L04 System Design II

Performance, Concurrency, and Scalability

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

A three-part series: System design in our course

L03: Design I

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

L04: Design II

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

L05: Design III

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

System implementation

Today's learning goals

- **Part I:** Performance
  - Performance metrics
  - A systems approach to designing for performance
  - Measurement-driven approach to build high-performance systems
  - Design hints for performance
- **Part II:** Concurrency (or Scale Up!)
  - Why concurrency?
  - The thread model
  - Thread scheduling
  - Communication mechanisms
  - Parallelizing a program

Today's learning goals

- **Part III:** Scalability (or Scale Out!)
  - Scalability challenges
  - Scalability techniques
  - Scalable data management

Outline

- **Part I: Performance**
  - **Performance metrics:** Latency, throughput, utilization, SLAs
  - A systems approach to designing for performance
  - Measurement-driven approach to build high-performance systems
    - Identifying bottlenecks (time-based profiling)
    - Automated performance profiling tools (Linux perf and flamegraphs)
  - Design hints for performance:
    - Resource splitting
    - Caching
    - Compute in background
    - Batch processing
    - Parallelism
- **Part II:** Concurrency (or Scale up!)
- **Part III:** Scalability (or Scale out!)

Performance

- **Let's face it – performance matters!**
  - A large number of empirical studies show that **performance directly impact 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.

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

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

From L02: Performance

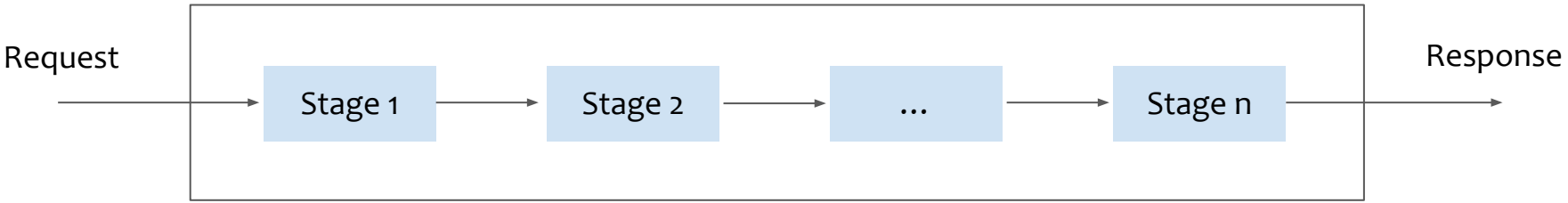
- **The need for performance**
  - The specification of a computer system typically includes explicit (or implicit) performance goals
- **Performance metrics:**
  - **Latency:** The time interval between a user's request and the system response
  - **Throughput:** Number of work units done (or requests 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

Metrics: Capacity, utilization, overhead, & useful work

- **Capacity of the service**
  - Measure of a service's size (or amount of resources)
- **Utilization**
  - Percentage of capacity of a resources that is used for a given workload of requests
  - E.g., 10% of CPU cycles were used to process a given requests
- **Overheads/Useful work**
  - Resources consumed to doing not "useful work", *i.e., imposing overheads w.r.t. a baseline*
  - Abstractions in systems introduce overheads. For e.g., CPU is utilized for 95% to run a Virtual Machine, but application is using only 70% CPU cycles
  - From the application point of view:
    - **Overheads** imposed by the Virtual Machine is 25%
    - Only 70% of **useful work** done on the system

Metric: Latency

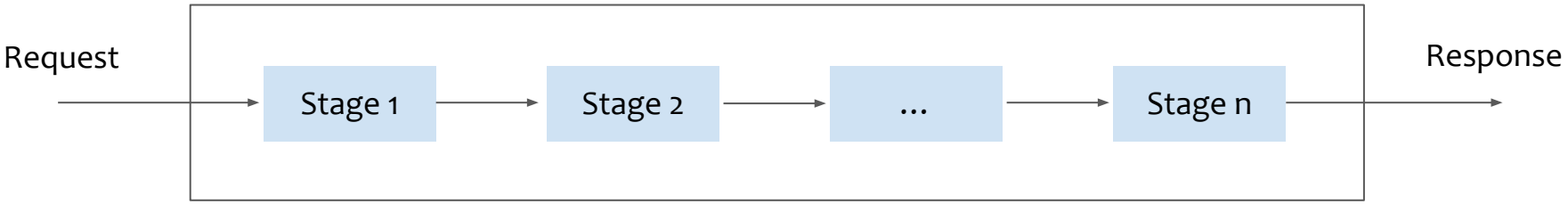
- **Latency** is the delay between a change at the input to a system and the corresponding change at its output
- From **the client-server perspective**, the latency of a request is the time from issuing the request until the time the response is received
  - Sending message + processing the request + 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:** *Use a back-of-the envelope model*
  - **If** removing the current bottleneck doesn't improve performance significantly
    - Stop iterating, consider re-designing the system
  - **Else** (i.e., likely improve the performance significantly)
    - Focus on design improvements – **We cover some performance hints later!**
- **Measure the new implementation** to verify the change effectiveness
- **Iterate**
  - Law of diminishing returns!

Measurement-driven approach

- Measure, Then Build
  - A great keynote from Prof. Remzi Arpaci Dusseau
    - https://www.usenix.org/conference/atc19/presentation/keynote
  - Argues an iterative, empirically-driven approach to build systems
  - Overall, a better system design and performance

- How do we measure systems?
  - Well, **modularity** is the starting point

- Computer systems are designed and organized as a set of subsystems/modules
  - Measure the impact of individual modules to identify the bottleneck

What tools do I have for performance measurement?

The manual approach

- **Measure the time**
  - **CPU time:** the time actually spent by CPU executing method code
  - **Wall time:** the real-world time elapsed between a pair of events
  -
- **The time approach is limiting:**
  - Manual approach, requires application instrumentation → doesn't scale!
  - Limited information → Mostly time measurements!
    - What about other metrics, e.g., memory?
  - Noisy data → Can't correctly profile the code for parallel programs
    - OS schedules threads – they are scheduled/descheduled to allow CPU multiplexing
    - This affects the measurements results

Alternative: Use an automated approach!

- **Linux perf** is a performance analyzing tool in Linux
  - perf is accessed from the command line and provides a number of subcommands
  - Capable of statistical profiling of the entire system (both kernel and userland code)
- Check out the following example:
  - https://stackoverflow.com/questions/2229336/linux-application-profiling
- Detailed tutorial:
  - https://perf.wiki.kernel.org/index.php/Tutorial

Application profilers

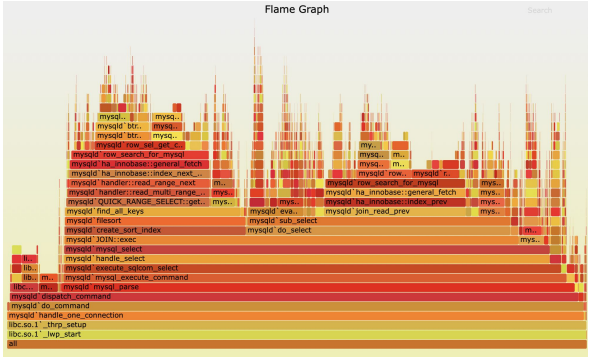
- **Java profiler:**
  - Supports profiling of Java application within IntelliJ IDE
  - Jetbrains Java profiler: https://lp.jetbrains.com/intellij-idea-profiler/
- **GNU gprofng:**
  - Supports the profiling of programs written in C, C++, Java, or Scala
  - GNU gprofng: https://sourceware.org/binutils/docs/gprofng.html
  - Also check: https://blogs.oracle.com/linux/post/gprofng-the-next-generation-gnu-profiling-tool
  - Perf vs gprofng: https://www.redhat.com/architect/perf-vs-gprofng

Use Flamegraphs to visualize the profiler report!

- Flame graphs are a visualization of hierarchical data, created to visualize stack traces of profiled software so that the most frequent code-paths to be identified quickly and accurately

- Flamegraphs are primarily used in conjunction with perf to find the hotpaths in your program execution

- See an example flamegraph: https://www.brendangregg.com/FlameGraphs/cpu-mysql-updated.svg
- Use perf w/ Flamegraphs: https://www.percona.com/blog/profiling-software-using-perf-and-flame-graphs/



A few hints for improving performance

1. Resource splitting
2. Caching
3. Compute in background
4. Batch processing
5. Parallelism

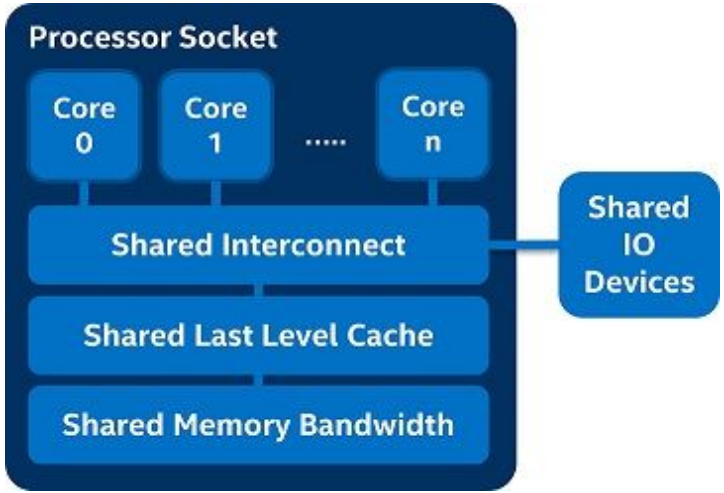
#1: Resource splitting/allocation

- **Split resources** in a fixed way if in doubt, rather than sharing them
- **Pros:** Dedicated resources are usually faster, and the behavior of the allocator is more predictable
- **Cons:** Inefficient resource usage
  - In many cases, however, the cost of the extra resources is small, or the overhead is larger than the fragmentation, or both.

An example of resource splitting: Cache allocation

- In data centers, many resources are shared by multiple tenants (CPUs, memory, network) , including the last level cache in modern multicore platforms

- The shared last level cache in the CPUs is used by all tenants, leading to interferences (noisy neighbors). This results in degraded performance (and security issues) !

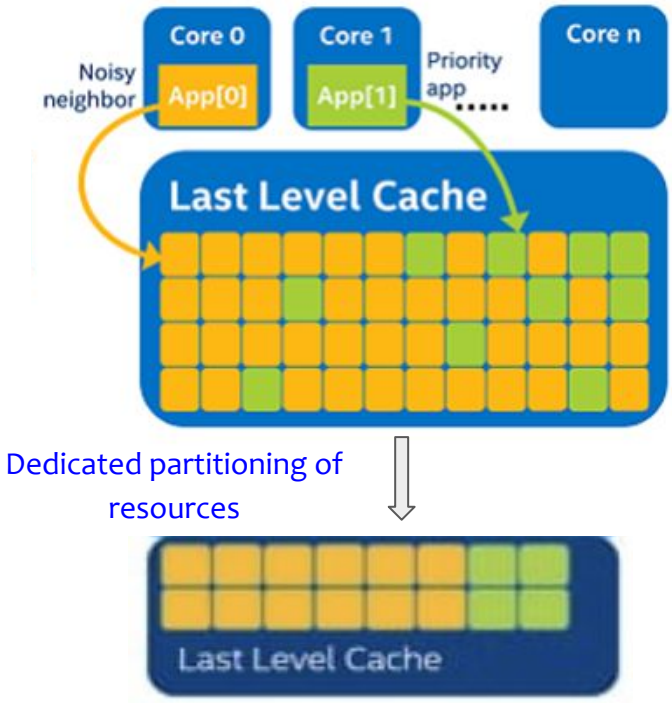


Shared last level cache
in multicore architectures

Performance design hint: Use dedicated resource splitting!

Fixed partitioning of Caches w/ Intel CAT

- Intel's Cache Allocation Technology (CAT) provides software-programmable control over the amount of cache space that can be consumed by a given thread, app, VM, or container
- In other words, Intel CAT allows dedicated partitioning of the shared cache memory
- Thus, it helps in giving better performance to a tenant – No noisy neighbors!



Intel Cache allocation Technology (CAT)
https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html

#2: Caching

- **Caching** is a general design hint, used widely across systems, for performance
  - **Cache answers to expensive computations, rather than doing them over**
- **Formally,** by storing the triple [f, x, f(x)] in an associative store with f and x as keys, we can retrieve f(x) with a lookup
  - This is faster if f(x) is needed again before it gets replaced in the cache, which presumably has limited capacity

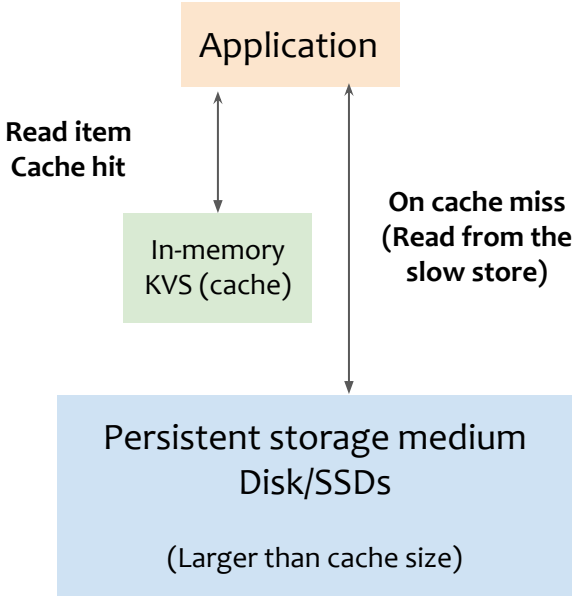
CPU
Cache: Main memory

Web cache
Last accessed webpages

Database/filesystem
Frequently accessed records/disk blocks

An example of caching for improving latency

- Use in-memory cache in databases/filesystems for reading frequently accessed items
  - Caching significantly helps to improve performance by hiding access latencies for disk/SSD
- Cache operation
  - **If** the item exists in the cache → **Cache Hit!**
    - Return the item from the cache → **FAST path**
  - **Else** → **Cache miss!**
    - Return the item from persistent storage → **SLOW path**



#3: Compute in background

- **Compute in background when possible**
  - In an interactive or real-time system, it is good to do as little work as possible before responding to a request
  - Also known as **asynchronous processing**
- The reason is twofold:
  - 1) A **rapid response** is better for the users → **Low latency!**
  - 2) The **load usually varies a great deal**, so there is likely to be idle processor time later in which to do background work → **Offline processing!**

An example of computing in background: Search indexing

- Search engines work in three stages
  - Crawling the Web
  - Compute Indexes
  - Serve search results

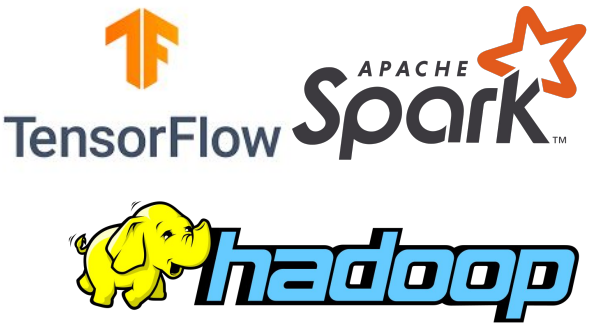
⅔ tasks are Computed in background!
Asynchronously

Foreground process!
Synchronously

#4: Batch processing

- **Batch processing** is a technique used in computer systems to process large volumes of data in batches, rather than processing individual transactions in real-time

- **Batch processing** involves collecting data over a period of time, storing it in a batch file or queue, and then processing it **all at once as a single unit**



