Developing Concurrent Web Application with Effect Handlers in Links

Steven Chang

4th Year Project Report
Electronics and Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

TODO

Research Ethics Approval

Declaration

Acknowledgements

Table of Contents

Chapter 1

Introduction

Note: This paragraph might be better for abstract, need to change this paragraph

This report explores the use of effect handlers in web development using the Links programming language. The primary objective is to enhance the user-friendliness and accessibility of concurrent programming with effect handlers by allowing users to easily define and switch their own handlers within the client code. The project achieves this by implementing various user-level thread schedulers using Links and effect handlers, with a focus on enabling users to switch between different effect handlers with minimal changes to the client code. The project successfully demonstrates a proof of concept that encapsulates effect handlers as independent modules, offering a more efficient and flexible approach for concurrent programming in the context of web development using effect handlers.

1.1 Motivations

Links [20] is a functional programming language that aims to simplify web programming by providing a unified language for all tiers of a web application, which includes the client-side, server-side, and database. To achieve this, Links translates code into the appropriate languages for each tier, such as JavaScript for the browser, Java for the server, and SQL for the database. Additionally, Links supports features that facilitate web development, including manipulation of the Document Object Model (DOM) and event listeners [3]. Furthermore, Links allows for concurrency on both the client and server-side [3], which enables users to create web applications that can handle multiple tasks simultaneously.

In contrast to conventional web development technologies, Links offers effect handlers [10] as an abstraction for managing effects such as exceptions, input/output operations, and state changes in a modular and composable way. An effect is defined as an operation that can be performed, while a handler is a function that specifies how to interpret those operations [10]. By separating the definition of effects from their implementation, it provides a modular and composable way of handling operations in a program, resulting in more flexible and maintainable code. One of the practical applications of effect handlers is its ability to express concurrency in programming languages.

Previous research on the use of effect handlers for concurrent programming has been mostly focused on the system level, despite the growing popularity of effect handlers in recent years. For instance, Multicore OCaml has provided a concurrency library that is implemented with effect handlers [4]. Links, on the other hand, is the first programming language that can build client-side web applications incorporating concurrency through the use of effect handlers.

Meanwhile, the field of web development has experienced a rapid growth over the past three decades, evolving from plain HTML, CSS, and JavaScript in 1995, to more powerful libraries like jQuery [28] in 2006, and then to modern front-end frameworks such as React [26] in 2013, followed by Vue [29] a year later. Nowadays, these frameworks are exploring new possibilities, with Vue introducing concurrency [30] and React exploring functional programming paradigms. Specifically, React developers have been inspired by algebraic effects and have made progress integrating similar features into the framework, such as React Hooks [25]. However, these features are more of a mimic of the effect handlers structure, as JavaScript does not yet fully support this feature.

Hence, compared with the other functional programming languages and web development technologies mentioned above, Links has its natural advantage as a functional programming language that is designed for building web applications and already supports the implementation of the effect handlers. As a result, this project aims to explore the possibility of effect handlers using Links in the context of web development. The primary focus of this project is to explore concurrent programming with effect handlers, which is becoming an important aspect of modern web development with all popular frameworks aiming to integrate concurrent mode into their production workflow.

1.2 Objectives

Previous work has shown that effect handlers in Links can be used to enable concurrency with user-level threads and schedulers [9]. However, there is still room for improvement in terms of the user-friendliness and accessibility of concurrent programming with effect handlers in Links. This project aims to enhance this aspect by allowing users to easily and conveniently define and switch their own handlers within the client code.

To achieve this, the primary objective of this project is to develop various user-level thread schedulers using Links and effect handlers, each having different scheduling schemes. Afterwards, these effect handlers will be integrated into the main application, allowing users to easily switch between them with minimal changes to the client code. Furthermore, several applications with distinct line rendering behaviors will be implemented as case studies to test and demonstrate the versatility of this switching scheme.

1.3 Contributions

In this project, I adapted from the existing code base racing_line.links [9] and accomplished the following objectives:

- Enhanced the user experience of racing_line.links
- Expanded the scheduler to support three different priorities and the the change of priority of the user-level threads
- Performed the separation of concerns to make the application code more readable and maintainable
- Extracted the effect interface into separate module
- Designed and implemented three different types of schedulers
- Modified client to allow seamless adding, removing, and switching between the effect handlers
- Designed and implemented three different applications for visualizing user-level threads as case studies to evaluate the switching scheme

With these contributions, I have successfully demonstrated a proof of concept demo that allows users to plug in different effect handlers into the client code with minimal effort. This highlights the feasibility of encapsulating effect handlers as independent modules and the ability to plug in different effect handlers for the same computational context without needing to rewrite the entire client code. The new structure offers a more efficient and flexible approach by keeping the client code generic when changing effect handlers for different behaviors.

1.4 Report Structure

The remainder of the report is structured as follows:

Chapter 2 provides an introduction to modern web development techniques and an overview of the Links programming language, including its syntax and relevant features. It also covers a discussion of effect handlers and concurrent programming using effect handlers, along with related work to provide context for the project.

Chapter 3 provides an in-depth discussion of the modifications made to the application, as well as the design and implementation details of different schedulers and case studies applications.

Chapter 4 evaluates the final work and discusses the challenges encountered during the development process.

Chapter 5 concludes the work and suggests possible future improvements.

Chapter 2

Background

2.1 Modern Web Development

The emergence of modern web development technologies marked a significant change in the evolution of the World Wide Web. With the advent of HTML, CSS, and JavaScript, these three technologies formed the backbone of modern web development. Hypertext Markup Language (HTML) [13] provided the basic structure and content of web pages, while Cascading Style Sheets (CSS) [31] is used to add styling to web pages, such as font, colors, and layout. JavaScript [14], on the other hand, enables the creation of interactive and responsive web applications. The integration of these three technologies has paved the way for the creation of functional and engaging websites that could be accessed from any device, anywhere in the world.

One of the very important and fundamental concepts in web development is the Document Object Model (DOM) [5]. It is an essential feature that allows developers to interact with HTML or XML documents dynamically. The DOM is a tree-like structure where each node represents an element in the document. With the DOM, developers can dynamically change the style, content, and structure of a web page in response to user interactions or other events.

Another fundamental concept in web development that is used closely with DOM is event listeners [32] which allow developers to write code that can respond to user events on a web page. An event is simply an action that happens in the browser, such as clicking, scrolling, or hovering over an element. By accessing these elements through the DOM and adding event listeners, developers can detect these events and trigger specific behaviors in response. This functionality allows web applications to respond to user input in real-time, enhancing the overall user experience.

In contrast to HTML, Links uses XML to create web pages and has its own set of DOM operations for XML documents and event listeners using l-event attributes [3]. Further details about the key features of Links will be explained in the next section.

2.2 Links

In traditional web development, developers are required to be proficient in a variety of programming languages, including HTML, JavaScript, Java, Python, and SQL, to build full-stack applications. For beginners, this can be particularly challenging, and the process of linking these languages together can be overwhelming. Moreover, as developers attempt to integrate the client, server, and database, they often encounter the impedance mismatch problem [3], which arises when data must be converted to the corresponding acceptable data types between each tier. This can be a significant limitation for developers seeking to build modern, efficient web applications.

Links was developed to address these issues. As a strict, typed functional programming language, it aims to overcome the challenges of mastering multiple programming languages and the impedance mismatch problem in web development. It provides a single source code for all tiers of a web application, including client-side, server-side, and database. However, due to its research-oriented nature, there are limited resources available for learning Links, and most of the available resources are in the form of example codes and Links Wiki page [21] on GitHub. This section aims to introduce some of the essential features of Links to enhance understanding of this project.

2.2.1 XML and DOM

In contrast to HTML, Links employs XML notations to construct web pages [3]. The syntax for the XML notations is similar to HTML, with the tag name enclosed in `<#>...</#>` syntax. The XML document is maintained by the client as a tree data structure, which is comparable to the DOM. Links offers two types of operations: DOM, which is mutable, and XML, which is for inspection only [3]. It also provides a set of conversion functions to convert custom data types to XML and insert them into the selected nodes. The full operations and functions reference can be accessed through the Links documentation [18].

2.2.2 Event Listeners

In order to enable interaction with the DOM, Links has provided syntax for defining event listeners in the XML code using attributes known as `l-event` attributes [18]. The format of these attributes is `l:name`, with `l:` serving as the prefix and `name` representing the event name. These `l-event` attributes must be assigned with a function that will be executed when the event is triggered [3]. For example,

```
<button l:onclick="onClick()">Click me</button>
```

This indicates the `onClick()` function is invoked when the `button` element is clicked. The complete list of event listeners available in Links can be found in the Links documentation [18].

2.2.3 Foreign Function Interface

In the context of web development, Links has the ability to call JavaScript functions directly within the Links program through the use of Foreign Function Interface (FFI) [19]. The FFI is made possible because modern web browsers run on JavaScript engines, and Links code on the client side is ultimately compiled into JavaScript code during runtime. By providing this feature, it offers extensive flexibility during the development process, enabling smooth collaboration between Links and JavaScript features, opening up new opportunities for developers.

A quick walk through on how to use JavaScript FFI in Links can begin by defining a function in a JavaScript file. The following code illustrates an example of how this can be done:

```
// js/log.js
function _logMessage(msg) {
    return console.log(msg);
}

var logMessage = LINKS.kify(_logMessage);
```

In the above example, `_logMessage(msg)` is a standard JavaScript function that logs the value of the `msg` variable in the console of the browser when it is called. This function is then wrapped inside a conversion function called `LINKS.kify` [19], which produces another function called `logMessage` that can be accessed in the Links program.

On the Links side, the JavaScript functions can be imported as a module (`Log`) for the entire program to access. To achieve this, the `alien` keyword is used in the module block that specifies the external language (`javascript`) and the file path of the JavaScript function. The following code block shows an example:

```
// log.links
module Log {
  alien javascript "js/log.js" {
    logMessage : (String) ~%~> ();
  }
}
```

Inside the `alien` block, the name and the types of the foreign function should be provided. Then the `logMessage` function can be assessed within the Links program by calling `Log.logMessage("Hello World").`

2.3 Effect Handlers

Algebraic effects handlers are a feature in functional programming where the concept of algebraic effects was first introduced by Plotkin and Power [22], and later, algebraic effect handlers was introduced by Power and Pretnar [23]. The purpose of the algebraic effects handlers is to structure effectful programs in a modular and compositional way. This is accomplished by separating the effects into operations for expressions and handlers for implementations (discussed in Section 2.3.1.1 and 2.3.1.2) using delimited continuations [7]. This approach enables the programs to pause, resume, and switch between different computation contexts, allowing them to express complex control-flow operations such as I/O, exception, state management, and concurrency. For consistency, the terms effect and effect handlers will be used instead of algebraic effect and algebraic effect handlers.

