CS 4350: Fundamentals of Software Engineering
Lesson 4.1: Concurrent Programming Models

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand

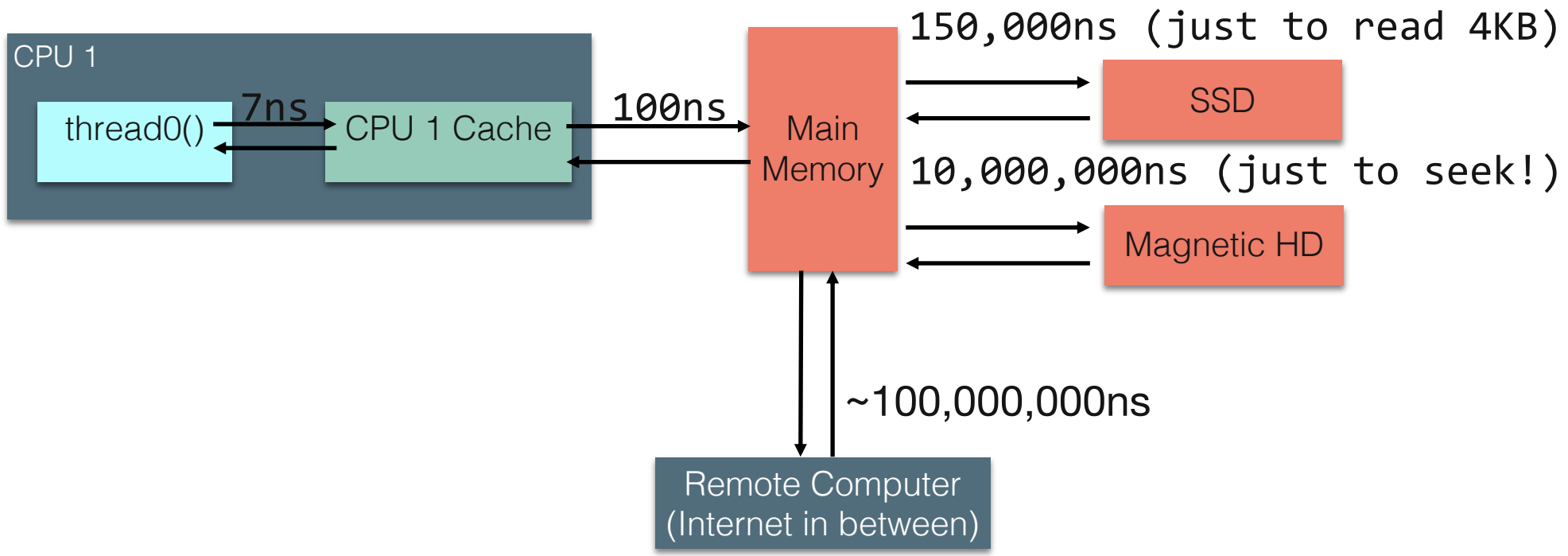
Khoury College of Computer Sciences

Learning Goals for this Lesson

- At the end of this lesson, you should be able to:
  - Explain why almost all programs need to support concurrent actions
  - Understand how to write code that uses asynchronous results using async/await

