노드 디자인 패턴 챕터 1, 2

이세호

짧은 후기

- 번역이 별로다...
- pdf 공유해드리겠습니다.

- 내용은 알찼음. 재밌었습니다.

목차

Chap 1

- 비동기 IO
  - 비동기 IO가 효율적이라 이거 써야함.
- 이벤트 멀티플렉싱
  - 이벤트 멀티플렉싱을 해야 비동기 IO 효율적으로 구현할 수 있음.
- Reactor Pattern
- Node.js Recipe

Chapt 2

- esm vs commonjs

애초에 노드는 왜 등장했는가

- 우리는 흔히 스레드 논 블로킹이라고만 생각하고 IO를 빼먹고 있음.
- 논 블로킹이 중요한 이유는 IO 때문임.

- IO는 컴퓨터 작업 중에 가장 느림
- 멀티 스레드 + 블로킹 IO에서는 필연적으로 아래 문제들이 발생:
  - 단일 스레드 내에서의 context switching에 따른 자원 낭비 => 대부분의 시간 동안 사용을 안함. => 왜? => 멀티스레드 서로 블로킹 최소화를 위한 최적화 개발 복잡도 too high

논 블로킹 IO의 실질적인 작동 방식

- 기본 패턴: 폴링 (busy waiting)

```
resources = [socketA, socketB, fileA]
while (!resources.isEmpty()) {
  for (resource of resources) {
    // try to read
    data = resource.read()
    if (data === NO_DATA_AVAILABLE) {
      // there is no data to read at the moment
      continue
    }
    if (data === RESOURCE_CLOSED) {
      // the resource was closed, remove it from the list
      resources.remove(i)
    } else {
      //some data was received, process it
      consumeData(data)
    }
  }
}
```

저자가 말하는 이 코드의 문제점:

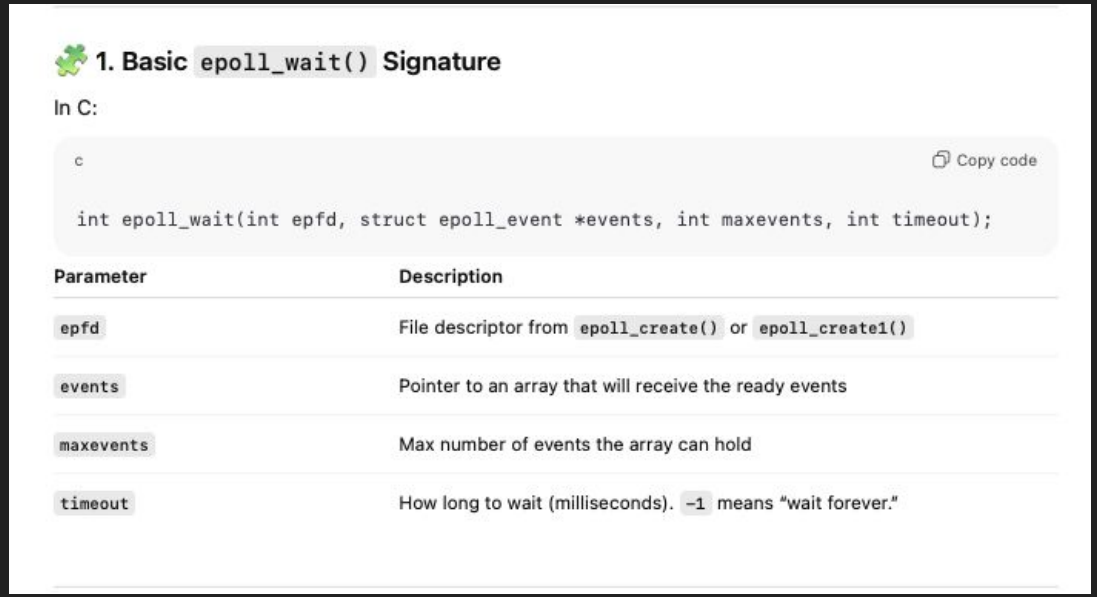
- while 로 date=resource.read() 계속하는 게 매우 리소스 낭비
- 보통 CPU 낭비가 매우 심함

