Concurrent Programming Report 1 (Week1-6)

Nur E Siam

103842784

Week 1 (Lab 1)

PART 1

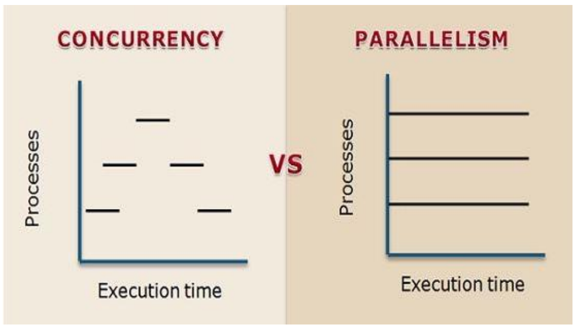


Figure 1. Concurrent Computing vs Parallel Computing

Concurrent Computing

Concurrent computing refers to the system's ability to execute multiple tasks in overlapping time periods (GeeksforGeeks, 2025). To the user, it seems as though tasks are executing at the same time, even though they may be utilising a single processor through time-slicing techniques. This principle was originally developed for single-CPU systems and has become essential in multi-core settings. An example of concurrent computing is a web browser that handles multiple open tabs one playing video, another loading a webpage, and a third executing scripts all of which are perceived to operate simultaneously.

In my view, concurrency is not solely about increasing speed; it focuses on effective task management and responsiveness. It tackles complexity via organised scheduling but also brings about challenges like race conditions and deadlocks, which require careful design and synchronisation to manage effectively.

Parallel Computing

Parallel computing involves breaking down a large computational problem into smaller subtasks that can be executed simultaneously across multiple processors or cores. This approach aims to enhance computational efficiency and speed. An example of this is the utilisation of Graphics Processing Units

(GPUs) in deep learning. Contemporary AI applications frequently depend on thousands of concurrent operations performed in parallel across GPU cores, facilitating the swift training of intricate models.

In my opinion, parallel computing signifies a remarkable advancement in performance capabilities. Nevertheless, it is not universally efficient for all problems tasks that cannot be evenly partitioned or that necessitate considerable inter-process communication may not experience the same level of benefit.

Distributed, Cluster and Grid Computing

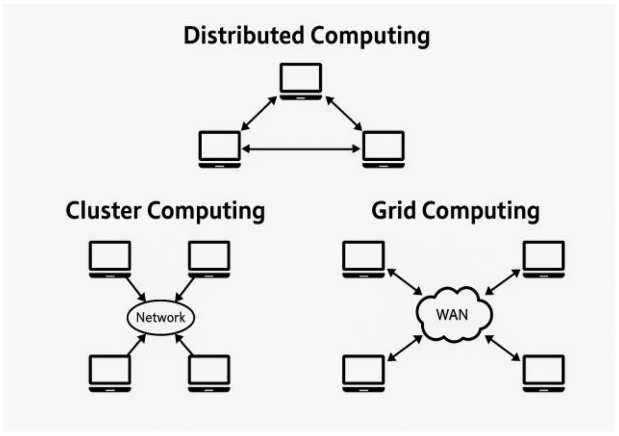


Figure 2. Distributed vs Cluster vs Grid Computing

Distributed Computing

Distributed computing involves multiple autonomous computers, each with its own memory and processing capability, working collaboratively over a network. Interaction among these systems generally takes place via message passing. A normal example would be in the infrastructure of Google's search engine, which allocates data and queries across a vast array of machines worldwide to deliver swift and dependable search outcomes.

I am particularly intrigued by distributed systems because of their intrinsic complexity. Although they provide remarkable scalability and fault tolerance, they also present challenges including synchronisation, latency, and consistency. Addressing these challenges necessitates the implementation of advanced protocols and architectures for effective management.

Cluster Computing

Cluster computing involves linking homogeneous computers via a local area network (LAN) to function as a unified system. This approach is frequently employed for high-performance computing (HPC) in areas like scientific research and financial modeling. Amazon EC2 High-Performance Computing clusters would be a great example for this, which facilitate demanding simulations such as weather forecasting and molecular modeling.

In my opinion, cluster computing provides a cost-efficient method to enhance local computing capabilities. Nevertheless, it necessitates considerable expertise for management and maintenance, and it is more constrained in geographic reach when compared to grid or cloud computing models.

Grid Computing

Grid computing connects diverse and geographically separated systems to address extensive computational challenges. In contrast to clusters, grids do not necessitate uniform hardware or proximity.

For instance, the SETI@home initiative, which enabled users to contribute unused CPU time to examine radio signals from outer space in the quest for extraterrestrial existence.

From my perspective, grid computing represents a significant historical advancement in distributed systems. Although it has largely been supplanted by cloud services in contemporary times, it pioneered the notion of decentralised collaboration in tackling intricate issues, a principle that continues to hold substantial relevance.

Cloud, Fog and Edge Computing

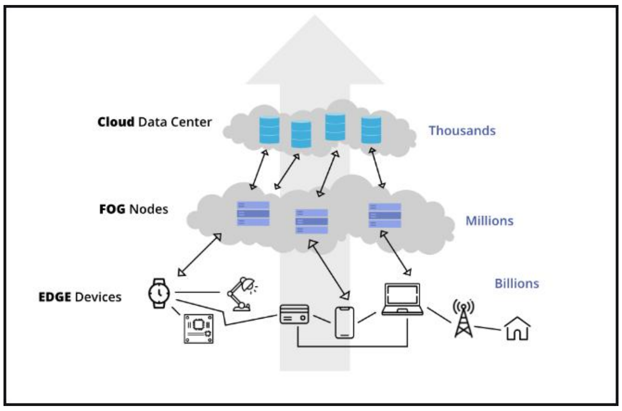


Figure 3. Cloud, Fog and Edge Computing

Cloud Computing

Cloud computing offers scalable and on-demand computing resources via the internet. Users can utilise services such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) without the need to manage the underlying hardware. Examples include AWS Lambda for serverless execution, Google Drive for document storage and collaboration, and Microsoft Azure virtual machines for infrastructure deployment.

From a personal perspective, cloud computing has been revolutionary. It has allowed startups, researchers, and enterprises to access extensive computing resources without the necessity of upfront investment. Nevertheless, I remain wary of the potential implications for data privacy and service dependency, considering that control predominantly lies with cloud providers.

Fog and Edge Computing

Fog and edge computing enhances the capabilities of cloud services by enabling data processing to be closer to its origin frequently at or in proximity to Internet of Things (IoT) devices thus minimising latency and reducing bandwidth demands. Edge computing generally pertains to processing that occurs directly on the device, whereas fog computing encompasses nearby network devices, including routers and gateways. An example of is the implementation of embedded processors in autonomous vehicles, which evaluate sensor data locally to execute immediate decisions without depending on cloud-based input.

I believe that fog and edge computing signify the future of real-time systems. They are crucial for applications where every millisecond is significant, such as in autonomous vehicles and industrial automation. The primary challenge resides in the management of resources, updates, and security within highly decentralised settings.

PART 2

I have used the Eclipse IDE and wrote the “Hello, Siam” program to test my workspace.
Please refer to Figure 4 for more information.

Week 2 (Process)

PART 1

What is a Process?

A process can be characterised as an active instance of a program that possesses its own memory space, process identifier (PID), and execution state. In contrast to a program that resides on disk, which is a static entity, a process serves as a dynamic unit of execution that engages with system resources such as the CPU, memory, and input/output devices. Processes generally move through various states, including new, ready, running, waiting (or blocked), and terminated, based on the tasks they are undertaking and the resources they need. The generation of processes is frequently overseen through system calls like fork(), which creates a child process as a duplicate of the parent, while exec() allows a process to be substituted by a new program. To ensure order and synchronisation, the wait() function permits a parent process to halt until its child process has completed execution. These mechanisms collectively establish the basis of multitasking, facilitating efficient and coordinated execution of multiple programs within contemporary operating systems.

Core Components of a Process

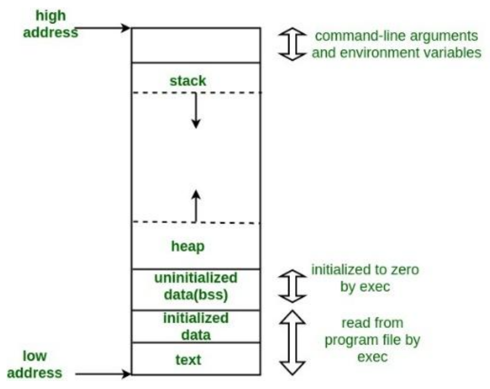

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
low address
text

Figure 5. Core component of process (GeeksforGeeks, 2025)

A process consists of several essential components:

- Text Section (Code): Comprises the directives for the program.
- Data Section: Contains global and static variables.
- Heap: Memory allocated dynamically during runtime (for instance, malloc in C, new in Java).
- Stack: Holds local variables, function parameters, and return addresses.
- Program Counter (PC): Signifies the address of the subsequent instruction.
- Process Control Block (PCB): Metadata preserved by the operating system, encompassing PID, process state, CPU registers, and scheduling details.

Together, these elements enable the operating system to effectively manage and execute processes.

PART 2: Experimentation with fork(), wait(), and exec()

fork() Demo

Observation:

- The first printf("Start") is shown once before any forks.
- After three forks, the number of processes increases to eight, each printing "Hello from a process".
- Output order may look jumbled because processes run concurrently.

Please refer to Figure 6 for more information.

wait() Demo

Observation:

- The child executes first: prints “working...” and “done.”
- The parent waits until the child exits, then prints the final message.
- This shows how wait() prevents the parent from finishing before its child.

Please refer to Figure 7 for more information.

exec() Demo

Observation:

- First message shows: “About to exec echo...”
- Then, the process image is replaced by /bin/echo, which prints “Hello from exec()”.
- If exec succeeds, the original program never resumes after it.

Please refer to Figure 8 for more information.

PART 3: Key Learnings

1. fork() multiplies processes (child copies parent).
2. wait() ensures parent doesn’t finish before child.
3. exec() replaces the current process with another program.
4. Loops with fork(), like in 02forkloop.c, create exponential process growth.
5. Understanding these calls helps explain how multitasking and process management work in real operating systems.

Week 3 (Scheduling)

PART 1

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

Processes tasks in the sequence of their arrival until they are fully executed. This method is straightforward and incurs minimal overhead; however, lengthy tasks may lead to the "convoy effect," resulting in suboptimal average response and turnaround times for shorter tasks that are delayed by longer ones.

Ideal for: Batch processing scenarios with comparable job durations.

Limitations: Inadequate responsiveness; vulnerable to delays caused by long-running tasks.

Shortest Job First (SJF, non-preemptive)

Consistently, choosing the task that has the least anticipated CPU duration, reduces the average waiting and turnaround times when execution times are predetermined. The non-preemptive version allows a currently executing task to be completed. There is a potential risk of long tasks being starved if shorter tasks continue to arrive.

Ideal for: Environments where accurate burst-time predictions are available.

Limitations: Predicting execution time is challenging; there is a risk of starvation.

Round-Robin (RR)

Each task that is ready is allocated a specific time quantum (time slice) before it is moved to the back of the ready queue. This approach promotes fairness and enhances interactivity; however, the response time is significantly influenced by the size of the quantum: if it is too small, it leads to excessive overhead; if it is too large, it tends to revert to First-Come, First-Served (FCFS).

Ideal for: Time-sharing and interactive workloads.

Limitations: Careful adjustment of the quantum is essential; throughput and turnaround times may be adversely affected when dealing with a variety of job sizes.

Lottery Scheduling (Proportional Share)

Each task is associated with tickets; the scheduler randomly selects a ticket during each time slice. On average, the CPU share is approximately equal to the ticket share; it is straightforward to add or remove tasks; this method circumvents the need for a global history per task. However, short intervals may lead to deviations from precise proportions.

Ideal for: Adaptable and equitable sharing without intricate accounting.

Limitations: Short-term variability; lacks awareness of deadlines.

Multi-level Feedback Queue (MLFQ)

A system of multiple queues is established, each with descending priorities. New tasks are initiated at the highest priority level; tasks that utilize their entire time allocation are relegated to lower priority queues; interactive tasks that complete early maintain their priority status. In cases of ties within a queue, a Round Robin (RR) approach is employed. To combat starvation, periodic "priority boosts" are implemented to elevate all tasks, while improved accounting measures are in place to deter manipulation through frequent I/O yields. This system is widely adopted across various platforms, including BSD, Solaris, and Windows.

Ideal for: Diverse workloads that include both interactive and CPU-bound tasks.

Limitations: Numerous adjustable parameters (such as queue count, time slices, and boost intervals) exist, with no definitive "correct" configuration.

Priority Scheduling of Aging

Fixed or dynamic priorities dictate the sequence of tasks; aging incrementally elevates the priorities of waiting tasks to avert starvation. This approach is frequently integrated into MLFQ-style systems as a practical implementation strategy.

