Principles of Computer Science II

Concurrency in Python

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 16

John Ousterhout

The best performance improvement is the transition from the nonworking to the working state.

Donald Knuth

Premature optimization is the root of all evil.

Unknown

You can always optimize it later.

Code Optimization vs Concurrency

Python

- A very high-level language
  . . . good at implementing complex systems in much less time
- Code is interpreted
  . . . many programs are "I/O bound"

- Python can be extended with C code
  Examples: ctypes, Cython, Swig, . . .
- If you need really high-performance, you're not coding Python–you're using C extensions
- This is what most of the big scientific computing hackers are doing . . . "using the right tool for the job"

Concurrency

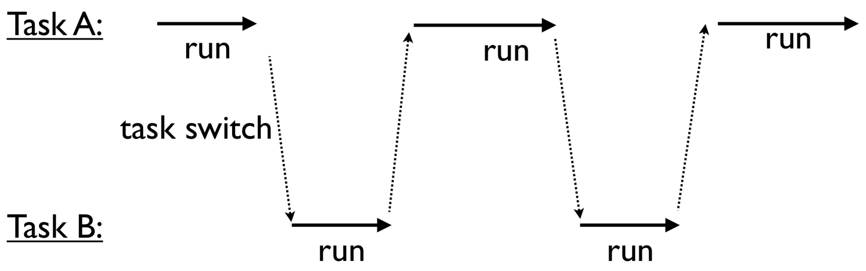
Doing more than one thing at a time. Important when we wish to take advantage of the full capabilities of multicore PCs. Usually a bad idea–except when it's not.

- Main concept of concurrent programming
  - Creation of programs that can work on more than one thing at a time.
  - Example: A network server that communicates with several hundred clients all connected at once
  - Example: A big number crunching job that spreads its work across multiple CPUs
- A look at tradeoffs and limitations
- Introduction to various parts of the standard library

Code Optimization vs Concurrency

- If you're trying to make an inefficient Python script run faster . . . Probably not a good idea
  - marginal improvement of parallelizing a slow script to run on a couple of CPU cores
- Huge gains by focusing on better algorithms
- Huge gains by utilising C extensions

Multitasking

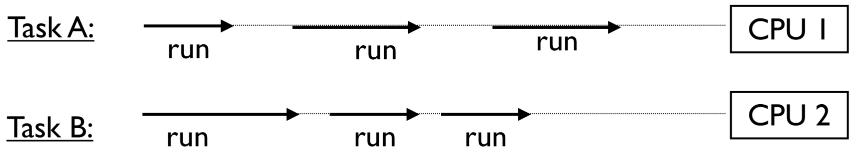


Task A:
run
run
run
task switch
Task B:
run
run

- Concurrency typically implies "multitasking"
- If only one CPU is available, the only way it can run multiple tasks is by rapidly switching between them

---

Parallel Processing

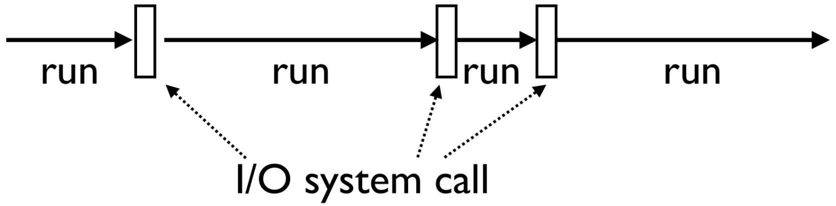


Task A:
run
run
run
CPU 1
Task B:
run
run
run
CPU 2

- You may have parallelism (many CPUs)
- Here, you often get simultaneous task execution
- Note: If the total number of tasks exceeds the number of CPUs, then each CPU also multitasks

---

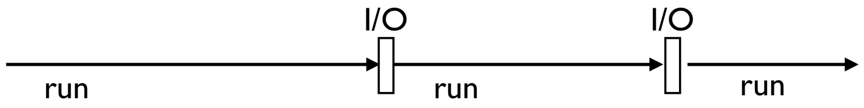
Task Execution



run
run
run
run
I/O system call

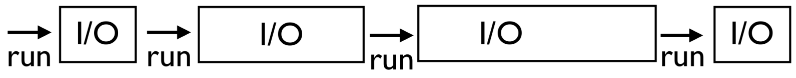
- All tasks execute by alternating between CPU processing and I/O handling
- For I/O, tasks must wait (sleep)
- Behind the scenes, the underlying system will carry out the I/O operation and wake the task when it's finished

