introduction to database systems storage, multicores, OS

Pınar Tözün

March 22, 2023

some slides are inspired by
Patterns in Data Management and Databases on Modern Hardware books &
Anastasia Ailamaki's Intro to Database Systems class at EPFL

agenda

- storage hierarchy
- hardware parallelism on multicores
- operating systems

why do we need to know these?

impacts how we design database systems & leads to better optimizations for faster data processing

systems stack overview

application

e.g., online shopping page, database system, code to read/write a file, etc.

operating system

e.g., linux, windows, etc.

hardware

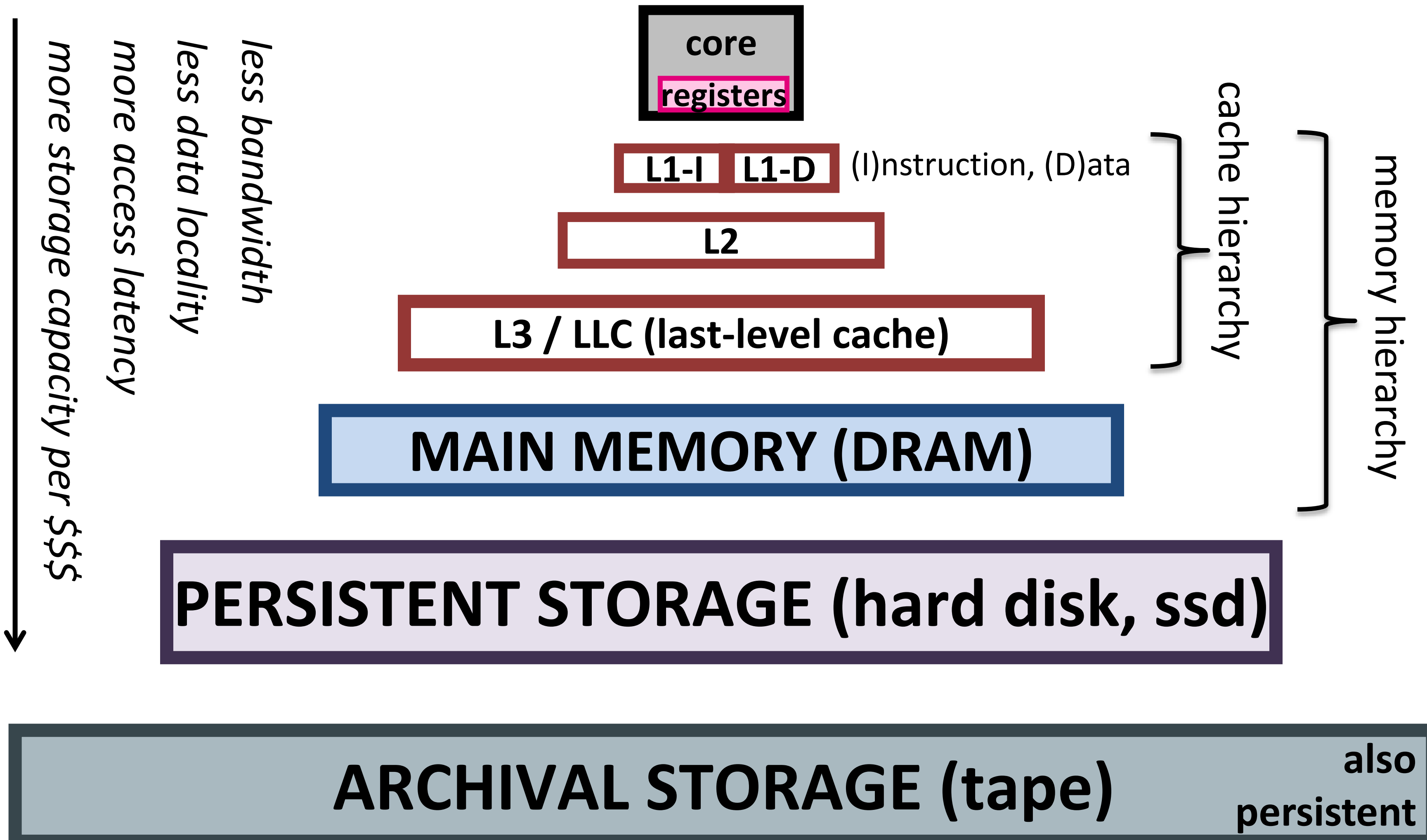
e.g., intel server, disks, etc.

agenda

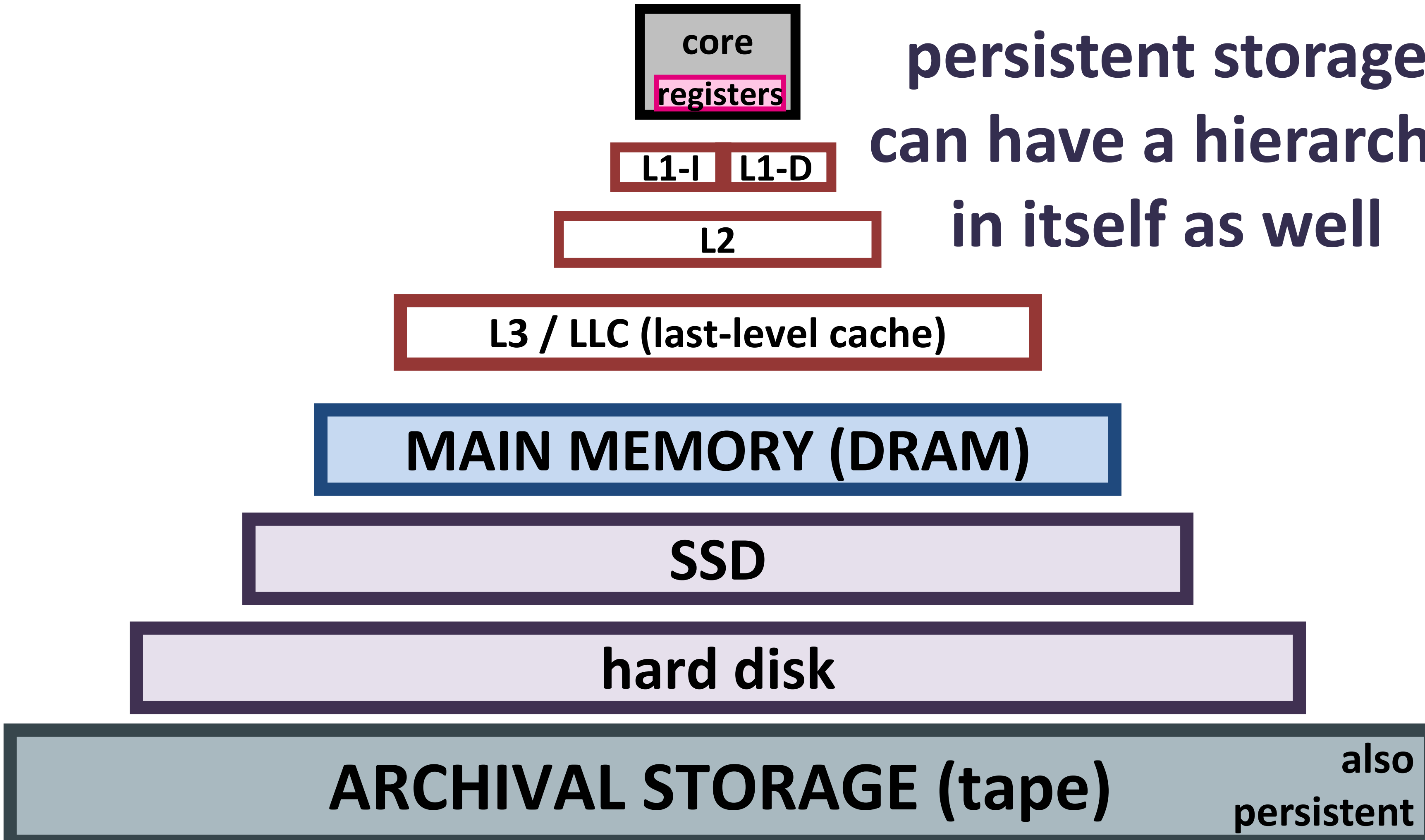
- **storage hierarchy**
- hardware parallelism on multicores
- operating systems

(typical) storage hierarchy

disclaimer: memory hierarchy is based on Intel Xeons' here



(typical) storage hierarchy



persistent storage can have a hierarchy in itself as well

(typical) storage hierarchy

distributed setting (e.g., cluster of machines)

persistent storage can have a hierarchy in itself as well

core

registers

L1-I | **L1-D**

L2

L3 / LLC (last-level cache)

MAIN MEMORY (DRAM)

