Implementing UNIX with Effects Handlers

Ramsay Carslaw

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

2024

Abstract

Algebraic effect handlers first outlined by Plotkin and Prenar allow for a computation to be split into an effect signature and an implementation in the form of handler. Effect handlers allow for programs to be written in an extremely modular fashion by composing multiple effect handlers or having multiple handlers for one effect. This approach leads to programs that are written in an *effect-oriented* style where most the core functionality is an effectful computation.

UNIX is an operating system created at AT&T's Bell Labs in 1971 by Ritchie and Kernighan. It features a file system, user space and process management. It has become one of the most widely used operating systems, being licensed in Apple's macOS and served as the main inspiration for Linux.

This project provides an effect-oriented implementation of Unix based on Daniel Hillerström's toy UNIX he outlines in his PhD thesis. In this project, UNIX is implemented in Unison, a functional language with support for effect handlers. This initial Unison version of UNIX is then extended with more advanced features such as permissions, generic users and environment variables and a better scheduler. Both Unison and effect-oriented programming are analysed with the UNIX implementation serving as a sufficiently complex program to demonstrate some of the selling points of effect-oriented programming.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ramsay Carslaw*)

Acknowledgements

I would like to thank my supervisor, Sam Lindley, who has taught me so much and given up so much of his time to help with this project.

Thank you also to Daniel Hillerstöm who met with me on several occasions to help me understand both his own work and errors in my code.

Finally, thank you to my family, Koré and Alex for their support and listening to me talk about this project far too often.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Effect handlers [1] are widely becoming adopted in functional programming languages [2] and even imperative languages as libraries. They provide a unique and expressive syntax for handling side effects and control flow. This approach has led to a new paradigm of programming known as *effect-oriented programming* [3]. Effect-oriented programming leverages the way effect handlers can be composed to create modular programs that can be easily extended and maintained. The aim of this project is to demonstrate that complex programs, such as UNIX [4] can be implemented in this effect oriented style. The project details techniques and observations on effect oriented programming through the context of implementing a UNIX-like operating system.

We choose to implement UNIX as it is a well known and widely used operating system. Choosing an operating system also introduces more advanced control flow and concepts through time sharing, filesystems and pipes. The primary implementation language of this project is *Unison*, a functional programming language with first class support for effect handlers.

1.2 Aims

While the primary goal is to implement a UNIX-like operating system in Unison, the project also aims to compare the effect oriented programming style to traditional programming styles. In doing so, we will also aim to provide a commentary on the performance of effect handlers in Unison. The goal is not to write a real operating system but to demonstrate that effect handlers can be the right choice for complex programming tasks. As such, metrics like performance and scalability are of secondary concern.

1.3 Objectives

The objectives of the project are as follows:

- Provide a background and context for effect oriented programming and its intersection with operating systems
- Implement a UNIX-like operating system in Unison based on the work of Hillerström [5]
- Extend this initial version with more interesting uses of effect handlers and more advanced operating system features
- Provide a commentary on the performance of effect handlers in Unison
- Compare the effect oriented programming style to traditional programming styles
- Provide a reflection on the project and its outcomes

1.4 Outline

Chapter 2 introduces the literature and background of effect handlers and provides a simple example in Unison. We then discuss the state of effect oriented programming and some of the pros and cons of various effect implementations. Chapter 2 also describes some of the other research that has been done in this area.

Chapter 3 details the Unison implementation of a Hillerström's toy Unix operating system. It provides a basic implementation of a subset of UNIX features including: users, a filesystem , timesharing and pipes. This chapter also provides examples on using the implemented features.

Chapter 4 outlines features implemented beyond Hillerström's original implementation. These features include a more advanced scheduler, errors and exceptions as well as an overhauled userspace complete with environment variables.

Chapter 5 provides a commentary on the performance of effect handlers in Unison and details some of the challenges faced when implementing UNIX in this way. It also compares the effect oriented programming style to traditional programming styles in areas such as modularity, extensibility and performance.

Chapter 6 provides a reflection on the project and its outcomes. It details some of the challenges faced and the lessons learned. It also provides some ideas for future work in this area.

Chapter 2

Background

2.1 Algebraic Effects and Effect Handlers

Algebraic effects [1] and their corresponding handlers [6] [3] are a programming paradigm that when paired together offers a novel way to compose programs. It starts with the definition of the effect or the *effect signature* that gives the effect a name in scope and specifies any input and the return type otherwise known as the *effect operation*. For example, we might define the effect signature *State* that stores state for some type `a`. In order to make use of our *State* effect we can define the effect operations *put* and *get* where put will update the value of type `a` stored in *State* and *get* will return the current value. At this stage the effect operation has no implementation and is more an acknowledgement to the compiler that it should expect an implementation. For this reason any function that references these effect operations is known as an *effectful function* or a function whose definition is not complete without an effect handler. In the *put* and *get* example, any function that uses put and get to store values would be an effectful function. The *effect handler* provides one implementation of the given effect operation. We could define a simple handler for state that simply updates a variable of the given type or we could define a more complex one that uses hash maps. In this way, we can change the semantics of an effectful function by handling it with a different handler that provides an alternative implementation to the effect. Crucially, we can have multiple handlers defined in the same program for one effect allowing for much more modular programming or *effect-oreinted programming*.

2.1.1 Example in Unison

Unison [1] is a functional language implemented in Haskell that offers built in support for effect handlers through its abilities system.

Unison provides the *ability* keyword which allows users to define their own effects. It also provides the *handle ... with ...* pattern to attach handlers to effectful functions.

```
structural ability State a
  where
```

[1] https://github.com/unisonweb/unison

```
    put: a -> {State a} ()
    get: {State a} a
```

Listing 2.1: The *put* and *get* example in Unison. Note that the *structural* keyword refers to the fact that Unison stores type definitions as a hash. Even if we changed all the variable names it would still view it as the same type. To avoid that behaviour you can swap the *structural* keyword for *unique*

This defines the two effect operations `put` and `get` that have the effect signature `State a`. Put takes a value of type `a` and returns the unit type `()`. The prefix of {`State a`} to the `()`, refers to the fact that in order to allow for `put` to return, it must be run from an effectful function that is handled with an appropriate handler for `State a`, the need for this is discussed later in more detail. Similarly, `put` takes an argument of type `a` and must be handled.

```
addStore : Nat -> {State Nat} ()
addStore x =
  y = get ()
  put (x + y)
```

Listing 2.2: An example of an effectful function that uses the `State` effect

The code in Listing 2.2 is an example of how you would use the effects in Unison. It takes an argument of type `Nat` and adds it to the current value by using `get`. Note that we specify in the braces in the type signature that we are using a `State` effect that operates on `Nat`s or natural numbers.

```
    runState : a -> Request {Store a} a -> a
    runState value = cases
        {Store.get -> resume} -> handle resume value with runState
   value
        {Store.put v -> resume} handle resume () with runState v
        {result} -> result
```

Listing 2.3: The handler for the `State` effect

The handlers in Unison use tail recursion to reduce to the case where just the value is left `result -> result`. For both store and put we use the resumption and the handler to reach the final value. The special type `Request` allows us to perform pattern matching on the possible types of the computation.

```
handle (addStore 10) with storeHandler 10
```

Finally, we can put it all together by calling the function `addStore` with the handler `storeHandler`.

2.2 The State of Effect-Oriented Programming

There are many implementations of effect handlers, some of which are implemented as libraries and some are built into the language as first class operators.

2.2.1 Library Based Effects

- libhandler [7] is a portable c99 library that implements algebraic effect handlers for C. It implements high performance multi-shot effects using standard C functions. It is limited by the assumptions it makes about the stack such as it being contiguous and not moving. In practice this could lead to memory leaks if it copies pointers.
- libmprompt [2] is a C/C++ library that adds effect handlers. It uses virtual memory to solve the problem mentioned with libhandler. By keeping the stack in a fixed location in virtual memory it restores safety. It also provides the higher level libmpeff interface. A downside is they recommend at least 2GiB of virtual memory to allow for 16000 stacks which may be challenging on some systems.
- cpp-effects [8] is a C++ implementation of effect handlers. It uses C++ template classes and types to create modular effects and handlers. Its performance has been shown to be comparable to C++20 coroutines. its limitations are it only supports one-shot resumptions.
- There are several Haskell libraries that implement effect handlers [9, 10, 11]. Some of these libraries are discussed in more detail below.
  - EvEff uses lambda calculus based evidence translation to implement its effects system. It provides deep effects.
  - fused-effects [3] fuses the effect handlers it provides with computation by applying *fusion laws* that avoid intermediate representation. The handlers in fused-effects are one-shot however.

2.2.2 First-Class Effects

- Unison is shown in more detail in Section 2.1.1
- Koka [2] is a statically typed functional language with effect types and handlers. It can also compile straight to C code without needing a garbage collector. Koka is developed by a small team and as such is still missing much of its standard library.
- Frank [12] is a strict functional language that is *effectful* in that it has first class support for bi-directional effects and effect handlers.
- Links [13] is a functional programming language designed for the web. Out of the box it does not support true algebraic effects, however through an extension [14] it gains first class support for continuations.

[2]https://github.com/koka-lang/libmprompt
[3]https://hackage.haskell.org/package/fused-effects

2.3 Shallow vs. Deep Effect Handlers

There are two types of effect handler implementation, *deep handlers*, as originally defined by Plotkin and Pretnar [6] and *shallow handlers* [15]. Deep handlers pass a copy of the full handler along with the computation which allows for the handler to be invoked again as the handlers receive themselves as an argument. Shallow handlers do not pass the handler with the computation. There are also *sheep handlers*, which while being shallow implement some of the behaviour of deep handlers leading to the name sheep or shallow + deep. In practice, the type of handler is more of an implementation detail, to the programmer it mostly just effects how code is structured and leads to different patterns.

2.4 Affine and 'Multi-Shot' Handlers

If remaining computation or continuation of an effect can be resumed once from a handler then the effect system implements *one-shot* or *affine* effect handlers. If it is able to resume the computation multiple times then it is a *multi-shot* handler.

2.5 UNIX

UNIX [4] is an operating system designed and implemented by Dennis M. Ritchie and Ken Thompson at AT&T's Bell Labs in 1974. It provides a file system (directories, file protection etc.), a shell, processes (pipe, fork etc) and a userspace. Since its first release it has been reimplemented for a variety of systems such as macOS. It also heavily inspired GNU/Linux.

2.5.1 The UNIX Philosophy

A phrase often associated with UNIX is the *Unix philosophy*. The UNIX philosophy refers to some of the core principles with which it was developed. The core principles involve composing many small simple programs that accomplish one task well to solve more complex tasks [16]. The idea of many small modular components has spread to many areas of computer science.

2.6 Effect Based File Systems

Continuations in operating systems [17] are not a new concept. Kiselyov has demonstrated that algebraic effects can be used in a real file system and provide advanced features like snapshots, an undo operation and copy-on-write behaviour. Although this publication does not consider the performace of implementing features in this way it demonstrates that file systems can be built around continuations.

2.7 Effect Handlers and UNIX

In chapter 2 of his 2022 thesis, Daniel Hillerström [5] outlines a theoretical implementation of UNIX using an original calculus syntax. In this he provides an implementation of the original UNIX paper [4] that includes a filesystem and timesharing. Hillerström makes several assumptions about the effect system that would need to be taken into account in order to implement this with a real language. For example, he uses Kammar et. al [18] style deep handlers for most sections, however he also makes use of shallow handlers and parametrised handlers. Most effect handler implementations are limited to just one type of handler.

Chapter 3

Base Implementation

3.1 A Basic UNIX Implementation

This chapter outlines and details a Unison implementation of the toy UNIX written by Hillerström [5]. Hillerström's original is written in a fictional Lambda calculus that allows for using Shallow, Deep and Parameterised handlers. in the Unison implemenation I use only shallow handlers.

3.2 Program Status

In `Unix` programs must provide a code when they exit (usually 0 for success and anything else for failure). The effect signature `Status` provides the `exit` operation which takes one argument of type `Nat` [1] and returns the empty type which is defined `unique type Empty =`. Given the empty type has no implementation it has the effect of terminating the program wherever it is returned by `exit`, therefore exiting.The argument represents the return code.

```
unique ability Status
       where
            exit: Nat -> Empty
```

We can now use this to indicate program status. For example:

```
  --- some functionality

  if somethingWentWrong == true then
    exit 1
  else
    print "Hello, World"
```

There is no explicit exit 0 on the else branch. This is because the default state of a program should be 0, it should not need an explicit `exit 0`.

[1] a positive integer in Unison

3.2.1 Unique vs. Structural Types

In Unison, `unique` types are used when the name of the type is semantically important. The alternative is `structural` types which are used when the name of the type is not important and it can be stored as a hash without its name. `unique` types are used for all effects as it has no real implication given the program is not distributed.

3.2.2 The Handler

The handler for `Status` is defined as:

```
exitHandler : Request {e, Status} x -> Nat
exitHandler request =
    match request with
          { result } -> 0
          { exit v -> resume } -> v
```

The implementation for exit has no effect it simply consumes the exit code and returns. The handler however returns a `Nat` return code. If an exit operation is encountered we return the value given to the exit operation. The return case simply returns 0 as if we reach the end of a function being handled by the handler then we can assume it was successful and return 0. This means that even though the program is terminated with the empty type, we will still have access to the return code of the program through the handlers return type.

3.2.3 The `Request` Type

The `Request` type is a special type in Unison that allows for pattern matching on operations of an effect. In the braces are the effect types for the handler. The `Status` is the effect signature that is explicitly being handled. The `e` allows for any other effects in the computation to be passed through. The `x` is the return type of the computation.

3.3 Basic I/O

The *effect signature* `BasicIO` is used for simple I/O operations. The first and only *effect operation* of `BasicIO` is `echo` which takes an argument of type `Text` and returns the unit type `()`.

```
unique ability BasicIO where
  echo: Text -> ()
```

The handler for `BasicIO` is simply a wrapper for Unison's `putText` function which it uses to print the given text to `stdout`. It then handles the resumption with the same handler to handle any further `echo` calls.

```
basicIO : Request {BasicIO} x ->{IO, Exception} ()
basicIO result =
    match result with
        { echo text -> resume } ->
          putText stdOut text;
```

```
        handle resume () with basicIO
      { result } -> ()
```

3.3.1 IO and Exception abilities

The handler for `BasicIO` uses the `putText` function from Unison's standard library because of this we must include the {`IO, Exeption`} in the type signature to indicate that this function needs access to both the `IO` and `Exception` abilities in order to be run. Both of these abilities are built in and used for all input and output in unison.

