

Concurrent Programming (Part 4)

Copyright © 2026 by
Robert M. Dondero, Ph.D.
Princeton University

Objectives

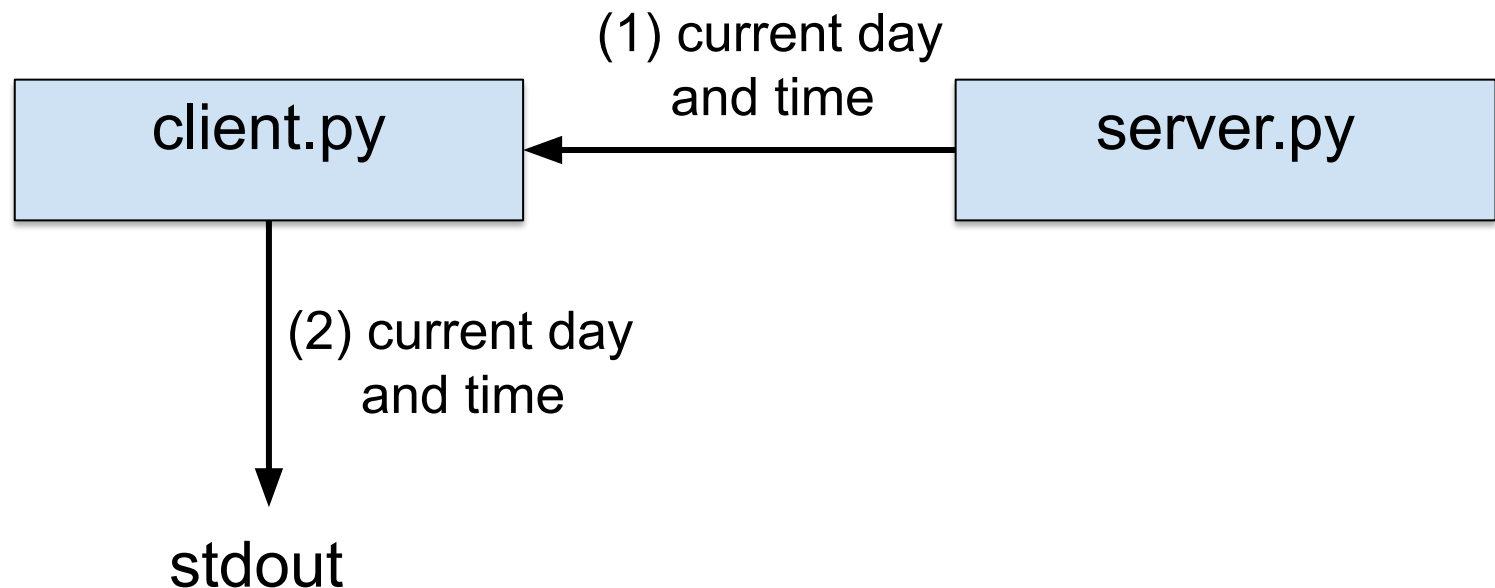
- We will cover:
 - Realistic example: I/O delays
 - Realistic example: compute delays
 - The Python GIL
 - Concurrency commentary

Agenda

- **Realistic example: I/O delays**
- Realistic example: compute delays
- The Python GIL
- Concurrency commentary

Realistic Example: I/O Delays

Recall daytime app



Realistic Example: I/O Delays

- See **DaytimeIODelay** application
 - Almost same as DayTime app from *Network Programming* lectures
 - **client.py**
 - **server.py**
 - Enhanced to implement **iodelay**
 - Delay caused by waiting for another service (e.g., database)

Realistic Example: I/O Delays

See DaytimeIODelay app:

```
$ date
Wed Sep 25 13:23:52 EDT 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:23:58 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:24:03 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:24:08 2024
$
```

```
$ export IODELAY=5
$ python server.py 55555
Opened server socket
Bound server socket to port
Listening
Accepted connection
Opened socket
Closed socket
Accepted connection
Opened socket
Closed socket
Accepted connection
Opened socket
Closed socket
```

Unacceptable

Realistic Example: I/O Delays

- See **DaytimeIODelayP**
 - client.py
 - **server.py**
 - Forks a new process to handle each client request

