Node.js Event Loop

2024, OSSCA, Node.js - Code and Learn

Daeyeon Jeong

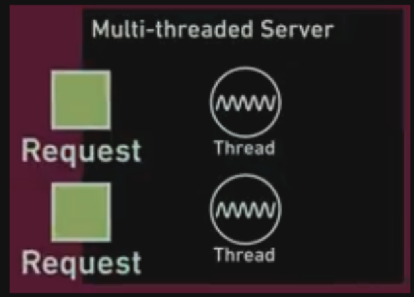
Index

- Node.js single-threaded model
- libuv and blocking/non-blocking operations handling
- Node.js event loop and phases

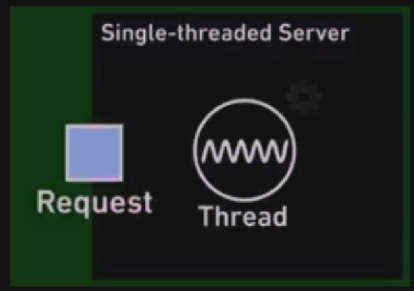
Execution Model

Server Execution Model

- Multi-threaded model (e.g., Apache HTTP Server)
  - Each client request gets its own thread (MPM - Worker + Prefork).
  - Requires managing synchronization (e.g., Mutex, Semaphore)



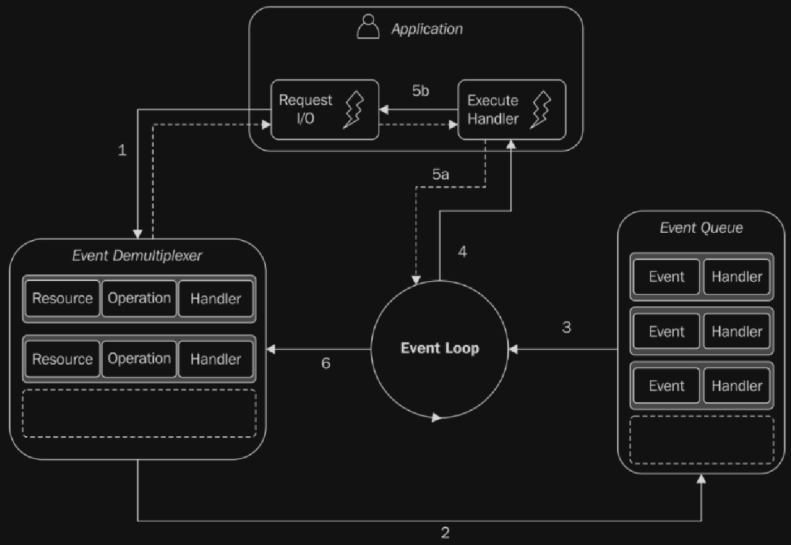
- Single-threaded model (e.g., Node.js HTTP Server)
  - Handles multiple client requests using a single thread.
  - Avoids synchronization issues.
  - Reduces context switching overhead and conserves resources.



Event-Driven Architecture

Handling Asynchronous Requests

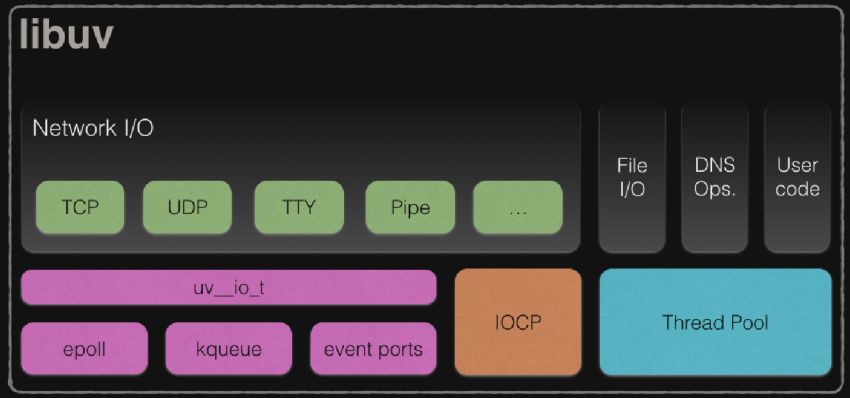
- Reactor Pattern
  - Components : Resources, Event De-multiplexer (Demux), Event Queue, Event Loop
  - Event Demux takes requests, delegates them to resources, and queues triggered events.
  - Manages asynchronous events without multi-threading overhead.



