

Fix a slow system with Python

 googlecoursera.qwiklabs.com/focuses/47754

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`

You'll have 90 minutes to complete this lab.

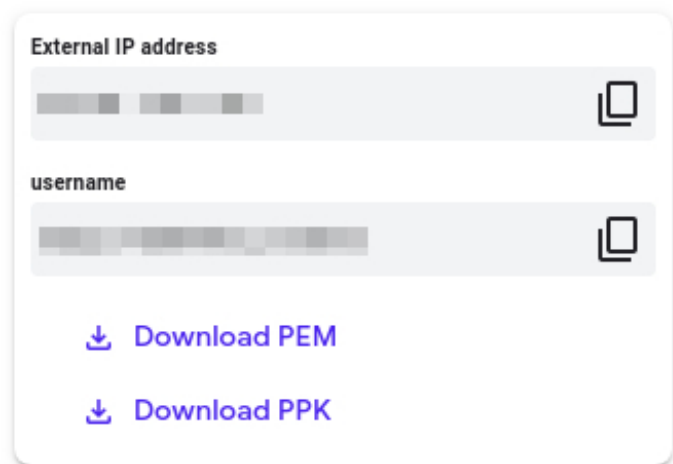
Start the lab

You'll need to start the lab before you can access the materials in the virtual machine OS. To do this, click the green “Start Lab” button at the top of the screen.

Note: For this lab you are going to access the **Linux VM** through your **local SSH Client**, and not use the **Google Console (Open GCP Console** button is not available for this lab).

After you click the “Start Lab” button, you will see all the SSH connection details on the left-hand side of your screen. You should have a screen that looks like this:

A green rectangular button with the text "Start Lab" in white.



External IP address

username

[Download PEM](#)

[Download PPK](#)

Accessing the virtual machine

Please find one of the three relevant options below based on your device's operating system.

Note: Working with Qwiklabs may be similar to the work you'd perform as an **IT Support Specialist**; you'll be interfacing with a cutting-edge technology that requires multiple steps to access, and perhaps healthy doses of patience and persistence(!). You'll also be using **SSH** to enter the labs -- a critical skill in IT Support that you'll be able to practice through the labs.

Option 1: Windows Users: Connecting to your VM

In this section, you will use the PuTTY Secure Shell (SSH) client and your VM's External IP address to connect.

Download your PPK key file

You can download the VM's private key file in the PuTTY-compatible **PPK** format from the Qwiklabs Start Lab page. Click on **Download PPK**.

