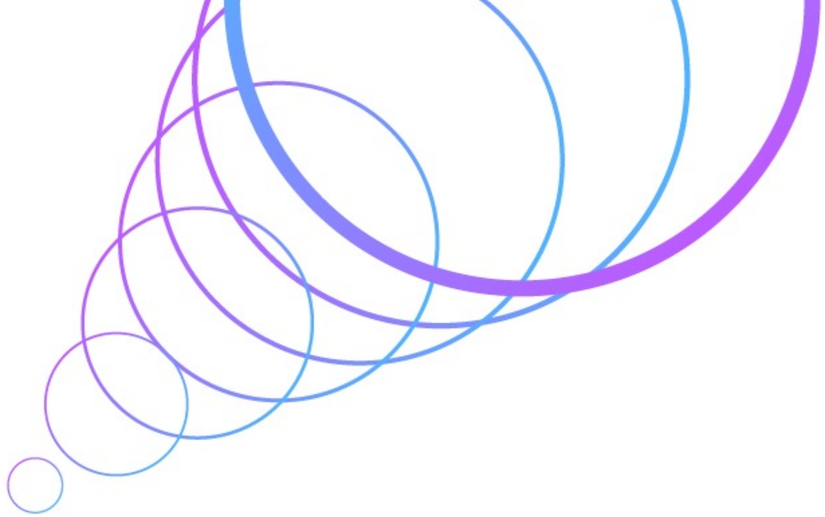
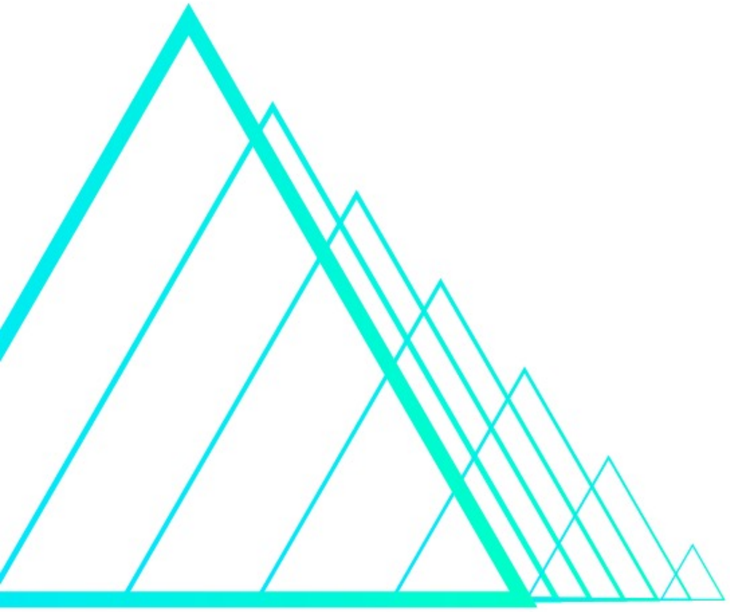


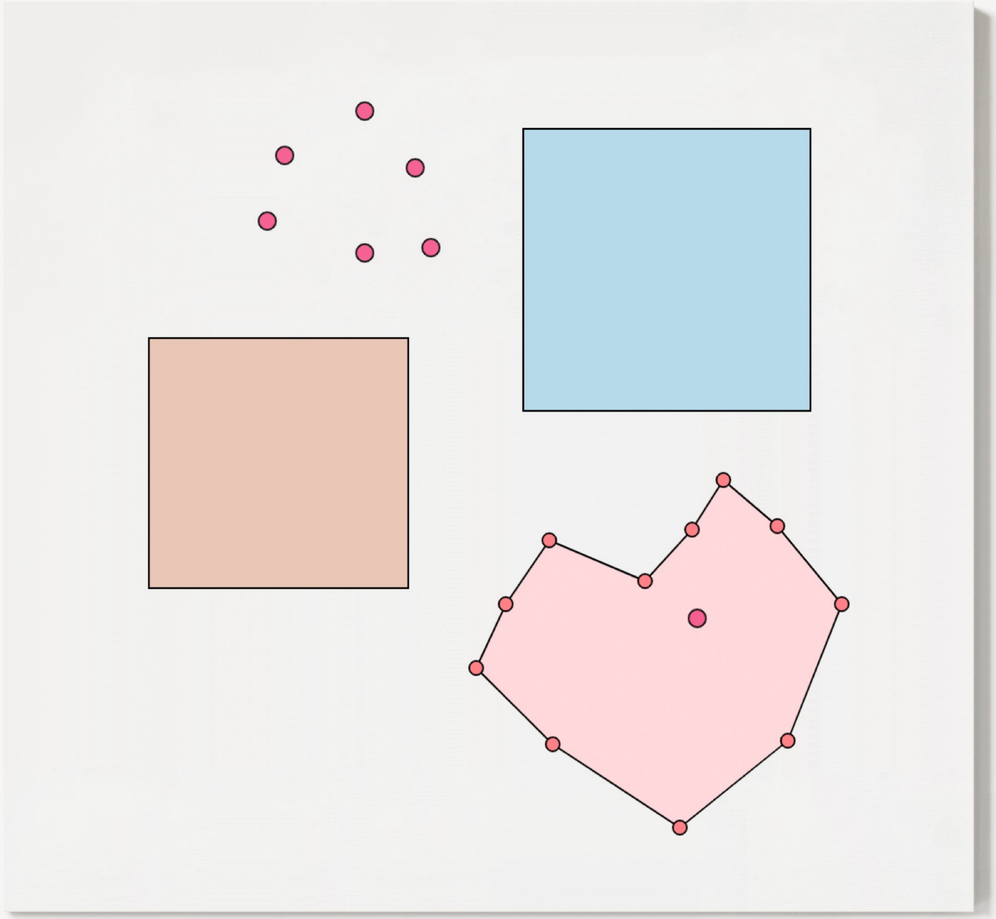


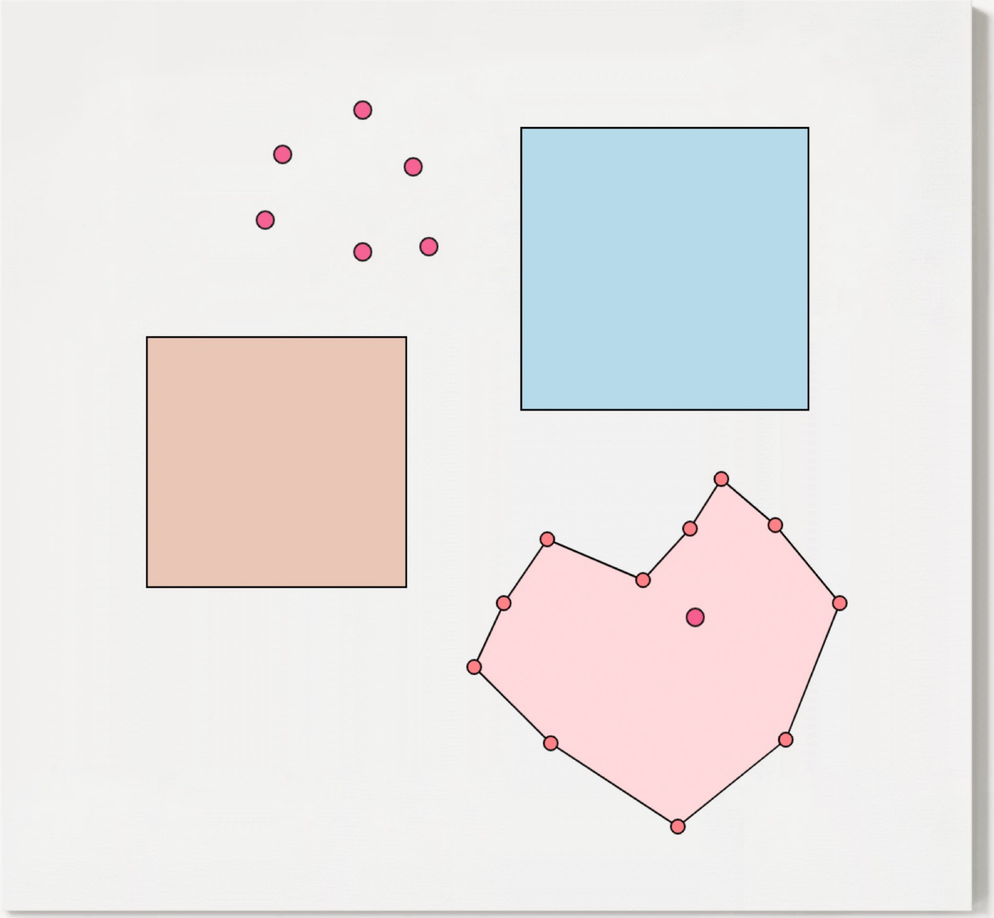
DEVIEW 2021

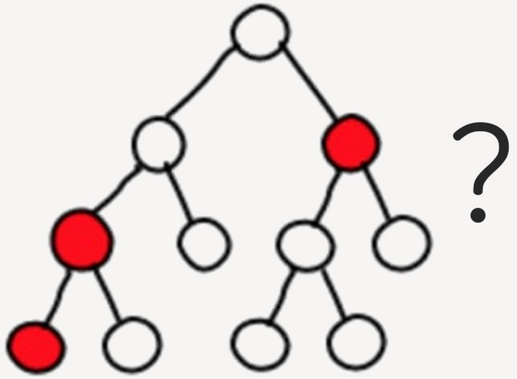
Inside React

동시성을 구현하는 기술

심흥운 NAVER / Platform Labs

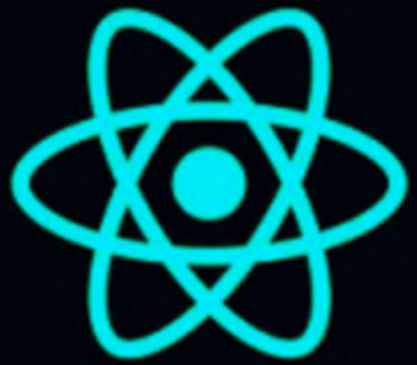






React 18 alpha

with Concurrent Features



CONTENTS

1. 5년의 실험

Five Years of Experiment

1.1 동시성 실험 연대기

DEVIEW 2021

2017. 9.

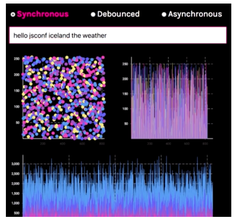
React v16.0

New core architecture

Codename “Fiber”

Async rendering

2018. 3.



Beyond React 16

JSConf 아이슬란드 2018

Time slicing

Suspense

2018. 10.

React v16.6

Scheduler

Priority levels

Async → Concurrent

2018. 11.

React Conf 2018

React.lazy for code splitting

Concurrent React

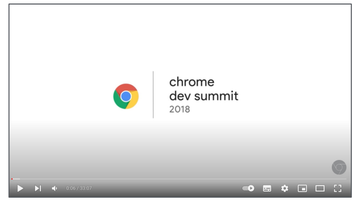
2018. 11.

React v16.x Roadmap

Concurrent mode (~Q2 2019)

ReactDOM.unstable_createRoot

스케쥴링 개선을 위해 크롬 팀과 협업



1.1 동시성 실험 연대기

DEVIEW 2021

2018. 12.

React v16.7

MessageChannel

for scheduler

2019. 8.

React v16.9.0
Roadmap Update

Is Suspense for Data

Fetching ready yet? No

동시성 모드 시간 걸릴 듯

기한 미정

2020. 2.

React v16.13

SuspenseList

Fix isPending

Warn for createRoot

2020. 10.

React v17.0

unstable_startTransition

More than 11 fix for

concurrent mode.

Lane for priority

2021. 6.

The Plan for React 18

React 18 alpha release

Working group

1.2 React 18 Alpha Release

DEVIEW 2021

React 18 릴리즈 계획

- 2021년 6월, 차세대 메이저 버전인 React 18 릴리즈 준비 계획을 발표
  https://reactjs.org/blog/2021/06/08/the-plan-for-react-18.html
- 서드 파티 라이브러리 개발자의 피드백을 받기 위해 알파 버전을 릴리즈
- 커뮤니티가 점진적으로 적용할 수 있도록 Working Group 설립
  https://github.com/reactwg/react-18

1.3 React 18 Release Timeline

DEVIEW 2021

Alpha

- 리엑트 생태계의 주요 라이브러리와 협력하여 필요한 지원을 받기
- 새로운 API 마무리 필요 (useMutableSource, useOpaqueIdentifier 등)

Beta (몇 달)

- 최종 릴리스에 필요한 주요 변경 사항과 새로운 기능이 포함
- 더 큰 커뮤니티를 초대하여 피드백을 수용

RC (몇 달)

- 사용자가 프로덕션 환경에서 테스트하는 데 문제 없는 안정 버전

Stable (RC 후 2~4주)

- React 18.0 Release

https://github.com/reactwg/react-18/discussions/9

1.4 무엇이 달라지나

Out-of-the-box Improvements

- Automatic batching for fewer renders
- SSR support for Suspense
- Fixes for Suspense behavior quirks

Concurrent Features

- startTransition
- useDeferredValue
- <SuspenseList>
- Streaming SSR with selective hydration

https://github.com/reactwg/react-18/discussions/4

1.4 무엇이 달라지나

DEVIEW 2021

Concurrent Rendering Mechanism

- 협력적 멀티태스킹
- 우선순위 기반 렌더링
- 스케쥴링
- 중단 (Interruptions)

https://github.com/reactwg/react-18/discussions/4

1.5 실험적 세 가지 모드

DEVIEW 2021

Installation experimental version

```
npm install react@experimental react-dom@experimental
```

Legacy mode, ~~Blocking mode~~, Concurrent mode 🤔

```
ReactDOM.render(<App />, root)

ReactDOM.createBlockingRoot(root).render(<App />)
// Deprecated (https://github.com/facebook/react/pull/20974)

ReactDOM.createRoot(root).render(<App />)
```

https://reactjs.org/docs/concurrent-mode-intro.html

1.5 실험적 세 가지 모드

DEVIEW 2021

Feature Comparison

	Legacy Mode	Blocking Mode	Concurrent Mode
String Refs	✅	🚫**	🚫**
Legacy Context	✅	🚫**	🚫**
findDOMNode	✅	🚫**	🚫**
Suspense	✅	✅	✅
SuspenseList	🚫	✅	✅
Suspense SSR + Hydration	🚫	✅	✅
Progressive Hydration	🚫	✅	✅
Selective Hydration	🚫	🚫	✅
Cooperative Multitasking	🚫	🚫	✅
Automatic batching of multiple setStates	🚫*	✅	✅
Priority-based Rendering	🚫	🚫	✅
Interruptible Prerendering	🚫	🚫	✅
useTransition	🚫	🚫	✅
useDeferredValue	🚫	🚫	✅
Suspense Reveal “Train”	🚫	🚫	✅

https://reactjs.org/docs/concurrent-mode-adoption.html

1.6 수정된 업그레이드 전략

Legacy mode, Concurrent mode 모두 사용 가능

- 점진적이고 안전한 업그레이드 지원

```
// Legacy Mode
ReactDOM.render(<App />, root)

// Concurrent Mode
ReactDOM.createRoot(root).render(<App />)
```

1.7 “Concurrent” 용어 정리

- ReactDOM.createRoot 실행시 내부적으로 "concurrent mode" 활성화
- 그러나 이 조건에서 반드시 "concurrent rendering" 하는 것은 아님
- "concurrent features" 사용할 때만 "concurrent rendering" 수행

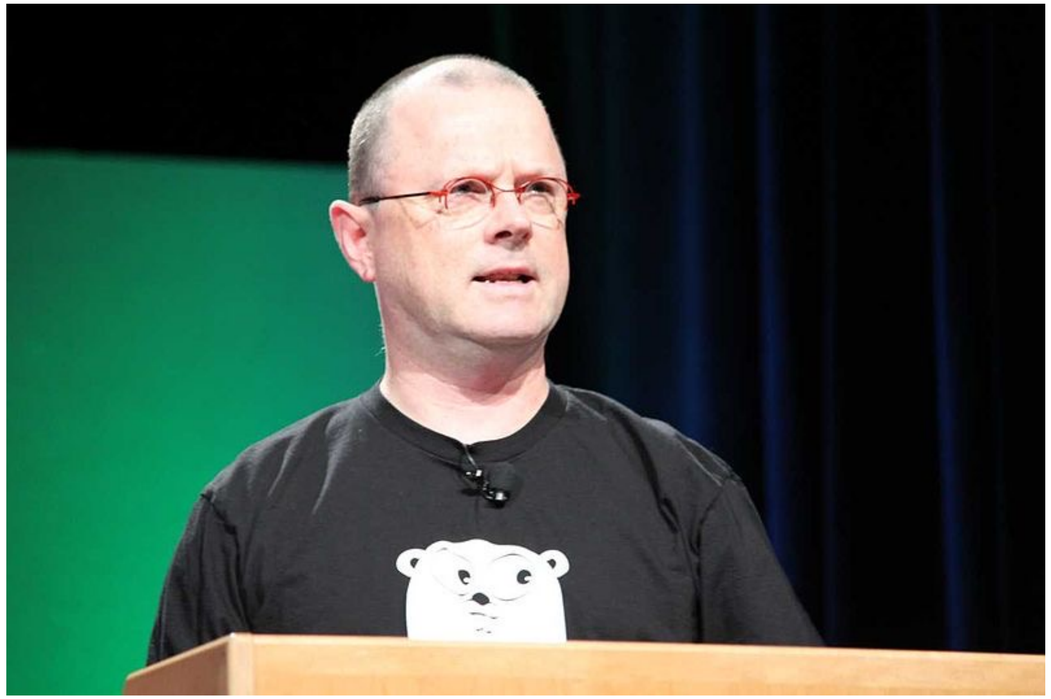
https://github.com/reactwg/react-18/discussions/64

2. 동시성으로 해결하려는 문제

What Problems Can Concurrency in React Help Us Solve?

2.1 Concurrency vs Parallelism

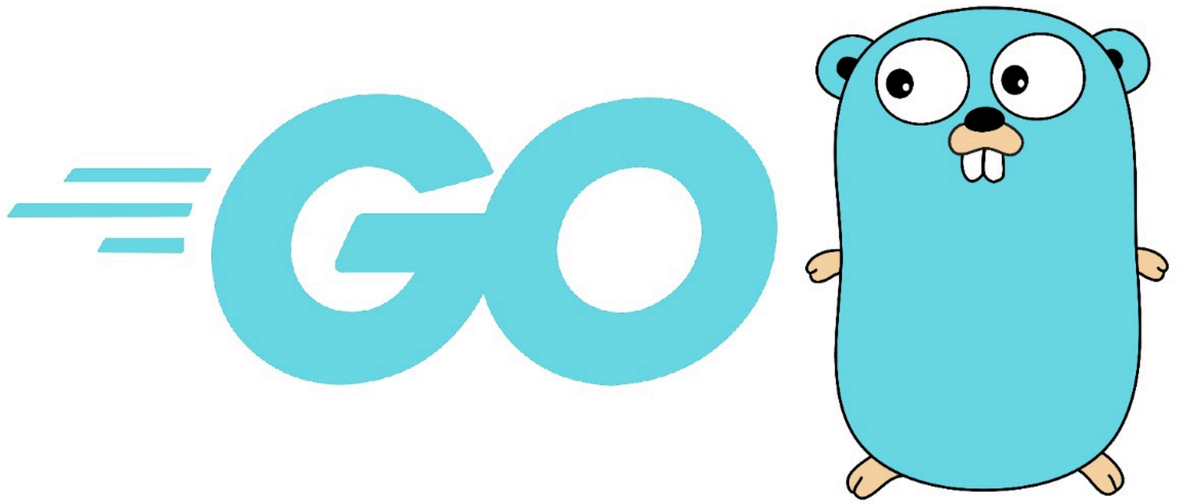
DEVIEW 2021



Robert "Rob" C. Pike

https://commons.wikimedia.org/wiki/File:Rob-pike-oscon.jpg

Photo by Kevin Shockey (CC BY 2.0)



https://golang.org/doc/gopher/README

Designed by Renee French (CC BY 3.0)

https://go.dev/blog/waza-talk

2.1 Concurrency vs Parallelism

DEVIEW 2021

“ 동시성은 독립적으로 실행되는 프로세스들의 조합이다.

병렬성은 연관된 복수의 연산들을 동시에 실행하는 것이다.

동시성은 여러 일을 한꺼번에 다루는 문제에 관한 것이다.

병렬성은 여러 일을 한꺼번에 실행하는 방법에 관한 것이다. ”

https://go.dev/blog/waza-talk

2.1 Concurrency vs Parallelism

DEVIEW 2021

“ 동시성이란 마음의 상태이다
실제로는 여러 개의 스레드가 사용되지는 않지만,
겉으로 보기에는 마치 여러 개의 스레드가
사용되고 있는 것처럼 보이게 만드는 것을 의미한다.
이것이 바로 협동적인 멀티태스킹을 의미한다. ”

7가지 동시성 모델 6.4.1

2.1 Concurrency vs Parallelism

DEVIEW 2021

1487

Concurrency is when two or more tasl
complete in overlapping time **periods**.
mean they'll ever both be running **at th**
example, *multitasking* on a single-core

Parallelism is when tasks *literally* run a
on a multicore processor.

