Understanding the Node.js Event Loop

Learn Node.js by Example

Axiom #1

I/O is expensive

The cost of I/O

L1-cache	3 cycles
L2-cache	14 cycles
RAM	250 cycles
Disk	41 000 000 cycles
Network	240 000 000 cycles

*http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/

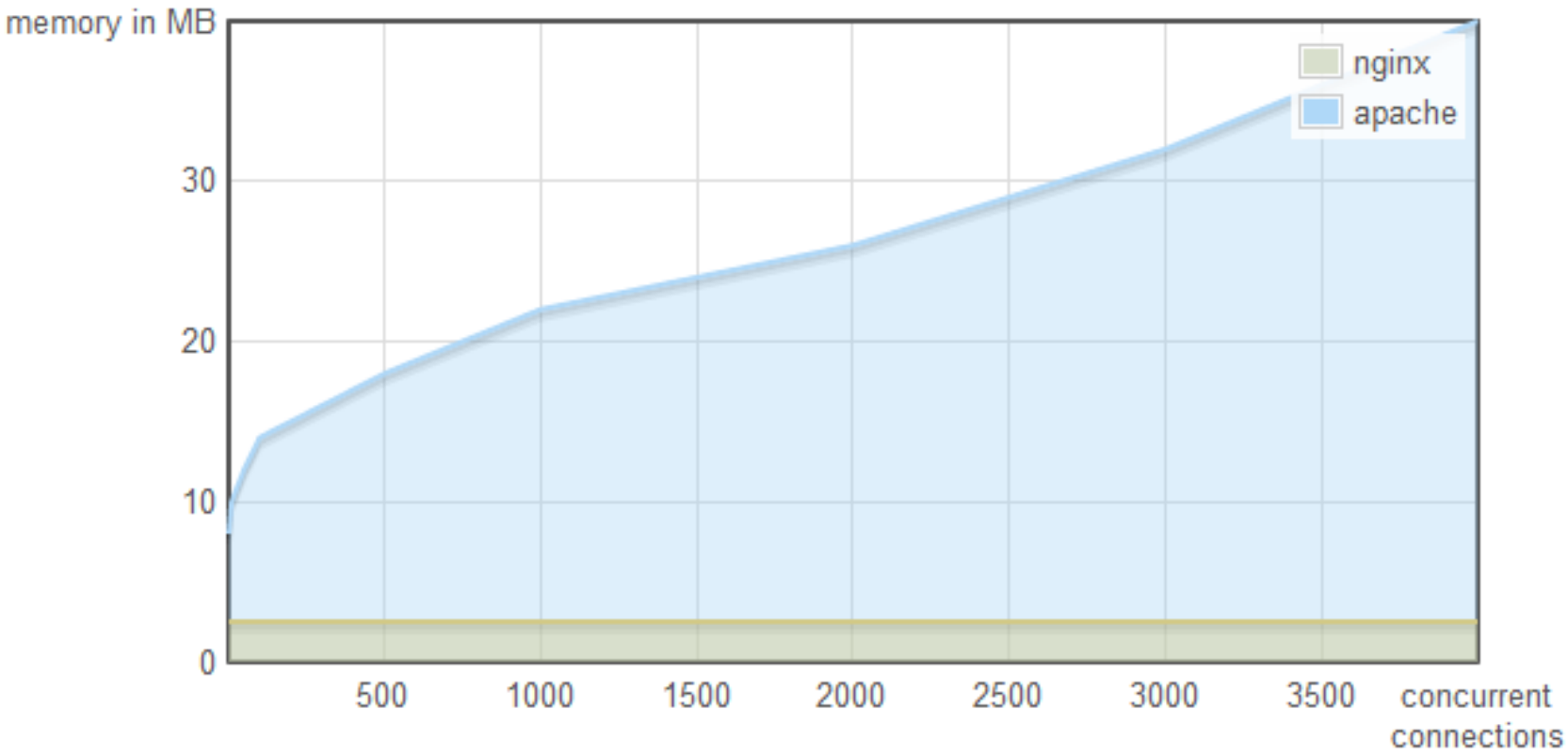
Ways to deal with I/O

- **Synchronous**
  - One requests at a time, first come, first serve
- **Fork**
  - New process for each request
- **Threads**
  - New thread for each request

