Heterogeneous Computing for AI - Lecture ~04

Parallel Programming using Multiple Processors 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 Inside Story on Python Threads
- Processes and Messages
- Multiprocessing Module
- Multiprocessing Module
  - Pipes
  - Queues
  - Process Pools

Learning goals for today

Theoretical

- Gain knowledge about various multi-process parallel programming primitives in Python
- Understand the foundations of Process Pools, Queues, Pipes, and others

Practical

- Be able to write programs that can run on multicore processors/CPUs
- Understand how to pass messages between different processes

The Inside Story on Python Threads

Python and GIL Lock

The Infamous GIL

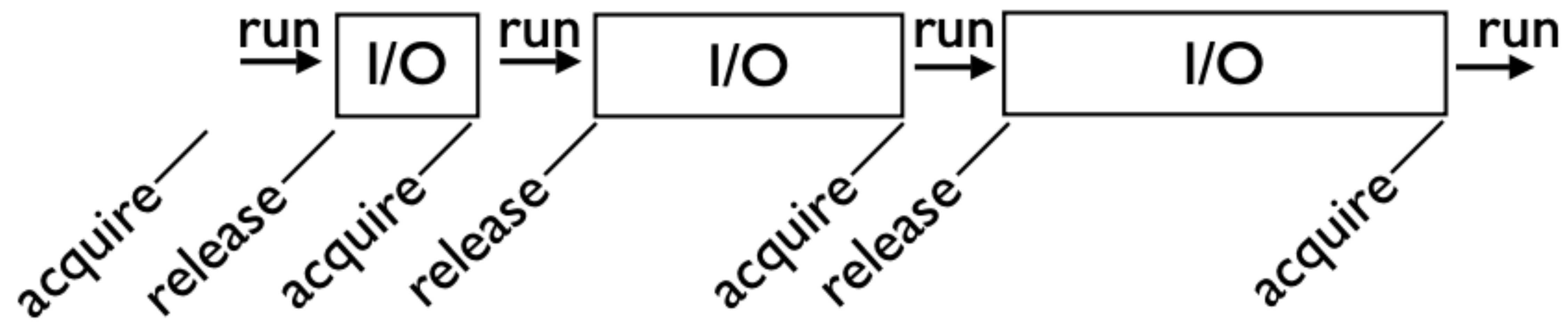
- Here's the rub...
- Only one Python thread can execute in the interpreter at once
- There is a "global interpreter lock" that carefully controls thread execution
- The GIL ensures that sure each thread gets exclusive access to the entire interpreter internals when it's running

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock

GIL Behavior

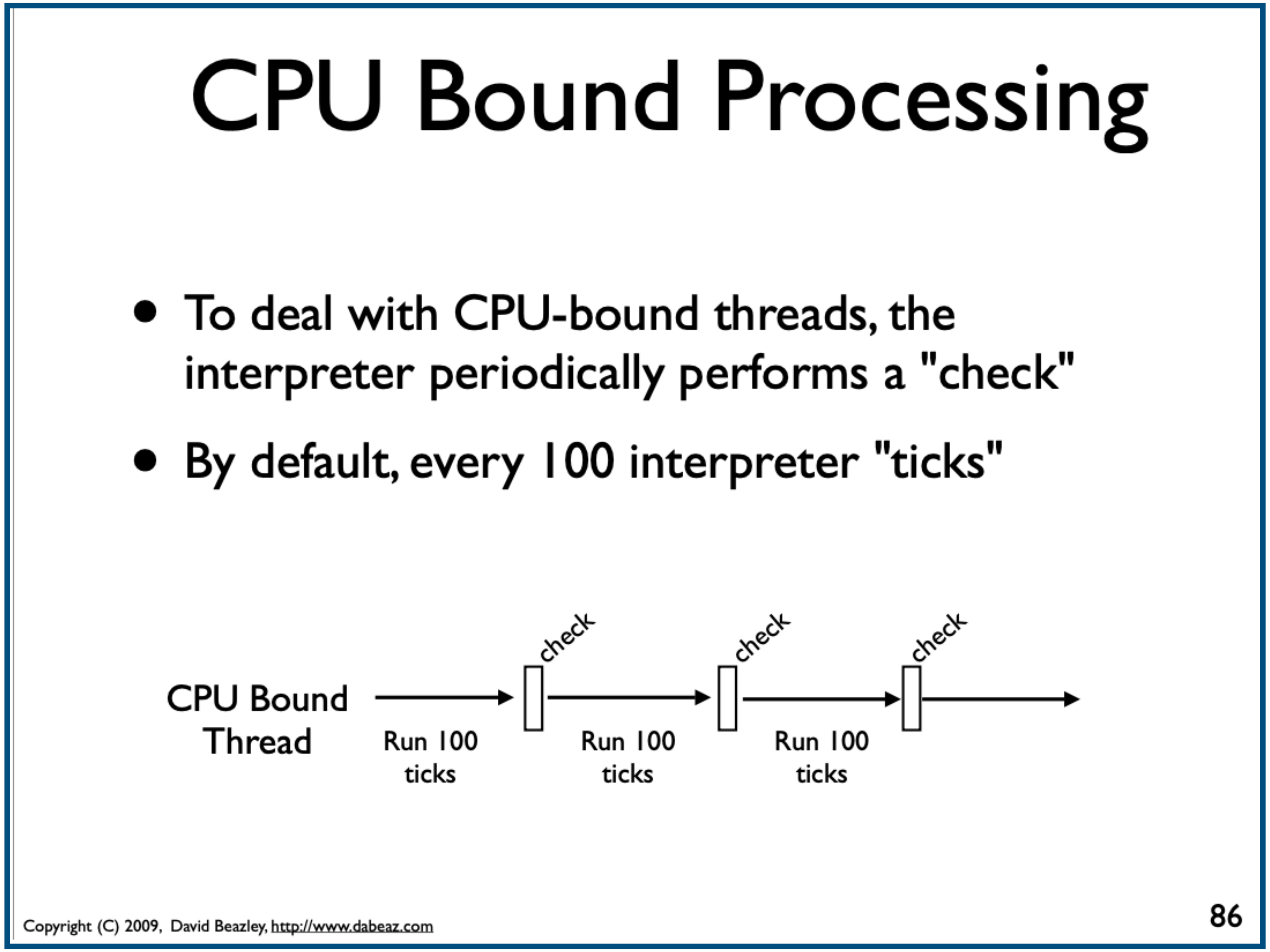
- Whenever a thread runs, it holds the GIL
- However, the GIL is released on blocking I/O