두 번째 방법: 이벤트 멀티플렉싱

```
watchedList.add(socketA, FOR_READ)                          // (1)
watchedList.add(fileB, FOR_READ)
while (events = demultiplexer.watch(watchedList)) {         // (2)
  // event loop
  for (event of events) {                                   // (3)
    // This read will never block and will always return data
    data = event.resource.read()
    if (data === RESOURCE_CLOSED) {
      // the resource was closed, remove it from the watched list
      demultiplexer.unwatch(event.resource)
    } else {
      // some actual data was received, process it
      consumeData(data)
    }
  }
}
```

- 엥 이것도 옵저버 달린 while loop 아닌가? 뭔 차이지…
- never block..?이라는데 왜 never block 이신가요?

while (events = demultiplexer.watch(watchedList)) {



🧩 1. Basic `epoll_wait()` Signature

In C:

c — Copy code

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

Parameter	Description
`epfd`	File descriptor from `epoll_create()` or `epoll_create1()`
`events`	Pointer to an array that will receive the ready events
`maxevents`	Max number of events the array can hold
`timeout`	How long to wait (milliseconds). `-1` means "wait forever."

demultiplexer.watch

=== epoll_wait()

Linux OS 커맨드이며, 호출시 스레드를 잠재운다.

=> 그렇기에 while이 안걸린다는 의미임.

=> demultiplexer? => 멀티플렉싱된 이벤트 핸들링하는애.

전후 비교

두 가지 모두 비동기 IO 구현 방식.

- 1번 예제: polling으로 while loop 계속 기다림
- 2번 예제: 하드웨어에서 깨우기를 epoll_wait으로 기다림. 하드웨어가 나중에 깨워줌.

2번 예제 덕분에? 싱글 스레드가 가능함. => 스레드 idle time이 최소화됨. (+race

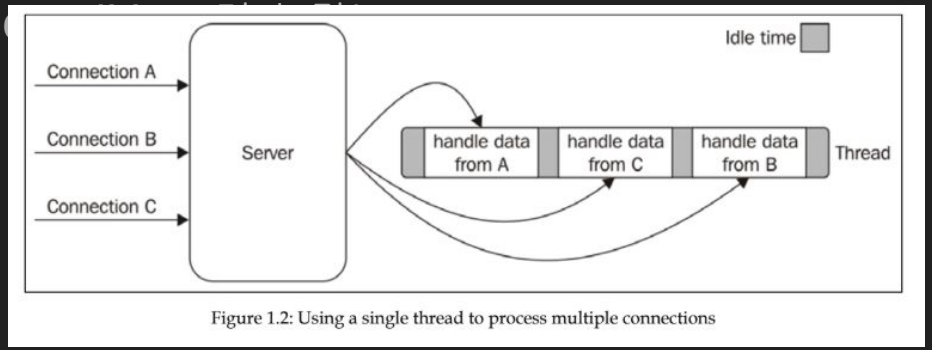


Figure 1.2: Using a single thread to process multiple connections

이제 헷갈리는 점

- 책이 남긴 인상:
  - 2000년대 어느날 - 삐빅 나는 천재 개발자 - 비동기 IO라는 개념이 있군 - 이걸 어떻게 잘 구현하는가 - multiplexing 쓰면 된다 삐빅 - 싱글 스레드로 구현하면 좋은 아키텍쳐이다 삐빅. EOD.
- 현실:
  - 아 멀티스레드 진짜 힘들다 => 걍 싱글 스레드로 하면 안됨? 비동기 IO로 구현해볼 수 있지 않나 => 엥 해봤더니 너무 잘되는데 => 이거 더 고도화시켜보자

⚙️ 2. "이벤트 루프"는 단순한 편법이자 천재적 해킹

이때 몇몇 OS 커널들은 `select()`, `poll()`, `epoll()`, `kqueue()` 같은 **event demultiplexer**를 제공하기 시작합니다.