*http://www.nightmare.com/medusa/async_sockets.html

Axiom #2

Thread-per-connection is memory-expensive

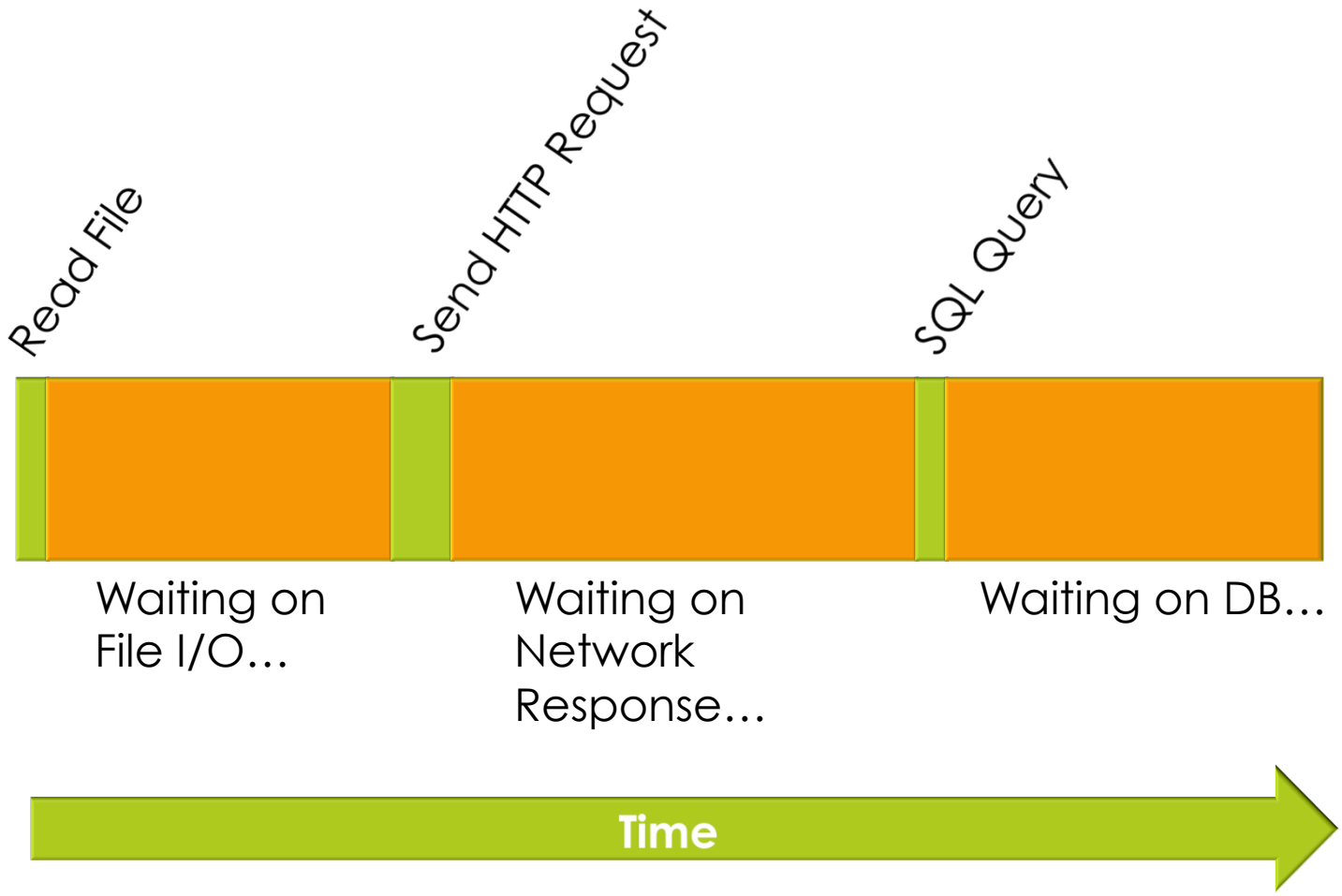


- Nginx is a web server based on an event loop
- The above graph shows:
  - nginx maintaining flat memory usage as connections increase
  - The Apache HTTP web server steadily increasing memory usage with connections

