T04 System Design II

Performance, Concurrency, and Scalability

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

Tutorial outline

- **Part I: Lecture summary**
  - **Q&A for the lecture material**

- **Part II:** Programming basics

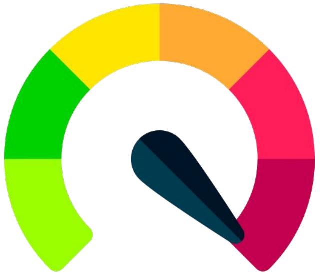
- **Part III:** Homework programming exercises (Artemis)

Lecture topics overview

- **Part I:** Performance
- **Part II:** Concurrency (or Scale Up!)
- **Part III:** Scalability (or Scale Out!)

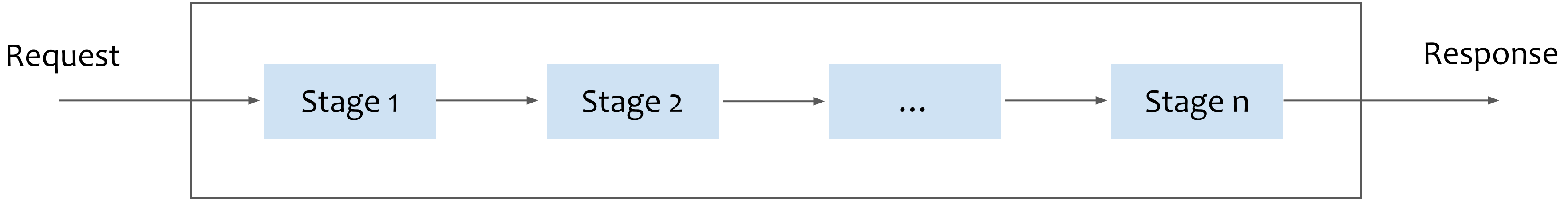
Performance matters

- **The need for performance**
  - The specification of a computer system typically includes explicit (or implicit) performance goals
- **Performance metrics:**
  - **Latency:** The time interval between a user's request and the system response
  - **Throughput:** Number of work units done (or requests served) per time unit
  - **Utilization:** The percentage of capacity used to serve a given workload of requests
- **Service level agreements (SLAs):**
  - An SLA is an agreement between provider (cloud software) and client (or users) about measurable metrics, e.g., performance metrics



Metric: Latency

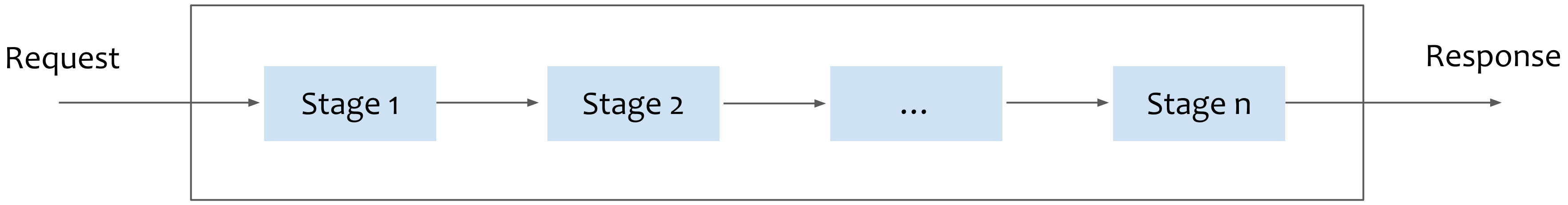
- **Latency** is the delay between a change at the input to a system and the corresponding change at its output
- From **the client-server perspective**, the latency of a request is the time from issuing the request until the time the response is received
  - Sending message + processing the request + Response returned



Latency (Stage-A+B) >= Latency (Stage-A) + Latency (Stage-B)

Metric: Throughput

- **Throughput** is a measure of the rate of useful work done by a service for some given workload of requests
  - *E.g., a key-value store (KVS) achieves a throughput of 160 million Operation per Seconds (OPS) on a single server*
    - *Operations for a KVS: Get/put*



Throughput (Stage-A+B) <= minimum(Throughput-Stage A, Throughput-Stage B)

An iterative approach for improving performance

- **Measure the system:** If performance enhancement is needed!
  - If yes, identify the performance metrics, e.g., latency / throughput
- **Measure again:** To identify the performance bottleneck w.r.t. the chosen metric
- **Predict the impact**
- **Measure the new implementation** to verify the change effectiveness
- **Iterate**

- **Useful tools:**
  - Linux Perf: performance analyzing tool in Linux
  - Java profiler, GNU gprofng: application-level profilers
  - Flamegraphs: visualization of the profiler report

