프로세스 동기화

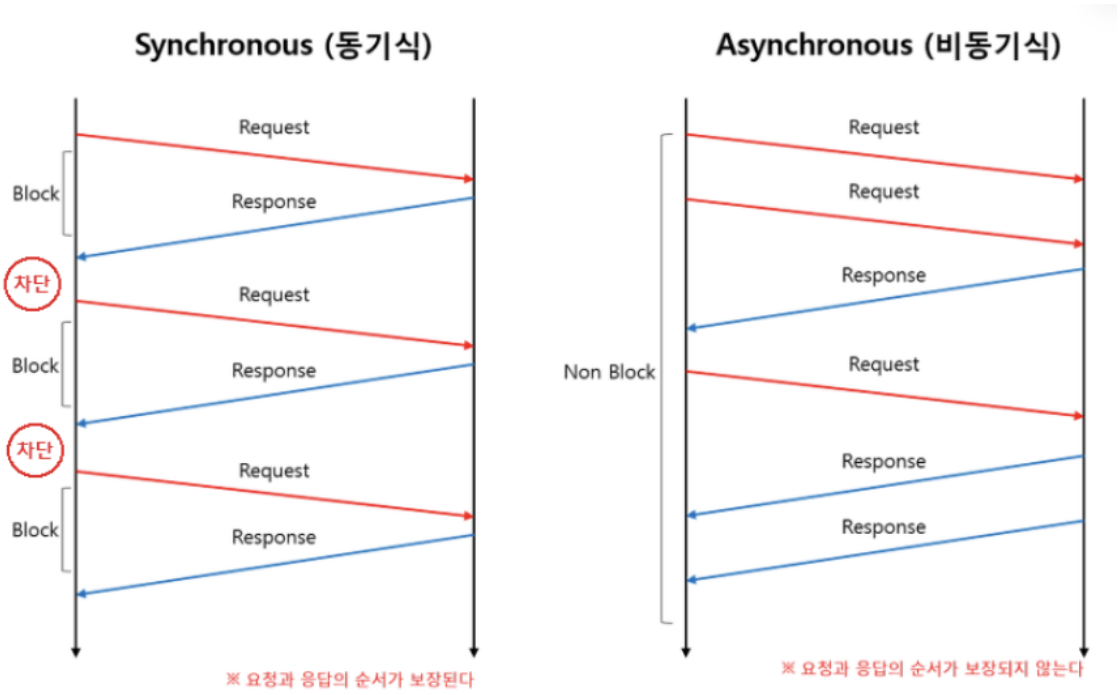
1. 동기(Synchronous) / 비동기(Asynchronous)

- **동기**

  : 요청한 작업에 대한 완료 여부를 따져 순차대로 처리하는 것으로, 요청한 작업에 대한 순서가 지켜진다.

