

# Processos e threads em Python

---

# Modelos de concorrência em Python

- Processos: módulos multiprocessing, futures
- Threads: módulos threading, futures
- Corrotinas: módulo asyncio
  - frameworks modernos como FastAPI

Vamos explorar os  
três modelos na  
teoria e na prática

Hoje veremos  
threads e  
processos

# Demonstração: imagens da Wikipédia

localhost:8888/notebooks/wikipics/wikipics-demo.ipynb

File Edit View Run Kernel Settings Help Trusted


JupyterLab Python 3 (ipykernel)

```
[11]: %%time
      from concurrent import futures

      num_images = 10
      gallery = Gallery(num_images)
      gallery.display()

      urls = get_sample_urls(5_000_000, num_images)
      with futures.ThreadPoolExecutor() as pool:
          img_records = pool.map(fetch, urls)
          for i, img_rec in enumerate(img_records):
              gallery.update(i, img_rec.pixels)

      print(f'TOTAL BYTES: {gallery.size:_}')
```



( 1 )	5_086_950 bytes	European Parliament Strasbourg Hemicycle -- Diliff.jpg
( 2 )	5_073_179 bytes	William Cranch.jpg
( 3 )	4_943_441 bytes	Lee Bollinger -- Daniella Zalcmann_less_noise.jpg
( 4 )	5_037_414 bytes	Sidney Hall -- Urania%27s Mirror -- Draco and Ursa Minor.jpg
( 5 )	5_007_245 bytes	The Day the Earth Smiled -- PIA17172.jpg
( 6 )	5_063_007 bytes	Elizabeth I Steven Van Der Meulen.jpg

<https://bit.ly/wikipics2024>

# Análise dos resultados

## Download sequencial

```
TOTAL BYTES: 19_955_565  
CPU times: user 732 ms, sys: 262 ms, total: 994 ms  
Wall time: 16.4 s
```

## Download com threads

```
TOTAL BYTES: 19_950_522  
CPU times: user 9.27 s, sys: 251 ms, total: 9.52 s  
Wall time: 2.67 s
```

# Entendendo as medidas de CPU time

```
CPU times: user 732 ms, sys: 262 ms, total: 994 ms  
Wall time: 16.4 s
```

- **user**: tempo executando código user-mode (interpretador, seu código, bibliotecas)
- **sys**: tempo executando código kernel-mode (tudo que envolve o hardware: armazenagem, rede, tela, etc.)
- **total**: user + sys
- **wall time**: tempo transcorrido no "relógio da parede", também conhecido como "real time"

# Análise: download sequencial

```
TOTAL BYTES: 19_955_565  
CPU times: user 732 ms, sys: 262 ms, total: 994 ms  
Wall time: 16.4 s
```

- Baixou  $\approx 20$  MB em 16.4 s:  $\approx 1.2$  MB/s
- A CPU trabalhou por 0.994 s (6% do total)
- 94% do tempo o SO executou outros processos enquanto aguardava respostas da rede
- Isso é uma tarefa I/O bound (limitada por E/S)

# Análise: download com threads

```
TOTAL BYTES: 19_950_522  
CPU times: user 9.27 s, sys: 251 ms, total: 9.52 s  
Wall time: 2.67 s
```

- Baixou  $\approx 20$  MB em 2.67 s:  $\approx 7.5$  MB/s
  - 6.1 vezes mais rápido que sequencial (16.4 s)
- A CPU trabalhou por 9.52 s (356% do total)
  - bibliotecas usaram vários núcleos da CPU
- Melhor uso da CPU em tarefa I/O bound