Quoting Sun's *Multithreaded Programm*

- Concurrency: A condition that exis
threads are making progress. A m
parallelism that can include time-slicing as a form of
virtual parallelism.
- Parallelism: A condition that arises when at least two
threads are executing simultaneously.

wiki.haskell.org/Parallelism_vs._Concurrency

Page Discussion Read View sour

Navigation
Haskell
Wiki community
Recent changes
Random page

Tools

Parallelism vs. Concurrency

The term **Parallelism** refers to techniques to make programs
the same time. This requires hardware with multiple process
are of the same structure, but this is not necessary. Graphic
problem of parallelism is to reduce data dependencies in orc
independent computation units with minimal communicatior
advantage to do the same computation twice on different ur

The Pragmatic Programmers

Seven Concurrency Models in Seven Weeks

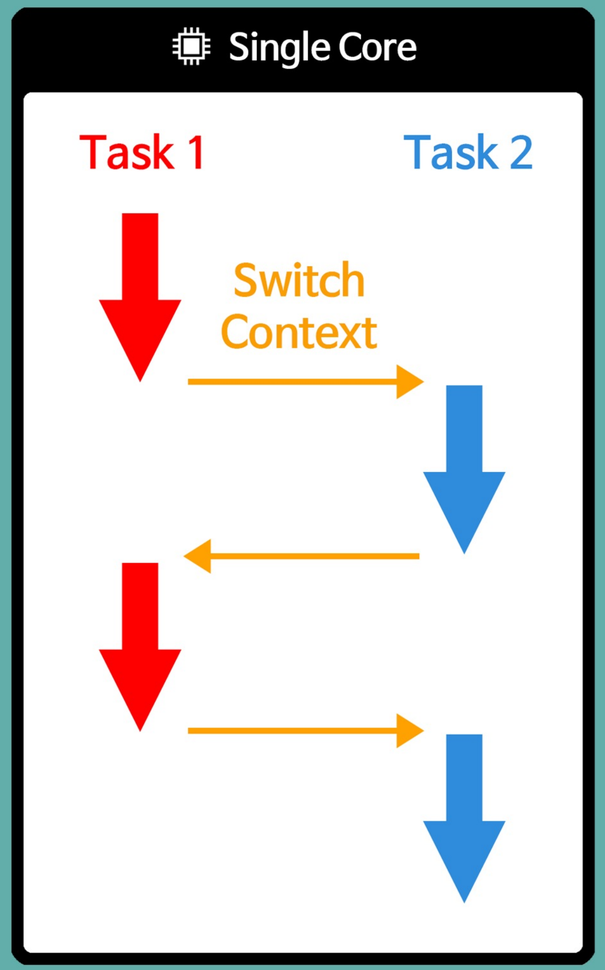
When Threads Unravel

Paul Butcher

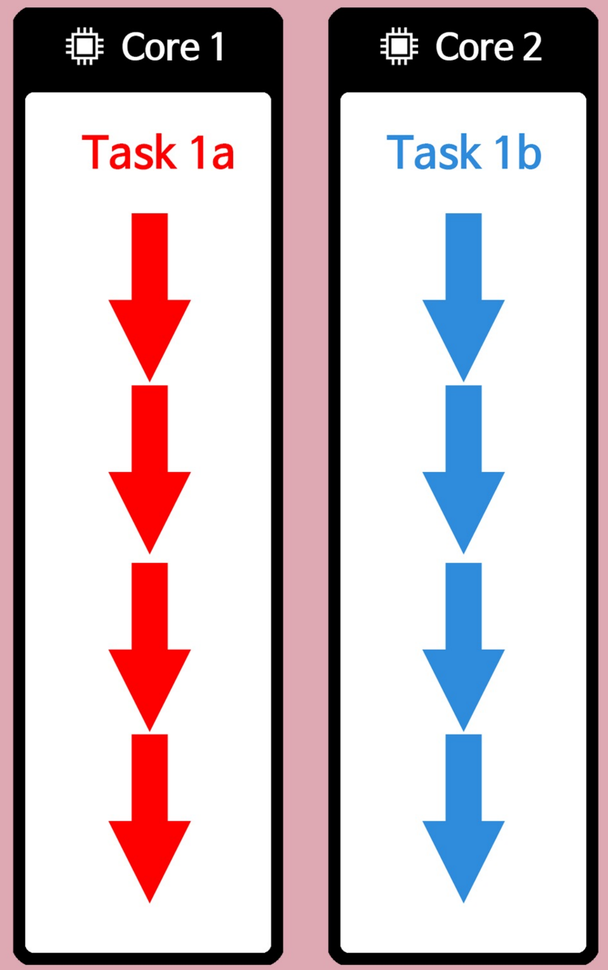
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism
https://wiki.haskell.org/Parallelism_vs._Concurrency
https://www.hanbit.co.kr/store/books/look.php?p_code=B3745244799

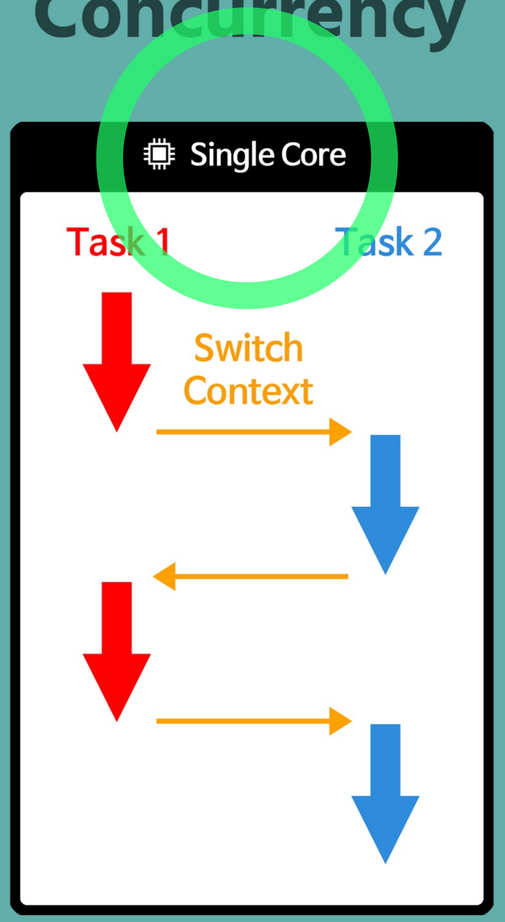
Concurrency



Parallelism

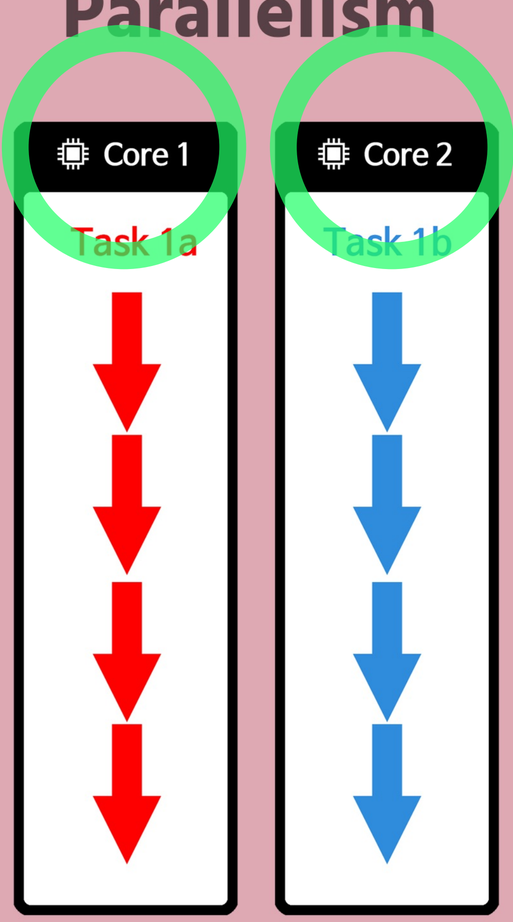


Concurrency



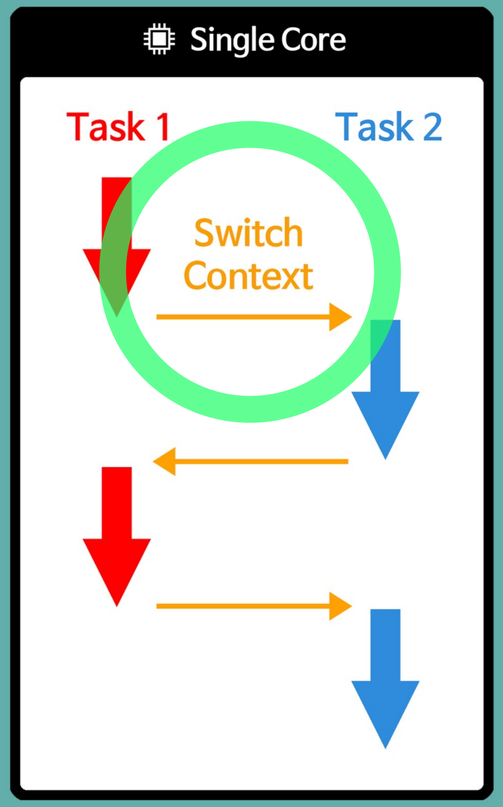
싱글 코어에서도 작동

Parallelism



멀티 코어가 필요

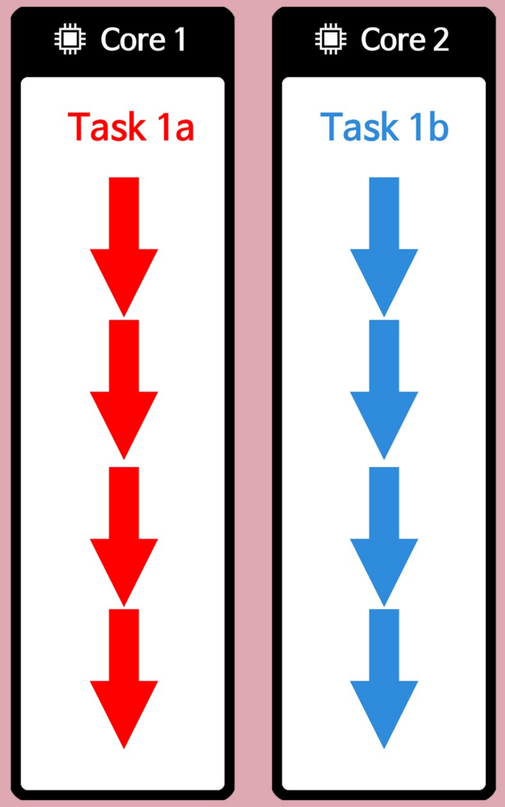
Concurrency



동시에 실행되는 것처럼 보임

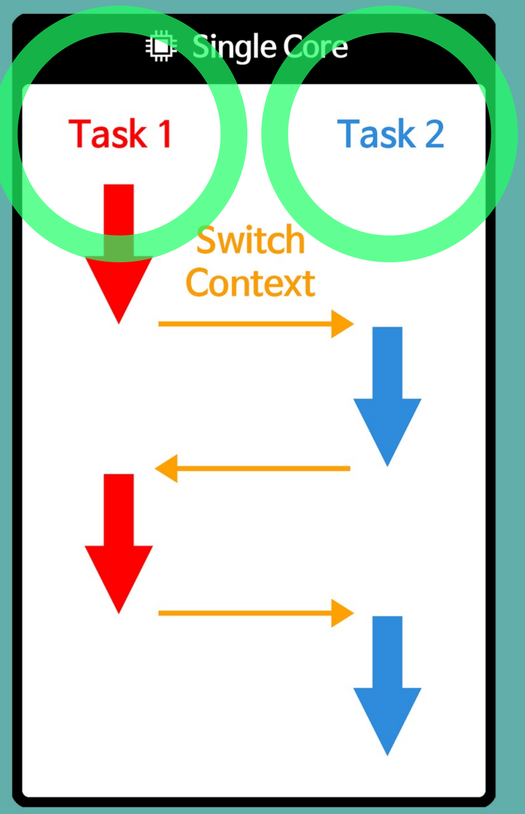
컨텍스트 스위칭 필요

Parallelism



실제로 동시에 실행 됨

Concurrency



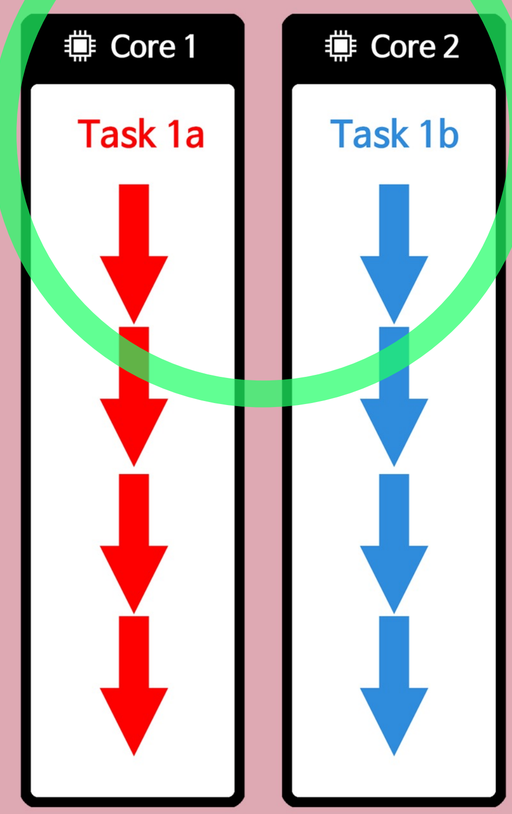
Task1

Task2

최소 두 개의 논리적 통제 흐름

예) 동영상 재생하면서 메모장 쓰기

Parallelism

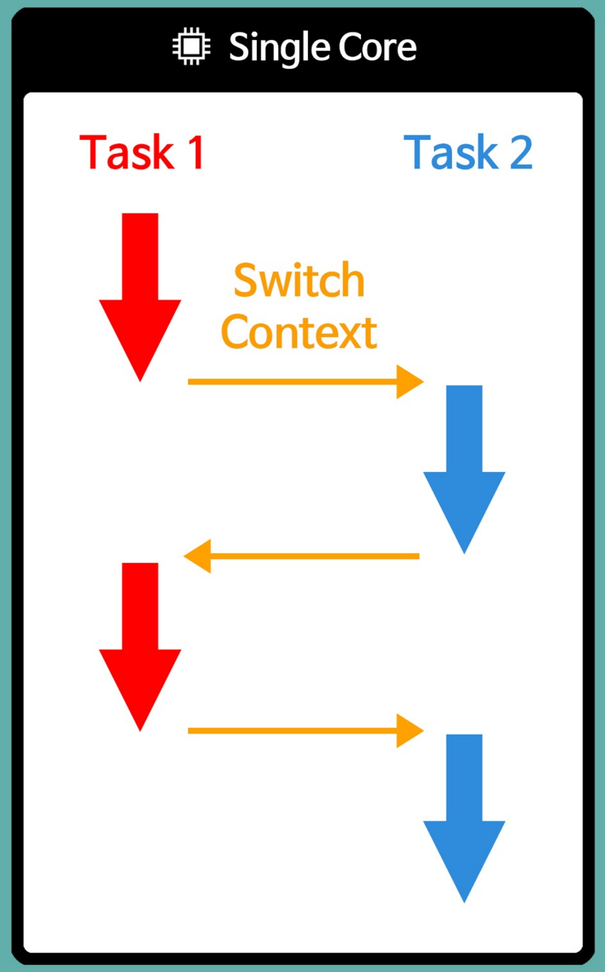


Task1

최소 한 개의 논리적 통제 흐름

예) GPU를 이용해 이미지 출력하기

Concurrency



“ 동시성이란
2개 이상의 독립적인 작업을 잘게 나누어
동시에 실행되는 것처럼 보이도록
프로그램을 구조화하는 방법이다 ”

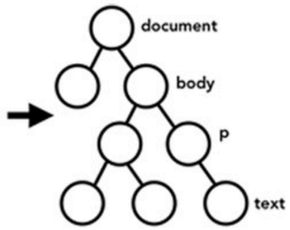
2.2 Blocking Rendering 문제

DEVIEW 2021

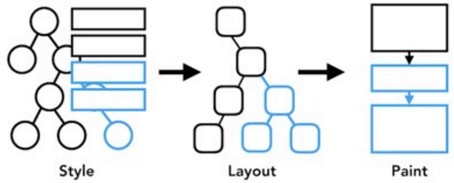
Renderer Process

한 번에
하나씩!

```
<!DOCTYPE html>
<html>
  <head>
    <link href="zzz.css">
  </head>
  <body>
    <img src="horse.png">
    <p>
      Who is HorseJS
    </p>
    <script src="zzz.js">
    </script>
  </body>
```




```
<script>
import awesomeJS from ...
awesomeJS.run()
...
</script>
```



https://developers.google.com/web/updates/2018/09/inside-browser-part1

2.2 Blocking Rendering 문제

Renderer Process


```
<script>
ReactDOM.render(<App />, root)
</script>
```

한번 리엑트
렌더링 연산에
돌입하면
멈출 수 없다!

2.2 Blocking Rendering 문제

Renderer Process

그런데
이 연산이
매우 오래
지속된다면?

```
<script>
ReactDOM.render(<App />, root)
</script>
```

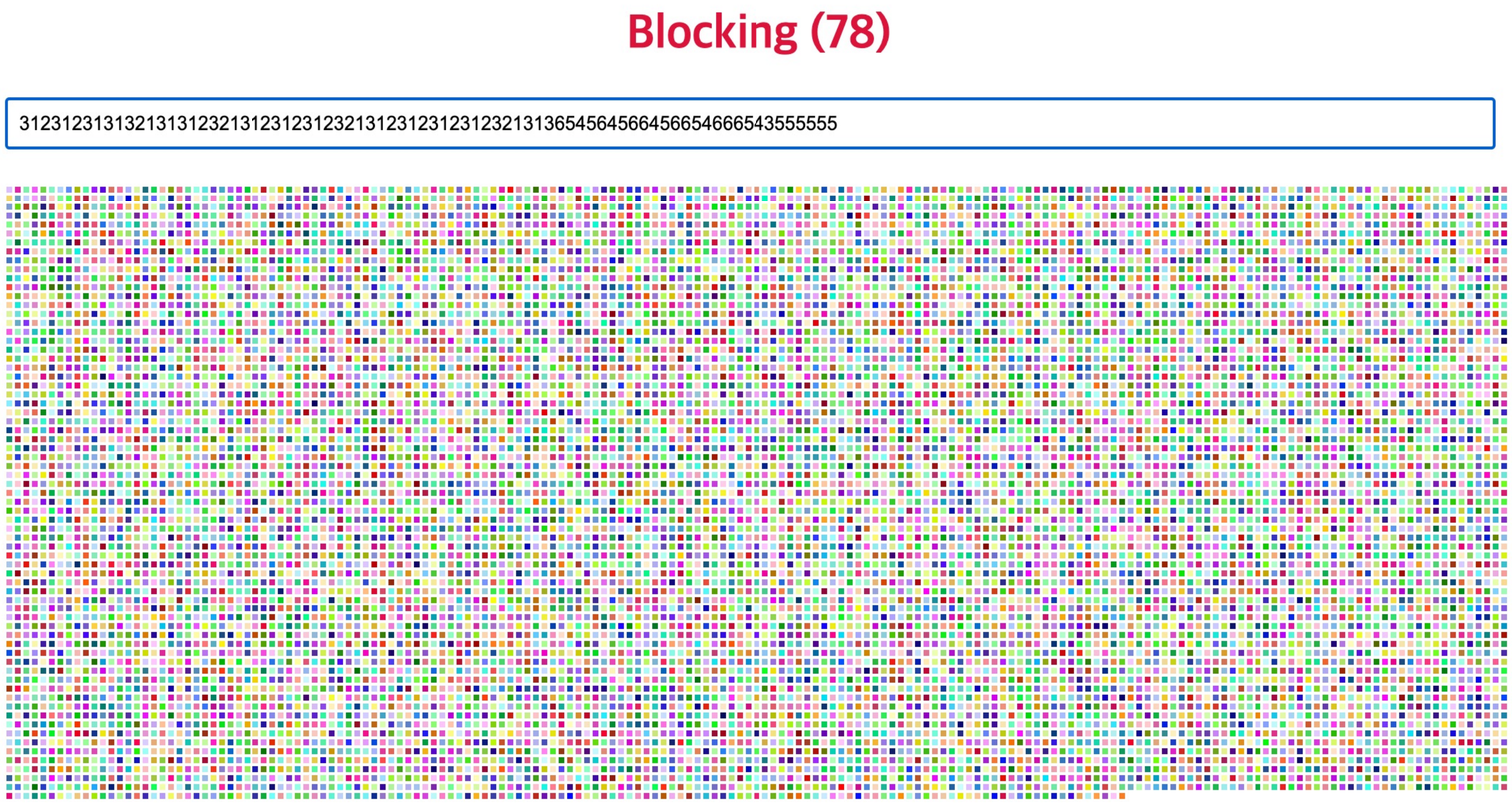
React Rendering Operation

2.2 Blocking Rendering 문제

DEVIEW 2021

https://ajaxlab.github.io/deview2021/blocking/

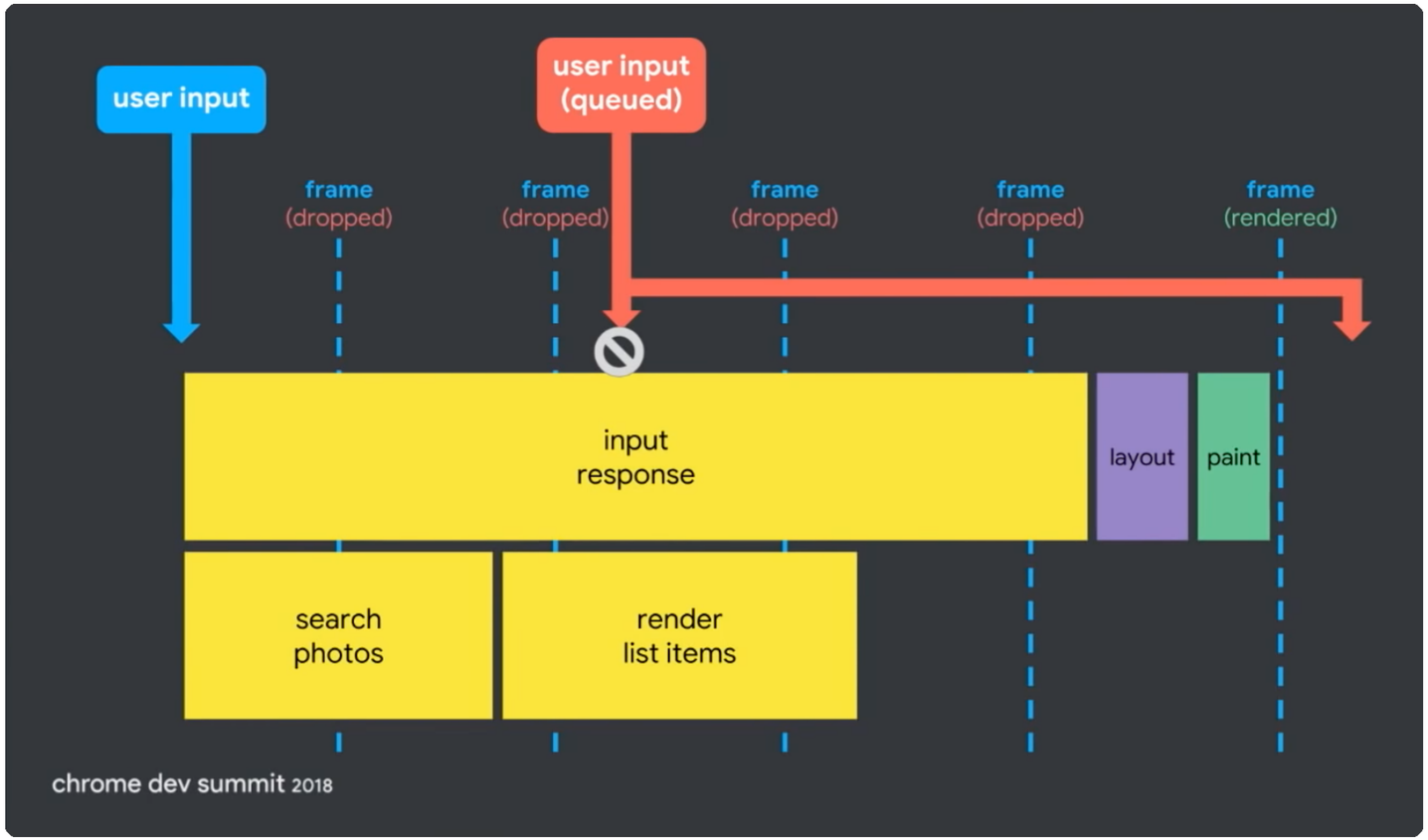
```
const { useState } = React;

function App() {

  const [text, setText] = useState('');

  return (
    <div className='container'>
      <h1>Blocking ({text.length})</h1>
      <input
        type="text"
        value={text}
        maxLength={100}
        onChange={({target}) => {
          setText(target.value);
        }}
      />
      <ColorList length={text.length} />
    </div>
  );
}
```