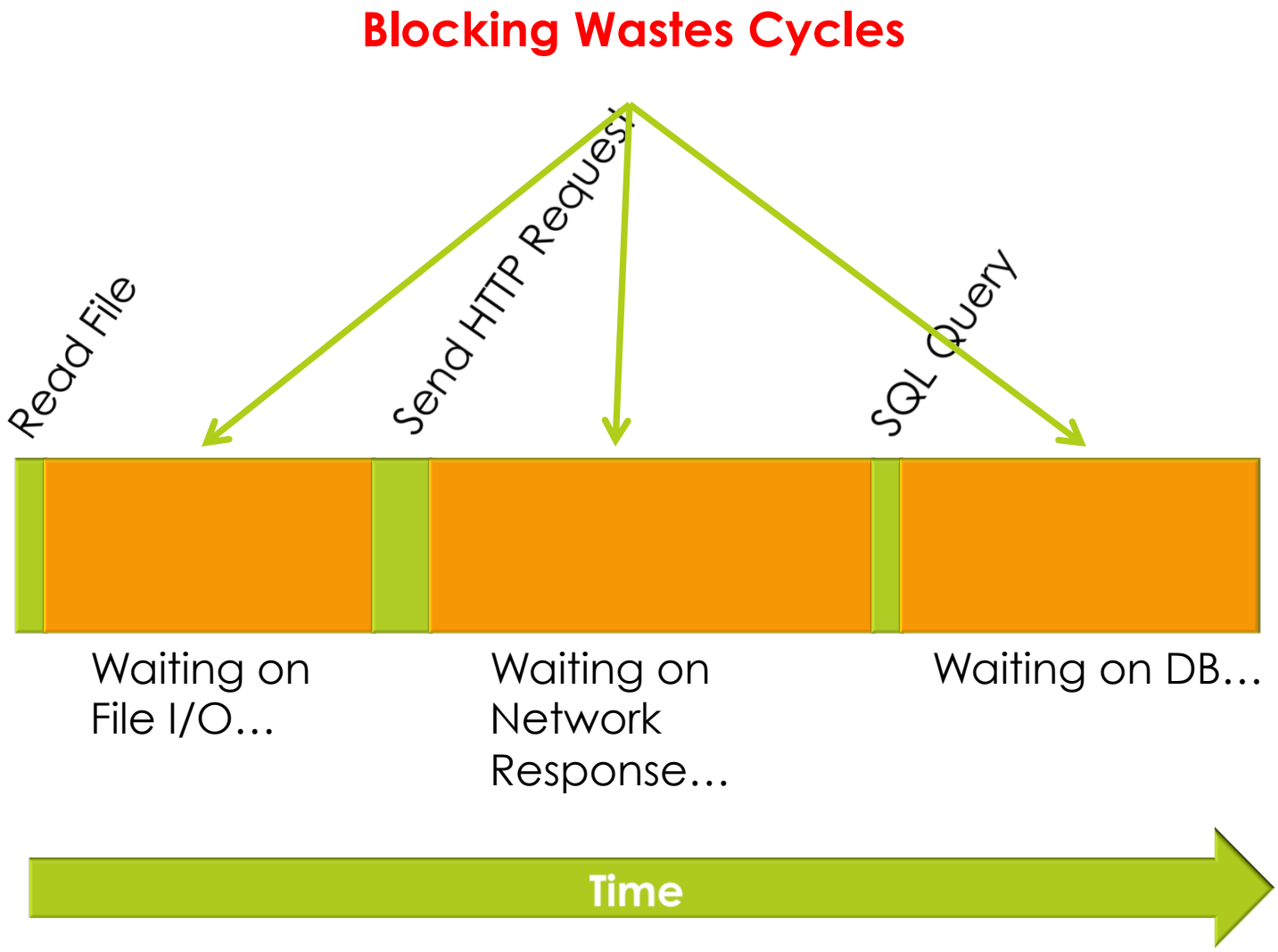
Traditional Threaded Model

- N worker threads/processes
- Each incoming connection handed to a worker
  - That worker is now "in use", and can handle no other connection, even if it is waiting on:
    - File I/O
    - DB I/O
    - Network I/O
    - etc