Realistic Example: I/O Delays

See DaytimeIODelayP app:

```
$ date
```

```
Wed Sep 25 13:25:48 EDT 2024
```

```
$
```

```
$ python client.py localhost 55555
```

```
Wed Sep 25 13:25:54 2024
```

```
$
```

```
$ python client.py localhost 55555
```

```
Wed Sep 25 13:25:55 2024
```

```
$
```

```
$ python client.py localhost 55555
```

```
Wed Sep 25 13:25:55 2024
```

```
$
```

```
$ export IODELAY=5
```

```
$ python server 55555
```

```
Opened server socket
```

```
Bound server socket to port
```

```
Listening
```

```
Accepted connection
```

```
Opened socket
```

```
Closed socket in parent process
```

```
Forked child process
```

```
Closed socket in child process
```

```
Exiting child process
```

```
Accepted connection
```

```
Opened socket
```

```
Closed socket in parent process
```

```
Forked child process
```

```
Closed socket in child process
```

```
Exiting child process
```

```
Accepted connection
```

```
Opened socket
```

```
Closed socket in parent process
```

```
Forked child process
```

```
Closed socket in child process
```

```
Exiting child process
```

Aside: Zombies

~~Parent process forks child process
Parent process waits for (reaps) child process~~

Proper
pattern

Parent process forks child process
Parent process proceeds
Child process exits
Parent process receives SIGCHLD signal
Parent process waits for (reaps) child process

Alternative
proper
pattern

Parent process forks child process
Parent process proceeds
Parent process forks child process

Acceptable
in
Python

Realistic Example: I/O Delays

- See **DaytimeIODelayT**
 - client.py
 - **server.py**
 - Spawns a new thread to handle each client request

Realistic Example: I/O Delays

See [DaytimeIODelayT](#) app:

```
$ date
```

```
Wed Sep 25 13:27:01 EDT 2024
```

```
$
```

```
$ python client.py localhost 55555
```

```
Wed Sep 25 13:27:06 2024
```

```
$
```

```
$ python client.py localhost 55555
```

```
Wed Sep 25 13:27:07 2024
```

```
$
```

```
$ python client.py localhost 55555
```

```
Wed Sep 25 13:27:07 2024
```

```
$
```

```
$ export IODELAY=5
$ python server.py 55555
Opened server socket
Bound server socket to port
Listening
Accepted connection
Opened socket
Spawned child thread
Closed socket in child thread
Exiting child thread
Accepted connection
Opened socket
Spawned child thread
Closed socket in child thread
Exiting child thread
Accepted connection
Opened socket
Spawned child thread
Closed socket in child thread
Exiting child thread
```

Agenda

- Environment variables
- Realistic example: I/O delays
- **Realistic example: compute delays**
- The Python GIL
- Concurrency commentary

Realistic Example: Compute Delays

- See **DaytimeCDelay** application
 - [Almost same as DayTime app from *Network Programming* lectures]
 - client.py
 - **server.py**
 - Enhanced to implement **cdelay**
 - Delay caused by performing a time-consuming computation (e.g., matrix manipulation)

Realistic Example: Compute Delays

See DaytimeCDelay app:

```
$ date
Wed Sep 25 13:40:48 EDT 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:40:54 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:40:59 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:41:04 2024
$
```

```
$ export CDELAY=5
$ python server.py 55555
Opened server socket
Bound server socket to port
Listening
Accepted connection
Opened socket
Closed socket
Accepted connection
Opened socket
Closed socket
Accepted connection
Opened socket
Closed socket
```

Unacceptable

Realistic Example: Compute Delays

- See **DaytimeCDelayP**
 - client.py
 - **server.py**
 - Forks a new process to handle each client request

Realistic Example: Compute Delays

See DaytimeCDelayP app:

```
$ date
Wed Sep 25 13:42:13 EDT 2024
$
```

```
$ python client.py localhost 5555
Wed Sep 25 13:42:18 2024
$
```

```
$ python client.py localhost 5555
Wed Sep 25 13:42:19 2024
$
```

```
$ python client.py localhost 5555
Wed Sep 25 13:42:19 2024
$
```

```
$ export CDELAY=5
$ python server.py 5555
Opened server socket
Bound server socket to port
Listening
Accepted connection
Opened socket
Closed socket in parent process
Forked child process
Closed socket in child process
Exiting child process
Accepted connection
Opened socket
Closed socket in parent process
Forked child process
Closed socket in child process
Exiting child process
Accepted connection
Opened socket
Closed socket in parent process
Forked child process
Closed socket in child process
Exiting child process
```

Realistic Example: Compute Delays

- See **DaytimeCDelayT**
 - client.py
 - **server.py**
 - Spawns a new thread to handle each client request

Realistic Example: Compute Delays

See DaytimeCDelayT app:

```
$ date
Wed Sep 25 13:45:09 EDT 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:45:24 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:45:23 2024
$
```

```
$ python client.py localhost 55555
Wed Sep 25 13:45:24 2024
$
```

```
$ export CDELAY=5
$ python server.py 55555
Opened server socket
Bound server socket to port
Listening
Accepted connection
Opened socket
Spawned child thread
Closed socket in child thread
Exiting child thread
Accepted connection
Opened socket
Spawned child thread
Closed socket in child thread
Exiting child thread
Accepted connection
Opened socket
Spawned child thread
Closed socket in child thread
Exiting child thread
```

Unacceptable!!!???

