Absolutely! 😊 Let's start **from the beginning**, step-by-step.

**🧠 What Is systemd?**

systemd is the **core part** of most modern Linux systems (like Ubuntu, Fedora, etc.). It **manages**:

* Starting the system when the computer turns on
* Running background services
* Logging
* Scheduling tasks (using **timers**)

Think of systemd like the **conductor of an orchestra**, making sure everything starts at the right time and runs smoothly.

**✅ Why Learn About Task Scheduling?**

Sometimes you want your system to **do things automatically**, like:

* Backing up files every day
* Sending a report every Monday
* Cleaning up old files every week

You can do that using:

1. cron jobs (easy and traditional)
2. systemd timers (more modern, powerful)

Since you're new, let's **start with cron**, and then I’ll explain the same with systemd after that.

**🕒 PART 1: CRON JOBS (The Traditional Way)**

**🔹 What is cron?**

cron is a service that **runs commands automatically at scheduled times**.

**🔹 How to Use It?**

You edit a special file called the **crontab** (short for "cron table").

To open your crontab:

crontab -e

(First time, it may ask to choose an editor — choose nano if unsure.)

**🔹 Example 1: Run a Script Every Day at 9 AM**

Let's say you want to run a script every morning.

1. First, create your script:

nano /home/yourname/backup.sh

Put this inside:

#!/bin/bash

tar czf /home/yourname/backup.tar.gz /home/yourname/Documents

Save and exit (Ctrl + X → Y → Enter)

1. Make it executable:

chmod +x /home/yourname/backup.sh

1. Now, add a cron job:

crontab -e

Add this line:

0 9 \* \* \* /home/yourname/backup.sh

This means:  
**At 09:00 (9 AM), every day**, run /home/yourname/backup.sh.

**🔹 What Do the cron Time Fields Mean?**

\* \* \* \* \* command\_to\_run

│ │ │ │ │

│ │ │ │ └── Day of week (0 = Sunday)

│ │ │ └──── Month (1 - 12)

│ │ └────── Day of month (1 - 31)

│ └──────── Hour (0 - 23)

└────────── Minute (0 - 59)

So:

* 0 9 \* \* \* → Run every day at **9:00 AM**
* 0 0 \* \* 0 → Run every **Sunday at midnight**
* \*/5 \* \* \* \* → Run **every 5 minutes**

**🕒 PART 2: systemd TIMERS (Modern Way)**

If you're curious to try systemd (the newer way), here's an example:

**🔹 Step 1: Create a script**

nano /home/yourname/hello.sh

Put this inside:

#!/bin/bash

echo "Hello from systemd!" >> /home/yourname/hello.log

Make it executable:

chmod +x /home/yourname/hello.sh

**🔹 Step 2: Create a .service file**

sudo nano /etc/systemd/system/hello.service

Paste this:

[Unit]

Description=Say Hello

[Service]

ExecStart=/home/yourname/hello.sh

**🔹 Step 3: Create a .timer file**

sudo nano /etc/systemd/system/hello.timer

Paste this:

[Unit]

Description=Run hello script every minute

[Timer]

OnBootSec=1min

OnUnitActiveSec=1min

Unit=hello.service

[Install]

WantedBy=timers.target

**🔹 Step 4: Enable and start the timer**

sudo systemctl daemon-reexec

sudo systemctl daemon-reload

sudo systemctl enable hello.timer

sudo systemctl start hello.timer

Now the script will run every minute, and you'll see the file /home/yourname/hello.log being updated.

**✅ Summary**

| **Feature** | **cron** | **systemd Timers** |
| --- | --- | --- |
| Easy to use | ✅ Yes | ❌ Slightly complex |
| Built-in logs | ❌ Not automatic | ✅ Logs are in journal |
| Good for | Simple tasks | Advanced or system tasks |
| Always running | Needs crond service | systemd is always running |

Would you like me to walk you through **a simple cron example live** with your system now? I can help step-by-step.

Absolutely! Let’s break this **big explanation** down into **simple steps**, so you can understand what's happening.

**🔧 GOAL OF THIS DEMO**

You're learning how to **automatically back up your /etc folder once a day** using:

🛠️ **Systemd Timer Units** — These are part of systemd, the system manager on most modern Linux systems.

We’ll create:

* A **service unit** (what to do – backup using tar)
* A **timer unit** (when to do it – daily)

**📦 What Is a systemd Timer?**

Think of it like an **alarm clock** ⏰ for running Linux commands.  
Instead of waking you up, it runs **scripts or services** at scheduled times.

