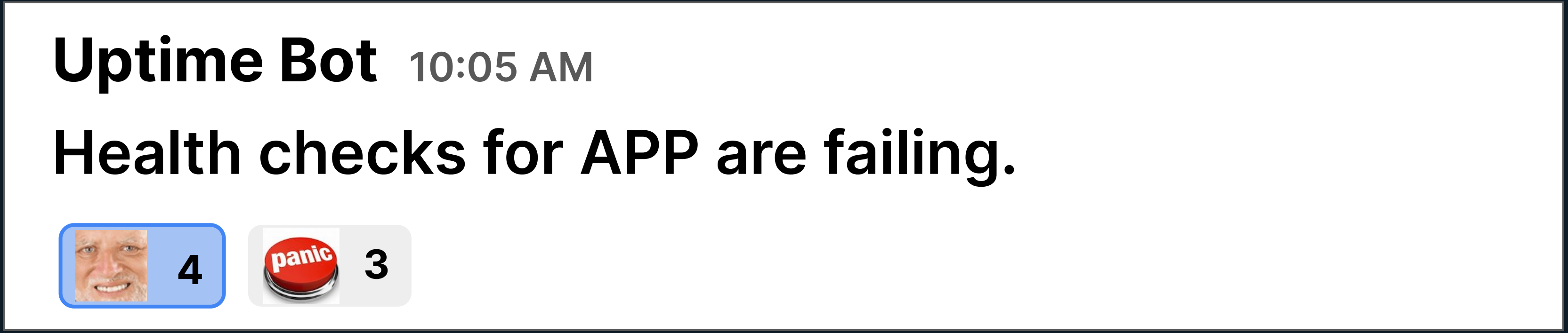
Thrown for an Event Loop

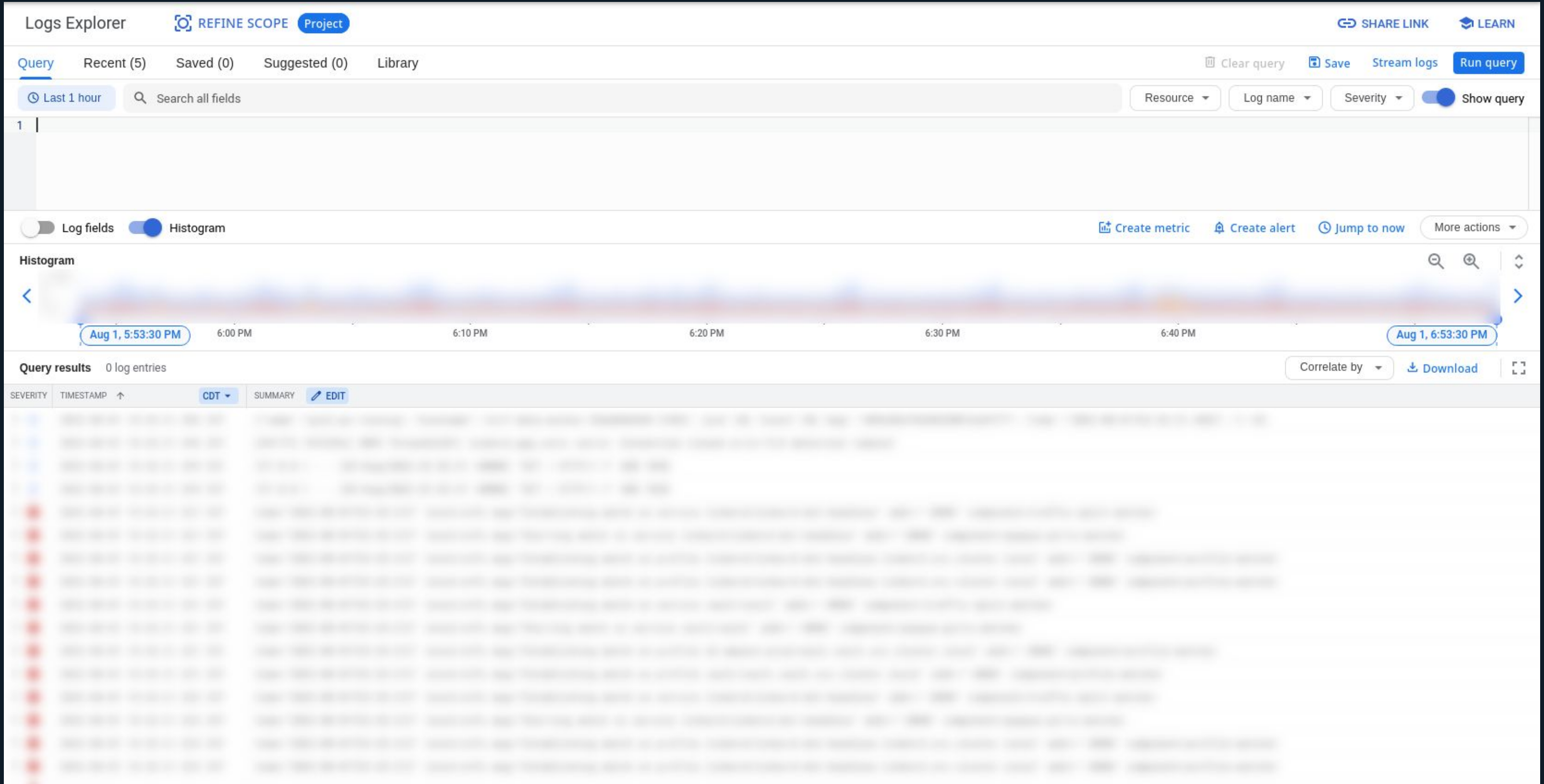
