Applications of Process-Oriented Design

James Whitehead II

Worcester College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary Term 2014

Dedicated to my big sister Michelle

Acknowledgements

I would like to begin by thanking Thomas Harper for his support. I am proud that we were able to take this step together. I would also like to thank my family: James and Lee Whitehead, Michelle Hastings, Robert Whitehead, and Gregory Whitehead for never doubting what I could accomplish. Throughout my life you have supported and nurtured my curiousity and my pecularity, and that is a huge part of who I am today.

My time at Oxford has been filled with incredible people: my friends and boat-mates at Worcester College, the "honorary" members of the Hertford MCR, and everyone at the Computing Laboratory. Each of you have played a role in this part of my life and you have my gratitude. I wish to extend special thanks to Jamie Anderson and Amelia Earl for their support and friendship, and to Thomas Gibson-Robinson for his assistance throughout the development of this research.

Thanks are also due to Bernard Sufrin and Michael Spivey, my research supervisors, for their patient guidance and encouragement.

Abstract

Concurrency is generally considered to be difficult due to a lack of appropriate abstraction, rather than inherent complexity. Lock-based approaches to mutual exclusion are pervasive, despite the presence of models that are easier to understand, such as the message-passing model present in CSP (Communicating Sequential Processes). CSP provides a rich framework for building and reasoning about concurrent systems, but has historically required a change of programming language or paradigm in order to work with it. The Go programming language is a modern, imperative programming language that includes native support for processes and channels. The popularity of this language has grown and more and more people are being exposed to the fundamental ideas of CSP.

There is a gap in the understanding of how a restrictive formal model can interact with and support the development of concurrent programs in a language such as Go. Through a series of case studies and analysis, we show how the CSP concurrency model can be used as the basis for the design of a concurrent system architecture without requiring the program to be written entirely as the composition of processes. It is also possible to use the CSP process algebra to build abstract models and use model-checking tools to verify properties of a concurrent system. These models can then be used to guide the decomposition of a system into a more fine-grained concurrent system.

This thesis bridges the gap between the development of CSP-style concurrent software and the formal model of CSP. In particular, it shows how it is not necessary for a program or programming language to conform to rigid structure in order for CSP to be a useful tool for the development of reliable and easy to understand concurrent systems.

Contents

1 Introduction **1**

1.1 Contributions 2

2 Background **4**

2.1 Communicating Sequential Processes 4

2.2 occam and the Transputer 6

2.3 CSP as a Process Calculus 8

2.4 occam-π 9

2.5 Libraries for CSP-Style Concurrency 10

2.6 Concurrency in Go 13

2.7 Summary 17

3 CSP Style Concurrency in Go **18**

3.1 Processing Data Using Pipelines 19

3.1.1 Prime Number Sieve 21

3.2 Process Networks 23

3.2.1 Graceful Termination 28

3.2.2 Broadcasting a Shutdown Signal 31

3.3 Asynchronous and Parallel Computation 32

3.4 Mutual Exclusion for Shared Resources 34

3.5 Process-Oriented Client-Server Pattern 38

3.6 Encapsulation of Concurrency 43

3.7 Summary 47

4 Serving Web Content with Dynamic Process Networks **48**

4.1 Serving Web Content 48

4.2 Concurrent Web Servers in Go 52

4.3 Collaborative Responses 58

4.4 Streaming Web Components 61

4.5 Defining Components . . . 68
4.5.1 Decision-making Components . . . 69
4.5.2 Stateful components . . . 70
4.6 Combining Web Handling Components . . . 73
4.7 Summary and Evaluation . . . 75

5 Building a Concurrent File Server **78**
5.1 Background . . . 79
5.1.1 UNIX file system . . . 79
5.1.2 Explicit Locking in the UNIX Operating System . . . 81
5.1.3 The MINIX File Server . . . 83
5.2 MINIX Implementation in Go . . . 85
5.3 Modelling the MINIX File Server . . . 91
5.4 Adding Concurrency for I/O Calls . . . 99
5.5 Modelling Internal Subsystems . . . 104
5.6 Investigating Additional Concurrency . . . 110
5.6.1 Block Cache . . . 110
5.6.2 Concurrent File I/O . . . 111
5.6.3 Inode Table . . . 114
5.6.4 Allocation Table . . . 114
5.7 Implementation . . . 114
5.7.1 *minixfs* Usage Example . . . 115
5.7.2 File-Backed Block Device . . . 115
5.7.3 Concurrent Inode Cache . . . 117
5.8 Avoiding Concurrency Pitfalls . . . 122
5.8.1 Block Ownership . . . 122
5.8.2 Managing Access to Inodes . . . 124
5.8.3 FS Server . . . 124
5.9 Summary . . . 126

6 Conclusions **127**
6.1 CSP-Style Concurrency . . . 127
6.2 Process-Oriented Design . . . 129
6.3 Modelling CSP-Style Concurrent Systems . . . 129
6.4 Future Work . . . 130
6.5 Summary . . . 131

A Introduction to the Non-Concurrency Aspects of Go **132**
A.1 Hello World . . . 132
A.2 Variable declarations . . . 133
A.3 Function declarations . . . 134
A.4 Arrays and slices . . . 135
A.4.1 Variadic functions . . . 136
A.5 Pointer types . . . 137
A.6 Structured data . . . 138
A.7 User-defined data types . . . 139
A.8 Defining methods . . . 140
A.9 Interface types . . . 141
A.10 Inheritance through composition . . . 145

B File Server Models **147**
B.1 Sequential Model . . . 147
B.1.1 Assertions: Sanity check . . . 149
B.1.2 Assertions: Normal . . . 149
B.1.3 Assertions: Not enough file descriptors . . . 149
B.2 Dual Servers Model . . . 149
B.2.1 Assertions: Sanity check . . . 152
B.2.2 Assertions: Normal . . . 152
B.2.3 Assertions: Not enough file descriptors . . . 153
B.3 Internal Servers Model . . . 153
B.3.1 Assertions: Sanity check . . . 158
B.3.2 Assertions: Normal . . . 158
B.3.3 Assertions: Not enough file descriptors . . . 159

References **160**

Chapter 1

Introduction

Concurrent programming is generally considered to be difficult. This difficulty is not the result of inherent complexity, but rather a result of lack of appropriate abstraction and familiarity. A majority of concurrent programs are written using multiple threads of control that share a single address space. Coordination between these threads is accomplished through the use of shared memory. Each follows an ad-hoc protocol for accessing and updating this memory, making it possible to execute multiple threads concurrently without having them interfere with each other. This type of concurrency is ubiquitous, being directly supported by most hardware and operating systems. However it is an extremely low-level mechanism for coordination; the lack of abstraction obscures important details about the dependencies and relationships between different tasks. This makes it difficult to draw conclusions about how such a system might behave.

There are other abstractions for coordination in concurrent systems that are more disciplined and easier to reason about. Communicating Sequential Processes (CSP) is a formal model for describing and reasoning about the behaviour of concurrent systems. One of the main principles of CSP is that a program can be structured as the parallel composition of simpler processes that communicate with each other by passing messages. More generally, the CSP-style of concurrency is characterized by independent, anonymous, concurrent processes that communicate with each other using synchronous message passing over explicit channels. Although it is not possible to write programs directly using CSP, the occam programming language was heavily inspired by its semantics and abstractions. It enables a style of development that can be called process-oriented programming, but has not seen widespread or high-profile use.

Having a reasonable abstraction and view of concurrency can make it easier to build reliable systems. These abstractions can be broken when necessary without

eliminating all of the benefits of using the formal model. Similarly, it is possible to create a portion of such a system that is mathematically pure, meaning it is possible to verify a portion of a program. The introduction of CSP-style concurrency in an imperative programming language can create an environment in which new conventions and disciplines emerge in a way that advances the development of concurrent software.

In 2009, the Go programming language [12] was introduced by a small team of distinguished software engineers at Google, including pioneers of UNIX, the Plan 9 operating system, and other notable projects. The language combines systems-level programming, managed memory, a static type system, dynamic composable interfaces, and CSP-style concurrency. Since its introduction, Go has been used to develop a wide range of applications. These include hardware emulation [32], distributed caching [2], Linux container management [1] and database query caching and optimization [6].

Many new developers have come to the language with no prior experience with process algebra or formal models of concurrency and a belief that explicit locks are the only way to write concurrent programs. As a result, there has been an emergence (and often re-invention) of designs, techniques, and patterns that harness the power of CSP-style concurrency for structuring and reasoning about concurrent programs. The inclusion of CSP-style concurrency in Go creates a powerful environment where process-oriented programming techniques can blend together with more traditional system design and architecture.

In this thesis we explore the restrictions, limitations, and expressiveness of a programming language that combines CSP-style concurrency semantics within a traditional imperative programming language. Using well-defined patterns from process-oriented programming, we explore two different case studies of software development. Throughout this process, we identify the drawbacks of from operating outside the restrictions of the formal model and show how the lack of restrictions make it possible to express certain patterns in an efficient manner.

1.1 Contributions

This thesis makes the following contributions:

1. A catalogue and analysis of architecture patterns for concurrent computation, including pipelines and process networks, asynchronous and parallel computation, and shared resource management. Each pattern is presented in terms of

how it can be accomplished using channels and processes, and the limitations of these approaches are discussed.

2. A compositional component-based concurrent web server framework, as evidence of the effectiveness of re-considering a well-known problem in terms of process-oriented program design.

3. A case study of using the CSP process algebra and associated model-checking tools to assist in the decomposition of a large existing monolithic system for the purpose of improved structure, isolation, and concurrent performance. This is accompanied by several models of a concurrent file system with various levels of concurrency.

The remainder of this thesis is organised as follows.

Chapter 2 introduces the ideas that underlie the CSP process algebra, and a style of programming based on sequential processes and communication using synchronous message passing over explicit channels. Chapter 3 illustrates several architectural patterns using the style of concurrency present in the Go programming language. These designs are utilized in the following chapters to build concurrent solutions to existing problems. Chapter 4 presents a case study of constructing a concurrent web server where multiple components in a pipeline or process network can collaborate in the generation of the response to a web request. Chapter 5 describes the decomposition of an existing monolithic and sequential implementation of the UNIX file system, using the CSP process algebra and automated model checking to verify the behaviour of the system. Finally, Chapter 6 draws conclusions from the work presented in this thesis.

Chapter 2

Background

This thesis explores the ways in which formal models of concurrency can influence and provide a disciplined framework for the development of programs. In particular we consider the CSP process calculus in which programs are composed of processes that interact with each other by passing messages over channels. Structurally, this is similar to object-oriented programming, where programs are built from objects that interact with each other by making method calls. In fact, both the Smalltalk [34] and Objective-C [26] families of language implement method calls using message passing. In CSP, communication between processes is inherently concurrent, whereas Simula [28] style method calls on objects are sequential.

This chapter introduces some of the core concepts that underlie the style of concurrent programming, derived from CSP, that is used throughout this thesis.

2.1 Communicating Sequential Processes

Communicating Sequential Processes generally refers to the formal model for describing and reasoning about the properties of concurrent systems [38, 60]. The ideas that would eventually evolve into the CSP_{85} process calculus were initially presented as a model parallel programming language. In this chapter the former is written as CSP_{85}, while the programming language is written as CSP_{78}. The numbers in subscript indicate the year in which the version of CSP was introduced.

The "Communicating Sequential Processes" paper [37] suggests that input and output, in the form of communication between concurrent processes, should be provided as programming language primitives. It also argues that the parallel composition of communicating sequential processes is a fundamental way of structuring programs. This paper abandons lower-level techniques for mutual exclusion such as semaphores and monitors. Instead, synchronization and data transfer are accomplished by passing

messages between processes. The paper uses this active, predictable communication mechanism construct elegant solutions to problems of structure clash, data representation, and mutual exclusion.

The CSP_{78} notation is fairly simple, providing primitive commands for assignment and communication (input and output). A form of Dijkstra's guarded commands enables a single mechanism for branching control and for input commands. If more than one guard is valid (or more than one communication is able to proceed) than one is chosen in a non-deterministic way. An alternation (set of guarded commands) can be performed repetitively, ending when all of the guards in the alternation fail. A collection of processes can be executed in parallel using parallel composition; such a composition will not terminate until all of the component processes have terminated.

For example, the following code defines a process that repeatedly copies characters from the process named `west` and transmits them to `east`. This process will continue to read and write characters until there is no more input to receive, at which point the repetition command will end and the process will terminate.

```
COPY :: *[c:character; west?c -> east!c]
```

One of the examples presented in this paper is the task of reformatting cards of printed text. Each input card consists of 80 characters which can be read from the card reader. The program must print the same sequence of characters to a line printer with 125 characters per line. This problem exhibits structure clash [39], where there are two incompatible ways of structuring the stream of data. Problems with structure clash are considered difficult to solve without using coroutines [25].

Hoare's solution consists of three single-purpose processes that run in parallel. The first process, called `DISASSEMBLE`, converts the 80-character cards into a stream of single characters. The `ASSEMBLE` process does the work of constructing new lines from this stream of characters and sending them to the line printer. The third process is called `COPY`, and acts as a buffer between the other two processes. Given appropriate definitions for these processes, they can be substituted in the following CSP_{78} parallel construct in order to solve the problem.

```
[west::DISASSEMBLE || X::COPY || east::ASSEMBLE]
```

In this parallel command, each process is assigned a name: `west` for the disassembly process, `X` for the copy process, and `east` for the assembler. These names match those used in the input and output communication commands in the process definitions; this early version of CSP did not include named channels. Directing messages

using these names makes it possible to decouple the behaviour of a single process definition from the behaviour of the parallel composition of several processes.

For example, the `COPY` process can be replaced by one with a different behaviour, so long as it reads input from `west` and sends its output to `east`. A slight variation of the card reformatting problem, called Conway's Problem [25], requires replacing adjacent asterisks with an upwards arrow. This can be solved by using the `SQUASH` process in place of `COPY`, without any other changes to the program.

```
[west::DISASSEMBLE || X::SQUASH || east::ASSEMBLE]
```

This example helps to illustrate the benefit of the `COPY` process. Although the original problem does not strictly require it, it helps to decouple the assembly and disassembly processes. This behaviour is precisely that which can be achieved by channels in later versions of CSP-style concurrent languages.

Each of the processes used in these examples has a simple purpose, enabling them to be written in a way that is clear and easy to understand. The communication between these processes is explicit, and decoupled through the use of process names. When the processes are connected together, they are able to accomplish a task that is significantly more complex than the behaviour of any individual process.

This pipeline of processes is also self-terminating, due to the way the that parallel composition is defined in CSP_{78}. When the disassembly process runs out of input, the process will terminate. This causes the copying process to break out of the receive loop, causing it to also terminate. Finally, the assembly process breaks out of its receive loop and then writes the final line of the input (if any), and the process and parallel composition terminate.

CSP_{78} advocates a simple, disciplined model of concurrency that makes it easy to construct programs with a concurrent structure. Rather than focusing on low-level monitors and semaphores, communication and synchronization is performed by sending messages between processes. The same approach is used when communicating with external input/output devices, providing a consistent view of how a program interacts internally and with its external environment.

2.2 occam and the Transputer

Part of the motivation given for CSP was that:

> "...developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each

with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a multiprocessor." [37]

The Transputer microprocessor architecture, introduced in 1984, was designed to enable the efficient execution of parallel programs by allowing multiple processors to be connected together with efficient communication links. The design of this architecture, and of the occam programming language used to develop for it, drew heavy inspiration from the work on CSP. Rather than using time-slicing and multiprogramming to simulate the parallel execution of programs, this architecture had the potential to simplify the development of operating systems while providing more opportunities for parallelism.

From the occam 2.1 Reference Manual [64]:

> "occam enables an application to be described as a collection of *processes*, where the processes execute concurrently, and communicate with each other through *channels*. Each process in such an application describes the behaviour of a particular aspect of the implementation, and each channel describes a connection between two processes. This approach has two important consequences. Firstly, it gives the program a clearly defined and simple structure. Secondly, it allows the application to exploit the performance of a system which consists of many parts."

Communication in occam is modelled after CSP, except that messages are sent over explicit point-to-point channels rather than being directed to a specific process name. These channels can be passed to processes as parameters, enabling an even stronger decoupling. An occam version of the parallel composition from the card reformatting example can be done with a channel in place of the copy process.

```
CHAN OF BYTE in, mid, out:
PAR
  disassemble(in, mid)
  assemble(mid, out)
```

In the occam version of this program, each process communicates using only the channels it receives as parameters; it does not need to know the name of the process on the other end of the channel or what sort of process it is. A new process can be inserted between `disassemble` and `assemble` by creating a new process and channels for communication and passing these new channels as parameters to the other processes,

without needing to alter either existing process definition. This makes it much easier to write processes that can be re-used across several programs.

The semantics of communication in occam are slightly different from those of CSP_{78}, although they precisely match those of CSP_{85}. Input and output, including within an alternation, will block indefinitely, even if the process on the other end of the channel has terminated. This is unlike the previous example, where the `disassembly` process terminated, causing receiving processes to fail and resulting in cascading termination of the entire program. In occam, processes are generally designed to run forever. This can make it more difficult to write parallel blocks that will gracefully and automatically terminate when a given task is complete; such behaviour must be programmed explicitly [72].

Alternation allows a process to wait for input on more than one channel, but receive from only one of the ready channels at a time. A simple multiplexing process can be written using the `ALT` construct, listening for input on two channels and simply forwarding the result to an output channel:

```
PROC muxer(CHAN OF BYTE in.a, in.b, out)
  WHILE TRUE
    BYTE v:
    ALT
      in.a ? v
        out ! v
      in.b ? v
        out ! v
```

The occam programming language embodies a practical realisation of the ideas that were described in Hoare's original paper: a way of writing consistent and understandable concurrent programs. What began as a theoretical model programming language was refined and used as the interface language for a family of parallel microprocessors.

2.3 CSP as a Process Calculus

Following its initial introduction in 1978, the ideas at the core of CSP_{78} were developed into a formal system for describing and reasoning about the behaviour of concurrent systems [38]. These systems are constructed by combining process definitions using a small set of primitive operators. These equations can then be manipulated using algebraic reasoning.

CSP_{85} makes it possible to compare two process definitions and to define relations of behavioural refinement between them. Using these techniques, a CSP model can

be analysed in order to determine whether it has a specific behaviour, whether it is susceptible to deadlock, or whether it has the potential to diverge. We will make use of the CSP process algebra in this thesis and the reader is assumed to have a basic knowledge of how it works. More detailed information about CSP can be found in [60, 38].

The term CSP is used throughout this thesis to refer to ideas from both the programming language and the process algebra. At its core, CSP-style concurrency is characterised by independent concurrent processes that communicate with each other by sending messages on explicit channels.

2.4 occam-π

The occam programming language is a good realisation of the core ideas of CSP: that input and output can be treated as primitives of a programming language, and that programs can be structured as the composition of sequential processes that communicate with each other. The language restricts the ways in which such systems can be constructed (for example by preventing the sharing of channels) in order to ensure that the communication primitives, while implemented in a simple and efficient way, continued to behave in a way that was consistent with the process algebra. This makes it possible to verify properties of such programs.

In 2005, Welch and Barnes introduced occam-π [73], a programming language that blends the static semantics of CSP with the more dynamic nature of the *pi*-calculus [48], another formal model of concurrency. Advances in the understanding of concurrency theory made it possible to introduce these features in a way that remained consistent with the process algebra. Some of the more notable features are the introduction of mobile channels and data types, and the safe sharing of channel ends. The 'mobile' type can be communicated over channels using a movement semantics, resulting in a transfer of ownership. An important consequence of this is that the identity of one channel can be communicated over another, making it possible to build a dynamically reconfigurable process network that can adapt to the needs and conditions of the environment unlike the `PAR` construct of occam.

occam-π also introduces a new mechanism for spawning processes, using the `FORK` keyword. A process that is forked in this manner is free-running, meaning that it is not connected to any other processes unless the process is given a channel as a parameter. This is particularly useful when building programs where processes start

and end at unpredictable times, such as a network server that spawns new processes for each incoming connection [62]:

```
PROC network.server (LISTEN.SOCKET listen)
  WHILE TRUE
    SOCKET sock:
    SEQ
      accept.connection (listen, sock)
      FORK client.worker (sock)
```

Although `FORK` provides a way to spawn new processes, it does not include a mechanism for waiting for such a process to complete. A `FORKING` block can be used to group several forked processes together, achieving the same termination behaviour as a `PAR` block: a join on each process. The major difference between `PAR` and `FORK` is that the latter runs in parallel with the original process, while the program stops running during the execution of `PAR` block. This means a program can spawn new processes and continue processing, with more fine-grained control over when those processes must terminate.

The language is supported by the powerful Kent Retargetable occam Compiler (KRoC) [29], which is able to provide several safety guarantees. During compilation, checks are made to ensure there are no common concurrency bugs, such as aliasing issues that might introduce data race hazards.

The occam family of programming language have been used in a wide range of applications, including large scale multi-agent simulations [55, 76, 58, 63, 21], operating systems research and development [40], and programming field-programmable gate arrays (FPGAs) [52, 41, 53]. In addition, there is a very active research community [7] that is committed to exploring the uses of CSP-style concurrency and occam and its boundaries.

2.5 Libraries for CSP-Style Concurrency

There are several benefits to the disciplined style of concurrent programming espoused by occam and CSP. The mechanism for synchronisation and data transfer is simple and easy to understand, analogous to communication in the real world. Although the occam family of programming languages provide an expressive way of writing such programs, the quest to keep programs formally verifiable means that the language is limited to a single programming paradigm, that of pure process-oriented programming.

There have been several attempts to bring the structured CSP-style of program development to other languages and environments, typically through the introduction of a third party library. These libraries focus on providing a way to create new processes, channels for communication between processes, a method for running multiple processes in parallel, and a mechanism for alternation.

These libraries make it possible to use CSP-style concurrency within a larger program, without requiring a process-centric design throughout. Implementations exist for a variety of languages and programming paradigms, including C [19], C++ [24], Java [71, 74], Scala [66], Python [20], and Haskell [23].

The particular style of integration will vary depending on the language and the library being used. For example, the JCSP library for Java is based on a well designed class hierarchy. This includes runnable processes, guards for use in alternation, and channels for communication. A process that reads integers from one channel and then copies them to another can be written as follows:

```
import org.jcsp.lang.*;

class CopyInt implements CSProcess {
  private ChannelInputInt in;
  private ChannelOutputInt out;

  public CopyInt (ChannelInputInt in, ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  public void run () {
    while (true) {
      out.write(in.read());
    }
  }
}
```

This example, while clearly still a Java program, consists of familiar CSP elements: definitions of channels and a long running process definition. New instances of this process could be easily combined in parallel with other processes that have compatible channel types and directions. There are several advantages to this familiarity: programs can be build using well known deadlock-free patterns take from years of CSP and occam research. It is then possible to reason about the behaviour of that part of the system while simultaneously taking advantage of the modern conveniences of the Java programming language. In order to support its use in teaching [49], the JCSP library provides a wide array of processes that are pre-defined, enabling students to

construct process networks without needing to re-implement these common building blocks.

As with any domain specific extension, a problem with this approach is that the process definition can be dominated by Java syntax, making it more difficult to quickly grasp the purpose of a process. Other languages are able to achieve a better integration of the CSP-style notation within a library. As a result of the flexible syntax of Scala, the Communicating Scala Objects (CSO) library [66] is written much closer to the original CSP notation [1]. A CSO version of the producer/consumer example above can be expressed as follows:

```
def Copy(in: ?[Int], out: ![Int]) =
proc {
    repeat { out!(in ?) }
}
```

Processes can be combined using the parallel operator:

```
val system = Producer(a) || Copy(a, b) || Consumer(b)
system()  // execute the system
```

There are several problems with this style of embedding. Both the Java and Scala programming languages execute on the Java Virtual Machine, most versions of which use heavyweight operating systems threads. As a result, they are similar at runtime and therefore in performance [66]. The cost of context switching (and thus communication) between processes is dramatically higher than that of the lightweight threads used in the occam runtime.

Additionally, when CSP primitives are embedded in a general purpose programming language, it is not reasonable to expect the compiler for that language to be able to enforce occam-like rules. In occam, all channels are point-to-point and the compiler is able to reject outright a program that violates the sharing rules. occam-π allows sharing channel-ends between concurrent processes, but only if those channel ends are explicitly declared as `SHARED`, forcing the sharing processes to use explicit `CLAIM` blocks before attempting to communicate.

Despite these difficulties, this does not render CSP-style concurrency unusable in these environments, it simply shifts the responsibility to the developer to prevent such conflicts when constructing a program.

[1] It should be noted that in both the JCSP and CSO libraries, and in fact the Go programming language, a receive operation on a channel is an expression whose evaluation has a side-effect. As a result, the result of $in? - in?$ depends on the order of evaluation of the programming language rather than being a simple algebraic property and care must be taken to avoid such ambiguity.

One advantage of these libraries is that they make it possible to write occam-style concurrent systems that sit alongside the other unrelated program code. The concurrent portions of the application can be modelled using CSP and implementing in a disciplined way without requiring the entire program to be process-oriented.

2.6 Concurrency in Go

The Go programming language was initially introduced in 2009, with the first official release in 2012 [10]. It is a statically typed programming language that includes native support for CSP-style concurrency. The concurrency model is based upon goroutines, which are the Go version of processes. A goroutine is the execution of a function or method call in an independent concurrent thread of control. All goroutines in a program reside within the same address space alongside other goroutines. This is quite different from occam processes, which are designed so that they can be efficiently segmented across several processors with no shared memory. Goroutines are designed to be fast and easy to create, enabling a style of programming where many goroutines can be spawned to complete a task.

Goroutines are implemented using a hybrid threading model, where many goroutines are multiplexed across a smaller number of operating system threads. The language runtime consists of one or more operating systems threads that can execute goroutines. If a goroutine would perform a blocking action, such as make a system call, this is handled by a separate worker thread. This enables the system to continue processing other goroutines while waiting for the result of the call.

Goroutines are implemented using segmented stacks, where each goroutine is initially allocated a small amount of stack space. As it executes, the goroutine may require additional stack space, for local parameters or recursive calls. When this happens, the runtime will allocate an additional block of stack space and attach it to the pre-allocated amount [2] This technique makes it possible to lower the initial memory footprint of the goroutine, reducing the time that it takes to allocate and spawn a new process. The occam-π programming language (unlike occam) allows for recursion and utilises a similar approach for stack allocation. However the occam-π compiler, unlike Go, is able to pre-calculate exactly the space needed for each recursion and delay allocation until precisely the point where this recursion happens.

[2]The allocation strategy for stack space has changed as of Go version 1.3. Rather than using the split stack described here, the language now utilizes contiguous stacks that are re-allocated and copied as necessary. More information on the implications and motivation for this change can be found in [11].

The execution of a Go program begins with the `main` function of the program running in a goroutine. This function may spawn new goroutines by prefixing a function or method call with the `go` keyword, causing the execution of that call to be performed in a new goroutine that runs alongside the original. This is similar to the way in which occam-π supports the creation of free-standing processes using the `FORK` construct.

Each new goroutine is created with no implicit connection to the rest of the program; there is no process hierarchy, and connections to an existing network of processes must be created explicitly. Go has no language provision for "joining" these processes analogous to the `FORKING` block in occam-π. The preferred mechanism for this is to use channels for communication. These channels are statically typed and are safe for use by any number of readers or writers. Readers will compete with each other to read messages from the channel, while writers will compete to send messages. Communications are synchronous by default, and involve exactly one reader and one writer.

A goroutine can send a value on a channel using a send statement, where `foo` is a channel and `x` is a value of the appropriate type:

```
foo <- x
```

The data in a communication always travels in the direction of the arrow (leftwards). This makes it easier to see which is the value being communicated over which channel. A receive operation is performed using `<-` as a unary operator:

```
x = <-baz
```

A receive operation yields a value of the channel's element type. This can be used as an expression, such as on the right-hand side of an assignment or as a parameter in a function call. Alternatively, the communication may be used purely for synchronization, in which case the data being communicated may not be relevant. In such a case, the receive operation can be used as a statement and the resulting value will be discarded.

Channels are unbuffered by default, making communication both synchronous and blocking. This makes it possible to use them as synchronization points between goroutines as well as a means for data transfer. Go also supports the construction of channels with a fixed-size buffer. As in CSP, such a channel behaves the same as a pair of channels that are linked together with a static buffer process, but with less overhead.

Communication on these buffered channels may be asynchronous. If the buffer is not full, then a send to the channel can be performed without needing to wait for a receiver. Similarly, if the buffer contains at least one item, then a receiver need not wait for a sender. In all other situations the communications will remain synchronous.[3] This makes it possible to use channels as simple semaphores.

A goroutine may attempt to communicate on several channels using the `select` construct, which is similar to an `ALT` in occam. If any of the communications can proceed, then one is chosen using uniform pseudo-random fair choice. For example, the following program fragment will attempt to read from two channels: `deposit` and `withdraw`, and can take different action depending on which communication occurs. At most one of these communications will proceed:

```
select {
case amt = <-deposit:
    balance = balance + amt
case amt = <-withdraw:
    balance = balance - amt
}
```

A `select` statement will block until one of the branches is able to proceed. Unlike occam alternation, being able to proceed does not mean that the communication itself will be successful. Go supports the notion of closing a channel; attempts to read from a channel that has been closed will always be able to proceed. In addition, a default branch can be added to a `select` block that is always capable of proceeding. This can be used to attempt non-blocking send or receive operations. In the following example, the receive branch will be chosen if the communication can be performed without blocking, otherwise the default branch will be taken:

```
select {
case msg = <-in:
    doSomething(msg)
default:
    // channel was not ready, do something else
}
```

Go's alternation construct does not have support for the sort of guarded commands that are present in both occam and CSP. However it is possible to simulate them by taking advantage of another feature of `select`. If a branch contains a communication

[3] Go does not currently support extended rendezvous as present in Ada [57], where the receiving process has an opportunity to take action after receiving a value but before the sending process is allowed to continue. The lack of support makes it difficult to build certain types of systems, such as those that include an additional "monitoring" channel without altering the behaviour of the existing system [66, 75, 73].

with a channel that is set to `nil`, the zero value for channels, then that branch will be ignored. Although it is more cumbersome than syntactically-supported guarded commands, the same effect can be achieved:

```
increment, decrement := evaluate_guards(n)
select {
case <-increment:
    n = n + 1
case <-decrement:
    n = n - 1
}
```

The call `evaluate_guards(n)` checks the value of `n` and then returns a pair of channels for `increment` and `decrement`, replacing each channel by `nil` if it should be disabled. This can be used to only allow increment operations when the value is below a certain value, and similarly to restrict when communications on the decrement channel are possible.

