[320] Parallelism

Department of Computer Sciences
University of Wisconsin-Madison

Parallelism: doing multiple things at once

Other Terms Today: process, thread, instruction pointer,
state (running, ready, blocked), CPU, GPU, core

Outline:

- Mental Model
- Two problems
- Parallelism: Thread, Process, GPU

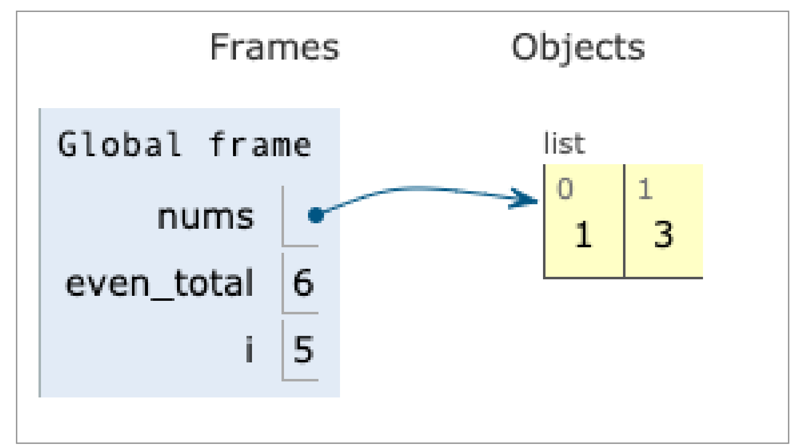
Mental Model: Tasks and Cores

One Python Program Running

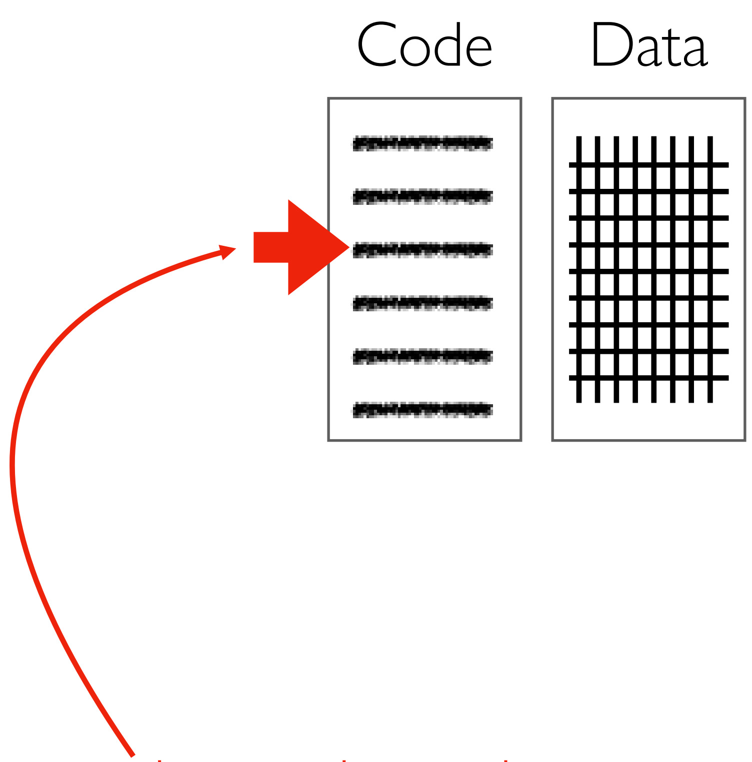
Code

```
nums = []
even_total = 0
for i in range(10):
    if i % 2 == 0:
        even_total += i
    else:
        nums.append(i)
print(i)
```

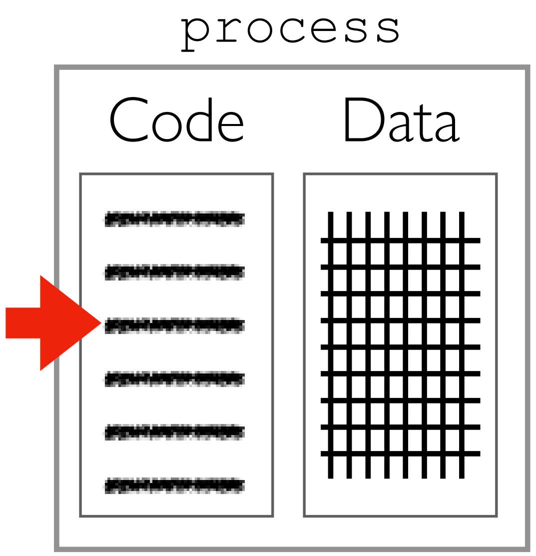
Data



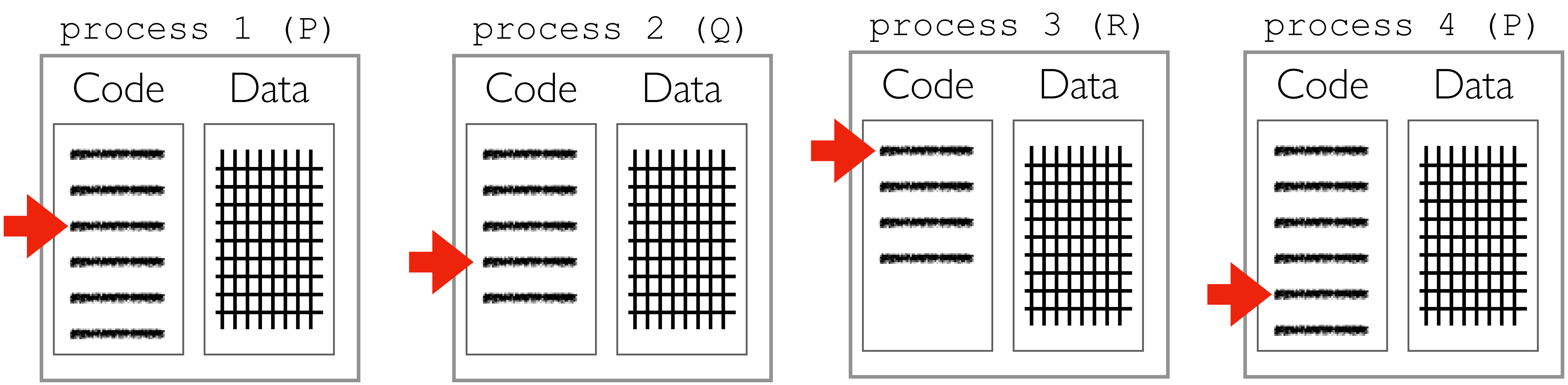
what is currently being done

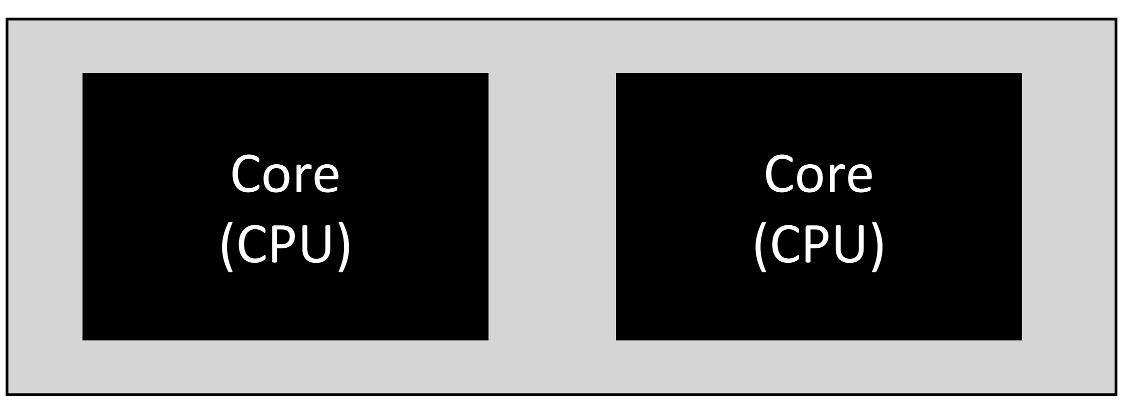


instruction pointer
(also called "program counter")

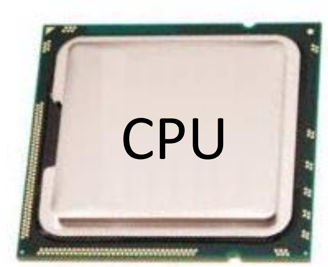


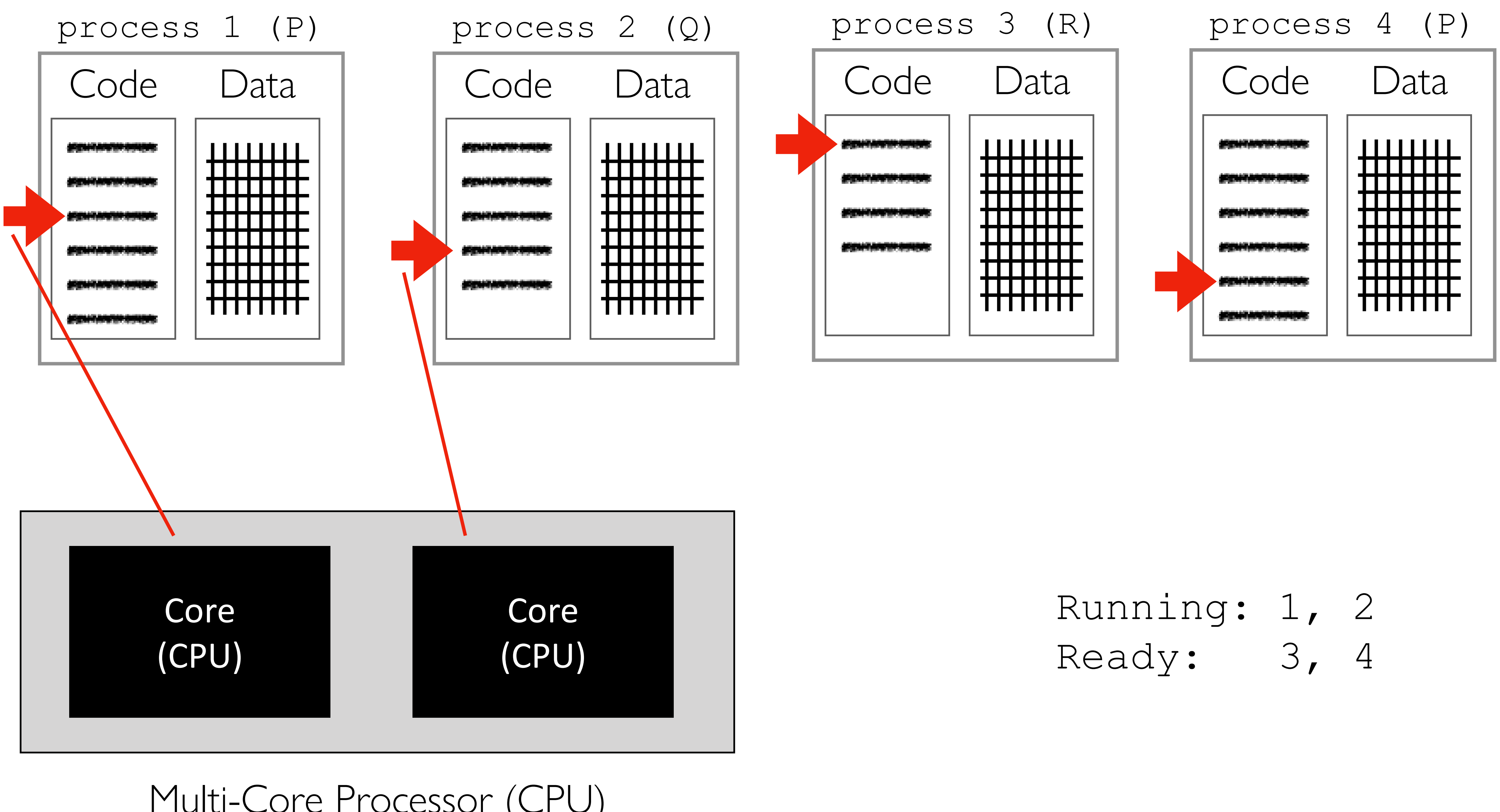
instruction pointer belongs to a ***thread*** within the process