**📝 What You’ll Create**

You’ll create **two files**:

1. **backup.service** → Tells systemd *what* to run (a backup command)
2. **backup.timer** → Tells systemd *when* to run it (daily, 5 mins after boot)

**🪜 STEP-BY-STEP GUIDE**

**✅ 1. Check Existing Timers (just to look)**

You can see current timers with:

systemctl list-timers

It shows:

* Which timers exist
* When they last ran
* When they will run next

**✅ 2. Create the Service File**

This is the action to perform — back up /etc.

Run this:

sudo systemctl edit --full --force backup.service

Paste this into the editor:

[Unit]

Description=Backup the /etc directory

[Service]

ExecStart=/bin/tar -czf /root/etc.tar.gz /etc

* ExecStart runs the tar command to back up /etc to /root/etc.tar.gz
* tar -czf compresses the backup

🔐 Note: Only root can write to /root, which is why we use sudo.

Press Ctrl+X, then Y, then Enter to save and exit.

**✅ 3. Check for Mistakes**

Make sure the file is correct:

systemd-analyze verify /etc/systemd/system/backup.service

If there's no error, it's valid!

**✅ 4. Create the Timer File**

Now we tell systemd **when** to run the backup.

Run:

sudo systemctl edit --full --force backup.timer

Paste this:

[Unit]

Description=Backup daily the /etc directory

[Timer]

OnBootSec=5min

OnUnitActiveSec=24h

Unit=backup.service

[Install]

WantedBy=timers.target

**Explanation:**

* OnBootSec=5min: run 5 minutes after the system boots
* OnUnitActiveSec=24h: repeat every 24 hours
* Unit=backup.service: run the backup.service file
* WantedBy=timers.target: allows the timer to start automatically on boot

Save with Ctrl+X, Y, and Enter.

**✅ 5. Verify the Timer**

Check for errors:

systemd-analyze verify /etc/systemd/system/backup.timer

If everything looks good — great!

**✅ 6. Enable the Timer**

Make it start automatically:

sudo systemctl enable backup.timer

(Optional, if you want it to run now too):

sudo systemctl start backup.timer

**✅ 7. Reboot to Test**

Now reboot your system:

sudo reboot

After logging back in, check:

systemctl list-timers

You should see something like:

NEXT LEFT UNIT ACTIVATES

Fri 2025-07-19 12:00:00 UTC 4min left backup.timer backup.service

This means: the timer is working, and your service will run soon!

**✅ 8. Check the Backup Result**

After it runs, check:

ls -lh /root/etc.tar.gz

You should see the backup file created!

**🎉 Summary**

| **Task** | **Tool Used** |
| --- | --- |
| Define action | backup.service |
| Define schedule | backup.timer |
| Create files | systemctl edit |
| Test files | systemd-analyze verify |
| Enable on boot | systemctl enable |
| Check timers | systemctl list-timers |

**✅ Next Step?**

If you’d like, I can help you create your **own timer** — maybe something like:

* Clearing temp files
* Running an update
* Sending a log

Just tell me what you'd like to automate, and I’ll guide you step-by-step!

Absolutely! Let's go step by step and explain this in **very simple words** so you understand how **cron jobs** work on Linux.

**🕒 What Is a Cron Job?**

A **cron job** is a way to **automatically run a command or script** at a scheduled time.

Think of it like an **alarm clock** ⏰ for commands:

* Run every 10 minutes
* Run every day at 2 AM
* Run every Monday at 6 PM
* … and so on!

Linux uses a background service called the **cron daemon** (crond) to do this.

**📁 Types of Cron Jobs**

There are **two types** of cron jobs:

| **Type** | **Used by** | **File/Command** |
| --- | --- | --- |
| **System cron** | Admin/root | /etc/crontab or files in /etc/cron.d/ |
| **User cron** | Normal users | Managed with crontab -e |

**🧠 Understanding the Cron Format**

Each line in a cron file looks like this:

\* \* \* \* \* <user> <command>

Let’s break this down:

MIN HOUR DOM MON DOW USER COMMAND

* **MIN** — Minute (0–59)
* **HOUR** — Hour (0–23)
* **DOM** — Day of month (1–31)
* **MON** — Month (1–12)
* **DOW** — Day of week (0–7) → Sunday can be 0 **or** 7
* **USER** — Which user should run the command (only in system crons)
* **COMMAND** — The command to run

📝 In **user crons** (crontab -e), you **do not** include the USER field.

**🖥️ System Cron (/etc/crontab or /etc/cron.d/)**

These are **cron jobs for the system**, usually managed by the **root** user.

**Example:**

You create a file in /etc/cron.d/ named test:

sudo nano /etc/cron.d/test

You write this inside:

\*/10 \* \* \* \* root date >> /tmp/date

**What this means:**

* Every **10 minutes**
* As the **root user**
* Run the date command and **append** the result to the file /tmp/date

After some time, if you check /tmp/date, you'll see the current date printed many times—one line for every 10 minutes.