The life of a worker…



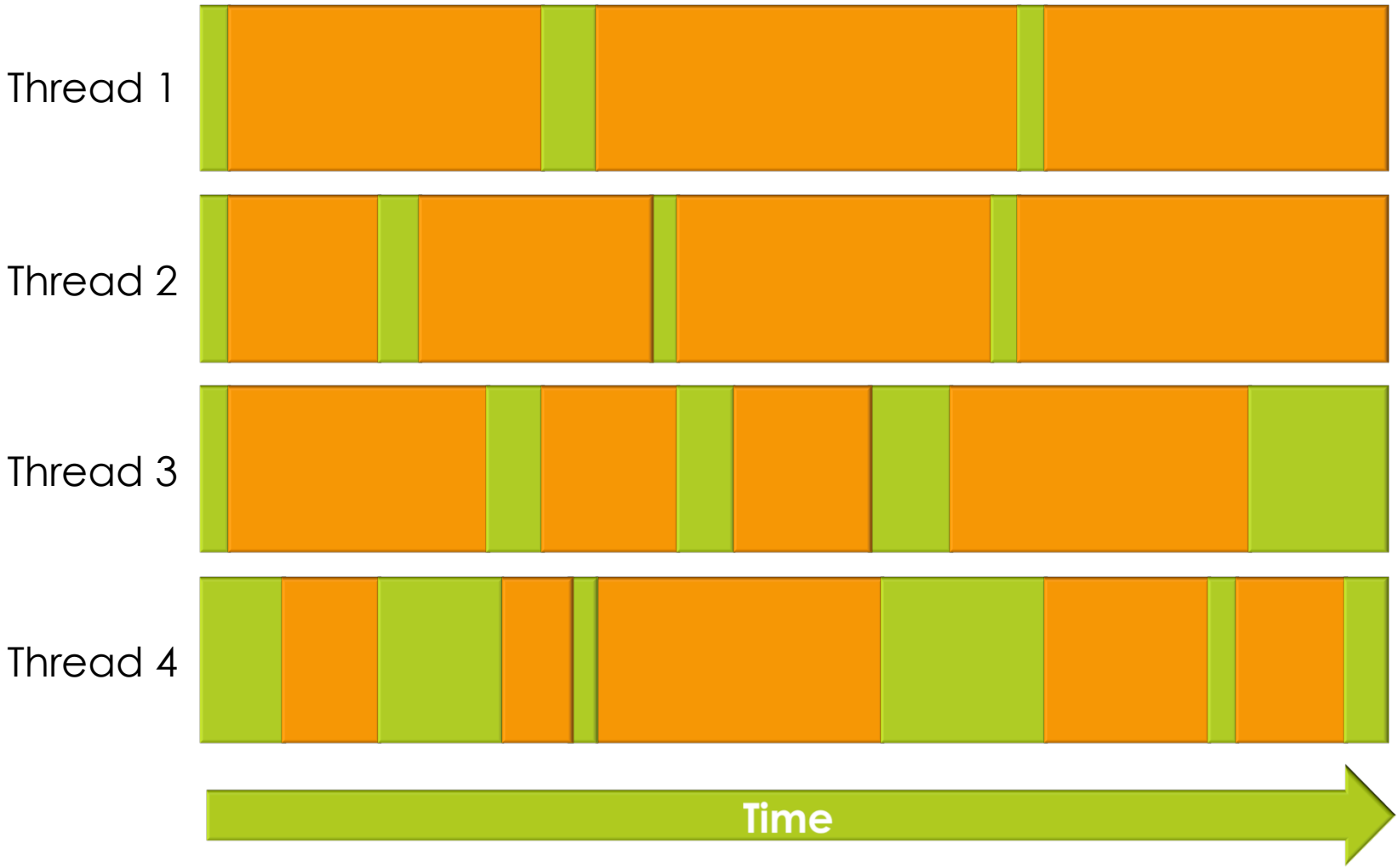
The life of a worker…



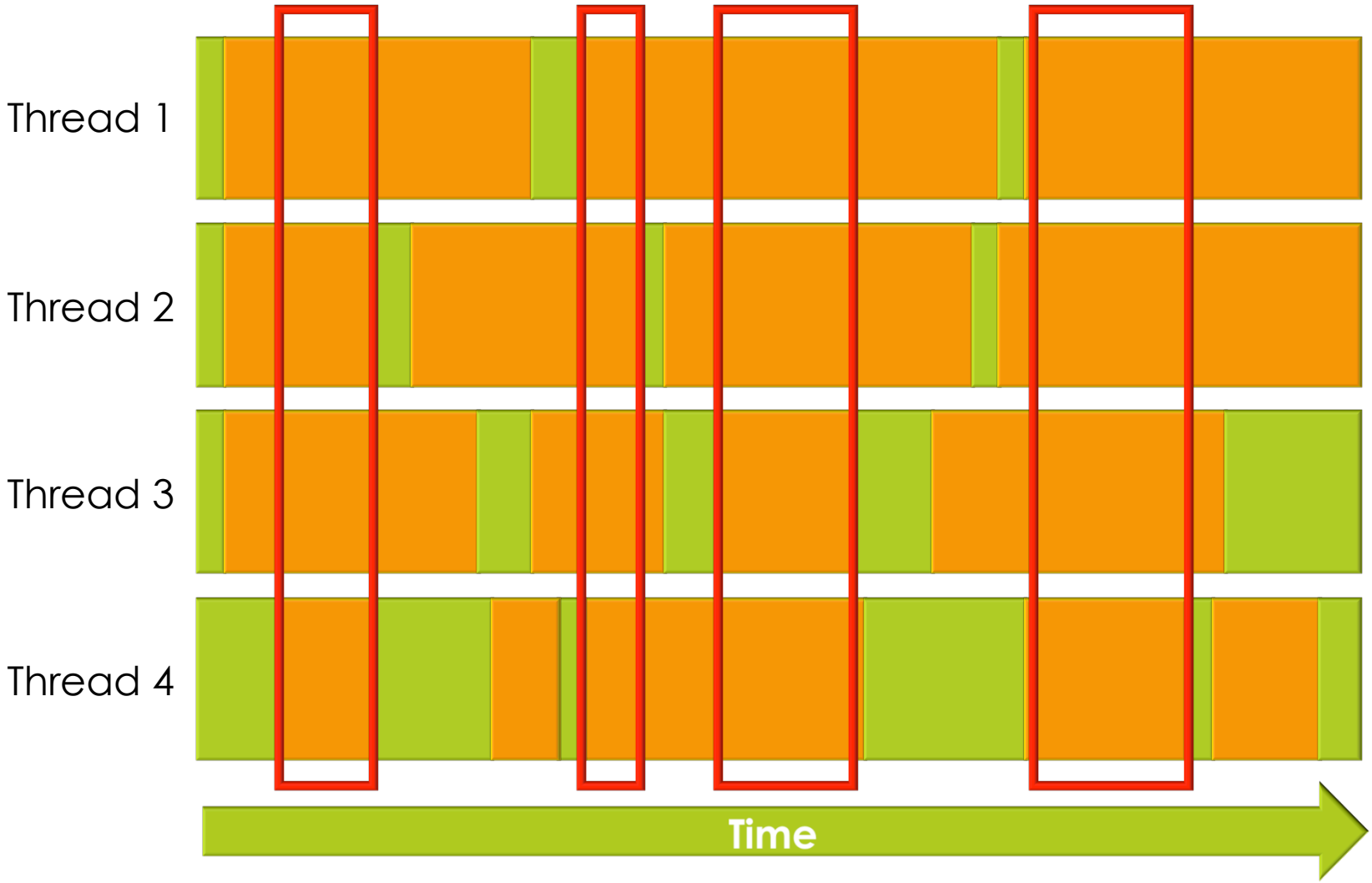
The Other Basic Idea

Writing (Good) Threaded Code is DIFFICULT

