Multithreading – Java vs. Node.js

Azat Satklichov
azat.satklichov@broadcom.com

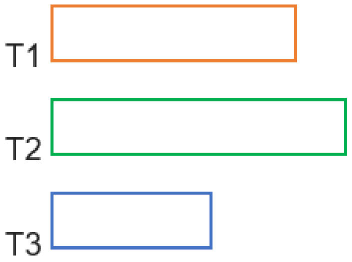
Agenda

- How can I run my slow operations faster?
- Multitasking and multithreading
- Java Thread Core API. Thread Synchronization
- Java Synchronization Limitations. Thread Safety
- Java Concurrency API
- Enhanced Java Concurrency (new era)
- Node.js Concurrency – Event Loop
- Node.js Concurrency – Worker Threads
- Measurement Matrix (Java vs. Node.js)
- Concurrency Models
- References

How can I run my slow operations faster?

Suppose, we have a task:

- Logistic company wants to generate quotation-report
  [download rate – T1, process – T2, generate report – T3, …)
- Or other example, bank wants to get clearance-report [1?, 2?, 3?, ..]



Slow operations performance is improved by using one of below mechanisms depending on use-case.

Synchronous

Fork new process from OS

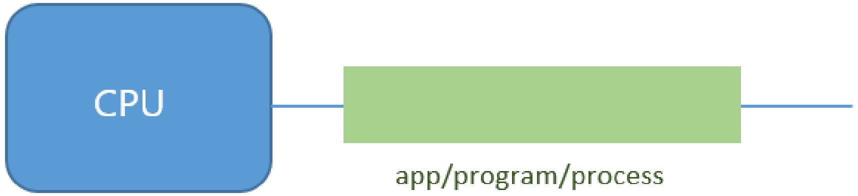
Threads

Event Driven

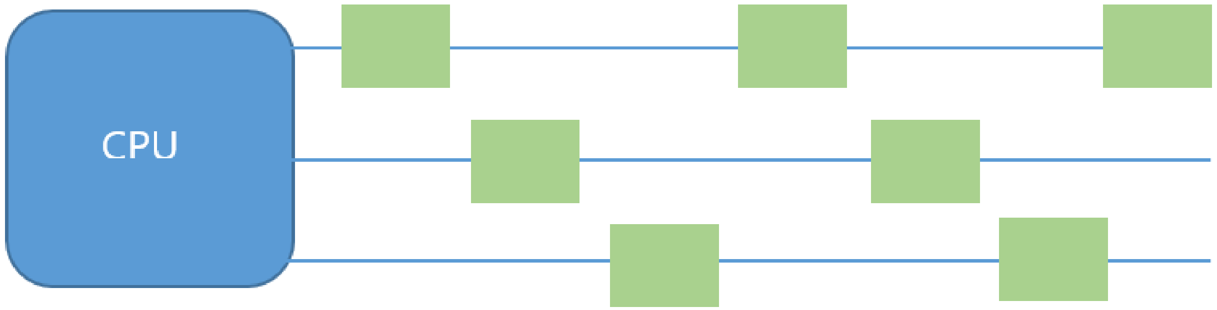
Multitasking

Process

Multitasking is the concurrent execution of multiple tasks over a certain period of time. It's Processor based and thread based.