Batch processing for data analytics

What is batch processing? https://aws.amazon.com/what-is/batch-processing/

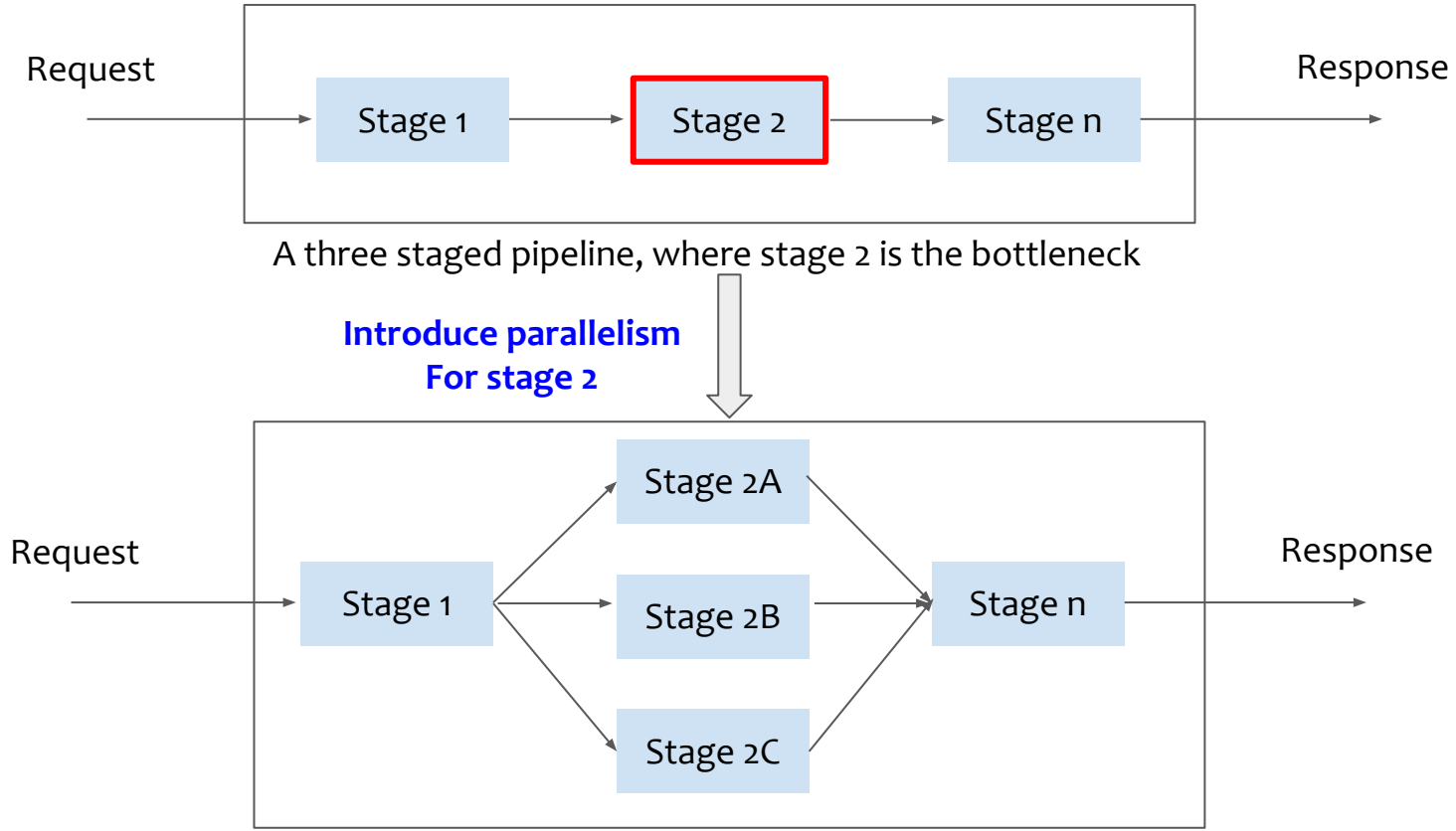
#5: Parallelism

- **Parallelism** is a technique used in computer systems to improve performance by dividing a task into smaller sub-tasks that can be executed simultaneously by multiple processors or threads

- Parallelism can be used in a variety of contexts, including
  - **Scale-up:** Multi-core CPUs, or GPUs
  - **Scale-out:** Distributed systems

Detailed topic in L06!

An example of parallelism to improve throughput



An interesting fact: Search engines use all hints!

- **Interesting fact:** Search engines apply all five performance hints
  - Use **dedicated resources** to do computation
  - Use **caching on the serving path** when returning frequently asked search queries
  - Use **computing in background** for crawling and computing indexes
  - Use **batch processing** for large-scale data analytics
  - Use **parallelism** to improve system throughput

References

- Hints for computer systems design
  - https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/acrobat-17.pdf
- Linux perf: https://perf.wiki.kernel.org/index.php/Main_Page
- Flamegraphs: https://www.brendangregg.com/flamegraphs.html
- Principles of Computer Systems Design by Saltzer and Kaaashoek
  - Chapter 6: Performance
  - https://ocw.mit.edu/courses/res-6-004-principles-of-computer-system-design-an-introduction-spring-2009/pages/online-textbook/
- Measure, Then Build
  - Keynote from Prof. Remzi Arpaci-Dusseau
  - https://www.usenix.org/conference/atc19/presentation/keynote

Outline

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

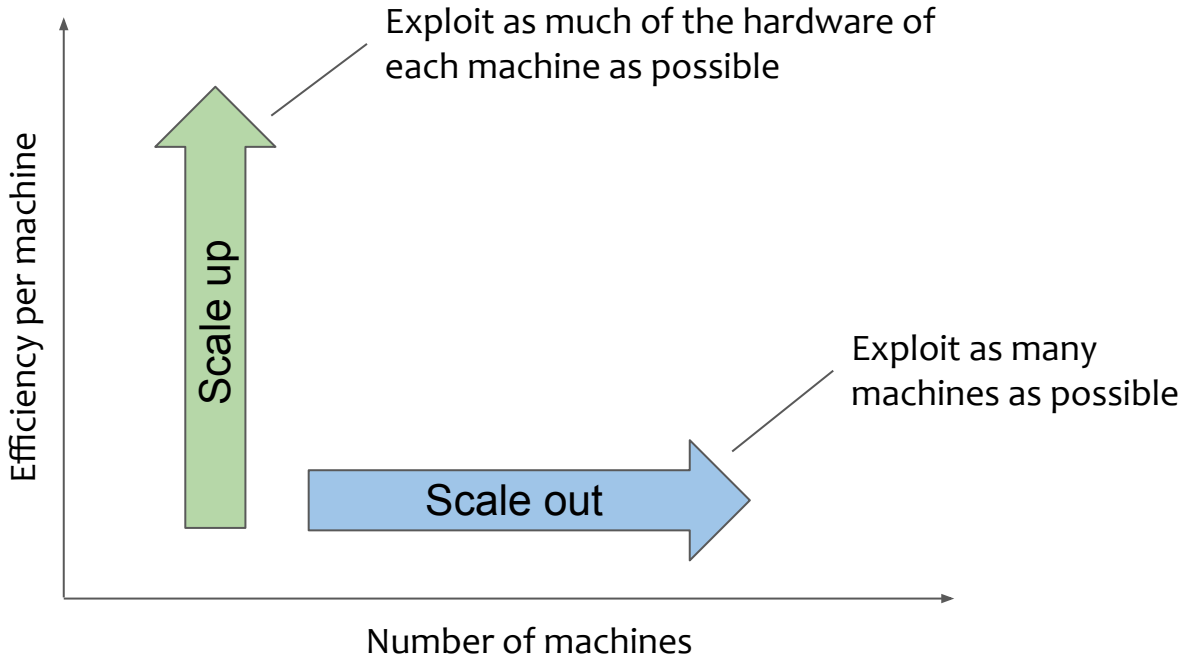
Performance is key

How can we exploit the hardware as much as possible?

Parallel and distributed software systems

(aka scale-up and scale-out!)

