CS144: Asynchronous Programming

Traditional synchronous programming

- Example: Sending the user’s profile picture over network

```javascript
function sendPicture(userid) {
    pictureName = db.find(userid, "profile_picture");
    picture = fs.readFile(pictureName);
    socket.write(picture);
    console.log("done!")
}
```

- Blocking operation in every step!
- Finishing the function may take a long time with long waits
- Q: How can the server handle many requests concurrently despite waits?

- Multi-threaded vs single-threaded
  - Multi threading!
    - Create one thread per each request
    - Used by most traditional servers, including Apache, Tomcat, …
    - Significant resource overhead (~ 10MB per thread)
      - Memory use (~ 10MB per thread)
      - Thread invocation overhead
    - Easy to program: Programming style matches with how we think
  - Single threading
    - Use one thread to handle all requests
    - Used by Node.js and browser JavaScript engines
    - More efficient resource usage
    - Nonblocking, asynchronous programming
      - Use callback function for any blocking call
      - Difficult to program, very different from traditional programming

Asynchronous nonblocking programming

- Example

Junghoo "John" Cho (cho@cs.ucl.edu)
function sendPicture(userid) {
    db.find(userid, "profile_picture", callback1);
}

function callback1(pictureName) {
    fs.readFile(pictureName, callback2);
}

function callback2(picture) {
    socket.write(picture, callback3);
}

function callback3() {
    console.log("done!");
}

- “callback hell”
- Very difficult to see the logical sequence of actions
- Possible solution: Nested callbacks using anonymous functions

function sendPicture(userid) {
    db.find(userid, "profile_picture", (pictureName) => {
        fs.readFile(pictureName, (picture) => {
            socket.write(picture, () => {
                console.log("done!");
            });
        });
    });
}

- Better, but still ugly, difficult to understand, and easy to make mistakes
- Q: Can we make it better, easier?
- Two new language constructs to mitigate problems with callbacks
  * Promise (ECMAScript 2015)
  * async/await (ECMAScript 2017)
Promise

- An asynchronous function may return a “promise”
- A promise represents the “guarantee” of eventual completion or failure of an asynchronous operation
- Once a promise is obtained, we can attach a callback to it using `then()`

```javascript
let promise = asyncOperation();
promise.then(resolveCallback, rejectCallback);
```

- Depending on the success (= resolve) or failure (= reject) of the operation, appropriate callback is eventually invoked
  - “Logical return value” from the asyncOperation will be passed as the parameter of resolveCallback in most cases
  - “Error code” is passed as the parameter of rejectCallback in most cases
- Arguments to `then()` are optional and can be omitted if not needed
- Q: Is it any useful?

“Promise Chain”

```javascript
function sendPicture(userid) {
    let promise1 = db.find(userid, "profile_picture");
    let promise2 = promise1.then(pictureName => fs.readFile(
      pictureName));
    let promise3 = promise2.then(picture => socket.write(
      picture));
    let promise4 = promise3.then(() => console.log("done"));
}
```

- If callbacks return a promise, calling `then()` returns the promise from the callback(s)
- We can “chain” a sequence of asynchronous callbacks to make our code look like a synchronous program!

Even further,

```javascript
function sendPicture(userid) {
    db.find(userid, "profile_picture")
```
```javascript
.then(pictureName => fs.readFile(pictureName))
.then(picture => socket.writeFile(picture))
.then(() => console.log("done!"))
catch(rejectCallback);
```

- `catch(rejectCallback)` is short for `then(null, rejectCallback)`
- If a rejection is not handled by a callback, it is forwarded to the next `then()`

**Promise guarantees:**
- Callbacks will never be called before the completion of the current run of the JavaScript event loop
- Callbacks added with `then()` even after the success/failure of the asynchronous operation will be called

**On Node.js, “promisified version” of asynchronous API modules exist**
- e.g., `fs-extra`, mongoDB driver returns a promise if no callback is passed

**async/await**

- In ECMAScript 2017, `async/await` can be used on any function that returns a promise

```javascript
async function sendPicture(userid) {
  try {
    pictureName = await db.find(userid, "profile_picture");
    picture = await fs.readFile(pictureName);
    await socket.writeFile(picture);
    console.log("done!");
  } catch (e) {
    console.log("Error in processing request!");
  }
}
```

- `await` can be used inside an `async` function to perform an asynchronous operation and “wait for” the result from the operation
-- await can be used in front of (any function that returns) a promise
  * If promise is resolved, the “resolved value” is returned from await
  * If promise is rejected, an exception is raised, which can be caught with
    try/catch
      ▶ Important to catch possible exceptions from await
      ▶ Otherwise the app terminates with exception

• Any function declared as async automatically returns a promise
  – The real return value of the function is returned when the promise is resolved
    (or passed as the parameter to the resolveCallback function)
  – Conceptually, async function yields control at await and comes back to the
    point once the request is resolved/rejected

• async/await makes asynchronous programming almost like a synchronous pro-
  gramming!

Parallel await

• Q: How long will it take to print out the result?

```javascript
async function addAsync(x) {
  return await doubleAfter2Seconds(x) + await doubleAfter2Seconds(x) + await doubleAfter2Seconds(x);
}
addAsync(10).then(v => console.log(v));
```

• Q: How long will it take?

```javascript
async function addAsync(x) {
  const a = doubleAfter2Seconds(x);
```
```cpp
const b = doubleAfter2Seconds(x);
const c = doubleAfter2Seconds(x);
return await a + await b + await c;
```