EVENT LOOP

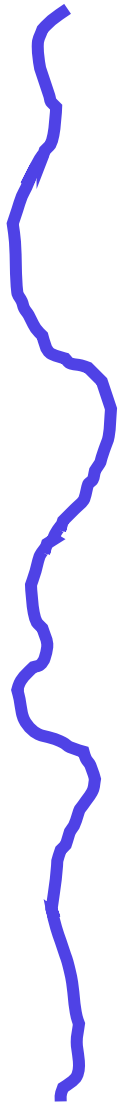
Node.js 의 핵심 이벤트 루프를 알아보자

발표자 : 다람쥐

발표날짜 : 2020년 12월 12일 토요일

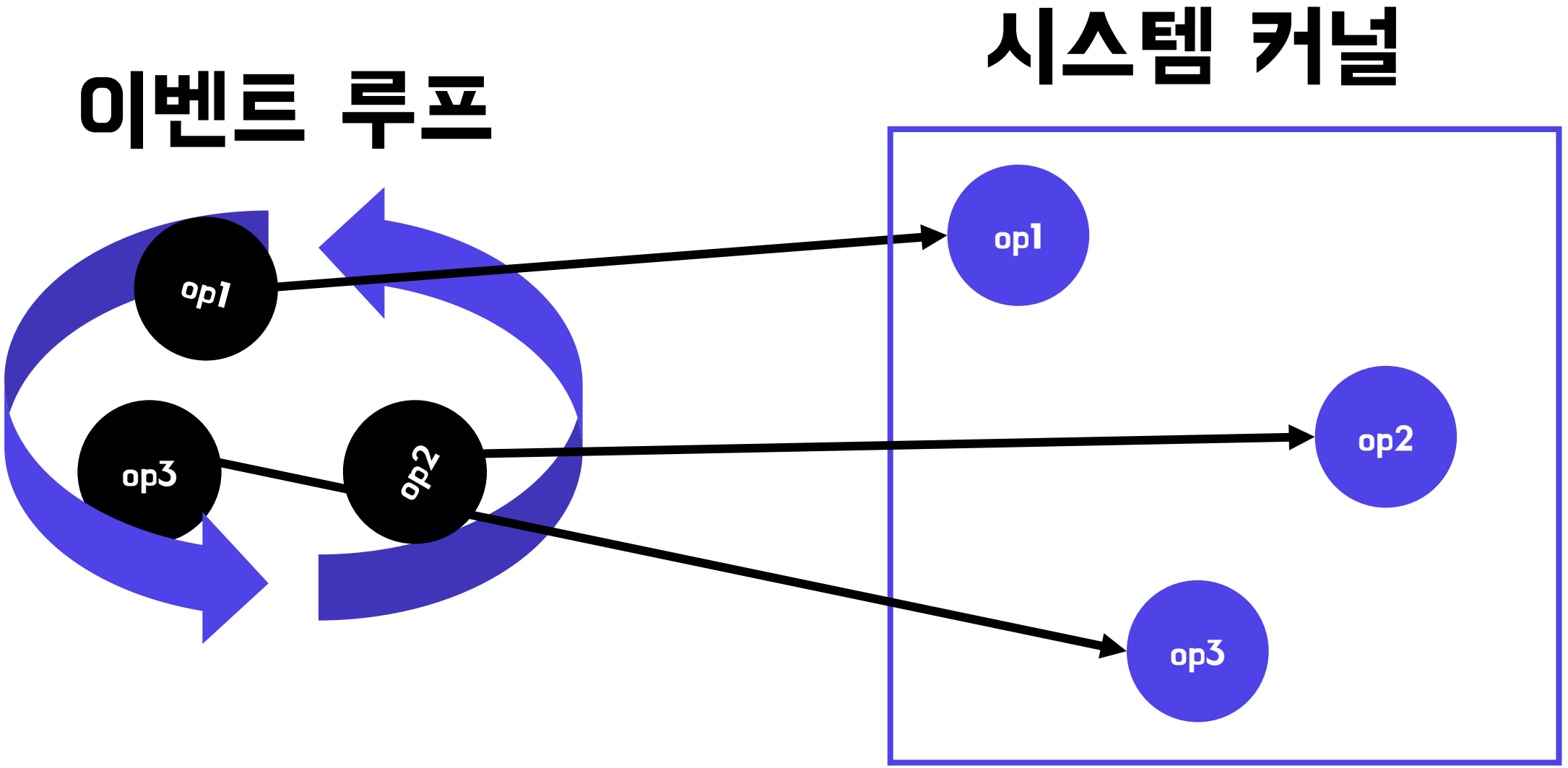
싱글 스레드

Node.js 동작

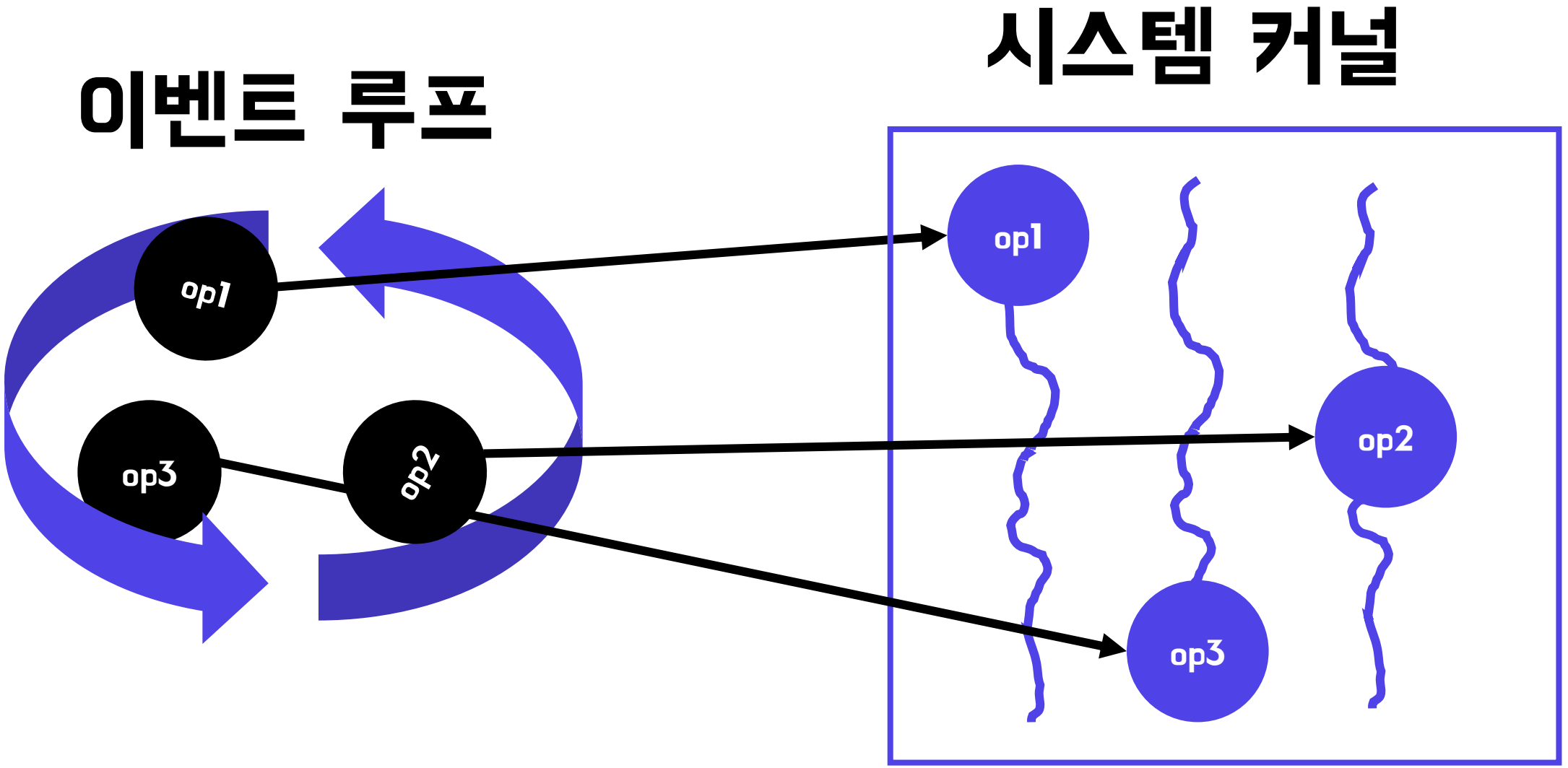


Node.js 동작