- So, any time a thread is forced to wait, other "ready" threads get their chance to run
- Basically a kind of "cooperative" multitasking

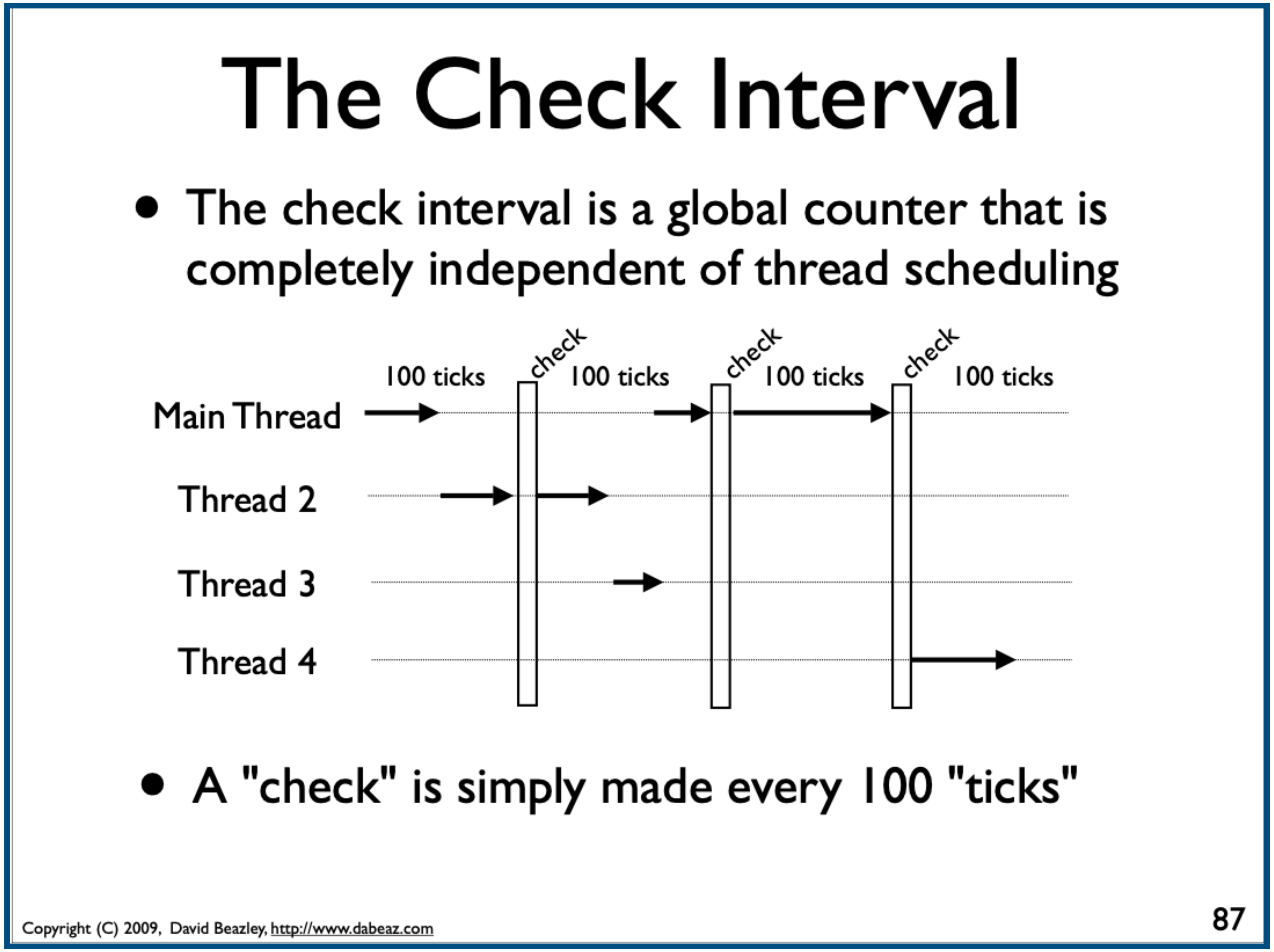
Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock



Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock



Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock

The Periodic Check

- What happens during the periodic check?
  - In the main thread only, signal handlers will execute if there are any pending signals
  - Release and reacquisition of the GIL
- That last bullet describes how multiple CPU-bound threads get to run (by briefly releasing the GIL, other threads get a chance to run).