Single CPU can run single program at a time



Single CPU can run multiple program by switching between execution

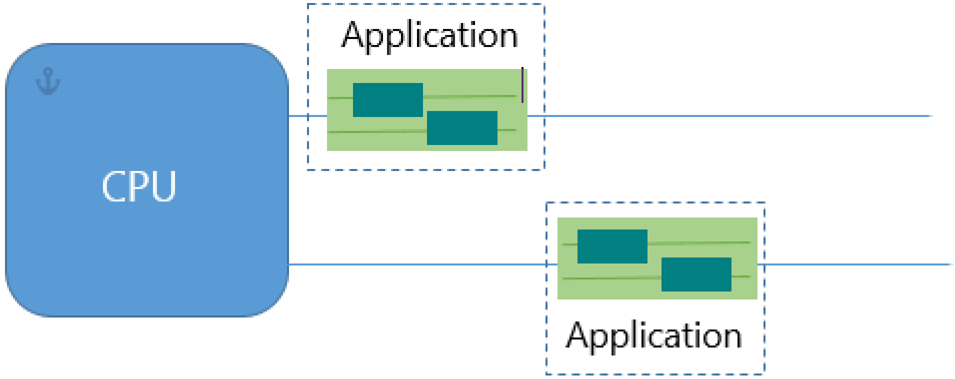
BROADCOM

Multithreading

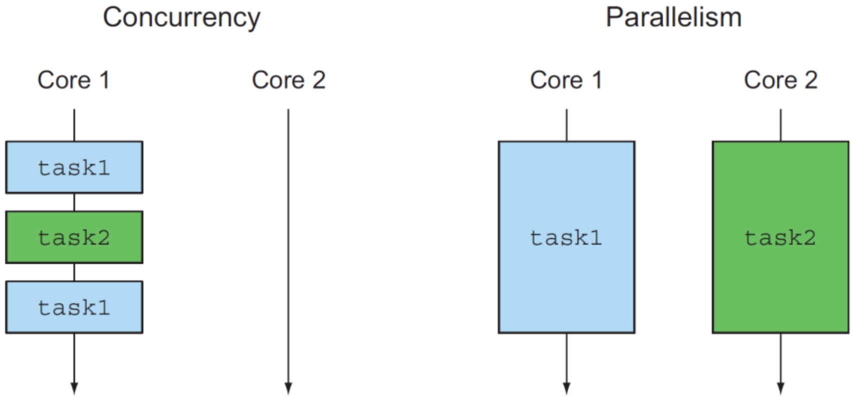
Thread

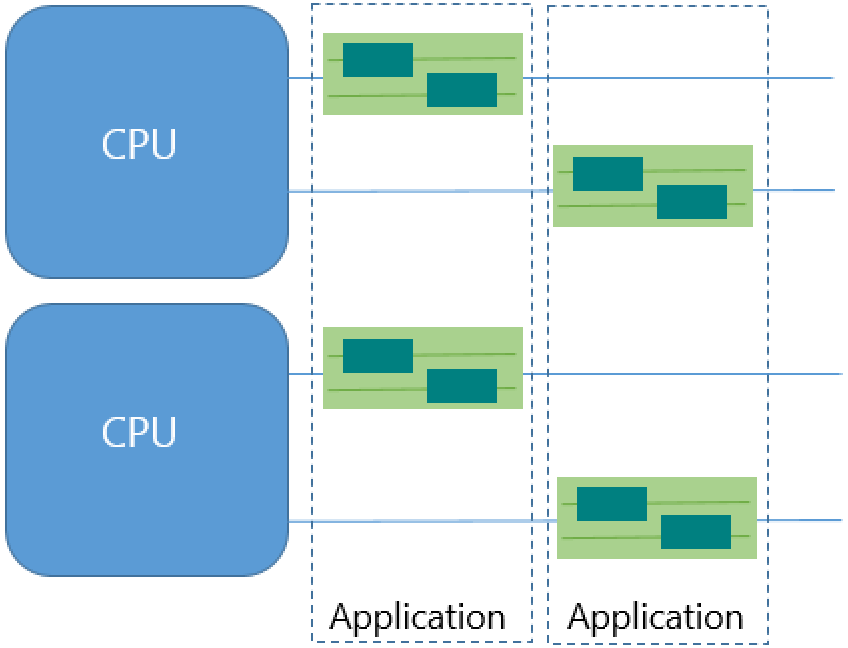
Multithreading is a type of execution model where multiple threads (light-weight processes) can run independently in a process.

Multiple threads executed on single CPU



Concurrency vs Parallelism





Multiple threads executed on multiple CPU, or CPU with multiple COREs

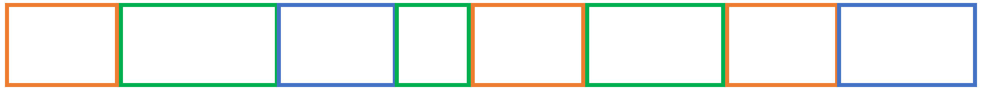
How can I run my slow operations faster?

1-way: (easiest): Execute synchronously

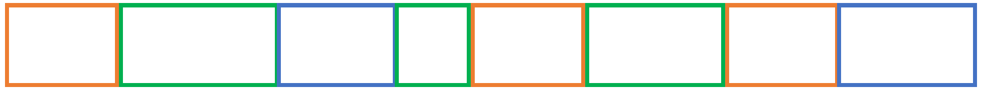
2-way: Multithreading execution

2a) thread per task, at least three cores CPU

2b) Multithreading execution (on single core CPU, *preemptive switching by OS)*



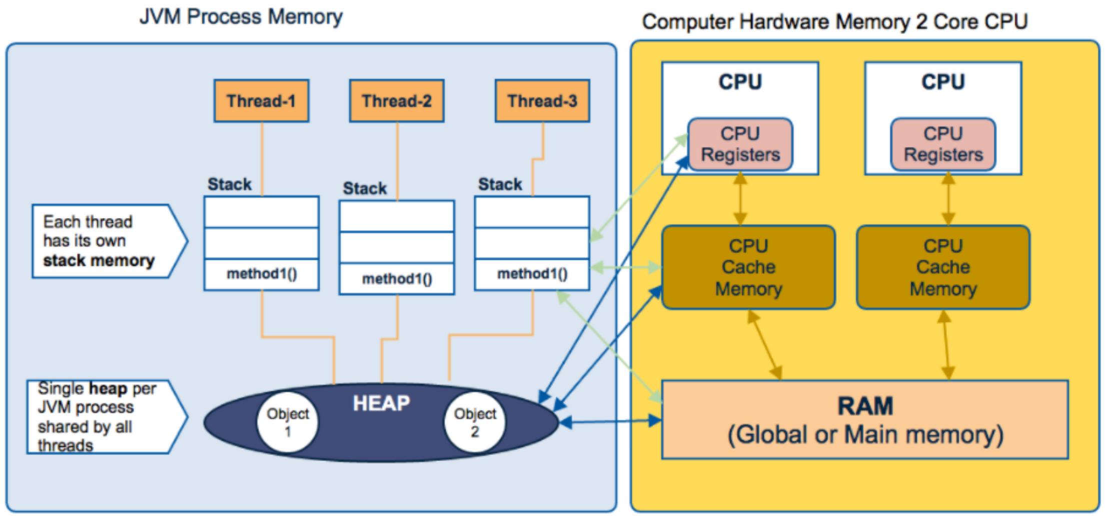
3-way: Asynchronous (from outside looks like **2b** but works differently inside, cooperative scheduling happens)

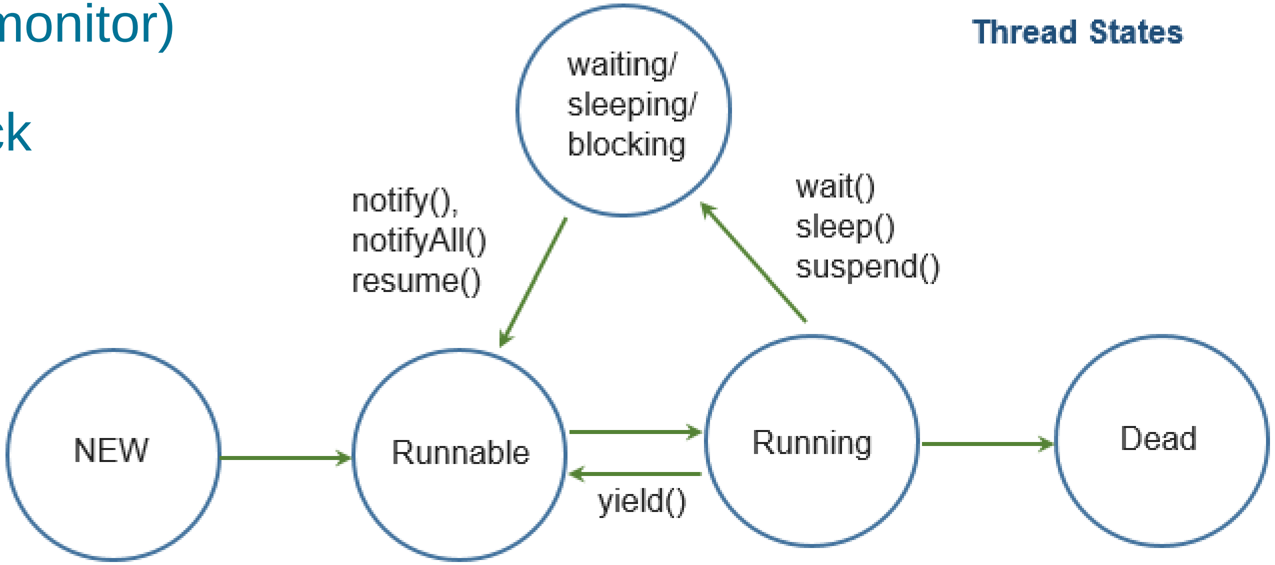


Which solution is faster?