**👤 User Cron (Personal crontab)**

Each user on the system (even non-root) can have **their own cron jobs**.

**Commands to manage user cron:**

| **Command** | **What it does** |
| --- | --- |
| crontab -e | Edit your cron jobs |
| crontab -l | List your cron jobs |
| crontab -r | Remove all your cron jobs |

**Example:**

1. Type:
2. crontab -e
3. Add this line:
4. \*/5 \* \* \* \* date >> /tmp/mydate

This will run the date command every 5 minutes and write the result to /tmp/mydate.

❗ No need to include a username here — it's for **your** user only.

**🧪 Try It Out**

1. Create a file:
2. crontab -e
3. Add a test line:
4. \*/2 \* \* \* \* echo "Hello from cron" >> /tmp/testcron.txt
5. Wait a few minutes and check the file:
6. cat /tmp/testcron.txt

**🧹 To Clean Up**

To delete your user cron jobs:

crontab -r

**✅ Summary**

| **Concept** | **Explanation** |
| --- | --- |
| cron | Tool to schedule tasks |
| crond | Background service that runs cron jobs |
| crontab -e | Edit your own scheduled tasks |
| /etc/cron.d/ | Place for system-level cron jobs |
| Time Format | MIN HOUR DOM MON DOW |
| Wildcard \* | Means "every" (e.g., every minute/hour) |
| Redirect >> | Adds output to a file |

Would you like me to help you create a **real example cron job** on your system? I can guide you live with it!

Absolutely! Let me explain this to you in **very simple words** so you really get what’s going on with **cron jobs** in Linux.

**🕐 What is a Cron Job?**

A **cron job** is like a **reminder** for your computer.  
You tell it: “Hey, every day at 2 PM, run this command.”  
Or: “Every 10 minutes, do this task.”

The Linux system will **automatically** run those commands at the times you tell it to.

**✅ Before Using Cron: Is It Running?**

Before using cron, we check if the cron system is actually working.

Run this command:

systemctl status cron

If it says:

* **Active** (🟢 green): Good, it's running.
* **Enabled**: Good, it will start on boot.

**📁 Where to Put Cron Jobs?**

There are **two places** to put scheduled tasks:

| **Type of Cron** | **Used By** | **Where You Put It** |
| --- | --- | --- |
| **System Cron** | For the whole system | Files like /etc/crontab or /etc/cron.d/ |
| **User Cron** | Just for one user | Use crontab -e |

**🖥️ System Cron Job (Example)**

You're the system admin (root user), and you want to run a command every 10 minutes.

**Step-by-step:**

1. Open a new cron file:
2. sudo nano /etc/cron.d/test
3. Add this line:
4. \*/10 9-17 \* \* \* root date >> /root/date

This means:

* + \*/10 → Every 10 minutes
  + 9-17 → From 9 AM to 5 PM
  + \* \* \* → Every day, every month, any day of week
  + root → Run the command as the root user
  + date >> /root/date → Run the date command and save the time in a file called /root/date

1. Save and exit (Ctrl + X, then Y, then Enter)

✅ Now every 10 minutes between 9–5, the system will record the current time in /root/date.

**👤 User Cron Job (Example)**

You can also make **your own cron jobs** without needing admin access.

**Step-by-step:**

1. Open your crontab:
2. crontab -e
3. Add this line:
4. \* \* \* \* \* date >> /tmp/mytime.txt

This means:

* + Every minute
  + Run the date command
  + Save the output into /tmp/mytime.txt

1. Save and exit (Ctrl + X, then Y, then Enter)

✅ Every minute, the system writes the current time into /tmp/mytime.txt.

**🧪 How to Check if It's Working**

You can check the output file:

cat /tmp/mytime.txt

Or to see system logs as it happens, use:

sudo journalctl -f

You’ll see messages like:

CRON[1234]: (your\_username) CMD (date >> /tmp/mytime.txt)

**🧹 To Remove Your Cron Job**

To stop the task from running every minute:

crontab -r

✅ This deletes your user cron jobs.

**🧠 Final Tips**

**Cron Time Format:**

\* \* \* \* \* <user> <command>

│ │ │ │ └── day of week (0-7)

│ │ │ └──── month (1-12)

│ │ └────── day of month (1-31)

│ └──────── hour (0-23)

└────────── minute (0-59)

**Example:**

30 14 \* \* 1 root /usr/bin/backup.sh

👉 Runs every **Monday at 2:30 PM** as **root**, running the script /usr/bin/backup.sh.

**🚀 Summary**

| **Task** | **Command or Explanation** |
| --- | --- |
| Check if cron is running | systemctl status cron |
| Create a system cron job | sudo nano /etc/cron.d/myjob |
| Create a user cron job | crontab -e |
| View user cron jobs | crontab -l |
| Remove user cron jobs | crontab -r |
| View output or logs | cat /tmp/file.txt or sudo journalctl -f |

If you’d like, I can walk you through creating a simple one on your system right now.