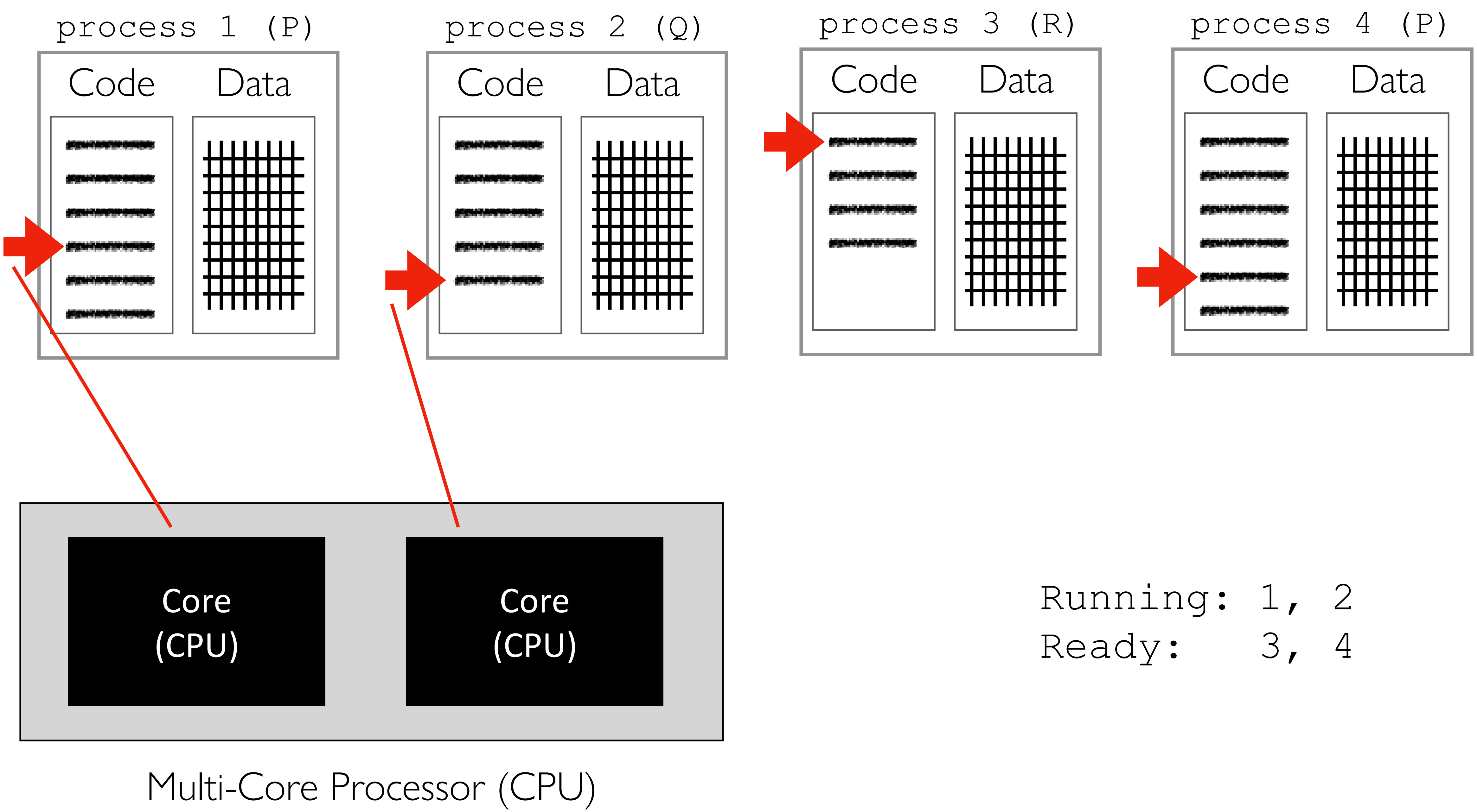




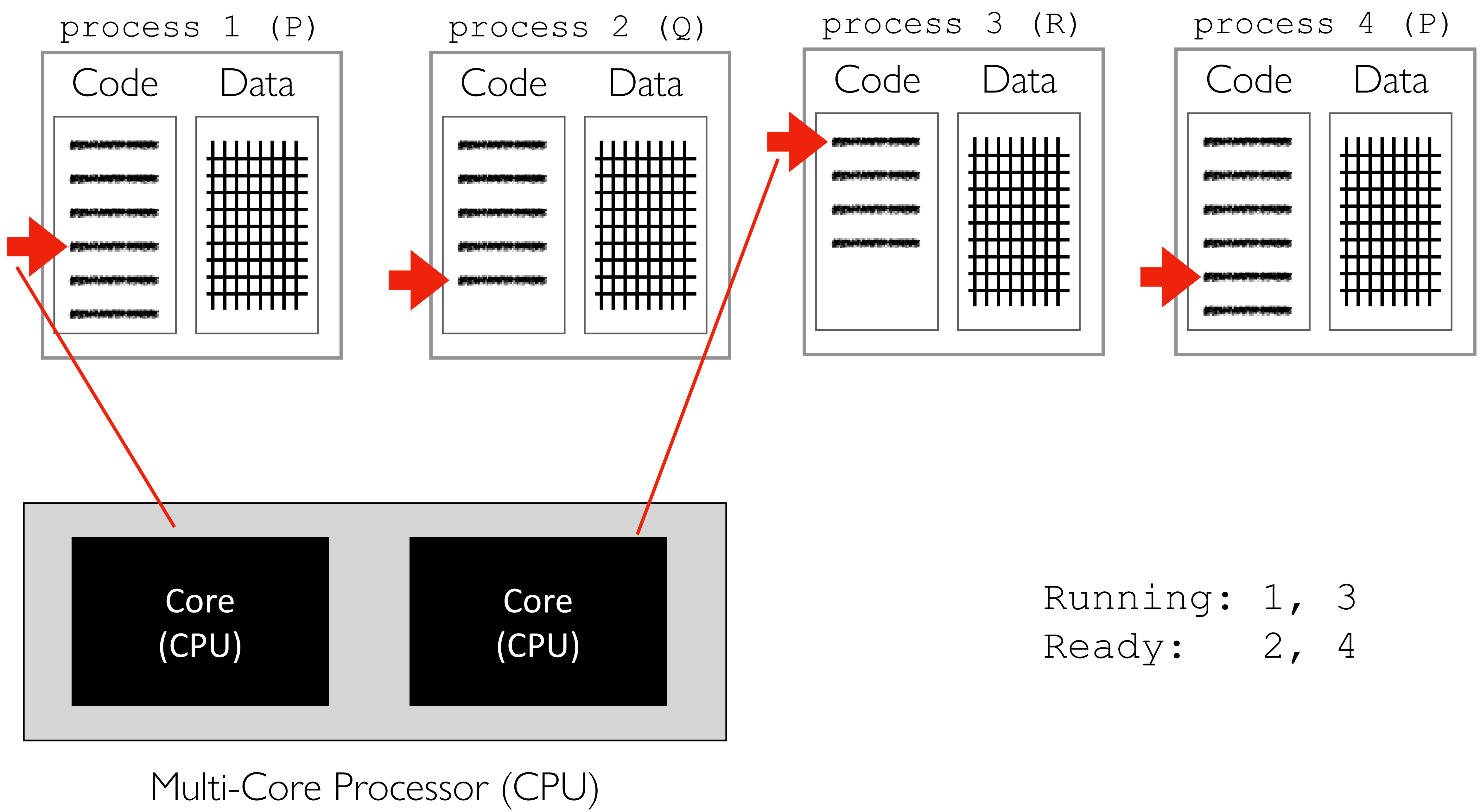
Multi-Core Processor (CPU)

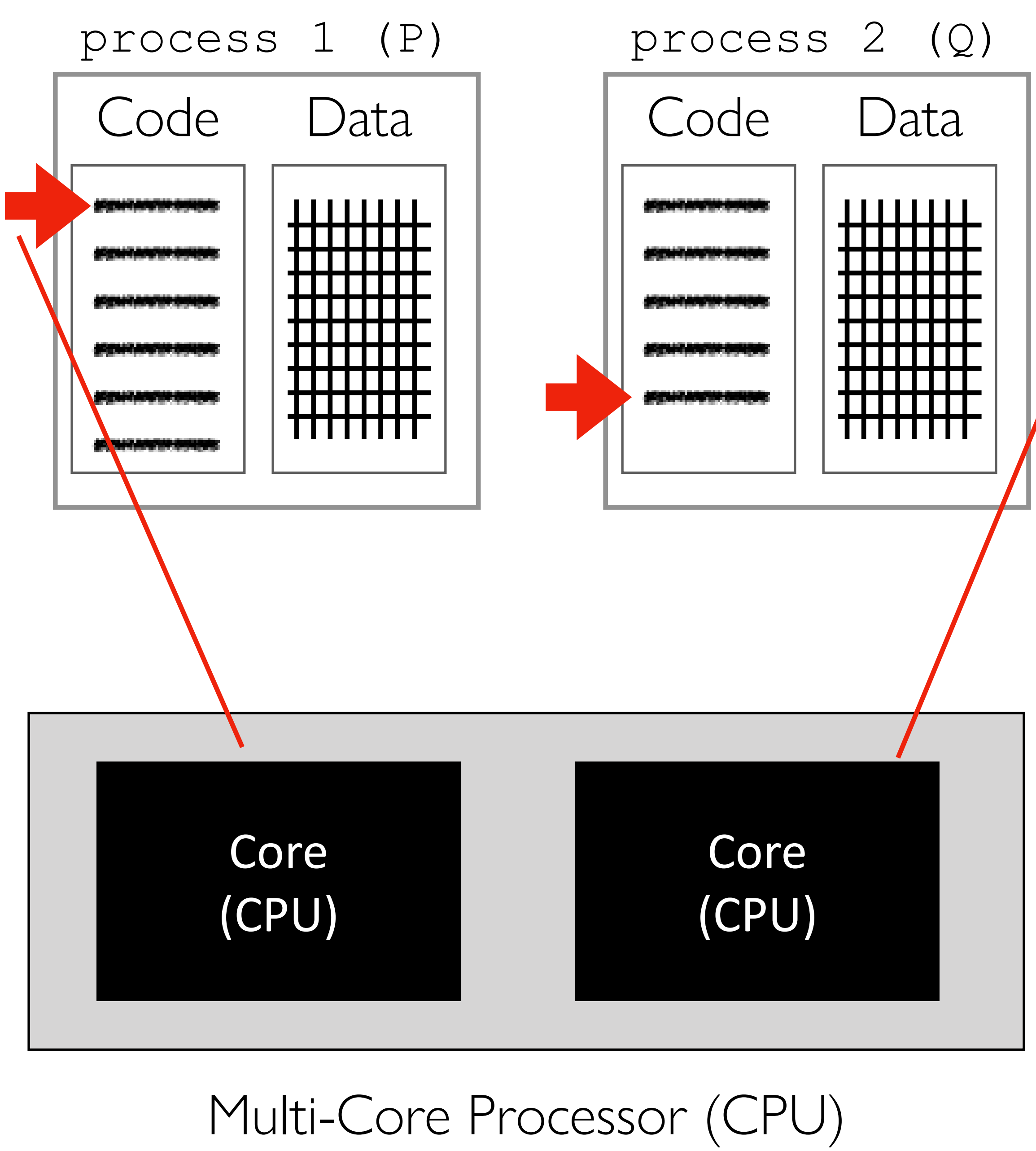



process 1 (P)
Code
Data
process 2 (Q)
Code
Data
process 3 (R)
Code
Data
process 4 (P)
Code
Data
Core
(CPU)
Core
(CPU)
Multi-Core Processor (CPU)
Running: 1, 2
Ready: 3, 4




```
Running: 1, 2
Ready:   3, 4
```


process 1 (P)
Code
Data
process 2 (Q)
Code
Data
process 3 (R)
Code
Data
process 4 (P)
Code
Data
Core (CPU)
Core (CPU)
Multi-Core Processor (CPU)
Running: 1, 3
Ready: 2, 4

