Heterogeneous Computing for AI - Lecture ~03

Advanced 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

- Recap - Threading
- Threading and Race Conditions
- Thread Synchronization Primitives
- Mutex Locks
- Semaphores
- Events and Condition Variables
- Thread-safe data structures: Queues

Learning goals for today

Theoretical

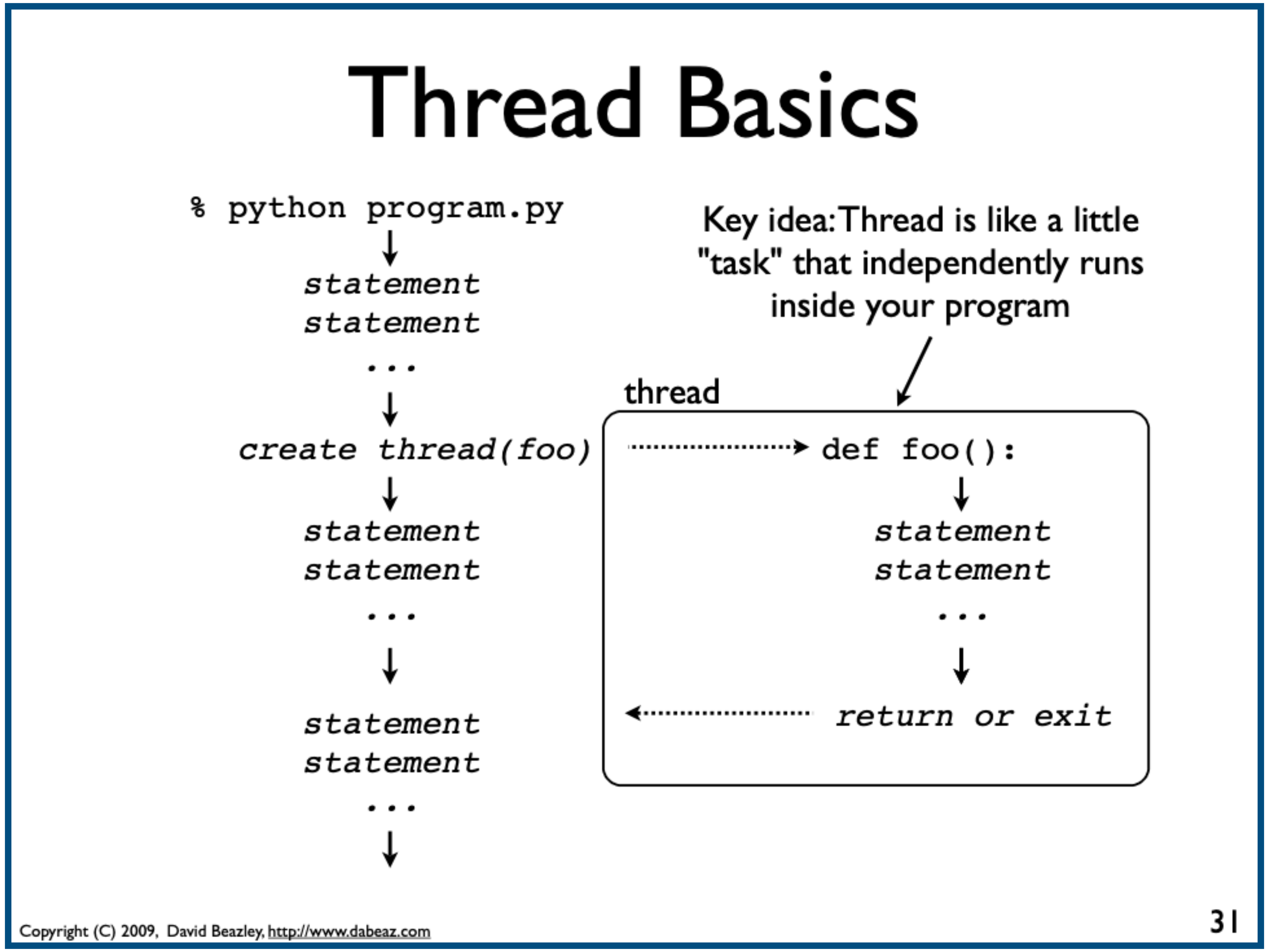
- Gain knowledge about various thread synchronization primitives in Python
- Learn the foundations of basic primitives for concurrency in Python
- Understand the foundations of advanced primitives for concurrency in Python

Practical

- Be able to program using basic thread synchronization primitives
- Understand how to use advanced thread synchronization primitives

