Semaphore Bulkhead in Resilience4j: Complete Notes

1. Overview

The **Bulkhead pattern** isolates failures and limits resource usage by partitioning system components. In microservices, it prevents one slow or failing service from exhausting shared resources and impacting the entire application.

Resilience4j provides two types of Bulkheads:

- **Semaphore Bulkhead**: Limits concurrent calls in the calling thread.
- **ThreadPool Bulkhead**: Uses a dedicated thread pool for protected calls.

These notes focus on **Semaphore Bulkhead** configured *programmatically* in a Spring Boot 3.5.3 application using Resilience4j 2.x.

---

2. Semaphore Bulkhead Concept

- **Analogy**: A restaurant with a fixed number of waiters (threads). Only a limited number of customers (requests) can be served simultaneously. Extra customers are rejected or wait.
- **Goal**: Restrict the number of concurrent executions to a service/method.
- **Key Parameters**:
- `maxConcurrentCalls`: Maximum threads allowed in parallel.
- `maxWaitDuration`: Time a call will wait for a permit before failing (can be zero).

---

3. Why Use Semaphore Bulkhead?

- Protects against thread-pool exhaustion.
- Prevents cascading failures when downstream services are slow/unreliable.
- Ensures system remains responsive under load by bounding parallelism.

---

4. Programmatic Configuration (No YAML)

1. **Add Dependency** in `pom.xml`:

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-bulkhead</artifactId>
  <version>2.2.0</version>
</dependency>
```

1. **Register Beans** in Java config:

```
@Configuration
public class BulkheadJavaConfig {

    @Bean
    public BulkheadConfig customBulkheadConfig() {
        return BulkheadConfig.custom()
                .maxConcurrentCalls(2)
                .maxWaitDuration(Duration.ZERO)
                .build();
    }

    @Bean
    public BulkheadRegistry bulkheadRegistry(BulkheadConfig 
customBulkheadConfig) {
        return BulkheadRegistry.of(customBulkheadConfig);
    }

    @Bean
    public Bulkhead customBulkhead(BulkheadRegistry registry) {
        return registry.bulkhead("customBulkhead");
    }
}
```

1. **RestTemplate Bean** (for HTTP calls):

```
@Configuration
public class RestTemplateConfig {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

---

5. Service Implementation

```
@Service
public class PaymentService {

    private final Bulkhead bulkhead;
    private final RestTemplate restTemplate;

    @Autowired
    public PaymentService(Bulkhead customBulkhead, RestTemplate restTemplate) {
        this.bulkhead = customBulkhead;
        this.restTemplate = restTemplate;
    }

    public Object processPayment() {
        Callable<Object> guardedCall = Bulkhead.decorateCallable(bulkhead, () -
> {
            System.out.println("Processing on " +
Thread.currentThread().getName());
            ResponseEntity<Object> resp = restTemplate.getForEntity(
                    "http://localhost:9090/", Object.class);
            return resp.getBody();
        });

        try {
            return guardedCall.call();
        } catch (BulkheadFullException ex) {
            return "Too many concurrent payments. Try later.";
        } catch (Exception ex) {
            return "Error: " + ex.getMessage();
        }
    }
}
```

Key Points:

- Use `decorateCallable` to wrap business logic.
- `BulkheadFullException` indicates concurrency limit reached.
- No use of `Timer`; direct invocation ensures compatibility.

---

6. Controller Example

```
@RestController
public class PaymentController {

    private final PaymentService paymentService;

    public PaymentController(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    @GetMapping("/pay")
    public Object pay() {
        return paymentService.processPayment();
    }
}
```

7. Testing Behavior

- **Concurrent Requests**: Try sending 5 simultaneous requests to `/pay`.
- **Expected**: 2 succeed, 3 immediately return fallback message.

8. Best Practices

- Tune `maxConcurrentCalls` based on **CPU**, **I/O-bound nature**, and **memory**.
- Combine with **RateLimiter** and **CircuitBreaker** for comprehensive resilience.
- Monitor via **Actuator** (`/actuator/metrics/jvm.threads.live`) or APM.

9. ThreadPool Bulkhead

- **Purpose**: Offloads work to a separate, dedicated thread pool. Ideal for asynchronous or CPU-intensive tasks.
- **Key Parameters**:
- `coreThreadPoolSize`: Number of threads in the pool.
- `maxThreadPoolSize`: Maximum threads allowed (if using dynamic sizing).
- `queueCapacity`: How many tasks can wait in the queue.
- `keepAliveTime`: Time to keep extra threads alive.

9.1 Programmatic Configuration

```
@Configuration
public class ThreadPoolBulkheadConfig {

    @Bean
    public ThreadPoolBulkheadConfigProperties tpProperties() {
        return ThreadPoolBulkheadConfigProperties.of(
            ThreadPoolBulkheadConfig.custom()
                .coreThreadPoolSize(2)
                .maxThreadPoolSize(4)
                .queueCapacity(10)
                .keepAliveTime(Duration.ofSeconds(30))
                .build()
        );
    }

    @Bean
```

```java
    public ThreadPoolBulkheadRegistry
threadPoolRegistry(ThreadPoolBulkheadConfigProperties props) {
        return ThreadPoolBulkheadRegistry.of(props.getConfigs());
    }

    @Bean
    public ThreadPoolBulkhead
paymentThreadPoolBulkhead(ThreadPoolBulkheadRegistry registry) {
        return registry.bulkhead("paymentThreadPoolBulkhead");
    }
}
```

9.2 Service Usage

```java
@Service
public class AsyncPaymentService {

    private final ThreadPoolBulkhead bulkhead;

    public AsyncPaymentService(ThreadPoolBulkhead paymentThreadPoolBulkhead) {
        this.bulkhead = paymentThreadPoolBulkhead;
    }

    public CompletableFuture<String> processAsync() {
        return CompletableFuture.supplyAsync(
            Bulkhead.decorateSupplier(bulkhead, () -> {
                // heavy computation or I/O-bound work
                return "✅ Async payment done";
            })
        );
    }
}
```

9.3 When to Use ThreadPool Bulkhead

- **Asynchronous methods** (`CompletableFuture`, `WebClient`) that should not occupy main thread pool.
- **CPU-intensive tasks** (image processing, data transformation).
- **Isolating** high-latency operations into their own pool.

---

10. Summary

Semaphore Bulkhead limits concurrent calls on the current thread. ThreadPool Bulkhead isolates execution into its own thread pool. Use programmatic configuration to fine-tune both patterns without touching property files.