Linux Basics

Lab

This work demonstrates that you can observe a live Linux system and explain resource-related risks without guessing. You will apply controlled pressure to CPU, disk, and memory, watch how the system responds, and translate those observations into clear, reviewable failure paths.

Nothing here is about diagnosing incidents or fixing performance problems. Everything is about learning how to see what is actually happening and describe it precisely.
What this work actually does

You will:

- Establish a baseline so “normal” is evidence, not intuition.
- Prove that processes are real, inspectable, and have a lifecycle.
- Apply controlled stress to CPU, disk, and memory.
- Observe how resource pressure appears in standard monitoring tools.
- Confirm recovery when pressure is removed.
- Translate each situation into one disciplined risk path.

The key mechanic is observation. You do not label problems. You describe what changed and what failed as a result.

Rules

- Observe first. Do not introduce load until a baseline is recorded.
- Make only controlled changes using the listed commands.
- Do not change system configuration or services.
- Do not introduce networking activity.
- Make only claims you can support with the tool output.
- Use the required risk format exactly.

Tools

Use only the following tools:

- `uptime`
- `top`
- `iotop`
- `ps`
- `pidof`
- `kill`

- `lsof`
- `/proc`

Do not use `journalctl`, `dmesg`, or networking tools.

The required risk format

For each workload, write one complete path:

> Asset → Entry Point → Weak Spot → Damage

Then classify the primary harm as Confidentiality, Integrity, or Availability.

Constraints:
- Entry Point must be local.
- Weak Spot must be a present-tense condition you observed.
- Damage must be something a teammate could recognize as a concrete failure.

Step 1: Baseline observation

Run uptime and record:
- Current time
- How long has the system been running
- Load averages

Run `top` and record:
- Load averages shown in `top`
- The top CPU consumer, if any
- Memory usage at baseline

This baseline serves as the reference point for all subsequent observations.

Step 2: Process lifecycle mechanics

Anchor your shell

Run ps and identify your interactive shell. Record:
- PID
- TTY
- Command name

Using `/proc` for that PID, record:

- Complete command line from `/proc/<PID>/cmdline`
- Executable path from `/proc/<PID>/exe`
- Current working directory from `/proc/<PID>/cwd`

Inspect open files and record at least three entries from `/proc/<PID>//fd/`. For each entry, include:

- File descriptor number
- Target path
- What the file appears to represent

Confirm the same open files using `lsof -p <PID>`.

Prove process termination

Open a second terminal.

From the original terminal:

- Use `pidof` to locate the PID of the new shell
- Verify the PID using ps and record the PID and TTY

Terminate the verified PID using `kill`.

Verify:

- The terminal closes
- `kill <PID> -0` fails
- `/proc/<PID>` no longer exists

Record all observations.

Step 3: Observe idle behaviour

Run `top` and do nothing else for five minutes.

Record:

- Load averages at the start
- Load averages at the end
- Whether the same processes remain present

This step exists to train recognition of stable, idle behaviour.

Step 4: Controlled CPU pressure

Record baseline in `top`:

- Load averages
- Top CPU consumer

Start the workload:

```
yes > /dev/null
```

Return to top and record:

- CPU usage of the workload
- Load averages while it runs

Stop the workload using `Ctrl+C`.

Return to top and record:

- Load averages after stopping

Write:

- One risk path for CPU pressure
- CIA classification justified using Damage only

Step 5: Controlled disk pressure

Record baseline load averages.

Start the workload:

```
dd if=/dev/zero of=/tmp/lab-io-test bs=1M count=1024
status=progress
```

While it runs, use iotop and record:

- Sustained write throughput
- Whether disk activity remains elevated

Remove the test file:

```
rm /tmp/lab-io-test
```

Write:

- One risk path for disk pressure
- CIA classification justified using Damage only

Step 6: Controlled memory pressure

Record baseline load averages.

Start one workload:

```
python3 -c "a='x'(2001024*1024); import time; time.sleep(60)"
```

or

```
perl -e '$x="x"x(20010241024); sleep 60'
```

Return to `top` and record:

- Resident memory usage
- Percent memory usage

Allow the process to exit naturally.

Confirm:

- The process disappears from `top`
- Memory usage drops

Write:

- One risk path for memory pressure
- CIA classification justified using Damage only

What to notice

- Resource pressure is visible, not theoretical.
- Processes consume resources, then release them when they exit.
- Load averages and usage change in response to real work.
- “High usage” is not, by itself, damage. Damage is what fails because of it.
- CIA classification comes after the damage is clear.

Submission

Submit a short write-up that includes:

- Baseline observations
- One complete risk path for CPU pressure
- One complete risk path for disk pressure
- One complete risk path for memory pressure
- CIA classification for each, justified from Damage only

Vague damage statements do not pass. Your explanations must be grounded in what you observed.

The system should return to its original state after each workload ends.