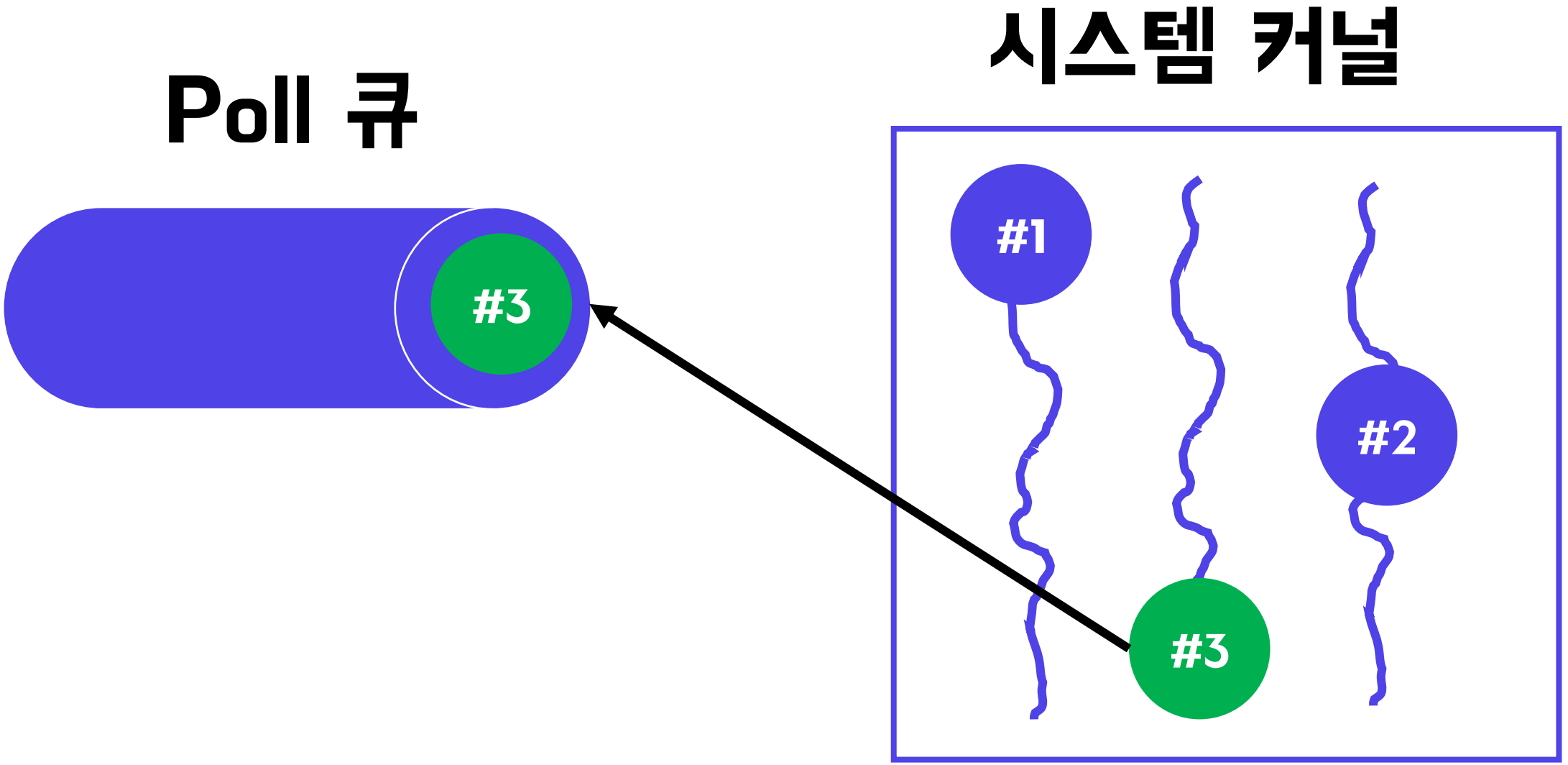
이벤트 루프란



이벤트 루프란



이벤트 루프란



이벤트 루프 실행 순서

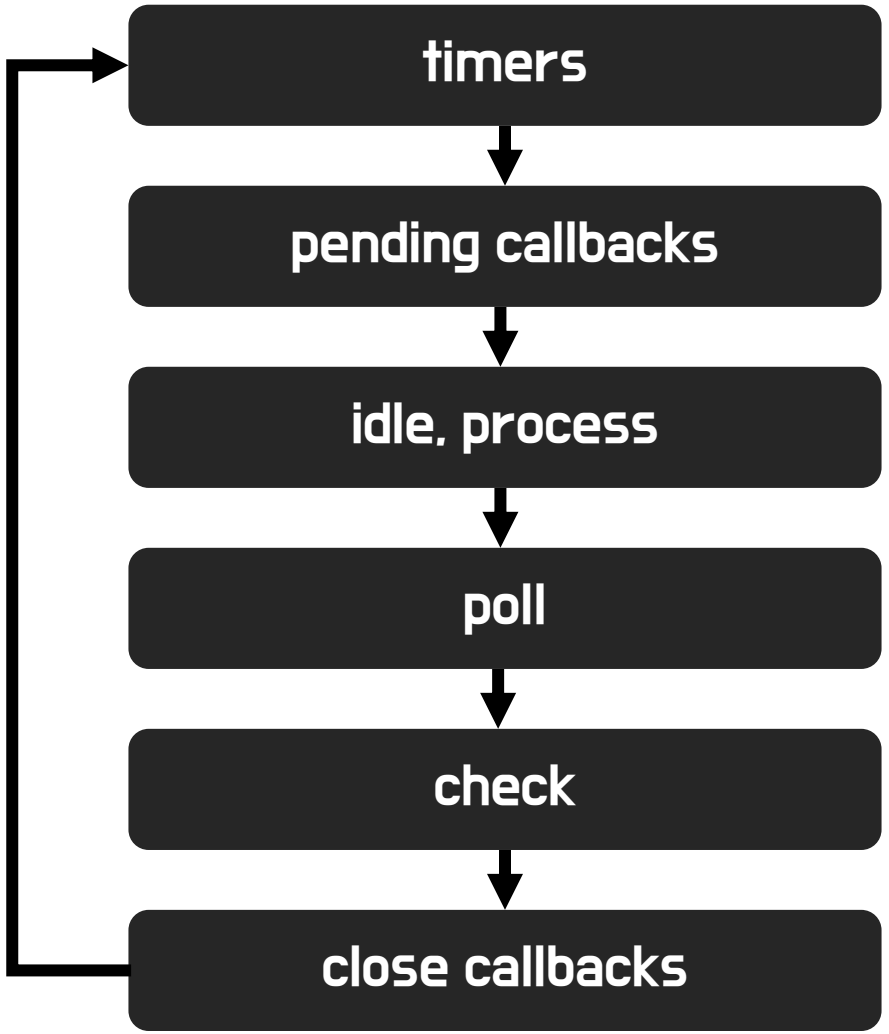
Node.js 실행 시작 시나리오

1. 이벤트 루프 초기화
2. 비동기 API 호출, 타이머 스케쥴, process.nextTick() 호출하는 스크립트 실행
3. 이벤트 루프 처리 수행

