Writing Concurrent Applications in Python

Bastian Venthur

Berlin Institute of Technology

2011-09-14

Outline

Introduction to Concurrency

- Starting and Joining Tasks
- Processes and Threads
- Concurrency Paradigms

Python's threading Module

- Thread Class
- Race Conditions
- Locks
- Starvation and Deadlocks
- Conditions
- Events
- Other Stuff not Covered here

Python's Multiprocessing Module

- Process
- Inter Process Communication
- Queues
- Pipes

What is Concurrency?

- Parallel Computing
- Several computations executing simultaneously
- ... potentially interacting with each other

Why Concurrency?

1970-2005

- CPUs became quicker and quicker every year
- Moore’s Law: The number of transistors [...] doubles approximately every two years.

Why Concurrency?

1970-2005

- CPUs became quicker and quicker every year
- Moore's Law: The number of transistors [...] doubles approximately every two years.

But!

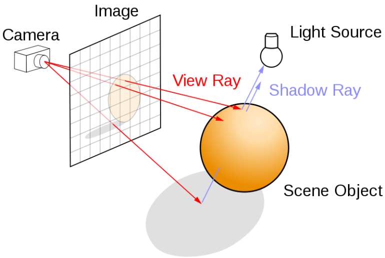
- Physical limits: Miniaturization at atomic levels, energy consumption, heat produced by CPUs, etc.
- Stagnation in CPU clock rates since 2005

Since 2005

Chip producers aimed for more cores instead of higher clock rates.

Useful Applications for Concurrency

Ray Tracing



Trace the path from an imaginary eye (camera) through each pixel in a screen and calculate the color of the object(s) visible through it.

Useful Applications for Concurrency

Ray Tracing

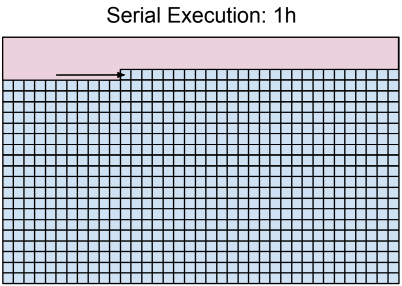


Figure: Ray Tracing performed by one task.

Useful Applications for Concurrency

Ray Tracing

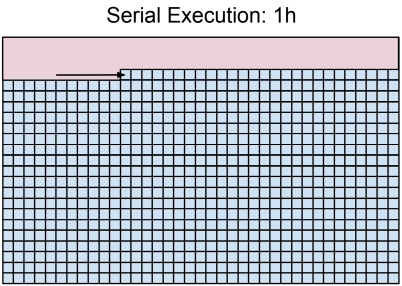


Figure: Ray Tracing performed by one task.

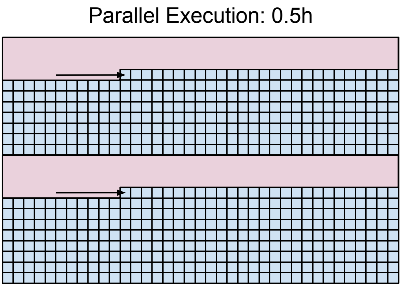


Figure: Ray Tracing performed by two tasks.

Useful Applications for Concurrency

Ray Tracing

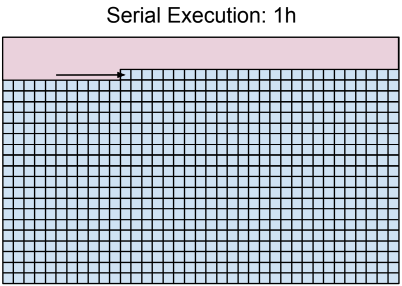


Figure: Ray Tracing performed by one task.

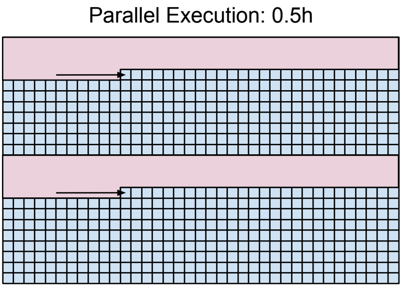


Figure: Ray Tracing performed by two tasks.

