[544] Processes and Threads

Tyler Caraza-Harter

Learning Objectives

- describe the interactions between schedulers, CPUs, threads, and address spaces
- decide for a given scenario whether to organize code as single-threaded, multi-threaded, or multi-process
- trace through different interleavings to identify race conditions

Motivation

Modern CPUs have many cores (maybe dozens)

Trend: **more** cores rather than **faster** cores

Problem: a simple Python program can use at most ONE core
(less if it accesses files or the Internet)

Understanding threads and processes will:

- let us write programs that fully utilize CPU resources
- decide the structure of our concurrent program (threads or processes) depending on the situation

Outline

Review: Virtual Address Spaces

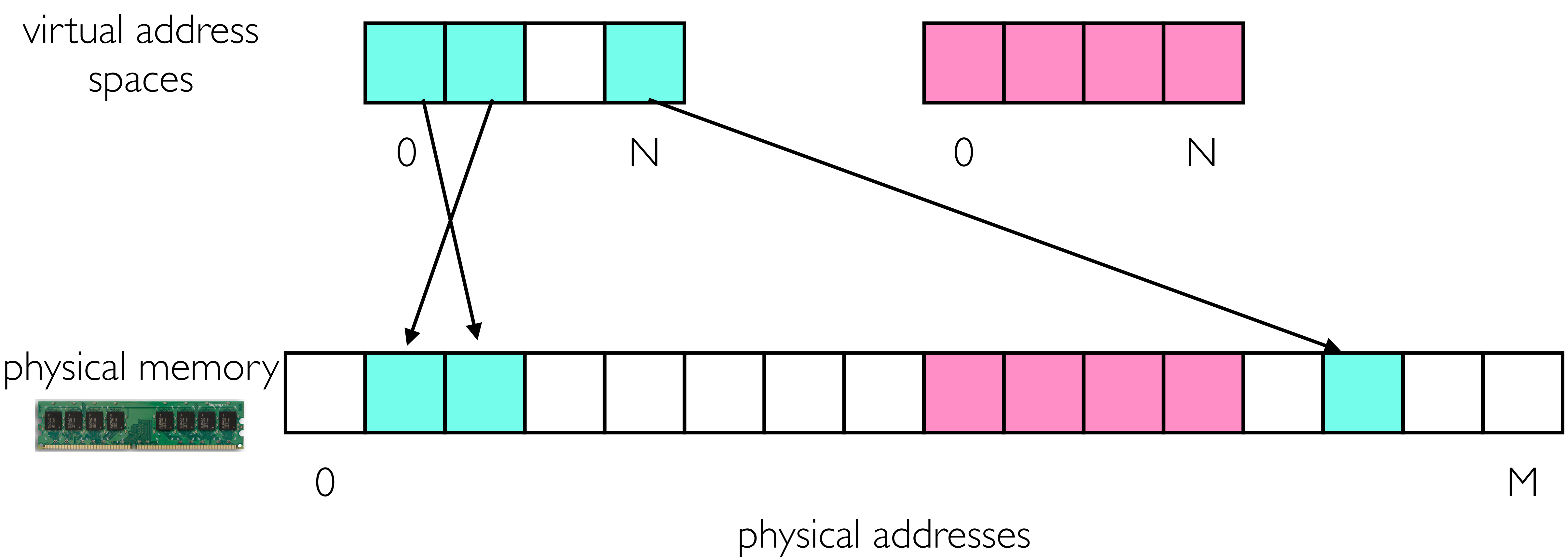
Threads

Demos and Worksheet

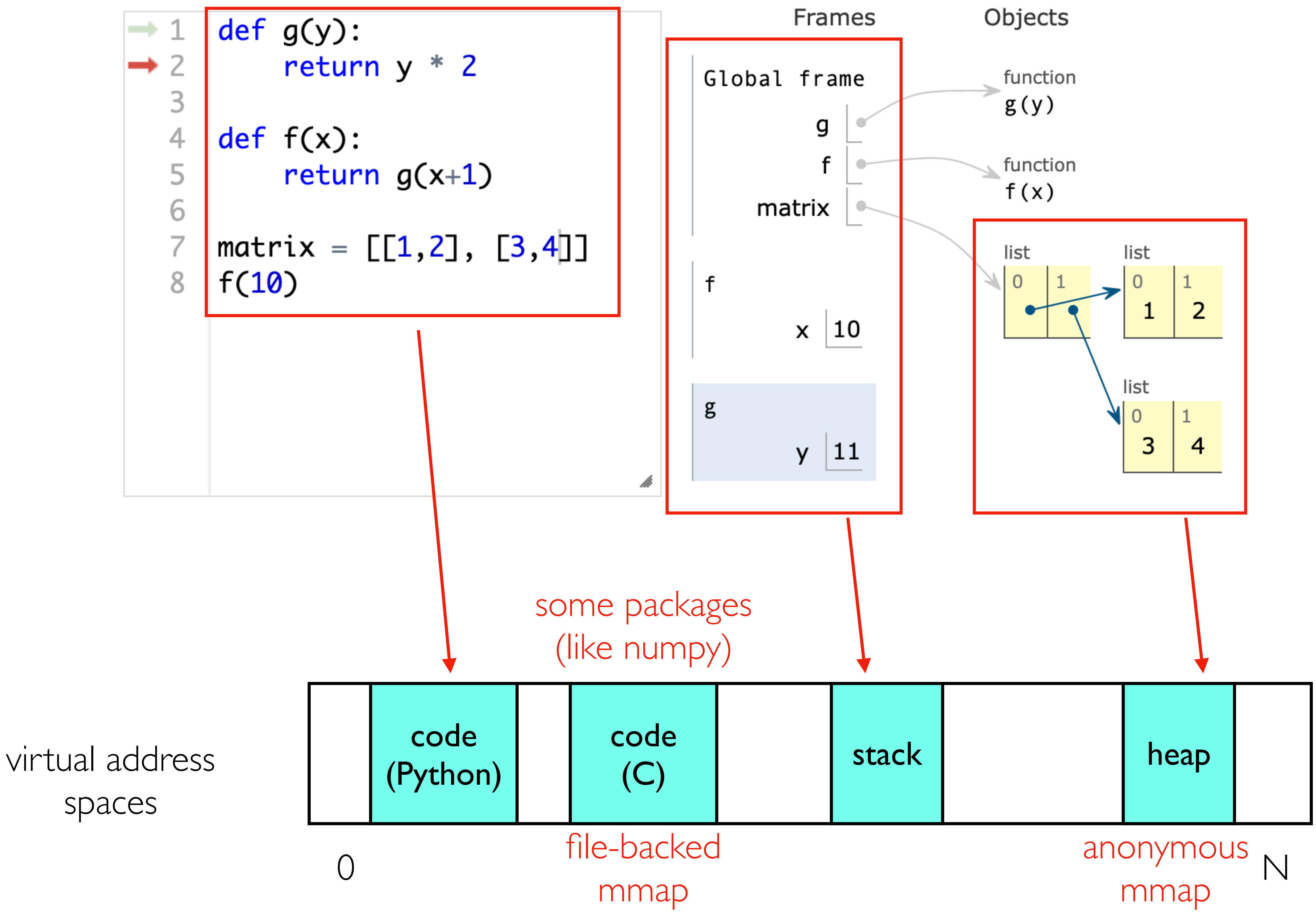
Processes and Address Spaces

Address spaces

- A process is a running program
- Each process has it's own virtual address space
- The same virtual address generally refers to different memory in different processes
- Regular processes cannot directly access physical memory or other addr spaces
- Address spaces can have holes (N is usually MUCH bigger than M)
- Physical memory for a process need not be contiguous



What goes in an address space?



