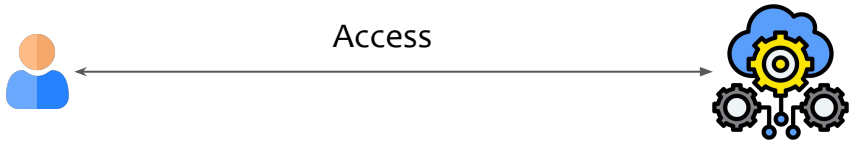
L06: System Design and Implementation III

Prof. Pramod Bhatotia, Prof. Redha Gouicem, and Evgeny Volynsky
Chair of Distributed Systems and Operating Systems
https://dse.in.tum.de/

Roadmap



Our focus:

How to analyze, design, implement, test, and deploy cloud software systems?

L03: Cloud Design Requirements → L04 + L05 + L06: System Design and Implementation → L07 + L08: Software Testing and Program Analysis → L09: Software Management, Deployment, & Monitoring

L10: Software Quality and Program management

L03 + L04: Cloud Software Architectures

L02: Cloud Systems Engineering

A three-part series: System design in our course

L04: Design I

- **Modularity**
  - How to design modular systems?
- **Data management**
  - How to manage your data?

L05: Design II

- **Security**
  - How to secure your systems?
- **Fault tolerance**
  - How to make systems reliable & available?
- **Performance**
  - How to design performant systems?

L06: Design III

- **Concurrency (Scale-up)**
  - How to scale-up systems?
- **Scalability (Scale-out)**
  - How to scale-out systems?

System implementation

Announcement #1: Nachteilausgleiche

- If you have a special circumstance approval from the examination board **(Nachteilausgleiche)**, please send your Nachteilausgleiche certificate to
  - **Evgeny Volynsky** (evgeny.volynsky@tum.de) by **June 23rd**

Announcement #2: Feedback survey

- **The feedback survey** period for EIST has started. You must have received an email to participate in the survey.

- We would appreciate your feedback about the course overall. In particular, we would appreciate your feedback on **the new curriculum focusing on cloud software systems**.

- **This feedback is critical for making this new curriculum permanent**, and further improve our offerings.

Today's learning goals

- **Part I:** Concurrency (or Scale Up!)
  - Why concurrency?
  - The thread model
  - Thread scheduling
  - Communication mechanisms
  - Parallelizing a program
  - Accelerators
- **Part II:** Scalability (or Scale Out!)
  - Scalability challenges
  - Scalability techniques
- **Part III:** Pattern implementation
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

Lecture schedule

#	Date	Subject
1	20.04	Introduction **(no tutorials)**
2	27.04	Cloud Systems Engineering
3	04.05	System Design Requirements + Cloud Software Architectures
	11.05	**EuroSys Conference (no lecture, no tutorials)**
	18.05	**Holiday (no lecture, no tutorials)**
4	25.05	System Design and Implementation I
5	01.06	System Design and Implementation II
	08.06	**Holiday (no lecture, no tutorials)**
6	15.06	System Design and Implementation III
7	22.06	Software Testing
8	29.06	Program Analysis
9	06.07	Software Management, Deployment, and Monitoring
10	13.07	Software Quality and Project Management + **Exam prep.**
	20.07	Guest Lecture **(no tutorials)**

Tutorial schedule

#	Date	Based on the lecture content of:
1	**No tutorials**	**Introduction**
2	03.05 – 05.05	Cloud Systems Engineering
3	10.05 – 12.05	System Design Requirements + Cloud Software Architectures
4	31.05 – 02.06	System Design and Implementation I
5	07.06 – 09.06	System Design and Implementation II
6	21.06 – 23.06	System Design and Implementation III
7	28.06 – 30.06	Software Testing
8	05.07 – 07.07	Program Analysis
9	12.07 – 14.07	Software Management, Deployment, and Monitoring
10	19.07 – 21.07	Software Quality and Project Management + **Exam prep.**
	No tutorials	**Guest lecture**

Outline

- **Part I: Design principle: Concurrency (or Scale Up!)**
  - Why concurrency?
  - The thread model
  - Thread scheduling
  - Communication mechanisms
  - Parallelizing a program
  - Accelerators
- **Part II:** Design principle: Scalability (or Scale Out!)
- **Part III:** Pattern implementation

Performance (from L05)

- **Let's face it – performance matters!**
  - A large number of empirical studies show that **performance directly impacts success ($$$)**
  - E.g., Tail at Scale from Google https://www.barroso.org/publications/TheTailAtScale.pdf
  - Other e.g., include shopping (Amazon), streaming services (Netflix/Spotify/YouTube), etc.

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

SYSTEMS THAT RESPOND to user actions quickly (within 100ms) feel more fluid and natural to users than those that take longer.[3] Improvements in Internet

Empirical research shows that the successful adoption of most modern software systems is directly related to their **performance metrics**!

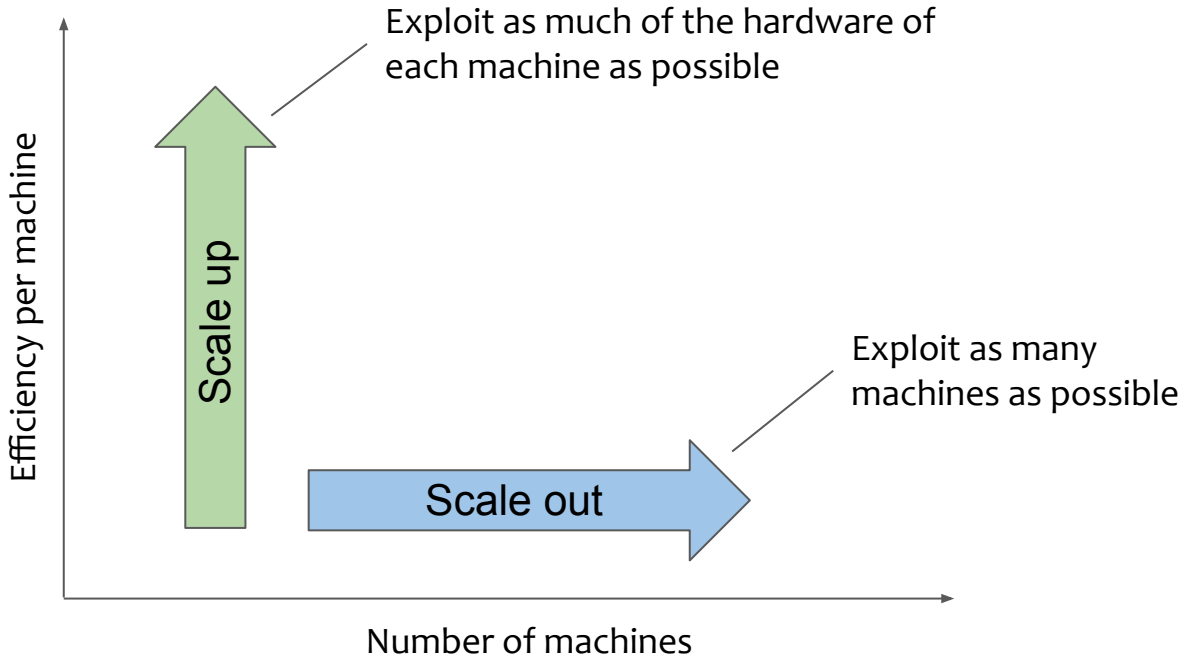
Performance is key

How can we exploit the hardware as much as possible?

Concurrent/Parallel software systems

(aka scale-up!)

Scaling your applications



Modern processor architecture

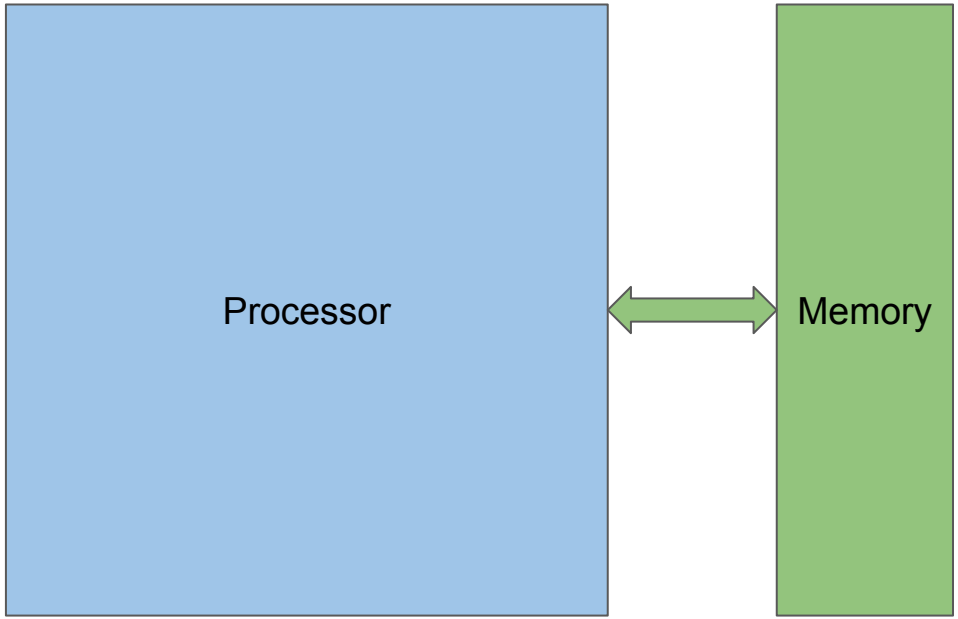
Computation

- Processing unit

  i.e., execute programs

Memory

- Stores data needed by programs during their execution



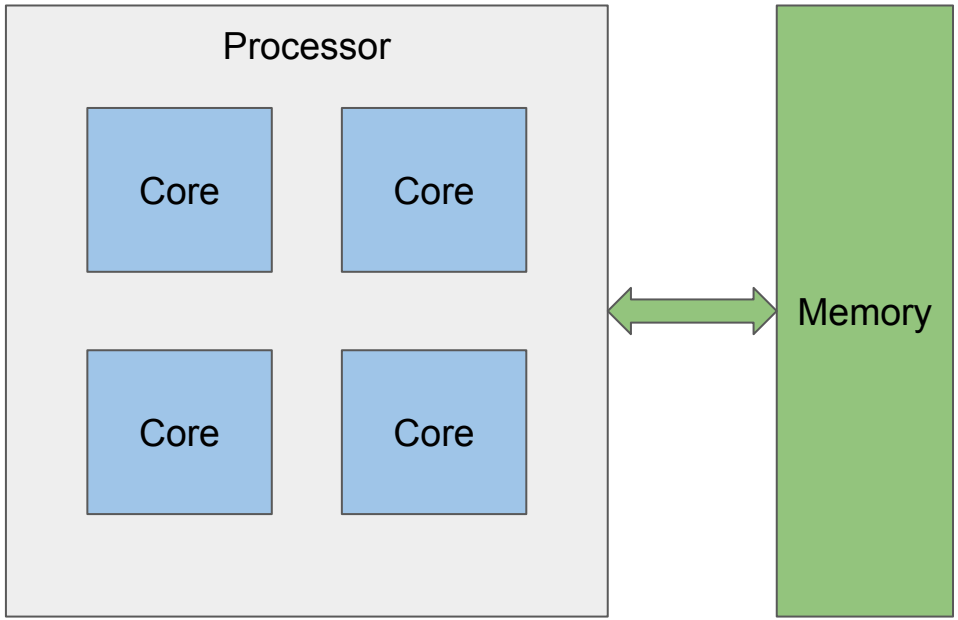
Modern processor architecture

Computation

- Processing unit**s** (cores)
- Perform computations independently,
  i.e., execute programs

Memory

- Stores data needed by programs during their execution



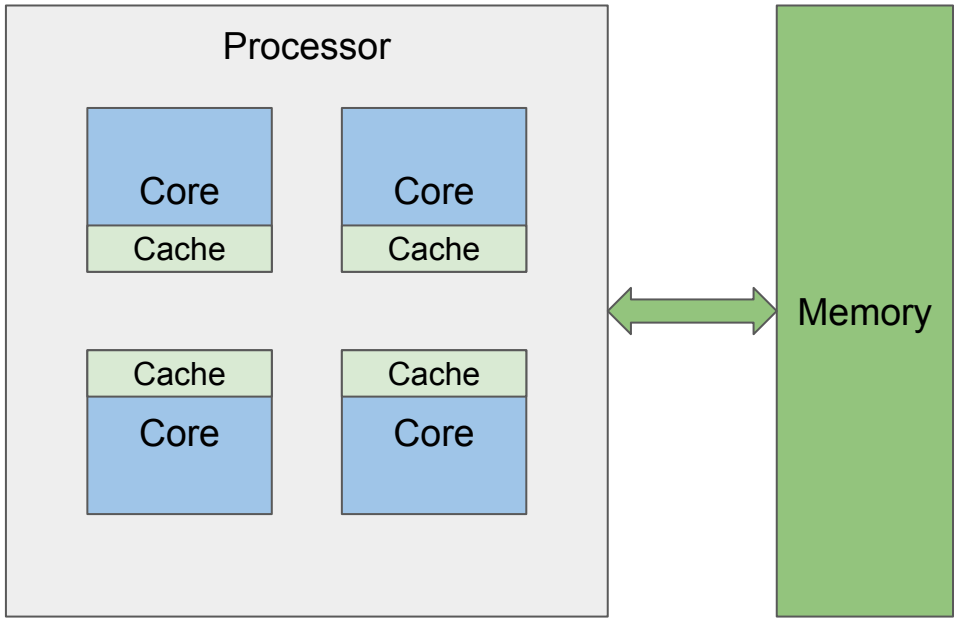
Modern processor architecture

Computation

- Processing unit**s** (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



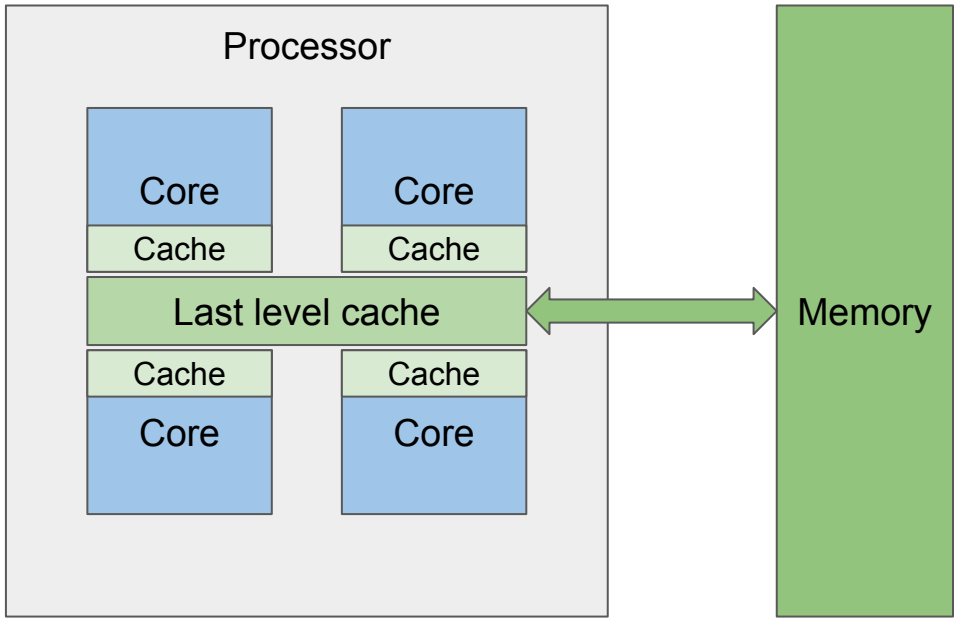
Modern processor architecture

Computation

- Processing unit**s** (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)



Modern processor architecture

Computation

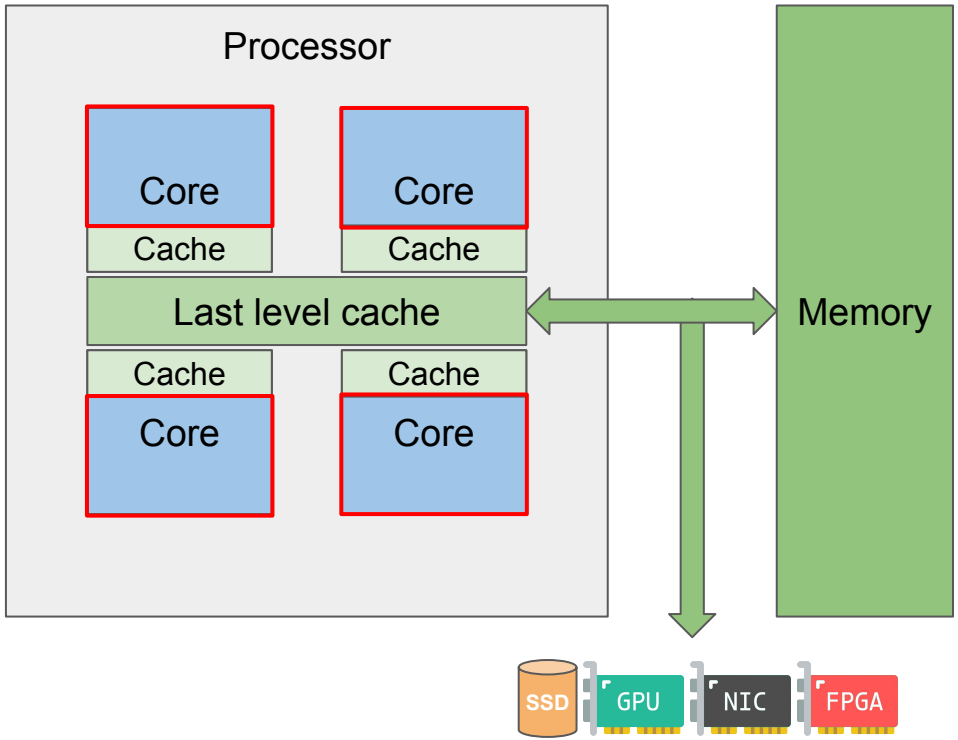
- Processing unit**s** (cores)
- Perform computations independently,
  i.e., execute programs

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…



Exploiting parallelism

Let's first focus on using the multiple cores available on our CPUs

Each core can independently execute a different task

But a **task ≠ program!!!**

A program can have multiple tasks

e.g., your web browser's tabs can be different tasks in the same program

How do operating systems provide this concept of task?

Thread model

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

It is the abstraction that represents the execution flow of a program

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

Threads vs Process

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

Threads vs Process

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

A process is:

Threads vs Process

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

A process is:

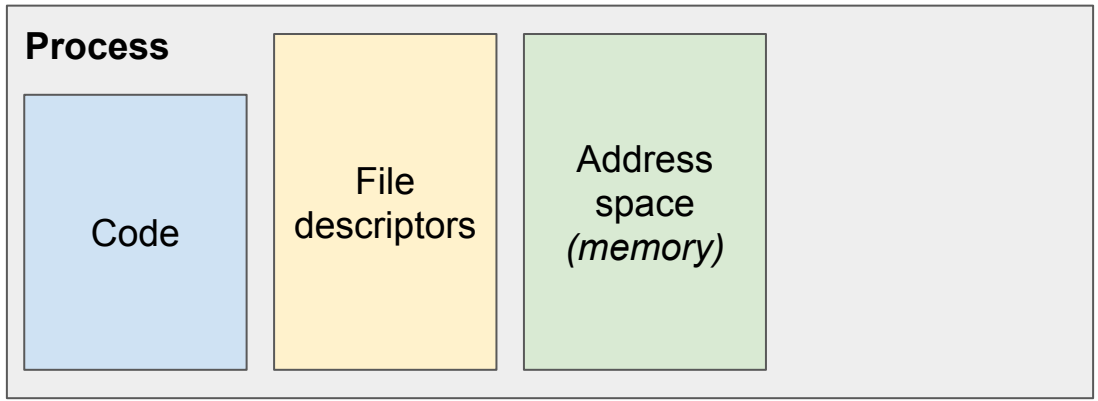
- A program (executable code)

Threads vs Process

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

A process is:

- A program (executable code)
- A set of resources (memory, files, …)

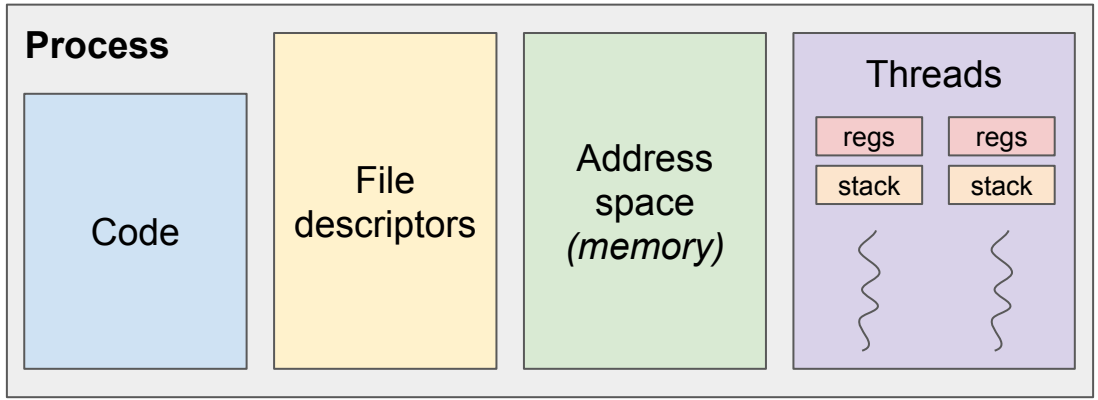


Threads vs Process

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



Thread scheduling

The scheduler is responsible for managing threads

- **When** is the scheduler invoked?

  Cooperative, preemptive? Time-based? Event-based?
- **Which** thread is executed?

  Election algorithm: FIFO? Priority? Real-time?
- **Where** is a thread executed?

  Hardware constraints (caches, SMT, NUMA, big.LITTLE, …)

  Load balancing

Thread scheduling: When?

Two main categories:

- **Cooperative schedulers**: the thread currently executing decides to yield the CPU and invokes the scheduler to choose a new thread.
- **Preemptive schedulers**: the scheduler forces out the executing thread to choose a new one. Preemption can occur for multiple reasons:
  - A periodic timer triggers an interrupt, e.g., for time-sharing
  - An IO event occurs, e.g., a network packet wakes up a waiting thread
  - A thread with a higher priority than the current one becomes available

Thread scheduling: Which?

