

Fix a slow system with Python

Introduction

You're an IT administrator for a media production company that uses Network-Attached Storage (NAS) to store all data generated daily (e.g., videos, photos). One of your daily tasks is to back up the data in the production NAS (mounted at `/data/prod` on the server) to the backup NAS (mounted at `/data/prod_backup` on the server). A former member of the team developed a Python script (full path `/scripts/dailysync.py`) that backs up data daily. But recently, there's been a lot of data generated and the script isn't catching up to the speed. As a result, the backup process now takes more than 20 hours to finish, which isn't efficient at all for a daily backup.

What you'll do

- Identify what limits the system performance: I/O, Network, CPU, or Memory
- Use `rsync` command instead of `cp` to transfer data
- Get system standard output and manipulate the output
- Find differences between threading and multiprocessing

CPU bound

CPU bound means the program is bottlenecked by the CPU (Central Processing Unit). When your program is waiting for I/O (e.g., disk read/write, network read/write), the CPU is free to do other tasks, even if your program is stopped. The speed of your program will mostly depend on how fast that I/O can happen; if you want to speed it up, you'll need to speed up the I/O. If your program is

running lots of program instructions and not waiting for I/O, then it's CPU bound. Speeding up the CPU will make the program run faster.

In either case, the key to speeding up the program might not be to speed up the hardware but to optimize the program to reduce the amount of I/O or CPU it needs. Or you can have it do I/O while it also does CPU-intensive work. CPU bound implies that upgrading the CPU or optimizing code will improve the overall computing performance.

In order to check how much your program utilizes CPU, you first need to install the **pip3** which is a Python package installer. This downloads and configures new Python modules with single-line commands. For any pop-up messages, when the prompt *Do you want to continue appears*, type 'Y'.

```
sudo apt install python3-pip
```

psutil (process and system utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. It's mainly useful for system monitoring, profiling, and limiting process resources and management of running processes. Install the psutil python library using pip3:

```
pip3 install psutil
```

Now open python3 interpreter.

```
python3
```

Import psutil python3 module for checking CPU usage as well as the I/O and network bandwidth.

```
import psutil
psutil.cpu_percent()
```

Output:

```
gcpstaging100358_student@linux-instance:~$ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import psutil
>>> psutil.cpu_percent()
1.2
>>> exit()
gcpstaging100358_student@linux-instance:~$
```

This shows that CPU utilization is low. Here, you have a CPU with multiple cores; this means one fully loaded CPU thread/virtual core equals 1.2% of total load. So, it only uses one core of the CPU regardless of having multiple cores.

After checking CPU utilization, you noticed that they're not reaching the limit.

So, you check the CPU usage, and it looks like the script only uses a single core to run. But your server has a bunch of cores, which means the task is **CPU-bound**.

Now, using `psutil.disk_io_counters()` and `psutil.net_io_counters()` you'll get **byte read** and **byte write** for disk I/O and **byte received** and **byte sent** for the network I/O bandwidth. For checking disk I/O, you can use the following command:

```
psutil.disk_io_counters()
```

Output:

```
>>> psutil.disk_io_counters()
sdiskio(read_count=6234, write_count=25723, read_bytes=174920704, write_bytes=728805376, read_time=16292, write_time=225216, read_merged_count=0, write_merged_count=28040, busy_time=15144)
```

For checking the network I/O bandwidth:

```
psutil.net_io_counters()
```

Output:

```
>>> psutil.net_io_counters()
snetworkio(bytes_sent=5377049, bytes_recv=1181422073, packets_sent=76538, packets_recv=79131, errin=0, errout=0, dropin=0, dropout=0)
>>> exit()
```

Exit from the Python shell using `exit()`.

After checking the disk I/O and network bandwidth, you noticed the amount of **byte read** and **byte write** for disk I/O and **byte received** and **byte sent** for the network I/O bandwidth.

Basics rsync command

`rsync(remote sync)` is a utility for efficiently transferring and synchronizing files between a computer and an external hard drive and across networked computers by comparing the modification time and size of files. One of the important features of `rsync` is that it works on the **delta transfer algorithm**, which means it'll only sync or copy the changes from the source to the destination instead of copying the whole file. This ultimately reduces the amount of data sent over the network.

The basic syntax of the `rsync` command is below:

```
rsync [Options] [Source-Files-Dir] [Destination]
```

Some of the commonly used options in rsync command are listed below:

Options	Uses
-v	Verbose output
-q	Suppress message output
-a	Archive files and directory while synchronizing
-r	Sync files and directories recursively
-b	Take the backup during synchronization
-z	Compress file data during the transfer

Example:

1. Copy or sync files locally:

```
rsync -zvh [Source-Files-Dir] [Destination]
```

2. Copy or sync directory locally:

```
rsync -zavh [Source-Files-Dir] [Destination]
```

3. Copy files and directories recursively locally:

```
rsync -zrvh [Source-Files-Dir] [Destination]
```

To learn more about *rsync* basic command, check out [this link](#).