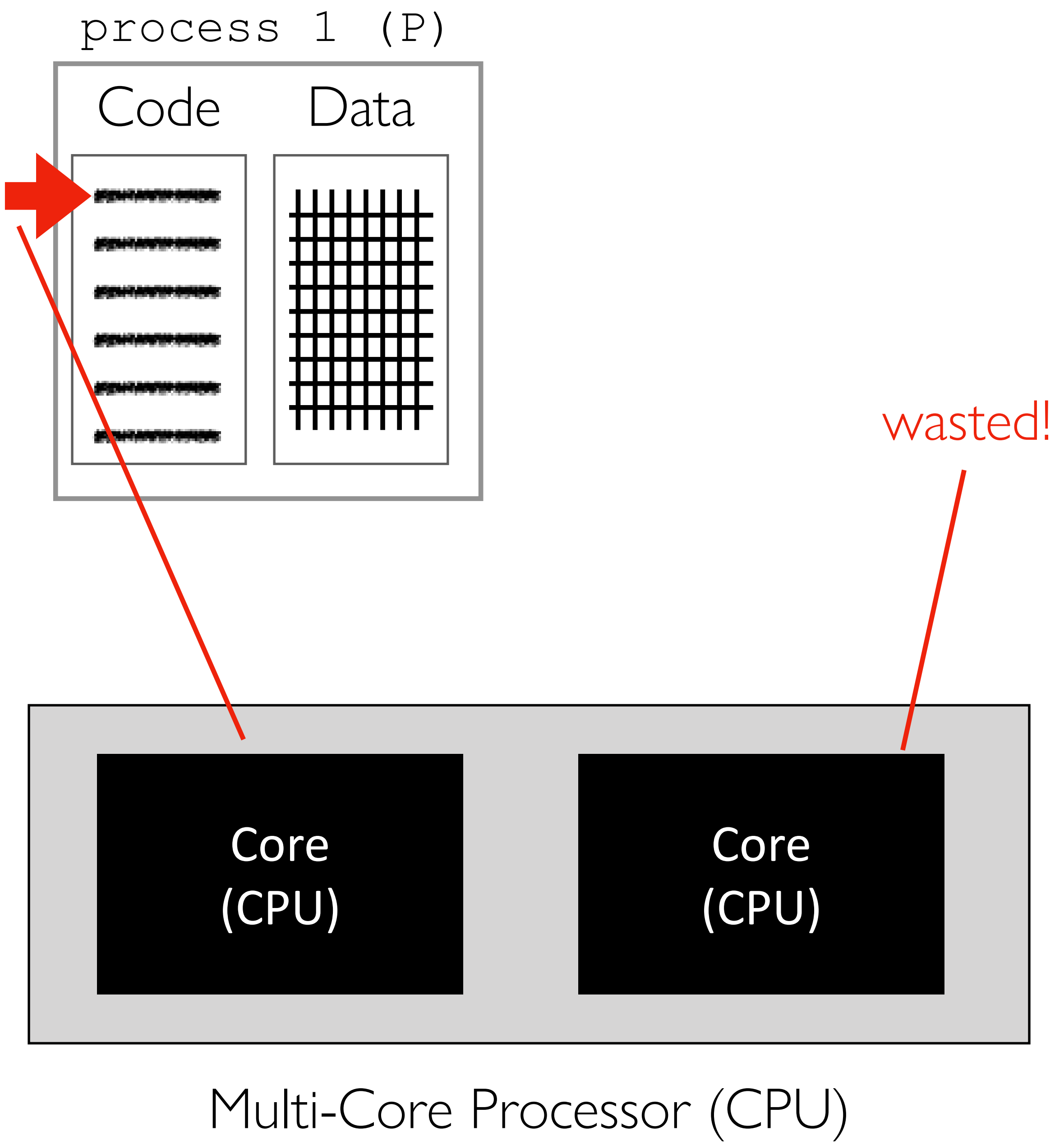


the more cores we have, the more tasks we can run simultaneously

```
Running: 1, 3
Ready:   2, 4
```

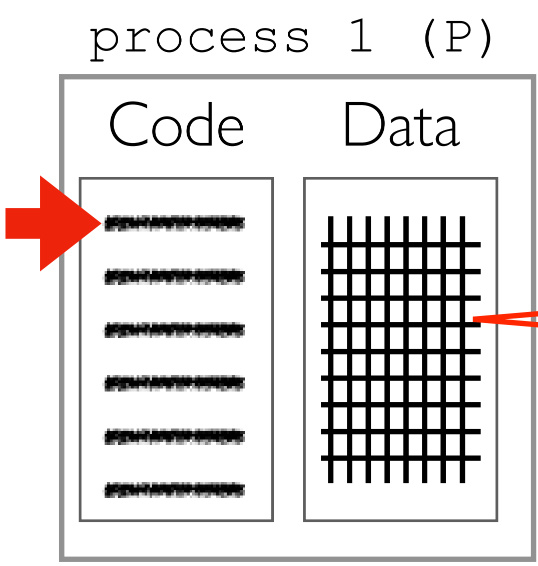
Wasted Compute Resources: Two Problems

Problem 1: not enough distinct tasks to utilize all cores




```
Running: 1
Ready:
```

Problem 2: some operations requires waiting (task is "blocked")

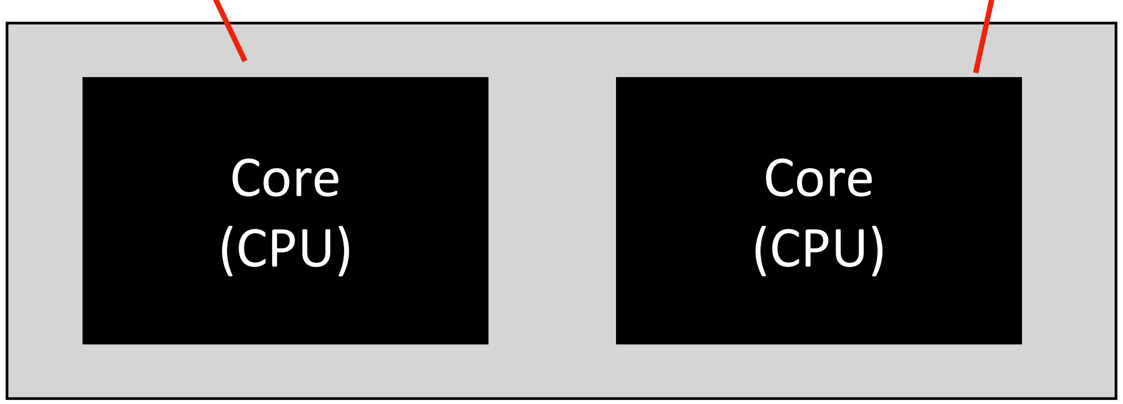


operation may require us to wait on an external resource

- f.read()
- requests.get(URL)
- time.sleep(SECONDS)
- x = input()

wasted!

wasted!



Multi-Core Processor (CPU)

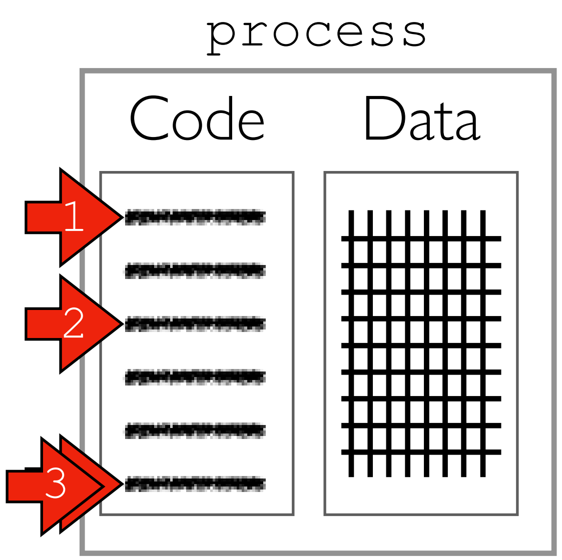
```
Running:
Ready:
Blocked: 1
```

Solution: Parallelism

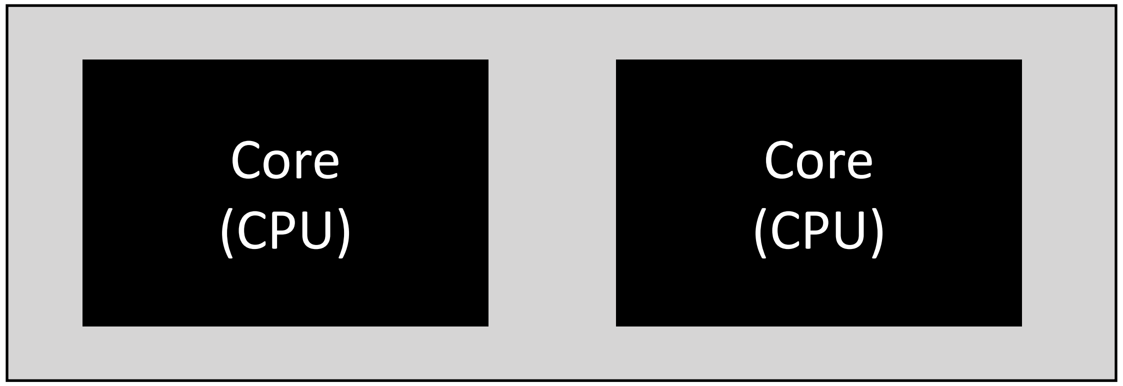
1. thread-level parallelism — very complicated, not covered in detail
2. process-level parallelism
3. GPU parallelism

covered in CS 320

(1) Thread-level Parallelism

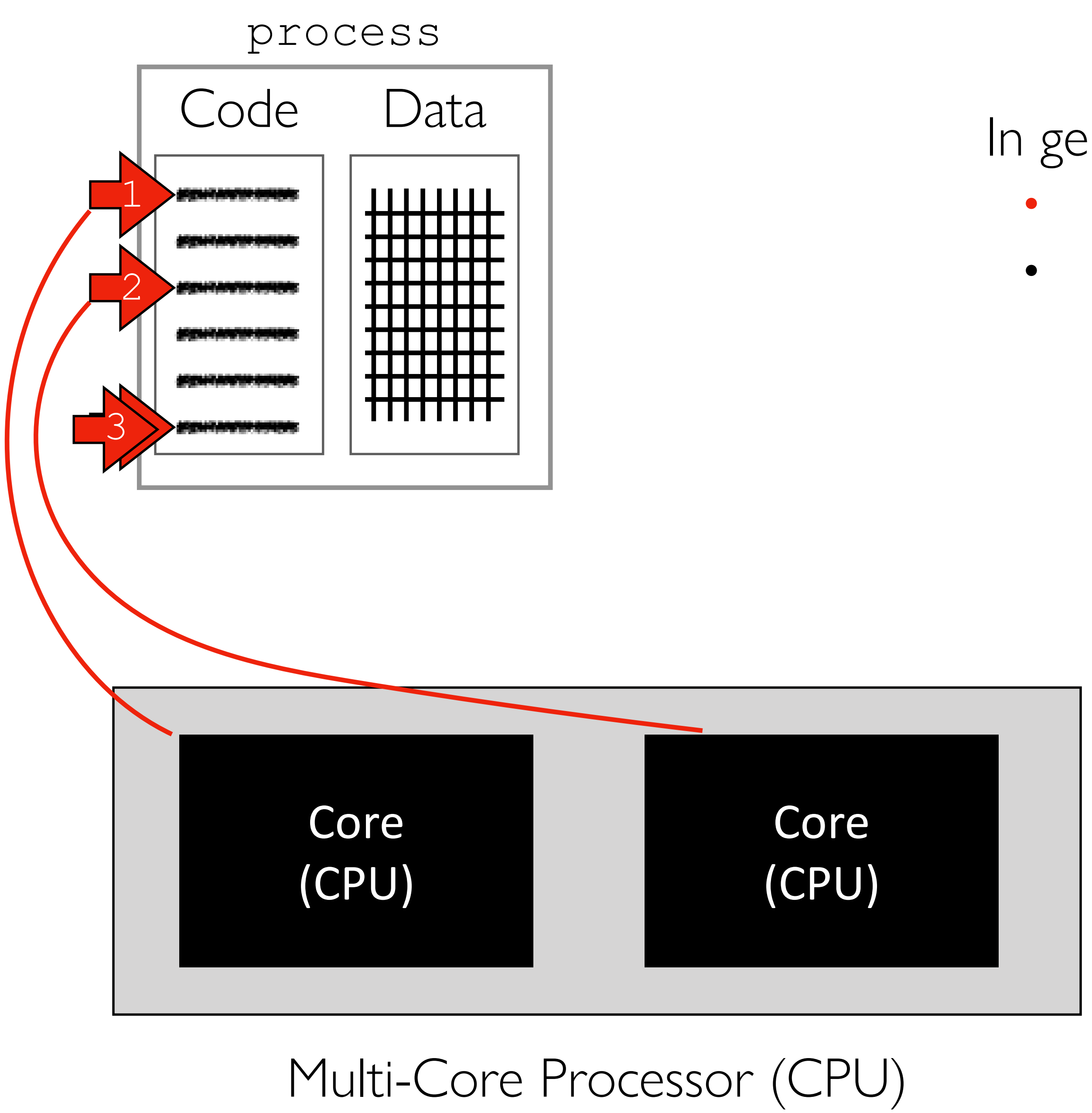


Threads give us multiple instruction pointers in a process, allowing us to execute multiple parts of the code, at the same time!