libuv

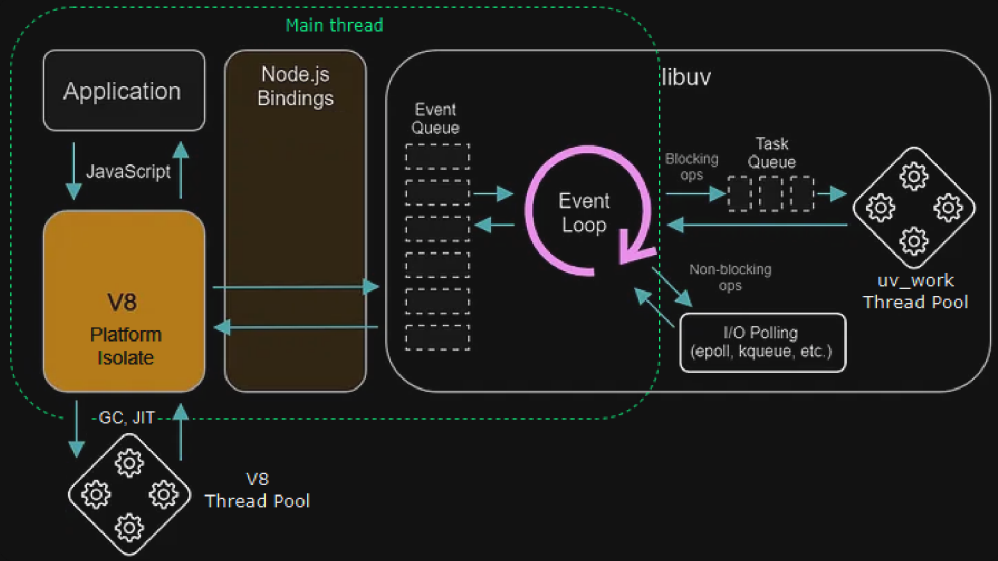
Event-driven Asynchronous I/O

- Cross-platform support library originally written for Node.js :
  - TCP/UDP sockets (`node:net/dgram`), DNS resolution (`node:dns`), TTY (`node:tty`)
  - File (`node:fs`), Child processes (`node:child-process`), HR clock (`process.hrtime`), ...
- A worker thread pool for blocking tasks
- Event loop backed by epoll, kqueue, IOCP, event ports, io_uring, ...



libuv in Node.js

Managing Event Loop and I/O Operations



- Node.js uses libuv to manage its event loop and cross-platform I/O.
- Userland code runs on a single-thread, the `main thread`. (`worker_threads` isn't for I/O)
  - Internally, additional threads run on the libuv and V8 thread pools.

libuv in Node.js

Handling of Operations

- Blocking operations are handled by threads :
  - Examples: crypto, zlib, sqlite, …
- Non-Blocking operations are handled by I/O Polling (uv_tcp, uv_udp, uv_fs_t, uv_tty, …) :
  - Examples: net, fs, tty, …

Blocking Operations with Threads

Case: `node:crypto` using `uv_work`

```
const { pbkdf2 } = require('node:crypto');

const doExpensiveHashing = () => {
  pbkdf2('password', 'salt', 100000, 512, 'sha512', () => { ... });
};

doExpensiveHashing();
```

```
// lib/internal/crypto/pbkdf2.js
function pbkdf2(password, salt, iterations, keylen, digest, callback) {
  const job = new PBKDF2Job(...);
  job.ondone = (err, result) => { ... };
  job.run();
}
```

