ISO/IEC/JTC 1/SC 22/WG 23 N1489
4 June 2025

Discussion of Java Concurrency, compared to Python
Submitted by Sean McDonagh

Recap of Python ...

Python and Java have *different* approaches to concurrency due to their underlying architectures. Thus, we will not be able to structure the vulnerability avoidance sections for concurrency as we did for Python.

It's essential to distinguish between *concurrency* and *parallelism*:

- **Concurrency** implies **dealing with** many things at once.
- **Parallelism** implies **doing** many things at once.

In Python, "concurrency" can be achieved through threading, multiprocessing, and asynchronous programming. While threading deals with concurrent execution, multiprocessing bypasses the GIL and achieves parallelism by using separate memory spaces.

Here's a breakdown for Python:

Attribute	Threading	Multiprocessing	Asyncio
Concurrency / Parallelism	• Achieve **concurrency** by running multiple threads within a single process. • Share the same memory space. • Use preemptive multitasking. The operating system scheduler interrupts running threads to allow others to execute • Limited by the GIL for CPU-bound tasks.	• Achieve **parallelism** on multi-core systems by creating multiple processes, each with its own memory space and Python interpreter.	• Achieve **concurrency** using a single thread and an event loop. • Coroutines (defined with async/await) are executed cooperatively (cooperative multitasking), yielding control to the event loop when waiting. • Multiple tasks can be managed concurrently, but they don't run in parallel.

Attribute	Threading	Multiprocessing	Asyncio
Global Interpreter Lock (**GIL)**	• Python's GIL allows only one thread to execute Python bytecode at a time, limiting true parallelism for CPU-bound tasks. • Can run truly in parallel on multi-core processors. • Each thread has its own call stack and resources.	• Each process has its own GIL, so CPU-bound tasks can run in parallel, utilizing multiple CPU cores.	• Tasks run sequentially within a single thread, so it is not affected by the GIL. • Tasks share the same stack and have lower memory overhead compared to threads.
Use Cases	• Ideal for I/O-bound tasks (e.g., waiting for network requests), where threads spend time waiting, and the GIL doesn't significantly hinder performance (not a bottleneck).	• Ideal for CPU-bound tasks (e.g., numerical computations, image processing) where true parallelism is needed.	• Best suited for I/O-bound tasks where there is a lot of waiting (e.g., handling multiple network requests, database queries). It's designed for high concurrency with minimal blocking.
Overhead	• Lower overhead compared to multiprocessing, as threads share memory. • Threads consume more memory due to their individual stacks	• Higher overhead compared to threading because processes are heavier and require inter-process communication (IPC) for data sharing.	• Lower overhead than threading and multiprocessing because it avoids context switching.
Complexity	• Relatively straightforward to implement but can introduce complexities like race conditions and deadlocks	• More complex than threading due to the need for IPC.	• Can have a steeper learning curve because it requires a different programming style. • Provides a more structured approach to concurrency, reducing the risk of race conditions.

In summary …

Concurrency in Python:

- **Threading**: The threading module is great for I/O-bound tasks but doesn't give true parallelism due to the GIL. Think of it as multitasking but with a bit of a bottleneck.
- **Asyncio**: The asyncio module is perfect for handling lots of I/O-bound tasks at once without the overhead of threading. It uses an event loop to switch between tasks, making it efficient.

Parallelism in Python:

- **Multiprocessing**: This module creates separate memory spaces and bypasses the GIL, making it ideal for CPU-bound tasks. Each process runs independently, so the application can truly run tasks in parallel.

The following pages are suggestions for the various sections in the Java document.

5.1.x Concurrency

Java excels in both concurrency and parallelism, but there are many scenarios that can be fairly complicated. Attempting to cover all possible concurrency scenarios, and their vulnerabilities, is beyond the scope of this document.

It is essential to distinguish between concurrency and parallelism. This document uses the general term "concurrency" to include both Concurrency and Parallelism as follows:

- Concurrency - Involves **handling** many things at once.

  Concurrency refers to the ability of a program to handle multiple tasks *seemingly simultaneously*, even if they are not truly running at the exact same time. This is achieved through techniques like time-slicing, where the CPU quickly switches between different tasks.

- Parallelism involves **doing** many things at once.

  Parallelism refers to the actual *simultaneous* execution of multiple tasks on multiple CPU cores or processing units.

Concurrency in Java:

- **Threading**: Java has built-in support for threading. The `Thread` class and the `java.util.concurrent` package facilitate the creation and management of multiple threads. Each thread can execute a different part of the program concurrently.

- **Executors**: The Executors framework enhances thread handling by automating the management of worker threads reducing the burden on the programmer. The Executors framework is primarily designed to facilitate concurrency, not necessarily parallelism. While the Executors framework can be used to achieve parallelism, its core functionality lies in managing concurrent execution of tasks through the efficient use of thread pools. Executors, particularly through the `ExecutorService` interface, manage a pool of threads, allowing for the reuse of threads instead of creating a new thread for every task. This significantly improves performance and resource utilization. The `ExecutorService` interface can track the progress and results of submitted tasks using `Future` objects, giving the programmer the ability to monitor and handle the outcomes of concurrent operations.