Scaling your applications



Modern processor architecture

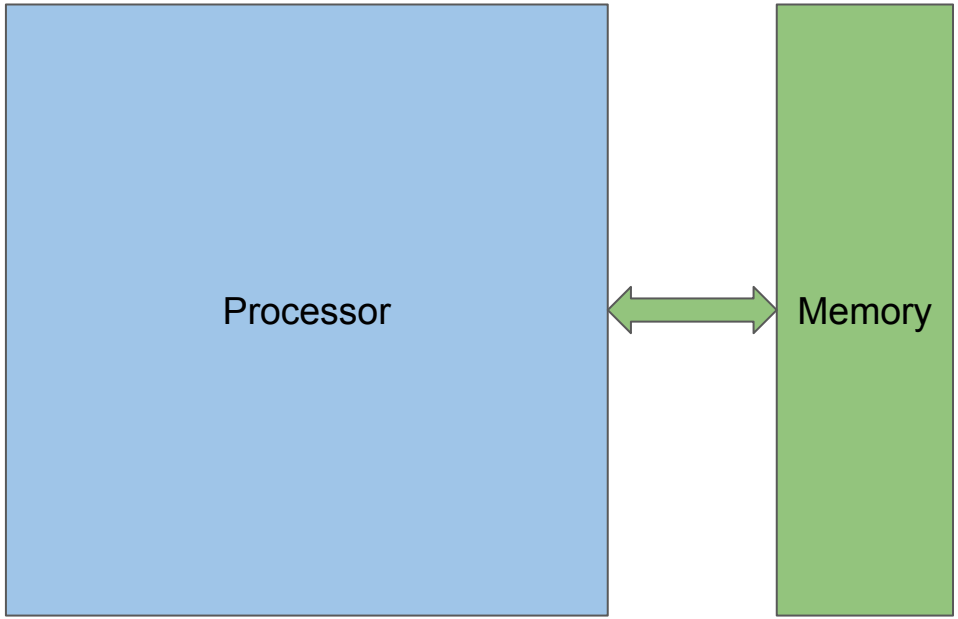
Computation

- Processing unit

  i.e., execute programs

Memory

- Stores data needed by programs during their execution



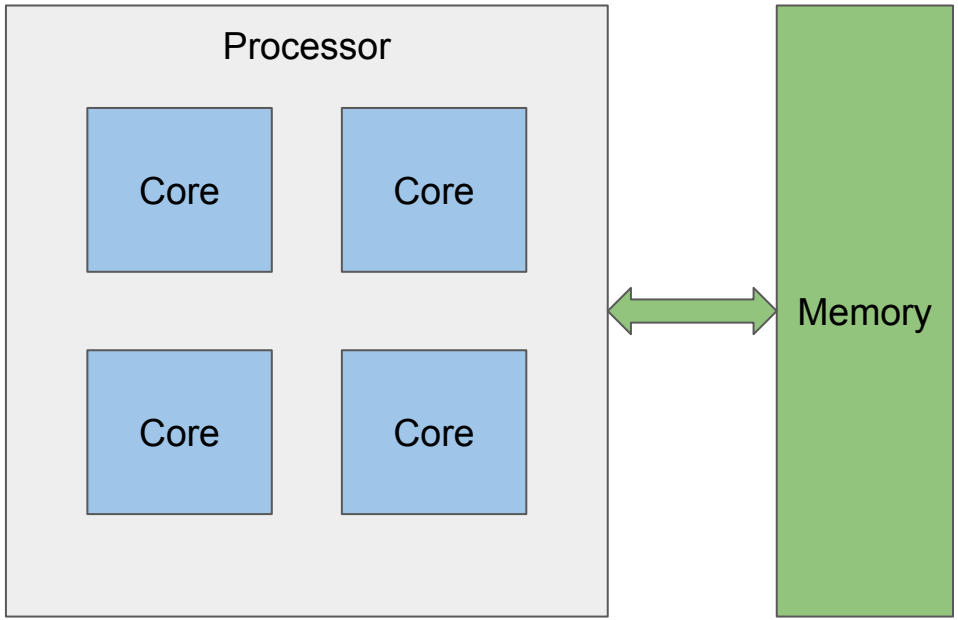
Modern processor architecture

Computation

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

Memory

- Stores data needed by programs during their execution



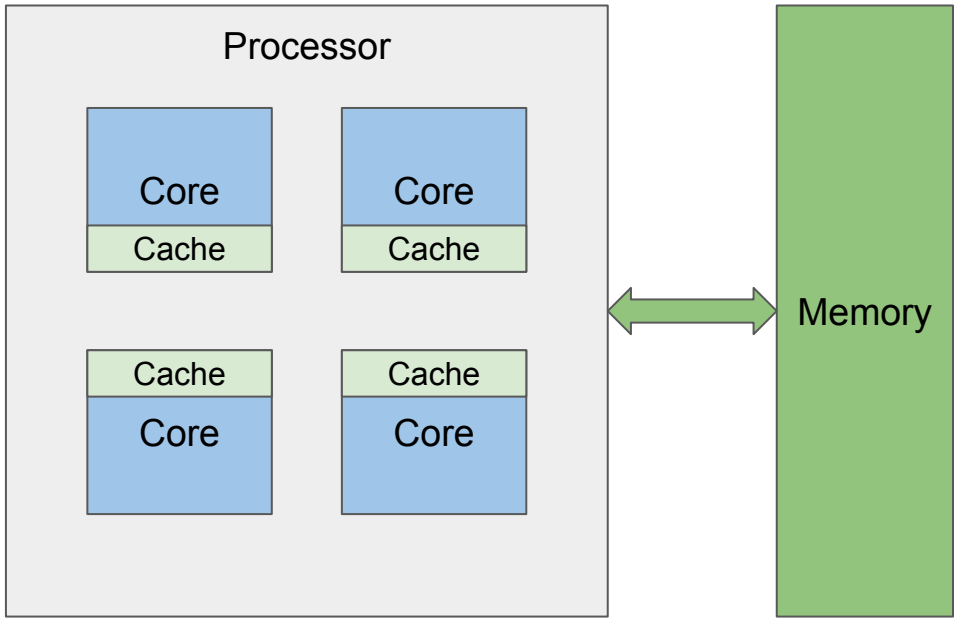
Modern processor architecture

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



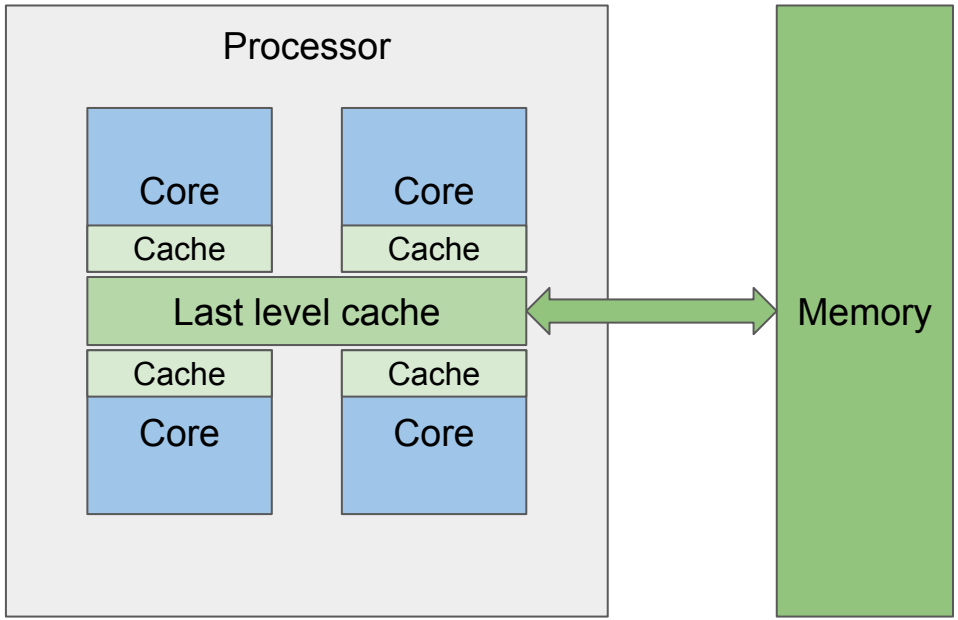
Modern processor architecture

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

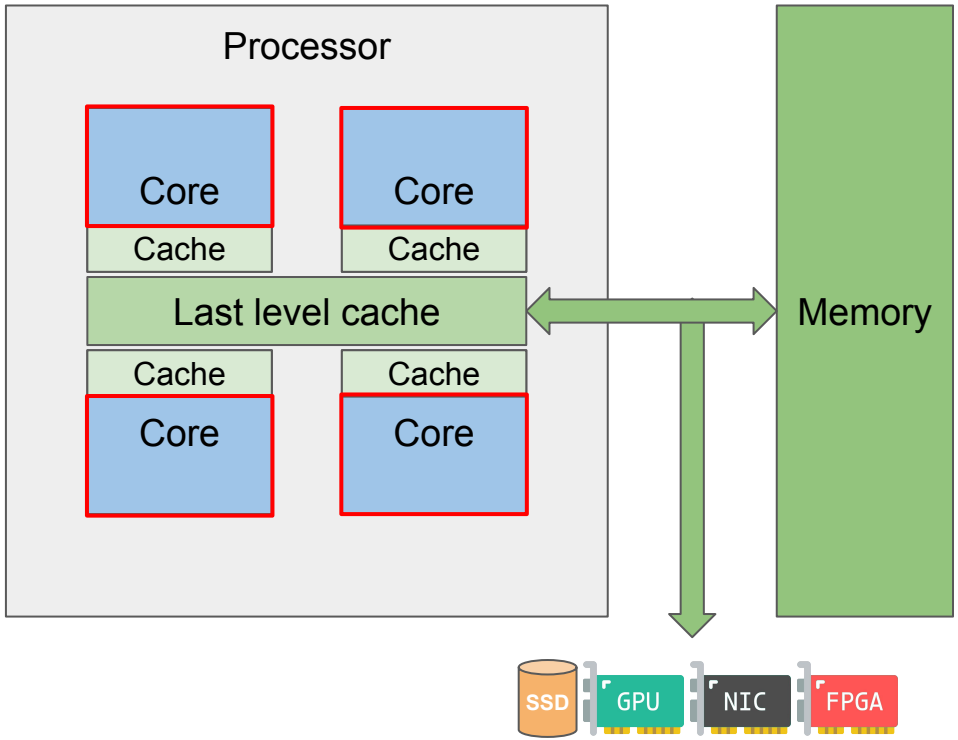
- Processing unit**s** (cores)
- Perform computations independently,
  i.e., execute programs

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:

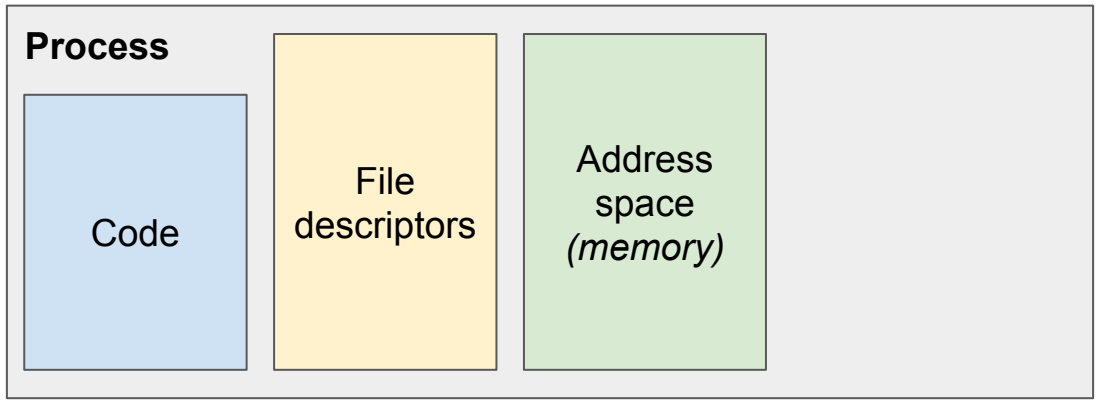
- A program (executable code)

Threads vs Process

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

A process is:

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

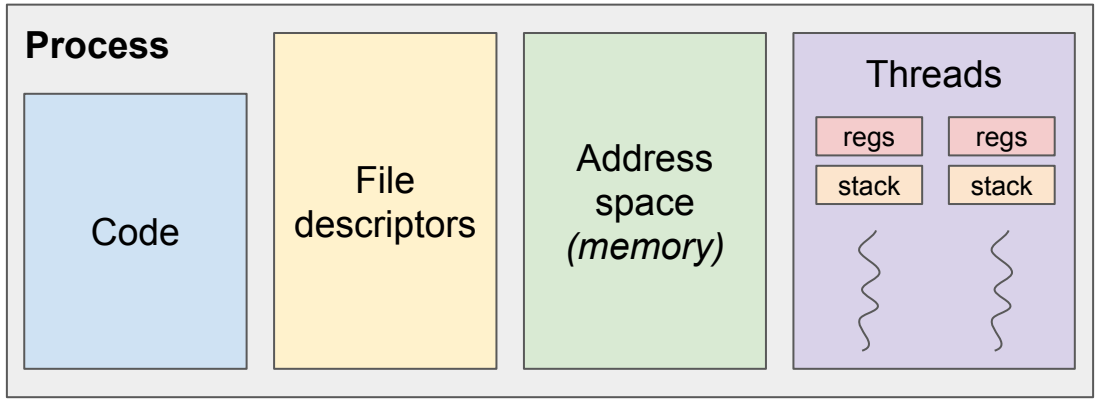


Threads vs Process

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

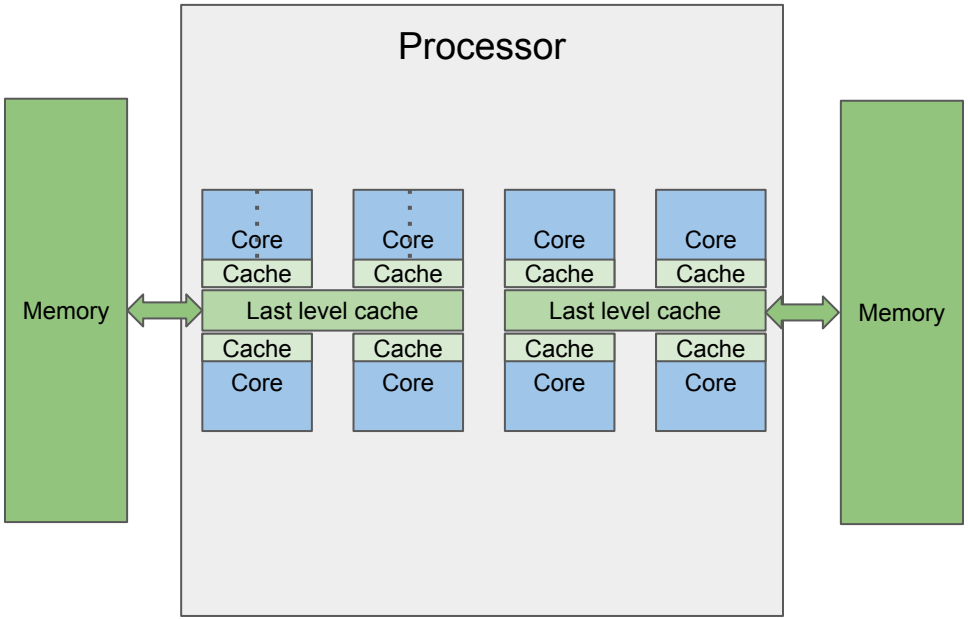
Some examples:

- **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

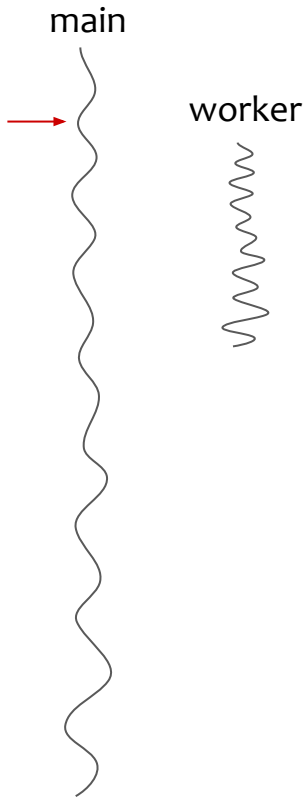
Your main thread waits for requests to arrive

Thread management: Spawning upon request

Your main thread waits for requests to arrive

When a new request arrives:

1. Create a new worker thread

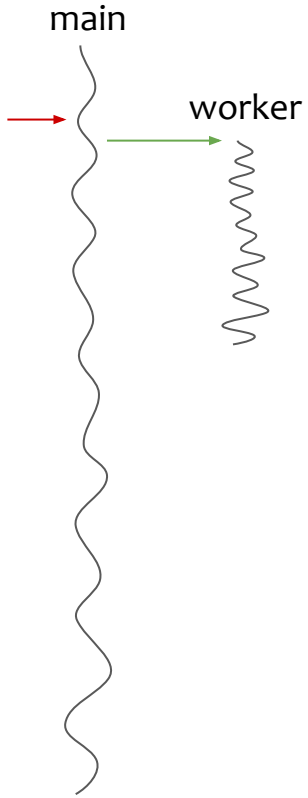


Thread management: Spawning upon request

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

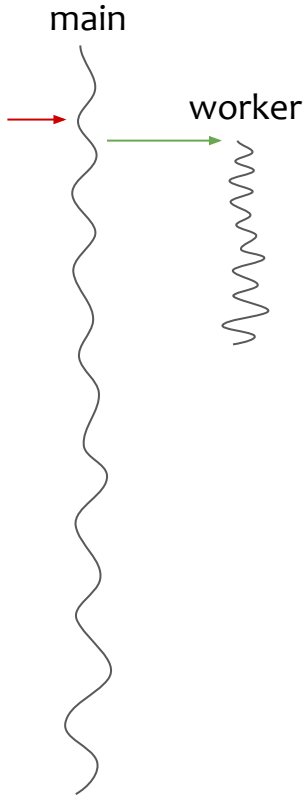


Thread management: Spawning upon request

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



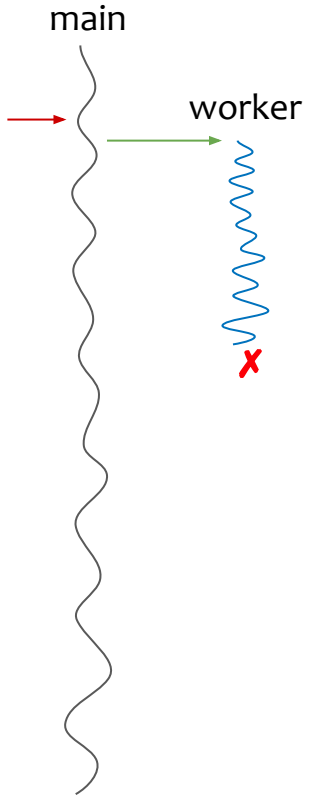
Thread management: Spawning upon request

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 ✗



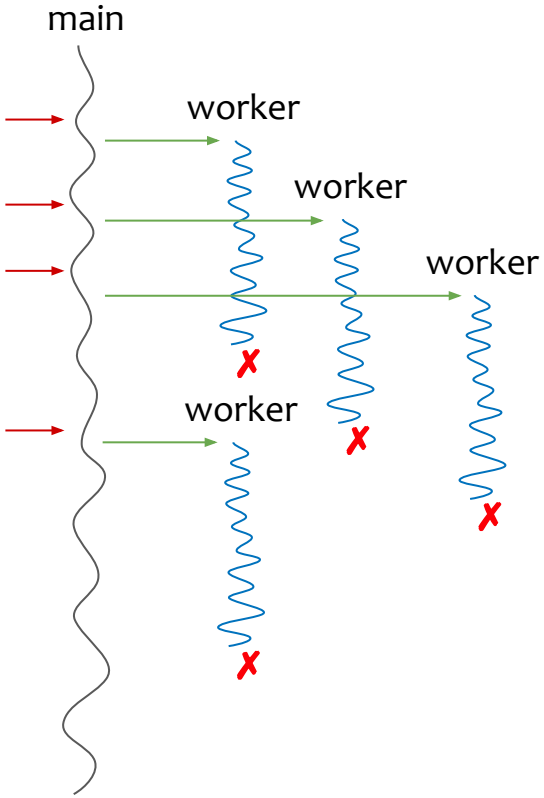
Thread management: Spawning upon request

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 ✗



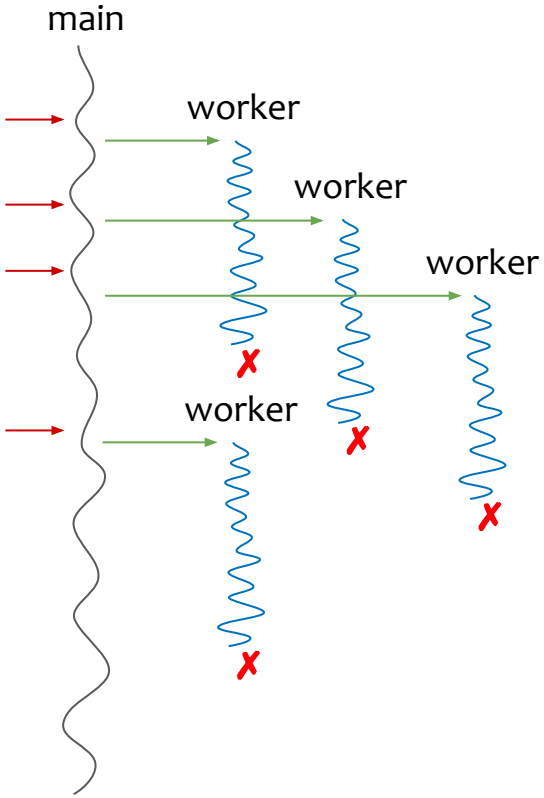
Thread management: Spawning upon request