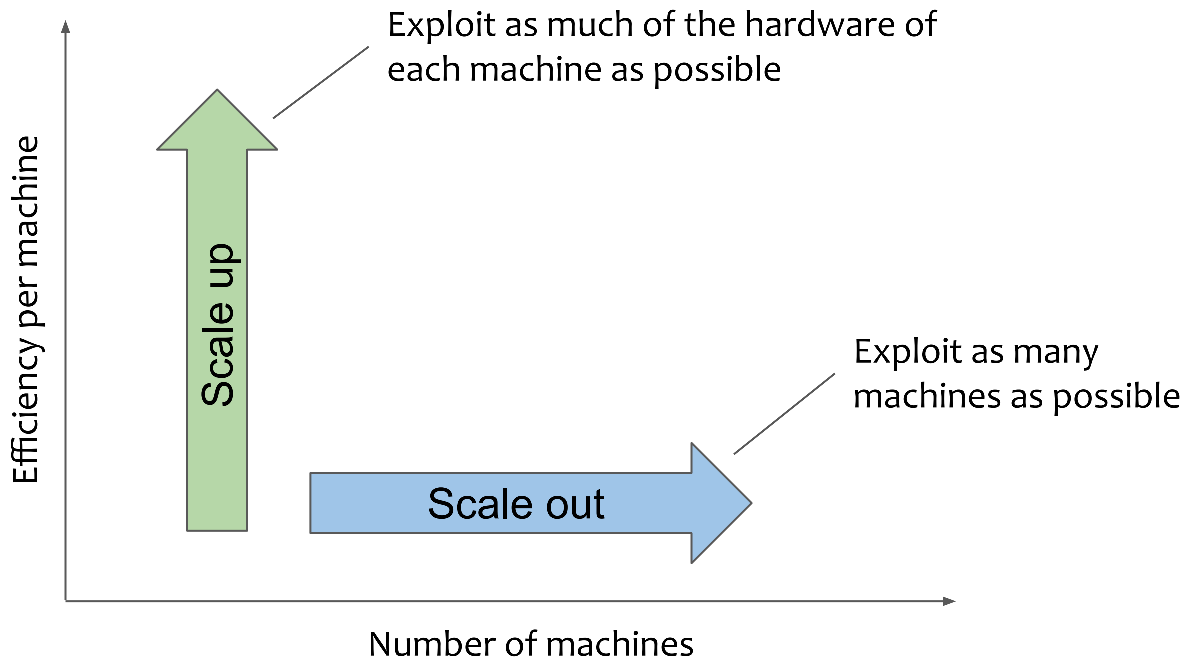
Directions for improving performance

- Resource splitting
  - Dedicated resources are usually faster, and the allocator’s behavior is more predictable
- Caching
  - Cache answers to expensive computations, rather than doing them over
- Compute in background
  - Asynchronous processing
- Batch processing
  - Process large volumes of data in batches, rather than processing them individually
- Parallelism
  - Divide a task into smaller sub-tasks that can be executed simultaneously

Lecture topics overview

- ~~**Part I:** Performance~~
- **Part II:** Concurrency (or Scale Up!)
- **Part III:** Scalability (or Scale Out!)

Scaling your applications



Modern processor architecture

Computation

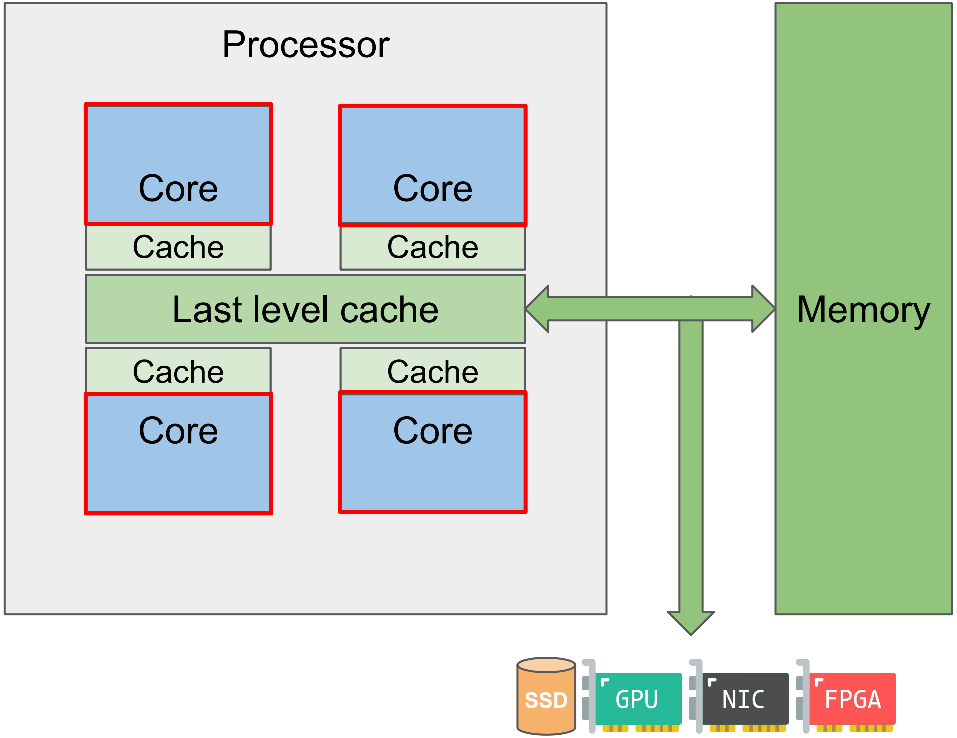
- Processing units (cores)
- Perform computations independently,
  i.e., execute programs

Memory

- Stores data needed by programs during their execution
- Cores have local caches to store recently used data closer to them
- A shared last level cache is the interface to "external" memory (RAM)

