Exercise 13 – Multitasking

Objective

To run external programs, in this case other Python scripts, using a variety of methods, first using the **subprocess** module and then using **multiprocessing**.

Questions

1. In the **labs** or (on Linux) your home directory, you will find a simple Python program, **client.py**, which lists files to STDOUT. The name of the file is specified at the command line, and if it cannot be read then an error is returned, using exit.
2. Now call the Python program **client.py** from another, passing a filename. If you can't think of a file to list, use the current program, or use the 'words' file.

Output an error message if, for some reason, the **client.py** fails.

Test this by:

passing a non-existent file name

calling a non-existent program

1. Modify the calling program to use a pipe and capture its output in a list. Print out the number of lines returned by the **client.py** program. Test as before.
2. The purpose of this exercise is to experiment with different scenarios using the **multiprocessing** module. This is best demonstrated using a multi-core machine, so you might first like to check if that is the case. If not, then the exercise is still valid, but not quite so interesting.

**Note**: IDLE, and some other IDEs, does not display output from the child processes run by the multiprocessing module. So, run your code from the command-line.

Word prefixes are also called *stems*. We have written a program, **stems.py**, that reads the words file and generates the most popular stems of 2 to ***n*** characters long. It uses the **mytimer** module we created in a previous exercise, which you should make available.

Run the supplied **stems.py** program and note the time taken. You will note that no word exceeds 28 characters, so ***n*** could be 28, however we can increase the value of ***n*** to obtain a longer runtime and demonstrate multiprocessing.

This time could be better used by splitting the task between cores. Using the **multiprocessing** module will require the stem search to be moved to a function. Make sure that all the rest of the code is only executed in main (if \_\_name\_\_ == '\_\_main\_\_': test).

Scenarios:

1. ***n*** worker processes

This is where we split the task such that each stem length search runs in its own child process.

1. 2 worker processes ***n***/2 stem sizes each.

This assumes 2 CPU cores. It will require two processes to be launched explicitly, and each to be given a range of stem lengths to handle.

1. 2 worker processes using a queue.

This assumes 2 CPU cores. As in b), but instead of passing a range, pass the stem lengths through a queue. Make sure you have a protocol for the worker processes to detect that the queue has finished.

**If time allows…**

1. Recall the shareprices.py program in the 05 Collections exercises. Your job in this exercise is to make the program multi-threaded. If you did not complete the previous exercise, then take a copy from the solutions/05 Collections directory.

The share\_prices dictionary itself needs to be changed. The value is now a list, the first element is the sequence (thread) number, and the second is the share price. Initialise share\_prices as follows:

share\_prices = {'Global Motors':['0',50],

'Big Blue Inc.':['0',50],

'Gates Software':['0',50],

'Banana Computers':['0',50]

}

You will need two functions:

set\_stock\_prices

Takes one argument – the sequence number.

Loop continually, setting the sequence number, and the share price (as before), in share\_prices.

read\_stock\_prices

No arguments are required to this function.

Loop, printing out the details of the share\_prices dictionary every two seconds. Each time we print out share\_prices, the sequence number on each line should be the same for each member.

For example:

1 Banana Computers $01.30

1 Global Motors $02.14

1 Big Blue Inc. $01.08

1 Gates Software $02.70

3 Banana Computers $01.03

3 Global Motors $09.89

3 Big Blue Inc. $01.16

3 Gates Software $01.11

is OK, but:

2 Banana Computers $01.30

1 Global Motors $02.14

3 Big Blue Inc. $01.08

1 Gates Software $02.70

1 Banana Computers $01.03

3 Global Motors $09.89

2 Big Blue Inc. $01.16

3 Gates Software $01.11

is not!

You need to apply a lock each time you want to read or write to share\_prices.

**Run four threads for each function.** The sequence number is passed into each set\_stock\_prices thread and should be between 0 and 3.

1. Find the Python program Fcopy.py. It copies files from your machine to the Instructor's machine, timing the operation. On Linux, note that a shared folder should be setup.

Run this program first to get a benchmark timing.

1. Write a multi-threaded version, using Queues. Have two worker threads (you can experiment with other number of threads if you wish) which do the copy, the main thread will pass filenames to these threads using a queue.

After the copy has been done, each worker thread should pass the new filename to an additional thread that will delete the file, using a second queue.

Finally, don't forget to place a marker (like False) onto the queue to indicate the end of the list, and wait (join) for all threads to complete.

