

Heterogeneous Computing for AI - Lecture ~02

Introduction to Concurrency in Python

Raghava Mukkamala

Associate Professor, Director for Centre for Business Data Analytics (bda.cbs.dk)

Department of Digitalization, Copenhagen Business School, Denmark

Email: rrm.digi@cbs.dk

Associate Professor, Department of Technology,
Kristiania University College, Oslo, Norway

Many slides are taken from the following authors with due respect to their contributions.

1) An Introduction to Python Concurrency by David Beazley

Outline

- The von Neumann architecture
- Basic Concurrency Concepts
- Python for Concurrency
- Threading Model in Python
- Hands-on exercises on Python Threads

Learning goals for today

Theoretical

- Gain knowledge about CPU architecture
- Understand what concurrency is in programming
- Learn the Python Threading model

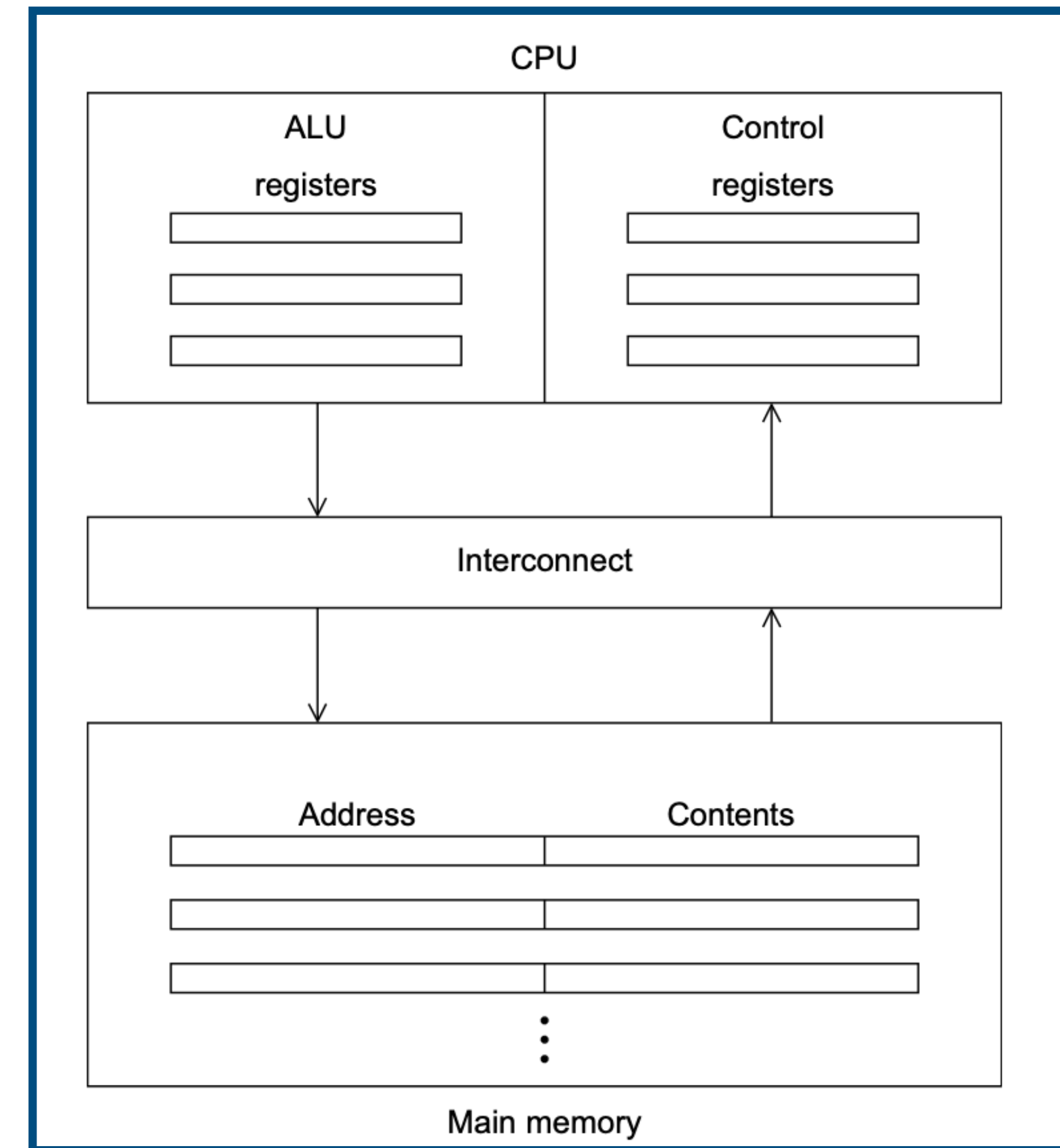
Practical

- Be able to identify the bottlenecks in programs
- Be able to program using threads in Python

