

CS 4530: Fundamentals of Software Engineering

Module 5: Concurrency Patterns in Typescript

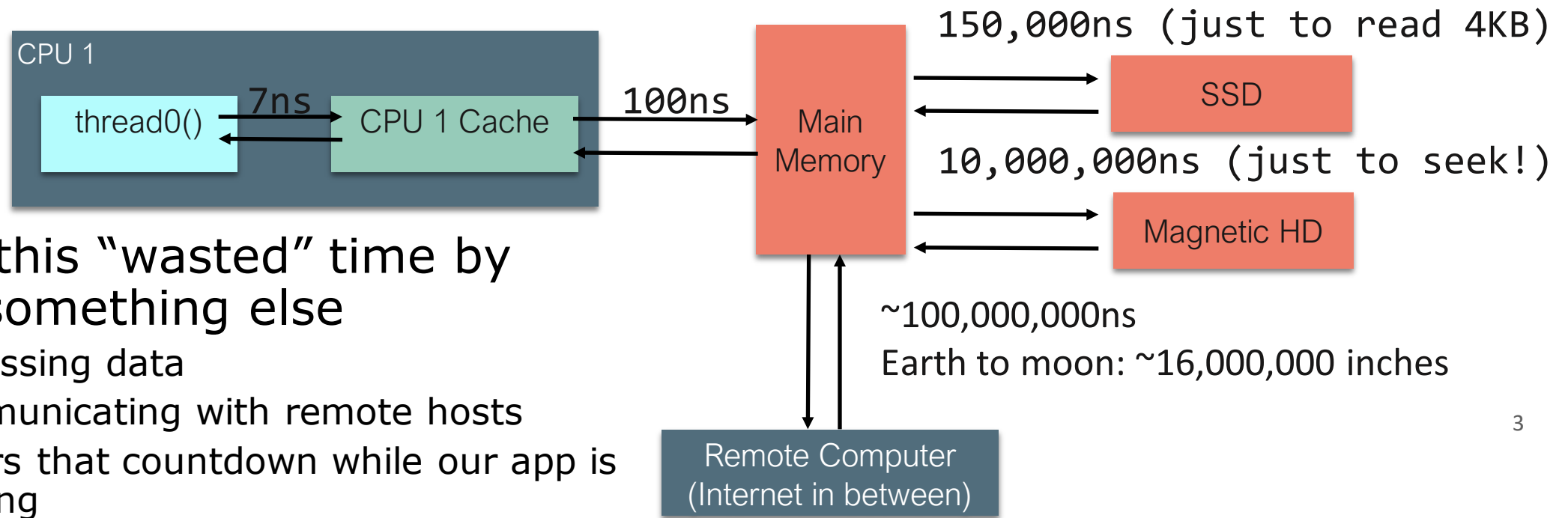
Adeel Bhutta, Jan Vitek, Mitch Wand
Khoury College of Computer Sciences

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`.
 - Explain the difference between JS run-to-completion semantics and interrupt-based semantics.

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
- Inter-process 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
- In Typescript, these “threads” are called **promises**.
- Each thread decides when it should *yield* to let other threads execute
- Typically via a **yield** or **await** operation

A computation is not suspended until it hits an 'await' or finishes.

- A computation is suspended when it hits an 'await'. The runtime system (node.js, for us) chooses what to do next.
- This means that a computation runs **continuously** until it is either suspended or completed.



This is known as "Run to Completion"

But where does the concurrency come from?

Answer: JS/TS has some primitives for starting a concurrent computation

- These are things like http requests, I/O operations, or timers.
- You will hardly ever call one of these primitives yourself; usually they are wrapped in a convenient procedure, e.g., we write

```
axios.get('https://rest-example.covey.town')
```

to make an http request, or

```
fs.readFile(filename)
```

to read the contents of a file.

Pattern for starting a concurrent computation

```
async function makeRequest(requestNumber:number) {  
    // some code  
    const response =  
        await axios.get('https://rest-example.covey.town')  
    // more code  
}
```

- The http request is sent immediately.
- A promise is created to run the *more code* *after* the http call returns (i.e., the code after “awaits” is blocked)
- The call to `makeRequest` returns immediately.

