# CMPUT391

# Hardware Basics and DBMS Applications

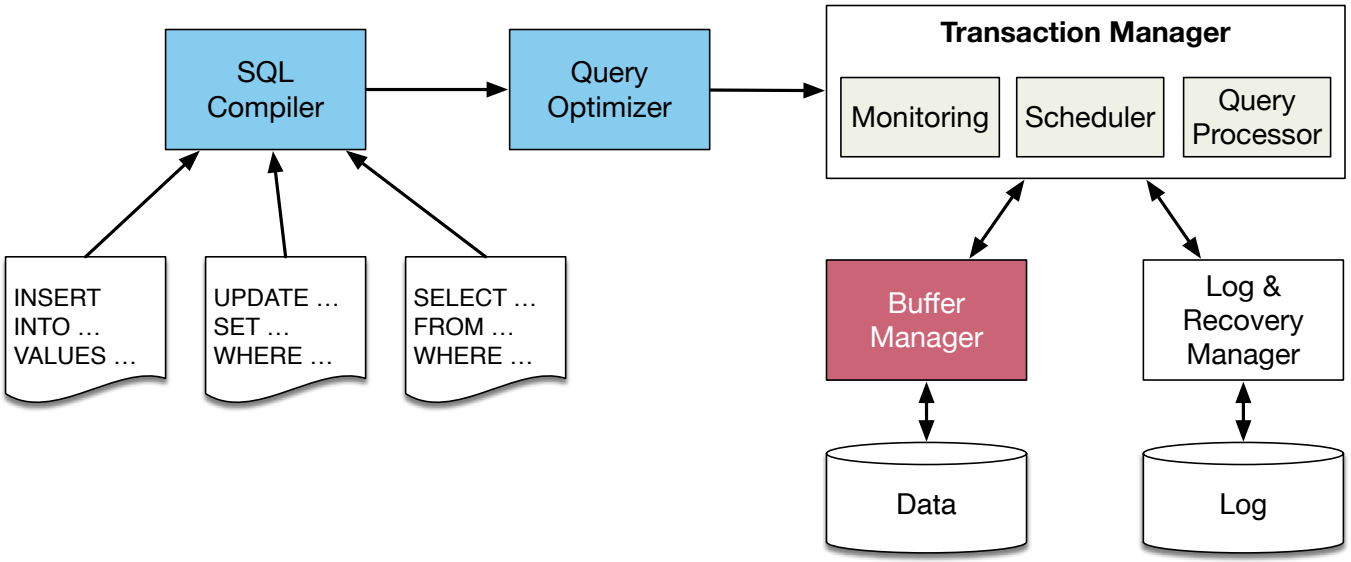
Instructor: Denilson Barbosa

University of Alberta

October 10, 2023

Slides by D. Barbosa, with suggested corrections and improvements by (in alphabetical order) C. Bins, D. Caminhas, K. Guhzva, Q. Lautischer, E. Macdonald, M. A. Nascimento, K. Newbury, M. Strobl, D. Sunderman, K. Wang, and K. Wong.

# Simplified DBMS architecture




These notes look into basic concepts from Computer Architecture (CMPUT229) and Operating Systems (CMPUT379) that relate to DBMSs and database applications.

Computer Architecture Basics

Virtual Memory

The role of the Operating System

DBMS Architecture Considerations

Architectures of Database **Applications**

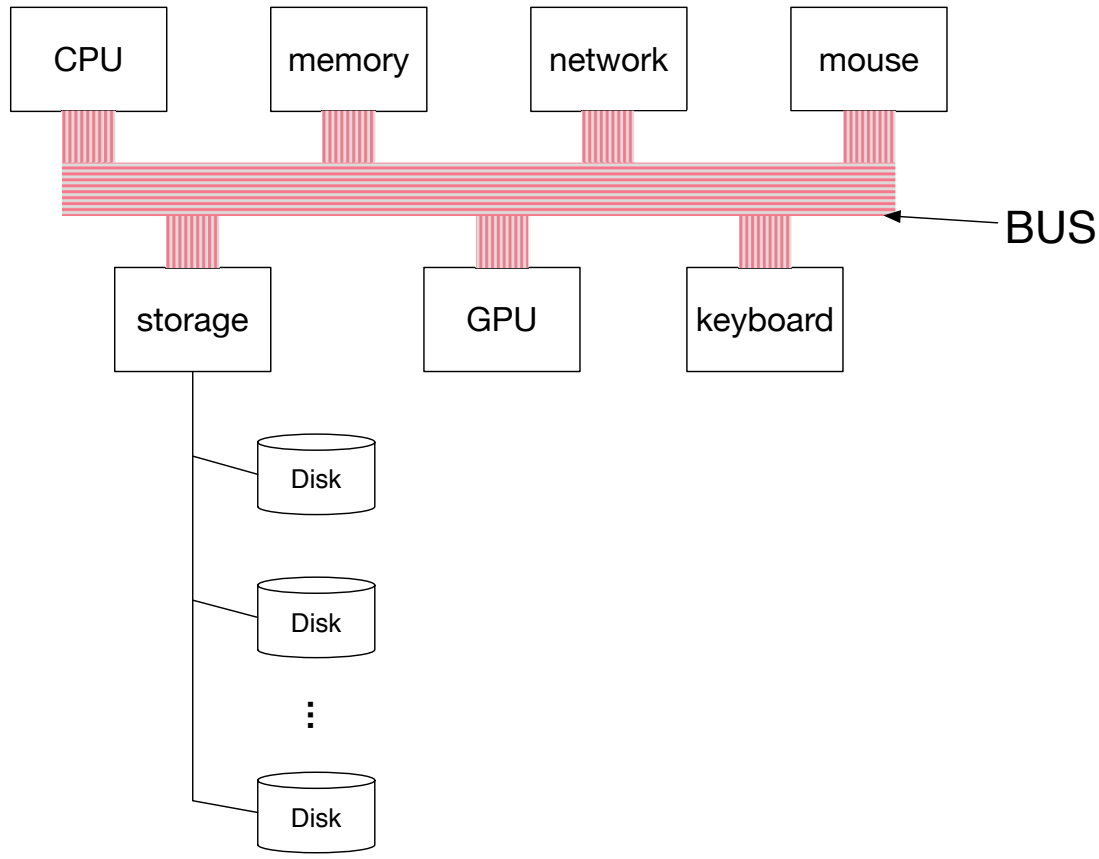
# Computer Architecture Basics

Virtually all computers today, from cell phones to servers, have the same internal architecture, introduced by John Von Neumann in the 1945[1].

Both programs and data are **stored in memory**.

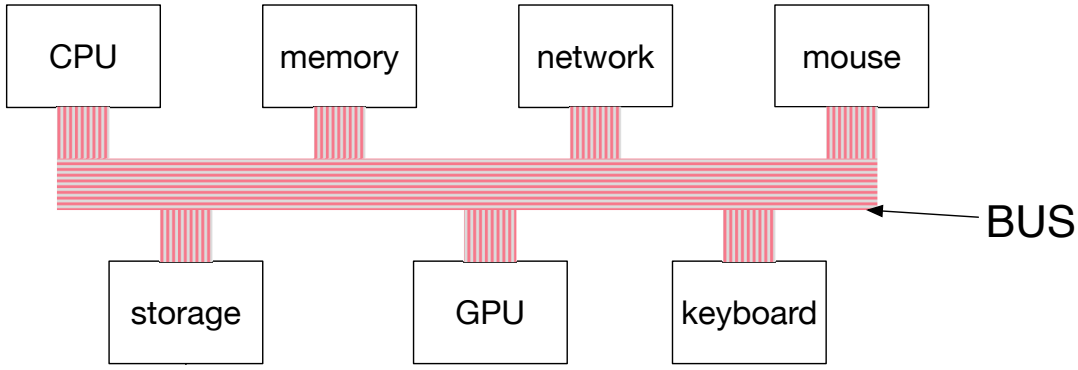
Programs are executed by the Central Processing Unit (CPU), one instruction at a time.

Constant move of code and data between memory and CPU.




[1] https://en.wikipedia.org/wiki/Von_Neumann_architecture

Components are interconnected by a Bus[2].



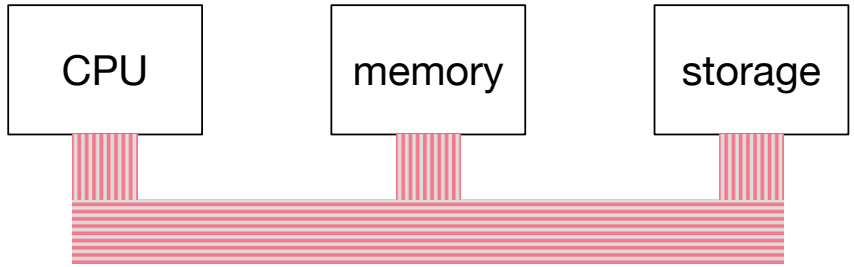

**The BUS must be shared by all components.**

It can only be used for **unidirectional** transfers between two components at a time (e.g., from memory to the CPU).

To maximize efficiency, transfers between **storage and memory** or **memory and CPU** are made in **fixed-sized batches** (called memory pages or disk blocks).

[2] `https://en.wikipedia.org/wiki/Bus_(computing)`

As far as the DBMS is concerned, these three components are the most important:




The CPU is where the DMBS code runs (e.g., where SQL queries are executed).

The storage is where the database is stored persistently (i.e., even when the computer is off).

Memory is where the data from storage resides before it can be used by the CPU.

However, the CPU does not access data in storage directly.

**Memory-mapped I/O:** access to data on files is done by mapping those files to buffers in memory.

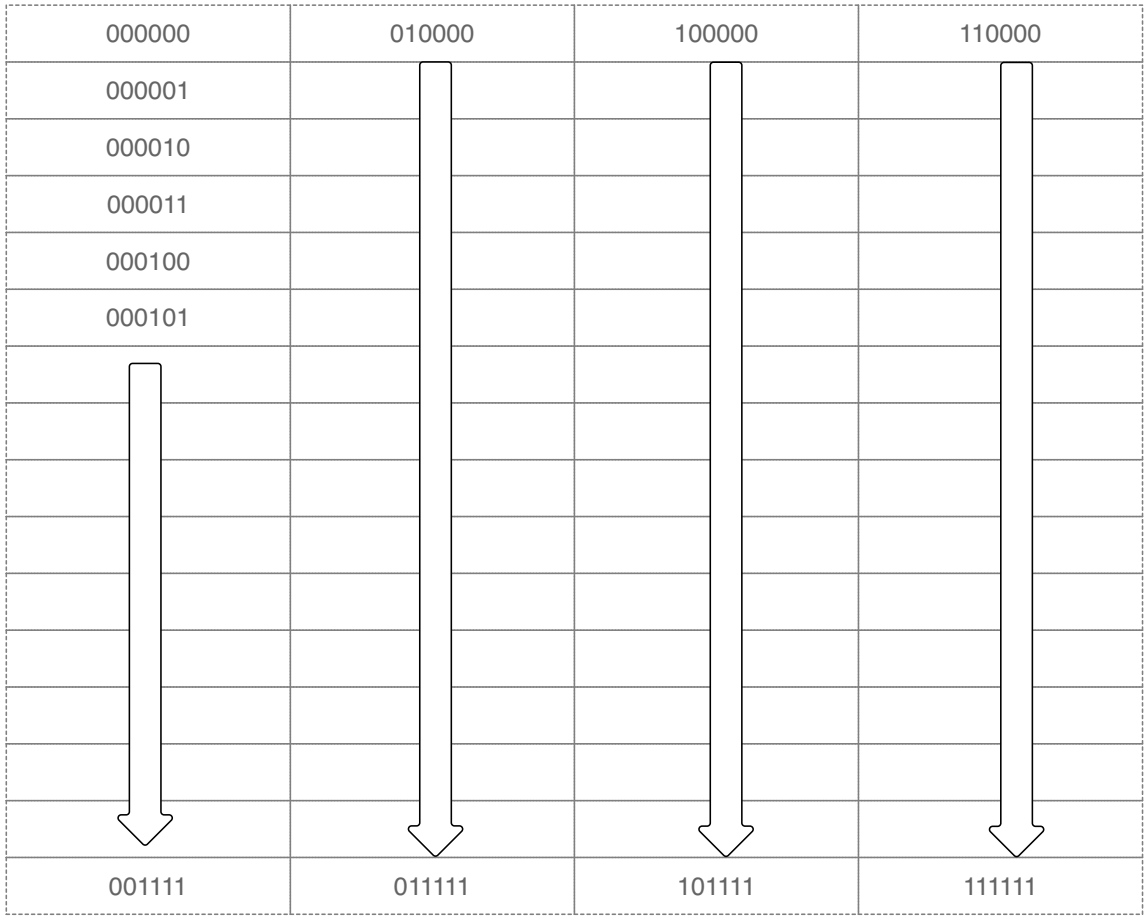

The memory is at the center of modern architectures, so we will look at how it works first.

