Introduction to Threads

Background

- Python is often used in applications where you want to the interpreter to be working on more than one task at once
- Example: An internet server handling hundreds of client connections

Background

- There is also interest in making Python run faster with multiple CPUs

"Can I make Python run 4 times faster on my quad-core desktop?"

"Can I make Python run 100 times faster on our mondo enterprise server?"

- A delicate issue surrounded by tremendous peril

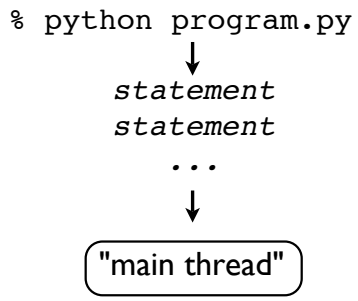
Overview

- In this section, we'll briefly introduce Python thread programming
- Mainly just to see what it looks like
- The devil is in the details (left as an "exercise")

Concept: Threads

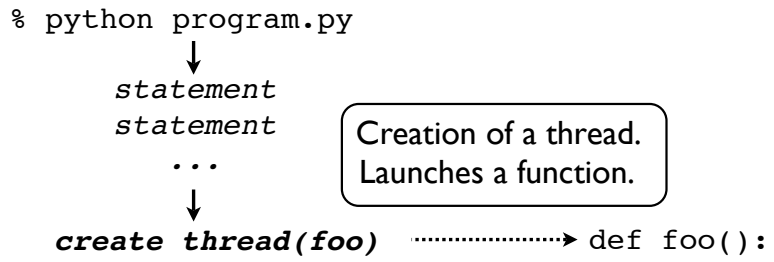
- An independent task running inside a process
- Shares resources with the process (memory, files, network connections, etc.)
- Has own flow of execution (stack, PC)

