6.009
Fundamentals of Programming

Week 13 Lecture:
Concurrency

Adam Hartz
hz@mit.edu

Concurrency

Concurrent Programming involves the creation of programs that can work on more than one thing at a time.

In the olden days, "task switching": a single CPU would do the processing, but would quickly switch back-and-forth between tasks to create the illusion of parallel processing.

Modern processors typically have multiple cores that can be used to *actually* compute things in parallel. But more tasks than CPUs $\rightarrow$ task switching on each CPU.

Many modern programs are written so as to take advantage of the ability to compute things in parallel (*parallel programming* across processes, or *distributed programming* across computers).

Threads

Threads are an abstraction of an independent task running inside a program (i.e., a way to execute a task with its own independent flow of execution (call stack, current instruction, etc)).

Key idea: a thread is like a separate task that runs independently inside your program.

Threads

In Python, threads are defined by a class `threading.Thread`. You can create threads by inheriting from this class and overriding a method called `run`.

```
import time
import threading

class Countdown(threading.Thread):
    def __init__(self, initial_value):
        self.count = initial_value
        threading.Thread.__init__(self)

    def run(self):
        print('Starting the thread!')
        for i in range(self.count):
            print(self.count - i, '...')
            time.sleep(1)
        print('Blastoff!')
```

Threads

To start a thread, create an instance and call its `start` method.

```
c = Countdown(10)
c.start()
```

Threads

Alternatively, we can create threads for calling functions:

```
def countdown(count):
    print('Starting the thread!')
    for i in range(count):
        print(count - i, '...')
        time.sleep(1)
    print('Blastoff!')

t = threading.Thread(target=countdown, args=(20, ))
t.start()
```

Threading

When you `start` a thread, it runs independently.

You can use the `join` method to wait for a thread to exit.

Threading: Timing

Timing between threads is unpredictable
(non-deterministic)!

Threading: A Joke

Why did the multi-threaded chicken cross the road?

Threading: A Joke

Why did the multi-threaded chicken cross the road?

side. To the get other to

Threading: Timing

Is this really a concern?

Threading: Timing

Is this really a concern?

YES!!!

In particular, this is an issue when threads have access to a shared piece of data. Consider an example program.

Threading: Shared Data

```python
x = 0

def adder():
    global x
    for i in range(1000000):
        x += 1

def subber():
    global x
    for i in range(1000000):
        x -= 1

t1 = threading.Thread(target=adder)
t2 = threading.Thread(target=subber)
t1.start(); t2.start()
t1.join(); t2.join()
print(x)
```

What?!

Why does this happen?

In low-level Python code, `adder` looks like:

```
>>> import dis
>>> dis.disco(adder.__code__)
  7           0 SETUP_LOOP              24 (to 26)
              2 LOAD_GLOBAL              0 (range)
              4 LOAD_CONST               1 (1000000)
              6 CALL_FUNCTION            1
              8 GET_ITER
        >>   10 FOR_ITER                12 (to 24)
             12 STORE_FAST               0 (i)

  8          14 LOAD_GLOBAL              1 (x)
             16 LOAD_CONST               2 (1)
             18 INPLACE_ADD
             20 STORE_GLOBAL             1 (x)
             22 JUMP_ABSOLUTE           10
        >>   24 POP_BLOCK
```

Race Conditions

The corruption of shared data due to thread scheduling is called a "race condition."

These kinds of errors can be particularly difficult to debug, primarily due to the nondeterministic nature of thread scheduling (might produce slightly different results each run, or might mysteriously fail only once every few weeks).

Race Conditions

The corruption of shared data due to thread scheduling is called a “race condition.”

These kinds of errors can be particularly difficult to debug, primarily due to the nondeterministic nature of thread scheduling (might produce slightly different results each run, or might mysteriously fail only once every few weeks).

But...is it a *real* concern?

Synchronizing Threads

The `threading` module defines several objects for synchronizing threads. The most commonly used is a "mutex" lock called `Lock`, used to prevent multiple threads from modifying shared data at the same time.

```
m = threading.Lock()
m.acquire()  # acquire the lock (waits for other threads)
# ... DO SOMETHING INVOLVING SHARED STATE
m.release()  # release the lock (so another thread can grab it)
```

Alternatively:

```
m = threading.Lock()
with m:
    # ... DO SOMETHING INVOLVING SHARED STATE
```

Locking

Locking seems like a quick fix, but it can lead to its own set of problems (particularly with non-linear forms of control flow like Exceptions).

Let's take a look at some examples.

Threading

Summary:

- Threads are a nice tool that let us handle multiple tasks in parallel
- BUT! We need to be careful when those multiple threads handle shared state.

Multiprocessing

Because of details of Python's implementation, threads can only run on one CPU (can't take advantage of the fact that modern CPU's can *actually* do computations in parallel).

Python provides a way to take fuller advantage of this, via the `multiprocessing` module.

It has a very similar interface to the `threading` module, but with some important differences. Let's take a look.