Programming with Concurrent Processes

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover:
  - What a **process** is
  - How to **fork** and **wait** for processes
  - How to compose a multiprocessing server

Objectives

- COS 217
  - Covers concurrent processes…
  - As implemented in C
- COS 333
  - Covers same concepts…
  - As implemented in Python

Agenda

- **Process-level concurrency**
- A single process server
- A multiprocess server

Process Concurrency

- ***Program***
  - Executable code
- ***Process***
  - An instance of a program in execution
  - Each process has its own distinct **context**

Process Concurrency

- ***Context*** consists of:
  - Process id
  - Address space: TEXT, RODATA, DATA, BSS, HEAP, STACK
  - Processor state: general purpose registers, flags register, instruction pointer register, etc.

Process Concurrency

- To implement concurrency...
- **Option 1**: Process-level concurrency
  - Multiple processes run concurrently
  - This lecture
- **Option 2**: Thread-level concurrency
  - Multiple threads run concurrently within same process
  - Next lecture

Process Concurrency

- Process-level concurrency
  - Process P1 ***forks*** process P2
  - P1 and P2 run ***concurrently***
    - > 1 CPU available => P1 and P2 run in ***parallel***
    - 1 CPU available => OS ***context switches*** between P1 and P2
    - For next two examples, assume only 1 CPU is available

Process Concurrency: Forking

- See **forking.py**
  - The job:
    - Illustrate process creation
    - Parent process forks “blue” child process
    - Parent process forks “red” child process
    - Three processes run concurrently
    - Context switches between machine lang instructions

9

Process Concurrency: Forking

[see slide]

Code notes:

Parent process forks child process
Child writes “blue” 5 times
Parent process forks child process
Child writes “red” 5 times
Three processes run concurrently
Context switch may occur after execution of any machine lang instruction

Process Concurrency: Forking

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

```
$ python forking.py
parent process terminated
blue
blue
blue
blue
blue
blue process terminated
red
red
red
red
red
red process terminated
$
```

```
$ python forking.py
parent process terminated
red
red
red
red
red
red process terminated
blue
blue
blue
blue
blue
blue process terminated
$
```

Process Concurrency: Waiting

- **Definition**:
  - A ***zombie process*** is a process that has exited but has not been **waited for** (**reaped**) by its parent process
- Zombie processes needlessly clutter the operating system's data structures

Process Concurrency: Waiting

- **Problem:**
  - forking.py creates zombie child processes
- **Solution:**
  - Define parent process to wait for (reap) its child processes…

Process Concurrency: Waiting

- See **waiting.py**
  - The job:
    - Same as forking.py, except…
    - Parent process **waits** for (**reaps**) (Python: ***joins***) its child processes
    - Parent process is *blocked* until both child processes exit

Process Concurrency: Waiting

[see slide]

Code notes:

Same as forking.py, except…
Parent process waits for (reaps) its child processes
Parent process is *blocked* until both child processes exit
Proper pattern

