Name: Dang Khoa Le
Student ID: 103844421

Week 1 Report – Introduction

1. Lecture Reflection

1.1 Understanding Concurrent Programming

Concurrent programming refers to the design and implementation of programs that allow multiple tasks to be in progress at the same time from the observer's perspective. It addresses:

- **Time-sharing:** Efficient CPU allocation to multiple tasks.
- **Concurrency control:** Managing race conditions and deadlocks.
- **Fairness:** Ensuring equitable CPU time for tasks.

From a **conceptual standpoint**:

- **Concurrent computing** focuses on the illusion of simultaneity.
- **Parallel computing** focuses on actual simultaneous execution using multiple processors/cores.

1.2 Computing Paradigms Overview

Key Paradigms:

1. **Concurrent Computing:** Multiple tasks logically execute at once; common in OS multitasking.
2. **Parallel Computing:** Tasks physically run simultaneously (e.g., multi-core CPU, GPU computing).
3. **Distributed Computing:** Multiple autonomous computers, each with private memory, communicate via message passing.
4. **Cluster Computing:** Networked, often homogeneous machines working as a single system.
5. **Grid Computing:** Loosely coupled, geographically dispersed, heterogeneous systems.
6. **Cloud Computing:** Service-oriented computing (SaaS, PaaS, IaaS), hiding computational complexity.
7. **Fog/Edge Computing:** Processing occurs closer to the data source, minimising latency.

1.3 Real-world Examples

- **Concurrent:** Java servlet container managing 500 users on a single-core machine via context switching.
- **Parallel:** Scientific simulations running on a 64-core HPC node.
- **Distributed:** Hadoop MapReduce cluster.
- **Cloud:** AWS Lambda serverless functions.
- **Edge:** AI inference on IoT camera without sending video to cloud.

2. Lab Reflection

2.1 Part I – Paradigm Summary

Concurrency should be considered a software-level design philosophy, enabling responsiveness and resource efficiency regardless of hardware capabilities. Parallelism is a hardware-level capability that enables true simultaneous execution. Distributed computing extends beyond one physical machine, introducing network-related challenges such as latency and consistency.

2.2 Concurrent Server Simulation

For lab experiment, I additionally implemented a Java-based simulation of a concurrent web server to demonstrate handling multiple clients simultaneously. Key implementation steps:

1. Thread Pool Creation: Used an ExecutorService with a fixed pool of 4 worker threads.
2. Task Submission: Simulated 10 client requests, submitting each as a task to the pool.

3. Concurrent Handling: Each task ran the handleRequest method, which simulated variable processing time.

This approach efficiently manages resources and ensures server responsiveness by allowing threads to process requests concurrently without blocking each other during I/O operations.

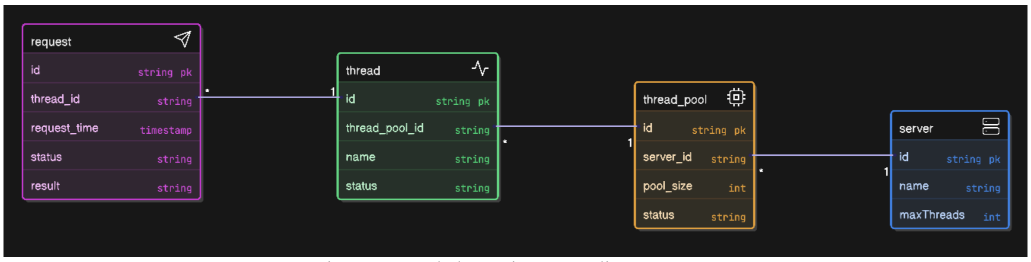


Figure. Extended Practice UML diagram

3. Diagram – Concept Map

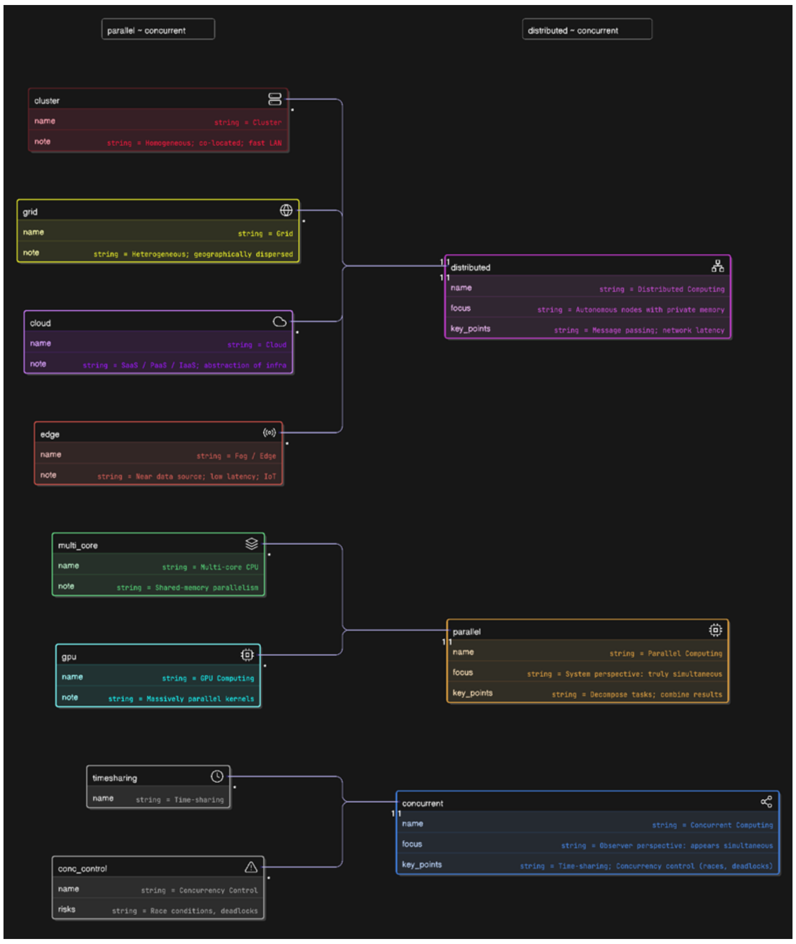

parallel ~ concurrent
distributed ~ concurrent
cluster
name string = Cluster
note string = Homogeneous; co-located; fast LAN
grid
name string = Grid
note string = Heterogeneous; geographically dispersed
cloud
name string = Cloud
note string = SaaS / PaaS / IaaS; abstraction of infra
edge
name string = Fog / Edge
note string = Near data source; low latency; IoT
distributed
name string = Distributed Computing
focus string = Autonomous nodes with private memory
key_points string = Message passing; network latency
multi_core
name string = Multi-core CPU
note string = Shared-memory parallelism
gpu
name string = GPU Computing
note string = Massively parallel kernels
parallel
name string = Parallel Computing
focus string = System perspective: truly simultaneous
key_points string = Decompose tasks; combine results
timesharing
name string = Time-sharing
conc_control
name string = Concurrency Control
risks string = Race conditions, deadlocks
concurrent
name string = Concurrent Computing
focus string = Observer perspective: appears simultaneous
key_points string = Time-sharing; Concurrency control (races, deadlocks)

Figure. Concept UML diagrams

4. Conclusion

Concurrent programming is a core design principle for modern systems. While parallelism relies on multi-core hardware for speed, concurrency ensures responsive and scalable applications, even on limited resources.

The key is understanding that concurrency (managing multiple tasks) is different from parallelism (executing them simultaneously). This distinction is crucial for designing efficient systems, from edge computing to distributed architectures, where effective concurrency management is essential for performance and user experience.

Week 2 Report – Process

1. Lecture Reflection

1.1 Program vs. Process

- **Program (static):**

A set of binary instructions stored on disk.