Parallelism in Java:

- **Fork/Join Framework**: The `ForkJoinPool` is a specialized implementation of the `ExecutorService` interface and facilitates dividing tasks into smaller subtasks that can be executed in parallel and then combining the results. This is especially useful when dealing with a large number of small tasks. It uses a work-stealing algorithm to efficiently distribute tasks across threads. Work stealing was introduced in Java 7 with the aim of reducing contention in multi-threaded applications.

6.59 Concurrency – Activation [CGA]

Java's concurrency be generally categorized into either the "Concurrency Model" or "Parallelism Model" (reference 5.1.x Concurrency).

The potential vulnerabilities for each concurrency model are described in the following sections:

- 6.59 Concurrency – Activation [CGA]
- 6.60 Concurrency – Directed termination [CGT])
- 6.61 Concurrent data access [CGX]
- 6.62 Concurrency – Premature termination [CGS]

Concurrency Model

Threading:

Here are some common problems related to thread activation in Java and their mitigations:

1. Race Conditions:

Problem: Multiple threads attempt to access and modify shared data simultaneously, leading to unpredictable and inconsistent results.

Mitigations:

- Synchronization: Use keywords like `synchronized` or Lock objects to ensure only one thread can access the critical section (code that modifies shared resources) at a time.
- Atomic Operations: Use atomic variables like `AtomicInteger` or `AtomicReference` for simple, atomic operations on shared data without explicit synchronization.
- Thread-Safe Collections: Use concurrent collections like `ConcurrentHashMap` or `CopyOnWriteArrayList` to handle shared data structures safely.
- Immutability: Design objects as immutable when possible to prevent race conditions by making them unmodifiable once created.

2. Deadlocks:

Problem: Two or more threads are blocked indefinitely, each waiting for a resource held by another thread.

Mitigations:

- Avoid Nested Locks: Refrain from having threads acquire locks in a circular order.
- Use `try-lock`: Employ `tryLock`() to attempt to acquire a lock without blocking indefinitely.
- Resource Ordering: Establish a consistent order for acquiring locks to prevent circular waiting.

3. Thread Starvation:

Problem: A thread is unable to make progress because other threads repeatedly acquire resources, leaving it "starved".

Mitigations:

- Thread Pools: Use thread pools to manage threads efficiently and ensure fair access to resources.
- Avoid Long-Running Tasks: Break down lengthy or blocking operations to prevent them from dominating resources.
- Careful Synchronization: Optimize the scope and duration of locks to minimize contention.

- Thread Priorities: Use thread priorities judiciously, but avoid relying solely on them as they may not be guaranteed.
- Monitoring and Diagnosis: Use tools like `jstack` or profilers to identify and troubleshoot thread starvation issues.

4. Creating Too Many Threads:

Problem: Excessive thread creation can lead to high resource consumption and performance degradation.

Mitigation:
- Use thread pools like those provided by `ExecutorService` to manage threads efficiently.

5. Ignoring Exception Handling:

Problem: Uncaught exceptions in threads can cause unpredictable application behavior.

Mitigation:
- Handle exceptions within threads properly to prevent unexpected termination.

6. Using Non-Thread-Safe Objects:

Problem: Modifying objects not designed for concurrent access from multiple threads can lead to inconsistent data.

Mitigation:
- Use thread-safe alternatives like `CopyOnWriteArrayList` or `Collections.synchronizedList`.

7. Improperly Stopping Threads:

Problem: Using deprecated methods like `Thread.stop`() can leave the application in an unpredictable state.

Mitigation:
- Design threads to be interruptible by checking an interrupt flag and exiting their `run`() method gracefully.

8. Calling `run`() instead of `start`():

Problem: Invoking the `run`() method directly executes the code in the current thread, not a new one.

Mitigation:

- Always call the `start`() method to create and launch a new thread that will execute the `run`() method.

9. Attempting to restart a stopped thread:

Problem: Once a thread has finished execution, it cannot be restarted.

Mitigation:

- Create a new thread instance to execute the desired code.

Key Concepts:

- Synchronization: A mechanism to control access to shared resources by multiple threads.
- Critical Section: A portion of code that must be executed by only one thread at a time.
- Atomic Operation: An operation that completes in a single, indivisible step, ensuring data integrity.
- Thread-Safe: Describes code or data structures that are free of race conditions when accessed by multiple threads.
- Thread Pool: A collection of reusable threads that can execute tasks, reducing the overhead of thread creation.

By understanding and addressing these common threading problems and applying the appropriate solutions, developers can write robust and efficient multithreaded applications in Java.

Executors:

Here are some common problems encountered with Java Executors during activation, and their mitigations:

1. Resource Management:

Problem: Creating too many threads can lead to excessive resource consumption and performance degradation.

Mitigation:

- Use `ThreadPoolExecutor` with a fixed or cached thread pool. This allows the programmer to control the number of threads and reuse them for multiple tasks.

2. Task Submission:

Problem: Tasks may fail to execute due to exceptions.

Mitigation:

- Use `Callable` instead of `Runnable` to handle exceptions in tasks, or use `ManagedTask` and `ManagedTaskListener`. Additionally, consider using a dynamic proxy to achieve the desired behavior.

3. Shutdown Issues:

Problem: Threads may not terminate properly after the executor is shut down (also applies to **6.62 Concurrency - Premature Termination**).