Do you want help doing that?

Absolutely! Let me break it down for you in **simple words** — this is all about the **at command**, which is used to **schedule one-time tasks** on Linux.

**🧠 What is at?**

🔹 at is used to tell your system:

"Hey, run this command at a specific time — **just once**."

So if you want something to happen **only one time**, and not daily or weekly, at is perfect.

**🛠️ Make Sure at is Working**

First, you check if the atd service (the "at daemon") is running:

sudo systemctl status atd

You should see:

* Active: ✅ Yes
* Enabled: ✅ Yes (starts at boot)

If it's not running, you can start it with:

sudo systemctl start atd

sudo systemctl enable atd

**✅ How to Create a One-Time Job**

**Step 1: Schedule a task**

You use the at command followed by the time:

at 16:00

That means "Run at 4 PM today".

You can also use fun keywords like:

* at now + 1 minute
* at teatime (4 PM)
* at midnight
* at 10:30 AM August 23

**Step 2: Type your command**

After you type the at command, it opens a mini-editor.

Here you type what you want to run, like:

date >> /tmp/oneday.txt

✅ This means “Save the current date into a file.”

**Step 3: Finish with Ctrl + D**

You **must press Ctrl + D** to save and exit the job editor.

**👀 View Your Scheduled Jobs**

To see what jobs are waiting to run:

atq

You’ll see a list of jobs like:

1 Thu Jul 25 16:00:00 2025 a username

2 Fri Aug 23 03:31:00 2024 a username

**🧾 View What a Job Will Do**

To see the details of a scheduled job:

at -c <job-number>

Example:

at -c 2

Scroll to the bottom of the output to see the command that will be run.

**❌ Remove a Job**

If you made a mistake or don’t need it anymore:

atrm <job-number>

Example:

atrm 2

That job is now deleted.

**💡 Example: Full Flow**

1. Type:

at now + 2 minutes

1. Type the command:

echo "Hello from the future" >> /tmp/future.txt

1. Press Ctrl + D to save and exit.
2. In 2 minutes, the file /tmp/future.txt will have the message.

**🔁 Difference from Cron**

| **cron** | **at** |
| --- | --- |
| Runs repeatedly (daily, weekly, etc.) | Runs **only once** |
| Needs time fields like \* \* \* \* \* | Just type at 10:00 |
| Good for regular backups | Good for quick one-time tasks |

**🧠 Summary**

| **Command** | **What it does** |
| --- | --- |
| at TIME | Schedule a one-time job |
| Ctrl + D | Finish writing the job |
| atq | List all your scheduled at jobs |
| at -c <job-id> | Show what a job will do |
| atrm <job-id> | Delete a scheduled job |
| sudo systemctl status atd | Check if at is running |

Let me know if you want help scheduling a real job on your machine right now — I’ll walk you through it step by step 😊

Here's a **simple summary** of everything Andrew Mallett covered in this final wrap-up of the module about **task scheduling in Linux**:

**🧭 What Was This Module About?**

It was all about **how to schedule tasks** in Linux using 3 different methods:

**1. Systemd Timers 🕒 (Modern Way)**

✅ This is the **new way** of scheduling tasks, used on most modern Linux systems.

* You create two files:
  + A **Service Unit** → defines *what* to do (e.g. backup /etc)
  + A **Timer Unit** → defines *when* to do it (e.g. daily, after boot, every hour)
* The **timer triggers the service**.

📌 Key commands and points:

* Create with: sudo systemctl edit backup.service
* Timer uses OnCalendar=, OnBootSec=, etc.
* Only the **timer** needs to be **enabled**:  
  sudo systemctl enable backup.timer
* Use systemctl list-timers to see what’s running.
* Use systemd-analyze verify to check for errors.

**2. Cron Jobs 📅 (Traditional Way)**

✅ This is the **classic way** used for a long time in Linux.

* System-level cron:  
  Configured in /etc/cron.d/yourfile  
  (Don’t edit /etc/crontab directly — it might be overwritten!)
* User-level cron:
  + crontab -e → Edit your own cron jobs
  + crontab -l → List your cron jobs
  + crontab -r → Remove all your cron jobs

🕐 Format of cron job timing:

\* \* \* \* \* <command>

| | | | |

| | | | +––– Day of week (0-7, Sun = 0 or 7)

| | | +––––– Month (1-12)

| | +–––––––– Day of month (1-31)

| +––––––––––– Hour (0-23)

+–––––––––––––– Minute (0-59)

Example:

\*/10 9-17 \* \* \* root date >> /root/date

Runs every 10 mins from 9 AM to 5 PM, saves date to a file.