supplypike



Uptime Bot 10:05 AM

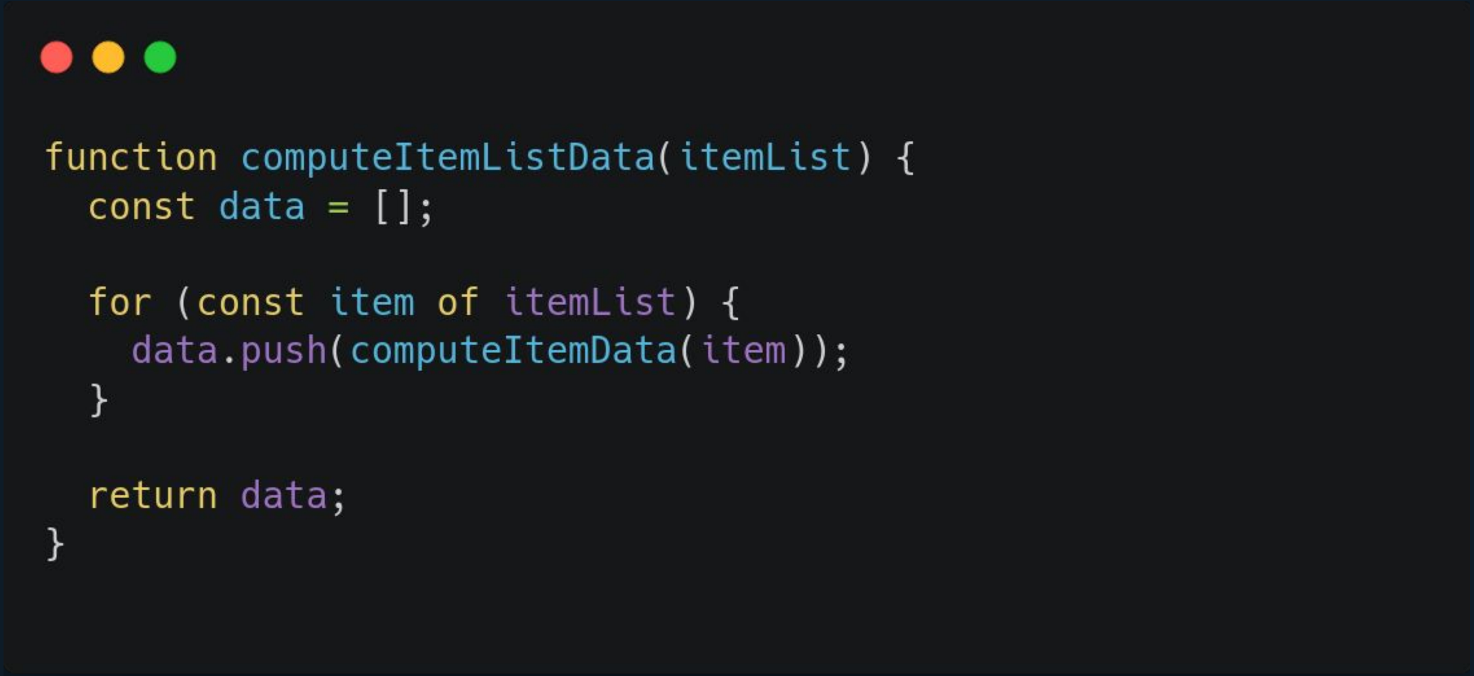
Health checks for APP are failing.

