Storage Performance

Juncheng Yang

March 2

Harvard John A. Paulson
School of Engineering
and Applied Sciences

HARVARD UNIVERSITY
MADSYS
Measurement and Design of Systems Lab

Update

- No office hour today
- Cache competition is open
  - play around and ask questions
  - we will purge the database on Sunday

Agenda

- Performance measurement
  - metrics
  - Little’s law
  - open vs closed systems
  - the hidden pitfalls
- Performance optimization
  - Amdahl’s law
  - application
  - OS
  - hardware

Key Questions to Think About After Class

- What are the common pitfalls in measuring system performance?
- My friend says his disk is slow, how do I help him diagnose?
- What are different ways to improve storage system performance?

Performance Measurement

- Metrics
- Little’s law
- Open vs closed systems
- The hidden pitfalls

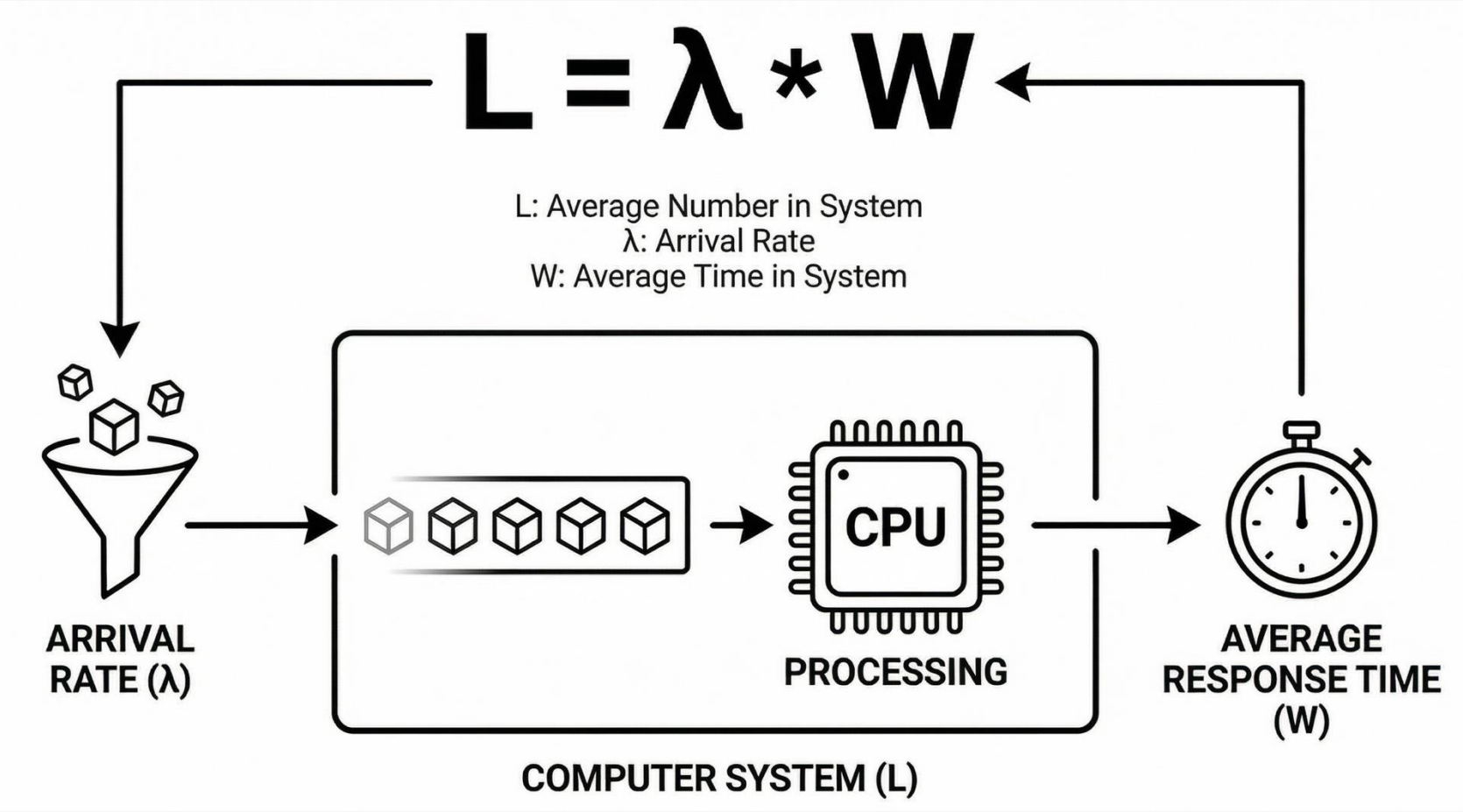
Metrics

- The big three (surface level)
  - IOPS (Input/Output Operations Per Second)
  - bandwidth
  - latency
- The hidden ones (system dynamics)
  - percentile latency (e.g., P99)
    - request fanout: latency determined by the slowest one
  - queue depth (QD)
    - #requests pending in the system
    - high QD increased bandwidth, but also latency
    - modern SSDs require a QD of 32-128 to saturate

