Programming with Concurrent Threads (Part 4)

Copyright © 2022 by
Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover:
  - A larger example
  - Processes vs. threads
  - Commentary
  - (if time) Some appendices

Agenda

- **A larger example: version 1**
- A larger example: version 2
- A larger example: version 3
- A larger example: version 4
- A larger example: version 5
- Processes vs. threads
- Commentary

Larger Example: Version 1

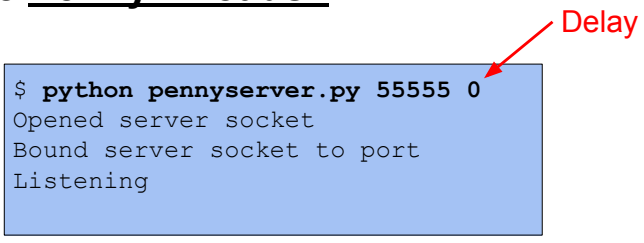
- *Programming with Concurrent Processes* lecture
  - Penny client-server apps
  - Focus on improving **server** by programming with concurrent **processes**
- This lecture
  - Penny client-server apps
  - Focus on improving **client** by programming with concurrent **threads**

Larger Example: Version 1

- Just to refresh your memory…
- See **PennyThreads1** app
  - Same as PennyProcesses2 app
    - From *Programming with Concurrent Processes* lecture

Larger Example: Version 1

- See **PennyThreads1**

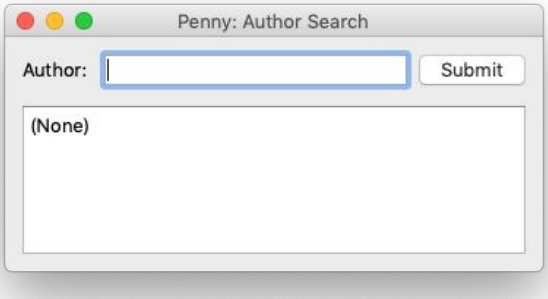



```
$ python pennyserver.py 55555 0
Opened server socket
Bound server socket to port
Listening
```

```
$ python penny.py localhost 55555
```

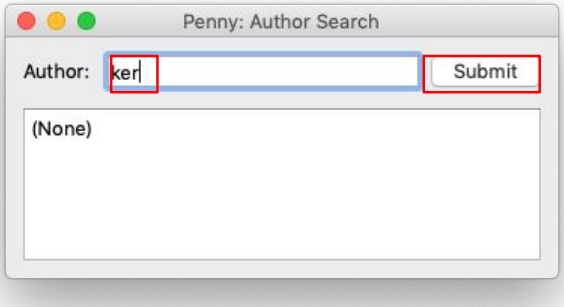
Larger Example: Version 1

- See **PennyThreads1** (cont.)



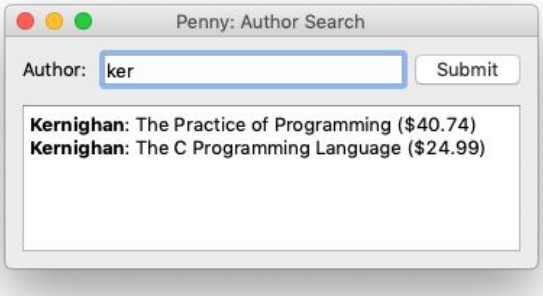
Larger Example: Version 1

- See **PennyThreads1** (cont.)



Larger Example: Version 1

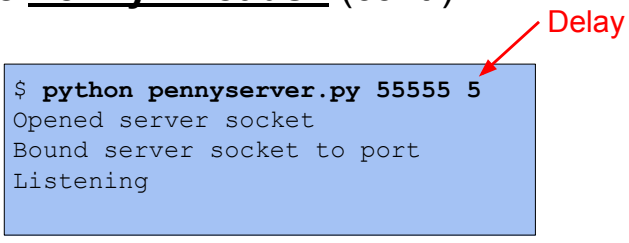
- See **PennyThreads1** (cont.)



With essentially no delay

Larger Example: Version 1

- See **PennyThreads1** (cont.)

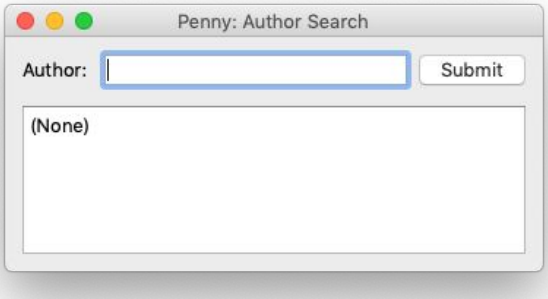



```
$ python pennyserver.py 55555 5
Opened server socket
Bound server socket to port
Listening
```

```
$ python penny.py localhost 55555
```

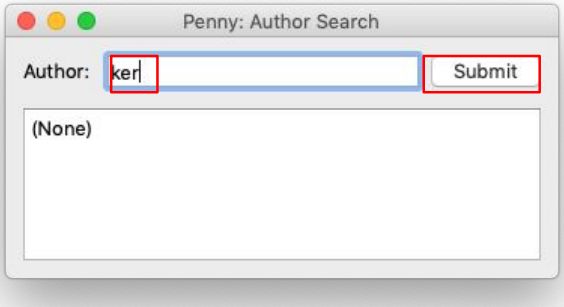
Larger Example: Version 1

- See **PennyThreads1** (cont.)