- **Process (dynamic):**

A program in execution, with its own allocated memory, CPU state, and OS resources.

1.2 Why Processes Exist

Before processes were formalized:

- Early computers ran **one command at a time** (extremely inefficient).
- Batch processing allowed multiple programs in sequence, but **CPU idle** during I/O.
- Processes were introduced to:
  - Encapsulate a program's state.
  - Allow **time-sharing** between CPU-bound and I/O-bound tasks.
  - Provide **isolation** and **state restoration** for multitasking.

1.3 Anatomy of a Process

A process consists of:

- **Program code** (text segment).
- **Data** (static variables, heap).
- **Stack** (function calls, local variables).
- **CPU registers** and **program counter**.
- **OS-level state** (open file descriptors, scheduling priority, etc.).

1.4 Process States & Transitions

Common states:

Created → **Ready** → **Running** →**Blocked** (waiting I/O or event) → **Terminated**

Diagram – Process State Transitions:

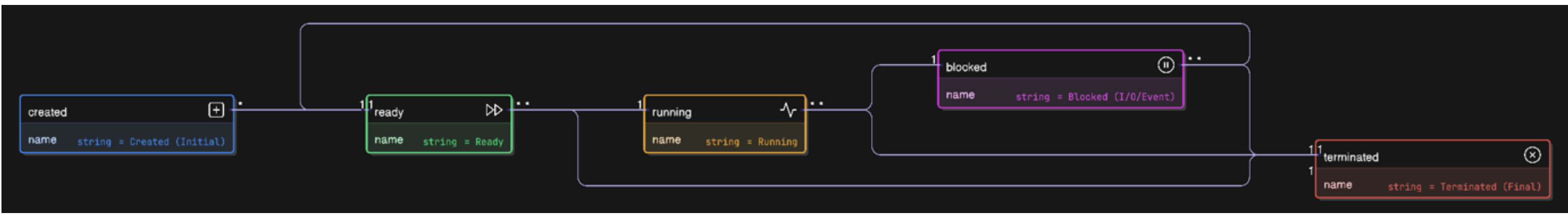

created
name string = Created (Initial)
ready
name string = Ready
running
name string = Running
blocked
name string = Blocked (I/O/Event)
terminated
name string = Terminated (Final)

Figure. Process State Transitions UML diagrams

1.5 Context Switching

When switching from one process to another, the OS:

1. Saves CPU registers, program counter, and stack pointer of the current process.
2. Loads these values for the next process.
3. Updates scheduler's process queues.

Context switching introduces overhead and impacts throughput.

1.6 Key Process APIs

- **fork()** – Create a child process by duplicating the parent.
- **wait()** – Suspend the parent until the child terminates.
- **exec()** – Replace the process image with a new program.

2. Lab Reflection

Summary

A process is the unit of execution managed by the OS. Understanding lifecycle is critical for:

- Building multitasking systems.
- Handling I/O without blocking the entire program.
- Managing CPU utilization effectively.

3. Diagram – Fork–Exec–Wait Sequence

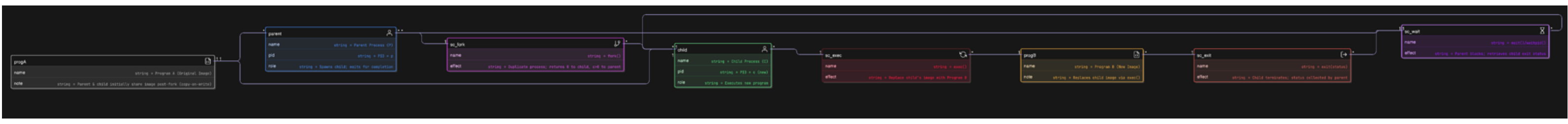

progA
name string = Program A (Original Image)
note string = Parent & child initially share image post-fork (copy-on-write)
parent
name string = Parent Process (P)
pid string = PID = p
role string = Spawns child; waits for completion
sc_fork
name string = fork()
effect string = Duplicate process; returns 0 to child, c>0 to parent
child
name string = Child Process (C)
pid string = PID = c (new)
role string = Executes new program
sc_exec
name string = exec()
effect string = Replace child's image with Program B
progB
name string = Program B (New Image)
note string = Replaces child image via exec()
sc_exit
name string = exit(status)
effect string = Child terminates; status collected by parent
sc_wait
name string = wait()/waitpid()
effect string = Parent blocks; retrieves child exit status

Figure. Fork–Exec–Wait Sequence UML diagrams

4. Conclusion

Processes are the fundamental execution abstraction in operating systems. They provide critical isolation for system stability and security by separating program memory and resources.

The transition from low-level C (e.g., fork-exec-wait) to Java's ProcessBuilder demonstrates that while higher-level languages abstract direct CPU control, they still rely on underlying OS process primitives.

This process model is vital in server environments for:

- **Lifecycle Management:** Parent processes control child processes.
- **Isolation:** Child processes execute tasks securely.
- **Efficiency:** Controlled cleanup (wait()) prevents resource leaks.

Understanding processes is essential groundwork for advanced topics like threads, scheduling, and inter-process communication.

Week 3 Report – Scheduling

1. Lecture Reflection

1.1 Scheduling Fundamentals

The scheduler decides which process in the ready queue will run next on the CPU. Key evaluation criteria:

- **CPU Utilization:** Keep CPU as busy as possible.
- **Throughput:** Number of processes completed per time unit.
- **Turnaround Time:** Time from submission to completion.
- **Response Time:** Time to produce first output (important for interactivity).
- **Fairness:** All processes should get equitable CPU share.
- **Deadline adherence:** For real-time systems.

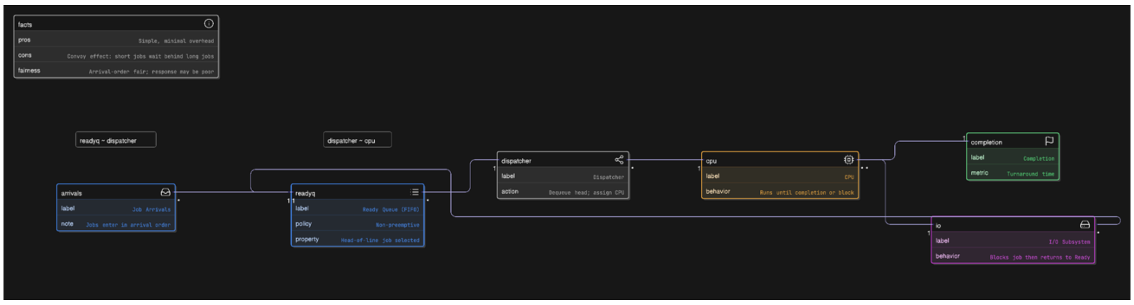
Schedulers can be:

- **Preemptive:** Can interrupt a running process to give CPU to another (better responsiveness).
- **Non-preemptive:** Once a process gets CPU, it keeps it until it finishes or blocks (simpler, but less responsive).