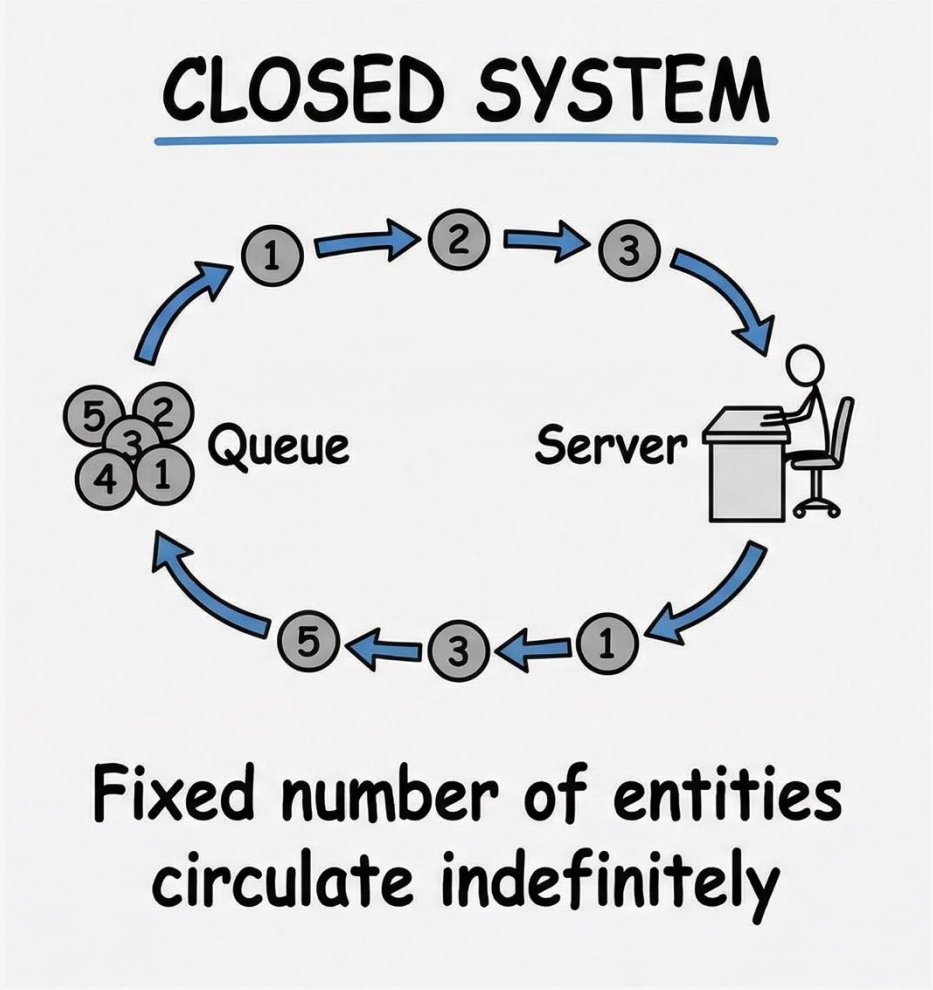
Little’s Law

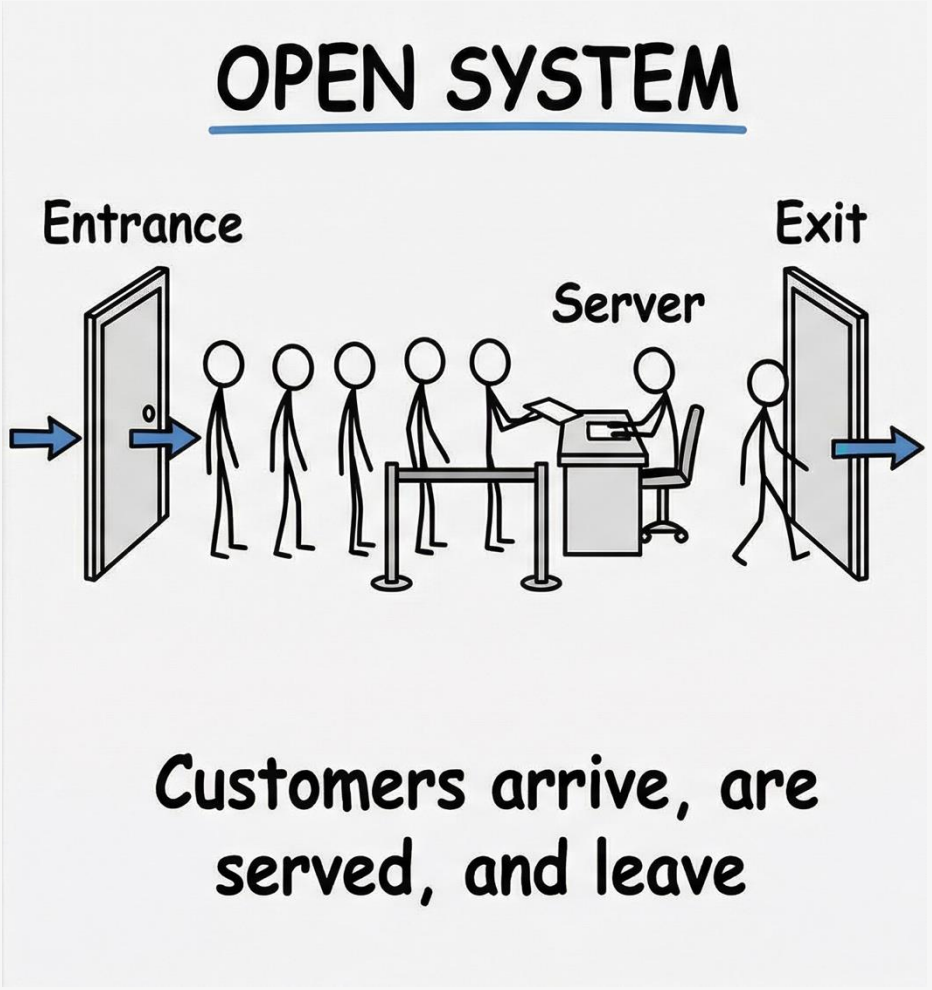
Assume disk latency 100µs

- QD=1: 10K IOPS
- to achieve 1M IOPS: QD=100

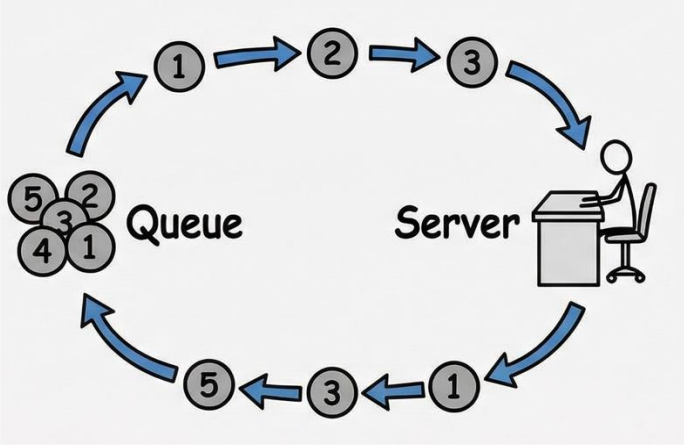


Open System and Closed System



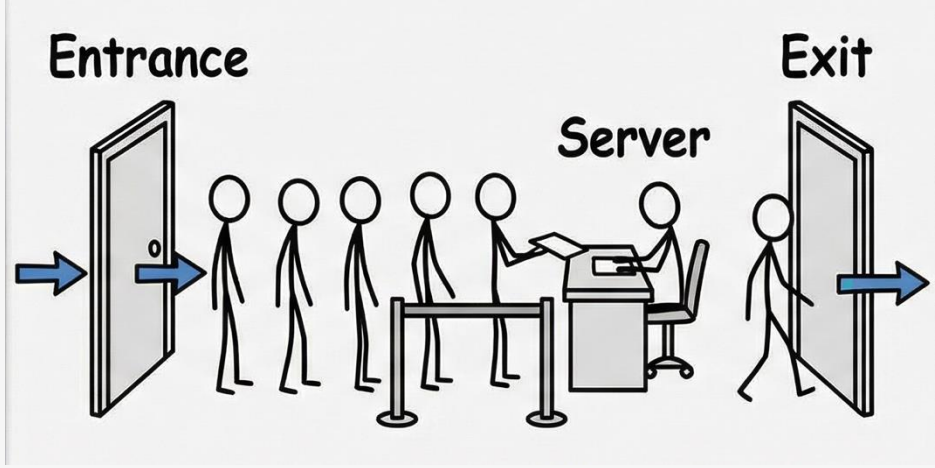


Closed Systems



- Definition
  - a fixed #workers and the next request starts after the previous one finishes
  - think about it as *N* threads each doing “issue I/O → wait → issue next”
- Key property
  - bounded concurrency: outstanding requests ≤ *N*
- Behavior near saturation
  - longer service time leads to lower *arrival rate:* clients are blocked waiting
  - throughput ≈ concurrency / latency (Little’s Law)
- Example
  - benchmark disk bandwidth using 8 threads