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock

Thread Scheduling

- Python does not have a thread scheduler
- There is no notion of thread priorities, preemption, round-robin scheduling, etc.
- For example, the list of threads in the interpreter isn't used for anything related to thread execution
- All thread scheduling is left to the host operating system (e.g., Linux, Windows, etc.)

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

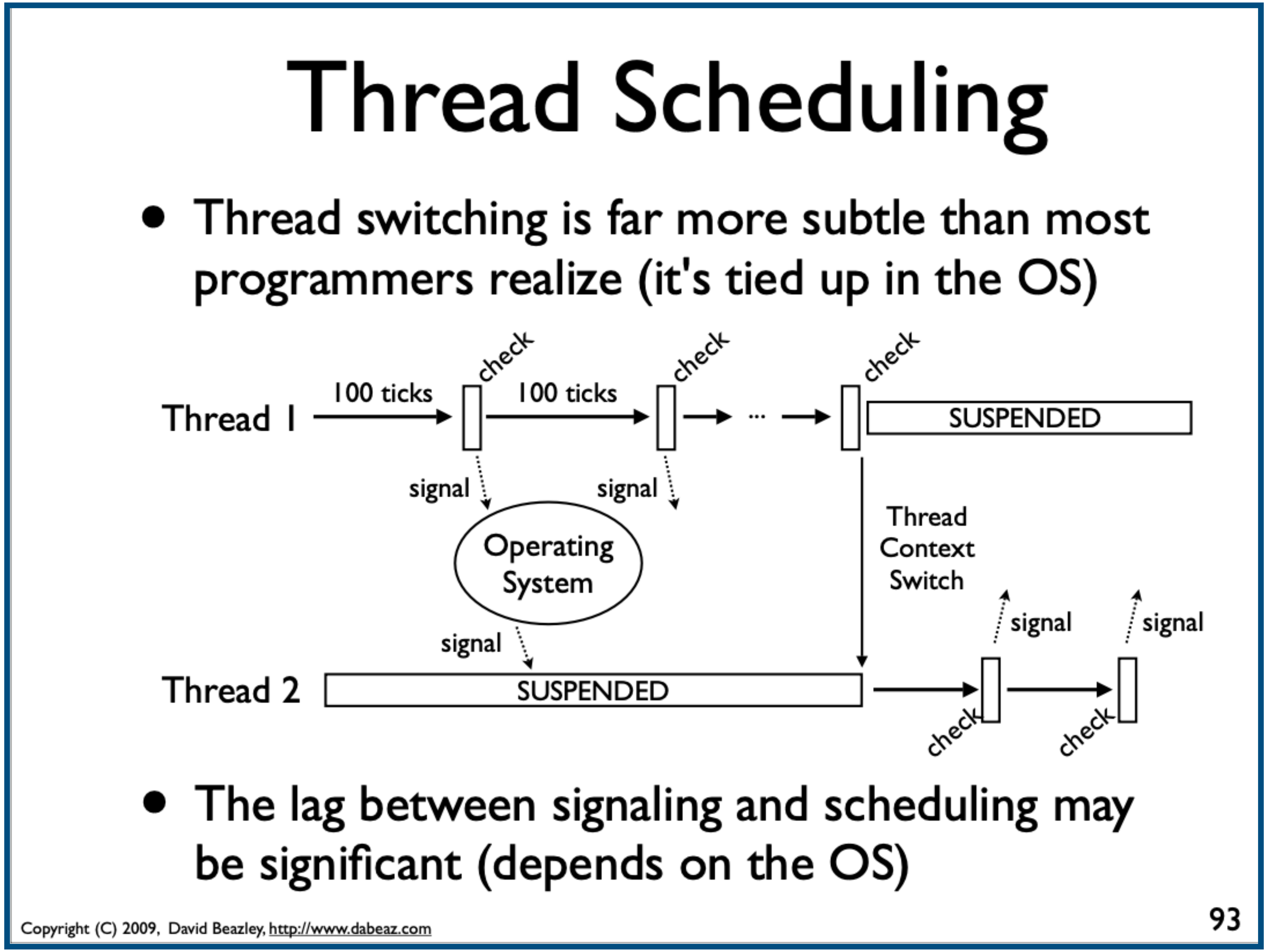
Python and GIL Lock

GIL Implementation

- The GIL is not a simple mutex lock
- The implementation (Unix) is either...
  - A POSIX unnamed semaphore
  - Or a pthreads condition variable
- All interpreter locking is based on signaling
  - To acquire the GIL, check if it's free. If not, go to sleep and wait for a signal
  - To release the GIL, free it and signal

