Facundo Batista
@facundobatista

Gentil introducción al mundo Asincrónico

Temario

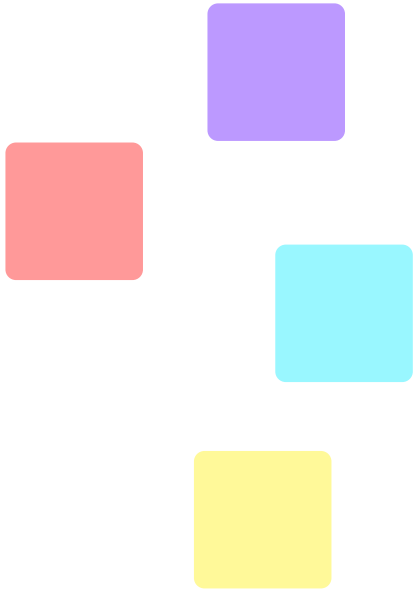
- Problemática a resolver
- Un approach clásico
- Pensemos algo alternativo
- Veamos cómo funciona

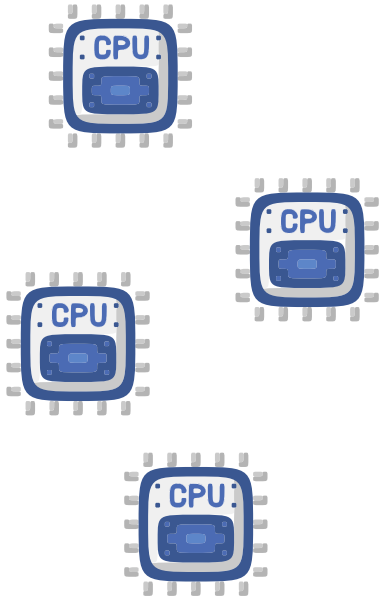
Slides y código: `bit.ly/gentil-async`

¿Qué queremos resolver?

Ejecución simultánea

- Tenemos varias tareas a ejecutar
- Tenemos varios procesadores





Repartimos

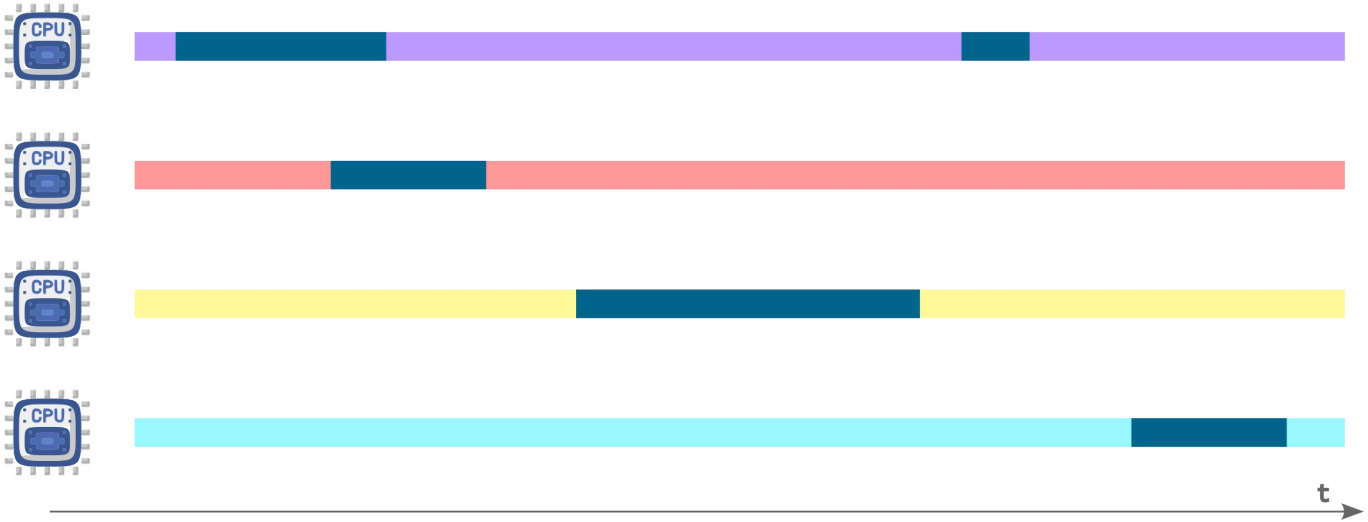
- Acomodamos cada tarea en un procesador
- ¡Paralelismo real!