Multi-Core Processor (CPU)

(1) Thread-level Parallelism

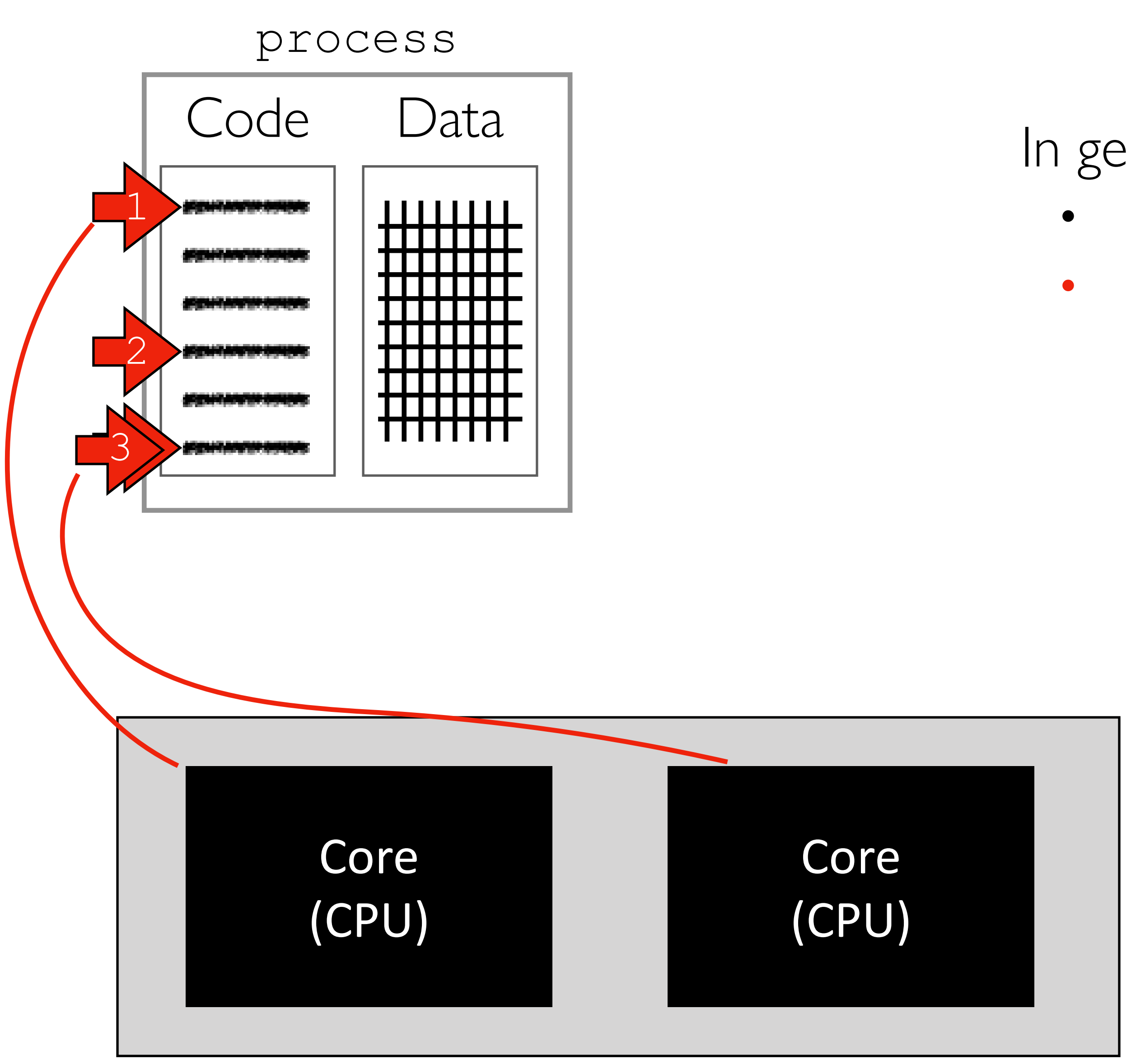


In general, threads help:

- use multiple cores
- do useful work when threads are blocking

```
Running: 1, 2
Ready:   3, 4
Blocked:
```

(1) Thread-level Parallelism



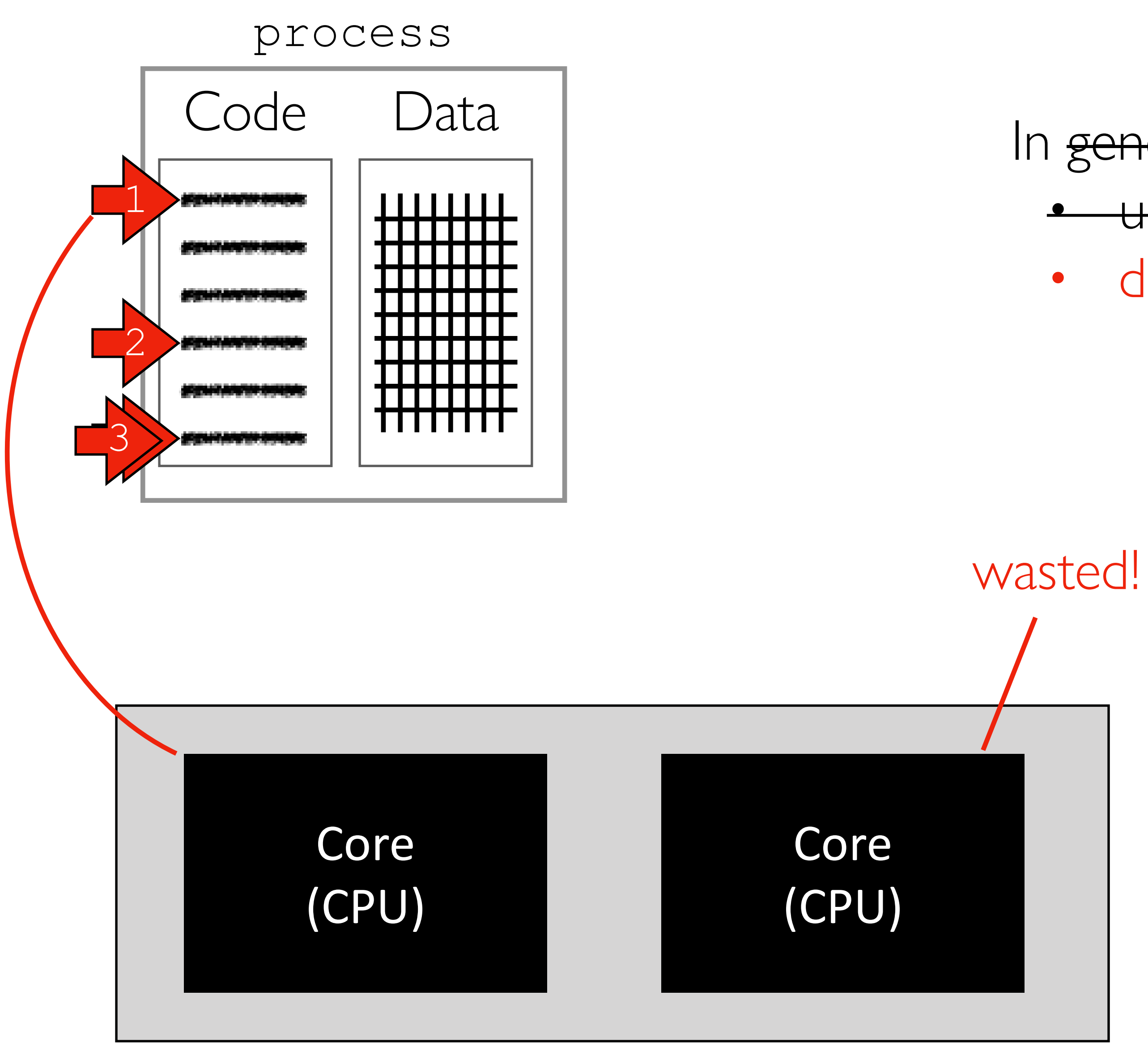
Multi-Core Processor (CPU)

In general, threads help:

- use multiple cores
- do useful work when threads are blocking

```
Running: 1, 3
Ready:   4
Blocked: 2
```

(1) Thread-level Parallelism



Multi-Core Processor (CPU)

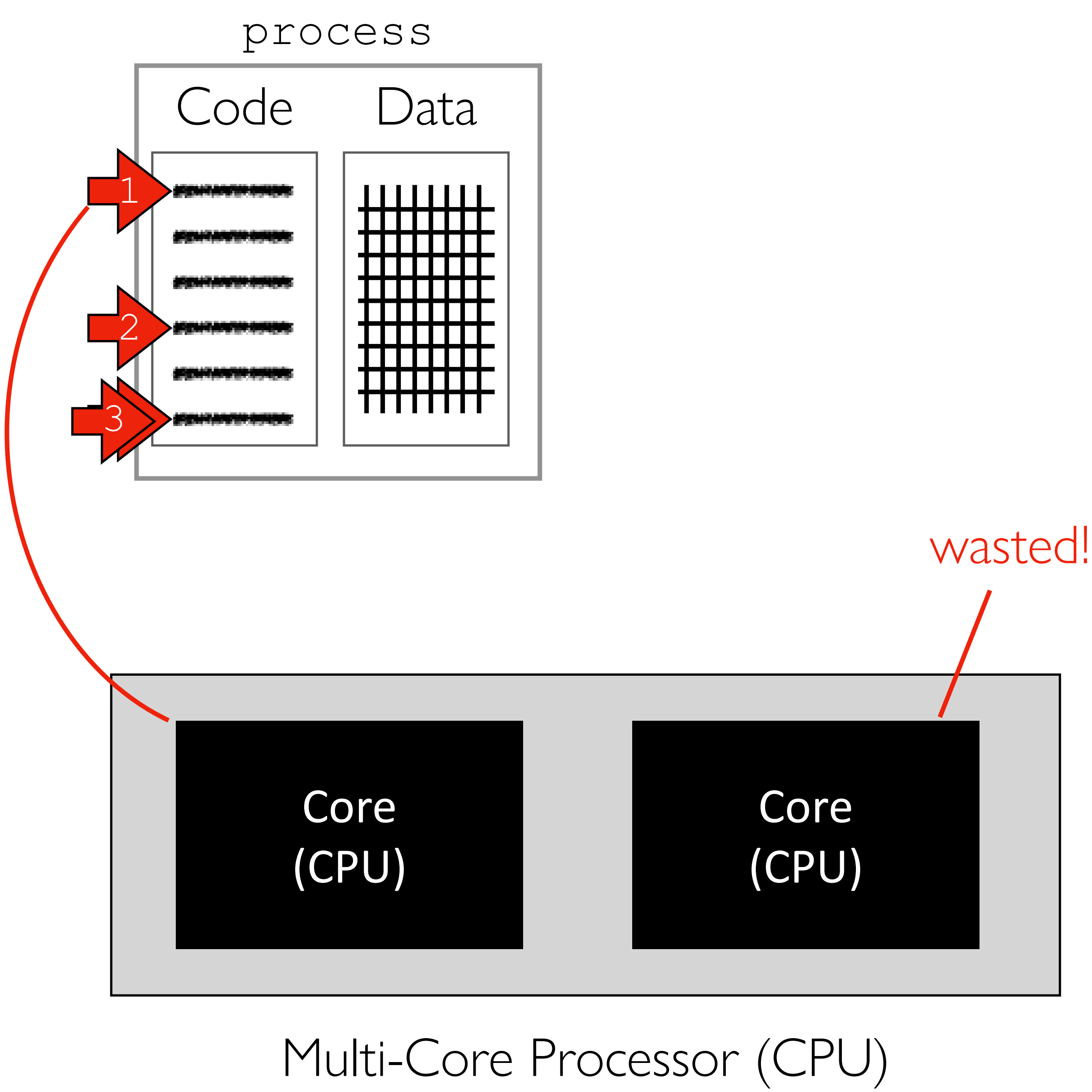
In ~~general~~ Python, threads help:

- ~~use multiple cores~~
- do useful work when threads are blocking

```
Running: 1
Ready:   3, 4
Blocked: 2
```

(1) Thread-level Parallelism

recommendation: don't use threads unless you learn a LOT more about multi-threading than covered in CS 320



Example: two countdown threads

```
import time
from threading import Thread

def f(name, n):
    for i in range(n):
        print(name, n-i)
        time.sleep(1)

# f("A", 3)
# f("B", 5)

t1 = Thread(target=f, args=("A", 3))
t2 = Thread(target=f, args=("B", 5))
t1.start()
t2.start()
t1.join()
t2.join()
```

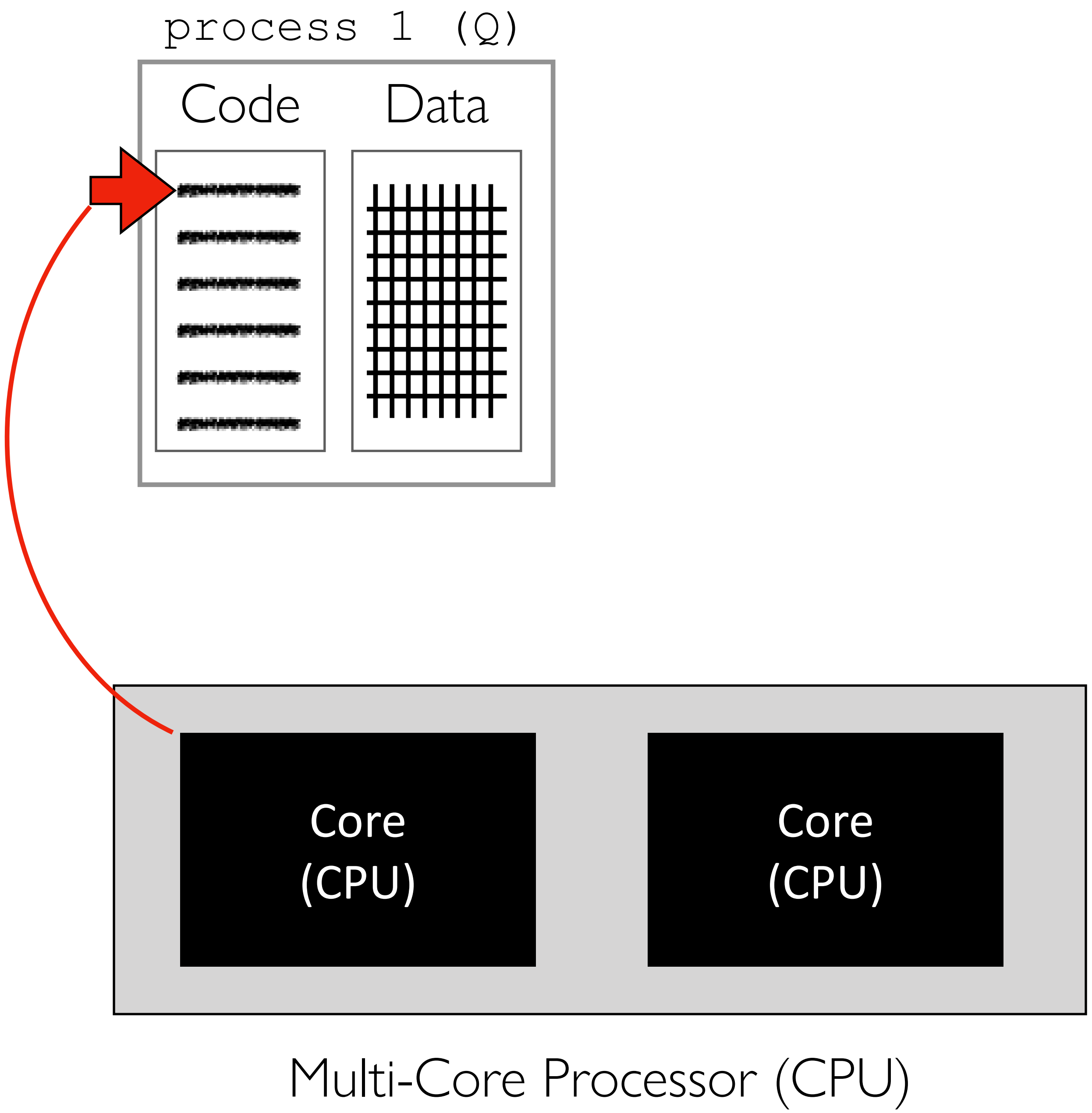
```
Running: 1
Ready:   3, 4
Blocked: 2
```

Solution: Parallelism

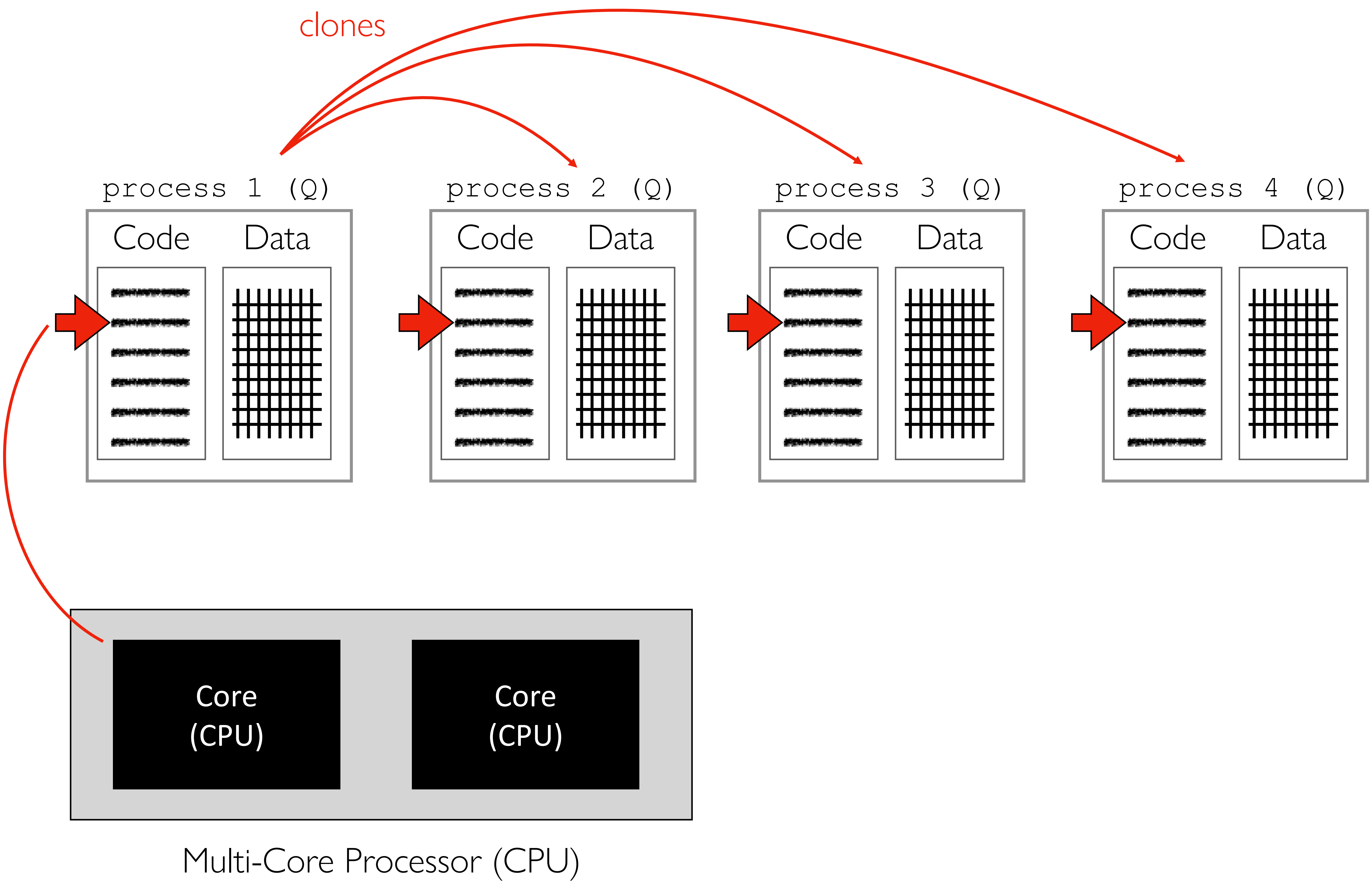
1. thread-level parallelism — very complicated, not covered in detail
2. process-level parallelism
3. GPU parallelism

covered in CS 320 (items 2 and 3)

