# Week 2B: Tail Latency + Systematic Debugging

**Percentiles, USE method, and page-fault driven tail spikes**

# Today's Plan

**Session 1: Lecture** (80 min):

- Part 2: Modern Context — percentiles, experimental control, coordinated omission (brief), USE method
- Part 3: Case Study — memory pressure → page faults → p99 spikes
- Part 4: Lab Preview — Lab 2 (Route A)

**Session 2: Lab Workshop** (80 min):

- Start Lab 2: reproduce tail inflation under cgroup memory pressure

# Learning Objectives

By the end of this session, you should be able to:

1. Compute and interpret percentiles (p50/p90/p99)
2. Explain *why p99 happens* using OS "fast path vs slow path" thinking
3. Apply the USE method to debug a performance regression
4. Build an evidence chain linking tail spikes to major faults
5. Verify a fix with before/after measurements

# Part 2: Modern Context

## Measurement Methodology (But OS-Aware)

# Why Measurement Methodology Matters

**In cloud-native systems:**

- Services call each other many times per user action
- Containers share CPU/memory/disk/network with neighbors
- Performance is a **system property** (not just code)

**You need to:**

- Measure correctly (avoid common traps)
- Find the bottleneck (USE method)
- Explain the mechanism (OS concepts)

# The Average Lies

**Service has "typical" latency around 5ms. Is it fast?**

| Percentile | Latency |
|---|---|
| p50 (median) | 3ms |
| p90 | 8ms |
| p99 | 150ms |
| p99.9 | 800ms |

**Reality:** 1 in 100 requests waits **150ms** — 50× the median.
The "typical" hides the tail that users actually experience.

# What Are Percentiles?

**p50 (median):** 50% of requests are faster

**p95:** 95% of requests are faster

**p99:** 99% of requests are faster

```
Sorted latencies: [1, 2, 2, 3, 5, 8, 10, 15, 50, 500] ms
                   p10 p20 p30 p40 p50 p60 p70 p80 p90 p100
```

Rule of thumb:

- **p99 ≈ the slowest 1%** (you are *studying rare events*)

# How Many Samples Do You Need for p99?

If you collect **N** samples:

- p99 is about the **top 0.01·N** samples.

Examples:

- N = 100 → top 1 sample ("p99" ≈ max) → **very noisy**
- N = 1,000 → top 10 samples → better
- N = 10,000 → top 100 samples → **you can see patterns**

**Lab implication:** we run many iterations so tail is measurable.

# Why p99 Matters (Tail Amplification)

**For a user making 10 requests:**

P(hitting p99 at least once) = 1 - 0.99^10 ≈ **10%**

**For a page with 100 backend calls:**

P(hitting p99 at least once) = 1 - 0.99^100 ≈ **63%**

**Most users experience the tail, not the average.**

# p99 Is Usually an OS "Slow Path"

A helpful mental model:

- **Most requests** follow a **fast path** (hot cache, no contention)
- **A few requests** trigger a **slow path** (rare event, huge cost)

Slow paths are often *OS-mediated*:

- scheduler delays (runqueue / preemption)
- blocking I/O (disk / network)
- page faults (minor vs major)
- lock contention / futex waits

