

MULTIPROCESSING, HASHES AND HTTP REQUESTS

CENG2034, OPERATING SYSTEMS

Mehmet Pekcan
mehmetpekcan@posta.mu.edu.tr

GitHub username
mehmetpekcan

More detail check documentation
mehmetpekcan.gitbook.io/ceng2034

Thursday 4th June, 2020

Abstract

Threading and multiprocessing always being a trouble in every cases. Both cases has benefits for their usage areas. In our case, we will implement a script to make HTTP request some URLs and download files. After downloading files, script will be identify them if there is a duplicate files. In this case, using multiprocessing is more beneficial for speed if the URLs count decreased insanely.

1 Introduction

Purpose of this report is learning how to use multiprocessing in our codes and how it will be increased our script performance. Meanwhile we will learn how to uniquely identify a file, how to make HTTP request, how to create child process and so on..

2 Assignments

In this report, duplicate image finding performed using multiprocessing and normal threading. And also which one is more efficient?

2.1 What is used in the project

2.1.1 Which kernel and system version used in the project

System Version: macOS 10.15.3 (19D76) Kernel Version: Darwin 19.3.0

2.1.2 Which language and version used in the project

Python and version 3.8.0 were used.

2.1.3 Which libraries used in the project

os, for using operating systems command which is basically codes that uses in terminal

time, this for testing start and finish time to check which way is more efficient in speed way

requests, this is for making HTTP requests to given URLs

uuid, to generate a unique name

hashlib, to uniquely identify files if they are unique or duplicate

multiprocessing, this is for making multi processing at a time

product, to send two parameter to our mapping function (will be more details)

re, to use regex functionality in script

2.2 Problems and How to solve them in the project

2.2.1 How to create child process and using it

Creating child process in Python is so easy, after importing **os** library, we should first forking a process from parent. After forking child process, Python create a sub process of parent and when process id more than zero means that Python running at parent process, when process id is equal zero means that Python running at child process id. And also, if our parent process took less time compare to child process, it means that our script will be die before child process done. To prevent this we can say that process id to wait until child process done.

Listing 1: Creating child process then writing its process id

```
def fork_childProcess_showID():
    childProcess = os.fork()
    if childProcess > 0:
        os.waitpid(childProcess, 0)
    elif childProcess == 0:
        print("Child process id is:", os.getpid())
```

2.2.2 Sending Request to Download Files

Sending HTTP request in Python can make using **request** library. We can make all HTTP request with this library such as GET, POST, PUT, DELETE.

Listing 2: Downloading files

```
def download_file(url, file_name=None):
```

```
file_name = find_image_format(url, file_name)
response = requests.get(url, allow_redirects=True)
open(file_name, "wb").write(response.content)
```

Above code is making get request and setting it's name (we'll mention this later) after downloading, we're opening a file then writing request response content in it. Basically we are taking URLs file content, then opening a file and write in it to save this URLs content as a file our directory. And additionally find image format function is look like this:

Listing 3: Finding image format then renaming it

```
def find_image_format(url, file_name):
    # if file has no parameter comes from user
    # generate an unique id
    file_name = file_name if file_name else str(uuid.uuid4())
    formats = ["jpeg", "jpg", "png", "gif", "tiff", "eps", "svg", "pdf"]

    # if any known files format has "url" find it
    # and suppose that this file is that format
    found_format = ""
    for extension in formats:
        for match in regex.finditer(extension, url):
            found_format = match.group()

    # return new file name with founded format of file
    return file_name+"."+found_format
```

2.2.3 Creating Multiprocess

Before continue with our task we should know to use multiprocesses. Multiprocesses run at a same time. Normally Python runs synchronously. So let's create a multiprocess using **multiprocessing** library.

Above code create a process pool to call function. Then mapping them with a function and parameter. And the output is:

Listing 4: How to create multiprocess

```
import multiprocessing
import time # this is for simulating

def do_staff(second):
    print(f"do_staff function starts with parameter: {second}")
    time.sleep(second)
    print("do_staff function ends")

if __name__ == "__main__":
    processPool = multiprocessing.Pool()
    processPool.map(do_staff, [1,2,3,4])
```

2.2.4 Creating Multiprocess for our task

First let's download files using **multiprocessing** library.

Listing 5: Download File using Multiprocess

```
processPool = multiprocessing.Pool()

# Assume that "...." end of the urls
request_urls = [
    "http://wiki.netseclab.mu.edu.tr/images/thumb.....",
    "https://upload.wikimedia.org/wikipedia/tr/9....",
    "https://upload.wikimedia.org/wikipedia/c....",
    "http://wiki.netseclab.mu.edu.tr/images/thumb....",
    "https://upload.wikimedia.org/wikipedia/commons...."]

processPool.map(download_file, request_urls)
```

Difference between normal way and multiprocessing way is that the way of calling methods in this case. Because our function was taking an URL then sending a request then writing into a file, nothing change when we want to use this function with multiprocessing on function. But calling is different. Let's look close above code, first we create a Pool to hold our multiprocess variable, after then we have URLs to send request. Using **Pool** variable, we use mapping function which is core function in Python. In this way, for every list item, our function will be work and all of them will be work asynchronously so all of them can be start at same time. Our first multiprocesssing magic done successfully.

Time for implementing multiprocessing for duplicate image finding. We should know the way to handle this. Firstly we have to uniquely identify our image files. To do this we can use **md5** function in **hashlib**.

Listing 6: Hash Generator Function

```
from hashlib import md5

def hashinize(image):
    with open(image, "rb") as imageFile:
        return imageHash = md5(imageFile.read()).hexdigest()
```

Above code takes image as a parameter then opens it as a file. After opening, it reads file in the **md5** function after reading it converts this value to 16 base system using **hexdigest** function. Then return it. And driver code for above code is this:

Listing 7: Driver function at main

```
hashes = [ (hashinize(h), h) for h in cwdf ]
```

We downloaded our files and hashed them. Next thing is finding duplicate values? Take one image hash then check with the others, if equals it means that they are

duplicate, if not it is unique.

Listing 8: Function and driver code for duplicate image finding

```
def find_duplicates_by_multiproc(h, hashes):
    if h[0] == hashes[0]:
        return (h[1], hashes[1])

duplicateRaw = processPool.starmap(find_duplicates_by_multiproc, product(hashes,
repeat=2) )

# This is just for filtering to duplicate tuples from their duplicate :)
duplicateFiltered = []
for i in duplicateRaw:
    if i != None:
        if i[0] != i[1] and i[1]:
            if i[0] not in duplicateFiltered:
                duplicateFiltered.append(i[0])
            elif i[1] not in duplicateFiltered:
                duplicateFiltered.append(i[1])
```

We used **starmap** above code. Because built-in **map** function is taking just one parameter to send function as a parameter, but in our case we should send two parameter to function. So we used **starmap** to send two parameter using **repeat** arguments.

3 Results

As a result, we found duplicate files in given directory. But we should see the big fish at the above explaining. It is about multiprocessing. Finding duplicate files using multiprocessing makes difference than normal threading. Let's see the difference in output.

Listing 9: Output of difference

```
Elapsed time with serial threading is: 2.09sec
Elapsed time with multiprocessing is: 1.02sec
```

As you see the result multiprocessing is running in a half time of serial threading. Think that you have thousands of data to making something, multiprocessing will be make huge different for you.

4 Conclusion

As a result, before starting project we have to determine to structure of project. We should test if i use multiprocessing will be more efficient or not? In many cases, multiprocessing make huge different in time.