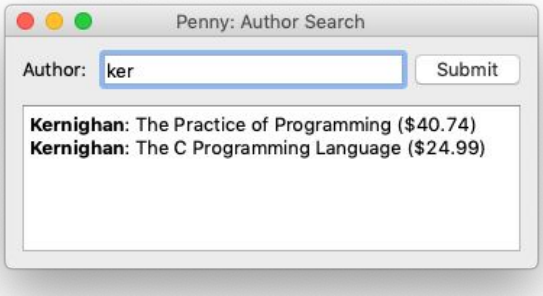
Larger Example: Version 1

- See **PennyThreads1** (cont.)



Larger Example: Version 1

- See **PennyThreads1** (cont.)



After a
5 second
delay

Larger Example: Version 1

- See **PennyThreads1** (cont.)
  - penny.sql, penny.sqlite
  - book.py, database.py
  - **pennyserver.py**
  - **penny.py**

14

Larger Example: Version 1

[see slide]

Code notes: pennyserver.py
Uses multiple processes

Code notes: penny.py
Traditional PyQt5 program
Uses one thread

Larger Example: Version 1

- Problem:
  - Inconsistent window state after typing author and before clicking Submit button

Agenda

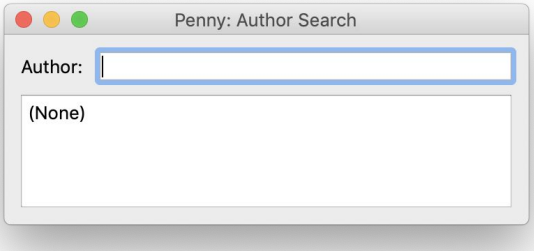
- A larger example: version 1
- **A larger example: version 2**
- A larger example: version 3
- A larger example: version 4
- A larger example: version 5
- Processes vs. threads
- Commentary

Larger Example: Version 1

- Solution: redesign...
  - Eliminate Submit button
  - GUI refreshes with each keystroke

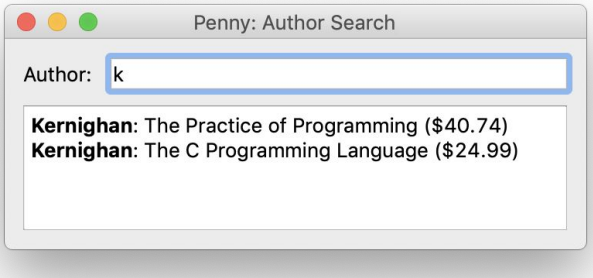
Larger Example: Version 2

- See **PennyThreads2** app



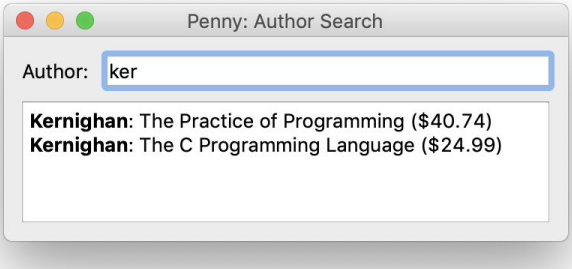
Larger Example: Version 2

- See **PennyThreads2** app



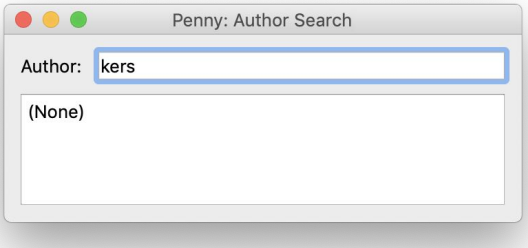
Larger Example: Version 2

- See **PennyThreads2** app



Larger Example: Version 2

- See **PennyThreads2** app



Larger Example: Version 2

- See **PennyThreads2** app (cont.)
  - penny.sql, penny.sqlite
  - book.py, database.py
  - pennyserver.py
  - **penny.py**

Larger Example: Version 2

- Problem:
  - Each keystroke initiates network comm
  - GUI is “laggy”

23

Larger Example: Version 2

Problem:

Each keystroke initiates network comm

GUI is “laggy”

GUI is unresponsive during network comm

Agenda

- A larger example: version 1
- A larger example: version 2
- **A larger example: version 3**
- A larger example: version 4
- A larger example: version 5
- Processes vs. threads
- Commentary

Larger Example: Version 2

- Solution: redesign
  - Use threads!!!