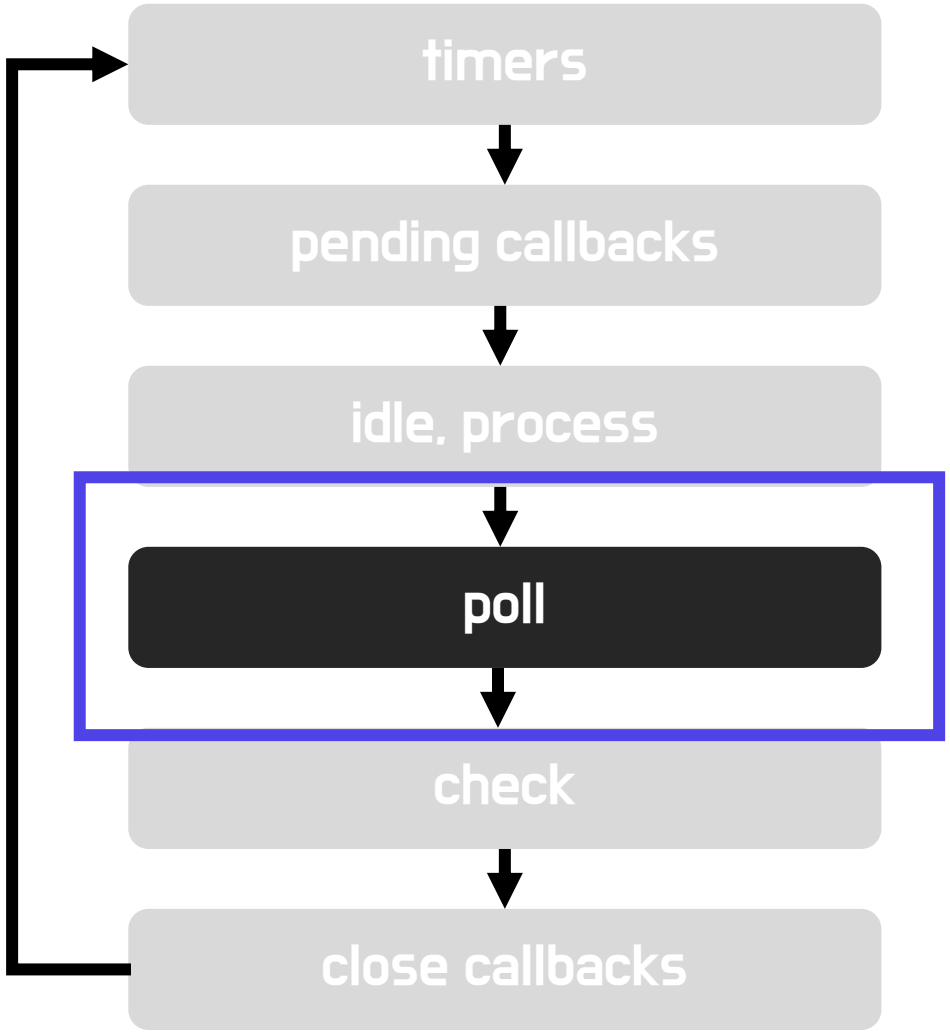
이벤트 루프 실행 순서

각 단계는 phase 로 구분

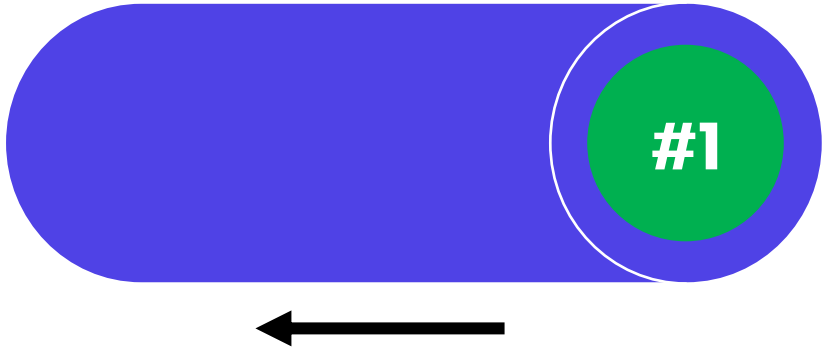
이벤트 루프 실행 순서


timers
pending callbacks
idle, process
poll
check
close callbacks

이벤트 루프 실행 순서

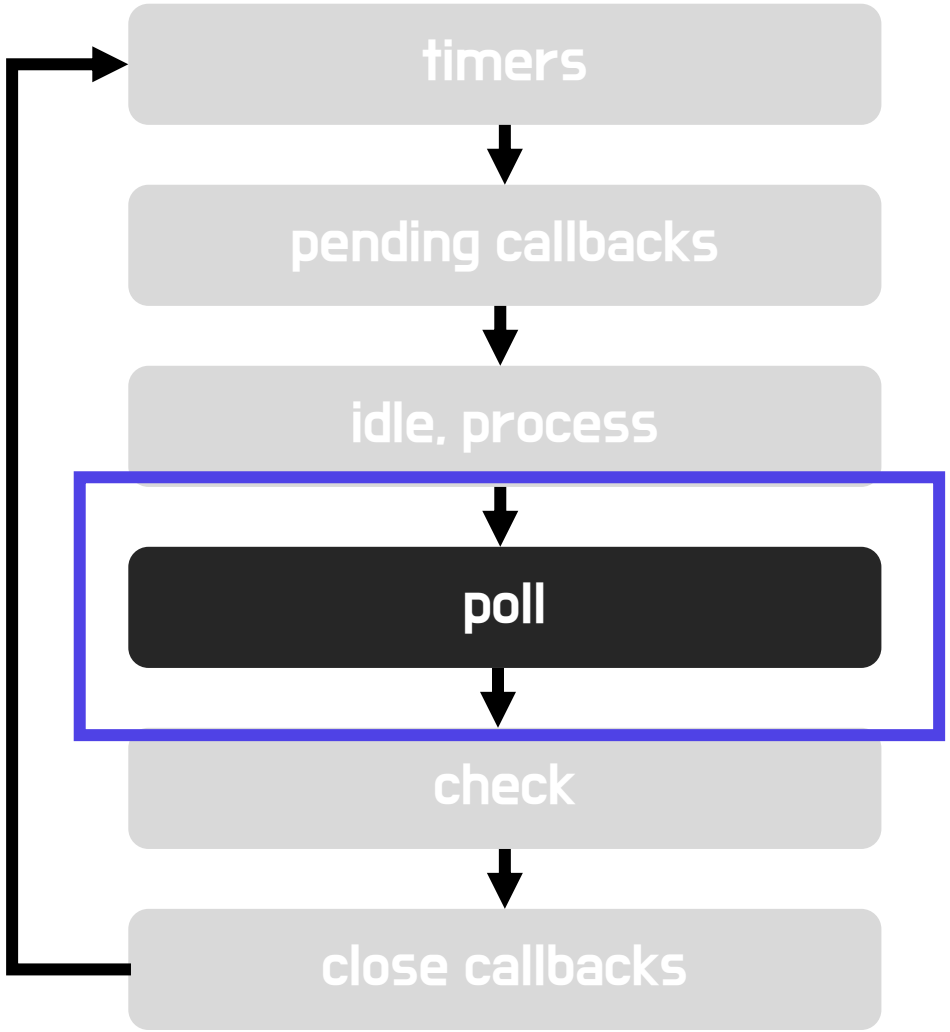


FIFO 큐

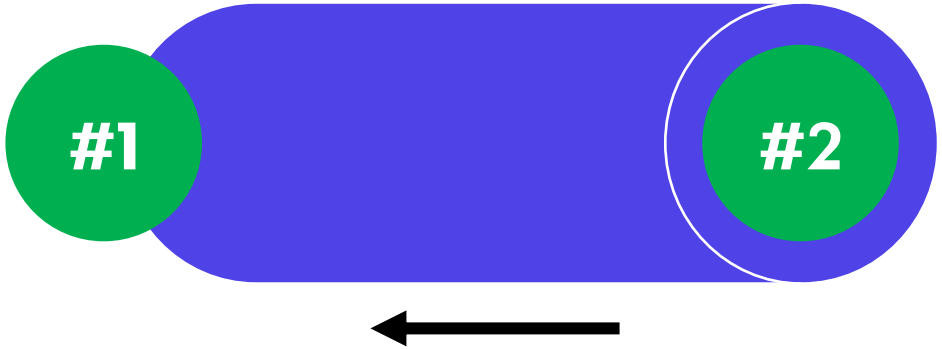


First In
First Out

이벤트 루프 실행 순서

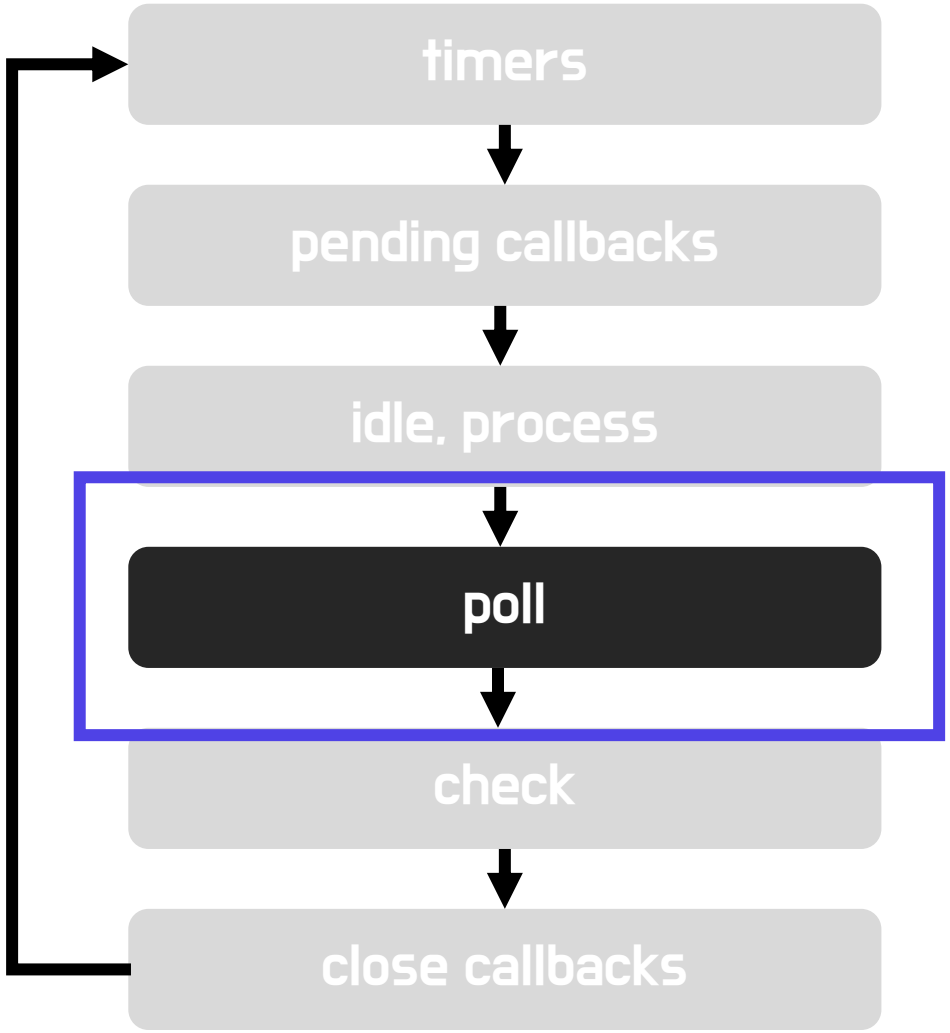