Java Thread Core API. Thread Synchronization

- **Java 1.0, Jan. 1996**
- Core APIs -   Object, Thread, Runnable.
- Thread creation and states. Worker & Task.
- ThreadScheduler, priority
- Thread Synchronization (method, block, monitor)
- Thread Race Condition, Visibility, Deadlock
- Volatile, re-ordering
- Threads interactions (producer/consumer)



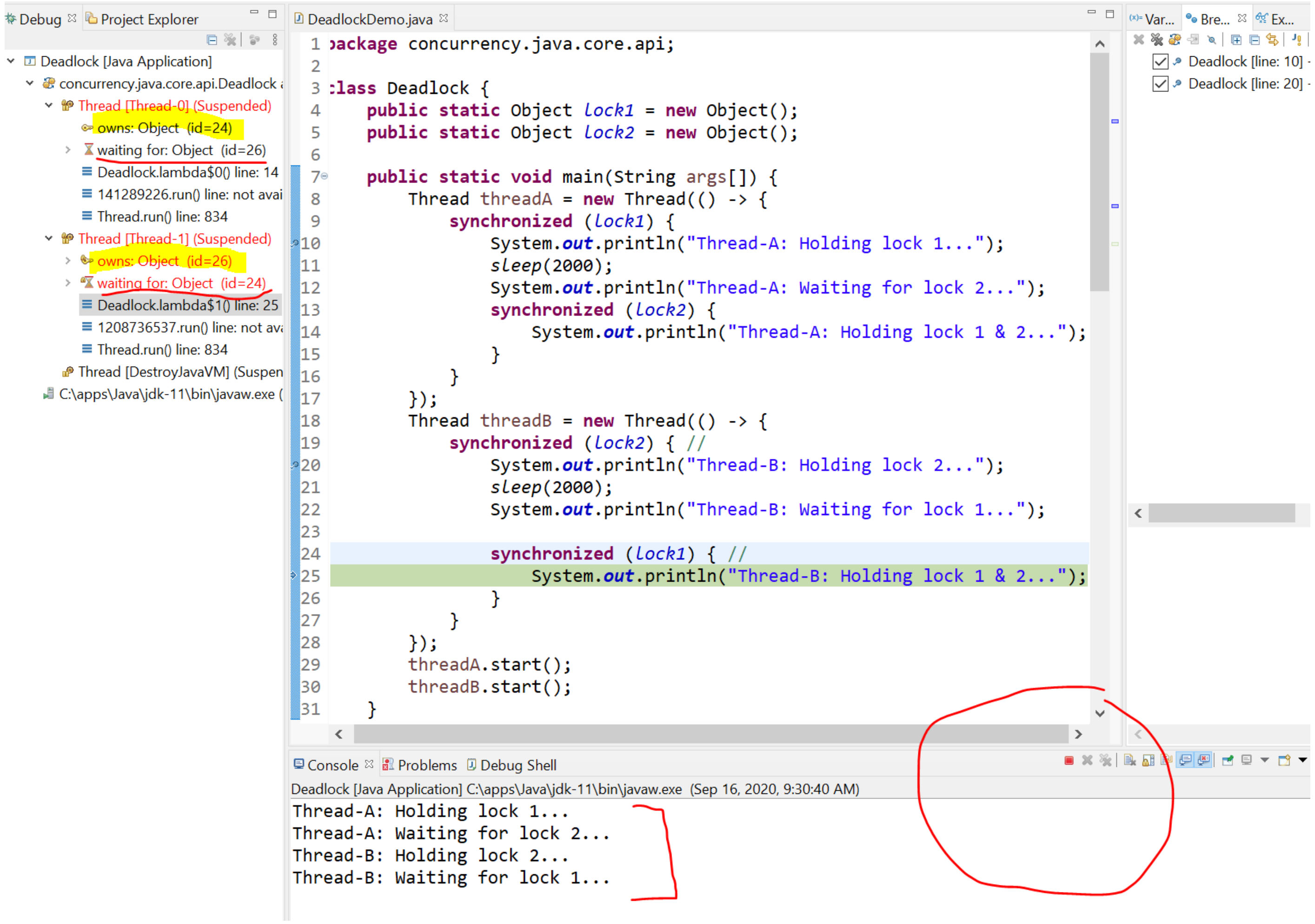


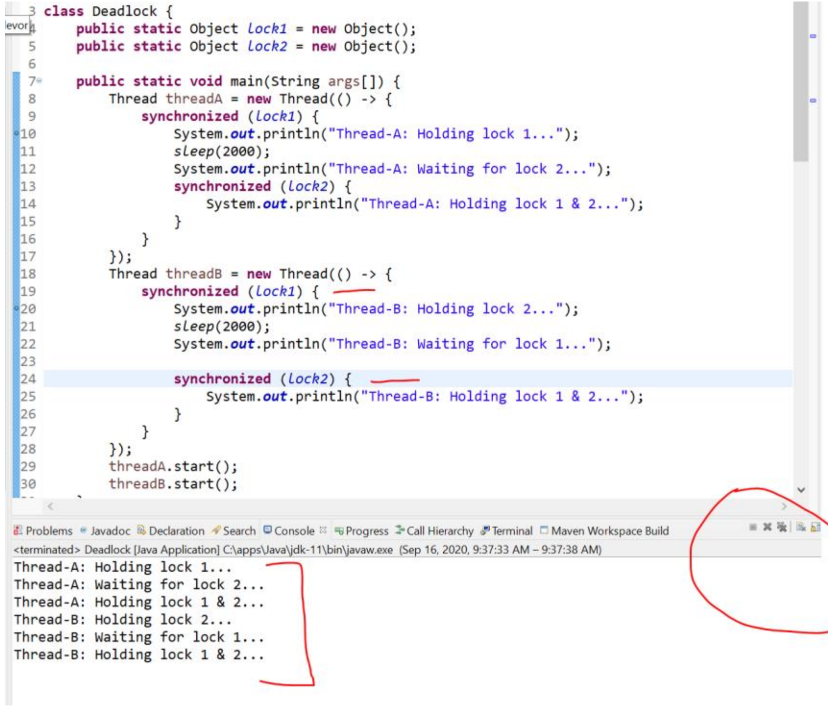
BROADCOM

```java
//1-way
Thread t1 = new MyThread();
t1.start();

//2-way
MyRunnable myRunnable = new MyRunnable();
Thread t2 = new Thread(myRunnable);
t2.start();

//3-way
Thread t3 = new Thread(new Runnable() {
    @Override
    public void run() {
        printInfo();
    }
});
t3.start();

//4-way
Runnable r = () -> {
    printInfo();
};
Thread t4 = new Thread(r); // Thread(r, "Thread Four");
```

```java
class MyThread extends Thread {

    @Override
    public void run() {
        AnIntroThreadBasicsDemo.printInfo();

    }

}
```

```java
class MyRunnable implements Runnable {

    private boolean terminated;

    @Override
    public void run() {
        String name = AnIntroThreadBasicsDemo.getCurrentThreadName();
        System.out.println(name + " is running");
        while (!isTerminated()) {
            System.out.println(name + "  is sleeping");
            AnIntroThreadBasicsDemo.sleep(1000);
        }

        System.out.println(name + " FINISHED");
    }

    public synchronized boolean isTerminated() {
        return terminated;
    }

    public synchronized void terminate() {
        this.terminated = true;
    }
}
```



Debug
Project Explorer
Deadlock [Java Application]
concurrency.java.core.api.Deadlock
Thread [Thread-0] (Suspended)
owns: Object (id=24)
waiting for: Object (id=26)
Deadlock.lambda$0() line: 14
141289226.run() line: not avai
Thread.run() line: 834
Thread [Thread-1] (Suspended)
owns: Object (id=26)
waiting for: Object (id=24)
Deadlock.lambda$1() line: 25
1208736537.run() line: not ava
Thread.run() line: 834
Thread [DestroyJavaVM] (Suspen
C:\apps\Java\jdk-11\bin\javaw.exe (
DeadlockDemo.java
package concurrency.java.core.api;
class Deadlock {
public static Object lock1 = new Object();
public static Object lock2 = new Object();
public static void main(String args[]) {
Thread threadA = new Thread(() -> {
synchronized (lock1) {
System.out.println("Thread-A: Holding lock 1...");
sleep(2000);
System.out.println("Thread-A: Waiting for lock 2...");
synchronized (lock2) {
System.out.println("Thread-A: Holding lock 1 & 2...");
}
}
});
Thread threadB = new Thread(() -> {
synchronized (lock2) { //
System.out.println("Thread-B: Holding lock 2...");
sleep(2000);
System.out.println("Thread-B: Waiting for lock 1...");
synchronized (lock1) { //
System.out.println("Thread-B: Holding lock 1 & 2...");
}
}
});
threadA.start();
threadB.start();
}
Var...
Bre...
Ex...
Deadlock [line: 10]
Deadlock [line: 20]
Console
Problems
Debug Shell