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 ✗

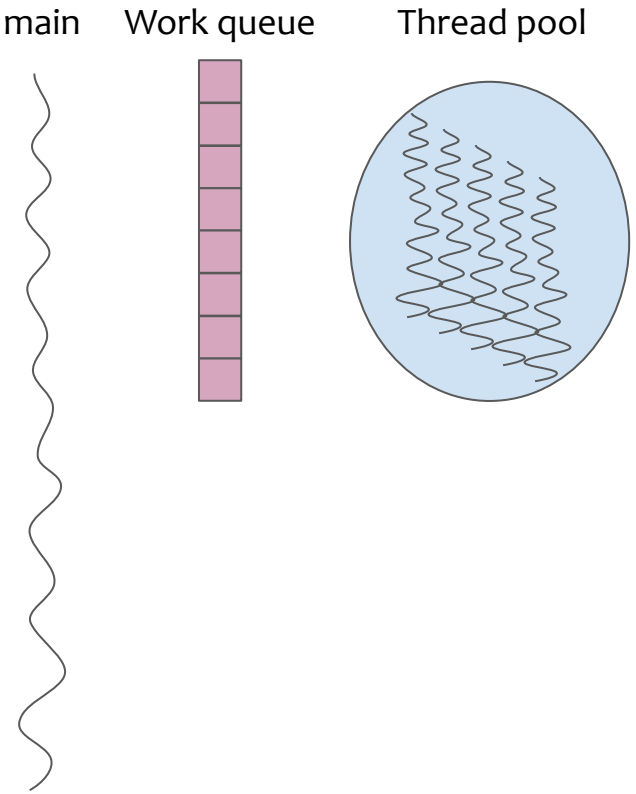


Pro: Very simple logic to implement

Con: Overhead of creating threads

Thread management: Thread pools

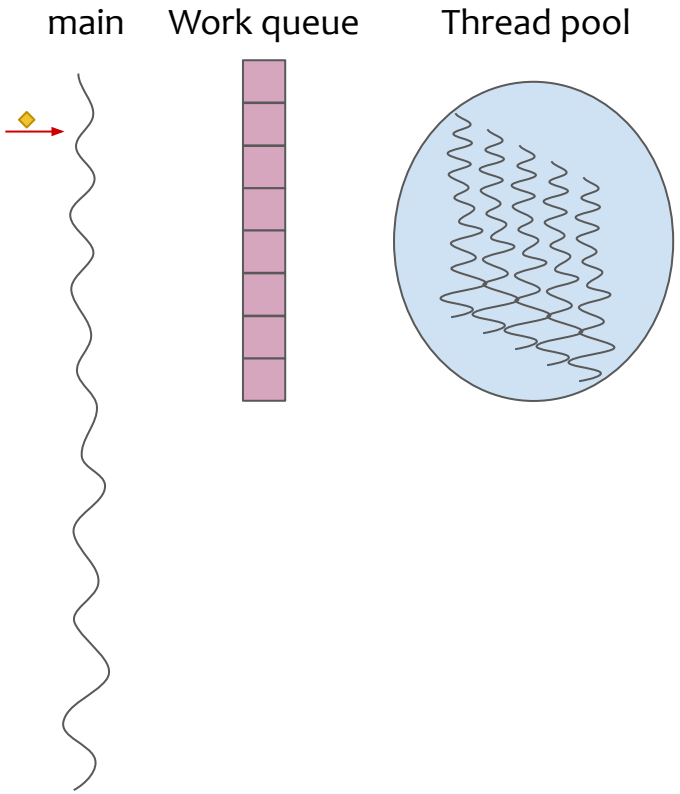
At initialization: The main thread creates a shared work queue and a pool of threads, then waits for requests



Thread management: Thread pools

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

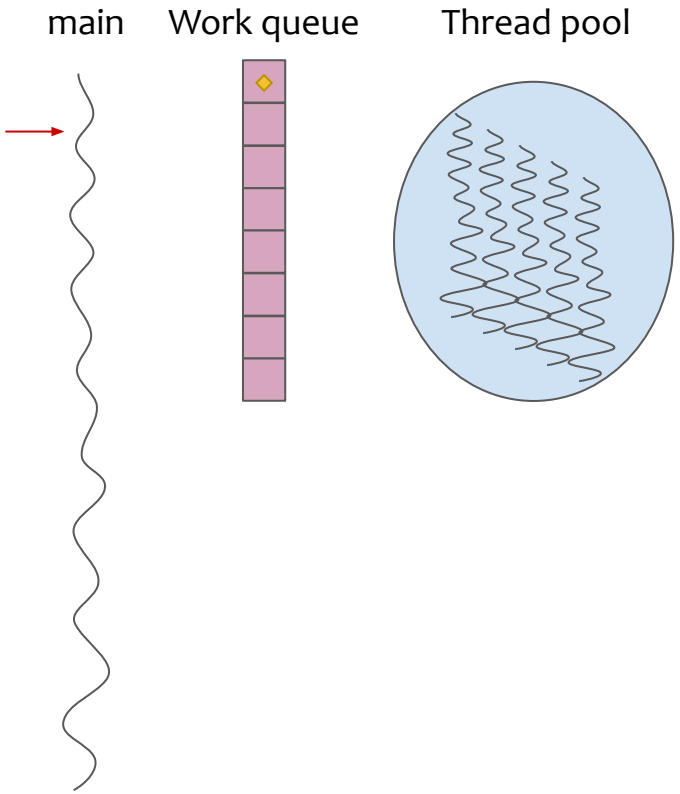
When a request arrives, the main thread enqueues it in the work queue



Thread management: Thread pools

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

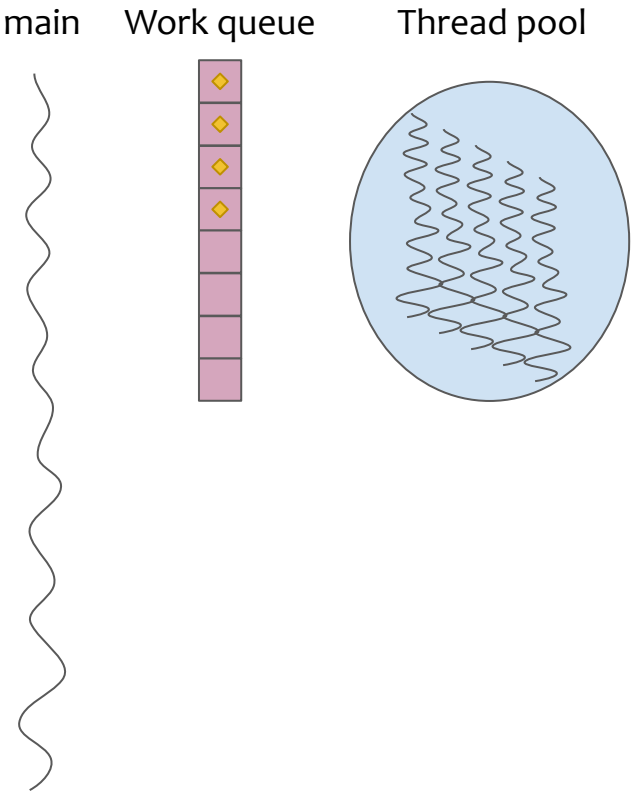
When a request arrives, the main thread enqueues it in the work queue



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



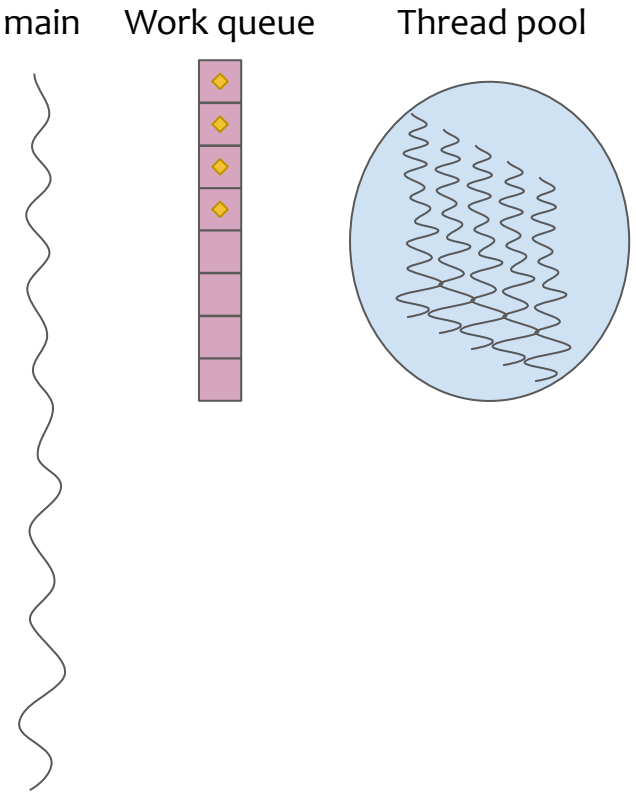
Thread management: Thread pools

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



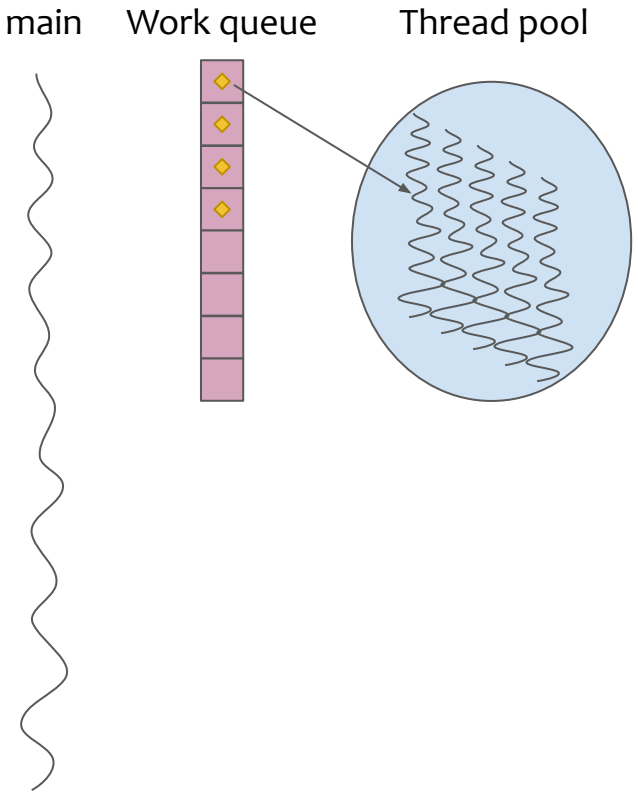
Thread management: Thread pools

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



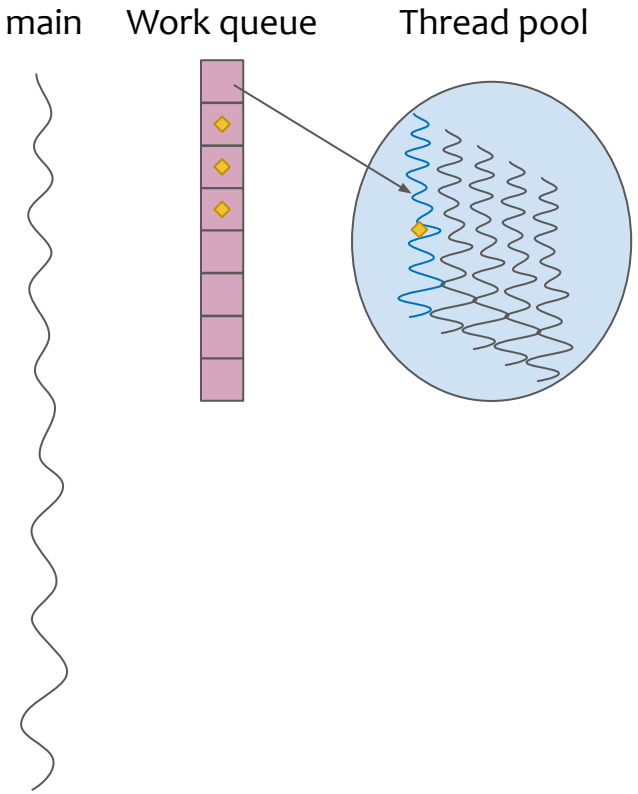
Thread management: Thread pools

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



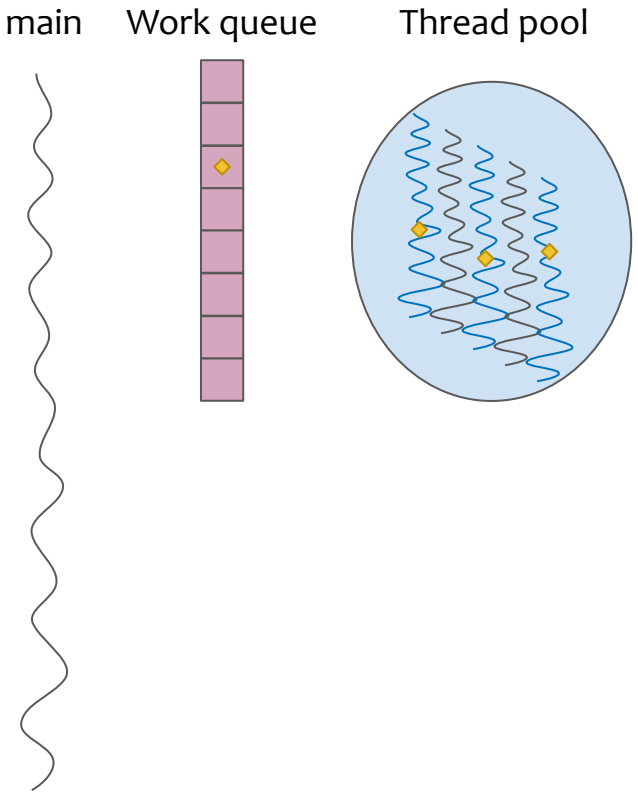
Thread management: Thread pools

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



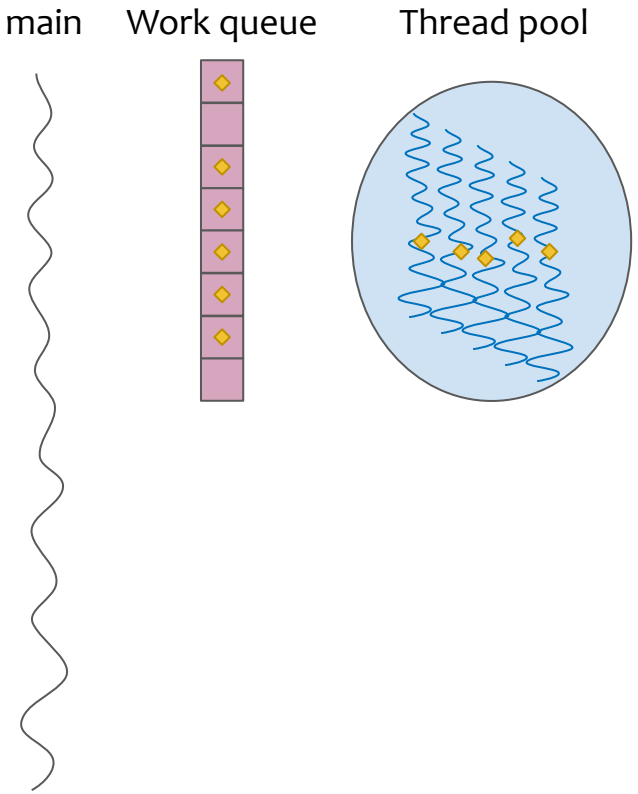
Thread management: Thread pools

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

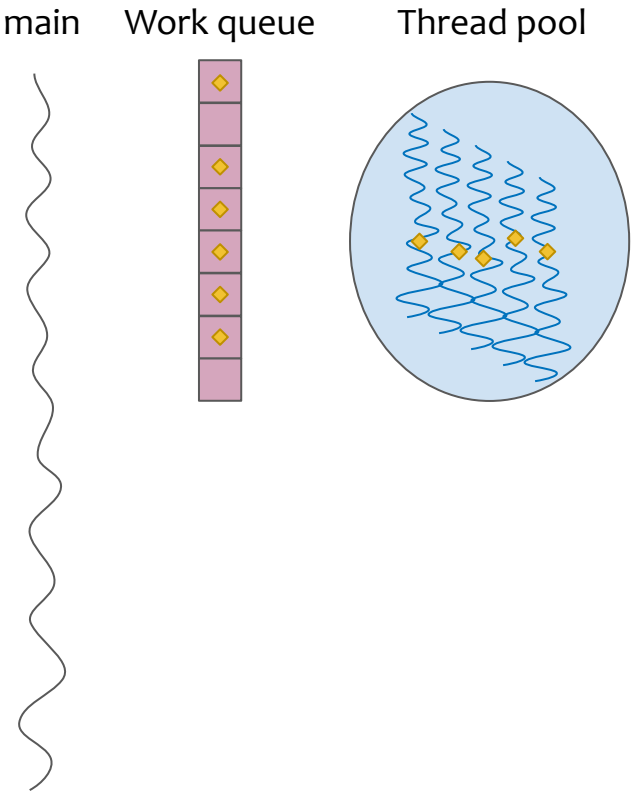
When a request arrives, the main thread enqueues it in the work queue

Worker threads:

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

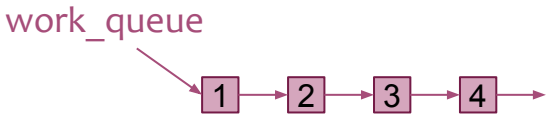
Pro: No thread creation overhead

Con: Synchronization required between workers



Why do we need synchronization?