The von Neumann architecture

The von Neumann architecture

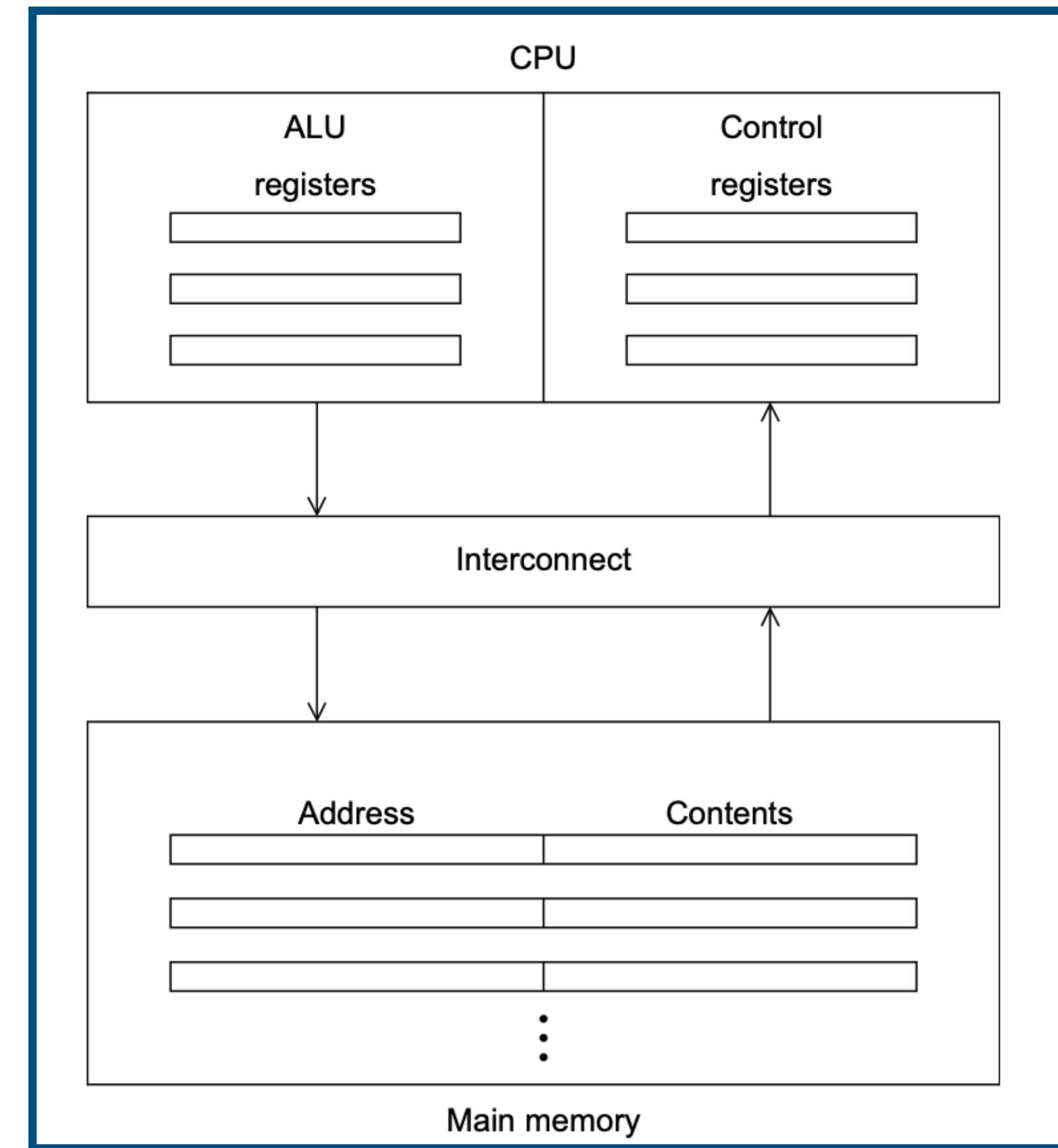
- Classical architecture consists of
 - Central Processing Unit (CPU)/ Process/Core
 - Main Memory: For storage of both data and instructions
 - Interconnection: transfer of data & instructions from memory to CPU and vice versa
- CPU consists of
 - Control Unit decides which instructions should be executed
 - Arithmetic Logic Unit (ALU), responsible for executing instructions
 - Registers, very fast storage to store the state of executing program and CPU data



The von Neumann architecture - II

- **Bottleneck**

- interconnect determines the rate at which instructions and data can be accessed
- In 2010, CPUs can execute 100 times faster than they fetch/write data from memory
- Analogy: a large company has a single factory (the CPU) in one town and a single warehouse (main memory) in another. All the raw materials used in manufacturing the products are stored in the warehouse.
- If rate of manufacturing > transportation of materials and finished products —> idle time for machinery in the factory