¡Pero!

- Muchas veces tenemos más tareas que procesadores

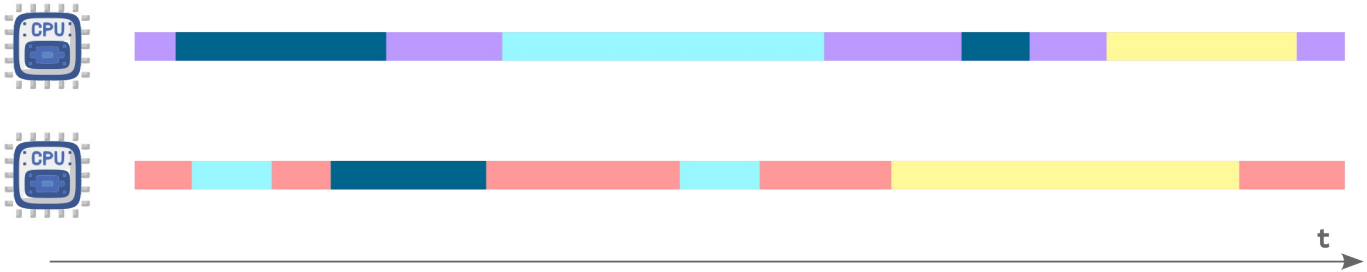
Modelo *preventivo*

- Los procesadores tienen una magia que juega con el OS
- Permite que el sistema operativo “robe” el procesador