Connect to your VM using SSH and PuTTY

1. You can download Putty from [here](#)

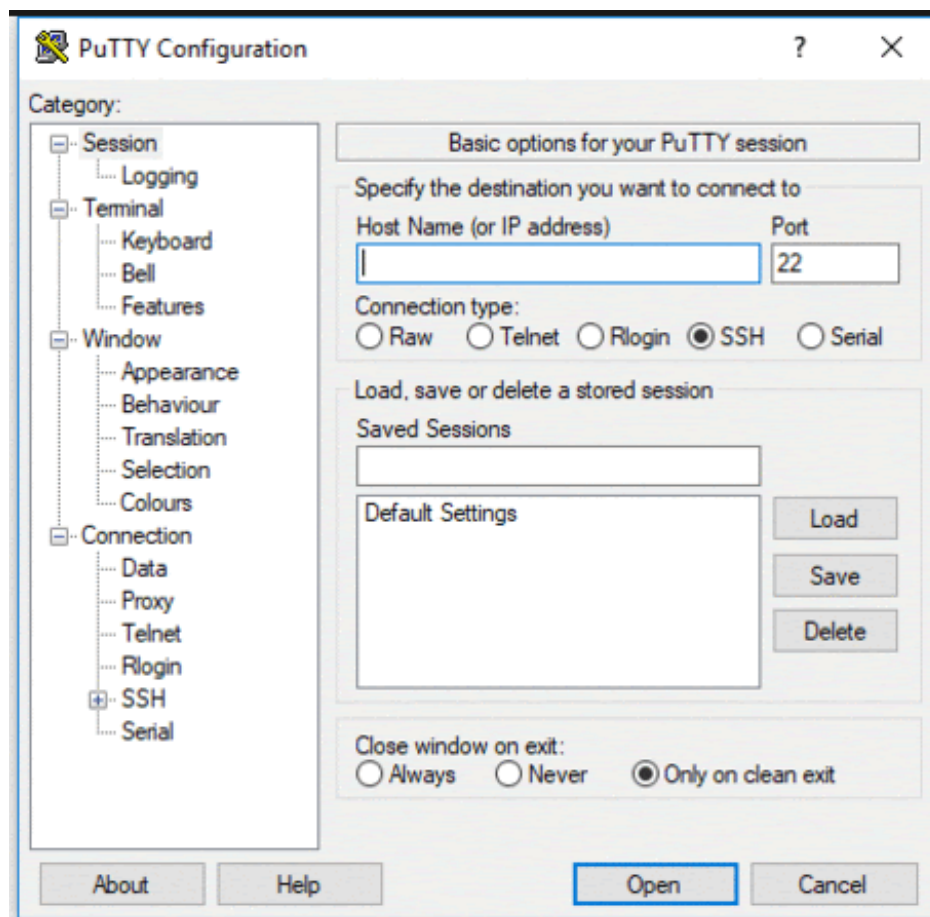
[Download PEM](#)

[Download PPK](#)



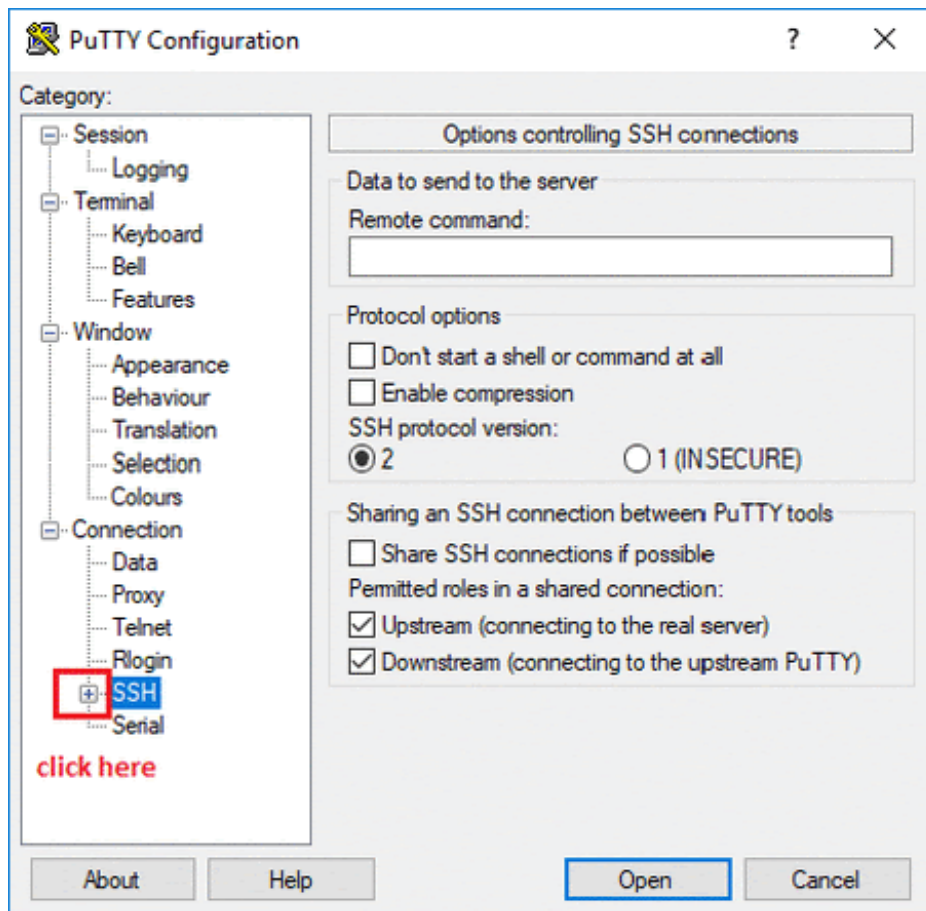
2. In the **Host Name (or IP address)** box, enter `username@external_ip_address`.

