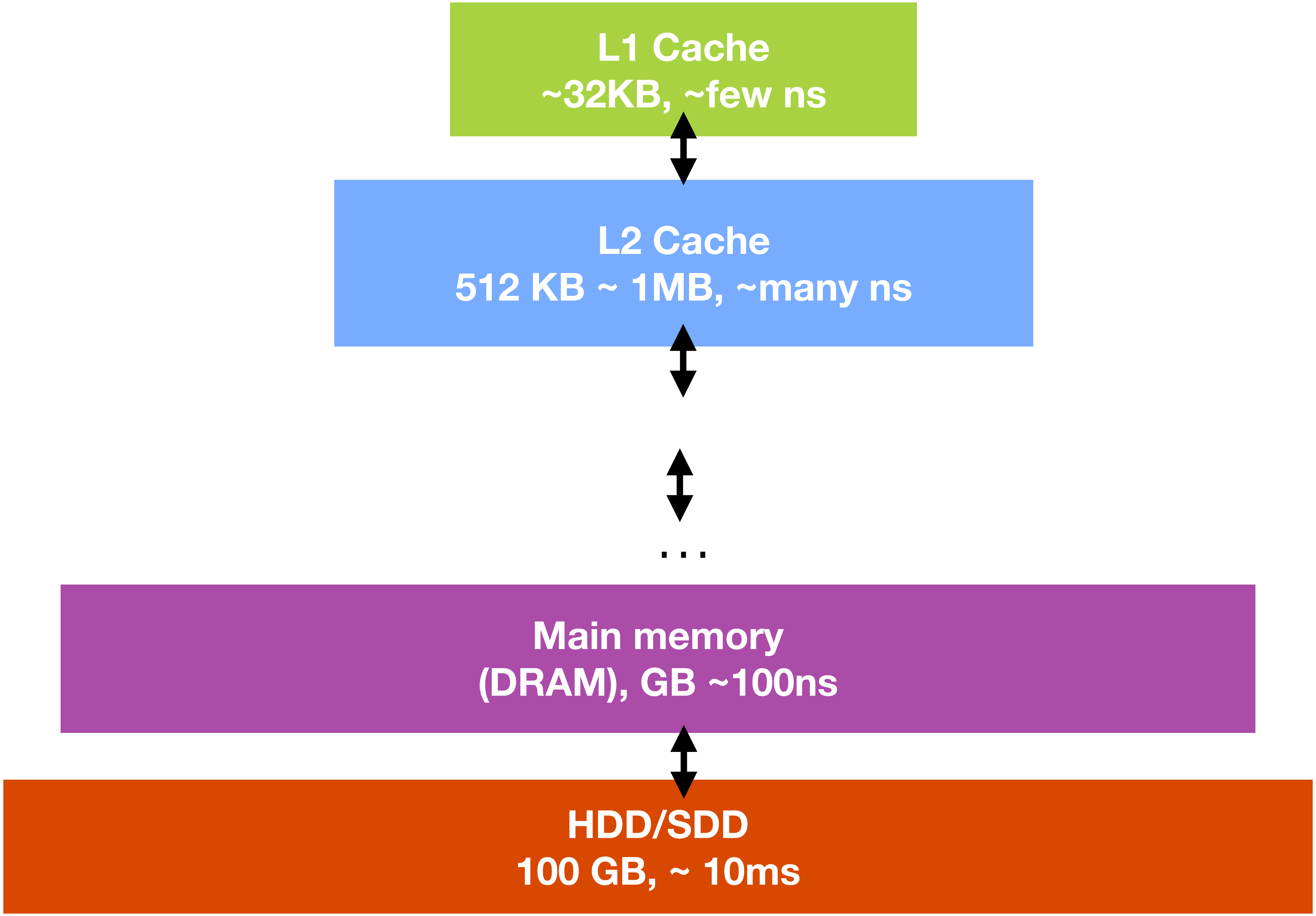
L2 Cache
512 KB ~ 1MB, ~many ns

…

Main memory
(DRAM), GB ~100ns

HDD/SDD
100 GB, ~ 10ms

Recall: Memory Hierarchy



