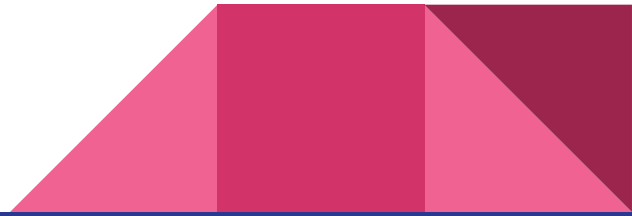
Threads

Una introducción pragmática

Temas

- Concurrencia != Paralelismo
- Threads en distintos lenguajes
- Global Interpreter Lock (GIL)

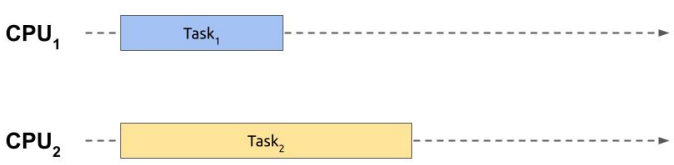


Repositorio de ejemplos

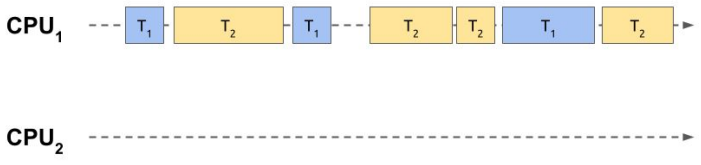
https://gitlab.com/unlp-so/teoria/threads-gil-ejemplos

concurrencia != paralelismo

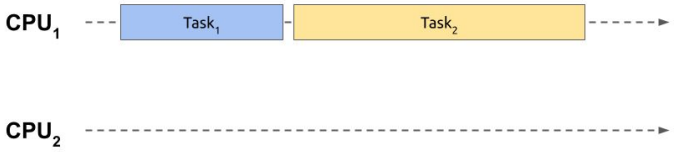
Esquemas de procesamiento



En paralelo



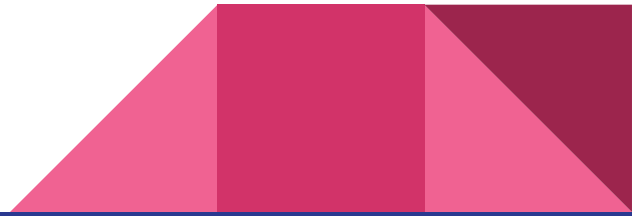
Context switch



Secuencial

Procesamiento paralelo

- Tareas independientes (o la misma tarea) se asignan a diferentes CPU simultáneamente.
- Requiere una arquitectura multi core (hoy día muy común).
- El procesamiento paralelo implica concurrencia



Procesamiento concurrente

- Implica correr tareas simultáneamente, pero no garantiza que se ejecuten en paralelo.
- Puede implementarse en arquitecturas de un único core usando context switch.
- Analogías:
  - Como en Gambito de Dama, cuando la protagonista juega simultáneamente partidos de ajedrez contra varios oponentes en una modalidad Round Robin: ella mueve y avanza al siguiente oponente.
  - Un desarrollador debe entregar un proyecto A y un proyecto B en 2 días, entonces alterna el primer día entre proyecto A y B, lo mismo el día 2.

Procesamiento secuencial

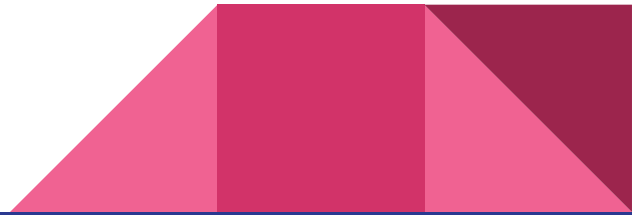
- Implica correr las tareas una tras otra, en orden.
- Analogías:
  - La partida de ajedrez ahora será una tras otra. El último jugador deberá esperar la protagonista termine cada juego, siendo más tolerante.
  - Un desarrollador trabajará en el proyecto A el día 1. El día 2 sólo en el proyecto B.

¿Tareas simultáneas?

- En términos de los SO, las tareas serán procesos o threads
- Los procesos se caracterizan por ser más presados:
  - La multiprogramación se logra con mútiples procesos. Procesos diferentes o hijos creados por un proceso padre.
  - No comparten nada, se copia el PCB. Son más seguros porque aíslan stacks.
  - Comunicación costosa: los mensajes deben serializarse. Impone una restricción en el volúmen de datos a compartir.
  - Los cambios de contexto son más costosos.
  - La finalización es más lenta.
- Los threads son más livianos que los procesos
  - La multiprogramación se logra por un proceso que creará threads
  - Comparten memoria, son más veloces pero no están aislados
  - Cambios de contexto más rápidos
  - La finalización es más rápida.