Recap - Threading

Python Thread Programming



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

Python Thread Programming

Joining a Thread

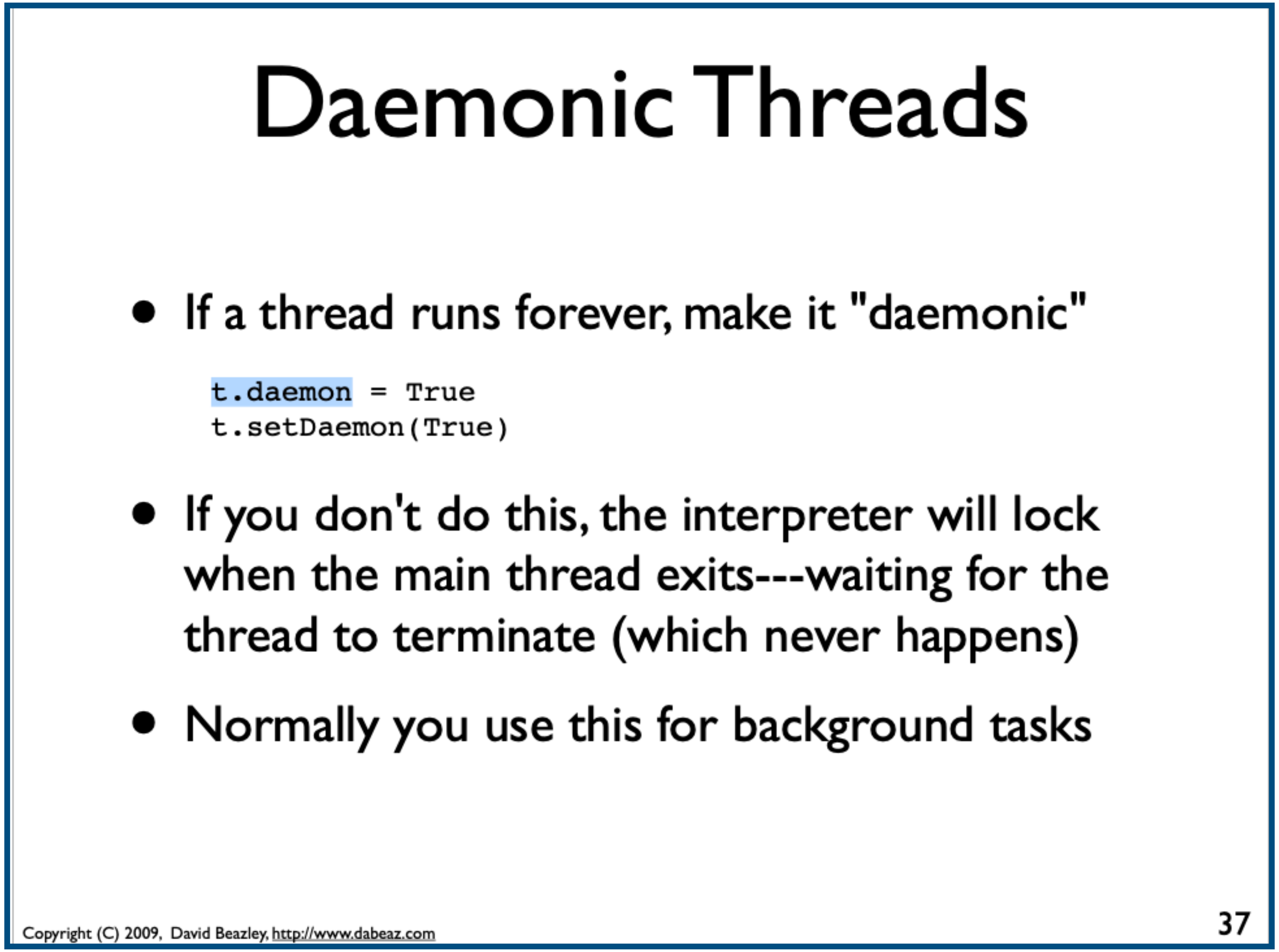
- Once you start a thread, it runs independently
- Use t.join() to wait for a thread to exit

```
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

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

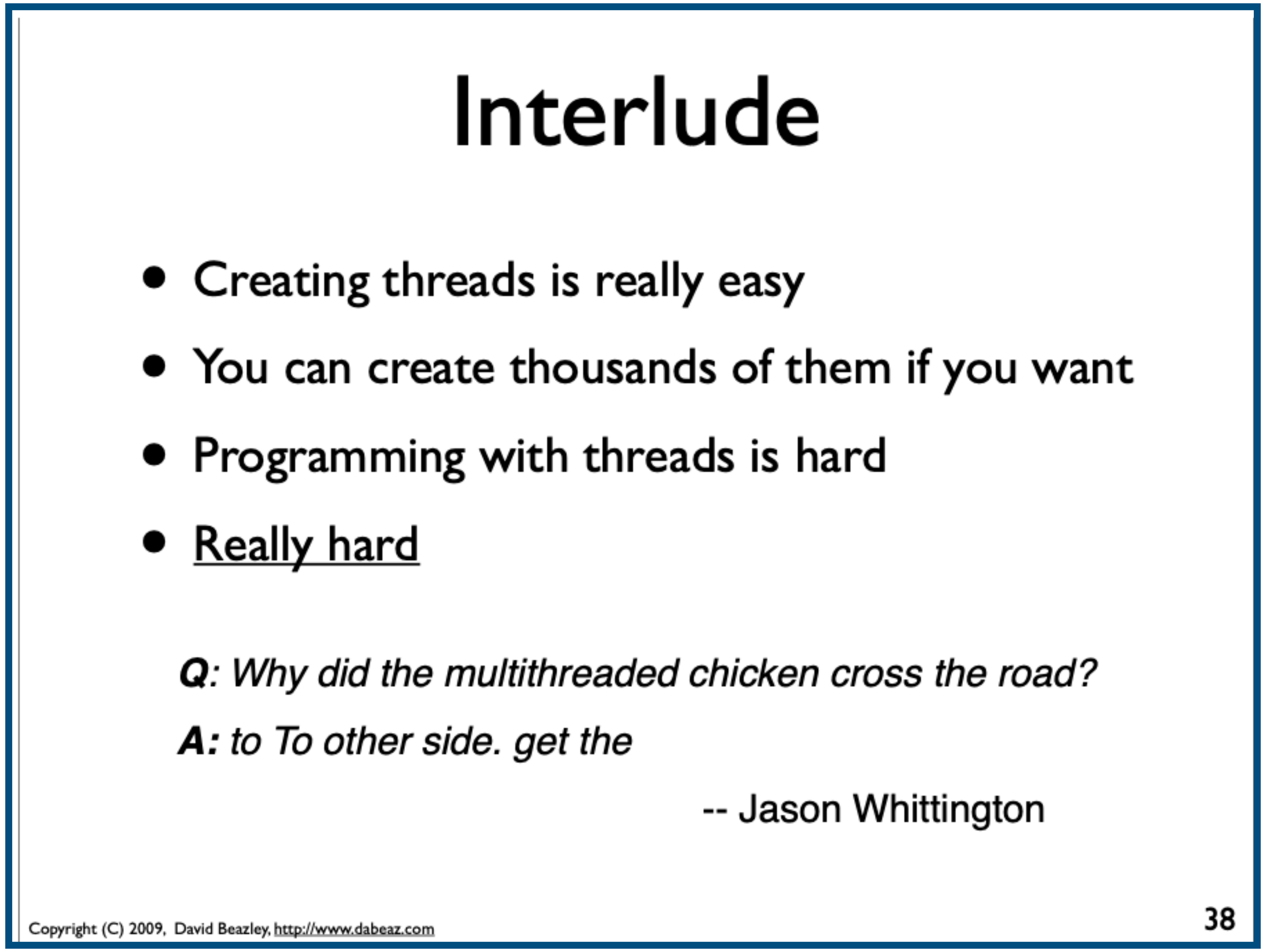
Python Thread Programming



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

Threading and Race Conditions

Threading and Race Conditions



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

Threading and Race Conditions

Access to Shared Data

- Threads share all of the data in your program
- Thread scheduling is non-deterministic
- Operations often take several steps and might be interrupted mid-stream (non-atomic)
- Thus, access to any kind of shared data is also non-deterministic (which is a really good way to have your head explode)

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

Threading and Race Conditions

Accessing Shared Data

- Consider a shared object