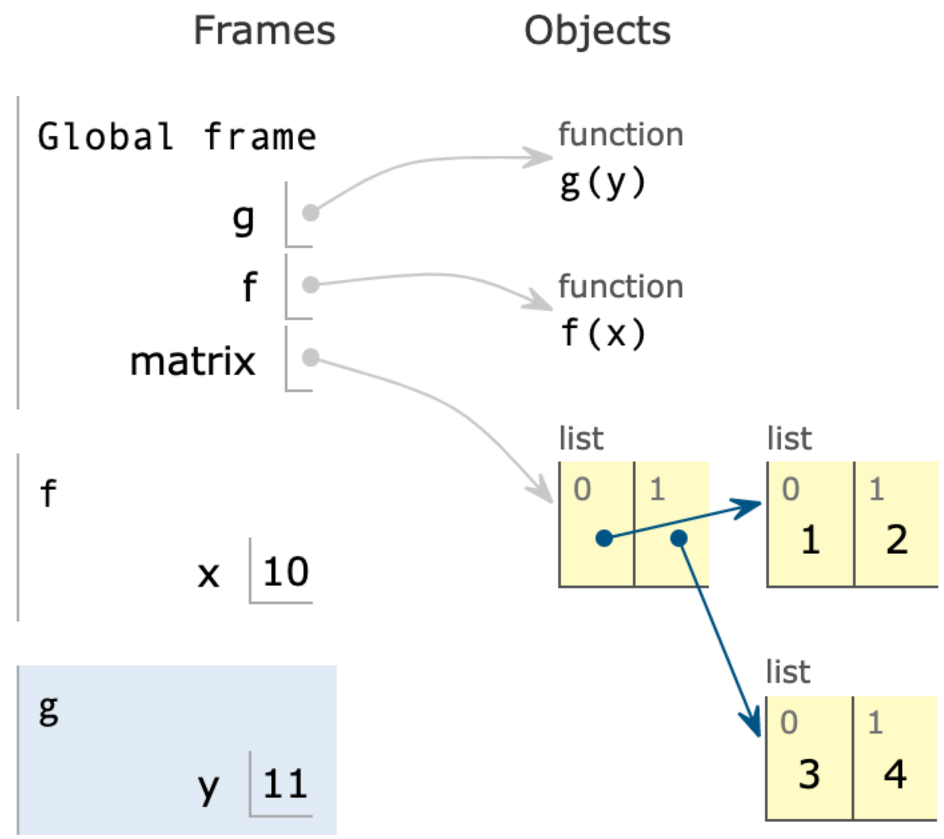
Note: the stack is contiguous, but code and heap generally are not

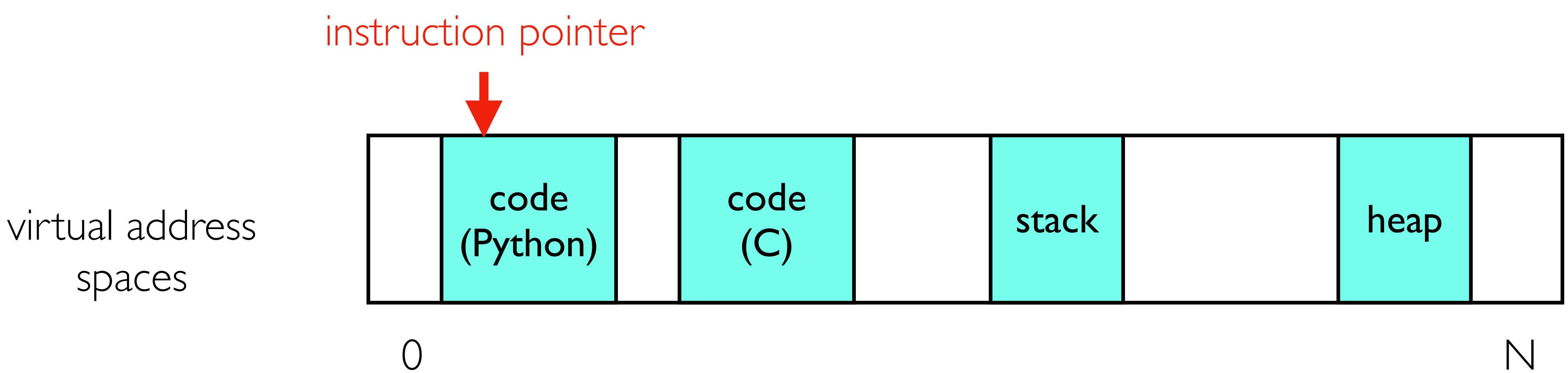
How does code execute?

```python
def g(y):
    return y * 2

def f(x):
    return g(x+1)

matrix = [[1,2], [3,4]]
f(10)
```