Communications with time-outs are handled without the need for additional keywords or syntax. The *time* package contains a function called `After` that returns a channel on which a message will be delivered after a certain amount of time. The following statement block will wait to send a message on the output channel for five seconds, after which the timeout message will be received and that branch will proced.

```
select {
case out <- msg:
    // message was sent successfully
case <-time.After(5 * time.Second):
    // timed out, do something else
}
```

In addition to channels, Go provides support for lower-level synchronization primitives such as mutexes, condition variables, wait groups (which are similar to barriers), and atomic compare-and-swap operations. These are provided through a package which must be explicitly imported, unlike channels which are an integral part of the language. These low-level primitives can be understood in terms of an equivalent mechanism using channels. For example, a semaphore can be built from a buffered channel that transmits tokens. However, for efficiency, they are not implemented this way.

Message passing over channels is not always the most appropriate synchronization or communication mechanism, particularly with regard to performance. One of the interesting aspects of Go is that although it provides lower-level mechanisms for

synchronisation, these are second class citizens compared to channels. An often-heard mantra is "Do not communicate by sharing memory. Instead, share memory by communicating.". Although process-oriented programming is not mandated by the language, Go makes it easy to write process-oriented programs in a natural way.

2.7 Summary

This thesis explores the ways in which formal methods of concurrency interact with and support the development of concurrent software. The CSP process calculus and the occam family of programming languages provide a structure and discipline, but are currently fairly restrictive. These have been several attempts to bring CSP-style concurrency to other programming languages, but these domain specific embeddings have seen limited use.

Go is a relatively new programming language that natively supports CSP-style concurrency. Although the concurrency semantics are slightly different from those of occam, they provide a way to write easy to understand concurrent systems in a modern programming language.

Chapter 3

CSP Style Concurrency in Go

The previous chapter provided an introduction to the ideas behind a style of concurrent programming based on the CSP process calculus. Until recently, the design and development of such programs was done using the occam programming language or one of its domain-specific embeddings. Recently, a new programming language, Go, has made it possible to work natively with CSP-style processes and channels.

Although the building blocks are the same, programs written in Go are structured quite differently to those written in occam. Goroutines can be spawned dynamically throughout the execution of a program and there are no explicit blocks for parallel composition. As a result, tasks that are simple in occam, such as waiting for several processes running in parallel to terminate, require a bit more effort. On the other hand, dynamically reconfigurable networks of processes can be readily expressed in language such as Go and occam-π, where channels can be communicated and processes can be spawned during the execution of the program.

This chapter explores several different ways of structuring concurrent systems in Go, using well known CSP-style concurrent patterns in order to illustrate these differences. We present several structural patterns that are used throughout this thesis, and provide an introduction to the concurrent features of Go and the caveats that affect them. Channels and goroutines provide the tools needed to build these systems without the need for explicit locking mechanisms. This greatly influences the way developers can think about building programs and solving problems. These examples also serve as an introduction to the features and patterns used in Go programs. More background on the non-concurrency aspects of Go can be found in Appendix A.

3.1 Processing Data Using Pipelines

Anyone who is familiar with the UNIX family of operating systems is likely to have encountered the idea of using a one-line sequence of shell commands to accomplish a task. These sequences consist of simple programs that are composed together to create a pipeline, with the output of each program fed as input to the next. For example, the following snippet can be used to construct a frequency table of the words that appear in an input file, sorted in descending order of frequency:

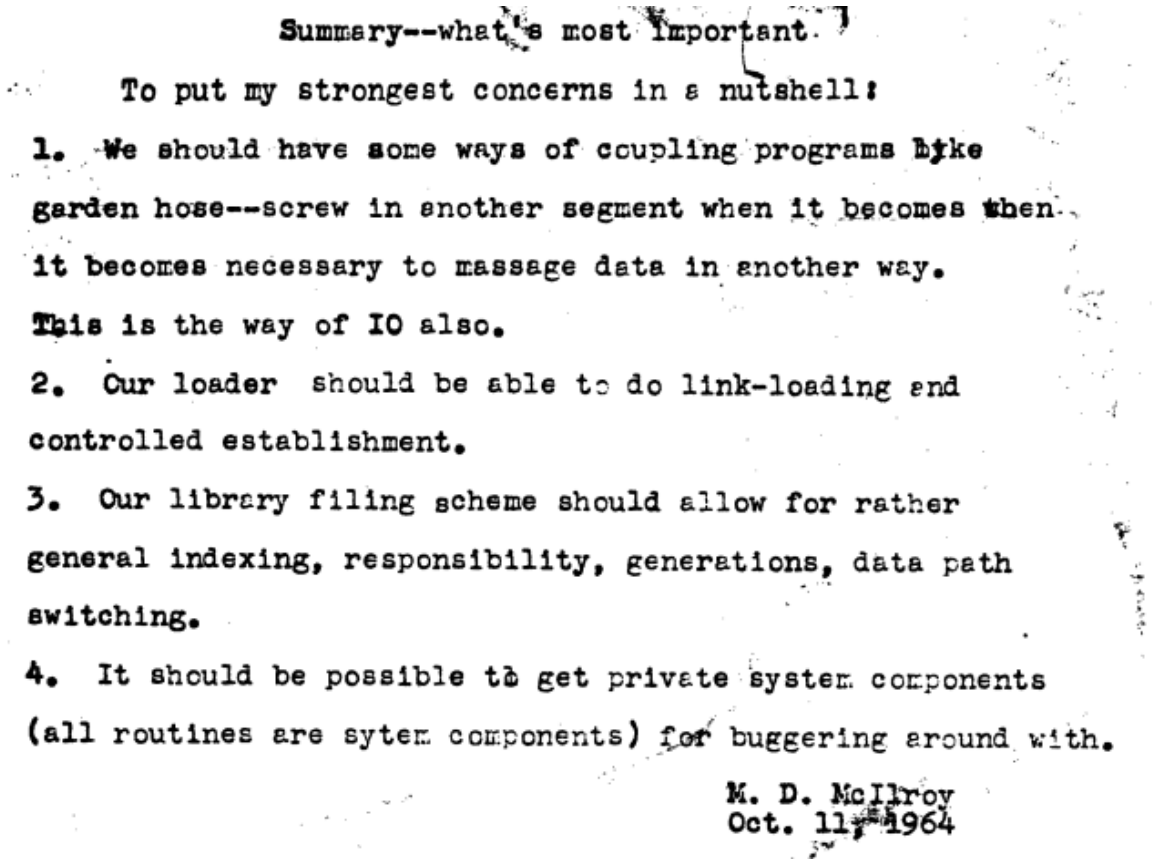
```
cat input.txt | tr -c A-Za-z '\n' | sort | uniq -c | sort -n -r
```

The magic that allows this construction to work is that each of these individual programs is designed to accept input as a stream of bytes on the "standard input" file descriptor, and communicate results on "standard output". Although none of these programs have any prior knowledge of the others, this allows them to be connected together into a useful pipeline. Each time the format of the data needs to be changed, another program can be added to the pipeline. In many cases, these processes can execute concurrently, enabling a pipeline to solve problems with volumes of data that are too large to be stored on disk. In this example, the `tr` program is able to begin the task of translating the characters in the data stream almost immediately, without needing to wait for the entire file to be read from disk. Even the `sort` program, which by its very nature has to input the entire data before producing any output, can still do intermediate processing while translation is still in progress.

Douglas McIlroy introduced this concept [5] during the development of UNIX, likening it to the way different bits of garden hose can be connected using couplers. The same principle can be extended to input and output, where devices are treated as streams of characters. Figure 3.1 shows the typewritten memo where these ideas were laid out several years before CSP was first introduced.

The same task, counting word frequencies, could have been accomplished by a single monolithic program with greater efficiency. However, any such program would still need to accomplish the same general steps: split the input into words, count the instances of each word, sort the words by frequency, and then output the list in descending order. The potential inefficiency of composing multiple programs together is a price worth paying for the simplicity, adaptability, flexibility, and extensibility that it provides.

Although pipelines are normally encountered at the operating systems level, where each component is a standalone program, they can also be used as a structuring principle in program development. For example, program compilation can be accomplished



Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

Figure 3.1: Doug McIlroy's description of the concept that would become UNIX pipes

in several phases. A lexer can read a source file and produce a stream of tokens [54], which are in turn consumed by the parser to produce an intermediate representation. The compiler can then perform multiple optimization passes before producing the compiled result.

This design makes it possible to develop, test, document, and maintain each bit of functionality independently. So long as the components use a common means of accepting input and communicating their results, they can be connected together in this fashion, regardless of the fact that each program was designed independently of the others. Gstreamer [3] is a multimedia framework based on this design, enabling the creation of media components for video editing, streaming media broadcasters, and media players. The ideas and uses of processing pipelines are explored further in Chapter 4, which details the design and development of a concurrent web server that is capable of expressing behaviour in terms of pipelines.

The following section presents an example of pipeline construction in Go.

3.1.1 Prime Number Sieve

In [47], McIlroy builds a pipeline that is capable of producing prime numbers by filtering out any known composite numbers. Each stage of the pipeline is a simple process that silently consumes any numbers that are multiples of an already found prime, and passes any other candidates on to the next stage.

Such a program is easy to construct using Go, by dynamically spawning new goroutines and connecting them together using channels. The `dump` function[1] produces candidate numbers for the pipeline, and will continue generating new numbers until the program terminates.

```
// Generate successive integers, starting with 2
func dump(out chan int) {
  i := 2
  for {
    out <- i
    i = i + 1
  }
}
```

This function introduces the `:=` operator, which is a short-hand form of variable declaration and initialization with type inference. The same declaration/assignment could have been written using the long-hand form, i.e., `var int i = 2`, but the short form is normally sufficient and considered easier to read.

[1] The names of the functions used in this program match those from the original paper.

Each stage of the sieve pipeline will be an instance of the `mesh` function running in a new goroutine, set to filter out multiples of a given value. The function reads new candidate numbers from its input channel, and discards any that are a multiple of this value. Any other values are passed on to the next stage of the pipeline.

```
func mesh(value int, in chan int, out chan int) {
  for {
    candidate := <-in
    if candidate%value != 0 {
      out <- candidate
    }
  }
```

In the original paper the `hopper` function itself is responsible for printing out the prime numbers. Our implementation changes this slightly by having it output the prime numbers to an output channel. The main goroutine is then connected to the other end of this channel, and is responsible for reading and printing the values. Owing to the synchronous nature of the channels, this is a simple way to control the flow of prime numbers computed by the sieve.

```
func hopper(in chan int, out chan int) {
  for {
    prime := <-in
    out <- prime

    pipe := make(chan int)
    go mesh(prime, in, pipe)
    in = pipe
  }
```

The other purpose of the `hopper` function is to spawn new instances of the `mesh` function and connect them to the end of the pipeline. Here, the built-in `make` function is used to construct a new channel of the appropriate type. The hopper's existing input channel and the new channel are passed as parameters to the new instance of `mesh`, and the hopper is then connected to the other end of the new channel.

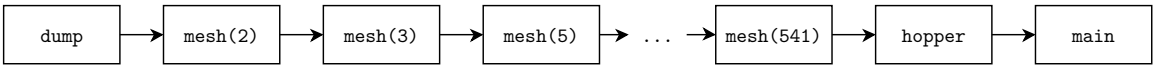


Figure 3.2: The process diagram of the prime sieve after generating one hundred prime numbers

These components are instantiated in the main function, which is also responsible for reading and printing the prime numbers:

```
func main() {
  pipe := make(chan int)   // connect the dump to the hopper
  primes := make(chan int) // primes will be delivered on this channel

  go dump(pipe)           // dump numbers into the sieve
  go hopper(pipe, primes) // catch primes and extend pipeline

  // Read and print the first 100 prime numbers
  for i := 0; i < 100; i++ {
    prime := <-primes
    fmt.Printf("%d ", prime)
  }

  fmt.Printf("\n")
}
```

The main function spawns instances of `dump` and `hopper` and connects them together using a channel. The hopper is then given a different channel on which it can deliver the stream of prime numbers. No further orchestration is necessary; once the `go` keyword has been used to spawn a new goroutine, it will begin executing alongside the other goroutines in the program. The main goroutine is then free to read 100 prime numbers from the other end of this channel and print them to standard output. The Go runtime and scheduler will work behind the scenes to execute the pipeline in an efficient way.

Figure 3.2 illustrates the network of processes present after printing 541, the last prime number that is read from the pipeline in this example.

3.2 Process Networks

A structured sequential pipeline is a special case of a more general process network, where processes can be wired together in almost any configuration. Such networks can be used to structure systems where distinct processes interact with each other through the course of execution. The simple concurrent systems that are introduced

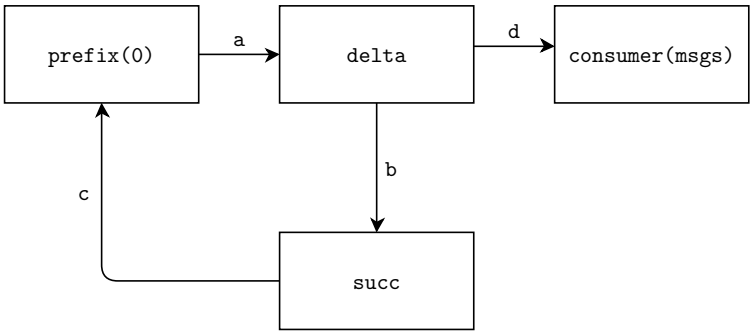


Figure 3.3: The *commstime* process network

in most CSP textbooks— vending machines and automated bank teller machines— can easily be represented by a connected system of processes.

Process networks lie at the heart of process-oriented programming. In [62], Sampson describes a wide range of process and structural design patterns. A series of case studies help to illustrate how these patterns have been used in the construction of biological simulations, operating systems development, and video games. Although the Go approach to building such networks is less structured than that of occam, the section shows by example how some of the same patterns can be applied with similar benefits.

Commstime is a micro-benchmark that has become part of the folklore of process-oriented concurrent programming. It is a process network that consists of three processes connected in a cycle, and a consumer process which measures and reports the benchmark results. This program is purposely quite small, ensuring that it fits in the CPU cache on most target machines. The processes do not perform any real work, meaning the majority of the execution time is spent in the concurrency kernel or program runtime. This makes it possible to measure the overhead that is introduced by the runtime for context switching between different processes, and for communication over channels. Figure 3.3 depicts the configuration of the *commstime* process network.

The purpose of the `prefix` process is to inject a number into the process network, and then copy numbers from its input channel to its output channel.

```
func prefix(in <-chan int, out chan<- int, num int) {
  for {
    out <- num
    num = <-in
  }
}
```

This definition illustrates how the type of a channel can be narrowed from a bidirectional channel to a unidirectional channel end. If T is the element type of the channel (`int` in this case), then `<-chan T` is the type of a receive-only channel. Similarly, `chan<- T` is the type of a send-only channel. A bidirectional channel value can be converted to a directional channel end, but there is no way to convert in the opposite direction.

Directional channel ends makes it possible for the programmer to express conventions about which channels a process may use for sending and which it may use for receiving, and to have these conventions checked by the Go compiler. This can help to reduce obvious communication errors during development.

The `delta` process reads a value from its input channel and then communicates that value on two output channels, in sequence (similar to the 'tee' program in the UNIX world). Only when both communications have been successful will it read another input value.

```
func delta(in <-chan int, out1 chan<- int, out2 chan<- int) {
  for {
    num := <-in
    out1 <- num
    out2 <- num
  }
}
```

The `succ` process is connected to one of the outputs from `delta`. It reads a number from the input channel and sends its successor to the `prefix` process.

```
func succ(in <-chan int, out chan<- int) {
  for {
    num := <-in
    out <- num + 1
  }
}
```

The `delta` component here is the simpler sequential variant of the typical "parallel" delta, which sends to both of the output channels in parallel and waits for both to complete before continuing. This is done by spawning a new process to perform one

of the sends in the background and then waiting for it to complete. The Go version of a parallel `delta` requires quite a bit additional work: a function that can be spawned in a new goroutine, a channel that can be used to signal completion of the send, and the communication on this channel.

```
func par_delta(in <-chan int, out1 chan<- int, out2 chan<- int) {
  send1 := func(x int, done chan<- bool) {
    out1 <- x
    done <- true
  }

  done := make(chan bool)
  for {
    num := <-in
    go send1(num, done)
    out2 <- num
    <-done
  }
}
```

This is much more complicated than the same pattern in occam-π, where it is sufficient to change the `SEQ` block into a `PAR` block. In occam-π, the parallel version of `delta` adds the cost of spawning and waiting for the termination of a new process in each cycle of the network. By comparing the timings for a network with the "sequential delta" and one with a "parallel delta", it is possible to measure the minimum cost of process creation and waiting for completion.[2]

The overall effect of this cycle of processes (`prefix`, `delta`, and `succ`) is to generate the integer sequence beginning with the start number, with each number 'leaking' out over the second `delta` output channel. This is connected to the `consumer` process, where the benchmarking and accounting is performed. The consumer consists of a simple loop that reads a certain number of messages, specified by the `msgs` parameter, and measures the time needed to do so.

[2]The language runtime should restrict itself to a single CPU core during the execution of this benchmark to prevent the overhead from the process scheduler from dominating the benchmark. This should not negatively impact the benchmark, as there is little opportunity for parallelism.

```
func consumer(in <-chan int, msgs int) {
  start := time.Now()
  for i := 0; i < msgs; i++ {
    <-in
  }
  end := time.Now()

  // print the timing statistics
  delta_ns := end.Sub(start).Nanoseconds()
  fmt.Printf("Receiving %d messages took %d nanoseconds\n", msgs, delta_ns)
}
```

From this measurement, the program can derive the cost of one cycle through the process network by dividing the total cost by the number of messages sent. Each 'loop' through the network requires four channel communications, resulting in eight context switches between processes.

The process network is assembled by allocating the channels that connect the processes together, and then spawning new goroutines for the different processes with the channels as parameters.

```
func main() {
  a := make(chan int)
  b := make(chan int)
  c := make(chan int)
  d := make(chan int)

  go prefix(c, a, 0)
  go delta(a, b, d)
  go succ(b, c)
  consumer(d, 100000)
}
```

In this function, the cycle of processes is spawned in the background, while the consumer process is run using a normal function call in the main goroutine. Since a Go program ends as soon as the `main` function terminates, this ensures that the benchmark has been completed before the program exits.

Although *commstime* is a micro-benchmark for which occam is well-suited, it can be useful as a rough comparison of the relative performance between different CSP-style concurrent programming languages. Table 3.1 shows the thread style, median, and standard deviation for completing a single cycle of the process network for a selection of programming languages and environments with the program written in each case in a straightforward way.

These benchmarks were performed on the same machine with an Intel(R) Core(TM)

Language/Library	Thread style	Median	Standard deviation
Occam	lightweight	179	6.7
Go	lightweight	1716	11.6
C with LibCSP	pthreads	28592	217.7
Scala with CSO	JVM	41713	221.9
Java with JCSP	JVM	53914	254.1

Table 3.1: Time to complete a cycle of the *commstime* network (in nanoseconds)

i3-2367M @ 1.40GHz processor with 4 cores and 8 GB of memory. Measurements were taken as the elapsed time to complete either 1,000,000 or 100,000 cycles over the course of 10 different runs, ensuring that benchmarks could be completed within a reasonable time. Go and occam, which both map multiple process/goroutines to operating system threads each perform significantly better than the alternatives which map each process its own operating system thread (pthreads, JVM threading).

There is nevertheless a factor of ten difference between Go and occam. This is likely due to several factors related to the maturity of the respective runtimes, and in particular their schedulers. The Go scheduler is currently undergoing a redesign that should help to reduce the overhead of communicating between goroutines but may not improve results in this particular benchmark.

3.2.1 Graceful Termination

The tools that a programming language provides can greatly influence the way programs written in the language are designed and structured. For example, in the occam family of languages, there is no support for terminating a running process. As a result, systems written using the language are typically designed assuming that they will run indefinitely (or at least until the program itself has terminated). This makes them easier to read and develop, since it removes the need to handle special cases of termination. However there are obviously cases where the safe termination of a process network is desirable.

In [72], Welch describes several systematic approaches for terminating or resetting a network of processes. One such technique spreads a 'special' value using the existing channels of the process network. For example, if the channel normally carries positive integers, a negative value could be used to indicate termination. Alternatively, each process could be equipped with a 'quit' channel on which termination signals could be delivered. However, as discussed in the paper, both of these techniques are difficult

to apply safely to arbitrary process networks. Care must be taken to avoid discarding messages that are still important.

The occam style libraries for C, Java, etc., discussed in Chapter 2 typically address this issue by expanding the interface of channels. For example, the JCSP library defines the semantics of "poisoning" a channel [65], a concept which was first introduced in the C++CSP library [22]. A process can 'poison' a channel, which causes an exception to be thrown in any process that is currently waiting for the channel. These other processes can then catch this exception and perform any steps necessary to terminate cleanly, including spreading the poison to its other channels. A similar technique is used by other libraries, with varying terminology and delivery mechanism.

JCSP extends the idea of channel poison by introducing "immunity". A channel can be created with a level of "immunity" to these poison signals, which are given a numeric strength value. Such a channel will ignore any poison values that are below its immunity threshold, making it possible to selectively reset or terminate a small sub-network of a larger process network.

Go provides more support for automatic graceful termination than occam, but significantly less than JCSP and other similar libraries. A channel can be closed using the `close` function, rendering it unusable for sending. However, any values that are currently in the channel's buffer can still be received as usual. This is primarily a mechanism which a producer can use to signal the end of a sequence to its consumers, allowing them to terminate cleanly. For example, the following producer will send three different values to its output channel and then close it:

```
func producer(out chan<- int) {
  out <- 1
  out <- 2
  out <- 3
  close(out)
}
```

Go does not support exceptions in the style of Java or C++. Instead, error conditions are communicated through additional return values that can be checked explicitly. Each receive operation has a second return value which can be checked to get additional information about the communication (these can be and have been omitted thus far).

```
data, ok := <-foo
```

When `ok` is true, this indicates that the value was received as a result of communication. This is always the case for channels that haven't been closed, otherwise the

communication would not succeed. However, if `ok` is false, then the communication was unblocked due to the channel being closed. In this case, `data` will contain the zero value for the channel's element type.

As a result of this, it is important that the receiver check the second return value in cases where it is possible for the channel to be closed. Otherwise, the process could loop forever receiving zero values, never being given an opportunity to terminate.

In such cases, it is customary to use the Go mechanism for iterating over a data structure or sequence via use of the `range` keyword. This pattern correctly handles channels that may be closed, removing the need to check each communication explicitly.

```
func consumerRange(in <-chan int) {
  for num := range in {
    fmt.Printf("%d, ", num)
  }

  fmt.Printf("closed")
}
```

The fact that channel closure can only be safely communicated from producer to consumer can influence the way in which programs are designed and structured. Consider the prime sieve pipeline, which was written to generate the first 100 prime numbers. In this original design, it is not easy for the producing goroutine to initiate the shut down of the pipeline; it has no information about how many prime numbers have emerged from the sieve. When the 100^{th} prime emerges at the end of the pipeline, more primes and composite values will already have begun their progress through it, and would have to be flushed out explicitly, making termination from that source rather difficult.

A simpler problem would be to produce the sequence of prime numbers that are less than some specific maximum value. Then the `dump` process could close its output channel, causing the pipeline to shut down.

```
func dump(max int, out chan int) {
  for i := 2; i < max; i++ {
    out <- i
  }
  close(out)
}
```

In order for this to work, the `mesh` and `hopper` functions need to be altered to handle the channel closure. The mesh process can be rewritten to use the `range` clause and then close its output channel when the loop has finished.

The `hopper` process is slightly more difficult to change, since the channel on which it listens changes each time a new prime number is found. However, the `range` clause refers to a single unvarying channel, making it unsuitable for this task. The loop can be written to explicitly check for channel closure, taking the swapping of channels into account.

```
func hopper(in chan int, out chan int) {
  for {
    prime, ok := <-in
    if !ok {
      break
    }
    out <- prime

    pipe := make(chan int)
    go mesh(prime, in, pipe)
    in = pipe
  }
  close(out)
}
```

The result is a program that is more idiomatic: the producer notifies the consumer of the end of the data stream, and the pipeline of processes are able to terminate cleanly. This required a change to the system for generating primes, to move the limiting of results from the ultimate consumer to the producer.

3.2.2 Broadcasting a Shutdown Signal

Once a channel has been closed, an attempt to read from it will always proceed immediately, regardless of how many zero values have been read from the channel in its closed state. This makes it possible to use a single channel to broadcast a shutdown request to a collection of processes, provided that the processes do not communicate with each other.

Each process can listen to the broadcast channel in a `select` block. When the signalling process closes the channel, that branch will be immediately eligible for communication. No actual data can be conveyed when using a broadcast channel in this message, since each process will just receive the zero value for the channel's element type. However, it does provide an easy way to notify (in a single operation) an arbitrary number of goroutines of a change in condition.

3.3 Asynchronous and Parallel Computation

Thus far in this chapter, channels and goroutines have been used to build pipelines for data processing and networks of processes that are connected together. Although some Go programs are structured in this way, goroutines and channels are more commonly used to augment an existing program with some form of concurrency. Since spawning new goroutines is relatively low-cost, they are frequently used to spawn tasks to be completed asynchronously, or in parallel.

This is particularly useful, as network and file operations are accomplished using blocking system calls. Channels make it quite easy to convert a normal synchronous function call into one that can compute the result in the background and then deliver the result asynchronously. This can be used to keep the main event loop of a program responsive during CPU-intensive work, or to enable a program to initiate multiple I/O requests without waiting for others to complete.

Consider a basic cache for web pages, which fetches and stores the content of multiple web pages. We could start by defining a function `FetchWebpage` that takes the URL of a page to fetch and then returns the contents of the URL. This function is an example of a blocking I/O operation that may involve one or more system calls. A typical program will invoke this function directly for each web page that needs to be fetched, possibly storing the results for further processing:

```
pages := make(map[string]string)

pages["google"] = FetchWebpage("http://google.com")
pages["yahoo"] = FetchWebpage("http://yahoo.com")
pages["comlab"] = FetchWebpage("http://www.cs.ox.ac.uk")
```

This code first constructs[3] a new hash table (called a *map* in Go), and then goes on to fetch and store the home pages of three websites. This is a good example of a problem that could benefit from asynchronous computation: there is no reason the second and third requests must wait until the first has completed. This is the same technique that is used by modern web browsers: making multiple requests in parallel to speed up the fetching of all the required resources.

By augmenting the existing code, it is possible to write a function that can fetch webpages asynchronously:

[3]The `make` function here is the same one used to construct channels. This is a built-in function used to allocate channel, map, and slice (views on arrays) values.

```
func FetchWebpageAsync(url string) chan string {
  rchan := make(chan string)
  go func() {
    body := FetchWebpage(url)
    rchan <- body
  }()
  return rchan
}
```

This new function is a wrapper around the existing version, but does not return the result directly. Instead, it returns a new channel on which the result will be delivered when it becomes available. The actual call to `FetchWebpage` is performed in an anonymous function that is executed in a new goroutine. This is a common idiom in Go programs for spawning a goroutine whose body is specified inline and has access to values from the enclosing environment. The new function makes a call to compute the result, and then attempts to deliver it via the channel that was returned to the caller.

`FetchWebpageAsync` returns a channel that can be used at a later time. A program can make multiple requests for different web pages, and then later wait for the responses.

```
google := FetchWebpageAsync("http://google.com")
yahoo := FetchWebpageAsync("http://yahoo.com")
comlab := FetchWebpageAsync("http://www.cs.ox.ac.uk")

pages["google"] = <-google
pages["yahoo"] = <-yahoo
pages["comlab"] = <-comlab
```

The added cost of the asynchronous version of this function is the time needed to create a new closure and channel, spawn a goroutine in which the computation will be performed, the overhead of synchronously communicating a single value on a channel, and the eventual cost of reclaiming the channel, goroutine, and closure during garbage collection.

Each call to `FetchWebpageAsync` spawns a new goroutine to absorb the latency, freeing the main program up to take other action. Since the result is eventually communicated over a channel, the program can even test to see if the result is available:

```
select {
    case pages["google"] = <-google:
        // result ready to be used
    default:
        // result not available
}
```

This makes it possible to spawn several instances of long-running computations, and to make an informed decision about when it is most appropriate to block and wait for the results. The result channel can even be passed on to another goroutine to do the collecting.

When a computation being performed is CPU-bound, the Go runtime supports the concurrent execution of goroutines on several operating system threads. This allows programs to take advantage of multiple cores or processors.

This general technique— spawning a new goroutine to perform some task and then using a channel to join when the task is complete— is used frequently in Go programs. Go lacks the structured `PAR` block that is present in the occam family of languages, but the same functionality can be implemented manually using channels. This technique is used in Chapter 4 to change the semantics of a synchronous interface, and to impose limits on the number of outstanding requests in a web server.

3.4 Mutual Exclusion for Shared Resources

In a language with support for concurrency, care must be taken to regulate access to any resources that are shared among multiple processes. These resources might be file descriptors, network sockets, global state, or values that are shared using references. This is trivially achieved in occam programs by construction, but requires more attention in a language like Go where it is possible to share goroutines implicitly.

A traditional approach to regulating access is to couple the resource with a mutex or other explicit lock, and then establish a convention for safe access. A process must acquire this lock before accessing the resource, and be sure to release it when complete. If this convention is followed consistently for all processes, this establishes 'critical sections' during which a process can expect to have exclusive access to the resource. As the program executes, these critical sections will be serialized in some manner.

In the example from the previous section, the hash map in which the contents are being stored is only accessible from a single goroutine. As a result, there is no need to implement a mutual exclusion policy, despite the mutation of a reference value. However, if the program was changed so that the map was shared among multiple goroutines, it would need to be protected as a shared resource.