```
x = 0
```

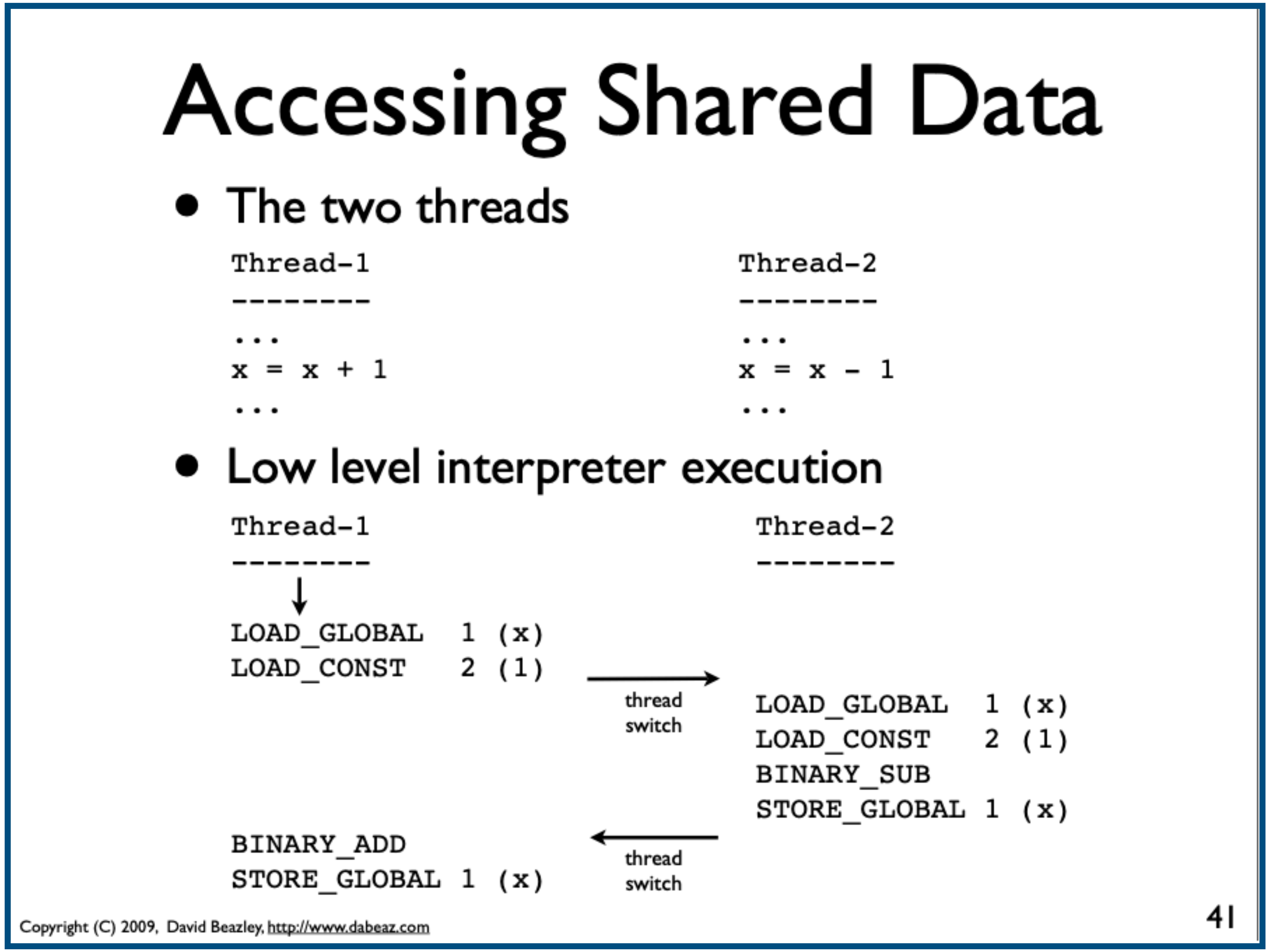
- And two threads that modify it

