Java Concurrency Utilities Summary

Multicore Computing Assignement n’2

Joshua BRIONNE - 50241647

1. BlockingQueue and ArrayBlockingQueue

A thread-safe queue that facilitates waiting for elements to be added or removed is represented by the `BlockingQueue` interface (from `java.util.concurrent`). A thread will halt until an element becomes available if it attempts to use `take()` to retrieve an element from an empty queue. Likewise, a thread that calls `put()` will wait until there is room in the queue if it is full. Coordinating producer and consumer threads without creating your own locking logic is made simpler by this behavior.

One implementation of `BlockingQueue` is `ArrayBlockingQueue`. Internally, it makes use of a fixed-size array, meaning that its capacity is predetermined at creation and cannot be altered afterwards. The first thing added will be the first item withdrawn since items are handled in a First-In First-Out (FIFO) order. Due to its thread-safe and blocking properties, it's frequently

Code example:

```
import java.util.concurrent.*;
public class ex1 {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(1);
        Thread producer = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                try {
                    System.out.println("Producer putting: " + i);
                    queue.put(i);
                    System.out.println("Producer put: " + i);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        Thread consumer = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                try {
                    int item = queue.take();
                    System.out.println("Consumer took: " + item);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
    }
}
```

Exemple Output:

```
Producer putting: 0
Producer put: 0
Producer putting: 1
Consumer took: 0
Producer put: 1
Producer putting: 2
Consumer took: 1
Producer put: 2
Producer putting: 3
Consumer took: 2
Producer put: 3
Producer putting: 4
Consumer took: 3
Producer put: 4
Consumer took: 4
```

In the example, the producer thread uses `put()` to add digits from 0 to 4. After adding the first item, the producer stalls and waits for the consumer to remove it using `take()` because the queue can only hold one value.

In order to process and remove values from the queue, the consumer thread repeatedly invokes `take()`. An alternate pattern can be seen in the program output: the producer puts a value first, and the consumer then takes it. This demonstrates how the queue handles thread synchronization automatically without requiring manual locking.

2. ReadWriteLock

In the example, the producer thread uses `put()` to add digits from 0 to 4. After adding the first item, the producer stalls and waits for the consumer to remove it using `take()` because the queue can only hold one value.

In order to process and remove values from the queue, the consumer thread repeatedly invokes `take()`. An alternate pattern can be seen in the program output: the producer puts a value first, and the consumer then takes it. This demonstrates how the queue handles thread synchronization automatically without requiring manual locking.

Data races are lessened by this arrangement. It guarantees exclusive access when modifications are performed while avoiding needless blocking during reads.

Code example:

```
import java.util.concurrent.locks.*;
class SharedData {
    private int value = 0;
    private ReadWriteLock rwLock = new ReentrantReadWriteLock();
    // Write
    public void write(int newValue) {
        rwLock.writeLock().lock();
        try {
            value = newValue;
            System.out.println(Thread.currentThread().getName() + " wrote "
+ newValue);
        } finally {
            rwLock.writeLock().unlock();
        }
    }
    // Reade
    public int read() {
        rwLock.readLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + " read "
+ value);
            return value;
        } finally {
            rwLock.readLock().unlock();
        }
    }
}
public class ex2 {
    public static void main(String[] args) throws InterruptedException {
        SharedData data = new SharedData();
        Thread writer = new Thread(() -> {
            for (int i = 1; i <= 3; i++) {
                data.write(i);
                try { Thread.sleep(100); } catch (InterruptedException e)
```

```
{}
            }
        }, "Writer-Thread");
        Thread reader1 = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                data.read();
                try { Thread.sleep(100); } catch (InterruptedException e)
{}
            }
        }, "Reader-Thread-1");
        Thread reader2 = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                data.read();
                try { Thread.sleep(100); } catch (InterruptedException e)
{}
            }
        }, "Reader-Thread-2");
        writer.start();
        reader1.start();
        reader2.start();
        writer.join();
        reader1.join();
        reader2.join();
    }
}
```

Exemple Output:

```
Reader-Thread-1 read 0
Reader-Thread-2 read 0
Writer-Thread wrote 1
Reader-Thread-1 read 1
Reader-Thread-2 read 1
Writer-Thread wrote 2
Reader-Thread-1 read 2
Reader-Thread-2 read 2
Writer-Thread wrote 3
```

In the example, a separate writer thread acquires the write lock to update the value, while two reader threads repeatedly acquire the read lock to show the current value. The initial value (0) was read simultaneously by both readers at the start. Readers see the updated value on their subsequent reading once the author sets it to 1.