We dig into the logs



Requests are timing out right after we kick off a specific job

The Culprit*



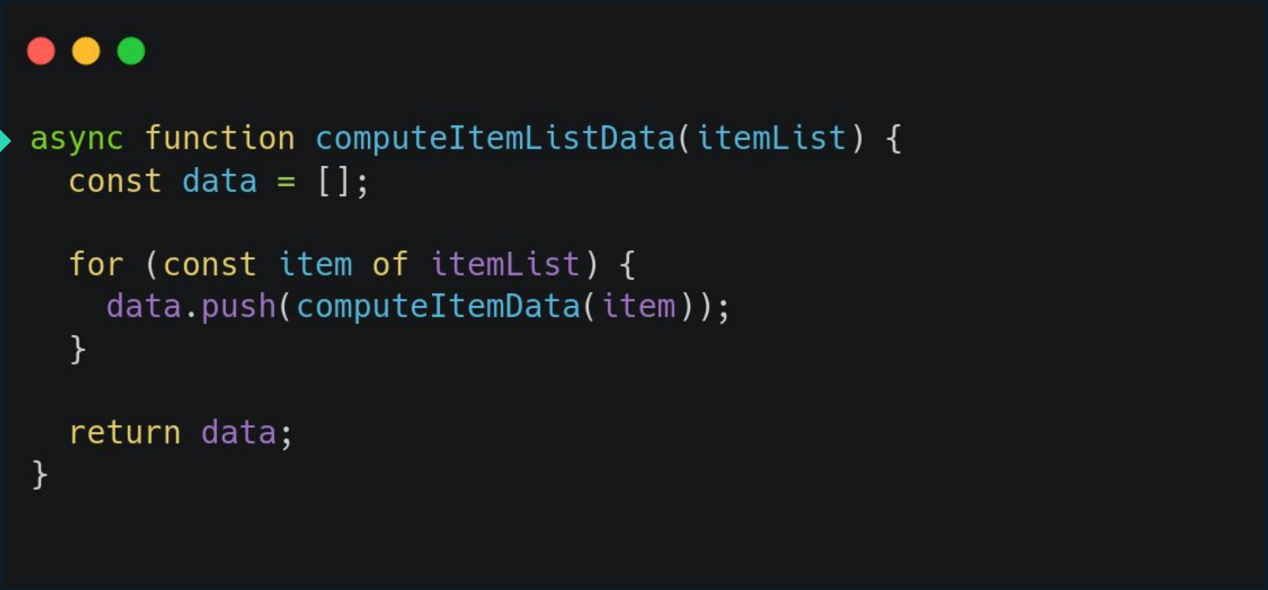
```
function computeItemListData(itemList) {
  const data = [];

  for (const item of itemList) {
    data.push(computeItemData(item));
  }

  return data;
}
```

*Based on a true story

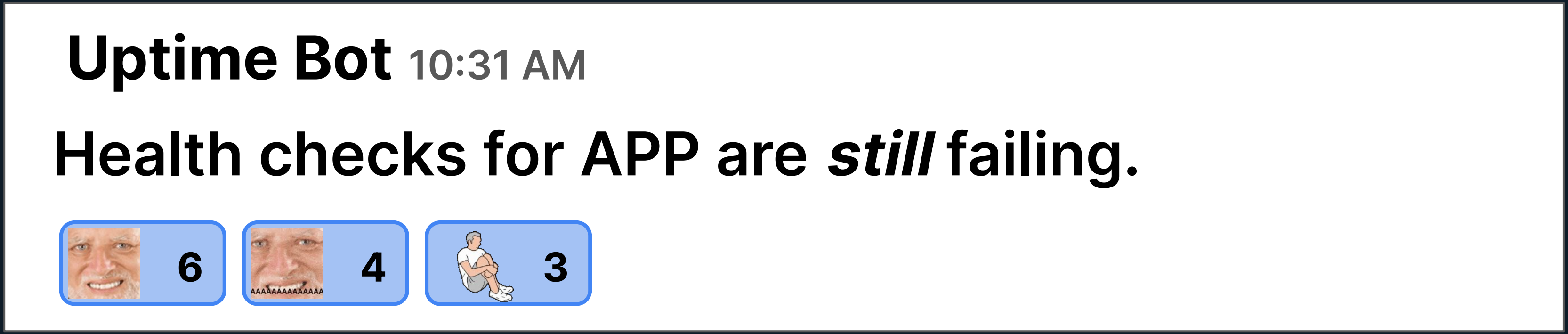
“Blocked event loop?” Easy.



```
async function computeItemListData(itemList) {
  const data = [];

  for (const item of itemList) {
    data.push(computeItemData(item));
  }

  return data;
}
```



Uptime Bot 10:31 AM

Health checks for APP are *still* failing.

