Concurrency in Python

An overview of multithreading and multiprocessing

Why?


LUDICROUS SPEED GO !!!


MOAR FASTER!!!

But remember…

Premature optimization is the root of all evil.

-Donald Knuth

Problems Concurrency Solves

I/O bound (multithreading & asynchronous)

- Your program is spending more time than you want sending or receiving data.
- Examples: web requests, harddrive i/o, slow API calls, etc.

CPU bound (multiprocessing)

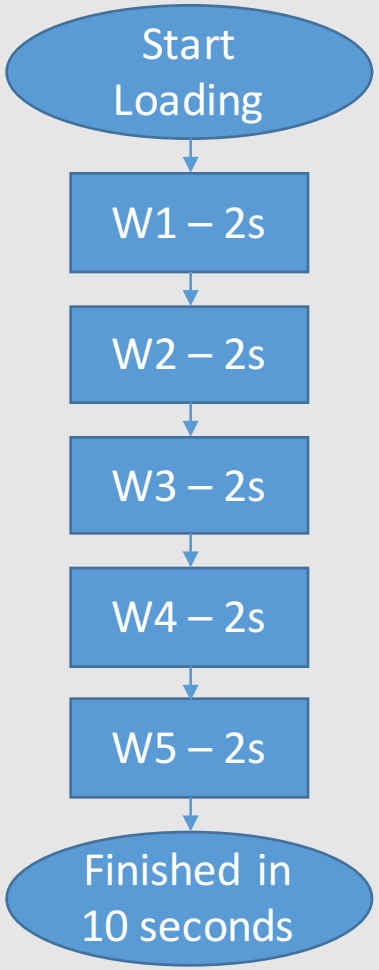
- Your program is spending more time than you want pegging the processor.
- Examples: calculations, image processing, data analysis, etc.

Or both…

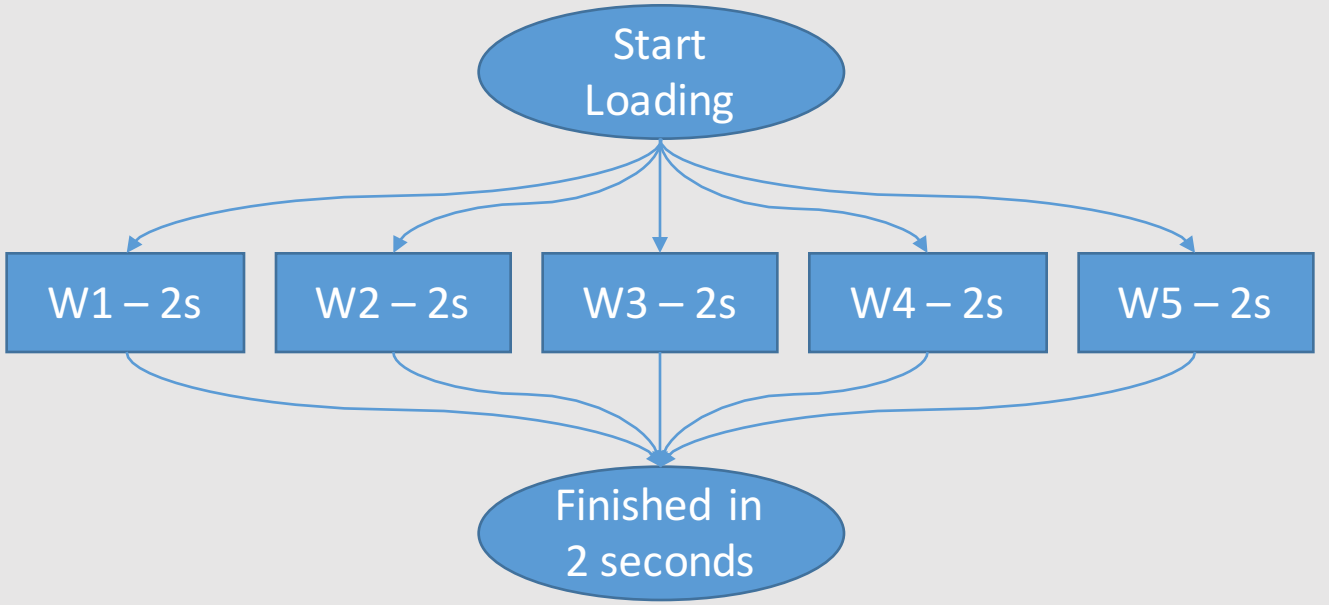
- E.g. loading and analyzing web pages or other data.