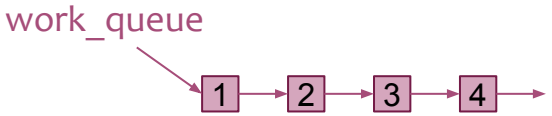
An easy implementation of a work queue would be a linked list, where each node is a request



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

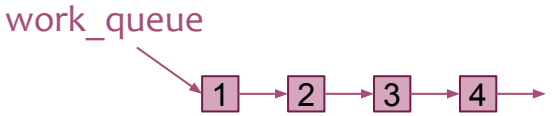
To get the next request, we need to read the current head of the list and modify the head

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

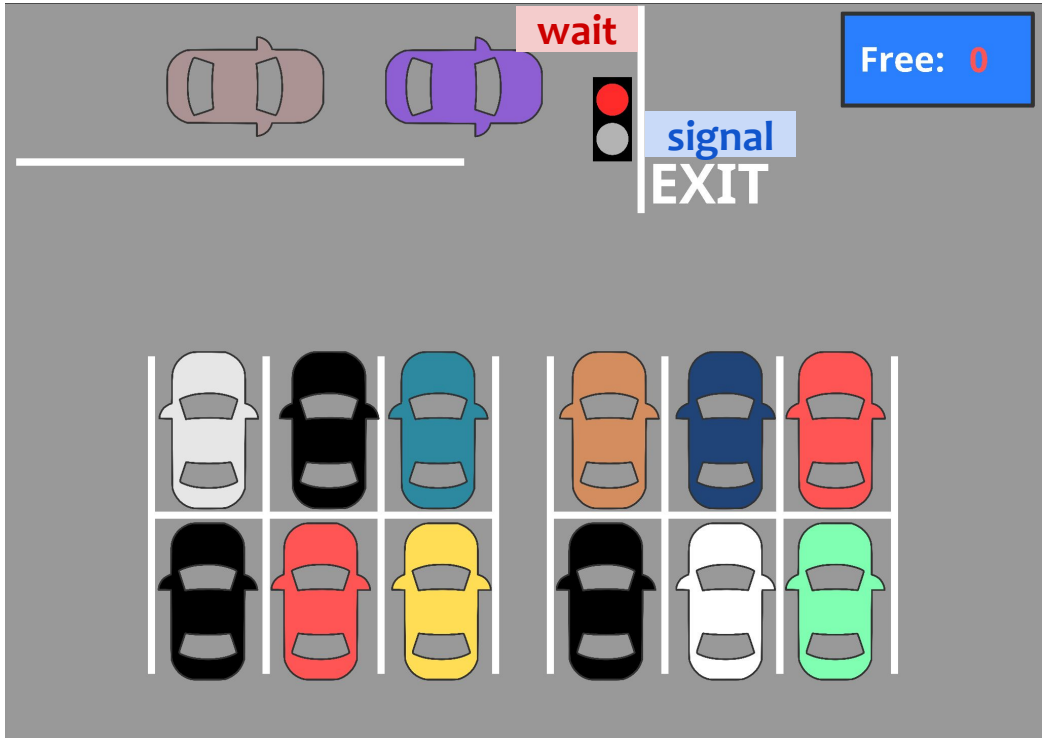
A semaphore is composed of:

- 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()`

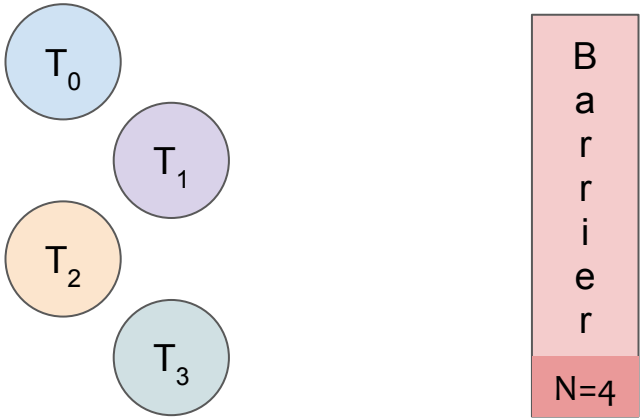
Release a permit [K += 1] (*signal*)

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

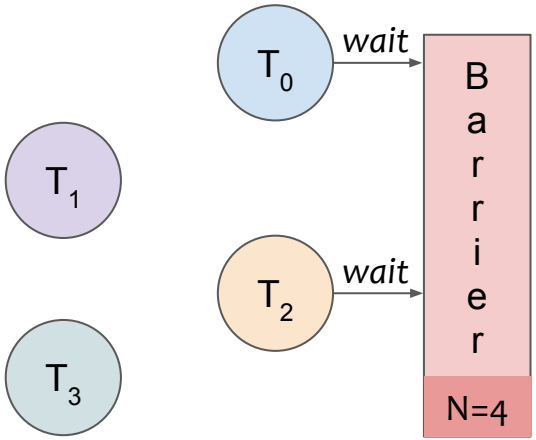


Barriers

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

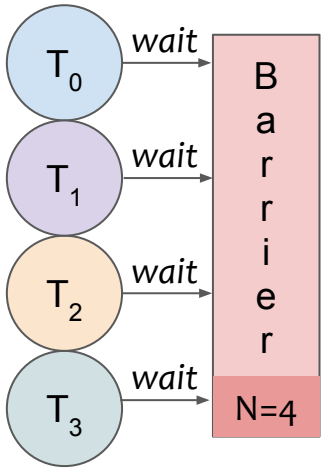


Barriers

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

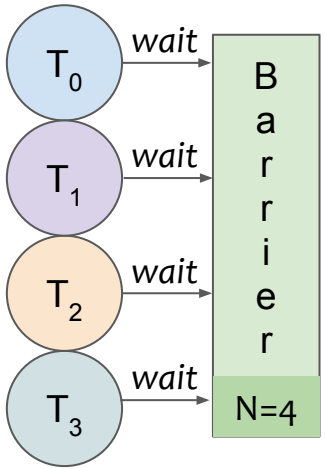


Barriers

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

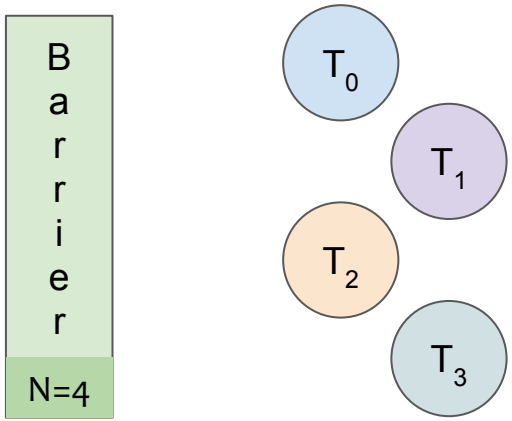


Barriers

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

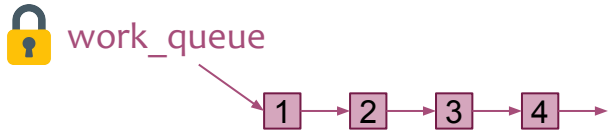
`void await()`

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    }
```

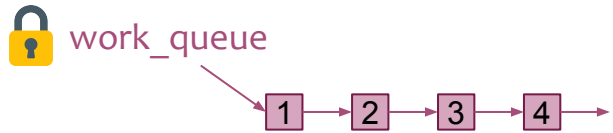
Worker thread

```
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?

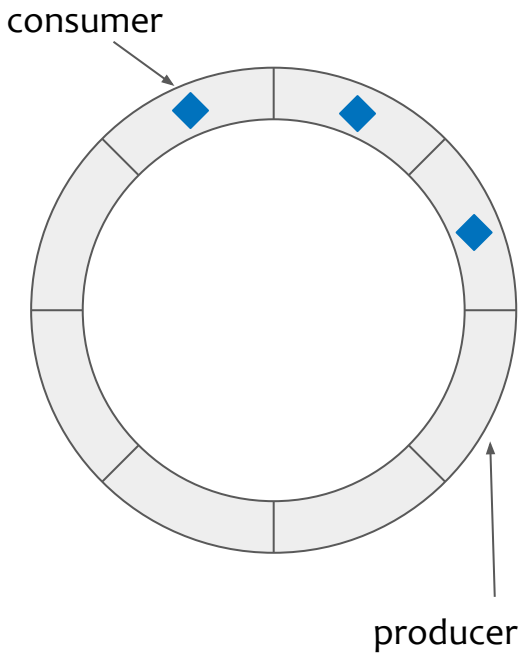
Lock-free data structure

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*



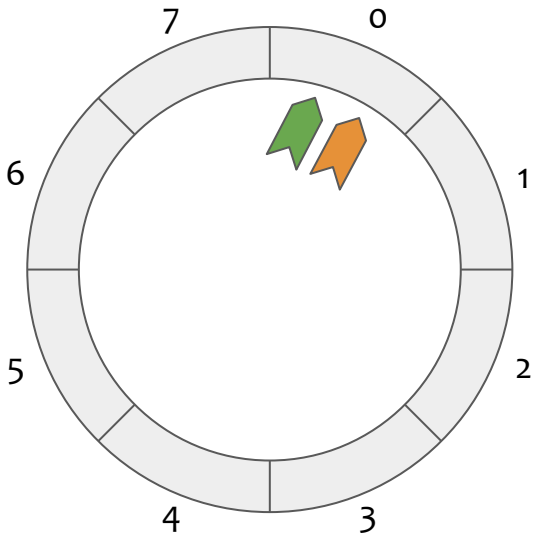
Lock-free ring buffer

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

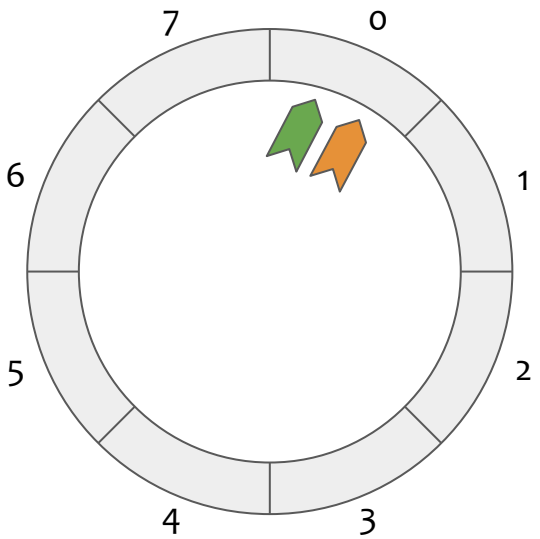


Lock-free ring buffer

```
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;
}
```



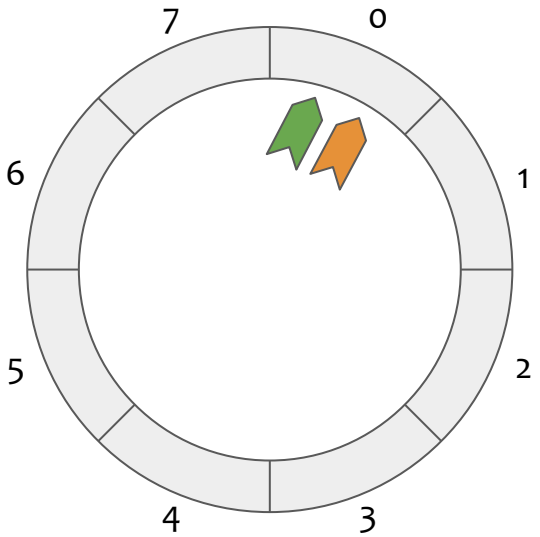
Lock-free ring buffer

```
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

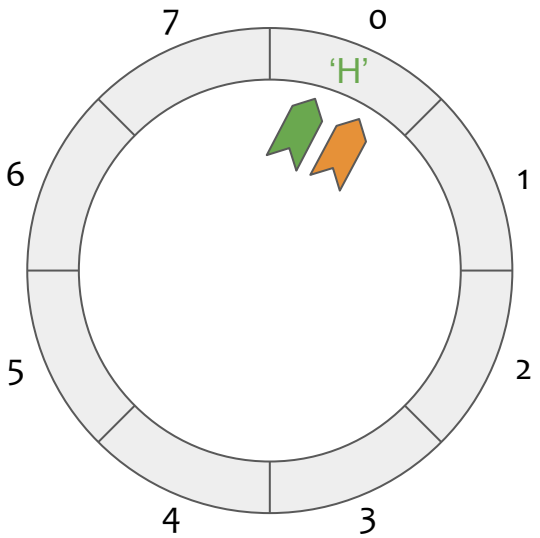


Lock-free ring buffer

```
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;
}
```

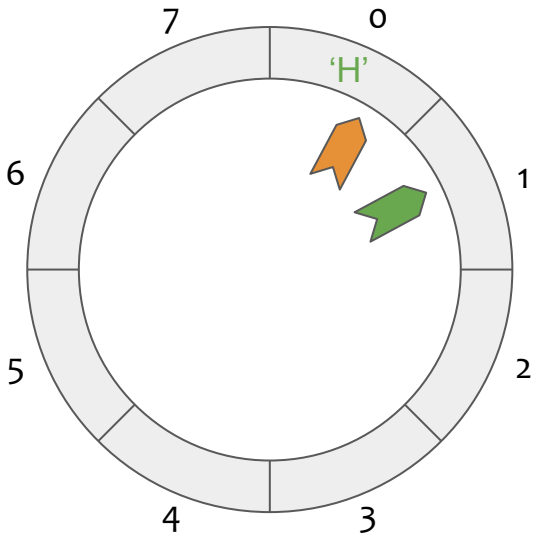


Lock-free ring buffer

```
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

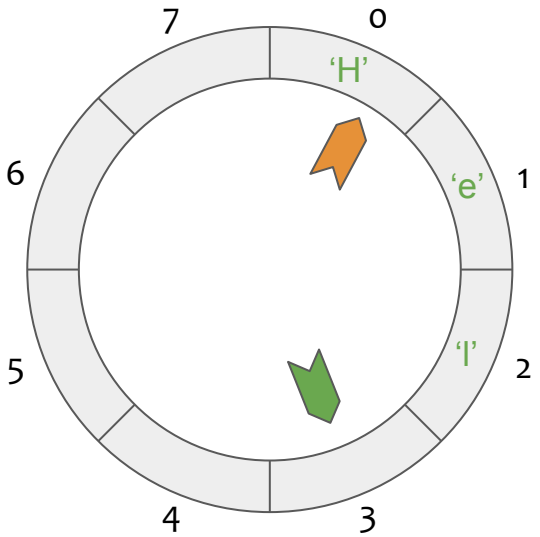
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

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

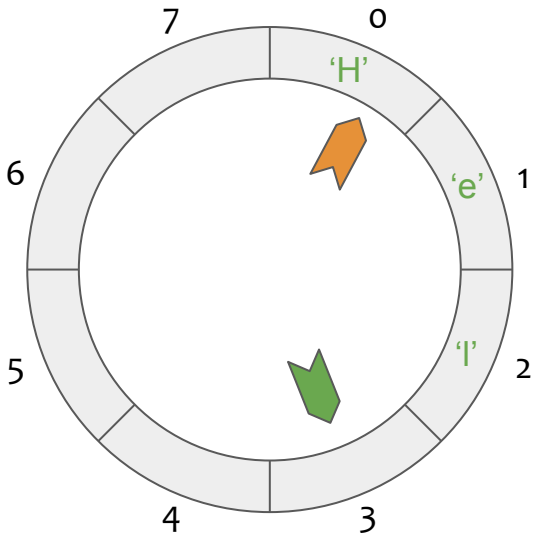
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