Blocking Operations with Threads

Case: `node:crypto` using `uv_work`

```cpp
// src/crypto/crypto_util.h
static void Run(const v8::FunctionCallbackInfo<v8::Value>& args) {
  ...
  if (job->mode() == kCryptoJobAsync)
    return job->ScheduleWork();
}
```

```cpp
// src/threadpoolwork-inl.h
void ThreadPoolWork::ScheduleWork() {
  ...
  int status = uv_queue_work(
      env_->event_loop(),
      &work_req_,
      [](uv_work_t* req) {
        ThreadPoolWork* self = ContainerOf(&ThreadPoolWork::work_req_, req);
        self->DoThreadPoolWork();
      },
      [](uv_work_t* req, int status) {
        ThreadPoolWork* self = ContainerOf(&ThreadPoolWork::work_req_, req);
        self->AfterThreadPoolWork(status); // This calls the "ondone" callback.
      });
}
```

Non-Blocking Operations with I/O Polling

Case : TCP Socket with `epoll` (Optimized for linux)

```c
void on_new_connection_callback(int client_fd) { ... }

int main() {
  server_fd = socket(AF_INET, SOCK_STREAM, 0);
  server_addr.sin_port = htons(PORT);
  bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
  listen(server_fd, 10);
  epoll_fd = epoll_create1(0); // OS specific
  ...
  // Run epoll loop
  while (1) {
    int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1); // FD status is notified only on changes in kernel.
    for (int i = 0; i < nfds; ++i) {
      if (events[i].data.fd == server_fd) {
        client_fd = accept(server_fd, (struct sockaddr *)&client_addr, ...);
        on_new_connection_callback(client_fd);
      } else {
        read(events[i].data.fd, buffer, READ_BUFFER_SIZE); ...
      }
    }
  }
}
```

Non-Blocking Operations with I/O Polling

Case : TCP Socket with `libuv` (Use OS-optimized APIs)

```c
void on_new_connection_callback(uv_stream_t* handle, int status) { ... }

int main() {
  uv_tcp_t handle;
  struct sockaddr_in addr;

  uv_tcp_init(uv_default_loop(), &handle);
  uv_ip4_addr("0.0.0.0", PORT, &addr);``
  uv_tcp_bind(&handle, (const struct sockaddr*)&addr, 0);
  uv_listen((uv_stream_t*) &handle, 128, on_new_connection_callback);

  // Run uv loop
  uv_run(uv_default_loop(), UV_RUN_DEFAULT);

  return 0;
}
```

Non-Blocking Operations with I/O Polling

Case: TCP Socket with Node.js

```
const net = require('node:net');
const on_new_connection_callback = (socket) => { ... };
const server = net.createServer(on_new_connection_callback);

server.listen(PORT, () => { ... });
```