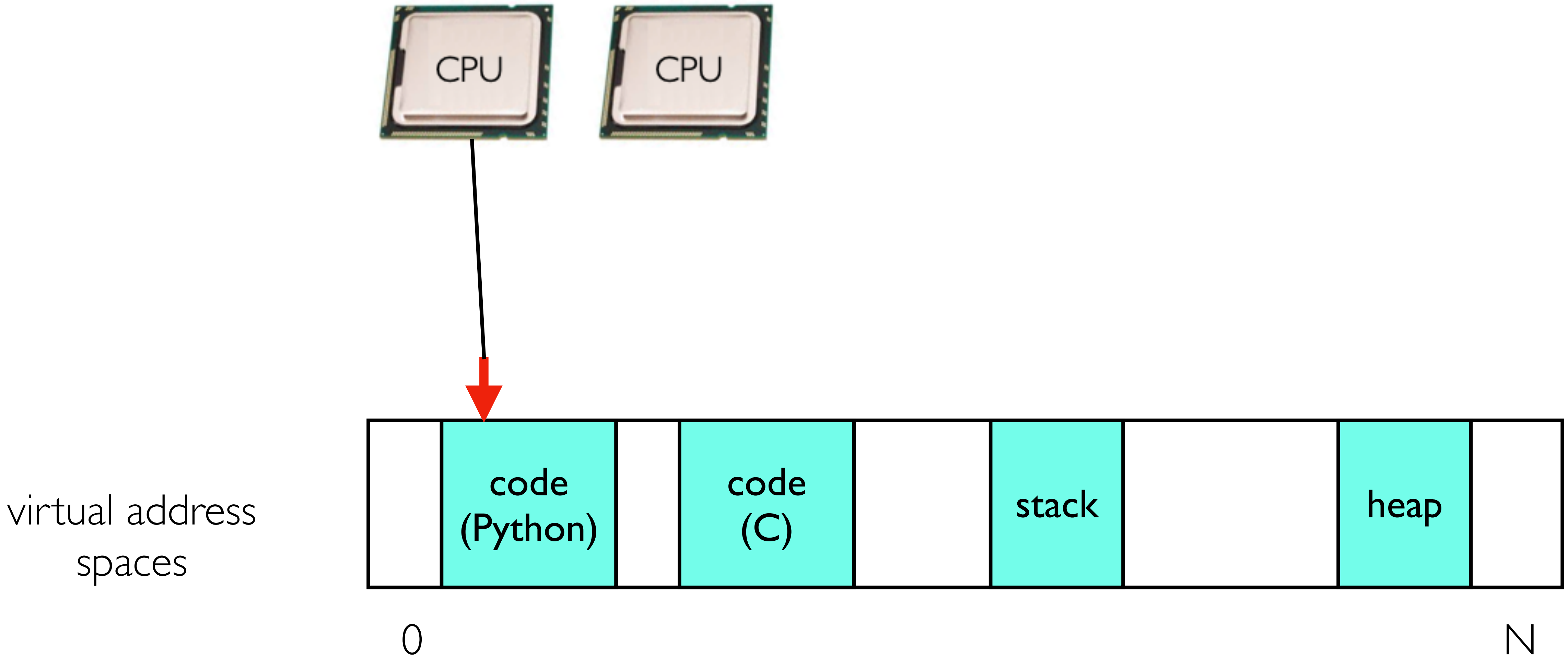




How does code execute?

CPUs

- CPUs are attached to at most one instruction pointer at any given time
- they run code by executing instructions and advancing the instruction pointer
- **Note**: interpreter left out for simplicity (CPU points to interpreter code, which points to Python bytecode)