1.2 Scheduling Algorithms

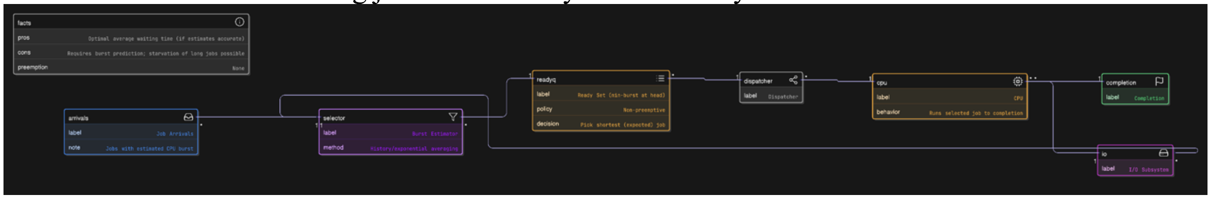
First-Come, First-Served (FCFS/FIFO)

- **Non-preemptive**; processes run in arrival order.
- Simple but **can cause convoy effect** (short jobs wait behind long ones).
- Example use: Batch systems.


facts
pros Simple, minimal overhead
cons Convoy effect: short jobs wait behind long jobs
fairness Arrival-order fair; response may be poor
readyq ~ dispatcher
dispatcher ~ cpu
arrivals
label Job Arrivals
note Jobs enter in arrival order
readyq
label Ready Queue (FIFO)
policy Non-preemptive
property Head-of-line job selected
dispatcher
label Dispatcher
action Dequeue head; assign CPU
cpu
label CPU
behavior Runs until completion or block
completion
label Completion
metric Turnaround time
io
label I/O Subsystem
behavior Blocks job then returns to Ready

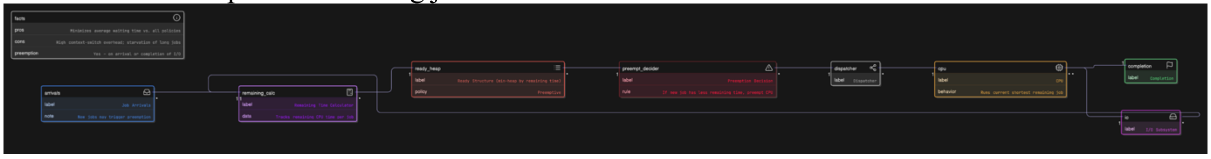
Shortest Job First (SJF)

- **Non-preemptive**; run the job with the shortest estimated burst time.
- Optimal for average waiting time if burst lengths are known.
- **Problem:** Requires accurate burst time estimation.
- **Starvation risk:** Long jobs can be delayed indefinitely.


facts
pros Optimal average waiting time (if estimates accurate)
cons Requires burst prediction; starvation of long jobs possible
preemption None
arrivals
label Job Arrivals
note Jobs with estimated CPU burst
selector
label Burst Estimator
method History/exponential averaging
readyq
label Ready Set (min-burst at head)
policy Non-preemptive
decision Pick shortest (expected) job
dispatcher
label Dispatcher
cpu
label CPU
behavior Runs selected job to completion
completion
label Completion
io
label I/O Subsystem

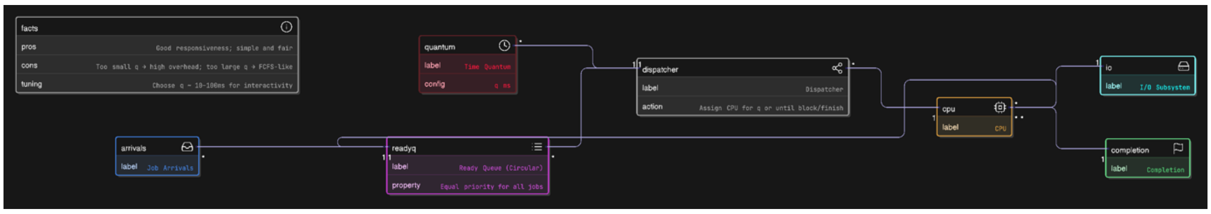
Shortest Remaining Time First (SRTF/PSJF)

- **Preemptive**; choose process with least remaining time.
- Minimizes average waiting time but increases context switching overhead.
- Starvation possible for long jobs.


facts
pros Minimizes average waiting time vs. all policies
cons High context-switch overhead; starvation of long jobs
preemption Yes — on arrival or completion of I/O
arrivals
label Job Arrivals
note New jobs may trigger preemption
remaining_calc
label Remaining Time Calculator
data Tracks remaining CPU time per job
ready_heap
label Ready Structure (min-heap by remaining time)
policy Preemptive
preempt_decider
label Preemption Decision
rule If new job has less remaining time, preempt CPU
dispatcher
label Dispatcher
cpu
label CPU
behavior Runs current shortest remaining job
completion
label Completion
io
label I/O Subsystem

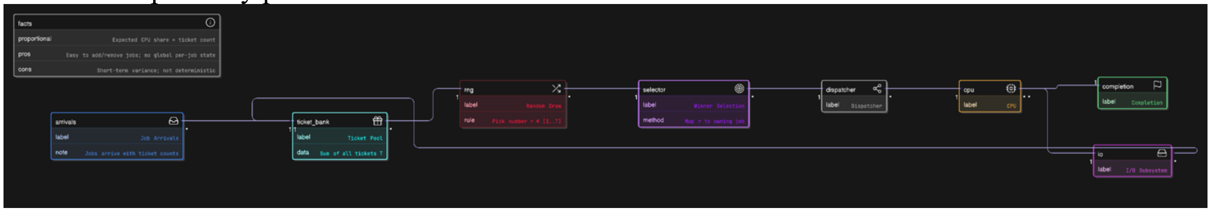
Round Robin (RR)

- **Preemptive**; time slice (quantum) is allocated to each process in cyclic order.
- **Small quantum:** Better responsiveness, but higher context-switch overhead.
- **Large quantum:** Closer to FCFS.
- Favours interactive workloads.


facts
pros Good responsiveness; simple and fair
cons Too small q → high overhead; too large q → FCFS-like
tuning Choose q ~ 10-100ms for interactivity
quantum
label Time Quantum
config q ms
arrivals
label Job Arrivals
readyq
label Ready Queue (Circular)
property Equal priority for all jobs
dispatcher
label Dispatcher
action Assign CPU for q or until block/finish
cpu
label CPU
io
label I/O Subsystem
completion
label Completion

Lottery Scheduling (Random)

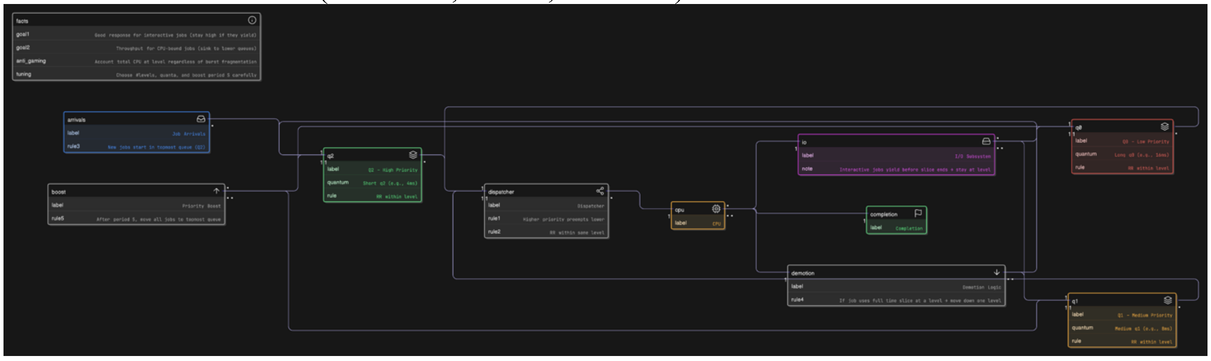
- **Proportional share**; processes get tickets, random draw decides next process.
- Simpler to add/remove jobs dynamically.
- Not perfectly precise in short-term allocations.


facts
proportional Expected CPU share ∝ ticket count
pros Easy to add/remove jobs; no global per-job state
cons Short-term variance; not deterministic
arrivals
label Job Arrivals
note Jobs arrive with ticket counts
ticket_bank
label Ticket Pool
data Sum of all tickets T
rng
label Random Draw
rule Pick number r ∈ [1..T]
selector
label Winner Selection
method Map r to owning job
dispatcher
label Dispatcher
cpu
label CPU
completion
label Completion
io
label I/O Subsystem

Multi-Level Feedback Queue (MLFQ)

- Multiple queues with decreasing priority levels.
- Jobs start at highest priority; if they use full quantum, they move down.