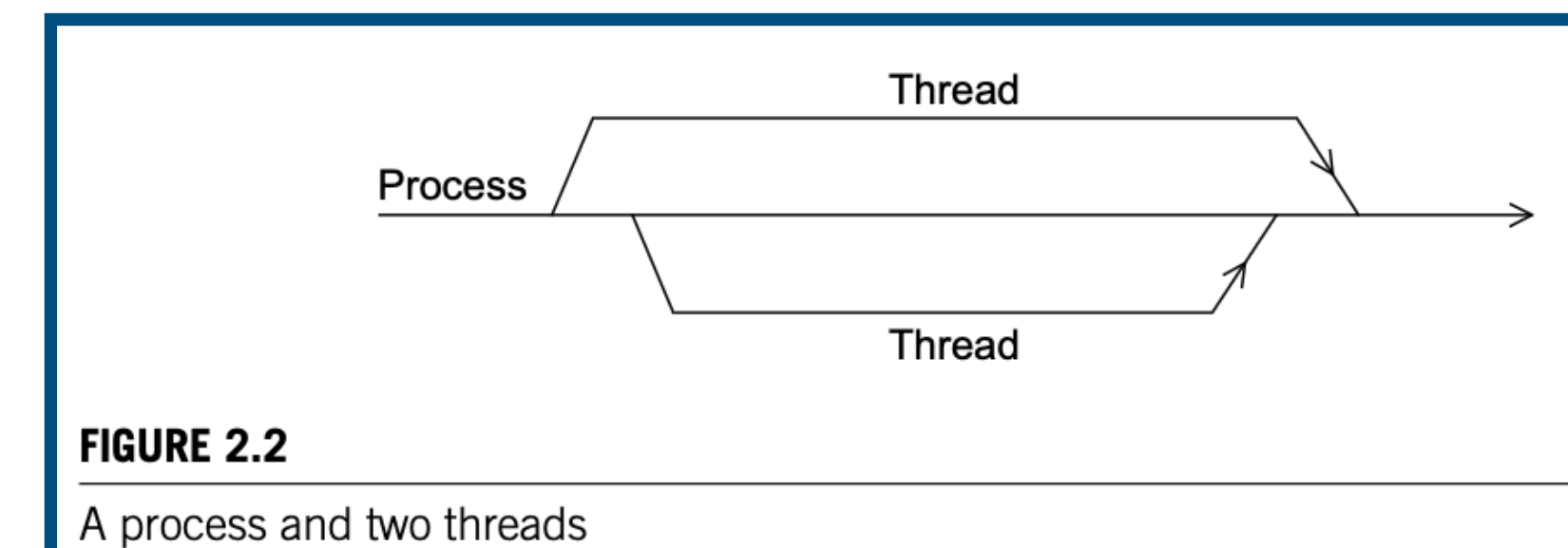
Processes, Multitasking, and Threads

- OS is a major piece of software
 - that determines which programs can run and when they can run
 - controls the allocation of memory to running programs and access to peripheral devices such as hard disks and network interface cards.
- When a user runs a program, OS creates a **process**, an instance of a computer program that is being executed and contains
 - executable machine language code
 - block of memory, heap, a call stack and others
 - process information such as state, whether process is ready to run, content of registers, etc.

Processes, Multitasking, and Threads

- Modern OS supports **multitasking**, support for apparent simultaneous execution of multiple programs
 - Even possible for single-core, as CPU can switch threads for small **time slice** (few milliseconds)
- If process needs to wait (e.g. I/O), then it will **block**, which means stop executing the program and switch to another
- **Threading** provides a mechanism for programmers to divide their programs into more or less independent tasks with the property that when one thread is blocked, another thread can be run

[IPP: p. 18]



Basic Concurrency Concepts

Basic concepts

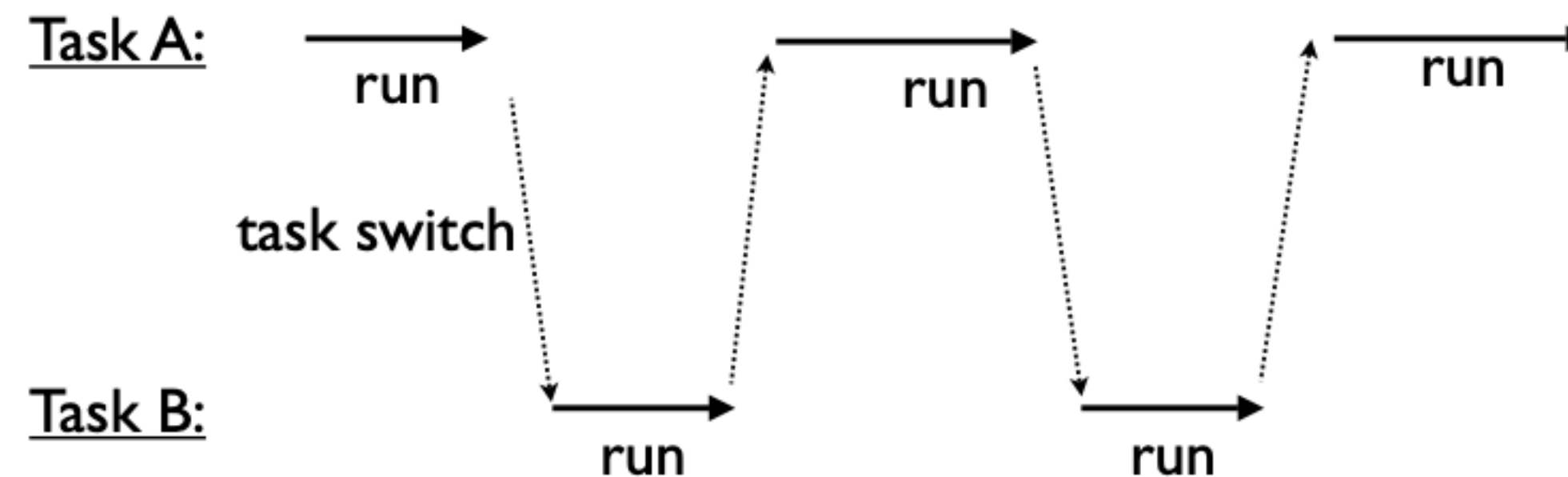
Concurrent Programming

- Creation of programs that can work on more than one thing at a time
- Example : A network server that communicates with several hundred clients all connected at once
- Example : A big number crunching job that spreads its work across multiple CPUs

Basic concepts

Multitasking

- Concurrency typically implies "multitasking"