Program 1 — Hello World

By combining the operations of Status and BasicIO we can write a simple program that prints "Hello, World!" and then exits with the successful error code. Notice that the operations are invoked in the same way as functions. In this case they are being used inside a function. It would be possible to implement a simple shell for these commands however that is outside the scope of this project.

```
greetAndExit : a ->{BasicIO, Status} ()
greetAndExit _ = echo "Hello, World!\n"; exit 0
```

By composing the two handlers in sections 3.2 and 3.3 we can run the program.

```
runGreetAndExit _ = handle (handle !greetAndExit with basicIO)
   with exitHandler
```

By running this function with the unison codebase manager we get

```
Hello, World!

  0
```

3.3.2 Defining Multiple Handlers

Effects are not limited to just one handler. The semantics of echo can be changed without altering its definition. For example, the `backwardsIO` handler below.

```
backwardsIO : Request {BasicIO} x ->{IO, Exception} ()
backwardsIO result =
  match result with
  { echo text -> resume } ->
    handle resume () with basicIO
    putText stdOut text;
  { result } -> ()
```

In this case, the resumption is handled first and then the text is printed. The effect of this is best shown by running it side by side with `basicIO` on the following program.

```
helloworld _ = echo "Hello,"; echo " World!\n"
```

The output of running this program with each handler is shown in Figure 3.1.

By handling the resumption first 'World' is printed first.

```
> handle !helloworld with
   basicIO

Hello, World!

  ()
```

```
> handle !helloworld with
   backwardsIO

 World!
Hello,
  ()
```

Figure 3.1: The output of running `helloworld` with each handler.

3.4 Users and Environment

To introduce the concept of a user-space and users we can start by adding some hard coded users. For now, alice, bob and a root user: `unique type User = Alice | Bob | Root`.

Next we introduce the `Session` signature for operations involving users. The operation `su` or ***substitute user*** is used to change the environment to that of a different user. The `ask` operation can be used to access environment variables. Since the only variable we have now is `USER` the argument to ask is a unit.

```
unique ability Session
    where
        su: User -> {Session } ()
        ask: () -> {Session } Text
```

We can now implement the UNIX command `whoami` with a wrapper around ask.

```
whoami: '{Session} Text
whoami _ = ask ()
```

We now have all the tools to keep track of which user is logged in and display that information:

Program 2 — Session Management

We can now compose the handlers we have written so far to switch between the users and invoke `whoami`.

```
    session _ = su Alice
                echo (!whoami)
                echo "\n"
                su Bob
                echo (!whoami)
                echo "\n"
                su Root
                echo(!whoami)
                echo "\n"
```

The function `runsession` simply invokes `session` using our `unix` function.

```
    runsession _ = handle (handle (handle (session) with env
   Root) with basicIO) with exitHandler
```

```
    alice
    bob
    root

      0
```

3.4.1 The Apostrophe in Unison

The ′ character in Unison is syntactic sugar for a function with a unit as the type of its first argument. For example, the type signature of `whoami` could be rewritten as `() ->{Session} Text`. This is equivalent to `'{Session} Text`.

3.4.2 Environment as a handler

The handler for `Session` also takes a user as an argument, this is the user that is currently logged in. To switch user we simply handle the rest of the computation with the new user provided as the argument to `su`. Then when the computation ends we will be back in the environment of the old user.

Due to the single environment variable being `USER`, `ask` performs the action of `whoami`. It keeps the user the same and returns the user as a string.

```
env: User -> Request {Session} a -> a
env user request =
    match request with
        {result} -> result
        { ask () -> resume } -> match user with
            Alice -> handle resume "alice" with env user
            Bob -> handle resume "bob" with env user
            Root -> handle resume "root" with env user
```

```
        {su user' -> resume} -> handle resume () with env user'
```

In this way the environment is the handler itself as it contains the information such as which user is logged in. The handler can be extended to have parameterised environment variables making it the complete environment.

3.4.3 Remark on Handlers as State

In this example, the handler replaces an algebraic datatype as state. When the user is substituted the handler handles the remaining computation with with the newly logged in user as an argument to itself. In this way the state only needs to be set when the handler is initially called i.e. `handle ... with env Root` and it is automatically managed for the whole program. This is discussed more in Section 5.2.

3.5 Nondeterminism

To implement the `fork` command from UNIX we can leverage deliberate non-determinism that is possible with effect handlers. We define the `fork` operation which returns a `Boolean` as a member of the `TimeSharing` signature.

```
unique ability TimeSharing
    where
        fork: Boolean
```

To use `fork` we can simply use it in control flow to create a branch. Where normally only one branch would be executed, the two branches become our two processes. For example:

```
if fork then
  echo "Heads\n"
else
  echo "Tails\n"
```

By running that code with the handler for `fork` we expect to get:

```
  Heads
  Tails

    0
```

The handler for fork is fairly simple:

```
nondet : Request {TimeSharing} a -> [a]
nondet request =
    match request with
        { fork -> resume } -> (handle resume true with nondet) lib.
   base.data.List.++ (handle resume false with nondet)
        { result } -> [result]
```

The handler returns a list of values with the type `a` which is the return type of the computation. When we encounter a `fork` we resume with the values true and false and join the two lists that are created. The return case wraps the value in a list so that we can use the ++ operator.

3.5.1 Remark on Joining Lists in Unison

Unison's typechecker sometimes struggles inferring the type of ++. For this reason we include the full path to the standard library where ++ is defined i.e. `lib.base.data.List.++`. This is discussed more in Section 5.1.1.

3.6 Scheduling

Now that we can create processes through `fork` it would be useful to be able to write scheduling algorithms. Currently `fork` will run the first process to completion, and then run the second process to completion. To begin scheduling we need to give the processes a method of stopping execution and giving control to the other process. We introduce the `Interrupt` signature with one operation also called `interrupt`.

```
unique ability Interrupt
    where
        interrupt: {Interrupt } ()
```

Now that we have `interrupt` we can write an alternative handler for `BasicIO` that will `interrupt` before every IO operation, allowing for the other process to run first.

```
interruptWrite : Request {e, BasicIO} x ->{e, Interrupt, BasicIO} ()
interruptWrite result =
    match result with
        { echo text -> resume } ->
            interrupt
            echo text
            handle resume () with interruptWrite
        { result } -> ()
```

Note that we still need to provide a handler for `echo`, this handler simply injects interrupt in front of every instance of `echo`.

In order to schedule processes we need to introduce state. Each process can either be `Done` (It has produced a return value) or `Paused` (It has been interrupted). `Paused` is a recursive definition as it contains a `PState` in its type.

```
unique type PState a e = Done a | Paused ('{e} PState a e)
```

The type `a` is the return type of the process and the `e` is an effect variable that represents any effects that are needed to run the `PState`. This can be thought of as analagous to the *resumption monad* first introduced by Milner in 1975 [19], in that computation is split into either the result or another computation i.e. the resumption. We can now implement the handler for `interrupt`.

```
reifyProcess : Request {Interrupt, e} a -> PState a e
reifyProcess request =
    match request with
```

```
        { interrupt -> resume } -> Paused (_ -> handle !resume with
    reifyProcess )
        { result } -> Done result
```

In the case of an interrupt, the handler suspends the computation by making it an anonymous function with a unit type as its first argument, and wrapping it in the `Paused` datatype. This means we can run the `Paused` computations later by invoking that function we created. The return case simply wraps the value in the `Done` type.

```
sched : [PState a {e, TimeSharing}] -> [a] ->{e} [a]
sched ps done =
    match ps with
        [] -> done
        (Done res) +: ps' -> sched ps' (res lib.base.data.List.+:
    done)
        (Paused m) +: ps' -> sched (ps' lib.base.data.List.++ (
    handle !m with nondet)) done

timeshare : '{e, Interrupt, TimeSharing} a ->{e} [a]
timeshare m = sched [Paused (_ -> handle !m with reifyProcess)] []
```

The `timeshare` function can be wrapped around a computation and will schedule set the first process as `Paused`. This can we wrapped around any function that uses `interrupt` or `fork` to handle them correctly.

3.7 Serial File System

3.7.1 State

To implement a file system we need to introduce a method of storing and retrieving state. The effect signature and operations introduced in section 2.1.1 provide the perfect interface as it takes a generic type `a` as an argument, we can introduce our own type to represent the filesystem and use it as an argument to `State`.

3.7.2 Definitions

File System — Unlike a real UNIX implementation we only implement the most basic operations on files, i.e. creation, deletion, reading and writing. Additionally we treat everything as a file, unlike UNIX which has directories and special files, we only allow basic files. Thus the file tree is completely flat.

Serial — Each file can only be read from in order, additionally when we write to file, there are no write modes, we only append to the file. Semantically, different write modes can be achieved with the four basic operations and can be implemented by composing handlers. For example, overwriting a file is equivalent to deleting the existing file, creating a new one with the same name and writing the content to the new file.

3.7.3 Types

```
unique type DirectoryT = Directory (Text, Nat)
unique type DataRegionT = DataRegion (Nat, Text)
unique type INodeT = INode Nat Nat
unique type IListT = IList (Nat, INodeT)
unique type FileSystemT = FileSystem (List DirectoryT) (List IListT)
    (List DataRegionT) Nat Nat
```

- `DirectoryT` — A directory stores a file name with its associated I-number
- `INode` — An I-Node stores the metadata for a file along with a pointer to a `DataRegion`
- `IList` — An I-List stores an I-number with an I-Node
- `DataRegion` — A DataRegion contains the actual file contents along with the pointer from the `INode`

Finally, the `FileSystem` type collects the above types into lists along with two `Nat`'s to represent the next directory number and the next I-number.

3.7.4 Initial File System

Much like `Root` is the initial user, we introduce an initial file system by initialising the types in Section 3.7.3. We create the file `stdout` to represent the standard output file at I-number 0.

```
initialINode : INodeT
initialINode = INode 0 0

initialDirectory : DirectoryT
initialDirectory = (Directory ("stdout", 0))


initialDataRegion : DataRegionT
initialDataRegion = DataRegion (0, "")

initialIList : IListT
initialIList = IList (0, initialINode)

initialFileSystem : FileSystemT
initialFileSystem = FileSystem [initialDirectory] [initialIList] [
   initialDataRegion] 0 0
```

3.7.5 Effect Types and Operations

Now we have the types and the state we can introduce the new effect signatures and operations. Firstly, `FileRW` which provides the `read` and `write`.

```
unique ability FileRW
  where
      read: Nat -> Text
      write: (Nat, Text) -> ()
```

read — Read takes an I-number and returns the text at the corresponding data region.

write – Write takes an I-number and some text and appends the text to the end of the data region pointed to by the I-number.

Next `FileCO` which is used for creating and opening files.

```
unique ability FileCO
  where
      create: Text -> Nat
      open: Text -> Nat
```

`FileCO` provides two operations, `create` and `open`.

create — Create takes a filename and returns a fresh I-number for the new file. If the provided filename exists it will overwrite the file to be blank again.

open — Open takes a filename and returns the I-number associated with it.

Finally, we have `FileLU` which links and unlinks files.

```
unique ability FileLU
    where
        link: (Text, Text) -> ()
        unlink: Text -> ()
```

link — Links two files such that changes to one happen to the other by making their I-Nodes point to the same data region.

unlink — Undoes the effect of link by making the two files have separate data regions again.

Program 3 – `mv`

We can now use these effect operations to define a `mv` command. While it can be used to move files between directories we have a flat file system so in this case its more of a rename command.

```
mv : Text -> Text ->{State FileSystemT, FileRW, FileCO} ()
mv src dest =
    let file = read (open src)
        _ = create src
        write ((create dest), file)
```

First, we use open to obtain the I-number of of the `INode` of the source file, we can then use this I-number as an argument to `read` to obtain the contents of the source file. Now that the contents of the source file are stored in the `file` varaible, we can safely delete the source file by calling `create` on it. If `create` is called on an existing file it will delete that file by overwriting it. Since we no longer need its I-number we assign the return value of the `create` to an empty variable. Finally, in one step we create the destination file (overwriting it if it exists) and write the value of the variable `file` to this new file using the I-number returned from `create`.

3.7.6 File System Handlers

The handler for `FileRW` is fairly simple. It takes a request and matches on the operations. If the operation is `read` it returns the text at the corresponding data region. If the operation is `write` it appends the text to the end of the data region pointed to by the I-number. It makes use of the underlying `fwrite` and `fread` functions that traverse the filesystem data structure, they are listed in the appendix. It will silently fail currently which we will address in the next chapter.

```
fileRW : Request {FileRW} a ->{State FileSystemT, Error} a
fileRW result =
    match result with
        { read i -> resume } ->
            let fs = get ()
                text = fread i fs
                match text with
                    Left text -> handle resume text with fileRW
                    Right () ->
                        handle resume "" with fileRW
        { write (i, text) -> resume } ->
            let fs = get ()
                fs' = fwrite i text fs
                put fs'
                handle resume () with fileRW
        { result } -> result
```

The handler for `FileCO` is also fairly simple. It takes a request and matches on the operations. If the operation is `create` it returns a fresh I-number for the new file. If the operation is `open` it returns the I-number associated with the filename. It makes use of the underlying `fcreate` and `fopen`.

```
fileCO : Request {FileCO} a ->{FileRW, State FileSystemT, Error} a
fileCO result =
    match result with
        { create name -> resume } ->
            let fs = get ()
                (ino, fs') = fcreate name fs
                put fs'
                handle resume ino with fileCO
        { open name -> resume } ->
            let fs = get ()
                ino = fopen name fs
                put fs
                handle resume ino with fileCO
        { result } -> result
```

Finally, `FileLU` which follows the same structure as the other handlers. The operation `link` takes a pair of filenames as it's argument and alters thier I-Nodes such that they both point at the same data region. This has the effect of 'linking' the files where changing or modifying one file will do the same as the other. The operation `unlink` operates in the same way, separating two linked files by creating a new I-Number and data region for the file in question. Once again these use underlying function `flink` and `funlink`.

```
fileLU : Request {FileLU} a ->{FileRW, State FileSystemT, Error} a
fileLU result =
    match result with
        { link (src, dest) -> resume } ->
            let fs = get ()
                fs' = flink src dest fs
                put fs'
                handle resume () with fileLU
        { unlink name -> resume } ->
            let fs = get ()
                fs' = funlink name fs
                put fs'
                handle resume () with fileLU
        { result } -> result
```

3.7.7 Unlisted Functions

This section references lots of functions that are not listed here i.e. `fcreate`, `fopen` etc. These functions all traverse and modify the `FileSystem` data type. They are all too long to be listed here but can be found in the Appendix.