Ray Tracing is embarrassingly parallel:

- Little or no effort to separate the problem into parallel tasks
- No dependencies or communication between the tasks

Another Example

Some random calculation

```
L1:    a = 2
L2:    b = 3

L3:    p = a + b
L4:    q = a * b

L5:    r = q - p
```

Another Example

Some random calculation

```
L1:    a = 2
L2:    b = 3

L3:    p = a + b
L4:    q = a * b

L5:    r = q - p
```

- $L1 || L2, \ L3 || L4, \ L5$
- L3 and L4 have to wait for L1 and L2
- L5 has to wait for L3 and L4

Another Example

Some random calculation

```
L1:    a = 2
L2:    b = 3

L3:    p = a + b
L4:    q = a * b

L5:    r = q - p
```

- $L1 || L2,\ L3 || L4,\ L5$
- L3 and L4 have to wait for L1 and L2
- L5 has to wait for L3 and L4

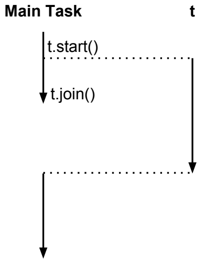
Some synchronization or communication between the tasks is required to solve this calculation correctly. (More on that later)

Getting Started

Starting and Joining a Task

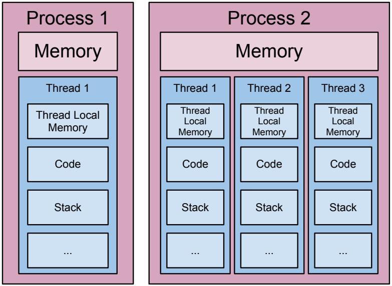
A task is a program or method that runs concurrently.

```
# start task t
# t will run concurrently and the
# (i.e. *this*) program will continue
t = Task()
t.start()
...
# wait for t to finish
t.join()
```



Join synchronises the parent task with the child task by waiting for the child task to terminate.

Two Kinds of Tasks: Threads and Processes



- A process has one or more threads
- Processes have their own memory (Variables, etc.)
- Threads share the memory of the process they belong to
- Threads are also called lightweight processes:
  - They spawn faster than processes
  - Context switches (if necessary) are faster

Communication between Tasks

Shared Memory and Message Passing

Basically you have two paradigms:

1. Shared Memory
   - Taks A and B share some memory
   - Whenever a task modifies a variable in the shared memory, the other task(s) see that change immediately
2. Message Passing
   - Task A sends a message to Task B
   - Task B receives the message and does something with it

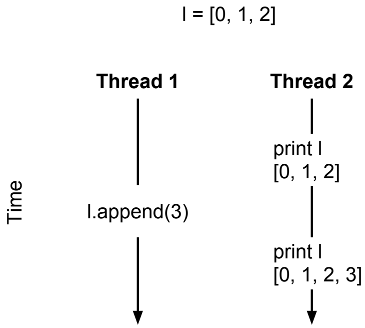
The former paradigm is usually used with threads and the latter one with processes (more on that later).

Outline

- Introduction to Concurrency
  - Starting and Joining Tasks
  - Processes and Threads
  - Concurrency Paradigms
- Python's threading Module
  - Thread Class
  - Race Conditions
  - Locks
  - Starvation and Deadlocks
  - Conditions
  - Events
  - Other Stuff not Covered here
- Python's Multiprocessing Module
  - Process
  - Inter Process Communication
  - Queues
  - Pipes

Threads

They share memory!



Modifying a variable from the processes memory space in one thread immediately affects the corresponding value in the other thread as both variables point to the same address in the process' memory space.

Threads

But they don't share everything.

- Threads have also thread-local memory
- Every variable in this scope is only visible within that thread
- In Python every variable in a thread is thread-local by default.
- Access to a process variable is explicit (e.g. by passing it as an argument to the thread or via `global`)

Python's Thread Class

- Subclass `Thread` class and override `run` method

or Pass callable object to the constructor

- Start thread by calling its `start` method
- Wait for thread to terminate by calling the `join` method

Python's Thread Class

Usage

Subclassing Thread

```
from threading import Thread


# Subclass Thread
class MyThread(Thread):

    def run(self):
        print self.name, "Hello World!"


if __name__ == '__main__':
    threads = []
    # Initialize the threads
    for i in range(10):
        threads.append(MyThread())
    # Start the threads
    for thread in threads:
        thread.start()
    # Wait for threads to terminate
    for thread in threads:
        thread.join()
```

