Aula 8 - Multiprocessamento

Objetivos

- Apresentar o pacote de multiprocessamento.
- Comparar threads com processos.

```
import multiprocessing

def worker():
    """worker function"""
    print ('Worker‘)
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p =
multiprocessing.Process(target=worker)
        jobs.append(p)
        p.start()
        p.join()
```

```
$ python
multiprocessing_simple.py

Worker
Worker
Worker
Worker
Worker
```

```
import multiprocessing

def worker(num):
    """thread worker function"""
    print ('Worker:', num)
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker,
args=(i,))
        jobs.append(p)
        p.start()
```

```
$ python
multiprocessing_simpleargs.
py

Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

```
import multiprocessing
import multiprocessing_import_worker

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=multiprocessing_import_worker.worker)
        jobs.append(p)
        p.start()
```

```
def worker():
    """worker function"""
    print 'Worker'
    return
```

Determinar o processo corrente

```
import multiprocessing
import time

def worker():
    name = multiprocessing.current_process().name
    print name, 'Starting'
    time.sleep(2)
    print name, 'Exiting'

def my_service():
    name = multiprocessing.current_process().name
    print name, 'Starting'
    time.sleep(3)
    print name, 'Exiting'

if __name__ == '__main__':
    service = multiprocessing.Process(name='my_service',
target=my_service)
    worker_1 = multiprocessing.Process(name='worker 1', target=worker)
    worker_2 = multiprocessing.Process(target=worker) # use default
name

    worker_1.start()
```

```
$ python
multiprocessing_names.py

worker 1 Starting
worker 1 Exiting
Process-3 Starting
Process-3 Exiting
my_service Starting
my_service Exiting
```

Processos Daemon

```
import multiprocessing
import time
import sys

def daemon():
    p = multiprocessing.current_process()
    print 'Starting:', p.name, p.pid
    sys.stdout.flush()
    time.sleep(2)
    print 'Exiting :', p.name, p.pid
    sys.stdout.flush()

def non_daemon():
    p = multiprocessing.current_process()
    print 'Starting:', p.name, p.pid
    sys.stdout.flush()
    print 'Exiting :', p.name, p.pid
    sys.stdout.flush()
```

```
if __name__ == '__main__':
    d = multiprocessing.Process(name='daemon', 
target=daemon)
    d.daemon = True

    n = multiprocessing.Process(name='non-daemon', 
target=non_daemon)
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()
```

```
$ python
multiprocessing_daemon.py

Starting: daemon 13866
Starting: non-daemon 13867
Exiting : non-daemon 13867
```

```
import multiprocessing
import time
import sys

def daemon():
    print 'Starting:', multiprocessing.current_process().name
    time.sleep(2)
    print 'Exiting :', multiprocessing.current_process().name

def non_daemon():
    print 'Starting:', multiprocessing.current_process().name
    print 'Exiting :', multiprocessing.current_process().name
```

```
if __name__ == '__main__':
    d = multiprocessing.Process(name='daemon',
target=daemon)
    d.daemon = True

    n = multiprocessing.Process(name='non-daemon',
target=non_daemon)
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()

    d.join()
    n.join()
```

```
$ python multiprocessing_daemon_join.py

Starting: non-daemon
Exiting : non-daemon
Starting: daemon
Exiting : daemon
```

Terminar processos

```
import multiprocessing
import time

def slow_worker():
    print 'Starting worker'
    time.sleep(0.1)
    print 'Finished worker'

if __name__ == '__main__':
    p = multiprocessing.Process(target=slow_worker)
    print 'BEFORE:', p, p.is_alive()

    p.start()
    print 'DURING:', p, p.is_alive()

    p.terminate()
    print 'TERMINATED:', p, p.is_alive()

    p.join()
    print 'JOINED:', p, p.is_alive()
```

```
$ python multiprocessing_terminate.py

BEFORE: <Process(Process-1, initial)> False
DURING: <Process(Process-1, started)> True
TERMINATED: <Process(Process-1, started)>
True
JOINED: <Process(Process-1,
stopped[SIGTERM])> False
```

Status de saída de processos

- ==0 sem erros
- >0   o processo tem um erro (o valor é o código do erro)
- <0   o processo foi morto com um sinal -1* código

```
import multiprocessing
import sys
import time

def exit_error():
    sys.exit(1)

def exit_ok():
    return

def return_value():
    return 1

def raises():
    raise RuntimeError('There was an error!')

def terminated():
    time.sleep(3)
```

```
if __name__ == '__main__':
    jobs = []
    for f in [exit_error, exit_ok, return_value, raises,
terminated]:
        print 'Starting process for', f.func_name
        j = multiprocessing.Process(target=f,
name=f.func_name)
        jobs.append(j)
        j.start()

    jobs[-1].terminate()

    for j in jobs:
        j.join()
        print '%s.exitcode = %s' % (j.name, j.exitcode)
```

```
$ python multiprocessing_exitcode.py

Starting process for exit_error
Starting process for exit_ok
Starting process for return_value
Starting process for raises
Starting process for terminated
Process raises:
Traceback (most recent call last):
  File
"/Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/multiprocessing/process.py", line 258, in _bootstrap
    self.run()
  File
"/Library/Frameworks/Python.framework/Versions/2.7/lib/python
2.7/multiprocessing/process.py", line 114, in run
    self._target(*self._args, **self._kwargs)
  File "multiprocessing_exitcode.py", line 24, in raises
    raise RuntimeError('There was an error!')
RuntimeError: There was an error!
exit_error.exitcode = 1
exit_ok.exitcode = 0
return_value.exitcode = 0
raises.exitcode = 1
```

- **Multiprocessing**
- Pros
- Separate memory space
- Code is usually straightforward
- Takes advantage of multiple CPUs & cores
- Avoids GIL limitations for cPython
- Eliminates most needs for synchronization primitives unless if you use shared memory (instead, it's more of a communication model for IPC)
- Child processes are interruptible/killable
- Python multiprocessing module includes useful abstractions with an interface much like threading.Thread
- A must with cPython for CPU-bound processing
- Cons
- IPC a little more complicated with more overhead (communication model vs. shared memory/objects)
- Larger memory footprint

- **Threading**
- Pros
- Lightweight - low memory footprint
- Shared memory - makes access to state from another context easier
- Allows you to easily make responsive UIs
- cPython C extension modules that properly release the GIL will run in parallel
- Great option for I/O-bound applications
- Cons
- cPython - subject to the GIL
- Not interruptible/killable
- If not following a command queue/message pump model (using the Queue module), then manual use of synchronization primitives become a necessity (decisions are needed for the granularity of locking)
- Code is usually harder to understand and to get right - the potential for race conditions increases dramatically