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

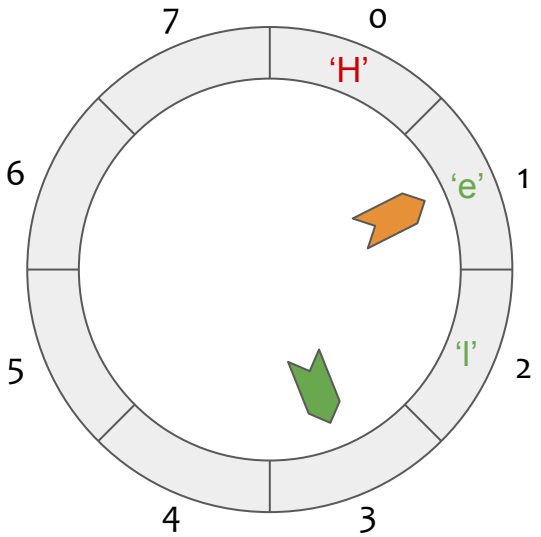
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

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

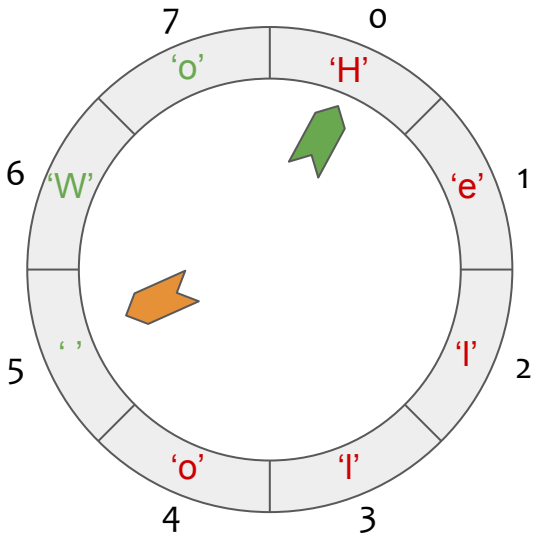
```
int capacity = 8;
char[] array = new char[capacity];
int prod_idx = 0;
int cons_idx = 0;

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

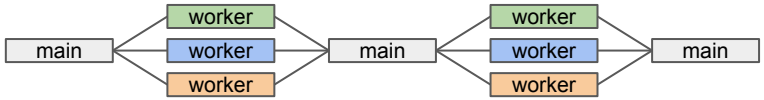
Wrong usage of synchronization can create different problems:

- **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.

Parallel programming patterns

Fork-join

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

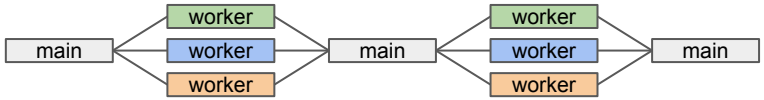


E.g., "*Spawning upon request*" version of our web server

Parallel programming patterns

Fork-join

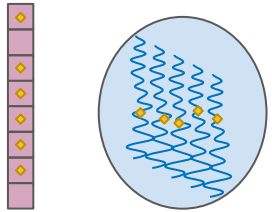
- Create threads to perform a task, *i.e.*, *fork*
- Wait for them to be done, *i.e.*, *join*



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

Outline

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

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

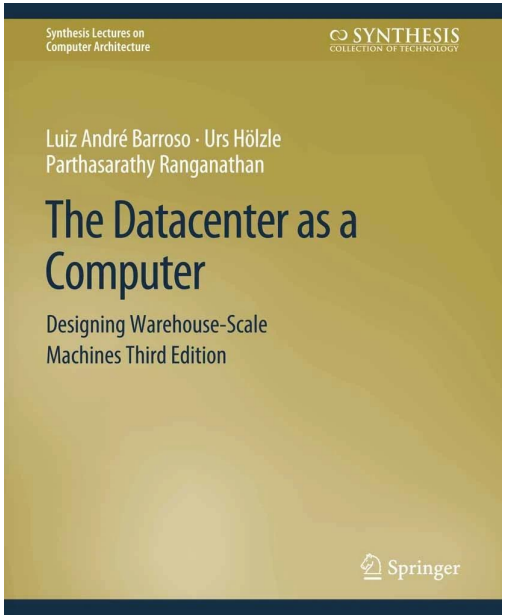
Scale-out (horizontal scaling): use multiple smaller machines instead of one big one

- 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

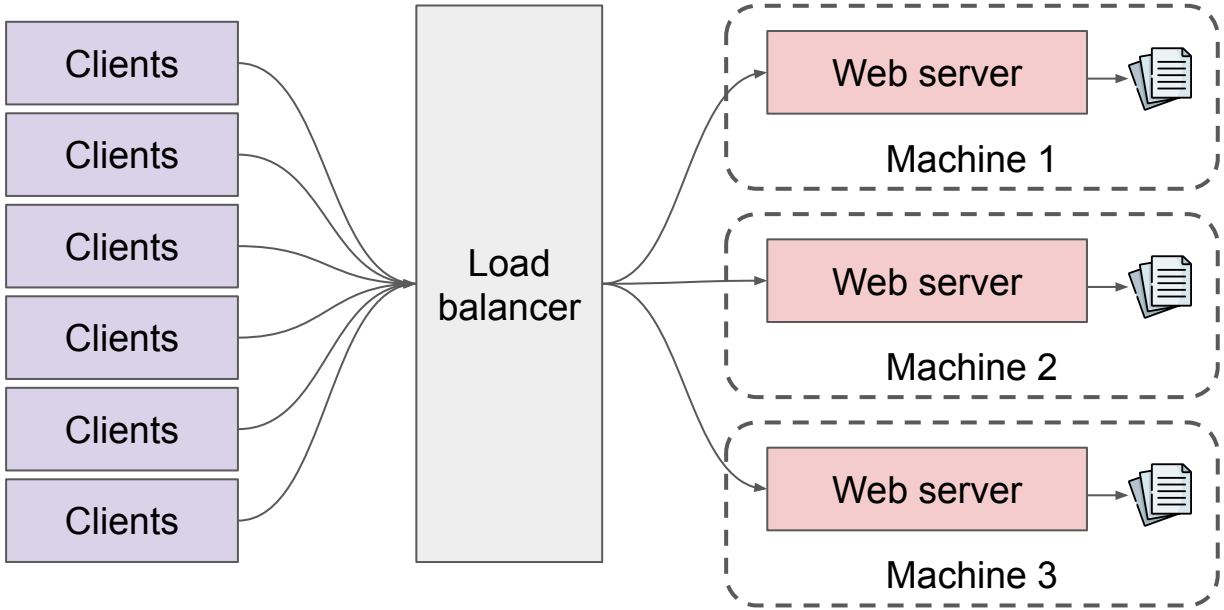
Techniques for fault tolerance are developed in the study of **distributed systems:**

- 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**



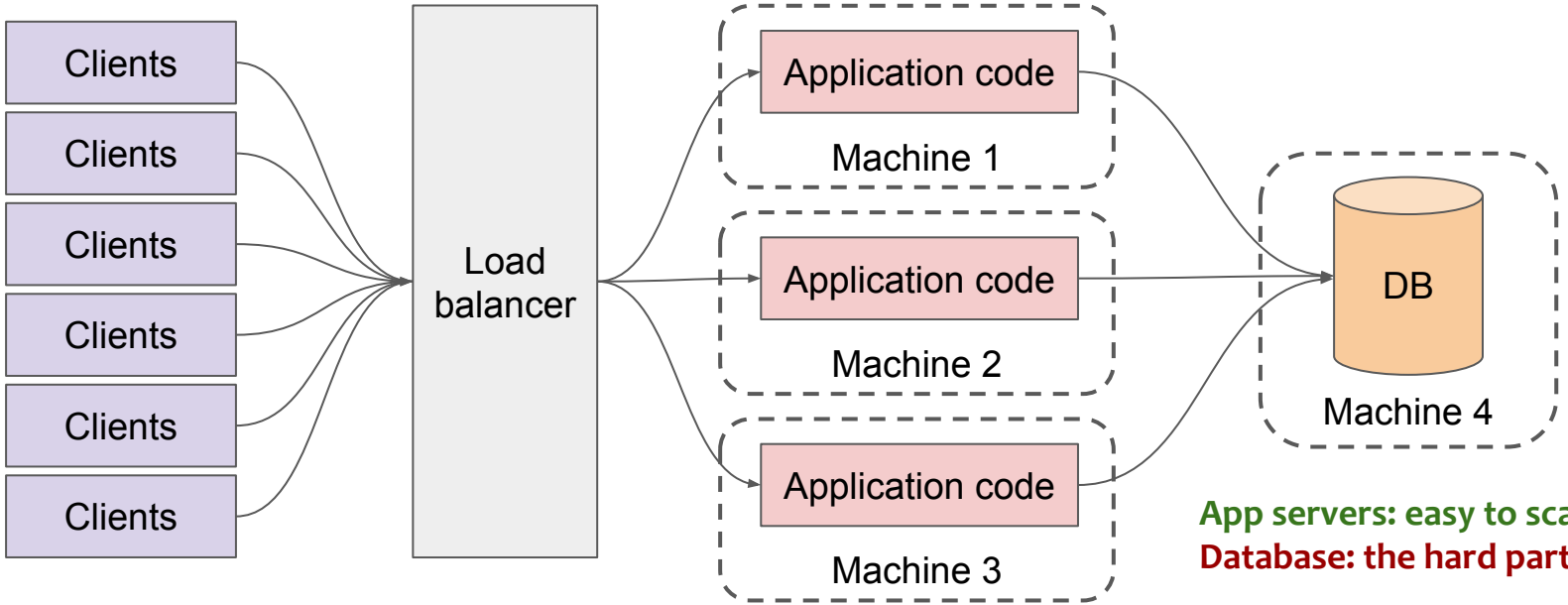
Each machine has a copy of the files that it is serving. As long as the files don't change, it's easy to add more web server machines.

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.

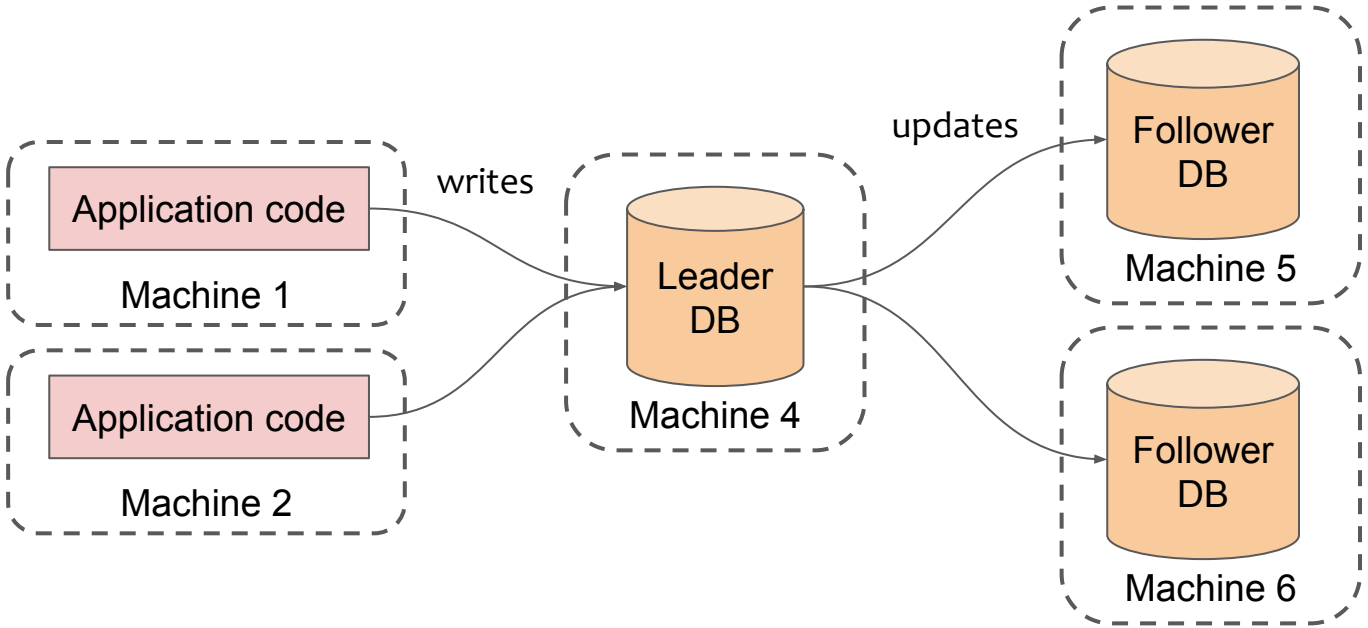
Databases and other stateful data systems are harder. Two main techniques:

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

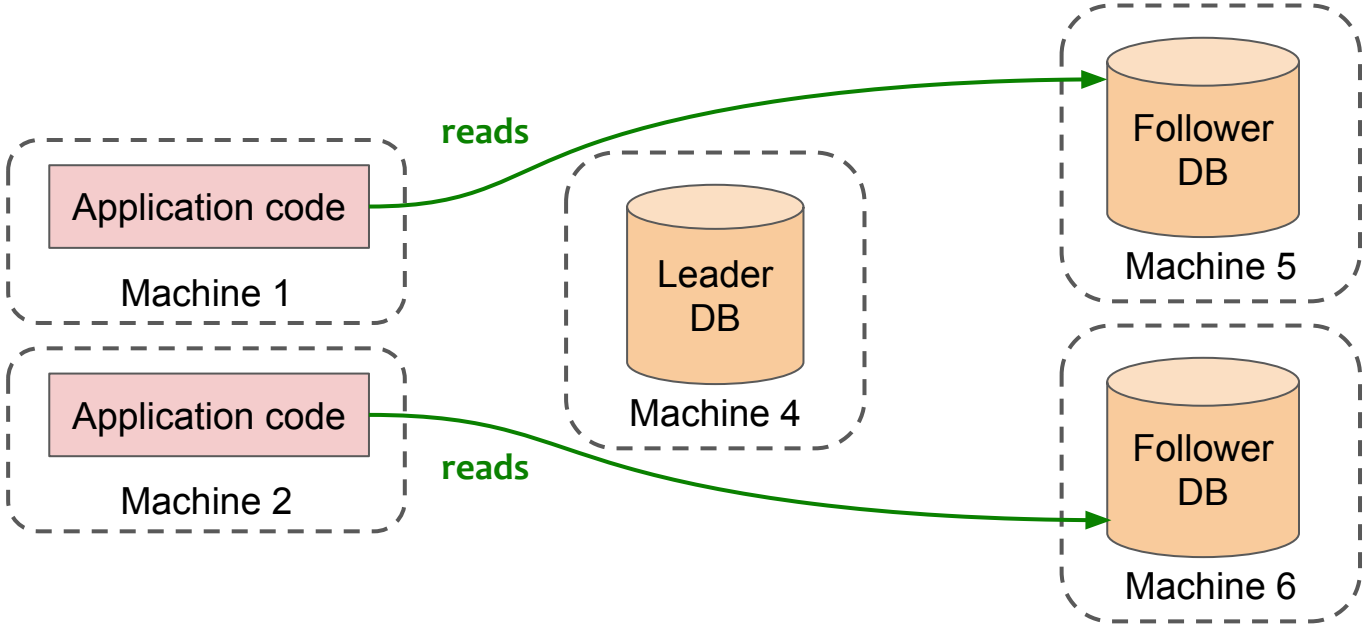
One leader, multiple followers. All database write requests are sent to the leader, who executes them and forwards the data updates on to the followers.