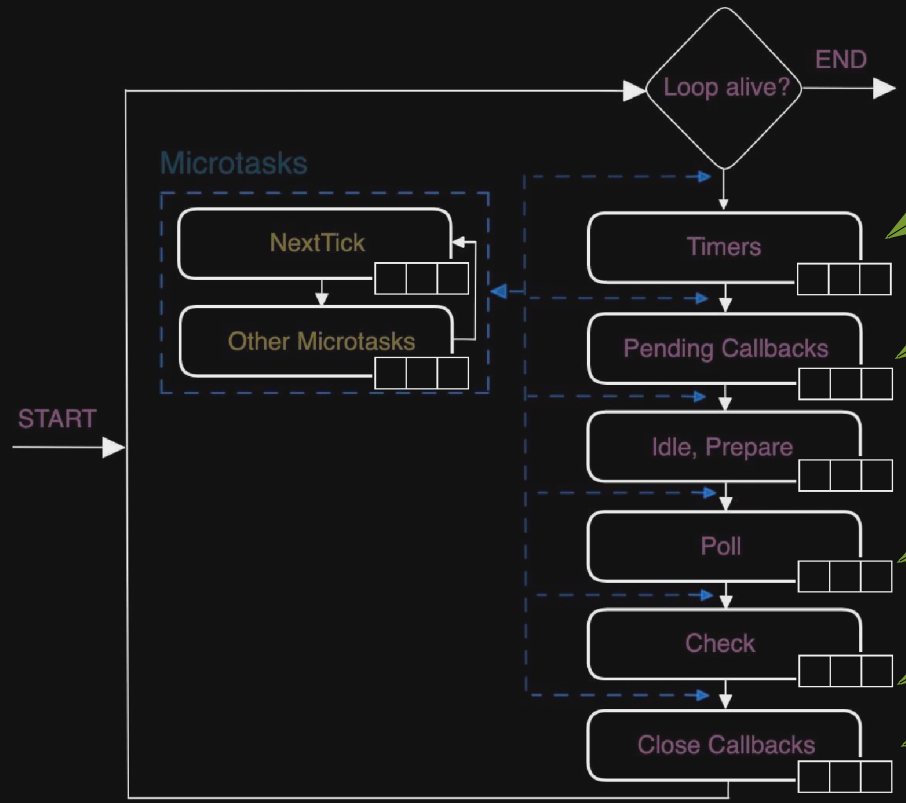
```
// src/tcp_wrap.cc
void TCPWrap::Listen(const FunctionCallbackInfo<Value>& args) {
  ...
  if (!args[0]->Int32Value(env->context()).To(&backlog)) return;
  int err = uv_listen(reinterpret_cast<uv_stream_t*>(&wrap->handle_),
                      backlog,
                      OnConnection); ...
}
```

```
// src/api/embed_helpers.cc
do {
    if (env->is_stopping()) break;
    uv_run(env->event_loop(), UV_RUN_DEFAULT);
    if (env->is_stopping()) break; ...
} while (more == true && !env->is_stopping());
```

```
uv_run(env->event_loop(), UV_RUN_DEFAULT);
```