Go provides low-level synchronization through the standard libraries, in the *sync* package. These include condition variables (which allow multiple goroutines to wait

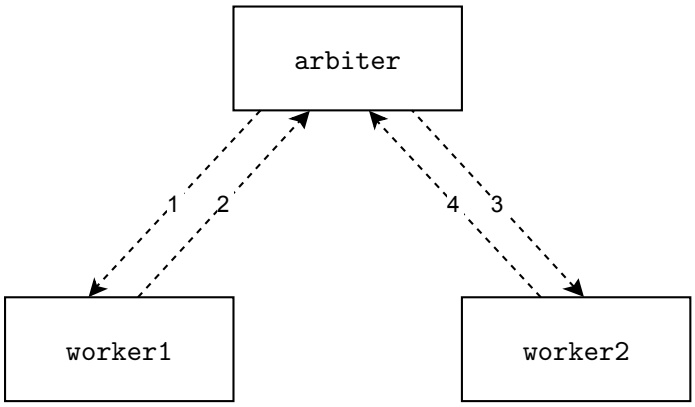


Figure 3.4: Two workers sharing a resource using an arbiter process

for some form of event), two forms of mutexes (traditional and concurrent-read-exclusive-write) and a limited form of barrier (called a wait group). We could build a `FetchAndStoreWebpage` function that can add contents to a hash map in a safe way, by adding the map and a mutex as additional parameters:

```
func FetchAndStoreWebpage(url string, pages map[string]string, m *sync.Mutex) {
  body := FetchWebpage(url)

  m.Lock()
  pages[url] = body
  m.Unlock()
}
```

This is the more traditional lock-based approach to mutual exclusion. Many instances of this function can execute concurrently without fear of race hazards. Although familiar, the use of explicit shared locking mechanisms such as this is generally discouraged in Go development. A more idiomatic (but no more safe) approach is to pass the shared resource between different goroutines using a channel. Rather than having immediate access to the resource (and agreeing to a specific locking convention for initial access), a function can request access to the resource by attempting to receive from this channel.

When using this approach, channel communication serves (by convention) as a means of transferring ownership of the data structure. When the communication is completed, the process is considered the owner of the resource and has exclusive access to it. Once complete, the reference is then sent back on the channel, ready for the next client process. The channel may be connected to an explicit arbiter process that manages the ownership transfer, or the channel itself could have a one-place buffer.

Responsibility for following these ownership transfer rules belongs to the programmer, unlike in occam-π, where mobile data types bind these rules into the language in a way that cannot be violated.

Figure 3.4 shows the way in which two workers may interact with an arbiter process to share a resource. `worker1` requests the resource by receiving on a channel (shown by 1 in the figure). When the operation is complete, it returns by sending the resource back to the abiter (2). At the same time, `worker2` can try to request the resource (3); this operation will block until it becomes available. When the arbiter receives the resource, it can immediately be sent to `worker2` (4).

A version of `FetchAndStoreWebpage` can be written as follows:

```
func FetchAndStoreWebpage(url string, arbiter chan map[string]string) {
  body := FetchWebpage(url)

  pages := <-arbiter
  pages[url] = body
  arbiter <- pages
}
```

One advantage of using channels rather than locks for this purpose is that the semantics of channel operations are clearly laid out in the language specification, making the behaviour of this function clear and unambiguous. The channel receive operation will block until a message has been received, and the channel send that corresponds to the unlock operation will similarly block until the hash map has successfully been sent on the channel. These synchronization points are clearly defined when using channels, and include the notion of communicating with some outside agent (a detail that is lost when making use of a mutex to coordinate access to the resource).

A tangible benefit of this approach is that the function cannot possibly access the resource before it has been granted (temporary) ownership. This helps to alleviate a type of race hazard that occurs when a procedure accesses a resource without properly 'locking' it first. This is a particularly dangerous programming mistake because it is inherently silent; the other processes that continue following the locking conventions would have no way of knowing that there is a potential issue.

The other type of error, where a procedure accesses a resource after it has been 'unlocked', is very difficult to catch in Go, regardless of whether a channel or mutex is used as the locking mechanism. This can lead to data corruption, timing bugs, and a host of other concurrency issues, since it invalidates the agreed-upon conventions for how the data will be accessed. The same is not true in occam-π which applies

a ‘movement semantics’ for mobile data types [14, 73], rather than the copying semantics used elsewhere. When a process communicates a mobile value, the variable referencing it becomes undefined and, until re-assigned, inoperable. In addition it is not possible to alias mobile values. All of this is verified by the compiler, resulting in compile-time errors when a program attempts to use a mobile data value that has been communicated.

It may be possible to alter the Go programming language in order to make such an ownership transfer explicit, rather than a matter of convention. By restricting arbitrary aliasing and using some form of separation logic, it may be possible to enable to Go compiler to enforce such transfer rules. It is not clear what the overhead of such a feature would impose on the compiler or to what degree the semantics of the language would need to change. However, we consider this a question that is open for further exploration, considering the potential gain for communicating reference values over channels.

In order for this approach to work, the channel must be buffered (or attached to a goroutine which serves as a 1-buffer). Otherwise a process would not be able to send the resource back on the channel until another process was interested in acquiring it. This would prevent a system with a single process from being able to function, since it could never release the resource. The resource and channel can be initialised as follows:

```
pages := make(map[string]string)
pagesCh := make(chan map[string]string, 1)

pagesCh <- pages
```

When creating the channel, the second parameter to the `make` function is used to specify a fixed size buffer for the channel, in this case the size should be `1`. Finally, the counter is initially ‘unlocked’ by communicating the resource on the channel, making it available to anyone that has access to the channel. Since there is space in the channel’s buffer, the send operation is completed without blocking. As long as the counter is present in the channel’s buffer, receive operations are also non-blocking. In any other situation, communications are synchronous, as if the channel were unbuffered. A 1-buffer ensures that locking (receiving on the channel) blocks until the resource is available, while unlocking (sending on the channel) can complete without needing to block.

It is possible to emulate a binary semaphore using a channel with a buffer or arbiter process. Although there is a performance penalty for using this technique instead of

explicit locks or mutexes, there are clear benefits to using channels to regulate access to shared resources.

The presence of a channel is an indication that a process or goroutine has some external dependency. The types of communication (sending versus receiving) also indicate the direction of the dependency, information that is obscured when using locks. In addition, it is possible to attempt multiple communications in a `select` block, including receiving a shared resource. This makes it possible to implement "locking" operations that can safely be aborted. A process can also attempt to require multiple shared resources at a time, only proceeding with the first one that is available. Although unconventional, channels provide a simple and consistent mechanism for accessing shared resources.

3.5 Process-Oriented Client-Server Pattern

The previous section introduced a technique for regulating access to a shared resource using a channel to transfer ownership between different goroutines. This is a communication-oriented approach to explicit locking, and has similar drawbacks and limitations; each process must conform to the conventions for access, or the mutual exclusion property can no longer be guaranteed.

A radically different approach that comes from process-oriented programming is to encapsulate functionality into server processes. These processes accept requests from clients (as messages on channels), take some action on the client's behalf, and deliver the result back to the client. This prevents the client from improperly accessing the resource and inadvertently causing livelock or deadlock. Guidelines for using the *client-server* structural pattern in designing occam programs are laid out in [61, 46]; here we show how similar guidelines can be developed for Go.

Informally, we may find the behaviour of such a system understandable due to how it aligns with a way that we commonly interact with agents in our daily lives at the bank, the grocery store, and with services via the Internet. We communicate in order to make a request, and then wait for a response from the service.

Consider the hash map used to store the contents of web pages in the previous section. Rather than using a channel or mutex to regulate access, it would be possible to build a server process that handles all operations involving the resource. A concurrent hash table service would support querying, updating, and removing entries from the mapping. Requests would be submitted via a channel exposed to clients, and responses would be delivered back using the same mechanism.

Go does not have direct support for protocol definitions, which can be used to specify the types and expected sequence of messages that should be sent on a channel. Instead, each of the request types can be defined as a new data type:

```
type Query struct {
  url string
}

type Update struct {
  url string
  body string
}

type Remove struct {
  url string
}
```

Go uses name equivalence for `struct` types, making it possible to distinguish between a `Query` and a `Remove` request, even though both types have the same fields.

It will also be necessary to define a result type, so the server can communicate the result of a hash table lookup. In Go, such a lookup will return two values:

```
type QueryResult struct {
  body string
  ok bool
}
```

Channels in Go can only be used to communicate values of a specific type. In order to utilize a single input channel to communicate with the web cache service, rather than having one channel per request, we need a type that encompasses all of the possible requests types. In Go, this is accomplished using an interface type, which provides a form of subtype polymorphism. Appendix A provides a more comprehensive introduction to the way that interfaces are typically used in Go programming.

For the purposes of this section, it is sufficient to say that an interface is a collection of methods. For example, the `Reader` interface is defined by the *io* package and consists of a single method that can be used to read an array of bytes from a source:

```
type Reader interface {
    Read([]byte) (int, error)
}
```

Any type that implements this method will satisfy the interface and can be used wherever a `Reader` is expected. This makes it possible to write code that can operate on "things that can be read from", rather than being written specifically to work with network streams or files.

The interface that consists of no methods, `interface{}`, is an interface that *every* type in Go implements, since its requirement is vacuously true; every type includes the empty method set. This makes it possible to use the empty interface in order to enable a channel to carry messages of any type, using a `chan interface{}`. This is a bit like how the `Object` type can be used in Java programs to accomplish the same, albeit in a way that is not type-safe.

For the concurrent web cache, the server process could be connected to two channels: one for receiving requests and one for sending query responses (the only type of response it can possibly send).

```
func server(in <-chan interface{}, out chan<- QueryResult) {
    webCache := make(map[string]string)

    for req := range in {
        // handle incoming request
    }
}
```

A server process such as this then needs a way to distinguish between each of the different request types. This is accomplished using a type assertion, which has the form `x.(T)`, where `x` is a value and `T` is a type. The assertion results in a value of type `T` and a boolean value that indicates whether or not the assertion was successful.

Given a value of type `interface{}`, it can be converted into a `Reader` only if the underlying concrete type satisfies the interface:

```
reader, ok := val.(Reader)
```

If `ok` is true, then `reader` will be a non-zero value of type `Reader`.

As an extension of type assertions, it is possible to select an action depending on the type of request by writing a special form of the `switch` statement that utilizes the `type` keyword.

```
for req := range in {
  switch req := req.(type) {
  case Query:
    body, ok := webCache[req.url]
    out <- QueryResult{body, ok}
  case Update:
    webCache[req.url] = req.body
  case Remove:
    delete(webCache, req.url)
  default:
    panic("bad request")
  }
}
```

Though `req` has type `interface{}` outside the `switch` statement, the form `switch req := req.(type)` causes it to have a more specific type in the arms of the `switch`, making fields like `req.url` accessible. Depending on the underlying type of the incoming message, some work is performed and if necessary a response is constructed and sent back via the output channel.

The empty interface makes it possible for more than one type to be sent on the input channel, but it is too general; any value whatsoever might be sent in by a client process. In order to prevent this, the request types can be defined in a way that groups them together, using a new non-empty interface type. The interface only needs to have a single method that is used to identify the valid request types:

```
type request interface {
  is_request()
}
```

This interface can then be satisfied by each of the request types; it is sufficient to define a method with an empty body:

```
func (req Query) is_request() {}
func (req Update) is_request() {}
func (req Remove) is_request() {}
```

This creative abuse of interface types is an established Go idiom and means that the channel type can be changed from the overly general `chan interface{}` to the very specific `chan request`. In this way, the type system can ensure that only valid request types are being sent to the data server. The same could be done for response types, but in this case there is only one type of valid response.

Note that the name of the type, `request`, and the name of the method, `is_request`, both begin with a lower-case letter. There is a very simple convention for visibility in Go: identifiers that begin with an uppercase letter are visible outside the package, whereas any others are not. This can be used to prevent implementation details (such as these marker methods) from leaking out to other packages. A positive side-effect of this is that it prevents a malicious user from building a new type that has an `is_request` method and sending it to the service.

The creation of these 'marker methods' and interfaces is only necessary if there isn't already an interface that ties together a collection of types. As an example, the *go/ast* package provides support for constructing and manipulating abstract syntax trees for Go, and uses this technique as a poor-man's algebraic data type for the different nodes that can appear in a program. However, unlike the no-op methods

used here, the `Node` interface actually includes useful methods which can be used to identify the source location of a given node.

In order to use the web cache in a program, the input and output channels must be created and the server goroutine must be launched. Go packages typically provide a helper function that takes care of any initialisation and returns the pair of channels needed to access the service:

```
func NewWebCache() (chan<- request, <-chan QueryResult) {
  in := make(chan request)
  out := make(chan QueryResult)

  go server(in, out)
  return in, out
}
```

A client can interact with the service using the channels that are returned by this constructor:

```
in, out := NewWebCache()

// fill in some entries in the cache
in <- Update{"http://google.com", FetchWebpage("http://google.com")}
in <- Update{"http://yahoo.com", FetchWebpage("http://yahoo.com")}
in <- Update{"http://www.cs.ox.ac.uk", FetchWebpage("http://www.cs.ox.ac.uk")}

// remove one of the entries
in <- Remove{"http://yahoo.com"}

// look up an entry
in <- Query{"http://www.cs.ox.ac.uk"}
result := <-out
if !result.ok {
  fmt.Printf("Cache miss\n")
} else {
  fmt.Printf("Cache hit: %s\n", result.body)
}
```

The result is a data structure that can be safely accessed by multiple processes at the same time, so long as each client adheres to the interface protocol: clients must immediately wait for a response when they query the phone book, but update and remove requests do not have to be followed by a wait for a response. A client that forgets to wait for the response to a query, or attempts to listen for a response from an update/remove operation, would cause the service to become unresponsive or corrupt. The server goroutine might be blocked waiting for someone read from the output channel, or a client goroutine may receive a message that was intended for another client.

Some of these issues can be mitigated using the asynchronous result channel technique from Section 3.3. The client could send a response channel as part of the request, or the server could generate new channels as requests arrive. This would serve to de-couple the server from the client, providing it with the freedom to decide whether requests should be served sequentially, or if the server can perform multiple requests concurrently.

Building a server process in the process-oriented style is rather clumsy in Go, particularly when compared to occam. However, this design helps to create a clear separation between the concurrency of the clients that are interacting with the resource, and the server implementation itself. A lock-based approach would consist of several threads contending for the same set of locks, with their execution being serialized in some way by the scheduler or runtime. With this arrangement, the potential for concurrency is an implicit property of the type and granularity of the locks and the relationship between them.

In contrast, the process-oriented approach make concurrency an explicit property of the implementation. The server loop might accept multiple messages on several different input channels and multiplex them to worker goroutines, but these details can be seen in the construction of the server implementation itself. The block cache implementation in Chapter 5 is implemented in this way, making it easy to manage the potential concurrency of a critical part of a concurrent file system.

3.6 Encapsulation of Concurrency

Regardless of whether an implementation uses explicit locks or channels to regulate access to a shared resource, it still relies quite heavily on the client following the proper protocol. If a process fails to unlock the mutex when it is done with the resource, other processes will block waiting to acquire a mutex which will never be unlocked. If a client fails to consume the response on the output channel of a service or waits for a response when one is not expected, then messages will be incorrectly delivered to the wrong clients.

In many cases, such an implementation can be improved by abstracting away from the underlying synchronization mechanism being used, providing an interface via which clients can interact with the resource. This interface can then be used as a normal synchronous API that can be safely accessed by multiple processes; the client can be written without concern for the specific mechanism used for mutual exclusion.

For the web cache server, there are three main operations that make up its interface:

```
type WebCache interface {
  Query(url string) (string, bool)
  Update(url, body string)
  Remove(url string)
}
```

One reason for utilizing an interface type rather than a concrete type with a set of methods is that multiple types may implement the same interface, with different implementations of the resource or locking mechanisms. So long as a program only has access to this interface, the implementation can ensure that the methods in the interface can be invoked from multiple goroutines without fear of race hazards or conditions.

Methods must be associated with a user-defined type; in this case it makes sense to bundle the input and output channels together in a structured data type, since the methods will need to send to and receive messages from the server.

```
type webCacheChans struct {
  in chan<- request
  out <-chan QueryResult
}
```

This channel bundle type is private, as are each of its fields, since they begin with a lower-case letter. The interface type `WebCache` will be exported, but the `webCacheChans` type will be visible only within this package. This ensures that the only way another package is able to interact with the object is using the methods that are defined in the interface; even a type assertion would be rejected by the compiler since the concrete type is not exported.

In order to properly implement this interface, the data type must provide these three methods:

```
func (wc webCacheChans) Query(url string) (string, bool) {
  wc.in <- Query{url}
  result := <-wc.out
  return result.body, result.ok
}

func (wc webCacheChans) Update(url, body string) {
  wc.in <- Update{url, body}
}

func (wc webCacheChans) Remove(url string) {
  wc.in <- Remove{url}
}
```

These methods handle the fine-grained details of interacting with the resource, such as communicating with the server process using channels and performing any type assertions that might be necessary.

Finally, the constructor must be changed to return an instance of the implementation type that can be used by the clients.

```
func NewWebCache() WebCache {
  in := make(chan request)
  out := make(chan QueryResult)

  go server(in, out)
  wc := webCacheChans{in, out}
  return wc
}
```

Rather than returning the input and output channels, a new `webCacheChans` value is created to bundle the input and output channels together. This value is then returned to the caller as a `WebCache` interface type. It doesn't matter that the concrete data structure is not exported; it implements the interface which is visible by other packages. The client will still be able to access the methods that are published without having any further information about how they are provided. This provides a more natural way of accessing the web cache:

```
cache := NewWebCache()

// fill some entries in the cache
cache.Update("http://google.com", FetchWebpage("http://google.com"))
cache.Update("http://yahoo.com", FetchWebpage("http://yahoo.com"))
cache.Update("http://www.cs.ox.ac.uk", FetchWebpage("http://www.cs.ox.ac.uk"))

// remove one of the entries
cache.Remove("http://yahoo.com")
```

An interface type can be used to encapsulate a concurrent data structure, obscuring the details of how it has been implemented. Implementers of clients must know that the web cache is safe to be accessed from multiple goroutines, but they do not need to know the details of how this is achieved. This makes it possible for the service developer to use whatever mechanism is most appropriate for the specific case; channels can be used without fear of clients violating the protocol, and mutexes can be used as an invisible optimization.

It is at this point that the ideas of CSP seem to clash quite strongly with the way that programs are typically written in Go. Rather than having a simple process-oriented architecture where services speak to each other using message passing, an entire service may be hidden from view behind an opaque type. A direct impact of this is that the service can no longer be used directly in an alternation or select statements. Clients that wish to attempt communication with multiple services implemented in this fashion would need to wrap a set of channels around the nicely encapsulated type. In addition, if the server process is limited to a single request at a time, the encapsulation only serves as an inefficient monitor for the shared resource.

However, one of the most interesting aspects of Go is the way that it manages to combine powerful *orthogonal* language features, allowing programs to be written in a way that best suits the problem being solved. User-defined data types and methods make it possible to write programs in an object-oriented style, without requiring everything in the language to be an object. Interface types (and the implicit nature of satisfying an interface) provide a powerful means of abstraction that gives the language a more 'dynamic' feel, while still providing the safety of a static type system. None of these features are as powerful or complete as they are in their more extreme forms: Java's form of object-oriented programming gives the programmer the ability to create rich consistent type hierarchies, while Python's 'duck typing' can be taken to an extreme causing the type system to all but disappear!

The way that CSP-style concurrency integrates with Go is simple and powerful, but very different from what has been widely used before. Many of the features that are considered central to occam style programming, such as the explicit sequential and parallel composition of processes and the barrier-style synchronization that is present at the end of such blocks must be accomplished explicitly in Go programs. It is also difficult to systematically check and verify the properties of a concurrent system, when it is composed entirely of free-standing processes that can be spawned at will.

However, the ideas of CSP lie at the heart of concurrency in Go. Within the documentation, the implementation, and discussions on the mailing lists, concurrent ideas are almost universally described in terms of synchronous channel communications between processes. Attempts to prematurely optimize and use a mutex to share data are discarded and replaced by a solution using processes and channels. Go makes it possible to utilize CSP-style concurrency in ways (and to a degree) that it has not been used before. It also presents a collection of challenges due to the way concurrency interacts with other features, such as interface types, shared memory, and the garbage collector.

3.7 Summary

This chapter presented a catalogue of architectural patterns for CSP-style concurrent computation, and introduced several ways in which these patterns can be implemented using concurrency mechanisms in the Go programming language. Although goroutines and channels are based on the concepts of CSP, they differ sufficiently from the way they are implemented in other languages, such as occam-π and Java with JCSP. These patterns are used throughout this thesis, and this chapter helped to place them in context and highlight their advantages, drawbacks, and limitations when implemented in Go. Collectively, these patterns showcase a variety of ways in which goroutines and channels can be used to express a range of different architectural designs in order to solve a variety of problems.

Chapter 4

Serving Web Content with Dynamic Process Networks

The previous chapter provided an introduction to a variety of structural patterns that are commonly used in Go programming. These include pipelines, process networks, asynchronous result delivery, and various ways to accomplish mutual exclusion. Although these patterns are in no way unique to Go, they demonstrate the ways in which the language benefits from goroutines and channels; a disciplined vocabulary and model for reasoning about the structure and behaviour of concurrent systems.

This chapter details the application of process-oriented structure and design to a well-known problem: serving web requests using HTTP. There are many mature web server implementations that utilize a variety of architectural patterns, but there are few that exhibit process-based CSP-style concurrency. The *webpipes* concurrent web server toolkit [78] is a set of abstractions that make it possible to build web servers by compiling simple feature components in order to generate web responses collaboratively and concurrently.

4.1 Serving Web Content

HTTP is a simple plain text protocol that is defined by RFC1945 [18] and lies at the heart of nearly all communication on the web. A client connects to a web server and submits a request and a list of headers, which contain additional information about the request:

```
GET /hello.txt HTTP/1.0
Accept-Encoding: gzip,deflate
Host: www.example.com
```

The first line of the request indicates the desired action from a fixed set of verbs (`GET`, `PUT`, `POST`, `DELETE`, etc.), the resource that is the target of the action, and the version of the protocol that should be used for the request. Other details of the request are specified in the headers, such as whether or not the client is willing to accept a response that has been compressed or encoded in some fashion. For example, the `Host` header specifies for which host the request is intended, in case the server is configured to respond to requests for more than one host. The server will respond using a similar format:

```
HTTP/1.0 200 OK
Content-Length: 13

Hello World!
```

The response status line indicates the protocol version, the numeric status code of the response, and a text version of the status code. A code of 200 indicates that the request was successful, as echoed by the status text `OK`. Following this, a list of headers can provide additional metadata for the response, such as the date on which the response was generated, or how long the content should be cached by the browser. In this example, the response contains a single header that indicates how long the content of the response is (in bytes). The web browser uses this number to limit the number of bytes that it expects to read, enabling the connection to be re-used for additional requests.

One of the more difficult parts of writing a web server is negotiating this protocol, ensuring that the responses sent by the server can be correctly read and interpreted by the client. Given a library that can parse requests and assist in the construction of responses, a simple web server for static content will take the following steps for each connection:

1. Read and parse a request from the network in order to determine which file is being requested
2. Open the corresponding file on the file system
3. Send the content-type and other appropriate headers to the client
4. Write the contents of the file to the network connection

Each of these stages requires some form of I/O; reading and writing to the client's network connection or interacting with the file system. Any of these operations could

potentially block while waiting for data to be read from the disk or for the network connection to become available. This leads to a natural requirement: a web server must be prepared to handle multiple concurrent requests from different clients. While the server is waiting for the results of an I/O operation for one request, it should be able to begin processing a request from another client.

There are two main approaches to providing this sort of client-level concurrency. A *thread-based* web server handles each request with an independent worker thread that uses normal blocking I/O operations. In the Apache web server, these threads are provided and scheduled by the operating system. Both the YAWS [9] web server, written in Erlang, and the server provided by Go's *net/http* package utilize lightweight processes/threads which are scheduled by the programming language runtime.

An *event-based* server eschews the conventional threading model, opting instead for a single-threaded event loop that may be supported by one or more I/O worker threads. Both Node.js [4] and Lighttpd [8] utilize this approach, albeit with very different implementations. Requests are converted to events as they arrive and added to an event queue. The event loop will process an event from the queue until it encounters an operation that might potentially block, such as I/O. These calls are handled in the background by a worker thread that is specialised to handle multiple requests for asynchronous I/O. In the meantime, the event loop will begin processing another event, and the worker thread will trigger a new event when the I/O is complete, driving the finite state machine for each request.

In this design, the event loop is constantly busy processing requests. By avoiding the thread-per-request model the server avoids the cost of repeated context switches when there are a large number of clients connected to the server, assuming that the event dispatch is faster than switching between threads. This makes event-based servers very efficient for server loads that are not CPU-bound.

There are countless arguments in favour of each of the two approaches [51, 27, 68, 42] for the development of network servers. The flow of control in a thread-based application is easy to follow because it is sequential, but requires synchronization for any shared resources if the threads can be pre-empted by the scheduler. In contrast, an event-based program ensures that only a single handler is being executed at a time. These handlers must be written as fast-executing sequences that are guaranteed not to block, since a long-running handler would prevent any others from being run. As a result, operations such as I/O are typically performed by making a call to initiate the operation, and registering a function to be called when the results of the operation are

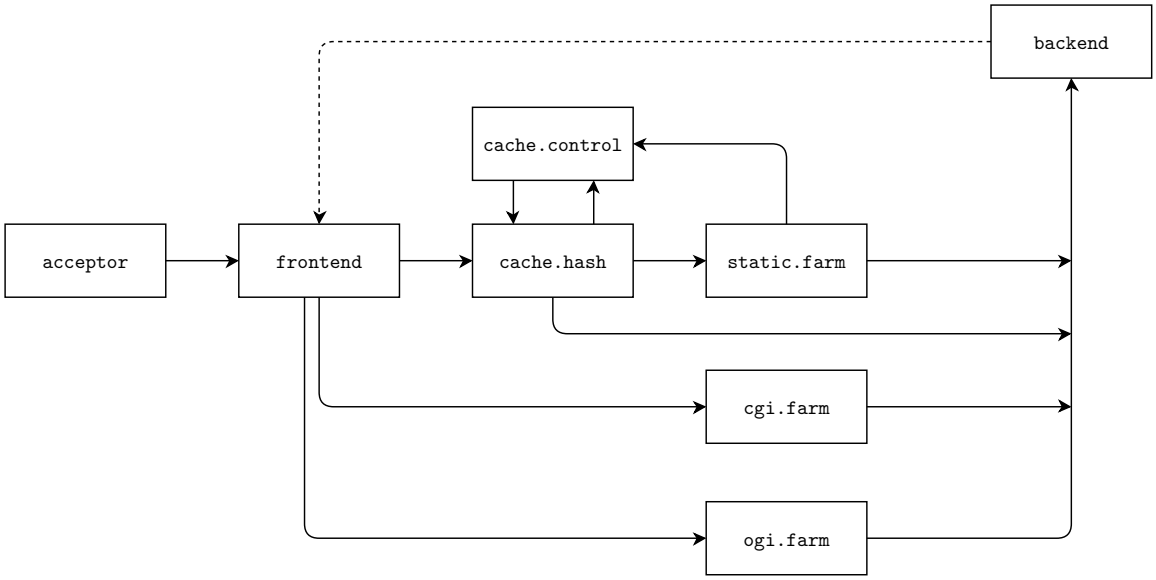


Figure 4.1: The top-level architecture of the occam web server

available. This style of programming (when it is not handled automatically) requires the developer to track the state of the program explicitly.

These arguments persist (and are continuously reiterated in various forms) despite Lauer and Needham having shown a clear correspondence between the two different models [43]. There have also been several attempts to combine the performance of events with the expressivity of threads [77, 44, 69, 36] for network applications, with varying degrees of success.

A third approach is based on the process-oriented style of software development. The occam web server [16] comprises a network of processes that are connected together using synchronous channels. Rather than the connection or request being the basis for concurrency in the system, each request is converted into a message that is sent through a network of processes. Figure 4.1 shows the top-level architecture of this process network.

The `acceptor` process accepts new connections from clients and forwards them as messages to the rest of the server. The `frontend` process receives a message (which includes the network connection) and then reads and parses a request from the network connection. There are several specialised processes (`static.farm`, `cgi.farm` and `ogi.farm`) that maintain a farm of worker processes to handle incoming requests for different types of resources. As these processes are occupied by incoming requests, the farm process will spawn new processes to ensure spare capacity. Requests for static files are first sent to a cache process that can either respond with a cached

response or forward the request to the static file process farm. When the request has been fulfilled, the connection is forwarded to the `backend` process, which determines whether the connection should be re-used for another request or closed.

This illustration of the process network provides insight into the way in which the server functions, particularly when compared to the arcane configuration files that are used to configure most web servers. In addition, the compositional nature of CSP and occam means that any of the processes comprising the server may themselves consist of a network of sub-processes, which could be diagrammed similarly. This is the key motivation behind the *webpipes* toolkit, originally presented in [78], and discussed further in this chapter. It enables individual web handlers to be defined by composing simple bits of functionality in order to achieve a more complex behaviour in a way that is easy to understand. This is particularly advantageous when a web server requires advanced functionality, such as URL rewriting, caching, compression, or other forms of content transformation.

Despite being inspired by the structure of the occam web server, *webpipes* is distinct, embodying an attempt to bring the compositional nature of the process-oriented architecture to web servers written in Go, without *requiring* that the program be written in a process-oriented way. A collection of *webpipes* components can be spawned as a network of processes, but could also be executed using sequential functional composition; the way in which components are executed is independent of the functionality of the component.

The implementation of the toolkit is based on the *net/http* package that is part of the Go standard libraries and is used to handle the low-level details of the HTTP protocol (such as parsing requests and formatting responses). There are several places in which this standard library makes assumptions about the way in which connections are handled, or responses are generated. These points of conflict are interesting because they provide an opportunity to show how two seemingly incompatible concurrent structures can be made compatible through abstraction.