Mitigation:

- Call `shutdown`() to stop accepting new tasks and then `awaitTermination`() to wait for existing tasks to complete. Use `shutdownNow`() to attempt to stop all active tasks immediately.

4. Deadlocks:

Problem: Circular dependencies between tasks can cause deadlocks.

Mitigation:

- Carefully design code to avoid circular dependencies and use proper synchronization mechanisms like locks or semaphores.

5. Uncaught Exceptions:

Problem: Uncaught exceptions in tasks can lead to unexpected behavior.

Mitigation:

- Implement proper exception handling within tasks to prevent unexpected behavior, and log errors.

6. Blocking Operations:

Problem: Calling `Future.get`() immediately after submitting a task can block the main thread, negating the benefits of multithreading.

Mitigation:

- Submit all tasks first, save the Future results, and then call `get`() to retrieve results. Alternatively, use `invokeAll`() to submit a collection of `Callable` objects.

7. Premature Optimization:

Problem: Introducing database connection pools without a clear bottleneck can lead to unnecessary complexity.

Mitigation:

- Start with fresh connections and only consider a pool if it's proven to be a performance bottleneck.

8. Infinite Loops:

Problem: Tasks with infinite loops can cause issues if not handled correctly.

Mitigation:

- Catch `Errors` in addition to `Exceptions` and implement a back-off strategy to avoid escalating problems.

9. Incorrect Use of Threads:

Problem: Passing a `Thread` object directly to `Executor.execute`() is not appropriate since, as the `run`() method of a basic `Thread` does nothing.

Mitigation:

- Implement `Runnable` or `Callable` for tasks that need to be executed.

10. Task Cancellation:

Problem: Tasks may need to be canceled before completion (also applies to **6.62 Concurrency - Premature Termination**).

.

Mitigation:

- Use `Future.cancel()` to attempt to cancel a task and check `Future.isCancelled()` to verify if the task was canceled.

11. Executor Types:

Problem: Not choosing the appropriate executor type for the task.

Mitigation:

- Use `Executors` class to create different types of executor services like `newFixedThreadPool`, `newCachedThreadPool`, `newSingleThreadExecutor`, or `newWorkStealingPool` based on the specific needs.

Parallelism Model

Fork/Join Framework:

When using the Fork/Join framework in Java, the programmer might encounter issues related to task activation and performance. Here are some common problems and their mitigations:

1. Overhead of Task Splitting:

- **Problem**: Splitting tasks too finely can introduce excessive overhead, negating the benefits of parallel execution.
- **Mitigation**: Implement a proper threshold for task splitting. This means processing small subtasks sequentially within a single thread instead of creating further smaller tasks.

2. Blocking Operations:

- **Problem**: Using blocking operations within `ForkJoinTasks` can reduce performance.
- **Mitigation**: Avoid blocking calls in tasks or handle them outside the framework.

3. Inefficient Merging of Results:

- **Problem**: Inefficient result merging can negate parallelism benefits.
- **Mitigation**: Optimize the merge logic for efficiency.

4. Not Handling Exceptions Properly:

- **Problem**: Exceptions in tasks may not propagate as expected.
- **Mitigation**: Implement proper exception handling using methods like `getException()`.

5. Not Tuning the `ForkJoinPool`:

- **Problem**: Default configurations may not be optimal for all applications.
- **Mitigation**: Tune the `ForkJoinPool` by adjusting the parallelism level.

6. Over-reliance on the Common Pool:

- **Problem**: Using the common pool for tasks with blocking operations or different parallelism needs can cause performance issues.
- **Mitigation**: Use dedicated executors for tasks that are not strictly CPU-bound.

7. Deadlocks and Starvation:

- **Problem**: Deadlocks can occur when tasks are submitted to a busy thread's local queue.
- **Mitigation**: Avoid submitting tasks to the common pool from within a task already in that pool, especially for CPU-bound tasks.

8. Work-Stealing Issues:

- **Problem**: The default Last-In, First Out (LIFO) work-stealing approach can lead to thread imbalance.
- **Mitigation**: Consider a First-In, First Out (FIFO) approach if thread balance is critical and understand the trade-offs.

9. Task Granularity:

- **Problem**: Incorrect task granularity can hinder performance.
- **Mitigation**: Find a balance between task size and splitting/merging overhead.

10. Debugging Complexity:

- **Problem**: Debugging parallel execution is challenging.
- **Mitigation**: Use tools and techniques for concurrent programming debugging.

6.60 Concurrency – Directed termination [CGT])

Concurrency Model

Threading:

Effectively terminating Java threads requires careful consideration to avoid potential issues such as resource leaks and inconsistent states. Here are some methods to mitigate the pitfalls of directed thread termination:

Cooperative Thread Interruption:

Mitigation:

- Instead of forcefully stopping a thread, use `Thread.interrupt()` to set the thread's interrupt status to `true`. The thread's code should then periodically check this flag using `Thread.currentThread().isInterrupted()`and gracefully terminate when the flag is set. This approach allows the thread to finish its current operation and clean up resources before stopping.
- Many blocking operations (like `Thread.sleep()`,`Object.wait()`,etc.) throw an `InterruptedException` when the thread is interrupted. The code should catch this exception and use it as a signal to terminate the thread gracefully, ensuring proper cleanup.