2.2 Blocking Rendering 문제

DEVIEW 2021

Blocking Rendering 과정 : Recursive



https://www.youtube.com/watch?v=mDdgfyRB5kg&t=94s

2.3 적당한 지연은 어떤가요?

입력하는 동안 화면을 그리지 않도록 하는 디바운싱

```
_.debounce(callback, [wait=0])
```

입력하는 동안 일정한 주기로 화면을 그리도록 하는 스로틀링

```
_.throttle(callback, [wait=0])
```

2.4 동시성 렌더링으로 해결해 보자

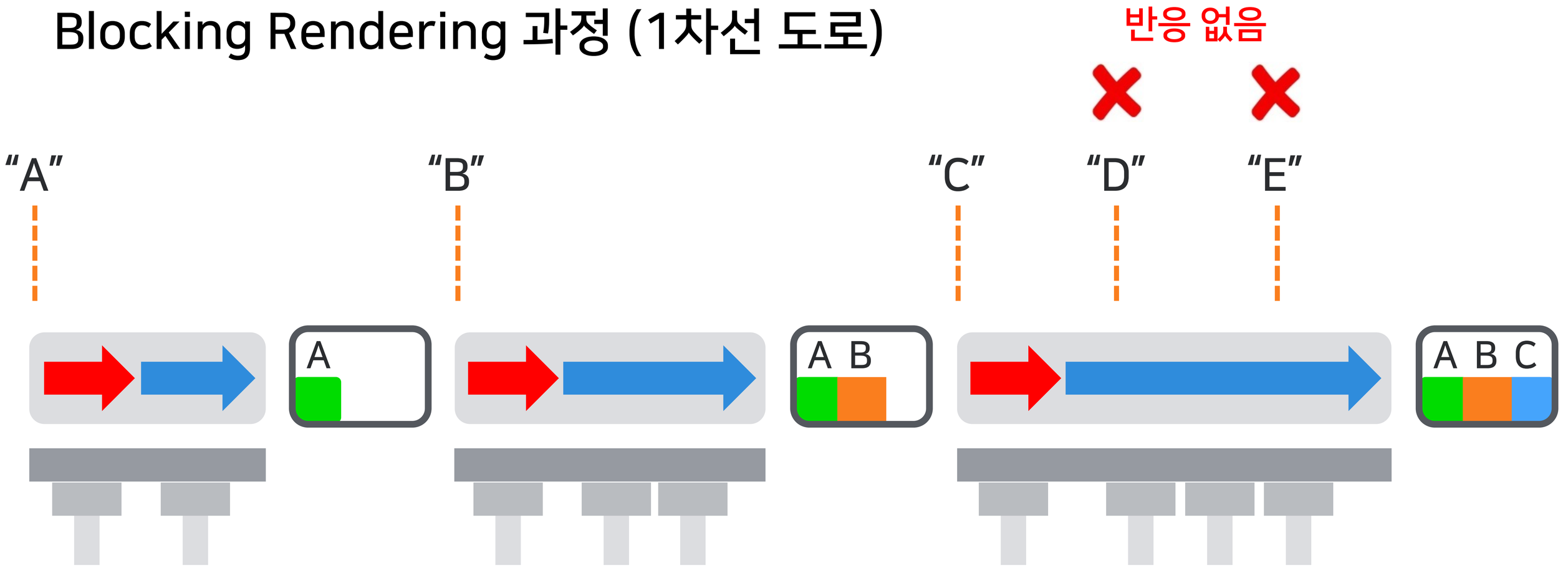
급한 작업

덜 급한 작업

2.4 동시성 렌더링으로 해결해 보자

DEVIEW 2021

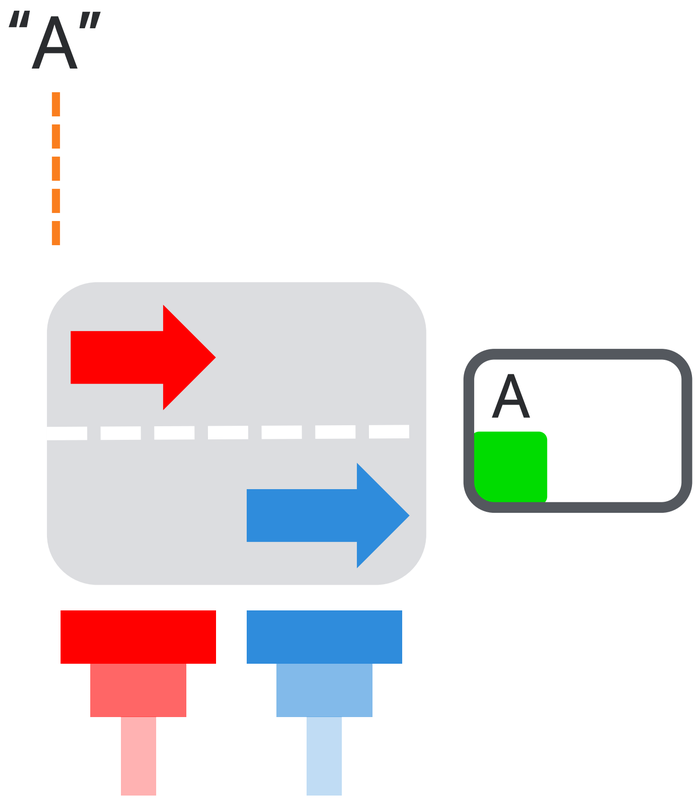
Blocking Rendering 과정 (1차선 도로)



2.4 동시성 렌더링으로 해결해 보자

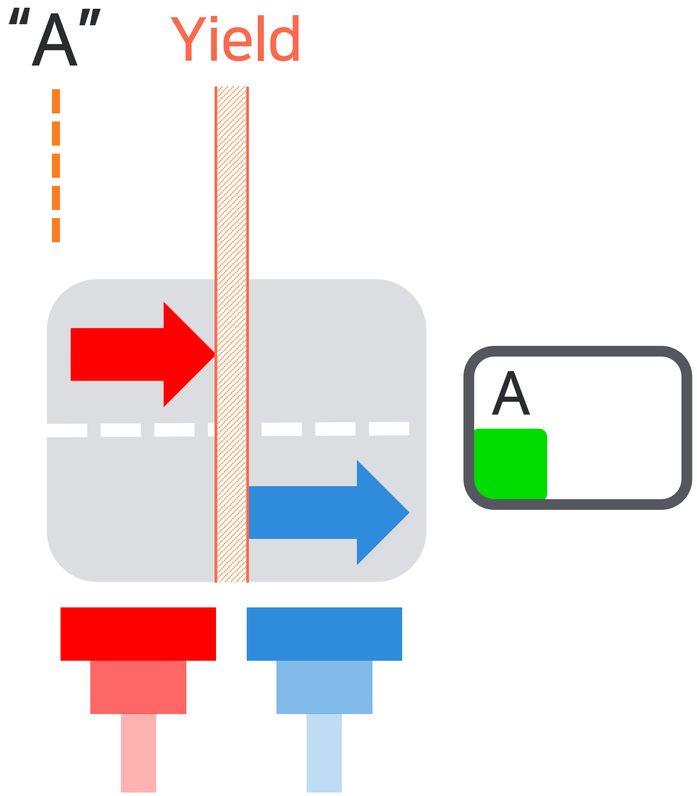
DEVIEW 2021

작업을 병렬로 배치 (2차선 도로)



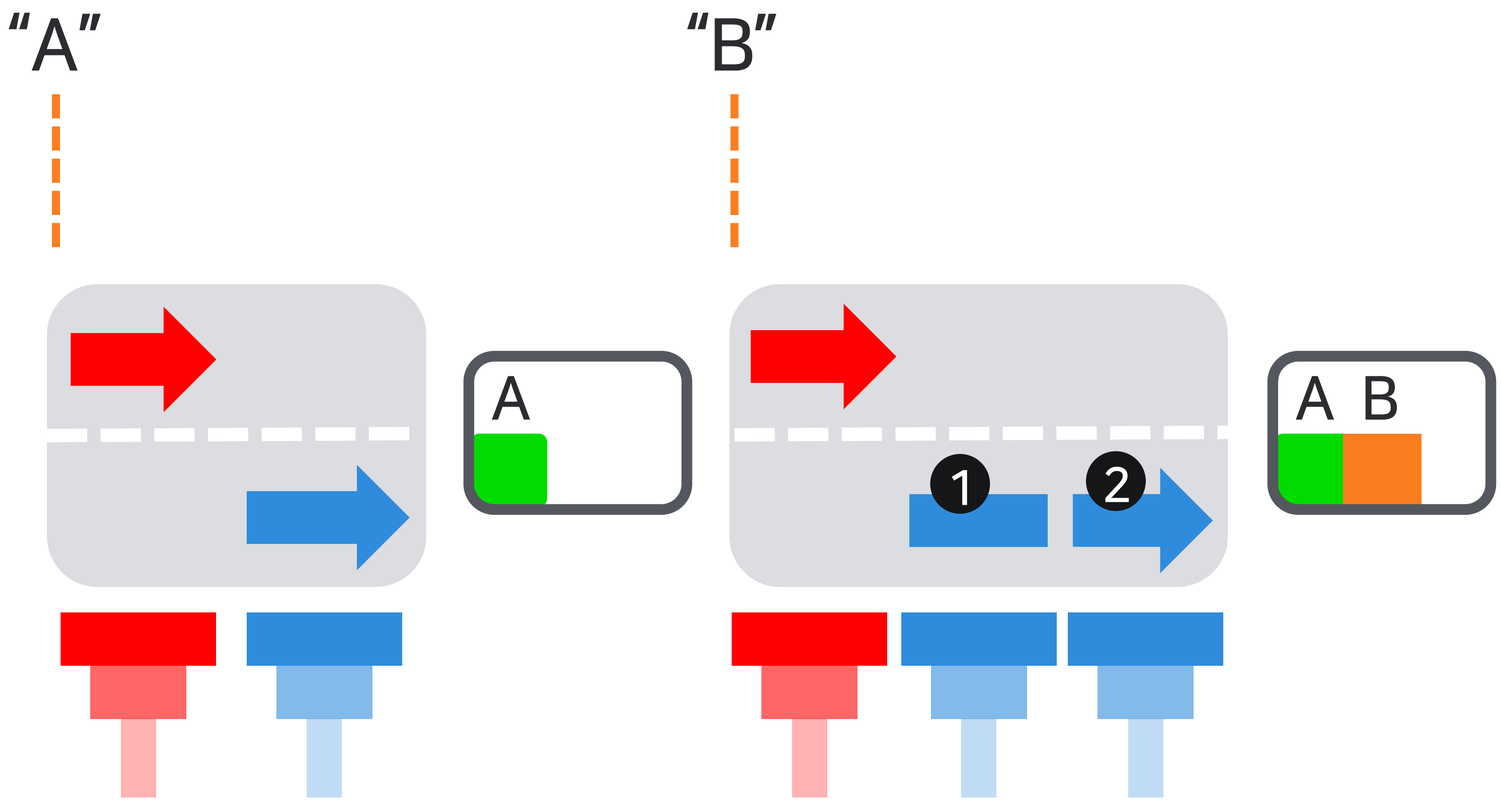
2.4 동시성 렌더링으로 해결해 보자

메인 스레드에 일정 시간을 양보



2.4 동시성 렌더링으로 해결해 보자

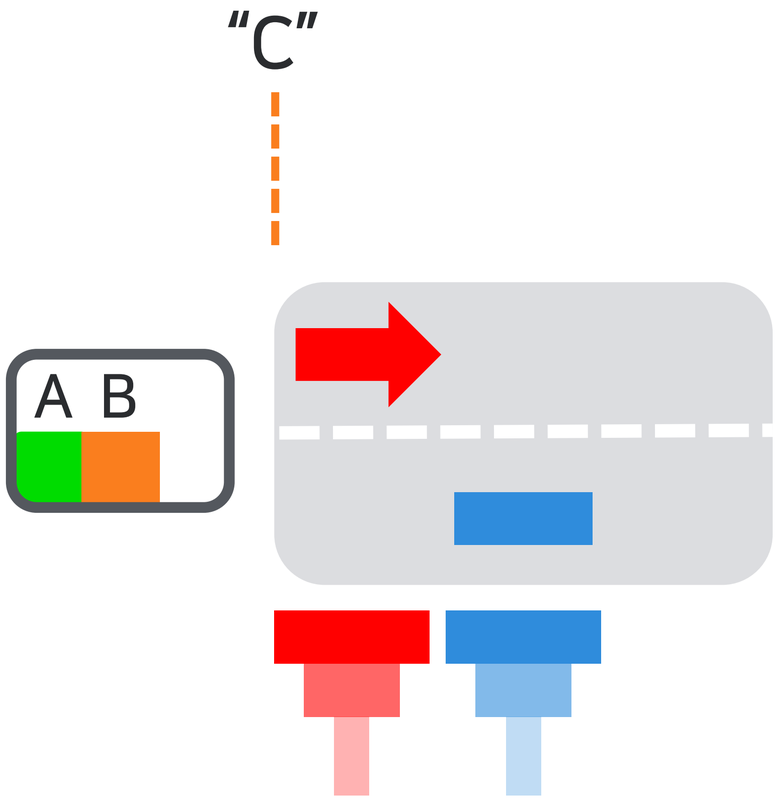
하위 컴포넌트 렌더링을 잘게 분해



2.4 동시성 렌더링으로 해결해 보자

DEVIEW 2021

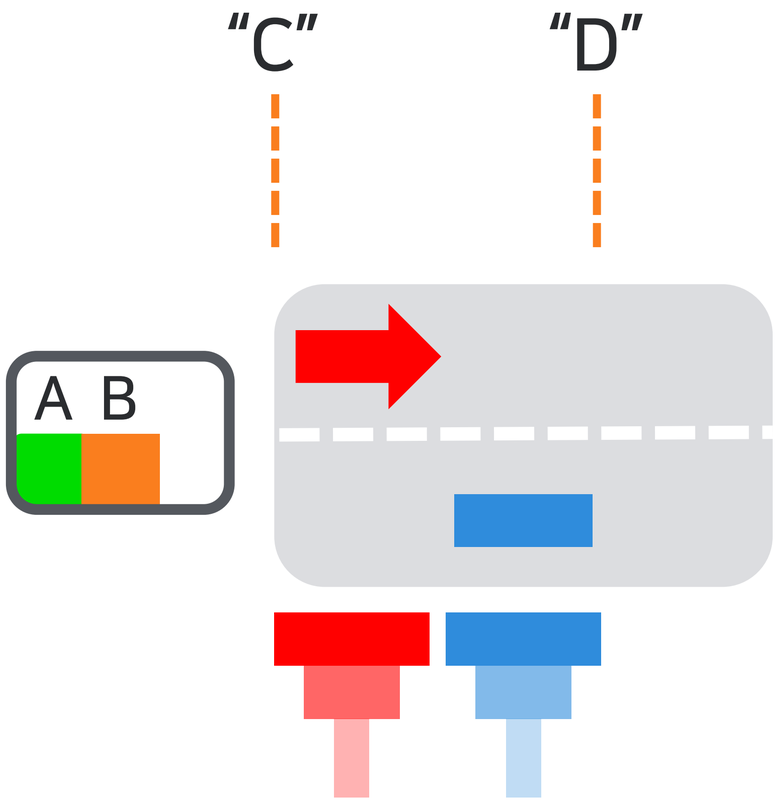
"C" 렌더링 도중에



2.4 동시성 렌더링으로 해결해 보자

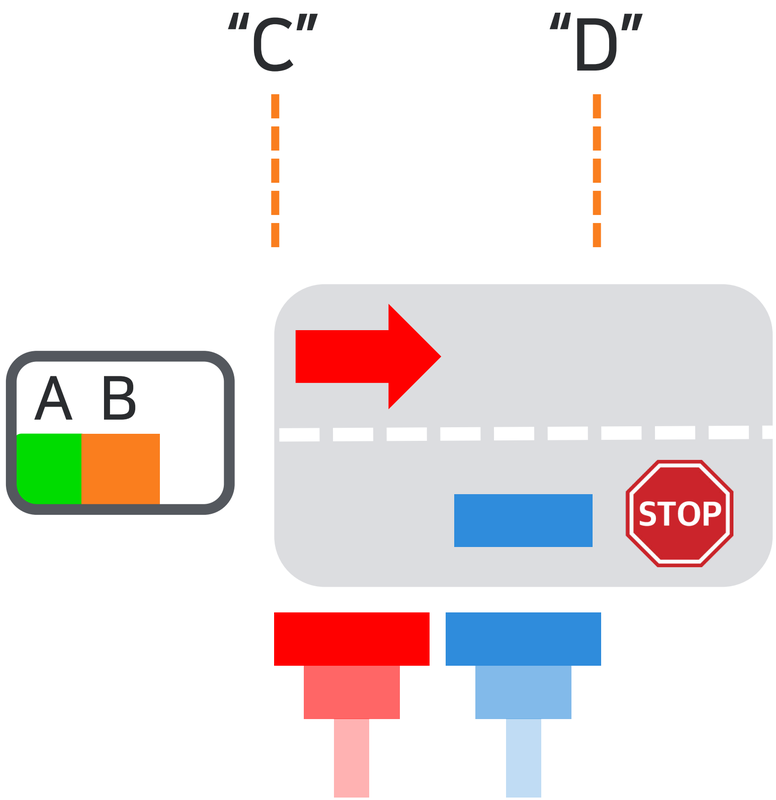
DEVIEW 2021

"C" 렌더링 도중에 "D" 입력 이벤트가 발생하면?