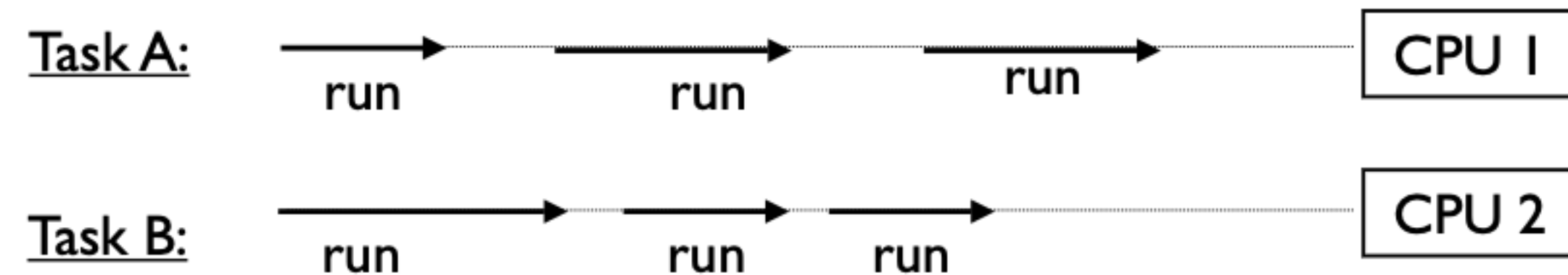


- If only one CPU is available, the only way it can run multiple tasks is by rapidly switching between them

Basic concepts

Parallel Processing

- You may have parallelism (many CPUs)
- Here, you often get simultaneous task execution

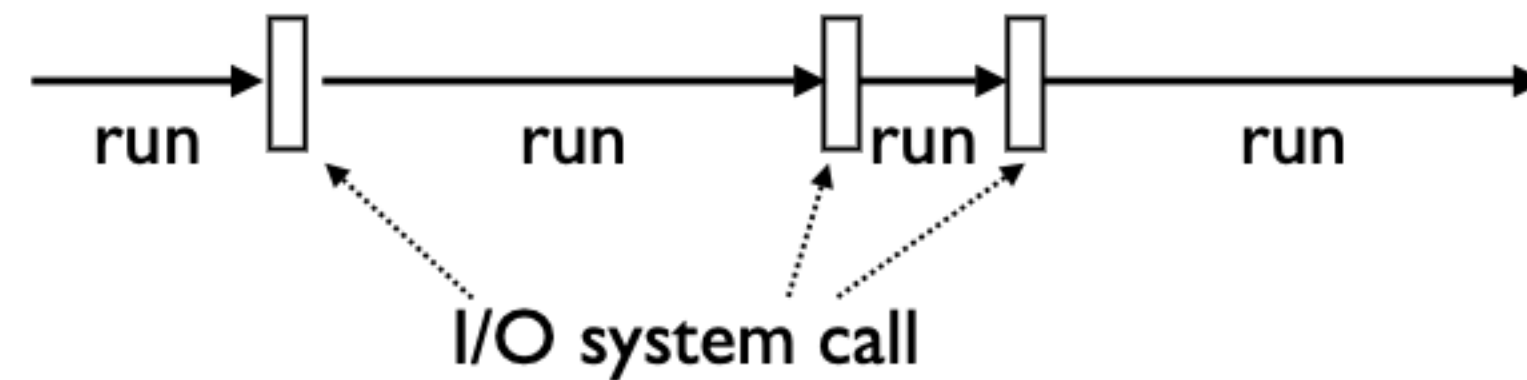


- Note: If the total number of tasks exceeds the number of CPUs, then each CPU also multitasks

Basic concepts

Task Execution

- All tasks execute by alternating between CPU processing and I/O handling

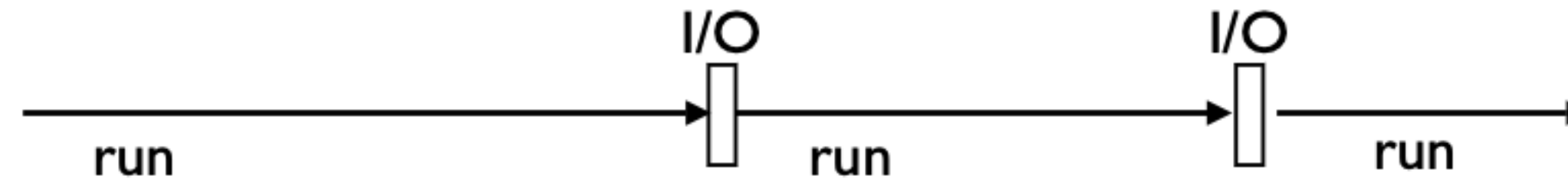


- For I/O, tasks must wait (sleep)
- Behind the scenes, the underlying system will carry out the I/O operation and wake the task when it's finished