The output makes it evident that the readers can work simultaneously because they both read the same value prior to the subsequent update. However, the writer only updates the value once, making sure that no reads or additional writes take place simultaneously. This shows how the `ReadWriteLock` manages access and protects the information.

3. AtomicInteger

A utility class called `AtomicInteger` (from `java.util.concurrent.atomic`) allows several threads to safely read and modify an integer value simultaneously without the need for explicit synchronization. All of its activities are carried out atomically, meaning that other threads won't interfere with them and they will either finish completely or not.

Calling `get()`, for instance, will thread-safely return the current value. You can increase the value and obtain the result in a single atomic step by using methods like `addAndGet()` or `getAndAdd()`. When you need to update shared data or counters in concurrent contexts, this is really helpful.

Code example:

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

public class ex3 {

    public static void main(String[] args) {

        AtomicInteger ai = new AtomicInteger(0);

        System.out.println("Initial value: " + ai.get());

        ai.set(5);

        System.out.println("Value after set(5): " + ai.get());

        int oldValue = ai.getAndAdd(10);

        System.out.println("getAndAdd(10) returned: " + oldValue + ", new
value: " + ai.get());

        int newValue = ai.addAndGet(3);

        System.out.println("addAndGet(3) returned: " + newValue + ", new
value: " + ai.get());

    }

}
```

Exemple Output:

```
Initial value: 0
Value after set(5): 5
getAndAdd(10) returned: 5, new value: 15
addAndGet(3) returned: 18, new value: 18
```

An `AtomicInteger` is initially set to 0 in this example. Then, `set()` is used to set its value to 5. The value is then updated to 15 with the `getAndAdd(10)` method, which adds 10 while returning the prior value, which was 5. The new number, 18, is then returned instantly after `addAndGet(3)` adds 3.

These operations are all atomic, which means they all take place in a single, uninterrupted step. Therefore, there would be no need for further synchronization or locks if several threads were altering the same `AtomicInteger` simultaneously. These changes would still be secure and reliable.

4. CyclicBarrier

With the help of a `CyclicBarrier` (from `java.util.concurrent`), a collection of threads can be made to wait for one another at the same time. It is created by specifying how many threads, or "parties," must cross the barrier. Every thread stops there until the final one arrives after using `await()` on the barrier. The barrier opens to allow all threads to proceed after they have all invoked `await()`.

This is particularly helpful for jobs that must be completed in steps, where each thread must complete a step before proceeding to the next. When threads must maintain synchronization at specific program places, it facilitates the coordination of concurrent tasks.

Code example:

```
import java.util.concurrent.*;

public class ex4 {

    public static void main(String[] args) {

        final int parties = 3;

        CyclicBarrier barrier = new CyclicBarrier(parties,

            () -> System.out.println("All threads have reached the barrier.
Continuing...")

        );

        Runnable worker = () -> {

            try {

                System.out.println(Thread.currentThread().getName() + "
waiting at barrier");

                barrier.await();

                System.out.println(Thread.currentThread().getName() + "
passed the barrier");

            } catch (Exception e) {

                e.printStackTrace();

            }

        };

        for (int i = 1; i <= parties; i++) {

            new Thread(worker, "Thread-" + i).start();

        }

    }

}
```

Exemple Output:

```
Thread-0 reached barrier
Thread-1 reached barrier
Thread-2 reached barrier
Barrier reached, all threads can proceed
Thread-0 passed barrier
Thread-1 passed barrier
Thread-2 passed barrier
```

We can clearly see here that all threads are running to the barrier and once they reach it, they wait for everyone to arrive before going all at once beyond it !

5. ExecutorService, Executors, Callable, and Future

The `ExecutorService` interface in Java offers a framework for asynchronous task execution as well as a pool of threads. You can submit tasks without being concerned about creating or managing threads because it abstracts away the direct management of `Thread` objects. Factory methods for creating several kinds of `ExecutorService` instances are provided by the `Executors` utility class. `Executors.newFixedThreadPool(n)`, for example, generates a thread pool with a predetermined number of threads.

Below is a summary of the key elements:

An interface for carrying out asynchronous activities is called **ExecutorService**. It provides ways to submit tasks, manage shutdown, and oversee a pool of threads. The `submit()` method returns a `Future` for each job that is submitted, and tasks can be submitted as either `Runnable` or `Callable` objects.

Executors: A factory class that offers static methods such as `newFixedThreadPool` and `newCachedThreadPool` to facilitate the creation of various `ExecutorService` implementations effortlessly. `Executors.newFixedThreadPool(4)`, for instance, generates a thread pool with four threads.

Comparable to `Runnable`, **Callable** is intended for jobs that yield a result. `V call() throws Exception` is the only method defined by the `Callable` interface. The value of type `V` may be returned when it is executed, usually by a `ExecutorService`. A `Future` can be used to get the outcome later.

Future: Indicates the outcome of an asynchronous calculation. A `Future` is returned by a `ExecutorService` when you submit a `Callable` task to it. The Future`serves as a handle to the outcome that is still pending. You can access the result by calling`future.get()`after the task is finished. `get()`will halt until the outcome is available if the task isn't completed. additionally, a`Future` has methods to abort a job (`cancel()`) and determine whether it is finished (`isDone()`).

Together, these elements function as follows: you establish a `ExecutorService` (such as a fixed-size thread pool) using `Executors`, and then submit tasks that implement `Callable`. A

`Future` is returned by each `submit()`. When the results are ready, you can then call `get()` on the `Future` to get them.

```
import java.util.concurrent.*;
import java.util.List;
import java.util.ArrayList;
public class ex5 {
    public static void main(String[] args) throws Exception {
        final int RANGE = 200_000;
        final int THREADS = 4;
        ExecutorService executor = Executors.newFixedThreadPool(THREADS);
        int chunkSize = RANGE / THREADS;
        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < THREADS; i++) {
            int start = i * chunkSize + 1;
            int end = (i == THREADS - 1) ? RANGE : (i + 1) * chunkSize;
            futures.add(executor.submit(new PrimeCountTask(start, end)));
        }
        int totalPrimes = 0;
        for (Future<Integer> future : futures) {
            totalPrimes += future.get();
        }
        executor.shutdown();
        System.out.println("Total primes between 1 and " + RANGE + ": " +
totalPrimes);
    }
}
class PrimeCountTask implements Callable<Integer> {
    private final int start, end;
    public PrimeCountTask(int s, int e) { this.start = s; this.end = e; }
    @Override
    public Integer call() {
        int count = 0;
        for (int i = start; i <= end; i++) {
            if (isPrime(i)) {
                count++;
```

```
            }
        }
        return count;
    }
    private boolean isPrime(int n) {
        if (n < 2) return false;
        if (n % 2 == 0) return n == 2;
        int sqrt = (int)Math.sqrt(n);
        for (int i = 3; i <= sqrt; i += 2) {
            if (n % i == 0) return false;
        }
        return true;
    }
}
```

Exemple Output:

```
Total primes between 1 and 200000: 17984
```

Future: Indicates the outcome of an asynchronous calculation. A `Future` is returned by a `ExecutorService` when you submit a `Callable` task to it. The `Future` serves as a handle to the outcome that is still pending. You can access the result by calling `future.get()` after the task is finished. `get()` will halt until the outcome is available if the task isn't completed.

Additionally, a `Future` has methods to abort a job (`cancel()`) and determine whether it is finished (`isDone()`).

Together, these elements function as follows: you establish a `ExecutorService` (such as a fixed-size thread pool) using `Executors`, and then submit tasks that implement `Callable`. A `Future` is returned by each `submit()`. When the results are ready, you can then call `get()` on the `Future` to get them.

We print the total number of primes discovered at the end. This illustrates how `ExecutorService`, `Callable`, and `Future` cooperate: tasks are carried out in parallel in different threads, and the `Future` objects collect the results asynchronously.