Open Systems



- Definition
  - requests arrive from the outside according to an arrival process, new requests can keep arriving even if previous ones haven't completed
  - think about web traffic— users don't stop clicking just because it is slow
- Key property
  - arrival rate is (mostly) independent of service completion
- Behavior near saturation
  - as utilization approaches 100%, queue length and latency can blow up dramatically (everything was fine… then p99 went vertical)
  - if arrivals exceed capacity even slightly, backlog grows without bound

Hybrid

- Most real systems are hybrids
  - open at the edge (users) but partially closed inside (e.g., thread pools)
  - performance tuning: make the hot path behave more closed via backpressure
- Many systems are partly open (NSDI’06)
  - closed within a session, but open between sessions
  - example: users wait for a page to load (closed), but they might open multiple tabs

Implication 1: More Concurrency

- Closed system
  - concurrency ↑, throughput ↑, until saturation, then stabilizes
  - you can find a stable knee by sweeping concurrency
- Open system
  - concurrency ↑, throughput ↑, up to a point, then latency *skyrockets*
  - relationship between utilization and response time is non-linear, as utilization approaches 1, response time explodes
    - basic open queue (M/M/1): utilization $\rho = \frac{\lambda}{\mu}$, λ is the arrival rate, μ is the service rate
    - mean response time $R = \frac{1}{\mu - \lambda} = \frac{1/\mu}{1-\rho}$
  - intuition: low utilization → arrivals often find the server idle → little/no waiting; high utilization → more wait

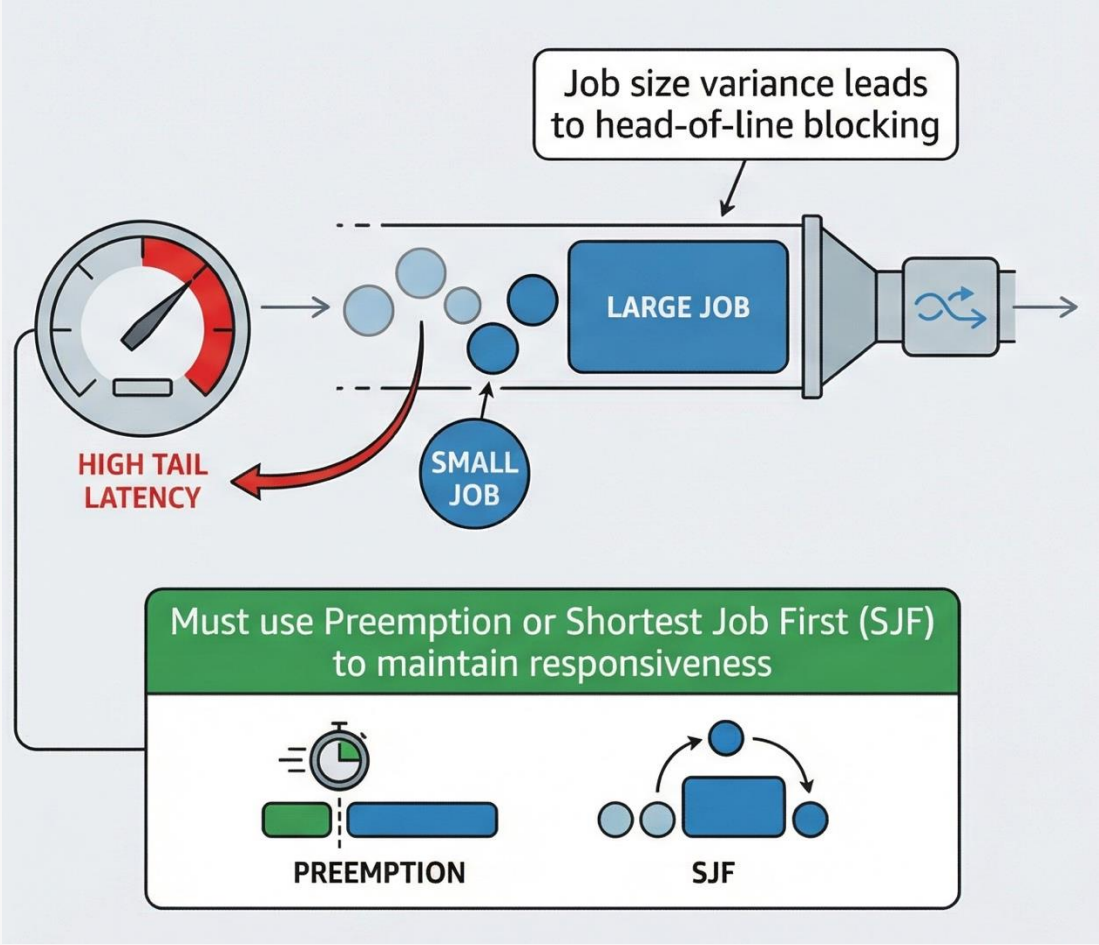
Retries make open systems worse!

Implication 2: Job Scheduling

Open system: tail latency battle

Risk of FCFS (First-Come, First-Serve)

- job size variance leads to head-of-line blocking → high tail latency
- must use Preemption or Shortest Job First (SJF) to maintain responsiveness

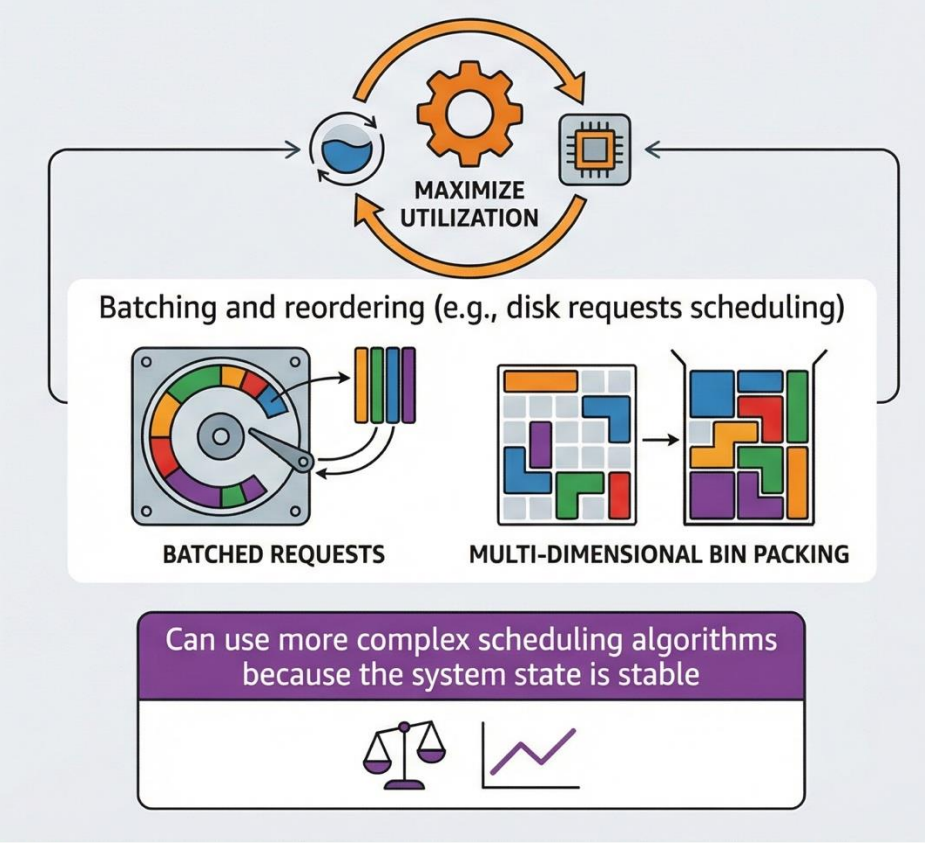


Implication 2: Job Scheduling

Closed system: maximize resource utilization

Goal: focus on throughput

- batching and reordering (disk requests scheduling)
- can use more complex scheduling algorithms (e.g., multi-dimensional bin packing) because the system state is stable