Python's Thread Class

Usage

Subclassing `Thread`

```
from threading import Thread


# Subclass Thread
class MyThread(Thread):

    def run(self):
        print self.name, "Hello World!"


if __name__ == '__main__':
    threads = []
    # Initialize the threads
    for i in range(10):
        threads.append(MyThread())
    # Start the threads
    for thread in threads:
        thread.start()
    # Wait for threads to terminate
    for thread in threads:
        thread.join()
```

Passing callable to the constructor

```
from threading import Thread, current_thread


def run():
    print current_thread().name, "Hello World!"


if __name__ == '__main__':
    threads = []
    # Initialize the threads
    for i in range(10):
        # Pass callable object to the constructor
        threads.append(Thread(target=run, args=()))
    # Start the threads
    for thread in threads:
        thread.start()
    # Wait for threads to terminate
    for thread in threads:
        thread.join()
```

Output...

The above script produces the following output:

```
$ python simplethread.py
Thread-1 Hello World!
Thread-2 Hello World!
Thread-3 Hello World!
Thread-4 Hello World!
Thread-5 Hello World!
Thread-6 Hello World!
Thread-7 Hello World!
Thread-8 Hello World!
Thread-9 Hello World!
Thread-10 Hello World!
```

Output...

The above script produces the following output:

```
$ python simplethread.py
Thread-1 Hello World!
Thread-2 Hello World!
Thread-3 Hello World!
Thread-4 Hello World!
Thread-5 Hello World!
Thread-6 Hello World!
Thread-7 Hello World!
Thread-8 Hello World!
Thread-9 Hello World!
Thread-10 Hello World!
```

... and this one:

```
$ python simplethread.py
Thread-1 Hello World!
Thread-3 Hello World! # <- Sweet!
Thread-2 Hello World!
Thread-4 Hello World!
Thread-5 Hello World!
Thread-6 Hello World!
Thread-7 Hello World!
Thread-8 Hello World!
Thread-9 Hello World!
Thread-10 Hello World!
```

Example

```
import urllib2, time, threading, sys, itertools

HOSTS = ['http://google.com', 'http://yahoo.com', 'http://amazon.com',
         'http://apple.com', 'http://reuters.com', 'http://ibm.com']


class MyThread(threading.Thread):

    def __init__(self, hosts):
        # this line is important!
        threading.Thread.__init__(self)
        self.hosts = hosts

    def run(self):
        for i in itertools.count():
            try:
                host = self.hosts.pop()
            except IndexError:
                break
            url = urllib2.urlopen(host)
            url.read(1024)
        print self.name, "processed %i URLs." % i

if __name__ == '__main__':
    t1 = time.time()
    threads = [MyThread(HOSTS) for i in range(int(sys.argv[1]))]
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
    print 'Elapsed time: %.2fs' % (time.time() - t1)
```

Output...

```
$ python urlfetchthreaded.py 1
Thread—1 processed 6 URLs.
Elapsed time: 4.19s

$ python urlfetchthreaded.py 3
Thread—1 processed 1 URLs.
Thread—2 processed 2 URLs.
Thread—3 processed 3 URLs.
Elapsed time: 1.61s

$ python urlfetchthreaded.py 6
Thread—6 processed 1 URLs.
Thread—3 processed 1 URLs.
Thread—2 processed 1 URLs.
Thread—4 processed 1 URLs.
Thread—5 processed 1 URLs.
Thread—1 processed 1 URLs.
Elapsed time: 1.79s

$ python urlfetchthreaded.py 12
Thread—7 processed 0 URLs.
Thread—8 processed 0 URLs.
Thread—9 processed 0 URLs.
Thread—10 processed 0 URLs.
Thread—11 processed 0 URLs.
Thread—12 processed 0 URLs.
Thread—6 processed 1 URLs.
Thread—3 processed 1 URLs.
Thread—2 processed 1 URLs.
Thread—4 processed 1 URLs.
Thread—5 processed 1 URLs.
Thread—1 processed 1 URLs.
Elapsed time: 1.27s
```

Race Conditions

- Concurrent tasks are cool and now you have the tools to unleash the full power of your multicore system/cluster/supercomputer, but...
- There is one major drawback: you have absolutely no guarantees about the timing when specific parts of your tasks are executed.
- (And there is also the GIL – but more on that later)

Meet the Race Conditions!

