HW1

學號:0716049
姓名:詹凱傑

Task 1

Thread

	第一次測試	第二次測試	第三次測試
單執行緒	1987.35	2449.09	2353.608
雙執行緒	1582.86	1534.62	1820.69
4 個執行緒	1500.45	1514.98	1517.64
100 個執行緒	1050.37	1169.36	1161.97

Process

	第一次測試	第二次測試	第三次測試
單行程	1406.25	1134.02	1113.19
雙行程	632.105	528.29	568.70
4 個行程	404.97	306.40	383.67
100 個行程	189.64	211.12	188.68

協程

	第一次測試	第二次測試	第三次測試
協程	910.66	915.89	902.83

(單位: 秒)

Task 2

Thread

	第一次測試	第二次測試	第三次測試
單執行緒	393.14	343.53	99.64
雙執行緒	173.67	80.77	23.94
4 個執行緒	86.78	35.06	17.202
100 個執行緒	2.25	1.044	1.069

Process

	第一次測試	第二次測試	第三次測試
單行程	81.67	131.48	150.51
雙行程	27.53	30.24	50.03
4 個行程	13.63	22.02	21.97
100 個行程	8.64	10.06	9.35

協程

	第一次測試	第二次測試	第三次測試
協程	62.58	81.89	68.81

(單位: 秒)

Python 版本: 3.8.1
用到的 **module:**

hashlib, requests, random, time, threading, multiprocessing, asyncio

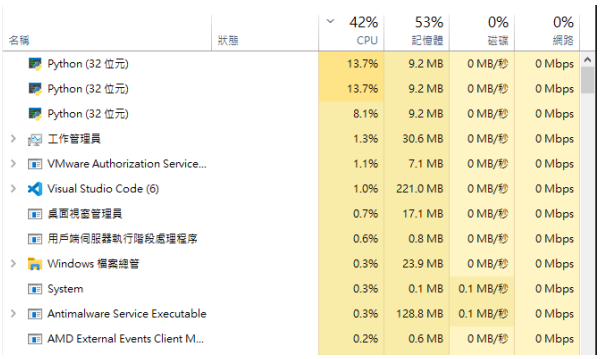
1. 執行緒對效能的影響

不管從 task1 還是從 task2 來看，增加 thread 的數量都能夠讓執行時間縮短，但縮短的幅度不太一樣。先從 task2 來看，送出 request 後要等到 server 回應。在一個 thread 的情況下，送出一個 request 後，等到 server 回應後在送下一個 request，這樣等待的時間就會是 100 個 request 的時間相加，算是浪費的很多時間在等待，所以如果能用多個 thread，在前一個 thread 送出 request 然後在等待時，下一個 thread 就先送出 request，這樣可以節省等待的時間，所以等待的時間就會少很多。再來，我有觀察到比較有趣的地方，就是我的結果會越跑越快，例如:第三次的 4 個 thread 時就比第一次測 4 個 thread 快很多，由於我的測資是用助教給的檔案，且三次都是同一組測資，所以可能是經過幾次 request 後有一些 cache 產生，才會讓速度越來越快。

再來會到第一個 task，第一個 task 跟第二的 task 的性質不一樣，第一個 task 要 cpu 大量的運算來得到答案，所以在多個 thread 的情況下，得到答案的時間會變短，但減少的比例更第二的 task 比起來就沒有快很多。而我推測可能的原因，是因為不管是多少個 thread，要完成 task 1 的運算量都是差不多的，所以時間也不會快很多。而在 task 1 中，我也有注意到一個有趣的地方，由於我的算法是 random 選一個數，然後看 hash 完後的結果有沒有符合條件，不符合就 random 再選一個，直到找到答案，由於這種算法也有點運氣的成分，再加上我們只要測試 100 筆，看起來很多但從機率的觀點來看，每一次運算還是有很大的運氣成分，這可能是導致相同數量的 thread，但每次的結果有些差異的原因。

2. 行程數量對效能的影響

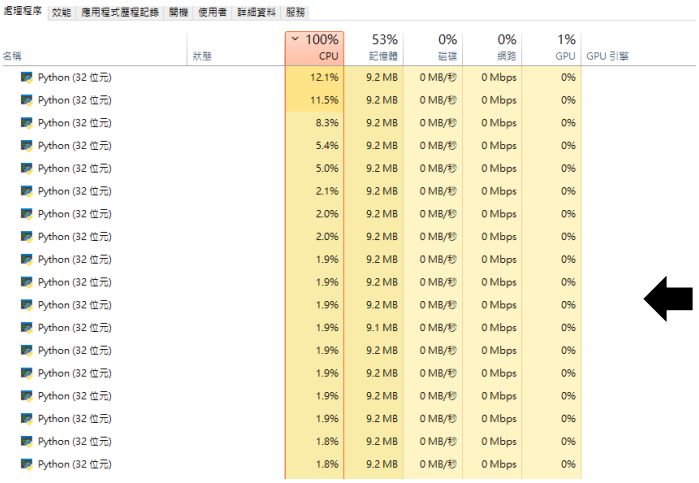
多的 process 的情況下，趨勢也是差不多的，就是越多 process，完成 task 的時間越短。這邊先討論 task1 的結果，從直覺來看，一個 process 分成兩個 process，執行時間應該要變一半，然後變成 100 個 process，則執行時間會變 100 分之一，但實際上來看不可能變這麼快，從工作管理員來