Memory is organized as a grid of cells, each with a unique address.

The number of addresses depends on the computer. Typically, 32-bit computers have $2^{32}$ addresses, 64-bit computers have $2^{64}$ addresses, and so on.

Ex: computer with **6-bit** addresses, for a memory with $2^6 = 64$ cells.

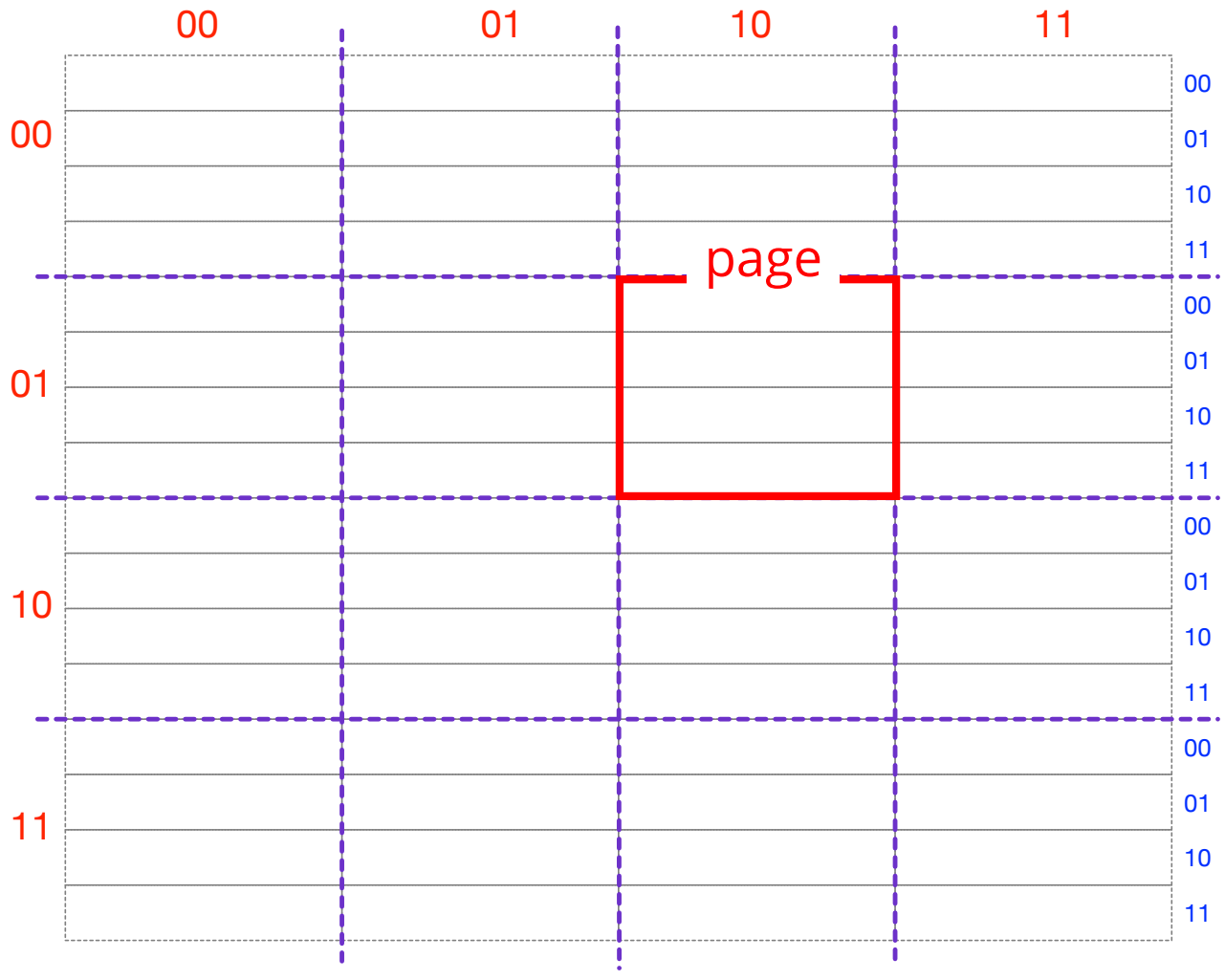
Each cell holds a fixed-sized **word** (typically 64 bits).



The bits in the memory address are divided into two parts. Some bits are used to indicate a **page of memory**, and the other bits are used to indicate an **offset** inside the page.

Ex: 4 bits for a page and 2 bits for the offset.

So, the memory in our example is divided into $2^4 = 16$ pages, each with $2^2 = 4$ cells.



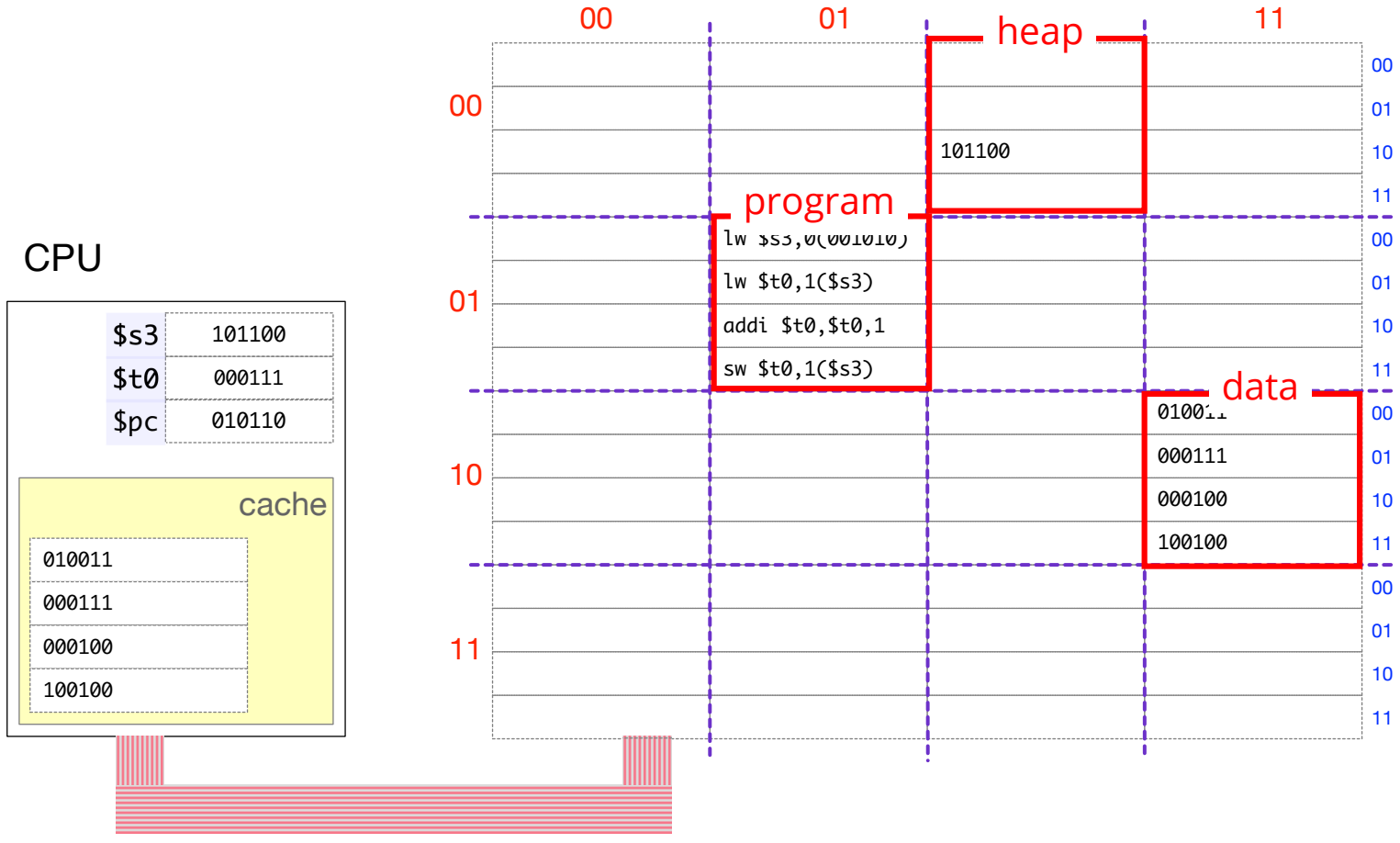
Consider a hypothetical C program:

```
data = (int*) malloc(20 * sizeof(int));

// initialize data array...

for (int i = 0; i < 20; i++){
    data[i] = data[i] + 1;
}
```

The program is compiled into binary code and stored into one or more disk blocks. When the program is executed, each block is brought to a page in memory.

Dynamically allocated data structures are stored on (possibly many) pages in memory as well.

The local variable `data` is represented in the program *heap* and contain the address in memory where the actual array is located.




Even though the memory sends entire pages to the CPU, the program instructions use individual memory cells.

A cache memory inside the CPU holds copies of pages in memory, from where the CPU can access the cells.

# The CPU

The CPU works in **cycles**, defined by internal ticking clock (1 GHz = 1 billion ticks per second).

Abstractly speaking, in each cycle, the CPU:
(1) fetches the **next**[3] instruction of the program, from memory
(2) decodes and executes the instruction
(3) increments the program counter to the next instruction

**Program vs CPU instructions**

Each instruction in a program in a high-level language (e.g., a `printf()` statement) gets compiled into many (sometimes thousands) of CPU instructions.

[3] The **program counter** ($pc) register keeps the address of that instruction.

The registers (e.g., `$s3`, `$pc`) are circuits inside the CPU that can hold the content of a **single word** in memory.

Arithmetic (and other) operations values stored in registers:
- ex: `addi $t0,$t0,1` adds a constant (1) to the value in register `$t0`
- the results of these operations are written back to registers.