The *election algorithm* decides the next thread to get access to the CPU

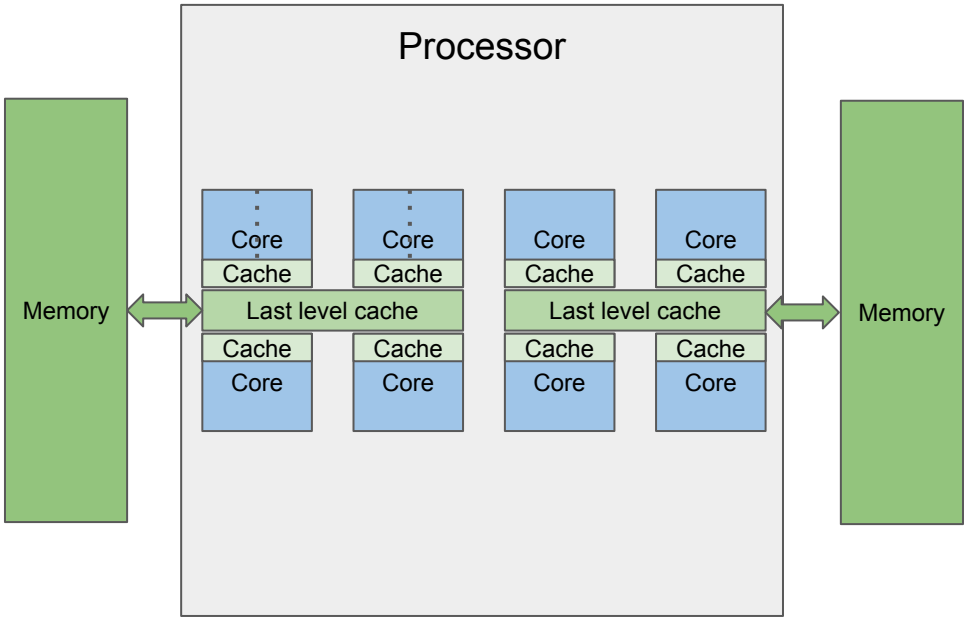
Some examples:

- **Round robin** → *First scheduler in Linux*
  - Store threads in a circular data structure, e.g., circular list
  - Elect the next one in the list
- **Fairness-based** → *CFS, the current Linux scheduler*
  - Give each thread the same amount of CPU time
  - Elect the thread that needs to "catch up" the most
- **Priority-based** → *O(1), Linux scheduler from 2.6.0 to 2.6.22 (2003–2007)*
  - A priority value is assigned to each thread
  - Elect the thread with the highest priority, FIFO if same priority
- **Earliest Deadline First (EDF)** → *Real time systems, Linux*
  - In a real-time context, threads have a deadline that they have to respect
  - Elect the thread with the earliest deadline

Thread scheduling: Where?

The scheduler chooses which core will execute a thread, with regards to some hardware constraints:

- Shared caches
- NUMA architectures
- Simultaneous Multi-Threading (SMT)
- Hybrid cores (e.g., big.LITTLE)



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

Let's have a brief overview of parallelization techniques/tools:

- Managing threads
- Communication mechanisms
- Synchronization primitives
- Parallel programming patterns

We will illustrate this with a web server example

Thread management

To allow your web server to handle more requests, you want to have each request handled by a different thread

Let’s see two ways of achieving this:

- Spawning threads for requests (event-based)
- Using a pool of threads

Thread management: Spawning upon request

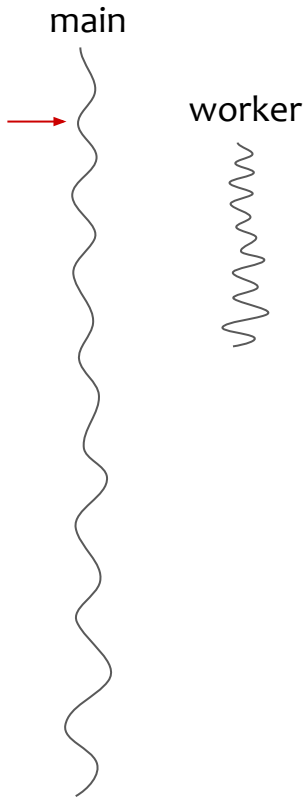
Your main thread waits for requests to arrive

Thread management: Spawning upon request

Your main thread waits for requests to arrive

When a new request arrives:

1. Create a new worker thread

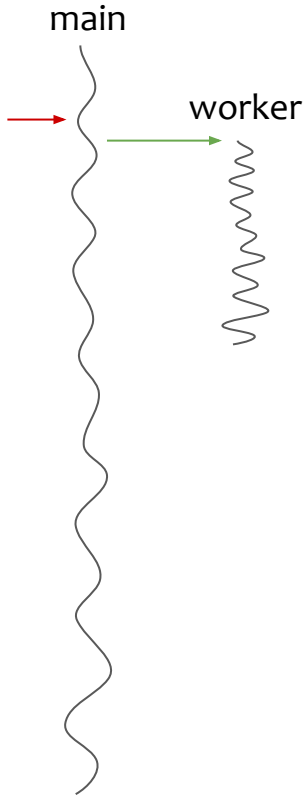


Thread management: Spawning upon request

Your main thread waits for requests to arrive

When a new request arrives:

1. Create a new worker thread
2. Pass the data of the request to the thread

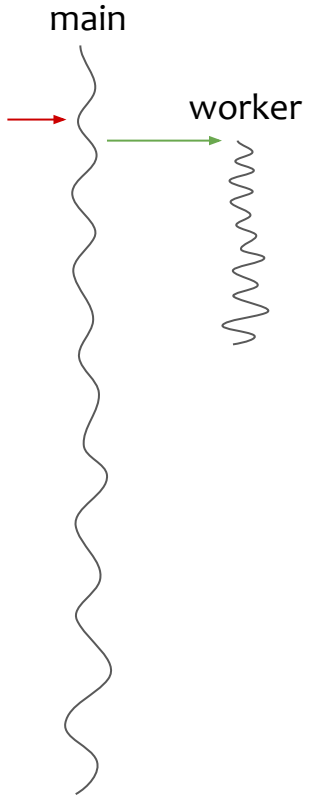


Thread management: Spawning upon request

Your main thread waits for requests to arrive

When a new request arrives:

1. Create a new worker thread
2. Pass the data of the request to the thread
3. Wait for a new request



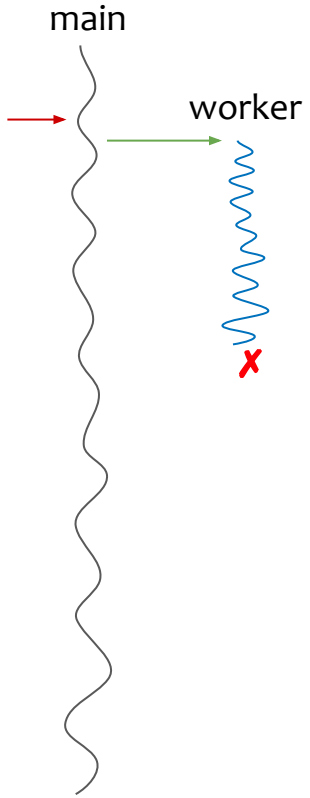
Thread management: Spawning upon request

Your main thread waits for requests to arrive

When a new request arrives:

1. Create a new worker thread
2. Pass the data of the request to the thread
3. Wait for a new request

Worker threads just process the request, reply to the client and terminate ✗



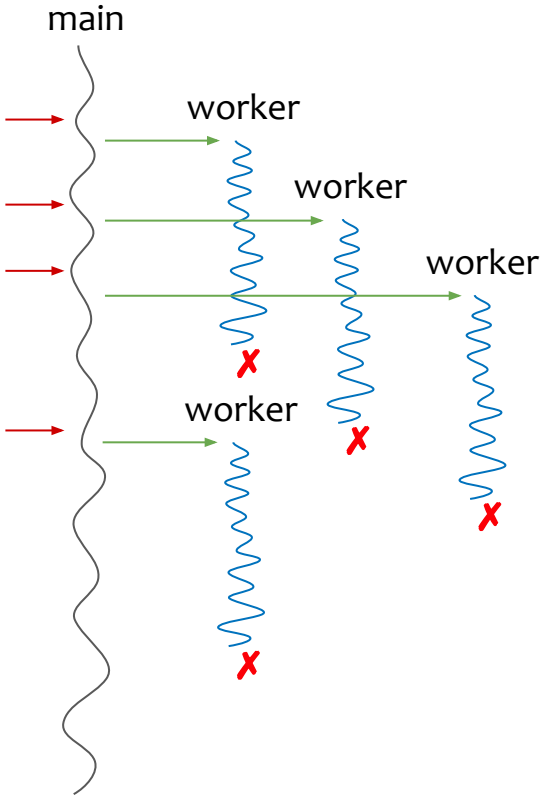
Thread management: Spawning upon request

Your main thread waits for requests to arrive

When a new request arrives:

1. Create a new worker thread
2. Pass the data of the request to the thread
3. Wait for a new request

Worker threads just process the request, reply to the client and terminate ✗



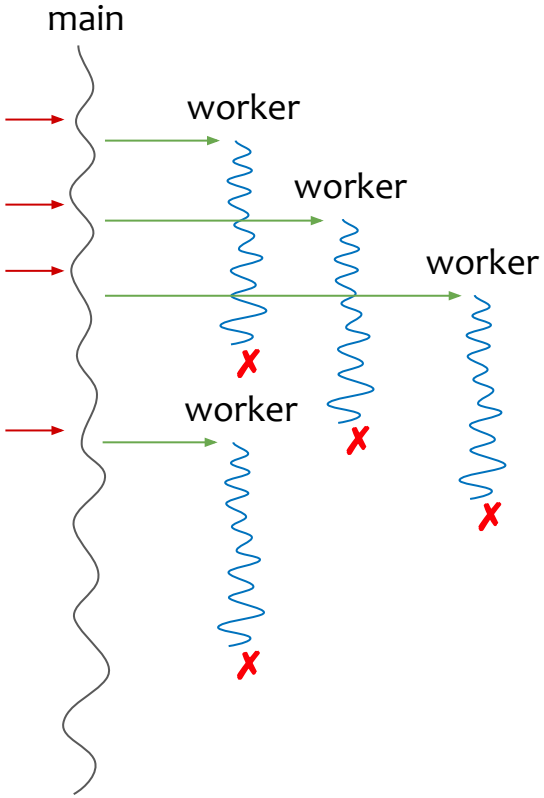
Thread management: Spawning upon request

Your main thread waits for requests to arrive

When a new request arrives:

1. Create a new worker thread
2. Pass the data of the request to the thread
3. Wait for a new request

Worker threads just process the request, reply to the client and terminate ✗

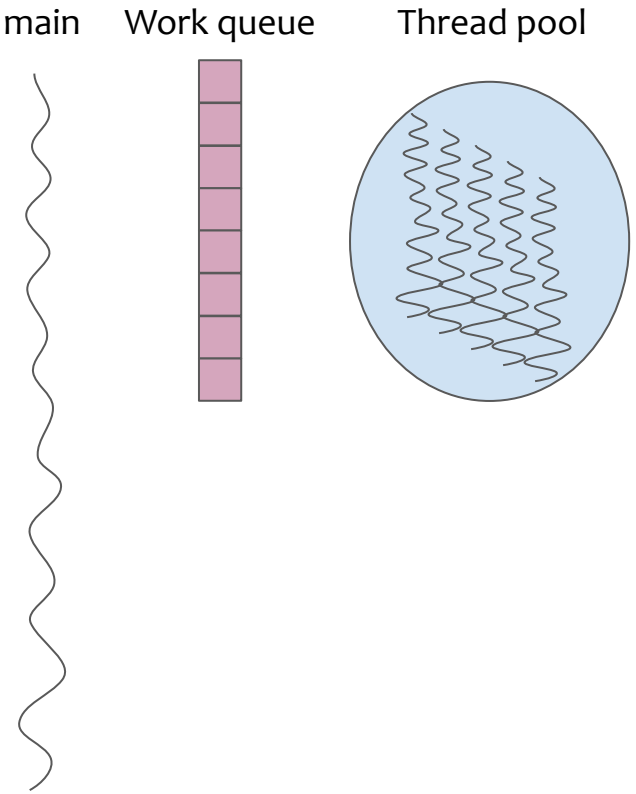


Pro: Very simple logic to implement

Con: Overhead of creating threads

Thread management: Thread pools

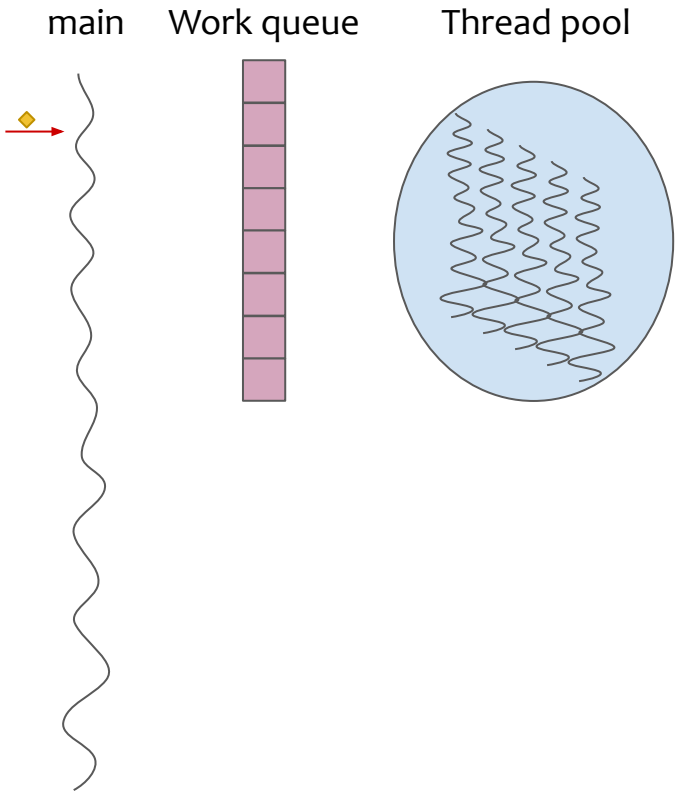
At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests



Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

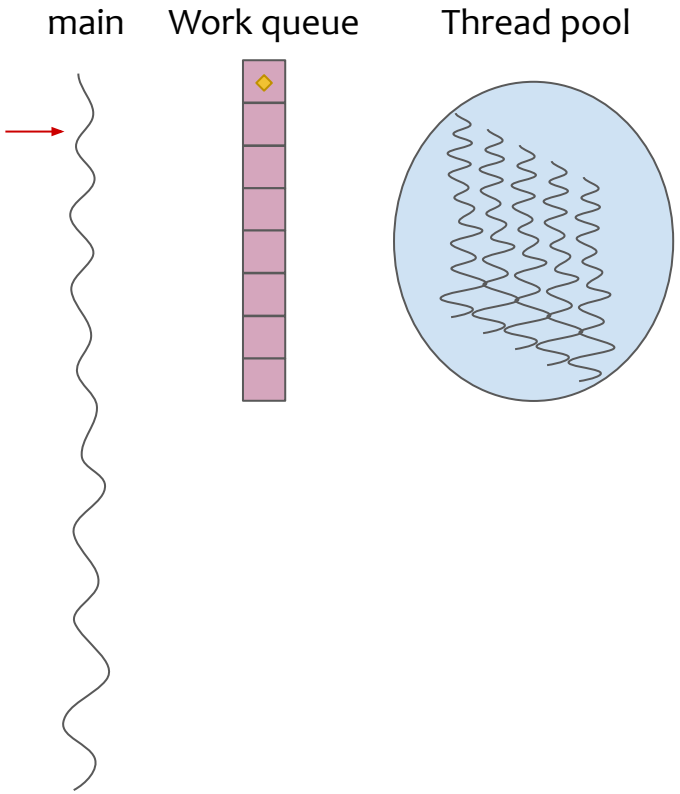
When a request arrives, the main thread enqueues it in the work queue



Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

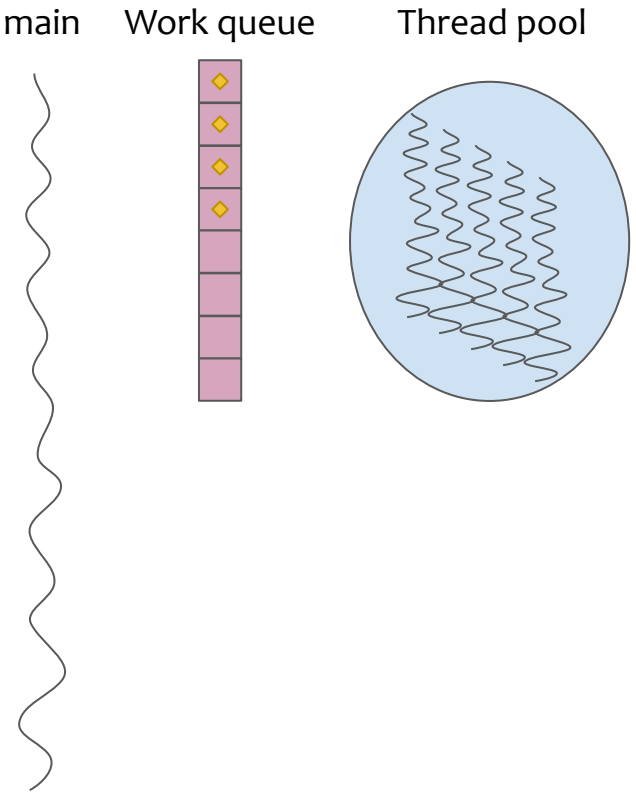
When a request arrives, the main thread enqueues it in the work queue



Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

When a request arrives, the main thread enqueues it in the work queue



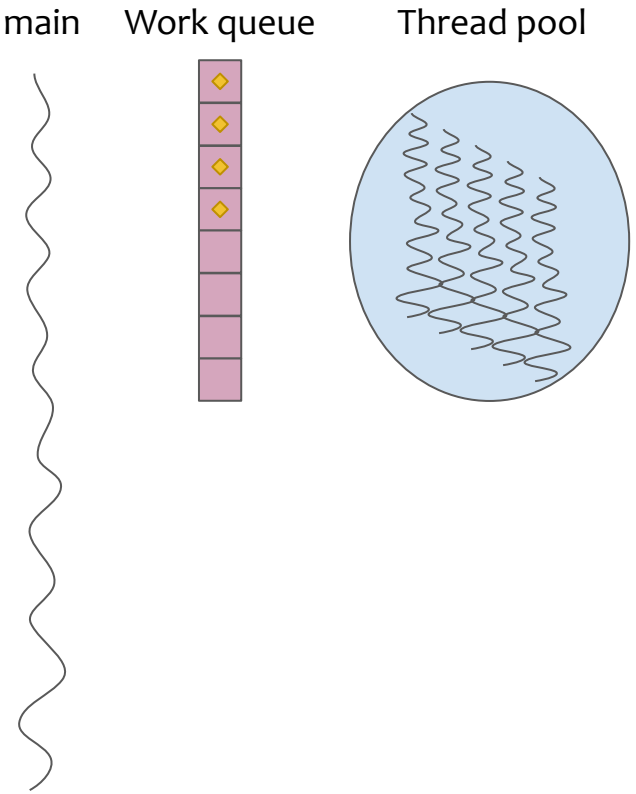
Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

When a request arrives, the main thread enqueues it in the work queue

Worker threads:

1. Wait for work to be available in the queue



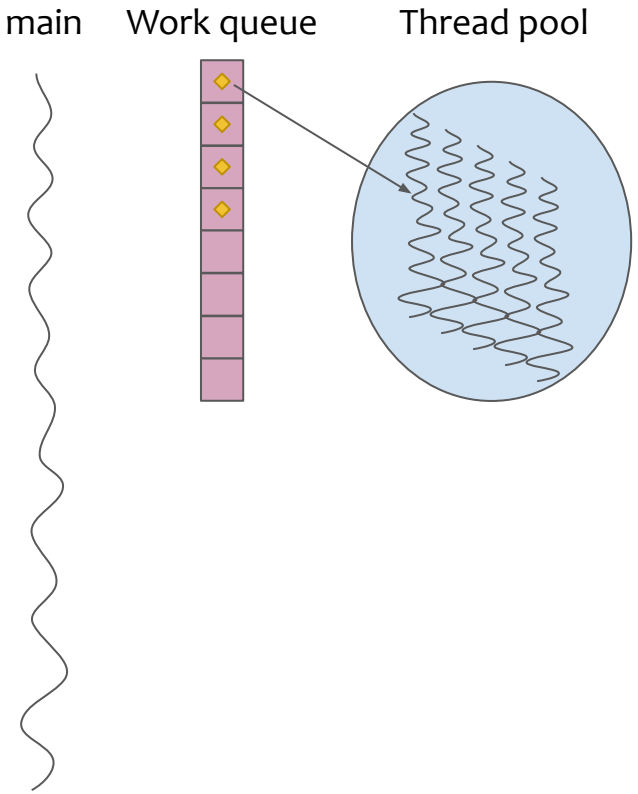
Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

When a request arrives, the main thread enqueues it in the work queue

Worker threads:

1. Wait for work to be available in the queue
2. Pick a request, process it, reply



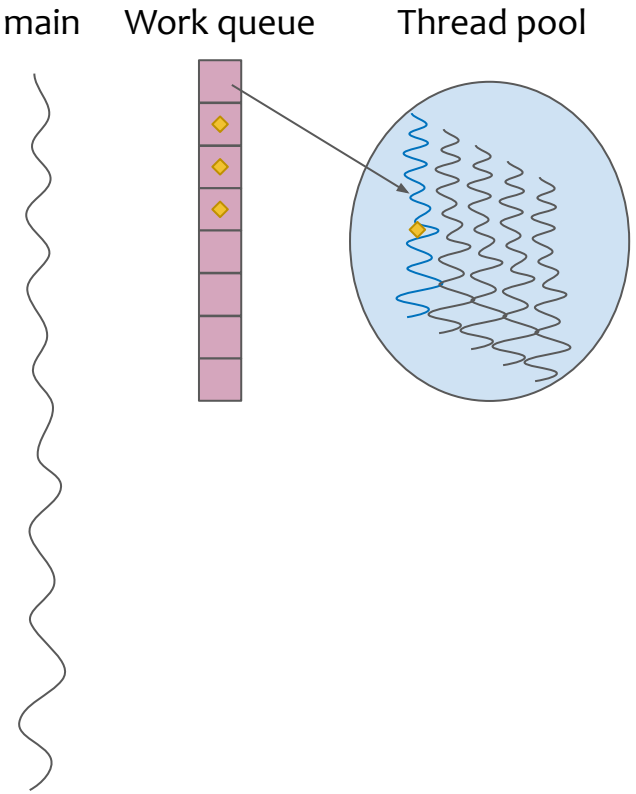
Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

When a request arrives, the main thread enqueues it in the work queue

Worker threads:

1. Wait for work to be available in the queue
2. Pick a request, process it, reply



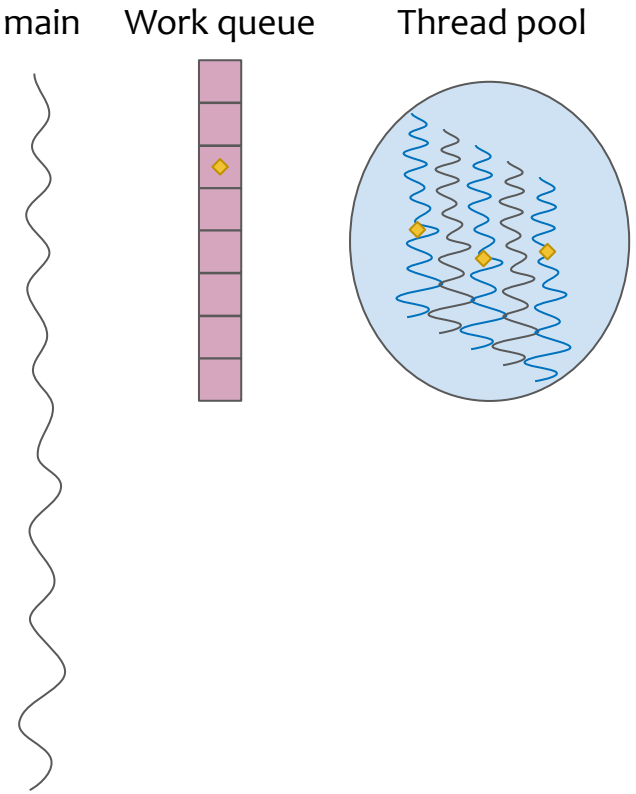
Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

When a request arrives, the main thread enqueues it in the work queue

Worker threads:

1. Wait for work to be available in the queue
2. Pick a request, process it, reply



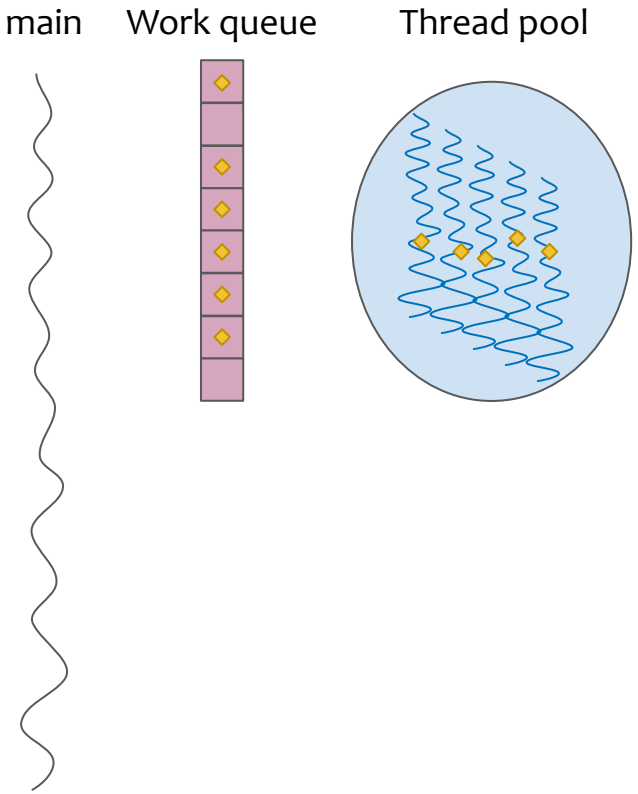
Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

When a request arrives, the main thread enqueues it in the work queue

Worker threads:

1. Wait for work to be available in the queue
2. Pick a request, process it, reply
3. Repeat



Thread management: Thread pools

At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests

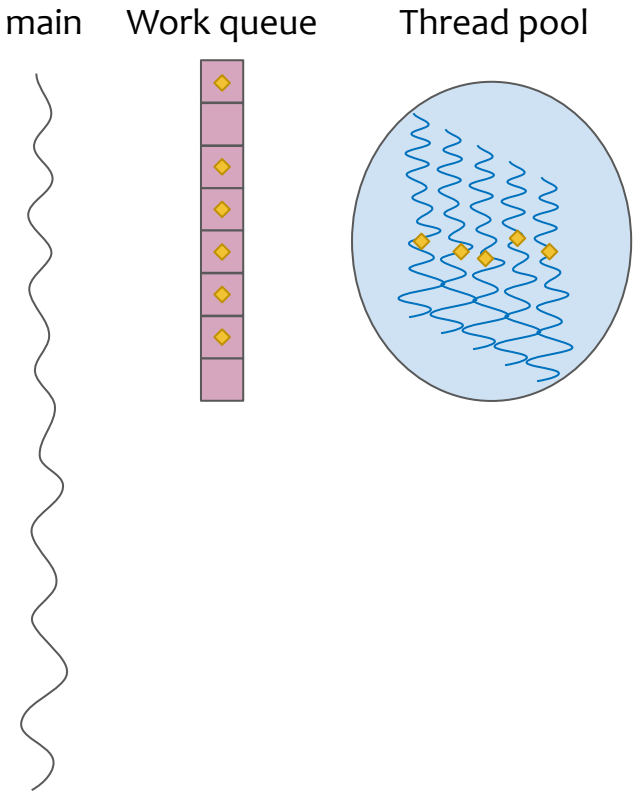
When a request arrives, the main thread enqueues it in the work queue

Worker threads:

1. Wait for work to be available in the queue
2. Pick a request, process it, reply
3. Repeat

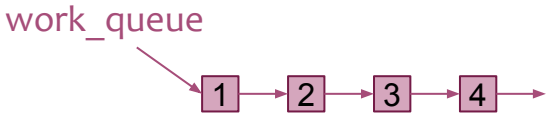
Pro: No thread creation overhead

Con: Synchronization required between workers



Why do we need synchronization?

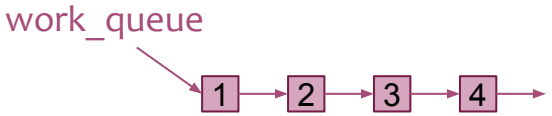
An easy implementation of a work queue would be a linked list, where each node is a request



Why do we need synchronization?

An easy implementation of a work queue would be a linked list, where each node is a request

To get the next request, we need to read the current head of the list and modify the head




```
1    Node get_request() {
2         Node n = work_queue;
3         work_queue = n.next;
4         return n;
5    }
```

Why do we need synchronization?

An easy implementation of a work queue would be a linked list, where each node is a request

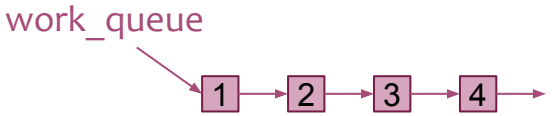
To get the next request, we need to read the current head of the list and modify the head

If two threads do this concurrently in that order:

- Worker 1 executes line 2
- Worker 2 executes line 2
- Worker 1 executes line 3
- Worker 2 executes line 3

Both workers will get the same request

→ **undesirable behavior!**




```
1    Node get_request() {
2         Node n = work_queue;
3         work_queue = n.next;
4         return n;
5    }
```

Definition time

A *race condition* is a situation where the behavior of a program depends on the order of the operations. It can become a source of bugs when an undesired behavior is made possible.

A *critical section* is a portion of code that can be executed by only one thread at a time to avoid wrong behaviors or bugs.

A multithreaded code is *thread safe* if shared data is accessed in a way that ensures that **all threads behave properly**, in a controlled manner, without *race conditions.*

Synchronization primitives

Let’s make a quick pause with our web server, we need some new tools

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

We will briefly introduce some widely used primitives

- Mutexes
- Readers-writer locks
- Semaphores
- Barriers

Mutexes

A mutex, or **mut**ual **ex**clusion lock, allows **only one** thread to enter a *critical section*

You can see this like a bathroom lock

It provides two primitives:

- *Lock*
  - If the mutex is available, acquire it
  - Else, wait for it to become available
- *Unlock*
  - Make the mutex available
  - Has to be called by the mutex owner

Mutexes in Java

Class **ReentrantLock**

`void lock()`

Acquire the lock if available, wait if not

`void unlock()`

Release the lock

Throws IllegalMonitorStateException if the thread is not the lock owner

Readers-writer locks

An asymmetric lock where you can have unlimited concurrent readers, but only one writer at a time with no concurrent readers

Useful for data structures there are heavily read, but rarely modified

Exposes four primitives:

- *read_lock*: acquire the readers lock (can be acquired by multiple threads)
- *read_unlock*: release the readers lock
- *write_lock*: acquire the writer lock (exclusive access)
- *write_unlock*: release the writer lock

Readers-writer lock in Java

Class **ReentrantReadWriteLock**

`ReentrantReadWriteLock.ReadLock readLock()`

Returns the lock used for readers

`ReentrantReadWriteLock.WriteLock writeLock()`

Returns the lock used for the writer

Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

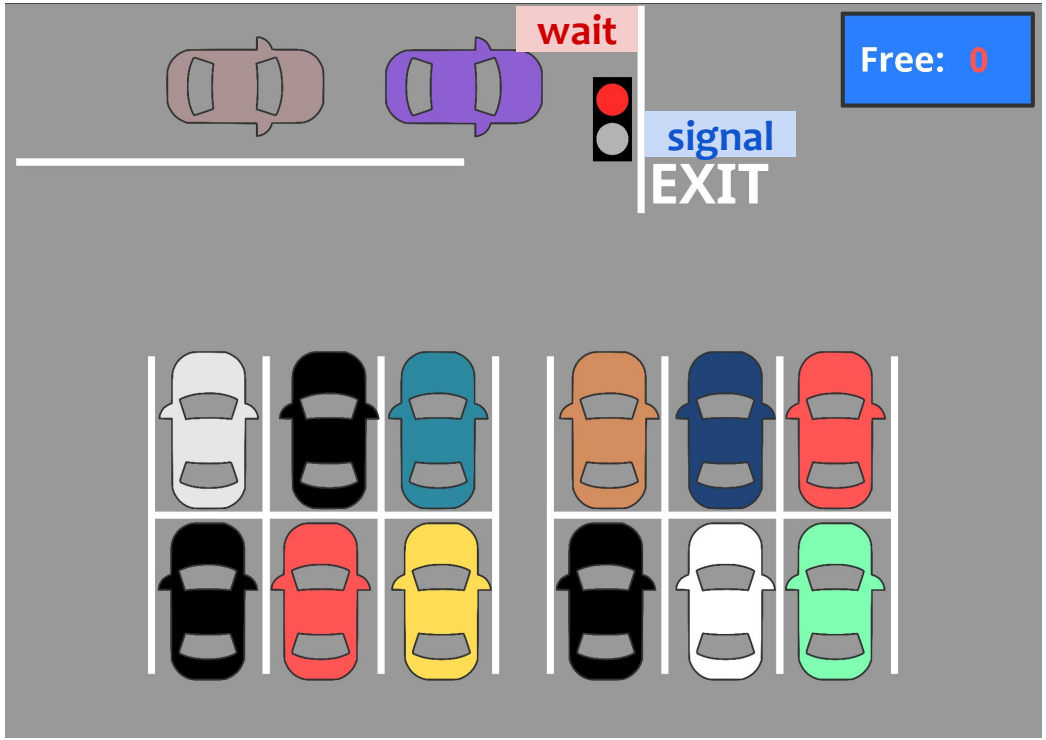
A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

Let's see a quick example!

Semaphore: Example

Managing a parking lot with **semaphores**



Semaphores in Java

Class **Semaphore**

`Semaphore(int permits)`

Create a semaphore with *permits* resources (K)

`void acquire()`

Acquire a permit [K -= 1], and block if no permit is available (*wait*)

`void release()`

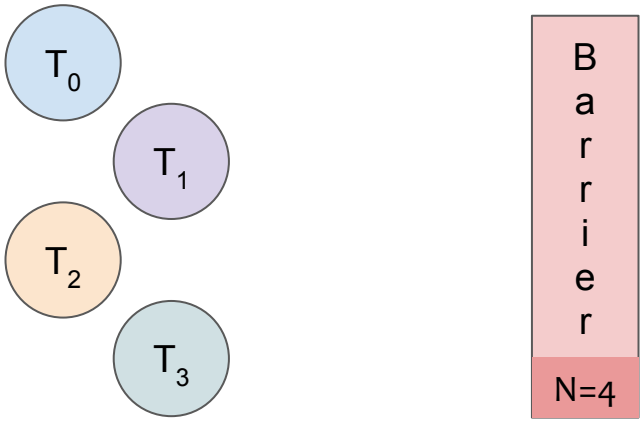
Release a permit [K += 1] (*signal*)

Barriers

A barrier allows to wait for a given number of threads to wait for each other at a certain point in the program

It has only one primitive:

- *Wait*: for a barrier waiting for N threads, wait until N threads have called *wait()* on the barrier

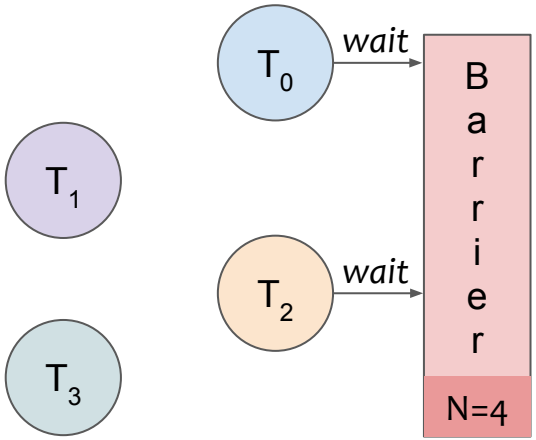


Barriers

A barrier allows to wait for a given number of threads to wait for each other at a certain point in the program

It has only one primitive:

- *Wait*: for a barrier waiting for N threads, wait until N threads have called *wait()* on the barrier

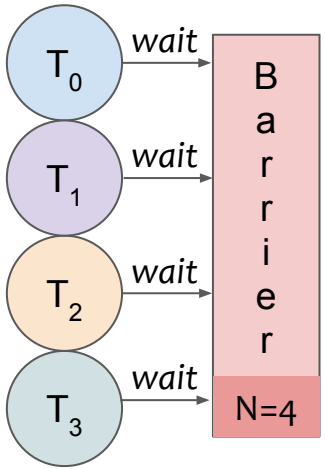


Barriers

A barrier allows to wait for a given number of threads to wait for each other at a certain point in the program

It has only one primitive:

- *Wait*: for a barrier waiting for N threads, wait until N threads have called *wait()* on the barrier

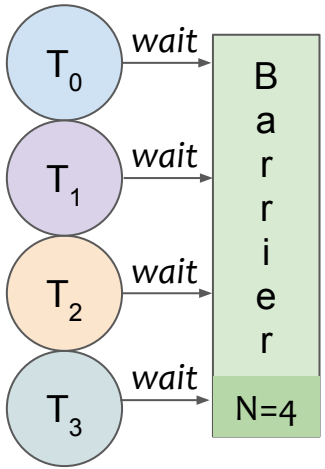


Barriers

A barrier allows to wait for a given number of threads to wait for each other at a certain point in the program

It has only one primitive:

- *Wait*: for a barrier waiting for N threads, wait until N threads have called *wait()* on the barrier

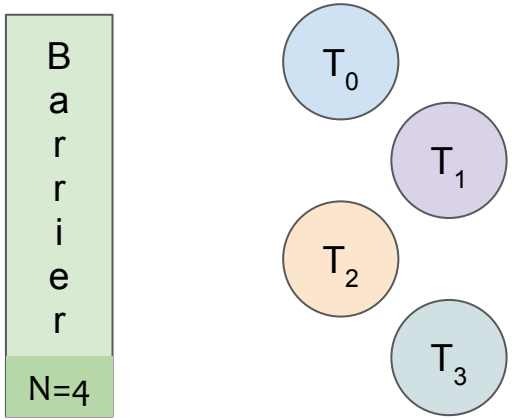


Barriers

A barrier allows to wait for a given number of threads to wait for each other at a certain point in the program

It has only one primitive:

- *Wait*: for a barrier waiting for N threads, wait until N threads have called *wait()* on the barrier



Barriers in Java

Class **CyclicBarrier**

`CyclicBarrier(int parties)`

Create a barrier waiting for *parties* threads

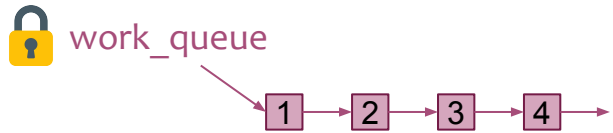
`void await()`

Wait on the barrier until *parties* threads are waiting

Back to our shared work queue

Now, let's go back to our web server and its work queue and use our new tools

How can we make our linked list safe to use concurrently?



We could add a **mutex lock** to only allow one thread to access the list at a time

Main thread

```
1    while (true) {
2        req = waitForRequest();
3        work_queue.push(req);
4    }
```

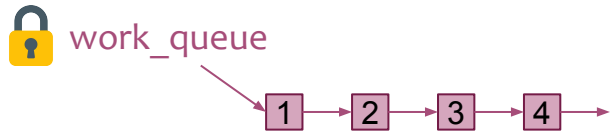
Worker thread

```
1    while (true) {
2        waitUntilWorkAvailable();
3        r = work_queue.pop();
4        process(r);
5    }
```

Back to our shared work queue

Now, let's go back to our web server and its work queue and use our new tools

How can we make our linked list safe to use concurrently?



We could add a **mutex lock** to only allow one thread to access the list at a time

Main thread

```
1    while (true) {
2        req = waitForRequest();
3        work_queue.lock();
4        work_queue.push(req);
5        work_queue.unlock();
6    }
```

Worker thread

```
1    while (true) {
2        work_queue.lock();
3        waitUntilWorkAvailable();
4        r = work_queue.pop();
5        work_queue.unlock();
6        process(r);
7    }
```

Another approach to data sharing

If we have a lot of worker threads, we might have contention on our work queue

→ a large number of threads competing for the same mutex lock

Is there a way to minimize contention on our work queue and spend less time doing synchronization?

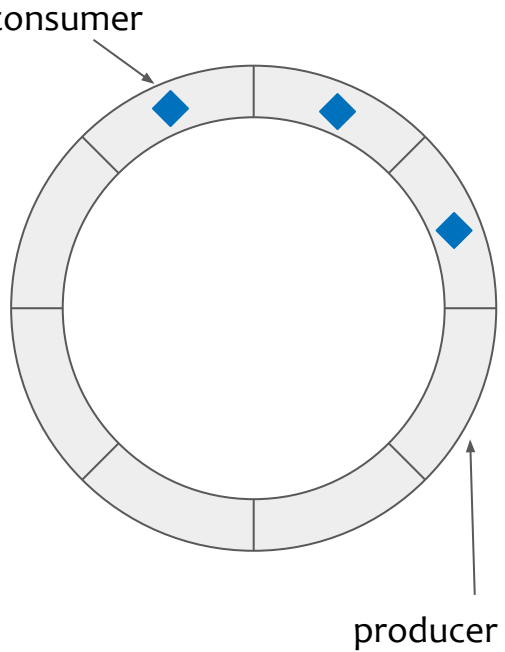
Lock-free data structure

We can use a **lock-free** data structure

i.e., a data structure that does not require locks

With our work queue, we can achieve this with two changes:

- Each worker thread has their own work queue
- Work queues now have one producer (main thread) and one consumer (worker), which means we can avoid heavy synchronization by implementing the queue in a smart way, *e.g. with a ring buffer*



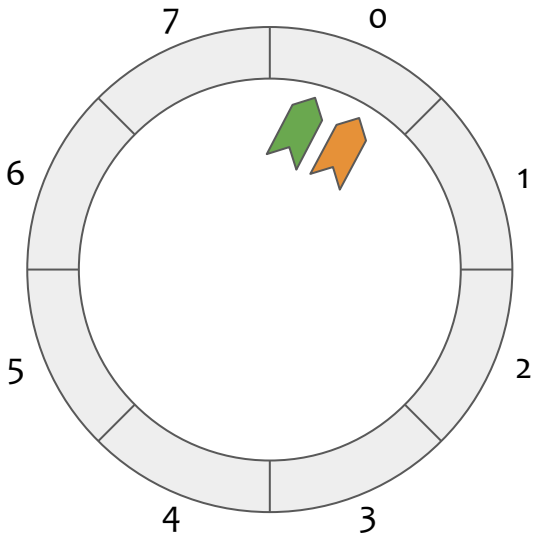
Lock-free ring buffer

Let's design a lock-free thread safe data structure!