Deadlock [Java Application] C:\apps\Java\jdk-11\bin\javaw.exe (Sep 16, 2020, 9:30:40 AM)
Thread-A: Holding lock 1...
Thread-A: Waiting for lock 2...
Thread-B: Holding lock 2...
Thread-B: Waiting for lock 1...


class Deadlock {
public static Object lock1 = new Object();
public static Object lock2 = new Object();
public static void main(String args[]) {
Thread threadA = new Thread(() -> {
synchronized (lock1) {
System.out.println("Thread-A: Holding lock 1...");
sleep(2000);
System.out.println("Thread-A: Waiting for lock 2...");
synchronized (lock2) {
System.out.println("Thread-A: Holding lock 1 & 2...");
}
}
});
Thread threadB = new Thread(() -> {
synchronized (lock1) {
System.out.println("Thread-B: Holding lock 2...");
sleep(2000);
System.out.println("Thread-B: Waiting for lock 1...");
synchronized (lock2) {
System.out.println("Thread-B: Holding lock 1 & 2...");
}
}
});
threadA.start();
threadB.start();
Problems Javadoc Declaration Search Console Progress Call Hierarchy Terminal Maven Workspace Build
<terminated> Deadlock [Java Application] C:\apps\Java\jdk-11\bin\javaw.exe (Sep 16, 2020, 9:37:33 AM – 9:37:38 AM)
Thread-A: Holding lock 1...
Thread-A: Waiting for lock 2...
Thread-A: Holding lock 1 & 2...
Thread-B: Holding lock 2...
Thread-B: Waiting for lock 1...
Thread-B: Holding lock 1 & 2...

CodeTime

```java
public class ProducerConsumerDemo {

    /**
     * Producer can not publish if buffer is full and consumer can not consume if it is empty.
     * So, always keep buffer under control to prevent race condition.
     */
    private static int[] buffer;
    private static int count;

    private static Object monitor = new Object();
```

```java
static class Producer {

    public void produce() {
        synchronized (monitor) {
            if (isFull(buffer)) {
                try {
                    monitor.wait(); //release KEY, will be available for consumer
                    Thread currentThread = Thread.currentThread();
                    System.out.println(currentThread.getName() + " state: " + currentThread.getState());
                } catch (InterruptedException e) {
                    System.err.println(e.getMessage());
                }
            }
            buffer[count++] = 1; //without synchronization causes race condition
            monitor.notifyAll();
        }
    }
}
```

```java
static class Consumer {

    public void consume() {
        synchronized (monitor) {
            if (isEmpty(buffer)) {
                try {
                    monitor.wait();
                    Thread currentThread = Thread.currentThread();
                    System.out.println(currentThread.getName() + " state: " + currentThread.getState());
                } catch (InterruptedException e) {
                    System.err.println(e.getMessage());
                }
            }
            buffer[--count] = 0; //without synchronization causes race condition
            monitor.notifyAll();
        }
    }
}
```