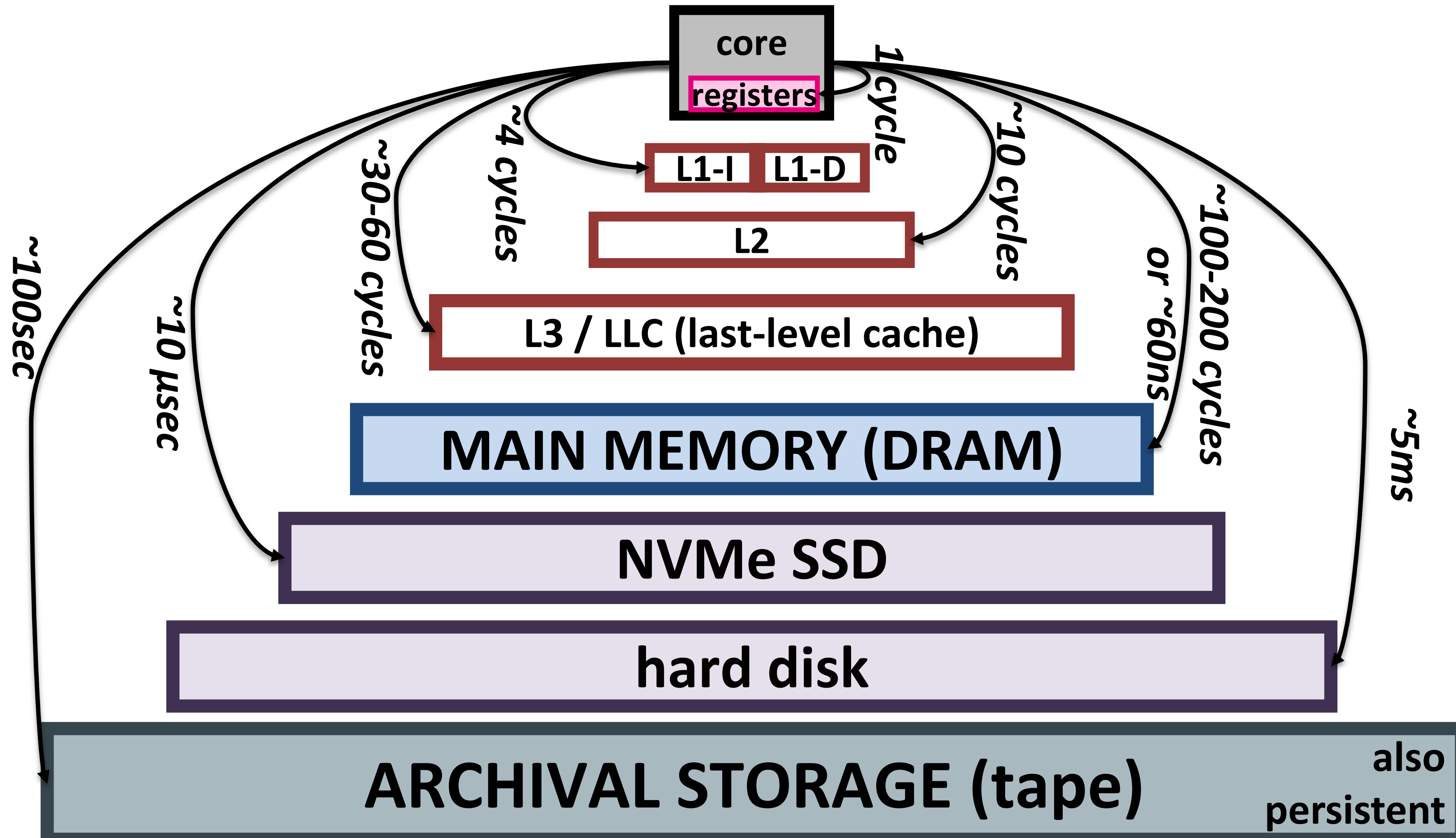
local disk

remote disk

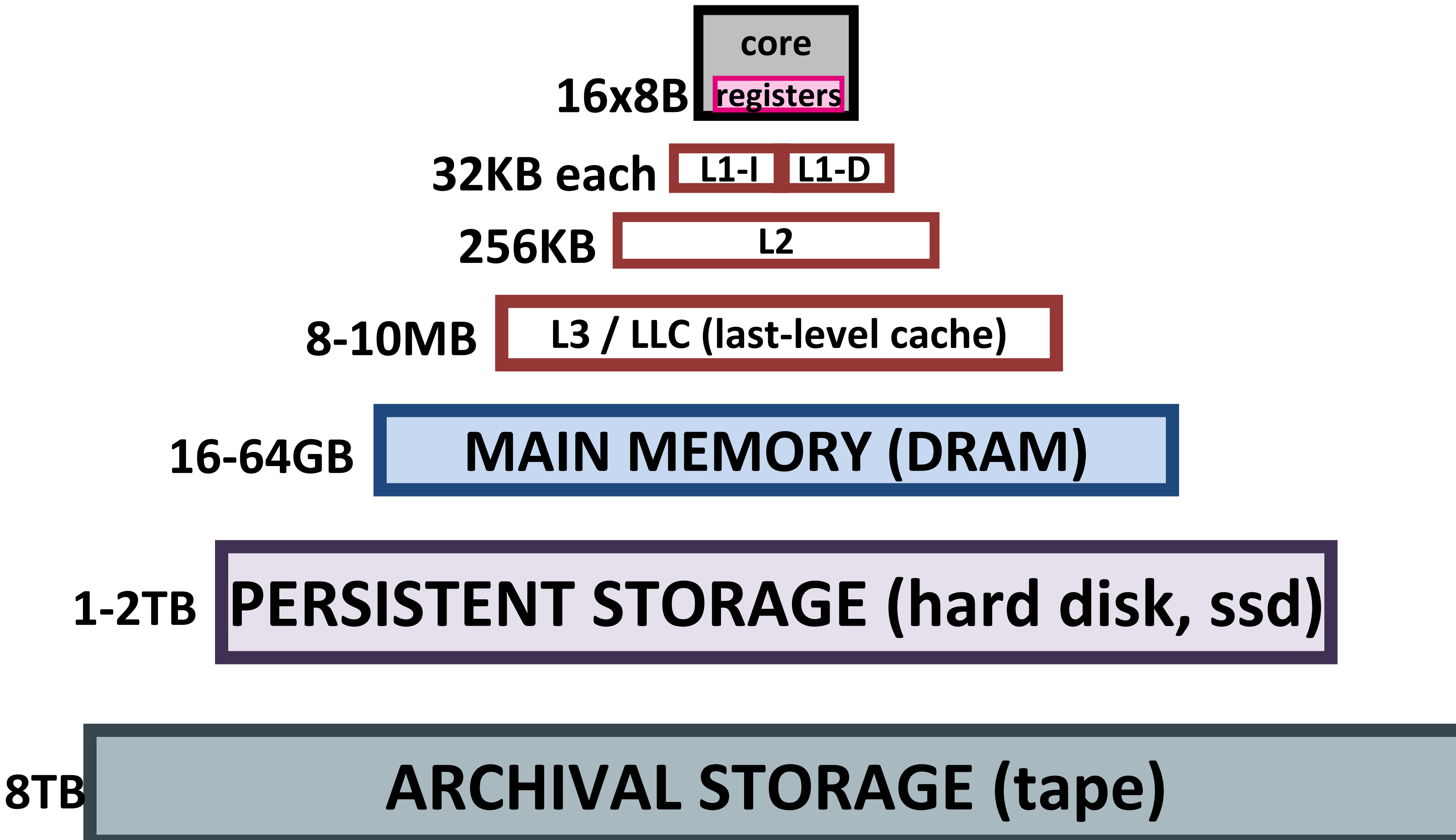
ARCHIVAL STORAGE (tape)

also persistent

(typical) storage hierarchy – access latency



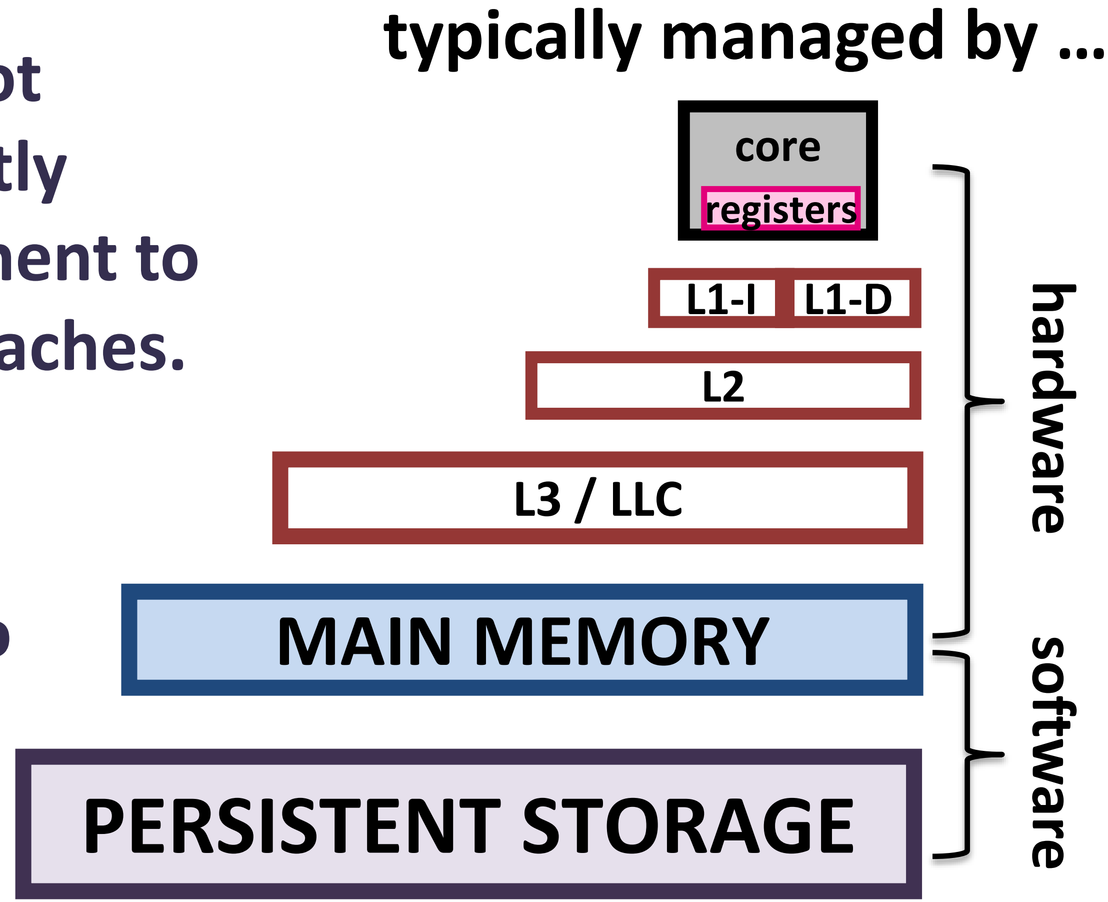
(typical) storage hierarchy – capacity



(typical) storage hierarchy – management

means that we do not write code to explicitly manage data movement to registers, L1, L2, L3 caches.

but we do this for moving data from/to persistent storage.



storage hierarchy

goal is to increase data locality for cores

to reduce access latency for frequently accessed data

higher levels cache data from lower levels

inclusivity

- lower levels include all the data from higher levels (usually)
- most hardware vendors build inclusive cache hierarchy
- software controlled caching can be more complex

  e.g., some database systems don’t persist indexes on disk,

  they keep them in main memory

data replacement when no space left

replacement policy can be a crucial system optimization

tape

cheap way of storing voluminous data

only allows ***sequential access***

does not allow ***random access***

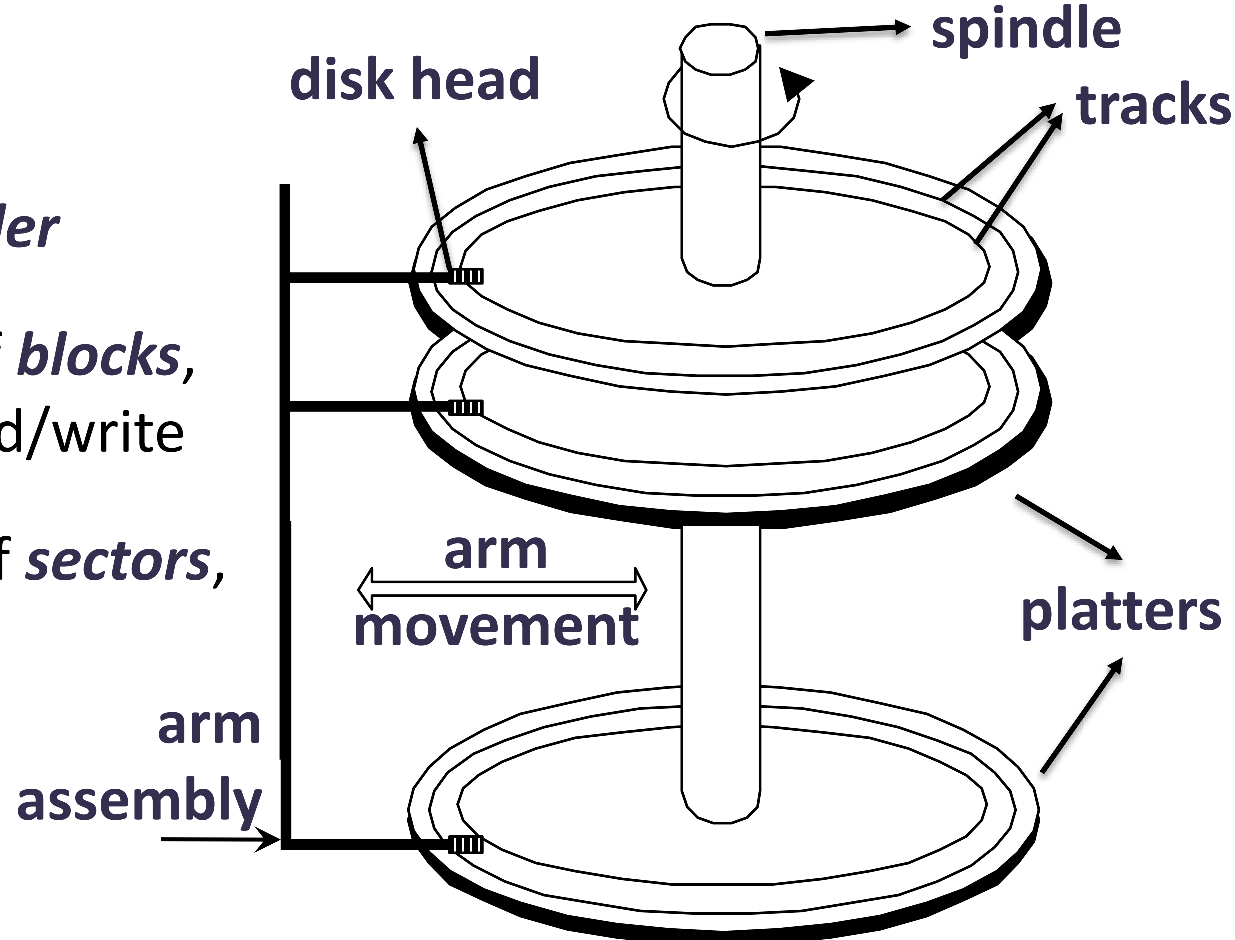
only used for archival storage today

hard disk

tracks with the same diameter make a ***cylinder***

tracks are composed of ***blocks***, which is the unit of read/write

blocks are composed of ***sectors***, which are of fixed-size

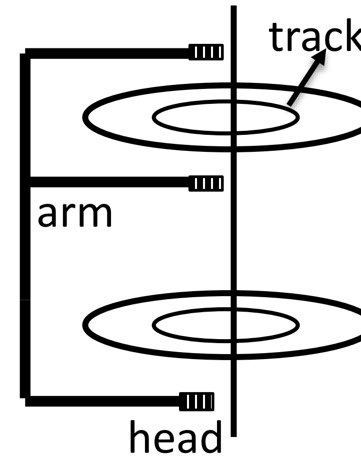


placement of data on disk is a crucial concern for access latency!

access latency on hard disk

also called I/O (input/output) latency

= seek time + rotational delay + transfer time



seek time

moving disk arm to the right track – (~1-20 ms)

rotational delay

reaching the desired block on the track – (~0-10 ms)

transfer time

reading/writing the desired data on the block – (~<1ms for 4KB)