The pattern in action

```
export async function makeRequest(requestNumber:number) {  
  console.log(`makeRequest is about to start request ${requestNumber}`);  
  const response = await axios.get('https://rest-exa...');  
  console.log(`makeRequest resumes request ${requestNumber}`);  
  console.log(`makeRequest reports that for request  
response.data);  
}
```

```
console.log("main thread is about to call makeRequest");  
makeRequest(1000);  
console.log("main thread continues after makeRequest returns");  
console.log("end of main thread");
```

1. Axios.get starts the http request in the background, and
2. Creates a promise to do the code after the await.
3. The call to makeRequest returns.

\$ npx ts-node example1
main thread is about to call makeRequest
makeRequest is about to start request 1000
main thread continues after makeRequest returns
end of main thread

4. The main thread finishes.
5. The computation resumes the promise

makeRequest resumes request 1000
makeRequest reports that for request '1000', server replied: This is GET number 200 on the current server

```
import makeRequest from './makeRequest';
import timeIt from './timeIt'
```

example2.ts

```
async function makeThreeSimpleRequests() {
  makeRequest(1);
  makeRequest(2);
  makeRequest(3);
  console.log("Three requests made; this thread finishes")
}
```

```
timeIt("main thread", makeThreeSimpleRequests)
```

This makes it simple to run several concurrent requests

Requests are made in
order

But the response for
request 3 arrived from
the server before
request 1.

```
$ npx ts-node example2
```

```
makeRequest is about to start request 1
```

```
makeRequest is about to start request 2
```

```
makeRequest is about to start request 3
```

```
Three requests made; this thread finishes
```

```
Elapsed time for main thread: 41.064 milliseconds
```

```
makeRequest reports that for request '3', server replied: This is GET number 223  
on the current server
```

```
makeRequest reports that for request '1', server replied: This is GET number 224  
on the current server
```

```
makeRequest reports that for request '2', server replied: This is GET number 225  
on the current server
```

```
import makeRequest from './makeRequest';  
import timeIt from './timeIt'
```

example3.ts

```
async function makeThreeSerialRequests() {  
  await makeRequest(1);  
  await makeRequest(2);  
  await makeRequest(3);  
  console.log("Three requests made; this thread finishes")  
}  
  
timeIt("main thread", makeThreeSerialRequests)
```

await makes your code more sequential

```
$ npx ts-node example3
```

makeRequest is about to start request 1

makeRequest reports that for request '1', server replied:
number 232 on the current server

makeRequest is about to start request 2

makeRequest reports that for request '2', server replied: This is GET
number 233 on the current server

makeRequest is about to start request 3

makeRequest reports that for request '3', server replied: This is GET
number 234 on the current server

Three requests made; this thread finishes

Elapsed time for main thread: 800.270 milliseconds

Second request doesn't start
until first request returns

Promises are values; async functions return promises

```
async function makeThreeSimpleRequests() {  
  const p1 : Promise<void> = makeRequest(1);  
  const p2 : Promise<void> = makeRequest(2);  
  const p3 : Promise<void> = makeRequest(3);  
  const thePromises = [p1,p2,p3]  
  console.log(`This thread reports: thePromises = [${thePromises}]`)  
  console.log(`This thread finishes`)  
}
```

example4.ts

So, you can make lists of them!

```
timeIt("main thread", makeThreeSimpleRequests)
```

```
$ npx ts-node example4
```

makeRequest is about to start request 1

makeRequest is about to start request 2

makeRequest is about to start request 3

This thread reports: thePromises = [[object Promise],[object Promise],[object Promise]]

This thread finishes

Elapsed time for main thread: 36.501 milliseconds

makeRequest reports that for request '2', server replied: This is GET number 248 on the current server

makeRequest reports that for request '3', server replied: This is GET number 247 on the current server

makeRequest reports that for request '1', server replied: This is GET number 249 on the current server

