Multiprocessing

-- Created by David Song

Contents

The Python GIL Issue

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time.

Removing the GIL would have made Python 3 slower in comparison to Python 2 in single-threaded performance and you can imagine what that would have resulted in. You can't argue with the single-threaded performance benefits of the GIL. So the result is that Python 3 still has the GIL.

Multiprocess

multiprocessing is a package that supports spawning processes using an API similar to the threading module. In multiprocessing, processes are spawned by creating a Process object and then calling its start() method. Process follows the API of threading.Thread.

Below example shows how to create and run a process:

```
import os
from multiprocessing import Process, get_logger
import logging

def worker(name):

    logger = get_logger()
    formatter = logging.Formatter('%(asctime)s:  %(processName)-10s %  %(message)s', 
datefmt='%Y/%m/%d %H:%M:%S')

    handler = logging.StreamHandler()
    handler.setFormatter(formatter)
    logger.addHandler(handler)
    logger.setLevel(logging.INFO)

    logger.info('process id:%d', os.getpid())
    logger.info('hello %s', name)


if __name__ == '__main__':
    print(os.getpid())
    p = Process(target=worker, name='worker', args=('bob', ))
    p.start()
    p.join()
```

Exchanging objects between processes

Queue

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    #blocking
    p.join()
```

Pipe

Pipe is one-way communication, means you can only write at one end, and read at another end.

```
from multiprocessing import Process, Pipe

def f(conn):
    for i in range(10):
        conn.send(i)

    conn.send('END')


if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    while True:
        data = parent_conn.recv()
        if data == 'END':
            break
        print(data)

    p.join()
    child_conn.close()
    parent_conn.close()
```

Synchronization between processes

multiprocessing contains equivalents of all the synchronization primitives from threading.

Sharing state between processes

When doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes

Shared memory

Data can be stored in a shared memory map using Value or Array.

Server process

A manager object returned by Manager() controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

Using a pool of workers

The Pool class represents a pool of worker processes.

```
from multiprocessing import Pool, TimeoutError

def f(x):
    return x*x

if __name__ == '__main__':

    with Pool(processes=4) as pool:
        print(pool.map(f, range(10)))
```

Note: the pool needs to be created inside the __main__ module, otherwise, execution will throw exceptions, since each process can have a pool.

Logging

One way for multiprocess logging is to use a separate log file for each process.

To use a single log file for all processes, we need to create an additional process (the log listener), which receive log messages from other process and write them into a log file, while worker processes send log messages to the listener. One way to achieve this is use a multiprocess queue, as illustrated in below:

```
'''

multiprocess logging: use a multiprocess queue to log messages to a single file

1.  create a multiprocess queue
2.  start a listener process listen(wait) on the message queue
    write received messages to a log file

3.  worker process init a logger use QueueHandler,
    with the process name as the logger name
    and use the logger log message

Note that, QueueListener cannot be used in the log listener in multiprocess,
    it's used in a single process only. The log listener has to use queue.get
    to get a message from the queue, and write it to the log file

Tested in Python 3

'''

import logging
import logging.handlers
import multiprocessing
import time
from random import random


#
# logging listener configuration setup
# write log messages to a file
#
def log_listener_configurer(queue):
    root = logging.getLogger()

    log_handler = logging.FileHandler('apps.log', 'a')
    log_format = logging.Formatter('%(asctime)s %(processName)-10s %(levelname)-8s
%(message)s')
    log_handler.setFormatter(log_format)
    root.addHandler(log_handler)


#
# the process waits on log messages arrive on queue,
# and write message to log file(s)
#
def log_listener_process(queue, log_listener_configurer):

    #
    # set up log listener configuration
    #
    log_listener_configurer(queue)
    while True:
        try:
            record = queue.get()
            #
            # main process sends None as a sentinel to tell the listener to quit.
            #
            if record is None:
                break

            #
            # use process name to distinguish each other in log
            #
            logger = logging.getLogger(record.name)
            logger.handle(record)
        except Exception as ex:
            import sys, traceback
            print('Unexpected exception:{}'.format(str(ex)),file=sys.stderr)
            traceback.print_exc(file=sys.stderr)


#
# logging config used by processes, write log message to a Queue
#
def worker_logging_configurer(queue):
    h = logging.handlers.QueueHandler(queue)
    root = logging.getLogger()
    root.addHandler(h)
    root.setLevel(logging.DEBUG)


#
# worker processes use the queue for logging
#
def worker_process(queue, worker_log_configurer):

    #
    # set up worker log configuration
    #
    worker_log_configurer(queue)

    #
    # init the logger with process name
    #
    name = multiprocessing.current_process().name
    logger = logging.getLogger(name)

    print('Worker started: %s' % name)
    for i in range(10):

        logger.log(logging.INFO, 'Start working on {}'.format(str(i)))
        #
        # do some useful work
        #
        time.sleep(random())
        logger.log(logging.INFO, 'Finished working on {}'.format(str(i)))

    print('Worker finished: %s' % name)
```

```
def main():

    #
    # init the log queue
    #
    log_queue = multiprocessing.Queue(-1)

    #
    # set up and start the log listener process
    #
    listener = multiprocessing.Process(target=log_listener_process,
                                       args=(log_queue, log_listener_configurer))
    listener.start()

    #
    # create worker processes
    #
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process, name='woker '+str(i),
                                         args=(log_queue, worker_logging_configurer))
        workers.append(worker)
        worker.start()

    #
    # wait all works to complete
    #
    for w in workers:
        w.join()


    #
    # send a sentinel to tell the log listener to quit
    #
    log_queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()
```