Concurrency!

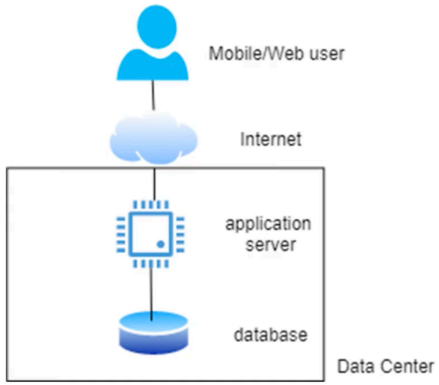
… in Java 7 ☺

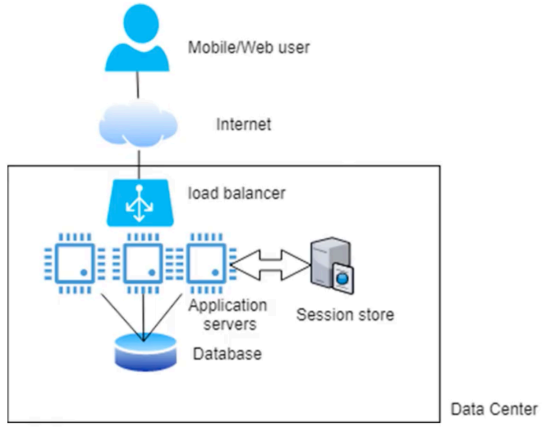
One tutorial https://docs.oracle.com/javase/tutorial/

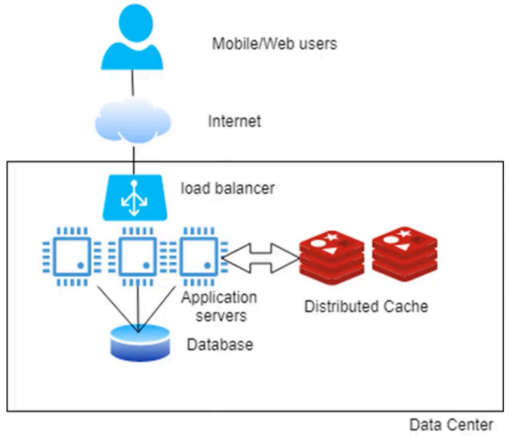
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

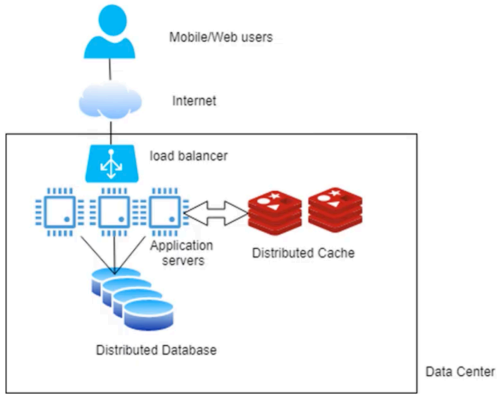
Please share others!!!

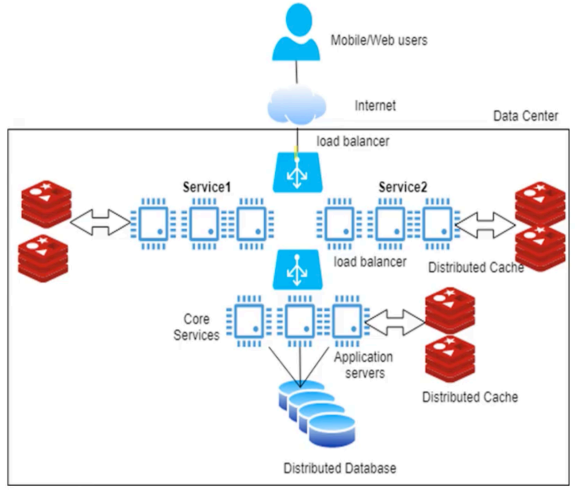
Scalability: tell the story!