Masking Latency with Concurrency

Consider: a 1Ghz CPU executes an instruction every 1 ns



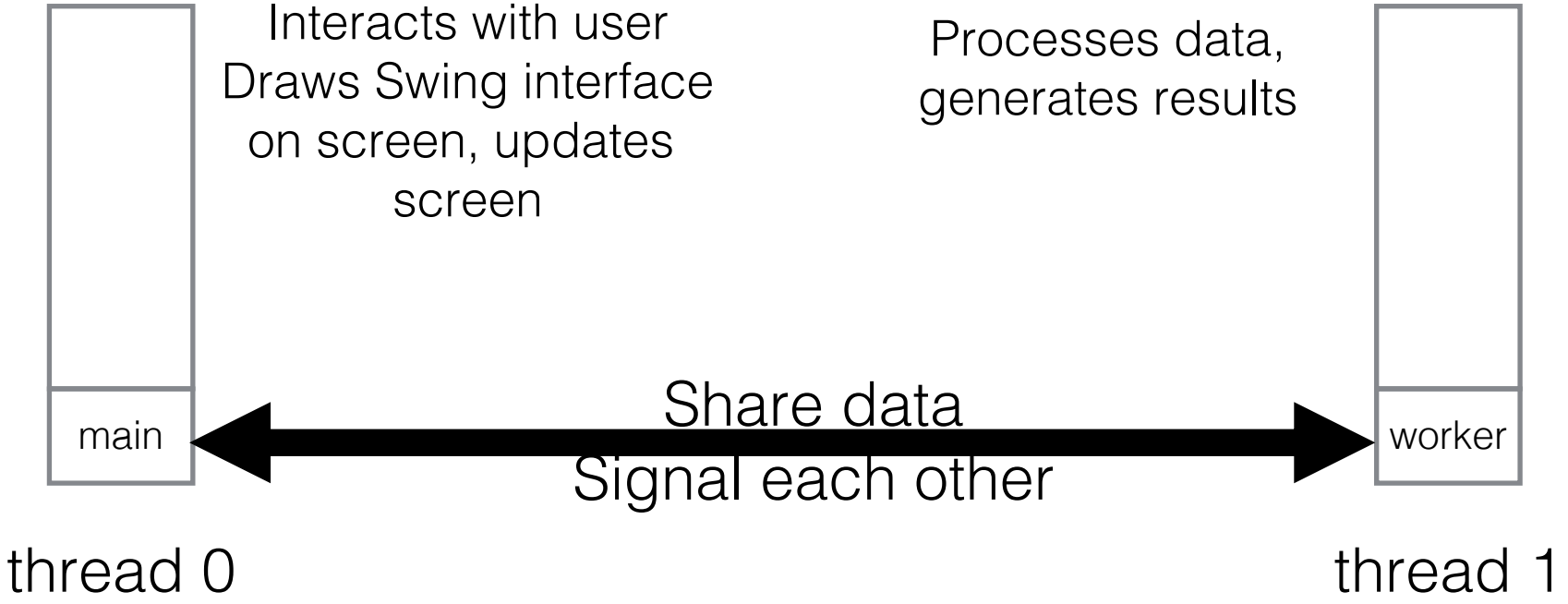
Why Concurrency?

- Maintain an interactive application while…
  - Processing data
  - Communicating with remote hosts
  - Timers that countdown while our app is running
  - Waiting for users to provide input
- Anytime that an app is doing more than one thing at a time, it is asynchronous