# Aprendizados da demonstração

---

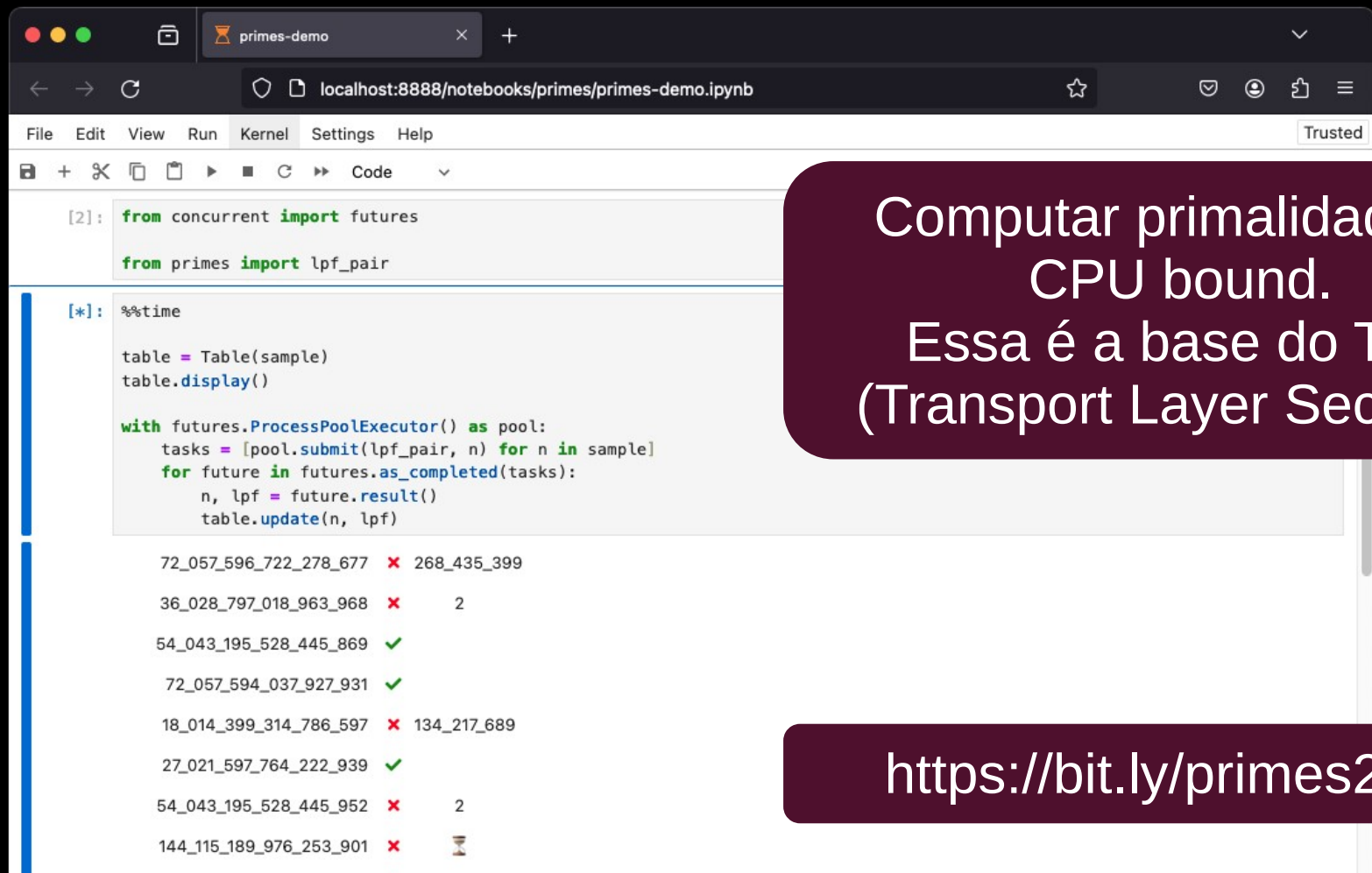
- Baixar imagem é tarefa I/O bound
  - assim como acessar arquivos, bancos de dados, APIs remotas, receber/enviar pacotes TCP/IP etc.
- Threads funcionam bem para I/O concorrente
  - Apesar da famosa GIL (Global Interpreter Lock)
    - A GIL limita todo bytecode Python a uma thread
    - Uma thread só usa um núcleo de CPU
    - O SO e certas bibliotecas escritas em C, C++, Rust, Fortran etc. não são limitadas pela GIL



# Porque não usar threads sempre

- Processos, threads, e corrotinas têm características muito diferentes
  - Utilização de recursos: núcleos de CPU, consumo de memória, custos de inicialização
- Threads e corrotinas são eficientes para processos I/O bound
- O que acontece com CPU bound?

# Demonstração: números primos



The screenshot shows a Jupyter Notebook titled 'primes-demo' running on 'localhost:8888/notebooks/primes/primes-demo.ipynb'. The notebook has a 'Trusted' status. The code is as follows:

```
[2]: from concurrent import futures
     from primes import lpf_pair

[*]: %%time

table = Table(sample)
table.display()

with futures.ProcessPoolExecutor() as pool:
    tasks = [pool.submit(lpf_pair, n) for n in sample]
    for future in futures.as_completed(tasks):
        n, lpf = future.result()
        table.update(n, lpf)
```

The output of the notebook shows a table with two columns: a range of numbers and a result. The results are marked with red 'X' for failure, green checkmarks for success, and a small icon for a pending state.

72_057_596_722_278_677	✗	268_435_399
36_028_797_018_963_968	✗	2
54_043_195_528_445_869	✓	
72_057_594_037_927_931	✓	
18_014_399_314_786_597	✗	134_217_689
27_021_597_764_222_939	✓	
54_043_195_528_445_952	✗	2
144_115_189_976_253_901	✗	🕒

Computar primalidade é  
CPU bound.  
Essa é a base do TLS  
(Transport Layer Security)

<https://bit.ly/primes2024>

# Aprendizados da demonstração

---

- Por causa da GIL, threads não servem para fazer processamento intensivo em CPU no código Python
  - Mas bibliotecas como Numpy são escritas em linguagens compiladas não limitadas pela GIL
    - mas depende da implementação da biblioteca

# Resumo: threads x processos

	<b>threads</b>	<b>processos</b>
<b>uso de CPU em código Python</b>	limitado a 1 núcleo da CPU para todas as threads executando bytecode Python	cada processo pode usar um núcleo da CPU
<b>uso de memória</b>	memória alocada para um processo + cerca de 4MB por thread no mínimo	uso maior de memória: memória isolada para cada processo + threads
<b>comunicação entre unidades de execução</b>	todas as threads podem acessar os mesmos objetos na memória	comunicação entre processos é mais lenta e complicada
<b>custo de inicialização</b>	criar novo processo é caro; criar threads, nem tanto	custo maior, pois é multiplicado pelo número de processos