Asynchronous Programming

ESUG 2024, Lille, France

James Foster

Agenda

- Asynchronous Programming and Synchronization
- Exploring Other Languages
- Exploring Smalltalk Implementations
- Observations
- Questions

Asynchronous Programming and Synchronization

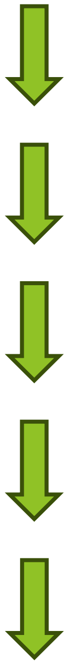
Futures and Promises

Synchronous: Blocking

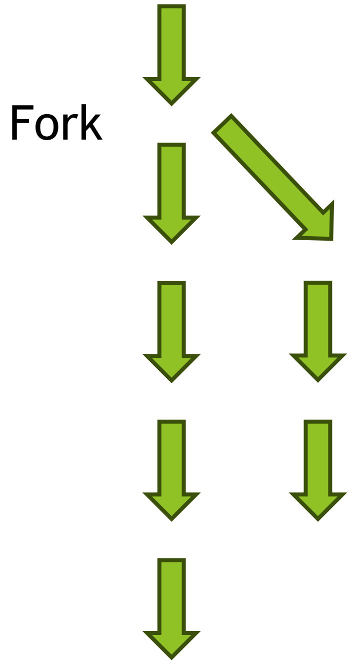
- Blocking & Sequential

Asynchronous: Nonblocking and Parallel