Basic concepts

CPU Bound Tasks

- A task is "CPU Bound" if it spends most of its time processing with little I/O



- Examples:
 - Crunching big matrices
 - Image processing

Basic concepts

I/O Bound Tasks

- A task is "I/O Bound" if it spends most of its time waiting for I/O

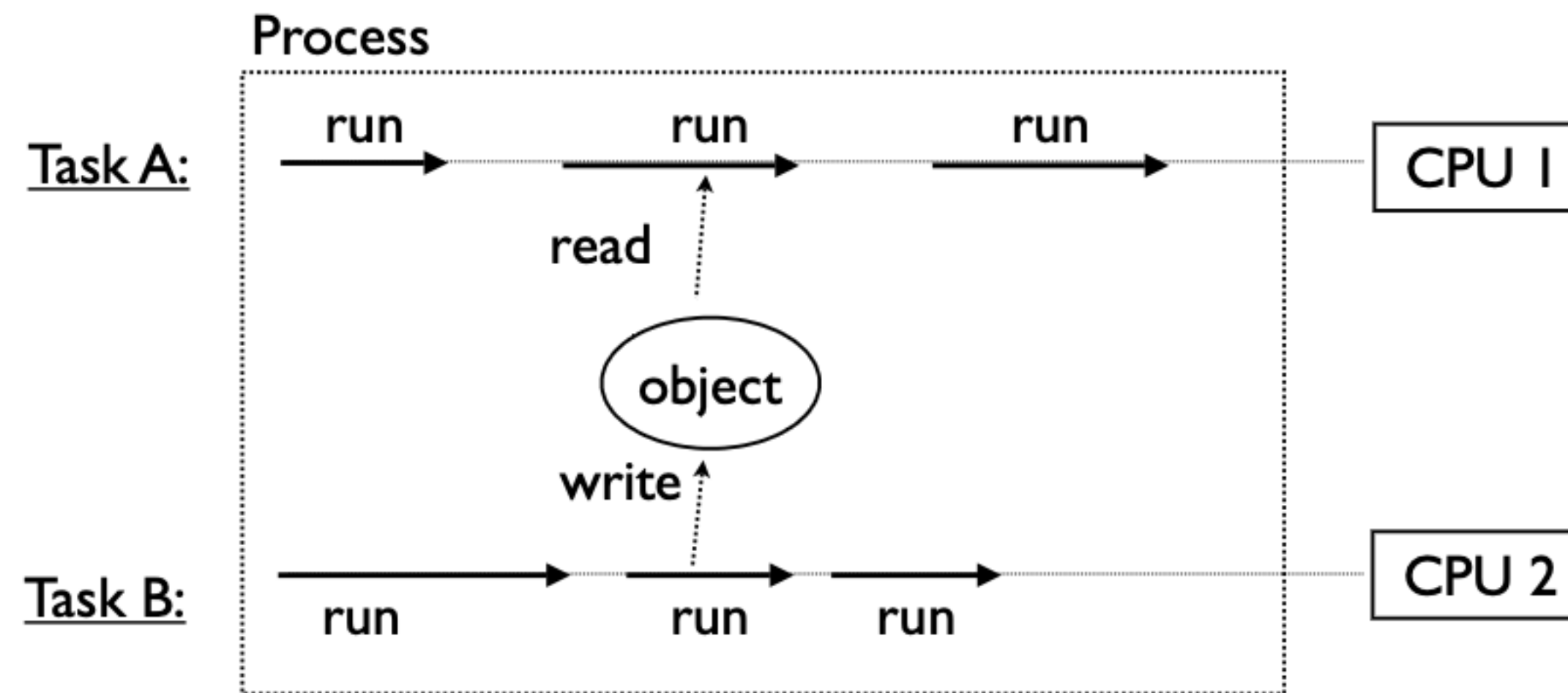


- Examples:
 - Reading input from the user
 - Networking
 - File processing
- Most "normal" programs are I/O bound