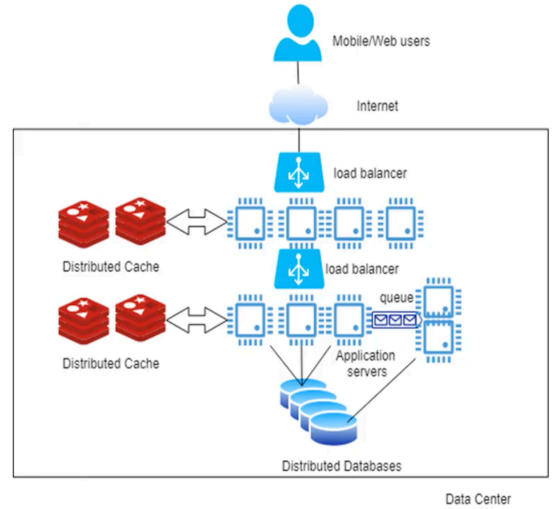












Scale UP versus Scale OUT?

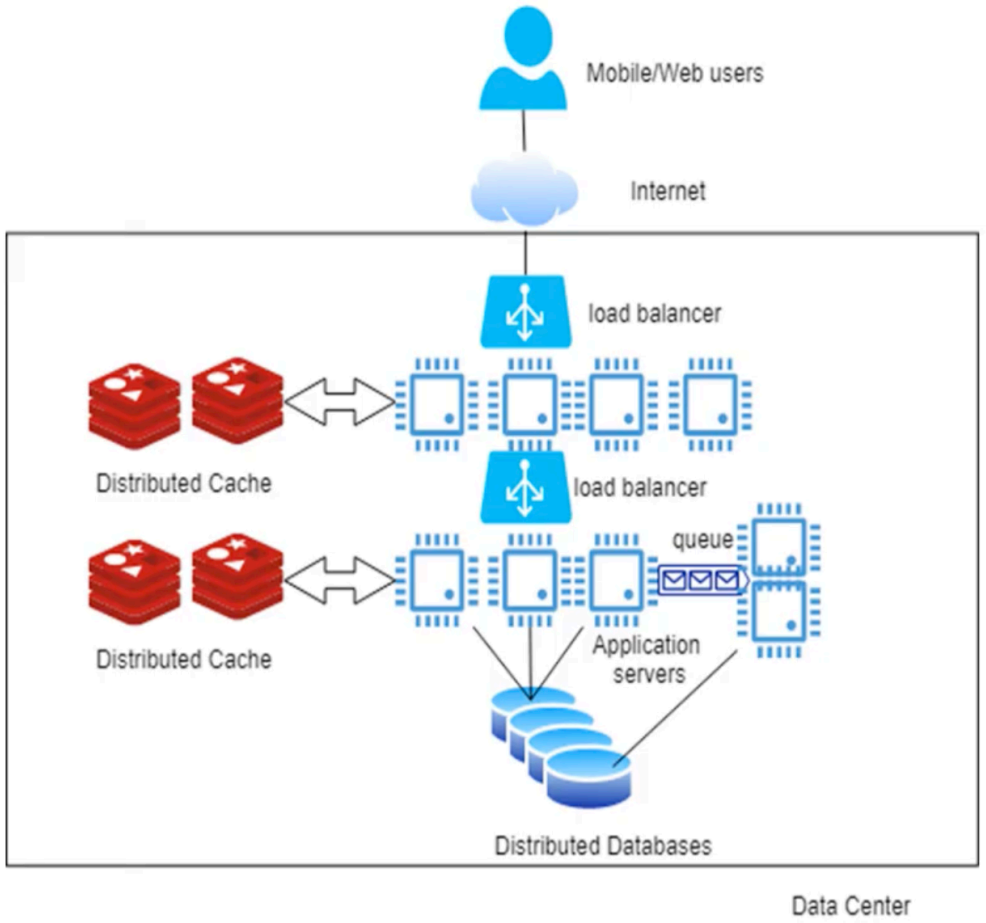
- What is the first bottleneck?
  - Yup, the database!
  - Can Scale Up the database…
    - More CPUs and RAM
    - *Still Single Point of Failure*
  - Can Scale Out the application servers

  - Also use a distributed cache
    - Can be quickly retrieved
    - Distributed key-value stores

  - AND if you need more, distributed databases!
    - SQL versus noSQL
    - Replication has tradeoffs
    - Cloud provider can handle the administration
      - dynamoDB

Eventual consistency

- Application servers use a queue
- Not waiting for the database!
- Request gets acknowledged
- Another service writes it out
- Data updated eventually
  - Server “promises” to do it!