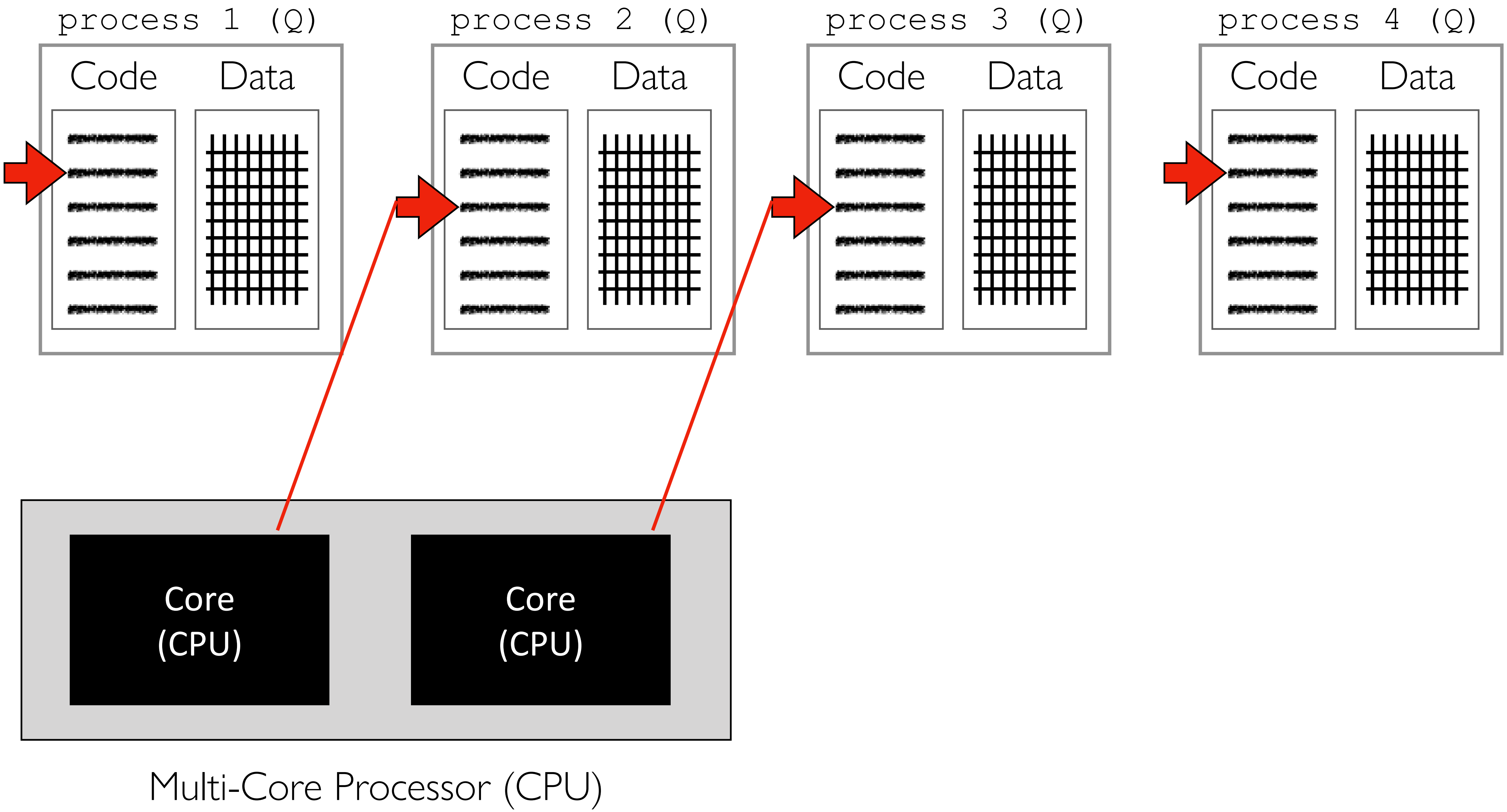
(2) Process-level Parallelism



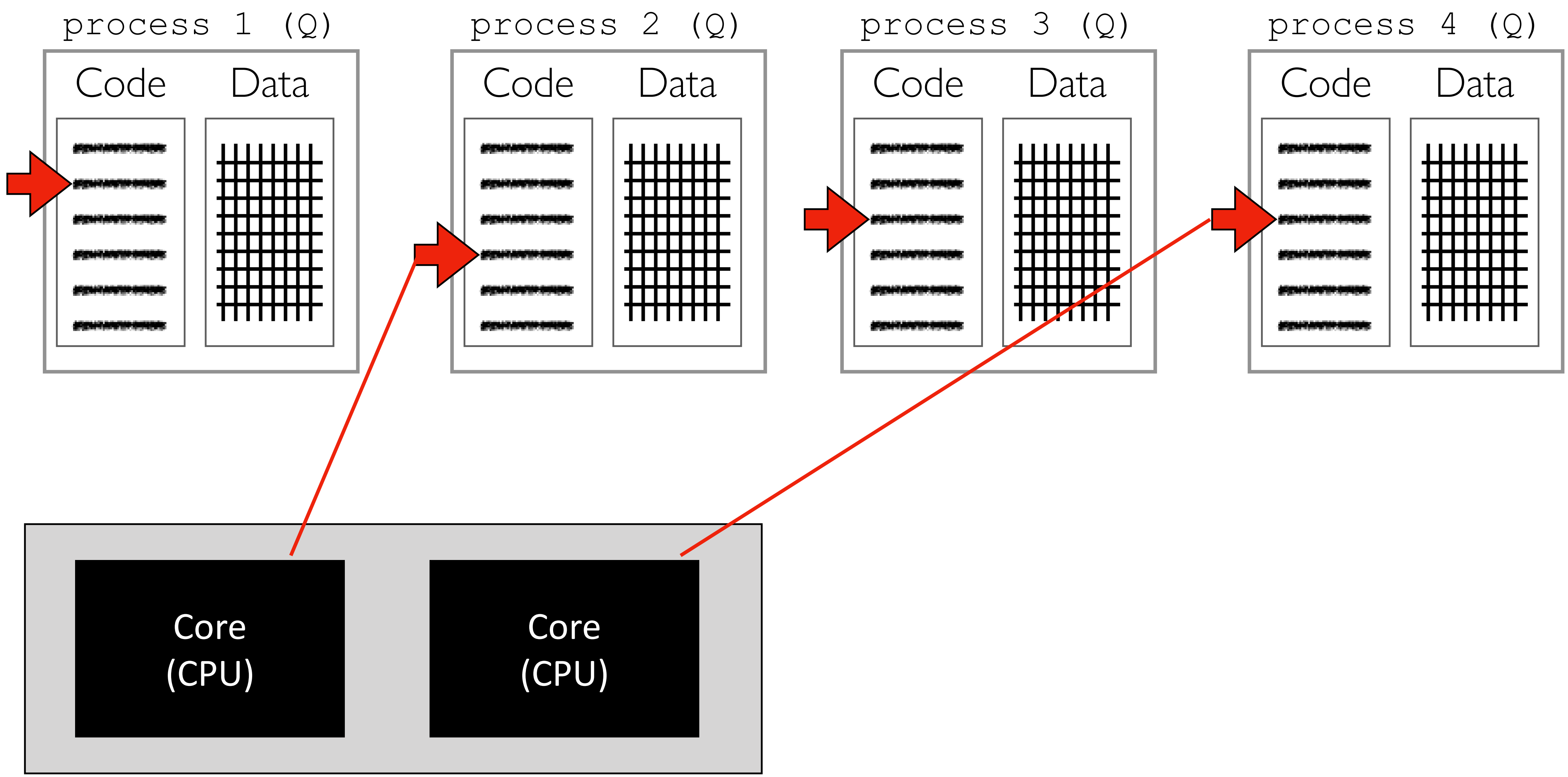
(2) Process-level Parallelism



(2) Process-level Parallelism

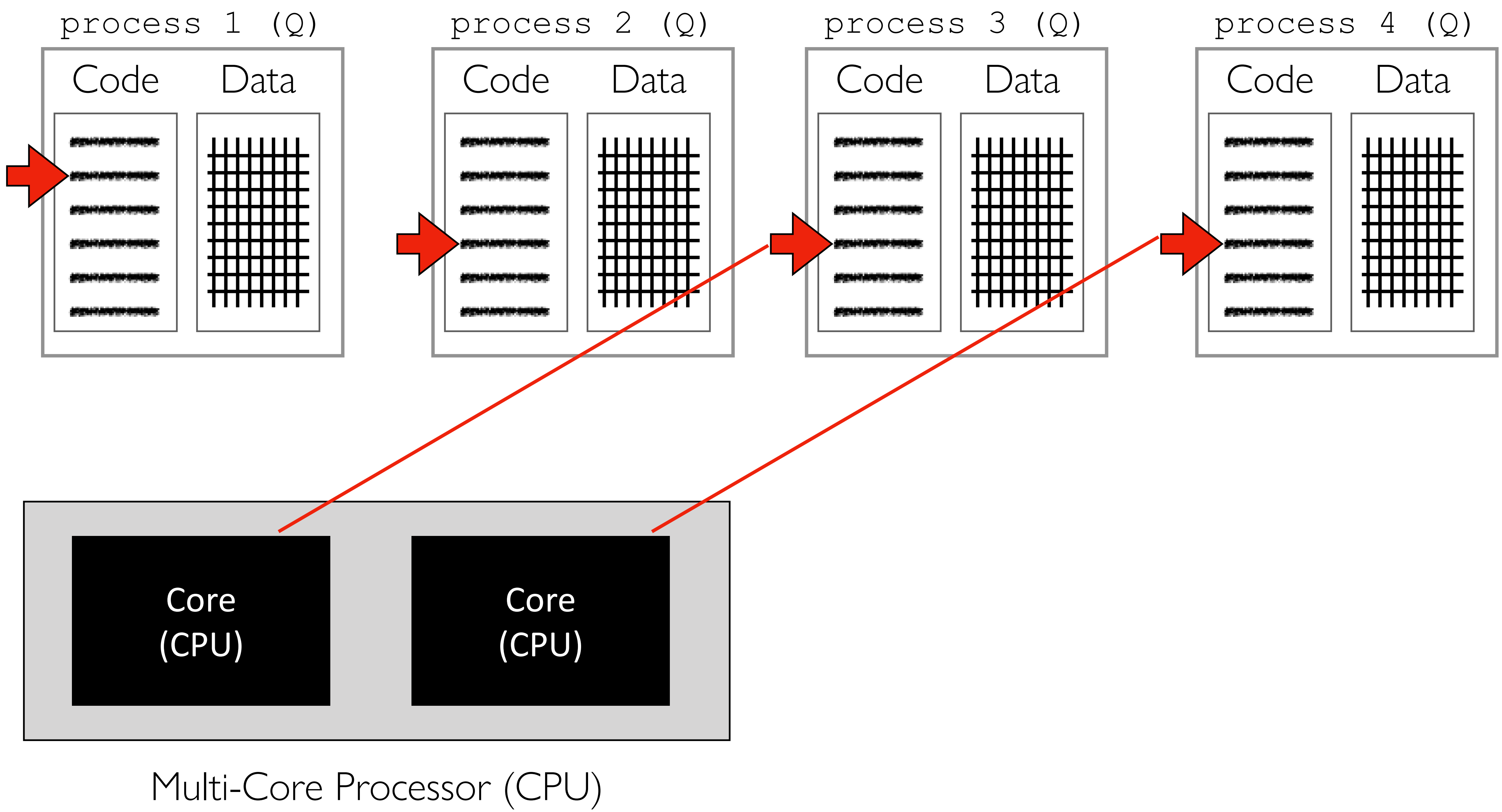


(2) Process-level Parallelism

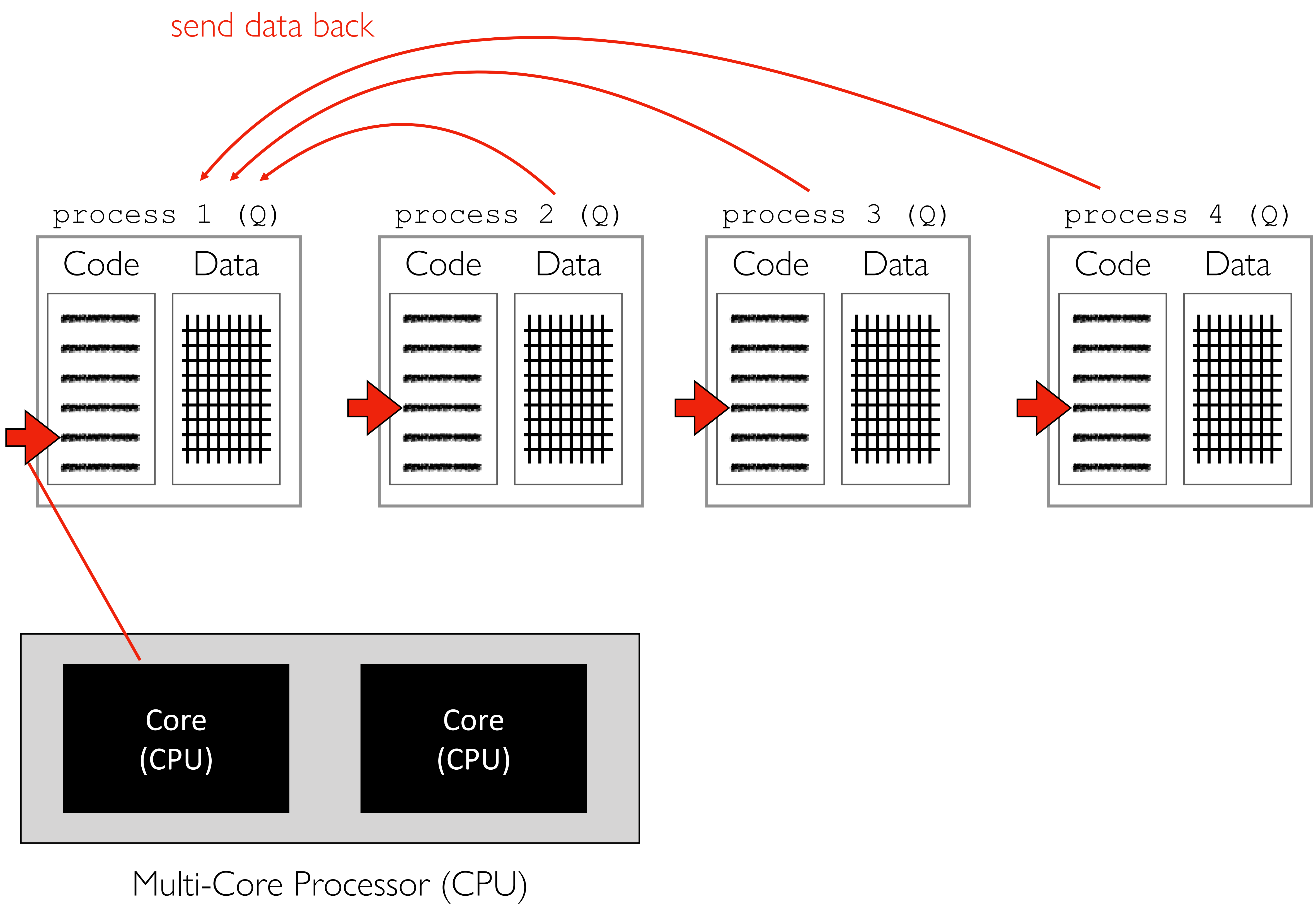


Multi-Core Processor (CPU)

