Java - Concurrency Group 2

Comparison of the currency models (Procs & Cons)

The basic concept of Multithreaded Usage of Energy from the Battery

The BatteryUsageSimulator class in the Energy Management System simulates devices that consume energy from a central battery source, with each instance representing a distinct device with specific usage patterns. This class has attributes including battery (the shared energy source), usageAmount (the fixed energy each device consumes), device (the device's name for logging purposes), and isVariableUsage (a flag indicating whether the device's consumption is constant or variable). When isVariableUsage is true, the energy usage is randomized up to the maximum usageAmount, simulating real-world fluctuations.

The run method continuously checks battery availability and consumes energy, logging usage details. In the EnergyManagementSystemMain class, instances like highPowerDevice, lowPowerDevice, and variablePowerDevice are created to represent devices with different energy needs. For example, BatteryUsageSimulator highPowerDevice = new BatteryUsageSimulator(battery, 20, "HighPowerDevice", false); models a device with a set energy consumption, while variablePowerDevice consumes varying energy levels. Each instance runs concurrently to simulate multiple devices consuming energy from the shared battery.

In an energy management system with multiple devices, ExecutorService is the best concurrency model. It allows efficient handling of tasks by reusing threads from a pool, controlling the number of threads, and thus preventing overload.

Comparing Concurrency Models

Thread-Based Model

What it is: This is the basic way of handling concurrency, where each device is a separate thread created directly in the code.

Pros:

- Simple to set up and understand for a small number of devices.
- Offers control over each thread individually, which can be useful for basic tasks.

Cons:

- Not scalable for a large number of devices; each thread takes up memory and processing power.
- The more threads you have, the more difficult it is to manage them effectively.

- Increased risk of errors like memory leaks and deadlocks if threads aren't managed properly.

For our system: A thread-based model may work if there are only a few devices. But as the number of devices increases, it can become inefficient, slowing down the system and causing resource issues.

ExecutorService (Thread Pool)

What it is: Instead of creating a new thread for each device, an ExecutorService manages a pool of reusable threads. Devices submit tasks to this pool, and the pool assigns threads as needed.

Pros:

- Efficient resource usage because threads are reused instead of constantly creating and destroying them.
- Limits the number of threads running at once, preventing system overload.
- Simplifies the management of thread lifecycle, as ExecutorService handles creation, execution, and termination of threads.

Cons:

- Slightly more complex to set up, requiring configuration of the pool size.
- May have some setup overhead, but it's manageable compared to the benefits.

For our system: ExecutorService is ideal because it allows us to control the number of active devices drawing energy, making sure the system doesn't slow down or crash. By setting up a fixed number of threads, we prevent overloading the CPU or draining the battery too quickly.

3. Fork-Join Model

What it is: Fork-Join splits tasks into smaller sub-tasks that can be processed in parallel. It's mainly used for tasks that can be divided and then recombined (like sorting or data analysis).

Pros:

- Very efficient for parallel tasks that can be split and executed independently.
- Features work-stealing, where idle threads take on more tasks, maximizing CPU usage.

Cons:

- Not suitable for independent tasks like ours where each device is just drawing energy without depending on others.
- Adds unnecessary complexity for tasks that don't require splitting.

For our system: Fork-Join doesn’t fit well, as our energy usage simulation doesn’t benefit from splitting tasks. Each device is independent, so Fork-Join’s parallel processing features would be wasted.

Differences between Concurrency vs Parallelism

Concurrency and parallelism are related concepts in computing, but they refer to different ways of managing tasks, especially in the context of multitasking and multithreading.

1. Concurrency

Concurrency is about **dealing with multiple tasks at the same time**. It refers to the ability of a system to handle multiple tasks, making progress on each without necessarily completing them one after the other. In concurrency, tasks can start, run, and complete in overlapping time periods, but they don't necessarily have to run at the exact same moment.

Key Characteristics of Concurrency:

- **Interleaving Tasks:** Concurrency allows multiple tasks to make progress by interleaving their execution, often switching between them quickly (time-slicing).
- **Context Switching:** Concurrency often involves switching between tasks in a way that makes it appear as though they are running simultaneously.
- **Shared Resources:** In a concurrent system, multiple tasks may share resources like CPU time or memory.
- **Example:** A single-core CPU can switch between tasks, giving the appearance that multiple tasks are being handled simultaneously, even though only one task is executed at any instant.