3.8 Pipes

In UNIX, a pipe is essentially syntactic sugar for connecting the input and output of two files. Given the simple nature of the file-system described in Section 3.7, and the lack of true `stdout` and `stdin` files, pipes are represented as effect operations that are connected via handlers.

`Yield` and `Await` are two new effect signatures for implementing pipes. `Yield` performs some computation and returns or 'yields' a value, `Await` takes that value as an argument and then performs computation. Using the `yield` operation can be thought of as writing to `stdout` and `await` is reading from `stdin`.

```
unique ability Await a
    where
        await: () -> a

unique ability Yield a
    where
        yield: a -> ()
```

3.8.1 Cat

In UNIX, `cat` prints the contents of a file to `stdout`. In this case, pipes can be used, by yielding the file a character at a time other utilities can simply await input.

```
cat: Text -> {FileRW, FileCO, Yield Text, e} -> ()
cat fname =
  let ino = open fname
      iter (ch -> yield ch) (read ino)
      yield '\0'
```

3.8.2 Find

We define a new program `find` that searches for a string in a the output of a pipe. This is done by yielding the output of the pipe to the `find` function which then awaits the string to search for. If the string is found it yields true, otherwise it yields false.

```
find: Text ->{Await Text} Boolean
find target =
  findRec target buffer n length =
    if n < length then
      findRec target (buffer ++ !await) n+1 length
    else
      if buffer == target then
        True
      else
        if buffer == "" then
          False
        else
          findRec target ((drop 1 buffer) ++ !await) n length
  findRec target "" 0 (length target)
```

Program 4 – Searching in a set of files

Given a list of filenames we want to return the name of each file that contains a particular string. We can compose `cat` and `find` to achieve this.

```
searchFiles: Text -> [Text] ->{FileRW, FileCO, Await Text,
 Yield Text, e} [Text]
searchFiles target fnames = match fnames with
  [] -> []
  fname +: rest ->
    if pipe (cat fname) (find target) then
      fname ++ searchFiles target rest
    else
      searchFiles target rest
```

3.8.3 Pipe and Copipe Handler

The handlers can now be defined:

```
pipe : ('{Yield b, e} a) -> ('{Await b, e} a) ->{e} a
pipe p c = handle c () with
                (cases
                    { x } -> x
                    { await () -> resume } -> copipe (resume) p)

copipe : (b -> {Await b, e} a) -> ('{Yield b, e} a) ->{e} a
copipe c p = handle p () with
                (cases
                    { x } -> x
                    { yield y -> resume } -> pipe resume '(c y) )
```

Each handler takes two arguments: a producer, `p` and a consumer, `c`. Both arguments are suspended computations that produce a value of type `a`. A producer may invoke `Yield` and a consumer may invoke `Await`.

The `pipe` handler immediately handles the consumer and defines an inline function to handle it with. If the consumer invokes an await it is handled with the `copipe` with the producer and the resumption of the consumer. This means that the consumer process is blocked until the producer can produce its value.

Similarly, the `copipe` handler runs the producer until it yields a value, that value is then given to the suspended consumer and given back to the pipe handler.

3.9 Unix Fork

3.9.1 Process ID's

It would be useful to be able to keep track of multiple processes. UNIX uses process ID's or `pid`'s for this purpose. Whenever a program forks, fork should return a process ID of the newly created process. This then allows programs to 'wait' for a particular process to finish.

3.9.2 Effect Signature

The updated effect signature now includes `wait` which will wait for a process with the specified `pid`. Fork now returns the `pid` of the newly created process. The type of interrupt remains unchanged. Fork and interrupt are renamed to `ufork` and `uinterrupt` to avoid having to overwrite the previous definition.

```
unique ability Co
    where
        ufork: Nat
        wait: Nat -> ()
        uinterrupt: ()
```

3.9.3 Types

Now that the program must also handle `pid`'s there must be more state that is capable of storing this information. `Done` and `Paused` become `Ready` and `Blocked` as now, a process is either ready to run, or blocked by another process. Instead of returning just a return value of `a` it must now also return which process returned that value, hence the `List (Nat, a)` type.

```
unique type Proc a e = Proc (Sstate a e ->{e} List (Nat, a))
unique type Pstate a e = Ready (Proc a e) | Blocked Nat (Proc a e)
unique type Sstate a e = {q: List (Nat, Pstate a e), done: List (Nat
   , a), pid: Nat, pnext: Nat}
```

Finally, there is the `Sstate` type which has the queue of process to be run, the list of process that are finished or done and the current and next process ID's.

Program 5 – Init process

When UNIX is initialised it forks to create a new process to run all programs on. The original process is then the parent process of every process created by the operating system.

```
init: '{e} a ->{e, Co} ()
init main = let pid = ufork
                if pid == 0 then
                    let a = main ()
                        ()
                else
                    wait pid
```

We accept one argument `main` which is the function to be ran as the first program. If we are on the ancestor process ($pid = 0$) then we capture the return value of main while running it and return the unit type. If we are on any other process we wait for the main process.

3.9.4 Running a process

The `runNext` function takes an argument of type `Sstate` and runs it to produce the list of pid's and return values.

```
runNext: Sstate a e ->{e} List (Nat, a)
runNext st =
    let (Sstate q done pid pnext) = st
        match q with
            [] -> done
            (pid', Blocked pid'' resume) +: q' ->
                runNext (Sstate (q' lib.base.data.List.++ [(pid',
    Blocked pid'' resume)]) done pid pnext)
            (pid', Ready resume) +: q' ->
                let st' = (Sstate q' done pid' pnext)
                    Proc (resume') = resume
                    resume' st'
```

It unpacks the `Sstate` and matches on the queue. When it encounters a blocked process, it sends it to the back of the queue and recursively calls the function on the new queue. When it encounters a process that is ready to be run it unpacks the `Proc` type and gives the `Sstate` as an argument to the resumption, thus creating the list.

3.9.5 The handler

```
scheduler: Sstate a e -> Request {Co, e} a ->{e} List (Nat, a)
scheduler st request = match request with
    { result } ->
        let (Sstate q done pid pnext) = st
            done' = done lib.base.data.List.++ [(pid, result)]
            runNext (Sstate q done' pid pnext)
    { ufork -> resume } ->
```

```
        let resume' = (Proc (st -> handle resume 0 with scheduler st
))
            (Sstate q done pid pnext) = st
            pid' = pnext
            pnext' = pnext + 1
            q' = q lib.base.data.List.++ [(pid', Ready resume')]
            handle resume pid' with scheduler (Sstate q' done pid
pnext')
 { wait pid -> resume } ->
        let resume' = (Proc (st -> handle resume () with scheduler
st))
            (Sstate q done pid pnext) = st
            q' = if processExists pid q then
                    q lib.base.data.List.++ [(pid, Blocked pid
resume')]
                 else q lib.base.data.List.++ [(pid, Ready resume')]
            runNext (Sstate q' done pid pnext)
 { uinterrupt -> resume } ->
        let resume' = (Proc (st -> handle resume () with scheduler
st))
            (Sstate q done pid pnext) = st
            q' = q lib.base.data.List.++ [(pid, Ready resume')]
            runNext (Sstate q' done pid pnext)
```

The handler takes a `Sstate` and a request. If the request is a return value it appends the result to the list of done processes and runs the next process. If the request is a fork it creates a new process with the next pid and handles its resumption. The parent process is put back into a `Proc` type and added to the back of the queue with a process ID of zero. Interrupting simply causes the process to be put back into the queue as ready to run while running the next operation. Waiting for a process is more complex. If the process exists in the queue it is blocked and the resumption is handled with the new state. If the process does not exist in the queue it is added to the back of the queue as ready to run. Therefore, if the process ID it has been asked to wait on does not exist it behaves the same as interrupt.

Chapter 4

Extensions

4.1 Error Handling

Currently, whenever this implementation encounters an error or problem it will silently fail. To address this we will introduce different types of error through the `EType` type.

```
unique type EType = PermissionDenied | FileNotFound | FileExists |
    UserExists | UnknownError
```

We provide common errors that might occur in UNIX as well as a catch-all unknown error. Now is also a good time to provide a `toText` implementation for `EType` such that we can print them later. This is simply pattern matching on each possible value of `EType` and returning a sensible string for the error message:

```
toText: EType -> Text
toText = cases
    PermissionDenied -> "Permission denied"
    FileNotFound -> "File not found"
    FileExists -> "File exists"
    UserExists -> "User exists"
    UnknownError -> "Unknown error"
```

Now we can introduce the `Error` signature which provides only one operation `throw`. Throw takes an argument of type `EType` and returns the unit.

```
unique ability Error
    where
        throw: EType -> ()

fail : Request {e, Error} a ->{e, IO, Exception, Status} Empty
fail request =
    match request with
        { throw err -> resume } ->
            printLine (toText err)
            exit 1
        { result } -> exit 0

warn : Request {e, Error} a ->{e, IO, Exception} a
warn request =
```

```
    match request with
        { throw err -> resume } ->
          printLine (toText err)
          handle resume () with warn
        { result } -> result
```

We provide two handlers, `fail` and `warn`. `fail` will print the error message and exit the program with a return code of 1 thus halting execution. `warn` will print the error message and continue execution by handling the resumption.

4.2 Environment Variables

In the implementation outlined in Chapter 3, the environment is solely the user that is currently logged in. No other information is stored or can be stored. In UNIX, *environment variables* are used to store and get information about the current environment from within and outside applications. In the UNIX shell, a user can 'ask' for the value of a shell with the `$` prefix. For example, `echo $USER` will print the username of the currently logged in user. its not just the shell – scripts and programs can also access this information and use it in control flow.

4.2.1 Getting and Setting Environment Variables

In the implementation detailed in Section 3.4 the operation `ask` may only ever return the name of the current user. Instead of `ask` having the type `() -> Text` it now takes an argument of type `Text` which represents the name of the environment variable it should lookup and return. In this way it now acts as a get operation for environment variables.

Now that there is a way to 'get' environment variables it makes sense to introduce a 'set' operation. `setvar` takes the name of an environment variable and another argument of type `Text` representing its new value and updates it in the store.

```
unique ability Session
    where
        su: User -> ()
        ask: Text -> Text
        setvar: Text -> Text -> ()
```

It is now possible to update `whoami` to use this new syntax.

```
whoami: '{Session} Text
whoami _ = ask "USER"
```

4.2.2 Remark on Storing Environment Variables

If this program was written in a more conventional style the arguments to `env` (the handler for `Session`) would have to be modified to accommodate a new argument that is the state for environment variables or a global data structure would have to be introduced. Since this implementation uses effect handlers the `State` handler used in the filesystem can be added to the type signature of `env` as an effect variable meaning the arguments to `env` do not change and no additional data structures need be implemented. This is discussed more in Section 5.2.

4.2.3 Updated Environment Handler

The handler can now be updated and extended to handle the new and updated operations.

```
env: User -> Request {Session} a ->{State [(User, [(Text, Text)])]}
    a
env user request =
    match request with
        {result} -> result

        { ask var -> resume } ->
            match var with
                "USER" ->
                    match user with
                        Alice -> handle resume "alice" with env user
                        Bob -> handle resume "bob" with env user
                        Root -> handle resume "root" with env user
                var ->
                    let st = get ()
                        envs = lookupEnvs user st
                        val = lookupEnvVar var envs
                        handle resume val with env user

        {su user' -> resume} ->
            handle resume () with env user'

        {setvar var val -> resume} ->
            let st = get ()
                envs = lookupEnvs user st
                envs' = modifyEnvVar var val envs
                put (modifyEnvs user envs' st)
                handle resume () with env user
```

The main difference aside from the new operations, is the type signature which now includes `State [(User, [(Text, Text)])]`

The `su` operation did not require any modification. `ask` now uses its first argument to check if the environment variable being requested is `''USER''` or another variable. If it is user it calls resume with the hard-coded `Text` version of the user. In the general case it uses lookup functions that navigate the environment variables data structure and return the correct value for the correct user. This mix of hard coded and user defined environment variables is caused by the hard coded users, and is fixed in the next section, Section 4.3.

Another interesting feature is the case where a program requests the value of an environment variable that is not set. UNIX will return an empty string in this case so `lookupEnvVar` will return an empty string if it does not exist.

4.2.4 Unlisted Functions

The above handler makes use of the functions `lookupEnvVar`, `modifyEnvVar`, `lookupEnvs`, `modifyEnvs` and `userEquals`. The lookup and modify functions traverse and update the `[(User, [(Text, Text)])]` data structure that environment variables are stored in and `userEquals` returns true if the two given users are the same. These functions are listed in the appendix.

4.3 Generic Users

Now that there are user-defined environment variables it makes sense to add user-defined users as well. The `User` type is modified to have a `Username` constructor which allows a user to be constructed with an argument of type `Text`.

```
unique type User = Username Text
```

4.3.1 Effect Operation

The `Session` operation can now be extended to allow privileged users to create new users. `adduser` takes one argument of type `Text` which is the username of the new user.

```
unique ability Session
    where
        su: Text -> ()
        ask: Text -> Text
        export: Text -> Text -> ()
        adduser: Text -> ()
```

4.3.2 Changes to the Handler

The only operations that need to be modified is `su` and `ask`. *Substitute user* now checks if the user exists through the `userExists` function. There is no need to introduce additional state to create new users, the handler simply uses the `[(User, [(Text, Text)])]` data structure to keep track of both the users and their environment variables.

```
{su user' -> resume} ->
    if userExists (Username user') (get ()) then
        handle resume () with env (Username user')
    else
        throw UserDoesntExist
        handle resume () with env user   -- fail
```

To add a new user the handler checks if there is already an instance of that user in the state. If there is it just handles the resumption without modifying the state. If the user does not exist then it adds the user to the state along with an entry in the new users environment variables called `USER` which can be accessed by `ask`.

```
{adduser user' -> resume} ->
            let st = get ()
                    newuser = (Username user')
                    if not (userExists newuser st) then
```

```
                        newvars = [("USER", user')]
                        newenv = modifyEnvs newuser newvars st
                        put newenv
                        handle resume () with env newuser
                    else
                        handle resume () with env user
```

Finally, the match statement can be removed from `ask` as now when a user is created they are created with the `"USER"` environment variable set. The last step is to add an initial userspace which just contains the root user and their environment variable.

```
initialUserspace : [(User, [(Text, Text)])]
initialUserspace = [(Username "root", [("USER", "root")] )]
```

Userspace code can now be run with `handle (handle ... with env (Username "root")) with initialUserspace`.