**3. at Jobs 🕓 (One-Time Tasks)**

✅ Use this if you want a task to run **just once** at a certain time.

* Make sure the atd service is running:
* sudo systemctl status atd
* To schedule a one-time task:
* at 16:00

Then type your command (e.g., date >> /tmp/hello.txt), and press Ctrl + D to save.

🧾 Useful commands:

* atq → Show scheduled jobs
* at -c <job#> → Show what a job will do
* atrm <job#> → Remove a scheduled job

**🧠 Tips to Remember**

| **Tool** | **Use Case** |
| --- | --- |
| systemd timers | For **modern**, reliable scheduling |
| cron | For **regular, repeating** tasks |
| at | For **one-time** tasks |

**📦 Where Are the Examples?**

He mentions that there's a Git repository where you can download the sample files (like the timer and service unit files). If you want, I can help you re-create them too.

**✅ What’s Next?**

The next part of the course will cover **monitoring processes** in Linux:

* top
* ps
* uptime
* and more...

Would you like:

* A **cheat sheet** summarizing all the syntax?
* Help **making your own timer/cron job**?
* Or to move on to the **process monitoring module**?

Let me know!

Absolutely! Let's break it down into the **simplest possible explanation** — like I'm explaining it to a beginner who's just starting to learn Linux. Here we go:

**🧠 What are we learning?**

This part of the course is teaching you **how to watch and control programs (called “processes”)** that are running in Linux.

Think of it like checking:

* What apps are running on your phone
* Which app is using the most battery
* And maybe stopping an app if it’s causing problems

**🐞 What’s the problem we found?**

There’s a Linux program (called a **“daemon”**) called multipathd.  
This program is meant for **big servers with complex storage** (lots of hard drives, backups, etc.).

But… in our case, we’re just using a simple test system (a virtual machine) that doesn't need it.

**❗ So what happens?**

Because our system doesn’t need multipathd, it keeps showing **errors in the system log** every few seconds. It’s like it’s confused and keeps complaining:  
🗣️ *“I can’t find the storage path! What’s wrong?!”*

**🛠️ How do we fix it?**

We have **two options**:

**✅ Option 1: Turn off multipathd**

* This is the easiest fix.
* We just **disable the service** because we don’t need it.

👉 Like turning off an app that keeps crashing.

**✅ Option 2: Fix its configuration**

* Instead of turning it off, we **tell it to ignore our simple hard drives**.
* We do this by editing a file called /etc/multipath.conf.

📝 That file controls how multipathd works.  
We add rules like:

“Hey multipathd, don’t touch disk sda or sdb.”

**💡 But wait… how do we make this automatic?**

Let’s say you are using a system that gets **reset or recreated often** (like in a test lab or classroom).  
You’d have to fix this **every single time** you create a new machine. That’s annoying!

So instead, we use a tool called **Ansible** to:

* Automatically fix the problem
* Copy the correct config file every time the system starts

💬 Think of Ansible like a robot that follows your instructions and fixes things for you.

**📂 What does the fix look like?**

We create a file using **Ansible** that says:

- name: Fix multipath config

copy:

content: |

blacklist {

devnode "sda"

devnode "sdb"

}

dest: /etc/multipath.conf

It says:

Copy this config to /etc/multipath.conf and tell multipathd to ignore sda and sdb.

**🧪 How do we check if multipathd is the problem?**

We use these Linux commands:

| **Command** | **What it does** |
| --- | --- |
| journalctl -u multipathd -f | Shows the live error messages from multipathd |
| systemctl status multipathd | Shows if the multipathd program is running |
| man 5 multipath.conf | Shows the manual for the config file |

**🧾 Summary in Super Simple Words**

* A program called **multipathd** is running, but we don’t need it.
* It’s causing **errors** in the logs.
* We can **turn it off** or **reconfigure it**.
* We use **Ansible** to **fix it automatically**.
* After the fix, no more annoying error messages!

**✅ Want me to show you the exact command to turn it off?**

Here it is:

sudo systemctl stop multipathd

sudo systemctl disable multipathd

That will:

* Stop the program now
* Stop it from starting again in the future

Would you like me to:

* Show this in pictures or diagrams?
* Explain how Ansible works more slowly?
* Help you try this on your own system?

Just tell me what you’d like help with next! 😊

No worries at all — let me explain it **step-by-step in a super simple way**, like you're learning this for the first time. 😊  
We’ll break it down like a story.

**🧩 WHAT IS HAPPENING?**

You are running a **virtual Linux machine** using a tool called **Vagrant**. Inside this Linux machine, there's a service called:

**multipathd**  
(*It helps manage complex disk setups — but you don’t need it right now!*)

This service is trying to use your local hard drives (sda, sdb), and it’s confused because those drives are **too simple** for it.

As a result, it keeps writing **error messages** to the system.