Concurrency through Threads

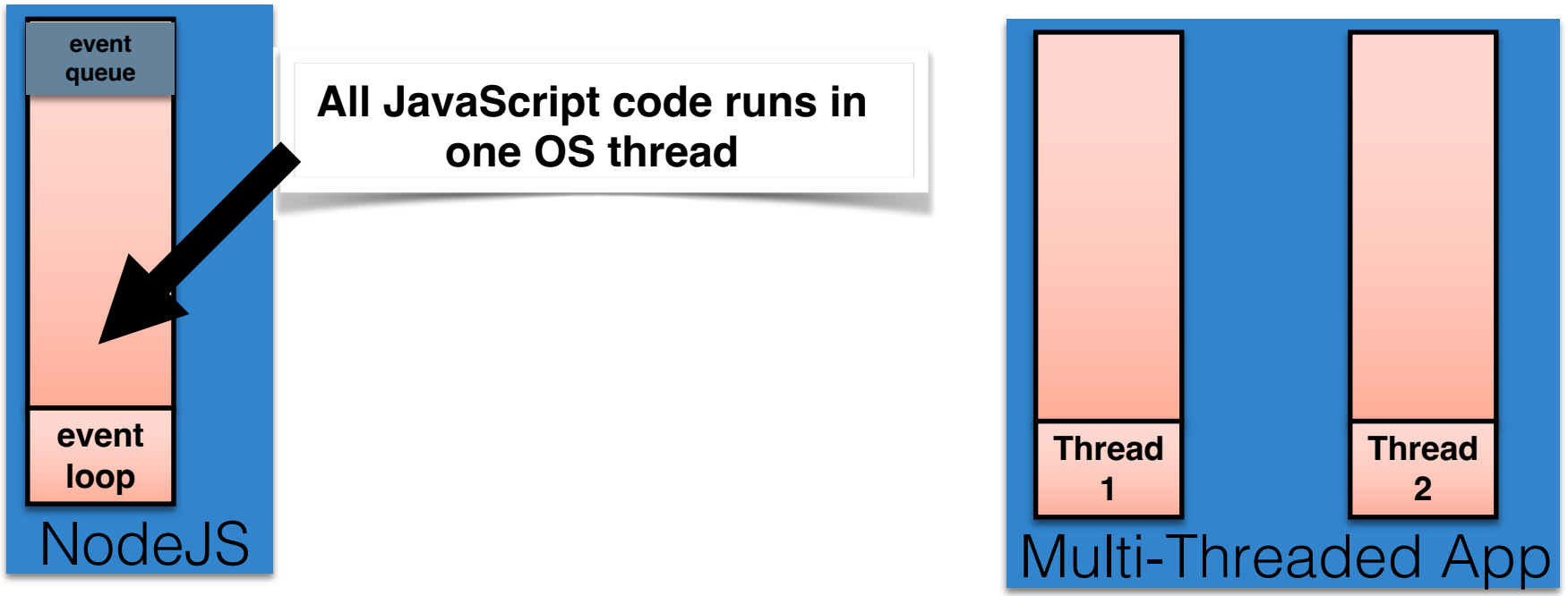
Typical Java Example

- Multi-Threading allows us to do more than one thing at a time
- Physically, through multiple cores and/or OS scheduler
- Example: Process data while interacting with user



Concurrency through Asynchronous Programming

- Everything you write will run in a single thread* (event loop)
- Since you are not sharing data between threads, races don't happen as easily
- Inside of the JS engine: perhaps more threads
- Event loop processes events, and calls your listeners ("event handlers")



Running Asynchronous Example: HTTP Request

```
let nGets = 0;
app.get('/', (req, res) => {
    nGets++;
    res.status(200).send(`This is GET number ${nGets} on the current
server`);
});
```

A Promise is a Representation of a Listener

The “Promise” lets us register a listener for something that will come in the future

To call a function that returns a Promise, you must ‘await’ it, from inside of an ‘async’ function

```
async function makeOneGetRequest(){
    console.log('Making Request');
    const response = await axios.get('https://rest-example.covey.town');
    console.log('Heard back from server');
    console.log(response.data);
}
makeOneGetRequest();
```

Output:

```
Making Request
Heard back from server
This is GET number 1 on the current server
```

Awaiting a Promise Prevents Your Method from Continuing

Example: calling our makeOneGetRequest multiple times with await

```
async function makeThreeSerialRequests(): Promise<void> {
  await makeOneGetRequest();
  await makeOneGetRequest();
  await makeOneGetRequest();
}

makeThreeSerialRequests();
```

Output:

```
Making Request
Heard back from server
This is GET number 2 on the current server
Making Request
Heard back from server
This is GET number 3 on the current server
Making Request
Heard back from server
This is GET number 4 on the current server
```

The Event Loop Resolves Promises

Event Being Processed:

The Event Loop Resolves Promises

Event Being Processed:

Response #3 from covey.town

Are there any listeners registered for this event?

If so, call listener with event

After the listener is finished, repeat

The Event Loop Resolves Promises

Event Being Processed:

Response #1 from covey.town

Are there any listeners registered for this event?

If so, call listener with event

After the listener is finished, repeat

The Event Loop Resolves Promises

Event Being Processed:

Response #2 from covey.town

Are there any listeners registered for this event?

If so, call listener with event

After the listener is finished, repeat

The Event Loop Calls Listeners

- JavaScript (and TypeScript) offer “event driven” concurrency: asynchronous tasks happen in the background, by the language runtime
- Event loop is responsible for dispatching events when they occur
- Main thread for event loop (buried somewhere in NodeJS) :