4.4 Permissions

In Unix file permissions are stored in the I-node of a file. In this implementation, we demonstrate how an effect handler can be used to manage permissions. First, we introduce a new type `Permission` which represents the different types of permissions that can be granted to a user. We also introduce `all` which is a list containing every permission.

```
unique type Permission = Read | Write | AddUser | Grant | Revoke |
    Execute

all : [Permission]
all = [Read, Write, AddUser, Grant, Revoke, Execute]
```

Now we need a way to modify a user's permissions. `grant` and `revoke` are two new operations that take a username and a permission and either add or remove that permission from the user's list of permissions.

```
unique ability Permit
    where
        grant: Text -> Permission -> ()
        revoke: Text -> Permission -> ()
```

Finally we introduce the monolithic handler that we use to implement permissions. Notice the handler handles every effect we have defined thus far although notice from the right hand side of the type signature, that it only handles the `Permit` abilities.

```
permissions: User -> Request {e, Permit, Session, FileRW, FileLU,
    FileCO, Co} a ->{e, Session, FileRW, FileLU, FileCO, Co, Error,
    State [(Text, [Permission])], IO, Exception} a
permissions user request =
    match request with
        -- Permissions
        {grant user' perm -> resume} ->
            checkPermission user Grant !get
            existingPerms = lookupPermission user' !get
```

```
            newPerms = perm +: existingPerms
            put (modifyPermission user' newPerms !get)
            handle resume () with permissions user

        {revoke user' perm -> resume} ->
            checkPermission user Revoke !get
            newPerms = removePermission perm (lookupPermission user'
    !get)
            put (modifyPermission user' newPerms !get)
            handle resume () with permissions user

        -- Users
        {ask var -> resume} ->
            checkPermission user Read !get
            answer = ask var
            handle resume answer with permissions user
        {su user' -> resume} ->
            su user'
            handle resume () with permissions (Username user')
        {adduser user' -> resume} ->
            checkPermission user AddUser !get
            adduser user'
            handle resume () with permissions user
        {export var val -> resume} ->
            checkPermission user Write !get
            export var val
            handle resume () with permissions user

        -- Files
        {read i -> resume} ->
            checkPermission user Read !get
            text = read i
            handle resume text with permissions user

        {write (i, text) -> resume} ->
            checkPermission user Write !get
            write (i, text)
            handle resume () with permissions user

        {link (src, dest) -> resume} ->
            checkPermission user Write !get
            link (src, dest)
            handle resume () with permissions user

        {unlink name -> resume} ->
            checkPermission user Write !get
            unlink name
            handle resume () with permissions user

        {create name -> resume} ->
            checkPermission user Write !get
            ino = create name
            handle resume ino with permissions user

        {open name -> resume} ->
            checkPermission user Read !get
```

```
            ino = open name
            handle resume ino with permissions user

        {ufork -> resume} ->
            checkPermission user Execute !get
            let pid = ufork
                handle resume pid with permissions user

        {wait pid -> resume} ->
            checkPermission user Execute !get
            wait pid
            handle resume () with permissions user

        {uinterrupt -> resume} ->
            checkPermission user Execute !get
            uinterrupt
            handle resume () with permissions user

        {result} -> result
```

The handler works by once again using the `State` effect to store a list of users and their permissions. Whenever the handler encounters an effect it will check the currently logged in users permissions, and if the user has the corerct permissions, it will run the original effect with its original arguments. The handler keeps track of which user is logged in through the user argument to itself. If it encounters a `su` operation it will update this value.

The `grant` and `revoke` operations are implemented by simply traversing and modifying the data structure stored by the `State` effect.

The final step is to add an initial permissions list which contains the root user and all permissions.

```
initialPermissions : [(Text, [Permission])]
initialPermissions = [("root", all)]
```

4.4.1 Remark on Multihandlers Handlers in Unison

In the implementation of the permissions handler, there is a lot of repeated code this is because we are forced to explicitly handle every effect. There is no way to condense the repeat definitions into a pattern match or similar within the match statement. This is discussed further in section 5.1.2.

4.5 An Alternate Scheduler

The scheduler in Section 4 is a simple round-robin scheduler while not dissimilar to UNIX's multilevel feedback queue round robin scheduler it is much more simplistic. Even in widely used systems like Linux which switches between multiple algorithms, scheduling remains very much unsolved [20].

To improve the scheduler we introduce the concept of priority levels through a nice value. In Linux, nice values range from -20 to 19 with -20 being the lowest priority. Each process has a nice value associated with it that the user can manually change to increase or decrease the priority of a process.

4.5.1 Effect Signature

The `Co` effect signature is updated to include `nice` and `renice` operations for getting and setting nice values respectively.

```
unique ability Co
    where
        ufork: Nat
        wait: Nat -> ()
        uinterrupt: ()
        nice: Nat -> Int
        renice: Nat -> Int -> ()
```

To avoid breaking the original `scheduler` these effects are handled but simply handle the resumption and perform no computation.

4.5.2 Priority Queue

The next step is to create a new `runNext` function that takes into account the nice value of each process.

```
runNextNice: Sstate a e -> [(Nat, Int)] ->{e} List (Nat, a)
runNextNice st niceValues =
    let (Sstate q done pid pnext) = st
        match q with
            [] -> done
            (pid', Blocked pid'' resume) +: q' ->
                runNextNice (Sstate (q' lib.base.data.List.++ [(pid
   ', Blocked pid'' resume)]) done pid pnext) niceValues
            (pid', Ready resume) +: q' ->
                match lowestNiceInQueue niceValues q with
                    Left (pid', Ready resume) ->
                        let st' = (Sstate q' done pid' pnext)
                            Proc (resume') = resume
                            resume' st'
                    Left (pid', Blocked pid'' resume) ->
    -- unreachable
                        let st' = (Sstate q' done pid' pnext)
                            Proc (resume') = resume
                            resume' st'
                    Right () ->
                        let st' = (Sstate q' done pid' pnext)
                            Proc (resume') = resume
                            resume' st'
```

The function `lowestNiceInQueue` takes a list of pairs of procces ID's and nice values and the `Sstate` it is the same as `runNext` apart from the ready branch. Instead it checks if the is a process that is ready to be run with a lower nice value than the one at the front of the list. If there is it runs that instead. The Blocked branch is unreachable as `lowestNiceInQueue` will always return a ready process but it must be put there to satisfy the typechecker.

4.5.3 An Aging Scheduler

An aging scheduler is a type of scheduler that increases the priority of a process the longer it has been waiting. This is done to prevent starvation of low priority processes that otherwise would not be run. This is achieved by increasing the priority of a process every time it forks and setting its child to have its old nice value.

The scheduler is implemented through a modified version of the `scheduler` handler. As such only the operations that have changed or been added are listed, the others just have `runNext` swapped for `runNextNice`.

```
schedAging: Sstate a e -> Request {Co, e} a ->{e, State [(Nat, Int)
    ]} List (Nat, a)
schedAging st request = match request with
     ....
     { ufork -> resume } ->
          let resume' = (Proc (st -> handle resume 0 with scheduler st
    ))
               (Sstate q done pid pnext) = st
               nicevalue = lookupNice pid !get

               if nicevalue - +1 <= minNice then
                    let q' = q lib.base.data.List.++ [(pid, Ready resume
    ')]
                         pid' = pnext
                         pnext' = pnext + 1
                         handle resume pid' with scheduler (Sstate q'
    done pid pnext')
               else
                    put (modifyNice pnext nicevalue !get)
                    put (modifyNice pnext (nicevalue - +1) !get)

                    pid' = pnext
                    pnext' = pnext + 1

                    q' = q lib.base.data.List.++ [(pid', Ready resume')]
                    handle resume pid' with schedAging (Sstate q' done
    pid pnext)

     { nice pid -> resume } ->
          let (Sstate q done pid pnext) = st
               nicevalue = lookupNice pid !get
               handle resume nicevalue with schedAging st

     { renice pid newNice -> resume} ->
          let resume' = (Proc (st -> handle resume () with scheduler
    st))
               (Sstate q done pid pnext) = st
               put (modifyNice pid newNice !get)
               runNextNice (Sstate q done pid pnext) !get
```

```
    ...
```

The interesting case here is fork where the nice value of the parent is decreased by one and the child is set to the old nice value of the parent. This will cause process that have been waiting for a long time to have a higher and hopefully be run sooner.

4.5.4 Simple Starvation Heuristic

Yet another improvement to the scheduler is to introduce a simple heuristic to prevent starvation. The heuristic is that if a process reached minimum nice through a fork then it has probable been interrupt a lot and as such is unlikely to run even if we adjust the nice. To save on the extra computation of handling nice values we can switch back to the round robin scheduler just by handling the same state with the old handler.

4.5.5 Remark on Switching Handlers

Although the two schedulers are not too different they still handle the queue differently. We can define as many schedulers as we like (as long as they operate on the same types) and seamless switch between them by handling resumptions differently. This is discussed more in Section 5.2.

Chapter 5

Evaluation and Discussion

5.1 Unison

The following section details some of the quirks and missing features of the Unison effect systsem that were encountered during the implementation of UNIX.

5.1.1 Typechecker

As mentioned in Section 3.5.1, the Unison typechecker could not infer that `resume true/false` was going to return a type of `[a]` despite their being a type signature to this effect. This forces the programmer to use the fully qualified `lib.base.data.List.++` instead of just ++. This hurts the readability of the code and makes it more irksome to program with as you don't know which types it can infer and which it can't. This seems like an issue that shouldn't arise in a modern functional language.

5.1.2 Multihandler Pattern Matching

If we consider a handler, like the one in Section 4.4 for permissions, that handles multiple effects, it would be useful to be able to pattern match on the effect signature or across multiple operations and invoke them once.

For example, in the permissions handler there is lots of repeated code of the general form:

```
{ [effect] [args] -> resume } ->
  checkPermission user perm !get
  [effect] [args]
  handle resume () with permissions user
```

If we were allows to have a syntax like {`effect1 | effect2 | effect3`} `[args] -> resume` then we could condense the repeated code into a single branch of the match statement. Even if we could only perform this match for operations of the same effect signature it would still massively reduce code reuse.

```
permissions: User -> Request {e, Permit, Session, FileRW, FileLU,
    FileCO, Co} a ->{e, Session, FileRW, FileLU, FileCO, Co, Error,
    State [(Text, [Permission])], IO, Exception} a
  {grant user' perm -> resume} -> ...

  {revoke user' perm -> resume} -> ...

  { read i -> resume } | { write (i, text) -> resume } | { link (src
    , dest) -> resume } | { unlink name -> resume } | { create name
    -> resume } | { open name -> resume } | ... ->
    checkPermission user Read !get
    {ability args}
    handle resume () with permissions user
```

You could even borrow the `!` syntax and have `!ability` to represent running an ability with its given arguments. With this theoretical syntax the size and complexity of the handler is massively reduced.

5.1.3 Effect Variables in Definitions

In the Unison documentation [21] they suggest defining abilities with the effct signature as part of the type signature of the effect to represent needing access to the ability to be run. For example

```
unique ability Await a
  where
    await : () ->{Await a} a
    yield : a ->{Await a} ()
```

However, the effect signature {`Await a`} is inferred by the fact `await` and `yield` are operations of the `Await` ability. You might think it is useful to be able to include other effects but if we were to do `await : () ->`{`Await a, Yield a`} `a` then the type checker stops us with the error `EffectConstructorHadMultipleEffects: Await a3420, Yield a3420 a3420`. Furthermore, if you mistakenly put the signature `await () ->`{`Await`} `a` i.e. you missed the type of a from the `Await` effect variable, then it will attempt to infer the type of `Await` and will no longer throw an error. This is fine until you try to use the effect in a function when you are met with abstract errors like:

```
  The expression in red

                needs the abilities: {Yield b3407}
    but was assumed to only require: {Yield a3434, e3439}

      This is likely a result of using an un-annotated function as
    an argument with concrete abilities. Try adding an annotation
    to the function definition whose body is red.

    511 |                     { await () -> resume } -> copipe
    (resume) p)
```

This error points towards the `copipe` function as the source of the error which in this case is completely fine.

I suggest to anyone using Unison to avoid including effect variables in the definition of an ability as it can lead to confusing and abstract errors. Furthermore, I would encourage the Unison team to consider removing this feature as it seems to cause more problems than it solves. If it is always inferred to be the correct type then there is no need to allow a user to set it to an incorrect type.

5.2 Effect Oriented vs Conventional Programming

Consider the following hypothetical implementation of the environment operations from Section 3.4, which has been implemented in a more conventional style. .

```
adduser': [User] -> Text -> [User]
adduser' store user =
    store :+ (Username user)

su' : [User] -> User -> [User]
su' store user =
    match store with
        [] -> store
        (u +: rest) ->
            let uname = userToText u
                username = userToText user
                if uname == username then
                    u +: rest
                else
                    store

ask': [User] -> Text
ask' store =
    match store with
        [] -> ""
        (u +: rest) ->
            let uname = userToText u
                uname
```

In this version, all the state for the users is stored in a list of users where the first one in the list is the currently logged in user. This is already more irksome than the handler implementation as the user of the functions has to keep track of the state themselves through the return values of `su'` and `adduser'`.

If we compare this to the handler version the state is set once at the beginning by setting which user is logged in initially and then once a block of code is being handled the user can ignore the state as it is hidden from them.

5.2.1 Modularity

The comparison in Figure 5.1 shows that although both versions require initial state, the power of effect handlers allow us to keep the implementation separate from the use and as such there is no requirement for the function to manage the state of users. As long as the program is run within scope of the handler all operations will have access to

```
createAndSwitch: Text -> ()
createAndSwitch user =
  adduser user
  su user


> handle createAndSwitch
    ''alice'' with (Username
    ''root'')
```

```
createAndSwitch': Text ->
  [User] -> [User]
createAndSwitch' user state =
  let newState = adduser'
  state user
     su' newState user

> createAndSwitch' ''alice''
  [(Username ''root'')]
```

Figure 5.1: Effect oriented version (*left*) and the standard version (*right*)

this state. The result of this is that the effect operations behave much more like UNIX commands as they can be called without having to pass around variables.