Using a Volatile Flag:

Mitigation:

- Define a `volatile` boolean variable (e.g., `running` or `shouldStop`) within the thread's class. The `volatile` keyword ensures that changes to the flag are immediately visible to all threads. The thread's main loop should check this flag and exit when it is set to `false`.
- Create a public method (e.g., `stopThread()`) to set the volatile flag to `false`, signaling the thread to terminate.

Utilize `ExecutorService`:

Mitigation:

- Leverage `ExecutorService` for managed thread termination: If using an `ExecutorService`, use its `shutdown()` method to initiate an orderly shutdown of the thread pool. This allows submitted tasks to complete before the threads terminate.
- If a more urgent termination is needed, use `shutdownNow()`. However, be aware that this attempts to stop all running threads immediately, which might lead to unpredictable behavior if not handled carefully.

Avoid Deprecated Methods:

Mitigation:

- Refrain from using `Thread.stop()`, `Thread.suspend()`, and `Thread.resume()`. These methods are deprecated because they are inherently unsafe and can lead to data inconsistencies and deadlocks. They should not be used in modern Java applications.

Key Pitfalls and Mitigation Strategies:

Problem:
Resource leaks: Forceful termination can leave resources (e.g., file handles, database connections) open.

Mitigation:

- Implement cleanup logic within the thread's termination handling (e.g., in a `finally` block or by catching `InterruptedException`).

Problem:
Inconsistent state: Abruptly stopping a thread might leave shared data structures in an inconsistent state.

Mitigation:

- Use synchronization mechanisms (e.g., `synchronized` blocks, `Lock` objects) and ensure atomic operations to protect shared resources.

Problem:
Deadlocks: If a thread holds locks while being terminated, it can lead to deadlocks.

Mitigation:

- Design the threading logic to avoid holding locks during termination or use the cooperative termination approaches described above.

By following these best practices, the programmer can ensure the graceful termination of Java threads and mitigate the risks associated with directed thread termination.

Executors:

Using `ExecutorServices` in Java is a common way to manage thread pools and execute tasks concurrently. However, terminating an `ExecutorService` and ensuring graceful shutdown can sometimes present challenges.

Here are some common problems and their solutions regarding `ExecutorService` termination in Java:

1. ExecutorService Not Terminating Immediately:

Problem: After calling `shutdown`(), the `ExecutorService` might not terminate immediately because it waits for currently executing tasks to complete. If the tasks are long-running or get stuck, the `ExecutorService` won't terminate until those tasks finish or are interrupted.

Solution:

- Properly Shutdown: Call `shutdown`() to initiate an orderly shutdown where previously submitted tasks are allowed to complete, but no new tasks are accepted.
- Use `awaitTermination`(): Use `awaitTermination (long timeout, TimeUnit unit)` after calling `shutdown`() to wait for a specified period for all tasks to complete. This allows for a graceful shutdown within a set time limit.
- Handle Tasks that Don't Respond to Interrupts: If `awaitTermination`() returns `false` (meaning the timeout was reached before all tasks finished), the programmer can consider calling `shutdownNow`() as a last resort. This attempts to interrupt running tasks, but there's no guarantee they will terminate if they don't properly handle interruptions.
- Always call `shutdown()` after submitting tasks. Use `awaitTermination()` to wait for tasks to complete or use `shutdownNow()` to cancel them. For example:

```
executor.shutdown();

try {
   if (!executor.awaitTermination(5, TimeUnit.SECONDS)) {
       executor.shutdownNow();
   }
} catch (InterruptedException e) {
    executor.shutdownNow();
}
```

2. JVM Not Terminating Due to Non-Daemon Threads:

Problem: If the `ExecutorService` uses non-daemon threads and is not shut down, the JVM won't exit even after all tasks submitted to the executor have completed, because the non-daemon threads are still alive and waiting for new tasks.

Solution:

- Call `shutdown`(): Ensure to call `shutdown`() on the `ExecutorService` after submitting all tasks to allow for graceful shutdown and resource cleanup.
- Use Daemon Threads (with caution): Programmers can create the `ExecutorService` with a custom `ThreadFactory` that creates daemon threads. However, be mindful that daemon threads will be abruptly terminated when the JVM exits, potentially leading to incomplete tasks.

3. Tasks Not Being Interrupted by shutdownNow():

Problem: Calling `shutdownNow`() attempts to interrupt running tasks, but tasks that don't check for or respond to the interruption signal (`Thread.currentThread`().`isInterrupted`()) will continue executing.

Solution:

- Implement Interruption Logic: Design tasks (Runnables or Callables) to periodically check if their thread has been interrupted and handle the interruption appropriately (e.g., by stopping execution and cleaning up resources).
- Consider Alternatives for Non-Responsive Tasks: If tasks cannot be reliably interrupted, the programmer may need to find alternative ways to manage their lifecycle or use more robust shutdown mechanisms if necessary.

4. `awaitTermination`() Getting Stuck:

Problem: `awaitTermination`() might get stuck indefinitely if there are tasks that never complete or never respond to interruption signals.

Solution:

- Use a Timeout: Always use the version of `awaitTermination`() that includes a timeout to prevent the application from hanging indefinitely.
- Investigate Stalling Tasks: If `awaitTermination`() times out, debug the tasks that are still running to understand why they are not completing or responding to interruptions.