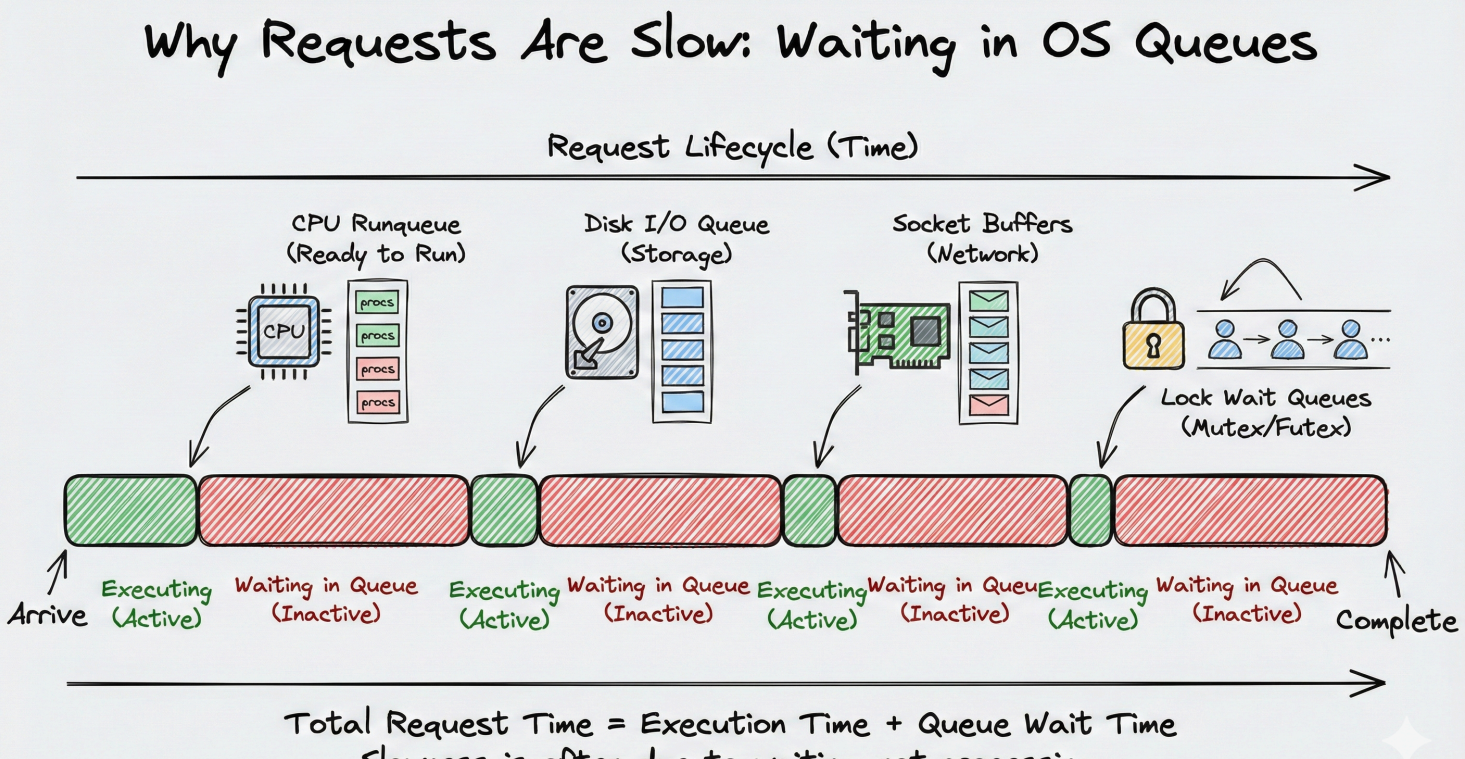
**p99 work = identify the slow path, then explain why it triggers.**

# Tail Latency Is Mostly "Waiting"

When a request is slow, it's often **not executing** — it's waiting in a queue.

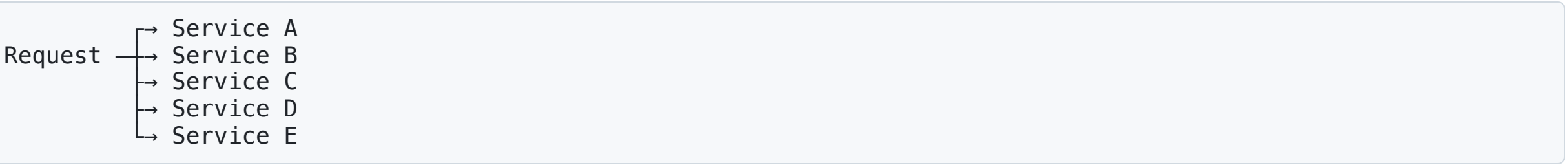
Queues exist everywhere in the OS:

- CPU runqueue (ready to run)
- disk I/O queue (waiting for storage)
- socket buffers (waiting for network)
- lock wait queues (mutex/futex)



# Tail Latency Amplification (Simple Picture)

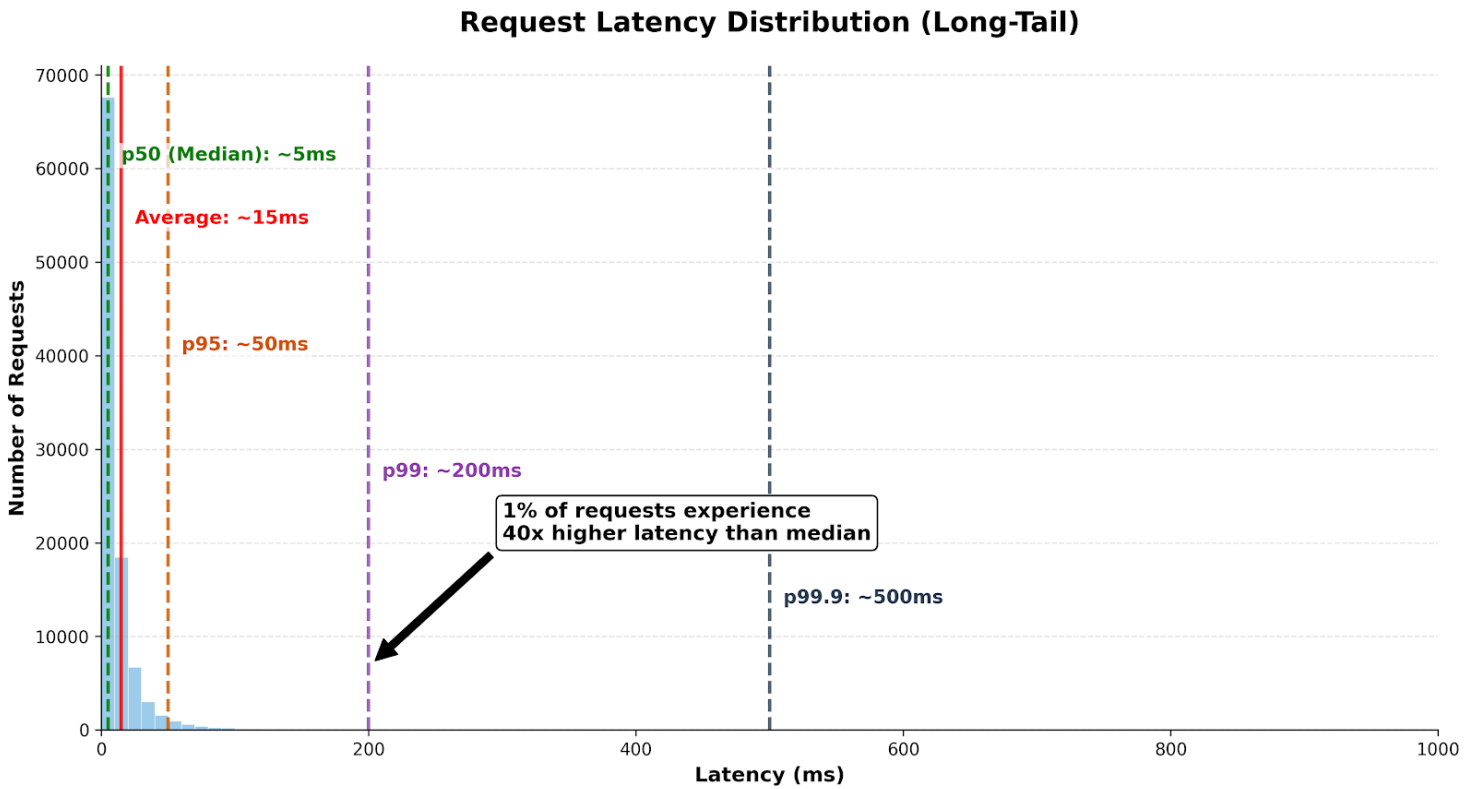
Your service calls 5 backends.