The concept of effect handlers was mostly focused on its theoretical aspects in the early stages of the development. However, in recent years, there has been a growing interest in the applicability of effect handlers. As a result, a variety of implementations of effect handlers have emerged, ranging from programming languages to libraries [2, 4, 11, 16].

In the effect handlers implementation, an effect is conceptualized as a signature of operations [17]. Operations are defined by the developers, and they can be thought of as abstract interfaces that describe the desired effect such as `Get` or `Set`. The concrete implementation of the operations, on the other hand, is defined by the effect handlers. Effect handlers provide a way to interpret these operations by defining a custom function that implements the behavior for the effectful operation. The following section will delve into the implementation details of effect handlers in Links.

2.3.1 Programming with Effect Handlers in Links

Note: Should I introduce the how effect handlers manage state in Links

In Links, a handler function is defined with one parameter: the program's computation context. The handler function then defines the operation definition for each effectful operation as well as its concrete implementation. The operation definition accepts two kinds of arguments: the payload passed into the function, and the captured continuation. The upcoming sections will provide a more detailed explanation of the concept of operations and handlers in Links.

2.3.1.1 Operation

Operation is like a function that will produce an output after being performed [10]. The operation itself does not have any semantical meaning towards the program, and the actual implementations for operation is defined entirely by the handlers. By convention, when defining an operation, the name should start with a capital letter and should not be left unhandled in the program or otherwise an error will be raised. In order to perform an operation, the syntax is `do Operation(arg1, arg2, ..., argn).`

2.3.1.2 Handlers

The syntax for implementing handlers in Links is similar as the switch statement in imperative programming or pattern matching in functional programming:

```
handle(m) {
    case <Op1 => k> -> # Implementation for Op1
    case <Op2(p1, p2,..., pn) => k> -> # Implementation for Op2
    case <Op3(p1, p2) => k> -> # Implementation for Op3
    ...
    case x -> x
}
```

The `handle` function takes in the computation context `m` as input. For every operation that is performed within the computation `m`, it will be mapped to the corresponding operation definition inside the `case` block for execution. The `case` block is composed of two elements: the operation definition, and its concrete implication. The operation definition can have any number of parameters, followed by a captured continuation `k` at the end, using the `=>` syntax. `k` is represented as a function that can take one or more arguments when invoking. By calling `k`, the control flow of the program is transferred back to the point in `m` where the operation was performed [10]. Finally, the `case x -> x` statement is an implicit return statement that will be executed when the computation `m` finishes its execution.

2.3.2 Code Example in Links

A straightforward way to explain the implementation of effect handlers in Links is through a series of examples. We begin with a program that prints two strings "Hello" and "World" in sequence. Without effect handlers, it is straightforward to implement in Links:

```
fun printMessage() {
    print("Hello\n");
    print("World\n");
}
```

Which yields

```
Hello
World
() : ()
```

When you call `printMessage()`

2.3.2.1 Printing with Effect Handlers

In order to integrate effect handlers into this example, we can convert the `print` method into an operation called `Print` and then define a handler called `forward` that interprets this operation.

```
fun printMessage() {
    do Print("Hello\n");
    do Print("World\n")
}

fun forward(m) {
    handle(m()) {
        case <Print(val) => k> -> print(val); k(())
        case x -> ()
    }
}
```

To execute the function, we can call `forward(printMessage)` and it will yield the same result as the previous example:

```
Hello
World
() : ()
```

It first replaces `print` method into `Print` operation inside `printMessage` function, and then when executing the program, `printMessage` is passed to the `forward` handler as the computation `m`. Therefore, by invoking `printMessage` inside the handler, `Print` operation will be performed, then captured and handled by the `forward` handler.

The concrete implementation for each operation is defined in the `case` patterns inside `forward` handler. In this example, after performing `Print` operation, it will first be mapped to the `Print` case in the handler where it receives two arguments, `val` as the actual parameter being passed into `Print` operation, and `k` which is the continuation function. In the `Print` case, `print` method is invoked before the continuation function `k`, which means the execution will be in order. Therefore in the program `"Hello"` is printed before `"World"`.

2.3.2.2 Continuation

It might seems unnecessary to convert two lines of code into more complicated structures with effect handlers. However, a powerful feature in effect handlers is that the developers can manipulate the functions' order of executions in a program by making use of the continuation `k`.

Now on top of the previous example, if we want to print the same strings but show them in the reverse order without changing the order of invocation in `printMessage`, the continuation in effect handlers can be handy when dealing with this issue:

```
fun reverse(m) {
    handle(m()) {
        case <Print(val) => k> -> k(()); print(val)
        case x -> ()
    }
}
```

We defined a new handler called `reverse`, and then when interpreting `Print` operation inside the `reverse` handler, the continuation `k` is called first, which means `print(val)` will only be invoked after `k` is finished executing. By wrapping the program continuation into a function variable enables the developer to easily take control over the call stack of the program. Now if we call `reverse(printMessage)`, the result will be:

```
World
Hello
```

```
() : ()
```

As demonstrated, by simply switching different effect handlers to interpret the same operation, the order of printing can be controlled without modifying the `printMessage` function.

2.4 Concurrent Programming with Effect Handlers

While the section above only provides simple examples of effect handlers, it demonstrates some of their powerful features. Modularity, particularly, is highlighted as one of the strengths of effect handlers because they provide a way to separate the implementation from the expression of the effectful computation. Additionally, with the use of delimited continuation, which enables the pausing, resuming, and switching between computational contexts, effect handlers are ideal for implementing a user-level thread scheduler that can achieve concurrency.

On the other hand, effect handlers also offer the advantage of writing concurrent programs in direct style, making them more readable and easier to debug. Compared with continuous-passing styles (CPS) which is callback-oriented, using direct-style code can usually reach better performance, as there are fewer function calls and less overhead associated with managing continuations [1]. In Multicore Ocaml, relevant work had already been deployed by implementing asynchronous effects and their handlers to construct an asynchronous I/O library using direct style programming [4].

In Links, concurrent programming is supported by using message passing, known as actor model [8]. In the actor model, the basic building blocks of the concurrent computation are "actors", which can be thought of as lightweight agents or objects that can react to incoming messages and perform actions based on their internal state. Actors communicate with each other through asynchronous message-passing, which means that they send and receive messages without waiting for a response. This enables concurrency and prevents potential deadlocks and race conditions that can occur in traditional shared-memory concurrency models. Finally, Links programs are compiled into JavaScript using CPS translation [12] in order to preserve concurrency on the client side.

Given the fact that Links supports effect handlers, it presents an alternative to actor models for achieving concurrency in Links. Specifically, effect handlers in Links enable us to implement a user-level thread scheduler that can facilitate concurrent programming. The following section will first demonstrate a state-of-art implementation deployed in the field of web development that enables concurrent operations, followed by a web application developed in Links that illustrates the practicality of implementing concurrency using effect handlers in Links.

2.5 Related work

2.5.1 React and React Fiber

React is a popular JavaScript library for building user interfaces [26]. One of the key features of React is the creation of a virtual DOM [27], which allows it to efficiently update the browser DOM only where necessary, resulting in faster and more efficient rendering of components.

React Fiber [15] is a new implementation of React's core algorithm, which aims to make React faster, more efficient, and more intelligent. It is a complete rewrite of the older versions of React's reconciliation algorithm and is now the default reconciler for React 16 and newer versions. The name "Fiber" comes from the use of fibers to represent nodes in the DOM tree.

One of the primary features of React Fiber is its ability to pause, resume, and prioritize updates during the reconciliation process [6]. This allows React to interrupt an update in progress to handle a higher-priority update, and then resume the original update when the higher-priority update is complete. This makes React more responsive and efficient, especially in cases where there are a large number of updates or complex animations.

React Fiber's ability to prioritize updates is also critical for applications with complex UIs. By giving priority to updates that require immediate attention, React Fiber can ensure that the user interface remains responsive even when there are multiple updates happening simultaneously. Overall, React Fiber represents a significant improvement to the React library, making it more efficient, more responsive, and more capable of handling complex and dynamic user interfaces.

2.5.2 racing_lines.links

As with the React Fiber example discussed earlier, there is a similar implementation in Links called `racing_lines.links` [9] that mimics the interruption and resumption behaviors of user-level threads with different priorities. This program visualizes the user-level threads, also known as fibers, as lines that are rendered on the canvas. Throughout this report, the term "fiber" will be used to refer to these lightweight user-level threads for consistency.

The user interface of `racing_lines.links` contains a rectangular canvas, two drop-down menus and one button. The rectangular canvas at the top is where the fibers are displayed. The dropdown menu offers 6 colors and 2 priority levels, namely, "Low" and "High". Users can then select their desired line color and priority level for the fiber from the dropdown menu. Once they have made their selections, they can render the chosen fiber as a horizontal line on the canvas by clicking the "Draw Line" button.

In Figure 2.1 and 2.2, the process of creating four fibers is shown. First, the green fiber with low priority is created, followed by the red fiber with high priority, then the blue fiber with low priority, and finally the yellow fiber with high priority. It can be observed that, although the green fiber is the first to be created, it is interrupted during its rendering process by the red fiber because the red fiber has a higher priority.

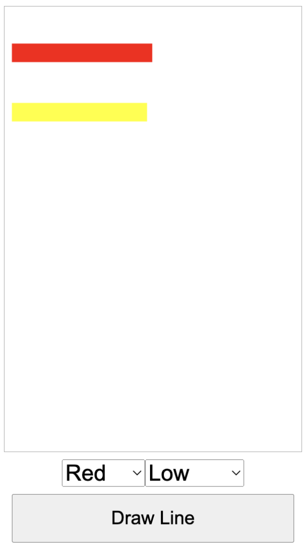

Figure 2.1: Rendering process of the High-priority fibers

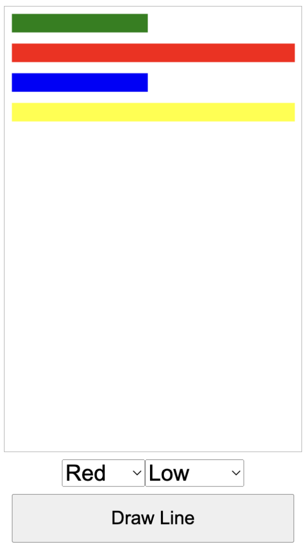

Figure 2.2: Rendering process of the Low-priority fibers

Similarly, the rendering process of the blue fiber is also blocked due to the fact that it has lower priority than the red fiber. On the other hand, since the yellow fiber has the same priority as the red fiber, both fibers will be rendered alternatively. The green and blue fibers can only resume their rendering process after the red and yellow fibers finish rendering and give control back to the lower-priority fibers. Then, they will resume their rendering process and render alternatively until they complete their rendering process.

