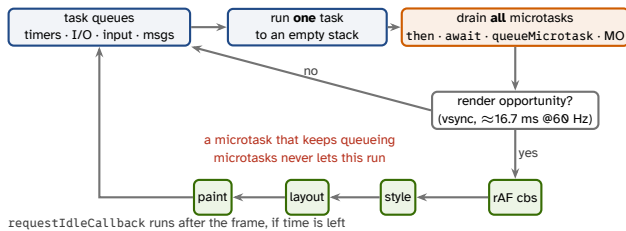
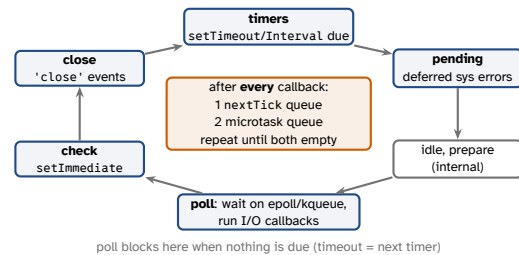


In one line: One thread runs JS **to completion**; when the **call stack** empties the host takes **one task** (macrotask), then **drains the whole microtask queue**, then (browser) maybe **renders**. Node's loop is libuv's **phases**, with `process.nextTick` and then microtasks drained **after every callback**.

1 browser (HTML spec loop)



2 Node (libuv phases)



How it works

- **Stack** of frames, objects in the GC **heap**; nothing interrupts running JS (run-to-completion, no data races).
- **Task** (macrotask) sources: timers, network/I/O, user input, `postMessage` / `MessageChannel`, `setImmediate` (Node). The loop picks **one**.
- **Microtasks**: promise reactions (then/catch/finally, each await resumption), `queueMicrotask`, `MutationObserver`. Checkpoint = run until the queue is **empty**, including ones queued meanwhile $\Rightarrow$ **starvation**: function `f(){Promise.resolve().then(f)}` freezes the page (no task, no paint). A `setTimeout` loop does not.
- **Rendering** is a loop step, not a task; `requestAnimationFrame` = “just before the next paint” (paused in background tabs). Reading layout after a write forces sync layout.
- `setTimeout(f, 0)` = “a task, no earlier than now”. HTML clamps to ≥ 4 ms **once nesting level** > 5 ; background tabs throttle to ≥ 1 s.
- **Node**: `setTimeout(0)` is coerced to 1 ms. Since **Node 11**, `nextTicks` + microtasks run between *each* timer/immediate callback (as browsers do), not once per phase. Recursive `nextTick` starves I/O like microtasks.
- **setImmediate vs setTimeout(0)**: in the main module the order is **nondeterministic** (is 1 ms already up?). Inside an I/O callback **immediate always wins**: `poll` $\rightarrow$ `check` comes before timers.

Puzzle — trace it

```
console.log('A');
setTimeout(() => console.log('B'), 0); // task
Promise.resolve().then(() => {
  console.log('C'); // micro [C]
  queueMicrotask(() => console.log('D')); // micro, queued by C
});
(async () => { console.log('E'); // body runs sync
  await null; console.log('F'); // micro [C,F]
})();
process.nextTick(() => console.log('G')); // Node only
console.log('H');
```

- Sync: **A E H**. Micro queue `[C, F]`; `C` runs, appends `D` $\rightarrow$ `[F, D]`.
- **Browser** (no `G` line): **A E H C F D B**.
- **Node, CommonJS**: tick queue first $\rightarrow$ **A E H G C F D B**.
- **Node, ES module**: the body runs inside a promise job, microtasks drain first $\rightarrow$ **A E H C F D G B**.

```
setTimeout(() => log('t'), 0); setImmediate(() => log('i'));
// main module: "t i" OR "i t" -- depends on process timing
fs.readFile(file, () => { // we are in the poll phase
  setTimeout(() => log('t2'), 0);
  setImmediate(() => log('i2')); // always: i2, t2
});
```

Long tasks and yielding

- **Long task** = > 50 ms on the main thread: input, paint, `requestAnimationFrame` all wait (INP).
- `await` on a resolved promise does **not** yield — it is a microtask. Yield = end the task: `scheduler.yield()` (Chromium; keeps priority), `scheduler.postTask(fn, {priority})`, `setTimeout(r, 0)` (clamped), `MessageChannel` (unclamped; React's scheduler, ≈ 5 ms slices). Chunk: every `N` items await `new Promise(r => setTimeout(r, 0))`. Or move it off-thread: `Web Worker` / `worker_threads`.

React Native and iOS

- RN runs one JS event loop on its **JS thread** (Hermes, microtasks real); a long task there drops JS frames and delays touch handling while UI-thread animations keep going — budgets in *rn-performance*, threads and JSI in *rn-architecture*.
- iOS: main `RunLoop` $\approx$ the loop; `DispatchQueue.main.async` $\approx$ a task; the Core Animation commit at the end of a turn $\approx$ the render step. No microtask queue: a `@MainActor` hop is enqueued work.

Interview traps

- ! “`setTimeout(0)` runs next” — no: every microtask (and in Node every `nextTick`) goes first, then the clamp.
- ! `new Promise(executor)` runs the executor **synchronously**; only the reactions are async.
- ! async does not mean “another thread”: CPU work in an async function still blocks the loop.
- ! `nextTick` fires *before* promises in CJS but *after* them in ESM top-level code; prefer `queueMicrotask` (portable).

Remember

One task $\rightarrow$ all microtasks $\rightarrow$ maybe paint. Node: phases, and `nextTick` $\rightarrow$ micro after every callback.

Likely questions

1. Microtask vs task? — micro drains fully after each task/callback; tasks run one per turn.
2. Why does the UI freeze with only promises? — the checkpoint never ends, so no render step.
3. `setImmediate` vs `setTimeout(0)`? — check vs timers phase; ordered only inside I/O callbacks.
4. How to split a 2 s job? — chunk & yield via a task (`scheduler.yield()`), or a worker.