L1 Cache
~32KB, ~few ns

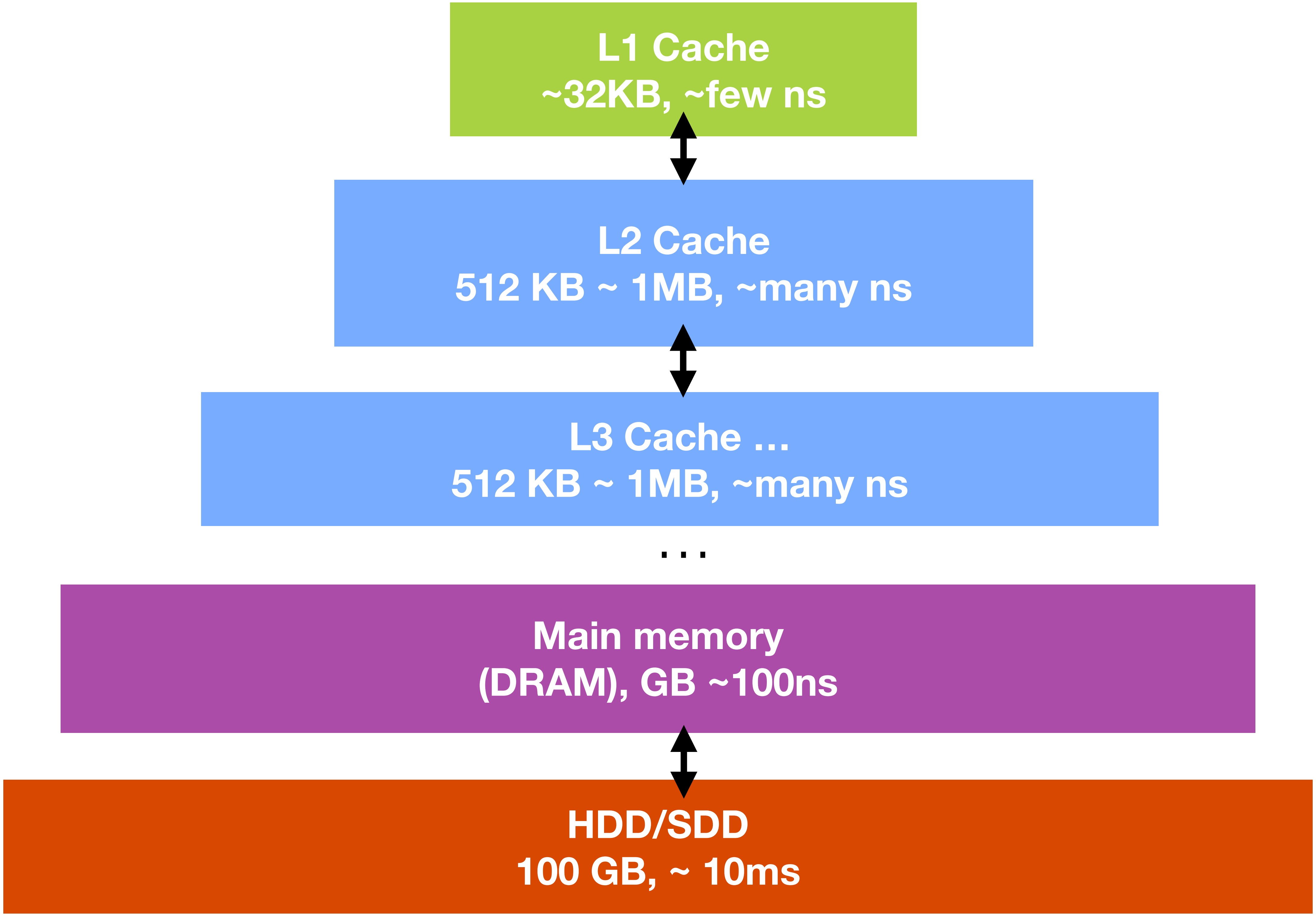
L2 Cache
512 KB ~ 1MB, ~many ns

…

Main memory
(DRAM), GB ~100ns

HDD/SDD
100 GB, ~ 10ms