Basic concepts

Shared Memory

- Tasks may run in the same memory space

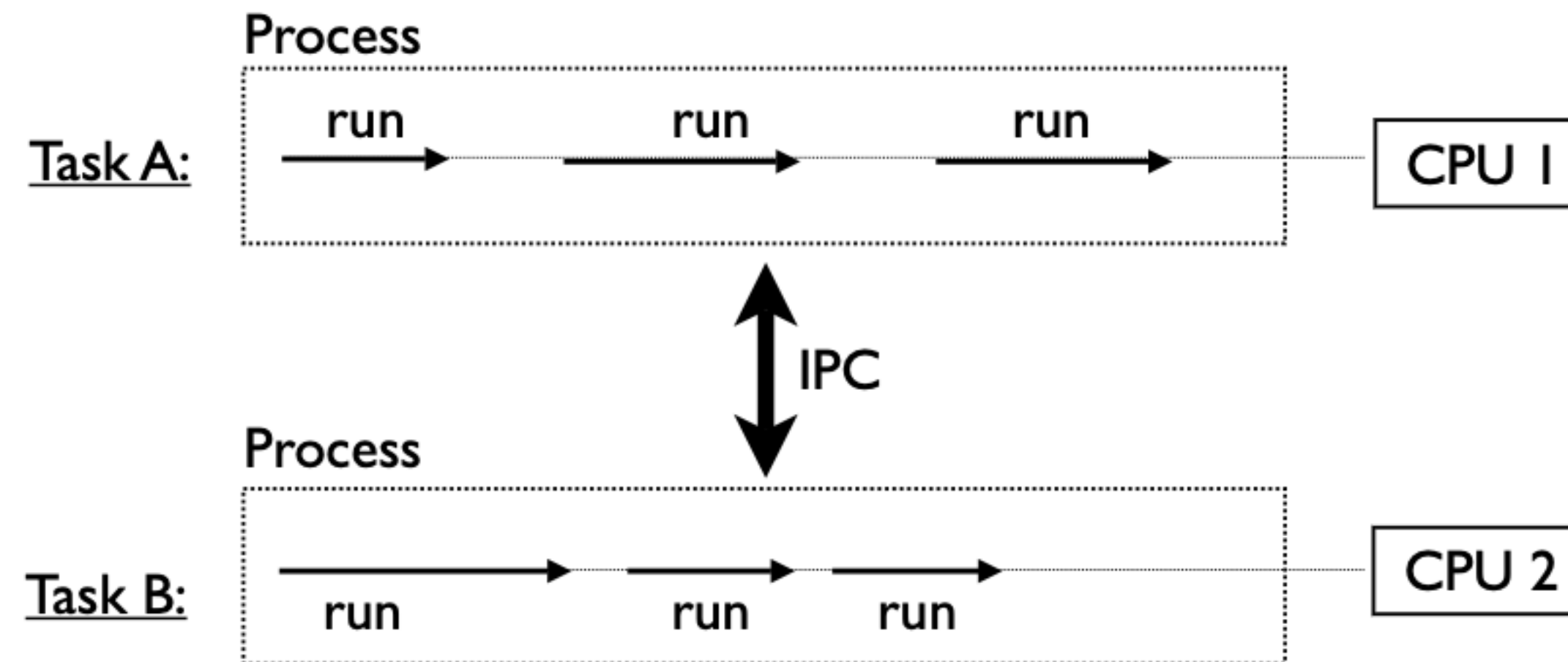


- Simultaneous access to objects
- Often a source of unspeakable peril

Basic concepts

Processes

- Tasks might run in separate processes

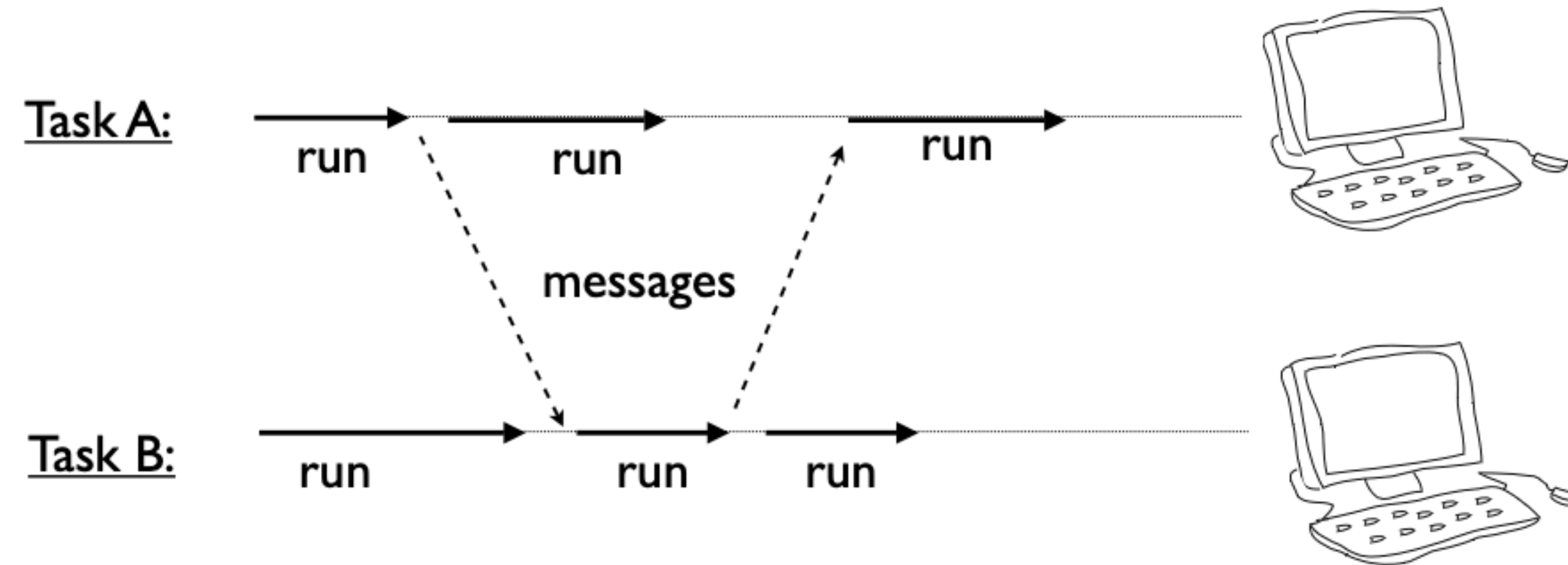


- Processes coordinate using IPC
- Pipes, FIFOs, memory mapped regions, etc.

Basic concepts

Distributed Computing

- Tasks may be running on distributed systems



- For example, a cluster of workstations
- Communication via sockets

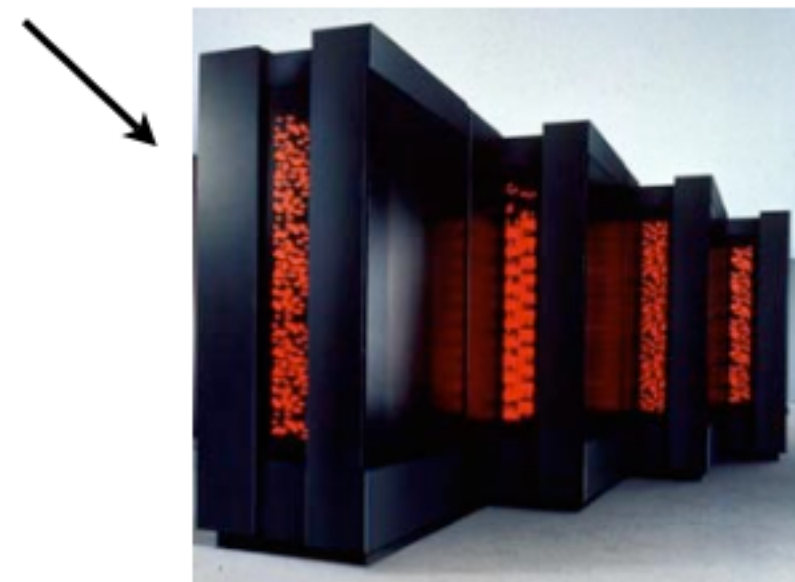
Python for Concurrency

Python for Concurrency

Some Issues

- Python is interpreted

"What the hardware giveth, the software taketh away."
- Frankly, it doesn't seem like a natural match for any sort of concurrent programming
- Isn't concurrent programming all about high performance anyways???



Copyright (C) 2009, David Beazley, <http://www.dabeaz.com>

Python for Concurrency

Why Use Python at All?