In the following section, we introduce the various abstractions and concepts that are used by the *net/http* package for building web servers, paying specific attention to the way in which such servers handle concurrent requests.

4.2 Concurrent Web Servers in Go

The *net/http* package in the Go standard libraries provides a basic multiplexing web server framework that is capable of handing multiple requests at a time. The server

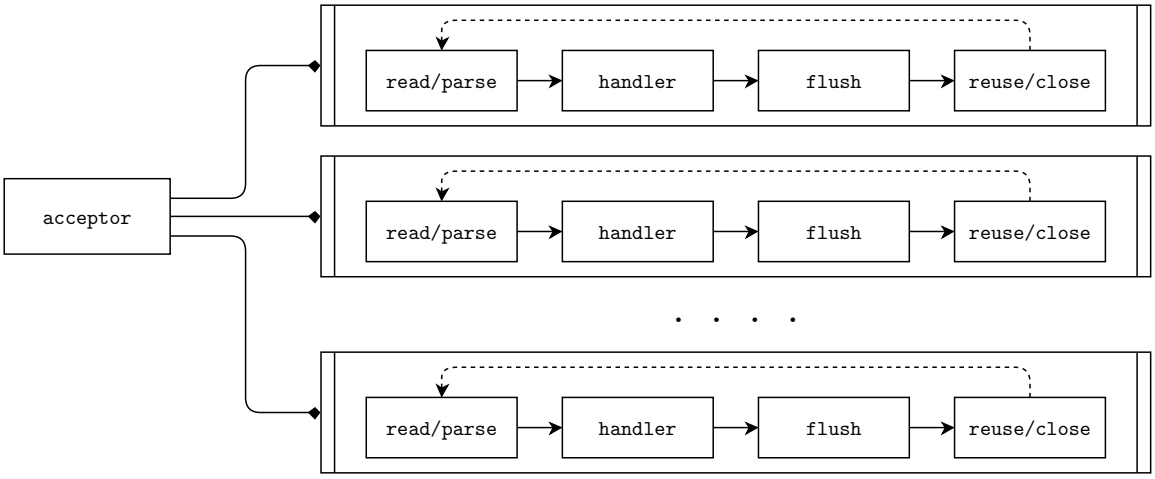


Figure 4.2: Architecture of Go web servers

is thread-based, using goroutines to create an independent thread of control for each incoming connection. When the server is started, it consists of a single goroutine listens for new connections on the server's network socket.

When a client connects, a new goroutine is spawned to handle all communication for that client. This goroutine is able to use normal blocking I/O operations to read, parse, and handle a request from the network connection. For valid requests, a handler function will be invoked within the same goroutine in order to generate an appropriate response. Figure 4.2 shows the structure of these servers: a line ending with a diamond indicates the spawning of a new goroutine, and the arrows within a goroutine indicate sequential execution. Each of the request-handling goroutines is fully independent; there is no communication between requests or with the main server. Because each request is handled in a separate goroutine, any errors that are encountered while processing a request will not affect any of the other outstanding requests.

After a request has been read from the client, it is packed into a `Request` object along with other details about the connection, such as which protocol version was requested, and the headers that are sent by the web browser. This is then passed as a parameter to the `Handler`, which generates the response and sends it on the network connection. If both the server and client support it, the goroutine will then attempt to read another request from the network, otherwise the connection is closed. The majority of these steps are performed behind the scenes by the web server. The application developer only needs to provide the `Handler` that generates responses.

The following code shows a fully functional Go program that is capable of responding to all incoming requests with a static text response:

```
package main

import (
  "io"
  "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/plain")
  w.WriteHeader(http.StatusOK)
  io.WriteString(w, "Hello, world!\n")
}

func main() {
  // register a handler for the root URL and start the server
  http.HandleFunc("/", helloHandler)
  http.ListenAndServe("localhost:8080", nil)
}
```

Within the `main` function, a new handler is registered for the root path[1], and the web server accept/serve loop is started. The `http.ListenAndServe` function will block until the server is terminated.

The handler function itself takes two parameters and returns no result. The first parameter is an `http.ResponseWriter`, an interface that provides a way for a handler to construct and write a response to the client. The second parameter is the request data structure, including the request method, HTTP protocol version, and the headers that were sent by the client's web browser.

The `http.ResponseWriter` interface consists of three methods:

- `Header()` returns a hash table that contains the headers to be sent with the response.
- `WriteHeader(int)` writes an HTTP response line with the given status code and flushes the headers to the client.
- `Write([] byte)` writes data to the connection as part of an HTTP reply.

The handler shown in this example sets the `content-type` header for the response, which indicates that the browser should display the response as plain text content

[1]The web server supports simple text prefixes for choosing a handler based on URL. The longest matching prefix will take preference. Since all paths include the root path, this handler can be used for all incoming requests.

rather than trying to render it as HTML. These headers are then written to the client using the `http.StatusOk` status code, signalling that the request was successful. The message string is then written to the network connection using the `io.WriteString` function, which converts a string into a sequence of bytes and writes these to the response writer.

Since the response writer has a `Write` method, it fulfils the `io.Writer` interface, an abstraction for 'things to which a stream of bytes can be written'. This abstraction follows from the UNIX "everything is a file" philosophy, in which all manner of input/output resources (files, directories, network sockets, and peripheral devices) can all be accessed as simple streams of bytes. This abstraction is surprisingly powerful and permeates the Go standard libraries.

This is in contrast with the philosophy of CSP and occam where "everything is a process" and connections to resources are made using channels. In order to conform to interfaces more commonly expected in Go programs, this thesis will hide those channel interfaces behind the more conventional object/method interfaces used for file input and output.

For example, the compression packages are written to expect `io.Writer` types rather than being tied to files or network streams explicitly. As a result, it is easy to write a handler that compresses the contents of the response, using the *compress/gzip* package to 'wrap' the response writer.

The previously defined handler function can be altered to output the same message, but compress the response for transport:

```
func helloGzipHandler(w http.ResponseWriter, r *http.Request) {
  headers := w.Header()
  headers.Set("Content-Type", "text/plain")
  headers.Set("Content-Encoding", "gzip")
  gzipw := gzip.NewWriter(w)
  w.WriteHeader(http.StatusOK)
  io.WriteString(gzipw, "Hello, world (but gzipped!)\n")
  gzipw.Close()
}
```

This requires the server to set a header indicating that the response has been encoded, so that it can be properly interpreted by the client. In addition, the gzip compression format requires a checksum at the end of the compressed data, so the `GzipWriter` must be closed in order to ensure that this footer is written.

The same overall approach can be used for other types of content transformations. For example, the *encoding/json* and *encoding/xml* packages provide a similar mechanism for marshalling data to an `io.Writer`. In this way, an application developer

can provide an API that communicates using JSON or XML without much additional work.

The structure of this server architecture is quite different from that of the occam web server. Both have acceptor processes, but the Go server lacks explicit front-end and back-end components. Once a handler goroutine has been spawned, it does not communicate with the server or any other handler goroutines; it only interfaces with the client.

As a result, there is no easy way to track the number of requests that are currently waiting to be processed. However, if the server is running on a machine with limited resources, it can be useful to limit the number of connections that are being handled concurrently in order to degrade gracefully before approaching these limits. More importantly, the server must know how many requests are outstanding in order to determine whether or not it is safe to shut down or restart the server.

The process-oriented structure of the occam web server, where client connections are passed as messages between processes, gives a finer degree of control over the flow and rate of requests through the system. This can be accomplished by limiting the number of worker processes that are spawned to handle each type of request, or by introducing buffering between processes. In contrast, the goroutine-per-connection architecture in the Go web server provides much better isolation, ensuring that problems encountered while handling a request do not impact the rest of the system.

These two approaches can be combined with little effort, by creating a collection of worker processes and then creating a handler that forwards requests to the worker pool. Figure 4.3 shows multiple handler goroutines connected to a pool of worker processes. As requests arrive, the handler goroutine will build a new `Connection` object, which bundles the response writer, the request, and a channel that will be used to signal when the request has been handled.

```
type Connection struct {
  rw   http.ResponseWriter
  req  *http.Request
  done chan bool
}
```

The request handling loop can be written as a separate function that takes a handler function and a channel on which to receive new requests.

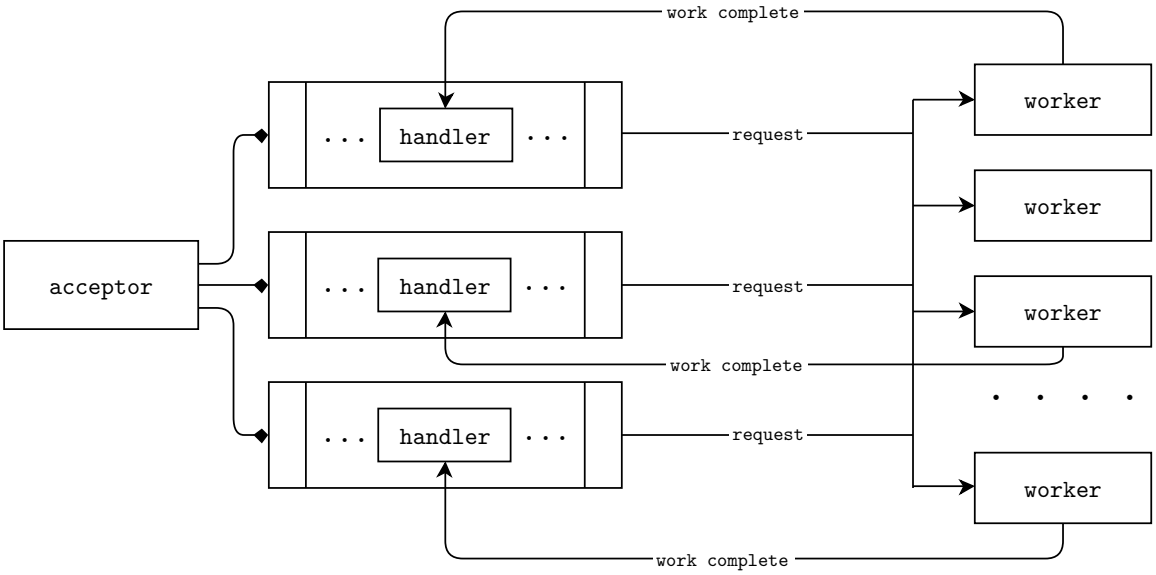


Figure 4.3: Network of collaborative web handlers

```
func handlerLoop(handler http.HandlerFunc, queue chan Connection) {
  for conn := range queue {
    handler(conn.rw, conn.req)
    conn.done <- true
  }
}
```

New worker pools can be created using a factory function that takes a handler function and a number of workers to spawn and returns a new handler function that takes care of packing the request into a `Connection`, sending it to the work queue, and waiting for the response.

```
func handlerServer(numWorkers int, handler http.HandlerFunc) http.HandlerFunc {
  queue := make(chan Connection)
  for i := 0; i < numWorkers; i++ {
    go handlerLoop(handler, queue)
  }

  return func(w http.ResponseWriter, r *http.Request) {
    done := make(chan bool)
    queue <- Connection{w, r, done}
    <-done
  }
}
```

It is important that the handler function waits for the response from the worker processes. The connection-handling goroutine that is used by the web server expects

this; once the handler returns, the server will attempt to read a new request from the connection.

With a small amount of additional work, it is possible to convert a handler from the unbounded goroutine-per-connection used in the Go web server to the process-oriented style used by the occam web server. Underneath there will still be one goroutine for each connection, blocked waiting for a response from the handler. The process-oriented style makes it easier to control the flow (and rate) of requests through the server.

4.3 Collaborative Responses

The previous section provided an introduction to the types of web servers that are typically constructed using the Go standard libraries. These servers handle incoming connections in independent goroutines using a series of sequential procedure calls. This helps to ensure the server can handle many concurrent requests, while preventing issues with one request from affecting other connections, but makes it more difficult to control the flow of requests through the server. Goroutines and channels make it easy to adapt this style of concurrency to one that is closer to the occam web server, where requests are sent as messages through a network of handler processes.

A typical web handler examines the incoming request in order to discern which resource is being requested and what options are supported by the requesting client. The handler will then generate and send an appropriate set of headers for the response, followed by actual contents of the requested resource. In reality, the process of properly generating this response requires quite a bit of work in order to negotiate the various extensions and options that are used by web servers and clients. The previous section illustrated an example handler that supported compression of HTTP responses.

A problem with this handler is that it is monolithic and there is no separation between the different concerns. The code to implement compression is intermingled with the code that generates the response. This might not be a problem if there are just one or two features, but the reality of the modern web is that servers need to be able to support a wide range of configurable functionality. Most servers have solved this by providing a way to create modules or plug-ins that can intervene in the connection handling process at various pre-defined stages.

For example, the Apache web server contains hooks for pre-connection, post-head-request, translate-name, check-authentication, and many others. This makes it

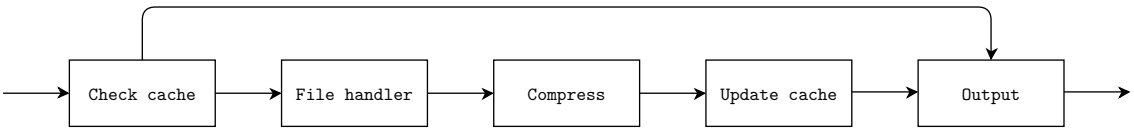


Figure 4.4: Network of collaborative web handlers

possible to build extensions that are very powerful and capable of changing almost any aspect of the server's behaviour. Unfortunately, this power comes at a cost; these plug-ins are notoriously difficult to configure. Writing such a plug-in requires knowledge of the detailed sequence of stages and hooks, or the inner workings of the web server if the desired behaviour does not fit the fixed set of patterns. From the user perspective, it can also be impossibly difficult to determine how a given configuration might actually affect the handling of requests, or how multiple plug-ins might interact.

The occam web server architecture showcases another sort of design, where the response generation is itself described as a network of processes, such as the pipeline shown in Figure 4.4. This pipeline follows the UNIX philosophy, where simple components are connected together in order to perform a more complex task. The illustration of the components also serves as a description of the behaviour of the handler, requiring only a limited understanding of the implementation of the components.

Most web servers assume that only one web handler will be responsible for generating the contents of a response. This simplifies the design of the server, since the handler can be given direct access to the client's network connection and can begin sending the response as soon as possible. Unfortunately, this assumption also makes it difficult to build distinct features that can be composed together. Once a handler has started writing to the network connection, no other handler will have a chance to change the response.

The Go standard libraries take a different approach, providing only *indirect* access to the network through the `ResponseWriter` interface. In normal situations, this response writer will be connected to the network socket, but the interface is intentionally more general. The indirection makes it possible to provide a custom implementation of the interface that alters the response before it is actually sent to the client.

For example, the following function will take a web handler function as a parameter and return a new handler function that compresses the output of the original handler

before it is sent to the client. This is done by 'wrapping' the response writer with a custom implementation and then passing the wrapped version to the original handler:[2]

```
func gzipHandler(orig http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Encoding", "gzip")
    gz := gzip.NewWriter(w)
    wrappedResponseWriter := NewGzipResponseWriter(gz, w)
    orig(wrappedResponseWriter, r)
    gz.Close()
  }
}
```

For each handler that is passed into this factory function, a new anonymous function is created and returned as a replacement handler. Using this function, an existing handler can be converted into one that compresses the response without requiring any changes to the original handler:

```
http.HandleFunc("/gzippedhello/", gzipHandler(helloHandler))
```

This technique is used by the Go standard libraries to implement a `TimeoutHandler` wrapper function. This can be used to wrap a handler and limit the amount of time that a request can be outstanding before it is aborted and a timeout response is returned to the client. Using this it is possible to gracefully degrade the system when it is under load without having to change the way that each of the handlers is implemented.

It is possible to compose multiple behaviours in this way. The following example shows a handler that will send an error message if the request takes more than 15 seconds to complete, will instruct the browser to cache results for up to three days, and compress any text content that is served by a static file handler.

```
handler = TimeoutHandler(time.Second*15, "Response timed out",
  CacheHandler(time.Day*3,
    GzipHandler("plain/text",
      FileHandler("static"))))
http.HandleFunc("/static", handler)
```

This composed version of this handler is easier to understand than an equivalent monolithic version. Each stage of response handling can be defined independently

[2]The implementation of `NewGzippedResponseWriter` has been omitted; it returns a `ResponseWriter` object that delegates `Write` calls to the gzip writer and fulfils all other methods using the original `ResponseWriter` object.

and then the stages can be combined to achieve the desired behaviour without the various stages needing to be aware of each other.

4.4 Streaming Web Components

The previous section details one approach to building web handlers that are composable. By creating a custom `ResponseWriter` implementation and wrapping an existing handler, it is possible to alter the response as it is being generated, but before it has been written to the client. The result is not concurrent, and lacks a clear thread of control for the generation and transformation of responses. In fact, this style of handler is reminiscent of the callback-style handlers that are written in some event-based web servers.

In addition, it is not always obvious how a handler should be implemented in order to achieve the desired behaviour: interaction with the content must be done in the response writer implementation, while a wrapper function must be written to check the request and inject the new response writer. Because of the way the components interact with each other using functional composition, the order of the handlers is therefore logically reversed. The compression handler must begin executing *before* the content is generated in order to ensure that the compression writer is properly injected. This is the opposite of how compression is done in UNIX stream processing pipelines, where the output of the content generation is the input to the compression program.

This is the point where the `ResponseWriter` interface, despite being more than capable for most situations, seems to constrain the design of the system. In order to build handlers that can be composed in the stream processing style, handlers must each be given an opportunity to examine and possibly alter the response before it is sent to the client.

This can be provided by an extension of the interface that stores the contents of the response in memory rather than sending them immediately to the client. Existing handlers can use this new type without any alterations, and collaborative handlers can be written to take advantage of the ability to examine and change the response. A basic implementation stores the status code, the contents of the response, and the original response writer that is connected to the network connection.

```
type StoredResponseWriter struct {
  StatusCode int
  Content    *bytes.Buffer
  Original   http.ResponseWriter
}

func (rw *StoredResponseWriter) Header() http.Header {
  return rw.Original.Header()
}

func (rw *StoredResponseWriter) Write(data []byte) (int, error) {
  return rw.Content.Write(data)
}

func (rw *StoredResponseWriter) WriteHeader(code int) {
  rw.StatusCode = code
}
```

With a normal HTTP handler function, any calls to `WriteHeader` would flush the status code and headers to the network, and this is precisely the behaviour we would like to avoid. Instead, this implementation stores the status code so it can be utilized only when it is finally time to send the response to the client. Any calls to the `Write` method will be delegated to the `bytes.Buffer`, an in-memory buffer that satisfies both the `io.Reader` and `io.Writer` interfaces. Because the `StoredResponseWriter` fulfils the `ResponseWriter` interface, handlers can be given this substitute type without any changes.

A version of the compression filter that uses the `StoredResponseWriter` can be written in a more natural way:

```
func gzipHandler(w *StoredResponseWriter, r *http.Request) {
  headers := w.Header()
  if headers.Get("Content-Type") == "text/plain" {
    headers.Set("Content-Encoding", "gzip")
    buffer := new(bytes.Buffer)
    gzipw := gzip.NewWriter(buffer)
    io.Copy(gzipw, w.Content)
    gzipw.Close()
    w.Content = buffer
  }
}
```

Unlike the previous version of this handler, which could make decisions only based on the request headers, this version is able to examine the current state of the response in order to determine its behaviour. In this case, only those responses that have `plain/text` content will be compressed. The compressed data is written into a

separate buffer that replaces the original content buffer in the response writer data structure.

Combining these handlers is easy, due to the way in which they are decoupled, with the order of execution having the more logical stream processing order:

```
func helloGzipHandler(w http.ResponseWriter, r *http.Request) {
  srw := NewStoredResponseWriter(w)
  helloHandler(srw, r)
  gzipHandler(srw, r)
  srw.Commit()
}
```

The `Commit` method is needed to flush the content from the `StoredResponseWriter` to the original response writer, sending the response to the client. Both the creation of the stored response and the commit could be hidden by the server framework, making it easier for the developer.

A clear problem with this approach is that it requires the entire contents of the response to be present in memory at once. Although this is probably reasonable for small responses, it would be wasteful for even medium sized files under high load. In addition, the response must be stored and passed to each handler in turn, preventing concurrency within the request generation process itself.

Rather than storing the entire contents of the response, it would be sufficient to store an abstract representation of the contents. For example, the response writer could contain the reading end of a stream of bytes on which the contents can be read, rather than the raw data. Filters could then insert themselves into this content stream, making it possible to do on-the-fly transformations in a style that is much closer to that of UNIX pipelines. The server could then read from the end of this channel and send the data to the client as it is generated.

For the moment, assume a `helloHandler` component whose output is connected to a `gzipHandler`. The handlers must still be executed sequentially: re-ordering the components so that compression handler runs before the content is ever generated doesn't make much sense. However, the writing to (or filtering of) the content stream should not be performed during the execution of the handler. Doing so would require the entire contents to be buffered in memory. In order to continue with this approach, it is necessary to alter the semantics of web handlers to introduce some explicit concurrency.

Handler functions generally consist of two main stages: setup and content generation. During the setup stage, a handler examines the request, sets any headers that are needed for the response, and prepares to generate or filter the content. For

a handler that is serving static files from the file system, this involves checking to see that the file resides on disk, opening the file, and possibly examining the type of the content so that the appropriate headers can be added to the response. During the content generation phase, the handler reads the data from this file and writes it to the response stream. This is something that can easily be accomplished in a separate goroutine.

The *webpipes* toolkit replaces the standard `http.ResponseWriter` interface type with the `StreamedResponse` type, which is more suited to collaborative response generation. Rather than providing direct access to the underlying network connection, but has methods that allow a handler to register as either a 'content source' or a 'content filter'.

A content source is responsible for generating the initial contents of the response, serving the same role as the monolithic web handlers seen previously in this chapter. `StreamedResponse` provides a `ContentSource()` method that creates a new pair of channel ends, stores the reading end, and returns the writing end to the caller.

```
func (res *streamedResponse) ContentSource() io.WriteCloser {
  if res.body != nil {
    // A content source has already registered
    return nil
  }

  reader, writer := io.Pipe()
  res.body = reader
  return writer
}
```

The `helloHandler` can be written as a content source using the `StreamedResponse`.

```
func helloHandler(res StreamedResponse, req *http.Request) {
  res.SetHeader("Content-Type", "text/plain")
  writer := res.ContentSource()

  generate := func() {
    io.WriteString(writer, "Hello, world!\n")
    writer.Close()
  }
  go generate()
}
```

In contrast, a filter handler needs access to the content that has already been written as well as a way to write the transformed content to the stream, provided by the `ContentFilter()` method. This method returns the 'dangling' read end of the

channel that was provided to the source (or in fact another filter), creates a new pair of channels, and stores the reading end.

```
func (res *streamedResponse) ContentFilter() (io.ReadCloser, io.WriteCloser) {
  if res.body == nil {
    // There is no content source yet, so cannot filter
    return nil, nil
  }

  // Create a new reader/writer pair
  var reader io.ReadCloser
  var writer io.WriteCloser
  reader, writer = io.Pipe()

  // Swap the dangling reader with the new one
  reader, res.body = res.body, reader

  return reader, writer
}
```

The compression handler can be written as a filter in the same style where the setup of the response is separated from the transformation of the content, which is performed in another goroutine:

```
func gzipHandler(res StreamedResponse, req *http.Request) {
  if res.GetHeader("Content-Type") == "text/plain" {
    res.SetHeader("Content-Encoding", "gzip")
    reader, writer := res.ContentFilter()
    gzipw := gzip.NewWriter(writer)

    filter := func() {
      io.Copy(gzipw, reader)
      gzipw.Close()
      writer.Close()
    }
    go filter()
  }
}
```

Figure 4.5 depicts the way that these two components can be combined and executed. The server passes the request (and a response writer) to `helloHandler`, which registers a new content source and spawns a new goroutine that will actually generate the contents of the response. The request is then passed to the `gzipHandler` component, which registers itself as a content filter. It spawns a goroutine that reads data from the content stream and then outputs it in a compressed form to the writer.

The execution of these two components results in the dynamic creation of a pipeline of goroutines connected together with a channel, shown by `generate` and

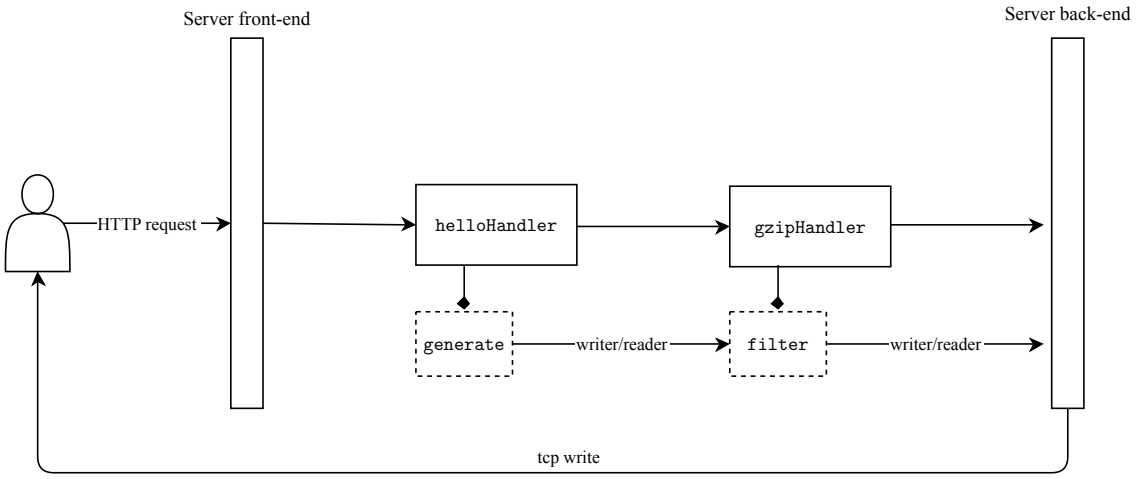


Figure 4.5: A server with two components, showing the spawning of goroutines for generating and for the transformation of the content stream.

`filter` in the figure. The filter is connected to the server back-end using a separate channel with a dangling read end. The server front-end performs the set-up, including spawning the connection goroutine and injecting the response writer, while the back-end read from the final channel read end and streams the response to the network connection.

The *webpipes* library uses the `io.WriteCloser` and `io.ReadCloser` interfaces, rather than utilizing channel types directly. The primary reason for this is that these interfaces make it possible for web handlers to use the existing code in the standard libraries, which make heavy use of these interfaces. A more subtle reason for utilizing the interface abstraction is that these methods include a mechanism for propagating errors between different handlers; this functionality is not present when using direct channel communications in Go. For example, the user might cancel the loading of a page in their web browser, causing the network connection to close. This will cause the `Write` method to terminate with an error when the first write is attempted to the network connection. This error will be propagated backwards through the content pipeline, effectively halting the generation and transformation of content. The same is difficult to accomplish using a single set of channels, but could be implemented using pairs of channels, albeit in a less straightforward way.

Regardless of the way in which the medium for content delivery is implemented, a streamed response will consist of a sequence of channels, each with a writing end and a reading end. A content source will be given the writing end of the first channel.

As a content filter is added, it is placed between the reading end of one channel and the writing end of the next. The source writes the initial contents of the response, which is then read and transformed in turn by the filters. The server can then read the data from the end of this content pipeline, and send the response to the client in accordance with the HTTP protocol. This makes it possible to build web handlers so that they can be composed together in order to generate responses collaboratively without requiring an unintuitive ordering of their execution.

These handlers were written explicitly in order to illustrate how the content generation stages of request handling can be done in separate goroutines. The same could be accomplished implicitly by the web toolkit. Rather than the `ContentSource` and `ContentFilter` functions returning readers and writers, they could take the content generation closures as parameters. The toolkit could then handle the spawning of multiple goroutines behind the scenes.

This would make these handlers look much closer to the event-based handlers that are written using frameworks like Node.js, for example. The difference is that these content generation stages can all be executed in parallel. Further examples will continue to use the explicit style of spawning new goroutines in order to highlight this concurrency.

So far, we have shown three different ways of writing web handlers in Go. Monolithic handlers generate a response and write it directly to the network connection. Although it is possible to alter this sort of handler by wrapping the handler function in order to inject a custom `ResponseWriter` implementation, such handlers are cumbersome to implement and go against the stream processing view of handling web requests.

This section presented the `StoredResponse` type, which can be used to construct and hold responses in memory, giving multiple handlers the opportunity to alter the response before it is sent. In contrast, the `StreamedResponse` uses readers and writers connected in pairs to enable the streaming generation and transformation of responses. Both make it possible to write web handlers that can explicitly examine the response that has been generated in order to determine if and in which way it should be altered.

The next section shows how these techniques apply to a variety of different component types.

4.5 Defining Components

Throughout this chapter, the terms *component* and *handler* have been used interchangeably to refer to 'things' that are involved in the handling of a web request. The *webpipes* package explicitly defines a `Component` interface type that encompasses these. This type contains a single method called `HandleHTTP` that takes a `StreamedResponse` and an HTTP request.

```
type Component interface {
  HandleHTTP(res StreamedResponse, req *http.Request)
}
```

This type is intentionally quite general, enabling it to cover a range of functionality within a single interface that supports collaborative response generation. Components will fall into one of three categories. A *content source* produces the initial contents of the streamed response by reading data from a file on disk, or by producing some form of dynamic content. A *content filter* reads the contents of the response in order to process, filter, or transform it before it is sent to the client. A third type of component, *pipe*, might examine or alter a request but does not necessarily require access to the content stream itself.

The *webpipes* package defines each of these component types as an explicit function type:

```
type Pipe func(StreamedResponse, *http.Request)
type Source func(StreamedResponse, *http.Request, io.WriteCloser)
type Filter func(StreamedResponse, *http.Request, io.ReadCloser, io.WriteCloser)
```

Each component receives the streamed response writer and the request as the first two parameters. The `Source` and `Filter` types also receive the readers and writers that are needed to serve their purpose. This enables the programmer to write components without explicitly needing to request content readers and writers from the response. The *webpipes* package defines a `HandleHTTP` method for each of these types that manages the allocation of the content readers and writers, allowing the developer to focus on the application logic of their components.