Example Scalable Application (Seattle too!)

You work for Upic - a global acquirer of ski resorts that is homogenezing skiing around the world. Upic ski resorts all use RFID lift ticket readers so that every time a skier gets on a ski lift, the time of the ride and the skier ID are recorded.

We'll build a scalable distributed cloud-based system that can record all lift rides from all Upic resorts. This data can then be used as a basis for data analysis, answering such questions as:

- which lifts are most heavily used?
- which skiers ride the most lifts?
- How many lifts do skiers ride on average per day at resort X?

Could be *ANY* Scalable application!

You work for the analytics department at *Brandcamp*, a metaverse-based competitor to Bandcamp (let's NOT even get started on what that might mean... for now!).

Brandcamp provides a listening platform for music, and collects data on it's listeners. Every time a listener starts playing a song, the time spent listening to the song and the listener ID are recorded.

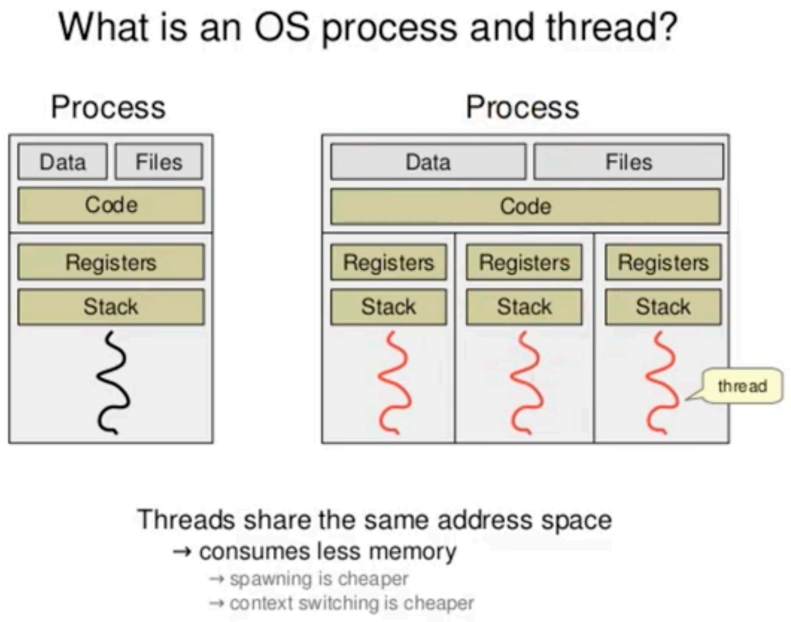
In this course, through a series of labs and milestones, we'll build a scalable distributed cloud-based system that can handle data sets like this. This data can then be used as a basis for data analysis, answering such questions as:

- which songs are most heavily favoured?
- which listeners play the most songs?
- how many songs do listeners play on average per day from band X?

Thread versus process?

- Every process has at least one thread!

- It's possible to have more…
  - Sharing memory (fast sharing!)
  - Can exploit parallelism!
  - Lighter weight than processes
  - Scheduling is nondeterministic!



- Simple Java threads example
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/simple.html

Defining and starting a thread

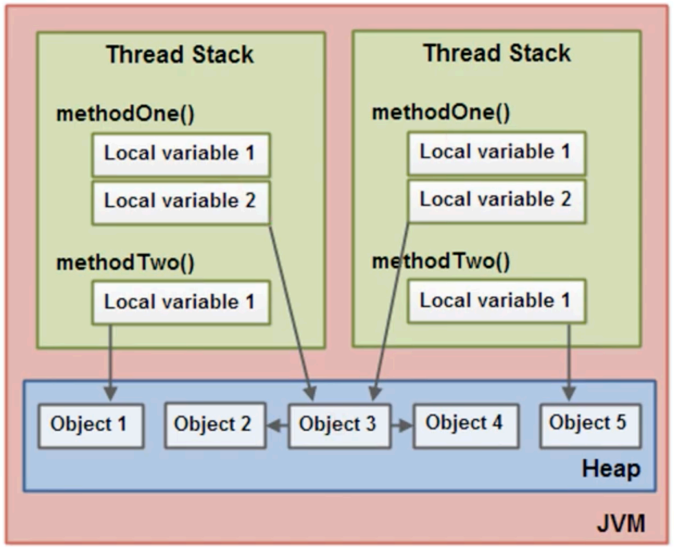
An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

Provide a Runnable object.
The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor

Subclass Thread.
The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Fundamental mechanisms



- Synchronization
- Coordination
- Thread pools
- Thread safe collections (phew!)
  - Libraries of data structures that are thread safe!
  - BUT THEY DO HAVE OVERHEADS!!!