5. Handling Task Failures During Shutdown:

Problem: If a task within the `ExecutorService` throws an unhandled exception, it can affect the overall shutdown process and may prevent the `ExecutorService` from terminating gracefully.

Solution:

- Exception Handling within Tasks: Implement proper exception handling within tasks to catch and manage exceptions that occur during execution.
- Monitor Task Completion: Use Future objects returned by `submit`() or `invokeAll`() to monitor task completion and handle exceptions asynchronously.
- Centralized Exception Handling: Consider using an `ExecutorCompletionService` to process the results of tasks and handle exceptions in a centralized manner.

By addressing these common issues and implementing proper shutdown and exception handling mechanisms, the programmer can ensure the correct and reliable termination of `ExecutorService` in Java applications.

Parallelism Model

Fork/Join Framework:

NOTE: This section needs to be reworked since it mistakeably addresses 6.61 Concurrent data access only (it was inadvertently misplaced and duplicated) ...

~~The Fork/Join framework in Java facilitates parallel execution by dividing tasks into smaller subtasks, processing them concurrently, and then combining the results. However, it's essential to address potential pitfalls related to concurrent data access.~~

~~Pitfalls and Mitigations~~

~~Race Conditions:~~
~~Multiple threads attempting to modify shared data simultaneously can lead to data corruption.~~

- ~~**Mitigation**: Use thread-safe data structures from the `java.util.concurrent` package, such as `ConcurrentHashMap` or `AtomicInteger`, which provide atomic operations for safe concurrent access.~~

~~Lost Updates:~~
~~When multiple threads read, modify, and write to shared data, updates can be lost if not properly synchronized.~~

- ~~**Mitigation**: Employ atomic operations like `putIfAbsent`, `incrementAndGet`, or use locks to ensure that modifications are performed as a single, indivisible operation.~~

~~Deadlocks:~~
~~When threads are blocked indefinitely, waiting for each other to release resources.~~

- ~~**Mitigation**: Avoid circular dependencies in locking. Use timeouts when acquiring locks, or consider lock-free algorithms.~~

~~Excessive Thread Creation:~~
~~Creating too many threads can lead to high overhead, impacting performance.~~

- ~~**Mitigation**: Use a `ForkJoinPool` with a suitable level of parallelism, typically based on the number of available processors. Avoid creating a large number of threads.~~

~~Incorrect Threshold:~~
~~When implementing a Fork/Join algorithm, it's important to choose a threshold that determines whether a task will execute sequentially or fork into subtasks.~~

- ~~**Mitigation**: Experiment to find the optimal threshold. For very small tasks, sequential execution may be faster than forking.~~

~~Work Stealing Issues:~~
~~If a thread in the ForkJoinPool runs out of tasks, it can steal tasks from other threads.~~

- ~~**Mitigation**: Ensure that tasks are reasonably sized and balanced to avoid contention and work stealing overhead.~~

~~Misuse of Common Pool:~~
~~The common ForkJoinPool can be overloaded if used for all tasks, leading to performance issues.~~

- ~~**Mitigation**: Use custom ForkJoinPools for specific tasks, especially if they are long-running or have special requirements.~~

~~Non-Associative Operations:~~
~~The Fork/Join framework relies on operations being associative for combining results.~~

- ~~**Mitigation**: Ensure that the operation used to combine results is associative to avoid incorrect results.~~

~~General Recommendations~~

~~Use Thread-Safe Data Structures:~~

- ~~When sharing data between tasks, use thread-safe data structures from the `java.util.concurrent package`.~~

~~Minimize Shared Mutable Data:~~

- ~~Reduce the use of shared mutable data to minimize the need for synchronization.~~

~~Understand the Fork/Join Framework:~~

- ~~Thoroughly understand the Fork/Join framework's mechanics, including work stealing and task decomposition.~~

~~Proper Task Decomposition:~~

- ~~Decompose tasks into smaller, independent subtasks.~~

~~Test Thoroughly:~~

- ~~Test concurrent code thoroughly to identify and resolve potential concurrency issues.~~

6.61 Concurrent data access [CGX]

Concurrency Model

Threading:

Concurrent data access in Java arises when multiple threads attempt to read or modify shared data simultaneously. This can lead to several problems:

Problems:

1. Race Conditions:

- Occur when the outcome of a program depends on the unpredictable order in which threads execute.
- Two threads incrementing a shared counter variable. The final value might be incorrect if the increment operations are not atomic, for example:
- Can result in data corruption or inconsistent state if multiple threads try to modify the same data concurrently.

  When multiple threads access and modify the same shared resource concurrently without proper synchronization, the outcome becomes unpredictable, leading to race conditions. This can result in data corruption, incorrect results, or even application crashes. The following example illustrates a race condition scenario when accessing synchronized and un-synchronized data:

```
public class RaceConditionExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);

        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();

        System.out.println("Counter: " + counter.getCount());
    }
}
```

```
class Counter {
    private int count = 0;

    // NOTE: Scenario without the "synchronized" keyword is unpredictable
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

Output:

```
Scenario 1, (with synchronized): 2000
Scenario 2, (without synchronized): < varies, unpredictable >
```

In the above example, two threads increment a shared count variable. The `increment`() operation is not atomic, which means it can be broken down into three steps:

1. Read the current value of count
2. Increment count
3. Write the new value back to the variable count

If both threads execute these steps simultaneously, they can interfere with each other, leading to a lower final count than expected. The lack of synchronization on the `increment`() method creates a race condition, where the final value of count is unpredictable. Depending on usage, unsynchronized data can result in deadlocks, livelocks, and starvation.

2. Data Inconsistency:

- When threads read data while another thread is modifying it, the reading thread might get an inconsistent or outdated view of the data.
- This can lead to unexpected program behavior.

3. Deadlocks:

- Occur when two or more threads are blocked indefinitely, waiting for each other to release resources.
- Example: Thread A holds a lock on resource X and waits for a lock on resource Y, while Thread B holds a lock on resource Y and waits for a lock on resource X.

4. Starvation:

- A thread is repeatedly denied access to necessary resources, preventing it from completing its task.
- Can happen if some threads have higher priority or if a thread repeatedly loses the race for a resource.

5. Memory Consistency Errors:

- Occur when different threads have inconsistent views of shared data due to caching or compiler optimizations.
- One thread might update a variable, but other threads might not see the updated value.

6. Performance Issues:

- Excessive contention for shared resources can lead to performance degradation.
- Threads might spend a lot of time waiting for locks, reducing overall throughput.

Solutions:

Synchronization:

- Use synchronized blocks or methods to ensure that only one thread can access a critical section of code at a time.

Locks:

- Use explicit locks from the `java.util.concurrent.locks` package for more fine-grained control over synchronization.

Concurrent Collections:

- Use thread-safe collections like `ConcurrentHashMap`, `CopyOnWriteArrayList`, and others from the `java.util.concurrent` package, which are designed for concurrent access.

Atomic Variables:

- Use atomic classes like `AtomicInteger`, `AtomicLong`, etc., for thread-safe updates to single variables.

Immutable Objects:

- Use immutable objects whenever possible, as they are inherently thread-safe.

Thread Pools:

- Use `ExecutorService` to manage a pool of threads efficiently, reducing the overhead of creating and destroying threads.

Careful Design:

- Design code to minimize shared mutable state and the need for complex synchronization mechanisms.

Executors:

Concurrent data access in Java executors can lead to issues like race conditions, deadlocks, and data corruption. By following the mitigations listed below, the programmer can effectively lessen the consequences of concurrent data access in Java executors and build robust, reliable, and scalable applications.

Mitigation:

- Use Thread-Safe Data Structures: Employ concurrent collections from `java.util.concurrent`, such as `ConcurrentHashMap`, `ConcurrentLinkedQueue`, or `CopyOnWriteArrayList`. These are designed for concurrent access and minimize the need for explicit synchronization.
- Use the `synchronized` keyword to protect critical sections of code where shared data is accessed. This ensures that only one thread can access the synchronized code at a time.
- Utilize explicit `Lock` objects (e.g., `ReentrantLock`) from `java.util.concurrent.locks`. These offer more flexibility and control over locking than synchronized.
- For simple operations on single variables, use atomic classes like `AtomicInteger`, `AtomicLong`, or `AtomicReference`. These provide thread-safe operations without explicit locking.
- If possible, reduce the amount of shared mutable data. Prefer immutable objects or make copies of data before passing it to different threads.
- `ThreadLocal` variables provide each thread with its own copy of a variable, avoiding the need for synchronization when a variable is specific to a thread.
- Use thread pools from `Executors` to manage threads efficiently. Avoid creating a new thread for every task.
- Be mindful of lock acquisition order. Acquire locks in a consistent order to prevent deadlocks.
- Implement monitoring and logging mechanisms to diagnose and resolve concurrency issues.
- Use message passing or data aggregation techniques to reduce the need for shared state. For example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;

public class ConcurrentCounter {
    private AtomicInteger count = new AtomicInteger(0);
    public void increment() {
        count.incrementAndGet();
    }

    public int getCount() {
        return count.get();
    }

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(10);
        ConcurrentCounter counter = new ConcurrentCounter();

        for (int i = 0; i < 1000; i++) {
            executor.submit(counter::increment);
        }

        executor.shutdown();
        while (!executor.isTerminated()) {
        // Wait for all threads to finish
        }
        System.out.println("Count: " + counter.getCount());
    }
}
```

Output:

```
Count: 1000
```

In the above example, `AtomicInteger` ensures thread-safe increment operations.

Parallelism Model

Fork/Join Framework:

The Fork/Join framework in Java facilitates parallel execution by dividing tasks into smaller subtasks, processing them concurrently, and then combining the results. However, it's essential to address potential pitfalls related to concurrent data access.

Pitfalls and Mitigations

Race Conditions:
Multiple threads attempting to modify shared data simultaneously can lead to data corruption.

- **Mitigation**: Use thread-safe data structures from the `java.util.concurrent` package, such as `ConcurrentHashMap` or `AtomicInteger`, which provide atomic operations for safe concurrent access.

Lost Updates:
When multiple threads read, modify, and write to shared data, updates can be lost if not properly synchronized.

- **Mitigation**: Employ atomic operations like `putIfAbsent`, `incrementAndGet`, or use locks to ensure that modifications are performed as a single, indivisible operation.

Deadlocks:
When threads are blocked indefinitely, waiting for each other to release resources.