Race Conditions

Example

- Your company transfers 2.000 EUR to your account
- Later Ebay charges your account with 1.000 EUR

Time	Thread 1 (your company)	Balance	Thread 2 (ebay)
1	Read Value (10.000)	10.000	
2	Increment Value (12.000)	10.000	
3	Write Value	12.000	
4		12.000	Read Value (12.000)
5		12.000	Decrement Value (11.000)
6		11.000	Write Value

Race Conditions

Same Example - Now a Bit Quicker

Time	Thread 1 (your company)	Balance	Thread 2 (ebay)
1	Read Value (10.000)	10.000	
2	Increment Value (12.000)	10.000	
3		10.000	Read Value (10.000)
4	Write Value	12.000	
5		12.000	Decrement Value (9.000)
6		9.000	Write Value

Race Condition:

- T2 reads the old Value before T1 has written the result
- T2 overwrites the result of T1

Reading, manipulating, and writing Value is a critical section

Critical Section

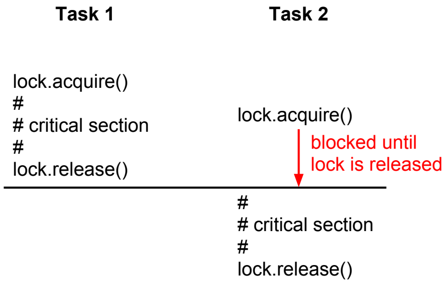
Piece of code that access a shared resource that must not be concurrently accessed by more than one task.

...
Read Ressource
Manipulate Ressource
Write Ressource
...

Critical Section

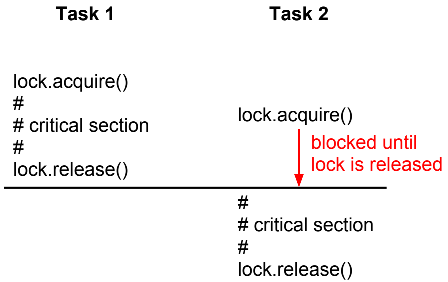
Locks

- Simplest synchronisation primitive
- Two methods: acquire() and release()
- Once acquired, no other task can acquire the same lock until it is released
- At any time, at most one task can hold a lock



Locks

- Simplest synchronisation primitive
- Two methods: acquire() and release()
- Once acquired, no other task can acquire the same lock until it is released
- At any time, at most one task can hold a lock



Warning: Locks may cause Starvation and Deadlocks!

Starvation

- A task is constantly denied necessary resources
- The task can never finish (starves)

Task 1

```
lock.acquire()
...
# never releases lock
```

Task 2

```
lock.acquire()
```

blocked until
lock is released

:(

Deadlock

Figure: Classic deadlock situation as seen in nature.

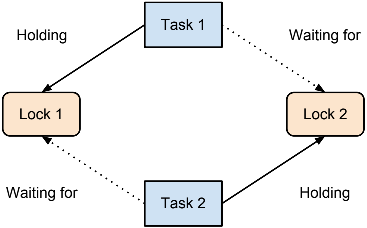


Figure: Classic deadlock situation as seen in Computer Science.

Usually a deadlock occurs when two or more tasks wait cyclically for each other.

Deadlock

Figure: Classic deadlock situation as seen in nature.

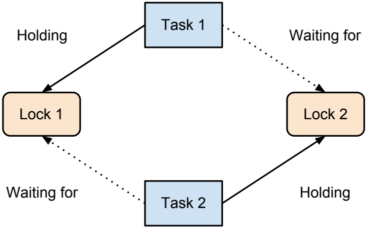


Figure: Classic deadlock situation as seen in Computer Science.

Usually a deadlock occurs when two or more tasks wait cyclically for each other.

One Solution: If a task holds a lock and cannot acquire a second one, release the first one and try again.