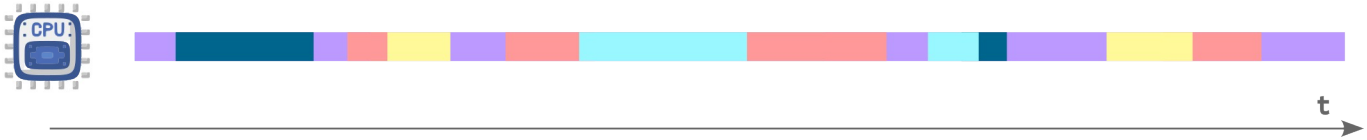


Modelo *preventivo*

- No importa la cantidad de procesadores disponibles

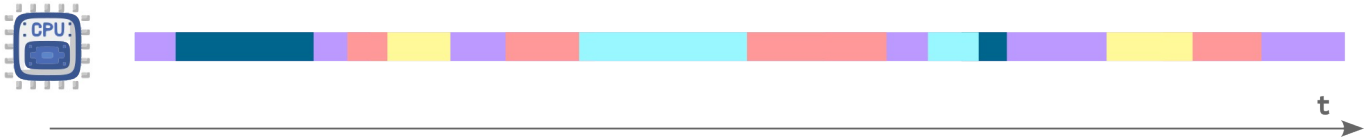


- Incluso teniendo uno solo



Ya no es paralelismo real

- A esto lo llamamos *concurrencia*



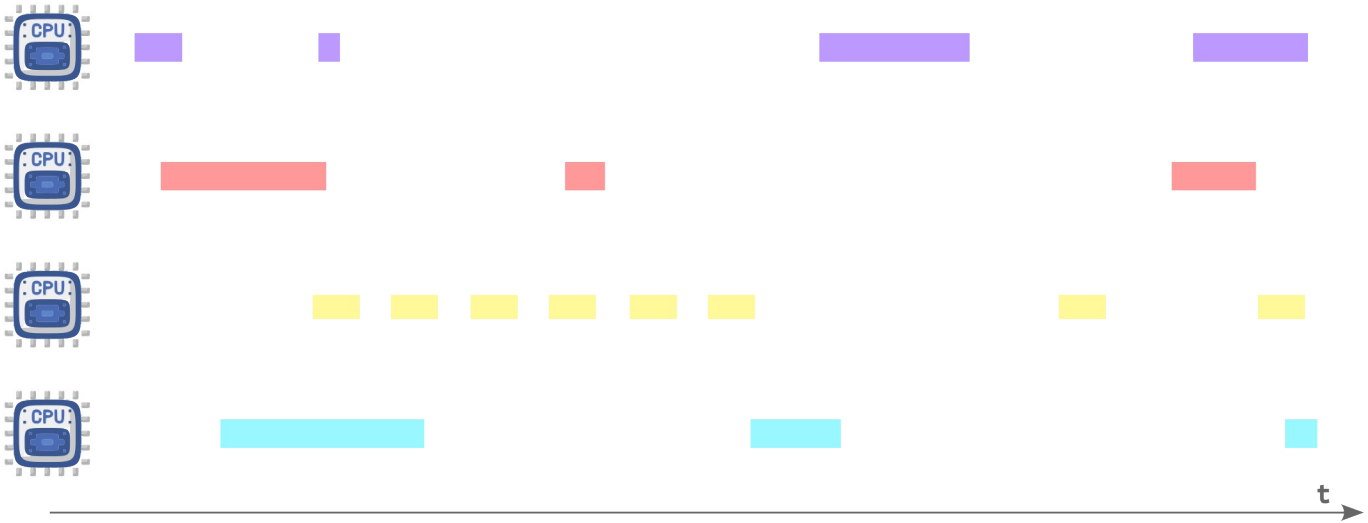
- Parece que las tareas se ejecutan al mismo tiempo
- Pero no :)

En realidad es más sencillo

- El ejemplo que vimos era *CPU bound*

En realidad es más sencillo

- El ejemplo que vimos era *CPU bound*
- En general las tareas son *IO bound* (entrada/salida)
  - A menos que estemos cruncheando números … y también hay IO ahí

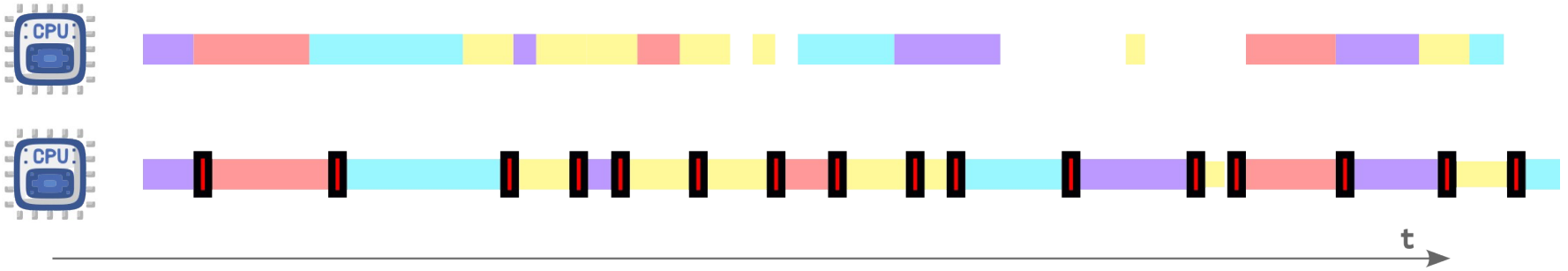


La concurrencia resulta más natural

- No se están forzando tareas todo el tiempo a salir
- Se intercalan más sencillo
- Pensemos con un sólo microprocesador
  - Es más sencillo de visualizar
  - Es lo mismo que con muchos microprocesadores y mayor cantidad de tareas

No es todo tan lindo

- En realidad el SO trabaja bastante para cambiar las tareas
- Ese costo se lo conceptualiza como un *cambio de contexto*
  - *context switch*
- Comparemos la situación ideal previa con este agregado



Bajando el costo

- Queremos minimizar el costo del cambio de contexto
  - Es MUY importante en algunas situaciones

- ¿Cómo hacemos?