Algunas conclusiones

- Las soluciones concurrentes son más complejas que las secuenciales.
  - Deben considerar la sincronización y protección de datos.
- Ejecutar más tareas simultáneas que núcleos, resultará en la saturación de la CPU que se traduce en tiempos de respuesta altos.
  - *Esto aplica a tareas ligadas a la CPU y que hacen un uso intensivo de cómputo.*
- Un diseño concurrente puede ejecutar en paralelo o no.
- Cada context switch tiene un overhead que podría ralentizar una solución por sobre otra secuencial.
  - La misma solución en paralelo debe arrojar mejores resultados.



Tipos de tareas

Es importante distinguir **dos tipos de tareas:**

- **Ligadas a la CPU:** estas tareas consumen muchos ciclos de CPU por lo que sacarán mejor provecho si se las paraleliza.
- **Ligadas a la IO:** ejecutando simultáneamente tareas que esperan por operaciones de IO, reducen la latencia.

Threads

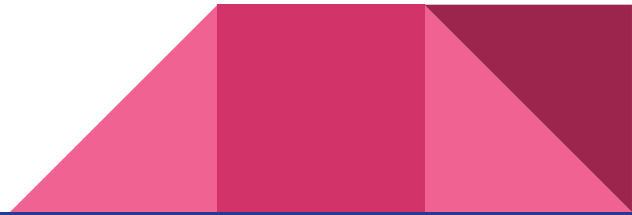
en distintos lenguajes

Ejemplos

En otras materias se estudian las consideraciones a la hora de desarrollar soluciones concurrentes. Estrategias de sincronización y protección de datos.

Queremos entonces analizar qué visualizamos desde la perspectiva del Sistema Operativo, como así también comprender cómo diferentes lenguajes implementan threads.

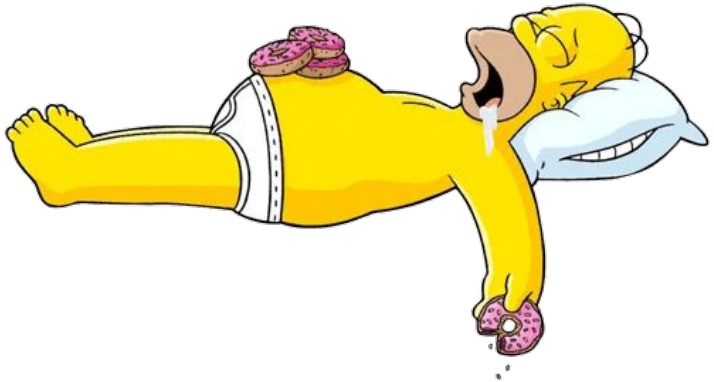
Trataremos de no apegarnos a un lenguaje y tomar aquellos que son elegidos por el mercado: **java, python, ruby,** php, node, dotnet, go, rust.



Un programa que crea threads haraganes

“Un programa que crea 100 threads que duermen todo el día”

Lo importante no es el ejemplo en sí, sino analizar los procesos con el comando ps de Linux



Otro ejemplo con necesidad de cómputo: **Fibonacci**

Haremos un programa que siempre calculará la la sucesión de **Fibonacci** hasta un número. Elegimos esta función porque está estrechamente ligada al uso de CPU. La idea es analizar el uso de la CPU **con y sin** threads en diferentes lenguajes.

```
int fib(int n) {
  return n < 2 ? n : fib(n - 2) + fib(n - 1);
}
```

Vemos métricas **sin threads**

- Consumo de la CPU:
  - Con el comando **time**
  - Usando el comando **ps**
  - Usando métricas de la CPU con Prometheus
- Probamos en el siguiente orden:
  - Java
  - Ruby
  - Python

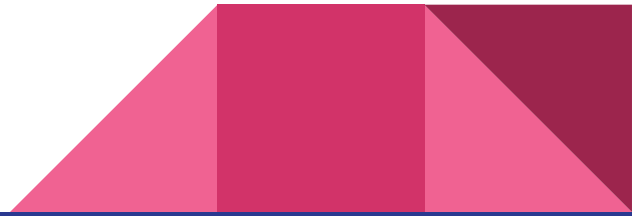
Vemos métricas **con threads**

- Consumo de la CPU:
  - Con el comando **time**
  - Usando el comando **ps**
  - Usando métricas de la CPU con Prometheus
- Probamos en el siguiente orden:
  - Java
  - Ruby
  - Python

¿Conclusiones?