- Interactive (I/O-bound) jobs remain high priority; CPU-bound jobs sink.
- Rule refinements: Priority boosts after set interval to prevent starvation; anti-gaming rules.
- Used in real OSes (BSD Unix, Solaris, Windows).


facts
goal1 Good response for interactive jobs (stay high if they yield)
goal2 Throughput for CPU-bound jobs (sink to lower queues)
anti_gaming Account total CPU at level regardless of burst fragmentation
tuning Choose #levels, quanta, and boost period S carefully
arrivals
label Job Arrivals
rule3 New jobs start in topmost queue (Q2)
boost
label Priority Boost
rule5 After period S, move all jobs to topmost queue
q2
label Q2 - High Priority
quantum Short q2 (e.g., 4ms)
rule RR within level
dispatcher
label Dispatcher
rule1 Higher priority preempts lower
rule2 RR within same level
cpu
label CPU
io
label I/O Subsystem
note Interactive jobs yield before slice ends → stay at level
completion
label Completion
demotion
label Demotion Logic
rule4 If job uses full time slice at a level → move down one level
q0
label Q0 - Low Priority
quantum Long q0 (e.g., 16ms)
rule RR within level
q1
label Q1 - Medium Priority
quantum Medium q1 (e.g., 8ms)
rule RR within level

2. Lab Reflection

2.1 Part I – Summary of Scheduling Models

From the lecture and lab tasks, I compiled the following reflection table:

Algorithm	Type	Strengths	Weaknesses	Starvation Risk
FCFS/FIFO	Non-preemptive	Simple, fair in arrival order	Poor response for short jobs after long ones	Low
SJF	Non-preemptive	Optimal waiting time	Needs burst estimation, starvation possible	Yes
SRTF/PSJF	Preemptive	Optimal average wait	High overhead, starvation	Yes
RR	Preemptive	Good responsiveness	Higher overhead	No
Lottery	Preemptive	Flexible, easy job add/remove	Not deterministic short-term	Low
MLFQ	Preemptive	Balances interactivity & throughput	Complex tuning, starvation possible without boosts	Yes (mitigated by boosts)

2.2 Part II – Short Answer Responses

Main tasks of the scheduler: Decide process execution order, ensure fairness, optimize CPU utilization, and meet system performance goals.

Preemptive vs. Non-preemptive: Preemptive allows interrupting processes; non-preemptive runs processes to completion or block.

Shorter quantum advantages: Better responsiveness for interactive processes.

Longer quantum advantages: Less context-switch overhead, better throughput for CPU-bound workloads.

Feedback scheduling usefulness: Helps interactive jobs by adapting their priority dynamically.

Starvation:
SJF: Yes | **RR:** No | **FCFS:** No | **Fixed Priority:** Yes | **MLFQ:** Yes, unless priority boost

3. Conclusion

Week 3 clarified that **scheduling is a balancing act** between responsiveness, throughput, and fairness. While FCFS is predictable and fair by arrival order, it fails in interactive contexts. SJF and SRTF deliver optimal waiting times but require accurate runtime predictions, which are rarely available in real systems and can cause starvation.

Round Robin's simplicity and fairness make it ideal for interactive workloads, but quantum size tuning is critical — too short increases overhead, too long mimics FCFS's drawbacks. MLFQ emerges as a practical hybrid, combining adaptive priority management with fairness controls like periodic boosts, although it adds complexity in tuning parameters.

In practical system design, the scheduler choice must match **workload characteristics**: compute-heavy HPC clusters might prefer batch-friendly policies like SJF variants, while multi-user interactive systems benefit from RR or MLFQ.

Week 4 Report – Threads

1. Lecture Reflection

1.1 Purpose (motivation & context)

Early operating systems introduced **processes** to encapsulate a running program with its own memory and saved state, creating the illusion of multitasking on a single CPU via context switching. However, a single process that performs multiple responsibilities (e.g., ingesting data, updating the UI, handling operator input) can still feel unresponsive if those responsibilities run sequentially. The lecture motivates **threads** with a monitoring-system scenario: while a long I/O task is running, user input should be handled immediately; separating subtasks into threads allows pausing the I/O thread, servicing the UI thread, then resuming the I/O thread—all within the same process address space. This avoids the data duplication, consistency, and communication overheads that would arise if we split the subtasks into separate processes.

1.2 Definition (concept & state)

A **thread** is the unit of execution (control flow) inside a process. Practically it consists of a processor context and a stack; it is scheduled by the OS and executes within the **same address space** as other threads of the process. Each thread has private state (program counter, stack pointer, registers, stack, scheduling properties), while threads **share** process-level state such as the address space (code, statics, heap), open files, sockets, and locks. Compared with processes, threads make the **process itself** appear multi-tasking and enable cheaper communication through shared memory.

Why threads: overlap I/O & compute, exploit multi-CPUs, communicate cheaply via shared memory.

1.3 How threads work (models & APIs)

Programs can be decomposed into routines that execute in **true** or **pseudo** parallelism, using patterns like **manager/worker** (a manager thread handles I/O and delegates tasks to workers) and **pipeline** (producer-consumer stages). Classic POSIX APIs show the lifecycle: pthread_create() to start a thread, pthread_join() to synchronize and retrieve status, and pthread_exit() to terminate; yielding lets a thread re-enter the run queue.

Common patterns: manager/worker and pipeline (producer–consumer stages). POSIX lifecycle: pthread_create() → run; pthread_join() → synchronize/collect status; pthread_exit() → terminate; yielding returns a thread to the run queue.

1.4 Threads in Java (creation & caveats)

Two patterns:

- Extend Thread and override run(), call start().
- Implement Runnable, pass to Thread, call start() (usually preferable). Useful ops: start(), join(), wait()/notify(), setPriority(); legacy stop/suspend/resume are deprecated.

Thread-safety notes: many classes in java.util are not thread-safe (e.g., ArrayList), GUIs are not thread-safe, while some classes like Random and certain I/O (e.g., System.out) are. Also, even a simple integer increment is **not atomic** in Java, motivating synchronization or atomic classes for shared counters.

1.5 Thread pools & executors (scalability & control)

Creating a large number of short-lived threads is inefficient; **thread pools** keep a fixed set of worker threads alive and feed them tasks from a **queue**. The lecture walks through the pool's behavior: workers wait when the queue is empty; when a task arrives, a notify wakes workers; tasks are executed; completed workers pick up remaining tasks; and this repeats. In Java, ExecutorService offers execute(), shutdown(), and awaitTermination() for lifecycle management. Proper pool sizing is a tuning problem; both too large and too small can cause forms of starvation, and systems should also handle **overload** (e.g., bounded queues or shedding load).