Table of Different Scheduling Algorithms

Algorithm	Preemptive	Strengths	Weaknesses	Suitable for
FCFS (First Come First Served)	No	Simple, fair in arrival order	Convoy effect (long job delays others)	Batch jobs of similar length
SJF (Shortest Job First)	No	Minimizes average waiting time	Starvation of long jobs	Known burst times
SRT (Shortest Remaining Time)	Yes	Good response time	High context switch cost, starvation	Interactive workloads
Round Robin (RR)	Yes	Fair, predictable response	Throughput depends on time quantum	Time-sharing systems
Priority Scheduling (with aging)	Both	Flexible, prevents starvation (with aging)	Without aging → starvation	Systems needing priority control
MLFQ (Multi-Level Feedback Queue)	Yes	Handles diverse workloads, balances priorities	Complex tuning	General-purpose OS
Lottery Scheduling	Yes	Proportional fairness, flexible	Random short-term unfairness	Experimental/flexible systems

PART 2

1. Main tasks of the scheduler

Select the next task that is ready to run (policy), determine when to interrupt (preemptive control), allocate CPU time slices, and balance objectives such as CPU utilization, turnaround time, response time, and missed deadlines.

2. Preemptive vs non-preemptive

Preemptive policies can interrupt a currently running task (due to a timer or the arrival of a higher-priority task) to execute another for instance, SRT, RR, MLFQ in contrast, non-preemptive policies (FCFS, SJF) allow a running task to retain control of the CPU until it either blocks or completes.

3. Advantage of a shorter quantum (RR/MLFQ)

Leads to a quicker average response for interactive tasks; the system appears more responsive. Trade-off: increased context switches and overhead.

4. Advantage of a longer quantum

Results in reduced overhead and improved throughput; however, it may lead to slower response times tending towards FCFS behavior.

5. Why feedback scheduling helps interactive jobs

MLFQ interprets CPU bursts as signals: short or interactive bursts maintain a position in high-priority queues ensuring quick service, longer CPU bursts result in a downward shift backgrounded. This method approximates SJF without requiring perfect knowledge and maintains rapid responsiveness for tasks driven by I/O.

6. Starvation Analysis

SJF (non-preemptive): Yes, it is possible. A continuous influx of short jobs can indefinitely postpone the execution of long jobs.

RR: No (in practice). Given a finite time, quantum and a limited ready set, all processes take turns; thus, starvation is mitigated although overall throughput may be compromised.

FCFS: There is no genuine starvation, but prolonged waiting times can be significantly known as the convoy effect.

Fixed Priority: Yes. Tasks with low priority may experience starvation if there is an abundance of high-priority tasks. Aging or boosts are necessary.

MLFQ: Starvation can occur without boosts. Appropriate priority boosting as per Rule 5 can avert starvation and counteract “gaming.”

Week 4 (Threads)

PART 1

Why Threads?

Processes enabled multitasking; however, each process possesses its own memory space, resulting in costly data sharing and expensive communication. Threads address this issue by permitting subtasks within a process to share memory while executing independently. This capability is essential for maintaining responsiveness —for instance, a monitoring system can halt video transmission, react to user input, and subsequently resume transmission without any delays.

What is a Thread?

A thread constitutes a lightweight subtask within a program, sharing the address space of the process while maintaining its own CPU context, which includes the program counter, stack pointer, registers, and stack. Threads serve as the fundamental unit for CPU scheduling, in contrast to processes, which are the primary units for resource allocation.

Advantages and Disadvantages

Advantages: The ability to perform overlap computation alongside input/output operations, reduced costs associated with context switching compared to processes, enhanced efficiency on multiprocessor systems, and streamlined communication through shared memory

Disadvantages: Increased complexity in debugging, the possibility of race conditions, and challenges in maintaining thread safety.

User-level vs Kernel-level Threads

Threads can be implemented at either the user level or the kernel level. User-level threads are overseen by a runtime library within user space. They are characterized by their lightweight nature and rapid context switching, as they circumvent system calls. However, a significant disadvantage is that if one thread becomes blocked for instance, during I/O operations, the entire process may also become blocked, since the kernel lacks awareness of the individual threads. In contrast, kernel-level threads are managed directly by the operating system. This management allows the kernel to schedule these threads independently across multiple processors, facilitating genuine parallelism and improved responsiveness during blocking operations. Nonetheless, this advantage comes with the trade-off of slower context switching due to the involvement of the kernel. Many contemporary systems employ a hybrid model for example, Java’s threads are directly mapped to OS-level threads to achieve a balance between flexibility and performance.

Java Threads

Java provides support for threads through two primary methods:

1. Extending the Thread class, by overriding the run() method and initiating it with .start()
2. Implementing the Runnable interface, which offers greater flexibility as Java does not allow multiple inheritance

Key APIs include .start(), .join(), .setPriority() (with a range from MIN_PRIORITY=1 to MAX_PRIORITY=10; the default is 5), .wait(), and .notify(). Methods that have been deprecated include .stop() and .suspend(). Thread pools (Executor Service) are utilized to efficiently manage many short-lived tasks, thereby minimizing the overhead associated with the continuous creation and destruction of threads.

PART 2

Example 1 & 1b – Main vs Child Threads, Priorities and Naming

Objective: To examine the concurrent execution of the main thread alongside child threads in Java and subsequently enhance the example by assigning specific priorities and names to these threads.

Method:

- In the initial program (Example 1), the main thread initiated and commenced several child threads without the allocation of names or priorities.
- Each thread executed the PrintNumbers class method, which displayed a series of numbers on the console.
- In the modified program (Example 1b), the child threads were assigned meaningful names (T0...T9) through the use of .setName(). The main thread was renamed to "Main".
- Priorities were designated using .setPriority(int): T0 received the highest priority (Thread.MAX_PRIORITY = 10), T1 was assigned the lowest priority (Thread.MIN_PRIORITY = 1), while the remaining threads were given normal priority (Thread.NORM_PRIORITY = 5).
- The run() method was revised to output each thread's name and priority upon initiation.

Observation:

1. In Example 1, the console output illustrated the main thread (Main Thread Id: 1) operating concurrently with child threads (e.g., Thread id: 29, Thread id: 30, Thread id: 31). Their outputs were interspersed, showcasing concurrent execution as managed by the JVM/OS scheduler.
2. In Example 1b, the console verified that names and priorities were accurately assigned (e.g., T0 initiated (priority 10), T1 initiated (priority 1)). The output remained interleaved, yet the custom names enhanced readability. Although priorities affected scheduling, the order of execution was not strictly enforced, as the JVM regards priorities merely as suggestions.

Key Concepts Demonstrated:

- Concurrent execution: The main and child threads operate simultaneously, yielding interleaved output.
- Thread priorities: The range extends from MIN_PRIORITY = 1 to MAX_PRIORITY = 10, with NORM_PRIORITY = 5 as the default. Priorities serve as scheduling suggestions rather than certainties.
- Naming threads: The setName() method facilitates the assignment of descriptive identifiers to threads, thereby enhancing clarity during debugging and output evaluation.

Reflection:

Example 1 and 1b collectively demonstrate the way Java threads allocate CPU time, the default occurrence of interleaving, and the ways in which developers can improve control and clarity using priorities and naming conventions. Although priorities do not ensure a rigid sequence, they serve to indicate the significance of threads. Additionally, assigning names to threads facilitates the interpretation of multithreaded outputs.

Please refer to Figure 9 and Figure 10 for more information.

Example 2 – Finding Min and Max Using Threads

Objective: To adjust classes A and B so that one outputs the minimum value and the other outputs the maximum value from a specified array

Method: The array was passed into the constructor of each thread, with the min/max logic implemented in their respective run() methods, and both threads were initiated in the main program. The join() method was utilized to ensure that the program awaited the completion of both threads.

Observation: The console successfully displayed both the minimum (-2) and maximum (13) values. Although the execution order varied, both results were accurately presented, followed by the message "Both threads finished."

Reflection: This exercise illustrates the concept of parallel computation for independent tasks using threads and emphasizes the importance of safe data transmission through constructors.

Please refer to Figure 11 for more information.

Example 3 & 3b – Data Sharing and Thread Behaviour

Objective: The aim of this exercise was to investigate the behaviour of various types of data, local variables, instance fields, and class (static) fields when accessed concurrently by multiple threads. Furthermore, we assessed the impact of creating threads with distinct instances and introducing a delay in thread execution to determine how these modifications affect data sharing and race conditions.

Method:

1. **Example 3 (Same Instance):**
   Two threads (t1 and t2) were created using the same IncrementTest object. Each thread increments three categories of variables: localData: defined within the run() method (thread-local). instanceData: an instance field of the IncrementTest object (shared if threads utilize the same instance). classData: a static field that is shared across all instances of IncrementTest. Both threads were initiated simultaneously.
2. **Example 3 (Two Instances):**
   The program was modified so that t1 and t2 operated on different instances of IncrementTest. This guarantees that instanceData is not shared between the two threads, while classData continues to be shared.
3. **Example 3b (Staggered Start):**
   This example is similar to Example 3 (shared instance), but the execution was staggered by incorporating Thread.sleep(1000) between the initiation of t1 and t2. This adjustment minimized the overlap between the two threads, enabling t1 to complete the majority of its increments prior to the commencement of t2.

Observations:

1. **Example 3 (Same Instance):**
   a. Each thread accurately reported localData = 1,000,000

    b. The instanceData was incremented by both threads but ultimately fell short of the anticipated total (≈20,000,000) due to lost updates resulting from race conditions.
    c. The classData, being static, was also shared and consistently concluded with erroneous values, thereby confirming contention between the two threads.
2. **Example 3 (Two Instances):**
    a. Once again, both threads accurately reported localData = 1,000,000
    b. Each thread's instanceData was approximately 1,000,000 since the threads now functioned on distinct objects, preventing any interference.
    c. The classData remained shared across all instances, and its value was lower than expected due to concurrent modifications.
3. **Example 3b (Staggered Start):**
    a. Both threads displayed localData = 1,000,000.
    b. The instanceData achieved the full expected total of 20,000,000.
    c. The classData also reached the anticipated 20,000,000.
    d. As t2 commenced after t1 had largely completed, there was no significant overlap, thus race conditions did not occur.

Analysis & Discussion:

- Local variables (localData): are maintained on the stack of each thread, ensuring they remain private and accurate, unaffected by concurrent operations.
- Instance fields (instanceData): are only shared when multiple threads access the same object. In the case of a shared instance, both threads may increment the same field simultaneously, leading to lost updates. Conversely, using separate instances eliminates this interference.
- Class/static fields (classData): are globally shared, resulting in contention whenever they are incremented concurrently.
- Race conditions: The operation x++ is not atomic; it involves a sequence of read, increment, and write actions. When threads interleave, increments may be lost.
- Staggered start (Example 3b): By delaying thread execution, the overlap was minimized, and updates seemed "perfect." However, this does not constitute a genuine solution. If both threads had executed simultaneously, the race conditions would still exist. Ensuring correctness necessitates explicit synchronization (synchronized, locks, or atomic variables).

Reflection:

This lab underscores the impact of variable scope on sharing in multithreaded applications and illustrates how race conditions arise when updates are not atomic. Utilizing separate objects can prevent interference at the instance level, yet static fields remain susceptible. Staggering execution may obscure race conditions, but effective concurrency control demands synchronization mechanisms.

Please refer to Figure 12, 13 and 14 for more information.

Week 5 (Locks 1)

PART 1

Theory summary: Concurrent programs frequently involve several threads that access shared data. In the absence of proper coordination, this can lead to race conditions within the critical section, resulting in indeterminate behaviour and unpredictable outputs. To maintain correctness, mutual exclusion (ME) is essential, ensuring that only one thread can execute the critical section at any given time, thereby preserving atomicity.

In Java, ME can be implemented using the synchronized keyword or through explicit lock objects. Synchronized methods and blocks utilize an object's intrinsic lock, automatically managing the acquisition and release of locks, although they lack certain flexibilities such as interruption or timeouts. The ReentrantLock class offers more sophisticated features, including tryLock(), timeouts, and interruptible acquisition, making it a more suitable choice in complex situations.

A significant difference exists between instance-level locks for example synchronized(this), which only protect the state of individual objects, and class-level locks such as static final objects or the class itself, which are necessary for coordinating access to shared static fields across all instances.

PART 2 (Implementation & Results)

Task 0: Baseline (no synchronisation)

The original program yielded variable and unpredictable outcomes across several executions. Although localData consistently achieved the anticipated figure of 10,000,000, both instanceData and classData did not align with the expected totals. This behavior indicates the presence of race conditions within the critical section, where multiple threads simultaneously modified shared data without proper mutual exclusion.

Please refer to Figure 15 for more information.

Task 1: Synchronised Method(s)

The implementation of a synchronized method guaranteed that only a single thread could perform the increment operation at any given time on a shared instance. When both threads interacted with the same object, the outcomes became consistent, with classData and instanceData accurately reflecting the correct totals.

Nevertheless, when two distinct objects were utilized, a discrepancy emerged: the initial printed values frequently indicated that classData was greater than instanceData. This situation arose because synchronization on a method applies locks per object, implying that each instance possesses its own lock. Given that classData is a static field shared among all instances, its updates were not effectively coordinated, leading to inconsistent outputs.

Please refer to Figure 16 and 17 for more information.

Task 2: Synchronised Block with Shared Lock

To address the inconsistency observed across various instances, a class-level lock static final Object was implemented. By synchronizing on this common lock, both threads rrespective of their respective instances were compelled to coordinate their access to the critical section. The findings indicated that classData achieved a value of approximately 20,000,000, while each object's instanceData stabilized at roughly 10,000,000, thereby resolving the previous discrepancy. This validated the necessity of a shared lock across all instances to effectively safeguard static variables in concurrent programming.