The `racing_lines.links` program is implemented using effect handlers to create a fiber scheduler. The scheduler uses two operations, `Fork` and `Yield`, to create fiber instances and control their interruption and resumption. The fibers are maintained in a scheduler queue and will be popped out from the queue once they finish their execution.

Chapter 3

Design and Implementation

Section 2.5.2 has laid the groundwork for this project. In this section, we will delve into the specific details of how the schedulers were designed and implemented, as well as the modifications that were made to the application. Additionally, we will introduce the applications that were developed and used as case studies for this work.

3.1 Overview

The user interface design of the application has been revised to enhance the user experience. To simplify the process of creating fibers with different priorities, the color and priorities dropdown menu has been merged into one. Instead of allowing users to select from six different colors, each priority level can be assigned a specific color, such as red for Low priority and green for High priority. This change allows users to create fibers as lines on the canvas more efficiently with just one click instead of three. Additionally, each priority level has been assigned a specific color, making it easier for users to identify the state of the fibers. These improvements enable users to focus on the main task of the application, which is to create fibers and observe their behaviors when encountering fibers with different priorities that may preempt the control flow. A detailed description of the user interface implementation will be provided in Section 3.3.

To enhance the user experience of the application and show the strength of Links in concurrent programming, the user interface design can be further improved by providing more priority levels for fibers. In addition to Low and High, a Medium Fiber can be introduced, represented by the color blue. This new priority level would allow the fiber to preempt the control of a Low Fiber but would need to give up the control to High Fibers. This expansion of priority levels offers more possibilities for visualizing fiber behavior and allows users to explore a wider range of scenarios. More importantly, by introducing three priority levels, Links can demonstrate its superiority in concurrent programming compared to other modern front-end frameworks. For instance, React's Concurrent Mode [24] only supports two levels of priority due to the single-threaded nature of JavaScript. The specific implementation of the user interface with three priority levels will be discussed in Section 3.2.

In order to expand the capabilities of Links in concurrent programming, it is necessary to increase the level of control over the fiber scheduler. The goal is to have better control over the fibers, such as the ability to retrieve and modify the priority level of a fiber. Currently, the effect handler only provides the functionality to create a new fiber and yield control from one fiber to another. To achieve this objective, two new operations, namely `GetPrio` and `SetPrio`, can be defined and added to the effect handler. These new functionalities will enhance the versatility of the application and provide developers with more options to manipulate the fibers. Further details regarding the implementation of `SetPrio` and `GetPrio` will be presented in Section 3.4, highlighting the potential benefits of the Links programming language in concurrent programming.

Lastly, in the current implementation, all functionalities of the application, including the HTML, CSS code for the web interface, the JavaScript FFI for handling the drawing function on the canvas, Links code for maintaining the scheduler queue, as well as the fiber scheduler written in effect handlers, are all contained in one file. It would be more desirable to separate these functionalities into different modules, which aligns with standard software engineering coding practices. This would not only make the code more organized and easier to maintain, but also improve its reusability and extensibility in the long run. The details of how to modularize the code will be discussed in Section 3.5.

3.2 Three Priority Level

The addition of one more priority level is being implemented first as it is the core functionality of the program. Before discussing the technical details of how to add the additional priority level, a brief overview of how fibers are created, stored, and scheduled by the schedulers is necessary. Each fiber has two states, `prio` and `f`, representing its priority level and the function that will be executed when the fiber is scheduled. In the two priority levels setting, fibers are stored in separate priority queues for Low and High Priority. These two priority queues are combined into one data structure called `PrioQueue` for better management. During the enqueue stage, fibers are enqueued to their respective priority queues based on their `prio` value. During the dequeue stage, fibers with High Priority are dequeued first, followed by fibers with Low Priority only if the High Priority Queue is empty. This ensures that High Priority fibers can always run before Low Priority fibers when performing the `Yield` operation. A diagram illustrating how the `PrioQueue` work is presented in Figure 3.1.

The fiber scheduler, which is responsible for managing all the fibers, is implemented using effect handlers and interprets two operations: `Fork` and `Yield`. While Section 3.6 provides a detailed explanation of the fiber scheduler, this section focuses on priority processing. In the `Yield` operation definition, its implementation processes the rendering operations on the canvas and yields control to other fibers depending on the scheduling scheme. In the `Fork` operation definition, it determines whether the newly forked fiber should interrupt the current running fiber or be stored in the `PrioQueue` for later use using a switch statement. To add another priority level, it is necessary to modify the `PrioQueue` data structure to accommodate the new priority level, adjust the enqueue and dequeue logic accordingly, and change the implementation of the `Fork`

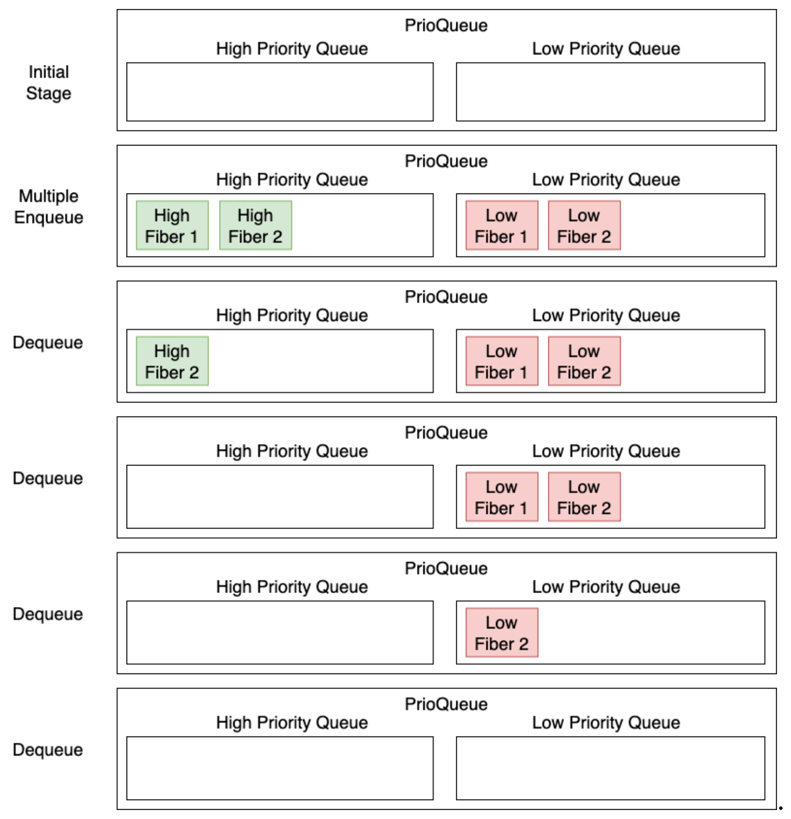

Figure 3.1: PrioQueue behavior for fibers with different priorities

and `Yield` operations.

The modifications made include defining a new queue for fibers with medium priority to the `PrioQueue` data structure, as well as modifying the `priorityEnqueue` and `priorityDequeue` functions to accommodate this change. The structure of the new `PrioQueue` has been illustrated in Figure 3.2.

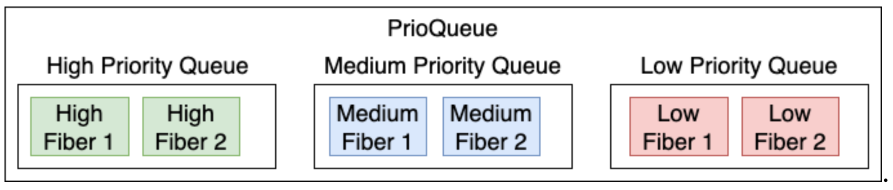

Figure 3.2: new PrioQueue structure

Furthermore, in order to integrate the new `PrioQueue` structure to the fiber scheduler, the `Fork` operation definition was also modified to incorporate the relationship between low, medium, and high priority fibers, allowing newly created fibers to preempt lower-priority fibers and enqueued into the `PrioQueue` if the current running fiber has a higher priority. The `Yield` operation was also updated to include the rendering process for fibers with medium priority. A more detailed explanation of the fiber schedulers can be found in Section 3.6.

3.3 User Interface Implementation

Now that the implementation of the three priority levels is complete, the focus can now shift towards improving the user interface of the application. As shown in Figure 3.3, the current process of creating a new fiber requires users to interact with two separate dropdown menus to select the color and priority of the fiber, followed by clicking the ”Draw Line” button, which involves three clicks. This process can be further simplified by reducing the number of clicks required to create a fiber to one, providing users with greater convenience and more straightforward control over the creation of fibers.