Node.js Event Loop

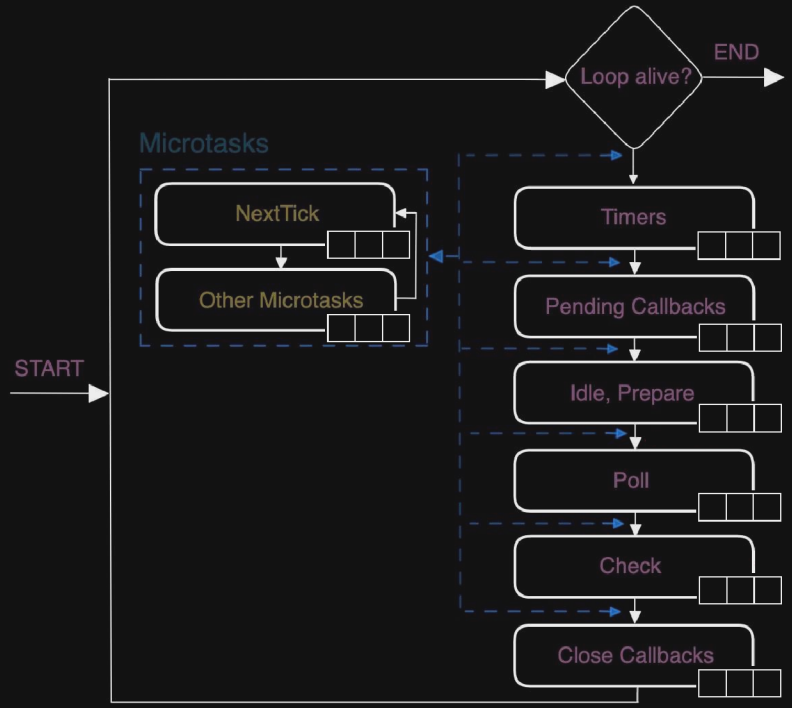
Phases Inside uv_run()




```c
int uv_run(uv_loop_t* loop, uv_run_mode mode) {
  int timeout;
  int r;
  int can_sleep;

  r = uv__loop_alive(loop);
  if (!r)
    uv__update_time(loop);

  if (mode == UV_RUN_DEFAULT && r != 0 && loop->stop_flag == 0) {
    uv__update_time(loop);
    uv__run_timers(loop);
  }

  while (r != 0 && loop->stop_flag == 0) {
    can_sleep =
        uv__queue_empty(&loop->pending_queue) &&
        uv__queue_empty(&loop->idle_handles);

    uv__run_pending(loop);
    uv__run_idle(loop);
    uv__run_prepare(loop);

    timeout = 0;
    if ((mode == UV_RUN_ONCE && can_sleep) || mode == UV_RUN_DEFAULT)
      timeout = uv__backend_timeout(loop);

    uv__metrics_inc_loop_count(loop);

    uv__io_poll(loop, timeout);

    for (r = 0; r < 8 && !uv__queue_empty(&loop->pending_queue); r++)
      uv__run_pending(loop);

    uv__metrics_update_idle_time(loop);

    uv__run_check(loop);
    uv__run_closing_handles(loop);

    uv__update_time(loop);
    uv__run_timers(loop);

    r = uv__loop_alive(loop);
    if (mode == UV_RUN_ONCE || mode == UV_RUN_NOWAIT)
      break;
  }

  if (loop->stop_flag != 0)
    loop->stop_flag = 0;

  return r;
}
```

Node.js Event Loop

Loop Phases and Microtasks



- Each phase has a FIFO queue of callbacks.
- Event Loop
  - Timers : `setTimeout()` and `setInterval()`
  - Pending CBs : Typically used for error handling
  - Idle, Prepare : Used libuv internally
  - Poll : Retrieve I/O events (`fs` , `net` , ...)
  - Check : `setImmediate()`
  - Close CBs : `e.g., socket.on('close', ...)`
- Microtasks
  - `process.nextTick()` , `PromiseJobs` , ...
  - Microtasks are executed between each phases.

Event Queue

Enqueue Timing

- Timer or I/O Task :
  - When a timer or I/O task triggers an event, the registered callback is added to the respective queue. This callback is conceptually considered an event.
  - Errors from I/O tasks are typically enqueued in the `Pending callbacks` queue.
  - Close events are queued in the `Close callbacks` queue after I/O resources are closed.

- Calling `setImmediate` :
  - The given callback is immediately enqueued in the `Check` queue.
  - It’s guaranteed to be executed after the `Poll` phase.

Node.js Microtasks

Microtask Queue and Execution

- Node.js microtask queue : `nextTick queue` + `microtask queue`
  - 🤯 microtasks in the Web Spec focus on async jobs, distinct from `macrotasks (tasks)`.
- Calling `process.nextTick()`, `Promise.then()`, or `queueMicrotask()` queues their callbacks.
- Microtasks only defer execution and do not create a thread.
- All queued tasks will be drained *before* the event loop moves to the next phase.
- `nextTick queue` has higher priority.