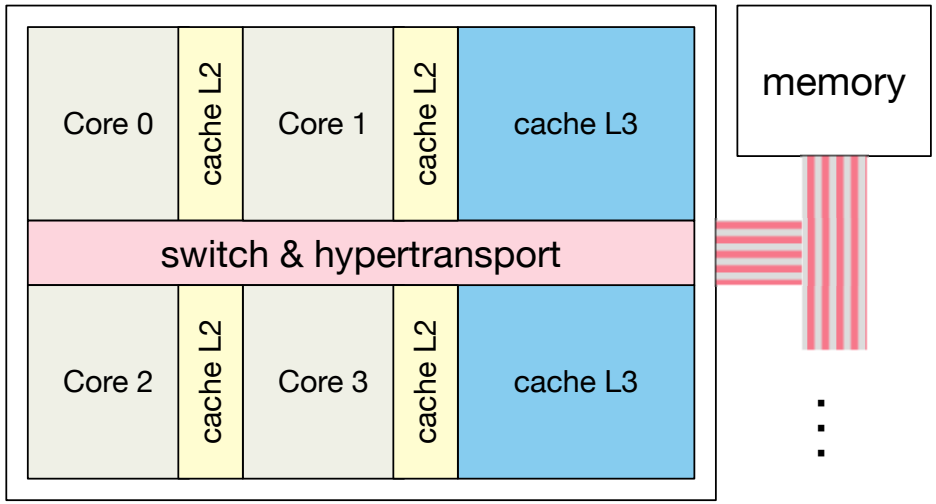
**The CPU does not manipulate memory cells directly**

instead it must move data from memory into registers (via the cache memory) and back:
- ex: **load a word from memory**: `lw <register>,<address>`
- ex: **store a word into memory** : `sw <register>, <address>`

# Multi-core CPUs

Modern CPUs have many (typically, between 2 and 32) independent processors (or **cores**).

Each core has its own private (L2) cache, but can also use a larger shared (L3) cache.




Multiple programs can run at the same time, on different cores.

Moreover, the same program can be broken into multiple processes (also called *threads*) which can run **at the same time** on different cores.

# Volatile and persistent memory

## Volatile Memory

Memory that needs continuous power to keep its contents. Also called *transient* memory.

Ex: registers, caches, RAM

## Persistent Memory

Memory that keeps its contents even without continuous power.
Also known as *storage*.

Ex: HDDs and SSDs

The "main memory" refers to RAM (Random Access Memory) circuits that are much larger (and much slower) than the CPU caches.

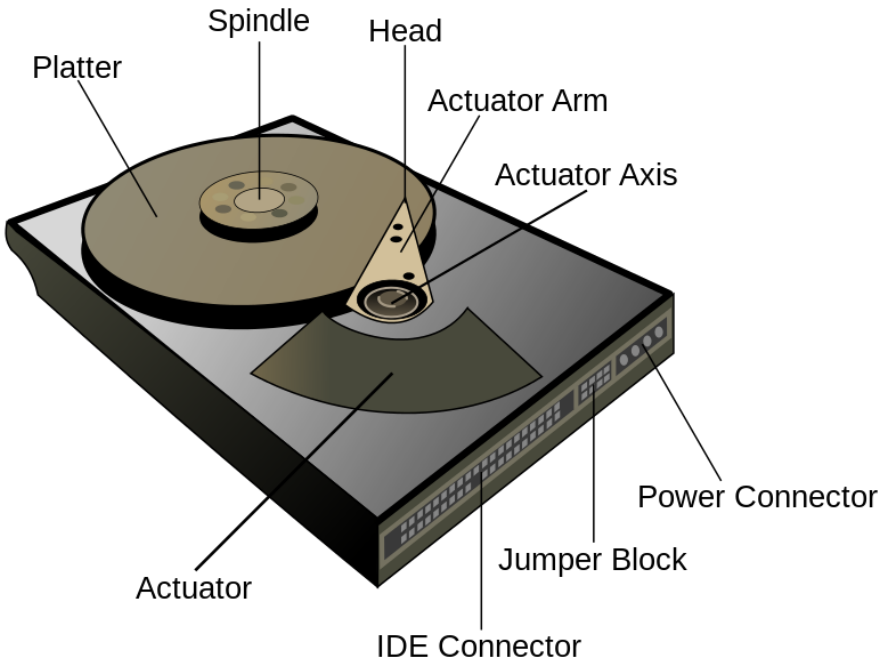
# Compared to main memory... storage is very slow...

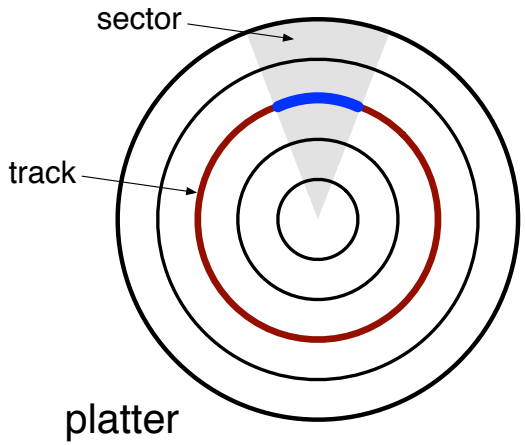
Many reasons:

- Reliable media that is (1) rewritable, (2) persistent and (3) affordable is also slow (compared to even DRAM).
- **Error-correction**: to prevent data loss, devices check everything that has been read or written against error correction codes (e.g., hash values), which takes time.
- **Synchronization**: data can only be transferred between the device and memory when the data BUS is not in use.
- Some devices have mechanical parts that are orders of magnitude slower than electronic components.

Hard Disk Drives (HDD) store data on the surface of spinning disks (platters). Many drives have 2 or more disks.

- data is read or written by a "head" traveling close to (but never touching) the platter
- data are stored on circular *tracks* on the platter
- with multiple platters, a *cylinder* is formed by all tracks at the same distance from the spindle
- each track/cylinder is divided into *sectors*



## HDD Seek Time

Time until the actuator arm moves to the desired track.
- *in expectation*: average time to travel between any two tracks;
- *typical*: between 3ms to 13ms

**Data locality**: the seek time between *adjacent* tracks is very short.

## HDD Rotational delay

Time until the desired sector arrives under the head.
- *in expectation*: time for a half-rotation of the disk;

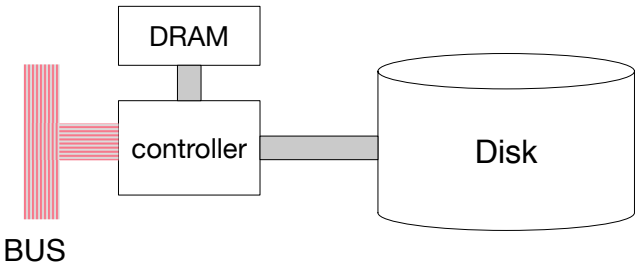
Practice: rotational delay of a 5400rpm disk?

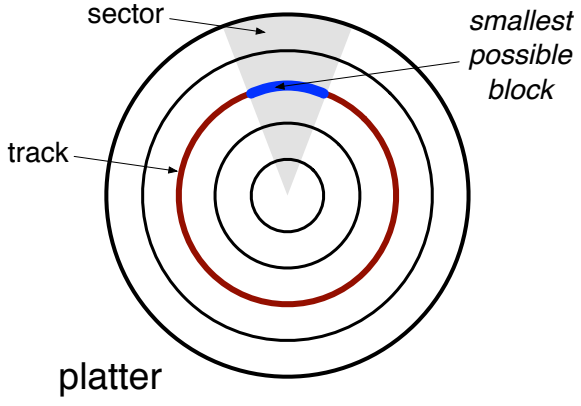
$$\frac{0.5r}{5400rpm/(60s/m)} = 5.6ms$$

# Data transfer between memory and storage

Data moves between memory and storage in **fixed-sized** chunks, called blocks.

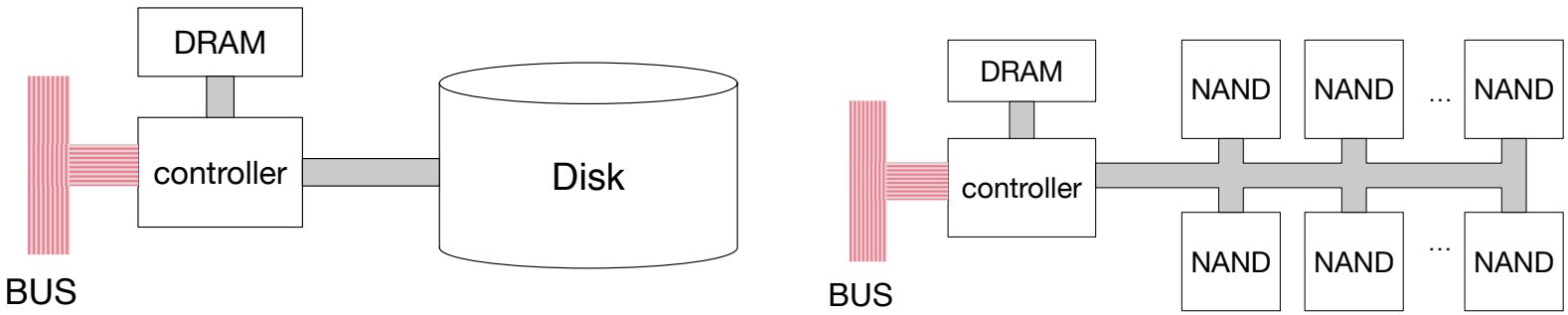
- the block size is a system parameter configured when the device is formatted
- the smallest possible block size with a HDD is a single sector;
- *typical* sizes are 4KB or 8KB.



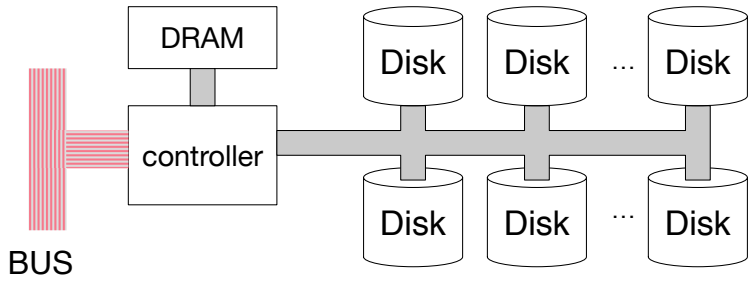

In a modern HDD, the controller has an internal, (volatile) DRAM cache, typically of a few MB, which it uses to mediate data transfers with main memory.

Solid State Drives (SSDs) have the same architecture as HDDs:




Even though they use a much faster medium (NAND) and have no mechanical parts, they are still block-oriented storage devices.