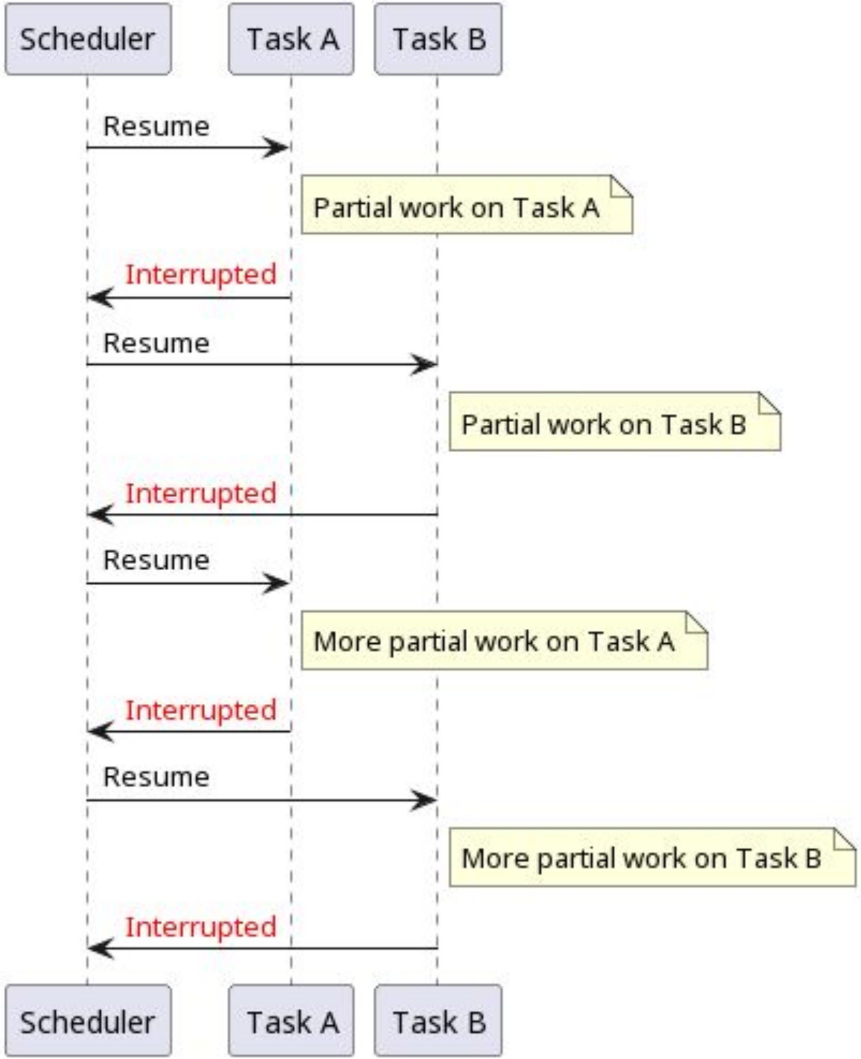
Let’s take some time to really figure this out

Preemptive Scheduling

Execution is interrupted at arbitrary points.

For example:

Anything using separate OS processes or threads

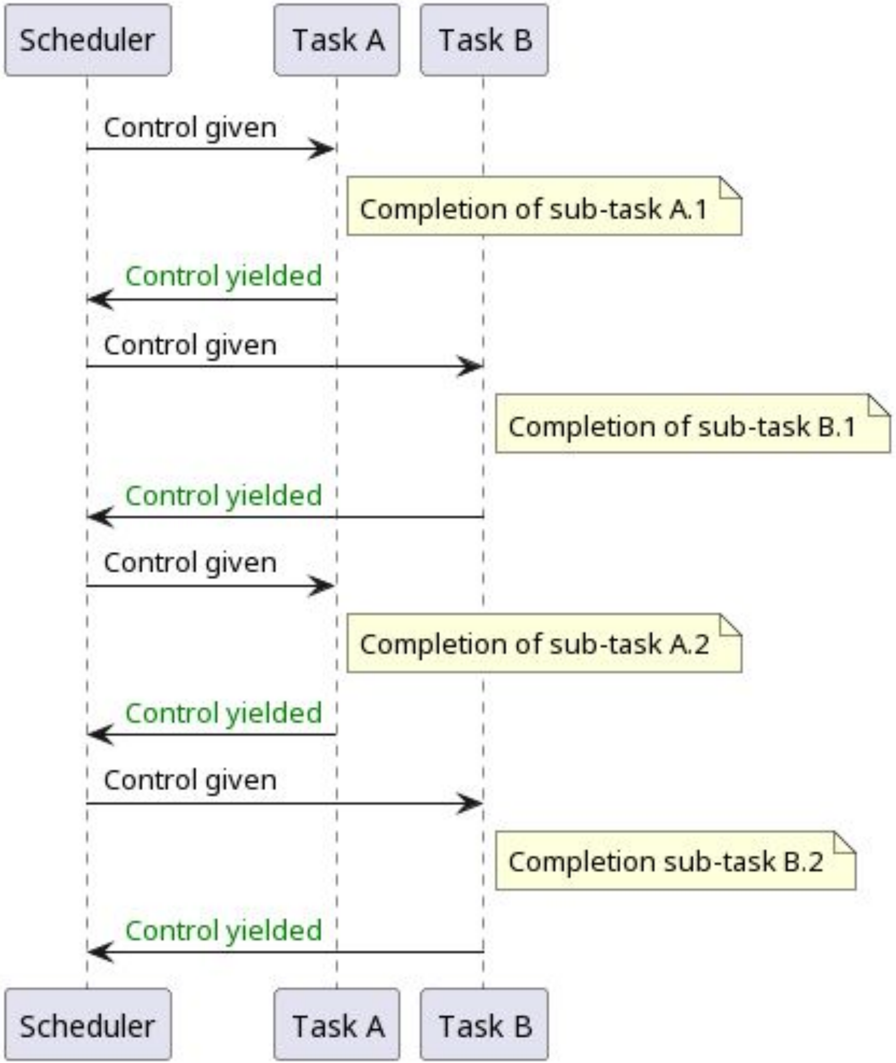


Cooperative Scheduling

Has explicit point where execution is suspended.