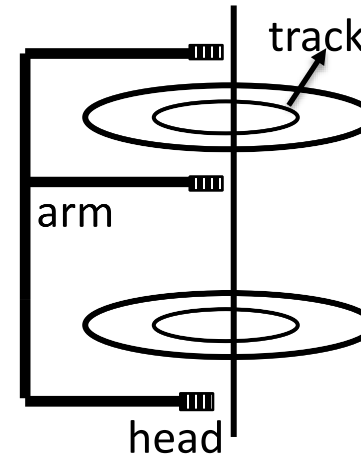
i.e., disk head rotating over the block

for faster access →

minimize seek time & rotational delay!

random vs. sequential access on hard disk

access latency for reaching a disk block = *a* =
seek time + rotational delay
reading a block = *a* + transfer time



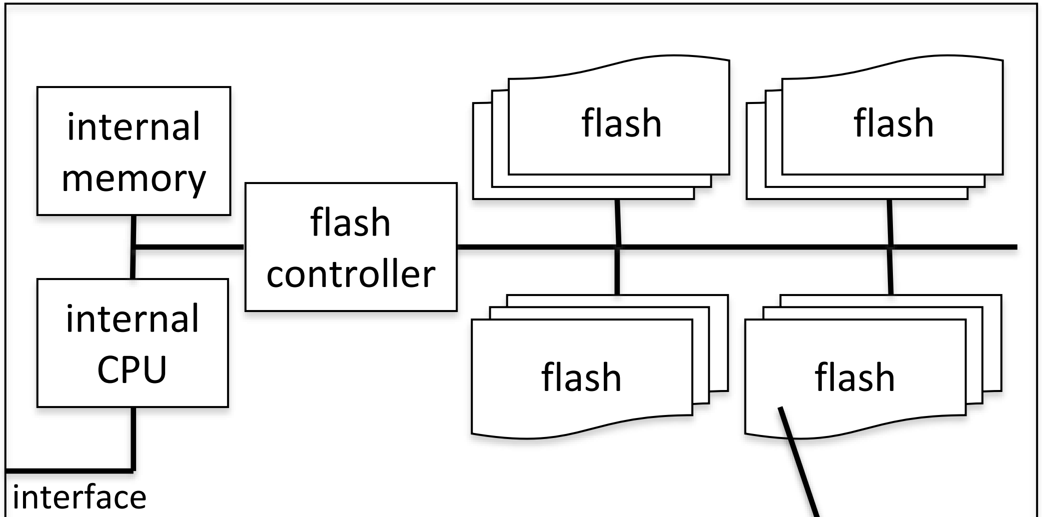
access latency for 100 random blocks =
100 x (*a* + transfer time)

access latency for 100 sequential blocks =
a + 100 x (transfer time)

sequential access is much faster than random access!

underlying principle for many optimizations in data systems

solid-state disk (SSD)

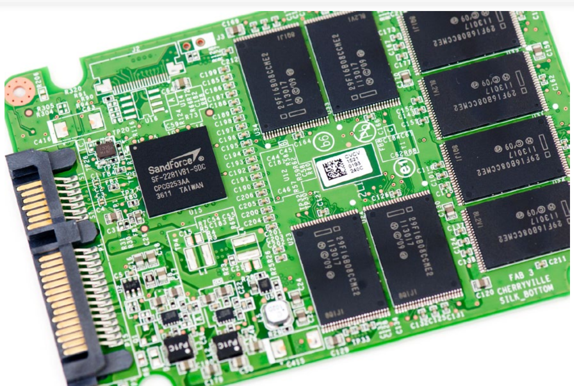


interconnected flash chips

efficient random access

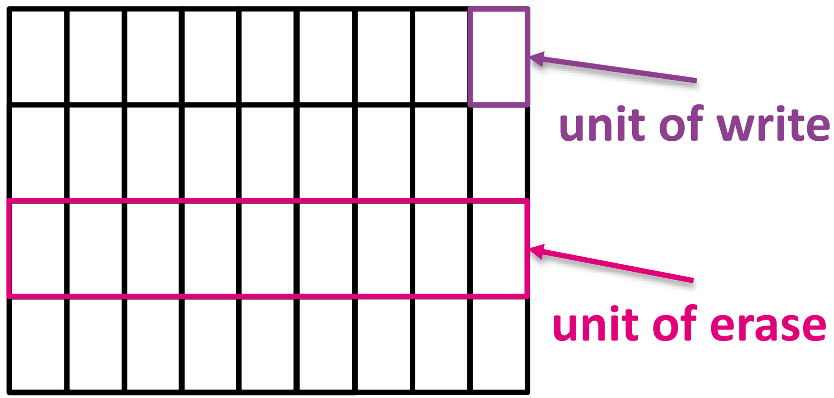
internal parallelism

hard disk compatible API



why not have as drop-in replacement for hard disks?

solid-state disk (SSD)



can use it as drop-in replacement for hard disks,

but need to be smarter to more effectively exploit SSDs & not to burn money!

cannot override a unit before erasing it first

garbage collection – for not used blocks so we can rewrite them

write amplification = data physically written / data logically written >= 1

writing data might cause rewrites & garbage collection

wear leveling – some cells/blocks die over time

unpredictable read/write latencies

if a request gets stuck after a write triggering garbage collection

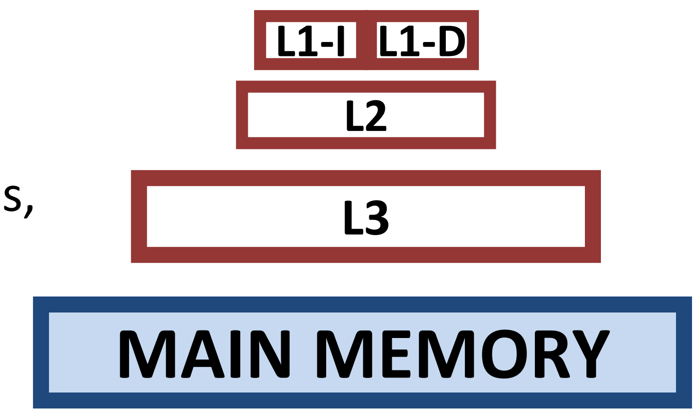
random-access memory (RAM)

(almost) ***constant random-access latency*** wherever the data is

volatile*
will loose data once power is lost

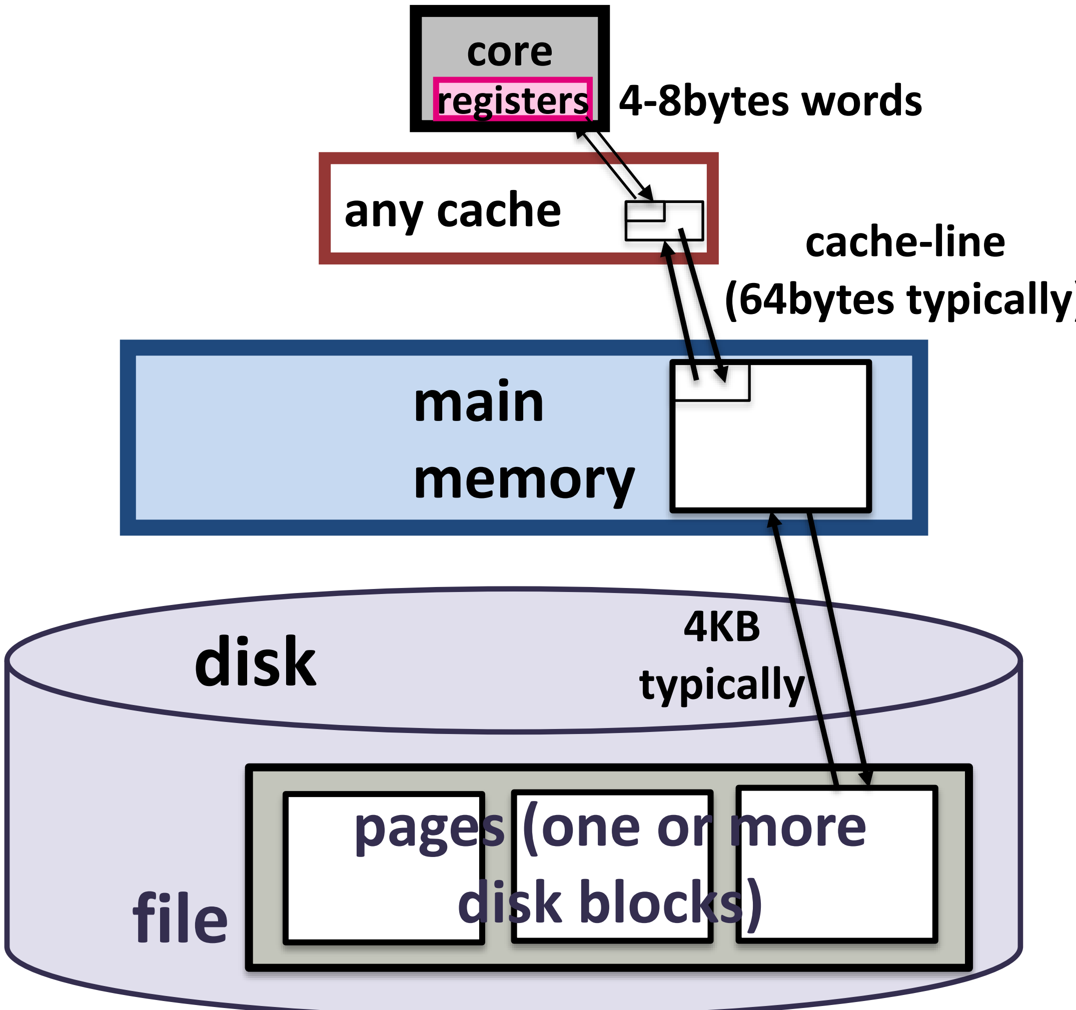
sequential access is slightly faster still

- ***prefetching***:
  if you fetch a block from main-memory to caches,
  hardware usually prefetches the adjacent block



*non-volatile/persistent memory is available, but not adopted in main-stream

movement of data in storage hierarchy



summary – storage hierarchy

Storage hierarchy is there to improve locality for frequently accessed data.

Different layers of the hierarchy have different characteristics & require different optimizations from the software side.

Sequential access is faster than random access for all layers, but especially for hard disks.

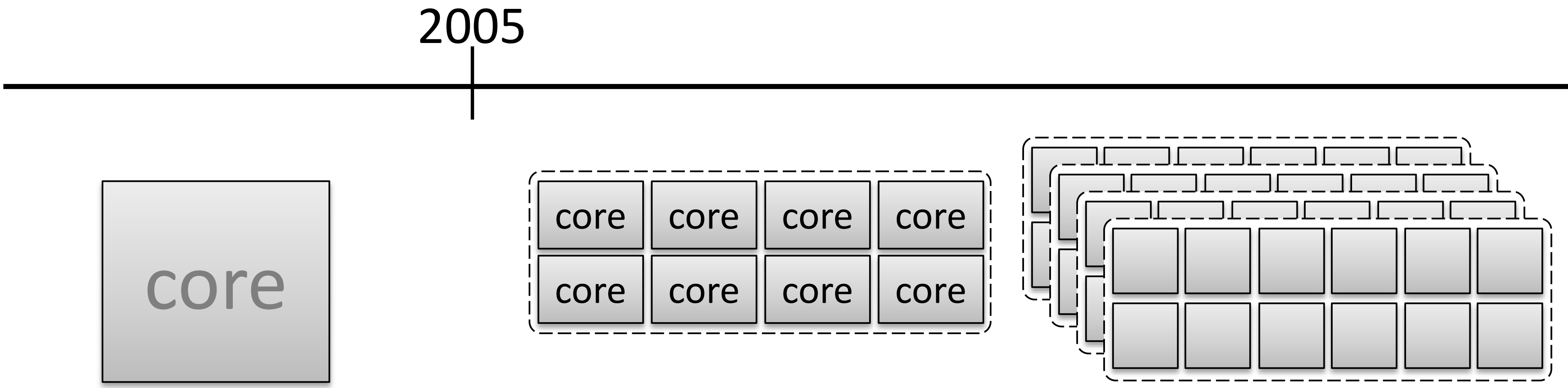
Data management systems have various optimizations to exploit the storage hierarchy the best possible way.

goal: minimize data access latency!