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock



Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock

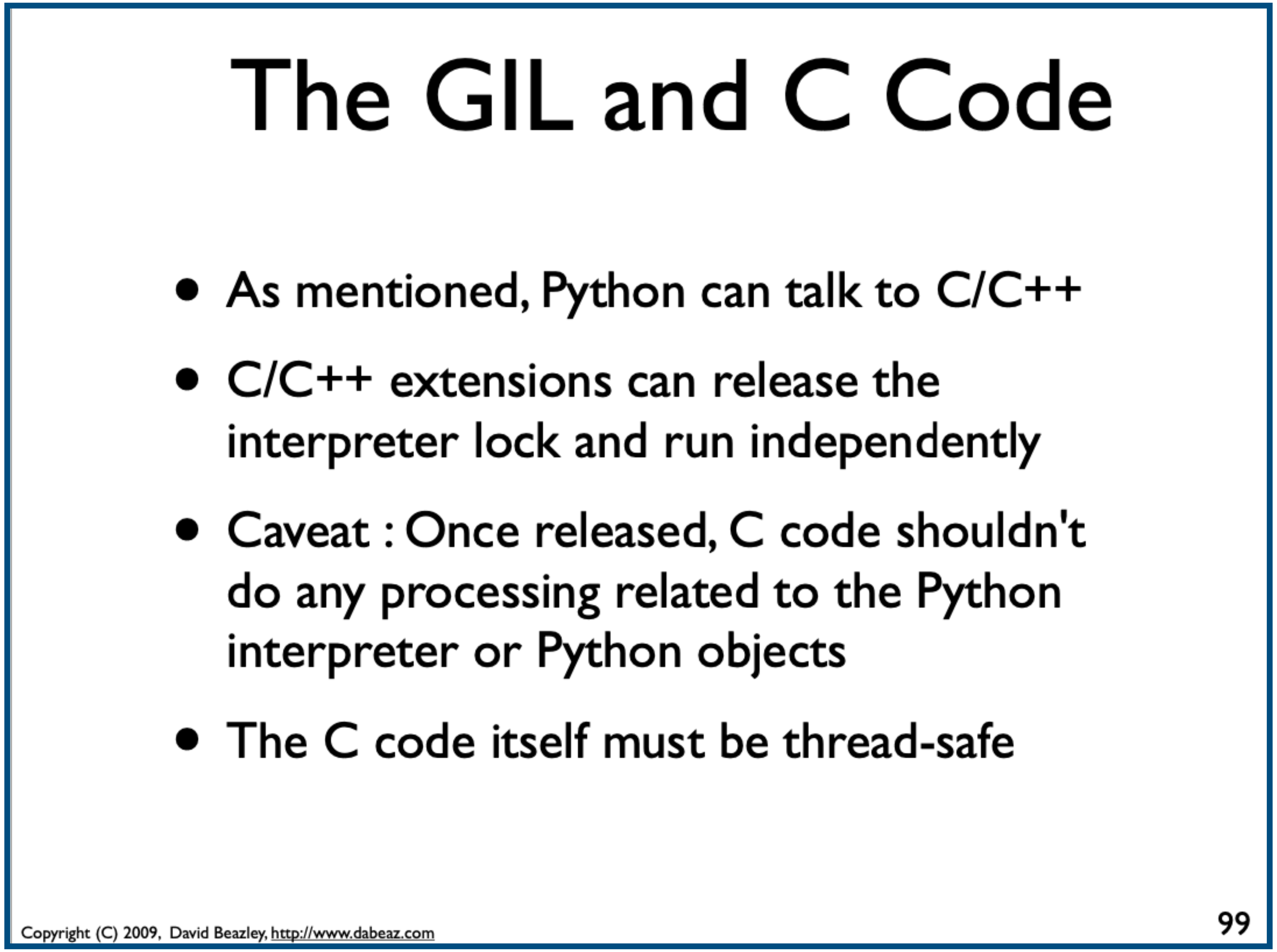
CPU-Bound Threads

- As we saw earlier, CPU-bound threads have <u>horrible</u> performance properties
- Far worse than simple sequential execution
  - 24.6 seconds (sequential)
  - 45.5 seconds (2 threads)
- A big question : Why?
  - What is the source of that overhead?

example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock



Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Python and GIL Lock

Why is the GIL there?

- Simplifies the implementation of the Python interpreter (okay, sort of a lame excuse)
- Better suited for reference counting (Python's memory management scheme)
- Simplifies the use of C/C++ extensions. Extension functions do not need to worry about thread synchronization
- And for now, it's here to stay... (although people continue to try and eliminate it)

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

Processes and Messages

Concept: Message Passing

- An alternative to threads is to run multiple independent copies of the Python interpreter
- In separate processes
- Possibly on different machines
- Get the different interpreters to cooperate by having them send messages to each other

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

Message Passing

- On the surface, it's simple
- Each instance of Python is independent
- Programs just send and receive messages
- Two main issues
  - What is a message?
  - What is the transport mechanism?

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

Messages

- A message is just a bunch of bytes (a buffer)
- A "serialized" representation of some data
- Creating serialized data in Python is easy

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

pickle Module

- A module for serializing objects
- Serializing an object onto a "file"

```
import pickle
...
pickle.dump(someobj,f)
```

- Unserializing an object from a file

```
someobj = pickle.load(f)
```

- Here, a file might be a file, a pipe, a wrapper around a socket, etc.

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

Pickle Commentary

- Using pickle is almost too easy
- Almost any Python object works
    - Builtins (lists, dicts, tuples, etc.)
    - Instances of user-defined classes
    - Recursive data structures
- Exceptions
    - Files and network connections
    - Running generators, etc.