BROADCOM

Java Synchronization limitations. Thread Safety

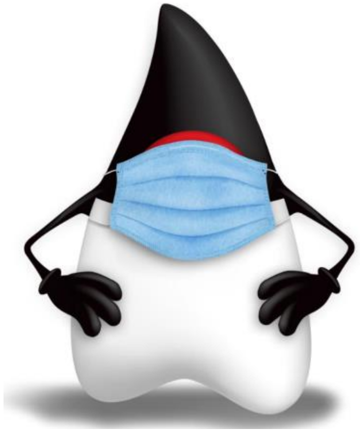
- Complex design (thread interaction, error detection), Context Switching Overhead, Resource Consumption
- Race condition, Invisible writes, Deadlocks, Crash, Starvation, nested monitor lockout
- Performance overhead - intrinsic lock

Thread Safety Techniques

- Use Stateless and Immutable (shares states: immutable class, or clojure) implementations
- Use java.util.concurrent utilities, e.g. prefer Concurrent Collections over Synchronized Collections
- Avoid using Strings or reusable obj. for locking purposes (cached)
- Volatile Fields - prevents visibility issue.

BROADCOM

Java Concurrency API - (2004, 2007, 2014, ..)



- **Utility Classes** - ThreadLocal, ThreadLocalRandom
- **Locks -** ReentrantLock, StampedLock (Java 8), ReadWriteLock, Atomic Variables, ..
- **Synchronizers** - Semaphore, CountDownLatch, CyclicBarrier, ReentrantLock
- **Concurrent Collections** - ConcurrentHashMap, BlockingQueue, ..
- **ExecutorService** - represents an async-exec mechanism, gives a task(Callable) gets a Future
- **Thread Pools**: CachedThreadPool, FixedThreadPool, ScheduledThreadPool, SingleThreadExecutor
- **ForkJoinPool** - provides parallel mechanism for cpu intensive calc, based on work stealing alg.
- **Customizable Thread pools** - use ThreadPoolExecutor | ScheduledThreadPoolExecutor

```java
public class ExecutorServiceDemo {
    public static void main(String[] args) {

        ExecutorService es = Executors.newFixedThreadPool(3);
        BankAccount ba = new BankAccount(100);
        for (int i = 0; i < 5; i++) {
            Cashier work = new Cashier(ba, OperationType.DEPOSIT, 20);
            Cashier work2 = new Cashier(ba, OperationType.WITHDRAWAL, 10);
            es.submit(work);
            es.submit(work2);

        }
    }
}
```

```java
class MyThreadPoolExecutor {

    public static ExecutorService getMyExecutorService(int corePoolSize, int maxPoolSize, long keepAliveTime) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                corePoolSize, maxPoolSize, keepAliveTime, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
        //executor.setMaximumPoolSize(8);
        return executor;
    }

    public static ScheduledThreadPoolExecutor getMyScheduledExecutorService() {
        return new ScheduledThreadPoolExecutor(4);
    }

}
```

```java
var boundedQueue = new ArrayBlockingQueue<Runnable>(1000);
new ThreadPoolExecutor(10, 20, 60, SECONDS, boundedQueue, new AbortPolicy());
```

Enhanced Java Concurrency (new era)