In the current implementation of the application, the `buttonPressed` function is linked to the `button` tag in the code using l-event attributes. This function will be executed when a click event on the button is detected. It retrieves the values for priority and color using the `getValueFromSelection` function, and then it uses switch statement to determine the priority level of the fiber, with the appropriate action taken based on the priority level. The code for the `buttonPressed` function is shown below:

```
fun buttonPressed(){
    var prio = getValueFromSelection("prio");
    var f = setUpLineDrawing(getValueFromSelection("line-color"));
    switch(prio){
        case "High" -> sysEnqueue(makeFiber(High, f))
        case "Medium" -> sysEnqueue(makeFiber(Medium, f))
        case "Low" -> sysEnqueue(makeFiber(Low, f))
        case _ -> ()
    }
}
```

Based on the design choice discussed in Section 3.1, the two dropdown menus and the ”Draw Line” button functionalities can be simplified into one single button for users. This new button should enable users to create a fiber with a fixed color and priority level. In order to achieve this, three buttons are needed, with each button responsible for creating fibers of a specific color and priority level. The implementation can be accomplished by replacing the HTML code for the dropdown menus and button with the new code for the three buttons. Additionally, since the dropdown menus are no longer necessary, the buttonPressed function can be updated to hardcode the values for priority and color. As a result, the existing code can be replaced from:

```
<div class="selection margin-10 center">
    <select id="line-color">
        <option value="red">Red</option>
        <option value="green">Green</option>
        <option value="blue">Blue</option>
        <option value="yellow">Yellow</option>
        <option value="#801638">Berry </option>
        <option value="#027878">Teal</option>
    </select>
    <select id="prio">
```

```
        <option value="Low">Low</option>
        <option value="Medium">Medium</option>
        <option value="High">High</option>
    </select>
</div>
<button class="block button center" l:onclick="{buttonPressed()}">
    Draw Line
</button>
```

to the new code as shown below:

```
<div class="selection margin-10 center">
    <button
        class="block button center"
        l:onclick="{buttonPressed("Low", "red")}">
        Draw Low Line
    </button>
    <button
        class="block button center"
        l:onclick="{buttonPressed("Medium", "blue")}">
        Draw Medium Line
    </button>
    <button
        class="block button center"
        l:onclick="{buttonPressed("High", "green")}">
        Draw High Line
    </button>
</div>
```

The proposed modifications to the user interface simplify the process of creating fibers and improve user experience by making the interface more intuitive and user-friendly. Additionally, these changes result in cleaner and more maintainable code. It is important to note that the function signature of `buttonPressed` is modified to receive hardcoded color and priority values as arguments, which leads to a revised implementation for `buttonPressed`. The new function implementation is shown below:

```
fun buttonPressed(prio, color){
    var f = setUpLineDrawing(color);
    switch(prio){
        case "High" -> sysEnqueue(makeFiber(High, f))
        case "Medium" -> sysEnqueue(makeFiber(Medium, f))
        case "Low" -> sysEnqueue(makeFiber(Low, f))
        case _ -> ()
    }
}
```

Finally, the user interface before and after the modifications are displayed in Figure 3.3 and Figure 3.4, respectively.

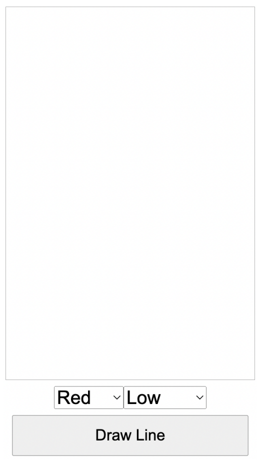

Figure 3.3: Before modification

Draw Low Line
Draw Medium Line
Draw High Line

Figure 3.4: After modification

3.4 Editing Priority of the Fiber

As I described in Section 2.3.1.2, defining an operation inside the effect handlers requires providing both of its definition and implementation. Defining the `GetPrio` operation is straightforward since the state of the fiber can be accessed by having an additional parameter in the handle function. Therefore, every time this operation is captured by the handlers, we can simply resume the current rendering process and return the current priority level back to the client.

The `SetPrio` operation is more challenging to implement than `GetPrio` since it requires updating both the priority level in the fiber's state and its position in the correct Priority Queue inside the `PrioQueue` data structure shown in Figure 3.2. To implement `SetPrio`, a parameter `p` for priority level is defined in the operation definition, and a helper function called `newPrioToFiber` is defined. This function takes `p` as an argument, creates a new fiber with the same information as the original fiber but with the priority level changed to the selected level, and enqueues this new fiber into the appropriate Priority Queue in `PrioQueue`. The scheduler will then yield control to the next fiber according to its scheduling scheme.

The implementation code for `GetPrio` and `SetPrio` operations is presented below. The variable `state` holds the current state of the fiber's scheduler, including its priority level and the `PrioQueue` data structure. This state is passed to the handler function, so that the operations can be executed within the context of the current fiber's state. As the fiber interacts with the operations defined in the program, the `state` parameter is updated throughout the execution of the fiber.

```
handle(fiber.f()) ( state <- (
                        prio=fiber.prio,
                        runQ=runQ
                    )){
    case x -> runNext(poll(state.runQ))
    case <GetPrio => resume> -> resume(state.prio, state)
    case <SetPrio(p) => resume> ->
```

```
            var q = fiberEnqueue(
                        newPrioToFiber(resume, p),
                        state.runQ
                    );
            runNext(poll(q))
    case <Fork(f) => resume> -> ...
    case <Yield => resume> -> ...
}
```

3.5 Modularization

The next step is to improve the overall quality of the program and make it more maintainable, readable and modular. To achieve this, the application can be first broken down into separate functionalities as follows:

- HTML and CSS code for the web user interface
- Links code for the fibers and fiber schedulers
- Links code for the scheduler queues, including its enqueue and dequeue functionalities
- Links code for the user interaction
- Links code for handling drawing functionalities
- JavaScript FFIs for accessing the actual canvas rendering functionalities and maintaining the fibers

The process of separation has been divided into two sections. Section 3.5.2 focuses on the modularization process, which separates each individual functionality into separate modules. On the other hand, Section 3.5.3 focuses on the separation process specifically in the context of effect handlers.

3.5.1 JavaScript FFIs

Before proceeding with the separation process, it is necessary to discuss the role played by JavaScript FFIs in the application. The main application involves two JavaScript FFIs: one to render the lines on the canvas using Canvas API (`canvas.js`), and the other to manage fibers in the JavaScript runtime by providing a similar scheduler queue structure in the `runtime.js`. While the purpose of `canvas.js` is straightforward, which is to provide Links access to the drawing methods in the Canvas API, the purpose of `runtime.js` may not be immediately obvious. It might seem redundant to define another scheduler queue in the JavaScript runtime when a `PrioQueue` structure is already defined within the Links program.

The main purpose of having this additional scheduler queue also known as the external scheduler queue in `runtime.js` is to temporarily store the newly created fibers triggered by button-click events. This external scheduler queue includes its own enqueue and dequeue functions. When a button is clicked, the application creates a new fiber using the makeFiber function. Instead of performing a Fork operation, the enqueue function of the external scheduler queue is called, and the new fiber is added to this queue. Then, when the scheduler decides to yield control to other fibers, it first calls the dequeue function of the external scheduler queue to load the fibers from the external queue into the actual PrioQueue. The scheduler then selects the next appropriate fiber in the PrioQueue to run according to its scheduling scheme. The entire process of fiber creation and yielding has been presented in Figure 3.5.

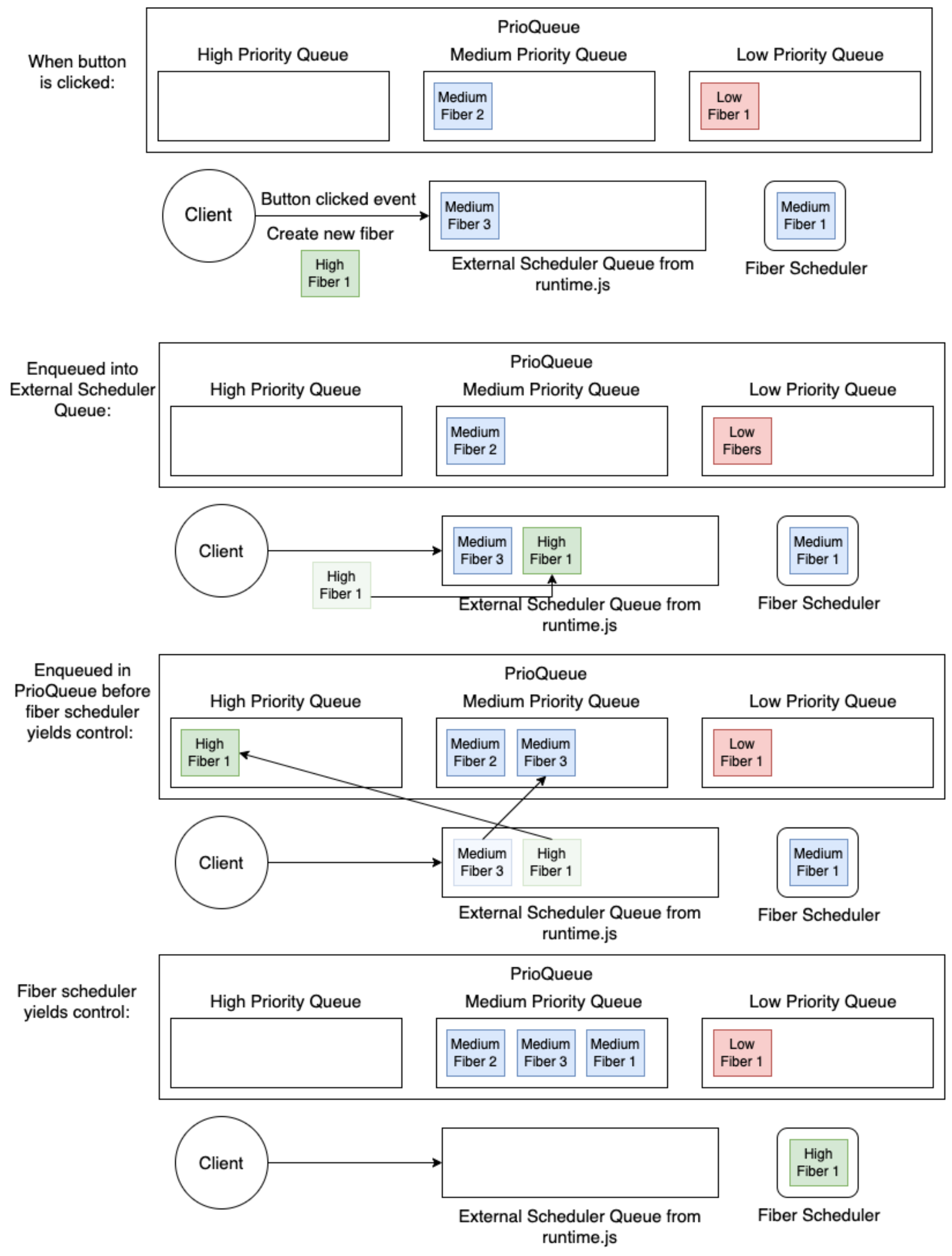

Figure 3.5: How external scheduler queue interacts with PrioQueue

3.5.2 Separation of Concerns

Prior to modularizing the application, it is important to analyze the relationships and dependencies between its functionalities. According to the functionalities listed earlier in Section 3.5, each one can be separated into its own module. For example, the main application should only contain the user interaction code, while other operations, such as drawing and styling, should be separated into independent modules. Additionally, the scheduler can be divided into separate modules for the scheduling scheme and the scheduler queue. This approach can help to manage the codebase more effectively and improve overall code quality.