Please refer to Figure 18 for more information.

Task 3: Reentrant Lock

The ultimate implementation employed a ReentrantLock to explicitly manage access to the critical section. The outcomes were in alignment with those achieved through synchronized blocks: accurate totals were generated even in the presence of multiple instances. In contrast to intrinsic locks, ReentrantLock offers enhanced capabilities, including timed lock acquisition, non-blocking tryLock(), and interruptible locking. These attributes render it more adaptable and resilient in intricate concurrency situations, even though the behaviour in this instance reflected that of synchronised blocks.

Please refer to Figure 19 for more information.

Overall Analysis

- Baseline: Showed inconclusive outcomes as a result of race conditions.
- Synchronized Method: Achieved accuracy with a single shared instance, yet encountered failures with multiple instances due to the independent nature of each object's intrinsic lock.
- Synchronized Block with Shared Lock: Resolved the inconsistency by implementing a class-level lock, thereby guaranteeing appropriate mutual exclusion among all instances.
- ReentrantLock: Delivered reliable results while providing enhanced flexibility compared to intrinsic locks.

Week 6 (Locks 2)

PART 1

Theory Summary: In concurrent programming, it is essential to access shared variables with caution to prevent race conditions, which occur when multiple threads interleave their updates to the same data, resulting in unpredictable outcomes. Lecture 6 concentrated on the protection of static variables, which are accessible across all instances of a class. Instance-level locks, such as synchronized(this), are inadequate for static fields since they only safeguard per-object state. To achieve mutual exclusion across all instances, class-level locks are necessary. Two common methods in Java include:

- synchronized(ClassName.class): this approach synchronizes on the Class object, ensuring that a single intrinsic lock is utilized across all threads, irrespective of the instance
- Static ReentrantLock: this offers explicit control over locking, permitting flexible operations such as tryLock(), timeouts, and interruptible acquisition

The lecture also explored the implementation of locks at the hardware and operating system levels. Earlier techniques, such as disabling interrupts or employing a binary flag, proved inadequate, as they either failed on multiprocessor systems or resulted in wasted CPU cycles. Effective solutions necessitate atomic hardware primitives like test-and-set or compare-and-swap to ensure proper exclusion. While spinlocks may perform efficiently on multiprocessors for brief critical sections, blocking is typically more advantageous on uniprocessors to prevent unnecessary CPU cycle consumption.

PART 2

1. Baseline (no concurrency control)

Objective: To demonstrate the occurrence of race conditions when a shared static variable is incremented simultaneously without any synchronization mechanisms.

Approach: Several threads concurrently increment the shared static counter, no locking mechanisms are implemented.

Findings: The ultimate value remains consistently lower than the expected total (indicating lost updates), and variations between runs are noted, suggesting an unpredictable program behaviour due to data races.

Please refer to Figure 20 for more information.

2. Synchronisation with synchronized (Another.class)

The increment of the static variable has been relocated within a block that is synchronized on Another.class. This guarantees that only a single thread can modify Another.i at any given moment, irrespective of the instance that is executing. Upon multiple executions, the ultimate value of Another.i consistently reached 20,000,000, aligning perfectly with the theoretical total. This behaviour is deterministic, demonstrating that synchronization at the class level effectively safeguards shared static fields.

Please refer to Figure 21 for more information.

3. Synchronisation with Static ReentrantLock

A static final ReentrantLock was implemented to protect modifications to the static variable. Each increment occurred within a lock() / unlock() block. As a result, the final value of i was once more 20,000,000, aligning with the anticipated total. The behavior remained consistent, showcasing correctness. Although it yields outcomes similar to synchronized locking, ReentrantLock offers extra functionalities, including timed and non-blocking lock attempts.

Please refer to Figure 22 for more information.

Comparative Analysis

- Baseline: A race condition resulted in the loss of updates to the static field.
- Synchronized Class Lock: Produces accurate results; straightforward to implement with synchronized(Another.class).
- ReentrantLock: Also yields correct outcomes; offers greater flexibility for complex concurrency situations, albeit with a slight decrease in speed due to overhead.

PART 3 (Conceptual Questions)

What is a race condition?

A race condition arises when multiple threads concurrently access and alter shared data without adequate synchronization. Due to the lack of control over the execution order, the ultimate result becomes unpredictable and may differ across various executions of the program. In the context of this lab, the static variable Another.i yielded inconsistent and erroneous results as a consequence of a race condition.

How can we protect against race conditions?

Race conditions can be mitigated by implementing mutual exclusion within the critical section. This guarantees that only a single thread is allowed to modify the shared data at any given moment. In Java, this can be accomplished through the use of intrinsic locks (such as synchronized methods or blocks) or explicit locks (like ReentrantLock). Both of these approaches serialize access to the shared variable, resulting in deterministic and accurate outcomes.

Can locks be implemented by simply reading and writing to a binary variable in memory?

No. A basic flag variable that relies on load/store operations lacks atomicity and fails to ensure mutual exclusion. It is possible for two threads to concurrently evaluate the flag, resulting in both entering the critical section and compromising correctness. Effective lock implementations necessitate atomic hardware primitives like test-and-set or compare-and-swap, which ensure that the checking and updating occur as a singular, indivisible operation.

Why is it better to block rather than spin on a uniprocessor?

On a uniprocessor, spinning consumes CPU cycles unnecessarily since no other thread can execute while one thread possesses the lock. In contrast, blocking permits the waiting thread to relinquish the CPU, allowing the thread that holds the lock to finish its task and release the lock more quickly. Consequently, blocking is more efficient than spin-waiting in a single-processor environment.

Why is it sometimes better to spin rather than block on a multiprocessor?

On multiprocessors, spinning proves to be efficient when the critical section is brief. While one thread is engaged in spinning on a particular CPU, another thread may swiftly release the lock on a separate CPU, facilitating progress without incurring the costs associated with a context switch. In these scenarios, spinning circumvents the expenses related to blocking and waking threads, rendering it a more favourable option for transient lock contention.

Conclusion

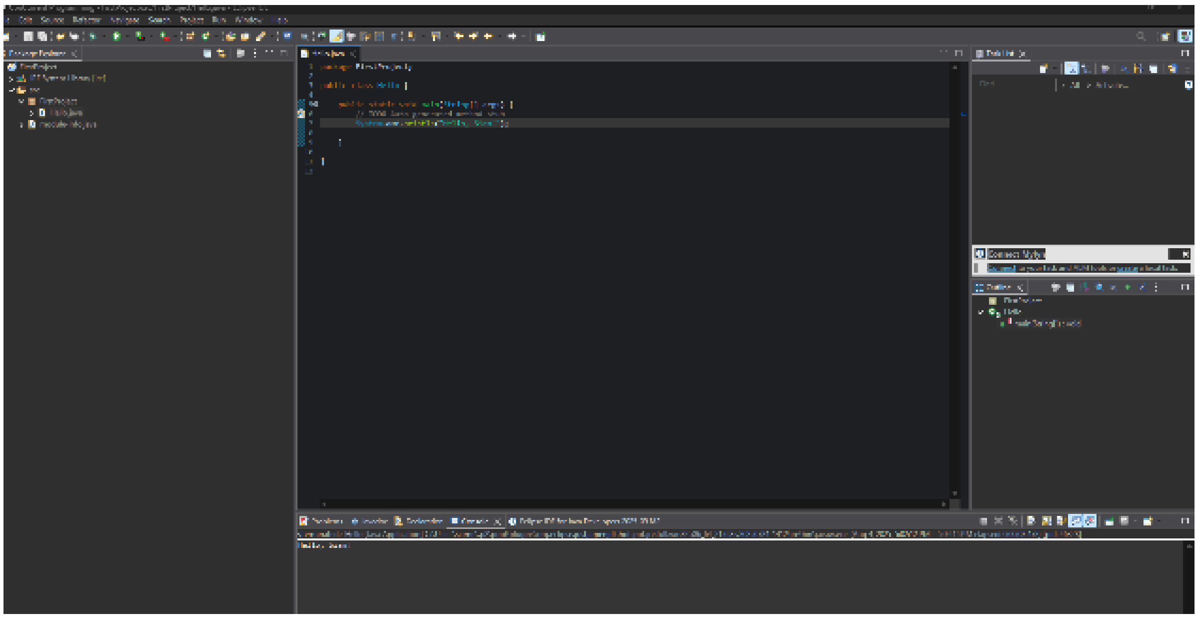
Throughout the initial six weeks of concurrent programming laboratories, I have systematically enhanced both my theoretical knowledge and practical abilities related to processes, threads, and synchronization. The introductory labs elucidated essential concepts such as concurrency, parallelism, and process management, while the following weeks emphasized the complexities of scheduling, thread behaviour, and the impact of variable scope on shared data.

The later labs focused on locks illustrated how race conditions can occur when shared state is accessed without adequate protection, and how various synchronization mechanisms can address these issues. By experimenting with synchronized methods, synchronized blocks, class-level locks, and ReentrantLock, I was able to discern the trade-offs between simplicity, flexibility, and performance. Notably, I discovered that local variables are inherently safe, instance variables can be safeguarded by per-object locks, and static variables necessitate class-wide or explicit locks to maintain correctness.

In summary, the laboratories established a robust foundation in the principles of concurrency control. They highlighted that while concurrent systems enhance responsiveness and performance, they also pose risks such as unpredictable behavior, deadlocks, and inefficiency if not meticulously managed. The proper application of synchronization constructs, along with an understanding of the hardware implications of spinning versus blocking, is crucial for developing reliable and efficient concurrent applications.

Appendix

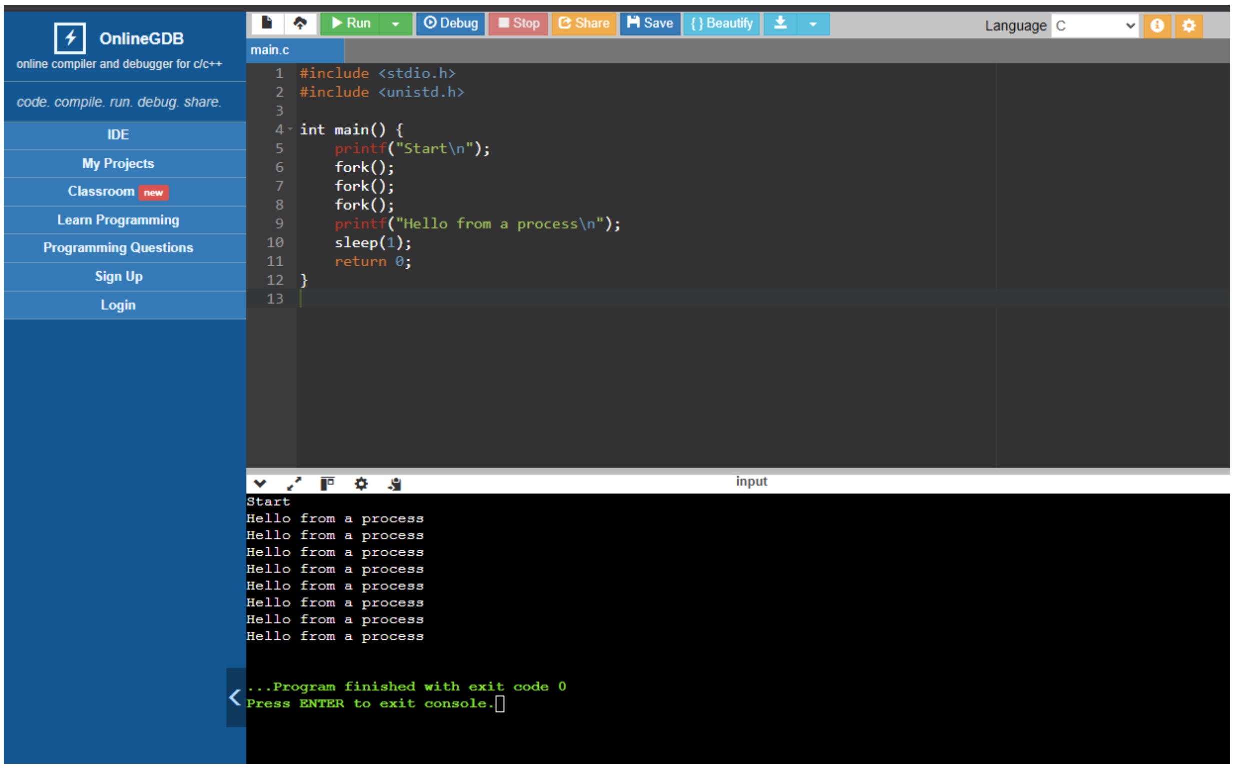
Figure 4



```
package FirstProject;

public class Hello {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello, [illegible]");

    }

}
```