Note: Replace **username** and **external_ip_address** with values provided in the lab.



3. In the **Category** list, expand **SSH**.
4. Click **Auth** (don't expand it).
5. In the **Private key file for authentication** box, browse to the PPK file that you downloaded and double-click it.
6. Click on the **Open** button.

Note: PPK file is to be imported into PuTTY tool using the Browse option available in it. It should not be opened directly but only to be used in PuTTY.



7. Click **Yes** when prompted to allow a first connection to this remote SSH server. Because you are using a key pair for authentication, you will not be prompted for a password.

Common issues

If PuTTY fails to connect to your Linux VM, verify that:

- You entered **<username>@<external ip address>** in PuTTY.
- You downloaded the fresh new PPK file for this lab from Qwiklabs.
- You are using the downloaded PPK file in PuTTY.

Option 2: OSX and Linux users: Connecting to your VM via SSH

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.

Connect to the VM using the local Terminal application

A **terminal** is a program which provides a **text-based interface for typing commands**. Here you will use your terminal as an SSH client to connect with lab provided Linux VM.



1. Open the Terminal application.
 - To open the terminal in Linux use the shortcut key **Ctrl+Alt+t**.
 - To open terminal in **Mac (OSX)** enter **cmd + space** and search for **terminal**.
2. Enter the following commands.

Note: Substitute the **path/filename for the PEM** file you downloaded, **username** and **External IP Address**.

You will most likely find the PEM file in **Downloads**. If you have not changed the download settings of your system, then the path of the PEM key will be **~/Downloads/qwikLABS-XXXXX.pem**

```
chmod 600 ~/Downloads/qwikLABS-XXXXX.pem
```

```
ssh -i ~/Downloads/qwikLABS-XXXXX.pem username@External Ip Address
```

```
gcpstagingedit1370_student@linux-instance:~$ ssh -i ~/Downloads/qwikLABS-L923-42090.pem gcpstagingedit1370_student@35.239.106.192
The authenticity of host '35.239.106.192 (35.239.106.192)' can't be established.
ECDSA key fingerprint is SHA256:vrz8b4ayUtruFh0A6wZn60zy1oqqPEfh931o1vxlTm8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.239.106.192' (ECDSA) to the list of known hosts.
Linux linux-instance 4.9.0-9-amd64 #1 SMP Debian 4.9.168-1+deb9u2 (2019-05-13) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
gcpstagingedit1370_student@linux-instance:~$
```

Option 3: Chrome OS users: Connecting to your VM via SSH

Note: Make sure you are not in **Incognito/Private mode** while launching the application.

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.

Connect to your VM

1. Add Secure Shell from [here](#) to your Chrome browser.
2. Open the Secure Shell app and click on **[New Connection]**.

A screenshot of the 'New Connection' dialog box in a Secure Shell application. The dialog has a title bar '[New Connection]'. Below the title bar is a text field containing 'username@hostname or free form text'. Underneath are three input fields: 'username', 'hostname', and 'port'. Below these is a section for 'SSH relay server options'. Further down, there is an 'Identity:' dropdown menu set to '[default]' with an 'Import...' button next to it. Below that is an 'SSH Arguments:' text field containing 'extra command line arguments'. Then a 'Current profile:' dropdown set to 'default'. Finally, a 'Mount Path:' text field containing 'the default path is the user's home directory'. At the bottom left are buttons '[DEL] Delete' and 'Options'. At the bottom right are buttons 'SFTP Mount', 'SFTP', and '[ENTER] Connect'.