The **ring buffer** (aka *circular buffer*) is an easy way of solving the producer-consumer problem we have (main produces, worker consumes)

We need four elements:

- An array
- The capacity of the array
- Producer index
- Consumer index

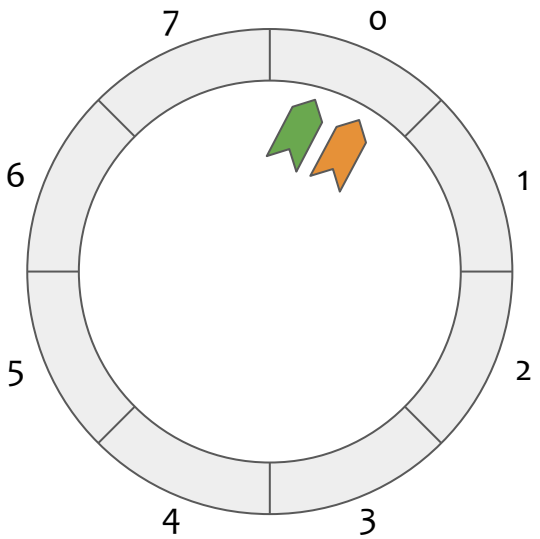


Lock-free ring buffer

```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}
```



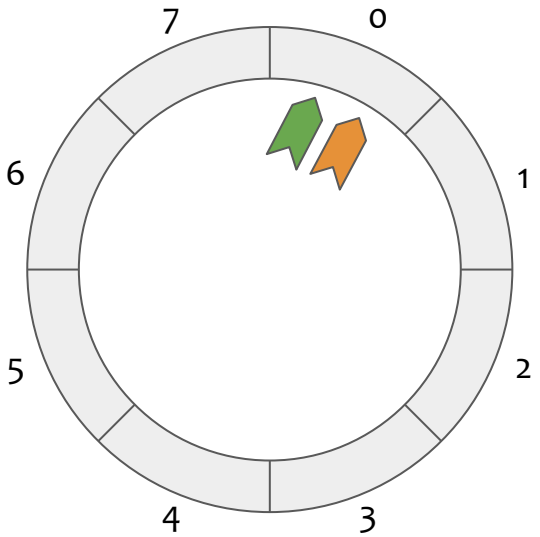
Lock-free ring buffer

```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}
```

Check if buffer is full

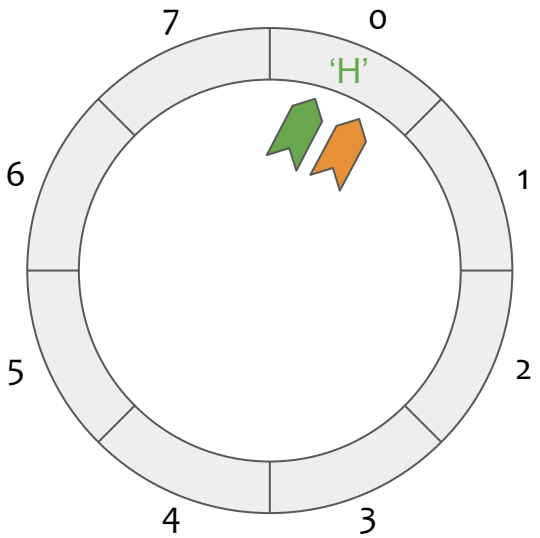


Lock-free ring buffer

```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}
```

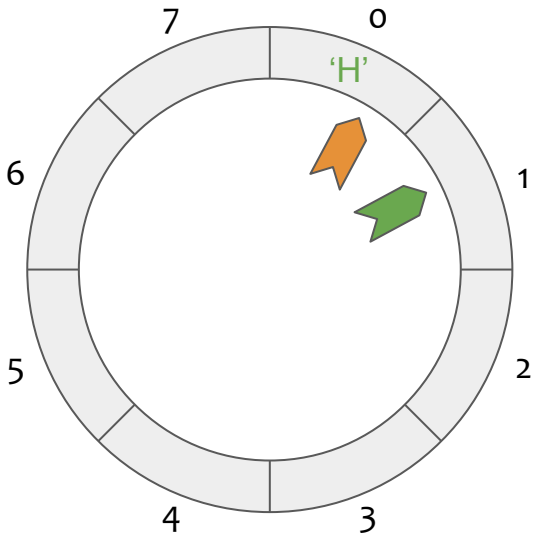


Lock-free ring buffer

```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}
```



Lock-free ring buffer

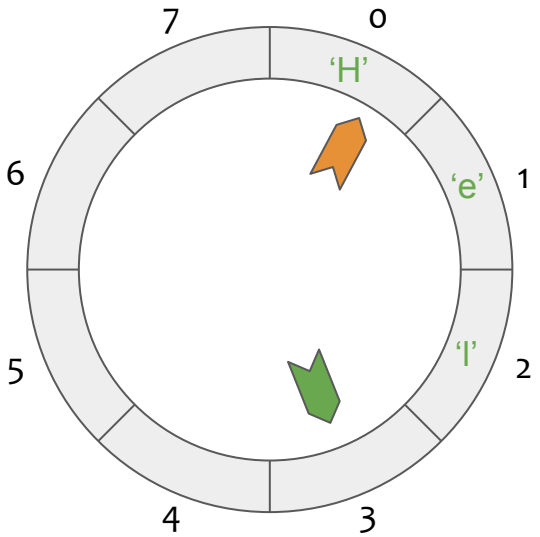
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}

char pop() {
     if (prod_idx == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     char c = array[cons_idx];
     cons_idx = (cons_idx + 1) % capacity;
     return c;
}
```



Lock-free ring buffer

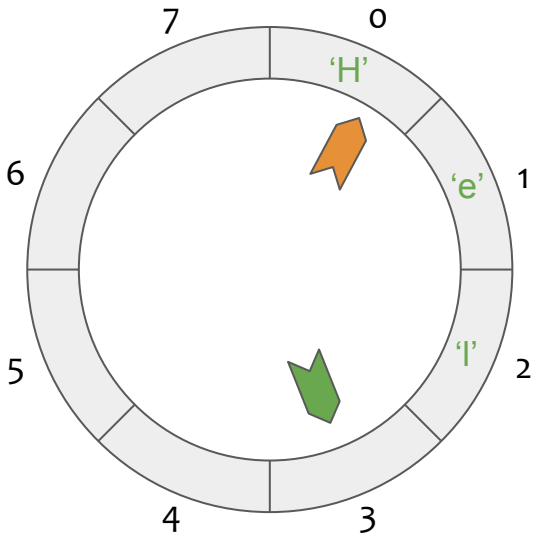
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}

char pop() {
     if (prod_idx == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     char c = array[cons_idx];
     cons_idx = (cons_idx + 1) % capacity;
     return c;
}
```



Check if buffer is empty

Lock-free ring buffer

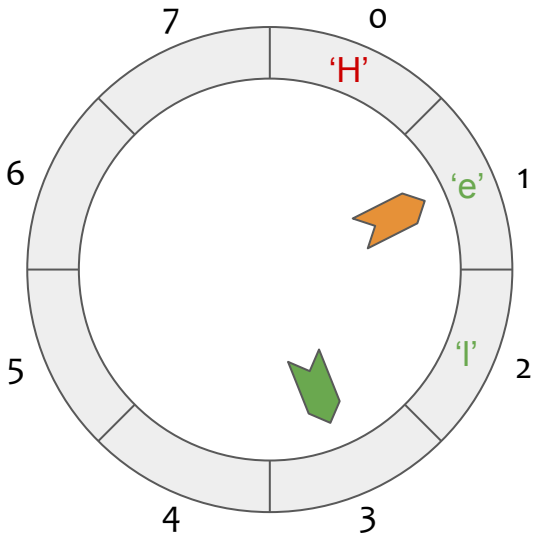
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}

char pop() {
     if (prod_idx == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     char c = array[cons_idx];
     cons_idx = (cons_idx + 1) % capacity;
     return c;
}
```



Lock-free ring buffer

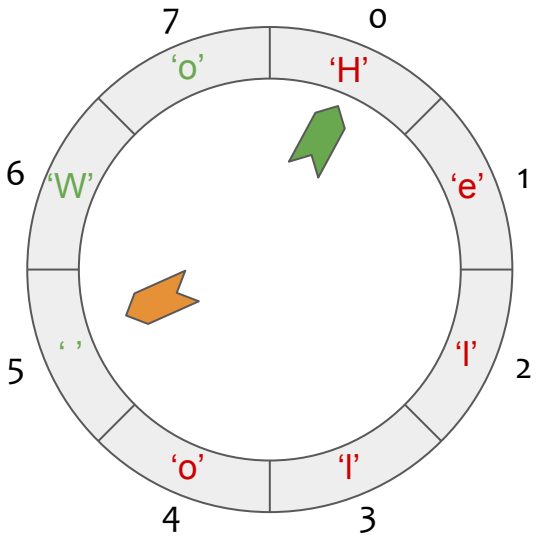
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

void push(char c) {
     if ((prod_idx + 1) % capacity == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     array[prod_idx] = c;
     prod_idx = (prod_idx + 1) % capacity;
}

char pop() {
     if (prod_idx == cons_idx)
          throw new ArrayIndexOutOfBoundsException();

     char c = array[cons_idx];
     cons_idx = (cons_idx + 1) % capacity;
     return c;
}
```



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.

Classical synchronization problems

- Producer-consumer problem, aka bounded buffer problem
  → See our web server work queue
- Sleeping barber
- Dining philosophers

We'll just quickly present the barber problem here, you can discuss solutions during tutorials, and implement them as practice exercises

Dining philosophers problem

Five philosophers dine together on a round table

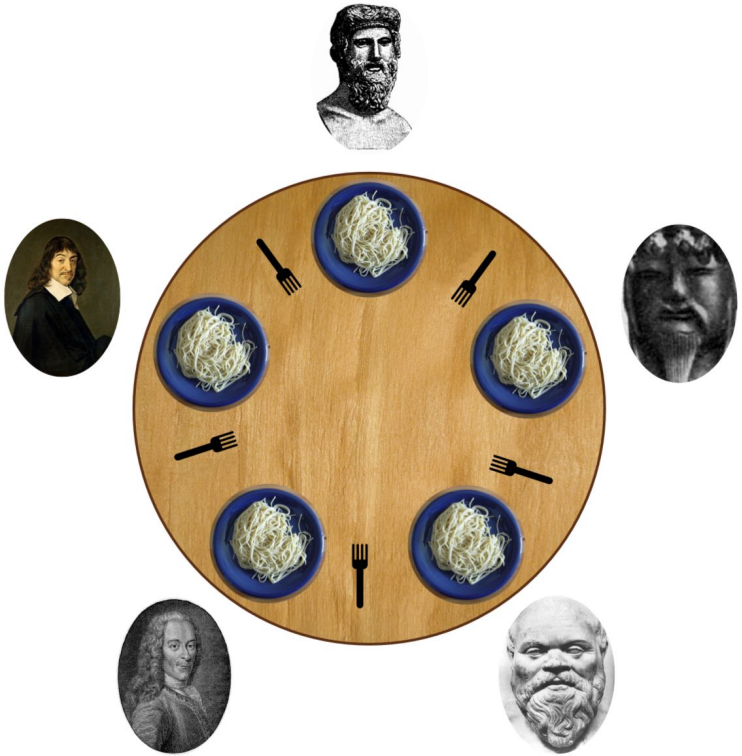
Each philosopher has a plate, and there is one fork between each plate

Each philosopher can either eat or think

In order to eat, a philosopher needs both the fork to their left and to their right

When a philosopher is done eating, they put down both forks

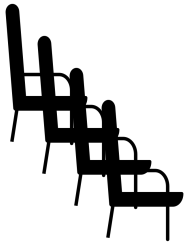
Design an algorithm that is free of starvation, deadlock and livelock



Sleeping barber problem

Imagine a barber shop with:

- One barber
- One barber chair
- *N* waiting chairs



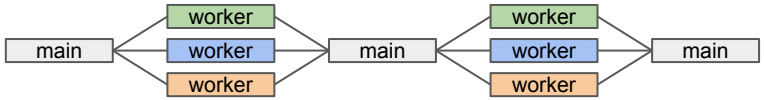
The barber shop has the following rules:

- If there are no customers, the barber sleeps in the barber chair
- A customer must wake the barber if they are asleep
- If a customer arrives while the barber is working, they go sit in an empty waiting chair if one is available, and leaves otherwise
- When the barber finishes a haircut, he checks for waiting customers. If one is available, he cuts their hair. Otherwise he falls asleep

Parallel programming patterns

Fork-join

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

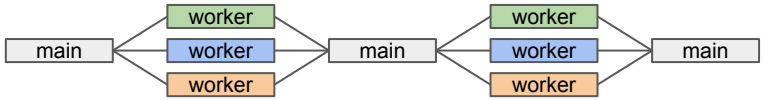


E.g., "*Spawning upon request*" version of our web server

Parallel programming patterns

Fork-join

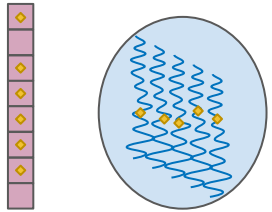
- Create threads to perform a task, *i.e.*, *fork*
- Wait for them to be done, *i.e.*, *join*



E.g., "*Spawning upon request*" version of our web server

Work stealing

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



E.g., "*Thread pool*" version of our web server

Communication paradigms

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

Polling

Active query of the resource until new data is available, synchronously

E.g., when waiting for a network packet:

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

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);
```

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

- Blocking access until the resource is available

  E.g., most basic IO functions are blocking

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

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

- Blocking access until the resource is available

  E.g., most basic IO functions are blocking

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

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

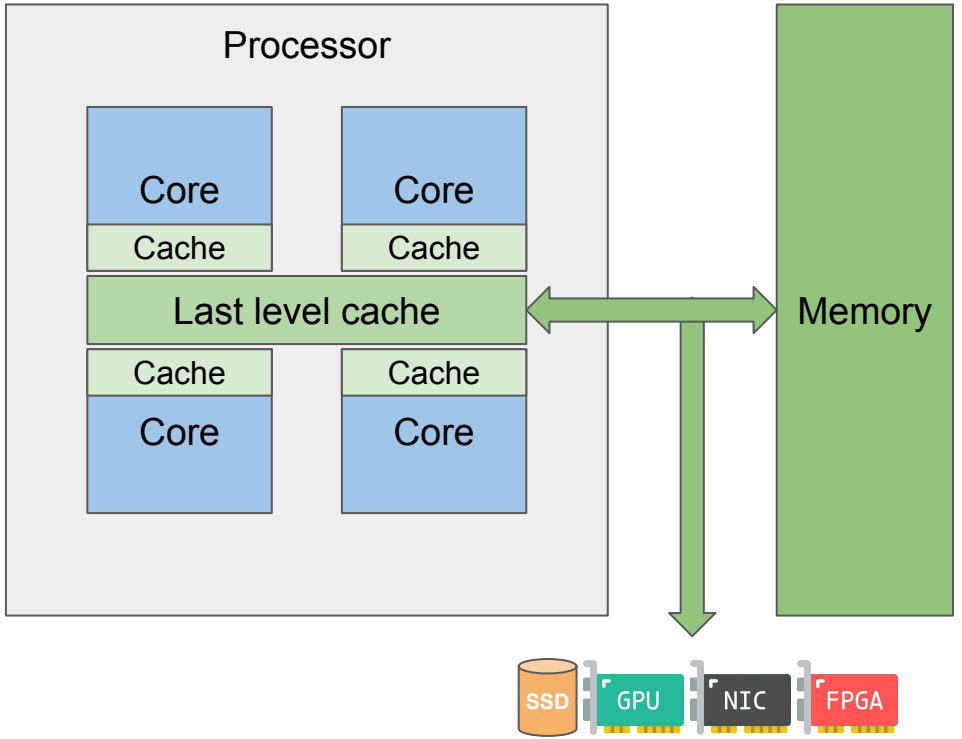
- Register to be notified when the resource is available

  E.g., receive a signal

- Poll the resource between the processing of other tasks

Moving away from the CPU

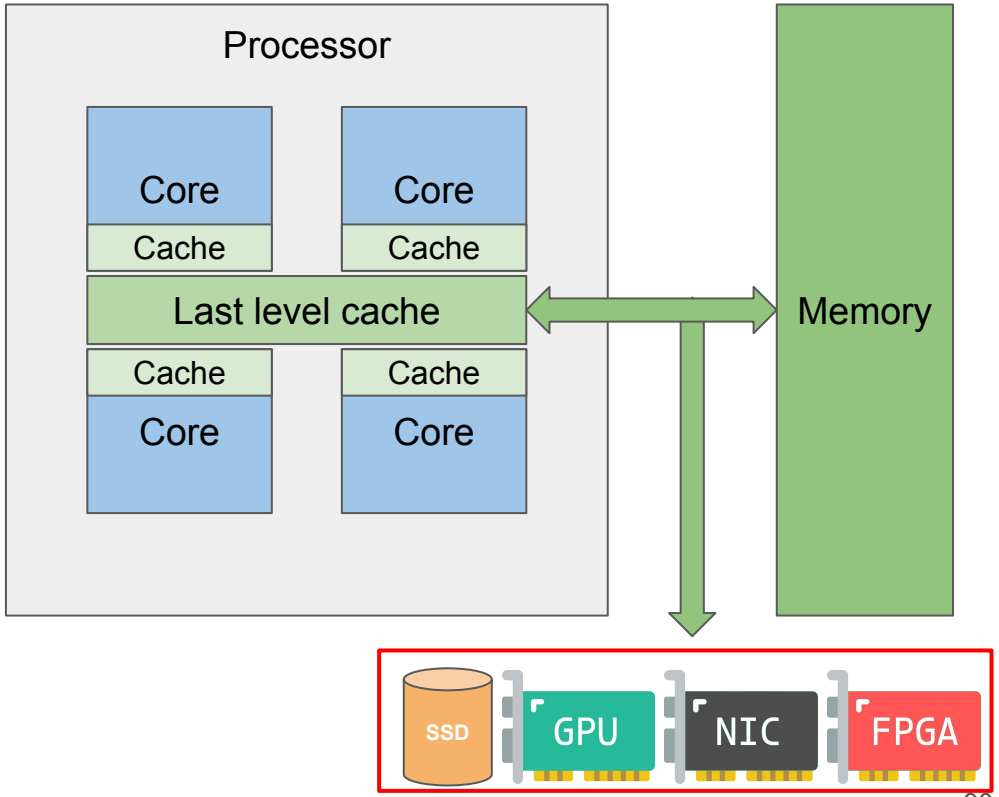
Now that we have seen how to use the parallel processing power of CPUs, let's take a step back.



Moving away from the CPU

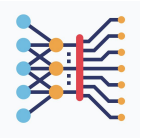
Now that we have seen how to use the parallel processing power of CPUs, let's take a step back.

Let's have a quick look at how we can use the rest of the computer to accelerate computation!



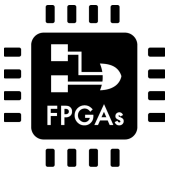
The need for high performance computing

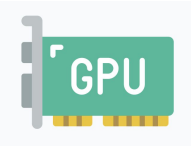
- The rise of AI is powered by **large-scale data-driven learning**
  - To meet the computational requirements of these modern workloads, we need **high-performance computing**



- ***Only* CPU-centric computing is still limiting**
  - We need large numbers of high performance cores!
  - Led to the rise of accelerators for compute-intensive tasks

- **Accelerated computing systems** offer massively parallel cores
  - Graphic processing units (GPUs)
  - Field-programmable gate arrays (FPGAs)
  - Tensor processing units (TPUs) or specialized AI accelerators

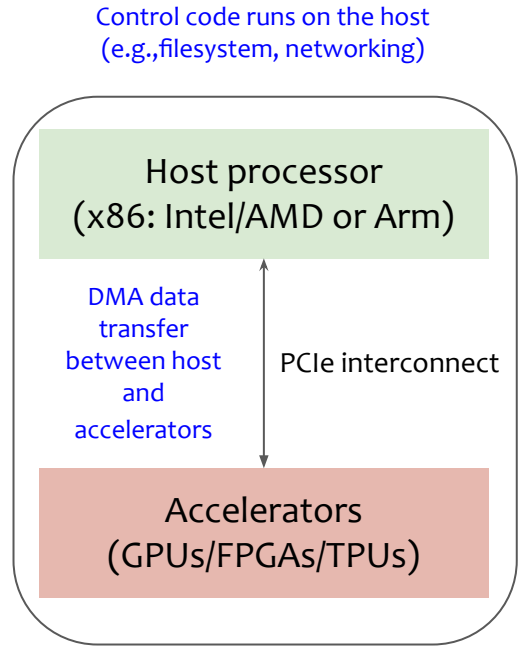




TPUs or
Specialized AI accelerators

Accelerated computing systems

- **Accelerated computing** is a modern style of computing that separates the data-intensive parts of an application and processes them on a separate acceleration device, while leaving the control functionality to be processed on the CPU
  - Having separate types of hardware processors, including accelerators, is known as **heterogeneous computing** because there are multiple types of compute resources available for the application to utilize



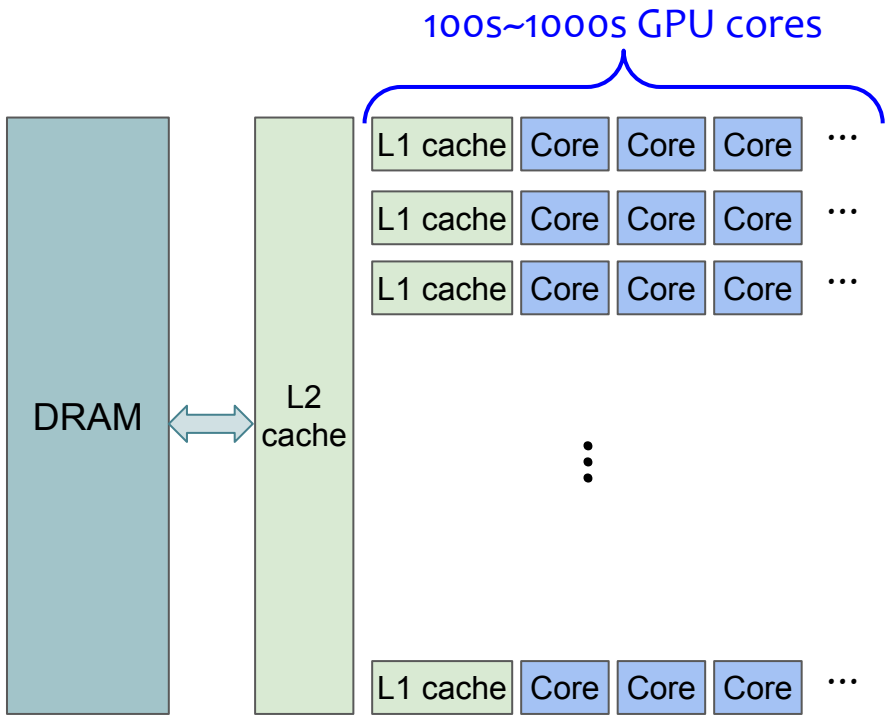
A server architecture w/ accelerators

https://www.xilinx.com/applications/adaptive-computing/what-is-accelerated-computing-and-why-is-it-important.html

GPU architecture

GPUs offer a larger number of "general-purpose" cores

- **Single Instruction Multiple Data** (SIMD): all cores execute the same instruction on different data in parallel
- **GPUs have their own memory system**: a main memory, a shared L2 cache, and private L1 caches (across a set of cores)



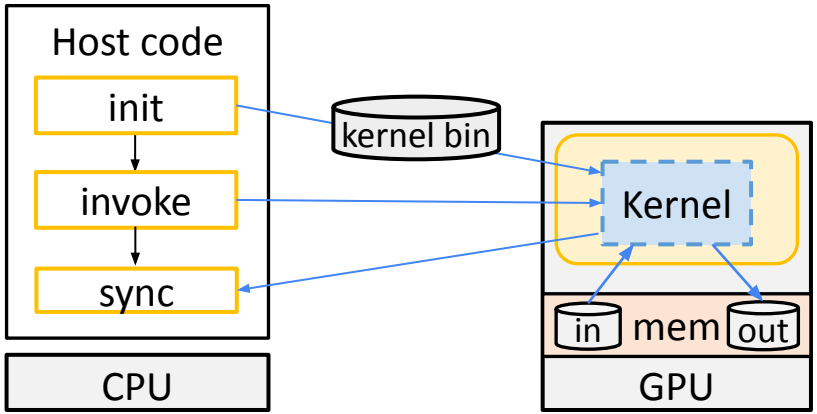
GPU programming model

CPU-driven task execution on GPU

- ***Host code*** running on CPU is responsible for GPU initialization, data transfer, and synchronization
- ***Kernel code*** represents the accelerated task and is loaded to GPU by host code

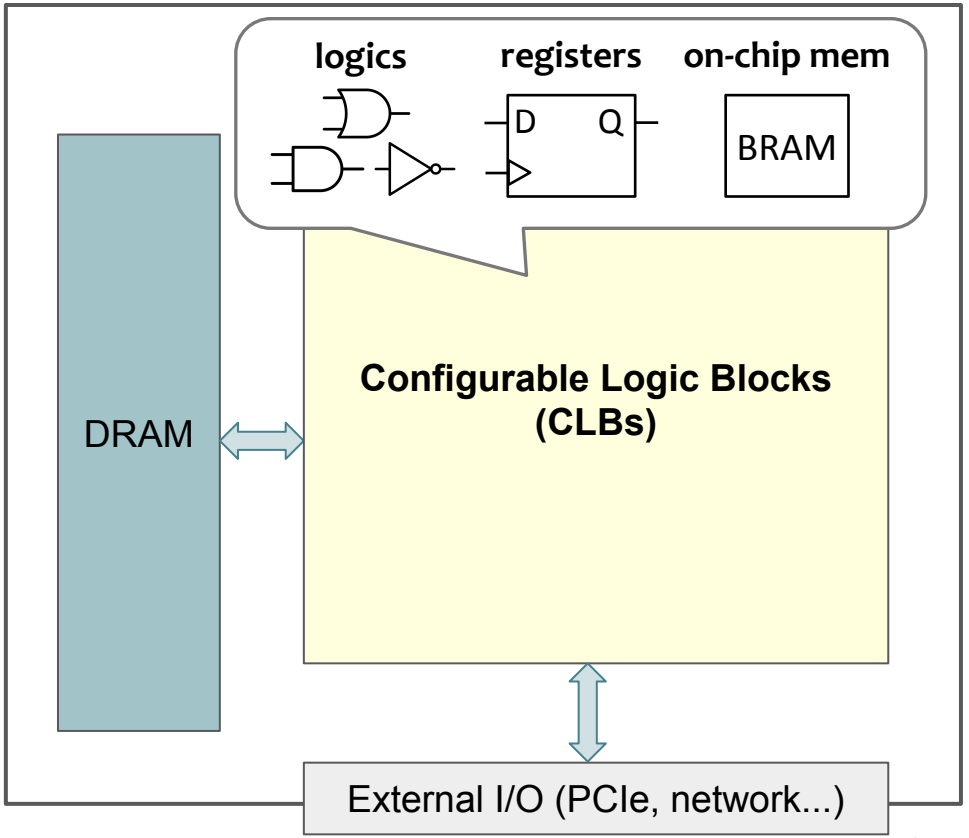
Execution flow (CPU side)

1. Initialize GPU with kernel's binary
2. Write inputs to GPU memory
3. Invoke the execution on GPU
4. Wait for a completion
5. Read outputs from GPU memory



FPGA architecture

- FPGAs help us to synthesize a specialized compute logic directly on hardware, i.e., the hardware is specialized to process a specific compute logic
- FPGAs consist of electrical logics (LUTs), registers, memories, and I/O interconnects
- FPGA can configure any custom hardware by loading ***bitstream,*** a file that describes logics, routing, initial values of register/memory
- Bitstreams are generated by synthesizing a hardware design written in ***Hardware Description Language (HDL)***

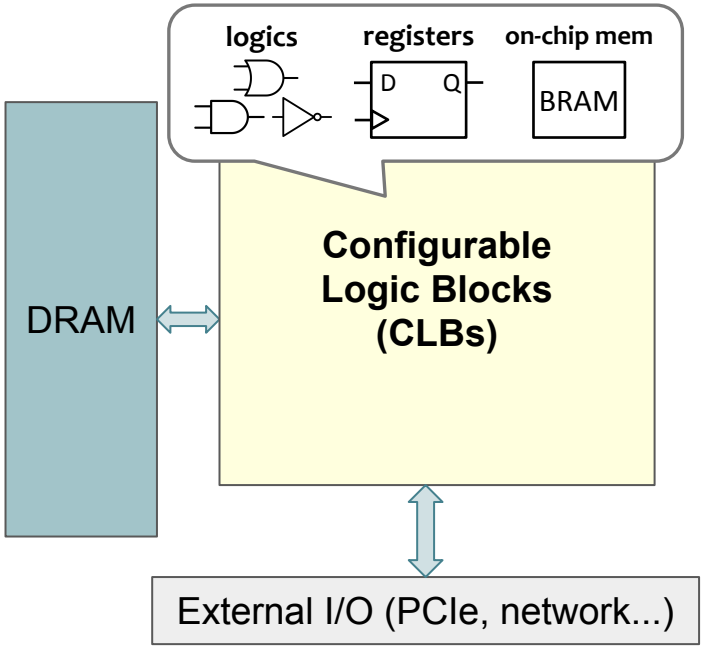


FPGA architecture

A **Field-Programmable Gate Architecture** (FPGA) is an integrated circuit that can be programmed by users with their own circuit (called *bitstream*)

Bitstreams are specified with a *hardware description language* (HDL) or with higher level languages through libraries, and then synthesized on the FPGA

FPGAs allow users to have a hardware implementation of their program, providing high performance and energy efficiency



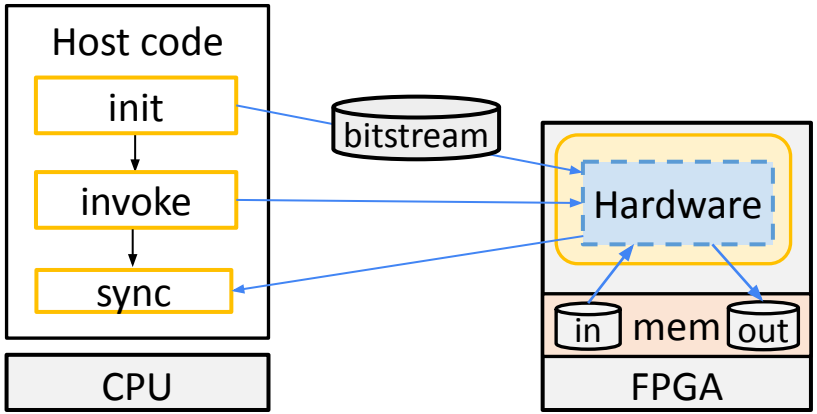
FPGA programming model

FPGA reconfiguration

- FPGA can adopt host-driven task execution like GPU
- Kernel code is written in HDL and represents ***hardware modules*** reconfigured on FPGA.

Execution flow (CPU side)

1. Program FPGA with kernel's bitstream
2. Write inputs to FPGA memory
3. Invoke the execution on FPGA
4. Wait for a completion
5. Read outputs from FPGA memory



Outline

- ~~**Part I:** Concurrency (or Scale Up!)~~
- **Part II: Scalability (or Scale Out!)**
  - Scalability challenges
  - Scalability techniques:
    - Load balancing
    - Replication
    - Sharding
    - Secondary indexing
    - MapReduce
- **Part III:** Pattern implementation

Limitations of using one machine

Multi-threaded parallel programming:

- Use all the cores in the same machine
- Threads communicate using shared memory
- Scale-up (vertical scaling): if you need more resources, switch to a bigger machine

Problems with scale-up:

- Need machine with 2x the cores and 2x the RAM => more than 2x the cost
- Large enough scale => eventually you simply cannot buy a big enough machine
- Need to reboot machine for maintenance => service outage
- Machine in one location, users around the world => high latency for remote users

Using many machines instead of one

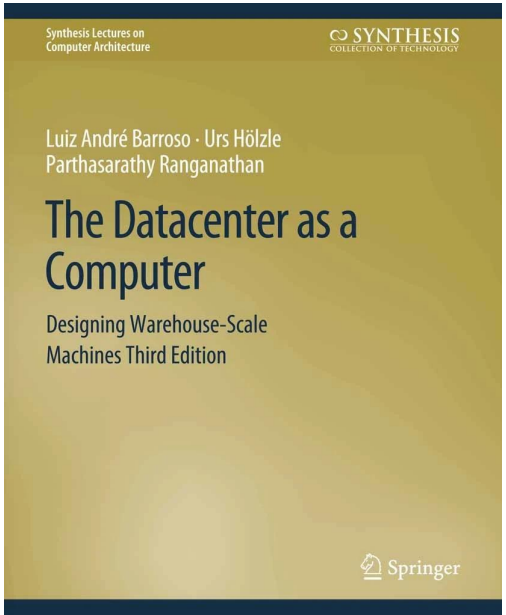