Using Multithreading

Simple procedural way:

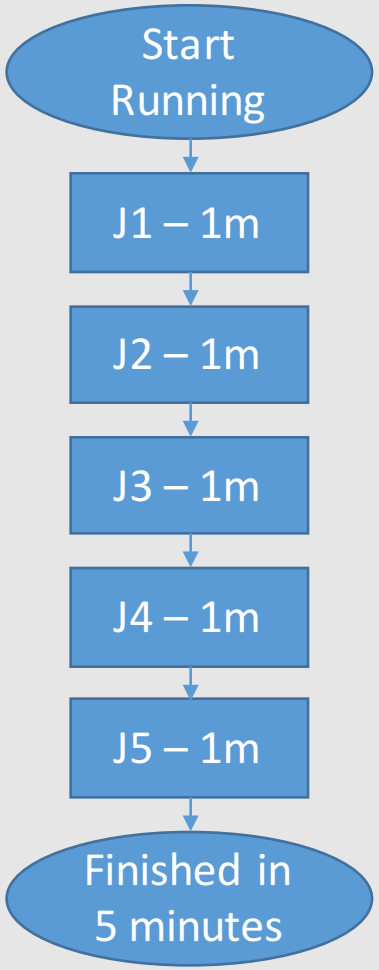


Multithreaded with 5 threads:

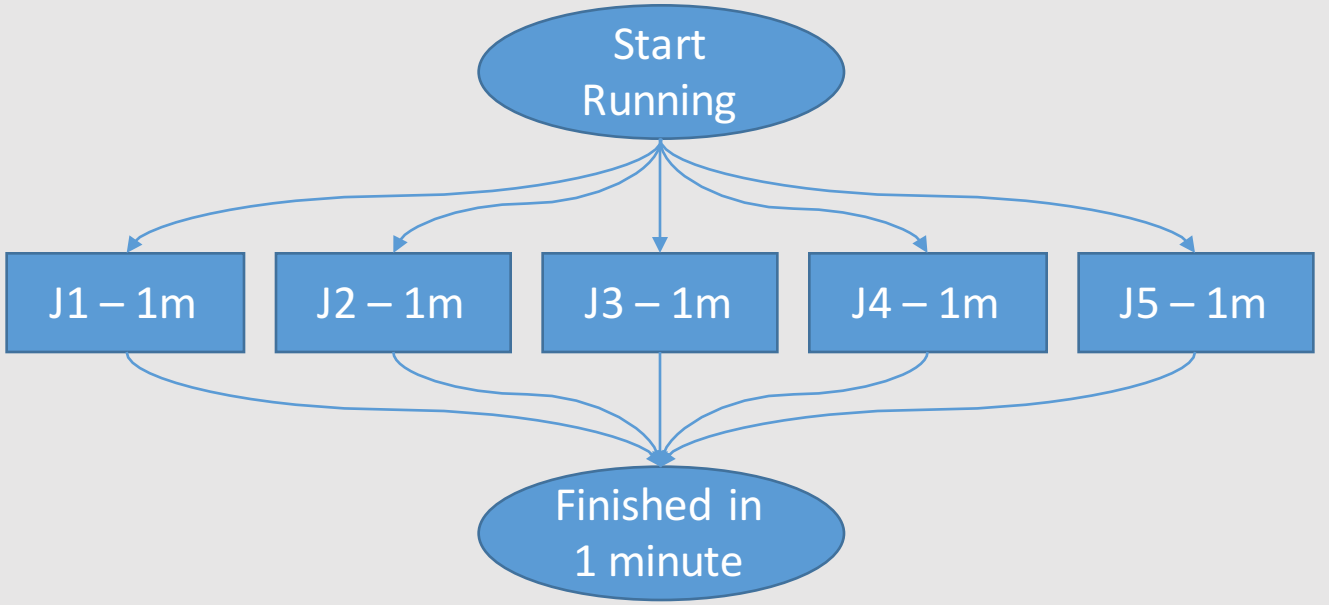


Using Multiprocessing

Simple procedural way:



Multiprocessed with 5 processes and 5 available cores:



Details...

Multithreading

- Threads share global memory.
- Threads have independent stacks.
- Because of the GIL, execution is only logically parallel, not actually.
- Threads can communicate through a variety of methods since they share global memory.