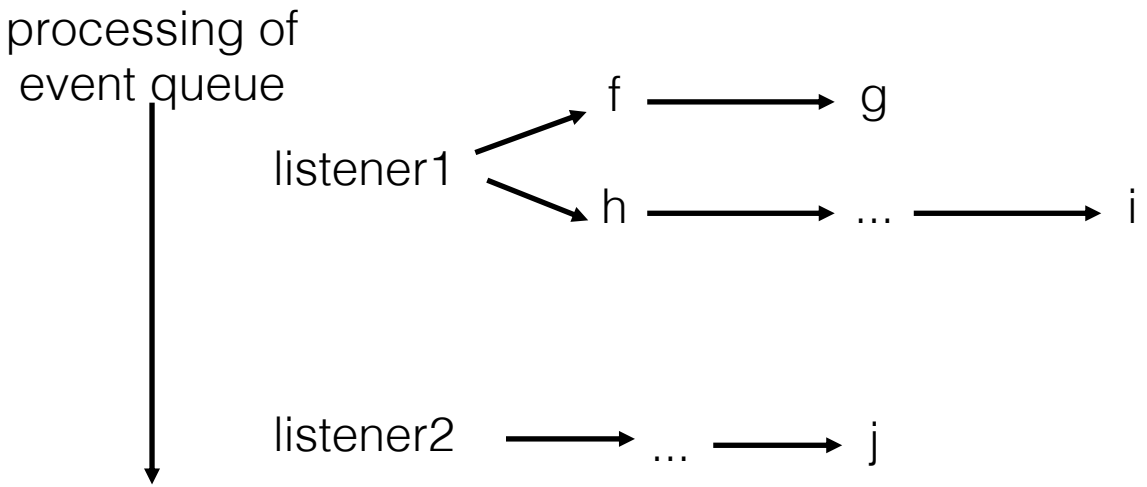
```
while(queue.waitForMessage()){
   queue.processNextMessage();
}
```

- The order of event processing is (in the general sense) unpredictable

Event Handlers “Run To Completion”

AKA: Your code will not be “interrupted”

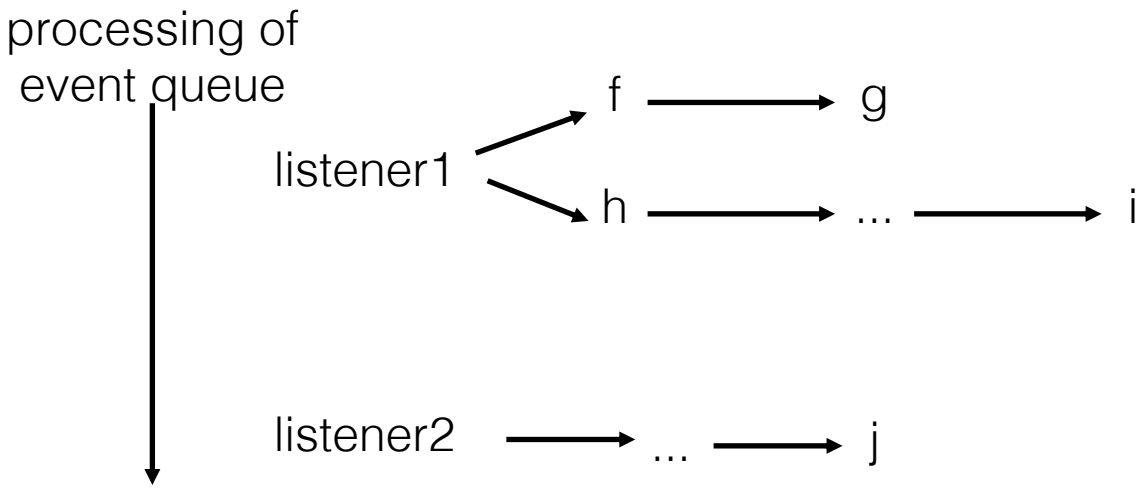
- The listenerhandling an event and the functions that it (transitively) synchronously calls will keep executing until the function finishes.
- The JS engine will not handle the next event until the listener finishes.



Implications of Run-to-Completion

The good news: no interruptions/context switching

No other code will run until you finish (no worries about other threads overwriting your data)



j will not execute until after i

Listeners Complete when they Return or Await

Adding 'async' to our function definition makes it return a Promise!

```
async function makeOneGetRequest(): Promise<void> {
  console.log('1. Making Request');
  const response = await axios.get('https://rest-example.covey.town');
  console.log('2. Heard back from server');
  console.log(response.data);
}

makeOneGetRequest();
console.log('3. All done!');
```

makeOneGetRequest returns the promise immediately upon hitting await!

Output:

```
1. Making Request
3. All done!
2. Heard back from server
This is GET number 5 on the current server
```

Learning Goals for this Lesson

- At the end of this lesson, you should be able to:
  - Explain why almost all programs need to support concurrent actions
  - Understand how to write code that uses asynchronous results using async/await