Process Concurrency: Waiting

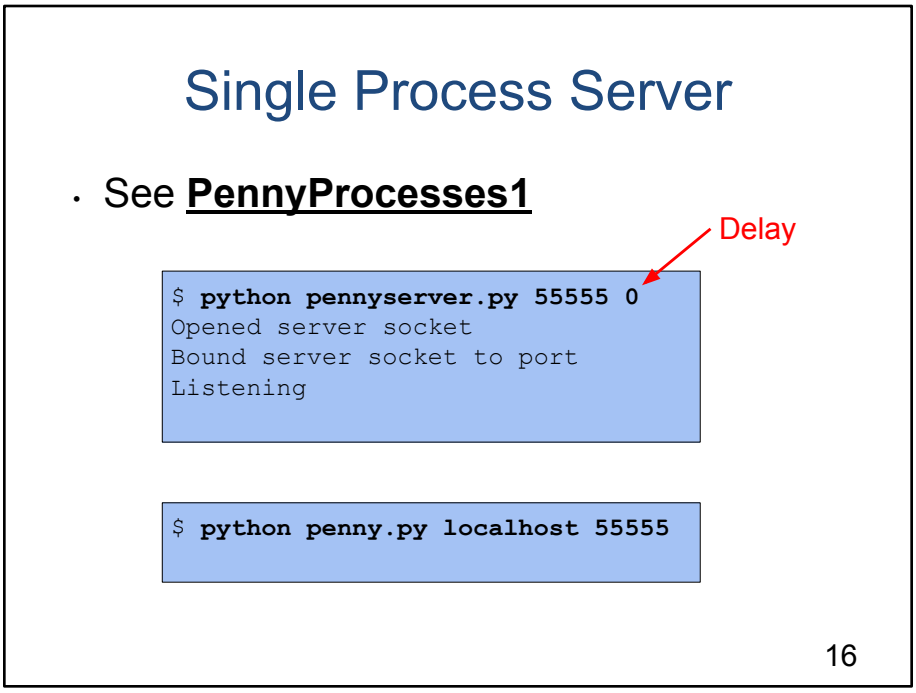
- See **waiting.py**

```
$ python waiting.py
blue
blue
blue
blue
blue
blue process terminated
red
red
red
red
red
red process terminated
parent process terminated
$
```

```
$ python waiting.py
red
red
red
red
red
red process terminated
blue
blue
blue
blue
blue
blue process terminated
parent process terminated
$
```

Agenda

- Process-level concurrency
- **A single process server**
- A multiprocess server



Multiprocessing Servers: Motivation

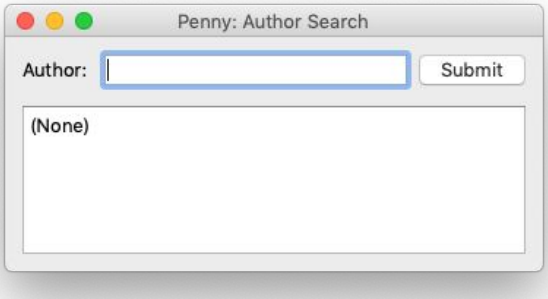
We've seen many Penny applications
All have been web applications
Now: a Penny desktop application

[see slide]

Delay allows us to observe the behavior of the app in the presence of a slow/busy server

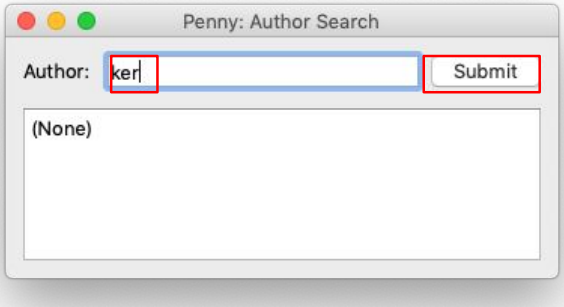
Single Process Server

- See **PennyProcesses1** (cont.)



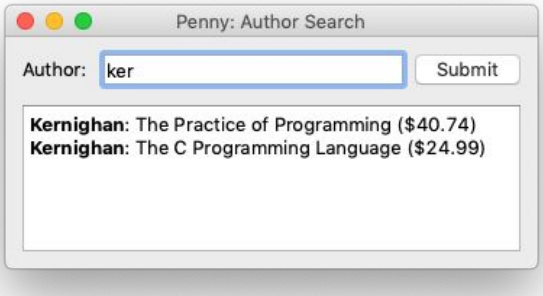
Single Process Server

- See **PennyProcesses1** (cont.)



Single Process Server

- See **PennyProcesses1** (cont.)



With essentially no delay

Single Process Server

- See **PennyProcesses1** (cont.)