Scale-out (horizontal scaling): use multiple smaller machines instead of one big one

- 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

Many machines working together



Challenges of scale-out

At large scale, **something** is constantly broken:

- For example, approx. 2–5% of hard disks fail per year
- => If you have 10,000 disks, expect on average one disk failure per day
- Replace failed disk in 1 day => system is in a degraded state most of the time
- Lower (but still significant) failure rates for CPUs, RAM, power supplies, etc.

Fault tolerance: even if some components are broken, the service should still work

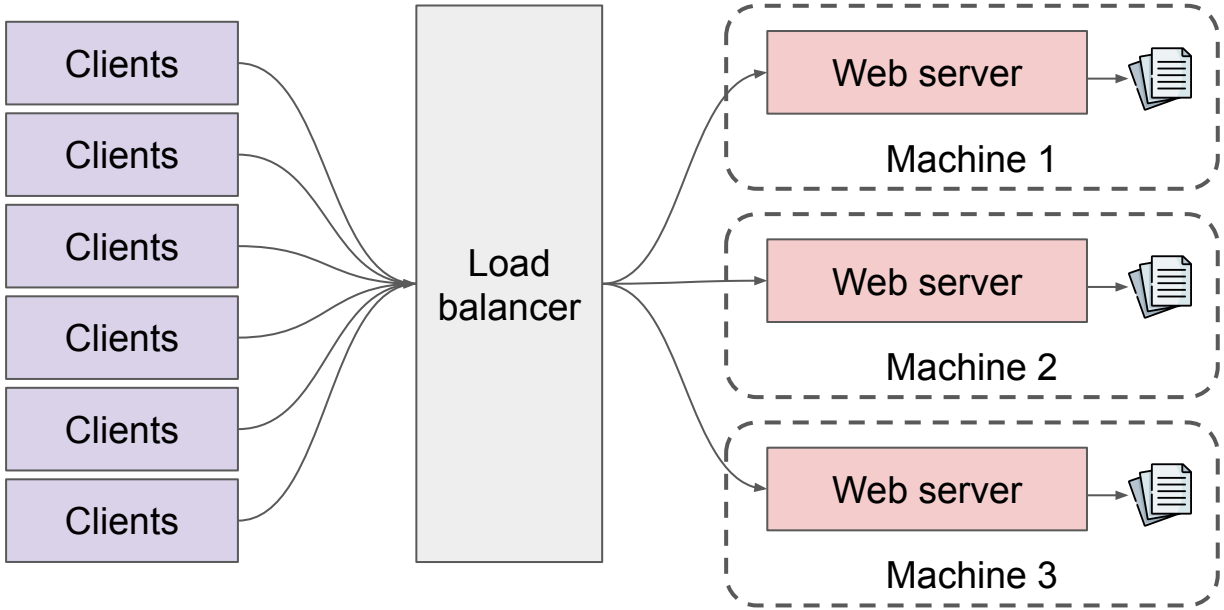
Techniques for fault tolerance are developed in the study of **distributed systems:**

- Handling some machines being unavailable, or network interruptions
- Difficult to get right: **see TUM master's course IN2259 on distributed systems**

Web servers for static files

Example that is easy to scale out: **serving web requests for static files**

Any machine can handle any request => **load balancer forwards requests to any server**



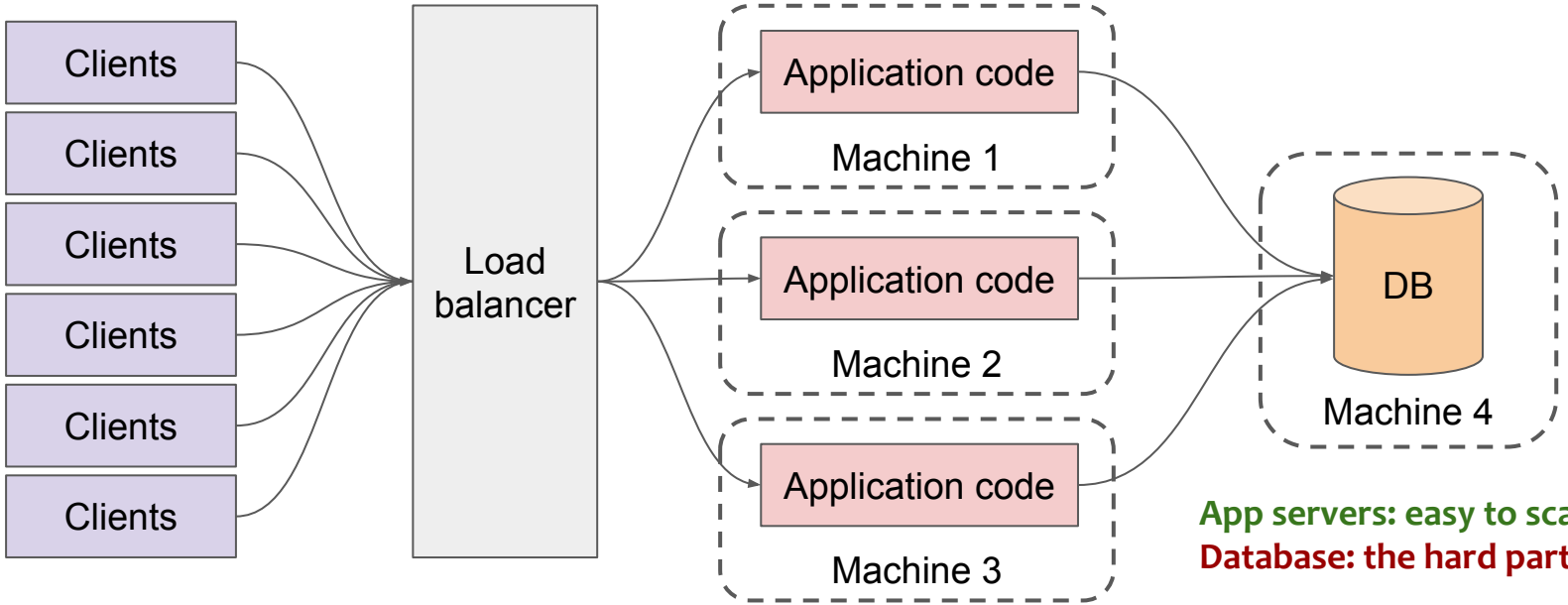
Each machine has a copy of the files that it is serving. As long as the files don't change, it's easy to add more web server machines.

Use e.g. Kubernetes for managing servers.

kubernetes

A stateless cloud service

When application code receives a request, it looks up the user's state in a database. Any updates are also saved in the database. App is **stateless**: after response is sent, app code forgets everything about the request. Any app server can handle a request.



App servers: easy to scale out.
Database: the hard part!

Scaling out the data layer

Scaling out a stateless service is easy: provision more (virtual) machines, install the app on them, add them to the load balancer.

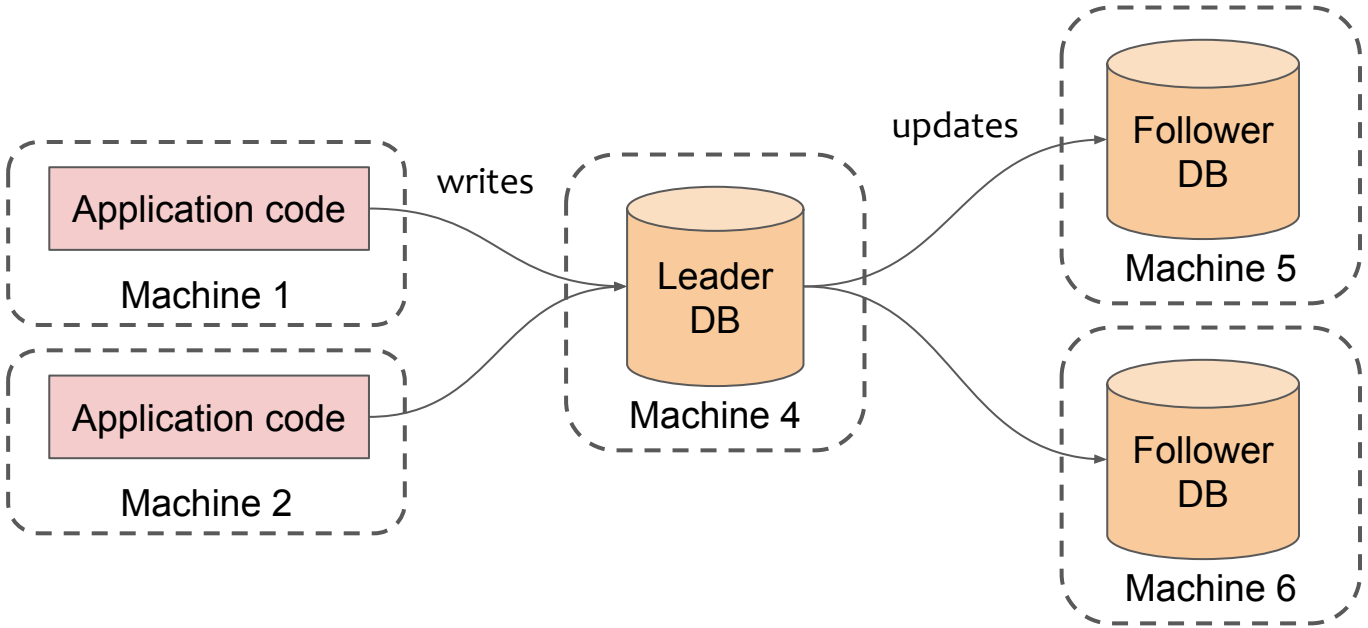
Databases and other stateful data systems are harder. Two main techniques:

1. **Replication** — maintaining copies of data on multiple machines; when the data changes, make sure all copies are updated.
2. **Sharding** (aka partitioning) — splitting a large dataset into smaller parts, so that each machine only stores some of the data (recall L04).

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

State machine replication

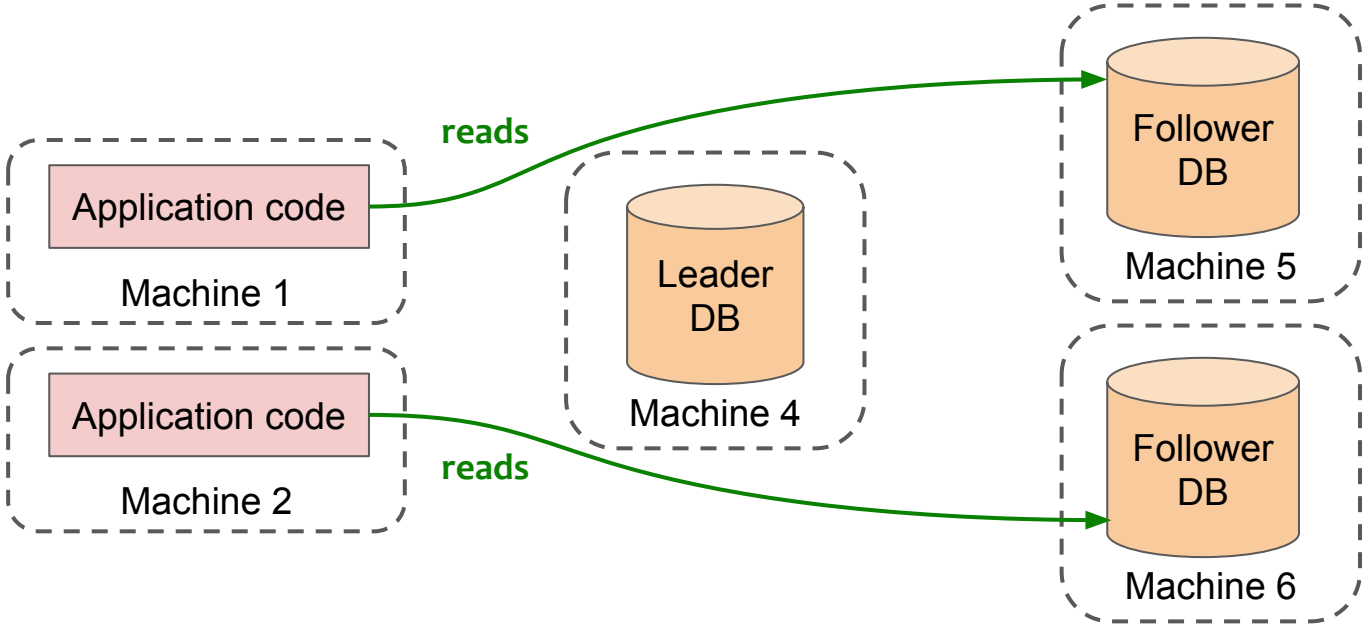
One leader, multiple followers. All database write requests are sent to the leader, who executes them and forwards the data updates on to the followers.



State machine replication (2)

Database reads can be handled by either leader or followers. If there are more reads than writes, this can reduce load on the leader.

Warning: followers might lag behind the latest updates, so reads may be stale!



When does replication help with scalability?

Replication helps:

- When **reads** are much more common than **writes** (this is often true on websites: e.g. more people read reviews than write reviews)
  - can load-balance reads across the followers
- **Fault tolerance:** if one machine experiences a hardware failure, you still have a copy of the data on the other replicas
  - if the leader fails, make one of the followers the new leader ("failover")

Replication does not help:

- If the dataset is **too big** to fit on one machine
- If the **writes** happen too fast for one machine to handle

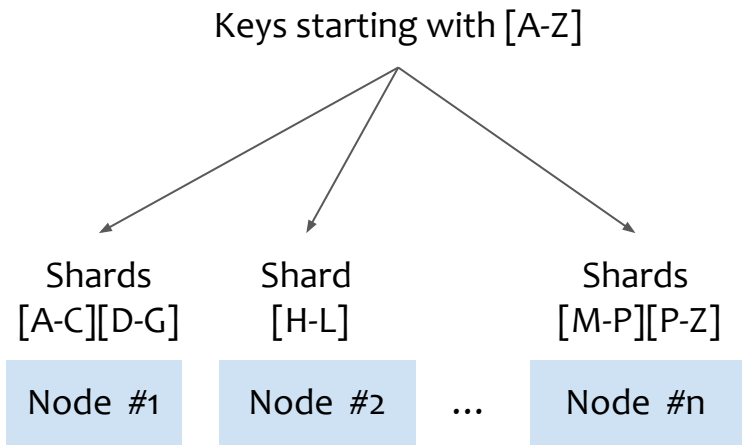
Sharding for scalability

If the dataset is too big, or the volume of writes is too high for one machine to handle:

→ Split dataset into smaller parts, and put different parts on different machines

— **but how do you decide to split?**

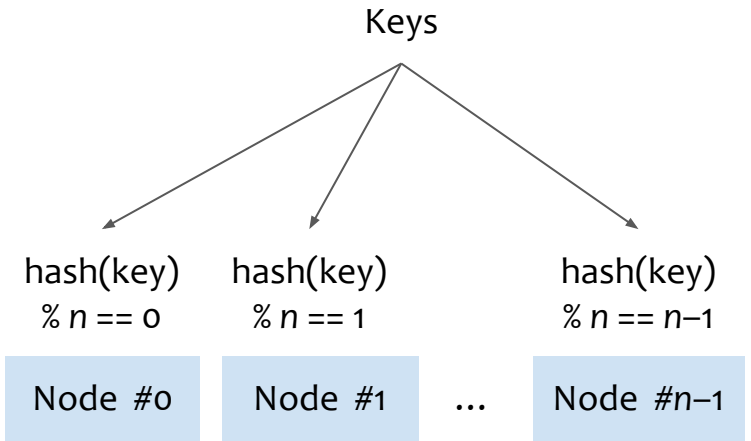
- **Example from L04:**
  In a key-value store, shard by the first letter of the key
- **Problem:** if all your keys start with "A", they will all go in shard #1 and the other shards will be idle!
- **Problem:** need to split a shard when it gets too big — how?



“Hash modulo n” sharding

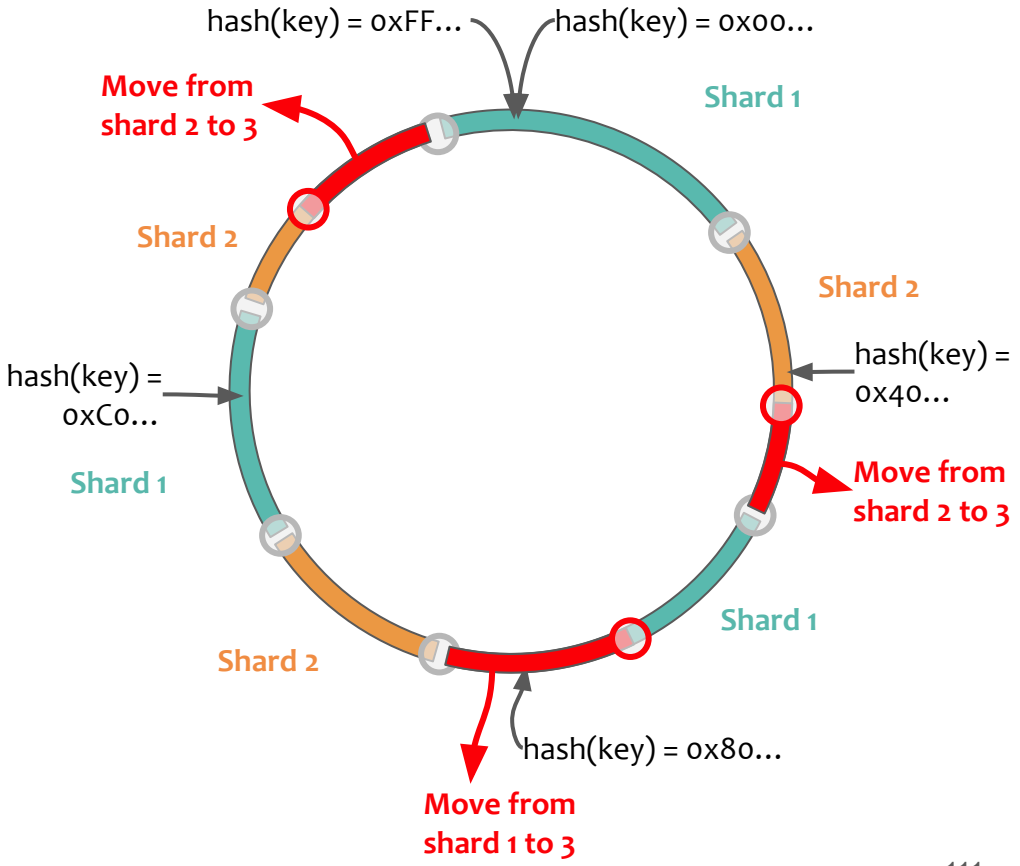
If you have *n* shards, use a hash function to map each key to an integer, then take modulo *n* to get an integer between 0 and *n*–1.

- **Pros:**
  - More uniform load distribution
- **Cons:**
  - Inefficient to find keys with the same prefix (“range query”)
  - If you add machines (*n* changes), most keys need to move to a different shard
  - Doesn’t help with “hot keys” (e.g. a celebrity on a social network)

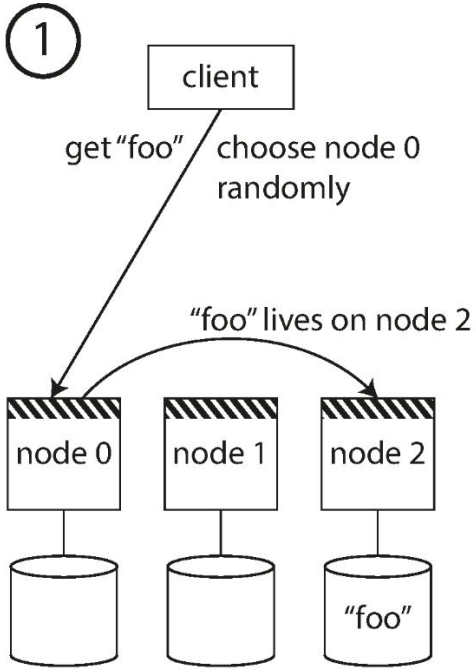


Consistent hashing

- Think of hashes as a ring
- Pick *k* random hashes per shard as boundaries (here, *k* = 3)
- All the keys that hash to a ring position between one shard's boundary and the next are stored on that shard
- To add a shard, pick *k* random boundaries and re-assign data in the new ranges to new shard
- Get even distribution with large enough *k*

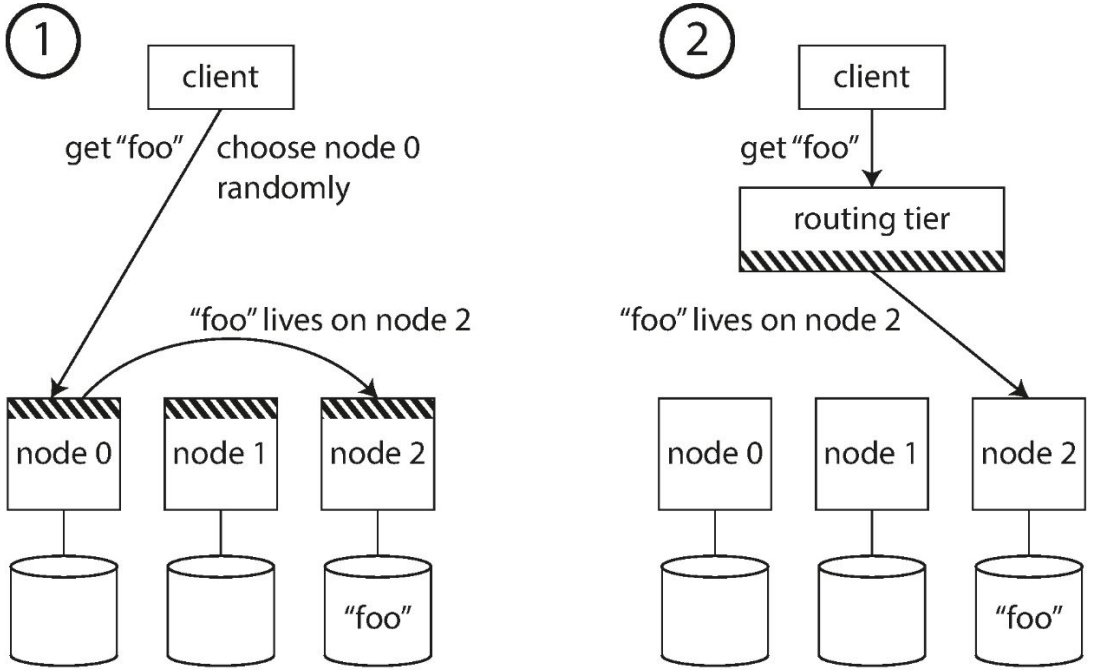


Routing queries to the correct shard



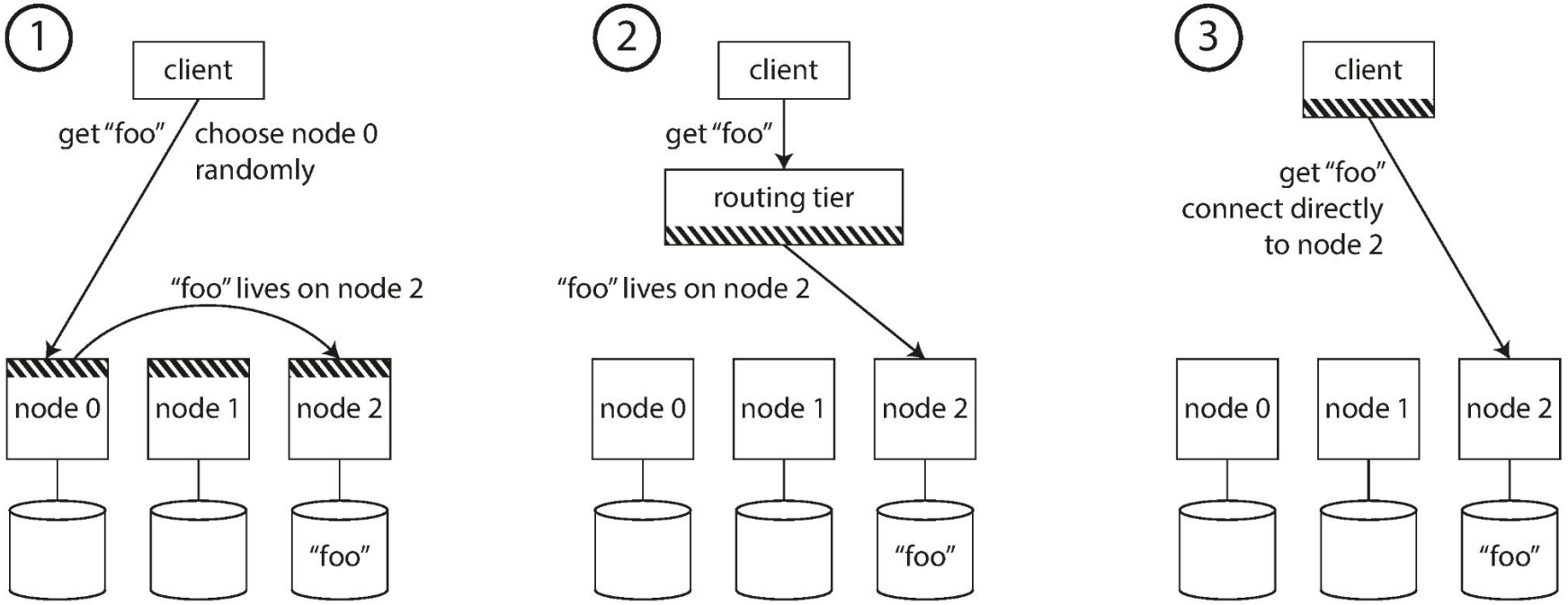
= the knowledge of which partition is assigned to which node

Routing queries to the correct shard



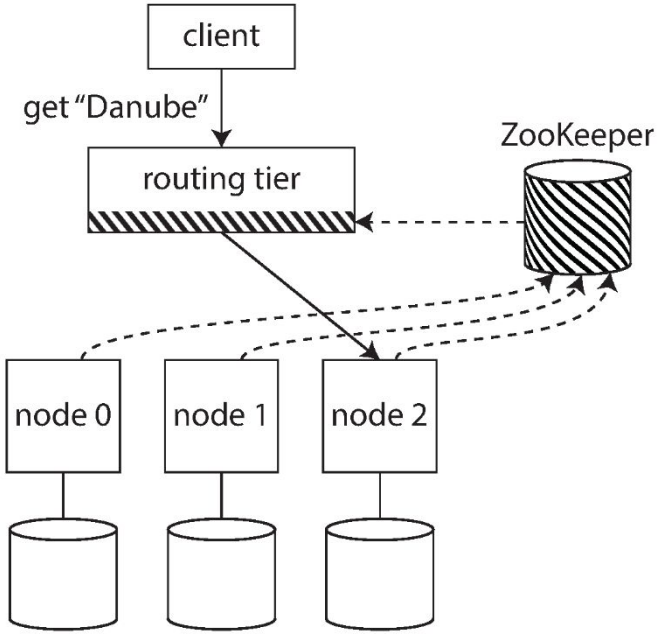
= the knowledge of which partition is assigned to which node

Routing queries to the correct shard



= the knowledge of which partition is assigned to which node

Cluster management tracks which shard is where



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

= the knowledge of which partition is assigned to which node

Sharding beyond key-value data

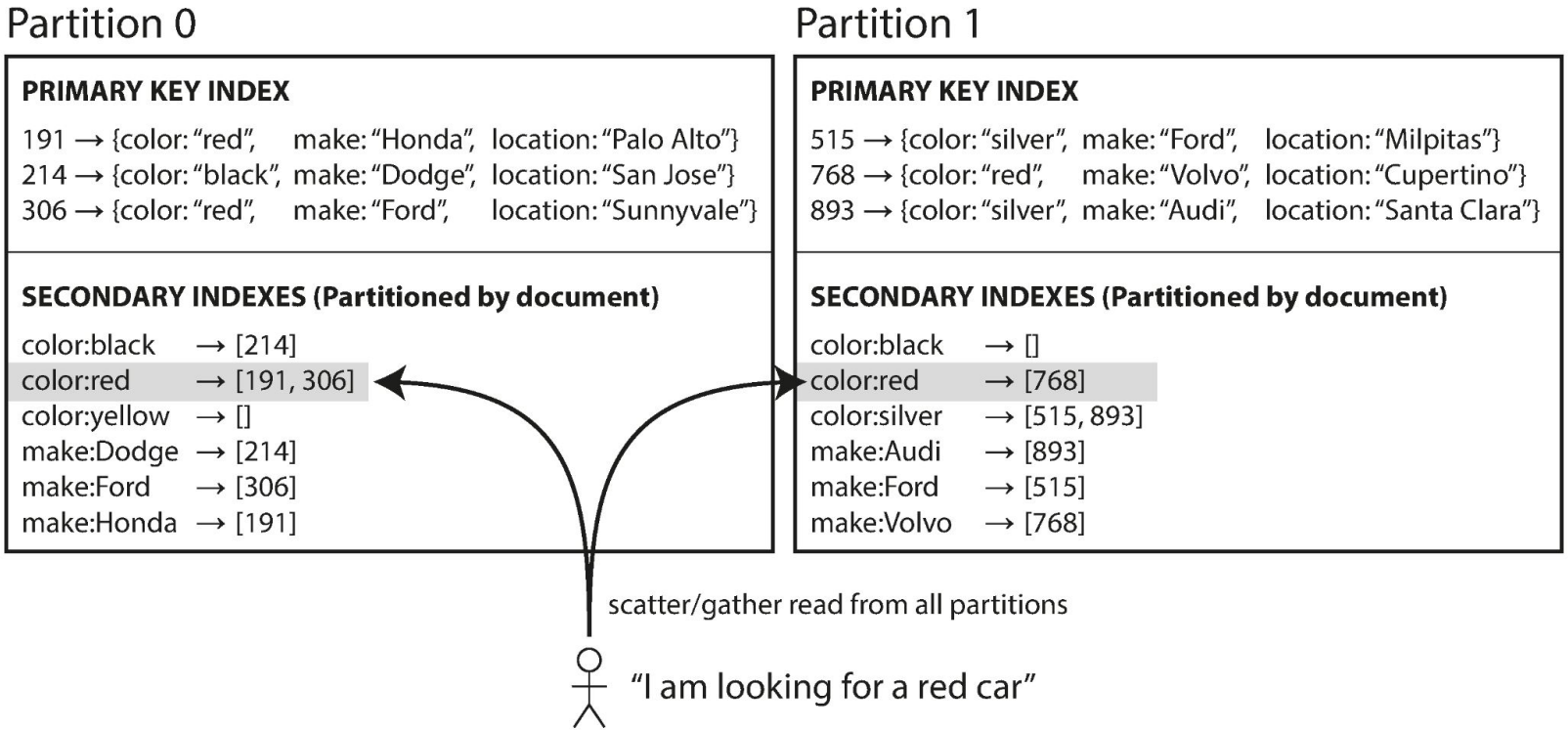
- In a key-value store, data is always accessed by key. If we shard by key, the key is sufficient information to know which shard you need to query.
- In other database types (especially relational/graph), sharding is harder. Instead of querying by primary key, it is common to use a secondary index. For example:

```
SELECT * FROM products
WHERE price < 10 AND color = 'red'
```

- How do we shard this database? By product ID? By price? By color?
- How do we execute queries that involve data from several shards?
- In a SaaS product, you can perhaps shard by user ID / customer ID (assuming each individual user/customer is small enough to fit on a single shard)

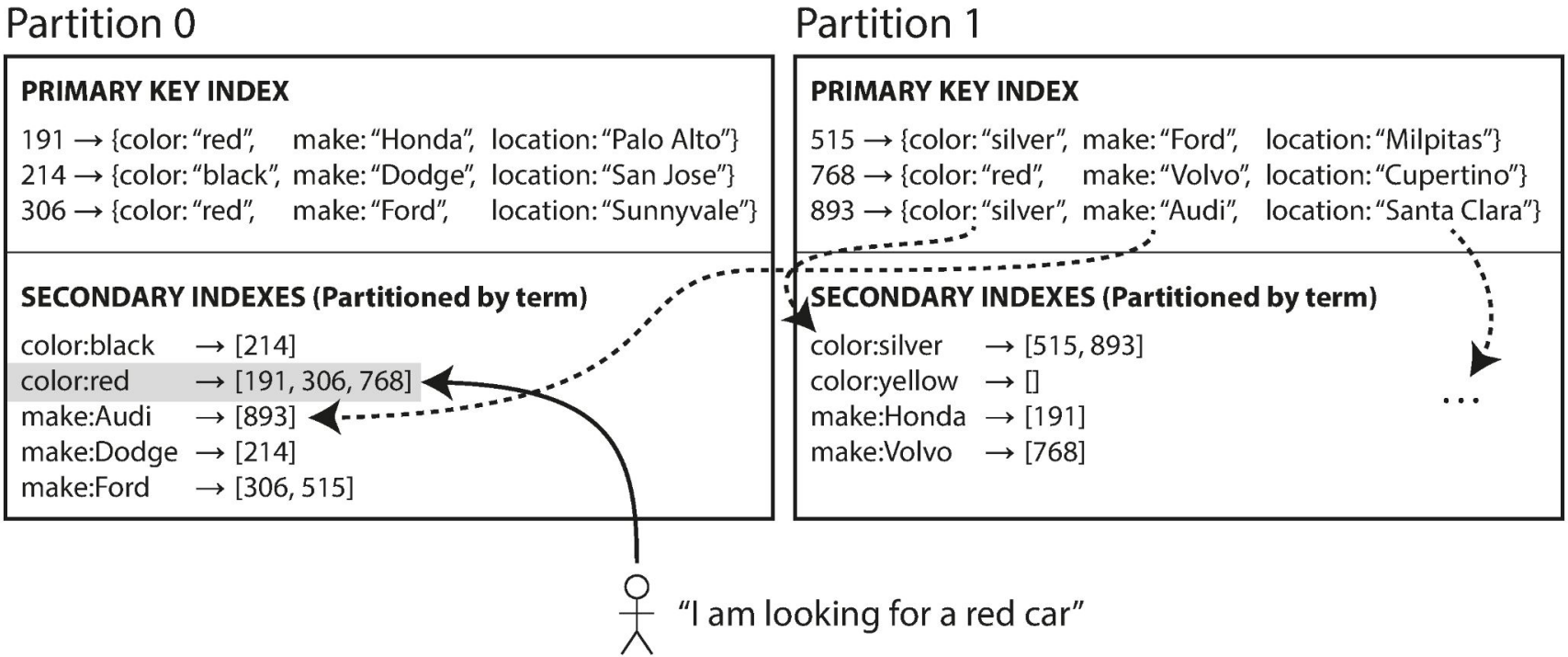
Local (document-partitioned) secondary indexes

Each shard's secondary indexes contains only data from that shard. When querying a secondary index, the query has to be sent to all of the shards.



Global (term-partitioned) secondary indexes

Secondary indexes cover data in all the shards. Now a query only needs to go to one shard, but when you write a record, you have to update indexes in multiple shards.



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.

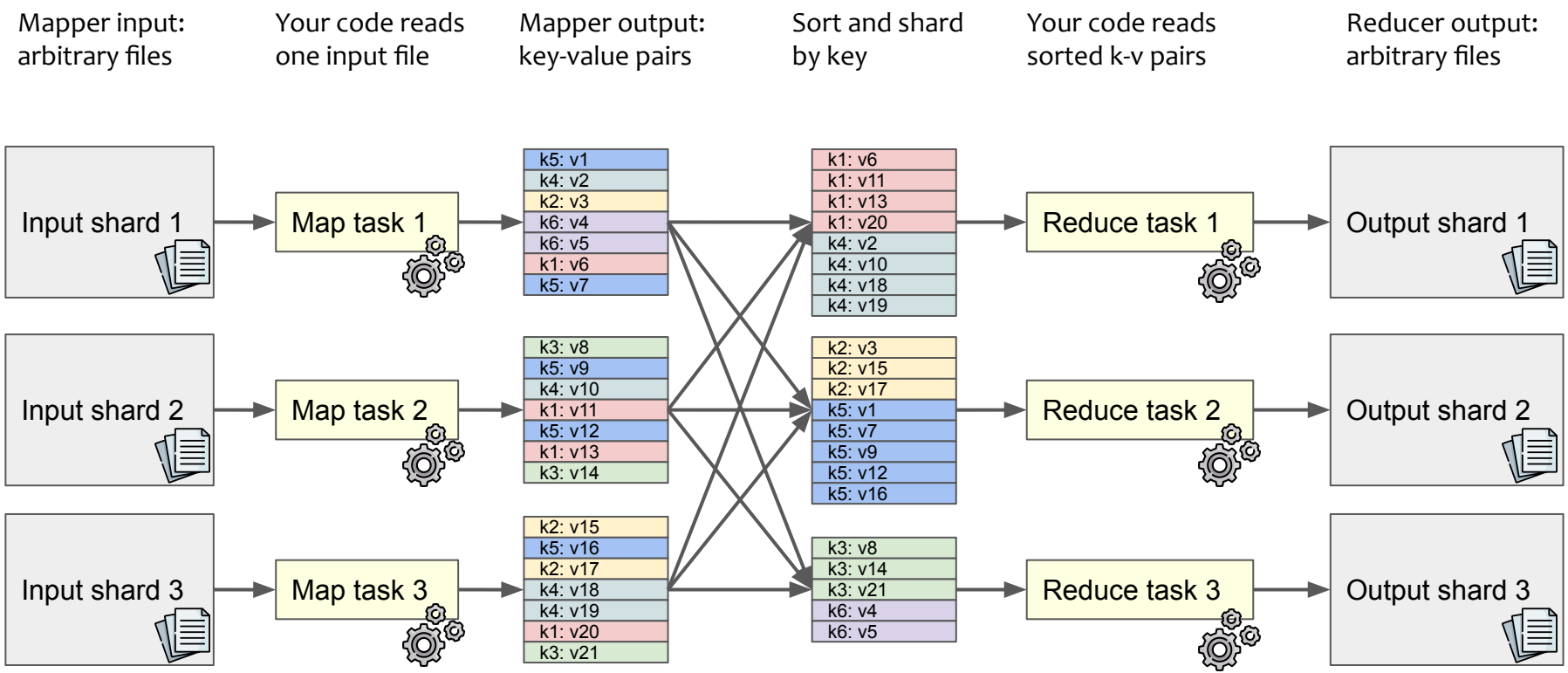
What about batch processing, when you want to **process all records**?

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



Fault tolerance in MapReduce

With a large job, there is a significant chance of one of the machines failing.
Nevertheless we want to get the correct result (= as if no failure had happened):

1. Input/output shards are replicated in distributed file system
2. If a map or reduce task crashes before it finishes, automatically restart it
3. Discard output from failed tasks, so that we don't get duplicate output

Ability to restart relies on a task having no side-effects apart from its output (which is managed by MapReduce) — in particular, **no calls to other services**

(This also maximizes throughput: each map/reduce task is single-threaded, sequentially reads input file and sequentially writes output file)

Similar approach used for low-latency stream processing, using distributed shared logs instead of distributed file system (recall L04)

Scalability summary

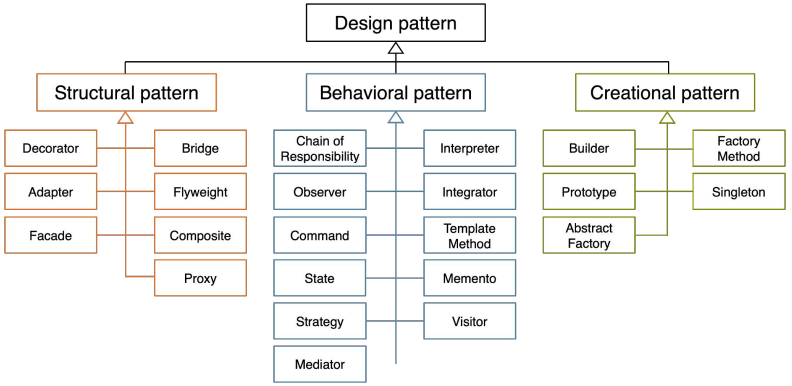
- There is no **"one true way"** to achieving scalability!
  - **Ask:** which aspect of your load is giving you the greatest problems? Amount of data? Rate of writes? Volume of reads? Expensive queries? Celebrity users?
  - **Ask:** if your load grows 10x, how can you add resources to handle it (without harming performance)?
- **Various techniques:** load balancing, replication, sharding, secondary indexing, …
- Sharding storage (file system, databases) and computation (e.g. MapReduce)
- **Word of caution!** Premature scaling is like premature optimisation
  - Only worth investing in scalability once you actually have a scaling problem
  - If you have an app with only 100 users, build it in the simplest way possible and don't worry about scalability

Outline

- **Part I: Design principle: Concurrency (or Scale Up!)**
- **Part II: Design principle: Scalability (or Scale Out!)**
- **Part III: Pattern implementation**
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

Pattern implementation

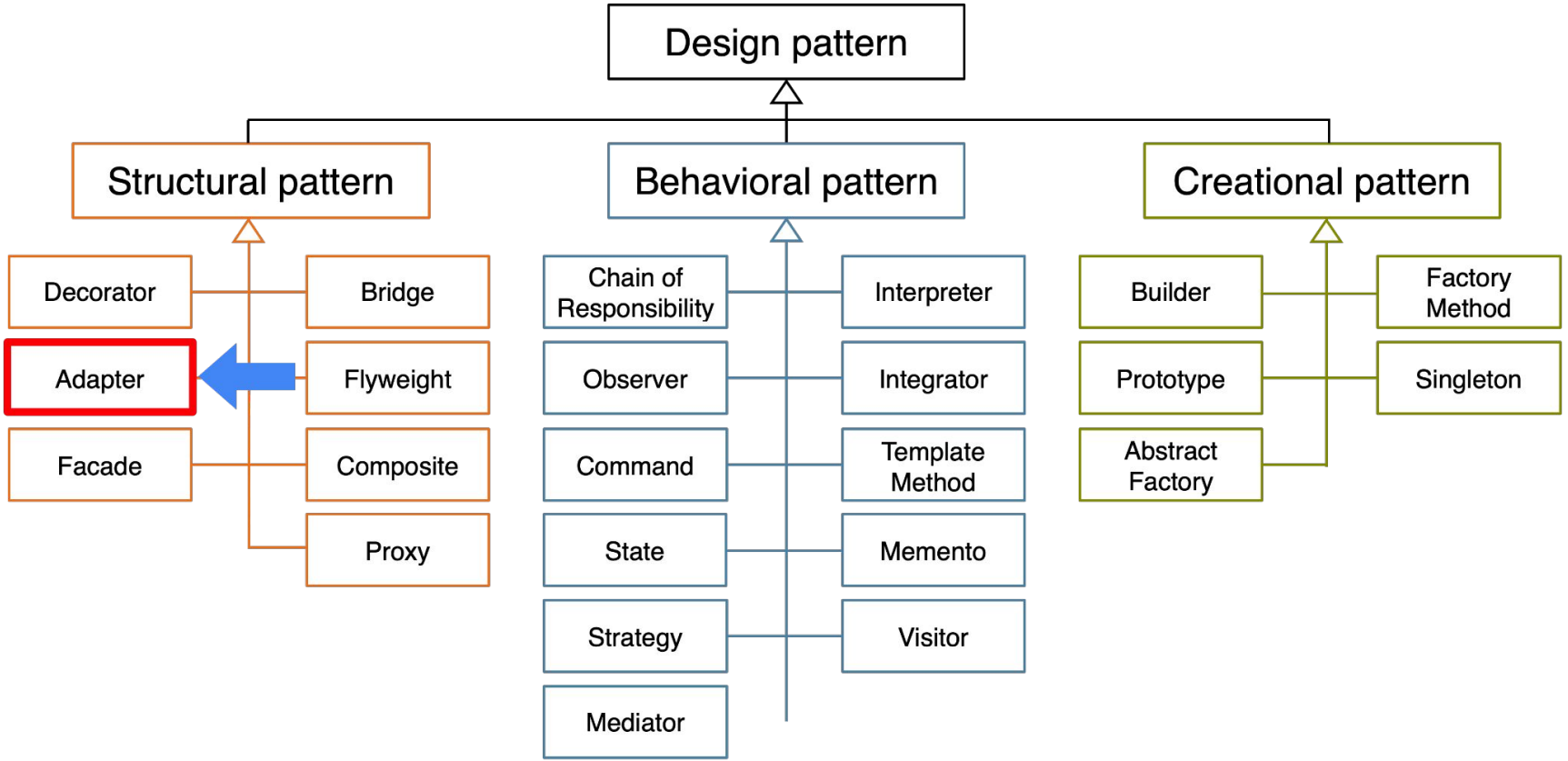
- **What are design patterns?**
  - Reusable solutions to common problems in software design
  - Best practices that can be adapted to specific situations
  - Improve code maintainability, flexibility, and extensibility
- **Design patterns taxonomy:**
  - Structural: Concerned with the composition of classes and objects
  - Behavioural: Define the ways objects interact and communicate with one another
  - Creational: Deal with the process of object creation



Outline

- ~~**Part I:** Design principle: Concurrency (or Scale Up!)~~
- ~~**Part II:** Design principle: Scalability (or Scale Out!)~~
- **Part III: Pattern implementation**
  - **Adapter pattern**
  - Observer pattern
  - Strategy pattern

Design patterns taxonomy

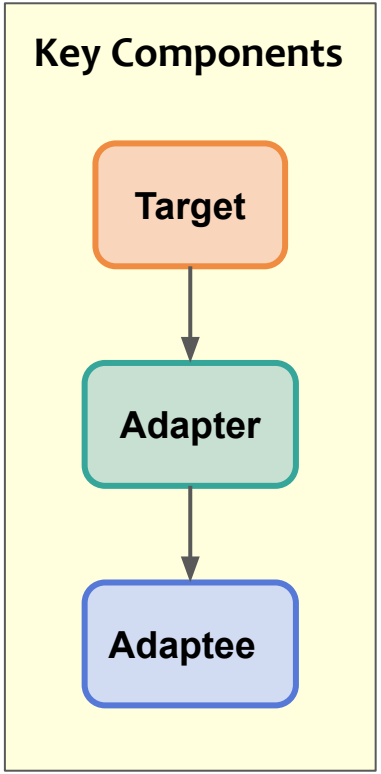


Adapter pattern

- **Problem**: An existing component offers functionality, but is incompatible with the new system being developed
- **Solution**: The adapter pattern connects incompatible components
  - Allows the reuse of existing components
  - Converts the interface of the existing component into another interface expected by the calling component
  - Useful in **interface engineering** projects and in **reengineering** projects
  - Often used to provide a new interface for a legacy system
- Also called a **wrapper**

Adapter pattern: Key components

- **Target:** The desired interface that the client wants to use
  - Defines the methods that the client will call
- **Adaptee:** The existing interface that needs to be adapted
  - Provides the functionality that needs to be used by the client, but its interface is incompatible with the Target
- **Adapter:** The class that converts the Adaptee's interface into the Target's interface
  - Implements the Target interface and holds a reference to the Adaptee, making it possible to call the Adaptee's methods using the Target's interface

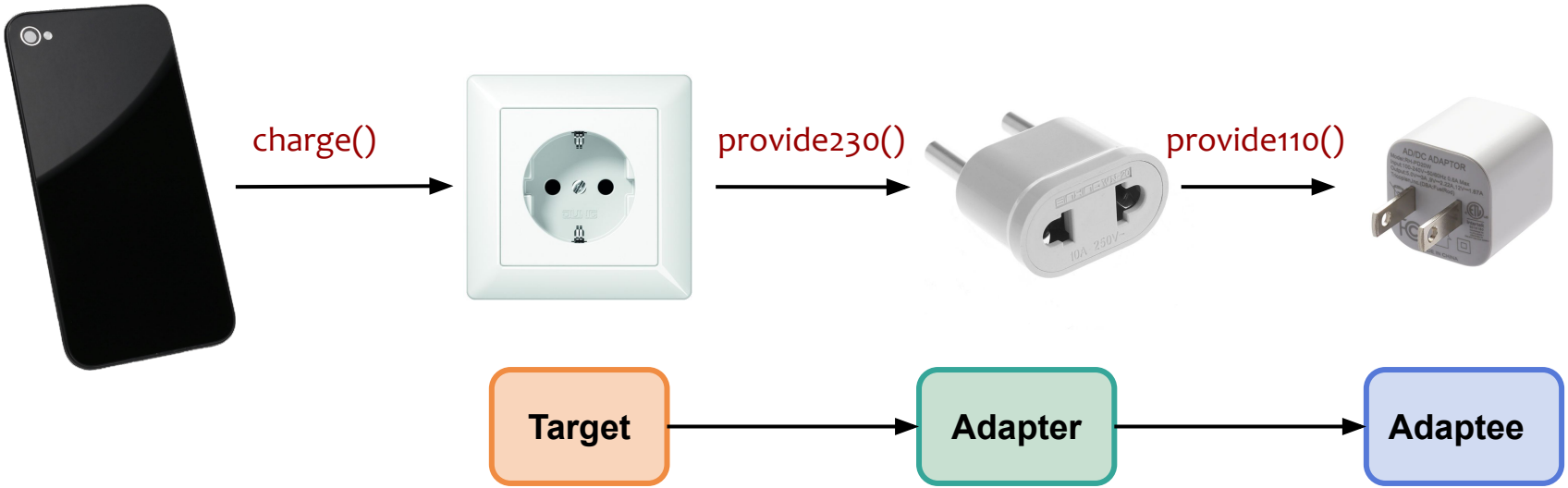


Example1: Accessing a power charger

- **Scenario:** Evgeny is using a phone that requires power
- **Problem:** Evgeny's phone battery is empty, he has access to a US Charger that offers 110 Volt charging
- **Challenge:** provide power to the US Charger in Germany

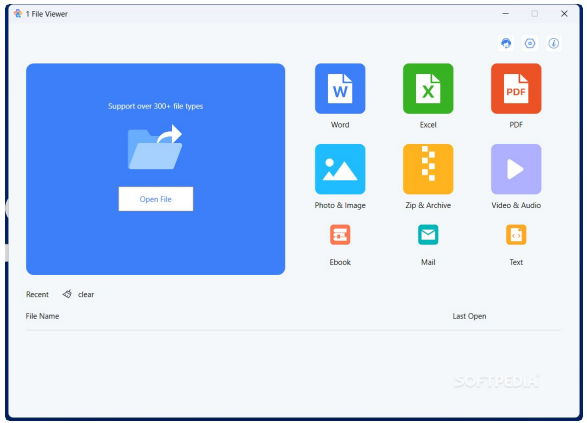
Example1: Accessing a power charger

- **Solution:** Provide the voltage adapter to deliver the correct voltage



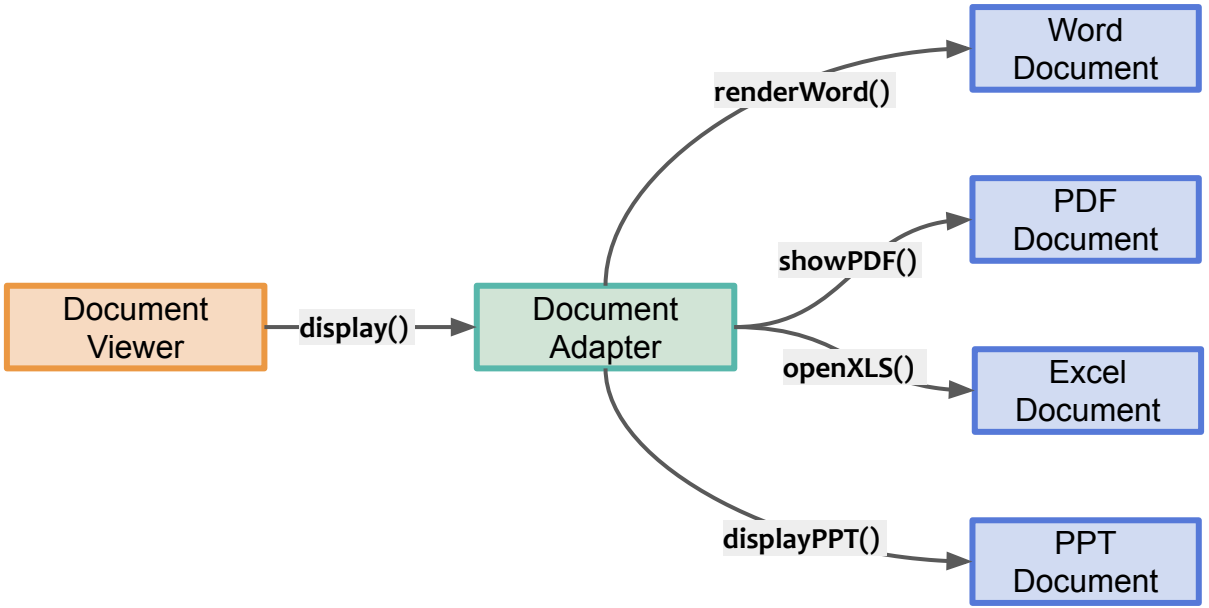
Example2: A document viewer application

- **Scenario:** A document viewer application needs to open and display different types of documents (e.g., PDF, Word, Excel)
- **Problem:** Each document type has a different interface for opening and rendering its content
- **Challenge:** Allow the document viewer application to open and display various document types without changing its existing code



Example2: A document viewer application

- **Solution:** Create adapters for each type of document



Adapter Pattern

- **Benefits**
  - Increased flexibility: Adapter allows classes to work together even with incompatible interfaces
  - Improved reusability: Adaptee's functionalities can be reused without changing its existing code
  - Easier maintenance: Changes in the Adaptee do not affect the client code using the Target interface
- **Use Cases**
  - When you want to use an existing class with an incompatible interface
  - When you need to create a reusable class that cooperates with unrelated or unforeseen classes
  - When you need to provide a uniform interface to different APIs or libraries

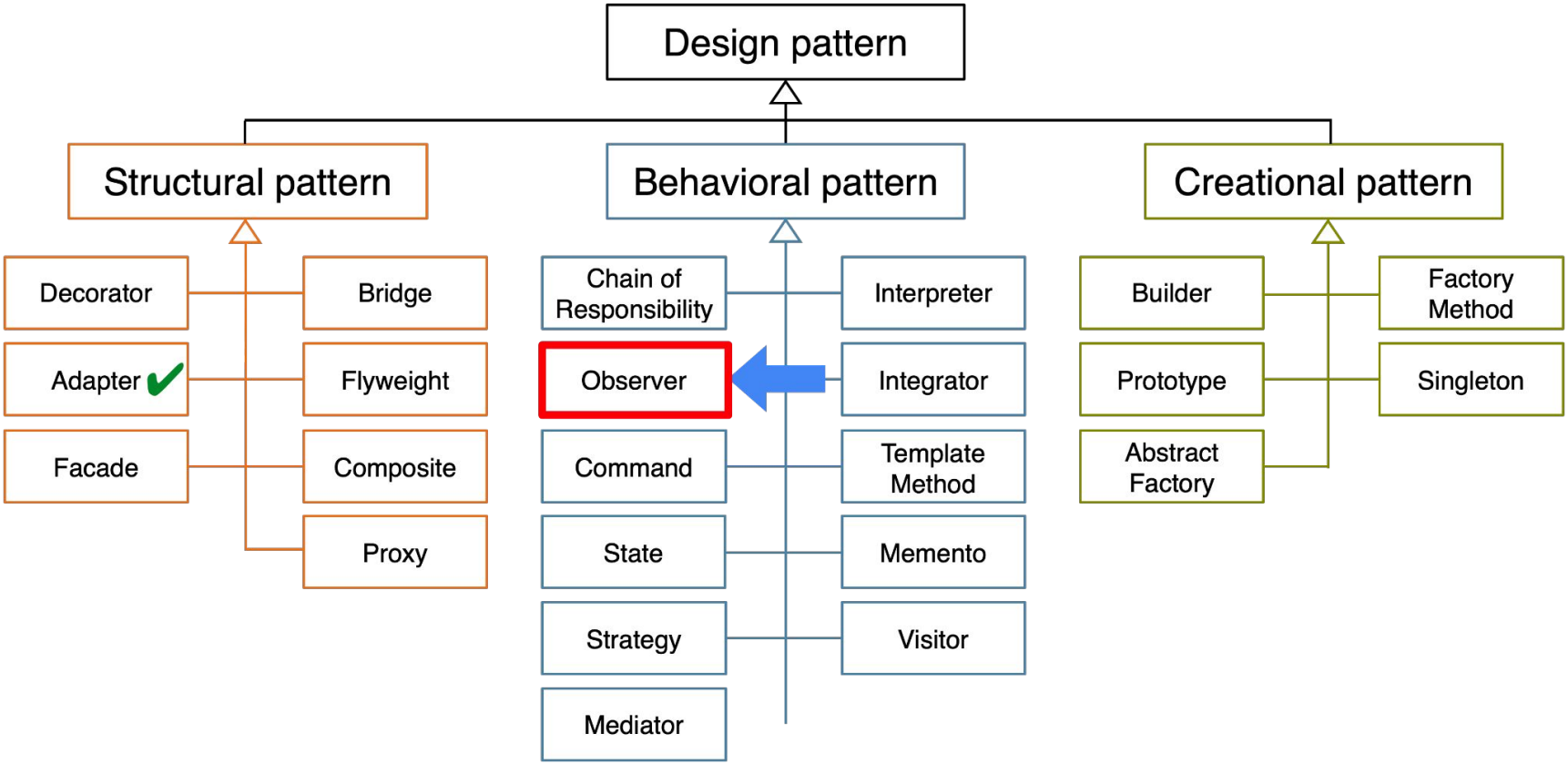
References for Adapter Pattern

- **Design Pattern Taxonomy:** Structural design pattern
- **Key Components:**
  - *Target:* The desired interface for the client
  - *Adaptee:* The existing, incompatible interface
  - *Adapter:* The class that converts the Adaptee's interface into the Target's interface
- **Purpose:** Interface conversion
- **Benefits:** Flexibility, reusability, and easier maintenance
- **References:**
  - https://refactoring.guru/design-patterns/adapter
  - https://sourcemaking.com/design_patterns/adapter
  - https://www.geeksforgeeks.org/adapter-pattern/
  - https://en.wikipedia.org/wiki/Adapter_pattern

Outline

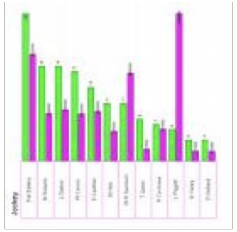
- ~~**Part I:** Design principle: Concurrency (or Scale Up!)~~
- ~~**Part II:** Design principle: Scalability (or Scale Out!)~~
- **Part III: Pattern implementation**
  - ~~Adapter pattern~~
  - **Observer pattern**
  - Strategy pattern

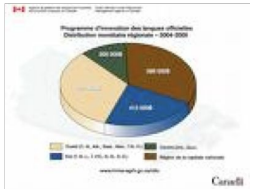
Design patterns taxonomy



Observer pattern

- **Problem:**
  - An object that changes its **state** often, e.g., a portfolio of stocks
  - Multiple views of the current state, e.g., histogram view, pie chart view, timeline view
- Requirements
  - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
  - The system design should be highly extensible
  - It should be possible to add new views - for example, an alarm - without having to recompile the observed object or existing views





Observer pattern

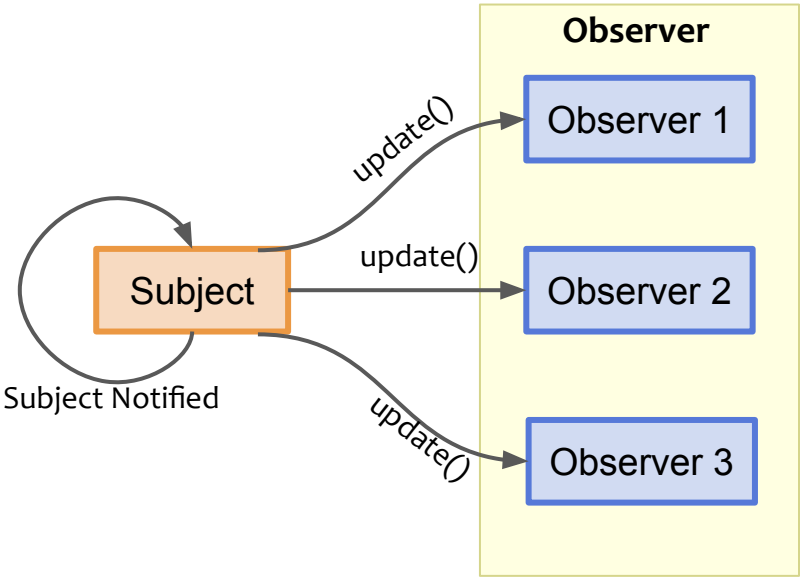
- **Solution:** model a one-to-many dependency between objects
  - Connect the state of an observed object - **subject** with many observing objects - the **observers**
  - When the state of the **subject** changes, all its dependents (**observers**) are automatically notified and updated.

→ Also called **Publish and Subscribe**

  - The subject acts as the publisher, while the observers are the subscribers who receive notifications.

Observer pattern: Key components

- **Subject:** The object that has one-to-many dependencies with other objects (observers)