Multiprocessing

- Execution paths are completely independent (they do not share global memory)
- Allows you to use multiple cores and actually run in parallel.
- Communication between processes is more restricted since memory is not shared (pipes, sockets, message queues, explicitly shared memory).

A Worker Queue Example

- In Python multithreading and multiprocessing programs can be written very similarly.
- But there are some important differences…

Multithreading

```python
import threading as tr
import Queue
import time

globalID = None
start = time.time()

# Function to call on every job
def jobThread(jobQueue, threadID):
    global globalID
    while True:
        try:
            job = jobQueue.get_nowait()
        except Queue.Empty:
            break
        else:
            sleeptime = job
            now = time.time() - start
            print("{:.2f}: [{}] read globalID {}, setting to {}".format(
                now, threadID, globalID, threadID))
            globalID = threadID
            time.sleep(sleeptime)
            jobQueue.task_done()

# Function to start the threads
def doWork(jobs, maxworkers=10):
    jobCount = len(jobs)
    jobQueue = Queue.Queue()
    for job in jobs:
        jobQueue.put(job)

    maxworkers = min(maxworkers, jobCount)

    threads = [tr.Thread(target=jobThread, args=(jobQueue,n)) for n in range(maxworkers)]
    for t in threads:
        t.start()

    jobQueue.join()

# create job list
jobList = []
for i in xrange(20):
    jobList.append(1)

doWork(jobList)
print("{:.2f}: All done!".format(time.time() - start))
```

SIMPLE EXAMPLES

Multiprocessing

```python
import multiprocessing as mp
import Queue
import time

globalID = None
start = time.time()

# Function to call on every job
def jobThread(jobQueue, threadID):
    global globalID
    while True:
        try:
            job = jobQueue.get_nowait()
        except Queue.Empty:
            if jobQueue.empty(): break  # get_nowait() can timeout on a non-empty queue
        else:
            sleeptime = job
            now = time.time() - start
            print("{:.2f}: [{}] read globalID={}, setting to {}".format(
                now, threadID, globalID, threadID))
            globalID = threadID
            time.sleep(sleeptime)
            jobQueue.task_done()

# Function to start the processes
def doWork(jobs, maxworkers=10):
    jobCount = len(jobs)
    jobQueue = mp.JoinableQueue()
    for job in jobs:
        jobQueue.put(job)

    maxworkers = min(maxworkers, jobCount)

    processes = [mp.Process(target=jobThread, args=(jobQueue,n)) for n in range(maxworkers)]
    for p in processes:
        p.start()

    jobQueue.join()

# create job list
jobList = []
for i in xrange(20):
    jobList.append(1)

doWork(jobList)
print("{:.2f}: All done!".format(time.time() - start))
```