Promise.all allows you to wait for all of the promises in a list to finish

```
async function makeThreeConcurrentRequests() {  
  const p1 : Promise<void> = makeRequest(1);  
  const p2 : Promise<void> = makeRequest(2);  
  const p3 : Promise<void> = makeRequest(3);  
  const thePromises = [p1,p2,p3]  
  await Promise.all(thePromises)  
  console.log(`This thread reports: thePromises = [${thePromises}]`)  
  console.log(`This thread finishes`)  
}
```

example5.ts

```
timeIt("main thread", makeThreeConcurrentRequests)
```

The thread doesn't resume until
ALL of the promises are satisfied

```
$ npx ts-node example5
```

makeRequest is about to start request 1

makeRequest is about to start request 2

makeRequest is about to start request 3

makeRequest reports that for request '2', server replied: This is GET number 259 on the current server

makeRequest reports that for request '1', server replied: This is GET number 260 on the current server

makeRequest reports that for request '3', server replied: This is GET number 261 on the current server

This thread reports: thePromises = [[object Promise],[object Promise],[object Promise]]

This thread finishes

Elapsed time for main thread: 256.518 milliseconds

Visualizing Promise.all (1)

Sequential version: ~206 msec

```
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: ~80 msec

```
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 now,
then wait for all of the
responses”

Visualizing Promise.all (2)

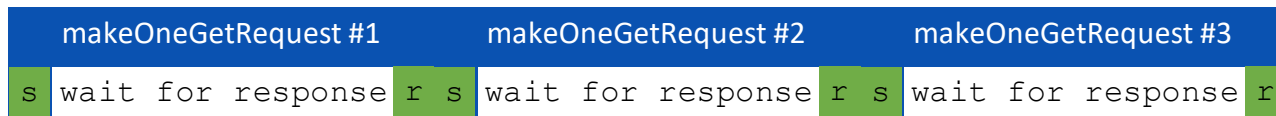
Sequential version: ~206 msec

```
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: ~80 msec

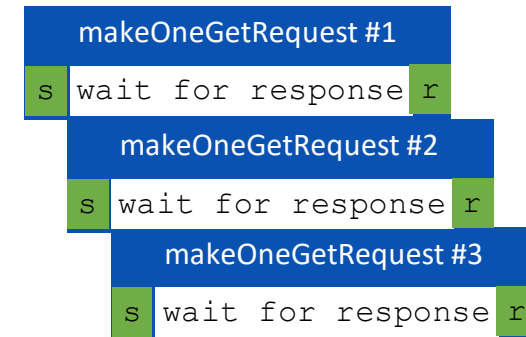
```
async function makeThreeConcurrentRequests():  
  Promise<void> {  
    await Promise.all([  
      makeOneGetRequest(1),  
      makeOneGetRequest(2),  
      makeOneGetRequest(3)  
    ])  
    console.log('Heard back from all of the requests')  
  }
```

Time →



s send
r receive

Time →



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, and calculate its size
 - 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 each student

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

```
async function asyncProcessStudent(studentID: number) : Promise<number> {  
  // wait to get the student data  
  const response = await asyncGetStudentData(studentID)  
  // asynchronously write the file  
  await fsPromises.writeFile(  
    dataFileName(studentID),  
    JSON.stringify(response.data))  
  // last, extract its size  
  const stats = await fsPromises.stat(dataFileName(studentID))  
  const size : number = stats.size  
  return size  
}
```

Calling await gives other processes a chance to run.

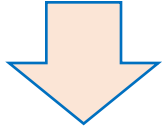
Running the student processes concurrently

```
async function runClientAsync(studentIDs:number[]) {  
  console.log(`Generating Promises for ${studentIDs}`);  
  const studentPromises =  
    studentIDs.map(studentID => asyncProcessStudent(studentID)) ;  
  console.log('Promises Created!');  
  console.log('Satisfying Promises Concurrently')  
  const sizes = await Promise.all(studentPromises);  
  console.log(sizes)  
  const totalSize = sum(sizes)  
  console.log(`Finished calculating size: ${totalSize}`);  
  console.log('Done');  
}
```

Map-promises pattern: take a list of elements and generate a list of promises, one per element

Output

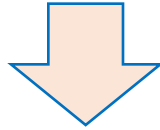
```
runClientAsync([411,412,423])
```