CPU Bound Tasks



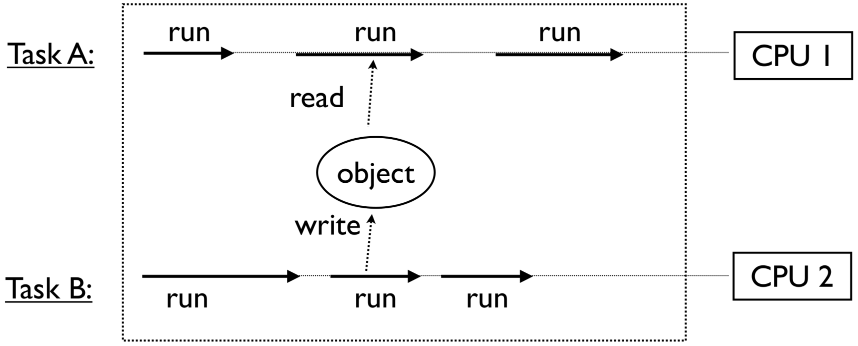
- A task is “CPU Bound” if it spends most of its time processing with little I/O
- Examples
  - Crunching big matrices
  - Image processing

I/O Bound Tasks



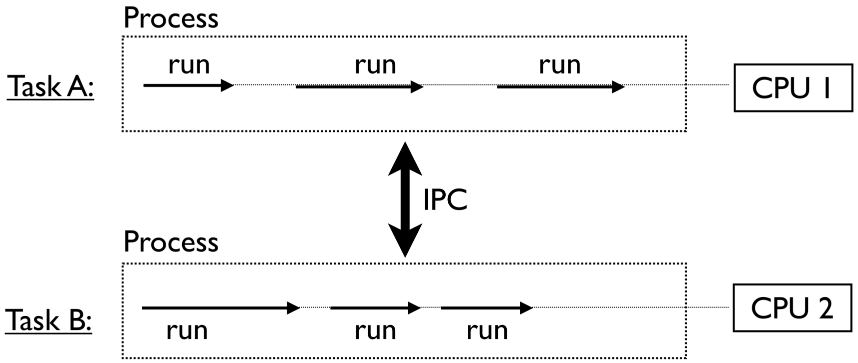
- A task is “I/O Bound” if it spends most of its time waiting for I/O
- Examples
  - Reading input from the user
  - Networking
  - File processing
- Most “normal” programs are I/O bound

Shared Memory



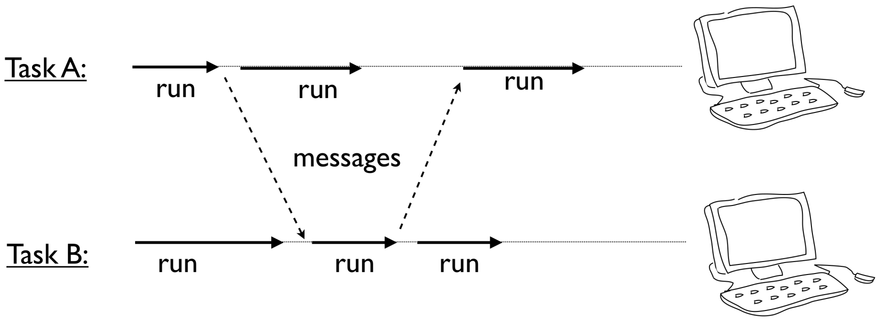
- Tasks may run in the same memory space
- Simultaneous access to objects
- A technically difficult task

Process



- Tasks might run in separate processes
- Processes coordinate using IPC
- Pipes, FIFOs, memory mapped regions, etc.

Distributed Computing

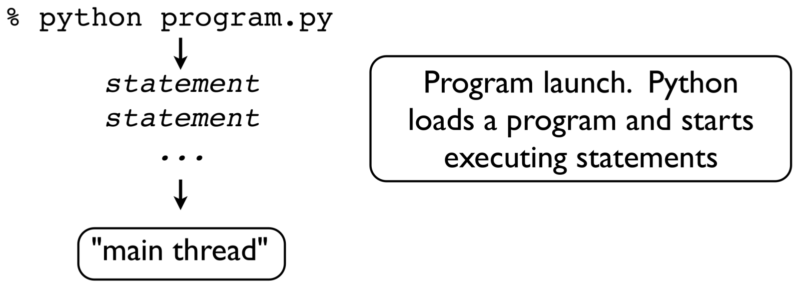


- Tasks may be running on distributed systems
- For example, a cluster of servers
- Communication via sockets

Threads

- Mainstream concurrency programming paradigm
- An independent task running inside a program
- Shares resources with the main program (memory, files, network connections, etc.)
- Has its own independent flow of execution (stack, current instruction, etc.)