Recall: Memory Hierarchy



L1 Cache
~32KB, ~few ns

L2 Cache
512 KB ~ 1MB, ~many ns

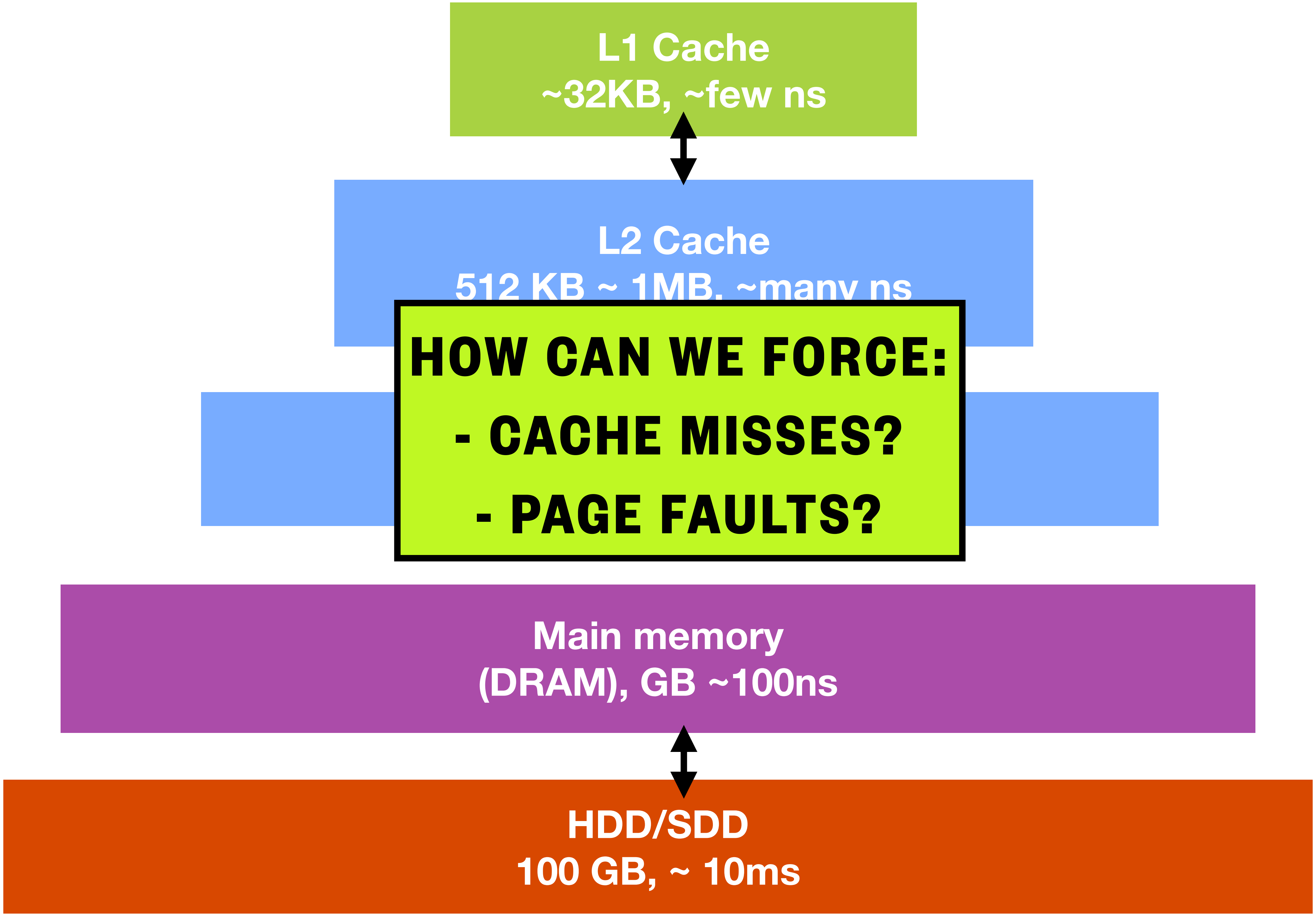
L3 Cache …
512 KB ~ 1MB, ~many ns

…