```
Thread-1                       Thread-2
--------                       --------
...                            ...
x = x + 1                      x = x - 1
...                            ...
```

- It's possible that the resulting value will be unpredictably corrupted

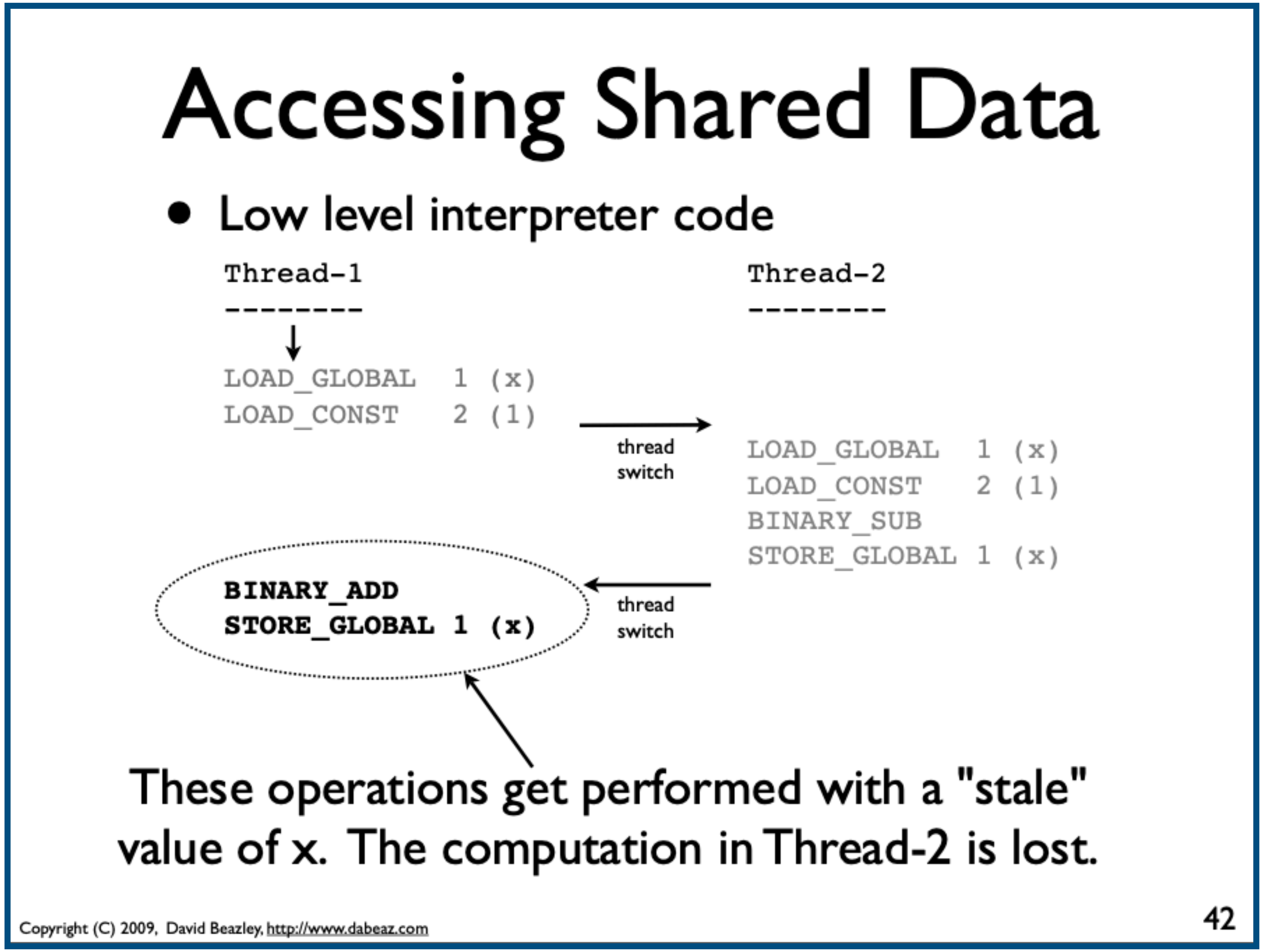
Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Threading and Race Conditions



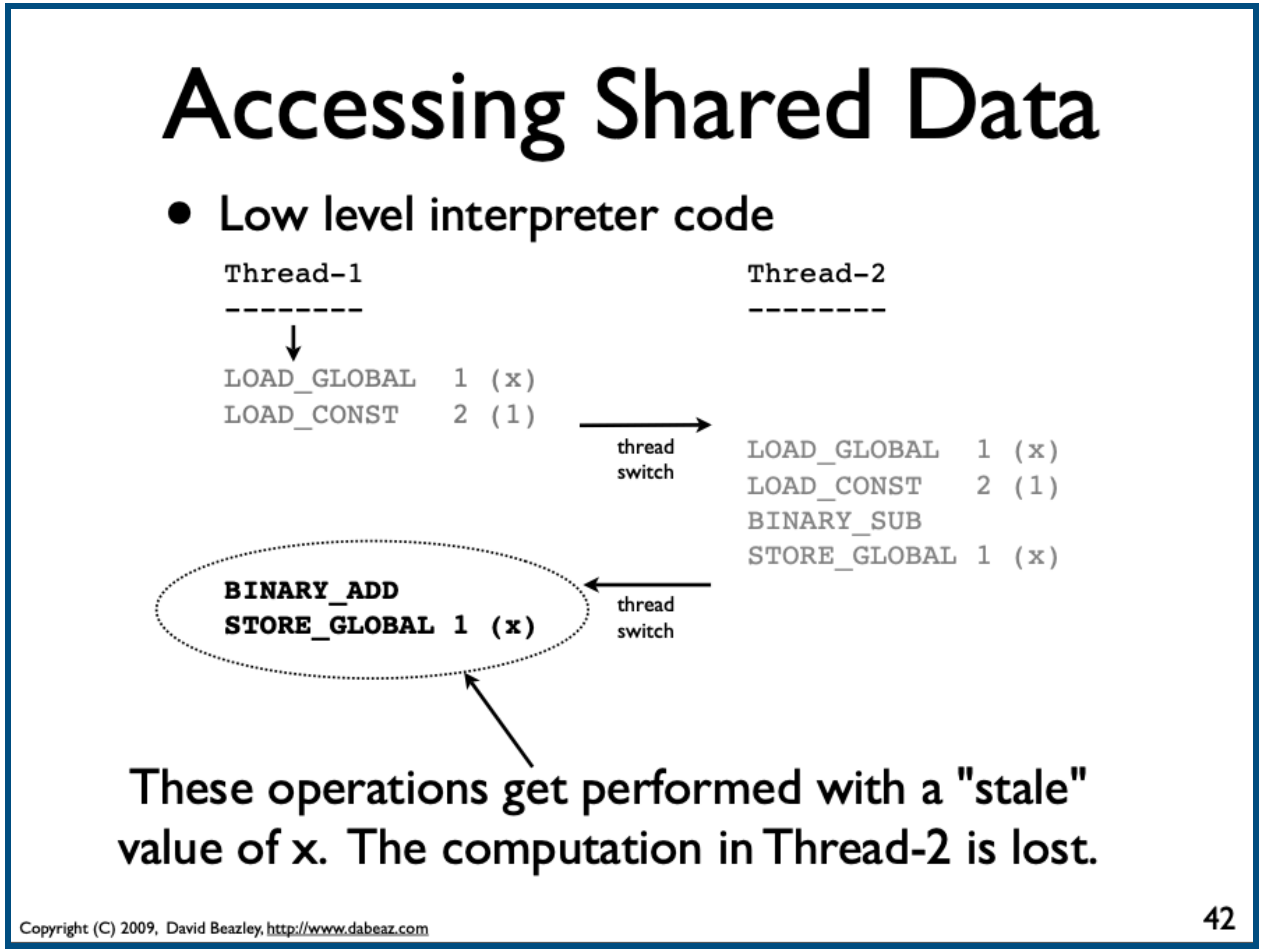
Source: An Introduction to Python Concurrency http://www.dabeaz.com/tutorials.html

Threading and Race Conditions



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

Threading and Race Conditions



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

Threading and Race Conditions

Accessing Shared Data

- Is this actually a real concern?