Locks in Python

- The lowest synchronisation primitive in Python
- Two methods: `Lock.acquire(blocking=True)` and `Lock.release()`
- A thread calls `acquire()` before entering a critical section and `release()` after leaving
- Other threads that call `acquire()` while the `Lock` is already acquired will wait until it is released (blocking)
- Calling `acquire(False)` makes it non-blocking; the method will return immediately `False` instead of waiting

Locks in Python

Usage:

```
from threading import Lock

...

lock = threading.Lock()
lock.acquire()
# critical section
# ...
# critical section
lock.release()
```

Locks in Python

Usage:

```
from threading import Lock

...

lock = threading.Lock()
lock.acquire()
# critical section
# ...
# critical section
lock.release()
```

Better, using context manager:

```
lock = threading.Lock()
with lock:
    # critical section
    # ...
    # critical section
```

Example

Two threads using the same resource w/o locking

```python
from threading import Thread
import sys
import time


class MyThread(Thread):

    def run(self):
        for i in range(20):
            # we simulate a very long write access
            sys.stdout.write(self.name)
            time.sleep(0.1)
            sys.stdout.write(' Hello World!\n')
            sys.stdout.flush()
            time.sleep(0.1)

if __name__ == '__main__':
    threads = []
    for i in range(2):
        threads.append(MyThread())
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
```

Example

Two threads using the same resource w/o locking

```python
from threading import Thread
import sys
import time


class MyThread(Thread):

    def run(self):
        for i in range(20):
            # we simulate a very long write access
            sys.stdout.write(self.name)
            time.sleep(0.1)
            sys.stdout.write(' Hello World!\n')
            sys.stdout.flush()
            time.sleep(0.1)

if __name__ == '__main__':
    threads = []
    for i in range(2):
        threads.append(MyThread())
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
```

```
Thread-1Thread-2 Hello World!
 Hello World!
Thread-1Thread-2 Hello World!
 Hello World!
Thread-1Thread-2 Hello World!
 Hello World!
Thread-2Thread-1 Hello World!
 Hello World!
Thread-1Thread-2 Hello World!
 Hello World!
Thread-1Thread-2 Hello World!
 Hello World!
Thread-1Thread-2 Hello World!
 Hello World!
...
Thread-2Thread-1 Hello World!
 Hello World!
Thread-2Thread-1 Hello World!
 Hello World!
Thread-2Thread-1 Hello World!
 Hello World!
Thread-2Thread-1 Hello World!
 Hello World!
Thread-2Thread-1 Hello World!
 Hello World!
Thread-2Thread-1 Hello World!
 Hello World!
Thread-2Thread-1 Hello World!
 Hello World!
```

Example

Two threads using the same resource w/ locking

```python
from threading import Thread, Lock
import sys
import time


class MyThread(Thread):

    def __init__(self, lock):
        Thread.__init__(self)
        self.lock = lock

    def run(self):
        for i in range(20):
            with self.lock:
                # we simulate a very long write access
                sys.stdout.write(self.name)
                time.sleep(0.1)
                sys.stdout.write(' Hello World!\n')
                sys.stdout.flush()
            time.sleep(0.1)

if __name__ == '__main__':
    lock = Lock()
    threads = []
    for i in range(2):
        threads.append(MyThread(lock))
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
```

Example

Two threads using the same resource w/ locking

```python
from threading import Thread, Lock
import sys
import time


class MyThread(Thread):

    def __init__(self, lock):
        Thread.__init__(self)
        self.lock = lock

    def run(self):
        for i in range(20):
            with self.lock:
                # we simulate a very long write access
                sys.stdout.write(self.name)
                time.sleep(0.1)
                sys.stdout.write(' Hello World!\n')
                sys.stdout.flush()
            time.sleep(0.1)

if __name__ == '__main__':
    lock = Lock()
    threads = []
    for i in range(2):
        threads.append(MyThread(lock))
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
```

```
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
...
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
Thread-1 Hello World!
Thread-2 Hello World!
```

Conditions

Motivation

- If a precondition for an operation is not fulfilled, wait until notified
- Waiting temporarily releases the lock and blocks until notified

Example:

```
# Consumer thread
condition.acquire()
while not item_available():
    condition.wait()    # temporarily release the lock and sleep
get_an_available_item()
condition.release()

# Producer thread
condition.acquire()
make_an_item_available()
condition.notify()      # wake up a thread waiting
condition.release()
```

Conditions can be implemented using several locks!