Even if each backend is "usually fast", tail composes badly:

- a single slow dependency can dominate your end-to-end latency
- tails add up via *max*, retries, timeouts, and queues

# Visualizing Latency Distributions



# Coordinated Omission (Why Many Benchmarks Lie)

**Common buggy benchmark loop:**

```
while True:
    start = time.now()
    response = send_request()  # What if this takes 1 second?
    latency = time.now() - start
    record(latency)
```

**Problem:** if one request stalls, you stop *sending* during that stall.

**Result:** you under-sample slow periods → p99 looks better than reality.

# Avoiding Coordinated Omission

Practical fixes:

1. **Constant arrival rate** ("send every X ms")
2. **Use tools designed for this** (wrk2, Gatling, HDRHistogram)
3. **Measure from the client side** (include queue time)

Quote (Gil Tene):

> "Most published latency numbers are wrong."

## The USE Method (Brendan Gregg)

For every resource, check:

- **U**tilization: How busy is it?
- **S**aturation: Is work queuing?
- **E**rrors: Are operations failing?

USE is not a tool. It's a *coverage algorithm*.

# USE Method: Resources & Tools (Starter Version)

| Resource | Utilization | Saturation | Errors |
|---|---|---|---|
| **CPU** | `top`, `mpstat` | `vmstat r`, load avg | high invol. CS |
| **Memory** | `free -m`, RSS | reclaim / swap activity | OOM, **major faults** |
| **Disk** | `iostat -x` | `await`, queue | `dmesg` I/O errors |
| **Network** | `sar -n DEV` | `ss -s` | `netstat -s` |

Today we go deep on **memory** because our lab is a memory slow-path story.

# USE Method for Memory (What It Means in OS Terms)

**Utilization ("how much memory is used?")**

- process RSS, `free -m`, `memory.current` (if in cgroup)

**Saturation ("are we fighting for memory?")**

- reclaim activity, swapping, direct reclaim (concept)
- symptoms: CPU time in kernel, stalled requests

**Errors ("did something exceptional happen?")**

- OOM kill
- allocation failures
- **major page faults** (slow path that often implies I/O)

The key idea: memory pressure changes the *execution mode* of the system.

# What Makes a Good Performance Claim?

1. **Clear metric:** p99 latency, throughput, error rate
2. **Defined baseline:** config A vs B (before vs after)
3. **Multiple runs:** repeat and report variance
4. **Controlled conditions:** same machine, same input, same environment
5. **Causal explanation:** evidence → mechanism → explanation

# Experimental Control in Shared Environments

**Common pitfalls in VMs, laptops, and shared servers:**

| Factor | Symptom | Mitigation |
|---|---|---|
| Background load | noisy variance | close apps; run on AC power |
| Cold vs warm | first run slow | warm up, discard first run |
| Core migration | jitter | pin with `taskset -c 0` |
| VM host noise | spikes | repeat runs; report percentiles |

Note:

- Dropping caches (`/proc/sys/vm/drop_caches`) is **intrusive**; do it only if the lab explicitly asks.

# Bad vs Good Performance Claims

**Bad:**

> "I optimized the code and it's faster now."

**Good:**

> "Under memory pressure, p99 rose from 2ms to 80ms. At the same time, `pgmajfault` increased 20×. After increasing `memory.max`, `pgmajfault` returned to baseline and p99 recovered."