Figure 4. Screenshot of IDE and Program

Figure 6

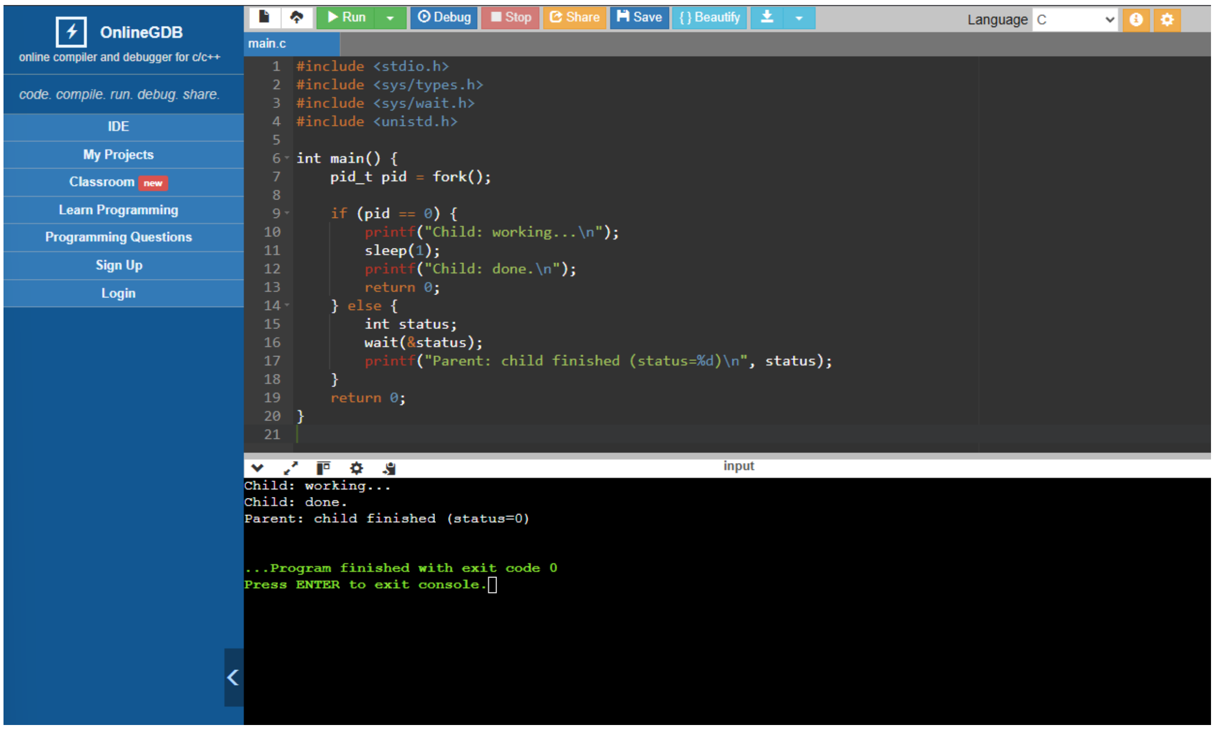


```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Start\n");
    fork();
    fork();
    fork();
    printf("Hello from a process\n");
    sleep(1);
    return 0;
}
```

```
Start
Hello from a process
Hello from a process
Hello from a process
Hello from a process
Hello from a process
Hello from a process
Hello from a process
Hello from a process


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 6. Code use for fork()

Figure 7

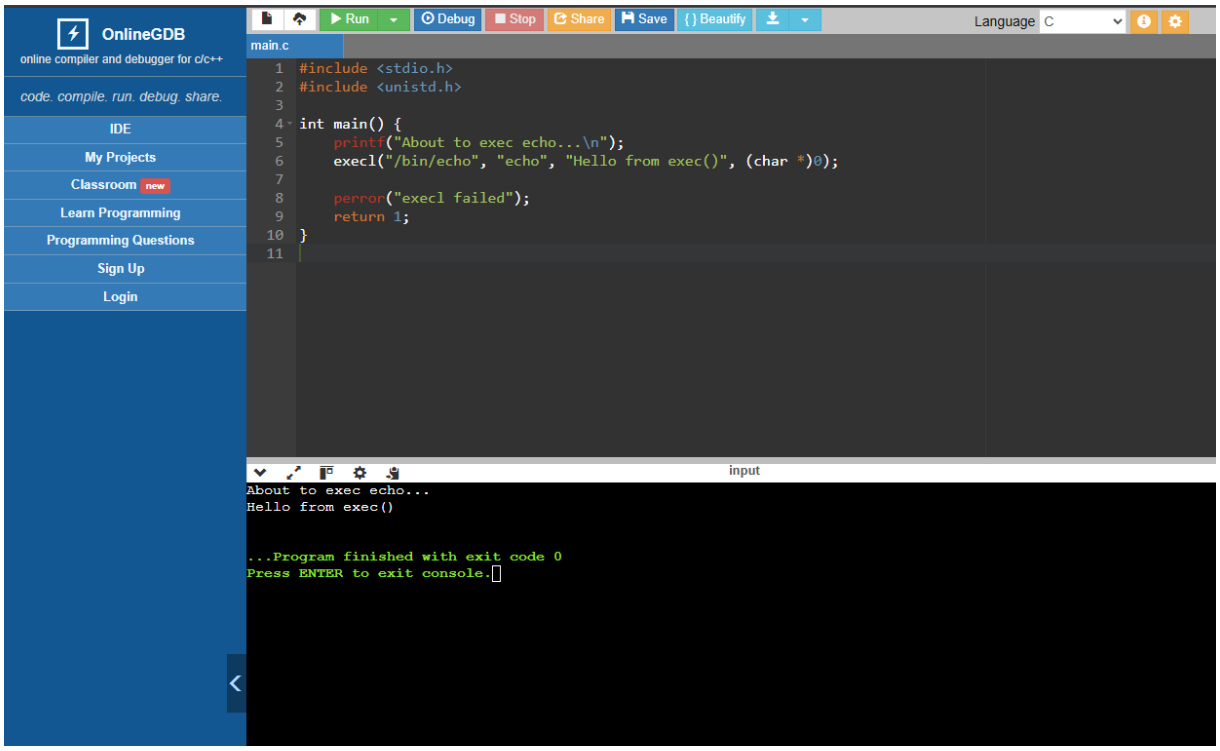


```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child: working...\n");
        sleep(1);
        printf("Child: done.\n");
        return 0;
    } else {
        int status;
        wait(&status);
        printf("Parent: child finished (status=%d)\n", status);
    }
    return 0;
}
```

```
Child: working...
Child: done.
Parent: child finished (status=0)


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 7. Code used for wait()

Figure 8

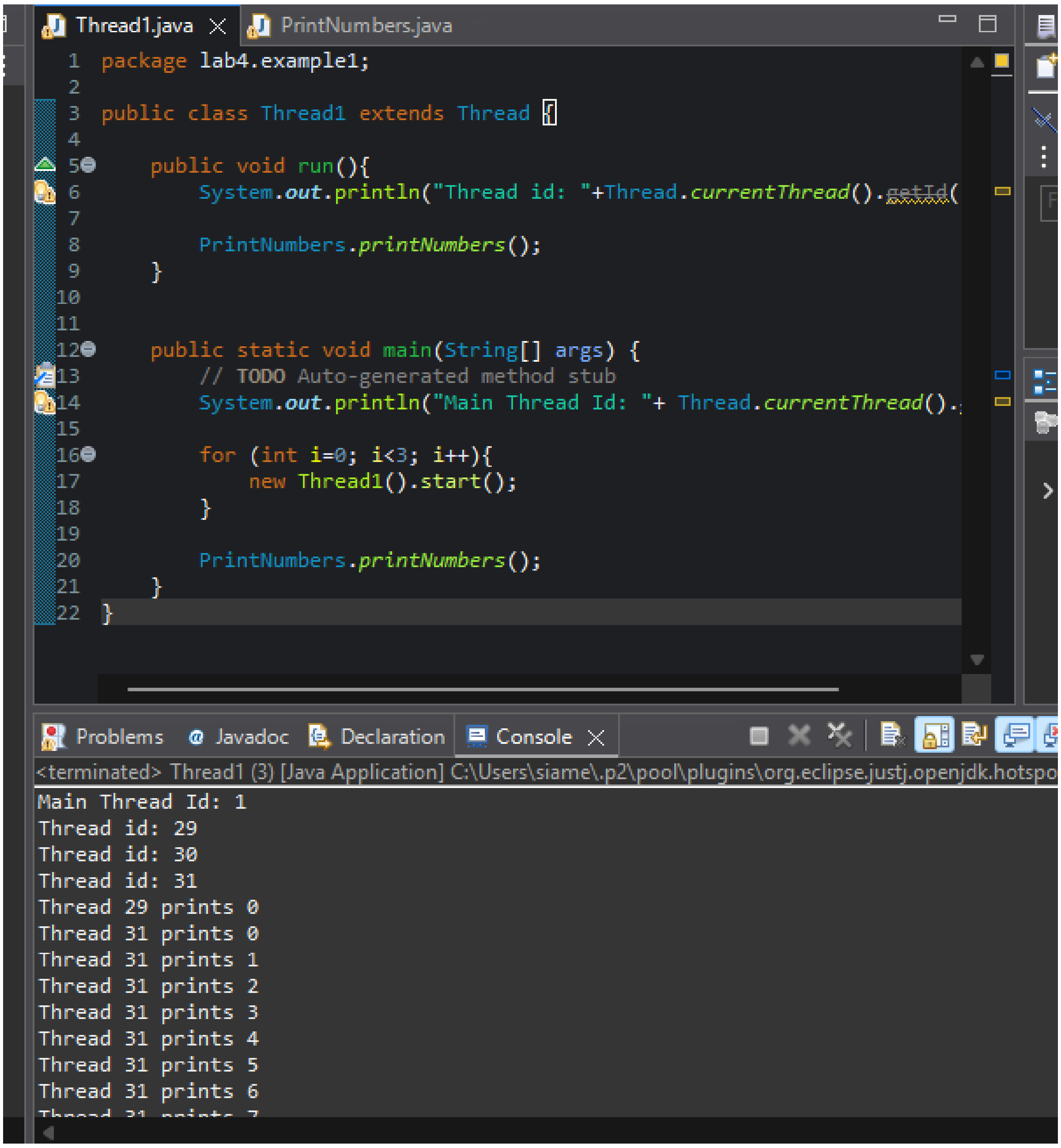


```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("About to exec echo...\n");
    execl("/bin/echo", "echo", "Hello from exec()", (char *)0);

    perror("execl failed");
    return 1;
}
```

```
About to exec echo...
Hello from exec()


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 8. Code used for exec()

Figure 9


Thread1.java PrintNumbers.java

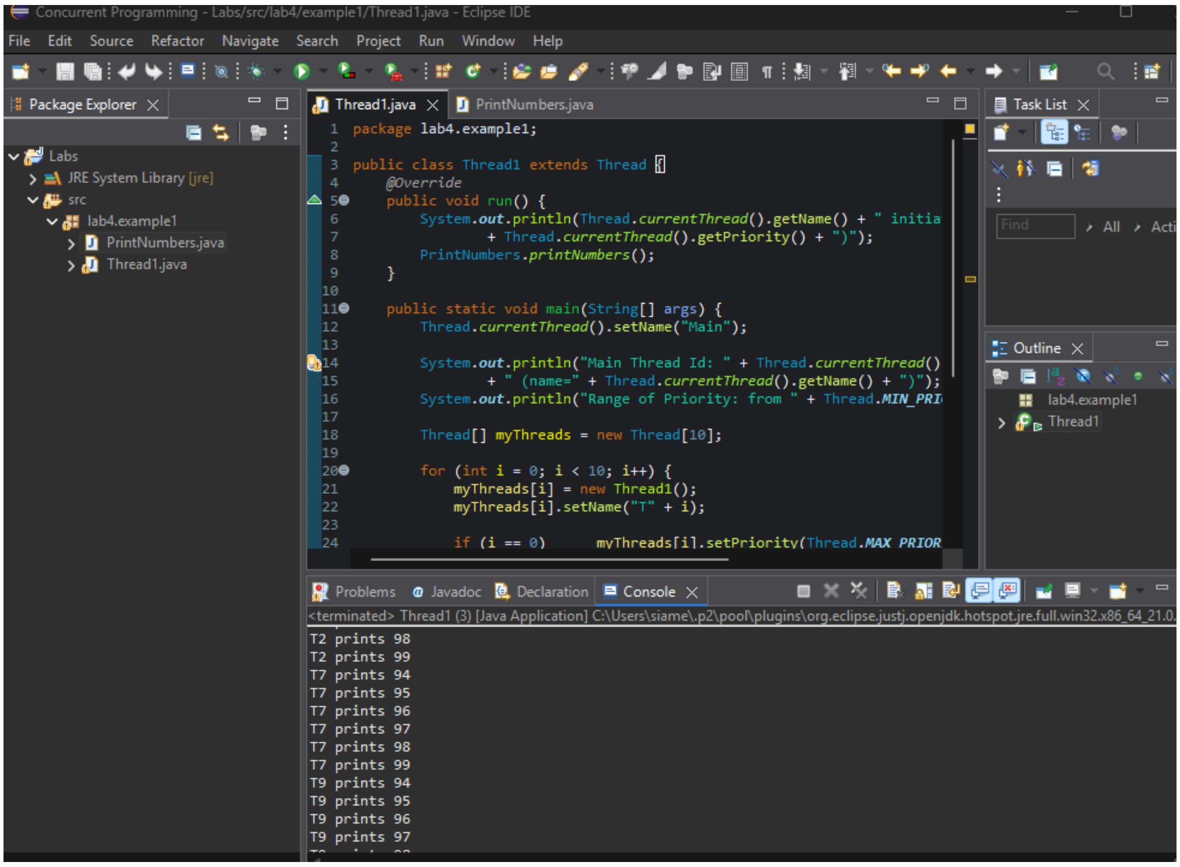
```java
package lab4.example1;

public class Thread1 extends Thread {

    public void run(){
        System.out.println("Thread id: "+Thread.currentThread().getId(

