Lab 2: Processes & Threads

All code bellow available on
https://gist.github.com/gordinmitya/a517924f9bb12324c37c377e5184d0ee

Using threads allows a program to run multiple operations concurrently in the same process space.

Exercise 1. Threads Creation

1. Write the following code in Python in any text editor and save it as `task1.py`:

```
#!/usr/bin/python3
import threading
from threading import Thread
import time

def func(a, b, name):
    while True:
        a = a + b
        print("{}: {}".format(name, a))
        time.sleep(1)

if __name__ == "__main__":
    thread1 = Thread(target=func, args=(5, 6, "thread1"))
    thread2 = Thread(target=func, args=(1, 2, "thread2"))

    thread1.start()
    thread2.start()

    while True:
        print("Total number of threads:
{}".format(threading.activeCount()))
        print("List of threads:
{}".format(threading.enumerate()))
        time.sleep(5)
```

2. Analyze what the program does.
3. Make the `task1.py` file executable:

```
$ chmod u+x task1.py
```

4. Run the program:

```
$ ./task1.py
```

5. Try to stop program with CTRL + C
6. Did you succeed? Why?
7. Suspend the process using CTRL+Z

8. Find the process ID (PID) of the running program:

```
$ pidof -x task1.py
15091
```

   The number `15091` is the PID.

9. List of all threads associated with a process,

```
$ pstree -p `pidof -x task1.py`


task1.py(15091)─┬─{task1.py}(15092)
                └─{task1.py}(15093)
```

   The numbers 15092 and 15093 are the thread IDs

10. Kill the process:

```
$ kill -9 15091
```

Exercise 2. Daemon Threads

If a program runs *threads* that are not *daemons*, then the program will wait for those threads to complete before it terminates. *Threads* that are *daemons*, however, are just killed when the program is exiting.

1. Copy the previous file `task1.py` as `task2.py`:

```
$ cp task1.py task2.py
```

2. Change lines in the `task2.py` file as follows:

```
...
thread1 = threading.Thread(target=func1, args=(5, 6,
"thread1"), daemon=True)
     thread2 = threading.Thread(target=func1, args=(1, 2,
"thread2"), daemon=True)
...
```

3. Make the `task2.py` file executable:

```
$ chmod u+x task2.py
```

4. Run the program:

```
$ ./task2.py
```

5. Try to stop program with CTRL + C
6. Did you succeed? Why?

Exercise 3. Waiting For Threads To Complete

1. Before discussing the significance of the waiting for threads to complete, let us see the following `task3.py` program file:

```
#!/usr/bin/python3
from threading import Thread
import time

list = []

def func(a):
    time.sleep(1)
    list.append(a)

if __name__ == "__main__":
    thread1 = Thread(target=func, args=(1,))
    thread1.start()
    thread2 = Thread(target=func, args=(6,))
    thread2.start()

    print("List is: ", list)
```

2. Run the program and see a result. Why is the list `list` empty?
3. The main thread must be paused until all threads complete their jobs. To achieve this, we shall use the `join` method. Add the `join` method before the `print()` statement:

```
...
thread2.start()

thread1.join()
thread2.join()

print("List is: ", list)
```

4. Save program as `task4.py` and run it to see a result.

Exercise 4. Thread Class Implementation

1. To implement a new thread using the Class model, you have to do the following:
    - Define a new subclass of the `Thread` class.
    - Override the `__init__(self [,args])` method to add additional arguments.
    - Then, override the `run(self [,args])` method to implement what the thread should do when started.
2. Once you have created the new `Thread` subclass, you can create an instance of it and then start a new thread by invoking the `start()`, which in turn calls `run()` method.