Additionally, if we wanted to provide an alternate implementation of `adduser` that logs in the newly created user we could easily accomplish this by writing an alternate handler for `adduser`. The conventional version would require a whole new function to be written just to change the line `store :+ (Username user)` to `(Username user) +: store`. Then once that new function was written, we would have to manually change every occurence of `adduser'` where we want to use the new semantics. The effect oriented version keeps the same operation and we simply handle any functions where we need the new version with the new handler. A much more seamless style of coding.

5.2.2 Composition

When we added environment variables to the userspace it was as simple as adding the `State` effect signature with the type being the data structure we need. This means in the scope of the `env` function where this was added we could immediately start using `put` and `get` to manage the state of users and environment variables. We didn't even need to define a new handler the `State` handler can accept any algebraic data type as the type to put and get meaning it was as simple as adding another `handle` statement and then modifying the operations.

In contrast, to add environment variables to the conventional code we must modify the data type used to store users to be something like `unique type Environment = {usr: User, envs: [(Text, Text)]}`. Then all occurrences of the operations must be modified to give them values of the new type. The effect oriented one simply adds a handle statement and once again leaves the type signatures of the operations unchanged.

Another example of composing handlers is in the scheduler outlined in Section 4.5.3. We could develop different scheduling algorithms that are completely isolated from one and other but still easily switch between them just by handling their resumptions with a different algorithm. You could use heuristics to determine which is the best scheduler to use at that moment. To implement this in a conventional way it would require at the very least some shared queue structure (or lots of copying data) and lots of boilerplate to switch between them. In the effect-oriented version we can simply `handle ... with otherAlgorithm`.

5.2.3 Performance

While performance was not the focus of this project, I was still interested in observing the tradeoffs between the two implementations provided in Section 5.2. Using the Unison Timers library [22] I ran code that created and switched to a random user 1000 times. The results are shown in Table 5.1.

Metric	Effectful	Conventional
Samples	1	1
Total (realtime)	2.021658s	15µs
Mean (realtime)	2.021658s	15µs
Median (realtime)	2.021658s	15µs
Min (realtime)	2.021658s	15µs
Max (realtime)	2.021658s	15µs
Total (cpu)	2.152157s	26µs
Mean (cpu)	2.152157s	26µs
Median (cpu)	2.152157s	26µs
Min (cpu)	2.152157s	26µs
Max (cpu)	2.152157s	26µs

Table 5.1: Benchark results for creating and switching to 1000 randomly generated users

The poor performance of the effect oriented version is not surprising. In most effect implementations layering effect handlers leads to poor performance although there is work focused on improving the performance in these cases [23] [24] [25].

Chapter 6

Conclusion and Future Work

6.1 Base Implementation

6.1.1 Summary

The section demonstrates a Unison implmentation of Hillerström's UNIX and provides several programs that give examples of composing handlers to implement more of UNIX. I provide Unison implementations for status, basic I/O, users, a basic serial filesystem, pipes and two methods of timesharing. The programs build on top of these handlers to create additional UNIX programs like `cat`.

6.1.2 Future Work

Shell A logical extension to this project would be creating a shell-like `bash` or similar to allow users to run the commands in a more real-time way. Writing a shell would be mostly implementing the parser which is usually unrelated to effect-oriented programming which is why it was omitted from this project. It may also be interesting to see if parsers can be written in an effect oriented style and if there is any benefit to this approach.

Implementation in other languages Another interesting project would be to implement the same Unix in another language that has effect handlers like Frank or Koka and compare the two implementations. This approach would allow for a more direct comparison of the two languages and their effect systems.

6.2 Extended Implementation

6.2.1 Summary

This section introduces improvements to the userspace through generic user's and environment variables. Both of these features leverage the `State` effect and demonstrate how effect handlers can be composed to create new features. We also introduce a scheduler that demonstrates how different handlers can be used to achieve more advanced control flow. This creates the final version of UNIX which can now be used as a test platform for analysis.

6.2.2 Future Work

Grep Implementing a version of grep would be a good way to provide some more advanced examples of effect handlers. This implementation would involve parsing regular expressions from the pipe command and then matching based on the regular expression.

Linux Another interesting but extremly difficult project would be attaching effect handlers to Linux. This could be done through a kernel module and then used for other applications like writing another scheduling algorithm or making the filesystem closer to the one outlined by Kiselyov [17]. The scope of the project would have to be reduced due to the complexity and size of Linux. It would also be a good opportunity to explore performant effects.

6.3 Evaluation

6.3.1 Summary

This section describes some desirable features the Unison team could consider adding and the rationale behind adding them. It also outlines and addresses some of the quirks that were encountered and referenced during the implementation. It also addresses effect-oriented programming as a concept by comparing the Unix implementation to a more conventional implementation. This demonstrates the effect-oriented versions superior modularity and the unique ability to compose effects to create new features. The performance of the two is also shown, despite this not being an aim of the project, it is always interesting to compare.

6.3.2 Future Work

Performance As was mentioned before, there is lots of research concerned with improving the performance of effect handlers. A useful extension would be attempting to apply these techniques to Unison to see if the performance could be improved.

Comparison with a more traditional implementation Although this was partly covered in the evaluation section it would be interesting to see how the effect oriented version of Unix compares to a more traditional version in a more complete and thorough way. This would involve implementing the same features without the use of effect handlers and comparing them.