SIMPLE EXAMPLES OUTPUT

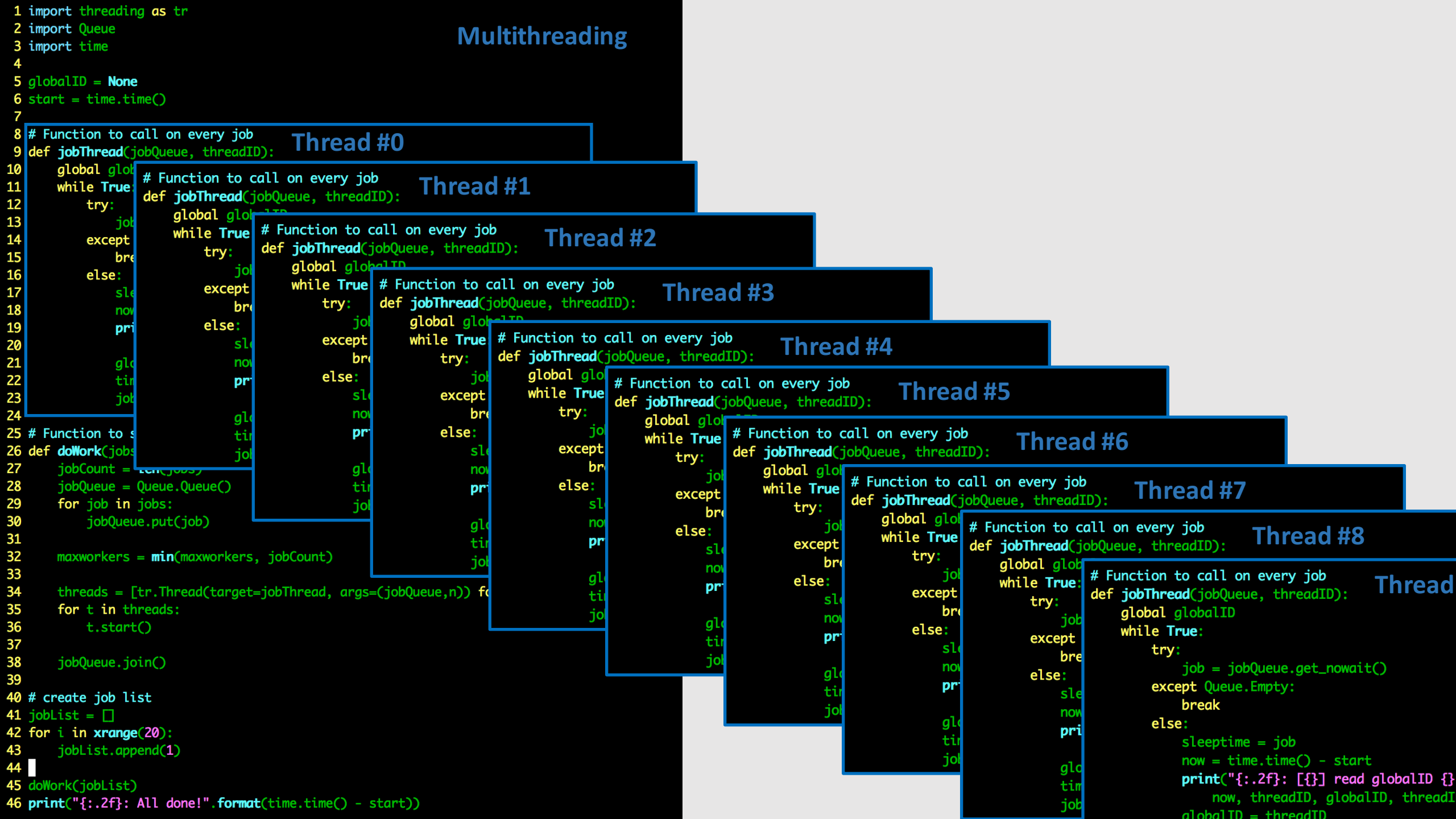
```
thread_example$ python difference_thread.py
0.00: [0] read globalID None, setting to 0
0.00: [1] read globalID 0, setting to 1
0.00: [2] read globalID 1, setting to 2
0.00: [3] read globalID 2, setting to 3
0.00: [4] read globalID 3, setting to 4
 0.00: [5] read globalID 3, setting to 5
 0.00: [6] read globalID 3, setting to 6
0.00: [7] read globalID 5, setting to 7
0.00: [8] read globalID 7, setting to 8
0.00: [9] read globalID 8, setting to 9
1.00: [0] read globalID 9, setting to 0
1.00: [8] read globalID 0, setting to 8
 1.00: [9] read globalID 0, setting to 9
1.00: [2] read globalID 9, setting to 2
1.00: [1] read globalID 2, setting to 1
1.00: [4] read globalID 1, setting to 4
 1.00: [6] read globalID 1, setting to 6
1.00: [5] read globalID 6, setting to 5
1.00: [7] read globalID 5, setting to 7
 1.00: [3] read globalID 5, setting to 3
2.00: All done!
thread_example$
```