The four different component types, `Component`, `Source`, `Filter` and `Pipe`, provide the means to implement almost any type of web-handling functionality. The `helloHandler` component seen previously in this chapter could be used as it has been written, since the function matches the type signature for a `Pipe` component.

Alternatively, it could be changed into a source component by removing the call to `ContentSource` and adding the `WriteCloser` as a parameter to the function.

4.5.1 Decision-making Components

Not all components will apply to every request. For example, the `gzipHandler` should probably not compress binary responses, because they are typically already compressed in some way. The HTTP simple authentication protocol requires clients to specify credentials using headers. The server must examine and validate this information before proceeding with the request. If the credentials are not valid, the server will send a special response that causes the web browser to prompt the user for authentication.

It is not immediately obvious how this can be accomplished within a pipeline of processes, since the authentication component may need to abort the pipeline in order to prompt for authentication (or to deny access). A similar issue exists for caching components (such as the one shown in Figure 4.4), which need to be able to serve a response from the cache invoking the components normally responsible for generating the response. Such components need to be able to change the way a request is routed through the system at execution time.

Consider a more general case: given a predicate function on a request, a component needs to take one of two different actions (which are themselves implemented as components). This can be defined as a `Pipe` component that takes the predicate along with two components.

```
type Predicate func(*http.Request) bool

func IfThenElse(pred Predicate, pass Component, fail Component) Pipe {
  return func(res StreamedResponse, req *http.Request) {
    if pred(req) {
      pass.HandleHTTP(res, req)
    } else {
      fail.HandleHTTP(res, req)
    }
  }
}
```

This can then be used to implement a poor-man's authentication that checks to see if the query parameter 'secret' has a certain value:

```
func HasSecretKey(req *http.Request) bool {
  queryValues := req.URL.Query()
  param, ok := queryValues["secret"]

  return ok && paramHasValue(param, "opensesame")
}
```

These can all be put together to create a handler that can decide which components should be used to generate the response:

```
  public := TextSource("Nothing to see here")
  private := TextSource("Secret information is here")
  http.Handle("/", IfThenElse(HasSecretKey, private, public))
```

It is even possible to chain conditional components together. The `IfThenElse` function returns a component which will invoke one of the two parameter components, either of which could be another conditional component.

4.5.2 Stateful components

The simple components shown thus far are all stateless and produce the same response every time. This is generally a desirable property because these responses are simple to produce and can be cached by the web server or browser. However, many applications need to track some state, normally in a persistent storage layer such as a database, but this data could also reside within the web server memory itself. For example, a web page might include a hit-counter that is incremented each time the page is requested by a client. In many cases such functionality is implemented by a CGI [59] script that is then invoked by the web server.

The same can be implemented as a data server that runs within the server itself, as a source component. Implementing this as a `Source` that uses a data server goroutine to carry state is straightforward. The handler will forward messages to the data server in order to fetch the text of the counter and wait for the response. Each request to the counter will be serialized, but that is to be expected in order to keep an accurate count of the number of page visits.

The server goroutine stores the hit counts in a hash map that is keyed by path and repeatedly accepts requests to increment and return the counter value for a given path:

```
func HitCounterServer(in chan *http.Request, out chan int) {
  hits := make(map[string]int)
  for {
    req := <-in
    count := hits[req.URL.Path] + 1
    hits[req.URL.Path] = count
    out <- count
  }
}
```

In this handler, the counters reside only in the data server and will be reset each time the server is restarted. The impact of such an issue could be reduced by periodically flushing the current values of the counters out to disk and loading the values from disk when instantiating a new counter. Because the data server accepts requests via channels, it is easy to add a second case to the `select` statement to take a periodic action such as this.

```
func HitCounterServer(in chan *http.Request, out chan int) {
  // ... previous initialisation ...
  ticker := time.NewTicker(time.Second * 60)
  for {
    select {
    // ... previous case ...
    case <- ticker.C:
      writeToFile("counters.txt", hits)
    }
  }
}
```

The `NewTicker` function spawns a new goroutine that sends the current time on a channel at a given interval, in this case 60 seconds.

Like the data servers introduced in Chapter 3, this implementation obscures the concurrency that is happening behind a sequential source component. This may seem like a bad thing, but recall that one of the problems with our initial definition of HTTP handlers was that functional components needed to be aware of each other (and how they were implemented). In general, there is no way to see from looking at a handler whether it might be backed by a data server goroutine. The `Component` abstraction used by the *webpipes* toolkit does not mandate a particular type of concurrency; persistent concurrent processes can happily live alongside other straightforward components.

A rudimentary caching system could be built by combining a stateful component with a conditional component. The cache itself could be managed by a data server goroutine that accepts lookup requests to check if the response to a given request is

cached, and update requests for when a new entry is able to be cached. The following wrapper function takes a component and returns a new `Pipe` component that will cache all responses generated by the original component, and serve the cached result on subsequent requests. This new component can be used in multiple web handlers and each will share the same cache server goroutine behind the scenes.

```
func SimpleCache(original Component) Pipe {
  in := make(chan Request)
  out := make(chan Source)
  go CacheServer(in, out)

  // Make a filter that will capture a response and update the cache
  // that is attached to the channel 'in'
  updateFilter := CaptureAndUpdateCache(in)

  return func(res StreamedResponse, req *http.Request) {
    in <- GetRequest{req}
    cachedSource := <-out
    if cachedSource != nil {
      cachedSource.HandleHTTP(res, req)
    } else {
      original.HandleHTTP(res, req)
      updateFilter.HandleHTTP(res, req)
    }
  }
}
```

The `CaptureAndUpdateCache` component used here is a *filter* that clones the response headers and then reads the content and stores it in a buffer. It combines these into a source that can repeatedly produce the cached response, and sends this in an update message to the cache server.

Given a dynamic source that generates a different response each time it is invoked, it can be cached by passing it as a parameter to the `SimpleCache` factory function.

```
http.Handle("/", SimpleCache(DynamicSource))
```

The cache is keyed by URL, so each request to a different URL on the server should generate a different response. However, repeated requests for the same URL should always return the same response. A more comprehensive cache would also store the headers and status code that can be safely cached.

4.6 Combining Web Handling Components

The previous sections introduced the `Component` interface type, along with the three different specialized component types: sources, filters, and pipes. Several example components were constructed using this framework in order to illustrate the different ways in which tasks might be accomplished using composable components that collaborate. Higher-order components provide an easy means for implementing rich functionality. The components that were utilized in these examples were either single components, conditional components, or multiple components were manually called in sequential order. This section illustrates how components can be composed, with varying concurrency properties that can impact the performance and behaviour of the web server.

Where possible, the *webpipes* toolkit uses the existing web server infrastructure from the Go standard libraries. This server spawns a new goroutine for each incoming client connection. Since each request is already being executed in a separate goroutine, it is sufficient (in many cases) to invoke components in sequence. Rather than forcing one to do this manually for each web handler, a list of components can be converted into a static pipeline component using the `Pipeline` function.

```
func Pipeline(components ...Component) Pipe {
  return func(res StreamedResponse, req *http.Request) {
    for _, stage := range components {
      stage.HandleHTTP(res, req)
    }
  }
}
```

This factory function is a variadic function, meaning it can take any number of components as arguments. The body of the function can then access this list of functions as an array. For each component that is passed as an argument, the `HandleHTTP` method is invoked and the request and response are passed as parameters. The result is a new component that inherits the properties the components it comprises; if the pipeline contains a source than the resulting pipeline is also a source. Since the pipeline is a component, it can also be used in all of the higher-order components that were defined in the previous section.

```
http.Handle("/other",
  SimpleCache(
    Pipeline(DynamicSource, GzipFilter)))
```

The `Component` type only defines the interface for interacting with components; it doesn't specify anything about how the components can be connected together or how they are actually executed during the handling of a client connection. This means that components can be used in process networks that give the developer more control over the flow of requests through the system. The `ProcPipeline` factory is similar to the `Pipeline` function, but spawns a single goroutine for each component in the parameter list and connects these goroutines together using channels:

```
type Msg struct {
  req  *http.Request    // the incoming request
  res  StreamedResponse // to construct the response
  done chan bool        // channel to indicate request handling is done
}

func ProcPipeline(components ...Component) Pipe {
  in := make(chan Msg)
  pipe := in

  // spawn a server for each component
  for _, stage := range components {
    next := make(chan Msg)
    go componentServer(pipe, next, stage)
    pipe = next
  }

  // final stage to 'finish' each request
  go func() {
    for msg := range pipe {
      msg.done <- true
    }
  }()

  return func(res StreamedResponse, req *http.Request) {
    // inject the request/response into the process network
    done := make(chan bool)
    in <- Msg{req, res, done}

    // wait for the message to emerge before returning
    _ = <-done
  }
}
```

At the boundary to the resulting process network there will still be one goroutine for each request that is waiting, but only one request will be sent through the network at a time.

At first glance, it may seem undesirable to add a process network to an already concurrent web server, since it may form a bottleneck. However there are many things that are simple to accomplish in a process network (since we are not limited in the

number of goroutines we may use) that are more difficult when there is a 1-to-1 correspondence between the number of threads/goroutines and the number of requests. The `select` statement combined with component servers accepting messages on channels makes it trivial to handle time-outs gracefully. Non-blocking sends can be used to reject requests that cannot be served immediately, possibly serving a static version of the content instead of the fully dynamic version.

More importantly, the `Component` abstraction makes it possible to mix freely process networks, pipelines, and any other sort of component implementation.

4.7 Summary and Evaluation

In this chapter, we have presented a compositional architecture for constructing processing pipelines. This architecture was demonstrated through the `webpipes` toolkit, a framework for constructing component-based web servers using the Go programming language. This framework makes use of a series of type and interface definitions that allow multiple components to participate in the generation of responses. This is accomplished by decoupling the generation of response headers from the generation of the actual content of the response. This toolkit encourages the use of simple components that are strung together to create more complicated behaviours.

The behaviour of web servers constructed using the `webpipes` toolkit is easy to understand; often there is a direct correlation between the structure of the code used to create a server and the definition of that server's behaviour. The same might be accomplished in mainstream web servers through the clever use of domain specific configuration languages, however we believe the component model used by the `webpipes` toolkit enables a more natural order of components.

This work is the product of a case study in the ways that disciplined models of concurrency can influence concurrent software design. There are countless concurrent web servers, but none that are able to combine the compositional nature of process networks and collaborative response generation pipelines with explicit concurrency. Throughout the design process we have restricted ourselves to a model of concurrency where components communicate with each other solely via message passing over explicit channels. This sort of collaboration between components is easy when the response is generated and processed in memory, but this approach is inherently sequential. Streaming responses make it possible to create networks for collaborative response generation that can execute multiple stages concurrently. These process

networks are able to share resources and better regulate the flow of requests through the system.

This design was heavily inspired by the process-oriented occam web server, but differs in some important ways. It is not important that a *webpipes* component be written as a server process, so long as it satisfies the `Component` interface. Given a list of components, it is easy to spawn a new process pipeline consisting of those components. Alternatively, each stage can be executed in sequence using standard procedure calls. Regardless of the component architecture, as the request travels through the system a new pipeline of processes will be dynamically constructed to handle the generation and filtering of the contents of the response. These content pipelines are concurrent, efficient, and can correctly handle the premature termination of a request.

This architecture is not an immediately obvious solution to serving concurrent web requests. By intentionally considering the ways in which a web server could be built using a collection of concurrent processes, we were able to evolve the design and find an appropriate level of abstraction that is surprisingly flexible and powerful.

Through the development and evaluation of the `webpipes` toolkit, we have identified many opportunities that require further investigation. There is some overhead introduced by the content reader/writer system that can negatively impact the performance of the toolkit. It would be worth investigating a design that removes the explicit use of content readers and writers by providing a higher level of abstraction for the registration of filter components. This abstraction could remove an additional level of copying and synchronization from the toolkit and would be much closer to the way Go programs are normally written.

The request and response objects are mutable values that are passed between multiple components. It is possible for a component to retain a reference to some part of these objects, creating an unintentional sharing among the goroutines in the content generation pipeline. This problem might be alleviated by introducing a concept of ownership to the type system, perhaps by utilizing some form of linear types [70] or uniqueness typing [30]. This would allow us to specify at the language level that only one reference to an object should exist at any time. The Rust[31] programming language has support for several different reference types that support 'borrowing' and can be used to enforce rules of this sort. As previously discussed in Chapter 3, occam-π has well defined semantics for mobile data type that help to address these issues [73, 14].

This chapter shows how CSP-style concurrency and process-oriented systems can be used as a lens for re-imagining concurrent application development. The following chapter takes a different approach, showing how the CSP process calculus can be used to assist in the iterative development and refinement of a concurrent system.

Chapter 5

Building a Concurrent File Server

The previous chapter presented the development of a new concurrent web server architecture. This architecture was created by considering the problem in terms of process-oriented systems based on CSP-style concurrency. The resulting *webpipes* framework is able to utilize this disciplined model of concurrency in order to manage the asynchronous nature of handling web requests, control the flow of requests through the system, and coordinate between multiple requests in a safe way. Web servers written in this way are more explicit and easier to understand than their traditional counterparts.

Operating systems allow multiple users and programs to share the computing resources on a single machine. They also serve as an intermediary between programs and the physical computing hardware, such as persistent storage devices. In the UNIX family of operating systems, disk access is accomplished by interacting with named files and directories through a hierarchical file system, using system calls. This enables applications to store and access data using simple operations (open a file, read, write, create directory, etc.) without having to worry about where or how the data is stored on the device.

Although a large amount of effort is normally taken in designing an operating system to isolate processes from each other, the file system is a mutable resource shared by all running processes. As a result, the system is subject to the same sorts of common pitfalls that arise in a concurrent environment with shared resources. Due to the relatively slow access times of storage devices, file systems make heavy use of caching and sharing of data in order to improve performance. With so many user programs accessing the file system at the same time, care must be taken to ensure that a program is not able to corrupt the state of the file system or the data stored on disk through normal operations.

In this chapter we design and implement a file system for a microkernel-based operating system. Beginning with a straightforward sequential implementation, we decompose the system into distinct interacting components, using the CSP [38, 60] process algebra and the FDR3 [67] model checker to reason about and explore the properties of (and the interactions between) these components. This shows the interplay that is possible between conventional software design, formal methods and tools for model checking in the development of concurrent applications. The resulting architecture is then used to create an implementation of a concurrent file system using the techniques that have been illustrated throughout this thesis.

5.1 Background

In this section we introduce the UNIX file system model, which is used throughout this chapter, including some terminology and common data structures. We then go on to describe two different implementations of this model: one that is highly concurrent and uses explicit locking mechanisms, and a sequential version that is designed to be simple and easy to understand. Throughout the rest of the chapter we develop a third implementation that introduces concurrency to the file system without the need to resort to explicit locking mechanisms.

5.1.1 UNIX file system

UNIX provides access to persistent storage devices through a single hierarchical file system. Each directory can contain files (which contain raw data) and other directories. This hierarchy is a graph with a single root directory.

Each file comprises metadata, stored in a data structure called an *inode*, and the actual content of the file. This metadata includes the owner of the file, the access permissions, the size of the file, and other similar fields. More importantly, the inode contains a list of the data blocks where the contents of the file may be found on the disk device. Although the system call API refers to file system entries using path names, such as */var/log/syslog*, the metadata does not contain this name or any other identifier. Files have an existence that is independent of their names; in fact, files can exist without having a name if they are opened and then unlinked from the directory in which they were created.

Instead, each directory contains a list of entries that map names to inode numbers, describing the child entries of the directory. In order to convert a path name into an inode, the file system must perform a series of lookups for each directory in the path.

For the path given above, the root directory must be searched for an entry called *var*, then that directory searched for an entry called *log*, and finally that directory searched for a file called *syslog*.

This method decouples the concept of a file from the name used to access it. The UNIX file system model takes advantage of this by allowing an inode to be accessible by several different paths, simply by having the same inode as a child of multiple directories. In order to track this, each inode also contains a count of the number of directories that link to it. This count is used to determine at what point a file is eligible to be deleted as long as the link count is greater than zero, the inode is still a child of some directory and cannot be removed.

There are countless file system operations that are defined by the Single Unix Standard [35] and the POSIX [56] specifications. In this chapter we only consider a subset of these operations, shown in Table 5.1. This particular selection of system calls is representative of the functionality that is present in UNIX operating systems without being exhaustive. The system calls are roughly grouped into hierarchy-altering operations, file open/close operations, inode operations, miscellaneous system calls, and file I/O operations.

Category	System call	Description
Hierarchy	mount(path, device)	Attach a file system to an existing directory
	unmount(path)	Detach a mounted file system
File	open(path, flags, mode)	Open a file for reading/writing
	creat(path, flags, mode)	Create a new file
	close(fd)	Close an open file
Inode	stat(path)	Get information about a file
	chmod(path, mode)	Change the mode of a file
	link(path, newpath)	Create a link to a file within a directory
	unlink(path)	Remove a link to a file from a directory
	mkdir(path)	Create a new directory
	rmdir(path)	Remove an existing empty directory
Misc	sync()	Synchronize and commit file system
	fork()	Fork an existing process
	chdir(path)	Change working directory of a process
	exit()	Exit the calling process
I/O	read(fd, buf, len)	Read the contents of an open file
	write(fd, buf, len)	Write contents to an open file
	seek(fd, pos, whence)	Move the position marker for an open file

Table 5.1: A sample of the UNIX file system calls

There are some interesting conflicts that can arise between these system calls in a concurrent environment. The most obvious case is when one process attempts to read from a file while another is writing to the same file. In addition there is a more subtle clash between those system calls that accept a path name as a parameter and those that fundamentally alter the hierarchy of the file system. The process to resolve a path name into a specific inode involves performing a file lookup in each successive directory in the path. If the hierarchy can be altered during this resolution process, it would be possible for the file system to fail unexpectedly or even to silently return a different file (possibly bypassing any permissions or restrictions in place).

Below the system call level, in the underlying implementations, there are similar issues that must be considered when multiple processes can access the file system simultaneously. There are several different strategies for ensuring the necessary mutual exclusion.

5.1.2 Explicit Locking in the UNIX Operating System

The approach to multi-tasking in the early implementations of UNIX [45] is to utilize interrupt masking and semaphores to define small "critical sections" of the program code. The goal of this is to prevent two processes that share the same data structure from entering the critical section at the same time, ensuring each has exclusive access within that section. This ensures that the two concurrent operations can proceed until the point where they both attempt to enter competing critical sections. Practically, this means that any number of processes can perform file system operations concurrently and their execution of those critical sections will be serialised in some order.

Figure 5.1 shows a slightly simplified example of a critical section in the V6 UNIX kernel [45]. This figure contains a portion of the `free` procedure, which is used to return a disk block to the free list for its device, allowing it to be re-allocated. The list requires exclusive access to the file system, and so it is protected by the `fp->s_flock` lock. The procedure checks to see if the lock is already held by another process and suspends itself using the `sleep` function when this is the case. At some point in the future the lock will be released and the process will be awakened. The process will continue to loop, checking to see if the lock is available (another process might have acquired it in the meantime), until it finally acquires the lock.

The `sleep` procedure is shown in Figure 5.2 and allows a process to voluntarily yield control of the processor until a corresponding `wakeup` call occurs for the specified parameter, in this case the lock variable itself. However, it is possible for the execution

of `sleep` to be pre-empted by an interrupt handler that might try to suspend the same process for some other reason, such as the scheduling quantum expiring. In order to prevent this, the procedure calls the `spl6` function to change the priority level of the processor, inhibiting clock interrupts. Once the critical changes to the lock data structure are complete, the priority level is restored.

```
free(dev, bno) {
    register *fp, *bp, *ip;

    fp = getfs(dev);
    while (fp->s_flock)
        sleep(&fp->s_flock, PINOD);

    fp->s_flock++;
    bp = getblk(dev, bno);
    ip = bp->b_addr;
    *ip++ = fp->s_nfree;
    bcopy(fp->s_free, ip, 100);
    fp->s_nfree = 0;
    bwrite(bp)
    fp->s_flock = 0;
    wakeup(&fp->s_flock);
}
```

Figure 5.1: A portion of the UNIX `free` procedure

```
sleep(chan, pri) {
    register *rp, s

    s = PS->integ;
    rp = u.ui_procp;
    if (pri >= 0) {
        // Handle signals
    } else {
        spl6();
        rp->p_wchan = chan;
        rp->p_stat = SSLEEP;
        rp->p_pri = pri;
        spl0();
        swtch();
    }
    PS->integ = s;
    return;
}
```

Figure 5.2: A portion of the UNIX `sleep` procedure

Both the `free` and the `sleep` procedures utilize critical sections to ensure mutual exclusion, although they vary in the context in which they are called and the style of protection that is used. Unfortunately, this approach is highly tuned to the specific details of the underlying hardware and the approach is not scalable to machines with multiple processors.

In particular, the `sleep` function in Figure 5.2 works by assuming that its execution is not interleaved with or interrupted by another thread, and that its execution is not in parallel with another thread. Failing either of these assumptions, two threads could be executing the `free` procedure at the same time. Both threads could discover that the value of `fp->s_flock` is zero very close together and increment the lock, believing that the other thread has been locked out. The result is a classic 'time of check to time of update' (TOCTOU) software bug that could be remedied on modern hardware using atomic compare-and-set instructions.

Modern day UNIX derivatives such as the Linux kernel and FreeBSD utilize a combination of semaphores, spin locks, and mutexes rather than interrupt masking

to implement the same strategy. These locking mechanisms are provided by the respective kernels, and are implemented in a portable, efficient, and scalable way.

For example, Figure 5.3 shows the `unlazy_walk` function, which is used in path name resolution in the Linux 2.6 kernel. Ten of the lines shown are explicit lock and unlock operations on five different data structures. In addition, the documentation for this function indicates that it must be called from a specific context, where a set number of locks have already been acquired. In fact, the Linux kernel is littered with comments and text documentation about these locking conventions and mechanisms, which can vary substantially from subsystem to subsystem.

There is no doubt that this strategy is effective for the user of the system, given an efficient implementation. However, it should be clear from these code examples that utilizing this style of locking is anything but straightforward. It is difficult to develop an intuition about how, why, and where the different critical sections are being used.

5.1.3 The MINIX File Server

MINIX is a UNIX-like operating system that was created as an educational tool for studying and experimenting with the internals of operating systems. As a result of this focus, MINIX often opts for simplicity over features that would make the operating system more practically useful. One such simplification is that the file server can only process a single file system call at a time.

MINIX is designed to be compatible with the UNIX system call interface, but the underlying architecture is quite different. Rather than consisting of a single monolithic program that runs in privileged mode, MINIX is a collection of server processes that run on top of a microkernel. This microkernel provides only a small amount of functionality: address space management, process management and scheduling, and inter-process communication. Device drivers, networking support, and the file system are all provided by server processes that run in user mode and call the microkernel to perform privileged operations such as reading and writing device registers. These servers communicate and co-operate with each other to provide the full functionality of the operating system.

This arrangement is itself a process-oriented system. The different system servers communicate with each other using synchronous message-passing that is provided by the microkernel. Messages are directed to services by name, and servers utilize a request/response pattern to communicate results. The operating system provides mechanisms for this, leaving the policy to the server process. Each server may restrict the number of requests that it accepts, typically one at a time.

```
static int unlazy_walk(struct nameidata *nd, struct dentry *dentry)
{
    if (nd->root.mnt && !(nd->flags & LOOKUP_ROOT)) {
        want_root = 1;
        spin_lock(&fs->lock);
        if (nd->root.mnt != fs->root.mnt ||
                nd->root.dentry != fs->root.dentry)
            goto err_root;
    }
    spin_lock(&parent->d_lock);
    if (!dentry) {
        if (!__d_rcu_to_refcount(parent, nd->seq))
            goto err_parent;
    } else {
        if (dentry->d_parent != parent)
            goto err_parent;
        spin_lock_nested(&dentry->d_lock, DENTRY_D_LOCK_NESTED);
        if (!__d_rcu_to_refcount(dentry, nd->seq))
            goto err_child;
        parent->d_count++;
        spin_unlock(&dentry->d_lock);
    }
    spin_unlock(&parent->d_lock);
    if (want_root) {
        path_get(&nd->root);
        spin_unlock(&fs->lock);
    }
    mntget(nd->path.mnt);

    unlock_rcu_walk();
    nd->flags &= ~LOOKUP_RCU;
    return 0;

err_child:
    spin_unlock(&dentry->d_lock);
err_parent:
    spin_unlock(&parent->d_lock);
err_root:
    if (want_root)
        spin_unlock(&fs->lock);
    return -ECHILD;
}
```

Figure 5.3: An example of the locking mechanisms used in the Linux file system

The file server sub-system consists of a single loop that receives a system call request, performs the required work, and then sends a response with the results. Although MINIX does support suspending certain types of requests and accepting others, e.g., waiting for keyboard input and operations on pipes between processes, these are treated as a special case.

This lack of concurrency helps to simplify the implementation, since there is no opportunity for two processes to contend for resources. As a point of comparison, Figure 5.4 shows a portion of the path resolution code from the MINIX file system server. Compared to the equivalent code from the Linux kernel, this code is well documented, and much easier to understand. Since the server will only handle one request at a time, the program code can focus on the logic needed to navigate the file system hierarchy, rather than being littered with the details of multiple levels of lock acquisition.

The sequential limitation is not without its disadvantages, however. A file system request that needs to access a slow device or requires a large data transfer may cause the file server to be unresponsive for the duration of that operation. This can manifest itself as a pause that is observable by clients of the file system. For example, if a second user were to attempt even a cheap operation like attempting to change directory while the file server was handling such a request, the system would appear to hang until the first operation was completed. In practice, this matters little on a single user computer with reasonably fast I/O devices, although it was particularly frustrating on older machines when a background task brought the system to its knees by accessing a diskette drive.

For this case study, the sequential design of the MINIX file server is an advantage since it embodies one extreme of concurrency with a real UNIX-style file system. It provides a reference implementation that isn't already littered with compromises and special cases designed to handle a particular style of concurrency.

5.2 MINIX Implementation in Go

The MINIX file server provides an attractive starting point for design exploration. The file system logic, data structures, and algorithms are already present and free from explicit mechanisms for mutual exclusion. The existing source code is written in C, so most of the concepts are directly translatable into Go code. A major exception is the use of pointer arithmetic, which is not possible in Go. However, this is generally only used when stepping through two arrays in order to compare them, something

```
PUBLIC struct inode *last_dir(char *path, char string[NAME_MAX]) {
    register struct inode *rip;
    register char *new_name;
    register struct inode *new_ip;

    /* Is the path absolute or relative?  Initialize 'rip' accordingly. */
    rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);

    /* error handling removed */

    dup_inode(rip);    /* inode will be returned with put_inode */

    /* Scan the path component by component. */
    while (TRUE) {
        /* Extract one component. */
        if ( (new_name = get_name(path, string)) == (char*) 0) {
            put_inode(rip);  /* bad path in user space */
            return(NIL_INODE);
        }
        if (*new_name == '\0') {
            if ( (rip->i_mode & I_TYPE) == I_DIRECTORY) {
                return(rip);  /* normal exit */
            } else {
                /* last file of path prefix is not a directory */
                put_inode(rip);
                err_code = ENOTDIR;
                return(NIL_INODE);
            }
        }

        /* There is more path.  Keep parsing. */
        new_ip = advance(rip, string);
        put_inode(rip);    /* rip either obsolete or irrelevant */
        if (new_ip == NIL_INODE) return(NIL_INODE);

        /* The call to advance() succeeded.  Fetch next component. */
        path = new_name;
        rip = new_ip;
    }
}
```

Figure 5.4: A portion of the path resolution code from MINIX

that can be implemented by introducing a variable to represent the current index of the iteration.

As an introduction to the data structures and algorithms used in the MINIX file system code, we first ported the userspace utility `fsck` program to Go. This program is used to examine the state of a file system image in order to ensure that it is consistent. For example, it verifies that the allocation bitmaps that are stored on disk match the known list of allocated inodes and blocks, and that the only files stored are those that are listed somewhere in the hierarchy. The source code for our implementation can be found at `http://github.com/jnwhiteh/minixfs/fsck`.

The resulting program is mostly a line-by-line translation of the existing C code and as a result is not well-structured Go code. Figure 5.5 shows the `chkmap` function from the original MINIX source, while Figure 5.6 shows the same function written in Go. This function is used to compare a bitmap that is stored on disk with the one that was computed during the processing of the file system image. It steps through each word of the two bitmaps and ensures they are the same. The major difference between these two definitions is the use of explicit arrays and array indexing rather than directly accessing contiguous memory as an array. In addition, the Go version requires explicit type casting when working with two incompatible types (such as unsigned integers and integers, as seen in this example). The Go version is more explicit, which is no surprise given that clarity is one of the key goals of the language.

Perhaps more interesting is a comparison between the higher-level file system code. Figure 5.7 shows a Go version of the same `last_dir` function that was shown in Figure 5.4, taken from the file system implementation detailed later in this chapter. This version was written using the MINIX source as a reference, but is no longer a direct port of the existing code. As a result it uses idioms that are more familiar to Go programmers, rather than those that are several decades old and from another language. Rather than using a global error variable, errors are communicated using an additional return value that must be checked explicitly by the caller. Strings are handled natively, meaning we are able to split, join, and compare them in a simple way.