Implication 3: System Designs

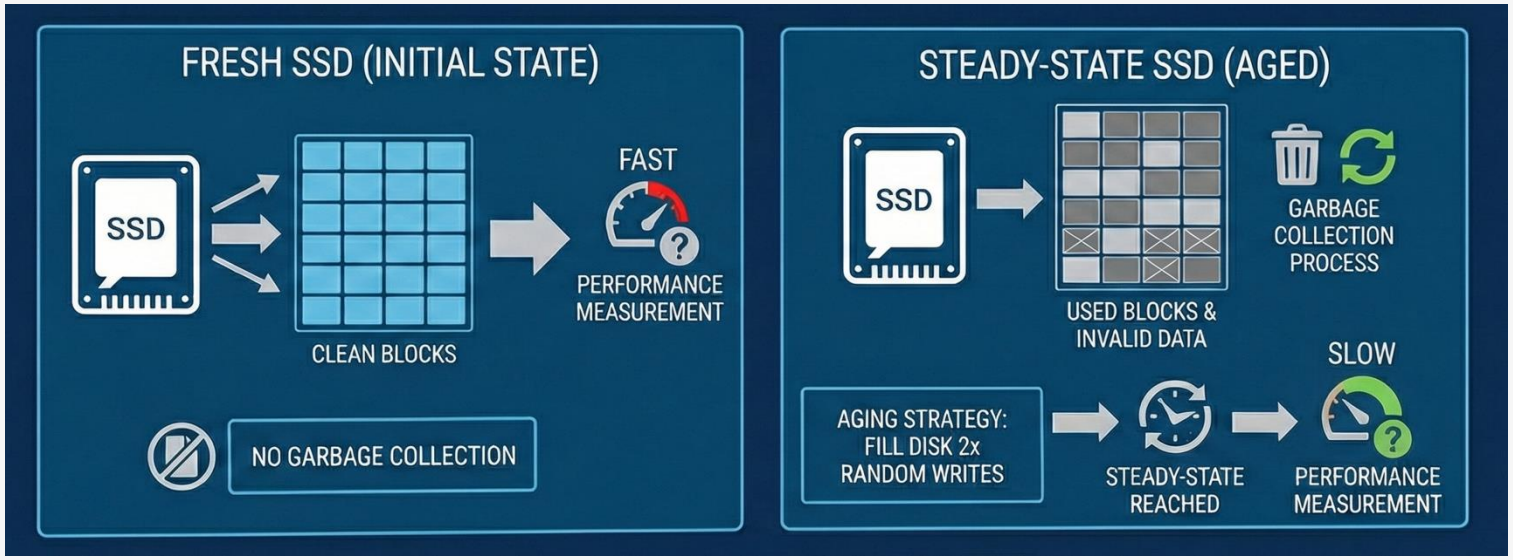
- Open systems
  - explicit backpressure
  - admission control and rate limiting
  - use deadlines on jobs
  - cancellation propagation

- Separate interactive vs batch using priority

Measuring Storage Systems: Many Pitfalls

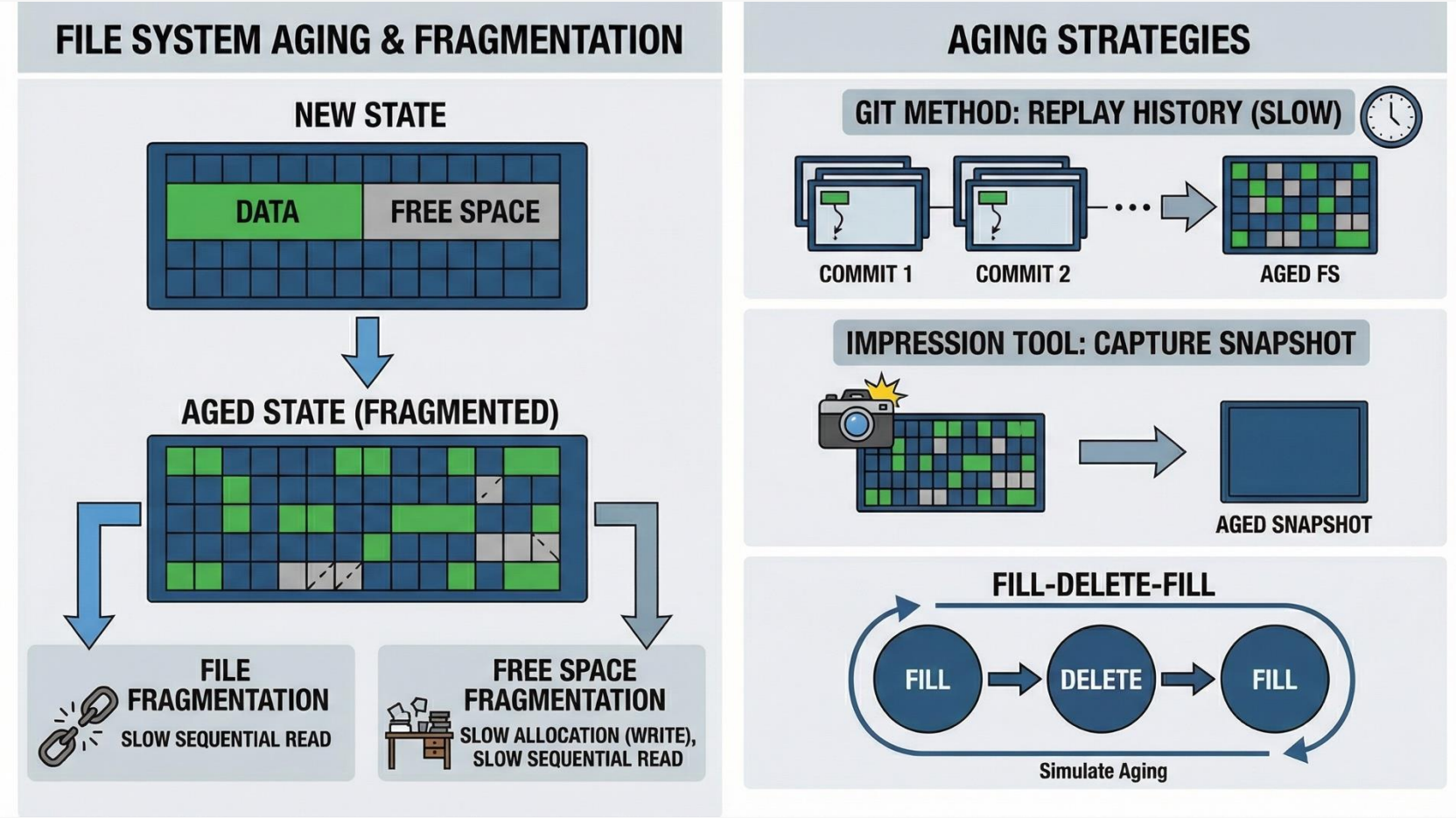
- Correct measurement is VERY HARD
  - real-world system is complex with many layers and optimizations
    - incorrect setup, measuring wrong thing
  - specifically for storage system: state is important
    - free space
    - fragmentation
    - …
  - synthetic workload is not sufficient
    - example: 80% read / 20% write, Zipf/Uniform popularity
    - real-world pattern is different and more complex, e.g., clustered access, burst, metadata-heavy