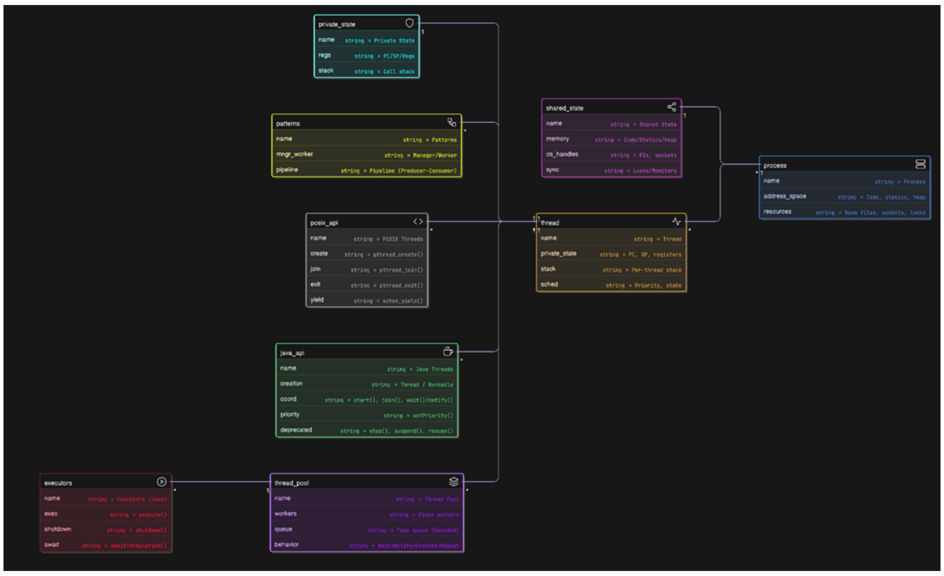
1.6 Pros, cons, and debugging realities

Pros: overlap I/O & compute, cheaper context switches than processes, good for multi-CPU. **Cons:** subtle inter-thread bugs, higher complexity, compatibility concerns → requires careful design, synchronization, and testing.

Key takeaways

- Threads provide intra-process concurrency with shared memory and per-thread stacks/contexts for responsiveness and parallel speed-ups.
- Java offers simple creation patterns and modern executors; correctness depends on thread-safe structures and coordination (join, wait/notify, atomics).
- Production systems must tune pools and handle overload to prevent starvation and thrash.

2. Diagram - Threads



private_state
name string = Private State
regs string = PC/SP/Regs
stack string = Call stack
patterns
name string = Patterns
mngr_worker string = Manager/Worker
pipeline string = Pipeline (Producer-Consumer)
shared_state
name string = Shared State
memory string = Code/Statics/Heap
os_handles string = FDs, sockets
sync string = Locks/Monitors
process
name string = Process
address_space string = Code, statics, heap
resources string = Open files, sockets, locks
posix_api
name string = POSIX Threads
create string = pthread_create()
join string = pthread_join()
exit string = pthread_exit()
yield string = sched_yield()
thread
name string = Thread
private_state string = PC, SP, registers
stack string = Per-thread stack
sched string = Priority, state
java_api
name string = Java Threads
creation string = Thread / Runnable
coord string = start(), join(), wait()/notify()
priority string = setPriority()
deprecated string = stop(), suspend(), resume()
executors
name string = Executors (Java)
exec string = execute()
shutdown string = shutdown()
await string = awaitTermination()
thread_pool
name string = Thread Pool
workers string = Fixed workers
queue string = Task queue (bounded)
behavior string = Wait→Notify→Execute→Repeat

Figure. Threads Overview - Concepts & APIs UML diagrams

- A **Process** has its **address space** (code, statics, heap) and **resources** (files, sockets, locks).
- Each **Thread** runs inside the process, with its own **private state** (PC, registers, stack, priority) but shares the **process's address space and resources**.
- **Shared state** (memory, open handles, locks) is accessible by all threads, requiring synchronization.
- **Patterns** like **Manager/Worker** and **Pipeline** represent common designs for decomposing work across threads.
- **APIs**:
  - **POSIX threads** (pthread_create, pthread_join, etc.) for system-level thread control.
  - **Java APIs** (Thread, Runnable, synchronized, ExecutorService) for higher-level concurrency.
- **Thread pools** and **executors** provide scalable management, reusing workers and handling queued tasks efficiently.

3. Real-world Practices

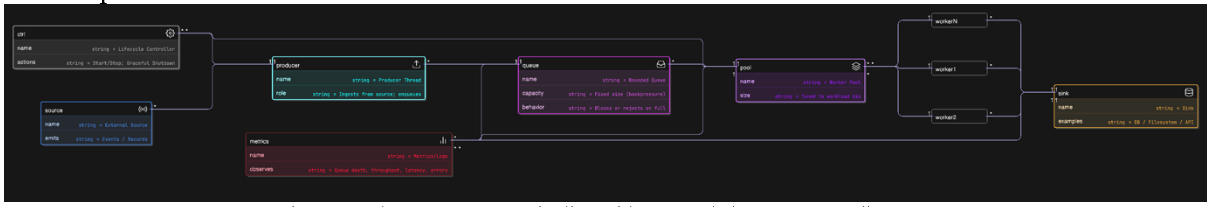
3.1 Producer–Consumer Pipeline with a Bounded Queue (Backpressure)

Implementation:

- Start a **Producer** thread that reads from an external **Source** and tries to enqueue work to a **Bounded Queue**.
- Use a **Worker Pool** (fixed size) that repeatedly dequeues items and processes them, writing results to a **Sink** (DB, FS, or another service).
- The **Bounded Queue** enforces **backpressure**: when full, the producer either **blocks** or a policy rejects/drops/backs off.
- Add a **Lifecycle Controller** for clean startup/shutdown, and **Metrics/Logs** for queue depth, throughput, per-stage latencies, and error counts.

Purpose:

- Maintain **stability** during spikes by preventing unbounded memory growth, and smooth load via controlled ingestion and processing.
- Clear **separation of concerns**: ingestion vs. processing, easier scaling and tuning of the worker pool.



ctrl
name string = Lifecycle Controller
actions string = Start/Stop; Graceful Shutdown
source
name string = External Source
emits string = Events / Records
producer
name string = Producer Thread
role string = Ingests from source; enqueues
metrics
name string = Metrics/Logs
observes string = Queue depth, throughput, latency, errors
queue
name string = Bounded Queue
capacity string = Fixed size (backpressure)
behavior string = Blocks or rejects on full
pool
name string = Worker Pool
size string = Tuned to workload mix
workerN
worker1
worker2
sink
name string = Sink
examples string = DB / Filesystem / API

Figure. Producer–Consumer Pipeline with a Bounded Queue UML diagrams

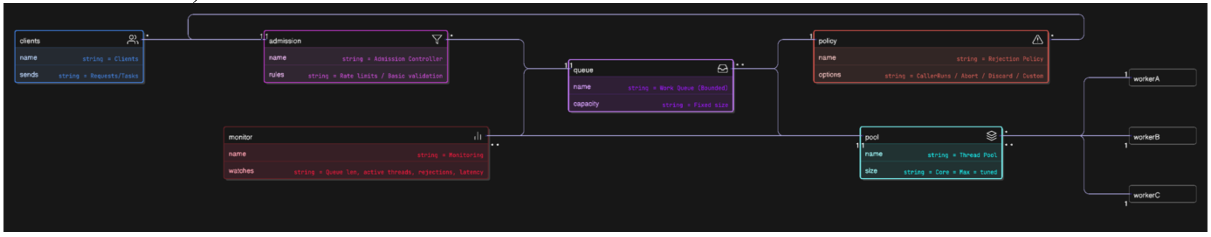
3.2 Bounded Thread Pool with a Rejection Policy (Overload Control)

Implementation:

- Route incoming tasks through **Admission** (rate-limit, shape traffic). Enqueue to a **bounded queue** feeding a fixed-size **Thread Pool**.
- When the queue is full, a **Rejection Policy** triggers: options include back-pressure to callers, immediate error/abort, or selective discard.
- **Monitoring** surfaces saturation: queue length, active threads, rejection count, latency SLOs.

