```
/**
 * @license React
 * react-dom.development.js
 *
 * Copyright (c) Facebook, Inc. and its affiliates.
 *
 * This source code is licensed under the MIT license found in the
 * LICENSE file in the root directory of this source tree.
 */
(function (global, factory) {
  typeof exports === 'object' && typeof module !== 'undefined' ? factory(exports,
require('react')) :
  typeof define === 'function' && define.amd ? define(['exports', 'react'], factory) :
  (global = global || self, factory(global.ReactDOM = {}, global.React));
}(this, (function (exports, React) { 'use strict';

  var ReactSharedInternals = React.__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED;

  var suppressWarning = false;
  function setSuppressWarning(newSuppressWarning) {
    {
      suppressWarning = newSuppressWarning;
    }
  } // In DEV, calls to console.warn and console.error get replaced
  // by calls to these methods by a Babel plugin.
  //
  // In PROD (or in packages without access to React internals),
  // they are left as they are instead.

  function warn(format) {
    {
      if (!suppressWarning) {
        for (var _len = arguments.length, args = new Array(_len > 1 ? _len - 1 : 0), _key
= 1; _key < _len; _key++) {
          args[_key - 1] = arguments[_key];
        }

        printWarning('warn', format, args);
      }
    }
  }
  function error(format) {
    {
      if (!suppressWarning) {
        for (var _len2 = arguments.length, args = new Array(_len2 > 1 ? _len2 - 1 : 0),
_key2 = 1; _key2 < _len2; _key2++) {
          args[_key2 - 1] = arguments[_key2];
        }

        printWarning('error', format, args);
      }
    }
  }

  function printWarning(level, format, args) {
    // When changing this logic, you might want to also
    // update consoleWithStackDev.www.js as well.
    {
      var ReactDebugCurrentFrame = ReactSharedInternals.ReactDebugCurrentFrame;
      var stack = ReactDebugCurrentFrame.getStackAddendum();

      if (stack !== '') {
        format += '%s';
        args = args.concat([stack]);
      } // eslint-disable-next-line react-internal/safe-string-coercion


      var argsWithFormat = args.map(function (item) {
        return String(item);
      }); // Careful: RN currently depends on this prefix

      argsWithFormat.unshift('Warning: ' + format); // We intentionally don't use spread
(or .apply) directly because it
      // breaks IE9: https://github.com/facebook/react/issues/13610
      // eslint-disable-next-line react-internal/no-production-logging

      Function.prototype.apply.call(console[level], console, argsWithFormat);
    }
  }

  var FunctionComponent = 0;
  var ClassComponent = 1;
  var IndeterminateComponent = 2; // Before we know whether it is function or class

  var HostRoot = 3; // Root of a host tree. Could be nested inside another node.

  var HostPortal = 4; // A subtree. Could be an entry point to a different renderer.

  var HostComponent = 5;
  var HostText = 6;
  var Fragment = 7;
  var Mode = 8;
  var ContextConsumer = 9;
  var ContextProvider = 10;
  var ForwardRef = 11;
  var Profiler = 12;
  var SuspenseComponent = 13;
  var MemoComponent = 14;
  var SimpleMemoComponent = 15;
  var LazyComponent = 16;
  var IncompleteClassComponent = 17;
  var DehydratedFragment = 18;
  var SuspenseListComponent = 19;
  var ScopeComponent = 21;
  var OffscreenComponent = 22;
  var LegacyHiddenComponent = 23;
  var CacheComponent = 24;
  var TracingMarkerComponent = 25;

  // -----------------------------------------------------------------------------

  var enableClientRenderFallbackOnTextMismatch = true; // TODO: Need to review this code
one more time before landing
  // the react-reconciler package.

  var enableNewReconciler = false; // Support legacy Primer support on internal FB www

  var enableLazyContextPropagation = false; // FB-only usage. The new API has different
semantics.

  var enableLegacyHidden = false; // Enables unstable_avoidThisFallback feature in Fiber

  var enableSuspenseAvoidThisFallback = false; // Enables unstable_avoidThisFallback
feature in Fizz
  // React DOM Chopping Block
  //
  // Similar to main Chopping Block but only flags related to React DOM. These are
  // grouped because we will likely batch all of them into a single major release.
  // -----------------------------------------------------------------------------
  // Disable support for comment nodes as React DOM containers. Already disabled
  // in open source, but www codebase still relies on it. Need to remove.

  var disableCommentsAsDOMContainers = true; // Disable javascript: URL strings in href
for XSS protection.
  // and client rendering, mostly to allow JSX attributes to apply to the custom
  // element's object properties instead of only HTML attributes.
  // https://github.com/facebook/react/issues/11347
  var enableCustomElementPropertySupport = false; // Disables children for <textarea>
elements
  var warnAboutStringRefs = true; // -----------------------------------------------------
-------------------------
  // Debugging and DevTools
  // -----------------------------------------------------------------------------
  // Adds user timing marks for e.g. state updates, suspense, and work loop stuff,
  // for an experimental timeline tool.

  var enableSchedulingProfiler = true; // Helps identify side effects in render-phase
lifecycle hooks and setState

  var enableProfilerTimer = true; // Record durations for commit and passive effects
phases.

  var enableProfilerCommitHooks = true; // Phase param passed to onRender callback
differentiates between an "update" and a "cascading-update".

  var allNativeEvents = new Set();
  /**
   * Mapping from registration name to event name
   */


  var registrationNameDependencies = {};
  /**
   * Mapping from lowercase registration names to the properly cased version,
   * used to warn in the case of missing event handlers. Available
   * only in true.
   * @type {Object}
   */

  var possibleRegistrationNames =  {} ; // Trust the developer to only use
possibleRegistrationNames in true

  function registerTwoPhaseEvent(registrationName, dependencies) {
    registerDirectEvent(registrationName, dependencies);
    registerDirectEvent(registrationName + 'Capture', dependencies);
  }
  function registerDirectEvent(registrationName, dependencies) {
    {
      if (registrationNameDependencies[registrationName]) {
        error('EventRegistry: More than one plugin attempted to publish the same ' +
'registration name, `%s`.', registrationName);
      }
    }

    registrationNameDependencies[registrationName] = dependencies;

    {
      var lowerCasedName = registrationName.toLowerCase();
      possibleRegistrationNames[lowerCasedName] = registrationName;

      if (registrationName === 'onDoubleClick') {
        possibleRegistrationNames.ondblclick = registrationName;
      }
    }

    for (var i = 0; i < dependencies.length; i++) {
      allNativeEvents.add(dependencies[i]);
    }
  }

  var canUseDOM = !!(typeof window !== 'undefined' && typeof window.document !==
'undefined' && typeof window.document.createElement !== 'undefined');

  var hasOwnProperty = Object.prototype.hasOwnProperty;

  /*
```

```
   * The `'' + value` pattern (used in in perf-sensitive code) throws for Symbol
   * and Temporal.* types. See https://github.com/facebook/react/pull/22064.
   *
   * The functions in this module will throw an easier-to-understand,
   * easier-to-debug exception with a clear errors message message explaining the
   * problem. (Instead of a confusing exception thrown inside the implementation
   * of the `value` object).
   */
  // $FlowFixMe only called in DEV, so void return is not possible.
  function typeName(value) {
    {
      // toStringTag is needed for namespaced types like Temporal.Instant
      var hasToStringTag = typeof Symbol === 'function' && Symbol.toStringTag;
      var type = hasToStringTag && value[Symbol.toStringTag] || value.constructor.name ||
'Object';
      return type;
    }
  } // $FlowFixMe only called in DEV, so void return is not possible.


  function willCoercionThrow(value) {
    {
      try {
        testStringCoercion(value);
        return false;
      } catch (e) {
        return true;
      }
    }
  }

  function testStringCoercion(value) {
    // If you ended up here by following an exception call stack, here's what's
    // happened: you supplied an object or symbol value to React (as a prop, key,
    // DOM attribute, CSS property, string ref, etc.) and when React tried to
    // coerce it to a string using `'' + value`, an exception was thrown.
    //
    // The most common types that will cause this exception are `Symbol` instances
    // and Temporal objects like `Temporal.Instant`. But any object that has a
    // `valueOf` or `[Symbol.toPrimitive]` method that throws will also cause this
    // exception. (Library authors do this to prevent users from using built-in
    // numeric operators like `+` or comparison operators like `>=` because custom
    // methods are needed to perform accurate arithmetic or comparison.)
    //
    // To fix the problem, coerce this object or symbol value to a string before
    // passing it to React. The most reliable way is usually `String(value)`.
    //
    // To find which value is throwing, check the browser or debugger console.
    // Before this exception was thrown, there should be `console.error` output
    // that shows the type (Symbol, Temporal.PlainDate, etc.) that caused the
    // problem and how that type was used: key, atrribute, input value prop, etc.
    // In most cases, this console output also shows the component and its
    // ancestor components where the exception happened.
    //
    // eslint-disable-next-line react-internal/safe-string-coercion
    return '' + value;
  }

  function checkAttributeStringCoercion(value, attributeName) {
    {
      if (willCoercionThrow(value)) {
        error('The provided `%s` attribute is an unsupported type %s.' + ' This value
must be coerced to a string before before using it here.', attributeName,
typeName(value));

        return testStringCoercion(value); // throw (to help callers find troubleshooting
comments)
      }
    }
  }
  function checkKeyStringCoercion(value) {
    {
      if (willCoercionThrow(value)) {
        error('The provided key is an unsupported type %s.' + ' This value must be
coerced to a string before before using it here.', typeName(value));

        return testStringCoercion(value); // throw (to help callers find troubleshooting
comments)
      }
    }
  }
  function checkPropStringCoercion(value, propName) {
    {
      if (willCoercionThrow(value)) {
        error('The provided `%s` prop is an unsupported type %s.' + ' This value must be
coerced to a string before before using it here.', propName, typeName(value));

        return testStringCoercion(value); // throw (to help callers find troubleshooting
comments)
      }
    }
  }
  function checkCSSPropertyStringCoercion(value, propName) {
    {
      if (willCoercionThrow(value)) {
        error('The provided `%s` CSS property is an unsupported type %s.' + ' This value
must be coerced to a string before before using it here.', propName, typeName(value));

        return testStringCoercion(value); // throw (to help callers find troubleshooting
comments)
      }
    }
  }
  function checkHtmlStringCoercion(value) {
    {
      if (willCoercionThrow(value)) {
        error('The provided HTML markup uses a value of unsupported type %s.' + ' This
value must be coerced to a string before before using it here.', typeName(value));

        return testStringCoercion(value); // throw (to help callers find troubleshooting
comments)
      }
    }
  }
  function checkFormFieldValueStringCoercion(value) {
    {
      if (willCoercionThrow(value)) {
        error('Form field values (value, checked, defaultValue, or defaultChecked props)'
+ ' must be strings, not %s.' + ' This value must be coerced to a string before before
using it here.', typeName(value));

        return testStringCoercion(value); // throw (to help callers find troubleshooting
comments)
      }
    }
  }

  // A reserved attribute.
  // It is handled by React separately and shouldn't be written to the DOM.
  var RESERVED = 0; // A simple string attribute.
  // Attributes that aren't in the filter are presumed to have this type.

  var STRING = 1; // A string attribute that accepts booleans in React. In HTML, these
are called
  // "enumerated" attributes with "true" and "false" as possible values.
  // When true, it should be set to a "true" string.
  // When false, it should be set to a "false" string.
  var BOOLEANISH_STRING = 2; // A real boolean attribute.
  // When true, it should be present (set either to an empty string or its name).
  // When false, it should be omitted.

  var BOOLEAN = 3; // An attribute that can be used as a flag as well as with a value.
  // When true, it should be present (set either to an empty string or its name).
  // When false, it should be omitted.
  // For any other value, should be present with that value.

  var OVERLOADED_BOOLEAN = 4; // An attribute that must be numeric or parse as a numeric.
  // When falsy, it should be removed.

  var NUMERIC = 5; // An attribute that must be positive numeric or parse as a positive
numeric.
  // When falsy, it should be removed.

  var POSITIVE_NUMERIC = 6;

  /* eslint-disable max-len */
  var ATTRIBUTE_NAME_START_CHAR = ":A-Z_a-z\\u00C0-\\u00D6\\u00D8-\\u00F6\\u00F8-
\\u02FF\\u0370-\\u037D\\u037F-\\u1FFF\\u200C-\\u200D\\u2070-\\u218F\\u2C00-
\\u2FEF\\u3001-\\uD7FF\\uF900-\\uFDCF\\uFDF0-\\uFFFD";
  /* eslint-enable max-len */

  var ATTRIBUTE_NAME_CHAR = ATTRIBUTE_NAME_START_CHAR + "\\-.0-9\\u00B7\\u0300-
\\u036F\\u203F-\\u2040";
  var VALID_ATTRIBUTE_NAME_REGEX = new RegExp('^[' + ATTRIBUTE_NAME_START_CHAR + '][' +
ATTRIBUTE_NAME_CHAR + ']*$');
  var illegalAttributeNameCache = {};
  var validatedAttributeNameCache = {};
  function isAttributeNameSafe(attributeName) {
    if (hasOwnProperty.call(validatedAttributeNameCache, attributeName)) {
      return true;
    }

    if (hasOwnProperty.call(illegalAttributeNameCache, attributeName)) {
      return false;
    }

    if (VALID_ATTRIBUTE_NAME_REGEX.test(attributeName)) {
      validatedAttributeNameCache[attributeName] = true;
      return true;
    }

    illegalAttributeNameCache[attributeName] = true;

    {
      error('Invalid attribute name: `%s`', attributeName);
    }

    return false;
  }
  function shouldIgnoreAttribute(name, propertyInfo, isCustomComponentTag) {
    if (propertyInfo !== null) {
      return propertyInfo.type === RESERVED;
    }

    if (isCustomComponentTag) {
      return false;
    }

    if (name.length > 2 && (name[0] === 'o' || name[0] === 'O') && (name[1] === 'n' ||
name[1] === 'N')) {
      return true;
    }

    return false;
  }
  function shouldRemoveAttributeWithWarning(name, value, propertyInfo,
isCustomComponentTag) {
    if (propertyInfo !== null && propertyInfo.type === RESERVED) {
      return false;
    }

    switch (typeof value) {
      case 'function': // $FlowIssue symbol is perfectly valid here

      case 'symbol':
        // eslint-disable-line
        return true;

      case 'boolean':
        {
          if (isCustomComponentTag) {
            return false;
          }

          if (propertyInfo !== null) {
            return !propertyInfo.acceptsBooleans;
          } else {
            var prefix = name.toLowerCase().slice(0, 5);
            return prefix !== 'data-' && prefix !== 'aria-';
          }
        }

      default:
        return false;
    }
  }
  function shouldRemoveAttribute(name, value, propertyInfo, isCustomComponentTag) {
    if (value === null || typeof value === 'undefined') {
      return true;
    }

    if (shouldRemoveAttributeWithWarning(name, value, propertyInfo,
isCustomComponentTag)) {
      return true;
    }

    if (isCustomComponentTag) {

      return false;
    }

    if (propertyInfo !== null) {

      switch (propertyInfo.type) {
        case BOOLEAN:
          return !value;

        case OVERLOADED_BOOLEAN:
          return value === false;

        case NUMERIC:
          return isNaN(value);

        case POSITIVE_NUMERIC:
          return isNaN(value) || value < 1;
      }
    }

    return false;
  }
  function getPropertyInfo(name) {
    return properties.hasOwnProperty(name) ? properties[name] : null;
  }

  function PropertyInfoRecord(name, type, mustUseProperty, attributeName,
attributeNamespace, sanitizeURL, removeEmptyString) {
    this.acceptsBooleans = type === BOOLEANISH_STRING || type === BOOLEAN || type ===
OVERLOADED_BOOLEAN;
    this.attributeName = attributeName;
    this.attributeNamespace = attributeNamespace;
    this.mustUseProperty = mustUseProperty;
    this.propertyName = name;
    this.type = type;
    this.sanitizeURL = sanitizeURL;
    this.removeEmptyString = removeEmptyString;
  } // When adding attributes to this list, be sure to also add them to
  // the `possibleStandardNames` module to ensure casing and incorrect
  // name warnings.


  var properties = {}; // These props are reserved by React. They shouldn't be written to
the DOM.

  var reservedProps = ['children', 'dangerouslySetInnerHTML', // TODO: This prevents the
assignment of defaultValue to regular
  // elements (not just inputs). Now that ReactDOMInput assigns to the
  // defaultValue property -- do we need this?
  'defaultValue', 'defaultChecked', 'innerHTML', 'suppressContentEditableWarning',
'suppressHydrationWarning', 'style'];

  reservedProps.forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, RESERVED, false, // mustUseProperty
    name, // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // A few React string attributes have a different name.
  // This is a mapping from React prop names to the attribute names.

  [['acceptCharset', 'accept-charset'], ['className', 'class'], ['htmlFor', 'for'],
['httpEquiv', 'http-equiv']].forEach(function (_ref) {
    var name = _ref[0],
        attributeName = _ref[1];
    properties[name] = new PropertyInfoRecord(name, STRING, false, // mustUseProperty
    attributeName, // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These are "enumerated" HTML attributes that accept "true" and "false".
  // In React, we let users pass `true` and `false` even though technically
  // these aren't boolean attributes (they are coerced to strings).

  ['contentEditable', 'draggable', 'spellCheck', 'value'].forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, BOOLEANISH_STRING, false, //
mustUseProperty
    name.toLowerCase(), // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These are "enumerated" SVG attributes that accept "true" and "false".
  // In React, we let users pass `true` and `false` even though technically
  // these aren't boolean attributes (they are coerced to strings).
  // Since these are SVG attributes, their attribute names are case-sensitive.

  ['autoReverse', 'externalResourcesRequired', 'focusable',
'preserveAlpha'].forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, BOOLEANISH_STRING, false, //
mustUseProperty
    name, // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These are HTML boolean attributes.
  ['allowFullScreen', 'async', // Note: there is a special case that prevents it from
being written to the DOM
  // on the client side because the browsers are inconsistent. Instead we call focus().
  'autoFocus', 'autoPlay', 'controls', 'default', 'defer', 'disabled',
'disablePictureInPicture', 'disableRemotePlayback', 'formNoValidate', 'hidden', 'loop',
'noModule', 'noValidate', 'open', 'playsInline', 'readOnly', 'required', 'reversed',
'scoped', 'seamless', // Microdata
  'itemScope'].forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, BOOLEAN, false, // mustUseProperty
    name.toLowerCase(), // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These are the few React props that we set as DOM properties
  // rather than attributes. These are all booleans.

  ['checked', // Note: `option.selected` is not updated if `select.multiple` is
  // disabled with `removeAttribute`. We have special logic for handling this.
  'multiple', 'muted', 'selected' // NOTE: if you add a camelCased prop to this list,
  // you'll need to set attributeName to name.toLowerCase()
  // instead in the assignment below.
  ].forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, BOOLEAN, true, // mustUseProperty
    name, // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These are HTML attributes that are "overloaded booleans": they behave like
  // booleans, but can also accept a string value.

  ['capture', 'download' // NOTE: if you add a camelCased prop to this list,
  // you'll need to set attributeName to name.toLowerCase()
  // instead in the assignment below.
  ].forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, OVERLOADED_BOOLEAN, false, //
mustUseProperty
    name, // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These are HTML attributes that must be positive numbers.

  ['cols', 'rows', 'size', 'span' // NOTE: if you add a camelCased prop to this list,
  // you'll need to set attributeName to name.toLowerCase()
  // instead in the assignment below.
  ].forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, POSITIVE_NUMERIC, false, //
mustUseProperty
    name, // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These are HTML attributes that must be numbers.

  ['rowSpan', 'start'].forEach(function (name) {
    properties[name] = new PropertyInfoRecord(name, NUMERIC, false, // mustUseProperty
    name.toLowerCase(), // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  });
  var CAMELIZE = /[\-\:]([a-z])/g;

  var capitalize = function (token) {
    return token[1].toUpperCase();
  }; // This is a list of all SVG attributes that need special casing, namespacing,
  // or boolean value assignment. Regular attributes that just accept strings
  // and have the same names are omitted, just like in the HTML attribute filter.
  // Some of these attributes can be hard to find. This list was created by
  // scraping the MDN documentation.


  ['accent-height', 'alignment-baseline', 'arabic-form', 'baseline-shift', 'cap-height',
'clip-path', 'clip-rule', 'color-interpolation', 'color-interpolation-filters', 'color-
profile', 'color-rendering', 'dominant-baseline', 'enable-background', 'fill-opacity',
'fill-rule', 'flood-color', 'flood-opacity', 'font-family', 'font-size', 'font-size-
adjust', 'font-stretch', 'font-style', 'font-variant', 'font-weight', 'glyph-name',
'glyph-orientation-horizontal', 'glyph-orientation-vertical', 'horiz-adv-x', 'horiz-
origin-x', 'image-rendering', 'letter-spacing', 'lighting-color', 'marker-end', 'marker-
mid', 'marker-start', 'overline-position', 'overline-thickness', 'paint-order', 'panose-
1', 'pointer-events', 'rendering-intent', 'shape-rendering', 'stop-color', 'stop-
opacity', 'strikethrough-position', 'strikethrough-thickness', 'stroke-dasharray',
'stroke-dashoffset', 'stroke-linecap', 'stroke-linejoin', 'stroke-miterlimit', 'stroke-
opacity', 'stroke-width', 'text-anchor', 'text-decoration', 'text-rendering', 'underline-
position', 'underline-thickness', 'unicode-bidi', 'unicode-range', 'units-per-em', 'v-
alphabetic', 'v-hanging', 'v-ideographic', 'v-mathematical', 'vector-effect', 'vert-adv-
y', 'vert-origin-x', 'vert-origin-y', 'word-spacing', 'writing-mode', 'xmlns:xlink', 'x-
height' // NOTE: if you add a camelCased prop to this list,
  // you'll need to set attributeName to name.toLowerCase()
  // instead in the assignment below.
  ].forEach(function (attributeName) {
    var name = attributeName.replace(CAMELIZE, capitalize);
    properties[name] = new PropertyInfoRecord(name, STRING, false, // mustUseProperty
    attributeName, null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // String SVG attributes with the xlink namespace.

  ['xlink:actuate', 'xlink:arcrole', 'xlink:role', 'xlink:show', 'xlink:title',
'xlink:type' // NOTE: if you add a camelCased prop to this list,
  // you'll need to set attributeName to name.toLowerCase()
  // instead in the assignment below.
  ].forEach(function (attributeName) {
    var name = attributeName.replace(CAMELIZE, capitalize);
    properties[name] = new PropertyInfoRecord(name, STRING, false, // mustUseProperty
    attributeName, 'http://www.w3.org/1999/xlink', false, // sanitizeURL
    false);
  }); // String SVG attributes with the xml namespace.

  ['xml:base', 'xml:lang', 'xml:space' // NOTE: if you add a camelCased prop to this
list,
  // you'll need to set attributeName to name.toLowerCase()
  // instead in the assignment below.
  ].forEach(function (attributeName) {
    var name = attributeName.replace(CAMELIZE, capitalize);
    properties[name] = new PropertyInfoRecord(name, STRING, false, // mustUseProperty
    attributeName, 'http://www.w3.org/XML/1998/namespace', false, // sanitizeURL
    false);
  }); // These attribute exists both in HTML and SVG.
  // The attribute name is case-sensitive in SVG so we can't just use
  // the React name like we do for attributes that exist only in HTML.

  ['tabIndex', 'crossOrigin'].forEach(function (attributeName) {
    properties[attributeName] = new PropertyInfoRecord(attributeName, STRING, false, //
mustUseProperty
    attributeName.toLowerCase(), // attributeName
    null, // attributeNamespace
    false, // sanitizeURL
    false);
  }); // These attributes accept URLs. These must not allow javascript: URLS.
  // These will also need to accept Trusted Types object in the future.

  var xlinkHref = 'xlinkHref';
  properties[xlinkHref] = new PropertyInfoRecord('xlinkHref', STRING, false, //
mustUseProperty
  'xlink:href', 'http://www.w3.org/1999/xlink', true, // sanitizeURL
  false);
  ['src', 'href', 'action', 'formAction'].forEach(function (attributeName) {
    properties[attributeName] = new PropertyInfoRecord(attributeName, STRING, false, //
mustUseProperty
    attributeName.toLowerCase(), // attributeName
    null, // attributeNamespace
    true, // sanitizeURL
    true);
  });

  // and any newline or tab are filtered out as if they're not part of the URL.
  // https://url.spec.whatwg.org/#url-parsing
  // Tab or newline are defined as \r\n\t:
  // https://infra.spec.whatwg.org/#ascii-tab-or-newline
  // A C0 control is a code point in the range \u0000 NULL to \u001F
  // INFORMATION SEPARATOR ONE, inclusive:
  // https://infra.spec.whatwg.org/#c0-control-or-space

  /* eslint-disable max-len */

  var isJavaScriptProtocol = /^[\u0000-\u001F
]*j[\r\n\t]*a[\r\n\t]*v[\r\n\t]*a[\r\n\t]*s[\r\n\t]*c[\r\n\t]*r[\r\n\t]*i[\r\n\t]*p[\r\n\
t]*t[\r\n\t]*\:/i;
  var didWarn = false;

  function sanitizeURL(url) {
    {
      if (!didWarn && isJavaScriptProtocol.test(url)) {
        didWarn = true;

        error('A future version of React will block javascript: URLs as a security
precaution. ' + 'Use event handlers instead if you can. If you need to generate unsafe
HTML try ' + 'using dangerouslySetInnerHTML instead. React was passed %s.',
JSON.stringify(url));
      }
    }
  }

  /**
   * Get the value for a property on a node. Only used in DEV for SSR validation.
   * The "expected" argument is used as a hint of what the expected value is.
   * Some properties have multiple equivalent values.
   */
  function getValueForProperty(node, name, expected, propertyInfo) {
    {
      if (propertyInfo.mustUseProperty) {
        var propertyName = propertyInfo.propertyName;
        return node[propertyName];
      } else {
        // This check protects multiple uses of `expected`, which is why the
        // react-internal/safe-string-coercion rule is disabled in several spots
        // below.
        {
          checkAttributeStringCoercion(expected, name);
        }

        if ( propertyInfo.sanitizeURL) {
          // If we haven't fully disabled javascript: URLs, and if
          // the hydration is successful of a javascript: URL, we
          // still want to warn on the client.
          // eslint-disable-next-line react-internal/safe-string-coercion
          sanitizeURL('' + expected);
        }

        var attributeName = propertyInfo.attributeName;
        var stringValue = null;

        if (propertyInfo.type === OVERLOADED_BOOLEAN) {
          if (node.hasAttribute(attributeName)) {
            var value = node.getAttribute(attributeName);
            if (value === '') {
              return true;
            }

            if (shouldRemoveAttribute(name, expected, propertyInfo, false)) {
              return value;
            } // eslint-disable-next-line react-internal/safe-string-coercion


            if (value === '' + expected) {
              return expected;
            }

            return value;
          }
        } else if (node.hasAttribute(attributeName)) {
          if (shouldRemoveAttribute(name, expected, propertyInfo, false)) {
            // We had an attribute but shouldn't have had one, so read it
            // for the error message.
            return node.getAttribute(attributeName);
          }

          if (propertyInfo.type === BOOLEAN) {
            // If this was a boolean, it doesn't matter what the value is
            // the fact that we have it is the same as the expected.
            return expected;
          } // Even if this property uses a namespace we use getAttribute
          // because we assume its namespaced name is the same as our config.
          // To use getAttributeNS we need the local name which we don't have
          // in our config atm.


          stringValue = node.getAttribute(attributeName);
        }

        if (shouldRemoveAttribute(name, expected, propertyInfo, false)) {
          return stringValue === null ? expected : stringValue; // eslint-disable-next-
line react-internal/safe-string-coercion
        } else if (stringValue === '' + expected) {
          return expected;
        } else {
          return stringValue;
        }
      }
    }
  }
  /**
   * Get the value for a attribute on a node. Only used in DEV for SSR validation.
   * The third argument is used as a hint of what the expected value is. Some
   * attributes have multiple equivalent values.
   */

  function getValueForAttribute(node, name, expected, isCustomComponentTag) {
    {
      if (!isAttributeNameSafe(name)) {
        return;
      }

      if (!node.hasAttribute(name)) {
        return expected === undefined ? undefined : null;
      }

      var value = node.getAttribute(name);

      {
        checkAttributeStringCoercion(expected, name);
      }

      if (value === '' + expected) {
        return expected;
      }

      return value;
    }
  }
  /**
   * Sets the value for a property on a node.
   *
   * @param {DOMElement} node
   * @param {string} name
   * @param {*} value
   */

  function setValueForProperty(node, name, value, isCustomComponentTag) {
    var propertyInfo = getPropertyInfo(name);

    if (shouldIgnoreAttribute(name, propertyInfo, isCustomComponentTag)) {
      return;
    }

    if (shouldRemoveAttribute(name, value, propertyInfo, isCustomComponentTag)) {
      value = null;
    }


    if (isCustomComponentTag || propertyInfo === null) {
      if (isAttributeNameSafe(name)) {
        var _attributeName = name;

        if (value === null) {
          node.removeAttribute(_attributeName);
        } else {
          {
            checkAttributeStringCoercion(value, name);
          }

          node.setAttribute(_attributeName,  '' + value);
        }
      }

      return;
    }

    var mustUseProperty = propertyInfo.mustUseProperty;

    if (mustUseProperty) {
      var propertyName = propertyInfo.propertyName;

      if (value === null) {
        var type = propertyInfo.type;
        node[propertyName] = type === BOOLEAN ? false : '';
      } else {
        // Contrary to `setAttribute`, object properties are properly
        // `toString`ed by IE8/9.
        node[propertyName] = value;
      }

      return;
    } // The rest are treated as attributes with special cases.


    var attributeName = propertyInfo.attributeName,
        attributeNamespace = propertyInfo.attributeNamespace;

    if (value === null) {
      node.removeAttribute(attributeName);
    } else {
      var _type = propertyInfo.type;
      var attributeValue;

      if (_type === BOOLEAN || _type === OVERLOADED_BOOLEAN && value === true) {
        // If attribute type is boolean, we know for sure it won't be an execution sink
        // and we won't require Trusted Type here.
        attributeValue = '';
      } else {
        // `setAttribute` with objects becomes only `[object]` in IE8/9,
        // ('' + value) makes it output the correct toString()-value.
        {
          {
            checkAttributeStringCoercion(value, attributeName);
          }

          attributeValue = '' + value;
        }

        if (propertyInfo.sanitizeURL) {
          sanitizeURL(attributeValue.toString());
        }
      }

      if (attributeNamespace) {
        node.setAttributeNS(attributeNamespace, attributeName, attributeValue);
      } else {
        node.setAttribute(attributeName, attributeValue);
      }
    }
  }

  // ATTENTION
  // When adding new symbols to this file,
  // Please consider also adding to 'react-devtools-shared/src/backend/ReactSymbols'
  // The Symbol used to tag the ReactElement-like types.
  var REACT_ELEMENT_TYPE = Symbol.for('react.element');
  var REACT_PORTAL_TYPE = Symbol.for('react.portal');
  var REACT_FRAGMENT_TYPE = Symbol.for('react.fragment');
  var REACT_STRICT_MODE_TYPE = Symbol.for('react.strict_mode');
  var REACT_PROFILER_TYPE = Symbol.for('react.profiler');
  var REACT_PROVIDER_TYPE = Symbol.for('react.provider');
  var REACT_CONTEXT_TYPE = Symbol.for('react.context');
  var REACT_FORWARD_REF_TYPE = Symbol.for('react.forward_ref');
  var REACT_SUSPENSE_TYPE = Symbol.for('react.suspense');
  var REACT_SUSPENSE_LIST_TYPE = Symbol.for('react.suspense_list');
  var REACT_MEMO_TYPE = Symbol.for('react.memo');
  var REACT_LAZY_TYPE = Symbol.for('react.lazy');
  var REACT_SCOPE_TYPE = Symbol.for('react.scope');
  var REACT_DEBUG_TRACING_MODE_TYPE = Symbol.for('react.debug_trace_mode');
  var REACT_OFFSCREEN_TYPE = Symbol.for('react.offscreen');
  var REACT_LEGACY_HIDDEN_TYPE = Symbol.for('react.legacy_hidden');
  var REACT_CACHE_TYPE = Symbol.for('react.cache');
  var REACT_TRACING_MARKER_TYPE = Symbol.for('react.tracing_marker');
  var MAYBE_ITERATOR_SYMBOL = Symbol.iterator;
  var FAUX_ITERATOR_SYMBOL = '@@iterator';
  function getIteratorFn(maybeIterable) {
    if (maybeIterable === null || typeof maybeIterable !== 'object') {
      return null;
    }

    var maybeIterator = MAYBE_ITERATOR_SYMBOL && maybeIterable[MAYBE_ITERATOR_SYMBOL] ||
maybeIterable[FAUX_ITERATOR_SYMBOL];

    if (typeof maybeIterator === 'function') {
      return maybeIterator;
    }

    return null;
  }
  var assign = Object.assign;

  // Helpers to patch console.logs to avoid logging during side-effect free
  // replaying on render function. This currently only patches the object
  // lazily which won't cover if the log function was extracted eagerly.
  // We could also eagerly patch the method.
  var disabledDepth = 0;
  var prevLog;
  var prevInfo;
  var prevWarn;
  var prevError;
  var prevGroup;
  var prevGroupCollapsed;
  var prevGroupEnd;

  function disabledLog() {}

  disabledLog.__reactDisabledLog = true;
  function disableLogs() {
    {
      if (disabledDepth === 0) {
        /* eslint-disable react-internal/no-production-logging */
        prevLog = console.log;
        prevInfo = console.info;
        prevWarn = console.warn;
        prevError = console.error;
        prevGroup = console.group;
        prevGroupCollapsed = console.groupCollapsed;
        prevGroupEnd = console.groupEnd; //
https://github.com/facebook/react/issues/19099

        var props = {
          configurable: true,
          enumerable: true,
          value: disabledLog,
          writable: true
        }; // $FlowFixMe Flow thinks console is immutable.

        Object.defineProperties(console, {
          info: props,
          log: props,
          warn: props,
          error: props,
          group: props,
          groupCollapsed: props,
          groupEnd: props
        });
        /* eslint-enable react-internal/no-production-logging */
      }

      disabledDepth++;
    }
  }
  function reenableLogs() {
    {
      disabledDepth--;

      if (disabledDepth === 0) {
        /* eslint-disable react-internal/no-production-logging */
        var props = {
          configurable: true,
          enumerable: true,
          writable: true
        }; // $FlowFixMe Flow thinks console is immutable.

        Object.defineProperties(console, {
          log: assign({}, props, {
            value: prevLog
          }),
          info: assign({}, props, {
            value: prevInfo
          }),
          warn: assign({}, props, {
            value: prevWarn
          }),
          error: assign({}, props, {
            value: prevError
          }),
          group: assign({}, props, {
            value: prevGroup
          }),
          groupCollapsed: assign({}, props, {
            value: prevGroupCollapsed
          }),
          groupEnd: assign({}, props, {
            value: prevGroupEnd
          })
        });
        /* eslint-enable react-internal/no-production-logging */
      }

      if (disabledDepth < 0) {
        error('disabledDepth fell below zero. ' + 'This is a bug in React. Please file an
issue.');
      }
    }
  }

  var ReactCurrentDispatcher = ReactSharedInternals.ReactCurrentDispatcher;
  var prefix;
  function describeBuiltInComponentFrame(name, source, ownerFn) {
    {
      if (prefix === undefined) {
        // Extract the VM specific prefix used by each line.
        try {
          throw Error();
        } catch (x) {
          var match = x.stack.trim().match(/\n( *(at )?)/);
          prefix = match && match[1] || '';
        }
      } // We use the prefix to ensure our stacks line up with native stack frames.


      return '\n' + prefix + name;
    }
  }
  var reentry = false;
  var componentFrameCache;

  {
    var PossiblyWeakMap = typeof WeakMap === 'function' ? WeakMap : Map;
    componentFrameCache = new PossiblyWeakMap();
  }

  function describeNativeComponentFrame(fn, construct) {
    // If something asked for a stack inside a fake render, it should get ignored.
    if ( !fn || reentry) {
      return '';
    }

    {
      var frame = componentFrameCache.get(fn);

      if (frame !== undefined) {
        return frame;
      }
    }
    var control;
    reentry = true;
    var previousPrepareStackTrace = Error.prepareStackTrace; // $FlowFixMe It does accept
undefined.

    Error.prepareStackTrace = undefined;
    var previousDispatcher;

    {
      previousDispatcher = ReactCurrentDispatcher.current; // Set the dispatcher in DEV
because this might be call in the render function
      // for warnings.

      ReactCurrentDispatcher.current = null;
      disableLogs();
    }

    try {
      // This should throw.
      if (construct) {
        // Something should be setting the props in the constructor.
        var Fake = function () {
          throw Error();
        }; // $FlowFixMe


        Object.defineProperty(Fake.prototype, 'props', {
          set: function () {
            // We use a throwing setter instead of frozen or non-writable props
            // because that won't throw in a non-strict mode function.
            throw Error();
          }
        });

        if (typeof Reflect === 'object' && Reflect.construct) {
          // We construct a different control for this case to include any extra
          // frames added by the construct call.
          try {
            Reflect.construct(Fake, []);
          } catch (x) {
            control = x;
          }

          Reflect.construct(fn, [], Fake);
        } else {
          try {
            Fake.call();
          } catch (x) {
            control = x;
          }

          fn.call(Fake.prototype);
        }
      } else {
        try {
          throw Error();
        } catch (x) {
          control = x;
        }

        fn();
      }
    } catch (sample) {
      // This is inlined manually because closure doesn't do it for us.
      if (sample && control && typeof sample.stack === 'string') {
        // This extracts the first frame from the sample that isn't also in the control.
        // Skipping one frame that we assume is the frame that calls the two.
        var sampleLines = sample.stack.split('\n');
        var controlLines = control.stack.split('\n');
        var s = sampleLines.length - 1;
        var c = controlLines.length - 1;

        while (s >= 1 && c >= 0 && sampleLines[s] !== controlLines[c]) {
          // We expect at least one stack frame to be shared.
          // Typically this will be the root most one. However, stack frames may be
          // cut off due to maximum stack limits. In this case, one maybe cut off
          // earlier than the other. We assume that the sample is longer or the same
          // and there for cut off earlier. So we should find the root most frame in
          // the sample somewhere in the control.
          c--;
        }

        for (; s >= 1 && c >= 0; s--, c--) {
          // Next we find the first one that isn't the same which should be the
          // frame that called our sample function and the control.
          if (sampleLines[s] !== controlLines[c]) {
            // In V8, the first line is describing the message but other VMs don't.
            // If we're about to return the first line, and the control is also on the
same
            // line, that's a pretty good indicator that our sample threw at same line as
            // the control. I.e. before we entered the sample frame. So we ignore this
result.
            // This can happen if you passed a class to function component, or non-
function.
            if (s !== 1 || c !== 1) {
              do {
                s--;
                c--; // We may still have similar intermediate frames from the construct
call.
                // The next one that isn't the same should be our match though.

                if (c < 0 || sampleLines[s] !== controlLines[c]) {
                  // V8 adds a "new" prefix for native classes. Let's remove it to make
it prettier.
                  var _frame = '\n' + sampleLines[s].replace(' at new ', ' at '); // If
our component frame is labeled "<anonymous>"
                  // but we have a user-provided "displayName"
                  // splice it in to make the stack more readable.


                  if (fn.displayName && _frame.includes('<anonymous>')) {
                    _frame = _frame.replace('<anonymous>', fn.displayName);
                  }

                  {
                    if (typeof fn === 'function') {
                      componentFrameCache.set(fn, _frame);
                    }
                  } // Return the line we found.


                  return _frame;
                }
              } while (s >= 1 && c >= 0);
            }

            break;
          }
        }
      }
    } finally {
      reentry = false;

      {
        ReactCurrentDispatcher.current = previousDispatcher;
        reenableLogs();
      }
```

```
        Error.prepareStackTrace = previousPrepareStackTrace;
      } // Fallback to just using the name if we couldn't make it throw.


      var name = fn ? fn.displayName || fn.name : '';
      var syntheticFrame = name ? describeBuiltInComponentFrame(name) : '';

      {
        if (typeof fn === 'function') {
          componentFrameCache.set(fn, syntheticFrame);
        }
      }

      return syntheticFrame;
    }

    function describeClassComponentFrame(ctor, source, ownerFn) {
      {
        return describeNativeComponentFrame(ctor, true);
      }
    }
    function describeFunctionComponentFrame(fn, source, ownerFn) {
      {
        return describeNativeComponentFrame(fn, false);
      }
    }

    function shouldConstruct(Component) {
      var prototype = Component.prototype;
      return !!(prototype && prototype.isReactComponent);
    }

    function describeUnknownElementTypeFrameInDEV(type, source, ownerFn) {

      if (type == null) {
        return '';
      }

      if (typeof type === 'function') {
        {
          return describeNativeComponentFrame(type, shouldConstruct(type));
        }
      }

      if (typeof type === 'string') {
        return describeBuiltInComponentFrame(type);
      }

      switch (type) {
        case REACT_SUSPENSE_TYPE:
          return describeBuiltInComponentFrame('Suspense');

        case REACT_SUSPENSE_LIST_TYPE:
          return describeBuiltInComponentFrame('SuspenseList');
      }

      if (typeof type === 'object') {
        switch (type.$$typeof) {
          case REACT_FORWARD_REF_TYPE:
            return describeFunctionComponentFrame(type.render);

          case REACT_MEMO_TYPE:
            // Memo may contain any component type so we recursively resolve it.
            return describeUnknownElementTypeFrameInDEV(type.type, source, ownerFn);

          case REACT_LAZY_TYPE:
            {
              var lazyComponent = type;
              var payload = lazyComponent._payload;
            var init = lazyComponent._init;

            try {
              // Lazy may contain any component type so we recursively resolve it.
              return describeUnknownElementTypeFrameInDEV(init(payload), source,
ownerFn);
            } catch (x) {}
          }
        }
      }

      return '';
    }

    function describeFiber(fiber) {
      var owner =  fiber._debugOwner ? fiber._debugOwner.type : null ;
      var source =  fiber._debugSource ;

      switch (fiber.tag) {
        case HostComponent:
          return describeBuiltInComponentFrame(fiber.type);

        case LazyComponent:
          return describeBuiltInComponentFrame('Lazy');

        case SuspenseComponent:
          return describeBuiltInComponentFrame('Suspense');

        case SuspenseListComponent:
          return describeBuiltInComponentFrame('SuspenseList');

        case FunctionComponent:
        case IndeterminateComponent:
        case SimpleMemoComponent:
          return describeFunctionComponentFrame(fiber.type);

        case ForwardRef:
          return describeFunctionComponentFrame(fiber.type.render);

        case ClassComponent:
          return describeClassComponentFrame(fiber.type);

        default:
          return '';
      }
    }

    function getStackByFiberInDevAndProd(workInProgress) {
      try {
        var info = '';
        var node = workInProgress;

        do {
          info += describeFiber(node);
          node = node.return;
        } while (node);

        return info;
      } catch (x) {
        return '\nError generating stack: ' + x.message + '\n' + x.stack;
      }
    }

    function getWrappedName(outerType, innerType, wrapperName) {
      var displayName = outerType.displayName;

      if (displayName) {
        return displayName;
      }
      var functionName = innerType.displayName || innerType.name || '';
      return functionName !== '' ? wrapperName + "(" + functionName + ")" : wrapperName;
    } // Keep in sync with react-reconciler/getComponentNameFromFiber


    function getContextName(type) {
      return type.displayName || 'Context';
    } // Note that the reconciler package should generally prefer to use
  getComponentNameFromFiber() instead.


    function getComponentNameFromType(type) {
      if (type == null) {
        // Host root, text node or just invalid type.
        return null;
      }

      {
        if (typeof type.tag === 'number') {
          error('Received an unexpected object in getComponentNameFromType(). ' + 'This is
  likely a bug in React. Please file an issue.');
        }
      }

      if (typeof type === 'function') {
        return type.displayName || type.name || null;
      }

      if (typeof type === 'string') {
        return type;
      }

      switch (type) {
        case REACT_FRAGMENT_TYPE:
          return 'Fragment';

        case REACT_PORTAL_TYPE:
          return 'Portal';

        case REACT_PROFILER_TYPE:
          return 'Profiler';

        case REACT_STRICT_MODE_TYPE:
          return 'StrictMode';

        case REACT_SUSPENSE_TYPE:
          return 'Suspense';

        case REACT_SUSPENSE_LIST_TYPE:
          return 'SuspenseList';

      }

      if (typeof type === 'object') {
        switch (type.$$typeof) {
          case REACT_CONTEXT_TYPE:
            var context = type;
            return getContextName(context) + '.Consumer';

          case REACT_PROVIDER_TYPE:
            var provider = type;
            return getContextName(provider._context) + '.Provider';

          case REACT_FORWARD_REF_TYPE:
            return getWrappedName(type, type.render, 'ForwardRef');

          case REACT_MEMO_TYPE:
            var outerName = type.displayName || null;
            if (outerName !== null) {
              return outerName;
            }

            return getComponentNameFromType(type.type) || 'Memo';

          case REACT_LAZY_TYPE:
            {
              var lazyComponent = type;
              var payload = lazyComponent._payload;
              var init = lazyComponent._init;

              try {
                return getComponentNameFromType(init(payload));
              } catch (x) {
                return null;
              }
            }

          // eslint-disable-next-line no-fallthrough
        }
      }

      return null;
    }

    function getWrappedName$1(outerType, innerType, wrapperName) {
      var functionName = innerType.displayName || innerType.name || '';
      return outerType.displayName || (functionName !== '' ? wrapperName + "(" +
  functionName + ")" : wrapperName);
    } // Keep in sync with shared/getComponentNameFromType


    function getContextName$1(type) {
      return type.displayName || 'Context';
    }

    function getComponentNameFromFiber(fiber) {
      var tag = fiber.tag,
          type = fiber.type;

      switch (tag) {
        case CacheComponent:
          return 'Cache';

        case ContextConsumer:
          var context = type;
          return getContextName$1(context) + '.Consumer';

        case ContextProvider:
          var provider = type;
          return getContextName$1(provider._context) + '.Provider';

        case DehydratedFragment:
          return 'DehydratedFragment';

        case ForwardRef:
          return getWrappedName$1(type, type.render, 'ForwardRef');

        case Fragment:
          return 'Fragment';

        case HostComponent:
          // Host component type is the display name (e.g. "div", "View")
          return type;

        case HostPortal:
          return 'Portal';
        case HostRoot:
          return 'Root';

        case HostText:
          return 'Text';

        case LazyComponent:
          // Name comes from the type in this case; we don't have a tag.
          return getComponentNameFromType(type);

        case Mode:
          if (type === REACT_STRICT_MODE_TYPE) {
            // Don't be less specific than shared/getComponentNameFromType
            return 'StrictMode';
          }

          return 'Mode';

        case OffscreenComponent:
          return 'Offscreen';

        case Profiler:
          return 'Profiler';

        case ScopeComponent:
          return 'Scope';

        case SuspenseComponent:
          return 'Suspense';

        case SuspenseListComponent:
          return 'SuspenseList';

        case TracingMarkerComponent:
          return 'TracingMarker';
        // The display name for this tags come from the user-provided type:

        case ClassComponent:
        case FunctionComponent:
        case IncompleteClassComponent:
        case IndeterminateComponent:
        case MemoComponent:
        case SimpleMemoComponent:
          if (typeof type === 'function') {
            return type.displayName || type.name || null;
          }

          if (typeof type === 'string') {
            return type;
          }

          break;

      }

      return null;
    }

    var ReactDebugCurrentFrame = ReactSharedInternals.ReactDebugCurrentFrame;
    var current = null;
    var isRendering = false;
    function getCurrentFiberOwnerNameInDevOrNull() {
      {
        if (current === null) {
          return null;
        }

        var owner = current._debugOwner;
```

```
        if (owner !== null && typeof owner !== 'undefined') {
          return getComponentNameFromFiber(owner);
        }
      }

      return null;
    }

    function getCurrentFiberStackInDev() {
      {
        if (current === null) {
          return '';
        } // Safe because if current fiber exists, we are reconciling,
        // and it is guaranteed to be the work-in-progress version.


        return getStackByFiberInDevAndProd(current);
      }
    }

    function resetCurrentFiber() {
      {
        ReactDebugCurrentFrame.getCurrentStack = null;
        current = null;
        isRendering = false;
      }
    }
    function setCurrentFiber(fiber) {
      {
        ReactDebugCurrentFrame.getCurrentStack = fiber === null ? null :
  getCurrentFiberStackInDev;
        current = fiber;
        isRendering = false;
      }
    }
    function getCurrentFiber() {
      {
        return current;
      }
    }
    function setIsRendering(rendering) {
      {
        isRendering = rendering;
      }
    }

    // Flow does not allow string concatenation of most non-string types. To work
    // around this limitation, we use an opaque type that can only be obtained by
    // passing the value through getToStringValue first.
    function toString(value) {
      // The coercion safety check is performed in getToStringValue().
      // eslint-disable-next-line react-internal/safe-string-coercion
      return '' + value;
    }
    function getToStringValue(value) {
      switch (typeof value) {
        case 'boolean':
        case 'number':
        case 'string':
        case 'undefined':
          return value;

        case 'object':
          {
            checkFormFieldValueStringCoercion(value);
          }

          return value;
        default:
          // function, symbol are assigned as empty strings
          return '';
      }
    }

    var hasReadOnlyValue = {
      button: true,
      checkbox: true,
      image: true,
      hidden: true,
      radio: true,
      reset: true,
      submit: true
    };
    function checkControlledValueProps(tagName, props) {
      {
        if (!(hasReadOnlyValue[props.type] || props.onChange || props.onInput ||
  props.readOnly || props.disabled || props.value == null)) {
          error('You provided a `value` prop to a form field without an ' + '`onChange`
  handler. This will render a read-only field. If ' + 'the field should be mutable use
  `defaultValue`. Otherwise, ' + 'set either `onChange` or `readOnly`.');
        }

        if (!(props.onChange || props.readOnly || props.disabled || props.checked == null))
  {
          error('You provided a `checked` prop to a form field without an ' + '`onChange`
  handler. This will render a read-only field. If ' + 'the field should be mutable use
  `defaultChecked`. Otherwise, ' + 'set either `onChange` or `readOnly`.');
        }
      }
    }

    function isCheckable(elem) {
      var type = elem.type;
      var nodeName = elem.nodeName;
      return nodeName && nodeName.toLowerCase() === 'input' && (type === 'checkbox' || type
  === 'radio');
    }

    function getTracker(node) {
      return node._valueTracker;
    }

    function detachTracker(node) {
      node._valueTracker = null;
    }

    function getValueFromNode(node) {
      var value = '';

      if (!node) {
        return value;
      }

      if (isCheckable(node)) {
        value = node.checked ? 'true' : 'false';
      } else {
        value = node.value;
      }

      return value;
    }

    function trackValueOnNode(node) {
      var valueField = isCheckable(node) ? 'checked' : 'value';
      var descriptor = Object.getOwnPropertyDescriptor(node.constructor.prototype,
  valueField);
```

```
    {
      checkFormFieldValueStringCoercion(node[valueField]);
    }

    var currentValue = '' + node[valueField]; // if someone has already defined a value
or Safari, then bail
    // and don't track value will cause over reporting of changes,
    // but it's better then a hard failure
    // (needed for certain tests that spyOn input values and Safari)

    if (node.hasOwnProperty(valueField) || typeof descriptor === 'undefined' || typeof
descriptor.get !== 'function' || typeof descriptor.set !== 'function') {
      return;
    }

    var get = descriptor.get,
        set = descriptor.set;
    Object.defineProperty(node, valueField, {
      configurable: true,
      get: function () {
        return get.call(this);
      },
      set: function (value) {
        {
          checkFormFieldValueStringCoercion(value);
        }

        currentValue = '' + value;
        set.call(this, value);
      }
    }); // We could've passed this the first time
    // but it triggers a bug in IE11 and Edge 14/15.
    // Calling defineProperty() again should be equivalent.
    // https://github.com/facebook/react/issues/11768

    Object.defineProperty(node, valueField, {
      enumerable: descriptor.enumerable
    });
    var tracker = {
      getValue: function () {
        return currentValue;
      },
      setValue: function (value) {
        {
          checkFormFieldValueStringCoercion(value);
        }

        currentValue = '' + value;
      },
      stopTracking: function () {
        detachTracker(node);
        delete node[valueField];
      }
    };
    return tracker;
  }

  function track(node) {
    if (getTracker(node)) {
      return;
    } // TODO: Once it's just Fiber we can move this to node._wrapperState


    node._valueTracker = trackValueOnNode(node);
  }
  function updateValueIfChanged(node) {
    if (!node) {
      return false;
    }

    var tracker = getTracker(node); // if there is no tracker at this point it's unlikely
    // that trying again will succeed

    if (!tracker) {
      return true;
    }

    var lastValue = tracker.getValue();
    var nextValue = getValueFromNode(node);

    if (nextValue !== lastValue) {
      tracker.setValue(nextValue);
      return true;
    }

    return false;
  }

  function getActiveElement(doc) {
    doc = doc || (typeof document !== 'undefined' ? document : undefined);

    if (typeof doc === 'undefined') {
      return null;
    }

    try {
      return doc.activeElement || doc.body;
    } catch (e) {
      return doc.body;
    }
  }

  var didWarnValueDefaultValue = false;
  var didWarnCheckedDefaultChecked = false;
  var didWarnControlledToUncontrolled = false;
  var didWarnUncontrolledToControlled = false;

  function isControlled(props) {
    var usesChecked = props.type === 'checkbox' || props.type === 'radio';
    return usesChecked ? props.checked != null : props.value != null;
  }
  /**
   * Implements an <input> host component that allows setting these optional
   * props: `checked`, `value`, `defaultChecked`, and `defaultValue`.
   *
   * If `checked` or `value` are not supplied (or null/undefined), user actions
   * that affect the checked state or value will trigger updates to the element.
   *
   * If they are supplied (and not null/undefined), the rendered element will not
   * trigger updates to the element. Instead, the props must change in order for
   * the rendered element to be updated.
   *
   * The rendered element will be initialized as unchecked (or `defaultChecked`)
   * with an empty value (or `defaultValue`).
   *
   * See http://www.w3.org/TR/2012/WD-html5-20121025/the-input-element.html
   */


  function getHostProps(element, props) {
    var node = element;
    var checked = props.checked;
    var hostProps = assign({}, props, {
      defaultChecked: undefined,
      defaultValue: undefined,
      value: undefined,
      checked: checked != null ? checked : node._wrapperState.initialChecked
    });
    return hostProps;
  }
  function initWrapperState(element, props) {
    {
      checkControlledValueProps('input', props);

      if (props.checked !== undefined && props.defaultChecked !== undefined &&
!didWarnCheckedDefaultChecked) {
        error('%s contains an input of type %s with both checked and defaultChecked
props. ' + 'Input elements must be either controlled or uncontrolled ' + '(specify either
the checked prop, or the defaultChecked prop, but not ' + 'both). Decide between using a
controlled or uncontrolled input ' + 'element and remove one of these props. More info: '
+ 'https://reactjs.org/link/controlled-components', getCurrentFiberOwnerNameInDevOrNull()
|| 'A component', props.type);

        didWarnCheckedDefaultChecked = true;
      }

      if (props.value !== undefined && props.defaultValue !== undefined &&
!didWarnValueDefaultValue) {
        error('%s contains an input of type %s with both value and defaultValue props. '
+ 'Input elements must be either controlled or uncontrolled ' + '(specify either the
value prop, or the defaultValue prop, but not ' + 'both). Decide between using a
controlled or uncontrolled input ' + 'element and remove one of these props. More info: '
+ 'https://reactjs.org/link/controlled-components', getCurrentFiberOwnerNameInDevOrNull()
|| 'A component', props.type);

        didWarnValueDefaultValue = true;
      }
    }

    var node = element;
    var defaultValue = props.defaultValue == null ? '' : props.defaultValue;
    node._wrapperState = {
      initialChecked: props.checked != null ? props.checked : props.defaultChecked,
      initialValue: getToStringValue(props.value != null ? props.value : defaultValue),
      controlled: isControlled(props)
    };
  }
  function updateChecked(element, props) {
    var node = element;
    var checked = props.checked;

    if (checked != null) {
      setValueForProperty(node, 'checked', checked, false);
    }
  }
  function updateWrapper(element, props) {
    var node = element;

    {
      var controlled = isControlled(props);

      if (!node._wrapperState.controlled && controlled &&
!didWarnUncontrolledToControlled) {
        error('A component is changing an uncontrolled input to be controlled. ' + 'This
is likely caused by the value changing from undefined to ' + 'a defined value, which
should not happen. ' + 'Decide between using a controlled or uncontrolled input ' +
'element for the lifetime of the component. More info:
https://reactjs.org/link/controlled-components');

        didWarnUncontrolledToControlled = true;
      }

      if (node._wrapperState.controlled && !controlled &&
!didWarnControlledToUncontrolled) {
        error('A component is changing a controlled input to be uncontrolled. ' + 'This
is likely caused by the value changing from a defined to ' + 'undefined, which should not
happen. ' + 'Decide between using a controlled or uncontrolled input ' + 'element for the
lifetime of the component. More info: https://reactjs.org/link/controlled-components');

        didWarnControlledToUncontrolled = true;
      }
    }

    updateChecked(element, props);
    var value = getToStringValue(props.value);
    var type = props.type;

    if (value != null) {
      if (type === 'number') {
        if (value === 0 && node.value === '' || // We explicitly want to coerce to number
here if possible.
        // eslint-disable-next-line
        node.value != value) {
          node.value = toString(value);
        }
      } else if (node.value !== toString(value)) {
        node.value = toString(value);
      }
    } else if (type === 'submit' || type === 'reset') {
      // Submit/reset inputs need the attribute removed completely to avoid
      // blank-text buttons.
      node.removeAttribute('value');
      return;
    }

    {
      // When syncing the value attribute, the value comes from a cascade of
      // properties:
      //  1. The value React property
      //  2. The defaultValue React property
      //  3. Otherwise there should be no change
      if (props.hasOwnProperty('value')) {
        setDefaultValue(node, props.type, value);
      } else if (props.hasOwnProperty('defaultValue')) {
        setDefaultValue(node, props.type, getToStringValue(props.defaultValue));
      }
    }

    {
      // When syncing the checked attribute, it only changes when it needs
      // to be removed, such as transitioning from a checkbox into a text input
      if (props.checked == null && props.defaultChecked != null) {
        node.defaultChecked = !!props.defaultChecked;
      }
    }
  }
  function postMountWrapper(element, props, isHydrating) {
    var node = element; // Do not assign value if it is already set. This prevents user
text input
    // from being lost during SSR hydration.

    if (props.hasOwnProperty('value') || props.hasOwnProperty('defaultValue')) {
      var type = props.type;
      var isButton = type === 'submit' || type === 'reset'; // Avoid setting value
attribute on submit/reset inputs as it overrides the
      // default value provided by the browser. See: #12872

      if (isButton && (props.value === undefined || props.value === null)) {
        return;
      }

      var initialValue = toString(node._wrapperState.initialValue); // Do not assign
value if it is already set. This prevents user text input
      // from being lost during SSR hydration.
```

```
        if (!isHydrating) {
          {
            // When syncing the value attribute, the value property should use
            // the wrapperState._initialValue property. This uses:
            //
            //   1. The value React property when present
            //   2. The defaultValue React property when present
            //   3. An empty string
            if (initialValue !== node.value) {
              node.value = initialValue;
            }
          }
        }

        {
          // Otherwise, the value attribute is synchronized to the property,
          // so we assign defaultValue to the same thing as the value property
          // assignment step above.
          node.defaultValue = initialValue;
        }
      } // Normally, we'd just do `node.checked = node.checked` upon initial mount, less
this bug
      // this is needed to work around a chrome bug where setting defaultChecked
      // will sometimes influence the value of checked (even after detachment).
      // Reference: https://bugs.chromium.org/p/chromium/issues/detail?id=608416
      // We need to temporarily unset name to avoid disrupting radio button groups.


      var name = node.name;

      if (name !== '') {
        node.name = '';
      }

      {
        // When syncing the checked attribute, both the checked property and
        // attribute are assigned at the same time using defaultChecked. This uses:
        //
        //   1. The checked React property when present
        //   2. The defaultChecked React property when present
        //   3. Otherwise, false
        node.defaultChecked = !node.defaultChecked;
        node.defaultChecked = !!node._wrapperState.initialChecked;
      }

      if (name !== '') {
        node.name = name;
      }
    }
    function restoreControlledState(element, props) {
      var node = element;
      updateWrapper(node, props);
      updateNamedCousins(node, props);
    }

    function updateNamedCousins(rootNode, props) {
      var name = props.name;

      if (props.type === 'radio' && name != null) {
        var queryRoot = rootNode;

        while (queryRoot.parentNode) {
          queryRoot = queryRoot.parentNode;
        } // If `rootNode.form` was non-null, then we could try `form.elements`,
        // but that sometimes behaves strangely in IE8. We could also try using
        // `form.getElementsByName`, but that will only return direct children
        // and won't include inputs that use the HTML5 `form=` attribute. Since
        // the input might not even be in a form. It might not even be in the
        // document. Let's just use the local `querySelectorAll` to ensure we don't
        // miss anything.


        {
          checkAttributeStringCoercion(name, 'name');
        }

        var group = queryRoot.querySelectorAll('input[name=' + JSON.stringify('' + name) +
  '][type="radio"]');

        for (var i = 0; i < group.length; i++) {
          var otherNode = group[i];

          if (otherNode === rootNode || otherNode.form !== rootNode.form) {
            continue;
          } // This will throw if radio buttons rendered by different copies of React
          // and the same name are rendered into the same form (same as #1939).
          // That's probably okay; we don't support it just as we don't support
          // mixing React radio buttons with non-React ones.


          var otherProps = getFiberCurrentPropsFromNode(otherNode);

          if (!otherProps) {
            throw new Error('ReactDOMInput: Mixing React and non-React radio inputs with
  the ' + 'same `name` is not supported.');
          } // We need update the tracked value on the named cousin since the value
          // was changed but the input saw no event or value set


          updateValueIfChanged(otherNode); // If this is a controlled radio button group,
  forcing the input that
          // was previously checked to update will cause it to be come re-checked
          // as appropriate.

          updateWrapper(otherNode, otherProps);
        }
      }
    } // In Chrome, assigning defaultValue to certain input types triggers input
  validation.
    // For number inputs, the display value loses trailing decimal points. For email
  inputs,
    // Chrome raises "The specified value <x> is not a valid email address".
    //
    // Here we check to see if the defaultValue has actually changed, avoiding these
  problems
    // when the user is inputting text
    //
    // https://github.com/facebook/react/issues/7253


    function setDefaultValue(node, type, value) {
      if ( // Focused number inputs synchronize on blur. See ChangeEventPlugin.js
      type !== 'number' || getActiveElement(node.ownerDocument) !== node) {
        if (value == null) {
          node.defaultValue = toString(node._wrapperState.initialValue);
        } else if (node.defaultValue !== toString(value)) {
          node.defaultValue = toString(value);
        }
      }
    }

    var didWarnSelectedSetOnOption = false;
    var didWarnInvalidChild = false;
    var didWarnInvalidInnerHTML = false;
    /**
     * Implements an <option> host component that warns when `selected` is set.
     */
    function validateProps(element, props) {
      {
        // If a value is not provided, then the children must be simple.
        if (props.value == null) {
          if (typeof props.children === 'object' && props.children !== null) {
            React.Children.forEach(props.children, function (child) {
              if (child == null) {
                return;
              }

              if (typeof child === 'string' || typeof child === 'number') {
                return;
              }

              if (!didWarnInvalidChild) {
                didWarnInvalidChild = true;

                error('Cannot infer the option value of complex children. ' + 'Pass a
  `value` prop or use a plain string as children to <option>.');
              }
            });
          } else if (props.dangerouslySetInnerHTML != null) {
            if (!didWarnInvalidInnerHTML) {
              didWarnInvalidInnerHTML = true;

              error('Pass a `value` prop if you set dangerouslyInnerHTML so React knows ' +
  'which value should be selected.');
            }
          }
        } // TODO: Remove support for `selected` in <option>.


        if (props.selected != null && !didWarnSelectedSetOnOption) {
          error('Use the `defaultValue` or `value` props on <select> instead of ' +
  'setting `selected` on <option>.');

          didWarnSelectedSetOnOption = true;
        }
      }
    }
    function postMountWrapper$1(element, props) {
      // value="" should make a value attribute (#6219)
      if (props.value != null) {
        element.setAttribute('value', toString(getToStringValue(props.value)));
      }
    }

    var isArrayImpl = Array.isArray; // eslint-disable-next-line no-redeclare

    function isArray(a) {
      return isArrayImpl(a);
    }

    var didWarnValueDefaultValue$1;

    {
      didWarnValueDefaultValue$1 = false;
    }

    function getDeclarationErrorAddendum() {
      var ownerName = getCurrentFiberOwnerNameInDevOrNull();

      if (ownerName) {
        return '\n\nCheck the render method of `' + ownerName + '`.';
      }

      return '';
    }
```

```
    var valuePropNames = ['value', 'defaultValue'];
    /**
     * Validation function for `value` and `defaultValue`.
     */

    function checkSelectPropTypes(props) {
      {
        checkControlledValueProps('select', props);

        for (var i = 0; i < valuePropNames.length; i++) {
          var propName = valuePropNames[i];

          if (props[propName] == null) {
            continue;
          }

          var propNameIsArray = isArray(props[propName]);

          if (props.multiple && !propNameIsArray) {
            error('The `%s` prop supplied to <select> must be an array if ' + '`multiple`
  is true.%s', propName, getDeclarationErrorAddendum());
          } else if (!props.multiple && propNameIsArray) {
            error('The `%s` prop supplied to <select> must be a scalar ' + 'value if
  `multiple` is false.%s', propName, getDeclarationErrorAddendum());
          }
        }
      }
    }

    function updateOptions(node, multiple, propValue, setDefaultSelected) {
      var options = node.options;

      if (multiple) {
        var selectedValues = propValue;
        var selectedValue = {};

        for (var i = 0; i < selectedValues.length; i++) {
          // Prefix to avoid chaos with special keys.
          selectedValue['$' + selectedValues[i]] = true;
        }

        for (var _i = 0; _i < options.length; _i++) {
          var selected = selectedValue.hasOwnProperty('$' + options[_i].value);

          if (options[_i].selected !== selected) {
            options[_i].selected = selected;
          }

          if (selected && setDefaultSelected) {
            options[_i].defaultSelected = true;
          }
        }
      } else {
        // Do not set `select.value` as exact behavior isn't consistent across all
        // browsers for all cases.
        var _selectedValue = toString(getToStringValue(propValue));

        var defaultSelected = null;

        for (var _i2 = 0; _i2 < options.length; _i2++) {
          if (options[_i2].value === _selectedValue) {
            options[_i2].selected = true;

            if (setDefaultSelected) {
              options[_i2].defaultSelected = true;
            }

            return;
          }
          if (defaultSelected === null && !options[_i2].disabled) {
            defaultSelected = options[_i2];
          }
        }

        if (defaultSelected !== null) {
          defaultSelected.selected = true;
        }
      }
    }
    /**
     * Implements a <select> host component that allows optionally setting the
     * props `value` and `defaultValue`. If `multiple` is false, the prop must be a
     * stringable. If `multiple` is true, the prop must be an array of stringables.
     *
     * If `value` is not supplied (or null/undefined), user actions that change the
     * selected option will trigger updates to the rendered options.
     *
     * If it is supplied (and not null/undefined), the rendered options will not
     * update in response to user actions. Instead, the `value` prop must change in
     * order for the rendered options to update.
     *
     * If `defaultValue` is provided, any options with the supplied values will be
     * selected.
     */


    function getHostProps$1(element, props) {
      return assign({}, props, {
        value: undefined
      });
    }
    function initWrapperState$1(element, props) {
      var node = element;

      {
        checkSelectPropTypes(props);
      }

      node._wrapperState = {
        wasMultiple: !!props.multiple
      };

      {
        if (props.value !== undefined && props.defaultValue !== undefined &&
  !didWarnValueDefaultValue$1) {
          error('Select elements must be either controlled or uncontrolled ' + '(specify
  either the value prop, or the defaultValue prop, but not ' + 'both). Decide between using
  a controlled or uncontrolled select ' + 'element and remove one of these props. More
  info: ' + 'https://reactjs.org/link/controlled-components');

          didWarnValueDefaultValue$1 = true;
        }
      }
    }
    function postMountWrapper$2(element, props) {
      var node = element;
      node.multiple = !!props.multiple;
      var value = props.value;

      if (value != null) {
        updateOptions(node, !!props.multiple, value, false);
      } else if (props.defaultValue != null) {
        updateOptions(node, !!props.multiple, props.defaultValue, true);
      }
    }
    function postUpdateWrapper(element, props) {
      var node = element;
      var wasMultiple = node._wrapperState.wasMultiple;
      node._wrapperState.wasMultiple = !!props.multiple;
      var value = props.value;

      if (value != null) {
        updateOptions(node, !!props.multiple, value, false);
      } else if (wasMultiple !== !!props.multiple) {
        // For simplicity, reapply `defaultValue` if `multiple` is toggled.
        if (props.defaultValue != null) {
          updateOptions(node, !!props.multiple, props.defaultValue, true);
        } else {
          // Revert the select back to its default unselected state.
          updateOptions(node, !!props.multiple, props.multiple ? [] : '', false);
        }
      }
    }
    function restoreControlledState$1(element, props) {
      var node = element;
      var value = props.value;

      if (value != null) {
        updateOptions(node, !!props.multiple, value, false);
      }
    }

    var didWarnValDefaultVal = false;

    /**
     * Implements a <textarea> host component that allows setting `value`, and
     * `defaultValue`. This differs from the traditional DOM API because value is
     * usually set as PCDATA children.
     *
     * If `value` is not supplied (or null/undefined), user actions that affect the
     * value will trigger updates to the element.
     *
     * If `value` is supplied (and not null/undefined), the rendered element will
     * not trigger updates to the element. Instead, the `value` prop must change in
     * order for the rendered element to be updated.
     *
     * The rendered element will be initialized with an empty value, the prop
     * `defaultValue` if specified, or the children content (deprecated).
     */
    function getHostProps$2(element, props) {
      var node = element;

      if (props.dangerouslySetInnerHTML != null) {
        throw new Error('`dangerouslySetInnerHTML` does not make sense on <textarea>.');
      } // Always set children to the same thing. In IE9, the selection range will
      // get reset if `textContent` is mutated.  We could add a check in setTextContent
      // to only set the value if/when the value differs from the node value (which would
      // completely solve this IE9 bug), but Sebastian+Sophie seemed to like this
      // solution. The value can be a boolean or object so that's why it's forced
      // to be a string.


      var hostProps = assign({}, props, {
        value: undefined,
        defaultValue: undefined,
        children: toString(node._wrapperState.initialValue)
      });

      return hostProps;
    }
    function initWrapperState$2(element, props) {
      var node = element;

      {
        checkControlledValueProps('textarea', props);
        if (props.value !== undefined && props.defaultValue !== undefined &&
  !didWarnValDefaultVal) {
          error('%s contains a textarea with both value and defaultValue props. ' +
  'Textarea elements must be either controlled or uncontrolled ' + '(specify either the
  value prop, or the defaultValue prop, but not ' + 'both). Decide between using a
  controlled or uncontrolled textarea ' + 'and remove one of these props. More info: ' +
  'https://reactjs.org/link/controlled-components', getCurrentFiberOwnerNameInDevOrNull()
  || 'A component');

          didWarnValDefaultVal = true;
        }
      }

      var initialValue = props.value; // Only bother fetching default value if we're going
  to use it

      if (initialValue == null) {
        var children = props.children,
            defaultValue = props.defaultValue;

        if (children != null) {
          {
            error('Use the `defaultValue` or `value` props instead of setting ' + 'children
  on <textarea>.');
          }

          {
            if (defaultValue != null) {
              throw new Error('If you supply `defaultValue` on a <textarea>, do not pass
  children.');
            }

            if (isArray(children)) {
              if (children.length > 1) {
                throw new Error('<textarea> can only have at most one child.');
              }

              children = children[0];
            }

            defaultValue = children;
          }
        }

        if (defaultValue == null) {
          defaultValue = '';
        }

        initialValue = defaultValue;
      }

      node._wrapperState = {
        initialValue: getToStringValue(initialValue)
      };
    }
    function updateWrapper$1(element, props) {
      var node = element;
      var value = getToStringValue(props.value);
      var defaultValue = getToStringValue(props.defaultValue);

      if (value != null) {
        // Cast `value` to a string to ensure the value is set correctly. While
        // browsers typically do this as necessary, jsdom doesn't.
        var newValue = toString(value); // To avoid side effects (such as losing text
  selection), only set value if changed

        if (newValue !== node.value) {
          node.value = newValue;
        }
        if (props.defaultValue == null && node.defaultValue !== newValue) {
          node.defaultValue = newValue;
        }
      }

      if (defaultValue != null) {
        node.defaultValue = toString(defaultValue);
      }
    }
    function postMountWrapper$3(element, props) {
      var node = element; // This is in postMount because we need access to the DOM node,
  which is not
      // available until after the component has mounted.

      var textContent = node.textContent; // Only set node.value if textContent is equal to
  the expected
      // initial value. In IE10/IE11 there is a bug where the placeholder attribute
      // will populate textContent as well.
      // https://developer.microsoft.com/microsoft-edge/platform/issues/101525/

      if (textContent === node._wrapperState.initialValue) {
        if (textContent !== '' && textContent !== null) {
          node.value = textContent;
        }
      }
    }
    function restoreControlledState$2(element, props) {
      // DOM component is still mounted; update
      updateWrapper$1(element, props);
    }

    var HTML_NAMESPACE = 'http://www.w3.org/1999/xhtml';
    var MATH_NAMESPACE = 'http://www.w3.org/1998/Math/MathML';
    var SVG_NAMESPACE = 'http://www.w3.org/2000/svg'; // Assumes there is no parent
  namespace.

    function getIntrinsicNamespace(type) {
      switch (type) {
        case 'svg':
          return SVG_NAMESPACE;

        case 'math':
          return MATH_NAMESPACE;

        default:
          return HTML_NAMESPACE;
      }
    }
    function getChildNamespace(parentNamespace, type) {
      if (parentNamespace == null || parentNamespace === HTML_NAMESPACE) {
        // No (or default) parent namespace: potential entry point.
        return getIntrinsicNamespace(type);
      }

      if (parentNamespace === SVG_NAMESPACE && type === 'foreignObject') {
        // We're leaving SVG.
        return HTML_NAMESPACE;
      } // By default, pass namespace below.


      return parentNamespace;
    }

    /* globals MSApp */

    /**
     * Create a function which has 'unsafe' privileges (required by windows8 apps)
     */
```

```
    var createMicrosoftUnsafeLocalFunction = function (func) {
      if (typeof MSApp !== 'undefined' && MSApp.execUnsafeLocalFunction) {
        return function (arg0, arg1, arg2, arg3) {
          MSApp.execUnsafeLocalFunction(function () {
            return func(arg0, arg1, arg2, arg3);
          });
        };
      } else {
        return func;
      }
    };

    var reusableSVGContainer;
    /**
     * Set the innerHTML property of a node
     *
     * @param {DOMElement} node
     * @param {string} html
     * @internal
     */

    var setInnerHTML = createMicrosoftUnsafeLocalFunction(function (node, html) {
      if (node.namespaceURI === SVG_NAMESPACE) {

        if (!('innerHTML' in node)) {
          // IE does not have innerHTML for SVG nodes, so instead we inject the
          // new markup in a temp node and then move the child nodes across into
          // the target node
          reusableSVGContainer = reusableSVGContainer || document.createElement('div');
          reusableSVGContainer.innerHTML = '<svg>' + html.valueOf().toString() + '</svg>';
          var svgNode = reusableSVGContainer.firstChild;

          while (node.firstChild) {
            node.removeChild(node.firstChild);
          }

          while (svgNode.firstChild) {
            node.appendChild(svgNode.firstChild);
          }

          return;
        }
      }

      node.innerHTML = html;
    });

    /**
     * HTML nodeType values that represent the type of the node
     */
    var ELEMENT_NODE = 1;
    var TEXT_NODE = 3;
    var COMMENT_NODE = 8;
    var DOCUMENT_NODE = 9;
    var DOCUMENT_FRAGMENT_NODE = 11;

    /**
     * Set the textContent property of a node. For text updates, it's faster
     * to set the `nodeValue` of the Text node directly instead of using
     * `.textContent` which will remove the existing node and create a new one.
     *
     * @param {DOMElement} node
     * @param {string} text
     * @internal
     */

    var setTextContent = function (node, text) {
      if (text) {
        var firstChild = node.firstChild;
        if (firstChild && firstChild === node.lastChild && firstChild.nodeType ===
  TEXT_NODE) {
          firstChild.nodeValue = text;
          return;
        }
      }

      node.textContent = text;
    };

    // List derived from Gecko source code:
    // https://github.com/mozilla/gecko-
  dev/blob/4e638efc71/layout/style/test/property_database.js
    var shorthandToLonghand = {
      animation: ['animationDelay', 'animationDirection', 'animationDuration',
  'animationFillMode', 'animationIterationCount', 'animationName', 'animationPlayState',
  'animationTimingFunction'],
      background: ['backgroundAttachment', 'backgroundClip', 'backgroundColor',
  'backgroundImage', 'backgroundOrigin', 'backgroundPositionX', 'backgroundPositionY',
  'backgroundRepeat', 'backgroundSize'],
      backgroundPosition: ['backgroundPositionX', 'backgroundPositionY'],
      border: ['borderBottomColor', 'borderBottomStyle', 'borderBottomWidth',
  'borderImageOutset', 'borderImageRepeat', 'borderImageSlice', 'borderImageSource',
  'borderImageWidth', 'borderLeftColor', 'borderLeftStyle', 'borderLeftWidth',
  'borderRightColor', 'borderRightStyle', 'borderRightWidth', 'borderTopColor',
  'borderTopStyle', 'borderTopWidth'],
      borderBlockEnd: ['borderBlockEndColor', 'borderBlockEndStyle',
  'borderBlockEndWidth'],
      borderBlockStart: ['borderBlockStartColor', 'borderBlockStartStyle',
  'borderBlockStartWidth'],
      borderBottom: ['borderBottomColor', 'borderBottomStyle', 'borderBottomWidth'],
      borderColor: ['borderBottomColor', 'borderLeftColor', 'borderRightColor',
  'borderTopColor'],
      borderImage: ['borderImageOutset', 'borderImageRepeat', 'borderImageSlice',
  'borderImageSource', 'borderImageWidth'],
      borderInlineEnd: ['borderInlineEndColor', 'borderInlineEndStyle',
  'borderInlineEndWidth'],
      borderInlineStart: ['borderInlineStartColor', 'borderInlineStartStyle',
  'borderInlineStartWidth'],
      borderLeft: ['borderLeftColor', 'borderLeftStyle', 'borderLeftWidth'],
      borderRadius: ['borderBottomLeftRadius', 'borderBottomRightRadius',
  'borderTopLeftRadius', 'borderTopRightRadius'],
      borderRight: ['borderRightColor', 'borderRightStyle', 'borderRightWidth'],
      borderStyle: ['borderBottomStyle', 'borderLeftStyle', 'borderRightStyle',
  'borderTopStyle'],
      borderTop: ['borderTopColor', 'borderTopStyle', 'borderTopWidth'],
      borderWidth: ['borderBottomWidth', 'borderLeftWidth', 'borderRightWidth',
  'borderTopWidth'],
      columnRule: ['columnRuleColor', 'columnRuleStyle', 'columnRuleWidth'],
      columns: ['columnCount', 'columnWidth'],
      flex: ['flexBasis', 'flexGrow', 'flexShrink'],
      flexFlow: ['flexDirection', 'flexWrap'],
      font: ['fontFamily', 'fontFeatureSettings', 'fontKerning', 'fontLanguageOverride',
  'fontSize', 'fontSizeAdjust', 'fontStretch', 'fontStyle', 'fontVariant',
  'fontVariantAlternates', 'fontVariantCaps', 'fontVariantEastAsian',
  'fontVariantLigatures', 'fontVariantNumeric', 'fontVariantPosition', 'fontWeight',
  'lineHeight'],
      fontVariant: ['fontVariantAlternates', 'fontVariantCaps', 'fontVariantEastAsian',
  'fontVariantLigatures', 'fontVariantNumeric', 'fontVariantPosition'],
      gap: ['columnGap', 'rowGap'],
      grid: ['gridAutoColumns', 'gridAutoFlow', 'gridAutoRows', 'gridTemplateAreas',
  'gridTemplateColumns', 'gridTemplateRows'],
      gridArea: ['gridColumnEnd', 'gridColumnStart', 'gridRowEnd', 'gridRowStart'],
      gridColumn: ['gridColumnEnd', 'gridColumnStart'],
      gridColumnGap: ['columnGap'],
      gridGap: ['columnGap', 'rowGap'],
      gridRow: ['gridRowEnd', 'gridRowStart'],
      gridRowGap: ['rowGap'],
      gridTemplate: ['gridTemplateAreas', 'gridTemplateColumns', 'gridTemplateRows'],
      listStyle: ['listStyleImage', 'listStylePosition', 'listStyleType'],
      margin: ['marginBottom', 'marginLeft', 'marginRight', 'marginTop'],
      marker: ['markerEnd', 'markerMid', 'markerStart'],
      mask: ['maskClip', 'maskComposite', 'maskImage', 'maskMode', 'maskOrigin',
  'maskPositionX', 'maskPositionY', 'maskRepeat', 'maskSize'],
      maskPosition: ['maskPositionX', 'maskPositionY'],
      outline: ['outlineColor', 'outlineStyle', 'outlineWidth'],
      overflow: ['overflowX', 'overflowY'],
      padding: ['paddingBottom', 'paddingLeft', 'paddingRight', 'paddingTop'],
      placeContent: ['alignContent', 'justifyContent'],
      placeItems: ['alignItems', 'justifyItems'],
      placeSelf: ['alignSelf', 'justifySelf'],
      textDecoration: ['textDecorationColor', 'textDecorationLine', 'textDecorationStyle'],
      textEmphasis: ['textEmphasisColor', 'textEmphasisStyle'],
      transition: ['transitionDelay', 'transitionDuration', 'transitionProperty',
  'transitionTimingFunction'],
      wordWrap: ['overflowWrap']
    };

    /**
     * CSS properties which accept numbers but are not in units of "px".
     */
    var isUnitlessNumber = {
      animationIterationCount: true,
      aspectRatio: true,
      borderImageOutset: true,
      borderImageSlice: true,
      borderImageWidth: true,
      boxFlex: true,
      boxFlexGroup: true,
      boxOrdinalGroup: true,
      columnCount: true,
      columns: true,
      flex: true,
      flexGrow: true,
      flexPositive: true,
      flexShrink: true,
      flexNegative: true,
      flexOrder: true,
      gridArea: true,
      gridRow: true,
      gridRowEnd: true,
      gridRowSpan: true,
      gridRowStart: true,
      gridColumn: true,
      gridColumnEnd: true,
      gridColumnSpan: true,
      gridColumnStart: true,
      fontWeight: true,
      lineClamp: true,
      lineHeight: true,
      opacity: true,
      order: true,
      orphans: true,
      tabSize: true,
      widows: true,
      zIndex: true,
      zoom: true,
      // SVG-related properties
      fillOpacity: true,
      floodOpacity: true,
      stopOpacity: true,
      strokeDasharray: true,
      strokeDashoffset: true,
      strokeMiterlimit: true,
      strokeOpacity: true,
      strokeWidth: true
    };
    /**
     * @param {string} prefix vendor-specific prefix, eg: Webkit
     * @param {string} key style name, eg: transitionDuration
     * @return {string} style name prefixed with `prefix`, properly camelCased, eg:
     * WebkitTransitionDuration
     */

    function prefixKey(prefix, key) {
      return prefix + key.charAt(0).toUpperCase() + key.substring(1);
    }
    /**
     * Support style names that may come passed in prefixed by adding permutations
     * of vendor prefixes.
     */


    var prefixes = ['Webkit', 'ms', 'Moz', 'O']; // Using Object.keys here, or else the
  vanilla for-in loop makes IE8 go into an
    // infinite loop, because it iterates over the newly added props too.

    Object.keys(isUnitlessNumber).forEach(function (prop) {
      prefixes.forEach(function (prefix) {
        isUnitlessNumber[prefixKey(prefix, prop)] = isUnitlessNumber[prop];
      });
    });

    /**
     * Convert a value into the proper css writable value. The style name `name`
     * should be logical (no hyphens), as specified
     * in `CSSProperty.isUnitlessNumber`.
     *
     * @param {string} name CSS property name such as `topMargin`.
     * @param {*} value CSS property value such as `10px`.
     * @return {string} Normalized style value with dimensions applied.
     */

    function dangerousStyleValue(name, value, isCustomProperty) {
      // Note that we've removed escapeTextForBrowser() calls here since the
      // whole string will be escaped when the attribute is injected into
      // the markup. If you provide unsafe user data here they can inject
      // arbitrary CSS which may be problematic (I couldn't repro this):
      // https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
      // http://www.thespanner.co.uk/2007/11/26/ultimate-xss-css-injection/
      // This is not an XSS hole but instead a potential CSS injection issue
      // which has lead to a greater discussion about how we're going to
      // trust URLs moving forward. See #2115901
      var isEmpty = value == null || typeof value === 'boolean' || value === '';

      if (isEmpty) {
        return '';
      }

      if (!isCustomProperty && typeof value === 'number' && value !== 0 && !
  (isUnitlessNumber.hasOwnProperty(name) && isUnitlessNumber[name])) {
        return value + 'px'; // Presumes implicit 'px' suffix for unitless numbers
      }

      {
        checkCSSPropertyStringCoercion(value, name);
      }

      return ('' + value).trim();
    }

    var uppercasePattern = /([A-Z])/g;
    var msPattern = /^ms-/;
    /**
     * Hyphenates a camelcased CSS property name, for example:
     *
     *   > hyphenateStyleName('backgroundColor')
     *   < "background-color"
     *   > hyphenateStyleName('MozTransition')
     *   < "-moz-transition"
     *   > hyphenateStyleName('msTransition')
     *   < "-ms-transition"
     *
     * As Modernizr suggests (http://modernizr.com/docs/#prefixed), an `ms` prefix
     * is converted to `-ms-`.
     */

    function hyphenateStyleName(name) {
      return name.replace(uppercasePattern, '-$1').toLowerCase().replace(msPattern, '-ms-
  ');
    }

    var warnValidStyle = function () {};

    {
      // 'msTransform' is correct, but the other prefixes should be capitalized
      var badVendoredStyleNamePattern = /^(?:webkit|moz|o)[A-Z]/;
      var msPattern$1 = /^-ms-/;
      var hyphenPattern = /-(.)/g; // style values shouldn't contain a semicolon

      var badStyleValueWithSemicolonPattern = /;\s*$/;
      var warnedStyleNames = {};
      var warnedStyleValues = {};
      var warnedForNaNValue = false;
      var warnedForInfinityValue = false;

      var camelize = function (string) {
        return string.replace(hyphenPattern, function (_, character) {
          return character.toUpperCase();
        });
      };

      var warnHyphenatedStyleName = function (name) {
        if (warnedStyleNames.hasOwnProperty(name) && warnedStyleNames[name]) {
          return;
        }

        warnedStyleNames[name] = true;

        error('Unsupported style property %s. Did you mean %s?', name, // As Andi Smith
  suggests
        // (http://www.andismith.com/blog/2012/02/modernizr-prefixed/), an `-ms` prefix
        // is converted to lowercase `ms`.
        camelize(name.replace(msPattern$1, 'ms-')));
      };

      var warnBadVendoredStyleName = function (name) {
        if (warnedStyleNames.hasOwnProperty(name) && warnedStyleNames[name]) {
          return;
        }

        warnedStyleNames[name] = true;

        error('Unsupported vendor-prefixed style property %s. Did you mean %s?', name,
  name.charAt(0).toUpperCase() + name.slice(1));
      };

      var warnStyleValueWithSemicolon = function (name, value) {
        if (warnedStyleValues.hasOwnProperty(value) && warnedStyleValues[value]) {
          return;
        }

        warnedStyleValues[value] = true;

        error("Style property values shouldn't contain a semicolon. " + 'Try "%s: %s"
  instead.', name, value.replace(badStyleValueWithSemicolonPattern, ''));
      };

      var warnStyleValueIsNaN = function (name, value) {
        if (warnedForNaNValue) {
          return;
        }

        warnedForNaNValue = true;

        error('`NaN` is an invalid value for the `%s` css style property.', name);
      };

      var warnStyleValueIsInfinity = function (name, value) {
        if (warnedForInfinityValue) {
          return;
        }

        warnedForInfinityValue = true;

        error('`Infinity` is an invalid value for the `%s` css style property.', name);
      };

      warnValidStyle = function (name, value) {
        if (name.indexOf('-') > -1) {
          warnHyphenatedStyleName(name);
        } else if (badVendoredStyleNamePattern.test(name)) {
          warnBadVendoredStyleName(name);
        } else if (badStyleValueWithSemicolonPattern.test(value)) {
          warnStyleValueWithSemicolon(name, value);
        }

        if (typeof value === 'number') {
          if (isNaN(value)) {
            warnStyleValueIsNaN(name, value);
          } else if (!isFinite(value)) {
            warnStyleValueIsInfinity(name, value);
          }
        }
      };
    }

    var warnValidStyle$1 = warnValidStyle;

    /**
     * Operations for dealing with CSS properties.
     */

    /**
     * This creates a string that is expected to be equivalent to the style
     * attribute generated by server-side rendering. It by-passes warnings and
     * security checks so it's not safe to use this value for anything other than
     * comparison. It is only used in DEV for SSR validation.
     */

    function createDangerousStringForStyles(styles) {
      {
        var serialized = '';
        var delimiter = '';

        for (var styleName in styles) {
          if (!styles.hasOwnProperty(styleName)) {
            continue;
          }

          var styleValue = styles[styleName];

          if (styleValue != null) {
            var isCustomProperty = styleName.indexOf('--') === 0;
            serialized += delimiter + (isCustomProperty ? styleName :
  hyphenateStyleName(styleName)) + ':';
            serialized += dangerousStyleValue(styleName, styleValue, isCustomProperty);
            delimiter = ';';
          }
        }

        return serialized || null;
      }
    }
    /**
     * Sets the value for multiple styles on a node.  If a value is specified as
     * '' (empty string), the corresponding style property will be unset.
     *
     * @param {DOMElement} node
     * @param {object} styles
     */

    function setValueForStyles(node, styles) {
      var style = node.style;

      for (var styleName in styles) {
        if (!styles.hasOwnProperty(styleName)) {
          continue;
        }

        var isCustomProperty = styleName.indexOf('--') === 0;

        {
          if (!isCustomProperty) {
            warnValidStyle$1(styleName, styles[styleName]);
          }
        }

        var styleValue = dangerousStyleValue(styleName, styles[styleName],
  isCustomProperty);

        if (styleName === 'float') {
          styleName = 'cssFloat';
        }

        if (isCustomProperty) {
          style.setProperty(styleName, styleValue);
        } else {
          style[styleName] = styleValue;
        }
      }
    }

    function isValueEmpty(value) {
      return value == null || typeof value === 'boolean' || value === '';
    }
    /**
     * Given {color: 'red', overflow: 'hidden'} returns {
     *   color: 'color',
     *   overflowX: 'overflow',
     *   overflowY: 'overflow',
     * }. This can be read as "the overflowY property was set by the overflow
     * shorthand". That is, the values are the property that each was derived from.
     */


    function expandShorthandMap(styles) {
      var expanded = {};

      for (var key in styles) {
        var longhands = shorthandToLonghand[key] || [key];

        for (var i = 0; i < longhands.length; i++) {
          expanded[longhands[i]] = key;
        }
      }

      return expanded;
    }
    /**
     * When mixing shorthand and longhand property names, we warn during updates if
     * we expect an incorrect result to occur. In particular, we warn for:
     *
     * Updating a shorthand property (longhand gets overwritten):
     *   {font: 'foo', fontVariant: 'bar'} -> {font: 'baz', fontVariant: 'bar'}
     *   becomes .style.font = 'baz'
     * Removing a shorthand property (longhand gets lost too):
     *   {font: 'foo', fontVariant: 'bar'} -> {fontVariant: 'bar'}
     *   becomes .style.font = ''
     * Removing a longhand property (should revert to shorthand; doesn't):
     *   {font: 'foo', fontVariant: 'bar'} -> {font: 'foo'}
     *   becomes .style.fontVariant = ''
     */


    function validateShorthandPropertyCollisionInDev(styleUpdates, nextStyles) {
      {
        if (!nextStyles) {
          return;
        }

        var expandedUpdates = expandShorthandMap(styleUpdates);
        var expandedStyles = expandShorthandMap(nextStyles);
        var warnedAbout = {};

        for (var key in expandedUpdates) {
          var originalKey = expandedUpdates[key];
          var correctOriginalKey = expandedStyles[key];

          if (correctOriginalKey && originalKey !== correctOriginalKey) {
            var warningKey = originalKey + ',' + correctOriginalKey;

            if (warnedAbout[warningKey]) {
              continue;
            }

            warnedAbout[warningKey] = true;

            error('%s a style property during rerender (%s) when a ' + 'conflicting
  property is set (%s) can lead to styling bugs. To ' + "avoid this, don't mix shorthand
  and non-shorthand properties " + 'for the same value; instead, replace the shorthand with
  ' + 'separate values.', isValueEmpty(styleUpdates[originalKey]) ? 'Removing' :
  'Updating', originalKey, correctOriginalKey);
          }
        }
      }
    }

    // For HTML, certain tags should omit their close tag. We keep a list for
    // those special-case tags.
    var omittedCloseTags = {
      area: true,
      base: true,
      br: true,
      col: true,
      embed: true,
      hr: true,
      img: true,
      input: true,
      keygen: true,
      link: true,
      meta: true,
      param: true,
      source: true,
      track: true,
      wbr: true // NOTE: menuitem's close tag should be omitted, but that causes problems.

    };

    // `omittedCloseTags` except that `menuitem` should still have its closing tag.

    var voidElementTags = assign({
      menuitem: true
    }, omittedCloseTags);

    var HTML = '__html';

    function assertValidProps(tag, props) {
      if (!props) {
        return;
      } // Note the use of `==` which checks for null or undefined.


      if (voidElementTags[tag]) {
        if (props.children != null || props.dangerouslySetInnerHTML != null) {
          throw new Error(tag + " is a void element tag and must neither have `children`
  nor " + 'use `dangerouslySetInnerHTML`.');
        }
      }

      if (props.dangerouslySetInnerHTML != null) {
        if (props.children != null) {
          throw new Error('Can only set one of `children` or
  `props.dangerouslySetInnerHTML`.');
        }

        if (typeof props.dangerouslySetInnerHTML !== 'object' || !(HTML in
  props.dangerouslySetInnerHTML)) {
          throw new Error('`props.dangerouslySetInnerHTML` must be in the form `{__html:
  ...}`. ' + 'Please visit https://reactjs.org/link/dangerously-set-inner-html ' + 'for
  more information.');
        }
      }

      {
        if (!props.suppressContentEditableWarning && props.contentEditable &&
  props.children != null) {
          error('A component is `contentEditable` and contains `children` managed by ' +
  'React. It is now your responsibility to guarantee that none of ' + 'those nodes are
  unexpectedly modified or duplicated. This is ' + 'probably not intentional.');
        }
      }

      if (props.style != null && typeof props.style !== 'object') {
        throw new Error('The `style` prop expects a mapping from style properties to
  values, ' + "not a string. For example, style={{marginRight: spacing + 'em'}} when " +
  'using JSX.');
      }
    }

    function isCustomComponent(tagName, props) {
      if (tagName.indexOf('-') === -1) {
        return typeof props.is === 'string';
      }

      switch (tagName) {
        // These are reserved SVG and MathML elements.
        // We don't mind this list too much because we expect it to never grow.
        // The alternative is to track the namespace in a few places which is convoluted.
        // https://w3c.github.io/webcomponents/spec/custom/#custom-elements-core-concepts
        case 'annotation-xml':
        case 'color-profile':
        case 'font-face':
        case 'font-face-src':
        case 'font-face-uri':
        case 'font-face-format':
        case 'font-face-name':
        case 'missing-glyph':
          return false;

        default:
          return true;
      }
    }

    // When adding attributes to the HTML or SVG allowed attribute list, be sure to
    // also add them to this module to ensure casing and incorrect name
    // warnings.
    var possibleStandardNames = {
      // HTML
      accept: 'accept',
      acceptcharset: 'acceptCharset',
      'accept-charset': 'acceptCharset',
      accesskey: 'accessKey',
      action: 'action',
      allowfullscreen: 'allowFullScreen',
      alt: 'alt',
      as: 'as',
      async: 'async',
      autocapitalize: 'autoCapitalize',
      autocomplete: 'autoComplete',
      autocorrect: 'autoCorrect',
      autofocus: 'autoFocus',
      autoplay: 'autoPlay',
      autosave: 'autoSave',
      capture: 'capture',
      cellpadding: 'cellPadding',
      cellspacing: 'cellSpacing',
      challenge: 'challenge',
      charset: 'charSet',
      checked: 'checked',
      children: 'children',
      cite: 'cite',
      class: 'className',
      classid: 'classID',
      classname: 'className',
      cols: 'cols',
      colspan: 'colSpan',
      content: 'content',
      contenteditable: 'contentEditable',
      contextmenu: 'contextMenu',
      controls: 'controls',
      controlslist: 'controlsList',
      coords: 'coords',
      crossorigin: 'crossOrigin',
      dangerouslysetinnerhtml: 'dangerouslySetInnerHTML',
      data: 'data',
      datetime: 'dateTime',
      default: 'default',
      defaultchecked: 'defaultChecked',
      defaultvalue: 'defaultValue',
      defer: 'defer',
      dir: 'dir',
      disabled: 'disabled',
      disablepictureinpicture: 'disablePictureInPicture',
      disableremoteplayback: 'disableRemotePlayback',
      download: 'download',
      draggable: 'draggable',
      enctype: 'encType',
      enterkeyhint: 'enterKeyHint',
      for: 'htmlFor',
      form: 'form',
      formmethod: 'formMethod',
      formaction: 'formAction',
      formenctype: 'formEncType',
      formnovalidate: 'formNoValidate',
      formtarget: 'formTarget',
      frameborder: 'frameBorder',
      headers: 'headers',
      height: 'height',
      hidden: 'hidden',
      high: 'high',
      href: 'href',
      hreflang: 'hrefLang',
      htmlfor: 'htmlFor',
      httpequiv: 'httpEquiv',
      'http-equiv': 'httpEquiv',
      icon: 'icon',
      id: 'id',
      imagesizes: 'imageSizes',
      imagesrcset: 'imageSrcSet',
      innerhtml: 'innerHTML',
      inputmode: 'inputMode',
      integrity: 'integrity',
      is: 'is',
      itemid: 'itemID',
      itemprop: 'itemProp',
      itemref: 'itemRef',
      itemscope: 'itemScope',
      itemtype: 'itemType',
      keyparams: 'keyParams',
      keytype: 'keyType',
      kind: 'kind',
      label: 'label',
      lang: 'lang',
      list: 'list',
      loop: 'loop',
      low: 'low',
      manifest: 'manifest',
      marginwidth: 'marginWidth',
      marginheight: 'marginHeight',
      max: 'max',
      maxlength: 'maxLength',
      media: 'media',
      mediagroup: 'mediaGroup',
      method: 'method',
      min: 'min',
      minlength: 'minLength',
      multiple: 'multiple',
      muted: 'muted',
      name: 'name',
      nomodule: 'noModule',
      nonce: 'nonce',
      novalidate: 'noValidate',
      open: 'open',
      optimum: 'optimum',
      pattern: 'pattern',
      placeholder: 'placeholder',
      playsinline: 'playsInline',
      poster: 'poster',
      preload: 'preload',
      profile: 'profile',
      radiogroup: 'radioGroup',
      readonly: 'readOnly',
      referrerpolicy: 'referrerPolicy',
      rel: 'rel',
      required: 'required',
      reversed: 'reversed',
      role: 'role',
      rows: 'rows',
      rowspan: 'rowSpan',
      sandbox: 'sandbox',
      scope: 'scope',
      scoped: 'scoped',
      scrolling: 'scrolling',
      seamless: 'seamless',
      selected: 'selected',
      shape: 'shape',
      size: 'size',
      sizes: 'sizes',
      span: 'span',
      spellcheck: 'spellCheck',
      src: 'src',
      srcdoc: 'srcDoc',
      srclang: 'srcLang',
      srcset: 'srcSet',
      start: 'start',
      step: 'step',
      style: 'style',
      summary: 'summary',
      tabindex: 'tabIndex',
      target: 'target',
      title: 'title',
      type: 'type',
      usemap: 'useMap',
      value: 'value',
      width: 'width',
      wmode: 'wmode',
      wrap: 'wrap',
      // SVG
      about: 'about',
      accentheight: 'accentHeight',
      'accent-height': 'accentHeight',
      accumulate: 'accumulate',
      additive: 'additive',
      alignmentbaseline: 'alignmentBaseline',
      'alignment-baseline': 'alignmentBaseline',
      allowreorder: 'allowReorder',
      alphabetic: 'alphabetic',
      amplitude: 'amplitude',
      arabicform: 'arabicForm',
      'arabic-form': 'arabicForm',
      ascent: 'ascent',
      attributename: 'attributeName',
      attributetype: 'attributeType',
      autoreverse: 'autoReverse',
      azimuth: 'azimuth',
      basefrequency: 'baseFrequency',
      baselineshift: 'baselineShift',
      'baseline-shift': 'baselineShift',
      baseprofile: 'baseProfile',
      bbox: 'bbox',
      begin: 'begin',
      bias: 'bias',
      by: 'by',
      calcmode: 'calcMode',
      capheight: 'capHeight',
      'cap-height': 'capHeight',
      clip: 'clip',
      clippath: 'clipPath',
      'clip-path': 'clipPath',
      clippathunits: 'clipPathUnits',
      cliprule: 'clipRule',
      'clip-rule': 'clipRule',
      color: 'color',
      colorinterpolation: 'colorInterpolation',
      'color-interpolation': 'colorInterpolation',
      colorinterpolationfilters: 'colorInterpolationFilters',
      'color-interpolation-filters': 'colorInterpolationFilters',
      colorprofile: 'colorProfile',
      'color-profile': 'colorProfile',
      colorrendering: 'colorRendering',
      'color-rendering': 'colorRendering',
      contentscripttype: 'contentScriptType',
      contentstyletype: 'contentStyleType',
      cursor: 'cursor',
      cx: 'cx',
      cy: 'cy',
      d: 'd',
      datatype: 'datatype',
      decelerate: 'decelerate',
      descent: 'descent',
      diffuseconstant: 'diffuseConstant',
      direction: 'direction',
      display: 'display',
      divisor: 'divisor',
      dominantbaseline: 'dominantBaseline',
      'dominant-baseline': 'dominantBaseline',
      dur: 'dur',
      dx: 'dx',
      dy: 'dy',
      edgemode: 'edgeMode',
      elevation: 'elevation',
      enablebackground: 'enableBackground',
      'enable-background': 'enableBackground',
      end: 'end',
      exponent: 'exponent',
      externalresourcesrequired: 'externalResourcesRequired',
      fill: 'fill',
      fillopacity: 'fillOpacity',
      'fill-opacity': 'fillOpacity',
      fillrule: 'fillRule',
      'fill-rule': 'fillRule',
      filter: 'filter',
      filterres: 'filterRes',
      filterunits: 'filterUnits',
      floodopacity: 'floodOpacity',
      'flood-opacity': 'floodOpacity',
      floodcolor: 'floodColor',
      'flood-color': 'floodColor',
      focusable: 'focusable',
      fontfamily: 'fontFamily',
      'font-family': 'fontFamily',
      fontsize: 'fontSize',
      'font-size': 'fontSize',
      fontsizeadjust: 'fontSizeAdjust',
      'font-size-adjust': 'fontSizeAdjust',
      fontstretch: 'fontStretch',
      'font-stretch': 'fontStretch',
      fontstyle: 'fontStyle',
      'font-style': 'fontStyle',
      fontvariant: 'fontVariant',
      'font-variant': 'fontVariant',
      fontweight: 'fontWeight',
      'font-weight': 'fontWeight',
      format: 'format',
      from: 'from',
      fx: 'fx',
      fy: 'fy',
      g1: 'g1',
      g2: 'g2',
      glyphname: 'glyphName',
      'glyph-name': 'glyphName',
      glyphorientationhorizontal: 'glyphOrientationHorizontal',
      'glyph-orientation-horizontal': 'glyphOrientationHorizontal',
      glyphorientationvertical: 'glyphOrientationVertical',
      'glyph-orientation-vertical': 'glyphOrientationVertical',
      glyphref: 'glyphRef',
      gradienttransform: 'gradientTransform',
      gradientunits: 'gradientUnits',
      hanging: 'hanging',
      horizadvx: 'horizAdvX',
      'horiz-adv-x': 'horizAdvX',
      horizoriginx: 'horizOriginX',
      'horiz-origin-x': 'horizOriginX',
      ideographic: 'ideographic',
      imagerendering: 'imageRendering',
      'image-rendering': 'imageRendering',
      in2: 'in2',
      in: 'in',
      inlist: 'inlist',
      intercept: 'intercept',
      k1: 'k1',
      k2: 'k2',
      k3: 'k3',
      k4: 'k4',
      k: 'k',
      kernelmatrix: 'kernelMatrix',
      kernelunitlength: 'kernelUnitLength',
      kerning: 'kerning',
      keypoints: 'keyPoints',
      keysplines: 'keySplines',
      keytimes: 'keyTimes',
      lengthadjust: 'lengthAdjust',
      letterspacing: 'letterSpacing',
      'letter-spacing': 'letterSpacing',
      lightingcolor: 'lightingColor',
      'lighting-color': 'lightingColor',
      limitingconeangle: 'limitingConeAngle',
      local: 'local',
      markerend: 'markerEnd',
      'marker-end': 'markerEnd',
      markerheight: 'markerHeight',
      markermid: 'markerMid',
      'marker-mid': 'markerMid',
      markerstart: 'markerStart',
      'marker-start': 'markerStart',
      markerunits: 'markerUnits',
      markerwidth: 'markerWidth',
      mask: 'mask',
      maskcontentunits: 'maskContentUnits',
      maskunits: 'maskUnits',
      mathematical: 'mathematical',
      mode: 'mode',
      numoctaves: 'numOctaves',
      offset: 'offset',
      opacity: 'opacity',
      operator: 'operator',
      order: 'order',
      orient: 'orient',
      orientation: 'orientation',
      origin: 'origin',
      overflow: 'overflow',
      overlineposition: 'overlinePosition',
      'overline-position': 'overlinePosition',
      overlinethickness: 'overlineThickness',
      'overline-thickness': 'overlineThickness',
      paintorder: 'paintOrder',
      'paint-order': 'paintOrder',
      panose1: 'panose1',
      'panose-1': 'panose1',
      pathlength: 'pathLength',
      patterncontentunits: 'patternContentUnits',
      patterntransform: 'patternTransform',
      patternunits: 'patternUnits',
      pointerevents: 'pointerEvents',
      'pointer-events': 'pointerEvents',
      points: 'points',
      pointsatx: 'pointsAtX',
      pointsaty: 'pointsAtY',
      pointsatz: 'pointsAtZ',
      prefix: 'prefix',
      preservealpha: 'preserveAlpha',
      preserveaspectratio: 'preserveAspectRatio',
      primitiveunits: 'primitiveUnits',
      property: 'property',
      r: 'r',
      radius: 'radius',
      refx: 'refX',
      refy: 'refY',
      renderingintent: 'renderingIntent',
      'rendering-intent': 'renderingIntent',
      repeatcount: 'repeatCount',
      repeatdur: 'repeatDur',
      requiredextensions: 'requiredExtensions',
      requiredfeatures: 'requiredFeatures',
      resource: 'resource',
      restart: 'restart',
      result: 'result',
      results: 'results',
      rotate: 'rotate',
      rx: 'rx',
      ry: 'ry',
      scale: 'scale',
      security: 'security',
      seed: 'seed',
      shaperendering: 'shapeRendering',
      'shape-rendering': 'shapeRendering',
      slope: 'slope',
      spacing: 'spacing',
      specularconstant: 'specularConstant',
      specularexponent: 'specularExponent',
      speed: 'speed',
      spreadmethod: 'spreadMethod',
      startoffset: 'startOffset',
      stddeviation: 'stdDeviation',
      stemh: 'stemh',
      stemv: 'stemv',
      stitchtiles: 'stitchTiles',
      stopcolor: 'stopColor',
      'stop-color': 'stopColor',
      stopopacity: 'stopOpacity',
      'stop-opacity': 'stopOpacity',
      strikethroughposition: 'strikethroughPosition',
      'strikethrough-position': 'strikethroughPosition',
      strikethroughthickness: 'strikethroughThickness',
      'strikethrough-thickness': 'strikethroughThickness',
      string: 'string',
      stroke: 'stroke',
      strokedasharray: 'strokeDasharray',
      'stroke-dasharray': 'strokeDasharray',
      strokedashoffset: 'strokeDashoffset',
      'stroke-dashoffset': 'strokeDashoffset',
      strokelinecap: 'strokeLinecap',
      'stroke-linecap': 'strokeLinecap',
      strokelinejoin: 'strokeLinejoin',
      'stroke-linejoin': 'strokeLinejoin',
      strokemiterlimit: 'strokeMiterlimit',
      'stroke-miterlimit': 'strokeMiterlimit',
      strokewidth: 'strokeWidth',
      'stroke-width': 'strokeWidth',
      strokeopacity: 'strokeOpacity',
      'stroke-opacity': 'strokeOpacity',
      suppresscontenteditablewarning: 'suppressContentEditableWarning',
      suppresshydrationwarning: 'suppressHydrationWarning',
      surfacescale: 'surfaceScale',
      systemlanguage: 'systemLanguage',
      tablevalues: 'tableValues',
      targetx: 'targetX',
      targety: 'targetY',
      textanchor: 'textAnchor',
      'text-anchor': 'textAnchor',
      textdecoration: 'textDecoration',
      'text-decoration': 'textDecoration',
      textlength: 'textLength',
      textrendering: 'textRendering',
      'text-rendering': 'textRendering',
      to: 'to',
      transform: 'transform',
      typeof: 'typeof',
      u1: 'u1',
      u2: 'u2',
      underlineposition: 'underlinePosition',
      'underline-position': 'underlinePosition',
      underlinethickness: 'underlineThickness',
      'underline-thickness': 'underlineThickness',
      unicode: 'unicode',
      unicodebidi: 'unicodeBidi',
      'unicode-bidi': 'unicodeBidi',
      unicoderange: 'unicodeRange',
      'unicode-range': 'unicodeRange',
      unitsperem: 'unitsPerEm',
      'units-per-em': 'unitsPerEm',
      unselectable: 'unselectable',
      valphabetic: 'vAlphabetic',
      'v-alphabetic': 'vAlphabetic',
      values: 'values',
      vectoreffect: 'vectorEffect',
      'vector-effect': 'vectorEffect',
      version: 'version',
      vertadvy: 'vertAdvY',
      'vert-adv-y': 'vertAdvY',
      vertoriginx: 'vertOriginX',
      'vert-origin-x': 'vertOriginX',
      vertoriginy: 'vertOriginY',
      'vert-origin-y': 'vertOriginY',
      vhanging: 'vHanging',
      'v-hanging': 'vHanging',
      videographic: 'vIdeographic',
      'v-ideographic': 'vIdeographic',
      viewbox: 'viewBox',
      viewtarget: 'viewTarget',
      visibility: 'visibility',
      vmathematical: 'vMathematical',
      'v-mathematical': 'vMathematical',
      vocab: 'vocab',
      widths: 'widths',
      wordspacing: 'wordSpacing',
      'word-spacing': 'wordSpacing',
      writingmode: 'writingMode',
      'writing-mode': 'writingMode',
      x1: 'x1',
      x2: 'x2',
      x: 'x',
      xchannelselector: 'xChannelSelector',
      xheight: 'xHeight',
      'x-height': 'xHeight',
      xlinkactuate: 'xlinkActuate',
      'xlink:actuate': 'xlinkActuate',
      xlinkarcrole: 'xlinkArcrole',
      'xlink:arcrole': 'xlinkArcrole',
      xlinkhref: 'xlinkHref',
      'xlink:href': 'xlinkHref',
      xlinkrole: 'xlinkRole',
      'xlink:role': 'xlinkRole',
      xlinkshow: 'xlinkShow',
      'xlink:show': 'xlinkShow',
      xlinktitle: 'xlinkTitle',
      'xlink:title': 'xlinkTitle',
      xlinktype: 'xlinkType',
      'xlink:type': 'xlinkType',
      xmlbase: 'xmlBase',
      'xml:base': 'xmlBase',
      xmllang: 'xmlLang',
      'xml:lang': 'xmlLang',
      xmlns: 'xmlns',
      'xml:space': 'xmlSpace',
      xmlnsxlink: 'xmlnsXlink',
      'xmlns:xlink': 'xmlnsXlink',
      xmlspace: 'xmlSpace',
      y1: 'y1',
      y2: 'y2',
      y: 'y',
      ychannelselector: 'yChannelSelector',
      z: 'z',
      zoomandpan: 'zoomAndPan'
    };

    var ariaProperties = {
      'aria-current': 0,
      // state
      'aria-description': 0,
      'aria-details': 0,
      'aria-disabled': 0,
      // state
      'aria-hidden': 0,
      // state
      'aria-invalid': 0,
      // state
      'aria-keyshortcuts': 0,
      'aria-label': 0,
      'aria-roledescription': 0,
      // Widget Attributes
      'aria-autocomplete': 0,
      'aria-checked': 0,
      'aria-expanded': 0,
      'aria-haspopup': 0,
      'aria-level': 0,
      'aria-modal': 0,
      'aria-multiline': 0,
      'aria-multiselectable': 0,
      'aria-orientation': 0,
      'aria-placeholder': 0,
      'aria-pressed': 0,
      'aria-readonly': 0,
      'aria-required': 0,
      'aria-selected': 0,
      'aria-sort': 0,
      'aria-valuemax': 0,
      'aria-valuemin': 0,
      'aria-valuenow': 0,
      'aria-valuetext': 0,
      // Live Region Attributes
      'aria-atomic': 0,
      'aria-busy': 0,
      'aria-live': 0,
      'aria-relevant': 0,
      // Drag-and-Drop Attributes
      'aria-dropeffect': 0,
      'aria-grabbed': 0,
      // Relationship Attributes
      'aria-activedescendant': 0,
      'aria-colcount': 0,
      'aria-colindex': 0,
      'aria-colspan': 0,
      'aria-controls': 0,
      'aria-describedby': 0,
      'aria-errormessage': 0,
      'aria-flowto': 0,
      'aria-labelledby': 0,
      'aria-owns': 0,
      'aria-posinset': 0,
      'aria-rowcount': 0,
      'aria-rowindex': 0,
      'aria-rowspan': 0,
      'aria-setsize': 0
    };

    var warnedProperties = {};
    var rARIA = new RegExp('^(aria)-[' + ATTRIBUTE_NAME_CHAR + ']*$');
    var rARIACamel = new RegExp('^(aria)[A-Z][' + ATTRIBUTE_NAME_CHAR + ']*$');

    function validateProperty(tagName, name) {
      {
        if (hasOwnProperty.call(warnedProperties, name) && warnedProperties[name]) {
          return true;
        }

        if (rARIACamel.test(name)) {
          var ariaName = 'aria-' + name.slice(4).toLowerCase();
          var correctName = ariaProperties.hasOwnProperty(ariaName) ? ariaName : null; //
  If this is an aria-* attribute, but is not listed in the known DOM
          // DOM properties, then it is an invalid aria-* attribute.

          if (correctName == null) {
            error('Invalid ARIA attribute `%s`. ARIA attributes follow the pattern aria-*
  and must be lowercase.', name);

            warnedProperties[name] = true;
            return true;
          } // aria-* attributes should be lowercase; suggest the lowercase version.


          if (name !== correctName) {
            error('Invalid ARIA attribute `%s`. Did you mean `%s`?', name, correctName);

            warnedProperties[name] = true;
            return true;
          }
        }

        if (rARIA.test(name)) {
          var lowerCasedName = name.toLowerCase();
          var standardName = ariaProperties.hasOwnProperty(lowerCasedName) ? lowerCasedName
  : null; // If this is an aria-* attribute, but is not listed in the known DOM
          // DOM properties, then it is an invalid aria-* attribute.

          if (standardName == null) {
            warnedProperties[name] = true;
            return false;
          } // aria-* attributes should be lowercase; suggest the lowercase version.


          if (name !== standardName) {
            error('Unknown ARIA attribute `%s`. Did you mean `%s`?', name, standardName);

            warnedProperties[name] = true;
            return true;
          }
        }
      }
```

```
      return true;
    }

    function warnInvalidARIAProps(type, props) {
      {
        var invalidProps = [];

        for (var key in props) {
          var isValid = validateProperty(type, key);

          if (!isValid) {
            invalidProps.push(key);
          }
        }

        var unknownPropString = invalidProps.map(function (prop) {
          return '`' + prop + '`';
        }).join(', ');

        if (invalidProps.length === 1) {
          error('Invalid aria prop %s on <%s> tag. ' + 'For details, see 
  https://reactjs.org/link/invalid-aria-props', unknownPropString, type);
        } else if (invalidProps.length > 1) {
          error('Invalid aria props %s on <%s> tag. ' + 'For details, see 
  https://reactjs.org/link/invalid-aria-props', unknownPropString, type);
        }
      }
    }

    function validateProperties(type, props) {
      if (isCustomComponent(type, props)) {
        return;
      }

      warnInvalidARIAProps(type, props);
    }

    var didWarnValueNull = false;
    function validateProperties$1(type, props) {
      {
        if (type !== 'input' && type !== 'textarea' && type !== 'select') {
          return;
        }

        if (props != null && props.value === null && !didWarnValueNull) {
          didWarnValueNull = true;

          if (type === 'select' && props.multiple) {
            error('`value` prop on `%s` should not be null. ' + 'Consider using an empty 
  array when `multiple` is set to `true` ' + 'to clear the component or `undefined` for 
  uncontrolled components.', type);
          } else {
            error('`value` prop on `%s` should not be null. ' + 'Consider using an empty 
  string to clear the component or `undefined` ' + 'for uncontrolled components.', type);
          }
        }
      }
    }

    var validateProperty$1 = function () {};

    {
      var warnedProperties$1 = {};
      var EVENT_NAME_REGEX = /^on./;
      var INVALID_EVENT_NAME_REGEX = /^on[^A-Z]/;
      var rARIA$1 = new RegExp('^(aria)-[' + ATTRIBUTE_NAME_CHAR + ']*$');
      var rARIACamel$1 = new RegExp('^(aria)[A-Z][' + ATTRIBUTE_NAME_CHAR + ']*$');

      validateProperty$1 = function (tagName, name, value, eventRegistry) {
```

```
      if (hasOwnProperty.call(warnedProperties$1, name) && warnedProperties$1[name]) {
        return true;
      }

      var lowerCasedName = name.toLowerCase();

      if (lowerCasedName === 'onfocusin' || lowerCasedName === 'onfocusout') {
        error('React uses onFocus and onBlur instead of onFocusIn and onFocusOut. ' +
'All React events are normalized to bubble, so onFocusIn and onFocusOut ' + 'are not
needed/supported by React.');

        warnedProperties$1[name] = true;
        return true;
      } // We can't rely on the event system being injected on the server.


      if (eventRegistry != null) {
        var registrationNameDependencies = eventRegistry.registrationNameDependencies,
            possibleRegistrationNames = eventRegistry.possibleRegistrationNames;

        if (registrationNameDependencies.hasOwnProperty(name)) {
          return true;
        }

        var registrationName = possibleRegistrationNames.hasOwnProperty(lowerCasedName) ?
possibleRegistrationNames[lowerCasedName] : null;

        if (registrationName != null) {
          error('Invalid event handler property `%s`. Did you mean `%s`?', name,
registrationName);

          warnedProperties$1[name] = true;
          return true;
        }

        if (EVENT_NAME_REGEX.test(name)) {
          error('Unknown event handler property `%s`. It will be ignored.', name);

          warnedProperties$1[name] = true;
          return true;
        }
      } else if (EVENT_NAME_REGEX.test(name)) {
        // If no event plugins have been injected, we are in a server environment.