2.4 동시성 렌더링으로 해결해 보자

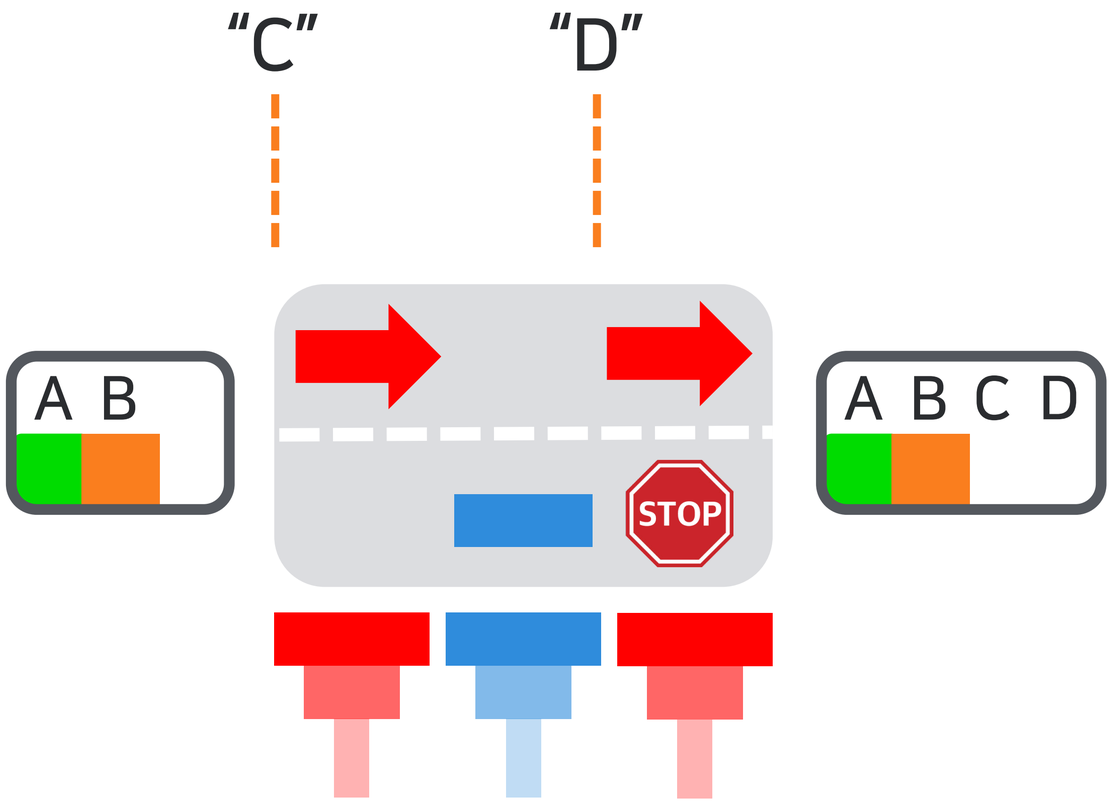
낮은 순위 렌더링을 멈추고



2.4 동시성 렌더링으로 해결해 보자

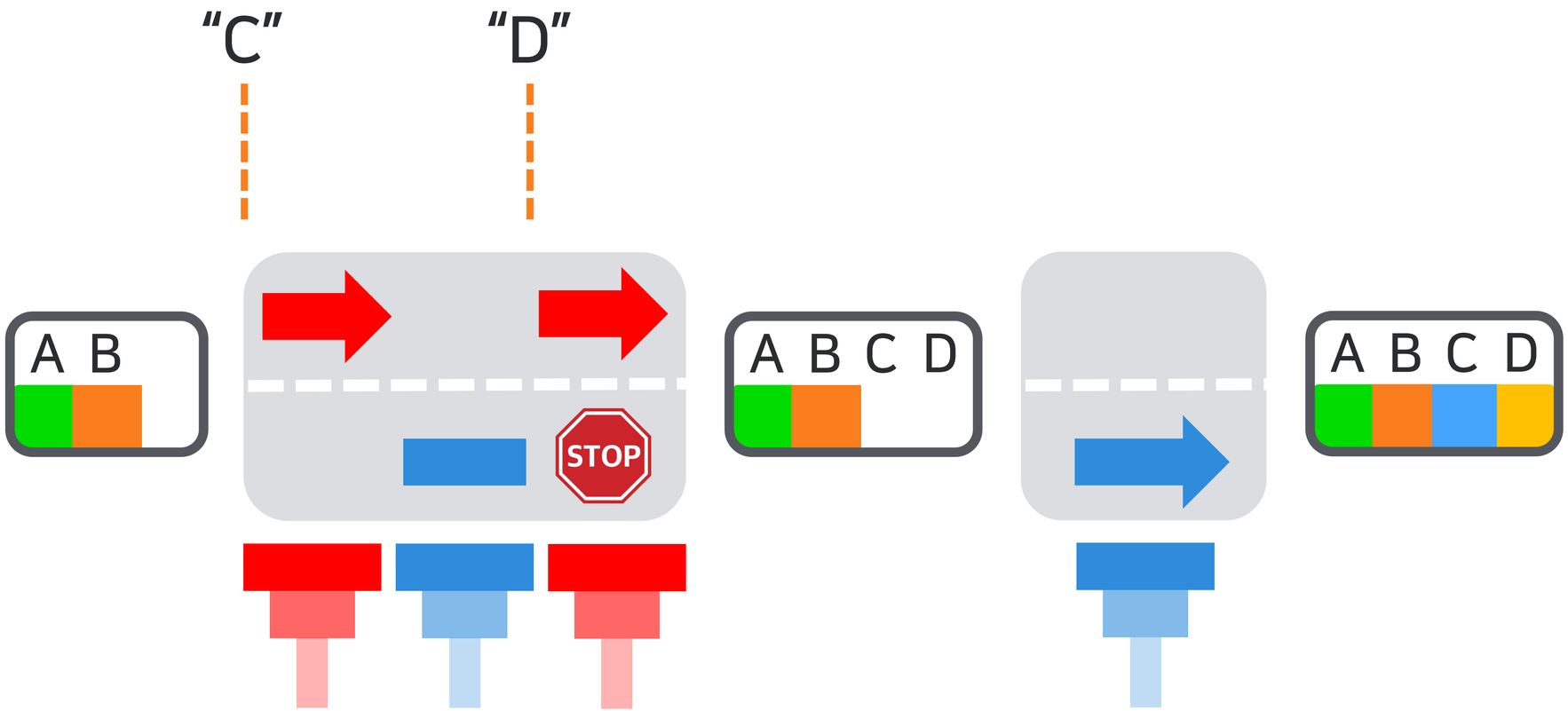
DEVIEW 2021

높은 순위 렌더링과 페인팅을 먼저 수행한다



2.4 동시성 렌더링으로 해결해 보자

그리고 Pending 상태의 낮은 순위 렌더링을 리베이스 한다



3.React 18의 동시성 기능

Concurrent Features in React 18

3.1 startTransition

API 및 내부 구현

```
// 높은 우선순위 업데이트 예약
setStateA(val1)

// 상태 업데이트를 전환으로 표시
startTransition(() => {
  // 낮은 우선순위 업데이트 예약
  setStateB(val2)
}
```

```
let isInTransition = false

function startTransition(fn) {
  isInTransition = true
  fn()
  isInTransition = false
}

function setState(value) {
  stateQueue.push({
    nextState: value,
    isTransition: isInTransition
  }
}
```

Pseudo Code

3.1 startTransition

DEVIEW 2021

무엇을 해결하는가

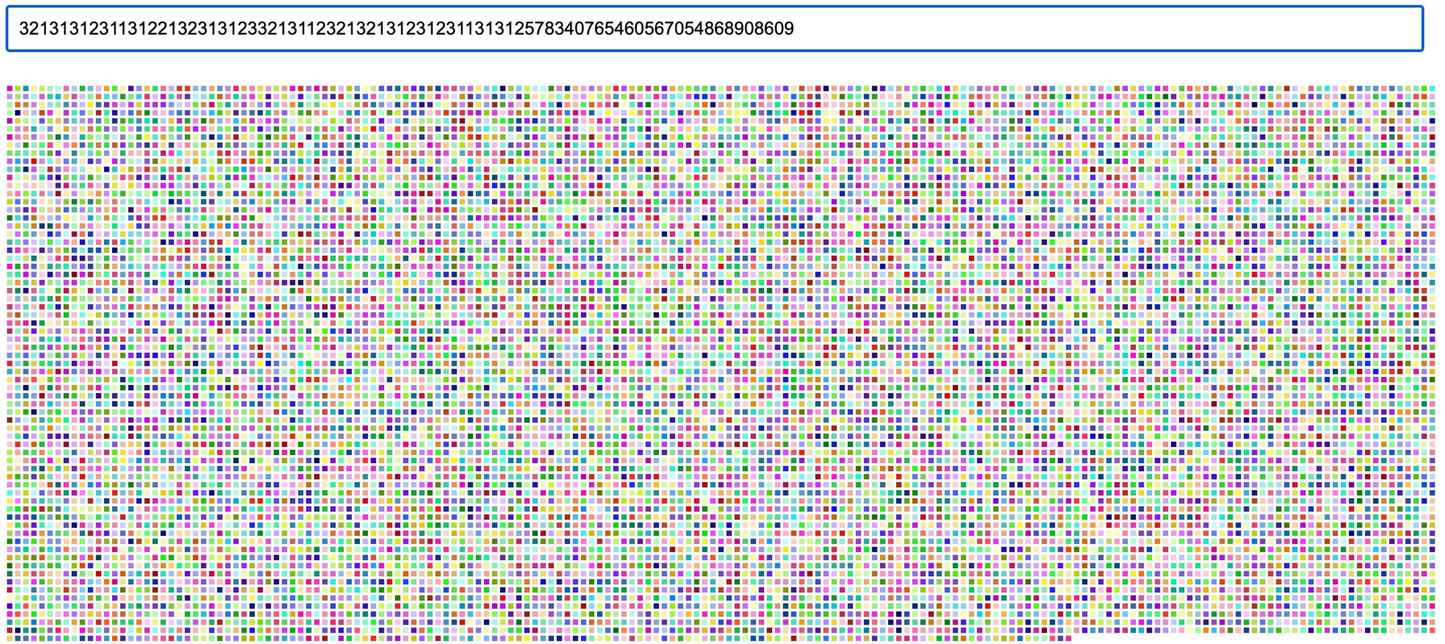
- CPU Bound UX 문제를 해결
- 대규모 화면 업데이트 중 응답성을 유지
- 상태 전환 중에 시각적 피드백을 제공

3.1 startTransition

DEVIEW 2021

```
const { useState, useTransition } = React;

function TextInput({ onChange }) {
  const [text, setText] = useState('');
  return (
    <input type="text" value={text}
      onChange={({target}) => {
        setText(target.value);
        onChange(target.value);
      }}
    />
  );
}

function App() {

  const [size, setSize] = useState(0);
  const [isPending, startTransition] = useTransition();

  function handleChange(text) {
    startTransition(() => {
      setSize(text.length);
    });
  };

  return (
    <div className='container'>
      <h1>Concurrent ({size})</h1>
      <TextInput onChange={handleChange} />
      <div className={isPending ? 'pending' : ''}>
        <ColorList length={size} />
      </div>
    </div>
  );
}
```

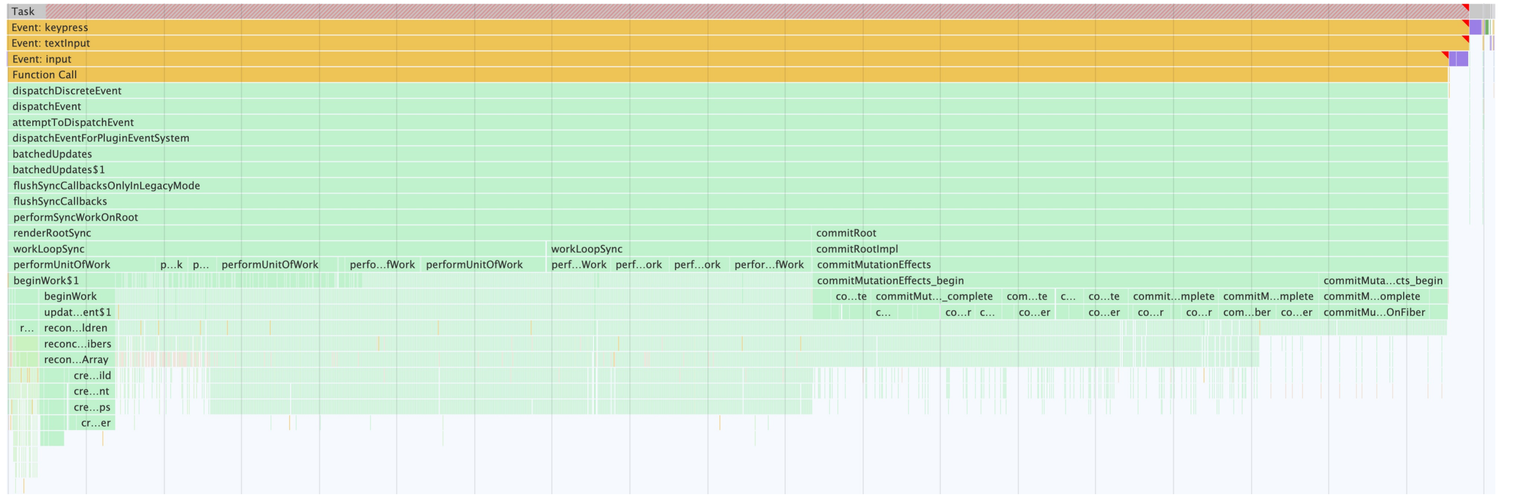
https://ajaxlab.github.io/deview2021/concurrent/



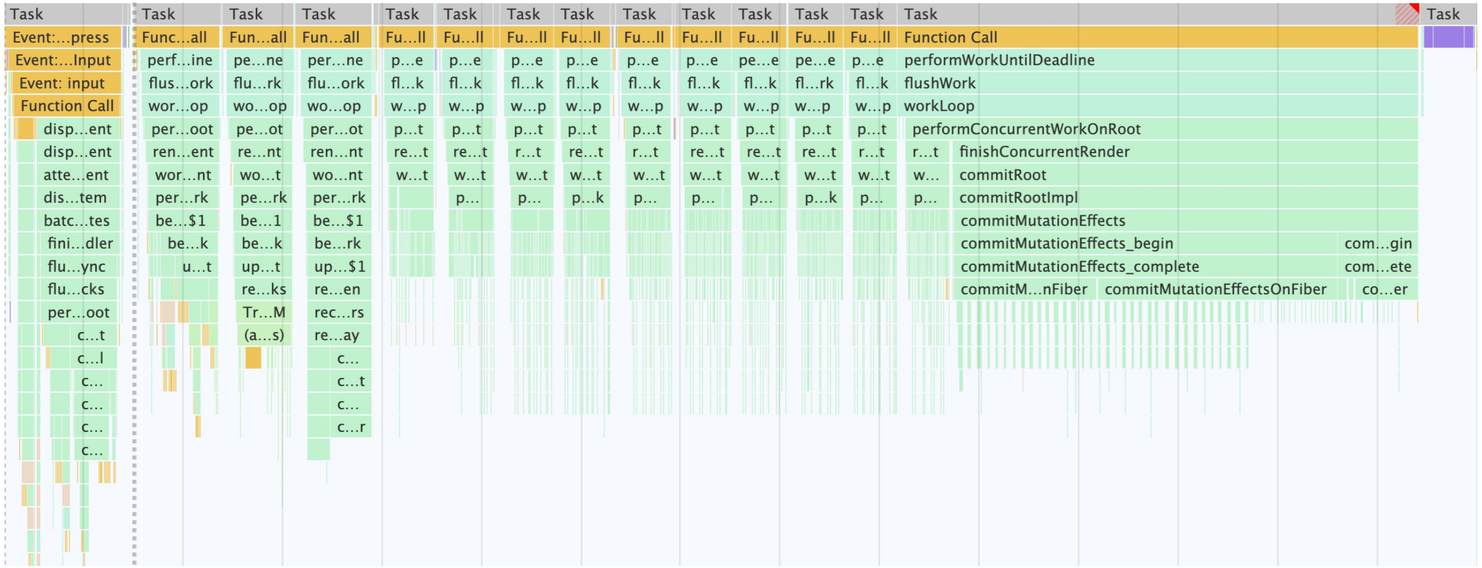
3.1 startTransition

DEVIEW 2021

Blocking



Concurrent



3.1 startTransition

어떻게 작동하는가

- Yielding : 랜더링 과정을 작게 분할하고 일시 중지할 수 있음
- Interrupting : 동시성 모드에서 업데이트에 대해 우선 순위가 있음
- 이전 결과 건너뛰기 : 현재 상태만 반영하도록 중간 상태 반영을 건너 뜀

setTimeout과 어떻게 다른가

- setTimeout으로 Task Queue에 들어간 작업은 들어간 순서대로 처리되고 취소될 수 없습니다.

https://github.com/reactwg/react-18/discussions/65

3.1 startTransition

DEVIEW 2021

Urgent Updates

- 입력, 클릭, 누르기 등과 같은 직접적인 상호 작용을 반영하기
- 즉각적 응답이 필요하다
- React 18은 업데이트를 기본적으로 urgent로 처리

Transition Updates

- 하나의 view에서 다른 view로 전환하기
- 전환되는 중간과정을 기대하지 않는다
- 결과에 시간이 소요되는 것을 예상한다
- Load Transition, Refresh Transition

3.1 startTransition

전환은 항상 일괄 처리

```
startTransition(() => {
  setDependent1(value)
  setDependent2(value)
})
```

```
startTransition(() => {
  setDependent1(value)
})
startTransition(() => {
  setDependent2(value)
})
```

짧은 순간의 업데이트는 마지막 값만 반영

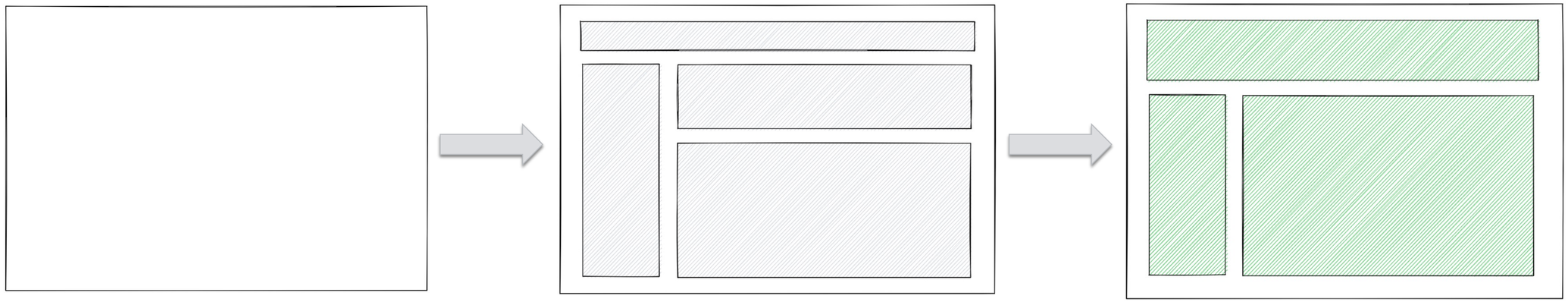
```
startTransition(() => {
  setSearchQuery("h");
})

// very soon...
startTransition(() => {
  setSearchQuery("he");
})

// very soon...
startTransition(() => {
  setSearchQuery("hello");
})
```

3.2 Streaming SSR with selective hydration

DEVIEW 2021

기존 SSR의 문제

- 어떤 것이라도 보여주기 위해 모든 data를 fetch해야 함
- 어떤 것이라도 hydrate하기 전에 필요한 모든 것을 로드해야 함
- 어떤 것이라도 상호작용하기 위해 모든 부분을 hydrate해야 함



All or Nothing

3.2 Streaming SSR with selective hydration

React 18에서 잠금 해제된 두 가지 기능

- Server : Streaming HTML
  - 기존의 renderToString() 대신 pipeToNodeWritable() 사용
- Client : Selective Hydration
  - createRoot()와 용량이 크거나 처리가 느린 부분을 Suspense로 감싸기

3.2 Streaming SSR with selective hydration

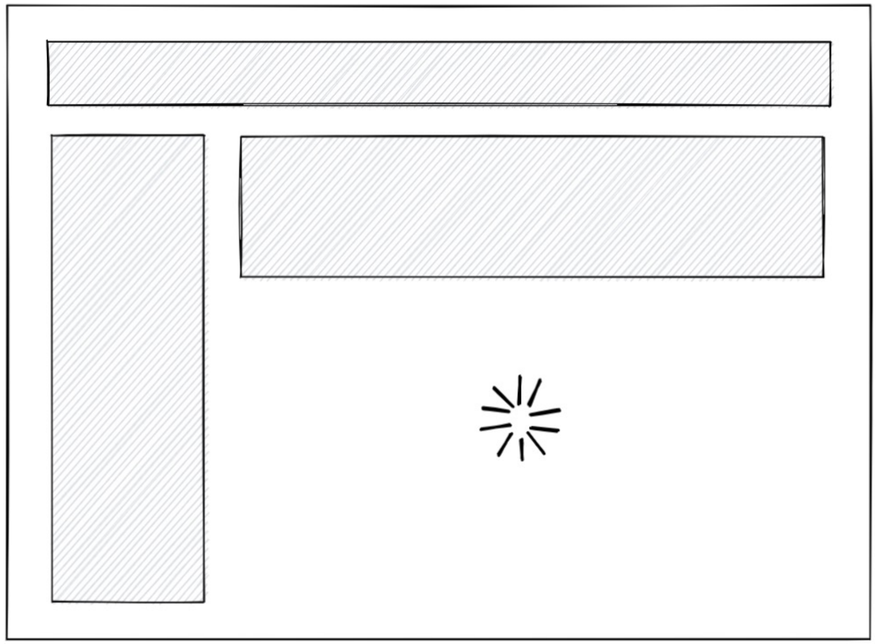
DEVIEW 2021

Case 1. 필요한 데이터를 모두 불러오기 전에 HTML 스트리밍하기

```
<Layout>
  <NavBar />
  <Sidebar />
  <RightPane>
    <Post />
    <Suspense fallback={<Spinner />}>
      <Comments />
    </Suspense>
  </RightPane>
</Layout>
```