名稱	狀態	42% CPU	53% 記憶體	0% 磁碟	0% 網路
Python (32 位元)		13.7%	9.2 MB	0 MB/秒	0 Mbps
Python (32 位元)		13.7%	9.2 MB	0 MB/秒	0 Mbps
Python (32 位元)		8.1%	9.2 MB	0 MB/秒	0 Mbps
工作管理員		1.3%	30.6 MB	0 MB/秒	0 Mbps
VMware Authorization Service...		1.1%	7.1 MB	0 MB/秒	0 Mbps
Visual Studio Code (6)		1.0%	221.0 MB	0 MB/秒	0 Mbps
桌面視窗管理員		0.7%	17.1 MB	0 MB/秒	0 Mbps
用戶端伺服器執行階段處理程序		0.6%	0.8 MB	0 MB/秒	0 Mbps
Windows 檔案總管		0.3%	23.9 MB	0 MB/秒	0 Mbps
System		0.3%	0.1 MB	0.1 MB/秒	0 Mbps
Antimalware Service Executable		0.3%	128.8 MB	0.1 MB/秒	0 Mbps
AMD External Events Client M...		0.2%	0.6 MB	0 MB/秒	0 Mbps

2 個 process
(1 個 main+產生的 2 的 process)

看，



處理程序 效能 應用程式歷程記錄 開機 使用者 詳細資料 服務

名稱	狀態	100% CPU	53% 記憶體	0% 磁碟	0% 網路	1% GPU	GPU 引擎
Python (32 位元)		12.1%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		11.5%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		8.3%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		5.4%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		5.0%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		2.1%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		2.0%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		2.0%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.9%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.9%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.9%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.9%	9.1 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.9%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.9%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.9%	9.2 MB	0 MB/秒	0 Mbps	0%	
Python (32 位元)		1.8%	9.2 MB	0 MB/秒	0 Mbps	0%	

100 的 process

當開到 100 個 processes 時，cpu 的使用率已經達到 100%了，所以到越多 thread，增快的比例也越來越少，一開始單個 process 變兩個 process 時，有可能真的增快兩倍，但便 100 個 processes 時，只有增快十倍。然後在測試這些數據時，由於要跑很久，所以我不太可能就放著電腦等全部的結果跑完，可能還會做一些其他的事，而我發些這似乎也會影響到 task 要執行的時間，畢竟系統上還有很多其他的 process 要一起競爭技統的資源，所以當 task 1 在執行時，若我在電腦上也做一些要複雜運算的操作，就會讓執行時間變更長。

在 task 2 的部分，第一題有說到兩個 task 的性質不同，task 1 是 CPU bound， task 2 是 IO bound，所以用多個 process 一起等待，也會縮短總共的等待時間。然後在測試 multi process 時，有了前面的經驗，可能會有 Cache，所以我是在不同天測試的，除了第一次測試是和 multi thread 測試時同一天外，另外兩次都是在不同天跑的，或許這樣測出來會更準確。

3. 多執行緒、多行程、協程的效能比較

在第一個 task 中，得到答案的時間排序是，100 個 processes 最短，再來從結果來看，100 個 threads 跟協程的結果很接近，用協程好像快一點，最後是單個 thread 最慢。雖然用 100 個 processes 跑的效能是最好的，但同時這個方式用到的 cpu 資源也是最多的，而在 CPU bound 的 process 中 multi thread 和協程都沒辦法達到很好的效果，不過我認為協程可以比 100 個 threads 還要快一點的原因在於協程的切換比較快，且切換的 overhead 也比較小。

第二個 task 中，屬於 IO bound 的 task，而得到答案的時間排序是，100 的 threads 跟 100 的 processes 都很快就求出答案，而 100 的 threads 又比 100 個 processes 快，再來依序是協程和單個 thread。首先是 100 個 threads 和 100 個 processes 的比較，我認為會造成這個結果的原因是因為新產生一個 process 所需要的 overhead 比新產生一個 thread 大，所需的資源也比較多，所以會造成 100 個 processes 所需的時間比較久。再來是協程的部分，雖然協程不是真正的多線程，但協程之間最切換是由程式控制，不會由 OS 控制，所以切換的速度比較快，再加上這個 task 是屬於 IO bound，所以協程跟單個 thread 比起來，就變得好有效率。

心得:

由於我之前之有用 C++寫過 multithread 的程式，但沒有用 python 寫過，所以第一次用 Python 寫這類型的 code 覺得還蠻新鮮的，此外，這次的作業每個個別的運算彼此間都是獨立的，所以不用用到 Lock，由於我之前寫 multithread 時，覺得 Lock 很麻煩，且因為有 Lock，沒辦法真的看出 multithread 的優勢，跑起來都差不多，但在這次作業中，可以明確的看出使用 multithread 和 multiprocessing 的優點，這方面我覺得蠻有趣的，且有些結果也有符合上課所學的，讓我對這方面又更了解。最後，我覺得這次的作業還算是蠻有趣的，且我也學到了如何用 python 寫多執行緒、多核心、協程的程式，算是學到了不少。