For example:

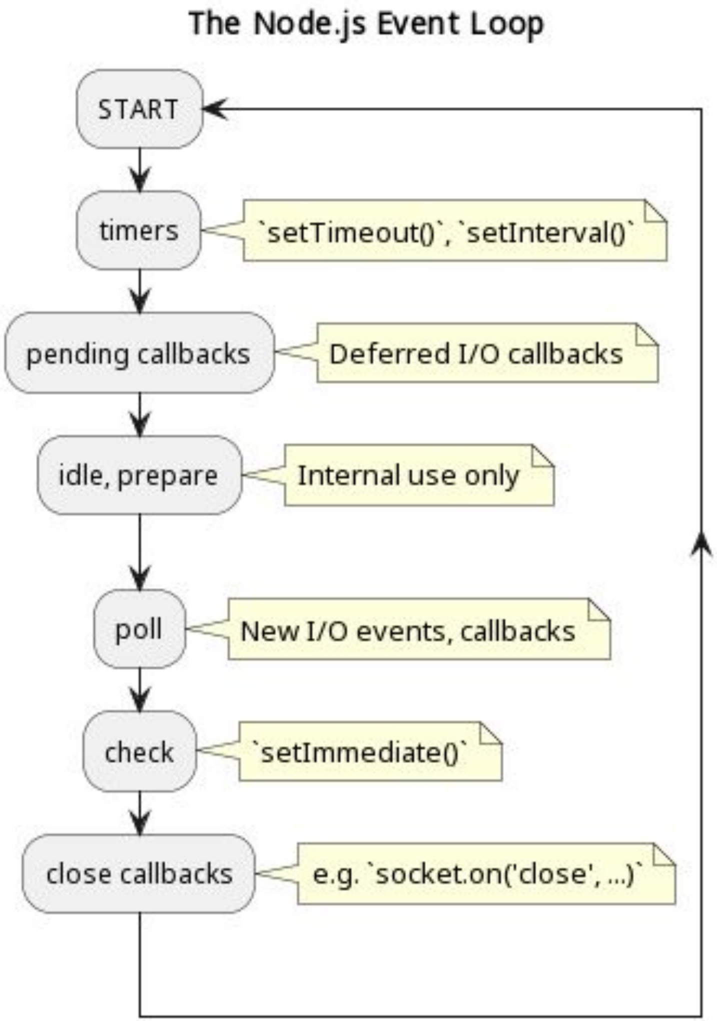
- NGINX modules
- Python asyncio
- Node.js



Node.js

- Ryan Dahl inspired by NGINX and Rack; Web-centric
- Single-threaded, async I/O using event-loop
- Google’s V8 + standard library for I/O