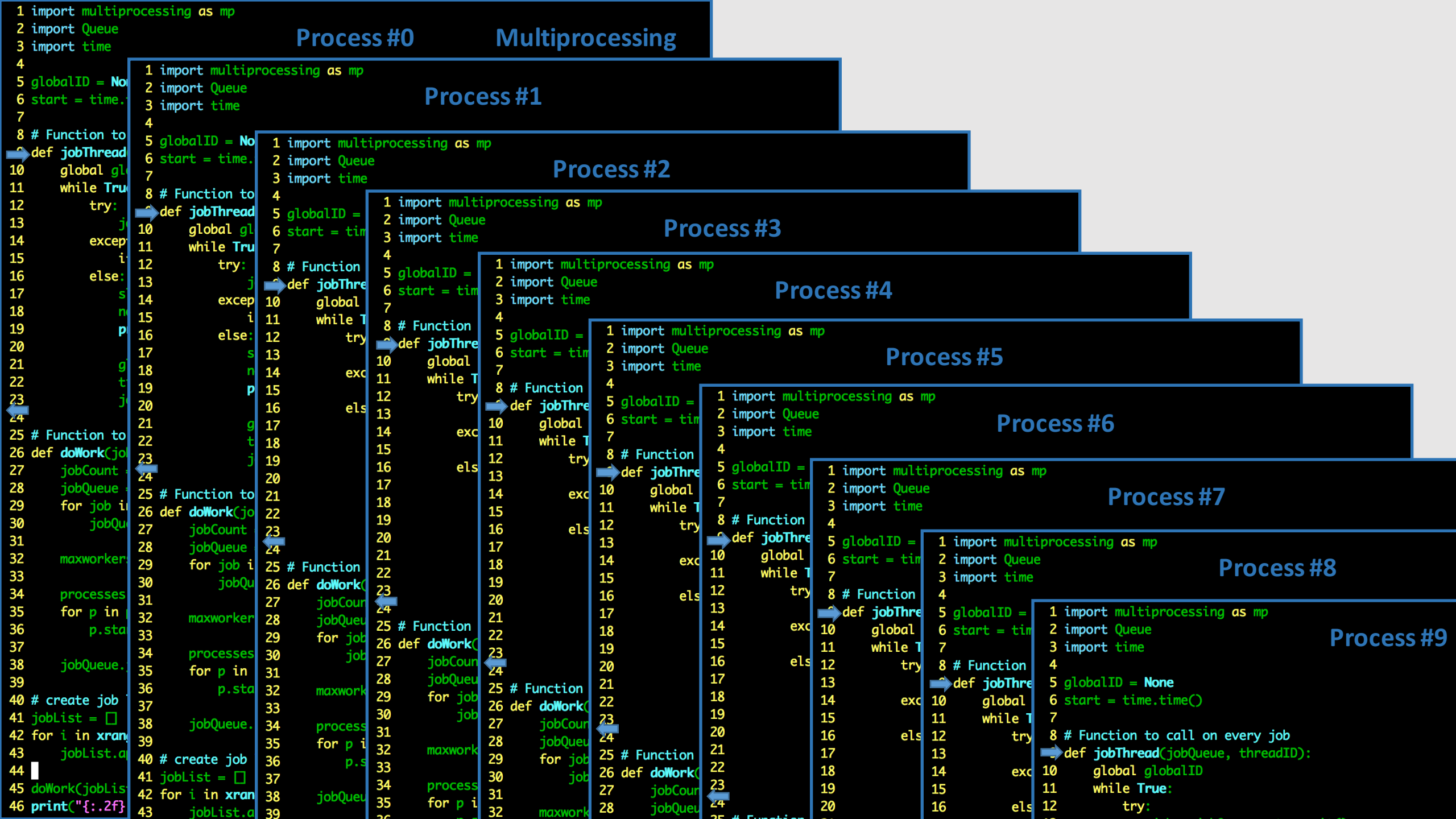
Contention / Races

Multithreading

```
thread_example$ python difference_process.py
0.05: [0] read globalID=None, setting to 0
0.05: [1] read globalID=None, setting to 1
0.05: [2] read globalID=None, setting to 2
0.05: [3] read globalID=None, setting to 3
0.05: [4] read globalID=None, setting to 4
0.05: [5] read globalID=None, setting to 5
0.05: [6] read globalID=None, setting to 6
0.05: [7] read globalID=None, setting to 7
0.05: [8] read globalID=None, setting to 8
0.05: [9] read globalID=None, setting to 9
1.05: [0] read globalID=0, setting to 0
1.05: [1] read globalID=1, setting to 1
1.05: [2] read globalID=2, setting to 2
1.05: [3] read globalID=3, setting to 3
1.05: [4] read globalID=4, setting to 4
1.05: [5] read globalID=5, setting to 5
1.05: [6] read globalID=6, setting to 6
1.05: [7] read globalID=7, setting to 7
1.05: [8] read globalID=8, setting to 8
1.05: [9] read globalID=9, setting to 9
2.06: All done!
thread_example$
```