RAID (Redundant Arrays of Inexpensive Disks) group multiple disks into a single logical storage device:




Typical RAID settings[4] store the same (logical) block in more than one disk. This yields higher performance and error recovery:

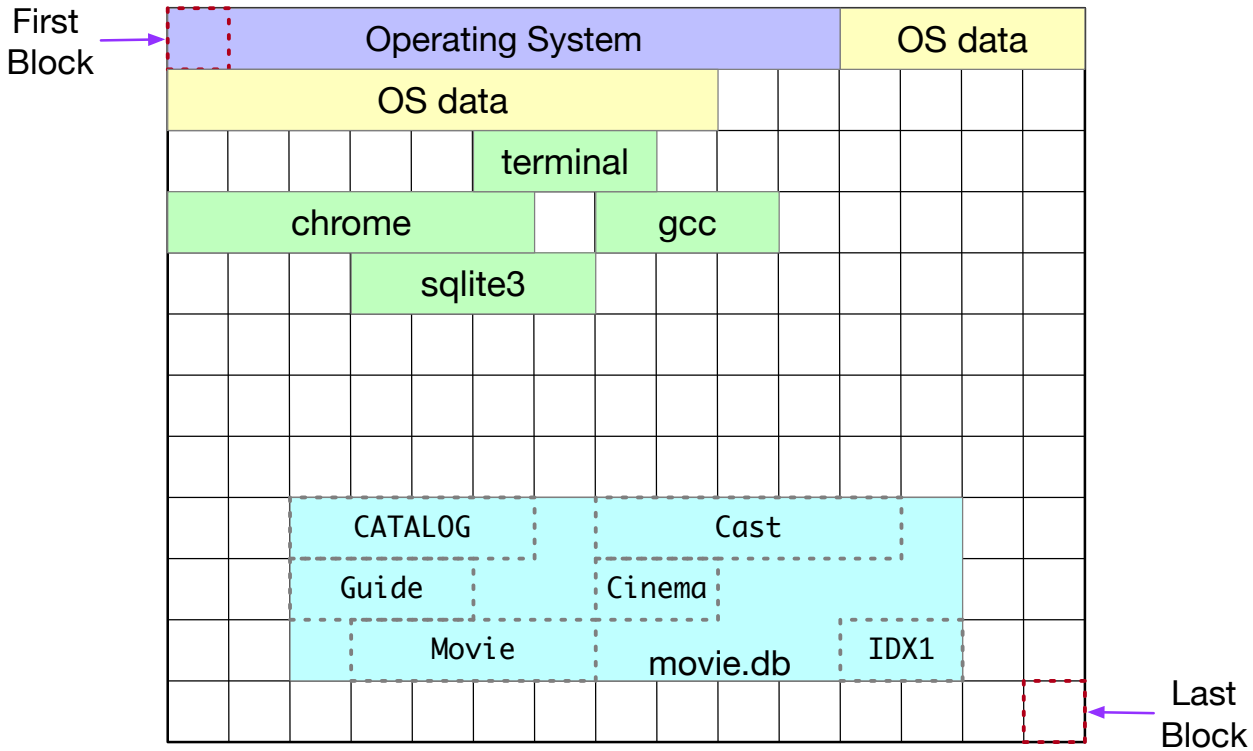
- Each block can be read from any disk containing it;
- If one disk crashes, its blocks can be read from another disk.

[4]There are many ways of configuring RAID systems.

# Storage is block oriented

The Operating System abstracts all storage devices (HDDs, SDDs, RAID arrays, etc.) as a collection of blocks:

- each block has an address;
- file (names) are associated to blocks by the OS



There are two main costs when using storage:

- **access time**: setting the device to read/write the desired block;
- **transfer time**: moving data between *primary memory* and the device's internal cache.[5]

## Access times

- For SSDs and RAID stores, the access time is negligible (and usually very hard to model)
- For HDDs, access times are:

**read**: average seek time + 0.5 rotational delay

**write**: average seek time + 1.5 rotational delay[6]

[5] The time to move data inside the device is usually ignored.

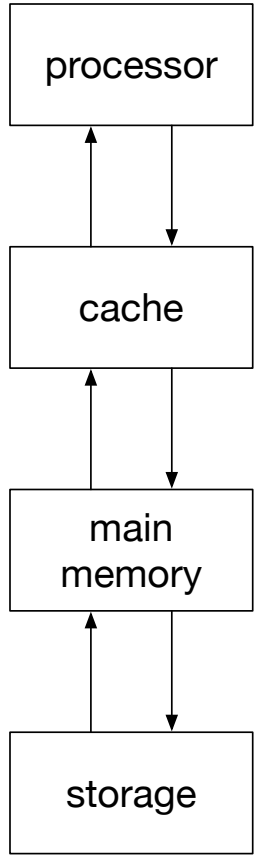
[6] To verify the data was correctly written.

# Memory Hierarchy

The “memory” of a computer is complex and layered sub-system comprising persistent storage, RAM, caches, and registers.

**Cost/Speed**

The closer to the CPU, the faster and more expensive the memory is.




| component | technology | access time (ns) | $/GB (2012) |
|---|---|---|---|
| cache | SRAM | 0.5–2.5 | 500–700 |
| main memory | DRAM | 50–70 | 10–20 |
| storage | flash | 5E3–5E4 | 0.75–1.00 |
| | disk | 5E6 – 2E7 | 0.05 – 0.10 |

[Patterson & Hennessy 2014]

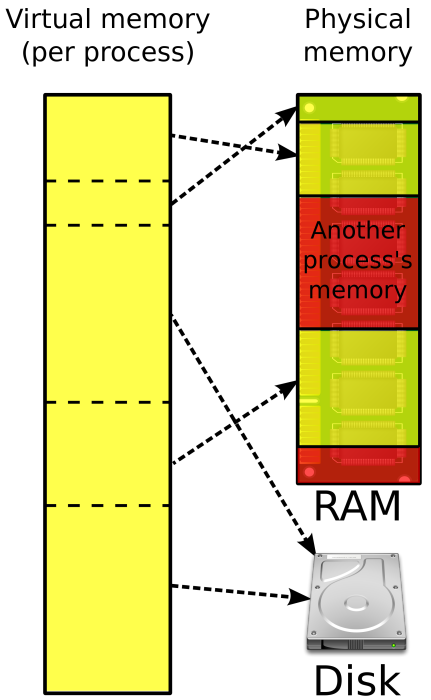
# Virtual Memory

# Virtual Memory

Virtual Memory[7] is a powerful programming metaphor that allows the efficient and secure simultaneous execution of multiple programs in the same computer.

Each program is allowed to use virtually every memory address possible in the computer (e.g., $2^{64}$ addresses in a 64-bit machine).

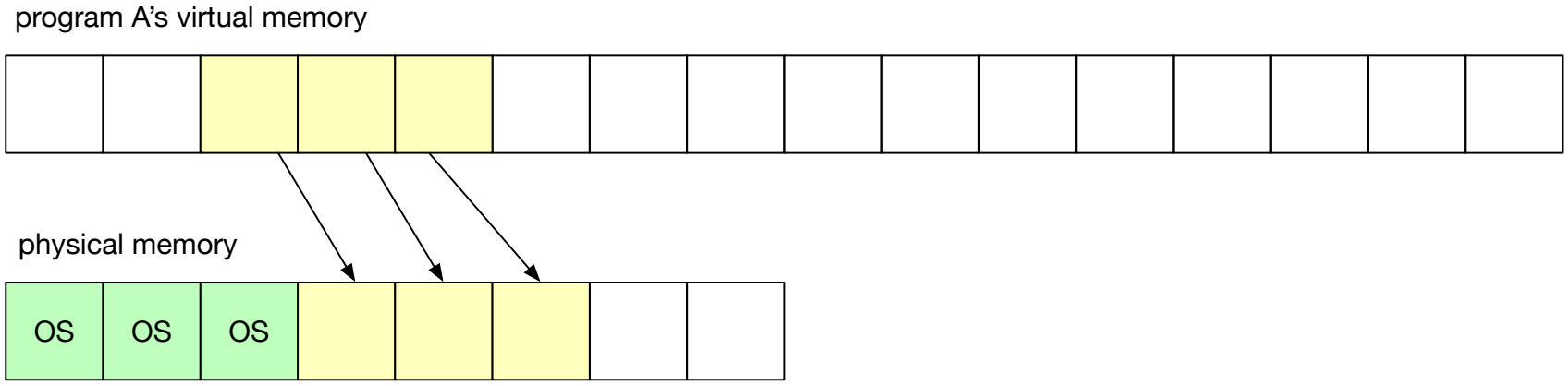
If the program requires more memory (pages) than is *physically available*, the operating system automatically moves some memory pages to disk, making room for the new requests.




Source: Wikipedia.

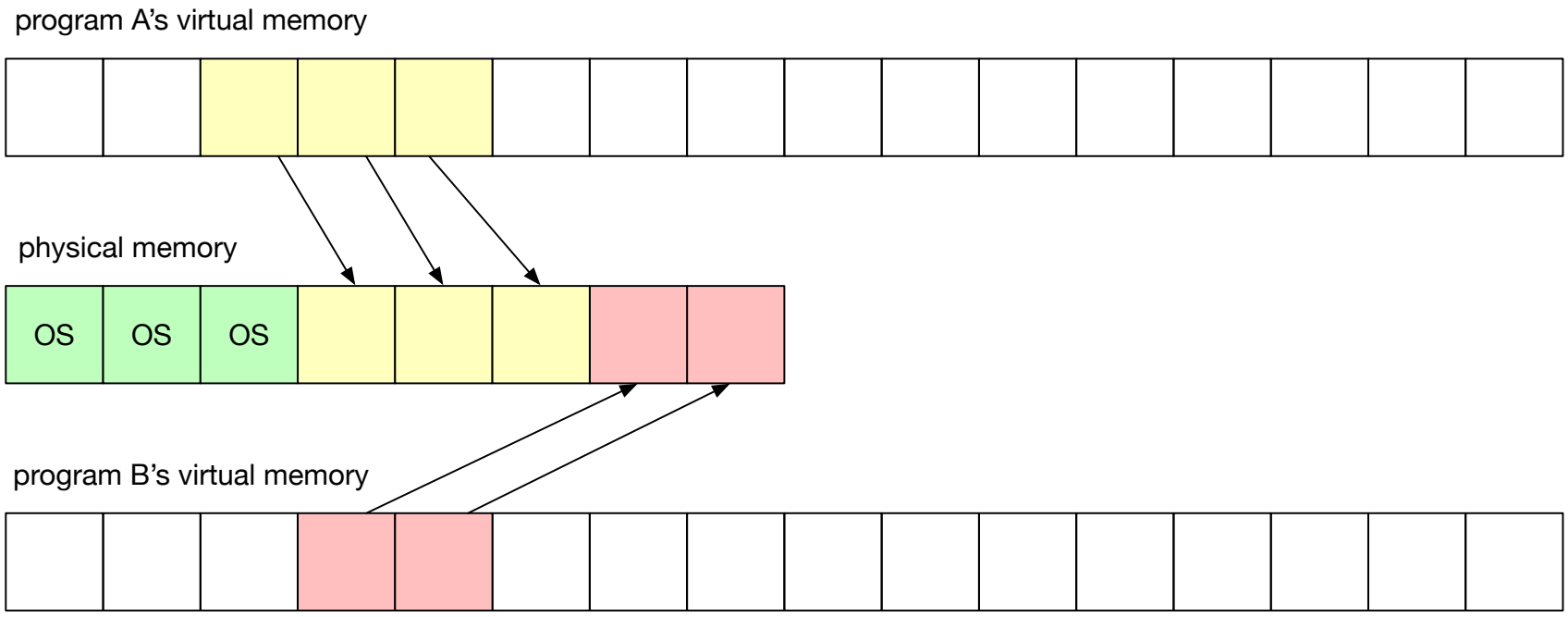
[7] `https://en.wikipedia.org/wiki/Virtual_memory`

Programs can use all possible memory addresses (e.g., $2^{64}$ addresses on a 64-bit machine). As a result, virtual memory can have more pages than what is available in physical memory.