3. Save the following code as `task5.py` and run it:

```
#!/usr/bin/python3
from threading import Thread
import time


class MyThread(Thread):
    def __init__(self, threadID, name, delay):
        super().__init__()
        self.threadID = threadID
        self.name = name
        self.delay = delay

    def run(self):
        print("Starting ", self.name)
        print_time(self.name, 5, self.delay)
        print("Exiting ", self.name)


def print_time(threadName, counter, delay):
    while counter:
        time.sleep(delay)
        print("%s: %s sec" % (threadName, time.strftime("%S",
time.gmtime())))
        counter -= 1


if __name__ == "__main__":
    # Create new threads
    thread1 = MyThread(1, "Thread-1", 1)
    thread2 = MyThread(2, "Thread-2", 2)

    # Start new Threads
    thread1.start()
    thread2.start()

    print("Exiting Main Thread")
```

4. Analyze the result

Exercise 5. Performance, Thread vs Process, Global Interpreter Lock (GIL)

Performance does not always grow due to the increase in the number of threads, especially in Python. Threading may not speed up all tasks. This is due to interactions with the GIL that essentially limit one Python thread to run at a time.

1. Consider the example bellow:

```
#!/usr/bin/python3
import hashlib
from threading import Thread
from multiprocessing import Process
from timeit import timeit

# hashes of random str(int) from [0 ... 1_000_000]
TASKS = ['604678604882550e79d90fd9b29ecf34',
'87456e18f180720ebaaf070f7d1e6e1c',
'98222663d6fe9ea55efff46179d7c9b2',
         'c64a9829fa4638ff5de86330dd227e35',
'ca6bff62f4e46cbb192152ec843ebdbf',
'df7c2b3c3966426c14e4b3005c931eb1',
         '626103abae1be890f7d1c8148f9d690a',
'e31d05da308bf27ad15fedde779f2bc5',
'a2b12d7cf762d6cfb6ea086f9f492626',
         'a8573e231edaaedfb49ebfc14f4be808']

def solve(task):
    for i in range(10 ** 6):
        h = hashlib.md5(str(i).encode("utf-8")).hexdigest()
        if h == task:
            return

def multi():
    executors = []
    for task in TASKS:
        e = Thread(target=solve, args=(task,))
        e.start()
        executors.append(e)
    for e in executors:
        e.join()

def single():
    for task in TASKS:
        solve(task)

if __name__ == '__main__':
  # run function `number` times and return avg execution time
    res = timeit(multi, number=10)
    print("multi", res)
    res = timeit(single, number=10)
    print("single", res)
```

2. Save file as `task6.py` and run it.
3. How many seconds does it take to find a hash with multiple threads and single thread?
4. Let's use Process instead of Thread. Change the word Thread to the word Process in the `multi()` function in the script above and save it as `task7.py`:

```
...
e = Process(target=solve, args=(task,))
...
```

5. How many seconds does it take to find a hash with multiple processes and single process?
6. Is there a boost from using threads/processes?
7. Create an instance in the EC2 AWS (t2.micro instance type) and test `task6.py` and `task7.py` program on it. Use `scp` command to copy files to the remote machine.

Assignment:

Create a multi-threaded File Downloader that downloads files from the Internet.
First you need to find a set of links, you may use these:

```
URLS = ["http://www.irs.gov/pub/irs-pdf/f1040.pdf",
        "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
        "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
        "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
        "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"]
```

Then write a script to download all these files and measure total time. Test your script under these conditions:

1. Spawn threads after previous finished. So they will work one after another.
2. Spawn all threads at once.
3. Change Thread to Process.

Your submission should contain one screenshot with the results of testing the script. And brief note with your thoughts about results: Why in this case Threads give boost as well as Processes?

If you find it difficult to solve the assignment, just use the hint on the next page.

The example of the Downloader class implementation:

```
class Downloader(Thread):
    def __init__(self, url):
        super().__init__()
        self.url = url

    def run(self):
        handle = urllib.request.urlopen(self.url)
        fname = os.path.basename(self.url)

        with open(fname, "wb") as f_handler:
            while True:
                chunk = handle.read(1024)
                if not chunk:
                    break
                f_handler.write(chunk)
```