Did the multithreaded version run quicker?

1. Convert your multithreaded program to use the multiprocessing module. Does that run quicker or slower?

Solutions

import subprocess

import os

import sys

#(a)

proc = subprocess.run([sys.executable, 'client.py', 'words'])

print('Child exited with', proc.returncode)

#(b)

proc = subprocess.run([sys.executable, 'client.py', 'words'],

stdout=subprocess.PIPE, stderr=subprocess.PIPE)

if proc.stderr != None:

print('error:', proc.stderr.decode())

print('output:', proc.stdout.decode())

1. The timings will obviously vary depending on the machine:

a)

import mytimer

from multiprocessing import Process

def stem\_search(stems, stem\_size):

best\_stem = ''

best\_count = 0

for (stem, count) in stems.items():

if stem\_size == len(stem) and count > best\_count:

best\_stem = stem

best\_count = count

if best\_stem:

print ('Most popular stem of size', stem\_size, 'is:',

best\_stem, '(occurs', best\_count, 'times)')

return

if \_\_name\_\_ == '\_\_main\_\_':

mytimer.start\_timer()

stems = {}

for row in open('words', 'r'):

for count in range(1, len(row)):

stem = row[0:count]

if stem in stems:

stems[stem] += 1

else:

stems[stem] = 1

mytimer.end\_timer('Load')

# Process the stems.

mytimer.start\_timer()

n = 30

for stem\_size in range(2, n+1):

proc = Process(target=stem\_search,

args=(stems, stem\_size))

proc.start()

processes.append(proc)

for proc in processes:

proc.join()

mytimer.end\_timer('Process')

b)

import mytimer

from multiprocessing import Process

def stem\_search(stems, start, end):

for stem\_size in range(start, end):

best\_stem = ''

best\_count = 0

for (stem, count) in stems.items():

if stem\_size == len(stem) and

count > best\_count:

best\_stem = stem

best\_count = count

if best\_stem:

print ('Most popular stem of size',

stem\_size, 'is:', best\_stem,

'(occurs', best\_count, 'times)')

return

if \_\_name\_\_ == '\_\_main\_\_':

mytimer.start\_timer()

stems = {}

for row in open('words', 'r'):

for count in range(1, len(row)):

stem = row[0:count]

if stem in stems:

stems[stem] += 1

else:

stems[stem] = 1

mytimer.end\_timer('Load')

# Process the stems.

mytimer.start\_timer()

n = 30

proc1 = Process(target=stem\_search,

args=(stems, 2, int(n/2) + 1))

proc1.start()

proc2 = Process(target=stem\_search,

args=(stems, int(n/2) + 1, n + 1))

proc2.start()

proc1.join()

proc2.join()

mytimer.end\_timer('Process')

c)

import mytimer

from multiprocessing import Process, Queue

def stem\_search(stems, queue):

stem\_size = 1

while stem\_size > 0:

stem\_size = queue.get()

best\_stem = ''

best\_count = 0

for (stem, count) in stems.items():

if stem\_size == len(stem) and count > best\_count:

best\_stem = stem

best\_count = count

if best\_stem:

print ('Most popular stem of size', stem\_size,

'is:', best\_stem, '(occurs', best\_count,

'times)')

return

if \_\_name\_\_ == '\_\_main\_\_':

mytimer.start\_timer()

stems = {}

for row in open('words', 'r'):

for count in range(1, len(row)):

stem = row[0:count]

if stem in stems:

stems[stem] += 1

else:

stems[stem] = 1

mytimer.end\_timer('Load')

mytimer.start\_timer()

n = 30

queue = Queue()

proc1 = Process(target=stem\_search, args=(stems, queue))

proc2 = Process(target=stem\_search, args=(stems, queue))

proc1.start()

proc2.start()

for stem\_size in range(2, n):

queue.put(stem\_size)

queue.put(0)

queue.put(0)

proc1.join()

proc2.join()

mytimer.end\_timer('Process')

**If time allows…**

from threading import Thread

from threading import Lock

import time

import random

share\_prices = {'Global Motors':['0',50],

'Big Blue Inc.':['0',50],

'Gates Software':['0',50],

'Banana Computers':['0',50]

}

cs\_share\_prices = Lock()

#########################################################

def set\_stock\_prices(seq):