agenda

- storage hierarchy
- **hardware parallelism on multicores**
- operating systems

central processing unit (CPU) evolution



single-core CPUs

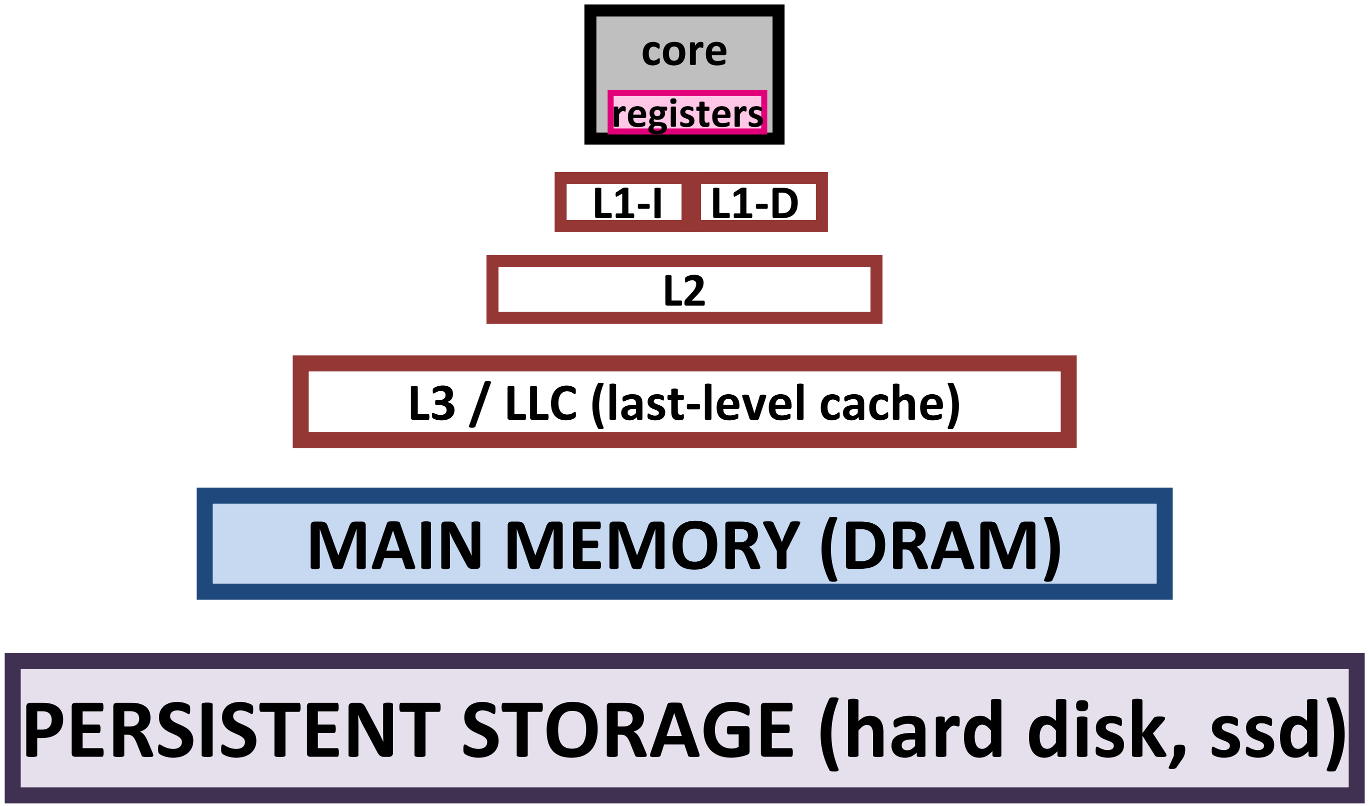
multicore CPUs

multisocket multicore CPUs

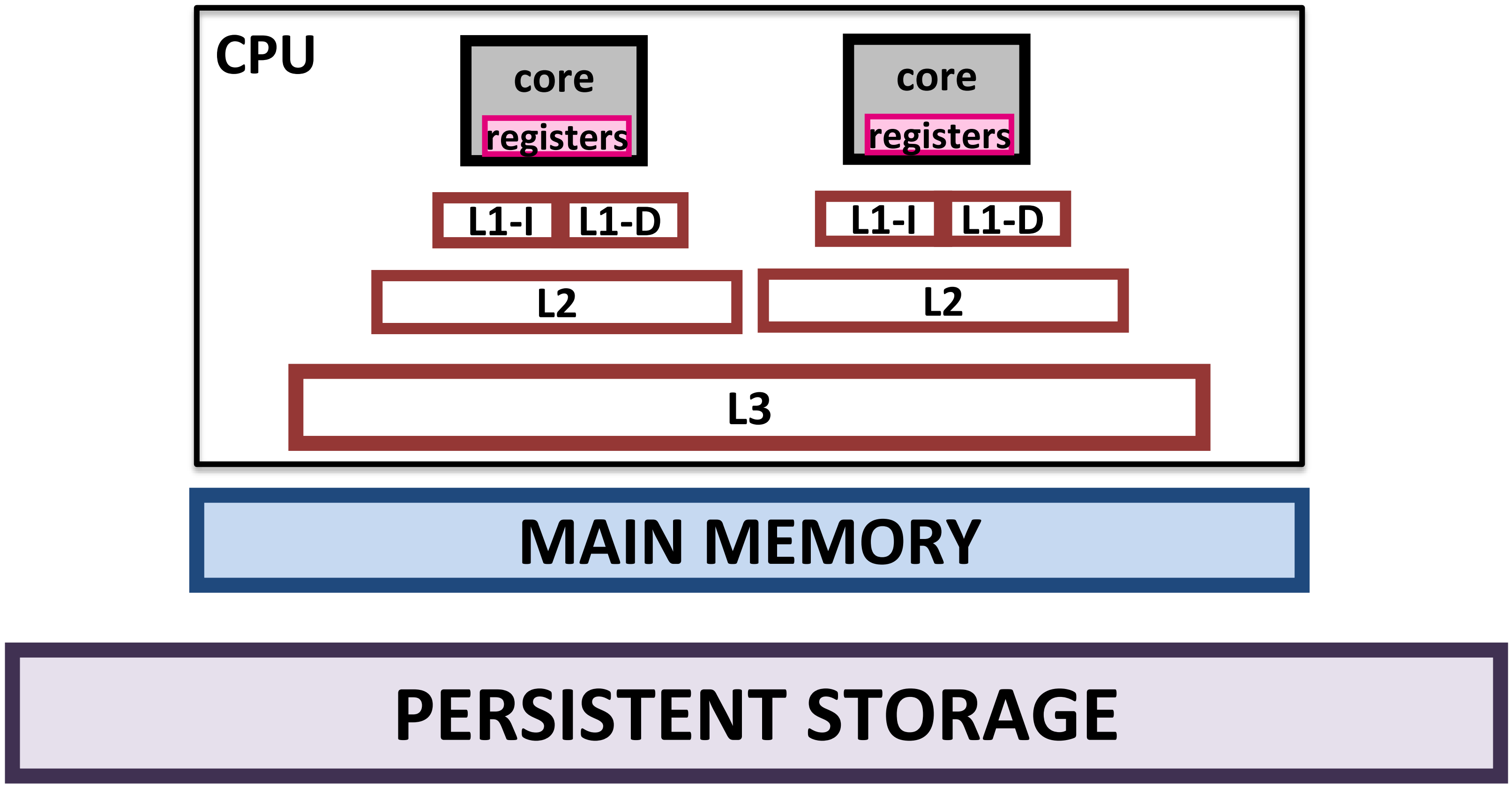
faster & more-complex cores over time

similar speed & complexity in a core, more parallelism over time

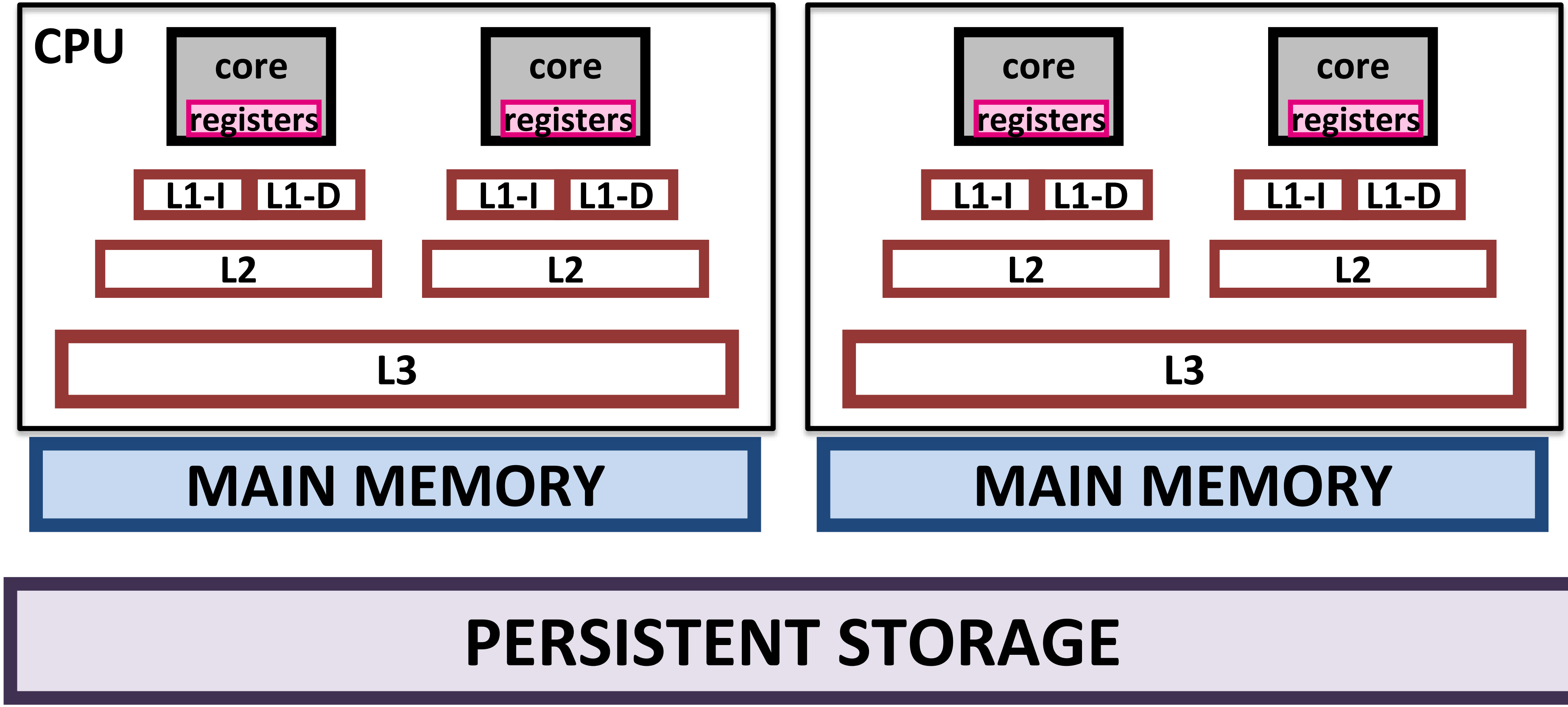
single-core storage hierarchy



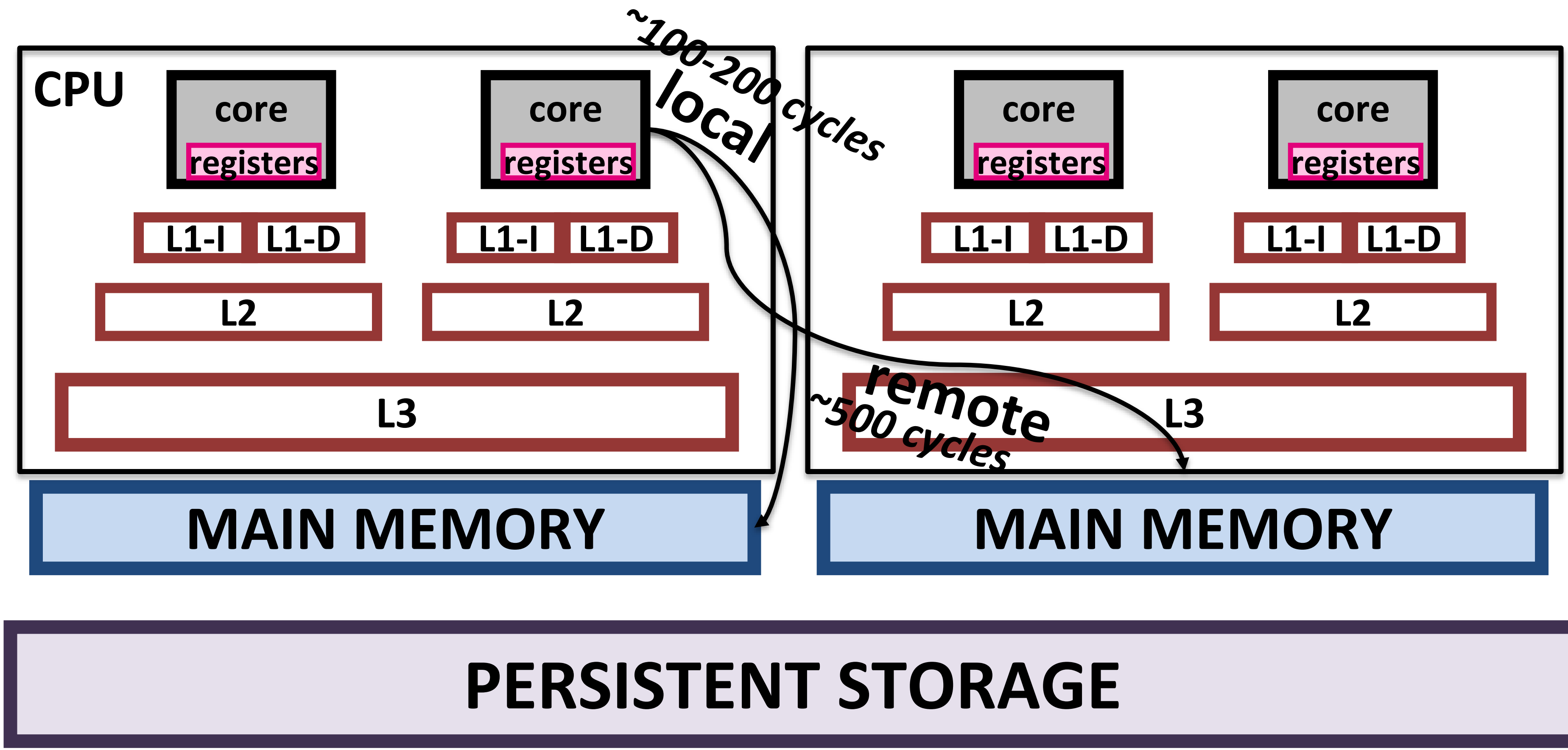
multicore storage hierarchy



multi-socket multicore storage hierarchy

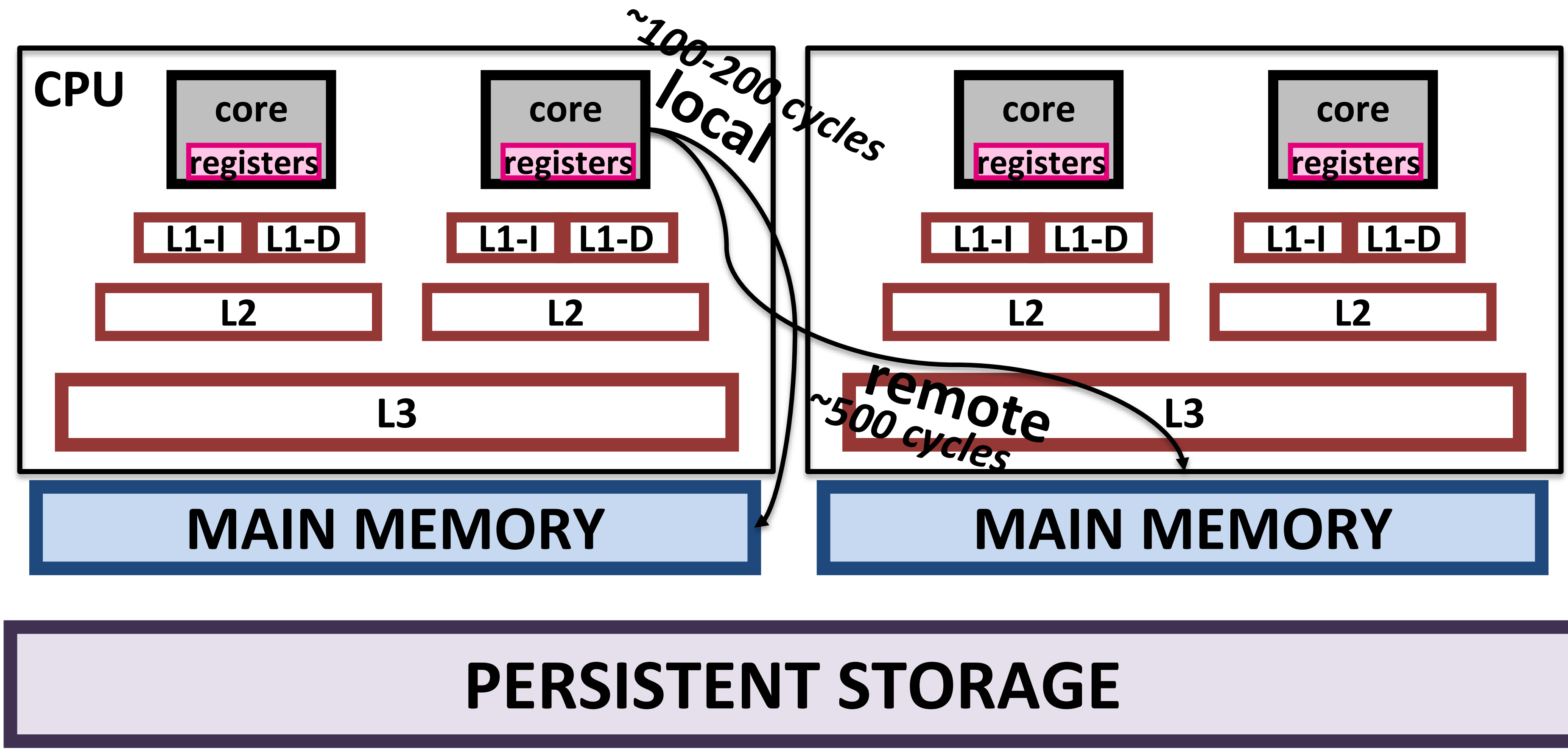


multi-socket multicore storage hierarchy



local memory access is faster than remote one!

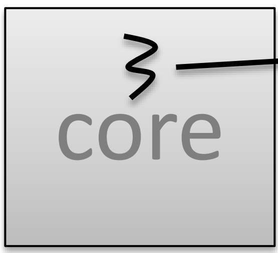
multi-socket multicore storage hierarchy



also called NUMA, non-uniform memory access

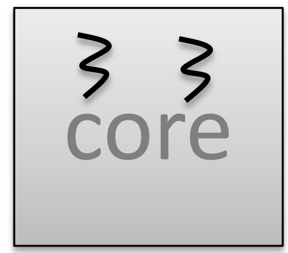
types of hardware parallelism

implicit parallelism

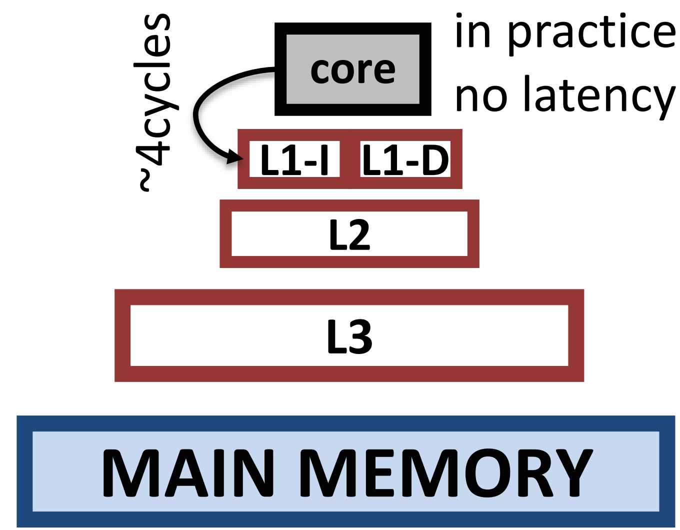


thread
what runs
your programs

instruction & data parallelism
hardware does this automatically

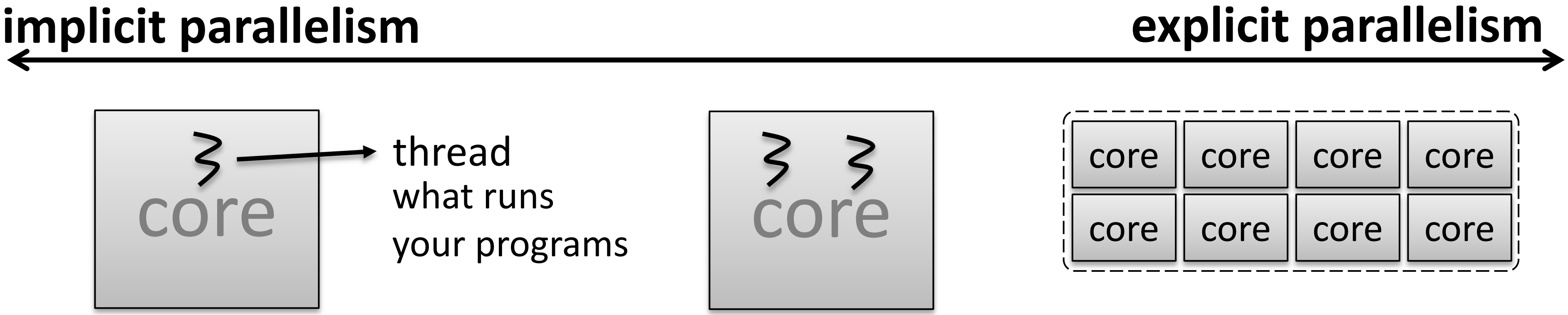


multithreading
threads share
execution cycles
on the same core



goal: minimize stall time due to cache/memory accesses