        PrintNumbers.printNumbers();
    }


    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Main Thread Id: "+ Thread.currentThread().

        for (int i=0; i<3; i++){
            new Thread1().start();
        }

        PrintNumbers.printNumbers();
    }
}
```

Problems Javadoc Declaration Console

```
<terminated> Thread1 (3) [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspo
Main Thread Id: 1
Thread id: 29
Thread id: 30
Thread id: 31
Thread 29 prints 0
Thread 31 prints 0
Thread 31 prints 1
Thread 31 prints 2
Thread 31 prints 3
Thread 31 prints 4
Thread 31 prints 5
Thread 31 prints 6
Thread 31 prints 7
```

Figure 9. Example 1.1: Main and child threads interleaving with default priorities and names

Figure 10

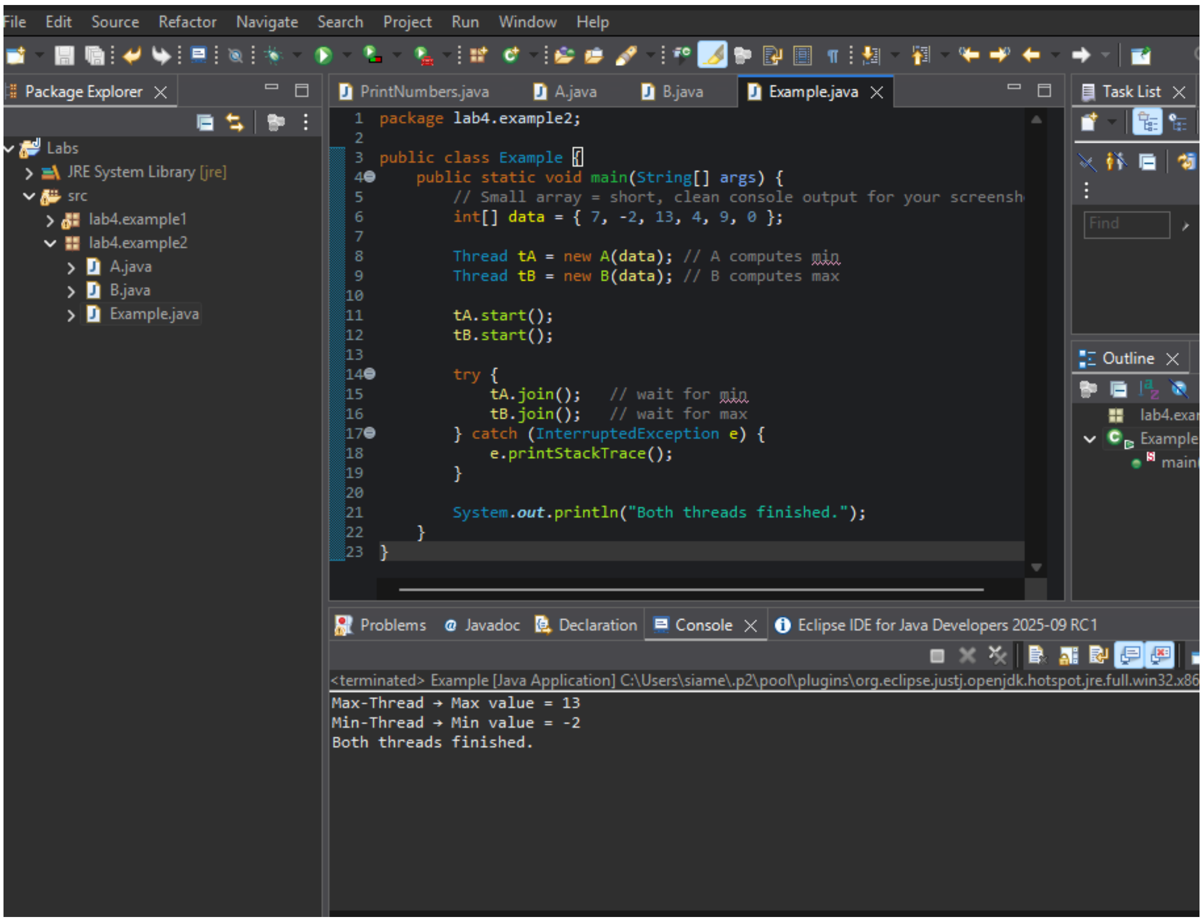


```java
package lab4.example1;

public class Thread1 extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " initia
                + Thread.currentThread().getPriority() + ")");
        PrintNumbers.printNumbers();
    }

    public static void main(String[] args) {
        Thread.currentThread().setName("Main");

        System.out.println("Main Thread Id: " + Thread.currentThread()
                + " (name=" + Thread.currentThread().getName() + ")");
        System.out.println("Range of Priority: from " + Thread.MIN_PRI

        Thread[] myThreads = new Thread[10];

        for (int i = 0; i < 10; i++) {
            myThreads[i] = new Thread1();
            myThreads[i].setName("T" + i);

            if (i == 0)       myThreads[i].setPriority(Thread.MAX_PRIOR
```

```
<terminated> Thread1 (3) [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.
T2 prints 98
T2 prints 99
T7 prints 94
T7 prints 95
T7 prints 96
T7 prints 97
T7 prints 98
T7 prints 99
T9 prints 94
T9 prints 95
T9 prints 96
T9 prints 97
```

Figure 10. Example 1.2: Threads with assigned names and priorities running concurrently.

Figure 11

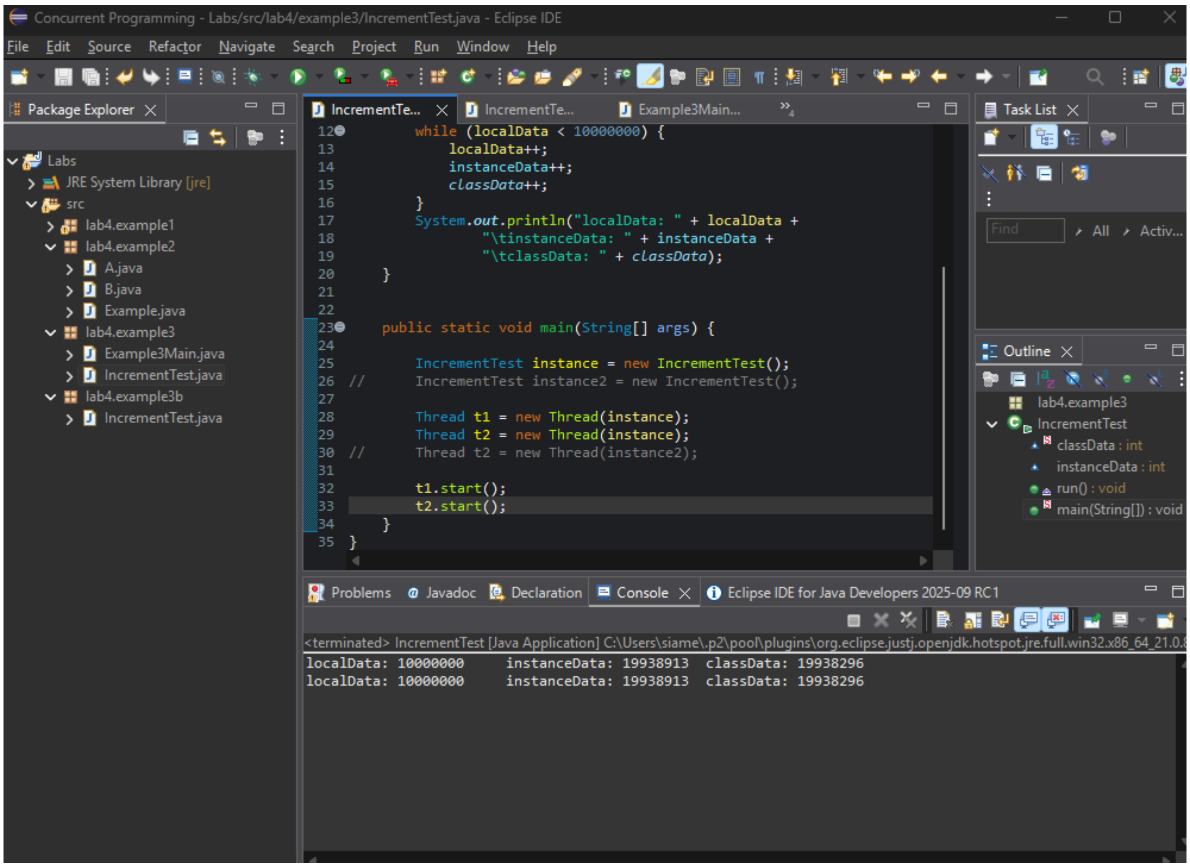


```java
package lab4.example2;

public class Example {
    public static void main(String[] args) {
        // Small array = short, clean console output for your screensh
        int[] data = { 7, -2, 13, 4, 9, 0 };

        Thread tA = new A(data); // A computes min
        Thread tB = new B(data); // B computes max

        tA.start();
        tB.start();

        try {
            tA.join();   // wait for min
            tB.join();   // wait for max
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Both threads finished.");
    }
}
```

```
<terminated> Example [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86
Max-Thread → Max value = 13
Min-Thread → Min value = -2
Both threads finished.
```

Figure 11. Example 2: Threads computing minimum and maximum concurrently

Figure 12

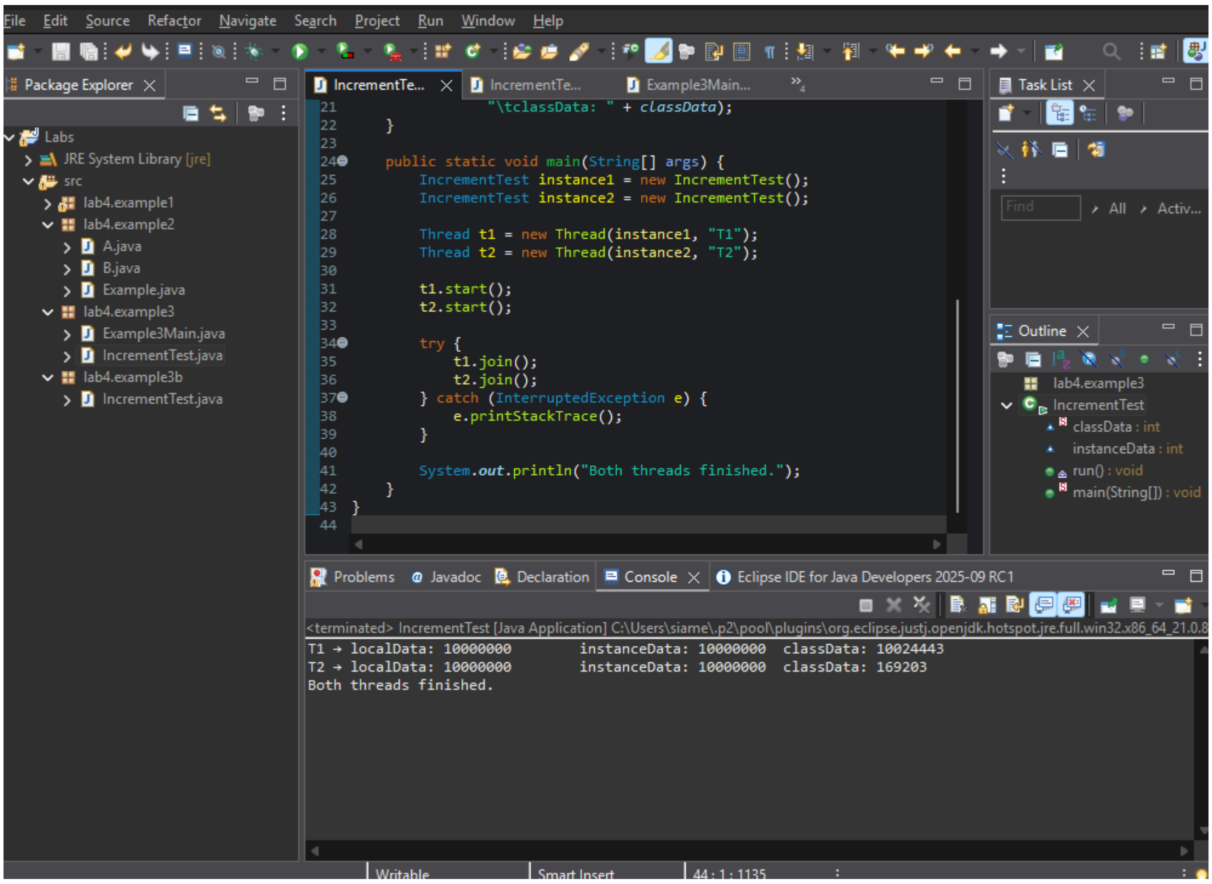

Concurrent Programming - Labs/src/lab4/example3/IncrementTest.java - Eclipse IDE

```java
        while (localData < 10000000) {
            localData++;
            instanceData++;
            classData++;
        }
        System.out.println("localData: " + localData +
                "\tinstanceData: " + instanceData +
                "\tclassData: " + classData);
    }


    public static void main(String[] args) {

        IncrementTest instance = new IncrementTest();
//      IncrementTest instance2 = new IncrementTest();