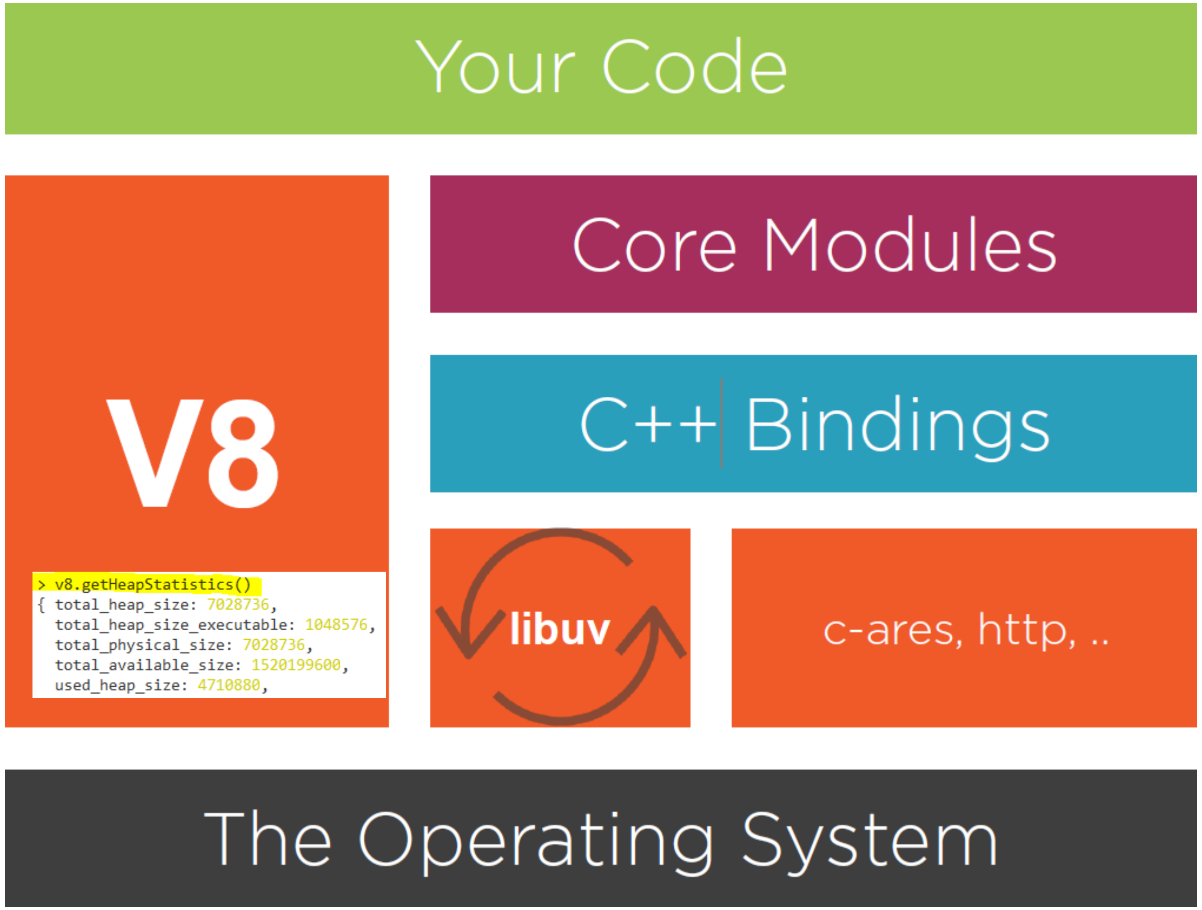
- **ThreadJoinPool** (Java 7) – ForkJoin Framework
- **Parallel Streams** - Java 8 Streams and their parallel processing uses
- **Composing Futures** - CompletableFuture (Java 8, 9) support non-blocking operations
- **Reactive Programming** - creating systems that are responsive to events (event-driven)
- **Concepts behind** CompletableFuture and reactive programming
- **Manifesto of Reactive** – responsive, resilient, elastic, and message-driven
- **Reactive Steams API** – SubmissionPublisher, Flow [Publisher, Subscriber, Subscription, Processor]
- **httpclient Java11** – embraces concurrency and reactive programming ideas
- **Reactive libraries** (non-blocking): Akka, RxJava, Reactor
- **Reactive frameworks**: Netty, Vert.x, Spring(Webflux, Spring Cloud …), Kafka, RabbitMQ


```java
public static void main(String[] args) throws IOException, InterruptedException {
    httpClient = HttpClient.newHttpClient();
    List<CompletableFuture<String>> completableFutureStringListResponse = Files
            .lines(Path
                    .of(DOMAINS_TXT))
            .map(F_HttpClientAsyncronousDemo::validateLink).collect(Collectors.toList());
    completableFutureStringListResponse.stream().map(CompletableFuture::join).forEach(System.out::println);
}
```

```java
public static void main(String[] args) throws IOException, InterruptedException {

    System.out.println("Async tasks run in parallel by 5 therads .. ");
    httpClient = HttpClient.newBuilder().followRedirects(Redirect.NORMAL).connectTimeout(Duration.ofSeconds(5))
            .executor(Executors.newFixedThreadPool(5)).build();

    List<CompletableFuture<String>> completableFutureStringListResponse = Files
            .lines(Path.of(DOMAINS_TXT))
            .map(H_HttpClientConfigDemo::validateLink).collect(Collectors.toList());
    completableFutureStringListResponse.stream().map(CompletableFuture::join).forEach(System.out::println);
}

private static CompletableFuture<String> validateLink(String link) {
    HttpRequest httpRequest = HttpRequest.newBuilder(URI.create(link)).GET().build();
    return httpClient.sendAsync(httpRequest, HttpResponse.BodyHandlers.discarding())
            .thenApply(
                    asynResult -> 200 == asynResult.statusCode() ? link + " access OK  " : link + " access Failed'
            .exceptionally(e -> " Error occured " + "once accessing to " + link + ", reson is: " + e.getMessage()

}
```

```java
@Async
public CompletableFuture<User> findUser(String userName) throws InterruptedException {
    log.info("Looking up " + userName);
    User user = restTemplate.getForObject(String.format(URL, userName), User.class);
    // delay to demon
    // Thread.sleep(3000L);
    return CompletableFuture.completedFuture(user);
}
```