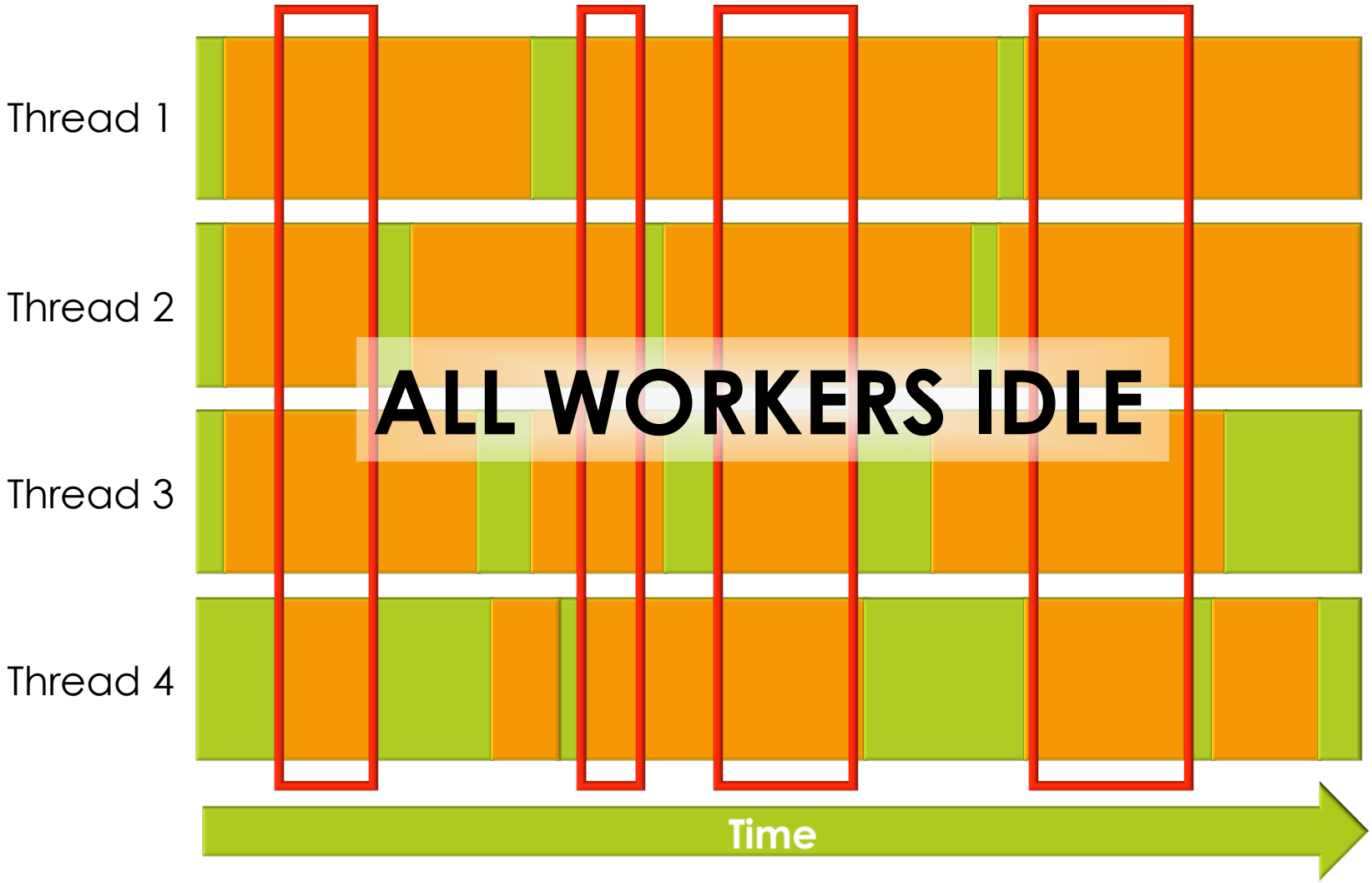
The life of N workers…



The life of N workers…



The life of N workers…



Even worse…

- If all threads are in use, every incoming connection is blocked