Larger Example: Version 3

- See **PennyThreads3** app
  - penny.sql, penny.sqlite
  - book.py, database.py
  - pennyserver.py
  - **penny.py**

26

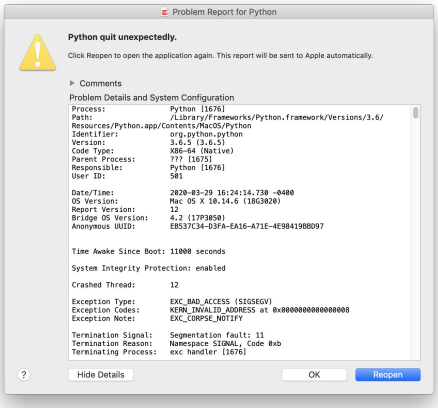
Larger Example: Version 3

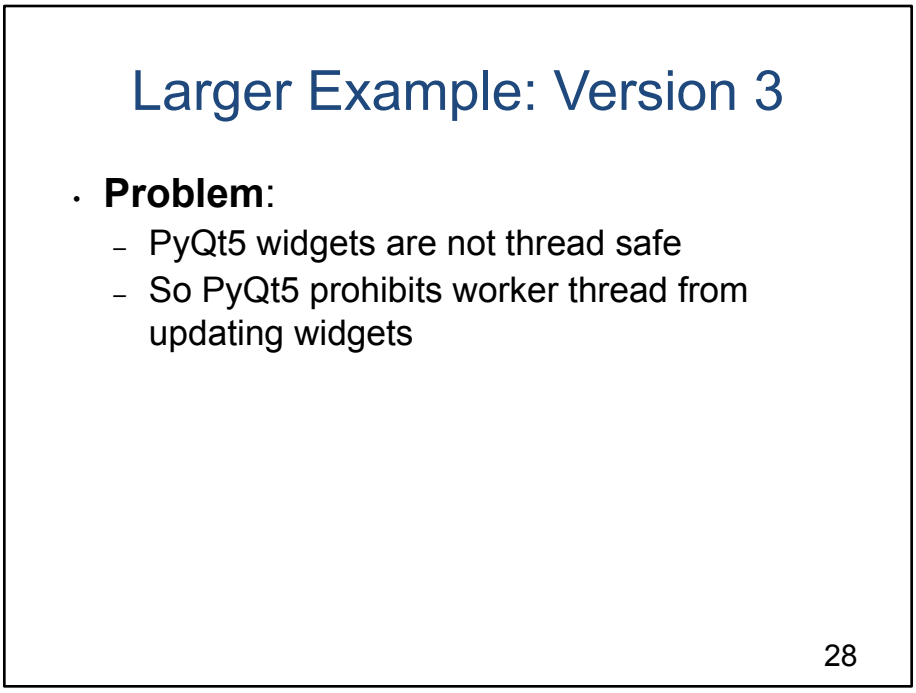
Code notes

Client-side uses multiple threads
After each keystroke main/GUI thread spawns “worker” thread
Worker thread does network comm, updates GUI when finished
Meanwhile main/GUI thread remains responsive

Larger Example: Version 3

- **Problem**:





Larger Example: Version 3

Problem:

PyQt5 widgets are not thread safe

Race condition if worker thread and main/GUI thread update widgets concurrently

And so:

PyQt5 doesn't allow worker thread to update widgets

PyQt5 generates a run-time error if worker (non-GUI) thread tries to update widgets!!!

Agenda

- A larger example: version 1
- A larger example: version 2
- A larger example: version 3
- **A larger example: version 4**
- A larger example: version 5
- Processes vs. threads
- Commentary

Larger Example: Version 4

- **Solution**: redesign...
  - Worker thread communicates book list to main thread
  - Main thread updates GUI

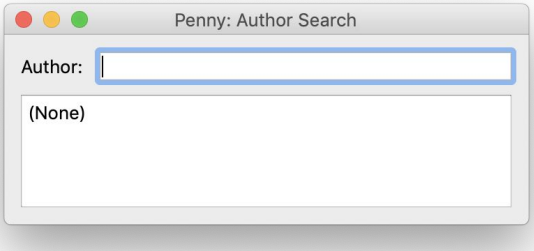
Larger Example: Version 4

- Worker thread must communicate changes to main thread

- **Question**: How?
- **Answer**: Python `Queue` class
  - Semi-thread-safe queue
  - Worker thread **puts** book list into queue
  - Main thread **gets** book list from queue and updates GUI
  - See Appendix