A Simple Example



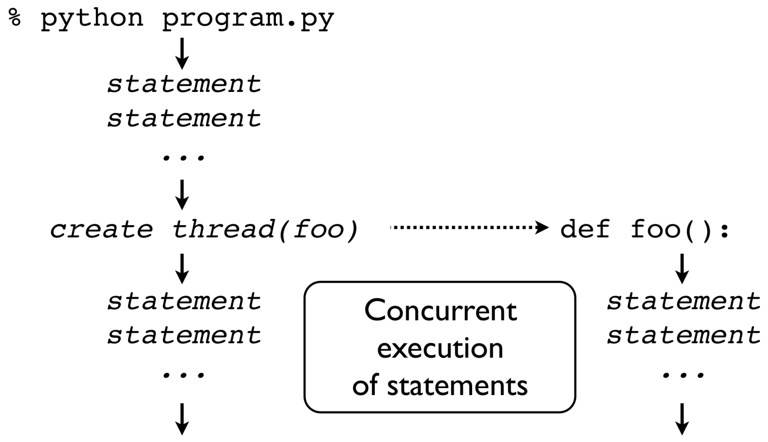
```
% python program.py
```

statement
statement
...

"main thread"

Program launch. Python loads a program and starts executing statements

A Simple Example



```
% python program.py
```

statement
statement
...

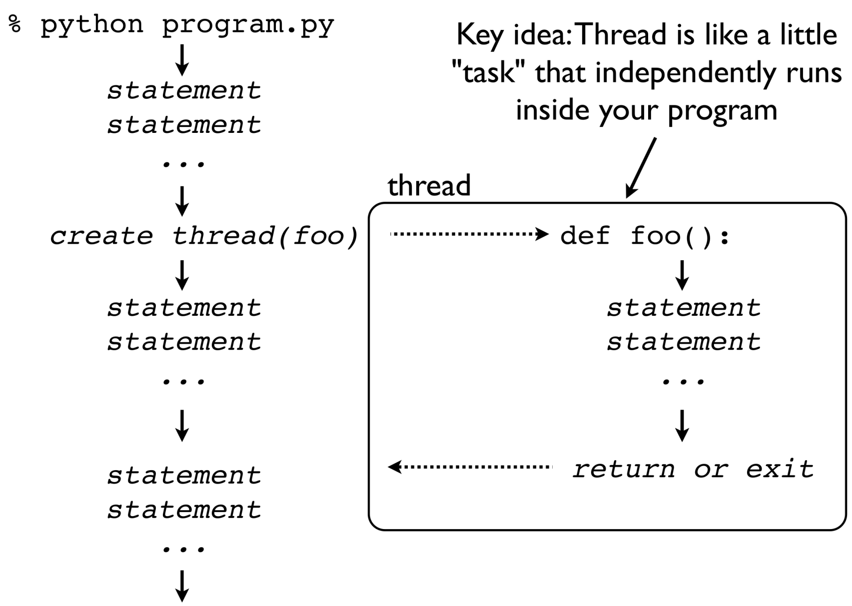
create thread(foo) → `def foo():`

statement
statement
...

statement
statement
...

Concurrent execution of statements

A Simple Example



```
% python program.py
        statement
        statement
           ...
  create thread(foo)  ........>  def foo():
        statement                     statement
        statement                     statement
           ...                           ...
                      <........   return or exit
        statement
        statement
           ...
```

Key idea: Thread is like a little "task" that independently runs inside your program

---

Threading module – definition

- Python threads are defined by a class

```python
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
```

- You inherit from Thread and redefine run()
- The code within the run() function executes in the separate thread

---

Threading module – execution

- To launch, create thread objects and call start()

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

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

Threading module – alternative

- Alternative method of launching thread
- Create a Thread object, but its run() method just calls the given function

```python
def countdown(count):
    while count > 0:
    print "Counting down", count
    count -= 1
    time.sleep(5)

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
```

Threading module – joining a thread

- Once you start a thread, it runs independently
- Use t.join() to wait for a thread to exit

```
t.start() # Launch a thread

...
# Do other work
...

# Wait for thread to finish
t.join() # Waits for thread t to exit
```

- This only works from other threads
- A thread cannot join itself

Threading module – daemonic mode

- If a thread runs forever, make it "daemonic"

```
t.daemon = True
t.setDaemon(True)
```

- If you do not do this, the interpreter will lock when the main thread exits–waiting for the thread to terminate (which never happens)
- Normally you use this for background tasks

Access to Shared Data

- Threads share all of the data in your program
- Thread scheduling is non-deterministic
- Operations often take several steps and might be interrupted mid-stream (non-atomic)
- Thus, access to any kind of shared data is also non-deterministic
- Main reason for errors arising in concurrent programs