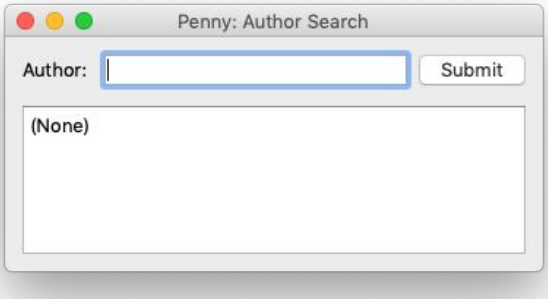
Delay

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

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

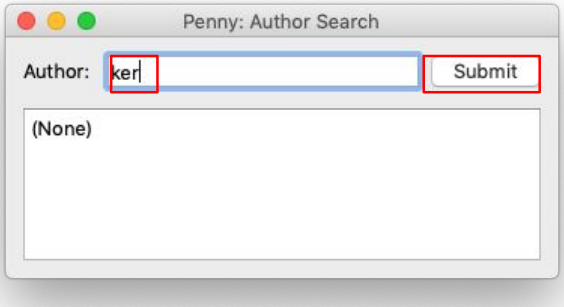
Single Process Server

- See **PennyProcesses1** (cont.)



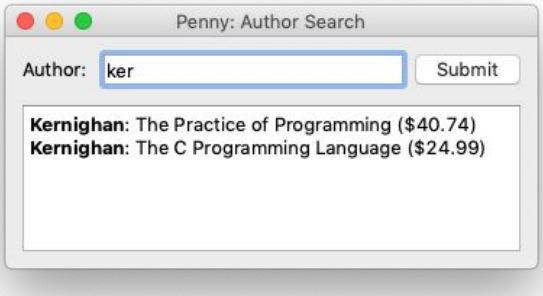
Single Process Server

- See **PennyProcesses1** (cont.)



Single Process Server

- See **PennyProcesses1** (cont.)



After a
5 second
delay

Single Process Server

- See **PennyProcesses1** (cont.)
  - penny.sql, penny.sqlite
  - book.py, database.py
  - **penny.py**
  - **pennyserver.py**

24

Multiprocessing Servers: Motivation

[see slide]

Code notes: penny.py
Traditional PyQt5 program

Code notes: pennyserver.py
Uses only one process

Single Process Server

- Try this:
  - Run pennyserver with delay=5
  - Run 3 penny clients
    - In each, enter “ker”
    - In each, rapidly, click on Submit button
  - Note:
    - pennyserver responds to 1st client after ~5 seconds
    - pennyserver responds to 2nd client after ~10 seconds
    - pennyserver responds to 3rd client after ~15 seconds
- Unacceptable!!!

Single Process Server

- **Problem**
  - Server handles requests **sequentially**
- **Solution**
  - Multiprocessing on server side
  - If multiple CPUs are available, server handles requests in **parallel**

Agenda

- Process-level concurrency
- A single process server
- **A multiprocess server**

Multiprocess Server

- See **PennyProcesses2**
  - penny.sql, penny.sqlite
  - book.py, database.py
  - **pennyserver.py**
  - penny.py

28

Multiprocessing Servers: Example

[see slide]

Code notes: pennyserver.py

Forks a new child process to handle each request

Multiprocess Server

- Try this:
  - Run pennyserver with delay=5
  - Run 3 penny clients
    - In each, enter “ker”
    - In each, rapidly, click on Submit button
  - Note (assuming multiple processors available):
    - pennyserver responds to 1st client after ~5 seconds
    - pennyserver responds to 2nd client after ~5 seconds
    - pennyserver responds to 3rd client after ~5 seconds
- Acceptable!!!

Multiprocess Server: Waiting

- **Problem**
  - Parent process should wait for (reap) child
  - In Penny, parent process must wait for (reap) child **after child exits**

30

Multiprocessing Servers: Waiting

Problem

Parent process should wait for child

Thereby **reaping** the child

Thereby avoiding **zombie** processes

In Penny

Parent process should **not** wait for (reap) child immediately after fork

Parent process should wait for (reap) the child after the child exits

Multiprocess Server: Waiting

- **Solution**
  - Don’t worry about it!
  - In Python, each time a new thread starts, the parent process checks for zombie child processes and waits for (reaps) them

31

Multiprocessing Servers: Waiting

Solution

Don’t worry about it!
In Python, each time a new thread starts, the parent process checks for zombie child processes and reaps them
Typically only a few zombies exist at any given time

(Python on Unix offers a more complete solution)

Summary

- We have covered:
  - What a **process** is
  - How to **fork** and **wait** for processes
  - How to compose a multiprocessing server