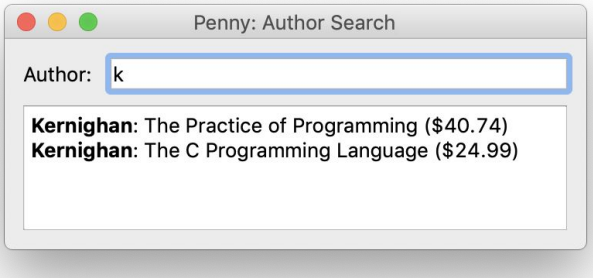
Larger Example: Version 4

- See **PennyThreads4** app



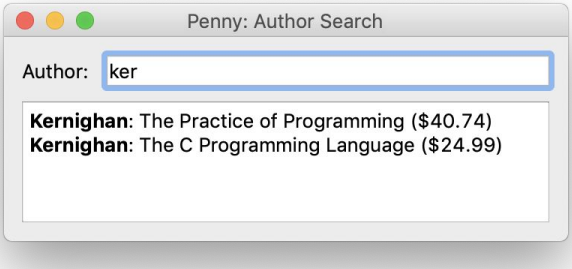
Larger Example: Version 4

- See **PennyThreads4** app



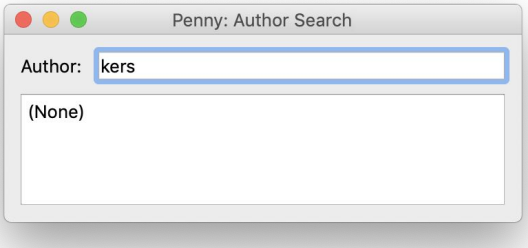
Larger Example: Version 4

- See **PennyThreads4** app



Larger Example: Version 4

- See **PennyThreads4** app



Larger Example: Version 4

- See **PennyThreads4** app
  - penny.sql, penny.sqlite
  - book.py, database.py
  - pennyserver.py
  - **penny.py**

36

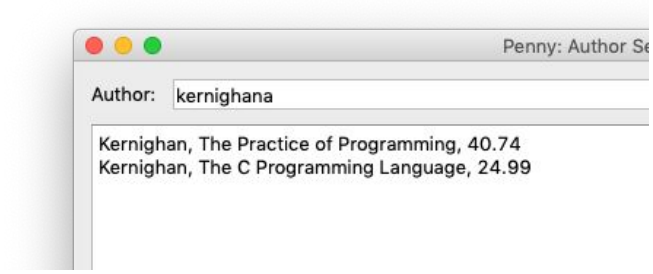
Larger Example: Version 4

Code notes:

Client-side uses multiple threads
After each keystroke main/GUI thread spawns “worker” thread
Worker thread does network comm, puts result object in Queue object
Main thread “polls” Queue object periodically, gets result object, updates widgets
Meanwhile main thread remains responsive

Larger Example: Version 4

- **Problem**:



Larger Example: Version 4

- **Problem (cont.)**:
  - Order of server responses is undefined
  - Order of worker thread completion is undefined
  - May yield inconsistent window state

38

Larger Example: Version 4

[see slide]

Example:

Query for ‘k’ followed by query for “ka”

Agenda

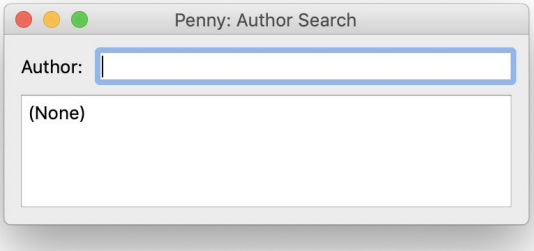
- A larger example: version 1
- A larger example: version 2
- A larger example: version 3
- A larger example: version 4
- **A larger example: version 5**
- Processes vs. threads
- Commentary

Larger Example: Version 5

- **Solution**: redesign...
  - Client, when about to spawn a new worker thread, tells existing worker thread to stop