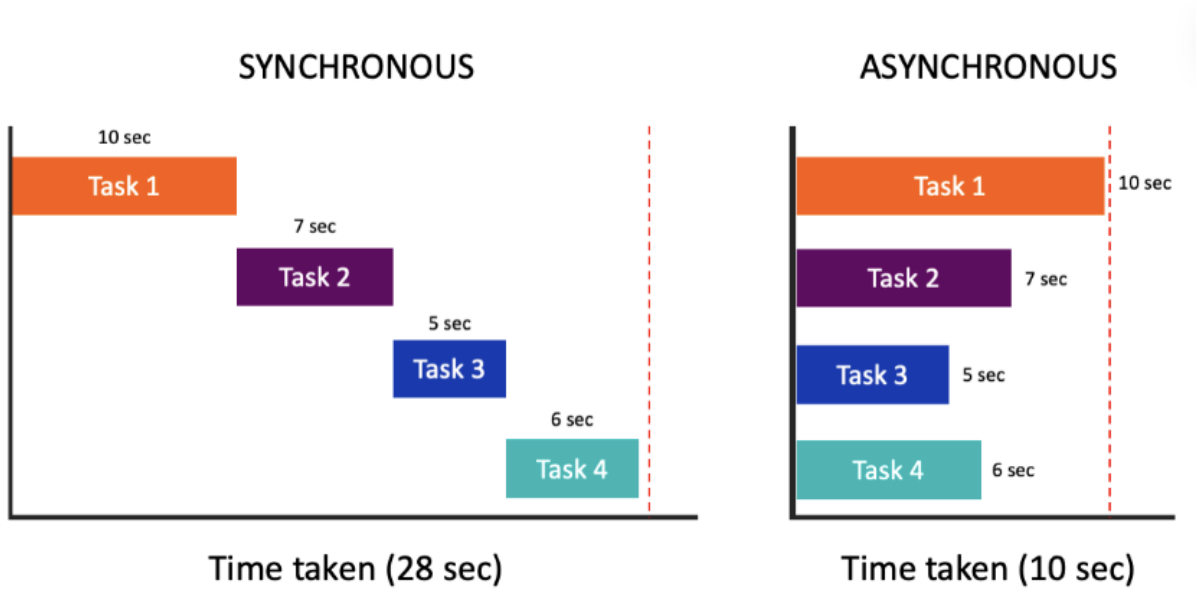
- **비동기**

  : 요청한 작업에 대해 완료 여부를 따지지 않기 때문에 자신의 다음 작업을 그대로 수행하게 된다. 요청한 작업에 대한 순서가 지켜지지 않을 수 있다.



2. 비동기의 성능 이점

: 보통 비동기 특징을 이용하여 성능과 연관지어 말한다. 왜냐하면 요청한 작업에 대하여 완료 여부를 신경쓰지 않고 자신의 그 다음 작업을 수행한다는 것은 **I/O 작업과 같은 느린 작업이 발생할 때, 기다리지 않고 다른 작업을 처리하면서 동시에 처리하여 멀티 작업을 진행할 수 있기 때문이다**. 이는 전반적인 시스템 성능 향상에 도움을 줄 수 있다.



예를 들어, 웹 애플리케이션 데이터베이스 쿼리를 수행하는 작업이 있다고 가정해보자. 이 작업을 동기적으로 수행하면, 데이터베이스에서 응답이 올 때까지 기다려야 한다. 이때 웹 애플리케이션은 다른 요청을 처리하지 못하므로, 대규모 트래픽이 발생할 경우 웹 애플리케이션의 성능이 저하될 수 있다.

하지만, **비동기 방식**으로 데이터베이스 쿼리를 수행하면 데이터베이스에서 응답이 올 때까지 기다리는 동안에도 **다른 요청을 동시에 처리할 수 있어 대규모 트래픽에서도 안정적으로 동작할 수 있는 웹 애플리케이션을 만들 수 있다.**

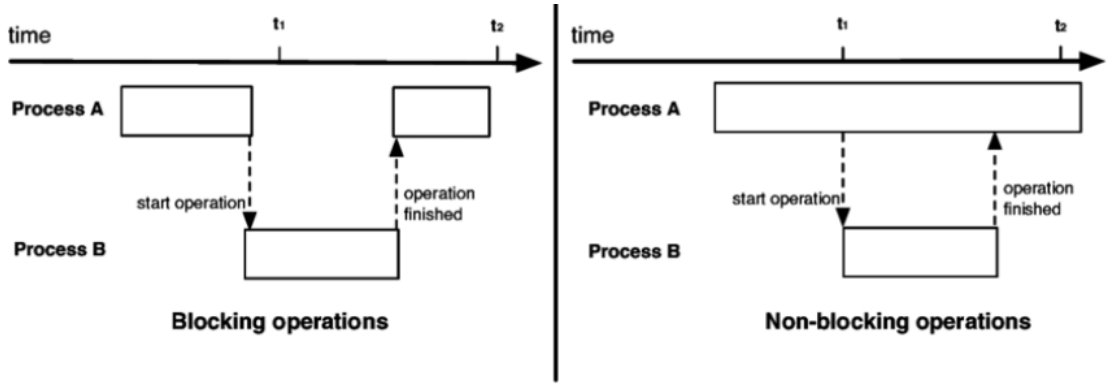
여기서 **'동시 처리'**라는 개념은 두 개 이상의 작업이 동시에 실행되는 것을 의미한다. 이는 **멀티 스레드나 멀티 프로세싱**과 같은 방식으로 구현할 수 있다.

