CS 4530: Fundamentals of Software Engineering

Module 5: Concurrency Patterns in Typescript

Jonathan Bell, Adeel Bhutta, Mitch Wand

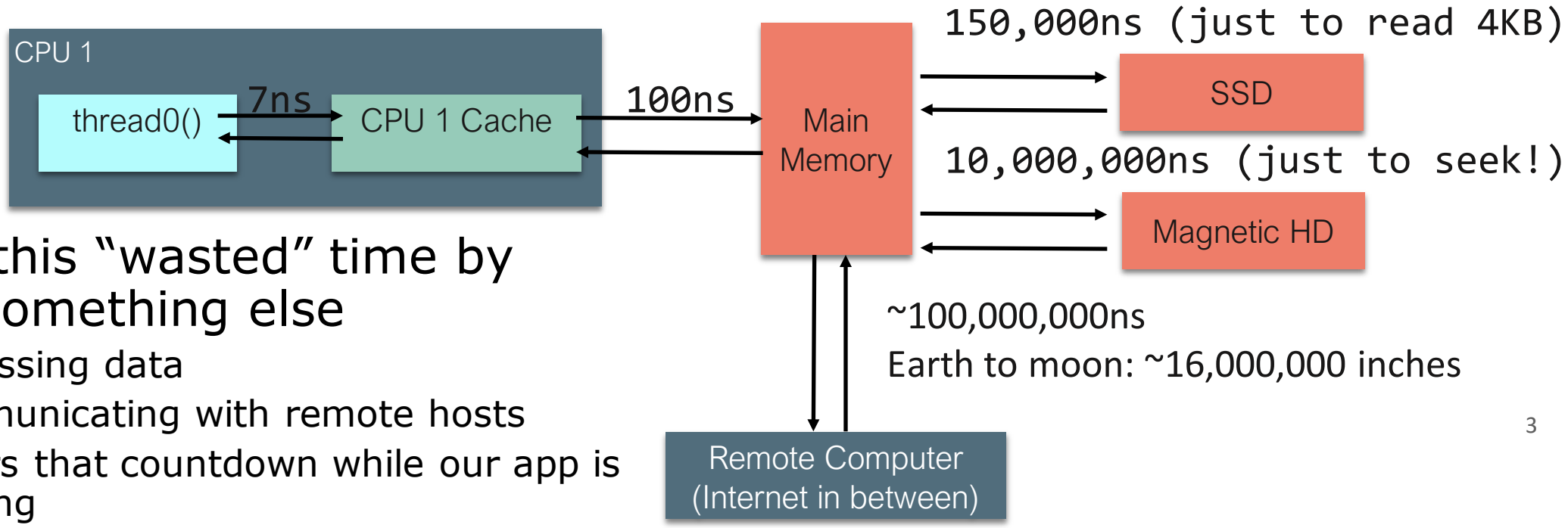
Khoury College of Computer Sciences

Learning Goals for this Lesson

- At the end of this lesson, you should be prepared to:
  - Explain how to achieve concurrency through asynchronous operations and Promise.all in TypeScript.
  - Write asynchronous and concurrent code in TypeScript using async/await and Promise.all.

Masking Latency with Concurrency

- Consider: a 1Ghz CPU executes an instruction every 1 ns
- Almost anything else takes forever (approximately)



- Utilize this "wasted" time by doing something else
  - Processing data
  - Communicating with remote hosts
  - Timers that countdown while our app is running
  - Waiting for users to provide input

Pre-emptive Multiprocessing

- OS manages multiprocessing with multiple threads of execution
- Processes may be interrupted at unpredictable times
- Interprocess communication by shared memory
- Data races abound
- Really, really hard to get right: need critical sections, semaphores, monitors (all that stuff you learned about in op. sys.)

An alternative model: cooperative multiprocessing

- OS manages multiprocessing with multiple threads of execution
- Each thread decides when it should yield to let other threads execute
- Typically via a **yield** or **await** operation

JavaScript/TypeScript implements Cooperative Multiprocessing Using “run-to-completion” semantics

- JS has primitives that allow one computation to start another computation that runs concurrently with the first.
- These are almost always IO operations.
- However, the original computation **always runs to completion.**

Run-to-completion semantics

- A computation runs continuously until it is either suspended or completed.
  - This means that only one of your computations is running at any time (in addition to whatever asynchronous IO is running)
- A computation is suspended when it hits an ‘await’. The runtime system (node.js, for us) chooses what to do next. (In addition to whatever asynchronous IO it may be doing).