**💥 THE PROBLEM**

You're seeing messages like this over and over again:

❌ “Can’t find path to sda”  
❌ “Can’t handle sdb”

These errors keep repeating and **fill up your logs**.

**🎯 THE GOAL**

You want to **stop those error messages** by telling multipathd:

"Hey, ignore sda and sdb. Leave them alone!"

**🛠️ HOW DO WE FIX IT?**

We have two choices:

1. **Fix it manually** every time (boring and slow)
2. **Fix it automatically** using Ansible + Vagrant (smart and fast ✅)

**🤖 WHO ARE THESE TOOLS?**

Let’s explain them like characters in a story:

| **Tool** | **What it does** |
| --- | --- |
| 🧱 Vagrant | Builds and runs your virtual Linux machine |
| 🤖 Ansible | Automatically configures your machine (like magic) |

**🧰 WHAT’S THE FIX?**

We need to:

1. Create a config file called /etc/multipath.conf
2. Inside that file, **tell multipathd to ignore sda and sdb**
3. Restart the multipathd service after changing the config

**✅ HOW WE DO IT (IN PLAIN WORDS)**

1. Open the file ubuntu.yml — this is your Ansible **playbook** (a list of automatic tasks).
2. Add this new task (just copy this in):

- name: Edit Multipath

copy:

dest: /etc/multipath.conf

content: |

defaults {

user\_friendly\_names yes

}

blacklist {

devnode "^sd[a-b]"

}

notify: RestartMP

This creates a config file and tells multipathd to **ignore sda and sdb**.

1. Add the **handler** at the end of the file (this restarts the service if needed):

handlers:

- name: RestartMP

service:

name: multipathd

state: restarted

**📦 FINAL STEP: Rebuild the Machine**

Now do:

vagrant destroy # Deletes the old broken VM

vagrant up # Builds a new one, and runs your Ansible fix

🎉 Now the multipath errors are gone — all done automatically.

**👀 HOW TO CHECK IF IT WORKED**

Once the VM is running:

vagrant ssh ubuntu1

sudo journalctl -u multipathd -f

If you **don’t see new error messages**, then you fixed it!

**🔁 IN ONE LINE:**

You used Ansible and Vagrant to **automatically fix** a noisy Linux service (multipathd) by telling it to ignore your basic hard drives.

Still unclear? If yes, tell me where you're confused:

* Is it what Vagrant or Ansible are?
* Is it how the configuration works?
* Is it the Linux part?

I’ll help step-by-step! 💬

Absolutely, let me explain all of that in a **very simple and beginner-friendly** way — like you're new to Linux. 😊

**🧠 What’s This All About?**

We're learning how to **see how busy your Linux system is** using a tool called:

🔧 **uptime command**

This command tells you how hard your CPU is working over time.

**🖥️ What Does uptime Show?**

When you run:

uptime

You’ll see something like:

14:05:12 up 3:06, 1 user, load average: 0.25, 0.18, 0.10

Let’s break this down:

| **Part** | **What it means** |
| --- | --- |
| 14:05:12 | Current time |
| up 3:06 | The system has been ON for 3 hours |
| 1 user | 1 user is logged in |
| load average: | CPU load average |
| 0.25, 0.18, 0.10 | Load in the last **1, 5, and 15 minutes** |

**🤔 What Does Load Average Mean?**

The **load average** numbers tell you:

How many tasks (programs/processes) want to run **at the same time**.

It’s like how many people are waiting in line to use the CPU.

**🧮 How Busy Is "Too Busy"?**

This depends on **how many CPUs** your system has.

To check:

nproc

If you see 2, then your system has **2 CPUs (or cores)**.

Now compare the **load average** to this number:

| **Load Avg** | **CPUs** | **Status** |
| --- | --- | --- |
| 1.00 | 2 | 🟢 Not busy |
| 2.00 | 2 | 🟡 Fully busy |
| 3.00+ | 2 | 🔴 Overloaded (too busy!) |

So, a value **above 2.0** (when you have 2 CPUs) means your system is **struggling** and tasks are waiting in line.

**💡 Example**

If uptime shows:

load average: 0.25, 0.18, 0.10

And nproc shows:

2

Then this means:

* In the last 1 minute, only 0.25 tasks wanted to run at the same time.
* That's **only 12.5% CPU usage** (because 0.25 ÷ 2 = 0.125).
* So your system is very relaxed 💤

**🧪 What If You Want to Simulate a Busy System?**

The video shows creating fake CPU work using:

tar -caf share.tar.xz /usr/share/doc &> /dev/null &

Run this multiple times to **increase CPU load**.

Then run:

watch -n 2 uptime

This will refresh the uptime output every 2 seconds so you can **watch the load go up**.