Node.JS – 2009, 2018



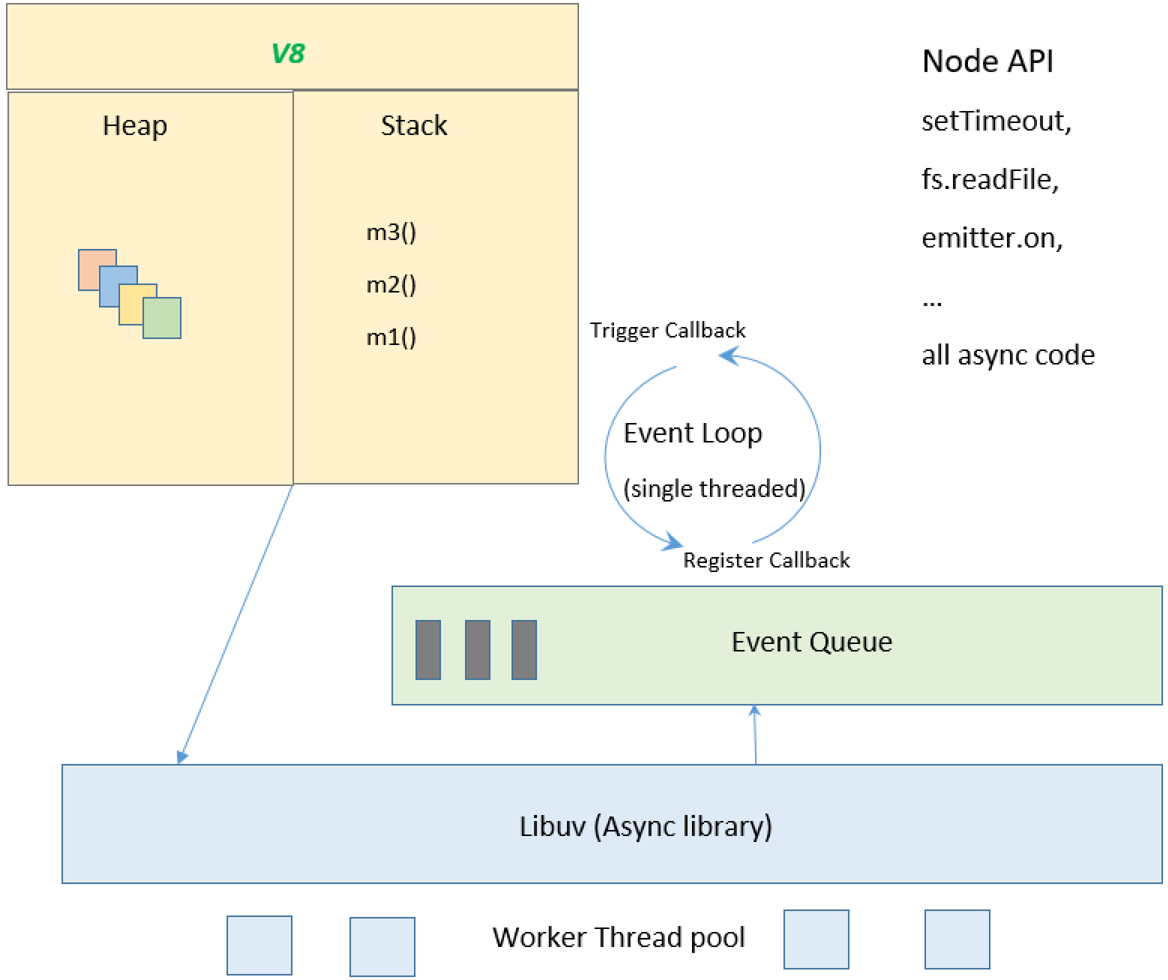
Benefits

Node.js - a runtime environment based on Chrome's V8
Single threaded architecture – used as opportunity
Robust technology stack – built-in modules, providing rich features via asynchronous APIs
Single threaded - no race condition, locking issues, ..
Non-Blocking I/O - Not waiting till I/O operation is complete.
Asynchronous - Handle dependent code later once its complete.

Drawbacks

Low performance on heavy computation - CPU bound tasks.
Callback hell issue - asynchronous nature relies heavily on callbacks …

Event Loop & Event Emitters



- **Event Loop** (single threaded) is an Event Dispatcher
- Event Loop adds its own queues to be processed by the **libuv thread pool**.
- **Event Emitters** - instances of EventEmitter. E.g. Streams, Files, … built-in events
- If task is written one of asynchronous ways below, then it will be handled by Event Loop afford.

  Callbacks | Promises | Async/Await

Libuv by default creates thread pool with **4** threads max-size is **128**, can be tuned at startup time

CodeTime

```javascript
const fs = require('fs');

// Asynchronous Form:
fs.readFile(__filename, (err, data) => {
  if (err) throw err;
  // process data
});


// Synchronous Form:
const data = fs.readFileSync(__filename);
// exceptions are immediately thrown
// process data
```

```javascript
const https = require('https');
https.get('https://www.javascript.com/', (resp) => {
  let data = '';
  // A chunk of data has been received.
  resp.on('data', (chunk) => {
    data += chunk;
  });
  // The whole response has been received.
  resp.on('end', () => {
    console.log(data.length);
  });
}).on("error", (err) => {
  console.log("Error: " + err.message);
});
```

```javascript
const https = require('https');
function fetch (url) {
  return new Promise((resolve, reject) => {
    https.get(url, (res) => {
      let data = '';
      res.on('data', (rd) => data = data + rd);
      res.on('end', () => resolve(data));
      res.on('error', reject);
    });
  });
}
//simpler than callback
(async function read() {
  const data = await fetch('https://www.javascript.com/');
  console.log(data.length);
})();
```


Node.js Concurrency - Worker Threads

- Before Worker Threads:
- All time consuming tasks are not considered I/O (CPU intensive)
- CPU intensive operations blocks main thread
- Never execute anything from event queue of any pending I/O tasks
- Used child_process | cluster | Napa.js for CPU intensive tasks

- Worker threads introduced in 2018, v12LTS – has stable "worker_thread"
- new Worker(..) - represents an independent JS execution thread
- Each worker owns instance of V8 and EventLoop by V8 isolate.
- Unlike child process or cluster workers share memory
- Creating Worker instance inside of other Worker is possible
- Two-way communication like in like WebWorkers, cluster
- Two ways using workers - (new threads for each incoming task or parent keeps worker live (worker pool))

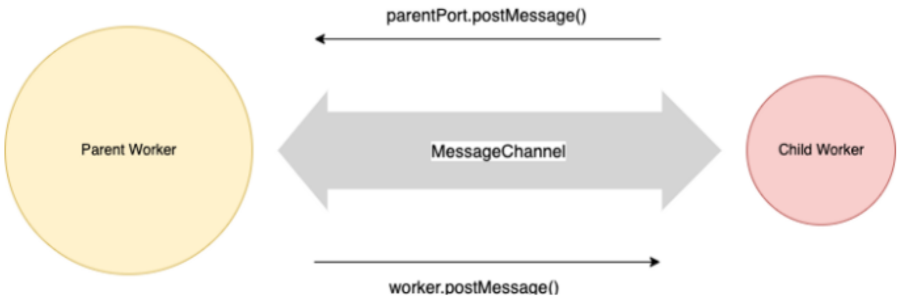


Diagram 1: Message Channel between the parent and the child workers

CodeTime

```javascript
const got = require('got');
//npm install got@10.6.0
(async () => {
    try {
        //const response = await got('https://www.javascript.com/', { json: true });
        const response = await got('https://www.javascript.com/');
        console.log(response.body.length);
        /*console.log(response.body.url);
        console.log(response.body.explanation);*/
    } catch (error) {
        console.log(error.response.body);
    }
})();
```

```javascript
//npm install axios@0.19.2
const axios = require('axios');

(async () => {
    try {
        const [response1, response2] = await axios.all([
            axios.get('https://www.javascript.com/'),
            axios.get('http://www.sahet.net/')
        ]);
        console.log(response1.data);
        console.log(response2.data);
    } catch (error) {
        console.log(error.response.body);
    }
})();
```

```javascript
//npm install node-fetch@2.6.0
const fetch = require('node-fetch');

(async () => {
    try {
        const response = await fetch('https://www.javascript.com/').then(response => response.json());
        /*console.log(json.url);
    } catch (error) {
        console.log(error.response.body);
    }
})();
```