Devices

- Storage, network, GPUs, etc…



Thread model

A thread is a ***unit of execution*** *that can be scheduled on a core*

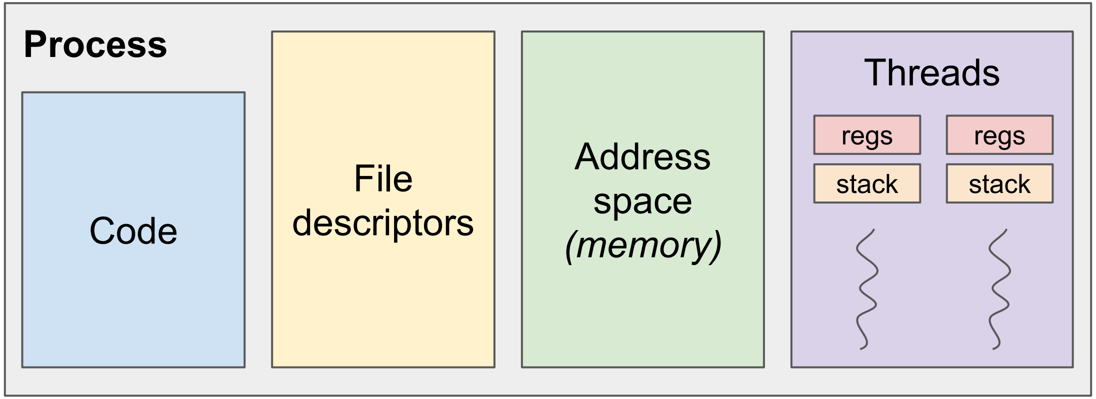
- It contains:
  - A set of registers, including an instruction pointer
  - A stack
- When scheduled on a core, a thread:
  - Executes the instruction located at the address pointed by its *instruction pointer*
  - Updates the instruction pointer (increments it or new value in case of a jump)
  - Repeat
- Scheduling:
  - When, which and where a thread is running
  - Cooperative and preemptive schedulers
  - Various election algorithms

Threads vs Process

A thread **IS NOT** a process!!!

A process is:

- A program (executable code)
- A set of resources (memory, files, …)
- A set of one or more threads that share these resources



Parallel programming

- Threads can collaborate on the same task to accelerate it (scale up)
- They need to communicate to share data and synchronize their work
- Parallelization techniques/tools:
  - Managing threads
  - Communication mechanisms
  - Synchronization primitives
  - Parallel programming patterns

Synchronization primitives

- Synchronization primitives are used to communicate between threads to avoid clashes on shared data, or simply to order operations

- Widely used synchronization primitives
    - Mutexes: mutual exclusion lock
    - Readers-writer locks: multiple concurrent readers, only one writer
    - Semaphores: mechanism to wait on a given number of resources
    - Barriers: mechanism to wait for a given number of threads at a certain point

Synchronization hazards

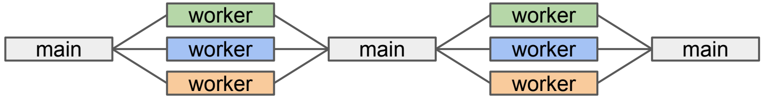
Wrong usage of synchronization can create different problems:

- **Deadlock:** A situation where threads cannot progress because they are all waiting for each other to progress.
- **Livelock:** Similar to a deadlock, but threads are not waiting for each other. Instead, they are still performing actions, but no progress is done.
- **Starvation:** State where a thread is perpetually denied access to a resource.

Parallel programming patterns

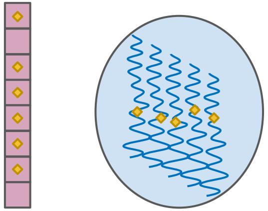
Fork-join:

- Create threads to perform a task, *i.e.*, *fork*
- Wait for them to be done, *i.e.*, *join*



Work stealing:

- Create a *pool of worker threads*
- Create a *work queue*
- Workers get tasks from the work queue and process them



Communication paradigms

When accessing a shared resource (memory, device), there are two access paradigms:

Synchronous: accessing thread doesn’t do anything until the resource is available

- Active **polling** until the resource is available

  E.g., when waiting for a network packet

```
while (!isPacketAvailable()) {}
p = getPacket();
process(p);
```

- Blocking access until the resource is available

  E.g., most basic IO functions are blocking

Asynchronous: accessing thread does something else until the resource is available

- Register to be notified when the resource is available

  E.g., receive a signal

- Poll the resource between the processing of other tasks

Lecture topics overview

- ~~**Part I:** Performance~~
- ~~**Part II:** Concurrency (or Scale Up!)~~
- **Part III:** Scalability (or Scale Out!)

Using many machines instead of one

Scale-out (horizontal scaling): use multiple smaller machines instead of one big one

- No shared memory, no mutexes etc. across machines
- Code on different machines communicates via requests/responses over a network
- If you need more resources, add more machines

Advantages:

- Smaller machines are mass-produced –> cheaper
- Machines can be placed around the world, close to users –> lower latency
- Fault tolerance: one machine fails –> the rest continues providing service
- Split data and computation across machines –> can handle extremely large workloads

Scaling out the data layer

Scaling out techniques:

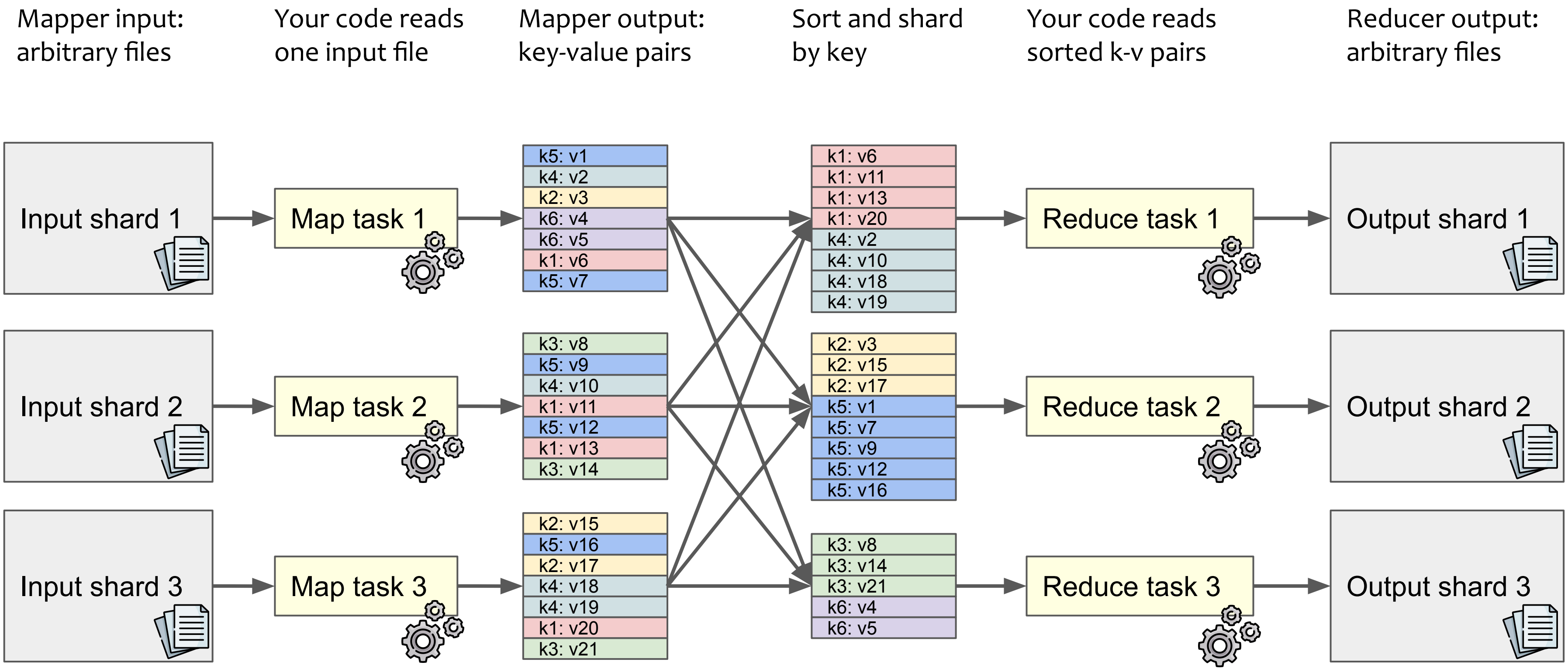
- **Replication**: maintain copies of data on multiple machines; when the data changes, make sure all copies are updated.
  - State machine replication
- **Sharding**: split a large dataset into smaller parts, so that each machine only stores some of the data.
  - “Hash modulo n” sharding
  - Consistent hashing
  - Local (document-partitioned) secondary indexes
  - Global (term-partitioned) secondary indexes

Both techniques are used together: typically first split a dataset into shards, then use replication to have copies of each shard.

Beyond scalable storage: scalable computation

- Request routing and secondary indexing:
  - useful when you want to look up **a specific record** in a sharded storage system.
- What about batch processing, when you want to **process all records**?
  - Processing everything on one machine is too slow –>
    need to distribute program across many nodes
- **Examples:**
  - Analytics (e.g. business intelligence, data science)
  - Training AI/machine learning models on large amounts of data
  - Searching for patterns in the data (e.g. fraud detection)
  - Scientific computing (getting results from experiments that generate lots of data: e.g., particle accelerators, astronomical telescopes, genome sequencing)

