Lab 0: Environment Verification and First Measurement

> ***Goal:*** *Verify your environment works and run your first reproducible measurement.*

Overview

This lab is started in class (Session 2) and completed as homework. By the end, you'll have:

- A working Ubuntu VM
- Basic tools installed (gcc, perf, strace)
- Your first performance measurement

Structure (Tiered Difficulty):

Part	Difficulty	Required?	Estimated Time
Part A	Basic	✅ Yes	30-45 min
Part B	Intermediate	✅ Yes	20-30 min
Part C	Advanced	⭐ Optional	30-45 min

Deadline: Before Week 2 Lecture

Part A: Environment Setup (Basic — Required)

Step 1: Get Ubuntu VM Running

If you don't have an Ubuntu VM yet:

Recommended: VirtualBox (or VMware/UTM/Parallels)

- Download VirtualBox: https://www.virtualbox.org/
- Download an Ubuntu LTS ISO (recommended: **Ubuntu 22.04 LTS** or **Ubuntu 24.04 LTS**)
- Create VM: 2+ cores, 4GB+ RAM, 25GB+ disk
- Install Ubuntu, reboot
- See `README_VirtualBox_Setup.md` for detailed instructions

Alternative: Native Linux

- If you have Linux on bare metal, that works too

Kernel version requirement (do not skip)

Many later labs rely on modern kernel features (cgroup v2 details, eBPF + BTF). As a rule of thumb:

- **Minimum:** Linux kernel **5.10+**
- **Recommended:** Linux kernel **5.15+** (this is the default GA kernel series for Ubuntu 22.04 LTS)

Ubuntu guidance (so you do not accidentally pick an incompatible OS image):

- **Ubuntu 22.04 LTS** ships with a **5.15** kernel series by default (GA kernel).
- **Ubuntu 24.04 LTS** ships with a newer kernel series (also fine for this course).
- **Ubuntu 20.04 LTS** often runs kernel **5.4** by default (still 5.x, but may be missing features we rely on).

References:

- Ubuntu kernel lifecycle / GA vs HWE overview: https://ubuntu.com/kernel/lifecycle

You will also verify your actual kernel with `uname -r` below.

> ⚠️ ***NOT Supported (as your main course environment):***
>
> - ***WSL2****: Many labs require kernel-level features that don't work reliably in WSL*
> - ***Docker containers****: You cannot access/modify the host kernel properly (perf/eBPF/cgroups are typically blocked)*
> - ***macOS****: Different kernel, no /proc, no perf*
>
> *Note on Docker:* ***Installing Docker Engine inside your Ubuntu VM is required later (Weeks 6–7) for kind/Kubernetes labs.***

Step 2: Install Basic Tools

Open a terminal in Ubuntu and run:

```
# Update package list
sudo apt update

# Install build tools
sudo apt install -y build-essential

# Install perf
sudo apt install -y linux-tools-common linux-tools-$(uname -r)

# Install strace
sudo apt install -y strace

# Install git and curl
sudo apt install -y git curl wget
```

Step 2.5: Install "Stage 2" tools (required by Week 2)

Install these early so you do not get stuck in Week 2 labs:

```
sudo apt update
sudo apt install -y \
  python3 \
  valgrind
```

> *`python3` is used by multiple labs for parsing logs and generating CSV summaries. `valgrind` (massif + `ms_print`) is required by Week 2A.*

Step 3: Verify Installation

```
# Check GCC
gcc --version
```

```
# Check perf
sudo perf stat ls

# Check strace
strace --version
```

All commands should work without "command not found" errors.

Step 4: Run Environment Check Script

```
# Run the course environment checker (provided in week1/)
bash env_check.sh | tee lab0_env_check.txt
```

Review the output:

- `[OK]` for gcc/make/perf → Good!
- `[WARN]` for missing advanced tools (eBPF / Kubernetes / security) → OK for now, we will install them in later stages
- `[ERROR]` → Fix before continuing

✅ Part A Checklist

- [ ] Ubuntu VM boots and is accessible
- [ ] `gcc --version` works
- [ ] `sudo perf stat ls` works
- [ ] `strace --version` works
- [ ] `env_check.sh` runs with no critical errors

Part A2: Course Toolchain (Staged Installs)

Lab 0 only verifies the minimum needed for Week 1. Later labs require additional packages and kernel features. We provide staged install blocks so you can install *only when needed*.