FIFO 큐

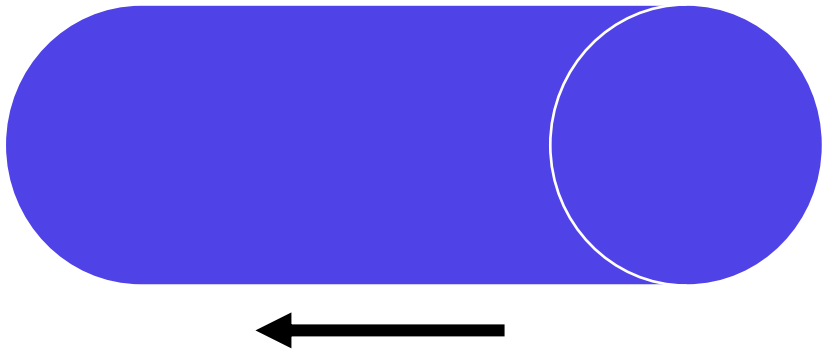


First In
First Out

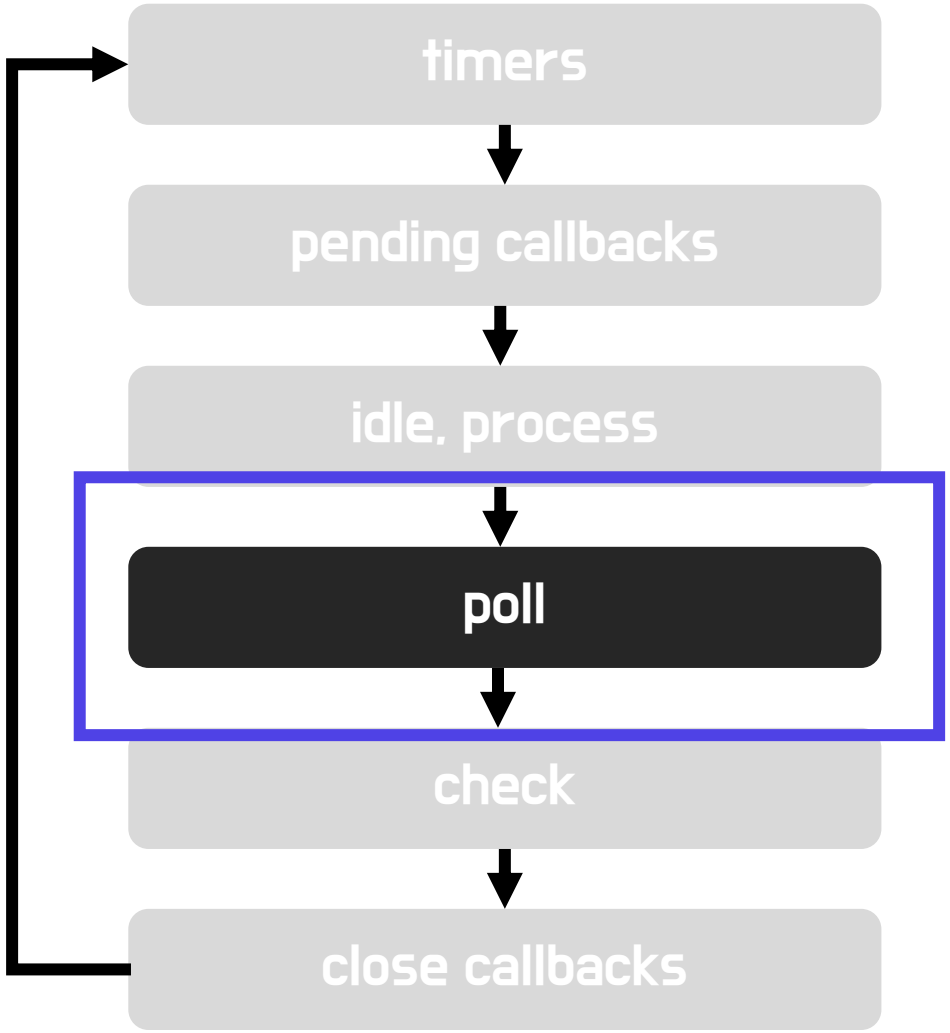
이벤트 루프 실행 순서



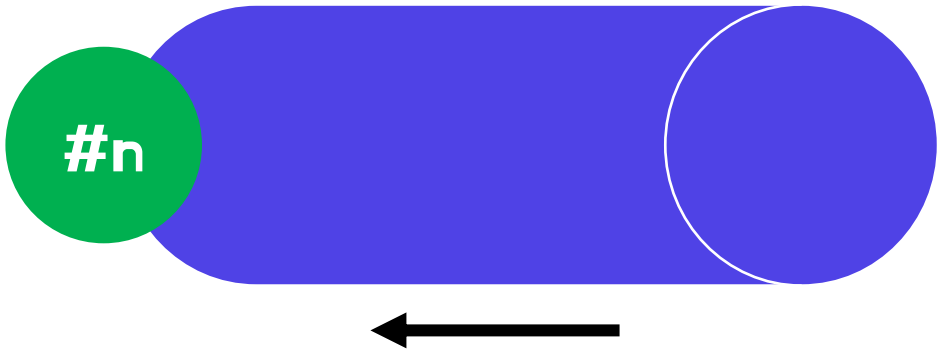
FIFO 큐



이벤트 루프 실행 순서

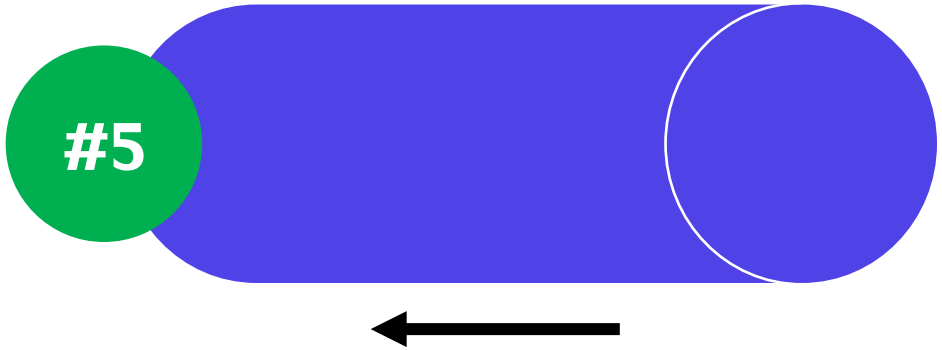


FIFO 큐

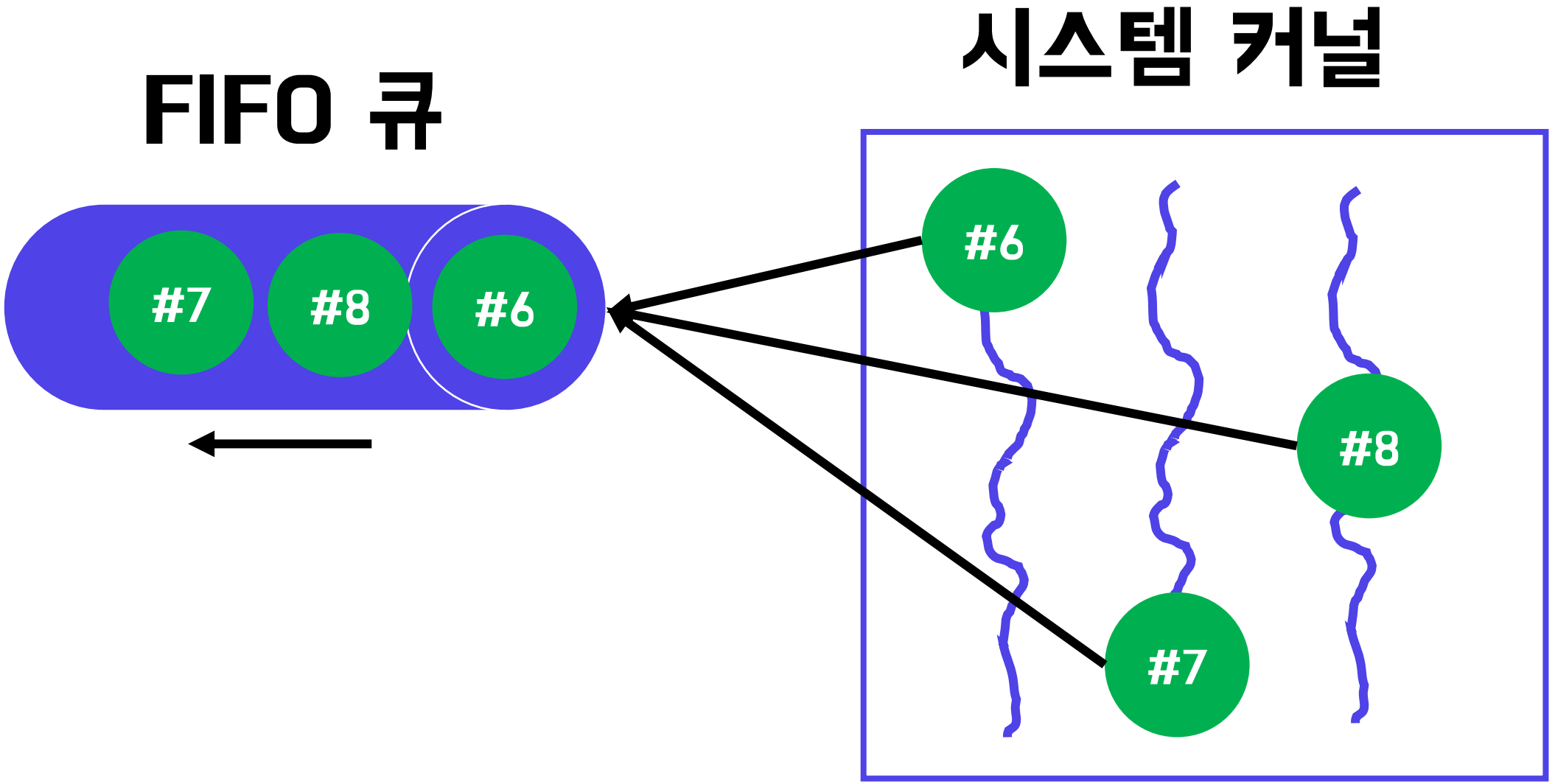


이벤트 루프 실행 순서

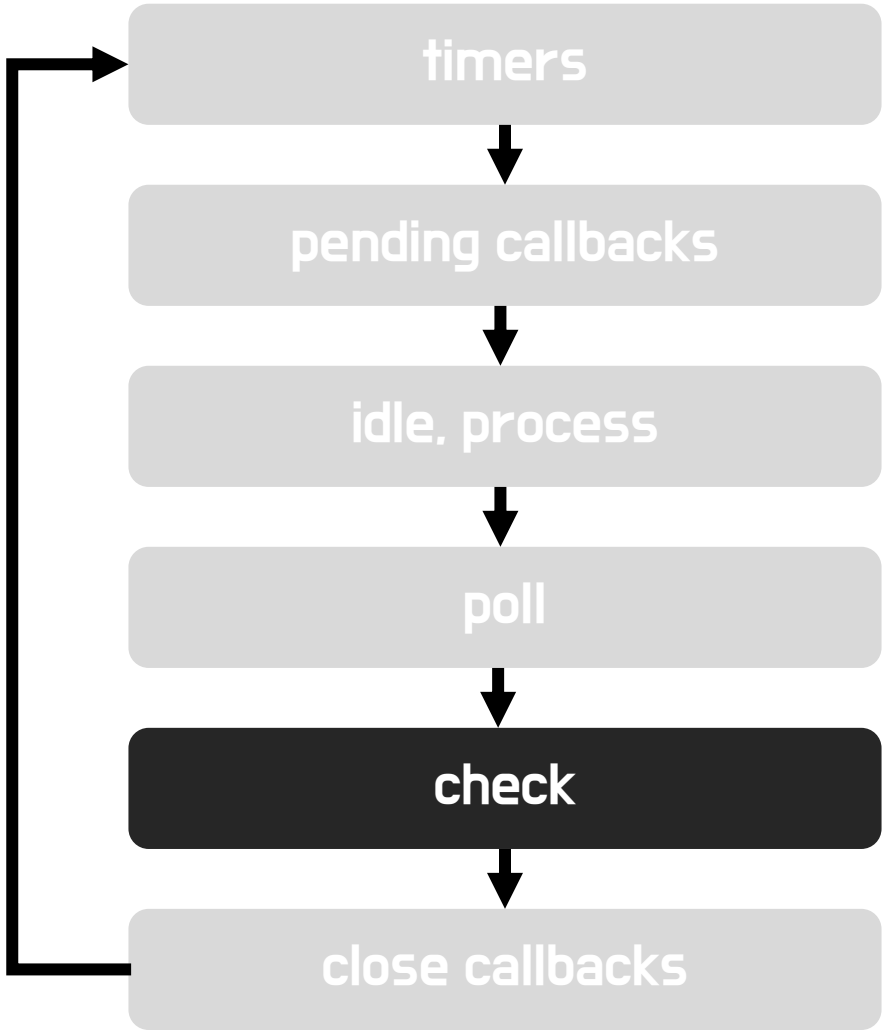
“해치웠나..?”