Measuring Storage Systems: The “State” Problem



- Pre-conditioning is critical for storage performance measurement
- SSD aging
  - fresh out-of-box SSD is completely different from a steady-state SSD
  - no garbage collection
  - aging strategy: fill the disk 2x capacity with random writes

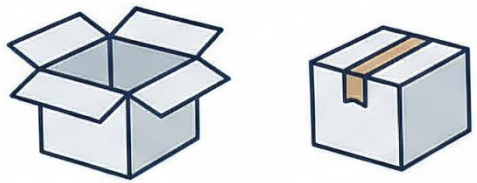
Measuring Storage Systems: The “State” Problem



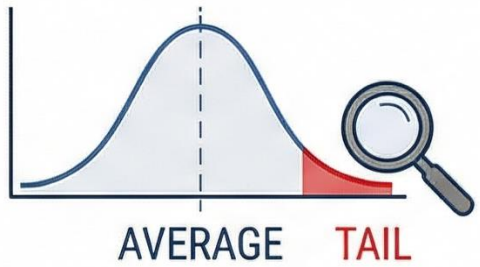
performance degrade over time because on-disk data structures become fragmented

Measuring Storage Systems: Other Pitfalls

WRONG MODEL

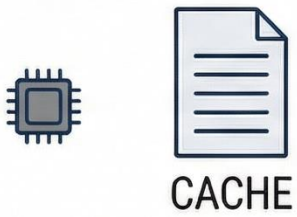


WRONG METRICS

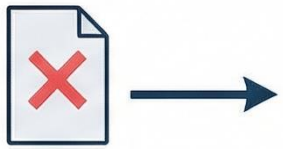


BACKGROUND NOISE

WORKLOAD TOO SMALL



PAGE CACHE (O_DIRECT) & FSYNC

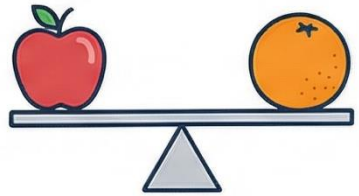


CONTENT PROBLEM

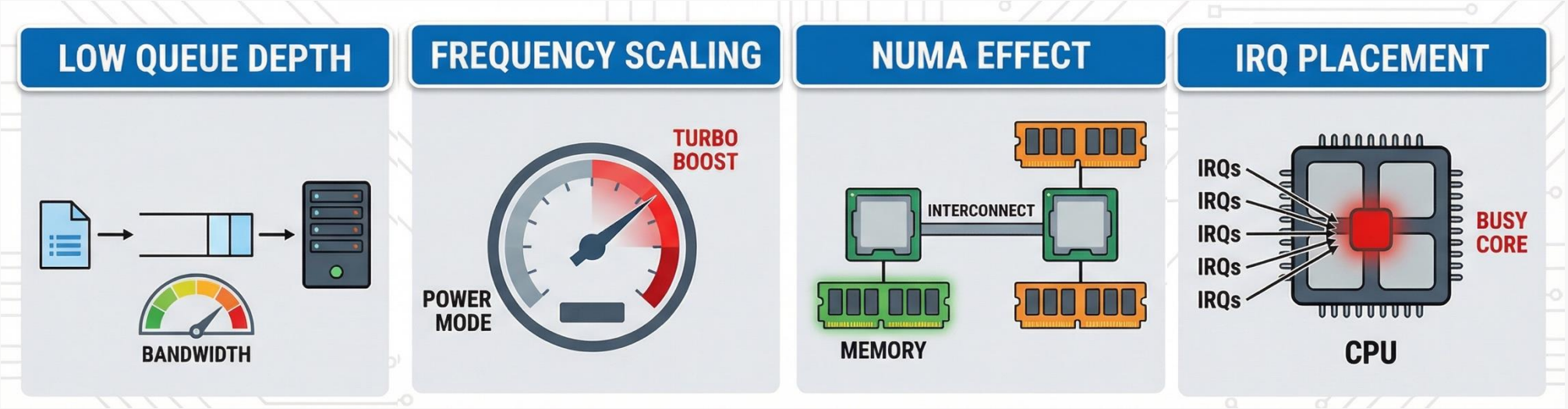
NOT STEADY-STATE

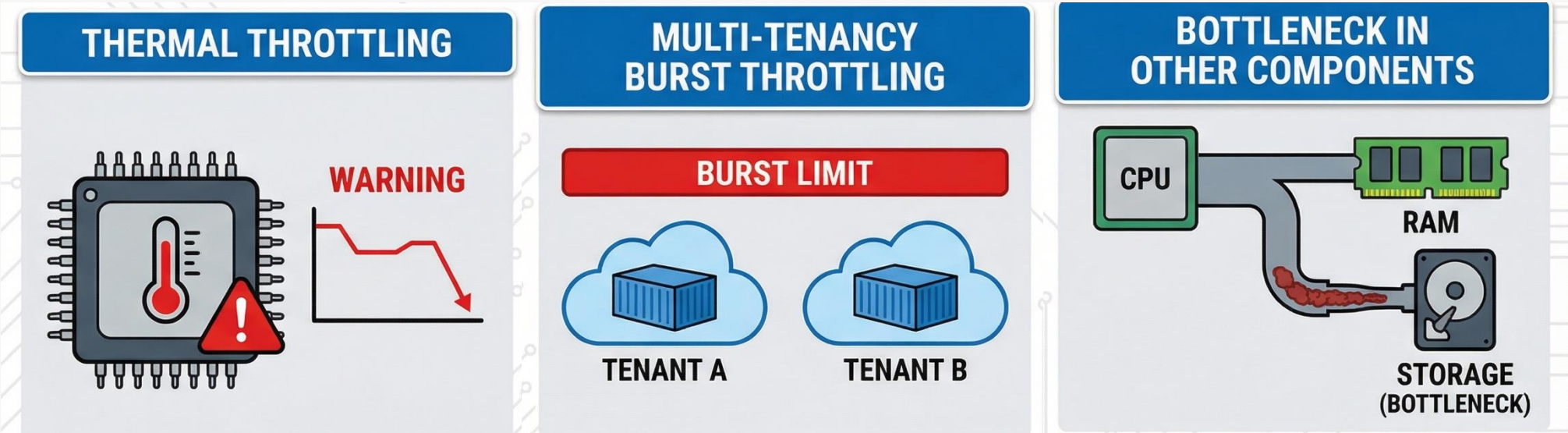
TOO SHORT

UNCONTROLLED COMPARISON



Measuring Storage Systems: System Setup Pitfalls





Always Measure One-Level Deeper

- Key to good performance measurement
  - make many more measurements besides the ones you think are important
  - not only understand what, but also why
- Performance measurement done well lead to
  - new discovery
  - new intuition about system behavior