- Python is a very high level language
- And it comes with a large library
 - Useful data types (dictionaries, lists, etc.)
 - Network protocols
 - Text parsing (regexs, XML, HTML, etc.)
 - Files and the file system
 - Databases
- Programmers like using this stuff...

Python for Concurrency

Programmer Performance

- Programmers are often able to get complex systems to "work" in much less time using a high-level language like Python than if they're spending all of their time hacking C code.

"The best performance improvement is the transition from the nonworking to the working state."

- John Ousterhout

"Premature optimization is the root of all evil."

- Donald Knuth

"You can always optimize it later."

- Unknown

Python for Concurrency

Performance is Irrelevant

- Many concurrent programs are "I/O bound"
- They spend virtually all of their time sitting around waiting
- Python can "wait" just as fast as C (maybe even faster--although I haven't measured it).
- If there's not much processing, who cares if it's being done in an interpreter? (One exception : if you need an extremely rapid response time as in real-time systems)

Threading Model in Python

Python Thread Programming

Concept: Threads

- What most programmers think of when they hear about "concurrent programming"
- An independent task running inside a program
- Shares resources with the main program (memory, files, network connections, etc.)
- Has its own independent flow of execution (stack, current instruction, etc.)

Python Thread Programming

Thread Basics

% python program.py

↓
statement
statement
...
↓

"main thread"

Program launch. Python
loads a program and starts
executing statements

Python Thread Programming

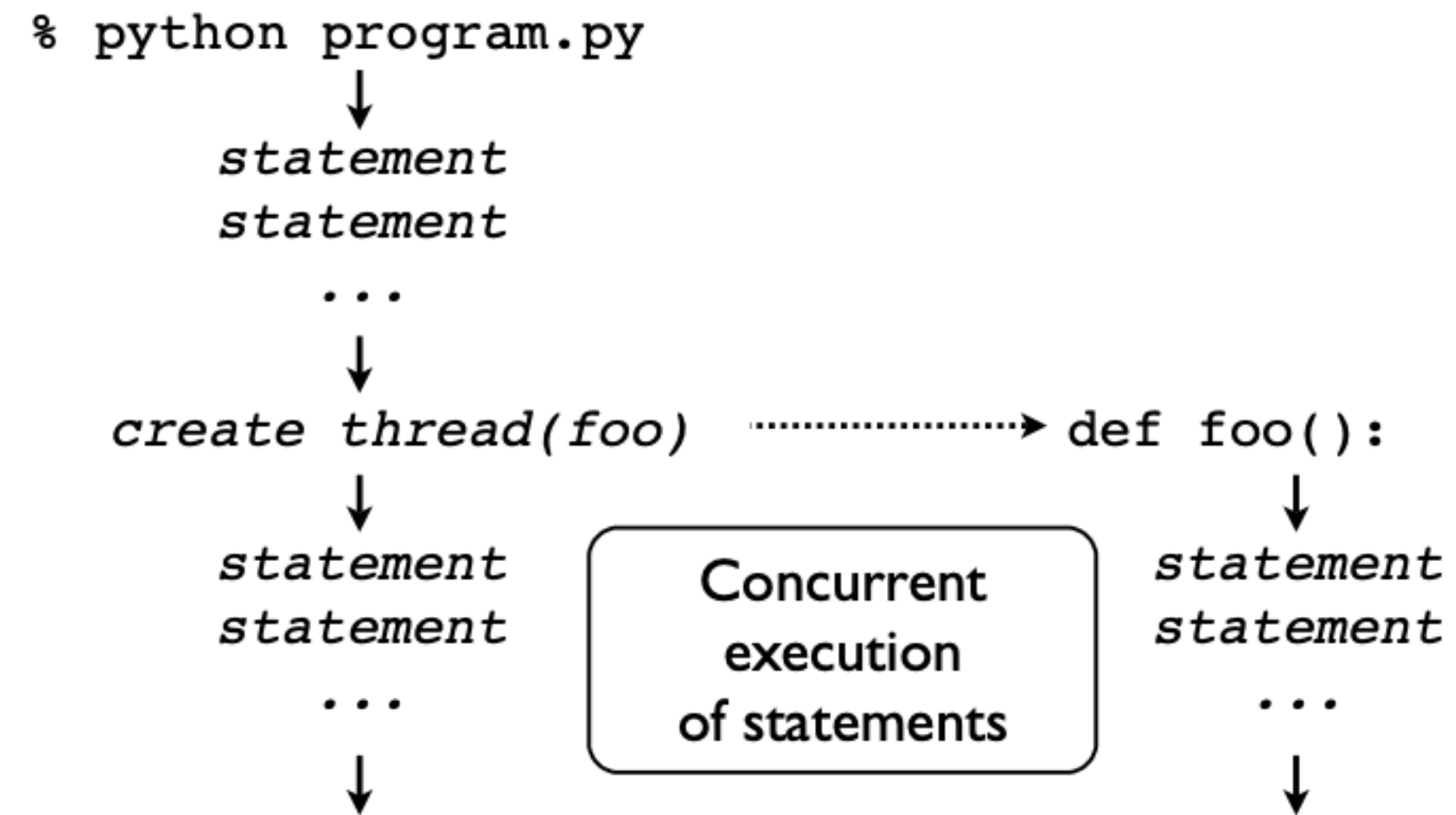
Thread Basics

`% python program.py`
↓
statement
statement
...
↓
create thread(foo)→ `def foo():`

Creation of a thread.
Launches a function.