- **Java utiliza todos los threads en paralelo:** cada thread corre en cada CPU.
- **Ruby y Python no, sólo usan un procesador:** no aprovechan el paralelismo.

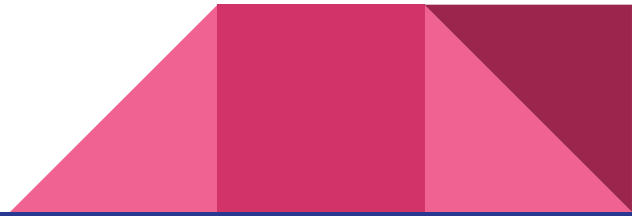
El problema que vemos con Ruby y Python es por el Global Interpreter Lock



Un ejemplo de IO con **fake delay**

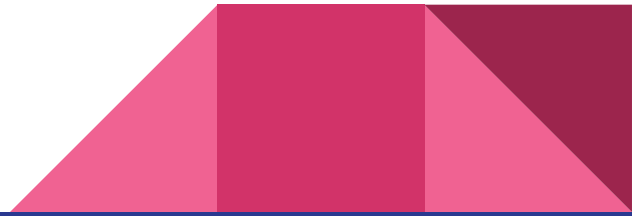
“Invocaremos a un endpoint http que se demora en responder X segundos”

https://httpbin.org/delay/10



Analicemos qué pasa ahora

- Consumo de la CPU secuencial vs threads.
- Analizarlo con el comando **time** y Prometheus.
- ¿Qué sucede con una versión que repita la versión de threads pero de forma secuencial? probar los ejemplos con **-sm** en vez de **-s.**
- **¿Conclusiones?**



Global Interpreter Lock (GIL)

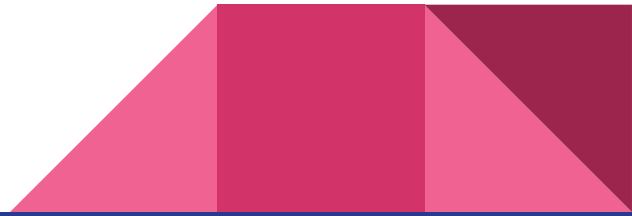
¿Que sería entonces GIL?

El **Global Interpreter Lock** es un mecanismo utilizado en **algunos intérpretes de lenguajes** para sincronizar la ejecución de threads de forma tal que **sólo un thread nativo (por proceso)** pueda ejecutar una operación crítica a la vez usando **mutex**.

Esto significa que si se ejecutan dos instancias del intérprete, serán dos procesos y cada uno tendrá su propio GIL.

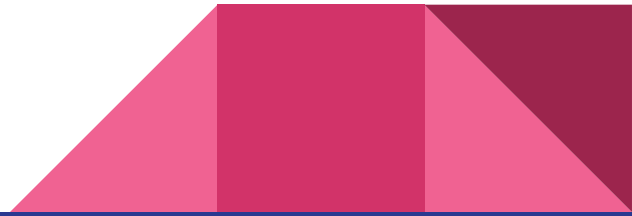
Por qué se mantiene GIL

- Para simplificar el uso de la concurrencia, que aún es eficaz cuando las tareas están ligadas a IO como vimos antes
- Para simplificar la integración de librerías C que no son thread safe.



Evitando GIL en Python y Ruby

- Existen versiones de ambos lenguajes basadas en la JVM o Dotnet, pero a veces no están tan actualizadas como las versiones de referencia que están desarrolladas en C:
  - **Versión de referencia Ruby:** CRuby o MRI.
  - **Versión de referencia de Python:** CPython.
- Cambiamos el interprete y veamos qué sucede:
  - JRuby
  - Python nogil



Analicemos la concurrencia con y sin GIL

Veamos cómo se comporta un programa que no es thread safe de forma impredecible:

```
counter = 0
def bad_done():
    global counter
    temp = counter
    # No es thread safe esta función
    temp = temp + 1
    counter = temp
```

Probamos qué sucede con ruby y python

- Probar con los intérpretes de referencia.
- Luego con los que soportan paralelismo de threads.
- **¿Conclusiones?**

Preguntas

Pensemos en los web servers

¿Cómo funcionan las aplicaciones que desarrollamos en lenguajes interpretados entonces?

¿Son eficientes?