Always measure one-level deeper!

https://cacm.acm.org/research/always-measure-one-level-deeper

The Layers of Measurements

Level	Question	Tool	Metric
0 (User)	"Is it slow?"	fio, dd	IOPS, MB/s

The Layers of Measurements

Level	Question	Tool	Metric
0 (User)	"Is it slow?"	fio, dd	IOPS, MB/s
1 (Kernel)	"Is it the Disk or the OS?"	iostat, sar	%iowait, avgqu-sz
2 (Trace)	"Which function is slow?"	blktrace, eBPF	Q2D (queue-to-dispatch) vs D2C (dispatch-to-complete)
3 (Code)	"Why is it sleeping?"	offcputime, perf	lock contention, context switches
4 (Phys)	"Is the NAND busy?"	nvme tool	GC Cycles, Die Collision

Performance Optimization

- Amdahl's law
- Application-level optimizations
- OS-level optimizations
- Leveraging hardware

Obviously, caching is the important one (but we do not talk about them in today's class)

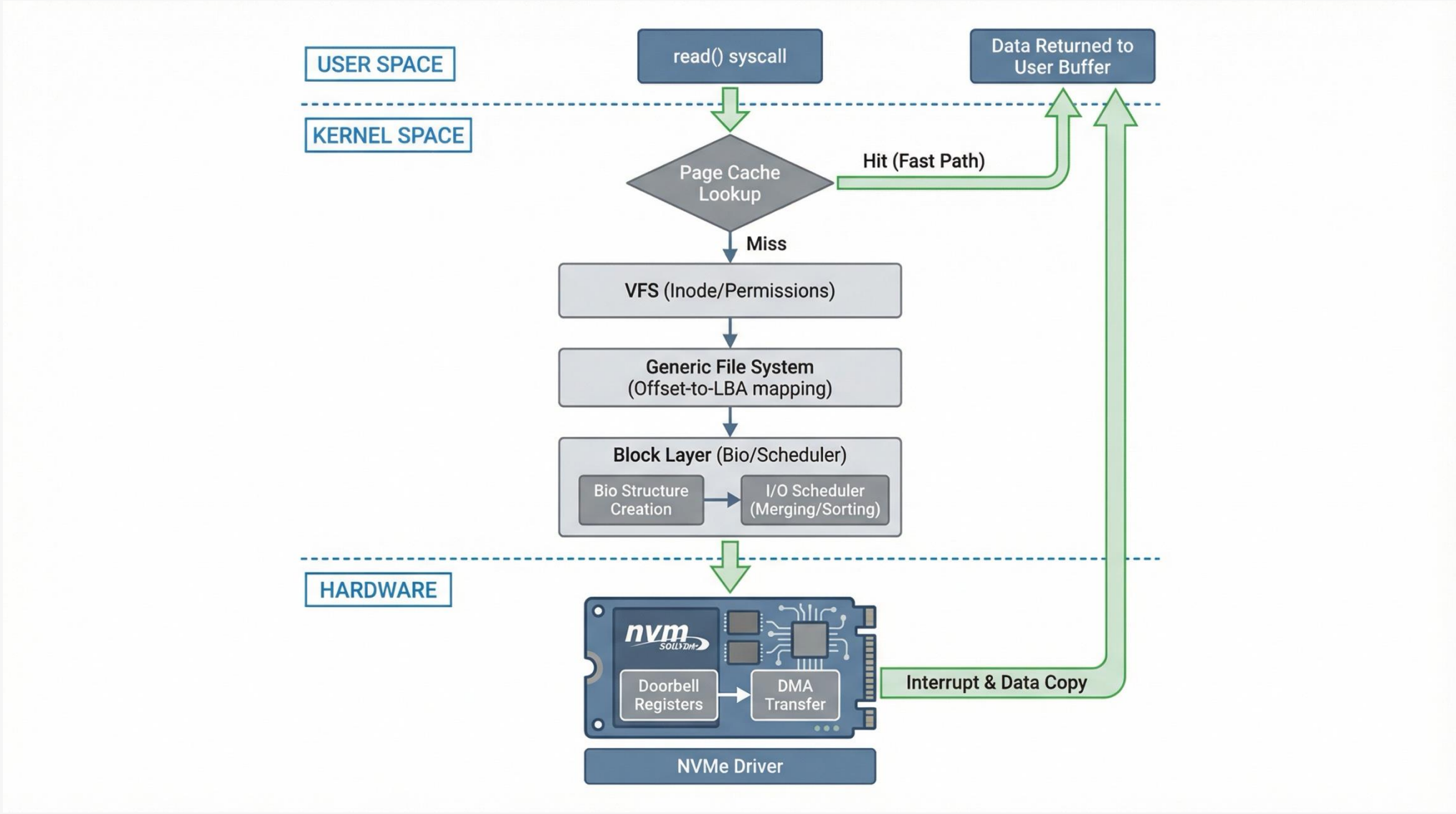
Amdahl's Law

- How much can we improve a system's performance if we improve the performance of one component?
  - $S(s) = \frac{1}{(1-p)+\frac{p}{s}}$
  - $\boldsymbol{S}$ is the theoretical speedup of the whole system
  - $\boldsymbol{s}$ is the speedup of the component
  - $\boldsymbol{p}$ is the original execution time of the component (in fraction)
- Key property
  - diminishing return with *s*
  - performance decided by 1-*p*

Why Not Keep Improving Hardware?

- Amdahl’s Law applied to Storage
  - HDD Era
    - disk seek = 10 ms, kernel overhead = 5 µs
    - strategy: optimize the disk (seek reordering, elevators)
  - NVMe Era
    - Disk read = 10 µs. Kernel overhead = 5 µs
    - OS software stack is now 33%–50% of the end-to-end latency
    - strategy: optimize the software stack

Where Are the Overheads?