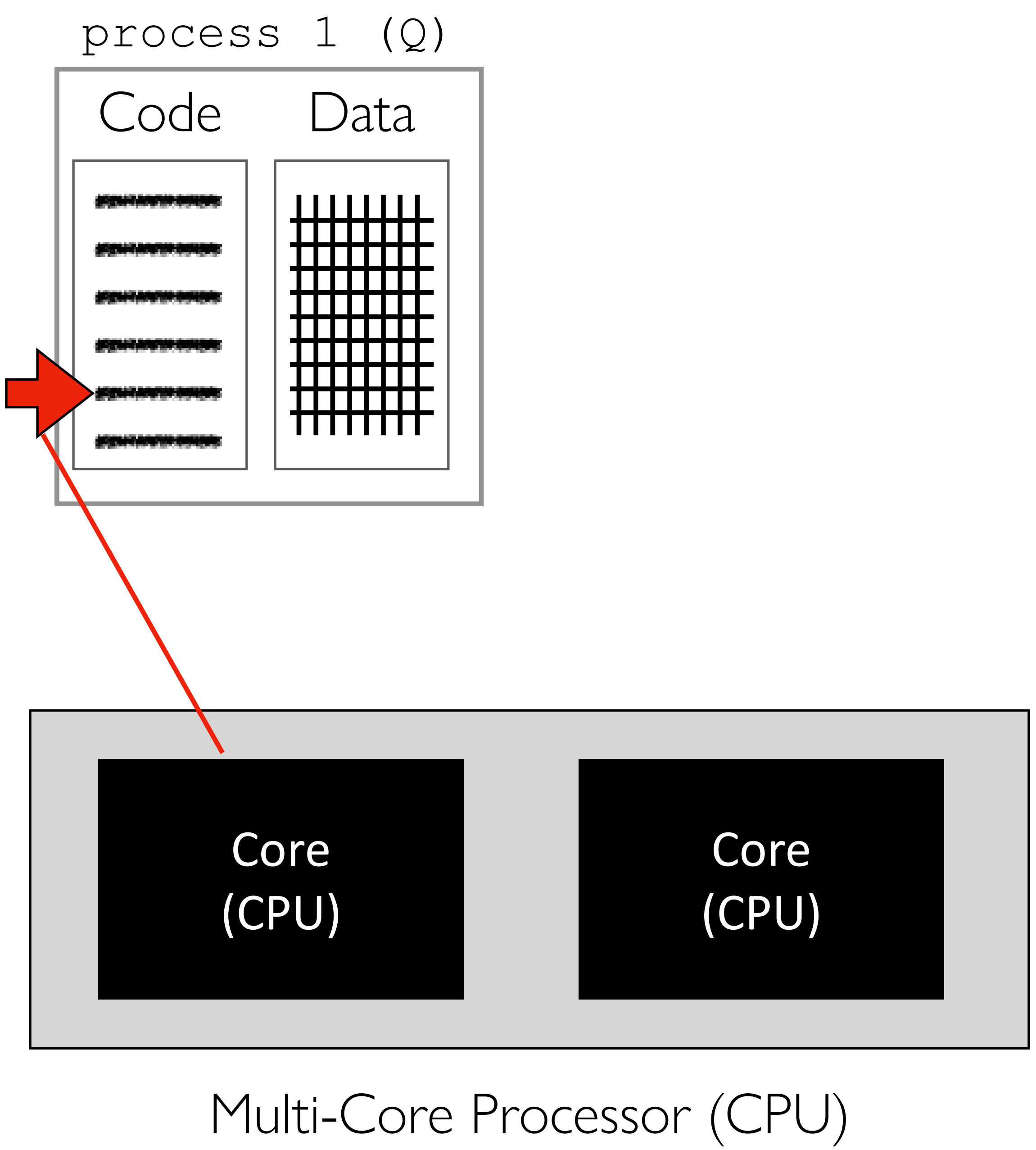
(2) Process-level Parallelism



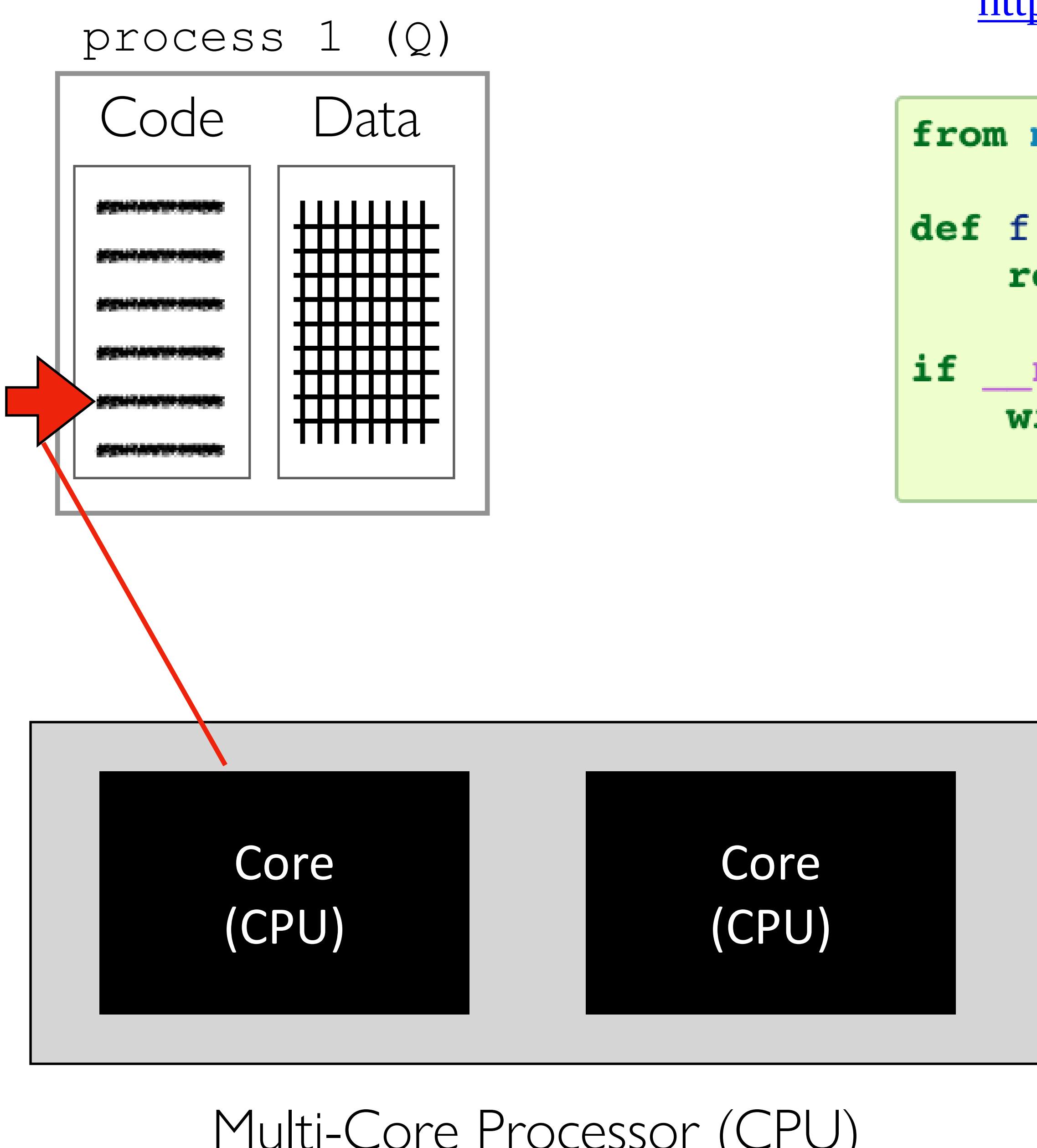
(2) Process-level Parallelism



(2) Process-level Parallelism



(2) Process-level Parallelism



https://docs.python.org/3/library/multiprocessing.html

```python
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

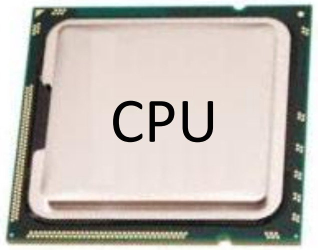
Solution: Parallelism

1. thread-level parallelism
2. process-level parallelism
3. GPU parallelism

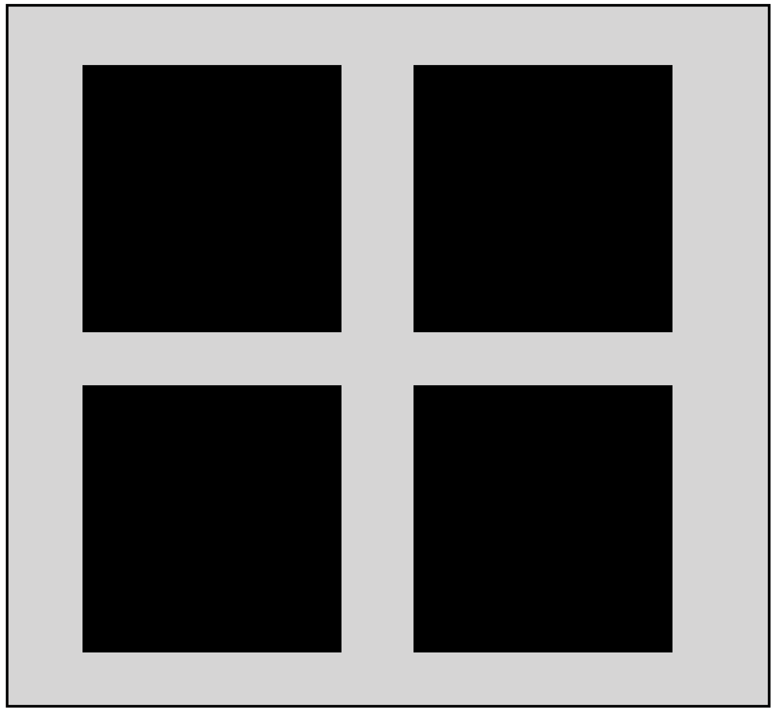
very complicated, not covered in detail

covered in CS 320

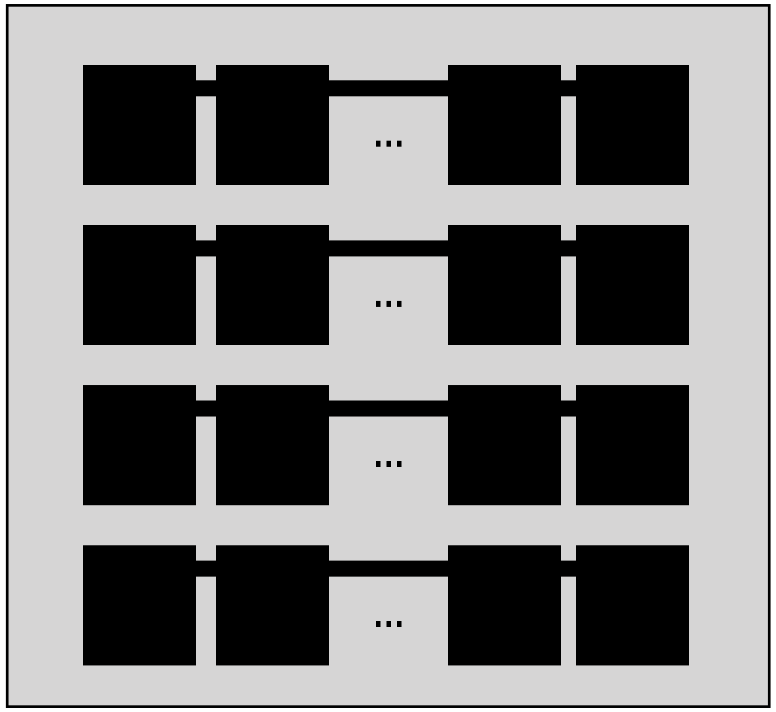
(3) GPU Parallelism



https://en.wikipedia.org/wiki/Nvidia_Tesla

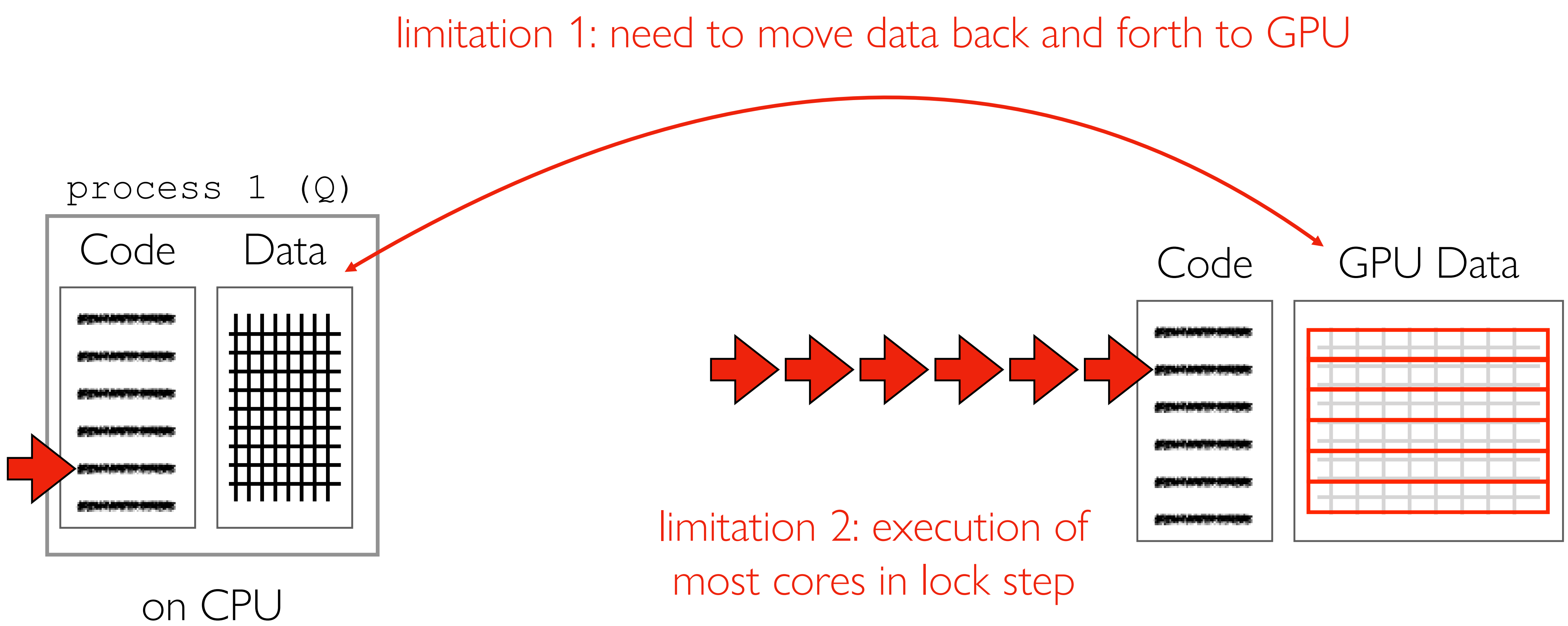


few cores that are fast,
flexible, independent

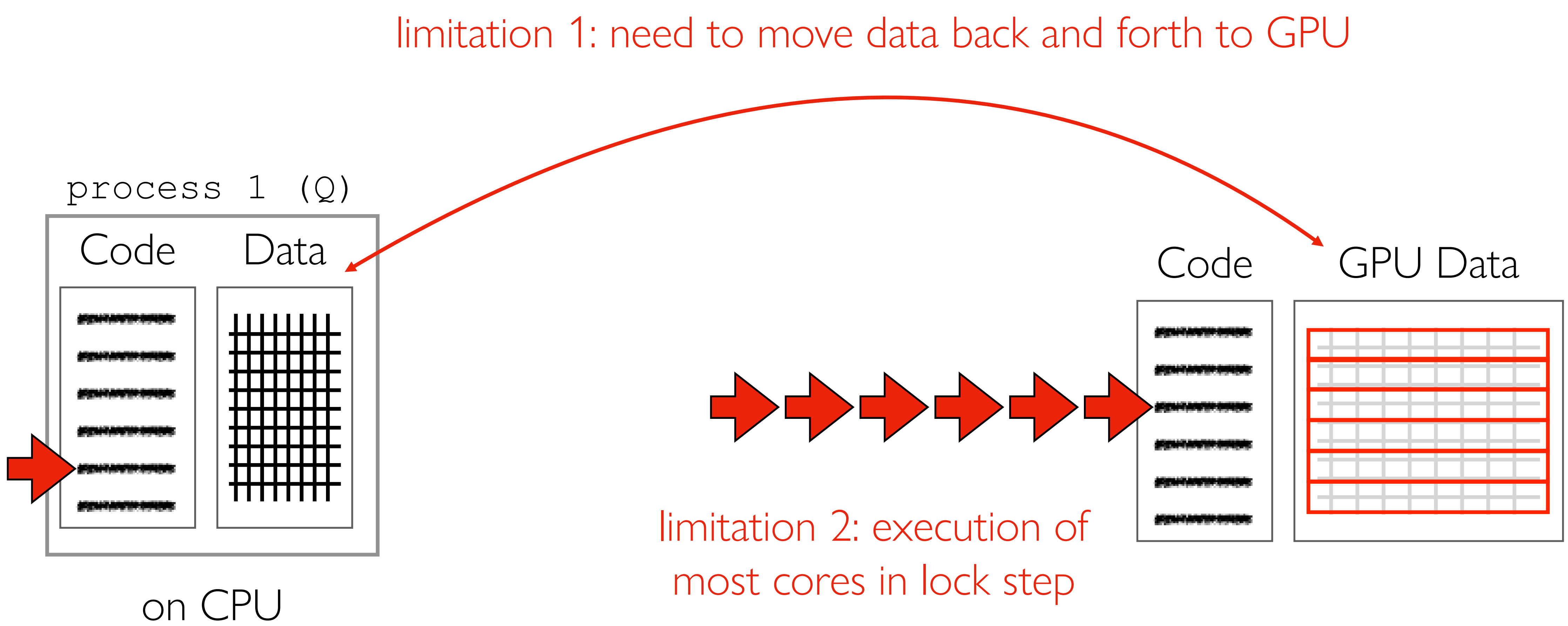


many cores that are slow,
float-optimized, coordinated

GPU Limitations

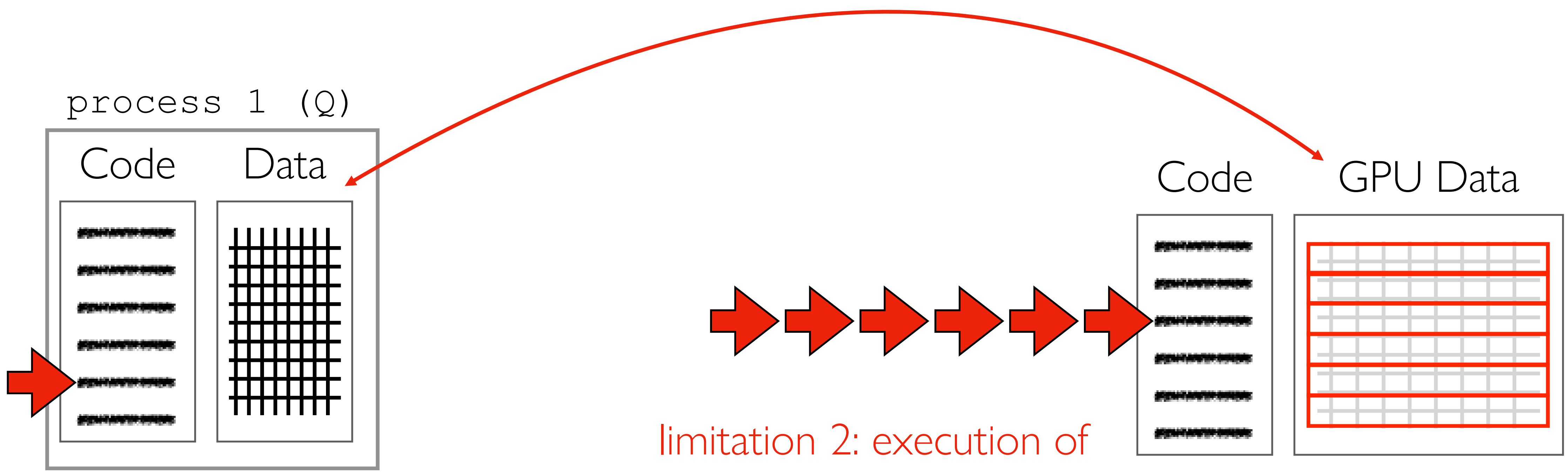


GPU Limitations



GPU Limitations

limitation 1: need to move data back and forth to GPU



on CPU

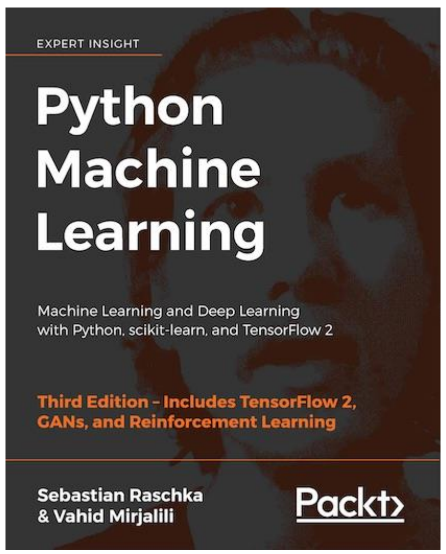
limitation 2: execution of most cores in lock step

great use case: matrix multiplication