Purpose:

- Preserve system **health under overload**: bounded resources, predictable backpressure, and explicit **shed-load** behavior rather than OOM crashes.
- Make capacity **transparent** so upstream systems can adapt (retry with jitter, rate limit, degrade features).



clients
name string = Clients
sends string = Requests/Tasks
admission
name string = Admission Controller
rules string = Rate limits / Basic validation
monitor
name string = Monitoring
watches string = Queue len, active threads, rejections, latency
queue
name string = Work Queue (Bounded)
capacity string = Fixed size
policy
name string = Rejection Policy
options string = CallerRuns / Abort / Discard / Custom
pool
name string = Thread Pool
size string = Core = Max = tuned
workerA
workerB
workerC

Figure. Bounded Thread Pool with a Rejection Policy UML diagrams

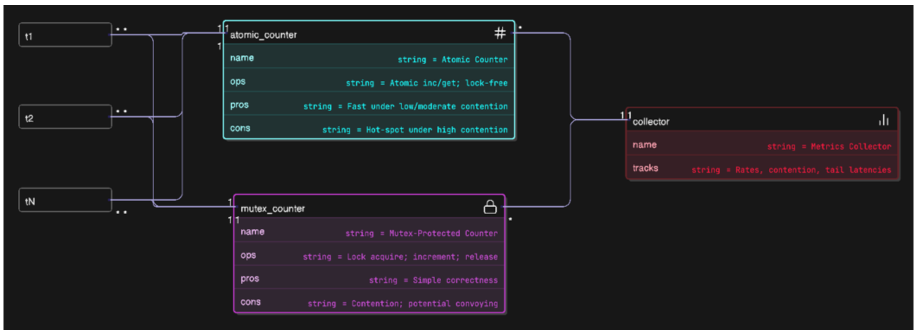
3.3 Correct Counters: Atomics vs. Mutex (Synchronized) for Shared State

Implementation:

- For a shared counter, choose **Atomic** (lock-free increments) or **Mutex** (synchronize around critical section).
- Measure contention and latency via a **Metrics Collector** to decide which strategy fits (atomics for light-to-moderate contention; mutex or **striped counters**/adders for very high contention).

Purpose:

- Ensure **correctness** (avoid lost updates) and **predictable performance**.
- Make concurrency choices **data-driven** by observing contention patterns.


t1
t2
tN
atomic_counter
name string = Atomic Counter
ops string = Atomic inc/get; lock-free
pros string = Fast under low/moderate contention
cons string = Hot-spot under high contention
mutex_counter
name string = Mutex-Protected Counter
ops string = Lock acquire; increment; release
pros string = Simple correctness
cons string = Contention; potential convoying
collector
name string = Metrics Collector
tracks string = Rates, contention, tail latencies
Figure. Correct counter UML diagrams

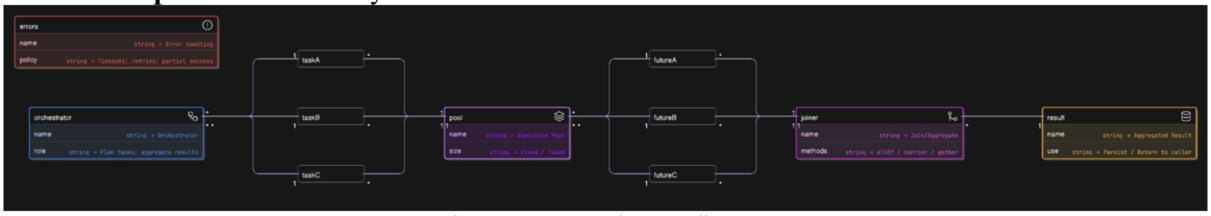
3.4 Fan-Out / Fan-In with Futures (Task Composition)

Implementation:

- The **Orchestrator** fans out multiple independent **Tasks** to a tuned **Pool**, getting **Futures** back.
- A **Joiner** waits for all (or a quorum) to complete, merges outputs, and applies **Error Handling** (timeouts/retries/partial aggregation).
- The **Result** is persisted or returned to the caller.

Purpose:

- Achieve **concurrency** across independent subtasks (I/O or CPU-bound), reduce wall-clock latency, and **compose** results cleanly.


errors
name string = Error Handling
policy string = Timeouts; retries; partial success
orchestrator
name string = Orchestrator
role string = Plan tasks; aggregate results
taskA
taskB
taskC
pool
name string = Execution Pool
size string = Fixed / Tuned
futureA
futureB
futureC
joiner
name string = Join/Aggregate
methods string = allOf / barrier / gather
result
name string = Aggregated Result
use string = Persist / Return to caller
Figure. Fan-out Fan-in UML diagrams

4. Conclusion

This week consolidates threads as the **intra-process concurrency** mechanism that balances responsiveness and throughput when used with discipline. The lectures underscore (i) what threads are and **which state is shared vs. private**, (ii) patterns such as **manager/worker** and **pipeline** to structure concurrency, and (iii) practical pooling and overload strategies in Java's ExecutorService to avoid the overhead and instability of unbounded thread creation.

From a systems perspective, the decisive factor is **control**: bound your queues, set sane pool sizes, adopt explicit rejection/backpressure behavior, and prefer atomic or synchronized primitives where correctness is paramount. These measures transform threading from a source of subtle bugs into a **predictable, scalable** tool aligned with your workload's I/O and CPU profile.

Week 5 Report – Lock I

1. Lecture Reflection

1.1 Purpose (motivation & context)

In Week 4, we learned how **threads** provide intra-process concurrency. However, when multiple threads **share variables or resources**, concurrency introduces new problems. For example, if two threads update a shared counter, the outcome depends on the timing of low-level operations (load, increment, store). This leads to **race conditions** and **indeterminate outputs**.

The motivation for **locks** is to provide **mutual exclusion (ME)**, ensuring that only one thread executes a **critical section** at a time. With locks, race conditions are eliminated, and the program produces deterministic results.

1.2 Definition (concept & key terms)

Key concurrency terms introduced:

- **Critical section:** Code block that accesses shared resources (e.g., shared variable, data structure).
- **Race condition:** 2 or more threads enter a critical section concurrently, lead to corrupted state.
- **Indeterminate program:** A program with race conditions whose output changes unpredictably across runs.
- **Mutual exclusion:** Guarantee that only one thread can enter a critical section at a time.
- **Atomicity:** An operation runs **completely** or not at all, there are no intermediate visible states.

Lock: A synchronization primitive that enforces mutual exclusion by allowing only one thread to acquire the lock and enter a critical section.

1.3 How locks work (concepts & APIs)

Locks can be understood through analogy:

- A **phone booth** can only hold one person; others wait until it is free.
- Similarly, a **lock** is acquired before a thread enters a critical section and released afterwards.

APIs:

- **POSIX (C):** pthread_mutex_lock, pthread_mutex_unlock.
- **Java:**
  - **Lock interface** (e.g., ReentrantLock): lock(), unlock(), tryLock(), lockInterruptibly().
  - **Synchronized keyword:**
    - synchronized methods | synchronized(object) blocks.
    - Automatically handles lock → try → finally → unlock.

1.4 Locks in Java (examples & mechanisms)

Two common mechanisms:

a) ReentrantLock:

- Explicit, flexible.
- Supports tryLock() with optional timeout and lockInterruptibly().

b) synchronized keyword:

- Implicitly uses the intrinsic object lock (this).
- Ensures mutual exclusion and provides a **happens-before relationship** (visibility guarantee).
- Constructors cannot be synchronized.

Comparison:

- synchronized is simpler, built into the language, but less flexible.
- Lock objects provide finer-grained control, try-lock semantics, timeouts, and interruptible acquisition.

1.5 Pros, cons, and debugging realities

Pros:

- Guarantees mutual exclusion.
- Eliminates race conditions and makes outputs deterministic.
- ReentrantLock offers more control than synchronized.

Cons:

- Introduces potential for **deadlocks** if multiple locks are acquired inconsistently.
- Can cause **performance bottlenecks** if contention is high.
- Debugging thread synchronization issues is complex and non-deterministic.

2. IncrementTest Experiment

We experimented with a simple Java class (IncrementTest) that increments:

- **localData** (local variable, not shared → always correct).
- **instanceData** (each object has its own copy → no conflicts if different objects).