```
x = 0              # A shared value
def foo():
     global x
     for i in xrange(100000000): x += 1

def bar():
     global x
     for i in xrange(100000000): x -= 1

t1 = threading.Thread(target=foo)
t2 = threading.Thread(target=bar)
t1.start(); t2.start()
t1.join(); t2.join()    # Wait for completion
print x                 # Expected result is 0
```

- Yes, the print produces a random nonsensical value each time (e.g., -83412 or 1627732)

example in tutorial notebook

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

Threading and Race Conditions

Race Conditions

- The corruption of shared data due to thread scheduling is often known as a "race condition."
- It's often quite diabolical--a program may produce slightly different results each time it runs (even though you aren't using any random numbers)
- Or it may just flake out mysteriously once every two weeks

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

Threading and Race Conditions

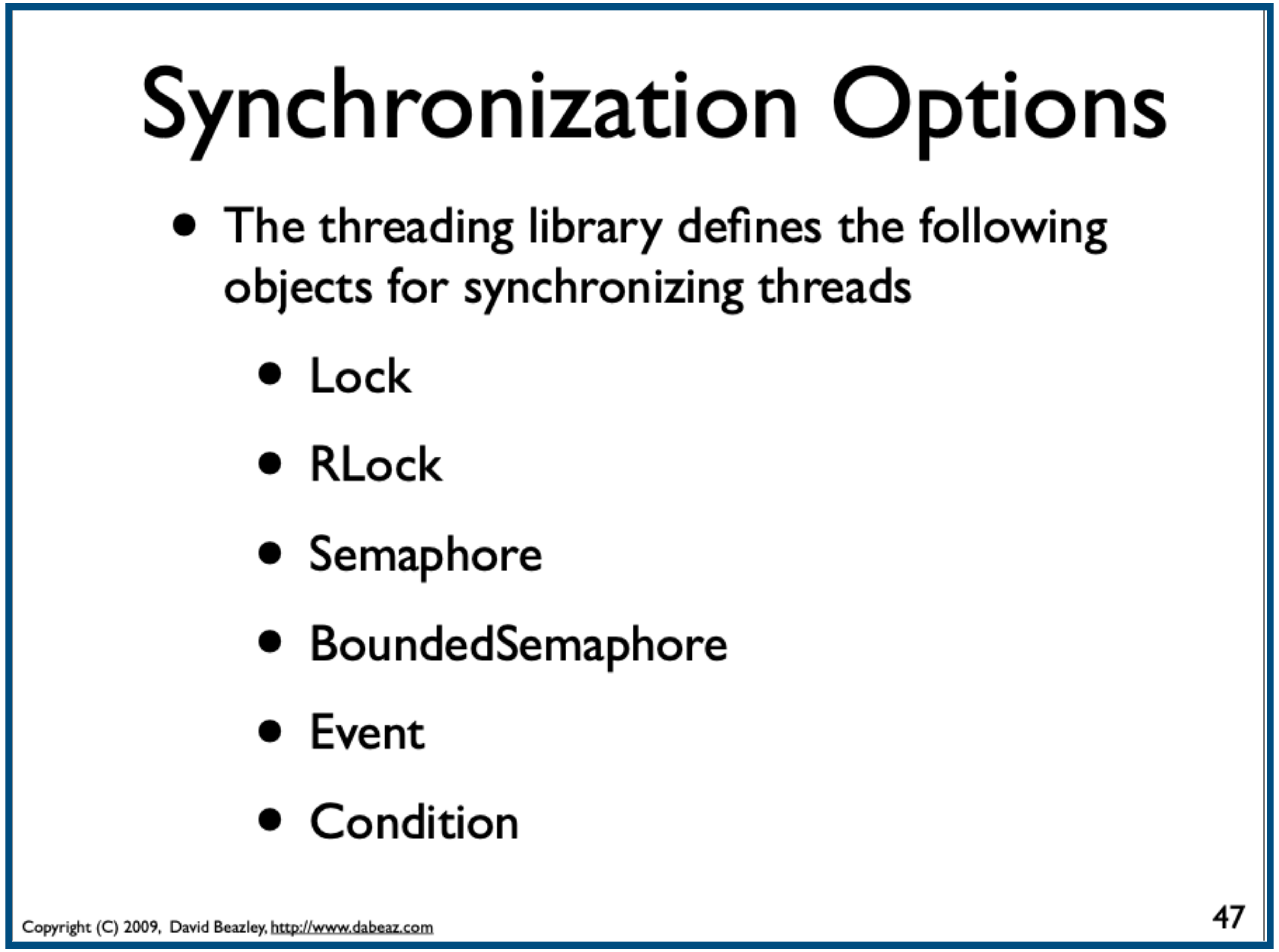
Thread Synchronization

- Identifying and fixing a race condition will make you a better programmer (e.g., it "builds character")
- However, you'll probably never get that month of your life back...
- To fix :You have to synchronize threads

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

Thread Synchronization Primitives

Threading and Race Conditions



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

Mutex Locks

Mutex Locks

Mutex Locks

- Mutual Exclusion Lock

```
m = threading.Lock()
```

- Probably the most commonly used synchronization primitive
- Primarily used to synchronize threads so that only one thread can make modifications to <u>shared data</u> at any given time

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

Mutex Locks

Mutex Locks

- There are two basic operations

```
m.acquire()                 # Acquire the lock
m.release()                 # Release the lock
```

- Only one thread can successfully acquire the lock at any given time
- If another thread tries to acquire the lock when its already in use, it gets blocked until the lock is released

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

Mutex Locks

Use of Mutex Locks

- Commonly used to enclose critical sections

```
x = 0
x_lock = threading.Lock()

Thread-1                        Thread-2
--------                        --------
...                             ...
x_lock.acquire()                x_lock.acquire()

x = x + 1                       x = x - 1

x_lock.release()                x_lock.release()
...                             ...
```

Critical Section

- Only one thread can execute in critical section at a time (lock gives exclusive access)

example in tutorial notebook

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

Mutex Locks

Using a Mutex Lock

- It is your responsibility to identify and lock all "critical sections"

```
x = 0
x_lock = threading.Lock()

Thread-1                         Thread-2
--------                         --------
...                              ...
x_lock.acquire()                 x = x - 1
x = x + 1                        ...
x_lock.release()
...
```

If you use a lock in one place, but not another, then you're missing the whole point. All modifications to shared state must be enclosed by lock acquire()/release().

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

Mutex Locks

Locking Perils

- Locking looks straightforward
- Until you start adding it to your code
- Managing locks is a lot harder than it looks

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

Mutex Locks

Lock Management

- Acquired locks must always be released
- However, it gets evil with exceptions and other non-linear forms of control-flow
- Always try to follow this prototype:

```
x = 0
x_lock = threading.Lock()

# Example critical section
x_lock.acquire()
try:
    statements using x
finally:
    x_lock.release()
```


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

Mutex Locks

Lock Management

- Python 2.6/3.0 has an improved mechanism for dealing with locks and critical sections

```
x = 0
x_lock = threading.Lock()

# Critical section
with x_lock:
    statements using x
...
```

- This automatically acquires the lock and releases it when control enters/exits the associated block of statements

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

Mutex Locks

Locks and Deadlock

- Don't write code that acquires more than one mutex lock at a time

```
x = 0
y = 0
x_lock = threading.Lock()
y_lock = threading.Lock()

with x_lock:
    statements using x
    ...
    with y_lock:
        statements using x and y
        ...
```

- This almost invariably ends up creating a program that mysteriously deadlocks (even more fun to debug than a race condition)

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

Semaphores

Semaphores

Semaphores

- A counter-based synchronization primitive

```
m = threading.Semaphore(n) # Create a semaphore
m.acquire()                # Acquire
m.release()                # Release
```

- acquire() - Waits if the count is 0, otherwise decrements the count and continues
- release() - Increments the count and signals waiting threads (if any)
- Unlike locks, acquire()/release() can be called in any order and by any thread

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

Semaphores

Semaphore Uses

- Resource control. You can limit the number of threads performing certain operations. For example, performing database queries, making network connections, etc.
- Signaling. Semaphores can be used to send "signals" between threads. For example, having one thread wake up another thread.

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

Semaphores

Resource Control

- Using a semaphore to limit resources

```
sema = threading.Semaphore(5)    # Max: 5-threads

def fetch_page(url):
    sema.acquire()
    try:
        u = urllib.urlopen(url)
        return u.read()
    finally:
        sema.release()
```

- In this example, only 5 threads can be executing the function at once (if there are more, they will have to wait)

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

Semaphores

Thread Signaling

- Using a semaphore to signal

```
done = threading.Semaphore(0)
```

Thread 1

```
...
statements
statements
statements
done.release()
```

Thread 2

```
done.acquire()
statements
statements
statements
...
```

- Here, acquire() and release() occur in <u>different</u> threads and in a different order
- Often used with producer-consumer problems

example in tutorial notebook

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

Events and Condition Variables

Events

Events

- Event Objects

```
e = threading.Event()
e.isSet()         # Return True if event set
e.set()           # Set event
e.clear()         # Clear event
e.wait()          # Wait for event
```

- This can be used to have one or more threads wait for something to occur
- Setting an event will unblock <u>all</u> waiting threads simultaneously (if any)
- Common use : barriers, notification

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

Events

Event Example

- Using an event to ensure proper initialization

```
init = threading.Event()

def worker():
    init.wait()      # Wait until initialized
    statements
    ...

def initialize():
    statements       # Setting up
    statements       # ...
    ...
    init.set()       # Done initializing

Thread(target=worker).start()   # Launch workers
Thread(target=worker).start()
Thread(target=worker).start()
initialize()                    # Initialize
```


Event Example

- Using an event to signal "completion"

```
def master():
    ...
    item = create_item()
    evt = Event()
    worker.send((item,evt))
    ...
    # Other processing
    ...
    ...
    ...
    ...
    ...
    # Wait for worker
    evt.wait()
```

Worker Thread

```
item, evt = get_work()
processing
processing
...
...
# Done
evt.set()
```

- Might use for asynchronous processing, etc.

example in tutorial notebook

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

Condition Variables

Condition Variables

- Condition Objects

```
cv = threading.Condition([lock])
cv.acquire()     # Acquire the underlying lock
cv.release()     # Release the underlying lock
cv.wait()        # Wait for condition
cv.notify()      # Signal that a condition holds
cv.notifyAll()   # Signal all threads waiting
```

- A combination of locking/signaling
- Lock is used to protect code that establishes some sort of "condition" (e.g., data available)
- Signal is used to notify other threads that a "condition" has changed state

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

Condition Variables

Condition Variables

- Common Use : Producer/Consumer patterns

```
items = []
items_cv = threading.Condition()
```

Producer Thread

```
item = produce_item()
with items_cv:
    items.append(item)
```

Consumer Thread

```
with items_cv:
    ...
    x = items.pop(0)

# Do something with x
...
```

- First, you use the locking part of a CV synchronize access to shared data (items)

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

Condition Variables

Condition Variables

- Common Use : Producer/Consumer patterns

```
items = []
items_cv = threading.Condition()
```

<u>Producer Thread</u>

```
item = produce_item()
with items_cv:
    items.append(item)
    items_cv.notify()
```

<u>Consumer Thread</u>

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)

# Do something with x
...
```

- Next you add signaling and waiting
- Here, the producer signals the consumer that it put data into the shared list

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

Condition Variables

Condition Variables

- Some tricky bits involving wait()
- Before waiting, you have to acquire the lock
- wait() releases the lock when waiting and reacquires when woken
- Conditions are often transient and may not hold by the time wait() returns. So, you must always double-check (hence, the while loop)

Consumer Thread

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)