Thread Basics

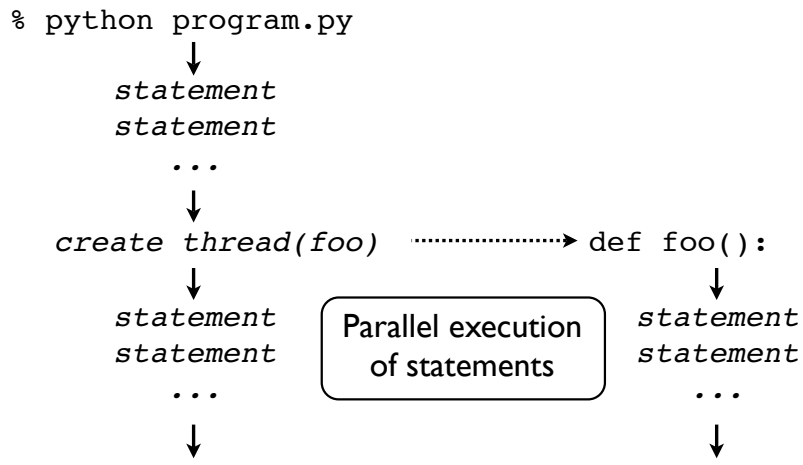


Program launch. Python loads a program and starts executing statements

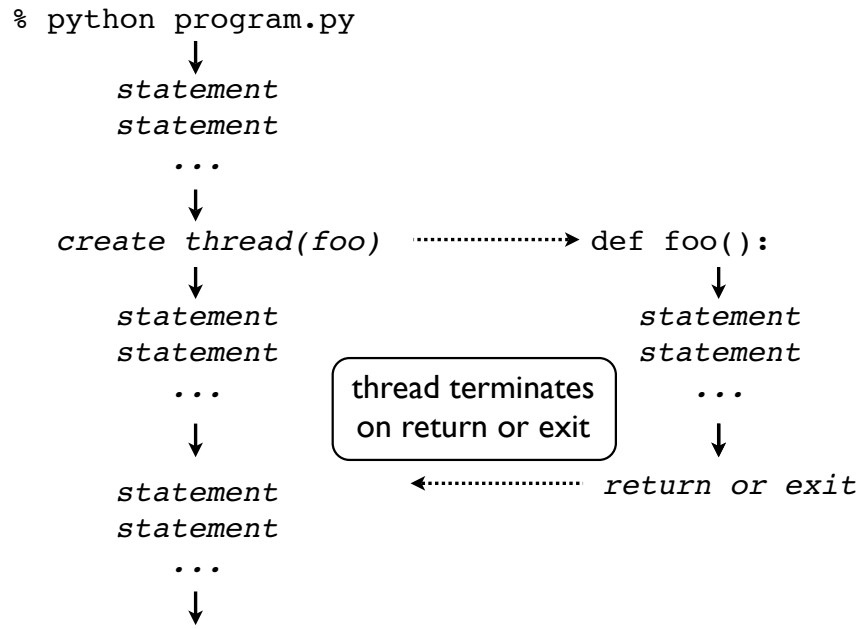
Thread Basics



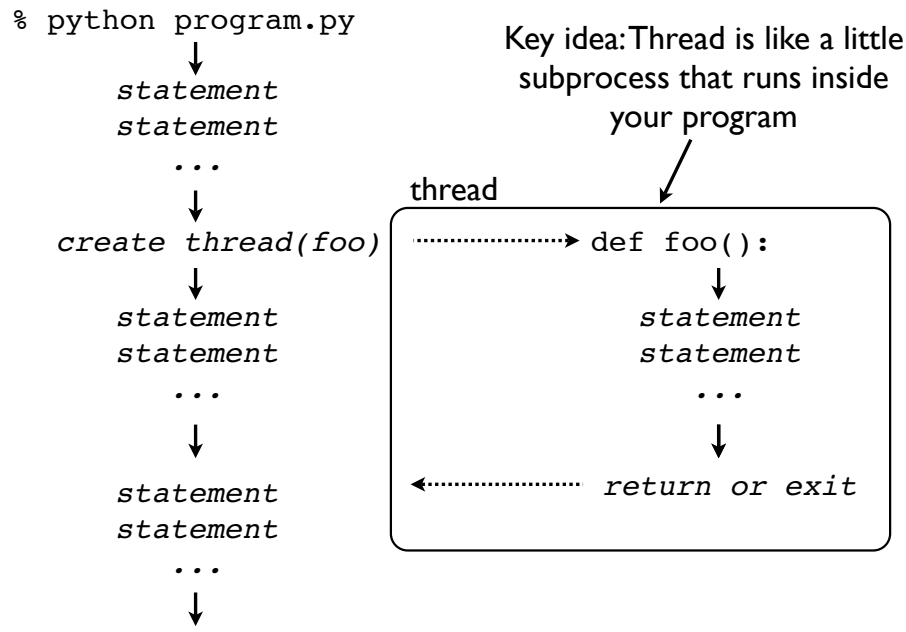
Thread Basics



Thread Basics



Thread Basics



threading module

- Threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

- Inherit from Thread and redefine run()

threading module

- To launch, create objects and use start()

```
t1 = CountdownThread(10)  # Create the thread object
t1.start()                # Launch the thread

t2 = CountdownThread(20)  # Create another thread
t2.start()                # Launch
```

- Threads execute until the run() method stops

Joining a Thread

- Use t.join() to wait for a thread

```
t.start()         # Launch a thread
...
# Do other work
...
# Wait for thread to finish
t.join()          # Waits for thread t to exit
```

- Only works from *other* threads
- A thread can't join itself

Daemonic Threads

- Creating a daemon thread (detached thread)

```
t.setDaemon(True)
```

- Daemon threads run forever
- Can't be joined and is destroyed automatically when the interpreter exits
- Typically used to set up background tasks

Thread Synchronization

- Different threads may share common data
- Extreme care is required
- One thread must not modify data while another thread is reading it
- Otherwise, will get a "race condition"

Race Condition

- Consider a shared object

```
x = 0
```

- And two threads

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

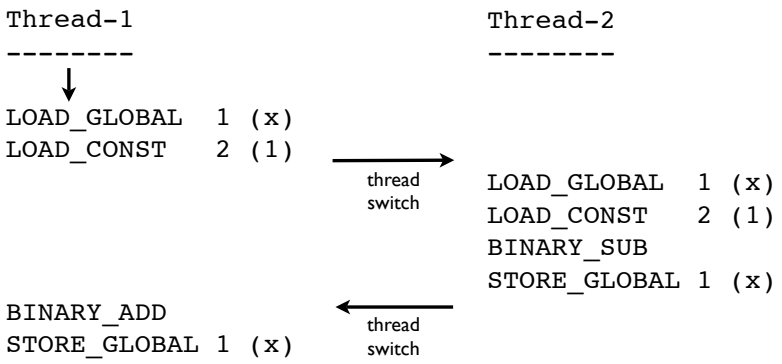
- Possible that the value will be corrupted
- If one thread modifies the value just after the other has read it.