        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);
//      Thread t2 = new Thread(instance2);

        t1.start();
        t2.start();
    }
}
```

```
<terminated> IncrementTest [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.
localData: 10000000     instanceData: 19938913  classData: 19938296
localData: 10000000     instanceData: 19938913  classData: 19938296
```

Figure 12. Example 3.1: Same Instance: Output showing localData correct while instanceData and classData fall short due to race conditions

Figure 13

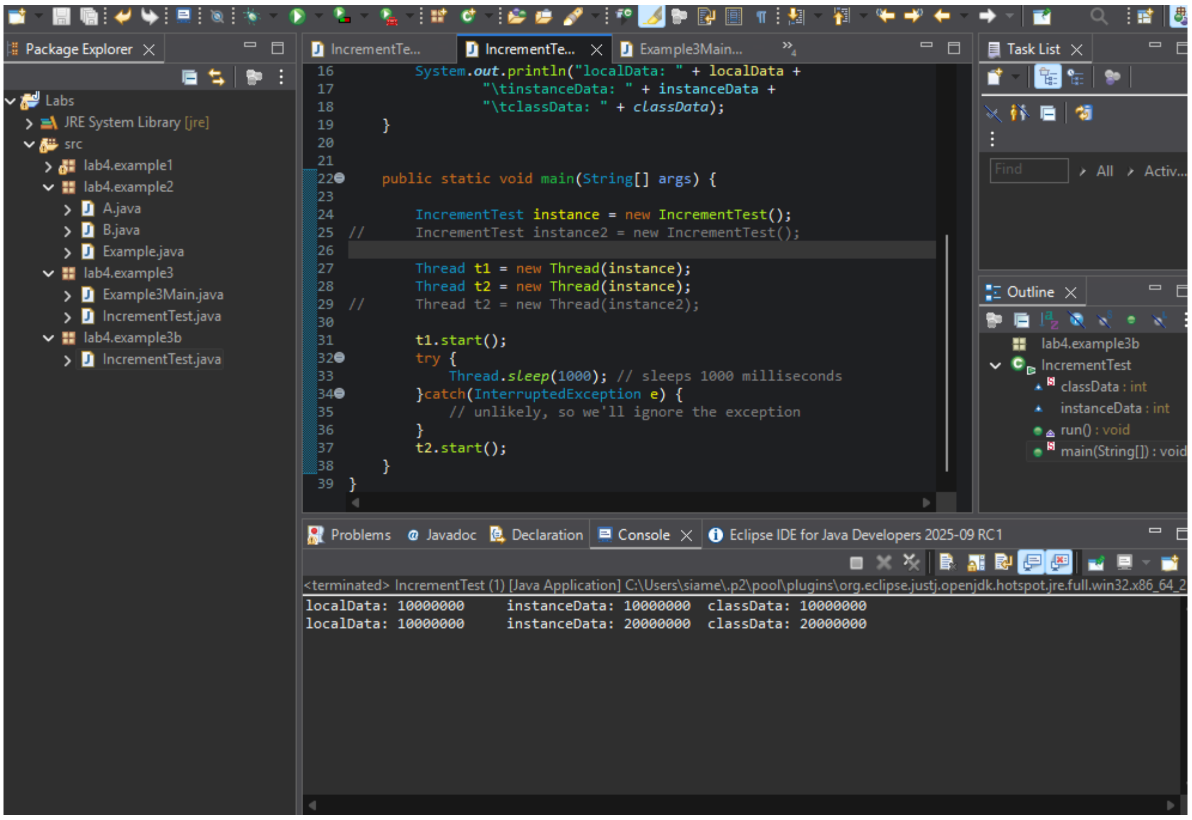


```java
                "\tclassData: " + classData);
    }

    public static void main(String[] args) {
        IncrementTest instance1 = new IncrementTest();
        IncrementTest instance2 = new IncrementTest();

        Thread t1 = new Thread(instance1, "T1");
        Thread t2 = new Thread(instance2, "T2");

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Both threads finished.");
    }
}
```

```
<terminated> IncrementTest [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8
T1 → localData: 10000000        instanceData: 10000000  classData: 10024443
T2 → localData: 10000000        instanceData: 10000000  classData: 169203
Both threads finished.
```

Figure 13. Example 3.2: Two Instances: Output showing instanceData isolated per object, but classData still shared and inconsistent

Figure 14

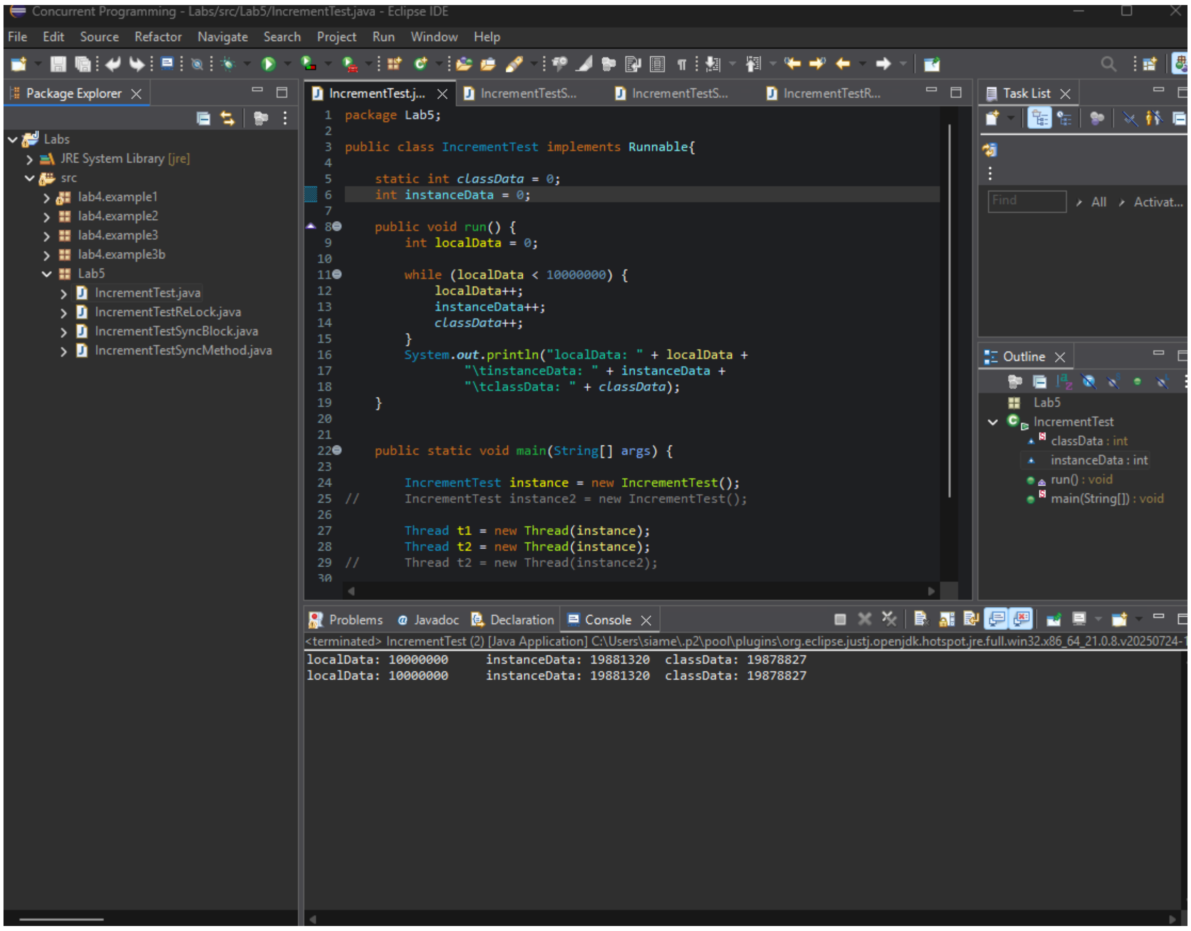


```java
        System.out.println("localData: " + localData +
                "\tinstanceData: " + instanceData +
                "\tclassData: " + classData);
    }


    public static void main(String[] args) {

        IncrementTest instance = new IncrementTest();
//      IncrementTest instance2 = new IncrementTest();

        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);
//      Thread t2 = new Thread(instance2);

        t1.start();
        try {
            Thread.sleep(1000); // sleeps 1000 milliseconds
        }catch(InterruptedException e) {
            // unlikely, so we'll ignore the exception
        }
        t2.start();
    }
}
```

```
<terminated> IncrementTest (1) [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_2
localData: 10000000     instanceData: 10000000  classData: 10000000
localData: 10000000     instanceData: 20000000  classData: 20000000
```

Figure 14. Example 3b: Staggered Start: Output showing 'perfect' totals for instanceDatqa and classData because threads ran sequentially, not concurrently

Figure 15

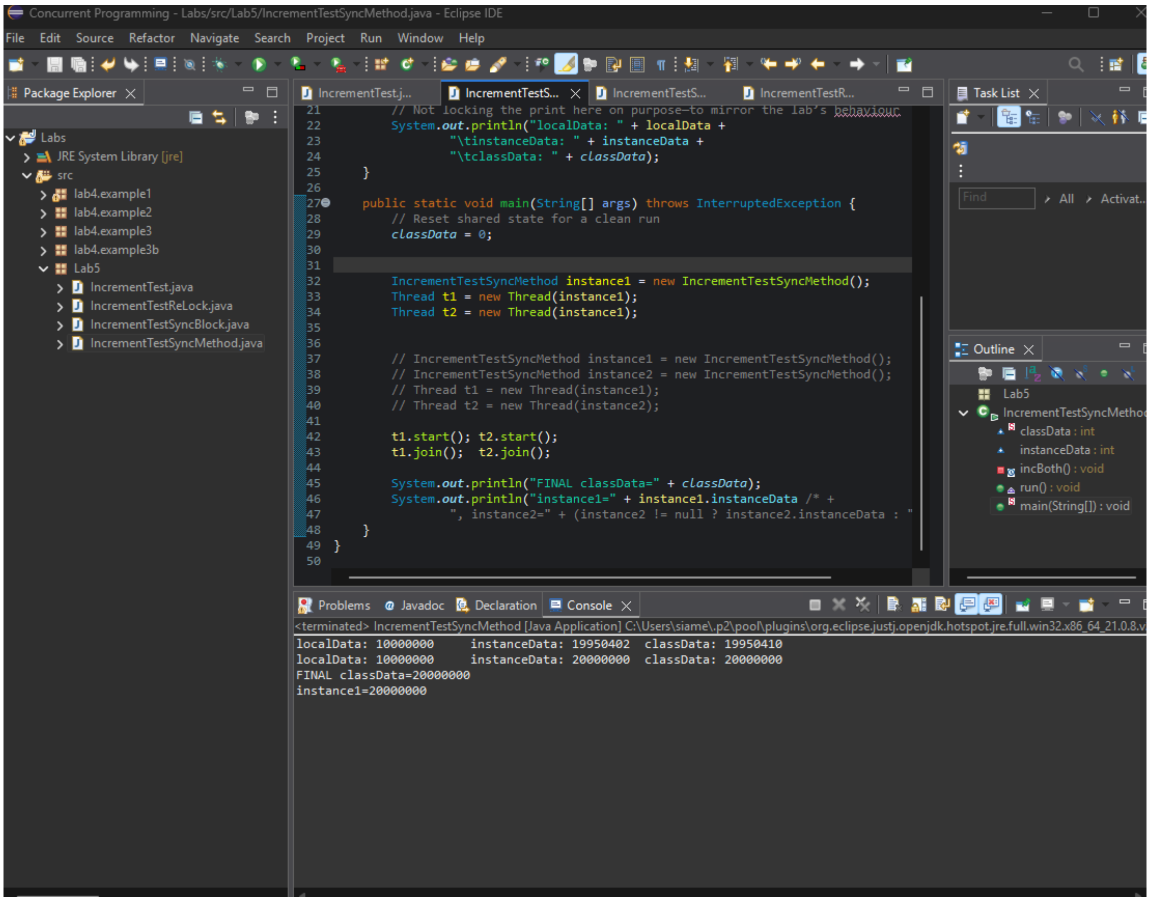


```java
package Lab5;

public class IncrementTest implements Runnable{

    static int classData = 0;
    int instanceData = 0;

    public void run() {
        int localData = 0;

        while (localData < 10000000) {
            localData++;
            instanceData++;
            classData++;
        }
        System.out.println("localData: " + localData +
                "\tinstanceData: " + instanceData +
                "\tclassData: " + classData);
    }


    public static void main(String[] args) {

        IncrementTest instance = new IncrementTest();
//      IncrementTest instance2 = new IncrementTest();

        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);
//      Thread t2 = new Thread(instance2);
```