Appendix A

Final State of the Code

A.1 Base Implementation

```
unique ability BasicIO
    where
        echo: Text -> ()

basicIO : Request {BasicIO} a ->{IO, Exception} a
basicIO result =
    match result with
        { echo text -> resume } -> putText stdOut text; handle
    resume () with basicIO
        { result } -> result


{-
    Status
    =========
-}

unique type Empty =

-- The unix exit command that allows you to exit with error code
unique ability Status
       where
            exit: Nat -> Empty

-- handles the exit command which just returns an integer
exitHandler : Request {g, Status} a -> Nat
exitHandler request =
    match request with
          { result } -> 0
          { exit v -> resume } -> v

{-
Userspace
==========
```

```
This handles the hard coded users and their environments.
It allows for whoami and su commands to be run.

-}

-- The users (hard coded)
unique type User = Alice | Bob | Root
structural type Environment = Environment User

-- Each user has a unique environment
environments : List (User, Environment)
environments = [(Alice, Environment Alice),
                (Bob, Environment Bob),
                (Root, Environment Root)]

unique ability Session
    where
        su: User -> {Session } Environment

-- Helper function because unison cannot infer equity on custom
   types
userEquals: User -> User -> Boolean
userEquals user1 user2 =
    match user1 with
        Alice -> match user2 with
            Alice -> true
            _ -> false
        Bob -> match user2 with
            Bob -> true
            _ -> false
        Root -> match user2 with
            Root -> true
            _ -> false

whoami: '{Session} Text
whoami _ = ask ()

env: User -> Request {Session} a -> a env user request =
  match request with
    {result} -> result
    { ask () -> resume } -> match user with
      Alice -> handle resume "alice" with env user
      Bob -> handle resume "bob" with env user
      Root -> handle resume "root" with env user

{-
      Time Sharing
   ==================
-}

unique ability Interrupt
    where
        interrupt: {Interrupt } ()


-- unique type PState a = Done a | Paused (Unit ->{Interrupt} a)
```

```
unique type PState a e = Done a | Paused ('{e} PState a e)

interruptWrite result =
    match result with
        { echo text -> resume } ->
            interrupt
            echo text
            handle resume () with interruptWrite
        { result } -> ()

--reifyProcess : Request {Interrupt} a -> PState a e
--reifyProcess request =
--    match request with
--        { interrupt -> resume } -> ( handle resume with
   reifyProcess )
--        { result } -> Done result


reifyProcess request =
    match request with
        { interrupt -> resume } -> Paused (_ -> handle !resume with
   reifyProcess )
        { result } -> Done result

unique ability TimeSharing
    where
        fork: {TimeSharing } Boolean

-- handler for time sharing ability
nondet : Request {TimeSharing} a -> [a]
nondet request =
    match request with
        { fork -> resume } -> (handle resume true with nondet) lib.
   base.data.List.++ (handle resume false with nondet)
        { result } -> [result]

sched : [PState a {e, TimeSharing}] -> [a] ->{e} [a]
sched ps done =
    match ps with
        [] -> done
        (Done res) +: ps' -> sched ps' (res lib.base.data.List.+:
   done)
        (Paused m) +: ps' -> sched (ps' lib.base.data.List.++ (
   handle !m with nondet)) done

--schedule : [PState a {e, TimeSharing}] ->{e} [a]
--schedule processes =
--    sched processes []

timeshare : '{g, Interrupt, TimeSharing} o ->{g} [o]
timeshare m = sched [Paused (_ -> handle !m with reifyProcess)] []

{-
  Serial File System
  ==================
-}
```

```
unique ability State a
       where
            put: a -> ()
            get: () -> a

--runState : (b, Request {State b} a) -> (b, a)
--runState pair =
--    match pair with
--        (s, request) ->
--        match request with
--            { result } -> (s, result)
--            { put s' -> resume } -> (s', resume ())
--            { get () -> resume } -> (s, resume s)

runState : a -> Request {State a} b -> b
runState v request =
    match request with
        { put v' -> resume } -> handle resume () with runState v'
        { get () -> resume } -> handle resume v with runState v
        { result } -> result

unique type DirectoryT = Directory (Text, Nat)
unique type DataRegionT = DataRegion (Nat, Text)
unique type INodeT = INode Nat Nat
unique type IListT = IList (Nat, INodeT)
unique type FileSystemT = FileSystem (List DirectoryT) (List IListT)
     (List DataRegionT) Nat Nat


initialINode : INodeT
initialINode = INode 0 0

initialDirectory : DirectoryT
initialDirectory = (Directory ("stdout", 0))

initialDataRegion : DataRegionT
initialDataRegion = DataRegion (0, "")

initialIList : IListT
initialIList = IList (0, initialINode)

initialFileSystem : FileSystemT
initialFileSystem = FileSystem [initialDirectory] [initialIList] [
   initialDataRegion] 0 0

lookupINode : Nat -> [IListT] -> Either INodeT ()
lookupINode i ilists =
    match ilists with
        [] -> Right ()
        (IList (i', inode)) +: rest ->
            if i == i' then Left inode
            else lookupINode i rest

lookupFName : Text -> [DirectoryT] -> Either Nat ()
lookupFName name directories =
```

```
    match directories with
        [] -> Right ()
        (Directory (name', i)) +: rest ->
            if name == name' then Left i
            else lookupFName name rest

modifyINode : Nat -> INodeT -> [IListT] -> [IListT]
modifyINode i inode ilists =
    match ilists with
        [] -> []
        (IList (i', inode')) +: rest ->
            if i == i' then (IList (i, inode)) +: rest
            else (IList (i', inode')) +: modifyINode i inode rest

lookupDataRegion : Nat -> [DataRegionT] -> Either Text ()
lookupDataRegion i dataRegions =
    match dataRegions with
        [] -> Right ()
        (DataRegion (i', text)) +: rest ->
            if i == i' then Left text
            else lookupDataRegion i rest

modifyDataRegion : Nat -> Text -> [DataRegionT] -> [DataRegionT]
modifyDataRegion i text dataRegions =
    match dataRegions with
        [] -> []
        (DataRegion (i', text')) +: rest ->
            if i == i' then (DataRegion (i, (text' ++ text))) +:
   rest
            else (DataRegion (i', text')) +: modifyDataRegion i text
    rest

-- fread, implementation of system read
fread : Nat -> FileSystemT -> Either Text ()
fread i fs =
    match fs with
        FileSystem directories ilists dataRegions _ _ ->
            match lookupINode i ilists with
                Left inode ->
                    match inode with
                        INode _ dataRegion ->
                            match lookupDataRegion dataRegion
   dataRegions with
                                Left text -> Left text
                                Right () -> Right ()
                Right () -> Right ()

-- fwrite, writes to the file system at the given inode with the
   given text
fwrite : Nat -> Text -> FileSystemT -> FileSystemT
fwrite i text fs =
    match fs with
        FileSystem directories ilists dataRegions _ _ ->
            match lookupINode i ilists with
                Left inode ->
                    match inode with
```

```
                         INode _ dataRegion ->
                             FileSystem directories (modifyINode i (
   INode i dataRegion) ilists) (modifyDataRegion dataRegion text
   dataRegions) 0 0
                  Right () -> fs

unique ability FileRW
         where
                  read: Nat -> {FileRW } Text
                  write: (Nat, Text) -> {FileRW } ()

fileRW : Request {FileRW} a ->{State FileSystemT} a
fileRW result =
    match result with
        { read i -> resume } ->
            let fs = get ()
                text = fread i fs
                match text with
                    Left text -> handle resume text with fileRW
                    Right () -> handle resume "" with fileRW    --
   make this fail
        { write (i, text) -> resume } ->
            let fs = get ()
                fs' = fwrite i text fs
                put fs'
                handle resume () with fileRW
        { result } -> result

echoWrite : Text ->{FileRW} ()
echoWrite text = write (0, text)

systemIO : Request {BasicIO} a ->{FileRW, State FileSystemT} a
systemIO result =
    match result with
        { echo text -> resume } ->
            handle write (0, text) with fileRW
            handle resume () with systemIO
        { result } -> result

fopen : Text -> FileSystemT -> Nat
fopen name fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
            match lookupFName name directories with
                Left i -> i
                Right () -> inext

has : Text -> [DirectoryT] -> Boolean
has name directories =
    match directories with
        [] -> false
        (Directory (name', i)) +: rest ->
            if name == name' then true
            else has name rest

fcreate : Text -> FileSystemT -> (Nat, FileSystemT)
```

```
fcreate name fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
            -- file already exists, overwrite it
            if has name directories then
                let ino = (fopen name fs)
                    inode = lookupINode ino ilists
                    match inode with
                        Left inode ->
                            match inode with
                                INode ino loc ->
                                    let dreg' = modifyDataRegion loc
    "" dataRegions
                                        (ino , FileSystem
   directories ilists dreg' dnext inext)
                        Right () -> (ino, fs) -- unreacable
            else
                let inext' = inext + 1
                    dnext' = dnext + 1
                    inode = INode inext dnext
                    ilists' = (IList (inext, inode)) +: ilists
                    directories' = (Directory (name, inext)) +:
   directories
                    (inext, FileSystem directories' ilists'
   dataRegions dnext' inext')

unique ability FileCO
    where
        open: Text -> {FileCO } Nat
        close: Nat -> {FileCO } ()

fileCO : Request {FileCO} a ->{FileRW, State FileSystemT} a
fileCO result =
    match result with
        { open name -> resume } ->
            let fs = get ()
                (ino, fs') = fcreate name fs
                put fs'
                handle resume ino with fileCO
        { close i -> resume } ->
            let fs = get ()
                put fs
                handle resume () with fileCO
        { result } -> result

flink: Text -> Text -> FileSystemT -> FileSystemT
flink src dest fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
            if has dest directories then
               fs -- error, file exists
            else
                let ino = lookupFName src directories
                    match ino with
                        Left ino ->
                            let directories' = (Directory (dest, ino
```

```
   )) +: directories
                                inode = lookupINode ino ilists
                                match inode with
                                    Left inode ->
                                        match inode with
                                            INode ino loc ->
                                                  let loc' = loc + 1
                                                      inode' = INode
    ino loc'
                                                      ilists' =
   modifyINode ino inode' ilists
                                                      FileSystem
   directories' ilists' dataRegions dnext inext
                                    Right () -> fs -- unreachable,
   we know the file exists
                        Right () -> fs -- no such file src

removeINode : Nat -> [IListT] -> [IListT]
removeINode i ilists =
    match ilists with
        [] -> []
        (IList (i', inode)) +: rest ->
            if i == i' then rest
            else (IList (i', inode)) +: removeINode i rest

removeDataRegion : Nat -> [DataRegionT] -> [DataRegionT]
removeDataRegion i dataRegions =
    match dataRegions with
        [] -> []
        (DataRegion (i', text)) +: rest ->
            if i == i' then rest
            else (DataRegion (i', text)) +: removeDataRegion i rest

removeDirectory : Text -> [DirectoryT] -> [DirectoryT]
removeDirectory name directories =
    match directories with
        [] -> []
        (Directory (name', i)) +: rest ->
            if name == name' then rest
            else (Directory (name', i)) +: removeDirectory name rest

funlink: Text -> FileSystemT -> FileSystemT
funlink name fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
            if has name directories then
                let ino = lookupFName name directories
                    match ino with
                        Left ino ->
                             let directories' = removeDirectory name
    directories
                                 inode = lookupINode ino ilists
                                 match inode with
                                        Left inode ->
                                            match inode with
                                                INode ino loc ->
```

```
                                                            if loc > 1 then
                                                                let loc' =
   loc - 1
                                                                    inode' =
    INode ino loc'
                                                                    ilists'
   = modifyINode ino inode' ilists

   FileSystem directories' ilists' dataRegions dnext inext
                                                            else
                                                                let ilists'
   = removeINode ino ilists

   dataRegions' = removeDataRegion loc dataRegions

   FileSystem directories' ilists' dataRegions' dnext inext
                                                  Right () -> fs --
   unreachable, we know the file exists
                                    Right () -> fs -- no such file src
            else
                fs -- no such file


unique ability FileLU
       where
            link: (Text, Text) -> {FileLU } ()
            unlink: Text -> {FileLU } ()

fileLU : Request {FileLU} a ->{FileRW, State FileSystemT} a
fileLU result =
    match result with
        { link (src, dest) -> resume } ->
            let fs = get ()
                fs' = flink src dest fs
                put fs'
                handle resume () with fileLU
        { unlink name -> resume } ->
            let fs = get ()
                fs' = funlink name fs
                put fs'
                handle resume () with fileLU
        { result } -> result

fileIO m = handle (handle (handle !m with fileRW) with fileCO) with
   fileLU


{-
    pipes
    ======
-}

unique ability Await a
    where
        await: () -> a
```

```
unique ability Yeild a
    where
        yeild: a -> ()

pipe : '{Yeild b, e} a -> '{Await b, e} a ->{e} a
pipe p c = handle c () with
                (cases
                    { x } -> x
                    { await () -> resume } -> copipe resume c p)

copipe : b -> {Await b, e} a -> '{Yeild b, e} a ->{e} a
copipe c p = handle p () with
                (cases
                    { x } -> x
                    { yeild y -> resume } -> pipe ('resume) c y )

{-
    Process Syncronization
    ======================
-}

unique ability Co
    where
        ufork: () -> {Co } Nat
        wait: Nat -> {Co } ()
        uinterrupt: () -> {Co } ()

unique type ProcessState a = Ready a | Blocked Nat a

ready: [(Nat, a)] -> [(Nat, ProcessState a)]
ready processes =
    match processes with
        [] -> []
        (pid, process) +: rest -> (pid, Ready process) +: ready rest

blocked: [(Nat, a)] -> [(Nat, ProcessState a)]
blocked processes =
    match processes with
        [] -> []
        (pid, process) +: rest -> (pid, Blocked pid process) +:
    blocked rest

processExists: Nat -> [(Nat, ProcessState a)] -> Boolean
processExists pid processes =
    match processes with
        [] -> false
        (pid', process) +: rest ->
            if pid == pid' then true
            else processExists pid rest

runNext : List (Nat, ProcessState a) -> List (Nat, a) -> Nat -> Nat
    ->{e} List (Nat, a)
runNext queue done pid pnext =
    match queue with
        []          -> done
        fst +: rest -> match fst with
```

```
            (pid’, Ready resume)         -> handle resume with
   scheduler rest done pid’ pnext
            (pid’, Blocked pid’’ resume) -> runNext (rest lib.base.
   data.List.:+ (pid, Blocked pid’ resume)) done pid pnext

scheduler : List (Nat, ProcessState a) -> List (Nat, a) -> Nat ->
   Nat -> Request {Co} a -> List (Nat, a)
scheduler queue done pid pnext proc =
    match proc with
        { ufork () -> resume } ->
            let resume’ = handle resume 0 with scheduler queue done
   pid pnext
                pid’ = pnext
                pnext’ = pnext + 1
                queue’ = queue lib.base.data.List.++ ready resume’
                handle resume pid with scheduler queue’ done pid’
   pnext’
        { wait pid’ -> resume } ->
            let resume’ = handle resume () with scheduler queue done
    pid pnext
                queue’ = if processExists pid’ queue then
                             queue lib.base.data.List.++ blocked
   resume’
                         else queue ++ ready resume’
                runNext queue’ done pid pnext
        { uinterrupt () -> resume } ->
            let resume’ = handle resume () with scheduler queue done
    pid pnext
                queue’ = queue ++ ready resume’
                runNext queue’ done pid pnext
        { result } -> runNext queue (done lib.base.data.List.++ [(
   pid, result)]) pid pnext

timeshare2 : ’{Co} a ->{Co} [(Nat, a)]
timeshare2 m = handle m () with scheduler [] [] 1 2

init : ’{e} () ->{e, Co} ()
init main = let pid = ufork ()
                if pid == 0 then
                    main ()
                else
                    wait pid

 {-
    Util
    ======
-}

--unique ability Logging
--    where
--        log: a -> {Logging } ()
--
--logHandler : Request {Logging} a -> a
--logHandler request =
--    match request with
--        { log x -> resume } -> putText stdOut x; handle resume ()
```

```
   with logHandler
--        { result } -> result


{-
    Examples
    =========
-}

ioAndUsers : a ->{Session, Status, BasicIO} ()
ioAndUsers _ =
    if whoami == "root" then
        echo "Logged in as root\n";
        exit 0
    else
        echo "Permission denied\n";
        exit 1

runIOandUsers _ = handle (handle (handle !ioAndUsers with
   sessionManager initialEnv) with exitHandler) with basicIO

ritchie _ = echo "UNIX is basically\n"; echo "a simple operating
   system\n"; echo "but you have to be a genius to understand the
   simplicity\n"
hamlet _ = echo "To be, or not to be,\n"; echo "that is the question
   :\n"; echo "Wether 'tis nobler in the mind to suffer\n";

forkAndIO : a ->{BasicIO, TimeSharing} ()
forkAndIO _ =
    if fork then
        !ritchie
    else
        !hamlet

runForkAndIO _ = handle (handle !forkAndIO with basicIO) with nondet

{-
    Tests
    ======
-}

-- Test exiting
testProgram0 _ = exit 42
--> handle !testProgram0 with exitHandler

testProgram1 _ =
    whoami
--> handle !testProgram1 with whoamiHandler

testProgram2 _ =
    handle whoami with sessionManager (handle su Alice with
   sessionManager initialEnv)
--> handle !testProgram2 with sessionManager initialEnv

proc1 _ = handle [echo "Hello, ", echo "World!"] with basicIO
proc2 _ = handle [echo "Goodbye, ", echo "Code!"] with basicIO
```

```
testProgram3 _ =
    handle whoami with sessionManager (handle su Bob with
   sessionManager initialEnv)

testProgram4 _ =
    if fork then
        [handle whoami with sessionManager (handle su Bob with
   sessionManager initialEnv)]
     else
        [handle whoami with sessionManager (handle su Alice with
   sessionManager initialEnv)]
--> handle !testProgram4 with nondet

--ritchie _ = echo "UNIX is basically\n"; echo "a simple operating
   system\n"; echo "but you have to be a genius to understand the
   simplicity\n"
--hamlet _ = echo "To be, or not to be,\n"; echo "that is the
   question:\n"; echo "Wether 'tis nobler in the mind to suffer\n";

testProgram5 _ =
    handle (handle (if fork then [!ritchie] else [!hamlet]) with
   basicIO) with nondet

testProgram6 _ = timeshare (_ -> (handle (handle (if fork then [!
   ritchie] else [!hamlet]) with interruptWrite) with basicIO))

testProgram7 _ = handle (get (handle (handle (echo "Hello, World!\n"
   ) with systemIO) with fileRW)) with runState initialFileSystem

-- testProgram8 _ = timeshare2 (_ -> (handle (handle (if ufork () ==
    0 then [!ritchie] else [!hamlet]) with interruptWrite) with
   basicIO))
```