Application-level Optimizations: Reduce Kernel Overheads

- Zero-copy I/O
  - `mmap()`: remove copy data from kernel to user-space
  - `sendfile()` : transfer data from disk to network card
  - similarly, splice() move data from fd to fd
- Asynchronous I/O
  - blocking I/O: slow and context switch
  - io_uring
    - a shared ring buffer between user and kernel
    - submission queue and completion queue
    - zero syscall

Application-level Optimizations: Shaping Workload

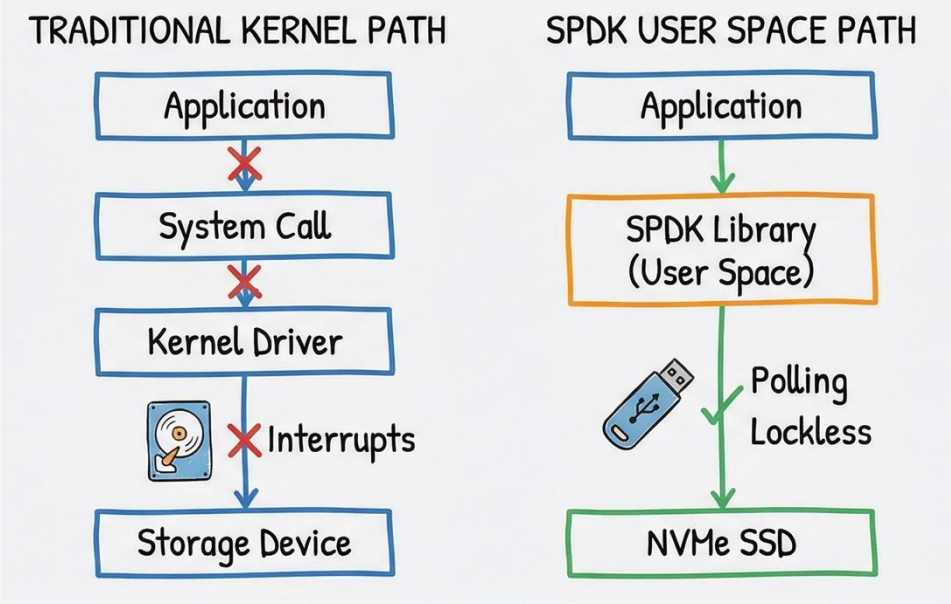
- Random-to-Sequential Transformation
  - SSDs hate random writes (write amplification)
  - HDDs hate random read/writes (seek time)
  - solution: Log-Structured Merge Trees (LSM-Trees), e.g., RocksDB, LevelDB
    - buffer writes in RAM (MemTable)
    - flush to disk as a sorted, immutable file (SSTable) when full
  - cost: read amplification (check multiple SSTables to find a key)
- Group commit
  - fsync() forces a cache flush to disk, doing frequently kills performance
  - solution: batching
  - trade-off: latency vs. throughput

Application-level Optimizations: Others

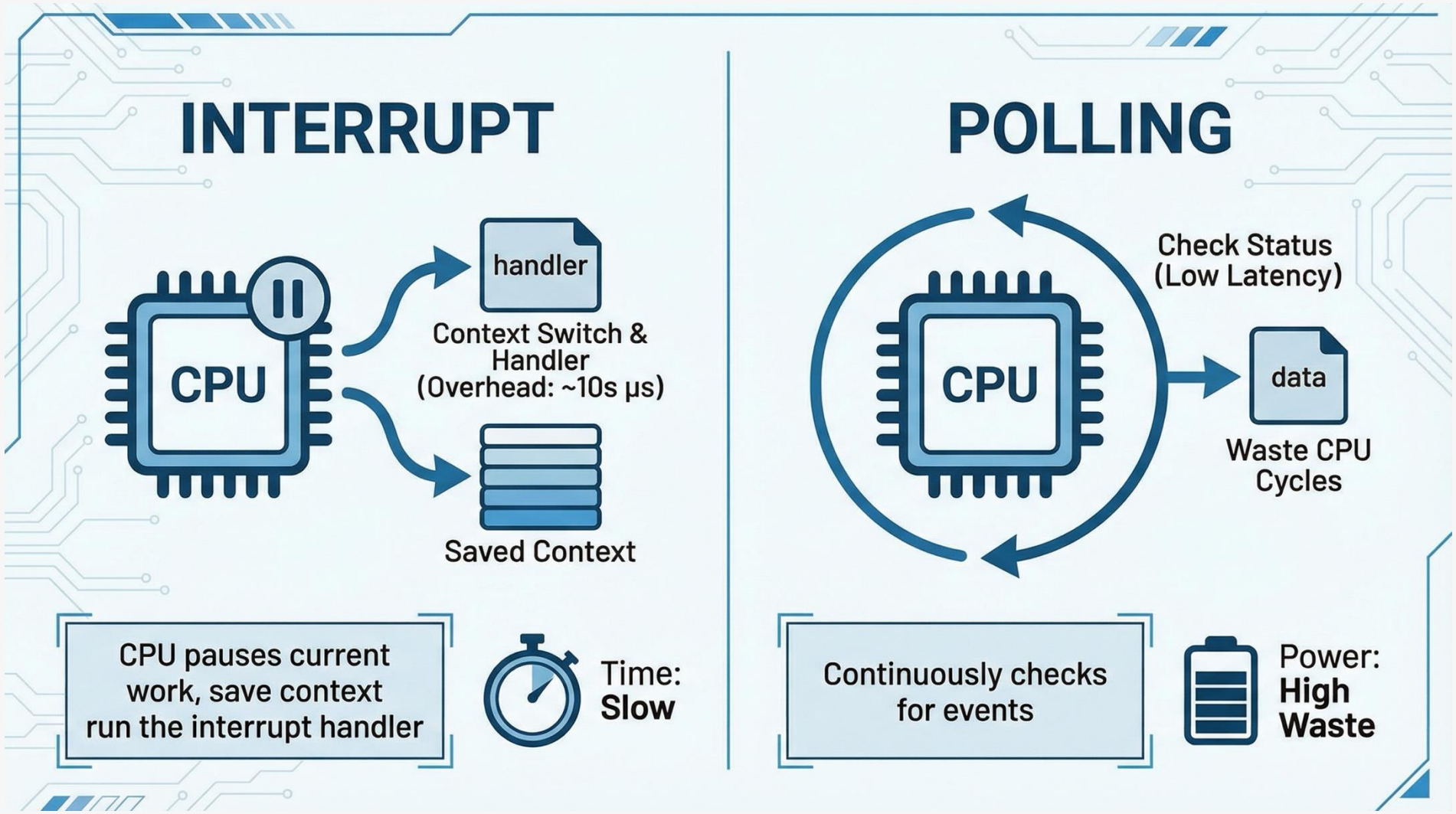
- Change file system parameters, mount options
  - only journal metadata
  - noatime
- Change block layer scheduler
  - none for SSD
- Ensure proper alignment between layers

OS-level Optimizations: Bypass Kernel

- Storage Performance Development Kit (SPDK)
  - move the NVMe driver into User Space
  - use polling instead of interrupts with per-core threading
  - trade-off: no POSIX compliance
- Traditional stack
  - App → VFS → page cache → FS → block layer → driver → device
- SPDK fast path
  - App → SPDK bdev → user-space driver → device
-