Main memory
(DRAM), GB ~100ns

HDD/SDD
100 GB, ~ 10ms

Recall: Memory Hierarchy



HOW CAN WE FORCE:

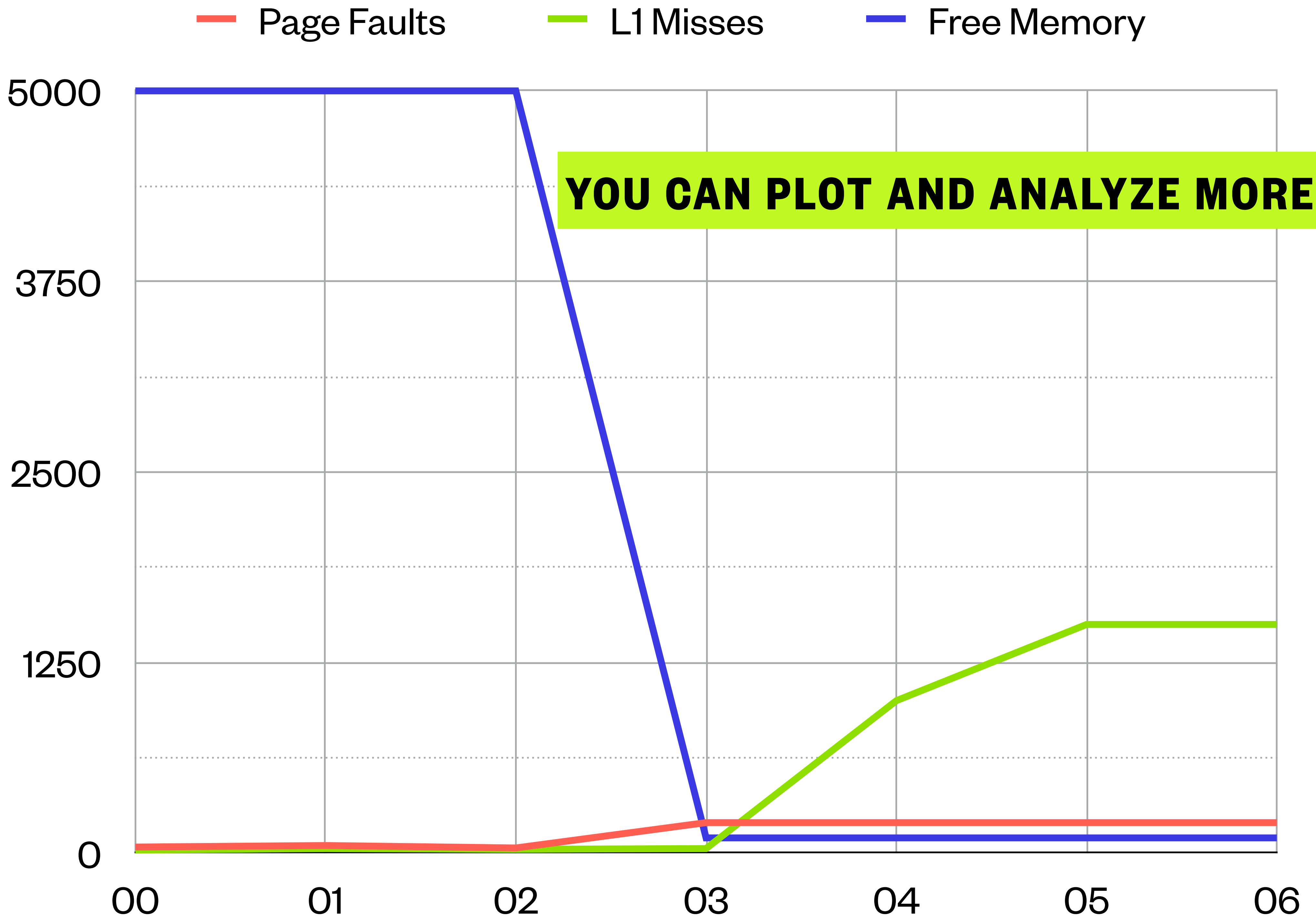
- CACHE MISSES?