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

Message Transport

- Python has various low-level mechanisms
  - Pipes
  - Sockets
  - FIFOs
- Libraries provide access to other systems
  - MPI
  - XML-RPC (and many others)

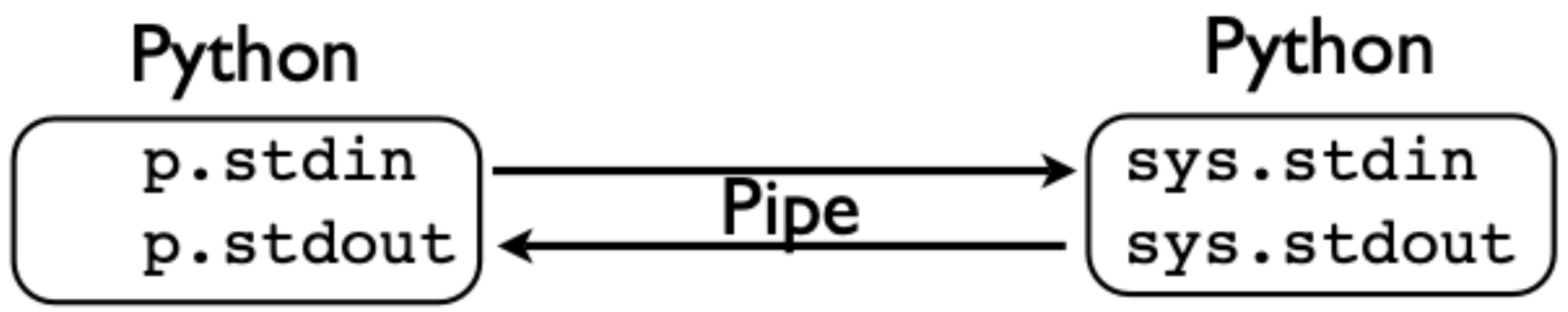
Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

An Example

- Launching a subprocess and hooking up the child process via a pipe
- Use the subprocess module

```
import subprocess

p = subprocess.Popen(['python','child.py'],
                     stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE)

p.stdin.write(data)     # Send data to subprocess
p.stdout.read(size)     # Read data from subprocess
```



Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

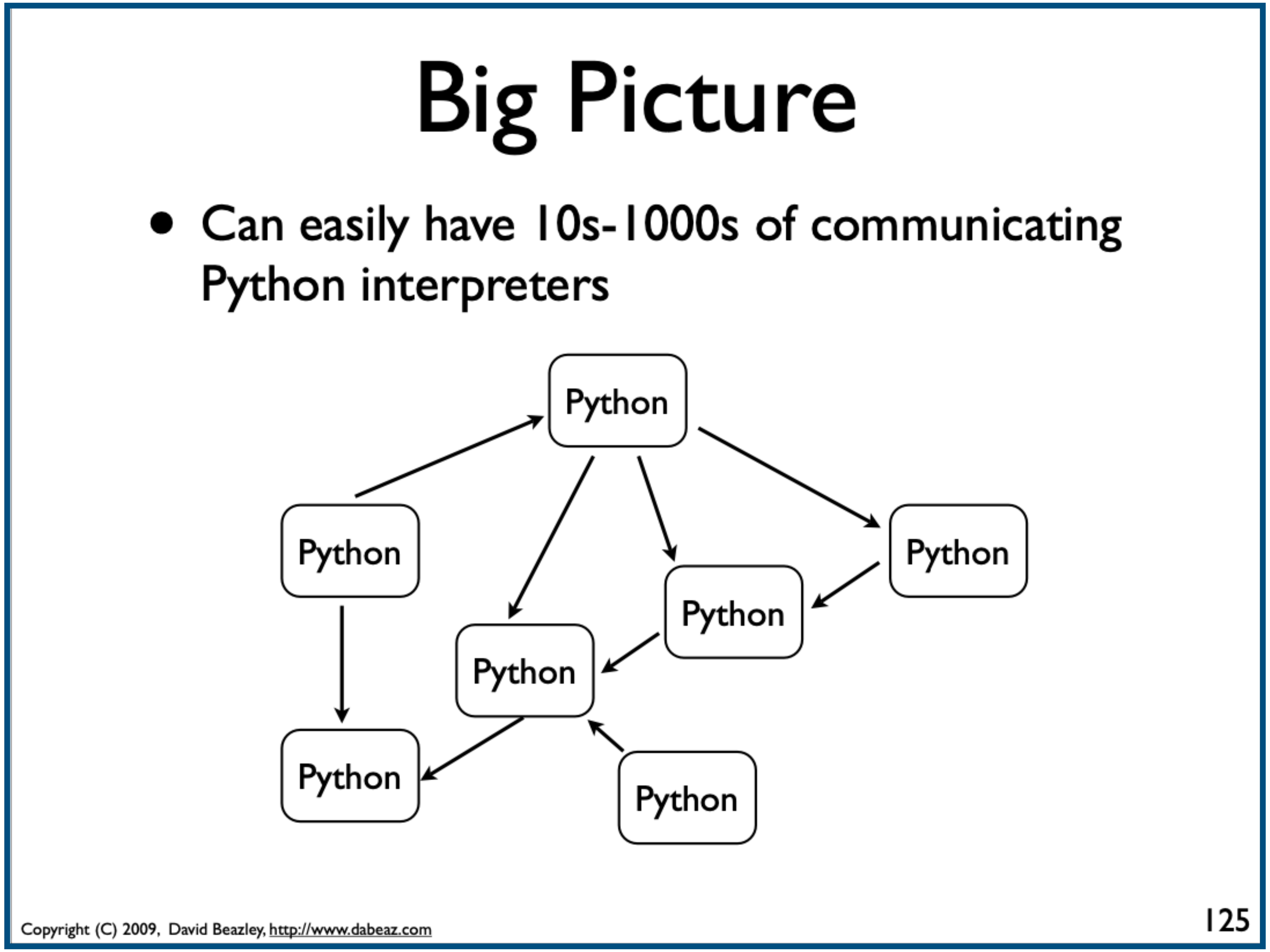
Processes and Messages

Pipes and Pickle

- Most programmers would use the subprocess module to run separate programs and collect their output (e.g., system commands)
- However, if you put a pickling layer around the files, it becomes much more interesting
- Becomes a communication channel where you can send just about any Python object

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages



Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Processes and Messages

Interlude

- Message passing is a fairly general concept
- However, it's also kind of nebulous in Python
- No agreed upon programming interface
- Vast number of implementation options
- Intersects with distributed objects, RPC, cross-language messaging, etc.