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

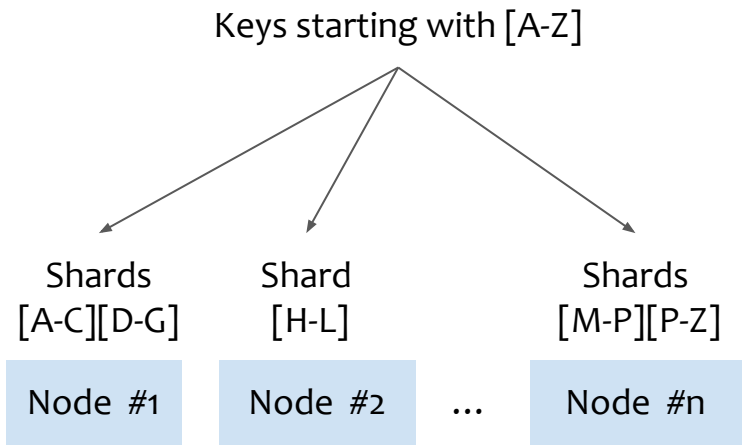
Sharding for scalability

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?**

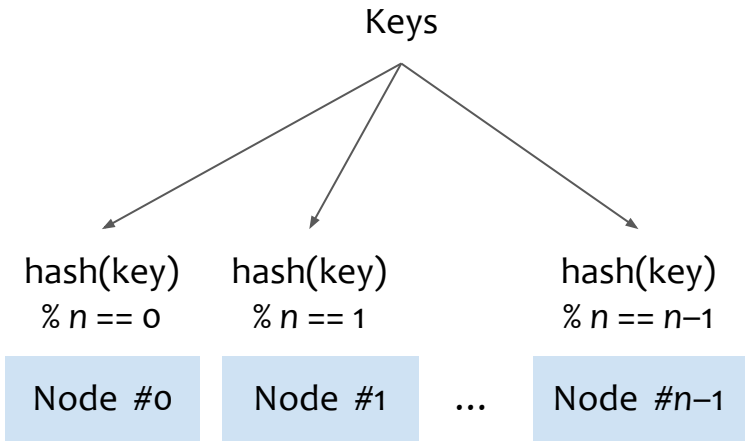
- **Example from L04:**
  In a key-value store, shard by the first letter of the key
- **Problem:** if all your keys start with "A", they will all go in shard #1 and the other shards will be idle!
- **Problem:** need to split a shard when it gets too big — how?



“Hash modulo n” sharding

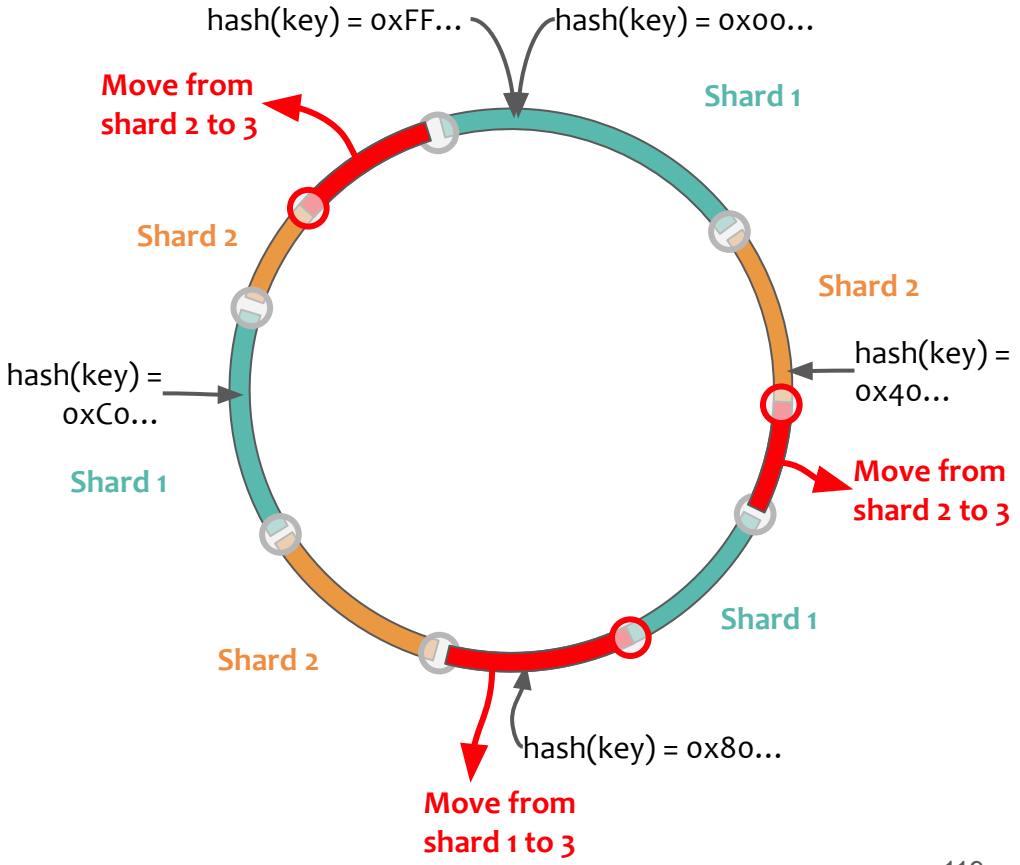
If you have *n* shards, use a hash function to map each key to an integer, then take modulo *n* to get an integer between 0 and *n*–1.

- **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)

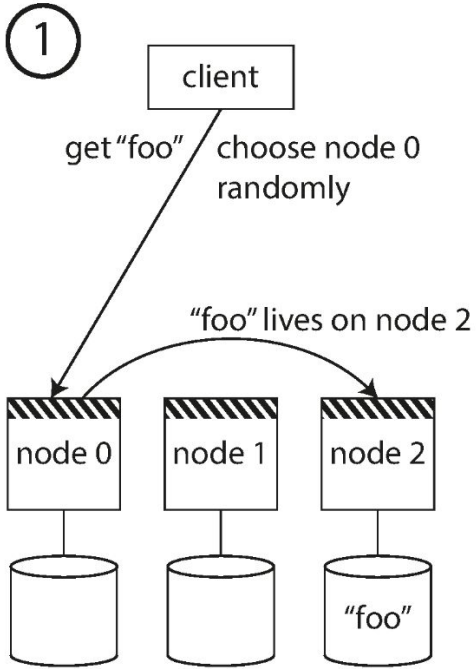


Consistent hashing

- 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*

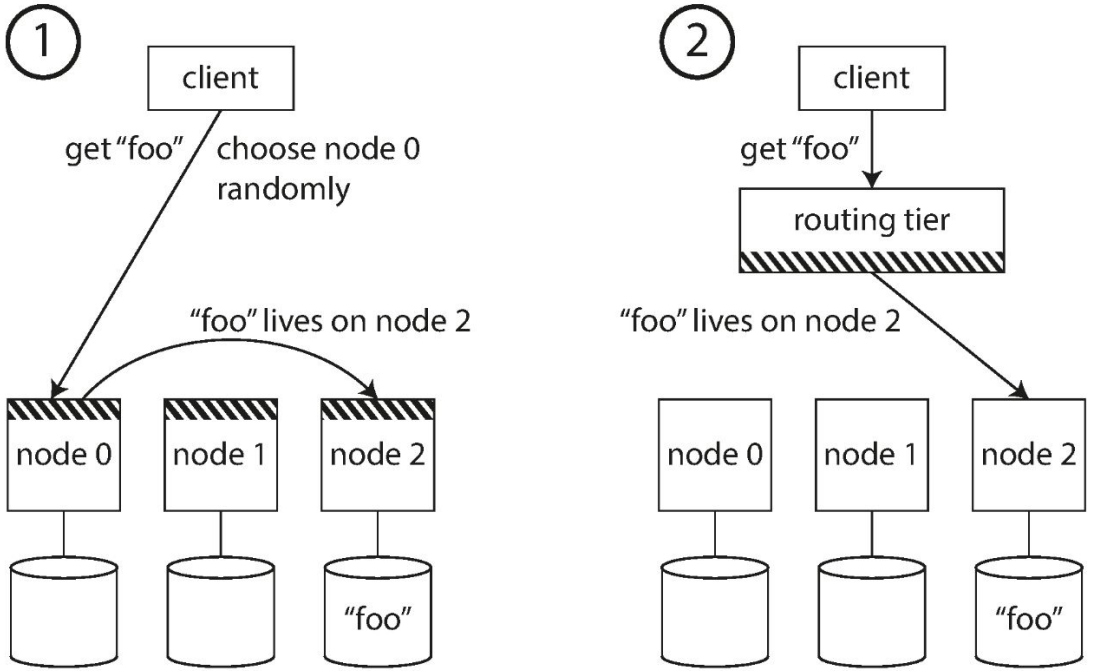


Routing queries to the correct shard



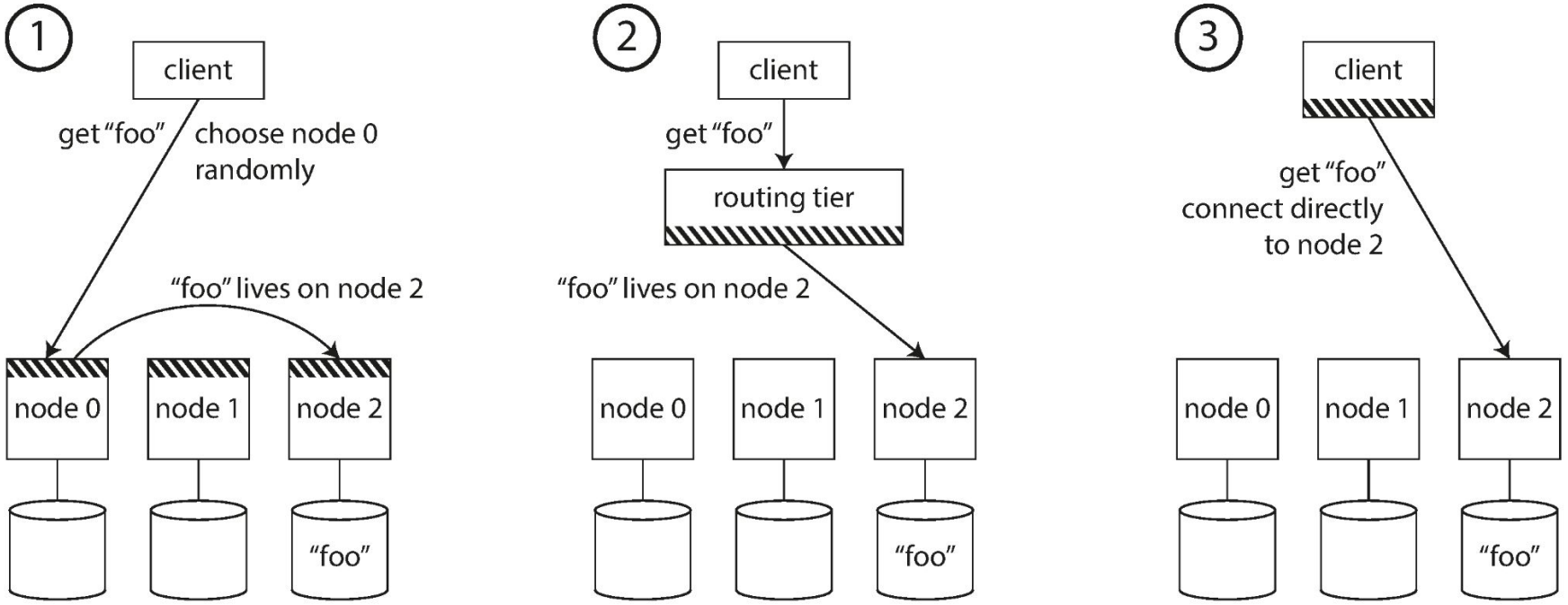
= the knowledge of which partition is assigned to which node

Routing queries to the correct shard



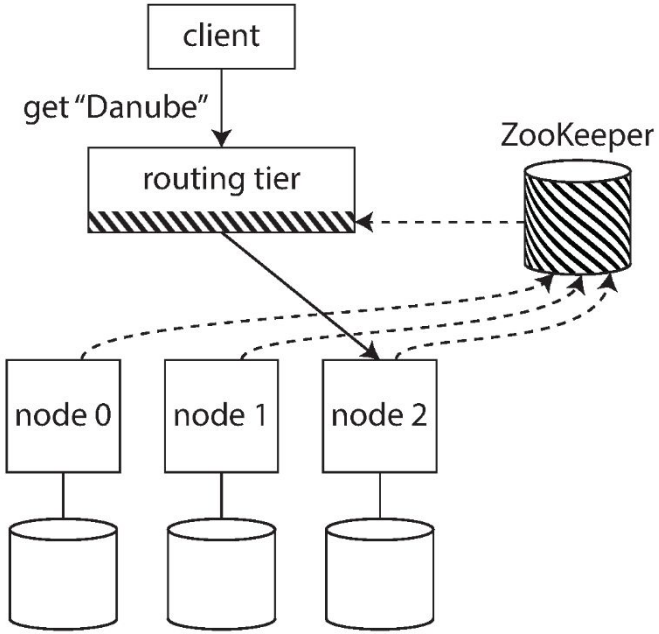
= the knowledge of which partition is assigned to which node

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

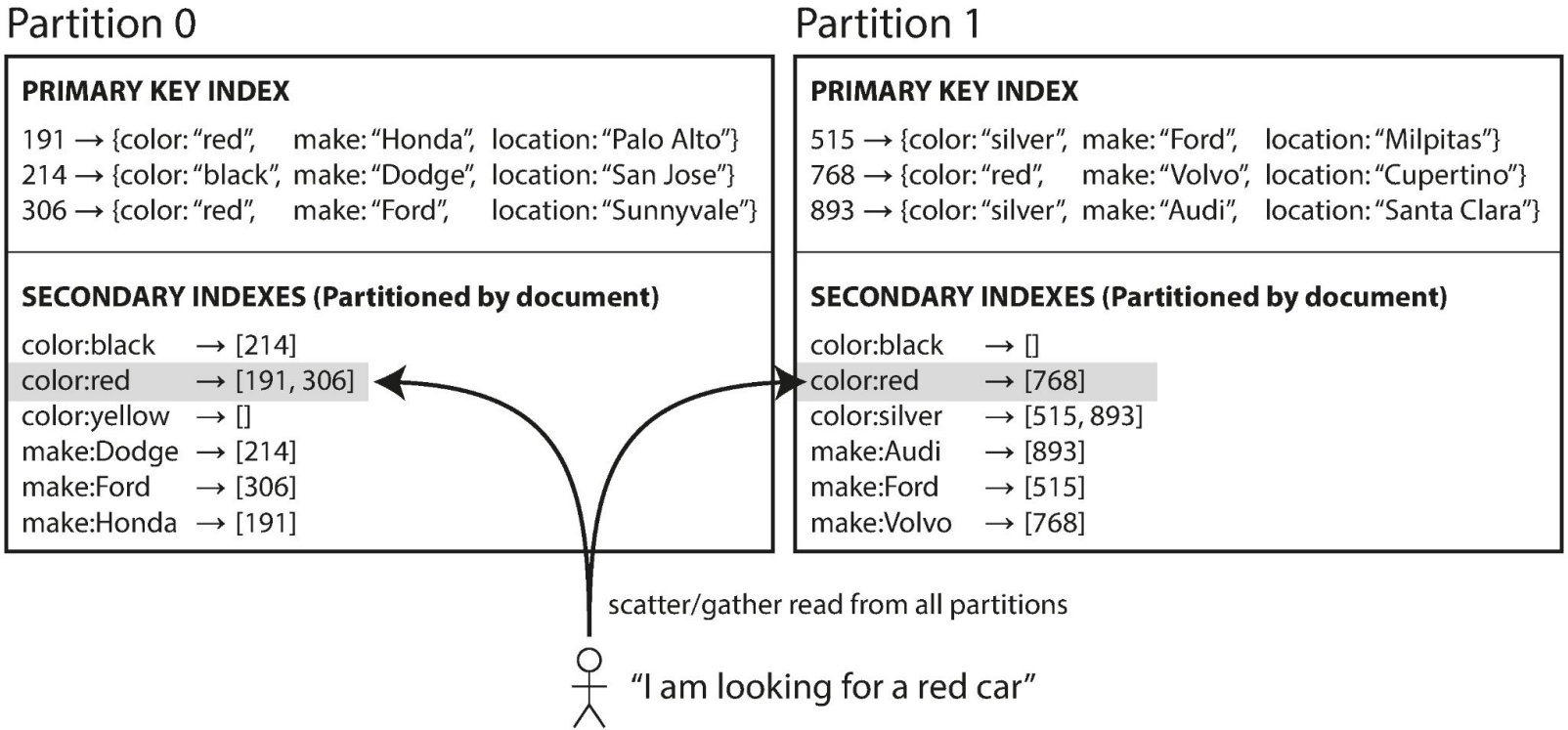
- In a key-value store, data is always accessed by key. If we shard by key, the key is sufficient information to know which shard you need to query.
- In other database types (especially relational/graph), sharding is harder. Instead of querying by primary key, it is common to use a secondary index.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

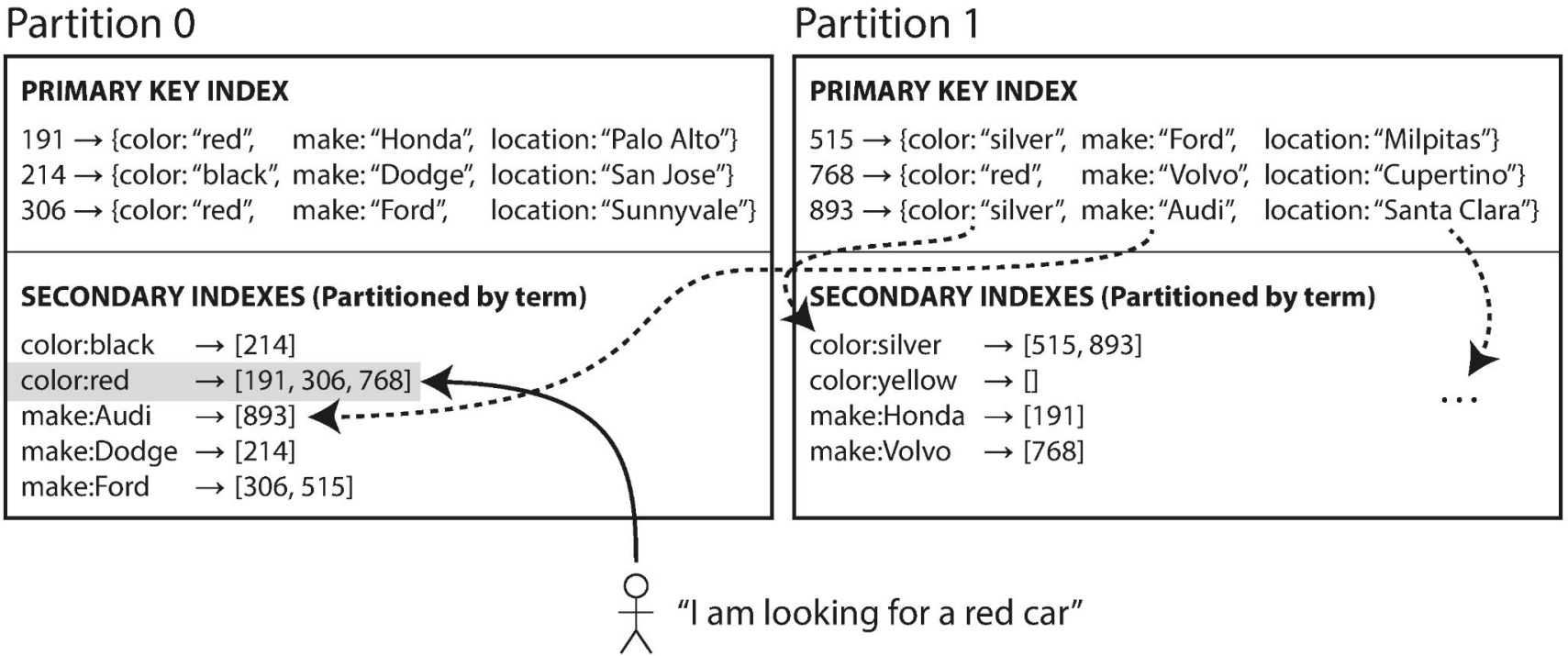
Local (document-partitioned) secondary indexes

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.

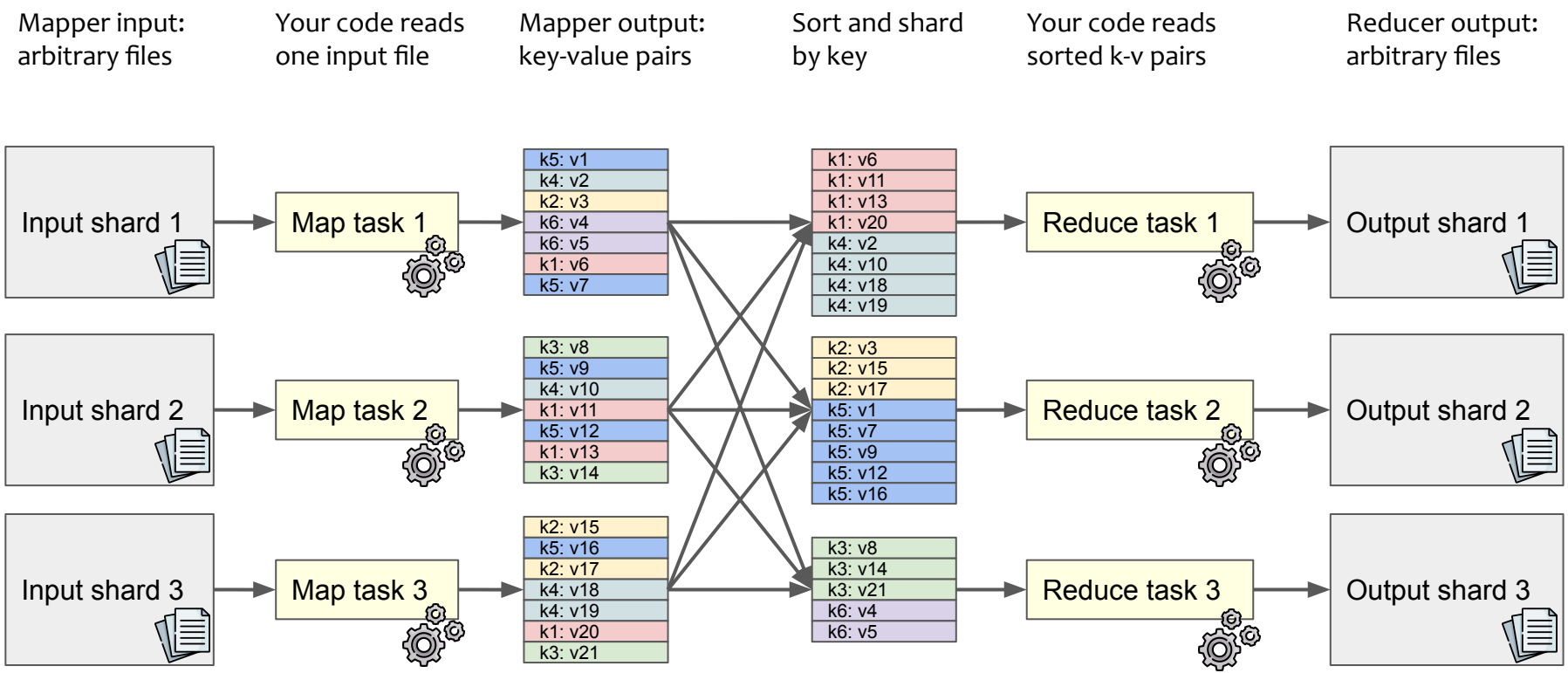
What about batch processing, when you want to **process all records**?

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)

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")!

An advanced topic,
so please don't panic!

Distributed data management (Key idea)