Defining a concurrent computation

```
async function makeOneGetRequest(requestNumber:number) {
    const response = await axios.get('https://rest-example.covey.town');
    console.log(`For request ${requestNumber}, server replied: `,
response.data);
}
```

- An async function is a function that creates a concurrent computation.
- Calling the function will tell the operating system to start the computation.
- TS vocabulary: This computation is called a **promise**

This is the address of a server that returns the number of calls that have been made to this server.

One concurrent computation can wait for the result of another one.

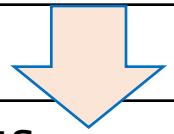
```
async function makeOneGetRequest(requestNumber:number) {
    const response = await axios.get('https://rest-example.covey.town');
    console.log(`For request ${requestNumber}, server replied: `,
response.data);
}
```

- Axios.get is also an async function, so it returns a promise (let's call it **p**)
- The **await** suspends the current computation until the promise **p** returns.
- While the current computation is suspended, other computations (including **p**) can run.

Example:

```
async function makeThreeSimpleRequests() {
    makeOneGetRequest(1);
    makeOneGetRequest(2);
    makeOneGetRequest(3);
    console.log("Three requests made")
}

makeThreeSimpleRequests()
```

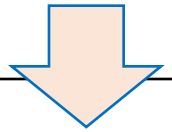


```
$ npx ts-node example2.ts
Three requests made
For request 2, server replied:  This is GET number 280 on the current server
For request 3, server replied:  This is GET number 281 on the current server
For request 1, server replied:  This is GET number 282 on the current server
```

Awaiting a promise prevents your method from continuing

```
async function makeThreeSerialRequests(): Promise<void> {
    await makeOneGetRequest(1);
    await makeOneGetRequest(2);
    await makeOneGetRequest(3);
    console.log('Heard back from all of the requests')
}

makeThreeSerialRequests();
```



```
For request 1, server replied:  This is GET number 37 on the current server
For request 2, server replied:  This is GET number 38 on the current server
For request 3, server replied:  This is GET number 39 on the current server
Heard back from all of the requests
Elapsed time: 364.0822000205517 milliseconds
```

Promise.all starts several promises concurrently

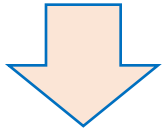
```
async function makeThreeConcurrentRequests(): Promise<void> {
    await Promise.all([
        makeOneGetRequest(1),
        makeOneGetRequest(2),
        makeOneGetRequest(3)
    ])
    console.log('Heard back from all of the requests')
}
```

- **Promise.all** takes a list of promises and runs them all concurrently.
- It finishes when all the promises have finished.

Promise.all allows for concurrency

```
async function makeThreeConcurrentRequests(): Promise<void> {
    await Promise.all([
        makeOneGetRequest(1),
        makeOneGetRequest(2),
        makeOneGetRequest(3)
    ])
    console.log('Heard back from all of the requests')
}

makeThreeConcurrentRequests();
```



```
For request 2, server replied:  This is GET number 58 on the current server
For request 1, server replied:  This is GET number 59 on the current server
For request 3, server replied:  This is GET number 60 on the current server
Heard back from all of the requests
Elapsed time: 203.7674999833107 milliseconds
```

Visualizing Promise.all (1)

Sequential version: ~400msec

```
async function makeThreeSerialRequests():
Promise<void> {
    await makeOneGetRequest(1);
    await makeOneGetRequest(2);
    await makeOneGetRequest(3);
    console.log('Heard back from all of the
requests')
}
```

"Don't make another request until you got the last response back"

Concurrent version: ~126msec

```
async function makeThreeConcurrentRequests():
Promise<void> {
    await Promise.all([
        makeOneGetRequest(1),
        makeOneGetRequest(2),
        makeOneGetRequest(3)
    ])
    console.log('Heard back from all of the requests')
}
```

"Make all of the requests at the same time, then wait for all of the responses"