After the separation process, the file `racingLines.links` will have dependencies on the files `racingLines.css` and `lineDrawer.links`, while `scheduler.links` will depend on `queue.links`. Furthermore, the two JavaScript files, namely `canvas.js` and `runtime.js`, are being used for the JavaScript FFI, which will be utilized in both `scheduler.links` and `racingLines.links`. The file structure of the application before and after the separation of concerns is shown in the left and right panels, respectively.

```
/
└── /js
│   ├── runtime.js
│   └── canvas.js
└── racing_lines.links
```

```
/
├── /js
│   ├── runtime.js
│   └── canvas.js
├── /css
│   └── racingLines.css
├── scheduler.links
├── LineDrawer.links
├── queue.links
└── racingLines.links
```

The separation of the code into individual functionalities not only allows developers to work on specific functionalities without conflicts, but also makes it easier to identify and resolve bugs, since issues can be isolated to specific modules, thereby improving the stability and reliability of the application.

3.5.3 Extract Effect Interface

To achieve the primary objective of providing a more convenient approach for the application to use different effect handlers on the same effectful operations, it is essential to extract the effect interface, which represents the signatures of the effectful operations, into its own module. This separation ensures the application executes on the same operation sets, regardless of the effect handlers (schedulers) we integrate. This allows to integrate various implementations of scheduling schemes on the same operations.

The application includes four operations, namely, such as `Fork` for creating a new fiber, `Yield` for yielding control to another fiber, and `GetPrio` and `SetPrio` for getting and setting the priority level of the fiber. These four operations are wrapped by corresponding effect interfaces and functions, which are extracted into a new file called fiberInterface.links. Additionally, certain utility functions such as makeFiber is also included in this file, as it is related to the functionality of fiber creation.

After this modification, racingLines.links and scheduler.links modules will incorporate the functionalities and types offered by the fiberInterface.links module to create and manipulate the fibers. The updated file structure is as follows:

```
/
├── /js
│   ├── runtime.js
│   └── canvas.js
├── /css
│   └── racingLines.css
├── scheduler.links
├── LineDrawer.links
├── queue.links
├── fiberInterface.links
└── racingLines.links
```

3.6 Schedulers

The heart of this project is the scheduler, which works in conjunction with the PrioQueue. To achieve our objective stated in Section 1.2, it is necessary to implement multiple schedulers with different fiber scheduling schemes. This serves as the foundation for the later implementation in Section 3.6.4 which focuses on enabling the easy switching of schedulers within the application. The objective is to make the application run smoothly while also making it easy for developers to switch between different effect handlers with minimal code changes. Ultimately, the aim is to simplify the process enough that only a few lines of code are necessary to switch between different effect handlers.

Before delving into the specific design and implementation of each different fiber scheduler, it is important to provide an overview of how a fiber scheduler operates in general. When a fiber is created, the scheduler compares its priority level with the currently running fiber to determine whether it should be executed. Then the unselected fiber is enqueued into the PrioQueue, while the selected fiber is executed by rendering a slice of lines on the canvas until the scheduler decides to yield control. If the executing fiber is not finished when control is yielded, it is re-enqueued to the PrioQueue. On the other hand, if it finishes its execution, it simply returns. The scheduler then selects the next fiber from the PrioQueue by calling its dequeue function. The entire process repeats until the queue is empty. The flowchart of the scheduler operation is illustrated in Figure 3.6. In this figure, the conditional block enclosed with the dashed line will be determined by the scheduling scheme from each scheduler. With this basic understanding, we can now proceed to explore the specific design and implementation of each different fiber scheduler.

The following sections present the design for the three schedulers that we developed: time-based, steps-based, and probability-based schedulers. Each section provides an overview of how the scheduler operates, along with a diagram to aid in understanding.

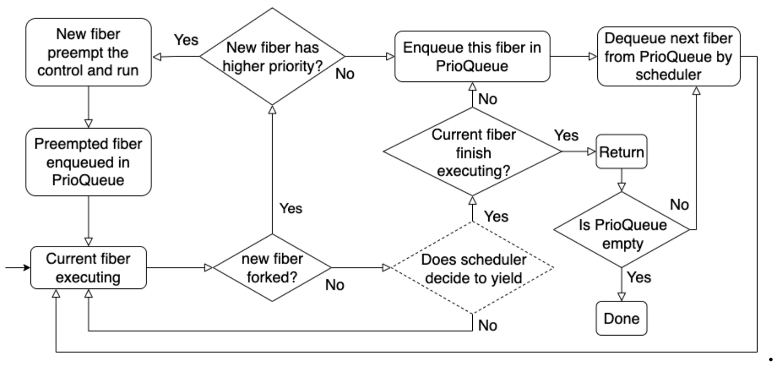
New fiber preempt the control and run
Yes
New fiber has higher priority?
No
Enqueue this fiber in PrioQueue
Dequeue next fiber from PrioQueue by scheduler
No
Preempted fiber enqueued in PrioQueue
Current fiber finish executing?
Yes
Return
No
Yes
Is PrioQueue empty
Yes
Current fiber executing
new fiber forked?
No
Does scheduler decide to yield
Yes
Done
No

Figure 3.6: Flowchart of how the fiber scheduler operates

We also present the actual implementation of the scheduler and the modifications made to the `PrioQueue` and effect handlers to achieve it.

3.6.1 Time-based Scheduler

The time-based scheduler is designed to allocate different execution times to fibers based on their priority levels. When yielded by the scheduler, high-priority fibers are allocated 400ms, medium-priority fibers are allocated 200ms, and low-priority fibers are allocated 100ms for execution. This approach enables higher-priority fibers to execute their operations for a longer period of time before yielding control, enabling them to complete their execution more quickly than lower-priority fibers.

The structure of the `PrioQueue` and the enqueue and dequeue functions remain the same as described in Section 3.2, where the fiber with the highest priority is dequeued first, and lower-priority fibers are dequeued only when the queues for higher-priority fibers are empty. The precise process of this scheduling scheme and the `PrioQueue` structure is illustrated in Figure 3.7 below. The modified part inside the scheduler is highlighted in yellow.

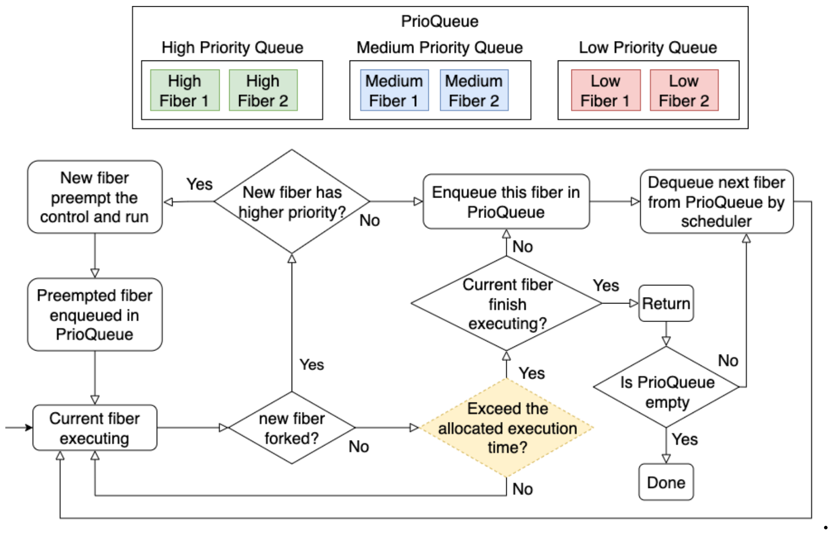
PrioQueue
High Priority Queue
Medium Priority Queue
Low Priority Queue
High Fiber 1
High Fiber 2
Medium Fiber 1
Medium Fiber 2
Low Fiber 1
Low Fiber 2
New fiber preempt the control and run
Yes
New fiber has higher priority?
No
Enqueue this fiber in PrioQueue
Dequeue next fiber from PrioQueue by scheduler
No
Preempted fiber enqueued in PrioQueue
Current fiber finish executing?
Yes
Return
No
Yes
Is PrioQueue empty
Yes
Current fiber executing
new fiber forked?
No
Exceed the allocated execution time?
Yes
Done
No

Figure 3.7: Time-based Scheduler

As can be observed, the modifications required for implementing the time-based scheduler are narrowed down to the `Yield` operation and the state structure attached to the handler, as the `PrioQueue` remains unchanged. Specifically, a new state field named `startTime`, which represents the start time of the fiber's execution in milliseconds, needs to be added. Upon yielding control, the `startTime` of the next running fiber is updated. Then, execution times are assigned to fibers according to their priority levels, and we must check if the fiber has used up its allocated execution time. If not, the fiber's execution is resumed; otherwise, control is yielded to the next fiber. The main code implementing these modifications is provided below.

```
handle(fiber.f()) ( state <- (
                        prio=fiber.prio,
                        runQ=runQ,
                        startTime=clientTimeMilliseconds())
                    ){
    ...
    case <Yield => resume> ->
        var currentTime = clientTimeMilliseconds();
        var buffer = switch(state.prio){
            case High -> 400
            case Medium -> 200
            case Low -> 100
        };
        if (currentTime - state.startTime <= buffer) {
            resume((), state)
        } else{
            var q = fiberEnqueue(
                        resumptionToFiber(resume, state.prio),
                        state.runQ
                    );
            runNext(poll(q))
        }
}
```

3.6.2 Step-based Scheduler

Step-based schedulers allocate different numbers of steps for fibers based on their priority levels, with high-priority fibers receiving 40 steps, medium-priority fibers receiving 20 steps, and low-priority fibers receiving 10 steps. Each step represents a segment of a line. Allocating more steps to a fiber allows it to render a longer line during its execution time.

This scheduler introduces a significant modification to the enqueue and dequeue behavior in the `PrioQueue`. Instead of prioritizing the dequeuing of higher-priority fibers, it implements a first-in, first-out (FIFO) policy like a classic queue. As a result, each fiber gets to run alternately without interruption from higher-priority fibers. The structure of the `PrioQueue` and the scheduling schemes is illustrated in Figure 3.8.

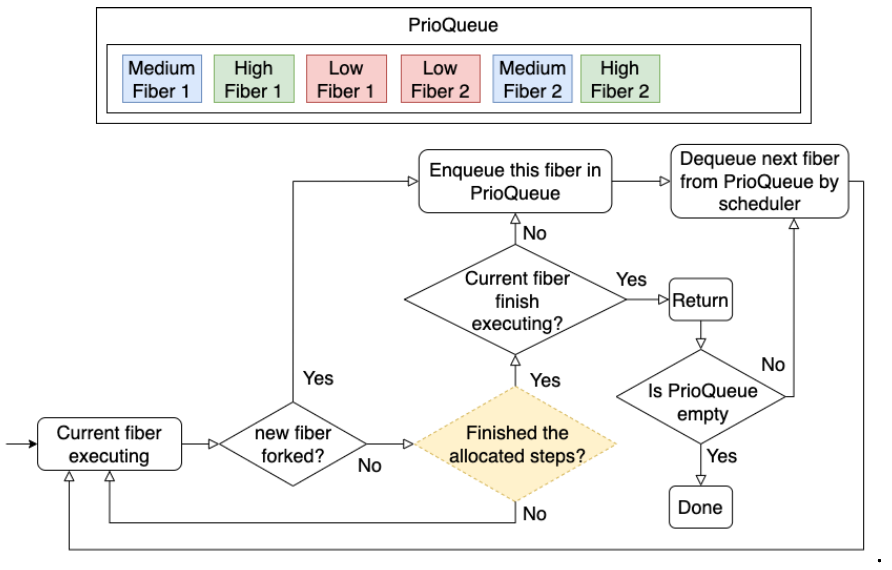

Figure 3.8: Step-based Scheduler

As depicted in the figure, this scheduler operates in a simpler way as there is no preemption when a new fiber is created, and the `PrioQueue` behaves like a standard queue. Regarding the scheduler aspect, we made modifications to both the `Fork` and `Yield` operations. The implementation for the `Fork` operation is much simpler, where the new fibers are directly enqueued into `PrioQueue` regardless of their priority levels. For `Yield` operation, a new state called `step` is added to keep track of the number of steps taken by the currently executing fiber. The scheduler yields control only when the fiber has completed all its allocated steps. The main code implementing these modifications is provided below.

```
handle(fiber.f())(state <- (prio=fiber.prio, runQ=runQ, step = 0)){
    ...
    case <Fork(f) => resume> ->
        var q = poll(state.runQ);
        var qq = fiberEnqueue(fiber0ToFiber(f), q);
        resume((), (state with runQ = qq))
    case <Yield => resume> ->
        var steps = switch(state.prio){
            case High -> 40
            case Medium -> 20
            case Low -> 10
            case None -> 0
        };
        var c = state.step + 1;
        if (c < steps) {
            resume((), (state with step = c))
        } else{
            var q = fiberEnqueue(
                        resumptionToFiber(resume, state.prio),
                        state.runQ
                    );
            runNext(poll(q))
        }
}
```

3.6.3 Probability-based Scheduler

Like rolling a dice, the probability-based scheduler selects the next fiber to run by randomly choosing the priority level and dequeuing the fiber from the corresponding priority queue. The High Priority Queue has a 50% chance of being chosen, the Medium Priority Queue has a 30% chance, and the Low Priority Queue has a 20% chance. The scheduling scheme is similar to the time-based scheduler discussed in Section 3.6.1, with the difference being that each fiber now has the same amount of allocated time (200ms) during its execution period. This setup ensures that even though each fiber has the same allocated execution time, higher priority fibers have a higher chance of being chosen and finishing their rendering sooner.

The `PrioQueue` structure remains unchanged in the probability-based scheduler, and the only modification is made to the dequeue scheme. Instead of dequeuing the fibers from the highest nonempty priority queue, this scheduler dequeues fibers from a particular priority queue determined by random selection based on the specified probabilities for each priority level. If the selected priority queue is empty, the scheduler attempts to dequeue fibers from the highest priority queues to the lowest until the entire `PrioQueue` is emptied. For example, if the selected priority level is Medium but the corresponding priority queue is empty, the scheduler will attempt to dequeue fibers from the High Priority Queue. If that queue is also empty, then the scheduler will attempt to select the fiber from the Low Priority Queue. The structure of the `PrioQueue` and the scheduling scheme is demonstrated in Figure 3.9.

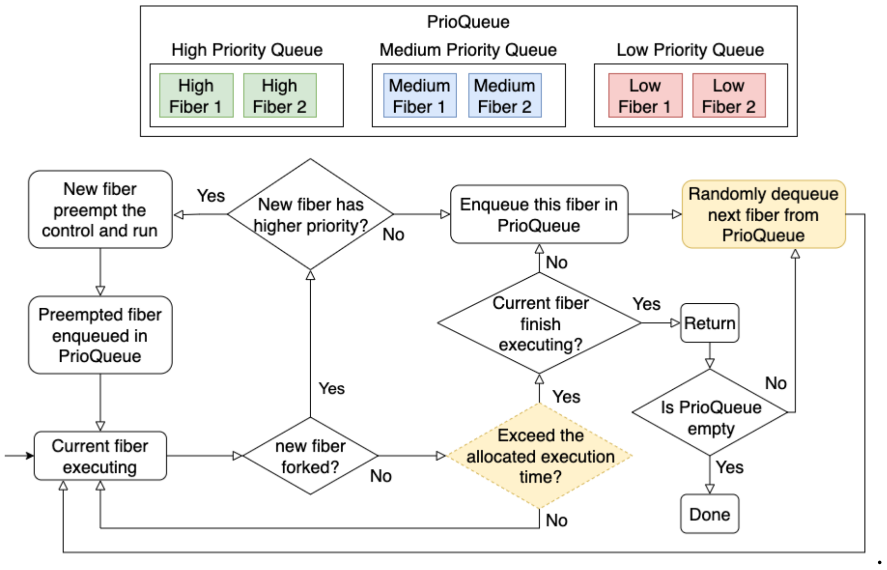
PrioQueue
High Priority Queue
Medium Priority Queue
Low Priority Queue
High Fiber 1
High Fiber 2
Medium Fiber 1
Medium Fiber 2
Low Fiber 1
Low Fiber 2
New fiber preempt the control and run
Yes
New fiber has higher priority?
No
Enqueue this fiber in PrioQueue
Randomly dequeue next fiber from PrioQueue
No
Preempted fiber enqueued in PrioQueue
Current fiber finish executing?
Yes
Return
Yes
No
Yes
Is PrioQueue empty
Current fiber executing
new fiber forked?
No
Exceed the allocated execution time?
Yes
No
Done

Figure 3.9: Probability-based Scheduler

The figure illustrates that the handler function's state is modified in a manner similar to the time-based scheduler discussed in Section 3.6.1. The implementation of the `Yield` operation is also similar to the time-based scheduler with a minor change in which fibers from all priority levels are allotted 200ms for execution. A crucial addition is the helper function called `getNextPrio`, which is included in the implementations of every operation that involves the invocation of `runNext`. `getNextPrio` randomly selects the priority level, which is then passed to `runNext` as an additional argument. Inside `runNext`, the dequeue function of the `PrioQueue` also uses this specified priority level as an additional argument to dequeue the fiber from the corresponding priority queue. The implementation of `getNextPrio`, `runNext` and the `Yield` operation is shown below:

```
fun getNextPrio() {
    var prob = floatToInt(random() *. 100.0);
    if (prob <= 20) {
        Low
    } else if (prob <= 50) {
        Medium
    } else {
        High
    }
}

fun runNext(q, prio){
    switch(fiberDequeue(q, prio)){
        case (Nothing, _) -> ()
        case (Just((fiber, prio)), q) ->
            fiber.f(makeSchedulerState(prio, q))
    }
}

case <Yield => resume> ->
    var currentTime = clientTimeMilliseconds();
    if (currentTime - state.startTime <= 200) {
        resume((), state)
    } else{
        var nextPrio = getNextPrio();
        var q = fiberEnqueue(
            resumptionToFiber(resume, state.prio),
            state.runQ
        );
        runNext(poll(q), nextPrio)
    }
```

3.6.4 Switch Schedulers

The next step after defining and implementing the three schedulers is to integrate them into the application and allow users to switch between them. To achieve this, the schedulers need to be parameterized as variables that can be passed as arguments to the application. Our solution is to define each scheduler as a subroute inside the main function and pass it as an argument to the page generation function. To facilitate this, three clickable links are added to the user interface, allowing users to switch between schedulers by clicking the appropriate link. Upon clicking a link, the browser refreshes and navigates to the designated subroute for the selected scheduler. The scheduler is then passed to the page generation function and used to initialize the schedulers. The modified code to enable this functionality has been provided.

```
- fun main_page(_){
+ fun main_page(schedule){
    - var pId = spawnClient{
                    Scheduler.schedule(makeFiber(High, start))
                };
    + var pId = spawnClient{
                    schedule(FiberInterface.makeFiber(High, start))
                };

    page
    <html>
    ...
    <body>
        + <a href="/scheduler1.links">Scheduler 1</a>
        + <a href="/scheduler2.links">Scheduler 2</a>
        + <a href="/scheduler3.links">Scheduler 3</a>
    </body>
    </html>
}

fun main() {
    ...
    - addRoute("/", main_page);
    + addRoute("/", fun(_) {
        main_page(Scheduler.schedule)
    });
    + addRoute("/scheduler1.links", fun(_) {
        main_page(Scheduler1.schedule)
    });
    + addRoute("/scheduler2.links", fun(_) {
        main_page(Scheduler2.schedule)
    });
    + addRoute("/scheduler3.links", fun(_) {
        main_page(Scheduler3.schedule)
    });
    servePages()
}
```

This modification greatly simplifies the process of adding or removing schedulers from the application. The developer only needs to add a subroute in the main function and a hyperlink on the UI that links to this subroute, which only requires two lines of code. Users can then switch between schedulers during runtime by clicking the hyperlink. This approach achieves our objective of making it easy for developers to switch and integrate different effect handlers with minimal changes to the code. The updated user interface is presented in Figure 3.10.

Draw Low Line
Draw Medium Line
Draw High Line
Scheduler 1
Scheduler 2
Scheduler 3

Figure 3.10: Improved user interface with new scheduler switch functionality

3.7 Case Studies Implementation

The objective now is to validate the general applicability of our scheduler switching approach by building three different applications for case studies that exhibit distinct rendering behaviors. These case studies are necessary because they ensure that our scheduler switching approach is not limited to a specific type of application and can work effectively across various contexts. Once the applications are developed, we will evaluate and test them in detail in Chapter 4 to demonstrate the ease and robustness of incorporating different schedulers and ensuring the correct functioning of the applications.