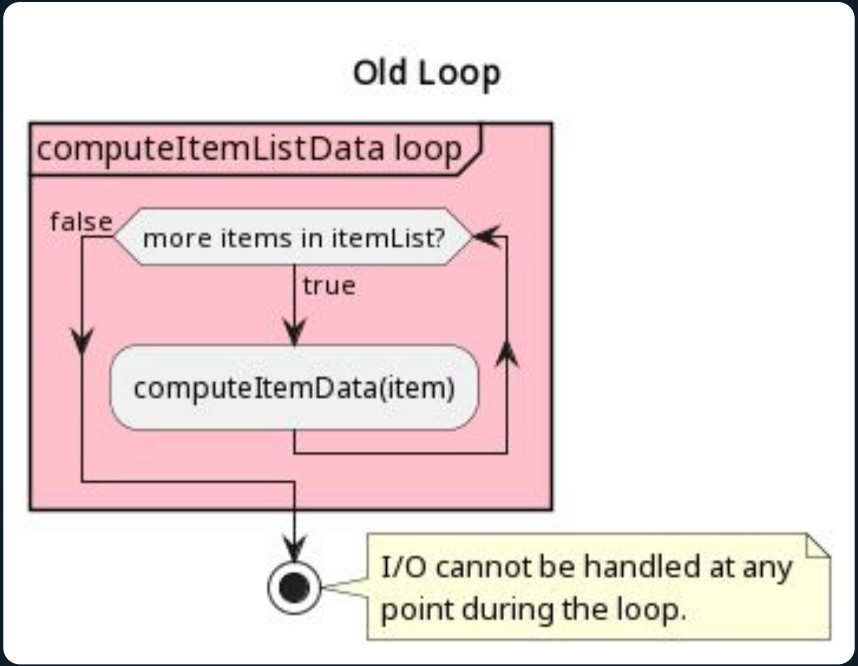
Node.js Event Loop

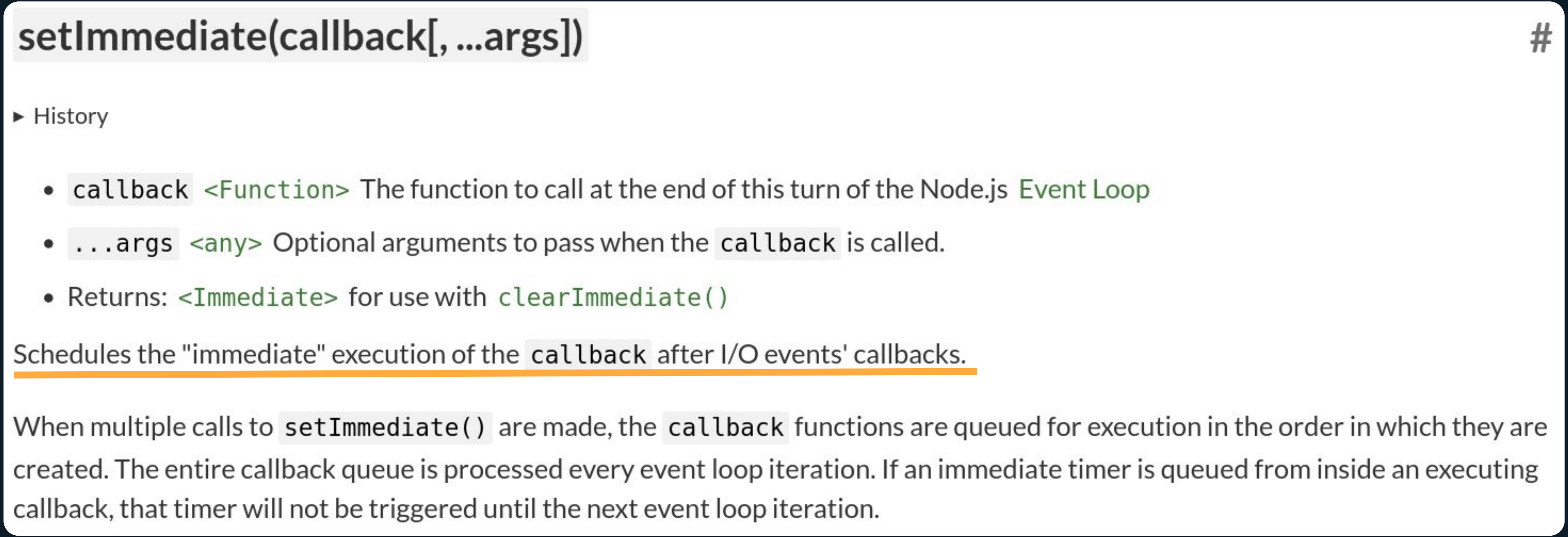


Back to the issue at hand

No async of any kind involved here, so no other I/O is handled

How can we simply yield control without involving I/O?





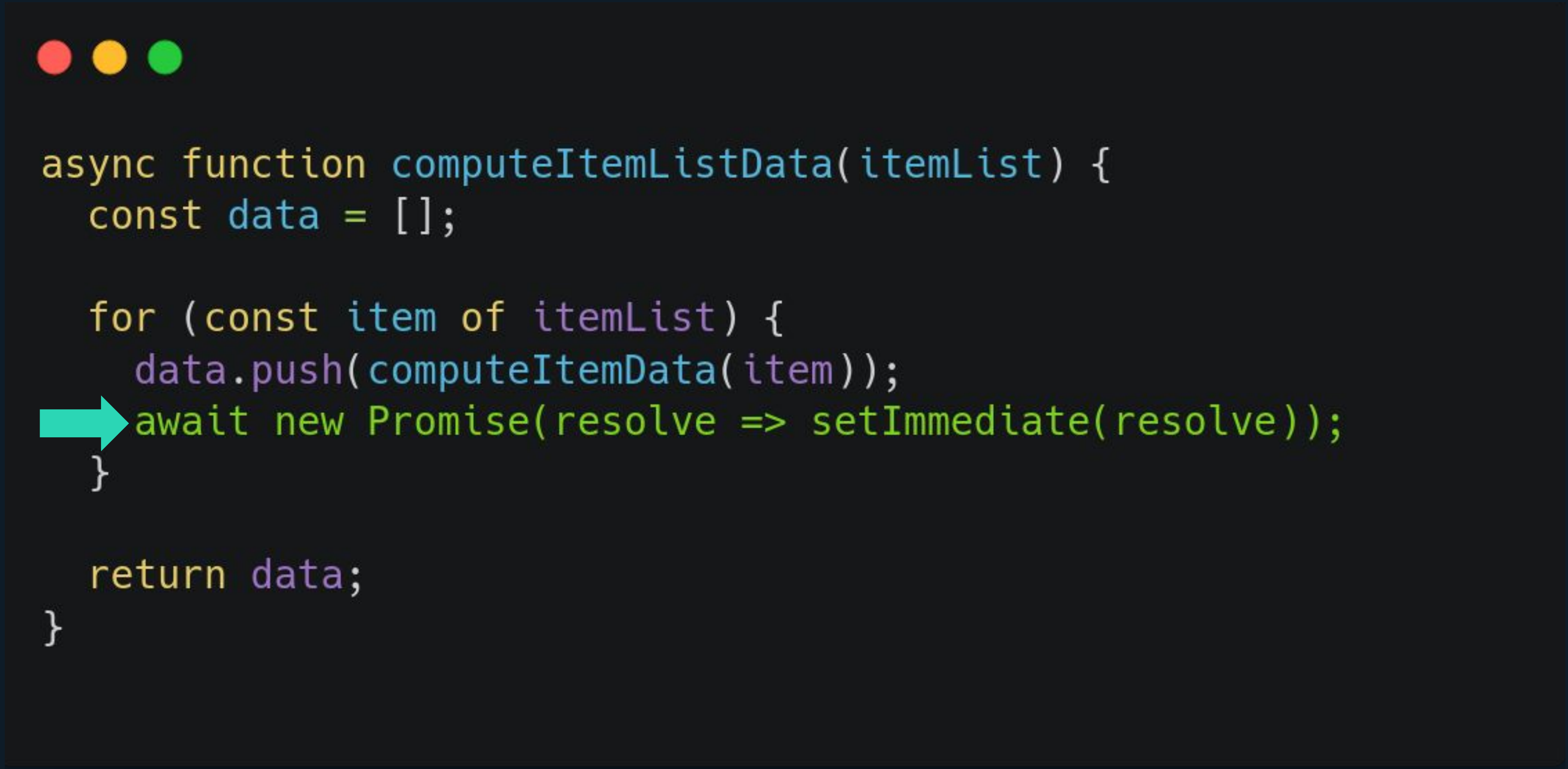
setImmediate(callback[, ...args])