특정 컴포넌트를 Suspense로 감싸면

해당 부분의 데이터 패칭을 기다리지 않는다



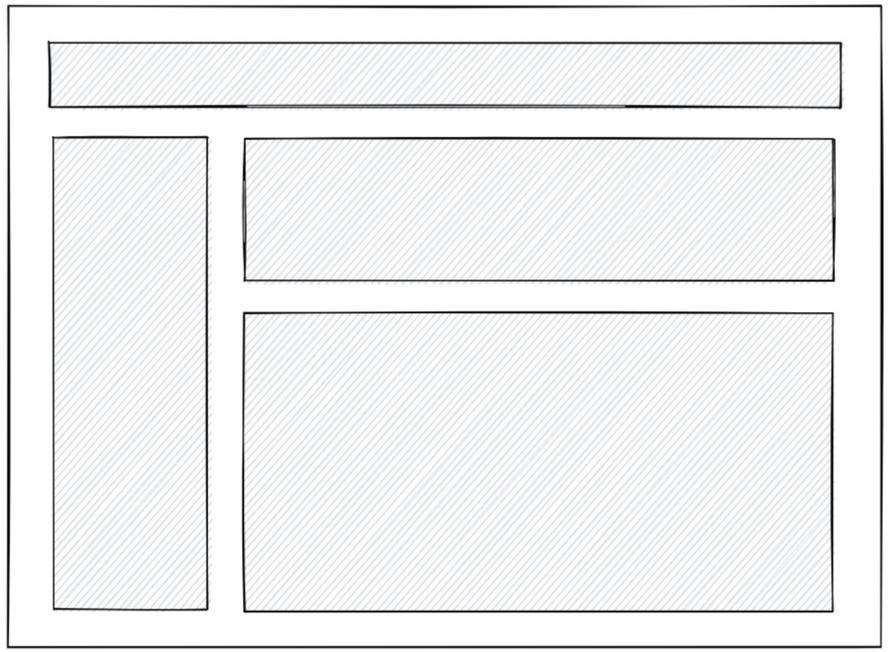
3.2 Streaming SSR with selective hydration

DEVIEW 2021

Case 1. 필요한 데이터를 모두 불러오기 전에 HTML 스트리밍하기

```
<div hidden id="comments">
  <!-- Comments -->
  <p>First comment</p>
  <p>Second comment</p>
</div>
<script>
  // simplified implementation
  document.getElementById('comments-spinner')
    .replaceChildren(
      document.getElementById('comments')
    );
</script>
```

data fetch가 끝나면 약간의 script 조각과 함께 HTML Stream 으로 전송한다



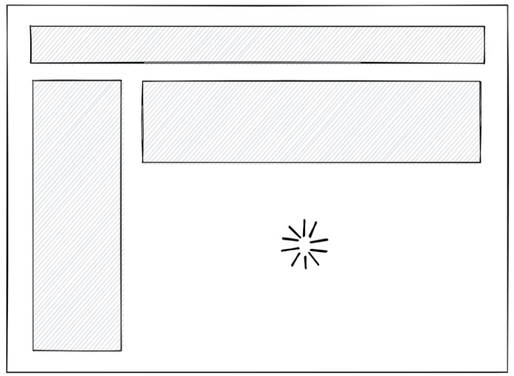
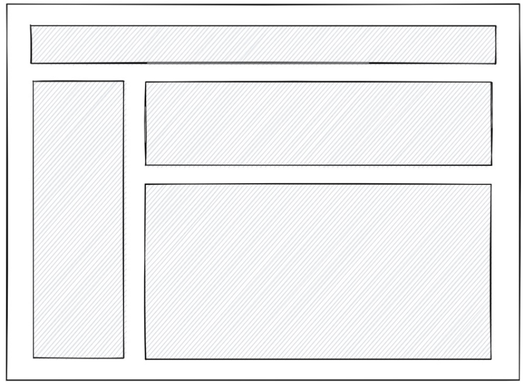
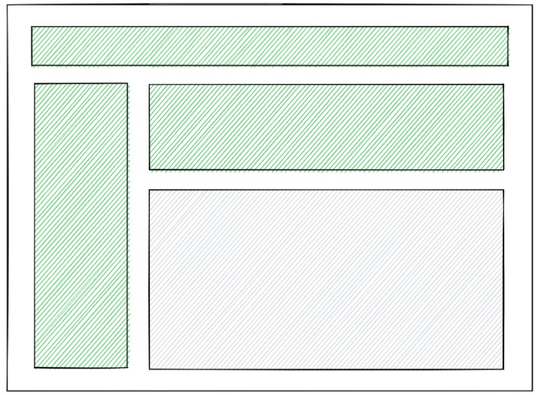
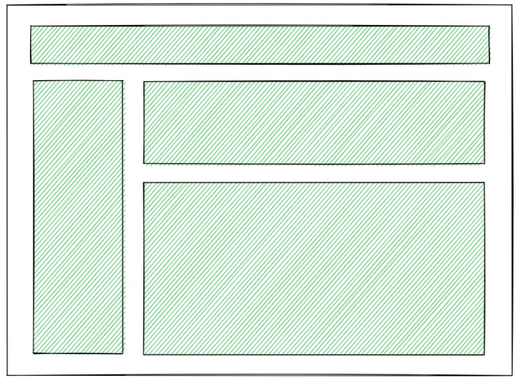
3.2 Streaming SSR with selective hydration

DEVIEW 2021

Case 2. React.lazy와 함께 모든 코드가 로드되기 전에 Hydrate

```
import { lazy } from 'react';

const Comments = lazy(() => import('./Comments.js'));

// ...

<Suspense fallback={<Spinner />}>
  <Comments />
</Suspense>
```

Suspense로 감싸면 스트림에서
다른 부분을 블록하지 않게 됩니다.
이것이 선택적 하이드레이션 입니다.

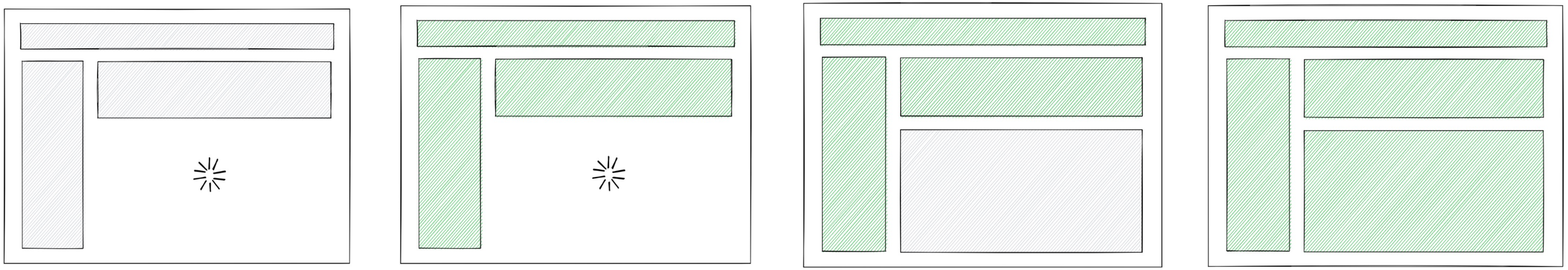
3.2 Streaming SSR with selective hydration

DEVIEW 2021

Case 3. 모든 HTML이 스트리밍되기 전에 하이드레이션

JavaScript 코드가 모든 HTML보다 먼저 로드되면 기다릴 이유가 없습니다.

페이지의 나머지 부분을 하이드레이션 합니다.

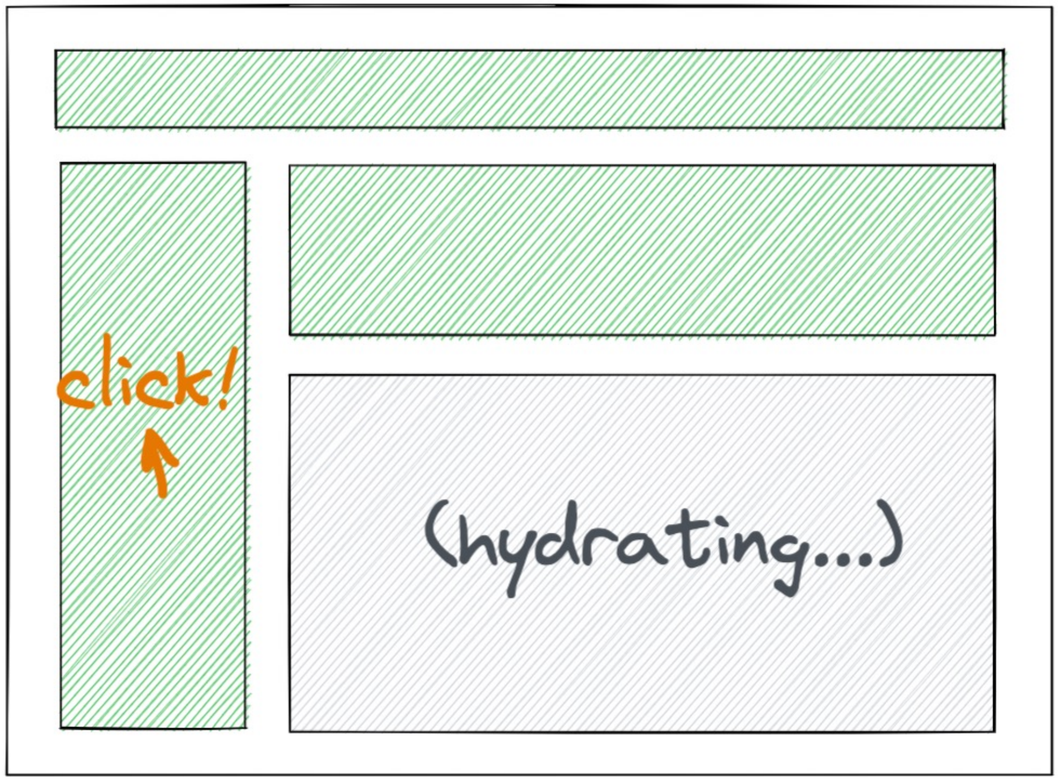


3.2 Streaming SSR with selective hydration

DEVIEW 2021

Case 4. 모든 컴포넌트가 하이드레이션되기 전에 상호 작용

React 18에서 하이드레이션은
매우 작은 간격으로 실행되므로
메인 스레드를 블록하지 않음



3.2 Streaming SSR with selective hydration

DEVIEW 2021

Case 5. 하이드레이션 우선 순위 조정

React는 가능한 한 빨리 모든 것을 하이드레이션하기 시작하고

사용자 상호 작용을 기반으로 화면에서 가장 긴급한 부분의 우선 순위를 지정합니다

https://9hfqt.sse.codesandbox.io/

https://codesandbox.io/s/festive-star-9hfqt?file=/src/App.js

3.3 React 18의 Suspense 변화

DEVIEW 2021

추가될 기능

- Show old data while refetching pattern : startTransition과 함께 사용
- Built-in throttling : 스피너가 너무 자주 나타나지 않도록 조절

추가될 수도 있는 기능

- Backup suspense boundaries : 초기 렌더링 중에 경계 안쪽이 없는 것 처럼 무시되도록 하는 방법
- CPU-Bound tree를 위한 Suspense : I/O 측면이 아닌 CPU 측면에서 무리한 연산시 Placeholder 표시

추가되지 않는 기능

- 18에 도입되는 사항은 기반 아키텍쳐의 개선이고 data fetching 같은 전략은 제시하지 않는다
- react-fetch (I/O library), Built-in Suspense Cache, Server Components 등은 18.x 에서 예정

https://github.com/reactwg/react-18/discussions/47

3.4 Automatic Batching

DEVIEW 2021

자동 배칭이란

- React 17이전에 상태 업데이트는 이벤트 핸들러 안에서만 배치로 작동하였다 (demo)
- React 18이후 createRoot() API를 사용하면 모든 배치는 자동 배치로 작동한다 (demo)

```
function handleClick() {
  setCount(c => c + 1)
  setFlag(f => !f)
}
```

```
fetch(/*...*/).then(() => {
  setCount(c => c + 1)
  setFlag(f => !f)
})
```

```
setTimeout(() => {
  setCount(c => c + 1)
  setFlag(f => !f)
}, 1000)
```

```
elm.addEventListener('click', () => {
  setCount(c => c + 1)
  setFlag(f => !f)
})
```

```
import { flushSync } from 'react-dom';

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

https://github.com/reactwg/react-18/discussions/21

3.5 Concurrent 모드 기능에 대한 공통 주제

HCI에 대한 연구 결과가 실제 UI와 통합되도록 돕는 것

- 화면 간 전환에서 로딩 중 상태를 너무 많이 표시하면 UX 품질이 낮아짐
- 빠르게 처리되기 기대하는 상호작용과 느려도 문제 없는 상호작용
- 동시성 모드의 목적은 HCI 연구 결과를 추상화하고 구현할 수 있는 방법을 제공하는 것

```
// The most recent time we committed a fallback.
// This lets us ensure a train
// model where we don't commit new loading states
// in too quick succession.
let globalMostRecentFallbackTime: number = 0;
const FALLBACK_THROTTLE_MS: number = 500;

// The absolute time for when we should start
// giving up on rendering more and prefer
// CPU suspense heuristics instead.
let workInProgressRootRenderTargetTime: number = Infinity;
// How long a render is supposed to take before
// we start following CPU suspense heuristics and
// opt out of rendering more content.
const RENDER_TIMEOUT_MS = 500;
```

CPU Suspense Heuristics

```
// Computes the next Just Noticeable Difference (JND) boundary.
function jnd(timeElapsed: number) {
  return timeElapsed < 120
    ? 120
    : timeElapsed < 480
    ? 480
    : timeElapsed < 1080
    ? 1080
    : timeElapsed < 1920
    ? 1920
    : timeElapsed < 3000
    ? 3000
    : timeElapsed < 4320
    ? 4320
    : ceil(timeElapsed / 1960) * 1960;
}
```

Just Noticeable Difference (JND) boundary

DEVIEW 2021

4. React 동시성의 기반 기술

Under the Hood of Concurrency in React

4장에서 소개하는 내용의 멘탈 모델

DEVIEW 2021

```
function workLoop() {
  // 시간을 사용하고 메인 스레드에게 돌려주자
  while (workInProgress && !shouldYield()) {
    // 주어진 짧은 시간 동안 가상 DOM을 현행화하자
    workInProgress = performUnitOfWork(workInProgress)
  }
  // DOM에 반영하자
  if (!workInProgress && rootToCommit) {
    commitRoot()
  }
}
```

```
function performUnitOfWork(work) {
  const { children } = work.props
  reconcileChildren(work, children)
  // 다음 작업을 얻는 방법 필요
  return getNextWork(work)
}
```

- 스레드를 블록하지 않는 루프가 필요
- 메인 스레드에게 양보하기 위한 방법 필요

```
function reconcileChildren(fiber, children) {
  // 모든 자식을 순회하되 일시정지 할 수 있어야 한다
  iterate(children, (child) => {
    const status = compareToOldWork(child)
    switch (status) {
      case 'toAdd':
        fiber.child = createFiber('PLACEMENT')
      case 'toUpdate':
        fiber.child = createFiber('UPDATE')
      case 'toRemove':
        deletions.push(child);
    }
  });
}
```

- 컴포넌트 트리 순회할 때 블로킹을 방지할 방법

4.1 React Packages

DEVIEW 2021

개발 및 빌드 환경

- 개발 언어 : Flow, 빌드 도구 : Rollup, Babel
- 그리고 Google Closure Compiler
  - Better eliminate dead code
  - Automatically inline small functions
- 빌드 결과는 umd, node, rn, fb_www

```
yarn run build react/index,react-dom/index --type=UMD
```

```
\home\react>yarn build react/index,react-dom/index --type=UMD
yarn run v1.22.11
$ node ./scripts/rollup/build.js react/index,react-dom/index --type=UMD
BUILDING  react.development.js (umd_dev)
COMPLETE  react.development.js (umd_dev)

BUILDING  react.production.min.js (umd_prod)
COMPLETE  react.production.min.js (umd_prod)

BUILDING  react.profiling.min.js (umd_profiling)
COMPLETE  react.profiling.min.js (umd_profiling)

BUILDING  react-dom.development.js (umd_dev)
COMPLETE  react-dom.development.js (umd_dev)

BUILDING  react-dom.production.min.js (umd_prod)
COMPLETE  react-dom.production.min.js (umd_prod)

BUILDING  react-dom.profiling.min.js (umd_profiling)
COMPLETE  react-dom.profiling.min.js (umd_profiling)
```

Bundle	Prev Size	Current Size	Diff	Prev Gzip	Current Gzip	Diff
react.development.js (UMD_DEV)	0 B	111.21 KB	n/a	0 B	28.52 KB	n/a
react.production.min.js (UMD_PROD)	0 B	11.52 KB	n/a	0 B	4.5 KB	n/a
react.profiling.min.js (UMD_PROFILING)	0 B	11.52 KB	n/a	0 B	4.5 KB	n/a
react-dom.development.js (UMD_DEV)	0 B	1.01 MB	n/a	0 B	224.33 KB	n/a
react-dom.production.min.js (UMD_PROD)	0 B	129.76 KB	n/a	0 B	41.95 KB	n/a
react-dom.profiling.min.js (UMD_PROFILING)	0 B	137.44 KB	n/a	0 B	44.33 KB	n/a