# Do something with x
...
```


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

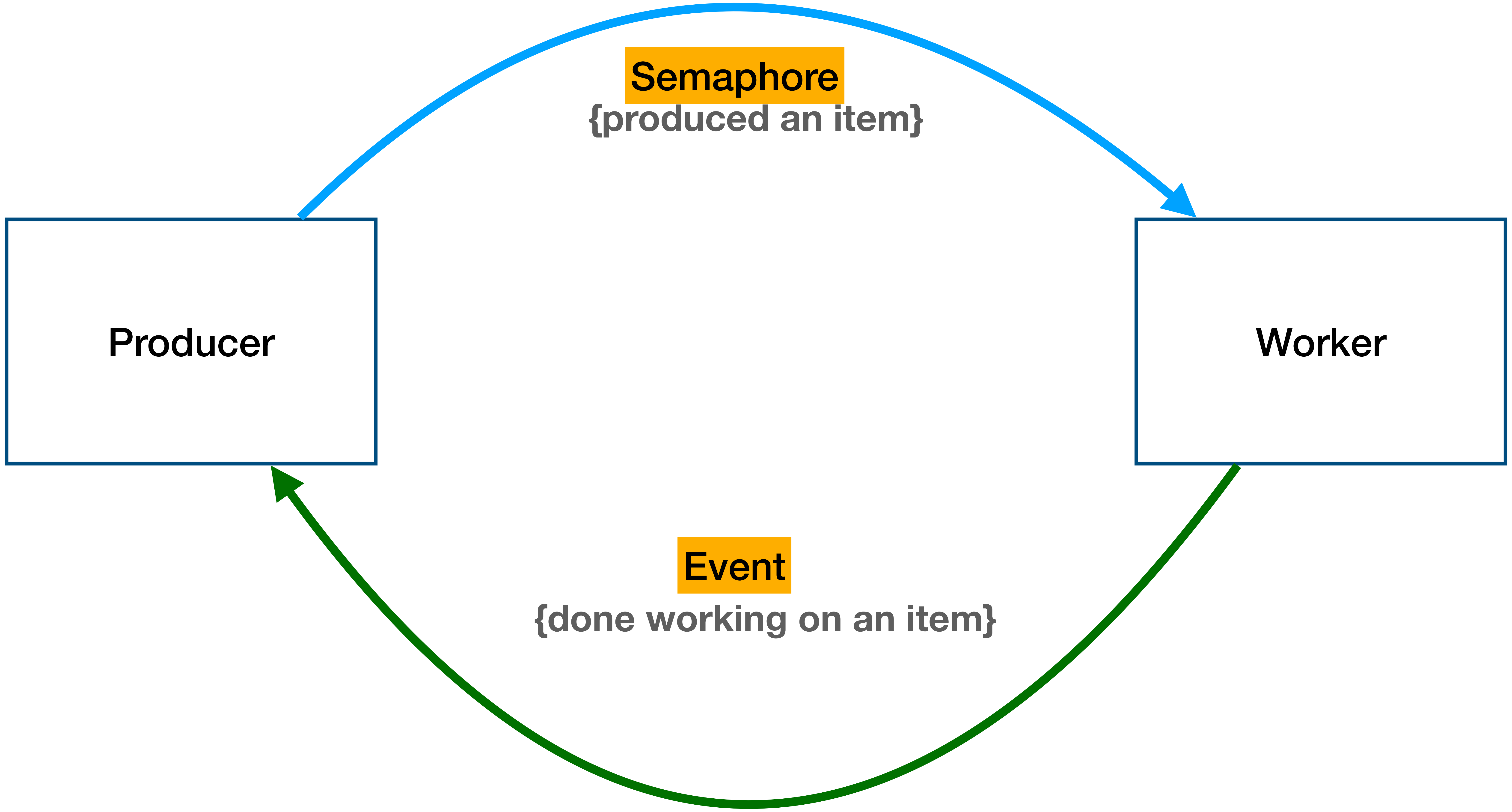
Condition Variables

Interlude

- Working with all of the synchronization primitives is a lot trickier than it looks
- There are a lot of nasty corner cases and horrible things that can go wrong
- Bad performance, deadlock, livelock, starvation, bizarre CPU scheduling, etc...
- All are valid reasons to <u>not</u> use threads

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

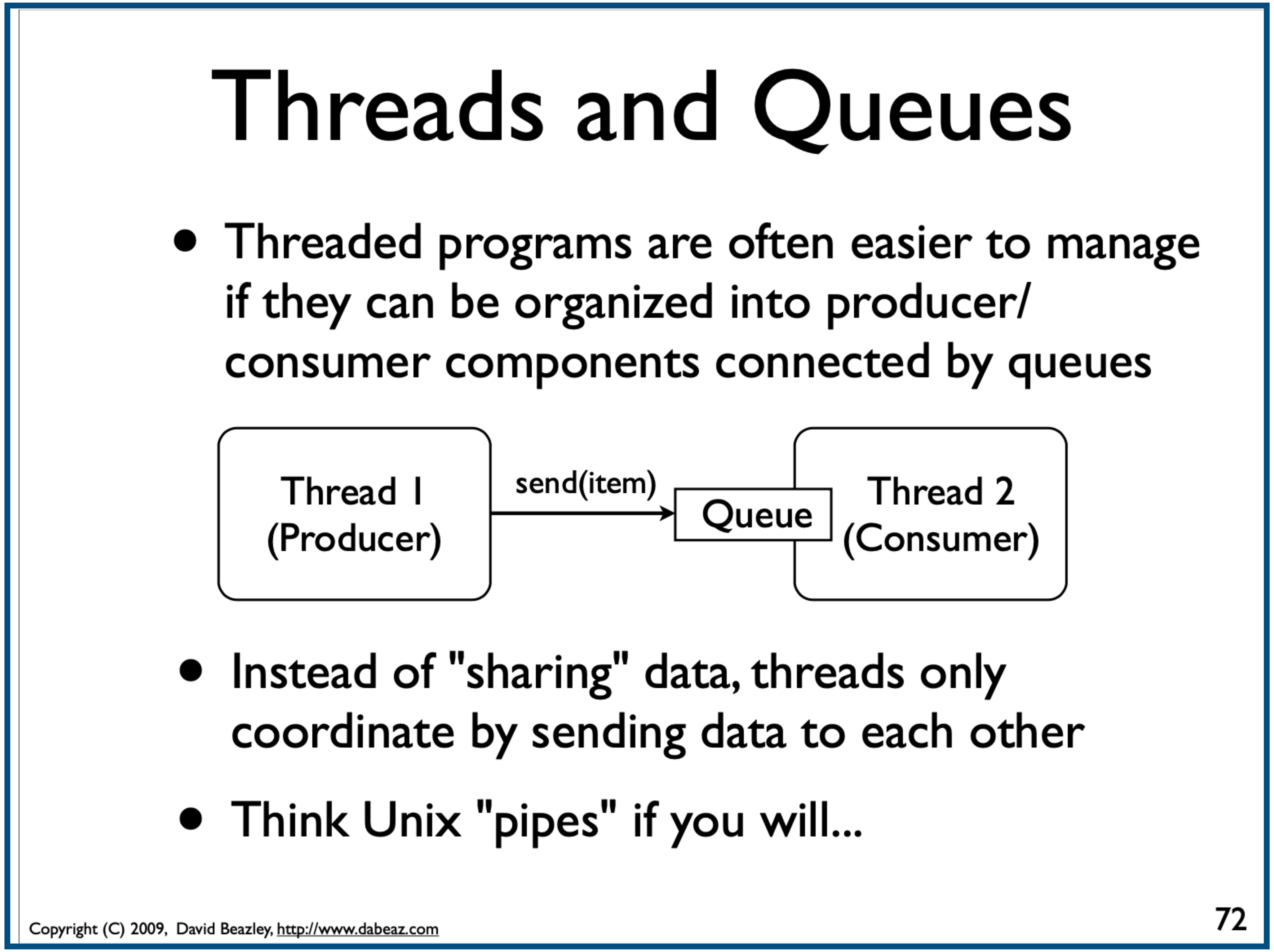
Example on Events and Semaphore



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

Threads and Queues

Threads and Queues



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

Threads and Queues

Queue Library Module

- Python has a thread-safe queuing module
- Basic operations