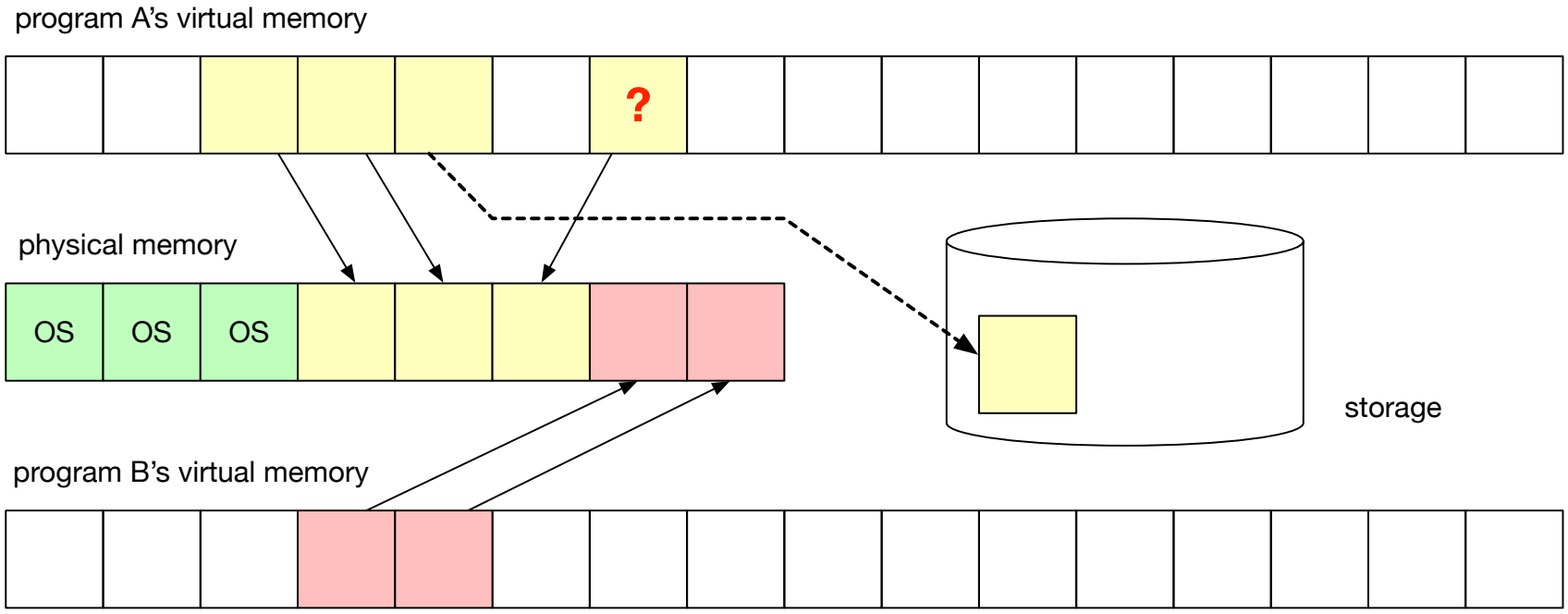

As the program requests more memory, the Operating System (OS) assigns physical pages to the virtual pages of the program.

Physical memory is shared by all programs in execution, including the OS itself.




Note that two different programs can have pages in the same virtual address, each mapped to a different physical page in main memory.

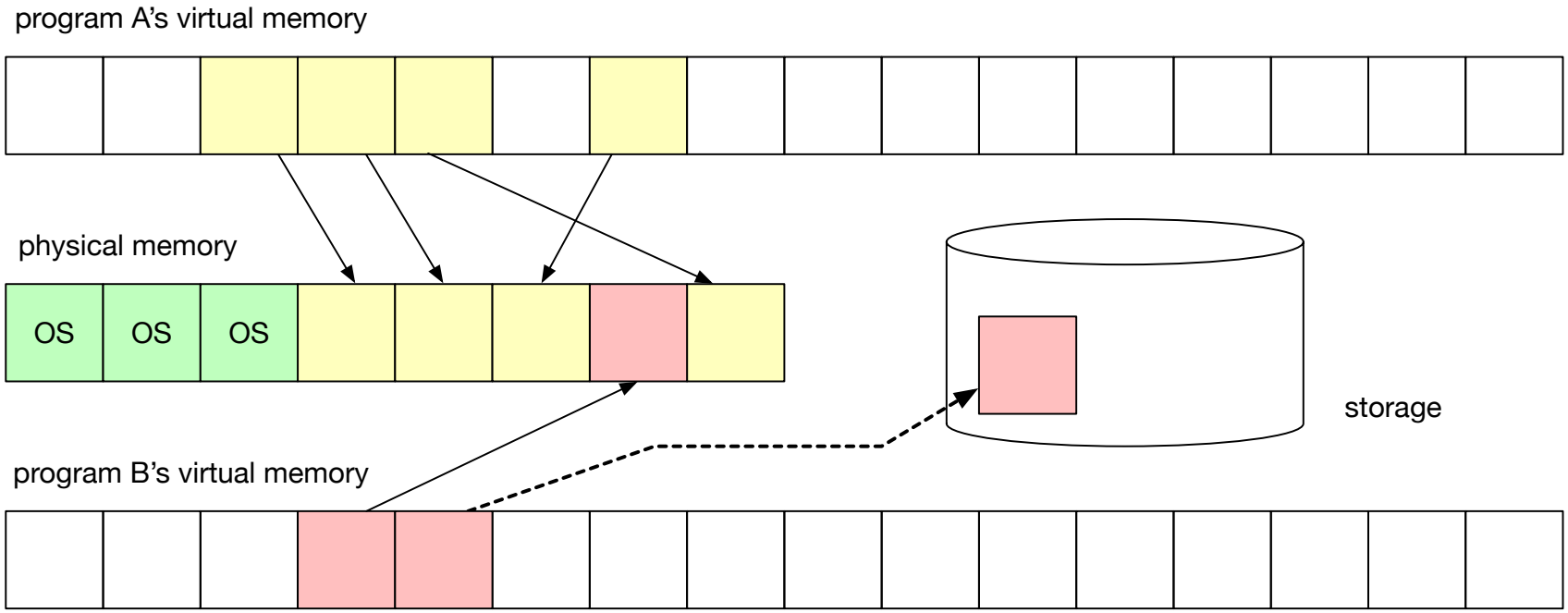
# What if a program needs another page but physical memory is full?




The OS takes a page from main memory and **swaps it out** to storage. This is easy to do because disk blocks and pages have the same size[8]

[8] Or one is a multiple of the other.

What if a program reads/writes a virtual page that has been swapped out?




The OS takes another page from main memory, **swaps it out** to storage, making room to **swap in** the page that was on disk.

Recap:

(1) Programs always use virtual addresses.
(2) The OS places virtual pages into physical pages in memory.
(3) The OS moves pages in and out of memory to keep the programs running.

Recall that memory addresses have 2 parts: $m$ bits for the page, $n$ bits for the offset.

Because virtual and physical pages have the same size, the "offsets" in the memory addresses remain unchanged, so all we need to do is to translate "page" portion of the address.

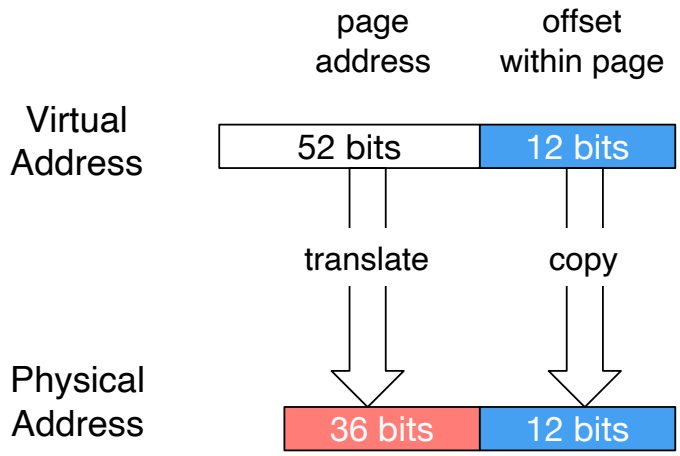
The “size” of virtual memory in a modern 64-bit CPU is 16 exabytes! No single computer with that much memory exists (and it is unlikely one will ever be built).

It is common for hardware manufacturers to use fewer bits for physical addresses, based on practical limits.

In any case, the translation of virtual to physical pages considers only the “page” part of the address.

Example of a modern AMD CPU:

- 64-bit virtual addresses (16 EB)
- 4KB pages (12 bit offsets)
- 48-bit real addresses (256 TB[9])




[9] Note that this is the maximum allowed.

# Implementing Virtual Memory

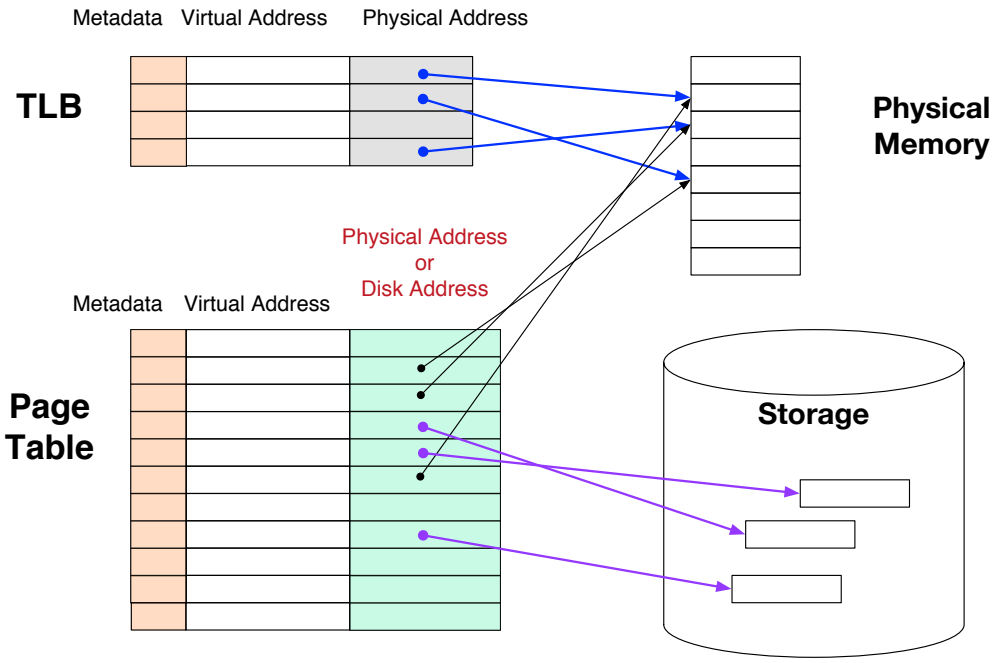
Translating virtual page addresses into real page addresses needs to be done fast, and is accomplished with the help of dedicated circuits.

## Virtual Memory Implementation

- the Operating Systems keeps a **Page Table** mapping virtual pages to real pages;
- the CPU keeps a *cache* of the table in special hardware called **Translation Lookaside Buffer (TLB)**
- every program has its own page table and TLB entries

**Q: Which page(s) should the OS swap out when needed?**

**A:** Pages that have not been used in a while would be best.




The **metadata** fields in the page table and the TLB keep this kind of information, which the Operating System uses to decide which page to evict.