이벤트 루프 실행 순서

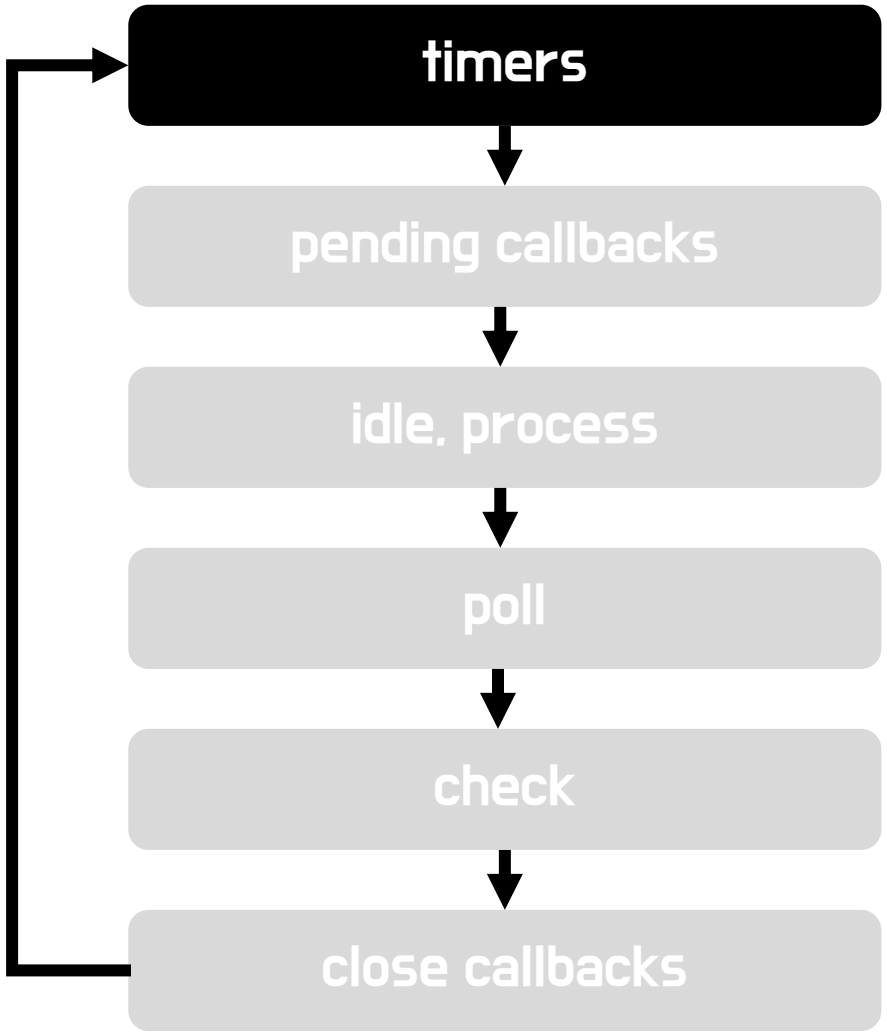


이벤트 루프 실행 순서


timers
pending callbacks
idle, process
poll
check
close callbacks

페이즈 간략 소개

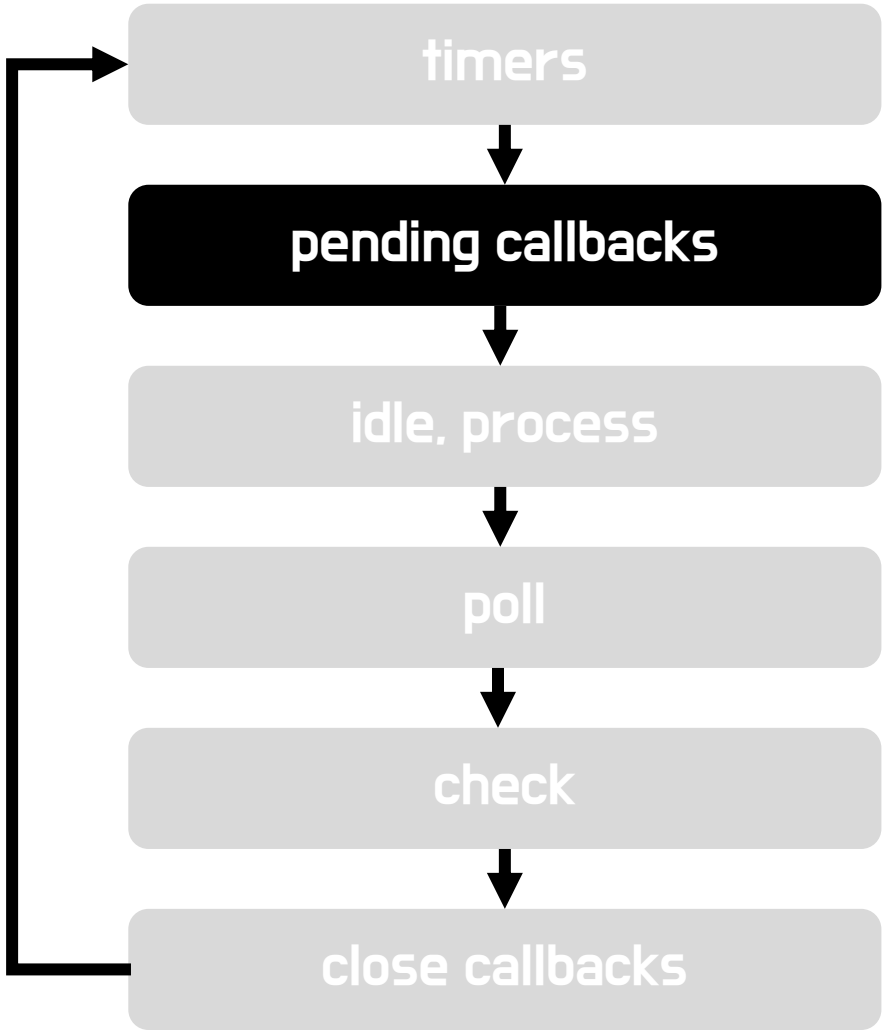
페이즈 간략 소개



setTimeout()

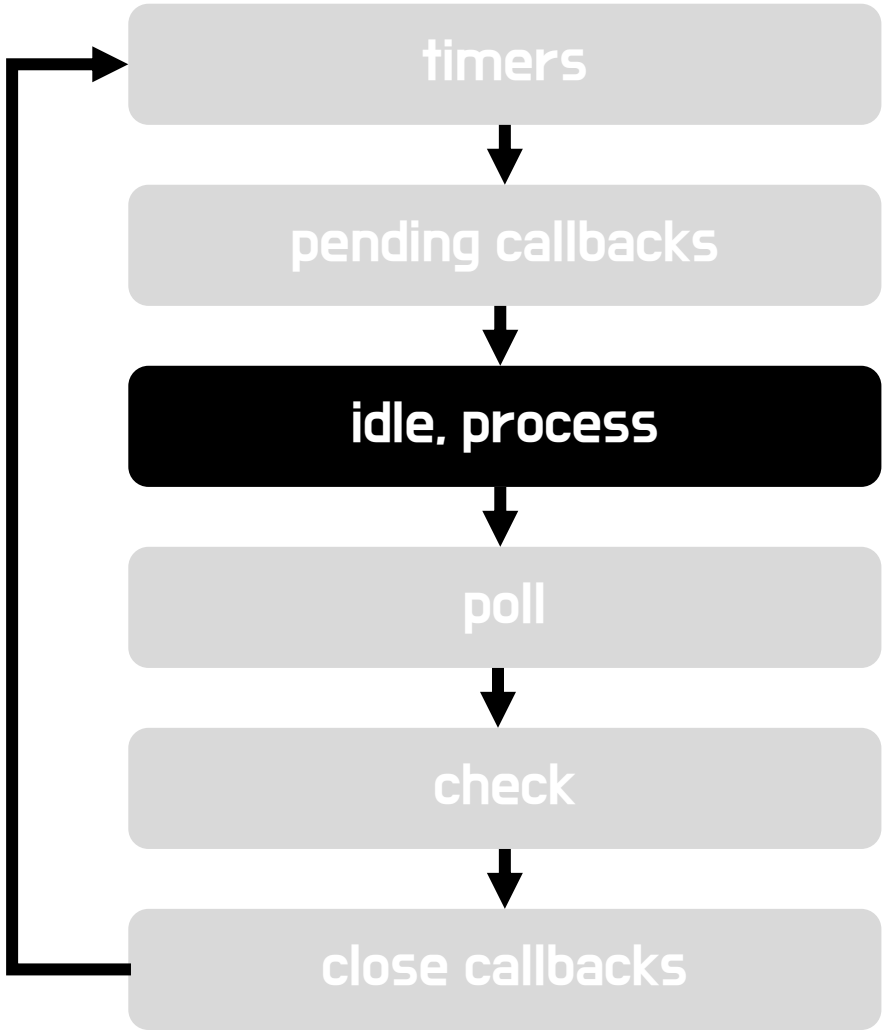
setInterval()

