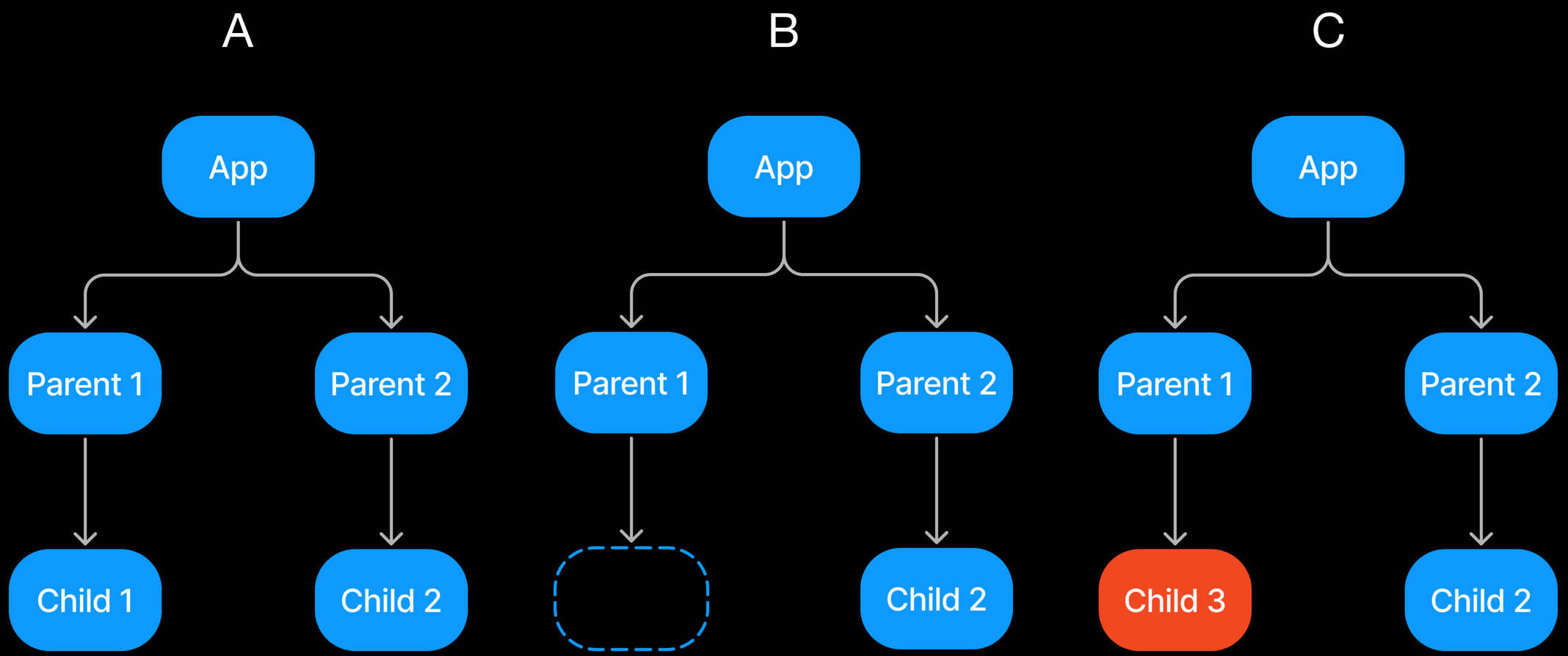
React Fiber

How does react draw UI

Junepil Lee @Konkuk University

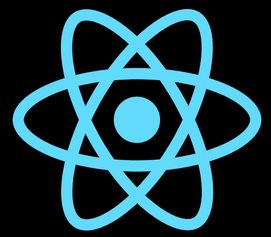
Reconciliation

Reconcile : 조화시키다


A
App
Parent 1
Parent 2
Child 1
Child 2
B
App
Parent 1
Parent 2
Child 2
C
App
Parent 1
Parent 2
Child 3
Child 2

변경된 부분만 다시 렌더링

Virtual DOM 얘기하는 건가요?



When?

Update가 발생할 때

- useState
- useEffect
- useReducer
- And so on…

Scheduling

Work

update로 인해 실행되어야 하는 연산

- Fiber tree의 구조를 변경한다
- Fiber의 prop을 변경한다
- etc…

리엑트의 디자인 원칙

- UI 라이브러리는 모든 update가 즉시 적용되지 않아도 된다
- animation의 update의 우선순위가 data 저장보다 높다
- Fiber가 도입되기 전까지는 이러한 원칙이 적용되지 않았다 🤔

Scheduling

우선적으로 처리해야 할 것들부터 처리하자

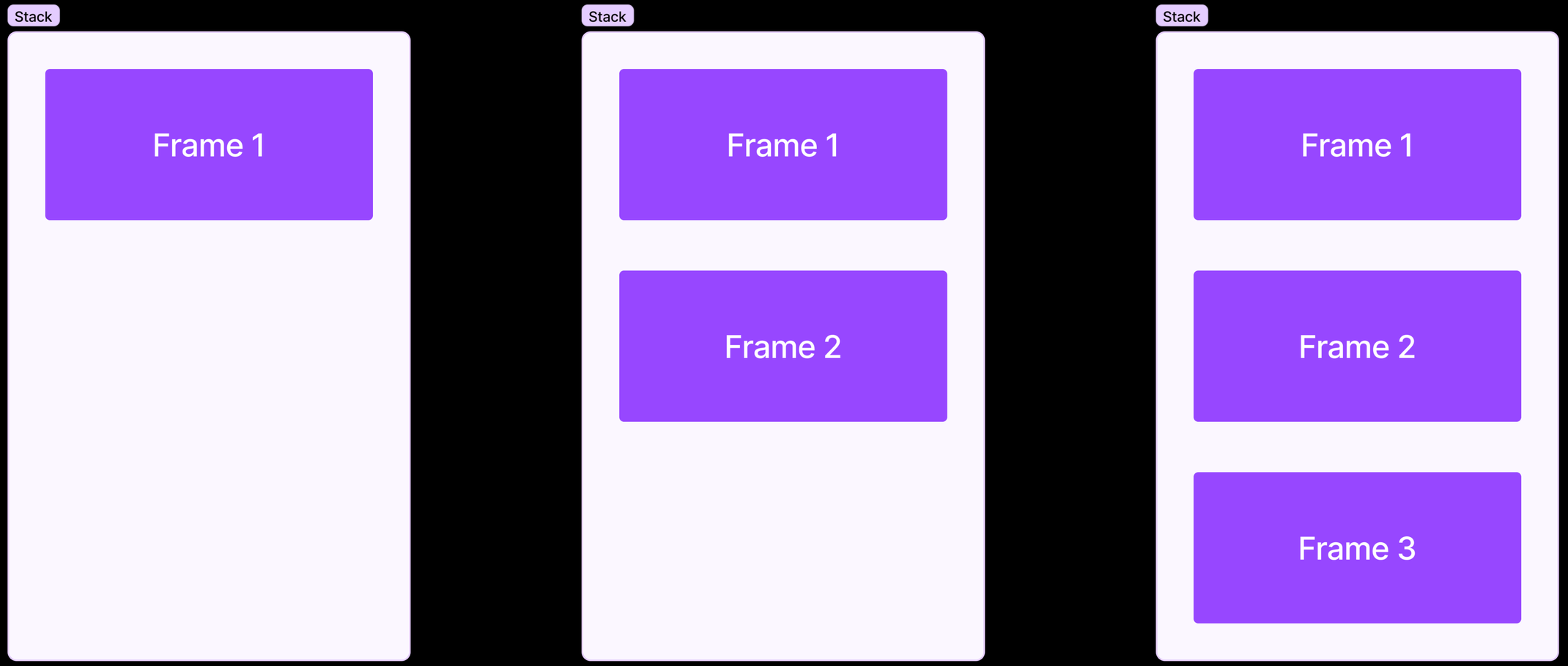
Fiber

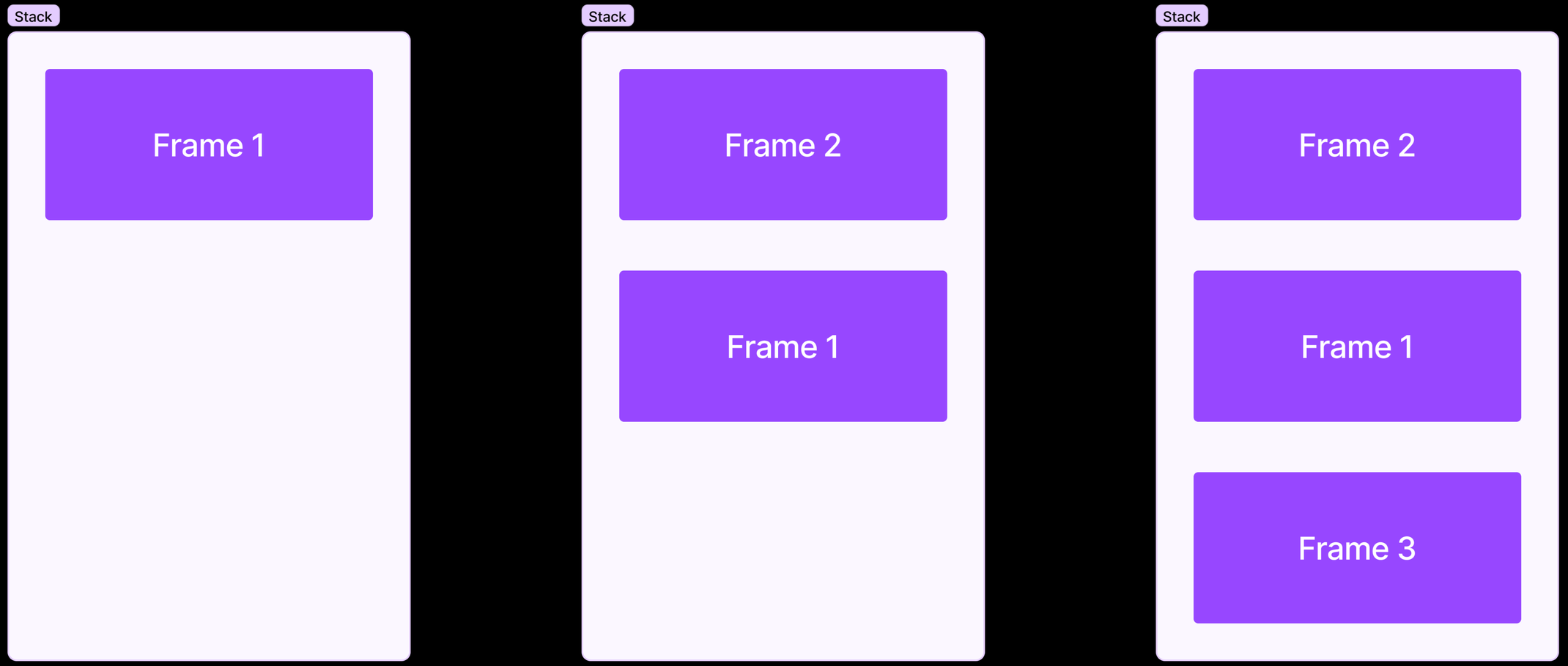
ReactElement

```jsx
export const NumberBox = () => {
  let number = Math.round(Math.random() * 100);

  return <div>Todays random number is: {number}</div>;
};

```

Fiber

```jsx
function createFiberImplObject(
  tag: WorkTag,
  pendingProps: mixed,
  key: null | string,
  mode: TypeOfMode,
): Fiber {
  const fiber: Fiber = {
    // Instance
    // tag, key - defined at the bottom as dynamic properties
    elementType: null,
    type: null,
    stateNode: null,

    // Fiber
    return: null,
    child: null,
    sibling: null,
    index: 0,

```


Stack
Frame 1
Stack
Frame 1
Frame 2
Stack
Frame 1
Frame 2
Frame 3


Stack
Frame 1
Stack
Frame 2
Frame 1
Stack
Frame 2
Frame 1
Frame 3

우선순위 판단

Lanes

- 발생하는 이벤트별로 우선순위를 미리 지정해 Work를 스케쥴링
- setState( ) API 호출 시 또는 DOM 이벤트 호출 시 Lane이 할당된다.

```
1
2  export const NoLanes: Lanes = /*                        */ 0b0000000000000000000000000000000;
3  export const NoLane: Lane = /*                          */ 0b0000000000000000000000000000000;
4
5  export const SyncHydrationLane: Lane = /*               */ 0b0000000000000000000000000000001;
6  export const SyncLane: Lane = /*                        */ 0b0000000000000000000000000000010;
7  export const SyncLaneIndex: number = 1;
8
9  export const InputContinuousHydrationLane: Lane = /*    */ 0b0000000000000000000000000000100;
10 export const InputContinuousLane: Lane = /*              */ 0b0000000000000000000000000001000;
11
12 export const DefaultHydrationLane: Lane = /*            */ 0b0000000000000000000000000010000;
13 export const DefaultLane: Lane = /*                      */ 0b0000000000000000000000000100000;
14
15 export const SyncUpdateLanes: Lane =
16   SyncLane | InputContinuousLane | DefaultLane;
17
18 const TransitionHydrationLane: Lane = /*                 */ 0b0000000000000000000000001000000;
19 const TransitionLanes: Lanes = /*                        */ 0b0000000001111111111111110000000;
20 const TransitionLane1: Lane = /*                         */ 0b0000000000000000000000010000000;
21 const TransitionLane2: Lane = /*                         */ 0b0000000000000000000000100000000;
22 const TransitionLane3: Lane = /*                         */ 0b0000000000000000000001000000000;
23 const TransitionLane4: Lane = /*                         */ 0b0000000000000000000010000000000;
24 const TransitionLane5: Lane = /*                         */ 0b0000000000000000000100000000000;
25 const TransitionLane6: Lane = /*                         */ 0b0000000000000000001000000000000;
26 const TransitionLane7: Lane = /*                         */ 0b0000000000000000010000000000000;
27 const TransitionLane8: Lane = /*                         */ 0b0000000000000000100000000000000;
28 const TransitionLane9: Lane = /*                         */ 0b0000000000000001000000000000000;
29 const TransitionLane10: Lane = /*                        */ 0b0000000000000010000000000000000;
30 const TransitionLane11: Lane = /*                        */ 0b0000000000000100000000000000000;
31 const TransitionLane12: Lane = /*                        */ 0b0000000000001000000000000000000;
32 const TransitionLane13: Lane = /*                        */ 0b0000000000010000000000000000000;
33 const TransitionLane14: Lane = /*                        */ 0b0000000000100000000000000000000;
34 const TransitionLane15: Lane = /*                        */ 0b0000000001000000000000000000000;
35
36 const RetryLanes: Lanes = /*                             */ 0b0000011110000000000000000000000;
37 const RetryLane1: Lane = /*                              */ 0b0000000010000000000000000000000;
38 const RetryLane2: Lane = /*                              */ 0b0000000100000000000000000000000;
39 const RetryLane3: Lane = /*                              */ 0b0000001000000000000000000000000;
40 const RetryLane4: Lane = /*
```

Fiber의 구조

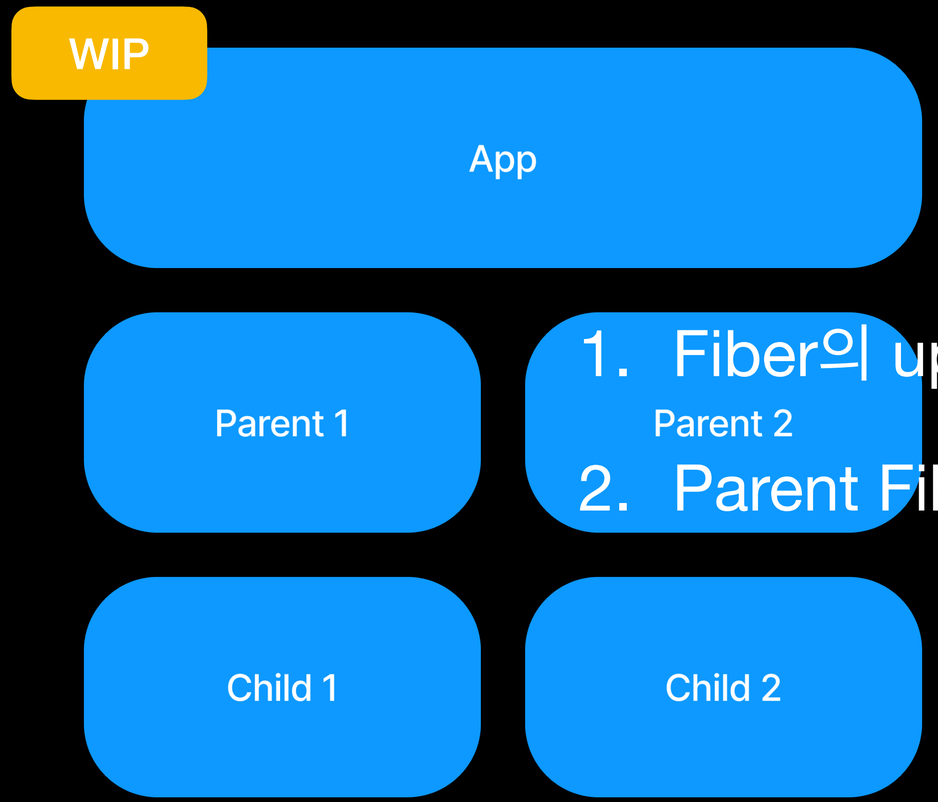
- type & key
- child & sibling
- return
- pendingProps & memoizedProps

Type & Key

- 대응하는 ReactElement가 함수형 컴포넌트라면 function( ), HTMLElement이면 요소의 이름를 type으로 가진다.
- Key를 이용해 재사용 가능한 Fiber인지 판단한다.

Child & Sibling

```
1   export const Parent = () => {
2     return (
3       <>
4         <Child />;
5         <Child />;
6       </>
7     );
8   };
9
10  export const Child = () => {
11    return <div>I am child</div>;
12  };
13
```

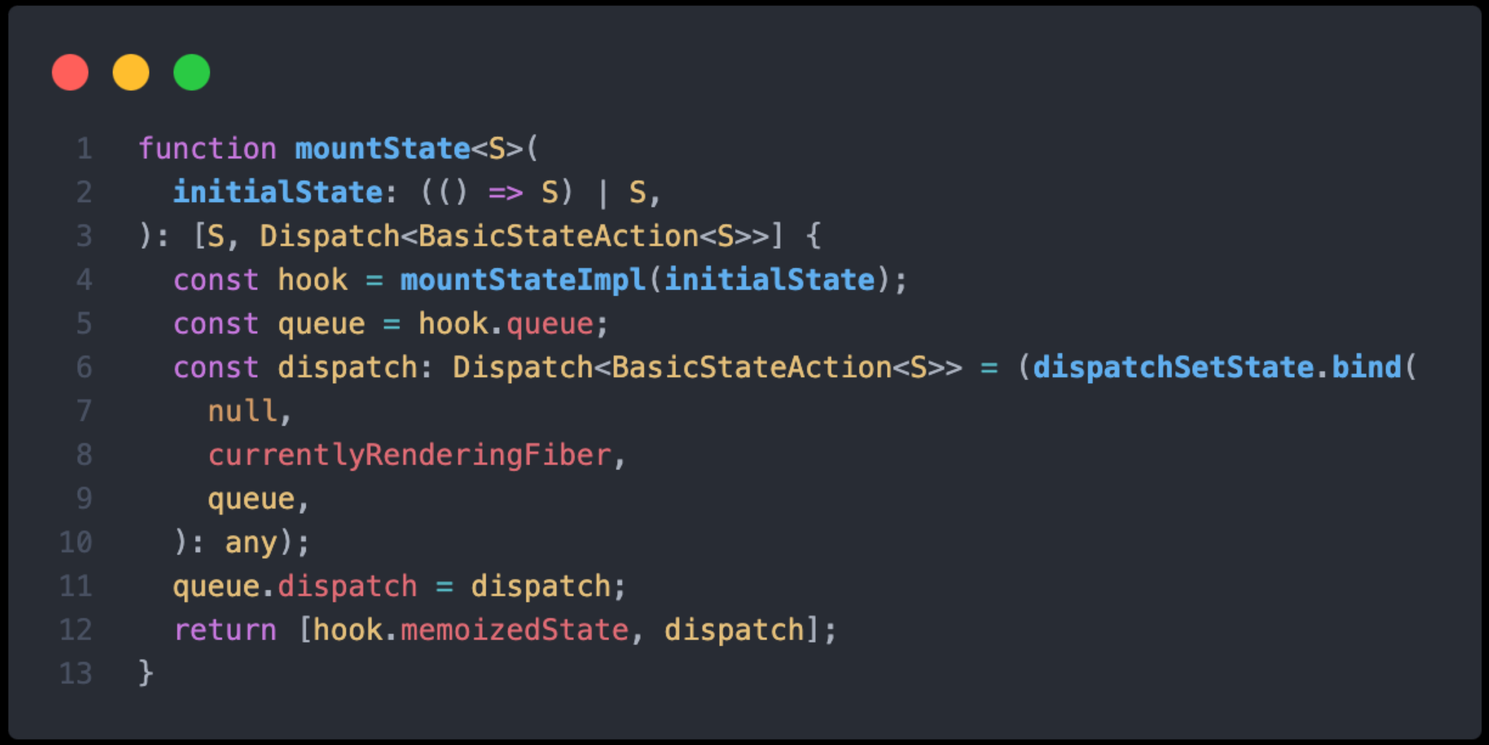


1. Fiber의 update가 발생했을 때
2. Parent Fiber가 새로 렌더링 됐을 때

```
function workLoopConcurrent() {
  // Perform work until Scheduler asks us to yield
  while (workInProgress !== null && !shouldYield()) {
    // $FlowFixMe[incompatible-call] found when upgrading Flow
    performUnitOfWork(workInProgress);
  }
}
```

```
function performUnitOfWork(unitOfWork: Fiber): void {
  // The current, flushed, state of this fiber is the alternate. Ideally
  // nothing should rely on this, but relying on it here means that we don't
  // need an additional field on the work in progress.
  const current = unitOfWork.alternate;

  let next;
  ...
      next = beginWork(current, unitOfWork, entangledRenderLanes);
  ...

  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next === null) {
    // If this doesn't spawn new work, complete the current work.
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}
```

Fiber는 update를 어떻게 알아요?



```
function mountState<S>(
  initialState: (() => S) | S,
): [S, Dispatch<BasicStateAction<S>>] {
  const hook = mountStateImpl(initialState);
  const queue = hook.queue;
  const dispatch: Dispatch<BasicStateAction<S>> = (dispatchSetState.bind(
    null,
    currentlyRenderingFiber,
    queue,
  ): any);
  queue.dispatch = dispatch;
  return [hook.memoizedState, dispatch];
}
```

```
function dispatchSetState<S, A>(
  fiber: Fiber,
  queue: UpdateQueue<S, A>,
  action: A,
): void {
  if (__DEV__) {
    if (typeof arguments[3] === 'function') {
      console.error(
        "State updates from the useState() and useReducer() Hooks don't support the " +
          'second callback argument. To execute a side effect after ' +
          'rendering, declare it in the component body with useEffect().',
      );
    }
  }

  const lane = requestUpdateLane(fiber);
  const didScheduleUpdate = dispatchSetStateInternal(
    fiber,
    queue,
    action,
    lane,
  );
  if (didScheduleUpdate) {
    startUpdateTimerByLane(lane);
  }
  markUpdateInDevTools(fiber, lane, action);
}
```

References

- https://en.wikipedia.org/wiki/Call_stack
- https://github.com/acdlite/react-fiber-architecture?tab=readme-ov-file
- https://github.com/facebook/react/tree/main/packages/react-reconciler
- https://youtu.be/6-MYouU_GGk?si=HGbGf_EUHrE0n2QQ
- https://react.dev/learn/understanding-your-ui-as-a-tree
- https://goidle.github.io/react/in-depth-react-hooks_1/
- https://goidle.github.io/react/in-depth-react18-lane/