즉, "스레드 1000개 만들지 말고,
하나의 스레드에서 여러 I/O를 감시할 수 있게 해줄게."

이건 원래 **네트워크 서버 최적화 해커들**이 효율을 위해 썼던 기법이에요 —
Node.js의 철학적 조상은 사실 **nginx**, **libevent**, **libuv** 같은 시스템들이에요.

> 즉, "싱글스레드 모델"은 '깨끗한 디자인 목표'라기보단
> "멀티스레딩이 너무 아프니까, 그걸 피해 가는 똑똑한 꼼수"였어요.

🧠 3. 근데 그 꼼수가 너무 잘 작동했다

Node.js가 등장했을 때(2009년), 대부분의 서버 언어는 이런 상태였어요:

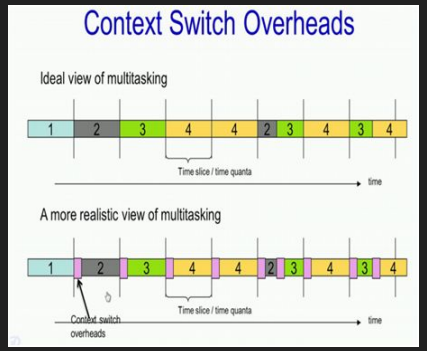
- Java: 멀티스레딩 + 복잡한 동기화
- Python/Ruby: GIL + 블로킹 I/O
- PHP: 요청당 프로세스

그 와중에 Node.js는 이렇게 말했죠:

> "그냥 싱글 스레드 하나만 돌리고,
> 커널이 I/O 이벤트를 주면 콜백만 실행하면 되잖아?"

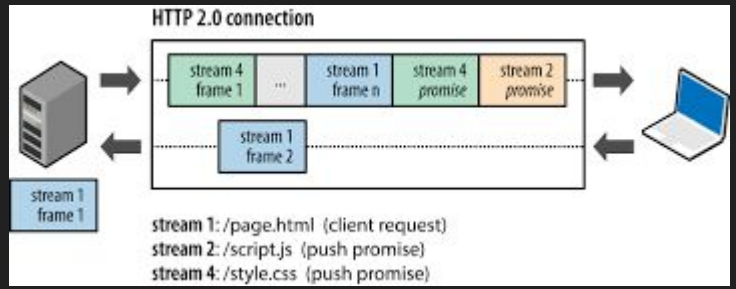
결과:

+상식: 멀티플렉싱이란 뭘까?