- **classData** (static shared variable across instances → race conditions).

Observations from multiple runs:

- localData increments consistently as expected.
- instanceData increments correctly **per object**, but not across threads without synchronization.
- classData showed **inconsistent results**, highlighting race conditions.

Understanding gained:

- Race conditions arise because increment (x++) is **not atomic**; it decomposes into load–increment–store.
- Multiple threads interleave those steps, overwriting each other's updates.
- Locks (synchronized or ReentrantLock) are necessary to make the increment **atomic** in shared contexts.

This experiment made the theory concrete: without mutual exclusion, shared variables cannot be safely updated in a multithreaded environment.

3. Diagram – Lock with multiple threads

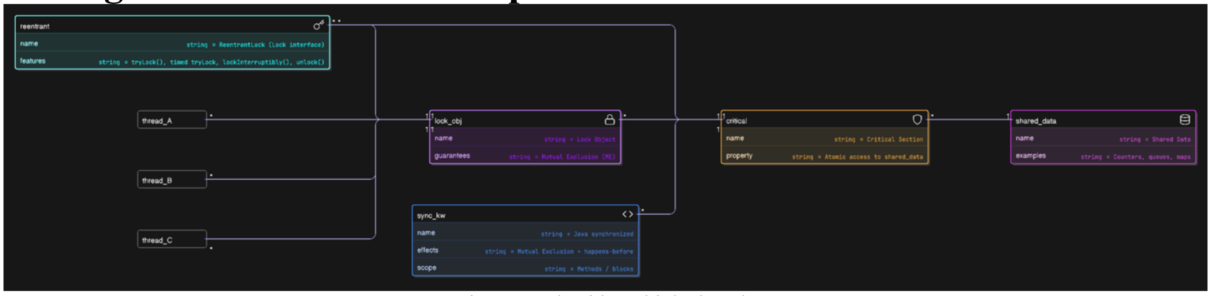

reentrant
name string = ReentrantLock (Lock interface)
features string = tryLock(), timed tryLock, lockInterruptibly(), unlock()
thread_A
thread_B
thread_C
lock_obj
name string = Lock Object
guarantees string = Mutual Exclusion (ME)
sync_kw
name string = Java synchronized
effects string = Mutual Exclusion + happens-before
scope string = Methods / blocks
critical
name string = Critical Section
property string = Atomic access to shared_data
shared_data
name string = Shared Data
examples string = Counters, queues, maps

Figure. Lock with multiple threads

Notation: Multiple threads contend for a lock to enter the critical section and touch shared data atomically. In Java, we can enforce ME with either the synchronized mechanism (which also gives a happens-before visibility guarantee) or with ReentrantLock (which adds tryLock, timed acquisition, and interruptible lock attempts).

4. Real-world Practices

4.1 tryLock() with Timeout & Interruptible Acquisition

Implementation:

- Use **non-blocking or timed acquisition** (**tryLock()**, timed **try**, or **interruptible** acquire) to **avoid indefinite blocking** on contended locks.
- Define a **policy** for what to do when the lock **isn't available**: queue/defer, return a "busy" response, or **degrade** non-critical features.
- Add a **watchdog** to detect **long waits** and trigger interrupts/retries.

Purpose: improve **responsiveness** and operational **recoverability**; prevent threads from being stuck on **intrinsic locks** (which are not interruptible), and encode **time-bounded** behavior during contention.

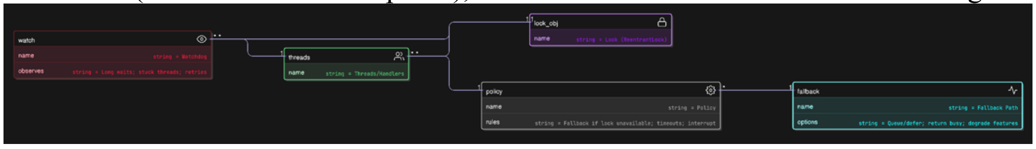

watch
name string = Watchdog
observes string = Long waits; stuck threads; retries
threads
name string = Threads/Handlers
lock_obj
name string = Lock (ReentrantLock)
policy
name string = Policy
rules string = Fallback if lock unavailable; timeouts; interrupt
fallback
name string = Fallback Path
options string = Queue/defer; return busy; degrade features

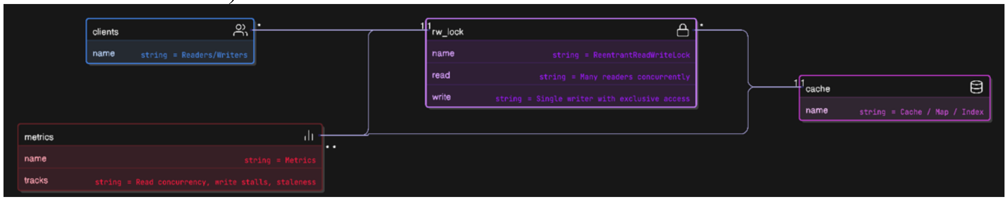
Figure. tryLock

4.2 Read–Write Lock for Read-Heavy Caches (scale readers safely)

Implementation:

- Guard a **read-heavy** structure (e.g., in-memory cache) with a **read–write lock**: many threads can hold a **read lock** concurrently; **writes** take the **write lock** exclusively.
- Keep **write** sections tight and, if needed, use **copy-on-write** or **double-buffer** strategies to shorten write holds.
- Track **stalls** and **staleness** to tune read/write ratios.

Purpose: increase **throughput** for read-dominated workloads while preserving correctness—still using locks, but allowing **concurrent readers** when safe. (Lecture notes list **ReentrantReadWriteLock** as an implementation of Lock.)

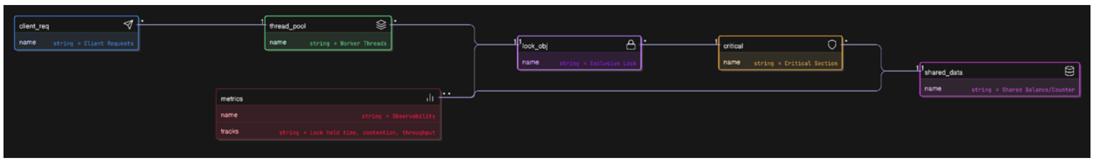

clients
name string = Readers/Writers
rw_lock
name string = ReentrantReadWriteLock
read string = Many readers concurrently
write string = Single writer with exclusive access
cache
name string = Cache / Map / Index
metrics
name string = Metrics
tracks string = Read concurrency, write stalls, staleness
Figure. Read/write Lock

4.3 Exclusive Lock for a Shared Balance/Counter (deterministic updates)

Implementation:

- Route concurrent balance updates through a **single lock** protecting the **critical section** that reads/updates **shared_data**.
- Keep critical sections **small**; measure **lock hold time** and **contention** (queueing at the lock).
- If contention is high, consider **sharding/partitioning** the data or migrating to a **read/write** pattern for read-heavy paths.

Purpose: determinism and correctness: a single thread **at a time** enters the critical section, eliminating **race conditions** on the shared value.


client_req
name string = Client Requests
thread_pool
name string = Worker Threads
lock_obj
name string = Exclusive Lock
critical
name string = Critical Section
shared_data
name string = Shared Balance/Counter
metrics
name string = Observability
tracks string = Lock hold time, contention, throughput
Figure. Exclusive Lock

5. Conclusion

Mutual exclusion is essential for correct concurrent programming, preventing race conditions by ensuring only one thread accesses a critical section at a time.

Java provides two main mechanisms:

- **synchronized**: Simple and ensures visibility.
- **ReentrantLock**: Offers advanced control (e.g., timeouts, interruptible locks).

Best practices include keeping critical sections small, monitoring contention, and using read-write locks for read-heavy workloads. These strategies make locking robust and scalable.