Example:

In order to use the rsync command in Python, use the **subprocess** module by calling **call** methods and passing a list as an argument. You can do this by opening the python3 shell:

```
python3
```

Now, import the subprocess module and call the **call** method and pass the arguments:

```
import subprocess  
src = "<source-path>" # replace <source-path> with the source directory
```

```
dest = "<destination-path>" # replace <destination-path> with the destination directory
```

```
subprocess.call(["rsync", "-arq", src, dest])
```

By using the above script, you can sync your data recursively from the source path to the destination path.

Exit from the Python shell using `exit()`.

Multiprocessing

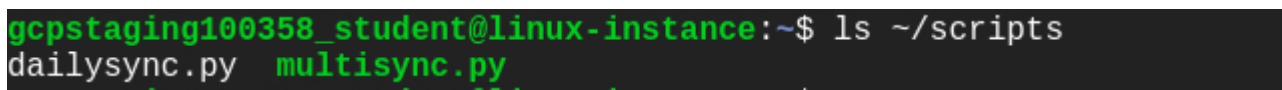
Now, when you go through the hierarchy of the subfolders of `/data/prod`, data is from different projects (e.g., , beta, gamma, kappa) and they're independent of each other. So, in order to efficiently back up parallelly, use **multiprocessing** to take advantage of the idle CPU cores. Initially, because of CPU bound, the backup process takes more than 20 hours to finish, which isn't efficient for a daily backup. Now, by using **multiprocessing**, you can back up your data from the source to the destination parallelly by utilizing the multiple cores of the CPU.

User practice

Navigate to the script/ directory using the command below:

```
ls ~/scripts
```

Output:



```
gcpstaging100358_student@linux-instance:~$ ls ~/scripts
dailysync.py  multisync.py
```

Now, you'll get the Python script `multisync.py` for practice in order to understand how multiprocessing works. We used the **Pool** class of the **multiprocessing** Python module. Here, we define a run method to perform the tasks. Next, we have a few tasks. Create a `pool` object of the **Pool** class of a specific number of CPUs your system has by passing a number of tasks you have. Start each task within the `pool` object by calling the `map` instance method, and pass the **run** function and the list of tasks as an argument.

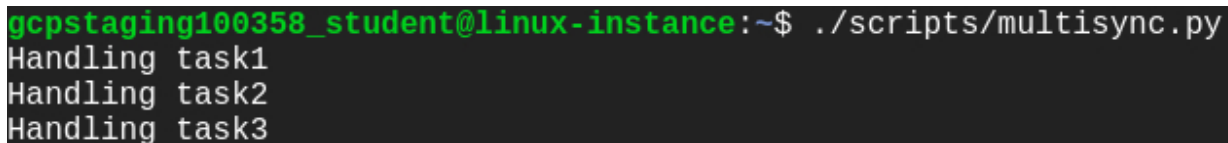
Now, grant executable permission to the `multisync.py` Python script for running this file.

```
sudo chmod +x ~/scripts/multisync.py
```

Run the `multisync.py` Python script:

```
./scripts/multisync.py
```

Output:

A terminal window with a dark background. The prompt is 'gcpstaging100358_student@linux-instance:~\$'. The command './scripts/multisync.py' has been entered. The output shows three lines: 'Handling task1', 'Handling task2', and 'Handling task3'.

```
gcpstaging100358_student@linux-instance:~$ ./scripts/multisync.py
Handling task1
Handling task2
Handling task3
```

To learn more about multiprocessing, check out this [link](#).

User exercise

Now that you understand how multiprocessing works, let's fix CPU bound so that it doesn't take more than 20 hours to finish. Try applying **multiprocessing**, which takes advantage of the idle CPU cores for parallel processing.

Open the `dailysync.py` Python script in the nano editor that needs to be modified. It's similar to `multisync.py` that utilizes idle CPU cores for the backup.

```
nano ~/scripts/dailysync.py
```

Here, you have to use **multiprocessing** and **subprocess** module methods to sync the data from `/data/prod` to `/data/prod_backup` folder.

Hint: `os.walk()` generates the file names in a directory tree by walking the tree either top-down or bottom-up. This is used to traverse the file system in Python.

Once you're done writing the Python script, save the file by clicking Ctrl-o, the Enter key, and Ctrl-x.

Now, grant the executable permission to the `dailysync.py` Python script for running this file.

```
sudo chmod +x ~/scripts/dailysync.py
```

Run the `dailysync.py` Python script:

```
./scripts/dailysync.py
```