MapReduce framework for large-scale data processing

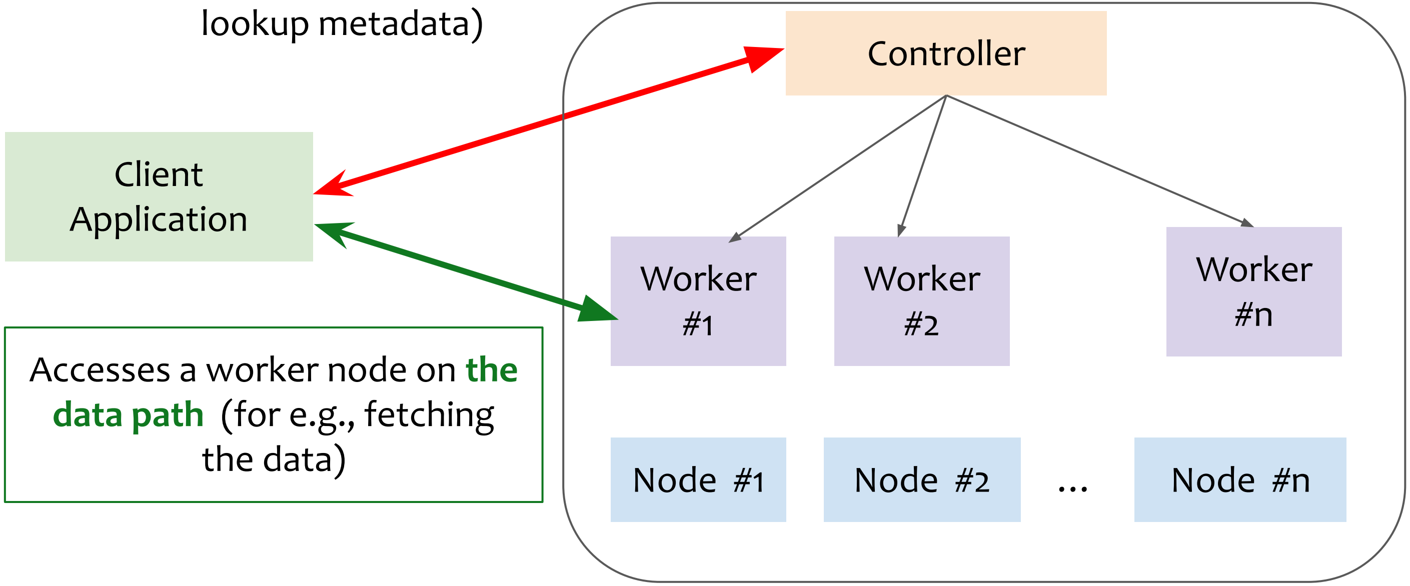


Scalable data management system

- **Single-node abstraction**
  - Usage model
  - APIs
  - Illustrative systems
- **Distributed system architecture**
  - For **scalable data management in the cloud**
  - A **single machine can't store and serve** large amounts of data (or " Big Data")!

Controller-worker architecture: Control vs data paths

Accesses the controller node on **the control path** (for e.g., lookup metadata)



Controller-worker distributed system architecture

The **controller node** manages the worker nodes

The **worker processes** do the actual work (Computation “OR” data management)

Machines/nodes in a data center

Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- **Part II:** **Programming basics**

- **Part III:** Homework programming exercises (Artemis)

Programming Basics (PB) exercises

L04PB01 Banking: a Multi-threaded Saga [Concurrency]

Start exercise | Not released | Optional | tutorial | Easy

L04PB02 Sharding in Pickledb [Scalability]

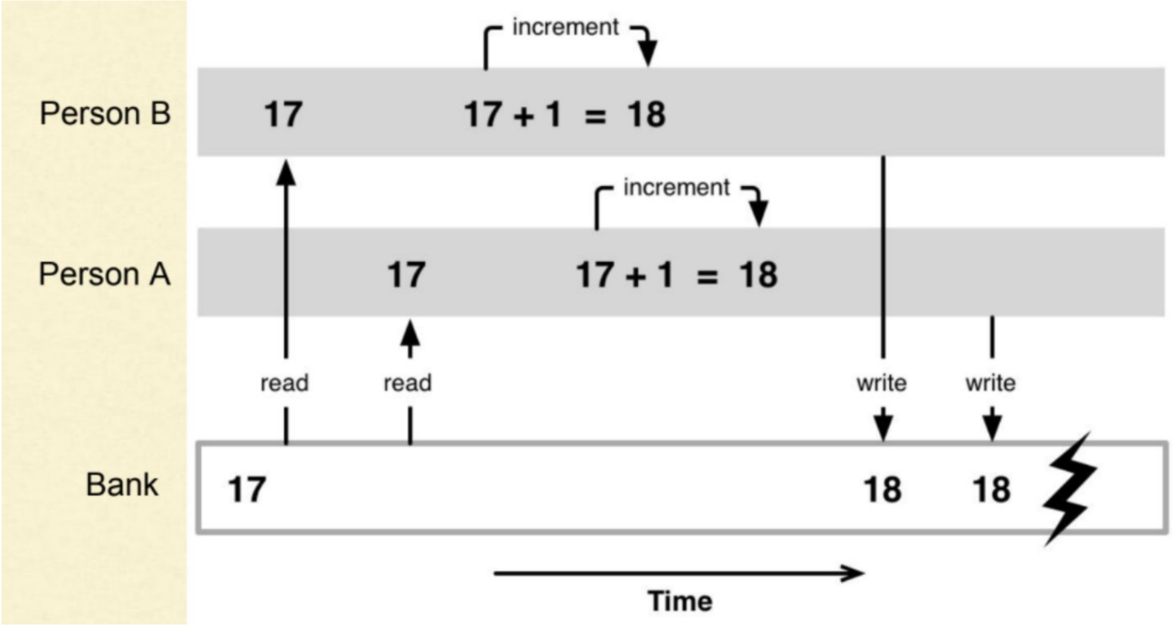
Start exercise | Not released | Optional | tutorial | Easy

L04PB01 Banking: a Multi-threaded Saga [Concurrency]

- Tasks:
  - Your task is to ensure the integrity of the banking system by preventing race conditions and deadlocks
  - Assume an online banking system that supports the withdraw and deposit functions
    - Fix the race conditions
    - Fix the deadlocks
- Goals:
  - **Understand** the concept of threads, race conditions, and deadlocks
  - **Experience** how thread synchronization ensures data consistency and prevents race conditions in a multi-threaded banking system