  - This can cause massive traffic jams on high-throughput applications

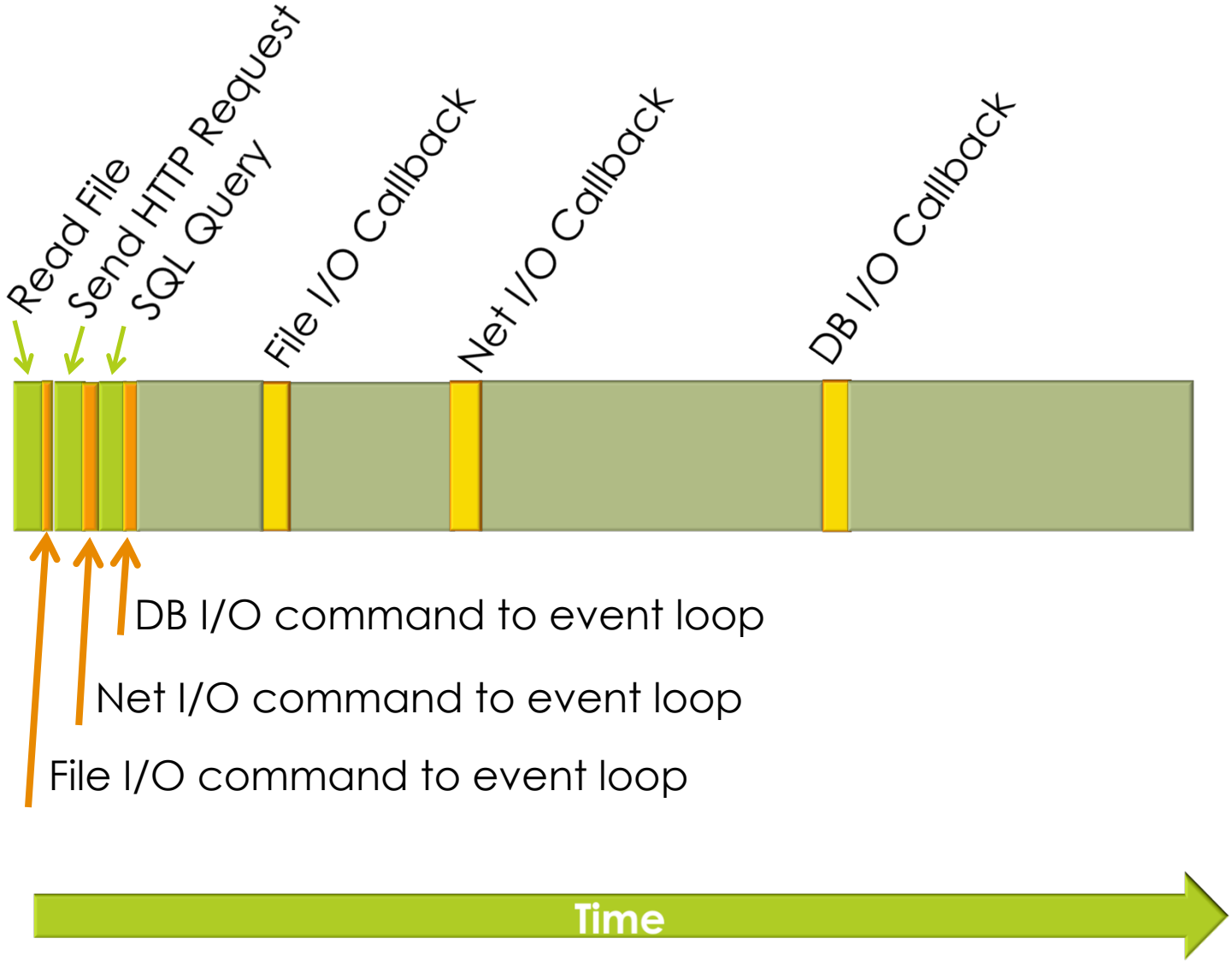
The Node.js way

- Problem:
  - Multi-Threaded code
    - Is difficult to write
    - Is difficult to debug
    - Sucks up more dev/test/maintenance cycles
    - Most often has inefficient performance
- Conclusion:
  - Write code using a **single thread**