overlapping access latency for one item with other work

types of hardware parallelism



instruction & data parallelism
hardware does this automatically

multithreading
threads share
execution cycles
on the same core

multicores
multiple threads run in
parallel on different cores

implicit parallelism → (almost) free lunch

explicit parallelism → must work hard to exploit it

summary – hardware parallelism

Hardware gives different parallelism opportunities.

Database systems used to be ignorant of this parallelism because it used to be implicit.

Today, data management systems do not have this luxury because we also have explicit parallelism.

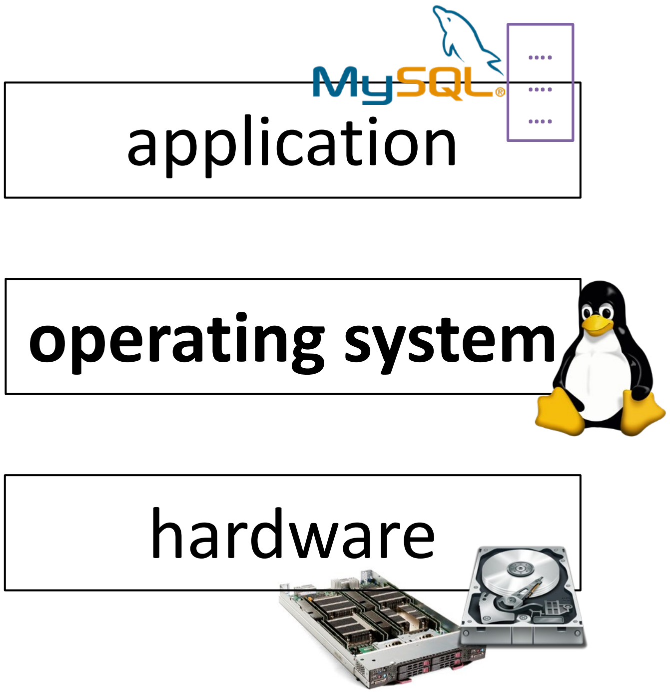
Explicit parallelism also complicates memory hierarchy.

goal: design systems that are aware of the hardware parallelism (ideally all types of it) & its implications!

agenda

- storage hierarchy
- hardware parallelism on multicores
- **operating systems**

why do we need operating systems?



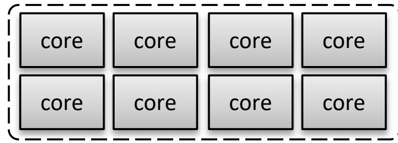
resource management

- many applications running
- many users want to use them
- hardware resources are limited

need something to reliably & efficiently share hardware resources across many users

what are the resources?