A.2 Extended Implementation

The final version of the code after all the extensions outlined in chapter 4 were implemented.

```
{-
    BasicIO
    =========
-}

unique ability BasicIO
    where
        echo: Text -> ()

basicIO : Request {BasicIO} a ->{IO, Exception} a
basicIO result =
    match result with
        { echo text -> resume } -> putText stdOut text; handle
   resume () with basicIO
        { result } -> result
```

```
{-
    Status
    =========
-}

unique type Empty =

-- The unix exit command that allows you to exit with error code
unique ability Status
       where
            exit: Nat -> Empty

-- handles the exit command which just returns an integer
exitHandler : Request {g, Status} a -> Nat
exitHandler request =
    match request with
          { result } -> 0
          { exit v -> resume } -> v

{-
Userspace
==========

This handles the hard coded users and their environments.
It allows for whoami and su commands to be run.

-}

-- The users (hard coded)
unique type User = Username Text

unique ability Session
    where
        su: Text -> ()
        ask: Text -> Text
        setvar: Text -> Text -> ()
        adduser: Text -> ()

whoami: '{Session} Text
whoami _ = ask "USER"

env: User -> Request {Session} a ->{Error, State [(User, [(Text,
   Text)])]} a
env user request =
    match request with
        {result} -> result

        { ask var -> resume } ->
           let st = get ()
               envs = lookupEnvs user st
               val = lookupEnvVar var envs
               handle resume val with env user

        {su user' -> resume} ->
```

```
            if userExists (Username user') (get ()) then
                handle resume () with env (Username user')
            else
                throw NoSuchUser
                handle resume () with env user    -- fail

        {setvar var val -> resume} ->
            let st = get ()
                envs = lookupEnvs user st
                envs' = modifyEnvVar var val envs
                put (modifyEnvs user envs' st)
                handle resume () with env user

        {adduser user' -> resume} ->
            let st = get ()
                newuser = (Username user')
                newvars = [("USER", user')]
                newenv = modifyEnvs newuser newvars st
                if not (userExists newuser st) then
                    put newenv
                    handle resume () with env newuser
                else
                    throw UserExists
                    handle resume () with env user

lookupEnvVar: Text -> [(Text, Text)] -> Text
lookupEnvVar var env =
    match env with
        [] -> ""
        (var', val) +: rest ->
            if var == var' then val
            else lookupEnvVar var rest

modifyEnvVar: Text -> Text -> [(Text, Text)] -> [(Text, Text)]
modifyEnvVar var val env =
    match env with
        [] -> [(var, val)]
        (var', val') +: rest ->
            if var == var' then (var, val) +: rest
            else (var', val') +: modifyEnvVar var val rest

lookupEnvs: User -> [(User, [(Text, Text)])] -> [(Text, Text)]
lookupEnvs user envs =
    match envs with
        [] -> []
        (user', env) +: rest ->
            if userToText user == userToText user' then env
            else lookupEnvs user rest

modifyEnvs: User -> [(Text, Text)] -> [(User, [(Text, Text)])] -> [(
   User, [(Text, Text)])]
modifyEnvs user env envs =
    match envs with
        [] -> [(user, env)]
        (user', env') +: rest ->
```

```
            if userToText user == userToText user' then (user, env)
   +: rest
            else (user', env') +: modifyEnvs user env rest

userExists: User -> [(User, [(Text, Text)])] -> Boolean
userExists user envs =
    match envs with
        [] -> false
        (user', env) +: rest ->
            if userToText user == userToText user' then true
            else userExists user rest

userToText: User -> Text
userToText user =
    let (Username username) = user
        username

initialUserspace : [(User, [(Text, Text)])]
initialUserspace = [(Username "root", [("USER", "root")] )]

{-
      Time Sharing
   ==================
-}

unique ability Interrupt
    where
        interrupt: {Interrupt } ()


unique type PState a e = Done a | Paused ('{e} PState a e)

interruptWrite : Request {e, BasicIO} x ->{e, Co, BasicIO} ()
interruptWrite result =
    match result with
        { echo text -> resume } ->
            uinterrupt
            echo text
            handle resume () with interruptWrite
        { result } -> ()

reifyProcess : Request {Interrupt, e} a -> PState a e
reifyProcess request =
    match request with
        { interrupt -> resume } -> Paused (_ -> handle !resume with
   reifyProcess )
        { result } -> Done result

unique ability TimeSharing
    where
        fork: {TimeSharing } Boolean

-- handler for time sharing ability
nondet : Request {TimeSharing} a -> [a]
nondet request =
    match request with
```

```
        { fork -> resume } -> (handle resume true with nondet) lib.
    base.data.List.++ (handle resume false with nondet)
        { result } -> [result]

sched : [PState a {e, TimeSharing}] -> [a] ->{e} [a]
sched ps done =
    match ps with
        [] -> done
        (Done res) +: ps' -> sched ps' (res lib.base.data.List.+:
    done)
        (Paused m) +: ps' -> sched (ps' lib.base.data.List.++ (
    handle !m with nondet)) done

timeshare : '{g, Interrupt, TimeSharing} o ->{g} [o]
timeshare m = sched [Paused (_ -> handle !m with reifyProcess)] []

{-
  Serial File System
  ==================
-}

unique ability State a
       where
            put: a -> ()
            get: () -> a

runState : a -> Request {State a} b -> b
runState v request =
    match request with
        { put v' -> resume } -> handle resume () with runState v'
        { get () -> resume } -> handle resume v with runState v
        { result } -> result

unique type DirectoryT = Directory (Text, Nat)
unique type DataRegionT = DataRegion (Nat, Text)
unique type INodeT = INode Nat Nat
unique type IListT = IList (Nat, INodeT)
unique type FileSystemT = FileSystem (List DirectoryT) (List IListT)
     (List DataRegionT) Nat Nat


initialINode : INodeT
initialINode = INode 0 0

initialDirectory : DirectoryT
initialDirectory = (Directory ("stdout", 0))


initialDataRegion : DataRegionT
initialDataRegion = DataRegion (0, "")

initialIList : IListT
initialIList = IList (0, initialINode)

initialFileSystem : FileSystemT
```

```
initialFileSystem = FileSystem [initialDirectory] [initialIList] [
   initialDataRegion] 0 0

lookupINode : Nat -> [IListT] -> Either INodeT ()
lookupINode i ilists =
    match ilists with
        [] -> Right ()
        (IList (i', inode)) +: rest ->
            if i == i' then Left inode
            else lookupINode i rest

lookupFName : Text -> [DirectoryT] -> Either Nat ()
lookupFName name directories =
    match directories with
        [] -> Right ()
        (Directory (name', i)) +: rest ->
            if name == name' then Left i
            else lookupFName name rest

modifyINode : Nat -> INodeT -> [IListT] -> [IListT]
modifyINode i inode ilists =
    match ilists with
        [] -> []
        (IList (i', inode')) +: rest ->
            if i == i' then (IList (i, inode)) +: rest
            else (IList (i', inode')) +: modifyINode i inode rest

lookupDataRegion : Nat -> [DataRegionT] -> Either Text ()
lookupDataRegion i dataRegions =
    match dataRegions with
        [] -> Right ()
        (DataRegion (i', text)) +: rest ->
            if i == i' then Left text
            else lookupDataRegion i rest

modifyDataRegion : Nat -> Text -> [DataRegionT] -> [DataRegionT]
modifyDataRegion i text dataRegions =
    match dataRegions with
        [] -> []
        (DataRegion (i', text')) +: rest ->
            if i == i' then (DataRegion (i, (text' ++ text))) +:
   rest
            else (DataRegion (i', text')) +: modifyDataRegion i text
    rest

-- fread, implementation of system read
fread : Nat -> FileSystemT -> Either Text ()
fread i fs =
    match fs with
        FileSystem directories ilists dataRegions _ _ ->
            match lookupINode i ilists with
                Left inode ->
                    match inode with
                        INode _ dataRegion ->
                            match lookupDataRegion dataRegion
   dataRegions with
```

```
                                Left text -> Left text
                                Right () -> Right ()
                Right () -> Right ()

-- fwrite, writes to the file system at the given inode with the
   given text
fwrite : Nat -> Text -> FileSystemT -> FileSystemT
fwrite i text fs =
    match fs with
        FileSystem directories ilists dataRegions _ _ ->
            match lookupINode i ilists with
                Left inode ->
                    match inode with
                        INode _ dataRegion ->
                            FileSystem directories (modifyINode i (
   INode i dataRegion) ilists) (modifyDataRegion dataRegion text
   dataRegions) 0 0
                Right () -> fs

unique ability FileRW
         where
                read: Nat -> {FileRW } Text
                write: (Nat, Text) -> {FileRW } ()

fileRW : Request {FileRW} a ->{State FileSystemT, Error} a
fileRW result =
    match result with
        { read i -> resume } ->
            let fs = get ()
                text = fread i fs
                match text with
                    Left text -> handle resume text with fileRW
                    Right () ->
                        throw FileNotFound
                        handle resume "" with fileRW
        { write (i, text) -> resume } ->
            let fs = get ()
                fs' = fwrite i text fs
                put fs'
                handle resume () with fileRW
        { result } -> result


fileEcho: Request {BasicIO} a ->{State FileSystemT} a
fileEcho m = match m with
    { echo text -> resume } ->
        let fs = get ()
            put (fwrite 0 text fs)
            handle resume () with fileEcho
    { result } -> result


fopen : Text -> FileSystemT ->{Error} Nat
fopen name fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
```

```
            match lookupFName name directories with
                Left i -> i
                Right () ->
                    throw FileNotFound
                    inext

has : Text -> [DirectoryT] -> Boolean
has name directories =
    match directories with
        [] -> false
        (Directory (name', i)) +: rest ->
            if name == name' then true
            else has name rest

fcreate : Text -> FileSystemT -> (Nat, FileSystemT)
fcreate name fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
            -- file already exists, overwrite it
            if has name directories then
                let ino = (fopen name fs)
                    inode = lookupINode ino ilists
                    match inode with
                        Left inode ->
                            match inode with
                                INode ino loc ->
                                    let dreg' = modifyDataRegion loc
    "" dataRegions
                                        (ino , FileSystem
   directories ilists dreg' dnext inext)
                        Right () -> (ino, fs) -- unreacable
            else
                let inext' = inext + 1
                    dnext' = dnext + 1
                    inode = INode inext dnext
                    ilists' = (IList (inext, inode)) +: ilists
                    directories' = (Directory (name, inext)) +:
   directories
                    (inext, FileSystem directories' ilists'
   dataRegions dnext' inext')

unique ability FileCO
    where
        create: Text -> Nat
        open: Text -> Nat

fileCO : Request {FileCO} a ->{FileRW, State FileSystemT, Error} a
fileCO result =
    match result with
        { create name -> resume } ->
            let fs = get ()
                (ino, fs') = fcreate name fs
                put fs'
                handle resume ino with fileCO
        { open name -> resume } ->
            let fs = get ()
```

```
                ino = fopen name fs
                put fs
                handle resume ino with fileCO
        { result } -> result

flink: Text -> Text -> FileSystemT ->{Error} FileSystemT
flink src dest fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
            if has dest directories then
               fs -- error, file exists
            else
                let ino = lookupFName src directories
                    match ino with
                        Left ino ->
                            let directories' = (Directory (dest, ino
   )) +: directories
                                inode = lookupINode ino ilists
                                match inode with
                                    Left inode ->
                                        match inode with
                                            INode ino loc ->
                                                  let loc' = loc + 1
                                                      inode' = INode
    ino loc'
                                                      ilists' =
   modifyINode ino inode' ilists
                                                      FileSystem
   directories' ilists' dataRegions dnext inext
                                    Right () ->
                                        throw FileExists
                                        fs -- unreachable, we know
   the file exists
                        Right () ->
                            throw FileNotFound
                            fs -- no such file src

removeINode : Nat -> [IListT] -> [IListT]
removeINode i ilists =
    match ilists with
        [] -> []
        (IList (i', inode)) +: rest ->
            if i == i' then rest
            else (IList (i', inode)) +: removeINode i rest

removeDataRegion : Nat -> [DataRegionT] -> [DataRegionT]
removeDataRegion i dataRegions =
    match dataRegions with
        [] -> []
        (DataRegion (i', text)) +: rest ->
            if i == i' then rest
            else (DataRegion (i', text)) +: removeDataRegion i rest

removeDirectory : Text -> [DirectoryT] -> [DirectoryT]
removeDirectory name directories =
    match directories with
```

```
        [] -> []
        (Directory (name', i)) +: rest ->
            if name == name' then rest
            else (Directory (name', i)) +: removeDirectory name rest

funlink: Text -> FileSystemT -> FileSystemT
funlink name fs =
    match fs with
        FileSystem directories ilists dataRegions dnext inext ->
            if has name directories then
                let ino = lookupFName name directories
                    match ino with
                        Left ino ->
                             let directories' = removeDirectory name
    directories
                                 inode = lookupINode ino ilists
                                 match inode with
                                        Left inode ->
                                            match inode with
                                                INode ino loc ->
                                                    if loc > 1 then
                                                        let loc' =
   loc - 1
                                                            inode' =
    INode ino loc'
                                                            ilists'
   = modifyINode ino inode' ilists

   FileSystem directories' ilists' dataRegions dnext inext
                                                    else
                                                        let ilists'
   = removeINode ino ilists

   dataRegions' = removeDataRegion loc dataRegions

   FileSystem directories' ilists' dataRegions' dnext inext
                                        Right () -> fs --
   unreachable, we know the file exists
                        Right () -> fs -- no such file src
            else
                fs -- no such file


unique ability FileLU
       where
            link: (Text, Text) -> {FileLU } ()
            unlink: Text -> {FileLU } ()

fileLU : Request {FileLU} a ->{FileRW, State FileSystemT, Error} a
fileLU result =
    match result with
        { link (src, dest) -> resume } ->
            let fs = get ()
                fs' = flink src dest fs
                put fs'
                handle resume () with fileLU
```

```
        { unlink name -> resume } ->
            let fs = get ()
                fs' = funlink name fs
                put fs'
                handle resume () with fileLU
        { result } -> result

fileIO m = handle (handle (handle !m with fileRW) with fileCO) with
    fileLU


{-
    pipes
    ======
-}

unique ability Await a
    where
        await: () -> a

unique ability Yield b
    where
        yield: b -> ()

pipe : ('{Yield b, e} a) -> ('{Await b, e} a) ->{e} a
pipe p c = handle c () with
                (cases
                    { x } -> x
                    { await () -> resume } -> copipe (resume) p)

copipe : (b -> {Await b, e} a) -> ('{Yield b, e} a) ->{e} a
copipe c p = handle p () with
                (cases
                    { x } -> x
                    { yield y -> resume } -> pipe resume '(c y) )

{-
    Process Syncronization
    ======================
-}

unique ability Co
    where
        ufork: Nat
        wait: Nat -> ()
        uinterrupt: ()
        nice: Nat -> Int
        renice: Nat -> Int -> ()

unique type Proc a e = Proc (Sstate a e ->{e} List (Nat, a))
unique type Pstate a e = Ready (Proc a e) | Blocked Nat (Proc a e)
unique type Sstate a e = {q: List (Nat, Pstate a e), done: List (Nat
    , a), pid: Nat, pnext: Nat}

runNext: Sstate a e ->{e} List (Nat, a)
runNext st =
```

```
    let (Sstate q done pid pnext) = st
        match q with
            [] -> done
            (pid’, Blocked pid’’ resume) +: q’ ->
                runNext (Sstate (q’ lib.base.data.List.++ [(pid’,
   Blocked pid’’ resume)]) done pid pnext)
            (pid’, Ready resume) +: q’ ->
                let st’ = (Sstate q’ done pid’ pnext)
                    Proc (resume’) = resume
                    resume’ st’

modifyQueue: Nat -> [(Nat, Pstate a e)] -> [(Nat, Pstate a e)]
modifyQueue pid q =
    match q with
        [] -> []
        (pid’, pstate) +: rest ->
            if pid == pid’ then rest
            else (pid’, pstate) +: modifyQueue pid rest

lookupNice: Nat -> [(Nat, Int)] -> Int
lookupNice pid prio =
    match prio with
        [] -> -1 -- maybe warn here
        (pid’, renice) +: rest ->
            if pid’ == pid then
                renice
            else
                lookupNice pid rest

modifyNice: Nat -> Int -> [(Nat, Int)] -> [(Nat, Int)]
modifyNice pid renice prio =
    match prio with
        [] -> [(pid, renice)]
        (pid’, renice’) +: rest ->
            if pid’ == pid then
                (pid, renice) +: rest
            else
                (pid’, renice’) +: modifyNice pid renice rest

lowestNiceInQueue: [(Nat, Int)] -> [(Nat, Pstate a e)] -> Either (
   Nat, Pstate a e) ()
lowestNiceInQueue niceValues q =
    match q with
        [] -> Right ()
        (pid, Blocked pid’ resume) +: rest ->
            lowestNiceInQueue niceValues rest
        (pid, Ready resume) +: rest ->
            let nextnice = lookupNice pid niceValues
                match lowestNiceInQueue niceValues rest with
                    Left (pid’, pstate) ->
                        let nextnice’ = lookupNice pid’ niceValues
                            if nextnice < nextnice’ then
                                Left (pid, Ready resume)
                            else
                                Left (pid’, pstate)
                    Right () ->
```

```
                         Left (pid, Ready resume)

runNextNice: Sstate a e -> [(Nat, Int)] ->{e} List (Nat, a)
runNextNice st niceValues =
    let (Sstate q done pid pnext) = st
        match q with
            [] -> done
            (pid', Blocked pid'' resume) +: q' ->
                runNextNice (Sstate (q' lib.base.data.List.++ [(pid
   ', Blocked pid'' resume)]) done pid pnext) niceValues
            (pid', Ready resume) +: q' ->
                match lowestNiceInQueue niceValues q with
                    Left (pid', Ready resume) ->
                        let st' = (Sstate q' done pid' pnext)
                            Proc (resume') = resume
                            resume' st'
                    Left (pid', Blocked pid'' resume) ->
    -- unreachable
                        let st' = (Sstate q' done pid' pnext)
                            Proc (resume') = resume
                            resume' st'
                    Right () ->
                        let st' = (Sstate q' done pid' pnext)
                            Proc (resume') = resume
                            resume' st'

minNice : Int
minNice = -20

schedAging: Sstate a e -> Request {Co, e} a ->{e, State [(Nat, Int)
   ]} List (Nat, a)
schedAging st request = match request with
    {result} ->
        let (Sstate q done pid pnext) = st
            done' = done lib.base.data.List.++ [(pid, result)]
            runNextNice (Sstate q done' pid pnext) !get
    { ufork -> resume } ->
        let resume' = (Proc (st -> handle resume 0 with scheduler st
   ))
            (Sstate q done pid pnext) = st
            nicevalue = lookupNice pid !get

            -- simple heuristic to avoid starvation, switch back to
   round robin if we reach min nice
            if nicevalue - +1 <= minNice then
                let q' = q lib.base.data.List.++ [(pid, Ready resume
   ')]
                    pid' = pnext
                    pnext' = pnext + 1
                    handle resume pid' with scheduler (Sstate q'
   done pid pnext')
            else
                put (modifyNice pnext nicevalue !get)
                put (modifyNice pnext (nicevalue - +1) !get)

                pid' = pnext
```

```
                pnext' = pnext + 1

                q' = q lib.base.data.List.++ [(pid', Ready resume')]
                handle resume pid' with schedAging (Sstate q' done
   pid pnext)

    { nice pid -> resume } ->
        let (Sstate q done pid pnext) = st
            nicevalue = lookupNice pid !get
            handle resume nicevalue with schedAging st

    { renice pid newNice -> resume} ->
        let resume' = (Proc (st -> handle resume () with scheduler
   st))
            (Sstate q done pid pnext) = st
            put (modifyNice pid newNice !get)
            runNextNice (Sstate q done pid pnext) !get

    { wait pid -> resume } ->
        let resume' = (Proc (st -> handle resume () with scheduler
   st))
            (Sstate q done pid pnext) = st
            q' = if processExists pid q then
                     q lib.base.data.List.++ [(pid, Blocked pid
   resume')]
                 else q lib.base.data.List.++ [(pid, Ready resume')]
            runNextNice (Sstate q' done pid pnext) !get
    { uinterrupt -> resume } ->
        let resume' = (Proc (st -> handle resume () with scheduler
   st))
            (Sstate q done pid pnext) = st
            q' = q lib.base.data.List.++ [(pid, Ready resume')]
            runNextNice (Sstate q' done pid pnext) !get

scheduler: Sstate a e -> Request {Co, e} a ->{e} List (Nat, a)
scheduler st request = match request with
    { result } ->
        let (Sstate q done pid pnext) = st
            done' = done lib.base.data.List.++ [(pid, result)]
            runNext (Sstate q done' pid pnext)
    { ufork -> resume } ->
        let resume' = (Proc (st -> handle resume 0 with scheduler st
   ))
            (Sstate q done pid pnext) = st
            pid' = pnext
            pnext' = pnext + 1
            q' = q lib.base.data.List.++ [(pid', Ready resume')]
            handle resume pid' with scheduler (Sstate q' done pid
   pnext')
    { wait pid -> resume } ->
        let resume' = (Proc (st -> handle resume () with scheduler
   st))
            (Sstate q done pid pnext) = st
            q' = if processExists pid q then
                     q lib.base.data.List.++ [(pid, Blocked pid
   resume')]
```

```
                 else q lib.base.data.List.++ [(pid, Ready resume')]
            runNext (Sstate q' done pid pnext)

    { nice pid -> resume } ->
        handle resume +0 with scheduler st

    {renice pid newNice -> resume} ->
        handle resume () with scheduler st

    { uinterrupt -> resume } ->
        let resume' = (Proc (st -> handle resume () with scheduler
   st))
            (Sstate q done pid pnext) = st
            q' = q lib.base.data.List.++ [(pid, Ready resume')]
            runNext (Sstate q' done pid pnext)

processExists: Nat -> [(Nat, Pstate a e)] -> Boolean
processExists pid processes =
    match processes with
        [] -> false
        (pid', process) +: rest ->
            if pid == pid' then true
            else processExists pid rest

timeshare2 : '{g, Co} a ->{g} List (Nat, a)
timeshare2 m = handle !m with scheduler (Sstate [] [] 1 2)

init: '{e} a ->{e, Co} ()
init main = let pid = ufork
                if pid == 0 then
                    let a = main ()
                        ()
                else
                    wait pid

{-
    Permissions
    ============
-}

unique type Permission = Read | Write | AddUser | Grant | Revoke |
   Execute

all : [Permission]
all = [Read, Write, AddUser, Grant, Revoke, Execute]

unique ability Permit
    where
        grant: Text -> Permission -> ()
        revoke: Text -> Permission -> ()

checkPermission : User -> Permission -> [(Text, [Permission])] ->{e,
    Error, IO, Exception} ()
checkPermission user required perms =
    match perms with
        [] -> throw PermissionDenied
```

```
        (user', perms') +: rest ->
            if userToText user == user' then
                if allowed required perms' then
                    ()
                else
                    throw PermissionDenied
            else checkPermission user required rest

permissions: User -> Request {e, Permit, Session, FileRW, FileLU,
   FileCO, Co} a ->{e, Session, FileRW, FileLU, FileCO, Co, Error,
   State [(Text, [Permission])], IO, Exception} a
permissions user request =
    match request with
        -- Permissions
        {grant user' perm -> resume} ->
            checkPermission user Grant !get
            existingPerms = lookupPermission user' !get
            newPerms = perm +: existingPerms
            put (modifyPermission user' newPerms !get)
            handle resume () with permissions user

        {revoke user' perm -> resume} ->
            checkPermission user Revoke !get
            newPerms = removePermission perm (lookupPermission user'
    !get)
            put (modifyPermission user' newPerms !get)
            handle resume () with permissions user

        -- Users
        {ask var -> resume} ->
            checkPermission user Read !get
            answer = ask var
            handle resume answer with permissions user
        {su user' -> resume} ->
            su user'
            handle resume () with permissions (Username user')
        {adduser user' -> resume} ->
            checkPermission user AddUser !get
            adduser user'
            handle resume () with permissions user
        {setvar var val -> resume} ->
            checkPermission user Write !get
            setvar var val
            handle resume () with permissions user

        -- Files
        {read i -> resume} ->
            checkPermission user Read !get
            text = read i
            handle resume text with permissions user

        {write (i, text) -> resume} ->
            checkPermission user Write !get
            write (i, text)
            handle resume () with permissions user
```

```
        {link (src, dest) -> resume} ->
            checkPermission user Write !get
            link (src, dest)
            handle resume () with permissions user

        {unlink name -> resume} ->
            checkPermission user Write !get
            unlink name
            handle resume () with permissions user

        {create name -> resume} ->
            checkPermission user Write !get
            ino = create name
            handle resume ino with permissions user

        {open name -> resume} ->
            checkPermission user Read !get
            ino = open name
            handle resume ino with permissions user

        {ufork -> resume} ->
            checkPermission user Execute !get
            let pid = ufork
                handle resume pid with permissions user

        {nice pid -> resume} ->
            checkPermission user Execute !get
            let nicevalue = nice pid
                handle resume nicevalue with permissions user

        {renice pid newnice -> resume} ->
            checkPermission user Execute !get
            renice pid newnice
            handle resume () with permissions user

        {wait pid -> resume} ->
            checkPermission user Execute !get
            wait pid
            handle resume () with permissions user

        {uinterrupt -> resume} ->
            checkPermission user Execute !get
            uinterrupt
            handle resume () with permissions user

        {result} -> result

lookupPermission: Text -> [(Text, [Permission])] -> [Permission]
lookupPermission var perms =
    match perms with
        [] -> []
        (var', perms') +: rest ->
            if var == var' then perms'
            else lookupPermission var rest
```

```
modifyPermission: Text -> [Permission] -> [(Text, [Permission])] ->
    [(Text, [Permission])]
modifyPermission var perms perms' =
    match perms' with
        [] -> [(var, perms)]
        (var', perms'') +: rest ->
            if var == var' then (var, perms) +: rest
            else (var', perms'') +: modifyPermission var perms rest

removePermission: Permission -> [Permission] -> [Permission]
removePermission perm perms =
    match perms with
        [] -> []
        perm' +: rest ->
            if permEquals perm perm' then rest
            else perm' +: removePermission perm rest

allowed: Permission -> [Permission] -> Boolean
allowed perm perms =
    match perms with
        [] -> false
        perm' +: rest ->
            if permEquals perm perm' then true
            else allowed perm rest

permEquals : Permission -> Permission -> Boolean
permEquals perms1 perms2 =
    match perms1 with
        Read ->
            match perms2 with
                Read -> true
                _ -> false
        Write ->
            match perms2 with
                Write -> true
                _ -> false
        AddUser ->
            match perms2 with
                AddUser -> true
                _ -> false
        Grant ->
            match perms2 with
                Grant -> true
                _ -> false
        Revoke ->
            match perms2 with
                Revoke -> true
                _ -> false
        Execute ->
            match perms2 with
                Execute -> true
                _ -> false

initialPermissions : [(Text, [Permission])]
initialPermissions = [("root", all)]
```

```
{-
    Errors
-}

unique type EType = PermissionDenied | FileNotFound | FileExists |
   UserExists | NoSuchUser | UnknownError

toText: EType -> Text
toText = cases
    PermissionDenied -> "Permission denied"
    FileNotFound -> "File not found"
    FileExists -> "File exists"
    UserExists -> "User exists"
    NoSuchUser -> "No such user"
    UnknownError -> "Unknown error"

unique ability Error
    where
        throw: EType -> ()

fail : Request {e, Error} a ->{e, IO, Exception, Status} Empty
fail request =
    match request with
        { throw err -> resume } ->
            printLine (toText err)
            exit 1
        { result } -> exit 0

warn : Request {e, Error} a ->{e, IO, Exception} a
warn request =
    match request with
        { throw err -> resume } ->
            printLine (toText err)
            handle resume () with warn
        { result } -> result

{-
    Retrofitting fork
-}

nondet2 : Request {TimeSharing} a -> [a]
nondet2 request =
    match request with
        { fork -> resume } ->
            let pid = ufork
                handle resume (pid != 0) with nondet2
        { result } -> [result]

 {-
     Unix
    ======
-}

unix : '{e, BasicIO, FileRW, FileCO, FileLU, Error, Session, Permit,
    Co} a ->{e, IO, Exception} [(Nat, Nat)]
```

```
unix m = handle
            (handle
                (handle
                    (handle
                        (handle
                        (handle
                            (handle
                                (handle
                                    (handle
                                        (handle
                                            (handle
                                                (handle
                                                    (handle
                                                        init m
                                                    with permissions
     (Username "root"))
                                                with runState
    initialPermissions)
                                            with env (Username "root
    "))
                                        with runState
    initialUserspace)
                                    with fileCO)
                                with fileLU)
                        with fileRW)
                    with runState initialFileSystem)
                    with interruptWrite)
                with basicIO)
            with warn)
        with exitHandler)
    with scheduler (Sstate [] [] 1 2)
```