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

Multiprocessing Module

multiprocessing Module

- A new library module added in Python 2.6
- Originally known as pyprocessing (a third-party extension module)
- This is a module for writing concurrent Python programs based on communicating processes
- A module that is especially useful for concurrent CPU-bound processing

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

Using multiprocessing

- Here's the cool part...
- You already know how to use multiprocessing
- At a very high-level, it simply mirrors the thread programming interface
- Instead of "Thread" objects, you now work with "Process" objects.

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

multiprocessing Example

- Define tasks using a Process class

```
import time
import multiprocessing

class CountdownProcess(multiprocessing.Process):
    def __init__(self,count):
        multiprocessing. Process.__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 Process and redefine run()

example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

Launching Processes

- To launch, same idea as with threads

```
if __name__ == '__main__':
    p1 = CountdownProcess(10)  # Create the process object
    p1.start()                 # Launch the process

    p2 = CountdownProcess(20)   # Create another process
    p2.start()                  # Launch
```

- Processes execute until run() stops
- A critical detail : Always launch in main as shown (required for Windows)

example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

Functions as Processes

- Alternative method of launching processes

```
def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

if __name__ == '__main__':
    p1 = multiprocessing.Process(target=countdown,
                                 args=(10,))
p1.start()
```

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

example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

Does it Work?

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential Execution:

```
count(100000000)
count(100000000)
```

→ 24.6s

- Multiprocessing Execution

```
p1 = Process(target=count,args=(100000000,))
p1.start()
p2 = Process(target=count,args=(100000000,))
p2.start()
```

→ 12.5s

- Yes, it seems to work

example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

Other Process Features

- Joining a process (waits for termination)

```
p = Process(target=somefunc)
p.start()
...
p.join()
```

- Making a daemonic process

```
p = Process(target=somefunc)
p.daemon = True
p.start()
```

- Terminating a process

```
p = Process(target=somefunc)
...
p.terminate()
```

- These mirror similar thread functions

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module

Distributed Memory

- With multiprocessing, there are no shared data structures
- Every process is completely isolated
- Since there are no shared structures, forget about all of that locking business
- Everything is focused on messaging

Some are implemented in later versions of Python e.g. > 3.0

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module - Pipes

Pipes in Multiprocessing Module

Pipes

- A channel for sending/receiving objects

```
(c1, c2) = multiprocessing.Pipe()
```

- Returns a pair of connection objects (one for each end-point of the pipe)
- Here are methods for communication

```
c.send(obj)              # Send an object
c.recv()                 # Receive an object

c.send_bytes(buffer)     # Send a buffer of bytes
c.recv_bytes([max])      # Receive a buffer of bytes

c.poll([timeout])        # Check for data
```


Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Pipes in Multiprocessing Module

Using Pipes

- The Pipe() function largely mimics the behavior of Unix pipes
- However, it operates at a higher level
- It's not a low-level byte stream
- You send discrete messages which are either Python objects (pickled) or buffers

137

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Pipes in Multiprocessing Module

Pipe Example

- A simple data consumer

```
def consumer(p1, p2):
    p1.close()   # Close producer's end (not used)
    while True:
        try:
            item = p2.recv()
        except EOFError:
            break
        print item      # Do other useful work here
```

- A simple data producer

```
def producer(sequence, output_p):
    for item in sequence:
        output_p.send(item)
```


example in
Python file
in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Pipes in Multiprocessing Module

Pipe Example

```
if __name__ == '__main__':
    p1, p2 = multiprocessing.Pipe()

    cons = multiprocessing.Process(
               target=consumer,
               args=(p1,p2))
    cons.start()

    # Close the input end in the producer
    p2.close()

    # Go produce some data
    sequence = xrange(100)  # Replace with useful data
    producer(sequence, p1)

    # Close the pipe
    p1.close()
```


example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module - Queues

Queue in Multiprocessing Module

Message Queues

- multiprocessing also provides a queue
- The programming interface is the same

```
from multiprocessing import Queue

q = Queue()
q.put(item)        # Put an item on the queue
item = q.get()     # Get an item from the queue
```

- There is also a joinable Queue

```
from multiprocessing import JoinableQueue

q = JoinableQueue()
q.task_done()      # Signal task completion
q.join()           # Wait for completion
```


Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Queue in Multiprocessing Module

Queue Implementation

- Queues are implemented on top of pipes
- A subtle feature of queues is that they have a "feeder thread" behind the scenes
- Putting an item on a queue returns immediately (allowing the producer to keep working)
- The feeder thread works on its own to transmit data to consumers

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Queue in Multiprocessing Module

Queue Example

- A consumer process

```
def consumer(input_q):
    while True:
        # Get an item from the queue
        item = input_q.get()
        # Process item
        print item
        # Signal completion
        input_q.task_done()
```

- A producer process

```
def producer(sequence,output_q):
    for item in sequence:
        # Put the item on the queue
        output_q.put(item)
```


example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Queue in Multiprocessing Module

Queue Example

- Running the two processes