자바스크립트 같은 경우 비동기로 작업을 요청하면 브라우저에 내장된 멀티 스레드로 이루어진 Web API에 작업이 인가되어 메인 Call Stack과 작업이 동시에 처리되게 된다. 쉽게 비유하자면 작업을 백그라운드에 인가한다고 생각하면 된다. 대표적인 **비동기 작업**의 종류로는 **애니메이션 실행, 네트워크 통신, 마우스 키보드 입력, 타이머** 등 많다. 다만 자바스크립트 코드 실행 자체는 Web API가 아닌 Call Stack에서 실행된다.

> 💡 **콜 스택**(call stack)이란 컴퓨터 프로그램에서 현재 실행 중인 서브루틴에 관한 정보를 저장하는 스택 자료구조이다.

3. Blocking / Non-Blocking

: 블로킹과 논블로킹은 다른 요청의 작업을 처리하기 위해 현재 작업을 block(차단, 대기) 하냐 안하냐의 유무를 나타내는 프로세스의 실행 방식이다.



동기/비동기가 전체적인 작업에 대한 순차적인 흐름 유무라면, **블로킹/논블로킹은 전체적인 작업의 흐름 자체를 막냐 안막냐로 볼수 있는 것**이다.

예를 들어, 파일을 읽는 작업이 있을 때, **블로킹 방식으로 읽으면 파일을 다 읽을 때까지 대기하고**, 논블로킹 방식으로 읽으면 파일을 다 읽지 않아도 다른 작업을 할 수 있다.

```
console.log("시작");

setTimeout(() => {
  console.log("1초 후에 실행됩니다!");
}, 1000);

console.log("끝");
```

위 코드의 실행 결과는

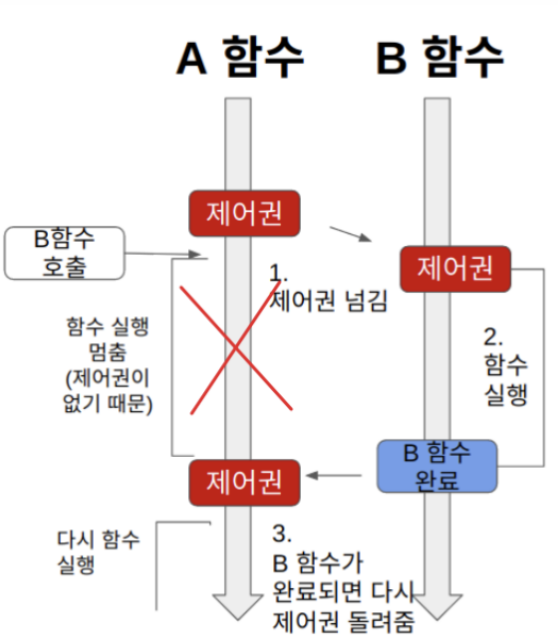
```
시작
끝
1초 후에 실행됩니다!
```

이는 `setTimeout` 함수가 비동기 함수여서 타이머 작업 완료 여부를 신경 쓰지 않고 바로 그 다음 콘솔 작업을 수행하였기 때문이다.