Nondeterministic Behaviour means…

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

- Sharing is NOT caring!
- Every time you run your program it could output something different!
  - If threads share data
    - Assume thread 1 does: x=x+6
    - Assume thread 2 does: x=x+1
- Try it!
  - Can you make a RACE CONDITION happen in Java?
  - How do you prevent this?

Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
Writes register value to (x)	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)

Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)
Writes register value to (x)	

LOCKS! AKA mutexs, semaphores…

- YAY!
- Imposes ordering
  - shared variables can be made SAFE!
- ONE thread at a time
  - **Serialize** the access to the shared object
  - Locks other threads out!
- *Synchronize* keyword in Java!
  - Creates a critical section
  - Mutually exclusive
  - ONE thread at a time, others block!
  - Known as a *Monitor lock*

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

```
synchronized public  void inc() {
  count++;
}
```

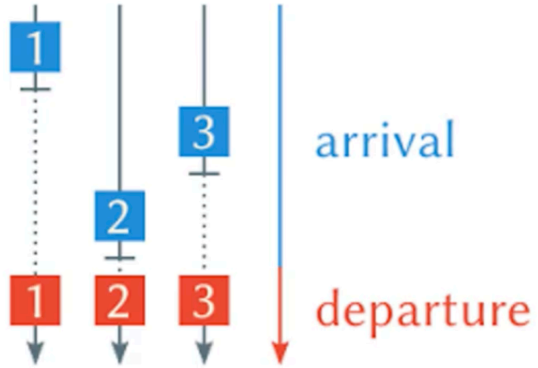
What about coordination?

- Dining Philosophers
- Readers/Writers
- Producer/Consumer

- Lots of examples out there!!!
  - Mechanisms are evolving
- One for Producer/Consumer here
  - https://docs.oracle.com/javase/tutorial/displayCode.html?code=https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ProducerConsumerExample.java

ANOTHER solution! Barrier synchronization

- A couple of ways to do it…
- Countdown Barrier
- Once we allllllll hit zero we can GO!



- Initialize a count
  - await()/countdown()

- Example https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html

CountDownLatch: Like a gate!

one or more threads **wait** until a set of operations being performed in other threads completes.

A CountDownLatch is initialized with a given *count*.

The await methods block until the current count reaches zero due to invocations of the countDown() method

This is a one-shot phenomenon -- the count cannot be reset.
If you need a version that resets the count, consider using a CyclicBarrier.

```
class Driver { // ...
  void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);

    for (int i = 0; i < N; ++i) // create and start threads
      new Thread(new Worker(startSignal, doneSignal)).start();

    doSomethingElse();            // don't let run yet
    startSignal.countDown();      // let all threads proceed
    doSomethingElse();
    doneSignal.await();           // wait for all to finish
  }
}

class Worker implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;
  Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
     this.startSignal = startSignal;
     this.doneSignal = doneSignal;
  }
  public void run() {
     try {
       startSignal.await();
       doWork();
       doneSignal.countDown();
     } catch (InterruptedException ex) {} // return;
  }

  void doWork() { ... }
}
```

KNOW there is much more!

- *Waitgroups* in Go!
  - Many examples
  - This one came from
    - https://www.educative.io/edpresso/how-to-use-waitgroup-in-golang!

```go
package main

import (
  "fmt"
  "sync"
)


func main() {
//Goroutine execution 3
var wg sync.WaitGroup
wg.Add(2)
  go func(){
    count("sheep")
    wg.Done()
  }()

  go func(){
    count("cow")
    wg.Done()
  }()
  wg.Wait()
}
func count(thing string){
  for i:=0; i<=3; i++{
    fmt.Println(thing)
  }
}
```