  - Maintains a list of observers and provides methods to add, remove, and notify observers
- **Observer:** An interface that defines the methods to be implemented by objects that need to be notified when the Subject's state changes
  - Specifies the methods to update the observer's state when notified by the Subject



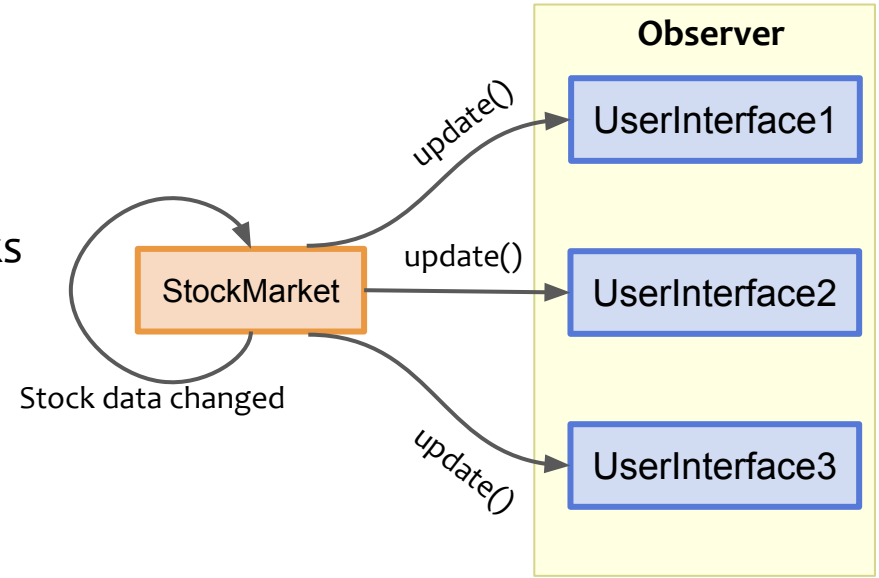
Variants of the observer pattern

3 variants for maintaining the consistency

1. **Pull Model:** Observer requests state from Subject (MVC: View requests updates from Model)
2. **Push Model:** Subject automatically sends updates to Observer (MVC: Model sends updates to View)
3. **Hybrid Model:** Observer can use either Pull or Push (MVC: Model provides both Pull and Push updates to View)

Example: A stock market application

- **Scenario:** A stock market application that displays stock prices for various stocks to multiple users
- **Problem:** When stock prices change, all users should receive updates for the stocks they are interested in
- **Challenge:** Update users in real-time without tightly coupling the stock data source to the individual user interfaces



Observer pattern

- **Benefits**
  - Maintain consistency across redundant observers
  - Optimize a batch of changes to maintain consistency
  - Promotes loose coupling between the subject and its observers
- **Use Cases**
  - When a change in one object requires updating other objects, and the number of objects is unknown or can change
  - When an object should be able to notify other objects without making assumptions about who those objects are or what they do

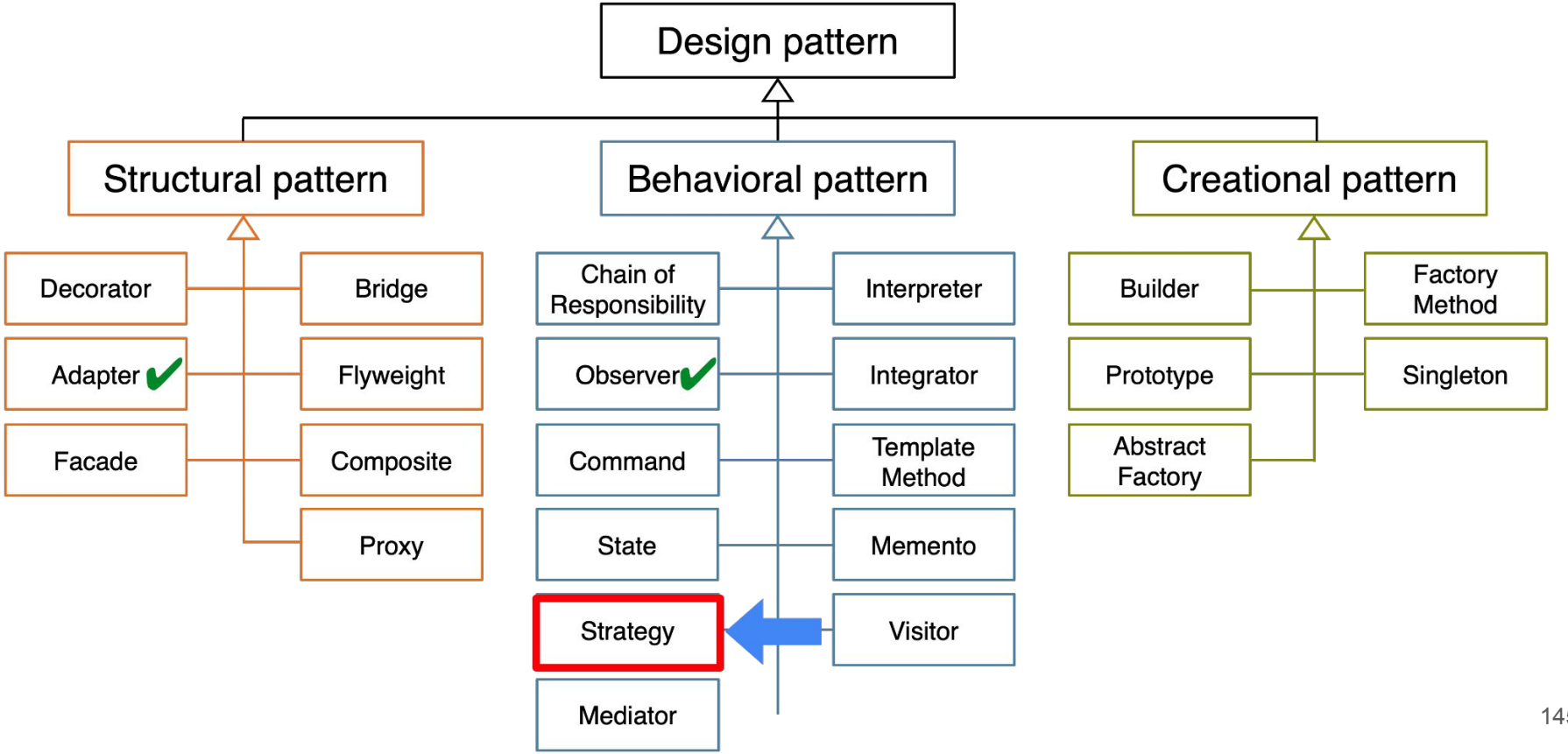
References for Observer Pattern

- **Design Pattern Taxonomy:** Behavioural design pattern
- **Key Components:** Subject, Observer
- **Purpose:** Manage one-to-many dependencies between objects
- **Benefits:** Loose coupling, dynamic observer management, simultaneous notifications
- **References:**
  - https://refactoring.guru/design-patterns/observer
  - https://sourcemaking.com/design_patterns/observer
  - https://www.javadesignpatterns.com/patterns/observer/
  - https://www.geeksforgeeks.org/observer-pattern-in-java/

Outline

- ~~**Part I:** Design principle: Concurrency (or Scale Up!)~~
- ~~**Part II:** Design principle: Scalability (or Scale Out!)~~
- **Part III: Pattern implementation**
  - ~~Adapter pattern~~
  - ~~Observer pattern~~
  - **Strategy pattern**

Design patterns taxonomy

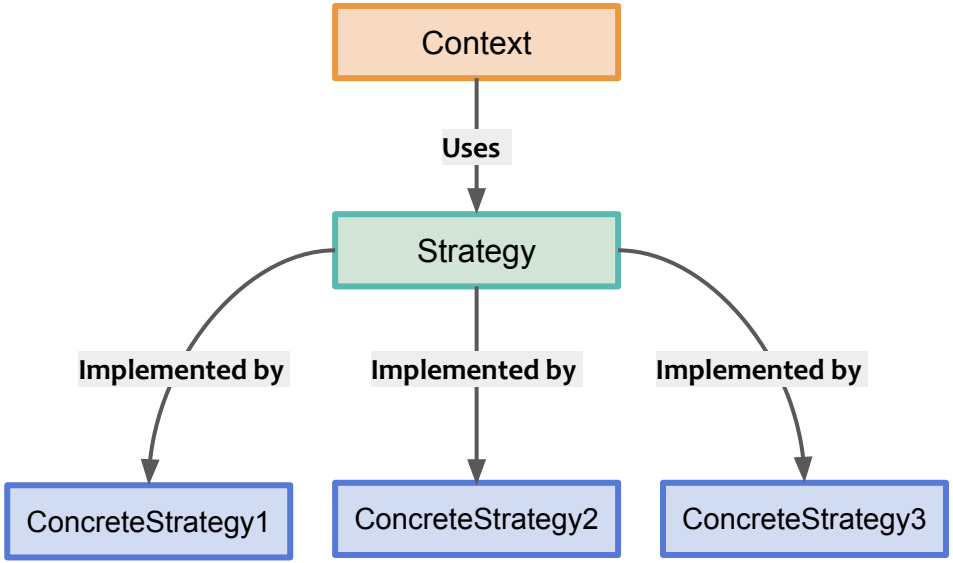


Strategy pattern

- **Problem:** Different algorithms exist for a specific task
- **Requirement:**
  - If we need a new algorithm, we want to add it without changing the rest of the application or the other algorithms
- **Solution:** the strategy pattern allows switching between different algorithms at run time based on the context and a policy

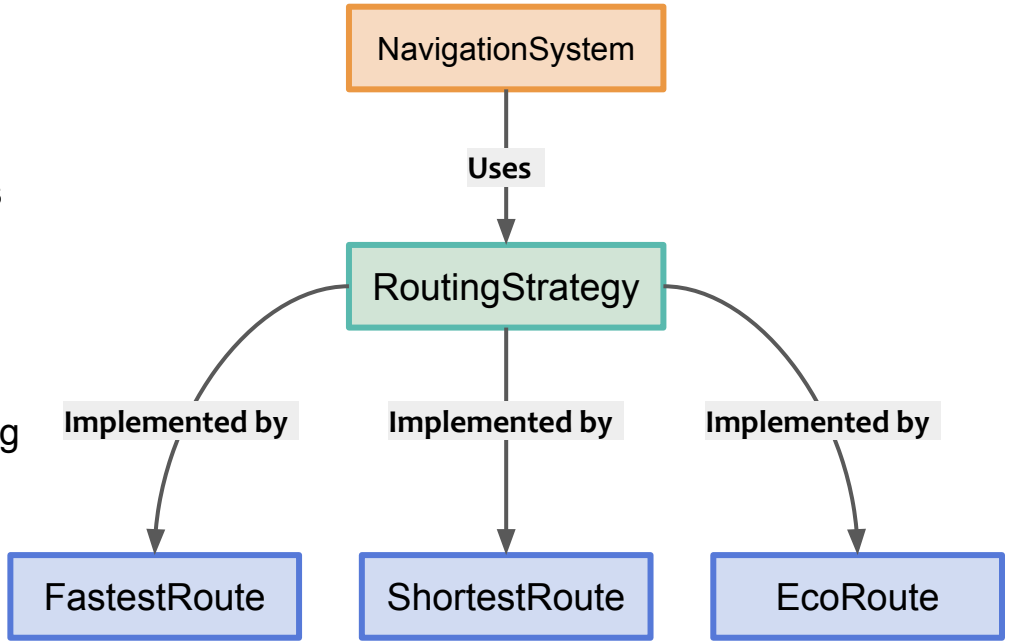
Strategy pattern: Key components

- **Context:** The class that uses a Strategy to perform an operation
  - Contains a reference to a Strategy object and delegates the execution of the algorithm to it
- **Strategy:** An interface that defines the methods to be implemented by different algorithms
  - Specifies the methods to execute the algorithm
- **Concrete Strategies:** Classes that implement the Strategy interface and define a specific algorithm
  - Provides a specific implementation of the algorithm



Example: Navigation routing options

- **Scenario:** A navigation application that provides different routing options based on user preferences
- **Problem:** Users may choose different types of routes: fastest, shortest, or most fuel-efficient
- **Challenge:** Implement a system that can easily switch between routing algorithms based on user preferences without modifying the core navigation code



Strategy pattern

- **Benefits**
  - Encourages separation of concerns and code reusability
  - Allows adding new algorithms without modifying the Context or existing algorithms
  - Supports selecting algorithms at runtime
- **Use-cases**
  - When different variations of an algorithm are required
  - When a class has a behaviour that is implemented by multiple algorithms and can be switched during runtime

References for Strategy Pattern

- **Design Pattern Taxonomy:** Behavioural design pattern
- **Key Components:** Context, Strategy, Concrete Strategies
- **Purpose:** Encapsulate algorithms and make them interchangeable
- **Benefits:** Separation of concerns, code reusability, runtime algorithm selection
- **References:**
  - https://refactoring.guru/design-patterns/strategy
  - https://www.javadesignpatterns.com/patterns/strategy/
  - https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

Clues for the use of design patterns

- **For Adapter Pattern:**
  - Need to convert the interface of a class into another interface, clients expect
  - Want to reuse an existing class, but its interface is incompatible with the rest of the system
  - Need to provide a unified interface to a set of classes with diverse interfaces
- **For Observer Pattern:**
  - Need to establish a one-to-many relationship between objects
  - Need to update a set of dependent objects automatically when an object's state changes
  - Want to decouple a subject from its observers, promoting loose coupling
- **For Strategy Pattern:**
  - Need to define a family of algorithms and make them interchangeable
  - Want to provide a way to select an algorithm at runtime
  - Need to encapsulate algorithms and avoid code duplication

References

- Design Patterns. Elements of Reusable Object-Oriented Software – Gamma, Helm, Johnson & Vlissides
- Pattern-Oriented Software Architecture, Volume 1, A System of Patterns - Buschmann, Meunier, Rohnert, Sommerlad, Stal
- Pattern-Oriented Analysis and Design - Composing Patterns to Design Software Systems - Yacoub & Ammar
- https://sourcemaking.com

Summary

- **Part I:** Concurrency (or Scale Up!)
  - Why concurrency?
  - The thread model
  - Thread scheduling
  - Communication mechanisms
  - Parallelizing a program
  - Accelerators
- **Part II:** Scalability (or Scale Out!)
  - Scalability challenges
  - Scalability techniques
- **Part III:** Pattern implementation
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

Homework exercises

- Quiz exercises
  - **L06Q01**
- Text exercises **(Discussed during the tutorial, NOT graded)**
  - **L06T01:** Barrier implementations
  - **L06T02:** Lock-free and non-lock-free data structures
  - **L06T03:** Parallelization of single-threaded application
  - **L06T04:** Observer design pattern
  - **L06T05:** Strategy design pattern
- Programming exercise
  - **L06P01:** Strategy Pattern
  - **L06P02:** Adapter Pattern
  - **L06P03:** Parallelism
- Side projects **(For top/advanced students, NOT graded)**
  - **L06SP01:** Building a concurrent (scale-up) client/server architecture
  - **L06SP02:** Building a scalable (scale-out) client/server architecture

Submission schedule for homework exercises

#	Submission deadline (Friday, 5pm)	Based on content of:
1	**NOT GRADED**	Introduction
2	12.05	Cloud Systems Engineering
3	19.05	System Design Requirements + Cloud Software Architectures
4	09.06	System Design and Implementation I
5	16.06	System Design and Implementation II
6	30.06	System Design and Implementation III
7	07.07	Software Testing
8	14.07	Program Analysis
9	21.07	Software Management, Deployment, and Monitoring
10	28.07	Software Quality and Project Management + **Exam prep.**
	NO HOMEWORK	Guest Lecture

Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

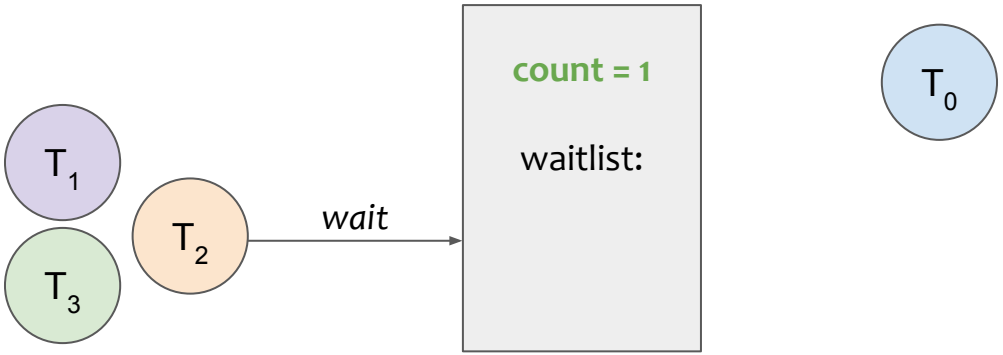
- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

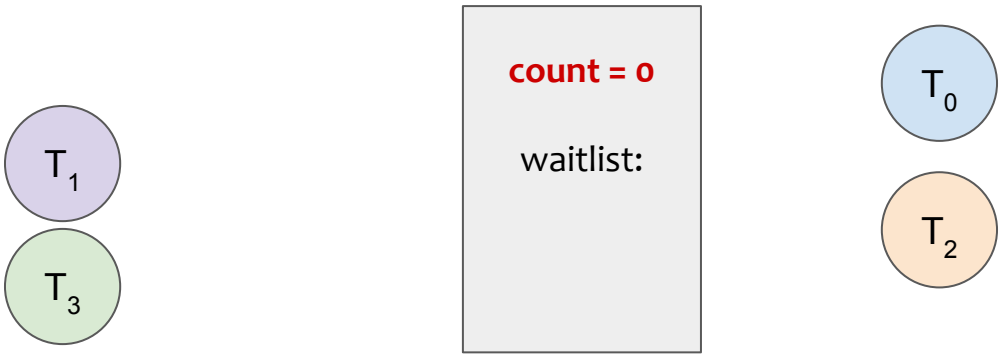


Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

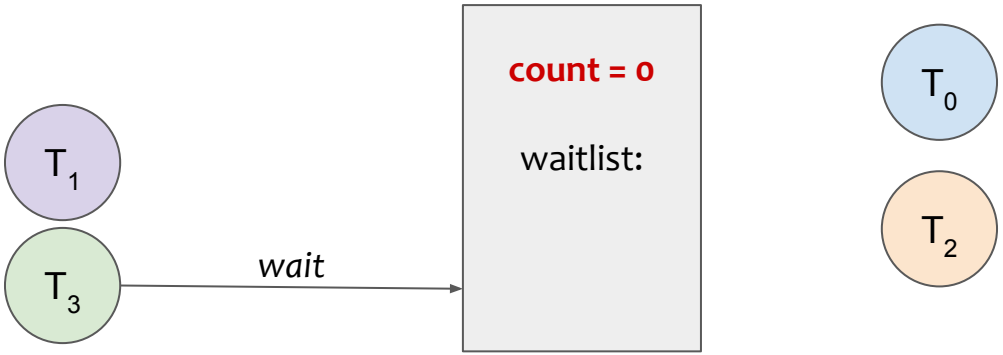


Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

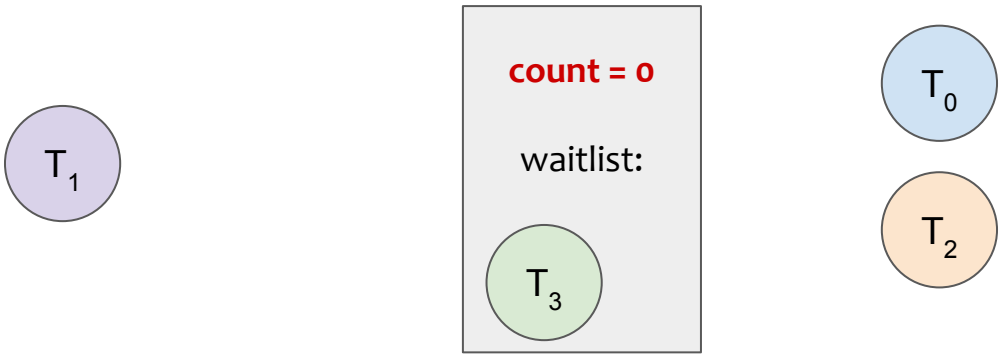


Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

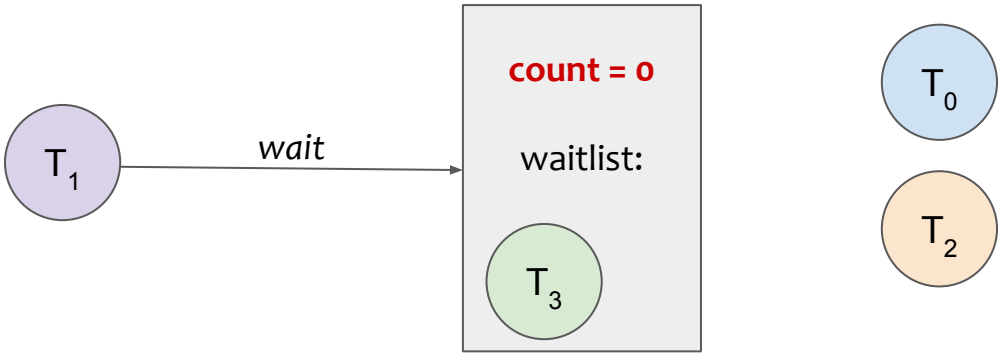


Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

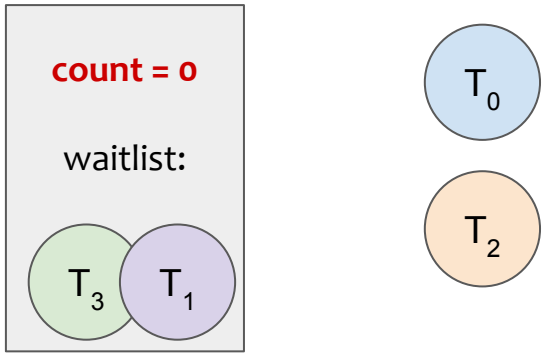


Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

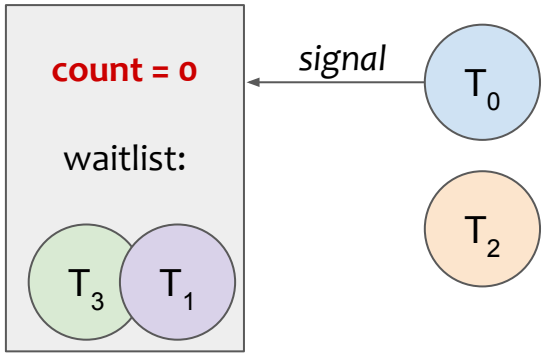


Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1

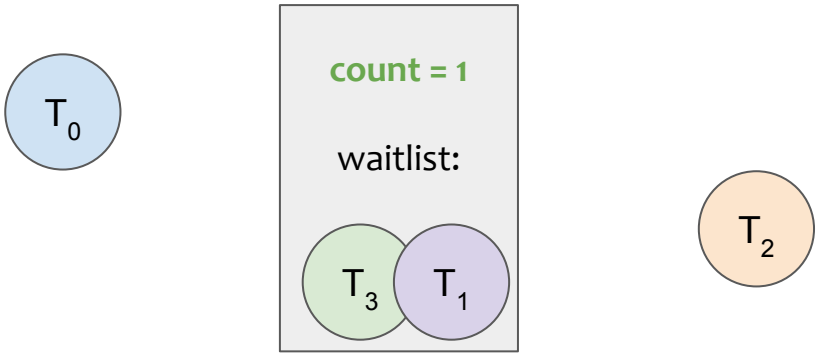


Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1



Semaphores

A semaphore allows threads to wait on a given number of resources
It is a basic block to build other synchronization mechanisms

A semaphore is composed of:

- An *atomic counter* K ≥ 0, *i.e., the number of resources available*
- A *wait* method: wait until K > 0, then decrement K by 1
- A *signal* method: increment K by 1