- 동기식 콜백의 예 : `Array.prototype.map()` , `Array.prototype.forEach()`
- 비동기식 콜백의 예 : `setTimeout()` , `Promise.prototype.then()`

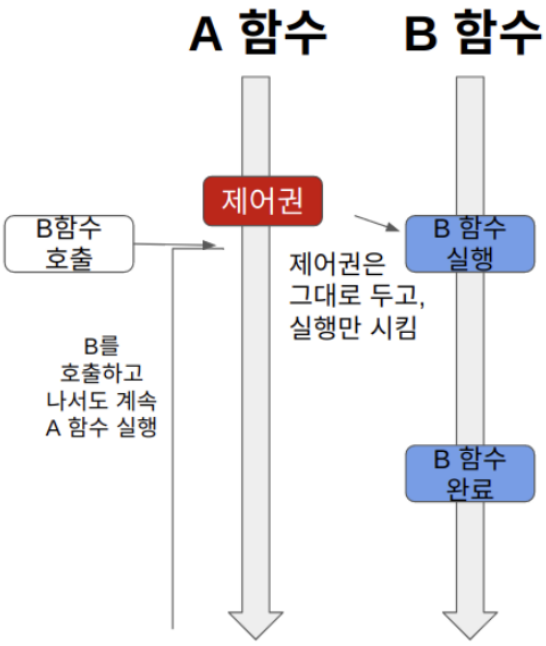
4. Blocking과 Non-Blocking의 제어권

- **Blocking**



1. A함수가 B함수를 호출하면 B에게 제어권이 넘어간다.
2. 제어권을 넘겨 받은 B는 함수를 실행한다.
3. 이때, A는 B에게 제어권을 넘겨주었기 때문에 함수 실행을 잠시 멈춘다. (Block)
4. B함수는 실행이 끝나면 A에게 제어권을 다시 넘겨주고, A 함수는 그 다음 작업을 실행한다.

- **Non-Blocking**



1. A 함수가 B 함수를 호출한다.
2. 호출된 B 함수는 실행되지만, 제어권은 A 함수가 그대로 가지고 있는다.
3. A 함수는 계속 제어권을 가지고 있기 때문에 B 함수를 호출한 이후에도 자신의 코드를 계속 실행된다.

5. 동기/비동기 + 블로킹/논블로킹 조합

1. **Sync Blocking (동기 + 블로킹)**

```
다른 작업이 진행되는 동안 자신의 작업을 처리하지 않고 (Blocking), 다른 작업의 완료 여부를 바로 받아 순차적으로 처리하는(Sync) 방식
```

2. **Async Blocking (비동기 + 블로킹)**

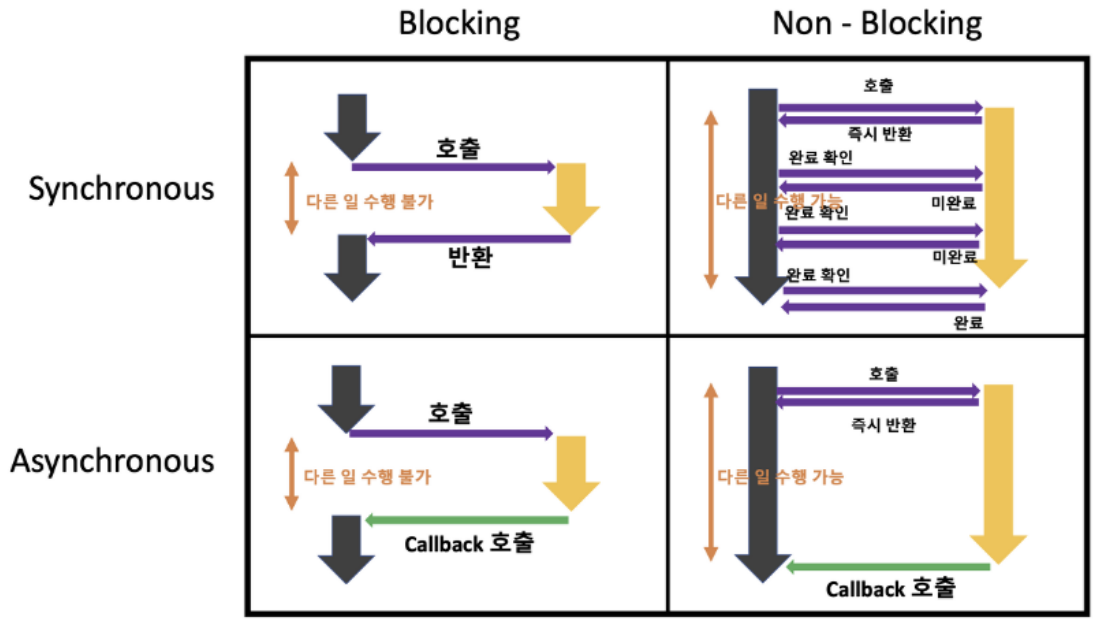
```
다른 작업이 진행되는 동안에도 자신의 작업을 처리하고 (Non Blocking), 다른 작업의 결과를 바로 처리하지 않아 작업 순서가 지켜지지 않는 (Async)
```