Appendix B

Profiling Code

The following code was used to profile the effect oriented and conventional versions of the code. The code was run with the `timers` library [22] and the results were recorded in Section 5.2.

```
adduser': [User] -> User -> [User]
adduser' store user =
    store :+ user

su' : [User] -> User -> [User]
su' store user =
    match store with
        [] -> store
        (u +: rest) ->
            let uname = userToText u
                username = userToText user
                if uname == username then
                    u +: rest
                else
                    store

ask': [User] -> Text
ask' store =
    match store with
        [] -> ""
        (u +: rest) ->
            let uname = userToText u
                uname

timings _ =
    Timer.start "conventional"
    internal n env =
        if n == 0 then
            []
        else
            let uname = Nat.toText (!randomNat)
                addeduser = adduser' [] (Username uname)
                newEnv = su' addeduser (Username uname)
                internal (n-1) newEnv
    Timer.stop "conventional"
```

```
report _ = printReport timings

timings' _ =
    Timer.start "effectful"
    internal n =
        if n == 0 then
            ()
        else
            let uname = Nat.toText (!randomNat)
                adduser uname
                su uname
                internal (n-1)

    internal 1000
    Timer.stop "effectful"


timingsHandler' _ = handle (handle (handle !timings' with env (
   Username "root")) with runState initialUserspace) with warn

report' _ = printReport timingsHandler'
```

Bibliography

[1] Gordon Plotkin and John Power. Computational effects and operations: An overview. 2002.

[2] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014.

[3] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic notes in theoretical computer science*, 319:19–35, 2015.

[4] Dennis M Ritchie and Ken Thompson. The UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[5] Daniel Hillerström. Foundations for programming and implementing effect handlers. *Ph.D Thesis*, 2022.

[6] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

[7] Daan Leijen. Implementing algebraic effects in c. Technical Report MSR-TR-2017-23, June 2017.

[8] Dan R. Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. High-level effect handlers in C++. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1639–1667, 2022.

[9] Ningning Xie and Daan Leijen. Effect handlers in haskell, evidently. In Tom Schrijvers, editor, *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, pages 95–108. ACM, 2020.

[10] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM, 2015.

[11] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 1–12. ACM, 2014.

[12] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 500–514. ACM, 2017.

[13] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

[14] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In James Chapman and Wouter Swierstra, editors, *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pages 15–27. ACM, 2016.

[15] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2–6, 2018, Proceedings 16*, pages 415–435. Springer, 2018.

[16] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[17] Oleg Kiselyov and Chung-chieh Shan. Delimited continuations in operating systems. In Boicho N. Kokinov, Daniel C. Richardson, Thomas Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007, Roskilde, Denmark, August 20-24, 2007, Proceedings*, volume 4635 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2007.

[18] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *ACM SIGPLAN Notices*, 48(9):145–158, 2013.

[19] Robin Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 157–173. Elsevier, 1975.

[20] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Unison Computing. User-defined abilities · Unison programming language `https://www.unison-lang.org/docs/language-reference/user-defined-abilities/`.

[22] Rúnar Bjarnason. Unison timers library. `https://share.unison-lang.org/@runarorama/timers`.

[23] Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. From capabilities to regions: Enabling

efficient compilation of lexical effect handlers. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[24] Naoya Furudono, Youyou Cong, Hidehiko Masuhara, and Daan Leijen. Towards efficient adjustment of effect rows. In *Trends in Functional Programming: 23rd International Symposium, TFP 2022, Virtual Event, March 17–18, 2022, Revised Selected Papers*, page 169–191, Berlin, Heidelberg, 2023. Springer-Verlag.

[25] Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. Efficient compilation of algebraic effect handlers. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.