페이즈 간략 소개



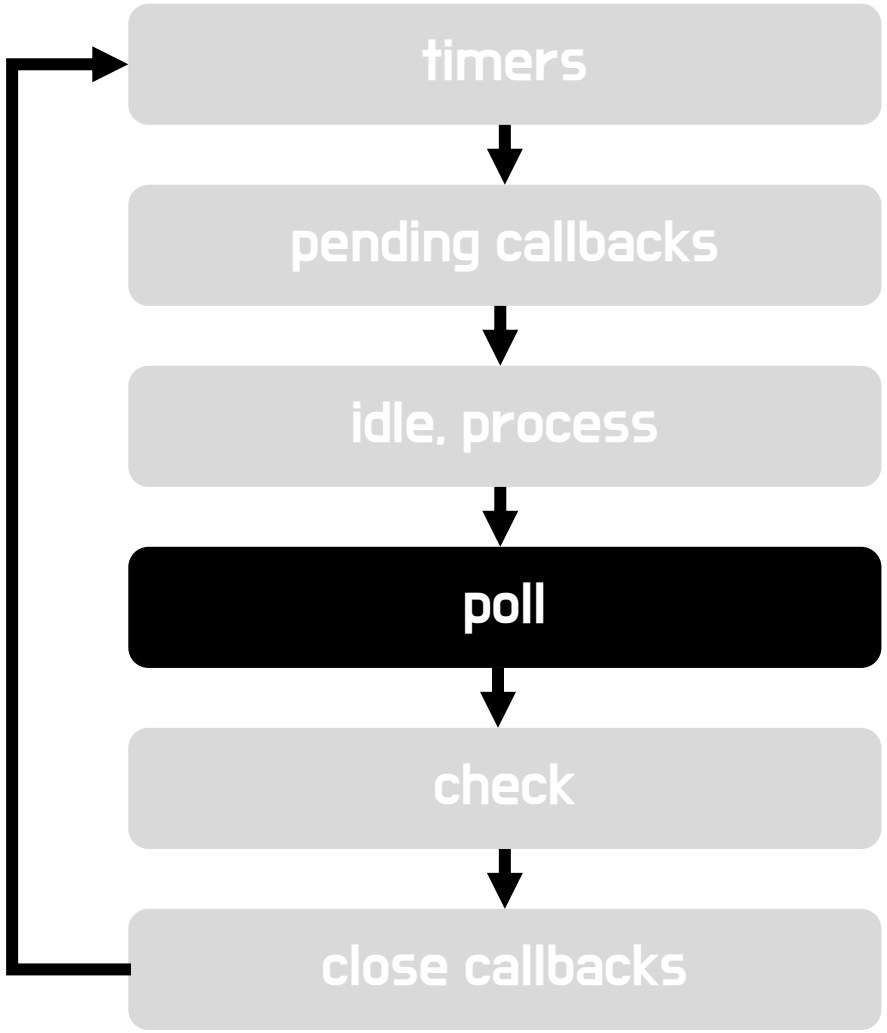
다음 루프까지 연기된 I/O 콜백

페이즈 간략 소개



내부에서만 사용

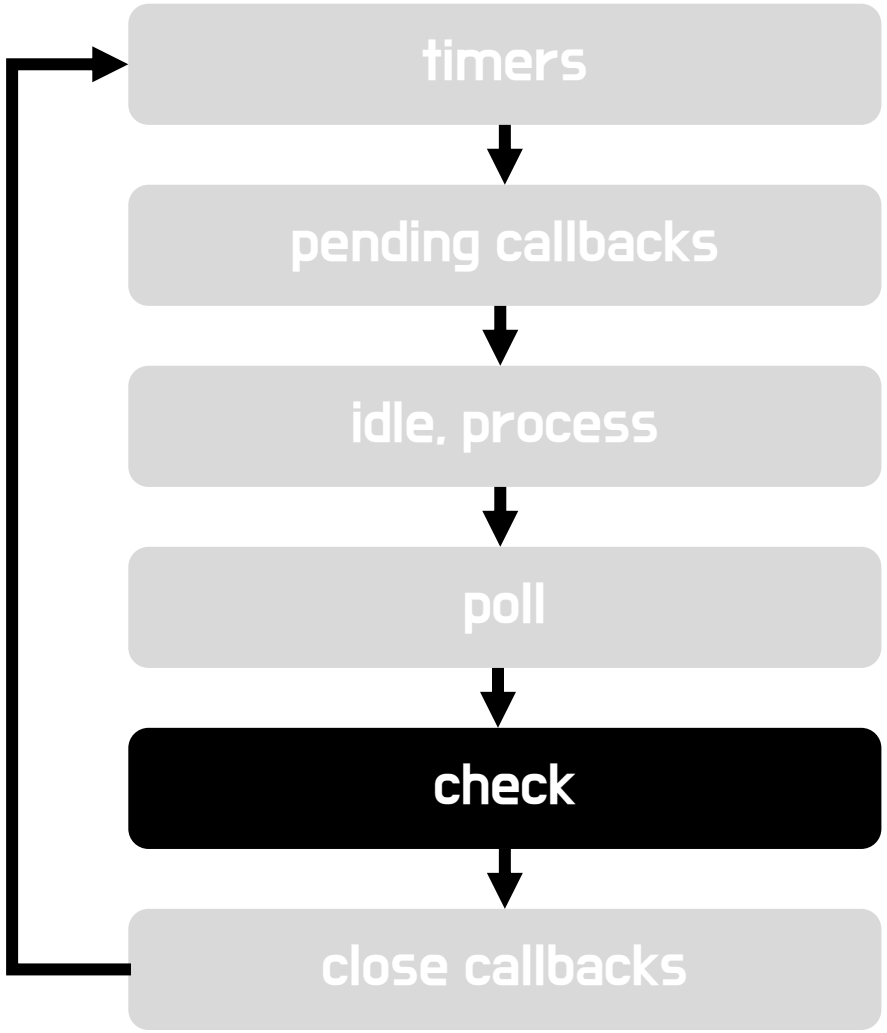
페이즈 간략 소개



I/O 이벤트 탐색
I/O 콜백 실행

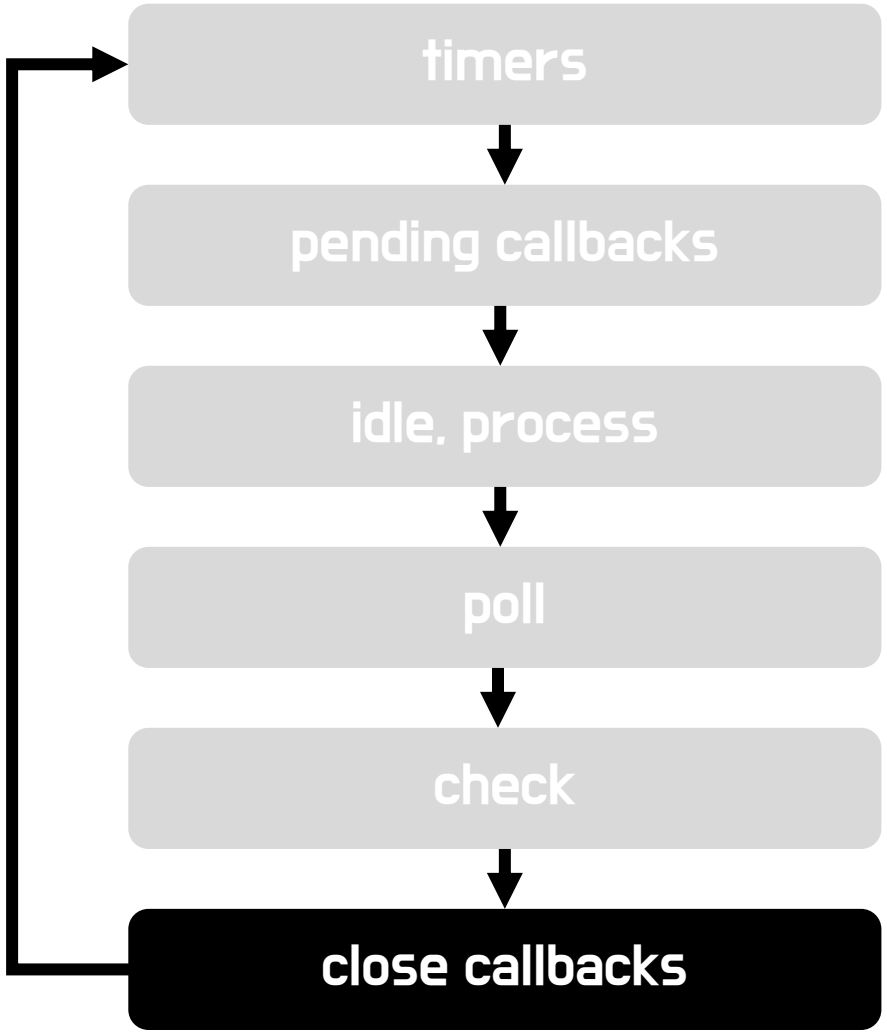
close callbacks 예외 콜백,
타이머 스케쥴링 콜백,
setImmediate() 콜백, ...

페이즈 간략 소개



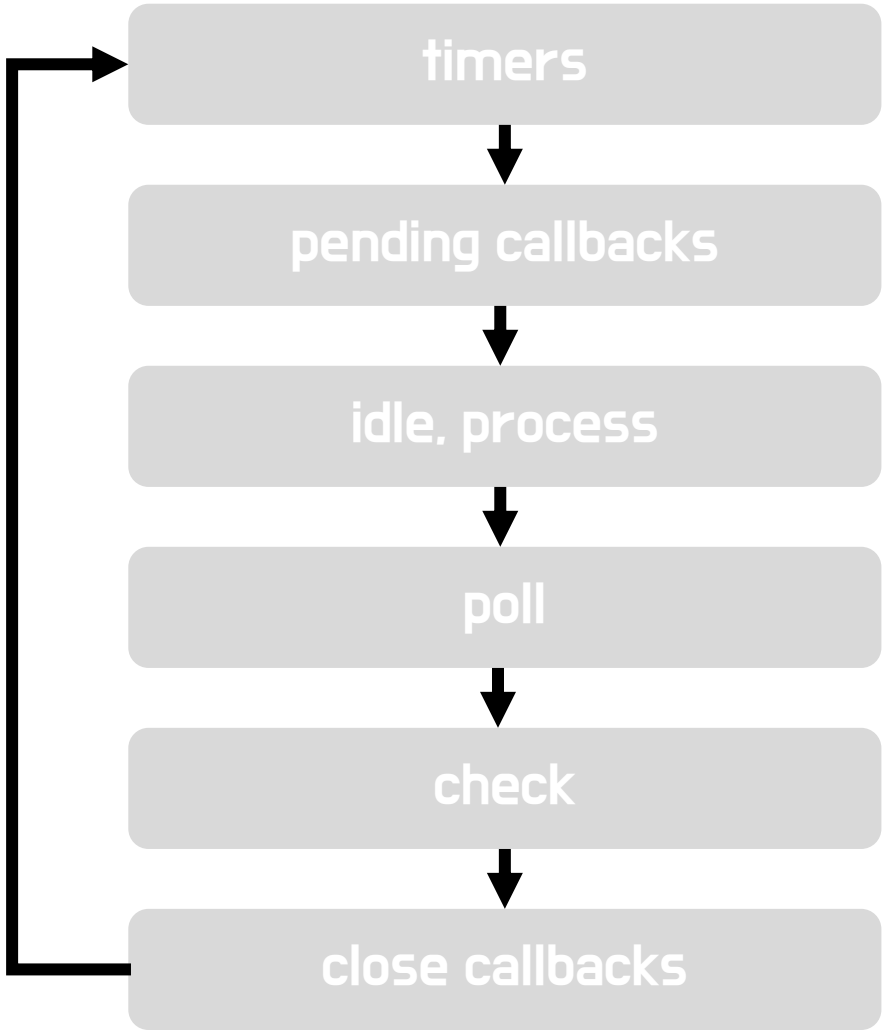
setImmediate()