BROADCOM

Computation Measures Matrix

# of URLS	Java sync (java 11 httpclient sync)	Java async (java 11 httpclient async)	Java executors (completable future in parallel with 4 threads)	Node sync	Node async (4 worker threads)	Node axios	Node got
10	4706 ms 5585 ms **5672** ms	2464 ms 3069 ms 3671 ms	2534 ms 3067 ms 3968 ms	**250** ms 268 ms 606 ms	819 ms 945 ms 2533 ms	1813 ms 1934 ms 1961 ms	922 ms 1219 ms 1236 ms
100	32261 ms 38175 ms **43802** ms	**7985** ms 9111 ms 11413 ms	10058 ms 11245 ms 14621 ms	11111 ms 12707 ms 16683 ms	10557 ms 11297 ms 11961 ms	10972 ms 11072 ms 12021 ms	11072 ms 13313 ms 15241 ms
500	Poor performance (takes so long)	12906 ms 22040 ms 33896 ms (for 500 lines)	12146 ms 16694 ms 23411 ms (for 500 lines)		11104 ms 12331ms 12719 ms (for 449 lines)	Fails with HTTP 429 Too Many Requests	After 356th line crashed
100 (CPU intensive operation)	Poor performance (takes so long)	94052 ms 94266 ms 94781 ms	**79055** ms 90532 ms 91739 ms	Poor performance (takes so long, ..) Just for 10 line **76489** ms	Poor performance (takes so long, after 30th line crashed)	Poor performance (takes so long, after 42th line crashed)	Poor performance (takes so long, after 35th line crashed)

Task1: Read a file with URLs, parse it and validate the connection
Task2: Read a file with URLs, parse it and validate the connection with CPU intensive calculation
Result: Best, Good and Worst results

BROADCOM

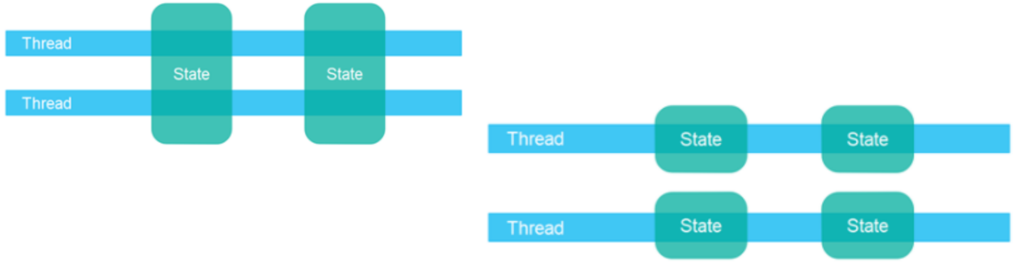
Concurrency Measures Matrix

Performance Factors	JAVA	Node.JS
Concurrency model	Thread and event-based	Event-based (event-loop)
Tuned Thread pool	More predefined Executor options, also can be tuned	Single non-blocking thread, Event loops threads and Worker threads
Options for different use-cases	Semaphores, CyclicBarrier,	no locks in nodejs
Cooperative vs. Preemptive scheduling	kernel-level threading model, with preemptive scheduler. Eg: Java (JVM), C, Rust Hybrid threading model, preemptive + cooperative scheduling. E.g. RxJava, Kotlin coroutines, Akka, uThreads, Go (goroutines)	user-level threading model, cooperative scheduling Eg: Node.js, Twisted, EventMachine, Lwt
IO model	Blocking, and non-blocking	non-blocking by default, but later added support for sync operations
Application Developer	Nature of project - involved as full stack in many technologies and languages (FE, BE, DB)	Full stack (one language for FE(TS,JS), BE (Node.js), DB (Mongo, Postgres))
Concurrency on non-CPU intensive tasks	good performance	good performance (not axios, got)
Concurrency on CPU intensive heavy tasks	good performance	poor performance, mostly crashes

More factors that impacts performance: Memory Management, Application Design, Data Structures, Algorithms, Concurrency, Network Communication, Scalability, Hardware capability (CPU, RAM, disk)

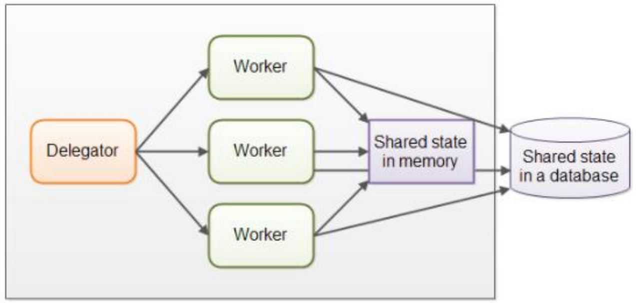
BROADCOM

Concurrency Models [3]



Shared State vs. Separate State

- Shared state: race conditions and deadlock may occur
- Separate state: via exchanges immutable objects/copies

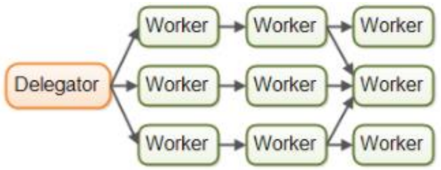


Parallel Workers

- Incoming tasks(jobs) are assigned to parallel workers (threads) on diff. CPUs.
- Java concurrency.util.concurrent package is designed with this model.
- Disadvantages: shared state,
- workers wait each other. Also nondeterministic job ordering and stateless workers.

Shared Nothing Concurrency Model (also called Reactive or Event Driven)

- Workers (each has duty) are organized like assembly line in a factory.
- Stateful workers. Job ordering is possible,
- Threads not share state, designed to use **non-blocking IO**.
- Assembly lines varies, and also job maybe executed concurrently
- Disadvantages: Job execution in multiple workers, level of code complexity, callback hell
- Reactive, Event Driven Systems:
  Based on **Java** CompletableFuture and ForkAndJoin: akka
  Based on **Node.js** event-loop: axios, got, superagent, …



Functional Parallelism

Using function (agent or actor) calls (like sending messages) in independent way executed on separate CPU.

- Disadvantage: learning curve, if not used correctly (splitting) performance slow down – adding more threads to pool
- Based on **Java7** ForkAndJoin and Java8 Parallel Streams: rxjava

References

- [1] https://docs.oracle.com/javase/tutorial/essential/concurrency/
- [2] https://howtodoinjava.com/java/multi-threading/
- [3] http://tutorials.jenkov.com/java-concurrency/concurrency-models.html
- [4] https://www.baeldung.com/java-thread-safety
- [5] https://jscomplete.com/learn/node-beyond-basics
- [6] https://nodejs.org/fa/docs/guides/event-loop-timers-and-nexttick/
- [7] https://blog.insiderattack.net/deep-dive-into-worker-threads-in-node-js-e75e10546b11

- Open Source repository: https://github.com/asatklichov/multithreading-java-vs-nodejs

BROADCOM®

Thank You