# The role of the Operating System

**definition**

An **operating system (OS)** is system software that manages computer hardware, software resources, and provides common services for computer programs.[10]

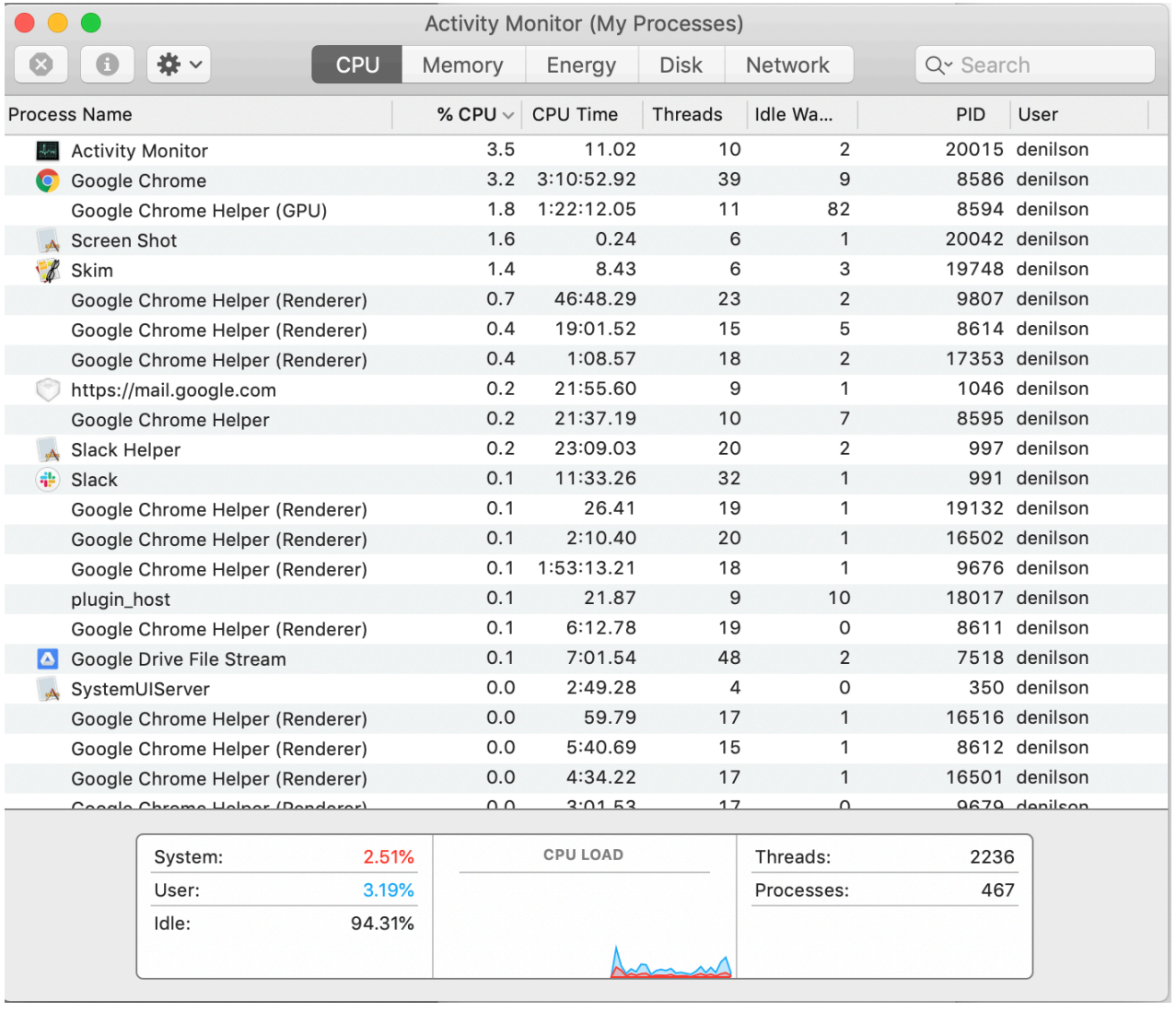
The OS is responsible for ensuring that applications don't interfere with one another and that all applications and users have a fair share of the computing resources.

To do that, the OS acts as the arbiter that gives and takes access to resources, as needed.

---

[10] `https://en.wikipedia.org/wiki/Operating_system`

In a modern multi-tasking OS, each application is executed by one or more **processes**.



Processes cannot interfere with one another, meaning:

- Each user process has its own (virtual) memory buffers and its own files.
- No process can read/write buffers (and files) “owned” by another user/application.

The OS, however, can swap (in/out) data memory buffers of any process as needed.

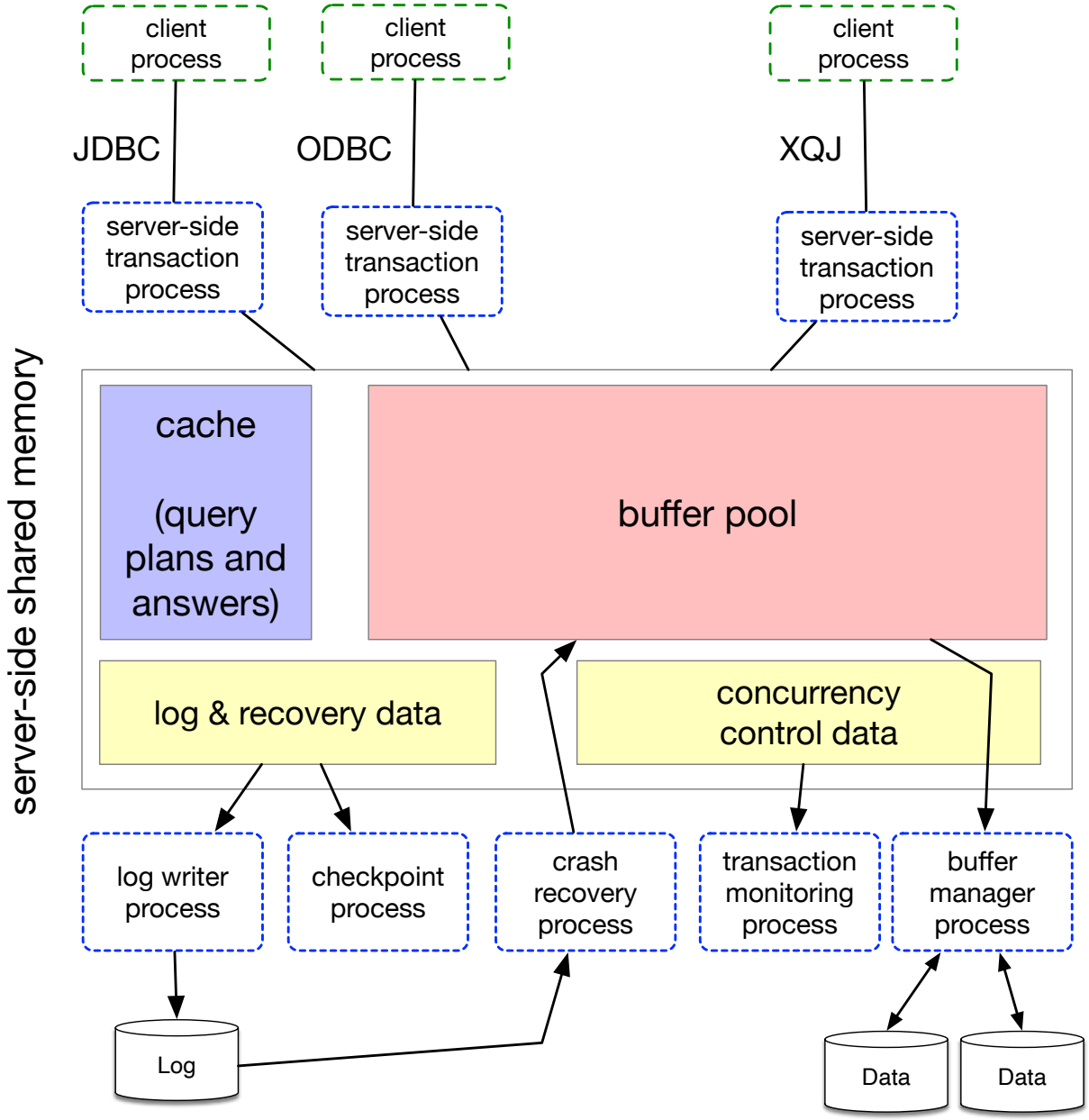
The OS gives the illusion of real time parallel execution of multiple processes by constantly swapping processes in/out of the CPU.

What does this have to do with databases?

A DBMS is a complex application, that is implemented by **many** processes:

- In some systems, each user query is executed by a separate OS process (normally owned by the same OS user).
- Often, there will be processes for each user connection in a multi-user system.
- At least, each large module of the DBMS is implemented by a different process (e.g., query processing, backups, logging, indexing, ...).

Over-simplified view of the various OS processes comprising a DBMS:



# DBMS Architecture Considerations

# DBMS Operational Requirements

The DBMS must meet these conflicting goals:

**Preserving data**

To **preserve the data**, the DBMS must make sure that it is safely stored in persistent storage[11].

**Using data**

Queries and updates are processed by the CPU!
- the data needs to reach the registers

As a result: data is always on the move.

[11] In fact, the privacy-conscious should note that it can be *very hard* to erase data that has been properly written into persistent storage.

# How can the DBMS use persistent storage?

**Option #1**: via a File System:[12]

- there are many good file systems out there;
- however, they are optimized to handle a *large number of small files*, logically organized into hierarchies (directories); databases typically have a *small number of large tables*.

**Option #2**: implementing its own I/O management:

- the Buffer Manager reads/writes directly from/to the device;
- requires integration with the Operating System kernel.[13]

Most DBMSs have their own I/O stack, and some require changing the Operating System to work.

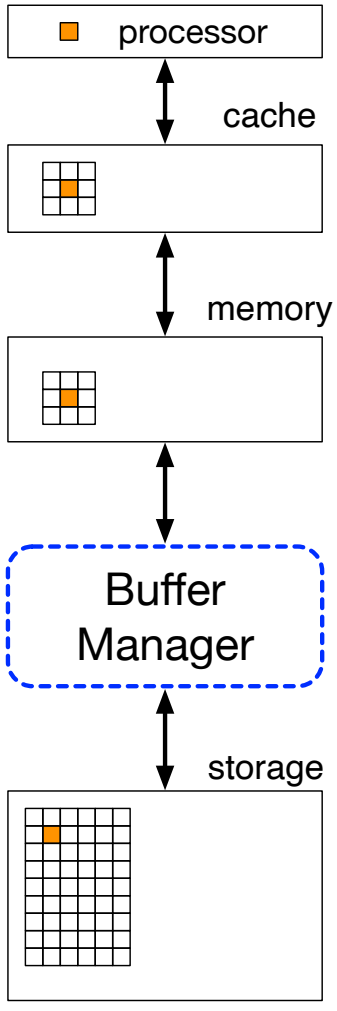
[12] https://en.wikipedia.org/wiki/File_system

[13] https://en.wikipedia.org/wiki/Kernel_(operating_system)

## The Buffer Manager

DBMS module responsible for moving data between storage and memory.

It splits main memory into logical units called *buffers*, which are often the same size of a memory page.




Before data becomes visible to the CPU, it must be *copied* in every level of the memory hierarchy!

Because I/O is so slow, a badly designed Buffer Manager can make the DBMS unusable.

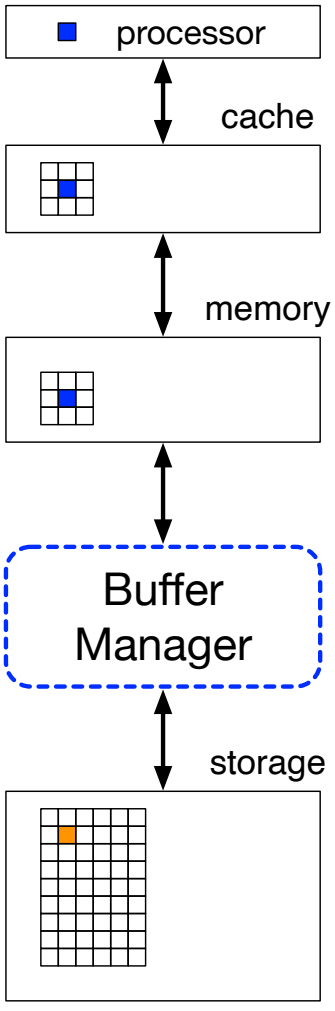
## Updates

When the processor changes an attribute of a tuple, the new value is modified in the cache and the memory first.