3. **Sync Non-Blocking (동기 + 논블로킹)**

다른 작업이 진행되는 동안에도 자신의 작업을 처리하고 (Non Blocking),
다른 작업의 결과를 바로 처리하여 작업을 순차대로 수행 하는 (Sync) 방식

4. **Async Non-Blocking (비동기 + 논블로킹)**

다른 작업이 진행되는 동안에도 자신의 작업을 처리하고 (Non Blocking),
다른 작업의 결과를 바로 처리하지 않아 작업 순서가 지켜지지 않는 (Async) 방식



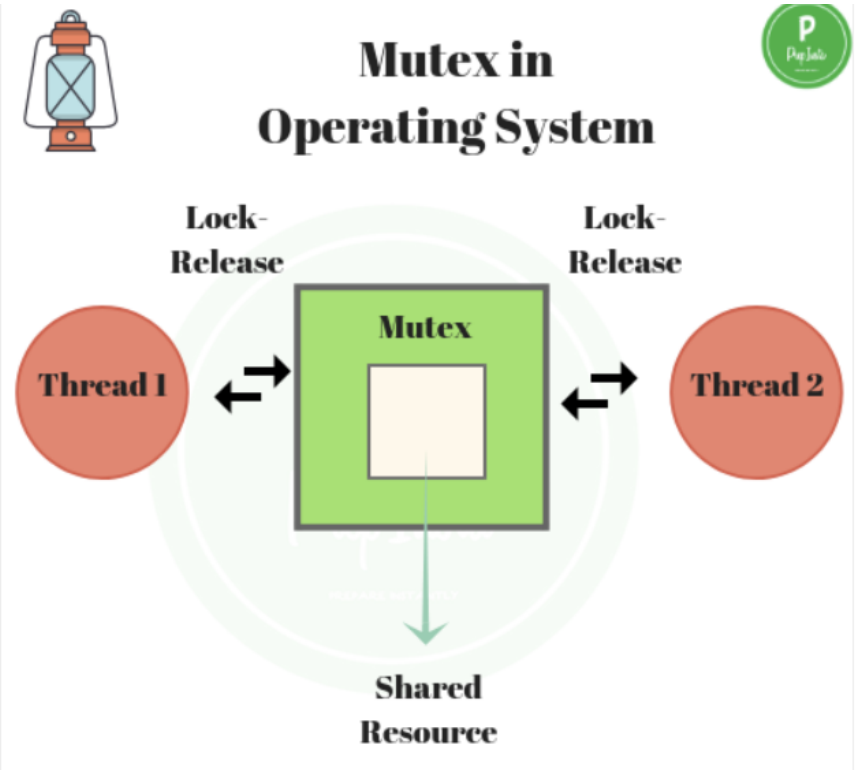
6. 뮤텍스와 세마포어

공통점 : 동기화를 위해 이용되는 도구

차이점 :

`뮤텍스 (Mutex)`

Key에 해당하는 어떤 오브젝트가 있으며, 이 오브젝트를 소유한 쓰레드 혹은 프로세스만이 공유 자원에 접근할 수 있다.