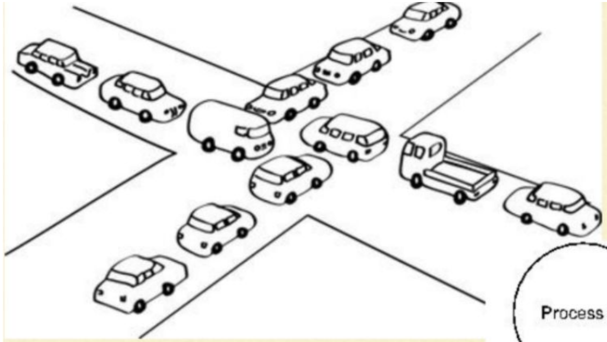
L04PB01 Banking: a Multi-threaded Saga [Concurrency]

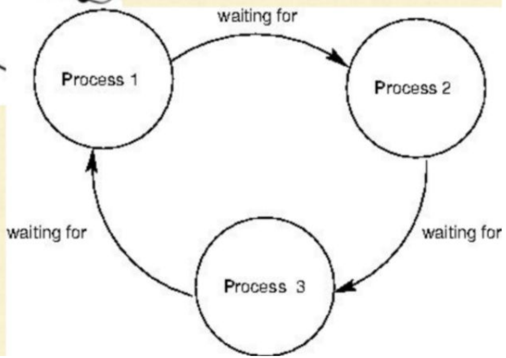
- Race condition:
  - Two processes/threads try to execute a task simultaneously and they cause a data corruption



L04PB01 Banking: a Multi-threaded Saga [Concurrency]

- Deadlock:
  - A situation in computing where two or more processes are unable to proceed because each is waiting for the other to release a resource





L04PB01 Banking: a Multi-threaded Saga [Concurrency]

```java
// TODO 1: Add synchronization mechanisms to prevent Race Condition
// Deposits the given amount to this bank account
public void deposit(long amount) {
    this.raceConditionLock.lock();
    this.balance += amount;
    this.raceConditionLock.unlock();
}
```

Create a raceConditionLock

```java
// TODO 1: Add synchronization mechanisms to prevent Race Condition
// Withdraws the given amount from this bank account
public void withdraw(long amount) {
    this.raceConditionLock.lock();
    this.balance -= amount;
    this.raceConditionLock.unlock();
}
```

Create a raceConditionLock

L04PB01 Banking: a Multi-threaded Saga [Concurrency]

```java
// Another possible solution to fixing the race condition

// TODO 1: Add synchronization mechanisms to prevent Race Condition
// Deposits the given amount to this bank account
public synchronized void deposit(long amount) {
    this.balance += amount;
}

// TODO 1: Add synchronization mechanisms to prevent Race Condition
// Withdraws the given amount from this bank account
public synchronized void withdraw(long amount) {
    this.balance -= amount;
}
```

Create a synchronization mechanism

L04PB01 Banking: a Multi-threaded Saga [Concurrency]

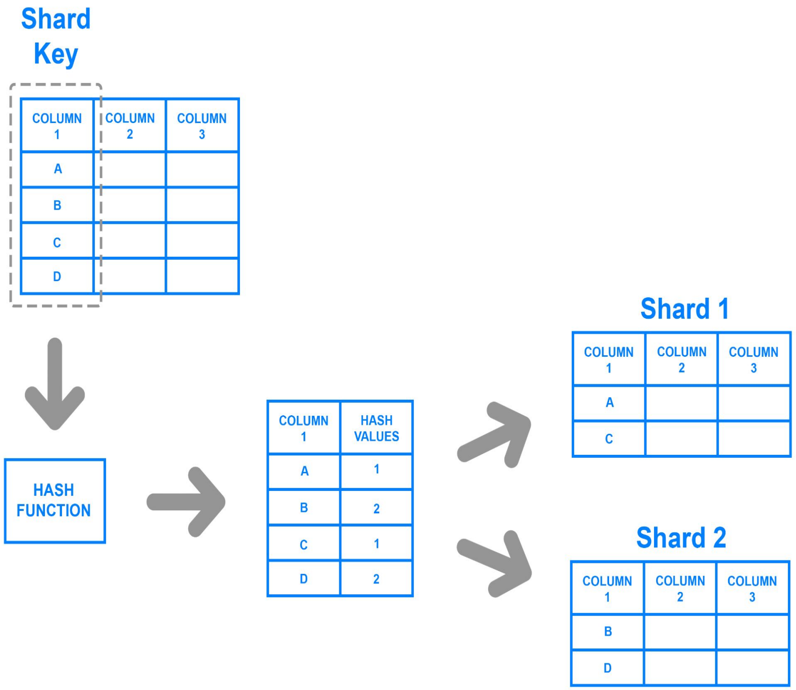
```java
// The given amount is transferred from this bank account to the destination bank account.
public void transfer(BankAccount destination, long amount) {
    // TODO 2: Prevent the deadlock
    ReentrantLock firstMutex = (this.accountId < destination.accountId) ? this.securityMutex : destination.securityMutex;
    ReentrantLock secondMutex = (this.accountId < destination.accountId) ? destination.securityMutex : this.securityMutex;

    secureTransfer(destination, amount, firstMutex, secondMutex);
}
```

Create a mutex

L04PB02 Sharding in Pickledb [Scalability]

- Tasks:
  - Implement sharding to improve the performance of a Python Key-Value Store
  - Divide keys (book titles) into smaller, manageable chunks
  - Store them in separate database nodes
- Goals:
  - **Understand** the concept of sharding
  - **Implement** a consistent hashing algorithm to assign keys to shards
  - **Experience** how hashing enables scalable data distribution



L04PB02 Sharding in Pickledb [Scalability]