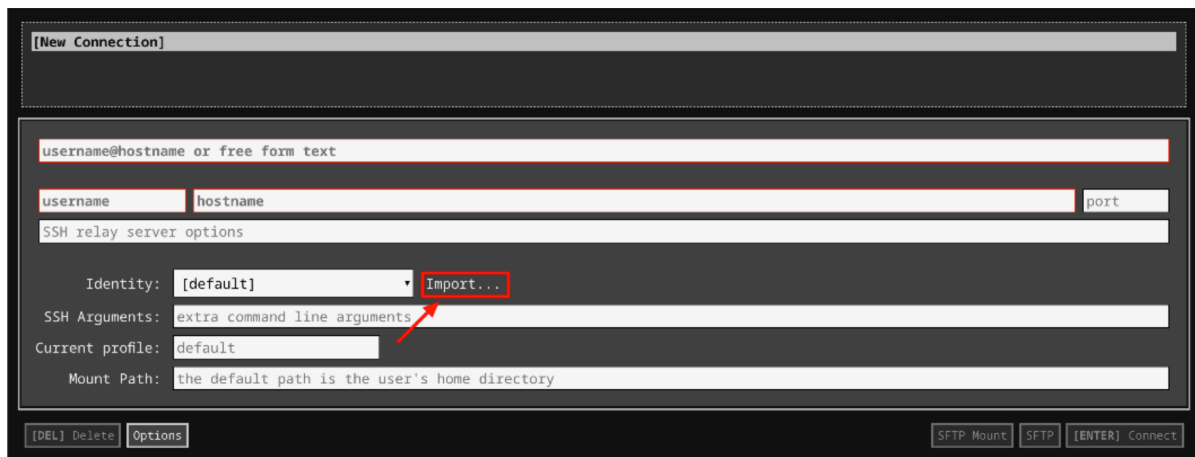
3. In the **username** section, enter the username given in the Connection Details Panel of the lab. And for the **hostname** section, enter the external IP of your VM instance that is mentioned in the Connection Details Panel of the lab.

A screenshot of the 'New Connection' dialog box, similar to the one above. In this version, the 'username' and 'hostname' input fields are highlighted with red rectangular boxes. A red arrow points from the left towards the 'username' field.

4. In the **Identity** section, import the downloaded PEM key by clicking on the **Import...** button beside the field. Choose your PEM key and click on the **OPEN** button.

Note: If the key is still not available after importing it, refresh the application, and select it from the **Identity** drop-down menu.

5. Once your key is uploaded, click on the **[ENTER] Connect** button below.



6. For any prompts, type **yes** to continue.
7. You have now successfully connected to your Linux VM.

You're now ready to continue with the lab!

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:

```
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
```

Click *Check my progress* to verify the objective. Backup from production to production backup server

Congratulations!

You've successfully synced or copied data from different multimedia projects from the source location to the destination using `rsync` command used in the Python script. And you've reduced the backup time by taking advantage of the idle CPU cores for parallel processing using multiprocessing. Backing up a large amount of data from one place to another place will definitely help you in the field of IT.

End your lab

When you have completed your lab, click **End Lab**. Qwiklabs removes the resources you've used and cleans the account for you.

You will be given an opportunity to rate the lab experience. Select the applicable number of stars, type a comment, and then click **Submit**.

The number of stars indicates the following:

- 1 star = Very dissatisfied
- 2 stars = Dissatisfied
- 3 stars = Neutral
- 4 stars = Satisfied
- 5 stars = Very satisfied

You can close the dialog box if you don't want to provide feedback.

For feedback, suggestions, or corrections, please use the **Support** tab.