페이즈 간략 소개



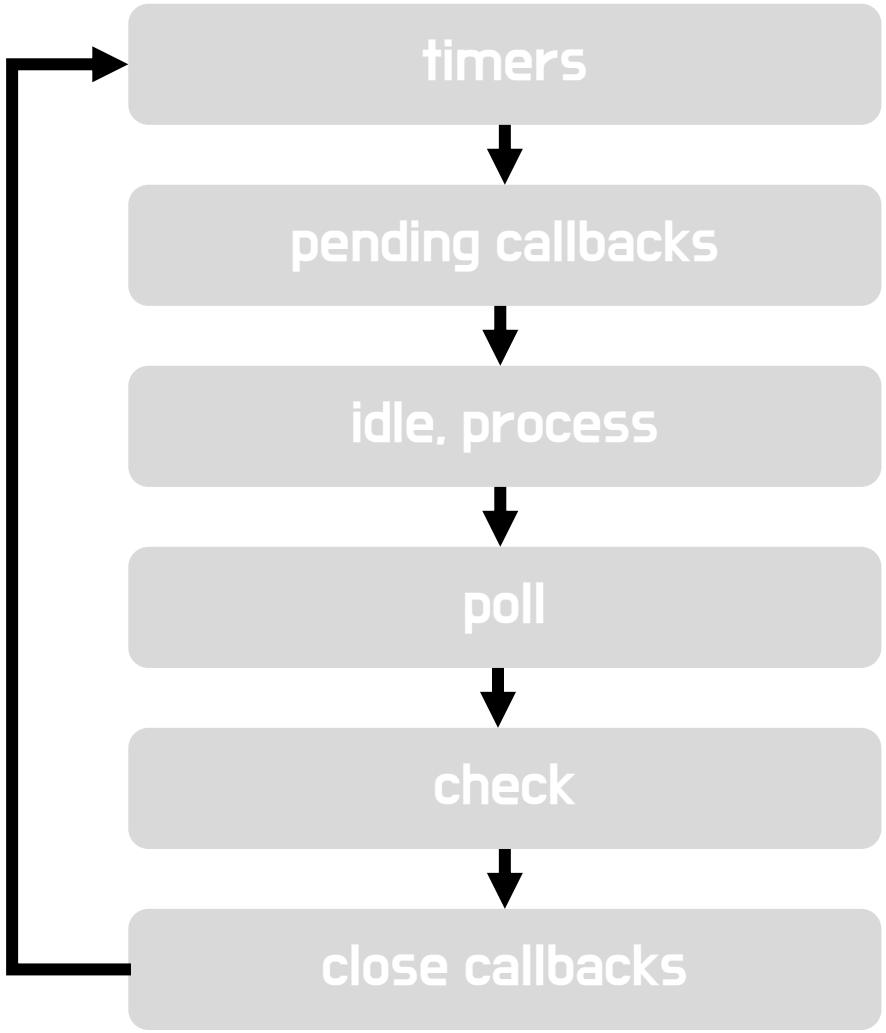

```
socket.on('close', ...);
```

페이즈 간략 소개



비동기 I/O 또는 타이머 대기 중인지 확인

페이즈 간략 소개



없다면 깨끗이 정지

페이즈 상세

그래서 정확히
어떻게 동작하는데?

페이즈 상세 - timers

“ 100ms 타이머를 기다리면서 95ms 걸리는 비동기 파일 읽기 콜백함수를 실행할거야 ”

페이즈 상세 - timers

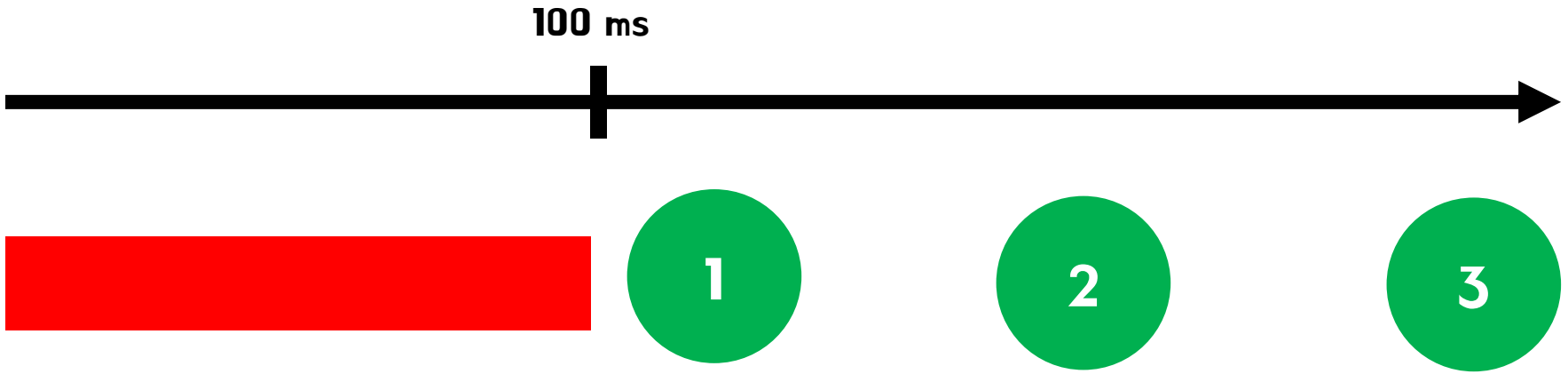
100 ms 동안은
절대 실행되지 않을거야.
그 이후에 되면 실행해줄게

페이즈 상세 - timers

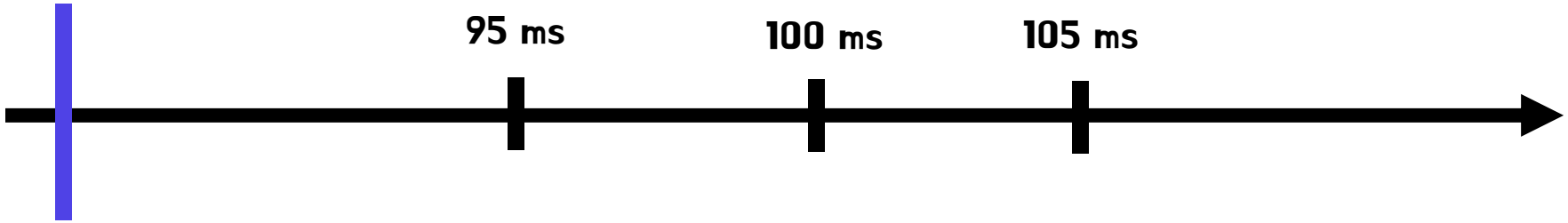
정확한 시간 : X

기준 시간 : O

페이즈 상세 - timers



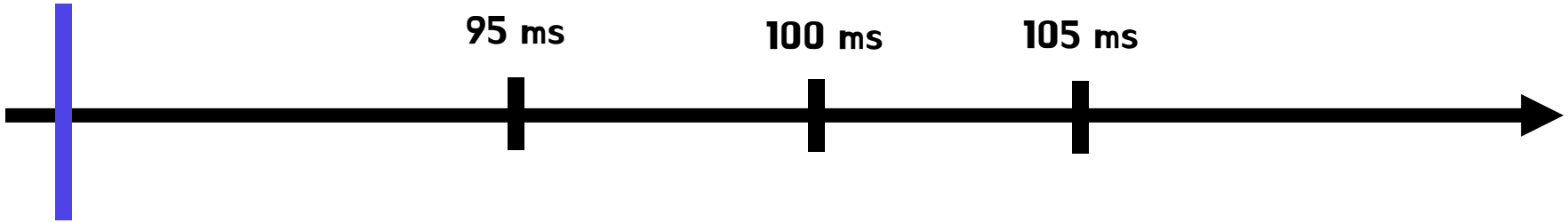
페이즈 상세 - timers



현재 poll 단계

1. 빈 큐를 가짐 (95ms 걸리는 비동기 파일 읽기가 아직 완료되지 않음)
2. 가장 빠른 타이머 임계 값에 도달할 때 까지 수 밀리 초를 기다림
3. 그러던 중 파일 읽기를 마치고 10 ms 가 걸리는 콜백이 poll 큐에 추가되어 실행
4. 콜백이 완료되었을 때, 가장 빠른 타이머의 임계 값을 확인
5. 타이머의 콜백을 실행하려고 timers 단계로 돌아감
6. 전체 지연 시간 : 105 ms

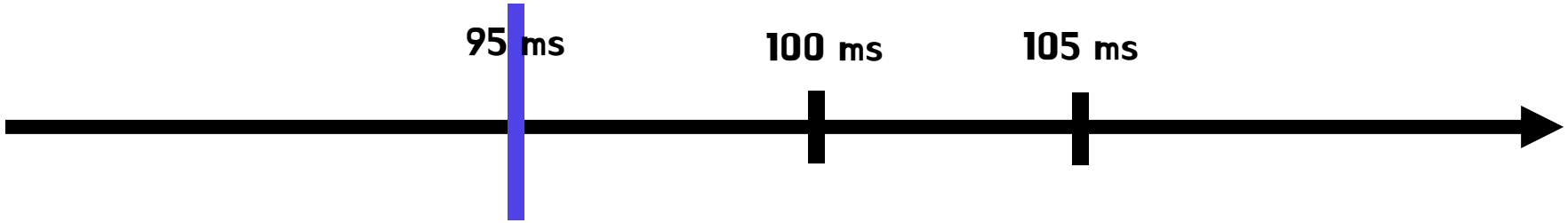
페이즈 상세 - timers



현재 poll 단계

1. 빈 큐를 가짐 (95ms 걸리는 비동기 파일 읽기가 아직 완료되지 않음)
2. 가장 빠른 타이머 임계 값에 도달할 때 까지 수 밀리 초를 기다림
3. 그러던 중 파일 읽기를 마치고 10 ms 가 걸리는 콜백이 poll 큐에 추가되어 실행
4. 콜백이 완료되었을 때, 가장 빠른 타이머의 임계 값을 확인
5. 타이머의 콜백을 실행하려고 timers 단계로 돌아감
6. 전체 지연 시간 : 105 ms

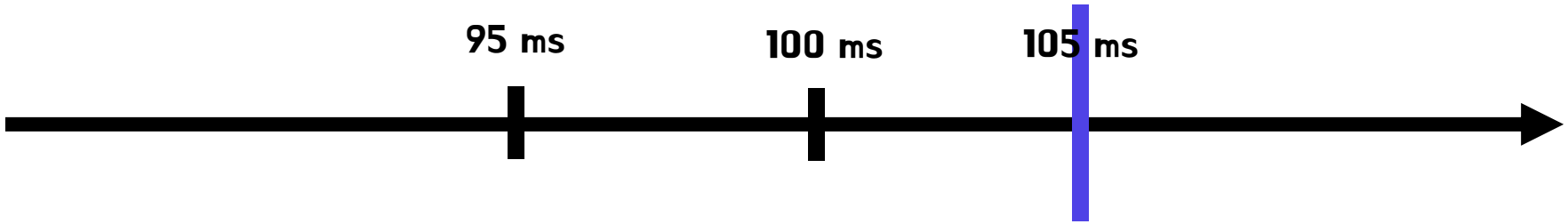
페이즈 상세 - timers



현재 poll 단계

1. 빈 큐를 가짐 (95ms 걸리는 비동기 파일 읽기가 아직 완료되지 않음)
2. 가장 빠른 타이머 임계 값에 도달할 때 까지 수 밀리 초를 기다림
3. 그러던 중 파일 읽기를 마치고 10 ms 가 걸리는 콜백이 poll 큐에 추가되어 실행
4. 콜백이 완료되었을 때, 가장 빠른 타이머의 임계 값을 확인
5. 타이머의 콜백을 실행하려고 timers 단계로 돌아감
6. 전체 지연 시간 : 105 ms