- PAGE FAULTS?

Exercise 1:

1. Identify the architecture of your system: cache levels, cache size, block size, frequency, dram size, tlb levels. **Summarize in a table.**
2. **Implement a program Pfaults** to force misses and page faults:
    1. Create an array that exceeds the DRAM size. **Notice:** are we using virtual memory ;) ?
    2. Perform random accesses over the array in memory (reads/loads).
    3. Control the execution time of your program (define **X** minutes).
3. Obtain all the data from cache (misses), TLB (misses), page faults, free memory using **perf** of the (total execution time **2X** minutes) **:**
    1. **First X minutes:** in your normal system state, while you are working and not executing the Pfaults program.
    2. **Last X minutes**: while executing the Pfaults program.
    3. Parse the perf output and plot your results. **Summarize** in a table.
4. Explain your plot. Justify.

Plot example



YOU CAN PLOT AND ANALYZE MORE VARIABLES

TODO: Check with perf:

```
# Various CPU level 1 data cache statistics for the specified command:
perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores command

# Various CPU data TLB statistics for the specified command:
perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses command

perf stat -e cpu-clock ./programa
perf stat -e cpu-clock,faults ./programa
perf report --stdio --sort comm,dso
```

IOstat

- Install IOstat:

```
sudo apt-get install sysstat
```

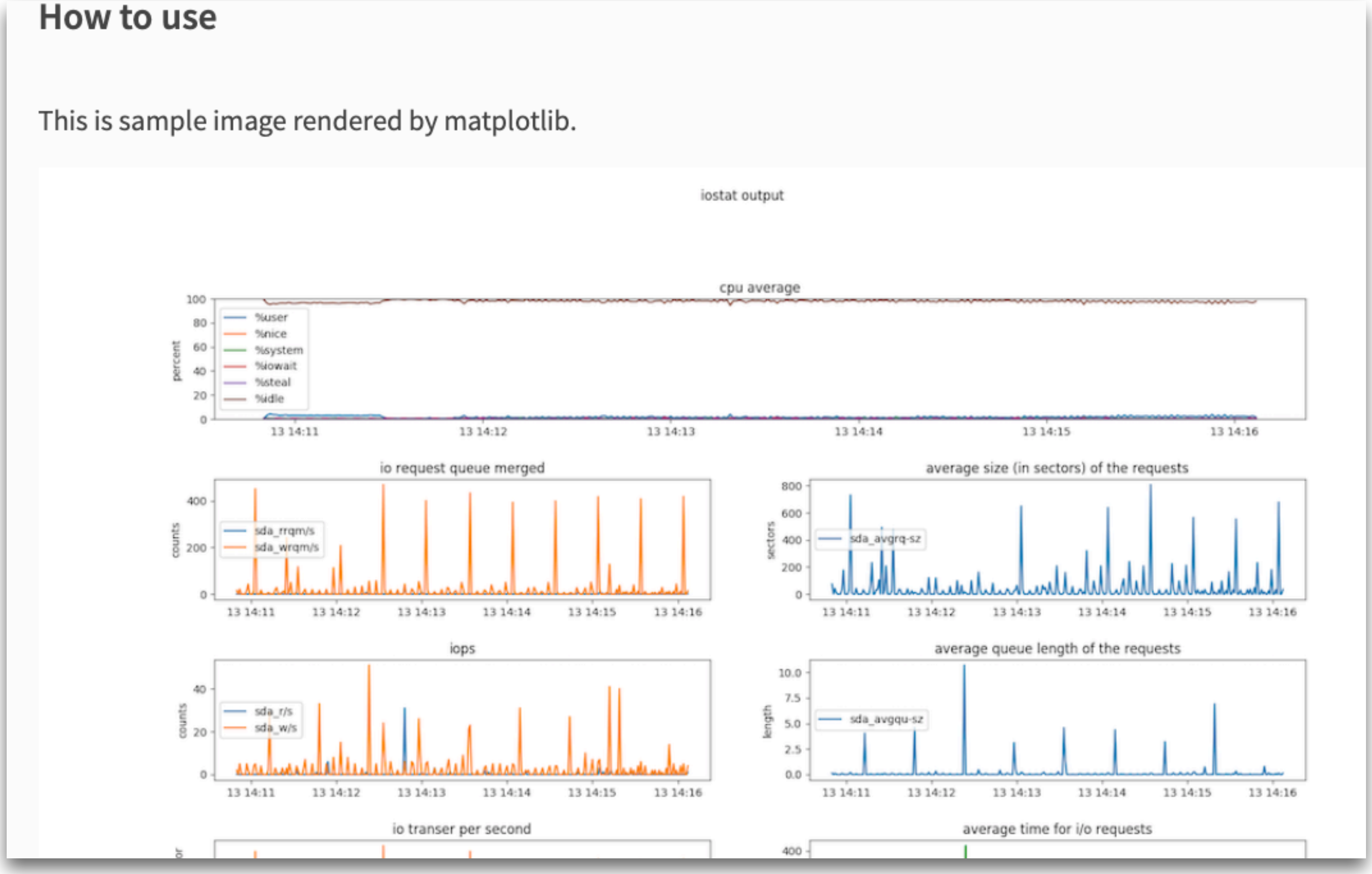
- Examples

```
$iostat
$iostat -d 5 3

Linux 3.19.0-25-generic (Ubuntu-PC)    Saturday 16 December 2017
_x86_64_  (4 CPU)

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda              11.77       340.71        98.95     771022     223928

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               2.00         0.00         8.00          0         40

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               0.60         0.00         3.20          0         16
```

IOstat tool

- Python library for parsing IOstat results.

HTTPS://PYPI.ORG/PROJECT/IOSTAT-TOOL/



How to use

This is sample image rendered by matplotlib.