```
<terminated> IncrementTest (2) [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8.v20250724-1
localData: 10000000     instanceData: 19881320  classData: 19878827
localData: 10000000     instanceData: 19881320  classData: 19878827
```

Figure 15. Console output of baseline run (showing inconsistent totals)

Figure 16

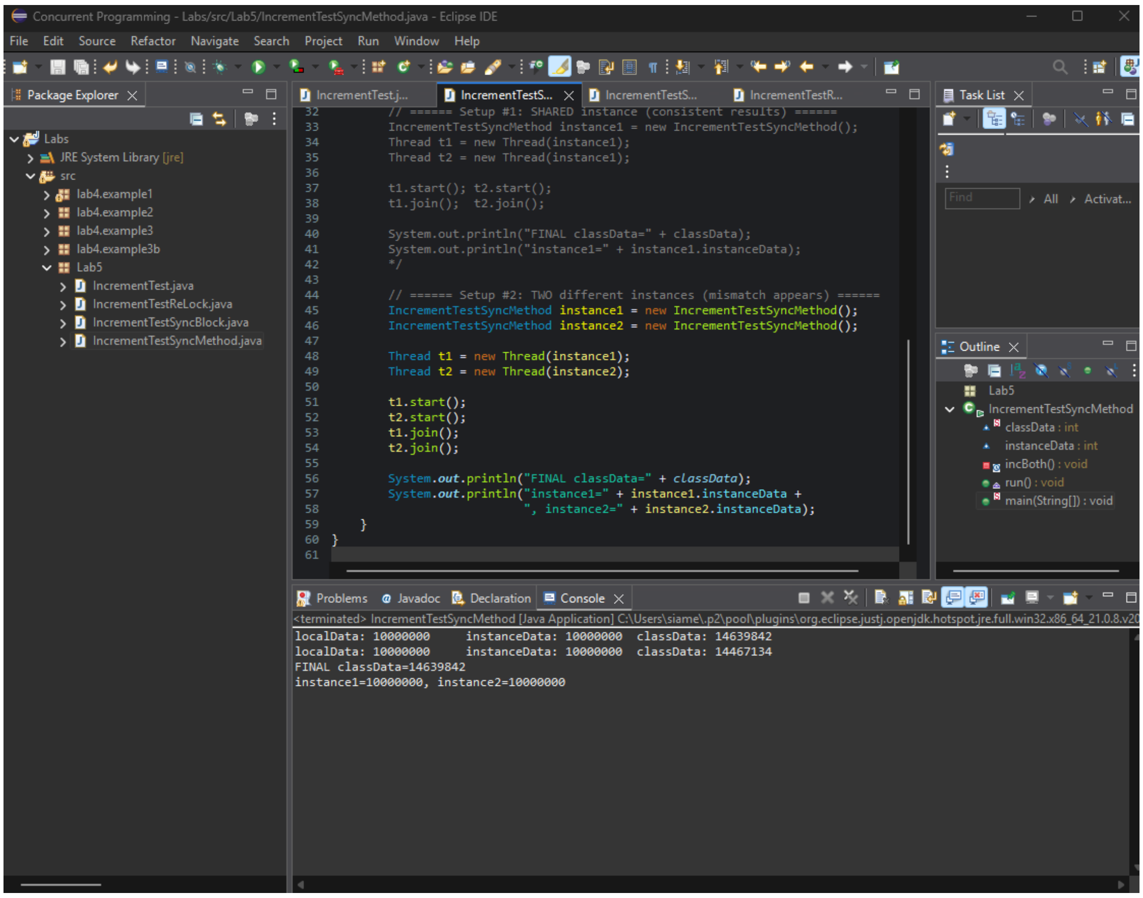


```java
        // Not locking the print here on purpose—to mirror the lab's behaviour
        System.out.println("localData: " + localData +
                "\tinstanceData: " + instanceData +
                "\tclassData: " + classData);
    }

    public static void main(String[] args) throws InterruptedException {
        // Reset shared state for a clean run
        classData = 0;


        IncrementTestSyncMethod instance1 = new IncrementTestSyncMethod();
        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance1);


        // IncrementTestSyncMethod instance1 = new IncrementTestSyncMethod();
        // IncrementTestSyncMethod instance2 = new IncrementTestSyncMethod();
        // Thread t1 = new Thread(instance1);
        // Thread t2 = new Thread(instance2);

        t1.start(); t2.start();
        t1.join();  t2.join();

        System.out.println("FINAL classData=" + classData);
        System.out.println("instance1=" + instance1.instanceData /* +
                ", instance2=" + (instance2 != null ? instance2.instanceData : "
    }
}
```

```
<terminated> IncrementTestSyncMethod [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8.v
localData: 10000000	instanceData: 19950402	classData: 19950410
localData: 10000000	instanceData: 20000000	classData: 20000000
FINAL classData=20000000
instance1=20000000
```

Figure 16. Shared instance: consistent results

Figure 17



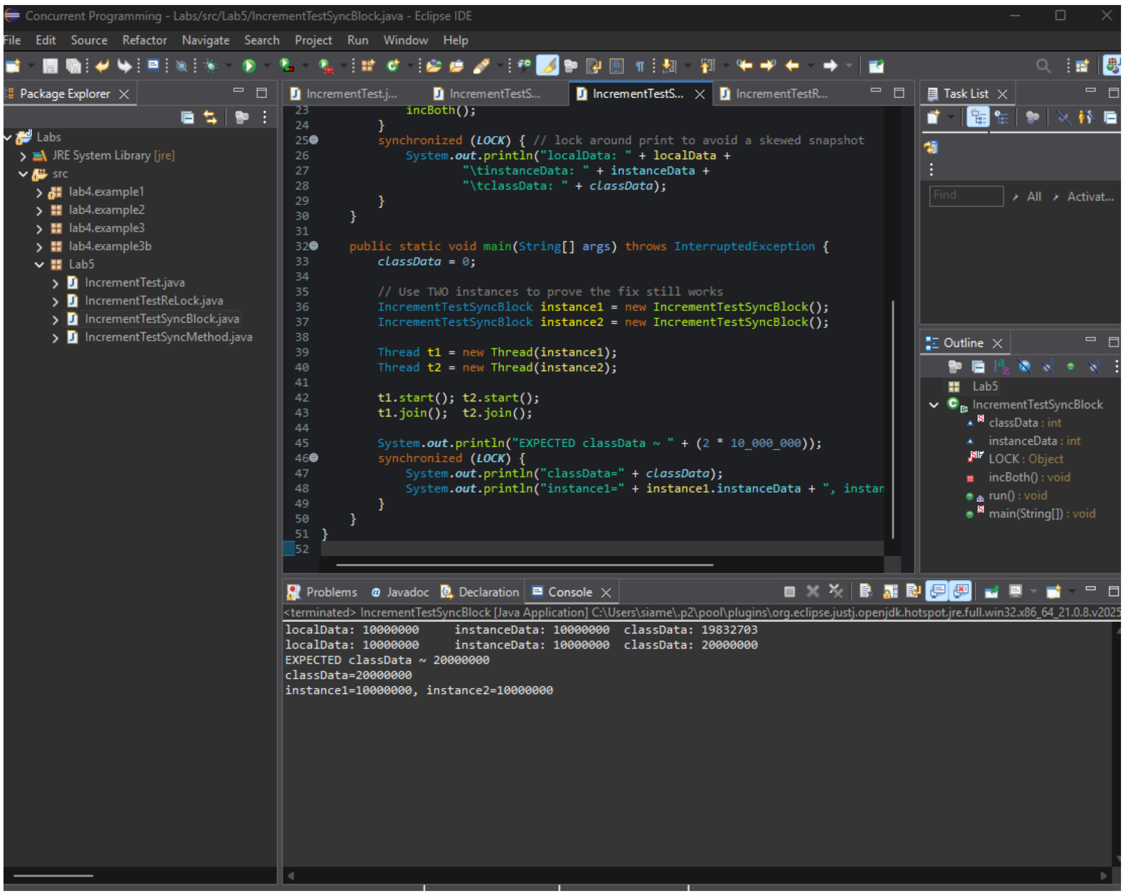
Concurrent Programming - Labs/src/Lab5/IncrementTestSyncMethod.java - Eclipse IDE

```java
        // ====== Setup #1: SHARED instance (consistent results) ======
        IncrementTestSyncMethod instance1 = new IncrementTestSyncMethod();
        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance1);

        t1.start(); t2.start();
        t1.join();  t2.join();

        System.out.println("FINAL classData=" + classData);
        System.out.println("instance1=" + instance1.instanceData);
        */

        // ====== Setup #2: TWO different instances (mismatch appears) ======
        IncrementTestSyncMethod instance1 = new IncrementTestSyncMethod();
        IncrementTestSyncMethod instance2 = new IncrementTestSyncMethod();

        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance2);

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("FINAL classData=" + classData);
        System.out.println("instance1=" + instance1.instanceData +
                           ", instance2=" + instance2.instanceData);
    }
}
```

```
<terminated> IncrementTestSyncMethod [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8.v20
localData: 10000000     instanceData: 10000000  classData: 14639842
localData: 10000000     instanceData: 10000000  classData: 14467134
FINAL classData=14639842
instance1=10000000, instance2=10000000
```

Figure 17. Two instances: classData>instanceData

Figure 18



Concurrent Programming - Labs/src/Lab5/IncrementTestSyncBlock.java - Eclipse IDE

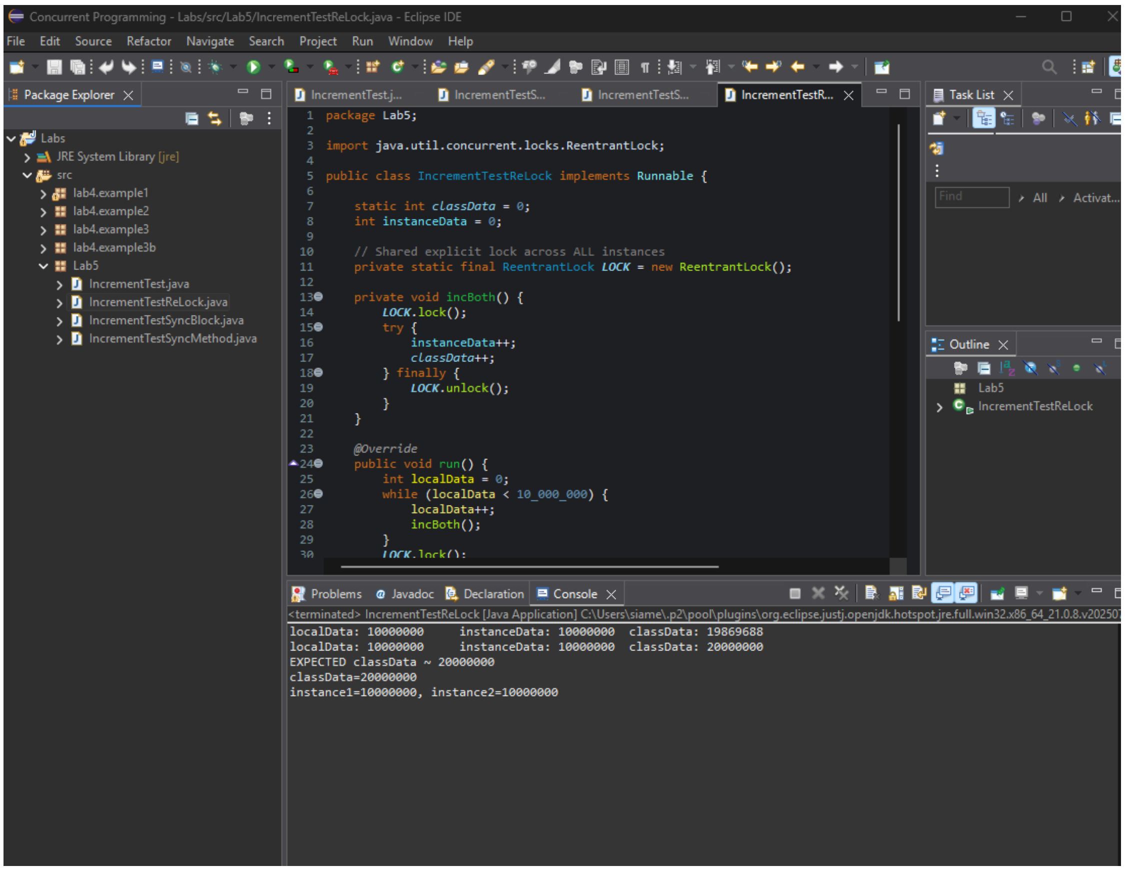
```java
            incBoth();
        }
        synchronized (LOCK) { // lock around print to avoid a skewed snapshot
            System.out.println("localData: " + localData +
                    "\tinstanceData: " + instanceData +
                    "\tclassData: " + classData);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        classData = 0;

        // Use TWO instances to prove the fix still works
        IncrementTestSyncBlock instance1 = new IncrementTestSyncBlock();
        IncrementTestSyncBlock instance2 = new IncrementTestSyncBlock();

        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance2);

        t1.start(); t2.start();
        t1.join();  t2.join();

        System.out.println("EXPECTED classData ~ " + (2 * 10_000_000));
        synchronized (LOCK) {
            System.out.println("classData=" + classData);
            System.out.println("instance1=" + instance1.instanceData + ", instar
        }
    }
}
```

<terminated> IncrementTestSyncBlock [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8.v2025