```
if __name__ == '__main__':
    from multiprocessing import Process, JoinableQueue
    q = JoinableQueue()

    # Launch the consumer process
    cons_p = Process(target=consumer,args=(q,))
    cons_p.daemon = True
    cons_p.start()

    # Run the producer function on some data
    sequence = range(100)    # Replace with useful data
    producer(sequence,q)

    # Wait for the consumer to finish
    q.join()
```


example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Queue in Multiprocessing Module

Commentary

- If you have written threaded programs that strictly stick to the queuing model, they can probably be ported to multiprocessing
- The following restrictions apply
  - Only objects compatible with pickle can be queued
  - Tasks can not rely on any shared data other than a reference to the queue

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Queue in Multiprocessing Module

Other Features

- multiprocessing has many other features
  - Process Pools
  - Shared objects and arrays
  - Synchronization primitives
  - Managed objects
  - Connections
- Will briefly look at one of them

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Multiprocessing Module - Process Pools

Process Pools in Multiprocessing Module

Process Pools

- Creating a process pool

```
p = multiprocessing.Pool([numprocesses])
```

- Pools provide a high-level interface for executing functions in worker processes
- Let's look at an example...

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Process Pools in Multiprocessing Module

Pool Example

- Define a function that does some work
- Example : Compute a SHA-512 digest of a file

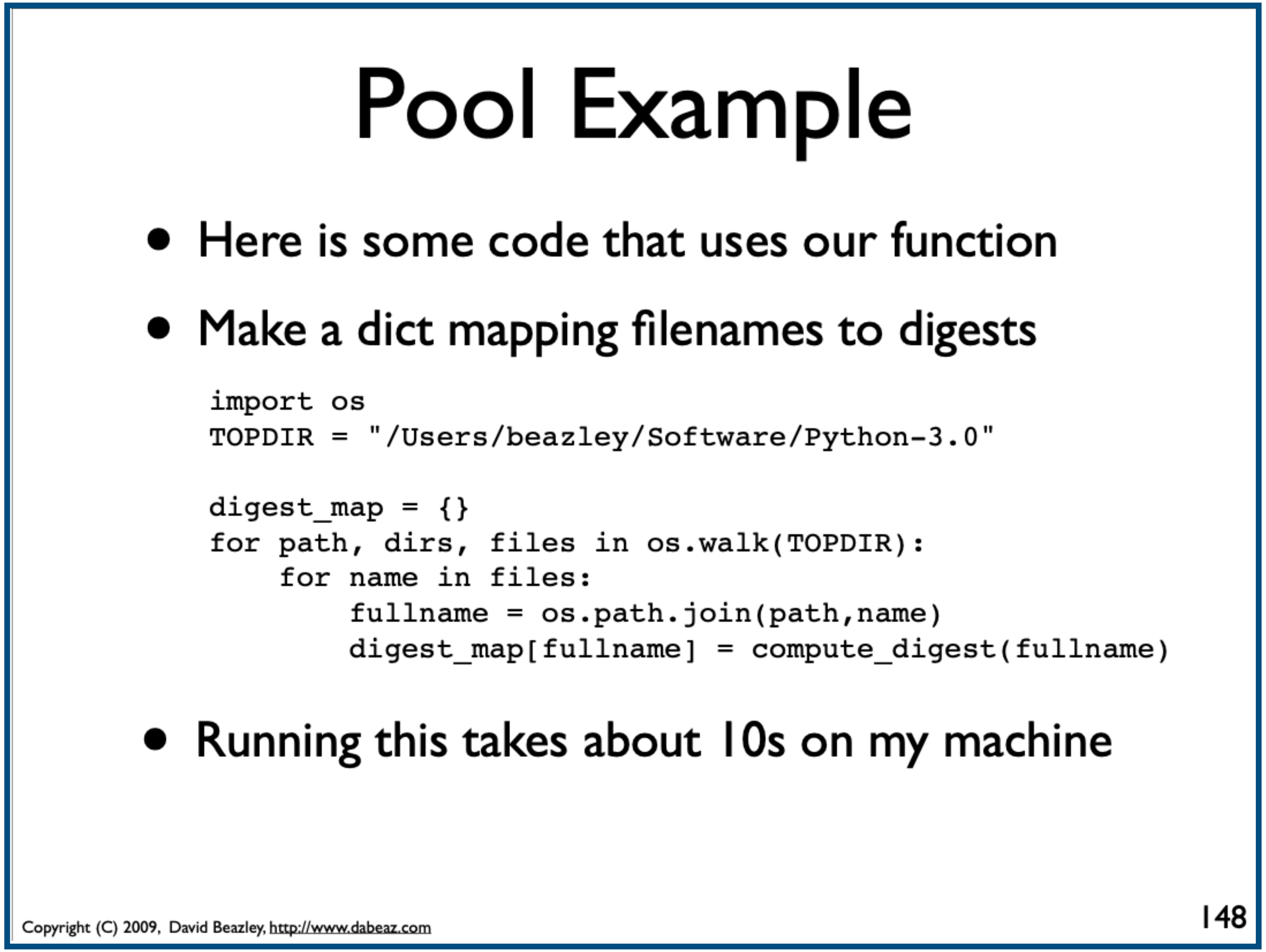
```
import hashlib