# Understanding `time` Output (Fast Diagnosis)

```
$ time ./my_program

real    0m5.123s    # Wall clock (what the user experiences)
user    0m3.456s    # Time in user space (your code)
sys     0m1.234s    # Time in kernel (syscalls, faults, etc.)
```

Quick interpretations:

- `real ≈ user + sys` → CPU-bound
- `real >> user + sys` → blocked (I/O, contention, scheduling)
- high `sys` can indicate kernel work: syscalls, faults, reclaim

## `/usr/bin/time -v` (OS Counters That Help With Tails)

```
/usr/bin/time -v ./my_program > /dev/null
```

Useful fields:

- **Major (requiring I/O) page faults**
- Minor page faults
- Context switches (voluntary / involuntary)
- Max RSS

This is a great fallback in VMs when some `perf` events are unavailable.

# Part 2 References

**Percentiles & Latency Measurement:**

- Gil Tene, "How NOT to Measure Latency"
- HdrHistogram: github.com/HdrHistogram

**Load Testing Tools (handle coordinated omission):**

- wrk2: github.com/giltene/wrk2
- Gatling: gatling.io

**The USE Method:**

- Brendan Gregg, "The USE Method"
- Brendan Gregg, *Systems Performance*, 2nd ed. (2020)

# Part 3: Case Study

## Debugging a p99 Latency Spike (Memory Slow Path)

# The Scenario

**Alert at 2am:**

```
Service: order-service
Alert: p99 latency > 500ms (threshold: 100ms)
Started: 1:47am
Affected: ~1% of requests
```

Question:

- why only ~1%?
- what OS event hits rarely but costs a lot?

## Step 0: Don't Panic

Before touching anything:

1. Is this affecting users? (errors, timeouts)
2. Is it getting worse? (trend)
3. Did anything change? (deploy/config/traffic)

**In this case:** errors normal, no deploys, traffic steady.

# Step 1: Gather Evidence (Start With USE)

```
# Within a pod/container (illustrative)
top -bn1 # bash mode, run once and exit
cat /proc/meminfo | head
cat /proc/vmstat | egrep "pgmajfault|pgfault"
```

**Finding:** memory usage is 95% of limit.

So we ask (USE):

- utilization high (yes)
- saturation? (reclaim / paging?)
- errors? (major faults / OOM?)

## Step 2: Form Hypotheses

High memory can mean:

1. **Memory leak** (usage trending up)
2. **Traffic spike** (more concurrent requests)
3. **Noisy neighbor** (shared host pressure)
4. **Limit too tight** (app needs more headroom)

We need more evidence before changing anything.

## Step 2b: Test Hypotheses

Findings:

- Traffic is normal
- Memory has been high for hours but not growing fast

So it's likely not a sudden leak.