In general, the application works as follows: when the user clicks a button, the corresponding event listener triggers the `buttonPressed` function. This function creates the fiber based on the priority level, and the fiber is enqueued into the external scheduler queue in `runtime.js` through JavaScript FFI. The fiber is later dequeued and stored in the actual `PrioQueue` when the fiber scheduler yields control. The fiber has two properties: its priority level and a drawing function responsible for rendering lines on the canvas. During the fiber's execution, the rendering function is invoked, which draws the line slice by slice by calling the `drawCustomUnit` function in `canvas.js` through JavaScript FFI. The scheduler tries to yield control every time a slice of line is rendered. When the rendering process is complete, the function's execution finishes, and the fiber is returned.

3.7.1 Case Study 1

The first case study has a general and intuitive rendering behavior, where each fiber starts from the left and proceeds towards the right of the canvas. Once the line reaches the right edge of the canvas, it is considered finished and the fiber returns from the effect handlers. A visual representation of this behavior is provided in Figure 3.11. As illustrated, the low-priority fiber is created first and starts rendering. However, its execution is interrupted when the higher-priority fiber is created and takes control of the execution. The higher-priority fiber then renders until completion before the lower-priority fiber resumes rendering until it is also finished.

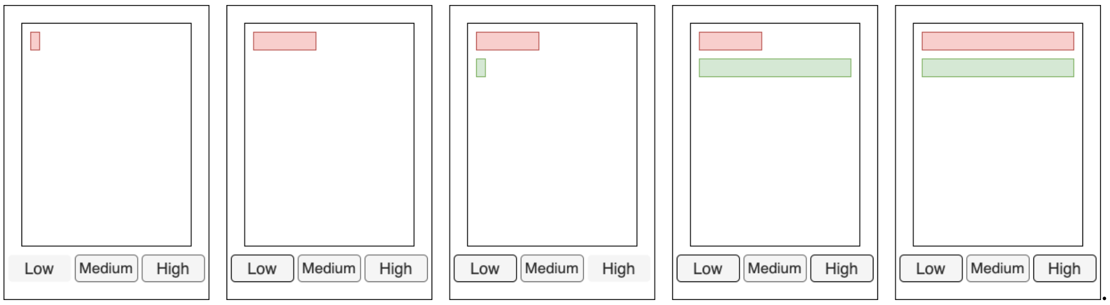

Figure 3.11: Rendering behavior of case study 1

The rendering function begins by retrieving the start and end coordinates of the line to be drawn. When the fiber is executing, the function `drawLineInColor` is called, which renders the line with the specified color and then calls the `drawHorizontalLine` function. This function first selects the canvas using DOM operation and defines an inner function called `aux`, which is responsible for rendering the line on the canvas slice by slice using the `drawCustomUnit` function through JavaScript FFI. The slices of the line are drawn progressively from left to right, with control being yielded to other fibers after each slice is rendered. Then, after the resumption of execution, `aux` delays the execution of the next slice of the line, allowing the line to be drawn progressively. This process repeats recursively with the coordinates being updated at each iteration until the entire line is drawn.

3.7.2 Case Study 2

The second case study is an extension of the first case study, but with a modification in the rendering behavior. In this case, after the line is rendered from the left to the right edge of the canvas, instead of returning the corresponding fiber, the line reverses its direction and moves back and forth repeatedly. As a result, the fiber never return. Figure 3.12 provides a visual representation of this rendering behavior to aid in understanding.

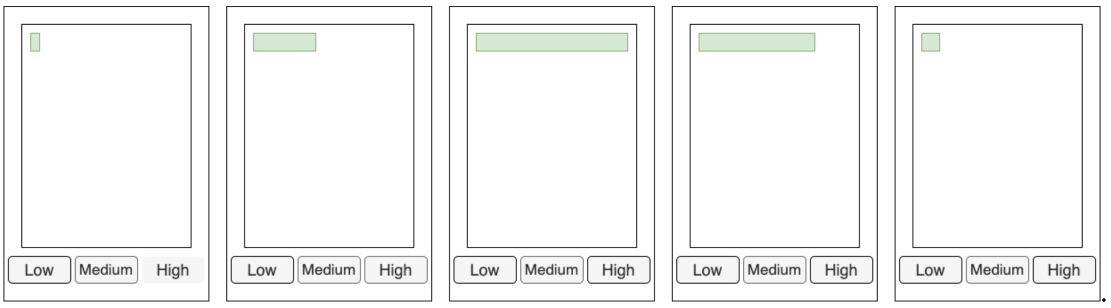

Figure 3.12: Rendering behavior of case study 2

Similarly, the implementation of this rendering behavior is also an extension of the implementation of case study 1, with the primary modification being made to the `drawLineInColor` function. Once the initial line is rendered from left to right using `drawHorizontalLine`, another additional invocation of `drawHorizontalLine` is made, but this time the line is rendered from right to left. After the line is drawn in both directions, `drawLineInColor` is recursively called to render the line back and forth repeatedly.

3.7.3 Case Study 3

In this case study, we investigate the effects of changing the priority level of fibers during runtime. Similar to case study 2, the line is rendered back and forth repeatedly after reaching the end. However, the key difference is that when the line is rendered from right to left, the priority level of the fiber is set to Low. When the line starts rendering from left to right again, the priority level is reset back to its original level. To illustrate this behavior clearly, Figure 3.13 has been included. This modification allows us to examine how changes in priority levels can affect the rendering behavior of the application.

In the given figure, an arrow is used to indicate the direction of rendering for the two lines that are rendered back and forth. The high-priority fiber interrupts the execution of the low-priority fiber when it renders the line from left to right. When the high-priority fiber renders the line from right to left, both fibers render alternatively as the priority level of the high-priority fiber has temporarily set to low. To achieve this behavior, in addition to the implementation of case study 2, the `var curPrio = FiberInterface.getPrio()` and `FiberInterface.setPrio(curPrio)` functions are used before and after the second `drawHorizontalLine` function, respectively.

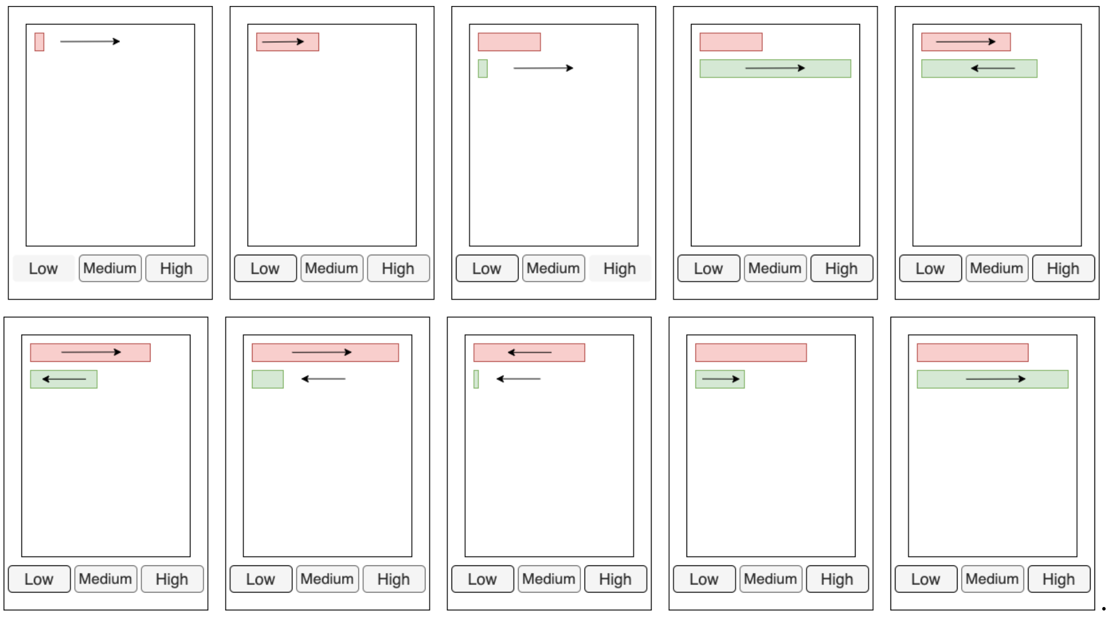

Figure 3.13: Rendering behavior of case study 3

Chapter 4

Evaluation

4.1 Case Studies

Note: Are figures necessary in this section?

This section evaluates the three schedulers described in Section 3.6 and the scheduler switch approach presented in Section 3.6.4 using the case study applications developed in Section 3.7. In this evaluation, we will create fibers in the following order for each scheduler: Low, Medium, High, Medium, High, Low. This ordering is chosen to demonstrate the interruption and resumption of the lower-priority fibers and validate the effectiveness of the scheduler switching approach across various applications.

4.1.1 Case Study 1

In the time-based scheduler, the two high-priority lines are rendered first, causing interruption to all other fibers, including the lower priority fibers that started rendering before them. Once the high-priority fibers finish execution, the medium-priority fibers start executing, followed by the low-priority fibers after the medium-priority fibers finish execution.

In the step-based scheduler, fibers are executed alternately in the same order as they were created. Typically, the high-priority fibers finish their execution faster due to the fact that they have more steps in their execution period. The medium-priority fibers finish in second place, followed by the low-priority fibers. However, it is possible for lower-priority fibers to finish rendering faster than higher-priority fibers by creating the higher-priority fibers much later than the lower-priority fibers since this scheduler has no interruption.

The probability-based scheduler is less predictable, with higher-priority fibers having a higher chance of finishing first, but with the possibility of lower-priority fibers finishing first in some cases. Under certain extreme test cases, such as creating one low-priority fiber followed by ten high-priority fibers, the low-priority fiber has a high probability of finishing first. This is because, although the High Priority Queue has a 80% chance of being selected (due to the empty Medium Priority Queue so the scheduler will also dequeue fibers from the High Priority Queue when the medium priority level is randomly selected), the ten high-priority fibers render alternately, which can slow down their individual rendering process. Meanwhile, the low-priority fiber is the only fiber in the Low Priority Queue and is thus always dequeued to render more efficiently when the low priority level is randomly selected by the scheduler.

4.1.2 Case Study 2

4.1.3 Case Study 3

4.2 Triumph

Note: Should I include triumph and challenges in conclusion or evaluation, the idea below seems to be a better fit under "analysis of result" section

Largely reduced the lines of code (from 444 to 129)

The canvas can hold maximum 15 lines, should add a scrollable functionality

By implementing the editing priority funcitonalities in effect handlers, I also found a way to allow users to pause and resume the fibers during runtime.