def compute_digest(filename):
    digest = hashlib.sha512()
    f = open(filename,'rb')
    while True:
        chunk = f.read(8192)
        if not chunk: break
        digest.update(chunk)
    f.close()
    return digest.digest()
```

- This is just a normal function (no magic)

example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

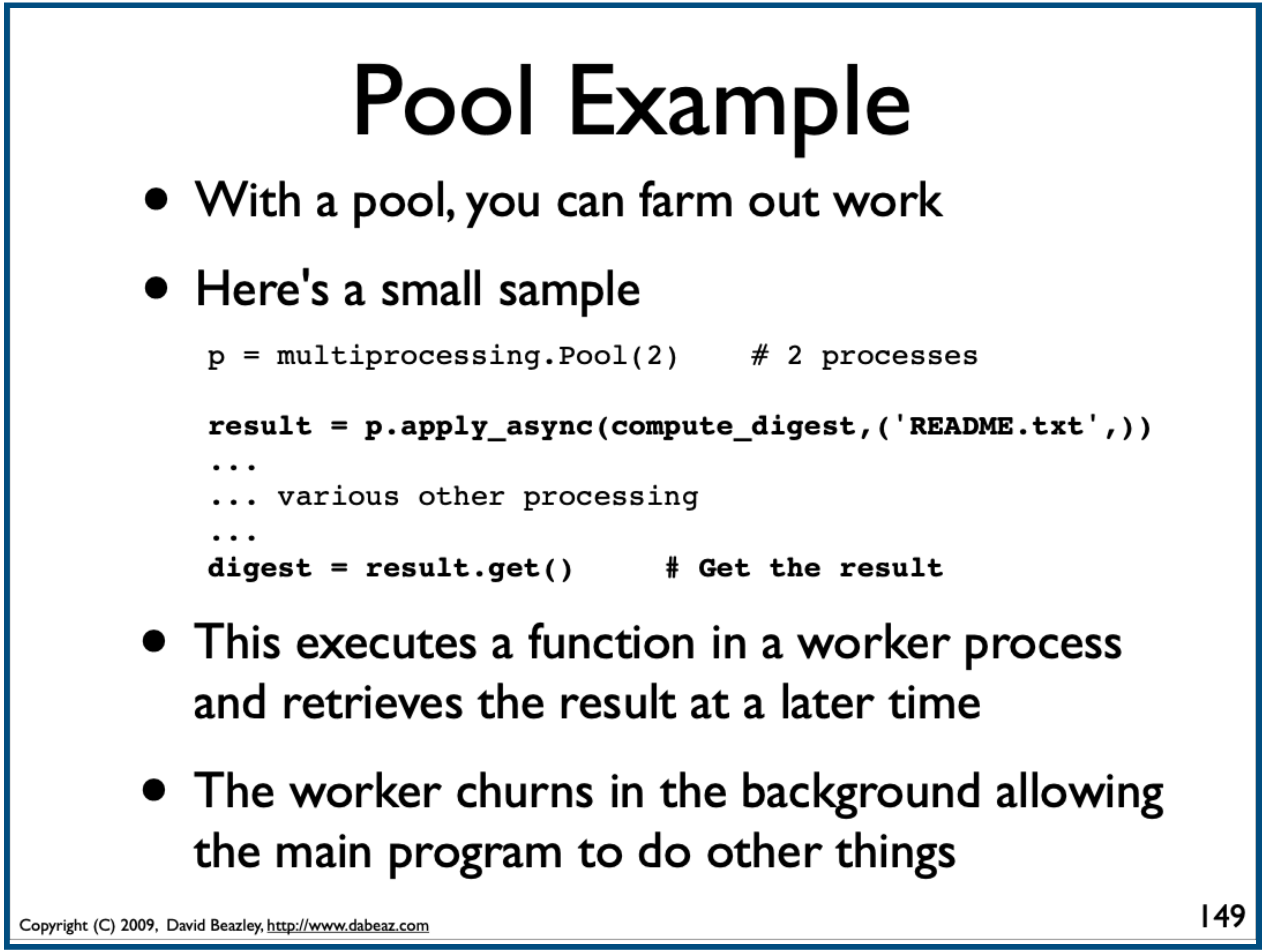
Process Pools in Multiprocessing Module



example in Python file in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Process Pools in Multiprocessing Module



Pool Example

- With a pool, you can farm out work
- Here's a small sample

```
p = multiprocessing.Pool(2)    # 2 processes

result = p.apply_async(compute_digest,('README.txt',))
...
... various other processing
...
digest = result.get()      # Get the result
```

- This executes a function in a worker process and retrieves the result at a later time
- The worker churns in the background allowing the main program to do other things

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

example in Python file in PyCharm

Process Pools in Multiprocessing Module

Pool Example

- Make a dictionary mapping names to digests

```
import multiprocessing
import os
TOPDIR = "/Users/beazley/Software/Python-3.0"

p = multiprocessing.Pool(2)      # Make a process pool
digest_map = {}
for path, dirs, files in os.walk(TOPDIR):
    for name in files:
        fullname = os.path.join(path,name)
        digest_map[fullname] = p.apply_async(
              compute_digest, (fullname,)
        )

# Go through the final dictionary and collect results
for filename, result in digest_map.items():
    digest_map[filename] = result.get()
```

- This runs in about 5.6 seconds

example in
Python file
in PyCharm

Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Resources

- **An Introduction to Python Concurrency**. http://www.dabeaz.com/usenix2009/concurrent/index.html
- **Slides on Concurrency** http://www.dabeaz.com/usenix2009/concurrent/Concurrent.pdf
- **Python Multiprocessing: The Complete Guide** https://superfastpython.com/multiprocessing-in-python/

The End