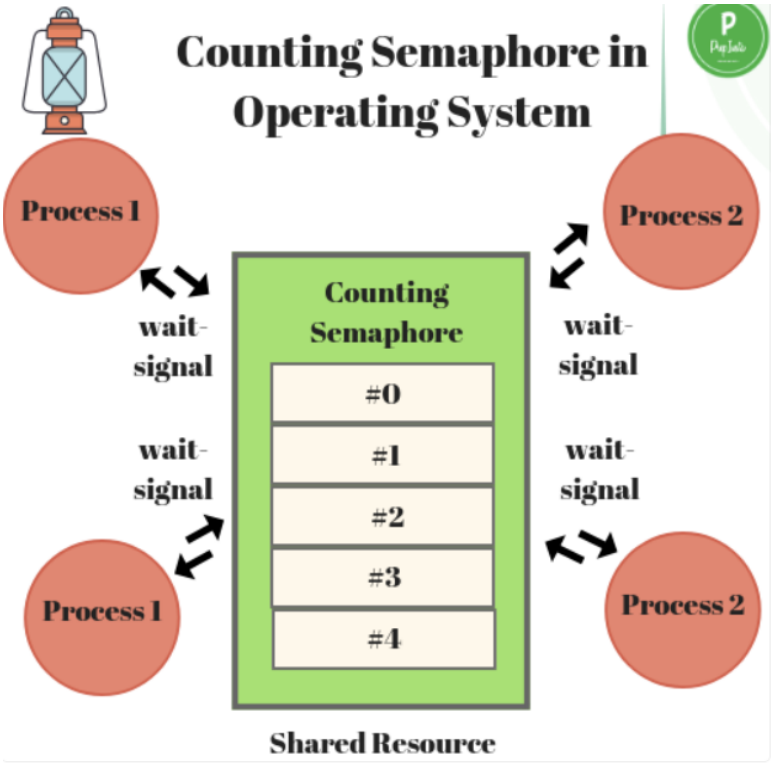


과정

- 1번 프로세스가 자원을 접근하기 위해 Key를 점유한다.
- 1번 프로세스는 키를 점유했기 때문에 공유 자원을 사용한다.
- 2번 프로세스가 공유 자원을 사용하기를 원한다.
- 2번 프로세스는 키를 점유하기 위해 대기한다.
- 3번 프로세스 또한 공유 자원을 사용하기 위해 2번 프로세스 다음 순번으로 대기한다.
- 1번 프로세스가 공유 자원을 다 사용하고 Key를 반환한다.
- 2번 프로세스는 대기하고 있다가 반환된 Key를 점유하고 공유 자원을 사용한다.

`세마포어 (Semaphore)`

: 공유 리소스에 접근할 수 있는 최대 허용치만큼 동시 사용자(쓰레드, 프로세스) 접근을 허용한다.



과정

- 공유 자원에 대한 최대 허용치를 정의한다. 우선, 3으로 해보겠다.
- 1번 프로세스가 공유 자원에 접근한다. 허용치는 2로 감소하였다.
- 2번 프로세스가 공유 자원에 접근한다. 허용치는 1로 감소하였다.
- 3번 프로세스가 공유 자원에 접근한다. 허용치는 0로 감소하였다.
- 4번 프로세스가 공유 자원에 접근한다. 허용치가 0이므로 대기한다.
- 2번 프로세스가 공유 자원을 다 사용하였다. 허용치는 1로 증가하였다.
- 4번 프로세스는 대기 하다 허용치가 1로 증가되어 공유 자원에 접근한다. 허용치는 다시 0으로 감소하였다.

<정리>

1. 세마포어는 뮤텍스가 될 수 있지만, 뮤텍스는 세마포어가 될 수 없다.
2. 세마포어는 소유할 수 없지만, 뮤텍스는 소유할 수 있고 소유주가 그에 대한 책임을 가진다.

3. 세마포어는 동기화 대상이 여러 개일 때 사용하고, 뮤텍스는 동기화 대상이 오로지 하나일 때 사용된다.