The fact that Go and C are both imperative programming languages with a similar heritage simplified the process of creating a new implementation of the file system server. Substantially more time was spent decoding pointer arithmetic in order to determine what the equivalent operation on arrays or strings would be, rather than chasing down bugs in the implementation. The resulting code is substantially easier

```
void chkmap(cmap, dmap, bit, blkno, nblk, type)
bitchunk_t *cmap, *dmap;
bit_nr bit;
block_nr blkno;
int nblk;
char *type;
{
  register bitchunk_t *p = dmap, *q = cmap;
  int report = 1, nerr = 0;
  int w = nblk * WORDS_PER_BLOCK;
  bit_nr phys = 0;

  printf("Checking %s map\n", type);
  loadbitmap(dmap, blkno, nblk);
  do {
    if (*p != *q) chkword(*p, *q, bit, type, &nerr, &report, phys);
    p++;
    q++;
    bit += 8 * sizeof(bitchunk_t);
    phys += 8 * sizeof(bitchunk_t);
  } while (--w > 0);

  if (nerr > MAXPRINT || nerr > 10) printf("%d errors found. ", nerr);
  if (nerr != 0 && yes("install a new map")) dumpbitmap(cmap, blkno, nblk);
  if (nerr > 0) printf("\n");
}
```

Figure 5.5: Bitmap comparison code from `fsck` MINIX utility

```
func chkmap(cmap, dmap []bitchunk_t, bit, blkno, nblk int, mtype string) {
    var nerr int
    var report bool
    var w int = nblk * WORDS_PER_BLOCK()
    var phys int = 0

    fmt.Printf("Checking %s map\n", mtype)
    loadbitmap(dmap, blkno, nblk)

    // the size of bitmaps should be the same
    index := 0
    for i := w; i > 0; i-- {
        if cmap[index] != dmap[index] {
            cword := uint32(cmap[index])
            dword := uint32(dmap[index])
            chkword(cword, dword, bit, mtype, &nerr, &report, phys)
        }
        index++
        bit += int(8 * Sizeof_bitchunk_t)
        phys += int(8 * Sizeof_bitchunk_t)
    }

    if nerr > MAXPRINT || nerr > 10 {
        fmt.Printf("%d errors found. ", nerr)
    }
    if nerr != 0 && yes("install a new map") {
        dumpbitmap(cmap, blkno, nblk)
    }
    if nerr > 0 {
        fmt.Printf("\n")
    }
}
```

Figure 5.6: Bitmap comparison code from `fsck` MINIX utility, ported to Go

```
func (fs *FileSystem) lastDir(proc *Proc, path string) (*Inode, string, error) {
    var rip *Inode

    absolute := strings.HasPrefix(path, "/")
    if absolute {
        rip = proc.rootdir
    } else {
        rip = proc.workdir
    }

    // If directory has been removed or path is empty, return ENOENT
    if rip.Nlinks == 0 || len(path) == 0 {
        return nil, "", ENOENT
    }

    // We're going to use this inode, so make a copy of it
    rip = fs.itable.DupInode(rip)

    pathlist := strings.Split(path, "/")
    if absolute {
        pathlist = pathlist[1:]
    }

    // Scan the path component by component
    for i := 0; i < len(pathlist)-1; i++ {
        // Fetch the next component in the path
        newrip, err := fs.advance(proc, rip, pathlist[i])

        // Current inode obsolete or irrelevant
        fs.itable.PutInode(rip)
        if newrip == nil || err != nil {
            return nil, "", ENOENT
        }
        // Continue to the next component
        rip = newrip
    }

    if rip.Type() != I_DIRECTORY {
        // The penultimate path entry was not a directory, so return nil
        fs.itable.PutInode(rip)
        return nil, "", ENOTDIR
    }

    return rip, pathlist[len(pathlist)-1], nil
}
```

Figure 5.7: Implementation of path resolution in the *minixfs* Go package

to read, and could be further improved by departing from the naming and structure used by the original implementation.

Porting the `fsck` function provided us with valuable information about the internal data structures and algorithms that are used by the MINIX file system implementation. Unfortunately this provides little insight into the behaviour and structure of the file system server. In the next section we will study the existing file system architecture. We will then use the CSP process algebra to construct a model of the file system that can be used to verify properties of the implementation. This model will then be used as a basis for exploring opportunities for concurrency.

5.3 Modelling the MINIX File Server

Thus far in this thesis we have focused on the core ideas of CSP-style concurrency (processes and channels) and the ways in which they can be applied when building concurrent software. When programs are built using occam, the compiler is able to check certain properties automatically in order to ensure that such programs are free from data race hazards. In addition, there has recently been work on the automatic generation of CSP models and specifications [17] that can used to prove extended properties of such a system.

One of the problems with the concurrency mechanisms provided by Go is that they do not align semantically with those that are described in CSP. This is an advantage in some ways, since it allows for freer constructions where processes are spawned dynamically and channels can be freely shared and passed between processes. These differences mean that the existing techniques and tools for automatic verification of such programs cannot be applied directly. However, those that have support for extensions to occam-π, such as `FORK` and mobile data types, should be adaptable to Go programs.

That isn't to say that it is not possible to model programs written in Go using CSP. After all, CSP provides ways of describing processes and the communications between them using a limited set of primitive operators. This section introduces the MINIX file system architecture and illustrates one method for building a CSP model of the architecture in order to gain a better understanding of how the system functions.

This architecture is depicted in Figure 5.8, which shows three processes interacting with the file system by issuing system calls; the system call interface converts these

requests into inter-process communication messages to the file system server, or *FS server*.

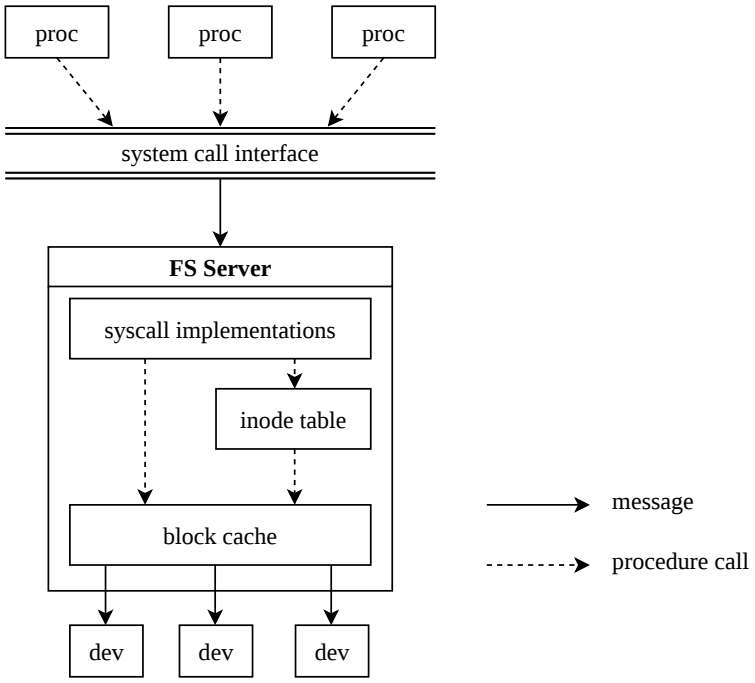


Figure 5.8: The architecture of the existing MINIX file server

A model in the CSP process algebra consists of process definitions, where a process is described by the events (communications) that it is willing to participate in. Two processes can then be combined using a variety of different primitive operators that require different synchronisation behaviour on a set of events. Given a process definition, it is possible to write a specification of desired behaviour in terms of observable events and then assert that a process is a refinement of a specification.

One of the more difficult parts of modelling a system is choosing an appropriate level of abstraction and detail. For example, it might be important to model the process of resolving a name to a particular file by abstracting away from the underlying blocks and inodes and focusing on the hierarchical contents of the file system. Similarly, the process of unlinking or renaming files requires the locking of multiple inodes and could be modelled by focusing only on the way those system calls might potentially interact, ignoring the rest of the file system behaviour.

We begin by modelling the interaction between user programs and the file system server through the sending and receiving responses from system call requests. The model will support a subset of system calls that cover most of the different types of

system call behaviour: opening and closing of files, reading from and writing to open files, and changing working directory. This requires modelling the allocation and use of file descriptors: opening a file returns a file descriptor that can be used in future `read`, `write`, and `close` calls.

What isn't covered by this abstraction is the distinction between different files in the file system and many of the lower-level details such as blocks, inodes, and the data contained within each file. We are primarily interested in the patterns of communication between user processes and the different parts of the file system. The goal is to show how the file system enables a certain type of user program behaviour and ensure that the model fails predictably in cases where failure is possible.

The code given in this section is written in CSP-M [50], a machine-readable functional programming language for writing CSP process specifications that is used by the FDR3 model checking tool. A full listing of this code, together with the verification assertions that will be discussed is given in Appendix B. To begin, we define some parameters and declare a set of channels (events) in which the processes will be able to participate.

```
Pid = {0 .. MAX_PROC-1}  -- set of user processes
Fd = {0 .. MAX_FD-1}     -- set of file descriptors

datatype Syscall = Fopen | Fclose | Fread | Fwrite | Chdir
datatype Calltype = Call | CallArg.Fd | Ret | RetVal.Fd

channel syscall : Pid.Syscall.Calltype
```

The `Syscall` datatype holds each of the various system calls that are supported by this version of the model. `Calltype` is used to indicate whether an event corresponds to an invocation of a system call or a return from it, and whether or not the message has an associated file descriptor. Each system call event is identified by the process number that is requesting the operation, the system call being requested, and the type of the call.

The file server implementation can be written to handle each of these different system calls:

```
FSServer = let
    Ready(open) =
        -- open a file
        card(open) < MAX_FD & syscall?pid.Fopen.Call ->
        syscall.pid.Fopen.RetVal?fd:diff(Fd, open) -> Ready(union(open, {fd}))
        -- close an open file
     [] syscall?pid.Fclose!CallArg?fd:open ->
        syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))
        -- read from an open file
     [] syscall?pid.Fread!CallArg?fd:open -> syscall.pid.Fread.Ret ->
        Ready(open)
        -- write to an open file
     [] syscall?pid.Fwrite!CallArg?fd:open -> syscall.pid.Fwrite.Ret ->
        Ready(open)
        -- change working directory
     [] syscall?pid.Chdir.Call -> syscall.pid.Chdir.Ret -> Ready(open)
within Ready({})
```

There are some interesting restrictions that are captured by this process definition. The file server must keep track of which file descriptors have been opened in order to offer the appropriate events. The set of file descriptors is finite in order to limit the potential state space of the model for model checking. If the set of open file descriptors is a subset of the set of all possible file descriptors (i.e., there are unallocated file descriptors) then it is possible to handle an `open` request and allocate a new file descriptor. Note that if there are no unallocated file descriptors, the server will deadlock. Once a file has been opened, it can be used in a parameter to a `read`, `write`, or `close` request. If such a request comes in for a file descriptor that is unallocated, the server will deadlock. In addition, it is possible for a process to change its working directory at any time.

One thing that this process definition does not ensure is that the process which reads from a file is the one that opened the file in the first place. That means it would allow process 0 to open a file only to have it suddenly closed by process 1. This would make it difficult to write meaningful specifications, so it is necessary to include a regulator process to ensure that a process can take action only on the file descriptors that it has received through `open` requests.

```
FDRegulator(pid, open_fds) =
    card(open_fds) < MAX_PROC_FILES &
    syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd ->
    FDRegulator(pid, union(open_fds, {fd}))
 [] syscall.pid.Fclose.CallArg?fd:open_fds -> syscall.pid.Fclose.Ret ->
    FDRegulator(pid, diff(open_fds, {fd}))
 [] syscall.pid.Fread.CallArg?fd:open_fds -> syscall.pid.Fread.Ret ->
    FDRegulator(pid, open_fds)
 [] syscall.pid.Fwrite.CallArg?fd:open_fds -> syscall.pid.Fwrite.Ret ->
    FDRegulator(pid, open_fds)
 [] syscall.pid.Chdir.Call -> syscall.pid.Chdir.Ret ->
    FDRegulator(pid, open_fds)
```

In the model, one instance of the `FDRegulator` process is spawned for each user program and these are composed in parallel, with each process only synchronising on system call events for its assigned process number. This collection of regulator processes is then combined with the file system server such that each side must synchronise on all system call events.

```
FDRegulators = (|| pid:Pid @ [syscall_events({pid})] FDRegulator(pid, {}))
Filesystem = FSServer [| syscall_events(Pid) |] FDRegulators
```

The file system process handles the allocation of file descriptors, while the regulator processes ensure that the system is only required to support behaviour that is well-defined according to the POSIX file system interface. The parallel composition combines these two properties into a single process. In order for an event to proceed, both of these internal processes must be willing to allow it.

Given this implementation, we can write a specification of behaviour that we would like the system to be capable of. For example, we could model the system call behaviour needed to implement the UNIX `cp` program as follows:

```
OpenCopyClose(pid) = let
    Open =
        syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd1 ->
        syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd2 -> Copy(fd1, fd2)
    Copy(fd1, fd2) =
        syscall.pid.Fread.CallArg.fd1 -> syscall.pid.Fread.Ret ->
        syscall.pid.Fwrite.CallArg.fd2 -> syscall.pid.Fwrite.Ret -> Close(fd1, fd2)
    Close(fd1, fd2) =
        (syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret ->
        syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret -> Open)
     [] (syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret ->
        syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret -> Open)
within Open
```

In order for a program to copy a file, the process must open both the source and the destination files, read data from the source file and write it to the destination file, and then close both of the files. Note that the communication captured here does not include information about the names of these files of their contents. This process only shows how a program can successfully perform the system calls that are needed to implement the copy functionality.

What we would like to show in this case is that the file system implementation allows the behaviour that is required by the copy program. In CSP this means we need to define a relation between a process specification (the behaviour that is required) and a process implementation. Our strategy here is to constrain process 0 to performing a specific set of actions (the copy script given above) while allowing all other processes to perform whatever system calls are possible. Using this, we can then show that every possible behaviour allowed by this specification is supported by the file server process implementation.

However, in order to assert such a refinement relation we will need a few additional processes. The `SeqLimit` process ensures that the start of a system call is always followed by a return event for the same process ID.

```
SeqLimit = syscall?pid?call?calltype -> syscall.pid!call?rettype -> SeqLimit
```

In order to model an environment with more than one user process, we will also need to be able to model the behaviour of the other processes in the system (since process 0 will be performing the copy operation).

```
OtherProcs = CHAOS(syscall_events(diff(Pid, {0})))
```

It is also necessary to make a small alteration to the `OpenCopyClose` process when it appears on the left-hand side of the refinement. We want to require that a file descriptor is returned for each `Fopen` call, but we do not care which file descriptor this might be (and do not require that the implementation offer more than one). The process `OpenAny` specifies this behaviour using the non-deterministic input operator (`$`) that was introduced in FDR 3.0 [67]. It behaves exactly the same as the normal input operator, but instead offers the non-deterministic choice of the available events using internal choice.

```
OpenAny(pid) = syscall.pid.Fopen.RetVal$fd1 -> OpenAny(pid)
CopySpec = OpenCopyClose(0) [| {| syscall.0.Fopen.RetVal |} |] OpenAny(0)
```

Finally we can build the specification and implementation and assert the refinement relation.

```
SeqSpec = (CopySpec ||| OtherProcs) [| {| syscall |} |] SeqLimit
syscall_0_events = {| syscall.0 |}

assert SeqSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))
```

The specification in this assertion (the left-hand side of the refinement) is a system in which process 0 repeatedly runs the copy program, and at the same time, all other processes are allowed to take any action that is allowed by the system. The right-hand side of the assertion is a process that allows normal file system behaviour for all processes other than process 0, which is only allowed to follow the behaviour specified by the `OpenCopyClose` process.

Running this assertion within the FDR3 model checker, we can see that the refinement relation between these processes holds. This proves some fairly strong properties about the file system. Since the refinement is being checked in the failures and divergences model (FD) this shows that it is not possible for the implementation to diverge[1]. In addition, the system cannot refuse to allow process 0 to open two files, copy data between them, and close both of the files again. The system may also not restrict the order in which these files are opened or closed; those decisions must be left up to the program that is making the system calls.

Since the specification includes the behaviour of the other processes in the system, the model also shows that these properties hold *regardless* of what actions are being taken by other processes in the system. Again, this is not a very strong guarantee for a sequential file server implementation, but becomes much stronger when it becomes possible for processes to compete with each other for file system resources.

A problem with this sort of model is that it isn't abundantly clear that the specification and implementation accurately reflect the intention or reality of the system. The model abstracts away from both the communication patterns and the internal architecture of the file server, making it difficult to have confidence in a model that only has passing assertions. It can be useful to generate some contradictions in order to better understand the limitations of the model.

For example, we could remove the `SeqLimit` from the specification. Recall that this prevents the entirety of all processes (both the copying process and all other processes) from executing more than one system call at a time. We would expect the

[1]While this is not terribly important for this sequential file system model, it becomes a more powerful guarantee in more complex versions of the model.

same assertion without this regulator to fail, since the file system only supports one system call at a time.

```
assert not NonSeqSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))
```

The model checker confirms this failure and provides a counter-example:

```
After performing the trace:
    <syscall.1.Fopen.Call>
the implementation offers the set of events:
    {syscall.1.Fopen.RetVal.0,
 syscall.1.Fopen.RetVal.1,
 syscall.1.Fopen.RetVal.2,
 syscall.1.Fopen.RetVal.3,
 syscall.1.Fopen.RetVal.4,
 syscall.1.Fopen.RetVal.5}
which is not a superset of one of the specification acceptances:
    {syscall.0.Fopen.Call}
```

These counter-examples can be somewhat difficult to read, but they are based on sets of accepted (or refused) events. After the implementation begins the `open` request for process 1, the server is (correctly) only willing to provide a response for that call. However, since the specification is no longer limited to sequential behaviour, it still allows process 0 to begin the copy behaviour. As a result, the set of events that are accepted differ between the specification and implementation, resulting in the failed refinement relation.

It isn't necessary to make structural changes to the model in order to explore its behaviour. This particular model has three different configurable parameters: the number of processes in the system, the maximum number of open files per process, and the maximum number of file descriptors that are available across all processes[2] Since the copy process requires two open files to function, it would be interesting to limit the number of file descriptors and see how the model reacts. With three processes, two open files per process, but only *five* available file descriptors, the model checker is able to find a state in which the `SeqSpec` assertion fails.

```
After performing the trace:
    <syscall.2.Fopen.Call, syscall.2.Fopen.RetVal.1,
    syscall.2.Fopen.Call, syscall.2.Fopen.RetVal.2,
    syscall.1.Fopen.Call, syscall.1.Fopen.RetVal.4,
    syscall.0.Fopen.Call, syscall.0.Fopen.RetVal.0,
    syscall.1.Fopen.Call, syscall.1.Fopen.RetVal.3>
```

[2]The assertion scripts listed in Appendix B instantiate new instances of processes using these arguments. At the time of writing there is a bug in the FDR3 parser that causes these arguments to be reversed, hence they are given in reverse order.

```
the implementation offers the set of events:
    {syscall.1.Fclose.CallArg.3,
 syscall.1.Fclose.CallArg.4,
 syscall.1.Fread.CallArg.3,
 syscall.1.Fread.CallArg.4,
 syscall.1.Fwrite.CallArg.3,
 syscall.1.Fwrite.CallArg.4,
 syscall.1.Chdir.Call,
 syscall.2.Fclose.CallArg.1,
 syscall.2.Fclose.CallArg.2,
 syscall.2.Fread.CallArg.1,
 syscall.2.Fread.CallArg.2,
 syscall.2.Fwrite.CallArg.1,
 syscall.2.Fwrite.CallArg.2,
 syscall.2.Chdir.Call}
which is not a superset of one of the specification acceptances:
    {syscall.0.Fopen.Call}
```

This counter-example shows a sequence of events after which the copy process is prevented from performing its work. If process 1 and process 2 both open two files, then when the copy process tries to open its second file there won't be an available file descriptor, so the file server will not offer that event. Despite there being plenty of actions that can be taken by the other processes in the system, the refinement still fails because the system does not accept at least the specification's behaviour.

This model, although it is only an abstraction of the communication between user processes and a file system server, provides quite a bit of insight into subtle details of the system. Although we know intuitively the sequential MINIX file system architecture is able to support the programs that are running on the system, this model provides a much stronger vehicle for talking about and proving the details of the behaviour of the system.

5.4 Adding Concurrency for I/O Calls

Armed with a better understanding of how the sequential file server in MINIX functions, in this section we introduce concurrency to the file system. The existing monolithic design is replaced by a collection of server processes that are individually responsible for some portion of the functionality of the file system (introduced in in Figure 5.9. This echoes the principles of process-oriented design that are already present in the microkernel architecture of the MINIX operating system (shown in Figure 5.8).

As previously mentioned, the system calls listed in Table 5.1 are representative of five different categories. Each category functions with a different part of the file

system: hierarchy, file operations, permissions, permissions and links, process management and file I/O. It is not abundantly obvious that it is safe to allow for the concurrent handling of system call requests without some mechanism for mutual exclusion.

It does, however, seem that the I/O system calls have a very different interface than the others. When a process calls `creat` or `open`, the file system establishes a link called a file descriptor between the inode of the file and the process. A file descriptor contains a reference to the inode, the permissions with which the file was opened, and the current position within the file. The file position normally starts at zero and is incremented during each read or write operation, or explicitly through the `seek` system call. A file descriptor acts like a capability, granting the holder access to the file with a certain set of permissions.

The nature of the POSIX file system API is such that once a process holds a file descriptor, it will continue to have access to the file (even if the file is unlinked from the hierarchy or the permissions change). With the exception of the process exiting (explicitly or by being killed through some means) or closing the file, other file system operations are unable prevent the process from performing I/O. This makes it possible to view the I/O operations as independent from the rest of the file system, making it an ideal way to enable more concurrency in the file server.

This could be accomplished by introducing a second server that sits alongside the main file system server with the sole purpose of handling the I/O operations on open files. In order to add support for this to the model, we need to change the implementation of the file system server to be internally concurrent.[3]

We can define a new process called `IOServer` that can sit alongside the `FSServer`, with each being willing to handle a subset of the possible `syscall` events. The file system server will continue to handle `open`, `close`, and `chdir` requests, since these calls require resolving a path to an inode and making changes to the file system's internal data structures. In order to complete the opening or closing of files, the file system server will also need to communicate with the I/O server so that it can properly allocate or deallocate the file descriptors that are used as parameters for I/O calls. To do this, we can introduce a new channel for communicating these sort of 'internal call' events.

[3]The complete CSP script for this model is provided in Appendix B.2.

```
datatype Intcall = FopenIO | FcloseIO
channel intcall : Pid.Intcall.Calltype

fsio_events =
    {| intcall.pid.FopenIO.Call, intcall.pid.FopenIO.RetVal,
       intcall.pid.FcloseIO.CallArg, intcall.pid.FcloseIO.Ret
     | pid <- Pid |}
```

These calls mirror those that are sent between the user program and the file server but are instead communicated on an internal channel and use a separate data type. This is primarily to make it easier to distinguish between the internal and external versions of the calls. With this change, the allocation of file descriptors will move into the I/O server, hence the return value from the `FopenIO` event. The changes to the file system server are minimal: the `open` and `close` branches must be updated to communicate on the internal channel, and the `read` and `write` event paths are removed.

These will be handled by the new I/O server implementation.

```
FSServer = let
    Ready(open) =
        -- open a file
        card(open) < MAX_FD & syscall?pid.Fopen.Call ->
        intcall.pid.FopenIO.Call -> intcall.pid.FopenIO.RetVal?fd ->
        syscall.pid.Fopen.RetVal.fd -> Ready(union(open, {fd}))
        -- close an open file
     [] syscall?pid.Fclose!CallArg?fd:open ->
        intcall.pid.FcloseIO.CallArg.fd -> intcall.pid.FcloseIO.Ret ->
        syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))
        -- change working directory
     [] syscall?pid.Chdir.Call -> syscall.pid.Chdir.Ret -> Ready(open)
    within Ready({})
```

The I/O server has the same structure as the file system server, except that it listens for requests from the `intcall` channel as well as those coming directly from user programs on the `syscall` channel. This helps to illustrate an interesting property of this approach: internal requests and external requests will compete for the sequential request handling in the I/O server. If one process begins a read request on an open file, then that system call must finish before the I/O server will be prepared to handle any other requests, including opening and closing of files. This subtle detail will be important when writing a specification for the concurrent server implementation.

```
IOServer = let
    Ready(open) =
        -- handle an open file request
        intcall?pid.FopenIO.Call -> intcall.pid!FopenIO.RetVal?fd:diff(Fd, open) ->
        Ready(union(open, {fd}))
        -- close an open file
     [] intcall?pid.FcloseIO!CallArg?fd:open -> intcall.pid.FcloseIO.Ret ->
        Ready(diff(open, {fd}))
        -- read from an open file
     [] syscall?pid.Fread!CallArg?fd:open -> syscall.pid.Fread.Ret -> Ready(open)
        -- write to an open file
     [] syscall?pid.Fwrite!CallArg?fd:open -> syscall.pid.Fwrite.Ret -> Ready(open)
within Ready({})
```

These two servers can be composed together so that they synchronise on events on the `intcall` channel. Since the rest of the file system doesn't care about these events, they must be hidden.

```
Servers = (FSServer [| fsio_events |] IOServer) \ fsio_events
Filesystem = Servers [| syscall_events(Pid) |] FDRegulators
```

Despite being a fairly simple change to the model, we would expect this change to enable concurrent I/O alongside normal file system calls. To begin, we can re-check the existing assertions with the new implementation of the file system in order to verify that the behaviour has not changed. To do so, we can limit the server implementation to only allow a single system call to be processed at at time using the `SeqLimit` process.

```
SeqFilesystem = Filesystem [| {| syscall |} |] SeqLimit
```

Using this new process, the previously held assertions continue to hold. Despite the fact that the internal implementation has changed to allow additional concurrency, the behaviour of the sequentially restricted system has not changed. It is also possible to check that the (slightly) concurrent `Filesystem` process no longer refines the `SeqSpec` specification.

```
assert SeqSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0))

assert not ConcSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0))
```

Verifying the concurrent nature of the implementation is slightly more tricky due to the internal communication that is required between the FS and I/O servers for open and close calls. For example the following sequence of events results in surprising

behaviour: process 1 begins and completes an `open` system call; process 1 begins a `read` system call; process 0 begins an `open` system call. When the second process comes along only the I/O server is handling requests, so the FS server is able to begin processing the `open` request. However, since the file system server needs to contact the I/O in order to allocate a file descriptor, the request is forced to wait for the completion of the outstanding I/O request.

To account for this, we can write a new regulator specification that takes this into consideration, called `ConcLimit`.

```
channel syncio
ConcLimit = let
    FS = syscall?pid.Chdir.Call -> syscall.pid.Chdir.Ret -> FS
      [] syscall?pid.Fopen.Call -> syncio -> syscall.pid.Fopen.RetVal?fd -> FS
      [] syscall?pid.Fclose!CallArg?fd -> syncio -> syscall.pid.Fclose.Ret -> FS

    IO = syscall?pid.Fread!CallArg?fd -> syscall.pid.Fread.Ret -> IO
      [] syscall?pid.Fwrite!CallArg?fd -> syscall.pid.Fwrite.Ret -> IO
      [] syncio -> IO

within (FS [| {| syncio |} |] IO) \ {| syncio |}

-- A specification that allows concurrency
ConcSpec = (CopySpec ||| OtherProcs) [| {| syscall |} |] ConcLimit
```

This can be combined with the copy specification and the concurrent file system implementation in order to test the behaviour of the model:

```
assert ConcSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))
```

This refinement relation holds and provides a similar set of guarantees to those in the previous section, with the additional requirement that the implementation must support concurrent I/O and file system calls. We can verify this by changing the right-hand side of the assertion to use the sequential file system.

```
assert not ConcSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0))
```

Running this assertion results in the following counter-trace:

```
After performing the trace:
    <syscall.1.Fopen.Call, tau, tau,
    syscall.1.Fopen.RetVal.0,
    syscall.1.Fread.CallArg.0>
the implementation offers the set of events:
    {syscall.1.Fread.Ret}
which is not a superset of one of the specification acceptances:
    {syscall.0.Fopen.Call}
```

The specification expects the server to allow an open call to begin immediately after a `read` call so the refinement fails. Even though the server would be forced to return from the `read` call, any deviation in the set of events that are accepted or refused is sufficient to fail the refinement check.

Although the guarantees provided by the model are strong, it does not currently reflect the reality of the implementation. Recall that the original sequential implementation contained several subsystems that were used to cache blocks and inodes. In the sequential design, both of these caches are data structures in the main file system server. With the I/O server accepting some system calls, it will need to be able to interface with these subsystems in order to fetch the contents of files and update the details stored in the inode. Although it would be possible for the I/O server to communicate with the file system server in order to do this, such requests would then compete with the other system call requests. This would increase the complexity of the implementation with very little practical gain. An alternative is presented in the next section.

5.5 Modelling Internal Subsystems

Process-oriented abstraction allows us to extract the block cache and inode tables from the FS server into independent server processes. Both the I/O server and the file system server can then communicate with them independently. Even if each of these new servers were sequential, this would still be an improvement over the existing MINIX system, where the smallest atomic operation is the system call. With an independent block cache server, each of the I/O and FS servers would be able to process one system call concurrently, with block cache requests being serialized when they conflict.

Another consequence of allowing I/O to occur alongside other system calls is a potential conflict when allocating new inodes or data blocks. When the size of a file or directory is extended, a new data block must be allocated. This is accomplished by fetching the bitmap blocks from the front of the disk, searching for a spare slot, and writing the updated blocks back to disk. Since both the file I/O server and the FS server might need to allocate new blocks, it is important to ensure they do not both try to access the bitmap blocks at the same time. A separate allocation server can be created to be responsible for this functionality.

Figure 5.9 shows a process diagram of this architecture, with each rectangle representing an independent server. The arrows between these servers represent messages,

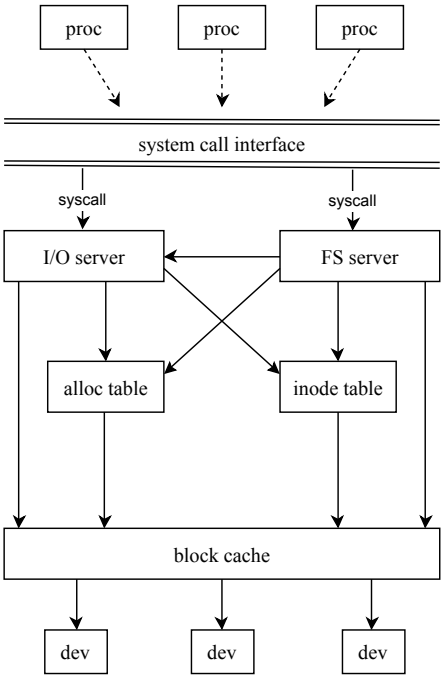


Figure 5.9: File server that supports concurrent file I/O

with the direction of the arrow showing the initial communication. System call requests from programs are converted by the system call interface into messages for either the I/O server or the FS server. In turn, each of these servers might need to communicate with the block cache, inode table, or the allocation server. Since the communications in this design follow the client-server protocol and there are no client-server cycles in the topology, we know that the system is deadlock-free [62, 46].