- Blocking & Sequential

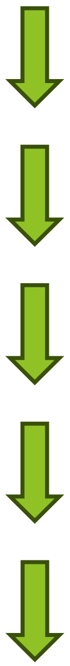


- Nonblocking & Parallel

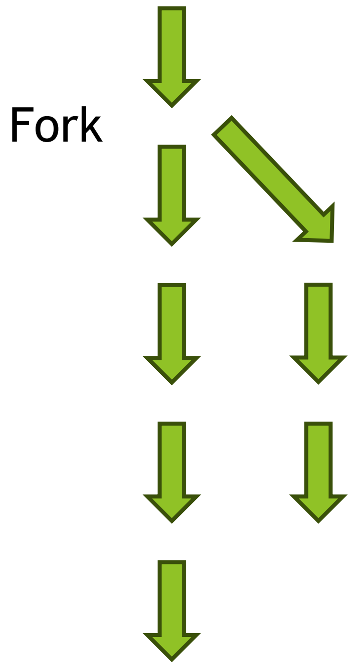


Asynchronous:
Nonblocking and Concurrent

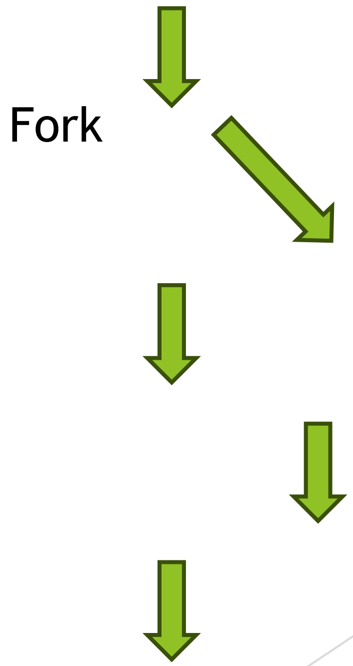
- Blocking & Sequential



- Nonblocking & Parallel

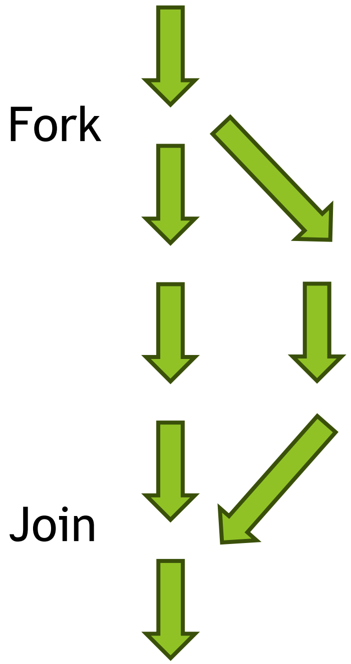


- Nonblocking & Concurrent

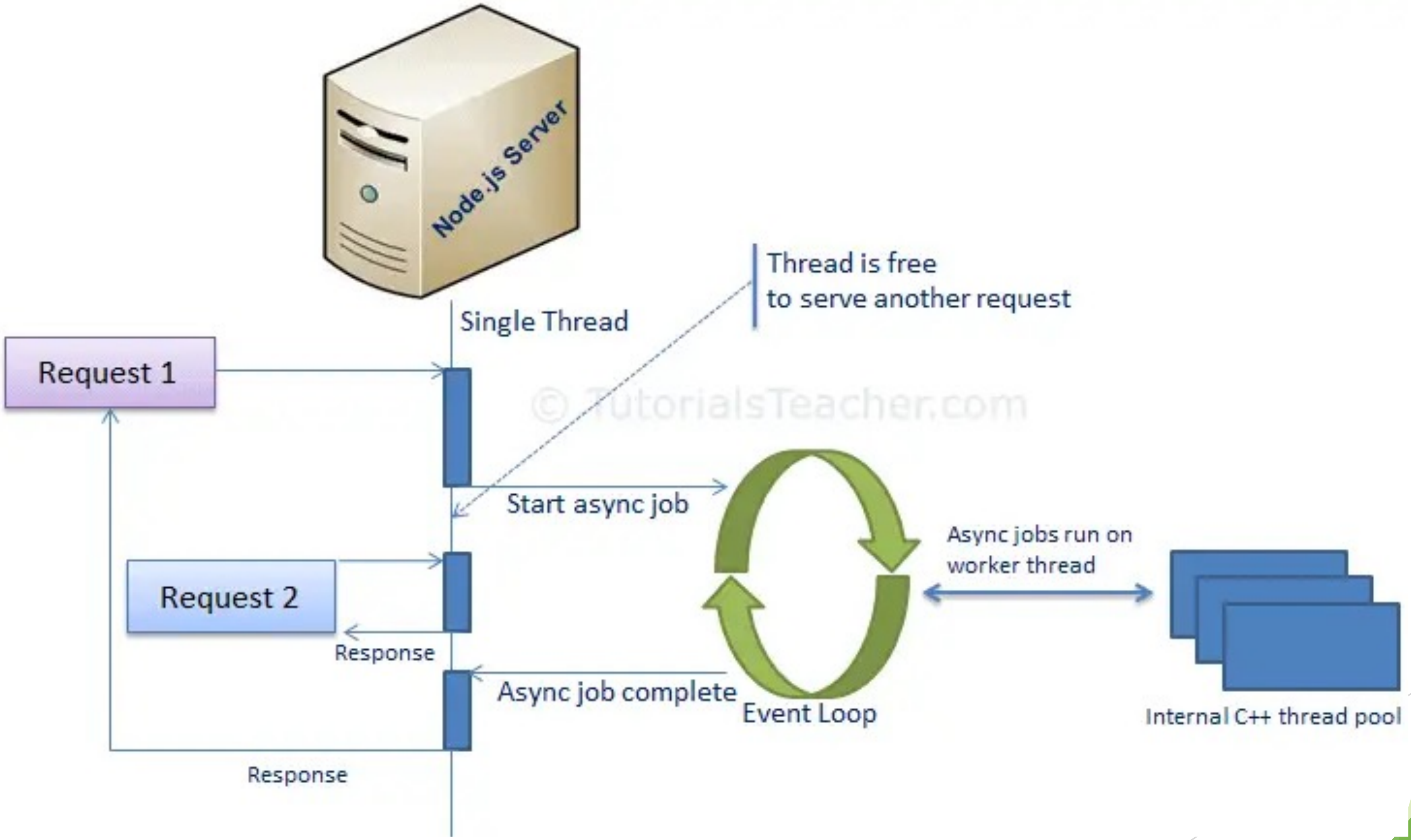


Synchronization

- Coordinating multiple processes to join up.