▸ History

- `callback` `<Function>` The function to call at the end of this turn of the Node.js Event Loop
- `...args` `<any>` Optional arguments to pass when the `callback` is called.
- Returns: `<Immediate>` for use with `clearImmediate()`

Schedules the "immediate" execution of the `callback` after I/O events' callbacks.

When multiple calls to `setImmediate()` are made, the `callback` functions are queued for execution in the order in which they are created. The entire callback queue is processed every event loop iteration. If an immediate timer is queued from inside an executing callback, that timer will not be triggered until the next event loop iteration.

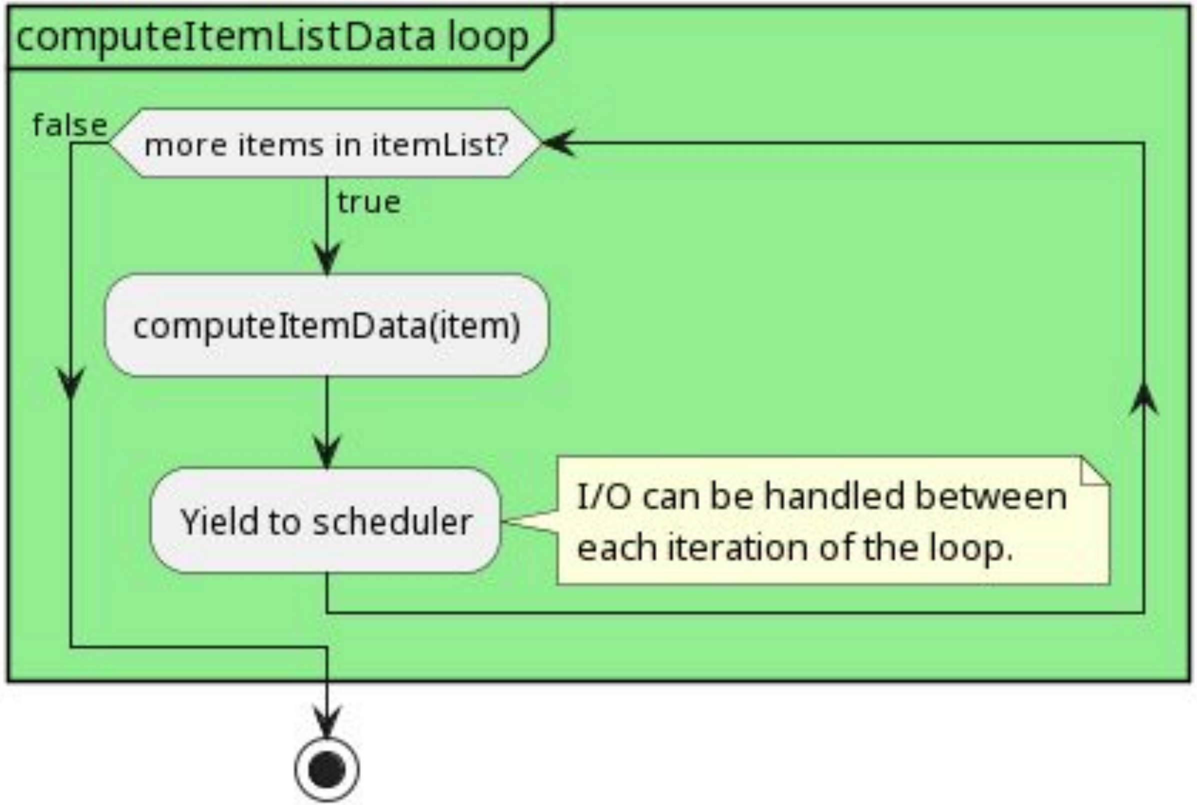


```
async function computeItemListData(itemList) {
  const data = [];

  for (const item of itemList) {
    data.push(computeItemData(item));
    await new Promise(resolve => setImmediate(resolve));
  }

  return data;
}
```

New Loop



I/O can be handled between each iteration of the loop.