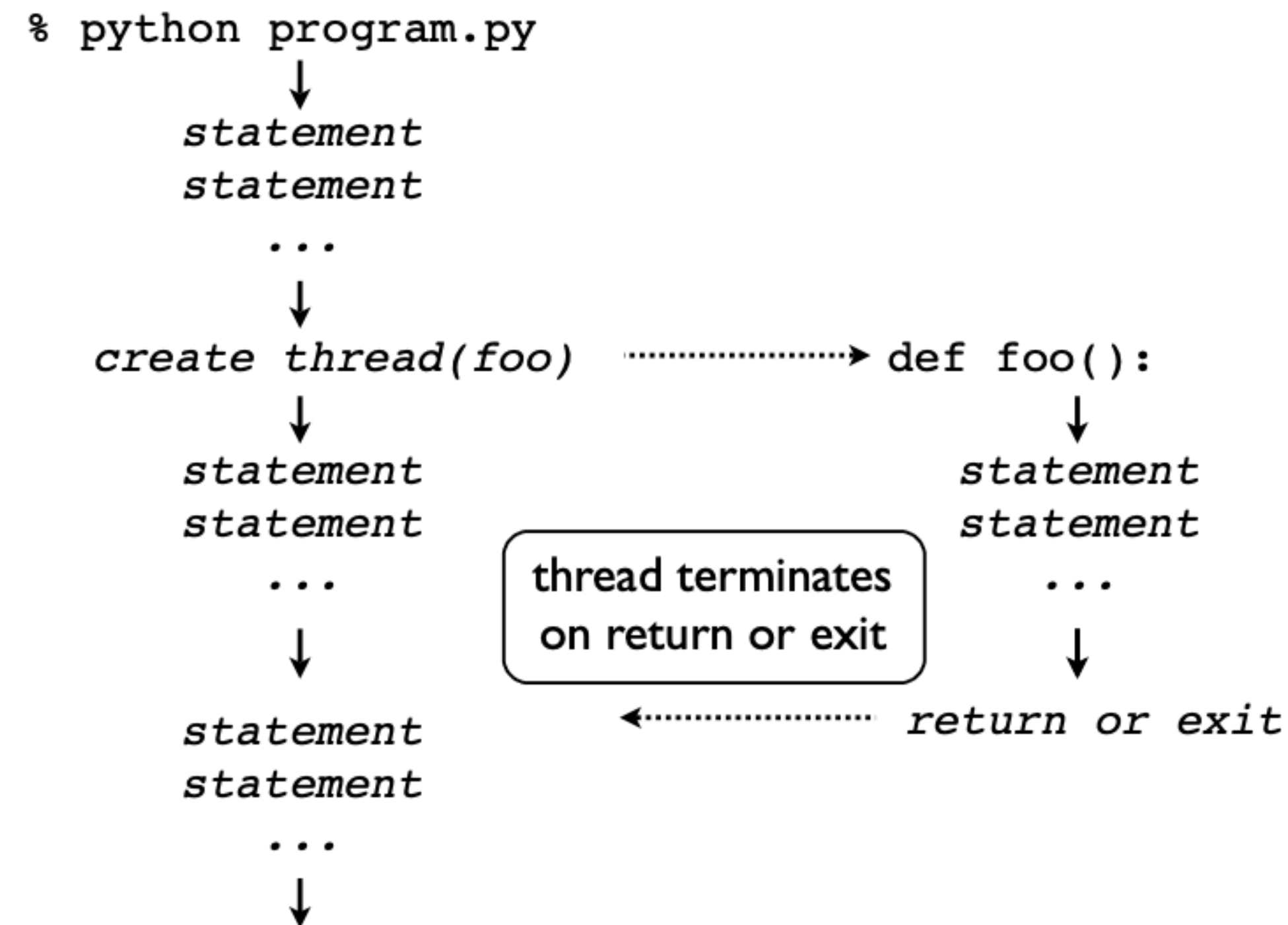
Python Thread Programming

Thread Basics



Python Thread Programming

Thread Basics



Python Thread Programming

Thread Basics

`% python program.py`

`statement`
`statement`

`...`

`create thread(foo)`

`statement`
`statement`

`...`

`statement`
`statement`

`...`



Key idea: Thread is like a little "task" that independently runs inside your program

thread

`def foo():`
↓
`statement`
`statement`
`...`
↓
`return or exit`

Python Thread Programming

threading module

- Python threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self, count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

- You inherit from Thread and redefine run()

Python Thread Programming


threading module

- Python threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self, count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

This code
executes in
the thread



- You inherit from Thread and redefine run()

Python Thread Programming

threading module

- To launch, create thread objects and call start()

```
t1 = CountdownThread(10) # Create the thread object  
t1.start()               # Launch the thread
```

```
t2 = CountdownThread(20) # Create another thread  
t2.start()               # Launch
```

- Threads execute until the run() method stops

Python Thread Programming

Functions as threads

- Alternative method of launching threads

```
def countdown(count):  
    while count > 0:  
        print "Counting down", count  
        count -= 1  
        time.sleep(5)  
  
t1 = threading.Thread(target=countdown, args=(10,))  
t1.start()
```

- Creates a Thread object, but its run() method just calls the given function

Python Thread Programming

Joining a Thread

- Once you start a thread, it runs independently
- Use `t.join()` to wait for a thread to exit

```
t.start()           # Launch a thread
...
# Do other work
...
# Wait for thread to finish
t.join()            # Waits for thread t to exit
```

- This only works from *other* threads
- A thread can't join itself

Python Thread Programming

Daemonic Threads

- If a thread runs forever, make it "daemonic"

```
t.daemon = True  
t.setDaemon(True)
```

- If you don't do this, the interpreter will lock when the main thread exits---waiting for the thread to terminate (which never happens)
- Normally you use this for background tasks

The End