**📊 Why Is This Important?**

Understanding load average helps you answer:

* Is my system **too slow** because it's **too busy**?
* Do I need to **add more CPUs**?
* Is some program causing **high CPU usage**?

**✅ TL;DR (Too Long; Didn’t Read)**

| **Thing to Check** | **Command** | **Example Result** | **Meaning** |
| --- | --- | --- | --- |
| CPU count | nproc | 2 | You have 2 CPUs |
| Load average | uptime | 0.25, 0.18, 0.10 | System is not busy |
| High CPU load? | uptime > CPUs | 3.00 vs 2 CPUs | System is overloaded (🔴) |
| Watch live load | watch uptime | Live numbers | See how busy the system gets |

Still confused? Let me know which part doesn’t make sense — I’ll explain it with diagrams or animations if needed!

Absolutely! Here’s a simple breakdown of what you just shared about the **top** command:

**What is top?**

* **top** is a command you run in the terminal that shows you what’s going on with your system *right now*.
* It shows you a list of all the programs (processes) running on your computer.
* By default, it sorts this list by which programs are using the **most CPU** (the brain of your computer).

**What do you see when you run top?**

* At the top of the screen, you get some summary info like:
  + How long the system has been running (uptime).
  + How many tasks (programs) are running.
  + How busy your CPU is.
  + How much memory (RAM) is being used.
* Below that, you get a list of processes sorted by CPU usage. The ones using the most CPU are at the top.

**Changing what you sort by — like sorting by memory usage**

* Sometimes, you want to see which programs use the **most memory** instead of CPU.
* To do this inside top, you press the **f key** (which stands for *fields*).
* This opens a list where you can pick which column to sort by.
* Use the arrow keys to find the **%MEM** (memory usage) field.
* Press **s** (for *sort*) to sort the list by memory usage instead of CPU.
* Press **q** to go back to the main screen.
* Now the list shows programs using the most memory at the top.
* Memory usage usually changes less often than CPU usage, so the list looks more stable.

**Switching back and exiting top**

* You can switch back to sorting by CPU by doing the same steps:
  + Press f, select the CPU column, press s, then q.
* When you’re done, press **q** to quit the top command and return to your normal terminal.

**Why is this useful?**

* It helps you **see what’s using your computer’s resources** right now.
* You can find programs that might be slowing your system down.
* If you see something like the **multipath daemon** using CPU or memory but you don’t need it, you might want to stop or disable it to save resources.

Does that make sense? If you want, I can even guide you through some sample commands or explain more about how to interpret the output!

Sure! Here's a simple explanation of everything you shared about **managing processes in Linux** using ps, pgrep, pkill, and kill commands:

**1. Seeing processes with ps**

* If you just type **ps**, it shows the processes running **in your current terminal session**.
* Use **ps -f** for a **full list** with more details.
* Use **ps -l** for a **long listing** with even more info.
* You can combine these options like **ps -lf** to get detailed info while keeping usernames.
* To see **all processes on the system**, including background ones, use **ps -e** (means *everything*).

**2. Finding processes with pgrep**

* To find a process by name, you might use ps combined with grep, like ps -elf | grep bash.
* But that can sometimes show the grep process itself, which is confusing.
* **pgrep bash** is better — it **only shows the process IDs** of the bash processes without extra noise.
* It’s faster and simpler than ps + grep.

**3. Working with processes in the /proc directory**

* The **/proc** directory holds folders named after process IDs.
* You can explore /proc to see info about running processes.
* The variable **$$** gives you the process ID of your current shell.
* This is useful for understanding how processes work under the hood, but you usually don’t manage processes directly here.

**4. Stopping (killing) processes**

* To stop a process, you send it a **signal**.
* The basic command is **kill** followed by the process ID.
* Example: kill 12345 sends the default **terminate signal (SIGTERM)** to the process 12345.
* To see all possible signals you can send, use **kill -l**.
* The default signal is **signal 15 (terminate)**, which asks the process nicely to stop.
* If a process doesn’t stop, you can force it with **signal 9 (kill)** which kills it immediately.

**5. Starting and managing processes**

* You can start a process to run in the background by adding **&** at the end.
  + Example: sleep 1000 & runs the sleep command for 1000 seconds in the background.
* You can find the background process with pgrep sleep.
* You can stop it with kill [processID] or more simply with:
  + **pkill sleep** — kills all sleep processes.
  + To force kill: pkill -9 sleep.

**Summary:**

* **ps** — lists processes.
* **pgrep** — finds process IDs by name, cleaner than ps + grep.
* **pkill** — kills processes by name, simpler than kill + finding IDs.
* **kill** — sends signals to processes by ID.
* **Signals** — different ways to stop a process (terminate politely or force kill).
* **&** — runs a process in the background so you keep control of your terminal.

If you want, I can show you quick example commands or explain how to check which signals to send. Just let me know!