```
$ npx ts-node transcript-v2.simple.ts  
Generating Promises for 411,412,423  
Promises Created!  
Satisfying Promises Concurrently  
[ 151, 92, 145 ]  
Finished calculating size: 388  
Done
```

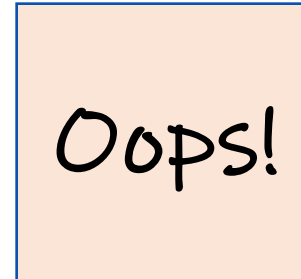
But what if there's an error?

```
runClientAsync([411,412,87065,423,23044])
```



```
$ npx ts-node transcript-v2.simple.ts
Generating Promises for 411,412,87065,423,23044
Promises Created!
Satisfying Promises Concurrently

C:\Users\wand\OneDrive\Documents\Work\Courses\CS 4530
Future\My Modules Workspace\Module 05 Concurrency
Patterns\Examples\Lecture05-
Async\node_modules\axios\lib\core\createError.js:16
  var error = new Error(message);
                ^
Error: Request failed with status code 404
```



Need to catch the error

```
type StudentData = {isOK: boolean, id: number, payload?: any }

/** asynchronously retrieves student data, */
async function asyncGetStudentData(studentID: number): Promise<StudentData> {
  try {
    const returnValue =
      await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    return { isOK: true, id: studentID, payload: returnValue }
  } catch (e) {
    return { isOK: false, id: studentID }
  }
}
```

Catch the error and transmit it in a form the rest of the caller can handle.

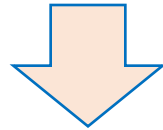
And recover from the error...

```
async function asyncProcessStudent(studentID: number): Promise<number> {  
  // wait to get the student data  
  const response = await asyncGetStudentData(studentID)  
  if (!(response.isOK)) {  
    console.error(`bad student ID ${studentID}`)  
    return 0  
  } else {  
    await fsPromises.writeFile(  
      dataFileName(studentID),  
      JSON.stringify(response.payload.data))  
    // last, extract its size  
    const stats = await fsPromises.stat(dataFileName(studentID))  
    const size: number = stats.size  
    return size  
  }  
}
```

Design decision: if we have a bad student ID, we'll print out an error message, and count that as 0 towards the total.

New output

```
runClientAsync([411,32789,412,423,10202040])
```



```
$ npx ts-node transcript-v2.handle-errors.ts  
Generating Promises for 411,32789,412,423,10202040  
Promises Created!  
Wait for all promises to be satisfied  
bad student ID 32789  
bad student ID 10202040  
[ 151, 0, 92, 145, 0 ]  
Finished calculating size: 388  
Done
```


Pattern for testing an async function

```
import axios from 'axios'

async function echo(str: string) : Promise<string> {
  const res =
    await axios.get(`https://httpbin.org/get?answer=${str}`)
  return res.data.args.answer
}

test('request should return its argument', async () => {
  expect.assertions(1)
  await expect(echo("33")).resolves.toEqual("33")
})
```

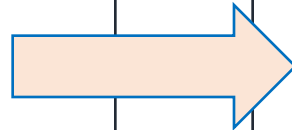
General Rules for Writing Asynchronous Code

- You can't return a value from an async procedure to an ordinary procedure.
 - Call async procedures only from other async functions or from the top level.
- Break up any long-running computation into **async/await** segments so other processes will have a chance to run.
- Leverage concurrency when possible
 - Use **promise.all** if you need to wait for multiple promises to return.
- Check for errors with **try/catch**

Odds and Ends You Should Know About

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!');
});
```

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()`

You can still have a data race

```
let x : number = 10
```

```
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)  
}
```

This is not Java!

```
let x : number = 10
```

```
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  
    // nothing can happen between these two statements!!  
    x = x + 1; // statement 3  
}
```

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

The Self-Ticking Clock

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

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


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`.
 - Explain the difference between JS run-to-completion semantics and interrupt-based semantics.