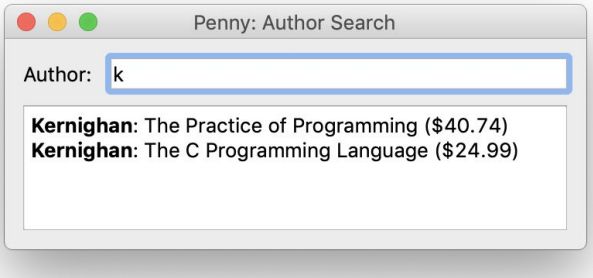
Larger Example: Version 5

- See **PennyThreads5** app



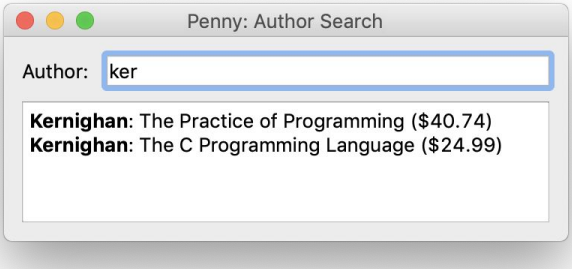
Larger Example: Version 5

- See **PennyThreads5** app



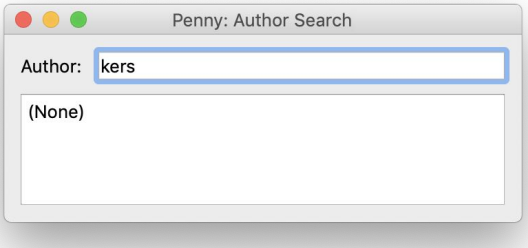
Larger Example: Version 5

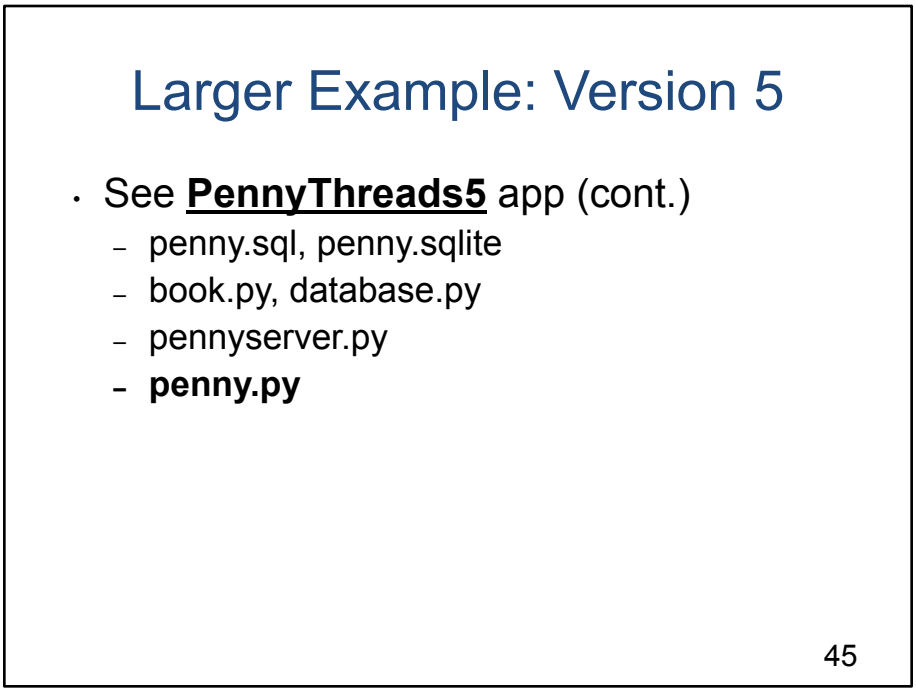
- See **PennyThreads5** app



Larger Example: Version 5

- See **PennyThreads5** app





Larger Example: Version 5

Code note

Before starting new worker thread, aborts old worker thread

Agenda

- A larger example: version 1
- A larger example: version 2
- A larger example: version 3
- A larger example: version 4
- A larger example: version 5
- **Processes vs. threads**
- Commentary

Processes vs. Threads

- **Question**:
  - You decide that you need concurrency; do you use multiple **processes** or multiple **threads**?
- **Answer (partial)**:
  - In part, depends upon programming lang...

Processes vs. Threads

Language	Multiple Processes	Multiple Threads
Java	very rare	common
C/Unix	common	rare
JavaScript	very rare	impossible
Python	depends	depends

Concerning Python...

48

Processes vs. Threads

Java:

Multiple threads: common; there is language (not just library) support for them
Multiple processes: very rare; forking a child JVM is expensive!!! And there's no need

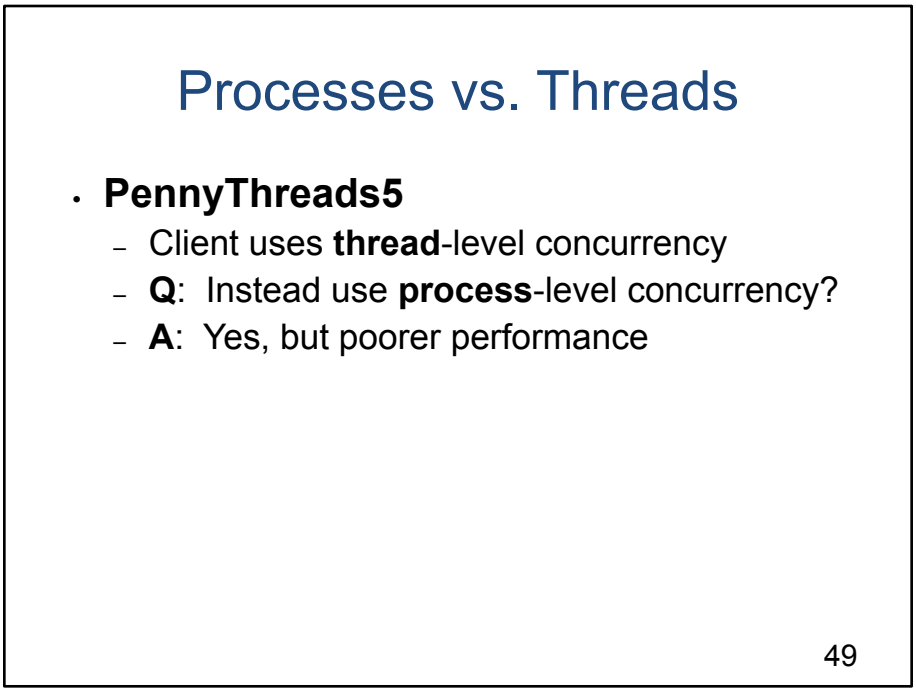
C/Unix:

Multiple processes: common; Unix systems have fork() and wait() POSIX standard functions
Multiple threads: rare; must use a non-standard library such as pthread

JavaScript:

Multiple processes: very rare
Multiple threads: impossible
JavaScript doesn't allow multiple threads; there is only one thread
Instead: Event driven
Each event handler executes without preemption to completion
By the way…
Important that no event handler consume too much time
Routine to ask browser/node to do background work, giving it a callback function
Brower/node then puts new event (call to callback function) on event queue when finished

Python...