Web Server


Node.js Server
Thread is free
to serve another request
Single Thread
Request 1
© TutorialsTeacher.com
Start async job
Async jobs run on
worker thread
Request 2
Response
Async job complete
Event Loop
Internal C++ thread pool
Response

Asynchronous Programming in GemStone

- Demo of WebGS with parallel sessions

Futures and Promises

- "In computer science, future, promise, delay, and deferred refer to constructs used for synchronizing program execution in some concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is not yet complete."
  - — https://en.wikipedia.org/wiki/Futures_and_promises

Exploring Other Languages

Futures and Promises

JavaScript: Promises

- A Promise is in one of these states:
  - **Pending**: Initial state, neither fulfilled nor rejected.
  - **Fulfilled**: The operation completed successfully.
  - **Rejected**: The operation failed.

JavaScript: Example

```
let promise = new Promise(function(resolve, reject) {
    // Asynchronous operation here
    if (/* operation successful */) {
        resolve(value); // Resolve with a value
    } else {
        reject(error); // Reject with an error
    }
});

promise.then(
    function(value) { /* handle a successful operation */ },
    function(error) { /* handle an error */ }
);
```

JavaScript: Summary

- **Creating a Promise**: The Promise constructor is used to create a promise. It takes a function (executor) that should start an asynchronous operation and eventually call either the resolve (to indicate success) or reject (to indicate failure) function to settle the promise.
- **Consuming a Promise**: The .then() method is used to attach callbacks to handle the fulfillment or rejection of the promise. The .catch() method is used to handle rejection, and .finally() method allows you to execute logic regardless of the promise's outcome.
- **Chaining Promises**: Promises can be chained to perform a series of asynchronous operations in sequence. Each .then() returns a new promise, allowing for further methods to be called in sequence.

JavaScript: Conclusion

- Promises are a core part of asynchronous programming in JavaScript, making it easier to work with asynchronous operations by avoiding the complexity of nested callbacks, known as "callback hell."

Python: ThreadPoolExecutor

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

def task(n):
    return n + 1

# Create a ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=5) as executor:
    # Submit tasks to the executor
    futures = [executor.submit(task, i) for i in range(5)]
    # Wait for the futures to complete and get their results
    for future in as_completed(futures):
        print(future.result())
```

Python: asyncio.Future

```python
import asyncio

async def set_after(fut, delay, value):
    # Wait
    await asyncio.sleep(delay)
    # Set the result
    fut.set_result(value)
```

```python
async def main():
    # Create a Future object
    fut = asyncio.Future()

    # Schedule the future
    await set_after(fut, 1, 'hello!')

    # Wait for the future
    print(await fut)

asyncio.run(main())
```

Java: CompletableFuture

```java
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureExample {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        // Create a CompletableFuture
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            try {
                // Simulate a long-running job
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Hello";
        });

        // Chain a computation stage
        CompletableFuture<String> greetingFuture =
            future.thenApply(result -> result + ", World!");

        // Block and get the result
        System.out.println(greetingFuture.get()); // Prints "Hello, World!" after 1 second
    }
}
```

Dart: async and await

- A long running method is (should be!) annotated with `async`.
  - An async method returns a `Future`.
- A callback may be added to a Future to handle:
  - A normal result; or,
  - An error.
- Instead of adding a callback, you can `await` for a Future to complete.
  - This will block, so should *not* be done in the primary (UI) thread.
  - In background threads this allows synchronous (linear) code.

```dart
import 'dart:async';

Future<String> fetchUserOrder() async {
   // Simulate a network request to fetch a user order
   await Future.delayed(Duration(seconds: 2));
   return 'Cappuccino';
}

void main() async {
   print('Fetching user order...');
   try {
      // Wait for the Future to complete and extract its result
      String order = await fetchUserOrder();
      print('Your order is: $order');
   } catch (err) {
      print('Failed to fetch user order: $err');
   }
}
```

Exploring Smalltalk Implementations

Not exhaustive!

VAST Platform

- Modeled on Dart
- Demo

Pharo

- Semaphore approach
- TaskIt package
- Demo

Glamorous Toolkit

- Documentation

Observations

- Application developers
  - Avoid long-running (blocking) tasks in the UI thread.
  - Futures/Promises simplify the handling of asynchronous tasks.
- Library developers
  - Use Futures/Promises for long-running operations (disk, network, etc.)
  - Force application developers to use Futures!

Questions?

- James.Foster@GemTalkSystems.com
- VP of Finance and Operations, GemTalk Systems LLC

- James.Foster@WallaWalla.edu
- Associate Professor of Computer Science, Walla Walla University