Race Condition

- The two threads

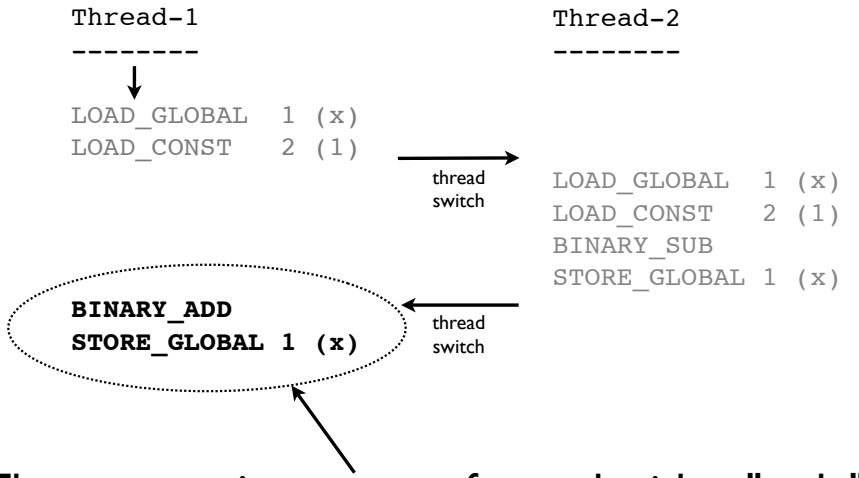
```
Thread-1                    Thread-2
--------                    --------
...                         ...
x = x + 1                   x = x - 1
...                         ...
```

- Low level interpreter execution



Race Condition

- Low level interpreter code



These operations get performed with a "stale" value of x. The computation in Thread-2 is lost.

Mutex Locks

- Mutual exclusion locks

```
m = threading.Lock()      # Create a lock
m.acquire()               # Acquire the lock
m.release()               # Release the lock
```

- Only one thread may hold the lock
- If another thread tries to acquire the lock, it blocks until the lock is released
- Use a lock to make sure only one thread updates shared data at once

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)

Other Locking Primitives

- Reentrant Mutex Lock

```
m = threading.RLock()      # Create a lock
m.acquire()                # Acquire the lock
m.release()                # Release the lock
```

- Can be acquired multiple times by same thread
- Semaphores

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

- Lock based on a counter
- Won't cover in detail here

Thread Programming

- Programming with threads is hell
  - Complex algorithm design
  - Must identify all shared data structures
  - Add locks to critical sections
  - Cross fingers and pray that it works
- Typically you would spend several weeks of a graduate operating systems course covering all of the gory details

The Bad News

- Even if you can get your multithreaded program to work, it won't run any faster
- In fact, it will probably run slower!
- The C Python interpreter itself is single-threaded and protected by a global interpreter lock (GIL)
- Python only utilizes one CPU--even on multi-CPU systems!

Is There a Fix?

- No fix for the GIL is planned
- A big part of the problem concerns reference counting--which is an especially poor memory management strategy for multithreading
- May get true concurrency using Jython or IronPython which are built on JVM/.Net
- C/C++ extensions can also release the GIL

Final Words

- Even though threads are crippled, Python programmers still use them---just not for algorithms involving parallel processing
- Most Python programmers probably encounter threads when working on network server programs (where they can be used to manage multiple client connections)
- More details in an Advanced Python class

A Thread Alternative

- Use message passing
- Multiple independent Python processes (possibly running on different machines) that perform their own processing, but which communicate by sending/receiving messages
- This approach is widely used in supercomputing for massive parallelization (1000s of processors)
- It can also work well for multiple CPU cores if you know what you're doing