Más barato que procesos

- Existen los hilos
  - *threads*
  - Comparten algunas estructuras, especialmente la memoria
  - Más eficientes de generar
  - El context switch es más barato (lo sigue haciendo el SO y CPU)

- Los hilos comparten memoria
  - ¡Buenísimo!
  - ¡Malísimo!

Threading y compartir memoria

- No tenemos control de cuando se ejecuta qué
  - característico del modelo preventivo
- Aparecen las condiciones de carrera (*race condition*)

```
class Counter:

    def __init__(self):
        self.val = 0

    def increment(self):
        self.val = self.val + 1
```

Perspectiva

Intentemos algo diferente

- ¿Qué queremos?
  - Efecto de concurrencia
  - Minimizar el context switch
- Otros efectos
  - Compartir memoria está bueno
  - Perder el control de qué se ejecuta no está bueno

Vayamos a lo simple

- No podemos usar… ¿funciones?
  - Es lo más sencillo que se nos ocurre
- En vez de hilos o procesos
  - Todo en el *mismo proceso*
  - Que ya no lo maneje el SO
- Pero que se ejecuten concurrentemente
  - Que es la idea base

¿Cómo sería el modelo?

- *Algo* llama a cada función
- La función se ejecuta, hace lo suyo, termina
- Indica si necesita un recurso o un evento de E/S
- Al terminar, devuelve el control a ese a*lgo*
- Ese *algo* también administra las E/S a nivel del proceso completo
- Todas las funciones y ese *algo* corren en el mismo proceso

Ese *algo*: el bucle de eventos

- Algo que está girando todo el tiempo
  - escuchando los eventos que se suceden
  - llamando a nuestro código
  - esperando que devolvamos rápido el control

```
while not self.stop:
    events = self.get_events()
    for event in events:
        self.dispatch(event)
```

- Se llama *event loop* o *reactor*

Marcando el paso

Repasemos lo que hace

- Escucha los eventos que se suceden
  - El sistema operativo sigue manejando la E/S del proceso
- Llama a nuestro código
  - Decide qué funciones puede volver a ejecutar
  - El context switch “entre funciones” es muy barato
- Espera que devolvamos rápido el control
  - No puede interrumpir las funciones
  - Modelo *cooperativo*

Modelo *cooperativo*

- Escala mucho mejor
- Acceso fácil a objetos compartidos
- Es determinístico

- Hay que pensar qué bloquea y qué no
- El código es más difícil de escribir

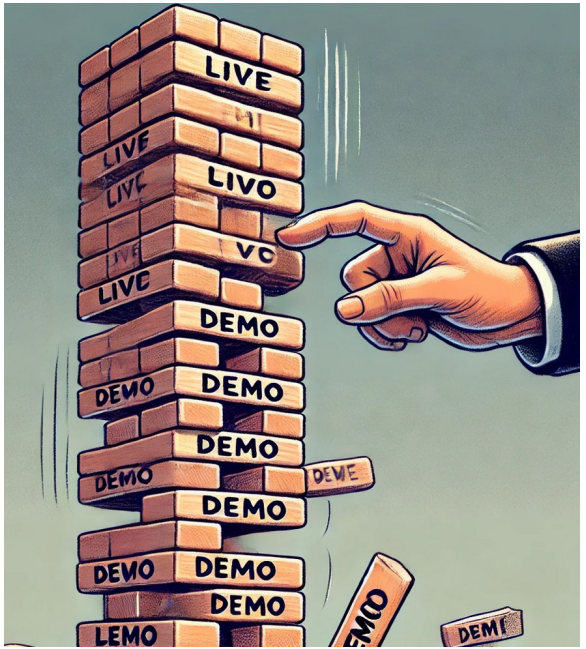
Volviendo con el recurso obtenido

- ¿Cómo funciona que el *event loop* nos llame cuando tenga data?
- Le tenemos que dar un *callback*
  - Algo llamable (ej. una función)
  - Será ejecutada en el futuro (“a la vuelta de”)
- Es sencillo de implementar en un lenguaje
  - Porque no necesita soporte especial del mismo