        // So we can't tell if the event name is correct for sure, but we can filter
        // out known bad ones like `onclick`. We can't suggest a specific replacement
though.
        if (INVALID_EVENT_NAME_REGEX.test(name)) {
          error('Invalid event handler property `%s`. ' + 'React events use the camelCase
naming convention, for example `onClick`.', name);
        }

        warnedProperties$1[name] = true;
        return true;
      } // Let the ARIA attribute hook validate ARIA attributes


      if (rARIA$1.test(name) || rARIACamel$1.test(name)) {
        return true;
      }

      if (lowerCasedName === 'innerhtml') {
        error('Directly setting property `innerHTML` is not permitted. ' + 'For more
information, lookup documentation on `dangerouslySetInnerHTML`.');

        warnedProperties$1[name] = true;
        return true;
      }

      if (lowerCasedName === 'aria') {
        error('The `aria` attribute is reserved for future use in React. ' + 'Pass
individual `aria-` attributes instead.');

        warnedProperties$1[name] = true;
        return true;
      }

      if (lowerCasedName === 'is' && value !== null && value !== undefined && typeof
value !== 'string') {
        error('Received a `%s` for a string attribute `is`. If this is expected, cast ' +
'the value to a string.', typeof value);

        warnedProperties$1[name] = true;
        return true;
      }

      if (typeof value === 'number' && isNaN(value)) {
        error('Received NaN for the `%s` attribute. If this is expected, cast ' + 'the
value to a string.', name);

        warnedProperties$1[name] = true;
        return true;
      }

      var propertyInfo = getPropertyInfo(name);
      var isReserved = propertyInfo !== null && propertyInfo.type === RESERVED; // Known
attributes should match the casing specified in the property config.

      if (possibleStandardNames.hasOwnProperty(lowerCasedName)) {
        var standardName = possibleStandardNames[lowerCasedName];

        if (standardName !== name) {
          error('Invalid DOM property `%s`. Did you mean `%s`?', name, standardName);

          warnedProperties$1[name] = true;
          return true;
        }
      } else if (!isReserved && name !== lowerCasedName) {
        // Unknown attributes should have lowercase casing since that's how they
        // will be cased anyway with server rendering.
        error('React does not recognize the `%s` prop on a DOM element. If you ' +
'intentionally want it to appear in the DOM as a custom ' + 'attribute, spell it as
lowercase `%s` instead. ' + 'If you accidentally passed it from a parent component,
remove ' + 'it from the DOM element.', name, lowerCasedName);

        warnedProperties$1[name] = true;
        return true;
      }

      if (typeof value === 'boolean' && shouldRemoveAttributeWithWarning(name, value,
propertyInfo, false)) {
        if (value) {
          error('Received `%s` for a non-boolean attribute `%s`.\n\n' + 'If you want to
write it to the DOM, pass a string instead: ' + '%s="%s" or %s={value.toString()}.',
value, name, name, value, name);
        } else {
          error('Received `%s` for a non-boolean attribute `%s`.\n\n' + 'If you want to
write it to the DOM, pass a string instead: ' + '%s="%s" or %s={value.toString()}.\n\n' +
'If you used to conditionally omit it with %s={condition && value}, ' + 'pass %s=
{condition ? value : undefined} instead.', value, name, name, value, name, name, name);
        }

        warnedProperties$1[name] = true;
        return true;
      } // Now that we've validated casing, do not validate
      // data types for reserved props


      if (isReserved) {
        return true;
      } // Warn when a known attribute is a bad type


      if (shouldRemoveAttributeWithWarning(name, value, propertyInfo, false)) {
        warnedProperties$1[name] = true;
        return false;
      } // Warn when passing the strings 'false' or 'true' into a boolean prop


      if ((value === 'false' || value === 'true') && propertyInfo !== null &&
propertyInfo.type === BOOLEAN) {
        error('Received the string `%s` for the boolean attribute `%s`. ' + '%s ' + 'Did
you mean %s={%s}?', value, name, value === 'false' ? 'The browser will interpret it as a
truthy value.' : 'Although this works, it will not work as expected if you pass the
string "false".', name, value);

        warnedProperties$1[name] = true;
        return true;
      }

      return true;
    };
  }

  var warnUnknownProperties = function (type, props, eventRegistry) {
    {
      var unknownProps = [];

      for (var key in props) {
        var isValid = validateProperty$1(type, key, props[key], eventRegistry);

        if (!isValid) {
          unknownProps.push(key);
        }
      }

      var unknownPropString = unknownProps.map(function (prop) {
        return '`' + prop + '`';
      }).join(', ');

      if (unknownProps.length === 1) {
        error('Invalid value for prop %s on <%s> tag. Either remove it from the element,
' + 'or pass a string or number value to keep it in the DOM. ' + 'For details, see
https://reactjs.org/link/attribute-behavior ', unknownPropString, type);
      } else if (unknownProps.length > 1) {
        error('Invalid values for props %s on <%s> tag. Either remove them from the
element, ' + 'or pass a string or number value to keep them in the DOM. ' + 'For details,
see https://reactjs.org/link/attribute-behavior ', unknownPropString, type);
      }
    }
  };

  function validateProperties$2(type, props, eventRegistry) {
    if (isCustomComponent(type, props)) {
      return;
    }

    warnUnknownProperties(type, props, eventRegistry);
  }

  var IS_EVENT_HANDLE_NON_MANAGED_NODE = 1;
  var IS_NON_DELEGATED = 1 << 1;
  var IS_CAPTURE_PHASE = 1 << 2;
  // set to LEGACY_FB_SUPPORT. LEGACY_FB_SUPPORT only gets set when
  // we call willDeferLaterForLegacyFBSupport, thus not bailing out
  // will result in endless cycles like an infinite loop.
  // We also don't want to defer during event replaying.
  var SHOULD_NOT_PROCESS_POLYFILL_EVENT_PLUGINS = IS_EVENT_HANDLE_NON_MANAGED_NODE |
IS_NON_DELEGATED | IS_CAPTURE_PHASE;

  // This exists to avoid circular dependency between ReactDOMEventReplaying
  // and DOMPluginEventSystem.
  var currentReplayingEvent = null;
  function setReplayingEvent(event) {
    {
      if (currentReplayingEvent !== null) {
        error('Expected currently replaying event to be null. This error ' + 'is likely
caused by a bug in React. Please file an issue.');
      }
    }

    currentReplayingEvent = event;
  }
  function resetReplayingEvent() {
    {
      if (currentReplayingEvent === null) {
        error('Expected currently replaying event to not be null. This error ' + 'is
likely caused by a bug in React. Please file an issue.');
      }
    }

    currentReplayingEvent = null;
  }
  function isReplayingEvent(event) {
    return event === currentReplayingEvent;
  }

  /**
   * Gets the target node from a native browser event by accounting for
   * inconsistencies in browser DOM APIs.
   *
   * @param {object} nativeEvent Native browser event.
   * @return {DOMEventTarget} Target node.
   */

  function getEventTarget(nativeEvent) {
    // Fallback to nativeEvent.srcElement for IE9
    // https://github.com/facebook/react/issues/12506
    var target = nativeEvent.target || nativeEvent.srcElement || window; // Normalize SVG
<use> element events #4963

    if (target.correspondingUseElement) {
      target = target.correspondingUseElement;
    } // Safari may fire events on text nodes (Node.TEXT_NODE is 3).
    // @see http://www.quirksmode.org/js/events_properties.html


    return target.nodeType === TEXT_NODE ? target.parentNode : target;
  }

  var restoreImpl = null;
  var restoreTarget = null;
  var restoreQueue = null;

  function restoreStateOfTarget(target) {
    // We perform this translation at the end of the event loop so that we
    // always receive the correct fiber here
    var internalInstance = getInstanceFromNode(target);

    if (!internalInstance) {
      // Unmounted
      return;
    }

    if (typeof restoreImpl !== 'function') {
      throw new Error('setRestoreImplementation() needs to be called to handle a target
for controlled ' + 'events. This error is likely caused by a bug in React. Please file an
issue.');
    }

    var stateNode = internalInstance.stateNode; // Guard against Fiber being unmounted.

    if (stateNode) {
      var _props = getFiberCurrentPropsFromNode(stateNode);

      restoreImpl(internalInstance.stateNode, internalInstance.type, _props);
    }
  }

  function setRestoreImplementation(impl) {
    restoreImpl = impl;
  }
  function enqueueStateRestore(target) {
    if (restoreTarget) {
      if (restoreQueue) {
        restoreQueue.push(target);
      } else {
        restoreQueue = [target];
      }
    } else {
      restoreTarget = target;
    }
  }
  function needsStateRestore() {
    return restoreTarget !== null || restoreQueue !== null;
  }
  function restoreStateIfNeeded() {
    if (!restoreTarget) {
      return;
    }

    var target = restoreTarget;
    var queuedTargets = restoreQueue;
    restoreTarget = null;
    restoreQueue = null;
    restoreStateOfTarget(target);

    if (queuedTargets) {
      for (var i = 0; i < queuedTargets.length; i++) {
        restoreStateOfTarget(queuedTargets[i]);
      }
    }
  }

  // the renderer. Such as when we're dispatching events or if third party
  // libraries need to call batchedUpdates. Eventually, this API will go away when
  // everything is batched by default. We'll then have a similar API to opt-out of
  // scheduled work and instead do synchronous work.
  // Defaults

  var batchedUpdatesImpl = function (fn, bookkeeping) {
    return fn(bookkeeping);
  };

  var flushSyncImpl = function () {};

  var isInsideEventHandler = false;

  function finishEventHandler() {
    // Here we wait until all updates have propagated, which is important
    // when using controlled components within layers:
    // https://github.com/facebook/react/issues/1698
    // Then we restore state of any controlled component.
    var controlledComponentsHavePendingUpdates = needsStateRestore();
```

```
      if (controlledComponentsHavePendingUpdates) {
        // If a controlled event was fired, we may need to restore the state of
        // the DOM node back to the controlled value. This is necessary when React
        // bails out of the update without touching the DOM.
        // TODO: Restore state in the microtask, after the discrete updates flush,
        // instead of early flushing them here.
        flushSyncImpl();
        restoreStateIfNeeded();
      }
    }

    function batchedUpdates(fn, a, b) {
      if (isInsideEventHandler) {
        // If we are currently inside another batch, we need to wait until it
        // fully completes before restoring state.
        return fn(a, b);
      }

      isInsideEventHandler = true;

      try {
        return batchedUpdatesImpl(fn, a, b);
      } finally {
        isInsideEventHandler = false;
        finishEventHandler();
      }
    } // TODO: Replace with flushSync
    function setBatchingImplementation(_batchedUpdatesImpl, _discreteUpdatesImpl,
  _flushSyncImpl) {
      batchedUpdatesImpl = _batchedUpdatesImpl;
      flushSyncImpl = _flushSyncImpl;
    }

    function isInteractive(tag) {
      return tag === 'button' || tag === 'input' || tag === 'select' || tag === 'textarea';
    }

    function shouldPreventMouseEvent(name, type, props) {
      switch (name) {
        case 'onClick':
        case 'onClickCapture':
        case 'onDoubleClick':
        case 'onDoubleClickCapture':
        case 'onMouseDown':
        case 'onMouseDownCapture':
        case 'onMouseMove':
        case 'onMouseMoveCapture':
        case 'onMouseUp':
        case 'onMouseUpCapture':
        case 'onMouseEnter':
          return !!(props.disabled && isInteractive(type));

        default:
          return false;
      }
    }
    /**
     * @param {object} inst The instance, which is the source of events.
     * @param {string} registrationName Name of listener (e.g. `onClick`).
     * @return {?function} The stored callback.
     */


    function getListener(inst, registrationName) {
      var stateNode = inst.stateNode;

      if (stateNode === null) {
        // Work in progress (ex: onload events in incremental mode).
        return null;
      }

      var props = getFiberCurrentPropsFromNode(stateNode);

      if (props === null) {
        // Work in progress.
        return null;
      }

      var listener = props[registrationName];

      if (shouldPreventMouseEvent(registrationName, inst.type, props)) {
        return null;
      }

      if (listener && typeof listener !== 'function') {
        throw new Error("Expected `" + registrationName + "` listener to be a function,
  instead got a value of `" + typeof listener + "` type.");
      }

      return listener;
    }

    var passiveBrowserEventsSupported = false; // Check if browser support events with
  passive listeners
    // https://developer.mozilla.org/en-
  US/docs/Web/API/EventTarget/addEventListener#Safely_detecting_option_support

    if (canUseDOM) {
      try {
        var options = {}; // $FlowFixMe: Ignore Flow complaining about needing a value

        Object.defineProperty(options, 'passive', {
          get: function () {
            passiveBrowserEventsSupported = true;
          }
        });
        window.addEventListener('test', options, options);
        window.removeEventListener('test', options, options);
      } catch (e) {
        passiveBrowserEventsSupported = false;
      }
    }

    function invokeGuardedCallbackProd(name, func, context, a, b, c, d, e, f) {
      var funcArgs = Array.prototype.slice.call(arguments, 3);

      try {
        func.apply(context, funcArgs);
      } catch (error) {
        this.onError(error);
      }
    }

    var invokeGuardedCallbackImpl = invokeGuardedCallbackProd;

    {
      // In DEV mode, we swap out invokeGuardedCallback for a special version
      // that plays more nicely with the browser's DevTools. The idea is to preserve
      // "Pause on exceptions" behavior. Because React wraps all user-provided
      // functions in invokeGuardedCallback, and the production version of
      // invokeGuardedCallback uses a try-catch, all user exceptions are treated
      // like caught exceptions, and the DevTools won't pause unless the developer
      // takes the extra step of enabling pause on caught exceptions. This is
      // unintuitive, though, because even though React has caught the error, from
      // the developer's perspective, the error is uncaught.
      //
      // To preserve the expected "Pause on exceptions" behavior, we don't use a
      // try-catch in DEV. Instead, we synchronously dispatch a fake event to a fake
      // DOM node, and call the user-provided callback from inside an event handler
      // for that fake event. If the callback throws, the error is "captured" using
      // a global event handler. But because the error happens in a different
      // event loop context, it does not interrupt the normal program flow.
      // Effectively, this gives us try-catch behavior without actually using
      // try-catch. Neat!
      // Check that the browser supports the APIs we need to implement our special
      // DEV version of invokeGuardedCallback
      if (typeof window !== 'undefined' && typeof window.dispatchEvent === 'function' &&
  typeof document !== 'undefined' && typeof document.createEvent === 'function') {
        var fakeNode = document.createElement('react');

        invokeGuardedCallbackImpl = function invokeGuardedCallbackDev(name, func, context,
  a, b, c, d, e, f) {
          // If document doesn't exist we know for sure we will crash in this method
          // when we call document.createEvent(). However this can cause confusing
          // errors: https://github.com/facebook/create-react-app/issues/3482
          // So we preemptively throw with a better message instead.
          if (typeof document === 'undefined' || document === null) {
            throw new Error('The `document` global was defined when React was initialized,
  but is not ' + 'defined anymore. This can happen in a test environment if a component ' +
  'schedules an update from an asynchronous callback, but the test has already ' +
  'finished running. To solve this, you can either unmount the component at ' + 'the end of
  your test (and ensure that any asynchronous operations get ' + 'canceled in
  `componentWillUnmount`), or you can change the test itself ' + 'to be asynchronous.');
          }

          var evt = document.createEvent('Event');
          var didCall = false; // Keeps track of whether the user-provided callback threw
  an error. We
          // set this to true at the beginning, then set it to false right after
          // calling the function. If the function errors, `didError` will never be
          // set to false. This strategy works even if the browser is flaky and
          // fails to call our global error handler, because it doesn't rely on
          // the error event at all.

          var didError = true; // Keeps track of the value of window.event so that we can
  reset it
          // during the callback to let user code access window.event in the
          // browsers that support it.

          var windowEvent = window.event; // Keeps track of the descriptor of window.event
  to restore it after event
          // dispatching: https://github.com/facebook/react/issues/13688

          var windowEventDescriptor = Object.getOwnPropertyDescriptor(window, 'event');

          function restoreAfterDispatch() {
            // We immediately remove the callback from event listeners so that
            // nested `invokeGuardedCallback` calls do not clash. Otherwise, a
            // nested call would trigger the fake event handlers of any call higher
            // in the stack.
            fakeNode.removeEventListener(evtType, callCallback, false); // We check for
  window.hasOwnProperty('event') to prevent the
            // window.event assignment in both IE <= 10 as they throw an error
            // "Member not found" in strict mode, and in Firefox which does not
            // support window.event.

            if (typeof window.event !== 'undefined' && window.hasOwnProperty('event')) {
              window.event = windowEvent;
            }
          } // Create an event handler for our fake event. We will synchronously
          // dispatch our fake event using `dispatchEvent`. Inside the handler, we
          // call the user-provided callback.


          var funcArgs = Array.prototype.slice.call(arguments, 3);

          function callCallback() {
            didCall = true;
            restoreAfterDispatch();
            func.apply(context, funcArgs);
            didError = false;
          } // Create a global error event handler. We use this to capture the value
          // that was thrown. It's possible that this error handler will fire more
          // than once; for example, if non-React code also calls `dispatchEvent`
          // and a handler for that event throws. We should be resilient to most of
          // those cases. Even if our error event handler fires more than once, the
          // last error event is always used. If the callback actually does error,
          // we know that the last error event is the correct one, because it's not
          // possible for anything else to have happened in between our callback
          // erroring and the code that follows the `dispatchEvent` call below. If
          // the callback doesn't error, but the error event was fired, we know to
          // ignore it because `didError` will be false, as described above.


          var error; // Use this to track whether the error event is ever called.

          var didSetError = false;
          var isCrossOriginError = false;

          function handleWindowError(event) {
            error = event.error;
            didSetError = true;

            if (error === null && event.colno === 0 && event.lineno === 0) {
              isCrossOriginError = true;
            }

            if (event.defaultPrevented) {
              // Some other error handler has prevented default.
              // Browsers silence the error report if this happens.
              // We'll remember this to later decide whether to log it or not.
              if (error != null && typeof error === 'object') {
                try {
                  error._suppressLogging = true;
                } catch (inner) {// Ignore.
                }
              }
            }
          } // Create a fake event type.


          var evtType = "react-" + (name ? name : 'invokeguardedcallback'); // Attach our
  event handlers

          window.addEventListener('error', handleWindowError);
          fakeNode.addEventListener(evtType, callCallback, false); // Synchronously
  dispatch our fake event. If the user-provided function
          // errors, it will trigger our global error handler.

          evt.initEvent(evtType, false, false);
          fakeNode.dispatchEvent(evt);

          if (windowEventDescriptor) {
            Object.defineProperty(window, 'event', windowEventDescriptor);
          }

          if (didCall && didError) {
            if (!didSetError) {
              // The callback errored, but the error event never fired.
              // eslint-disable-next-line react-internal/prod-error-codes
              error = new Error('An error was thrown inside one of your components, but
  React ' + "doesn't know what it was. This is likely due to browser " + 'flakiness. React
  does its best to preserve the "Pause on ' + 'exceptions" behavior of the DevTools, which
  requires some ' + "DEV-mode only tricks. It's possible that these don't work in " + 'your
  browser. Try triggering the error in production mode, ' + 'or switching to a modern
  browser. If you suspect that this is ' + 'actually an issue with React, please file an
  issue.');
            } else if (isCrossOriginError) {
              // eslint-disable-next-line react-internal/prod-error-codes
              error = new Error("A cross-origin error was thrown. React doesn't have access
  to " + 'the actual error object in development. ' + 'See
  https://reactjs.org/link/crossorigin-error for more information.');
            }

            this.onError(error);
          } // Remove our event listeners


          window.removeEventListener('error', handleWindowError);

          if (!didCall) {
            // Something went really wrong, and our event was not dispatched.
            // https://github.com/facebook/react/issues/16734
            // https://github.com/facebook/react/issues/16585
            // Fall back to the production implementation.
            restoreAfterDispatch();
            return invokeGuardedCallbackProd.apply(this, arguments);
          }
        };
      }
    }

    var invokeGuardedCallbackImpl$1 = invokeGuardedCallbackImpl;

    var hasError = false;
    var caughtError = null; // Used by event system to capture/rethrow the first error.

    var hasRethrowError = false;
    var rethrowError = null;
    var reporter = {
      onError: function (error) {
        hasError = true;
        caughtError = error;
      }
    };
    /**
     * Call a function while guarding against errors that happens within it.
     * Returns an error if it throws, otherwise null.
     *
     * In production, this is implemented using a try-catch. The reason we don't
     * use a try-catch directly is so that we can swap out a different
     * implementation in DEV mode.
     *
     * @param {String} name of the guard to use for logging or debugging
     * @param {Function} func The function to invoke
     * @param {*} context The context to use when calling the function
     * @param {...*} args Arguments for function
     */

    function invokeGuardedCallback(name, func, context, a, b, c, d, e, f) {
      hasError = false;
      caughtError = null;
      invokeGuardedCallbackImpl$1.apply(reporter, arguments);
    }
    /**
     * Same as invokeGuardedCallback, but instead of returning an error, it stores
     * it in a global so it can be rethrown by `rethrowCaughtError` later.
     * TODO: See if caughtError and rethrowError can be unified.
     *
     * @param {String} name of the guard to use for logging or debugging
     * @param {Function} func The function to invoke
     * @param {*} context The context to use when calling the function
     * @param {...*} args Arguments for function
     */
    function invokeGuardedCallbackAndCatchFirstError(name, func, context, a, b, c, d, e, f)
  {
      invokeGuardedCallback.apply(this, arguments);

      if (hasError) {
        var error = clearCaughtError();

        if (!hasRethrowError) {
          hasRethrowError = true;
          rethrowError = error;
        }
      }
    }
    /**
     * During execution of guarded functions we will capture the first error which
     * we will rethrow to be handled by the top level error handler.
     */

    function rethrowCaughtError() {
      if (hasRethrowError) {
        var error = rethrowError;
        hasRethrowError = false;
        rethrowError = null;
        throw error;
      }
    }
    function hasCaughtError() {
      return hasError;
    }
    function clearCaughtError() {
      if (hasError) {
        var error = caughtError;
        hasError = false;
        caughtError = null;
        return error;
      } else {
        throw new Error('clearCaughtError was called but no error was captured. This error
  ' + 'is likely caused by a bug in React. Please file an issue.');
      }
    }

    var ReactInternals = React.__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED;
    var _ReactInternals$Sched = ReactInternals.Scheduler,
        unstable_cancelCallback = _ReactInternals$Sched.unstable_cancelCallback,
        unstable_now = _ReactInternals$Sched.unstable_now,
        unstable_scheduleCallback = _ReactInternals$Sched.unstable_scheduleCallback,
        unstable_shouldYield = _ReactInternals$Sched.unstable_shouldYield,
        unstable_requestPaint = _ReactInternals$Sched.unstable_requestPaint,
        unstable_getFirstCallbackNode =
  _ReactInternals$Sched.unstable_getFirstCallbackNode,
        unstable_runWithPriority = _ReactInternals$Sched.unstable_runWithPriority,
        unstable_next = _ReactInternals$Sched.unstable_next,
        unstable_continueExecution = _ReactInternals$Sched.unstable_continueExecution,
        unstable_pauseExecution = _ReactInternals$Sched.unstable_pauseExecution,
        unstable_getCurrentPriorityLevel =
  _ReactInternals$Sched.unstable_getCurrentPriorityLevel,
        unstable_ImmediatePriority = _ReactInternals$Sched.unstable_ImmediatePriority,
        unstable_UserBlockingPriority =
  _ReactInternals$Sched.unstable_UserBlockingPriority,
        unstable_NormalPriority = _ReactInternals$Sched.unstable_NormalPriority,
        unstable_LowPriority = _ReactInternals$Sched.unstable_LowPriority,
        unstable_IdlePriority = _ReactInternals$Sched.unstable_IdlePriority,
        unstable_forceFrameRate = _ReactInternals$Sched.unstable_forceFrameRate,
        unstable_flushAllWithoutAsserting =
  _ReactInternals$Sched.unstable_flushAllWithoutAsserting,
        unstable_yieldValue = _ReactInternals$Sched.unstable_yieldValue,
        unstable_setDisableYieldValue =
  _ReactInternals$Sched.unstable_setDisableYieldValue;
```

```
  /**
   * `ReactInstanceMap` maintains a mapping from a public facing stateful
   * instance (key) and the internal representation (value). This allows public
   * methods to accept the user facing instance as an argument and map them back
   * to internal methods.
   *
   * Note that this module is currently shared and assumed to be stateless.
   * If this becomes an actual Map, that will break.
   */
  function get(key) {
    return key._reactInternals;
  }
  function has(key) {
    return key._reactInternals !== undefined;
  }
  function set(key, value) {
    key._reactInternals = value;
  }

  // Don't change these two values. They're used by React Dev Tools.
  var NoFlags =
  /*                      */
  0;
  var PerformedWork =
  /*                */
  1; // You can change the rest (and add more).

  var Placement =
  /*                    */
  2;
  var Update =
  /*                       */
  4;
  var ChildDeletion =
  /*                */
  16;
  var ContentReset =
  /*                 */
  32;
  var Callback =
  /*                     */
  64;
  var DidCapture =
  /*                   */
  128;
  var ForceClientRender =
  /*            */
  256;
  var Ref =
  /*                          */
  512;
  var Snapshot =
  /*                     */
  1024;
  var Passive =
  /*                      */
  2048;
  var Hydrating =
  /*                    */
  4096;
  var Visibility =
  /*                   */
  8192;
  var StoreConsistency =
  /*             */
  16384;
  var LifecycleEffectMask = Passive | Update | Callback | Ref | Snapshot |
StoreConsistency; // Union of all commit flags (flags with the lifetime of a particular
commit)
```

```
  var HostEffectMask =
  /*              */
  32767; // These are not really side effects, but we still reuse this field.

  var Incomplete =
  /*                 */
  32768;
  var ShouldCapture =
  /*              */
  65536;
  var ForceUpdateForLegacySuspense =
  /* */
  131072;
  var Forked =
  /*                       */
  1048576; // Static tags describe aspects of a fiber that are not specific to a render,
  // e.g. a fiber uses a passive effect (even if there are no updates on this particular
render).
  // This enables us to defer more work in the unmount case,
  // since we can defer traversing the tree during layout to look for Passive effects,
  // and instead rely on the static flag as a signal that there may be cleanup work.

  var RefStatic =
  /*                    */
  2097152;
  var LayoutStatic =
  /*              */
  4194304;
  var PassiveStatic =
  /*              */
  8388608; // These flags allow us to traverse to fibers that have effects on mount
  // without traversing the entire tree after every commit for
  // double invoking

  var MountLayoutDev =
  /*              */
  16777216;
  var MountPassiveDev =
  /*             */
  33554432; // Groups of flags that are used in the commit phase to skip over trees that
  // don't contain effects, by checking subtreeFlags.

  var BeforeMutationMask = // TODO: Remove Update flag from before mutation phase by re-
landing Visibility
  // flag logic (see #20043)
  Update | Snapshot | ( 0);
  var MutationMask = Placement | Update | ChildDeletion | ContentReset | Ref | Hydrating
| Visibility;
  var LayoutMask = Update | Callback | Ref | Visibility; // TODO: Split into
PassiveMountMask and PassiveUnmountMask

  var PassiveMask = Passive | ChildDeletion; // Union of tags that don't get reset on
clones.
  // This allows certain concepts to persist without recalculating them,
  // e.g. whether a subtree contains passive effects or portals.

  var StaticMask = LayoutStatic | PassiveStatic | RefStatic;

  var ReactCurrentOwner = ReactSharedInternals.ReactCurrentOwner;
  function getNearestMountedFiber(fiber) {
    var node = fiber;
    var nearestMounted = fiber;

    if (!fiber.alternate) {
      // If there is no alternate, this might be a new tree that isn't inserted
      // yet. If it is, then it will have a pending insertion effect on it.
      var nextNode = node;
      do {
        node = nextNode;

        if ((node.flags & (Placement | Hydrating)) !== NoFlags) {
          // This is an insertion or in-progress hydration. The nearest possible
          // mounted fiber is the parent but we need to continue to figure out
          // if that one is still mounted.
          nearestMounted = node.return;
        }

        nextNode = node.return;
      } while (nextNode);
    } else {
      while (node.return) {
        node = node.return;
      }
    }

    if (node.tag === HostRoot) {
      // TODO: Check if this was a nested HostRoot when used with
      // renderContainerIntoSubtree.
      return nearestMounted;
    } // If we didn't hit the root, that means that we're in an disconnected tree
    // that has been unmounted.


    return null;
  }
  function getSuspenseInstanceFromFiber(fiber) {
    if (fiber.tag === SuspenseComponent) {
      var suspenseState = fiber.memoizedState;

      if (suspenseState === null) {
        var current = fiber.alternate;

        if (current !== null) {
          suspenseState = current.memoizedState;
        }
      }

      if (suspenseState !== null) {
        return suspenseState.dehydrated;
      }
    }

    return null;
  }
  function getContainerFromFiber(fiber) {
    return fiber.tag === HostRoot ? fiber.stateNode.containerInfo : null;
  }
  function isFiberMounted(fiber) {
    return getNearestMountedFiber(fiber) === fiber;
  }
  function isMounted(component) {
    {
      var owner = ReactCurrentOwner.current;

      if (owner !== null && owner.tag === ClassComponent) {
        var ownerFiber = owner;
        var instance = ownerFiber.stateNode;

        if (!instance._warnedAboutRefsInRender) {
          error('%s is accessing isMounted inside its render() function. ' + 'render()
should be a pure function of props and state. It should ' + 'never access something that
requires stale data from the previous ' + 'render, such as refs. Move this logic to
componentDidMount and ' + 'componentDidUpdate instead.',
getComponentNameFromFiber(ownerFiber) || 'A component');
        }
```

```
          instance._warnedAboutRefsInRender = true;
        }
      }

      var fiber = get(component);

      if (!fiber) {
        return false;
      }

      return getNearestMountedFiber(fiber) === fiber;
    }

    function assertIsMounted(fiber) {
      if (getNearestMountedFiber(fiber) !== fiber) {
        throw new Error('Unable to find node on an unmounted component.');
      }
    }

    function findCurrentFiberUsingSlowPath(fiber) {
      var alternate = fiber.alternate;

      if (!alternate) {
        // If there is no alternate, then we only need to check if it is mounted.
        var nearestMounted = getNearestMountedFiber(fiber);

        if (nearestMounted === null) {
          throw new Error('Unable to find node on an unmounted component.');
        }

        if (nearestMounted !== fiber) {
          return null;
        }

        return fiber;
      } // If we have two possible branches, we'll walk backwards up to the root
      // to see what path the root points to. On the way we may hit one of the
      // special cases and we'll deal with them.


      var a = fiber;
      var b = alternate;

      while (true) {
        var parentA = a.return;

        if (parentA === null) {
          // We're at the root.
          break;
        }

        var parentB = parentA.alternate;

        if (parentB === null) {
          // There is no alternate. This is an unusual case. Currently, it only
          // happens when a Suspense component is hidden. An extra fragment fiber
          // is inserted in between the Suspense fiber and its children. Skip
          // over this extra fragment fiber and proceed to the next parent.
          var nextParent = parentA.return;

          if (nextParent !== null) {
            a = b = nextParent;
            continue;
          } // If there's no parent, we're at the root.


          break;
        } // If both copies of the parent fiber point to the same child, we can
        // assume that the child is current. This happens when we bailout on low
        // priority: the bailed out fiber's child reuses the current child.


        if (parentA.child === parentB.child) {
          var child = parentA.child;

          while (child) {
            if (child === a) {
              // We've determined that A is the current branch.
              assertIsMounted(parentA);
              return fiber;
            }

            if (child === b) {
              // We've determined that B is the current branch.
              assertIsMounted(parentA);
              return alternate;
            }

            child = child.sibling;
          } // We should never have an alternate for any mounting node. So the only
          // way this could possibly happen is if this was unmounted, if at all.


          throw new Error('Unable to find node on an unmounted component.');
        }

        if (a.return !== b.return) {
          // The return pointer of A and the return pointer of B point to different
          // fibers. We assume that return pointers never criss-cross, so A must
          // belong to the child set of A.return, and B must belong to the child
          // set of B.return.
          a = parentA;
          b = parentB;
        } else {
          // The return pointers point to the same fiber. We'll have to use the
          // default, slow path: scan the child sets of each parent alternate to see
          // which child belongs to which set.
          //
          // Search parent A's child set
          var didFindChild = false;
          var _child = parentA.child;

          while (_child) {
            if (_child === a) {
              didFindChild = true;
              a = parentA;
              b = parentB;
              break;
            }

            if (_child === b) {
              didFindChild = true;
              b = parentA;
              a = parentB;
              break;
            }

            _child = _child.sibling;
          }

          if (!didFindChild) {
            // Search parent B's child set
            _child = parentB.child;

            while (_child) {
              if (_child === a) {
                didFindChild = true;
                a = parentB;
                b = parentA;
                break;
              }

              if (_child === b) {
                didFindChild = true;
                b = parentB;
                a = parentA;
                break;
              }

              _child = _child.sibling;
            }

            if (!didFindChild) {
              throw new Error('Child was not found in either parent set. This indicates a
  bug ' + 'in React related to the return pointer. Please file an issue.');
            }
          }
        }

        if (a.alternate !== b) {
          throw new Error("Return fibers should always be each others' alternates. " +
  'This error is likely caused by a bug in React. Please file an issue.');
        }
      } // If the root is not a host container, we're in a disconnected tree. I.e.
      // unmounted.


      if (a.tag !== HostRoot) {
        throw new Error('Unable to find node on an unmounted component.');
      }

      if (a.stateNode.current === a) {
        // We've determined that A is the current branch.
        return fiber;
      } // Otherwise B has to be current branch.


      return alternate;
    }
    function findCurrentHostFiber(parent) {
      var currentParent = findCurrentFiberUsingSlowPath(parent);
      return currentParent !== null ? findCurrentHostFiberImpl(currentParent) : null;
    }

    function findCurrentHostFiberImpl(node) {
      // Next we'll drill down this component to find the first HostComponent/Text.
      if (node.tag === HostComponent || node.tag === HostText) {
        return node;
      }

      var child = node.child;

      while (child !== null) {
        var match = findCurrentHostFiberImpl(child);

        if (match !== null) {
          return match;
        }

        child = child.sibling;
      }

      return null;
    }

    function findCurrentHostFiberWithNoPortals(parent) {
      var currentParent = findCurrentFiberUsingSlowPath(parent);
```

```
      return currentParent !== null ? findCurrentHostFiberWithNoPortalsImpl(currentParent)
  : null;
    }

    function findCurrentHostFiberWithNoPortalsImpl(node) {
      // Next we'll drill down this component to find the first HostComponent/Text.
      if (node.tag === HostComponent || node.tag === HostText) {
        return node;
      }

      var child = node.child;

      while (child !== null) {
        if (child.tag !== HostPortal) {
          var match = findCurrentHostFiberWithNoPortalsImpl(child);

          if (match !== null) {
            return match;
          }
        }

        child = child.sibling;
      }

      return null;
    }

    // This module only exists as an ESM wrapper around the external CommonJS
    var scheduleCallback = unstable_scheduleCallback;
    var cancelCallback = unstable_cancelCallback;
    var shouldYield = unstable_shouldYield;
    var requestPaint = unstable_requestPaint;
    var now = unstable_now;
    var getCurrentPriorityLevel = unstable_getCurrentPriorityLevel;
    var ImmediatePriority = unstable_ImmediatePriority;
    var UserBlockingPriority = unstable_UserBlockingPriority;
    var NormalPriority = unstable_NormalPriority;
    var LowPriority = unstable_LowPriority;
    var IdlePriority = unstable_IdlePriority;
    // this doesn't actually exist on the scheduler, but it *does*
    // on scheduler/unstable_mock, which we'll need for internal testing
    var unstable_yieldValue$1 = unstable_yieldValue;
    var unstable_setDisableYieldValue$1 = unstable_setDisableYieldValue;

    var rendererID = null;
    var injectedHook = null;
    var injectedProfilingHooks = null;
    var hasLoggedError = false;
    var isDevToolsPresent = typeof __REACT_DEVTOOLS_GLOBAL_HOOK__ !== 'undefined';
    function injectInternals(internals) {
      if (typeof __REACT_DEVTOOLS_GLOBAL_HOOK__ === 'undefined') {
        // No DevTools
        return false;
      }

      var hook = __REACT_DEVTOOLS_GLOBAL_HOOK__;

      if (hook.isDisabled) {
        // This isn't a real property on the hook, but it can be set to opt out
        // of DevTools integration and associated warnings and logs.
        // https://github.com/facebook/react/issues/3877
        return true;
      }

      if (!hook.supportsFiber) {
        {
          error('The installed version of React DevTools is too old and will not work ' +
  'with the current version of React. Please update React DevTools. ' +
  'https://reactjs.org/link/react-devtools');
```

```
      } // DevTools exists, even though it doesn't support Fiber.


      return true;
    }

    try {
      if (enableSchedulingProfiler) {
        // Conditionally inject these hooks only if Timeline profiler is supported by
this build.
        // This gives DevTools a way to feature detect that isn't tied to version number
        // (since profiling and timeline are controlled by different feature flags).
        internals = assign({}, internals, {
          getLaneLabelMap: getLaneLabelMap,
          injectProfilingHooks: injectProfilingHooks
        });
      }

      rendererID = hook.inject(internals); // We have successfully injected, so now it is
safe to set up hooks.

      injectedHook = hook;
    } catch (err) {
      // Catch all errors because it is unsafe to throw during initialization.
      {
        error('React instrumentation encountered an error: %s.', err);
      }
    }

    if (hook.checkDCE) {
      // This is the real DevTools.
      return true;
    } else {
      // This is likely a hook installed by Fast Refresh runtime.
      return false;
    }
  }
  function onScheduleRoot(root, children) {
    {
      if (injectedHook && typeof injectedHook.onScheduleFiberRoot === 'function') {
        try {
          injectedHook.onScheduleFiberRoot(rendererID, root, children);
        } catch (err) {
          if ( !hasLoggedError) {
            hasLoggedError = true;

            error('React instrumentation encountered an error: %s', err);
          }
        }
      }
    }
  }
  function onCommitRoot(root, eventPriority) {
    if (injectedHook && typeof injectedHook.onCommitFiberRoot === 'function') {
      try {
        var didError = (root.current.flags & DidCapture) === DidCapture;

        if (enableProfilerTimer) {
          var schedulerPriority;

          switch (eventPriority) {
            case DiscreteEventPriority:
              schedulerPriority = ImmediatePriority;
              break;

            case ContinuousEventPriority:
              schedulerPriority = UserBlockingPriority;
              break;
            case DefaultEventPriority:
              schedulerPriority = NormalPriority;
              break;

            case IdleEventPriority:
              schedulerPriority = IdlePriority;
              break;

            default:
              schedulerPriority = NormalPriority;
              break;
          }

          injectedHook.onCommitFiberRoot(rendererID, root, schedulerPriority, didError);
        } else {
          injectedHook.onCommitFiberRoot(rendererID, root, undefined, didError);
        }
      } catch (err) {
        {
          if (!hasLoggedError) {
            hasLoggedError = true;

            error('React instrumentation encountered an error: %s', err);
          }
        }
      }
    }
  }
  function onPostCommitRoot(root) {
    if (injectedHook && typeof injectedHook.onPostCommitFiberRoot === 'function') {
      try {
        injectedHook.onPostCommitFiberRoot(rendererID, root);
      } catch (err) {
        {
          if (!hasLoggedError) {
            hasLoggedError = true;

            error('React instrumentation encountered an error: %s', err);
          }
        }
      }
    }
  }
  function onCommitUnmount(fiber) {
    if (injectedHook && typeof injectedHook.onCommitFiberUnmount === 'function') {
      try {
        injectedHook.onCommitFiberUnmount(rendererID, fiber);
      } catch (err) {
        {
          if (!hasLoggedError) {
            hasLoggedError = true;

            error('React instrumentation encountered an error: %s', err);
          }
        }
      }
    }
  }
  function setIsStrictModeForDevtools(newIsStrictMode) {
    {
      if (typeof unstable_yieldValue$1 === 'function') {
        // We're in a test because Scheduler.unstable_yieldValue only exists
        // in SchedulerMock. To reduce the noise in strict mode tests,
        // suppress warnings and disable scheduler yielding during the double render
        unstable_setDisableYieldValue$1(newIsStrictMode);
        setSuppressWarning(newIsStrictMode);
      }

      if (injectedHook && typeof injectedHook.setStrictMode === 'function') {
        try {
          injectedHook.setStrictMode(rendererID, newIsStrictMode);
        } catch (err) {
          {
            if (!hasLoggedError) {
              hasLoggedError = true;

              error('React instrumentation encountered an error: %s', err);
            }
          }
        }
      }
    }
  } // Profiler API hooks

  function injectProfilingHooks(profilingHooks) {
    injectedProfilingHooks = profilingHooks;
  }

  function getLaneLabelMap() {
    {
      var map = new Map();
      var lane = 1;

      for (var index = 0; index < TotalLanes; index++) {
        var label = getLabelForLane(lane);
        map.set(lane, label);
        lane *= 2;
      }

      return map;
    }
  }

  function markCommitStarted(lanes) {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markCommitStarted === 'function') {
        injectedProfilingHooks.markCommitStarted(lanes);
      }
    }
  }
  function markCommitStopped() {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markCommitStopped === 'function') {
        injectedProfilingHooks.markCommitStopped();
      }
    }
  }
  function markComponentRenderStarted(fiber) {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentRenderStarted === 'function') {
        injectedProfilingHooks.markComponentRenderStarted(fiber);
      }
    }
  }
  function markComponentRenderStopped() {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentRenderStopped === 'function') {
        injectedProfilingHooks.markComponentRenderStopped();
      }
    }
  }
  function markComponentPassiveEffectMountStarted(fiber) {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentPassiveEffectMountStarted === 'function') {
        injectedProfilingHooks.markComponentPassiveEffectMountStarted(fiber);
      }
    }
  }
  function markComponentPassiveEffectMountStopped() {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentPassiveEffectMountStopped === 'function') {
        injectedProfilingHooks.markComponentPassiveEffectMountStopped();
      }
    }
  }
  function markComponentPassiveEffectUnmountStarted(fiber) {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentPassiveEffectUnmountStarted === 'function') {
        injectedProfilingHooks.markComponentPassiveEffectUnmountStarted(fiber);
      }
    }
  }
  function markComponentPassiveEffectUnmountStopped() {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentPassiveEffectUnmountStopped === 'function') {
        injectedProfilingHooks.markComponentPassiveEffectUnmountStopped();
      }
    }
  }
  function markComponentLayoutEffectMountStarted(fiber) {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentLayoutEffectMountStarted === 'function') {
        injectedProfilingHooks.markComponentLayoutEffectMountStarted(fiber);
      }
    }
  }
  function markComponentLayoutEffectMountStopped() {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentLayoutEffectMountStopped === 'function') {
        injectedProfilingHooks.markComponentLayoutEffectMountStopped();
      }
    }
  }
  function markComponentLayoutEffectUnmountStarted(fiber) {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentLayoutEffectUnmountStarted === 'function') {
        injectedProfilingHooks.markComponentLayoutEffectUnmountStarted(fiber);
      }
    }
  }
  function markComponentLayoutEffectUnmountStopped() {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentLayoutEffectUnmountStopped === 'function') {
        injectedProfilingHooks.markComponentLayoutEffectUnmountStopped();
      }
    }
  }
  function markComponentErrored(fiber, thrownValue, lanes) {
    {
      if (injectedProfilingHooks !== null && typeof
injectedProfilingHooks.markComponentErrored === 'function') {
        injectedProfilingHooks.markComponentErrored(fiber, thrownValue, lanes);
      }
    }
  }
```

```
    function markComponentSuspended(fiber, wakeable, lanes) {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markComponentSuspended === 'function') {
          injectedProfilingHooks.markComponentSuspended(fiber, wakeable, lanes);
        }
      }
    }
    function markLayoutEffectsStarted(lanes) {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markLayoutEffectsStarted === 'function') {
          injectedProfilingHooks.markLayoutEffectsStarted(lanes);
        }
      }
    }
    function markLayoutEffectsStopped() {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markLayoutEffectsStopped === 'function') {
          injectedProfilingHooks.markLayoutEffectsStopped();
        }
      }
    }
    function markPassiveEffectsStarted(lanes) {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markPassiveEffectsStarted === 'function') {
          injectedProfilingHooks.markPassiveEffectsStarted(lanes);
        }
      }
    }
    function markPassiveEffectsStopped() {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markPassiveEffectsStopped === 'function') {
          injectedProfilingHooks.markPassiveEffectsStopped();
        }
      }
    }
    function markRenderStarted(lanes) {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markRenderStarted === 'function') {
          injectedProfilingHooks.markRenderStarted(lanes);
        }
      }
    }
    function markRenderYielded() {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markRenderYielded === 'function') {
          injectedProfilingHooks.markRenderYielded();
        }
      }
    }
    function markRenderStopped() {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markRenderStopped === 'function') {
          injectedProfilingHooks.markRenderStopped();
        }
      }
    }
    function markRenderScheduled(lane) {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markRenderScheduled === 'function') {
          injectedProfilingHooks.markRenderScheduled(lane);
        }
      }
    }
    function markForceUpdateScheduled(fiber, lane) {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markForceUpdateScheduled === 'function') {
          injectedProfilingHooks.markForceUpdateScheduled(fiber, lane);
        }
      }
    }
    function markStateUpdateScheduled(fiber, lane) {
      {
        if (injectedProfilingHooks !== null && typeof
  injectedProfilingHooks.markStateUpdateScheduled === 'function') {
          injectedProfilingHooks.markStateUpdateScheduled(fiber, lane);
        }
      }
    }

    var NoMode =
    /*                         */
    0; // TODO: Remove ConcurrentMode by reading from the root tag instead

    var ConcurrentMode =
    /*                 */
    1;
    var ProfileMode =
    /*                    */
    2;
    var StrictLegacyMode =
    /*               */
    8;
    var StrictEffectsMode =
    /*              */
    16;

    // TODO: This is pretty well supported by browsers. Maybe we can drop it.
    var clz32 = Math.clz32 ? Math.clz32 : clz32Fallback; // Count leading zeros.
    // Based on:
    // https://developer.mozilla.org/en-
  US/docs/Web/JavaScript/Reference/Global_Objects/Math/clz32

    var log = Math.log;
    var LN2 = Math.LN2;

    function clz32Fallback(x) {
      var asUint = x >>> 0;

      if (asUint === 0) {
        return 32;
      }

      return 31 - (log(asUint) / LN2 | 0) | 0;
    }

    // If those values are changed that package should be rebuilt and redeployed.

    var TotalLanes = 31;
    var NoLanes =
    /*                        */
    0;
    var NoLane =
    /*                          */
    0;
    var SyncLane =
    /*                        */
    1;
    var InputContinuousHydrationLane =
    /*    */
    2;
    var InputContinuousLane =
    /*             */
    4;
    var DefaultHydrationLane =
    /*            */
    8;
    var DefaultLane =
    /*                     */
    16;
    var TransitionHydrationLane =
    /*                */
    32;
    var TransitionLanes =
    /*                       */
    4194240;
    var TransitionLane1 =
    /*                        */
    64;
    var TransitionLane2 =
    /*                        */
    128;
    var TransitionLane3 =
    /*                        */
    256;
    var TransitionLane4 =
    /*                        */
    512;
    var TransitionLane5 =
    /*                        */
    1024;
    var TransitionLane6 =
    /*                        */
    2048;
    var TransitionLane7 =
    /*                        */
    4096;
    var TransitionLane8 =
    /*                        */
    8192;
    var TransitionLane9 =
    /*                        */
    16384;
    var TransitionLane10 =
    /*                       */
    32768;
    var TransitionLane11 =
    /*                       */
    65536;
    var TransitionLane12 =
    /*                       */
    131072;
    var TransitionLane13 =
    /*                       */
    262144;
    var TransitionLane14 =
    /*                       */
    524288;
    var TransitionLane15 =
    /*                       */
    1048576;
    var TransitionLane16 =
    /*                       */
    2097152;
    var RetryLanes =
    /*                            */
    130023424;
    var RetryLane1 =
    /*                              */
    4194304;
    var RetryLane2 =
    /*                              */
    8388608;
    var RetryLane3 =
    /*                              */
    16777216;
    var RetryLane4 =
    /*                              */
    33554432;
    var RetryLane5 =
    /*                              */
    67108864;
    var SomeRetryLane = RetryLane1;
    var SelectiveHydrationLane =
    /*          */
    134217728;
    var NonIdleLanes =
    /*                           */
    268435455;
    var IdleHydrationLane =
    /*               */
    268435456;
    var IdleLane =
    /*                         */
    536870912;
    var OffscreenLane =
    /*                    */
    1073741824; // This function is used for the experimental timeline (react-devtools-
  timeline)
    // It should be kept in sync with the Lanes values above.

    function getLabelForLane(lane) {
      {
        if (lane & SyncLane) {
          return 'Sync';
        }

        if (lane & InputContinuousHydrationLane) {
          return 'InputContinuousHydration';
        }

        if (lane & InputContinuousLane) {
          return 'InputContinuous';
        }

        if (lane & DefaultHydrationLane) {
          return 'DefaultHydration';
        }

        if (lane & DefaultLane) {
          return 'Default';
        }

        if (lane & TransitionHydrationLane) {
          return 'TransitionHydration';
        }

        if (lane & TransitionLanes) {
          return 'Transition';
        }

        if (lane & RetryLanes) {
          return 'Retry';
        }

        if (lane & SelectiveHydrationLane) {
          return 'SelectiveHydration';
        }

        if (lane & IdleHydrationLane) {
          return 'IdleHydration';
        }

        if (lane & IdleLane) {
          return 'Idle';
        }

        if (lane & OffscreenLane) {
          return 'Offscreen';
        }
      }
    }
    var NoTimestamp = -1;
    var nextTransitionLane = TransitionLane1;
    var nextRetryLane = RetryLane1;

    function getHighestPriorityLanes(lanes) {
      switch (getHighestPriorityLane(lanes)) {
        case SyncLane:
          return SyncLane;

        case InputContinuousHydrationLane:
          return InputContinuousHydrationLane;

        case InputContinuousLane:
          return InputContinuousLane;

        case DefaultHydrationLane:
          return DefaultHydrationLane;

        case DefaultLane:
          return DefaultLane;

        case TransitionHydrationLane:
          return TransitionHydrationLane;

        case TransitionLane1:
        case TransitionLane2:
        case TransitionLane3:
        case TransitionLane4:
        case TransitionLane5:
        case TransitionLane6:
        case TransitionLane7:
        case TransitionLane8:
        case TransitionLane9:
        case TransitionLane10:
        case TransitionLane11:
        case TransitionLane12:
        case TransitionLane13:
        case TransitionLane14:
        case TransitionLane15:
        case TransitionLane16:
          return lanes & TransitionLanes;

        case RetryLane1:
        case RetryLane2:
        case RetryLane3:
        case RetryLane4:
        case RetryLane5:
          return lanes & RetryLanes;

        case SelectiveHydrationLane:
          return SelectiveHydrationLane;

        case IdleHydrationLane:
          return IdleHydrationLane;
        case IdleLane:
          return IdleLane;

        case OffscreenLane:
          return OffscreenLane;

        default:
          {
            error('Should have found matching lanes. This is a bug in React.');
          } // This shouldn't be reachable, but as a fallback, return the entire bitmask.


          return lanes;
      }
    }

    function getNextLanes(root, wipLanes) {
      // Early bailout if there's no pending work left.
      var pendingLanes = root.pendingLanes;

      if (pendingLanes === NoLanes) {
        return NoLanes;
      }

      var nextLanes = NoLanes;
      var suspendedLanes = root.suspendedLanes;
      var pingedLanes = root.pingedLanes; // Do not work on any idle work until all the
  non-idle work has finished,
      // even if the work is suspended.

      var nonIdlePendingLanes = pendingLanes & NonIdleLanes;

      if (nonIdlePendingLanes !== NoLanes) {
        var nonIdleUnblockedLanes = nonIdlePendingLanes & ~suspendedLanes;

        if (nonIdleUnblockedLanes !== NoLanes) {
          nextLanes = getHighestPriorityLanes(nonIdleUnblockedLanes);
        } else {
          var nonIdlePingedLanes = nonIdlePendingLanes & pingedLanes;

          if (nonIdlePingedLanes !== NoLanes) {
            nextLanes = getHighestPriorityLanes(nonIdlePingedLanes);
          }
        }
      } else {
        // The only remaining work is Idle.
        var unblockedLanes = pendingLanes & ~suspendedLanes;

        if (unblockedLanes !== NoLanes) {
          nextLanes = getHighestPriorityLanes(unblockedLanes);
        } else {
          if (pingedLanes !== NoLanes) {
            nextLanes = getHighestPriorityLanes(pingedLanes);
          }
        }
      }

      if (nextLanes === NoLanes) {
        // This should only be reachable if we're suspended
        // TODO: Consider warning in this path if a fallback timer is not scheduled.
        return NoLanes;
      } // If we're already in the middle of a render, switching lanes will interrupt
      // it and we'll lose our progress. We should only do this if the new lanes are
      // higher priority.


      if (wipLanes !== NoLanes && wipLanes !== nextLanes && // If we already suspended with
  a delay, then interrupting is fine. Don't
```

```
      // bother waiting until the root is complete.
      (wipLanes & suspendedLanes) === NoLanes) {
        var nextLane = getHighestPriorityLane(nextLanes);
        var wipLane = getHighestPriorityLane(wipLanes);

        if ( // Tests whether the next lane is equal or lower priority than the wip
        // one. This works because the bits decrease in priority as you go left.
        nextLane >= wipLane || // Default priority updates should not interrupt transition
  updates. The
        // only difference between default updates and transition updates is that
        // default updates do not support refresh transitions.
        nextLane === DefaultLane && (wipLane & TransitionLanes) !== NoLanes) {
          // Keep working on the existing in-progress tree. Do not interrupt.
          return wipLanes;
        }
      }

      if ((nextLanes & InputContinuousLane) !== NoLanes) {
        // When updates are sync by default, we entangle continuous priority updates
        // and default updates, so they render in the same batch. The only reason
        // they use separate lanes is because continuous updates should interrupt
        // transitions, but default updates should not.
        nextLanes |= pendingLanes & DefaultLane;
      } // Check for entangled lanes and add them to the batch.
      //
      // A lane is said to be entangled with another when it's not allowed to render
      // in a batch that does not also include the other lane. Typically we do this
      // when multiple updates have the same source, and we only want to respond to
      // the most recent event from that source.
      //
      // Note that we apply entanglements *after* checking for partial work above.
      // This means that if a lane is entangled during an interleaved event while
      // it's already rendering, we won't interrupt it. This is intentional, since
      // entanglement is usually "best effort": we'll try our best to render the
      // lanes in the same batch, but it's not worth throwing out partially
      // completed work in order to do it.
      // TODO: Reconsider this. The counter-argument is that the partial work
      // represents an intermediate state, which we don't want to show to the user.
      // And by spending extra time finishing it, we're increasing the amount of
      // time it takes to show the final state, which is what they are actually
      // waiting for.
      //
      // For those exceptions where entanglement is semantically important, like
      // useMutableSource, we should ensure that there is no partial work at the
      // time we apply the entanglement.


      var entangledLanes = root.entangledLanes;

      if (entangledLanes !== NoLanes) {
        var entanglements = root.entanglements;
        var lanes = nextLanes & entangledLanes;

        while (lanes > 0) {
          var index = pickArbitraryLaneIndex(lanes);
          var lane = 1 << index;
          nextLanes |= entanglements[index];
          lanes &= ~lane;
        }
      }

      return nextLanes;
    }
    function getMostRecentEventTime(root, lanes) {
      var eventTimes = root.eventTimes;
      var mostRecentEventTime = NoTimestamp;

      while (lanes > 0) {
        var index = pickArbitraryLaneIndex(lanes);
        var lane = 1 << index;
        var eventTime = eventTimes[index];

        if (eventTime > mostRecentEventTime) {
          mostRecentEventTime = eventTime;
        }

        lanes &= ~lane;
      }

      return mostRecentEventTime;
    }

    function computeExpirationTime(lane, currentTime) {
      switch (lane) {
        case SyncLane:
        case InputContinuousHydrationLane:
        case InputContinuousLane:
          // User interactions should expire slightly more quickly.
          //
          // NOTE: This is set to the corresponding constant as in Scheduler.js.
          // When we made it larger, a product metric in www regressed, suggesting
          // there's a user interaction that's being starved by a series of
          // synchronous updates. If that theory is correct, the proper solution is
          // to fix the starvation. However, this scenario supports the idea that
          // expiration times are an important safeguard when starvation
          // does happen.
          return currentTime + 250;

        case DefaultHydrationLane:
        case DefaultLane:
        case TransitionHydrationLane:
        case TransitionLane1:
        case TransitionLane2:
        case TransitionLane3:
        case TransitionLane4:
        case TransitionLane5:
        case TransitionLane6:
        case TransitionLane7:
        case TransitionLane8:
        case TransitionLane9:
        case TransitionLane10:
        case TransitionLane11:
        case TransitionLane12:
        case TransitionLane13:
        case TransitionLane14:
        case TransitionLane15:
        case TransitionLane16:
          return currentTime + 5000;

        case RetryLane1:
        case RetryLane2:
        case RetryLane3:
        case RetryLane4:
        case RetryLane5:
          // TODO: Retries should be allowed to expire if they are CPU bound for
          // too long, but when I made this change it caused a spike in browser
          // crashes. There must be some other underlying bug; not super urgent but
          // ideally should figure out why and fix it. Unfortunately we don't have
          // a repro for the crashes, only detected via production metrics.
          return NoTimestamp;

        case SelectiveHydrationLane:
        case IdleHydrationLane:
        case IdleLane:
        case OffscreenLane:
          // Anything idle priority or lower should never expire.
          return NoTimestamp;
        default:
          {
            error('Should have found matching lanes. This is a bug in React.');
          }

          return NoTimestamp;
      }
    }

    function markStarvedLanesAsExpired(root, currentTime) {
      // TODO: This gets called every time we yield. We can optimize by storing
      // the earliest expiration time on the root. Then use that to quickly bail out
      // of this function.
      var pendingLanes = root.pendingLanes;
      var suspendedLanes = root.suspendedLanes;
      var pingedLanes = root.pingedLanes;
      var expirationTimes = root.expirationTimes; // Iterate through the pending lanes and
  check if we've reached their
      // expiration time. If so, we'll assume the update is being starved and mark
      // it as expired to force it to finish.

      var lanes = pendingLanes;

      while (lanes > 0) {
        var index = pickArbitraryLaneIndex(lanes);
        var lane = 1 << index;
        var expirationTime = expirationTimes[index];

        if (expirationTime === NoTimestamp) {
          // Found a pending lane with no expiration time. If it's not suspended, or
          // if it's pinged, assume it's CPU-bound. Compute a new expiration time
          // using the current time.
          if ((lane & suspendedLanes) === NoLanes || (lane & pingedLanes) !== NoLanes) {
            // Assumes timestamps are monotonically increasing.
            expirationTimes[index] = computeExpirationTime(lane, currentTime);
          }
        } else if (expirationTime <= currentTime) {
          // This lane expired
          root.expiredLanes |= lane;
        }

        lanes &= ~lane;
      }
    } // This returns the highest priority pending lanes regardless of whether they
    // are suspended.

    function getHighestPriorityPendingLanes(root) {
      return getHighestPriorityLanes(root.pendingLanes);
    }
    function getLanesToRetrySynchronouslyOnError(root) {
      var everythingButOffscreen = root.pendingLanes & ~OffscreenLane;

      if (everythingButOffscreen !== NoLanes) {
        return everythingButOffscreen;
      }

      if (everythingButOffscreen & OffscreenLane) {
        return OffscreenLane;
      }

      return NoLanes;
    }
    function includesSyncLane(lanes) {
      return (lanes & SyncLane) !== NoLanes;
    }
    function includesNonIdleWork(lanes) {
      return (lanes & NonIdleLanes) !== NoLanes;
    }
    function includesOnlyRetries(lanes) {
      return (lanes & RetryLanes) === lanes;
    }
    function includesOnlyNonUrgentLanes(lanes) {
      var UrgentLanes = SyncLane | InputContinuousLane | DefaultLane;
      return (lanes & UrgentLanes) === NoLanes;
    }
    function includesOnlyTransitions(lanes) {
      return (lanes & TransitionLanes) === lanes;
    }
    function includesBlockingLane(root, lanes) {

      var SyncDefaultLanes = InputContinuousHydrationLane | InputContinuousLane |
  DefaultHydrationLane | DefaultLane;
      return (lanes & SyncDefaultLanes) !== NoLanes;
    }
    function includesExpiredLane(root, lanes) {
      // This is a separate check from includesBlockingLane because a lane can
      // expire after a render has already started.
      return (lanes & root.expiredLanes) !== NoLanes;
    }
    function isTransitionLane(lane) {
      return (lane & TransitionLanes) !== NoLanes;
    }
    function claimNextTransitionLane() {
      // Cycle through the lanes, assigning each new transition to the next lane.
      // In most cases, this means every transition gets its own lane, until we
      // run out of lanes and cycle back to the beginning.
      var lane = nextTransitionLane;
      nextTransitionLane <<= 1;

      if ((nextTransitionLane & TransitionLanes) === NoLanes) {
        nextTransitionLane = TransitionLane1;
      }

      return lane;
    }
    function claimNextRetryLane() {
      var lane = nextRetryLane;
      nextRetryLane <<= 1;

      if ((nextRetryLane & RetryLanes) === NoLanes) {
        nextRetryLane = RetryLane1;
      }

      return lane;
    }
    function getHighestPriorityLane(lanes) {
      return lanes & -lanes;
    }
    function pickArbitraryLane(lanes) {
      // This wrapper function gets inlined. Only exists so to communicate that it
      // doesn't matter which bit is selected; you can pick any bit without
      // affecting the algorithms where its used. Here I'm using
      // getHighestPriorityLane because it requires the fewest operations.
      return getHighestPriorityLane(lanes);
    }

    function pickArbitraryLaneIndex(lanes) {
      return 31 - clz32(lanes);
    }

    function laneToIndex(lane) {
      return pickArbitraryLaneIndex(lane);
    }

    function includesSomeLane(a, b) {
      return (a & b) !== NoLanes;
    }
    function isSubsetOfLanes(set, subset) {
```

```
    return (set & subset) === subset;
  }
  function mergeLanes(a, b) {
    return a | b;
  }
  function removeLanes(set, subset) {
    return set & ~subset;
  }
  function intersectLanes(a, b) {
    return a & b;
  } // Seems redundant, but it changes the type from a single lane (used for
  // updates) to a group of lanes (used for flushing work).

  function laneToLanes(lane) {
    return lane;
  }
  function higherPriorityLane(a, b) {
    // This works because the bit ranges decrease in priority as you go left.
    return a !== NoLane && a < b ? a : b;
  }
  function createLaneMap(initial) {
    // Intentionally pushing one by one.
    // https://v8.dev/blog/elements-kinds#avoid-creating-holes
    var laneMap = [];

    for (var i = 0; i < TotalLanes; i++) {
      laneMap.push(initial);
    }

    return laneMap;
  }
  function markRootUpdated(root, updateLane, eventTime) {
    root.pendingLanes |= updateLane; // If there are any suspended transitions, it's
possible this new update
    // could unblock them. Clear the suspended lanes so that we can try rendering
    // them again.
    //
    // TODO: We really only need to unsuspend only lanes that are in the
    // `subtreeLanes` of the updated fiber, or the update lanes of the return
    // path. This would exclude suspended updates in an unrelated sibling tree,
    // since there's no way for this update to unblock it.
    //
    // We don't do this if the incoming update is idle, because we never process
    // idle updates until after all the regular updates have finished; there's no
    // way it could unblock a transition.

    if (updateLane !== IdleLane) {
      root.suspendedLanes = NoLanes;
      root.pingedLanes = NoLanes;
    }

    var eventTimes = root.eventTimes;
    var index = laneToIndex(updateLane); // We can always overwrite an existing timestamp
because we prefer the most
    // recent event, and we assume time is monotonically increasing.

    eventTimes[index] = eventTime;
  }
  function markRootSuspended(root, suspendedLanes) {
    root.suspendedLanes |= suspendedLanes;
    root.pingedLanes &= ~suspendedLanes; // The suspended lanes are no longer CPU-bound.
Clear their expiration times.

    var expirationTimes = root.expirationTimes;
    var lanes = suspendedLanes;

    while (lanes > 0) {
      var index = pickArbitraryLaneIndex(lanes);
      var lane = 1 << index;
```

```
        expirationTimes[index] = NoTimestamp;
        lanes &= ~lane;
      }
    }
    function markRootPinged(root, pingedLanes, eventTime) {
      root.pingedLanes |= root.suspendedLanes & pingedLanes;
    }
    function markRootFinished(root, remainingLanes) {
      var noLongerPendingLanes = root.pendingLanes & ~remainingLanes;
      root.pendingLanes = remainingLanes; // Let's try everything again

      root.suspendedLanes = NoLanes;
      root.pingedLanes = NoLanes;
      root.expiredLanes &= remainingLanes;
      root.mutableReadLanes &= remainingLanes;
      root.entangledLanes &= remainingLanes;
      var entanglements = root.entanglements;
      var eventTimes = root.eventTimes;
      var expirationTimes = root.expirationTimes; // Clear the lanes that no longer have
pending work

      var lanes = noLongerPendingLanes;

      while (lanes > 0) {
        var index = pickArbitraryLaneIndex(lanes);
        var lane = 1 << index;
        entanglements[index] = NoLanes;
        eventTimes[index] = NoTimestamp;
        expirationTimes[index] = NoTimestamp;
        lanes &= ~lane;
      }
    }
    function markRootEntangled(root, entangledLanes) {
      // In addition to entangling each of the given lanes with each other, we also
      // have to consider _transitive_ entanglements. For each lane that is already
      // entangled with *any* of the given lanes, that lane is now transitively
      // entangled with *all* the given lanes.
      //
      // Translated: If C is entangled with A, then entangling A with B also
      // entangles C with B.
      //
      // If this is hard to grasp, it might help to intentionally break this
      // function and look at the tests that fail in ReactTransition-test.js. Try
      // commenting out one of the conditions below.
      var rootEntangledLanes = root.entangledLanes |= entangledLanes;
      var entanglements = root.entanglements;
      var lanes = rootEntangledLanes;

      while (lanes) {
        var index = pickArbitraryLaneIndex(lanes);
        var lane = 1 << index;

        if ( // Is this one of the newly entangled lanes?
        lane & entangledLanes | // Is this lane transitively entangled with the newly
entangled lanes?
        entanglements[index] & entangledLanes) {
          entanglements[index] |= entangledLanes;
        }

        lanes &= ~lane;
      }
    }
    function getBumpedLaneForHydration(root, renderLanes) {
      var renderLane = getHighestPriorityLane(renderLanes);
      var lane;

      switch (renderLane) {
        case InputContinuousLane:
          lane = InputContinuousHydrationLane;
          break;

        case DefaultLane:
          lane = DefaultHydrationLane;
          break;

        case TransitionLane1:
        case TransitionLane2:
        case TransitionLane3:
        case TransitionLane4:
        case TransitionLane5:
        case TransitionLane6:
        case TransitionLane7:
        case TransitionLane8:
        case TransitionLane9:
        case TransitionLane10:
        case TransitionLane11:
        case TransitionLane12:
        case TransitionLane13:
        case TransitionLane14:
        case TransitionLane15:
        case TransitionLane16:
        case RetryLane1:
        case RetryLane2:
        case RetryLane3:
        case RetryLane4:
        case RetryLane5:
          lane = TransitionHydrationLane;
          break;

        case IdleLane:
          lane = IdleHydrationLane;
          break;

        default:
          // Everything else is already either a hydration lane, or shouldn't
          // be retried at a hydration lane.
          lane = NoLane;
          break;
      } // Check if the lane we chose is suspended. If so, that indicates that we
      // already attempted and failed to hydrate at that level. Also check if we're
      // already rendering that lane, which is rare but could happen.


      if ((lane & (root.suspendedLanes | renderLanes)) !== NoLane) {
        // Give up trying to hydrate and fall back to client render.
        return NoLane;
      }

      return lane;
    }
    function addFiberToLanesMap(root, fiber, lanes) {

      if (!isDevToolsPresent) {
        return;
      }

      var pendingUpdatersLaneMap = root.pendingUpdatersLaneMap;

      while (lanes > 0) {
        var index = laneToIndex(lanes);
        var lane = 1 << index;
        var updaters = pendingUpdatersLaneMap[index];
        updaters.add(fiber);
        lanes &= ~lane;
      }
    }
    function movePendingFibersToMemoized(root, lanes) {
      if (!isDevToolsPresent) {
        return;
      }

      var pendingUpdatersLaneMap = root.pendingUpdatersLaneMap;
      var memoizedUpdaters = root.memoizedUpdaters;

      while (lanes > 0) {
        var index = laneToIndex(lanes);
        var lane = 1 << index;
        var updaters = pendingUpdatersLaneMap[index];

        if (updaters.size > 0) {
          updaters.forEach(function (fiber) {
            var alternate = fiber.alternate;

            if (alternate === null || !memoizedUpdaters.has(alternate)) {
              memoizedUpdaters.add(fiber);
            }
          });
          updaters.clear();
        }

        lanes &= ~lane;
      }
    }
    function getTransitionsForLanes(root, lanes) {
      {
        return null;
      }
    }

    var DiscreteEventPriority = SyncLane;
    var ContinuousEventPriority = InputContinuousLane;
    var DefaultEventPriority = DefaultLane;
    var IdleEventPriority = IdleLane;
    var currentUpdatePriority = NoLane;
    function getCurrentUpdatePriority() {
      return currentUpdatePriority;
    }
    function setCurrentUpdatePriority(newPriority) {
      currentUpdatePriority = newPriority;
    }
    function runWithPriority(priority, fn) {
      var previousPriority = currentUpdatePriority;

      try {
        currentUpdatePriority = priority;
        return fn();
      } finally {
        currentUpdatePriority = previousPriority;
      }
    }
    function higherEventPriority(a, b) {
      return a !== 0 && a < b ? a : b;
    }
    function lowerEventPriority(a, b) {
      return a === 0 || a > b ? a : b;
    }
    function isHigherEventPriority(a, b) {
      return a !== 0 && a < b;
    }
    function lanesToEventPriority(lanes) {
      var lane = getHighestPriorityLane(lanes);

      if (!isHigherEventPriority(DiscreteEventPriority, lane)) {
        return DiscreteEventPriority;
      }
      if (!isHigherEventPriority(ContinuousEventPriority, lane)) {
        return ContinuousEventPriority;
      }

      if (includesNonIdleWork(lane)) {
        return DefaultEventPriority;
      }

      return IdleEventPriority;
    }

    // This is imported by the event replaying implementation in React DOM. It's
    // in a separate file to break a circular dependency between the renderer and
    // the reconciler.
    function isRootDehydrated(root) {
      var currentState = root.current.memoizedState;
      return currentState.isDehydrated;
    }

    var _attemptSynchronousHydration;

    function setAttemptSynchronousHydration(fn) {
      _attemptSynchronousHydration = fn;
    }
    function attemptSynchronousHydration(fiber) {
      _attemptSynchronousHydration(fiber);
    }
    var attemptContinuousHydration;
    function setAttemptContinuousHydration(fn) {
      attemptContinuousHydration = fn;
    }
    var attemptHydrationAtCurrentPriority;
    function setAttemptHydrationAtCurrentPriority(fn) {
      attemptHydrationAtCurrentPriority = fn;
    }
    var getCurrentUpdatePriority$1;
    function setGetCurrentUpdatePriority(fn) {
      getCurrentUpdatePriority$1 = fn;
    }
    var attemptHydrationAtPriority;
    function setAttemptHydrationAtPriority(fn) {
      attemptHydrationAtPriority = fn;
    } // TODO: Upgrade this definition once we're on a newer version of Flow that
    // has this definition built-in.

    var hasScheduledReplayAttempt = false; // The queue of discrete events to be replayed.

    var queuedDiscreteEvents = []; // Indicates if any continuous event targets are non-
  null for early bailout.
    // if the last target was dehydrated.

    var queuedFocus = null;
    var queuedDrag = null;
    var queuedMouse = null; // For pointer events there can be one latest event per
  pointerId.

    var queuedPointers = new Map();
    var queuedPointerCaptures = new Map(); // We could consider replaying selectionchange
  and touchmoves too.

    var queuedExplicitHydrationTargets = [];
    var discreteReplayableEvents = ['mousedown', 'mouseup', 'touchcancel', 'touchend',
  'touchstart', 'auxclick', 'dblclick', 'pointercancel', 'pointerdown', 'pointerup',
  'dragend', 'dragstart', 'drop', 'compositionend', 'compositionstart', 'keydown',
  'keypress', 'keyup', 'input', 'textInput', // Intentionally camelCase
    'copy', 'cut', 'paste', 'click', 'change', 'contextmenu', 'reset', 'submit'];
    function isDiscreteEventThatRequiresHydration(eventType) {
      return discreteReplayableEvents.indexOf(eventType) > -1;
    }
    function createQueuedReplayableEvent(blockedOn, domEventName, eventSystemFlags,
  targetContainer, nativeEvent) {
      return {
        blockedOn: blockedOn,
        domEventName: domEventName,
        eventSystemFlags: eventSystemFlags,
        nativeEvent: nativeEvent,
        targetContainers: [targetContainer]
      };
    }

    function clearIfContinuousEvent(domEventName, nativeEvent) {
      switch (domEventName) {
        case 'focusin':
        case 'focusout':
          queuedFocus = null;
          break;

        case 'dragenter':
        case 'dragleave':
          queuedDrag = null;
          break;

        case 'mouseover':
        case 'mouseout':
          queuedMouse = null;
          break;

        case 'pointerover':
        case 'pointerout':
          {
            var pointerId = nativeEvent.pointerId;
            queuedPointers.delete(pointerId);
            break;
          }

        case 'gotpointercapture':
        case 'lostpointercapture':
          {
            var _pointerId = nativeEvent.pointerId;
            queuedPointerCaptures.delete(_pointerId);
            break;
          }
      }
    }

    function accumulateOrCreateContinuousQueuedReplayableEvent(existingQueuedEvent,
  blockedOn, domEventName, eventSystemFlags, targetContainer, nativeEvent) {
      if (existingQueuedEvent === null || existingQueuedEvent.nativeEvent !== nativeEvent)
  {
        var queuedEvent = createQueuedReplayableEvent(blockedOn, domEventName,
  eventSystemFlags, targetContainer, nativeEvent);

        if (blockedOn !== null) {
          var _fiber2 = getInstanceFromNode(blockedOn);

          if (_fiber2 !== null) {
            // Attempt to increase the priority of this target.
            attemptContinuousHydration(_fiber2);
          }
        }

        return queuedEvent;
      } // If we have already queued this exact event, then it's because
      // the different event systems have different DOM event listeners.
      // We can accumulate the flags, and the targetContainers, and
      // store a single event to be replayed.
    existingQueuedEvent.eventSystemFlags |= eventSystemFlags;
    var targetContainers = existingQueuedEvent.targetContainers;

    if (targetContainer !== null && targetContainers.indexOf(targetContainer) === -1) {
      targetContainers.push(targetContainer);
    }

    return existingQueuedEvent;
  }

  function queueIfContinuousEvent(blockedOn, domEventName, eventSystemFlags,
targetContainer, nativeEvent) {
    // These set relatedTarget to null because the replayed event will be treated as if
we
    // moved from outside the window (no target) onto the target once it hydrates.
    // Instead of mutating we could clone the event.
    switch (domEventName) {
      case 'focusin':
        {
          var focusEvent = nativeEvent;
          queuedFocus = accumulateOrCreateContinuousQueuedReplayableEvent(queuedFocus,
blockedOn, domEventName, eventSystemFlags, targetContainer, focusEvent);
          return true;
        }

      case 'dragenter':
        {
          var dragEvent = nativeEvent;
          queuedDrag = accumulateOrCreateContinuousQueuedReplayableEvent(queuedDrag,
blockedOn, domEventName, eventSystemFlags, targetContainer, dragEvent);
          return true;
        }

      case 'mouseover':
        {
          var mouseEvent = nativeEvent;
          queuedMouse = accumulateOrCreateContinuousQueuedReplayableEvent(queuedMouse,
blockedOn, domEventName, eventSystemFlags, targetContainer, mouseEvent);
          return true;
        }

      case 'pointerover':
        {
          var pointerEvent = nativeEvent;
          var pointerId = pointerEvent.pointerId;
          queuedPointers.set(pointerId,
accumulateOrCreateContinuousQueuedReplayableEvent(queuedPointers.get(pointerId) || null,
blockedOn, domEventName, eventSystemFlags, targetContainer, pointerEvent));
          return true;
        }

      case 'gotpointercapture':
        {
          var _pointerEvent = nativeEvent;
          var _pointerId2 = _pointerEvent.pointerId;
          queuedPointerCaptures.set(_pointerId2,
accumulateOrCreateContinuousQueuedReplayableEvent(queuedPointerCaptures.get(_pointerId2)
|| null, blockedOn, domEventName, eventSystemFlags, targetContainer, _pointerEvent));
          return true;
        }
    }

    return false;
  } // Check if this target is unblocked. Returns true if it's unblocked.

  function attemptExplicitHydrationTarget(queuedTarget) {
    // TODO: This function shares a lot of logic with findInstanceBlockingEvent.
    // Try to unify them. It's a bit tricky since it would require two return
```

```
      // values.
      var targetInst = getClosestInstanceFromNode(queuedTarget.target);

      if (targetInst !== null) {
        var nearestMounted = getNearestMountedFiber(targetInst);

        if (nearestMounted !== null) {
          var tag = nearestMounted.tag;

          if (tag === SuspenseComponent) {
            var instance = getSuspenseInstanceFromFiber(nearestMounted);

            if (instance !== null) {
              // We're blocked on hydrating this boundary.
              // Increase its priority.
              queuedTarget.blockedOn = instance;
              attemptHydrationAtPriority(queuedTarget.priority, function () {
                attemptHydrationAtCurrentPriority(nearestMounted);
              });
              return;
            }
          } else if (tag === HostRoot) {
            var root = nearestMounted.stateNode;

            if (isRootDehydrated(root)) {
              queuedTarget.blockedOn = getContainerFromFiber(nearestMounted); // We don't
  currently have a way to increase the priority of
              // a root other than sync.

              return;
            }
          }
        }
      }

      queuedTarget.blockedOn = null;
    }

    function queueExplicitHydrationTarget(target) {
      // TODO: This will read the priority if it's dispatched by the React
      // event system but not native events. Should read window.event.type, like
      // we do for updates (getCurrentEventPriority).
      var updatePriority = getCurrentUpdatePriority$1();
      var queuedTarget = {
        blockedOn: null,
        target: target,
        priority: updatePriority
      };
      var i = 0;

      for (; i < queuedExplicitHydrationTargets.length; i++) {
        // Stop once we hit the first target with lower priority than
        if (!isHigherEventPriority(updatePriority,
  queuedExplicitHydrationTargets[i].priority)) {
          break;
        }
      }

      queuedExplicitHydrationTargets.splice(i, 0, queuedTarget);

      if (i === 0) {
        attemptExplicitHydrationTarget(queuedTarget);
      }
    }

    function attemptReplayContinuousQueuedEvent(queuedEvent) {
      if (queuedEvent.blockedOn !== null) {
        return false;
      }
      var targetContainers = queuedEvent.targetContainers;

      while (targetContainers.length > 0) {
        var targetContainer = targetContainers[0];
        var nextBlockedOn = findInstanceBlockingEvent(queuedEvent.domEventName,
  queuedEvent.eventSystemFlags, targetContainer, queuedEvent.nativeEvent);

        if (nextBlockedOn === null) {
          {
            var nativeEvent = queuedEvent.nativeEvent;
            var nativeEventClone = new nativeEvent.constructor(nativeEvent.type,
  nativeEvent);
            setReplayingEvent(nativeEventClone);
            nativeEvent.target.dispatchEvent(nativeEventClone);
            resetReplayingEvent();
          }
        } else {
          // We're still blocked. Try again later.
          var _fiber3 = getInstanceFromNode(nextBlockedOn);

          if (_fiber3 !== null) {
            attemptContinuousHydration(_fiber3);
          }

          queuedEvent.blockedOn = nextBlockedOn;
          return false;
        } // This target container was successfully dispatched. Try the next.


        targetContainers.shift();
      }

      return true;
    }

    function attemptReplayContinuousQueuedEventInMap(queuedEvent, key, map) {
      if (attemptReplayContinuousQueuedEvent(queuedEvent)) {
        map.delete(key);
      }
    }

    function replayUnblockedEvents() {
      hasScheduledReplayAttempt = false;


      if (queuedFocus !== null && attemptReplayContinuousQueuedEvent(queuedFocus)) {
        queuedFocus = null;
      }

      if (queuedDrag !== null && attemptReplayContinuousQueuedEvent(queuedDrag)) {
        queuedDrag = null;
      }

      if (queuedMouse !== null && attemptReplayContinuousQueuedEvent(queuedMouse)) {
        queuedMouse = null;
      }

      queuedPointers.forEach(attemptReplayContinuousQueuedEventInMap);
      queuedPointerCaptures.forEach(attemptReplayContinuousQueuedEventInMap);
    }

    function scheduleCallbackIfUnblocked(queuedEvent, unblocked) {
      if (queuedEvent.blockedOn === unblocked) {
        queuedEvent.blockedOn = null;

        if (!hasScheduledReplayAttempt) {
          hasScheduledReplayAttempt = true; // Schedule a callback to attempt replaying as
  many events as are
```

```
        // now unblocked. This first might not actually be unblocked yet.
        // We could check it early to avoid scheduling an unnecessary callback.

        unstable_scheduleCallback(unstable_NormalPriority, replayUnblockedEvents);
      }
    }
  }

  function retryIfBlockedOn(unblocked) {
    // Mark anything that was blocked on this as no longer blocked
    // and eligible for a replay.
    if (queuedDiscreteEvents.length > 0) {
      scheduleCallbackIfUnblocked(queuedDiscreteEvents[0], unblocked); // This is a
exponential search for each boundary that commits. I think it's
      // worth it because we expect very few discrete events to queue up and once
      // we are actually fully unblocked it will be fast to replay them.

      for (var i = 1; i < queuedDiscreteEvents.length; i++) {
        var queuedEvent = queuedDiscreteEvents[i];

        if (queuedEvent.blockedOn === unblocked) {
          queuedEvent.blockedOn = null;
        }
      }
    }

    if (queuedFocus !== null) {
      scheduleCallbackIfUnblocked(queuedFocus, unblocked);
    }

    if (queuedDrag !== null) {
      scheduleCallbackIfUnblocked(queuedDrag, unblocked);
    }

    if (queuedMouse !== null) {
      scheduleCallbackIfUnblocked(queuedMouse, unblocked);
    }

    var unblock = function (queuedEvent) {
      return scheduleCallbackIfUnblocked(queuedEvent, unblocked);
    };

    queuedPointers.forEach(unblock);
    queuedPointerCaptures.forEach(unblock);

    for (var _i = 0; _i < queuedExplicitHydrationTargets.length; _i++) {
      var queuedTarget = queuedExplicitHydrationTargets[_i];

      if (queuedTarget.blockedOn === unblocked) {
        queuedTarget.blockedOn = null;
      }
    }

    while (queuedExplicitHydrationTargets.length > 0) {
      var nextExplicitTarget = queuedExplicitHydrationTargets[0];

      if (nextExplicitTarget.blockedOn !== null) {
        // We're still blocked.
        break;
      } else {
        attemptExplicitHydrationTarget(nextExplicitTarget);

        if (nextExplicitTarget.blockedOn === null) {
          // We're unblocked.
          queuedExplicitHydrationTargets.shift();
        }
      }
    }
  }
  var ReactCurrentBatchConfig = ReactSharedInternals.ReactCurrentBatchConfig; // TODO:
can we stop exporting these?

  var _enabled = true; // This is exported in FB builds for use by legacy FB layer infra.
  // We'd like to remove this but it's not clear if this is safe.

  function setEnabled(enabled) {
    _enabled = !!enabled;
  }
  function isEnabled() {
    return _enabled;
  }
  function createEventListenerWrapperWithPriority(targetContainer, domEventName,
eventSystemFlags) {
    var eventPriority = getEventPriority(domEventName);
    var listenerWrapper;

    switch (eventPriority) {
      case DiscreteEventPriority:
        listenerWrapper = dispatchDiscreteEvent;
        break;

      case ContinuousEventPriority:
        listenerWrapper = dispatchContinuousEvent;
        break;

      case DefaultEventPriority:
      default:
        listenerWrapper = dispatchEvent;
        break;
    }

    return listenerWrapper.bind(null, domEventName, eventSystemFlags, targetContainer);
  }

  function dispatchDiscreteEvent(domEventName, eventSystemFlags, container, nativeEvent)
{
    var previousPriority = getCurrentUpdatePriority();
    var prevTransition = ReactCurrentBatchConfig.transition;
    ReactCurrentBatchConfig.transition = null;

    try {
      setCurrentUpdatePriority(DiscreteEventPriority);
      dispatchEvent(domEventName, eventSystemFlags, container, nativeEvent);
    } finally {
      setCurrentUpdatePriority(previousPriority);
      ReactCurrentBatchConfig.transition = prevTransition;
    }
  }

  function dispatchContinuousEvent(domEventName, eventSystemFlags, container,
nativeEvent) {
    var previousPriority = getCurrentUpdatePriority();
    var prevTransition = ReactCurrentBatchConfig.transition;
    ReactCurrentBatchConfig.transition = null;

    try {
      setCurrentUpdatePriority(ContinuousEventPriority);
      dispatchEvent(domEventName, eventSystemFlags, container, nativeEvent);
    } finally {
      setCurrentUpdatePriority(previousPriority);
      ReactCurrentBatchConfig.transition = prevTransition;
    }
  }

  function dispatchEvent(domEventName, eventSystemFlags, targetContainer, nativeEvent) {
    if (!_enabled) {
      return;
    }

    {

dispatchEventWithEnableCapturePhaseSelectiveHydrationWithoutDiscreteEventReplay(domEventN
ame, eventSystemFlags, targetContainer, nativeEvent);
    }
  }

  function
dispatchEventWithEnableCapturePhaseSelectiveHydrationWithoutDiscreteEventReplay(domEventN
ame, eventSystemFlags, targetContainer, nativeEvent) {
    var blockedOn = findInstanceBlockingEvent(domEventName, eventSystemFlags,
targetContainer, nativeEvent);

    if (blockedOn === null) {
      dispatchEventForPluginEventSystem(domEventName, eventSystemFlags, nativeEvent,
return_targetInst, targetContainer);
      clearIfContinuousEvent(domEventName, nativeEvent);
      return;
    }

    if (queueIfContinuousEvent(blockedOn, domEventName, eventSystemFlags,
targetContainer, nativeEvent)) {
      nativeEvent.stopPropagation();
      return;
    } // We need to clear only if we didn't queue because
    // queueing is accumulative.


    clearIfContinuousEvent(domEventName, nativeEvent);

    if (eventSystemFlags & IS_CAPTURE_PHASE &&
isDiscreteEventThatRequiresHydration(domEventName)) {
      while (blockedOn !== null) {
        var fiber = getInstanceFromNode(blockedOn);

        if (fiber !== null) {
          attemptSynchronousHydration(fiber);
        }

        var nextBlockedOn = findInstanceBlockingEvent(domEventName, eventSystemFlags,
targetContainer, nativeEvent);

        if (nextBlockedOn === null) {
          dispatchEventForPluginEventSystem(domEventName, eventSystemFlags, nativeEvent,
return_targetInst, targetContainer);
        }

        if (nextBlockedOn === blockedOn) {
          break;
        }

        blockedOn = nextBlockedOn;
      }

      if (blockedOn !== null) {
        nativeEvent.stopPropagation();
      }

      return;
    } // This is not replayable so we'll invoke it but without a target,
    // in case the event system needs to trace it.


    dispatchEventForPluginEventSystem(domEventName, eventSystemFlags, nativeEvent, null,
targetContainer);
  }
  var return_targetInst = null; // Returns a SuspenseInstance or Container if it's
blocked.
  // The return_targetInst field above is conceptually part of the return value.

  function findInstanceBlockingEvent(domEventName, eventSystemFlags, targetContainer,
nativeEvent) {
    // TODO: Warn if _enabled is false.
    return_targetInst = null;
    var nativeEventTarget = getEventTarget(nativeEvent);
    var targetInst = getClosestInstanceFromNode(nativeEventTarget);

    if (targetInst !== null) {
      var nearestMounted = getNearestMountedFiber(targetInst);

      if (nearestMounted === null) {
        // This tree has been unmounted already. Dispatch without a target.
        targetInst = null;
      } else {
        var tag = nearestMounted.tag;

        if (tag === SuspenseComponent) {
          var instance = getSuspenseInstanceFromFiber(nearestMounted);

          if (instance !== null) {
            // Queue the event to be replayed later. Abort dispatching since we
            // don't want this event dispatched twice through the event system.
            // TODO: If this is the first discrete event in the queue. Schedule an
increased
            // priority for this boundary.
            return instance;
          } // This shouldn't happen, something went wrong but to avoid blocking
          // the whole system, dispatch the event without a target.
          // TODO: Warn.


          targetInst = null;
        } else if (tag === HostRoot) {
          var root = nearestMounted.stateNode;

          if (isRootDehydrated(root)) {
            // If this happens during a replay something went wrong and it might block
            // the whole system.
            return getContainerFromFiber(nearestMounted);
          }

          targetInst = null;
        } else if (nearestMounted !== targetInst) {
          // If we get an event (ex: img onload) before committing that
          // component's mount, ignore it for now (that is, treat it as if it was an
          // event on a non-React tree). We might also consider queueing events and
          // dispatching them after the mount.
          targetInst = null;
        }
      }
    }

    return_targetInst = targetInst; // We're not blocked on anything.

    return null;
  }
  function getEventPriority(domEventName) {
    switch (domEventName) {
      // Used by SimpleEventPlugin:
      case 'cancel':
      case 'click':
      case 'close':
      case 'contextmenu':
      case 'copy':
      case 'cut':
```

```
      case 'auxclick':
      case 'dblclick':
      case 'dragend':
      case 'dragstart':
      case 'drop':
      case 'focusin':
      case 'focusout':
      case 'input':
      case 'invalid':
      case 'keydown':
      case 'keypress':
      case 'keyup':
      case 'mousedown':
      case 'mouseup':
      case 'paste':
      case 'pause':
      case 'play':
      case 'pointercancel':
      case 'pointerdown':
      case 'pointerup':
      case 'ratechange':
      case 'reset':
      case 'resize':
      case 'seeked':
      case 'submit':
      case 'touchcancel':
      case 'touchend':
      case 'touchstart':
      case 'volumechange': // Used by polyfills:
      // eslint-disable-next-line no-fallthrough

      case 'change':
      case 'selectionchange':
      case 'textInput':
      case 'compositionstart':
      case 'compositionend':
      case 'compositionupdate': // Only enableCreateEventHandleAPI:
      // eslint-disable-next-line no-fallthrough

      case 'beforeblur':
      case 'afterblur': // Not used by React but could be by user code:
      // eslint-disable-next-line no-fallthrough

      case 'beforeinput':
      case 'blur':
      case 'fullscreenchange':
      case 'focus':
      case 'hashchange':
      case 'popstate':
      case 'select':
      case 'selectstart':
        return DiscreteEventPriority;

      case 'drag':
      case 'dragenter':
      case 'dragexit':
      case 'dragleave':
      case 'dragover':
      case 'mousemove':
      case 'mouseout':
      case 'mouseover':
      case 'pointermove':
      case 'pointerout':
      case 'pointerover':
      case 'scroll':
      case 'toggle':
      case 'touchmove':
      case 'wheel': // Not used by React but could be by user code:
      // eslint-disable-next-line no-fallthrough
      case 'mouseenter':
      case 'mouseleave':
      case 'pointerenter':
      case 'pointerleave':
        return ContinuousEventPriority;

      case 'message':
        {
          // We might be in the Scheduler callback.
          // Eventually this mechanism will be replaced by a check
          // of the current priority on the native scheduler.
          var schedulerPriority = getCurrentPriorityLevel();

          switch (schedulerPriority) {
            case ImmediatePriority:
              return DiscreteEventPriority;

            case UserBlockingPriority:
              return ContinuousEventPriority;

            case NormalPriority:
            case LowPriority:
              // TODO: Handle LowSchedulerPriority, somehow. Maybe the same lane as
hydration.
              return DefaultEventPriority;

            case IdlePriority:
              return IdleEventPriority;

            default:
              return DefaultEventPriority;
          }
        }

      default:
        return DefaultEventPriority;
    }
  }

  function addEventBubbleListener(target, eventType, listener) {
    target.addEventListener(eventType, listener, false);
    return listener;
  }
  function addEventCaptureListener(target, eventType, listener) {
    target.addEventListener(eventType, listener, true);
    return listener;
  }
  function addEventCaptureListenerWithPassiveFlag(target, eventType, listener, passive) {
    target.addEventListener(eventType, listener, {
      capture: true,
      passive: passive
    });
    return listener;
  }
  function addEventBubbleListenerWithPassiveFlag(target, eventType, listener, passive) {
    target.addEventListener(eventType, listener, {
      passive: passive
    });
    return listener;
  }

  /**
   * These variables store information about text content of a target node,
   * allowing comparison of content before and after a given event.
   *
   * Identify the node where selection currently begins, then observe
   * both its text content and its current position in the DOM. Since the
   * browser may natively replace the target node during composition, we can
   * use its position to find its replacement.
   *
   *
   */
  var root = null;
  var startText = null;
  var fallbackText = null;
  function initialize(nativeEventTarget) {
    root = nativeEventTarget;
    startText = getText();
    return true;
  }
  function reset() {
    root = null;
    startText = null;
    fallbackText = null;
  }
  function getData() {
    if (fallbackText) {
      return fallbackText;
    }

    var start;
    var startValue = startText;
    var startLength = startValue.length;
    var end;
    var endValue = getText();
    var endLength = endValue.length;

    for (start = 0; start < startLength; start++) {
      if (startValue[start] !== endValue[start]) {
        break;
      }
    }

    var minEnd = startLength - start;

    for (end = 1; end <= minEnd; end++) {
      if (startValue[startLength - end] !== endValue[endLength - end]) {
        break;
      }
    }

    var sliceTail = end > 1 ? 1 - end : undefined;
    fallbackText = endValue.slice(start, sliceTail);
    return fallbackText;
  }
  function getText() {
    if ('value' in root) {
      return root.value;
    }

    return root.textContent;
  }

  /**
   * `charCode` represents the actual "character code" and is safe to use with
   * `String.fromCharCode`. As such, only keys that correspond to printable
   * characters produce a valid `charCode`, the only exception to this is Enter.
   * The Tab-key is considered non-printable and does not have a `charCode`,
   * presumably because it does not produce a tab-character in browsers.
   *
   * @param {object} nativeEvent Native browser event.
   * @return {number} Normalized `charCode` property.
   */
  function getEventCharCode(nativeEvent) {
    var charCode;
    var keyCode = nativeEvent.keyCode;
    if ('charCode' in nativeEvent) {
      charCode = nativeEvent.charCode; // FF does not set `charCode` for the Enter-key,
check against `keyCode`.

      if (charCode === 0 && keyCode === 13) {
        charCode = 13;
      }
    } else {
      // IE8 does not implement `charCode`, but `keyCode` has the correct value.
      charCode = keyCode;
    } // IE and Edge (on Windows) and Chrome / Safari (on Windows and Linux)
    // report Enter as charCode 10 when ctrl is pressed.


    if (charCode === 10) {
      charCode = 13;
    } // Some non-printable keys are reported in `charCode`/`keyCode`, discard them.
    // Must not discard the (non-)printable Enter-key.


    if (charCode >= 32 || charCode === 13) {
      return charCode;
    }

    return 0;
  }

  function functionThatReturnsTrue() {
    return true;
  }

  function functionThatReturnsFalse() {
    return false;
  } // This is intentionally a factory so that we have different returned constructors.
  // If we had a single constructor, it would be megamorphic and engines would deopt.


  function createSyntheticEvent(Interface) {
    /**
     * Synthetic events are dispatched by event plugins, typically in response to a
     * top-level event delegation handler.
     *
     * These systems should generally use pooling to reduce the frequency of garbage
     * collection. The system should check `isPersistent` to determine whether the
     * event should be released into the pool after being dispatched. Users that
     * need a persisted event should invoke `persist`.
     *
     * Synthetic events (and subclasses) implement the DOM Level 3 Events API by
     * normalizing browser quirks. Subclasses do not necessarily have to implement a
     * DOM interface; custom application-specific events can also subclass this.
     */
    function SyntheticBaseEvent(reactName, reactEventType, targetInst, nativeEvent,
nativeEventTarget) {
      this._reactName = reactName;
      this._targetInst = targetInst;
      this.type = reactEventType;
      this.nativeEvent = nativeEvent;
      this.target = nativeEventTarget;
      this.currentTarget = null;

      for (var _propName in Interface) {
        if (!Interface.hasOwnProperty(_propName)) {
          continue;
        }

        var normalize = Interface[_propName];

        if (normalize) {
          this[_propName] = normalize(nativeEvent);
        } else {
          this[_propName] = nativeEvent[_propName];
        }
      }

      var defaultPrevented = nativeEvent.defaultPrevented != null ?
nativeEvent.defaultPrevented : nativeEvent.returnValue === false;

      if (defaultPrevented) {
        this.isDefaultPrevented = functionThatReturnsTrue;
      } else {
        this.isDefaultPrevented = functionThatReturnsFalse;
      }

      this.isPropagationStopped = functionThatReturnsFalse;
      return this;
    }

    assign(SyntheticBaseEvent.prototype, {
      preventDefault: function () {
        this.defaultPrevented = true;
        var event = this.nativeEvent;

        if (!event) {
          return;
        }

        if (event.preventDefault) {
          event.preventDefault(); // $FlowFixMe - flow is not aware of `unknown` in IE
        } else if (typeof event.returnValue !== 'unknown') {
          event.returnValue = false;
        }

        this.isDefaultPrevented = functionThatReturnsTrue;
      },
      stopPropagation: function () {
        var event = this.nativeEvent;

        if (!event) {
          return;
        }

        if (event.stopPropagation) {
          event.stopPropagation(); // $FlowFixMe - flow is not aware of `unknown` in IE
        } else if (typeof event.cancelBubble !== 'unknown') {
          // The ChangeEventPlugin registers a "propertychange" event for
          // IE. This event does not support bubbling or cancelling, and
          // any references to cancelBubble throw "Member not found".  A
          // typeof check of "unknown" circumvents this issue (and is also
          // IE specific).
          event.cancelBubble = true;
        }

        this.isPropagationStopped = functionThatReturnsTrue;
      },

      /**
       * We release all dispatched `SyntheticEvent`s after each event loop, adding
       * them back into the pool. This allows a way to hold onto a reference that
       * won't be added back into the pool.
       */
      persist: function () {// Modern event system doesn't use pooling.
      },

      /**
       * Checks if this event should be released back into the pool.
       *
       * @return {boolean} True if this should not be released, false otherwise.
       */
      isPersistent: functionThatReturnsTrue
    });
    return SyntheticBaseEvent;
  }
  /**
   * @interface Event
   * @see http://www.w3.org/TR/DOM-Level-3-Events/
   */


  var EventInterface = {
    eventPhase: 0,
    bubbles: 0,
    cancelable: 0,
    timeStamp: function (event) {
      return event.timeStamp || Date.now();
    },
    defaultPrevented: 0,
    isTrusted: 0
  };
  var SyntheticEvent = createSyntheticEvent(EventInterface);

  var UIEventInterface = assign({}, EventInterface, {
    view: 0,
    detail: 0
  });

  var SyntheticUIEvent = createSyntheticEvent(UIEventInterface);
  var lastMovementX;
  var lastMovementY;
  var lastMouseEvent;

  function updateMouseMovementPolyfillState(event) {
    if (event !== lastMouseEvent) {
      if (lastMouseEvent && event.type === 'mousemove') {
        lastMovementX = event.screenX - lastMouseEvent.screenX;
        lastMovementY = event.screenY - lastMouseEvent.screenY;
      } else {
        lastMovementX = 0;
        lastMovementY = 0;
      }

      lastMouseEvent = event;
    }
  }
  /**
   * @interface MouseEvent
   * @see http://www.w3.org/TR/DOM-Level-3-Events/
   */


  var MouseEventInterface = assign({}, UIEventInterface, {
    screenX: 0,
    screenY: 0,
    clientX: 0,
    clientY: 0,
    pageX: 0,
    pageY: 0,
    ctrlKey: 0,
    shiftKey: 0,
    altKey: 0,
    metaKey: 0,
    getModifierState: getEventModifierState,
    button: 0,
    buttons: 0,
    relatedTarget: function (event) {
      if (event.relatedTarget === undefined) return event.fromElement ===
event.srcElement ? event.toElement : event.fromElement;
      return event.relatedTarget;
    },
    movementX: function (event) {
      if ('movementX' in event) {
        return event.movementX;
      }

      updateMouseMovementPolyfillState(event);
      return lastMovementX;
    },
    movementY: function (event) {
      if ('movementY' in event) {
        return event.movementY;
      } // Don't need to call updateMouseMovementPolyfillState() here
      // because it's guaranteed to have already run when movementX
      // was copied.


      return lastMovementY;
    }
  });

  var SyntheticMouseEvent = createSyntheticEvent(MouseEventInterface);
  /**
   * @interface DragEvent
   * @see http://www.w3.org/TR/DOM-Level-3-Events/
   */

  var DragEventInterface = assign({}, MouseEventInterface, {
    dataTransfer: 0
  });

  var SyntheticDragEvent = createSyntheticEvent(DragEventInterface);
  /**
   * @interface FocusEvent
   * @see http://www.w3.org/TR/DOM-Level-3-Events/
   */

  var FocusEventInterface = assign({}, UIEventInterface, {
    relatedTarget: 0
  });

  var SyntheticFocusEvent = createSyntheticEvent(FocusEventInterface);
  /**
   * @interface Event
   * @see http://www.w3.org/TR/css3-animations/#AnimationEvent-interface
   * @see https://developer.mozilla.org/en-US/docs/Web/API/AnimationEvent
   */

  var AnimationEventInterface = assign({}, EventInterface, {
    animationName: 0,
    elapsedTime: 0,
    pseudoElement: 0
  });

  var SyntheticAnimationEvent = createSyntheticEvent(AnimationEventInterface);
  /**
   * @interface Event
   * @see http://www.w3.org/TR/clipboard-apis/
   */

  var ClipboardEventInterface = assign({}, EventInterface, {
    clipboardData: function (event) {
      return 'clipboardData' in event ? event.clipboardData : window.clipboardData;
    }
  });

  var SyntheticClipboardEvent = createSyntheticEvent(ClipboardEventInterface);
  /**
   * @interface Event
   * @see http://www.w3.org/TR/DOM-Level-3-Events/#events-compositionevents
   */

  var CompositionEventInterface = assign({}, EventInterface, {
    data: 0
  });

  var SyntheticCompositionEvent = createSyntheticEvent(CompositionEventInterface);
  /**
   * @interface Event
   * @see http://www.w3.org/TR/2013/WD-DOM-Level-3-Events-20131105
   *      /#events-inputevents
   */
  // Happens to share the same list for now.

  var SyntheticInputEvent = SyntheticCompositionEvent;
  /**
   * Normalization of deprecated HTML5 `key` values
   * @see https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent#Key_names
   */

  var normalizeKey = {
    Esc: 'Escape',
    Spacebar: ' ',
    Left: 'ArrowLeft',
    Up: 'ArrowUp',
    Right: 'ArrowRight',
    Down: 'ArrowDown',
    Del: 'Delete',
    Win: 'OS',
    Menu: 'ContextMenu',
    Apps: 'ContextMenu',
    Scroll: 'ScrollLock',
    MozPrintableKey: 'Unidentified'
  };
  /**
   * Translation from legacy `keyCode` to HTML5 `key`
   * Only special keys supported, all others depend on keyboard layout or browser
   * @see https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent#Key_names
   */

  var translateToKey = {
    '8': 'Backspace',
    '9': 'Tab',
    '12': 'Clear',
    '13': 'Enter',
    '16': 'Shift',
    '17': 'Control',
    '18': 'Alt',
    '19': 'Pause',
    '20': 'CapsLock',
    '27': 'Escape',
    '32': ' ',
    '33': 'PageUp',
    '34': 'PageDown',
    '35': 'End',
    '36': 'Home',
    '37': 'ArrowLeft',
    '38': 'ArrowUp',
    '39': 'ArrowRight',
    '40': 'ArrowDown',
    '45': 'Insert',
    '46': 'Delete',
    '112': 'F1',
    '113': 'F2',
    '114': 'F3',
    '115': 'F4',
    '116': 'F5',
    '117': 'F6',
    '118': 'F7',
    '119': 'F8',
    '120': 'F9',
    '121': 'F10',
    '122': 'F11',
    '123': 'F12',
    '144': 'NumLock',
    '145': 'ScrollLock',
    '224': 'Meta'
  };
  /**
   * @param {object} nativeEvent Native browser event.
   * @return {string} Normalized `key` property.
   */

  function getEventKey(nativeEvent) {
    if (nativeEvent.key) {
      // Normalize inconsistent values reported by browsers due to
      // implementations of a working draft specification.
      // FireFox implements `key` but returns `MozPrintableKey` for all
      // printable characters (normalized to `Unidentified`), ignore it.
      var key = normalizeKey[nativeEvent.key] || nativeEvent.key;

      if (key !== 'Unidentified') {
        return key;
      }
    } // Browser does not implement `key`, polyfill as much of it as we can.


    if (nativeEvent.type === 'keypress') {
      var charCode = getEventCharCode(nativeEvent); // The enter-key is technically both
printable and non-printable and can
      // thus be captured by `keypress`, no other non-printable key should.

      return charCode === 13 ? 'Enter' : String.fromCharCode(charCode);
    }

    if (nativeEvent.type === 'keydown' || nativeEvent.type === 'keyup') {
      // While user keyboard layout determines the actual meaning of each
      // `keyCode` value, almost all function keys have a universal value.
      return translateToKey[nativeEvent.keyCode] || 'Unidentified';
    }

    return '';
  }
  /**
   * Translation from modifier key to the associated property in the event.
   * @see http://www.w3.org/TR/DOM-Level-3-Events/#keys-Modifiers
   */


  var modifierKeyToProp = {
    Alt: 'altKey',
    Control: 'ctrlKey',
    Meta: 'metaKey',
    Shift: 'shiftKey'
  }; // Older browsers (Safari <= 10, iOS Safari <= 10.2) do not support
  // getModifierState. If getModifierState is not supported, we map it to a set of
  // modifier keys exposed by the event. In this case, Lock-keys are not supported.

  function modifierStateGetter(keyArg) {
    var syntheticEvent = this;
    var nativeEvent = syntheticEvent.nativeEvent;

    if (nativeEvent.getModifierState) {
      return nativeEvent.getModifierState(keyArg);
    }

    var keyProp = modifierKeyToProp[keyArg];
    return keyProp ? !!nativeEvent[keyProp] : false;
  }

  function getEventModifierState(nativeEvent) {
    return modifierStateGetter;
  }
  /**
   * @interface KeyboardEvent
   * @see http://www.w3.org/TR/DOM-Level-3-Events/
   */


  var KeyboardEventInterface = assign({}, UIEventInterface, {
    key: getEventKey,
    code: 0,
    location: 0,
    ctrlKey: 0,
    shiftKey: 0,
    altKey: 0,
    metaKey: 0,
    repeat: 0,
    locale: 0,
    getModifierState: getEventModifierState,
    // Legacy Interface
    charCode: function (event) {
      // `charCode` is the result of a KeyPress event and represents the value of
      // the actual printable character.
      // KeyPress is deprecated, but its replacement is not yet final and not
      // implemented in any major browser. Only KeyPress has charCode.
      if (event.type === 'keypress') {
        return getEventCharCode(event);
      }

      return 0;
    },
    keyCode: function (event) {
      // `keyCode` is the result of a KeyDown/Up event and represents the value of
      // physical keyboard key.
      // The actual meaning of the value depends on the users' keyboard layout
      // which cannot be detected. Assuming that it is a US keyboard layout
      // provides a surprisingly accurate mapping for US and European users.
      // Due to this, it is left to the user to implement at this time.
      if (event.type === 'keydown' || event.type === 'keyup') {
        return event.keyCode;
      }

      return 0;
    },
    which: function (event) {
      // `which` is an alias for either `keyCode` or `charCode` depending on the
      // type of the event.
      if (event.type === 'keypress') {
        return getEventCharCode(event);
      }

      if (event.type === 'keydown' || event.type === 'keyup') {
        return event.keyCode;
      }

      return 0;
    }
  });

  var SyntheticKeyboardEvent = createSyntheticEvent(KeyboardEventInterface);
  /**
   * @interface PointerEvent
   * @see http://www.w3.org/TR/pointerevents/
   */
```

```
  var PointerEventInterface = assign({}, MouseEventInterface, {
    pointerId: 0,
    width: 0,
    height: 0,
    pressure: 0,
    tangentialPressure: 0,
    tiltX: 0,
    tiltY: 0,
    twist: 0,
    pointerType: 0,
    isPrimary: 0
  });

  var SyntheticPointerEvent = createSyntheticEvent(PointerEventInterface);
  /**
   * @interface TouchEvent
   * @see http://www.w3.org/TR/touch-events/
   */

  var TouchEventInterface = assign({}, UIEventInterface, {
    touches: 0,
    targetTouches: 0,
    changedTouches: 0,
    altKey: 0,
    metaKey: 0,
    ctrlKey: 0,
    shiftKey: 0,
    getModifierState: getEventModifierState
  });

  var SyntheticTouchEvent = createSyntheticEvent(TouchEventInterface);
  /**
   * @interface Event
   * @see http://www.w3.org/TR/2009/WD-css3-transitions-20090320/#transition-events-
   * @see https://developer.mozilla.org/en-US/docs/Web/API/TransitionEvent
   */

  var TransitionEventInterface = assign({}, EventInterface, {
    propertyName: 0,
    elapsedTime: 0,
    pseudoElement: 0
  });

  var SyntheticTransitionEvent = createSyntheticEvent(TransitionEventInterface);
  /**
   * @interface WheelEvent
   * @see http://www.w3.org/TR/DOM-Level-3-Events/
   */

  var WheelEventInterface = assign({}, MouseEventInterface, {
    deltaX: function (event) {
      return 'deltaX' in event ? event.deltaX : // Fallback to `wheelDeltaX` for Webkit
and normalize (right is positive).
      'wheelDeltaX' in event ? -event.wheelDeltaX : 0;
    },
    deltaY: function (event) {
      return 'deltaY' in event ? event.deltaY : // Fallback to `wheelDeltaY` for Webkit
and normalize (down is positive).
      'wheelDeltaY' in event ? -event.wheelDeltaY : // Fallback to `wheelDelta` for IE<9
and normalize (down is positive).
      'wheelDelta' in event ? -event.wheelDelta : 0;
    },
    deltaZ: 0,
    // Browsers without "deltaMode" is reporting in raw wheel delta where one
    // notch on the scroll is always +/- 120, roughly equivalent to pixels.
    // A good approximation of DOM_DELTA_LINE (1) is 5% of viewport size or
    // ~40 pixels, for DOM_DELTA_SCREEN (2) it is 87.5% of viewport size.
    deltaMode: 0
  });
  var SyntheticWheelEvent = createSyntheticEvent(WheelEventInterface);

  var END_KEYCODES = [9, 13, 27, 32]; // Tab, Return, Esc, Space

  var START_KEYCODE = 229;
  var canUseCompositionEvent = canUseDOM && 'CompositionEvent' in window;
  var documentMode = null;

  if (canUseDOM && 'documentMode' in document) {
    documentMode = document.documentMode;
  } // Webkit offers a very useful `textInput` event that can be used to
  // directly represent `beforeInput`. The IE `textinput` event is not as
  // useful, so we don't use it.


  var canUseTextInputEvent = canUseDOM && 'TextEvent' in window && !documentMode; // In
IE9+, we have access to composition events, but the data supplied
  // by the native compositionend event may be incorrect. Japanese ideographic
  // spaces, for instance (\u3000) are not recorded correctly.

  var useFallbackCompositionData = canUseDOM && (!canUseCompositionEvent || documentMode
&& documentMode > 8 && documentMode <= 11);
  var SPACEBAR_CODE = 32;
  var SPACEBAR_CHAR = String.fromCharCode(SPACEBAR_CODE);

  function registerEvents() {
    registerTwoPhaseEvent('onBeforeInput', ['compositionend', 'keypress', 'textInput',
'paste']);
    registerTwoPhaseEvent('onCompositionEnd', ['compositionend', 'focusout', 'keydown',
'keypress', 'keyup', 'mousedown']);
    registerTwoPhaseEvent('onCompositionStart', ['compositionstart', 'focusout',
'keydown', 'keypress', 'keyup', 'mousedown']);
    registerTwoPhaseEvent('onCompositionUpdate', ['compositionupdate', 'focusout',
'keydown', 'keypress', 'keyup', 'mousedown']);
  } // Track whether we've ever handled a keypress on the space key.


  var hasSpaceKeypress = false;
  /**
   * Return whether a native keypress event is assumed to be a command.
   * This is required because Firefox fires `keypress` events for key commands
   * (cut, copy, select-all, etc.) even though no character is inserted.
   */

  function isKeypressCommand(nativeEvent) {
    return (nativeEvent.ctrlKey || nativeEvent.altKey || nativeEvent.metaKey) && //
ctrlKey && altKey is equivalent to AltGr, and is not a command.
    !(nativeEvent.ctrlKey && nativeEvent.altKey);
  }
  /**
   * Translate native top level events into event types.
   */


  function getCompositionEventType(domEventName) {
    switch (domEventName) {
      case 'compositionstart':
        return 'onCompositionStart';

      case 'compositionend':
        return 'onCompositionEnd';

      case 'compositionupdate':
        return 'onCompositionUpdate';
    }
  }
  /**
   * Does our fallback best-guess model think this event signifies that
```

```
     * composition has begun?
     */


    function isFallbackCompositionStart(domEventName, nativeEvent) {
      return domEventName === 'keydown' && nativeEvent.keyCode === START_KEYCODE;
    }
    /**
     * Does our fallback mode think that this event is the end of composition?
     */


    function isFallbackCompositionEnd(domEventName, nativeEvent) {
      switch (domEventName) {
        case 'keyup':
          // Command keys insert or clear IME input.
          return END_KEYCODES.indexOf(nativeEvent.keyCode) !== -1;

        case 'keydown':
          // Expect IME keyCode on each keydown. If we get any other
          // code we must have exited earlier.
          return nativeEvent.keyCode !== START_KEYCODE;

        case 'keypress':
        case 'mousedown':
        case 'focusout':
          // Events are not possible without cancelling IME.
          return true;

        default:
          return false;
      }
    }
    /**
     * Google Input Tools provides composition data via a CustomEvent,
     * with the `data` property populated in the `detail` object. If this
     * is available on the event object, use it. If not, this is a plain
     * composition event and we have nothing special to extract.
     *
     * @param {object} nativeEvent
     * @return {?string}
     */


    function getDataFromCustomEvent(nativeEvent) {
      var detail = nativeEvent.detail;

      if (typeof detail === 'object' && 'data' in detail) {
        return detail.data;
      }

      return null;
    }
    /**
     * Check if a composition event was triggered by Korean IME.
     * Our fallback mode does not work well with IE's Korean IME,
     * so just use native composition events when Korean IME is used.
     * Although CompositionEvent.locale property is deprecated,
     * it is available in IE, where our fallback mode is enabled.
     *
     * @param {object} nativeEvent
     * @return {boolean}
     */


    function isUsingKoreanIME(nativeEvent) {
      return nativeEvent.locale === 'ko';
    } // Track the current IME composition status, if any.
```

```
  var isComposing = false;
  /**
   * @return {?object} A SyntheticCompositionEvent.
   */

  function extractCompositionEvent(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget) {
    var eventType;
    var fallbackData;

    if (canUseCompositionEvent) {
      eventType = getCompositionEventType(domEventName);
    } else if (!isComposing) {
      if (isFallbackCompositionStart(domEventName, nativeEvent)) {
        eventType = 'onCompositionStart';
      }
    } else if (isFallbackCompositionEnd(domEventName, nativeEvent)) {
      eventType = 'onCompositionEnd';
    }

    if (!eventType) {
      return null;
    }

    if (useFallbackCompositionData && !isUsingKoreanIME(nativeEvent)) {
      // The current composition is stored statically and must not be
      // overwritten while composition continues.
      if (!isComposing && eventType === 'onCompositionStart') {
        isComposing = initialize(nativeEventTarget);
      } else if (eventType === 'onCompositionEnd') {
        if (isComposing) {
          fallbackData = getData();
        }
      }
    }

    var listeners = accumulateTwoPhaseListeners(targetInst, eventType);

    if (listeners.length > 0) {
      var event = new SyntheticCompositionEvent(eventType, domEventName, null,
nativeEvent, nativeEventTarget);
      dispatchQueue.push({
        event: event,
        listeners: listeners
      });

      if (fallbackData) {
        // Inject data generated from fallback path into the synthetic event.
        // This matches the property of native CompositionEventInterface.
        event.data = fallbackData;
      } else {
        var customData = getDataFromCustomEvent(nativeEvent);

        if (customData !== null) {
          event.data = customData;
        }
      }
    }
  }

  function getNativeBeforeInputChars(domEventName, nativeEvent) {
    switch (domEventName) {
      case 'compositionend':
        return getDataFromCustomEvent(nativeEvent);

      case 'keypress':
        /**
         * If native `textInput` events are available, our goal is to make
         * use of them. However, there is a special case: the spacebar key.
         * In Webkit, preventing default on a spacebar `textInput` event
         * cancels character insertion, but it *also* causes the browser
         * to fall back to its default spacebar behavior of scrolling the
         * page.
         *
         * Tracking at:
         * https://code.google.com/p/chromium/issues/detail?id=355103
         *
         * To avoid this issue, use the keypress event as if no `textInput`
         * event is available.
         */
        var which = nativeEvent.which;

        if (which !== SPACEBAR_CODE) {
          return null;
        }

        hasSpaceKeypress = true;
        return SPACEBAR_CHAR;

      case 'textInput':
        // Record the characters to be added to the DOM.
        var chars = nativeEvent.data; // If it's a spacebar character, assume that we
have already handled
        // it at the keypress level and bail immediately. Android Chrome
        // doesn't give us keycodes, so we need to ignore it.

        if (chars === SPACEBAR_CHAR && hasSpaceKeypress) {
          return null;
        }

        return chars;

      default:
        // For other native event types, do nothing.
        return null;
    }
  }
  /**
   * For browsers that do not provide the `textInput` event, extract the
   * appropriate string to use for SyntheticInputEvent.
   */


  function getFallbackBeforeInputChars(domEventName, nativeEvent) {
    // If we are currently composing (IME) and using a fallback to do so,
    // try to extract the composed characters from the fallback object.
    // If composition event is available, we extract a string only at
    // compositionevent, otherwise extract it at fallback events.
    if (isComposing) {
      if (domEventName === 'compositionend' || !canUseCompositionEvent &&
isFallbackCompositionEnd(domEventName, nativeEvent)) {
        var chars = getData();
        reset();
        isComposing = false;
        return chars;
      }

      return null;
    }

    switch (domEventName) {
      case 'paste':
        // If a paste event occurs after a keypress, throw out the input
        // chars. Paste events should not lead to BeforeInput events.
        return null;

      case 'keypress':
        /**
         * As of v27, Firefox may fire keypress events even when no character
         * will be inserted. A few possibilities:
         *
         * - `which` is `0`. Arrow keys, Esc key, etc.
         *
         * - `which` is the pressed key code, but no char is available.
         *   Ex: 'AltGr + d` in Polish. There is no modified character for
         *   this key combination and no character is inserted into the
         *   document, but FF fires the keypress for char code `100` anyway.
         *   No `input` event will occur.
         *
         * - `which` is the pressed key code, but a command combination is
         *   being used. Ex: `Cmd+C`. No character is inserted, and no
         *   `input` event will occur.
         */
        if (!isKeypressCommand(nativeEvent)) {
          // IE fires the `keypress` event when a user types an emoji via
          // Touch keyboard of Windows.  In such a case, the `char` property
          // holds an emoji character like `\uD83D\uDE0A`.  Because its length
          // is 2, the property `which` does not represent an emoji correctly.