Processes vs. Threads

PennyThreads5

Client uses **thread-level** concurrency

Q: Instead use **process-level** concurrency?

A: Yes, but poorer performance

Thread-level concurrency:

Inexpensive spawns

Inexpensive data comm: main thread comms with worker threads via shared Queue object

Process-level concurrency:

Expensive forks

Expensive data comm: parent/main process comms with child/worker processes via pipes

Processes vs. Threads

- **PennyThreads5**
  - Server uses **process**-level concurrency
  - **Q**: Instead use **thread**-level concurrency?
  - **A**: No!!!...

Processes vs. Threads

- Suppose process P1 has threads T1 and T2
- In principle:
  - Multiple CPUs available => T1 and T2 run in parallel
- In Java and C/pthread:
  - Multiple CPUs available => T1 and T2 run in parallel

Processes vs. Threads

- In Python (specifically CPython):
  - Multiple CPUs available => T1 and T2 **do not** run in parallel!!!
  - ***Global Interpreter Lock (GIL)***
    - Allows only one of P1's threads to execute at a time…
    - As if each Python process has only 1 CPU

Processes vs. Threads

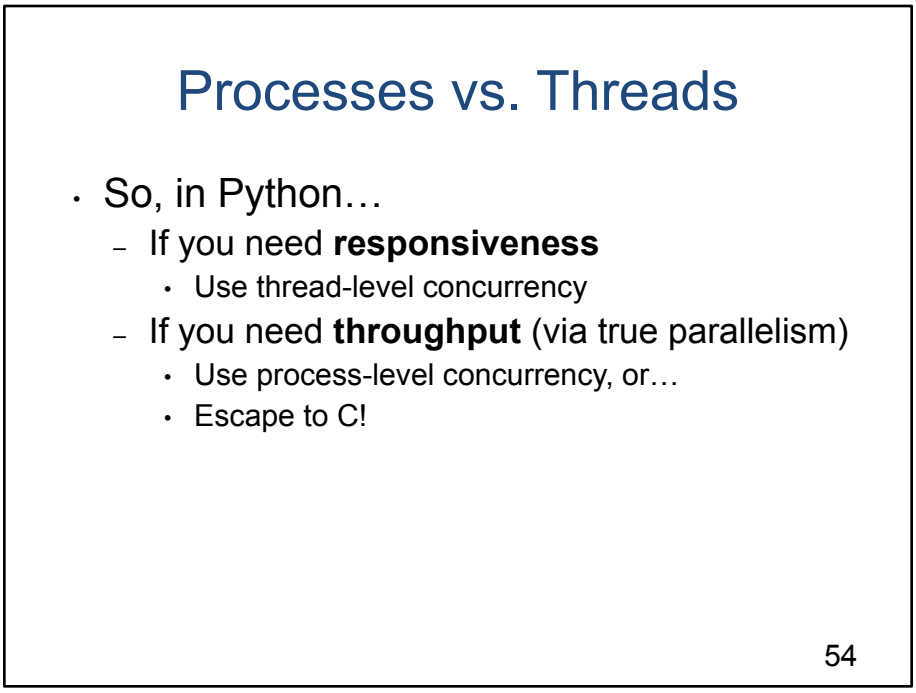
- GIL advantages
  - Simplifies Python memory management (reference counting)
  - Makes single-threaded programs faster
- GIL disadvantage
  - Multi-threaded programs run slower

53

Processes vs. Threads

Mostly beyond our scope

[see slide]



Processes vs. Threads

So, in Python…

If you need **responsiveness**

Use thread-level concurrency

If you need **throughput** (via true parallelism)

Use process-level concurrency, or…

Escape to C

Write Python code to call C code; write C code to use thread-level concurrency

See Numpy library

Agenda

- A larger example: version 1
- A larger example: version 2
- A larger example: version 3
- A larger example: version 4
- A larger example: version 5
- Processes vs. threads
- **Commentary**

Commentary

- **Process**-level concurrency is:
  - Essential
  - Safe: distinct processes share no data
- **Thread**-level concurrency is:
  - Essential in many pgmming langs
  - **Dangerous**: distinct threads can share objects => potential race conditions, potential deadlocks

Commentary

- Some questions:
  - Should all objects automatically be thread-safe?
    - Should all fields automatically be private and all methods automatically be “locked”?

> “It is astounding to me that Java’s insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.”
> -- Per Brinch Hansen, 1999

Commentary

- Some questions (cont.):
  - Should methods be “locked” by default?
  - Should we use process-level concurrency instead of thread-level concurrency whenever possible?
  - In the long run, is thread-level concurrency a passing phase?