Week 6 Report – Lock II

1. Lecture Reflection

1.1 What's in a lock (the abstract "lock_t" struct)

A lock isn't just a boolean; it encapsulates **ownership** (which thread holds it) and usually **a wait queue** for contenders. Conceptually:

- lock_t hold metadata like "owner thread id" and **ordered queue** so future acquisition is fair(er).
- The API level (e.g., lock(), unlock()) sits on top of this metadata; **mutual exclusion** is the minimal guarantee, while **fairness and performance** depend on the chosen strategy.

1.2 When to use which lock in Java (intrinsic vs. object locks, and static data)

The lecture contrasts **intrinsic locks** (via synchronized) with **explicit locks** (ReentrantLock) and stresses scoping:

- **One critical section in one instance** → synchronized(this) is adequate.
- **Multiple independent critical sections** → use **distinct lock objects** (e.g., lock1, lock2) to avoid over-serialization.
- **Static (class) data** cannot be protected by an instance's intrinsic lock; use synchronized(ClassName.class) or a **static ReentrantLock**.

1.3 How locks are implemented (from hardware up)

The lecture surveys **why naïve approaches fail** and how real locks work:

- **Bad/insufficient ideas**
    - Disable interrupts → unsafe for apps, useless on multiprocessors, and can break I/O completion; only OS kernels use it sparingly.
    - Plain loads/stores on a "flag" → not atomic; fails mutual exclusion and **spin-waits** waste cycles.
- **Atomic hardware instructions**
    - **Test-and-Set (TAS)** → enables a working **spinlock**; correct but can be unfair and wasteful on a single CPU.
    - **Compare-and-Swap (CAS)** → more powerful; similar usage for spinning acquisition.
    - **Fetch-and-Add (FAA)** → enables **ticket locks** (first-come, first-served) that improve **fairness** (each thread gets its turn).
- **From spinning to yielding to sleeping**
    - *Spin*—ok for very short holds or when CPUs ≈ contenders; bad on single CPU with preemption.
    - *Yield*—better than blind spinning but can still burn many context switches under heavy contention.
    - **Queue + sleep** (**park()/unpark()**)—place waiting threads on a queue and **sleep** them; the unlock path wakes the **next** waiter, reducing waste and improving predictability.

1.4 Practical goals & trade-offs

A "good lock" should deliver:

- **Mutual exclusion** (correctness),
- **Fairness** (no starvation),
- **Performance** (low overhead, minimal wasted CPU, fast handover). Choosing between intrinsic locks, ReentrantLock, or more advanced policies is about **contention patterns** and **service goals** (latency vs throughput).

2. Diagram

2.1 Lock Internals & Implementation Pathways

- lock_struct in the center represents the abstract lock object: it tracks the **owner, waiting queue**, and overall goal (mutual exclusion, fairness, performance).
- **Hardware primitives** (tas, cas, faa) feed into lock strategies:
    - TAS (Test-and-Set): basis for spinlocks.
    - CAS (Compare-and-Swap): stronger primitive for atomic updates.
    - FAA (Fetch-and-Add): supports ticket locks (FIFO fairness).
- **Lock strategies:**
    - spin: busy-wait, fine for very short holds.
    - ticket: fairness via ordered service.
    - yielding: threads back off politely instead of hard spinning.
    - queued: threads parked in queue and woken one by one (efficient in high contention).
- **Usage scope in Java:**
    - inst_lock: intrinsic lock tied to an object instance (synchronized(this)).
    - split_locks: separate locks for independent state, to reduce contention.
    - class_lock: class/static-level lock, used for protecting shared static data.

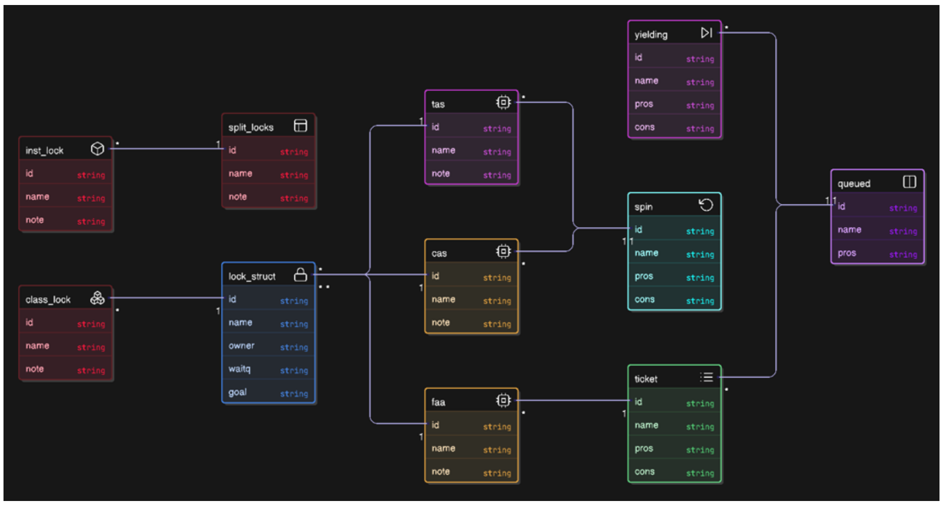


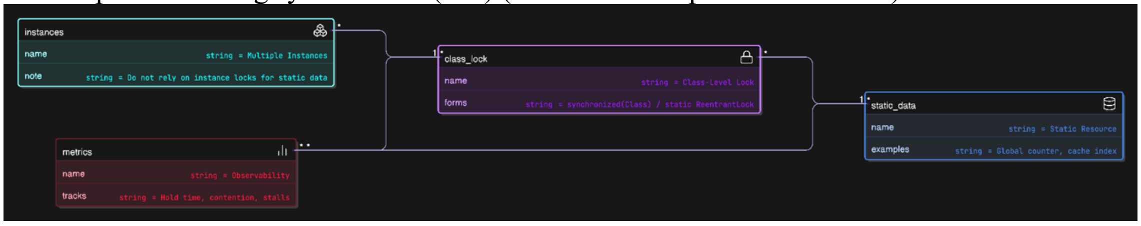
Figure. Lock internals and acquisition

2.2 Guarding static/shared state (class-level lock)

Implementation:

- Identify **class-wide** state (e.g., a static counter, cache index, or singleton stats map).
- Protect it with a **class-level lock**: either synchronized(ClassName.class) or a **static lock object**.
- Keep the critical section **tight**; measure contention and hold times; if high, **partition/stripe** the state to multiple locks.

Purpose: determinism and safety when multiple instances/threads update **the same static resource**; avoids the pitfall of using synchronized(this) (which **doesn't** protect class data).


instances
name string = Multiple Instances
note string = Do not rely on instance locks for static data
class_lock
name string = Class-Level Lock
forms string = synchronized(Class) / static ReentrantLock
static_data
name string = Static Resource
examples string = Global counter, cache index
metrics
name string = Observability
tracks string = Hold time, contention, stalls
Figure. Class-level lock

2. Reflection

This week highlighted that locks are structured objects (lock_t) managing ownership and queues, not simple flags. They enforce **mutual exclusion** but also encode policies for **fairness** and **performance**.

In Java, scope matters:

- synchronized(this) secures instance data,
- separate locks prevent false contention,
- class/static data requires **class-level locks**.

Implementation-wise, real locks build on hardware primitives (**TAS, CAS, FAA**) and evolve from **spinlocks** to **yielding** to **queue-based sleeping** for efficiency under contention. Ticket locks and queues provide fairness and predictable handoff.

The key insight is that **lock choice must match workload**: intrinsic locks suffice for light contention, while ReentrantLock or queue-based designs suit heavy or shared scenarios. A "good lock" balances correctness, fairness, and performance, aligning concurrency co