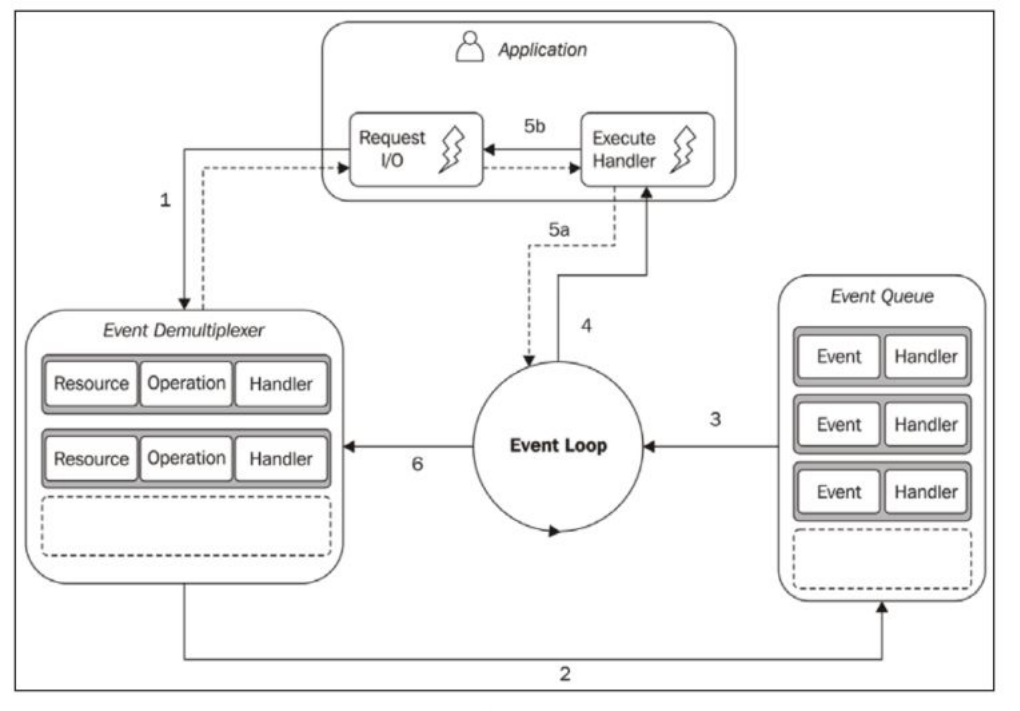
- 우린 이미 숱하게 멀티 플렉싱을 봐왔습니다…

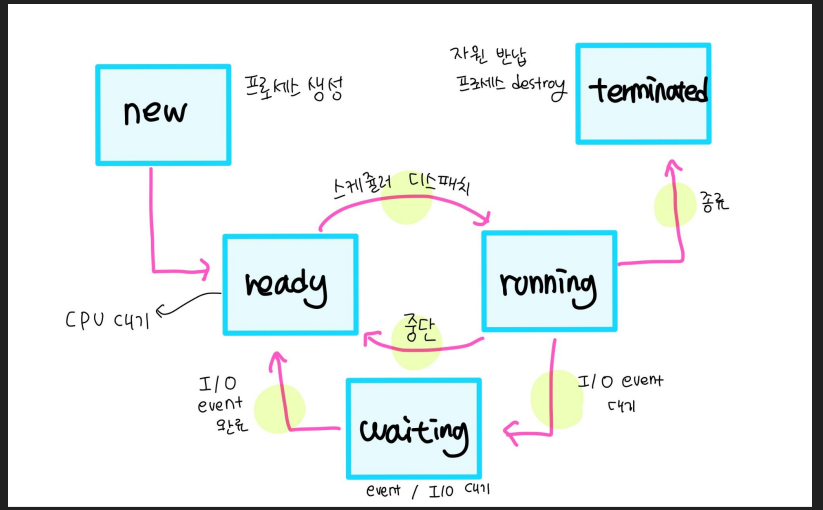
1. CPU 물리 스레드 스케줄링
2. http2 멀티플렉싱
3. React Fiber의 스케줄링
4. Node 싱글 스레드 시스템



- 모두 time division multiplexing.
- code, wave-length, bandwidth… 는 SW에서 잘 안보임.

The Reactor Pattern





cpu 프로세스랑 똑같이 생겼네

The Reactor Pattern

🧠 3. Why It's Called "Reactor"

Because the program **reacts** to events rather than **polling** or **blocking**.
It's literally the abstraction of *reacting to demultiplexed events* from an event source (the OS).

So the name "Reactor" was chosen to contrast with procedural "proactor" or "polling" loops.

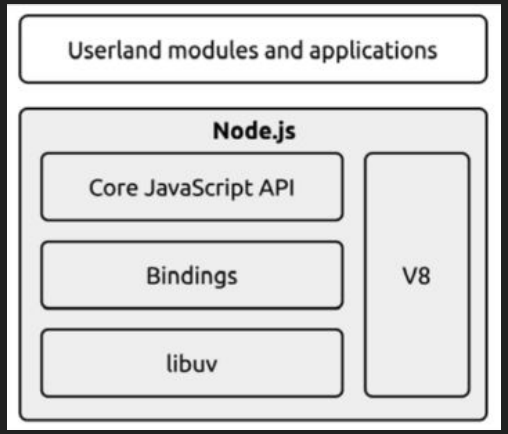
📘 4. Later Influence

Doug Schmidt's Reactor pattern became a cornerstone for almost every modern event-driven system:

System / Framework	Core Mechanism	Rooted In Reactor Pattern
ACE (Adaptive Communication Environment)	C++ framework by Schmidt himself	Direct implementation
Java NIO (Selector)	Java's non-blocking I/O system	Inspired by Reactor
libevent / libev / libuv	C event libraries used by Node.js, nginx, etc.	Reactor core
nginx	Event-driven architecture using epoll/kqueue	Reactor model
Node.js	Single-threaded event loop built on libuv	Reactor pattern
Twisted (Python)	Event-driven networking engine	Reactor explicitly named

- 창시자는 Douglas C. Schmidt
- 교수임.

Node.js Recipe - libuv



2. 엥 브라우저 V8 쓰는 애들은 libuv 없이도 싱글 스레드 자바스크립트 잘 돌아가는데..?
=> 그건 브라우저에 libuv와 유사하지만 더 큰 OS 추상화 계층이 있기 때문임.

1. libuv란?
- event demultiplexing이 각 OS 마다 다 조금씩 달라서, 이걸 예측 가능하도록 하나의 인터페이스로 통일해주는 라이브러리.
- 또한 the reactor pattern 도 구현되어있음.

3. 그니까.. v8은 자바스크립트만 돌릴 수 있는 애고, 얘를 브라우저에서 빼내어서 서버에서 쓰려다보니 가장 크게 필요했던 구현체가 libuv.

4. 챕터 1에서 그렇게 디멀티플렉싱을 강조했던 이유: libuv 구현체가 node의 핵심이기 때문임.

esm vs commonjs

탄생 이유: 애초에 브라우저에서는 <script> 쓰다가 서버에 넘어오니 모듈 시스템이 갑분싸됨.

그래서 node 진영에서 cjs 만들어서 쓰다가 ecmascript에서 구현 방향성 나와서 몇 년 동안 브라우저 회사들이 열심히 esm 만듦.

- require:
  - home made script로 쉽게 재현 가능
  - 파일 어디서든 require 가능
  - 순환 참조 취약함. 사전에 예방 어려움.
- import:
  - 항상 파일 위쪽에 import 정의구조 있어야함.
  - import 위계 구조를 정리하기에 tree shaking 가능
  - 순환 참조에 강함. 맨 처음에 먼저 호출 구조 정리를 하기에 사전에 에러 던짐.

bonus: esm ⇔ commonjs 서로 참조 방법

✅ 1. CommonJS → ESM 모듈 불러오기

CommonJS(`require`)에서 ESM(`import/export`) 모듈을 불러올 때는 **비동기 import()**를 사용해야 합니다.

js Copy code

```js
// commonjs.js
(async () => {
  const { hello } = await import('./esm.mjs');
  hello();
})();
```

✅ 2. ESM → CommonJS 모듈 불러오기

ESM(`.mjs` 또는 `"type": "module"`)에서 CommonJS 모듈을 불러올 때는 **`createRequire()`**를 사용합니다.

js Copy code

```js
// esm.mjs
import { createRequire } from 'module';
const require = createRequire(import.meta.url);

const cjs = require('./commonjs.cjs');
cjs.hello();
```

⚙️ ESM ↔ CommonJS 상호 참조 요약

방향	가능 여부	이유
CommonJS → ESM	⚠️ *부분적으로 가능 (비동기 import)*	CommonJS는 동기 로딩 기반이라, ESM을 불러올 때 `await import()`로만 가능. 따라서 **초기 로드 시점에는 참조 불가**.
ESM → CommonJS	✅ *완전히 가능*	`createRequire()`를 통해 동기적으로 CommonJS 모듈을 로드 가능.
양방향 순환 참조	❌ *불안정 / 대부분 실패*	로더 단계 차이로 인해, 한쪽이 초기화되지 않은 상태(undefined export)로 참조됨.