$$\begin{bmatrix} row1 \\ row2 \\ \dots \\ rowN \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} output1 \\ output2 \\ \dots \\ outputN \end{bmatrix}$$

multiply row 1 of matrix by vector,
multiply row 2 of matrix by vector,
multiply row 3 of matrix by vector,
...

GPU vs. CPU: Cost Comparison



Specifications	Intel® Core™ i7-6900K Processor Extreme Ed.	NVIDIA GeForce® GTX™ 1080 Ti
Base Clock Frequency	3.2 GHz	< 1.5 GHz
Cores	8	3584
Memory Bandwidth	64 GB/s	484 GB/s
Floating-Point Calculations	409 GFLOPS	11300 GFLOPS
Cost	~ $1000.00	~ $700.00

https://sebastianraschka.com/books.html

The GPU is 30% cheaper but 28x faster at floating-point operations!

PyTorch

```
import numpy as np
import torch
A = np.random.normal(size=(1000,20))
x = np.random.normal(size=(20,1))
A = torch.from_numpy(A).to("cuda") # GPU
x = torch.from_numpy(x).to("cuda") # GPU
b = A @ x
b = b.to("cpu")
b
```

- CUDA: Compute Unified Device Architecture
- pytorch tensor is like numpy array
- .to("cuda") moves data to GPU
- .to("cpu") moves output back to CPU

Parallelism

1. thread-level parallelism
2. process-level parallelism
3. GPU parallelism