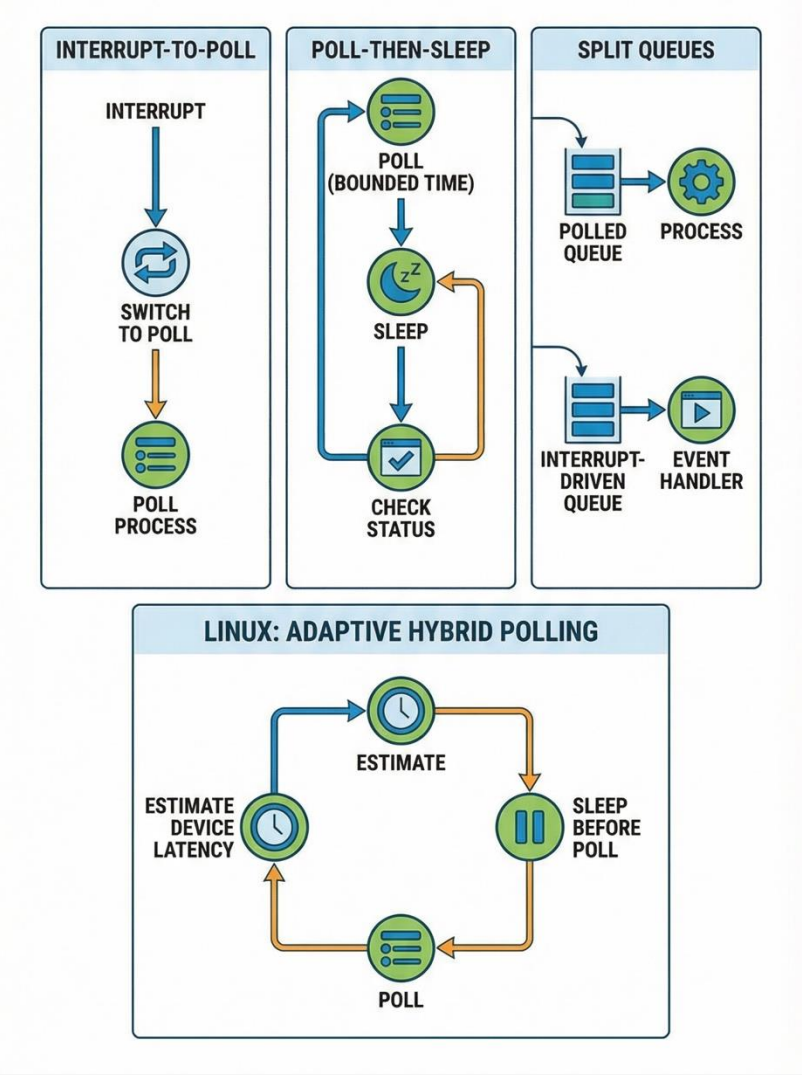


OS-level Optimizations: Hybrid Polling



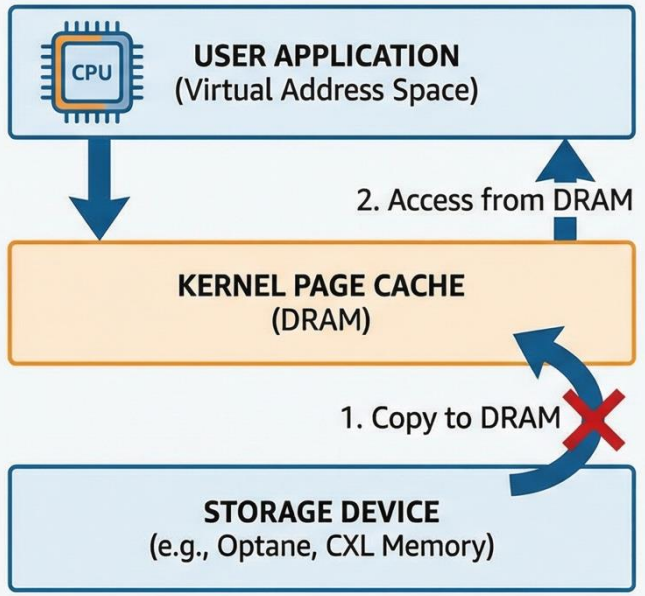
OS-level Optimizations: Hybrid Polling

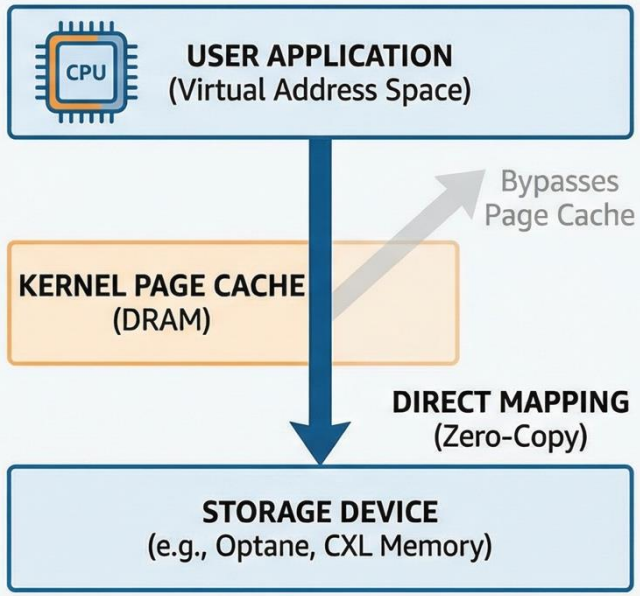
- Hybrid polling
  - common pattern
    - interrupt-to-poll: switch to poll after an interrupt
    - poll-then-sleep: poll for bounded time
    - split queues: some polled, some interrupt-driven
  - Linux: adaptive hybrid polling
    - estimate device latency, sleep before poll



OS-level Optimizations: DAX Mode

- DAX: Direct Access
- Used for fast memory/storage
  - Intel Optane: low latency 100s ns
  - problem: double copy (copy from storage to DRAM)
  - solution: bypass page cache, directly access device using load/store





Leveraging Hardware: Near-Data Processing

- Data movement tax
  - latency and bandwidth
  - energy consumption: moving data consumes orders of magnitude more energy than compute
- Move compute to data
- Near-Data Processing (NDP)
  - processing-in-memory (PIM)
    - integrate simple logic unit (ALU) into memory dies
  - computational storage drive (CSD)
    - put ARM core or FPGA inside storage, e.g., SSD

Leveraging Hardware: Near-Data Processing

- Key techniques
  - predicate push-down
    - `select * from logs where level='ERROR'`
  - near-data aggregation
    - sum, max
  - offloading
    - encryption, compression

Leveraging Hardware: Near-Data Processing

- Why not used everywhere?
  - programming model
  - coherency: if SSD modifies data, how does CPU know
  - security/isolation: multi-tenant concerns
  - debuggability
  - hardware constraint: you cannot run an LLM :(

Summary

- Performance measurement
  - metrics
  - Little’s law
  - open vs closed systems
  - the hidden pitfalls
- Performance optimization
  - Amdahl’s law
  - application
  - OS
  - hardware

Next time

- Software cache