Ejemplo 1

DEMO

- Veamos el sistema asincrónico por excelencia: una interfaz gráfica
- Fuertemente orientada a eventos
  - Principalmente de le humane, que siempre tarda mucho
- Veamos...
  - El setup necesario, porque hay que poner un event loop a trabajar
  - Un callback



Ejemplo 2

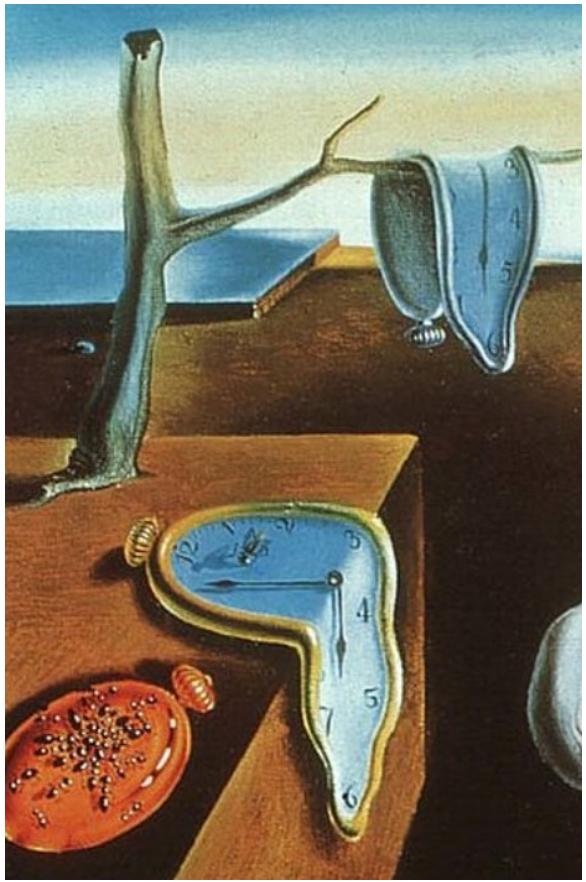
DEMO

- Construimos sobre el anterior
- Vemos un evento que no viene “de afuera”

Ejemplo 3

DEMO

- El código tiene que ser *cooperativo*
- Si bloqueamos estamos en problemas
  - O si tardamos mucho



Pero queremos más

- En las GUIs los callbacks son suficientes
  - Pero siempre son funciones que empiezan y terminan
- En otros sistemas queremos nosotres poder ejecutar funciones asincrónicas
- Hola *corrutinas*

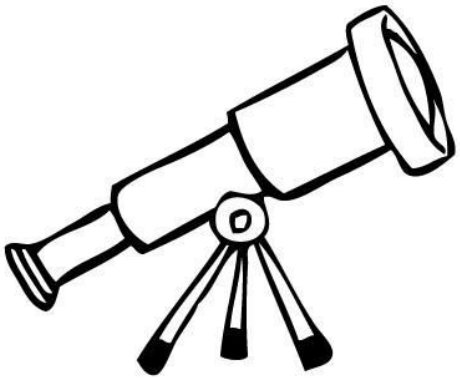
¿Qué son las corrutinas?

- Nuestras “funciones asincrónicas”
- Las podemos a poner a disposición del event loop
- Podemos esperarlas a que terminen
  - Pero no hace falta que terminen para ir y volver
- La corrutina puede esperar que otras corrutinas terminen

Veamos algunas corrutinas en acción

DEMO

- Lo mínimo
- Más corrutinas
- Podemos mezclar funciones normales
- Tareas
- Con callbacks!



Dejamos afuera

- Stream de bytes, transportes y protocolos
- Manejar situaciones bloqueantes
- Colas y otras sincronizaciones
- Comunicación entre procesos
- Señales del sistema operativo
- Etc...

Conclusiones

- Lo Asincrónico es bastante poderoso
  - Tiene sus ventajas
  - También sus desventajas
  - No hay una solución mágica
- Puede ser complicado
  - No estamos acostumbrades a pensarlo
- Hay que ponerse a jugar
  - Diviértanse :)


¿Preguntas?
Facundo Batista
@facundobatista
¡Muchas gracias!