OS Project2

- 學號 : 0616018
- 姓名 : 林哲宇
- Language : python2
- usage : python project2_0616018.py
- library : hashlib, socket, random, thread, multiprocessing

這次作業要求我們實作 Proof of Work(POW) 的 server，其中要使用 multiprogram 來提升速度，計算平均每秒鐘可以算出多少個難度為 PoWDifficulty 的 hash

how do you implement the server program

1. 建立一個 socket 監聽在 localhost:20000
2. 根據 connectionNumber 決定 multiprocess 要開多少個 process
3. 實作每個 process 需要做的事情(包含 recv 與 send)
4. 利用 multithread 計算出符合條件的 hash

建立一個 socket 監聽在 localhost:20000

這項 Project 我使用 python 來寫，首先建立一個 class，在 init 中把 socket 定義完畢。其中還有定義其他初始變數

```
class Server():
  def __init__(self, connectionNumber, PoWDifficulty, procThread):
    self.connectionNumber = connectionNumber
    self.PoWDifficulty = PoWDifficulty
    self.procThread = procThread

    self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.server.bind(('127.0.0.1', 20000))
    self.server.listen(100)

    self.message = [''] * self.connectionNumber
    self.seed = [''] * self.connectionNumber
    self.hash = [''] * self.connectionNumber

    self.printable = string.uppercase + string.lowercase + '0123456789'
```

根據 connectionNumber 決定 multiprocess 要開多少個 process

將每個 process 存在 allpro 中，是為了在所有 process 執行 start 函數之後，才執行 join 函數。因為如果先用 join 的話，main process 就會被擋住。

```
# each process receive the message and send the seed
def mul_proc(self):
    allpro = []
    for index in range(self.connectionNumber):
      p = Process(target = self.proc_con, args = (index,))
      allpro.append(p)
      p.start()
    for p in allpro:
      p.join()
```

實作每個 process 需要做的事情(包含 recv 與 send)

`proc_con` 主要是在與 client 互動。先接收來自 client 的資料，然後算出對應的 seed 和 hash，最後將結果傳回 client。

```
# each process receive the message and send the seed
  def proc_con(self, index):
    conn, addr = self.server.accept()
    #print addr
    while 1:
      self.message[index] = ''
      self.message[index] = conn.recv(6).strip()
      #print '[{}]: {}'.format(index, self.message[index])
      self.seed[index] = ''
      self.hash[index] = ''
      self.mul_thread(index)
      while not self.seed[index]:
        continue
      response = '{message},{seed},{hash_}\n'.format(message = self.message[index], see
      try:
        conn.send(response)
      except:
        return
```

利用 multithread 計算出符合條件的 hash

這邊是實作在上面 proc_con 有使用到的 mul_thread 函數。要開的 thread 數量(procThread)在 init 有做過初始化，每個 thread 執行 cal_hash 函數。

cal_hash 實際上就是持續產生長度為五的隨機字串 guess，並且確認 `sha256(message + guess)` 是否符合 Difficulty。當有其他 thread 算出來，就會透過 `if self.seed[index]` 這個判斷做 return。

```
# multhread for calculating hash parrallely
  def mul_thread(self, index):
    for i in range(self.procThread):
      thread.start_new_thread(self.cal_hash, (index,))

  # calculate hash for corresponding message
  def cal_hash(self, index):
    while 1:
      if self.seed[index]:
        return
      guess = ''.join(random.sample(self.printable, 5))
      res = hashlib.sha256(guess + self.message[index]).hexdigest()
      if (res[:self.PoWDifficulty] == '0' * self.PoWDifficulty) and (not self.seed[inde
        self.hash[index] = res
        self.seed[index] = guess
        #print('message: {}, seed: {}, hash: {}'.format(self.message[index], self.seed[
```

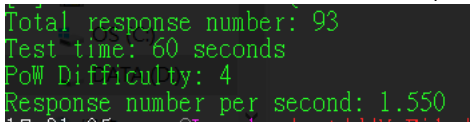
the performance of the server program, and how do you evaluate it

減少 syscall

在速度方面，我有特別注意不用到 system call，例如產生隨機變數可以用 `os.urandom(5)` 來產生隨機字串。但是我使用 lib 的 random.sample 產生隨機變數。還有把一些 debug 用的 printf 註解。這樣一來應該可以減少 conetxt switch 的次數。

調整 process 和 thread 數量

我測試的方式就是調整 process 和 thread 的數量，去測出分數最高的值。process 是 connection 的數量，thread 是算 hash 的分支數量。

1. process x 1 , thread x 1
   這是在只有建立一個 connection 的情形，也就是說只有一個 process；算 hash 的 thread 也只有開一個。結果只有每秒平均 1.550 個 response。
   

2. process x 4, thread x 1
   這是在只有建立四個 connection 的情形，也就是說只有四個 process；算 hash 的 thread 只有開一個。結果每秒平均 4.417 個 response，是完全沒有平行程式的三倍左右。

```
Total response number: 265
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 4.417
```

3. process x 8, thread x 1
   從這邊開始，已經比上面建立四個 process 還慢了，推測可能是因為我是四核心筆電。所以開超過四個就有變慢的趨勢

```
Total response number: 256
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 4.267
```

4. process x 1, thread x 4
   一個 process, 四個 thread，竟然只有 1.383，比完全沒優化還慢，看來速度的重點應該是在於建立的 connection 數量吧

```
Total response number: 83
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 1.383
```

5. process x 4, thread x 4

```
Total response number: 345
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 5.750
```

6. process x 8, thread x 4

```
Total response number: 291
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 4.850
```

7. process x 12, thread x 4

```
Total response number: 296
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 4.933
```

8. process x 4, thread x 8
   四個 process, 八個 thread，這是我測過最快的方案

```
Total response number: 358
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 5.967
```

9. process x 8, thread x 8
   又變慢了(汗

```
Total response number: 312
Test time: 60 seconds
PoW Difficulty: 4
Response number per second: 5.200
```

心得

這次作業讓我學到平行程式的大致概念，用 python 寫真的挺簡單的，然而速度卻比同學用 C++ 的慢上許多，看來 python 效能真的不高。

照理來說，thread 開越大應該得到的分數越高，實際上也是如此，然而進步的幅度卻有邊際效應。也就是說，當 thread 的數量為 10，可能平均是 4，但是就算我調到 100，也只有 6。我覺得這部分可能是跟電腦本身的效能有關。