```
localData: 10000000     instanceData: 10000000  classData: 19832703
localData: 10000000     instanceData: 10000000  classData: 20000000
EXPECTED classData ~ 20000000
classData=20000000
instance1=10000000, instance2=10000000
```

Figure 18. Two instances with shared lock: consistent results

Figure 19

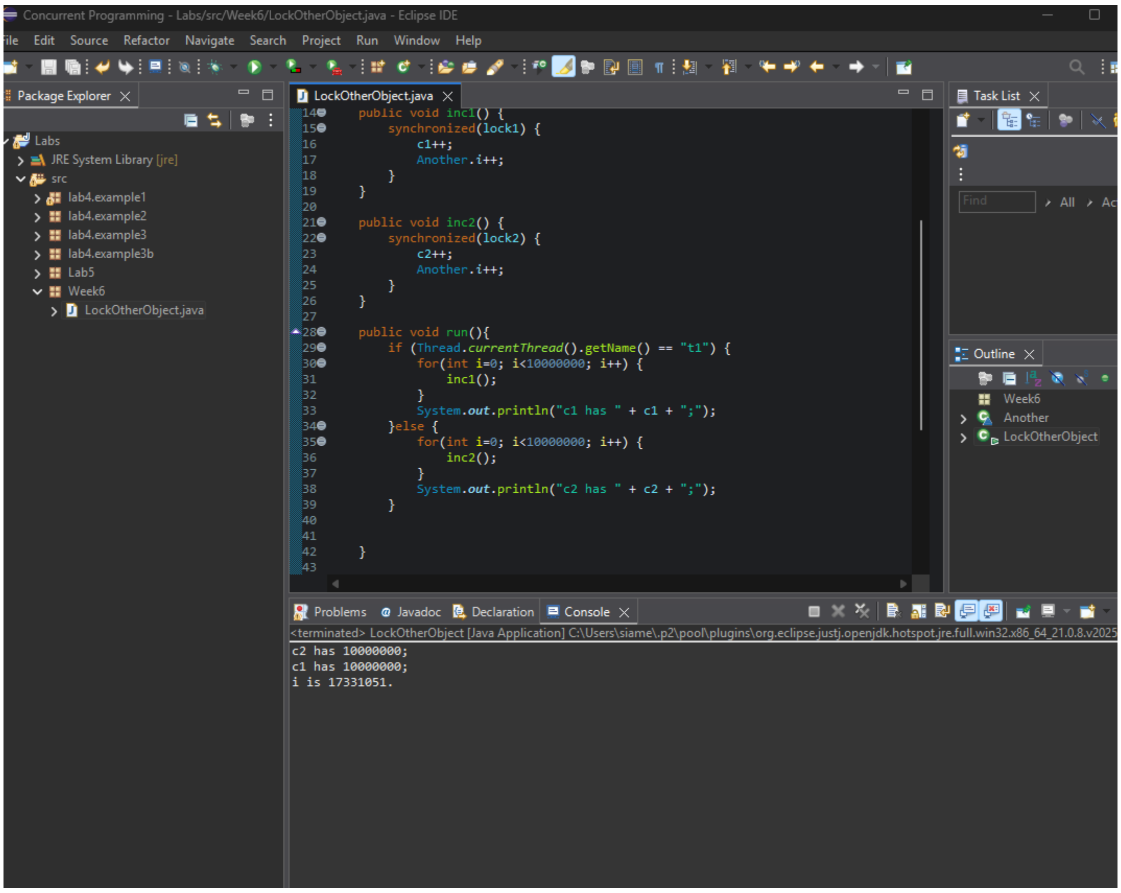


```java
package Lab5;

import java.util.concurrent.locks.ReentrantLock;

public class IncrementTestReLock implements Runnable {

    static int classData = 0;
    int instanceData = 0;

    // Shared explicit lock across ALL instances
    private static final ReentrantLock LOCK = new ReentrantLock();

    private void incBoth() {
        LOCK.lock();
        try {
            instanceData++;
            classData++;
        } finally {
            LOCK.unlock();
        }
    }

    @Override
    public void run() {
        int localData = 0;
        while (localData < 10_000_000) {
            localData++;
            incBoth();
        }
        LOCK.lock();
```

```
<terminated> IncrementTestReLock [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8.v20250
localData: 10000000     instanceData: 10000000  classData: 19869688
localData: 10000000     instanceData: 10000000  classData: 20000000
EXPECTED classData ~ 20000000
classData=20000000
instance1=10000000, instance2=10000000
```

Figure 19. Two instances with ReentrantLock: consistent results

Figure 20

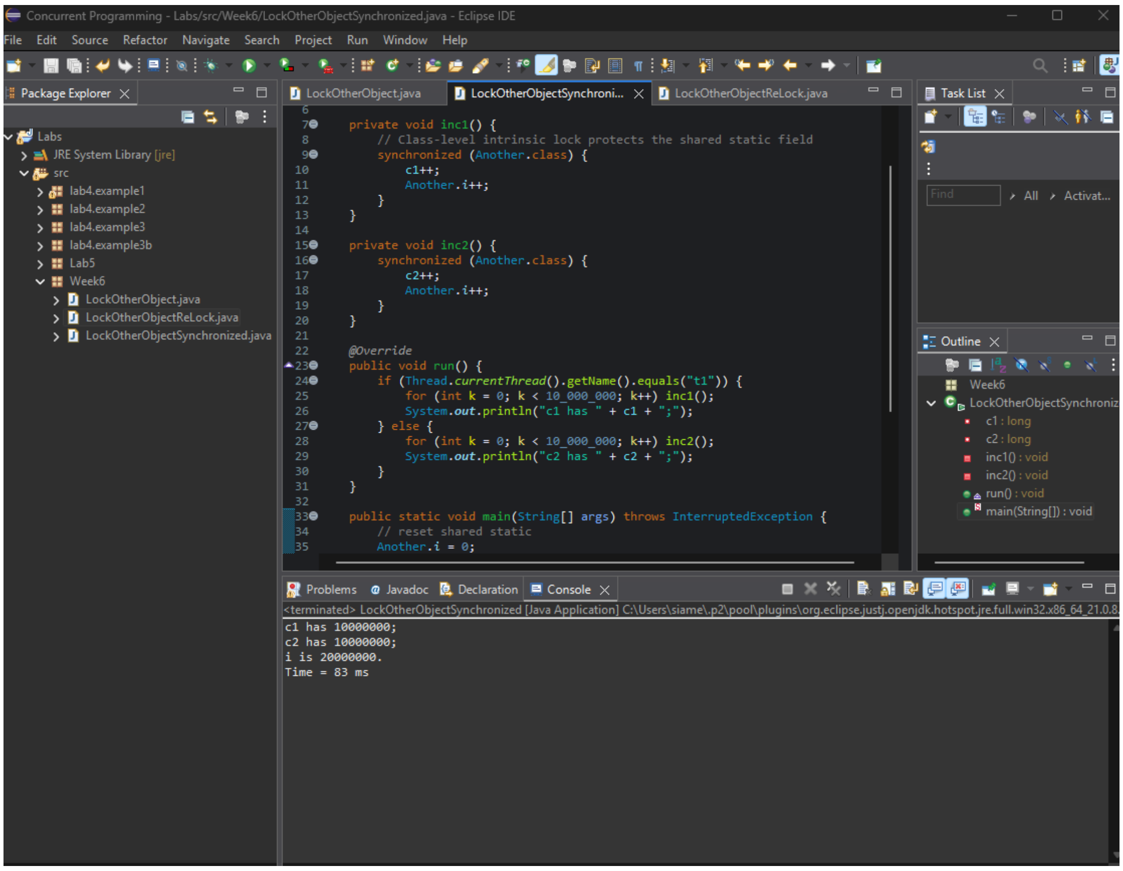


```java
    public void inc1() {
        synchronized(lock1) {
            c1++;
            Another.i++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
            Another.i++;
        }
    }

    public void run(){
        if (Thread.currentThread().getName() == "t1") {
            for(int i=0; i<10000000; i++) {
                inc1();
            }
            System.out.println("c1 has " + c1 + ";");
        }else {
            for(int i=0; i<10000000; i++) {
                inc2();
            }
            System.out.println("c2 has " + c2 + ";");
        }


    }
```

```
<terminated> LockOtherObject [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8.v2025
c2 has 10000000;
c1 has 10000000;
i is 17331051.
```

Figure 10. Baseline run showing actual<expected

Figure 21

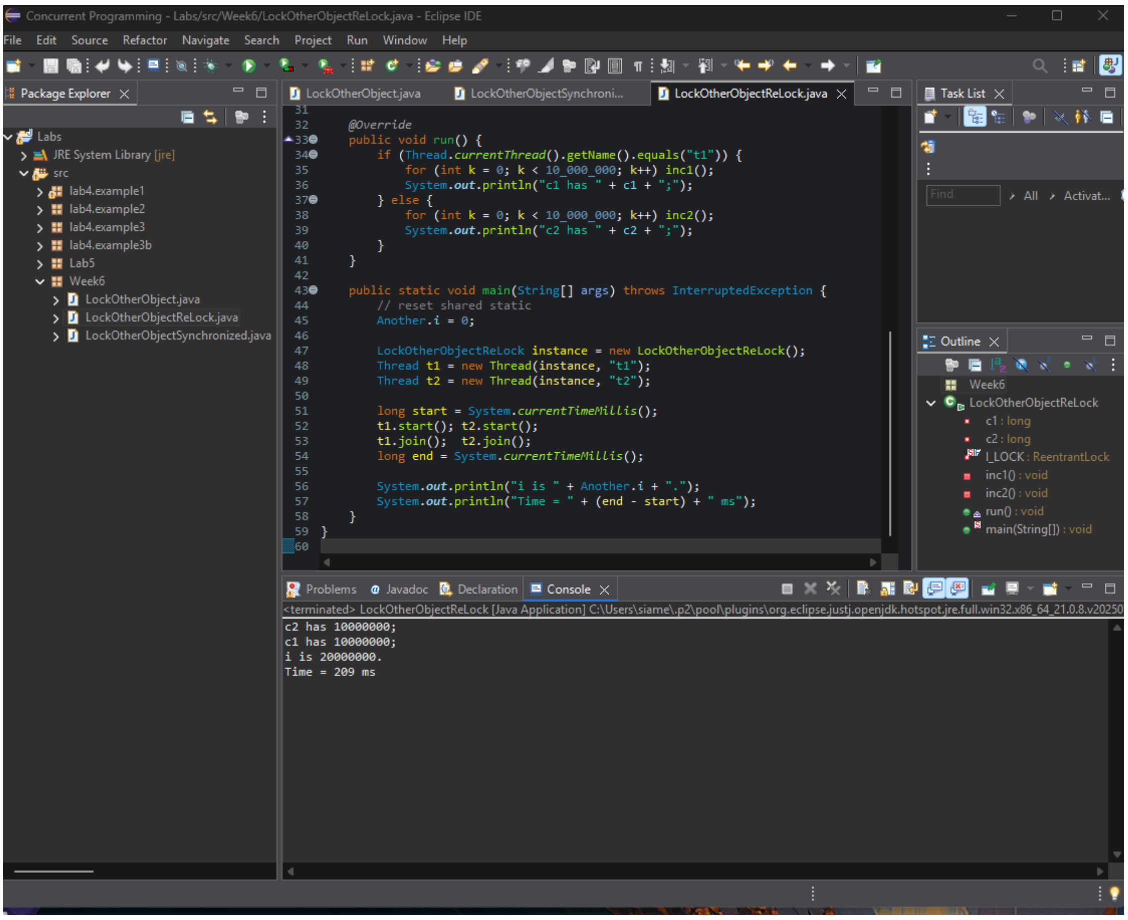


```java
    private void inc1() {
        // Class-level intrinsic lock protects the shared static field
        synchronized (Another.class) {
            c1++;
            Another.i++;
        }
    }

    private void inc2() {
        synchronized (Another.class) {
            c2++;
            Another.i++;
        }
    }

    @Override
    public void run() {
        if (Thread.currentThread().getName().equals("t1")) {
            for (int k = 0; k < 10_000_000; k++) inc1();
            System.out.println("c1 has " + c1 + ";");
        } else {
            for (int k = 0; k < 10_000_000; k++) inc2();
            System.out.println("c2 has " + c2 + ";");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // reset shared static
        Another.i = 0;
```

```
<terminated> LockOtherObjectSynchronized [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8
c1 has 10000000;
c2 has 10000000;
i is 20000000.
Time = 83 ms
```

Figure 11. Output with i is 20000000

Figure 22



```java
    @Override
    public void run() {
        if (Thread.currentThread().getName().equals("t1")) {
            for (int k = 0; k < 10_000_000; k++) inc1();
            System.out.println("c1 has " + c1 + ";");
        } else {
            for (int k = 0; k < 10_000_000; k++) inc2();
            System.out.println("c2 has " + c2 + ";");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // reset shared static
        Another.i = 0;

        LockOtherObjectReLock instance = new LockOtherObjectReLock();
        Thread t1 = new Thread(instance, "t1");
        Thread t2 = new Thread(instance, "t2");

        long start = System.currentTimeMillis();
        t1.start(); t2.start();
        t1.join();  t2.join();
        long end = System.currentTimeMillis();

        System.out.println("i is " + Another.i + ".");
        System.out.println("Time = " + (end - start) + " ms");
    }
}
```

```
<terminated> LockOtherObjectReLock [Java Application] C:\Users\siame\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.8.v20250
c2 has 10000000;
c1 has 10000000;
i is 20000000.
Time = 209 ms
```

Figure 12. Output with i is 20000000 using ReentrantLock

References

GeeksforGeeks. (2025, February 5). *Go - Concurrency and Parallelism*. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/go-language/go-concurrency-and-parallelism/

GeeksforGeeks. (2025, July 28). *Different Segments in C Program's Memory* [Image]. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/c/memory-layout-of-c-program/