Provide a comparison of the concurrency models (Procs & Cons)

Answer

The application, modeled around `Battery`, `EnergySource`, `ChargingStation`, and `UserInput`, uses Java threads and the `ExecutorService` for concurrent execution. Each class has a distinct role: `Battery` manages power storage, `EnergySource` produces power, `ChargingStation` consumes power, and `UserInput` interacts with the user, creating and managing these entities. Below is a comparison of concurrency models with respect to these classes:

---

1. Thread-based Concurrency

In this model, each `ChargingStation` and `EnergySource` instance runs in its own thread. `UserInput` uses `ExecutorService` to manage thread execution for these classes.

Pros:

- **Simplicity**: Easy to understand and implement, especially for individual tasks like charging or draining a battery.
- **Isolation**: Each thread can execute independently, suitable for tasks that don't share state.
- **Direct Execution Control**: Directly starts and stops threads, useful for simulation loops in `ChargingStation` and `EnergySource`.

Cons:

- **Resource Intensive**: Each thread has memory overhead, and too many threads can cause excessive CPU context-switching, which may degrade performance.
- **Shared State Issues**: `Battery` uses `synchronized` methods to prevent concurrent modification of `currentPower`, but this may lead to bottlenecks, especially under high load.
- **Difficulty in Scaling**: Creating multiple instances of `ChargingStation` or `EnergySource` each requires new threads, which can be challenging to manage.

---

2. ExecutorService (Thread Pool)

`UserInput` uses `ExecutorService` to manage threads for `ChargingStation` and `EnergySource` instances, allowing thread reuse.

Pros:

- **Efficient Resource Management**: By reusing threads, `ExecutorService` minimizes memory and CPU usage, reducing the impact of creating multiple `ChargingStation` and `EnergySource` instances.
- **Control Over Thread Lifecycle**: Automatically manages threads, simplifying shutdown and error handling.
- **Scalability**: More flexible than individual threads as it can dynamically adjust the number of threads based on workload.

Cons:

- **Complex Error Handling**: Handling exceptions and errors across tasks within a pool is more complex, as seen in the `dataExchange` method, which must manage multiple exceptions.
- **Limited Customization**: Thread pools limit control over individual task execution, which could be problematic in real-time adjustments for power production/consumption.
- **Risk of Resource Starvation**: A busy pool may delay tasks if all threads are occupied, potentially impacting real-time needs for charging or draining the `Battery`.

---

3. Synchronous Approach (Within Methods)

The `synchronized` keyword in `Battery` methods `charge` and `drain` ensures only one thread can modify `currentPower` at a time.

Pros:

- **Data Consistency**: Prevents race conditions, ensuring consistent `currentPower` value.
- **Simplicity**: Direct, clear way to protect critical sections, making it easier to understand shared resource management.

Cons:

- **Reduced Performance**: Threads are blocked while waiting, which can lead to slower performance if multiple `ChargingStation` or `EnergySource` instances frequently access `Battery`.
- **Scalability Issues**: With more concurrent users, the blocking nature of `synchronized` methods may become a bottleneck, limiting scalability.

Explain differences between Concurrency vs Parallelism

Answer

Concurrency and parallelism are two fundamental concepts in computing, often used in the context of multi-threaded programming to improve the efficiency of programs. Here's a breakdown of each and how they apply to the following classes (`Battery`, `EnergySource`, `ChargingStation`, and `UserInput`).

1. Concurrency

Concurrency is the concept of managing multiple tasks at once within a program, even if not executed simultaneously. It focuses on the structure of handling multiple operations by interleaving execution, usually within the same CPU core. Concurrency enables a program to make progress on several tasks by switching between them, managing resources and tasks efficiently.

In the context of the provided classes:

- **`Battery` Synchronization**: The `charge` and `drain` methods in the `Battery` class are marked as `synchronized`. This synchronization ensures that only one thread at a time can modify the battery's power level, allowing safe access when multiple threads are charging or draining power concurrently.