The models that have been used thus far in this chapter have abstracted away from many of the details of the file system implementation. However this architecture requires internal communication that must be part of the model.[4]. The calls from the system call interface to the I/O and FS servers are delivered via the `syscall` channel, where all other communications inside the system will be sent over the `intcall` channel.[5] In addition to the calls between the FS servers and the I/O server, both servers must communicate with the underlying cache and allocation servers. In order to model these, we can alter the `Intcall` event to include the calls that are supported

[4]A full listing of this CSP script along with verification assertions that will be discussed are provided in Appendix B.3

[5]The term "'channel"' is used somewhat loosely here to refer to a channel definition in CSP, in contrast with the channels depicted in Figure 5.9 which are point-to-point channels in the more traditional sense. In the CSP model the communication over these point-to-point channels is accomplished using the same CSP channel.

by the block cache, inode cache, and allocation server.

```
datatype Intcall = FopenIO | FcloseIO | GetBlock | PutBlock |
                   GetInode | PutInode | AllocInode | FreeInode |
                   AllocBlock | FreeBlock
```

Our implementation of these internal servers will continue to focus on patterns of communication, rather than the underlying contents of blocks and nodes. Figure 5.10 shows the implementation and composition of these three servers. The block cache internally is able to model the communication with underlying disk devices, providing a framework for testing interaction with multiple devices in future models. Both the `InodeCache` and the `AllocServer` contact the block cache as part of handling their own calls, but there is no need for them to synchronize with each other.

```
channel devio : Pid.Calltype
BlockCache = let
    devio_events = {| devio |}
    Device = devio?pid.Call -> devio.pid.Ret -> Device
    Cache = intcall?pid.GetBlock.Call -> devio.pid.Call -> devio.pid.Ret ->
            intcall.pid.GetBlock.Ret -> Cache
         [] intcall?pid.PutBlock.Call -> devio.pid.Call -> devio.pid.Ret ->
            intcall.pid.PutBlock.Ret -> Cache
    within (Cache [| devio_events |] Device) \ devio_events

InodeCache =
    -- Fetch an inode
    intcall?pid.GetInode.Call -> intcall.pid.GetBlock.Call ->
    intcall.pid.GetBlock.Ret -> intcall.pid.PutBlock.Call ->
    intcall.pid.PutBlock.Ret -> intcall.pid.GetInode.Ret -> InodeCache
    -- Release an inode
 [] intcall?pid.PutInode.Call -> intcall.pid.GetBlock.Call ->
    intcall.pid.GetBlock.Ret -> intcall.pid.PutBlock.Call ->
    intcall.pid.PutBlock.Ret -> intcall.pid.PutInode.Ret -> InodeCache

AllocServer = let
    Ready = intcall?pid.AllocBlock.Call -> DoBlockIO(pid, AllocBlock)
         [] intcall?pid.FreeBlock.Call -> DoBlockIO(pid, FreeBlock)
         [] intcall?pid.AllocInode.Call -> DoBlockIO(pid, AllocInode)
         [] intcall?pid.FreeInode.Call -> DoBlockIO(pid, FreeInode)
    DoBlockIO(pid, calltype) =
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        intcall.pid.calltype.Ret -> Ready
    within Ready

InodeAlloc = InodeCache ||| AllocServer
```

Figure 5.10: Implementation of internal file system servers

It is actually rather difficult to model the interactions that might occur between

the main system call handling servers and these internal support processes. Writing data to a file may result in the allocation of a new block, updating of an existing block, and/or the truncation of a set of blocks. As shown in Figure 5.4, the process of resolving a path into an inode may involve the fetching of multiple inodes, scanning of directory listing, and then releasing the inodes to move onto the next component. In this model, we have implemented versions of the system call handling servers that are able to accurately reflect these more complex patterns of communication. For example, Figure 5.11 shows an implementation of the I/O server with support for multi-block reads, writes, and the allocation and truncation of blocks.

In this process definition, a read request is eventually sent to the `MultiBlockRead` process which non-deterministically chooses between reading additional blocks or completing the read operation. Again, this model abstracts away from individual files or the contents of blocks, but is capable of modelling the possible communication that might occur between the internal servers. More interesting is the process for writing contents to a file. Either the call may result in the update of an existing block, the allocation and writing of a new block, the truncation and freeing of an existing block, or the completion of the write event. The implementation may choose any of these paths at any point in the model, which helps to cover a wide range of possible behaviours.

The file system implementation, shown in Figure 5.12, follows a similar but slightly more involved pattern. The process of opening a file or changing a directory involves path resolution, so those calls are redirected to the `Lookup` process definition. A lookup can either succeed immediately or require a recursive lookup, with each intermediate directory level being released back to the inode cache. When the lookup for a change directory is complete the system call is essentially done, so the process returns the old working directory and returns to the caller. An open file request can either result in the immediate opening of the inode that was returned, or it may result in the allocation of a new file. If a new file is created then it must be linked into the appropriate directory before the file descriptor can be returned to the calling process.

It should be noted that at each point of recursion in both of these implementations, the process will communicate either on the `devio_recurse` channel or the `lookup_recurse` channel. These are 'marker' events that help to prevent the unbounded loops in the implementation from leading to divergence in the model. The first indicates that some work is being done involving device I/O, while the second communicates that the server is attempting to resolve a path into an inode. A divergence is detected in a CSP model when a process reaches a state whereby it could

```
IOServer = let
    Ready(open) =
        -- handle an open file request
        intcall?pid.FopenIO.Call ->
        intcall.pid!FopenIO.RetVal?fd:diff(Fd, open) -> Ready(union(open, {fd}))
        -- close an open file
     [] intcall?pid.FcloseIO!CallArg?fd:open -> intcall.pid.FcloseIO.Ret ->
        Ready(diff(open, {fd}))
        -- read from an open file
     [] syscall?pid.Fread!CallArg?fd:open -> MultiBlockRead(open, pid)
        -- write to an open file
     [] syscall?pid.Fwrite!CallArg?fd:open -> MultiBlockWrite(open, pid)

    MultiBlockRead(open, pid) =
        -- more blocks to read
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockRead(open, pid)
        -- done reading
    |~| intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        syscall.pid.Fread.Ret -> Ready(open)

    MultiBlockWrite(open, pid) =
        -- update an existing block
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockWrite(open, pid)
        -- allocate and write to a new block
    |~| intcall.pid.AllocBlock.Call -> intcall.pid.AllocBlock.Ret ->
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockWrite(open, pid)
        -- block is truncate, free
    |~| intcall.pid.FreeBlock.Call -> intcall.pid.FreeBlock.Ret ->
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockWrite(open, pid)
        -- write to a block and finish
    |~| intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        syscall.pid.Fwrite.Ret -> Ready(open)
within Ready({})
```

Figure 5.11: I/O server implementation

```
FSServer = let
    Ready(open) =
        -- open a file
        card(open) < MAX_FD & syscall?pid.Fopen.Call -> Lookup(open, pid, Opening)
        -- close an open file
     [] syscall?pid.Fclose!CallArg?fd:open -> Closing(open, pid, fd)
        -- change working directory
     [] syscall?pid.Chdir.Call -> Lookup(open, pid, ChangingDir)

    Lookup(open, pid, P) =
        -- Path is found in this directory
        intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get dirp
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret -> -- lookup path
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret -> P(open, pid)
        -- Recurse to another directory level
    |~| intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get dirp
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret -> -- lookup path
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- put dirp
        lookup_recurse -> Lookup(open, pid, P)

    Opening(open, pid) =
        -- opening an existing file
        intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get file inode
        intcall.pid.FopenIO.Call -> intcall.pid.FopenIO.RetVal?fd ->
        syscall.pid.Fopen.RetVal.fd -> Ready(union(open, {fd}))
        -- create a new file
    |~| intcall.pid.AllocInode.Call -> intcall.pid.AllocInode.Ret -> -- alloc inode
        intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get file inode
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret -> -- add to directory
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- put dirp
        intcall.pid.FopenIO.Call -> intcall.pid.FopenIO.RetVal?fd ->
        syscall.pid.Fopen.RetVal.fd -> Ready(union(open, {fd}))

    Closing(open, pid, fd) =
        -- close an open file still in hierarchy
        intcall.pid.FcloseIO.CallArg.fd -> intcall.pid.FcloseIO.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- return inode
        syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))
        -- close and delete a file
    |~| intcall.pid.FcloseIO.CallArg.fd -> intcall.pid.FcloseIO.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret ->   -- return inode
        intcall.pid.FreeInode.Call -> intcall.pid.FreeInode.Ret -> -- free inode
        syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))

    ChangingDir(open, pid) =
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- return curdir
        syscall.pid.Chdir.Ret -> Ready(open)

    within Ready({})
```

Figure 5.12: File system implementation

execute a series of internal actions or tau events indefinitely. Since each process may non-deterministically choose any of the possible behaviours, it is possible to continue internal work indefinitely without actually returning anything to the calling process. Leaking these 'recursion' events, the implementation is able to indicate that some actually work is being accomplished and that it has not diverged.

Each of the non-deterministic paths in this model represents a different code-path that was taken from the MINIX file server implementation. They are not completely accurate, since we are not currently modelling the accumulation of inodes for open files and working directories, but they help to illustrate how it is possible to extend a model to include more and more details of the implementation when those details may have some impact on the potential behaviour.

5.6 Investigating Additional Concurrency

In the previous few sections we have detailed the deconstruction of a sequential, monolithic file server architecture in order to enable the server to handle more than one request concurrently. This process has been guided by the creation and refinement of models of the abstract communication behaviour of the system. This new architecture consists of a network of sequential processes that communicate with each other using message passing. Despite the fact that no single server is able to handle more than one request concurrently, the overall system is capable of handling I/O requests alongside regular file system calls.

This section contains an analysis of additional opportunities for concurrency within each of the major components of the file server.

5.6.1 Block Cache

Each file system operation eventually involves reading or writing blocks from the connected storage. As a result, two systems calls that do not involve the same block will still compete with each other for access to the block cache. When system calls are handled sequentially this isn't a problem, but as more concurrent requests are handled by the file server, the block cache will become a bottleneck. If two requests must access the same device this may be the desired behaviour, but the structure of the system suggests that it might be possible to handle one request per underlying block device. Done correctly, this could isolate requests for different devices, preventing a slower device from delaying a request for a high-performance once. Allowing the block cache

to accept multiple requests for the same device will make it possible to schedule the requests for increased performance.

However, care must be taken to ensure that the manipulation of internal cache data structures, such as the list of free cache blocks, is done in a safe and disciplined way. This is possibly more complicated than is immediately obvious, since a request to read a block might trigger the eviction of another block and cause it to be written to a device.

In our implementation of this architecture, which is discussed in Section 5.7, we have chosen a straightforward approach that enables multiple outstanding block requests without overly complicating the system. All block cache operations are handled synchronously and sequentially, with the exception of "get block" requests. This ensures that the more complicated operations, such as mounting new devices and invalidating the cache for a given device, can be handled without the risk of conflict.

When a request to retrieve a block is received, the server first checks to see if the block is already present in the cache. If it is, the block is returned to the caller and the block cache server is immediately available to serve another request. If the block is not available, the server evicts another block (if necessary), or allocates a new cache buffer to hold the block. A new process is then spawned to load the block asynchronously from the device, which may or may not itself allow concurrent requests. When the block has been loaded successfully, it is returned to the caller. While the cache is waiting for this response, it is able to handle other requests.

This design separates the management of the cache data structures from the actual block operations on the device. Mutual exclusion for the list of cache blocks is provided by the block cache server, which only handles one request at a time. When there is a potential conflict between an outstanding asynchronous request and a block cache operation, such as a read request for a block that is already being loaded from the device), the server can use more fine-grained mutual exclusion techniques. These details can all be hidden from the consumers and dependencies of the block cache process.

5.6.2 Concurrent File I/O

The main benefit of enabling concurrent I/O alongside other file system operations is to improve performance and reduce the impact that I/O can have on other (seemingly unrelated) system operations, such as changing the working directory. This decomposition was possible because the semantics of I/O follow a different implementation pattern than the other system calls, since operations must be performed on open files.

Each of the two servers serialises incoming requests, ensuring that a maximum of two operations can be outstanding at a given time.

The previous section noted that each block in the MINIX file system is owned by a single entity in the new file system architecture. In MINIX, it is not possible to share inodes and data blocks between two different files. Since each file has a distinct set of blocks that don't overlap in any way, it should be safe to perform multiple I/O operations on different files concurrently. For a single file, we can adopt a straightforward concurrency policy of allowing concurrent read operations while requiring exclusive access for any writes.

This can be accomplished by spawning a new server process the first time a specific file/inode is opened for reading or writing. This process will be responsible for enforcing the concurrent read/exclusive write policy for handling system calls, and will continue to run as long as the file is opened by any process. When the file is closed, the server can be safely shut down.

However, there is a slight complication due to the way that file positions can be shared between processes in UNIX. If one process opens a file and then performs the *fork* system call, the newly created process will have a copy of each of the file descriptors from the original. When one of the processes moves the position in a file descriptor by seeking, reading, or writing, this change should be reflected in the other process as well. This is necessary so that (for example) multiple commands in a shell script can write one after another to the same standard output file

This introduces an additional constraint on our system: any two operations on the same file descriptor must be serialized in order to maintain the semantics of the UNIX file system. So although multiple file operations can be handled concurrently, any two that use the same shared file descriptor cannot. One way to accomplish this is by introducing a new process for each file descriptor to serialize incoming requests.

Figure 5.13 shows a version of the file server where the I/O functionality is provided by a cluster of file descriptor and open file servers. When a file is opened for the first time, two server processes will be spawned. The file descriptor will manage the current position within the file for any processes that share the same descriptor and will delegate the actual I/O operations to the underlying process that manages the file.

The general notion of connecting a file descriptor server to a user process is the approach taken in RMoX [40], an experimental operating system written in occam-π [15, 73]. The technique is used in both the network stack [61] and in several of the file system implementations. Each of these drivers takes a different approach

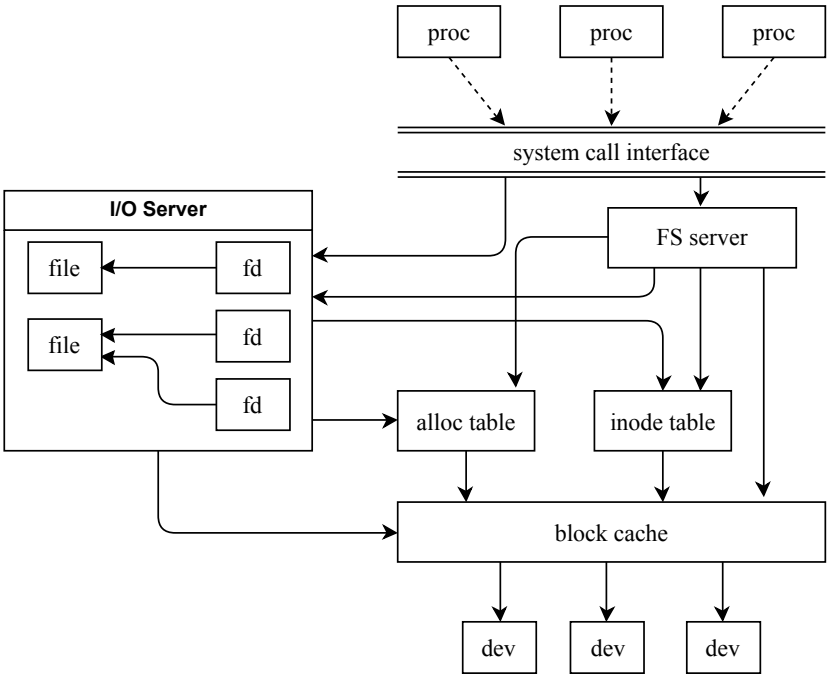


Figure 5.13: Concurrent I/O server

to synchronizing between multiple file descriptors that refer to the same file. For example, the *ramdiskfs* driver requires exclusive access to the actual ramdisk device for all read or write operations, while the *hostfs* driver relies on the blocking nature of the host operating system's file operations in order to prevent conflicts.

We have chosen in our design to separate the concerns of the file descriptor from the process that actually interacts with the block cache. This allows us to make the potential concurrency of this "open file" server explicit, and different from that of the necessarily sequential file descriptor.

It is possible, although difficult and cumbersome, to add this sort of concurrency to the existing CSP model. The `IOServer` process can be decomposed into sub-processes for the file descriptors and open files in the way indicated in Figure 5.13. Each file descriptor process could handle only the `iocall` events that are associated with its file descriptor ID. The open file servers could directly handle any other operations that don't utilize the file position.

The true difficulty comes in writing a specification that can be used to verify that the file server is actually capable of concurrent file I/O, while retaining all of the guarantees that previously hold. Such a specification must be sufficiently expressive, ensuring that all requests to the file descriptors are serialized, while at the same time allowing concurrency in the open file servers. For the purposes of this thesis we have

chosen not to include those details in the complete CSP model of the file system architecture.

5.6.3 Inode Table

The structure of the inode table is similar to the block cache, except that it utilizes the block cache as the backing store instead of directly using the devices. The same sort of concurrency policy can be used, where all requests are handled sequentially, but loads of inodes from the block cache can be performed and returned asynchronously. This is less important than the block cache, since the inode table is primarily accessed by the FS server, which is still limited to a single request at a time.

5.6.4 Allocation Table

The allocation table actually consists of two different logical sets of bitmap blocks: the inode bitmap and the data block bitmap. Operations on these two tables are completely independent. Each set of bitmap blocks could be managed by a separate server process, making it possible to handle one allocation request for each of the tables at a time.

5.7 Implementation

Many of the insights that were a factor in the development of the architecture presented in this chapter came from examination of the existing MINIX3 source code and the process of porting that code to the Go programming language. Although the focus of this work was never to find a practical drop-in replacement for the existing MINIX file system, we have succeeded in creating a collection of Go packages that exhibit the architecture laid out in this chapter and make it possible to interact with MINIX file system images using a POSIX-like API. This package can be found at `http://github.com/jnwhiteh/minixfs` with links to the documentation.

In this section we will cover the overall structure of the *minixfs* package, and detail the implementation of one of the more critical components: the block cache. Finally, we show an example of how a program can create a new instance of the MINIX file server and use it to interact with a file system image stored as a single file on the host operating system.

The source code is split up into several different packages: *common*, *bcache*, *inode*, *alloctbl*, *device*, *file* and *fs*, for better encapsulation and to ensure that there is no

implicit sharing between two independent server implementations. This structure makes sense: the inode cache needs to make use of the block cache but does not need access to its internals. By using separate packages and explicit visibility, the compiler is able to automatically ensure that this is the case.

5.7.1 *minixfs* Usage Example

Figure 5.14 shows an example Go program that illustrates how the *minixfs* package can be used. A new file system instance is created by calling the `OpenFileSystemFile` function and supplying the name of a file that contains a MINIX disk image. This function will create a new block device and spawn a file server instance to respond to system call requests, and returns a file system object and a process object. The Go program interacts with the file system using the process object, which corresponds to an operating system process on the 'guest' operating system.

This program is a test of the write functionality of the file system implementation. A large file (large enough to require a double indirect block with the default MINIX block size) is opened on the host operating system and the contents are read into memory. A new file is then created in the MINIX file system and the contents are written to this new file. The program then reads the contents that were just written and verifies that they match the contents of the original file. The file is then closed and unlinked from the file system, the process exits, and the file system server is shut down.

5.7.2 File-Backed Block Device

Each of the individual services in *minixfs* is written as an encapsulated data server, patterns for which were introduced in Sections 3.5 and 3.6. This means the services can be accessed using idiomatic objects and methods calls, while still allowing the server to have simple control over the incoming requests on channels. This involves quite a bit of boilerplate code for the message definitions and the methods to wrap the communication, but the majority of this code was auto-generated from interface definitions.

Consider the code in Figure 5.15, which shows the implementation of the main server loop for a block device. This particular block device implementation is backed by a file on the host file system and uses the *encoding/binary* package to read bytes from these files into structured data types (such as an inode block, a block of directory entries or the superblock). This package expects to read these bytes from the current

```
// Open file system and spawn a new process
fs, proc := minixfs.OpenFileSystemFile("minix3root.img")

// Open a data file on the host operating system
ofile := os.OpenFile("europarl-en.txt", os.O_RDONLY, 0666)

// Read the entire contents of the source file
filedata := ioutil.ReadAll(ofile)

// Open an output file in the guest (MINIX) file system
gfile := proc.Open("/tmp/europarl-en.txt", O_CREAT|O_TRUNC|O_RDWR)

// Write the contents to the new file
numBytes, err := gfile.Write(filedata)

// Seek to beginning of guest file and read the contents into a new buffer
written := make([]byte, numBytes)
gfile.Seek(0, 0)
gfile.Read(written)

compare := bytes.Compare(filedata, written)
if compare != 0 {
    Error("File contents did not match what was written!")
}

// Close and unlink the file on guest filesystem
proc.Close(gfile)
proc.Unlink("/tmp/europarl-en.txt")

// Terminate this process and shut down the file system
proc.Exit()
fs.Shutdown()
```

Figure 5.14: Example of how the *minixfs* package can be used to interact with a MINIX file system

Figure 5.15: File-backed block device implementation

position within the data stream, so it is necessary for us to seek to various positions when blocks are requested. This is pleasingly analogous to the seek delay required by mechanical disk drives.

Since each operation involves a seek followed by a read or a write, the server can only handle one request at a time. The main loop will repeatedly read requests from the input channel, perform the required work, and then return the result to the caller. Clients access the device using `Read` and `Write` methods that wrap calls into request messages, send these to the server using a channel, and then wait for a response[6].

When it is necessary for one service to communicate with another, this is accomplished using composition, by giving the dependant service an instance of the interface. For example, the block cache interface type includes a `MountDevice` method:

This method is called during the handling of a `mount` system call and is passed a `BlockDevice` parameter. This enables the cache to begin serving blocks for the new device without being strongly coupled to the details of its implementation.

5.7.3 Concurrent Inode Cache

The previous section detailed a block device implementation that is sequential as a result of the underlying storage medium. However, as mentioned in Section 5.6, there are many opportunities for additional concurrency within the file server. This section details the structure and implementation of an inode cache that supports concurrent requests. The inode cache server was introduced in Figure 5.13, where the FS server and individual open file servers all act as clients to the inode cache which in turn communicates with the block cache to fetch information from the underlying disk devices.

Like all of the subsystems, the inode cache is implemented as a data server goroutine that accepts requests on a channel and sends responses via another channel. The primary purpose of this server process is to regulate access to the underlying cache data structures, ensuring that they can be updated correctly and free from race

[6]It should be noted that the encapsulation of communication is fairly common in concurrent programming languages when following a request/response pattern. Erlang, for example, uses asynchronous message passing in the Actor model, and this technique is often used to perform synchronous or two-way communication with an actor process.

conditions. Since the request/response pattern used by data servers is inherently sequential, this is trivially achieved. The real challenge is ensuring that the inode cache is able to respond to requests quickly and without blocking for longer than necessary. For example, on a cache miss the inode will need to be loaded from the block cache, which may need to access the underlying block device. Without any concurrency, the inode table would be unable to serve requests until the I/O request is complete, even if the inodes being requested are already present in the cache.

In order to make the server as responsive as possible, we use asynchronous result channels (introduced in Section 3.3). In cases where additional work is needed, a new goroutine is spawned to complete the request and then deliver the result via the intermediate channel when it becomes available. Since clients can only access the inode cache through the encapsulated interface, this asynchronicity is done without requiring the client to make any changes.

The inode cache implementation has the following behaviour on receiving a request for a specific inode:

- If the inode is present in the cache, it should be returned immediately.
- If the inode is not present in the cache, it should loaded the block cache server into an available inode buffer and returned.
- If there are no available inode buffers, an error should be returned to the client, since inodes that are in use cannot be evicted.
- If the inode being requested is currently being loaded from the block cache, the inode cache should arrange for the inode to be returned when it is available.

Figure 5.16 shows a portion of the main server loop that implements this functionality in the inode cache. For each incoming `GetInode` request a new intermediate result channel, called `callback`, is created and returned to the client. This channel acts in a way that is similar to a promise [33] or future [13], as a placeholder for a value that will be available at some point in the future. The client will perform a receive on this channel, causing it to block until the inode is available.

While the server loop is handling a request it has exclusive access to all of the underlying data structures. During this time, the cache slots are searched to see if the requested inode is already present. If not, and there isn't a spare slot available in which to load the inode, then the intermediate channel is returned and an error result is immediately communicated to the client. Since we have control over the client

implementation (via the encapsulated interface), we know that this communication will be able to proceed immediately. It is therefore simpler to perform it inline, rather than spawning a new goroutine to deliver the error result asynchronously.

The search can result in the inode that was requested or a empty cache slot in which the inode can be loaded. If the inode was found, the usage counter can be incremented. Despite this being a shared field within the inode data structure, it is only altered by the main loop of the inode table in response to requests and is therefore protected by that mutual exclusion guarantee.

It may be that although the appropriate slot for an inode was found, it is still being loaded as a result of another request. The `ReturnOrQueue` method (detailed later in this section) will check the load status and determine if the inode can be returned immediately, or whether it is necessary to queue the `callback` channel so that it is notified when the inode is loaded.

The final portion of the `GetInode` branch handles the case where the inode is not present, but can be loaded into an empty cache slot. The slot is first claimed by setting the device number and the inode number, ensuring that any subsequent searches will return this cache slot The count is then incremented, showing that the inode should be considered in use. The requesting client's `callback` channel is then entered in a queue of channels that should be notified when the loading is complete. Finally, an anonymous function is created and then spawned in a new goroutine to handle the loading of the inode and the subsequent notifications.

After the loading goroutine is spawned, the server loop will be free to accept a new request. As a result, it is important that the work performed asynchronously cannot conflict with the work that could be done by the main server loop for the handling of other requests. As an example, the `DupInode` method alters the usage counter in the inode data structure, so this must not be changed during the loading or notification process.

What is more interesting is the handling of individual cache slots and the associated 'wait lists' for outstanding requests. A slot might be examined by more than one goroutine: a new `GetInode` request might arrive for the same block while it is still be loaded by a background goroutine. As a result it is necessary to ensure mutual exclusion when accessing the internal fields of the cache slot. Although it is possible to implement this as a process-oriented data server, it is not clear that there is a benefit to doing so for such a simple case. As a result, the `cacheSlot` data type has been implemented using a `sync.Mutex`. Figure 5.17 shows the entirety of this implementation.

```
func (itable *server_InodeTbl) loop() {
    alive := true
    for alive {
        req := <-itable.in
        switch req := req.(type) {
        case req_InodeTbl_GetInode:
            // Return a channel on which the inode result will be delivered
            callback := make(chan resInodeTbl)
            itable.out <- res_InodeTbl_Async{callback}

            // Find the cache slot for an inode, or an empty slot that can be used
            slotIndex := itable.findSlot(req.devnum, req.inum)
            if slotIndex == common.NO_INODE || slotIndex >= len(itable.slots) {
                // Inode table is completely full!
                callback <- res_InodeTbl_GetInode{nil, common.ENFILE}
            } else {
                slot := itable.slots[slotIndex]
                if slot.inode.Count > 0 {
                    // We found the inode, so return it
                    slot.inode.Count++
                    slot.ReturnOrQueue(callback)
                } else {
                    // The inode can be loaded into this slot
                    slot.inode.Devinfo = itable.devices[req.devnum]
                    slot.inode.Inum = req.inum
                    slot.inode.Count++
                    slot.Queue(callback)

                    go func() {
                        // Load the inode into the Inode
                        itable.loadInode(slot.inode)
                        // Notify anyone waiting for this slot
                        slot.FinishedLoading(slot.inode)
                    }()
                }
            }
        case req_InodeTbl_DupInode:
            rip := req.inode
            rip.Count++
            itable.out <- res_InodeTbl_DupInode{rip}
        case req_InodeTbl_PutInode:
            rip := req.inode
            rip.Count--
            if rip.Count <= 0 {
                slot.FreeInode()
            }
            itable.out <- res_InodeTbl_PutInode{}
        }
    }
}
```

Figure 5.16: Main server loop for a concurrent inode cache

The data structure itself consists of the inode data structure, a list of channels that are waiting for the inode to be loaded, and a mutex that is used to protect the wait list. It isn't necessary to create a separate data type with method definitions; the same manipulations could have been done in the main server loop and the asynchronous loading function. However, this abstraction makes the implementation of the inode cache slightly simpler and enforces a better separation of concerns. The mutual exclusion required for an individual cache slot can be managed by the slot itself rather than being distributed across several different goroutines.

```
type cacheSlot struct {
    inode *common.Inode // the inode itself

    waiting []chan resInodeTbl // a list of waiting goroutines
    m       sync.Mutex         // mutex for wait list
}

// Either return an inode result if it is immediately available, or queue
// the return when the inode finishes loading.
func (cs *cacheSlot) ReturnOrQueue(ch chan resInodeTbl) {
    cs.m.Lock()

    if len(cs.waiting) > 0 {
        cs.waiting = append(cs.waiting, ch)
    } else {
        ch <- res_InodeTbl_GetInode{cs.inode, nil}
    }
    cs.m.Unlock()
}

// Queue a channel to receive the inode result when it finishes loading.
func (cs *cacheSlot) Queue(ch chan resInodeTbl) {
    cs.m.Lock()
    cs.waiting = append(cs.waiting, ch)
    cs.m.Unlock()
}

// An inode has finished loading (triggered from main server loop), so deliver
// the inode result to all queued goroutines.
func (cs *cacheSlot) FinishedLoading(rip *common.Inode) {
    cs.m.Lock()

    for _, ch := range cs.waiting {
        ch <- res_InodeTbl_GetInode{cs.inode, nil}
    }
    cs.waiting = nil
    cs.m.Unlock()
}
```

Figure 5.17: Cache slot implementation for the inode cache

In this section we presented an implementation of the inode cache that uses asynchronous result delivery via channels in order to handle multiple requests concurrently. This technique makes it possible to utilize the mutual exclusion of the main server loop for the primary data structures, while handling mutual exclusion for individual cache slots in a more limited fashion. However, being able to make any of these changes requires a fair bit of understanding of the data structures that are internal to the MINIX file system implementation. In the next section we discuss how notions of data ownership can provide insight into how data structures are used by the file system.