- **Mitigation**: Avoid circular dependencies in locking. Use timeouts when acquiring locks, or consider lock-free algorithms.

Excessive Thread Creation:
Creating too many threads can lead to high overhead, impacting performance.

- **Mitigation**: Use a `ForkJoinPool` with a suitable level of parallelism, typically based on the number of available processors. Avoid creating a large number of threads.

Incorrect Threshold:
When implementing a Fork/Join algorithm, it's important to choose a threshold that determines whether a task will execute sequentially or fork into subtasks.

- **Mitigation**: Experiment to find the optimal threshold. For very small tasks, sequential execution may be faster than forking.

Work Stealing Issues:
If a thread in the `ForkJoinPool` runs out of tasks, it can steal tasks from other threads.

- **Mitigation**: Ensure that tasks are reasonably sized and balanced to avoid contention and work stealing overhead.

Misuse of Common Pool:
The common `ForkJoinPool` can be overloaded if used for all tasks, leading to performance issues.

- **Mitigation**: Use custom `ForkJoinPools` for specific tasks, especially if they are long-running or have special requirements.

Non-Associative Operations:
The Fork/Join framework relies on operations being associative for combining results.

- **Mitigation**: Ensure that the operation used to combine results is associative to avoid incorrect results. An associative operation is one where the grouping of operands does not affect the result. This is particularly relevant when dealing with multiple operations of the same type

General Recommendations

Use Thread-Safe Data Structures:

- When sharing data between tasks, use thread-safe data structures from the `java.util.concurrent` package.

Minimize Shared Mutable Data:

- Reduce the use of shared mutable data to minimize the need for synchronization.

6.62 Concurrency – Premature termination [CGS]

Concurrency Model

Threading:

Premature thread termination in Java can occur for several reasons, including uncaught exceptions, explicit thread termination, or unhandled interruptions. Here's how to address and mitigate premature thread termination:

1. Proper Exception Handling:

- `Try-catch` blocks: Wrap potentially problematic code within a `try-catch` block to handle exceptions gracefully within the thread's execution.
- Unhandled exceptions: If an exception is not caught, it can lead to thread termination. The programmer might need to use `UncaughtExceptionHandler` to log unhandled exceptions and potentially restart the thread or trigger a process-wide shutdown if necessary.
- Logging: Log any exceptions to understand the root cause of the termination.

2. Graceful Thread Termination:

- Avoid `Thread.stop`(): This method is deprecated and unsafe as it can lead to inconsistent state and resource leaks.
- Use volatile flags: Use a `volatile` boolean flag that the thread checks periodically to determine if it should stop.
- Use `Thread.interrupt`(): This method sets the thread's interrupt status to `true`, allowing the thread to check this status and exit gracefully.
- Handle `InterruptedException`: When a thread is interrupted while waiting or sleeping, it throws `InterruptedException`. Catch this exception and take appropriate actions, like breaking out of the current task and exiting the thread.
- ExecutorService: For managing a pool of threads, use `shutdown`() to allow tasks to complete gracefully or `shutdownNow`() to attempt an immediate stop.

3. Thread Lifecycle Management:

- Daemon threads: If it is desired to have a thread to terminate when the main thread exits, set it as a daemon thread using `set Daemon(true)`.
- Join threads: Use `thread.join`() to wait for a thread to complete before the main thread continues.

4. Best Practices:

- Limit the number of threads: Creating too many threads can overload the CPU and lead to performance issues.
- Use thread pools: Use `ExecutorService` to manage a pool of reusable threads for better performance and resource management.
- Minimize synchronization: Use `java.util.concurrent` utilities like `ConcurrentHashMap` or `BlockingQueue` to reduce contention and improve performance.
- Monitor thread states: Use `Thread.getState()` to debug and understand thread behavior.

Executors:

Premature termination of an `ExecutorService` in Java can lead to unexpected program behavior, resource leaks, or incomplete processing. Here are common problems and mitigations for premature termination:

1. Improper Shutdown:

Problem:
- Not properly shutting down the `ExecutorService` after use. This can lead to resource leaks (e.g., threads not being released) and prevent the application from exiting.

Mitigation:
- Always call `shutdown()` or `shutdownNow()` when the `ExecutorService` is no longer needed.
- `shutdown()`: Allows currently executing tasks to complete before terminating.
- `shutdownNow()`: Attempts to stop currently running tasks and prevents waiting tasks from starting.
- Consider using `awaitTermination()` after calling `shutdown()` to wait for a specific duration for tasks to finish.
- Use a `finally` block to ensure shutdown is called even if exceptions occur.
- In application server environments (like Java EE), use a `ManagedExecutorService` which is managed by the framework and handled automatically.

2. Task Failures and Exceptions:

Problem:
- Unhandled exceptions within tasks can cause individual threads to terminate prematurely, impacting the overall `ExecutorService`.

Mitigation:

- Implement robust exception handling within the tasks themselves using `try-catch` blocks.
- Use `Future` objects (returned by `submit`()) and call `get`() to retrieve the result or exception of the task after completion.
- Set an `UncaughtExceptionHandler` for threads using a custom `ThreadFactory` to log or manage exceptions globally or per-thread.
- If using `scheduleAtFixedRate`, and tasks take longer than the period, consider using `scheduleWithFixedDelay` to avoid overlapping executions.