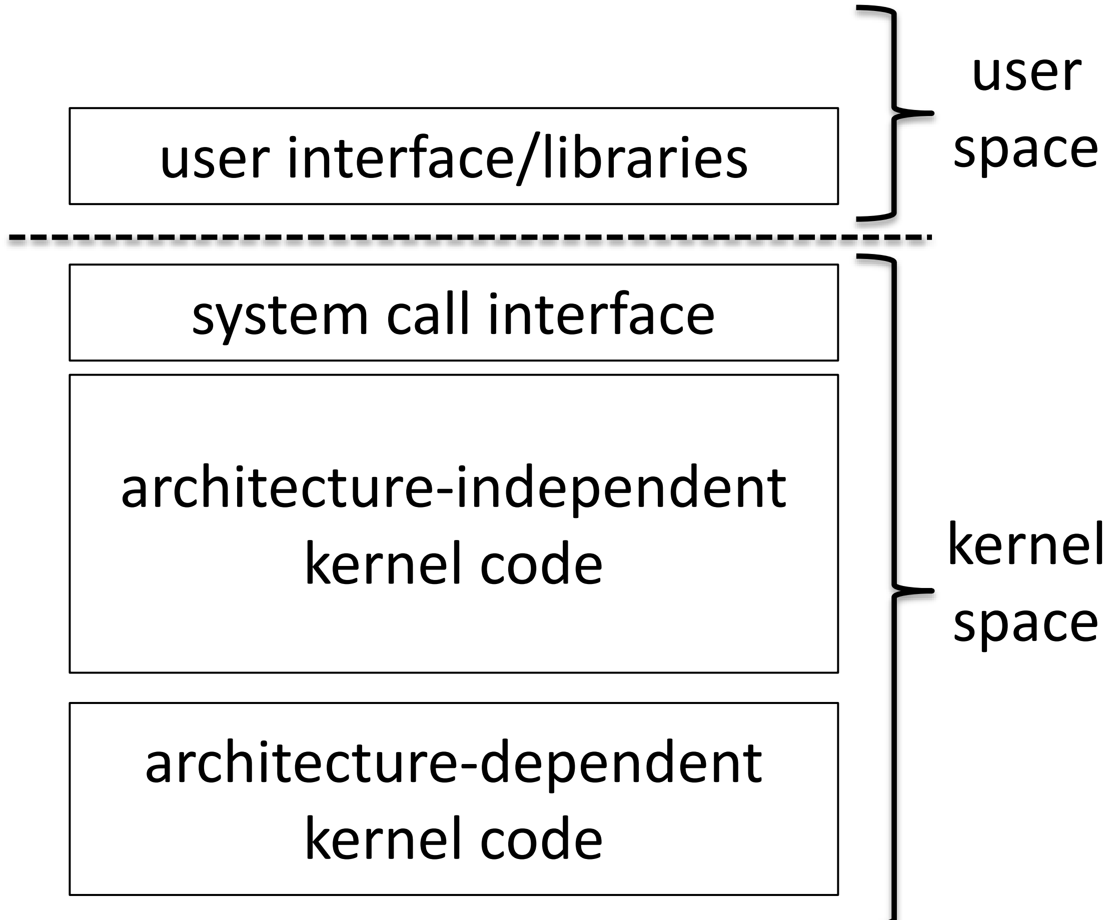
CPU



memory

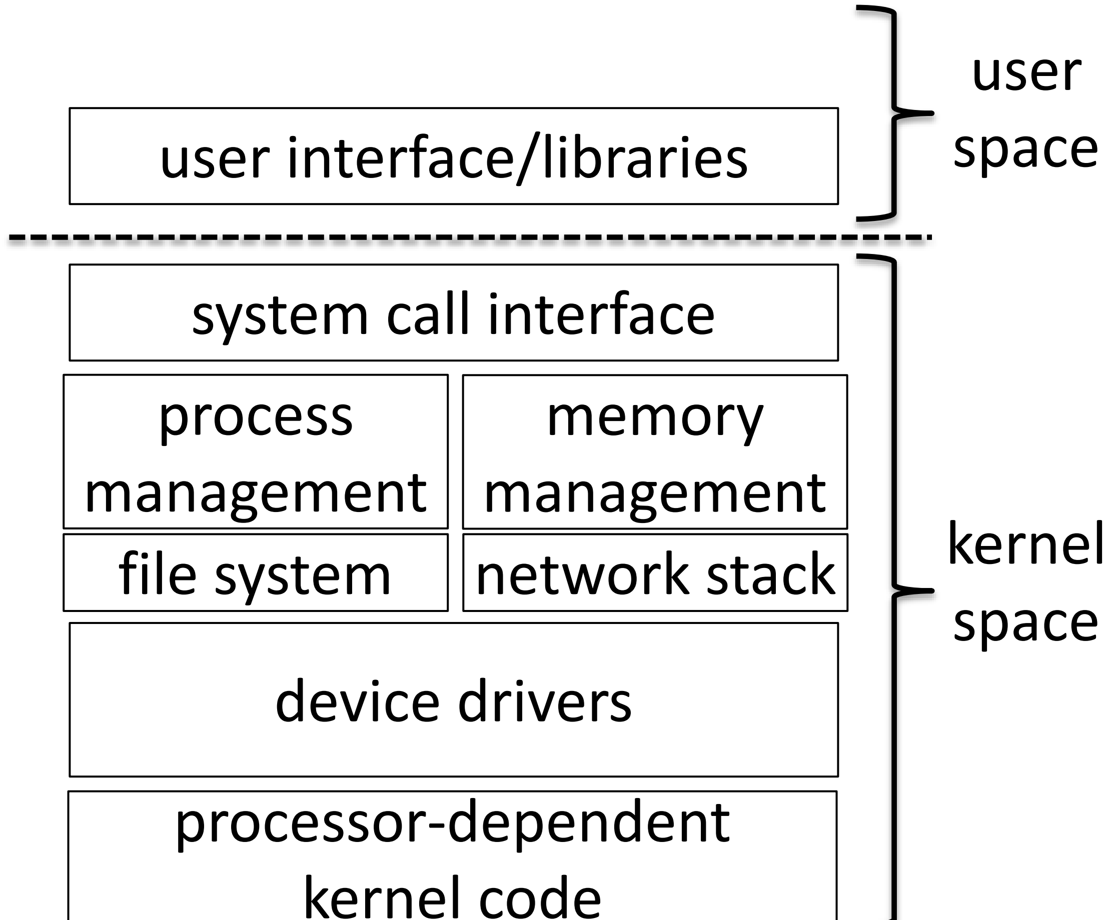
I/O devices

typical operating system components



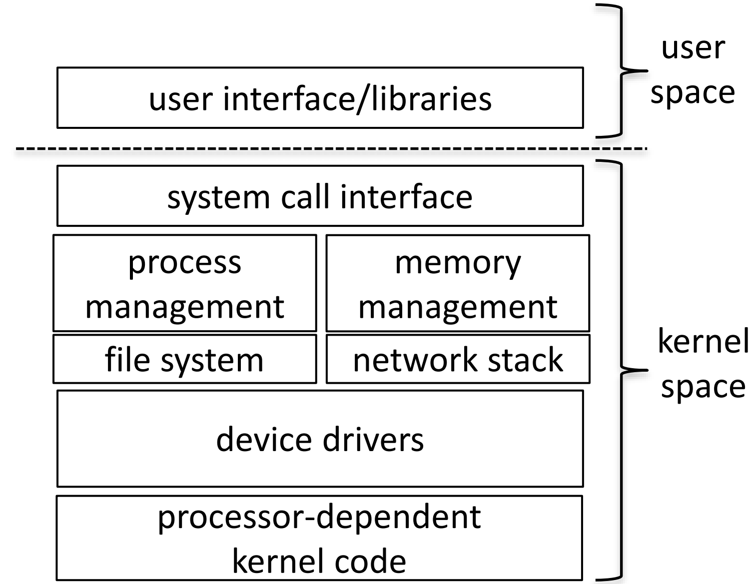
disclaimer: mainly based on Linux

typical operating system components

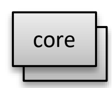


disclaimer: mainly based on Linux

process management



virtualizes a processor
giving the illusion of infinite #cores

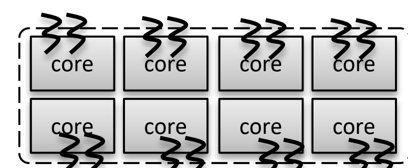
user space has application(s)
an application has process(es)
a process has ***thread(s)*** 
threads are mapped to ***cores*** 

by default operating systems handles this mapping
application can request to run on specific cores

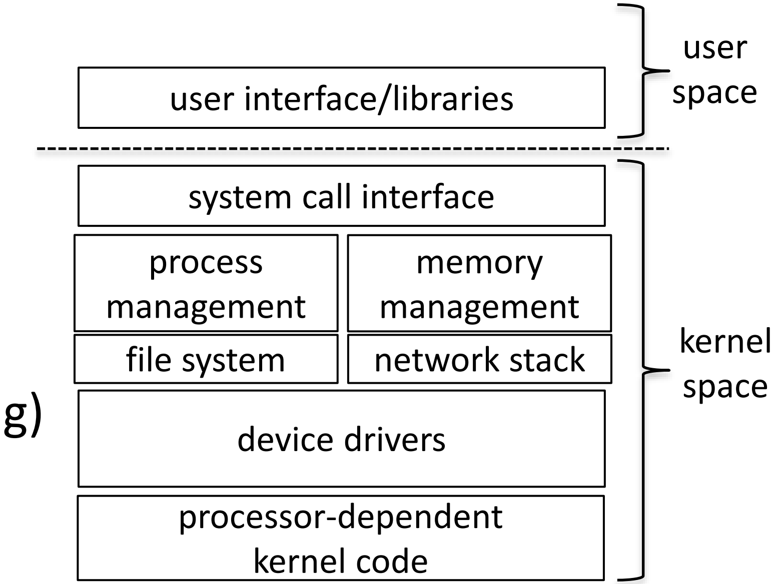
```
int sched_setaffinity(pid_t pid, // thread id
      size_t cpusetsize, // sizeof(cpu_set_t)
      const cpu_set_t *mask); // bitmap indicating cores you want the thread to run on
```

or taskset, numactl command line options

multicore processor



process management

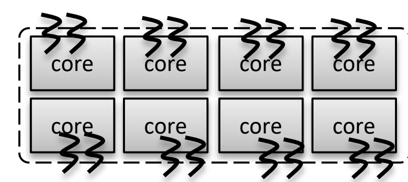


#threads that can be active at a time
➔ ***#hardware contexts (logical cores)***
a processor supports

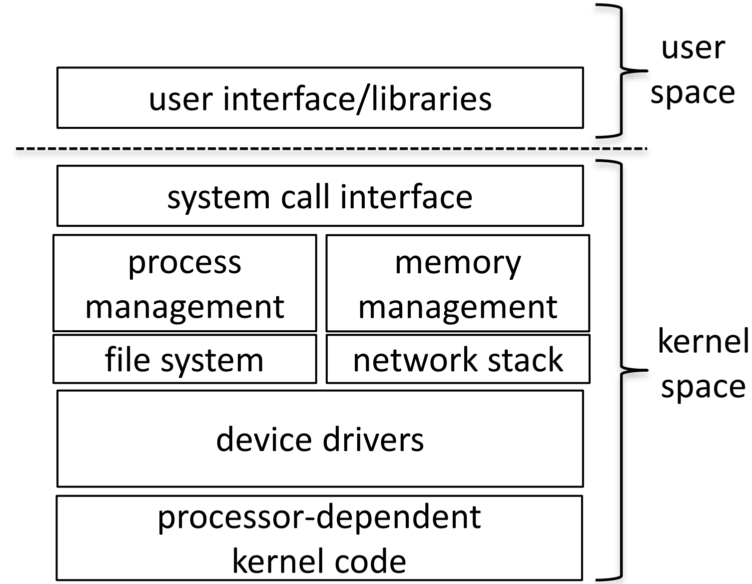
- typically 2 per core on Intel (hyper-threading)
- Sun Spark T2 had 8 per core

what if we have more threads to run?