- Database Nodes
  - Create and manage multiple database nodes by using the pickledb library
  - Each node is responsible for storing a specific range of keys based on the consistent hashing algorithm

- Practical Implementation
  - Implement the consistent hashing algorithm to assign keys to shards
  - The sharded database is designed to handle a range of book titles, with each node responsible for a specific set of keys

L04PB02 Sharding in Pickledb [Scalability]

- Sharding
  - A database scaling technique where a large dataset is partitioned into smaller, more manageable parts called shards
  - Each shard is then stored on a separate database node
  - This approach distributes the workload, improving system performance and enabling horizontal scaling

- Consistent Hashing
  - Consistent hashing is a technique used to distribute keys uniformly across shards
    - While minimizing the reassignment of keys when the number of shards changes
  - It provides a balance between data distribution and avoids unnecessary data movement

L04PB02 Sharding in Pickledb [Scalability]

- Ensure you have Python 3.12.0 and pip installed
  - py -m ensurepip --default-pip (windows)
  - sudo apt-get install python3-pip (mac or linux)
- And pickledb installed
  - pip install pickledb
- Review sharding concepts
  - Familiarize yourself with sharding and consistent hashing algorithms

L04PB02 Sharding in Pickledb [Scalability]

```python
class ShardedDatabase:
    def __init__(self):
        self.num_nodes = 10
        self.nodes = {i: pickledb.load(f"database_node_{i}.db", False) for i in range(0, 10)}
        self.store_books()

    def hash_key(self, book):
        # For this example, we determine the node based on the first letter of the key
        if not book[0].isalpha():
            return 9
        first_letter = book[0].upper()
        if 'A' <= first_letter <= 'C':
            return 0
        elif 'D' <= first_letter <= 'F':
            return 1
        elif 'G' <= first_letter <= 'I':
            return 2
        elif 'J' <= first_letter <= 'L':
            return 3
        elif 'M' <= first_letter <= 'O':
            return 4
        elif 'P' <= first_letter <= 'R':
            return 5
        elif 'S' <= first_letter <= 'U':
            return 6
        elif 'V' <= first_letter <= 'X':
            return 7
        elif 'Y' <= first_letter <= 'Z':
            return 8
        else:
            return -1

    def store_books(self):
        for book in books:
            # Map study courses to hash-modulo keys
            node_index = self.hash_key(book)
            if node_index != -1:
                self.nodes[node_index].set(book, node_index)
                self.nodes[node_index].dump()
```

The hash function!

Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- ~~**Part II:** Programming basics~~

- **Part III: Homework programming exercises (Artemis)**

Programming (P) exercises

L04P01 Swimming Pool Facility Management [Concurrency]

Start exercise

Not released | homework | Bonus | Easy

L04P02 Book Management System [Scalability]

Start exercise

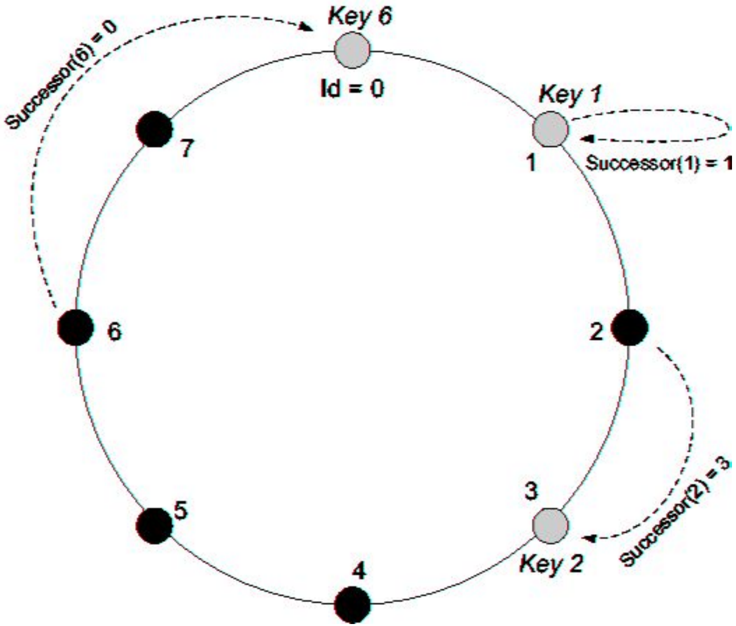
Not released | homework | Bonus | Easy

L04P01 Swimming Pool Facility Management [Concurrency]

