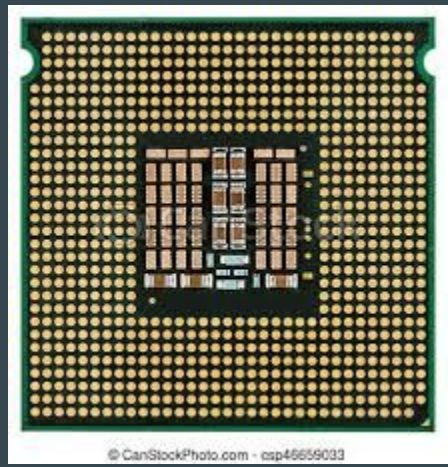
Concurrency Models for a multicore world

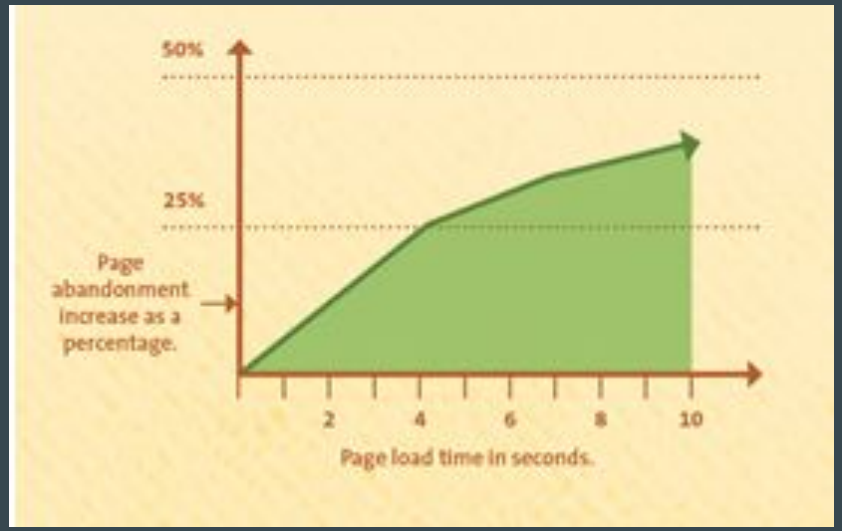
...

Shared State vs Share Nothing



El tiempo que se pierde, es dinero...

- Público demandante requiere ser lo más veloces posibles
  - 1/4 abandona páginas que tardan más de 3 segundos



A 1 SECOND DELAY IN PAGE RESPONSE CAN RESULT IN A 7% REDUCTION IN CONVERSIONS.

If an e-commerce site is making $100,000 per day, a **1 second page delay could potentially cost you $2.5 million in lost sales every year.**

... y el tiempo no vuelve.

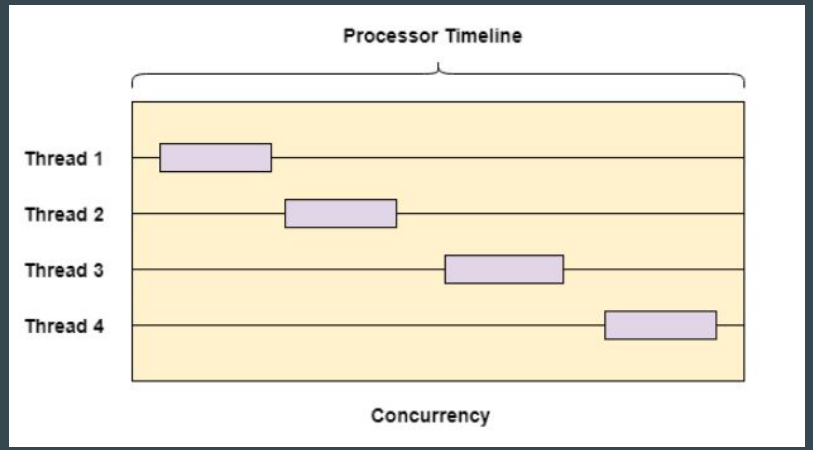
La experiencia del usuario, es dinero...

... y la frustración no se borra, se comparte.

¿Cómo acelerar nuestros programas?

- *Concurrency*

Concurrencia es la habilidad de hacer más de una cosa en un periodo de tiempo, pero no más de una cosa por unidad de tiempo.



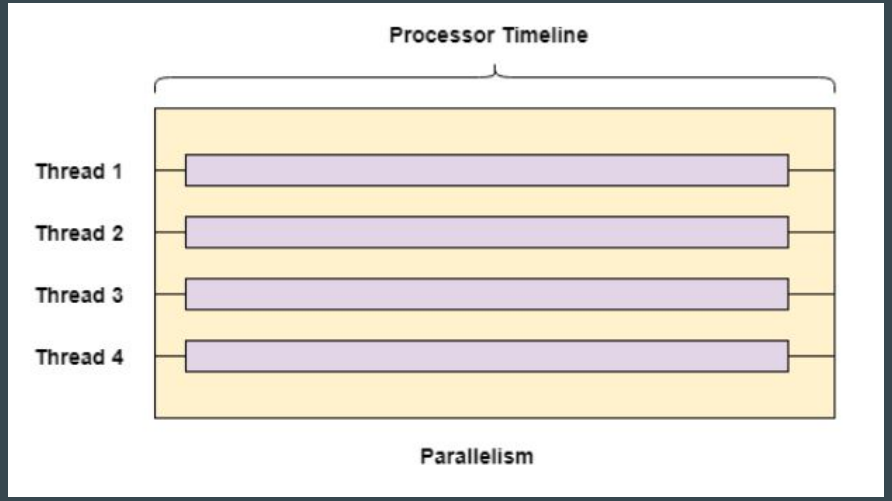
PARALLELISM

beautiful It's thing. a

¿Cómo acelerar nuestros programas?