Multiprocessing


Multithreading
Thread #0
Thread #1
Thread #2
Thread #3
Thread #4
Thread #5
Thread #6
Thread #7
Thread #8
import threading as tr
import Queue
import time
globalID = None
start = time.time()
# Function to call on every job
def jobThread(jobQueue, threadID):
global globalID
while True:
try:
job = jobQueue.get_nowait()
except Queue.Empty:
break
else:
sleeptime = job
now = time.time() - start
print("{:.2f}: [{}] read globalID {}
Function to
def doWork(jobs
jobCount =
jobQueue = Queue.Queue()
for job in jobs:
jobQueue.put(job)
maxworkers = min(maxworkers, jobCount)
threads = [tr.Thread(target=jobThread, args=(jobQueue,n)) fo
for t in threads:
t.start()
jobQueue.join()
# create job list
jobList = []
for i in xrange(20):
jobList.append(1)
doWork(jobList)
print("{:.2f}: All done!".format(time.time() - start))


Process #0
Multiprocessing
Process #1
Process #2
Process #3
Process #4
Process #5
Process #6
Process #7
Process #8
Process #9
import multiprocessing as mp
import Queue
import time
globalID = None
start = time.time()
# Function to call on every job
def jobThread(jobQueue, threadID):
global globalID
while True:
try:
# Function to
def doWork(jo
jobCount
jobQueue
for job i
jobQu
maxworker
processes
for p in
p.sta
jobQueue.
# create job
jobList = []
for i in xran
jobList.a
doWork(jobLis
print("{:.2f}


Y U NO FASTER?!?!


Suck! Suck! Suck!

Problems Concurrency Creates

- Adds complexity:
  - Can be much more difficult to debug.
  - Can have unknown bugs and race conditions.
- Switching between multithreading and multiprocessing can be confusing b/c of the different memory scopes.
- Proper testing is more difficult (edge cases, etc) and even impractical.
- Can hurt performance (e.g. multithreading CPU bound code).
- Exceptions can disappear if you don’t catch them.
- Program may not die normally, may need a SIGTERM or SIGKILL.

How To…

1. Write a mostly usable modular program without concurrency.
2. Find the bottlenecks in performance (I/O or CPU bound).
3. Fix the worst bottlenecks by adding concurrency where appropriate.
4. Repeat from step 2 until you are satisfied with the performance or there is nothing left to optimize (within reason).

- Avoid optimizing things that don’t need to be optimized (see Knuth).

MOAR Details...

Communication

- Multithreading:
  - Global variables (not ideal)
  - Shared variables (i.e. thread function parameters)
  - Queues

- Multiprocessing:
  - Multiprocessing.Queue
  - Multiprocessing.Pipe
  - UNIX sockets or TCP sockets
  - Shared memory (memory mapped files)
  - Windows and Linux message queues (OS persistence)

Locks

- Safely control access to global or shared variables.
- Use ‘with’!
- If a module or class it ‘threadsafe’, it does that for you (like Queue).

```
1  import threading as tr
2  import time
3
4  globalDict = {}  # Be very careful with global mutable objects
5  globalDictLock = tr.RLock() # Just use RLock, don't deadlock yourself
6  shutdown = False
7
8  def taskThread1(taskparams):
9      while not shutdown:
10         # ... do task ...
11         state = 'state1'
12         with globalDictLock:
13             globalDict['task1'] = state
14         time.sleep(0.1)
15
16
17 def taskThread2(taskparams):
18     while not shutdown:
19         # ... do task ...
20         state = 'state2'
21         with globalDictLock:
22             globalDict['task2'] = state
23         time.sleep(0.1)
```

Termination Requests

- Python doesn’t kill children
- Use signal for clean exits

```
1 import signal
2 import time
3
4 shutdown = False
5
6 def sighandler(signum, frame):
7     global shutdown
8     print("Detected a shutdown request with signal {}".format(signum))
9     shutdown = True
10
11 signal.signal(signal.SIGTERM, sighandler)
12 signal.signal(signal.SIGINT, sighandler)
13 signal.signal(signal.SIGHUP, sighandler)
14
15 while not shutdown:
16     time.sleep(0.1)
```

External Modules

- Can be helpful but can also add more unnecessary complexity.
- E.g. I tried using grequests, but for my use case it was much more effort, adding numerous callbacks and conditions… A simple threaded worker function gave me total control of the threads.

Resources

- The Little Book of Semaphores (free) - http://www.greenteapress.com/semaphores/
- https://docs.python.org/2/library/threading.html (or 3.5)
- https://docs.python.org/2/library/multiprocessing.html (or 3.5)