          // In such a case, we directly return the `char` property instead of
          // using `which`.
          if (nativeEvent.char && nativeEvent.char.length > 1) {
            return nativeEvent.char;
          } else if (nativeEvent.which) {
            return String.fromCharCode(nativeEvent.which);
          }
        }

        return null;

      case 'compositionend':
        return useFallbackCompositionData && !isUsingKoreanIME(nativeEvent) ? null :
nativeEvent.data;

      default:
        return null;
    }
  }
  /**
   * Extract a SyntheticInputEvent for `beforeInput`, based on either native
   * `textInput` or fallback behavior.
   *
   * @return {?object} A SyntheticInputEvent.
   */


  function extractBeforeInputEvent(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget) {
    var chars;

    if (canUseTextInputEvent) {
      chars = getNativeBeforeInputChars(domEventName, nativeEvent);
    } else {
      chars = getFallbackBeforeInputChars(domEventName, nativeEvent);
    } // If no characters are being inserted, no BeforeInput event should
    // be fired.


    if (!chars) {
      return null;
    }

    var listeners = accumulateTwoPhaseListeners(targetInst, 'onBeforeInput');

    if (listeners.length > 0) {
      var event = new SyntheticInputEvent('onBeforeInput', 'beforeinput', null,
nativeEvent, nativeEventTarget);
```

```
      dispatchQueue.push({
        event: event,
        listeners: listeners
      });
      event.data = chars;
    }
  }
  /**
   * Create an `onBeforeInput` event to match
   * http://www.w3.org/TR/2013/WD-DOM-Level-3-Events-20131105/#events-inputevents.
   *
   * This event plugin is based on the native `textInput` event
   * available in Chrome, Safari, Opera, and IE. This event fires after
   * `onKeyPress` and `onCompositionEnd`, but before `onInput`.
   *
   * `beforeInput` is spec'd but not implemented in any browsers, and
   * the `input` event does not provide any useful information about what has
   * actually been added, contrary to the spec. Thus, `textInput` is the best
   * available event to identify the characters that have actually been inserted
   * into the target node.
   *
   * This plugin is also responsible for emitting `composition` events, thus
   * allowing us to share composition fallback code for both `beforeInput` and
   * `composition` event types.
   */


  function extractEvents(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget, eventSystemFlags, targetContainer) {
    extractCompositionEvent(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget);
    extractBeforeInputEvent(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget);
  }

  /**
   * @see http://www.whatwg.org/specs/web-apps/current-work/multipage/the-input-
element.html#input-type-attr-summary
   */
  var supportedInputTypes = {
    color: true,
    date: true,
    datetime: true,
    'datetime-local': true,
    email: true,
    month: true,
    number: true,
    password: true,
    range: true,
    search: true,
    tel: true,
    text: true,
    time: true,
    url: true,
    week: true
  };

  function isTextInputElement(elem) {
    var nodeName = elem && elem.nodeName && elem.nodeName.toLowerCase();

    if (nodeName === 'input') {
      return !!supportedInputTypes[elem.type];
    }

    if (nodeName === 'textarea') {
      return true;
    }

    return false;
  }

  /**
   * Checks if an event is supported in the current execution environment.
   *
   * NOTE: This will not work correctly for non-generic events such as `change`,
   * `reset`, `load`, `error`, and `select`.
   *
   * Borrows from Modernizr.
   *
   * @param {string} eventNameSuffix Event name, e.g. "click".
   * @return {boolean} True if the event is supported.
   * @internal
   * @license Modernizr 3.0.0pre (Custom Build) | MIT
   */

  function isEventSupported(eventNameSuffix) {
    if (!canUseDOM) {
      return false;
    }

    var eventName = 'on' + eventNameSuffix;
    var isSupported = (eventName in document);

    if (!isSupported) {
      var element = document.createElement('div');
      element.setAttribute(eventName, 'return;');
      isSupported = typeof element[eventName] === 'function';
    }

    return isSupported;
  }

  function registerEvents$1() {
    registerTwoPhaseEvent('onChange', ['change', 'click', 'focusin', 'focusout', 'input',
'keydown', 'keyup', 'selectionchange']);
  }

  function createAndAccumulateChangeEvent(dispatchQueue, inst, nativeEvent, target) {
    // Flag this event loop as needing state restore.
    enqueueStateRestore(target);
    var listeners = accumulateTwoPhaseListeners(inst, 'onChange');

    if (listeners.length > 0) {
      var event = new SyntheticEvent('onChange', 'change', null, nativeEvent, target);
      dispatchQueue.push({
        event: event,
        listeners: listeners
      });
    }
  }
  /**
   * For IE shims
   */


  var activeElement = null;
  var activeElementInst = null;
  /**
   * SECTION: handle `change` event
   */

  function shouldUseChangeEvent(elem) {
    var nodeName = elem.nodeName && elem.nodeName.toLowerCase();
    return nodeName === 'select' || nodeName === 'input' && elem.type === 'file';
  }

  function manualDispatchChangeEvent(nativeEvent) {
    var dispatchQueue = [];
    createAndAccumulateChangeEvent(dispatchQueue, activeElementInst, nativeEvent,
getEventTarget(nativeEvent)); // If change and propertychange bubbled, we'd just bind to
it like all the
    // other events and have it go through ReactBrowserEventEmitter. Since it
    // doesn't, we manually listen for the events and so we have to enqueue and
    // process the abstract event manually.
    //
    // Batching is necessary here in order to ensure that all event handlers run
    // before the next rerender (including event handlers attached to ancestor
    // elements instead of directly on the input). Without this, controlled
    // components don't work properly in conjunction with event bubbling because
    // the component is rerendered and the value reverted before all the event
    // handlers can run. See https://github.com/facebook/react/issues/708.

    batchedUpdates(runEventInBatch, dispatchQueue);
  }

  function runEventInBatch(dispatchQueue) {
    processDispatchQueue(dispatchQueue, 0);
  }

  function getInstIfValueChanged(targetInst) {
    var targetNode = getNodeFromInstance(targetInst);

    if (updateValueIfChanged(targetNode)) {
      return targetInst;
    }
  }

  function getTargetInstForChangeEvent(domEventName, targetInst) {
    if (domEventName === 'change') {
      return targetInst;
    }
  }
  /**
   * SECTION: handle `input` event
   */


  var isInputEventSupported = false;

  if (canUseDOM) {
    // IE9 claims to support the input event but fails to trigger it when
    // deleting text, so we ignore its input events.
    isInputEventSupported = isEventSupported('input') && (!document.documentMode ||
document.documentMode > 9);
  }
  /**
   * (For IE <=9) Starts tracking propertychange events on the passed-in element
   * and override the value property so that we can distinguish user events from
   * value changes in JS.
   */


  function startWatchingForValueChange(target, targetInst) {
    activeElement = target;
    activeElementInst = targetInst;
    activeElement.attachEvent('onpropertychange', handlePropertyChange);
  }
  /**
   * (For IE <=9) Removes the event listeners from the currently-tracked element,
   * if any exists.
   */


  function stopWatchingForValueChange() {
    if (!activeElement) {
      return;
    }
```

```
      activeElement.detachEvent('onpropertychange', handlePropertyChange);
      activeElement = null;
      activeElementInst = null;
    }
    /**
     * (For IE <=9) Handles a propertychange event, sending a `change` event if
     * the value of the active element has changed.
     */


    function handlePropertyChange(nativeEvent) {
      if (nativeEvent.propertyName !== 'value') {
        return;
      }

      if (getInstIfValueChanged(activeElementInst)) {
        manualDispatchChangeEvent(nativeEvent);
      }
    }

    function handleEventsForInputEventPolyfill(domEventName, target, targetInst) {
      if (domEventName === 'focusin') {
        // In IE9, propertychange fires for most input events but is buggy and
        // doesn't fire when text is deleted, but conveniently, selectionchange
        // appears to fire in all of the remaining cases so we catch those and
        // forward the event if the value has changed
        // In either case, we don't want to call the event handler if the value
        // is changed from JS so we redefine a setter for `.value` that updates
        // our activeElementValue variable, allowing us to ignore those changes
        //
        // stopWatching() should be a noop here but we call it just in case we
        // missed a blur event somehow.
        stopWatchingForValueChange();
        startWatchingForValueChange(target, targetInst);
      } else if (domEventName === 'focusout') {
        stopWatchingForValueChange();
      }
    } // For IE8 and IE9.


    function getTargetInstForInputEventPolyfill(domEventName, targetInst) {
      if (domEventName === 'selectionchange' || domEventName === 'keyup' || domEventName
  === 'keydown') {
        // On the selectionchange event, the target is just document which isn't
        // helpful for us so just check activeElement instead.
        //
        // 99% of the time, keydown and keyup aren't necessary. IE8 fails to fire
        // propertychange on the first input event after setting `value` from a
        // script and fires only keydown, keypress, keyup. Catching keyup usually
        // gets it and catching keydown lets us fire an event for the first
        // keystroke if user does a key repeat (it'll be a little delayed: right
        // before the second keystroke). Other input methods (e.g., paste) seem to
        // fire selectionchange normally.
        return getInstIfValueChanged(activeElementInst);
      }
    }
    /**
     * SECTION: handle `click` event
     */


    function shouldUseClickEvent(elem) {
      // Use the `click` event to detect changes to checkbox and radio inputs.
      // This approach works across all browsers, whereas `change` does not fire
      // until `blur` in IE8.
      var nodeName = elem.nodeName;
      return nodeName && nodeName.toLowerCase() === 'input' && (elem.type === 'checkbox' ||
  elem.type === 'radio');
```

```
    }

    function getTargetInstForClickEvent(domEventName, targetInst) {
      if (domEventName === 'click') {
        return getInstIfValueChanged(targetInst);
      }
    }

    function getTargetInstForInputOrChangeEvent(domEventName, targetInst) {
      if (domEventName === 'input' || domEventName === 'change') {
        return getInstIfValueChanged(targetInst);
      }
    }

    function handleControlledInputBlur(node) {
      var state = node._wrapperState;

      if (!state || !state.controlled || node.type !== 'number') {
        return;
      }

      {
        // If controlled, assign the value attribute to the current value on blur
        setDefaultValue(node, 'number', node.value);
      }
    }
    /**
     * This plugin creates an `onChange` event that normalizes change events
     * across form elements. This event fires at a time when it's possible to
     * change the element's value without seeing a flicker.
     *
     * Supported elements are:
     * - input (see `isTextInputElement`)
     * - textarea
     * - select
     */


    function extractEvents$1(dispatchQueue, domEventName, targetInst, nativeEvent,
  nativeEventTarget, eventSystemFlags, targetContainer) {
      var targetNode = targetInst ? getNodeFromInstance(targetInst) : window;
      var getTargetInstFunc, handleEventFunc;

      if (shouldUseChangeEvent(targetNode)) {
        getTargetInstFunc = getTargetInstForChangeEvent;
      } else if (isTextInputElement(targetNode)) {
        if (isInputEventSupported) {
          getTargetInstFunc = getTargetInstForInputOrChangeEvent;
        } else {
          getTargetInstFunc = getTargetInstForInputEventPolyfill;
          handleEventFunc = handleEventsForInputEventPolyfill;
        }
      } else if (shouldUseClickEvent(targetNode)) {
        getTargetInstFunc = getTargetInstForClickEvent;
      }

      if (getTargetInstFunc) {
        var inst = getTargetInstFunc(domEventName, targetInst);

        if (inst) {
          createAndAccumulateChangeEvent(dispatchQueue, inst, nativeEvent,
  nativeEventTarget);
          return;
        }
      }

      if (handleEventFunc) {
        handleEventFunc(domEventName, targetNode, targetInst);
      } // When blurring, set the value attribute for number inputs
    if (domEventName === 'focusout') {
      handleControlledInputBlur(targetNode);
    }
  }

  function registerEvents$2() {
    registerDirectEvent('onMouseEnter', ['mouseout', 'mouseover']);
    registerDirectEvent('onMouseLeave', ['mouseout', 'mouseover']);
    registerDirectEvent('onPointerEnter', ['pointerout', 'pointerover']);
    registerDirectEvent('onPointerLeave', ['pointerout', 'pointerover']);
  }
  /**
   * For almost every interaction we care about, there will be both a top-level
   * `mouseover` and `mouseout` event that occurs. Only use `mouseout` so that
   * we do not extract duplicate events. However, moving the mouse into the
   * browser from outside will not fire a `mouseout` event. In this case, we use
   * the `mouseover` top-level event.
   */


  function extractEvents$2(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget, eventSystemFlags, targetContainer) {
    var isOverEvent = domEventName === 'mouseover' || domEventName === 'pointerover';
    var isOutEvent = domEventName === 'mouseout' || domEventName === 'pointerout';

    if (isOverEvent && !isReplayingEvent(nativeEvent)) {
      // If this is an over event with a target, we might have already dispatched
      // the event in the out event of the other target. If this is replayed,
      // then it's because we couldn't dispatch against this target previously
      // so we have to do it now instead.
      var related = nativeEvent.relatedTarget || nativeEvent.fromElement;

      if (related) {
        // If the related node is managed by React, we can assume that we have
        // already dispatched the corresponding events during its mouseout.
        if (getClosestInstanceFromNode(related) || isContainerMarkedAsRoot(related)) {
          return;
        }
      }
    }

    if (!isOutEvent && !isOverEvent) {
      // Must not be a mouse or pointer in or out - ignoring.
      return;
    }

    var win; // TODO: why is this nullable in the types but we read from it?

    if (nativeEventTarget.window === nativeEventTarget) {
      // `nativeEventTarget` is probably a window object.
      win = nativeEventTarget;
    } else {
      // TODO: Figure out why `ownerDocument` is sometimes undefined in IE8.
      var doc = nativeEventTarget.ownerDocument;

      if (doc) {
        win = doc.defaultView || doc.parentWindow;
      } else {
        win = window;
      }
    }

    var from;
    var to;

    if (isOutEvent) {
      var _related = nativeEvent.relatedTarget || nativeEvent.toElement;
      from = targetInst;
      to = _related ? getClosestInstanceFromNode(_related) : null;

      if (to !== null) {
        var nearestMounted = getNearestMountedFiber(to);

        if (to !== nearestMounted || to.tag !== HostComponent && to.tag !== HostText) {
          to = null;
        }
      }
    } else {
      // Moving to a node from outside the window.
      from = null;
      to = targetInst;
    }

    if (from === to) {
      // Nothing pertains to our managed components.
      return;
    }

    var SyntheticEventCtor = SyntheticMouseEvent;
    var leaveEventType = 'onMouseLeave';
    var enterEventType = 'onMouseEnter';
    var eventTypePrefix = 'mouse';

    if (domEventName === 'pointerout' || domEventName === 'pointerover') {
      SyntheticEventCtor = SyntheticPointerEvent;
      leaveEventType = 'onPointerLeave';
      enterEventType = 'onPointerEnter';
      eventTypePrefix = 'pointer';
    }

    var fromNode = from == null ? win : getNodeFromInstance(from);
    var toNode = to == null ? win : getNodeFromInstance(to);
    var leave = new SyntheticEventCtor(leaveEventType, eventTypePrefix + 'leave', from,
nativeEvent, nativeEventTarget);
    leave.target = fromNode;
    leave.relatedTarget = toNode;
    var enter = null; // We should only process this nativeEvent if we are processing
    // the first ancestor. Next time, we will ignore the event.

    var nativeTargetInst = getClosestInstanceFromNode(nativeEventTarget);

    if (nativeTargetInst === targetInst) {
      var enterEvent = new SyntheticEventCtor(enterEventType, eventTypePrefix + 'enter',
to, nativeEvent, nativeEventTarget);
      enterEvent.target = toNode;
      enterEvent.relatedTarget = fromNode;
      enter = enterEvent;
    }

    accumulateEnterLeaveTwoPhaseListeners(dispatchQueue, leave, enter, from, to);
  }

  /**
   * inlined Object.is polyfill to avoid requiring consumers ship their own
   * https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Global_Objects/Object/is
   */
  function is(x, y) {
    return x === y && (x !== 0 || 1 / x === 1 / y) || x !== x && y !== y // eslint-
disable-line no-self-compare
    ;
  }

  var objectIs = typeof Object.is === 'function' ? Object.is : is;
  /**
   * Performs equality by iterating through keys on an object and returning false
   * when any key has values which are not strictly equal between the arguments.
   * Returns true when the values of all keys are strictly equal.
   */

  function shallowEqual(objA, objB) {
    if (objectIs(objA, objB)) {
      return true;
    }

    if (typeof objA !== 'object' || objA === null || typeof objB !== 'object' || objB ===
null) {
      return false;
    }

    var keysA = Object.keys(objA);
    var keysB = Object.keys(objB);

    if (keysA.length !== keysB.length) {
      return false;
    } // Test for A's keys different from B.


    for (var i = 0; i < keysA.length; i++) {
      var currentKey = keysA[i];

      if (!hasOwnProperty.call(objB, currentKey) || !objectIs(objA[currentKey],
objB[currentKey])) {
        return false;
      }
    }

    return true;
  }

  /**
   * Given any node return the first leaf node without children.
   *
   * @param {DOMElement|DOMTextNode} node
   * @return {DOMElement|DOMTextNode}
   */

  function getLeafNode(node) {
    while (node && node.firstChild) {
      node = node.firstChild;
    }

    return node;
  }
  /**
   * Get the next sibling within a container. This will walk up the
   * DOM if a node's siblings have been exhausted.
   *
   * @param {DOMElement|DOMTextNode} node
   * @return {?DOMElement|DOMTextNode}
   */


  function getSiblingNode(node) {
    while (node) {
      if (node.nextSibling) {
        return node.nextSibling;
      }

      node = node.parentNode;
    }
  }
  /**
```

```
   * Get object describing the nodes which contain characters at offset.
   *
   * @param {DOMElement|DOMTextNode} root
   * @param {number} offset
   * @return {?object}
   */


  function getNodeForCharacterOffset(root, offset) {
    var node = getLeafNode(root);
    var nodeStart = 0;
    var nodeEnd = 0;

    while (node) {
      if (node.nodeType === TEXT_NODE) {
        nodeEnd = nodeStart + node.textContent.length;

        if (nodeStart <= offset && nodeEnd >= offset) {
          return {
            node: node,
            offset: offset - nodeStart
          };
        }

        nodeStart = nodeEnd;
      }

      node = getLeafNode(getSiblingNode(node));
    }
  }

  /**
   * @param {DOMElement} outerNode
   * @return {?object}
   */

  function getOffsets(outerNode) {
    var ownerDocument = outerNode.ownerDocument;
    var win = ownerDocument && ownerDocument.defaultView || window;
    var selection = win.getSelection && win.getSelection();

    if (!selection || selection.rangeCount === 0) {
      return null;
    }

    var anchorNode = selection.anchorNode,
        anchorOffset = selection.anchorOffset,
        focusNode = selection.focusNode,
        focusOffset = selection.focusOffset; // In Firefox, anchorNode and focusNode can
be "anonymous divs", e.g. the
    // up/down buttons on an <input type="number">. Anonymous divs do not seem to
    // expose properties, triggering a "Permission denied error" if any of its
    // properties are accessed. The only seemingly possible way to avoid erroring
    // is to access a property that typically works for non-anonymous divs and
    // catch any error that may otherwise arise. See
    // https://bugzilla.mozilla.org/show_bug.cgi?id=208427

    try {
      /* eslint-disable no-unused-expressions */
      anchorNode.nodeType;
      focusNode.nodeType;
      /* eslint-enable no-unused-expressions */
    } catch (e) {
      return null;
    }

    return getModernOffsetsFromPoints(outerNode, anchorNode, anchorOffset, focusNode,
focusOffset);
  }
  /**
   * Returns {start, end} where `start` is the character/codepoint index of
   * (anchorNode, anchorOffset) within the textContent of `outerNode`, and
   * `end` is the index of (focusNode, focusOffset).
   *
   * Returns null if you pass in garbage input but we should probably just crash.
   *
   * Exported only for testing.
   */

  function getModernOffsetsFromPoints(outerNode, anchorNode, anchorOffset, focusNode,
focusOffset) {
    var length = 0;
    var start = -1;
    var end = -1;
    var indexWithinAnchor = 0;
    var indexWithinFocus = 0;
    var node = outerNode;
    var parentNode = null;

    outer: while (true) {
      var next = null;

      while (true) {
        if (node === anchorNode && (anchorOffset === 0 || node.nodeType === TEXT_NODE)) {
          start = length + anchorOffset;
        }

        if (node === focusNode && (focusOffset === 0 || node.nodeType === TEXT_NODE)) {
          end = length + focusOffset;
        }

        if (node.nodeType === TEXT_NODE) {
          length += node.nodeValue.length;
        }

        if ((next = node.firstChild) === null) {
          break;
        } // Moving from `node` to its first child `next`.


        parentNode = node;
        node = next;
      }

      while (true) {
        if (node === outerNode) {
          // If `outerNode` has children, this is always the second time visiting
          // it. If it has no children, this is still the first loop, and the only
          // valid selection is anchorNode and focusNode both equal to this node
          // and both offsets 0, in which case we will have handled above.
          break outer;
        }

        if (parentNode === anchorNode && ++indexWithinAnchor === anchorOffset) {
          start = length;
        }

        if (parentNode === focusNode && ++indexWithinFocus === focusOffset) {
          end = length;
        }

        if ((next = node.nextSibling) !== null) {
          break;
        }

        node = parentNode;
        parentNode = node.parentNode;
      } // Moving from `node` to its next sibling `next`.
      node = next;
    }

    if (start === -1 || end === -1) {
      // This should never happen. (Would happen if the anchor/focus nodes aren't
      // actually inside the passed-in node.)
      return null;
    }

    return {
      start: start,
      end: end
    };
  }
  /**
   * In modern non-IE browsers, we can support both forward and backward
   * selections.
   *
   * Note: IE10+ supports the Selection object, but it does not support
   * the `extend` method, which means that even in modern IE, it's not possible
   * to programmatically create a backward selection. Thus, for all IE
   * versions, we use the old IE API to create our selections.
   *
   * @param {DOMElement|DOMTextNode} node
   * @param {object} offsets
   */

  function setOffsets(node, offsets) {
    var doc = node.ownerDocument || document;
    var win = doc && doc.defaultView || window; // Edge fails with "Object expected" in
some scenarios.
    // (For instance: TinyMCE editor used in a list component that supports pasting to
add more,
    // fails when pasting 100+ items)

    if (!win.getSelection) {
      return;
    }

    var selection = win.getSelection();
    var length = node.textContent.length;
    var start = Math.min(offsets.start, length);
    var end = offsets.end === undefined ? start : Math.min(offsets.end, length); // IE 11
uses modern selection, but doesn't support the extend method.
    // Flip backward selections, so we can set with a single range.

    if (!selection.extend && start > end) {
      var temp = end;
      end = start;
      start = temp;
    }

    var startMarker = getNodeForCharacterOffset(node, start);
    var endMarker = getNodeForCharacterOffset(node, end);

    if (startMarker && endMarker) {
      if (selection.rangeCount === 1 && selection.anchorNode === startMarker.node &&
selection.anchorOffset === startMarker.offset && selection.focusNode === endMarker.node
&& selection.focusOffset === endMarker.offset) {
        return;
      }

      var range = doc.createRange();
      range.setStart(startMarker.node, startMarker.offset);
      selection.removeAllRanges();

      if (start > end) {
        selection.addRange(range);
        selection.extend(endMarker.node, endMarker.offset);
      } else {
        range.setEnd(endMarker.node, endMarker.offset);
        selection.addRange(range);
      }
    }
  }

  function isTextNode(node) {
    return node && node.nodeType === TEXT_NODE;
  }

  function containsNode(outerNode, innerNode) {
    if (!outerNode || !innerNode) {
      return false;
    } else if (outerNode === innerNode) {
      return true;
    } else if (isTextNode(outerNode)) {
      return false;
    } else if (isTextNode(innerNode)) {
      return containsNode(outerNode, innerNode.parentNode);
    } else if ('contains' in outerNode) {
      return outerNode.contains(innerNode);
    } else if (outerNode.compareDocumentPosition) {
      return !!(outerNode.compareDocumentPosition(innerNode) & 16);
    } else {
      return false;
    }
  }

  function isInDocument(node) {
    return node && node.ownerDocument && containsNode(node.ownerDocument.documentElement,
node);
  }

  function isSameOriginFrame(iframe) {
    try {
      // Accessing the contentDocument of a HTMLIframeElement can cause the browser
      // to throw, e.g. if it has a cross-origin src attribute.
      // Safari will show an error in the console when the access results in "Blocked a
frame with origin". e.g:
      // iframe.contentDocument.defaultView;
      // A safety way is to access one of the cross origin properties: Window or Location
      // Which might result in "SecurityError" DOM Exception and it is compatible to
Safari.
      // https://html.spec.whatwg.org/multipage/browsers.html#integration-with-idl
      return typeof iframe.contentWindow.location.href === 'string';
    } catch (err) {
      return false;
    }
  }

  function getActiveElementDeep() {
    var win = window;
    var element = getActiveElement();

    while (element instanceof win.HTMLIFrameElement) {
      if (isSameOriginFrame(element)) {
        win = element.contentWindow;
      } else {
        return element;
      }

      element = getActiveElement(win.document);
    }

    return element;
  }
  /**
   * @ReactInputSelection: React input selection module. Based on Selection.js,
   * but modified to be suitable for react and has a couple of bug fixes (doesn't
   * assume buttons have range selections allowed).
   * Input selection module for React.
   */

  /**
   * @hasSelectionCapabilities: we get the element types that support selection
   * from https://html.spec.whatwg.org/#do-not-apply, looking at `selectionStart`
   * and `selectionEnd` rows.
   */


  function hasSelectionCapabilities(elem) {
    var nodeName = elem && elem.nodeName && elem.nodeName.toLowerCase();
    return nodeName && (nodeName === 'input' && (elem.type === 'text' || elem.type ===
'search' || elem.type === 'tel' || elem.type === 'url' || elem.type === 'password') ||
nodeName === 'textarea' || elem.contentEditable === 'true');
  }
  function getSelectionInformation() {
    var focusedElem = getActiveElementDeep();
    return {
      focusedElem: focusedElem,
      selectionRange: hasSelectionCapabilities(focusedElem) ? getSelection(focusedElem) :
null
    };
  }
  /**
   * @restoreSelection: If any selection information was potentially lost,
   * restore it. This is useful when performing operations that could remove dom
   * nodes and place them back in, resulting in focus being lost.
   */

  function restoreSelection(priorSelectionInformation) {
    var curFocusedElem = getActiveElementDeep();
    var priorFocusedElem = priorSelectionInformation.focusedElem;
    var priorSelectionRange = priorSelectionInformation.selectionRange;

    if (curFocusedElem !== priorFocusedElem && isInDocument(priorFocusedElem)) {
      if (priorSelectionRange !== null && hasSelectionCapabilities(priorFocusedElem)) {
        setSelection(priorFocusedElem, priorSelectionRange);
      } // Focusing a node can change the scroll position, which is undesirable


      var ancestors = [];
      var ancestor = priorFocusedElem;

      while (ancestor = ancestor.parentNode) {
        if (ancestor.nodeType === ELEMENT_NODE) {
          ancestors.push({
            element: ancestor,
            left: ancestor.scrollLeft,
            top: ancestor.scrollTop
          });
        }
      }

      if (typeof priorFocusedElem.focus === 'function') {
        priorFocusedElem.focus();
      }

      for (var i = 0; i < ancestors.length; i++) {
        var info = ancestors[i];
        info.element.scrollLeft = info.left;
        info.element.scrollTop = info.top;
      }
    }
  }
  /**
   * @getSelection: Gets the selection bounds of a focused textarea, input or
   * contentEditable node.
   * -@input: Look up selection bounds of this input
   * -@return {start: selectionStart, end: selectionEnd}
   */

  function getSelection(input) {
    var selection;

    if ('selectionStart' in input) {
      // Modern browser with input or textarea.
      selection = {
        start: input.selectionStart,
        end: input.selectionEnd
      };
    } else {
      // Content editable or old IE textarea.
      selection = getOffsets(input);
    }

    return selection || {
      start: 0,
      end: 0
    };
  }
  /**
   * @setSelection: Sets the selection bounds of a textarea or input and focuses
   * the input.
   * -@input     Set selection bounds of this input or textarea
   * -@offsets   Object of same form that is returned from get*
   */

  function setSelection(input, offsets) {
    var start = offsets.start;
    var end = offsets.end;

    if (end === undefined) {
      end = start;
    }

    if ('selectionStart' in input) {
      input.selectionStart = start;
      input.selectionEnd = Math.min(end, input.value.length);
    } else {
      setOffsets(input, offsets);
    }
  }

  var skipSelectionChangeEvent = canUseDOM && 'documentMode' in document &&
document.documentMode <= 11;

  function registerEvents$3() {
    registerTwoPhaseEvent('onSelect', ['focusout', 'contextmenu', 'dragend', 'focusin',
'keydown', 'keyup', 'mousedown', 'mouseup', 'selectionchange']);
  }

  var activeElement$1 = null;
  var activeElementInst$1 = null;
  var lastSelection = null;
  var mouseDown = false;
  /**
   * Get an object which is a unique representation of the current selection.
   *
   * The return value will not be consistent across nodes or browsers, but
   * two identical selections on the same node will return identical objects.
   */

  function getSelection$1(node) {
    if ('selectionStart' in node && hasSelectionCapabilities(node)) {
      return {
        start: node.selectionStart,
        end: node.selectionEnd
      };
    } else {
      var win = node.ownerDocument && node.ownerDocument.defaultView || window;
      var selection = win.getSelection();
      return {
        anchorNode: selection.anchorNode,
        anchorOffset: selection.anchorOffset,
        focusNode: selection.focusNode,
        focusOffset: selection.focusOffset
      };
    }
  }
  /**
   * Get document associated with the event target.
   */


  function getEventTargetDocument(eventTarget) {
    return eventTarget.window === eventTarget ? eventTarget.document :
eventTarget.nodeType === DOCUMENT_NODE ? eventTarget : eventTarget.ownerDocument;
  }
  /**
   * Poll selection to see whether it's changed.
   *
   * @param {object} nativeEvent
   * @param {object} nativeEventTarget
   * @return {?SyntheticEvent}
   */


  function constructSelectEvent(dispatchQueue, nativeEvent, nativeEventTarget) {
    // Ensure we have the right element, and that the user is not dragging a
    // selection (this matches native `select` event behavior). In HTML5, select
    // fires only on input and textarea thus if there's no focused element we
    // won't dispatch.
    var doc = getEventTargetDocument(nativeEventTarget);

    if (mouseDown || activeElement$1 == null || activeElement$1 !==
getActiveElement(doc)) {
      return;
    } // Only fire when selection has actually changed.


    var currentSelection = getSelection$1(activeElement$1);

    if (!lastSelection || !shallowEqual(lastSelection, currentSelection)) {
      lastSelection = currentSelection;
      var listeners = accumulateTwoPhaseListeners(activeElementInst$1, 'onSelect');

      if (listeners.length > 0) {
        var event = new SyntheticEvent('onSelect', 'select', null, nativeEvent,
nativeEventTarget);
        dispatchQueue.push({
          event: event,
          listeners: listeners
        });
        event.target = activeElement$1;
      }
    }
  }
  /**
   * This plugin creates an `onSelect` event that normalizes select events
   * across form elements.
   *
   * Supported elements are:
```

```
   * - input (see `isTextInputElement`)
   * - textarea
   * - contentEditable
   *
   * This differs from native browser implementations in the following ways:
   * - Fires on contentEditable fields as well as inputs.
   * - Fires for collapsed selection.
   * - Fires after user input.
   */


  function extractEvents$3(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget, eventSystemFlags, targetContainer) {
    var targetNode = targetInst ? getNodeFromInstance(targetInst) : window;

    switch (domEventName) {
      // Track the input node that has focus.
      case 'focusin':
        if (isTextInputElement(targetNode) || targetNode.contentEditable === 'true') {
          activeElement$1 = targetNode;
          activeElementInst$1 = targetInst;
          lastSelection = null;
        }

        break;

      case 'focusout':
        activeElement$1 = null;
        activeElementInst$1 = null;
        lastSelection = null;
        break;
      // Don't fire the event while the user is dragging. This matches the
      // semantics of the native select event.

      case 'mousedown':
        mouseDown = true;
        break;

      case 'contextmenu':
      case 'mouseup':
      case 'dragend':
        mouseDown = false;
        constructSelectEvent(dispatchQueue, nativeEvent, nativeEventTarget);
        break;
      // Chrome and IE fire non-standard event when selection is changed (and
      // sometimes when it hasn't). IE's event fires out of order with respect
      // to key and input events on deletion, so we discard it.
      //
      // Firefox doesn't support selectionchange, so check selection status
      // after each key entry. The selection changes after keydown and before
      // keyup, but we check on keydown as well in the case of holding down a
      // key, when multiple keydown events are fired but only one keyup is.
      // This is also our approach for IE handling, for the reason above.

      case 'selectionchange':
        if (skipSelectionChangeEvent) {
          break;
        }

      // falls through

      case 'keydown':
      case 'keyup':
        constructSelectEvent(dispatchQueue, nativeEvent, nativeEventTarget);
    }
  }

  /**
   * Generate a mapping of standard vendor prefixes using the defined style property and
```

```
event name.
   *
   * @param {string} styleProp
   * @param {string} eventName
   * @returns {object}
   */

  function makePrefixMap(styleProp, eventName) {
    var prefixes = {};
    prefixes[styleProp.toLowerCase()] = eventName.toLowerCase();
    prefixes['Webkit' + styleProp] = 'webkit' + eventName;
    prefixes['Moz' + styleProp] = 'moz' + eventName;
    return prefixes;
  }
  /**
   * A list of event names to a configurable list of vendor prefixes.
   */


  var vendorPrefixes = {
    animationend: makePrefixMap('Animation', 'AnimationEnd'),
    animationiteration: makePrefixMap('Animation', 'AnimationIteration'),
    animationstart: makePrefixMap('Animation', 'AnimationStart'),
    transitionend: makePrefixMap('Transition', 'TransitionEnd')
  };
  /**
   * Event names that have already been detected and prefixed (if applicable).
   */

  var prefixedEventNames = {};
  /**
   * Element to check for prefixes on.
   */

  var style = {};
  /**
   * Bootstrap if a DOM exists.
   */

  if (canUseDOM) {
    style = document.createElement('div').style; // On some platforms, in particular some
releases of Android 4.x,
    // the un-prefixed "animation" and "transition" properties are defined on the
    // style object but the events that fire will still be prefixed, so we need
    // to check if the un-prefixed events are usable, and if not remove them from the
map.

    if (!('AnimationEvent' in window)) {
      delete vendorPrefixes.animationend.animation;
      delete vendorPrefixes.animationiteration.animation;
      delete vendorPrefixes.animationstart.animation;
    } // Same as above


    if (!('TransitionEvent' in window)) {
      delete vendorPrefixes.transitionend.transition;
    }
  }
  /**
   * Attempts to determine the correct vendor prefixed event name.
   *
   * @param {string} eventName
   * @returns {string}
   */


  function getVendorPrefixedEventName(eventName) {
    if (prefixedEventNames[eventName]) {
      return prefixedEventNames[eventName];
    } else if (!vendorPrefixes[eventName]) {
      return eventName;
    }

    var prefixMap = vendorPrefixes[eventName];

    for (var styleProp in prefixMap) {
      if (prefixMap.hasOwnProperty(styleProp) && styleProp in style) {
        return prefixedEventNames[eventName] = prefixMap[styleProp];
      }
    }

    return eventName;
  }

  var ANIMATION_END = getVendorPrefixedEventName('animationend');
  var ANIMATION_ITERATION = getVendorPrefixedEventName('animationiteration');
  var ANIMATION_START = getVendorPrefixedEventName('animationstart');
  var TRANSITION_END = getVendorPrefixedEventName('transitionend');

  var topLevelEventsToReactNames = new Map(); // NOTE: Capitalization is important in
this list!
  //
  // E.g. it needs "pointerDown", not "pointerdown".
  // This is because we derive both React name ("onPointerDown")
  // and DOM name ("pointerdown") from the same list.
  //
  // Exceptions that don't match this convention are listed separately.
  //
  // prettier-ignore

  var simpleEventPluginEvents = ['abort', 'auxClick', 'cancel', 'canPlay',
'canPlayThrough', 'click', 'close', 'contextMenu', 'copy', 'cut', 'drag', 'dragEnd',
'dragEnter', 'dragExit', 'dragLeave', 'dragOver', 'dragStart', 'drop', 'durationChange',
'emptied', 'encrypted', 'ended', 'error', 'gotPointerCapture', 'input', 'invalid',
'keyDown', 'keyPress', 'keyUp', 'load', 'loadedData', 'loadedMetadata', 'loadStart',
'lostPointerCapture', 'mouseDown', 'mouseMove', 'mouseOut', 'mouseOver', 'mouseUp',
'paste', 'pause', 'play', 'playing', 'pointerCancel', 'pointerDown', 'pointerMove',
'pointerOut', 'pointerOver', 'pointerUp', 'progress', 'rateChange', 'reset', 'resize',
'seeked', 'seeking', 'stalled', 'submit', 'suspend', 'timeUpdate', 'touchCancel',
'touchEnd', 'touchStart', 'volumeChange', 'scroll', 'toggle', 'touchMove', 'waiting',
'wheel'];

  function registerSimpleEvent(domEventName, reactName) {
    topLevelEventsToReactNames.set(domEventName, reactName);
    registerTwoPhaseEvent(reactName, [domEventName]);
  }

  function registerSimpleEvents() {
    for (var i = 0; i < simpleEventPluginEvents.length; i++) {
      var eventName = simpleEventPluginEvents[i];
      var domEventName = eventName.toLowerCase();
      var capitalizedEvent = eventName[0].toUpperCase() + eventName.slice(1);
      registerSimpleEvent(domEventName, 'on' + capitalizedEvent);
    } // Special cases where event names don't match.


    registerSimpleEvent(ANIMATION_END, 'onAnimationEnd');
    registerSimpleEvent(ANIMATION_ITERATION, 'onAnimationIteration');
    registerSimpleEvent(ANIMATION_START, 'onAnimationStart');
    registerSimpleEvent('dblclick', 'onDoubleClick');
    registerSimpleEvent('focusin', 'onFocus');
    registerSimpleEvent('focusout', 'onBlur');
    registerSimpleEvent(TRANSITION_END, 'onTransitionEnd');
  }

  function extractEvents$4(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget, eventSystemFlags, targetContainer) {
    var reactName = topLevelEventsToReactNames.get(domEventName);
```

```
      if (reactName === undefined) {
        return;
      }

      var SyntheticEventCtor = SyntheticEvent;
      var reactEventType = domEventName;

      switch (domEventName) {
        case 'keypress':
          // Firefox creates a keypress event for function keys too. This removes
          // the unwanted keypress events. Enter is however both printable and
          // non-printable. One would expect Tab to be as well (but it isn't).
          if (getEventCharCode(nativeEvent) === 0) {
            return;
          }

        /* falls through */

        case 'keydown':
        case 'keyup':
          SyntheticEventCtor = SyntheticKeyboardEvent;
          break;

        case 'focusin':
          reactEventType = 'focus';
          SyntheticEventCtor = SyntheticFocusEvent;
          break;

        case 'focusout':
          reactEventType = 'blur';
          SyntheticEventCtor = SyntheticFocusEvent;
          break;

        case 'beforeblur':
        case 'afterblur':
          SyntheticEventCtor = SyntheticFocusEvent;
          break;

        case 'click':
          // Firefox creates a click event on right mouse clicks. This removes the
          // unwanted click events.
          if (nativeEvent.button === 2) {
            return;
          }

        /* falls through */

        case 'auxclick':
        case 'dblclick':
        case 'mousedown':
        case 'mousemove':
        case 'mouseup': // TODO: Disabled elements should not respond to mouse events

        /* falls through */

        case 'mouseout':
        case 'mouseover':
        case 'contextmenu':
          SyntheticEventCtor = SyntheticMouseEvent;
          break;

        case 'drag':
        case 'dragend':
        case 'dragenter':
        case 'dragexit':
        case 'dragleave':
        case 'dragover':
        case 'dragstart':
```

```
      case 'drop':
        SyntheticEventCtor = SyntheticDragEvent;
        break;

      case 'touchcancel':
      case 'touchend':
      case 'touchmove':
      case 'touchstart':
        SyntheticEventCtor = SyntheticTouchEvent;
        break;

      case ANIMATION_END:
      case ANIMATION_ITERATION:
      case ANIMATION_START:
        SyntheticEventCtor = SyntheticAnimationEvent;
        break;

      case TRANSITION_END:
        SyntheticEventCtor = SyntheticTransitionEvent;
        break;

      case 'scroll':
        SyntheticEventCtor = SyntheticUIEvent;
        break;

      case 'wheel':
        SyntheticEventCtor = SyntheticWheelEvent;
        break;

      case 'copy':
      case 'cut':
      case 'paste':
        SyntheticEventCtor = SyntheticClipboardEvent;
        break;

      case 'gotpointercapture':
      case 'lostpointercapture':
      case 'pointercancel':
      case 'pointerdown':
      case 'pointermove':
      case 'pointerout':
      case 'pointerover':
      case 'pointerup':
        SyntheticEventCtor = SyntheticPointerEvent;
        break;
    }

    var inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;

    {
      // Some events don't bubble in the browser.
      // In the past, React has always bubbled them, but this can be surprising.
      // We're going to try aligning closer to the browser behavior by not bubbling
      // them in React either. We'll start by not bubbling onScroll, and then expand.
      var accumulateTargetOnly = !inCapturePhase && // TODO: ideally, we'd eventually add
all events from
      // nonDelegatedEvents list in DOMPluginEventSystem.
      // Then we can remove this special list.
      // This is a breaking change that can wait until React 18.
      domEventName === 'scroll';

      var _listeners = accumulateSinglePhaseListeners(targetInst, reactName,
nativeEvent.type, inCapturePhase, accumulateTargetOnly);

      if (_listeners.length > 0) {
        // Intentionally create event lazily.
        var _event = new SyntheticEventCtor(reactName, reactEventType, null, nativeEvent,
nativeEventTarget);
```

```
        dispatchQueue.push({
          event: _event,
          listeners: _listeners
        });
      }
    }
  }

  // TODO: remove top-level side effect.
  registerSimpleEvents();
  registerEvents$2();
  registerEvents$1();
  registerEvents$3();
  registerEvents();

  function extractEvents$5(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget, eventSystemFlags, targetContainer) {
    // TODO: we should remove the concept of a "SimpleEventPlugin".
    // This is the basic functionality of the event system. All
    // the other plugins are essentially polyfills. So the plugin
    // should probably be inlined somewhere and have its logic
    // be core the to event system. This would potentially allow
    // us to ship builds of React without the polyfilled plugins below.
    extractEvents$4(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget, eventSystemFlags);
    var shouldProcessPolyfillPlugins = (eventSystemFlags &
SHOULD_NOT_PROCESS_POLYFILL_EVENT_PLUGINS) === 0; // We don't process these events unless
we are in the
    // event's native "bubble" phase, which means that we're
    // not in the capture phase. That's because we emulate
    // the capture phase here still. This is a trade-off,
    // because in an ideal world we would not emulate and use
    // the phases properly, like we do with the SimpleEvent
    // plugin. However, the plugins below either expect
    // emulation (EnterLeave) or use state localized to that
    // plugin (BeforeInput, Change, Select). The state in
    // these modules complicates things, as you'll essentially
    // get the case where the capture phase event might change
    // state, only for the following bubble event to come in
    // later and not trigger anything as the state now
    // invalidates the heuristics of the event plugin. We
    // could alter all these plugins to work in such ways, but
    // that might cause other unknown side-effects that we
    // can't foresee right now.

    if (shouldProcessPolyfillPlugins) {
      extractEvents$2(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget);
      extractEvents$1(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget);
      extractEvents$3(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget);
      extractEvents(dispatchQueue, domEventName, targetInst, nativeEvent,
nativeEventTarget);
    }
  } // List of events that need to be individually attached to media elements.


  var mediaEventTypes = ['abort', 'canplay', 'canplaythrough', 'durationchange',
'emptied', 'encrypted', 'ended', 'error', 'loadeddata', 'loadedmetadata', 'loadstart',
'pause', 'play', 'playing', 'progress', 'ratechange', 'resize', 'seeked', 'seeking',
'stalled', 'suspend', 'timeupdate', 'volumechange', 'waiting']; // We should not delegate
these events to the container, but rather
  // set them on the actual target element itself. This is primarily
  // because these events do not consistently bubble in the DOM.

  var nonDelegatedEvents = new Set(['cancel', 'close', 'invalid', 'load', 'scroll',
'toggle'].concat(mediaEventTypes));
```

```
    function executeDispatch(event, listener, currentTarget) {
      var type = event.type || 'unknown-event';
      event.currentTarget = currentTarget;
      invokeGuardedCallbackAndCatchFirstError(type, listener, undefined, event);
      event.currentTarget = null;
    }

    function processDispatchQueueItemsInOrder(event, dispatchListeners, inCapturePhase) {
      var previousInstance;

      if (inCapturePhase) {
        for (var i = dispatchListeners.length - 1; i >= 0; i--) {
          var _dispatchListeners$i = dispatchListeners[i],
              instance = _dispatchListeners$i.instance,
              currentTarget = _dispatchListeners$i.currentTarget,
              listener = _dispatchListeners$i.listener;

          if (instance !== previousInstance && event.isPropagationStopped()) {
            return;
          }

          executeDispatch(event, listener, currentTarget);
          previousInstance = instance;
        }
      } else {
        for (var _i = 0; _i < dispatchListeners.length; _i++) {
          var _dispatchListeners$_i = dispatchListeners[_i],
              _instance = _dispatchListeners$_i.instance,
              _currentTarget = _dispatchListeners$_i.currentTarget,
              _listener = _dispatchListeners$_i.listener;

          if (_instance !== previousInstance && event.isPropagationStopped()) {
            return;
          }

          executeDispatch(event, _listener, _currentTarget);
          previousInstance = _instance;
        }
      }
    }

    function processDispatchQueue(dispatchQueue, eventSystemFlags) {
      var inCapturePhase = (eventSystemFlags & IS_CAPTURE_PHASE) !== 0;

      for (var i = 0; i < dispatchQueue.length; i++) {
        var _dispatchQueue$i = dispatchQueue[i],
            event = _dispatchQueue$i.event,
            listeners = _dispatchQueue$i.listeners;
        processDispatchQueueItemsInOrder(event, listeners, inCapturePhase); //  event
  system doesn't use pooling.
      } // This would be a good time to rethrow if any of the event handlers threw.


      rethrowCaughtError();
    }

    function dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent,
  targetInst, targetContainer) {
      var nativeEventTarget = getEventTarget(nativeEvent);
      var dispatchQueue = [];
      extractEvents$5(dispatchQueue, domEventName, targetInst, nativeEvent,
  nativeEventTarget, eventSystemFlags);
      processDispatchQueue(dispatchQueue, eventSystemFlags);
    }

    function listenToNonDelegatedEvent(domEventName, targetElement) {
      {
        if (!nonDelegatedEvents.has(domEventName)) {
          error('Did not expect a listenToNonDelegatedEvent() call for "%s". ' + 'This is a
  bug in React. Please file an issue.', domEventName);
        }
      }

      var isCapturePhaseListener = false;
      var listenerSet = getEventListenerSet(targetElement);
      var listenerSetKey = getListenerSetKey(domEventName, isCapturePhaseListener);

      if (!listenerSet.has(listenerSetKey)) {
        addTrappedEventListener(targetElement, domEventName, IS_NON_DELEGATED,
  isCapturePhaseListener);
        listenerSet.add(listenerSetKey);
      }
    }
    function listenToNativeEvent(domEventName, isCapturePhaseListener, target) {
      {
        if (nonDelegatedEvents.has(domEventName) && !isCapturePhaseListener) {
          error('Did not expect a listenToNativeEvent() call for "%s" in the bubble phase.
  ' + 'This is a bug in React. Please file an issue.', domEventName);
        }
      }

      var eventSystemFlags = 0;

      if (isCapturePhaseListener) {
        eventSystemFlags |= IS_CAPTURE_PHASE;
      }

      addTrappedEventListener(target, domEventName, eventSystemFlags,
  isCapturePhaseListener);
    } // This is only used by createEventHandle when the
    var listeningMarker = '_reactListening' + Math.random().toString(36).slice(2);
    function listenToAllSupportedEvents(rootContainerElement) {
      if (!rootContainerElement[listeningMarker]) {
        rootContainerElement[listeningMarker] = true;
        allNativeEvents.forEach(function (domEventName) {
          // We handle selectionchange separately because it
          // doesn't bubble and needs to be on the document.
          if (domEventName !== 'selectionchange') {
            if (!nonDelegatedEvents.has(domEventName)) {
              listenToNativeEvent(domEventName, false, rootContainerElement);
            }

            listenToNativeEvent(domEventName, true, rootContainerElement);
          }
        });
        var ownerDocument = rootContainerElement.nodeType === DOCUMENT_NODE ?
  rootContainerElement : rootContainerElement.ownerDocument;

        if (ownerDocument !== null) {
          // The selectionchange event also needs deduplication
          // but it is attached to the document.
          if (!ownerDocument[listeningMarker]) {
            ownerDocument[listeningMarker] = true;
            listenToNativeEvent('selectionchange', false, ownerDocument);
          }
        }
      }
    }

    function addTrappedEventListener(targetContainer, domEventName, eventSystemFlags,
  isCapturePhaseListener, isDeferredListenerForLegacyFBSupport) {
      var listener = createEventListenerWrapperWithPriority(targetContainer, domEventName,
  eventSystemFlags); // If passive option is not supported, then the event will be
      // active and not passive.

      var isPassiveListener = undefined;

      if (passiveBrowserEventsSupported) {
        // Browsers introduced an intervention, making these events
        // passive by default on document. React doesn't bind them
        // to document anymore, but changing this now would undo
        // the performance wins from the change. So we emulate
        // the existing behavior manually on the roots now.
        // https://github.com/facebook/react/issues/19651
        if (domEventName === 'touchstart' || domEventName === 'touchmove' || domEventName
  === 'wheel') {
          isPassiveListener = true;
        }
      }

      targetContainer =  targetContainer;
      var unsubscribeListener; // When legacyFBSupport is enabled, it's for when we


      if (isCapturePhaseListener) {
        if (isPassiveListener !== undefined) {
          unsubscribeListener = addEventCaptureListenerWithPassiveFlag(targetContainer,
  domEventName, listener, isPassiveListener);
        } else {
          unsubscribeListener = addEventCaptureListener(targetContainer, domEventName,
  listener);
        }
      } else {
        if (isPassiveListener !== undefined) {
          unsubscribeListener = addEventBubbleListenerWithPassiveFlag(targetContainer,
  domEventName, listener, isPassiveListener);
        } else {
          unsubscribeListener = addEventBubbleListener(targetContainer, domEventName,
  listener);
        }
      }
    }

    function isMatchingRootContainer(grandContainer, targetContainer) {
      return grandContainer === targetContainer || grandContainer.nodeType === COMMENT_NODE
  && grandContainer.parentNode === targetContainer;
    }

    function dispatchEventForPluginEventSystem(domEventName, eventSystemFlags, nativeEvent,
  targetInst, targetContainer) {
      var ancestorInst = targetInst;

      if ((eventSystemFlags & IS_EVENT_HANDLE_NON_MANAGED_NODE) === 0 && (eventSystemFlags
  & IS_NON_DELEGATED) === 0) {
        var targetContainerNode = targetContainer; // If we are using the legacy FB support
  flag, we

        if (targetInst !== null) {
          // The below logic attempts to work out if we need to change
          // the target fiber to a different ancestor. We had similar logic
          // in the legacy event system, except the big difference between
          // systems is that the modern event system now has an event listener
          // attached to each React Root and React Portal Root. Together,
          // the DOM nodes representing these roots are the "rootContainer".
          // To figure out which ancestor instance we should use, we traverse
          // up the fiber tree from the target instance and attempt to find
          // root boundaries that match that of our current "rootContainer".
          // If we find that "rootContainer", we find the parent fiber
          // sub-tree for that root and make that our ancestor instance.
          var node = targetInst;

          mainLoop: while (true) {
            if (node === null) {
              return;
            }

            var nodeTag = node.tag;
            if (nodeTag === HostRoot || nodeTag === HostPortal) {
              var container = node.stateNode.containerInfo;

              if (isMatchingRootContainer(container, targetContainerNode)) {
                break;
              }

              if (nodeTag === HostPortal) {
                // The target is a portal, but it's not the rootContainer we're looking
  for.
                // Normally portals handle their own events all the way down to the root.
                // So we should be able to stop now. However, we don't know if this portal
                // was part of *our* root.
                var grandNode = node.return;

                while (grandNode !== null) {
                  var grandTag = grandNode.tag;

                  if (grandTag === HostRoot || grandTag === HostPortal) {
                    var grandContainer = grandNode.stateNode.containerInfo;

                    if (isMatchingRootContainer(grandContainer, targetContainerNode)) {
                      // This is the rootContainer we're looking for and we found it as
                      // a parent of the Portal. That means we can ignore it because the
                      // Portal will bubble through to us.
                      return;
                    }
                  }

                  grandNode = grandNode.return;
                }
              } // Now we need to find it's corresponding host fiber in the other
              // tree. To do this we can use getClosestInstanceFromNode, but we
              // need to validate that the fiber is a host instance, otherwise
              // we need to traverse up through the DOM till we find the correct
              // node that is from the other tree.


              while (container !== null) {
                var parentNode = getClosestInstanceFromNode(container);

                if (parentNode === null) {
                  return;
                }

                var parentTag = parentNode.tag;

                if (parentTag === HostComponent || parentTag === HostText) {
                  node = ancestorInst = parentNode;
                  continue mainLoop;
                }

                container = container.parentNode;
              }
            }

            node = node.return;
          }
        }
      }

      batchedUpdates(function () {
        return dispatchEventsForPlugins(domEventName, eventSystemFlags, nativeEvent,
  ancestorInst);
      });
    }

    function createDispatchListener(instance, listener, currentTarget) {
      return {
        instance: instance,
        listener: listener,
        currentTarget: currentTarget
      };
    }

    function accumulateSinglePhaseListeners(targetFiber, reactName, nativeEventType,
  inCapturePhase, accumulateTargetOnly, nativeEvent) {
      var captureName = reactName !== null ? reactName + 'Capture' : null;
      var reactEventName = inCapturePhase ? captureName : reactName;
      var listeners = [];
      var instance = targetFiber;
      var lastHostComponent = null; // Accumulate all instances and listeners via the
  target -> root path.

      while (instance !== null) {
        var _instance2 = instance,
            stateNode = _instance2.stateNode,
            tag = _instance2.tag; // Handle listeners that are on HostComponents (i.e.
  <div>)

        if (tag === HostComponent && stateNode !== null) {
          lastHostComponent = stateNode; // createEventHandle listeners


          if (reactEventName !== null) {
            var listener = getListener(instance, reactEventName);

            if (listener != null) {
              listeners.push(createDispatchListener(instance, listener,
  lastHostComponent));
            }
          }
        } // If we are only accumulating events for the target, then we don't
        // continue to propagate through the React fiber tree to find other
        // listeners.


        if (accumulateTargetOnly) {
          break;
        } // If we are processing the onBeforeBlur event, then we need to take

        instance = instance.return;
      }

      return listeners;
    } // We should only use this function for:
    // - BeforeInputEventPlugin
    // - ChangeEventPlugin
    // - SelectEventPlugin
    // This is because we only process these plugins
    // in the bubble phase, so we need to accumulate two
    // phase event listeners (via emulation).

    function accumulateTwoPhaseListeners(targetFiber, reactName) {
      var captureName = reactName + 'Capture';
      var listeners = [];
      var instance = targetFiber; // Accumulate all instances and listeners via the target
  -> root path.

      while (instance !== null) {
        var _instance3 = instance,
            stateNode = _instance3.stateNode,
            tag = _instance3.tag; // Handle listeners that are on HostComponents (i.e.
  <div>)

        if (tag === HostComponent && stateNode !== null) {
          var currentTarget = stateNode;
          var captureListener = getListener(instance, captureName);

          if (captureListener != null) {
            listeners.unshift(createDispatchListener(instance, captureListener,
  currentTarget));
          }

          var bubbleListener = getListener(instance, reactName);

          if (bubbleListener != null) {
            listeners.push(createDispatchListener(instance, bubbleListener,
  currentTarget));
          }
        }

        instance = instance.return;
      }

      return listeners;
    }

    function getParent(inst) {
      if (inst === null) {
        return null;
      }

      do {
        inst = inst.return; // TODO: If this is a HostRoot we might want to bail out.
        // That is depending on if we want nested subtrees (layers) to bubble
        // events to their parent. We could also go through parentNode on the
        // host node but that wouldn't work for React Native and doesn't let us
        // do the portal feature.
      } while (inst && inst.tag !== HostComponent);

      if (inst) {
        return inst;
      }

      return null;
    }
    /**
     * Return the lowest common ancestor of A and B, or null if they are in
     * different trees.
     */


    function getLowestCommonAncestor(instA, instB) {
      var nodeA = instA;
      var nodeB = instB;
      var depthA = 0;

      for (var tempA = nodeA; tempA; tempA = getParent(tempA)) {
        depthA++;
      }

      var depthB = 0;

      for (var tempB = nodeB; tempB; tempB = getParent(tempB)) {
        depthB++;
      } // If A is deeper, crawl up.


      while (depthA - depthB > 0) {
        nodeA = getParent(nodeA);
        depthA--;
      } // If B is deeper, crawl up.


      while (depthB - depthA > 0) {
        nodeB = getParent(nodeB);
        depthB--;
      } // Walk in lockstep until we find a match.


      var depth = depthA;

      while (depth--) {
        if (nodeA === nodeB || nodeB !== null && nodeA === nodeB.alternate) {
          return nodeA;
        }

        nodeA = getParent(nodeA);
        nodeB = getParent(nodeB);
      }

      return null;
    }

    function accumulateEnterLeaveListenersForEvent(dispatchQueue, event, target, common,
  inCapturePhase) {
      var registrationName = event._reactName;
      var listeners = [];
      var instance = target;

      while (instance !== null) {
        if (instance === common) {
          break;
        }

        var _instance4 = instance,
            alternate = _instance4.alternate,
            stateNode = _instance4.stateNode,
            tag = _instance4.tag;

        if (alternate !== null && alternate === common) {
          break;
        }

        if (tag === HostComponent && stateNode !== null) {
          var currentTarget = stateNode;

          if (inCapturePhase) {
            var captureListener = getListener(instance, registrationName);

            if (captureListener != null) {
              listeners.unshift(createDispatchListener(instance, captureListener,
  currentTarget));
            }
          } else if (!inCapturePhase) {
            var bubbleListener = getListener(instance, registrationName);

            if (bubbleListener != null) {
              listeners.push(createDispatchListener(instance, bubbleListener,
  currentTarget));
            }
          }
        }

        instance = instance.return;
      }

      if (listeners.length !== 0) {
        dispatchQueue.push({
          event: event,
          listeners: listeners
        });
      }
    } // We should only use this function for:
```

```
  // - EnterLeaveEventPlugin
  // This is because we only process this plugin
  // in the bubble phase, so we need to accumulate two
  // phase event listeners.


  function accumulateEnterLeaveTwoPhaseListeners(dispatchQueue, leaveEvent, enterEvent,
from, to) {
    var common = from && to ? getLowestCommonAncestor(from, to) : null;

    if (from !== null) {
      accumulateEnterLeaveListenersForEvent(dispatchQueue, leaveEvent, from, common,
false);
    }

    if (to !== null && enterEvent !== null) {
      accumulateEnterLeaveListenersForEvent(dispatchQueue, enterEvent, to, common, true);
    }
  }
  function getListenerSetKey(domEventName, capture) {
    return domEventName + "__" + (capture ? 'capture' : 'bubble');
  }

  var didWarnInvalidHydration = false;
  var DANGEROUSLY_SET_INNER_HTML = 'dangerouslySetInnerHTML';
  var SUPPRESS_CONTENT_EDITABLE_WARNING = 'suppressContentEditableWarning';
  var SUPPRESS_HYDRATION_WARNING = 'suppressHydrationWarning';
  var AUTOFOCUS = 'autoFocus';
  var CHILDREN = 'children';
  var STYLE = 'style';
  var HTML$1 = '__html';
  var warnedUnknownTags;
  var validatePropertiesInDevelopment;
  var warnForPropDifference;
  var warnForExtraAttributes;
  var warnForInvalidEventListener;
  var canDiffStyleForHydrationWarning;
  var normalizeHTML;

  {
    warnedUnknownTags = {
      // There are working polyfills for <dialog>. Let people use it.
      dialog: true,
      // Electron ships a custom <webview> tag to display external web content in
      // an isolated frame and process.
      // This tag is not present in non Electron environments such as JSDom which
      // is often used for testing purposes.
      // @see https://electronjs.org/docs/api/webview-tag
      webview: true
    };

    validatePropertiesInDevelopment = function (type, props) {
      validateProperties(type, props);
      validateProperties$1(type, props);
      validateProperties$2(type, props, {
        registrationNameDependencies: registrationNameDependencies,
        possibleRegistrationNames: possibleRegistrationNames
      });
    }; // IE 11 parses & normalizes the style attribute as opposed to other
    // browsers. It adds spaces and sorts the properties in some
    // non-alphabetical order. Handling that would require sorting CSS
    // properties in the client & server versions or applying
    // `expectedStyle` to a temporary DOM node to read its `style` attribute
    // normalized. Since it only affects IE, we're skipping style warnings
    // in that browser completely in favor of doing all that work.
    // See https://github.com/facebook/react/issues/11807


    canDiffStyleForHydrationWarning = canUseDOM && !document.documentMode;
    warnForPropDifference = function (propName, serverValue, clientValue) {
      if (didWarnInvalidHydration) {
        return;
      }

      var normalizedClientValue = normalizeMarkupForTextOrAttribute(clientValue);
      var normalizedServerValue = normalizeMarkupForTextOrAttribute(serverValue);

      if (normalizedServerValue === normalizedClientValue) {
        return;
      }

      didWarnInvalidHydration = true;

      error('Prop `%s` did not match. Server: %s Client: %s', propName,
JSON.stringify(normalizedServerValue), JSON.stringify(normalizedClientValue));
    };

    warnForExtraAttributes = function (attributeNames) {
      if (didWarnInvalidHydration) {
        return;
      }

      didWarnInvalidHydration = true;
      var names = [];
      attributeNames.forEach(function (name) {
        names.push(name);
      });

      error('Extra attributes from the server: %s', names);
    };

    warnForInvalidEventListener = function (registrationName, listener) {
      if (listener === false) {
        error('Expected `%s` listener to be a function, instead got `false`.\n\n' + 'If
you used to conditionally omit it with %s={condition && value}, ' + 'pass %s={condition ?
value : undefined} instead.', registrationName, registrationName, registrationName);
      } else {
        error('Expected `%s` listener to be a function, instead got a value of `%s`
type.', registrationName, typeof listener);
      }
    }; // Parse the HTML and read it back to normalize the HTML string so that it
    // can be used for comparison.


    normalizeHTML = function (parent, html) {
      // We could have created a separate document here to avoid
      // re-initializing custom elements if they exist. But this breaks
      // how <noscript> is being handled. So we use the same document.
      // See the discussion in https://github.com/facebook/react/pull/11157.
      var testElement = parent.namespaceURI === HTML_NAMESPACE ?
parent.ownerDocument.createElement(parent.tagName) :
parent.ownerDocument.createElementNS(parent.namespaceURI, parent.tagName);
      testElement.innerHTML = html;
      return testElement.innerHTML;
    };
  } // HTML parsing normalizes CR and CRLF to LF.
  // It also can turn \u0000 into \uFFFD inside attributes.
  // https://www.w3.org/TR/html5/single-page.html#preprocessing-the-input-stream
  // If we have a mismatch, it might be caused by that.
  // We will still patch up in this case but not fire the warning.


  var NORMALIZE_NEWLINES_REGEX = /\r\n?/g;
  var NORMALIZE_NULL_AND_REPLACEMENT_REGEX = /\u0000|\uFFFD/g;

  function normalizeMarkupForTextOrAttribute(markup) {
    {
      checkHtmlStringCoercion(markup);
    }

    var markupString = typeof markup === 'string' ? markup : '' + markup;
    return markupString.replace(NORMALIZE_NEWLINES_REGEX,
'\n').replace(NORMALIZE_NULL_AND_REPLACEMENT_REGEX, '');
  }

  function checkForUnmatchedText(serverText, clientText, isConcurrentMode, shouldWarnDev)
{
    var normalizedClientText = normalizeMarkupForTextOrAttribute(clientText);
    var normalizedServerText = normalizeMarkupForTextOrAttribute(serverText);

    if (normalizedServerText === normalizedClientText) {
      return;
    }

    if (shouldWarnDev) {
      {
        if (!didWarnInvalidHydration) {
          didWarnInvalidHydration = true;

          error('Text content did not match. Server: "%s" Client: "%s"',
normalizedServerText, normalizedClientText);
        }
      }
    }

    if (isConcurrentMode && enableClientRenderFallbackOnTextMismatch) {
      // In concurrent roots, we throw when there's a text mismatch and revert to
      // client rendering, up to the nearest Suspense boundary.
      throw new Error('Text content does not match server-rendered HTML.');
    }
  }

  function getOwnerDocumentFromRootContainer(rootContainerElement) {
    return rootContainerElement.nodeType === DOCUMENT_NODE ? rootContainerElement :
rootContainerElement.ownerDocument;
  }

  function noop() {}

  function trapClickOnNonInteractiveElement(node) {
    // Mobile Safari does not fire properly bubble click events on
    // non-interactive elements, which means delegated click listeners do not
    // fire. The workaround for this bug involves attaching an empty click
    // listener on the target node.
    // https://www.quirksmode.org/blog/archives/2010/09/click_event_del.html
    // Just set it using the onclick property so that we don't have to manage any
    // bookkeeping for it. Not sure if we need to clear it when the listener is
    // removed.
    // TODO: Only do this for the relevant Safaris maybe?
    node.onclick = noop;
  }

  function setInitialDOMProperties(tag, domElement, rootContainerElement, nextProps,
isCustomComponentTag) {
    for (var propKey in nextProps) {
      if (!nextProps.hasOwnProperty(propKey)) {
        continue;
      }

      var nextProp = nextProps[propKey];

      if (propKey === STYLE) {
        {
          if (nextProp) {
            // Freeze the next style object so that we can assume it won't be
            // mutated. We have already warned for this in the past.
            Object.freeze(nextProp);
          }
        } // Relies on `updateStylesByID` not mutating `styleUpdates`.


        setValueForStyles(domElement, nextProp);
      } else if (propKey === DANGEROUSLY_SET_INNER_HTML) {
        var nextHtml = nextProp ? nextProp[HTML$1] : undefined;

        if (nextHtml != null) {
          setInnerHTML(domElement, nextHtml);
        }
      } else if (propKey === CHILDREN) {
        if (typeof nextProp === 'string') {
          // Avoid setting initial textContent when the text is empty. In IE11 setting
          // textContent on a <textarea> will cause the placeholder to not
          // show within the <textarea> until it has been focused and blurred again.
          // https://github.com/facebook/react/issues/6731#issuecomment-254874553
          var canSetTextContent = tag !== 'textarea' || nextProp !== '';

          if (canSetTextContent) {
            setTextContent(domElement, nextProp);
          }
        } else if (typeof nextProp === 'number') {
          setTextContent(domElement, '' + nextProp);
        }
      } else if (propKey === SUPPRESS_CONTENT_EDITABLE_WARNING || propKey ===
SUPPRESS_HYDRATION_WARNING) ; else if (propKey === AUTOFOCUS) ; else if
(registrationNameDependencies.hasOwnProperty(propKey)) {
        if (nextProp != null) {
          if ( typeof nextProp !== 'function') {
            warnForInvalidEventListener(propKey, nextProp);
          }

          if (propKey === 'onScroll') {
            listenToNonDelegatedEvent('scroll', domElement);
          }
        }
      } else if (nextProp != null) {
        setValueForProperty(domElement, propKey, nextProp, isCustomComponentTag);
      }
    }
  }

  function updateDOMProperties(domElement, updatePayload, wasCustomComponentTag,
isCustomComponentTag) {
    // TODO: Handle wasCustomComponentTag
    for (var i = 0; i < updatePayload.length; i += 2) {
      var propKey = updatePayload[i];
      var propValue = updatePayload[i + 1];

      if (propKey === STYLE) {
        setValueForStyles(domElement, propValue);
      } else if (propKey === DANGEROUSLY_SET_INNER_HTML) {
        setInnerHTML(domElement, propValue);
      } else if (propKey === CHILDREN) {
        setTextContent(domElement, propValue);
      } else {
        setValueForProperty(domElement, propKey, propValue, isCustomComponentTag);
      }
    }
  }

  function createElement(type, props, rootContainerElement, parentNamespace) {
    var isCustomComponentTag; // We create tags in the namespace of their parent
container, except HTML
    // tags get no namespace.

    var ownerDocument = getOwnerDocumentFromRootContainer(rootContainerElement);
    var domElement;
    var namespaceURI = parentNamespace;

    if (namespaceURI === HTML_NAMESPACE) {
      namespaceURI = getIntrinsicNamespace(type);
    }

    if (namespaceURI === HTML_NAMESPACE) {
      {
        isCustomComponentTag = isCustomComponent(type, props); // Should this check be
gated by parent namespace? Not sure we want to
        // allow <SVG> or <mATH>.

        if (!isCustomComponentTag && type !== type.toLowerCase()) {
          error('<%s /> is using incorrect casing. ' + 'Use PascalCase for React
components, ' + 'or lowercase for HTML elements.', type);
        }
      }

      if (type === 'script') {
        // Create the script via .innerHTML so its "parser-inserted" flag is
        // set to true and it does not execute
        var div = ownerDocument.createElement('div');

        div.innerHTML = '<script><' + '/script>'; // eslint-disable-line
        // This is guaranteed to yield a script element.

        var firstChild = div.firstChild;
        domElement = div.removeChild(firstChild);
      } else if (typeof props.is === 'string') {
        // $FlowIssue `createElement` should be updated for Web Components
        domElement = ownerDocument.createElement(type, {
          is: props.is
        });
      } else {
        // Separate else branch instead of using `props.is || undefined` above because of
a Firefox bug.
        // See discussion in https://github.com/facebook/react/pull/6896
        // and discussion in https://bugzilla.mozilla.org/show_bug.cgi?id=1276240
        domElement = ownerDocument.createElement(type); // Normally attributes are
assigned in `setInitialDOMProperties`, however the `multiple` and `size`
        // attributes on `select`s needs to be added before `option`s are inserted.
        // This prevents:
        // - a bug where the `select` does not scroll to the correct option because
singular
        //  `select` elements automatically pick the first item #13222
        // - a bug where the `select` set the first item as selected despite the `size`
attribute #14239
        // See https://github.com/facebook/react/issues/13222
        // and https://github.com/facebook/react/issues/14239

        if (type === 'select') {
          var node = domElement;

          if (props.multiple) {
            node.multiple = true;
          } else if (props.size) {
            // Setting a size greater than 1 causes a select to behave like
`multiple=true`, where
            // it is possible that no option is selected.
            //
            // This is only necessary when a select in "single selection mode".
            node.size = props.size;
          }
        }
      }
    } else {
      domElement = ownerDocument.createElementNS(namespaceURI, type);
    }
    {
      if (namespaceURI === HTML_NAMESPACE) {
        if (!isCustomComponentTag && Object.prototype.toString.call(domElement) ===
'[object HTMLUnknownElement]' && !hasOwnProperty.call(warnedUnknownTags, type)) {
          warnedUnknownTags[type] = true;

          error('The tag <%s> is unrecognized in this browser. ' + 'If you meant to
render a React component, start its name with ' + 'an uppercase letter.', type);
        }
      }
    }

    return domElement;
  }
  function createTextNode(text, rootContainerElement) {
    return getOwnerDocumentFromRootContainer(rootContainerElement).createTextNode(text);
  }
  function setInitialProperties(domElement, tag, rawProps, rootContainerElement) {
    var isCustomComponentTag = isCustomComponent(tag, rawProps);

    {
      validatePropertiesInDevelopment(tag, rawProps);
    } // TODO: Make sure that we check isMounted before firing any of these events.


    var props;

    switch (tag) {
      case 'dialog':
        listenToNonDelegatedEvent('cancel', domElement);
        listenToNonDelegatedEvent('close', domElement);
        props = rawProps;
        break;

      case 'iframe':
      case 'object':
      case 'embed':
        // We listen to this event in case to ensure emulated bubble
        // listeners still fire for the load event.
        listenToNonDelegatedEvent('load', domElement);
        props = rawProps;
        break;

      case 'video':
      case 'audio':
        // We listen to these events in case to ensure emulated bubble
        // listeners still fire for all the media events.
        for (var i = 0; i < mediaEventTypes.length; i++) {
          listenToNonDelegatedEvent(mediaEventTypes[i], domElement);
        }

        props = rawProps;
        break;

      case 'source':
        // We listen to this event in case to ensure emulated bubble
        // listeners still fire for the error event.
        listenToNonDelegatedEvent('error', domElement);
        props = rawProps;
        break;

      case 'img':
      case 'image':
      case 'link':
        // We listen to these events in case to ensure emulated bubble
        // listeners still fire for error and load events.
        listenToNonDelegatedEvent('error', domElement);
        listenToNonDelegatedEvent('load', domElement);
        props = rawProps;
        break;

      case 'details':
        // We listen to this event in case to ensure emulated bubble
        // listeners still fire for the toggle event.
        listenToNonDelegatedEvent('toggle', domElement);
        props = rawProps;
        break;

      case 'input':
        initWrapperState(domElement, rawProps);
        props = getHostProps(domElement, rawProps); // We listen to this event in case to
ensure emulated bubble
        // listeners still fire for the invalid event.

        listenToNonDelegatedEvent('invalid', domElement);
        break;

      case 'option':
        validateProps(domElement, rawProps);
        props = rawProps;
        break;

      case 'select':
        initWrapperState$1(domElement, rawProps);
        props = getHostProps$1(domElement, rawProps); // We listen to this event in case
to ensure emulated bubble
        // listeners still fire for the invalid event.

        listenToNonDelegatedEvent('invalid', domElement);
        break;

      case 'textarea':
        initWrapperState$2(domElement, rawProps);
        props = getHostProps$2(domElement, rawProps); // We listen to this event in case
to ensure emulated bubble
        // listeners still fire for the invalid event.

        listenToNonDelegatedEvent('invalid', domElement);
        break;

      default:
        props = rawProps;
    }

    assertValidProps(tag, props);
    setInitialDOMProperties(tag, domElement, rootContainerElement, props,
isCustomComponentTag);

    switch (tag) {
      case 'input':
        // TODO: Make sure we check if this is still unmounted or do any clean
        // up necessary since we never stop tracking anymore.
        track(domElement);
        postMountWrapper(domElement, rawProps, false);
        break;

      case 'textarea':
        // TODO: Make sure we check if this is still unmounted or do any clean
        // up necessary since we never stop tracking anymore.
        track(domElement);
        postMountWrapper$3(domElement);
        break;

      case 'option':
        postMountWrapper$1(domElement, rawProps);
        break;
      case 'select':
        postMountWrapper$2(domElement, rawProps);
        break;

      default:
        if (typeof props.onClick === 'function') {
          // TODO: This cast may not be sound for SVG, MathML or custom elements.
          trapClickOnNonInteractiveElement(domElement);
        }

        break;
    }
  } // Calculate the diff between the two objects.

  function diffProperties(domElement, tag, lastRawProps, nextRawProps,
rootContainerElement) {
    {
      validatePropertiesInDevelopment(tag, nextRawProps);
    }

    var updatePayload = null;
    var lastProps;
    var nextProps;

    switch (tag) {
      case 'input':
        lastProps = getHostProps(domElement, lastRawProps);
        nextProps = getHostProps(domElement, nextRawProps);
        updatePayload = [];
        break;

      case 'select':
        lastProps = getHostProps$1(domElement, lastRawProps);
        nextProps = getHostProps$1(domElement, nextRawProps);
        updatePayload = [];
        break;

      case 'textarea':
        lastProps = getHostProps$2(domElement, lastRawProps);
        nextProps = getHostProps$2(domElement, nextRawProps);
        updatePayload = [];
        break;

      default:
        lastProps = lastRawProps;
        nextProps = nextRawProps;

        if (typeof lastProps.onClick !== 'function' && typeof nextProps.onClick ===
'function') {
          // TODO: This cast may not be sound for SVG, MathML or custom elements.
          trapClickOnNonInteractiveElement(domElement);
        }

        break;
    }

    assertValidProps(tag, nextProps);
    var propKey;
    var styleName;
    var styleUpdates = null;

    for (propKey in lastProps) {
      if (nextProps.hasOwnProperty(propKey) || !lastProps.hasOwnProperty(propKey) ||
lastProps[propKey] == null) {
        continue;
      }

      if (propKey === STYLE) {
        var lastStyle = lastProps[propKey];
        for (styleName in lastStyle) {
          if (lastStyle.hasOwnProperty(styleName)) {
            if (!styleUpdates) {
              styleUpdates = {};
            }

            styleUpdates[styleName] = '';
          }
        }
      } else if (propKey === DANGEROUSLY_SET_INNER_HTML || propKey === CHILDREN) ; else
if (propKey === SUPPRESS_CONTENT_EDITABLE_WARNING || propKey ===
SUPPRESS_HYDRATION_WARNING) ; else if (propKey === AUTOFOCUS) ; else if
(registrationNameDependencies.hasOwnProperty(propKey)) {
        // This is a special case. If any listener updates we need to ensure
        // that the "current" fiber pointer gets updated so we need a commit
        // to update this element.
        if (!updatePayload) {
          updatePayload = [];
        }
      } else {
        // For all other deleted properties we add it to the queue. We use
        // the allowed property list in the commit phase instead.
        (updatePayload = updatePayload || []).push(propKey, null);
      }
    }

    for (propKey in nextProps) {
      var nextProp = nextProps[propKey];
      var lastProp = lastProps != null ? lastProps[propKey] : undefined;

      if (!nextProps.hasOwnProperty(propKey) || nextProp === lastProp || nextProp == null
&& lastProp == null) {
        continue;
      }

      if (propKey === STYLE) {
        {
          if (nextProp) {
            // Freeze the next style object so that we can assume it won't be
            // mutated. We have already warned for this in the past.
            Object.freeze(nextProp);
          }
        }

        if (lastProp) {
          // Unset styles on `lastProp` but not on `nextProp`.
          for (styleName in lastProp) {
            if (lastProp.hasOwnProperty(styleName) && (!nextProp ||
!nextProp.hasOwnProperty(styleName))) {
              if (!styleUpdates) {
                styleUpdates = {};
              }

              styleUpdates[styleName] = '';
            }
          } // Update styles that changed since `lastProp`.


          for (styleName in nextProp) {
            if (nextProp.hasOwnProperty(styleName) && lastProp[styleName] !==
nextProp[styleName]) {
              if (!styleUpdates) {
                styleUpdates = {};
              }

              styleUpdates[styleName] = nextProp[styleName];
            }
          }
        } else {
          // Relies on `updateStylesByID` not mutating `styleUpdates`.
          if (!styleUpdates) {
            if (!updatePayload) {
              updatePayload = [];
            }

            updatePayload.push(propKey, styleUpdates);
          }

          styleUpdates = nextProp;
        }
      } else if (propKey === DANGEROUSLY_SET_INNER_HTML) {
        var nextHtml = nextProp ? nextProp[HTML$1] : undefined;
        var lastHtml = lastProp ? lastProp[HTML$1] : undefined;

        if (nextHtml != null) {
          if (lastHtml !== nextHtml) {
            (updatePayload = updatePayload || []).push(propKey, nextHtml);
          }
        }
      } else if (propKey === CHILDREN) {
        if (typeof nextProp === 'string' || typeof nextProp === 'number') {
          (updatePayload = updatePayload || []).push(propKey, '' + nextProp);
        }
      } else if (propKey === SUPPRESS_CONTENT_EDITABLE_WARNING || propKey ===
SUPPRESS_HYDRATION_WARNING) ; else if
(registrationNameDependencies.hasOwnProperty(propKey)) {
        if (nextProp != null) {
          // We eagerly listen to this even though we haven't committed yet.
          if ( typeof nextProp !== 'function') {
            warnForInvalidEventListener(propKey, nextProp);
          }

          if (propKey === 'onScroll') {
            listenToNonDelegatedEvent('scroll', domElement);
          }
        }

        if (!updatePayload && lastProp !== nextProp) {
          // This is a special case. If any listener updates we need to ensure
          // that the "current" props pointer gets updated so we need a commit
          // to update this element.
          updatePayload = [];
        }
      } else {
        // For any other property we always add it to the queue and then we
        // filter it out using the allowed property list during the commit.
        (updatePayload = updatePayload || []).push(propKey, nextProp);
      }
    }

    if (styleUpdates) {
      {
        validateShorthandPropertyCollisionInDev(styleUpdates, nextProps[STYLE]);
      }

      (updatePayload = updatePayload || []).push(STYLE, styleUpdates);
    }

    return updatePayload;
  } // Apply the diff.

  function updateProperties(domElement, updatePayload, tag, lastRawProps, nextRawProps) {
    // Update checked *before* name.
    // In the middle of an update, it is possible to have multiple checked.
    // When a checked radio tries to change name, browser makes another radio's checked
false.
    if (tag === 'input' && nextRawProps.type === 'radio' && nextRawProps.name != null) {
      updateChecked(domElement, nextRawProps);
    }

    var wasCustomComponentTag = isCustomComponent(tag, lastRawProps);
    var isCustomComponentTag = isCustomComponent(tag, nextRawProps); // Apply the diff.

    updateDOMProperties(domElement, updatePayload, wasCustomComponentTag,
isCustomComponentTag); // TODO: Ensure that an update gets scheduled if any of the
special props
    // changed.

    switch (tag) {
      case 'input':
        // Update the wrapper around inputs *after* updating props. This has to
        // happen after `updateDOMProperties`. Otherwise HTML5 input validations
        // raise warnings and prevent the new value from being assigned.
        updateWrapper(domElement, nextRawProps);
        break;

      case 'textarea':
        updateWrapper$1(domElement, nextRawProps);
        break;

      case 'select':
        // <select> value update needs to occur after <option> children
        // reconciliation
        postUpdateWrapper(domElement, nextRawProps);
        break;
    }
  }

  function getPossibleStandardName(propName) {
    {
      var lowerCasedName = propName.toLowerCase();

      if (!possibleStandardNames.hasOwnProperty(lowerCasedName)) {
        return null;
      }

      return possibleStandardNames[lowerCasedName] || null;
    }
  }

  function diffHydratedProperties(domElement, tag, rawProps, parentNamespace,
rootContainerElement, isConcurrentMode, shouldWarnDev) {
    var isCustomComponentTag;
    var extraAttributeNames;

    {
      isCustomComponentTag = isCustomComponent(tag, rawProps);
      validatePropertiesInDevelopment(tag, rawProps);
    } // TODO: Make sure that we check isMounted before firing any of these events.


    switch (tag) {
      case 'dialog':
        listenToNonDelegatedEvent('cancel', domElement);
        listenToNonDelegatedEvent('close', domElement);
        break;

      case 'iframe':
      case 'object':
      case 'embed':
        // We listen to this event in case to ensure emulated bubble
        // listeners still fire for the load event.
        listenToNonDelegatedEvent('load', domElement);
        break;

      case 'video':
```

```
      case 'audio':
        // We listen to these events in case to ensure emulated bubble
        // listeners still fire for all the media events.
        for (var i = 0; i < mediaEventTypes.length; i++) {
          listenToNonDelegatedEvent(mediaEventTypes[i], domElement);
        }

        break;

      case 'source':
        // We listen to this event in case to ensure emulated bubble
        // listeners still fire for the error event.
        listenToNonDelegatedEvent('error', domElement);
        break;

      case 'img':
      case 'image':
      case 'link':
        // We listen to these events in case to ensure emulated bubble
        // listeners still fire for error and load events.
        listenToNonDelegatedEvent('error', domElement);
        listenToNonDelegatedEvent('load', domElement);
        break;

      case 'details':
        // We listen to this event in case to ensure emulated bubble
        // listeners still fire for the toggle event.
        listenToNonDelegatedEvent('toggle', domElement);
        break;

      case 'input':
        initWrapperState(domElement, rawProps); // We listen to this event in case to
ensure emulated bubble
        // listeners still fire for the invalid event.

        listenToNonDelegatedEvent('invalid', domElement);
        break;

      case 'option':
        validateProps(domElement, rawProps);
        break;

      case 'select':
        initWrapperState$1(domElement, rawProps); // We listen to this event in case to
ensure emulated bubble
        // listeners still fire for the invalid event.

        listenToNonDelegatedEvent('invalid', domElement);
        break;

      case 'textarea':
        initWrapperState$2(domElement, rawProps); // We listen to this event in case to
ensure emulated bubble
        // listeners still fire for the invalid event.

        listenToNonDelegatedEvent('invalid', domElement);
        break;
    }

    assertValidProps(tag, rawProps);

    {
      extraAttributeNames = new Set();
      var attributes = domElement.attributes;

      for (var _i = 0; _i < attributes.length; _i++) {
        var name = attributes[_i].name.toLowerCase();

        switch (name) {
          // Controlled attributes are not validated
          // TODO: Only ignore them on controlled tags.
          case 'value':
            break;

          case 'checked':
            break;

          case 'selected':
            break;

          default:
            // Intentionally use the original name.
            // See discussion in https://github.com/facebook/react/pull/10676.
            extraAttributeNames.add(attributes[_i].name);
        }
      }
    }

    var updatePayload = null;

    for (var propKey in rawProps) {
      if (!rawProps.hasOwnProperty(propKey)) {
        continue;
      }

      var nextProp = rawProps[propKey];

      if (propKey === CHILDREN) {
        // For text content children we compare against textContent. This
        // might match additional HTML that is hidden when we read it using
        // textContent. E.g. "foo" will match "f<span>oo</span>" but that still
        // satisfies our requirement. Our requirement is not to produce perfect
        // HTML and attributes. Ideally we should preserve structure but it's
        // ok not to if the visible content is still enough to indicate what
        // even listeners these nodes might be wired up to.
        // TODO: Warn if there is more than a single textNode as a child.
        // TODO: Should we use domElement.firstChild.nodeValue to compare?
        if (typeof nextProp === 'string') {
          if (domElement.textContent !== nextProp) {
            if (rawProps[SUPPRESS_HYDRATION_WARNING] !== true) {
              checkForUnmatchedText(domElement.textContent, nextProp, isConcurrentMode,
shouldWarnDev);
            }

            updatePayload = [CHILDREN, nextProp];
          }
        } else if (typeof nextProp === 'number') {
          if (domElement.textContent !== '' + nextProp) {
            if (rawProps[SUPPRESS_HYDRATION_WARNING] !== true) {
              checkForUnmatchedText(domElement.textContent, nextProp, isConcurrentMode,
shouldWarnDev);
            }

            updatePayload = [CHILDREN, '' + nextProp];
          }
        }
      } else if (registrationNameDependencies.hasOwnProperty(propKey)) {
        if (nextProp != null) {
          if ( typeof nextProp !== 'function') {
            warnForInvalidEventListener(propKey, nextProp);
          }

          if (propKey === 'onScroll') {
            listenToNonDelegatedEvent('scroll', domElement);
          }
        }
      } else if (shouldWarnDev && true && // Convince Flow we've calculated it (it's DEV-
only in this method.)
```

```
      typeof isCustomComponentTag === 'boolean') {
        // Validate that the properties correspond to their expected values.
        var serverValue = void 0;
        var propertyInfo = isCustomComponentTag && enableCustomElementPropertySupport ?
null : getPropertyInfo(propKey);

        if (rawProps[SUPPRESS_HYDRATION_WARNING] === true) ; else if (propKey ===
SUPPRESS_CONTENT_EDITABLE_WARNING || propKey === SUPPRESS_HYDRATION_WARNING || //
Controlled attributes are not validated
        // TODO: Only ignore them on controlled tags.
        propKey === 'value' || propKey === 'checked' || propKey === 'selected') ; else if
(propKey === DANGEROUSLY_SET_INNER_HTML) {
          var serverHTML = domElement.innerHTML;
          var nextHtml = nextProp ? nextProp[HTML$1] : undefined;

          if (nextHtml != null) {
            var expectedHTML = normalizeHTML(domElement, nextHtml);

            if (expectedHTML !== serverHTML) {
              warnForPropDifference(propKey, serverHTML, expectedHTML);
            }
          }
        } else if (propKey === STYLE) {
          // $FlowFixMe - Should be inferred as not undefined.
          extraAttributeNames.delete(propKey);

          if (canDiffStyleForHydrationWarning) {
            var expectedStyle = createDangerousStringForStyles(nextProp);
            serverValue = domElement.getAttribute('style');

            if (expectedStyle !== serverValue) {
              warnForPropDifference(propKey, serverValue, expectedStyle);
            }
          }
        } else if (isCustomComponentTag && !enableCustomElementPropertySupport) {
          // $FlowFixMe - Should be inferred as not undefined.
          extraAttributeNames.delete(propKey.toLowerCase());
          serverValue = getValueForAttribute(domElement, propKey, nextProp);

          if (nextProp !== serverValue) {
            warnForPropDifference(propKey, serverValue, nextProp);
          }
        } else if (!shouldIgnoreAttribute(propKey, propertyInfo, isCustomComponentTag) &&
!shouldRemoveAttribute(propKey, nextProp, propertyInfo, isCustomComponentTag)) {
          var isMismatchDueToBadCasing = false;

          if (propertyInfo !== null) {
            // $FlowFixMe - Should be inferred as not undefined.
            extraAttributeNames.delete(propertyInfo.attributeName);
            serverValue = getValueForProperty(domElement, propKey, nextProp,
propertyInfo);
          } else {
            var ownNamespace = parentNamespace;

            if (ownNamespace === HTML_NAMESPACE) {
              ownNamespace = getIntrinsicNamespace(tag);
            }

            if (ownNamespace === HTML_NAMESPACE) {
              // $FlowFixMe - Should be inferred as not undefined.
              extraAttributeNames.delete(propKey.toLowerCase());
            } else {
              var standardName = getPossibleStandardName(propKey);

              if (standardName !== null && standardName !== propKey) {
                // If an SVG prop is supplied with bad casing, it will
                // be successfully parsed from HTML, but will produce a mismatch
                // (and would be incorrectly rendered on the client).
                // However, we already warn about bad casing elsewhere.
                // So we'll skip the misleading extra mismatch warning in this case.
                isMismatchDueToBadCasing = true; // $FlowFixMe - Should be inferred as
not undefined.

                extraAttributeNames.delete(standardName);
              } // $FlowFixMe - Should be inferred as not undefined.


              extraAttributeNames.delete(propKey);
            }

            serverValue = getValueForAttribute(domElement, propKey, nextProp);
          }

          var dontWarnCustomElement = enableCustomElementPropertySupport  ;

          if (!dontWarnCustomElement && nextProp !== serverValue &&
!isMismatchDueToBadCasing) {
            warnForPropDifference(propKey, serverValue, nextProp);
          }
        }
      }
    }

    {
      if (shouldWarnDev) {
        if ( // $FlowFixMe - Should be inferred as not undefined.
        extraAttributeNames.size > 0 && rawProps[SUPPRESS_HYDRATION_WARNING] !== true) {
          // $FlowFixMe - Should be inferred as not undefined.
          warnForExtraAttributes(extraAttributeNames);
        }
      }
    }

    switch (tag) {
      case 'input':
        // TODO: Make sure we check if this is still unmounted or do any clean
        // up necessary since we never stop tracking anymore.
        track(domElement);
        postMountWrapper(domElement, rawProps, true);
        break;

      case 'textarea':
        // TODO: Make sure we check if this is still unmounted or do any clean
        // up necessary since we never stop tracking anymore.
        track(domElement);
        postMountWrapper$3(domElement);
        break;

      case 'select':
      case 'option':
        // For input and textarea we current always set the value property at
        // post mount to force it to diverge from attributes. However, for
        // option and select we don't quite do the same thing and select
        // is not resilient to the DOM state changing so we don't do that here.
        // TODO: Consider not doing this for input and textarea.
        break;

      default:
        if (typeof rawProps.onClick === 'function') {
          // TODO: This cast may not be sound for SVG, MathML or custom elements.
          trapClickOnNonInteractiveElement(domElement);
        }

        break;
    }

    return updatePayload;
  }
  function diffHydratedText(textNode, text, isConcurrentMode) {
    var isDifferent = textNode.nodeValue !== text;
    return isDifferent;
  }
  function warnForDeletedHydratableElement(parentNode, child) {
    {
      if (didWarnInvalidHydration) {
        return;
      }

      didWarnInvalidHydration = true;

      error('Did not expect server HTML to contain a <%s> in <%s>.',
child.nodeName.toLowerCase(), parentNode.nodeName.toLowerCase());
    }
  }
  function warnForDeletedHydratableText(parentNode, child) {
    {
      if (didWarnInvalidHydration) {
        return;
      }

      didWarnInvalidHydration = true;

      error('Did not expect server HTML to contain the text node "%s" in <%s>.',
child.nodeValue, parentNode.nodeName.toLowerCase());
    }
  }
  function warnForInsertedHydratedElement(parentNode, tag, props) {
    {
      if (didWarnInvalidHydration) {
        return;
      }

      didWarnInvalidHydration = true;

      error('Expected server HTML to contain a matching <%s> in <%s>.', tag,
parentNode.nodeName.toLowerCase());
    }
  }
  function warnForInsertedHydratedText(parentNode, text) {
    {
      if (text === '') {
        // We expect to insert empty text nodes since they're not represented in
        // the HTML.
        // TODO: Remove this special case if we can just avoid inserting empty
        // text nodes.
        return;
      }

      if (didWarnInvalidHydration) {
        return;
      }

      didWarnInvalidHydration = true;

      error('Expected server HTML to contain a matching text node for "%s" in <%s>.',
text, parentNode.nodeName.toLowerCase());
    }
  }
  function restoreControlledState$3(domElement, tag, props) {
    switch (tag) {
      case 'input':
        restoreControlledState(domElement, props);
        return;

      case 'textarea':
        restoreControlledState$2(domElement, props);
        return;
      case 'select':
        restoreControlledState$1(domElement, props);
        return;
    }
  }

  var validateDOMNesting = function () {};

  var updatedAncestorInfo = function () {};

  {
    // This validation code was written based on the HTML5 parsing spec:
    // https://html.spec.whatwg.org/multipage/syntax.html#has-an-element-in-scope
    //
    // Note: this does not catch all invalid nesting, nor does it try to (as it's
    // not clear what practical benefit doing so provides); instead, we warn only
    // for cases where the parser will give a parse tree differing from what React
    // intended. For example, <b><div></div></b> is invalid but we don't warn
    // because it still parses correctly; we do warn for other cases like nested
    // <p> tags where the beginning of the second element implicitly closes the
    // first, causing a confusing mess.
    // https://html.spec.whatwg.org/multipage/syntax.html#special
    var specialTags = ['address', 'applet', 'area', 'article', 'aside', 'base',
'basefont', 'bgsound', 'blockquote', 'body', 'br', 'button', 'caption', 'center', 'col',
'colgroup', 'dd', 'details', 'dir', 'div', 'dl', 'dt', 'embed', 'fieldset', 'figcaption',
'figure', 'footer', 'form', 'frame', 'frameset', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6',
'head', 'header', 'hgroup', 'hr', 'html', 'iframe', 'img', 'input', 'isindex', 'li',
'link', 'listing', 'main', 'marquee', 'menu', 'menuitem', 'meta', 'nav', 'noembed',
'noframes', 'noscript', 'object', 'ol', 'p', 'param', 'plaintext', 'pre', 'script',
'section', 'select', 'source', 'style', 'summary', 'table', 'tbody', 'td', 'template',
'textarea', 'tfoot', 'th', 'thead', 'title', 'tr', 'track', 'ul', 'wbr', 'xmp']; //
https://html.spec.whatwg.org/multipage/syntax.html#has-an-element-in-scope

    var inScopeTags = ['applet', 'caption', 'html', 'table', 'td', 'th', 'marquee',
'object', 'template', // https://html.spec.whatwg.org/multipage/syntax.html#html-
integration-point
    // TODO: Distinguish by namespace here -- for <title>, including it here
    // errs on the side of fewer warnings
    'foreignObject', 'desc', 'title']; //
https://html.spec.whatwg.org/multipage/syntax.html#has-an-element-in-button-scope

    var buttonScopeTags = inScopeTags.concat(['button']); //
https://html.spec.whatwg.org/multipage/syntax.html#generate-implied-end-tags

    var impliedEndTags = ['dd', 'dt', 'li', 'option', 'optgroup', 'p', 'rp', 'rt'];
    var emptyAncestorInfo = {
      current: null,
      formTag: null,
      aTagInScope: null,
      buttonTagInScope: null,
      nobrTagInScope: null,
      pTagInButtonScope: null,
      listItemTagAutoclosing: null,
      dlItemTagAutoclosing: null
    };

    updatedAncestorInfo = function (oldInfo, tag) {
      var ancestorInfo = assign({}, oldInfo || emptyAncestorInfo);

      var info = {
        tag: tag
      };

      if (inScopeTags.indexOf(tag) !== -1) {
        ancestorInfo.aTagInScope = null;
        ancestorInfo.buttonTagInScope = null;
        ancestorInfo.nobrTagInScope = null;
      }
      if (buttonScopeTags.indexOf(tag) !== -1) {
        ancestorInfo.pTagInButtonScope = null;
      } // See rules for 'li', 'dd', 'dt' start tags in
      // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-inbody


      if (specialTags.indexOf(tag) !== -1 && tag !== 'address' && tag !== 'div' && tag
!== 'p') {
        ancestorInfo.listItemTagAutoclosing = null;
        ancestorInfo.dlItemTagAutoclosing = null;
      }

      ancestorInfo.current = info;

      if (tag === 'form') {
        ancestorInfo.formTag = info;
      }

      if (tag === 'a') {
        ancestorInfo.aTagInScope = info;
      }

      if (tag === 'button') {
        ancestorInfo.buttonTagInScope = info;
      }

      if (tag === 'nobr') {
        ancestorInfo.nobrTagInScope = info;
      }

      if (tag === 'p') {
        ancestorInfo.pTagInButtonScope = info;
      }

      if (tag === 'li') {
        ancestorInfo.listItemTagAutoclosing = info;
      }

      if (tag === 'dd' || tag === 'dt') {
        ancestorInfo.dlItemTagAutoclosing = info;
      }

      return ancestorInfo;
    };
    /**
     * Returns whether
     */


    var isTagValidWithParent = function (tag, parentTag) {
      // First, let's check if we're in an unusual parsing mode...
      switch (parentTag) {
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-inselect
        case 'select':
          return tag === 'option' || tag === 'optgroup' || tag === '#text';

        case 'optgroup':
          return tag === 'option' || tag === '#text';
        // Strictly speaking, seeing an <option> doesn't mean we're in a <select>
        // but

        case 'option':
          return tag === '#text';
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-intd
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-incaption
        // No special behavior since these rules fall back to "in body" mode for
        // all except special table nodes which cause bad parsing behavior anyway.
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-intr
        case 'tr':
          return tag === 'th' || tag === 'td' || tag === 'style' || tag === 'script' ||
tag === 'template';
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-intbody

        case 'tbody':
        case 'thead':
        case 'tfoot':
          return tag === 'tr' || tag === 'style' || tag === 'script' || tag ===
'template';
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-incolgroup

        case 'colgroup':
          return tag === 'col' || tag === 'template';
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-intable

        case 'table':
          return tag === 'caption' || tag === 'colgroup' || tag === 'tbody' || tag ===
'tfoot' || tag === 'thead' || tag === 'style' || tag === 'script' || tag === 'template';
        // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-inhead

        case 'head':
          return tag === 'base' || tag === 'basefont' || tag === 'bgsound' || tag ===
'link' || tag === 'meta' || tag === 'title' || tag === 'noscript' || tag === 'noframes'
|| tag === 'style' || tag === 'script' || tag === 'template';
        // https://html.spec.whatwg.org/multipage/semantics.html#the-html-element

        case 'html':
          return tag === 'head' || tag === 'body' || tag === 'frameset';

        case 'frameset':
          return tag === 'frame';

        case '#document':
          return tag === 'html';
      } // Probably in the "in body" parsing mode, so we outlaw only tag combos
      // where the parsing rules cause implicit opens or closes to be added.
      // https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-inbody


      switch (tag) {
        case 'h1':
        case 'h2':
        case 'h3':
        case 'h4':
        case 'h5':
        case 'h6':
          return parentTag !== 'h1' && parentTag !== 'h2' && parentTag !== 'h3' &&
parentTag !== 'h4' && parentTag !== 'h5' && parentTag !== 'h6';

        case 'rp':
        case 'rt':
          return impliedEndTags.indexOf(parentTag) === -1;

        case 'body':
        case 'caption':
        case 'col':
        case 'colgroup':
        case 'frameset':
        case 'frame':
        case 'head':
        case 'html':
        case 'tbody':
        case 'td':
        case 'tfoot':
        case 'th':
        case 'thead':
        case 'tr':
```

```
        // These tags are only valid with a few parents that have special child
        // parsing rules -- if we're down here, then none of those matched and
        // so we allow it only if we don't know what the parent is, as all other
        // cases are invalid.
        return parentTag == null;
    }

    return true;
  };
  /**
   * Returns whether
   */


  var findInvalidAncestorForTag = function (tag, ancestorInfo) {
    switch (tag) {
      case 'address':
      case 'article':
      case 'aside':
      case 'blockquote':
      case 'center':
      case 'details':
      case 'dialog':
      case 'dir':
      case 'div':
      case 'dl':
      case 'fieldset':
      case 'figcaption':
      case 'figure':
      case 'footer':
      case 'header':
      case 'hgroup':
      case 'main':
      case 'menu':
      case 'nav':
      case 'ol':
      case 'p':
      case 'section':
      case 'summary':
      case 'ul':
      case 'pre':
      case 'listing':
      case 'table':
      case 'hr':
      case 'xmp':
      case 'h1':
      case 'h2':
      case 'h3':
      case 'h4':
      case 'h5':
      case 'h6':
        return ancestorInfo.pTagInButtonScope;

      case 'form':
        return ancestorInfo.formTag || ancestorInfo.pTagInButtonScope;

      case 'li':
        return ancestorInfo.listItemTagAutoclosing;

      case 'dd':
      case 'dt':
        return ancestorInfo.dlItemTagAutoclosing;

      case 'button':
        return ancestorInfo.buttonTagInScope;

      case 'a':
        // Spec says something about storing a list of markers, but it sounds
        // equivalent to this check.
```

```
          return ancestorInfo.aTagInScope;

        case 'nobr':
          return ancestorInfo.nobrTagInScope;
      }

      return null;
    };

    var didWarn$1 = {};

    validateDOMNesting = function (childTag, childText, ancestorInfo) {
      ancestorInfo = ancestorInfo || emptyAncestorInfo;
      var parentInfo = ancestorInfo.current;
      var parentTag = parentInfo && parentInfo.tag;

      if (childText != null) {
        if (childTag != null) {
          error('validateDOMNesting: when childText is passed, childTag should be null');
        }

        childTag = '#text';
      }

      var invalidParent = isTagValidWithParent(childTag, parentTag) ? null : parentInfo;
      var invalidAncestor = invalidParent ? null : findInvalidAncestorForTag(childTag,
ancestorInfo);
      var invalidParentOrAncestor = invalidParent || invalidAncestor;

      if (!invalidParentOrAncestor) {
        return;
      }

      var ancestorTag = invalidParentOrAncestor.tag;
      var warnKey = !!invalidParent + '|' + childTag + '|' + ancestorTag;

      if (didWarn$1[warnKey]) {
        return;
      }

      didWarn$1[warnKey] = true;
      var tagDisplayName = childTag;
      var whitespaceInfo = '';

      if (childTag === '#text') {
        if (/\S/.test(childText)) {
          tagDisplayName = 'Text nodes';
        } else {
          tagDisplayName = 'Whitespace text nodes';
          whitespaceInfo = " Make sure you don't have any extra whitespace between tags
on " + 'each line of your source code.';
        }
      } else {
        tagDisplayName = '<' + childTag + '>';
      }

      if (invalidParent) {
        var info = '';

        if (ancestorTag === 'table' && childTag === 'tr') {
          info += ' Add a <tbody>, <thead> or <tfoot> to your code to match the DOM tree
generated by ' + 'the browser.';
        }

        error('validateDOMNesting(...): %s cannot appear as a child of <%s>.%s%s',
tagDisplayName, ancestorTag, whitespaceInfo, info);
      } else {
        error('validateDOMNesting(...): %s cannot appear as a descendant of ' + '<%s>.',
tagDisplayName, ancestorTag);
```

```
      }
    };
  }

  var SUPPRESS_HYDRATION_WARNING$1 = 'suppressHydrationWarning';
  var SUSPENSE_START_DATA = '$';
  var SUSPENSE_END_DATA = '/$';
  var SUSPENSE_PENDING_START_DATA = '$?';
  var SUSPENSE_FALLBACK_START_DATA = '$!';
  var STYLE$1 = 'style';
  var eventsEnabled = null;
  var selectionInformation = null;
  function getRootHostContext(rootContainerInstance) {
    var type;
    var namespace;
    var nodeType = rootContainerInstance.nodeType;

    switch (nodeType) {
      case DOCUMENT_NODE:
      case DOCUMENT_FRAGMENT_NODE:
        {
          type = nodeType === DOCUMENT_NODE ? '#document' : '#fragment';
          var root = rootContainerInstance.documentElement;
          namespace = root ? root.namespaceURI : getChildNamespace(null, '');
          break;
        }

      default:
        {
          var container = nodeType === COMMENT_NODE ? rootContainerInstance.parentNode :
rootContainerInstance;
          var ownNamespace = container.namespaceURI || null;
          type = container.tagName;
          namespace = getChildNamespace(ownNamespace, type);
          break;
        }
    }

    {
      var validatedTag = type.toLowerCase();
      var ancestorInfo = updatedAncestorInfo(null, validatedTag);
      return {
        namespace: namespace,
        ancestorInfo: ancestorInfo
      };
    }
  }
  function getChildHostContext(parentHostContext, type, rootContainerInstance) {
    {
      var parentHostContextDev = parentHostContext;
      var namespace = getChildNamespace(parentHostContextDev.namespace, type);
      var ancestorInfo = updatedAncestorInfo(parentHostContextDev.ancestorInfo, type);
      return {
        namespace: namespace,
        ancestorInfo: ancestorInfo
      };
    }
  }
  function getPublicInstance(instance) {
    return instance;
  }
  function prepareForCommit(containerInfo) {
    eventsEnabled = isEnabled();
    selectionInformation = getSelectionInformation();
    var activeInstance = null;

    setEnabled(false);
    return activeInstance;
  }
  function resetAfterCommit(containerInfo) {
    restoreSelection(selectionInformation);
    setEnabled(eventsEnabled);
    eventsEnabled = null;
    selectionInformation = null;
  }
  function createInstance(type, props, rootContainerInstance, hostContext,
internalInstanceHandle) {
    var parentNamespace;

    {
      // TODO: take namespace into account when validating.
      var hostContextDev = hostContext;
      validateDOMNesting(type, null, hostContextDev.ancestorInfo);

      if (typeof props.children === 'string' || typeof props.children === 'number') {
        var string = '' + props.children;
        var ownAncestorInfo = updatedAncestorInfo(hostContextDev.ancestorInfo, type);
        validateDOMNesting(null, string, ownAncestorInfo);
      }

      parentNamespace = hostContextDev.namespace;
    }

    var domElement = createElement(type, props, rootContainerInstance, parentNamespace);
    precacheFiberNode(internalInstanceHandle, domElement);
    updateFiberProps(domElement, props);
    return domElement;
  }
  function appendInitialChild(parentInstance, child) {
    parentInstance.appendChild(child);
  }
  function finalizeInitialChildren(domElement, type, props, rootContainerInstance,
hostContext) {
    setInitialProperties(domElement, type, props, rootContainerInstance);

    switch (type) {
      case 'button':
      case 'input':
      case 'select':
      case 'textarea':
        return !!props.autoFocus;

      case 'img':
        return true;

      default:
        return false;
    }
  }
  function prepareUpdate(domElement, type, oldProps, newProps, rootContainerInstance,
hostContext) {
    {
      var hostContextDev = hostContext;

      if (typeof newProps.children !== typeof oldProps.children && (typeof
newProps.children === 'string' || typeof newProps.children === 'number')) {
        var string = '' + newProps.children;
        var ownAncestorInfo = updatedAncestorInfo(hostContextDev.ancestorInfo, type);
        validateDOMNesting(null, string, ownAncestorInfo);
      }
    }

    return diffProperties(domElement, type, oldProps, newProps);
  }
  function shouldSetTextContent(type, props) {
    return type === 'textarea' || type === 'noscript' || typeof props.children ===
'string' || typeof props.children === 'number' || typeof props.dangerouslySetInnerHTML
=== 'object' && props.dangerouslySetInnerHTML !== null &&
props.dangerouslySetInnerHTML.__html != null;
  }
  function createTextInstance(text, rootContainerInstance, hostContext,
internalInstanceHandle) {
    {
      var hostContextDev = hostContext;
      validateDOMNesting(null, text, hostContextDev.ancestorInfo);
    }

    var textNode = createTextNode(text, rootContainerInstance);
    precacheFiberNode(internalInstanceHandle, textNode);
    return textNode;
  }
  function getCurrentEventPriority() {
    var currentEvent = window.event;

    if (currentEvent === undefined) {
      return DefaultEventPriority;
    }

    return getEventPriority(currentEvent.type);
  }
  // if a component just imports ReactDOM (e.g. for findDOMNode).
  // Some environments might not have setTimeout or clearTimeout.

  var scheduleTimeout = typeof setTimeout === 'function' ? setTimeout : undefined;
  var cancelTimeout = typeof clearTimeout === 'function' ? clearTimeout : undefined;
  var noTimeout = -1;
  var localPromise = typeof Promise === 'function' ? Promise : undefined; // ------------
-------
  var scheduleMicrotask = typeof queueMicrotask === 'function' ? queueMicrotask : typeof
localPromise !== 'undefined' ? function (callback) {
    return localPromise.resolve(null).then(callback).catch(handleErrorInNextTick);
  } : scheduleTimeout; // TODO: Determine the best fallback here.

  function handleErrorInNextTick(error) {
    setTimeout(function () {
      throw error;
    });
  } // -------------------
  function commitMount(domElement, type, newProps, internalInstanceHandle) {
    // Despite the naming that might imply otherwise, this method only
    // fires if there is an `Update` effect scheduled during mounting.
    // This happens if `finalizeInitialChildren` returns `true` (which it
    // does to implement the `autoFocus` attribute on the client). But
    // there are also other cases when this might happen (such as patching
    // up text content during hydration mismatch). So we'll check this again.
    switch (type) {
      case 'button':
      case 'input':
      case 'select':
      case 'textarea':
        if (newProps.autoFocus) {
          domElement.focus();
        }

        return;

      case 'img':
        {
          if (newProps.src) {
            domElement.src = newProps.src;
          }

          return;
        }
    }
  }
  function commitUpdate(domElement, updatePayload, type, oldProps, newProps,
internalInstanceHandle) {
    // Apply the diff to the DOM node.
    updateProperties(domElement, updatePayload, type, oldProps, newProps); // Update the
props handle so that we know which props are the ones with
    // with current event handlers.

    updateFiberProps(domElement, newProps);
  }
  function resetTextContent(domElement) {
    setTextContent(domElement, '');
  }
  function commitTextUpdate(textInstance, oldText, newText) {
    textInstance.nodeValue = newText;
  }
  function appendChild(parentInstance, child) {
    parentInstance.appendChild(child);
  }
  function appendChildToContainer(container, child) {
    var parentNode;

    if (container.nodeType === COMMENT_NODE) {
      parentNode = container.parentNode;
      parentNode.insertBefore(child, container);
    } else {
      parentNode = container;
      parentNode.appendChild(child);
    } // This container might be used for a portal.
    // If something inside a portal is clicked, that click should bubble
    // through the React tree. However, on Mobile Safari the click would
    // never bubble through the *DOM* tree unless an ancestor with onclick
    // event exists. So we wouldn't see it and dispatch it.
    // This is why we ensure that non React root containers have inline onclick
    // defined.
    // https://github.com/facebook/react/issues/11918


    var reactRootContainer = container._reactRootContainer;

    if ((reactRootContainer === null || reactRootContainer === undefined) &&
parentNode.onclick === null) {
      // TODO: This cast may not be sound for SVG, MathML or custom elements.
      trapClickOnNonInteractiveElement(parentNode);
    }
  }
  function insertBefore(parentInstance, child, beforeChild) {
    parentInstance.insertBefore(child, beforeChild);
  }
  function insertInContainerBefore(container, child, beforeChild) {
    if (container.nodeType === COMMENT_NODE) {
      container.parentNode.insertBefore(child, beforeChild);
    } else {
      container.insertBefore(child, beforeChild);
    }
  }

  function removeChild(parentInstance, child) {
    parentInstance.removeChild(child);
  }
  function removeChildFromContainer(container, child) {
    if (container.nodeType === COMMENT_NODE) {
      container.parentNode.removeChild(child);
    } else {
      container.removeChild(child);
    }
  }
  function clearSuspenseBoundary(parentInstance, suspenseInstance) {
    var node = suspenseInstance; // Delete all nodes within this suspense boundary.
    // There might be nested nodes so we need to keep track of how
    // deep we are and only break out when we're back on top.
    var depth = 0;

    do {
      var nextNode = node.nextSibling;
      parentInstance.removeChild(node);

      if (nextNode && nextNode.nodeType === COMMENT_NODE) {
        var data = nextNode.data;

        if (data === SUSPENSE_END_DATA) {
          if (depth === 0) {
            parentInstance.removeChild(nextNode); // Retry if any event replaying was
blocked on this.

            retryIfBlockedOn(suspenseInstance);
            return;
          } else {
            depth--;
          }
        } else if (data === SUSPENSE_START_DATA || data === SUSPENSE_PENDING_START_DATA
|| data === SUSPENSE_FALLBACK_START_DATA) {
          depth++;
        }
      }

      node = nextNode;
    } while (node); // TODO: Warn, we didn't find the end comment boundary.
    // Retry if any event replaying was blocked on this.


    retryIfBlockedOn(suspenseInstance);
  }
  function clearSuspenseBoundaryFromContainer(container, suspenseInstance) {
    if (container.nodeType === COMMENT_NODE) {
      clearSuspenseBoundary(container.parentNode, suspenseInstance);
    } else if (container.nodeType === ELEMENT_NODE) {
      clearSuspenseBoundary(container, suspenseInstance);
    } // Retry if any event replaying was blocked on this.


    retryIfBlockedOn(container);
  }
  function hideInstance(instance) {
    // TODO: Does this work for all element types? What about MathML? Should we
    // pass host context to this method?
    instance = instance;
    var style = instance.style;

    if (typeof style.setProperty === 'function') {
      style.setProperty('display', 'none', 'important');
    } else {
      style.display = 'none';
    }
  }
  function hideTextInstance(textInstance) {
    textInstance.nodeValue = '';
  }
  function unhideInstance(instance, props) {
    instance = instance;
    var styleProp = props[STYLE$1];
    var display = styleProp !== undefined && styleProp !== null &&
styleProp.hasOwnProperty('display') ? styleProp.display : null;
    instance.style.display = dangerousStyleValue('display', display);
  }
  function unhideTextInstance(textInstance, text) {
    textInstance.nodeValue = text;
  }
  function clearContainer(container) {
    if (container.nodeType === ELEMENT_NODE) {
      container.textContent = '';
    } else if (container.nodeType === DOCUMENT_NODE) {
      if (container.documentElement) {
        container.removeChild(container.documentElement);
      }
    }
  } // -------------------
  function canHydrateInstance(instance, type, props) {
    if (instance.nodeType !== ELEMENT_NODE || type.toLowerCase() !==
instance.nodeName.toLowerCase()) {
      return null;
    } // This has now been refined to an element node.


    return instance;
  }
  function canHydrateTextInstance(instance, text) {
    if (text === '' || instance.nodeType !== TEXT_NODE) {
      // Empty strings are not parsed by HTML so there won't be a correct match here.
      return null;
    } // This has now been refined to a text node.


    return instance;
  }
  function canHydrateSuspenseInstance(instance) {
    if (instance.nodeType !== COMMENT_NODE) {
      // Empty strings are not parsed by HTML so there won't be a correct match here.
      return null;
    } // This has now been refined to a suspense node.


    return instance;
  }
  function isSuspenseInstancePending(instance) {
    return instance.data === SUSPENSE_PENDING_START_DATA;
  }
  function isSuspenseInstanceFallback(instance) {
    return instance.data === SUSPENSE_FALLBACK_START_DATA;
  }
  function getSuspenseInstanceFallbackErrorDetails(instance) {
    var dataset = instance.nextSibling && instance.nextSibling.dataset;
    var digest, message, stack;

    if (dataset) {
      digest = dataset.dgst;

      {
        message = dataset.msg;
        stack = dataset.stck;
      }
    }

    {
      return {
        message: message,
        digest: digest,
        stack: stack
      };
    } // let value = {message: undefined, hash: undefined};
    // const nextSibling = instance.nextSibling;
    // if (nextSibling) {
    //   const dataset = ((nextSibling: any): HTMLTemplateElement).dataset;
    //   value.message = dataset.msg;
    //   value.hash = dataset.hash;
    //   if (true) {
    //     value.stack = dataset.stack;
    //   }
    // }
    // return value;

  }
  function registerSuspenseInstanceRetry(instance, callback) {
    instance._reactRetry = callback;
  }

  function getNextHydratable(node) {
    // Skip non-hydratable nodes.
    for (; node != null; node = node.nextSibling) {
      var nodeType = node.nodeType;

      if (nodeType === ELEMENT_NODE || nodeType === TEXT_NODE) {
        break;
      }

      if (nodeType === COMMENT_NODE) {
        var nodeData = node.data;

        if (nodeData === SUSPENSE_START_DATA || nodeData === SUSPENSE_FALLBACK_START_DATA
|| nodeData === SUSPENSE_PENDING_START_DATA) {
          break;
        }

        if (nodeData === SUSPENSE_END_DATA) {
          return null;
        }
      }
    }

    return node;
  }

  function getNextHydratableSibling(instance) {
    return getNextHydratable(instance.nextSibling);
  }
  function getFirstHydratableChild(parentInstance) {
    return getNextHydratable(parentInstance.firstChild);
  }
  function getFirstHydratableChildWithinContainer(parentContainer) {
    return getNextHydratable(parentContainer.firstChild);
  }
  function getFirstHydratableChildWithinSuspenseInstance(parentInstance) {
    return getNextHydratable(parentInstance.nextSibling);
  }
  function hydrateInstance(instance, type, props, rootContainerInstance, hostContext,
internalInstanceHandle, shouldWarnDev) {
    precacheFiberNode(internalInstanceHandle, instance); // TODO: Possibly defer this
until the commit phase where all the events
    // get attached.

    updateFiberProps(instance, props);
    var parentNamespace;

    {
      var hostContextDev = hostContext;
      parentNamespace = hostContextDev.namespace;
    } // TODO: Temporary hack to check if we're in a concurrent root. We can delete
    // when the legacy root API is removed.


    var isConcurrentMode = (internalInstanceHandle.mode & ConcurrentMode) !== NoMode;
    return diffHydratedProperties(instance, type, props, parentNamespace,
rootContainerInstance, isConcurrentMode, shouldWarnDev);
  }
  function hydrateTextInstance(textInstance, text, internalInstanceHandle, shouldWarnDev)
{
    precacheFiberNode(internalInstanceHandle, textInstance); // TODO: Temporary hack to
```

```
check if we're in a concurrent root. We can delete
    // when the legacy root API is removed.

    var isConcurrentMode = (internalInstanceHandle.mode & ConcurrentMode) !== NoMode;
    return diffHydratedText(textInstance, text);
  }
  function hydrateSuspenseInstance(suspenseInstance, internalInstanceHandle) {
    precacheFiberNode(internalInstanceHandle, suspenseInstance);
  }
  function getNextHydratableInstanceAfterSuspenseInstance(suspenseInstance) {
    var node = suspenseInstance.nextSibling; // Skip past all nodes within this suspense
boundary.
    // There might be nested nodes so we need to keep track of how
    // deep we are and only break out when we're back on top.

    var depth = 0;

    while (node) {
      if (node.nodeType === COMMENT_NODE) {
        var data = node.data;

        if (data === SUSPENSE_END_DATA) {
          if (depth === 0) {
            return getNextHydratableSibling(node);
          } else {
            depth--;
          }
        } else if (data === SUSPENSE_START_DATA || data === SUSPENSE_FALLBACK_START_DATA
|| data === SUSPENSE_PENDING_START_DATA) {
          depth++;
        }
      }

      node = node.nextSibling;
    } // TODO: Warn, we didn't find the end comment boundary.


    return null;
  } // Returns the SuspenseInstance if this node is a direct child of a
  // SuspenseInstance. I.e. if its previous sibling is a Comment with
  // SUSPENSE_x_START_DATA. Otherwise, null.

  function getParentSuspenseInstance(targetInstance) {
    var node = targetInstance.previousSibling; // Skip past all nodes within this
suspense boundary.
    // There might be nested nodes so we need to keep track of how
    // deep we are and only break out when we're back on top.

    var depth = 0;

    while (node) {
      if (node.nodeType === COMMENT_NODE) {
        var data = node.data;

        if (data === SUSPENSE_START_DATA || data === SUSPENSE_FALLBACK_START_DATA || data
=== SUSPENSE_PENDING_START_DATA) {
          if (depth === 0) {
            return node;
          } else {
            depth--;
          }
        } else if (data === SUSPENSE_END_DATA) {
          depth++;
        }
      }

      node = node.previousSibling;
    }
```

```
      return null;
    }
    function commitHydratedContainer(container) {
      // Retry if any event replaying was blocked on this.
      retryIfBlockedOn(container);
    }
    function commitHydratedSuspenseInstance(suspenseInstance) {
      // Retry if any event replaying was blocked on this.
      retryIfBlockedOn(suspenseInstance);
    }
    function shouldDeleteUnhydratedTailInstances(parentType) {
      return parentType !== 'head' && parentType !== 'body';
    }
    function didNotMatchHydratedContainerTextInstance(parentContainer, textInstance, text,
  isConcurrentMode) {
      var shouldWarnDev = true;
      checkForUnmatchedText(textInstance.nodeValue, text, isConcurrentMode, shouldWarnDev);
    }
    function didNotMatchHydratedTextInstance(parentType, parentProps, parentInstance,
  textInstance, text, isConcurrentMode) {
      if (parentProps[SUPPRESS_HYDRATION_WARNING$1] !== true) {
        var shouldWarnDev = true;
        checkForUnmatchedText(textInstance.nodeValue, text, isConcurrentMode,
  shouldWarnDev);
      }
    }
    function didNotHydrateInstanceWithinContainer(parentContainer, instance) {
      {
        if (instance.nodeType === ELEMENT_NODE) {
          warnForDeletedHydratableElement(parentContainer, instance);
        } else if (instance.nodeType === COMMENT_NODE) ; else {
          warnForDeletedHydratableText(parentContainer, instance);
        }
      }
    }
    function didNotHydrateInstanceWithinSuspenseInstance(parentInstance, instance) {
      {
        // $FlowFixMe: Only Element or Document can be parent nodes.
        var parentNode = parentInstance.parentNode;

        if (parentNode !== null) {
          if (instance.nodeType === ELEMENT_NODE) {
            warnForDeletedHydratableElement(parentNode, instance);
          } else if (instance.nodeType === COMMENT_NODE) ; else {
            warnForDeletedHydratableText(parentNode, instance);
          }
        }
      }
    }
    function didNotHydrateInstance(parentType, parentProps, parentInstance, instance,
  isConcurrentMode) {
      {
        if (isConcurrentMode || parentProps[SUPPRESS_HYDRATION_WARNING$1] !== true) {
          if (instance.nodeType === ELEMENT_NODE) {
            warnForDeletedHydratableElement(parentInstance, instance);
          } else if (instance.nodeType === COMMENT_NODE) ; else {
            warnForDeletedHydratableText(parentInstance, instance);
          }
        }
      }
    }
    function didNotFindHydratableInstanceWithinContainer(parentContainer, type, props) {
      {
        warnForInsertedHydratedElement(parentContainer, type);
      }
    }
    function didNotFindHydratableTextInstanceWithinContainer(parentContainer, text) {
      {
        warnForInsertedHydratedText(parentContainer, text);
      }
    }
    function didNotFindHydratableInstanceWithinSuspenseInstance(parentInstance, type,
  props) {
      {
        // $FlowFixMe: Only Element or Document can be parent nodes.
        var parentNode = parentInstance.parentNode;
        if (parentNode !== null) warnForInsertedHydratedElement(parentNode, type);
      }
    }
    function didNotFindHydratableTextInstanceWithinSuspenseInstance(parentInstance, text) {
      {
        // $FlowFixMe: Only Element or Document can be parent nodes.
        var parentNode = parentInstance.parentNode;
        if (parentNode !== null) warnForInsertedHydratedText(parentNode, text);
      }
    }
    function didNotFindHydratableInstance(parentType, parentProps, parentInstance, type,
  props, isConcurrentMode) {
      {
        if (isConcurrentMode || parentProps[SUPPRESS_HYDRATION_WARNING$1] !== true) {
          warnForInsertedHydratedElement(parentInstance, type);
        }
      }
    }
    function didNotFindHydratableTextInstance(parentType, parentProps, parentInstance,
  text, isConcurrentMode) {
      {
        if (isConcurrentMode || parentProps[SUPPRESS_HYDRATION_WARNING$1] !== true) {
          warnForInsertedHydratedText(parentInstance, text);
        }
      }
    }
    function errorHydratingContainer(parentContainer) {
      {
        // TODO: This gets logged by onRecoverableError, too, so we should be
        // able to remove it.
        error('An error occurred during hydration. The server HTML was replaced with client
  content in <%s>.', parentContainer.nodeName.toLowerCase());
      }
    }
    function preparePortalMount(portalInstance) {
      listenToAllSupportedEvents(portalInstance);
    }

    var randomKey = Math.random().toString(36).slice(2);
    var internalInstanceKey = '__reactFiber$' + randomKey;
    var internalPropsKey = '__reactProps$' + randomKey;
    var internalContainerInstanceKey = '__reactContainer$' + randomKey;
    var internalEventHandlersKey = '__reactEvents$' + randomKey;
    var internalEventHandlerListenersKey = '__reactListeners$' + randomKey;
    var internalEventHandlesSetKey = '__reactHandles$' + randomKey;
    function detachDeletedInstance(node) {
      // TODO: This function is only called on host components. I don't think all of
      // these fields are relevant.
      delete node[internalInstanceKey];
      delete node[internalPropsKey];
      delete node[internalEventHandlersKey];
      delete node[internalEventHandlerListenersKey];
      delete node[internalEventHandlesSetKey];
    }
    function precacheFiberNode(hostInst, node) {
      node[internalInstanceKey] = hostInst;
    }
    function markContainerAsRoot(hostRoot, node) {
      node[internalContainerInstanceKey] = hostRoot;
    }
    function unmarkContainerAsRoot(node) {
      node[internalContainerInstanceKey] = null;
    }
    function isContainerMarkedAsRoot(node) {
      return !!node[internalContainerInstanceKey];
    } // Given a DOM node, return the closest HostComponent or HostText fiber ancestor.
    // If the target node is part of a hydrated or not yet rendered subtree, then
    // this may also return a SuspenseComponent or HostRoot to indicate that.
    // Conceptually the HostRoot fiber is a child of the Container node. So if you
    // pass the Container node as the targetNode, you will not actually get the
    // HostRoot back. To get to the HostRoot, you need to pass a child of it.
    // The same thing applies to Suspense boundaries.

    function getClosestInstanceFromNode(targetNode) {
      var targetInst = targetNode[internalInstanceKey];

      if (targetInst) {
        // Don't return HostRoot or SuspenseComponent here.
        return targetInst;
      } // If the direct event target isn't a React owned DOM node, we need to look
      // to see if one of its parents is a React owned DOM node.


      var parentNode = targetNode.parentNode;

      while (parentNode) {
        // We'll check if this is a container root that could include
        // React nodes in the future. We need to check this first because
        // if we're a child of a dehydrated container, we need to first
        // find that inner container before moving on to finding the parent
        // instance. Note that we don't check this field on  the targetNode
        // itself because the fibers are conceptually between the container
        // node and the first child. It isn't surrounding the container node.
        // If it's not a container, we check if it's an instance.
        targetInst = parentNode[internalContainerInstanceKey] ||
  parentNode[internalInstanceKey];

        if (targetInst) {
          // Since this wasn't the direct target of the event, we might have
          // stepped past dehydrated DOM nodes to get here. However they could
          // also have been non-React nodes. We need to answer which one.
          // If we the instance doesn't have any children, then there can't be
          // a nested suspense boundary within it. So we can use this as a fast
          // bailout. Most of the time, when people add non-React children to
          // the tree, it is using a ref to a child-less DOM node.
          // Normally we'd only need to check one of the fibers because if it
          // has ever gone from having children to deleting them or vice versa
          // it would have deleted the dehydrated boundary nested inside already.
          // However, since the HostRoot starts out with an alternate it might
          // have one on the alternate so we need to check in case this was a
          // root.
          var alternate = targetInst.alternate;

          if (targetInst.child !== null || alternate !== null && alternate.child !== null)
  {
            // Next we need to figure out if the node that skipped past is
            // nested within a dehydrated boundary and if so, which one.
            var suspenseInstance = getParentSuspenseInstance(targetNode);

            while (suspenseInstance !== null) {
              // We found a suspense instance. That means that we haven't
              // hydrated it yet. Even though we leave the comments in the
              // DOM after hydrating, and there are boundaries in the DOM
              // that could already be hydrated, we wouldn't have found them
              // through this pass since if the target is hydrated it would
              // have had an internalInstanceKey on it.
              // Let's get the fiber associated with the SuspenseComponent
              // as the deepest instance.
              var targetSuspenseInst = suspenseInstance[internalInstanceKey];

              if (targetSuspenseInst) {
```

```
              return targetSuspenseInst;
            } // If we don't find a Fiber on the comment, it might be because
            // we haven't gotten to hydrate it yet. There might still be a
            // parent boundary that hasn't above this one so we need to find
            // the outer most that is known.


            suspenseInstance = getParentSuspenseInstance(suspenseInstance); // If we
don't find one, then that should mean that the parent
            // host component also hasn't hydrated yet. We can return it
            // below since it will bail out on the isMounted check later.
          }
        }

        return targetInst;
      }

      targetNode = parentNode;
      parentNode = targetNode.parentNode;
    }

    return null;
  }
  /**
   * Given a DOM node, return the ReactDOMComponent or ReactDOMTextComponent
   * instance, or null if the node was not rendered by this React.
   */

  function getInstanceFromNode(node) {
    var inst = node[internalInstanceKey] || node[internalContainerInstanceKey];

    if (inst) {
      if (inst.tag === HostComponent || inst.tag === HostText || inst.tag ===
SuspenseComponent || inst.tag === HostRoot) {
        return inst;
      } else {
        return null;
      }
    }

    return null;
  }
  /**
   * Given a ReactDOMComponent or ReactDOMTextComponent, return the corresponding
   * DOM node.
   */

  function getNodeFromInstance(inst) {
    if (inst.tag === HostComponent || inst.tag === HostText) {
      // In Fiber this, is just the state node right now. We assume it will be
      // a host component or host text.
      return inst.stateNode;
    } // Without this first invariant, passing a non-DOM-component triggers the next
    // invariant for a missing parent, which is super confusing.


    throw new Error('getNodeFromInstance: Invalid argument.');
  }
  function getFiberCurrentPropsFromNode(node) {
    return node[internalPropsKey] || null;
  }
  function updateFiberProps(node, props) {
    node[internalPropsKey] = props;
  }
  function getEventListenerSet(node) {
    var elementListenerSet = node[internalEventHandlersKey];

    if (elementListenerSet === undefined) {
      elementListenerSet = node[internalEventHandlersKey] = new Set();
    }

    return elementListenerSet;
  }

  var loggedTypeFailures = {};
  var ReactDebugCurrentFrame$1 = ReactSharedInternals.ReactDebugCurrentFrame;

  function setCurrentlyValidatingElement(element) {
    {
      if (element) {
        var owner = element._owner;
        var stack = describeUnknownElementTypeFrameInDEV(element.type, element._source,
owner ? owner.type : null);
        ReactDebugCurrentFrame$1.setExtraStackFrame(stack);
      } else {
        ReactDebugCurrentFrame$1.setExtraStackFrame(null);
      }
    }
  }

  function checkPropTypes(typeSpecs, values, location, componentName, element) {
    {
      // $FlowFixMe This is okay but Flow doesn't know it.
      var has = Function.call.bind(hasOwnProperty);

      for (var typeSpecName in typeSpecs) {
        if (has(typeSpecs, typeSpecName)) {
          var error$1 = void 0; // Prop type validation may throw. In case they do, we
don't want to
          // fail the render phase where it didn't fail before. So we log it.
          // After these have been cleaned up, we'll let them throw.

          try {
            // This is intentionally an invariant that gets caught. It's the same
            // behavior as without this statement except with a better message.
            if (typeof typeSpecs[typeSpecName] !== 'function') {
              // eslint-disable-next-line react-internal/prod-error-codes
              var err = Error((componentName || 'React class') + ': ' + location + ' type
`' + typeSpecName + '` is invalid; ' + 'it must be a function, usually from the `prop-
types` package, but received `' + typeof typeSpecs[typeSpecName] + '`.' + 'This often
happens because of typos such as `PropTypes.function` instead of `PropTypes.func`.');
              err.name = 'Invariant Violation';
              throw err;
            }

            error$1 = typeSpecs[typeSpecName](values, typeSpecName, componentName,
location, null, 'SECRET_DO_NOT_PASS_THIS_OR_YOU_WILL_BE_FIRED');
          } catch (ex) {
            error$1 = ex;
          }

          if (error$1 && !(error$1 instanceof Error)) {
            setCurrentlyValidatingElement(element);

            error('%s: type specification of %s' + ' `%s` is invalid; the type checker '
+ 'function must return `null` or an `Error` but returned a %s. ' + 'You may have
forgotten to pass an argument to the type checker ' + 'creator (arrayOf, instanceOf,
objectOf, oneOf, oneOfType, and ' + 'shape all require an argument).', componentName ||
'React class', location, typeSpecName, typeof error$1);

            setCurrentlyValidatingElement(null);
          }

          if (error$1 instanceof Error && !(error$1.message in loggedTypeFailures)) {
            // Only monitor this failure once because there tends to be a lot of the
            // same error.
            loggedTypeFailures[error$1.message] = true;
            setCurrentlyValidatingElement(element);
            error('Failed %s type: %s', location, error$1.message);

            setCurrentlyValidatingElement(null);
          }
        }
      }
    }
  }

  var valueStack = [];
  var fiberStack;

  {
    fiberStack = [];
  }

  var index = -1;

  function createCursor(defaultValue) {
    return {
      current: defaultValue
    };
  }

  function pop(cursor, fiber) {
    if (index < 0) {
      {
        error('Unexpected pop.');
      }

      return;
    }

    {
      if (fiber !== fiberStack[index]) {
        error('Unexpected Fiber popped.');
      }
    }

    cursor.current = valueStack[index];
    valueStack[index] = null;

    {
      fiberStack[index] = null;
    }

    index--;
  }

  function push(cursor, value, fiber) {
    index++;
    valueStack[index] = cursor.current;

    {
      fiberStack[index] = fiber;
    }

    cursor.current = value;
  }

  var warnedAboutMissingGetChildContext;

  {
    warnedAboutMissingGetChildContext = {};
  }

  var emptyContextObject = {};
  {
    Object.freeze(emptyContextObject);
  } // A cursor to the current merged context object on the stack.


  var contextStackCursor = createCursor(emptyContextObject); // A cursor to a boolean
indicating whether the context has changed.

  var didPerformWorkStackCursor = createCursor(false); // Keep track of the previous
context object that was on the stack.
  // We use this to get access to the parent context after we have already
  // pushed the next context provider, and now need to merge their contexts.

  var previousContext = emptyContextObject;

  function getUnmaskedContext(workInProgress, Component, didPushOwnContextIfProvider) {
    {
      if (didPushOwnContextIfProvider && isContextProvider(Component)) {
        // If the fiber is a context provider itself, when we read its context
        // we may have already pushed its own child context on the stack. A context
        // provider should not "see" its own child context. Therefore we read the
        // previous (parent) context instead for a context provider.
        return previousContext;
      }

      return contextStackCursor.current;
    }
  }

  function cacheContext(workInProgress, unmaskedContext, maskedContext) {
    {
      var instance = workInProgress.stateNode;
      instance.__reactInternalMemoizedUnmaskedChildContext = unmaskedContext;
      instance.__reactInternalMemoizedMaskedChildContext = maskedContext;
    }
  }

  function getMaskedContext(workInProgress, unmaskedContext) {
    {
      var type = workInProgress.type;
      var contextTypes = type.contextTypes;

      if (!contextTypes) {
        return emptyContextObject;
      } // Avoid recreating masked context unless unmasked context has changed.
      // Failing to do this will result in unnecessary calls to
componentWillReceiveProps.
      // This may trigger infinite loops if componentWillReceiveProps calls setState.


      var instance = workInProgress.stateNode;

      if (instance && instance.__reactInternalMemoizedUnmaskedChildContext ===
unmaskedContext) {
        return instance.__reactInternalMemoizedMaskedChildContext;
      }

      var context = {};

      for (var key in contextTypes) {
        context[key] = unmaskedContext[key];
      }

      {
        var name = getComponentNameFromFiber(workInProgress) || 'Unknown';
        checkPropTypes(contextTypes, context, 'context', name);
      } // Cache unmasked context so we can avoid recreating masked context unless
necessary.
      // Context is created before the class component is instantiated so check for
```

```
instance.


      if (instance) {
        cacheContext(workInProgress, unmaskedContext, context);
      }

      return context;
    }
  }

  function hasContextChanged() {
    {
      return didPerformWorkStackCursor.current;
    }
  }

  function isContextProvider(type) {
    {
      var childContextTypes = type.childContextTypes;
      return childContextTypes !== null && childContextTypes !== undefined;
    }
  }

  function popContext(fiber) {
    {
      pop(didPerformWorkStackCursor, fiber);
      pop(contextStackCursor, fiber);
    }
  }

  function popTopLevelContextObject(fiber) {
    {
      pop(didPerformWorkStackCursor, fiber);
      pop(contextStackCursor, fiber);
    }
  }

  function pushTopLevelContextObject(fiber, context, didChange) {
    {
      if (contextStackCursor.current !== emptyContextObject) {
        throw new Error('Unexpected context found on stack. ' + 'This error is likely
caused by a bug in React. Please file an issue.');
      }

      push(contextStackCursor, context, fiber);
      push(didPerformWorkStackCursor, didChange, fiber);
    }
  }

  function processChildContext(fiber, type, parentContext) {
    {
      var instance = fiber.stateNode;
      var childContextTypes = type.childContextTypes; // TODO (bvaughn) Replace this
behavior with an invariant() in the future.
      // It has only been added in Fiber to match the (unintentional) behavior in Stack.

      if (typeof instance.getChildContext !== 'function') {
        {
          var componentName = getComponentNameFromFiber(fiber) || 'Unknown';

          if (!warnedAboutMissingGetChildContext[componentName]) {
            warnedAboutMissingGetChildContext[componentName] = true;

            error('%s.childContextTypes is specified but there is no getChildContext()
method ' + 'on the instance. You can either define getChildContext() on %s or remove ' +
'childContextTypes from it.', componentName, componentName);
          }
        }
        return parentContext;
      }

      var childContext = instance.getChildContext();

      for (var contextKey in childContext) {
        if (!(contextKey in childContextTypes)) {
          throw new Error((getComponentNameFromFiber(fiber) || 'Unknown') +
".getChildContext(): key \"" + contextKey + "\" is not defined in childContextTypes.");
        }
      }

      {
        var name = getComponentNameFromFiber(fiber) || 'Unknown';
        checkPropTypes(childContextTypes, childContext, 'child context', name);
      }

      return assign({}, parentContext, childContext);
    }
  }

  function pushContextProvider(workInProgress) {
    {
      var instance = workInProgress.stateNode; // We push the context as early as
possible to ensure stack integrity.
      // If the instance does not exist yet, we will push null at first,
      // and replace it on the stack later when invalidating the context.

      var memoizedMergedChildContext = instance &&
instance.__reactInternalMemoizedMergedChildContext || emptyContextObject; // Remember the
parent context so we can merge with it later.
      // Inherit the parent's did-perform-work value to avoid inadvertently blocking
updates.

      previousContext = contextStackCursor.current;
      push(contextStackCursor, memoizedMergedChildContext, workInProgress);
      push(didPerformWorkStackCursor, didPerformWorkStackCursor.current, workInProgress);
      return true;
    }
  }

  function invalidateContextProvider(workInProgress, type, didChange) {
    {
      var instance = workInProgress.stateNode;

      if (!instance) {
        throw new Error('Expected to have an instance by this point. ' + 'This error is
likely caused by a bug in React. Please file an issue.');
      }

      if (didChange) {
        // Merge parent and own context.
        // Skip this if we're not updating due to sCU.
        // This avoids unnecessarily recomputing memoized values.
        var mergedContext = processChildContext(workInProgress, type, previousContext);
        instance.__reactInternalMemoizedMergedChildContext = mergedContext; // Replace
the old (or empty) context with the new one.
        // It is important to unwind the context in the reverse order.

        pop(didPerformWorkStackCursor, workInProgress);
        pop(contextStackCursor, workInProgress); // Now push the new context and mark
that it has changed.

        push(contextStackCursor, mergedContext, workInProgress);
        push(didPerformWorkStackCursor, didChange, workInProgress);
      } else {
        pop(didPerformWorkStackCursor, workInProgress);
        push(didPerformWorkStackCursor, didChange, workInProgress);
      }
    }
  }

  function findCurrentUnmaskedContext(fiber) {
    {
      // Currently this is only used with renderSubtreeIntoContainer; not sure if it
      // makes sense elsewhere
      if (!isFiberMounted(fiber) || fiber.tag !== ClassComponent) {
        throw new Error('Expected subtree parent to be a mounted class component. ' +
'This error is likely caused by a bug in React. Please file an issue.');
      }

      var node = fiber;

      do {
        switch (node.tag) {
          case HostRoot:
            return node.stateNode.context;

          case ClassComponent:
            {
              var Component = node.type;

              if (isContextProvider(Component)) {
                return node.stateNode.__reactInternalMemoizedMergedChildContext;
              }

              break;
            }
        }

        node = node.return;
      } while (node !== null);

      throw new Error('Found unexpected detached subtree parent. ' + 'This error is
likely caused by a bug in React. Please file an issue.');
    }
  }

  var LegacyRoot = 0;
  var ConcurrentRoot = 1;

  var syncQueue = null;
  var includesLegacySyncCallbacks = false;
  var isFlushingSyncQueue = false;
  function scheduleSyncCallback(callback) {
    // Push this callback into an internal queue. We'll flush these either in
    // the next tick, or earlier if something calls `flushSyncCallbackQueue`.
    if (syncQueue === null) {
      syncQueue = [callback];
    } else {
      // Push onto existing queue. Don't need to schedule a callback because
      // we already scheduled one when we created the queue.
      syncQueue.push(callback);
    }
  }
  function scheduleLegacySyncCallback(callback) {
    includesLegacySyncCallbacks = true;
    scheduleSyncCallback(callback);
  }
  function flushSyncCallbacksOnlyInLegacyMode() {
    // Only flushes the queue if there's a legacy sync callback scheduled.
    // TODO: There's only a single type of callback: performSyncOnWorkOnRoot. So
    // it might make more sense for the queue to be a list of roots instead of a
    // list of generic callbacks. Then we can have two: one for legacy roots, one
    // for concurrent roots. And this method would only flush the legacy ones.
    if (includesLegacySyncCallbacks) {
      flushSyncCallbacks();
    }
  }
  function flushSyncCallbacks() {
    if (!isFlushingSyncQueue && syncQueue !== null) {
      // Prevent re-entrance.
      isFlushingSyncQueue = true;
      var i = 0;
      var previousUpdatePriority = getCurrentUpdatePriority();

      try {
        var isSync = true;
        var queue = syncQueue; // TODO: Is this necessary anymore? The only user code
that runs in this
        // queue is in the render or commit phases.

        setCurrentUpdatePriority(DiscreteEventPriority);

        for (; i < queue.length; i++) {
          var callback = queue[i];

          do {
            callback = callback(isSync);
          } while (callback !== null);
        }

        syncQueue = null;
        includesLegacySyncCallbacks = false;
      } catch (error) {
        // If something throws, leave the remaining callbacks on the queue.
        if (syncQueue !== null) {
          syncQueue = syncQueue.slice(i + 1);
        } // Resume flushing in the next tick


        scheduleCallback(ImmediatePriority, flushSyncCallbacks);
        throw error;
      } finally {
        setCurrentUpdatePriority(previousUpdatePriority);
        isFlushingSyncQueue = false;
      }
    }

    return null;
  }

  // TODO: Use the unified fiber stack module instead of this local one?
  // Intentionally not using it yet to derisk the initial implementation, because
  // the way we push/pop these values is a bit unusual. If there's a mistake, I'd
  // rather the ids be wrong than crash the whole reconciler.
  var forkStack = [];
  var forkStackIndex = 0;
  var treeForkProvider = null;
  var treeForkCount = 0;
  var idStack = [];
  var idStackIndex = 0;
  var treeContextProvider = null;
  var treeContextId = 1;
  var treeContextOverflow = '';
  function isForkedChild(workInProgress) {
    warnIfNotHydrating();
    return (workInProgress.flags & Forked) !== NoFlags;
  }
  function getForksAtLevel(workInProgress) {
    warnIfNotHydrating();
    return treeForkCount;
  }
  function getTreeId() {
    var overflow = treeContextOverflow;
    var idWithLeadingBit = treeContextId;
    var id = idWithLeadingBit & ~getLeadingBit(idWithLeadingBit);
    return id.toString(32) + overflow;
  }
  function pushTreeFork(workInProgress, totalChildren) {
    // This is called right after we reconcile an array (or iterator) of child
    // fibers, because that's the only place where we know how many children in
    // the whole set without doing extra work later, or storing addtional
    // information on the fiber.
    //
    // That's why this function is separate from pushTreeId — it's called during
    // the render phase of the fork parent, not the child, which is where we push
    // the other context values.
    //
    // In the Fizz implementation this is much simpler because the child is
    // rendered in the same callstack as the parent.
    //
    // It might be better to just add a `forks` field to the Fiber type. It would
    // make this module simpler.
    warnIfNotHydrating();
    forkStack[forkStackIndex++] = treeForkCount;
    forkStack[forkStackIndex++] = treeForkProvider;
    treeForkProvider = workInProgress;
    treeForkCount = totalChildren;
  }
  function pushTreeId(workInProgress, totalChildren, index) {
    warnIfNotHydrating();
    idStack[idStackIndex++] = treeContextId;
    idStack[idStackIndex++] = treeContextOverflow;
    idStack[idStackIndex++] = treeContextProvider;
    treeContextProvider = workInProgress;
    var baseIdWithLeadingBit = treeContextId;
    var baseOverflow = treeContextOverflow; // The leftmost 1 marks the end of the
sequence, non-inclusive. It's not part
    // of the id; we use it to account for leading 0s.

    var baseLength = getBitLength(baseIdWithLeadingBit) - 1;
    var baseId = baseIdWithLeadingBit & ~(1 << baseLength);
    var slot = index + 1;
    var length = getBitLength(totalChildren) + baseLength; // 30 is the max length we can
store without overflowing, taking into
    // consideration the leading 1 we use to mark the end of the sequence.

    if (length > 30) {
      // We overflowed the bitwise-safe range. Fall back to slower algorithm.
      // This branch assumes the length of the base id is greater than 5; it won't
      // work for smaller ids, because you need 5 bits per character.
      //
      // We encode the id in multiple steps: first the base id, then the
      // remaining digits.
      //
      // Each 5 bit sequence corresponds to a single base 32 character. So for
      // example, if the current id is 23 bits long, we can convert 20 of those
      // bits into a string of 4 characters, with 3 bits left over.
      //
      // First calculate how many bits in the base id represent a complete
      // sequence of characters.
      var numberOfOverflowBits = baseLength - baseLength % 5; // Then create a bitmask
that selects only those bits.

      var newOverflowBits = (1 << numberOfOverflowBits) - 1; // Select the bits, and
convert them to a base 32 string.

      var newOverflow = (baseId & newOverflowBits).toString(32); // Now we can remove
those bits from the base id.

      var restOfBaseId = baseId >> numberOfOverflowBits;
      var restOfBaseLength = baseLength - numberOfOverflowBits; // Finally, encode the
rest of the bits using the normal algorithm. Because
      // we made more room, this time it won't overflow.
      var restOfLength = getBitLength(totalChildren) + restOfBaseLength;
      var restOfNewBits = slot << restOfBaseLength;
      var id = restOfNewBits | restOfBaseId;
      var overflow = newOverflow + baseOverflow;
      treeContextId = 1 << restOfLength | id;
      treeContextOverflow = overflow;
    } else {
      // Normal path
      var newBits = slot << baseLength;

      var _id = newBits | baseId;

      var _overflow = baseOverflow;
      treeContextId = 1 << length | _id;
      treeContextOverflow = _overflow;
    }
  }
  function pushMaterializedTreeId(workInProgress) {
    warnIfNotHydrating(); // This component materialized an id. This will affect any ids
that appear
    // in its children.

    var returnFiber = workInProgress.return;

    if (returnFiber !== null) {
      var numberOfForks = 1;
      var slotIndex = 0;
      pushTreeFork(workInProgress, numberOfForks);
      pushTreeId(workInProgress, numberOfForks, slotIndex);
    }
  }

  function getBitLength(number) {
    return 32 - clz32(number);
  }

  function getLeadingBit(id) {
    return 1 << getBitLength(id) - 1;
  }

  function popTreeContext(workInProgress) {
    // Restore the previous values.
    // This is a bit more complicated than other context-like modules in Fiber
    // because the same Fiber may appear on the stack multiple times and for
    // different reasons. We have to keep popping until the work-in-progress is
    // no longer at the top of the stack.
    while (workInProgress === treeForkProvider) {
      treeForkProvider = forkStack[--forkStackIndex];
      forkStack[forkStackIndex] = null;
      treeForkCount = forkStack[--forkStackIndex];
      forkStack[forkStackIndex] = null;
    }

    while (workInProgress === treeContextProvider) {
      treeContextProvider = idStack[--idStackIndex];
      idStack[idStackIndex] = null;
      treeContextOverflow = idStack[--idStackIndex];
      idStack[idStackIndex] = null;
      treeContextId = idStack[--idStackIndex];
      idStack[idStackIndex] = null;
    }
  }
  function getSuspendedTreeContext() {
    warnIfNotHydrating();

    if (treeContextProvider !== null) {
      return {
        id: treeContextId,
        overflow: treeContextOverflow
      };
    } else {
      return null;
    }
  }
  function restoreSuspendedTreeContext(workInProgress, suspendedContext) {
    warnIfNotHydrating();
    idStack[idStackIndex++] = treeContextId;
    idStack[idStackIndex++] = treeContextOverflow;
    idStack[idStackIndex++] = treeContextProvider;
    treeContextId = suspendedContext.id;
    treeContextOverflow = suspendedContext.overflow;
    treeContextProvider = workInProgress;
  }

  function warnIfNotHydrating() {
    {
      if (!getIsHydrating()) {
        error('Expected to be hydrating. This is a bug in React. Please file ' + 'an
issue.');
      }
    }
  }

  // This may have been an insertion or a hydration.

  var hydrationParentFiber = null;
  var nextHydratableInstance = null;
  var isHydrating = false; // This flag allows for warning supression when we expect
there to be mismatches
  // due to earlier mismatches or a suspended fiber.

  var didSuspendOrErrorDEV = false; // Hydration errors that were thrown inside this
boundary

  var hydrationErrors = null;

  function warnIfHydrating() {
    {
      if (isHydrating) {
        error('We should not be hydrating here. This is a bug in React. Please file a
bug.');
      }
    }
  }

  function markDidThrowWhileHydratingDEV() {
    {
      didSuspendOrErrorDEV = true;
    }
  }
  function didSuspendOrErrorWhileHydratingDEV() {
    {
      return didSuspendOrErrorDEV;
    }
  }

  function enterHydrationState(fiber) {

    var parentInstance = fiber.stateNode.containerInfo;
    nextHydratableInstance = getFirstHydratableChildWithinContainer(parentInstance);
    hydrationParentFiber = fiber;
    isHydrating = true;
    hydrationErrors = null;
    didSuspendOrErrorDEV = false;
    return true;
  }
  function reenterHydrationStateFromDehydratedSuspenseInstance(fiber, suspenseInstance,
treeContext) {

    nextHydratableInstance =
getFirstHydratableChildWithinSuspenseInstance(suspenseInstance);
    hydrationParentFiber = fiber;
    isHydrating = true;
    hydrationErrors = null;
    didSuspendOrErrorDEV = false;

    if (treeContext !== null) {
      restoreSuspendedTreeContext(fiber, treeContext);
    }

    return true;
  }

  function warnUnhydratedInstance(returnFiber, instance) {
    {
      switch (returnFiber.tag) {
        case HostRoot:
          {
            didNotHydrateInstanceWithinContainer(returnFiber.stateNode.containerInfo,
instance);
            break;
          }

        case HostComponent:
          {
            var isConcurrentMode = (returnFiber.mode & ConcurrentMode) !== NoMode;
            didNotHydrateInstance(returnFiber.type, returnFiber.memoizedProps,
returnFiber.stateNode, instance, // TODO: Delete this argument when we remove the legacy
root API.
            isConcurrentMode);
            break;
          }

        case SuspenseComponent:
          {
            var suspenseState = returnFiber.memoizedState;
            if (suspenseState.dehydrated !== null)
didNotHydrateInstanceWithinSuspenseInstance(suspenseState.dehydrated, instance);
            break;
          }
      }
    }
  }

  function deleteHydratableInstance(returnFiber, instance) {
    warnUnhydratedInstance(returnFiber, instance);
    var childToDelete = createFiberFromHostInstanceForDeletion();
    childToDelete.stateNode = instance;
    childToDelete.return = returnFiber;
    var deletions = returnFiber.deletions;

    if (deletions === null) {
      returnFiber.deletions = [childToDelete];
      returnFiber.flags |= ChildDeletion;
    } else {
      deletions.push(childToDelete);
    }
  }

  function warnNonhydratedInstance(returnFiber, fiber) {
    {
      if (didSuspendOrErrorDEV) {
        // Inside a boundary that already suspended. We're currently rendering the
        // siblings of a suspended node. The mismatch may be due to the missing
        // data, so it's probably a false positive.
        return;
      }

      switch (returnFiber.tag) {
        case HostRoot:
          {
            var parentContainer = returnFiber.stateNode.containerInfo;

            switch (fiber.tag) {
              case HostComponent:
                var type = fiber.type;
                var props = fiber.pendingProps;
                didNotFindHydratableInstanceWithinContainer(parentContainer, type);
                break;

              case HostText:
                var text = fiber.pendingProps;
                didNotFindHydratableTextInstanceWithinContainer(parentContainer, text);
                break;
            }

            break;
          }

        case HostComponent:
          {
            var parentType = returnFiber.type;
            var parentProps = returnFiber.memoizedProps;
            var parentInstance = returnFiber.stateNode;

            switch (fiber.tag) {
              case HostComponent:
                {
                  var _type = fiber.type;
                  var _props = fiber.pendingProps;
                  var isConcurrentMode = (returnFiber.mode & ConcurrentMode) !== NoMode;
                  didNotFindHydratableInstance(parentType, parentProps, parentInstance,
 _type, _props, // TODO: Delete this argument when we remove the legacy root API.
                  isConcurrentMode);
                  break;
                }

              case HostText:
                {
                  var _text = fiber.pendingProps;

                  var _isConcurrentMode = (returnFiber.mode & ConcurrentMode) !== NoMode;

                  didNotFindHydratableTextInstance(parentType, parentProps,
 parentInstance, _text, // TODO: Delete this argument when we remove the legacy root API.
                  _isConcurrentMode);
                  break;
                }
            }

            break;
          }

        case SuspenseComponent:
          {
            var suspenseState = returnFiber.memoizedState;
            var _parentInstance = suspenseState.dehydrated;
            if (_parentInstance !== null) switch (fiber.tag) {
              case HostComponent:
                var _type2 = fiber.type;
                var _props2 = fiber.pendingProps;
                didNotFindHydratableInstanceWithinSuspenseInstance(_parentInstance,
 _type2);
                break;
              case HostText:
                var _text2 = fiber.pendingProps;
                didNotFindHydratableTextInstanceWithinSuspenseInstance(_parentInstance,
_text2);
                break;
            }
            break;
          }

        default:
          return;
      }
    }
  }

  function insertNonHydratedInstance(returnFiber, fiber) {
    fiber.flags = fiber.flags & ~Hydrating | Placement;
    warnNonhydratedInstance(returnFiber, fiber);
  }

  function tryHydrate(fiber, nextInstance) {
    switch (fiber.tag) {
      case HostComponent:
        {
          var type = fiber.type;
          var props = fiber.pendingProps;
          var instance = canHydrateInstance(nextInstance, type);

          if (instance !== null) {
            fiber.stateNode = instance;
            hydrationParentFiber = fiber;
            nextHydratableInstance = getFirstHydratableChild(instance);
            return true;
          }

          return false;
        }

      case HostText:
        {
          var text = fiber.pendingProps;
          var textInstance = canHydrateTextInstance(nextInstance, text);

          if (textInstance !== null) {
            fiber.stateNode = textInstance;
            hydrationParentFiber = fiber; // Text Instances don't have children so
there's nothing to hydrate.

            nextHydratableInstance = null;
            return true;
          }

          return false;
        }

      case SuspenseComponent:
        {
          var suspenseInstance = canHydrateSuspenseInstance(nextInstance);

          if (suspenseInstance !== null) {
            var suspenseState = {
              dehydrated: suspenseInstance,
              treeContext: getSuspendedTreeContext(),
              retryLane: OffscreenLane
            };
            fiber.memoizedState = suspenseState; // Store the dehydrated fragment as a
child fiber.
            // This simplifies the code for getHostSibling and deleting nodes,
            // since it doesn't have to consider all Suspense boundaries and
            // check if they're dehydrated ones or not.

            var dehydratedFragment = createFiberFromDehydratedFragment(suspenseInstance);
            dehydratedFragment.return = fiber;
            fiber.child = dehydratedFragment;
            hydrationParentFiber = fiber; // While a Suspense Instance does have
children, we won't step into
            // it during the first pass. Instead, we'll reenter it later.

            nextHydratableInstance = null;
            return true;
          }

          return false;
        }

      default:
        return false;
    }
  }

  function shouldClientRenderOnMismatch(fiber) {
    return (fiber.mode & ConcurrentMode) !== NoMode && (fiber.flags & DidCapture) ===
NoFlags;
  }

  function throwOnHydrationMismatch(fiber) {
    throw new Error('Hydration failed because the initial UI does not match what was ' +
'rendered on the server.');
  }

  function tryToClaimNextHydratableInstance(fiber) {
    if (!isHydrating) {
      return;
    }

    var nextInstance = nextHydratableInstance;

    if (!nextInstance) {
      if (shouldClientRenderOnMismatch(fiber)) {
        warnNonhydratedInstance(hydrationParentFiber, fiber);
        throwOnHydrationMismatch();
      } // Nothing to hydrate. Make it an insertion.


      insertNonHydratedInstance(hydrationParentFiber, fiber);
      isHydrating = false;
      hydrationParentFiber = fiber;
      return;
    }

    var firstAttemptedInstance = nextInstance;

    if (!tryHydrate(fiber, nextInstance)) {
      if (shouldClientRenderOnMismatch(fiber)) {
        warnNonhydratedInstance(hydrationParentFiber, fiber);
        throwOnHydrationMismatch();
      } // If we can't hydrate this instance let's try the next one.
      // We use this as a heuristic. It's based on intuition and not data so it
      // might be flawed or unnecessary.


      nextInstance = getNextHydratableSibling(firstAttemptedInstance);
      var prevHydrationParentFiber = hydrationParentFiber;

      if (!nextInstance || !tryHydrate(fiber, nextInstance)) {
        // Nothing to hydrate. Make it an insertion.
        insertNonHydratedInstance(hydrationParentFiber, fiber);
        isHydrating = false;
        hydrationParentFiber = fiber;
        return;
      } // We matched the next one, we'll now assume that the first one was
      // superfluous and we'll delete it. Since we can't eagerly delete it
      // we'll have to schedule a deletion. To do that, this node needs a dummy
      // fiber associated with it.


      deleteHydratableInstance(prevHydrationParentFiber, firstAttemptedInstance);
    }
  }

  function prepareToHydrateHostInstance(fiber, rootContainerInstance, hostContext) {

    var instance = fiber.stateNode;
    var shouldWarnIfMismatchDev = !didSuspendOrErrorDEV;
    var updatePayload = hydrateInstance(instance, fiber.type, fiber.memoizedProps,
rootContainerInstance, hostContext, fiber, shouldWarnIfMismatchDev); // TODO: Type this
specific to this type of component.

    fiber.updateQueue = updatePayload; // If the update payload indicates that there is a
change or if there
    // is a new ref we mark this as an update.

    if (updatePayload !== null) {
      return true;
    }

    return false;
  }

  function prepareToHydrateHostTextInstance(fiber) {

    var textInstance = fiber.stateNode;
    var textContent = fiber.memoizedProps;
    var shouldUpdate = hydrateTextInstance(textInstance, textContent, fiber);

    if (shouldUpdate) {
      // We assume that prepareToHydrateHostTextInstance is called in a context where the
      // hydration parent is the parent host component of this host text.
      var returnFiber = hydrationParentFiber;

      if (returnFiber !== null) {
        switch (returnFiber.tag) {
          case HostRoot:
            {
              var parentContainer = returnFiber.stateNode.containerInfo;
              var isConcurrentMode = (returnFiber.mode & ConcurrentMode) !== NoMode;
              didNotMatchHydratedContainerTextInstance(parentContainer, textInstance,
textContent, // TODO: Delete this argument when we remove the legacy root API.
              isConcurrentMode);
              break;
            }

          case HostComponent:
            {
              var parentType = returnFiber.type;
              var parentProps = returnFiber.memoizedProps;
              var parentInstance = returnFiber.stateNode;

              var _isConcurrentMode2 = (returnFiber.mode & ConcurrentMode) !== NoMode;

              didNotMatchHydratedTextInstance(parentType, parentProps, parentInstance,
textInstance, textContent, // TODO: Delete this argument when we remove the legacy root
API.
              _isConcurrentMode2);
              break;
            }
        }
      }
    }

    return shouldUpdate;
  }

  function prepareToHydrateHostSuspenseInstance(fiber) {

    var suspenseState = fiber.memoizedState;
    var suspenseInstance = suspenseState !== null ? suspenseState.dehydrated : null;

    if (!suspenseInstance) {
      throw new Error('Expected to have a hydrated suspense instance. ' + 'This error is
likely caused by a bug in React. Please file an issue.');
    }

    hydrateSuspenseInstance(suspenseInstance, fiber);
  }

  function skipPastDehydratedSuspenseInstance(fiber) {

    var suspenseState = fiber.memoizedState;
    var suspenseInstance = suspenseState !== null ? suspenseState.dehydrated : null;

    if (!suspenseInstance) {
      throw new Error('Expected to have a hydrated suspense instance. ' + 'This error is
likely caused by a bug in React. Please file an issue.');
    }

    return getNextHydratableInstanceAfterSuspenseInstance(suspenseInstance);
  }

  function popToNextHostParent(fiber) {
    var parent = fiber.return;

    while (parent !== null && parent.tag !== HostComponent && parent.tag !== HostRoot &&
parent.tag !== SuspenseComponent) {
      parent = parent.return;
    }

    hydrationParentFiber = parent;
  }

  function popHydrationState(fiber) {

    if (fiber !== hydrationParentFiber) {
      // We're deeper than the current hydration context, inside an inserted
      // tree.
      return false;
    }

    if (!isHydrating) {
      // If we're not currently hydrating but we're in a hydration context, then
      // we were an insertion and now need to pop up reenter hydration of our
      // siblings.
      popToNextHostParent(fiber);
      isHydrating = true;
      return false;
    } // If we have any remaining hydratable nodes, we need to delete them now.
    // We only do this deeper than head and body since they tend to have random
    // other nodes in them. We also ignore components with pure text content in
    // side of them. We also don't delete anything inside the root container.


    if (fiber.tag !== HostRoot && (fiber.tag !== HostComponent ||
shouldDeleteUnhydratedTailInstances(fiber.type) && !shouldSetTextContent(fiber.type,
fiber.memoizedProps))) {
      var nextInstance = nextHydratableInstance;
      if (nextInstance) {
        if (shouldClientRenderOnMismatch(fiber)) {
          warnIfUnhydratedTailNodes(fiber);
          throwOnHydrationMismatch();
        } else {
          while (nextInstance) {
            deleteHydratableInstance(fiber, nextInstance);
            nextInstance = getNextHydratableSibling(nextInstance);
          }
        }
      }
    }

    popToNextHostParent(fiber);

    if (fiber.tag === SuspenseComponent) {
      nextHydratableInstance = skipPastDehydratedSuspenseInstance(fiber);
    } else {
      nextHydratableInstance = hydrationParentFiber ?
getNextHydratableSibling(fiber.stateNode) : null;
    }

    return true;
  }

  function hasUnhydratedTailNodes() {
    return isHydrating && nextHydratableInstance !== null;
  }

  function warnIfUnhydratedTailNodes(fiber) {
    var nextInstance = nextHydratableInstance;

    while (nextInstance) {
      warnUnhydratedInstance(fiber, nextInstance);
      nextInstance = getNextHydratableSibling(nextInstance);
    }
  }

  function resetHydrationState() {

    hydrationParentFiber = null;
    nextHydratableInstance = null;
    isHydrating = false;
    didSuspendOrErrorDEV = false;
  }

  function upgradeHydrationErrorsToRecoverable() {
    if (hydrationErrors !== null) {
      // Successfully completed a forced client render. The errors that occurred
      // during the hydration attempt are now recovered. We will log them in
      // commit phase, once the entire tree has finished.
      queueRecoverableErrors(hydrationErrors);
      hydrationErrors = null;
    }
  }

  function getIsHydrating() {
    return isHydrating;
  }

  function queueHydrationError(error) {
    if (hydrationErrors === null) {
      hydrationErrors = [error];
    } else {
      hydrationErrors.push(error);
    }
  }
  var ReactCurrentBatchConfig$1 = ReactSharedInternals.ReactCurrentBatchConfig;
  var NoTransition = null;
  function requestCurrentTransition() {
    return ReactCurrentBatchConfig$1.transition;
  }

  var ReactStrictModeWarnings = {
    recordUnsafeLifecycleWarnings: function (fiber, instance) {},
    flushPendingUnsafeLifecycleWarnings: function () {},
    recordLegacyContextWarning: function (fiber, instance) {},
    flushLegacyContextWarning: function () {},
    discardPendingWarnings: function () {}
  };

  {
    var findStrictRoot = function (fiber) {
      var maybeStrictRoot = null;
      var node = fiber;

      while (node !== null) {
        if (node.mode & StrictLegacyMode) {
          maybeStrictRoot = node;
        }

        node = node.return;
      }

      return maybeStrictRoot;
    };

    var setToSortedString = function (set) {
      var array = [];
      set.forEach(function (value) {
        array.push(value);
      });
      return array.sort().join(', ');
    };

    var pendingComponentWillMountWarnings = [];
    var pendingUNSAFE_ComponentWillMountWarnings = [];
    var pendingComponentWillReceivePropsWarnings = [];
    var pendingUNSAFE_ComponentWillReceivePropsWarnings = [];
    var pendingComponentWillUpdateWarnings = [];
    var pendingUNSAFE_ComponentWillUpdateWarnings = []; // Tracks components we have
already warned about.

    var didWarnAboutUnsafeLifecycles = new Set();

    ReactStrictModeWarnings.recordUnsafeLifecycleWarnings = function (fiber, instance) {
      // Dedupe strategy: Warn once per component.
      if (didWarnAboutUnsafeLifecycles.has(fiber.type)) {
        return;
      }

      if (typeof instance.componentWillMount === 'function' && // Don't warn about react-
lifecycles-compat polyfilled components.
      instance.componentWillMount.__suppressDeprecationWarning !== true) {
        pendingComponentWillMountWarnings.push(fiber);
      }

      if (fiber.mode & StrictLegacyMode && typeof instance.UNSAFE_componentWillMount ===
'function') {
        pendingUNSAFE_ComponentWillMountWarnings.push(fiber);
      }

      if (typeof instance.componentWillReceiveProps === 'function' &&
instance.componentWillReceiveProps.__suppressDeprecationWarning !== true) {
        pendingComponentWillReceivePropsWarnings.push(fiber);
      }
      if (fiber.mode & StrictLegacyMode && typeof
instance.UNSAFE_componentWillReceiveProps === 'function') {
        pendingUNSAFE_ComponentWillReceivePropsWarnings.push(fiber);
      }

      if (typeof instance.componentWillUpdate === 'function' &&
instance.componentWillUpdate.__suppressDeprecationWarning !== true) {
        pendingComponentWillUpdateWarnings.push(fiber);
      }

      if (fiber.mode & StrictLegacyMode && typeof instance.UNSAFE_componentWillUpdate ===
'function') {
        pendingUNSAFE_ComponentWillUpdateWarnings.push(fiber);
      }
    };

    ReactStrictModeWarnings.flushPendingUnsafeLifecycleWarnings = function () {
      // We do an initial pass to gather component names
      var componentWillMountUniqueNames = new Set();

      if (pendingComponentWillMountWarnings.length > 0) {
        pendingComponentWillMountWarnings.forEach(function (fiber) {
          componentWillMountUniqueNames.add(getComponentNameFromFiber(fiber) ||
'Component');
          didWarnAboutUnsafeLifecycles.add(fiber.type);
        });
        pendingComponentWillMountWarnings = [];
      }

      var UNSAFE_componentWillMountUniqueNames = new Set();

      if (pendingUNSAFE_ComponentWillMountWarnings.length > 0) {
        pendingUNSAFE_ComponentWillMountWarnings.forEach(function (fiber) {
          UNSAFE_componentWillMountUniqueNames.add(getComponentNameFromFiber(fiber) ||
'Component');
          didWarnAboutUnsafeLifecycles.add(fiber.type);
        });
        pendingUNSAFE_ComponentWillMountWarnings = [];
      }

      var componentWillReceivePropsUniqueNames = new Set();

      if (pendingComponentWillReceivePropsWarnings.length > 0) {
        pendingComponentWillReceivePropsWarnings.forEach(function (fiber) {
          componentWillReceivePropsUniqueNames.add(getComponentNameFromFiber(fiber) ||
'Component');
          didWarnAboutUnsafeLifecycles.add(fiber.type);
        });
        pendingComponentWillReceivePropsWarnings = [];
      }

      var UNSAFE_componentWillReceivePropsUniqueNames = new Set();

      if (pendingUNSAFE_ComponentWillReceivePropsWarnings.length > 0) {
        pendingUNSAFE_ComponentWillReceivePropsWarnings.forEach(function (fiber) {

UNSAFE_componentWillReceivePropsUniqueNames.add(getComponentNameFromFiber(fiber) ||
'Component');
          didWarnAboutUnsafeLifecycles.add(fiber.type);
        });
        pendingUNSAFE_ComponentWillReceivePropsWarnings = [];
      }

      var componentWillUpdateUniqueNames = new Set();

      if (pendingComponentWillUpdateWarnings.length > 0) {
        pendingComponentWillUpdateWarnings.forEach(function (fiber) {
          componentWillUpdateUniqueNames.add(getComponentNameFromFiber(fiber) ||
'Component');
          didWarnAboutUnsafeLifecycles.add(fiber.type);
        });
        pendingComponentWillUpdateWarnings = [];
      }

      var UNSAFE_componentWillUpdateUniqueNames = new Set();

      if (pendingUNSAFE_ComponentWillUpdateWarnings.length > 0) {
        pendingUNSAFE_ComponentWillUpdateWarnings.forEach(function (fiber) {
          UNSAFE_componentWillUpdateUniqueNames.add(getComponentNameFromFiber(fiber) ||
'Component');
          didWarnAboutUnsafeLifecycles.add(fiber.type);
        });
        pendingUNSAFE_ComponentWillUpdateWarnings = [];
      } // Finally, we flush all the warnings
      // UNSAFE_ ones before the deprecated ones, since they'll be 'louder'


      if (UNSAFE_componentWillMountUniqueNames.size > 0) {
        var sortedNames = setToSortedString(UNSAFE_componentWillMountUniqueNames);

        error('Using UNSAFE_componentWillMount in strict mode is not recommended and may
indicate bugs in your code. ' + 'See https://reactjs.org/link/unsafe-component-lifecycles
for details.\n\n' + '* Move code with side effects to componentDidMount, and set initial
state in the constructor.\n' + '\nPlease update the following components: %s',
sortedNames);
      }

      if (UNSAFE_componentWillReceivePropsUniqueNames.size > 0) {
        var _sortedNames =
setToSortedString(UNSAFE_componentWillReceivePropsUniqueNames);

        error('Using UNSAFE_componentWillReceiveProps in strict mode is not recommended '
+ 'and may indicate bugs in your code. ' + 'See https://reactjs.org/link/unsafe-
component-lifecycles for details.\n\n' + '* Move data fetching code or side effects to
componentDidUpdate.\n' + "* If you're updating state whenever props change, " + 'refactor
your code to use memoization techniques or move it to ' + 'static
getDerivedStateFromProps. Learn more at: https://reactjs.org/link/derived-state\n' +
'\nPlease update the following components: %s', _sortedNames);
      }

      if (UNSAFE_componentWillUpdateUniqueNames.size > 0) {
        var _sortedNames2 = setToSortedString(UNSAFE_componentWillUpdateUniqueNames);

        error('Using UNSAFE_componentWillUpdate in strict mode is not recommended ' +
'and may indicate bugs in your code. ' + 'See https://reactjs.org/link/unsafe-component-
lifecycles for details.\n\n' + '* Move data fetching code or side effects to
componentDidUpdate.\n' + '\nPlease update the following components: %s', _sortedNames2);
      }

      if (componentWillMountUniqueNames.size > 0) {
        var _sortedNames3 = setToSortedString(componentWillMountUniqueNames);

        warn('componentWillMount has been renamed, and is not recommended for use. ' +
'See https://reactjs.org/link/unsafe-component-lifecycles for details.\n\n' + '* Move
code with side effects to componentDidMount, and set initial state in the constructor.\n'
+ '* Rename componentWillMount to UNSAFE_componentWillMount to suppress ' + 'this warning
in non-strict mode. In React 18.x, only the UNSAFE_ name will work. ' + 'To rename all
deprecated lifecycles to their new names, you can run ' + '`npx react-codemod rename-
unsafe-lifecycles` in your project source folder.\n' + '\nPlease update the following
components: %s', _sortedNames3);
      }

      if (componentWillReceivePropsUniqueNames.size > 0) {
        var _sortedNames4 = setToSortedString(componentWillReceivePropsUniqueNames);

        warn('componentWillReceiveProps has been renamed, and is not recommended for use.
' + 'See https://reactjs.org/link/unsafe-component-lifecycles for details.\n\n' + '* Move
```

```
data fetching code or side effects to componentDidUpdate.\n' + "* If you're updating
state whenever props change, refactor your " + 'code to use memoization techniques or
move it to ' + 'static getDerivedStateFromProps. Learn more at:
https://reactjs.org/link/derived-state\n' + '* Rename componentWillReceiveProps to
UNSAFE_componentWillReceiveProps to suppress ' + 'this warning in non-strict mode. In
React 18.x, only the UNSAFE_ name will work. ' + 'To rename all deprecated lifecycles to
their new names, you can run ' + '`npx react-codemod rename-unsafe-lifecycles` in your
project source folder.\n' + '\nPlease update the following components: %s',
_sortedNames4);
      }

      if (componentWillUpdateUniqueNames.size > 0) {
        var _sortedNames5 = setToSortedString(componentWillUpdateUniqueNames);

        warn('componentWillUpdate has been renamed, and is not recommended for use. ' +
'See https://reactjs.org/link/unsafe-component-lifecycles for details.\n\n' + '* Move
data fetching code or side effects to componentDidUpdate.\n' + '* Rename
componentWillUpdate to UNSAFE_componentWillUpdate to suppress ' + 'this warning in non-
strict mode. In React 18.x, only the UNSAFE_ name will work. ' + 'To rename all
deprecated lifecycles to their new names, you can run ' + '`npx react-codemod rename-
unsafe-lifecycles` in your project source folder.\n' + '\nPlease update the following
components: %s', _sortedNames5);
      }
    };

    var pendingLegacyContextWarning = new Map(); // Tracks components we have already
warned about.

    var didWarnAboutLegacyContext = new Set();

    ReactStrictModeWarnings.recordLegacyContextWarning = function (fiber, instance) {
      var strictRoot = findStrictRoot(fiber);

      if (strictRoot === null) {
        error('Expected to find a StrictMode component in a strict mode tree. ' + 'This
error is likely caused by a bug in React. Please file an issue.');

        return;
      } // Dedup strategy: Warn once per component.


      if (didWarnAboutLegacyContext.has(fiber.type)) {
        return;
      }

      var warningsForRoot = pendingLegacyContextWarning.get(strictRoot);

      if (fiber.type.contextTypes != null || fiber.type.childContextTypes != null ||
instance !== null && typeof instance.getChildContext === 'function') {
        if (warningsForRoot === undefined) {
          warningsForRoot = [];
          pendingLegacyContextWarning.set(strictRoot, warningsForRoot);
        }

        warningsForRoot.push(fiber);
      }
    };

    ReactStrictModeWarnings.flushLegacyContextWarning = function () {
      pendingLegacyContextWarning.forEach(function (fiberArray, strictRoot) {
        if (fiberArray.length === 0) {
          return;
        }

        var firstFiber = fiberArray[0];
        var uniqueNames = new Set();
        fiberArray.forEach(function (fiber) {
          uniqueNames.add(getComponentNameFromFiber(fiber) || 'Component');
          didWarnAboutLegacyContext.add(fiber.type);
        });
        var sortedNames = setToSortedString(uniqueNames);

        try {
          setCurrentFiber(firstFiber);

          error('Legacy context API has been detected within a strict-mode tree.' +
'\n\nThe old API will be supported in all 16.x releases, but applications ' + 'using it
should migrate to the new version.' + '\n\nPlease update the following components: %s' +
'\n\nLearn more about this warning here: https://reactjs.org/link/legacy-context',
sortedNames);
        } finally {
          resetCurrentFiber();
        }
      });
    };

    ReactStrictModeWarnings.discardPendingWarnings = function () {
      pendingComponentWillMountWarnings = [];
      pendingUNSAFE_ComponentWillMountWarnings = [];
      pendingComponentWillReceivePropsWarnings = [];
      pendingUNSAFE_ComponentWillReceivePropsWarnings = [];
      pendingComponentWillUpdateWarnings = [];
      pendingUNSAFE_ComponentWillUpdateWarnings = [];
      pendingLegacyContextWarning = new Map();
    };
  }

  var didWarnAboutMaps;
  var didWarnAboutGenerators;
  var didWarnAboutStringRefs;
  var ownerHasKeyUseWarning;
  var ownerHasFunctionTypeWarning;

  var warnForMissingKey = function (child, returnFiber) {};

  {
    didWarnAboutMaps = false;
    didWarnAboutGenerators = false;
    didWarnAboutStringRefs = {};
    /**
     * Warn if there's no key explicitly set on dynamic arrays of children or
     * object keys are not valid. This allows us to keep track of children between
     * updates.
     */

    ownerHasKeyUseWarning = {};
    ownerHasFunctionTypeWarning = {};

    warnForMissingKey = function (child, returnFiber) {
      if (child === null || typeof child !== 'object') {
        return;
      }

      if (!child._store || child._store.validated || child.key != null) {
        return;
      }

      if (typeof child._store !== 'object') {
        throw new Error('React Component in warnForMissingKey should have a _store. ' +
'This error is likely caused by a bug in React. Please file an issue.');
      }

      child._store.validated = true;
      var componentName = getComponentNameFromFiber(returnFiber) || 'Component';

      if (ownerHasKeyUseWarning[componentName]) {
        return;
      }
      ownerHasKeyUseWarning[componentName] = true;

      error('Each child in a list should have a unique ' + '"key" prop. See
https://reactjs.org/link/warning-keys for ' + 'more information.');
    };
  }

  function isReactClass(type) {
    return type.prototype && type.prototype.isReactComponent;
  }

  function coerceRef(returnFiber, current, element) {
    var mixedRef = element.ref;

    if (mixedRef !== null && typeof mixedRef !== 'function' && typeof mixedRef !==
'object') {
      {
        // TODO: Clean this up once we turn on the string ref warning for
        // everyone, because the strict mode case will no longer be relevant
        if ((returnFiber.mode & StrictLegacyMode || warnAboutStringRefs) && // We warn in
ReactElement.js if owner and self are equal for string refs
        // because these cannot be automatically converted to an arrow function
        // using a codemod. Therefore, we don't have to warn about string refs again.
        !(element._owner && element._self && element._owner.stateNode !== element._self)
&& // Will already throw with "Function components cannot have string refs"
        !(element._owner && element._owner.tag !== ClassComponent) && // Will already
warn with "Function components cannot be given refs"
        !(typeof element.type === 'function' && !isReactClass(element.type)) && // Will
already throw with "Element ref was specified as a string (someStringRef) but no owner
was set"
        element._owner) {
          var componentName = getComponentNameFromFiber(returnFiber) || 'Component';

          if (!didWarnAboutStringRefs[componentName]) {
            {
              error('Component "%s" contains the string ref "%s". Support for string refs
' + 'will be removed in a future major release. We recommend using ' + 'useRef() or
createRef() instead. ' + 'Learn more about using refs safely here: ' +
'https://reactjs.org/link/strict-mode-string-ref', componentName, mixedRef);
            }

            didWarnAboutStringRefs[componentName] = true;
          }
        }
      }

      if (element._owner) {
        var owner = element._owner;
        var inst;

        if (owner) {
          var ownerFiber = owner;

          if (ownerFiber.tag !== ClassComponent) {
            throw new Error('Function components cannot have string refs. ' + 'We
recommend using useRef() instead. ' + 'Learn more about using refs safely here: ' +
'https://reactjs.org/link/strict-mode-string-ref');
          }

          inst = ownerFiber.stateNode;
        }

        if (!inst) {
          throw new Error("Missing owner for string ref " + mixedRef + ". This error is
likely caused by a " + 'bug in React. Please file an issue.');
        } // Assigning this to a const so Flow knows it won't change in the closure
        var resolvedInst = inst;

        {
          checkPropStringCoercion(mixedRef, 'ref');
        }

        var stringRef = '' + mixedRef; // Check if previous string ref matches new string
ref

        if (current !== null && current.ref !== null && typeof current.ref === 'function'
&& current.ref._stringRef === stringRef) {
          return current.ref;
        }

        var ref = function (value) {
          var refs = resolvedInst.refs;

          if (value === null) {
            delete refs[stringRef];
          } else {
            refs[stringRef] = value;
          }
        };

        ref._stringRef = stringRef;
        return ref;
      } else {
        if (typeof mixedRef !== 'string') {
          throw new Error('Expected ref to be a function, a string, an object returned by
React.createRef(), or null.');
        }

        if (!element._owner) {
          throw new Error("Element ref was specified as a string (" + mixedRef + ") but
no owner was set. This could happen for one of" + ' the following reasons:\n' + '1. You
may be adding a ref to a function component\n' + "2. You may be adding a ref to a
component that was not created inside a component's render method\n" + '3. You have
multiple copies of React loaded\n' + 'See https://reactjs.org/link/refs-must-have-owner
for more information.');
        }
      }
    }

    return mixedRef;
  }

  function throwOnInvalidObjectType(returnFiber, newChild) {
    var childString = Object.prototype.toString.call(newChild);
    throw new Error("Objects are not valid as a React child (found: " + (childString ===
'[object Object]' ? 'object with keys {' + Object.keys(newChild).join(', ') + '}' :
childString) + "). " + 'If you meant to render a collection of children, use an array ' +
'instead.');
  }

  function warnOnFunctionType(returnFiber) {
    {
      var componentName = getComponentNameFromFiber(returnFiber) || 'Component';

      if (ownerHasFunctionTypeWarning[componentName]) {
        return;
      }

      ownerHasFunctionTypeWarning[componentName] = true;

      error('Functions are not valid as a React child. This may happen if ' + 'you return
a Component instead of <Component /> from render. ' + 'Or maybe you meant to call this
function rather than return it.');
    }
  }
```

```
    function resolveLazy(lazyType) {
      var payload = lazyType._payload;
      var init = lazyType._init;
      return init(payload);
    } // This wrapper function exists because I expect to clone the code in each path
    // to be able to optimize each path individually by branching early. This needs
    // a compiler or we can do it manually. Helpers that don't need this branching
    // live outside of this function.


    function ChildReconciler(shouldTrackSideEffects) {
      function deleteChild(returnFiber, childToDelete) {
        if (!shouldTrackSideEffects) {
          // Noop.
          return;
        }

        var deletions = returnFiber.deletions;

        if (deletions === null) {
          returnFiber.deletions = [childToDelete];
          returnFiber.flags |= ChildDeletion;
        } else {
          deletions.push(childToDelete);
        }
      }

      function deleteRemainingChildren(returnFiber, currentFirstChild) {
        if (!shouldTrackSideEffects) {
          // Noop.
          return null;
        } // TODO: For the shouldClone case, this could be micro-optimized a bit by
        // assuming that after the first child we've already added everything.


        var childToDelete = currentFirstChild;

        while (childToDelete !== null) {
          deleteChild(returnFiber, childToDelete);
          childToDelete = childToDelete.sibling;
        }

        return null;
      }

      function mapRemainingChildren(returnFiber, currentFirstChild) {
        // Add the remaining children to a temporary map so that we can find them by
        // keys quickly. Implicit (null) keys get added to this set with their index
        // instead.
        var existingChildren = new Map();
        var existingChild = currentFirstChild;

        while (existingChild !== null) {
          if (existingChild.key !== null) {
            existingChildren.set(existingChild.key, existingChild);
          } else {
            existingChildren.set(existingChild.index, existingChild);
          }

          existingChild = existingChild.sibling;
        }

        return existingChildren;
      }

      function useFiber(fiber, pendingProps) {
        // We currently set sibling to null and index to 0 here because it is easy
        // to forget to do before returning it. E.g. for the single child case.
        var clone = createWorkInProgress(fiber, pendingProps);
        clone.index = 0;
        clone.sibling = null;
        return clone;
      }

      function placeChild(newFiber, lastPlacedIndex, newIndex) {
        newFiber.index = newIndex;

        if (!shouldTrackSideEffects) {
          // During hydration, the useId algorithm needs to know which fibers are
          // part of a list of children (arrays, iterators).
          newFiber.flags |= Forked;
          return lastPlacedIndex;
        }

        var current = newFiber.alternate;

        if (current !== null) {
          var oldIndex = current.index;

          if (oldIndex < lastPlacedIndex) {
            // This is a move.
            newFiber.flags |= Placement;
            return lastPlacedIndex;
          } else {
            // This item can stay in place.
            return oldIndex;
          }
        } else {
          // This is an insertion.
          newFiber.flags |= Placement;
          return lastPlacedIndex;
        }
      }

      function placeSingleChild(newFiber) {
        // This is simpler for the single child case. We only need to do a
        // placement for inserting new children.
        if (shouldTrackSideEffects && newFiber.alternate === null) {
          newFiber.flags |= Placement;
        }

        return newFiber;
      }

      function updateTextNode(returnFiber, current, textContent, lanes) {
        if (current === null || current.tag !== HostText) {
          // Insert
          var created = createFiberFromText(textContent, returnFiber.mode, lanes);
          created.return = returnFiber;
          return created;
        } else {
          // Update
          var existing = useFiber(current, textContent);
          existing.return = returnFiber;
          return existing;
        }
      }

      function updateElement(returnFiber, current, element, lanes) {
        var elementType = element.type;

        if (elementType === REACT_FRAGMENT_TYPE) {
          return updateFragment(returnFiber, current, element.props.children, lanes,
  element.key);
        }

        if (current !== null) {
          if (current.elementType === elementType || ( // Keep this check inline so it only
  runs on the false path:
           isCompatibleFamilyForHotReloading(current, element) ) || // Lazy types should
  reconcile their resolved type.
          // We need to do this after the Hot Reloading check above,
          // because hot reloading has different semantics than prod because
          // it doesn't resuspend. So we can't let the call below suspend.
          typeof elementType === 'object' && elementType !== null && elementType.$$typeof
  === REACT_LAZY_TYPE && resolveLazy(elementType) === current.type) {
            // Move based on index
            var existing = useFiber(current, element.props);
            existing.ref = coerceRef(returnFiber, current, element);
            existing.return = returnFiber;

            {
              existing._debugSource = element._source;
              existing._debugOwner = element._owner;
            }

            return existing;
          }
        } // Insert


        var created = createFiberFromElement(element, returnFiber.mode, lanes);
        created.ref = coerceRef(returnFiber, current, element);
        created.return = returnFiber;
        return created;
      }

      function updatePortal(returnFiber, current, portal, lanes) {
        if (current === null || current.tag !== HostPortal ||
  current.stateNode.containerInfo !== portal.containerInfo ||
  current.stateNode.implementation !== portal.implementation) {
          // Insert
          var created = createFiberFromPortal(portal, returnFiber.mode, lanes);
          created.return = returnFiber;
          return created;
        } else {
          // Update
          var existing = useFiber(current, portal.children || []);
          existing.return = returnFiber;
          return existing;
        }
      }

      function updateFragment(returnFiber, current, fragment, lanes, key) {
        if (current === null || current.tag !== Fragment) {
          // Insert
          var created = createFiberFromFragment(fragment, returnFiber.mode, lanes, key);
          created.return = returnFiber;
          return created;
        } else {
          // Update
          var existing = useFiber(current, fragment);
          existing.return = returnFiber;
          return existing;
        }
      }

      function createChild(returnFiber, newChild, lanes) {
        if (typeof newChild === 'string' && newChild !== '' || typeof newChild ===
  'number') {
          // Text nodes don't have keys. If the previous node is implicitly keyed
          // we can continue to replace it without aborting even if it is not a text
          // node.
          var created = createFiberFromText('' + newChild, returnFiber.mode, lanes);
          created.return = returnFiber;
          return created;
        }

        if (typeof newChild === 'object' && newChild !== null) {
          switch (newChild.$$typeof) {
            case REACT_ELEMENT_TYPE:
              {
                var _created = createFiberFromElement(newChild, returnFiber.mode, lanes);

                _created.ref = coerceRef(returnFiber, null, newChild);
                _created.return = returnFiber;
                return _created;
              }

            case REACT_PORTAL_TYPE:
              {
                var _created2 = createFiberFromPortal(newChild, returnFiber.mode, lanes);

                _created2.return = returnFiber;
                return _created2;
              }

            case REACT_LAZY_TYPE:
              {
                var payload = newChild._payload;
                var init = newChild._init;
                return createChild(returnFiber, init(payload), lanes);
              }
          }

          if (isArray(newChild) || getIteratorFn(newChild)) {
            var _created3 = createFiberFromFragment(newChild, returnFiber.mode, lanes,
  null);

            _created3.return = returnFiber;
            return _created3;
          }

          throwOnInvalidObjectType(returnFiber, newChild);
        }

        {
          if (typeof newChild === 'function') {
            warnOnFunctionType(returnFiber);
          }
        }

        return null;
      }

      function updateSlot(returnFiber, oldFiber, newChild, lanes) {
        // Update the fiber if the keys match, otherwise return null.
        var key = oldFiber !== null ? oldFiber.key : null;

        if (typeof newChild === 'string' && newChild !== '' || typeof newChild ===
  'number') {
          // Text nodes don't have keys. If the previous node is implicitly keyed
          // we can continue to replace it without aborting even if it is not a text
          // node.
          if (key !== null) {
            return null;
          }

          return updateTextNode(returnFiber, oldFiber, '' + newChild, lanes);
        }

        if (typeof newChild === 'object' && newChild !== null) {
          switch (newChild.$$typeof) {
            case REACT_ELEMENT_TYPE:
              {
                if (newChild.key === key) {
                  return updateElement(returnFiber, oldFiber, newChild, lanes);
                } else {
                  return null;
                }
              }

            case REACT_PORTAL_TYPE:
              {
                if (newChild.key === key) {
                  return updatePortal(returnFiber, oldFiber, newChild, lanes);
                } else {
                  return null;
                }
              }

            case REACT_LAZY_TYPE:
              {
                var payload = newChild._payload;
                var init = newChild._init;
                return updateSlot(returnFiber, oldFiber, init(payload), lanes);
              }
          }

          if (isArray(newChild) || getIteratorFn(newChild)) {
            if (key !== null) {
              return null;
            }

            return updateFragment(returnFiber, oldFiber, newChild, lanes, null);
          }

          throwOnInvalidObjectType(returnFiber, newChild);
        }

        {
          if (typeof newChild === 'function') {
            warnOnFunctionType(returnFiber);
          }
        }

        return null;
      }

      function updateFromMap(existingChildren, returnFiber, newIdx, newChild, lanes) {
        if (typeof newChild === 'string' && newChild !== '' || typeof newChild ===
  'number') {
          // Text nodes don't have keys, so we neither have to check the old nor
          // new node for the key. If both are text nodes, they match.
          var matchedFiber = existingChildren.get(newIdx) || null;
          return updateTextNode(returnFiber, matchedFiber, '' + newChild, lanes);
        }

        if (typeof newChild === 'object' && newChild !== null) {
          switch (newChild.$$typeof) {
            case REACT_ELEMENT_TYPE:
              {
                var _matchedFiber = existingChildren.get(newChild.key === null ? newIdx :
  newChild.key) || null;

                return updateElement(returnFiber, _matchedFiber, newChild, lanes);
              }

            case REACT_PORTAL_TYPE:
              {
                var _matchedFiber2 = existingChildren.get(newChild.key === null ? newIdx :
  newChild.key) || null;

                return updatePortal(returnFiber, _matchedFiber2, newChild, lanes);
              }

            case REACT_LAZY_TYPE:
              var payload = newChild._payload;
              var init = newChild._init;
              return updateFromMap(existingChildren, returnFiber, newIdx, init(payload),
  lanes);
          }

          if (isArray(newChild) || getIteratorFn(newChild)) {
            var _matchedFiber3 = existingChildren.get(newIdx) || null;

            return updateFragment(returnFiber, _matchedFiber3, newChild, lanes, null);
          }

          throwOnInvalidObjectType(returnFiber, newChild);
        }

        {
          if (typeof newChild === 'function') {
            warnOnFunctionType(returnFiber);
          }
        }

        return null;
      }
      /**
       * Warns if there is a duplicate or missing key
       */


      function warnOnInvalidKey(child, knownKeys, returnFiber) {
        {
          if (typeof child !== 'object' || child === null) {
            return knownKeys;
          }

          switch (child.$$typeof) {
            case REACT_ELEMENT_TYPE:
            case REACT_PORTAL_TYPE:
              warnForMissingKey(child, returnFiber);
              var key = child.key;

              if (typeof key !== 'string') {
                break;
              }

              if (knownKeys === null) {
                knownKeys = new Set();
                knownKeys.add(key);
                break;
              }

              if (!knownKeys.has(key)) {
                knownKeys.add(key);
                break;
              }

              error('Encountered two children with the same key, `%s`. ' + 'Keys should be
  unique so that components maintain their identity ' + 'across updates. Non-unique keys
  may cause children to be ' + 'duplicated and/or omitted — the behavior is unsupported and
  ' + 'could change in a future version.', key);

              break;

            case REACT_LAZY_TYPE:
              var payload = child._payload;
              var init = child._init;
              warnOnInvalidKey(init(payload), knownKeys, returnFiber);
            break;
          }
        }

        return knownKeys;
      }

      function reconcileChildrenArray(returnFiber, currentFirstChild, newChildren, lanes) {
        // This algorithm can't optimize by searching from both ends since we
        // don't have backpointers on fibers. I'm trying to see how far we can get
        // with that model. If it ends up not being worth the tradeoffs, we can
        // add it later.
        // Even with a two ended optimization, we'd want to optimize for the case
        // where there are few changes and brute force the comparison instead of
        // going for the Map. It'd like to explore hitting that path first in
        // forward-only mode and only go for the Map once we notice that we need
        // lots of look ahead. This doesn't handle reversal as well as two ended
        // search but that's unusual. Besides, for the two ended optimization to
        // work on Iterables, we'd need to copy the whole set.
        // In this first iteration, we'll just live with hitting the bad case
        // (adding everything to a Map) in for every insert/move.
        // If you change this code, also update reconcileChildrenIterator() which
        // uses the same algorithm.
        {
          // First, validate keys.
          var knownKeys = null;

          for (var i = 0; i < newChildren.length; i++) {
            var child = newChildren[i];
            knownKeys = warnOnInvalidKey(child, knownKeys, returnFiber);
          }
        }

        var resultingFirstChild = null;
        var previousNewFiber = null;
        var oldFiber = currentFirstChild;
        var lastPlacedIndex = 0;
        var newIdx = 0;
        var nextOldFiber = null;

        for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
          if (oldFiber.index > newIdx) {
            nextOldFiber = oldFiber;
            oldFiber = null;
          } else {
            nextOldFiber = oldFiber.sibling;
          }

          var newFiber = updateSlot(returnFiber, oldFiber, newChildren[newIdx], lanes);

          if (newFiber === null) {
            // TODO: This breaks on empty slots like null children. That's
            // unfortunate because it triggers the slow path all the time. We need
            // a better way to communicate whether this was a miss or null,
            // boolean, undefined, etc.
            if (oldFiber === null) {
              oldFiber = nextOldFiber;
            }

            break;
          }

          if (shouldTrackSideEffects) {
            if (oldFiber && newFiber.alternate === null) {
              // We matched the slot, but we didn't reuse the existing fiber, so we
              // need to delete the existing child.
              deleteChild(returnFiber, oldFiber);
            }
          }
          lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);

          if (previousNewFiber === null) {
            // TODO: Move out of the loop. This only happens for the first run.
            resultingFirstChild = newFiber;
          } else {
            // TODO: Defer siblings if we're not at the right index for this slot.
            // I.e. if we had null values before, then we want to defer this
            // for each null value. However, we also don't want to call updateSlot
            // with the previous one.
            previousNewFiber.sibling = newFiber;
          }

          previousNewFiber = newFiber;
          oldFiber = nextOldFiber;
        }

        if (newIdx === newChildren.length) {
          // We've reached the end of the new children. We can delete the rest.
          deleteRemainingChildren(returnFiber, oldFiber);

          if (getIsHydrating()) {
            var numberOfForks = newIdx;
            pushTreeFork(returnFiber, numberOfForks);
          }

          return resultingFirstChild;
        }

        if (oldFiber === null) {
          // If we don't have any more existing children we can choose a fast path
          // since the rest will all be insertions.
          for (; newIdx < newChildren.length; newIdx++) {
            var _newFiber = createChild(returnFiber, newChildren[newIdx], lanes);

            if (_newFiber === null) {
              continue;
            }

            lastPlacedIndex = placeChild(_newFiber, lastPlacedIndex, newIdx);

            if (previousNewFiber === null) {
              // TODO: Move out of the loop. This only happens for the first run.
              resultingFirstChild = _newFiber;
            } else {
              previousNewFiber.sibling = _newFiber;
            }

            previousNewFiber = _newFiber;
          }

          if (getIsHydrating()) {
            var _numberOfForks = newIdx;
            pushTreeFork(returnFiber, _numberOfForks);
          }

          return resultingFirstChild;
        } // Add all children to a key map for quick lookups.


        var existingChildren = mapRemainingChildren(returnFiber, oldFiber); // Keep
  scanning and use the map to restore deleted items as moves.

        for (; newIdx < newChildren.length; newIdx++) {
          var _newFiber2 = updateFromMap(existingChildren, returnFiber, newIdx,
  newChildren[newIdx], lanes);

          if (_newFiber2 !== null) {
            if (shouldTrackSideEffects) {
              if (_newFiber2.alternate !== null) {
                // The new fiber is a work in progress, but if there exists a
                // current, that means that we reused the fiber. We need to delete
                // it from the child list so that we don't add it to the deletion
                // list.
                existingChildren.delete(_newFiber2.key === null ? newIdx : _newFiber2.key);
              }
            }

            lastPlacedIndex = placeChild(_newFiber2, lastPlacedIndex, newIdx);

            if (previousNewFiber === null) {
              resultingFirstChild = _newFiber2;
            } else {
              previousNewFiber.sibling = _newFiber2;
            }

            previousNewFiber = _newFiber2;
          }
        }

        if (shouldTrackSideEffects) {
          // Any existing children that weren't consumed above were deleted. We need
          // to add them to the deletion list.
          existingChildren.forEach(function (child) {
            return deleteChild(returnFiber, child);
          });
        }

        if (getIsHydrating()) {
          var _numberOfForks2 = newIdx;
          pushTreeFork(returnFiber, _numberOfForks2);
        }

        return resultingFirstChild;
      }

      function reconcileChildrenIterator(returnFiber, currentFirstChild,
  newChildrenIterable, lanes) {
        // This is the same implementation as reconcileChildrenArray(),
        // but using the iterator instead.
        var iteratorFn = getIteratorFn(newChildrenIterable);

        if (typeof iteratorFn !== 'function') {
          throw new Error('An object is not an iterable. This error is likely caused by a
  bug in ' + 'React. Please file an issue.');
        }

        {
          // We don't support rendering Generators because it's a mutation.
          // See https://github.com/facebook/react/issues/12995
          if (typeof Symbol === 'function' && // $FlowFixMe Flow doesn't know about
  toStringTag
          newChildrenIterable[Symbol.toStringTag] === 'Generator') {
            if (!didWarnAboutGenerators) {
              error('Using Generators as children is unsupported and will likely yield ' +
  'unexpected results because enumerating a generator mutates it. ' + 'You may convert it
  to an array with `Array.from()` or the ' + '`[...spread]` operator before rendering. Keep
  in mind ' + 'you might need to polyfill these features for older browsers.');
            }

            didWarnAboutGenerators = true;
          } // Warn about using Maps as children


          if (newChildrenIterable.entries === iteratorFn) {
            if (!didWarnAboutMaps) {
              error('Using Maps as children is not supported. ' + 'Use an array of keyed
  ReactElements instead.');
            }

            didWarnAboutMaps = true;
          } // First, validate keys.
          // We'll get a different iterator later for the main pass.


          var _newChildren = iteratorFn.call(newChildrenIterable);

          if (_newChildren) {
            var knownKeys = null;

            var _step = _newChildren.next();

            for (; !_step.done; _step = _newChildren.next()) {
              var child = _step.value;
              knownKeys = warnOnInvalidKey(child, knownKeys, returnFiber);
            }
          }
        }

        var newChildren = iteratorFn.call(newChildrenIterable);

        if (newChildren == null) {
          throw new Error('An iterable object provided no iterator.');
        }

        var resultingFirstChild = null;
        var previousNewFiber = null;
        var oldFiber = currentFirstChild;
        var lastPlacedIndex = 0;
        var newIdx = 0;
        var nextOldFiber = null;
        var step = newChildren.next();

        for (; oldFiber !== null && !step.done; newIdx++, step = newChildren.next()) {
          if (oldFiber.index > newIdx) {
            nextOldFiber = oldFiber;
            oldFiber = null;
          } else {
            nextOldFiber = oldFiber.sibling;
          }

          var newFiber = updateSlot(returnFiber, oldFiber, step.value, lanes);

          if (newFiber === null) {
            // TODO: This breaks on empty slots like null children. That's
            // unfortunate because it triggers the slow path all the time. We need
            // a better way to communicate whether this was a miss or null,
            // boolean, undefined, etc.
            if (oldFiber === null) {
              oldFiber = nextOldFiber;
            }

            break;
          }

          if (shouldTrackSideEffects) {
            if (oldFiber && newFiber.alternate === null) {
              // We matched the slot, but we didn't reuse the existing fiber, so we
              // need to delete the existing child.
              deleteChild(returnFiber, oldFiber);
            }
          }

          lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);

          if (previousNewFiber === null) {
            // TODO: Move out of the loop. This only happens for the first run.
            resultingFirstChild = newFiber;
          } else {
            // TODO: Defer siblings if we're not at the right index for this slot.
            // I.e. if we had null values before, then we want to defer this
            // for each null value. However, we also don't want to call updateSlot
            // with the previous one.
            previousNewFiber.sibling = newFiber;
          }

          previousNewFiber = newFiber;
          oldFiber = nextOldFiber;
        }

        if (step.done) {
          // We've reached the end of the new children. We can delete the rest.
          deleteRemainingChildren(returnFiber, oldFiber);

          if (getIsHydrating()) {
            var numberOfForks = newIdx;
            pushTreeFork(returnFiber, numberOfForks);
          }

          return resultingFirstChild;
        }

        if (oldFiber === null) {
          // If we don't have any more existing children we can choose a fast path
          // since the rest will all be insertions.
          for (; !step.done; newIdx++, step = newChildren.next()) {
            var _newFiber3 = createChild(returnFiber, step.value, lanes);

            if (_newFiber3 === null) {
              continue;
            }

            lastPlacedIndex = placeChild(_newFiber3, lastPlacedIndex, newIdx);

            if (previousNewFiber === null) {
              // TODO: Move out of the loop. This only happens for the first run.
              resultingFirstChild = _newFiber3;
            } else {
              previousNewFiber.sibling = _newFiber3;
            }

            previousNewFiber = _newFiber3;
          }

          if (getIsHydrating()) {
            var _numberOfForks3 = newIdx;
            pushTreeFork(returnFiber, _numberOfForks3);
          }

          return resultingFirstChild;
        } // Add all children to a key map for quick lookups.


        var existingChildren = mapRemainingChildren(returnFiber, oldFiber); // Keep
  scanning and use the map to restore deleted items as moves.

        for (; !step.done; newIdx++, step = newChildren.next()) {
          var _newFiber4 = updateFromMap(existingChildren, returnFiber, newIdx, step.value,
  lanes);

          if (_newFiber4 !== null) {
            if (shouldTrackSideEffects) {
              if (_newFiber4.alternate !== null) {
                // The new fiber is a work in progress, but if there exists a
                // current, that means that we reused the fiber. We need to delete
                // it from the child list so that we don't add it to the deletion
                // list.
                existingChildren.delete(_newFiber4.key === null ? newIdx : _newFiber4.key);
              }
            }

            lastPlacedIndex = placeChild(_newFiber4, lastPlacedIndex, newIdx);

            if (previousNewFiber === null) {
              resultingFirstChild = _newFiber4;
            } else {
              previousNewFiber.sibling = _newFiber4;
            }

            previousNewFiber = _newFiber4;
          }
        }

        if (shouldTrackSideEffects) {
          // Any existing children that weren't consumed above were deleted. We need
          // to add them to the deletion list.
          existingChildren.forEach(function (child) {
            return deleteChild(returnFiber, child);
          });
        }

        if (getIsHydrating()) {
          var _numberOfForks4 = newIdx;
          pushTreeFork(returnFiber, _numberOfForks4);
        }

        return resultingFirstChild;
      }

      function reconcileSingleTextNode(returnFiber, currentFirstChild, textContent, lanes)
  {
        // There's no need to check for keys on text nodes since we don't have a
        // way to define them.
        if (currentFirstChild !== null && currentFirstChild.tag === HostText) {
          // We already have an existing node so let's just update it and delete
          // the rest.
          deleteRemainingChildren(returnFiber, currentFirstChild.sibling);
          var existing = useFiber(currentFirstChild, textContent);
          existing.return = returnFiber;
          return existing;
        } // The existing first child is not a text node so we need to create one
        // and delete the existing ones.


        deleteRemainingChildren(returnFiber, currentFirstChild);
        var created = createFiberFromText(textContent, returnFiber.mode, lanes);
        created.return = returnFiber;
        return created;
      }

      function reconcileSingleElement(returnFiber, currentFirstChild, element, lanes) {
        var key = element.key;
        var child = currentFirstChild;

        while (child !== null) {
          // TODO: If key === null and child.key === null, then this only applies to
          // the first item in the list.
          if (child.key === key) {
            var elementType = element.type;

            if (elementType === REACT_FRAGMENT_TYPE) {
              if (child.tag === Fragment) {
                deleteRemainingChildren(returnFiber, child.sibling);
                var existing = useFiber(child, element.props.children);
```

```
              existing.return = returnFiber;

              {
                existing._debugSource = element._source;
                existing._debugOwner = element._owner;
              }

              return existing;
            }
          } else {
            if (child.elementType === elementType || ( // Keep this check inline so it
only runs on the false path:
             isCompatibleFamilyForHotReloading(child, element) ) || // Lazy types should
reconcile their resolved type.
            // We need to do this after the Hot Reloading check above,
            // because hot reloading has different semantics than prod because
            // it doesn't resuspend. So we can't let the call below suspend.
            typeof elementType === 'object' && elementType !== null &&
elementType.$$typeof === REACT_LAZY_TYPE && resolveLazy(elementType) === child.type) {
              deleteRemainingChildren(returnFiber, child.sibling);

              var _existing = useFiber(child, element.props);

              _existing.ref = coerceRef(returnFiber, child, element);
              _existing.return = returnFiber;

              {
                _existing._debugSource = element._source;
                _existing._debugOwner = element._owner;
              }

              return _existing;
            }
          } // Didn't match.


          deleteRemainingChildren(returnFiber, child);
          break;
        } else {
          deleteChild(returnFiber, child);
        }

        child = child.sibling;
      }

      if (element.type === REACT_FRAGMENT_TYPE) {
        var created = createFiberFromFragment(element.props.children, returnFiber.mode,
lanes, element.key);
        created.return = returnFiber;
        return created;
      } else {
        var _created4 = createFiberFromElement(element, returnFiber.mode, lanes);

        _created4.ref = coerceRef(returnFiber, currentFirstChild, element);
        _created4.return = returnFiber;
        return _created4;
      }
    }

    function reconcileSinglePortal(returnFiber, currentFirstChild, portal, lanes) {
      var key = portal.key;
      var child = currentFirstChild;

      while (child !== null) {
        // TODO: If key === null and child.key === null, then this only applies to
        // the first item in the list.
        if (child.key === key) {
          if (child.tag === HostPortal && child.stateNode.containerInfo ===
portal.containerInfo && child.stateNode.implementation === portal.implementation) {
```

```
            deleteRemainingChildren(returnFiber, child.sibling);
            var existing = useFiber(child, portal.children || []);
            existing.return = returnFiber;
            return existing;
          } else {
            deleteRemainingChildren(returnFiber, child);
            break;
          }
        } else {
          deleteChild(returnFiber, child);
        }

        child = child.sibling;
      }

      var created = createFiberFromPortal(portal, returnFiber.mode, lanes);
      created.return = returnFiber;
      return created;
    } // This API will tag the children with the side-effect of the reconciliation
    // itself. They will be added to the side-effect list as we pass through the
    // children and the parent.


    function reconcileChildFibers(returnFiber, currentFirstChild, newChild, lanes) {
      // This function is not recursive.
      // If the top level item is an array, we treat it as a set of children,
      // not as a fragment. Nested arrays on the other hand will be treated as
      // fragment nodes. Recursion happens at the normal flow.
      // Handle top level unkeyed fragments as if they were arrays.
      // This leads to an ambiguity between <>{[...]}</> and <>...</>.
      // We treat the ambiguous cases above the same.
      var isUnkeyedTopLevelFragment = typeof newChild === 'object' && newChild !== null
&& newChild.type === REACT_FRAGMENT_TYPE && newChild.key === null;

      if (isUnkeyedTopLevelFragment) {
        newChild = newChild.props.children;
      } // Handle object types


      if (typeof newChild === 'object' && newChild !== null) {
        switch (newChild.$$typeof) {
          case REACT_ELEMENT_TYPE:
            return placeSingleChild(reconcileSingleElement(returnFiber,
currentFirstChild, newChild, lanes));

          case REACT_PORTAL_TYPE:
            return placeSingleChild(reconcileSinglePortal(returnFiber, currentFirstChild,
newChild, lanes));

          case REACT_LAZY_TYPE:
            var payload = newChild._payload;
            var init = newChild._init; // TODO: This function is supposed to be non-
recursive.

            return reconcileChildFibers(returnFiber, currentFirstChild, init(payload),
lanes);
        }

        if (isArray(newChild)) {
          return reconcileChildrenArray(returnFiber, currentFirstChild, newChild, lanes);
        }

        if (getIteratorFn(newChild)) {
          return reconcileChildrenIterator(returnFiber, currentFirstChild, newChild,
lanes);
        }

        throwOnInvalidObjectType(returnFiber, newChild);
      }
      if (typeof newChild === 'string' && newChild !== '' || typeof newChild ===
'number') {
        return placeSingleChild(reconcileSingleTextNode(returnFiber, currentFirstChild,
'' + newChild, lanes));
      }

      {
        if (typeof newChild === 'function') {
          warnOnFunctionType(returnFiber);
        }
      } // Remaining cases are all treated as empty.


      return deleteRemainingChildren(returnFiber, currentFirstChild);
    }

    return reconcileChildFibers;
  }

  var reconcileChildFibers = ChildReconciler(true);
  var mountChildFibers = ChildReconciler(false);
  function cloneChildFibers(current, workInProgress) {
    if (current !== null && workInProgress.child !== current.child) {
      throw new Error('Resuming work not yet implemented.');
    }

    if (workInProgress.child === null) {
      return;
    }

    var currentChild = workInProgress.child;
    var newChild = createWorkInProgress(currentChild, currentChild.pendingProps);
    workInProgress.child = newChild;
    newChild.return = workInProgress;

    while (currentChild.sibling !== null) {
      currentChild = currentChild.sibling;
      newChild = newChild.sibling = createWorkInProgress(currentChild,
currentChild.pendingProps);
      newChild.return = workInProgress;
    }

    newChild.sibling = null;
  } // Reset a workInProgress child set to prepare it for a second pass.

  function resetChildFibers(workInProgress, lanes) {
    var child = workInProgress.child;

    while (child !== null) {
      resetWorkInProgress(child, lanes);
      child = child.sibling;
    }
  }

  var valueCursor = createCursor(null);
  var rendererSigil;

  {
    // Use this to detect multiple renderers using the same context
    rendererSigil = {};
  }

  var currentlyRenderingFiber = null;
  var lastContextDependency = null;
  var lastFullyObservedContext = null;
  var isDisallowedContextReadInDEV = false;
  function resetContextDependencies() {
    // This is called right before React yields execution, to ensure `readContext`
```

```
      // cannot be called outside the render phase.
      currentlyRenderingFiber = null;
      lastContextDependency = null;
      lastFullyObservedContext = null;

      {
        isDisallowedContextReadInDEV = false;
      }
    }
    function enterDisallowedContextReadInDEV() {
      {
        isDisallowedContextReadInDEV = true;
      }
    }
    function exitDisallowedContextReadInDEV() {
      {
        isDisallowedContextReadInDEV = false;
      }
    }
    function pushProvider(providerFiber, context, nextValue) {
      {
        push(valueCursor, context._currentValue, providerFiber);
        context._currentValue = nextValue;

        {
          if (context._currentRenderer !== undefined && context._currentRenderer !== null
&& context._currentRenderer !== rendererSigil) {
            error('Detected multiple renderers concurrently rendering the ' + 'same context
provider. This is currently unsupported.');
          }

          context._currentRenderer = rendererSigil;
        }
      }
    }
    function popProvider(context, providerFiber) {
      var currentValue = valueCursor.current;
      pop(valueCursor, providerFiber);

      {
        {
          context._currentValue = currentValue;
        }
      }
    }
    function scheduleContextWorkOnParentPath(parent, renderLanes, propagationRoot) {
      // Update the child lanes of all the ancestors, including the alternates.
      var node = parent;

      while (node !== null) {
        var alternate = node.alternate;

        if (!isSubsetOfLanes(node.childLanes, renderLanes)) {
          node.childLanes = mergeLanes(node.childLanes, renderLanes);

          if (alternate !== null) {
            alternate.childLanes = mergeLanes(alternate.childLanes, renderLanes);
          }
        } else if (alternate !== null && !isSubsetOfLanes(alternate.childLanes,
renderLanes)) {
          alternate.childLanes = mergeLanes(alternate.childLanes, renderLanes);
        }

        if (node === propagationRoot) {
          break;
        }

        node = node.return;
      }
      {
        if (node !== propagationRoot) {
          error('Expected to find the propagation root when scheduling context work. ' +
  'This error is likely caused by a bug in React. Please file an issue.');
        }
      }
    }
    function propagateContextChange(workInProgress, context, renderLanes) {
      {
        propagateContextChange_eager(workInProgress, context, renderLanes);
      }
    }

    function propagateContextChange_eager(workInProgress, context, renderLanes) {

      var fiber = workInProgress.child;

      if (fiber !== null) {
        // Set the return pointer of the child to the work-in-progress fiber.
        fiber.return = workInProgress;
      }

      while (fiber !== null) {
        var nextFiber = void 0; // Visit this fiber.

        var list = fiber.dependencies;

        if (list !== null) {
          nextFiber = fiber.child;
          var dependency = list.firstContext;

          while (dependency !== null) {
            // Check if the context matches.
            if (dependency.context === context) {
              // Match! Schedule an update on this fiber.
              if (fiber.tag === ClassComponent) {
                // Schedule a force update on the work-in-progress.
                var lane = pickArbitraryLane(renderLanes);
                var update = createUpdate(NoTimestamp, lane);
                update.tag = ForceUpdate; // TODO: Because we don't have a work-in-
  progress, this will add the
                // update to the current fiber, too, which means it will persist even if
                // this render is thrown away. Since it's a race condition, not sure it's
                // worth fixing.
                // Inlined `enqueueUpdate` to remove interleaved update check

                var updateQueue = fiber.updateQueue;

                if (updateQueue === null) ; else {
                  var sharedQueue = updateQueue.shared;
                  var pending = sharedQueue.pending;

                  if (pending === null) {
                    // This is the first update. Create a circular list.
                    update.next = update;
                  } else {
                    update.next = pending.next;
                    pending.next = update;
                  }

                  sharedQueue.pending = update;
                }
              }

              fiber.lanes = mergeLanes(fiber.lanes, renderLanes);
              var alternate = fiber.alternate;

              if (alternate !== null) {
                alternate.lanes = mergeLanes(alternate.lanes, renderLanes);
              }

              scheduleContextWorkOnParentPath(fiber.return, renderLanes, workInProgress);
  // Mark the updated lanes on the list, too.

              list.lanes = mergeLanes(list.lanes, renderLanes); // Since we already found a
  match, we can stop traversing the
              // dependency list.

              break;
            }

            dependency = dependency.next;
          }
        } else if (fiber.tag === ContextProvider) {
          // Don't scan deeper if this is a matching provider
          nextFiber = fiber.type === workInProgress.type ? null : fiber.child;
        } else if (fiber.tag === DehydratedFragment) {
          // If a dehydrated suspense boundary is in this subtree, we don't know
          // if it will have any context consumers in it. The best we can do is
          // mark it as having updates.
          var parentSuspense = fiber.return;

          if (parentSuspense === null) {
            throw new Error('We just came from a parent so we must have had a parent. This
  is a bug in React.');
          }

          parentSuspense.lanes = mergeLanes(parentSuspense.lanes, renderLanes);
          var _alternate = parentSuspense.alternate;

          if (_alternate !== null) {
            _alternate.lanes = mergeLanes(_alternate.lanes, renderLanes);
          } // This is intentionally passing this fiber as the parent
          // because we want to schedule this fiber as having work
          // on its children. We'll use the childLanes on
          // this fiber to indicate that a context has changed.


          scheduleContextWorkOnParentPath(parentSuspense, renderLanes, workInProgress);
          nextFiber = fiber.sibling;
        } else {
          // Traverse down.
          nextFiber = fiber.child;
        }

        if (nextFiber !== null) {
          // Set the return pointer of the child to the work-in-progress fiber.
          nextFiber.return = fiber;
        } else {
          // No child. Traverse to next sibling.
          nextFiber = fiber;

          while (nextFiber !== null) {
            if (nextFiber === workInProgress) {
              // We're back to the root of this subtree. Exit.
              nextFiber = null;
              break;
            }

            var sibling = nextFiber.sibling;

            if (sibling !== null) {
              // Set the return pointer of the sibling to the work-in-progress fiber.
              sibling.return = nextFiber.return;
              nextFiber = sibling;
              break;
            } // No more siblings. Traverse up.
            nextFiber = nextFiber.return;
          }
        }

        fiber = nextFiber;
      }
    }
    function prepareToReadContext(workInProgress, renderLanes) {
      currentlyRenderingFiber = workInProgress;
      lastContextDependency = null;
      lastFullyObservedContext = null;
      var dependencies = workInProgress.dependencies;

      if (dependencies !== null) {
        {
          var firstContext = dependencies.firstContext;

          if (firstContext !== null) {
            if (includesSomeLane(dependencies.lanes, renderLanes)) {
              // Context list has a pending update. Mark that this fiber performed work.
              markWorkInProgressReceivedUpdate();
            } // Reset the work-in-progress list


            dependencies.firstContext = null;
          }
        }
      }
    }
    function readContext(context) {
      {
        // This warning would fire if you read context inside a Hook like useMemo.
        // Unlike the class check below, it's not enforced in production for perf.
        if (isDisallowedContextReadInDEV) {
          error('Context can only be read while React is rendering. ' + 'In classes, you
  can read it in the render method or getDerivedStateFromProps. ' + 'In function
  components, you can read it directly in the function body, but not ' + 'inside Hooks like
  useReducer() or useMemo().');
        }
      }

      var value =  context._currentValue ;

      if (lastFullyObservedContext === context) ; else {
        var contextItem = {
          context: context,
          memoizedValue: value,
          next: null
        };

        if (lastContextDependency === null) {
          if (currentlyRenderingFiber === null) {
            throw new Error('Context can only be read while React is rendering. ' + 'In
  classes, you can read it in the render method or getDerivedStateFromProps. ' + 'In
  function components, you can read it directly in the function body, but not ' + 'inside
  Hooks like useReducer() or useMemo().');
          } // This is the first dependency for this component. Create a new list.


          lastContextDependency = contextItem;
          currentlyRenderingFiber.dependencies = {
            lanes: NoLanes,
            firstContext: contextItem
          };
        } else {
          // Append a new context item.
          lastContextDependency = lastContextDependency.next = contextItem;
        }
      }

      return value;
    }

    // render. When this render exits, either because it finishes or because it is
    // interrupted, the interleaved updates will be transferred onto the main part
    // of the queue.

    var concurrentQueues = null;
    function pushConcurrentUpdateQueue(queue) {
      if (concurrentQueues === null) {
        concurrentQueues = [queue];
      } else {
        concurrentQueues.push(queue);
      }
    }
    function finishQueueingConcurrentUpdates() {
      // Transfer the interleaved updates onto the main queue. Each queue has a
      // `pending` field and an `interleaved` field. When they are not null, they
      // point to the last node in a circular linked list. We need to append the
      // interleaved list to the end of the pending list by joining them into a
      // single, circular list.
      if (concurrentQueues !== null) {
        for (var i = 0; i < concurrentQueues.length; i++) {
          var queue = concurrentQueues[i];
          var lastInterleavedUpdate = queue.interleaved;

          if (lastInterleavedUpdate !== null) {
            queue.interleaved = null;
            var firstInterleavedUpdate = lastInterleavedUpdate.next;
            var lastPendingUpdate = queue.pending;

            if (lastPendingUpdate !== null) {
              var firstPendingUpdate = lastPendingUpdate.next;
              lastPendingUpdate.next = firstInterleavedUpdate;
              lastInterleavedUpdate.next = firstPendingUpdate;
            }

            queue.pending = lastInterleavedUpdate;
          }
        }

        concurrentQueues = null;
      }
    }
    function enqueueConcurrentHookUpdate(fiber, queue, update, lane) {
      var interleaved = queue.interleaved;

      if (interleaved === null) {
        // This is the first update. Create a circular list.
        update.next = update; // At the end of the current render, this queue's interleaved
  updates will
        // be transferred to the pending queue.

        pushConcurrentUpdateQueue(queue);
      } else {
        update.next = interleaved.next;
        interleaved.next = update;
      }

      queue.interleaved = update;
      return markUpdateLaneFromFiberToRoot(fiber, lane);
    }
    function enqueueConcurrentHookUpdateAndEagerlyBailout(fiber, queue, update, lane) {
      var interleaved = queue.interleaved;

      if (interleaved === null) {
```

```
      // This is the first update. Create a circular list.
      update.next = update; // At the end of the current render, this queue's interleaved
updates will
      // be transferred to the pending queue.

      pushConcurrentUpdateQueue(queue);
    } else {
      update.next = interleaved.next;
      interleaved.next = update;
    }

    queue.interleaved = update;
  }
  function enqueueConcurrentClassUpdate(fiber, queue, update, lane) {
    var interleaved = queue.interleaved;

    if (interleaved === null) {
      // This is the first update. Create a circular list.
      update.next = update; // At the end of the current render, this queue's interleaved
updates will
      // be transferred to the pending queue.

      pushConcurrentUpdateQueue(queue);
    } else {
      update.next = interleaved.next;
      interleaved.next = update;
    }

    queue.interleaved = update;
    return markUpdateLaneFromFiberToRoot(fiber, lane);
  }
  function enqueueConcurrentRenderForLane(fiber, lane) {
    return markUpdateLaneFromFiberToRoot(fiber, lane);
  } // Calling this function outside this module should only be done for backwards
  // compatibility and should always be accompanied by a warning.

  var unsafe_markUpdateLaneFromFiberToRoot = markUpdateLaneFromFiberToRoot;

  function markUpdateLaneFromFiberToRoot(sourceFiber, lane) {
    // Update the source fiber's lanes
    sourceFiber.lanes = mergeLanes(sourceFiber.lanes, lane);
    var alternate = sourceFiber.alternate;

    if (alternate !== null) {
      alternate.lanes = mergeLanes(alternate.lanes, lane);
    }

    {
      if (alternate === null && (sourceFiber.flags & (Placement | Hydrating)) !==
NoFlags) {
        warnAboutUpdateOnNotYetMountedFiberInDEV(sourceFiber);
      }
    } // Walk the parent path to the root and update the child lanes.


    var node = sourceFiber;
    var parent = sourceFiber.return;

    while (parent !== null) {
      parent.childLanes = mergeLanes(parent.childLanes, lane);
      alternate = parent.alternate;

      if (alternate !== null) {
        alternate.childLanes = mergeLanes(alternate.childLanes, lane);
      } else {
        {
          if ((parent.flags & (Placement | Hydrating)) !== NoFlags) {
            warnAboutUpdateOnNotYetMountedFiberInDEV(sourceFiber);
          }
        }
      }

      node = parent;
      parent = parent.return;
    }

    if (node.tag === HostRoot) {
      var root = node.stateNode;
      return root;
    } else {
      return null;
    }
  }

  var UpdateState = 0;
  var ReplaceState = 1;
  var ForceUpdate = 2;
  var CaptureUpdate = 3; // Global state that is reset at the beginning of calling
`processUpdateQueue`.
  // It should only be read right after calling `processUpdateQueue`, via
  // `checkHasForceUpdateAfterProcessing`.

  var hasForceUpdate = false;
  var didWarnUpdateInsideUpdate;
  var currentlyProcessingQueue;

  {
    didWarnUpdateInsideUpdate = false;
    currentlyProcessingQueue = null;
  }

  function initializeUpdateQueue(fiber) {
    var queue = {
      baseState: fiber.memoizedState,
      firstBaseUpdate: null,
      lastBaseUpdate: null,
      shared: {
        pending: null,
        interleaved: null,
        lanes: NoLanes
      },
      effects: null
    };
    fiber.updateQueue = queue;
  }
  function cloneUpdateQueue(current, workInProgress) {
    // Clone the update queue from current. Unless it's already a clone.
    var queue = workInProgress.updateQueue;
    var currentQueue = current.updateQueue;

    if (queue === currentQueue) {
      var clone = {
        baseState: currentQueue.baseState,
        firstBaseUpdate: currentQueue.firstBaseUpdate,
        lastBaseUpdate: currentQueue.lastBaseUpdate,
        shared: currentQueue.shared,
        effects: currentQueue.effects
      };
      workInProgress.updateQueue = clone;
    }
  }
  function createUpdate(eventTime, lane) {
    var update = {
      eventTime: eventTime,
      lane: lane,
      tag: UpdateState,
      payload: null,
      callback: null,
      next: null
    };
    return update;
  }
  function enqueueUpdate(fiber, update, lane) {
    var updateQueue = fiber.updateQueue;

    if (updateQueue === null) {
      // Only occurs if the fiber has been unmounted.
      return null;
    }

    var sharedQueue = updateQueue.shared;

    {
      if (currentlyProcessingQueue === sharedQueue && !didWarnUpdateInsideUpdate) {
        error('An update (setState, replaceState, or forceUpdate) was scheduled ' + 'from
inside an update function. Update functions should be pure, ' + 'with zero side-effects.
Consider using componentDidUpdate or a ' + 'callback.');

        didWarnUpdateInsideUpdate = true;
      }
    }

    if (isUnsafeClassRenderPhaseUpdate()) {
      // This is an unsafe render phase update. Add directly to the update
      // queue so we can process it immediately during the current render.
      var pending = sharedQueue.pending;

      if (pending === null) {
        // This is the first update. Create a circular list.
        update.next = update;
      } else {
        update.next = pending.next;
        pending.next = update;
      }

      sharedQueue.pending = update; // Update the childLanes even though we're most
likely already rendering
      // this fiber. This is for backwards compatibility in the case where you
      // update a different component during render phase than the one that is
      // currently renderings (a pattern that is accompanied by a warning).

      return unsafe_markUpdateLaneFromFiberToRoot(fiber, lane);
    } else {
      return enqueueConcurrentClassUpdate(fiber, sharedQueue, update, lane);
    }
  }
  function entangleTransitions(root, fiber, lane) {
    var updateQueue = fiber.updateQueue;

    if (updateQueue === null) {
      // Only occurs if the fiber has been unmounted.
      return;
    }

    var sharedQueue = updateQueue.shared;

    if (isTransitionLane(lane)) {
      var queueLanes = sharedQueue.lanes; // If any entangled lanes are no longer pending
on the root, then they must
      // have finished. We can remove them from the shared queue, which represents
      // a superset of the actually pending lanes. In some cases we may entangle
      // more than we need to, but that's OK. In fact it's worse if we *don't*
      // entangle when we should.

      queueLanes = intersectLanes(queueLanes, root.pendingLanes); // Entangle the new
transition lane with the other transition lanes.
```

```
      var newQueueLanes = mergeLanes(queueLanes, lane);
      sharedQueue.lanes = newQueueLanes; // Even if queue.lanes already include lane, we
don't know for certain if
      // the lane finished since the last time we entangled it. So we need to
      // entangle it again, just to be sure.

      markRootEntangled(root, newQueueLanes);
    }
  }
  function enqueueCapturedUpdate(workInProgress, capturedUpdate) {
    // Captured updates are updates that are thrown by a child during the render
    // phase. They should be discarded if the render is aborted. Therefore,
    // we should only put them on the work-in-progress queue, not the current one.
    var queue = workInProgress.updateQueue; // Check if the work-in-progress queue is a
clone.

    var current = workInProgress.alternate;

    if (current !== null) {
      var currentQueue = current.updateQueue;

      if (queue === currentQueue) {
        // The work-in-progress queue is the same as current. This happens when
        // we bail out on a parent fiber that then captures an error thrown by
        // a child. Since we want to append the update only to the work-in
        // -progress queue, we need to clone the updates. We usually clone during
        // processUpdateQueue, but that didn't happen in this case because we
        // skipped over the parent when we bailed out.
        var newFirst = null;
        var newLast = null;
        var firstBaseUpdate = queue.firstBaseUpdate;

        if (firstBaseUpdate !== null) {
          // Loop through the updates and clone them.
          var update = firstBaseUpdate;

          do {
            var clone = {
              eventTime: update.eventTime,
              lane: update.lane,
              tag: update.tag,
              payload: update.payload,
              callback: update.callback,
              next: null
            };

            if (newLast === null) {
              newFirst = newLast = clone;
            } else {
              newLast.next = clone;
              newLast = clone;
            }

            update = update.next;
          } while (update !== null); // Append the captured update the end of the cloned
list.


          if (newLast === null) {
            newFirst = newLast = capturedUpdate;
          } else {
            newLast.next = capturedUpdate;
            newLast = capturedUpdate;
          }
        } else {
          // There are no base updates.
          newFirst = newLast = capturedUpdate;
        }
```

```
          queue = {
            baseState: currentQueue.baseState,
            firstBaseUpdate: newFirst,
            lastBaseUpdate: newLast,
            shared: currentQueue.shared,
            effects: currentQueue.effects
          };
          workInProgress.updateQueue = queue;
          return;
        }
      } // Append the update to the end of the list.


      var lastBaseUpdate = queue.lastBaseUpdate;

      if (lastBaseUpdate === null) {
        queue.firstBaseUpdate = capturedUpdate;
      } else {
        lastBaseUpdate.next = capturedUpdate;
      }

      queue.lastBaseUpdate = capturedUpdate;
    }

    function getStateFromUpdate(workInProgress, queue, update, prevState, nextProps,
  instance) {
      switch (update.tag) {
        case ReplaceState:
          {
            var payload = update.payload;

            if (typeof payload === 'function') {
              // Updater function
              {
                enterDisallowedContextReadInDEV();
              }

              var nextState = payload.call(instance, prevState, nextProps);

              {
                if ( workInProgress.mode & StrictLegacyMode) {
                  setIsStrictModeForDevtools(true);

                  try {
                    payload.call(instance, prevState, nextProps);
                  } finally {
                    setIsStrictModeForDevtools(false);
                  }
                }

                exitDisallowedContextReadInDEV();
              }

              return nextState;
            } // State object


            return payload;
          }

        case CaptureUpdate:
          {
            workInProgress.flags = workInProgress.flags & ~ShouldCapture | DidCapture;
          }
        // Intentional fallthrough

        case UpdateState:
          {
            var _payload = update.payload;
            var partialState;

            if (typeof _payload === 'function') {
              // Updater function
              {
                enterDisallowedContextReadInDEV();
              }

              partialState = _payload.call(instance, prevState, nextProps);

              {
                if ( workInProgress.mode & StrictLegacyMode) {
                  setIsStrictModeForDevtools(true);

                  try {
                    _payload.call(instance, prevState, nextProps);
                  } finally {
                    setIsStrictModeForDevtools(false);
                  }
                }

                exitDisallowedContextReadInDEV();
              }
            } else {
              // Partial state object
              partialState = _payload;
            }

            if (partialState === null || partialState === undefined) {
              // Null and undefined are treated as no-ops.
              return prevState;
            } // Merge the partial state and the previous state.


            return assign({}, prevState, partialState);
          }

        case ForceUpdate:
          {
            hasForceUpdate = true;
            return prevState;
          }
      }

      return prevState;
    }

    function processUpdateQueue(workInProgress, props, instance, renderLanes) {
      // This is always non-null on a ClassComponent or HostRoot
      var queue = workInProgress.updateQueue;
      hasForceUpdate = false;

      {
        currentlyProcessingQueue = queue.shared;
      }

      var firstBaseUpdate = queue.firstBaseUpdate;
      var lastBaseUpdate = queue.lastBaseUpdate; // Check if there are pending updates. If
  so, transfer them to the base queue.

      var pendingQueue = queue.shared.pending;

      if (pendingQueue !== null) {
        queue.shared.pending = null; // The pending queue is circular. Disconnect the
  pointer between first
        // and last so that it's non-circular.

        var lastPendingUpdate = pendingQueue;
        var firstPendingUpdate = lastPendingUpdate.next;
        lastPendingUpdate.next = null; // Append pending updates to base queue

        if (lastBaseUpdate === null) {
          firstBaseUpdate = firstPendingUpdate;
        } else {
          lastBaseUpdate.next = firstPendingUpdate;
        }

        lastBaseUpdate = lastPendingUpdate; // If there's a current queue, and it's
  different from the base queue, then
        // we need to transfer the updates to that queue, too. Because the base
        // queue is a singly-linked list with no cycles, we can append to both
        // lists and take advantage of structural sharing.
        // TODO: Pass `current` as argument

        var current = workInProgress.alternate;

        if (current !== null) {
          // This is always non-null on a ClassComponent or HostRoot
          var currentQueue = current.updateQueue;
          var currentLastBaseUpdate = currentQueue.lastBaseUpdate;

          if (currentLastBaseUpdate !== lastBaseUpdate) {
            if (currentLastBaseUpdate === null) {
              currentQueue.firstBaseUpdate = firstPendingUpdate;
            } else {
              currentLastBaseUpdate.next = firstPendingUpdate;
            }

            currentQueue.lastBaseUpdate = lastPendingUpdate;
          }
        }
      } // These values may change as we process the queue.


      if (firstBaseUpdate !== null) {
        // Iterate through the list of updates to compute the result.
        var newState = queue.baseState; // TODO: Don't need to accumulate this. Instead, we
  can remove renderLanes
        // from the original lanes.

        var newLanes = NoLanes;
        var newBaseState = null;
        var newFirstBaseUpdate = null;
        var newLastBaseUpdate = null;
        var update = firstBaseUpdate;

        do {
          var updateLane = update.lane;
          var updateEventTime = update.eventTime;

          if (!isSubsetOfLanes(renderLanes, updateLane)) {
            // Priority is insufficient. Skip this update. If this is the first
            // skipped update, the previous update/state is the new base
            // update/state.
            var clone = {
              eventTime: updateEventTime,
              lane: updateLane,
              tag: update.tag,
              payload: update.payload,
              callback: update.callback,
              next: null
            };

            if (newLastBaseUpdate === null) {
              newFirstBaseUpdate = newLastBaseUpdate = clone;
              newBaseState = newState;
            } else {
              newLastBaseUpdate = newLastBaseUpdate.next = clone;
            } // Update the remaining priority in the queue.


            newLanes = mergeLanes(newLanes, updateLane);
          } else {
            // This update does have sufficient priority.
            if (newLastBaseUpdate !== null) {
              var _clone = {
                eventTime: updateEventTime,
                // This update is going to be committed so we never want uncommit
                // it. Using NoLane works because 0 is a subset of all bitmasks, so
                // this will never be skipped by the check above.
                lane: NoLane,
                tag: update.tag,
                payload: update.payload,
                callback: update.callback,
                next: null
              };
              newLastBaseUpdate = newLastBaseUpdate.next = _clone;
            } // Process this update.


            newState = getStateFromUpdate(workInProgress, queue, update, newState, props,
  instance);
            var callback = update.callback;

            if (callback !== null && // If the update was already committed, we should not
  queue its
            // callback again.
            update.lane !== NoLane) {
              workInProgress.flags |= Callback;
              var effects = queue.effects;

              if (effects === null) {
                queue.effects = [update];
              } else {
                effects.push(update);
              }
            }
          }

          update = update.next;

          if (update === null) {
            pendingQueue = queue.shared.pending;

            if (pendingQueue === null) {
              break;
            } else {
              // An update was scheduled from inside a reducer. Add the new
              // pending updates to the end of the list and keep processing.
              var _lastPendingUpdate = pendingQueue; // Intentionally unsound. Pending
  updates form a circular list, but we
              // unravel them when transferring them to the base queue.

              var _firstPendingUpdate = _lastPendingUpdate.next;
              _lastPendingUpdate.next = null;
              update = _firstPendingUpdate;
              queue.lastBaseUpdate = _lastPendingUpdate;
              queue.shared.pending = null;
            }
          }
        } while (true);

        if (newLastBaseUpdate === null) {
          newBaseState = newState;
        }

        queue.baseState = newBaseState;
        queue.firstBaseUpdate = newFirstBaseUpdate;
        queue.lastBaseUpdate = newLastBaseUpdate; // Interleaved updates are stored on a
  separate queue. We aren't going to
        // process them during this render, but we do need to track which lanes
        // are remaining.

        var lastInterleaved = queue.shared.interleaved;

        if (lastInterleaved !== null) {
          var interleaved = lastInterleaved;

          do {
            newLanes = mergeLanes(newLanes, interleaved.lane);
            interleaved = interleaved.next;
          } while (interleaved !== lastInterleaved);
        } else if (firstBaseUpdate === null) {
          // `queue.lanes` is used for entangling transitions. We can set it back to
          // zero once the queue is empty.
          queue.shared.lanes = NoLanes;
        } // Set the remaining expiration time to be whatever is remaining in the queue.
        // This should be fine because the only two other things that contribute to
        // expiration time are props and context. We're already in the middle of the
        // begin phase by the time we start processing the queue, so we've already
        // dealt with the props. Context in components that specify
        // shouldComponentUpdate is tricky; but we'll have to account for
        // that regardless.


        markSkippedUpdateLanes(newLanes);
        workInProgress.lanes = newLanes;
        workInProgress.memoizedState = newState;
      }

      {
        currentlyProcessingQueue = null;
      }
    }

    function callCallback(callback, context) {
      if (typeof callback !== 'function') {
        throw new Error('Invalid argument passed as callback. Expected a function. Instead
  ' + ("received: " + callback));
      }

      callback.call(context);
    }

    function resetHasForceUpdateBeforeProcessing() {
      hasForceUpdate = false;
    }
    function checkHasForceUpdateAfterProcessing() {
      return hasForceUpdate;
    }
    function commitUpdateQueue(finishedWork, finishedQueue, instance) {
      // Commit the effects
      var effects = finishedQueue.effects;
      finishedQueue.effects = null;

      if (effects !== null) {
        for (var i = 0; i < effects.length; i++) {
          var effect = effects[i];
          var callback = effect.callback;

          if (callback !== null) {
            effect.callback = null;
            callCallback(callback, instance);
          }
        }
      }
    }

    var NO_CONTEXT = {};
    var contextStackCursor$1 = createCursor(NO_CONTEXT);
    var contextFiberStackCursor = createCursor(NO_CONTEXT);
    var rootInstanceStackCursor = createCursor(NO_CONTEXT);

    function requiredContext(c) {
      if (c === NO_CONTEXT) {
        throw new Error('Expected host context to exist. This error is likely caused by a
  bug ' + 'in React. Please file an issue.');
      }

      return c;
    }

    function getRootHostContainer() {
      var rootInstance = requiredContext(rootInstanceStackCursor.current);
      return rootInstance;
    }

    function pushHostContainer(fiber, nextRootInstance) {
      // Push current root instance onto the stack;
      // This allows us to reset root when portals are popped.
      push(rootInstanceStackCursor, nextRootInstance, fiber); // Track the context and the
  Fiber that provided it.
      // This enables us to pop only Fibers that provide unique contexts.

      push(contextFiberStackCursor, fiber, fiber); // Finally, we need to push the host
  context to the stack.
      // However, we can't just call getRootHostContext() and push it because
      // we'd have a different number of entries on the stack depending on
      // whether getRootHostContext() throws somewhere in renderer code or not.
      // So we push an empty value first. This lets us safely unwind on errors.

      push(contextStackCursor$1, NO_CONTEXT, fiber);
      var nextRootContext = getRootHostContext(nextRootInstance); // Now that we know this
  function doesn't throw, replace it.

      pop(contextStackCursor$1, fiber);
      push(contextStackCursor$1, nextRootContext, fiber);
    }

    function popHostContainer(fiber) {
      pop(contextStackCursor$1, fiber);
      pop(contextFiberStackCursor, fiber);
      pop(rootInstanceStackCursor, fiber);
    }

    function getHostContext() {
      var context = requiredContext(contextStackCursor$1.current);
      return context;
    }

    function pushHostContext(fiber) {
      var rootInstance = requiredContext(rootInstanceStackCursor.current);
      var context = requiredContext(contextStackCursor$1.current);
      var nextContext = getChildHostContext(context, fiber.type); // Don't push this
  Fiber's context unless it's unique.

      if (context === nextContext) {
        return;
      } // Track the context and the Fiber that provided it.
      // This enables us to pop only Fibers that provide unique contexts.


      push(contextFiberStackCursor, fiber, fiber);
      push(contextStackCursor$1, nextContext, fiber);
    }
```

```
    function popHostContext(fiber) {
      // Do not pop unless this Fiber provided the current context.
      // pushHostContext() only pushes Fibers that provide unique contexts.
      if (contextFiberStackCursor.current !== fiber) {
        return;
      }

      pop(contextStackCursor$1, fiber);
      pop(contextFiberStackCursor, fiber);
    }

    var DefaultSuspenseContext = 0; // The Suspense Context is split into two parts. The
  lower bits is
    // inherited deeply down the subtree. The upper bits only affect
    // this immediate suspense boundary and gets reset each new
    // boundary or suspense list.

    var SubtreeSuspenseContextMask = 1; // Subtree Flags:
    // InvisibleParentSuspenseContext indicates that one of our parent Suspense
    // boundaries is not currently showing visible main content.
    // Either because it is already showing a fallback or is not mounted at all.
    // We can use this to determine if it is desirable to trigger a fallback at
    // the parent. If not, then we might need to trigger undesirable boundaries
    // and/or suspend the commit to avoid hiding the parent content.

    var InvisibleParentSuspenseContext = 1; // Shallow Flags:
    // ForceSuspenseFallback can be used by SuspenseList to force newly added
    // items into their fallback state during one of the render passes.

    var ForceSuspenseFallback = 2;
    var suspenseStackCursor = createCursor(DefaultSuspenseContext);
    function hasSuspenseContext(parentContext, flag) {
      return (parentContext & flag) !== 0;
    }
    function setDefaultShallowSuspenseContext(parentContext) {
      return parentContext & SubtreeSuspenseContextMask;
    }
    function setShallowSuspenseContext(parentContext, shallowContext) {
      return parentContext & SubtreeSuspenseContextMask | shallowContext;
    }
    function addSubtreeSuspenseContext(parentContext, subtreeContext) {
      return parentContext | subtreeContext;
    }
    function pushSuspenseContext(fiber, newContext) {
      push(suspenseStackCursor, newContext, fiber);
    }
    function popSuspenseContext(fiber) {
      pop(suspenseStackCursor, fiber);
    }

    function shouldCaptureSuspense(workInProgress, hasInvisibleParent) {
      // If it was the primary children that just suspended, capture and render the
      // fallback. Otherwise, don't capture and bubble to the next boundary.
      var nextState = workInProgress.memoizedState;

      if (nextState !== null) {
        if (nextState.dehydrated !== null) {
          // A dehydrated boundary always captures.
          return true;
        }

        return false;
      }

      var props = workInProgress.memoizedProps; // Regular boundaries always capture.

      {
        return true;
      } // If it's a boundary we should avoid, then we prefer to bubble up to the
    }
    function findFirstSuspended(row) {
      var node = row;

      while (node !== null) {
        if (node.tag === SuspenseComponent) {
          var state = node.memoizedState;

          if (state !== null) {
            var dehydrated = state.dehydrated;

            if (dehydrated === null || isSuspenseInstancePending(dehydrated) ||
  isSuspenseInstanceFallback(dehydrated)) {
              return node;
            }
          }
        } else if (node.tag === SuspenseListComponent && // revealOrder undefined can't be
  trusted because it don't
        // keep track of whether it suspended or not.
        node.memoizedProps.revealOrder !== undefined) {
          var didSuspend = (node.flags & DidCapture) !== NoFlags;

          if (didSuspend) {
            return node;
          }
        } else if (node.child !== null) {
          node.child.return = node;
          node = node.child;
          continue;
        }

        if (node === row) {
          return null;
        }

        while (node.sibling === null) {
          if (node.return === null || node.return === row) {
            return null;
          }

          node = node.return;
        }

        node.sibling.return = node.return;
        node = node.sibling;
      }

      return null;
    }

    var NoFlags$1 =
    /*   */
    0; // Represents whether effect should fire.

    var HasEffect =
    /* */
    1; // Represents the phase in which the effect (not the clean-up) fires.

    var Insertion =
    /*  */
    2;
    var Layout =
    /*    */
    4;
    var Passive$1 =
    /*   */
    8;
    // and should be reset before starting a new render.
    // This tracks which mutable sources need to be reset after a render.

    var workInProgressSources = [];
    function resetWorkInProgressVersions() {
      for (var i = 0; i < workInProgressSources.length; i++) {
        var mutableSource = workInProgressSources[i];

        {
          mutableSource._workInProgressVersionPrimary = null;
        }
      }

      workInProgressSources.length = 0;
    }
    // This ensures that the version used for server rendering matches the one
    // that is eventually read during hydration.
    // If they don't match there's a potential tear and a full deopt render is required.

    function registerMutableSourceForHydration(root, mutableSource) {
      var getVersion = mutableSource._getVersion;
      var version = getVersion(mutableSource._source); // TODO Clear this data once all
  pending hydration work is finished.
      // Retaining it forever may interfere with GC.

      if (root.mutableSourceEagerHydrationData == null) {
        root.mutableSourceEagerHydrationData = [mutableSource, version];
      } else {
        root.mutableSourceEagerHydrationData.push(mutableSource, version);
      }
    }

    var ReactCurrentDispatcher$1 = ReactSharedInternals.ReactCurrentDispatcher,
        ReactCurrentBatchConfig$2 = ReactSharedInternals.ReactCurrentBatchConfig;
    var didWarnAboutMismatchedHooksForComponent;
    var didWarnUncachedGetSnapshot;

    {
      didWarnAboutMismatchedHooksForComponent = new Set();
    }

    // These are set right before calling the component.
    var renderLanes = NoLanes; // The work-in-progress fiber. I've named it differently to
  distinguish it from
    // the work-in-progress hook.

    var currentlyRenderingFiber$1 = null; // Hooks are stored as a linked list on the
  fiber's memoizedState field. The
    // current hook list is the list that belongs to the current fiber. The
    // work-in-progress hook list is a new list that will be added to the
    // work-in-progress fiber.

    var currentHook = null;
    var workInProgressHook = null; // Whether an update was scheduled at any point during
  the render phase. This
    // does not get reset if we do another render pass; only when we're completely
    // finished evaluating this component. This is an optimization so we know
    // whether we need to clear render phase updates after a throw.

    var didScheduleRenderPhaseUpdate = false; // Where an update was scheduled only during
  the current render pass. This
    // gets reset after each attempt.
    // TODO: Maybe there's some way to consolidate this with
    // `didScheduleRenderPhaseUpdate`. Or with `numberOfReRenders`.

    var didScheduleRenderPhaseUpdateDuringThisPass = false; // Counts the number of useId
  hooks in this component.

    var localIdCounter = 0; // Used for ids that are generated completely client-side (i.e.
  not during
    // hydration). This counter is global, so client ids are not stable across
    // render attempts.

    var globalClientIdCounter = 0;
    var RE_RENDER_LIMIT = 25; // In DEV, this is the name of the currently executing
  primitive hook

    var currentHookNameInDev = null; // In DEV, this list ensures that hooks are called in
  the same order between renders.
    // The list stores the order of hooks used during the initial render (mount).
    // Subsequent renders (updates) reference this list.

    var hookTypesDev = null;
    var hookTypesUpdateIndexDev = -1; // In DEV, this tracks whether currently rendering
  component needs to ignore
    // the dependencies for Hooks that need them (e.g. useEffect or useMemo).
    // When true, such Hooks will always be "remounted". Only used during hot reload.

    var ignorePreviousDependencies = false;

    function mountHookTypesDev() {
      {
        var hookName = currentHookNameInDev;

        if (hookTypesDev === null) {
          hookTypesDev = [hookName];
        } else {
          hookTypesDev.push(hookName);
        }
      }
    }

    function updateHookTypesDev() {
      {
        var hookName = currentHookNameInDev;

        if (hookTypesDev !== null) {
          hookTypesUpdateIndexDev++;

          if (hookTypesDev[hookTypesUpdateIndexDev] !== hookName) {
            warnOnHookMismatchInDev(hookName);
          }
        }
      }
    }

    function checkDepsAreArrayDev(deps) {
      {
        if (deps !== undefined && deps !== null && !isArray(deps)) {
          // Verify deps, but only on mount to avoid extra checks.
          // It's unlikely their type would change as usually you define them inline.
          error('%s received a final argument that is not an array (instead, received
  `%s`). When ' + 'specified, the final argument must be an array.', currentHookNameInDev,
  typeof deps);
        }
      }
    }

    function warnOnHookMismatchInDev(currentHookName) {
      {
        var componentName = getComponentNameFromFiber(currentlyRenderingFiber$1);

        if (!didWarnAboutMismatchedHooksForComponent.has(componentName)) {
          didWarnAboutMismatchedHooksForComponent.add(componentName);

          if (hookTypesDev !== null) {
            var table = '';
            var secondColumnStart = 30;
            for (var i = 0; i <= hookTypesUpdateIndexDev; i++) {
              var oldHookName = hookTypesDev[i];
              var newHookName = i === hookTypesUpdateIndexDev ? currentHookName :
  oldHookName;
              var row = i + 1 + ". " + oldHookName; // Extra space so second column lines
  up
              // lol @ IE not supporting String#repeat

              while (row.length < secondColumnStart) {
                row += ' ';
              }

              row += newHookName + '\n';
              table += row;
            }

            error('React has detected a change in the order of Hooks called by %s. ' +
  'This will lead to bugs and errors if not fixed. ' + 'For more information, read the
  Rules of Hooks: https://reactjs.org/link/rules-of-hooks\n\n' + '   Previous render
  Next render\n' + '   ------------------------------------------------------\n' + '%s' + '
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n', componentName, table);
          }
        }
      }
    }

    function throwInvalidHookError() {
      throw new Error('Invalid hook call. Hooks can only be called inside of the body of a
  function component. This could happen for' + ' one of the following reasons:\n' + '1. You
  might have mismatching versions of React and the renderer (such as React DOM)\n' + '2.
  You might be breaking the Rules of Hooks\n' + '3. You might have more than one copy of
  React in the same app\n' + 'See https://reactjs.org/link/invalid-hook-call for tips about
  how to debug and fix this problem.');
    }

    function areHookInputsEqual(nextDeps, prevDeps) {
      {
        if (ignorePreviousDependencies) {
          // Only true when this component is being hot reloaded.
          return false;
        }
      }

      if (prevDeps === null) {
        {
          error('%s received a final argument during this render, but not during ' + 'the
  previous render. Even though the final argument is optional, ' + 'its type cannot change
  between renders.', currentHookNameInDev);
        }

        return false;
      }

      {
        // Don't bother comparing lengths in prod because these arrays should be
        // passed inline.
        if (nextDeps.length !== prevDeps.length) {
          error('The final argument passed to %s changed size between renders. The ' +
  'order and size of this array must remain constant.\n\n' + 'Previous: %s\n' + 'Incoming:
  %s', currentHookNameInDev, "[" + prevDeps.join(', ') + "]", "[" + nextDeps.join(', ') +
  "]");
        }
      }

      for (var i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
        if (objectIs(nextDeps[i], prevDeps[i])) {
          continue;
        }
        return false;
      }

      return true;
    }

    function renderWithHooks(current, workInProgress, Component, props, secondArg,
  nextRenderLanes) {
      renderLanes = nextRenderLanes;
      currentlyRenderingFiber$1 = workInProgress;

      {
        hookTypesDev = current !== null ? current._debugHookTypes : null;
        hookTypesUpdateIndexDev = -1; // Used for hot reloading:

        ignorePreviousDependencies = current !== null && current.type !==
  workInProgress.type;
      }

      workInProgress.memoizedState = null;
      workInProgress.updateQueue = null;
      workInProgress.lanes = NoLanes; // The following should have already been reset
      // currentHook = null;
      // workInProgressHook = null;
      // didScheduleRenderPhaseUpdate = false;
      // localIdCounter = 0;
      // TODO Warn if no hooks are used at all during mount, then some are used during
  update.
      // Currently we will identify the update render as a mount because memoizedState ===
  null.
      // This is tricky because it's valid for certain types of components (e.g.
  React.lazy)
      // Using memoizedState to differentiate between mount/update only works if at least
  one stateful hook is used.
      // Non-stateful hooks (e.g. context) don't get added to memoizedState,
      // so memoizedState would be null during updates and mounts.

      {
        if (current !== null && current.memoizedState !== null) {
          ReactCurrentDispatcher$1.current = HooksDispatcherOnUpdateInDEV;
        } else if (hookTypesDev !== null) {
          // This dispatcher handles an edge case where a component is updating,
          // but no stateful hooks have been used.
          // We want to match the production code behavior (which will use
  HooksDispatcherOnMount),
          // but with the extra DEV validation to ensure hooks ordering hasn't changed.
          // This dispatcher does that.
          ReactCurrentDispatcher$1.current = HooksDispatcherOnMountWithHookTypesInDEV;
        } else {
          ReactCurrentDispatcher$1.current = HooksDispatcherOnMountInDEV;
        }
      }

      var children = Component(props, secondArg); // Check if there was a render phase
  update

      if (didScheduleRenderPhaseUpdateDuringThisPass) {
        // Keep rendering in a loop for as long as render phase updates continue to
        // be scheduled. Use a counter to prevent infinite loops.
        var numberOfReRenders = 0;

        do {
          didScheduleRenderPhaseUpdateDuringThisPass = false;
          localIdCounter = 0;

          if (numberOfReRenders >= RE_RENDER_LIMIT) {
            throw new Error('Too many re-renders. React limits the number of renders to
  prevent ' + 'an infinite loop.');
```

```
        }

        numberOfReRenders += 1;

        {
          // Even when hot reloading, allow dependencies to stabilize
          // after first render to prevent infinite render phase updates.
          ignorePreviousDependencies = false;
        } // Start over from the beginning of the list


        currentHook = null;
        workInProgressHook = null;
        workInProgress.updateQueue = null;

        {
          // Also validate hook order for cascading updates.
          hookTypesUpdateIndexDev = -1;
        }

        ReactCurrentDispatcher$1.current =  HooksDispatcherOnRerenderInDEV ;
        children = Component(props, secondArg);
      } while (didScheduleRenderPhaseUpdateDuringThisPass);
    } // We can assume the previous dispatcher is always this one, since we set it
    // at the beginning of the render phase and there's no re-entrance.


    ReactCurrentDispatcher$1.current = ContextOnlyDispatcher;

    {
      workInProgress._debugHookTypes = hookTypesDev;
    } // This check uses currentHook so that it works the same in DEV and prod bundles.
    // hookTypesDev could catch more cases (e.g. context) but only in DEV bundles.


    var didRenderTooFewHooks = currentHook !== null && currentHook.next !== null;
    renderLanes = NoLanes;
    currentlyRenderingFiber$1 = null;
    currentHook = null;
    workInProgressHook = null;

    {
      currentHookNameInDev = null;
      hookTypesDev = null;
      hookTypesUpdateIndexDev = -1; // Confirm that a static flag was not added or
removed since the last
      // render. If this fires, it suggests that we incorrectly reset the static
      // flags in some other part of the codebase. This has happened before, for
      // example, in the SuspenseList implementation.

      if (current !== null && (current.flags & StaticMask) !== (workInProgress.flags &
StaticMask) && // Disable this warning in legacy mode, because legacy Suspense is weird
      // and creates false positives. To make this work in legacy mode, we'd
      // need to mark fibers that commit in an incomplete state, somehow. For
      // now I'll disable the warning that most of the bugs that would trigger
      // it are either exclusive to concurrent mode or exist in both.
      (current.mode & ConcurrentMode) !== NoMode) {
        error('Internal React error: Expected static flag was missing. Please ' + 'notify
the React team.');
      }
    }

    didScheduleRenderPhaseUpdate = false; // This is reset by checkDidRenderIdHook
    // localIdCounter = 0;

    if (didRenderTooFewHooks) {
      throw new Error('Rendered fewer hooks than expected. This may be caused by an
accidental ' + 'early return statement.');
    }
```

```
      return children;
    }
    function checkDidRenderIdHook() {
      // This should be called immediately after every renderWithHooks call.
      // Conceptually, it's part of the return value of renderWithHooks; it's only a
      // separate function to avoid using an array tuple.
      var didRenderIdHook = localIdCounter !== 0;
      localIdCounter = 0;
      return didRenderIdHook;
    }
    function bailoutHooks(current, workInProgress, lanes) {
      workInProgress.updateQueue = current.updateQueue; // TODO: Don't need to reset the
  flags here, because they're reset in the
      // complete phase (bubbleProperties).

      if ( (workInProgress.mode & StrictEffectsMode) !== NoMode) {
        workInProgress.flags &= ~(MountPassiveDev | MountLayoutDev | Passive | Update);
      } else {
        workInProgress.flags &= ~(Passive | Update);
      }

      current.lanes = removeLanes(current.lanes, lanes);
    }
    function resetHooksAfterThrow() {
      // We can assume the previous dispatcher is always this one, since we set it
      // at the beginning of the render phase and there's no re-entrance.
      ReactCurrentDispatcher$1.current = ContextOnlyDispatcher;

      if (didScheduleRenderPhaseUpdate) {
        // There were render phase updates. These are only valid for this render
        // phase, which we are now aborting. Remove the updates from the queues so
        // they do not persist to the next render. Do not remove updates from hooks
        // that weren't processed.
        //
        // Only reset the updates from the queue if it has a clone. If it does
        // not have a clone, that means it wasn't processed, and the updates were
        // scheduled before we entered the render phase.
        var hook = currentlyRenderingFiber$1.memoizedState;

        while (hook !== null) {
          var queue = hook.queue;

          if (queue !== null) {
            queue.pending = null;
          }

          hook = hook.next;
        }

        didScheduleRenderPhaseUpdate = false;
      }

      renderLanes = NoLanes;
      currentlyRenderingFiber$1 = null;
      currentHook = null;
      workInProgressHook = null;

      {
        hookTypesDev = null;
        hookTypesUpdateIndexDev = -1;
        currentHookNameInDev = null;
        isUpdatingOpaqueValueInRenderPhase = false;
      }

      didScheduleRenderPhaseUpdateDuringThisPass = false;
      localIdCounter = 0;
    }
    function mountWorkInProgressHook() {
      var hook = {
        memoizedState: null,
        baseState: null,
        baseQueue: null,
        queue: null,
        next: null
      };

      if (workInProgressHook === null) {
        // This is the first hook in the list
        currentlyRenderingFiber$1.memoizedState = workInProgressHook = hook;
      } else {
        // Append to the end of the list
        workInProgressHook = workInProgressHook.next = hook;
      }

      return workInProgressHook;
    }

    function updateWorkInProgressHook() {
      // This function is used both for updates and for re-renders triggered by a
      // render phase update. It assumes there is either a current hook we can
      // clone, or a work-in-progress hook from a previous render pass that we can
      // use as a base. When we reach the end of the base list, we must switch to
      // the dispatcher used for mounts.
      var nextCurrentHook;

      if (currentHook === null) {
        var current = currentlyRenderingFiber$1.alternate;

        if (current !== null) {
          nextCurrentHook = current.memoizedState;
        } else {
          nextCurrentHook = null;
        }
      } else {
        nextCurrentHook = currentHook.next;
      }

      var nextWorkInProgressHook;

      if (workInProgressHook === null) {
        nextWorkInProgressHook = currentlyRenderingFiber$1.memoizedState;
      } else {
        nextWorkInProgressHook = workInProgressHook.next;
      }

      if (nextWorkInProgressHook !== null) {
        // There's already a work-in-progress. Reuse it.
        workInProgressHook = nextWorkInProgressHook;
        nextWorkInProgressHook = workInProgressHook.next;
        currentHook = nextCurrentHook;
      } else {
        // Clone from the current hook.
        if (nextCurrentHook === null) {
          throw new Error('Rendered more hooks than during the previous render.');
        }

        currentHook = nextCurrentHook;
        var newHook = {
          memoizedState: currentHook.memoizedState,
          baseState: currentHook.baseState,
          baseQueue: currentHook.baseQueue,
          queue: currentHook.queue,
          next: null
        };

        if (workInProgressHook === null) {
          // This is the first hook in the list.
          currentlyRenderingFiber$1.memoizedState = workInProgressHook = newHook;
        } else {
          // Append to the end of the list.
          workInProgressHook = workInProgressHook.next = newHook;
        }
      }

      return workInProgressHook;
    }

    function createFunctionComponentUpdateQueue() {
      return {
        lastEffect: null,
        stores: null
      };
    }

    function basicStateReducer(state, action) {
      // $FlowFixMe: Flow doesn't like mixed types
      return typeof action === 'function' ? action(state) : action;
    }

    function mountReducer(reducer, initialArg, init) {
      var hook = mountWorkInProgressHook();
      var initialState;

      if (init !== undefined) {
        initialState = init(initialArg);
      } else {
        initialState = initialArg;
      }

      hook.memoizedState = hook.baseState = initialState;
      var queue = {
        pending: null,
        interleaved: null,
        lanes: NoLanes,
        dispatch: null,
        lastRenderedReducer: reducer,
        lastRenderedState: initialState
      };
      hook.queue = queue;
      var dispatch = queue.dispatch = dispatchReducerAction.bind(null,
  currentlyRenderingFiber$1, queue);
      return [hook.memoizedState, dispatch];
    }

    function updateReducer(reducer, initialArg, init) {
      var hook = updateWorkInProgressHook();
      var queue = hook.queue;

      if (queue === null) {
        throw new Error('Should have a queue. This is likely a bug in React. Please file an
  issue.');
      }

      queue.lastRenderedReducer = reducer;
      var current = currentHook; // The last rebase update that is NOT part of the base
  state.

      var baseQueue = current.baseQueue; // The last pending update that hasn't been
  processed yet.

      var pendingQueue = queue.pending;

      if (pendingQueue !== null) {
        // We have new updates that haven't been processed yet.
        // We'll add them to the base queue.
        if (baseQueue !== null) {
          // Merge the pending queue and the base queue.
          var baseFirst = baseQueue.next;
          var pendingFirst = pendingQueue.next;
          baseQueue.next = pendingFirst;
          pendingQueue.next = baseFirst;
        }

        {
          if (current.baseQueue !== baseQueue) {
            // Internal invariant that should never happen, but feasibly could in
            // the future if we implement resuming, or some form of that.
            error('Internal error: Expected work-in-progress queue to be a clone. ' + 'This
  is a bug in React.');
          }
        }

        current.baseQueue = baseQueue = pendingQueue;
        queue.pending = null;
      }

      if (baseQueue !== null) {
        // We have a queue to process.
        var first = baseQueue.next;
        var newState = current.baseState;
        var newBaseState = null;
        var newBaseQueueFirst = null;
        var newBaseQueueLast = null;
        var update = first;

        do {
          var updateLane = update.lane;

          if (!isSubsetOfLanes(renderLanes, updateLane)) {
            // Priority is insufficient. Skip this update. If this is the first
            // skipped update, the previous update/state is the new base
            // update/state.
            var clone = {
              lane: updateLane,
              action: update.action,
              hasEagerState: update.hasEagerState,
              eagerState: update.eagerState,
              next: null
            };

            if (newBaseQueueLast === null) {
              newBaseQueueFirst = newBaseQueueLast = clone;
              newBaseState = newState;
            } else {
              newBaseQueueLast = newBaseQueueLast.next = clone;
            } // Update the remaining priority in the queue.
            // TODO: Don't need to accumulate this. Instead, we can remove
            // renderLanes from the original lanes.


            currentlyRenderingFiber$1.lanes = mergeLanes(currentlyRenderingFiber$1.lanes,
  updateLane);
            markSkippedUpdateLanes(updateLane);
          } else {
            // This update does have sufficient priority.
            if (newBaseQueueLast !== null) {
              var _clone = {
                // This update is going to be committed so we never want uncommit
                // it. Using NoLane works because 0 is a subset of all bitmasks, so
                // this will never be skipped by the check above.
                lane: NoLane,
                action: update.action,
                hasEagerState: update.hasEagerState,
                eagerState: update.eagerState,
                next: null
              };
              newBaseQueueLast = newBaseQueueLast.next = _clone;
            } // Process this update.


            if (update.hasEagerState) {
              // If this update is a state update (not a reducer) and was processed
  eagerly,
              // we can use the eagerly computed state
              newState = update.eagerState;
            } else {
              var action = update.action;
              newState = reducer(newState, action);
            }
          }

          update = update.next;
        } while (update !== null && update !== first);

        if (newBaseQueueLast === null) {
          newBaseState = newState;
        } else {
          newBaseQueueLast.next = newBaseQueueFirst;
        } // Mark that the fiber performed work, but only if the new state is
        // different from the current state.


        if (!objectIs(newState, hook.memoizedState)) {
          markWorkInProgressReceivedUpdate();
        }

        hook.memoizedState = newState;
        hook.baseState = newBaseState;
        hook.baseQueue = newBaseQueueLast;
        queue.lastRenderedState = newState;
      } // Interleaved updates are stored on a separate queue. We aren't going to
      // process them during this render, but we do need to track which lanes
      // are remaining.


      var lastInterleaved = queue.interleaved;

      if (lastInterleaved !== null) {
        var interleaved = lastInterleaved;

        do {
          var interleavedLane = interleaved.lane;
          currentlyRenderingFiber$1.lanes = mergeLanes(currentlyRenderingFiber$1.lanes,
  interleavedLane);
          markSkippedUpdateLanes(interleavedLane);
          interleaved = interleaved.next;
        } while (interleaved !== lastInterleaved);
      } else if (baseQueue === null) {
        // `queue.lanes` is used for entangling transitions. We can set it back to
        // zero once the queue is empty.
        queue.lanes = NoLanes;
      }

      var dispatch = queue.dispatch;
      return [hook.memoizedState, dispatch];
    }

    function rerenderReducer(reducer, initialArg, init) {
      var hook = updateWorkInProgressHook();
      var queue = hook.queue;

      if (queue === null) {
        throw new Error('Should have a queue. This is likely a bug in React. Please file an
  issue.');
      }

      queue.lastRenderedReducer = reducer; // This is a re-render. Apply the new render
  phase updates to the previous
      // work-in-progress hook.

      var dispatch = queue.dispatch;
      var lastRenderPhaseUpdate = queue.pending;
      var newState = hook.memoizedState;

      if (lastRenderPhaseUpdate !== null) {
        // The queue doesn't persist past this render pass.
        queue.pending = null;
        var firstRenderPhaseUpdate = lastRenderPhaseUpdate.next;
        var update = firstRenderPhaseUpdate;

        do {
          // Process this render phase update. We don't have to check the
          // priority because it will always be the same as the current
          // render's.
          var action = update.action;
          newState = reducer(newState, action);
          update = update.next;
        } while (update !== firstRenderPhaseUpdate); // Mark that the fiber performed work,
  but only if the new state is
        // different from the current state.


        if (!objectIs(newState, hook.memoizedState)) {
          markWorkInProgressReceivedUpdate();
        }

        hook.memoizedState = newState; // Don't persist the state accumulated from the
  render phase updates to
        // the base state unless the queue is empty.
        // TODO: Not sure if this is the desired semantics, but it's what we
        // do for gDSFP. I can't remember why.

        if (hook.baseQueue === null) {
          hook.baseState = newState;
        }

        queue.lastRenderedState = newState;
      }

      return [newState, dispatch];
    }

    function mountMutableSource(source, getSnapshot, subscribe) {
      {
        return undefined;
      }
    }

    function updateMutableSource(source, getSnapshot, subscribe) {
      {
        return undefined;
      }
    }

    function mountSyncExternalStore(subscribe, getSnapshot, getServerSnapshot) {
      var fiber = currentlyRenderingFiber$1;
      var hook = mountWorkInProgressHook();
      var nextSnapshot;
      var isHydrating = getIsHydrating();

      if (isHydrating) {
        if (getServerSnapshot === undefined) {
          throw new Error('Missing getServerSnapshot, which is required for ' + 'server-
  rendered content. Will revert to client rendering.');
        }

        nextSnapshot = getServerSnapshot();

        {
          if (!didWarnUncachedGetSnapshot) {
            if (nextSnapshot !== getServerSnapshot()) {
              error('The result of getServerSnapshot should be cached to avoid an infinite
  loop');

              didWarnUncachedGetSnapshot = true;
            }
          }
        }
      } else {
        nextSnapshot = getSnapshot();

        {
          if (!didWarnUncachedGetSnapshot) {
            var cachedSnapshot = getSnapshot();

            if (!objectIs(nextSnapshot, cachedSnapshot)) {
              error('The result of getSnapshot should be cached to avoid an infinite
  loop');

              didWarnUncachedGetSnapshot = true;
            }
          }
        } // Unless we're rendering a blocking lane, schedule a consistency check.
        // Right before committing, we will walk the tree and check if any of the
        // stores were mutated.
        //
        // We won't do this if we're hydrating server-rendered content, because if
        // the content is stale, it's already visible anyway. Instead we'll patch
        // it up in a passive effect.


        var root = getWorkInProgressRoot();

        if (root === null) {
          throw new Error('Expected a work-in-progress root. This is a bug in React. Please
  file an issue.');
        }

        if (!includesBlockingLane(root, renderLanes)) {
          pushStoreConsistencyCheck(fiber, getSnapshot, nextSnapshot);
        }
      } // Read the current snapshot from the store on every render. This breaks the
      // normal rules of React, and only works because store updates are
      // always synchronous.


      hook.memoizedState = nextSnapshot;
      var inst = {
        value: nextSnapshot,
        getSnapshot: getSnapshot
      };
      hook.queue = inst; // Schedule an effect to subscribe to the store.

      mountEffect(subscribeToStore.bind(null, fiber, inst, subscribe), [subscribe]); //
  Schedule an effect to update the mutable instance fields. We will update
      // this whenever subscribe, getSnapshot, or value changes. Because there's no
      // clean-up function, and we track the deps correctly, we can call pushEffect
      // directly, without storing any additional state. For the same reason, we
      // don't need to set a static flag, either.
      // TODO: We can move this to the passive phase once we add a pre-commit
      // consistency check. See the next comment.
      fiber.flags |= Passive;
      pushEffect(HasEffect | Passive$1, updateStoreInstance.bind(null, fiber, inst,
  nextSnapshot, getSnapshot), undefined, null);
      return nextSnapshot;
    }

    function updateSyncExternalStore(subscribe, getSnapshot, getServerSnapshot) {
      var fiber = currentlyRenderingFiber$1;
      var hook = updateWorkInProgressHook(); // Read the current snapshot from the store on
  every render. This breaks the
      // normal rules of React, and only works because store updates are
      // always synchronous.

      var nextSnapshot = getSnapshot();

      {
        if (!didWarnUncachedGetSnapshot) {
          var cachedSnapshot = getSnapshot();

          if (!objectIs(nextSnapshot, cachedSnapshot)) {
            error('The result of getSnapshot should be cached to avoid an infinite loop');

            didWarnUncachedGetSnapshot = true;
          }
        }
      }

      var prevSnapshot = hook.memoizedState;
      var snapshotChanged = !objectIs(prevSnapshot, nextSnapshot);

      if (snapshotChanged) {
        hook.memoizedState = nextSnapshot;
        markWorkInProgressReceivedUpdate();
      }

      var inst = hook.queue;
      updateEffect(subscribeToStore.bind(null, fiber, inst, subscribe), [subscribe]); //
  Whenever getSnapshot or subscribe changes, we need to check in the
      // commit phase if there was an interleaved mutation. In concurrent mode
      // this can happen all the time, but even in synchronous mode, an earlier
      // effect may have mutated the store.

      if (inst.getSnapshot !== getSnapshot || snapshotChanged || // Check if the susbcribe
  function changed. We can save some memory by
      // checking whether we scheduled a subscription effect above.
      workInProgressHook !== null && workInProgressHook.memoizedState.tag & HasEffect) {
        fiber.flags |= Passive;
        pushEffect(HasEffect | Passive$1, updateStoreInstance.bind(null, fiber, inst,
  nextSnapshot, getSnapshot), undefined, null); // Unless we're rendering a blocking lane,
  schedule a consistency check.
        // Right before committing, we will walk the tree and check if any of the
        // stores were mutated.

        var root = getWorkInProgressRoot();

        if (root === null) {
          throw new Error('Expected a work-in-progress root. This is a bug in React. Please
  file an issue.');
        }

        if (!includesBlockingLane(root, renderLanes)) {
          pushStoreConsistencyCheck(fiber, getSnapshot, nextSnapshot);
        }
      }

      return nextSnapshot;
    }
```

```
    function pushStoreConsistencyCheck(fiber, getSnapshot, renderedSnapshot) {
      fiber.flags |= StoreConsistency;
      var check = {
        getSnapshot: getSnapshot,
        value: renderedSnapshot
      };
      var componentUpdateQueue = currentlyRenderingFiber$1.updateQueue;

      if (componentUpdateQueue === null) {
        componentUpdateQueue = createFunctionComponentUpdateQueue();
        currentlyRenderingFiber$1.updateQueue = componentUpdateQueue;
        componentUpdateQueue.stores = [check];
      } else {
        var stores = componentUpdateQueue.stores;

        if (stores === null) {
          componentUpdateQueue.stores = [check];
        } else {
          stores.push(check);
        }
      }
    }

    function updateStoreInstance(fiber, inst, nextSnapshot, getSnapshot) {
      // These are updated in the passive phase
      inst.value = nextSnapshot;
      inst.getSnapshot = getSnapshot; // Something may have been mutated in between render
  and commit. This could
      // have been in an event that fired before the passive effects, or it could
      // have been in a layout effect. In that case, we would have used the old
      // snapsho and getSnapshot values to bail out. We need to check one more time.

      if (checkIfSnapshotChanged(inst)) {
        // Force a re-render.
        forceStoreRerender(fiber);
      }
    }

    function subscribeToStore(fiber, inst, subscribe) {
      var handleStoreChange = function () {
        // The store changed. Check if the snapshot changed since the last time we
        // read from the store.
        if (checkIfSnapshotChanged(inst)) {
          // Force a re-render.
          forceStoreRerender(fiber);
        }
      }; // Subscribe to the store and return a clean-up function.


      return subscribe(handleStoreChange);
    }

    function checkIfSnapshotChanged(inst) {
      var latestGetSnapshot = inst.getSnapshot;
      var prevValue = inst.value;

      try {
        var nextValue = latestGetSnapshot();
        return !objectIs(prevValue, nextValue);
      } catch (error) {
        return true;
      }
    }

    function forceStoreRerender(fiber) {
      var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

      if (root !== null) {
        scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
      }
    }

    function mountState(initialState) {
      var hook = mountWorkInProgressHook();

      if (typeof initialState === 'function') {
        // $FlowFixMe: Flow doesn't like mixed types
        initialState = initialState();
      }

      hook.memoizedState = hook.baseState = initialState;
      var queue = {
        pending: null,
        interleaved: null,
        lanes: NoLanes,
        dispatch: null,
        lastRenderedReducer: basicStateReducer,
        lastRenderedState: initialState
      };
      hook.queue = queue;
      var dispatch = queue.dispatch = dispatchSetState.bind(null,
  currentlyRenderingFiber$1, queue);
      return [hook.memoizedState, dispatch];
    }

    function updateState(initialState) {
      return updateReducer(basicStateReducer);
    }

    function rerenderState(initialState) {
      return rerenderReducer(basicStateReducer);
    }

    function pushEffect(tag, create, destroy, deps) {
      var effect = {
        tag: tag,
        create: create,
        destroy: destroy,
        deps: deps,
        // Circular
        next: null
      };
      var componentUpdateQueue = currentlyRenderingFiber$1.updateQueue;

      if (componentUpdateQueue === null) {
        componentUpdateQueue = createFunctionComponentUpdateQueue();
        currentlyRenderingFiber$1.updateQueue = componentUpdateQueue;
        componentUpdateQueue.lastEffect = effect.next = effect;
      } else {
        var lastEffect = componentUpdateQueue.lastEffect;

        if (lastEffect === null) {
          componentUpdateQueue.lastEffect = effect.next = effect;
        } else {
          var firstEffect = lastEffect.next;
          lastEffect.next = effect;
          effect.next = firstEffect;
          componentUpdateQueue.lastEffect = effect;
        }
      }

      return effect;
    }

    function mountRef(initialValue) {
      var hook = mountWorkInProgressHook();

      {
```

```
      var _ref2 = {
        current: initialValue
      };
      hook.memoizedState = _ref2;
      return _ref2;
    }
  }

  function updateRef(initialValue) {
    var hook = updateWorkInProgressHook();
    return hook.memoizedState;
  }

  function mountEffectImpl(fiberFlags, hookFlags, create, deps) {
    var hook = mountWorkInProgressHook();
    var nextDeps = deps === undefined ? null : deps;
    currentlyRenderingFiber$1.flags |= fiberFlags;
    hook.memoizedState = pushEffect(HasEffect | hookFlags, create, undefined, nextDeps);
  }

  function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
    var hook = updateWorkInProgressHook();
    var nextDeps = deps === undefined ? null : deps;
    var destroy = undefined;

    if (currentHook !== null) {
      var prevEffect = currentHook.memoizedState;
      destroy = prevEffect.destroy;

      if (nextDeps !== null) {
        var prevDeps = prevEffect.deps;

        if (areHookInputsEqual(nextDeps, prevDeps)) {
          hook.memoizedState = pushEffect(hookFlags, create, destroy, nextDeps);
          return;
        }
      }
    }

    currentlyRenderingFiber$1.flags |= fiberFlags;
    hook.memoizedState = pushEffect(HasEffect | hookFlags, create, destroy, nextDeps);
  }

  function mountEffect(create, deps) {
    if ( (currentlyRenderingFiber$1.mode & StrictEffectsMode) !== NoMode) {
      return mountEffectImpl(MountPassiveDev | Passive | PassiveStatic, Passive$1,
create, deps);
    } else {
      return mountEffectImpl(Passive | PassiveStatic, Passive$1, create, deps);
    }
  }

  function updateEffect(create, deps) {
    return updateEffectImpl(Passive, Passive$1, create, deps);
  }

  function mountInsertionEffect(create, deps) {
    return mountEffectImpl(Update, Insertion, create, deps);
  }

  function updateInsertionEffect(create, deps) {
    return updateEffectImpl(Update, Insertion, create, deps);
  }

  function mountLayoutEffect(create, deps) {
    var fiberFlags = Update;

    {
      fiberFlags |= LayoutStatic;
    }

    if ( (currentlyRenderingFiber$1.mode & StrictEffectsMode) !== NoMode) {
      fiberFlags |= MountLayoutDev;
    }

    return mountEffectImpl(fiberFlags, Layout, create, deps);
  }

  function updateLayoutEffect(create, deps) {
    return updateEffectImpl(Update, Layout, create, deps);
  }

  function imperativeHandleEffect(create, ref) {
    if (typeof ref === 'function') {
      var refCallback = ref;

      var _inst = create();

      refCallback(_inst);
      return function () {
        refCallback(null);
      };
    } else if (ref !== null && ref !== undefined) {
      var refObject = ref;

      {
        if (!refObject.hasOwnProperty('current')) {
          error('Expected useImperativeHandle() first argument to either be a ' + 'ref
callback or React.createRef() object. Instead received: %s.', 'an object with keys {' +
Object.keys(refObject).join(', ') + '}');
        }
      }

      var _inst2 = create();

      refObject.current = _inst2;
      return function () {
        refObject.current = null;
      };
    }
  }

  function mountImperativeHandle(ref, create, deps) {
    {
      if (typeof create !== 'function') {
        error('Expected useImperativeHandle() second argument to be a function ' + 'that
creates a handle. Instead received: %s.', create !== null ? typeof create : 'null');
      }
    } // TODO: If deps are provided, should we skip comparing the ref itself?


    var effectDeps = deps !== null && deps !== undefined ? deps.concat([ref]) : null;
    var fiberFlags = Update;

    {
      fiberFlags |= LayoutStatic;
    }

    if ( (currentlyRenderingFiber$1.mode & StrictEffectsMode) !== NoMode) {
      fiberFlags |= MountLayoutDev;
    }

    return mountEffectImpl(fiberFlags, Layout, imperativeHandleEffect.bind(null, create,
ref), effectDeps);
  }

  function updateImperativeHandle(ref, create, deps) {
    {
      if (typeof create !== 'function') {
        error('Expected useImperativeHandle() second argument to be a function ' + 'that
creates a handle. Instead received: %s.', create !== null ? typeof create : 'null');
      }
    } // TODO: If deps are provided, should we skip comparing the ref itself?


    var effectDeps = deps !== null && deps !== undefined ? deps.concat([ref]) : null;
    return updateEffectImpl(Update, Layout, imperativeHandleEffect.bind(null, create,
ref), effectDeps);
  }

  function mountDebugValue(value, formatterFn) {// This hook is normally a no-op.
    // The react-debug-hooks package injects its own implementation
    // so that e.g. DevTools can display custom hook values.
  }

  var updateDebugValue = mountDebugValue;

  function mountCallback(callback, deps) {
    var hook = mountWorkInProgressHook();
    var nextDeps = deps === undefined ? null : deps;
    hook.memoizedState = [callback, nextDeps];
    return callback;
  }

  function updateCallback(callback, deps) {
    var hook = updateWorkInProgressHook();
    var nextDeps = deps === undefined ? null : deps;
    var prevState = hook.memoizedState;

    if (prevState !== null) {
      if (nextDeps !== null) {
        var prevDeps = prevState[1];

        if (areHookInputsEqual(nextDeps, prevDeps)) {
          return prevState[0];
        }
      }
    }

    hook.memoizedState = [callback, nextDeps];
    return callback;
  }

  function mountMemo(nextCreate, deps) {
    var hook = mountWorkInProgressHook();
    var nextDeps = deps === undefined ? null : deps;
    var nextValue = nextCreate();
    hook.memoizedState = [nextValue, nextDeps];
    return nextValue;
  }

  function updateMemo(nextCreate, deps) {
    var hook = updateWorkInProgressHook();
    var nextDeps = deps === undefined ? null : deps;
    var prevState = hook.memoizedState;

    if (prevState !== null) {
      // Assume these are defined. If they're not, areHookInputsEqual will warn.
      if (nextDeps !== null) {
        var prevDeps = prevState[1];

        if (areHookInputsEqual(nextDeps, prevDeps)) {
          return prevState[0];
        }
      }
    }
```

```
      var nextValue = nextCreate();
      hook.memoizedState = [nextValue, nextDeps];
      return nextValue;
    }

    function mountDeferredValue(value) {
      var hook = mountWorkInProgressHook();
      hook.memoizedState = value;
      return value;
    }

    function updateDeferredValue(value) {
      var hook = updateWorkInProgressHook();
      var resolvedCurrentHook = currentHook;
      var prevValue = resolvedCurrentHook.memoizedState;
      return updateDeferredValueImpl(hook, prevValue, value);
    }

    function rerenderDeferredValue(value) {
      var hook = updateWorkInProgressHook();

      if (currentHook === null) {
        // This is a rerender during a mount.
        hook.memoizedState = value;
        return value;
      } else {
        // This is a rerender during an update.
        var prevValue = currentHook.memoizedState;
        return updateDeferredValueImpl(hook, prevValue, value);
      }
    }

    function updateDeferredValueImpl(hook, prevValue, value) {
      var shouldDeferValue = !includesOnlyNonUrgentLanes(renderLanes);

      if (shouldDeferValue) {
        // This is an urgent update. If the value has changed, keep using the
        // previous value and spawn a deferred render to update it later.
        if (!objectIs(value, prevValue)) {
          // Schedule a deferred render
          var deferredLane = claimNextTransitionLane();
          currentlyRenderingFiber$1.lanes = mergeLanes(currentlyRenderingFiber$1.lanes,
  deferredLane);
          markSkippedUpdateLanes(deferredLane); // Set this to true to indicate that the
  rendered value is inconsistent
          // from the latest value. The name "baseState" doesn't really match how we
          // use it because we're reusing a state hook field instead of creating a
          // new one.

          hook.baseState = true;
        } // Reuse the previous value


        return prevValue;
      } else {
        // This is not an urgent update, so we can use the latest value regardless
        // of what it is. No need to defer it.
        // However, if we're currently inside a spawned render, then we need to mark
        // this as an update to prevent the fiber from bailing out.
        //
        // `baseState` is true when the current value is different from the rendered
        // value. The name doesn't really match how we use it because we're reusing
        // a state hook field instead of creating a new one.
        if (hook.baseState) {
          // Flip this back to false.
          hook.baseState = false;
          markWorkInProgressReceivedUpdate();
        }
        hook.memoizedState = value;
        return value;
      }
    }

    function startTransition(setPending, callback, options) {
      var previousPriority = getCurrentUpdatePriority();
      setCurrentUpdatePriority(higherEventPriority(previousPriority,
ContinuousEventPriority));
      setPending(true);
      var prevTransition = ReactCurrentBatchConfig$2.transition;
      ReactCurrentBatchConfig$2.transition = {};
      var currentTransition = ReactCurrentBatchConfig$2.transition;

      {
        ReactCurrentBatchConfig$2.transition._updatedFibers = new Set();
      }

      try {
        setPending(false);
        callback();
      } finally {
        setCurrentUpdatePriority(previousPriority);
        ReactCurrentBatchConfig$2.transition = prevTransition;

        {
          if (prevTransition === null && currentTransition._updatedFibers) {
            var updatedFibersCount = currentTransition._updatedFibers.size;

            if (updatedFibersCount > 10) {
              warn('Detected a large number of updates inside startTransition. ' + 'If this
is due to a subscription please re-write it to use React provided hooks. ' + 'Otherwise
concurrent mode guarantees are off the table.');
            }

            currentTransition._updatedFibers.clear();
          }
        }
      }
    }

    function mountTransition() {
      var _mountState = mountState(false),
          isPending = _mountState[0],
          setPending = _mountState[1]; // The `start` method never changes.


      var start = startTransition.bind(null, setPending);
      var hook = mountWorkInProgressHook();
      hook.memoizedState = start;
      return [isPending, start];
    }

    function updateTransition() {
      var _updateState = updateState(),
          isPending = _updateState[0];

      var hook = updateWorkInProgressHook();
      var start = hook.memoizedState;
      return [isPending, start];
    }

    function rerenderTransition() {
      var _rerenderState = rerenderState(),
          isPending = _rerenderState[0];

      var hook = updateWorkInProgressHook();
      var start = hook.memoizedState;
      return [isPending, start];
```

```
  }

  var isUpdatingOpaqueValueInRenderPhase = false;
  function getIsUpdatingOpaqueValueInRenderPhaseInDEV() {
    {
      return isUpdatingOpaqueValueInRenderPhase;
    }
  }

  function mountId() {
    var hook = mountWorkInProgressHook();
    var root = getWorkInProgressRoot(); // TODO: In Fizz, id generation is specific to
each server config. Maybe we
    // should do this in Fiber, too? Deferring this decision for now because
    // there's no other place to store the prefix except for an internal field on
    // the public createRoot object, which the fiber tree does not currently have
    // a reference to.

    var identifierPrefix = root.identifierPrefix;
    var id;

    if (getIsHydrating()) {
      var treeId = getTreeId(); // Use a captial R prefix for server-generated ids.

      id = ':' + identifierPrefix + 'R' + treeId; // Unless this is the first id at this
level, append a number at the end
      // that represents the position of this useId hook among all the useId
      // hooks for this fiber.

      var localId = localIdCounter++;

      if (localId > 0) {
        id += 'H' + localId.toString(32);
      }

      id += ':';
    } else {
      // Use a lowercase r prefix for client-generated ids.
      var globalClientId = globalClientIdCounter++;
      id = ':' + identifierPrefix + 'r' + globalClientId.toString(32) + ':';
    }

    hook.memoizedState = id;
    return id;
  }

  function updateId() {
    var hook = updateWorkInProgressHook();
    var id = hook.memoizedState;
    return id;
  }

  function dispatchReducerAction(fiber, queue, action) {
    {
      if (typeof arguments[3] === 'function') {
        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().');
      }
    }

    var lane = requestUpdateLane(fiber);
    var update = {
      lane: lane,
      action: action,
      hasEagerState: false,
      eagerState: null,
      next: null
    };
    if (isRenderPhaseUpdate(fiber)) {
      enqueueRenderPhaseUpdate(queue, update);
    } else {
      var root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);

      if (root !== null) {
        var eventTime = requestEventTime();
        scheduleUpdateOnFiber(root, fiber, lane, eventTime);
        entangleTransitionUpdate(root, queue, lane);
      }
    }

    markUpdateInDevTools(fiber, lane);
  }

  function dispatchSetState(fiber, queue, action) {
    {
      if (typeof arguments[3] === 'function') {
        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().');
      }
    }

    var lane = requestUpdateLane(fiber);
    var update = {
      lane: lane,
      action: action,
      hasEagerState: false,
      eagerState: null,
      next: null
    };

    if (isRenderPhaseUpdate(fiber)) {
      enqueueRenderPhaseUpdate(queue, update);
    } else {
      var alternate = fiber.alternate;

      if (fiber.lanes === NoLanes && (alternate === null || alternate.lanes === NoLanes))
{
        // The queue is currently empty, which means we can eagerly compute the
        // next state before entering the render phase. If the new state is the
        // same as the current state, we may be able to bail out entirely.
        var lastRenderedReducer = queue.lastRenderedReducer;

        if (lastRenderedReducer !== null) {
          var prevDispatcher;

          {
            prevDispatcher = ReactCurrentDispatcher$1.current;
            ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;
          }

          try {
            var currentState = queue.lastRenderedState;
            var eagerState = lastRenderedReducer(currentState, action); // Stash the
eagerly computed state, and the reducer used to compute
            // it, on the update object. If the reducer hasn't changed by the
            // time we enter the render phase, then the eager state can be used
            // without calling the reducer again.

            update.hasEagerState = true;
            update.eagerState = eagerState;

            if (objectIs(eagerState, currentState)) {
              // Fast path. We can bail out without scheduling React to re-render.
              // It's still possible that we'll need to rebase this update later,
              // if the component re-renders for a different reason and by that
              // time the reducer has changed.
              // TODO: Do we still need to entangle transitions in this case?
              enqueueConcurrentHookUpdateAndEagerlyBailout(fiber, queue, update, lane);
              return;
            }
          } catch (error) {// Suppress the error. It will throw again in the render
phase.
          } finally {
            {
              ReactCurrentDispatcher$1.current = prevDispatcher;
            }
          }
        }
      }

      var root = enqueueConcurrentHookUpdate(fiber, queue, update, lane);

      if (root !== null) {
        var eventTime = requestEventTime();
        scheduleUpdateOnFiber(root, fiber, lane, eventTime);
        entangleTransitionUpdate(root, queue, lane);
      }
    }

    markUpdateInDevTools(fiber, lane);
  }

  function isRenderPhaseUpdate(fiber) {
    var alternate = fiber.alternate;
    return fiber === currentlyRenderingFiber$1 || alternate !== null && alternate ===
currentlyRenderingFiber$1;
  }

  function enqueueRenderPhaseUpdate(queue, update) {
    // This is a render phase update. Stash it in a lazily-created map of
    // queue -> linked list of updates. After this render pass, we'll restart
    // and apply the stashed updates on top of the work-in-progress hook.
    didScheduleRenderPhaseUpdateDuringThisPass = didScheduleRenderPhaseUpdate = true;
    var pending = queue.pending;

    if (pending === null) {
      // This is the first update. Create a circular list.
      update.next = update;
    } else {
      update.next = pending.next;
      pending.next = update;
    }

    queue.pending = update;
  } // TODO: Move to ReactFiberConcurrentUpdates?


  function entangleTransitionUpdate(root, queue, lane) {
    if (isTransitionLane(lane)) {
      var queueLanes = queue.lanes; // If any entangled lanes are no longer pending on
the root, then they
      // must have finished. We can remove them from the shared queue, which
      // represents a superset of the actually pending lanes. In some cases we
      // may entangle more than we need to, but that's OK. In fact it's worse if
      // we *don't* entangle when we should.

      queueLanes = intersectLanes(queueLanes, root.pendingLanes); // Entangle the new
transition lane with the other transition lanes.

      var newQueueLanes = mergeLanes(queueLanes, lane);
      queue.lanes = newQueueLanes; // Even if queue.lanes already include lane, we don't
know for certain if
      // the lane finished since the last time we entangled it. So we need to
      // entangle it again, just to be sure.
      markRootEntangled(root, newQueueLanes);
    }
  }

  function markUpdateInDevTools(fiber, lane, action) {

    {
      markStateUpdateScheduled(fiber, lane);
    }
  }

  var ContextOnlyDispatcher = {
    readContext: readContext,
    useCallback: throwInvalidHookError,
    useContext: throwInvalidHookError,
    useEffect: throwInvalidHookError,
    useImperativeHandle: throwInvalidHookError,
    useInsertionEffect: throwInvalidHookError,
    useLayoutEffect: throwInvalidHookError,
    useMemo: throwInvalidHookError,
    useReducer: throwInvalidHookError,
    useRef: throwInvalidHookError,
    useState: throwInvalidHookError,
    useDebugValue: throwInvalidHookError,
    useDeferredValue: throwInvalidHookError,
    useTransition: throwInvalidHookError,
    useMutableSource: throwInvalidHookError,
    useSyncExternalStore: throwInvalidHookError,
    useId: throwInvalidHookError,
    unstable_isNewReconciler: enableNewReconciler
  };

  var HooksDispatcherOnMountInDEV = null;
  var HooksDispatcherOnMountWithHookTypesInDEV = null;
  var HooksDispatcherOnUpdateInDEV = null;
  var HooksDispatcherOnRerenderInDEV = null;
  var InvalidNestedHooksDispatcherOnMountInDEV = null;
  var InvalidNestedHooksDispatcherOnUpdateInDEV = null;
  var InvalidNestedHooksDispatcherOnRerenderInDEV = null;

  {
    var warnInvalidContextAccess = function () {
      error('Context can only be read while React is rendering. ' + 'In classes, you can
read it in the render method or getDerivedStateFromProps. ' + 'In function components,
you can read it directly in the function body, but not ' + 'inside Hooks like
useReducer() or useMemo().');
    };

    var warnInvalidHookAccess = function () {
      error('Do not call Hooks inside useEffect(...), useMemo(...), or other built-in
Hooks. ' + 'You can only call Hooks at the top level of your React function. ' + 'For
more information, see ' + 'https://reactjs.org/link/rules-of-hooks');
    };

    HooksDispatcherOnMountInDEV = {
      readContext: function (context) {
        return readContext(context);
      },
      useCallback: function (callback, deps) {
        currentHookNameInDev = 'useCallback';
        mountHookTypesDev();
        checkDepsAreArrayDev(deps);
        return mountCallback(callback, deps);
      },
      useContext: function (context) {
        currentHookNameInDev = 'useContext';
        mountHookTypesDev();
        return readContext(context);
      },
      useEffect: function (create, deps) {
        currentHookNameInDev = 'useEffect';
        mountHookTypesDev();
        checkDepsAreArrayDev(deps);
        return mountEffect(create, deps);
      },
      useImperativeHandle: function (ref, create, deps) {
        currentHookNameInDev = 'useImperativeHandle';
        mountHookTypesDev();
        checkDepsAreArrayDev(deps);
        return mountImperativeHandle(ref, create, deps);
      },
      useInsertionEffect: function (create, deps) {
        currentHookNameInDev = 'useInsertionEffect';
        mountHookTypesDev();
        checkDepsAreArrayDev(deps);
        return mountInsertionEffect(create, deps);
      },
      useLayoutEffect: function (create, deps) {
        currentHookNameInDev = 'useLayoutEffect';
        mountHookTypesDev();
        checkDepsAreArrayDev(deps);
        return mountLayoutEffect(create, deps);
      },
      useMemo: function (create, deps) {
        currentHookNameInDev = 'useMemo';
        mountHookTypesDev();
        checkDepsAreArrayDev(deps);
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountMemo(create, deps);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useReducer: function (reducer, initialArg, init) {
        currentHookNameInDev = 'useReducer';
        mountHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountReducer(reducer, initialArg, init);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useRef: function (initialValue) {
        currentHookNameInDev = 'useRef';
        mountHookTypesDev();
        return mountRef(initialValue);
      },
      useState: function (initialState) {
        currentHookNameInDev = 'useState';
        mountHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountState(initialState);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useDebugValue: function (value, formatterFn) {
        currentHookNameInDev = 'useDebugValue';
        mountHookTypesDev();
        return mountDebugValue();
      },
      useDeferredValue: function (value) {
        currentHookNameInDev = 'useDeferredValue';
        mountHookTypesDev();
        return mountDeferredValue(value);
      },
      useTransition: function () {
        currentHookNameInDev = 'useTransition';
        mountHookTypesDev();
        return mountTransition();
      },
      useMutableSource: function (source, getSnapshot, subscribe) {
        currentHookNameInDev = 'useMutableSource';
        mountHookTypesDev();
        return mountMutableSource();
      },
      useSyncExternalStore: function (subscribe, getSnapshot, getServerSnapshot) {
        currentHookNameInDev = 'useSyncExternalStore';
        mountHookTypesDev();
        return mountSyncExternalStore(subscribe, getSnapshot, getServerSnapshot);
      },
      useId: function () {
        currentHookNameInDev = 'useId';
        mountHookTypesDev();
        return mountId();
      },
      unstable_isNewReconciler: enableNewReconciler
    };

    HooksDispatcherOnMountWithHookTypesInDEV = {
      readContext: function (context) {
        return readContext(context);
      },
      useCallback: function (callback, deps) {
        currentHookNameInDev = 'useCallback';
        updateHookTypesDev();
        return mountCallback(callback, deps);
      },
      useContext: function (context) {
        currentHookNameInDev = 'useContext';
        updateHookTypesDev();
        return readContext(context);
      },
      useEffect: function (create, deps) {
        currentHookNameInDev = 'useEffect';
        updateHookTypesDev();
        return mountEffect(create, deps);
      },
      useImperativeHandle: function (ref, create, deps) {
        currentHookNameInDev = 'useImperativeHandle';
        updateHookTypesDev();
        return mountImperativeHandle(ref, create, deps);
      },
      useInsertionEffect: function (create, deps) {
        currentHookNameInDev = 'useInsertionEffect';
        updateHookTypesDev();
        return mountInsertionEffect(create, deps);
      },
      useLayoutEffect: function (create, deps) {
        currentHookNameInDev = 'useLayoutEffect';
        updateHookTypesDev();
        return mountLayoutEffect(create, deps);
      },
      useMemo: function (create, deps) {
        currentHookNameInDev = 'useMemo';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountMemo(create, deps);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useReducer: function (reducer, initialArg, init) {
        currentHookNameInDev = 'useReducer';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountReducer(reducer, initialArg, init);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useRef: function (initialValue) {
        currentHookNameInDev = 'useRef';
        updateHookTypesDev();
        return mountRef(initialValue);
      },
      useState: function (initialState) {
        currentHookNameInDev = 'useState';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountState(initialState);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useDebugValue: function (value, formatterFn) {
        currentHookNameInDev = 'useDebugValue';
        updateHookTypesDev();
        return mountDebugValue();
      },
      useDeferredValue: function (value) {
        currentHookNameInDev = 'useDeferredValue';
        updateHookTypesDev();
        return mountDeferredValue(value);
      },
      useTransition: function () {
        currentHookNameInDev = 'useTransition';
        updateHookTypesDev();
        return mountTransition();
      },
      useMutableSource: function (source, getSnapshot, subscribe) {
        currentHookNameInDev = 'useMutableSource';
        updateHookTypesDev();
        return mountMutableSource();
      },
      useSyncExternalStore: function (subscribe, getSnapshot, getServerSnapshot) {
        currentHookNameInDev = 'useSyncExternalStore';
        updateHookTypesDev();
        return mountSyncExternalStore(subscribe, getSnapshot, getServerSnapshot);
      },
      useId: function () {
        currentHookNameInDev = 'useId';
        updateHookTypesDev();
        return mountId();
      },
      unstable_isNewReconciler: enableNewReconciler
    };
    HooksDispatcherOnUpdateInDEV = {
      readContext: function (context) {
        return readContext(context);
      },
      useCallback: function (callback, deps) {
        currentHookNameInDev = 'useCallback';
        updateHookTypesDev();
        return updateCallback(callback, deps);
      },
      useContext: function (context) {
        currentHookNameInDev = 'useContext';
        updateHookTypesDev();
        return readContext(context);
      },
      useEffect: function (create, deps) {
        currentHookNameInDev = 'useEffect';
        updateHookTypesDev();
        return updateEffect(create, deps);
      },
      useImperativeHandle: function (ref, create, deps) {
        currentHookNameInDev = 'useImperativeHandle';
        updateHookTypesDev();
        return updateImperativeHandle(ref, create, deps);
      },
      useInsertionEffect: function (create, deps) {
        currentHookNameInDev = 'useInsertionEffect';
        updateHookTypesDev();
        return updateInsertionEffect(create, deps);
      },
      useLayoutEffect: function (create, deps) {
        currentHookNameInDev = 'useLayoutEffect';
        updateHookTypesDev();
        return updateLayoutEffect(create, deps);
      },
      useMemo: function (create, deps) {
        currentHookNameInDev = 'useMemo';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return updateMemo(create, deps);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useReducer: function (reducer, initialArg, init) {
        currentHookNameInDev = 'useReducer';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return updateReducer(reducer, initialArg, init);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useRef: function (initialValue) {
        currentHookNameInDev = 'useRef';
        updateHookTypesDev();
        return updateRef();
      },
      useState: function (initialState) {
        currentHookNameInDev = 'useState';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;
        try {
          return updateState(initialState);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useDebugValue: function (value, formatterFn) {
        currentHookNameInDev = 'useDebugValue';
        updateHookTypesDev();
        return updateDebugValue();
      },
      useDeferredValue: function (value) {
        currentHookNameInDev = 'useDeferredValue';
        updateHookTypesDev();
        return updateDeferredValue(value);
      },
      useTransition: function () {
        currentHookNameInDev = 'useTransition';
        updateHookTypesDev();
        return updateTransition();
      },
      useMutableSource: function (source, getSnapshot, subscribe) {
        currentHookNameInDev = 'useMutableSource';
        updateHookTypesDev();
        return updateMutableSource();
      },
      useSyncExternalStore: function (subscribe, getSnapshot, getServerSnapshot) {
        currentHookNameInDev = 'useSyncExternalStore';
        updateHookTypesDev();
        return updateSyncExternalStore(subscribe, getSnapshot);
      },
      useId: function () {
        currentHookNameInDev = 'useId';
        updateHookTypesDev();
        return updateId();
      },
      unstable_isNewReconciler: enableNewReconciler
    };

    HooksDispatcherOnRerenderInDEV = {
      readContext: function (context) {
        return readContext(context);
      },
      useCallback: function (callback, deps) {
        currentHookNameInDev = 'useCallback';
        updateHookTypesDev();
        return updateCallback(callback, deps);
      },
      useContext: function (context) {
        currentHookNameInDev = 'useContext';
        updateHookTypesDev();
        return readContext(context);
      },
      useEffect: function (create, deps) {
        currentHookNameInDev = 'useEffect';
        updateHookTypesDev();
        return updateEffect(create, deps);
      },
      useImperativeHandle: function (ref, create, deps) {
        currentHookNameInDev = 'useImperativeHandle';
        updateHookTypesDev();
        return updateImperativeHandle(ref, create, deps);
      },
      useInsertionEffect: function (create, deps) {
        currentHookNameInDev = 'useInsertionEffect';
        updateHookTypesDev();
        return updateInsertionEffect(create, deps);
      },
      useLayoutEffect: function (create, deps) {
        currentHookNameInDev = 'useLayoutEffect';
        updateHookTypesDev();
        return updateLayoutEffect(create, deps);
      },
      useMemo: function (create, deps) {
        currentHookNameInDev = 'useMemo';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnRerenderInDEV;

        try {
          return updateMemo(create, deps);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useReducer: function (reducer, initialArg, init) {
        currentHookNameInDev = 'useReducer';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnRerenderInDEV;

        try {
          return rerenderReducer(reducer, initialArg, init);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useRef: function (initialValue) {
        currentHookNameInDev = 'useRef';
        updateHookTypesDev();
        return updateRef();
      },
      useState: function (initialState) {
        currentHookNameInDev = 'useState';
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnRerenderInDEV;

        try {
          return rerenderState(initialState);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useDebugValue: function (value, formatterFn) {
        currentHookNameInDev = 'useDebugValue';
        updateHookTypesDev();
        return updateDebugValue();
      },
      useDeferredValue: function (value) {
        currentHookNameInDev = 'useDeferredValue';
        updateHookTypesDev();
        return rerenderDeferredValue(value);
      },
      useTransition: function () {
        currentHookNameInDev = 'useTransition';
        updateHookTypesDev();
        return rerenderTransition();
      },
      useMutableSource: function (source, getSnapshot, subscribe) {
        currentHookNameInDev = 'useMutableSource';
        updateHookTypesDev();
        return updateMutableSource();
      },
      useSyncExternalStore: function (subscribe, getSnapshot, getServerSnapshot) {
        currentHookNameInDev = 'useSyncExternalStore';
        updateHookTypesDev();
        return updateSyncExternalStore(subscribe, getSnapshot);
      },
      useId: function () {
        currentHookNameInDev = 'useId';
        updateHookTypesDev();
        return updateId();
      },
      unstable_isNewReconciler: enableNewReconciler
    };

    InvalidNestedHooksDispatcherOnMountInDEV = {
      readContext: function (context) {
        warnInvalidContextAccess();
        return readContext(context);
      },
      useCallback: function (callback, deps) {
        currentHookNameInDev = 'useCallback';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountCallback(callback, deps);
      },
      useContext: function (context) {
        currentHookNameInDev = 'useContext';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return readContext(context);
      },
      useEffect: function (create, deps) {
        currentHookNameInDev = 'useEffect';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountEffect(create, deps);
      },
      useImperativeHandle: function (ref, create, deps) {
        currentHookNameInDev = 'useImperativeHandle';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountImperativeHandle(ref, create, deps);
      },
      useInsertionEffect: function (create, deps) {
        currentHookNameInDev = 'useInsertionEffect';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountInsertionEffect(create, deps);
      },
      useLayoutEffect: function (create, deps) {
        currentHookNameInDev = 'useLayoutEffect';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountLayoutEffect(create, deps);
      },
      useMemo: function (create, deps) {
        currentHookNameInDev = 'useMemo';
        warnInvalidHookAccess();
        mountHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountMemo(create, deps);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useReducer: function (reducer, initialArg, init) {
        currentHookNameInDev = 'useReducer';
        warnInvalidHookAccess();
        mountHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountReducer(reducer, initialArg, init);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useRef: function (initialValue) {
        currentHookNameInDev = 'useRef';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountRef(initialValue);
      },
      useState: function (initialState) {
        currentHookNameInDev = 'useState';
        warnInvalidHookAccess();
        mountHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnMountInDEV;

        try {
          return mountState(initialState);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useDebugValue: function (value, formatterFn) {
        currentHookNameInDev = 'useDebugValue';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountDebugValue();
      },
      useDeferredValue: function (value) {
        currentHookNameInDev = 'useDeferredValue';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountDeferredValue(value);
      },
      useTransition: function () {
        currentHookNameInDev = 'useTransition';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountTransition();
      },
      useMutableSource: function (source, getSnapshot, subscribe) {
        currentHookNameInDev = 'useMutableSource';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountMutableSource();
      },
      useSyncExternalStore: function (subscribe, getSnapshot, getServerSnapshot) {
        currentHookNameInDev = 'useSyncExternalStore';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountSyncExternalStore(subscribe, getSnapshot, getServerSnapshot);
      },
      useId: function () {
        currentHookNameInDev = 'useId';
        warnInvalidHookAccess();
        mountHookTypesDev();
        return mountId();
      },
      unstable_isNewReconciler: enableNewReconciler
    };

    InvalidNestedHooksDispatcherOnUpdateInDEV = {
      readContext: function (context) {
        warnInvalidContextAccess();
```

```
        return readContext(context);
      },
      useCallback: function (callback, deps) {
        currentHookNameInDev = 'useCallback';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateCallback(callback, deps);
      },
      useContext: function (context) {
        currentHookNameInDev = 'useContext';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return readContext(context);
      },
      useEffect: function (create, deps) {
        currentHookNameInDev = 'useEffect';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateEffect(create, deps);
      },
      useImperativeHandle: function (ref, create, deps) {
        currentHookNameInDev = 'useImperativeHandle';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateImperativeHandle(ref, create, deps);
      },
      useInsertionEffect: function (create, deps) {
        currentHookNameInDev = 'useInsertionEffect';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateInsertionEffect(create, deps);
      },
      useLayoutEffect: function (create, deps) {
        currentHookNameInDev = 'useLayoutEffect';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateLayoutEffect(create, deps);
      },
      useMemo: function (create, deps) {
        currentHookNameInDev = 'useMemo';
        warnInvalidHookAccess();
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return updateMemo(create, deps);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useReducer: function (reducer, initialArg, init) {
        currentHookNameInDev = 'useReducer';
        warnInvalidHookAccess();
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return updateReducer(reducer, initialArg, init);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useRef: function (initialValue) {
        currentHookNameInDev = 'useRef';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateRef();
      },
      useState: function (initialState) {
        currentHookNameInDev = 'useState';
        warnInvalidHookAccess();
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return updateState(initialState);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useDebugValue: function (value, formatterFn) {
        currentHookNameInDev = 'useDebugValue';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateDebugValue();
      },
      useDeferredValue: function (value) {
        currentHookNameInDev = 'useDeferredValue';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateDeferredValue(value);
      },
      useTransition: function () {
        currentHookNameInDev = 'useTransition';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateTransition();
      },
      useMutableSource: function (source, getSnapshot, subscribe) {
        currentHookNameInDev = 'useMutableSource';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateMutableSource();
      },
      useSyncExternalStore: function (subscribe, getSnapshot, getServerSnapshot) {
        currentHookNameInDev = 'useSyncExternalStore';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateSyncExternalStore(subscribe, getSnapshot);
      },
      useId: function () {
        currentHookNameInDev = 'useId';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateId();
      },
      unstable_isNewReconciler: enableNewReconciler
    };

    InvalidNestedHooksDispatcherOnRerenderInDEV = {
      readContext: function (context) {
        warnInvalidContextAccess();
        return readContext(context);
      },
      useCallback: function (callback, deps) {
        currentHookNameInDev = 'useCallback';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateCallback(callback, deps);
      },
      useContext: function (context) {
        currentHookNameInDev = 'useContext';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return readContext(context);
      },
      useEffect: function (create, deps) {
        currentHookNameInDev = 'useEffect';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateEffect(create, deps);
      },
      useImperativeHandle: function (ref, create, deps) {
        currentHookNameInDev = 'useImperativeHandle';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateImperativeHandle(ref, create, deps);
      },
      useInsertionEffect: function (create, deps) {
        currentHookNameInDev = 'useInsertionEffect';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateInsertionEffect(create, deps);
      },
      useLayoutEffect: function (create, deps) {
        currentHookNameInDev = 'useLayoutEffect';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateLayoutEffect(create, deps);
      },
      useMemo: function (create, deps) {
        currentHookNameInDev = 'useMemo';
        warnInvalidHookAccess();
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return updateMemo(create, deps);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useReducer: function (reducer, initialArg, init) {
        currentHookNameInDev = 'useReducer';
        warnInvalidHookAccess();
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return rerenderReducer(reducer, initialArg, init);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useRef: function (initialValue) {
        currentHookNameInDev = 'useRef';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateRef();
      },
      useState: function (initialState) {
        currentHookNameInDev = 'useState';
        warnInvalidHookAccess();
        updateHookTypesDev();
        var prevDispatcher = ReactCurrentDispatcher$1.current;
        ReactCurrentDispatcher$1.current = InvalidNestedHooksDispatcherOnUpdateInDEV;

        try {
          return rerenderState(initialState);
        } finally {
          ReactCurrentDispatcher$1.current = prevDispatcher;
        }
      },
      useDebugValue: function (value, formatterFn) {
        currentHookNameInDev = 'useDebugValue';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateDebugValue();
      },
      useDeferredValue: function (value) {
        currentHookNameInDev = 'useDeferredValue';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return rerenderDeferredValue(value);
      },
      useTransition: function () {
        currentHookNameInDev = 'useTransition';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return rerenderTransition();
      },
      useMutableSource: function (source, getSnapshot, subscribe) {
        currentHookNameInDev = 'useMutableSource';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateMutableSource();
      },
      useSyncExternalStore: function (subscribe, getSnapshot, getServerSnapshot) {
        currentHookNameInDev = 'useSyncExternalStore';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateSyncExternalStore(subscribe, getSnapshot);
      },
      useId: function () {
        currentHookNameInDev = 'useId';
        warnInvalidHookAccess();
        updateHookTypesDev();
        return updateId();
      },
      unstable_isNewReconciler: enableNewReconciler
    };
  }

  var now$1 = unstable_now;
  var commitTime = 0;
  var layoutEffectStartTime = -1;
  var profilerStartTime = -1;
  var passiveEffectStartTime = -1;
  /**
   * Tracks whether the current update was a nested/cascading update (scheduled from a
 layout effect).
   *
   * The overall sequence is:
   *   1. render
   *   2. commit (and call `onRender`, `onCommit`)
   *   3. check for nested updates
   *   4. flush passive effects (and call `onPostCommit`)
   *
   * Nested updates are identified in step 3 above,
   * but step 4 still applies to the work that was just committed.
   * We use two flags to track nested updates then:
   * one tracks whether the upcoming update is a nested update,
   * and the other tracks whether the current update was a nested update.
   * The first value gets synced to the second at the start of the render phase.
   */

  var currentUpdateIsNested = false;
  var nestedUpdateScheduled = false;

  function isCurrentUpdateNested() {
    return currentUpdateIsNested;
  }

  function markNestedUpdateScheduled() {
    {
      nestedUpdateScheduled = true;
    }
  }

  function resetNestedUpdateFlag() {
    {
      currentUpdateIsNested = false;
      nestedUpdateScheduled = false;
    }
  }

  function syncNestedUpdateFlag() {
    {
      currentUpdateIsNested = nestedUpdateScheduled;
      nestedUpdateScheduled = false;
    }
  }

  function getCommitTime() {
    return commitTime;
  }

  function recordCommitTime() {

    commitTime = now$1();
  }

  function startProfilerTimer(fiber) {

    profilerStartTime = now$1();

    if (fiber.actualStartTime < 0) {
      fiber.actualStartTime = now$1();
    }
  }

  function stopProfilerTimerIfRunning(fiber) {

    profilerStartTime = -1;
  }

  function stopProfilerTimerIfRunningAndRecordDelta(fiber, overrideBaseTime) {

    if (profilerStartTime >= 0) {
      var elapsedTime = now$1() - profilerStartTime;
      fiber.actualDuration += elapsedTime;

      if (overrideBaseTime) {
        fiber.selfBaseDuration = elapsedTime;
      }

      profilerStartTime = -1;
    }
  }

  function recordLayoutEffectDuration(fiber) {

    if (layoutEffectStartTime >= 0) {
      var elapsedTime = now$1() - layoutEffectStartTime;
      layoutEffectStartTime = -1; // Store duration on the next nearest Profiler ancestor
      // Or the root (for the DevTools Profiler to read)

      var parentFiber = fiber.return;

      while (parentFiber !== null) {
        switch (parentFiber.tag) {
          case HostRoot:
            var root = parentFiber.stateNode;
            root.effectDuration += elapsedTime;
            return;

          case Profiler:
            var parentStateNode = parentFiber.stateNode;
            parentStateNode.effectDuration += elapsedTime;
            return;
        }

        parentFiber = parentFiber.return;
      }
    }
  }

  function recordPassiveEffectDuration(fiber) {

    if (passiveEffectStartTime >= 0) {
      var elapsedTime = now$1() - passiveEffectStartTime;
      passiveEffectStartTime = -1; // Store duration on the next nearest Profiler
ancestor
      // Or the root (for the DevTools Profiler to read)

      var parentFiber = fiber.return;

      while (parentFiber !== null) {
        switch (parentFiber.tag) {
          case HostRoot:
            var root = parentFiber.stateNode;

            if (root !== null) {
              root.passiveEffectDuration += elapsedTime;
            }

            return;

          case Profiler:
            var parentStateNode = parentFiber.stateNode;

            if (parentStateNode !== null) {
              // Detached fibers have their state node cleared out.
              // In this case, the return pointer is also cleared out,
              // so we won't be able to report the time spent in this Profiler's subtree.
              parentStateNode.passiveEffectDuration += elapsedTime;
            }

            return;
        }

        parentFiber = parentFiber.return;
      }
    }
  }

  function startLayoutEffectTimer() {

    layoutEffectStartTime = now$1();
  }

  function startPassiveEffectTimer() {

    passiveEffectStartTime = now$1();
  }

  function transferActualDuration(fiber) {
    // Transfer time spent rendering these children so we don't lose it
    // after we rerender. This is used as a helper in special cases
    // where we should count the work of multiple passes.
    var child = fiber.child;

    while (child) {
      fiber.actualDuration += child.actualDuration;
      child = child.sibling;
    }
  }

  function resolveDefaultProps(Component, baseProps) {
    if (Component && Component.defaultProps) {
      // Resolve default props. Taken from ReactElement
      var props = assign({}, baseProps);
      var defaultProps = Component.defaultProps;

      for (var propName in defaultProps) {
        if (props[propName] === undefined) {
          props[propName] = defaultProps[propName];
        }
      }

      return props;
    }

    return baseProps;
  }

  var fakeInternalInstance = {};
  var didWarnAboutStateAssignmentForComponent;
  var didWarnAboutUninitializedState;
  var didWarnAboutGetSnapshotBeforeUpdateWithoutDidUpdate;
  var didWarnAboutLegacyLifecyclesAndDerivedState;
  var didWarnAboutUndefinedDerivedState;
  var warnOnUndefinedDerivedState;
  var warnOnInvalidCallback;
  var didWarnAboutDirectlyAssigningPropsToState;
  var didWarnAboutContextTypeAndContextTypes;
  var didWarnAboutInvalidateContextType;
  var didWarnAboutLegacyContext$1;

  {
    didWarnAboutStateAssignmentForComponent = new Set();
    didWarnAboutUninitializedState = new Set();
    didWarnAboutGetSnapshotBeforeUpdateWithoutDidUpdate = new Set();
    didWarnAboutLegacyLifecyclesAndDerivedState = new Set();
    didWarnAboutDirectlyAssigningPropsToState = new Set();
    didWarnAboutUndefinedDerivedState = new Set();
    didWarnAboutContextTypeAndContextTypes = new Set();
    didWarnAboutInvalidateContextType = new Set();
    didWarnAboutLegacyContext$1 = new Set();
    var didWarnOnInvalidCallback = new Set();

    warnOnInvalidCallback = function (callback, callerName) {
      if (callback === null || typeof callback === 'function') {
        return;
      }

      var key = callerName + '_' + callback;

      if (!didWarnOnInvalidCallback.has(key)) {
        didWarnOnInvalidCallback.add(key);

        error('%s(...): Expected the last optional `callback` argument to be a ' +
'function. Instead received: %s.', callerName, callback);
      }
    };

    warnOnUndefinedDerivedState = function (type, partialState) {
      if (partialState === undefined) {
```

```
        var componentName = getComponentNameFromType(type) || 'Component';

        if (!didWarnAboutUndefinedDerivedState.has(componentName)) {
          didWarnAboutUndefinedDerivedState.add(componentName);

          error('%s.getDerivedStateFromProps(): A valid state object (or null) must be
returned. ' + 'You have returned undefined.', componentName);
        }
      }
    }; // This is so gross but it's at least non-critical and can be removed if
    // it causes problems. This is meant to give a nicer error message for
    // ReactDOM15.unstable_renderSubtreeIntoContainer(reactDOM16Component,
    // ...)) which otherwise throws a "_processChildContext is not a function"
    // exception.


    Object.defineProperty(fakeInternalInstance, '_processChildContext', {
      enumerable: false,
      value: function () {
        throw new Error('_processChildContext is not available in React 16+. This likely
' + 'means you have multiple copies of React and are attempting to nest ' + 'a React 15
tree inside a React 16 tree using ' + "unstable_renderSubtreeIntoContainer, which isn't
supported. Try " + 'to make sure you have only one copy of React (and ideally, switch ' +
'to ReactDOM.createPortal).');
      }
    });
    Object.freeze(fakeInternalInstance);
  }

  function applyDerivedStateFromProps(workInProgress, ctor, getDerivedStateFromProps,
nextProps) {
    var prevState = workInProgress.memoizedState;
    var partialState = getDerivedStateFromProps(nextProps, prevState);

    {
      if ( workInProgress.mode & StrictLegacyMode) {
        setIsStrictModeForDevtools(true);

        try {
          // Invoke the function an extra time to help detect side-effects.
          partialState = getDerivedStateFromProps(nextProps, prevState);
        } finally {
          setIsStrictModeForDevtools(false);
        }
      }

      warnOnUndefinedDerivedState(ctor, partialState);
    } // Merge the partial state and the previous state.


    var memoizedState = partialState === null || partialState === undefined ? prevState :
assign({}, prevState, partialState);
    workInProgress.memoizedState = memoizedState; // Once the update queue is empty,
persist the derived state onto the
    // base state.

    if (workInProgress.lanes === NoLanes) {
      // Queue is always non-null for classes
      var updateQueue = workInProgress.updateQueue;
      updateQueue.baseState = memoizedState;
    }
  }

  var classComponentUpdater = {
    isMounted: isMounted,
    enqueueSetState: function (inst, payload, callback) {
      var fiber = get(inst);
      var eventTime = requestEventTime();
      var lane = requestUpdateLane(fiber);
```

```
        var update = createUpdate(eventTime, lane);
        update.payload = payload;

        if (callback !== undefined && callback !== null) {
          {
            warnOnInvalidCallback(callback, 'setState');
          }

          update.callback = callback;
        }

        var root = enqueueUpdate(fiber, update, lane);

        if (root !== null) {
          scheduleUpdateOnFiber(root, fiber, lane, eventTime);
          entangleTransitions(root, fiber, lane);
        }

        {
          markStateUpdateScheduled(fiber, lane);
        }
      },
      enqueueReplaceState: function (inst, payload, callback) {
        var fiber = get(inst);
        var eventTime = requestEventTime();
        var lane = requestUpdateLane(fiber);
        var update = createUpdate(eventTime, lane);
        update.tag = ReplaceState;
        update.payload = payload;

        if (callback !== undefined && callback !== null) {
          {
            warnOnInvalidCallback(callback, 'replaceState');
          }

          update.callback = callback;
        }

        var root = enqueueUpdate(fiber, update, lane);

        if (root !== null) {
          scheduleUpdateOnFiber(root, fiber, lane, eventTime);
          entangleTransitions(root, fiber, lane);
        }

        {
          markStateUpdateScheduled(fiber, lane);
        }
      },
      enqueueForceUpdate: function (inst, callback) {
        var fiber = get(inst);
        var eventTime = requestEventTime();
        var lane = requestUpdateLane(fiber);
        var update = createUpdate(eventTime, lane);
        update.tag = ForceUpdate;

        if (callback !== undefined && callback !== null) {
          {
            warnOnInvalidCallback(callback, 'forceUpdate');
          }

          update.callback = callback;
        }

        var root = enqueueUpdate(fiber, update, lane);

        if (root !== null) {
          scheduleUpdateOnFiber(root, fiber, lane, eventTime);
          entangleTransitions(root, fiber, lane);
        }

        {
          markForceUpdateScheduled(fiber, lane);
        }
      }
    };

    function checkShouldComponentUpdate(workInProgress, ctor, oldProps, newProps, oldState,
  newState, nextContext) {
      var instance = workInProgress.stateNode;

      if (typeof instance.shouldComponentUpdate === 'function') {
        var shouldUpdate = instance.shouldComponentUpdate(newProps, newState, nextContext);

        {
          if ( workInProgress.mode & StrictLegacyMode) {
            setIsStrictModeForDevtools(true);

            try {
              // Invoke the function an extra time to help detect side-effects.
              shouldUpdate = instance.shouldComponentUpdate(newProps, newState,
  nextContext);
            } finally {
              setIsStrictModeForDevtools(false);
            }
          }

          if (shouldUpdate === undefined) {
            error('%s.shouldComponentUpdate(): Returned undefined instead of a ' + 'boolean
  value. Make sure to return true or false.', getComponentNameFromType(ctor) ||
  'Component');
          }
        }

        return shouldUpdate;
      }

      if (ctor.prototype && ctor.prototype.isPureReactComponent) {
        return !shallowEqual(oldProps, newProps) || !shallowEqual(oldState, newState);
      }

      return true;
    }

    function checkClassInstance(workInProgress, ctor, newProps) {
      var instance = workInProgress.stateNode;

      {
        var name = getComponentNameFromType(ctor) || 'Component';
        var renderPresent = instance.render;

        if (!renderPresent) {
          if (ctor.prototype && typeof ctor.prototype.render === 'function') {
            error('%s(...): No `render` method found on the returned component ' +
  'instance: did you accidentally return an object from the constructor?', name);
          } else {
            error('%s(...): No `render` method found on the returned component ' +
  'instance: you may have forgotten to define `render`.', name);
          }
        }

        if (instance.getInitialState && !instance.getInitialState.isReactClassApproved &&
  !instance.state) {
          error('getInitialState was defined on %s, a plain JavaScript class. ' + 'This is
  only supported for classes created using React.createClass. ' + 'Did you mean to define a
  state property instead?', name);
        }
        if (instance.getDefaultProps && !instance.getDefaultProps.isReactClassApproved) {
          error('getDefaultProps was defined on %s, a plain JavaScript class. ' + 'This is
  only supported for classes created using React.createClass. ' + 'Use a static property to
  define defaultProps instead.', name);
        }

        if (instance.propTypes) {
          error('propTypes was defined as an instance property on %s. Use a static ' +
  'property to define propTypes instead.', name);
        }

        if (instance.contextType) {
          error('contextType was defined as an instance property on %s. Use a static ' +
  'property to define contextType instead.', name);
        }

        {
          if (ctor.childContextTypes && !didWarnAboutLegacyContext$1.has(ctor) && // Strict
  Mode has its own warning for legacy context, so we can skip
          // this one.
          (workInProgress.mode & StrictLegacyMode) === NoMode) {
            didWarnAboutLegacyContext$1.add(ctor);

            error('%s uses the legacy childContextTypes API which is no longer ' +
  'supported and will be removed in the next major release. Use ' + 'React.createContext()
  instead\n\n.' + 'Learn more about this warning here: https://reactjs.org/link/legacy-
  context', name);
          }

          if (ctor.contextTypes && !didWarnAboutLegacyContext$1.has(ctor) && // Strict Mode
  has its own warning for legacy context, so we can skip
          // this one.
          (workInProgress.mode & StrictLegacyMode) === NoMode) {
            didWarnAboutLegacyContext$1.add(ctor);

            error('%s uses the legacy contextTypes API which is no longer supported ' +
  'and will be removed in the next major release. Use ' + 'React.createContext() with
  static contextType instead.\n\n' + 'Learn more about this warning here:
  https://reactjs.org/link/legacy-context', name);
          }

          if (instance.contextTypes) {
            error('contextTypes was defined as an instance property on %s. Use a static ' +
  'property to define contextTypes instead.', name);
          }

          if (ctor.contextType && ctor.contextTypes &&
  !didWarnAboutContextTypeAndContextTypes.has(ctor)) {
            didWarnAboutContextTypeAndContextTypes.add(ctor);

            error('%s declares both contextTypes and contextType static properties. ' +
  'The legacy contextTypes property will be ignored.', name);
          }
        }

        if (typeof instance.componentShouldUpdate === 'function') {
          error('%s has a method called ' + 'componentShouldUpdate(). Did you mean
  shouldComponentUpdate()? ' + 'The name is phrased as a question because the function is '
  + 'expected to return a value.', name);
        }

        if (ctor.prototype && ctor.prototype.isPureReactComponent && typeof
  instance.shouldComponentUpdate !== 'undefined') {
          error('%s has a method called shouldComponentUpdate(). ' + 'shouldComponentUpdate
  should not be used when extending React.PureComponent. ' + 'Please extend React.Component
  if shouldComponentUpdate is used.', getComponentNameFromType(ctor) || 'A pure
  component');
        }
        if (typeof instance.componentDidUnmount === 'function') {
          error('%s has a method called ' + 'componentDidUnmount(). But there is no such
  lifecycle method. ' + 'Did you mean componentWillUnmount()?', name);
        }

        if (typeof instance.componentDidReceiveProps === 'function') {
          error('%s has a method called ' + 'componentDidReceiveProps(). But there is no
  such lifecycle method. ' + 'If you meant to update the state in response to changing
  props, ' + 'use componentWillReceiveProps(). If you meant to fetch data or ' + 'run side-
  effects or mutations after React has updated the UI, use componentDidUpdate().', name);
        }

        if (typeof instance.componentWillRecieveProps === 'function') {
          error('%s has a method called ' + 'componentWillRecieveProps(). Did you mean
  componentWillReceiveProps()?', name);
        }

        if (typeof instance.UNSAFE_componentWillRecieveProps === 'function') {
          error('%s has a method called ' + 'UNSAFE_componentWillRecieveProps(). Did you
  mean UNSAFE_componentWillReceiveProps()?', name);
        }

        var hasMutatedProps = instance.props !== newProps;

        if (instance.props !== undefined && hasMutatedProps) {
          error('%s(...): When calling super() in `%s`, make sure to pass ' + "up the same
  props that your component's constructor was passed.", name, name);
        }

        if (instance.defaultProps) {
          error('Setting defaultProps as an instance property on %s is not supported and
  will be ignored.' + ' Instead, define defaultProps as a static property on %s.', name,
  name);
        }

        if (typeof instance.getSnapshotBeforeUpdate === 'function' && typeof
  instance.componentDidUpdate !== 'function' &&
  !didWarnAboutGetSnapshotBeforeUpdateWithoutDidUpdate.has(ctor)) {
          didWarnAboutGetSnapshotBeforeUpdateWithoutDidUpdate.add(ctor);

          error('%s: getSnapshotBeforeUpdate() should be used with componentDidUpdate(). '
  + 'This component defines getSnapshotBeforeUpdate() only.',
  getComponentNameFromType(ctor));
        }

        if (typeof instance.getDerivedStateFromProps === 'function') {
          error('%s: getDerivedStateFromProps() is defined as an instance method ' + 'and
  will be ignored. Instead, declare it as a static method.', name);
        }

        if (typeof instance.getDerivedStateFromError === 'function') {
          error('%s: getDerivedStateFromError() is defined as an instance method ' + 'and
  will be ignored. Instead, declare it as a static method.', name);
        }

        if (typeof ctor.getSnapshotBeforeUpdate === 'function') {
          error('%s: getSnapshotBeforeUpdate() is defined as a static method ' + 'and will
  be ignored. Instead, declare it as an instance method.', name);
        }

        var _state = instance.state;

        if (_state && (typeof _state !== 'object' || isArray(_state))) {
          error('%s.state: must be set to an object or null', name);
        }

        if (typeof instance.getChildContext === 'function' && typeof ctor.childContextTypes
  !== 'object') {
          error('%s.getChildContext(): childContextTypes must be defined in order to ' +
  'use getChildContext().', name);
        }
      }
    }

    function adoptClassInstance(workInProgress, instance) {
      instance.updater = classComponentUpdater;
      workInProgress.stateNode = instance; // The instance needs access to the fiber so
  that it can schedule updates

      set(instance, workInProgress);

      {
        instance._reactInternalInstance = fakeInternalInstance;
      }
    }

    function constructClassInstance(workInProgress, ctor, props) {
      var isLegacyContextConsumer = false;
      var unmaskedContext = emptyContextObject;
      var context = emptyContextObject;
      var contextType = ctor.contextType;

      {
        if ('contextType' in ctor) {
          var isValid = // Allow null for conditional declaration
          contextType === null || contextType !== undefined && contextType.$$typeof ===
  REACT_CONTEXT_TYPE && contextType._context === undefined; // Not a <Context.Consumer>

          if (!isValid && !didWarnAboutInvalidateContextType.has(ctor)) {
            didWarnAboutInvalidateContextType.add(ctor);
            var addendum = '';

            if (contextType === undefined) {
              addendum = ' However, it is set to undefined. ' + 'This can be caused by a
  typo or by mixing up named and default imports. ' + 'This can also happen due to a
  circular dependency, so ' + 'try moving the createContext() call to a separate file.';
            } else if (typeof contextType !== 'object') {
              addendum = ' However, it is set to a ' + typeof contextType + '.';
            } else if (contextType.$$typeof === REACT_PROVIDER_TYPE) {
              addendum = ' Did you accidentally pass the Context.Provider instead?';
            } else if (contextType._context !== undefined) {
              // <Context.Consumer>
              addendum = ' Did you accidentally pass the Context.Consumer instead?';
            } else {
              addendum = ' However, it is set to an object with keys {' +
  Object.keys(contextType).join(', ') + '}.';
            }

            error('%s defines an invalid contextType. ' + 'contextType should point to the
  Context object returned by React.createContext().%s', getComponentNameFromType(ctor) ||
  'Component', addendum);
          }
        }
      }

      if (typeof contextType === 'object' && contextType !== null) {
        context = readContext(contextType);
      } else {
        unmaskedContext = getUnmaskedContext(workInProgress, ctor, true);
        var contextTypes = ctor.contextTypes;
        isLegacyContextConsumer = contextTypes !== null && contextTypes !== undefined;
        context = isLegacyContextConsumer ? getMaskedContext(workInProgress,
  unmaskedContext) : emptyContextObject;
      }

      var instance = new ctor(props, context); // Instantiate twice to help detect side-
  effects.
```

```
    {
      if ( workInProgress.mode & StrictLegacyMode) {
        setIsStrictModeForDevtools(true);

        try {
          instance = new ctor(props, context); // eslint-disable-line no-new
        } finally {
          setIsStrictModeForDevtools(false);
        }
      }
    }

    var state = workInProgress.memoizedState = instance.state !== null && instance.state
!== undefined ? instance.state : null;
    adoptClassInstance(workInProgress, instance);

    {
      if (typeof ctor.getDerivedStateFromProps === 'function' && state === null) {
        var componentName = getComponentNameFromType(ctor) || 'Component';

        if (!didWarnAboutUninitializedState.has(componentName)) {
          didWarnAboutUninitializedState.add(componentName);

          error('`%s` uses `getDerivedStateFromProps` but its initial state is ' + '%s.
This is not recommended. Instead, define the initial state by ' + 'assigning an object to
`this.state` in the constructor of `%s`. ' + 'This ensures that
`getDerivedStateFromProps` arguments have a consistent shape.', componentName,
instance.state === null ? 'null' : 'undefined', componentName);
        }
      } // If new component APIs are defined, "unsafe" lifecycles won't be called.
      // Warn about these lifecycles if they are present.
      // Don't warn about react-lifecycles-compat polyfilled methods though.


      if (typeof ctor.getDerivedStateFromProps === 'function' || typeof
instance.getSnapshotBeforeUpdate === 'function') {
        var foundWillMountName = null;
        var foundWillReceivePropsName = null;
        var foundWillUpdateName = null;

        if (typeof instance.componentWillMount === 'function' &&
instance.componentWillMount.__suppressDeprecationWarning !== true) {
          foundWillMountName = 'componentWillMount';
        } else if (typeof instance.UNSAFE_componentWillMount === 'function') {
          foundWillMountName = 'UNSAFE_componentWillMount';
        }

        if (typeof instance.componentWillReceiveProps === 'function' &&
instance.componentWillReceiveProps.__suppressDeprecationWarning !== true) {
          foundWillReceivePropsName = 'componentWillReceiveProps';
        } else if (typeof instance.UNSAFE_componentWillReceiveProps === 'function') {
          foundWillReceivePropsName = 'UNSAFE_componentWillReceiveProps';
        }

        if (typeof instance.componentWillUpdate === 'function' &&
instance.componentWillUpdate.__suppressDeprecationWarning !== true) {
          foundWillUpdateName = 'componentWillUpdate';
        } else if (typeof instance.UNSAFE_componentWillUpdate === 'function') {
          foundWillUpdateName = 'UNSAFE_componentWillUpdate';
        }

        if (foundWillMountName !== null || foundWillReceivePropsName !== null ||
foundWillUpdateName !== null) {
          var _componentName = getComponentNameFromType(ctor) || 'Component';

          var newApiName = typeof ctor.getDerivedStateFromProps === 'function' ?
'getDerivedStateFromProps()' : 'getSnapshotBeforeUpdate()';

          if (!didWarnAboutLegacyLifecyclesAndDerivedState.has(_componentName)) {
            didWarnAboutLegacyLifecyclesAndDerivedState.add(_componentName);

            error('Unsafe legacy lifecycles will not be called for components using new
component APIs.\n\n' + '%s uses %s but also contains the following legacy
lifecycles:%s%s%s\n\n' + 'The above lifecycles should be removed. Learn more about this
warning here:\n' + 'https://reactjs.org/link/unsafe-component-lifecycles',
_componentName, newApiName, foundWillMountName !== null ? "\n  " + foundWillMountName :
'', foundWillReceivePropsName !== null ? "\n  " + foundWillReceivePropsName : '',
foundWillUpdateName !== null ? "\n  " + foundWillUpdateName : '');
          }
        }
      }
    } // Cache unmasked context so we can avoid recreating masked context unless
necessary.
    // ReactFiberContext usually updates this cache but can't for newly-created
instances.


    if (isLegacyContextConsumer) {
      cacheContext(workInProgress, unmaskedContext, context);
    }

    return instance;
  }

  function callComponentWillMount(workInProgress, instance) {
    var oldState = instance.state;

    if (typeof instance.componentWillMount === 'function') {
      instance.componentWillMount();
    }

    if (typeof instance.UNSAFE_componentWillMount === 'function') {
      instance.UNSAFE_componentWillMount();
    }

    if (oldState !== instance.state) {
      {
        error('%s.componentWillMount(): Assigning directly to this.state is ' +
"deprecated (except inside a component's " + 'constructor). Use setState instead.',
getComponentNameFromFiber(workInProgress) || 'Component');
      }

      classComponentUpdater.enqueueReplaceState(instance, instance.state, null);
    }
  }

  function callComponentWillReceiveProps(workInProgress, instance, newProps, nextContext)
{
    var oldState = instance.state;

    if (typeof instance.componentWillReceiveProps === 'function') {
      instance.componentWillReceiveProps(newProps, nextContext);
    }

    if (typeof instance.UNSAFE_componentWillReceiveProps === 'function') {
      instance.UNSAFE_componentWillReceiveProps(newProps, nextContext);
    }

    if (instance.state !== oldState) {
      {
        var componentName = getComponentNameFromFiber(workInProgress) || 'Component';

        if (!didWarnAboutStateAssignmentForComponent.has(componentName)) {
          didWarnAboutStateAssignmentForComponent.add(componentName);

          error('%s.componentWillReceiveProps(): Assigning directly to ' + "this.state is
deprecated (except inside a component's " + 'constructor). Use setState instead.',
componentName);
```

```
        }
      }

      classComponentUpdater.enqueueReplaceState(instance, instance.state, null);
    }
  } // Invokes the mount life-cycles on a previously never rendered instance.


  function mountClassInstance(workInProgress, ctor, newProps, renderLanes) {
    {
      checkClassInstance(workInProgress, ctor, newProps);
    }

    var instance = workInProgress.stateNode;
    instance.props = newProps;
    instance.state = workInProgress.memoizedState;
    instance.refs = {};
    initializeUpdateQueue(workInProgress);
    var contextType = ctor.contextType;

    if (typeof contextType === 'object' && contextType !== null) {
      instance.context = readContext(contextType);
    } else {
      var unmaskedContext = getUnmaskedContext(workInProgress, ctor, true);
      instance.context = getMaskedContext(workInProgress, unmaskedContext);
    }

    {
      if (instance.state === newProps) {
        var componentName = getComponentNameFromType(ctor) || 'Component';

        if (!didWarnAboutDirectlyAssigningPropsToState.has(componentName)) {
          didWarnAboutDirectlyAssigningPropsToState.add(componentName);

          error('%s: It is not recommended to assign props directly to state ' + "because
updates to props won't be reflected in state. " + 'In most cases, it is better to use
props directly.', componentName);
        }
      }

      if (workInProgress.mode & StrictLegacyMode) {
        ReactStrictModeWarnings.recordLegacyContextWarning(workInProgress, instance);
      }

      {
        ReactStrictModeWarnings.recordUnsafeLifecycleWarnings(workInProgress, instance);
      }
    }

    instance.state = workInProgress.memoizedState;
    var getDerivedStateFromProps = ctor.getDerivedStateFromProps;

    if (typeof getDerivedStateFromProps === 'function') {
      applyDerivedStateFromProps(workInProgress, ctor, getDerivedStateFromProps,
newProps);
      instance.state = workInProgress.memoizedState;
    } // In order to support react-lifecycles-compat polyfilled components,
    // Unsafe lifecycles should not be invoked for components using the new APIs.


    if (typeof ctor.getDerivedStateFromProps !== 'function' && typeof
instance.getSnapshotBeforeUpdate !== 'function' && (typeof
instance.UNSAFE_componentWillMount === 'function' || typeof instance.componentWillMount
=== 'function')) {
      callComponentWillMount(workInProgress, instance); // If we had additional state
updates during this life-cycle, let's
      // process them now.

      processUpdateQueue(workInProgress, newProps, instance, renderLanes);
```

```
        instance.state = workInProgress.memoizedState;
      }

      if (typeof instance.componentDidMount === 'function') {
        var fiberFlags = Update;

        {
          fiberFlags |= LayoutStatic;
        }

        if ( (workInProgress.mode & StrictEffectsMode) !== NoMode) {
          fiberFlags |= MountLayoutDev;
        }

        workInProgress.flags |= fiberFlags;
      }
    }

    function resumeMountClassInstance(workInProgress, ctor, newProps, renderLanes) {
      var instance = workInProgress.stateNode;
      var oldProps = workInProgress.memoizedProps;
      instance.props = oldProps;
      var oldContext = instance.context;
      var contextType = ctor.contextType;
      var nextContext = emptyContextObject;

      if (typeof contextType === 'object' && contextType !== null) {
        nextContext = readContext(contextType);
      } else {
        var nextLegacyUnmaskedContext = getUnmaskedContext(workInProgress, ctor, true);
        nextContext = getMaskedContext(workInProgress, nextLegacyUnmaskedContext);
      }

      var getDerivedStateFromProps = ctor.getDerivedStateFromProps;
      var hasNewLifecycles = typeof getDerivedStateFromProps === 'function' || typeof
  instance.getSnapshotBeforeUpdate === 'function'; // Note: During these life-cycles,
  instance.props/instance.state are what
      // ever the previously attempted to render - not the "current". However,
      // during componentDidUpdate we pass the "current" props.
      // In order to support react-lifecycles-compat polyfilled components,
      // Unsafe lifecycles should not be invoked for components using the new APIs.

      if (!hasNewLifecycles && (typeof instance.UNSAFE_componentWillReceiveProps ===
  'function' || typeof instance.componentWillReceiveProps === 'function')) {
        if (oldProps !== newProps || oldContext !== nextContext) {
          callComponentWillReceiveProps(workInProgress, instance, newProps, nextContext);
        }
      }

      resetHasForceUpdateBeforeProcessing();
      var oldState = workInProgress.memoizedState;
      var newState = instance.state = oldState;
      processUpdateQueue(workInProgress, newProps, instance, renderLanes);
      newState = workInProgress.memoizedState;

      if (oldProps === newProps && oldState === newState && !hasContextChanged() &&
  !checkHasForceUpdateAfterProcessing()) {
        // If an update was already in progress, we should schedule an Update
        // effect even though we're bailing out, so that cWU/cDU are called.
        if (typeof instance.componentDidMount === 'function') {
          var fiberFlags = Update;

          {
            fiberFlags |= LayoutStatic;
          }

          if ( (workInProgress.mode & StrictEffectsMode) !== NoMode) {
            fiberFlags |= MountLayoutDev;
          }
          workInProgress.flags |= fiberFlags;
        }

        return false;
      }

      if (typeof getDerivedStateFromProps === 'function') {
        applyDerivedStateFromProps(workInProgress, ctor, getDerivedStateFromProps,
  newProps);
        newState = workInProgress.memoizedState;
      }

      var shouldUpdate = checkHasForceUpdateAfterProcessing() ||
  checkShouldComponentUpdate(workInProgress, ctor, oldProps, newProps, oldState, newState,
  nextContext);

      if (shouldUpdate) {
        // In order to support react-lifecycles-compat polyfilled components,
        // Unsafe lifecycles should not be invoked for components using the new APIs.
        if (!hasNewLifecycles && (typeof instance.UNSAFE_componentWillMount === 'function'
  || typeof instance.componentWillMount === 'function')) {
          if (typeof instance.componentWillMount === 'function') {
            instance.componentWillMount();
          }

          if (typeof instance.UNSAFE_componentWillMount === 'function') {
            instance.UNSAFE_componentWillMount();
          }
        }

        if (typeof instance.componentDidMount === 'function') {
          var _fiberFlags = Update;

          {
            _fiberFlags |= LayoutStatic;
          }

          if ( (workInProgress.mode & StrictEffectsMode) !== NoMode) {
            _fiberFlags |= MountLayoutDev;
          }

          workInProgress.flags |= _fiberFlags;
        }
      } else {
        // If an update was already in progress, we should schedule an Update
        // effect even though we're bailing out, so that cWU/cDU are called.
        if (typeof instance.componentDidMount === 'function') {
          var _fiberFlags2 = Update;

          {
            _fiberFlags2 |= LayoutStatic;
          }

          if ( (workInProgress.mode & StrictEffectsMode) !== NoMode) {
            _fiberFlags2 |= MountLayoutDev;
          }

          workInProgress.flags |= _fiberFlags2;
        } // If shouldComponentUpdate returned false, we should still update the
        // memoized state to indicate that this work can be reused.


        workInProgress.memoizedProps = newProps;
        workInProgress.memoizedState = newState;
      } // Update the existing instance's state, props, and context pointers even
      // if shouldComponentUpdate returns false.
```

```
      instance.props = newProps;
      instance.state = newState;
      instance.context = nextContext;
      return shouldUpdate;
    } // Invokes the update life-cycles and returns false if it shouldn't rerender.


    function updateClassInstance(current, workInProgress, ctor, newProps, renderLanes) {
      var instance = workInProgress.stateNode;
      cloneUpdateQueue(current, workInProgress);
      var unresolvedOldProps = workInProgress.memoizedProps;
      var oldProps = workInProgress.type === workInProgress.elementType ?
  unresolvedOldProps : resolveDefaultProps(workInProgress.type, unresolvedOldProps);
      instance.props = oldProps;
      var unresolvedNewProps = workInProgress.pendingProps;
      var oldContext = instance.context;
      var contextType = ctor.contextType;
      var nextContext = emptyContextObject;

      if (typeof contextType === 'object' && contextType !== null) {
        nextContext = readContext(contextType);
      } else {
        var nextUnmaskedContext = getUnmaskedContext(workInProgress, ctor, true);
        nextContext = getMaskedContext(workInProgress, nextUnmaskedContext);
      }

      var getDerivedStateFromProps = ctor.getDerivedStateFromProps;
      var hasNewLifecycles = typeof getDerivedStateFromProps === 'function' || typeof
  instance.getSnapshotBeforeUpdate === 'function'; // Note: During these life-cycles,
  instance.props/instance.state are what
      // ever the previously attempted to render - not the "current". However,
      // during componentDidUpdate we pass the "current" props.
      // In order to support react-lifecycles-compat polyfilled components,
      // Unsafe lifecycles should not be invoked for components using the new APIs.

      if (!hasNewLifecycles && (typeof instance.UNSAFE_componentWillReceiveProps ===
  'function' || typeof instance.componentWillReceiveProps === 'function')) {
        if (unresolvedOldProps !== unresolvedNewProps || oldContext !== nextContext) {
          callComponentWillReceiveProps(workInProgress, instance, newProps, nextContext);
        }
      }

      resetHasForceUpdateBeforeProcessing();
      var oldState = workInProgress.memoizedState;
      var newState = instance.state = oldState;
      processUpdateQueue(workInProgress, newProps, instance, renderLanes);
      newState = workInProgress.memoizedState;

      if (unresolvedOldProps === unresolvedNewProps && oldState === newState &&
  !hasContextChanged() && !checkHasForceUpdateAfterProcessing() && !
  (enableLazyContextPropagation   )) {
        // If an update was already in progress, we should schedule an Update
        // effect even though we're bailing out, so that cWU/cDU are called.
        if (typeof instance.componentDidUpdate === 'function') {
          if (unresolvedOldProps !== current.memoizedProps || oldState !==
  current.memoizedState) {
            workInProgress.flags |= Update;
          }
        }

        if (typeof instance.getSnapshotBeforeUpdate === 'function') {
          if (unresolvedOldProps !== current.memoizedProps || oldState !==
  current.memoizedState) {
            workInProgress.flags |= Snapshot;
          }
        }

        return false;
      }
      if (typeof getDerivedStateFromProps === 'function') {
        applyDerivedStateFromProps(workInProgress, ctor, getDerivedStateFromProps,
  newProps);
        newState = workInProgress.memoizedState;
      }

      var shouldUpdate = checkHasForceUpdateAfterProcessing() ||
  checkShouldComponentUpdate(workInProgress, ctor, oldProps, newProps, oldState, newState,
  nextContext) || // TODO: In some cases, we'll end up checking if context has changed
  twice,
      // both before and after `shouldComponentUpdate` has been called. Not ideal,
      // but I'm loath to refactor this function. This only happens for memoized
      // components so it's not that common.
      enableLazyContextPropagation   ;

      if (shouldUpdate) {
        // In order to support react-lifecycles-compat polyfilled components,
        // Unsafe lifecycles should not be invoked for components using the new APIs.
        if (!hasNewLifecycles && (typeof instance.UNSAFE_componentWillUpdate === 'function'
  || typeof instance.componentWillUpdate === 'function')) {
          if (typeof instance.componentWillUpdate === 'function') {
            instance.componentWillUpdate(newProps, newState, nextContext);
          }

          if (typeof instance.UNSAFE_componentWillUpdate === 'function') {
            instance.UNSAFE_componentWillUpdate(newProps, newState, nextContext);
          }
        }

        if (typeof instance.componentDidUpdate === 'function') {
          workInProgress.flags |= Update;
        }

        if (typeof instance.getSnapshotBeforeUpdate === 'function') {
          workInProgress.flags |= Snapshot;
        }
      } else {
        // If an update was already in progress, we should schedule an Update
        // effect even though we're bailing out, so that cWU/cDU are called.
        if (typeof instance.componentDidUpdate === 'function') {
          if (unresolvedOldProps !== current.memoizedProps || oldState !==
  current.memoizedState) {
            workInProgress.flags |= Update;
          }
        }

        if (typeof instance.getSnapshotBeforeUpdate === 'function') {
          if (unresolvedOldProps !== current.memoizedProps || oldState !==
  current.memoizedState) {
            workInProgress.flags |= Snapshot;
          }
        } // If shouldComponentUpdate returned false, we should still update the
        // memoized props/state to indicate that this work can be reused.


        workInProgress.memoizedProps = newProps;
        workInProgress.memoizedState = newState;
      } // Update the existing instance's state, props, and context pointers even
      // if shouldComponentUpdate returns false.


      instance.props = newProps;
      instance.state = newState;
      instance.context = nextContext;
      return shouldUpdate;
    }

    function createCapturedValueAtFiber(value, source) {
      // If the value is an error, call this function immediately after it is thrown
      // so the stack is accurate.
      return {
        value: value,
        source: source,
        stack: getStackByFiberInDevAndProd(source),
        digest: null
      };
    }
    function createCapturedValue(value, digest, stack) {
      return {
        value: value,
        source: null,
        stack: stack != null ? stack : null,
        digest: digest != null ? digest : null
      };
    }

    // This module is forked in different environments.
    // By default, return `true` to log errors to the console.
    // Forks can return `false` if this isn't desirable.
    function showErrorDialog(boundary, errorInfo) {
      return true;
    }

    function logCapturedError(boundary, errorInfo) {
      try {
        var logError = showErrorDialog(boundary, errorInfo); // Allow injected
  showErrorDialog() to prevent default console.error logging.
        // This enables renderers like ReactNative to better manage redbox behavior.

        if (logError === false) {
          return;
        }

        var error = errorInfo.value;

        if (true) {
          var source = errorInfo.source;
          var stack = errorInfo.stack;
          var componentStack = stack !== null ? stack : ''; // Browsers support silencing
  uncaught errors by calling
          // `preventDefault()` in window `error` handler.
          // We record this information as an expando on the error.

          if (error != null && error._suppressLogging) {
            if (boundary.tag === ClassComponent) {
              // The error is recoverable and was silenced.
              // Ignore it and don't print the stack addendum.
              // This is handy for testing error boundaries without noise.
              return;
            } // The error is fatal. Since the silencing might have
            // been accidental, we'll surface it anyway.
            // However, the browser would have silenced the original error
            // so we'll print it first, and then print the stack addendum.


            console['error'](error); // Don't transform to our wrapper
            // For a more detailed description of this block, see:
            // https://github.com/facebook/react/pull/13384
          }

          var componentName = source ? getComponentNameFromFiber(source) : null;
          var componentNameMessage = componentName ? "The above error occurred in the <" +
  componentName + "> component:" : 'The above error occurred in one of your React
  components:';
          var errorBoundaryMessage;

          if (boundary.tag === HostRoot) {
            errorBoundaryMessage = 'Consider adding an error boundary to your tree to
  customize error handling behavior.\n' + 'Visit https://reactjs.org/link/error-boundaries
  to learn more about error boundaries.';
          } else {
            var errorBoundaryName = getComponentNameFromFiber(boundary) || 'Anonymous';
            errorBoundaryMessage = "React will try to recreate this component tree from
  scratch " + ("using the error boundary you provided, " + errorBoundaryName + ".");
          }

          var combinedMessage = componentNameMessage + "\n" + componentStack + "\n\n" + (""
  + errorBoundaryMessage); // In development, we provide our own message with just the
  component stack.
          // We don't include the original error message and JS stack because the browser
          // has already printed it. Even if the application swallows the error, it is
  still
          // displayed by the browser thanks to the DEV-only fake event trick in
  ReactErrorUtils.

          console['error'](combinedMessage); // Don't transform to our wrapper
        } else {
          // In production, we print the error directly.
          // This will include the message, the JS stack, and anything the browser wants to
  show.
          // We pass the error object instead of custom message so that the browser
  displays the error natively.
          console['error'](error); // Don't transform to our wrapper
        }
      } catch (e) {
        // This method must not throw, or React internal state will get messed up.
        // If console.error is overridden, or logCapturedError() shows a dialog that
  throws,
        // we want to report this error outside of the normal stack as a last resort.
        // https://github.com/facebook/react/issues/13188
        setTimeout(function () {
          throw e;
        });
      }
    }

    var PossiblyWeakMap$1 = typeof WeakMap === 'function' ? WeakMap : Map;

    function createRootErrorUpdate(fiber, errorInfo, lane) {
      var update = createUpdate(NoTimestamp, lane); // Unmount the root by rendering null.

      update.tag = CaptureUpdate; // Caution: React DevTools currently depends on this
  property
      // being called "element".

      update.payload = {
        element: null
      };
      var error = errorInfo.value;

      update.callback = function () {
        onUncaughtError(error);
        logCapturedError(fiber, errorInfo);
      };

      return update;
    }

    function createClassErrorUpdate(fiber, errorInfo, lane) {
      var update = createUpdate(NoTimestamp, lane);
      update.tag = CaptureUpdate;
      var getDerivedStateFromError = fiber.type.getDerivedStateFromError;

      if (typeof getDerivedStateFromError === 'function') {
        var error$1 = errorInfo.value;
```

```
        update.payload = function () {
          return getDerivedStateFromError(error$1);
        };

        update.callback = function () {
          {
            markFailedErrorBoundaryForHotReloading(fiber);
          }

          logCapturedError(fiber, errorInfo);
        };
      }

      var inst = fiber.stateNode;

      if (inst !== null && typeof inst.componentDidCatch === 'function') {
        update.callback = function callback() {
          {
            markFailedErrorBoundaryForHotReloading(fiber);
          }

          logCapturedError(fiber, errorInfo);

          if (typeof getDerivedStateFromError !== 'function') {
            // To preserve the preexisting retry behavior of error boundaries,
            // we keep track of which ones already failed during this batch.
            // This gets reset before we yield back to the browser.
            // TODO: Warn in strict mode if getDerivedStateFromError is
            // not defined.
            markLegacyErrorBoundaryAsFailed(this);
          }

          var error$1 = errorInfo.value;
          var stack = errorInfo.stack;
          this.componentDidCatch(error$1, {
            componentStack: stack !== null ? stack : ''
          });

          {
            if (typeof getDerivedStateFromError !== 'function') {
              // If componentDidCatch is the only error boundary method defined,
              // then it needs to call setState to recover from errors.
              // If no state update is scheduled then the boundary will swallow the error.
              if (!includesSomeLane(fiber.lanes, SyncLane)) {
                error('%s: Error boundaries should implement getDerivedStateFromError(). '
 + 'In that method, return a state update to display an error message or fallback UI.',
 getComponentNameFromFiber(fiber) || 'Unknown');
              }
            }
          }
        };
      }

      return update;
    }

    function attachPingListener(root, wakeable, lanes) {
      // Attach a ping listener
      //
      // The data might resolve before we have a chance to commit the fallback. Or,
      // in the case of a refresh, we'll never commit a fallback. So we need to
      // attach a listener now. When it resolves ("pings"), we can decide whether to
      // try rendering the tree again.
      //
      // Only attach a listener if one does not already exist for the lanes
      // we're currently rendering (which acts like a "thread ID" here).
      //
      // We only need to do this in concurrent mode. Legacy Suspense always
      // commits fallbacks synchronously, so there are no pings.
      var pingCache = root.pingCache;
      var threadIDs;

      if (pingCache === null) {
        pingCache = root.pingCache = new PossiblyWeakMap$1();
        threadIDs = new Set();
        pingCache.set(wakeable, threadIDs);
      } else {
        threadIDs = pingCache.get(wakeable);

        if (threadIDs === undefined) {
          threadIDs = new Set();
          pingCache.set(wakeable, threadIDs);
        }
      }

      if (!threadIDs.has(lanes)) {
        // Memoize using the thread ID to prevent redundant listeners.
        threadIDs.add(lanes);
        var ping = pingSuspendedRoot.bind(null, root, wakeable, lanes);

        {
          if (isDevToolsPresent) {
            // If we have pending work still, restore the original updaters
            restorePendingUpdaters(root, lanes);
          }
        }

        wakeable.then(ping, ping);
      }
    }

    function attachRetryListener(suspenseBoundary, root, wakeable, lanes) {
      // Retry listener
      //
      // If the fallback does commit, we need to attach a different type of
      // listener. This one schedules an update on the Suspense boundary to turn
      // the fallback state off.
      //
      // Stash the wakeable on the boundary fiber so we can access it in the
      // commit phase.
      //
      // When the wakeable resolves, we'll attempt to render the boundary
      // again ("retry").
      var wakeables = suspenseBoundary.updateQueue;

      if (wakeables === null) {
        var updateQueue = new Set();
        updateQueue.add(wakeable);
        suspenseBoundary.updateQueue = updateQueue;
      } else {
        wakeables.add(wakeable);
      }
    }

    function resetSuspendedComponent(sourceFiber, rootRenderLanes) {
      // A legacy mode Suspense quirk, only relevant to hook components.


      var tag = sourceFiber.tag;

      if ((sourceFiber.mode & ConcurrentMode) === NoMode && (tag === FunctionComponent ||
  tag === ForwardRef || tag === SimpleMemoComponent)) {
        var currentSource = sourceFiber.alternate;

        if (currentSource) {
          sourceFiber.updateQueue = currentSource.updateQueue;
          sourceFiber.memoizedState = currentSource.memoizedState;
          sourceFiber.lanes = currentSource.lanes;
        } else {
          sourceFiber.updateQueue = null;
          sourceFiber.memoizedState = null;
        }
      }
    }

    function getNearestSuspenseBoundaryToCapture(returnFiber) {
      var node = returnFiber;

      do {
        if (node.tag === SuspenseComponent && shouldCaptureSuspense(node)) {
          return node;
        } // This boundary already captured during this render. Continue to the next
        // boundary.


        node = node.return;
      } while (node !== null);

      return null;
    }

    function markSuspenseBoundaryShouldCapture(suspenseBoundary, returnFiber, sourceFiber,
  root, rootRenderLanes) {
      // This marks a Suspense boundary so that when we're unwinding the stack,
      // it captures the suspended "exception" and does a second (fallback) pass.
      if ((suspenseBoundary.mode & ConcurrentMode) === NoMode) {
        // Legacy Mode Suspense
        //
        // If the boundary is in legacy mode, we should *not*
        // suspend the commit. Pretend as if the suspended component rendered
        // null and keep rendering. When the Suspense boundary completes,
        // we'll do a second pass to render the fallback.
        if (suspenseBoundary === returnFiber) {
          // Special case where we suspended while reconciling the children of
          // a Suspense boundary's inner Offscreen wrapper fiber. This happens
          // when a React.lazy component is a direct child of a
          // Suspense boundary.
          //
          // Suspense boundaries are implemented as multiple fibers, but they
          // are a single conceptual unit. The legacy mode behavior where we
          // pretend the suspended fiber committed as `null` won't work,
          // because in this case the "suspended" fiber is the inner
          // Offscreen wrapper.
          //
          // Because the contents of the boundary haven't started rendering
          // yet (i.e. nothing in the tree has partially rendered) we can
          // switch to the regular, concurrent mode behavior: mark the
          // boundary with ShouldCapture and enter the unwind phase.
          suspenseBoundary.flags |= ShouldCapture;
        } else {
          suspenseBoundary.flags |= DidCapture;
          sourceFiber.flags |= ForceUpdateForLegacySuspense; // We're going to commit this
  fiber even though it didn't complete.
          // But we shouldn't call any lifecycle methods or callbacks. Remove
          // all lifecycle effect tags.

          sourceFiber.flags &= ~(LifecycleEffectMask | Incomplete);

          if (sourceFiber.tag === ClassComponent) {
            var currentSourceFiber = sourceFiber.alternate;

            if (currentSourceFiber === null) {
              // This is a new mount. Change the tag so it's not mistaken for a
              // completed class component. For example, we should not call
              // componentWillUnmount if it is deleted.
              sourceFiber.tag = IncompleteClassComponent;
            } else {
              // When we try rendering again, we should not reuse the current fiber,
              // since it's known to be in an inconsistent state. Use a force update to
              // prevent a bail out.
              var update = createUpdate(NoTimestamp, SyncLane);
              update.tag = ForceUpdate;
              enqueueUpdate(sourceFiber, update, SyncLane);
            }
          } // The source fiber did not complete. Mark it with Sync priority to
          // indicate that it still has pending work.


          sourceFiber.lanes = mergeLanes(sourceFiber.lanes, SyncLane);
        }

        return suspenseBoundary;
      } // Confirmed that the boundary is in a concurrent mode tree. Continue
      // with the normal suspend path.
      //
      // After this we'll use a set of heuristics to determine whether this
      // render pass will run to completion or restart or "suspend" the commit.
      // The actual logic for this is spread out in different places.
      //
      // This first principle is that if we're going to suspend when we complete
      // a root, then we should also restart if we get an update or ping that
      // might unsuspend it, and vice versa. The only reason to suspend is
      // because you think you might want to restart before committing. However,
      // it doesn't make sense to restart only while in the period we're suspended.
      //
      // Restarting too aggressively is also not good because it starves out any
      // intermediate loading state. So we use heuristics to determine when.
      // Suspense Heuristics
      //
      // If nothing threw a Promise or all the same fallbacks are already showing,
      // then don't suspend/restart.
      //
      // If this is an initial render of a new tree of Suspense boundaries and
      // those trigger a fallback, then don't suspend/restart. We want to ensure
      // that we can show the initial loading state as quickly as possible.
      //
      // If we hit a "Delayed" case, such as when we'd switch from content back into
      // a fallback, then we should always suspend/restart. Transitions apply
      // to this case. If none is defined, JND is used instead.
      //
      // If we're already showing a fallback and it gets "retried", allowing us to show
      // another level, but there's still an inner boundary that would show a fallback,
      // then we suspend/restart for 500ms since the last time we showed a fallback
      // anywhere in the tree. This effectively throttles progressive loading into a
      // consistent train of commits. This also gives us an opportunity to restart to
      // get to the completed state slightly earlier.
      //
      // If there's ambiguity due to batching it's resolved in preference of:
      // 1) "delayed", 2) "initial render", 3) "retry".
      //
      // We want to ensure that a "busy" state doesn't get force committed. We want to
      // ensure that new initial loading states can commit as soon as possible.


      suspenseBoundary.flags |= ShouldCapture; // TODO: I think we can remove this, since
  we now use `DidCapture` in
      // the begin phase to prevent an early bailout.

      suspenseBoundary.lanes = rootRenderLanes;
      return suspenseBoundary;
    }

    function throwException(root, returnFiber, sourceFiber, value, rootRenderLanes) {
      // The source fiber did not complete.
      sourceFiber.flags |= Incomplete;
      {
        if (isDevToolsPresent) {
          // If we have pending work still, restore the original updaters
          restorePendingUpdaters(root, rootRenderLanes);
        }
      }

      if (value !== null && typeof value === 'object' && typeof value.then === 'function')
  {
        // This is a wakeable. The component suspended.
        var wakeable = value;
        resetSuspendedComponent(sourceFiber);

        {
          if (getIsHydrating() && sourceFiber.mode & ConcurrentMode) {
            markDidThrowWhileHydratingDEV();
          }
        }


        var suspenseBoundary = getNearestSuspenseBoundaryToCapture(returnFiber);

        if (suspenseBoundary !== null) {
          suspenseBoundary.flags &= ~ForceClientRender;
          markSuspenseBoundaryShouldCapture(suspenseBoundary, returnFiber, sourceFiber,
  root, rootRenderLanes); // We only attach ping listeners in concurrent mode. Legacy
  Suspense always
          // commits fallbacks synchronously, so there are no pings.

          if (suspenseBoundary.mode & ConcurrentMode) {
            attachPingListener(root, wakeable, rootRenderLanes);
          }

          attachRetryListener(suspenseBoundary, root, wakeable);
          return;
        } else {
          // No boundary was found. Unless this is a sync update, this is OK.
          // We can suspend and wait for more data to arrive.
          if (!includesSyncLane(rootRenderLanes)) {
            // This is not a sync update. Suspend. Since we're not activating a
            // Suspense boundary, this will unwind all the way to the root without
            // performing a second pass to render a fallback. (This is arguably how
            // refresh transitions should work, too, since we're not going to commit
            // the fallbacks anyway.)
            //
            // This case also applies to initial hydration.
            attachPingListener(root, wakeable, rootRenderLanes);
            renderDidSuspendDelayIfPossible();
            return;
          } // This is a sync/discrete update. We treat this case like an error
          // because discrete renders are expected to produce a complete tree
          // synchronously to maintain consistency with external state.


          var uncaughtSuspenseError = new Error('A component suspended while responding to
  synchronous input. This ' + 'will cause the UI to be replaced with a loading indicator.
  To ' + 'fix, updates that suspend should be wrapped ' + 'with startTransition.'); // If
  we're outside a transition, fall through to the regular error path.
          // The error will be caught by the nearest suspense boundary.

          value = uncaughtSuspenseError;
        }
      } else {
        // This is a regular error, not a Suspense wakeable.
        if (getIsHydrating() && sourceFiber.mode & ConcurrentMode) {
          markDidThrowWhileHydratingDEV();

          var _suspenseBoundary = getNearestSuspenseBoundaryToCapture(returnFiber); // If
  the error was thrown during hydration, we may be able to recover by
```

```
        // discarding the dehydrated content and switching to a client render.
        // Instead of surfacing the error, find the nearest Suspense boundary
        // and render it again without hydration.


        if (_suspenseBoundary !== null) {
          if ((_suspenseBoundary.flags & ShouldCapture) === NoFlags) {
            // Set a flag to indicate that we should try rendering the normal
            // children again, not the fallback.
            _suspenseBoundary.flags |= ForceClientRender;
          }

          markSuspenseBoundaryShouldCapture(_suspenseBoundary, returnFiber, sourceFiber,
root, rootRenderLanes); // Even though the user may not be affected by this error, we
should
          // still log it so it can be fixed.

          queueHydrationError(createCapturedValueAtFiber(value, sourceFiber));
          return;
        }
      }
    }

    value = createCapturedValueAtFiber(value, sourceFiber);
    renderDidError(value); // We didn't find a boundary that could handle this type of
exception. Start
    // over and traverse parent path again, this time treating the exception
    // as an error.

    var workInProgress = returnFiber;

    do {
      switch (workInProgress.tag) {
        case HostRoot:
          {
            var _errorInfo = value;
            workInProgress.flags |= ShouldCapture;
            var lane = pickArbitraryLane(rootRenderLanes);
            workInProgress.lanes = mergeLanes(workInProgress.lanes, lane);
            var update = createRootErrorUpdate(workInProgress, _errorInfo, lane);
            enqueueCapturedUpdate(workInProgress, update);
            return;
          }

        case ClassComponent:
          // Capture and retry
          var errorInfo = value;
          var ctor = workInProgress.type;
          var instance = workInProgress.stateNode;

          if ((workInProgress.flags & DidCapture) === NoFlags && (typeof
ctor.getDerivedStateFromError === 'function' || instance !== null && typeof
instance.componentDidCatch === 'function' &&
!isAlreadyFailedLegacyErrorBoundary(instance))) {
            workInProgress.flags |= ShouldCapture;

            var _lane = pickArbitraryLane(rootRenderLanes);

            workInProgress.lanes = mergeLanes(workInProgress.lanes, _lane); // Schedule
the error boundary to re-render using updated state

            var _update = createClassErrorUpdate(workInProgress, errorInfo, _lane);

            enqueueCapturedUpdate(workInProgress, _update);
            return;
          }

          break;
      }
      workInProgress = workInProgress.return;
    } while (workInProgress !== null);
  }

  function getSuspendedCache() {
    {
      return null;
    } // This function is called when a Suspense boundary suspends. It returns the
  }

  var ReactCurrentOwner$1 = ReactSharedInternals.ReactCurrentOwner;
  var didReceiveUpdate = false;
  var didWarnAboutBadClass;
  var didWarnAboutModulePatternComponent;
  var didWarnAboutContextTypeOnFunctionComponent;
  var didWarnAboutGetDerivedStateOnFunctionComponent;
  var didWarnAboutFunctionRefs;
  var didWarnAboutReassigningProps;
  var didWarnAboutRevealOrder;
  var didWarnAboutTailOptions;
  var didWarnAboutDefaultPropsOnFunctionComponent;

  {
    didWarnAboutBadClass = {};
    didWarnAboutModulePatternComponent = {};
    didWarnAboutContextTypeOnFunctionComponent = {};
    didWarnAboutGetDerivedStateOnFunctionComponent = {};
    didWarnAboutFunctionRefs = {};
    didWarnAboutReassigningProps = false;
    didWarnAboutRevealOrder = {};
    didWarnAboutTailOptions = {};
    didWarnAboutDefaultPropsOnFunctionComponent = {};
  }

  function reconcileChildren(current, workInProgress, nextChildren, renderLanes) {
    if (current === null) {
      // If this is a fresh new component that hasn't been rendered yet, we
      // won't update its child set by applying minimal side-effects. Instead,
      // we will add them all to the child before it gets rendered. That means
      // we can optimize this reconciliation pass by not tracking side-effects.
      workInProgress.child = mountChildFibers(workInProgress, null, nextChildren,
renderLanes);
    } else {
      // If the current child is the same as the work in progress, it means that
      // we haven't yet started any work on these children. Therefore, we use
      // the clone algorithm to create a copy of all the current children.
      // If we had any progressed work already, that is invalid at this point so
      // let's throw it out.
      workInProgress.child = reconcileChildFibers(workInProgress, current.child,
nextChildren, renderLanes);
    }
  }

  function forceUnmountCurrentAndReconcile(current, workInProgress, nextChildren,
renderLanes) {
    // This function is fork of reconcileChildren. It's used in cases where we
    // want to reconcile without matching against the existing set. This has the
    // effect of all current children being unmounted; even if the type and key
    // are the same, the old child is unmounted and a new child is created.
    //
    // To do this, we're going to go through the reconcile algorithm twice. In
    // the first pass, we schedule a deletion for all the current children by
    // passing null.
    workInProgress.child = reconcileChildFibers(workInProgress, current.child, null,
renderLanes); // In the second pass, we mount the new children. The trick here is that we
    // pass null in place of where we usually pass the current child set. This has
    // the effect of remounting all children regardless of whether their
    // identities match.
```

```
      workInProgress.child = reconcileChildFibers(workInProgress, null, nextChildren,
renderLanes);
    }

    function updateForwardRef(current, workInProgress, Component, nextProps, renderLanes) {
      // TODO: current can be non-null here even if the component
      // hasn't yet mounted. This happens after the first render suspends.
      // We'll need to figure out if this is fine or can cause issues.
      {
        if (workInProgress.type !== workInProgress.elementType) {
          // Lazy component props can't be validated in createElement
          // because they're only guaranteed to be resolved here.
          var innerPropTypes = Component.propTypes;

          if (innerPropTypes) {
            checkPropTypes(innerPropTypes, nextProps, // Resolved props
            'prop', getComponentNameFromType(Component));
          }
        }
      }

      var render = Component.render;
      var ref = workInProgress.ref; // The rest is a fork of updateFunctionComponent

      var nextChildren;
      var hasId;
      prepareToReadContext(workInProgress, renderLanes);

      {
        markComponentRenderStarted(workInProgress);
      }

      {
        ReactCurrentOwner$1.current = workInProgress;
        setIsRendering(true);
        nextChildren = renderWithHooks(current, workInProgress, render, nextProps, ref,
renderLanes);
        hasId = checkDidRenderIdHook();

        if ( workInProgress.mode & StrictLegacyMode) {
          setIsStrictModeForDevtools(true);

          try {
            nextChildren = renderWithHooks(current, workInProgress, render, nextProps, ref,
renderLanes);
            hasId = checkDidRenderIdHook();
          } finally {
            setIsStrictModeForDevtools(false);
          }
        }

        setIsRendering(false);
      }

      {
        markComponentRenderStopped();
      }

      if (current !== null && !didReceiveUpdate) {
        bailoutHooks(current, workInProgress, renderLanes);
        return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
      }

      if (getIsHydrating() && hasId) {
        pushMaterializedTreeId(workInProgress);
      } // React DevTools reads this flag.
      workInProgress.flags |= PerformedWork;
      reconcileChildren(current, workInProgress, nextChildren, renderLanes);
      return workInProgress.child;
    }

    function updateMemoComponent(current, workInProgress, Component, nextProps, 
  renderLanes) {
      if (current === null) {
        var type = Component.type;

        if (isSimpleFunctionComponent(type) && Component.compare === null && // 
  SimpleMemoComponent codepath doesn't resolve outer props either.
        Component.defaultProps === undefined) {
          var resolvedType = type;

          {
            resolvedType = resolveFunctionForHotReloading(type);
          } // If this is a plain function component without default props,
          // and with only the default shallow comparison, we upgrade it
          // to a SimpleMemoComponent to allow fast path updates.


          workInProgress.tag = SimpleMemoComponent;
          workInProgress.type = resolvedType;

          {
            validateFunctionComponentInDev(workInProgress, type);
          }

          return updateSimpleMemoComponent(current, workInProgress, resolvedType, 
  nextProps, renderLanes);
        }

        {
          var innerPropTypes = type.propTypes;

          if (innerPropTypes) {
            // Inner memo component props aren't currently validated in createElement.
            // We could move it there, but we'd still need this for lazy code path.
            checkPropTypes(innerPropTypes, nextProps, // Resolved props
            'prop', getComponentNameFromType(type));
          }

          if ( Component.defaultProps !== undefined) {
            var componentName = getComponentNameFromType(type) || 'Unknown';

            if (!didWarnAboutDefaultPropsOnFunctionComponent[componentName]) {
              error('%s: Support for defaultProps will be removed from memo components ' + 
  'in a future major release. Use JavaScript default parameters instead.', componentName);

              didWarnAboutDefaultPropsOnFunctionComponent[componentName] = true;
            }
          }
        }

        var child = createFiberFromTypeAndProps(Component.type, null, nextProps, 
  workInProgress, workInProgress.mode, renderLanes);
        child.ref = workInProgress.ref;
        child.return = workInProgress;
        workInProgress.child = child;
        return child;
      }

      {
        var _type = Component.type;
        var _innerPropTypes = _type.propTypes;

        if (_innerPropTypes) {
          // Inner memo component props aren't currently validated in createElement.
          // We could move it there, but we'd still need this for lazy code path.
          checkPropTypes(_innerPropTypes, nextProps, // Resolved props
          'prop', getComponentNameFromType(_type));
        }
      }

      var currentChild = current.child; // This is always exactly one child

      var hasScheduledUpdateOrContext = checkScheduledUpdateOrContext(current,
  renderLanes);

      if (!hasScheduledUpdateOrContext) {
        // This will be the props with resolved defaultProps,
        // unlike current.memoizedProps which will be the unresolved ones.
        var prevProps = currentChild.memoizedProps; // Default to shallow comparison

        var compare = Component.compare;
        compare = compare !== null ? compare : shallowEqual;

        if (compare(prevProps, nextProps) && current.ref === workInProgress.ref) {
          return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
        }
      } // React DevTools reads this flag.


      workInProgress.flags |= PerformedWork;
      var newChild = createWorkInProgress(currentChild, nextProps);
      newChild.ref = workInProgress.ref;
      newChild.return = workInProgress;
      workInProgress.child = newChild;
      return newChild;
    }

    function updateSimpleMemoComponent(current, workInProgress, Component, nextProps,
  renderLanes) {
      // TODO: current can be non-null here even if the component
      // hasn't yet mounted. This happens when the inner render suspends.
      // We'll need to figure out if this is fine or can cause issues.
      {
        if (workInProgress.type !== workInProgress.elementType) {
          // Lazy component props can't be validated in createElement
          // because they're only guaranteed to be resolved here.
          var outerMemoType = workInProgress.elementType;

          if (outerMemoType.$$typeof === REACT_LAZY_TYPE) {
            // We warn when you define propTypes on lazy()
            // so let's just skip over it to find memo() outer wrapper.
            // Inner props for memo are validated later.
            var lazyComponent = outerMemoType;
            var payload = lazyComponent._payload;
            var init = lazyComponent._init;

            try {
              outerMemoType = init(payload);
            } catch (x) {
              outerMemoType = null;
            } // Inner propTypes will be validated in the function component path.


            var outerPropTypes = outerMemoType && outerMemoType.propTypes;

            if (outerPropTypes) {
              checkPropTypes(outerPropTypes, nextProps, // Resolved (SimpleMemoComponent
  has no defaultProps)
              'prop', getComponentNameFromType(outerMemoType));
            }
          }
        }
      }
    if (current !== null) {
      var prevProps = current.memoizedProps;

      if (shallowEqual(prevProps, nextProps) && current.ref === workInProgress.ref && (
 // Prevent bailout if the implementation changed due to hot reload.
       workInProgress.type === current.type )) {
        didReceiveUpdate = false; // The props are shallowly equal. Reuse the previous
 props object, like we
        // would during a normal fiber bailout.
        //
        // We don't have strong guarantees that the props object is referentially
        // equal during updates where we can't bail out anyway — like if the props
        // are shallowly equal, but there's a local state or context update in the
        // same batch.
        //
        // However, as a principle, we should aim to make the behavior consistent
        // across different ways of memoizing a component. For example, React.memo
        // has a different internal Fiber layout if you pass a normal function
        // component (SimpleMemoComponent) versus if you pass a different type
        // like forwardRef (MemoComponent). But this is an implementation detail.
        // Wrapping a component in forwardRef (or React.lazy, etc) shouldn't
        // affect whether the props object is reused during a bailout.

        workInProgress.pendingProps = nextProps = prevProps;

        if (!checkScheduledUpdateOrContext(current, renderLanes)) {
          // The pending lanes were cleared at the beginning of beginWork. We're
          // about to bail out, but there might be other lanes that weren't
          // included in the current render. Usually, the priority level of the
          // remaining updates is accumulated during the evaluation of the
          // component (i.e. when processing the update queue). But since since
          // we're bailing out early *without* evaluating the component, we need
          // to account for it here, too. Reset to the value of the current fiber.
          // NOTE: This only applies to SimpleMemoComponent, not MemoComponent,
          // because a MemoComponent fiber does not have hooks or an update queue;
          // rather, it wraps around an inner component, which may or may not
          // contains hooks.
          // TODO: Move the reset at in beginWork out of the common path so that
          // this is no longer necessary.
          workInProgress.lanes = current.lanes;
          return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
        } else if ((current.flags & ForceUpdateForLegacySuspense) !== NoFlags) {
          // This is a special case that only exists for legacy mode.
          // See https://github.com/facebook/react/pull/19216.
          didReceiveUpdate = true;
        }
      }
    }

    return updateFunctionComponent(current, workInProgress, Component, nextProps,
 renderLanes);
  }

  function updateOffscreenComponent(current, workInProgress, renderLanes) {
    var nextProps = workInProgress.pendingProps;
    var nextChildren = nextProps.children;
    var prevState = current !== null ? current.memoizedState : null;

    if (nextProps.mode === 'hidden' || enableLegacyHidden ) {
      // Rendering a hidden tree.
      if ((workInProgress.mode & ConcurrentMode) === NoMode) {
        // In legacy sync mode, don't defer the subtree. Render it now.
        // TODO: Consider how Offscreen should work with transitions in the future
        var nextState = {
          baseLanes: NoLanes,
          cachePool: null,
          transitions: null
        };
        workInProgress.memoizedState = nextState;

        pushRenderLanes(workInProgress, renderLanes);
      } else if (!includesSomeLane(renderLanes, OffscreenLane)) {
        var spawnedCachePool = null; // We're hidden, and we're not rendering at
Offscreen. We will bail out
        // and resume this tree later.

        var nextBaseLanes;

        if (prevState !== null) {
          var prevBaseLanes = prevState.baseLanes;
          nextBaseLanes = mergeLanes(prevBaseLanes, renderLanes);
        } else {
          nextBaseLanes = renderLanes;
        } // Schedule this fiber to re-render at offscreen priority. Then bailout.


        workInProgress.lanes = workInProgress.childLanes = laneToLanes(OffscreenLane);
        var _nextState = {
          baseLanes: nextBaseLanes,
          cachePool: spawnedCachePool,
          transitions: null
        };
        workInProgress.memoizedState = _nextState;
        workInProgress.updateQueue = null;
        // to avoid a push/pop misalignment.


        pushRenderLanes(workInProgress, nextBaseLanes);

        return null;
      } else {
        // This is the second render. The surrounding visible content has already
        // committed. Now we resume rendering the hidden tree.
        // Rendering at offscreen, so we can clear the base lanes.
        var _nextState2 = {
          baseLanes: NoLanes,
          cachePool: null,
          transitions: null
        };
        workInProgress.memoizedState = _nextState2; // Push the lanes that were skipped
when we bailed out.

        var subtreeRenderLanes = prevState !== null ? prevState.baseLanes : renderLanes;

        pushRenderLanes(workInProgress, subtreeRenderLanes);
      }
    } else {
      // Rendering a visible tree.
      var _subtreeRenderLanes;

      if (prevState !== null) {
        // We're going from hidden -> visible.
        _subtreeRenderLanes = mergeLanes(prevState.baseLanes, renderLanes);

        workInProgress.memoizedState = null;
      } else {
        // We weren't previously hidden, and we still aren't, so there's nothing
        // special to do. Need to push to the stack regardless, though, to avoid
        // a push/pop misalignment.
        _subtreeRenderLanes = renderLanes;
      }

      pushRenderLanes(workInProgress, _subtreeRenderLanes);
    }

    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
```

```
  } // Note: These happen to have identical begin phases, for now. We shouldn't hold

  function updateFragment(current, workInProgress, renderLanes) {
    var nextChildren = workInProgress.pendingProps;
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
  }

  function updateMode(current, workInProgress, renderLanes) {
    var nextChildren = workInProgress.pendingProps.children;
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
  }

  function updateProfiler(current, workInProgress, renderLanes) {
    {
      workInProgress.flags |= Update;

      {
        // Reset effect durations for the next eventual effect phase.
        // These are reset during render to allow the DevTools commit hook a chance to
read them,
        var stateNode = workInProgress.stateNode;
        stateNode.effectDuration = 0;
        stateNode.passiveEffectDuration = 0;
      }
    }

    var nextProps = workInProgress.pendingProps;
    var nextChildren = nextProps.children;
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
  }

  function markRef(current, workInProgress) {
    var ref = workInProgress.ref;

    if (current === null && ref !== null || current !== null && current.ref !== ref) {
      // Schedule a Ref effect
      workInProgress.flags |= Ref;

      {
        workInProgress.flags |= RefStatic;
      }
    }
  }

  function updateFunctionComponent(current, workInProgress, Component, nextProps,
renderLanes) {
    {
      if (workInProgress.type !== workInProgress.elementType) {
        // Lazy component props can't be validated in createElement
        // because they're only guaranteed to be resolved here.
        var innerPropTypes = Component.propTypes;

        if (innerPropTypes) {
          checkPropTypes(innerPropTypes, nextProps, // Resolved props
          'prop', getComponentNameFromType(Component));
        }
      }
    }

    var context;

    {
      var unmaskedContext = getUnmaskedContext(workInProgress, Component, true);
      context = getMaskedContext(workInProgress, unmaskedContext);
    }
    var nextChildren;
    var hasId;
    prepareToReadContext(workInProgress, renderLanes);

    {
      markComponentRenderStarted(workInProgress);
    }

    {
      ReactCurrentOwner$1.current = workInProgress;
      setIsRendering(true);
      nextChildren = renderWithHooks(current, workInProgress, Component, nextProps,
context, renderLanes);
      hasId = checkDidRenderIdHook();

      if ( workInProgress.mode & StrictLegacyMode) {
        setIsStrictModeForDevtools(true);

        try {
          nextChildren = renderWithHooks(current, workInProgress, Component, nextProps,
context, renderLanes);
          hasId = checkDidRenderIdHook();
        } finally {
          setIsStrictModeForDevtools(false);
        }
      }

      setIsRendering(false);
    }

    {
      markComponentRenderStopped();
    }

    if (current !== null && !didReceiveUpdate) {
      bailoutHooks(current, workInProgress, renderLanes);
      return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
    }

    if (getIsHydrating() && hasId) {
      pushMaterializedTreeId(workInProgress);
    } // React DevTools reads this flag.


    workInProgress.flags |= PerformedWork;
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
  }

  function updateClassComponent(current, workInProgress, Component, nextProps,
renderLanes) {
    {
      // This is used by DevTools to force a boundary to error.
      switch (shouldError(workInProgress)) {
        case false:
          {
            var _instance = workInProgress.stateNode;
            var ctor = workInProgress.type; // TODO This way of resetting the error
boundary state is a hack.
            // Is there a better way to do this?

            var tempInstance = new ctor(workInProgress.memoizedProps, _instance.context);
            var state = tempInstance.state;

            _instance.updater.enqueueSetState(_instance, state, null);

            break;
          }
        case true:
          {
            workInProgress.flags |= DidCapture;
            workInProgress.flags |= ShouldCapture; // eslint-disable-next-line react-
internal/prod-error-codes

            var error$1 = new Error('Simulated error coming from DevTools');
            var lane = pickArbitraryLane(renderLanes);
            workInProgress.lanes = mergeLanes(workInProgress.lanes, lane); // Schedule
the error boundary to re-render using updated state

            var update = createClassErrorUpdate(workInProgress,
createCapturedValueAtFiber(error$1, workInProgress), lane);
            enqueueCapturedUpdate(workInProgress, update);
            break;
          }
      }

      if (workInProgress.type !== workInProgress.elementType) {
        // Lazy component props can't be validated in createElement
        // because they're only guaranteed to be resolved here.
        var innerPropTypes = Component.propTypes;

        if (innerPropTypes) {
          checkPropTypes(innerPropTypes, nextProps, // Resolved props
          'prop', getComponentNameFromType(Component));
        }
      }
    } // Push context providers early to prevent context stack mismatches.
    // During mounting we don't know the child context yet as the instance doesn't exist.
    // We will invalidate the child context in finishClassComponent() right after
rendering.


    var hasContext;

    if (isContextProvider(Component)) {
      hasContext = true;
      pushContextProvider(workInProgress);
    } else {
      hasContext = false;
    }

    prepareToReadContext(workInProgress, renderLanes);
    var instance = workInProgress.stateNode;
    var shouldUpdate;

    if (instance === null) {
      resetSuspendedCurrentOnMountInLegacyMode(current, workInProgress); // In the
initial pass we might need to construct the instance.

      constructClassInstance(workInProgress, Component, nextProps);
      mountClassInstance(workInProgress, Component, nextProps, renderLanes);
      shouldUpdate = true;
    } else if (current === null) {
      // In a resume, we'll already have an instance we can reuse.
      shouldUpdate = resumeMountClassInstance(workInProgress, Component, nextProps,
renderLanes);
    } else {
      shouldUpdate = updateClassInstance(current, workInProgress, Component, nextProps,
renderLanes);
    }

    var nextUnitOfWork = finishClassComponent(current, workInProgress, Component,
shouldUpdate, hasContext, renderLanes);

    {
      var inst = workInProgress.stateNode;
      if (shouldUpdate && inst.props !== nextProps) {
        if (!didWarnAboutReassigningProps) {
          error('It looks like %s is reassigning its own `this.props` while rendering. '
 + 'This is not supported and can lead to confusing bugs.',
 getComponentNameFromFiber(workInProgress) || 'a component');
        }

        didWarnAboutReassigningProps = true;
      }
    }

    return nextUnitOfWork;
  }

  function finishClassComponent(current, workInProgress, Component, shouldUpdate,
 hasContext, renderLanes) {
    // Refs should update even if shouldComponentUpdate returns false
    markRef(current, workInProgress);
    var didCaptureError = (workInProgress.flags & DidCapture) !== NoFlags;

    if (!shouldUpdate && !didCaptureError) {
      // Context providers should defer to sCU for rendering
      if (hasContext) {
        invalidateContextProvider(workInProgress, Component, false);
      }

      return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
    }

    var instance = workInProgress.stateNode; // Rerender

    ReactCurrentOwner$1.current = workInProgress;
    var nextChildren;

    if (didCaptureError && typeof Component.getDerivedStateFromError !== 'function') {
      // If we captured an error, but getDerivedStateFromError is not defined,
      // unmount all the children. componentDidCatch will schedule an update to
      // re-render a fallback. This is temporary until we migrate everyone to
      // the new API.
      // TODO: Warn in a future release.
      nextChildren = null;

      {
        stopProfilerTimerIfRunning();
      }
    } else {
      {
        markComponentRenderStarted(workInProgress);
      }

      {
        setIsRendering(true);
        nextChildren = instance.render();

        if ( workInProgress.mode & StrictLegacyMode) {
          setIsStrictModeForDevtools(true);

          try {
            instance.render();
          } finally {
            setIsStrictModeForDevtools(false);
          }
        }

        setIsRendering(false);
      }

      {
        markComponentRenderStopped();
      }
    } // React DevTools reads this flag.


    workInProgress.flags |= PerformedWork;

    if (current !== null && didCaptureError) {
      // If we're recovering from an error, reconcile without reusing any of
      // the existing children. Conceptually, the normal children and the children
      // that are shown on error are two different sets, so we shouldn't reuse
      // normal children even if their identities match.
      forceUnmountCurrentAndReconcile(current, workInProgress, nextChildren,
renderLanes);
    } else {
      reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    } // Memoize state using the values we just used to render.
    // TODO: Restructure so we never read values from the instance.


    workInProgress.memoizedState = instance.state; // The context might have changed so
we need to recalculate it.

    if (hasContext) {
      invalidateContextProvider(workInProgress, Component, true);
    }

    return workInProgress.child;
  }

  function pushHostRootContext(workInProgress) {
    var root = workInProgress.stateNode;

    if (root.pendingContext) {
      pushTopLevelContextObject(workInProgress, root.pendingContext, root.pendingContext
!== root.context);
    } else if (root.context) {
      // Should always be set
      pushTopLevelContextObject(workInProgress, root.context, false);
    }

    pushHostContainer(workInProgress, root.containerInfo);
  }

  function updateHostRoot(current, workInProgress, renderLanes) {
    pushHostRootContext(workInProgress);

    if (current === null) {
      throw new Error('Should have a current fiber. This is a bug in React.');
    }

    var nextProps = workInProgress.pendingProps;
    var prevState = workInProgress.memoizedState;
    var prevChildren = prevState.element;
    cloneUpdateQueue(current, workInProgress);
    processUpdateQueue(workInProgress, nextProps, null, renderLanes);
    var nextState = workInProgress.memoizedState;
    var root = workInProgress.stateNode;
    // being called "element".


    var nextChildren = nextState.element;

    if ( prevState.isDehydrated) {
      // This is a hydration root whose shell has not yet hydrated. We should
      // attempt to hydrate.
      // Flip isDehydrated to false to indicate that when this render
      // finishes, the root will no longer be dehydrated.
      var overrideState = {
        element: nextChildren,
        isDehydrated: false,
        cache: nextState.cache,
        pendingSuspenseBoundaries: nextState.pendingSuspenseBoundaries,
        transitions: nextState.transitions
      };
      var updateQueue = workInProgress.updateQueue; // `baseState` can always be the last
state because the root doesn't
      // have reducer functions so it doesn't need rebasing.

      updateQueue.baseState = overrideState;
      workInProgress.memoizedState = overrideState;

      if (workInProgress.flags & ForceClientRender) {
        // Something errored during a previous attempt to hydrate the shell, so we
        // forced a client render.
        var recoverableError = createCapturedValueAtFiber(new Error('There was an error
while hydrating. Because the error happened outside ' + 'of a Suspense boundary, the
entire root will switch to ' + 'client rendering.'), workInProgress);
        return mountHostRootWithoutHydrating(current, workInProgress, nextChildren,
renderLanes, recoverableError);
      } else if (nextChildren !== prevChildren) {
        var _recoverableError = createCapturedValueAtFiber(new Error('This root received
an early update, before anything was able ' + 'hydrate. Switched the entire root to
client rendering.'), workInProgress);

        return mountHostRootWithoutHydrating(current, workInProgress, nextChildren,
renderLanes, _recoverableError);
      } else {
        // The outermost shell has not hydrated yet. Start hydrating.
        enterHydrationState(workInProgress);

        var child = mountChildFibers(workInProgress, null, nextChildren, renderLanes);
        workInProgress.child = child;
        var node = child;

        while (node) {
          // Mark each child as hydrating. This is a fast path to know whether this
          // tree is part of a hydrating tree. This is used to determine if a child
          // node has fully mounted yet, and for scheduling event replaying.
          // Conceptually this is similar to Placement in that a new subtree is
          // inserted into the React tree here. It just happens to not need DOM
          // mutations because it already exists.
          node.flags = node.flags & ~Placement | Hydrating;
          node = node.sibling;
        }
      }
    } else {
      // Root is not dehydrated. Either this is a client-only root, or it
      // already hydrated.
      resetHydrationState();

      if (nextChildren === prevChildren) {
        return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
      }

      reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    }

    return workInProgress.child;
  }

  function mountHostRootWithoutHydrating(current, workInProgress, nextChildren,
renderLanes, recoverableError) {
    // Revert to client rendering.
    resetHydrationState();
    queueHydrationError(recoverableError);
    workInProgress.flags |= ForceClientRender;
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
  }

  function updateHostComponent(current, workInProgress, renderLanes) {
    pushHostContext(workInProgress);

    if (current === null) {
      tryToClaimNextHydratableInstance(workInProgress);
    }

    var type = workInProgress.type;
    var nextProps = workInProgress.pendingProps;
    var prevProps = current !== null ? current.memoizedProps : null;
    var nextChildren = nextProps.children;
    var isDirectTextChild = shouldSetTextContent(type, nextProps);

    if (isDirectTextChild) {
      // We special case a direct text child of a host node. This is a common
      // case. We won't handle it as a reified child. We will instead handle
      // this in the host environment that also has access to this prop. That
      // avoids allocating another HostText fiber and traversing it.
      nextChildren = null;
    } else if (prevProps !== null && shouldSetTextContent(type, prevProps)) {
      // If we're switching from a direct text child to a normal child, or to
      // empty, we need to schedule the text content to be reset.
      workInProgress.flags |= ContentReset;
    }

    markRef(current, workInProgress);
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
  }

  function updateHostText(current, workInProgress) {
    if (current === null) {
      tryToClaimNextHydratableInstance(workInProgress);
    } // Nothing to do here. This is terminal. We'll do the completion step
    // immediately after.


    return null;
  }

  function mountLazyComponent(_current, workInProgress, elementType, renderLanes) {
    resetSuspendedCurrentOnMountInLegacyMode(_current, workInProgress);
    var props = workInProgress.pendingProps;
    var lazyComponent = elementType;
    var payload = lazyComponent._payload;
    var init = lazyComponent._init;
    var Component = init(payload); // Store the unwrapped component in the type.

    workInProgress.type = Component;
    var resolvedTag = workInProgress.tag = resolveLazyComponentTag(Component);
    var resolvedProps = resolveDefaultProps(Component, props);
    var child;

    switch (resolvedTag) {
      case FunctionComponent:
        {
          {
            validateFunctionComponentInDev(workInProgress, Component);
            workInProgress.type = Component = resolveFunctionForHotReloading(Component);
          }

          child = updateFunctionComponent(null, workInProgress, Component, resolvedProps,
renderLanes);
          return child;
        }

      case ClassComponent:
        {
          {
            workInProgress.type = Component = resolveClassForHotReloading(Component);
          }

          child = updateClassComponent(null, workInProgress, Component, resolvedProps,
renderLanes);
          return child;
        }

      case ForwardRef:
        {
          {
            workInProgress.type = Component =
resolveForwardRefForHotReloading(Component);
          }

          child = updateForwardRef(null, workInProgress, Component, resolvedProps,
renderLanes);
          return child;
        }

      case MemoComponent:
        {
          {
            if (workInProgress.type !== workInProgress.elementType) {
              var outerPropTypes = Component.propTypes;

              if (outerPropTypes) {
                checkPropTypes(outerPropTypes, resolvedProps, // Resolved for outer only
                'prop', getComponentNameFromType(Component));
              }
            }
          }

          child = updateMemoComponent(null, workInProgress, Component,
resolveDefaultProps(Component.type, resolvedProps), // The inner type can have defaults
too
          renderLanes);
          return child;
        }
    }

    var hint = '';

    {
      if (Component !== null && typeof Component === 'object' && Component.$$typeof ===
REACT_LAZY_TYPE) {
        hint = ' Did you wrap a component in React.lazy() more than once?';
      }
    } // This message intentionally doesn't mention ForwardRef or MemoComponent
    // because the fact that it's a separate type of work is an
    // implementation detail.


    throw new Error("Element type is invalid. Received a promise that resolves to: " +
Component + ". " + ("Lazy element type must resolve to a class or function." + hint));
  }

  function mountIncompleteClassComponent(_current, workInProgress, Component, nextProps,
renderLanes) {
    resetSuspendedCurrentOnMountInLegacyMode(_current, workInProgress); // Promote the
fiber to a class and try rendering again.

    workInProgress.tag = ClassComponent; // The rest of this function is a fork of
`updateClassComponent`
    // Push context providers early to prevent context stack mismatches.
    // During mounting we don't know the child context yet as the instance doesn't exist.
    // We will invalidate the child context in finishClassComponent() right after
```

```
rendering.

    var hasContext;

    if (isContextProvider(Component)) {
      hasContext = true;
      pushContextProvider(workInProgress);
    } else {
      hasContext = false;
    }

    prepareToReadContext(workInProgress, renderLanes);
    constructClassInstance(workInProgress, Component, nextProps);
    mountClassInstance(workInProgress, Component, nextProps, renderLanes);
    return finishClassComponent(null, workInProgress, Component, true, hasContext,
renderLanes);
  }

  function mountIndeterminateComponent(_current, workInProgress, Component, renderLanes)
{
    resetSuspendedCurrentOnMountInLegacyMode(_current, workInProgress);
    var props = workInProgress.pendingProps;
    var context;

    {
      var unmaskedContext = getUnmaskedContext(workInProgress, Component, false);
      context = getMaskedContext(workInProgress, unmaskedContext);
    }

    prepareToReadContext(workInProgress, renderLanes);
    var value;
    var hasId;

    {
      markComponentRenderStarted(workInProgress);
    }

    {
      if (Component.prototype && typeof Component.prototype.render === 'function') {
        var componentName = getComponentNameFromType(Component) || 'Unknown';

        if (!didWarnAboutBadClass[componentName]) {
          error("The <%s /> component appears to have a render method, but doesn't extend
React.Component. " + 'This is likely to cause errors. Change %s to extend React.Component
instead.', componentName, componentName);

          didWarnAboutBadClass[componentName] = true;
        }
      }

      if (workInProgress.mode & StrictLegacyMode) {
        ReactStrictModeWarnings.recordLegacyContextWarning(workInProgress, null);
      }

      setIsRendering(true);
      ReactCurrentOwner$1.current = workInProgress;
      value = renderWithHooks(null, workInProgress, Component, props, context,
renderLanes);
      hasId = checkDidRenderIdHook();
      setIsRendering(false);
    }

    {
      markComponentRenderStopped();
    } // React DevTools reads this flag.


    workInProgress.flags |= PerformedWork;
    {
      // Support for module components is deprecated and is removed behind a flag.
      // Whether or not it would crash later, we want to show a good message in DEV
first.
      if (typeof value === 'object' && value !== null && typeof value.render ===
'function' && value.$$typeof === undefined) {
        var _componentName = getComponentNameFromType(Component) || 'Unknown';

        if (!didWarnAboutModulePatternComponent[_componentName]) {
          error('The <%s /> component appears to be a function component that returns a
class instance. ' + 'Change %s to a class that extends React.Component instead. ' + "If
you can't use a class try assigning the prototype on the function as a workaround. " +
"`%s.prototype = React.Component.prototype`. Don't use an arrow function since it " +
'cannot be called with `new` by React.', _componentName, _componentName, _componentName);

          didWarnAboutModulePatternComponent[_componentName] = true;
        }
      }
    }

    if ( // Run these checks in production only if the flag is off.
    // Eventually we'll delete this branch altogether.
     typeof value === 'object' && value !== null && typeof value.render === 'function' &&
value.$$typeof === undefined) {
      {
        var _componentName2 = getComponentNameFromType(Component) || 'Unknown';

        if (!didWarnAboutModulePatternComponent[_componentName2]) {
          error('The <%s /> component appears to be a function component that returns a
class instance. ' + 'Change %s to a class that extends React.Component instead. ' + "If
you can't use a class try assigning the prototype on the function as a workaround. " +
"`%s.prototype = React.Component.prototype`. Don't use an arrow function since it " +
'cannot be called with `new` by React.', _componentName2, _componentName2,
_componentName2);

          didWarnAboutModulePatternComponent[_componentName2] = true;
        }
      } // Proceed under the assumption that this is a class instance


      workInProgress.tag = ClassComponent; // Throw out any hooks that were used.

      workInProgress.memoizedState = null;
      workInProgress.updateQueue = null; // Push context providers early to prevent
context stack mismatches.
      // During mounting we don't know the child context yet as the instance doesn't
exist.
      // We will invalidate the child context in finishClassComponent() right after
rendering.

      var hasContext = false;

      if (isContextProvider(Component)) {
        hasContext = true;
        pushContextProvider(workInProgress);
      } else {
        hasContext = false;
      }

      workInProgress.memoizedState = value.state !== null && value.state !== undefined ?
value.state : null;
      initializeUpdateQueue(workInProgress);
      adoptClassInstance(workInProgress, value);
      mountClassInstance(workInProgress, Component, props, renderLanes);
      return finishClassComponent(null, workInProgress, Component, true, hasContext,
renderLanes);
    } else {
      // Proceed under the assumption that this is a function component
      workInProgress.tag = FunctionComponent;
      {

        if ( workInProgress.mode & StrictLegacyMode) {
          setIsStrictModeForDevtools(true);

          try {
            value = renderWithHooks(null, workInProgress, Component, props, context,
renderLanes);
            hasId = checkDidRenderIdHook();
          } finally {
            setIsStrictModeForDevtools(false);
          }
        }
      }

      if (getIsHydrating() && hasId) {
        pushMaterializedTreeId(workInProgress);
      }

      reconcileChildren(null, workInProgress, value, renderLanes);

      {
        validateFunctionComponentInDev(workInProgress, Component);
      }

      return workInProgress.child;
    }
  }

  function validateFunctionComponentInDev(workInProgress, Component) {
    {
      if (Component) {
        if (Component.childContextTypes) {
          error('%s(...): childContextTypes cannot be defined on a function component.',
Component.displayName || Component.name || 'Component');
        }
      }

      if (workInProgress.ref !== null) {
        var info = '';
        var ownerName = getCurrentFiberOwnerNameInDevOrNull();

        if (ownerName) {
          info += '\n\nCheck the render method of `' + ownerName + '`.';
        }

        var warningKey = ownerName || '';
        var debugSource = workInProgress._debugSource;

        if (debugSource) {
          warningKey = debugSource.fileName + ':' + debugSource.lineNumber;
        }

        if (!didWarnAboutFunctionRefs[warningKey]) {
          didWarnAboutFunctionRefs[warningKey] = true;

          error('Function components cannot be given refs. ' + 'Attempts to access this
ref will fail. ' + 'Did you mean to use React.forwardRef()?%s', info);
        }
      }

      if ( Component.defaultProps !== undefined) {
        var componentName = getComponentNameFromType(Component) || 'Unknown';

        if (!didWarnAboutDefaultPropsOnFunctionComponent[componentName]) {
          error('%s: Support for defaultProps will be removed from function components '
+ 'in a future major release. Use JavaScript default parameters instead.',
componentName);
```

```
          didWarnAboutDefaultPropsOnFunctionComponent[componentName] = true;
        }
      }

      if (typeof Component.getDerivedStateFromProps === 'function') {
        var _componentName3 = getComponentNameFromType(Component) || 'Unknown';

        if (!didWarnAboutGetDerivedStateOnFunctionComponent[_componentName3]) {
          error('%s: Function components do not support getDerivedStateFromProps.',
_componentName3);

          didWarnAboutGetDerivedStateOnFunctionComponent[_componentName3] = true;
        }
      }

      if (typeof Component.contextType === 'object' && Component.contextType !== null) {
        var _componentName4 = getComponentNameFromType(Component) || 'Unknown';

        if (!didWarnAboutContextTypeOnFunctionComponent[_componentName4]) {
          error('%s: Function components do not support contextType.', _componentName4);

          didWarnAboutContextTypeOnFunctionComponent[_componentName4] = true;
        }
      }
    }
  }

  var SUSPENDED_MARKER = {
    dehydrated: null,
    treeContext: null,
    retryLane: NoLane
  };

  function mountSuspenseOffscreenState(renderLanes) {
    return {
      baseLanes: renderLanes,
      cachePool: getSuspendedCache(),
      transitions: null
    };
  }

  function updateSuspenseOffscreenState(prevOffscreenState, renderLanes) {
    var cachePool = null;

    return {
      baseLanes: mergeLanes(prevOffscreenState.baseLanes, renderLanes),
      cachePool: cachePool,
      transitions: prevOffscreenState.transitions
    };
  } // TODO: Probably should inline this back


  function shouldRemainOnFallback(suspenseContext, current, workInProgress, renderLanes)
{
    // If we're already showing a fallback, there are cases where we need to
    // remain on that fallback regardless of whether the content has resolved.
    // For example, SuspenseList coordinates when nested content appears.
    if (current !== null) {
      var suspenseState = current.memoizedState;

      if (suspenseState === null) {
        // Currently showing content. Don't hide it, even if ForceSuspenseFallback
        // is true. More precise name might be "ForceRemainSuspenseFallback".
        // Note: This is a factoring smell. Can't remain on a fallback if there's
        // no fallback to remain on.
        return false;
      }
    } // Not currently showing content. Consult the Suspense context.
```

```
      return hasSuspenseContext(suspenseContext, ForceSuspenseFallback);
    }

    function getRemainingWorkInPrimaryTree(current, renderLanes) {
      // TODO: Should not remove render lanes that were pinged during this render
      return removeLanes(current.childLanes, renderLanes);
    }

    function updateSuspenseComponent(current, workInProgress, renderLanes) {
      var nextProps = workInProgress.pendingProps; // This is used by DevTools to force a
  boundary to suspend.

      {
        if (shouldSuspend(workInProgress)) {
          workInProgress.flags |= DidCapture;
        }
      }

      var suspenseContext = suspenseStackCursor.current;
      var showFallback = false;
      var didSuspend = (workInProgress.flags & DidCapture) !== NoFlags;

      if (didSuspend || shouldRemainOnFallback(suspenseContext, current)) {
        // Something in this boundary's subtree already suspended. Switch to
        // rendering the fallback children.
        showFallback = true;
        workInProgress.flags &= ~DidCapture;
      } else {
        // Attempting the main content
        if (current === null || current.memoizedState !== null) {
          // This is a new mount or this boundary is already showing a fallback state.
          // Mark this subtree context as having at least one invisible parent that could
          // handle the fallback state.
          // Avoided boundaries are not considered since they cannot handle preferred
  fallback states.
          {
            suspenseContext = addSubtreeSuspenseContext(suspenseContext,
  InvisibleParentSuspenseContext);
          }
        }
      }

      suspenseContext = setDefaultShallowSuspenseContext(suspenseContext);
      pushSuspenseContext(workInProgress, suspenseContext); // OK, the next part is
  confusing. We're about to reconcile the Suspense
      // boundary's children. This involves some custom reconciliation logic. Two
      // main reasons this is so complicated.
      //
      // First, Legacy Mode has different semantics for backwards compatibility. The
      // primary tree will commit in an inconsistent state, so when we do the
      // second pass to render the fallback, we do some exceedingly, uh, clever
      // hacks to make that not totally break. Like transferring effects and
      // deletions from hidden tree. In Concurrent Mode, it's much simpler,
      // because we bailout on the primary tree completely and leave it in its old
      // state, no effects. Same as what we do for Offscreen (except that
      // Offscreen doesn't have the first render pass).
      //
      // Second is hydration. During hydration, the Suspense fiber has a slightly
      // different layout, where the child points to a dehydrated fragment, which
      // contains the DOM rendered by the server.
      //
      // Third, even if you set all that aside, Suspense is like error boundaries in
      // that we first we try to render one tree, and if that fails, we render again
      // and switch to a different tree. Like a try/catch block. So we have to track
      // which branch we're currently rendering. Ideally we would model this using
      // a stack.
      if (current === null) {
        // Initial mount
        // Special path for hydration
        // If we're currently hydrating, try to hydrate this boundary.
        tryToClaimNextHydratableInstance(workInProgress); // This could've been a
  dehydrated suspense component.

        var suspenseState = workInProgress.memoizedState;

        if (suspenseState !== null) {
          var dehydrated = suspenseState.dehydrated;

          if (dehydrated !== null) {
            return mountDehydratedSuspenseComponent(workInProgress, dehydrated);
          }
        }

        var nextPrimaryChildren = nextProps.children;
        var nextFallbackChildren = nextProps.fallback;

        if (showFallback) {
          var fallbackFragment = mountSuspenseFallbackChildren(workInProgress,
  nextPrimaryChildren, nextFallbackChildren, renderLanes);
          var primaryChildFragment = workInProgress.child;
          primaryChildFragment.memoizedState = mountSuspenseOffscreenState(renderLanes);
          workInProgress.memoizedState = SUSPENDED_MARKER;

          return fallbackFragment;
        } else {
          return mountSuspensePrimaryChildren(workInProgress, nextPrimaryChildren);
        }
      } else {
        // This is an update.
        // Special path for hydration
        var prevState = current.memoizedState;

        if (prevState !== null) {
          var _dehydrated = prevState.dehydrated;

          if (_dehydrated !== null) {
            return updateDehydratedSuspenseComponent(current, workInProgress, didSuspend,
  nextProps, _dehydrated, prevState, renderLanes);
          }
        }

        if (showFallback) {
          var _nextFallbackChildren = nextProps.fallback;
          var _nextPrimaryChildren = nextProps.children;
          var fallbackChildFragment = updateSuspenseFallbackChildren(current,
  workInProgress, _nextPrimaryChildren, _nextFallbackChildren, renderLanes);
          var _primaryChildFragment2 = workInProgress.child;
          var prevOffscreenState = current.child.memoizedState;
          _primaryChildFragment2.memoizedState = prevOffscreenState === null ?
  mountSuspenseOffscreenState(renderLanes) :
  updateSuspenseOffscreenState(prevOffscreenState, renderLanes);

          _primaryChildFragment2.childLanes = getRemainingWorkInPrimaryTree(current,
  renderLanes);
          workInProgress.memoizedState = SUSPENDED_MARKER;
          return fallbackChildFragment;
        } else {
          var _nextPrimaryChildren2 = nextProps.children;

          var _primaryChildFragment3 = updateSuspensePrimaryChildren(current,
  workInProgress, _nextPrimaryChildren2, renderLanes);

          workInProgress.memoizedState = null;
          return _primaryChildFragment3;
        }
      }
    }

    function mountSuspensePrimaryChildren(workInProgress, primaryChildren, renderLanes) {
      var mode = workInProgress.mode;
      var primaryChildProps = {
        mode: 'visible',
        children: primaryChildren
      };
      var primaryChildFragment = mountWorkInProgressOffscreenFiber(primaryChildProps,
  mode);
      primaryChildFragment.return = workInProgress;
      workInProgress.child = primaryChildFragment;
      return primaryChildFragment;
    }

    function mountSuspenseFallbackChildren(workInProgress, primaryChildren,
  fallbackChildren, renderLanes) {
      var mode = workInProgress.mode;
      var progressedPrimaryFragment = workInProgress.child;
      var primaryChildProps = {
        mode: 'hidden',
        children: primaryChildren
      };
      var primaryChildFragment;
      var fallbackChildFragment;

      if ((mode & ConcurrentMode) === NoMode && progressedPrimaryFragment !== null) {
        // In legacy mode, we commit the primary tree as if it successfully
        // completed, even though it's in an inconsistent state.
        primaryChildFragment = progressedPrimaryFragment;
        primaryChildFragment.childLanes = NoLanes;
        primaryChildFragment.pendingProps = primaryChildProps;

        if ( workInProgress.mode & ProfileMode) {
          // Reset the durations from the first pass so they aren't included in the
          // final amounts. This seems counterintuitive, since we're intentionally
          // not measuring part of the render phase, but this makes it match what we
          // do in Concurrent Mode.
          primaryChildFragment.actualDuration = 0;
          primaryChildFragment.actualStartTime = -1;
          primaryChildFragment.selfBaseDuration = 0;
          primaryChildFragment.treeBaseDuration = 0;
        }

        fallbackChildFragment = createFiberFromFragment(fallbackChildren, mode,
  renderLanes, null);
      } else {
        primaryChildFragment = mountWorkInProgressOffscreenFiber(primaryChildProps, mode);
        fallbackChildFragment = createFiberFromFragment(fallbackChildren, mode,
  renderLanes, null);
      }

      primaryChildFragment.return = workInProgress;
      fallbackChildFragment.return = workInProgress;
      primaryChildFragment.sibling = fallbackChildFragment;
      workInProgress.child = primaryChildFragment;
      return fallbackChildFragment;
    }

    function mountWorkInProgressOffscreenFiber(offscreenProps, mode, renderLanes) {
      // The props argument to `createFiberFromOffscreen` is `any` typed, so we use
      // this wrapper function to constrain it.
      return createFiberFromOffscreen(offscreenProps, mode, NoLanes, null);
    }

    function updateWorkInProgressOffscreenFiber(current, offscreenProps) {
      // The props argument to `createWorkInProgress` is `any` typed, so we use this
      // wrapper function to constrain it.
      return createWorkInProgress(current, offscreenProps);
    }

    function updateSuspensePrimaryChildren(current, workInProgress, primaryChildren,
  renderLanes) {
      var currentPrimaryChildFragment = current.child;
      var currentFallbackChildFragment = currentPrimaryChildFragment.sibling;
      var primaryChildFragment =
  updateWorkInProgressOffscreenFiber(currentPrimaryChildFragment, {
        mode: 'visible',
        children: primaryChildren
      });

      if ((workInProgress.mode & ConcurrentMode) === NoMode) {
        primaryChildFragment.lanes = renderLanes;
      }

      primaryChildFragment.return = workInProgress;
      primaryChildFragment.sibling = null;

      if (currentFallbackChildFragment !== null) {
        // Delete the fallback child fragment
        var deletions = workInProgress.deletions;

        if (deletions === null) {
          workInProgress.deletions = [currentFallbackChildFragment];
          workInProgress.flags |= ChildDeletion;
        } else {
          deletions.push(currentFallbackChildFragment);
        }
      }

      workInProgress.child = primaryChildFragment;
      return primaryChildFragment;
    }

    function updateSuspenseFallbackChildren(current, workInProgress, primaryChildren,
  fallbackChildren, renderLanes) {
      var mode = workInProgress.mode;
      var currentPrimaryChildFragment = current.child;
      var currentFallbackChildFragment = currentPrimaryChildFragment.sibling;
      var primaryChildProps = {
        mode: 'hidden',
        children: primaryChildren
      };
      var primaryChildFragment;

      if ( // In legacy mode, we commit the primary tree as if it successfully
      // completed, even though it's in an inconsistent state.
      (mode & ConcurrentMode) === NoMode && // Make sure we're on the second pass, i.e. the
  primary child fragment was
      // already cloned. In legacy mode, the only case where this isn't true is
      // when DevTools forces us to display a fallback; we skip the first render
      // pass entirely and go straight to rendering the fallback. (In Concurrent
      // Mode, SuspenseList can also trigger this scenario, but this is a legacy-
      // only codepath.)
      workInProgress.child !== currentPrimaryChildFragment) {
        var progressedPrimaryFragment = workInProgress.child;
        primaryChildFragment = progressedPrimaryFragment;
        primaryChildFragment.childLanes = NoLanes;
        primaryChildFragment.pendingProps = primaryChildProps;

        if ( workInProgress.mode & ProfileMode) {
          // Reset the durations from the first pass so they aren't included in the
          // final amounts. This seems counterintuitive, since we're intentionally
          // not measuring part of the render phase, but this makes it match what we
          // do in Concurrent Mode.
          primaryChildFragment.actualDuration = 0;
          primaryChildFragment.actualStartTime = -1;
          primaryChildFragment.selfBaseDuration =
  currentPrimaryChildFragment.selfBaseDuration;
          primaryChildFragment.treeBaseDuration =
  currentPrimaryChildFragment.treeBaseDuration;
        } // The fallback fiber was added as a deletion during the first pass.
        // However, since we're going to remain on the fallback, we no longer want
        // to delete it.


        workInProgress.deletions = null;
      } else {
        primaryChildFragment =
  updateWorkInProgressOffscreenFiber(currentPrimaryChildFragment, primaryChildProps); //
  Since we're reusing a current tree, we need to reuse the flags, too.
        // (We don't do this in legacy mode, because in legacy mode we don't re-use
        // the current tree; see previous branch.)

        primaryChildFragment.subtreeFlags = currentPrimaryChildFragment.subtreeFlags &
  StaticMask;
      }

      var fallbackChildFragment;

      if (currentFallbackChildFragment !== null) {
        fallbackChildFragment = createWorkInProgress(currentFallbackChildFragment,
  fallbackChildren);
      } else {
        fallbackChildFragment = createFiberFromFragment(fallbackChildren, mode,
  renderLanes, null); // Needs a placement effect because the parent (the Suspense
  boundary) already
        // mounted but this is a new fiber.

        fallbackChildFragment.flags |= Placement;
      }

      fallbackChildFragment.return = workInProgress;
      primaryChildFragment.return = workInProgress;
      primaryChildFragment.sibling = fallbackChildFragment;
      workInProgress.child = primaryChildFragment;
      return fallbackChildFragment;
    }

    function retrySuspenseComponentWithoutHydrating(current, workInProgress, renderLanes,
  recoverableError) {
      // Falling back to client rendering. Because this has performance
      // implications, it's considered a recoverable error, even though the user
      // likely won't observe anything wrong with the UI.
      //
      // The error is passed in as an argument to enforce that every caller provide
      // a custom message, or explicitly opt out (currently the only path that opts
      // out is legacy mode; every concurrent path provides an error).
      if (recoverableError !== null) {
        queueHydrationError(recoverableError);
      } // This will add the old fiber to the deletion list


      reconcileChildFibers(workInProgress, current.child, null, renderLanes); // We're now
  not suspended nor dehydrated.

      var nextProps = workInProgress.pendingProps;
      var primaryChildren = nextProps.children;
      var primaryChildFragment = mountSuspensePrimaryChildren(workInProgress,
  primaryChildren); // Needs a placement effect because the parent (the Suspense boundary)
  already
      // mounted but this is a new fiber.

      primaryChildFragment.flags |= Placement;
      workInProgress.memoizedState = null;
      return primaryChildFragment;
  }

  function mountSuspenseFallbackAfterRetryWithoutHydrating(current, workInProgress,
primaryChildren, fallbackChildren, renderLanes) {
    var fiberMode = workInProgress.mode;
    var primaryChildProps = {
      mode: 'visible',
      children: primaryChildren
    };
    var primaryChildFragment = mountWorkInProgressOffscreenFiber(primaryChildProps,
fiberMode);
    var fallbackChildFragment = createFiberFromFragment(fallbackChildren, fiberMode,
renderLanes, null); // Needs a placement effect because the parent (the Suspense
    // boundary) already mounted but this is a new fiber.

    fallbackChildFragment.flags |= Placement;
    primaryChildFragment.return = workInProgress;
    fallbackChildFragment.return = workInProgress;
    primaryChildFragment.sibling = fallbackChildFragment;
    workInProgress.child = primaryChildFragment;

    if ((workInProgress.mode & ConcurrentMode) !== NoMode) {
      // We will have dropped the effect list which contains the
      // deletion. We need to reconcile to delete the current child.
      reconcileChildFibers(workInProgress, current.child, null, renderLanes);
    }

    return fallbackChildFragment;
  }

  function mountDehydratedSuspenseComponent(workInProgress, suspenseInstance,
renderLanes) {
    // During the first pass, we'll bail out and not drill into the children.
    // Instead, we'll leave the content in place and try to hydrate it later.
    if ((workInProgress.mode & ConcurrentMode) === NoMode) {
      {
        error('Cannot hydrate Suspense in legacy mode. Switch from ' +
'ReactDOM.hydrate(element, container) to ' + 'ReactDOMClient.hydrateRoot(container, <App
/>)' + '.render(element) or remove the Suspense components from ' + 'the server rendered
components.');
      }

      workInProgress.lanes = laneToLanes(SyncLane);
    } else if (isSuspenseInstanceFallback(suspenseInstance)) {
      // This is a client-only boundary. Since we won't get any content from the server
      // for this, we need to schedule that at a higher priority based on when it would
      // have timed out. In theory we could render it in this pass but it would have the
      // wrong priority associated with it and will prevent hydration of parent path.
      // Instead, we'll leave work left on it to render it in a separate commit.
      // TODO This time should be the time at which the server rendered response that is
      // a parent to this boundary was displayed. However, since we currently don't have
      // a protocol to transfer that time, we'll just estimate it by using the current
      // time. This will mean that Suspense timeouts are slightly shifted to later than
      // they should be.
      // Schedule a normal pri update to render this content.
      workInProgress.lanes = laneToLanes(DefaultHydrationLane);
    } else {
      // We'll continue hydrating the rest at offscreen priority since we'll already
      // be showing the right content coming from the server, it is no rush.
      workInProgress.lanes = laneToLanes(OffscreenLane);
    }

    return null;
  }

  function updateDehydratedSuspenseComponent(current, workInProgress, didSuspend,
nextProps, suspenseInstance, suspenseState, renderLanes) {
    if (!didSuspend) {
      // This is the first render pass. Attempt to hydrate.
      // We should never be hydrating at this point because it is the first pass,
      // but after we've already committed once.
      warnIfHydrating();

      if ((workInProgress.mode & ConcurrentMode) === NoMode) {
        return retrySuspenseComponentWithoutHydrating(current, workInProgress,
renderLanes, // TODO: When we delete legacy mode, we should make this error argument
        // required — every concurrent mode path that causes hydration to
        // de-opt to client rendering should have an error message.
        null);
      }

      if (isSuspenseInstanceFallback(suspenseInstance)) {
        // This boundary is in a permanent fallback state. In this case, we'll never
        // get an update and we'll never be able to hydrate the final content. Let's just
try the
        // client side render instead.
        var digest, message, stack;

        {
          var _getSuspenseInstanceF =
getSuspenseInstanceFallbackErrorDetails(suspenseInstance);

          digest = _getSuspenseInstanceF.digest;
          message = _getSuspenseInstanceF.message;
          stack = _getSuspenseInstanceF.stack;
        }

        var error;

        if (message) {
          // eslint-disable-next-line react-internal/prod-error-codes
          error = new Error(message);
        } else {
          error = new Error('The server could not finish this Suspense boundary, likely '
+ 'due to an error during server rendering. Switched to ' + 'client rendering.');
        }

        var capturedValue = createCapturedValue(error, digest, stack);
        return retrySuspenseComponentWithoutHydrating(current, workInProgress,
renderLanes, capturedValue);
      }
      // any context has changed, we need to treat is as if the input might have changed.


      var hasContextChanged = includesSomeLane(renderLanes, current.childLanes);

      if (didReceiveUpdate || hasContextChanged) {
        // This boundary has changed since the first render. This means that we are now
unable to
        // hydrate it. We might still be able to hydrate it using a higher priority lane.
        var root = getWorkInProgressRoot();

        if (root !== null) {
          var attemptHydrationAtLane = getBumpedLaneForHydration(root, renderLanes);

          if (attemptHydrationAtLane !== NoLane && attemptHydrationAtLane !==
suspenseState.retryLane) {
            // Intentionally mutating since this render will get interrupted. This
            // is one of the very rare times where we mutate the current tree
            // during the render phase.
            suspenseState.retryLane = attemptHydrationAtLane; // TODO: Ideally this would
inherit the event time of the current render

            var eventTime = NoTimestamp;
            enqueueConcurrentRenderForLane(current, attemptHydrationAtLane);
            scheduleUpdateOnFiber(root, current, attemptHydrationAtLane, eventTime);
          }
        } // If we have scheduled higher pri work above, this will probably just abort
```

```
the render
        // since we now have higher priority work, but in case it doesn't, we need to
prepare to
        // render something, if we time out. Even if that requires us to delete
everything and
        // skip hydration.
        // Delay having to do this as long as the suspense timeout allows us.


        renderDidSuspendDelayIfPossible();

        var _capturedValue = createCapturedValue(new Error('This Suspense boundary
received an update before it finished ' + 'hydrating. This caused the boundary to switch
to client rendering. ' + 'The usual way to fix this is to wrap the original update ' +
'in startTransition.'));

        return retrySuspenseComponentWithoutHydrating(current, workInProgress,
renderLanes, _capturedValue);
      } else if (isSuspenseInstancePending(suspenseInstance)) {
        // This component is still pending more data from the server, so we can't hydrate
its
        // content. We treat it as if this component suspended itself. It might seem as
if
        // we could just try to render it client-side instead. However, this will perform
a
        // lot of unnecessary work and is unlikely to complete since it often will
suspend
        // on missing data anyway. Additionally, the server might be able to render more
        // than we can on the client yet. In that case we'd end up with more fallback
states
        // on the client than if we just leave it alone. If the server times out or
errors
        // these should update this boundary to the permanent Fallback state instead.
        // Mark it as having captured (i.e. suspended).
        workInProgress.flags |= DidCapture; // Leave the child in place. I.e. the
dehydrated fragment.

        workInProgress.child = current.child; // Register a callback to retry this
boundary once the server has sent the result.

        var retry = retryDehydratedSuspenseBoundary.bind(null, current);
        registerSuspenseInstanceRetry(suspenseInstance, retry);
        return null;
      } else {
        // This is the first attempt.
        reenterHydrationStateFromDehydratedSuspenseInstance(workInProgress,
suspenseInstance, suspenseState.treeContext);
        var primaryChildren = nextProps.children;
        var primaryChildFragment = mountSuspensePrimaryChildren(workInProgress,
primaryChildren); // Mark the children as hydrating. This is a fast path to know whether
this
        // tree is part of a hydrating tree. This is used to determine if a child
        // node has fully mounted yet, and for scheduling event replaying.
        // Conceptually this is similar to Placement in that a new subtree is
        // inserted into the React tree here. It just happens to not need DOM
        // mutations because it already exists.

        primaryChildFragment.flags |= Hydrating;
        return primaryChildFragment;
      }
    } else {
      // This is the second render pass. We already attempted to hydrated, but
      // something either suspended or errored.
      if (workInProgress.flags & ForceClientRender) {
        // Something errored during hydration. Try again without hydrating.
        workInProgress.flags &= ~ForceClientRender;

        var _capturedValue2 = createCapturedValue(new Error('There was an error while
hydrating this Suspense boundary. ' + 'Switched to client rendering.'));
```

```
        return retrySuspenseComponentWithoutHydrating(current, workInProgress,
renderLanes, _capturedValue2);
      } else if (workInProgress.memoizedState !== null) {
        // Something suspended and we should still be in dehydrated mode.
        // Leave the existing child in place.
        workInProgress.child = current.child; // The dehydrated completion pass expects
this flag to be there
        // but the normal suspense pass doesn't.

        workInProgress.flags |= DidCapture;
        return null;
      } else {
        // Suspended but we should no longer be in dehydrated mode.
        // Therefore we now have to render the fallback.
        var nextPrimaryChildren = nextProps.children;
        var nextFallbackChildren = nextProps.fallback;
        var fallbackChildFragment =
mountSuspenseFallbackAfterRetryWithoutHydrating(current, workInProgress,
nextPrimaryChildren, nextFallbackChildren, renderLanes);
        var _primaryChildFragment4 = workInProgress.child;
        _primaryChildFragment4.memoizedState = mountSuspenseOffscreenState(renderLanes);
        workInProgress.memoizedState = SUSPENDED_MARKER;
        return fallbackChildFragment;
      }
    }
  }

  function scheduleSuspenseWorkOnFiber(fiber, renderLanes, propagationRoot) {
    fiber.lanes = mergeLanes(fiber.lanes, renderLanes);
    var alternate = fiber.alternate;

    if (alternate !== null) {
      alternate.lanes = mergeLanes(alternate.lanes, renderLanes);
    }

    scheduleContextWorkOnParentPath(fiber.return, renderLanes, propagationRoot);
  }

  function propagateSuspenseContextChange(workInProgress, firstChild, renderLanes) {
    // Mark any Suspense boundaries with fallbacks as having work to do.
    // If they were previously forced into fallbacks, they may now be able
    // to unblock.
    var node = firstChild;

    while (node !== null) {
      if (node.tag === SuspenseComponent) {
        var state = node.memoizedState;

        if (state !== null) {
          scheduleSuspenseWorkOnFiber(node, renderLanes, workInProgress);
        }
      } else if (node.tag === SuspenseListComponent) {
        // If the tail is hidden there might not be an Suspense boundaries
        // to schedule work on. In this case we have to schedule it on the
        // list itself.
        // We don't have to traverse to the children of the list since
        // the list will propagate the change when it rerenders.
        scheduleSuspenseWorkOnFiber(node, renderLanes, workInProgress);
      } else if (node.child !== null) {
        node.child.return = node;
        node = node.child;
        continue;
      }

      if (node === workInProgress) {
        return;
      }
```

```
        while (node.sibling === null) {
          if (node.return === null || node.return === workInProgress) {
            return;
          }

          node = node.return;
        }

        node.sibling.return = node.return;
        node = node.sibling;
      }
    }

    function findLastContentRow(firstChild) {
      // This is going to find the last row among these children that is already
      // showing content on the screen, as opposed to being in fallback state or
      // new. If a row has multiple Suspense boundaries, any of them being in the
      // fallback state, counts as the whole row being in a fallback state.
      // Note that the "rows" will be workInProgress, but any nested children
      // will still be current since we haven't rendered them yet. The mounted
      // order may not be the same as the new order. We use the new order.
      var row = firstChild;
      var lastContentRow = null;

      while (row !== null) {
        var currentRow = row.alternate; // New rows can't be content rows.

        if (currentRow !== null && findFirstSuspended(currentRow) === null) {
          lastContentRow = row;
        }

        row = row.sibling;
      }

      return lastContentRow;
    }

    function validateRevealOrder(revealOrder) {
      {
        if (revealOrder !== undefined && revealOrder !== 'forwards' && revealOrder !==
'backwards' && revealOrder !== 'together' && !didWarnAboutRevealOrder[revealOrder]) {
          didWarnAboutRevealOrder[revealOrder] = true;

          if (typeof revealOrder === 'string') {
            switch (revealOrder.toLowerCase()) {
              case 'together':
              case 'forwards':
              case 'backwards':
                {
                  error('"%s" is not a valid value for revealOrder on <SuspenseList />. ' +
'Use lowercase "%s" instead.', revealOrder, revealOrder.toLowerCase());

                  break;
                }

              case 'forward':
              case 'backward':
                {
                  error('"%s" is not a valid value for revealOrder on <SuspenseList />. ' +
'React uses the -s suffix in the spelling. Use "%ss" instead.', revealOrder,
revealOrder.toLowerCase());

                  break;
                }

              default:
                error('"%s" is not a supported revealOrder on <SuspenseList />. ' + 'Did
you mean "together", "forwards" or "backwards"?', revealOrder);
```

```
            break;
          }
        } else {
          error('%s is not a supported value for revealOrder on <SuspenseList />. ' +
'Did you mean "together", "forwards" or "backwards"?', revealOrder);
        }
      }
    }
  }

  function validateTailOptions(tailMode, revealOrder) {
    {
      if (tailMode !== undefined && !didWarnAboutTailOptions[tailMode]) {
        if (tailMode !== 'collapsed' && tailMode !== 'hidden') {
          didWarnAboutTailOptions[tailMode] = true;

          error('"%s" is not a supported value for tail on <SuspenseList />. ' + 'Did you
mean "collapsed" or "hidden"?', tailMode);
        } else if (revealOrder !== 'forwards' && revealOrder !== 'backwards') {
          didWarnAboutTailOptions[tailMode] = true;

          error('<SuspenseList tail="%s" /> is only valid if revealOrder is ' +
'"forwards" or "backwards". ' + 'Did you mean to specify revealOrder="forwards"?',
tailMode);
        }
      }
    }
  }

  function validateSuspenseListNestedChild(childSlot, index) {
    {
      var isAnArray = isArray(childSlot);
      var isIterable = !isAnArray && typeof getIteratorFn(childSlot) === 'function';

      if (isAnArray || isIterable) {
        var type = isAnArray ? 'array' : 'iterable';

        error('A nested %s was passed to row #%s in <SuspenseList />. Wrap it in ' + 'an
additional SuspenseList to configure its revealOrder: ' + '<SuspenseList revealOrder=...>
... ' + '<SuspenseList revealOrder=...>{%s}</SuspenseList> ... ' + '</SuspenseList>',
type, index, type);

        return false;
      }
    }

    return true;
  }

  function validateSuspenseListChildren(children, revealOrder) {
    {
      if ((revealOrder === 'forwards' || revealOrder === 'backwards') && children !==
undefined && children !== null && children !== false) {
        if (isArray(children)) {
          for (var i = 0; i < children.length; i++) {
            if (!validateSuspenseListNestedChild(children[i], i)) {
              return;
            }
          }
        } else {
          var iteratorFn = getIteratorFn(children);

          if (typeof iteratorFn === 'function') {
            var childrenIterator = iteratorFn.call(children);

            if (childrenIterator) {
              var step = childrenIterator.next();
              var _i = 0;
              for (; !step.done; step = childrenIterator.next()) {
                if (!validateSuspenseListNestedChild(step.value, _i)) {
                  return;
                }

                _i++;
              }
            }
          } else {
            error('A single row was passed to a <SuspenseList revealOrder="%s" />. ' +
'This is not useful since it needs multiple rows. ' + 'Did you mean to pass multiple
children or an array?', revealOrder);
          }
        }
      }
    }
  }

  function initSuspenseListRenderState(workInProgress, isBackwards, tail, lastContentRow,
tailMode) {
    var renderState = workInProgress.memoizedState;

    if (renderState === null) {
      workInProgress.memoizedState = {
        isBackwards: isBackwards,
        rendering: null,
        renderingStartTime: 0,
        last: lastContentRow,
        tail: tail,
        tailMode: tailMode
      };
    } else {
      // We can reuse the existing object from previous renders.
      renderState.isBackwards = isBackwards;
      renderState.rendering = null;
      renderState.renderingStartTime = 0;
      renderState.last = lastContentRow;
      renderState.tail = tail;
      renderState.tailMode = tailMode;
    }
  } // This can end up rendering this component multiple passes.
  // The first pass splits the children fibers into two sets. A head and tail.
  // We first render the head. If anything is in fallback state, we do another
  // pass through beginWork to rerender all children (including the tail) with
  // the force suspend context. If the first render didn't have anything in
  // in fallback state. Then we render each row in the tail one-by-one.
  // That happens in the completeWork phase without going back to beginWork.


  function updateSuspenseListComponent(current, workInProgress, renderLanes) {
    var nextProps = workInProgress.pendingProps;
    var revealOrder = nextProps.revealOrder;
    var tailMode = nextProps.tail;
    var newChildren = nextProps.children;
    validateRevealOrder(revealOrder);
    validateTailOptions(tailMode, revealOrder);
    validateSuspenseListChildren(newChildren, revealOrder);
    reconcileChildren(current, workInProgress, newChildren, renderLanes);
    var suspenseContext = suspenseStackCursor.current;
    var shouldForceFallback = hasSuspenseContext(suspenseContext, ForceSuspenseFallback);

    if (shouldForceFallback) {
      suspenseContext = setShallowSuspenseContext(suspenseContext,
ForceSuspenseFallback);
      workInProgress.flags |= DidCapture;
    } else {
      var didSuspendBefore = current !== null && (current.flags & DidCapture) !==
NoFlags;
```

```
      if (didSuspendBefore) {
        // If we previously forced a fallback, we need to schedule work
        // on any nested boundaries to let them know to try to render
        // again. This is the same as context updating.
        propagateSuspenseContextChange(workInProgress, workInProgress.child,
renderLanes);
      }

      suspenseContext = setDefaultShallowSuspenseContext(suspenseContext);
    }

    pushSuspenseContext(workInProgress, suspenseContext);

    if ((workInProgress.mode & ConcurrentMode) === NoMode) {
      // In legacy mode, SuspenseList doesn't work so we just
      // use make it a noop by treating it as the default revealOrder.
      workInProgress.memoizedState = null;
    } else {
      switch (revealOrder) {
        case 'forwards':
          {
            var lastContentRow = findLastContentRow(workInProgress.child);
            var tail;

            if (lastContentRow === null) {
              // The whole list is part of the tail.
              // TODO: We could fast path by just rendering the tail now.
              tail = workInProgress.child;
              workInProgress.child = null;
            } else {
              // Disconnect the tail rows after the content row.
              // We're going to render them separately later.
              tail = lastContentRow.sibling;
              lastContentRow.sibling = null;
            }

            initSuspenseListRenderState(workInProgress, false, // isBackwards
            tail, lastContentRow, tailMode);
            break;
          }

        case 'backwards':
          {
            // We're going to find the first row that has existing content.
            // At the same time we're going to reverse the list of everything
            // we pass in the meantime. That's going to be our tail in reverse
            // order.
            var _tail = null;
            var row = workInProgress.child;
            workInProgress.child = null;

            while (row !== null) {
              var currentRow = row.alternate; // New rows can't be content rows.

              if (currentRow !== null && findFirstSuspended(currentRow) === null) {
                // This is the beginning of the main content.
                workInProgress.child = row;
                break;
              }

              var nextRow = row.sibling;
              row.sibling = _tail;
              _tail = row;
              row = nextRow;
            } // TODO: If workInProgress.child is null, we can continue on the tail
immediately.


            initSuspenseListRenderState(workInProgress, true, // isBackwards
            _tail, null, // last
            tailMode);
            break;
          }

        case 'together':
          {
            initSuspenseListRenderState(workInProgress, false, // isBackwards
            null, // tail
            null, // last
            undefined);
            break;
          }

        default:
          {
            // The default reveal order is the same as not having
            // a boundary.
            workInProgress.memoizedState = null;
          }
      }
    }

    return workInProgress.child;
  }

  function updatePortalComponent(current, workInProgress, renderLanes) {
    pushHostContainer(workInProgress, workInProgress.stateNode.containerInfo);
    var nextChildren = workInProgress.pendingProps;

    if (current === null) {
      // Portals are special because we don't append the children during mount
      // but at commit. Therefore we need to track insertions which the normal
      // flow doesn't do during mount. This doesn't happen at the root because
      // the root always starts with a "current" with a null child.
      // TODO: Consider unifying this with how the root works.
      workInProgress.child = reconcileChildFibers(workInProgress, null, nextChildren,
renderLanes);
    } else {
      reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    }

    return workInProgress.child;
  }

  var hasWarnedAboutUsingNoValuePropOnContextProvider = false;

  function updateContextProvider(current, workInProgress, renderLanes) {
    var providerType = workInProgress.type;
    var context = providerType._context;
    var newProps = workInProgress.pendingProps;
    var oldProps = workInProgress.memoizedProps;
    var newValue = newProps.value;

    {
      if (!('value' in newProps)) {
        if (!hasWarnedAboutUsingNoValuePropOnContextProvider) {
          hasWarnedAboutUsingNoValuePropOnContextProvider = true;

          error('The `value` prop is required for the `<Context.Provider>`. Did you
misspell it or forget to pass it?');
        }
      }

      var providerPropTypes = workInProgress.type.propTypes;

      if (providerPropTypes) {
        checkPropTypes(providerPropTypes, newProps, 'prop', 'Context.Provider');
      }
    }

    pushProvider(workInProgress, context, newValue);

    {
      if (oldProps !== null) {
        var oldValue = oldProps.value;

        if (objectIs(oldValue, newValue)) {
          // No change. Bailout early if children are the same.
          if (oldProps.children === newProps.children && !hasContextChanged()) {
            return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
          }
        } else {
          // The context value changed. Search for matching consumers and schedule
          // them to update.
          propagateContextChange(workInProgress, context, renderLanes);
        }
      }
    }

    var newChildren = newProps.children;
    reconcileChildren(current, workInProgress, newChildren, renderLanes);
    return workInProgress.child;
  }

  var hasWarnedAboutUsingContextAsConsumer = false;

  function updateContextConsumer(current, workInProgress, renderLanes) {
    var context = workInProgress.type; // The logic below for Context differs depending
on PROD or DEV mode. In
    // DEV mode, we create a separate object for Context.Consumer that acts
    // like a proxy to Context. This proxy object adds unnecessary code in PROD
    // so we use the old behaviour (Context.Consumer references Context) to
    // reduce size and overhead. The separate object references context via
    // a property called "_context", which also gives us the ability to check
    // in DEV mode if this property exists or not and warn if it does not.

    {
      if (context._context === undefined) {
        // This may be because it's a Context (rather than a Consumer).
        // Or it may be because it's older React where they're the same thing.
        // We only want to warn if we're sure it's a new React.
        if (context !== context.Consumer) {
          if (!hasWarnedAboutUsingContextAsConsumer) {
            hasWarnedAboutUsingContextAsConsumer = true;

            error('Rendering <Context> directly is not supported and will be removed in '
+ 'a future major release. Did you mean to render <Context.Consumer> instead?');
          }
        }
      } else {
        context = context._context;
      }
    }

    var newProps = workInProgress.pendingProps;
    var render = newProps.children;

    {
      if (typeof render !== 'function') {
        error('A context consumer was rendered with multiple children, or a child ' +
"that isn't a function. A context consumer expects a single child " + 'that is a
function. If you did pass a function, make sure there ' + 'is no trailing or leading
whitespace around it.');
      }
    }

    prepareToReadContext(workInProgress, renderLanes);
    var newValue = readContext(context);

    {
      markComponentRenderStarted(workInProgress);
    }

    var newChildren;

    {
      ReactCurrentOwner$1.current = workInProgress;
      setIsRendering(true);
      newChildren = render(newValue);
      setIsRendering(false);
    }

    {
      markComponentRenderStopped();
    } // React DevTools reads this flag.


    workInProgress.flags |= PerformedWork;
    reconcileChildren(current, workInProgress, newChildren, renderLanes);
    return workInProgress.child;
  }

  function markWorkInProgressReceivedUpdate() {
    didReceiveUpdate = true;
  }

  function resetSuspendedCurrentOnMountInLegacyMode(current, workInProgress) {
    if ((workInProgress.mode & ConcurrentMode) === NoMode) {
      if (current !== null) {
        // A lazy component only mounts if it suspended inside a non-
        // concurrent tree, in an inconsistent state. We want to treat it like
        // a new mount, even though an empty version of it already committed.
        // Disconnect the alternate pointers.
        current.alternate = null;
        workInProgress.alternate = null; // Since this is conceptually a new fiber,
schedule a Placement effect

        workInProgress.flags |= Placement;
      }
    }
  }

  function bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes) {
    if (current !== null) {
      // Reuse previous dependencies
      workInProgress.dependencies = current.dependencies;
    }

    {
      // Don't update "base" render times for bailouts.
      stopProfilerTimerIfRunning();
    }

    markSkippedUpdateLanes(workInProgress.lanes); // Check if the children have any
pending work.

    if (!includesSomeLane(renderLanes, workInProgress.childLanes)) {
      // The children don't have any work either. We can skip them.
      // TODO: Once we add back resuming, we should check if the children are
      // a work-in-progress set. If so, we need to transfer their effects.
      {
        return null;
      }
    } // This fiber doesn't have work, but its subtree does. Clone the child
    // fibers and continue.
```

```
      cloneChildFibers(current, workInProgress);
      return workInProgress.child;
    }

    function remountFiber(current, oldWorkInProgress, newWorkInProgress) {
      {
        var returnFiber = oldWorkInProgress.return;

        if (returnFiber === null) {
          // eslint-disable-next-line react-internal/prod-error-codes
          throw new Error('Cannot swap the root fiber.');
        } // Disconnect from the old current.
        // It will get deleted.


        current.alternate = null;
        oldWorkInProgress.alternate = null; // Connect to the new tree.

        newWorkInProgress.index = oldWorkInProgress.index;
        newWorkInProgress.sibling = oldWorkInProgress.sibling;
        newWorkInProgress.return = oldWorkInProgress.return;
        newWorkInProgress.ref = oldWorkInProgress.ref; // Replace the child/sibling
  pointers above it.

        if (oldWorkInProgress === returnFiber.child) {
          returnFiber.child = newWorkInProgress;
        } else {
          var prevSibling = returnFiber.child;

          if (prevSibling === null) {
            // eslint-disable-next-line react-internal/prod-error-codes
            throw new Error('Expected parent to have a child.');
          }

          while (prevSibling.sibling !== oldWorkInProgress) {
            prevSibling = prevSibling.sibling;

            if (prevSibling === null) {
              // eslint-disable-next-line react-internal/prod-error-codes
              throw new Error('Expected to find the previous sibling.');
            }
          }

          prevSibling.sibling = newWorkInProgress;
        } // Delete the old fiber and place the new one.
        // Since the old fiber is disconnected, we have to schedule it manually.


        var deletions = returnFiber.deletions;

        if (deletions === null) {
          returnFiber.deletions = [current];
          returnFiber.flags |= ChildDeletion;
        } else {
          deletions.push(current);
        }

        newWorkInProgress.flags |= Placement; // Restart work from the new fiber.

        return newWorkInProgress;
      }
    }

    function checkScheduledUpdateOrContext(current, renderLanes) {
      // Before performing an early bailout, we must check if there are pending
      // updates or context.
      var updateLanes = current.lanes;
      if (includesSomeLane(updateLanes, renderLanes)) {
        return true;
      } // No pending update, but because context is propagated lazily, we need

      return false;
    }

    function attemptEarlyBailoutIfNoScheduledUpdate(current, workInProgress, renderLanes) {
      // This fiber does not have any pending work. Bailout without entering
      // the begin phase. There's still some bookkeeping we that needs to be done
      // in this optimized path, mostly pushing stuff onto the stack.
      switch (workInProgress.tag) {
        case HostRoot:
          pushHostRootContext(workInProgress);
          var root = workInProgress.stateNode;

          resetHydrationState();
          break;

        case HostComponent:
          pushHostContext(workInProgress);
          break;

        case ClassComponent:
          {
            var Component = workInProgress.type;

            if (isContextProvider(Component)) {
              pushContextProvider(workInProgress);
            }

            break;
          }

        case HostPortal:
          pushHostContainer(workInProgress, workInProgress.stateNode.containerInfo);
          break;

        case ContextProvider:
          {
            var newValue = workInProgress.memoizedProps.value;
            var context = workInProgress.type._context;
            pushProvider(workInProgress, context, newValue);
            break;
          }

        case Profiler:
          {
            // Profiler should only call onRender when one of its descendants actually
  rendered.
            var hasChildWork = includesSomeLane(renderLanes, workInProgress.childLanes);

            if (hasChildWork) {
              workInProgress.flags |= Update;
            }

            {
              // Reset effect durations for the next eventual effect phase.
              // These are reset during render to allow the DevTools commit hook a chance
  to read them,
              var stateNode = workInProgress.stateNode;
              stateNode.effectDuration = 0;
              stateNode.passiveEffectDuration = 0;
            }
          }

          break;

        case SuspenseComponent:
```

```
        {
          var state = workInProgress.memoizedState;

          if (state !== null) {
            if (state.dehydrated !== null) {
              pushSuspenseContext(workInProgress,
setDefaultShallowSuspenseContext(suspenseStackCursor.current)); // We know that this
component will suspend again because if it has
              // been unsuspended it has committed as a resolved Suspense component.
              // If it needs to be retried, it should have work scheduled on it.

              workInProgress.flags |= DidCapture; // We should never render the children
of a dehydrated boundary until we
              // upgrade it. We return null instead of bailoutOnAlreadyFinishedWork.

              return null;
            } // If this boundary is currently timed out, we need to decide
            // whether to retry the primary children, or to skip over it and
            // go straight to the fallback. Check the priority of the primary
            // child fragment.


            var primaryChildFragment = workInProgress.child;
            var primaryChildLanes = primaryChildFragment.childLanes;

            if (includesSomeLane(renderLanes, primaryChildLanes)) {
              // The primary children have pending work. Use the normal path
              // to attempt to render the primary children again.
              return updateSuspenseComponent(current, workInProgress, renderLanes);
            } else {
              // The primary child fragment does not have pending work marked
              // on it
              pushSuspenseContext(workInProgress,
setDefaultShallowSuspenseContext(suspenseStackCursor.current)); // The primary children
do not have pending work with sufficient
              // priority. Bailout.

              var child = bailoutOnAlreadyFinishedWork(current, workInProgress,
renderLanes);

              if (child !== null) {
                // The fallback children have pending work. Skip over the
                // primary children and work on the fallback.
                return child.sibling;
              } else {
                // Note: We can return `null` here because we already checked
                // whether there were nested context consumers, via the call to
                // `bailoutOnAlreadyFinishedWork` above.
                return null;
              }
            }
          } else {
            pushSuspenseContext(workInProgress,
setDefaultShallowSuspenseContext(suspenseStackCursor.current));
          }

          break;
        }

      case SuspenseListComponent:
        {
          var didSuspendBefore = (current.flags & DidCapture) !== NoFlags;

          var _hasChildWork = includesSomeLane(renderLanes, workInProgress.childLanes);

          if (didSuspendBefore) {
            if (_hasChildWork) {
              // If something was in fallback state last time, and we have all the
              // same children then we're still in progressive loading state.
              // Something might get unblocked by state updates or retries in the
              // tree which will affect the tail. So we need to use the normal
              // path to compute the correct tail.
              return updateSuspenseListComponent(current, workInProgress, renderLanes);
            } // If none of the children had any work, that means that none of
            // them got retried so they'll still be blocked in the same way
            // as before. We can fast bail out.


            workInProgress.flags |= DidCapture;
          } // If nothing suspended before and we're rendering the same children,
          // then the tail doesn't matter. Anything new that suspends will work
          // in the "together" mode, so we can continue from the state we had.


          var renderState = workInProgress.memoizedState;

          if (renderState !== null) {
            // Reset to the "together" mode in case we've started a different
            // update in the past but didn't complete it.
            renderState.rendering = null;
            renderState.tail = null;
            renderState.lastEffect = null;
          }

          pushSuspenseContext(workInProgress, suspenseStackCursor.current);

          if (_hasChildWork) {
            break;
          } else {
            // If none of the children had any work, that means that none of
            // them got retried so they'll still be blocked in the same way
            // as before. We can fast bail out.
            return null;
          }
        }

      case OffscreenComponent:
      case LegacyHiddenComponent:
        {
          // Need to check if the tree still needs to be deferred. This is
          // almost identical to the logic used in the normal update path,
          // so we'll just enter that. The only difference is we'll bail out
          // at the next level instead of this one, because the child props
          // have not changed. Which is fine.
          // TODO: Probably should refactor `beginWork` to split the bailout
          // path from the normal path. I'm tempted to do a labeled break here
          // but I won't :)
          workInProgress.lanes = NoLanes;
          return updateOffscreenComponent(current, workInProgress, renderLanes);
        }
    }

    return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
  }

  function beginWork(current, workInProgress, renderLanes) {
    {
      if (workInProgress._debugNeedsRemount && current !== null) {
        // This will restart the begin phase with a new fiber.
        return remountFiber(current, workInProgress,
createFiberFromTypeAndProps(workInProgress.type, workInProgress.key,
workInProgress.pendingProps, workInProgress._debugOwner || null, workInProgress.mode,
workInProgress.lanes));
      }
    }

    if (current !== null) {
      var oldProps = current.memoizedProps;
      var newProps = workInProgress.pendingProps;

      if (oldProps !== newProps || hasContextChanged() || ( // Force a re-render if the
implementation changed due to hot reload:
       workInProgress.type !== current.type )) {
        // If props or context changed, mark the fiber as having performed work.
        // This may be unset if the props are determined to be equal later (memo).
        didReceiveUpdate = true;
      } else {
        // Neither props nor legacy context changes. Check if there's a pending
        // update or context change.
        var hasScheduledUpdateOrContext = checkScheduledUpdateOrContext(current,
renderLanes);

        if (!hasScheduledUpdateOrContext && // If this is the second pass of an error or
suspense boundary, there
        // may not be work scheduled on `current`, so we check for this flag.
        (workInProgress.flags & DidCapture) === NoFlags) {
          // No pending updates or context. Bail out now.
          didReceiveUpdate = false;
          return attemptEarlyBailoutIfNoScheduledUpdate(current, workInProgress,
renderLanes);
        }

        if ((current.flags & ForceUpdateForLegacySuspense) !== NoFlags) {
          // This is a special case that only exists for legacy mode.
          // See https://github.com/facebook/react/pull/19216.
          didReceiveUpdate = true;
        } else {
          // An update was scheduled on this fiber, but there are no new props
          // nor legacy context. Set this to false. If an update queue or context
          // consumer produces a changed value, it will set this to true. Otherwise,
          // the component will assume the children have not changed and bail out.
          didReceiveUpdate = false;
        }
      }
    } else {
      didReceiveUpdate = false;

      if (getIsHydrating() && isForkedChild(workInProgress)) {
        // Check if this child belongs to a list of muliple children in
        // its parent.
        //
        // In a true multi-threaded implementation, we would render children on
        // parallel threads. This would represent the beginning of a new render
        // thread for this subtree.
        //
        // We only use this for id generation during hydration, which is why the
        // logic is located in this special branch.
        var slotIndex = workInProgress.index;
        var numberOfForks = getForksAtLevel();
        pushTreeId(workInProgress, numberOfForks, slotIndex);
      }
    } // Before entering the begin phase, clear pending update priority.
    // TODO: This assumes that we're about to evaluate the component and process
    // the update queue. However, there's an exception: SimpleMemoComponent
    // sometimes bails out later in the begin phase. This indicates that we should
    // move this assignment out of the common path and into each branch.


    workInProgress.lanes = NoLanes;

    switch (workInProgress.tag) {
      case IndeterminateComponent:
        {
          return mountIndeterminateComponent(current, workInProgress,
workInProgress.type, renderLanes);
        }
      case LazyComponent:
        {
          var elementType = workInProgress.elementType;
          return mountLazyComponent(current, workInProgress, elementType, renderLanes);
        }

      case FunctionComponent:
        {
          var Component = workInProgress.type;
          var unresolvedProps = workInProgress.pendingProps;
          var resolvedProps = workInProgress.elementType === Component ? unresolvedProps
: resolveDefaultProps(Component, unresolvedProps);
          return updateFunctionComponent(current, workInProgress, Component,
resolvedProps, renderLanes);
        }

      case ClassComponent:
        {
          var _Component = workInProgress.type;
          var _unresolvedProps = workInProgress.pendingProps;

          var _resolvedProps = workInProgress.elementType === _Component ?
_unresolvedProps : resolveDefaultProps(_Component, _unresolvedProps);

          return updateClassComponent(current, workInProgress, _Component,
_resolvedProps, renderLanes);
        }

      case HostRoot:
        return updateHostRoot(current, workInProgress, renderLanes);

      case HostComponent:
        return updateHostComponent(current, workInProgress, renderLanes);

      case HostText:
        return updateHostText(current, workInProgress);

      case SuspenseComponent:
        return updateSuspenseComponent(current, workInProgress, renderLanes);

      case HostPortal:
        return updatePortalComponent(current, workInProgress, renderLanes);

      case ForwardRef:
        {
          var type = workInProgress.type;
          var _unresolvedProps2 = workInProgress.pendingProps;

          var _resolvedProps2 = workInProgress.elementType === type ? _unresolvedProps2 :
resolveDefaultProps(type, _unresolvedProps2);

          return updateForwardRef(current, workInProgress, type, _resolvedProps2,
renderLanes);
        }

      case Fragment:
        return updateFragment(current, workInProgress, renderLanes);

      case Mode:
        return updateMode(current, workInProgress, renderLanes);

      case Profiler:
        return updateProfiler(current, workInProgress, renderLanes);

      case ContextProvider:
        return updateContextProvider(current, workInProgress, renderLanes);

      case ContextConsumer:
        return updateContextConsumer(current, workInProgress, renderLanes);
      case MemoComponent:
        {
          var _type2 = workInProgress.type;
          var _unresolvedProps3 = workInProgress.pendingProps; // Resolve outer props
first, then resolve inner props.

          var _resolvedProps3 = resolveDefaultProps(_type2, _unresolvedProps3);

          {
            if (workInProgress.type !== workInProgress.elementType) {
              var outerPropTypes = _type2.propTypes;

              if (outerPropTypes) {
                checkPropTypes(outerPropTypes, _resolvedProps3, // Resolved for outer
only
                'prop', getComponentNameFromType(_type2));
              }
            }
          }

          _resolvedProps3 = resolveDefaultProps(_type2.type, _resolvedProps3);
          return updateMemoComponent(current, workInProgress, _type2, _resolvedProps3,
renderLanes);
        }

      case SimpleMemoComponent:
        {
          return updateSimpleMemoComponent(current, workInProgress, workInProgress.type,
workInProgress.pendingProps, renderLanes);
        }

      case IncompleteClassComponent:
        {
          var _Component2 = workInProgress.type;
          var _unresolvedProps4 = workInProgress.pendingProps;

          var _resolvedProps4 = workInProgress.elementType === _Component2 ?
_unresolvedProps4 : resolveDefaultProps(_Component2, _unresolvedProps4);

          return mountIncompleteClassComponent(current, workInProgress, _Component2,
_resolvedProps4, renderLanes);
        }

      case SuspenseListComponent:
        {
          return updateSuspenseListComponent(current, workInProgress, renderLanes);
        }

      case ScopeComponent:
        {

          break;
        }

      case OffscreenComponent:
        {
          return updateOffscreenComponent(current, workInProgress, renderLanes);
        }
    }

    throw new Error("Unknown unit of work tag (" + workInProgress.tag + "). This error is
likely caused by a bug in " + 'React. Please file an issue.');
  }

  function markUpdate(workInProgress) {
    // Tag the fiber with an update effect. This turns a Placement into
    // a PlacementAndUpdate.
    workInProgress.flags |= Update;
  }

  function markRef$1(workInProgress) {
    workInProgress.flags |= Ref;

    {
      workInProgress.flags |= RefStatic;
    }
  }

  var appendAllChildren;
  var updateHostContainer;
  var updateHostComponent$1;
  var updateHostText$1;

  {
    // Mutation mode
    appendAllChildren = function (parent, workInProgress, needsVisibilityToggle,
isHidden) {
      // We only have the top Fiber that was created but we need recurse down its
      // children to find all the terminal nodes.
      var node = workInProgress.child;

      while (node !== null) {
        if (node.tag === HostComponent || node.tag === HostText) {
          appendInitialChild(parent, node.stateNode);
        } else if (node.tag === HostPortal) ; else if (node.child !== null) {
          node.child.return = node;
          node = node.child;
          continue;
        }

        if (node === workInProgress) {
          return;
        }

        while (node.sibling === null) {
          if (node.return === null || node.return === workInProgress) {
            return;
          }

          node = node.return;
        }

        node.sibling.return = node.return;
        node = node.sibling;
      }
    };

    updateHostContainer = function (current, workInProgress) {// Noop
    };

    updateHostComponent$1 = function (current, workInProgress, type, newProps,
rootContainerInstance) {
      // If we have an alternate, that means this is an update and we need to
      // schedule a side-effect to do the updates.
      var oldProps = current.memoizedProps;

      if (oldProps === newProps) {
        // In mutation mode, this is sufficient for a bailout because
        // we won't touch this node even if children changed.
        return;
      } // If we get updated because one of our children updated, we don't
      // have newProps so we'll have to reuse them.
      // TODO: Split the update API as separate for the props vs. children.
      // Even better would be if children weren't special cased at all tho.


      var instance = workInProgress.stateNode;
      var currentHostContext = getHostContext(); // TODO: Experiencing an error where
oldProps is null. Suggests a host
      // component is hitting the resume path. Figure out why. Possibly
      // related to `hidden`.

      var updatePayload = prepareUpdate(instance, type, oldProps, newProps,
rootContainerInstance, currentHostContext); // TODO: Type this specific to this type of
component.

      workInProgress.updateQueue = updatePayload; // If the update payload indicates that
there is a change or if there
      // is a new ref we mark this as an update. All the work is done in commitWork.

      if (updatePayload) {
        markUpdate(workInProgress);
      }
    };

    updateHostText$1 = function (current, workInProgress, oldText, newText) {
      // If the text differs, mark it as an update. All the work in done in commitWork.
      if (oldText !== newText) {
        markUpdate(workInProgress);
      }
    };
  }

  function cutOffTailIfNeeded(renderState, hasRenderedATailFallback) {
    if (getIsHydrating()) {
      // If we're hydrating, we should consume as many items as we can
      // so we don't leave any behind.
      return;
    }

    switch (renderState.tailMode) {
      case 'hidden':
        {
          // Any insertions at the end of the tail list after this point
          // should be invisible. If there are already mounted boundaries
          // anything before them are not considered for collapsing.
          // Therefore we need to go through the whole tail to find if
          // there are any.
          var tailNode = renderState.tail;
          var lastTailNode = null;

          while (tailNode !== null) {
            if (tailNode.alternate !== null) {
              lastTailNode = tailNode;
            }

            tailNode = tailNode.sibling;
          } // Next we're simply going to delete all insertions after the
          // last rendered item.


          if (lastTailNode === null) {
            // All remaining items in the tail are insertions.
            renderState.tail = null;
          } else {
            // Detach the insertion after the last node that was already
            // inserted.
            lastTailNode.sibling = null;
          }

          break;
        }

      case 'collapsed':
        {
          // Any insertions at the end of the tail list after this point
          // should be invisible. If there are already mounted boundaries
          // anything before them are not considered for collapsing.
          // Therefore we need to go through the whole tail to find if
          // there are any.
          var _tailNode = renderState.tail;
          var _lastTailNode = null;

          while (_tailNode !== null) {
            if (_tailNode.alternate !== null) {
              _lastTailNode = _tailNode;
            }

            _tailNode = _tailNode.sibling;
          } // Next we're simply going to delete all insertions after the
          // last rendered item.


          if (_lastTailNode === null) {
            // All remaining items in the tail are insertions.
            if (!hasRenderedATailFallback && renderState.tail !== null) {
              // We suspended during the head. We want to show at least one
              // row at the tail. So we'll keep on and cut off the rest.
              renderState.tail.sibling = null;
            } else {
              renderState.tail = null;
            }
          } else {
            // Detach the insertion after the last node that was already
            // inserted.
            _lastTailNode.sibling = null;
          }

          break;
        }
    }
  }

  function bubbleProperties(completedWork) {
    var didBailout = completedWork.alternate !== null && completedWork.alternate.child
=== completedWork.child;
    var newChildLanes = NoLanes;
    var subtreeFlags = NoFlags;

    if (!didBailout) {
      // Bubble up the earliest expiration time.
      if ( (completedWork.mode & ProfileMode) !== NoMode) {
        // In profiling mode, resetChildExpirationTime is also used to reset
        // profiler durations.
        var actualDuration = completedWork.actualDuration;
        var treeBaseDuration = completedWork.selfBaseDuration;
        var child = completedWork.child;

        while (child !== null) {
          newChildLanes = mergeLanes(newChildLanes, mergeLanes(child.lanes,
child.childLanes));
          subtreeFlags |= child.subtreeFlags;
          subtreeFlags |= child.flags; // When a fiber is cloned, its actualDuration is
reset to 0. This value will
          // only be updated if work is done on the fiber (i.e. it doesn't bailout).
          // When work is done, it should bubble to the parent's actualDuration. If
          // the fiber has not been cloned though, (meaning no work was done), then
          // this value will reflect the amount of time spent working on a previous
          // render. In that case it should not bubble. We determine whether it was
          // cloned by comparing the child pointer.

          actualDuration += child.actualDuration;
          treeBaseDuration += child.treeBaseDuration;
          child = child.sibling;
        }
```

```
          completedWork.actualDuration = actualDuration;
          completedWork.treeBaseDuration = treeBaseDuration;
        } else {
          var _child = completedWork.child;

          while (_child !== null) {
            newChildLanes = mergeLanes(newChildLanes, mergeLanes(_child.lanes,
_child.childLanes));
            subtreeFlags |= _child.subtreeFlags;
            subtreeFlags |= _child.flags; // Update the return pointer so the tree is
consistent. This is a code
            // smell because it assumes the commit phase is never concurrent with
            // the render phase. Will address during refactor to alternate model.

            _child.return = completedWork;
            _child = _child.sibling;
          }
        }

        completedWork.subtreeFlags |= subtreeFlags;
      } else {
        // Bubble up the earliest expiration time.
        if ( (completedWork.mode & ProfileMode) !== NoMode) {
          // In profiling mode, resetChildExpirationTime is also used to reset
          // profiler durations.
          var _treeBaseDuration = completedWork.selfBaseDuration;
          var _child2 = completedWork.child;

          while (_child2 !== null) {
            newChildLanes = mergeLanes(newChildLanes, mergeLanes(_child2.lanes,
_child2.childLanes)); // "Static" flags share the lifetime of the fiber/hook they belong
to,
            // so we should bubble those up even during a bailout. All the other
            // flags have a lifetime only of a single render + commit, so we should
            // ignore them.

            subtreeFlags |= _child2.subtreeFlags & StaticMask;
            subtreeFlags |= _child2.flags & StaticMask;
            _treeBaseDuration += _child2.treeBaseDuration;
            _child2 = _child2.sibling;
          }

          completedWork.treeBaseDuration = _treeBaseDuration;
        } else {
          var _child3 = completedWork.child;

          while (_child3 !== null) {
            newChildLanes = mergeLanes(newChildLanes, mergeLanes(_child3.lanes,
_child3.childLanes)); // "Static" flags share the lifetime of the fiber/hook they belong
to,
            // so we should bubble those up even during a bailout. All the other
            // flags have a lifetime only of a single render + commit, so we should
            // ignore them.

            subtreeFlags |= _child3.subtreeFlags & StaticMask;
            subtreeFlags |= _child3.flags & StaticMask; // Update the return pointer so the
tree is consistent. This is a code
            // smell because it assumes the commit phase is never concurrent with
            // the render phase. Will address during refactor to alternate model.

            _child3.return = completedWork;
            _child3 = _child3.sibling;
          }
        }

        completedWork.subtreeFlags |= subtreeFlags;
      }
```

```
    completedWork.childLanes = newChildLanes;
    return didBailout;
  }

  function completeDehydratedSuspenseBoundary(current, workInProgress, nextState) {
    if (hasUnhydratedTailNodes() && (workInProgress.mode & ConcurrentMode) !== NoMode &&
(workInProgress.flags & DidCapture) === NoFlags) {
      warnIfUnhydratedTailNodes(workInProgress);
      resetHydrationState();
      workInProgress.flags |= ForceClientRender | Incomplete | ShouldCapture;
      return false;
    }

    var wasHydrated = popHydrationState(workInProgress);

    if (nextState !== null && nextState.dehydrated !== null) {
      // We might be inside a hydration state the first time we're picking up this
      // Suspense boundary, and also after we've reentered it for further hydration.
      if (current === null) {
        if (!wasHydrated) {
          throw new Error('A dehydrated suspense component was completed without a
hydrated node. ' + 'This is probably a bug in React.');
        }

        prepareToHydrateHostSuspenseInstance(workInProgress);
        bubbleProperties(workInProgress);

        {
          if ((workInProgress.mode & ProfileMode) !== NoMode) {
            var isTimedOutSuspense = nextState !== null;

            if (isTimedOutSuspense) {
              // Don't count time spent in a timed out Suspense subtree as part of the
base duration.
              var primaryChildFragment = workInProgress.child;

              if (primaryChildFragment !== null) {
                // $FlowFixMe Flow doesn't support type casting in combination with the -
= operator
                workInProgress.treeBaseDuration -= primaryChildFragment.treeBaseDuration;
              }
            }
          }
        }

        return false;
      } else {
        // We might have reentered this boundary to hydrate it. If so, we need to reset
the hydration
        // state since we're now exiting out of it. popHydrationState doesn't do that for
us.
        resetHydrationState();

        if ((workInProgress.flags & DidCapture) === NoFlags) {
          // This boundary did not suspend so it's now hydrated and unsuspended.
          workInProgress.memoizedState = null;
        } // If nothing suspended, we need to schedule an effect to mark this boundary
        // as having hydrated so events know that they're free to be invoked.
        // It's also a signal to replay events and the suspense callback.
        // If something suspended, schedule an effect to attach retry listeners.
        // So we might as well always mark this.


        workInProgress.flags |= Update;
        bubbleProperties(workInProgress);

        {
          if ((workInProgress.mode & ProfileMode) !== NoMode) {
            var _isTimedOutSuspense = nextState !== null;
            if (_isTimedOutSuspense) {
              // Don't count time spent in a timed out Suspense subtree as part of the
base duration.
              var _primaryChildFragment = workInProgress.child;

              if (_primaryChildFragment !== null) {
                // $FlowFixMe Flow doesn't support type casting in combination with the -
= operator
                workInProgress.treeBaseDuration -=
_primaryChildFragment.treeBaseDuration;
              }
            }
          }
        }

        return false;
      }
    } else {
      // Successfully completed this tree. If this was a forced client render,
      // there may have been recoverable errors during first hydration
      // attempt. If so, add them to a queue so we can log them in the
      // commit phase.
      upgradeHydrationErrorsToRecoverable(); // Fall through to normal Suspense path

      return true;
    }
  }

  function completeWork(current, workInProgress, renderLanes) {
    var newProps = workInProgress.pendingProps; // Note: This intentionally doesn't check
if we're hydrating because comparing
    // to the current tree provider fiber is just as fast and less error-prone.
    // Ideally we would have a special version of the work loop only
    // for hydration.

    popTreeContext(workInProgress);

    switch (workInProgress.tag) {
      case IndeterminateComponent:
      case LazyComponent:
      case SimpleMemoComponent:
      case FunctionComponent:
      case ForwardRef:
      case Fragment:
      case Mode:
      case Profiler:
      case ContextConsumer:
      case MemoComponent:
        bubbleProperties(workInProgress);
        return null;

      case ClassComponent:
        {
          var Component = workInProgress.type;

          if (isContextProvider(Component)) {
            popContext(workInProgress);
          }

          bubbleProperties(workInProgress);
          return null;
        }

      case HostRoot:
        {
          var fiberRoot = workInProgress.stateNode;
          popHostContainer(workInProgress);
          popTopLevelContextObject(workInProgress);
          resetWorkInProgressVersions();

          if (fiberRoot.pendingContext) {
            fiberRoot.context = fiberRoot.pendingContext;
            fiberRoot.pendingContext = null;
          }

          if (current === null || current.child === null) {
            // If we hydrated, pop so that we can delete any remaining children
            // that weren't hydrated.
            var wasHydrated = popHydrationState(workInProgress);

            if (wasHydrated) {
              // If we hydrated, then we'll need to schedule an update for
              // the commit side-effects on the root.
              markUpdate(workInProgress);
            } else {
              if (current !== null) {
                var prevState = current.memoizedState;

                if ( // Check if this is a client root
                !prevState.isDehydrated || // Check if we reverted to client rendering
(e.g. due to an error)
                (workInProgress.flags & ForceClientRender) !== NoFlags) {
                  // Schedule an effect to clear this container at the start of the
                  // next commit. This handles the case of React rendering into a
                  // container with previous children. It's also safe to do for
                  // updates too, because current.child would only be null if the
                  // previous render was null (so the container would already
                  // be empty).
                  workInProgress.flags |= Snapshot; // If this was a forced client
render, there may have been
                  // recoverable errors during first hydration attempt. If so, add
                  // them to a queue so we can log them in the commit phase.

                  upgradeHydrationErrorsToRecoverable();
                }
              }
            }
          }

          updateHostContainer(current, workInProgress);
          bubbleProperties(workInProgress);

          return null;
        }

      case HostComponent:
        {
          popHostContext(workInProgress);
          var rootContainerInstance = getRootHostContainer();
          var type = workInProgress.type;

          if (current !== null && workInProgress.stateNode != null) {
            updateHostComponent$1(current, workInProgress, type, newProps,
rootContainerInstance);

            if (current.ref !== workInProgress.ref) {
              markRef$1(workInProgress);
            }
          } else {
            if (!newProps) {
              if (workInProgress.stateNode === null) {
                throw new Error('We must have new props for new mounts. This error is
likely ' + 'caused by a bug in React. Please file an issue.');
              } // This can happen when we abort work.


              bubbleProperties(workInProgress);
              return null;
            }

            var currentHostContext = getHostContext(); // TODO: Move createInstance to
beginWork and keep it on a context
            // "stack" as the parent. Then append children as we go in beginWork
            // or completeWork depending on whether we want to add them top->down or
            // bottom->up. Top->down is faster in IE11.

            var _wasHydrated = popHydrationState(workInProgress);

            if (_wasHydrated) {
              // TODO: Move this and createInstance step into the beginPhase
              // to consolidate.
              if (prepareToHydrateHostInstance(workInProgress, rootContainerInstance,
currentHostContext)) {
                // If changes to the hydrated node need to be applied at the
                // commit-phase we mark this as such.
                markUpdate(workInProgress);
              }
            } else {
              var instance = createInstance(type, newProps, rootContainerInstance,
currentHostContext, workInProgress);
              appendAllChildren(instance, workInProgress, false, false);
              workInProgress.stateNode = instance; // Certain renderers require commit-
time effects for initial mount.
              // (eg DOM renderer supports auto-focus for certain elements).
              // Make sure such renderers get scheduled for later work.

              if (finalizeInitialChildren(instance, type, newProps,
rootContainerInstance)) {
                markUpdate(workInProgress);
              }
            }

            if (workInProgress.ref !== null) {
              // If there is a ref on a host node we need to schedule a callback
              markRef$1(workInProgress);
            }
          }

          bubbleProperties(workInProgress);
          return null;
        }

      case HostText:
        {
          var newText = newProps;

          if (current && workInProgress.stateNode != null) {
            var oldText = current.memoizedProps; // If we have an alternate, that means
this is an update and we need
            // to schedule a side-effect to do the updates.

            updateHostText$1(current, workInProgress, oldText, newText);
          } else {
            if (typeof newText !== 'string') {
              if (workInProgress.stateNode === null) {
                throw new Error('We must have new props for new mounts. This error is
likely ' + 'caused by a bug in React. Please file an issue.');
              } // This can happen when we abort work.

            }

            var _rootContainerInstance = getRootHostContainer();

            var _currentHostContext = getHostContext();

            var _wasHydrated2 = popHydrationState(workInProgress);
            if (_wasHydrated2) {
              if (prepareToHydrateHostTextInstance(workInProgress)) {
                markUpdate(workInProgress);
              }
            } else {
              workInProgress.stateNode = createTextInstance(newText,
_rootContainerInstance, _currentHostContext, workInProgress);
            }
          }

          bubbleProperties(workInProgress);
          return null;
        }

      case SuspenseComponent:
        {
          popSuspenseContext(workInProgress);
          var nextState = workInProgress.memoizedState; // Special path for dehydrated
boundaries. We may eventually move this
          // to its own fiber type so that we can add other kinds of hydration
          // boundaries that aren't associated with a Suspense tree. In anticipation
          // of such a refactor, all the hydration logic is contained in
          // this branch.

          if (current === null || current.memoizedState !== null &&
current.memoizedState.dehydrated !== null) {
            var fallthroughToNormalSuspensePath =
completeDehydratedSuspenseBoundary(current, workInProgress, nextState);

            if (!fallthroughToNormalSuspensePath) {
              if (workInProgress.flags & ShouldCapture) {
                // Special case. There were remaining unhydrated nodes. We treat
                // this as a mismatch. Revert to client rendering.
                return workInProgress;
              } else {
                // Did not finish hydrating, either because this is the initial
                // render or because something suspended.
                return null;
              }
            } // Continue with the normal Suspense path.

          }

          if ((workInProgress.flags & DidCapture) !== NoFlags) {
            // Something suspended. Re-render with the fallback children.
            workInProgress.lanes = renderLanes; // Do not reset the effect list.

            if ( (workInProgress.mode & ProfileMode) !== NoMode) {
              transferActualDuration(workInProgress);
            } // Don't bubble properties in this case.


            return workInProgress;
          }

          var nextDidTimeout = nextState !== null;
          var prevDidTimeout = current !== null && current.memoizedState !== null;
          // a passive effect, which is when we process the transitions


          if (nextDidTimeout !== prevDidTimeout) {
            // an effect to toggle the subtree's visibility. When we switch from
            // fallback -> primary, the inner Offscreen fiber schedules this effect
            // as part of its normal complete phase. But when we switch from
            // primary -> fallback, the inner Offscreen fiber does not have a complete
            // phase. So we need to schedule its effect here.
            //
            // We also use this flag to connect/disconnect the effects, but the same
            // logic applies: when re-connecting, the Offscreen fiber's complete
            // phase will handle scheduling the effect. It's only when the fallback
            // is active that we have to do anything special.


            if (nextDidTimeout) {
              var _offscreenFiber2 = workInProgress.child;
              _offscreenFiber2.flags |= Visibility; // TODO: This will still suspend a
synchronous tree if anything
              // in the concurrent tree already suspended during this render.
              // This is a known bug.

              if ((workInProgress.mode & ConcurrentMode) !== NoMode) {
                // TODO: Move this back to throwException because this is too late
                // if this is a large tree which is common for initial loads. We
                // don't know if we should restart a render or not until we get
                // this marker, and this is too late.
                // If this render already had a ping or lower pri updates,
                // and this is the first time we know we're going to suspend we
                // should be able to immediately restart from within throwException.
                var hasInvisibleChildContext = current === null &&
(workInProgress.memoizedProps.unstable_avoidThisFallback !== true ||
!enableSuspenseAvoidThisFallback);

                if (hasInvisibleChildContext ||
hasSuspenseContext(suspenseStackCursor.current, InvisibleParentSuspenseContext)) {
                  // If this was in an invisible tree or a new render, then showing
                  // this boundary is ok.
                  renderDidSuspend();
                } else {
                  // Otherwise, we're going to have to hide content so we should
                  // suspend for longer if possible.
                  renderDidSuspendDelayIfPossible();
                }
              }
            }
          }

          var wakeables = workInProgress.updateQueue;

          if (wakeables !== null) {
            // Schedule an effect to attach a retry listener to the promise.
            // TODO: Move to passive phase
            workInProgress.flags |= Update;
          }

          bubbleProperties(workInProgress);

          {
            if ((workInProgress.mode & ProfileMode) !== NoMode) {
              if (nextDidTimeout) {
                // Don't count time spent in a timed out Suspense subtree as part of the
base duration.
                var primaryChildFragment = workInProgress.child;

                if (primaryChildFragment !== null) {
                  // $FlowFixMe Flow doesn't support type casting in combination with the
-= operator
                  workInProgress.treeBaseDuration -=
primaryChildFragment.treeBaseDuration;
                }
              }
            }
          }

          return null;
        }

      case HostPortal:
        popHostContainer(workInProgress);
        updateHostContainer(current, workInProgress);

        if (current === null) {
          preparePortalMount(workInProgress.stateNode.containerInfo);
        }

        bubbleProperties(workInProgress);
        return null;

      case ContextProvider:
        // Pop provider fiber
        var context = workInProgress.type._context;
        popProvider(context, workInProgress);
        bubbleProperties(workInProgress);
        return null;

      case IncompleteClassComponent:
        {
          // Same as class component case. I put it down here so that the tags are
          // sequential to ensure this switch is compiled to a jump table.
          var _Component = workInProgress.type;

          if (isContextProvider(_Component)) {
            popContext(workInProgress);
          }

          bubbleProperties(workInProgress);
          return null;
        }

      case SuspenseListComponent:
        {
          popSuspenseContext(workInProgress);
          var renderState = workInProgress.memoizedState;

          if (renderState === null) {
            // We're running in the default, "independent" mode.
            // We don't do anything in this mode.
            bubbleProperties(workInProgress);
            return null;
          }

          var didSuspendAlready = (workInProgress.flags & DidCapture) !== NoFlags;
          var renderedTail = renderState.rendering;

          if (renderedTail === null) {
            // We just rendered the head.
            if (!didSuspendAlready) {
              // This is the first pass. We need to figure out if anything is still
              // suspended in the rendered set.
              // If new content unsuspended, but there's still some content that
              // didn't. Then we need to do a second pass that forces everything
              // to keep showing their fallbacks.
              // We might be suspended if something in this render pass suspended, or
              // something in the previous committed pass suspended. Otherwise,
              // there's no chance so we can skip the expensive call to
              // findFirstSuspended.
              var cannotBeSuspended = renderHasNotSuspendedYet() && (current === null ||
(current.flags & DidCapture) === NoFlags);

              if (!cannotBeSuspended) {
                var row = workInProgress.child;

                while (row !== null) {
                  var suspended = findFirstSuspended(row);

                  if (suspended !== null) {
                    didSuspendAlready = true;
                    workInProgress.flags |= DidCapture;
                    cutOffTailIfNeeded(renderState, false); // If this is a newly
suspended tree, it might not get committed as
                    // part of the second pass. In that case nothing will subscribe to
                    // its thenables. Instead, we'll transfer its thenables to the
                    // SuspenseList so that it can retry if they resolve.
                    // There might be multiple of these in the list but since we're
                    // going to wait for all of them anyway, it doesn't really matter
                    // which ones gets to ping. In theory we could get clever and keep
                    // track of how many dependencies remain but it gets tricky because
                    // in the meantime, we can add/remove/change items and dependencies.
                    // We might bail out of the loop before finding any but that
                    // doesn't matter since that means that the other boundaries that
                    // we did find already has their listeners attached.

                    var newThenables = suspended.updateQueue;

                    if (newThenables !== null) {
                      workInProgress.updateQueue = newThenables;
                      workInProgress.flags |= Update;
                    } // Rerender the whole list, but this time, we'll force fallbacks
                    // to stay in place.
                    // Reset the effect flags before doing the second pass since that's
now invalid.
                    // Reset the child fibers to their original state.


                    workInProgress.subtreeFlags = NoFlags;
                    resetChildFibers(workInProgress, renderLanes); // Set up the Suspense
Context to force suspense and immediately
                    // rerender the children.

                    pushSuspenseContext(workInProgress,
setShallowSuspenseContext(suspenseStackCursor.current, ForceSuspenseFallback)); // Don't
bubble properties in this case.

                    return workInProgress.child;
                  }

                  row = row.sibling;
                }
              }

              if (renderState.tail !== null && now() > getRenderTargetTime()) {
                // We have already passed our CPU deadline but we still have rows
                // left in the tail. We'll just give up further attempts to render
                // the main content and only render fallbacks.
                workInProgress.flags |= DidCapture;
                didSuspendAlready = true;
                cutOffTailIfNeeded(renderState, false); // Since nothing actually
suspended, there will nothing to ping this
                // to get it started back up to attempt the next item. While in terms
                // of priority this work has the same priority as this current render,
                // it's not part of the same transition once the transition has
                // committed. If it's sync, we still want to yield so that it can be
                // painted. Conceptually, this is really the same as pinging.
                // We can use any RetryLane even if it's the one currently rendering
                // since we're leaving it behind on this node.

                workInProgress.lanes = SomeRetryLane;
              }
            } else {
              cutOffTailIfNeeded(renderState, false);
            } // Next we're going to render the tail.

          } else {
            // Append the rendered row to the child list.
            if (!didSuspendAlready) {
              var _suspended = findFirstSuspended(renderedTail);
              if (_suspended !== null) {
                workInProgress.flags |= DidCapture;
                didSuspendAlready = true; // Ensure we transfer the update queue to the
parent so that it doesn't
                // get lost if this row ends up dropped during a second pass.

                var _newThenables = _suspended.updateQueue;

                if (_newThenables !== null) {
                  workInProgress.updateQueue = _newThenables;
                  workInProgress.flags |= Update;
                }

                cutOffTailIfNeeded(renderState, true); // This might have been modified.

                if (renderState.tail === null && renderState.tailMode === 'hidden' &&
!renderedTail.alternate && !getIsHydrating() // We don't cut it if we're hydrating.
                ) {
                    // We're done.
                    bubbleProperties(workInProgress);
                    return null;
                  }
              } else if ( // The time it took to render last row is greater than the
remaining
              // time we have to render. So rendering one more row would likely
              // exceed it.
              now() * 2 - renderState.renderingStartTime > getRenderTargetTime() &&
renderLanes !== OffscreenLane) {
                // We have now passed our CPU deadline and we'll just give up further
                // attempts to render the main content and only render fallbacks.
                // The assumption is that this is usually faster.
                workInProgress.flags |= DidCapture;
                didSuspendAlready = true;
                cutOffTailIfNeeded(renderState, false); // Since nothing actually
suspended, there will nothing to ping this
                // to get it started back up to attempt the next item. While in terms
                // of priority this work has the same priority as this current render,
                // it's not part of the same transition once the transition has
                // committed. If it's sync, we still want to yield so that it can be
                // painted. Conceptually, this is really the same as pinging.
                // We can use any RetryLane even if it's the one currently rendering
                // since we're leaving it behind on this node.

                workInProgress.lanes = SomeRetryLane;
              }
            }

            if (renderState.isBackwards) {
              // The effect list of the backwards tail will have been added
              // to the end. This breaks the guarantee that life-cycles fire in
              // sibling order but that isn't a strong guarantee promised by React.
              // Especially since these might also just pop in during future commits.
              // Append to the beginning of the list.
              renderedTail.sibling = workInProgress.child;
              workInProgress.child = renderedTail;
            } else {
              var previousSibling = renderState.last;

              if (previousSibling !== null) {
                previousSibling.sibling = renderedTail;
              } else {
                workInProgress.child = renderedTail;
              }

              renderState.last = renderedTail;
            }
          }
          if (renderState.tail !== null) {
            // We still have tail rows to render.
            // Pop a row.
            var next = renderState.tail;
            renderState.rendering = next;
            renderState.tail = next.sibling;
            renderState.renderingStartTime = now();
            next.sibling = null; // Restore the context.
            // TODO: We can probably just avoid popping it instead and only
            // setting it the first time we go from not suspended to suspended.

            var suspenseContext = suspenseStackCursor.current;

            if (didSuspendAlready) {
              suspenseContext = setShallowSuspenseContext(suspenseContext,
ForceSuspenseFallback);
            } else {
              suspenseContext = setDefaultShallowSuspenseContext(suspenseContext);
            }

            pushSuspenseContext(workInProgress, suspenseContext); // Do a pass over the
next row.
            // Don't bubble properties in this case.

            return next;
          }

          bubbleProperties(workInProgress);
          return null;
        }

      case ScopeComponent:
        {

          break;
        }

      case OffscreenComponent:
      case LegacyHiddenComponent:
        {
          popRenderLanes(workInProgress);
          var _nextState = workInProgress.memoizedState;
          var nextIsHidden = _nextState !== null;

          if (current !== null) {
            var _prevState = current.memoizedState;
            var prevIsHidden = _prevState !== null;

            if (prevIsHidden !== nextIsHidden && ( // LegacyHidden doesn't do any hiding
— it only pre-renders.
            !enableLegacyHidden )) {
              workInProgress.flags |= Visibility;
            }
          }

          if (!nextIsHidden || (workInProgress.mode & ConcurrentMode) === NoMode) {
            bubbleProperties(workInProgress);
          } else {
            // Don't bubble properties for hidden children unless we're rendering
            // at offscreen priority.
            if (includesSomeLane(subtreeRenderLanes, OffscreenLane)) {
              bubbleProperties(workInProgress);

              {
                // Check if there was an insertion or update in the hidden subtree.
                // If so, we need to hide those nodes in the commit phase, so
                // schedule a visibility effect.
                if ( workInProgress.subtreeFlags & (Placement | Update)) {
                  workInProgress.flags |= Visibility;
                }
              }
            }
          }
          return null;
        }

      case CacheComponent:
        {

          return null;
        }

      case TracingMarkerComponent:
        {

          return null;
        }
    }

    throw new Error("Unknown unit of work tag (" + workInProgress.tag + "). This error is
likely caused by a bug in " + 'React. Please file an issue.');
  }

  function unwindWork(current, workInProgress, renderLanes) {
    // Note: This intentionally doesn't check if we're hydrating because comparing
    // to the current tree provider fiber is just as fast and less error-prone.
    // Ideally we would have a special version of the work loop only
    // for hydration.
    popTreeContext(workInProgress);

    switch (workInProgress.tag) {
      case ClassComponent:
        {
          var Component = workInProgress.type;

          if (isContextProvider(Component)) {
            popContext(workInProgress);
          }

          var flags = workInProgress.flags;

          if (flags & ShouldCapture) {
            workInProgress.flags = flags & ~ShouldCapture | DidCapture;

            if ( (workInProgress.mode & ProfileMode) !== NoMode) {
              transferActualDuration(workInProgress);
            }

            return workInProgress;
          }

          return null;
        }

      case HostRoot:
        {
          var root = workInProgress.stateNode;
          popHostContainer(workInProgress);
          popTopLevelContextObject(workInProgress);
          resetWorkInProgressVersions();
          var _flags = workInProgress.flags;

          if ((_flags & ShouldCapture) !== NoFlags && (_flags & DidCapture) === NoFlags)
{
            // There was an error during render that wasn't captured by a suspense
            // boundary. Do a second pass on the root to unmount the children.
            workInProgress.flags = _flags & ~ShouldCapture | DidCapture;
            return workInProgress;
          } // We unwound to the root without completing it. Exit.


          return null;
        }

      case HostComponent:
        {
          // TODO: popHydrationState
          popHostContext(workInProgress);
          return null;
        }

      case SuspenseComponent:
        {
          popSuspenseContext(workInProgress);
          var suspenseState = workInProgress.memoizedState;

          if (suspenseState !== null && suspenseState.dehydrated !== null) {
            if (workInProgress.alternate === null) {
              throw new Error('Threw in newly mounted dehydrated component. This is
likely a bug in ' + 'React. Please file an issue.');
            }

            resetHydrationState();
          }

          var _flags2 = workInProgress.flags;

          if (_flags2 & ShouldCapture) {
            workInProgress.flags = _flags2 & ~ShouldCapture | DidCapture; // Captured a
suspense effect. Re-render the boundary.

            if ( (workInProgress.mode & ProfileMode) !== NoMode) {
              transferActualDuration(workInProgress);
            }

            return workInProgress;
          }

          return null;
        }

      case SuspenseListComponent:
        {
          popSuspenseContext(workInProgress); // SuspenseList doesn't actually catch
anything. It should've been
          // caught by a nested boundary. If not, it should bubble through.

          return null;
        }

      case HostPortal:
        popHostContainer(workInProgress);
        return null;

      case ContextProvider:
        var context = workInProgress.type._context;
        popProvider(context, workInProgress);
        return null;

      case OffscreenComponent:
      case LegacyHiddenComponent:
        popRenderLanes(workInProgress);
        return null;

      case CacheComponent:

        return null;
      default:
        return null;
    }
  }

  function unwindInterruptedWork(current, interruptedWork, renderLanes) {
    // Note: This intentionally doesn't check if we're hydrating because comparing
    // to the current tree provider fiber is just as fast and less error-prone.
    // Ideally we would have a special version of the work loop only
    // for hydration.
    popTreeContext(interruptedWork);

    switch (interruptedWork.tag) {
      case ClassComponent:
        {
          var childContextTypes = interruptedWork.type.childContextTypes;

          if (childContextTypes !== null && childContextTypes !== undefined) {
            popContext(interruptedWork);
          }

          break;
        }

      case HostRoot:
        {
          var root = interruptedWork.stateNode;
          popHostContainer(interruptedWork);
          popTopLevelContextObject(interruptedWork);
          resetWorkInProgressVersions();
          break;
        }

      case HostComponent:
        {
          popHostContext(interruptedWork);
          break;
        }

      case HostPortal:
        popHostContainer(interruptedWork);
        break;

      case SuspenseComponent:
        popSuspenseContext(interruptedWork);
        break;

      case SuspenseListComponent:
        popSuspenseContext(interruptedWork);
        break;

      case ContextProvider:
        var context = interruptedWork.type._context;
        popProvider(context, interruptedWork);
        break;

      case OffscreenComponent:
      case LegacyHiddenComponent:
        popRenderLanes(interruptedWork);
        break;
    }
  }

  var didWarnAboutUndefinedSnapshotBeforeUpdate = null;

  {
    didWarnAboutUndefinedSnapshotBeforeUpdate = new Set();
  } // Used during the commit phase to track the state of the Offscreen component stack.
  // Allows us to avoid traversing the return path to find the nearest Offscreen
ancestor.
  // Only used when enableSuspenseLayoutEffectSemantics is enabled.


  var offscreenSubtreeIsHidden = false;
  var offscreenSubtreeWasHidden = false;
  var PossiblyWeakSet = typeof WeakSet === 'function' ? WeakSet : Set;
  var nextEffect = null; // Used for Profiling builds to track updaters.

  var inProgressLanes = null;
  var inProgressRoot = null;
  function reportUncaughtErrorInDEV(error) {
    // Wrapping each small part of the commit phase into a guarded
    // callback is a bit too slow (https://github.com/facebook/react/pull/21666).
    // But we rely on it to surface errors to DEV tools like overlays
    // (https://github.com/facebook/react/issues/21712).
    // As a compromise, rethrow only caught errors in a guard.
    {
      invokeGuardedCallback(null, function () {
        throw error;
      });
      clearCaughtError();
    }
  }

  var callComponentWillUnmountWithTimer = function (current, instance) {
    instance.props = current.memoizedProps;
    instance.state = current.memoizedState;

    if ( current.mode & ProfileMode) {
      try {
        startLayoutEffectTimer();
        instance.componentWillUnmount();
      } finally {
        recordLayoutEffectDuration(current);
      }
    } else {
      instance.componentWillUnmount();
    }
  }; // Capture errors so they don't interrupt mounting.


  function safelyCallCommitHookLayoutEffectListMount(current, nearestMountedAncestor) {
    try {
      commitHookEffectListMount(Layout, current);
    } catch (error) {
      captureCommitPhaseError(current, nearestMountedAncestor, error);
    }
  } // Capture errors so they don't interrupt unmounting.


  function safelyCallComponentWillUnmount(current, nearestMountedAncestor, instance) {
    try {
      callComponentWillUnmountWithTimer(current, instance);
    } catch (error) {
      captureCommitPhaseError(current, nearestMountedAncestor, error);
    }
  } // Capture errors so they don't interrupt mounting.


  function safelyCallComponentDidMount(current, nearestMountedAncestor, instance) {
    try {
      instance.componentDidMount();
    } catch (error) {
      captureCommitPhaseError(current, nearestMountedAncestor, error);
    }
  } // Capture errors so they don't interrupt mounting.
  function safelyAttachRef(current, nearestMountedAncestor) {
    try {
      commitAttachRef(current);
    } catch (error) {
      captureCommitPhaseError(current, nearestMountedAncestor, error);
    }
  }

  function safelyDetachRef(current, nearestMountedAncestor) {
    var ref = current.ref;

    if (ref !== null) {
      if (typeof ref === 'function') {
        var retVal;

        try {
          if (enableProfilerTimer && enableProfilerCommitHooks && current.mode &
ProfileMode) {
            try {
              startLayoutEffectTimer();
              retVal = ref(null);
            } finally {
              recordLayoutEffectDuration(current);
            }
          } else {
            retVal = ref(null);
          }
        } catch (error) {
          captureCommitPhaseError(current, nearestMountedAncestor, error);
        }

        {
          if (typeof retVal === 'function') {
            error('Unexpected return value from a callback ref in %s. ' + 'A callback ref
should not return a function.', getComponentNameFromFiber(current));
          }
        }
      } else {
        ref.current = null;
      }
    }
  }

  function safelyCallDestroy(current, nearestMountedAncestor, destroy) {
    try {
      destroy();
    } catch (error) {
      captureCommitPhaseError(current, nearestMountedAncestor, error);
    }
  }

  var focusedInstanceHandle = null;
  var shouldFireAfterActiveInstanceBlur = false;
  function commitBeforeMutationEffects(root, firstChild) {
    focusedInstanceHandle = prepareForCommit(root.containerInfo);
    nextEffect = firstChild;
    commitBeforeMutationEffects_begin(); // We no longer need to track the active
instance fiber

    var shouldFire = shouldFireAfterActiveInstanceBlur;
    shouldFireAfterActiveInstanceBlur = false;
    focusedInstanceHandle = null;
    return shouldFire;
  }

  function commitBeforeMutationEffects_begin() {
    while (nextEffect !== null) {
      var fiber = nextEffect; // This phase is only used for beforeActiveInstanceBlur.
      var child = fiber.child;

      if ((fiber.subtreeFlags & BeforeMutationMask) !== NoFlags && child !== null) {
        child.return = fiber;
        nextEffect = child;
      } else {
        commitBeforeMutationEffects_complete();
      }
    }
  }

  function commitBeforeMutationEffects_complete() {
    while (nextEffect !== null) {
      var fiber = nextEffect;
      setCurrentFiber(fiber);

      try {
        commitBeforeMutationEffectsOnFiber(fiber);
      } catch (error) {
        captureCommitPhaseError(fiber, fiber.return, error);
      }

      resetCurrentFiber();
      var sibling = fiber.sibling;

      if (sibling !== null) {
        sibling.return = fiber.return;
        nextEffect = sibling;
        return;
      }

      nextEffect = fiber.return;
    }
  }

  function commitBeforeMutationEffectsOnFiber(finishedWork) {
    var current = finishedWork.alternate;
    var flags = finishedWork.flags;

    if ((flags & Snapshot) !== NoFlags) {
      setCurrentFiber(finishedWork);

      switch (finishedWork.tag) {
        case FunctionComponent:
        case ForwardRef:
        case SimpleMemoComponent:
          {
            break;
          }

        case ClassComponent:
          {
            if (current !== null) {
              var prevProps = current.memoizedProps;
              var prevState = current.memoizedState;
              var instance = finishedWork.stateNode; // We could update instance props
and state here,
              // but instead we rely on them being set during last render.
              // TODO: revisit this when we implement resuming.

              {
                if (finishedWork.type === finishedWork.elementType &&
!didWarnAboutReassigningProps) {
                  if (instance.props !== finishedWork.memoizedProps) {
                    error('Expected %s props to match memoized props before ' +
'getSnapshotBeforeUpdate. ' + 'This might either be because of a bug in React, or because
' + 'a component reassigns its own `this.props`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
```

```
                  }

                  if (instance.state !== finishedWork.memoizedState) {
                    error('Expected %s state to match memoized state before ' +
'getSnapshotBeforeUpdate. ' + 'This might either be because of a bug in React, or because
' + 'a component reassigns its own `this.state`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
                  }
                }
              }

              var snapshot = instance.getSnapshotBeforeUpdate(finishedWork.elementType
=== finishedWork.type ? prevProps : resolveDefaultProps(finishedWork.type, prevProps),
prevState);

              {
                var didWarnSet = didWarnAboutUndefinedSnapshotBeforeUpdate;

                if (snapshot === undefined && !didWarnSet.has(finishedWork.type)) {
                  didWarnSet.add(finishedWork.type);

                  error('%s.getSnapshotBeforeUpdate(): A snapshot value (or null) ' +
'must be returned. You have returned undefined.',
getComponentNameFromFiber(finishedWork));
                }
              }

              instance.__reactInternalSnapshotBeforeUpdate = snapshot;
            }

            break;
          }

        case HostRoot:
          {
            {
              var root = finishedWork.stateNode;
              clearContainer(root.containerInfo);
            }

            break;
          }

        case HostComponent:
        case HostText:
        case HostPortal:
        case IncompleteClassComponent:
          // Nothing to do for these component types
          break;

        default:
          {
            throw new Error('This unit of work tag should not have side-effects. This
error is ' + 'likely caused by a bug in React. Please file an issue.');
          }
      }

      resetCurrentFiber();
    }
  }

  function commitHookEffectListUnmount(flags, finishedWork, nearestMountedAncestor) {
    var updateQueue = finishedWork.updateQueue;
    var lastEffect = updateQueue !== null ? updateQueue.lastEffect : null;

    if (lastEffect !== null) {
      var firstEffect = lastEffect.next;
      var effect = firstEffect;
      do {
        if ((effect.tag & flags) === flags) {
          // Unmount
          var destroy = effect.destroy;
          effect.destroy = undefined;

          if (destroy !== undefined) {
            {
              if ((flags & Passive$1) !== NoFlags$1) {
                markComponentPassiveEffectUnmountStarted(finishedWork);
              } else if ((flags & Layout) !== NoFlags$1) {
                markComponentLayoutEffectUnmountStarted(finishedWork);
              }
            }

            {
              if ((flags & Insertion) !== NoFlags$1) {
                setIsRunningInsertionEffect(true);
              }
            }

            safelyCallDestroy(finishedWork, nearestMountedAncestor, destroy);

            {
              if ((flags & Insertion) !== NoFlags$1) {
                setIsRunningInsertionEffect(false);
              }
            }

            {
              if ((flags & Passive$1) !== NoFlags$1) {
                markComponentPassiveEffectUnmountStopped();
              } else if ((flags & Layout) !== NoFlags$1) {
                markComponentLayoutEffectUnmountStopped();
              }
            }
          }
        }

        effect = effect.next;
      } while (effect !== firstEffect);
    }
  }

  function commitHookEffectListMount(flags, finishedWork) {
    var updateQueue = finishedWork.updateQueue;
    var lastEffect = updateQueue !== null ? updateQueue.lastEffect : null;

    if (lastEffect !== null) {
      var firstEffect = lastEffect.next;
      var effect = firstEffect;

      do {
        if ((effect.tag & flags) === flags) {
          {
            if ((flags & Passive$1) !== NoFlags$1) {
              markComponentPassiveEffectMountStarted(finishedWork);
            } else if ((flags & Layout) !== NoFlags$1) {
              markComponentLayoutEffectMountStarted(finishedWork);
            }
          } // Mount


          var create = effect.create;

          {
            if ((flags & Insertion) !== NoFlags$1) {
              setIsRunningInsertionEffect(true);
            }
          }

          effect.destroy = create();

          {
            if ((flags & Insertion) !== NoFlags$1) {
              setIsRunningInsertionEffect(false);
            }
          }

          {
            if ((flags & Passive$1) !== NoFlags$1) {
              markComponentPassiveEffectMountStopped();
            } else if ((flags & Layout) !== NoFlags$1) {
              markComponentLayoutEffectMountStopped();
            }
          }

          {
            var destroy = effect.destroy;

            if (destroy !== undefined && typeof destroy !== 'function') {
              var hookName = void 0;

              if ((effect.tag & Layout) !== NoFlags) {
                hookName = 'useLayoutEffect';
              } else if ((effect.tag & Insertion) !== NoFlags) {
                hookName = 'useInsertionEffect';
              } else {
                hookName = 'useEffect';
              }

              var addendum = void 0;

              if (destroy === null) {
                addendum = ' You returned null. If your effect does not require clean ' +
'up, return undefined (or nothing).';
              } else if (typeof destroy.then === 'function') {
                addendum = '\n\nIt looks like you wrote ' + hookName + '(async () => ...)
or returned a Promise. ' + 'Instead, write the async function inside your effect ' + 'and
call it immediately:\n\n' + hookName + '(() => {\n' + '  async function fetchData() {\n'
+ '    // You can await here\n' + '    const response = await MyAPI.getData(someId);\n' +
'    // ...\n' + '  }\n' + '  fetchData();\n' + "}, [someId]); // Or [] if effect doesn't
need props or state\n\n" + 'Learn more about data fetching with Hooks:
https://reactjs.org/link/hooks-data-fetching';
              } else {
                addendum = ' You returned: ' + destroy;
              }

              error('%s must not return anything besides a function, ' + 'which is used
for clean-up.%s', hookName, addendum);
            }
          }
        }

        effect = effect.next;
      } while (effect !== firstEffect);
    }
  }

  function commitPassiveEffectDurations(finishedRoot, finishedWork) {
    {
      // Only Profilers with work in their subtree will have an Update effect scheduled.
      if ((finishedWork.flags & Update) !== NoFlags) {
        switch (finishedWork.tag) {
          case Profiler:
            {
              var passiveEffectDuration = finishedWork.stateNode.passiveEffectDuration;
              var _finishedWork$memoize = finishedWork.memoizedProps,
                  id = _finishedWork$memoize.id,
                  onPostCommit = _finishedWork$memoize.onPostCommit; // This value will
still reflect the previous commit phase.
              // It does not get reset until the start of the next commit phase.

              var commitTime = getCommitTime();
              var phase = finishedWork.alternate === null ? 'mount' : 'update';

              {
                if (isCurrentUpdateNested()) {
                  phase = 'nested-update';
                }
              }

              if (typeof onPostCommit === 'function') {
                onPostCommit(id, phase, passiveEffectDuration, commitTime);
              } // Bubble times to the next nearest ancestor Profiler.
              // After we process that Profiler, we'll bubble further up.


              var parentFiber = finishedWork.return;

              outer: while (parentFiber !== null) {
                switch (parentFiber.tag) {
                  case HostRoot:
                    var root = parentFiber.stateNode;
                    root.passiveEffectDuration += passiveEffectDuration;
                    break outer;

                  case Profiler:
                    var parentStateNode = parentFiber.stateNode;
                    parentStateNode.passiveEffectDuration += passiveEffectDuration;
                    break outer;
                }

                parentFiber = parentFiber.return;
              }

              break;
            }
        }
      }
    }
  }

  function commitLayoutEffectOnFiber(finishedRoot, current, finishedWork, committedLanes)
{
    if ((finishedWork.flags & LayoutMask) !== NoFlags) {
      switch (finishedWork.tag) {
        case FunctionComponent:
        case ForwardRef:
        case SimpleMemoComponent:
          {
            if ( !offscreenSubtreeWasHidden) {
              // At this point layout effects have already been destroyed (during
mutation phase).
              // This is done to prevent sibling component effects from interfering with
each other,
              // e.g. a destroy function in one component should never override a ref set
              // by a create function in another component during the same commit.
              if ( finishedWork.mode & ProfileMode) {
                try {
                  startLayoutEffectTimer();
                  commitHookEffectListMount(Layout | HasEffect, finishedWork);
                } finally {
                  recordLayoutEffectDuration(finishedWork);
                }
              } else {
                commitHookEffectListMount(Layout | HasEffect, finishedWork);
              }
            }

            break;
          }

        case ClassComponent:
          {
            var instance = finishedWork.stateNode;

            if (finishedWork.flags & Update) {
              if (!offscreenSubtreeWasHidden) {
                if (current === null) {
                  // We could update instance props and state here,
                  // but instead we rely on them being set during last render.
                  // TODO: revisit this when we implement resuming.
                  {
                    if (finishedWork.type === finishedWork.elementType &&
!didWarnAboutReassigningProps) {
                      if (instance.props !== finishedWork.memoizedProps) {
                        error('Expected %s props to match memoized props before ' +
'componentDidMount. ' + 'This might either be because of a bug in React, or because ' +
'a component reassigns its own `this.props`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
                      }

                      if (instance.state !== finishedWork.memoizedState) {
                        error('Expected %s state to match memoized state before ' +
'componentDidMount. ' + 'This might either be because of a bug in React, or because ' +
'a component reassigns its own `this.state`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
                      }
                    }
                  }

                  if ( finishedWork.mode & ProfileMode) {
                    try {
                      startLayoutEffectTimer();
                      instance.componentDidMount();
                    } finally {
                      recordLayoutEffectDuration(finishedWork);
                    }
                  } else {
                    instance.componentDidMount();
                  }
                } else {
                  var prevProps = finishedWork.elementType === finishedWork.type ?
current.memoizedProps : resolveDefaultProps(finishedWork.type, current.memoizedProps);
                  var prevState = current.memoizedState; // We could update instance
props and state here,
                  // but instead we rely on them being set during last render.
                  // TODO: revisit this when we implement resuming.

                  {
                    if (finishedWork.type === finishedWork.elementType &&
!didWarnAboutReassigningProps) {
                      if (instance.props !== finishedWork.memoizedProps) {
                        error('Expected %s props to match memoized props before ' +
'componentDidUpdate. ' + 'This might either be because of a bug in React, or because ' +
'a component reassigns its own `this.props`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
                      }

                      if (instance.state !== finishedWork.memoizedState) {
                        error('Expected %s state to match memoized state before ' +
'componentDidUpdate. ' + 'This might either be because of a bug in React, or because ' +
'a component reassigns its own `this.state`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
                      }
                    }
                  }

                  if ( finishedWork.mode & ProfileMode) {
                    try {
                      startLayoutEffectTimer();
                      instance.componentDidUpdate(prevProps, prevState,
instance.__reactInternalSnapshotBeforeUpdate);
                    } finally {
                      recordLayoutEffectDuration(finishedWork);
                    }
                  } else {
                    instance.componentDidUpdate(prevProps, prevState,
instance.__reactInternalSnapshotBeforeUpdate);
                  }
                }
              }
            } // TODO: I think this is now always non-null by the time it reaches the
            // commit phase. Consider removing the type check.


            var updateQueue = finishedWork.updateQueue;

            if (updateQueue !== null) {
              {
                if (finishedWork.type === finishedWork.elementType &&
!didWarnAboutReassigningProps) {
                  if (instance.props !== finishedWork.memoizedProps) {
                    error('Expected %s props to match memoized props before ' +
'processing the update queue. ' + 'This might either be because of a bug in React, or
because ' + 'a component reassigns its own `this.props`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
                  }

                  if (instance.state !== finishedWork.memoizedState) {
                    error('Expected %s state to match memoized state before ' +
'processing the update queue. ' + 'This might either be because of a bug in React, or
because ' + 'a component reassigns its own `this.state`. ' + 'Please file an issue.',
getComponentNameFromFiber(finishedWork) || 'instance');
                  }
                }
              } // We could update instance props and state here,
              // but instead we rely on them being set during last render.
              // TODO: revisit this when we implement resuming.


              commitUpdateQueue(finishedWork, updateQueue, instance);
            }

            break;
          }

        case HostRoot:
          {
            // TODO: I think this is now always non-null by the time it reaches the
            // commit phase. Consider removing the type check.
            var _updateQueue = finishedWork.updateQueue;

            if (_updateQueue !== null) {
              var _instance = null;

              if (finishedWork.child !== null) {
                switch (finishedWork.child.tag) {
                  case HostComponent:
                    _instance = getPublicInstance(finishedWork.child.stateNode);
                    break;

                  case ClassComponent:
                    _instance = finishedWork.child.stateNode;
                  break;
                }
              }

              commitUpdateQueue(finishedWork, _updateQueue, _instance);
            }

            break;
          }

        case HostComponent:
          {
            var _instance2 = finishedWork.stateNode; // Renderers may schedule work to be
done after host components are mounted
            // (eg DOM renderer may schedule auto-focus for inputs and form controls).
            // These effects should only be committed when components are first mounted,
            // aka when there is no current/alternate.

            if (current === null && finishedWork.flags & Update) {
              var type = finishedWork.type;
              var props = finishedWork.memoizedProps;
              commitMount(_instance2, type, props);
            }

            break;
          }

        case HostText:
          {
            // We have no life-cycles associated with text.
            break;
          }

        case HostPortal:
          {
            // We have no life-cycles associated with portals.
            break;
          }

        case Profiler:
          {
            {
              var _finishedWork$memoize2 = finishedWork.memoizedProps,
                  onCommit = _finishedWork$memoize2.onCommit,
                  onRender = _finishedWork$memoize2.onRender;
              var effectDuration = finishedWork.stateNode.effectDuration;
              var commitTime = getCommitTime();
              var phase = current === null ? 'mount' : 'update';

              {
                if (isCurrentUpdateNested()) {
                  phase = 'nested-update';
                }
              }

              if (typeof onRender === 'function') {
                onRender(finishedWork.memoizedProps.id, phase,
finishedWork.actualDuration, finishedWork.treeBaseDuration, finishedWork.actualStartTime,
commitTime);
              }

              {
                if (typeof onCommit === 'function') {
                  onCommit(finishedWork.memoizedProps.id, phase, effectDuration,
commitTime);
                } // Schedule a passive effect for this Profiler to call onPostCommit
hooks.
                // This effect should be scheduled even if there is no onPostCommit
callback for this Profiler,
                // because the effect is also where times bubble to parent Profilers.


                enqueuePendingPassiveProfilerEffect(finishedWork); // Propagate layout
effect durations to the next nearest Profiler ancestor.
                // Do not reset these values until the next render so DevTools has a
chance to read them first.

                var parentFiber = finishedWork.return;

                outer: while (parentFiber !== null) {
                  switch (parentFiber.tag) {
                    case HostRoot:
                      var root = parentFiber.stateNode;
                      root.effectDuration += effectDuration;
                      break outer;

                    case Profiler:
                      var parentStateNode = parentFiber.stateNode;
                      parentStateNode.effectDuration += effectDuration;
                      break outer;
                  }

                  parentFiber = parentFiber.return;
                }
              }
            }

            break;
          }

        case SuspenseComponent:
          {
            commitSuspenseHydrationCallbacks(finishedRoot, finishedWork);
            break;
          }

        case SuspenseListComponent:
        case IncompleteClassComponent:
        case ScopeComponent:
        case OffscreenComponent:
        case LegacyHiddenComponent:
        case TracingMarkerComponent:
          {
            break;
          }

        default:
          throw new Error('This unit of work tag should not have side-effects. This error
is ' + 'likely caused by a bug in React. Please file an issue.');
      }
    }

    if ( !offscreenSubtreeWasHidden) {
      {
        if (finishedWork.flags & Ref) {
          commitAttachRef(finishedWork);
        }
      }
    }
  }

  function reappearLayoutEffectsOnFiber(node) {
    // Turn on layout effects in a tree that previously disappeared.
    // TODO (Offscreen) Check: flags & LayoutStatic
    switch (node.tag) {
      case FunctionComponent:
      case ForwardRef:
      case SimpleMemoComponent:
        {
          if ( node.mode & ProfileMode) {
            try {
              startLayoutEffectTimer();
              safelyCallCommitHookLayoutEffectListMount(node, node.return);
            } finally {
              recordLayoutEffectDuration(node);
            }
          } else {
            safelyCallCommitHookLayoutEffectListMount(node, node.return);
          }

          break;
        }

      case ClassComponent:
        {
          var instance = node.stateNode;

          if (typeof instance.componentDidMount === 'function') {
            safelyCallComponentDidMount(node, node.return, instance);
          }

          safelyAttachRef(node, node.return);
          break;
        }

      case HostComponent:
        {
          safelyAttachRef(node, node.return);
          break;
        }
    }
  }

  function hideOrUnhideAllChildren(finishedWork, isHidden) {
    // Only hide or unhide the top-most host nodes.
    var hostSubtreeRoot = null;

    {
      // We only have the top Fiber that was inserted but we need to recurse down its
      // children to find all the terminal nodes.
      var node = finishedWork;

      while (true) {
        if (node.tag === HostComponent) {
          if (hostSubtreeRoot === null) {
            hostSubtreeRoot = node;

            try {
              var instance = node.stateNode;

              if (isHidden) {
                hideInstance(instance);
              } else {
                unhideInstance(node.stateNode, node.memoizedProps);
              }
            } catch (error) {
              captureCommitPhaseError(finishedWork, finishedWork.return, error);
            }
          }
        } else if (node.tag === HostText) {
          if (hostSubtreeRoot === null) {
            try {
              var _instance3 = node.stateNode;

              if (isHidden) {
                hideTextInstance(_instance3);
              } else {
```

```
              unhideTextInstance(_instance3, node.memoizedProps);
            }
          } catch (error) {
            captureCommitPhaseError(finishedWork, finishedWork.return, error);
          }
        }
      } else if ((node.tag === OffscreenComponent || node.tag ===
LegacyHiddenComponent) && node.memoizedState !== null && node !== finishedWork) ; else if
(node.child !== null) {
        node.child.return = node;
        node = node.child;
        continue;
      }

      if (node === finishedWork) {
        return;
      }

      while (node.sibling === null) {
        if (node.return === null || node.return === finishedWork) {
          return;
        }

        if (hostSubtreeRoot === node) {
          hostSubtreeRoot = null;
        }

        node = node.return;
      }

      if (hostSubtreeRoot === node) {
        hostSubtreeRoot = null;
      }

      node.sibling.return = node.return;
      node = node.sibling;
    }
  }
}

function commitAttachRef(finishedWork) {
  var ref = finishedWork.ref;

  if (ref !== null) {
    var instance = finishedWork.stateNode;
    var instanceToUse;

    switch (finishedWork.tag) {
      case HostComponent:
        instanceToUse = getPublicInstance(instance);
        break;

      default:
        instanceToUse = instance;
    } // Moved outside to ensure DCE works with this flag

    if (typeof ref === 'function') {
      var retVal;

      if ( finishedWork.mode & ProfileMode) {
        try {
          startLayoutEffectTimer();
          retVal = ref(instanceToUse);
        } finally {
          recordLayoutEffectDuration(finishedWork);
        }
      } else {
        retVal = ref(instanceToUse);
      }
```

```
        {
          if (typeof retVal === 'function') {
            error('Unexpected return value from a callback ref in %s. ' + 'A callback ref
should not return a function.', getComponentNameFromFiber(finishedWork));
          }
        }
      } else {
        {
          if (!ref.hasOwnProperty('current')) {
            error('Unexpected ref object provided for %s. ' + 'Use either a ref-setter
function or React.createRef().', getComponentNameFromFiber(finishedWork));
          }
        }

        ref.current = instanceToUse;
      }
    }
  }

  function detachFiberMutation(fiber) {
    // Cut off the return pointer to disconnect it from the tree.
    // This enables us to detect and warn against state updates on an unmounted
component.
    // It also prevents events from bubbling from within disconnected components.
    //
    // Ideally, we should also clear the child pointer of the parent alternate to let
this
    // get GC:ed but we don't know which for sure which parent is the current
    // one so we'll settle for GC:ing the subtree of this child.
    // This child itself will be GC:ed when the parent updates the next time.
    //
    // Note that we can't clear child or sibling pointers yet.
    // They're needed for passive effects and for findDOMNode.
    // We defer those fields, and all other cleanup, to the passive phase (see
detachFiberAfterEffects).
    //
    // Don't reset the alternate yet, either. We need that so we can detach the
    // alternate's fields in the passive phase. Clearing the return pointer is
    // sufficient for findDOMNode semantics.
    var alternate = fiber.alternate;

    if (alternate !== null) {
      alternate.return = null;
    }

    fiber.return = null;
  }

  function detachFiberAfterEffects(fiber) {
    var alternate = fiber.alternate;

    if (alternate !== null) {
      fiber.alternate = null;
      detachFiberAfterEffects(alternate);
    } // Note: Defensively using negation instead of < in case
    // `deletedTreeCleanUpLevel` is undefined.


    {
      // Clear cyclical Fiber fields. This level alone is designed to roughly
      // approximate the planned Fiber refactor. In that world, `setState` will be
      // bound to a special "instance" object instead of a Fiber. The Instance
      // object will not have any of these fields. It will only be connected to
      // the fiber tree via a single link at the root. So if this level alone is
      // sufficient to fix memory issues, that bodes well for our plans.
      fiber.child = null;
      fiber.deletions = null;
      fiber.sibling = null; // The `stateNode` is cyclical because on host nodes it
```

```
points to the host
      // tree, which has its own pointers to children, parents, and siblings.
      // The other host nodes also point back to fibers, so we should detach that
      // one, too.

      if (fiber.tag === HostComponent) {
        var hostInstance = fiber.stateNode;

        if (hostInstance !== null) {
          detachDeletedInstance(hostInstance);
        }
      }

      fiber.stateNode = null; // I'm intentionally not clearing the `return` field in
this level. We
      // already disconnect the `return` pointer at the root of the deleted
      // subtree (in `detachFiberMutation`). Besides, `return` by itself is not
      // cyclical — it's only cyclical when combined with `child`, `sibling`, and
      // `alternate`. But we'll clear it in the next level anyway, just in case.

      {
        fiber._debugOwner = null;
      }

      {
        // Theoretically, nothing in here should be necessary, because we already
        // disconnected the fiber from the tree. So even if something leaks this
        // particular fiber, it won't leak anything else
        //
        // The purpose of this branch is to be super aggressive so we can measure
        // if there's any difference in memory impact. If there is, that could
        // indicate a React leak we don't know about.
        fiber.return = null;
        fiber.dependencies = null;
        fiber.memoizedProps = null;
        fiber.memoizedState = null;
        fiber.pendingProps = null;
        fiber.stateNode = null; // TODO: Move to
`commitPassiveUnmountInsideDeletedTreeOnFiber` instead.

        fiber.updateQueue = null;
      }
    }
  }

  function getHostParentFiber(fiber) {
    var parent = fiber.return;

    while (parent !== null) {
      if (isHostParent(parent)) {
        return parent;
      }

      parent = parent.return;
    }

    throw new Error('Expected to find a host parent. This error is likely caused by a bug
' + 'in React. Please file an issue.');
  }

  function isHostParent(fiber) {
    return fiber.tag === HostComponent || fiber.tag === HostRoot || fiber.tag ===
HostPortal;
  }

  function getHostSibling(fiber) {
    // We're going to search forward into the tree until we find a sibling host
    // node. Unfortunately, if multiple insertions are done in a row we have to
    // search past them. This leads to exponential search for the next sibling.
    // TODO: Find a more efficient way to do this.
    var node = fiber;

    siblings: while (true) {
      // If we didn't find anything, let's try the next sibling.
      while (node.sibling === null) {
        if (node.return === null || isHostParent(node.return)) {
          // If we pop out of the root or hit the parent the fiber we are the
          // last sibling.
          return null;
        }

        node = node.return;
      }

      node.sibling.return = node.return;
      node = node.sibling;

      while (node.tag !== HostComponent && node.tag !== HostText && node.tag !==
DehydratedFragment) {
        // If it is not host node and, we might have a host node inside it.
        // Try to search down until we find one.
        if (node.flags & Placement) {
          // If we don't have a child, try the siblings instead.
          continue siblings;
        } // If we don't have a child, try the siblings instead.
        // We also skip portals because they are not part of this host tree.


        if (node.child === null || node.tag === HostPortal) {
          continue siblings;
        } else {
          node.child.return = node;
          node = node.child;
        }
      } // Check if this host node is stable or about to be placed.


      if (!(node.flags & Placement)) {
        // Found it!
        return node.stateNode;
      }
    }
  }

  function commitPlacement(finishedWork) {


    var parentFiber = getHostParentFiber(finishedWork); // Note: these two variables
*must* always be updated together.

    switch (parentFiber.tag) {
      case HostComponent:
        {
          var parent = parentFiber.stateNode;

          if (parentFiber.flags & ContentReset) {
            // Reset the text content of the parent before doing any insertions
            resetTextContent(parent); // Clear ContentReset from the effect tag

            parentFiber.flags &= ~ContentReset;
          }

          var before = getHostSibling(finishedWork); // We only have the top Fiber that
was inserted but we need to recurse down its
          // children to find all the terminal nodes.

          insertOrAppendPlacementNode(finishedWork, before, parent);
          break;
        }

      case HostRoot:
      case HostPortal:
        {
          var _parent = parentFiber.stateNode.containerInfo;

          var _before = getHostSibling(finishedWork);

          insertOrAppendPlacementNodeIntoContainer(finishedWork, _before, _parent);
          break;
        }
      // eslint-disable-next-line-no-fallthrough

      default:
        throw new Error('Invalid host parent fiber. This error is likely caused by a bug
' + 'in React. Please file an issue.');
    }
  }

  function insertOrAppendPlacementNodeIntoContainer(node, before, parent) {
    var tag = node.tag;
    var isHost = tag === HostComponent || tag === HostText;

    if (isHost) {
      var stateNode = node.stateNode;

      if (before) {
        insertInContainerBefore(parent, stateNode, before);
      } else {
        appendChildToContainer(parent, stateNode);
      }
    } else if (tag === HostPortal) ; else {
      var child = node.child;

      if (child !== null) {
        insertOrAppendPlacementNodeIntoContainer(child, before, parent);
        var sibling = child.sibling;

        while (sibling !== null) {
          insertOrAppendPlacementNodeIntoContainer(sibling, before, parent);
          sibling = sibling.sibling;
        }
      }
    }
  }

  function insertOrAppendPlacementNode(node, before, parent) {
    var tag = node.tag;
    var isHost = tag === HostComponent || tag === HostText;

    if (isHost) {
      var stateNode = node.stateNode;

      if (before) {
        insertBefore(parent, stateNode, before);
      } else {
        appendChild(parent, stateNode);
      }
    } else if (tag === HostPortal) ; else {
      var child = node.child;

      if (child !== null) {
        insertOrAppendPlacementNode(child, before, parent);
        var sibling = child.sibling;

        while (sibling !== null) {
          insertOrAppendPlacementNode(sibling, before, parent);
          sibling = sibling.sibling;
        }
      }
    }
  } // These are tracked on the stack as we recursively traverse a
  // deleted subtree.
  // TODO: Update these during the whole mutation phase, not just during
  // a deletion.


  var hostParent = null;
  var hostParentIsContainer = false;

  function commitDeletionEffects(root, returnFiber, deletedFiber) {
    {
      // We only have the top Fiber that was deleted but we need to recurse down its
      // children to find all the terminal nodes.
      // Recursively delete all host nodes from the parent, detach refs, clean
      // up mounted layout effects, and call componentWillUnmount.
      // We only need to remove the topmost host child in each branch. But then we
      // still need to keep traversing to unmount effects, refs, and cWU. TODO: We
      // could split this into two separate traversals functions, where the second
      // one doesn't include any removeChild logic. This is maybe the same
      // function as "disappearLayoutEffects" (or whatever that turns into after
      // the layout phase is refactored to use recursion).
      // Before starting, find the nearest host parent on the stack so we know
      // which instance/container to remove the children from.
      // TODO: Instead of searching up the fiber return path on every deletion, we
      // can track the nearest host component on the JS stack as we traverse the
      // tree during the commit phase. This would make insertions faster, too.
      var parent = returnFiber;

      findParent: while (parent !== null) {
        switch (parent.tag) {
          case HostComponent:
            {
              hostParent = parent.stateNode;
              hostParentIsContainer = false;
              break findParent;
            }

          case HostRoot:
            {
              hostParent = parent.stateNode.containerInfo;
              hostParentIsContainer = true;
              break findParent;
            }

          case HostPortal:
            {
              hostParent = parent.stateNode.containerInfo;
              hostParentIsContainer = true;
              break findParent;
            }
        }

        parent = parent.return;
      }

      if (hostParent === null) {
        throw new Error('Expected to find a host parent. This error is likely caused by '
+ 'a bug in React. Please file an issue.');
      }

      commitDeletionEffectsOnFiber(root, returnFiber, deletedFiber);
      hostParent = null;
      hostParentIsContainer = false;
    }

    detachFiberMutation(deletedFiber);
  }

  function recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
parent) {
    // TODO: Use a static flag to skip trees that don't have unmount effects
    var child = parent.child;

    while (child !== null) {
      commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor, child);
      child = child.sibling;
    }
  }

  function commitDeletionEffectsOnFiber(finishedRoot, nearestMountedAncestor,
deletedFiber) {
    onCommitUnmount(deletedFiber); // The cases in this outer switch modify the stack
before they traverse
    // into their subtree. There are simpler cases in the inner switch
    // that don't modify the stack.

    switch (deletedFiber.tag) {
      case HostComponent:
        {
          if (!offscreenSubtreeWasHidden) {
            safelyDetachRef(deletedFiber, nearestMountedAncestor);
          } // Intentional fallthrough to next branch

        }
      // eslint-disable-next-line-no-fallthrough

      case HostText:
        {
          // We only need to remove the nearest host child. Set the host parent
          // to `null` on the stack to indicate that nested children don't
          // need to be removed.
          {
            var prevHostParent = hostParent;
            var prevHostParentIsContainer = hostParentIsContainer;
            hostParent = null;
            recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
            hostParent = prevHostParent;
            hostParentIsContainer = prevHostParentIsContainer;

            if (hostParent !== null) {
              // Now that all the child effects have unmounted, we can remove the
              // node from the tree.
              if (hostParentIsContainer) {
                removeChildFromContainer(hostParent, deletedFiber.stateNode);
              } else {
                removeChild(hostParent, deletedFiber.stateNode);
              }
            }
          }

          return;
        }

      case DehydratedFragment:
        {
          // Delete the dehydrated suspense boundary and all of its content.


          {
            if (hostParent !== null) {
              if (hostParentIsContainer) {
                clearSuspenseBoundaryFromContainer(hostParent, deletedFiber.stateNode);
              } else {
                clearSuspenseBoundary(hostParent, deletedFiber.stateNode);
              }
            }
          }

          return;
        }

      case HostPortal:
        {
          {
            // When we go into a portal, it becomes the parent to remove from.
            var _prevHostParent = hostParent;
            var _prevHostParentIsContainer = hostParentIsContainer;
            hostParent = deletedFiber.stateNode.containerInfo;
            hostParentIsContainer = true;
            recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
            hostParent = _prevHostParent;
            hostParentIsContainer = _prevHostParentIsContainer;
          }

          return;
        }

      case FunctionComponent:
      case ForwardRef:
      case MemoComponent:
      case SimpleMemoComponent:
        {
          if (!offscreenSubtreeWasHidden) {
            var updateQueue = deletedFiber.updateQueue;

            if (updateQueue !== null) {
              var lastEffect = updateQueue.lastEffect;

              if (lastEffect !== null) {
                var firstEffect = lastEffect.next;
                var effect = firstEffect;

                do {
                  var _effect = effect,
                      destroy = _effect.destroy,
                      tag = _effect.tag;

                  if (destroy !== undefined) {
                    if ((tag & Insertion) !== NoFlags$1) {
                      safelyCallDestroy(deletedFiber, nearestMountedAncestor, destroy);
                    } else if ((tag & Layout) !== NoFlags$1) {
                      {
                        markComponentLayoutEffectUnmountStarted(deletedFiber);
                      }

                      if ( deletedFiber.mode & ProfileMode) {
                        startLayoutEffectTimer();
                        safelyCallDestroy(deletedFiber, nearestMountedAncestor, destroy);
                        recordLayoutEffectDuration(deletedFiber);
                      } else {
                        safelyCallDestroy(deletedFiber, nearestMountedAncestor, destroy);
                      }

                      {
                        markComponentLayoutEffectUnmountStopped();
                      }
                    }
                  }

                  effect = effect.next;
                } while (effect !== firstEffect);
              }
            }
          }

          recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
          return;
        }

      case ClassComponent:
        {
          if (!offscreenSubtreeWasHidden) {
            safelyDetachRef(deletedFiber, nearestMountedAncestor);
            var instance = deletedFiber.stateNode;

            if (typeof instance.componentWillUnmount === 'function') {
              safelyCallComponentWillUnmount(deletedFiber, nearestMountedAncestor,
instance);
            }
          }

          recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
          return;
        }

      case ScopeComponent:
        {

          recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
          return;
        }

      case OffscreenComponent:
        {
          if ( // TODO: Remove this dead flag
           deletedFiber.mode & ConcurrentMode) {
            // If this offscreen component is hidden, we already unmounted it. Before
            // deleting the children, track that it's already unmounted so that we
            // don't attempt to unmount the effects again.
            // TODO: If the tree is hidden, in most cases we should be able to skip
            // over the nested children entirely. An exception is we haven't yet found
            // the topmost host node to delete, which we already track on the stack.
            // But the other case is portals, which need to be detached no matter how
            // deeply they are nested. We should use a subtree flag to track whether a
            // subtree includes a nested portal.
            var prevOffscreenSubtreeWasHidden = offscreenSubtreeWasHidden;
            offscreenSubtreeWasHidden = prevOffscreenSubtreeWasHidden ||
deletedFiber.memoizedState !== null;
            recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
            offscreenSubtreeWasHidden = prevOffscreenSubtreeWasHidden;
          } else {
            recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
          }

          break;
        }

      default:
        {
          recursivelyTraverseDeletionEffects(finishedRoot, nearestMountedAncestor,
deletedFiber);
          return;
        }
    }
  }
```

```
    function commitSuspenseCallback(finishedWork) {
      // TODO: Move this to passive phase
      var newState = finishedWork.memoizedState;
    }

    function commitSuspenseHydrationCallbacks(finishedRoot, finishedWork) {

      var newState = finishedWork.memoizedState;

      if (newState === null) {
        var current = finishedWork.alternate;

        if (current !== null) {
          var prevState = current.memoizedState;

          if (prevState !== null) {
            var suspenseInstance = prevState.dehydrated;

            if (suspenseInstance !== null) {
              commitHydratedSuspenseInstance(suspenseInstance);
            }
          }
        }
      }
    }

    function attachSuspenseRetryListeners(finishedWork) {
      // If this boundary just timed out, then it will have a set of wakeables.
      // For each wakeable, attach a listener so that when it resolves, React
      // attempts to re-render the boundary in the primary (pre-timeout) state.
      var wakeables = finishedWork.updateQueue;

      if (wakeables !== null) {
        finishedWork.updateQueue = null;
        var retryCache = finishedWork.stateNode;

        if (retryCache === null) {
          retryCache = finishedWork.stateNode = new PossiblyWeakSet();
        }

        wakeables.forEach(function (wakeable) {
          // Memoize using the boundary fiber to prevent redundant listeners.
          var retry = resolveRetryWakeable.bind(null, finishedWork, wakeable);

          if (!retryCache.has(wakeable)) {
            retryCache.add(wakeable);

            {
              if (isDevToolsPresent) {
                if (inProgressLanes !== null && inProgressRoot !== null) {
                  // If we have pending work still, associate the original updaters with
  it.
                  restorePendingUpdaters(inProgressRoot, inProgressLanes);
                } else {
                  throw Error('Expected finished root and lanes to be set. This is a bug in
  React.');
                }
              }
            }

            wakeable.then(retry, retry);
          }
        });
      }
    } // This function detects when a Suspense boundary goes from visible to hidden.
    function commitMutationEffects(root, finishedWork, committedLanes) {
      inProgressLanes = committedLanes;
      inProgressRoot = root;
      setCurrentFiber(finishedWork);
      commitMutationEffectsOnFiber(finishedWork, root);
      setCurrentFiber(finishedWork);
      inProgressLanes = null;
      inProgressRoot = null;
    }

    function recursivelyTraverseMutationEffects(root, parentFiber, lanes) {
      // Deletions effects can be scheduled on any fiber type. They need to happen
      // before the children effects hae fired.
      var deletions = parentFiber.deletions;

      if (deletions !== null) {
        for (var i = 0; i < deletions.length; i++) {
          var childToDelete = deletions[i];

          try {
            commitDeletionEffects(root, parentFiber, childToDelete);
          } catch (error) {
            captureCommitPhaseError(childToDelete, parentFiber, error);
          }
        }
      }

      var prevDebugFiber = getCurrentFiber();

      if (parentFiber.subtreeFlags & MutationMask) {
        var child = parentFiber.child;

        while (child !== null) {
          setCurrentFiber(child);
          commitMutationEffectsOnFiber(child, root);
          child = child.sibling;
        }
      }

      setCurrentFiber(prevDebugFiber);
    }

    function commitMutationEffectsOnFiber(finishedWork, root, lanes) {
      var current = finishedWork.alternate;
      var flags = finishedWork.flags; // The effect flag should be checked *after* we
  refine the type of fiber,
      // because the fiber tag is more specific. An exception is any flag related
      // to reconcilation, because those can be set on all fiber types.

      switch (finishedWork.tag) {
        case FunctionComponent:
        case ForwardRef:
        case MemoComponent:
        case SimpleMemoComponent:
          {
            recursivelyTraverseMutationEffects(root, finishedWork);
            commitReconciliationEffects(finishedWork);

            if (flags & Update) {
              try {
                commitHookEffectListUnmount(Insertion | HasEffect, finishedWork,
  finishedWork.return);
                commitHookEffectListMount(Insertion | HasEffect, finishedWork);
              } catch (error) {
                captureCommitPhaseError(finishedWork, finishedWork.return, error);
              } // Layout effects are destroyed during the mutation phase so that all
              // destroy functions for all fibers are called before any create functions.
              // This prevents sibling component effects from interfering with each other,
              // e.g. a destroy function in one component should never override a ref set
              // by a create function in another component during the same commit.


              if ( finishedWork.mode & ProfileMode) {
```

```
              try {
                startLayoutEffectTimer();
                commitHookEffectListUnmount(Layout | HasEffect, finishedWork,
  finishedWork.return);
              } catch (error) {
                captureCommitPhaseError(finishedWork, finishedWork.return, error);
              }

              recordLayoutEffectDuration(finishedWork);
            } else {
              try {
                commitHookEffectListUnmount(Layout | HasEffect, finishedWork,
  finishedWork.return);
              } catch (error) {
                captureCommitPhaseError(finishedWork, finishedWork.return, error);
              }
            }
          }

          return;
        }

      case ClassComponent:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);

          if (flags & Ref) {
            if (current !== null) {
              safelyDetachRef(current, current.return);
            }
          }

          return;
        }

      case HostComponent:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);

          if (flags & Ref) {
            if (current !== null) {
              safelyDetachRef(current, current.return);
            }
          }

          {
            // TODO: ContentReset gets cleared by the children during the commit
            // phase. This is a refactor hazard because it means we must read
            // flags the flags after `commitReconciliationEffects` has already run;
            // the order matters. We should refactor so that ContentReset does not
            // rely on mutating the flag during commit. Like by setting a flag
            // during the render phase instead.
            if (finishedWork.flags & ContentReset) {
              var instance = finishedWork.stateNode;

              try {
                resetTextContent(instance);
              } catch (error) {
                captureCommitPhaseError(finishedWork, finishedWork.return, error);
              }
            }

            if (flags & Update) {
              var _instance4 = finishedWork.stateNode;

              if (_instance4 != null) {
                // Commit the work prepared earlier.
                var newProps = finishedWork.memoizedProps; // For hydration we reuse the
update path but we treat the oldProps
                // as the newProps. The updatePayload will contain the real change in
                // this case.

                var oldProps = current !== null ? current.memoizedProps : newProps;
                var type = finishedWork.type; // TODO: Type the updateQueue to be
specific to host components.

                var updatePayload = finishedWork.updateQueue;
                finishedWork.updateQueue = null;

                if (updatePayload !== null) {
                  try {
                    commitUpdate(_instance4, updatePayload, type, oldProps, newProps,
finishedWork);
                  } catch (error) {
                    captureCommitPhaseError(finishedWork, finishedWork.return, error);
                  }
                }
              }
            }
          }

          return;
        }

      case HostText:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);

          if (flags & Update) {
            {
              if (finishedWork.stateNode === null) {
                throw new Error('This should have a text node initialized. This error is
likely ' + 'caused by a bug in React. Please file an issue.');
              }

              var textInstance = finishedWork.stateNode;
              var newText = finishedWork.memoizedProps; // For hydration we reuse the
update path but we treat the oldProps
              // as the newProps. The updatePayload will contain the real change in
              // this case.

              var oldText = current !== null ? current.memoizedProps : newText;

              try {
                commitTextUpdate(textInstance, oldText, newText);
              } catch (error) {
                captureCommitPhaseError(finishedWork, finishedWork.return, error);
              }
            }
          }

          return;
        }

      case HostRoot:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);

          if (flags & Update) {
            {
              if (current !== null) {
                var prevRootState = current.memoizedState;

                if (prevRootState.isDehydrated) {
                  try {
                    commitHydratedContainer(root.containerInfo);
                  } catch (error) {
                    captureCommitPhaseError(finishedWork, finishedWork.return, error);
                  }
                }
              }
            }
          }

          return;
        }

      case HostPortal:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);

          return;
        }

      case SuspenseComponent:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);
          var offscreenFiber = finishedWork.child;

          if (offscreenFiber.flags & Visibility) {
            var offscreenInstance = offscreenFiber.stateNode;
            var newState = offscreenFiber.memoizedState;
            var isHidden = newState !== null; // Track the current state on the Offscreen
instance so we can
            // read it during an event

            offscreenInstance.isHidden = isHidden;

            if (isHidden) {
              var wasHidden = offscreenFiber.alternate !== null &&
offscreenFiber.alternate.memoizedState !== null;

              if (!wasHidden) {
                // TODO: Move to passive phase
                markCommitTimeOfFallback();
              }
            }
          }

          if (flags & Update) {
            try {
              commitSuspenseCallback(finishedWork);
            } catch (error) {
              captureCommitPhaseError(finishedWork, finishedWork.return, error);
            }

            attachSuspenseRetryListeners(finishedWork);
          }

          return;
        }

      case OffscreenComponent:
        {
          var _wasHidden = current !== null && current.memoizedState !== null;

          if ( // TODO: Remove this dead flag
           finishedWork.mode & ConcurrentMode) {
            // Before committing the children, track on the stack whether this
            // offscreen subtree was already hidden, so that we don't unmount the
            // effects again.
            var prevOffscreenSubtreeWasHidden = offscreenSubtreeWasHidden;
            offscreenSubtreeWasHidden = prevOffscreenSubtreeWasHidden || _wasHidden;
            recursivelyTraverseMutationEffects(root, finishedWork);
            offscreenSubtreeWasHidden = prevOffscreenSubtreeWasHidden;
          } else {
            recursivelyTraverseMutationEffects(root, finishedWork);
          }

          commitReconciliationEffects(finishedWork);

          if (flags & Visibility) {
            var _offscreenInstance = finishedWork.stateNode;
            var _newState = finishedWork.memoizedState;

            var _isHidden = _newState !== null;

            var offscreenBoundary = finishedWork; // Track the current state on the
  Offscreen instance so we can
            // read it during an event

            _offscreenInstance.isHidden = _isHidden;

            {
              if (_isHidden) {
                if (!_wasHidden) {
                  if ((offscreenBoundary.mode & ConcurrentMode) !== NoMode) {
                    nextEffect = offscreenBoundary;
                    var offscreenChild = offscreenBoundary.child;

                    while (offscreenChild !== null) {
                      nextEffect = offscreenChild;
                      disappearLayoutEffects_begin(offscreenChild);
                      offscreenChild = offscreenChild.sibling;
                    }
                  }
                }
              }
            }

            {
              // TODO: This needs to run whenever there's an insertion or update
              // inside a hidden Offscreen tree.
              hideOrUnhideAllChildren(offscreenBoundary, _isHidden);
            }
          }

          return;
        }

      case SuspenseListComponent:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);

          if (flags & Update) {
            attachSuspenseRetryListeners(finishedWork);
          }

          return;
        }

      case ScopeComponent:
        {

          return;
        }

      default:
        {
          recursivelyTraverseMutationEffects(root, finishedWork);
          commitReconciliationEffects(finishedWork);
          return;
        }
    }
  }

  function commitReconciliationEffects(finishedWork) {
    // Placement effects (insertions, reorders) can be scheduled on any fiber
    // type. They needs to happen after the children effects have fired, but
    // before the effects on this fiber have fired.
    var flags = finishedWork.flags;

    if (flags & Placement) {
      try {
        commitPlacement(finishedWork);
      } catch (error) {
        captureCommitPhaseError(finishedWork, finishedWork.return, error);
      } // Clear the "placement" from effect tag so that we know that this is
      // inserted, before any life-cycles like componentDidMount gets called.
      // TODO: findDOMNode doesn't rely on this any more but isMounted does
      // and isMounted is deprecated anyway so we should be able to kill this.


      finishedWork.flags &= ~Placement;
    }

    if (flags & Hydrating) {
      finishedWork.flags &= ~Hydrating;
    }
  }

  function commitLayoutEffects(finishedWork, root, committedLanes) {
    inProgressLanes = committedLanes;
    inProgressRoot = root;
    nextEffect = finishedWork;
    commitLayoutEffects_begin(finishedWork, root, committedLanes);
    inProgressLanes = null;
    inProgressRoot = null;
  }

  function commitLayoutEffects_begin(subtreeRoot, root, committedLanes) {
    // Suspense layout effects semantics don't change for legacy roots.
    var isModernRoot = (subtreeRoot.mode & ConcurrentMode) !== NoMode;

    while (nextEffect !== null) {
      var fiber = nextEffect;
      var firstChild = fiber.child;

      if ( fiber.tag === OffscreenComponent && isModernRoot) {
        // Keep track of the current Offscreen stack's state.
        var isHidden = fiber.memoizedState !== null;
        var newOffscreenSubtreeIsHidden = isHidden || offscreenSubtreeIsHidden;

        if (newOffscreenSubtreeIsHidden) {
          // The Offscreen tree is hidden. Skip over its layout effects.
          commitLayoutMountEffects_complete(subtreeRoot, root, committedLanes);
          continue;
        } else {
          // TODO (Offscreen) Also check: subtreeFlags & LayoutMask
          var current = fiber.alternate;
          var wasHidden = current !== null && current.memoizedState !== null;
          var newOffscreenSubtreeWasHidden = wasHidden || offscreenSubtreeWasHidden;
          var prevOffscreenSubtreeIsHidden = offscreenSubtreeIsHidden;
          var prevOffscreenSubtreeWasHidden = offscreenSubtreeWasHidden; // Traverse the
 Offscreen subtree with the current Offscreen as the root.

          offscreenSubtreeIsHidden = newOffscreenSubtreeIsHidden;
          offscreenSubtreeWasHidden = newOffscreenSubtreeWasHidden;
          if (offscreenSubtreeWasHidden && !prevOffscreenSubtreeWasHidden) {
            // This is the root of a reappearing boundary. Turn its layout effects
            // back on.
            nextEffect = fiber;
            reappearLayoutEffects_begin(fiber);
          }

          var child = firstChild;

          while (child !== null) {
            nextEffect = child;
            commitLayoutEffects_begin(child, // New root; bubble back up to here and
stop.
            root, committedLanes);
            child = child.sibling;
          } // Restore Offscreen state and resume in our-progress traversal.


          nextEffect = fiber;
          offscreenSubtreeIsHidden = prevOffscreenSubtreeIsHidden;
          offscreenSubtreeWasHidden = prevOffscreenSubtreeWasHidden;
          commitLayoutMountEffects_complete(subtreeRoot, root, committedLanes);
          continue;
        }
      }

      if ((fiber.subtreeFlags & LayoutMask) !== NoFlags && firstChild !== null) {
        firstChild.return = fiber;
        nextEffect = firstChild;
      } else {
        commitLayoutMountEffects_complete(subtreeRoot, root, committedLanes);
      }
    }
  }

  function commitLayoutMountEffects_complete(subtreeRoot, root, committedLanes) {
    while (nextEffect !== null) {
      var fiber = nextEffect;

      if ((fiber.flags & LayoutMask) !== NoFlags) {
        var current = fiber.alternate;
        setCurrentFiber(fiber);

        try {
          commitLayoutEffectOnFiber(root, current, fiber, committedLanes);
        } catch (error) {
          captureCommitPhaseError(fiber, fiber.return, error);
        }

        resetCurrentFiber();
      }

      if (fiber === subtreeRoot) {
        nextEffect = null;
        return;
      }

      var sibling = fiber.sibling;

      if (sibling !== null) {
        sibling.return = fiber.return;
        nextEffect = sibling;
        return;
      }

      nextEffect = fiber.return;
    }
  }
  function disappearLayoutEffects_begin(subtreeRoot) {
    while (nextEffect !== null) {
      var fiber = nextEffect;
      var firstChild = fiber.child; // TODO (Offscreen) Check: flags & (RefStatic |
LayoutStatic)

      switch (fiber.tag) {
        case FunctionComponent:
        case ForwardRef:
        case MemoComponent:
        case SimpleMemoComponent:
          {
            if ( fiber.mode & ProfileMode) {
              try {
                startLayoutEffectTimer();
                commitHookEffectListUnmount(Layout, fiber, fiber.return);
              } finally {
                recordLayoutEffectDuration(fiber);
              }
            } else {
              commitHookEffectListUnmount(Layout, fiber, fiber.return);
            }

            break;
          }

        case ClassComponent:
          {
            // TODO (Offscreen) Check: flags & RefStatic
            safelyDetachRef(fiber, fiber.return);
            var instance = fiber.stateNode;

            if (typeof instance.componentWillUnmount === 'function') {
              safelyCallComponentWillUnmount(fiber, fiber.return, instance);
            }

            break;
          }

        case HostComponent:
          {
            safelyDetachRef(fiber, fiber.return);
            break;
          }

        case OffscreenComponent:
          {
            // Check if this is a
            var isHidden = fiber.memoizedState !== null;

            if (isHidden) {
              // Nested Offscreen tree is already hidden. Don't disappear
              // its effects.
              disappearLayoutEffects_complete(subtreeRoot);
              continue;
            }

            break;
          }
      } // TODO (Offscreen) Check: subtreeFlags & LayoutStatic


      if (firstChild !== null) {
        firstChild.return = fiber;
        nextEffect = firstChild;
      } else {
        disappearLayoutEffects_complete(subtreeRoot);
      }
    }
  }

  function disappearLayoutEffects_complete(subtreeRoot) {
    while (nextEffect !== null) {
      var fiber = nextEffect;

      if (fiber === subtreeRoot) {
        nextEffect = null;
        return;
      }

      var sibling = fiber.sibling;

      if (sibling !== null) {
        sibling.return = fiber.return;
        nextEffect = sibling;
        return;
      }

      nextEffect = fiber.return;
    }
  }

  function reappearLayoutEffects_begin(subtreeRoot) {
    while (nextEffect !== null) {
      var fiber = nextEffect;
      var firstChild = fiber.child;

      if (fiber.tag === OffscreenComponent) {
        var isHidden = fiber.memoizedState !== null;

        if (isHidden) {
          // Nested Offscreen tree is still hidden. Don't re-appear its effects.
          reappearLayoutEffects_complete(subtreeRoot);
          continue;
        }
      } // TODO (Offscreen) Check: subtreeFlags & LayoutStatic


      if (firstChild !== null) {
        // This node may have been reused from a previous render, so we can't
        // assume its return pointer is correct.
        firstChild.return = fiber;
        nextEffect = firstChild;
      } else {
        reappearLayoutEffects_complete(subtreeRoot);
      }
    }
  }

  function reappearLayoutEffects_complete(subtreeRoot) {
    while (nextEffect !== null) {
      var fiber = nextEffect; // TODO (Offscreen) Check: flags & LayoutStatic

      setCurrentFiber(fiber);

      try {
        reappearLayoutEffectsOnFiber(fiber);
      } catch (error) {
        captureCommitPhaseError(fiber, fiber.return, error);
      }

      resetCurrentFiber();

      if (fiber === subtreeRoot) {
        nextEffect = null;
        return;
      }
      var sibling = fiber.sibling;

      if (sibling !== null) {
        // This node may have been reused from a previous render, so we can't
        // assume its return pointer is correct.
        sibling.return = fiber.return;
        nextEffect = sibling;
        return;
      }

      nextEffect = fiber.return;
    }
  }

  function commitPassiveMountEffects(root, finishedWork, committedLanes,
committedTransitions) {
    nextEffect = finishedWork;
    commitPassiveMountEffects_begin(finishedWork, root, committedLanes,
committedTransitions);
  }

  function commitPassiveMountEffects_begin(subtreeRoot, root, committedLanes,
committedTransitions) {
    while (nextEffect !== null) {
      var fiber = nextEffect;
      var firstChild = fiber.child;

      if ((fiber.subtreeFlags & PassiveMask) !== NoFlags && firstChild !== null) {
        firstChild.return = fiber;
        nextEffect = firstChild;
      } else {
        commitPassiveMountEffects_complete(subtreeRoot, root, committedLanes,
committedTransitions);
      }
    }
  }

  function commitPassiveMountEffects_complete(subtreeRoot, root, committedLanes,
committedTransitions) {
    while (nextEffect !== null) {
      var fiber = nextEffect;

      if ((fiber.flags & Passive) !== NoFlags) {
        setCurrentFiber(fiber);

        try {
          commitPassiveMountOnFiber(root, fiber, committedLanes, committedTransitions);
        } catch (error) {
          captureCommitPhaseError(fiber, fiber.return, error);
        }

        resetCurrentFiber();
      }

      if (fiber === subtreeRoot) {
        nextEffect = null;
        return;
      }

      var sibling = fiber.sibling;

      if (sibling !== null) {
        sibling.return = fiber.return;
        nextEffect = sibling;
        return;
      }

      nextEffect = fiber.return;
    }
  }

  function commitPassiveMountOnFiber(finishedRoot, finishedWork, committedLanes,
committedTransitions) {
    switch (finishedWork.tag) {
      case FunctionComponent:
      case ForwardRef:
      case SimpleMemoComponent:
        {
          if ( finishedWork.mode & ProfileMode) {
            startPassiveEffectTimer();

            try {
              commitHookEffectListMount(Passive$1 | HasEffect, finishedWork);
            } finally {
              recordPassiveEffectDuration(finishedWork);
            }
          } else {
            commitHookEffectListMount(Passive$1 | HasEffect, finishedWork);
          }

          break;
        }
    }
  }

  function commitPassiveUnmountEffects(firstChild) {
    nextEffect = firstChild;
    commitPassiveUnmountEffects_begin();
  }

  function commitPassiveUnmountEffects_begin() {
    while (nextEffect !== null) {
      var fiber = nextEffect;
      var child = fiber.child;

      if ((nextEffect.flags & ChildDeletion) !== NoFlags) {
        var deletions = fiber.deletions;

        if (deletions !== null) {
          for (var i = 0; i < deletions.length; i++) {
            var fiberToDelete = deletions[i];
            nextEffect = fiberToDelete;
            commitPassiveUnmountEffectsInsideOfDeletedTree_begin(fiberToDelete, fiber);
          }

          {
            // A fiber was deleted from this parent fiber, but it's still part of
            // the previous (alternate) parent fiber's list of children. Because
            // children are a linked list, an earlier sibling that's still alive
            // will be connected to the deleted fiber via its `alternate`:
            //
            //   live fiber
            //   --alternate--> previous live fiber
            //   --sibling--> deleted fiber
            //
            // We can't disconnect `alternate` on nodes that haven't been deleted
            // yet, but we can disconnect the `sibling` and `child` pointers.
            var previousFiber = fiber.alternate;

            if (previousFiber !== null) {
              var detachedChild = previousFiber.child;

              if (detachedChild !== null) {
                previousFiber.child = null;

                do {
                  var detachedSibling = detachedChild.sibling;
                  detachedChild.sibling = null;
                  detachedChild = detachedSibling;
                } while (detachedChild !== null);
              }
            }
          }

          nextEffect = fiber;
        }
      }

      if ((fiber.subtreeFlags & PassiveMask) !== NoFlags && child !== null) {
        child.return = fiber;
        nextEffect = child;
      } else {
        commitPassiveUnmountEffects_complete();
      }
    }
  }

  function commitPassiveUnmountEffects_complete() {
    while (nextEffect !== null) {
      var fiber = nextEffect;

      if ((fiber.flags & Passive) !== NoFlags) {
        setCurrentFiber(fiber);
        commitPassiveUnmountOnFiber(fiber);
        resetCurrentFiber();
      }

      var sibling = fiber.sibling;

      if (sibling !== null) {
        sibling.return = fiber.return;
        nextEffect = sibling;
        return;
      }

      nextEffect = fiber.return;
    }
  }

  function commitPassiveUnmountOnFiber(finishedWork) {
    switch (finishedWork.tag) {
      case FunctionComponent:
      case ForwardRef:
      case SimpleMemoComponent:
        {
          if ( finishedWork.mode & ProfileMode) {
            startPassiveEffectTimer();
            commitHookEffectListUnmount(Passive$1 | HasEffect, finishedWork,
finishedWork.return);
            recordPassiveEffectDuration(finishedWork);
          } else {
            commitHookEffectListUnmount(Passive$1 | HasEffect, finishedWork,
finishedWork.return);
          }

          break;
        }
    }
  }

  function commitPassiveUnmountEffectsInsideOfDeletedTree_begin(deletedSubtreeRoot,
nearestMountedAncestor) {
    while (nextEffect !== null) {
      var fiber = nextEffect; // Deletion effects fire in parent -> child order
      // TODO: Check if fiber has a PassiveStatic flag
      setCurrentFiber(fiber);
      commitPassiveUnmountInsideDeletedTreeOnFiber(fiber, nearestMountedAncestor);
      resetCurrentFiber();
      var child = fiber.child; // TODO: Only traverse subtree if it has a PassiveStatic
flag. (But, if we
      // do this, still need to handle `deletedTreeCleanUpLevel` correctly.)

      if (child !== null) {
        child.return = fiber;
        nextEffect = child;
      } else {
        commitPassiveUnmountEffectsInsideOfDeletedTree_complete(deletedSubtreeRoot);
      }
    }
  }

  function commitPassiveUnmountEffectsInsideOfDeletedTree_complete(deletedSubtreeRoot) {
    while (nextEffect !== null) {
      var fiber = nextEffect;
      var sibling = fiber.sibling;
      var returnFiber = fiber.return;

      {
        // Recursively traverse the entire deleted tree and clean up fiber fields.
        // This is more aggressive than ideal, and the long term goal is to only
        // have to detach the deleted tree at the root.
        detachFiberAfterEffects(fiber);

        if (fiber === deletedSubtreeRoot) {
          nextEffect = null;
          return;
        }
      }

      if (sibling !== null) {
        sibling.return = returnFiber;
        nextEffect = sibling;
        return;
      }

      nextEffect = returnFiber;
    }
  }

  function commitPassiveUnmountInsideDeletedTreeOnFiber(current, nearestMountedAncestor)
{
    switch (current.tag) {
      case FunctionComponent:
      case ForwardRef:
      case SimpleMemoComponent:
        {
          if ( current.mode & ProfileMode) {
            startPassiveEffectTimer();
            commitHookEffectListUnmount(Passive$1, current, nearestMountedAncestor);
            recordPassiveEffectDuration(current);
          } else {
            commitHookEffectListUnmount(Passive$1, current, nearestMountedAncestor);
          }

          break;
        }
    }
  } // TODO: Reuse reappearLayoutEffects traversal here?


  function invokeLayoutEffectMountInDEV(fiber) {
    {
      // We don't need to re-check StrictEffectsMode here.
      // This function is only called if that check has already passed.
      switch (fiber.tag) {
        case FunctionComponent:
        case ForwardRef:
        case SimpleMemoComponent:
          {
            try {
              commitHookEffectListMount(Layout | HasEffect, fiber);
            } catch (error) {
              captureCommitPhaseError(fiber, fiber.return, error);
            }

            break;
          }

        case ClassComponent:
          {
            var instance = fiber.stateNode;

            try {
              instance.componentDidMount();
            } catch (error) {
              captureCommitPhaseError(fiber, fiber.return, error);
            }

            break;
          }
      }
    }
  }

  function invokePassiveEffectMountInDEV(fiber) {
    {
      // We don't need to re-check StrictEffectsMode here.
      // This function is only called if that check has already passed.
      switch (fiber.tag) {
        case FunctionComponent:
        case ForwardRef:
        case SimpleMemoComponent:
          {
            try {
              commitHookEffectListMount(Passive$1 | HasEffect, fiber);
            } catch (error) {
              captureCommitPhaseError(fiber, fiber.return, error);
            }

            break;
          }
      }
    }
  }

  function invokeLayoutEffectUnmountInDEV(fiber) {
    {
      // We don't need to re-check StrictEffectsMode here.
      // This function is only called if that check has already passed.
      switch (fiber.tag) {
        case FunctionComponent:
        case ForwardRef:
        case SimpleMemoComponent:
          {
            try {
              commitHookEffectListUnmount(Layout | HasEffect, fiber, fiber.return);
            } catch (error) {
              captureCommitPhaseError(fiber, fiber.return, error);
            }

            break;
          }
        case ClassComponent:
          {
            var instance = fiber.stateNode;

            if (typeof instance.componentWillUnmount === 'function') {
              safelyCallComponentWillUnmount(fiber, fiber.return, instance);
            }

            break;
          }
      }
    }
  }

  function invokePassiveEffectUnmountInDEV(fiber) {
    {
      // We don't need to re-check StrictEffectsMode here.
      // This function is only called if that check has already passed.
      switch (fiber.tag) {
        case FunctionComponent:
        case ForwardRef:
        case SimpleMemoComponent:
          {
            try {
              commitHookEffectListUnmount(Passive$1 | HasEffect, fiber, fiber.return);
            } catch (error) {
              captureCommitPhaseError(fiber, fiber.return, error);
            }
          }
      }
    }
  }

  var COMPONENT_TYPE = 0;
  var HAS_PSEUDO_CLASS_TYPE = 1;
  var ROLE_TYPE = 2;
  var TEST_NAME_TYPE = 3;
  var TEXT_TYPE = 4;

  if (typeof Symbol === 'function' && Symbol.for) {
    var symbolFor = Symbol.for;
    COMPONENT_TYPE = symbolFor('selector.component');
    HAS_PSEUDO_CLASS_TYPE = symbolFor('selector.has_pseudo_class');
    ROLE_TYPE = symbolFor('selector.role');
    TEST_NAME_TYPE = symbolFor('selector.test_id');
    TEXT_TYPE = symbolFor('selector.text');
  }
  var commitHooks = [];
  function onCommitRoot$1() {
    {
      commitHooks.forEach(function (commitHook) {
        return commitHook();
      });
    }
  }

  var ReactCurrentActQueue = ReactSharedInternals.ReactCurrentActQueue;
  function isLegacyActEnvironment(fiber) {
    {
      // Legacy mode. We preserve the behavior of React 17's act. It assumes an
      // act environment whenever `jest` is defined, but you can still turn off
      // spurious warnings by setting IS_REACT_ACT_ENVIRONMENT explicitly
      // to false.
      var isReactActEnvironmentGlobal = // $FlowExpectedError – Flow doesn't know about
IS_REACT_ACT_ENVIRONMENT global
      typeof IS_REACT_ACT_ENVIRONMENT !== 'undefined' ? IS_REACT_ACT_ENVIRONMENT :
undefined; // $FlowExpectedError - Flow doesn't know about jest

      var jestIsDefined = typeof jest !== 'undefined';
      return  jestIsDefined && isReactActEnvironmentGlobal !== false;
    }
  }
  function isConcurrentActEnvironment() {
    {
      var isReactActEnvironmentGlobal = // $FlowExpectedError – Flow doesn't know about
IS_REACT_ACT_ENVIRONMENT global
      typeof IS_REACT_ACT_ENVIRONMENT !== 'undefined' ? IS_REACT_ACT_ENVIRONMENT :
undefined;

      if (!isReactActEnvironmentGlobal && ReactCurrentActQueue.current !== null) {
        // TODO: Include link to relevant documentation page.
        error('The current testing environment is not configured to support ' +
'act(...)');
      }

      return isReactActEnvironmentGlobal;
    }
  }

  var ceil = Math.ceil;
  var ReactCurrentDispatcher$2 = ReactSharedInternals.ReactCurrentDispatcher,
      ReactCurrentOwner$2 = ReactSharedInternals.ReactCurrentOwner,
      ReactCurrentBatchConfig$3 = ReactSharedInternals.ReactCurrentBatchConfig,
      ReactCurrentActQueue$1 = ReactSharedInternals.ReactCurrentActQueue;
  var NoContext =
  /*             */
  0;
  var BatchedContext =
  /*               */
  1;
  var RenderContext =
  /*                */
  2;
  var CommitContext =
  /*                */
  4;
  var RootInProgress = 0;
  var RootFatalErrored = 1;
  var RootErrored = 2;
  var RootSuspended = 3;
  var RootSuspendedWithDelay = 4;
  var RootCompleted = 5;
  var RootDidNotComplete = 6; // Describes where we are in the React execution stack

  var executionContext = NoContext; // The root we're working on

  var workInProgressRoot = null; // The fiber we're working on

  var workInProgress = null; // The lanes we're rendering

  var workInProgressRootRenderLanes = NoLanes; // Stack that allows components to change
the render lanes for its subtree
  // This is a superset of the lanes we started working on at the root. The only
  // case where it's different from `workInProgressRootRenderLanes` is when we
  // enter a subtree that is hidden and needs to be unhidden: Suspense and
  // Offscreen component.
  //
  // Most things in the work loop should deal with workInProgressRootRenderLanes.
  // Most things in begin/complete phases should deal with subtreeRenderLanes.

  var subtreeRenderLanes = NoLanes;
  var subtreeRenderLanesCursor = createCursor(NoLanes); // Whether to root completed,
errored, suspended, etc.

  var workInProgressRootExitStatus = RootInProgress; // A fatal error, if one is thrown

  var workInProgressRootFatalError = null; // "Included" lanes refer to lanes that were
worked on during this render. It's
```

```
  // slightly different than `renderLanes` because `renderLanes` can change as you
  // enter and exit an Offscreen tree. This value is the combination of all render
  // lanes for the entire render phase.

  var workInProgressRootIncludedLanes = NoLanes; // The work left over by components that
were visited during this render. Only
  // includes unprocessed updates, not work in bailed out children.

  var workInProgressRootSkippedLanes = NoLanes; // Lanes that were updated (in an
interleaved event) during this render.

  var workInProgressRootInterleavedUpdatedLanes = NoLanes; // Lanes that were updated
during the render phase (*not* an interleaved event).

  var workInProgressRootPingedLanes = NoLanes; // Errors that are thrown during the
render phase.

  var workInProgressRootConcurrentErrors = null; // These are errors that we recovered
from without surfacing them to the UI.
  // We will log them once the tree commits.

  var workInProgressRootRecoverableErrors = null; // 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.

  var globalMostRecentFallbackTime = 0;
  var FALLBACK_THROTTLE_MS = 500; // The absolute time for when we should start giving up
on rendering
  // more and prefer CPU suspense heuristics instead.

  var workInProgressRootRenderTargetTime = Infinity; // How long a render is supposed to
take before we start following CPU
  // suspense heuristics and opt out of rendering more content.

  var RENDER_TIMEOUT_MS = 500;
  var workInProgressTransitions = null;

  function resetRenderTimer() {
    workInProgressRootRenderTargetTime = now() + RENDER_TIMEOUT_MS;
  }

  function getRenderTargetTime() {
    return workInProgressRootRenderTargetTime;
  }
  var hasUncaughtError = false;
  var firstUncaughtError = null;
  var legacyErrorBoundariesThatAlreadyFailed = null; // Only used when
enableProfilerNestedUpdateScheduledHook is true;
  var rootDoesHavePassiveEffects = false;
  var rootWithPendingPassiveEffects = null;
  var pendingPassiveEffectsLanes = NoLanes;
  var pendingPassiveProfilerEffects = [];
  var pendingPassiveTransitions = null; // Use these to prevent an infinite loop of
nested updates

  var NESTED_UPDATE_LIMIT = 50;
  var nestedUpdateCount = 0;
  var rootWithNestedUpdates = null;
  var isFlushingPassiveEffects = false;
  var didScheduleUpdateDuringPassiveEffects = false;
  var NESTED_PASSIVE_UPDATE_LIMIT = 50;
  var nestedPassiveUpdateCount = 0;
  var rootWithPassiveNestedUpdates = null; // If two updates are scheduled within the
same event, we should treat their
  // event times as simultaneous, even if the actual clock time has advanced
  // between the first and second call.

  var currentEventTime = NoTimestamp;
  var currentEventTransitionLane = NoLanes;
```

```
    var isRunningInsertionEffect = false;
    function getWorkInProgressRoot() {
      return workInProgressRoot;
    }
    function requestEventTime() {
      if ((executionContext & (RenderContext | CommitContext)) !== NoContext) {
        // We're inside React, so it's fine to read the actual time.
        return now();
      } // We're not inside React, so we may be in the middle of a browser event.


      if (currentEventTime !== NoTimestamp) {
        // Use the same start time for all updates until we enter React again.
        return currentEventTime;
      } // This is the first update since React yielded. Compute a new start time.


      currentEventTime = now();
      return currentEventTime;
    }
    function requestUpdateLane(fiber) {
      // Special cases
      var mode = fiber.mode;

      if ((mode & ConcurrentMode) === NoMode) {
        return SyncLane;
      } else if ( (executionContext & RenderContext) !== NoContext &&
  workInProgressRootRenderLanes !== NoLanes) {
        // This is a render phase update. These are not officially supported. The
        // old behavior is to give this the same "thread" (lanes) as
        // whatever is currently rendering. So if you call `setState` on a component
        // that happens later in the same render, it will flush. Ideally, we want to
        // remove the special case and treat them as if they came from an
        // interleaved event. Regardless, this pattern is not officially supported.
        // This behavior is only a fallback. The flag only exists until we can roll
        // out the setState warning, since existing code might accidentally rely on
        // the current behavior.
        return pickArbitraryLane(workInProgressRootRenderLanes);
      }

      var isTransition = requestCurrentTransition() !== NoTransition;

      if (isTransition) {
        if ( ReactCurrentBatchConfig$3.transition !== null) {
          var transition = ReactCurrentBatchConfig$3.transition;

          if (!transition._updatedFibers) {
            transition._updatedFibers = new Set();
          }

          transition._updatedFibers.add(fiber);
        } // The algorithm for assigning an update to a lane should be stable for all
        // updates at the same priority within the same event. To do this, the
        // inputs to the algorithm must be the same.
        //
        // The trick we use is to cache the first of each of these inputs within an
        // event. Then reset the cached values once we can be sure the event is
        // over. Our heuristic for that is whenever we enter a concurrent work loop.


        if (currentEventTransitionLane === NoLane) {
          // All transitions within the same event are assigned the same lane.
          currentEventTransitionLane = claimNextTransitionLane();
        }

        return currentEventTransitionLane;
      } // Updates originating inside certain React methods, like flushSync, have
      // their priority set by tracking it with a context variable.
      //
      // The opaque type returned by the host config is internally a lane, so we can
      // use that directly.
      // TODO: Move this type conversion to the event priority module.


      var updateLane = getCurrentUpdatePriority();

      if (updateLane !== NoLane) {
        return updateLane;
      } // This update originated outside React. Ask the host environment for an
      // appropriate priority, based on the type of event.
      //
      // The opaque type returned by the host config is internally a lane, so we can
      // use that directly.
      // TODO: Move this type conversion to the event priority module.


      var eventLane = getCurrentEventPriority();
      return eventLane;
    }

    function requestRetryLane(fiber) {
      // This is a fork of `requestUpdateLane` designed specifically for Suspense
      // "retries" — a special update that attempts to flip a Suspense boundary
      // from its placeholder state to its primary/resolved state.
      // Special cases
      var mode = fiber.mode;

      if ((mode & ConcurrentMode) === NoMode) {
        return SyncLane;
      }

      return claimNextRetryLane();
    }

    function scheduleUpdateOnFiber(root, fiber, lane, eventTime) {
      checkForNestedUpdates();

      {
        if (isRunningInsertionEffect) {
          error('useInsertionEffect must not schedule updates.');
        }
      }

      {
        if (isFlushingPassiveEffects) {
          didScheduleUpdateDuringPassiveEffects = true;
        }
      } // Mark that the root has a pending update.


      markRootUpdated(root, lane, eventTime);

      if ((executionContext & RenderContext) !== NoLanes && root === workInProgressRoot) {
        // This update was dispatched during the render phase. This is a mistake
        // if the update originates from user space (with the exception of local
        // hook updates, which are handled differently and don't reach this
        // function), but there are some internal React features that use this as
        // an implementation detail, like selective hydration.
        warnAboutRenderPhaseUpdatesInDEV(fiber); // Track lanes that were updated during
  the render phase
      } else {
        // This is a normal update, scheduled from outside the render phase. For
        // example, during an input event.
        {
          if (isDevToolsPresent) {
            addFiberToLanesMap(root, fiber, lane);
          }
        }
        warnIfUpdatesNotWrappedWithActDEV(fiber);

        if (root === workInProgressRoot) {
          // Received an update to a tree that's in the middle of rendering. Mark
          // that there was an interleaved update work on this root. Unless the
          // `deferRenderPhaseUpdateToNextBatch` flag is off and this is a render
          // phase update. In that case, we don't treat render phase updates as if
          // they were interleaved, for backwards compat reasons.
          if ( (executionContext & RenderContext) === NoContext) {
            workInProgressRootInterleavedUpdatedLanes =
  mergeLanes(workInProgressRootInterleavedUpdatedLanes, lane);
          }

          if (workInProgressRootExitStatus === RootSuspendedWithDelay) {
            // The root already suspended with a delay, which means this render
            // definitely won't finish. Since we have a new update, let's mark it as
            // suspended now, right before marking the incoming update. This has the
            // effect of interrupting the current render and switching to the update.
            // TODO: Make sure this doesn't override pings that happen while we've
            // already started rendering.
            markRootSuspended$1(root, workInProgressRootRenderLanes);
          }
        }

        ensureRootIsScheduled(root, eventTime);

        if (lane === SyncLane && executionContext === NoContext && (fiber.mode &
  ConcurrentMode) === NoMode && // Treat `act` as if it's inside `batchedUpdates`, even in
  legacy mode.
        !( ReactCurrentActQueue$1.isBatchingLegacy)) {
          // Flush the synchronous work now, unless we're already working or inside
          // a batch. This is intentionally inside scheduleUpdateOnFiber instead of
          // scheduleCallbackForFiber to preserve the ability to schedule a callback
          // without immediately flushing it. We only do this for user-initiated
          // updates, to preserve historical behavior of legacy mode.
          resetRenderTimer();
          flushSyncCallbacksOnlyInLegacyMode();
        }
      }
    }
    function scheduleInitialHydrationOnRoot(root, lane, eventTime) {
      // This is a special fork of scheduleUpdateOnFiber that is only used to
      // schedule the initial hydration of a root that has just been created. Most
      // of the stuff in scheduleUpdateOnFiber can be skipped.
      //
      // The main reason for this separate path, though, is to distinguish the
      // initial children from subsequent updates. In fully client-rendered roots
      // (createRoot instead of hydrateRoot), all top-level renders are modeled as
      // updates, but hydration roots are special because the initial render must
      // match what was rendered on the server.
      var current = root.current;
      current.lanes = lane;
      markRootUpdated(root, lane, eventTime);
      ensureRootIsScheduled(root, eventTime);
    }
    function isUnsafeClassRenderPhaseUpdate(fiber) {
      // Check if this is a render phase update. Only called by class components,
      // which special (deprecated) behavior for UNSAFE_componentWillReceive props.
      return (// TODO: Remove outdated deferRenderPhaseUpdateToNextBatch experiment. We
        // decided not to enable it.
         (executionContext & RenderContext) !== NoContext
      );
    } // Use this function to schedule a task for a root. There's only one task per
    // root; if a task was already scheduled, we'll check to make sure the priority
    // of the existing task is the same as the priority of the next level that the
    // root has work on. This function is called on every update, and right before
    // exiting a task.
```

```
    function ensureRootIsScheduled(root, currentTime) {
      var existingCallbackNode = root.callbackNode; // Check if any lanes are being starved
  by other work. If so, mark them as
      // expired so we know to work on those next.

      markStarvedLanesAsExpired(root, currentTime); // Determine the next lanes to work on,
  and their priority.

      var nextLanes = getNextLanes(root, root === workInProgressRoot ?
  workInProgressRootRenderLanes : NoLanes);

      if (nextLanes === NoLanes) {
        // Special case: There's nothing to work on.
        if (existingCallbackNode !== null) {
          cancelCallback$1(existingCallbackNode);
        }

        root.callbackNode = null;
        root.callbackPriority = NoLane;
        return;
      } // We use the highest priority lane to represent the priority of the callback.


      var newCallbackPriority = getHighestPriorityLane(nextLanes); // Check if there's an
  existing task. We may be able to reuse it.

      var existingCallbackPriority = root.callbackPriority;

      if (existingCallbackPriority === newCallbackPriority && // Special case related to
  `act`. If the currently scheduled task is a
      // Scheduler task, rather than an `act` task, cancel it and re-scheduled
      // on the `act` queue.
      !( ReactCurrentActQueue$1.current !== null && existingCallbackNode !==
  fakeActCallbackNode)) {
        {
          // If we're going to re-use an existing task, it needs to exist.
          // Assume that discrete update microtasks are non-cancellable and null.
          // TODO: Temporary until we confirm this warning is not fired.
          if (existingCallbackNode == null && existingCallbackPriority !== SyncLane) {
            error('Expected scheduled callback to exist. This error is likely caused by a
  bug in React. Please file an issue.');
          }
        } // The priority hasn't changed. We can reuse the existing task. Exit.


        return;
      }

      if (existingCallbackNode != null) {
        // Cancel the existing callback. We'll schedule a new one below.
        cancelCallback$1(existingCallbackNode);
      } // Schedule a new callback.


      var newCallbackNode;

      if (newCallbackPriority === SyncLane) {
        // Special case: Sync React callbacks are scheduled on a special
        // internal queue
        if (root.tag === LegacyRoot) {
          if ( ReactCurrentActQueue$1.isBatchingLegacy !== null) {
            ReactCurrentActQueue$1.didScheduleLegacyUpdate = true;
          }

          scheduleLegacySyncCallback(performSyncWorkOnRoot.bind(null, root));
        } else {
          scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
        }
        {
          // Flush the queue in a microtask.
          if ( ReactCurrentActQueue$1.current !== null) {
            // Inside `act`, use our internal `act` queue so that these get flushed
            // at the end of the current scope even when using the sync version
            // of `act`.
            ReactCurrentActQueue$1.current.push(flushSyncCallbacks);
          } else {
            scheduleMicrotask(function () {
              // In Safari, appending an iframe forces microtasks to run.
              // https://github.com/facebook/react/issues/22459
              // We don't support running callbacks in the middle of render
              // or commit so we need to check against that.
              if ((executionContext & (RenderContext | CommitContext)) === NoContext) {
                // Note that this would still prematurely flush the callbacks
                // if this happens outside render or commit phase (e.g. in an event).
                flushSyncCallbacks();
              }
            });
          }
        }

        newCallbackNode = null;
      } else {
        var schedulerPriorityLevel;

        switch (lanesToEventPriority(nextLanes)) {
          case DiscreteEventPriority:
            schedulerPriorityLevel = ImmediatePriority;
            break;

          case ContinuousEventPriority:
            schedulerPriorityLevel = UserBlockingPriority;
            break;

          case DefaultEventPriority:
            schedulerPriorityLevel = NormalPriority;
            break;

          case IdleEventPriority:
            schedulerPriorityLevel = IdlePriority;
            break;

          default:
            schedulerPriorityLevel = NormalPriority;
            break;
        }

        newCallbackNode = scheduleCallback$1(schedulerPriorityLevel,
  performConcurrentWorkOnRoot.bind(null, root));
      }

      root.callbackPriority = newCallbackPriority;
      root.callbackNode = newCallbackNode;
    } // This is the entry point for every concurrent task, i.e. anything that
    // goes through Scheduler.


    function performConcurrentWorkOnRoot(root, didTimeout) {
      {
        resetNestedUpdateFlag();
      } // Since we know we're in a React event, we can clear the current
      // event time. The next update will compute a new event time.


      currentEventTime = NoTimestamp;
      currentEventTransitionLane = NoLanes;

      if ((executionContext & (RenderContext | CommitContext)) !== NoContext) {
```

```
      throw new Error('Should not already be working.');
    } // Flush any pending passive effects before deciding which lanes to work on,
    // in case they schedule additional work.


    var originalCallbackNode = root.callbackNode;
    var didFlushPassiveEffects = flushPassiveEffects();

    if (didFlushPassiveEffects) {
      // Something in the passive effect phase may have canceled the current task.
      // Check if the task node for this root was changed.
      if (root.callbackNode !== originalCallbackNode) {
        // The current task was canceled. Exit. We don't need to call
        // `ensureRootIsScheduled` because the check above implies either that
        // there's a new task, or that there's no remaining work on this root.
        return null;
      }
    } // Determine the next lanes to work on, using the fields stored
    // on the root.


    var lanes = getNextLanes(root, root === workInProgressRoot ?
workInProgressRootRenderLanes : NoLanes);

    if (lanes === NoLanes) {
      // Defensive coding. This is never expected to happen.
      return null;
    } // We disable time-slicing in some cases: if the work has been CPU-bound
    // for too long ("expired" work, to prevent starvation), or we're in
    // sync-updates-by-default mode.
    // TODO: We only check `didTimeout` defensively, to account for a Scheduler
    // bug we're still investigating. Once the bug in Scheduler is fixed,
    // we can remove this, since we track expiration ourselves.


    var shouldTimeSlice = !includesBlockingLane(root, lanes) &&
!includesExpiredLane(root, lanes) && ( !didTimeout);
    var exitStatus = shouldTimeSlice ? renderRootConcurrent(root, lanes) :
renderRootSync(root, lanes);

    if (exitStatus !== RootInProgress) {
      if (exitStatus === RootErrored) {
        // If something threw an error, try rendering one more time. We'll
        // render synchronously to block concurrent data mutations, and we'll
        // includes all pending updates are included. If it still fails after
        // the second attempt, we'll give up and commit the resulting tree.
        var errorRetryLanes = getLanesToRetrySynchronouslyOnError(root);

        if (errorRetryLanes !== NoLanes) {
          lanes = errorRetryLanes;
          exitStatus = recoverFromConcurrentError(root, errorRetryLanes);
        }
      }

      if (exitStatus === RootFatalErrored) {
        var fatalError = workInProgressRootFatalError;
        prepareFreshStack(root, NoLanes);
        markRootSuspended$1(root, lanes);
        ensureRootIsScheduled(root, now());
        throw fatalError;
      }

      if (exitStatus === RootDidNotComplete) {
        // The render unwound without completing the tree. This happens in special
        // cases where need to exit the current render without producing a
        // consistent tree or committing.
        //
        // This should only happen during a concurrent render, not a discrete or
        // synchronous update. We should have already checked for this when we
        // unwound the stack.
        markRootSuspended$1(root, lanes);
      } else {
        // The render completed.
        // Check if this render may have yielded to a concurrent event, and if so,
        // confirm that any newly rendered stores are consistent.
        // TODO: It's possible that even a concurrent render may never have yielded
        // to the main thread, if it was fast enough, or if it expired. We could
        // skip the consistency check in that case, too.
        var renderWasConcurrent = !includesBlockingLane(root, lanes);
        var finishedWork = root.current.alternate;

        if (renderWasConcurrent && !isRenderConsistentWithExternalStores(finishedWork)) {
          // A store was mutated in an interleaved event. Render again,
          // synchronously, to block further mutations.
          exitStatus = renderRootSync(root, lanes); // We need to check again if
something threw

          if (exitStatus === RootErrored) {
            var _errorRetryLanes = getLanesToRetrySynchronouslyOnError(root);

            if (_errorRetryLanes !== NoLanes) {
              lanes = _errorRetryLanes;
              exitStatus = recoverFromConcurrentError(root, _errorRetryLanes); // We
assume the tree is now consistent because we didn't yield to any
              // concurrent events.
            }
          }

          if (exitStatus === RootFatalErrored) {
            var _fatalError = workInProgressRootFatalError;
            prepareFreshStack(root, NoLanes);
            markRootSuspended$1(root, lanes);
            ensureRootIsScheduled(root, now());
            throw _fatalError;
          }
        } // We now have a consistent tree. The next step is either to commit it,
        // or, if something suspended, wait to commit it after a timeout.


        root.finishedWork = finishedWork;
        root.finishedLanes = lanes;
        finishConcurrentRender(root, exitStatus, lanes);
      }
    }

    ensureRootIsScheduled(root, now());

    if (root.callbackNode === originalCallbackNode) {
      // The task node scheduled for this root is the same one that's
      // currently executed. Need to return a continuation.
      return performConcurrentWorkOnRoot.bind(null, root);
    }

    return null;
  }

  function recoverFromConcurrentError(root, errorRetryLanes) {
    // If an error occurred during hydration, discard server response and fall
    // back to client side render.
    // Before rendering again, save the errors from the previous attempt.
    var errorsFromFirstAttempt = workInProgressRootConcurrentErrors;

    if (isRootDehydrated(root)) {
      // The shell failed to hydrate. Set a flag to force a client rendering
      // during the next attempt. To do this, we call prepareFreshStack now
      // to create the root work-in-progress fiber. This is a bit weird in terms
      // of factoring, because it relies on renderRootSync not calling
      // prepareFreshStack again in the call below, which happens because the
      // root and lanes haven't changed.
      //
      // TODO: I think what we should do is set ForceClientRender inside
      // throwException, like we do for nested Suspense boundaries. The reason
      // it's here instead is so we can switch to the synchronous work loop, too.
      // Something to consider for a future refactor.
      var rootWorkInProgress = prepareFreshStack(root, errorRetryLanes);
      rootWorkInProgress.flags |= ForceClientRender;

      {
        errorHydratingContainer(root.containerInfo);
      }
    }

    var exitStatus = renderRootSync(root, errorRetryLanes);

    if (exitStatus !== RootErrored) {
      // Successfully finished rendering on retry
      // The errors from the failed first attempt have been recovered. Add
      // them to the collection of recoverable errors. We'll log them in the
      // commit phase.
      var errorsFromSecondAttempt = workInProgressRootRecoverableErrors;
      workInProgressRootRecoverableErrors = errorsFromFirstAttempt; // The errors from
the second attempt should be queued after the errors
      // from the first attempt, to preserve the causal sequence.

      if (errorsFromSecondAttempt !== null) {
        queueRecoverableErrors(errorsFromSecondAttempt);
      }
    }

    return exitStatus;
  }

  function queueRecoverableErrors(errors) {
    if (workInProgressRootRecoverableErrors === null) {
      workInProgressRootRecoverableErrors = errors;
    } else {
      workInProgressRootRecoverableErrors.push.apply(workInProgressRootRecoverableErrors,
errors);
    }
  }

  function finishConcurrentRender(root, exitStatus, lanes) {
    switch (exitStatus) {
      case RootInProgress:
      case RootFatalErrored:
        {
          throw new Error('Root did not complete. This is a bug in React.');
        }
      // Flow knows about invariant, so it complains if I add a break
      // statement, but eslint doesn't know about invariant, so it complains
      // if I do. eslint-disable-next-line no-fallthrough

      case RootErrored:
        {
          // We should have already attempted to retry this tree. If we reached
          // this point, it errored again. Commit it.
          commitRoot(root, workInProgressRootRecoverableErrors,
workInProgressTransitions);
          break;
        }

      case RootSuspended:
        {
          markRootSuspended$1(root, lanes); // We have an acceptable loading state. We
need to figure out if we
          // should immediately commit it or wait a bit.
          if (includesOnlyRetries(lanes) && // do not delay if we're inside an act()
scope
          !shouldForceFlushFallbacksInDEV()) {
            // This render only included retries, no updates. Throttle committing
            // retries so that we don't show too many loading states too quickly.
            var msUntilTimeout = globalMostRecentFallbackTime + FALLBACK_THROTTLE_MS -
now(); // Don't bother with a very short suspense time.

            if (msUntilTimeout > 10) {
              var nextLanes = getNextLanes(root, NoLanes);

              if (nextLanes !== NoLanes) {
                // There's additional work on this root.
                break;
              }

              var suspendedLanes = root.suspendedLanes;

              if (!isSubsetOfLanes(suspendedLanes, lanes)) {
                // We should prefer to render the fallback of at the last
                // suspended level. Ping the last suspended level to try
                // rendering it again.
                // FIXME: What if the suspended lanes are Idle? Should not restart.
                var eventTime = requestEventTime();
                markRootPinged(root, suspendedLanes);
                break;
              } // The render is suspended, it hasn't timed out, and there's no
              // lower priority work to do. Instead of committing the fallback
              // immediately, wait for more data to arrive.


              root.timeoutHandle = scheduleTimeout(commitRoot.bind(null, root,
workInProgressRootRecoverableErrors, workInProgressTransitions), msUntilTimeout);
              break;
            }
          } // The work expired. Commit immediately.


          commitRoot(root, workInProgressRootRecoverableErrors,
workInProgressTransitions);
          break;
        }

      case RootSuspendedWithDelay:
        {
          markRootSuspended$1(root, lanes);

          if (includesOnlyTransitions(lanes)) {
            // This is a transition, so we should exit without committing a
            // placeholder and without scheduling a timeout. Delay indefinitely
            // until we receive more data.
            break;
          }

          if (!shouldForceFlushFallbacksInDEV()) {
            // This is not a transition, but we did trigger an avoided state.
            // Schedule a placeholder to display after a short delay, using the Just
            // Noticeable Difference.
            // TODO: Is the JND optimization worth the added complexity? If this is
            // the only reason we track the event time, then probably not.
            // Consider removing.
            var mostRecentEventTime = getMostRecentEventTime(root, lanes);
            var eventTimeMs = mostRecentEventTime;
            var timeElapsedMs = now() - eventTimeMs;

            var _msUntilTimeout = jnd(timeElapsedMs) - timeElapsedMs; // Don't bother
with a very short suspense time.
```

```
            if (_msUntilTimeout > 10) {
              // Instead of committing the fallback immediately, wait for more data
              // to arrive.
              root.timeoutHandle = scheduleTimeout(commitRoot.bind(null, root,
workInProgressRootRecoverableErrors, workInProgressTransitions), _msUntilTimeout);
              break;
            }
          } // Commit the placeholder.


          commitRoot(root, workInProgressRootRecoverableErrors,
workInProgressTransitions);
          break;
        }

      case RootCompleted:
        {
          // The work completed. Ready to commit.
          commitRoot(root, workInProgressRootRecoverableErrors,
workInProgressTransitions);
          break;
        }

      default:
        {
          throw new Error('Unknown root exit status.');
        }
    }
  }

  function isRenderConsistentWithExternalStores(finishedWork) {
    // Search the rendered tree for external store reads, and check whether the
    // stores were mutated in a concurrent event. Intentionally using an iterative
    // loop instead of recursion so we can exit early.
    var node = finishedWork;

    while (true) {
      if (node.flags & StoreConsistency) {
        var updateQueue = node.updateQueue;

        if (updateQueue !== null) {
          var checks = updateQueue.stores;

          if (checks !== null) {
            for (var i = 0; i < checks.length; i++) {
              var check = checks[i];
              var getSnapshot = check.getSnapshot;
              var renderedValue = check.value;

              try {
                if (!objectIs(getSnapshot(), renderedValue)) {
                  // Found an inconsistent store.
                  return false;
                }
              } catch (error) {
                // If `getSnapshot` throws, return `false`. This will schedule
                // a re-render, and the error will be rethrown during render.
                return false;
              }
            }
          }
        }
      }

      var child = node.child;

      if (node.subtreeFlags & StoreConsistency && child !== null) {
        child.return = node;
        node = child;
```

```
          continue;
        }

        if (node === finishedWork) {
          return true;
        }

        while (node.sibling === null) {
          if (node.return === null || node.return === finishedWork) {
            return true;
          }

          node = node.return;
        }

        node.sibling.return = node.return;
        node = node.sibling;
      } // Flow doesn't know this is unreachable, but eslint does
      // eslint-disable-next-line no-unreachable


      return true;
    }

    function markRootSuspended$1(root, suspendedLanes) {
      // When suspending, we should always exclude lanes that were pinged or (more
      // rarely, since we try to avoid it) updated during the render phase.
      // TODO: Lol maybe there's a better way to factor this besides this
      // obnoxiously named function :)
      suspendedLanes = removeLanes(suspendedLanes, workInProgressRootPingedLanes);
      suspendedLanes = removeLanes(suspendedLanes,
  workInProgressRootInterleavedUpdatedLanes);
      markRootSuspended(root, suspendedLanes);
    } // This is the entry point for synchronous tasks that don't go
    // through Scheduler


    function performSyncWorkOnRoot(root) {
      {
        syncNestedUpdateFlag();
      }

      if ((executionContext & (RenderContext | CommitContext)) !== NoContext) {
        throw new Error('Should not already be working.');
      }

      flushPassiveEffects();
      var lanes = getNextLanes(root, NoLanes);

      if (!includesSomeLane(lanes, SyncLane)) {
        // There's no remaining sync work left.
        ensureRootIsScheduled(root, now());
        return null;
      }

      var exitStatus = renderRootSync(root, lanes);

      if (root.tag !== LegacyRoot && exitStatus === RootErrored) {
        // If something threw an error, try rendering one more time. We'll render
        // synchronously to block concurrent data mutations, and we'll includes
        // all pending updates are included. If it still fails after the second
        // attempt, we'll give up and commit the resulting tree.
        var errorRetryLanes = getLanesToRetrySynchronouslyOnError(root);

        if (errorRetryLanes !== NoLanes) {
          lanes = errorRetryLanes;
          exitStatus = recoverFromConcurrentError(root, errorRetryLanes);
        }
      }
      if (exitStatus === RootFatalErrored) {
        var fatalError = workInProgressRootFatalError;
        prepareFreshStack(root, NoLanes);
        markRootSuspended$1(root, lanes);
        ensureRootIsScheduled(root, now());
        throw fatalError;
      }

      if (exitStatus === RootDidNotComplete) {
        throw new Error('Root did not complete. This is a bug in React.');
      } // We now have a consistent tree. Because this is a sync render, we
      // will commit it even if something suspended.


      var finishedWork = root.current.alternate;
      root.finishedWork = finishedWork;
      root.finishedLanes = lanes;
      commitRoot(root, workInProgressRootRecoverableErrors, workInProgressTransitions); //
  Before exiting, make sure there's a callback scheduled for the next
      // pending level.

      ensureRootIsScheduled(root, now());
      return null;
    }

    function flushRoot(root, lanes) {
      if (lanes !== NoLanes) {
        markRootEntangled(root, mergeLanes(lanes, SyncLane));
        ensureRootIsScheduled(root, now());

        if ((executionContext & (RenderContext | CommitContext)) === NoContext) {
          resetRenderTimer();
          flushSyncCallbacks();
        }
      }
    }
    function batchedUpdates$1(fn, a) {
      var prevExecutionContext = executionContext;
      executionContext |= BatchedContext;

      try {
        return fn(a);
      } finally {
        executionContext = prevExecutionContext; // If there were legacy sync updates,
  flush them at the end of the outer
        // most batchedUpdates-like method.

        if (executionContext === NoContext && // Treat `act` as if it's inside
  `batchedUpdates`, even in legacy mode.
        !( ReactCurrentActQueue$1.isBatchingLegacy)) {
          resetRenderTimer();
          flushSyncCallbacksOnlyInLegacyMode();
        }
      }
    }
    function discreteUpdates(fn, a, b, c, d) {
      var previousPriority = getCurrentUpdatePriority();
      var prevTransition = ReactCurrentBatchConfig$3.transition;

      try {
        ReactCurrentBatchConfig$3.transition = null;
        setCurrentUpdatePriority(DiscreteEventPriority);
        return fn(a, b, c, d);
      } finally {
        setCurrentUpdatePriority(previousPriority);
        ReactCurrentBatchConfig$3.transition = prevTransition;

        if (executionContext === NoContext) {
          resetRenderTimer();
        }
      }
    } // Overload the definition to the two valid signatures.
    // Warning, this opts-out of checking the function body.

    // eslint-disable-next-line no-redeclare
    function flushSync(fn) {
      // In legacy mode, we flush pending passive effects at the beginning of the
      // next event, not at the end of the previous one.
      if (rootWithPendingPassiveEffects !== null && rootWithPendingPassiveEffects.tag ===
  LegacyRoot && (executionContext & (RenderContext | CommitContext)) === NoContext) {
        flushPassiveEffects();
      }

      var prevExecutionContext = executionContext;
      executionContext |= BatchedContext;
      var prevTransition = ReactCurrentBatchConfig$3.transition;
      var previousPriority = getCurrentUpdatePriority();

      try {
        ReactCurrentBatchConfig$3.transition = null;
        setCurrentUpdatePriority(DiscreteEventPriority);

        if (fn) {
          return fn();
        } else {
          return undefined;
        }
      } finally {
        setCurrentUpdatePriority(previousPriority);
        ReactCurrentBatchConfig$3.transition = prevTransition;
        executionContext = prevExecutionContext; // Flush the immediate callbacks that were
  scheduled during this batch.
        // Note that this will happen even if batchedUpdates is higher up
        // the stack.

        if ((executionContext & (RenderContext | CommitContext)) === NoContext) {
          flushSyncCallbacks();
        }
      }
    }
    function isAlreadyRendering() {
      // Used by the renderer to print a warning if certain APIs are called from
      // the wrong context.
      return  (executionContext & (RenderContext | CommitContext)) !== NoContext;
    }
    function pushRenderLanes(fiber, lanes) {
      push(subtreeRenderLanesCursor, subtreeRenderLanes, fiber);
      subtreeRenderLanes = mergeLanes(subtreeRenderLanes, lanes);
      workInProgressRootIncludedLanes = mergeLanes(workInProgressRootIncludedLanes, lanes);
    }
    function popRenderLanes(fiber) {
      subtreeRenderLanes = subtreeRenderLanesCursor.current;
      pop(subtreeRenderLanesCursor, fiber);
    }

    function prepareFreshStack(root, lanes) {
      root.finishedWork = null;
      root.finishedLanes = NoLanes;
      var timeoutHandle = root.timeoutHandle;

      if (timeoutHandle !== noTimeout) {
        // The root previous suspended and scheduled a timeout to commit a fallback
        // state. Now that we have additional work, cancel the timeout.
        root.timeoutHandle = noTimeout; // $FlowFixMe Complains noTimeout is not a
  TimeoutID, despite the check above

        cancelTimeout(timeoutHandle);
      }

      if (workInProgress !== null) {
        var interruptedWork = workInProgress.return;

        while (interruptedWork !== null) {
          var current = interruptedWork.alternate;
          unwindInterruptedWork(current, interruptedWork);
          interruptedWork = interruptedWork.return;
        }
      }

      workInProgressRoot = root;
      var rootWorkInProgress = createWorkInProgress(root.current, null);
      workInProgress = rootWorkInProgress;
      workInProgressRootRenderLanes = subtreeRenderLanes = workInProgressRootIncludedLanes
  = lanes;
      workInProgressRootExitStatus = RootInProgress;
      workInProgressRootFatalError = null;
      workInProgressRootSkippedLanes = NoLanes;
      workInProgressRootInterleavedUpdatedLanes = NoLanes;
      workInProgressRootPingedLanes = NoLanes;
      workInProgressRootConcurrentErrors = null;
      workInProgressRootRecoverableErrors = null;
      finishQueueingConcurrentUpdates();

      {
        ReactStrictModeWarnings.discardPendingWarnings();
      }

      return rootWorkInProgress;
    }

    function handleError(root, thrownValue) {
      do {
        var erroredWork = workInProgress;

        try {
          // Reset module-level state that was set during the render phase.
          resetContextDependencies();
          resetHooksAfterThrow();
          resetCurrentFiber(); // TODO: I found and added this missing line while
  investigating a
          // separate issue. Write a regression test using string refs.

          ReactCurrentOwner$2.current = null;

          if (erroredWork === null || erroredWork.return === null) {
            // Expected to be working on a non-root fiber. This is a fatal error
            // because there's no ancestor that can handle it; the root is
            // supposed to capture all errors that weren't caught by an error
            // boundary.
            workInProgressRootExitStatus = RootFatalErrored;
            workInProgressRootFatalError = thrownValue; // Set `workInProgress` to null.
  This represents advancing to the next
            // sibling, or the parent if there are no siblings. But since the root
            // has no siblings nor a parent, we set it to null. Usually this is
            // handled by `completeUnitOfWork` or `unwindWork`, but since we're
            // intentionally not calling those, we need set it here.
            // TODO: Consider calling `unwindWork` to pop the contexts.

            workInProgress = null;
            return;
          }

          if (enableProfilerTimer && erroredWork.mode & ProfileMode) {
            // Record the time spent rendering before an error was thrown. This
            // avoids inaccurate Profiler durations in the case of a
            // suspended render.
            stopProfilerTimerIfRunningAndRecordDelta(erroredWork, true);
          }

          if (enableSchedulingProfiler) {
            markComponentRenderStopped();

            if (thrownValue !== null && typeof thrownValue === 'object' && typeof
  thrownValue.then === 'function') {
              var wakeable = thrownValue;
              markComponentSuspended(erroredWork, wakeable, workInProgressRootRenderLanes);
            } else {
              markComponentErrored(erroredWork, thrownValue,
  workInProgressRootRenderLanes);
            }
          }

          throwException(root, erroredWork.return, erroredWork, thrownValue,
  workInProgressRootRenderLanes);
          completeUnitOfWork(erroredWork);
        } catch (yetAnotherThrownValue) {
          // Something in the return path also threw.
          thrownValue = yetAnotherThrownValue;

          if (workInProgress === erroredWork && erroredWork !== null) {
            // If this boundary has already errored, then we had trouble processing
            // the error. Bubble it to the next boundary.
            erroredWork = erroredWork.return;
            workInProgress = erroredWork;
          } else {
            erroredWork = workInProgress;
          }

          continue;
        } // Return to the normal work loop.


        return;
      } while (true);
    }

    function pushDispatcher() {
      var prevDispatcher = ReactCurrentDispatcher$2.current;
      ReactCurrentDispatcher$2.current = ContextOnlyDispatcher;

      if (prevDispatcher === null) {
        // The React isomorphic package does not include a default dispatcher.
        // Instead the first renderer will lazily attach one, in order to give
        // nicer error messages.
        return ContextOnlyDispatcher;
      } else {
        return prevDispatcher;
      }
    }

    function popDispatcher(prevDispatcher) {
      ReactCurrentDispatcher$2.current = prevDispatcher;
    }

    function markCommitTimeOfFallback() {
      globalMostRecentFallbackTime = now();
    }
    function markSkippedUpdateLanes(lane) {
      workInProgressRootSkippedLanes = mergeLanes(lane, workInProgressRootSkippedLanes);
    }
    function renderDidSuspend() {
      if (workInProgressRootExitStatus === RootInProgress) {
        workInProgressRootExitStatus = RootSuspended;
      }
    }
    function renderDidSuspendDelayIfPossible() {
      if (workInProgressRootExitStatus === RootInProgress || workInProgressRootExitStatus
  === RootSuspended || workInProgressRootExitStatus === RootErrored) {
        workInProgressRootExitStatus = RootSuspendedWithDelay;
      } // Check if there are updates that we skipped tree that might have unblocked
      // this render.


      if (workInProgressRoot !== null &&
  (includesNonIdleWork(workInProgressRootSkippedLanes) ||
  includesNonIdleWork(workInProgressRootInterleavedUpdatedLanes))) {
        // Mark the current render as suspended so that we switch to working on
        // the updates that were skipped. Usually we only suspend at the end of
        // the render phase.
        // TODO: We should probably always mark the root as suspended immediately
        // (inside this function), since by suspending at the end of the render
        // phase introduces a potential mistake where we suspend lanes that were
        // pinged or updated while we were rendering.
        markRootSuspended$1(workInProgressRoot, workInProgressRootRenderLanes);
      }
    }
    function renderDidError(error) {
      if (workInProgressRootExitStatus !== RootSuspendedWithDelay) {
        workInProgressRootExitStatus = RootErrored;
      }

      if (workInProgressRootConcurrentErrors === null) {
        workInProgressRootConcurrentErrors = [error];
      } else {
        workInProgressRootConcurrentErrors.push(error);
      }
    } // Called during render to determine if anything has suspended.
    // Returns false if we're not sure.

    function renderHasNotSuspendedYet() {
      // If something errored or completed, we can't really be sure,
      // so those are false.
      return workInProgressRootExitStatus === RootInProgress;
    }

    function renderRootSync(root, lanes) {
      var prevExecutionContext = executionContext;
      executionContext |= RenderContext;
      var prevDispatcher = pushDispatcher(); // If the root or lanes have changed, throw
  out the existing stack
      // and prepare a fresh one. Otherwise we'll continue where we left off.

      if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
        {
          if (isDevToolsPresent) {
            var memoizedUpdaters = root.memoizedUpdaters;

            if (memoizedUpdaters.size > 0) {
              restorePendingUpdaters(root, workInProgressRootRenderLanes);
              memoizedUpdaters.clear();
            } // At this point, move Fibers that scheduled the upcoming work from the Map
  to the Set.
            // If we bailout on this work, we'll move them back (like above).
            // It's important to move them now in case the work spawns more work at the
  same priority with different updaters.
            // That way we can keep the current update and future updates separate.


            movePendingFibersToMemoized(root, lanes);
          }
        }

        workInProgressTransitions = getTransitionsForLanes();
        prepareFreshStack(root, lanes);
      }

      {
        markRenderStarted(lanes);
      }

      do {
        try {
          workLoopSync();
          break;
        } catch (thrownValue) {
          handleError(root, thrownValue);
        }
      } while (true);

      resetContextDependencies();
      executionContext = prevExecutionContext;
      popDispatcher(prevDispatcher);

      if (workInProgress !== null) {
        // This is a sync render, so we should have finished the whole tree.
        throw new Error('Cannot commit an incomplete root. This error is likely caused by a
  ' + 'bug in React. Please file an issue.');
      }

      {
        markRenderStopped();
      } // Set this to null to indicate there's no in-progress render.


      workInProgressRoot = null;
      workInProgressRootRenderLanes = NoLanes;
      return workInProgressRootExitStatus;
    } // The work loop is an extremely hot path. Tell Closure not to inline it.

    /** @noinline */


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

    function renderRootConcurrent(root, lanes) {
      var prevExecutionContext = executionContext;
      executionContext |= RenderContext;
      var prevDispatcher = pushDispatcher(); // If the root or lanes have changed, throw
  out the existing stack
      // and prepare a fresh one. Otherwise we'll continue where we left off.

      if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
        {
          if (isDevToolsPresent) {
            var memoizedUpdaters = root.memoizedUpdaters;

            if (memoizedUpdaters.size > 0) {
              restorePendingUpdaters(root, workInProgressRootRenderLanes);
              memoizedUpdaters.clear();
            } // At this point, move Fibers that scheduled the upcoming work from the Map
  to the Set.
            // If we bailout on this work, we'll move them back (like above).
            // It's important to move them now in case the work spawns more work at the
  same priority with different updaters.
            // That way we can keep the current update and future updates separate.


            movePendingFibersToMemoized(root, lanes);
          }
        }

        workInProgressTransitions = getTransitionsForLanes();
        resetRenderTimer();
        prepareFreshStack(root, lanes);
      }

      {
        markRenderStarted(lanes);
      }

      do {
        try {
          workLoopConcurrent();
          break;
        } catch (thrownValue) {
          handleError(root, thrownValue);
        }
      } while (true);

      resetContextDependencies();
      popDispatcher(prevDispatcher);
      executionContext = prevExecutionContext;


      if (workInProgress !== null) {
        // Still work remaining.
        {
          markRenderYielded();
        }

        return RootInProgress;
      } else {
        // Completed the tree.
        {
          markRenderStopped();
        } // Set this to null to indicate there's no in-progress render.


        workInProgressRoot = null;
        workInProgressRootRenderLanes = NoLanes; // Return the final exit status.

        return workInProgressRootExitStatus;
      }
    }
    /** @noinline */


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

    function performUnitOfWork(unitOfWork) {
      // 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.
      var current = unitOfWork.alternate;
      setCurrentFiber(unitOfWork);
      var next;

      if ( (unitOfWork.mode & ProfileMode) !== NoMode) {
        startProfilerTimer(unitOfWork);
        next = beginWork$1(current, unitOfWork, subtreeRenderLanes);
        stopProfilerTimerIfRunningAndRecordDelta(unitOfWork, true);
      } else {
        next = beginWork$1(current, unitOfWork, subtreeRenderLanes);
      }

      resetCurrentFiber();
      unitOfWork.memoizedProps = unitOfWork.pendingProps;

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

      ReactCurrentOwner$2.current = null;
    }

    function completeUnitOfWork(unitOfWork) {
      // Attempt to complete the current unit of work, then move to the next
      // sibling. If there are no more siblings, return to the parent fiber.
      var completedWork = unitOfWork;

      do {
        // 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.
        var current = completedWork.alternate;
        var returnFiber = completedWork.return; // Check if the work completed or if
  something threw.

        if ((completedWork.flags & Incomplete) === NoFlags) {
          setCurrentFiber(completedWork);
          var next = void 0;

          if ( (completedWork.mode & ProfileMode) === NoMode) {
            next = completeWork(current, completedWork, subtreeRenderLanes);
          } else {
            startProfilerTimer(completedWork);
            next = completeWork(current, completedWork, subtreeRenderLanes); // Update
  render duration assuming we didn't error.

            stopProfilerTimerIfRunningAndRecordDelta(completedWork, false);
          }

          resetCurrentFiber();

          if (next !== null) {
            // Completing this fiber spawned new work. Work on that next.
            workInProgress = next;
            return;
          }
        } else {
          // This fiber did not complete because something threw. Pop values off
          // the stack without entering the complete phase. If this is a boundary,
          // capture values if possible.
          var _next = unwindWork(current, completedWork); // Because this fiber did not
  complete, don't reset its lanes.


          if (_next !== null) {
            // If completing this work spawned new work, do that next. We'll come
            // back here again.
            // Since we're restarting, remove anything that is not a host effect
            // from the effect tag.
            _next.flags &= HostEffectMask;
            workInProgress = _next;
            return;
          }

          if ( (completedWork.mode & ProfileMode) !== NoMode) {
            // Record the render duration for the fiber that errored.
            stopProfilerTimerIfRunningAndRecordDelta(completedWork, false); // Include the
  time spent working on failed children before continuing.

            var actualDuration = completedWork.actualDuration;
            var child = completedWork.child;

            while (child !== null) {
              actualDuration += child.actualDuration;
              child = child.sibling;
            }

            completedWork.actualDuration = actualDuration;
          }

          if (returnFiber !== null) {
            // Mark the parent fiber as incomplete and clear its subtree flags.
            returnFiber.flags |= Incomplete;
            returnFiber.subtreeFlags = NoFlags;
            returnFiber.deletions = null;
          } else {
            // We've unwound all the way to the root.
            workInProgressRootExitStatus = RootDidNotComplete;
            workInProgress = null;
            return;
          }
        }

        var siblingFiber = completedWork.sibling;

        if (siblingFiber !== null) {
          // If there is more work to do in this returnFiber, do that next.
          workInProgress = siblingFiber;
          return;
        } // Otherwise, return to the parent


        completedWork = returnFiber; // Update the next thing we're working on in case
  something throws.

        workInProgress = completedWork;
      } while (completedWork !== null); // We've reached the root.


      if (workInProgressRootExitStatus === RootInProgress) {
        workInProgressRootExitStatus = RootCompleted;
      }
    }

    function commitRoot(root, recoverableErrors, transitions) {
      // TODO: This no longer makes any sense. We already wrap the mutation and
      // layout phases. Should be able to remove.
      var previousUpdateLanePriority = getCurrentUpdatePriority();
      var prevTransition = ReactCurrentBatchConfig$3.transition;

      try {
        ReactCurrentBatchConfig$3.transition = null;
        setCurrentUpdatePriority(DiscreteEventPriority);
        commitRootImpl(root, recoverableErrors, transitions, previousUpdateLanePriority);
      } finally {
        ReactCurrentBatchConfig$3.transition = prevTransition;
        setCurrentUpdatePriority(previousUpdateLanePriority);
      }

      return null;
    }

    function commitRootImpl(root, recoverableErrors, transitions, renderPriorityLevel) {
      do {
        // `flushPassiveEffects` will call `flushSyncUpdateQueue` at the end, which
        // means `flushPassiveEffects` will sometimes result in additional
        // passive effects. So we need to keep flushing in a loop until there are
        // no more pending effects.
        // TODO: Might be better if `flushPassiveEffects` did not automatically
        // flush synchronous work at the end, to avoid factoring hazards like this.
        flushPassiveEffects();
      } while (rootWithPendingPassiveEffects !== null);

      flushRenderPhaseStrictModeWarningsInDEV();

      if ((executionContext & (RenderContext | CommitContext)) !== NoContext) {
        throw new Error('Should not already be working.');
      }

      var finishedWork = root.finishedWork;
      var lanes = root.finishedLanes;

      {
        markCommitStarted(lanes);
      }

      if (finishedWork === null) {

        {
          markCommitStopped();
        }

        return null;
      } else {
        {
          if (lanes === NoLanes) {
            error('root.finishedLanes should not be empty during a commit. This is a ' +
  'bug in React.');
          }
        }
      }

      root.finishedWork = null;
      root.finishedLanes = NoLanes;

      if (finishedWork === root.current) {
        throw new Error('Cannot commit the same tree as before. This error is likely caused
  by ' + 'a bug in React. Please file an issue.');
      } // commitRoot never returns a continuation; it always finishes synchronously.
      // So we can clear these now to allow a new callback to be scheduled.


      root.callbackNode = null;
      root.callbackPriority = NoLane; // Update the first and last pending times on this
  root. The new first
      // pending time is whatever is left on the root fiber.

      var remainingLanes = mergeLanes(finishedWork.lanes, finishedWork.childLanes);
      markRootFinished(root, remainingLanes);

      if (root === workInProgressRoot) {
        // We can reset these now that they are finished.
        workInProgressRoot = null;
        workInProgress = null;
        workInProgressRootRenderLanes = NoLanes;
      } // If there are pending passive effects, schedule a callback to process them.
      // Do this as early as possible, so it is queued before anything else that
      // might get scheduled in the commit phase. (See #16714.)
      // TODO: Delete all other places that schedule the passive effect callback
      // They're redundant.


      if ((finishedWork.subtreeFlags & PassiveMask) !== NoFlags || (finishedWork.flags &
  PassiveMask) !== NoFlags) {
        if (!rootDoesHavePassiveEffects) {
          rootDoesHavePassiveEffects = true;
          // to store it in pendingPassiveTransitions until they get processed
          // We need to pass this through as an argument to commitRoot
          // because workInProgressTransitions might have changed between
          // the previous render and commit if we throttle the commit
          // with setTimeout

          pendingPassiveTransitions = transitions;
          scheduleCallback$1(NormalPriority, function () {
            flushPassiveEffects(); // This render triggered passive effects: release the
  root cache pool
            // *after* passive effects fire to avoid freeing a cache pool that may
            // be referenced by a node in the tree (HostRoot, Cache boundary etc)

            return null;
          });
        }
      } // Check if there are any effects in the whole tree.
      // TODO: This is left over from the effect list implementation, where we had
      // to check for the existence of `firstEffect` to satisfy Flow. I think the
      // only other reason this optimization exists is because it affects profiling.
      // Reconsider whether this is necessary.


      var subtreeHasEffects = (finishedWork.subtreeFlags & (BeforeMutationMask |
  MutationMask | LayoutMask | PassiveMask)) !== NoFlags;
      var rootHasEffect = (finishedWork.flags & (BeforeMutationMask | MutationMask |
  LayoutMask | PassiveMask)) !== NoFlags;

      if (subtreeHasEffects || rootHasEffect) {
        var prevTransition = ReactCurrentBatchConfig$3.transition;
        ReactCurrentBatchConfig$3.transition = null;
        var previousPriority = getCurrentUpdatePriority();
        setCurrentUpdatePriority(DiscreteEventPriority);
        var prevExecutionContext = executionContext;
        executionContext |= CommitContext; // Reset this to null before calling lifecycles

        ReactCurrentOwner$2.current = null; // The commit phase is broken into several sub-
  phases. We do a separate pass
        // of the effect list for each phase: all mutation effects come before all
        // layout effects, and so on.
        // The first phase a "before mutation" phase. We use this phase to read the
        // state of the host tree right before we mutate it. This is where
        // getSnapshotBeforeUpdate is called.

        var shouldFireAfterActiveInstanceBlur = commitBeforeMutationEffects(root,
  finishedWork);

        {
          // Mark the current commit time to be shared by all Profilers in this
          // batch. This enables them to be grouped later.
          recordCommitTime();
        }


        commitMutationEffects(root, finishedWork, lanes);

        resetAfterCommit(root.containerInfo); // The work-in-progress tree is now the
  current tree. This must come after
        // the mutation phase, so that the previous tree is still current during
        // componentWillUnmount, but before the layout phase, so that the finished
        // work is current during componentDidMount/Update.

        root.current = finishedWork; // The next phase is the layout phase, where we call
  effects that read

        {
          markLayoutEffectsStarted(lanes);
        }

        commitLayoutEffects(finishedWork, root, lanes);

        {
          markLayoutEffectsStopped();
        }
        // opportunity to paint.


        requestPaint();
        executionContext = prevExecutionContext; // Reset the priority to the previous non-
  sync value.

        setCurrentUpdatePriority(previousPriority);
        ReactCurrentBatchConfig$3.transition = prevTransition;
      } else {
        // No effects.
        root.current = finishedWork; // Measure these anyway so the flamegraph explicitly
  shows that there were
        // no effects.
        // TODO: Maybe there's a better way to report this.

        {
          recordCommitTime();
        }
      }

      var rootDidHavePassiveEffects = rootDoesHavePassiveEffects;

      if (rootDoesHavePassiveEffects) {
        // This commit has passive effects. Stash a reference to them. But don't
        // schedule a callback until after flushing layout work.
        rootDoesHavePassiveEffects = false;
        rootWithPendingPassiveEffects = root;
        pendingPassiveEffectsLanes = lanes;
      } else {

        {
          nestedPassiveUpdateCount = 0;
          rootWithPassiveNestedUpdates = null;
        }
      } // Read this again, since an effect might have updated it


      remainingLanes = root.pendingLanes; // Check if there's remaining work on this root
      // TODO: This is part of the `componentDidCatch` implementation. Its purpose
      // is to detect whether something might have called setState inside
      // `componentDidCatch`. The mechanism is known to be flawed because `setState`
      // inside `componentDidCatch` is itself flawed — that's why we recommend
      // `getDerivedStateFromError` instead. However, it could be improved by
      // checking if remainingLanes includes Sync work, instead of whether there's
      // any work remaining at all (which would also include stuff like Suspense
      // retries or transitions). It's been like this for a while, though, so fixing
      // it probably isn't that urgent.

      if (remainingLanes === NoLanes) {
        // If there's no remaining work, we can clear the set of already failed
        // error boundaries.
        legacyErrorBoundariesThatAlreadyFailed = null;
      }

      {
        if (!rootDidHavePassiveEffects) {
          commitDoubleInvokeEffectsInDEV(root.current, false);
        }
      }
```

```
      onCommitRoot(finishedWork.stateNode, renderPriorityLevel);

      {
        if (isDevToolsPresent) {
          root.memoizedUpdaters.clear();
        }
      }

      {
        onCommitRoot$1();
      } // Always call this before exiting `commitRoot`, to ensure that any
      // additional work on this root is scheduled.


      ensureRootIsScheduled(root, now());

      if (recoverableErrors !== null) {
        // There were errors during this render, but recovered from them without
        // needing to surface it to the UI. We log them here.
        var onRecoverableError = root.onRecoverableError;

        for (var i = 0; i < recoverableErrors.length; i++) {
          var recoverableError = recoverableErrors[i];
          var componentStack = recoverableError.stack;
          var digest = recoverableError.digest;
          onRecoverableError(recoverableError.value, {
            componentStack: componentStack,
            digest: digest
          });
        }
      }

      if (hasUncaughtError) {
        hasUncaughtError = false;
        var error$1 = firstUncaughtError;
        firstUncaughtError = null;
        throw error$1;
      } // If the passive effects are the result of a discrete render, flush them
      // synchronously at the end of the current task so that the result is
      // immediately observable. Otherwise, we assume that they are not
      // order-dependent and do not need to be observed by external systems, so we
      // can wait until after paint.
      // TODO: We can optimize this by not scheduling the callback earlier. Since we
      // currently schedule the callback in multiple places, will wait until those
      // are consolidated.


      if (includesSomeLane(pendingPassiveEffectsLanes, SyncLane) && root.tag !==
  LegacyRoot) {
        flushPassiveEffects();
      } // Read this again, since a passive effect might have updated it


      remainingLanes = root.pendingLanes;

      if (includesSomeLane(remainingLanes, SyncLane)) {
        {
          markNestedUpdateScheduled();
        } // Count the number of times the root synchronously re-renders without
        // finishing. If there are too many, it indicates an infinite update loop.


        if (root === rootWithNestedUpdates) {
          nestedUpdateCount++;
        } else {
          nestedUpdateCount = 0;
          rootWithNestedUpdates = root;
        }
      } else {
```

```
      nestedUpdateCount = 0;
    } // If layout work was scheduled, flush it now.


    flushSyncCallbacks();

    {
      markCommitStopped();
    }

    return null;
  }

  function flushPassiveEffects() {
    // Returns whether passive effects were flushed.
    // TODO: Combine this check with the one in flushPassiveEFfectsImpl. We should
    // probably just combine the two functions. I believe they were only separate
    // in the first place because we used to wrap it with
    // `Scheduler.runWithPriority`, which accepts a function. But now we track the
    // priority within React itself, so we can mutate the variable directly.
    if (rootWithPendingPassiveEffects !== null) {
      var renderPriority = lanesToEventPriority(pendingPassiveEffectsLanes);
      var priority = lowerEventPriority(DefaultEventPriority, renderPriority);
      var prevTransition = ReactCurrentBatchConfig$3.transition;
      var previousPriority = getCurrentUpdatePriority();

      try {
        ReactCurrentBatchConfig$3.transition = null;
        setCurrentUpdatePriority(priority);
        return flushPassiveEffectsImpl();
      } finally {
        setCurrentUpdatePriority(previousPriority);
        ReactCurrentBatchConfig$3.transition = prevTransition; // Once passive effects
have run for the tree - giving components a
      }
    }

    return false;
  }
  function enqueuePendingPassiveProfilerEffect(fiber) {
    {
      pendingPassiveProfilerEffects.push(fiber);

      if (!rootDoesHavePassiveEffects) {
        rootDoesHavePassiveEffects = true;
        scheduleCallback$1(NormalPriority, function () {
          flushPassiveEffects();
          return null;
        });
      }
    }
  }

  function flushPassiveEffectsImpl() {
    if (rootWithPendingPassiveEffects === null) {
      return false;
    } // Cache and clear the transitions flag


    var transitions = pendingPassiveTransitions;
    pendingPassiveTransitions = null;
    var root = rootWithPendingPassiveEffects;
    var lanes = pendingPassiveEffectsLanes;
    rootWithPendingPassiveEffects = null; // TODO: This is sometimes out of sync with
rootWithPendingPassiveEffects.
    // Figure out why and fix it. It's not causing any known issues (probably
    // because it's only used for profiling), but it's a refactor hazard.

    pendingPassiveEffectsLanes = NoLanes;
    if ((executionContext & (RenderContext | CommitContext)) !== NoContext) {
      throw new Error('Cannot flush passive effects while already rendering.');
    }

    {
      isFlushingPassiveEffects = true;
      didScheduleUpdateDuringPassiveEffects = false;
    }

    {
      markPassiveEffectsStarted(lanes);
    }

    var prevExecutionContext = executionContext;
    executionContext |= CommitContext;
    commitPassiveUnmountEffects(root.current);
    commitPassiveMountEffects(root, root.current, lanes, transitions); // TODO: Move to
commitPassiveMountEffects

    {
      var profilerEffects = pendingPassiveProfilerEffects;
      pendingPassiveProfilerEffects = [];

      for (var i = 0; i < profilerEffects.length; i++) {
        var _fiber = profilerEffects[i];
        commitPassiveEffectDurations(root, _fiber);
      }
    }

    {
      markPassiveEffectsStopped();
    }

    {
      commitDoubleInvokeEffectsInDEV(root.current, true);
    }

    executionContext = prevExecutionContext;
    flushSyncCallbacks();

    {
      // If additional passive effects were scheduled, increment a counter. If this
      // exceeds the limit, we'll fire a warning.
      if (didScheduleUpdateDuringPassiveEffects) {
        if (root === rootWithPassiveNestedUpdates) {
          nestedPassiveUpdateCount++;
        } else {
          nestedPassiveUpdateCount = 0;
          rootWithPassiveNestedUpdates = root;
        }
      } else {
        nestedPassiveUpdateCount = 0;
      }

      isFlushingPassiveEffects = false;
      didScheduleUpdateDuringPassiveEffects = false;
    } // TODO: Move to commitPassiveMountEffects


    onPostCommitRoot(root);

    {
      var stateNode = root.current.stateNode;
      stateNode.effectDuration = 0;
      stateNode.passiveEffectDuration = 0;
    }

    return true;
  }

  function isAlreadyFailedLegacyErrorBoundary(instance) {
    return legacyErrorBoundariesThatAlreadyFailed !== null &&
legacyErrorBoundariesThatAlreadyFailed.has(instance);
  }
  function markLegacyErrorBoundaryAsFailed(instance) {
    if (legacyErrorBoundariesThatAlreadyFailed === null) {
      legacyErrorBoundariesThatAlreadyFailed = new Set([instance]);
    } else {
      legacyErrorBoundariesThatAlreadyFailed.add(instance);
    }
  }

  function prepareToThrowUncaughtError(error) {
    if (!hasUncaughtError) {
      hasUncaughtError = true;
      firstUncaughtError = error;
    }
  }

  var onUncaughtError = prepareToThrowUncaughtError;

  function captureCommitPhaseErrorOnRoot(rootFiber, sourceFiber, error) {
    var errorInfo = createCapturedValueAtFiber(error, sourceFiber);
    var update = createRootErrorUpdate(rootFiber, errorInfo, SyncLane);
    var root = enqueueUpdate(rootFiber, update, SyncLane);
    var eventTime = requestEventTime();

    if (root !== null) {
      markRootUpdated(root, SyncLane, eventTime);
      ensureRootIsScheduled(root, eventTime);
    }
  }

  function captureCommitPhaseError(sourceFiber, nearestMountedAncestor, error$1) {
    {
      reportUncaughtErrorInDEV(error$1);
      setIsRunningInsertionEffect(false);
    }

    if (sourceFiber.tag === HostRoot) {
      // Error was thrown at the root. There is no parent, so the root
      // itself should capture it.
      captureCommitPhaseErrorOnRoot(sourceFiber, sourceFiber, error$1);
      return;
    }

    var fiber = null;

    {
      fiber = nearestMountedAncestor;
    }

    while (fiber !== null) {
      if (fiber.tag === HostRoot) {
        captureCommitPhaseErrorOnRoot(fiber, sourceFiber, error$1);
        return;
      } else if (fiber.tag === ClassComponent) {
        var ctor = fiber.type;
        var instance = fiber.stateNode;

        if (typeof ctor.getDerivedStateFromError === 'function' || typeof
instance.componentDidCatch === 'function' &&
!isAlreadyFailedLegacyErrorBoundary(instance)) {
          var errorInfo = createCapturedValueAtFiber(error$1, sourceFiber);
          var update = createClassErrorUpdate(fiber, errorInfo, SyncLane);
          var root = enqueueUpdate(fiber, update, SyncLane);
          var eventTime = requestEventTime();
          if (root !== null) {
            markRootUpdated(root, SyncLane, eventTime);
            ensureRootIsScheduled(root, eventTime);
          }

          return;
        }
      }

      fiber = fiber.return;
    }

    {
      // TODO: Until we re-land skipUnmountedBoundaries (see #20147), this warning
      // will fire for errors that are thrown by destroy functions inside deleted
      // trees. What it should instead do is propagate the error to the parent of
      // the deleted tree. In the meantime, do not add this warning to the
      // allowlist; this is only for our internal use.
      error('Internal React error: Attempted to capture a commit phase error ' + 'inside
a detached tree. This indicates a bug in React. Likely ' + 'causes include deleting the
same fiber more than once, committing an ' + 'already-finished tree, or an inconsistent
return pointer.\n\n' + 'Error message:\n\n%s', error$1);
    }
  }
  function pingSuspendedRoot(root, wakeable, pingedLanes) {
    var pingCache = root.pingCache;

    if (pingCache !== null) {
      // The wakeable resolved, so we no longer need to memoize, because it will
      // never be thrown again.
      pingCache.delete(wakeable);
    }

    var eventTime = requestEventTime();
    markRootPinged(root, pingedLanes);
    warnIfSuspenseResolutionNotWrappedWithActDEV(root);

    if (workInProgressRoot === root && isSubsetOfLanes(workInProgressRootRenderLanes,
pingedLanes)) {
      // Received a ping at the same priority level at which we're currently
      // rendering. We might want to restart this render. This should mirror
      // the logic of whether or not a root suspends once it completes.
      // TODO: If we're rendering sync either due to Sync, Batched or expired,
      // we should probably never restart.
      // If we're suspended with delay, or if it's a retry, we'll always suspend
      // so we can always restart.
      if (workInProgressRootExitStatus === RootSuspendedWithDelay ||
workInProgressRootExitStatus === RootSuspended &&
includesOnlyRetries(workInProgressRootRenderLanes) && now() -
globalMostRecentFallbackTime < FALLBACK_THROTTLE_MS) {
        // Restart from the root.
        prepareFreshStack(root, NoLanes);
      } else {
        // Even though we can't restart right now, we might get an
        // opportunity later. So we mark this render as having a ping.
        workInProgressRootPingedLanes = mergeLanes(workInProgressRootPingedLanes,
pingedLanes);
      }
    }

    ensureRootIsScheduled(root, eventTime);
  }

  function retryTimedOutBoundary(boundaryFiber, retryLane) {
    // The boundary fiber (a Suspense component or SuspenseList component)
    // previously was rendered in its fallback state. One of the promises that
    // suspended it has resolved, which means at least part of the tree was
    // likely unblocked. Try rendering again, at a new lanes.
    if (retryLane === NoLane) {
      // TODO: Assign this to `suspenseState.retryLane`? to avoid
      // unnecessary entanglement?
      retryLane = requestRetryLane(boundaryFiber);
    } // TODO: Special case idle priority?


    var eventTime = requestEventTime();
    var root = enqueueConcurrentRenderForLane(boundaryFiber, retryLane);

    if (root !== null) {
      markRootUpdated(root, retryLane, eventTime);
      ensureRootIsScheduled(root, eventTime);
    }
  }

  function retryDehydratedSuspenseBoundary(boundaryFiber) {
    var suspenseState = boundaryFiber.memoizedState;
    var retryLane = NoLane;

    if (suspenseState !== null) {
      retryLane = suspenseState.retryLane;
    }

    retryTimedOutBoundary(boundaryFiber, retryLane);
  }
  function resolveRetryWakeable(boundaryFiber, wakeable) {
    var retryLane = NoLane; // Default

    var retryCache;

    switch (boundaryFiber.tag) {
      case SuspenseComponent:
        retryCache = boundaryFiber.stateNode;
        var suspenseState = boundaryFiber.memoizedState;

        if (suspenseState !== null) {
          retryLane = suspenseState.retryLane;
        }

        break;

      case SuspenseListComponent:
        retryCache = boundaryFiber.stateNode;
        break;

      default:
        throw new Error('Pinged unknown suspense boundary type. ' + 'This is probably a
bug in React.');
    }

    if (retryCache !== null) {
      // The wakeable resolved, so we no longer need to memoize, because it will
      // never be thrown again.
      retryCache.delete(wakeable);
    }

    retryTimedOutBoundary(boundaryFiber, retryLane);
  } // Computes the next Just Noticeable Difference (JND) boundary.
  // The theory is that a person can't tell the difference between small differences in
time.
  // Therefore, if we wait a bit longer than necessary that won't translate to a
noticeable
  // difference in the experience. However, waiting for longer might mean that we can
avoid
  // showing an intermediate loading state. The longer we have already waited, the harder
it
  // is to tell small differences in time. Therefore, the longer we've already waited,
  // the longer we can wait additionally. At some point we have to give up though.
  // We pick a train model where the next boundary commits at a consistent schedule.
  // These particular numbers are vague estimates. We expect to adjust them based on
research.

  function jnd(timeElapsed) {
    return timeElapsed < 120 ? 120 : timeElapsed < 480 ? 480 : timeElapsed < 1080 ? 1080
: timeElapsed < 1920 ? 1920 : timeElapsed < 3000 ? 3000 : timeElapsed < 4320 ? 4320 :
ceil(timeElapsed / 1960) * 1960;
  }

  function checkForNestedUpdates() {
    if (nestedUpdateCount > NESTED_UPDATE_LIMIT) {
      nestedUpdateCount = 0;
      rootWithNestedUpdates = null;
      throw new Error('Maximum update depth exceeded. This can happen when a component '
+ 'repeatedly calls setState inside componentWillUpdate or ' + 'componentDidUpdate. React
limits the number of nested updates to ' + 'prevent infinite loops.');
    }

    {
      if (nestedPassiveUpdateCount > NESTED_PASSIVE_UPDATE_LIMIT) {
        nestedPassiveUpdateCount = 0;
        rootWithPassiveNestedUpdates = null;

        error('Maximum update depth exceeded. This can happen when a component ' + "calls
setState inside useEffect, but useEffect either doesn't " + 'have a dependency array, or
one of the dependencies changes on ' + 'every render.');
      }
    }
  }

  function flushRenderPhaseStrictModeWarningsInDEV() {
    {
      ReactStrictModeWarnings.flushLegacyContextWarning();

      {
        ReactStrictModeWarnings.flushPendingUnsafeLifecycleWarnings();
      }
    }
  }

  function commitDoubleInvokeEffectsInDEV(fiber, hasPassiveEffects) {
    {
      // TODO (StrictEffects) Should we set a marker on the root if it contains strict
effects
      // so we don't traverse unnecessarily? similar to subtreeFlags but just at the root
level.
      // Maybe not a big deal since this is DEV only behavior.
      setCurrentFiber(fiber);
      invokeEffectsInDev(fiber, MountLayoutDev, invokeLayoutEffectUnmountInDEV);

      if (hasPassiveEffects) {
        invokeEffectsInDev(fiber, MountPassiveDev, invokePassiveEffectUnmountInDEV);
      }

      invokeEffectsInDev(fiber, MountLayoutDev, invokeLayoutEffectMountInDEV);

      if (hasPassiveEffects) {
        invokeEffectsInDev(fiber, MountPassiveDev, invokePassiveEffectMountInDEV);
      }

      resetCurrentFiber();
    }
  }

  function invokeEffectsInDev(firstChild, fiberFlags, invokeEffectFn) {
    {
      // We don't need to re-check StrictEffectsMode here.
      // This function is only called if that check has already passed.
      var current = firstChild;
      var subtreeRoot = null;

      while (current !== null) {
        var primarySubtreeFlag = current.subtreeFlags & fiberFlags;

        if (current !== subtreeRoot && current.child !== null && primarySubtreeFlag !==
NoFlags) {
          current = current.child;
        } else {
          if ((current.flags & fiberFlags) !== NoFlags) {
            invokeEffectFn(current);
          }

          if (current.sibling !== null) {
            current = current.sibling;
          } else {
            current = subtreeRoot = current.return;
          }
        }
      }
    }
  }

  var didWarnStateUpdateForNotYetMountedComponent = null;
  function warnAboutUpdateOnNotYetMountedFiberInDEV(fiber) {
    {
      if ((executionContext & RenderContext) !== NoContext) {
        // We let the other warning about render phase updates deal with this one.
        return;
      }

      if (!(fiber.mode & ConcurrentMode)) {
        return;
      }

      var tag = fiber.tag;

      if (tag !== IndeterminateComponent && tag !== HostRoot && tag !== ClassComponent &&
tag !== FunctionComponent && tag !== ForwardRef && tag !== MemoComponent && tag !==
SimpleMemoComponent) {
        // Only warn for user-defined components, not internal ones like Suspense.
        return;
      } // We show the whole stack but dedupe on the top component's name because
      // the problematic code almost always lies inside that component.


      var componentName = getComponentNameFromFiber(fiber) || 'ReactComponent';

      if (didWarnStateUpdateForNotYetMountedComponent !== null) {
        if (didWarnStateUpdateForNotYetMountedComponent.has(componentName)) {
          return;
        }

        didWarnStateUpdateForNotYetMountedComponent.add(componentName);
      } else {
        didWarnStateUpdateForNotYetMountedComponent = new Set([componentName]);
      }

      var previousFiber = current;

      try {
        setCurrentFiber(fiber);

        error("Can't perform a React state update on a component that hasn't mounted yet.
" + 'This indicates that you have a side-effect in your render function that ' +
'asynchronously later calls tries to update the component. Move this work to ' +
'useEffect instead.');
      } finally {
        if (previousFiber) {
          setCurrentFiber(fiber);
        } else {
          resetCurrentFiber();
        }
      }
    }
  }
  var beginWork$1;

  {
    var dummyFiber = null;

    beginWork$1 = function (current, unitOfWork, lanes) {
      // If a component throws an error, we replay it again in a synchronously
      // dispatched event, so that the debugger will treat it as an uncaught
      // error See ReactErrorUtils for more information.
      // Before entering the begin phase, copy the work-in-progress onto a dummy
      // fiber. If beginWork throws, we'll use this to reset the state.
      var originalWorkInProgressCopy = assignFiberPropertiesInDEV(dummyFiber,
unitOfWork);

      try {
        return beginWork(current, unitOfWork, lanes);
      } catch (originalError) {
        if (didSuspendOrErrorWhileHydratingDEV() || originalError !== null && typeof
originalError === 'object' && typeof originalError.then === 'function') {
          // Don't replay promises.
          // Don't replay errors if we are hydrating and have already suspended or
handled an error
          throw originalError;
        } // Keep this code in sync with handleError; any changes here must have
        // corresponding changes there.


        resetContextDependencies();
        resetHooksAfterThrow(); // Don't reset current debug fiber, since we're about to
work on the
        // same fiber again.
        // Unwind the failed stack frame

        unwindInterruptedWork(current, unitOfWork); // Restore the original properties of
the fiber.

        assignFiberPropertiesInDEV(unitOfWork, originalWorkInProgressCopy);

        if ( unitOfWork.mode & ProfileMode) {
          // Reset the profiler timer.
          startProfilerTimer(unitOfWork);
        } // Run beginWork again.


        invokeGuardedCallback(null, beginWork, null, current, unitOfWork, lanes);

        if (hasCaughtError()) {
          var replayError = clearCaughtError();

          if (typeof replayError === 'object' && replayError !== null &&
replayError._suppressLogging && typeof originalError === 'object' && originalError !==
null && !originalError._suppressLogging) {
            // If suppressed, let the flag carry over to the original error which is the
one we'll rethrow.
            originalError._suppressLogging = true;
          }
        } // We always throw the original error in case the second render pass is not
idempotent.
        // This can happen if a memoized function or CommonJS module doesn't throw after
first invocation.
```

```
        throw originalError;
      }
    };
  }

  var didWarnAboutUpdateInRender = false;
  var didWarnAboutUpdateInRenderForAnotherComponent;

  {
    didWarnAboutUpdateInRenderForAnotherComponent = new Set();
  }

  function warnAboutRenderPhaseUpdatesInDEV(fiber) {
    {
      if (isRendering && !getIsUpdatingOpaqueValueInRenderPhaseInDEV()) {
        switch (fiber.tag) {
          case FunctionComponent:
          case ForwardRef:
          case SimpleMemoComponent:
            {
              var renderingComponentName = workInProgress &&
getComponentNameFromFiber(workInProgress) || 'Unknown'; // Dedupe by the rendering
component because it's the one that needs to be fixed.

              var dedupeKey = renderingComponentName;

              if (!didWarnAboutUpdateInRenderForAnotherComponent.has(dedupeKey)) {
                didWarnAboutUpdateInRenderForAnotherComponent.add(dedupeKey);
                var setStateComponentName = getComponentNameFromFiber(fiber) ||
'Unknown';

                error('Cannot update a component (`%s`) while rendering a ' + 'different
component (`%s`). To locate the bad setState() call inside `%s`, ' + 'follow the stack
trace as described in https://reactjs.org/link/setstate-in-render',
setStateComponentName, renderingComponentName, renderingComponentName);
              }

              break;
            }

          case ClassComponent:
            {
              if (!didWarnAboutUpdateInRender) {
                error('Cannot update during an existing state transition (such as ' +
'within `render`). Render methods should be a pure ' + 'function of props and state.');

                didWarnAboutUpdateInRender = true;
              }

              break;
            }
        }
      }
    }
  }

  function restorePendingUpdaters(root, lanes) {
    {
      if (isDevToolsPresent) {
        var memoizedUpdaters = root.memoizedUpdaters;
        memoizedUpdaters.forEach(function (schedulingFiber) {
          addFiberToLanesMap(root, schedulingFiber, lanes);
        }); // This function intentionally does not clear memoized updaters.
        // Those may still be relevant to the current commit
        // and a future one (e.g. Suspense).
      }
    }
  }
  var fakeActCallbackNode = {};

  function scheduleCallback$1(priorityLevel, callback) {
    {
      // If we're currently inside an `act` scope, bypass Scheduler and push to
      // the `act` queue instead.
      var actQueue = ReactCurrentActQueue$1.current;

      if (actQueue !== null) {
        actQueue.push(callback);
        return fakeActCallbackNode;
      } else {
        return scheduleCallback(priorityLevel, callback);
      }
    }
  }

  function cancelCallback$1(callbackNode) {
    if ( callbackNode === fakeActCallbackNode) {
      return;
    } // In production, always call Scheduler. This function will be stripped out.


    return cancelCallback(callbackNode);
  }

  function shouldForceFlushFallbacksInDEV() {
    // Never force flush in production. This function should get stripped out.
    return  ReactCurrentActQueue$1.current !== null;
  }

  function warnIfUpdatesNotWrappedWithActDEV(fiber) {
    {
      if (fiber.mode & ConcurrentMode) {
        if (!isConcurrentActEnvironment()) {
          // Not in an act environment. No need to warn.
          return;
        }
      } else {
        // Legacy mode has additional cases where we suppress a warning.
        if (!isLegacyActEnvironment()) {
          // Not in an act environment. No need to warn.
          return;
        }

        if (executionContext !== NoContext) {
          // Legacy mode doesn't warn if the update is batched, i.e.
          // batchedUpdates or flushSync.
          return;
        }

        if (fiber.tag !== FunctionComponent && fiber.tag !== ForwardRef && fiber.tag !==
SimpleMemoComponent) {
          // For backwards compatibility with pre-hooks code, legacy mode only
          // warns for updates that originate from a hook.
          return;
        }
      }

      if (ReactCurrentActQueue$1.current === null) {
        var previousFiber = current;

        try {
          setCurrentFiber(fiber);

          error('An update to %s inside a test was not wrapped in act(...).\n\n' + 'When
testing, code that causes React state updates should be ' + 'wrapped into act(...):\n\n'
+ 'act(() => {\n' + '  /* fire events that update state */\n' + '});\n' + '/* assert on
the output */\n\n' + "This ensures that you're testing the behavior the user would see "
+ 'in the browser.' + ' Learn more at https://reactjs.org/link/wrap-tests-with-act',
getComponentNameFromFiber(fiber));
        } finally {
          if (previousFiber) {
            setCurrentFiber(fiber);
          } else {
            resetCurrentFiber();
          }
        }
      }
    }
  }

  function warnIfSuspenseResolutionNotWrappedWithActDEV(root) {
    {
      if (root.tag !== LegacyRoot && isConcurrentActEnvironment() &&
ReactCurrentActQueue$1.current === null) {
        error('A suspended resource finished loading inside a test, but the event ' +
'was not wrapped in act(...).\n\n' + 'When testing, code that resolves suspended data
should be wrapped ' + 'into act(...):\n\n' + 'act(() => {\n' + '  /* finish loading
suspended data */\n' + '});\n' + '/* assert on the output */\n\n' + "This ensures that
you're testing the behavior the user would see " + 'in the browser.' + ' Learn more at
https://reactjs.org/link/wrap-tests-with-act');
      }
    }
  }

  function setIsRunningInsertionEffect(isRunning) {
    {
      isRunningInsertionEffect = isRunning;
    }
  }

  /* eslint-disable react-internal/prod-error-codes */
  var resolveFamily = null; // $FlowFixMe Flow gets confused by a WeakSet feature check
below.

  var failedBoundaries = null;
  var setRefreshHandler = function (handler) {
    {
      resolveFamily = handler;
    }
  };
  function resolveFunctionForHotReloading(type) {
    {
      if (resolveFamily === null) {
        // Hot reloading is disabled.
        return type;
      }

      var family = resolveFamily(type);

      if (family === undefined) {
        return type;
      } // Use the latest known implementation.


      return family.current;
    }
  }
  function resolveClassForHotReloading(type) {
    // No implementation differences.
    return resolveFunctionForHotReloading(type);
  }
  function resolveForwardRefForHotReloading(type) {
    {
      if (resolveFamily === null) {
        // Hot reloading is disabled.
        return type;
      }

      var family = resolveFamily(type);

      if (family === undefined) {
        // Check if we're dealing with a real forwardRef. Don't want to crash early.
        if (type !== null && type !== undefined && typeof type.render === 'function') {
          // ForwardRef is special because its resolved .type is an object,
          // but it's possible that we only have its inner render function in the map.
          // If that inner render function is different, we'll build a new forwardRef
type.
          var currentRender = resolveFunctionForHotReloading(type.render);

          if (type.render !== currentRender) {
            var syntheticType = {
              $$typeof: REACT_FORWARD_REF_TYPE,
              render: currentRender
            };

            if (type.displayName !== undefined) {
              syntheticType.displayName = type.displayName;
            }

            return syntheticType;
          }
        }

        return type;
      } // Use the latest known implementation.


      return family.current;
    }
  }
  function isCompatibleFamilyForHotReloading(fiber, element) {
    {
      if (resolveFamily === null) {
        // Hot reloading is disabled.
        return false;
      }

      var prevType = fiber.elementType;
      var nextType = element.type; // If we got here, we know types aren't === equal.

      var needsCompareFamilies = false;
      var $$typeofNextType = typeof nextType === 'object' && nextType !== null ?
nextType.$$typeof : null;

      switch (fiber.tag) {
        case ClassComponent:
          {
            if (typeof nextType === 'function') {
              needsCompareFamilies = true;
            }

            break;
          }

        case FunctionComponent:
          {
            if (typeof nextType === 'function') {
              needsCompareFamilies = true;
            } else if ($$typeofNextType === REACT_LAZY_TYPE) {
              // We don't know the inner type yet.
              // We're going to assume that the lazy inner type is stable,
              // and so it is sufficient to avoid reconciling it away.
              // We're not going to unwrap or actually use the new lazy type.
              needsCompareFamilies = true;
            }
            break;
          }

        case ForwardRef:
          {
            if ($$typeofNextType === REACT_FORWARD_REF_TYPE) {
              needsCompareFamilies = true;
            } else if ($$typeofNextType === REACT_LAZY_TYPE) {
              needsCompareFamilies = true;
            }

            break;
          }

        case MemoComponent:
        case SimpleMemoComponent:
          {
            if ($$typeofNextType === REACT_MEMO_TYPE) {
              // TODO: if it was but can no longer be simple,
              // we shouldn't set this.
              needsCompareFamilies = true;
            } else if ($$typeofNextType === REACT_LAZY_TYPE) {
              needsCompareFamilies = true;
            }

            break;
          }

        default:
          return false;
      } // Check if both types have a family and it's the same one.


      if (needsCompareFamilies) {
        // Note: memo() and forwardRef() we'll compare outer rather than inner type.
        // This means both of them need to be registered to preserve state.
        // If we unwrapped and compared the inner types for wrappers instead,
        // then we would risk falsely saying two separate memo(Foo)
        // calls are equivalent because they wrap the same Foo function.
        var prevFamily = resolveFamily(prevType);

        if (prevFamily !== undefined && prevFamily === resolveFamily(nextType)) {
          return true;
        }
      }

      return false;
    }
  }
  function markFailedErrorBoundaryForHotReloading(fiber) {
    {
      if (resolveFamily === null) {
        // Hot reloading is disabled.
        return;
      }

      if (typeof WeakSet !== 'function') {
        return;
      }

      if (failedBoundaries === null) {
        failedBoundaries = new WeakSet();
      }

      failedBoundaries.add(fiber);
    }
  }
  var scheduleRefresh = function (root, update) {
    {
      if (resolveFamily === null) {
        // Hot reloading is disabled.
        return;
      }

      var staleFamilies = update.staleFamilies,
          updatedFamilies = update.updatedFamilies;
      flushPassiveEffects();
      flushSync(function () {
        scheduleFibersWithFamiliesRecursively(root.current, updatedFamilies,
staleFamilies);
      });
    }
  };
  var scheduleRoot = function (root, element) {
    {
      if (root.context !== emptyContextObject) {
        // Super edge case: root has a legacy _renderSubtree context
        // but we don't know the parentComponent so we can't pass it.
        // Just ignore. We'll delete this with _renderSubtree code path later.
        return;
      }

      flushPassiveEffects();
      flushSync(function () {
        updateContainer(element, root, null, null);
      });
    }
  };

  function scheduleFibersWithFamiliesRecursively(fiber, updatedFamilies, staleFamilies) {
    {
      var alternate = fiber.alternate,
          child = fiber.child,
          sibling = fiber.sibling,
          tag = fiber.tag,
          type = fiber.type;
      var candidateType = null;

      switch (tag) {
        case FunctionComponent:
        case SimpleMemoComponent:
        case ClassComponent:
          candidateType = type;
          break;

        case ForwardRef:
          candidateType = type.render;
          break;
      }

      if (resolveFamily === null) {
        throw new Error('Expected resolveFamily to be set during hot reload.');
      }

      var needsRender = false;
      var needsRemount = false;

      if (candidateType !== null) {
        var family = resolveFamily(candidateType);

        if (family !== undefined) {
          if (staleFamilies.has(family)) {
            needsRemount = true;
          } else if (updatedFamilies.has(family)) {
            if (tag === ClassComponent) {
              needsRemount = true;
            } else {
```

```
              needsRender = true;
            }
          }
        }
      }

      if (failedBoundaries !== null) {
        if (failedBoundaries.has(fiber) || alternate !== null &&
  failedBoundaries.has(alternate)) {
          needsRemount = true;
        }
      }

      if (needsRemount) {
        fiber._debugNeedsRemount = true;
      }

      if (needsRemount || needsRender) {
        var _root = enqueueConcurrentRenderForLane(fiber, SyncLane);

        if (_root !== null) {
          scheduleUpdateOnFiber(_root, fiber, SyncLane, NoTimestamp);
        }
      }

      if (child !== null && !needsRemount) {
        scheduleFibersWithFamiliesRecursively(child, updatedFamilies, staleFamilies);
      }

      if (sibling !== null) {
        scheduleFibersWithFamiliesRecursively(sibling, updatedFamilies, staleFamilies);
      }
    }
  }

  var findHostInstancesForRefresh = function (root, families) {
    {
      var hostInstances = new Set();
      var types = new Set(families.map(function (family) {
        return family.current;
      }));
      findHostInstancesForMatchingFibersRecursively(root.current, types, hostInstances);
      return hostInstances;
    }
  };

  function findHostInstancesForMatchingFibersRecursively(fiber, types, hostInstances) {
    {
      var child = fiber.child,
          sibling = fiber.sibling,
          tag = fiber.tag,
          type = fiber.type;
      var candidateType = null;

      switch (tag) {
        case FunctionComponent:
        case SimpleMemoComponent:
        case ClassComponent:
          candidateType = type;
          break;

        case ForwardRef:
          candidateType = type.render;
          break;
      }

      var didMatch = false;

      if (candidateType !== null) {
        if (types.has(candidateType)) {
          didMatch = true;
        }
      }

      if (didMatch) {
        // We have a match. This only drills down to the closest host components.
        // There's no need to search deeper because for the purpose of giving
        // visual feedback, "flashing" outermost parent rectangles is sufficient.
        findHostInstancesForFiberShallowly(fiber, hostInstances);
      } else {
        // If there's no match, maybe there will be one further down in the child tree.
        if (child !== null) {
          findHostInstancesForMatchingFibersRecursively(child, types, hostInstances);
        }
      }

      if (sibling !== null) {
        findHostInstancesForMatchingFibersRecursively(sibling, types, hostInstances);
      }
    }
  }

  function findHostInstancesForFiberShallowly(fiber, hostInstances) {
    {
      var foundHostInstances = findChildHostInstancesForFiberShallowly(fiber,
hostInstances);

      if (foundHostInstances) {
        return;
      } // If we didn't find any host children, fallback to closest host parent.


      var node = fiber;

      while (true) {
        switch (node.tag) {
          case HostComponent:
            hostInstances.add(node.stateNode);
            return;

          case HostPortal:
            hostInstances.add(node.stateNode.containerInfo);
            return;

          case HostRoot:
            hostInstances.add(node.stateNode.containerInfo);
            return;
        }

        if (node.return === null) {
          throw new Error('Expected to reach root first.');
        }

        node = node.return;
      }
    }
  }

  function findChildHostInstancesForFiberShallowly(fiber, hostInstances) {
    {
      var node = fiber;
      var foundHostInstances = false;

      while (true) {
        if (node.tag === HostComponent) {
          // We got a match.
          foundHostInstances = true;
          hostInstances.add(node.stateNode); // There may still be more, so keep
```

```
searching.
        } else if (node.child !== null) {
          node.child.return = node;
          node = node.child;
          continue;
        }

        if (node === fiber) {
          return foundHostInstances;
        }

        while (node.sibling === null) {
          if (node.return === null || node.return === fiber) {
            return foundHostInstances;
          }

          node = node.return;
        }

        node.sibling.return = node.return;
        node = node.sibling;
      }
    }

    return false;
  }

  var hasBadMapPolyfill;

  {
    hasBadMapPolyfill = false;

    try {
      var nonExtensibleObject = Object.preventExtensions({});
      /* eslint-disable no-new */

      new Map([[nonExtensibleObject, null]]);
      new Set([nonExtensibleObject]);
      /* eslint-enable no-new */
    } catch (e) {
      // TODO: Consider warning about bad polyfills
      hasBadMapPolyfill = true;
    }
  }

  function FiberNode(tag, pendingProps, key, mode) {
    // Instance
    this.tag = tag;
    this.key = key;
    this.elementType = null;
    this.type = null;
    this.stateNode = null; // Fiber

    this.return = null;
    this.child = null;
    this.sibling = null;
    this.index = 0;
    this.ref = null;
    this.pendingProps = pendingProps;
    this.memoizedProps = null;
    this.updateQueue = null;
    this.memoizedState = null;
    this.dependencies = null;
    this.mode = mode; // Effects

    this.flags = NoFlags;
    this.subtreeFlags = NoFlags;
    this.deletions = null;
    this.lanes = NoLanes;
```

```
      this.childLanes = NoLanes;
      this.alternate = null;

      {
        // Note: The following is done to avoid a v8 performance cliff.
        //
        // Initializing the fields below to smis and later updating them with
        // double values will cause Fibers to end up having separate shapes.
        // This behavior/bug has something to do with Object.preventExtension().
        // Fortunately this only impacts DEV builds.
        // Unfortunately it makes React unusably slow for some applications.
        // To work around this, initialize the fields below with doubles.
        //
        // Learn more about this here:
        // https://github.com/facebook/react/issues/14365
        // https://bugs.chromium.org/p/v8/issues/detail?id=8538
        this.actualDuration = Number.NaN;
        this.actualStartTime = Number.NaN;
        this.selfBaseDuration = Number.NaN;
        this.treeBaseDuration = Number.NaN; // It's okay to replace the initial doubles
 with smis after initialization.
        // This won't trigger the performance cliff mentioned above,
        // and it simplifies other profiler code (including DevTools).

        this.actualDuration = 0;
        this.actualStartTime = -1;
        this.selfBaseDuration = 0;
        this.treeBaseDuration = 0;
      }

      {
        // This isn't directly used but is handy for debugging internals:
        this._debugSource = null;
        this._debugOwner = null;
        this._debugNeedsRemount = false;
        this._debugHookTypes = null;

        if (!hasBadMapPolyfill && typeof Object.preventExtensions === 'function') {
          Object.preventExtensions(this);
        }
      }
    } // This is a constructor function, rather than a POJO constructor, still
    // please ensure we do the following:
    // 1) Nobody should add any instance methods on this. Instance methods can be
    //    more difficult to predict when they get optimized and they are almost
    //    never inlined properly in static compilers.
    // 2) Nobody should rely on `instanceof Fiber` for type testing. We should
    //    always know when it is a fiber.
    // 3) We might want to experiment with using numeric keys since they are easier
    //    to optimize in a non-JIT environment.
    // 4) We can easily go from a constructor to a createFiber object literal if that
    //    is faster.
    // 5) It should be easy to port this to a C struct and keep a C implementation
    //    compatible.


    var createFiber = function (tag, pendingProps, key, mode) {
      // $FlowFixMe: the shapes are exact here but Flow doesn't like constructors
      return new FiberNode(tag, pendingProps, key, mode);
    };

    function shouldConstruct$1(Component) {
      var prototype = Component.prototype;
      return !!(prototype && prototype.isReactComponent);
    }

    function isSimpleFunctionComponent(type) {
      return typeof type === 'function' && !shouldConstruct$1(type) && type.defaultProps
 === undefined;
```

```
  }
  function resolveLazyComponentTag(Component) {
    if (typeof Component === 'function') {
      return shouldConstruct$1(Component) ? ClassComponent : FunctionComponent;
    } else if (Component !== undefined && Component !== null) {
      var $$typeof = Component.$$typeof;

      if ($$typeof === REACT_FORWARD_REF_TYPE) {
        return ForwardRef;
      }

      if ($$typeof === REACT_MEMO_TYPE) {
        return MemoComponent;
      }
    }

    return IndeterminateComponent;
  } // This is used to create an alternate fiber to do work on.

  function createWorkInProgress(current, pendingProps) {
    var workInProgress = current.alternate;

    if (workInProgress === null) {
      // We use a double buffering pooling technique because we know that we'll
      // only ever need at most two versions of a tree. We pool the "other" unused
      // node that we're free to reuse. This is lazily created to avoid allocating
      // extra objects for things that are never updated. It also allow us to
      // reclaim the extra memory if needed.
      workInProgress = createFiber(current.tag, pendingProps, current.key, current.mode);
      workInProgress.elementType = current.elementType;
      workInProgress.type = current.type;
      workInProgress.stateNode = current.stateNode;

      {
        // DEV-only fields
        workInProgress._debugSource = current._debugSource;
        workInProgress._debugOwner = current._debugOwner;
        workInProgress._debugHookTypes = current._debugHookTypes;
      }

      workInProgress.alternate = current;
      current.alternate = workInProgress;
    } else {
      workInProgress.pendingProps = pendingProps; // Needed because Blocks store data on
type.

      workInProgress.type = current.type; // We already have an alternate.
      // Reset the effect tag.

      workInProgress.flags = NoFlags; // The effects are no longer valid.

      workInProgress.subtreeFlags = NoFlags;
      workInProgress.deletions = null;

      {
        // We intentionally reset, rather than copy, actualDuration & actualStartTime.
        // This prevents time from endlessly accumulating in new commits.
        // This has the downside of resetting values for different priority renders,
        // But works for yielding (the common case) and should support resuming.
        workInProgress.actualDuration = 0;
        workInProgress.actualStartTime = -1;
      }
    } // Reset all effects except static ones.
    // Static effects are not specific to a render.


    workInProgress.flags = current.flags & StaticMask;
    workInProgress.childLanes = current.childLanes;
    workInProgress.lanes = current.lanes;
```

```
      workInProgress.child = current.child;
      workInProgress.memoizedProps = current.memoizedProps;
      workInProgress.memoizedState = current.memoizedState;
      workInProgress.updateQueue = current.updateQueue; // Clone the dependencies object.
  This is mutated during the render phase, so
      // it cannot be shared with the current fiber.

      var currentDependencies = current.dependencies;
      workInProgress.dependencies = currentDependencies === null ? null : {
        lanes: currentDependencies.lanes,
        firstContext: currentDependencies.firstContext
      }; // These will be overridden during the parent's reconciliation

      workInProgress.sibling = current.sibling;
      workInProgress.index = current.index;
      workInProgress.ref = current.ref;

      {
        workInProgress.selfBaseDuration = current.selfBaseDuration;
        workInProgress.treeBaseDuration = current.treeBaseDuration;
      }

      {
        workInProgress._debugNeedsRemount = current._debugNeedsRemount;

        switch (workInProgress.tag) {
          case IndeterminateComponent:
          case FunctionComponent:
          case SimpleMemoComponent:
            workInProgress.type = resolveFunctionForHotReloading(current.type);
            break;

          case ClassComponent:
            workInProgress.type = resolveClassForHotReloading(current.type);
            break;

          case ForwardRef:
            workInProgress.type = resolveForwardRefForHotReloading(current.type);
            break;
        }
      }

      return workInProgress;
    } // Used to reuse a Fiber for a second pass.

    function resetWorkInProgress(workInProgress, renderLanes) {
      // This resets the Fiber to what createFiber or createWorkInProgress would
      // have set the values to before during the first pass. Ideally this wouldn't
      // be necessary but unfortunately many code paths reads from the workInProgress
      // when they should be reading from current and writing to workInProgress.
      // We assume pendingProps, index, key, ref, return are still untouched to
      // avoid doing another reconciliation.
      // Reset the effect flags but keep any Placement tags, since that's something
      // that child fiber is setting, not the reconciliation.
      workInProgress.flags &= StaticMask | Placement; // The effects are no longer valid.

      var current = workInProgress.alternate;

      if (current === null) {
        // Reset to createFiber's initial values.
        workInProgress.childLanes = NoLanes;
        workInProgress.lanes = renderLanes;
        workInProgress.child = null;
        workInProgress.subtreeFlags = NoFlags;
        workInProgress.memoizedProps = null;
        workInProgress.memoizedState = null;
        workInProgress.updateQueue = null;
        workInProgress.dependencies = null;
        workInProgress.stateNode = null;
        {
          // Note: We don't reset the actualTime counts. It's useful to accumulate
          // actual time across multiple render passes.
          workInProgress.selfBaseDuration = 0;
          workInProgress.treeBaseDuration = 0;
        }
      } else {
        // Reset to the cloned values that createWorkInProgress would've.
        workInProgress.childLanes = current.childLanes;
        workInProgress.lanes = current.lanes;
        workInProgress.child = current.child;
        workInProgress.subtreeFlags = NoFlags;
        workInProgress.deletions = null;
        workInProgress.memoizedProps = current.memoizedProps;
        workInProgress.memoizedState = current.memoizedState;
        workInProgress.updateQueue = current.updateQueue; // Needed because Blocks store
  data on type.

        workInProgress.type = current.type; // Clone the dependencies object. This is
  mutated during the render phase, so
        // it cannot be shared with the current fiber.

        var currentDependencies = current.dependencies;
        workInProgress.dependencies = currentDependencies === null ? null : {
          lanes: currentDependencies.lanes,
          firstContext: currentDependencies.firstContext
        };

        {
          // Note: We don't reset the actualTime counts. It's useful to accumulate
          // actual time across multiple render passes.
          workInProgress.selfBaseDuration = current.selfBaseDuration;
          workInProgress.treeBaseDuration = current.treeBaseDuration;
        }
      }

      return workInProgress;
    }
    function createHostRootFiber(tag, isStrictMode, concurrentUpdatesByDefaultOverride) {
      var mode;

      if (tag === ConcurrentRoot) {
        mode = ConcurrentMode;

        if (isStrictMode === true) {
          mode |= StrictLegacyMode;

          {
            mode |= StrictEffectsMode;
          }
        }
      } else {
        mode = NoMode;
      }

      if ( isDevToolsPresent) {
        // Always collect profile timings when DevTools are present.
        // This enables DevTools to start capturing timing at any point–
        // Without some nodes in the tree having empty base times.
        mode |= ProfileMode;
      }

      return createFiber(HostRoot, null, null, mode);
    }
    function createFiberFromTypeAndProps(type, // React$ElementType
    key, pendingProps, owner, mode, lanes) {
      var fiberTag = IndeterminateComponent; // The resolved type is set if we know what
  the final type will be. I.e. it's not lazy.
```

```
    var resolvedType = type;

    if (typeof type === 'function') {
      if (shouldConstruct$1(type)) {
        fiberTag = ClassComponent;

        {
          resolvedType = resolveClassForHotReloading(resolvedType);
        }
      } else {
        {
          resolvedType = resolveFunctionForHotReloading(resolvedType);
        }
      }
    } else if (typeof type === 'string') {
      fiberTag = HostComponent;
    } else {
      getTag: switch (type) {
        case REACT_FRAGMENT_TYPE:
          return createFiberFromFragment(pendingProps.children, mode, lanes, key);

        case REACT_STRICT_MODE_TYPE:
          fiberTag = Mode;
          mode |= StrictLegacyMode;

          if ( (mode & ConcurrentMode) !== NoMode) {
            // Strict effects should never run on legacy roots
            mode |= StrictEffectsMode;
          }

          break;

        case REACT_PROFILER_TYPE:
          return createFiberFromProfiler(pendingProps, mode, lanes, key);

        case REACT_SUSPENSE_TYPE:
          return createFiberFromSuspense(pendingProps, mode, lanes, key);

        case REACT_SUSPENSE_LIST_TYPE:
          return createFiberFromSuspenseList(pendingProps, mode, lanes, key);

        case REACT_OFFSCREEN_TYPE:
          return createFiberFromOffscreen(pendingProps, mode, lanes, key);

        case REACT_LEGACY_HIDDEN_TYPE:

        // eslint-disable-next-line no-fallthrough

        case REACT_SCOPE_TYPE:

        // eslint-disable-next-line no-fallthrough

        case REACT_CACHE_TYPE:

        // eslint-disable-next-line no-fallthrough

        case REACT_TRACING_MARKER_TYPE:

        // eslint-disable-next-line no-fallthrough

        case REACT_DEBUG_TRACING_MODE_TYPE:

        // eslint-disable-next-line no-fallthrough

        default:
          {
            if (typeof type === 'object' && type !== null) {
              switch (type.$$typeof) {
```

```
              case REACT_PROVIDER_TYPE:
                fiberTag = ContextProvider;
                break getTag;

              case REACT_CONTEXT_TYPE:
                // This is a consumer
                fiberTag = ContextConsumer;
                break getTag;

              case REACT_FORWARD_REF_TYPE:
                fiberTag = ForwardRef;

                {
                  resolvedType = resolveForwardRefForHotReloading(resolvedType);
                }

                break getTag;

              case REACT_MEMO_TYPE:
                fiberTag = MemoComponent;
                break getTag;

              case REACT_LAZY_TYPE:
                fiberTag = LazyComponent;
                resolvedType = null;
                break getTag;
            }
          }

          var info = '';

          {
            if (type === undefined || typeof type === 'object' && type !== null &&
Object.keys(type).length === 0) {
              info += ' You likely forgot to export your component from the file ' +
"it's defined in, or you might have mixed up default and " + 'named imports.';
            }

            var ownerName = owner ? getComponentNameFromFiber(owner) : null;

            if (ownerName) {
              info += '\n\nCheck the render method of `' + ownerName + '`.';
            }
          }

          throw new Error('Element type is invalid: expected a string (for built-in ' +
'components) or a class/function (for composite components) ' + ("but got: " + (type ==
null ? type : typeof type) + "." + info));
        }
      }
    }

    var fiber = createFiber(fiberTag, pendingProps, key, mode);
    fiber.elementType = type;
    fiber.type = resolvedType;
    fiber.lanes = lanes;

    {
      fiber._debugOwner = owner;
    }

    return fiber;
  }
  function createFiberFromElement(element, mode, lanes) {
    var owner = null;

    {
      owner = element._owner;
    }
```

```
      var type = element.type;
      var key = element.key;
      var pendingProps = element.props;
      var fiber = createFiberFromTypeAndProps(type, key, pendingProps, owner, mode, lanes);

      {
        fiber._debugSource = element._source;
        fiber._debugOwner = element._owner;
      }

      return fiber;
    }
    function createFiberFromFragment(elements, mode, lanes, key) {
      var fiber = createFiber(Fragment, elements, key, mode);
      fiber.lanes = lanes;
      return fiber;
    }

    function createFiberFromProfiler(pendingProps, mode, lanes, key) {
      {
        if (typeof pendingProps.id !== 'string') {
          error('Profiler must specify an "id" of type `string` as a prop. Received the
  type `%s` instead.', typeof pendingProps.id);
        }
      }

      var fiber = createFiber(Profiler, pendingProps, key, mode | ProfileMode);
      fiber.elementType = REACT_PROFILER_TYPE;
      fiber.lanes = lanes;

      {
        fiber.stateNode = {
          effectDuration: 0,
          passiveEffectDuration: 0
        };
      }

      return fiber;
    }

    function createFiberFromSuspense(pendingProps, mode, lanes, key) {
      var fiber = createFiber(SuspenseComponent, pendingProps, key, mode);
      fiber.elementType = REACT_SUSPENSE_TYPE;
      fiber.lanes = lanes;
      return fiber;
    }
    function createFiberFromSuspenseList(pendingProps, mode, lanes, key) {
      var fiber = createFiber(SuspenseListComponent, pendingProps, key, mode);
      fiber.elementType = REACT_SUSPENSE_LIST_TYPE;
      fiber.lanes = lanes;
      return fiber;
    }
    function createFiberFromOffscreen(pendingProps, mode, lanes, key) {
      var fiber = createFiber(OffscreenComponent, pendingProps, key, mode);
      fiber.elementType = REACT_OFFSCREEN_TYPE;
      fiber.lanes = lanes;
      var primaryChildInstance = {
        isHidden: false
      };
      fiber.stateNode = primaryChildInstance;
      return fiber;
    }
    function createFiberFromText(content, mode, lanes) {
      var fiber = createFiber(HostText, content, null, mode);
      fiber.lanes = lanes;
      return fiber;
    }
    function createFiberFromHostInstanceForDeletion() {
      var fiber = createFiber(HostComponent, null, null, NoMode);
      fiber.elementType = 'DELETED';
      return fiber;
    }
    function createFiberFromDehydratedFragment(dehydratedNode) {
      var fiber = createFiber(DehydratedFragment, null, null, NoMode);
      fiber.stateNode = dehydratedNode;
      return fiber;
    }
    function createFiberFromPortal(portal, mode, lanes) {
      var pendingProps = portal.children !== null ? portal.children : [];
      var fiber = createFiber(HostPortal, pendingProps, portal.key, mode);
      fiber.lanes = lanes;
      fiber.stateNode = {
        containerInfo: portal.containerInfo,
        pendingChildren: null,
        // Used by persistent updates
        implementation: portal.implementation
      };
      return fiber;
    } // Used for stashing WIP properties to replay failed work in DEV.

    function assignFiberPropertiesInDEV(target, source) {
      if (target === null) {
        // This Fiber's initial properties will always be overwritten.
        // We only use a Fiber to ensure the same hidden class so DEV isn't slow.
        target = createFiber(IndeterminateComponent, null, null, NoMode);
      } // This is intentionally written as a list of all properties.
      // We tried to use Object.assign() instead but this is called in
      // the hottest path, and Object.assign() was too slow:
      // https://github.com/facebook/react/issues/12502
      // This code is DEV-only so size is not a concern.


      target.tag = source.tag;
      target.key = source.key;
      target.elementType = source.elementType;
      target.type = source.type;
      target.stateNode = source.stateNode;
      target.return = source.return;
      target.child = source.child;
      target.sibling = source.sibling;
      target.index = source.index;
      target.ref = source.ref;
      target.pendingProps = source.pendingProps;
      target.memoizedProps = source.memoizedProps;
      target.updateQueue = source.updateQueue;
      target.memoizedState = source.memoizedState;
      target.dependencies = source.dependencies;
      target.mode = source.mode;
      target.flags = source.flags;
      target.subtreeFlags = source.subtreeFlags;
      target.deletions = source.deletions;
      target.lanes = source.lanes;
      target.childLanes = source.childLanes;
      target.alternate = source.alternate;

      {
        target.actualDuration = source.actualDuration;
        target.actualStartTime = source.actualStartTime;
        target.selfBaseDuration = source.selfBaseDuration;
        target.treeBaseDuration = source.treeBaseDuration;
      }

      target._debugSource = source._debugSource;
      target._debugOwner = source._debugOwner;
      target._debugNeedsRemount = source._debugNeedsRemount;
      target._debugHookTypes = source._debugHookTypes;
      return target;
```

```
  }

  function FiberRootNode(containerInfo, tag, hydrate, identifierPrefix,
onRecoverableError) {
    this.tag = tag;
    this.containerInfo = containerInfo;
    this.pendingChildren = null;
    this.current = null;
    this.pingCache = null;
    this.finishedWork = null;
    this.timeoutHandle = noTimeout;
    this.context = null;
    this.pendingContext = null;
    this.callbackNode = null;
    this.callbackPriority = NoLane;
    this.eventTimes = createLaneMap(NoLanes);
    this.expirationTimes = createLaneMap(NoTimestamp);
    this.pendingLanes = NoLanes;
    this.suspendedLanes = NoLanes;
    this.pingedLanes = NoLanes;
    this.expiredLanes = NoLanes;
    this.mutableReadLanes = NoLanes;
    this.finishedLanes = NoLanes;
    this.entangledLanes = NoLanes;
    this.entanglements = createLaneMap(NoLanes);
    this.identifierPrefix = identifierPrefix;
    this.onRecoverableError = onRecoverableError;

    {
      this.mutableSourceEagerHydrationData = null;
    }

    {
      this.effectDuration = 0;
      this.passiveEffectDuration = 0;
    }

    {
      this.memoizedUpdaters = new Set();
      var pendingUpdatersLaneMap = this.pendingUpdatersLaneMap = [];

      for (var _i = 0; _i < TotalLanes; _i++) {
        pendingUpdatersLaneMap.push(new Set());
      }
    }

    {
      switch (tag) {
        case ConcurrentRoot:
          this._debugRootType = hydrate ? 'hydrateRoot()' : 'createRoot()';
          break;

        case LegacyRoot:
          this._debugRootType = hydrate ? 'hydrate()' : 'render()';
          break;
      }
    }
  }

  function createFiberRoot(containerInfo, tag, hydrate, initialChildren,
hydrationCallbacks, isStrictMode, concurrentUpdatesByDefaultOverride, // TODO: We have
several of these arguments that are conceptually part of the
  // host config, but because they are passed in at runtime, we have to thread
  // them through the root constructor. Perhaps we should put them all into a
  // single type, like a DynamicHostConfig that is defined by the renderer.
  identifierPrefix, onRecoverableError, transitionCallbacks) {
    var root = new FiberRootNode(containerInfo, tag, hydrate, identifierPrefix,
onRecoverableError);
    // stateNode is any.
    var uninitializedFiber = createHostRootFiber(tag, isStrictMode);
    root.current = uninitializedFiber;
    uninitializedFiber.stateNode = root;

    {
      var _initialState = {
        element: initialChildren,
        isDehydrated: hydrate,
        cache: null,
        // not enabled yet
        transitions: null,
        pendingSuspenseBoundaries: null
      };
      uninitializedFiber.memoizedState = _initialState;
    }

    initializeUpdateQueue(uninitializedFiber);
    return root;
  }

  var ReactVersion = '18.3.1';

  function createPortal(children, containerInfo, // TODO: figure out the API for cross-
renderer implementation.
  implementation) {
    var key = arguments.length > 3 && arguments[3] !== undefined ? arguments[3] : null;

    {
      checkKeyStringCoercion(key);
    }

    return {
      // This tag allow us to uniquely identify this as a React Portal
      $$typeof: REACT_PORTAL_TYPE,
      key: key == null ? null : '' + key,
      children: children,
      containerInfo: containerInfo,
      implementation: implementation
    };
  }

  var didWarnAboutNestedUpdates;
  var didWarnAboutFindNodeInStrictMode;

  {
    didWarnAboutNestedUpdates = false;
    didWarnAboutFindNodeInStrictMode = {};
  }

  function getContextForSubtree(parentComponent) {
    if (!parentComponent) {
      return emptyContextObject;
    }

    var fiber = get(parentComponent);
    var parentContext = findCurrentUnmaskedContext(fiber);

    if (fiber.tag === ClassComponent) {
      var Component = fiber.type;

      if (isContextProvider(Component)) {
        return processChildContext(fiber, Component, parentContext);
      }
    }

    return parentContext;
  }
  function findHostInstanceWithWarning(component, methodName) {
    {
      var fiber = get(component);

      if (fiber === undefined) {
        if (typeof component.render === 'function') {
          throw new Error('Unable to find node on an unmounted component.');
        } else {
          var keys = Object.keys(component).join(',');
          throw new Error("Argument appears to not be a ReactComponent. Keys: " + keys);
        }
      }

      var hostFiber = findCurrentHostFiber(fiber);

      if (hostFiber === null) {
        return null;
      }

      if (hostFiber.mode & StrictLegacyMode) {
        var componentName = getComponentNameFromFiber(fiber) || 'Component';

        if (!didWarnAboutFindNodeInStrictMode[componentName]) {
          didWarnAboutFindNodeInStrictMode[componentName] = true;
          var previousFiber = current;

          try {
            setCurrentFiber(hostFiber);

            if (fiber.mode & StrictLegacyMode) {
              error('%s is deprecated in StrictMode. ' + '%s was passed an instance of %s
which is inside StrictMode. ' + 'Instead, add a ref directly to the element you want to
reference. ' + 'Learn more about using refs safely here: ' +
'https://reactjs.org/link/strict-mode-find-node', methodName, methodName, componentName);
            } else {
              error('%s is deprecated in StrictMode. ' + '%s was passed an instance of %s
which renders StrictMode children. ' + 'Instead, add a ref directly to the element you
want to reference. ' + 'Learn more about using refs safely here: ' +
'https://reactjs.org/link/strict-mode-find-node', methodName, methodName, componentName);
            }
          } finally {
            // Ideally this should reset to previous but this shouldn't be called in
            // render and there's another warning for that anyway.
            if (previousFiber) {
              setCurrentFiber(previousFiber);
            } else {
              resetCurrentFiber();
            }
          }
        }
      }

      return hostFiber.stateNode;
    }
  }

  function createContainer(containerInfo, tag, hydrationCallbacks, isStrictMode,
concurrentUpdatesByDefaultOverride, identifierPrefix, onRecoverableError,
transitionCallbacks) {
    var hydrate = false;
    var initialChildren = null;
    return createFiberRoot(containerInfo, tag, hydrate, initialChildren,
hydrationCallbacks, isStrictMode, concurrentUpdatesByDefaultOverride, identifierPrefix,
onRecoverableError);
  }
  function createHydrationContainer(initialChildren, // TODO: Remove `callback` when we
delete legacy mode.
  callback, containerInfo, tag, hydrationCallbacks, isStrictMode,
concurrentUpdatesByDefaultOverride, identifierPrefix, onRecoverableError,
transitionCallbacks) {
    var hydrate = true;
    var root = createFiberRoot(containerInfo, tag, hydrate, initialChildren,
hydrationCallbacks, isStrictMode, concurrentUpdatesByDefaultOverride, identifierPrefix,
onRecoverableError); // TODO: Move this to FiberRoot constructor

    root.context = getContextForSubtree(null); // Schedule the initial render. In a
hydration root, this is different from
    // a regular update because the initial render must match was was rendered
    // on the server.
    // NOTE: This update intentionally doesn't have a payload. We're only using
    // the update to schedule work on the root fiber (and, for legacy roots, to
    // enqueue the callback if one is provided).

    var current = root.current;
    var eventTime = requestEventTime();
    var lane = requestUpdateLane(current);
    var update = createUpdate(eventTime, lane);
    update.callback = callback !== undefined && callback !== null ? callback : null;
    enqueueUpdate(current, update, lane);
    scheduleInitialHydrationOnRoot(root, lane, eventTime);
    return root;
  }
  function updateContainer(element, container, parentComponent, callback) {
    {
      onScheduleRoot(container, element);
    }

    var current$1 = container.current;
    var eventTime = requestEventTime();
    var lane = requestUpdateLane(current$1);

    {
      markRenderScheduled(lane);
    }

    var context = getContextForSubtree(parentComponent);

    if (container.context === null) {
      container.context = context;
    } else {
      container.pendingContext = context;
    }

    {
      if (isRendering && current !== null && !didWarnAboutNestedUpdates) {
        didWarnAboutNestedUpdates = true;

        error('Render methods should be a pure function of props and state; ' +
'triggering nested component updates from render is not allowed. ' + 'If necessary,
trigger nested updates in componentDidUpdate.\n\n' + 'Check the render method of %s.',
getComponentNameFromFiber(current) || 'Unknown');
      }
    }

    var update = createUpdate(eventTime, lane); // Caution: React DevTools currently
depends on this property
    // being called "element".

    update.payload = {
      element: element
    };
    callback = callback === undefined ? null : callback;

    if (callback !== null) {
      {
        if (typeof callback !== 'function') {
          error('render(...): Expected the last optional `callback` argument to be a ' +
'function. Instead received: %s.', callback);
        }
      }

      update.callback = callback;
    }

    var root = enqueueUpdate(current$1, update, lane);

    if (root !== null) {
      scheduleUpdateOnFiber(root, current$1, lane, eventTime);
      entangleTransitions(root, current$1, lane);
    }

    return lane;
  }
  function getPublicRootInstance(container) {
    var containerFiber = container.current;

    if (!containerFiber.child) {
      return null;
    }

    switch (containerFiber.child.tag) {
      case HostComponent:
        return getPublicInstance(containerFiber.child.stateNode);

      default:
        return containerFiber.child.stateNode;
    }
  }
  function attemptSynchronousHydration$1(fiber) {
    switch (fiber.tag) {
      case HostRoot:
        {
          var root = fiber.stateNode;

          if (isRootDehydrated(root)) {
            // Flush the first scheduled "update".
            var lanes = getHighestPriorityPendingLanes(root);
            flushRoot(root, lanes);
          }

          break;
        }

      case SuspenseComponent:
        {
          flushSync(function () {
            var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

            if (root !== null) {
              var eventTime = requestEventTime();
              scheduleUpdateOnFiber(root, fiber, SyncLane, eventTime);
            }
          }); // If we're still blocked after this, we need to increase
          // the priority of any promises resolving within this
          // boundary so that they next attempt also has higher pri.

          var retryLane = SyncLane;
          markRetryLaneIfNotHydrated(fiber, retryLane);
          break;
        }
    }
  }

  function markRetryLaneImpl(fiber, retryLane) {
    var suspenseState = fiber.memoizedState;
    if (suspenseState !== null && suspenseState.dehydrated !== null) {
      suspenseState.retryLane = higherPriorityLane(suspenseState.retryLane, retryLane);
    }
  } // Increases the priority of thenables when they resolve within this boundary.


  function markRetryLaneIfNotHydrated(fiber, retryLane) {
    markRetryLaneImpl(fiber, retryLane);
    var alternate = fiber.alternate;

    if (alternate) {
      markRetryLaneImpl(alternate, retryLane);
    }
  }
  function attemptContinuousHydration$1(fiber) {
    if (fiber.tag !== SuspenseComponent) {
      // We ignore HostRoots here because we can't increase
      // their priority and they should not suspend on I/O,
      // since you have to wrap anything that might suspend in
      // Suspense.
      return;
    }

    var lane = SelectiveHydrationLane;
    var root = enqueueConcurrentRenderForLane(fiber, lane);

    if (root !== null) {
      var eventTime = requestEventTime();
      scheduleUpdateOnFiber(root, fiber, lane, eventTime);
    }

    markRetryLaneIfNotHydrated(fiber, lane);
  }
  function attemptHydrationAtCurrentPriority$1(fiber) {
    if (fiber.tag !== SuspenseComponent) {
      // We ignore HostRoots here because we can't increase
      // their priority other than synchronously flush it.
      return;
    }

    var lane = requestUpdateLane(fiber);
    var root = enqueueConcurrentRenderForLane(fiber, lane);

    if (root !== null) {
      var eventTime = requestEventTime();
      scheduleUpdateOnFiber(root, fiber, lane, eventTime);
    }

    markRetryLaneIfNotHydrated(fiber, lane);
  }
  function findHostInstanceWithNoPortals(fiber) {
    var hostFiber = findCurrentHostFiberWithNoPortals(fiber);

    if (hostFiber === null) {
      return null;
    }

    return hostFiber.stateNode;
  }

  var shouldErrorImpl = function (fiber) {
    return null;
  };

  function shouldError(fiber) {
    return shouldErrorImpl(fiber);
  }

  var shouldSuspendImpl = function (fiber) {
    return false;
  };

  function shouldSuspend(fiber) {
    return shouldSuspendImpl(fiber);
  }
  var overrideHookState = null;
  var overrideHookStateDeletePath = null;
  var overrideHookStateRenamePath = null;
  var overrideProps = null;
  var overridePropsDeletePath = null;
  var overridePropsRenamePath = null;
  var scheduleUpdate = null;
  var setErrorHandler = null;
  var setSuspenseHandler = null;

  {
    var copyWithDeleteImpl = function (obj, path, index) {
      var key = path[index];
      var updated = isArray(obj) ? obj.slice() : assign({}, obj);

      if (index + 1 === path.length) {
        if (isArray(updated)) {
          updated.splice(key, 1);
        } else {
          delete updated[key];
        }

        return updated;
      } // $FlowFixMe number or string is fine here


      updated[key] = copyWithDeleteImpl(obj[key], path, index + 1);
      return updated;
    };

    var copyWithDelete = function (obj, path) {
      return copyWithDeleteImpl(obj, path, 0);
    };

    var copyWithRenameImpl = function (obj, oldPath, newPath, index) {
      var oldKey = oldPath[index];
      var updated = isArray(obj) ? obj.slice() : assign({}, obj);

      if (index + 1 === oldPath.length) {
        var newKey = newPath[index]; // $FlowFixMe number or string is fine here

        updated[newKey] = updated[oldKey];

        if (isArray(updated)) {
          updated.splice(oldKey, 1);
        } else {
          delete updated[oldKey];
        }
      } else {
        // $FlowFixMe number or string is fine here
        updated[oldKey] = copyWithRenameImpl( // $FlowFixMe number or string is fine here
        obj[oldKey], oldPath, newPath, index + 1);
      }

      return updated;
    };

    var copyWithRename = function (obj, oldPath, newPath) {
      if (oldPath.length !== newPath.length) {
        warn('copyWithRename() expects paths of the same length');

        return;
      } else {
```

```
        for (var i = 0; i < newPath.length - 1; i++) {
          if (oldPath[i] !== newPath[i]) {
            warn('copyWithRename() expects paths to be the same except for the deepest
key');

            return;
          }
        }
      }

      return copyWithRenameImpl(obj, oldPath, newPath, 0);
    };

    var copyWithSetImpl = function (obj, path, index, value) {
      if (index >= path.length) {
        return value;
      }

      var key = path[index];
      var updated = isArray(obj) ? obj.slice() : assign({}, obj); // $FlowFixMe number or
string is fine here

      updated[key] = copyWithSetImpl(obj[key], path, index + 1, value);
      return updated;
    };

    var copyWithSet = function (obj, path, value) {
      return copyWithSetImpl(obj, path, 0, value);
    };

    var findHook = function (fiber, id) {
      // For now, the "id" of stateful hooks is just the stateful hook index.
      // This may change in the future with e.g. nested hooks.
      var currentHook = fiber.memoizedState;

      while (currentHook !== null && id > 0) {
        currentHook = currentHook.next;
        id--;
      }

      return currentHook;
    }; // Support DevTools editable values for useState and useReducer.


    overrideHookState = function (fiber, id, path, value) {
      var hook = findHook(fiber, id);

      if (hook !== null) {
        var newState = copyWithSet(hook.memoizedState, path, value);
        hook.memoizedState = newState;
        hook.baseState = newState; // We aren't actually adding an update to the queue,
        // because there is no update we can add for useReducer hooks that won't trigger
an error.
        // (There's no appropriate action type for DevTools overrides.)
        // As a result though, React will see the scheduled update as a noop and bailout.
        // Shallow cloning props works as a workaround for now to bypass the bailout
check.

        fiber.memoizedProps = assign({}, fiber.memoizedProps);
        var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

        if (root !== null) {
          scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
        }
      }
    };

    overrideHookStateDeletePath = function (fiber, id, path) {
      var hook = findHook(fiber, id);
      if (hook !== null) {
        var newState = copyWithDelete(hook.memoizedState, path);
        hook.memoizedState = newState;
        hook.baseState = newState; // We aren't actually adding an update to the queue,
        // because there is no update we can add for useReducer hooks that won't trigger
an error.
        // (There's no appropriate action type for DevTools overrides.)
        // As a result though, React will see the scheduled update as a noop and bailout.
        // Shallow cloning props works as a workaround for now to bypass the bailout
check.

        fiber.memoizedProps = assign({}, fiber.memoizedProps);
        var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

        if (root !== null) {
          scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
        }
      }
    };

    overrideHookStateRenamePath = function (fiber, id, oldPath, newPath) {
      var hook = findHook(fiber, id);

      if (hook !== null) {
        var newState = copyWithRename(hook.memoizedState, oldPath, newPath);
        hook.memoizedState = newState;
        hook.baseState = newState; // We aren't actually adding an update to the queue,
        // because there is no update we can add for useReducer hooks that won't trigger
an error.
        // (There's no appropriate action type for DevTools overrides.)
        // As a result though, React will see the scheduled update as a noop and bailout.
        // Shallow cloning props works as a workaround for now to bypass the bailout
check.

        fiber.memoizedProps = assign({}, fiber.memoizedProps);
        var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

        if (root !== null) {
          scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
        }
      }
    }; // Support DevTools props for function components, forwardRef, memo, host
components, etc.


    overrideProps = function (fiber, path, value) {
      fiber.pendingProps = copyWithSet(fiber.memoizedProps, path, value);

      if (fiber.alternate) {
        fiber.alternate.pendingProps = fiber.pendingProps;
      }

      var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

      if (root !== null) {
        scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
      }
    };

    overridePropsDeletePath = function (fiber, path) {
      fiber.pendingProps = copyWithDelete(fiber.memoizedProps, path);

      if (fiber.alternate) {
        fiber.alternate.pendingProps = fiber.pendingProps;
      }

      var root = enqueueConcurrentRenderForLane(fiber, SyncLane);
      if (root !== null) {
        scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
      }
    };

    overridePropsRenamePath = function (fiber, oldPath, newPath) {
      fiber.pendingProps = copyWithRename(fiber.memoizedProps, oldPath, newPath);

      if (fiber.alternate) {
        fiber.alternate.pendingProps = fiber.pendingProps;
      }

      var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

      if (root !== null) {
        scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
      }
    };

    scheduleUpdate = function (fiber) {
      var root = enqueueConcurrentRenderForLane(fiber, SyncLane);

      if (root !== null) {
        scheduleUpdateOnFiber(root, fiber, SyncLane, NoTimestamp);
      }
    };

    setErrorHandler = function (newShouldErrorImpl) {
      shouldErrorImpl = newShouldErrorImpl;
    };

    setSuspenseHandler = function (newShouldSuspendImpl) {
      shouldSuspendImpl = newShouldSuspendImpl;
    };
  }

  function findHostInstanceByFiber(fiber) {
    var hostFiber = findCurrentHostFiber(fiber);

    if (hostFiber === null) {
      return null;
    }

    return hostFiber.stateNode;
  }

  function emptyFindFiberByHostInstance(instance) {
    return null;
  }

  function getCurrentFiberForDevTools() {
    return current;
  }

  function injectIntoDevTools(devToolsConfig) {
    var findFiberByHostInstance = devToolsConfig.findFiberByHostInstance;
    var ReactCurrentDispatcher = ReactSharedInternals.ReactCurrentDispatcher;
    return injectInternals({
      bundleType: devToolsConfig.bundleType,
      version: devToolsConfig.version,
      rendererPackageName: devToolsConfig.rendererPackageName,
      rendererConfig: devToolsConfig.rendererConfig,
      overrideHookState: overrideHookState,
      overrideHookStateDeletePath: overrideHookStateDeletePath,
      overrideHookStateRenamePath: overrideHookStateRenamePath,
      overrideProps: overrideProps,
      overridePropsDeletePath: overridePropsDeletePath,
      overridePropsRenamePath: overridePropsRenamePath,
      setErrorHandler: setErrorHandler,
      setSuspenseHandler: setSuspenseHandler,
      scheduleUpdate: scheduleUpdate,
      currentDispatcherRef: ReactCurrentDispatcher,
      findHostInstanceByFiber: findHostInstanceByFiber,
      findFiberByHostInstance: findFiberByHostInstance || emptyFindFiberByHostInstance,
      // React Refresh
      findHostInstancesForRefresh:  findHostInstancesForRefresh ,
      scheduleRefresh:  scheduleRefresh ,
      scheduleRoot:  scheduleRoot ,
      setRefreshHandler:  setRefreshHandler ,
      // Enables DevTools to append owner stacks to error messages in DEV mode.
      getCurrentFiber:  getCurrentFiberForDevTools ,
      // Enables DevTools to detect reconciler version rather than renderer version
      // which may not match for third party renderers.
      reconcilerVersion: ReactVersion
    });
  }

  /* global reportError */

  var defaultOnRecoverableError = typeof reportError === 'function' ? // In modern
browsers, reportError will dispatch an error event,
  // emulating an uncaught JavaScript error.
  reportError : function (error) {
    // In older browsers and test environments, fallback to console.error.
    // eslint-disable-next-line react-internal/no-production-logging
    console['error'](error);
  };

  function ReactDOMRoot(internalRoot) {
    this._internalRoot = internalRoot;
  }

  ReactDOMHydrationRoot.prototype.render = ReactDOMRoot.prototype.render = function
(children) {
    var root = this._internalRoot;

    if (root === null) {
      throw new Error('Cannot update an unmounted root.');
    }

    {
      if (typeof arguments[1] === 'function') {
        error('render(...): does not support the second callback argument. ' + 'To
execute a side effect after rendering, declare it in a component body with
useEffect().');
      } else if (isValidContainer(arguments[1])) {
        error('You passed a container to the second argument of root.render(...). ' +
"You don't need to pass it again since you already passed it to create the root.");
      } else if (typeof arguments[1] !== 'undefined') {
        error('You passed a second argument to root.render(...) but it only accepts ' +
'one argument.');
      }

      var container = root.containerInfo;

      if (container.nodeType !== COMMENT_NODE) {
        var hostInstance = findHostInstanceWithNoPortals(root.current);

        if (hostInstance) {
          if (hostInstance.parentNode !== container) {
            error('render(...): It looks like the React-rendered content of the ' + 'root
container was removed without using React. This is not ' + 'supported and will cause
errors. Instead, call ' + "root.unmount() to empty a root's container.");
          }
        }
      }
    }
    updateContainer(children, root, null, null);
  };

  ReactDOMHydrationRoot.prototype.unmount = ReactDOMRoot.prototype.unmount = function ()
{
    {
      if (typeof arguments[0] === 'function') {
        error('unmount(...): does not support a callback argument. ' + 'To execute a side
effect after rendering, declare it in a component body with useEffect().');
      }
    }

    var root = this._internalRoot;

    if (root !== null) {
      this._internalRoot = null;
      var container = root.containerInfo;

      {
        if (isAlreadyRendering()) {
          error('Attempted to synchronously unmount a root while React was already ' +
'rendering. React cannot finish unmounting the root until the ' + 'current render has
completed, which may lead to a race condition.');
        }
      }

      flushSync(function () {
        updateContainer(null, root, null, null);
      });
      unmarkContainerAsRoot(container);
    }
  };

  function createRoot(container, options) {
    if (!isValidContainer(container)) {
      throw new Error('createRoot(...): Target container is not a DOM element.');
    }

    warnIfReactDOMContainerInDEV(container);
    var isStrictMode = false;
    var concurrentUpdatesByDefaultOverride = false;
    var identifierPrefix = '';
    var onRecoverableError = defaultOnRecoverableError;
    var transitionCallbacks = null;

    if (options !== null && options !== undefined) {
      {
        if (options.hydrate) {
          warn('hydrate through createRoot is deprecated. Use
ReactDOMClient.hydrateRoot(container, <App />) instead.');
        } else {
          if (typeof options === 'object' && options !== null && options.$$typeof ===
REACT_ELEMENT_TYPE) {
            error('You passed a JSX element to createRoot. You probably meant to ' +
'call root.render instead. ' + 'Example usage:\n\n' + '  let root =
createRoot(domContainer);\n' + '  root.render(<App />);');
          }
        }
      }

      if (options.unstable_strictMode === true) {
        isStrictMode = true;
      }

      if (options.identifierPrefix !== undefined) {
        identifierPrefix = options.identifierPrefix;
      }

      if (options.onRecoverableError !== undefined) {
        onRecoverableError = options.onRecoverableError;
      }

      if (options.transitionCallbacks !== undefined) {
        transitionCallbacks = options.transitionCallbacks;
      }
    }

    var root = createContainer(container, ConcurrentRoot, null, isStrictMode,
concurrentUpdatesByDefaultOverride, identifierPrefix, onRecoverableError);
    markContainerAsRoot(root.current, container);
    var rootContainerElement = container.nodeType === COMMENT_NODE ? container.parentNode
: container;
    listenToAllSupportedEvents(rootContainerElement);
    return new ReactDOMRoot(root);
  }

  function ReactDOMHydrationRoot(internalRoot) {
    this._internalRoot = internalRoot;
  }

  function scheduleHydration(target) {
    if (target) {
      queueExplicitHydrationTarget(target);
    }
  }

  ReactDOMHydrationRoot.prototype.unstable_scheduleHydration = scheduleHydration;
  function hydrateRoot(container, initialChildren, options) {
    if (!isValidContainer(container)) {
      throw new Error('hydrateRoot(...): Target container is not a DOM element.');
    }

    warnIfReactDOMContainerInDEV(container);

    {
      if (initialChildren === undefined) {
        error('Must provide initial children as second argument to hydrateRoot. ' +
'Example usage: hydrateRoot(domContainer, <App />)');
      }
    } // For now we reuse the whole bag of options since they contain
    // the hydration callbacks.


    var hydrationCallbacks = options != null ? options : null; // TODO: Delete this
option

    var mutableSources = options != null && options.hydratedSources || null;
    var isStrictMode = false;
    var concurrentUpdatesByDefaultOverride = false;
    var identifierPrefix = '';
    var onRecoverableError = defaultOnRecoverableError;

    if (options !== null && options !== undefined) {
      if (options.unstable_strictMode === true) {
        isStrictMode = true;
      }

      if (options.identifierPrefix !== undefined) {
        identifierPrefix = options.identifierPrefix;
      }

      if (options.onRecoverableError !== undefined) {
        onRecoverableError = options.onRecoverableError;
      }
    }

    var root = createHydrationContainer(initialChildren, null, container, ConcurrentRoot,
hydrationCallbacks, isStrictMode, concurrentUpdatesByDefaultOverride, identifierPrefix,
onRecoverableError);
    markContainerAsRoot(root.current, container); // This can't be a comment node since
hydration doesn't work on comment nodes anyway.

    listenToAllSupportedEvents(container);

    if (mutableSources) {
      for (var i = 0; i < mutableSources.length; i++) {
        var mutableSource = mutableSources[i];
        registerMutableSourceForHydration(root, mutableSource);
      }
    }

    return new ReactDOMHydrationRoot(root);
  }
  function isValidContainer(node) {
    return !!(node && (node.nodeType === ELEMENT_NODE || node.nodeType === DOCUMENT_NODE
|| node.nodeType === DOCUMENT_FRAGMENT_NODE || !disableCommentsAsDOMContainers  ));
  } // TODO: Remove this function which also includes comment nodes.
  // We only use it in places that are currently more relaxed.

  function isValidContainerLegacy(node) {
    return !!(node && (node.nodeType === ELEMENT_NODE || node.nodeType === DOCUMENT_NODE
|| node.nodeType === DOCUMENT_FRAGMENT_NODE || node.nodeType === COMMENT_NODE &&
node.nodeValue === ' react-mount-point-unstable '));
  }

  function warnIfReactDOMContainerInDEV(container) {
    {
      if (container.nodeType === ELEMENT_NODE && container.tagName &&
container.tagName.toUpperCase() === 'BODY') {
        error('createRoot(): Creating roots directly with document.body is ' +
'discouraged, since its children are often manipulated by third-party ' + 'scripts and
browser extensions. This may lead to subtle ' + 'reconciliation issues. Try using a
container element created ' + 'for your app.');
      }

      if (isContainerMarkedAsRoot(container)) {
        if (container._reactRootContainer) {
          error('You are calling ReactDOMClient.createRoot() on a container that was
previously ' + 'passed to ReactDOM.render(). This is not supported.');
        } else {
          error('You are calling ReactDOMClient.createRoot() on a container that ' + 'has
already been passed to createRoot() before. Instead, call ' + 'root.render() on the
existing root instead if you want to update it.');
        }
      }
    }
  }

  var ReactCurrentOwner$3 = ReactSharedInternals.ReactCurrentOwner;
  var topLevelUpdateWarnings;

  {
    topLevelUpdateWarnings = function (container) {
      if (container._reactRootContainer && container.nodeType !== COMMENT_NODE) {
        var hostInstance =
findHostInstanceWithNoPortals(container._reactRootContainer.current);

        if (hostInstance) {
          if (hostInstance.parentNode !== container) {
            error('render(...): It looks like the React-rendered content of this ' +
'container was removed without using React. This is not ' + 'supported and will cause
errors. Instead, call ' + 'ReactDOM.unmountComponentAtNode to empty a container.');
          }
        }
      }

      var isRootRenderedBySomeReact = !!container._reactRootContainer;
      var rootEl = getReactRootElementInContainer(container);
      var hasNonRootReactChild = !!(rootEl && getInstanceFromNode(rootEl));

      if (hasNonRootReactChild && !isRootRenderedBySomeReact) {
        error('render(...): Replacing React-rendered children with a new root ' +
'component. If you intended to update the children of this node, ' + 'you should instead
have the existing children update their state ' + 'and render the new components instead
of calling ReactDOM.render.');
      }

      if (container.nodeType === ELEMENT_NODE && container.tagName &&
container.tagName.toUpperCase() === 'BODY') {
        error('render(): Rendering components directly into document.body is ' +
'discouraged, since its children are often manipulated by third-party ' + 'scripts and
browser extensions. This may lead to subtle ' + 'reconciliation issues. Try rendering
into a container element created ' + 'for your app.');
      }
    };
  }

  function getReactRootElementInContainer(container) {
    if (!container) {
      return null;
    }

    if (container.nodeType === DOCUMENT_NODE) {
      return container.documentElement;
    } else {
      return container.firstChild;
    }
  }

  function noopOnRecoverableError() {// This isn't reachable because onRecoverableError
isn't called in the
    // legacy API.
  }

  function legacyCreateRootFromDOMContainer(container, initialChildren, parentComponent,
callback, isHydrationContainer) {
    if (isHydrationContainer) {
      if (typeof callback === 'function') {
        var originalCallback = callback;

        callback = function () {
          var instance = getPublicRootInstance(root);
          originalCallback.call(instance);
        };
      }

      var root = createHydrationContainer(initialChildren, callback, container,
LegacyRoot, null, // hydrationCallbacks
      false, // isStrictMode
      false, // concurrentUpdatesByDefaultOverride,
      '', // identifierPrefix
      noopOnRecoverableError);
      container._reactRootContainer = root;
      markContainerAsRoot(root.current, container);
      var rootContainerElement = container.nodeType === COMMENT_NODE ?
container.parentNode : container;
      listenToAllSupportedEvents(rootContainerElement);
      flushSync();
      return root;
    } else {
      // First clear any existing content.
      var rootSibling;

      while (rootSibling = container.lastChild) {
        container.removeChild(rootSibling);
      }
      if (typeof callback === 'function') {
        var _originalCallback = callback;

        callback = function () {
          var instance = getPublicRootInstance(_root);

          _originalCallback.call(instance);
        };
      }

      var _root = createContainer(container, LegacyRoot, null, // hydrationCallbacks
      false, // isStrictMode
      false, // concurrentUpdatesByDefaultOverride,
      '', // identifierPrefix
      noopOnRecoverableError);

      container._reactRootContainer = _root;
      markContainerAsRoot(_root.current, container);

      var _rootContainerElement = container.nodeType === COMMENT_NODE ?
container.parentNode : container;

      listenToAllSupportedEvents(_rootContainerElement); // Initial mount should not be
batched.

      flushSync(function () {
        updateContainer(initialChildren, _root, parentComponent, callback);
      });
      return _root;
    }
  }

  function warnOnInvalidCallback$1(callback, callerName) {
    {
      if (callback !== null && typeof callback !== 'function') {
        error('%s(...): Expected the last optional `callback` argument to be a ' +
'function. Instead received: %s.', callerName, callback);
      }
    }
  }

  function legacyRenderSubtreeIntoContainer(parentComponent, children, container,
forceHydrate, callback) {
    {
      topLevelUpdateWarnings(container);
      warnOnInvalidCallback$1(callback === undefined ? null : callback, 'render');
    }

    var maybeRoot = container._reactRootContainer;
    var root;

    if (!maybeRoot) {
      // Initial mount
      root = legacyCreateRootFromDOMContainer(container, children, parentComponent,
callback, forceHydrate);
    } else {
      root = maybeRoot;

      if (typeof callback === 'function') {
        var originalCallback = callback;

        callback = function () {
          var instance = getPublicRootInstance(root);
          originalCallback.call(instance);
        };
      } // Update
      updateContainer(children, root, parentComponent, callback);
    }

    return getPublicRootInstance(root);
  }

  var didWarnAboutFindDOMNode = false;
  function findDOMNode(componentOrElement) {
    {
      if (!didWarnAboutFindDOMNode) {
        didWarnAboutFindDOMNode = true;

        error('findDOMNode is deprecated and will be removed in the next major ' +
'release. Instead, add a ref directly to the element you want ' + 'to reference. Learn
more about using refs safely here: ' + 'https://reactjs.org/link/strict-mode-find-node');
      }

      var owner = ReactCurrentOwner$3.current;

      if (owner !== null && owner.stateNode !== null) {
        var warnedAboutRefsInRender = owner.stateNode._warnedAboutRefsInRender;

        if (!warnedAboutRefsInRender) {
          error('%s is accessing findDOMNode inside its render(). ' + 'render() should be
a pure function of props and state. It should ' + 'never access something that requires
stale data from the previous ' + 'render, such as refs. Move this logic to
componentDidMount and ' + 'componentDidUpdate instead.',
getComponentNameFromType(owner.type) || 'A component');
        }

        owner.stateNode._warnedAboutRefsInRender = true;
      }
    }

    if (componentOrElement == null) {
      return null;
    }

    if (componentOrElement.nodeType === ELEMENT_NODE) {
      return componentOrElement;
    }

    {
      return findHostInstanceWithWarning(componentOrElement, 'findDOMNode');
    }
  }
  function hydrate(element, container, callback) {
    {
      error('ReactDOM.hydrate is no longer supported in React 18. Use hydrateRoot ' +
'instead. Until you switch to the new API, your app will behave as ' + "if it's running
React 17. Learn " + 'more: https://reactjs.org/link/switch-to-createroot');
    }

    if (!isValidContainerLegacy(container)) {
      throw new Error('Target container is not a DOM element.');
    }

    {
      var isModernRoot = isContainerMarkedAsRoot(container) &&
container._reactRootContainer === undefined;

      if (isModernRoot) {
        error('You are calling ReactDOM.hydrate() on a container that was previously ' +
'passed to ReactDOMClient.createRoot(). This is not supported. ' + 'Did you mean to call
hydrateRoot(container, element)?');
      }
    } // TODO: throw or warn if we couldn't hydrate?
    return legacyRenderSubtreeIntoContainer(null, element, container, true, callback);
  }
  function render(element, container, callback) {
    {
      error('ReactDOM.render is no longer supported in React 18. Use createRoot ' +
'instead. Until you switch to the new API, your app will behave as ' + "if it's running
React 17. Learn " + 'more: https://reactjs.org/link/switch-to-createroot');
    }

    if (!isValidContainerLegacy(container)) {
      throw new Error('Target container is not a DOM element.');
    }

    {
      var isModernRoot = isContainerMarkedAsRoot(container) &&
container._reactRootContainer === undefined;

      if (isModernRoot) {
        error('You are calling ReactDOM.render() on a container that was previously ' +
'passed to ReactDOMClient.createRoot(). This is not supported. ' + 'Did you mean to call
root.render(element)?');
      }
    }

    return legacyRenderSubtreeIntoContainer(null, element, container, false, callback);
  }
  function unstable_renderSubtreeIntoContainer(parentComponent, element, containerNode,
callback) {
    {
      error('ReactDOM.unstable_renderSubtreeIntoContainer() is no longer supported ' +
'in React 18. Consider using a portal instead. Until you switch to ' + "the createRoot
API, your app will behave as if it's running React " + '17. Learn more:
https://reactjs.org/link/switch-to-createroot');
    }

    if (!isValidContainerLegacy(containerNode)) {
      throw new Error('Target container is not a DOM element.');
    }

    if (parentComponent == null || !has(parentComponent)) {
      throw new Error('parentComponent must be a valid React Component');
    }

    return legacyRenderSubtreeIntoContainer(parentComponent, element, containerNode,
false, callback);
  }
  var didWarnAboutUnmountComponentAtNode = false;
  function unmountComponentAtNode(container) {
    {
      if (!didWarnAboutUnmountComponentAtNode) {
        didWarnAboutUnmountComponentAtNode = true;

        error('unmountComponentAtNode is deprecated and will be removed in the ' + 'next
major release. Switch to the createRoot API. Learn ' + 'more:
https://reactjs.org/link/switch-to-createroot');
      }
    }

    if (!isValidContainerLegacy(container)) {
      throw new Error('unmountComponentAtNode(...): Target container is not a DOM
element.');
    }

    {
      var isModernRoot = isContainerMarkedAsRoot(container) &&
container._reactRootContainer === undefined;

      if (isModernRoot) {
        error('You are calling ReactDOM.unmountComponentAtNode() on a container that was
previously ' + 'passed to ReactDOMClient.createRoot(). This is not supported. Did you
mean to call root.unmount()?');
      }
    }

    if (container._reactRootContainer) {
      {
        var rootEl = getReactRootElementInContainer(container);
        var renderedByDifferentReact = rootEl && !getInstanceFromNode(rootEl);

        if (renderedByDifferentReact) {
          error("unmountComponentAtNode(): The node you're attempting to unmount " + 'was
rendered by another copy of React.');
        }
      } // Unmount should not be batched.


      flushSync(function () {
        legacyRenderSubtreeIntoContainer(null, null, container, false, function () {
          // $FlowFixMe This should probably use `delete container._reactRootContainer`
          container._reactRootContainer = null;
          unmarkContainerAsRoot(container);
        });
      }); // If you call unmountComponentAtNode twice in quick succession, you'll
      // get `true` twice. That's probably fine?

      return true;
    } else {
      {
        var _rootEl = getReactRootElementInContainer(container);

        var hasNonRootReactChild = !!(_rootEl && getInstanceFromNode(_rootEl)); // Check
if the container itself is a React root node.

        var isContainerReactRoot = container.nodeType === ELEMENT_NODE &&
isValidContainerLegacy(container.parentNode) &&
!!container.parentNode._reactRootContainer;

        if (hasNonRootReactChild) {
          error("unmountComponentAtNode(): The node you're attempting to unmount " + 'was
rendered by React and is not a top-level container. %s', isContainerReactRoot ? 'You may
have accidentally passed in a React root node instead ' + 'of its container.' : 'Instead,
have the parent component update its state and ' + 'rerender in order to remove this
component.');
        }
      }

      return false;
    }
  }

  setAttemptSynchronousHydration(attemptSynchronousHydration$1);
  setAttemptContinuousHydration(attemptContinuousHydration$1);
  setAttemptHydrationAtCurrentPriority(attemptHydrationAtCurrentPriority$1);
  setGetCurrentUpdatePriority(getCurrentUpdatePriority);
  setAttemptHydrationAtPriority(runWithPriority);

  {
    if (typeof Map !== 'function' || // $FlowIssue Flow incorrectly thinks Map has no
prototype
    Map.prototype == null || typeof Map.prototype.forEach !== 'function' || typeof Set
!== 'function' || // $FlowIssue Flow incorrectly thinks Set has no prototype
    Set.prototype == null || typeof Set.prototype.clear !== 'function' || typeof
Set.prototype.forEach !== 'function') {
      error('React depends on Map and Set built-in types. Make sure that you load a ' +
'polyfill in older browsers. https://reactjs.org/link/react-polyfills');
    }
  }
```

```
    setRestoreImplementation(restoreControlledState$3);
    setBatchingImplementation(batchedUpdates$1, discreteUpdates, flushSync);

    function createPortal$1(children, container) {
      var key = arguments.length > 2 && arguments[2] !== undefined ? arguments[2] : null;

      if (!isValidContainer(container)) {
        throw new Error('Target container is not a DOM element.');
      } // TODO: pass ReactDOM portal implementation as third argument
      // $FlowFixMe The Flow type is opaque but there's no way to actually create it.


      return createPortal(children, container, null, key);
    }

    function renderSubtreeIntoContainer(parentComponent, element, containerNode, callback)
  {
      return unstable_renderSubtreeIntoContainer(parentComponent, element, containerNode,
  callback);
    }

    var Internals = {
      usingClientEntryPoint: false,
      // Keep in sync with ReactTestUtils.js.
      // This is an array for better minification.
      Events: [getInstanceFromNode, getNodeFromInstance, getFiberCurrentPropsFromNode,
  enqueueStateRestore, restoreStateIfNeeded, batchedUpdates$1]
    };

    function createRoot$1(container, options) {
      {
        if (!Internals.usingClientEntryPoint && !true) {
          error('You are importing createRoot from "react-dom" which is not supported. ' +
  'You should instead import it from "react-dom/client".');
        }
      }

      return createRoot(container, options);
    }

    function hydrateRoot$1(container, initialChildren, options) {
      {
        if (!Internals.usingClientEntryPoint && !true) {
          error('You are importing hydrateRoot from "react-dom" which is not supported. ' +
  'You should instead import it from "react-dom/client".');
        }
      }

      return hydrateRoot(container, initialChildren, options);
    } // Overload the definition to the two valid signatures.
    // Warning, this opts-out of checking the function body.


    // eslint-disable-next-line no-redeclare
    function flushSync$1(fn) {
      {
        if (isAlreadyRendering()) {
          error('flushSync was called from inside a lifecycle method. React cannot ' +
  'flush when React is already rendering. Consider moving this call to ' + 'a scheduler
  task or micro task.');
        }
      }

      return flushSync(fn);
    }
    var foundDevTools = injectIntoDevTools({
      findFiberByHostInstance: getClosestInstanceFromNode,
      bundleType:  1 ,
      version: ReactVersion,
      rendererPackageName: 'react-dom'
    });

    {
      if (!foundDevTools && canUseDOM && window.top === window.self) {
        // If we're in Chrome or Firefox, provide a download link if not installed.
        if (navigator.userAgent.indexOf('Chrome') > -1 &&
  navigator.userAgent.indexOf('Edge') === -1 || navigator.userAgent.indexOf('Firefox') >
  -1) {
          var protocol = window.location.protocol; // Don't warn in exotic cases like
  chrome-extension://.

          if (/^(https?|file):$/.test(protocol)) {
            // eslint-disable-next-line react-internal/no-production-logging
            console.info('%cDownload the React DevTools ' + 'for a better development
  experience: ' + 'https://reactjs.org/link/react-devtools' + (protocol === 'file:' ?
  '\nYou might need to use a local HTTP server (instead of file://): ' +
  'https://reactjs.org/link/react-devtools-faq' : ''), 'font-weight:bold');
          }
        }
      }
    }

    exports.__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED = Internals;
    exports.createPortal = createPortal$1;
    exports.createRoot = createRoot$1;
    exports.findDOMNode = findDOMNode;
    exports.flushSync = flushSync$1;
    exports.hydrate = hydrate;
    exports.hydrateRoot = hydrateRoot$1;
    exports.render = render;
    exports.unmountComponentAtNode = unmountComponentAtNode;
    exports.unstable_batchedUpdates = batchedUpdates$1;
    exports.unstable_renderSubtreeIntoContainer = renderSubtreeIntoContainer;
    exports.version = ReactVersion;

  })));
```