Concurrency Resources

- For more information:
  - Alex Martelli, Anna Ravenscroft, and Steve Holden. *Python in a Nutshell*, Chapter 14.
  - Cay Horstmann. *Core Java (Volume 1)*, Chapter 14.
  - And then OS textbooks

Summary

- We have covered:
  - A larger example
  - Processes vs. threads
  - Commentary

Summary

- We have covered:
  - What a thread is
  - How to spawn and join threads
  - Race conditions and how to avoid them
  - A larger example
  - Conclusion and commentary
- See also:
  - **Appendix 1**: Deadlocks
  - **Appendix 2**: The Python Queue Class
  - **Appendix 3**: Threads in Java
  - **Appendix 4**: Threads in C

Appendix 1: Deadlocks

Deadlocks

- Problem: ***deadlock***
  - Thread1
    - Has the lock on object1
    - Needs the lock on object2
  - Thread2
    - Has the lock on object2
    - Needs the lock on object1
  - Thread1 and thread2 block forever

Deadlocks: Example

- See **deadlock.py**
  - The job:
    - alice_acct: 0, bob_acct: 0
    - alice_to_bob_thread: transfer 1 from alice_acct to bob_acct, 1000 times
    - bob_to_alice_thread: transfer 1 from bob_acct to alice_acct, 1000 times
    - alice_acct: 0, bob_acct: 0?

64

Deadlock: Example

[see slide]

Code notes

Study code statically

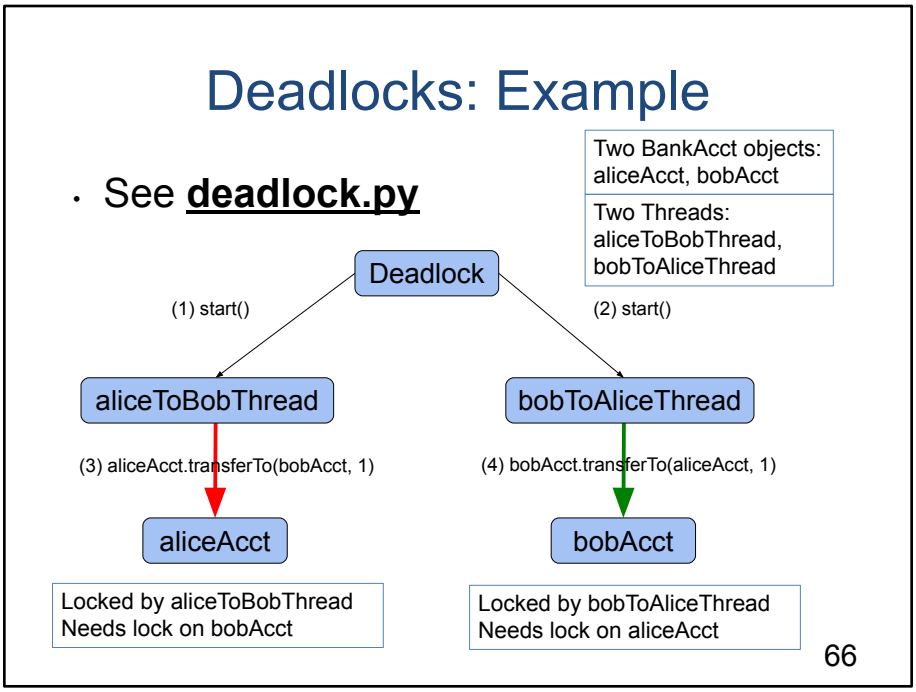
Deadlocks: Example

- See **deadlock.py** (cont.)

```
$ python deadlock.py
…
Alice: -26
Bob: 26
Alice: -27
Bob: 27
Alice: -28
Bob: 28
Alice: -29
Bob: 29
Alice: -30
Bob: 30
```

```
$ python deadlock.py
…
Alice: -100
Bob: 100
Alice: -101
Bob: 101
Alice: -102
Bob: 102
Alice: -103
Bob: 103
Alice: -104
Bob: 104
```

Usually, program halts at an unpredictable time



Deadlock: Example

[see slide]

Code notes

Study code dynamically

Deadlocks: Example

- See **deadlockw.py**
  - The job:
    - Same as deadlock.py
  - The difference:
    - Uses `with` statement

Deadlocks: Requirements

- Requirements for deadlock
  - (1) Mutual exclusion
  - (2) Hold and wait
  - (3) No preemption
  - (4) Circular chain
    - Thread1 has lock needed by thread2
    - Thread2 has lock needed by thread3
    - …
    - ThreadN has lock needed by thread1

Deadlocks: Prevention

- Preventing deadlocks
  - OS level
    - Negate one of the four requirements
  - App level
    - Avoid “circular chain”…

Deadlocks: Prevention

- **Solution**:
  - Give each BankAcct object a sequence number
  - Pact: Thread must acquire locks on BankAcct objects in order by sequence number

70

Preventing Deadlock

Solution:

Give each BankAcct object a sequence number

BankAcct object 1, BankAcct object 2, …

Pact: Thread must acquire locks on BankAcct objects in order by sequence number

Deadlocks: Prevention Example

- See **nodeadlock.py**
  - The job:
    - Same as deadlock.py

71

Preventing Deadlock: Example

Code notes:

Study code statically

Study code dynamically

Each thread acquires lock on lower-numbered BankAcct object before acquiring lock on higher-numbered BankAcct object

No possibility of deadlock

Deadlocks: Prevention Example

- See **nodeadlock.py** (cont.)

```
$ python nodeadlock.py
…
Alice: -4
Bob: 3
Alice: -3
Bob: 2
Alice: -2
Bob: 1
Alice: -1
Bob: 0
Alice: 0
Finished
$
```

```
$ python nodeadlock.py
…
Bob: -4
Alice: 3
Bob: -3
Alice: 2
Bob: -2
Alice: 1
Bob: -1
Alice: 0
Bob: 0
Finished
$
```

Program runs to completion

Deadlocks: Prevention Example

- See **nodeadlockw.py**
  - The job:
    - Same as nodeadlock.py
  - The difference:
    - Uses `with` statement

Appendix 2:
The Python Queue Class

The Python Queue Class

- Python `Queue` class
  - Semi-thread-safe
  - Designed for inter-thread comm

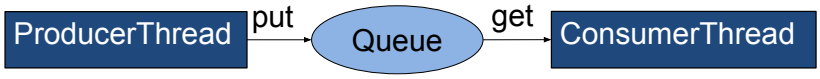
The Python Queue Class

- Python `Queue` class
  - Use case 1:

```
…
queue = Queue()
…
queue.put(item)
…
try:
    item = queue.get(block=False)
except Empty():
    # The queue is empty.
```

Queue object can contain unlimited number of items

The Python Queue Class



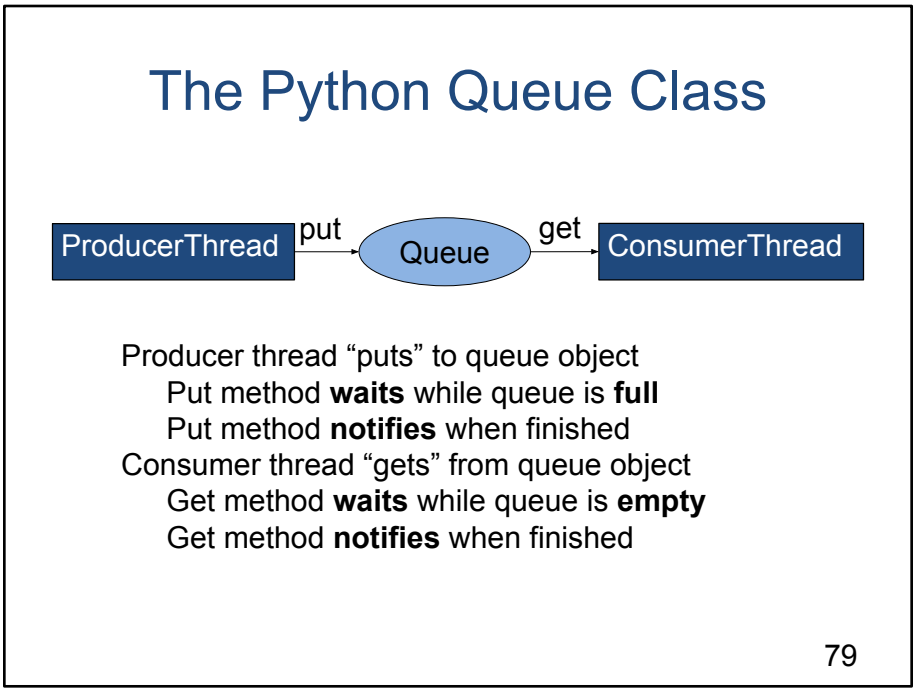
Producer thread “puts” to queue object
Consumer thread “gets” from queue object
“get” method throws exception if queue object is empty

The Python Queue Class

- Python `Queue` class
  - Use case 2:

```
…
queue = Queue(n)
…
queue.put(item)
# Waits if queue is full.
# Notifies when finished.
…
item = queue.get()
# Waits if queue is empty.
# Notifies when finished.
…
```

Queue object can contain up to n items



The Python Queue Class

[see slide]

Good example of the use of conditions

The Python Queue Class

- See **prodcon.py**
  - The job:
    - Producer thread writes 0…99 to queue
    - Consumer thread reads 0...99 from queue

```
$ python prodcon.py
...
Produced: 97
Consumed: 93
Produced: 98
Consumed: 94
Produced: 99
Consumed: 95
Consumed: 96
Consumed: 97
Consumed: 98
Consumed: 99
Finished
$
```

Appendix 3:
Threads in Java

Threads in Java

- See **Conditions.java**
  - The job:
    - Same as conditions.py
    - Uses multithreading features of the Java language

Appendix 4:
Threads in C

Threads in Java

- See **conditions.c**
  - The job:
    - Same as conditions.py
    - Uses the C **pthread** library