Use Cases: Concurrency is useful in applications that need to handle multiple tasks but don’t necessarily require them to run in perfect synchronization, such as handling user input and background tasks in a graphical user interface (GUI) or managing network requests in a server.

2. Parallelism

Parallelism is about **actually performing multiple tasks at the same time**. It requires hardware with multiple processing units (such as multiple CPU cores or GPUs), where each unit can handle a different task simultaneously. Parallelism aims to increase computational speed and efficiency by distributing tasks across multiple processors or cores.

Key Characteristics of Parallelism:

- **Simultaneous Execution:** Multiple tasks are executed simultaneously, often on separate processing units.
- **No Context Switching:** Parallelism does not require context switching since each task has its own processor or core.

- **Independent Execution:** Tasks are often independent, with minimal need for communication or coordination between them.
- **Example:** A multi-core CPU or GPU processing different parts of a dataset at the same time, where each core independently computes its part of the workload.

Use Cases: Parallelism is commonly used in high-performance computing (HPC) applications, scientific simulations, large data processing, machine learning, and any computational tasks that can be split into independent units.

Key Differences Between Concurrency and Parallelism

Aspect	Concurrency	Parallelism
Definition	Managing multiple tasks by switching between them or interleaving them	Performing multiple tasks simultaneously
Hardware Requirement	Can be done on single-core processors	Requires multi-core or multiple processors
Execution	Tasks are run in overlapping time periods but not necessarily at the same time	No context switching; each task has its own core
Context Switching	Involves context switching between tasks	No context switching; each task has its own core
Task Dependency	Often used for tasks that are interdependent or require coordination	Often used for independent tasks
Efficiency Goal	Improves responsiveness and resource utilization	Improves computational speed and processing power
Examples	Web servers, GUIs, handling multiple network requests	Scientific computing, image processing, data analysis

Blocking Concurrency Algorithms

Blocking concurrency algorithms prevent access to a shared resource until a thread or process is finished with it, ensuring data consistency. When a thread encounters a "lock," it must wait or "block" until the resource is free.

Characteristics:

- **Mutual Exclusion**: Only one thread can access the shared resource at a time.
- **Synchronization Primitives**: Uses locks, semaphores, and monitors to ensure safe access.
- **Performance**: Can reduce system performance if multiple threads frequently contend for the same resource.
- **Deadlocks**: If not handled carefully, blocking algorithms can lead to deadlocks, where two or more threads are waiting indefinitely for each other's resources.

Examples:

1. **Mutexes** and **Locks**: Only one thread can own the lock and access the resource. Other threads block until the lock is released.
2. **Synchronized Blocks** in Java: Allows only one thread to execute within a synchronized block of code at a time.
3. **Semaphores**: Allow a certain number of threads to access the resource at once, blocking the others.

Use Cases:

- **Database Access**: Where strict data consistency is critical, like updating bank account balances.
- **File I/O**: When handling sensitive data or writing to logs to prevent data corruption.
- **Transactional Systems**: Systems that rely on atomic, all-or-nothing transactions.

Non-Blocking Concurrency Algorithms

Non-blocking concurrency algorithms allow multiple threads to operate on shared data without locking it entirely. Instead of waiting, a thread can use techniques to proceed without being blocked, often by retrying or leveraging atomic operations.

Characteristics:

- **Optimistic Concurrency**: Threads proceed with operations assuming they won't conflict. If they do, they handle it by retrying or aborting.
- **Atomic Operations**: Uses atomic operations (e.g., compare-and-swap or CAS) to achieve thread safety without locks.
- **Reduced Waiting**: As threads are not forced to wait for a lock, they tend to reduce latency and increase throughput.
- **Complexity**: Non-blocking algorithms can be more complex and are typically challenging to design and debug.

Examples:

1. **Atomic Variables**: Classes like AtomicInteger and AtomicReference in Java offer atomic operations that do not require locks.
2. **Compare-And-Swap (CAS)**: A thread updates a shared resource only if its current state matches an expected state.
3. **Lock-Free Data Structures**: Structures like lock-free stacks and queues that don't use locks for data access.

Use Cases:

- **Real-Time Systems**: Where responsiveness is critical, like user interface updates or embedded systems.
- **High-Performance Computing**: Non-blocking algorithms are often used in multi-core processors to maximize performance.
- **Scalable Web Servers**: To reduce contention and improve throughput under heavy load.