```
Done in 47.63s.
```

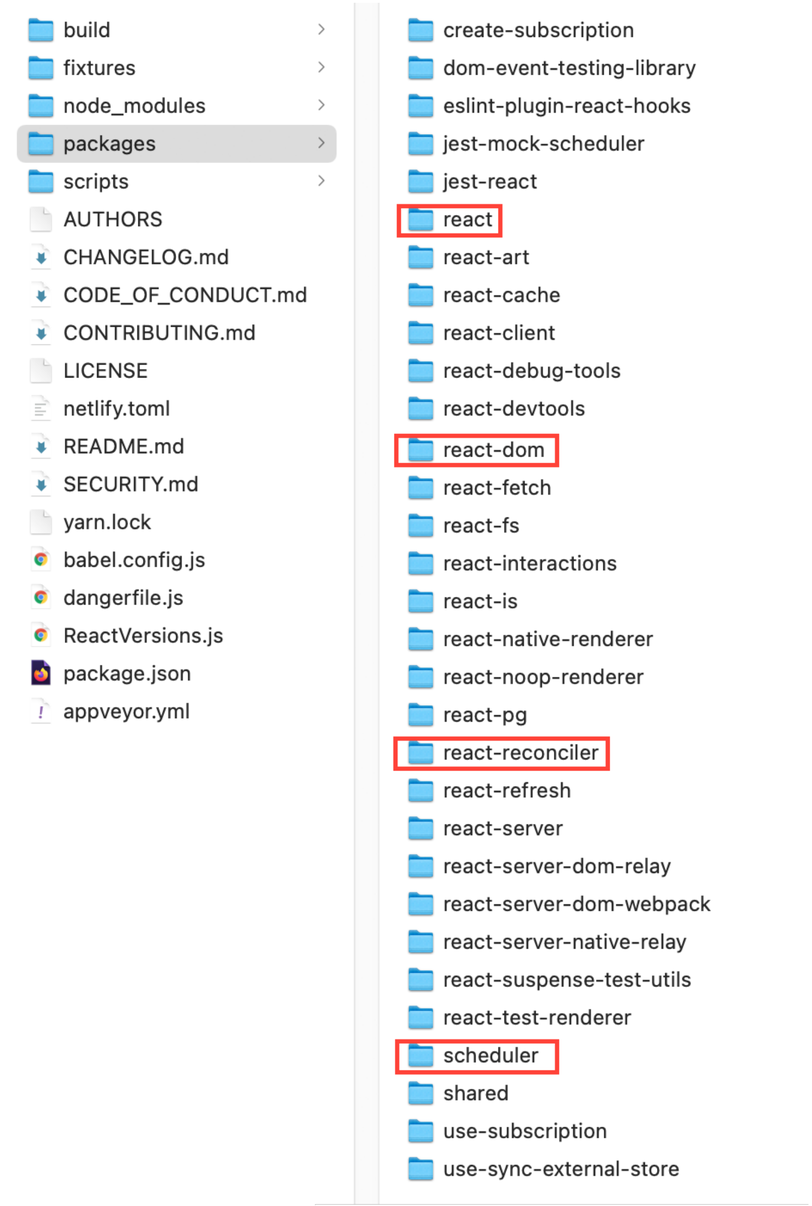
4.1 React Packages

Package 구조

- Yarn workspace 기반의 Monorepo
- /fixtures : 컨트리뷰터를 위한 테스트용 예제들

/packages

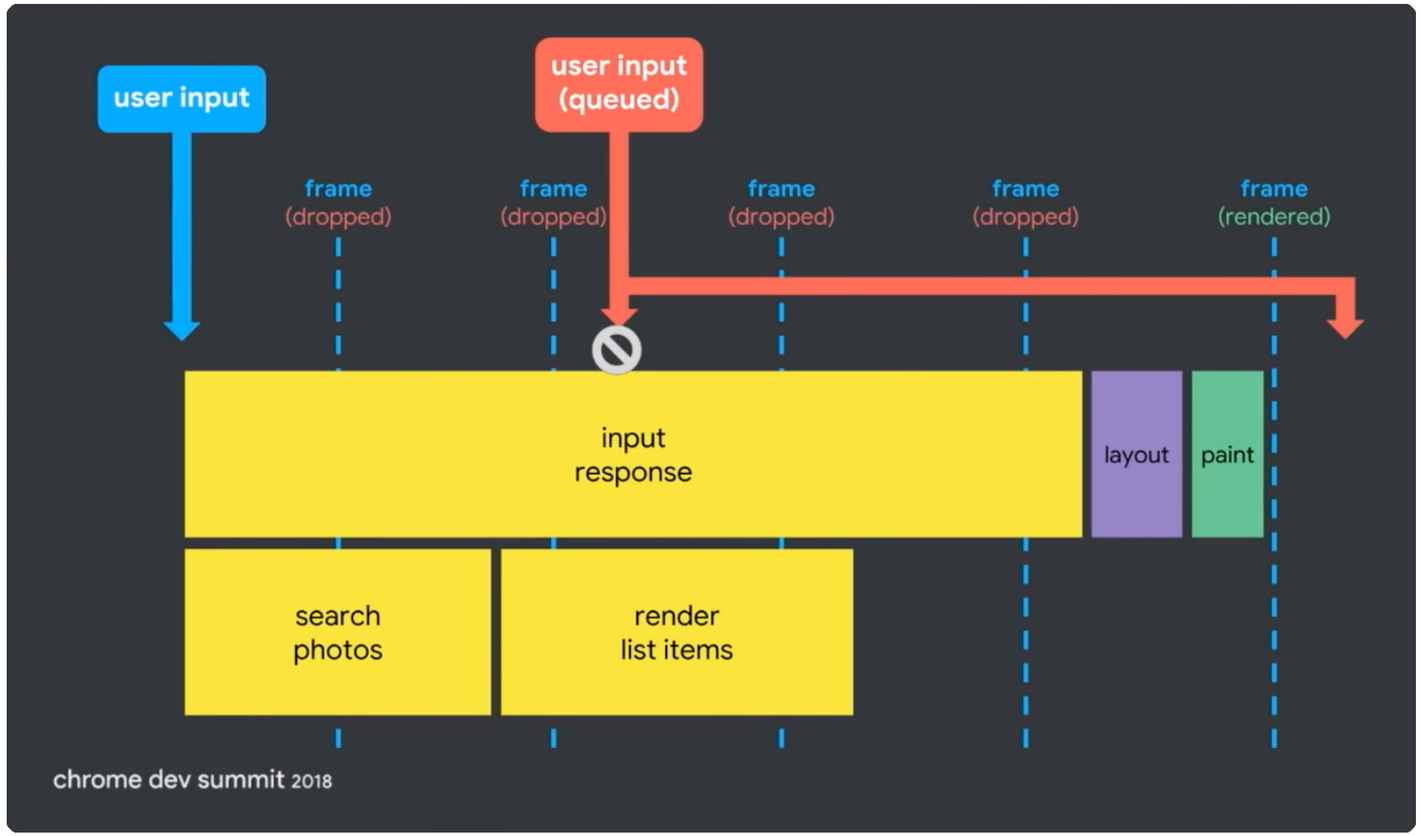
- React APIs : /react
- Scheduler : /scheduler
- Reconciler : /react-reconciler
- Renderers : /react-dom, /react-native-..., /react-noop- …
- Share : /share
- Experimental : /react-cache, /react-server, /react-client …



4.2 동시성 모델과 이벤트 루프

DEVIEW 2021

Stack Reconciler의 Blocking Rendering



https://www.youtube.com/watch?v=mDdgfyRB5kg&t=94s

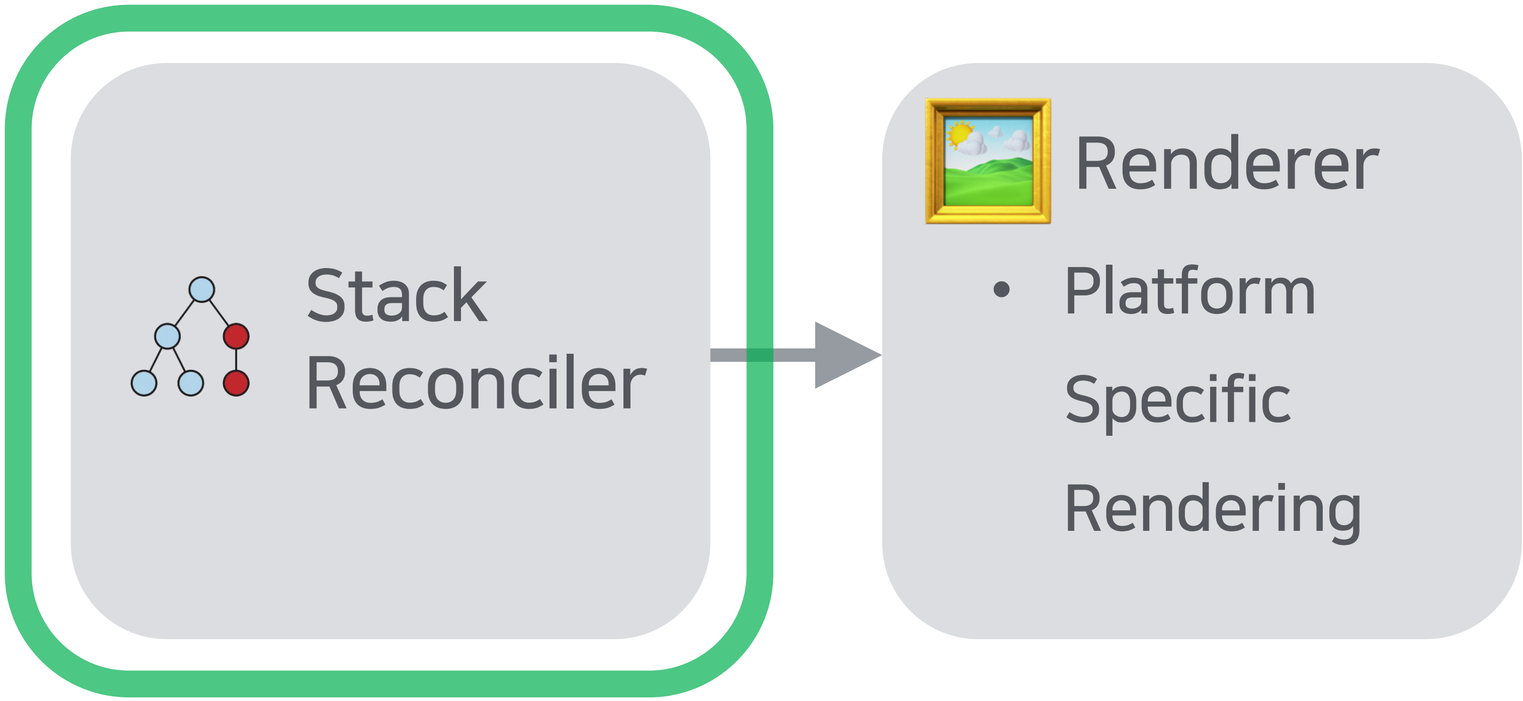
```
function mountHost(element) {
  var type = element.type;
  var props = element.props;
  var children = props.children || [];

  // Mount the children
  children.forEach(childElement => {
    var childNode = mount(childElement);
    node.appendChild(childNode);
  });

  return node;
}

function mount(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // Class components
    return mountComposite(element);
  } else if (typeof type === 'string') {
    // Host components
    return mountHost(element);
  }
}
```

https://ko.reactjs.org/docs/implementation-notes.html

4.2 동시성 모델과 이벤트 루프

DEVIEW 2021

React 15.x의 두 개의 레이어



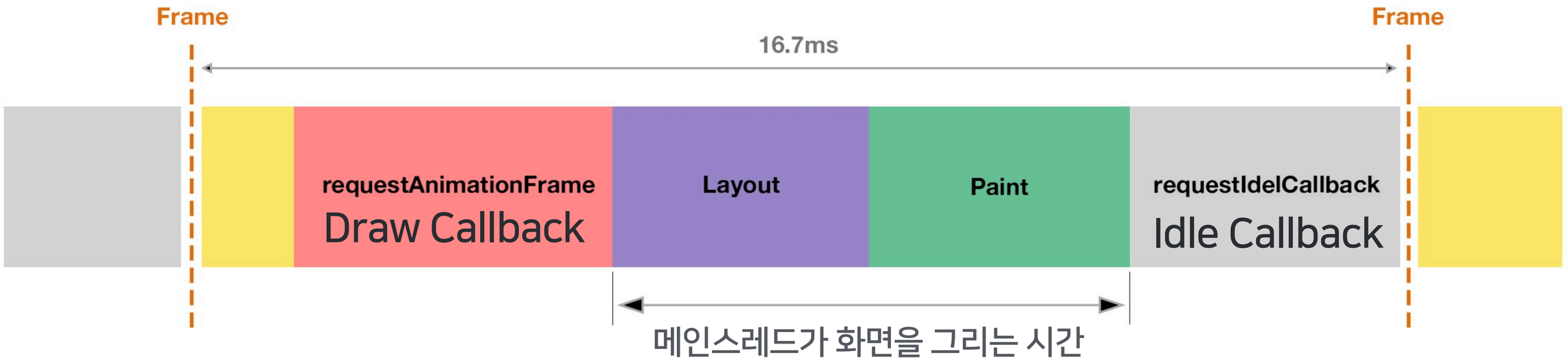
JavaScript로 개입할 수 있는 부분

4.2 동시성 모델과 이벤트 루프

DEVIEW 2021

requestAnimationFrame

- 레이아웃 계산 전에 dom을 수정할 때 유용
- 레이아웃까지 남은 시간을 알 수 없어 연산을 일시 정지하는데 한계

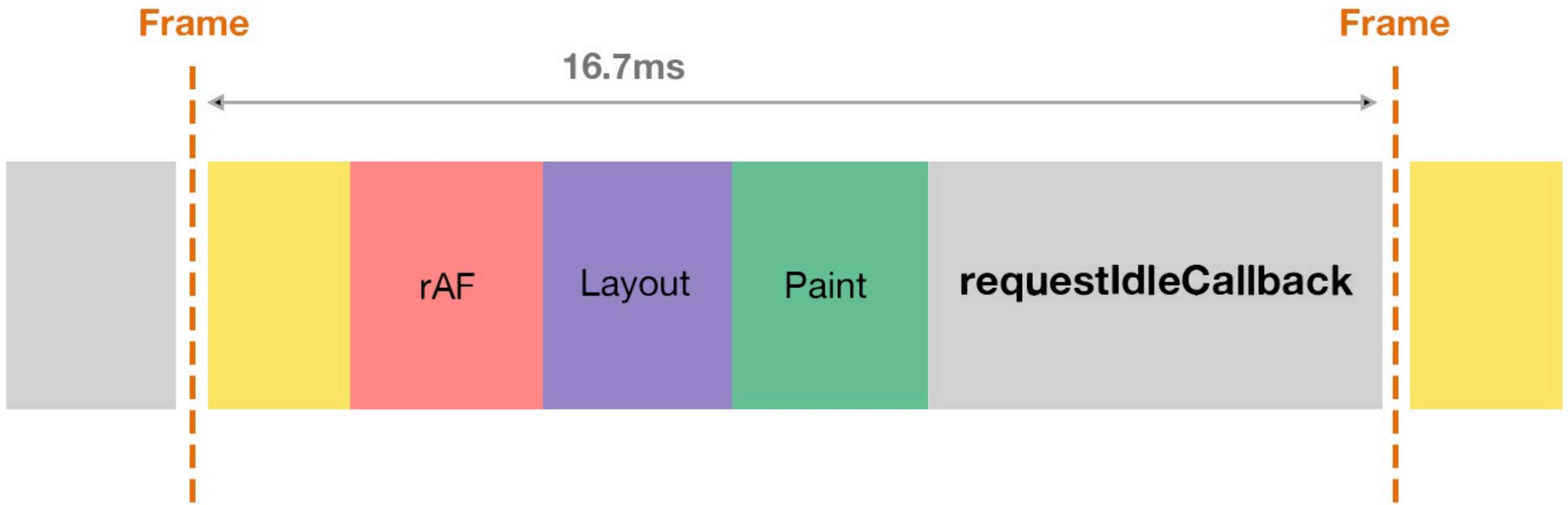


4.2 동시성 모델과 이벤트 루프

DEVIEW 2021

requestIdleCallback

- 급한 일 먼저 하시고 쉬는 시간에 요것 좀 해주세요
- 다음 급한 일까지 얼마 남았는지 좀 알려주세요



4.2 동시성 모델과 이벤트 루프

requestIdleCallback

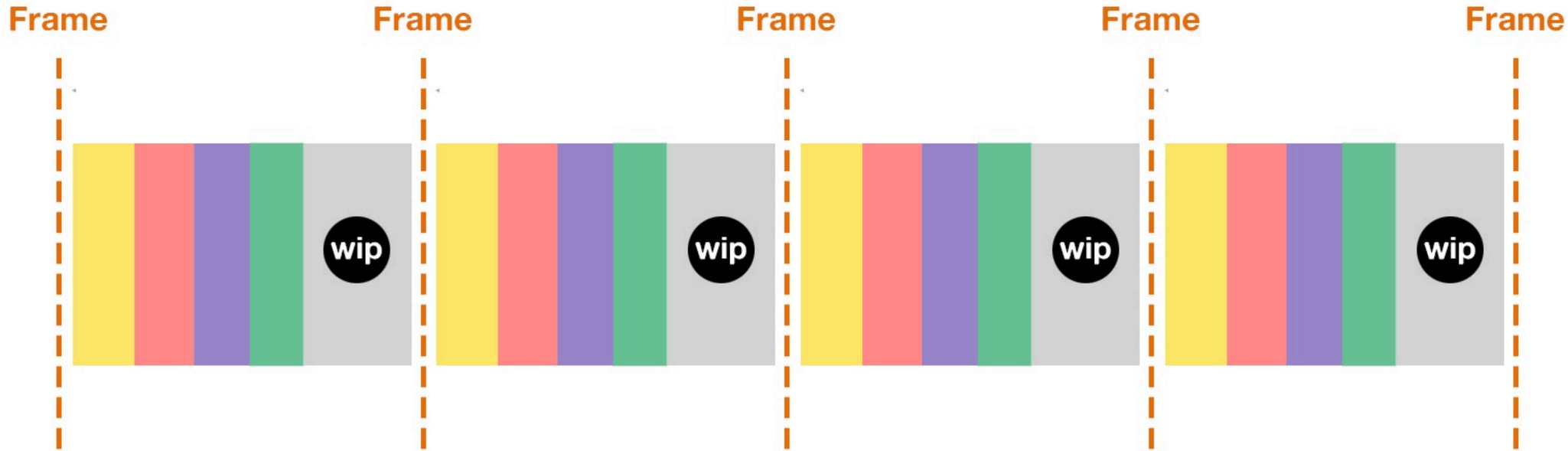
- jobLoop을 예약
- jobLoop에서는 메인 스레드 유휴 시간이 있으면 작업을 반복 수행
- 다음 유휴 시간을 예약

```
let job = null

function jobLoop(deadline) {
  // 할일이 있고 브라우저가 idle 상태이면
  while (job && deadline.timeRemaining() > 0) {
    // job을 수행하고 다음 job을 할당하라
    job = performJob(job)
  }
  // 다음번 idle 상태에 jobLoop을 다시 수행하라
  // 그때까지 브라우저는 더 중요한 일을 처리
  requestIdleCallback(jobLoop)
}

function performJob(job) {
  console.info(job)
  if (job && job.val < 10) {
    return { val: job.val + 1 }
  }
  return null
}

// 유휴시간이 되면 jobLoop을 실행하라
requestIdleCallback(jobLoop)
```

4.2 동시성 모델과 이벤트 루프

requestIdleCallback

- 메인 스레드의 동작이 블록되지 않도록
- 렌더링 작업을 작은 단위(Chunk)로 나누고
- 여러 타임 프레임에 걸쳐 분산하여 렌더링



4.2 동시성 모델과 이벤트 루프

DEVIEW 2021

requestIdleCallback의 문제

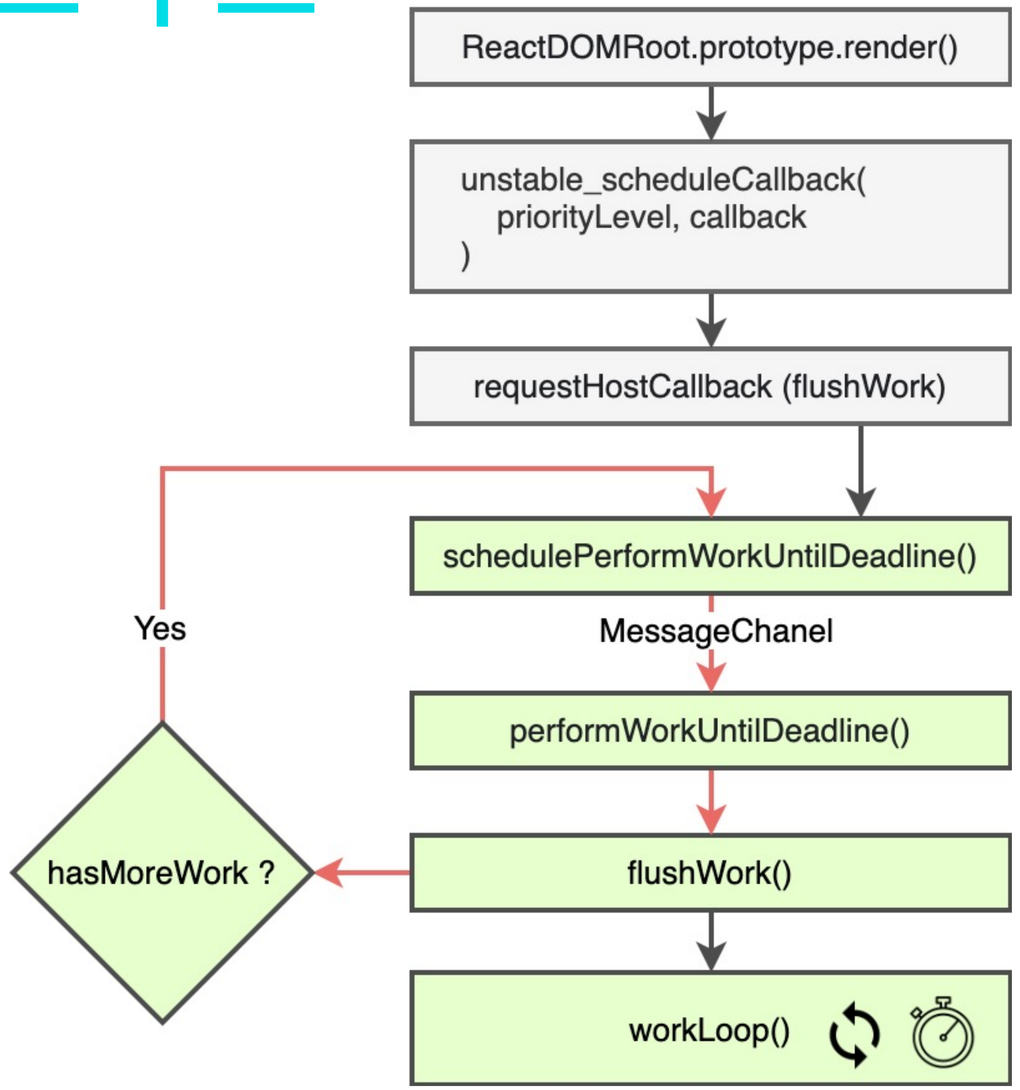
- Safari, IE 지원 안함
  - Cooperative Scheduling of Background Tasks (W3C)
- Callback 호출 주기가 불안정
- 브라우저 탭 전환 시 비활성 탭의 호출 주기가 매우 낮아짐

https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallback

4.2 동시성 모델과 이벤트 루프

Scheduler Package 탄생

- Cooperative Scheduling of Background Tasks
  - 개념상 requestIdleCallback
- 일정한 주기로 메인스레드 양보 (5ms)
- MessageChannel API 기반
- 작업 우선 순위 지원 (Heap)



https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel

4.2 동시성 모델과 이벤트 루프

DEVIEW 2021

Pseudo Event Loop Code

```
const performWorkUntilDeadline = () => {
  startTime = performance.now();
  currentTask = flushWork(currentTask)
  if (currentTask) {
    schedulePerformWorkUntilDeadline()
  }
}

const channel = new MessageChannel()
const port = channel.port2
channel.port1.onmessage = performWorkUntilDeadline
schedulePerformWorkUntilDeadline = () => {
  port.postMessage(null)
}
```

```
const queue = []
const frameInterval = 5
let currentTask = null
let startTime = -1

function shouldYield() {
  const timeElapsed = performance.now() - startTime
  return timeElapsed > frameInterval
}

function flushWork(currentTask) {
  let count = 0
  while (currentTask && !shouldYield()) {
    // 스케줄러가 시간을 가져온 상태
    // 여기서 무언가 작업을 처리한다
    count++
  }
  return queue.shift()
}
```

https://ajaxlab.github.io/deview2021/eventloop/

4.2 동시성 모델과 이벤트 루프

workLoopSync

```
function workLoopSync() {
  // Already timed out,
  // so perform work
  // without checking
  // if we need to yield.
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
```

workLoopConcurrent

```
function workLoopConcurrent() {
  // Perform work until
  // Scheduler asks us
  // to yield
  while (workInProgress !== null
         && !shouldYield()) {
    performUnitOfWork(workInProgress);
  }
}
```

4.2 동시성 모델과 이벤트 루프

DEVIEW 2021

isInputPending()

```
while (workQueue.length > 0) {
  if (navigator.scheduling.isInputPending([
    'mousedown', 'mouseup', 'keydown', 'keyup'
  ])) {
    // Stop doing work if we think we'll
    // start receiving a mouse or key event.
    break;
  }
  let job = workQueue.shift();
  job.execute();
}
```

scheduler.postTask()

```
async function runExample() {
  let resultPromise = scheduler.postTask(() => {
    appendToContent('h3', 'Scheduling Tasks');
  });
  await resultPromise;

  let results = [];