## Insertions

When a new tuple is inserted, it is stored in some buffer in memory first.
-if there is no room in any buffer, a *new one is created* in memory first.

Any buffer containing data not yet on the storage layer is said to be dirty.



## Dirty data sits in volatile memory

The Buffer Manager must *flush* dirty buffers to the storage layer for the changes to become persistent.

Trade-off:

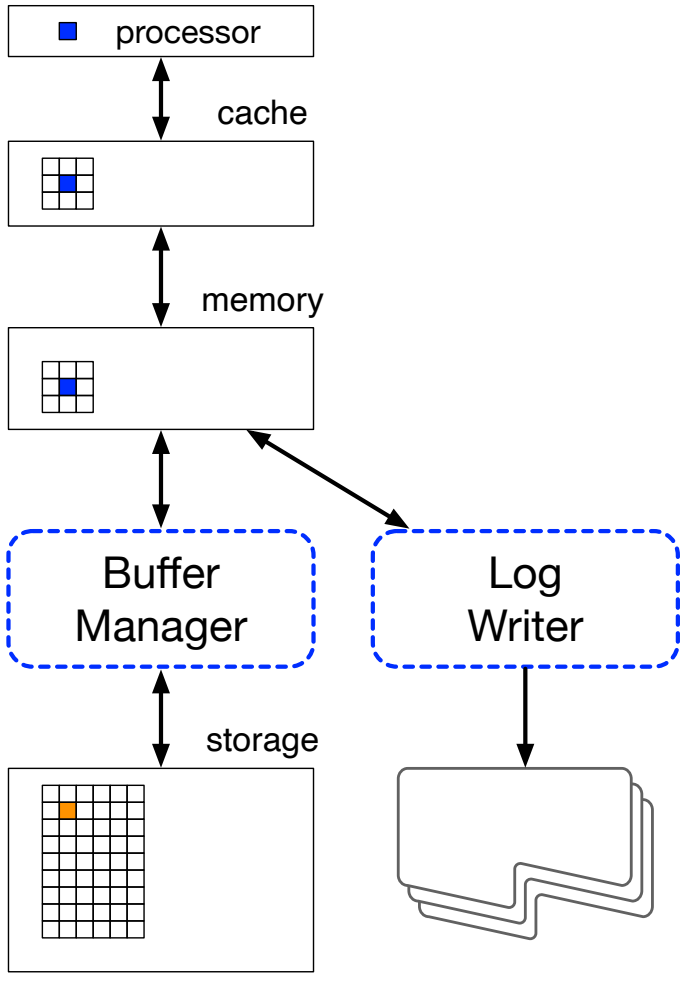
- Because storage is so slow, it is a bad idea to flush dirty buffers immediately after every insert/update.
- The changes to the database do not become permanent until they are written on persistent storage.

# Logging, Persistence and Recovery

The DBMS keeps a log of every modification (insertion, deletion, update) done to the database, *on a separate* file (or even a separate storage device).

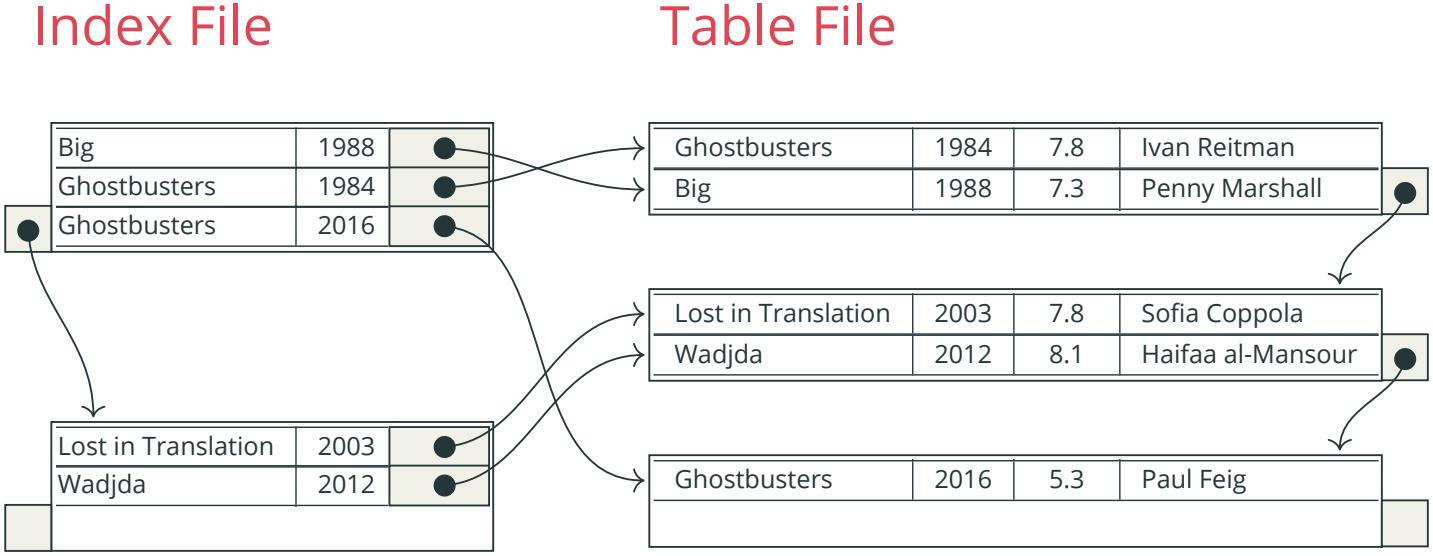
Dirty buffers can safely remain in memory as soon as the changes are written to the log.

If the computer crashes, all changes still in memory can be recovered from the log.



# Database “Files”?

Tables are collections of records stored on “files” (chains of fixed-sized blocks). Indexes help the DBMS find tuples through pointers.




A “database pointer” identifies the storage device *and* a block address which can be translated into a device-specific address (recall slide 21).

# Main Memory Databases

Some DBMSs (including SQLite) are optimized for main memory and use only virtual memory as its address space.

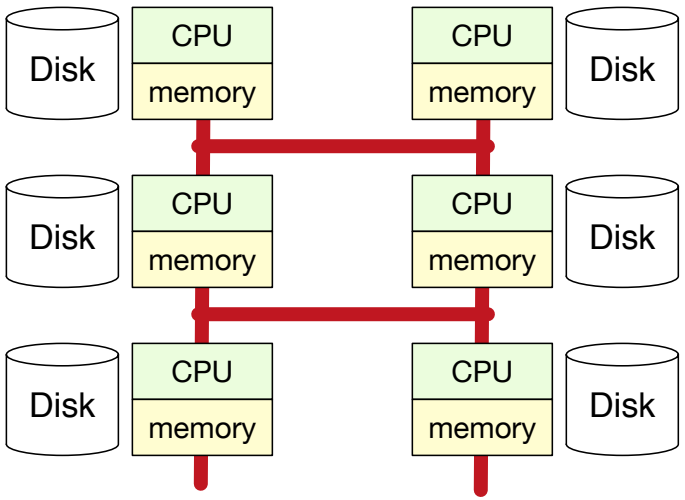
This is rarely a problem, as Virtual Memory can grow to Petabytes (recall Slide 30) with modern computers.

In any case, relying on virtual memory as the database address space simplifies the DBMS architecture a lot. For example “database” pointers can be simplified to virtual memory addresses.

Main-memory DBMSs are also optimized for single-user applications (like mobile applications and video games).

# What about bigger databases?

A very small number of applications (and organizations) deal with more than a Petabyte. In such cases, we need clusters of high-performance computers and (often) a custom DBMS.




In such cases, to identify a tuple we need to know:

- the specific block of a specific storage device holding the tuple
- the IP address of the server with that device (if using a cluster)

The standard DBMS implementations use **symbolic** pointers, even for single CPU DBMSs.

Symbolic addresses are typically much larger than a memory address. Also, symbolic addresses **must be resolved** into virtual memory addresses!

**Resolving pointers as blocks are loaded...**

When the Buffer Manager loads a symbolic pointer to memory, it must translate that symbolic address to an actual *virtual memory* address where the target of the link is actually loaded.

# Architectures of Database Applications

# Database Application Architecture

Multi-user applications are best developed as client-server applications.

**Client-Server applications**

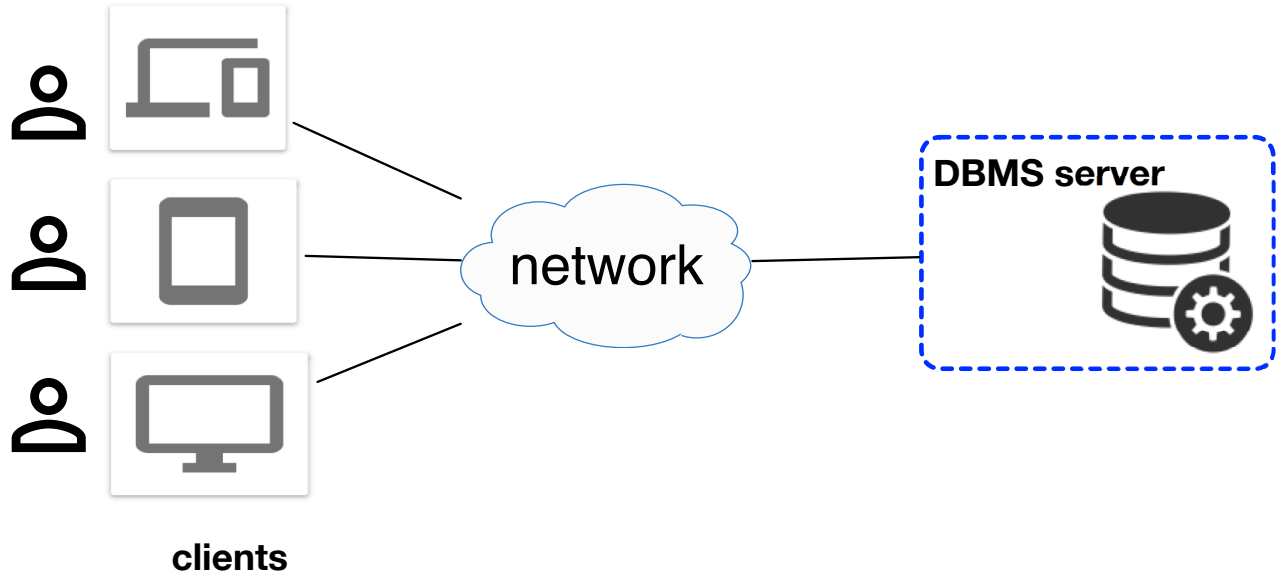
The DBMS (server) and the application (client) run as independent processes, often on different computers. There usually is one server for many clients.

Single-user applications (e.g., apps on mobile devices or some games) can be developed as a single process:

**Embedded DBMS applications**

The application and the DBMS are **compiled** together and run as a single process on the same device.

**Client-server** on separate computers:




**Example: Beartracks** (clients are Web apps on browsers)

**Connecting to the server**

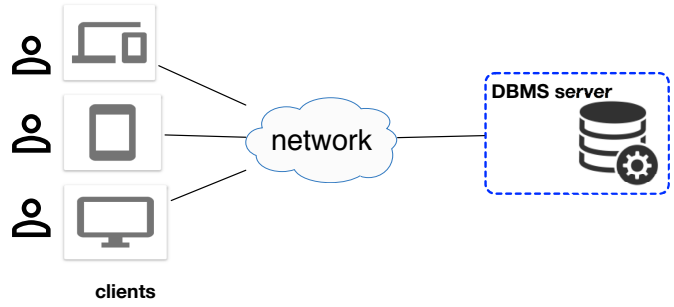
The client needs a *connector* API to talk to the DBMS.