Conditions in Python

- Like locks, conditions have `acquire(blocking=True)` and `release()` methods
- Additionally conditions have `wait(timeout=None)`, `notify()`, and `notify_all()` methods
- `wait(timeout=None)` temporarily releases the lock and blocks until notified and the lock is free
- The lock is automatically re-acquired after wait

Example

One Producer/Many Consumers

```
from threading import Condition, Thread, current_thread
import time

def consumer(cond, queue):
    name = current_thread().name    # equivalent to "self.name" when subclassing Thread
    print name, 'acquiring lock.'
    with cond:
        print name, 'acquired lock.'
        while len(queue) == 0:
            print name, 'waiting (released lock).'
            cond.wait()
        print name, 'consumed', queue.pop()
        print name, 'releasing lock.'

def producer(cond, queue):
    for i in range(5):
        print 'Producer: acquiring lock.'
        with cond:
            print 'Producer: acquired lock, producing one item.'
            queue.append(i)
            print 'Producer: notifying.'
            cond.notify()
            print 'Producer: releasing lock.'
        time.sleep(1)

if __name__ == '__main__':
    queue = []
    cond = Condition()
    consumers = [Thread(target=consumer, args=(cond, queue)) for i in range(5)]
    producer = Thread(target=producer, args=(cond, queue))
    producer.start()
    for consumer in consumers:
        consumer.start()
```

Output

One Producer/Many Consumers

```
Producer: acquiring lock.
Thread-2 acquiring lock.
Thread-1 acquiring lock.                         # Producer, T1 and T2 tried to acquire the lock
Thread-2 acquired lock.                          # T2 holds the lock
Thread-2 waiting (released lock).                # Nothing in the queue yet, release lock
Thread-1 acquired lock.
Thread-1 waiting (released lock).                # Same here with T1
Thread-3 acquiring lock.
Thread-4 acquiring lock.
Thread-5 acquiring lock.
Producer: acquired lock, producing one item. # Finally!
Producer: notifying.
Producer: releasing lock.
Thread-3 acquired lock.
Thread-3 consumed 0                              # First item consumed!
Thread-3 releasing lock.
Thread-4 acquired lock.
Thread-4 waiting (released lock).
Thread-5 acquired lock.
Thread-5 waiting (released lock).
Thread-2 waiting (released lock).
Producer: acquiring lock.
Producer: acquired lock, producing one item. # Second item produced
Producer: notifying.
Producer: releasing lock.
Thread-1 consumed 1
Thread-1 releasing lock.
...
```

Events

Motivation

- Several Tasks wait for a specific event
- A task can set the event, waking up all Tasks waiting for that event
- A task can clear the event so other task will block again when waiting for that event

Usage:

```
event = threading.Event()

# thread 1..n wait for an event
event.wait()

# thread x sets or resets the event
event.set()
event.clear()
```

Events can be implemented using Conditions (which can be implemented using locks!)

Stuff not covered here

... but which is still useful

- **RLock** A reentrant lock may be acquired several times by the same thread
- **Semaphore** Like a lock but with a counter
- **Timer** Action that should be run after a certain amount of time has passed
- **Queue** The `Queue` module provides a synchronized queue class (FIFO, LIFO and Priority)

Outline

- Introduction to Concurrency
  - Starting and Joining Tasks
  - Processes and Threads
  - Concurrency Paradigms
- Python's threading Module
  - Thread Class
  - Race Conditions
  - Locks
  - Starvation and Deadlocks
  - Conditions
  - Events
  - Other Stuff not Covered here
- Python's Multiprocessing Module
  - Process
  - Inter Process Communication
  - Queues
  - Pipes

The multiprocessing Module

- Follows closely the `threading` API
- `Process` class has almost the same methods as `Thread` (`run`, `start`, `join`, etc.)
- Contains equivalents of all synchronization primitives from `threading` (`Lock`, `Event`, `Condition`, etc.)

The multiprocessing Module

- Follows closely the `threading` API
- `Process` class has almost the same methods as `Thread` (`run`, `start`, `join`, etc.)
- Contains equivalents of all synchronization primitives from `threading` (`Lock`, `Event`, `Condition`, etc.)

But!

- Processes are not threads!
- Processes do not share memory (i.e. variables)!
  - Synchronization primitives are less important when working with processes
  - Inter Process Communication (IPC) is used for communication

Example

Processes do not share memory!