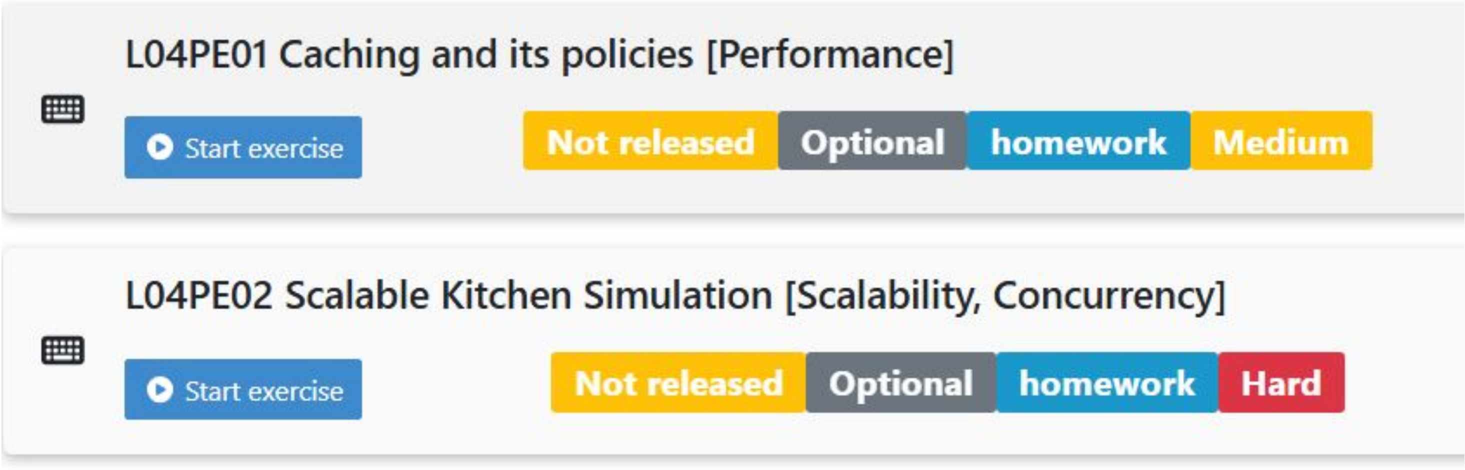
- Tasks and Goals:
  - Understand how resource acquisition can lead to deadlocks in multi-threaded environments.
  - Learn to reproduce, detect and prevent deadlocks using synchronization and ordering strategies.

L04P02 Book Management System [Scalability]

- Tasks and Goals:
  - Understand how Chord, a consistent hashing algorithm, distributes keys across a dynamic set of nodes.
  - Implement key components of a Chord Hash Ring: node addition, key hashing, finding successor and key insertion.



Programming Extras (PE) exercises

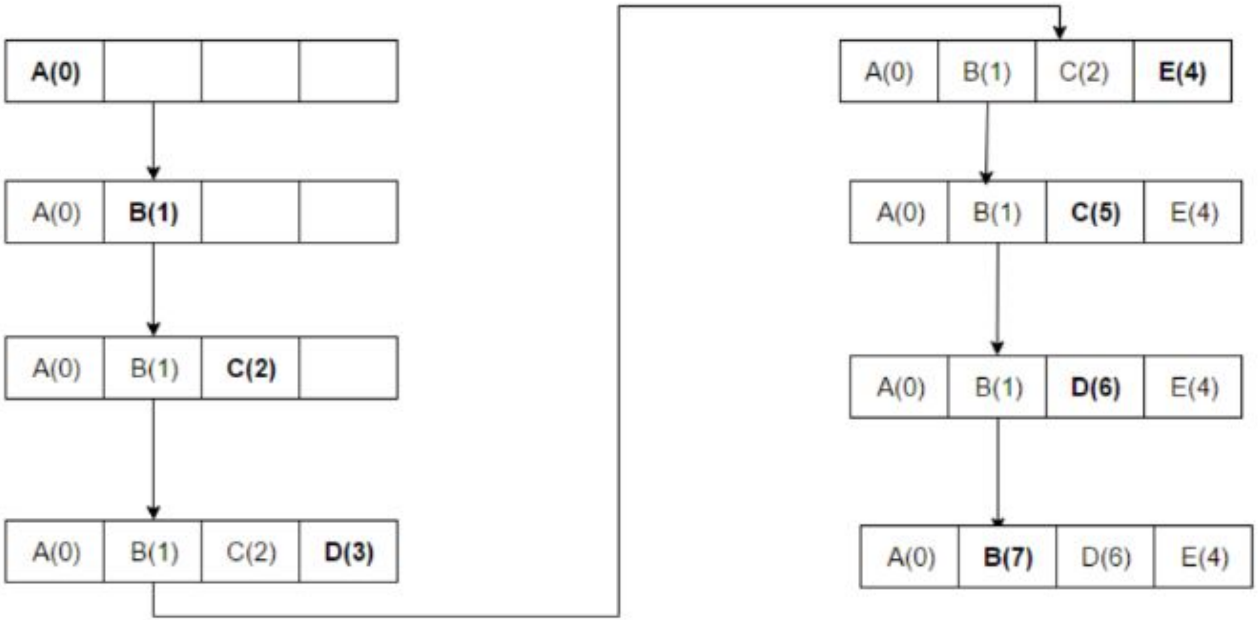


L04PE01 Caching and its policies [Performance]

- Goal:
  - Implement a KV-store with 5 different cache replacement policies to improve performance and reduce expensive DB calls.
- Policies:
  - FIFO: Evict first inserted item
  - LRU: Evict least *recently* used item
  - MRU: Evict most *recently* used item
  - LFU: Evict least *frequently* used item
  - MFU: Evict most *frequently* used item

L04PE01 Caching and its policies [Performance]

Example for MRU:



L04PE02 Scalable Kitchen Simulation [Scalability, Concurrency]

- Goal:
  - Build a thread-safe, efficient system that manages and balances restaurant orders across multiple kitchens using multithreading and synchronization.
- Tasks:
  - Implement a restaurant order management system with:
    - Multiple kitchens, each with a priority queue of orders and Chefs (threads) that continuously pull from this queue and process orders.
    - Restaurant manager, that receives new orders and periodically checks for overloaded kitchens and rebalances orders.

L04PE02 Scalable Kitchen Simulation [Scalability, Concurrency]

Basic UML class diagram: