React Fiber Architecture

What is reconciliation?

reconciliation
The algorithm React uses to diff one tree with another to determine which parts need to be changed.

update
A change in the data used to render a React app. Usually the result of `setState`. Eventually results in a re-render.

Reconciliation versus rendering

The DOM is just one of the rendering environments React can render to, the other major targets being native iOS and Android views via React Native. (This is why "virtual DOM" is a bit of a misnomer.)

The reason it can support so many targets is because React is designed so that reconciliation and rendering are separate phases. The reconciler does the work of computing which parts of a tree have changed; the renderer then uses that information to actually update the rendered app.

Scheduling

scheduling
the process of determining when work should be performed.

work
any computations that must be performed. Work is usually the result of an update (e.g. `setState`).

The key points are:

- In a UI, it's not necessary for every update to be applied immediately; in fact, doing so can be wasteful, causing frames to drop and degrading the user experience.
- Different types of updates have different priorities — an animation update needs to complete more quickly than, say, an update from a data store.
- A push-based approach requires the app (you, the programmer) to decide how to schedule work. A pull-based approach allows the framework (React) to be smart and make those decisions for you.

React doesn't currently take advantage of scheduling in a significant way; an update results in the entire subtree being re-rendered immediately. Overhauling React's core algorithm to take advantage of scheduling is the driving idea behind Fiber.

What is a fiber?

We're about to discuss the heart of React Fiber's architecture. Fibers are a much lower-level abstraction than application developers typically think about. If you find yourself frustrated in your attempts to understand it, don't feel discouraged. Keep trying and it will eventually make sense. (When you do finally get it, please suggest how to improve this section.)

We've established that a primary goal of Fiber is to enable React to take advantage of scheduling. Specifically, we need to be able to

- pause work and come back to it later.
- assign priority to different types of work.
- reuse previously completed work.
- abort work if it's no longer needed.

In order to do any of this, we first need a way to break work down into units. In one sense, that's what a fiber is. A fiber represents a **unit of work**.

To go further, let's go back to the conception of React components as functions of data, commonly expressed as

```
v = f(d)
```

It follows that rendering a React app is akin to calling a function whose body contains calls to other functions, and so on. This analogy is useful when thinking about fibers.

The way computers typically track a program's execution is using the call stack. When a function is executed, a new **stack frame** is added to the stack. That stack frame represents the work that is performed by that function.

When dealing with UIs, the problem is that if too much work is executed all at once, it can cause animations to drop frames and look choppy. What's more, some of that work may be unnecessary if it's superseded by a more recent update (i.e, **If React tries to update everything at once, it can block the browser and cause animations to lag or look choppy. React may start updating the UI based on old data, but if a new update comes in before it's done, the earlier work becomes useless and is thrown away**.) . This is where the comparison between UI components and function breaks down, because components have more specific concerns than functions in general.

Newer browsers (and React Native) implement APIs that help address this exact problem: `requestIdleCallback` schedules a low priority function to be called during an idle period, and `requestAnimationFrame` schedules a high priority function to be called on the next animation frame. The problem is that, in order to use those APIs, you need a way to break rendering work into incremental units. If you rely only on the call stack, it will keep doing work until the stack is empty.

Wouldn't it be great if we could **customize the behavior of the call stack to optimize for rendering UI**s? Wouldn't it be great if we could **interrupt the call stack at will and manipulate stack frames manually**?

That's the purpose of React Fiber. Fiber is reimplementation of the stack, specialized for React components. You can think of a single fiber as a virtual stack frame**.**

The advantage of reimplementing the stack is that **you can [keep stack frames in memory](#) and execute them however (and *whenever*) you want.** This is crucial for accomplishing the goals we have for scheduling.

Aside from scheduling, manually dealing with stack frames unlocks the potential for features such as concurrency and error boundaries. We will cover these topics in future sections.

In the next section, we'll look more at the structure of a fiber.

Structure of a fiber

Note: as we get more specific about implementation details, the likelihood that something may change increases. Please file a PR if you notice any mistakes or outdated information.

In concrete terms, a **fiber is a JavaScript object that contains information about a component, its input, and its output.**

React's Fiber architecture doesn't use the actual call stack, because that can't be paused or controlled. Instead, React creates its own custom data structure using Fiber objects.

A fiber corresponds to a stack frame, but it also corresponds to an instance of a component.

Here are some of the important fields that belong to a fiber. (This list is not exhaustive.)

`type` and `key`

The type and key of a fiber serve the same purpose as they do for React elements. (In fact, when a fiber is created from an element, these two fields are copied over directly.)

The type of a fiber describes the component that it corresponds to. For composite components, the type is the function or class component itself. For host components (`div`, `span`, etc.), the type is a string.

Conceptually, the type is the function (as in `v = f(d)`) whose execution is being tracked by the stack frame.

Along with the type, the key is used during reconciliation to determine whether the fiber can be reused.

`child` and `sibling`

These fields point to other fibers, describing the recursive tree structure of a fiber.

The child fiber corresponds to the value returned by a component's `render` method. So in the following example

```
function Parent() {
  return <Child />
}
```

The child fiber of `Parent` corresponds to `Child`.

The sibling field accounts for the case where `render` returns multiple children (a new feature in Fiber!):

```
function Parent() {
  return [<Child1 />, <Child2 />]
}
```

The **child fibers form a singly-linked list whose head is the first child.** So in this example, the child of `Parent` is `Child1` and the sibling of `Child1` is `Child2`.

Going back to our function analogy, you can think of a child fiber as a tail-called function.

`return`

The return fiber is the fiber to which the program should return after processing the current one. It is conceptually the same as the return address of a stack frame. It can also be thought of as the parent fiber.

If a fiber has multiple child fibers, each child fiber's return fiber is the parent. So in our example in the previous section, the return fiber of `Child1` and `Child2` is `Parent`.

`pendingProps` and `memoizedProps`

Memoization is an optimization technique used to speed up computer programs by storing the results of expensive function calls and reusing them when the same inputs occur again

Conceptually, props are the arguments of a function. A fiber's `pendingProps` are set at the beginning of its execution, and `memoizedProps` are set at the end.

When the incoming `pendingProps` are equal to `memoizedProps`, it signals that the fiber's previous output can be reused, preventing unnecessary work.

`pendingWorkPriority`

A number indicating the priority of the work represented by the fiber. The ReactPriorityLevel module lists the different priority levels and what they represent.

With the exception of `NoWork`, which is 0, a larger number indicates a lower priority. For example, you could use the following function to check if a fiber's priority is at least as high as the given level:

```
function matchesPriority(fiber, priority) {
  return fiber.pendingWorkPriority !== 0 &&
         fiber.pendingWorkPriority <= priority
}
```

This function is for illustration only; it's not actually part of the React Fiber codebase.

The scheduler uses the priority field to search for the next unit of work to perform. This algorithm will be discussed in a future section.

`alternate`

flush
To flush a fiber is to render its output onto the screen.

work-in-progress

A fiber that has not yet completed; conceptually, a stack frame which has not yet returned.

At any time, a component instance has at most two fibers that correspond to it: the current, flushed fiber, and the work-in-progress fiber.

The alternate of the current fiber is the work-in-progress, and the alternate of the work-in-progress is the current fiber.

A fiber's alternate is created lazily using a function called `cloneFiber`. Rather than always creating a new object, `cloneFiber` will attempt to reuse the fiber's alternate if it exists, minimizing allocations.

You should think of the `alternate` field as an implementation detail, but it pops up often enough in the codebase that it's valuable to discuss it here.

`output`

host component

The leaf nodes of a React application. They are specific to the rendering environment (e.g., in a browser app, they are `div`, `span`, etc.). In JSX, they are