# Updates stock prices with random price changes

global share\_prices

while True:

cs\_share\_prices.acquire() # TODO

for key, sp in share\_prices.items():

share\_prices[key][0] = seq

share\_prices[key][1] = max(1.0,

sp[1] \* ( 1 + ((random.random() –

0.5)/0.5) \* 0.05))

cs\_share\_prices.release() # TODO

return

#########################################################

def read\_stock\_prices():

global share\_prices

while True:

cs\_share\_prices.acquire() # TODO

for key, sp in SharePrices.items():

print('{} {:<18s} ${:05.2f}'.\

format(sp[0], key, sp[1]))

print()

cs\_share\_prices.release() # TODO

time.sleep(2)

return

if \_\_name\_\_ == '\_\_main\_\_':

tids = []

# Start share price update thread.

for i in range(0, 4):

th\_set = Thread(target=set\_stock\_prices, args=str(i))

th\_set.start()

# Wait for request from client.

for i in range(0, 4):

th\_st = Thread(target=read\_stock\_prices )

th\_st.start()

tids.append(th\_st)

for tid in tids:

tid.join()

1. Here is our multithreaded version (without the timing routines), which actually runs slightly slower than the single threaded version.

import platform

import os.path

import glob

from threading import Thread

from queue import Queue

#########################################################

def remove\_thread(\*args):

target\_dir, queue = args

while True:

filename = queue.get()

if not filename: break

os.remove(target\_dir + filename)

return

def worker\_thread(\*args):

target\_dir, queue, rqueue = args

while True:

filename = queue.get()

if not filename: break

data = open(filename, 'rb').read()

filename = os.path.basename(filename)

fh\_out = open(target\_dir + filename, 'wb')

fh\_out.write(Data)

fh\_out.close()

rqueue.put(filename)

return

opsys, host = platform.uname()[:2]

source = './Bitmaps/\*'

if opsys == 'Windows':

target\_dir = '\\\\INSTRUCTOR\\Shared\\' + host + '\\'

else:

target\_dir = '/mnt/hgfs/\\\\INSTRUCTOR/' + host + '/'

if not os.path.isdir(target\_dir):

os.mkdir(target\_dir)

start\_timer()

num\_threads = 2

for i in range(0, 10):

print('Loop', i)

queue = Queue()

rqueue = Queue()

Tids = []

for i in range(0, num\_threads):

Tids.append(Thread(target=worker\_thread,

args=(target\_dir, queue, rqueue)))

rth = Thread(target=remove\_thread, args=(target\_dir, rqueue))

for th in Tids:

th.start()

rth.start()

for filename in glob.iglob(source):

queue.put(filename)

for th in Tids:

queue.put(False)

for th in Tids:

th.join()

rqueue.put(False)

rth.join()

end\_timer('Threaded:')

This is our multiprocessing version (without the timing routines), which runs considerably faster:

import platform

import os.path

import glob

from multiprocessing import Process, Queue

def remove\_process(\*args):

target\_dir, queue = args

while True:

filename = queue.get()

if not filename: break

os.remove(target\_dir + filename)

return

def worker\_process(\*args):

target\_dir, queue, rqueue = args

while True:

filename = queue.get()

if not filename: break

data = open(filename, 'rb').read()

filename = os.path.basename(filename)

fh\_out = open(target\_dir + filename, 'wb')

fh.write(data)

fh.close()

rqueue.put(filename)

return

if \_\_name\_\_ == '\_\_main\_\_':

opsys, host = platform.uname()[:2]

source = './Bitmaps'

if not os.path.isdir(source):

sys.exit('Unable to access ' + source)

source = source + '/\*'

if opsys == 'Windows':

target\_dir =

'\\\\INSTRUCTOR\\Shared\\' + host + '\\'

else:

target\_dir =

'/mnt/hgfs/\\\\INSTRUCTOR/' + host + '/'

if not os.path.isdir(target\_dir):

os.mkdir(target\_dir)

start\_timer()

num\_procs = 2

for i in range(0, 10):

print('Loop', i)

queue = Queue()

rqueue = Queue()

pids = []

for i in range(0, num\_procs):

pids.append(Process(target=worker\_process,

args=(target\_dir, queue, rqueue)))

rth = Process(target=remove\_process,

args=(target\_dir, rqueue))

for th in pids:

th.start()

rth.start()

for filename in glob.iglob(source):

queue.put(filename)

for th in pids:

queue.put(False)

for th in pids:

th.join()

rqueue.put(False)

rth.join()

end\_timer('Multiprocessing:')