- operating system ***context switches*** to swap threads' state/context in & out
  if one thread is inactive (due to IO or sleeping), another one runs
- works well for general-purpose scenarios
  you typically have IOs that interrupt a thread



memory management



virtualizes memory
giving the illusion of infinite memory

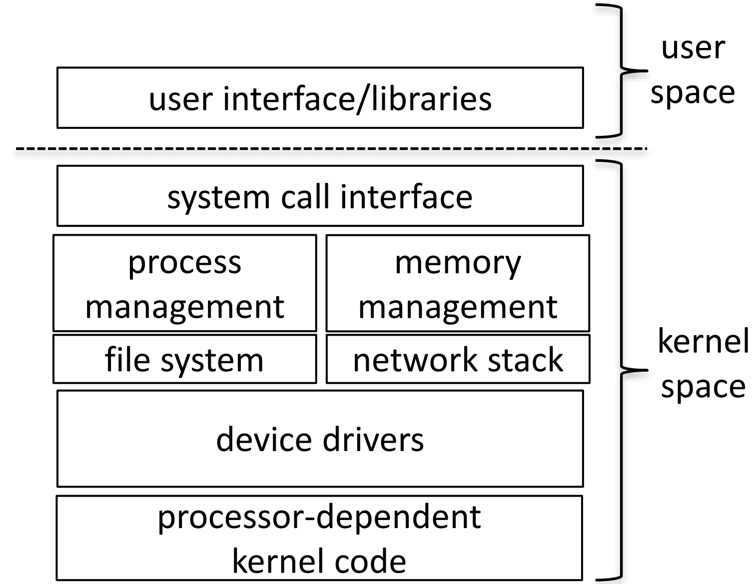
user space has application(s)
an application has process(es)
a process has its own
memory address space = virtual memory

by default operating systems maps a process' address space to the available bytes of physical memory

- manages free space, segmentation …
- numactl command line tool also allows binding a process to a memory region

physical memory = array of bytes

memory management



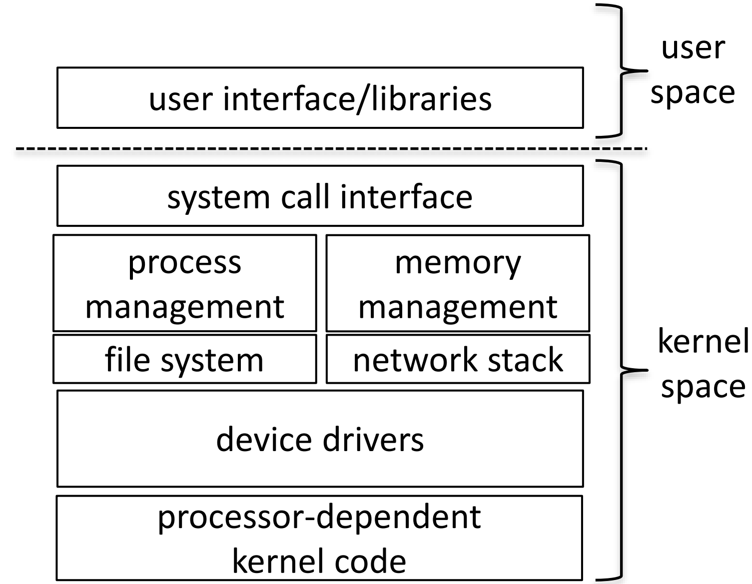
virtualizes memory
giving the illusion of infinite memory

user space has application(s)
an application has process(es)
a process has its own
memory address space = virtual memory

aggregate memory used by all processes can be larger than available physical memory

- operating system swaps things back & forth as needed
  → to/from swap space on disk
- if has to be done too frequently, your program won't perform well

memory management



virtualizes memory
giving the illusion of infinite memory

user space has application(s)
an application has process(es)
a process has its own
memory address space = virtual memory

threads in a process share the same memory space
it is application's responsibility to manage this shared space reliably

- use locks/mutexes/atomics
- partition this space to each thread

summary – operating systems

Operating system is a **resource manager virtualizing** hardware resources for applications/end-users.

More specifically, its goal is to manage hardware resources reliably & efficiently for many applications/end-users who are using these resources concurrently.

Operating system also provides an **abstraction** layer for applications to have a common and **easy-to-use interface** while interacting with a variety hardware resources / devices.

Virtualizations & abstractions come at a cost, though!

- indirect management of hardware resources
- need to think about trade-offs of the indirection

OS vs. DB

vs.

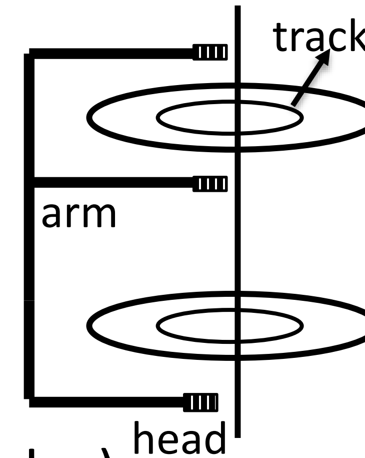
backup

what does sequential mean on hard disk?

(1) reach the first desired block

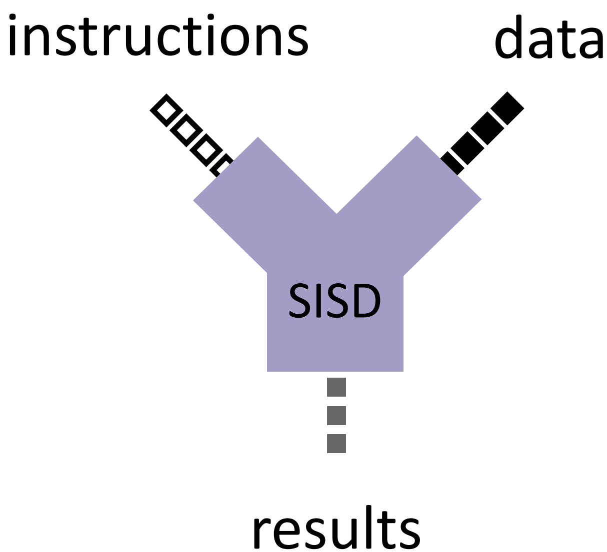
(2) read adjacent blocks on the same track

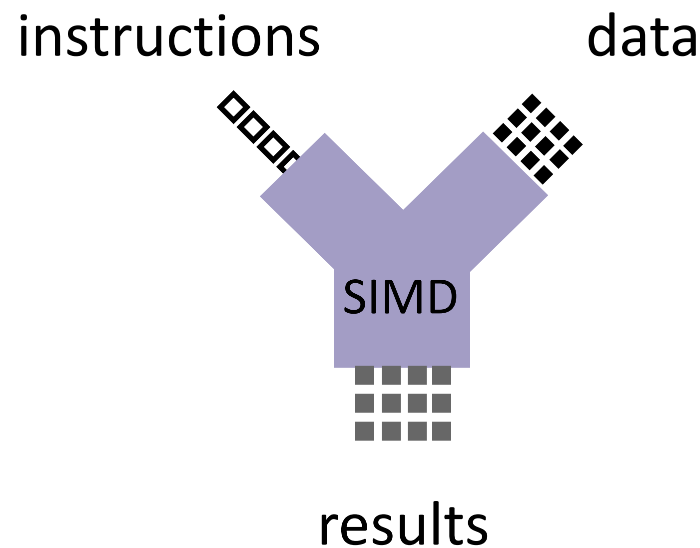
(3) read blocks on the same cylinder
(switch to different disk head, then short rotational delay)



(4) read blocks on the adjacent cylinder
(short-distance seek time, then short rotational delay)

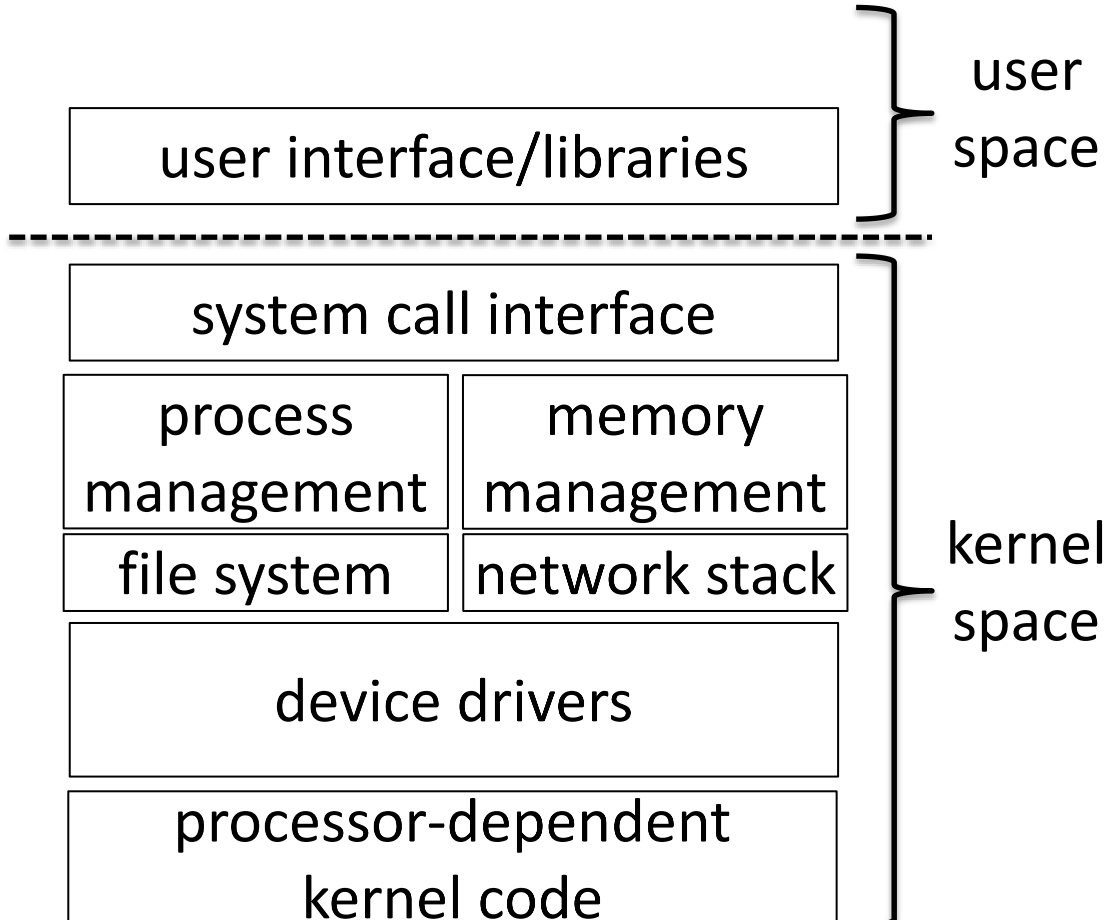
single instruction multiple data (SIMD)





GPUs are like SIMD machines
they support extreme parallelism

typical operating system components



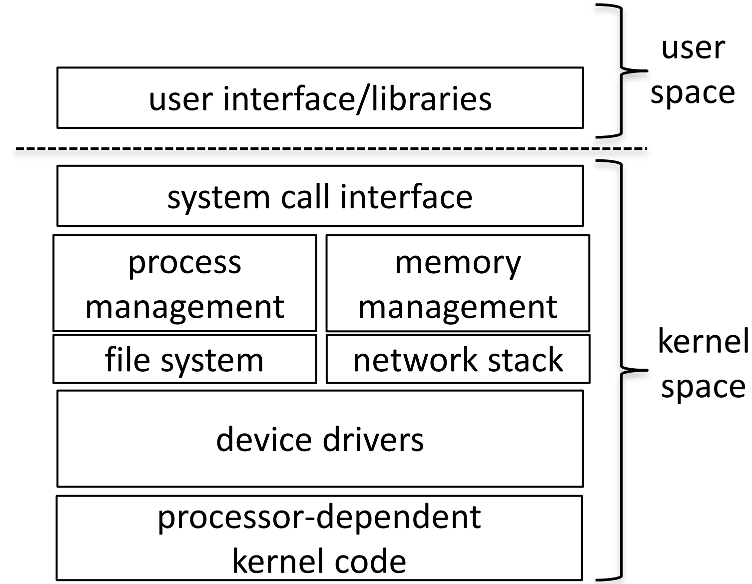
disclaimer: mainly based on Linux

user interface/libraries

gives applications uniform & easy-to-use primitives to communicate with the kernel