- *Parallelism*

Paralelismo es la habilidad de hacer más de una cosa al mismo tiempo



¿Cómo trabajamos la concurrencia y el paralelismo?

- Procesos independientes (computación distribuida)
- Proceso subdivido en hilos de ejecución
  - Comparten recursos / estado
  - Son más livianos que un proceso
  - Crearlos es costoso para la VM (Threadpools al rescate)
  - Execution Context

Model 1. Mutable Shared State

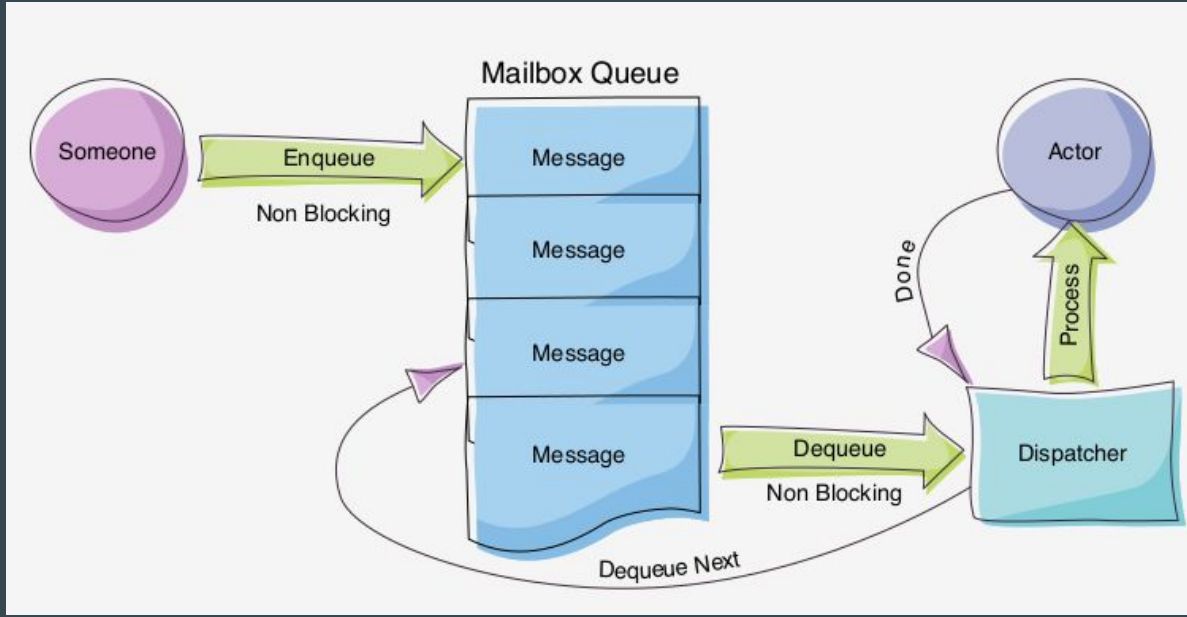
- Modelo que se basa en compartir recursos entre los distintos hilos de un proceso
- Compartir siempre fue un problema:
  - Race Condition (los threads pueden pisar el estado de una variable según el orden y tiempo de ejecución)
  - Contention (conductor se pelea en la cabina de peaje y no deja avanzar al resto)
  - Deadlocks (un arco y una flecha y dos niños que no se ponen de acuerdo para jugar)
  - Starvation (nunca le dan tiempo para ejecutarse)
  - Livelock (dos threads se piden permiso mutuamente y ninguno avanza)

Model 2. Share Nothing

- Nadie puede tocar tus cosas sin permiso
- Nadie puede cambiar tu comportamiento sin pedirtelo
- En consecuencia, podemos paralelizar no sólo en un nodo multicore, sino que es transparente escalar horizontalmente
- Modelos más conocidos
  - Message Passing: Actores (Akka, Elixir, Erlang) - ScyllaDB Seastar es Actores en C++
  - Channels (Go)

Share Nothing - the Actor Model (1973 - Carl Hewitt)

- Actor como unidad de computación asíncrona
- Comunicación entre Actores a través de mensajes inmutables



Akka - The Actor System

- Los actores viven en un Actor System
- Los actores solo conocen mailbox de otro actor que es FIFO
- Los actores definen jerarquía por lo que se pueden organizar y garantizar servicio
- Que puede hacer un actor
  - Crear otro actor
  - Enviar un mensaje
  - Saber atender un mensaje a la vez
  - Cambiar su estado interno solo a través de un mensaje

Live Code

- Crear un Sistema
- Codear un Actor
- Enviar un mensaje
- Cameo Pattern vs Future.sequence()

Akka Actor Model, lo bueno

- Mitiga los problemas de shared state
- Excelente para enviar a procesar cosas en paralelo y juntarlas para continuar
- Diseño de objetos de negocio y no de bajo nivel (locks, mutex, countdownlatch)
- Diseños de más alto nivel para procesamiento de streams
- Fault tolerant (supervisor strategies)
- escala vertical u horizontal (clustering nunca lo usamos... por ahora)

Akka Actor Model, lo malo

- Diseñar con criterio no siempre es fácil (no todo es un actor)
- Mailbox overflow, pero está mitigado con distintas estrategias (descartar lo primero, descartar lo último, etc).
- La documentación es extensa aun cuando el paradigma es simple
- No se puede seguir el código como en java saltando de método en método, pero se puede seguir siguiendo los mensajes
- Entender el dead letters
- Entender la jerarquía de supervisión para lograr sistemas que se auto sanan
- No está bueno para servicios grandes estilo monolíticos