- **`ChargingStation` and `EnergySource` Threads**: In `UserInput`, `ChargingStation` and `EnergySource` objects are submitted to an `ExecutorService` for concurrent execution. Each of these objects will run in a

separate thread but may interleave execution depending on the availability of CPU cores and resources.

2. Parallelism

Parallelism, on the other hand, is about performing multiple tasks simultaneously. It requires multiple CPU cores so that tasks can run at the same time rather than being interleaved. Parallelism is ideal for computations or operations that can truly run independently.

In this example:

- **Independent `ChargingStation` and `EnergySource` Operations**: With an executor service, if multiple CPU cores are available, `ChargingStation` and `EnergySource` instances could truly operate in parallel—charging and draining the battery simultaneously without waiting on each other.
- **CPU Cores and Thread Execution**: If the system has multiple cores, `ChargingStation` and `EnergySource` can leverage true parallelism by running on separate cores, allowing the battery's power level to be updated by both concurrently.

Practical Differences in These classes in the codebase

- **Concurrency** enables the `Battery` class to handle charging and draining requests without corrupting data, thanks to the `synchronized` keyword.
- **Parallelism** allows `ChargingStation` and `EnergySource` objects to run independently on separate threads, potentially on different CPU cores, if the system.

Explain the usage of Blocking Concurrency Algorithms and Non-blocking Concurrency Algorithms

Blocking and **Non-blocking concurrency algorithms** play a significant role in managing concurrent access to shared resources (like the `Battery` object) by multiple threads, as seen in the `EnergySource` and `ChargingStation` classes. Here's how these concepts apply:

1. Blocking Concurrency Algorithms

Blocking algorithms use locks or synchronization techniques to manage access to shared resources, ensuring only one thread can operate on a critical section at a time. This approach helps prevent issues like race conditions, but it can lead to performance bottlenecks if threads spend time waiting for locks. In the provided code:

- The `Battery` class uses **synchronized methods** (`charge`, `drain`, `getCurrentPower`). This is a **blocking approach** because when one thread is in a synchronized method (e.g., `charge`), any other thread attempting to enter a synchronized method on the same `Battery` object will be blocked until the first thread finishes.
- Blocking algorithms like this can reduce the chance of inconsistencies (e.g., overcharging or undercharging) but may limit parallelism. This approach ensures that each operation on the `Battery`'s `currentPower` happens in a predictable order, avoiding simultaneous access issues.

2. Non-blocking Concurrency Algorithms

Non-blocking algorithms allow multiple threads to operate on shared resources without requiring locks. Instead, they typically rely on atomic operations, making them more performant in high-contention scenarios. However, non-blocking algorithms can be complex to implement correctly.

- If a **non-blocking algorithm** were used here, it would allow `EnergySource` and `ChargingStation` to attempt concurrent `charge` and `drain` operations on the `Battery` without waiting. This could use Java's `AtomicDouble` for `currentPower`, leveraging atomic operations to ensure thread-safe updates without explicit locks.
- Non-blocking approaches are more efficient when threads perform small, frequent operations on shared resources, as it avoids the overhead of locking, but in this case, `synchronized` methods help keep `Battery` usage simpler and safe.

Applying the Concepts in the Codebase (Battery.java)

If there's a need for increased performance and reduced waiting, converting to a non-blocking algorithm would involve:

- Replacing `synchronized` methods with atomic data types or carefully managing access with `java.util.concurrent` classes like `AtomicDouble` or `StamperReference` to avoid data inconsistencies while supporting concurrency.

Conclusion

For the `Battery` class's charging and draining operations, blocking (using synchronized methods) offers simpler, reliable concurrency management by allowing one thread at a time to update `currentPower`. Non-blocking concurrency, while potentially more efficient, would require atomic data structures or additional logic to manage simultaneous access correctly.