Event Loop Starvation

```
function callback() {
  // Recursive process.nextTick() calls prevent moving to the next phase.
  process.nextTick(() => callback());
}

callback();
```

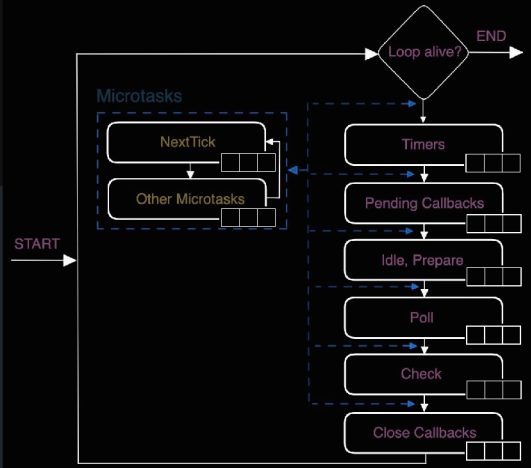
Microtasks Execution

```
// lib/internal/process/task_queues.js
function processTicksAndRejections() {
  do {
    while ((tock = queue.shift()) !== null) { ... // nextTick queue (process.nextTick)
      try {
        const callback = tock.callback;
        if (tock.args === undefined) {
          callback();
        } ...
    }
    runMicrotasks();                              // microtask queue (Promise.then)
  } while (!queue.isEmpty() || processPromiseRejections()); ...
} ...
setTickCallback(processTicksAndRejections);
```

```
// src/api/callback.cc
void InternalCallbackScope::Close() { // Invoked in ~InternalCallbackScope() in RAII manner.
  ...
  Local<Function> tick_callback = env_->tick_callback_function();
  if (tick_callback->Call(context, process, 0, nullptr).IsEmpty()) {
    failed_ = true;
  }
}
```

Event Loop Execution

```
const fs = require('node:fs');

console.log('A - Start');                                    // (A)
setTimeout(() => console.log('B - setTimeout 1'), 0);        // (B)
setImmediate(() => console.log('C - setImmediate'));         // (C)

fs.readFile(__filename, () => {                              // (D)
  console.log('E - readFile');                               // (E)
  setTimeout(() => console.log('F - readFile::setTimeout'), 0);  // (F)
  setImmediate(() => console.log('G - readFile::setImmediate')); // (G)
  process.nextTick(() => console.log('H - readFile::nextTick')); // (H)
});

Promise.resolve().then(() => {                               // (I)
  console.log('J - Promise');                                // (J)
  process.nextTick(() => console.log('K - Promise::nextTick'));  // (K)
  setImmediate(() => console.log('L - Promise::setImmediate'));  // (L)
});

queueMicrotask(() => console.log('M - queueMicrotask'));     // (M)
process.nextTick(() => console.log('N - nextTick'));         // (N)
setTimeout(() => console.log('O - setTimeout 2'), 0);        // (O)
console.log('P - End');                                      // (P)
```




```
A - Start
P - End
N - nextTick
J - Promise
M - queueMicrotask
K - Promise::nextTick
B - setTimeout 1
O - setTimeout 2 *
C - setImmediate
L - Promise::setImmediate
E - readFile
H - readFile::nextTick
G - readFile::setImmediate
F - readFile::setTimeout
```

Event Loop Termination

Reference count

- The event loop runs as long as there are active timers, pending callbacks, or I/O operations.
- The event loop determines its exit using reference counting for active libuv handles or requests.
  - Each activity adds +1 to its reference count, which is reduced by -1 when it complete.
  - Using `.unref()` manually reduces the reference count by -1.

```
const net = require('node:net');

const server = net.createServer((socket) => socket.end());
server.listen(8080, () => console.log('listening'));
server.unref(); // Don't let this server keep the process alive.

setTimeout(() => console.log('done'), 5000);
```

Summary

- Node.js runs one thread for userland code, but spins up multiple threads internally.
- libuv is a cross-platform library originally written for Node.js.
  - It enables single-threaded asynchronous I/O.
  - It abstracts I/O APIs for efficient performance across platforms.
- Node.js event loop phases and microtasks :
  - Timer ➔ Pending callbacks ➔ (Idle, Prepare) ➔ Poll ➔ Check ➔ Close callbacks ↻
  - All microtasks are drained before moving to the next event loop phase.

Thank you

https://github.com/daeyeon/code-and-learn

daeyeon.dev@gmail.com