Caveats

- Overheard associated with ‘setimmediate()’
- Some code may need significant structural change
  - i.e. function coloring
- Chunk size influenced by requirements and hardware, determined by experimentation

Overhead Table*

Chunk ms (computeItemData)	Sync. full ms (computeItemListData)	Async. full ms (computeItemListData)	Async. time mult.
1	810	837	1.03x
0.1	88	110	1.25x
0.05	43	68	1.68x
0.01	12	31	2.58x

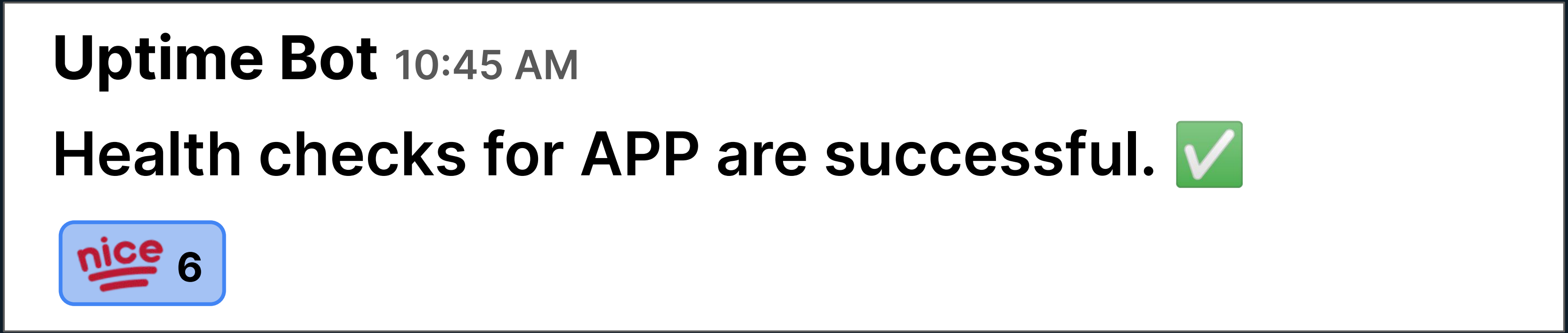
*The numbers will obviously vary by specs,
but the relative effect of overhead should be similar

Upsides

- Simpler to start with
- No need for GIL
- Lighter than threads or processes

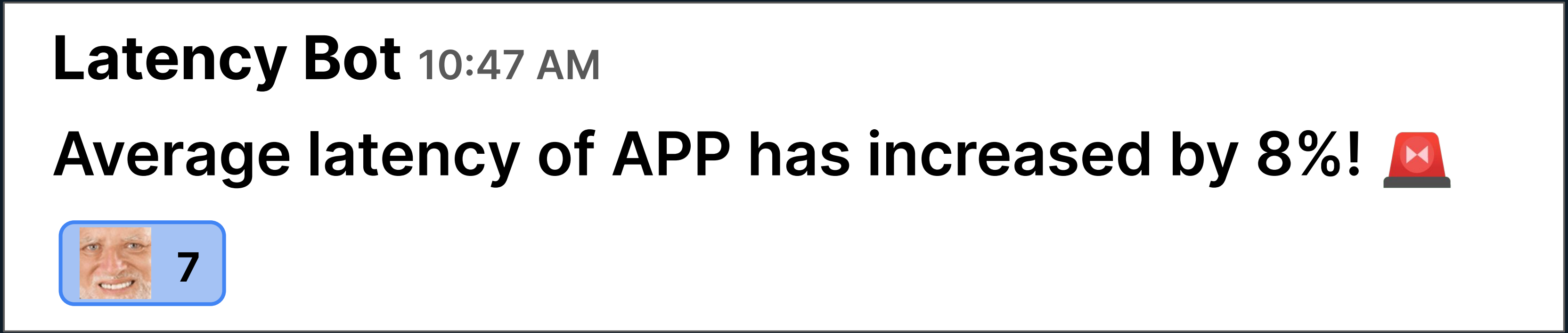
Downsides

- Blocking the event loop with CPU bound code
- Multiple processes to scale out for multi-core
    - And now worker threads
- Function coloring



Uptime Bot 10:45 AM

Health checks for APP are successful. ✅



Latency Bot 10:47 AM

Average latency of APP has increased by 8%! 🚨