Stage 3 (install by Week 4): eBPF labs

These packages are used in Week 4+ labs that observe the scheduler with eBPF and BTF.

```
sudo apt update
sudo apt install -y \
  clang llvm \
  libbpf-dev \
  bpftool \
  bpftrace \
  linux-headers-$(uname -r) \
  stress-ng
```

Quick checks:

```
# BTF should exist on modern Ubuntu kernels (required for some BPF workflows)
ls /sys/kernel/btf/vmlinux
```

```
# bpftrace should run (may require sudo)
sudo bpftrace -V
```

Stage 4 (install by Week 6): Kubernetes/kind labs

These labs require **Docker Engine inside your Ubuntu VM** (not "Docker as your OS environment"). We will provide exact install steps later to avoid distro/architecture pitfalls.

At minimum you will need:

- Docker Engine
- kind
- kubectl

Stage 5 (install by Week 8): storage observation tools

```
sudo apt update
sudo apt install -y sysstat

# iostat should be available
iostat -V
```

Stage 6 (install by Week 9): security labs

```
sudo apt update
sudo apt install -y \
  libcap-dev libcap2-bin \
  libseccomp-dev libseccomp2 \
  auditd
```

Environment feature checks (you will reuse these later)

```
# Kernel version (course minimum 5.10+, recommended 5.15+)
uname -r

# cgroup v2 (required for several labs)
mount | grep cgroup2 || true

# Dropping caches (used in storage/I/O labs; disruptive)
# sudo sysctl vm.drop_caches=3
```

Part B: First Measurement (Intermediate — Required)

Goal

Learn to use `perf stat` and understand what the numbers mean.

Step 1: Create a Simple Program

Create file `hello.c`:

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello from process %d\n", getpid());
    return 0;
}
```

Compile:

```
gcc -o hello hello.c
./hello
```

Step 2: Trace System Calls

```
strace ./hello
```

You'll see something like:

```
execve("./hello", ["./hello"], 0x7ffd...) = 0
...
write(1, "Hello from process 12345\n", 25) = 25
exit_group(0)                           = ?
```

Question: What system call does `printf` use under the hood?

Step 3: Measure with perf stat

```
sudo perf stat ./hello
```

Output looks like:

```
 Performance counter stats for './hello':

          0.42 msec task-clock
             1      context-switches
             0      cpu-migrations
            54      page-faults
       912,345      cycles
       456,789      instructions
```

Step 4: Understand the Numbers

Metric	What It Means

task-clock	CPU time used by your program
context-switches	Times OS paused your program
page-faults	Memory pages allocated or loaded
cycles	CPU clock cycles consumed
instructions	Machine instructions executed

Step 5: Record and Explain

Run `perf stat` **3 times** and record:

Run	task-clock (ms)	page-faults	cycles	instructions
1				
2				
3				

Questions to answer:

1. Are the numbers consistent across runs? Why or why not?
2. What is the Instructions Per Cycle (IPC = instructions / cycles)?
3. Why are there page faults for such a simple program?

✅ Part B Checklist

- ☐ Compiled and ran `hello.c`
- ☐ Ran `strace ./hello` and saw system calls
- ☐ Ran `perf stat ./hello` 3 times
- ☐ Recorded results in a table
- ☐ Answered the explanation questions

Part C: Context Switch Experiment (Advanced — Optional)

Goal

Measure how context switches scale with workload and explain why.

Step 1: Create Test Program

Create file `ctx_switch_test.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    int n = 1000;
    if (argc > 1) n = atoi(argv[1]);

    int pipefd[2];
    if (pipe(pipefd) < 0) {
        perror("pipe");
        return 1;
    }

    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        // Child: read from pipe n times
        close(pipefd[1]);
        char buf;
        for (int i = 0; i < n; i++) {
            if (read(pipefd[0], &buf, 1) < 0) {
                perror("read");
                exit(1);
            }
        }
        close(pipefd[0]);
        exit(0);
    } else {
        // Parent: write to pipe n times
        close(pipefd[0]);
        char buf = 'x';
        for (int i = 0; i < n; i++) {
            if (write(pipefd[1], &buf, 1) < 0) {
                perror("write");
                exit(1);
            }
        }
        close(pipefd[1]);
        wait(NULL);
    }

    return 0;
}
```