ODBC[14] is widely used for this.

[14] https://en.wikipedia.org/wiki/Open_Database_Connectivity

With this architecture, almost all computation happens on the server.

The client presents results to and takes input from the users.




Other implications:

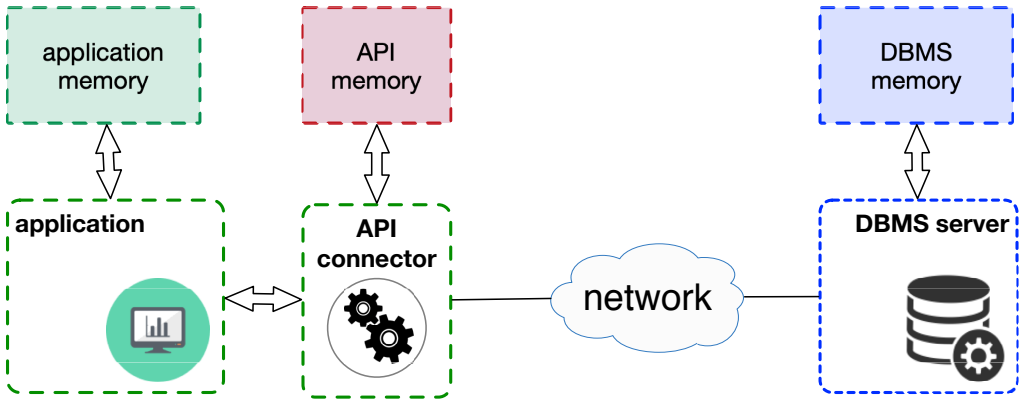
(1) scaling to more users means buying a bigger server

(2) all data input/output flows through the network

This model is best for centralized applications where the provider is responsible for the data (e.g., universities, banks, government agencies, etc.) **and** where Internet connectivity is a given.

# Client-Server with DBMS API

Application developers can use APIs to connect to the DBMS via the network.

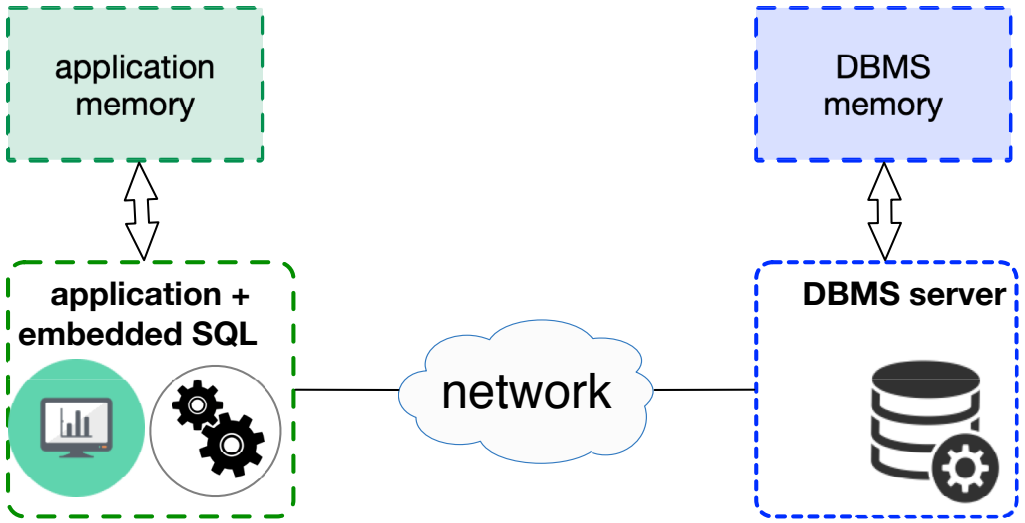



ODBC[15] (Open Database Connectivity) is an industry standard supported by many languages and DBMS vendors.

[15] https://en.wikipedia.org/wiki/Open_Database_Connectivity

# Client-Server with Embedded SQL

A more efficient way to deploy a client-server application is to **embed**[16] the libraries for connecting to the DBMS within the application itself.



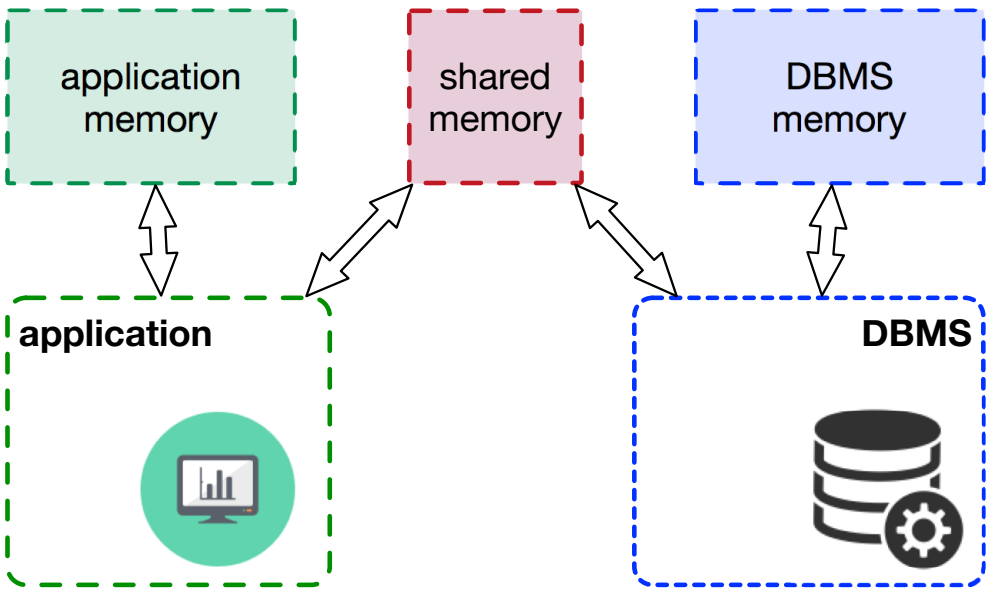

This is done by **compiling** the application and the libraries together.

[16] `https://en.wikipedia.org/wiki/Embedded_SQL`

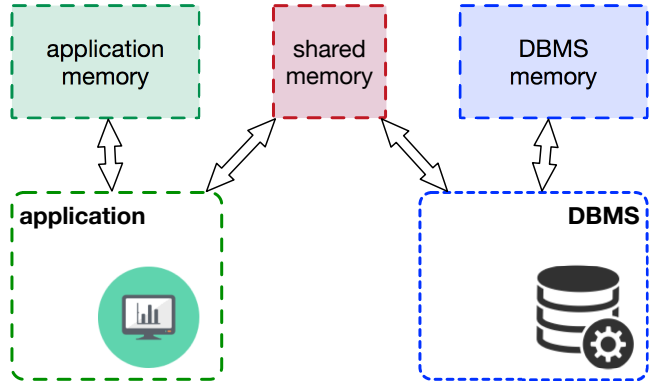
# Client-Server on the same device

DBMSs are being deployed more and more for **single-user** applications, e.g., mobile apps or games.

A simple (but inefficient) way to do that is to use the standard client-server architecture where both execute as separate processes on the same device:



What is so bad about this?




Data must be copied multiple times:

- from storage to DBMS buffers (and back)
- from DBMS buffers to shared pages in memory[17] (and back)
- from shared pages to the application (and back)

This is not only wasteful but also much slower:

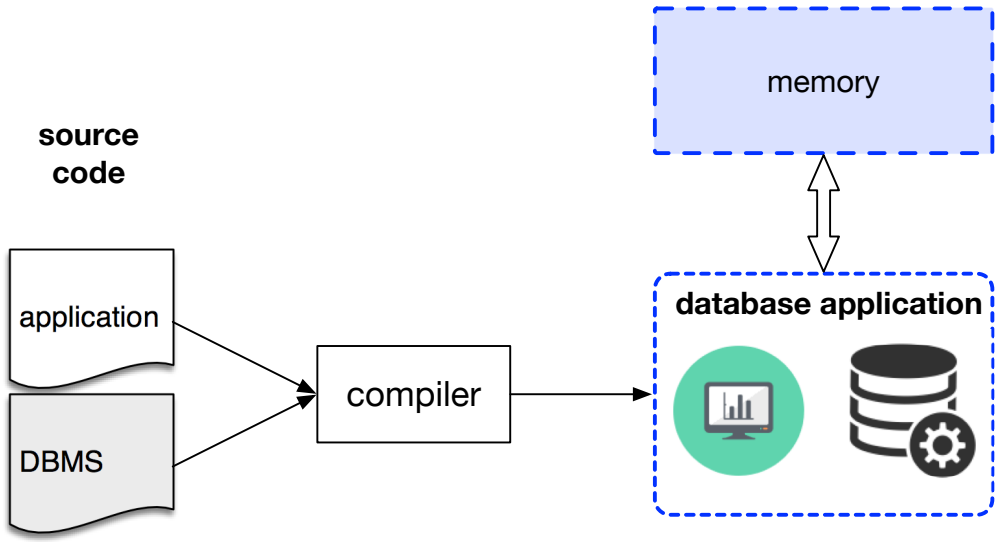
- even if the computer is *over-provisioned*

---

[17] One of the best ways of implementing inter-process communication… Read up `https://en.wikipedia.org/wiki/Inter-process_communication`

# Embedded DBMS

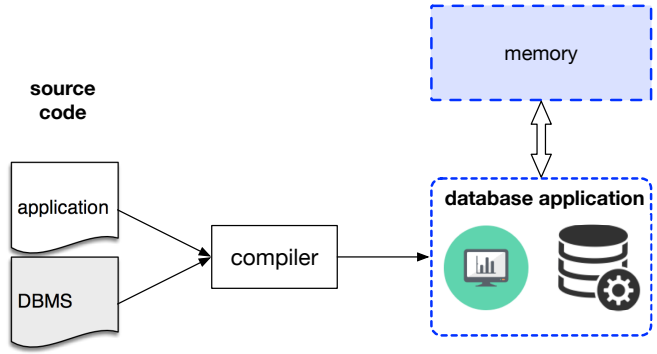
The right way of deploying single-user applications that need DBMS support is to **embed** the DBMS code inside the application:




SQLite was designed to be embedded in applications in ANSI C.[18]

[18] `https://sqlite.org/cintro.html`

What is so good about this?




No copying data around: “DBMS” buffers are easily accessible by the “application”.

Unused/unnecessary components of the DBMS are not compiled into the application, resulting in code with a smaller footprint.

Embedded data management is a strong trend with mobile apps, computer games, web browsers, etc.

# ASIDE: embedded SQL in the wild

Compiling SQL into native code?
- although still a research topic, this is not mad science!
- will be mainstream pretty soon.

Because of its small footprint, efficiency, main-memory option, and its public domain license, SQLite is probably the **most widely used** relational DBMS in the world today.

Modern MMORPG computer games comprise worlds with thousands of objects which are related in complex ways. Many of them already use relational DBMSs in a client-server mode, and some are moving to an embedded mode.

# Further references/resources

Other in-memory/main-memory databases besides SQLite:
`https://en.wikipedia.org/wiki/List_of_in-memory_databases`

PostgreSQL is an open-source, enterprise-scale client-server DBMSs:
`https://www.postgresql.org`

Apache Derby is an open-source relational DBMS written in and for Java.
It can be efficiently embedded inside Java applications:
`http://db.apache.org/derby/`