Make comparison with the original racing_line.links example such as the ease of user interaction and switching effect handlers

4.3 Challenges

No documentation of effect handlers, and several verison update during the development stage

Error messages can be very intimidating or too simple that doesn't help the debugging process

4.3.1 Mismatch Between Event Listeners and Effect Handlers

As this project mainly focuses on Links, minimizing or eliminating the dependency on JavaScript FFI has been attempted. Despite this effort, the use of `canvas.js` could not be avoided due to the lack of support for the Canvas API in Links, making it dependent on JavaScript. On the other hand, a significant amount of time and effort was invested in attempting to remove `runtime.js` from the application. This file contains functionalities such as managing fibers in the JavaScript runtime before enqueuing them into `PrioQueue`, which can, in theory, be replaced by the Links code.

Based on the introduction of `runtime.js` in Section 3.5.1 and the interaction between the external scheduler queue and the `PrioQueue` illustrated in Figure 3.5, it appears that the current implementation has resulted in the `Fork` operation being bypassed altogether. This goes against the principle of using effect handlers, as every action affecting the fiber should be carried out using operations. The use of `runtime.js` as JavaScript FFI for fiber management is a workaround that negates the use of operations. Therefore, there is a requirement to address the interpretation of the `Fork` operation and eliminate the use of the external scheduler queue in order to comply with the principles of using effect handlers.

One potential solution to address the bypassing of the `Fork` operation is to replace all enqueue functions with `Fork` operations. However, this approach may not work as expected since the event listener of the button is not handled as part of the computational context managed by the scheduler. As a result, this approach may lead to compile-time errors. Thus, a more nuanced solution is needed, which involves handling the event listener within the context of the scheduler to properly execute the `Fork` operation.

An alternative approach to solving the problem of bypassing the `Fork` operation is to use the actor model, which was introduced in Section 2.4 and is built into Links. The actor model replaces the event listener and uses a message-passing mechanism where each button clicks event dispatches a message to a receiver that executes the corresponding functionality based on the message. A comparison between the implementation of the actor model-based receiver and the event listener has been presented below.

```
// Agent model-based
fun receiver(){
    Scheduler.yield();
    receive {
        case CreateLowLine -> buttonPressed("Low", "red")
        case CreateMediumLine -> buttonPressed("Medium", "blue")
        case CreateHighLine -> buttonPressed("High", "green")
    };
    receiver()
}

fun main_page(_){

    var pId = spawnClient{
        Scheduler.schedule(Scheduler.makeFiber(High, receiver))
    };
      ...
    <button l:onclick="{pId ! CreateHighLine}">
        Draw High Line
    </button>
      ...
}

// Event listener based
fun main_page(){

    var pId = spawnClient{
        schedule(FiberInterface.makeFiber(High, start))
    };
    ...
    <button l:onclick="{buttonPressed("High", "green")}">
        Draw High Line
    </button>
    ...
}
```

By merging the three event listeners into one `receiver` function, it facilitates the handling of this function by the scheduler, which enables the possibility to perform the `Fork` operation. This approach eliminates the need to use `runtime.js`.

The approach described above, although promising, resulted in unexpected issues due to a false assumption made in the initial design. It was assumed that the `receiver` function listens for the message asynchronously, but in reality, the synchronous listening process of the `receiver` function blocks the rendering process, which in turn, blocks the entire scheduler when it is waiting for the message. As a result, if a button is clicked, the newly created fiber is only able to draw a portion of the line before yielding to the next fiber. Then, after the next fiber resumes its computation, the entire execution is blocked at the `receive` block until the user clicks the "Draw Line" button again. This behavior makes the implementation unsuitable for the intended purpose.

```
fun receiver(){
    Scheduler.yield();
    // Blocked until the block below receives new messages
    receive {
        case CreateLowLine -> buttonPressed("Low", "red")
        case CreateMediumLine -> buttonPressed("Medium", "blue")
        case CreateHighLine -> buttonPressed("High", "green")
    };
    receiver()
}
```

Although frustrating, This issue exposes the current limitations of wrapping a single effect handler around all event handlers in Links. In addition, there is a mismatch between effect handlers, event listeners, and actors, making it difficult to integrate the `receiver` function into the cooperative concurrency context of effect handlers. Therefore, currently, the only option is to use shared state and implement communication between event listeners and effect handlers through `runtime.js` using JavaScript FFI.

Chapter 5

Conclusion and Future Works

This project has successfully implemented an approach for switching effect handlers with minimal effort to interpret the same effectful operations under the same application context. To achieve this, the user experience of the application is improved, and the scheduler is expanded to support three different priorities and the changing of priority levels for the fibers. Separation of concerns is then performed to make the application code more readable and maintainable, and the effect interface is extracted into a separate module. Three different types of schedulers are then implemented to serve as the foundation for the seamless addition, removal, and switching of effect handlers. Additionally, three different case studies are designed and implemented to visualize user-level threads, which are then used to evaluate the switching scheme and its effectiveness.

While this project has successfully achieved the main objective stated in Section 1.2, the attempt to eliminate the use of `runtime.js` as JavaScript FFI to maintain the newly created fibers has not been fully achieved due to the mismatch between the effect handlers, event listeners, and the actors in Links. Nevertheless, this work provides valuable contributions to improving the user experience of defining, plugging in, and switching between different effect handlers within the same application, allowing the same effectful operations to have different implementations.

5.1 Future Works

There is still potential for further improvement in this project to enhance user experience and conduct a more comprehensive evaluation in the future. For example, given that `GetPrio` and `SetPrio` operations have already been defined, the project could explore implementing the pausing and resuming of fibers to demonstrate the power of effect handlers in developing concurrent web application. Additionally, to further demonstrate the applicability and versatility of the s effect handlers switching approach, a new set of completely different applications could be developed. Finally, a comparison between the same concurrent web applications written in Links and one written in React, could be conducted to provide a more comprehensive evaluation.

Bibliography

[1] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. "Attack of the Killer Microseconds". *Commun. ACM*, 60(4):48–54, mar 2017. `doi:10.1145/3015146`.

[2] Andrej Bauer and Matija Pretnar. "Programming with Algebraic Effects and Handlers". *CoRR*, abs/1203.1539, 2012. URL: `http://arxiv.org/abs/1203.1539`, `arXiv:1203.1539`.

[3] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. "Links: web programming without tiers". *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures.*, pages 266–296, 2006. `doi:10.1007/978-3-540-74792-5_12`.

[4] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K.C. Sivaramakrishnan, and Leo White. *"Concurrent System Programming with Effect Handlers"*, pages 98–117. Apr. 2018. `doi:10.1007/978-3-319-89719-6_6`.

[5] dom. "DOM". URL: `https://dom.spec.whatwg.org/`.

[6] flexiple. "Introduction to React Fiber". URL: `https://flexiple.com/react/react-fiber/`.

[7] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. "On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control". *Proceedings of the ACM on Programming Languages*, 1, 10 2016. `doi:10.1145/3110257`.

[8] Carl Hewitt. "Actor Model of Computation: Scalable Robust Information Systems", 2015. `arXiv:1008.1459`.

[9] Daniel Hillerström. "racing_lines.links". URL: `https://github.com/links-lang/links/blob/master/examples/handlers/racing-lines.links`.

[10] Daniel Hillerström. "Handlers for Algebraic Effects in Links". Master's thesis, Edinburgh, United Kingdom, 2015.

[11] Daniel Hillerström and Sam Lindley. "Liberating Effects with Rows and Handlers". In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, page 15–27, New York, NY, USA, 2016. Association for Computing

Machinery. doi:10.1145/2976022.2976033.

[12] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. "Continuation Passing Style for Effect Handlers". 84:18:1–18:19, 2017. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7739, doi:10.4230/LIPIcs.FSCD.2017.18.

[13] HTML. "HTML". URL: https://html.spec.whatwg.org/s.

[14] JavaScript. "JavaScript". URL: https://www.javascript.com/.

[15] Karthik Kalyanaraman. "A deep dive into React Fiber", 14 Mar, 2022. URL: https://blog.logrocket.com/deep-dive-react-fiber/.

[16] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in Action". *International Conference on Functional Programming*, pages 145–158, 2013. doi:10.1145/2500365.2500590.

[17] Sam Lindley. "Algebraic effects and effect handlers for idioms and arrows". *WGP 2014 - Proceedings of the 2014 ACM SIGPLAN Workshop on Generic Programming*, 08 2014. doi:10.1145/2633628.2633636.

[18] Links. "Docs". URL: https://links-lang.org/quick-help.html.

[19] Links-lang. "JavaScript FFI". URL: https://github.com/links-lang/links/wiki/JavaScript-FFI.

[20] Links-lang. "Links". URL: https://links-lang.org/.

[21] Links-lang. "Links wiki". URL: https://github.com/links-lang/links/wiki.

[22] Gordon Plotkin and John Power. "Adequacy for Algebraic Effects". volume 2030, Jan. 2001. doi:10.1007/3-540-45315-6_1.

[23] Gordon Plotkin and Matija Pretnar. "Handling Algebraic Effects". *Logical Methods in Computer Science*, 9(4), 2013. URL: https://doi.org/10.48550/arXiv.1312.1399, doi:10.2168/lmcs-9(4:23)2013.

[24] React. "Introducing Concurrent Mode". URL: https://17.reactjs.org/docs/concurrent-mode-intro.html.

[25] React. "Introducing Hooks". URL: https://reactjs.org/docs/hooks-intro.html.

[26] React. "React". URL: https://reactjs.org/.

[27] React. "Virtual DOM and Internals". URL: https://reactjs.org/docs/faq-internals.html.

[28] John Resig. "jQurey". URL: https://jquery.com/.

[29] Vue. "Vue". URL: https://vuejs.org/.

[30] Vue. "vue-concurrency". URL: https://vue-concurrency.netlify.app/.

[31] W3C. “CSS”. URL: https://www.w3.org/TR/CSS/.

[32] W3Schools. “JavaScript HTML DOM EventListener”. URL: https://www.w3schools.com/js/js_htmldom_eventlistener.asp.

Appendix A

Complete code for racing lines demo

A.1 racingLines.links

A.2 fiberInterface.links

A.3 lineDrawer.links

A.4 scheduler.links