thanks to these we don't have to express ourselves in assembly

- compilers (e.g., gcc)
- shells (e.g., bash)
- GNU libraries in general

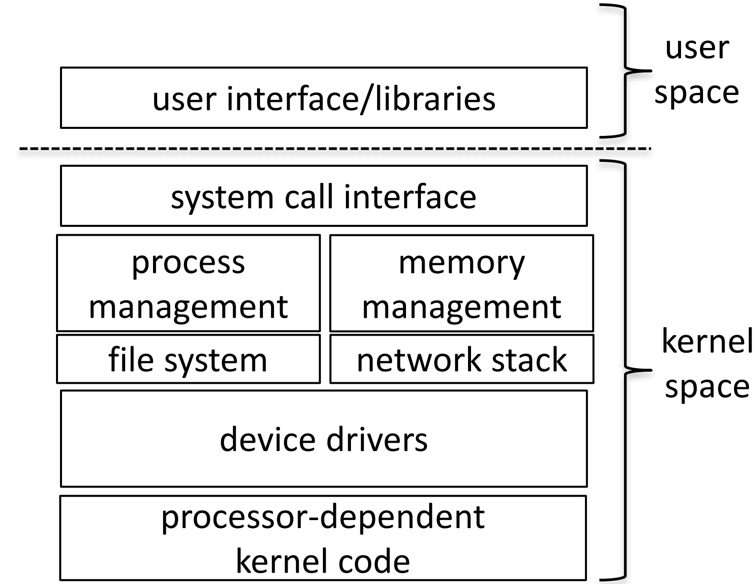


system call interface

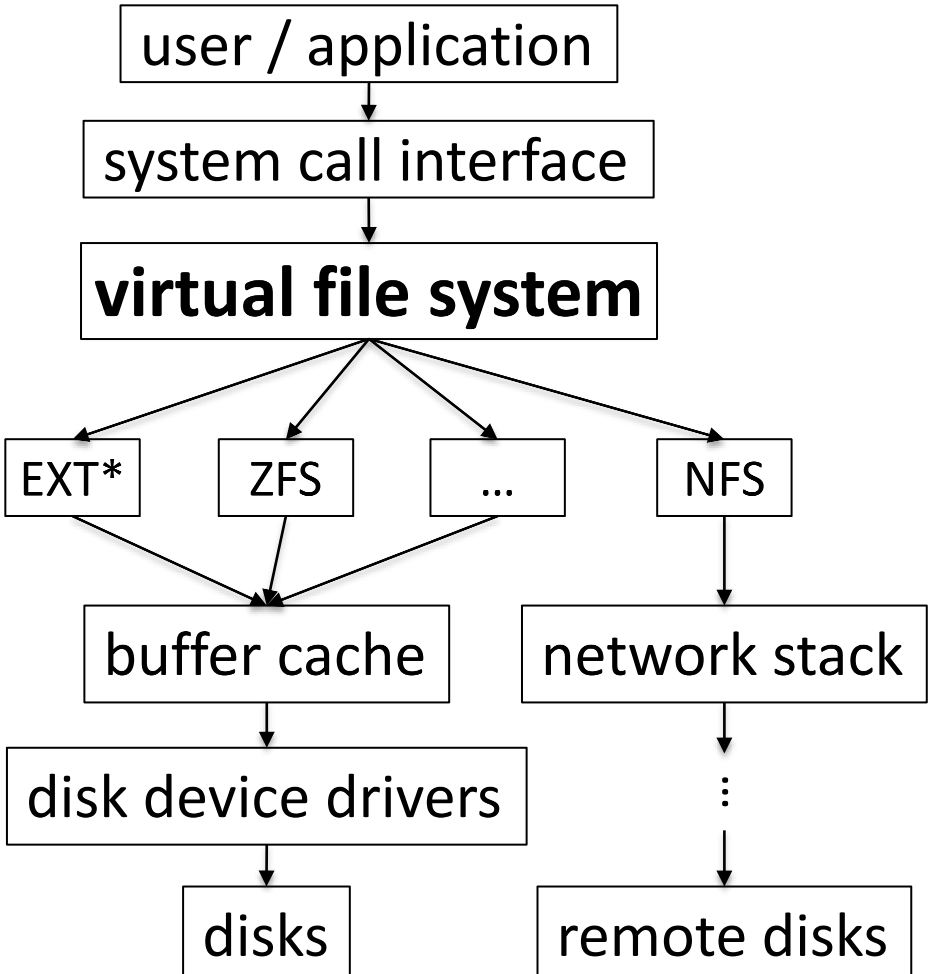
helps separating user space
from kernel space

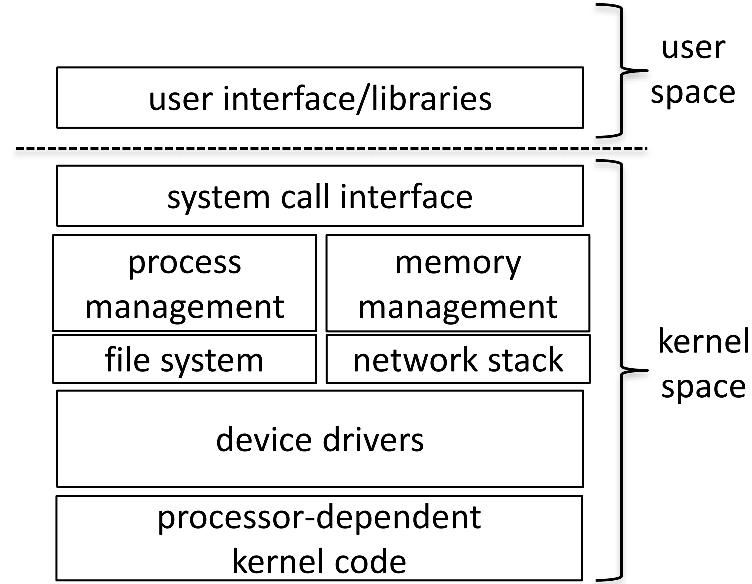
kernel talks to hardware since it
manages hardware resources
for many user requests

users just make request such as
reading/writing a file,
allocating more memory,
sending packets over network,
creating a new process, etc.



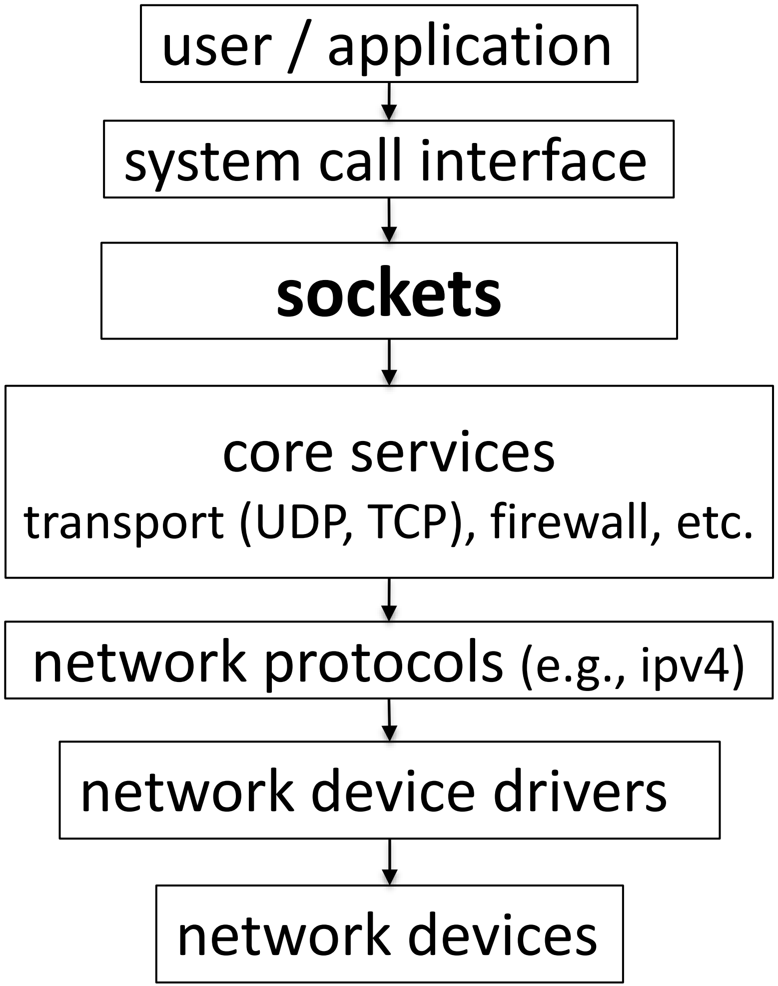
file system

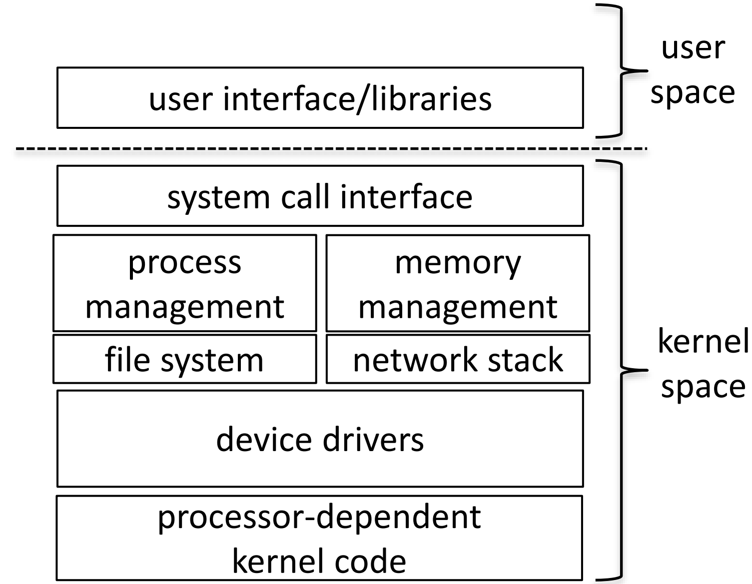




enables common interface to access different file systems

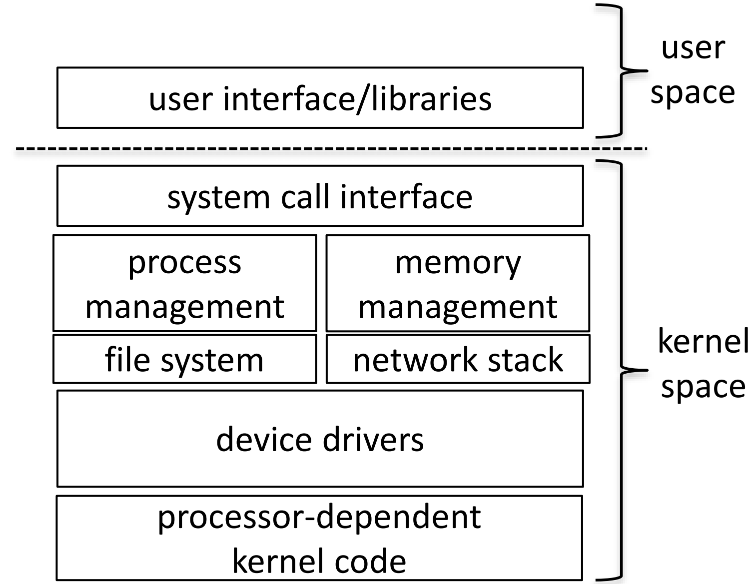
network stack





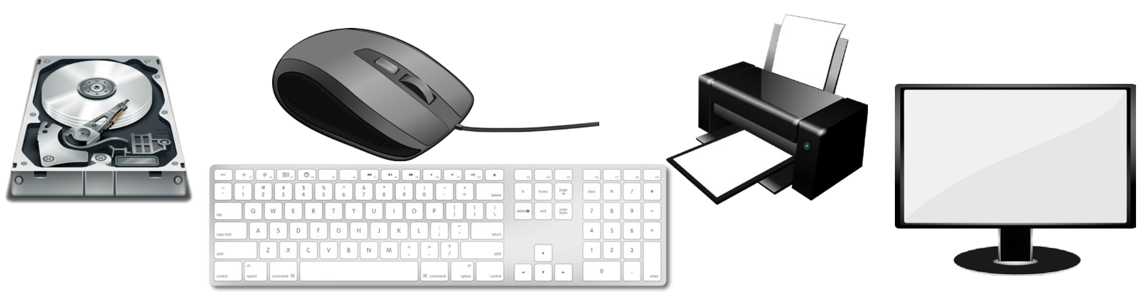
socket model enables a common interface to different protocols

device drivers

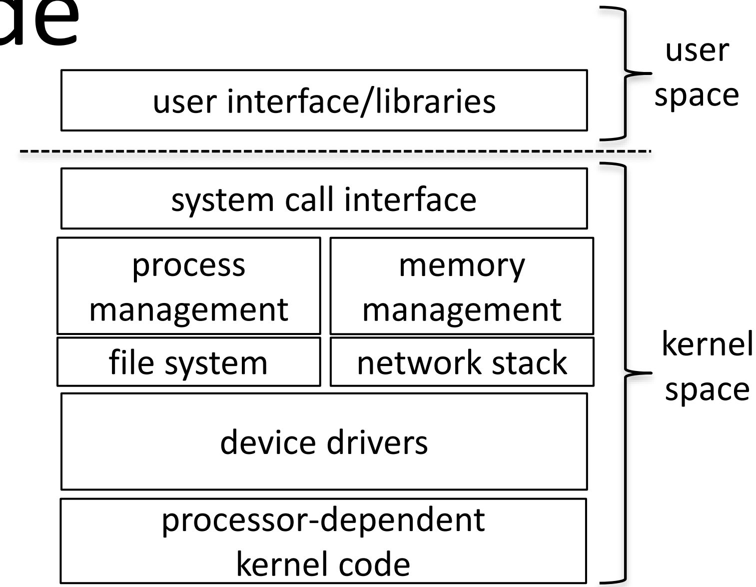


software interface to hardware devices

a bloated component of the kernel
- since it is both hardware dependent
- & operating system specific



processor-dependent code



- different processors support different instructions sets

- certain processors support functionalities like transactional memory, SIMD, memory alignment …

- desktop vs. server

need to be able to handle these differences without changing majority of the kernel code