Step 2: Compile

```
gcc -O2 -o ctx_switch_test ctx_switch_test.c
```

Step 3: Measure

```
sudo perf stat -e context-switches,cpu-migrations ./ctx_switch_test 100
sudo perf stat -e context-switches,cpu-migrations ./ctx_switch_test 1000
sudo perf stat -e context-switches,cpu-migrations ./ctx_switch_test 10000
```

Step 4: Record Results

n	context-switches	cpu-migrations	wall time (s)
100			
1000			
10000			

Step 5: Explain

Answer these questions:

1. **Scaling:** How does context-switch count scale with n? Is it linear?
2. **Mechanism:** Why does this program cause context switches?
   - Hint: What happens when child calls `read()` but no data is available?
   - Hint: What happens when parent calls `write()` but buffer might be full?
3. **Prediction:** What would happen if we used larger buffer sizes?

✅ Part C Checklist

- ☐ Created and compiled `ctx_switch_test.c`
- ☐ Measured with n = 100, 1000, 10000
- ☐ Recorded results in table
- ☐ Explained the scaling behavior and mechanism

Submission

Create a file `lab0_report.md` with this structure:

```
# Lab 0 Report

**Name:**
**Date:**

## Part A: Environment Setup

### Environment Check Output
(Paste key lines from env_check.sh output)

### Issues Encountered
(Describe any problems and how you fixed them, or "None")

## Part B: First Measurement
```

```
### strace Output
(Paste the key system calls you observed)

**Question:** What system call does printf use?
**Answer:**

### perf stat Results

| Run | task-clock (ms) | page-faults | cycles | instructions |
|-----|-----------------|-------------|--------|--------------|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

### Explanation Questions

1. Are the numbers consistent? Why?

2. What is the IPC?

3. Why are there page faults?

## Part C: Context Switch Experiment (Optional)

(Include your table and explanations if completed)
```

Submit: `lab0_report.md` before Week 2 Lecture

Troubleshooting

"perf: command not found"

```
sudo apt install -y linux-tools-common linux-tools-$(uname -r)
```

"Permission denied" with perf

```
# Option 1: Use sudo
sudo perf stat ./program

# Option 2: Lower security level (for your own VM only)
sudo sysctl kernel.perf_event_paranoid=1
```

perf shows "not supported"

Some hardware counters may not be available in VMs. That's OK — basic counters (cycles, instructions, task-clock) should work.

VM is very slow

- Allocate more RAM (4GB minimum, 8GB recommended)
- Allocate more CPU cores (2 minimum)
- Enable VT-x/AMD-V in BIOS settings

Can't install linux-tools for your kernel version

```
# Check your kernel version
uname -r

# Install generic tools
sudo apt install -y linux-tools-generic
```

Getting Help

- **During lab workshop:** Ask the instructor or TA
- **Outside of class:** Office hours, course forum
- **Environment issues:** Don't struggle alone — ask early!

Environment problems get harder to fix under deadline pressure. Get help now.

Grading Rubric

Criterion	Points
Part A (30 points)	
Environment check passes	20
Issues documented clearly	10
Part B (60 points)	
strace output included	10
perf stat results (3 runs)	20
Explanations are correct and thoughtful	30
Part C (10 bonus points)	
Experiment completed	5
Mechanism explained correctly	5

Total: 90 points (+ 10 bonus)