  // Valid priorities, from highest to lowest, are:
  // 'user-blocking', 'user-visible', 'background'.
  results.push(scheduler.postTask(() => 'line 3', {priority: 'background'}));
  results.push(scheduler.postTask(() => 'line 2', {priority: 'user-visible'}));
  results.push(scheduler.postTask(() => 'line 1', {priority: 'user-blocking'}));

  results.forEach((promise) => {
    promise.then((result) => appendToContent('div', result));
  });
}
```

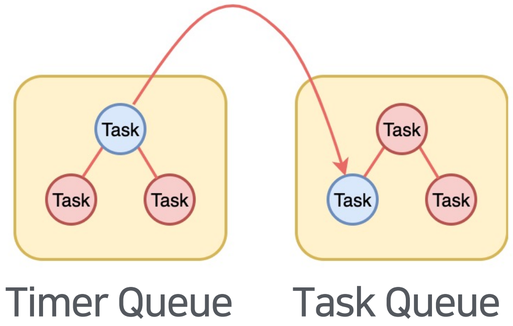
https://www.chromestatus.com/feature/6031161734201344

https://engineering.fb.com/2019/04/22/developer-tools/isinputpending-api/

4.2 동시성 모델과 이벤트 루프

Scheduler 우선 순위 제어

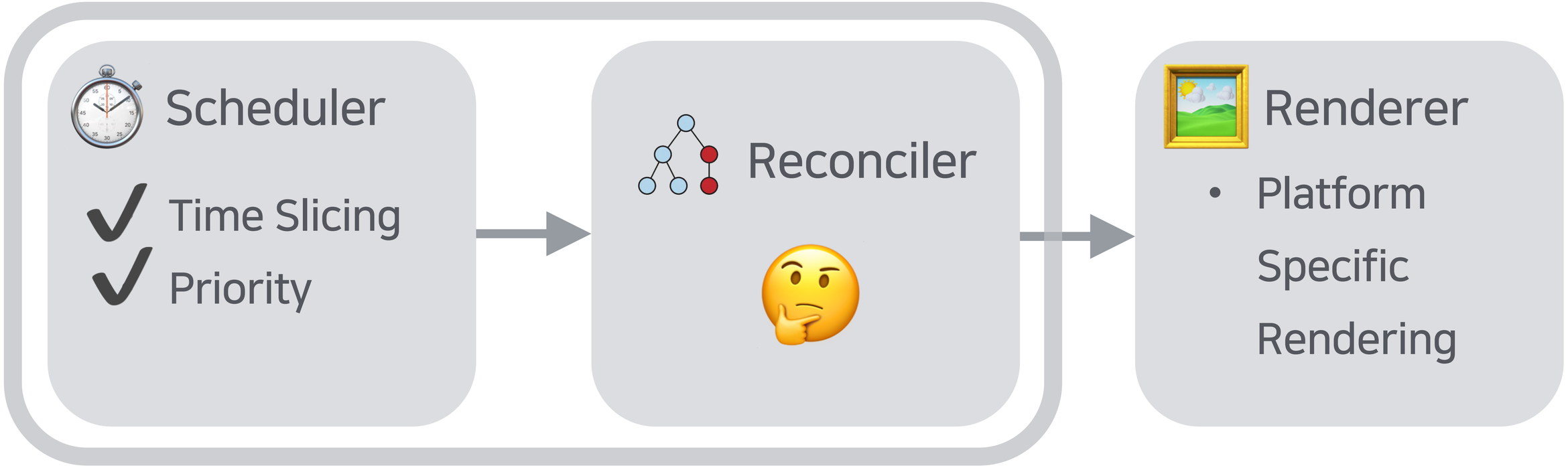
- Min Heap으로 구성된 Timer Queue, Task Queue
- Heap의 정렬 기준은 expirationTime
- scheduleCallback 호출시 우선 순위를 지정
- 우선 순위에 따라 expirationTime이 결정 된다 →
- Timer Queue에 들어갔다가 시간이 만료되면 Task Queue로 이동되어 실행된다




```
switch (priorityLevel) {
  case ImmediatePriority:
    timeout = IMMEDIATE_PRIORITY_TIMEOUT;     // -1
    break;
  case UserBlockingPriority:
    timeout = USER_BLOCKING_PRIORITY_TIMEOUT; // 250
    break;
  case IdlePriority:
    timeout = IDLE_PRIORITY_TIMEOUT;          // maxSigned31BitInt
    break;
  case LowPriority:
    timeout = LOW_PRIORITY_TIMEOUT;           // 10000
    break;
  case NormalPriority:
  default:
    timeout = NORMAL_PRIORITY_TIMEOUT;         // 5000
    break;
}

let expirationTime = startTime + timeout;
const newTask = {
  id: taskIdCounter++,
  callback: callback,
  ...
  sortIndex: -1
};

if (startTime > currentTime) {
  newTask.sortIndex = startTime;
} else {
  newTask.sortIndex = expirationTime;
```

4.2 동시성 모델과 이벤트 루프

세 가지 핵심 레이어



우리가 JavaScript로 일시정지 할 수 있는 부분

4.3 Fiber Architecture

DEVIEW 2021

What is Fiber?

- 랜더링 작업을 미세하게 분할한다
- 2016년 Sebastian Markbåge가 KC의 OCaml 동시성에서 영향을 받아 제안
- JavaScript Stack을 사용하지 않기 위해 작업을 힙 객체에 펼친 것
- Fiber라는 개념은 협력적 스케쥴링 모델로 Ruby 등 다양한 언어에 존재
- 스택 프레임을 메모리에 유지 하기 때문에 언제든지 실행 및 중지할 수 있다
- Algebraic Effects를 지원하기 위한 기반이 된다
- React요소 → Fiber → UnitOfWork

4.3 Fiber Architecture

DEVIEW 2021

Fiber Node Structure

```
type Fiber = {
  tag: WorkTag;                // Type of fiber
  type: any;                   // Resolved function/class
  return: Fiber | null,        // Return to after finishing
  child: Fiber | null,         // Singly Linked List Tree
  sibling: Fiber | null,       // Singly Linked List Tree
  alternate: Fiber | null,     // Double buffer
  lanes: Lanes,                // Priority
  childLanes: Lanes,           // Priority
  updateQueue: mixed,          // Queue of state updates, callbacks
  pendingProps: any,           // Input props
  memoizedProps: any,          // Output props
  memoizedState: any           // Output states
  nextEffect: Fiber | null, // Next fiber with side-effects
  firstEffect: Fiber | null,// To reuse the work done
  lastEffect: Fiber | null, // To reuse the work done
  ...
}
```

Fiber Tree를 구성하는 부분

```
type Fiber = {
  return: Fiber | null,
  child: Fiber | null,
  sibling: Fiber | null,
  alternate: Fiber | null,
}
```

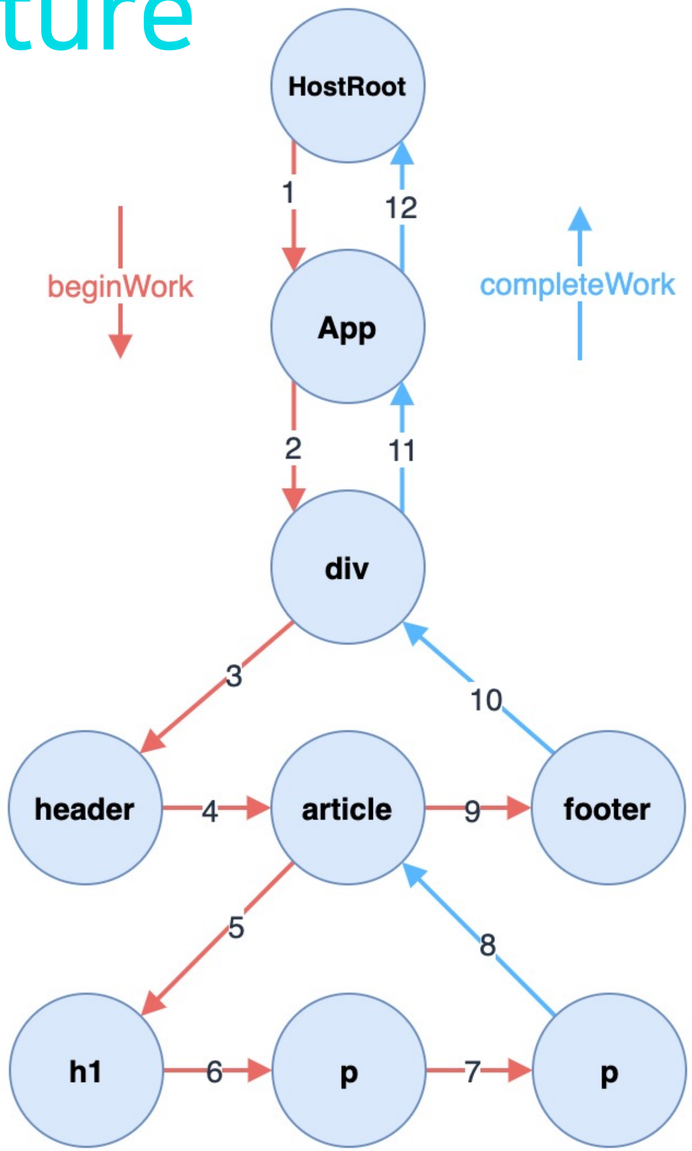
https://github.com/facebook/react/blob/main/packages/react-reconciler/src/ReactInternalTypes.js#L64

4.3 Fiber Architecture

DEVIEW 2021

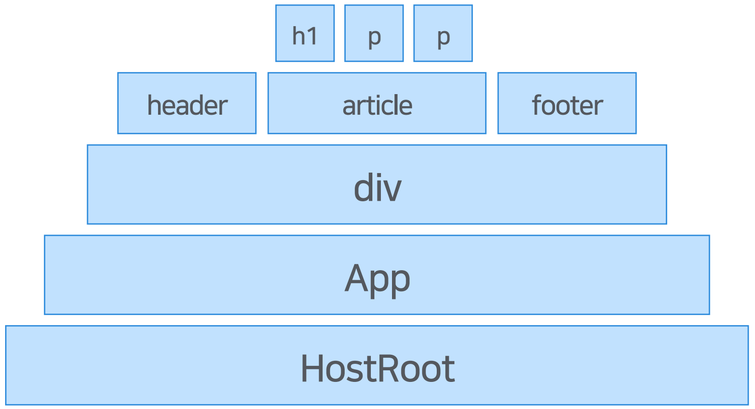
Fiber Tree

```
function App() {
  return (
    <div>
      <header></header>
      <article>
        <h1></h1>
        <p></p>
        <p></p>
      </article>
      <footer></footer>
    </div>
  )
}
```



Virtual Stack

```
App()
  div()
    header()
    article()
      h1()
      <----- interrupt!
      p()
      p()
    footer()
```



4.3 Fiber Architecture

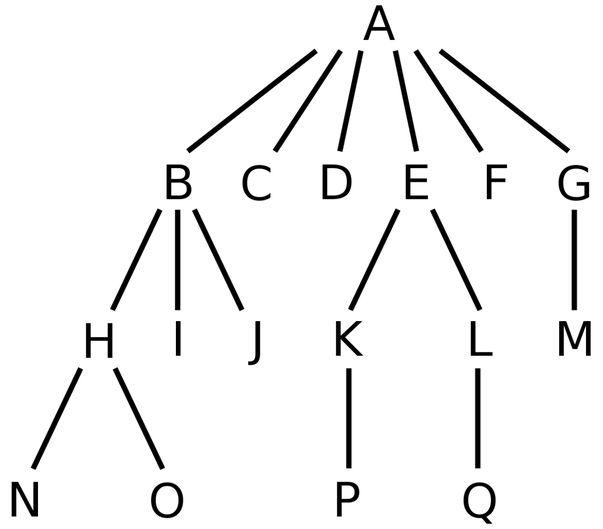
DEVIEW 2021

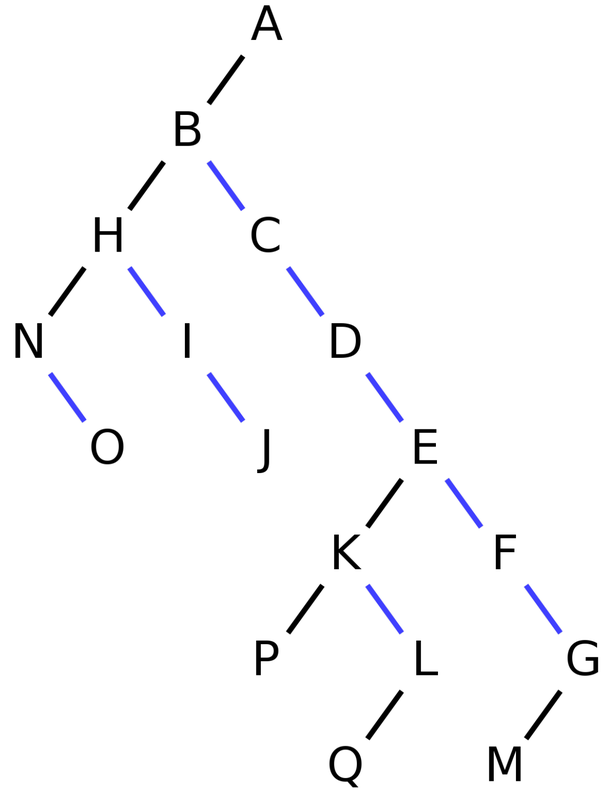
LC-RS Binary Tree

- 다음 작업을 쉽게 찾기 위해 사용

언제 사용하면 좋을까

- 메모리 효율성이 중요할 때
- 노드 랜덤 액세스가 필요하지 않을 때
- 트리의 루트를 제거하고 각 자식을 처리
- 한 트리를 다른 트리의 자식으로 만들 때





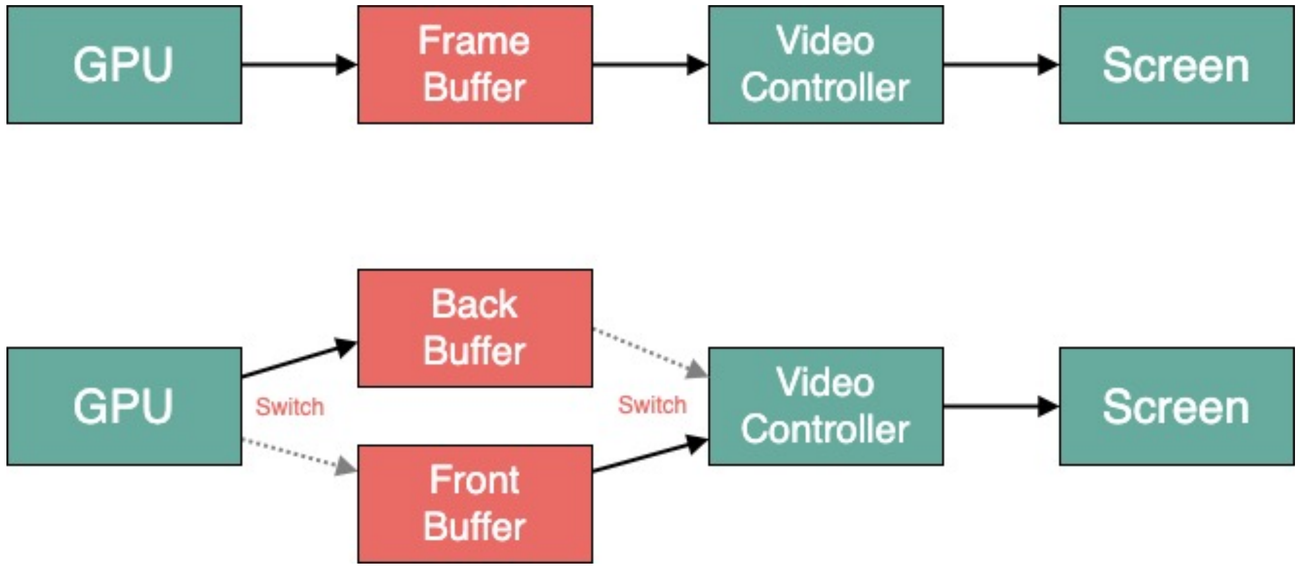
https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree

4.3 Fiber Architecture

Double Buffering Model

https://pixabay.com/photos/1002

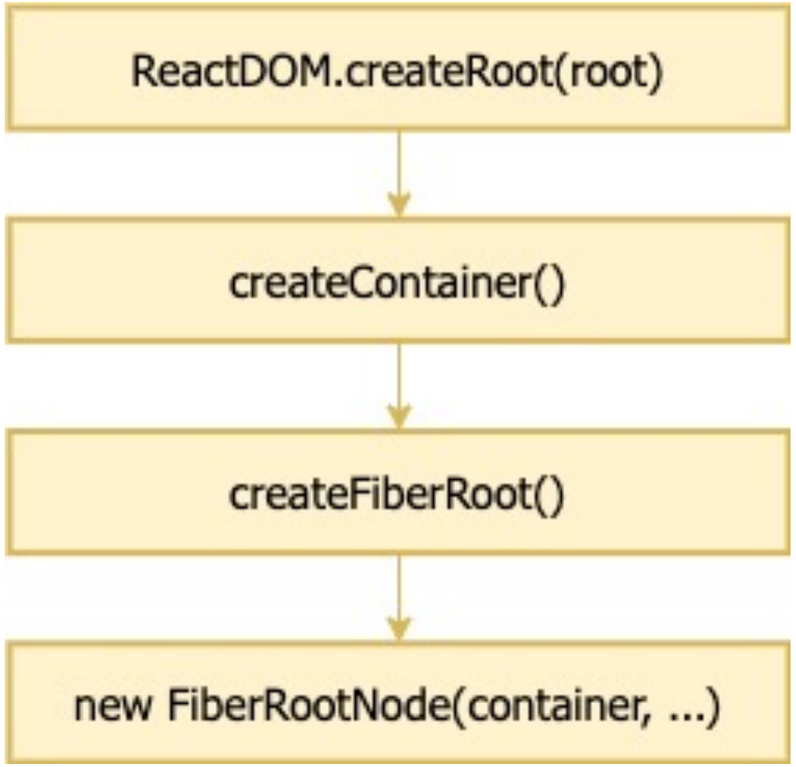
- 버퍼에 쓰기는 읽기보다 느리다
- 속도 차이로 불완전한 버퍼를 화면에
  그리면 Flickering이 발생한다
- 이를 해결하기 위해
  Front Buffer를 화면에 출력하는 동안
  Back Buffer에 다음에 그려질 내용을 쓴다
- 쓸 내용이 다 채워지면 두 버퍼를 스위칭
- GPU가 쓰고 있는 버퍼가 Back Buffer

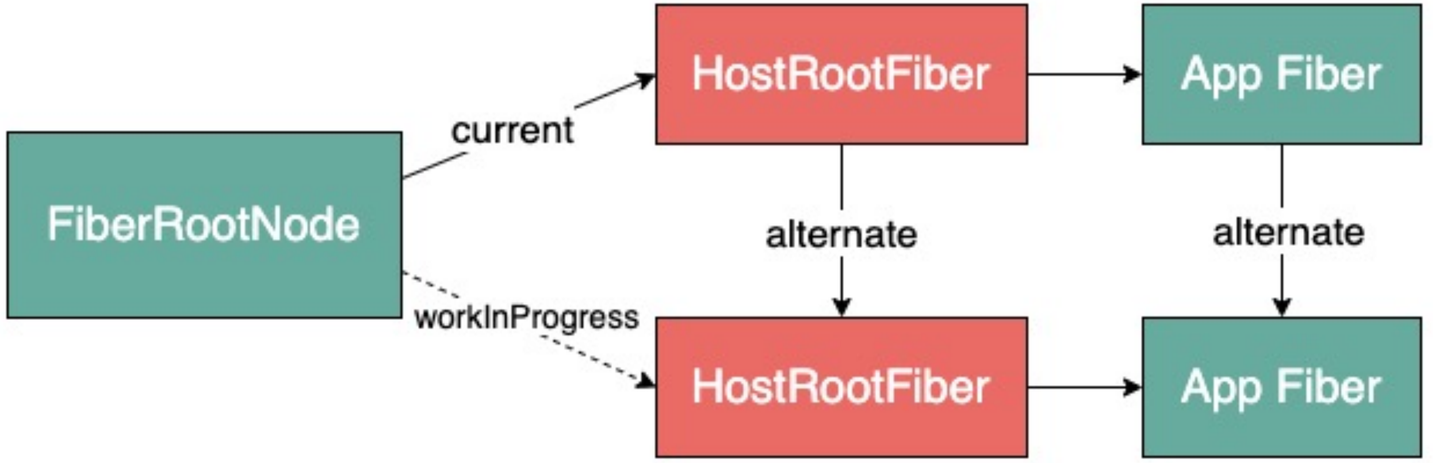


4.3 Fiber Architecture

DEVIEW 2021

Double Buffer 준비 과정





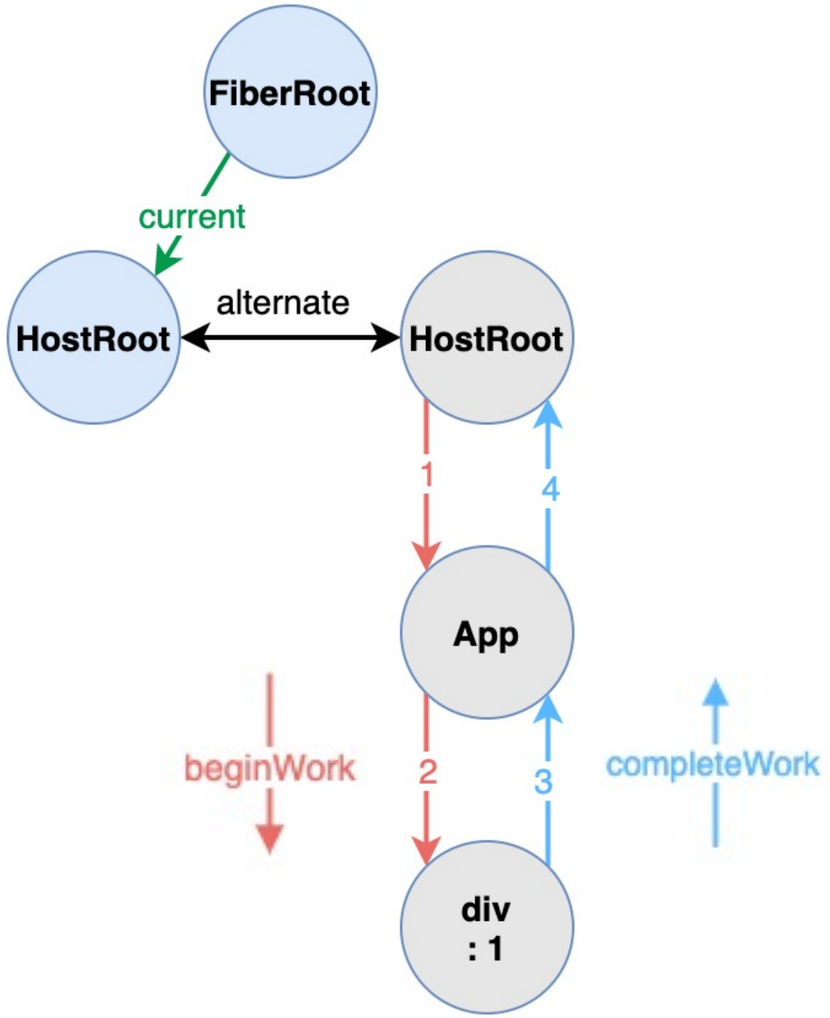
- FiberRootNode는 current로 HostRootFiber를 가리킨다
- HostRootFiber는 일반 Fiber로 alternate 속성을 갖는다
- 업데이트 할 때는 workInProgress를 수정 후 current 스위칭

4.3 Fiber Architecture

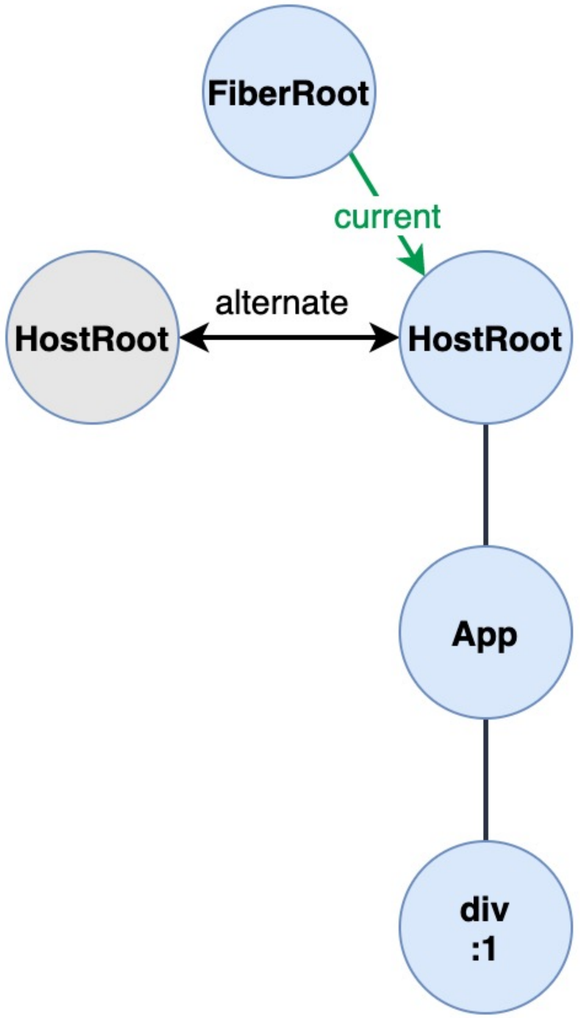
DEVIEW 2021

Mount Phase