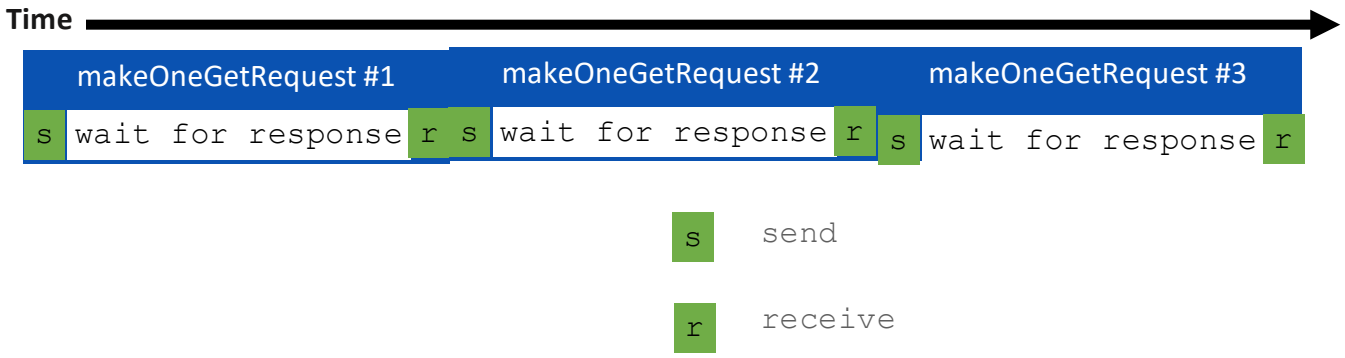
Visualizing Promise.all (2)

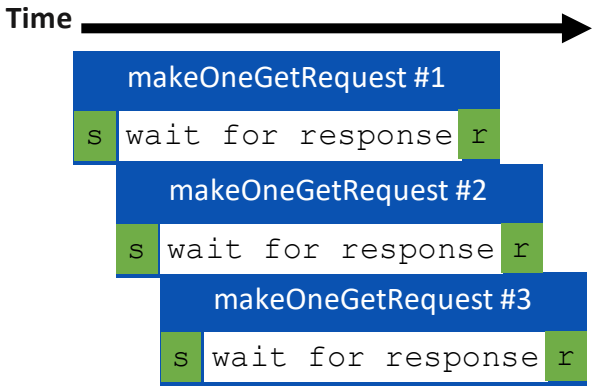
Sequential version: ~400msec

```
async function makeThreeSerialRequests():
Promise<void> {
    await makeOneGetRequest(1);
    await makeOneGetRequest(2);
    await makeOneGetRequest(3);
    console.log('Heard back from all of the
requests')
}
```

Concurrent version: ~126msec

```
async function makeThreeConcurrentRequests():
Promise<void> {
    await Promise.all([
        makeOneGetRequest(1),
        makeOneGetRequest(2),
        makeOneGetRequest(3)
    ])
    console.log('Heard back from all of the requests')
}
```





Patterns for Concurrent Code: Example: Using a Web Service

Here is a web service we'd like to talk to.

```
POST /transcripts
 -- adds a new student to the database,
 -- returns an ID for this student.
 -- requires a body parameter 'name'
 -- Multiple students may have the same name.
GET  /transcripts/:ID
 -- returns transcript for student with given ID.  Fails if no such student
DELETE /transcripts/:ID
 -- deletes transcript for student with the given ID, fails if no such student
POST /transcripts/:studentID/:courseNumber
 -- adds an entry in this student's transcript with given name and course.
 -- Requires a body parameter 'grade'
 -- Fails if there is already an entry for this course in the student's transcript
GET  /transcripts/:studentID/:courseNumber
 -- returns the student's grade in the specified course.
 -- Fails if student or course is missing.
GET  /studentids?name=string
 -- returns list of IDs for student with the given name
```

An Example Task Using the Transcript Server

- Given an array of StudentIDs:
  - Request each student's transcript, and save it to disk so that we have a copy
  - Once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

Generating a promise for a student

```
async function promiseForTranscript(studentID: number) {
        const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)


}
```

Here is something we plan to do later

The promise is to call axios and wait for the result.

Generating a promise for a student (cont'd)

```
async function promiseForTranscript(studentID: number) {
        const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
        await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`,
            JSON.stringify(response.data))
    }
```

After we get the response, make a new promise: this time to write the result to a file. Then wait for that to finish.