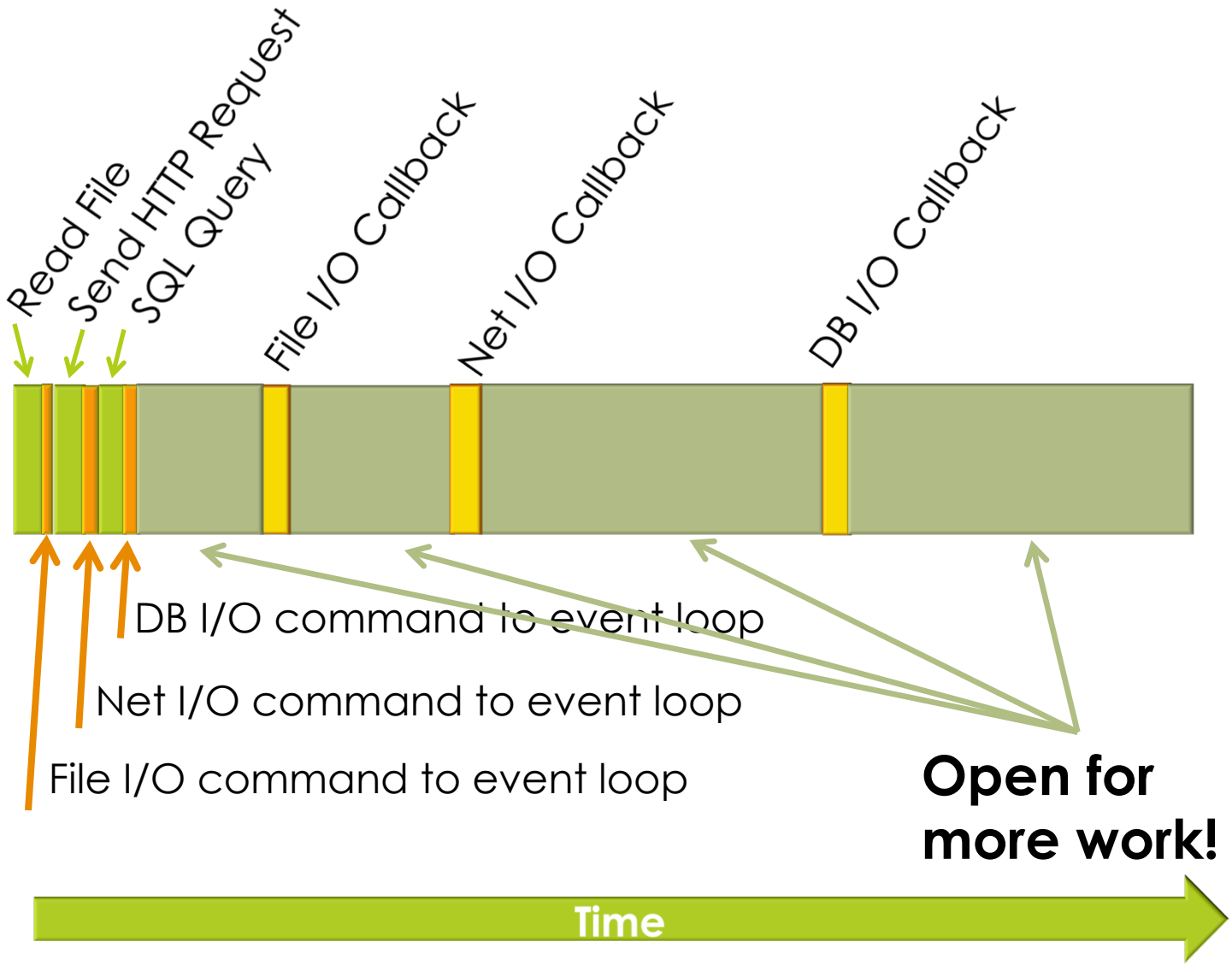
Node.js Event Loop

- Event Loop (from Wikipedia):
  - A "construct that waits for and dispatches events or messages in a program"
- Instead of performing I/O ourselves, we dispatch I/O events to Node's event loop
  - It handles threads, process optimization, concurrency, etc

Node.js Event Loop



Node.js Event Loop



Node.js app code…

- Is run entirely in a single thread
- Passes I/O requests to the event loop, along with callbacks
- Your code then:
  - Goes to sleep
  - Uses no system resources
  - Will be notified via callback when I/O is complete

Callback example

```
var filename = “test_file.txt”;

fs.open(filename, “w”, function(err,
file) {
    if (err) throw err;
});
```

Callback example

Filesystem module forwards task to event loop

```
var file = (“test_file.txt”);

fs.open(file, “w”, function(err, file) {
    if (err) throw err;
});
```

Callback example

Callback is invoked when work is complete

```
var file = ("test_file.txt");

fs.open(file, "w", function(err, file) {
    if (err) throw err;
});
```

This is not magic

- The following:

```
for(i=0; i<5; i++) {
    sleep(1000);
}
```

- Will block the entire Node event loop for 5 seconds

Node is in charge

- Let Node.js handle
  - Dispatch
  - Concurrency
  - (most) Async operations

- What Node doesn't promise:
  - To not block when you tell it to
  - Order of execution (e.g. forked parallel processes)