Of course! Here's a **simple explanation** of what was covered in that lesson about **AppArmor**:

**🔐 What Is AppArmor?**

* **AppArmor** is a **security tool** built into Ubuntu (and also used by SUSE Linux).
* It’s used to **control what programs (apps)** are allowed to do.
* Even if you're **root** (superuser), AppArmor still applies, so it gives an **extra layer of protection** beyond normal file permissions.

**💡 Why Use AppArmor?**

* Many apps and services (especially system ones) run as root, which means they can do anything.
* AppArmor **limits what even root-owned processes can do**, which makes your system more secure.
* It helps prevent programs from misbehaving, accessing files they shouldn't, or being used in attacks.

**🧰 What Will You Learn in This Module?**

1. **What AppArmor is** and why it's useful.
2. **How to check if AppArmor is running.**
   * It’s installed and running by default in Ubuntu 20.04.
   * Use systemctl status apparmor or sudo aa-status to check it.
3. **What are AppArmor profiles?**
   * These are **rules** that tell Linux what a program is allowed or not allowed to do.
   * Profiles are stored in /etc/apparmor.d/.
4. **How to find out which package installed a profile:**
   * Use dpkg -S filename to see which package owns a profile file.
5. **How to see details about AppArmor:**
   * Use apt show apparmor.

**📦 Extra Tools You Might Want to Install:**

AppArmor comes with basic tools, but you can install some **extra helpful tools**:

| **Tool** | **What it does** |
| --- | --- |
| apparmor-utils | Useful tools to work with AppArmor profiles |
| apparmor-easyprof | Helps you **create new profiles easily** |
| apparmor-notify | Shows **notifications** when something is blocked |

**✅ What Happens Next?**

Now, the instructor wants you to **open the terminal** and:

* Check if AppArmor is running
* Try out some of these commands
* Install the extra tools if needed

You’ll be using and exploring AppArmor with hands-on practice.

**🧠 In short:**

* AppArmor protects apps by using strict rules.
* It adds extra security even beyond the root user.
* You’ll learn how to check its status, find and manage rules (called profiles), and install helpful tools to work with it.

Let me know if you want help with any of the specific commands or tools mentioned!

Sure! Here's a **simple explanation** of everything that was just described in the AppArmor demo:

**🧪 What’s Happening in This Demo?**

You’re learning how to **check AppArmor**, **install tools**, and **understand how it works**.

Let’s break it down step by step:

**✅ Step 1: Check if AppArmor is Running**

You use this command:

sudo systemctl status apparmor

* This checks the AppArmor service.
* You’ll probably see:
  + **"active (exited)"** → This means it started, loaded the profiles (rules), and **then stopped**. That’s normal!
  + AppArmor doesn’t need to run in the background all the time. It just sets things up when the system boots.

**✅ Step 2: Check What Profiles Are Loaded**

Use:

sudo aa-status

* This shows all the **AppArmor profiles** currently loaded.
* A **profile** is a set of rules for how a program is allowed to behave.
* You’ll see something like:
  + **28 profiles loaded**
  + **28 in "enforcing" mode**
  + **0 in "complain" mode**

**💡 Modes:**

* **Enforcing** = Block bad behavior + log it.
* **Complain** = Don’t block, just log the bad behavior (good for testing/debugging).

**✅ Step 3: Alternative Way to Check Status**

You can also use:

sudo apparmor\_status

* Same results as aa-status.
* Both commands do the same thing. Use whichever you like — **"aa"** is just shorter.

**✅ Step 4: Install More AppArmor Tools**

Use:

sudo apt install apparmor-easyprof apparmor-utils apparmor-notify

These give you:

* easyprof: Helps **create new profiles** (rules).
* utils: Gives you **extra helpful tools** to manage AppArmor.
* notify: Shows **popup messages** (useful if you're running a desktop or GUI).

🔔 **Note:** Installing apparmor-notify might install **lots of extra GUI tools**, even if you don’t use them. That’s normal — you don’t have to use them.

**✅ Step 5: List All AppArmor Commands**

After installing the tools, you can type:

aa- + [Press Tab twice]

This shows you all the **aa- commands** you can now use, like:

* aa-enforce
* aa-complain
* aa-disable
* etc.

You won’t use all of them, but you’ll learn some important ones soon.

**🧠 In Simple Terms:**

* AppArmor protects your system by **controlling what programs are allowed to do**.
* It starts at boot, **loads rules**, and then exits — totally normal.
* You can see what rules (profiles) are loaded with aa-status.
* You can **install extra tools** to make it easier to manage and create rules.
* Now you’re ready to **dig deeper** into AppArmor in the next lessons.

Let me know if you want a **cheat sheet of common AppArmor commands** or a **summary diagram** — happy to help!