When the file-writing promise is fulfilled, then the whole original promise is fulfilled.

Now, actually generate all the promises

```
async function runClientAsync(studentIDs:number[]) {
    console.log('Making requests for ${studentIDs}');

    async function promiseForTranscript(studentID: number) { .. }

    const promisesForTranscripts = studentIDs.map(promiseForTranscript)
    console.log('Requests sent!');



}
```

map applies the function specified to each element in the array and returns a new array containing the result of each of those functions

Wait for all the promises to resolve

```
async function runClientAsync(studentIDs:number[]) {
    console.log('Making requests for ${studentIDs}');

    async function promiseForTranscript(studentID: number) { .. }

    const promisesForTranscripts = studentIDs.map(promiseForTranscript)
    console.log('Requests sent!');
    await Promise.all(promisesForTranscripts);




}
```

Asynchronously stat all the files

```
async function runClientAsync(studentIDs:number[]) {
    console.log('Making requests for ${studentIDs}');

    async function promiseForTranscript(studentID: number) { .. }

    const promisesForTranscripts = studentIDs.map(promiseForTranscript)
    console.log('Requests sent!');
    await Promise.all(promisesForTranscripts);
    const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-
${studentID}.json`)));



  }
```

..then total the sizes

```
async function runClientAsync(studentIDs:number[]) {
    console.log('Making requests for ${studentIDs}');

    async function promiseForTranscript(studentID: number) { .. }

    const promisesForTranscripts = studentIDs.map(promiseForTranscript)
    console.log('Requests sent!');
    await Promise.all(promisesForTranscripts);
    const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-
${studentID}.json`)));
    const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
    console.log(`Finished calculating size: ${totalSize}`);
    console.log('Done');
  }
```

'reduce' is what you called 'foldl' back in Fundies 1.

Leverage Concurrency When Possible

- Where you place awaits can make a big difference!

For each student: make an async handler to fetch their transcript and save it

The code we've seen on past slides:

```
async function runClientAsync() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  const promisesForTranscripts = studentIDs.map(
    async (studentID) => {
      const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
      await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
    });
  console.log('Requests sent!');
  await Promise.all(promisesForTranscripts);
  const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`)));
  const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
  console.log(`Finished calculating size: ${totalSize}`);
}
```

✓

Running time: 1.5 sec

For each student: wait to fetch their transcript, then wait to write it, then go on to the next student

This accomplishes the same function, but without concurrency:

```
async function runClientAsyncSerially() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  for(let studentID of studentIDs){
    const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`);
    await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
  }
  let totalSize = 0;
  for(let studentID of studentIDs){
    const stats = await fsPromises.stat(`transcript-${studentID}.json`);
    totalSize += stats.size;
  }
  console.log(`Finished calculating size: ${totalSize}`);
}
```

✗

Running time: 2.2 sec

This is what we mean by "your code can become synchronous

Async/Await Programming Activity

- Your task is to write a new `async` function, `importGrades`, which takes in input of the type `ImportTranscript[]`.
- `importGrades` should create a student record for each `ImportTranscript`, and then post the grades for each of those students.
- After posting the grades, it should fetch the transcripts for each student and return an array of transcripts.

Download the activity (includes instructions in README.md):
Linked from course webpage for Module 5

Learning Goals for this Lesson

- At the end of this lesson, you should be prepared to:
  - Explain how to achieve concurrency through asynchronous operations and Promise.all in TypeScript.
  - Write asynchronous and concurrent code in TypeScript using async/await and Promise.all.

Learning Goals for this Lesson (expanded)

- At the end of this lesson, you should be prepared to:
  - Explain how to achieve concurrency through asynchronous operations and Promise.all in TypeScript.
  - Write asynchronous and concurrent code in TypeScript using async/await and Promise.all.
  - Write asynchronous code using promises and .then().
  - Explain the difference between JS run-to-completion semantics and interrupt-based semantics.