<u>3. Stuck or Deadlocked Tasks:</u>

Problem:

- Tasks that get stuck (e.g., due to infinite loops or resource deadlocks) can prevent the `ExecutorService` from completing its work.

Mitigation:

- Use timeouts on tasks or futures to prevent them from running indefinitely.
- Monitor the `ExecutorService` to detect stuck tasks or deadlocks.
- Use thread dumps to diagnose hangs or loops within the application or library code.
- Consider implementing a monitoring task to set a flag or signal when other tasks fail, allowing for potential shutdown.

<u>4. Incorrect Thread Pool Configuration:</u>

Problem:

- An improperly sized thread pool can lead to problems like performance degradation, memory leaks, or tasks being rejected.

Mitigation:

- Choose the appropriate thread pool size based on the task nature (CPU-bound vs. I/O-bound), workload characteristics, and system constraints.
- Utilize bounded queues to prevent out-of-memory errors and handle task bursts.
- Avoid creating excessive threads, which can lead to context switching overhead and reduced performance.
- Consider using a `ThreadPoolExecutor` with a minimum of 0 threads and a keep-alive time to automatically clean up idle threads.

<u>5. Other Considerations:</u>

- Task Design: Avoid submitting tasks that block for extended periods on external resources, such as network or database operations.
- Thread Naming: Use a custom `ThreadFactory` to provide meaningful names for threads, aiding in debugging and monitoring.

- Monitoring: Regularly monitor thread pool metrics (active threads, queue size, task throughput) to identify bottlenecks and fine-tune configurations.
- `CompletableFuture`: Use `CompletableFuture` for more advanced asynchronous programming and task composition.
- Virtual Threads: If tasks involve blocking operations, consider using virtual threads (available in Java 21+) for increased throughput.
- Shutdown Hooks: Use `addShutdownHook` to shut down the `ExecutorService` gracefully when the application terminates naturally.

By carefully managing the lifecycle of the `ExecutorService`, handling exceptions, and configuring the thread pool appropriately, the programmer can mitigate the risk of premature termination and ensure the reliable execution of the tasks and a graceful shutdown.

Parallelism Model

Fork/Join Framework:

In Java's Fork/Join framework, premature termination (or unexpected termination) can occur in various scenarios, potentially leading to incomplete results or deadlocks. Here's a breakdown of common causes and mitigation strategies:

1. Blocking Operations:

Problem:

- Fork/Join is designed for CPU-bound, non-blocking tasks. When a `ForkJoinTask` performs blocking operations (like I/O, database calls, or waiting on locks), the worker thread becomes unavailable to execute other tasks, potentially causing deadlocks or starvation.

Mitigation:

- Avoid Blocking: Refactor tasks to avoid blocking operations within the `compute()` method.
- Use Asynchronous Alternatives: Consider using asynchronous programming models or non-blocking I/O when needed.
- External Blocking: If external blocking is unavoidable, use standard `ExecutorService` and Callable instead of `ForkJoinPool`.

2. Incorrect Usage of `join()`:

Problem:

- Calling `join()` too early or incorrectly can cause the current thread to block unnecessarily, hindering parallel execution.

Mitigation:

- Call `join()` as the last step: After forking subtasks, call `join()` on them only when their results are needed.
- Use `invokeAll()` for multiple tasks: This method efficiently forks and joins a collection of tasks, often avoiding manual joining logic errors.

3. Task Dependency Issues:

Problem:

- If a task depends on the completion of another task that is not forked or has stalled, it can lead to premature termination or deadlock.

Mitigation:

- Proper Task Structuring: Ensure tasks are structured as a Directed Acyclic Graph (DAG) to avoid cyclic dependencies that can cause deadlocks.
- Check `isDone()` before `join()`: While not always necessary, checking `isDone()` can provide a sanity check, but it's important to remember that it only indicates the completion state of a task, not the queue.

4. Exceptions:

Problem:

- Unhandled exceptions within a `ForkJoinTask` can cause the task to terminate prematurely, potentially leaving other tasks incomplete or in an inconsistent state.

Mitigation:

- Exception Handling: Wrap potentially problematic code within try-catch blocks to handle exceptions gracefully.
- Propagate Exceptions: If necessary, propagate exceptions to the calling task for appropriate handling or cancellation of other tasks.

5. ForkJoinPool Configuration Issues:

Problem:

- Incorrectly configuring the `ForkJoinPool` (e.g., inadequate parallelism or misuse of `asyncMode`) can impact performance and lead to unexpected behavior.

Mitigation:

- Use the Common Pool: For most applications, use the default common pool provided by `ForkJoinPool.commonPool().`
- Consider Custom Pools: If specific tuning is needed, create a custom pool with a suitable parallelism level.
- Tune for Performance: Test and tune the application's Fork/Join configuration for optimal performance on target hardware.

6. Daemon Threads:

Problem:

- By default, `ForkJoinPool` uses daemon threads, meaning the application can exit before submitted tasks complete if no non-daemon threads are active.

Mitigation:

- Block or Wait: If necessary, ensure the main thread waits for the completion of tasks (e.g., by calling `join()` on the root task) before exiting.

In summary, effective mitigation involves careful task design, avoiding blocking operations, correct use of `join()` and `invokeAll()`, robust exception handling, and appropriate `ForkJoinPool` configuration.