An example of shared access

- Consider a shared object

```
x = 0
```

- And two threads that modify it

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

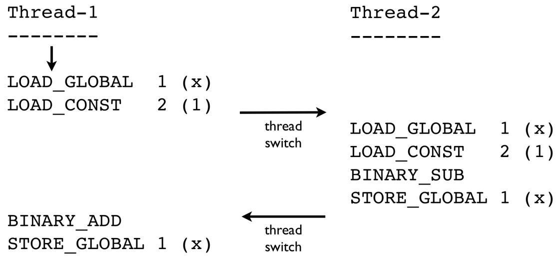
- It's possible that the resulting value will be unpredictably corrupted

An example of shared access

- The two threads

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

- Low level interpreter execution



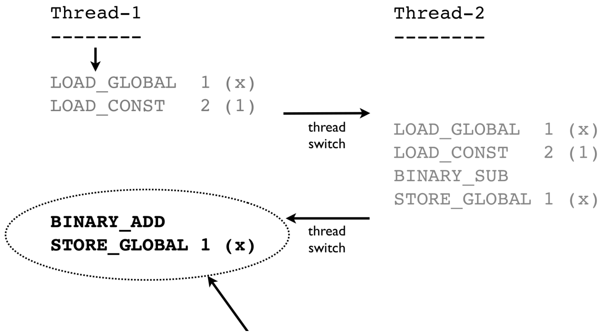
```
Thread-1                    Thread-2
--------                    --------
    ↓
LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)   ──→
                   thread   LOAD_GLOBAL  1 (x)
                   switch   LOAD_CONST   2 (1)
                            BINARY_SUB
                            STORE_GLOBAL 1 (x)
BINARY_ADD           ←──
STORE_GLOBAL 1 (x) thread
                   switch
```

An example of shared access

- The two threads

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

- Low level interpreter execution



```
Thread-1                    Thread-2
--------                    --------
    ↓
LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)   ──→
                   thread   LOAD_GLOBAL  1 (x)
                   switch   LOAD_CONST   2 (1)
                            BINARY_SUB
                            STORE_GLOBAL 1 (x)
BINARY_ADD           ←──
STORE_GLOBAL 1 (x) thread
                   switch
```

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

An example of shared access

- One more example:

```
x = 0                          # A shared value
def foo():
    global x
    for i in xrange(100000000): x += 1

def bar():
    global x
    for i in xrange(100000000): x -= 1

t1 = threading.Thread(target=foo)
t2 = threading.Thread(target=bar)
t1.start(); t2.start()
t1.join(); t2.join()           # Wait for completion
print(x)                       # Expected result is 0
```

- The print produces a random value each time

Race Conditions

- This phenomenon is also known as a "race condition"
- The corruption of shared data due to thread scheduling
- A program may produce slightly different results each time it runs
- Or result may rarely happen . . .
- It depends on the actual CPU, other programs being executed on the sme time, . . .
- Fix: use thread synchronization

Thread Syncronization

The threading library defines the following objects for synchronizing threads

- Lock
- RLock
- Semaphore
- BoundedSemaphore
- Event
- Condition

Mutual Exclusion Lock (Mutex)

```
m = threading.Lock()
```

- Probably the most commonly used synchronization primitive
- Synchronize threads so that only one thread can make modifications to shared data at any given time
- Concurrency is lost
- Only one thread can successfully acquire the lock at any given time
- If another thread tries to acquire the lock when its already in use, it gets blocked until the lock is released

---

Using 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)

Using Mutex Locks

- It is your responsibility to identify and lock all "critical sections"

```
x = 0
x_lock = threading.Lock()

Thread-1                        Thread-2
--------                        --------
...                             ...
x_lock.acquire()                x = x - 1
x = x + 1                       ...
x_lock.release()
...
```

If you use a lock in one place, but not another, then you're missing the whole point. All modifications to shared state must be enclosed by lock acquire()/release().

Lock Management

- Simple mechanism for dealing with locks and critical sections

```
x = 0
x_lock = threading.Lock()

# Critical section
with x_lock:
    statements using x

...
```

- This automatically acquires the lock and releases it when control enters/exits the associated block of statements

---

Locks and Deadlock

- Do not write code that acquires more than one mutex lock at a time

```
x = 0
y = 0
x_lock = threading.Lock()
y_lock = threading.Lock()

# Critical section
with x_lock:
    statements using x
    ...
    with y_lock:
        statements using x and y
        ...
...
```

- This almost invariably ends up creating a program that mysteriously deadlocks