```
from Queue import Queue

q = Queue([maxsize])   # Create a queue
q.put(item)            # Put an item on the queue
q.get()                # Get an item from the queue
q.empty()              # Check if empty
q.full()               # Check if full
```

- Usage : You try to strictly adhere to get/put operations. If you do this, you don't need to use other synchronization primitives.

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

Threads and Queues

Queue Usage

- Most commonly used to set up various forms of producer/consumer problems

```
from Queue import Queue
q = Queue()
```

<u>Producer Thread</u>

```
for item in produce_items():
    q.put(item)
```

<u>Consumer Thread</u>

```
while True:
    item = q.get()
    consume_item(item)
```

- Critical point : You don't need locks here

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

Threads and Queues

Queue Signaling

- Queues also have a signaling mechanism

```
q.task_done()      # Signal that work is done
q.join()           # Wait for all work to be done
```

- Many Python programmers don't know about this (since it's relatively new)
- Used to determine when processing is done

Producer Thread

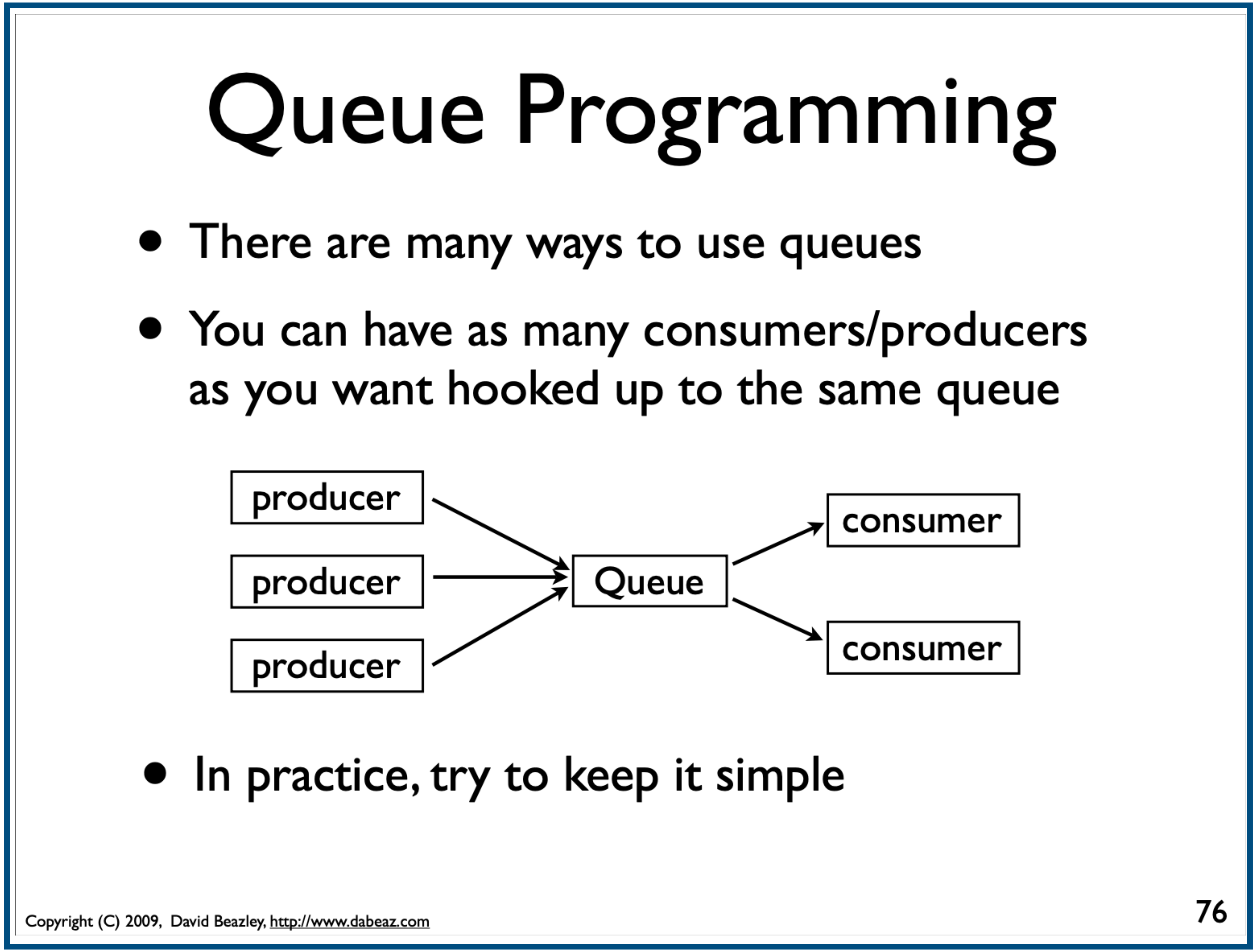
```
for item in produce_items():
    q.put(item)
# Wait for consumer
q.join()
```

Consumer Thread

```
while True:
    item = q.get()
    consume_item(item)
    q.task_done()
```


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

Threads and Queues



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
- **Python threads synchronization: Locks, RLocks, Semaphores, Conditions and Queues** http://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-events-and-queues/
- **Multithreading in Python | Set 1** https://www.geeksforgeeks.org/multithreading-python-set-1/
- **Multithreading in Python | Set 2** (Synchronization) https://www.geeksforgeeks.org/multithreading-in-python-set-2-synchronization/

The End