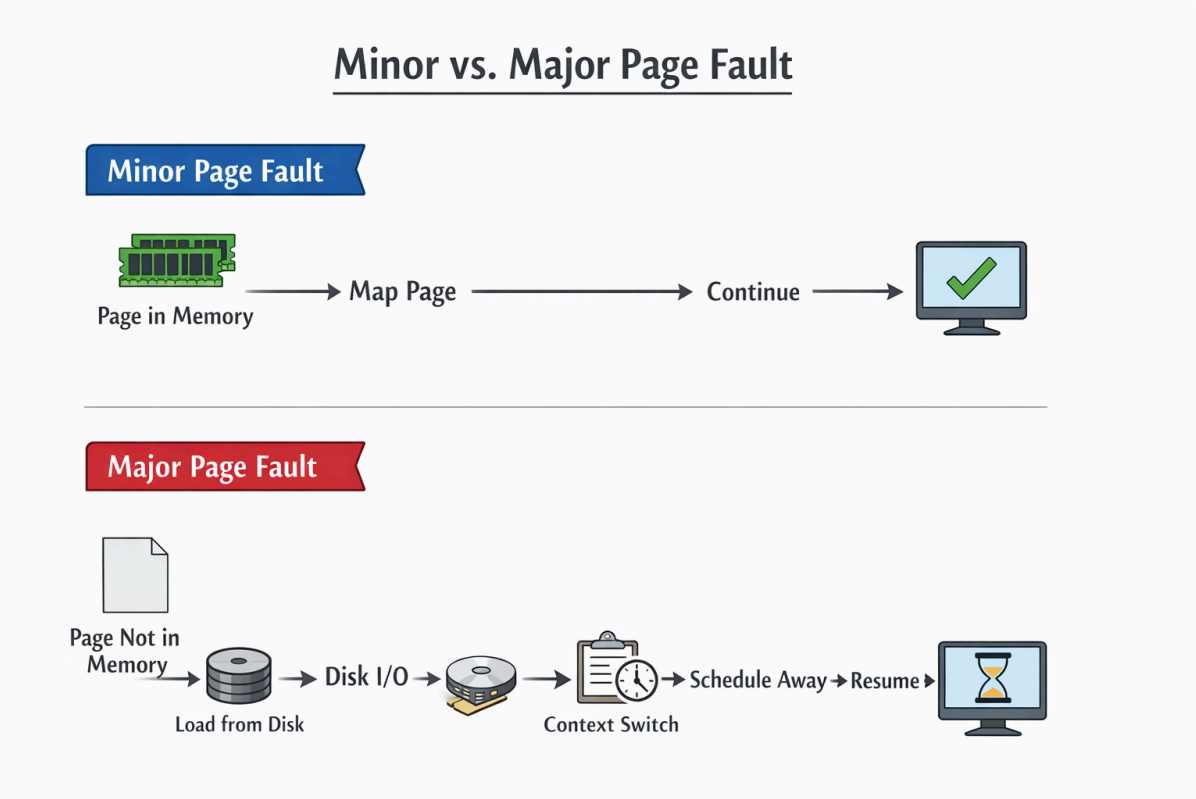
What else makes only 1% of requests slow?

# OS Concept: Minor vs Major Page Fault

A **page fault** means: the process accessed a virtual page that isn't mapped.

- **Minor fault**: the data is already in memory (e.g., page cache), map it → usually cheap
- **Major fault**: data is not in memory → must fetch from disk/swap → expensive

Major faults are classic *rare-but-huge* OS slow paths → perfect p99 suspects.



# Step 3: Look Deeper — Fault Evidence

```
cat /proc/vmstat | egrep "pgmajfault|pgfault"

# optional, if supported on your machine/VM
sudo perf stat -e page-faults,major-faults \
  ./your_workload
```

**Finding:** `pgmajfault` is **~10× higher** than baseline.

Now we have a plausible mechanism.

## Step 4: Identify the Mechanism (Evidence Chain)

**Root cause chain:**

1. memory close to limit (utilization)
2. kernel starts reclaiming pages (saturation)
3. later access triggers **major fault** (error / slow path)
4. major fault requires disk I/O; request blocks
5. blocked time inflates tail → **p99 spike**

This is why p50 may look fine: most requests never trigger the slow path.

## Step 4b: What Does the CPU Do During a Major Fault?

When a thread faults and needs I/O:

- it enters the kernel (fault handler)
- initiates I/O if needed
- then the scheduler runs *something else*

That request's latency includes:

- fault handling overhead
- I/O time
- scheduling delay (when it gets CPU again)

Tail latency often comes from **composition of multiple waits**.

## Step 5: Fix and Verify

Example fix (conceptual): increase memory limit.

```
# in Kubernetes (illustrative)
kubectl set resources deployment/order-service --limits=memory=1Gi
```

Verify after fix:

- memory pressure reduced
- major faults reduced
- p99 recovered

No verification = no confidence.

