Sincronização e comunicação entre Processos

Aula 9

Objetivos

- Logging
- Comunicação e sincronização entre processos

Logging

```
import multiprocessing
import logging
import sys

def worker():
    print ('Doing some work')
    sys.stdout.flush()

if __name__ == '__main__':

multiprocessing.log_to_stderr(logging.DEBUG)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

```
$ python multiprocessing_log_to_stderr.py

[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[DEBUG/Process-1] running all "atexit" finalizers with priority >= 0
[DEBUG/Process-1] running the remaining "atexit" finalizers
[INFO/Process-1] process exiting with exitcode 0
[INFO/MainProcess] process shutting down
[DEBUG/MainProcess] running all "atexit" finalizers with priority
>= 0
[DEBUG/MainProcess] running the remaining "atexit" finalizers
```

```
import multiprocessing
import logging
import sys

def worker():
    print ('Doing some work‘)
    sys.stdout.flush()

if __name__ == '__main__':
    multiprocessing.log_to_stderr()
    logger = multiprocessing.get_logger()
    logger.setLevel(logging.INFO)
    p =
multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

```
$ python multiprocessing_get_logger.py

[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[INFO/Process-1] process exiting with exitcode 0
[INFO/MainProcess] process shutting down
```

Subclasse de processo

```
import multiprocessing

class Worker(multiprocessing.Process):

    def run(self):
        print ('In %s' % self.name)
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = Worker()
        jobs.append(p)
        p.start()
    for j in jobs:
        j.join()
```

```
$ python
multiprocessing_subclass.py

In Worker-1
In Worker-2
In Worker-3
In Worker-4
In Worker-5
```

Passar mensagens para Processos

```
import multiprocessing

class MyFancyClass(object):

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print ('Doing something fancy in %s for %s!' % (proc_name,
self.name))


def worker(q):
    obj = q.get()
    obj.do_something()
```

```
if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

```
$ python multiprocessing_queue.py

Doing something fancy in Process-1 for Fancy Dan!
```

```
import multiprocessing
import time

class Consumer(multiprocessing.Process):

    def __init__(self, task_queue, result_queue):
        multiprocessing.Process.__init__(self)
        self.task_queue = task_queue
        self.result_queue = result_queue

    def run(self):
        proc_name = self.name
        while True:
            next_task = self.task_queue.get()
            if next_task is None:
                # Poison pill means shutdown
                print ('%s: Exiting' % proc_name)
                self.task_queue.task_done()
                break
            print ('%s: %s' % (proc_name, next_task))
            answer = next_task()
            self.task_queue.task_done()
            self.result_queue.put(answer)
        return
```

```
class Task(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __call__(self):
        time.sleep(0.1) # pretend to take some time to do the work
        return '%s * %s = %s' % (self.a, self.b, self.a * self.b)
    def __str__(self):
        return '%s * %s' % (self.a, self.b)
```

```
if __name__ == '__main__':
    # Establish communication queues
    tasks = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()

    # Start consumers
    num_consumers = multiprocessing.cpu_count() * 2
    print ('Creating %d consumers' % num_consumers)
    consumers = [ Consumer(tasks, results)
                  for i in range(num_consumers) ]
    for w in consumers:
        w.start()
```

```
# Enqueue jobs
   num_jobs = 10
   for i in range(num_jobs):
       tasks.put(Task(i, i))

   # Add a poison pill for each consumer
   for i in range(num_consumers):
       tasks.put(None)

   # Wait for all of the tasks to finish
   tasks.join()

   # Start printing results
   while num_jobs:
       result = results.get()
       print ('Result:', result)
       num_jobs -= 1
```

```
Creating 16 consumers
Consumer-1: 0 * 0
Consumer-2: 1 * 1
Consumer-3: 2 * 2
Consumer-4: 3 * 3
Consumer-5: 4 * 4
Consumer-6: 5 * 5
Consumer-7: 6 * 6
Consumer-8: 7 * 7
Consumer-9: 8 * 8
Consumer-10: 9 * 9
Consumer-11: Exiting
Consumer-12: Exiting
Consumer-13: Exiting
Consumer-14: Exiting
Consumer-15: Exiting
Consumer-16: Exiting
Consumer-1: Exiting
Consumer-4: Exiting
Consumer-5: Exiting
Consumer-6: Exiting
Consumer-2: Exiting
Consumer-3: Exiting
Consumer-9: Exiting
Consumer-7: Exiting
Consumer-8: Exiting
Consumer-10: Exiting
Result: 0 * 0 = 0
Result: 3 * 3 = 9
Result: 8 * 8 = 64
Result: 5 * 5 = 25
Result: 4 * 4 = 16
Result: 6 * 6 = 36
Result: 7 * 7 = 49
Result: 1 * 1 = 1
Result: 2 * 2 = 4
Result: 9 * 9 = 81
```

Sinalização entre processos

```
import multiprocessing
import time

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print ('wait_for_event: starting‘)
    e.wait()
    print 'wait_for_event: e.is_set()->', e.is_set()

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    print ('wait_for_event_timeout: starting‘)
    e.wait(t)
    print ('wait_for_event_timeout: e.is_set()->', e.is_set())
```

```
if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(name='block',
                                 target=wait_for_event,
                                 args=(e,))
    w1.start()

    w2 = multiprocessing.Process(name='non-block',
                                 target=wait_for_event_timeout,
                                 args=(e, 2))
    w2.start()

    print 'main: waiting before calling Event.set()'
    time.sleep(3)
    e.set()
    print 'main: event is set'
```

```
$ python -u multiprocessing_event.py

main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is set
wait_for_event: e.is_set()-> True
```

Exercício

- Faça um programa para comparar o desempenho de threads e processos. Crie uma situação onde usar thread é mais vantajoso que processos e vice-versa.