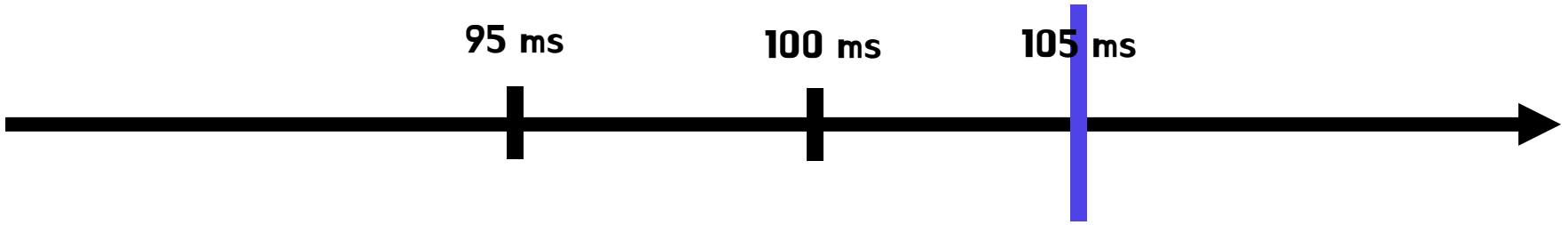
페이즈 상세 - timers



현재 poll 단계

1. 빈 큐를 가짐 (95ms 걸리는 비동기 파일 읽기가 아직 완료되지 않음)
2. 가장 빠른 타이머 임계 값에 도달할 때 까지 수 밀리 초를 기다림
3. 그러던 중 파일 읽기를 마치고 10 ms 가 걸리는 콜백이 poll 큐에 추가되어 실행
4. 콜백이 완료되었을 때, 가장 빠른 타이머의 임계 값을 확인
5. 타이머의 콜백을 실행하려고 timers 단계로 돌아감
6. 전체 지연 시간 : 105 ms

페이즈 상세 - timers



현재 poll 단계

1. 빈 큐를 가짐 (95ms 걸리는 비동기 파일 읽기가 아직 완료되지 않음)
2. 가장 빠른 타이머 임계 값에 도달할 때 까지 수 밀리 초를 기다림
3. 그러던 중 파일 읽기를 마치고 10 ms 가 걸리는 콜백이 poll 큐에 추가되어 실행
4. 콜백이 완료되었을 때, 가장 빠른 타이머의 임계 값을 확인
5. 타이머의 콜백을 실행하려고 timers 단계로 돌아감
6. 전체 지연 시간 : 105 ms

페이즈 상세 - pending callbacks

TCP 오류 같은 시스템 작업 콜백 실행

예를 들어 TCP 소켓이 연결 시도하다 ECONNREFUSED 를 받으면 일부 *NIX 시스템을 오류 보고를 대기한다.
pending callbacks 페이즈에 실행하기 위해 큐에 추가된다.

poll 페이즈의 주요 기능

1. I/O 를 얼마나 오래 블록킹하고 폴링해야 하는지 계산
2. poll 큐에 있는 이벤트 처리

스케줄링된 타이머가 없을 때

1. poll 큐가 비어있지 않다면
2. poll 큐가 비어있다면

1. poll 큐가 비어있지 않다면

poll 큐에 있는 콜백을 다 소진하거나
한계에 도달할 때 까지 동기로 콜백 실행

2. poll 큐가 비어있다면

1. setImmediate() 라면 poll 단계를 종료하고
스케줄링된 스크립트를 실행하기 위해 check 단계로 넘어감

2. poll 큐가 비어있다면

2. 그 외에 콜백이 큐에 추가되기를 기다린 후 즉시 실행

시간 임계점에 도달했는지 먼저 확인
하나 이상의 타이머가 준비됐다면 타이머 콜백을 실행하기 위해 timers 단계로 돌아감

페이즈 상세 - check

poll 단계 완료 직후 콜백 실행

poll 단계에서 큐가 비어있고 setImmediate()로 큐에
추가 되면 이벤트 루프를 기다리지 않고 check 단계로 이동

페이즈 상세 - check

setImmediate()

이벤트 루프의 별도 단계에서 실행되는 특수한 타이머
poll 단계 완료하고 콜백 실행을 스케줄링 하는데
Libuv API 를 사용

close 콜백

소켓이나 핸들이 닫힌 경우, ‘close’ 이벤트가 이 단계에서 실행됩니다.
그 외에는 process.nextTick() 으로 실행된다.

setImmediate()

v.s.

setTimeout()

setImmediate() v.s. setTimeout()

호출 시기에 따라 다르게 동작

- setImmediate() : poll 단계가 완료되면 스크립트 실행
- setTimeout() : 최소 임계 값(ms)이 지난 후 스크립트 실행

setImmediate() v.s. setTimeout()

호출 시기에 따라 다르게 동작

타이머 실행 순서는 어떤 컨텍스트에서 호출되었는지에
따라 다르다.
메인 모듈 내에서 호출된다면 프로세서 성능에 따라 달라진다.

setImmediate() v.s. setTimeout()

메인 모듈 호출

```
// timeout_vs_immediate.js
setTimeout(() => {
  console.log('timeout');
}, 0);

setImmediate(() => {
  console.log('immediate');
});
```

```
$ node timeout_vs_immediate.js
timeout
immediate

$ node timeout_vs_immediate.js
immediate
timeout
```

setImmediate() v.s. setTimeout()

호출 시기에 따라 다르게 동작

그러나 I/O 주기 안에서 둘을 호출한다면
setImmediate() 콜백이 항상 먼저 실행

setImmedate() v.s. setTimeout()

I/O 주기 안에서 호출

```
// timeout_vs_immediate.js
const fs = require('fs');

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout');
  }, 0);
  setImmediate(() => {
    console.log('immediate');
  });
});
```

```
$ node timeout_vs_immediate.js
immediate
timeout

$ node timeout_vs_immediate.js
immediate
timeout
```

setImmediate() v.s. setTimeout()

setImmediate의 장점

얼마나 많은 타이머가 있는지 상관 없이
I/O 주기에선 항상 먼저 실행

process.nextTick()

process.nextTick()

이해하기

비동기 API지만 다이어그램에는 표시되지 않음
기술적으로 이벤트 루프의 일부가 아니다.

process.nextTick()

이해하기

nextTickQueue는 이벤트 루프의 현재 단계와 관계없이 현재 작업이 완료된 후 처리된다.

작업 : C/C++ 핸들러에서 전환되는 것, 실행되어야 하는 JavaScript 처리

process.nextTick()

주의 사항

process.nextTick() 재귀로 호출 할 시 주의 사항
모든 콜백은 이벤트 루프를 계속 진행하기 전에 처리되므로
poll 단계 진입을 막아 I/O 작업이 실행되지 않을 수 있다.

process.nextTick()

왜 이런 동작을 포함했는지

API는 그럴 필요가 없더라도 항상 비동기여야 하는 설계 철학

process.nextTick()

예제

```
function apiCall(arg, callback) {
  if (typeof arg !== 'string')
    return process.nextTick(callback,
                            new TypeError('argument should be string'));
}
```

process.nextTick()

장점

이벤트 루프가 진행되기 전에 apiCall() 콜백 실행을 보장

process.nextTick()

예시 2

```
let bar;

// 비동기 시그니처를 갖지만, 동기로 콜백을 호출합니다.
function someAsyncApiCall(callback) { callback(); }

// `someAsyncApiCall`이 완료되면 콜백을 호출한다.
someAsyncApiCall(() => {
  // someAsyncApiCall는 완료되었지만, bar에는 어떤 값도 할당되지 않았다.
  console.log('bar', bar); // undefined
});

bar = 1;
```

process.nextTick()

예시 2

비동기 함수 이름이지만, 동기로 실행되어 없는 변수에 참조

process.nextTick()

예시 2

```
let bar;

function someAsyncApiCall(callback) {
  process.nextTick(callback);
}

someAsyncApiCall(() => {
  console.log('bar', bar); // 1
});

bar = 1;
```

process.nextTick()

예시 2

모든 변수, 함수 등이 호출되는 콜백보다 먼저 초기화
이벤트 루프가 계속 진행되지 않음

process.nextTick()

예시 3

```
const server = net.createServer(() => {}).listen(8080);

server.on('listening', () => {});
```

process.nextTick()

예시 3

‘listening’ 콜백을 호출하게 되지만
on(‘listening’) 콜백이 설정되지 않음
따라서 nextTick() 으로 큐에 넣어 스크립트가 완료할 때 까지 실행되도록 함
어떤 이벤트 핸들러라도 설정할 수 있음

process.nextTick()

v.s.

setImmedate()

process.nextTick() v.s. setImmedate()

차이점

- process.nextTick() : 같은 단계에서 바로 호출
- setImmediate() : 다음 순회나 이벤트 루프의 'tick' 실행
- 서로 이름이 바뀌어야 함
- setImmedate() 가 예상하기 쉬우므로 모든 경우에 권장

process.nextTick() v.s. setImmedate()

nextTick() 을 사용하는 이유

- 이벤트 루프 계속하기 전에 오류 처리 및 불필요한 자원 정리하여 요청을 다시 시도할 수 있음
- 호출 스택이 풀린 뒤에도 이벤트 루프를 계속하기 전에 콜백을 실행해야 하는 경우

process.nextTick() v.s. setImmedate()

예제 - 사용자의 기대 맞추기

```
const server = net.createServer();
server.on('connection', (conn) => { });

server.listen(8080);
server.on('listening', () => { });
```

process.nextTick() v.s. setImmedate()

예제 - 사용자의 기대 맞추기

- ‘listening’ 콜백 : setImmediate()
- poll 단계를 반드시 거쳐야 함
- 이 전에 연결을 받을 가능성이 존재

process.nextTick() v.s. setImmedate()

예제 - EventEmiter

```
const EventEmitter = require('events');
const util = require('util');

function MyEmitter() {
  EventEmitter.call(this);
  this.emit('event');
}
util.inherits(MyEmitter, EventEmitter);

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

process.nextTick() v.s. setImmedate()

예제 - EventEmiter

- 콜백을 이벤트에 할당한 시점에 스크립트가 실행되지 않음
- 생성자에서 발생시킨 이벤트는 즉시 실행되지 않음

process.nextTick() v.s. setImmedate()

예제 - EventEmiter

```
const EventEmitter = require('events');
const util = require('util');

function MyEmitter() {
  EventEmitter.call(this);

  // 핸들러가 할당되면 이벤트를 발생시키려고 nextTick을 사용합니다.
  process.nextTick(() => {
    this.emit('event');
  });
}
util.inherits(MyEmitter, EventEmitter);

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

참고 도서

영어 : https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

한글 : https://nodejs.org/ko/docs/guides/event-loop-timers-and-nexttick

감사합니다.