Agenda

- Realistic example: I/O delays
- Realistic example: Compute delays
- **The Python GIL**
- Concurrency commentary

The Python GIL

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

The Python GIL

- In **Python** (specifically **CPython**):
 - Multiple processors 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 only one processor exists

The Python GIL

- GIL advantages
 - Simplifies Python memory management (reference counting)
- GIL disadvantage
 - As described previously...
 - Multi-threaded programs can use only one processor (at a time)
 - Multiple threads cannot run in parallel

The Python GIL

- Important recent development:
 - **Python 3.14** has a *free-threaded build*
 - GIL-free Python
 - `python3.14t`
 - Slower for single-threaded programs
 - Breaks compatibility with some existing C extensions
 - May reveal existing race conditions

The Python GIL

- So, in Python...

Kind of Program	Example	Then use:
I/O-bound	Program waits for DB comm to complete	Thread-level concurrency
Compute-bound	Program performs complex math computation	Process-level concurrency *

* But better not to use Python at all!

The Python GIL

- So, more generally...

Kind of Pgmming	Likely to use:	Likely to implement concurrency via:
System-level programming	C	Multiple processes
Application-level programming	Java	Multiple threads For I/O-bound pgms For compute-bound pgms
Application-level programming	Python	Multiple threads For I/O-bound pgms

Agenda

- Realistic example: I/O delays
- Realistic example: compute delays
- The Python GIL
- **Concurrency commentary**

Concurrency Commentary

- **Process-level concurrency** is:
 - **Essential**, esp. at system level
 - **Safe**: concurrent processes share no data
 - **Slow**: forking processes is relatively slow
- **Thread-level concurrency** is:
 - **Essential**, esp. at application level
 - **Dangerous**: concurrent threads can share objects => potential race conditions, potential deadlocks
 - **Fast**: spawning threads is relatively fast

Concurrency Commentary

- Some rhetorical 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

Concurrency Commentary

- Some rhetorical 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?

Lecture Summary

- In this lecture we covered:
 - Realistic example: I/O delays
 - Realistic example: compute delays
 - The Python GIL
 - Concurrency commentary

Lecture Series Summary

- In this lecture series we covered:
 - How to fork and wait for processes
 - How to spawn and join threads
 - Race conditions and how to avoid them
 - Environment variables
 - Realistic examples
 - The Python GIL
 - Commentary
- See also:
 - **Appendix 1: Deadlocks**

More Information

- The COS 333 *Lectures* web page provides references to supplementary information

Appendix 1: Deadlocks

Deadlocks

- **Problem:** *Deadlock*
 - Simplest case...
 - 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

- See **deadlock.py**
 - 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

Deadlocks

- 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
```

Deadlocks

- See [deadlock.py](#)

Two BankAcct objects:
aliceAcct, bobAcct

Two Threads:
aliceToBobThread,
bobToAliceThread

Deadlock

(1) start()

alice_to_bob_thread

(2) start()

bob_to_alice_thread

(3) alice_acct.transfer_to(bob_acct, 1)

alice_acct

(4) bob_acct.transfer_to(alice_acct, 1)

bob_acct

alice_to_bob_thread
Has lock on alice_acct
Needs lock on bob_acct

bob_to_alice_thread
Has lock on bob_acct
Needs lock on alice_acct

Deadlocks

- See **deadlockw.py**
 - Uses `with` statement

Deadlocks

- Deadlock general case (circular chain):
 - Thread1 has the lock on object1; needs the lock on object2
 - Thread2 has the lock on object2; needs the lock on object3
 - ...
 - Thread N has the lock on object N; needs the lock on object 1

Deadlocks

- **Solution:**
 - Make sure there are no circular chains!
 - Give each shared resources a *sequence number*
 - **Pact:** Thread must acquire shared resources in order by *sequence number*

Deadlocks

- See [nodeadlock.py](#)

```
$ 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
$
```

Deadlocks

- See **nodeadlockw.py**
 - Uses `with` statement