Similar example like the threaded URL-fetcher:

```
from multiprocessing import Process, current_process
import itertools

ITEMS = [1, 2, 3, 4, 5, 6]

def worker(items):
    for i in itertools.count():
        try:
            items.pop()
        except IndexError:
            break
    print current_process().name, 'processed %i items.' % i


if __name__ == '__main__':
    workers = [Process(target=worker, args=(ITEMS,)) for i in range(3)]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    print 'ITEMS after all workers finished:', ITEMS
```

Example

Processes do not share memory!

Similar example like the threaded URL-fetcher:

```
from multiprocessing import Process, current_process
import itertools

ITEMS = [1, 2, 3, 4, 5, 6]

def worker(items):
    for i in itertools.count():
        try:
            items.pop()
        except IndexError:
            break
    print current_process().name, 'processed %i items.' % i


if __name__ == '__main__':
    workers = [Process(target=worker, args=(ITEMS,)) for i in range(3)]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    print 'ITEMS after all workers finished:', ITEMS
```

Output:

```
Process-1 processed 6 items.
Process-2 processed 6 items.
Process-3 processed 6 items.
ITEMS after all workers finished: [1, 2, 3, 4, 5, 6]
```

Inter Process Communication (IPC)

Pipes and Queues

Pipe

- For communication between two processes
- A Pipe has two ends: process A writes something into his end of the pipe and process B can read it from his
- Pipes are bidirectional

Queue

- Multi-producer, multi-consumer FIFO
- Multiple processes can put items into the Queue, others can get them

Solution

Use multiprocessing.Queue

```python
from multiprocessing import Process, current_process, Queue
import itertools

ITEMS = Queue()
for i in [1, 2, 3, 4, 5, 6, 'end', 'end', 'end']:
    ITEMS.put(i)

def worker(items):
    for i in itertools.count():
        item = items.get()
        if item == 'end':
            break
    print current_process().name, 'processed %i items.' % i


if __name__ == '__main__':
    workers = [Process(target=worker, args=(ITEMS,)) for i in range(3)]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    print '#ITEMS after all workers finished:', ITEMS.qsize()
```

Solution

Use multiprocessing.Queue

```
from multiprocessing import Process, current_process, Queue
import itertools

ITEMS = Queue()
for i in [1, 2, 3, 4, 5, 6, 'end', 'end', 'end']:
    ITEMS.put(i)

def worker(items):
    for i in itertools.count():
        item = items.get()
        if item == 'end':
            break
    print current_process().name, 'processed %i items.' % i


if __name__ == '__main__':
    workers = [Process(target=worker, args=(ITEMS,)) for i in range(3)]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    print '#ITEMS after all workers finished:', ITEMS.qsize()
```

Output:

```
Process-1 processed 1 items.
Process-2 processed 5 items.
Process-3 processed 0 items.
#ITEMS after all workers finished: 0
```

Pipes

- A pipe has two ends: `a, b = Pipe()`
- A process sends something into one end and the other process can recv it on the other
- recv will block if the pipe is empty

Fun Fact

Queues are implemented using Pipes and locks.

Example

```python
from multiprocessing import Process, Pipe

def worker(conn):
    while True:
        item = conn.recv()
        if item == 'end':
            break
        print item

def master(conn):
    conn.send('Is')
    conn.send('this')
    conn.send('on?')
    conn.send('end')

if __name__ == '__main__':
    a, b = Pipe()
    w = Process(target=worker, args=(a,))
    m = Process(target=master, args=(b,))
    w.start()
    m.start()
    w.join()
    m.join()
```

Example

```python
from multiprocessing import Process, Pipe

def worker(conn):
    while True:
        item = conn.recv()
        if item == 'end':
            break
        print item

def master(conn):
    conn.send('Is')
    conn.send('this')
    conn.send('on?')
    conn.send('end')

if __name__ == '__main__':
    a, b = Pipe()
    w = Process(target=worker, args=(a,))
    m = Process(target=master, args=(b,))
    w.start()
    m.start()
    w.join()
    m.join()
```

Output:

```
Is
this
on?
```

Summary
(aka Buzzword Bingo)

Now you know about:

- Concurrent tasks
- Semantics of starting and joining tasks
- Threads and Processes
- Race conditions and critical sections
- Locks, Conditions, Events
- Starvation and Deadlocks
- Pipes and Queues

Fin

PS: In the next lecture you will learn about Python's Global Interpreter Lock (GIL) and how to bypass it.