```
function App() {
  const [v, setV] = useState(0)
  return (
    <div onClick={() => {
      setV(v + 1)
    }}>
      {v}
    </div>
  )
}

ReactDOM.createRoot(root)
  .render(<App />);
```



Step 1. alternate node 순회

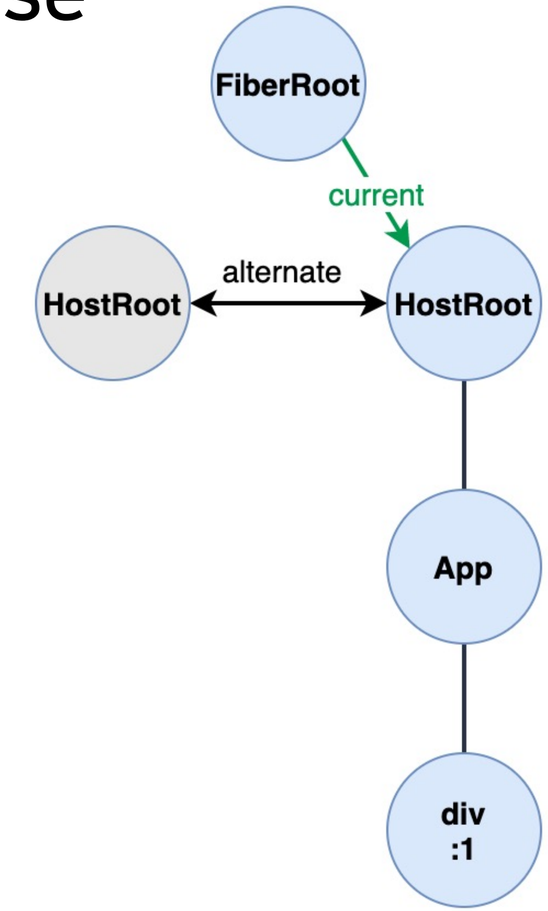


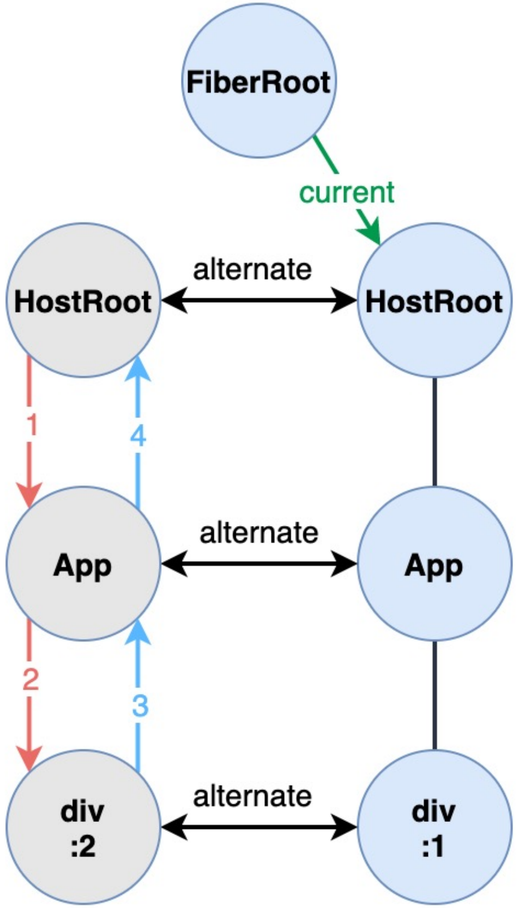
Step 2. current 변경

4.3 Fiber Architecture

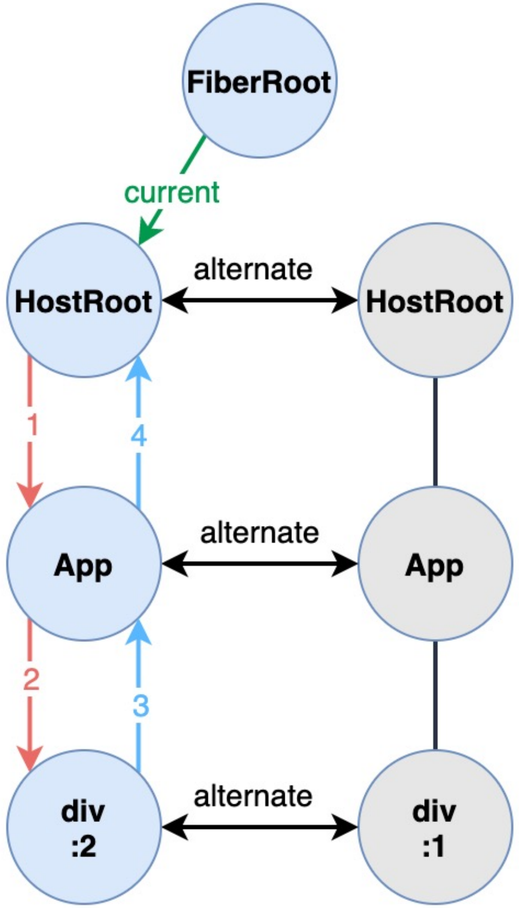
DEVIEW 2021

Update Phase





Step 1. alternate 순회

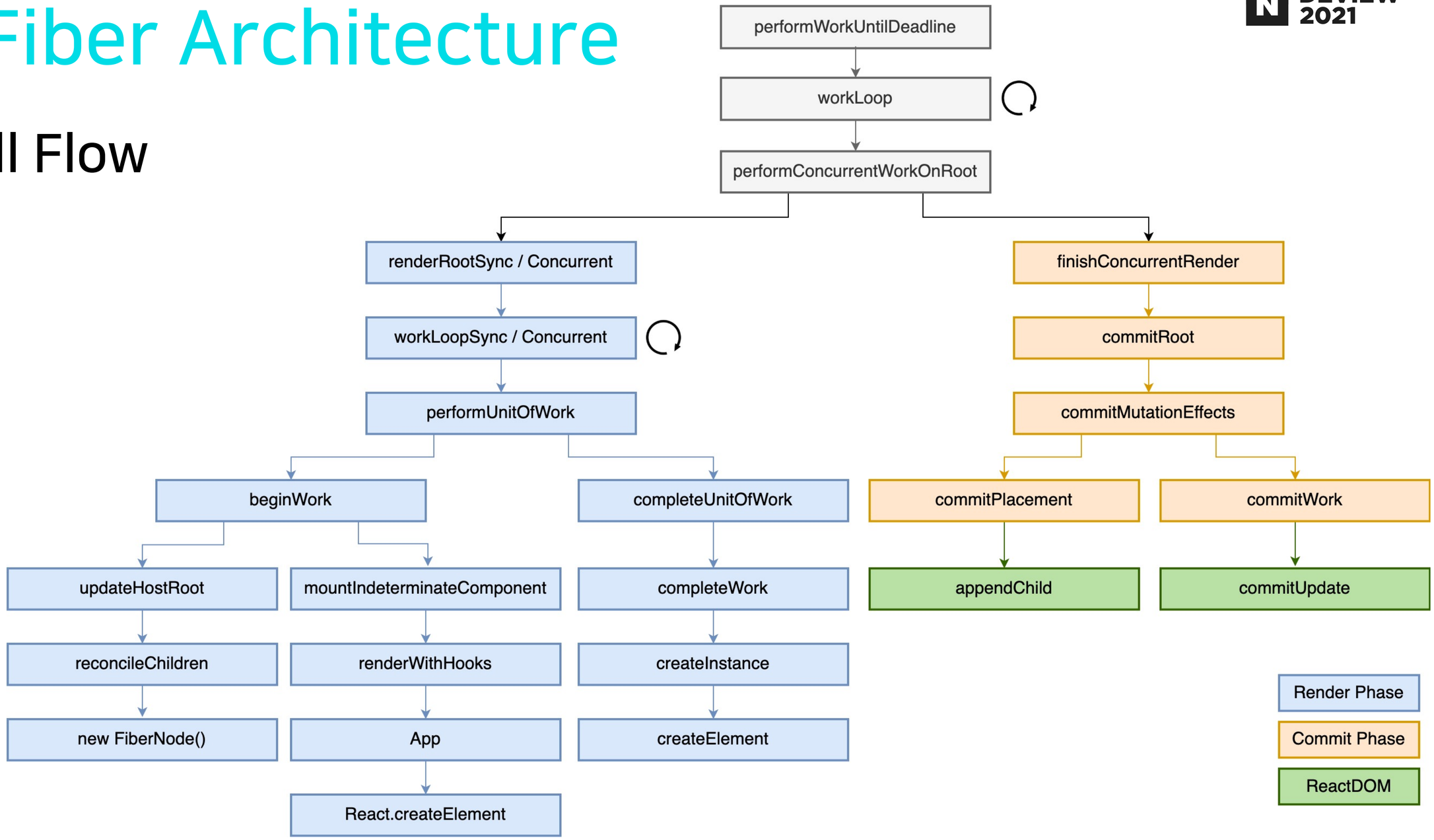


Step 2. current 변경

4.3 Fiber Architecture

DEVIEW 2021

Overall Flow



4.3 Fiber Architecture

DEVIEW 2021

Lane Model for Fiber Priority

- 이전에는 만료 시간을 기준으로 우선 순위를 처리했음
- Task prioritization : A가 B보다 급한가
- Task batching : A가 이 그룹 테스크에 속하는가
- Lane은 Task prioritization과 Task batching의 개념을 분리
  - CPU > I/O > CPU 작업 우선 순위가 있을 때
    그룹으로 분리하면 CPU 작업의 병목을 막을 수 있음
- 32비트 데이터로 다수의 고유한 작업 스레드 표현 가능
- 비트마스크의 특성을 살려 연산속도가 빠름

https://pixabay.com/photos/5108525

https://github.com/facebook/react/pull/18796

4.3 Fiber Architecture

DEVIEW 2021

Task prioritization : Lane Type

- 1 순위 : 기한이 지난 또는 동기화 작업
- 2 순위 : 사용자 상호 작용에 의한 업데이트
- 3 순위 : 일반 우선 순위

  (네트워크 요청에 의해 생성된 업데이트 등)
- 가장 낮은 순위 : Suspense

Task batching : Lanes Type

- InputDiscreteLanes : 사용자 상호 작용
- DefaultLanes : 데이터 요청에 의한 업데이트
- TransitionLanes :

  Suspense, useTransition, useDeferredValue

```
export const TotalLanes = 31;

export const NoLanes: Lanes = /*                        */ 0b0000000000000000000000000000000;
export const NoLane: Lane = /*                          */ 0b0000000000000000000000000000000;

export const SyncLane: Lane = /*                        */ 0b0000000000000000000000000000001;

export const InputContinuousHydrationLane: Lane = /*    */ 0b0000000000000000000000000000010;
export const InputContinuousLane: Lanes = /*            */ 0b0000000000000000000000000000100;

export const DefaultHydrationLane: Lane = /*            */ 0b0000000000000000000000000001000;
export const DefaultLane: Lanes = /*                    */ 0b0000000000000000000000000010000;

const TransitionHydrationLane: Lane = /*                */ 0b0000000000000000000000000100000;
const TransitionLanes: Lanes = /*                       */ 0b0000000001111111111111111000000;
const TransitionLane1: Lane = /*                        */ 0b0000000000000000000000001000000;
const TransitionLane2: Lane = /*                        */ 0b0000000000000000000000010000000;
const TransitionLane3: Lane = /*                        */ 0b0000000000000000000000100000000;
const TransitionLane4: Lane = /*                        */ 0b0000000000000000000001000000000;
const TransitionLane5: Lane = /*                        */ 0b0000000000000000000010000000000;
const TransitionLane6: Lane = /*                        */ 0b0000000000000000000100000000000;
const TransitionLane7: Lane = /*                        */ 0b0000000000000000001000000000000;
const TransitionLane8: Lane = /*                        */ 0b0000000000000000010000000000000;
const TransitionLane9: Lane = /*                        */ 0b0000000000000000100000000000000;
const TransitionLane10: Lane = /*                       */ 0b0000000000000001000000000000000;
const TransitionLane11: Lane = /*                       */ 0b0000000000000010000000000000000;
const TransitionLane12: Lane = /*                       */ 0b0000000000000100000000000000000;
const TransitionLane13: Lane = /*                       */ 0b0000000000001000000000000000000;
const TransitionLane14: Lane = /*                       */ 0b0000000000010000000000000000000;
const TransitionLane15: Lane = /*                       */ 0b0000000000100000000000000000000;
const TransitionLane16: Lane = /*                       */ 0b0000000001000000000000000000000;

const RetryLanes: Lanes = /*                            */ 0b0000011111000000000000000000000;
const RetryLane1: Lane = /*                             */ 0b0000000010000000000000000000000;
const RetryLane2: Lane = /*                             */ 0b0000000100000000000000000000000;
const RetryLane3: Lane = /*                             */ 0b0000001000000000000000000000000;
const RetryLane4: Lane = /*                             */ 0b0000010000000000000000000000000;
const RetryLane5: Lane = /*                             */ 0b0000100000000000000000000000000;

export const SomeRetryLane: Lane = RetryLane1;

export const SelectiveHydrationLane: Lane = /*          */ 0b0001000000000000000000000000000;

const NonIdleLanes = /*                                 */ 0b0001111111111111111111111111111;

export const IdleHydrationLane: Lane = /*               */ 0b0010000000000000000000000000000;
export const IdleLane: Lanes = /*                       */ 0b0100000000000000000000000000000;

export const OffscreenLane: Lane = /*                   */ 0b1000000000000000000000000000000;
```

4.3 Fiber Architecture

DEVIEW 2021

Lane Bit Operations

- 가장 높은 우선 순위 찾기
- 특정 배치 작업 순위에 현재 테스크가 속하는가
- 여러 순위 작업을 머지
- 배치 작업에서 지우기
- 두 Lanes에서 교집합을 구하기

```
export function getHighestPriorityLane(lanes: Lanes): Lane {
  return lanes & -lanes;
}

export function includesSomeLane(a: Lanes | Lane, b: Lanes | Lane) {
  return (a & b) !== NoLanes;
}

export function isSubsetOfLanes(set: Lanes, subset: Lanes | Lane) {
  return (set & subset) === subset;
}

export function mergeLanes(a: Lanes | Lane, b: Lanes | Lane): Lanes {
  return a | b;
}

export function removeLanes(set: Lanes, subset: Lanes | Lane): Lanes {
  return set & ~subset;
}

export function intersectLanes(a: Lanes | Lane, b: Lanes | Lane): Lanes
{
  return a & b;
}
```

4.3 Fiber Architecture

DEVIEW 2021

Lane Model로 얻는 것

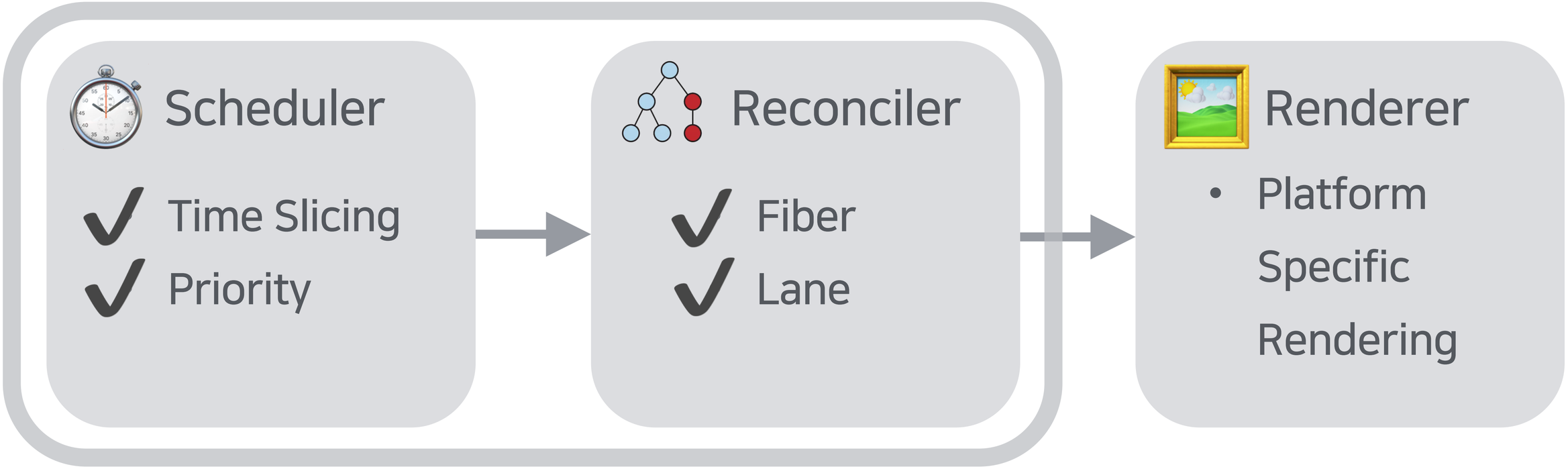
- useTransition에서 중간 상태 건너 뛰기 구현
- useTransition을 통한 Parallel UI transitions

  : 관련 없는 트리의 전환이 서로 블록하는 것을 막기
- useState후크 를 통해 예약된 상태 업데이트를 빠르게 계산

https://github.com/facebook/react/pull/18796

4.3 Fiber Architecture

세 가지 핵심 레이어



스케쥴러와 Fiber Reconciler로 Blocking 문제를 해결했습니다!

5. References

- https://reactjs.org/blog/2021/06/08/the-plan-for-react-18.html
- https://github.com/reactwg/react-18
- https://reactjs.org/docs/concurrent-mode-intro.html
- https://github.com/reactwg/react-18/discussions/64
- https://go.dev/blog/waza-talk
- https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism
- https://wiki.haskell.org/Parallelism_vs._Concurrency
- https://www.hanbit.co.kr/store/books/look.php?p_code=B3745244799
- https://developers.google.com/web/updates/2018/09/inside-browser-part1
- https://www.youtube.com/watch?v=mDdgfyRB5kg&t=94s
- https://ko.reactjs.org/docs/implementation-notes.html
- https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallback
- https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel
- https://www.chromestatus.com/feature/6031161734201344
- https://engineering.fb.com/2019/04/22/developer-tools/isinputpending-api/
- https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree
- https://www.hardwaretimes.com/what-is-v-sync-should-you-turn-it-on-or-off
- https://github.com/facebook/react/pull/18796
- https://overreacted.io/my-decade-in-review/

Thank You