Overview of Blocking vs Non-Blocking

This overview covers the difference between **blocking** and **non-blocking** calls in Node.js. This overview will refer to the event loop and libuv but no prior knowledge of those topics is required. Readers are assumed to have a basic understanding of the JavaScript language and Node.js callback pattern.

"I/O" refers primarily to interaction with the system's disk and network supported by libuv.

Blocking

Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a **blocking** operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as **blocking**. Synchronous methods in the Node.js standard library that use libuv are the most commonly used **blocking** operations. Native modules may also have **blocking** methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are **non-blocking**, and accept callback functions. Some methods also have **blocking** counterparts, which have names that end with `Sync`.

Comparing Code

Blocking methods execute **synchronously** and **non-blocking** methods execute **asynchronously**.

Using the File System module as an example, this is a **synchronous** file read:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is
read
```

And here is an equivalent **asynchronous** example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

The first example appears simpler than the second but has the disadvantage of the second line **blocking** the execution of any additional JavaScript until the entire file is read. Note that in the synchronous version if an error is thrown it will need to be caught or the process will crash. In the asynchronous version, it is up to the author to decide whether an error should throw as shown.

Let's expand our example a little bit:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is
read
console.log(data);
moreWork(); // will run after console.log
```

And here is a similar, but not equivalent asynchronous example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

In the first example above, `console.log` will be called before `moreWork()`. In the second example `fs.readFile()` is **non-blocking** so JavaScript execution can continue and `moreWork()` will be called first. The ability to run `moreWork()` without waiting for the file read to complete is a key design choice that allows for higher throughput.

Concurrency and Throughput

JavaScript execution in Node.js is single threaded, so concurrency refers to the event loop's capacity to execute JavaScript callback functions after completing other work. Any code that is expected to run in a concurrent manner must allow the event loop to continue running as non-JavaScript operations, like I/O, are occurring.

As an example, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing **non-blocking** asynchronous operations frees up that 45ms per request to handle other requests. This is a significant difference in capacity just by choosing to use **non-blocking** methods instead of **blocking** methods.

The event loop is different than models in many other languages where additional threads may be created to handle concurrent work.

Dangers of Mixing Blocking and Non-Blocking Code

There are some patterns that should be avoided when dealing with I/O. Let's look at an example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
fs.unlinkSync('/file.md');
```

In the above example, `fs.unlinkSync()` is likely to be run before `fs.readFile()`, which would delete `file.md` before it is actually read. A better way to write this, which is completely **non-blocking** and guaranteed to execute in the correct order is:

```
const fs = require('fs');
fs.readFile('/file.md', (readFileErr, data) => {
  if (readFileErr) throw readFileErr;
  console.log(data);
  fs.unlink('/file.md', (unlinkErr) => {
    if (unlinkErr) throw unlinkErr;
  });
});
```

The above places a **non-blocking** call to `fs.unlink()` within the callback of `fs.readFile()` which guarantees the correct order of operations.