## Step 6: Prevent Recurrence

Short term:

- alert on memory usage and fault rates
- dashboards for p99 + fault metrics

Long term:

- right-size limits
- test memory pressure in staging
- reduce working set / improve locality

## Case Study Takeaways

1. p99 is often an OS slow path (rare, expensive)
2. USE method gives you a systematic search pattern
3. "High memory" is not enough; **major faults** explain the tail
4. Always verify the fix with before/after measurements

# Part 3 References


- OSTEP, Virtual Memory (Ch. 18–23): https://pages.cs.wisc.edu/~remzi/OSTEP/
- `man 5 proc` (vmstat counters): https://man7.org/linux/man-pages/man5/proc.5.html
- Brendan Gregg, *Systems Performance*, 2nd ed. (2020)

# Part 4: Lab Preview

## Lab 2: Tail Latency Debugging Under Memory Pressure (Route A)

# Lab 2 Goal

Create controlled memory pressure, then show:

1. Tail latency (p50/p90/p99) gets worse under pressure
2. Evidence indicates major faults / reclaim / paging
3. A fix restores tail latency

Focus:

- not "the exact p99 number"
- but the **mechanism + evidence chain**

# Workload: CLI That Prints Per-Iteration Latency

```
make -C week2B/lab2_latcli
./week2B/lab2_latcli/latcli --iters 20000 --workset-mb 256
```

Output format:

```
iter=123 latency_us=847
```

# Just-Enough cgroup (What + Why)

**cgroup = control group**: kernel feature to group processes and apply:

- **limits** (e.g., memory.max)
- **accounting** (e.g., memory.current)
- **events** (e.g., memory.events)

Why we use it in this lab:

- gives a **reproducible knob** for memory pressure
- isolates the experiment from the rest of the machine

## cgroup v2: The 4 Files We Care About

Assume cgroup path: `/sys/fs/cgroup/lab2`

- `cgroup.procs` → put a PID into the group
- `memory.max` → set memory limit (`max` means unlimited)
- `memory.current` → current memory usage
- `memory.events` → counters (OOM, etc.)

We will not study cgroup internals today (Week 5 will).
Today: cgroup is **experimental control**.

# Creating Memory Pressure (cgroup v2)

```
sudo mkdir -p /sys/fs/cgroup/lab2

# set 512 MiB limit
printf "%d" $((512*1024*1024)) | sudo tee /sys/fs/cgroup/lab2/memory.max

# run the workload inside the cgroup
sudo bash week2B/lab2_latcli/scripts/run_in_cgroup.sh \
  /sys/fs/cgroup/lab2 \
  ./week2B/lab2_latcli/latcli --iters 20000 --workset-mb 256
```

Sanity checks:

```
cat /sys/fs/cgroup/lab2/memory.current
cat /sys/fs/cgroup/lab2/memory.events
```

# Cleanup: Remove the Limit and Delete the cgroup

```
# remove memory limit
echo max | sudo tee /sys/fs/cgroup/lab2/memory.max

# ensure no process remains in the cgroup
cat /sys/fs/cgroup/lab2/cgroup.procs

# delete the group
sudo rmdir /sys/fs/cgroup/lab2
```

If `rmdir` fails:

- a process is still inside (move it out / kill it)

# Evidence Chain: Tail + Faults

Show at least two independent signals:

1. Tail latency percentiles computed from samples

- p50 / p90 / p99 (from the raw `latency_us` values)

2. Fault or paging evidence (choose at least one):

- `/proc/vmstat`: `pgfault`, **`pgmajfault`**
- `/usr/bin/time -v`: Major page faults
- `perf stat -e page-faults,major-faults ...` (if supported in your VM)

Goal: connect **p99 spike ↔️ major faults**.

## Deliverables

Submit:

1. `report.md` (use template)
2. `results.csv` (one row per iteration)
3. `use_checklist.md` (filled)

Grading emphasis:

- clean baseline vs pressure comparison
- tail percentiles (not only averages)
- evidence chain + OS mechanism explanation

# Part 4 References

- `man 7 cgroups` (cgroup basics)
- cgroup v2 memory controller docs (kernel): https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html
- Brendan Gregg (USE): https://www.brendangregg.com/usemethod.html
- OSTEP (VM): https://pages.cs.wisc.edu/~remzi/OSTEP/

## Let's Start Lab 2

Open `week2B/lab2_instructions.md` and start with the baseline run.