Outline

Review: Virtual Address Spaces

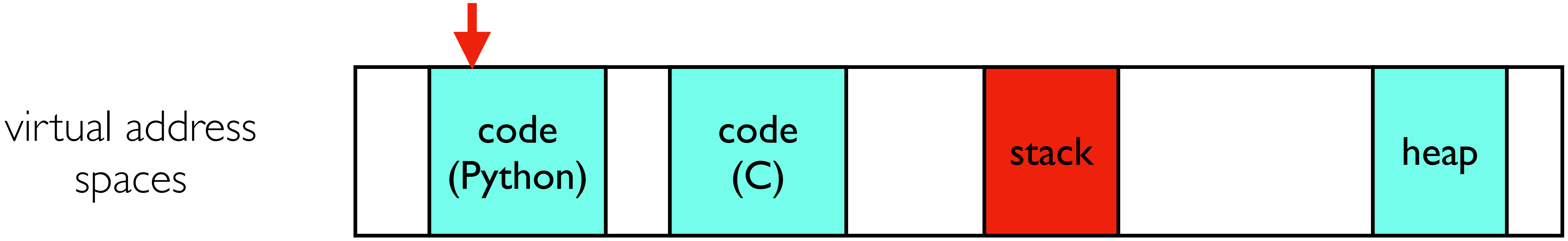
Threads

Demos and Worksheet

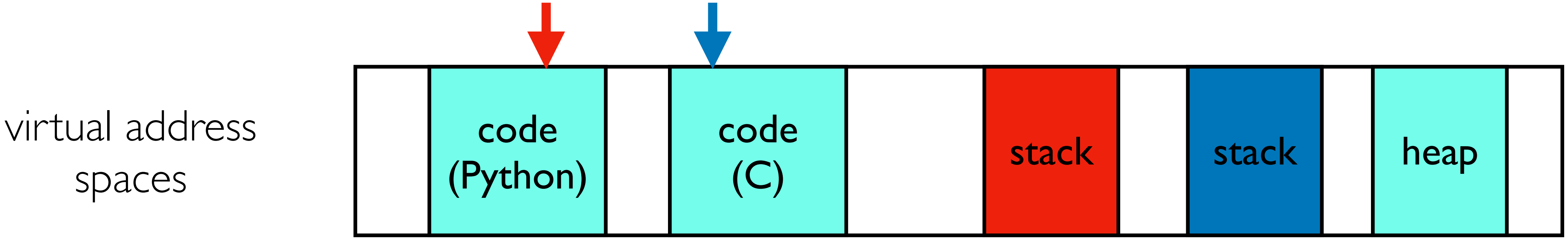
Threads

Threads have their own instruction pointers and stacks, but share the heap.

Single-threaded process:



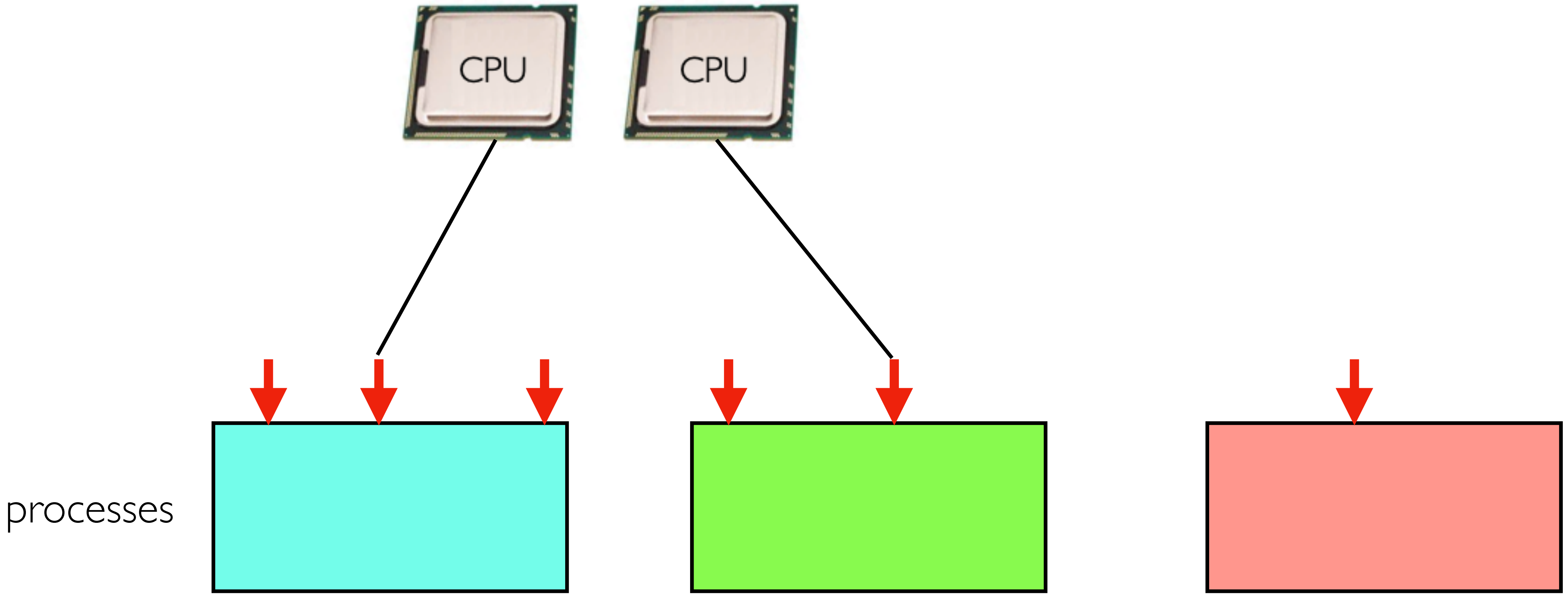
Multi-threaded process:



Context Switch

Schedulers

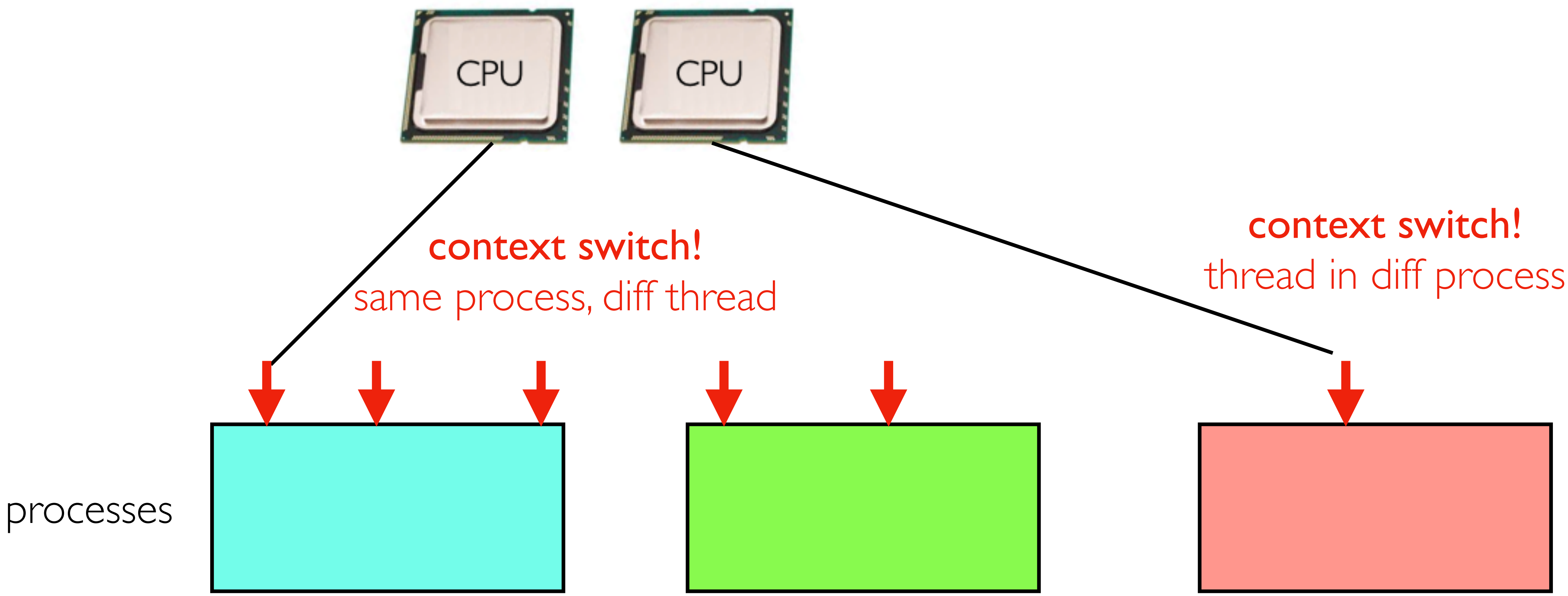
- CPU scheduler is an important sub system in an operating system
- schedulers decide when to context switch between threads
- context switch: change which thread a CPU is running