Let's understand the key idea using a simple example: "Managing books"

A small number of books:

- A single shelf can store the books
- No need to keep a catalogue of books

A large number of books:

- Multiple shelves are required to store the books
- We need a catalogue of books to locate a book

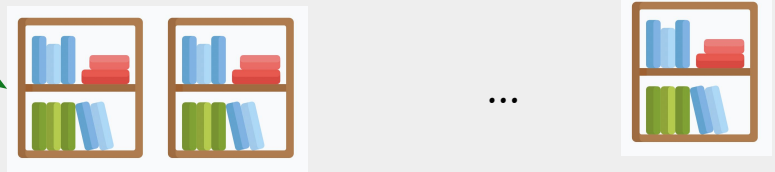
Library: A large-scale book management system!

Students (“clients”):
Searching a book!

Search the catalogue
(“The control path”)

Retrieve the book
(“The data path”)

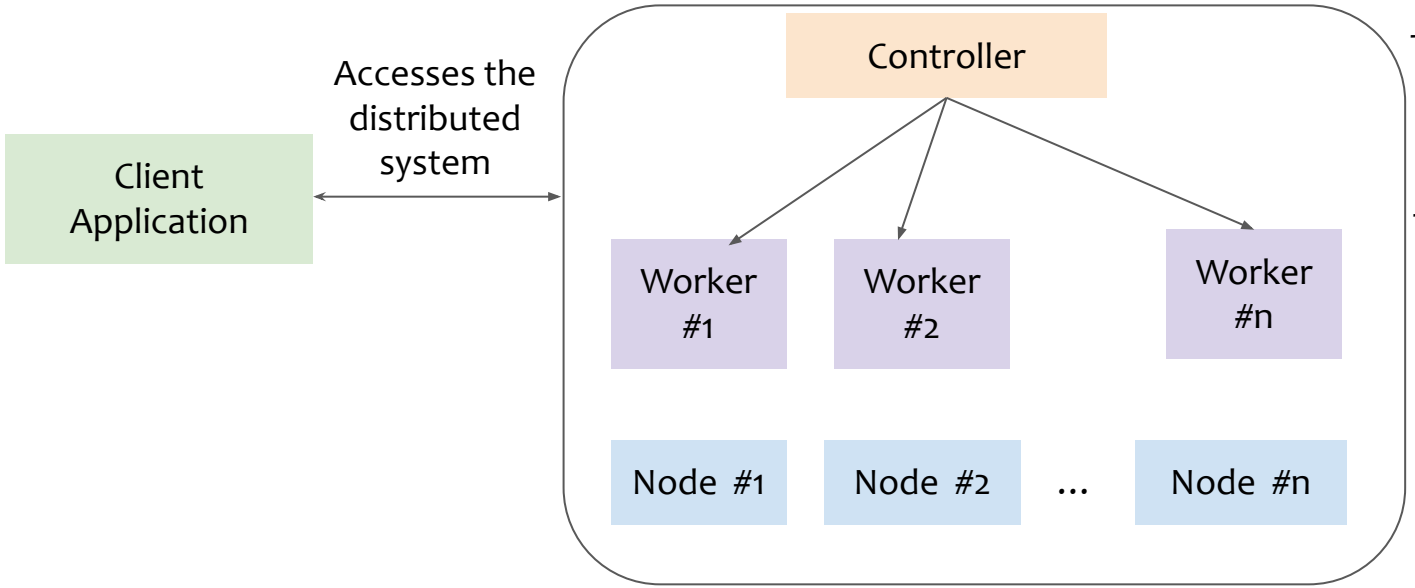
The catalogue (“the controller node”):
Search the catalogue for book location **(“metadata for data”)**



Book shelves (“Worker nodes”):
Storing the books **(storing the “data”)**

A library (a “distributed data management system”):
Managing and storing the books **(or “data”)**

Controller-worker architecture: A general recipe for building scalable distributed systems



Controller-worker distributed system architecture

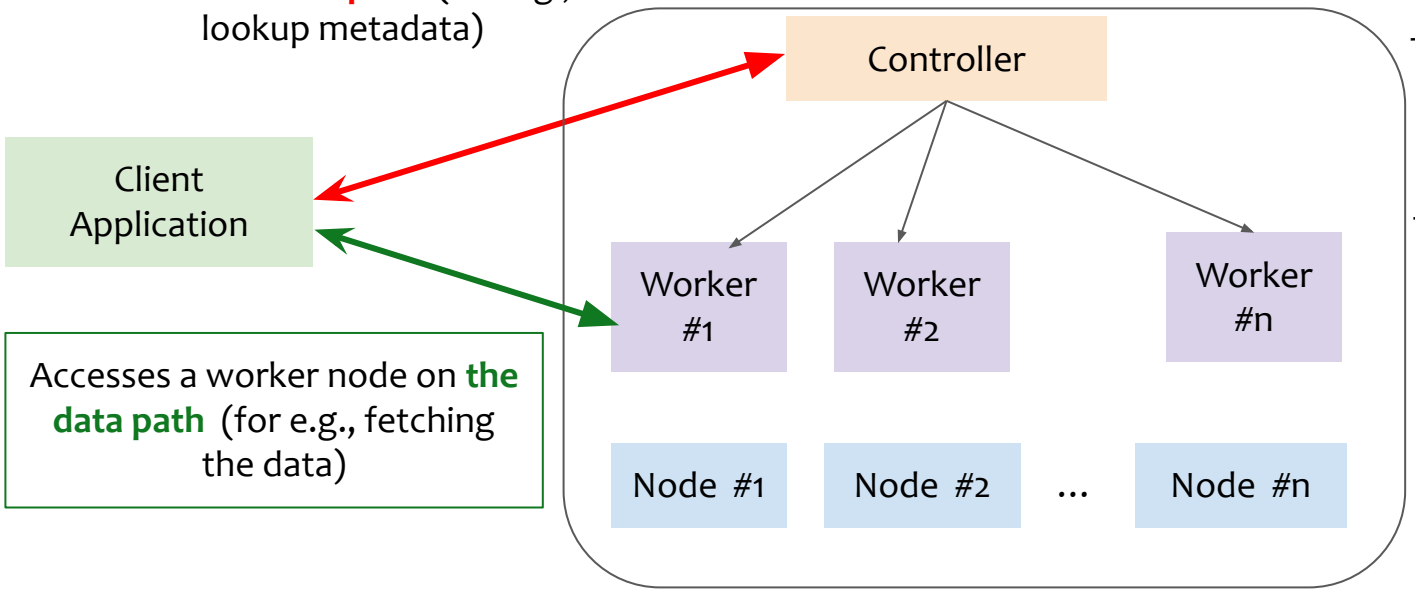
The **controller node** manages the worker processes

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

Machines/nodes in a datacenter

Controller-worker architecture: Control vs data paths

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



Accesses a worker node on **the data path** (for e.g., fetching the data)

The **controller node** manages the worker nodes

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

Machines/nodes in a data center

Controller-worker distributed system architecture

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

Summary

- **Part I:** Performance
  - Performance metrics
  - A systems approach to designing for performance
  - Measurement-driven approach to build high-performance systems
  - Design hints for performance
- **Part II:** Concurrency (or Scale Up!)
  - Why concurrency?
  - The thread model
  - Thread scheduling
  - Communication mechanisms
  - Parallelizing a program
  - Accelerators

Summary

- **Part III:** Scalability (or Scale Out!)
  - Scalability challenges
  - Scalability techniques
  - Scalable data management