Io streams with Python

HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/IO.HTML

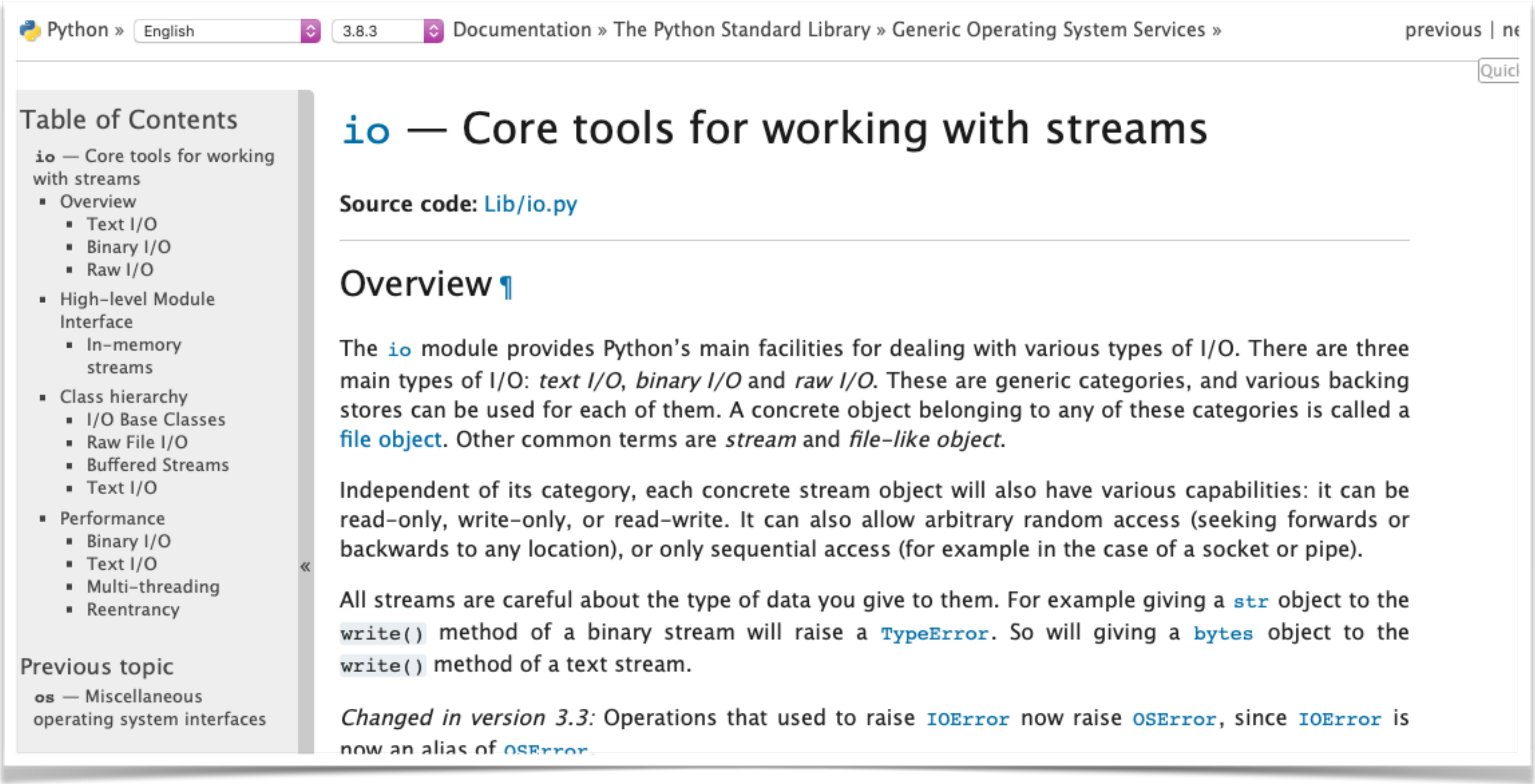

Python » English 3.8.3 Documentation » The Python Standard Library » Generic Operating System Services »
previous | ne
Quicl
«

Table of Contents

Previous topic

os — Miscellaneous operating system interfaces

io — Core tools for working with streams

Source code: Lib/io.py

Overview ¶

The io module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O: *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a file object. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a `str` object to the `write()` method of a binary stream will raise a `TypeError`. So will giving a `bytes` object to the `write()` method of a text stream.

Changed in version 3.3: Operations that used to raise `IOError` now raise `OSError`, since `IOError` is now an alias of `OSError`.

Exercise 2

- Perform I/O (read and write) operations.

  1. Implement a program with the following functions:

     A. Random reads size 200MB.

     B. Random writes size 200MB.

     C. Sequential reads size 200 MB.

     D. Sequential writes size 200 MB.

     E. Random writes and reads with variable size (max 500MB).

     F. Define an amount of X minutes for each function call (max 2min).

  2. Generate an output file using Iostat during your program execution.

  - **Notice: The iostat data can reach GB file size.**

  3. Process and plot your data using the Python library for IOstat.

  - Explain your results. **Justify.**