Context Switch

Schedulers

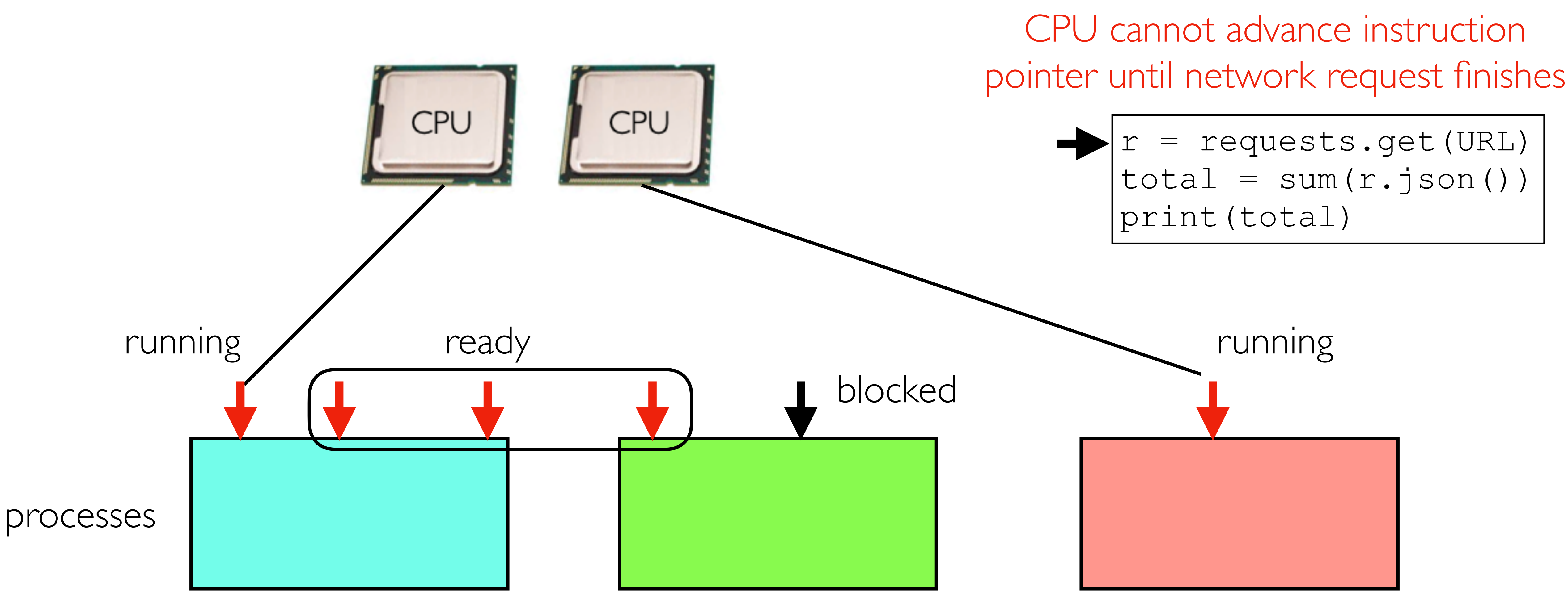
- CPU scheduler is an important sub system in an operating system
- schedulers decide when to context switch between threads
- context switch: change which thread a CPU is running



Scheduling Restrictions: Blocked Threads

Threads can be in one of three states

- running: CPU is executing it
- blocked: waiting on something other than CPU (network, input, disk, etc)
- ready: scheduler can choose to context switch to it



Efficient Use of Compute Resources

Wasted cores: (1) not enough threads (2) blocked threads

For 100% CPU utilization (difficult goal)

- need at least one ready/running thread for each CPU core
- generally need more threads than cores (threads are often blocked)
- **threads could be in one process (or many)**

Multi-threaded applications

- good when multiple threads need to access frequently modified data structures
- new kinds of bugs possible (race conditions, deadlock)

Multi-process applications (https://docs.python.org/3/library/multiprocessing.html)

- easier to program (or just manually launch several processes in background)
- better at keeping multiple cores busy simultaneously (Python specific)

Both approaches work well for dealing with blocked threads

Coding Demos, Worksheet

Thread operations

- t = threading.Thread()
- t.start(target=????, args=[????])
- t.join()
- t.get_native_id()