Additional Topics

General Rules for Writing Asynchronous Code

- Don’t perform long-running computations or synchronous IO
- Leverage concurrency when possible
  - Remember that events are processed in the order they are ***received***
  - But events may arrive in unexpected order!
- Always check for errors (try/catch for async/await, “.catch” for promises)

Async functions use Promises Under the Hood

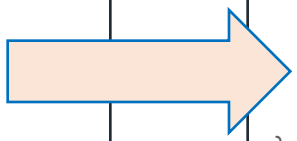
Promises Enforce Ordering Through “Then”

```
1. console.log('Making requests');
2. axios.get('https://rest-example.covey.town/')
    .then((response) =>{
        console.log('Heard back from server');
        console.log(response.data);
   });
3. axios.get('https://www.google.com/')
      .then((response) =>{
       console.log('Heard back from Google');
      });
4. axios.get('https://www.facebook.com/')
      .then((response) =>{
        console.log('Heard back from Facebook');
      });
5. console.log('Requests sent!');
```

- **axios.get** returns a promise.
- **p.then** mutates that promise so that the then block is run immediately after the original promise returns.
- The resulting promise isn’t completed until the then block finishes.
- You can chain .**then**’s, to get things that look like p.then().then().then()

Async/await code is compiled into promise/then code

```
async function
makeThreeSerialRequests(){
1.   console.log('Making first
request');
2.   await makeOneGetRequest();
3.   console.log('Making second
request');
4.   await makeOneGetRequest();
5.   console.log('Making third
request');
6.   await makeOneGetRequest();
7.   console.log('All done!');
}
makeThreeSerialRequests();
```



```
console.log('Making first request');
makeOneGetRequest().then( () =>{
  console.log('Making second request');
  return makeOneGetRequest();
}).then(() => {
  console.log('Making third request');
  return makeOneGetRequest();
}).then(()=>{
  console.log('All done!');
});
```

Syntax for Writing Asynchronous Code

- You can only call **await** from a function that is **async**
- You can only **await** on functions that return a **Promise**
- Beware: **await** makes your code synchronous (this is what we want it for)!
- Handle errors using try/catch instead of "catch" (common gotcha with promises)

```
async function makeOneGetRequest(): Promise<void> {
    console.log("Making Request");
    try {
        const response = await axios.get("https://rest-
example.covey.town");
        console.log("Heard back from server");
        console.log(response.data);
    } catch (err) {
        console.log('Uh oh!'); console.trace(err);
    }
}
```

```
function makeOneGetRequestNoAsync(): Promise<void> {
    console.log("Making Request");
    return axios.get("https://rest-
example.covey.town").then((response) => {
        console.log("Heard back from server");
        console.log(response.data);
    }).catch(err => {
        console.log('Uh oh!');
        console.trace(err);
    });
}
```

Data Races in TS vs. Java

Data Races in TS vs. Java

```
let x : number = 1

async function asyncDouble() {
    // start an asynchronous computation and wait for the result
    await makeOneGetRequest(1);
    x = x * 2  // statement 1
}


async function asyncIncrementTwice() {
    // start an asynchronous computation and wait for the result
    await makeOneGetRequest(2);
    x = x + 1;   // statement 2
    x = x + 1;   // statement 3
}

async function run() {
    await Promise.all([asyncDouble(), asyncIncrementTwice()])
    console.log(x)
}
```

Explanation

- In the JS run-to-completion semantics, statement 3 is guaranteed to run immediately after statement 2, so the only possible orders of execution are:

  1,2,3 (1 runs before 2 and 3, final value of x is 4)

  2,3,1 (2 and 3 run before 1, final value of x is 6)

- In an interrupt-based model, it is possible that statement 1 runs **BETWEEN** statement 2 and statement 3, yielding the order of execution

  2,1,3 (final value of x is 5).

Explanation (2)

- Notice that there is still a data race between statement 1 and statements 2 and 3;
- Run-to-completion semantics does not eliminate data races entirely, but it makes them much rarer.

The Self-Ticking Clock

- To make the clock self-ticking, add the following line to your clock:

```
constructor () {
  setInterval(() =>{this.tick()},50)
}
```

Learning Goals for this Lesson (expanded)

- At the end of this lesson, you should be prepared to:
  - Explain how to achieve concurrency through asynchronous operations and Promise.all in TypeScript.
  - Write asynchronous and concurrent code in TypeScript using async/await and Promise.all.
  - Write asynchronous code using promises and .then().
  - Explain the difference between JS run-to-completion semantics and interrupt-based semantics.