5.8 Avoiding Concurrency Pitfalls

So far in this chapter, we have discussed several strategies for introducing concurrency to the (initially) sequential MINIX file system server. The file system server was decomposed into two servers in order to be able to handle I/O requests alongside other system call requests. In addition, we have identified opportunities for concurrency in the internal systems such as the block and inode caches.

The model that has been presented in this chapter is an abstract view of the system, encapsulating the overall structure and communication between processes. For a complex system, such as this one, this abstraction can be rather disconnected from the details of the actual implementation. Although the model shows that a matching implementation will confirm to certain behavioural properties, it isn't clear that the server as a whole is free from other common concurrency issues such as data race hazards or timing-related bugs such as TOCTOU. This stems partly from the shared nature of the data on the underlying storage devices which makes up the file system.

5.8.1 Block Ownership

There are three main types of blocks that are stored in the MINIX file system: bitmap blocks, inode blocks, and data blocks. The bitmap blocks reside at the start of the file system and are used to indicate which of the inodes and data blocks have already been allocated, and which are free. Inode blocks contain a sequence of inode data structures, each representing a file or directory in the file system. The data blocks are used to store everything else: the contents of files, the list of files contained in a directory, and the maps that describe where the contents of files are stored.

Since these blocks are shared across the entire file system, it is important to ensure that two requests will not attempt to access the same block concurrently except under controlled circumstances. Understanding this, it might be tempting to introduce a mutual exclusion policy for blocks, using process-based data servers or some form of explicit locking. However, a more thorough understanding of how the blocks are being used by each of the processes in the system leads to the insight that such concerns are (mostly) unnecessary in the current design.

Each block in the MINIX file system can be seen to have a single "owner", or process that will access the block. The observation of this ownership allows us to shift the responsibility for the integrity of those blocks to the owning process. For example, the bitmap blocks are (indirectly) used by both the I/O and FS server to allocate and free data blocks and inodes, but the blocks are only read or written to by the allocation server. Because of this, the allocation server can manage mutual exclusion for the blocks. In the current design the allocation server allows only a single request at a time, so this is trivially achieved. Inode blocks can be treated similarly; despite the fact that each block contains multiple inode entries, the blocks are only ever read from or written to by the inode server process where such concurrency as it exists is kept under control.

The ownership of data blocks is slightly less obvious, since they encompass data that could be used by either the I/O or the FS server. Each block in the file system contains data for one file or directory. This makes it possible to view each file or directory as the owner for all of its constituent data blocks. Any operations that access the contents of a directory or add or remove links are performed by the FS server, which handles requests sequentially.

The `truncate` system call, on the other hand, can be used to change the contents of a file. If the underlying file is not currently open by another process this operation should be safe since the call is handled by the FS server no process will be able to open the file until the call is complete. However, if the file is already opened, then care must be taken to ensure the contents are not corrupted. This can be handled in the same way the FS server currently handles open and close requests. If the file is currently open, then the `truncate` call can be delegated to the I/O server, the designated owner of the contents of files.[7]

[7] A similar issue can occur when the `read` system call supports reading the contents of directories, an operation that is supported in some UNIX operating systems where directory entries are fixed-length. In these case the FS server must be responsible for handling these calls. UNIX does not provide guarantees about what happens if one process is reading a directory while another creates a file in that directory.

5.8.2 Managing Access to Inodes

A similar issue exists with the inode data structure an I/O system call might make use of the same inode as a system call that is being handled by the FS server. For example, the I/O server might use an inode in order to retrieve the contents of a file. At the time, the FS server might need to update the link count or the permissions of the file (both stored in the inode) as a result of a system call.

The majority of the operations on inodes are performed by the FS server, so we can assume that it will need to access most of the fields. The I/O server, on the other hand, needs access to the size counter, the list of data blocks, and the modification and access times (so they can be updated). If these sets are disjoint, then each server can be given only the portion of the inode that it needs, rather than the entire data structure.

The `stat` system call returns information about a file or directory, including its current size, and is handled by the FS server. For files that aren't open for writing, the server can read the inode and return all of the relevant information. However, for a file that has been opened for writing, only the I/O server can know how large it is at any given time; another process might be actively writing data to the file. Rather than returning the value stored on disk or adding a lock for the size field, the FS server can instead ask the I/O server for the current size of the file.

When an inode is written to disk, the two sets of attributes from the I/O and the FS server must be combined. This can be accomplished by the implementation of the `sync` and `close` system calls, handled by the FS server. This has a nice symmetry—when the file is opened, the FS server will read the inode from disk and send portions of it to the I/O server. When the last process closes a file, the FS server can request the new values and commit them to disk via the inode table.

The dependency between the `stat` system call and the `write` system call might not be obvious in a lock-based solution, but the communication dependency in our process network is explicit.

5.8.3 FS Server

The sequential behaviour of the FS server helps to prevent a class of software bugs, called "time-of-check-to-time-of-update" (TOCTOU) race conditions. Although normally viewed from a security context in the implementation of privileged user (setuid) programs, this is a class of bugs where a condition is checked and then, at some later time, action is taken based upon that check. If the state of the system can change

between the check and the update, this could result in incorrect behaviour or worse. An example of this can be seen in the inode link count, used to track the number of times a file is linked in directories on the file system.

The procedure for the *link* system call is as follows:

1. Fetch the inode of the file to be linked
2. Ensure the link count is not already the maximum value
3. Fetch the directory in which the new link will be created
4. Check to see if the target file name already exists
5. Attempt to add the link to the directory
6. If successful, update link count and flag the inode as dirty

The problem is with the second step, where the current value of the link count is checked to ensure it is not already the maximum allowed value. If the file server allowed concurrent system calls, then after this check is performed, another process could come along and add a link to the same file. Afterwards, the original system call would proceed, causing the link the count to overflow, possibly wrapping around to zero and resulting in a corrupted file system.

The most obvious way of solving this sort of problem while maintaining the existing POSIX interface for programs is to introduce some mechanism for mutual exclusion, so that the original *link* system call could ensure it has exclusive access to the inode. This could be done using an explicit locking mechanism such as a mutex. However this would introduce *implicit* channels of communication in an otherwise disciplined process network, reducing the clarity of the system.

A process-oriented solution that would be possible in occam-π involves encapsulating each inode in a server process that exposes shared channel ends to its clients. In order to interact with the process it would be necessary to claim the channel end, preventing other processes from interacting with a given inode. Since the type system would ensure channels ends are properly claimed, it would be possible to ensure this type of race condition does not occur.

Although this technique helps to address this particular type of race condition, the POSIX file system standard has similar issues at multiple levels of abstractions (mounting system calls must be mutually exclusive with path name resolution, for example). Even using a process-oriented approach, the dependencies between different

types of mutual exclusion starts to resemble the fine-grained layered locking that is present in the Linux kernel. Although beginning with the existing MINIX file system implementation helped to bootstrap this research, it constrained us to working with a large amount of legacy inherited from the file system format and API itself. It would be interesting to consider one would design a fully-functional file system format and API that were more amenable to process-oriented software design and see how those lessons might be applied in the domain of distributed file systems and cloud computing.

5.9 Summary

In this chapter we have introduced an architecture for a concurrent file server that allows for a high level of concurrent file I/O while serializing all other file system requests. This design was developed in conjunction with a formal model of an abstraction of the communication behaviour between the various servers, and checked using model checking tools to verify properties of the system. An implementation of this architecture was presented, written in the Go programming language, which allows a program to access a MINIX file system in a way that is idiomatic.

This helped to illustrate the ways in which formal methods and tools for model checking can influence the development of software, even when the programming language itself is not able to provide strict guarantees about the systems constructed using it.

Chapter 6

Conclusions

This thesis sought to identify the ways in which formal methods of concurrency can interact with and support the development of concurrent software. Through a series of case studies and analysis, it has shown how a concurrency model can serve as a basis for a system's architecture, and how it is possible to create a model of an abstraction of the system and use it to verify properties of the system. Furthermore, all of this is possible even if the concurrency that is provided by the programming language is more relaxed than the formal model, or if the program is not structured using a single abstraction. This clearly illustrates that there is much to be gained by further integrating formal models of concurrency into the development of concurrent software.

6.1 CSP-Style Concurrency

Concurrent programming is generally considered to be difficult, in spite of the fact that there are countless abstractions that seek to make it easier to build and understand concurrent systems. Some of this perception stems from the fact that concurrency brings a unique set of issues that are not present in sequential programs. Since multiple processes may interact with each other during their execution, there are a potentially large number of potential execution paths in the system. This can result in systems that compute different results depending on which ordering of events happens to occur. When resources are shared between multiple processes it is possible for a system to deadlock or to enter a state of starvation, where no work can be accomplished.

Communicating Sequential Processes is a formal model for describing and reasoning about concurrent systems. Processes are defined by describing their behaviour in terms of the communication events in which they might participate. Systems are built

by composing multiple processes together and describing the ways in which they will synchronize during communication. A process can have varying behaviour influenced by the external environment, such as other processes being willing to synchronize on a given event, or through non-deterministic choice, representing a decision that might be made by the process. These abstractions make it possible to describe a wide range of concurrent systems and to reason about the way they might behave.

The CSP process algebra was developed alongside (and in conjunction with) the Transputer microprocessor architecture, which was programmable using the occam programming language. Naturally, there is a correspondence between the functionality that is present in occam and that which can be described in CSP without a large amount of abstraction. This has enabled the research and introduction of extensions that have enabled new functionality while still conforming to the formal model. occam-π and similarly structured libraries for other languages have been the tool of choice for building CSP style concurrent programs.

The Go programming language also integrates CSP-style process and channels. Like occam-π, Go is an imperative programming that makes it easy to define and spawn new processes and connect them together using channels. Channels are first-class citizens of both languages and can be communicated over channels, enabling networks of processes that change dynamically during the programs runtime.

However, there are some slight differences in the structure and semantics of the languages. occam-π is based around process definition and explicit composition using `SEQ` and `PAR` blocks (which implements the CSP parallel operator). Shared channel ends must be claimed before communicating, making it possible to turn a shared channel end into a temporary private channel between two processes. Go lacks the structured composition of (and subsequent joining of) a block of processes, making it more difficult understand the lifetime of a process and to automatically prevent some potential concurrency issues by restricting the way in which values are shared among processes.

One of the costs of this departure is that only a small subset of programs can currently be modelled in the CSP process algebra without large amounts of abstraction. Although it would be possible to build an occam-style library for Go, but it is not entirely clear what benefit it would provide. There is potentially more to be gained by being able to automatically model certain idiomatic patterns of communication in order to highlight the ways in which the CSP process algebra can be used to support the development of programs.

6.2 Process-Oriented Design

Process-oriented programming is a style of programming where systems are built by composing concurrent processes. The occam family of programming languages, in a way that is similar to the "everything is an object" mentality of Java, treat all computations as processes that can be composed together. The rules for this structure and composition come from the CSP process algebra and are limited to those constructs which can be formally verified. As a result, it is possible to automatically generate a CSP model for a process-oriented program that is written in occam.

The Go programming language provides support for writing process-oriented programs, although it is a less granular approach. A new process can be spawned from any function or method call and channels enable communication between multiple concurrent processes. This is similar to the way the language supports object-oriented programming, by including support for creating rich object definitions but not requiring all programs to be written in an entirely object-oriented manner.

In this way, process-oriented design provides a framework for considering the way in which concurrent processes might be involved in a computation or system. In our work we examined the occam web server, written as a fully process-oriented program, in order to determine how a similar structure could be obtained in Go. This design provides benefits (such as being able to control the rate of flow of requests through the server), but did not directly support the collaborative generation of responses.

There are many different kinds of concurrent web servers, but none that are able to combine the compositional nature of process networks with explicit concurrency in the generation and transformation of the response. By intentionally considering the ways in which a web server could be built using a collection of concurrent processes, we were able to evolve the design and find an appropriate level of abstraction that is surprisingly flexible and powerful. The resulting web toolkit is particularly interesting because, although it does not require process networks for the initial stage of response generation, a new process network is dynamically created for each request.

6.3 Modelling CSP-Style Concurrent Systems

There are benefits of using occam-π directly for building programs. The compiler is able to verify automatically certain structural patterns that help to prevent common concurrent bugs. In addition, there has recently been experimental support for creating a CSP model directly from the source of the program. CSP has been used to

model cryptography protocols, hardware designs, and common patterns of process definitions. These models can then be used to prove properties of the systems.

In a complex concurrent system such as a file system server, it is necessary to have a clear understanding of the way in which the system is structured and the way in which it behaves. A model of the abstract communication within the system can be used as a basis for understanding and improving the design of such a program. The creation of such a model can come for free when using occam-π, but there has been little work thus far that shows how such a model can be constructed from a sequential imperative program.

Having this design makes it possible to explore the concurrency that is possible within the system. A monolithic process can be replaced with a version composed of smaller processes in order to expose more concurrency. The model can be used to verify these design decisions and create a framework for iterative refinement of concurrent program structure.

6.4 Future Work

The primary problem with using process-oriented design and CSP-style concurrency in Go is that there is no direct correspondence between the semantics of concurrency in the programming language and those of the formal model. In Go it is trivial to create a program where multiple goroutines have access to the same portion of shared memory, but utilize channels to synchronize access to this resource. Representing this program in CSP, however, requires an abstraction that may not accurately represent the program.

Similarly, in Go, it is common to use channels for mutual exclusion, but there is no way to indicate that a communication should be performed with movement semantics, where the sending process no longer has access to the value once it has been communicated. Having an explicit transfer of ownership rather than a convention would potentially make it easier to identify programming errors automatically, preventing a class of potential race hazards. The mobile data semantics used by occam-π could serve as a model for this, but it is not clear how substantial of a change this would be, or if this sort of feature would be incompatible with some aspect of Go.

The cost of communicating via channels in Go is still an order of magnitude higher than the equivalent occam-π program, at least for a simple process network such as `commstime`. In part, this is due to the design decision that were made during the

development of the occam programming language and the transputer microprocessor instruction set and micro-code. It was possible to leverage this industrial effort during research and development of the CCSP runtime for the occam-π programming language, which may account for some of the difference in performance. This prior research should be closely examined in the context of the Go runtime in order to determine how it could be used to improve performance or introduce reasonable restrictions that enable additional functionality.

One of the major challenges moving forward is that new developers come to the Go programming language from the world of threads and mutexes, but are able to adapt to nature of synchronous channel-based communication. However, for these developers, there is little connection to the large body of work that has been done in and around the CSP community. This means they will encounter many problems that have long been 'solved', and will likely repeat many of the mistakes from our past. It is vital that we continue to make connections between Go and CSP to show how the work done with occam-π and the formal model itself can be used to guide the development of Go programs.

In addition it is important to leverage the widespread popularity of the Go programming language in order to better understand the use of CSP style concurrent program structure. By monitoring the types of programs that are developed and by investigating their program structure, we will gain more insight into the ways that channels and processes can be used to build complex programs. This might help to identify situations where the assumptions or restrictions of occam-π run counter to patterns that are commonly used, highlighting an opportunity for clarifying research or changes to the underlying assumptions. It is possible that there is a higher level of abstraction for the process algebra that might be able to be used more directly with languages that include support for CSP-style concurrency in the less structured manner of Go.

6.5 Summary

There have been decades of research into the development and modelling of process-oriented programs written using the occam-π programming language. Recently there has been a resurgence of new concurrent programming languages that provide CSP-style concurrency, including Go. It is vitally important that we continue to explore ways of bringing together the practice of writing concurrent programs with theoretical models of concurrency.

Appendix A

Introduction to the Non-Concurrency Aspects of Go

The Go programming language was introduced in 2009 by a small team of distinguished engineers at Google. It is a compiled, garbage-collected, statically-typed programming language with native support for concurrency. Programs written in Go compile quickly, particularly when compared to C and C++. By avoiding C-style include files and libraries, dependencies analysis is simplified. Despite being a statically-typed language, Go achieves a more dynamic feel through type inference, a form of "duck typing" using interfaces, and supporting object oriented programming without classes or explicit type hierarchies.

In this thesis, Go is used to implement a variety of CSP-style concurrent systems. In particular, Chapter 3 details various ways of utilizing CSP-style concurrency in Go. This section provides an introduction to the major non-concurrency features of the programming language through a series of examples.

A.1 Hello World

Go uses a brace-delimited syntax that is similar to the C-family of programming languages. Although semicolons can be used to terminate statements, the compiler automatically performs semicolon injection. As a result, most Go programs will be written without a single terminating semicolon. The following is a standard "Hello World!" program written in Go:

```
package main

import "fmt"

func main() {
  fmt.Println("Hello world!")
}
```

Each source file in a Go program must belong to a package, which is the only form of name-spacing in the language. Executable programs (called commands) are placed in the *main* package. Other packages are imported using a fully qualified import path, such as *encoding/xml*. This program imports the *fmt* package, which provides facilities for the formatting and printing of strings. The entry point of a Go program is the `main` function, which takes no parameters and returns no result.

This program consists of a single function call to `fmt.Println`, which will print the string to standard output, followed by a newline character. It is convention in Go to access any imported functions and types through their package names in order to avoid ambiguity. Although the language provides a way to import functions into the current package's namespace, this is highly discouraged as it hinders readability.

A.2 Variable declarations

A new variable can be introduced using the `var` keyword. This creates a new value that is initialized to the "zero value" for the given type. For numeric types, this value is a zero of the appropriate type: 0 for integers, 0.0 for floating point numbers and $0 + 0i$ for complex numbers. Boolean values are initialized to $false$, and the zero value for strings is the empty string.

To declare a new string variable called `msg` and then initialize it to some value:

```
var msg string
msg = "Hello world!"
```

Alternatively, a variable can be initialized during declaration:

```
var msg string = "Hello world!"
```

If the type of the variable being declared can be inferred, it can be omitted from the declaration entirely.

```
var msg = "Hello world!"
```

However in these cases, the short form of declaration using the `:=` operator is normally used:

```
msg := "Hello world!"
```

A.3 Function declarations

A function declaration begins with the `func` keyword, followed by the name of the function, a list of parameters in brackets, and then a list of the return values of the function (if any). For example, a function that takes in a string and constructs a greeting message based on that string might look like this:

```
func greeting(name string) string {
  return fmt.Sprintf("Hello %s!\n", name)
}
```

A function in Go can return more than one value, such as the following function that computes the integer division of two numbers, with remainder:

```
func div(x, y int) (int, int) {
  div := x / y
  rem := x % y
  return div, rem
}
```

In this function there are two integer parameters named `x` and `y`. Since their types are the same, we can elide the first type in order to conserve space and make the function definition easier to read. The multiple return types are specified in brackets, showing that the function will return two `int` values.

A variable that is declared but is never used will cause a compile-time error. This can be problematic when one or more return values of a function call are not needed. However, Go provides the "blank identifier", an underscore, which indicates that a value is being explicitly ignored. For example, to get just the first return of the `div` function, you would write the following:

```
d, _ := div(7, 3)
```

A.4 Arrays and slices

Go provides support for arrays of any type, but requires that the length of the array be fixed. For example, `[10]int` is an array of 10 integers and is a distinct type from `[20]int`. As a result, a function that asks for an array can only be given one of the correct type (including length). The zero value of an array is an array of the correct size, with each element being initialized to the zero value for the given type. However, arrays are not widely used in Go programs due to their restrictive nature.

Go includes a higher-level data structure that is built on top of arrays, called slices. A slice is a reference to a contiguous segment of an array, containing a numbered sequence of elements. A slice type is similar to an array, only the length is omitted, i.e. `[]int` is a slice of integers. These allow code to be written where the length of the sequence may not be known ahead of time.

A slice can be created by "slicing" an array (or another slice) using the following expression:

```
nums := numarray[low:high]
```

`low` is the first element to be included in the slice, and `high`-1 is the last element to be included. The number of elements in the resulting slice is the difference between `high` and `low`. Either of these bounds can be omitted from the expression, with the lower defaulting to 0 and the upper defaulting to the length of the array or slice. The elements of the new slice are indexed starting at 0.

The slice data structure can make the segmentation of an array into smaller parts much easier. For example, the initial step of a merge sort could be accomplished by creating two sub-slices:

```
mid := len(nums) / 2

left := nums[:mid]
right := nums[mid:]
```

The built in function `len` returns the length of a slice or array, and can be used to determine an appropriate splitting point. The resulting slice `left` will contain all elements below index `mid`, while the `right` slice will contain the remaining elements beginning with the element at index `mid`. Each of these slices is backed by the `nums` array, so changes to the slice will be reflected in the original array.

Another common operation using slices is the accumulation of an unknown number of elements. For example, consider reading a list of numbers from standard input and

putting them into a slice so they can be passed as arguments to a function. This can be accomplished using the `append` built in function, which appends elements to the end of a slice. Such a loop might look like this:

```
var nums []int
for {
  num := getNumber()
  nums = append(nums, num)
}
```

The initial value of `nums` is `nil`, which is the zero value for slices. If there is not enough room in the slice for the new values, `append` will automatically create a new slice that is sufficient to hold the new values. The `append` function returns a slice, which may or may not be the same slice that was initially passed as a parameter. This is because the append operation may have required the allocation of a new array and slice.

The `make` function allows a slice to be manually allocated with a given length and capacity. For example, the following will create a new slice of integers that initially has a length of 10, but a capacity of 1024:

```
nums := make([]int, 10, 1024)
```

The first 10 elements will be initialized to the zero value, but the underlying array has space for more values. This ensures that 1014 more elements can be appended without requiring expensive memory allocation.

A.4.1 Variadic functions

When a function accepts multiple values of a single type, they can be passed using a slice. For example the following function can be used to add several numbers together:

```
func add(nums []int) int {
  total := 0
  for _, num := range nums {
    total = total + num
  }
  return total
}
```

However calling that would require constructing a new slice each time the function needs to be called:

```
total := add([]int{1, 2, 3, 4})
```

To help with these situations, Go allows the last argument to a function to be a variadic type. This enables a more natural style of function invocation while still providing an easy mechanism to access the arguments using a slice. A type such as `int` can be made variadic by adding `...` to the start.

```
func addMany(nums ...int) int {
  total := 0
  for _, num := range nums {
    total = total + num
  }
  return total
}
```

In this example the variable `nums` is still represented as a slice of integers. It's possible to get the length of the slice and iterate the list of arguments. The change is subtle, but enables a more natural style of function invocation:

```
total := addMany(1, 2, 3, 4)
```

Variadic functions are not widely used, but they are useful (for example) in the implementation of the `fmt.Sprintf()` function and other similar functions that retain the variadic invocations from their UNIX heritage.

A.5 Pointer types

Go provides support for pointer types, but takes a rather different approach than C and C++. Most importantly, the language itself does not support any form of pointer arithmetic. This immediately removes an entire class of bugs that comes from dereferencing pointers to arbitrary portions of memory. The notion of a pointer is a part of the type system, where a pointer type is distinct from its base type. This prevents a pointer from being used where a non-pointer is expected, etc. Here is a (type annotated) example that shows how they can be used:

```
var a int = 5
var b *int = &a

*b = *b + 3
fmt.Println(a) // prints 8
```

This example uses the address operator `&` to obtain the address of the variable `a` and store it in `b`. This can then be used to access or alter the contents of `a` using pointer indirection (i.e. `*b`), as shown. The important thing here is that the type

system helps to ensure that the pointer value cannot be altered arbitrarily. The zero value of a pointer type is `nil`, which must be checked by a program before attempting to use pointer indirection. If a program uses pointer indirection on a `nil` pointer, the program will crash (called a panic) at runtime.

A.6 Structured data

Go allows you to define structured data types, allowing you control over the memory layout in your application. This can have a particular benefit in data-heavy applications when the alignment of a data structure falls on word or cache boundaries. Structured data is normally defined as new data type using the following syntax:

```
type Node struct {
  value int
  next  *Node
}
```

This defines a new type called `Node` which is a data structure representing a node in a linked list. It contains two elements: the first is called `value` and holds an integer, the second is called `next` and is a pointer to a `Node`. The elements of a structured data type can be accessed by specifying the name of the field in a selector expression, such as `head.value`, which corresponds to the field `value` in the value `head`.

Code can access a structure using the value type `Node` or a pointer to that type `*Node`. Since most data structures written in Go programs are designed to be altered, the latter is the more common case. Go code will rarely make use of values of the type itself, since they cannot be altered. In fact, whenever a value type is used in an assignment, or as a parameter in a function or method call, the data is copied. For a large data structure this can be quite expensive, so pointers are typically used instead.

A pointer can be obtained by using the address operator `&` on an existing addressable value. Alternatively, a new instance and pointer can be created using the `new` function. Given a type T, the expression

```
node := new(Node)
```

will return new value with the type `*Node`. When a value is created in this way, each of the fields of the data structure will be initialized to their *zero value*. For numeric types this is 0, for strings it is the empty string, and `nil` for any reference

type (maps, slices, arrays, pointers). These zero types are composeable, such that the zero value for the `Node` type is a struct containing the elements 0 and nil.

A pointer to a structure data type can be used in the same way as the value type, allowing a selector to access the fields and methods of the data structure.

Go code typically uses composite literal syntax, which is a way of constructing new structure type values without needing to explicitly call `new`. For example, a new tail element `Node` could be created as follows:

```
node := Node{17, nil}
```

One can use the address of operator to convert such a value into a pointer, often combined into a single operation:

```
node2 := &Node{17, nil}
```

A.7 User-defined data types

User-defined data types are not restricted to structured data types. In fact, a new type can be created that has any of the built-in data types as a concrete representation.

Consider for a moment a function that takes some sort of enumerated constant as a parameter, such as:

```
func DoSomething(flag int) {
  switch flag {
  case 0:
    // do something here
  case 1:
    // something else
  }
}
```

Depending on the integer that is passed into the function, the function may behave in slightly different ways. Only convention and explicit error-checking prevent invalid values from being passed into the program and causing unexpected behaviour. User-defined types can help to alleviate this situation in a type-safe way.

For example, we can define a new type called `flag` that has the same underlying representation as the `int` type:

```
type flag int
```

We could then define a series of constant values that have this type.

```
const READ flag = 1
const WRITE flag = 2
```

By specifying the type of the variable, we cause the compiler to coerce the integer values to our new type. Alternatively we could explicitly cast the integers to the new type. Now, we can re-define the `DoSomething` function such that it only accepts the new type.

```
func DoSomething(options flag) {
  switch options {
  case READ:
    // handle reads
  case WRITE:
    // handle writes
  }
}
```

In order to pass an invalid flag to the function, code would need to explicit create or cast to get a new `flag`. However, when combined with package visibility, this suddenly becomes impossible.

A package only exports those variables, types, field names, and method names that begin with a Unicode upper case letter. These exported elements are visible when the package is imported, but anything else is unaccessible.

In this case the only things exported by this package are the two constant values and the `DoSomething` function. As a result, it is impossible for someone to call the function without specifying one of the pre-defined acceptable flag values. Since the underlying type is not exported, no other package can construct new values for it. This may be a slightly paranoid example of how user-defined types can be useful, but it remains a good illustration of a powerful feature.

A.8 Defining methods

Go supports the definition of methods on any user-defined data type. A function is defined as a method by including a receiver in the function signature. The receiver is the thing on which the method is being invoked, and can be either a value type or a pointer type. Normally the receiver is a pointer, since many times the action of a method will be some change to the object itself. Let's extend the flag data type with a method called `String` that will be used to convert a flag constant into a readable string representation.

```
func (f flag) String() string {
  switch f {
  case READ:
    return "READ"
  case WRITE:
    return "WRITE"
  }

  return fmt.Sprintf("UNKNOWN_FLAG %d", f)
}
```

The signature for a method (as opposed to a function) includes the receiver name and types in brackets before the name of the function, while the rest of the signature remain unchanged. This makes it easy to see on what type of object a method will be invoked on without requiring an additional level of wrapping or scoping, as is required in Java. This method performs a case switch on the known values of the flag, and has a default case that handled a possible unknown flag. The compiler is not currently able to perform analysis to determine if the case switch is exhaustive (even with a default branch), so the compiler will complain if there is not a guaranteed return value.

With no other changes to the program code, this exported method may be invoked on the constants that are defined as part of the package. For example, client code might wish to log the options that are being passed in an invocation of `DoSomething`:

```
log.Printf("Calling DoSomething() with flag %s", READ)
```

This makes it much easier to write code that may need to display some information without having to directly alter the messages being logged. In this case, the *log* package uses the *fmt* package, which knows to look for the `String` method when it encounters a `"%s"` in the format string.

A.9 Interface types

Interfaces are a way to group together methods on objects and introduce subtype polymorphism in the language. An interface is defined similarly to a struct, but with a list of method signatures rather than fields. For example, the Stringer interface which is used by the fmt package is defined as follows:

```
type Stringer interface {
    String() string
}
```

Interfaces are implicit in Go, meaning that any object which implements a `String` method that takes no arguments and returns a string is considered to implement the `Stringer` interface. Once an interface is defined, it can be used as a type for variables, function parameters or returns, and even in fields of structured data. This allows any type that satisfies the interface to be used wherever the program calls a `Stringer`. We could write a more general logging function that takes a `Stringer` and a message to be printed and takes care of the details:

```
func debugparam(name string, param fmt.Stringer) {
  log.Printf("Parameter '%s' is '%s'", name, param)
}
```

We can then invoke it, passing the given flag in as the second argument.

```
  debugparam("flag", READ)
```

At compile time, the type of the second argument is examined to ensure that it implements the `fmt.Stringer` interface.

Interfaces can also be used as a grouping tool by acting similar to a tagged union. Say you want to define a `Message` data type that can be any of several different messages. If the data included in each message is the same, then a simple data type definition will do. However, if the data is different between the message types, then interfaces can be used to provide a sort of super-type. We can define two data types for different messages that might occur in a chat server application:

```
type ChatMessage struct {
  dest string
  msg  string
}

type QuitMessage struct {
}
```

At some point we may need to write a function that simply takes in a message, without specifying a particular message type. This can be accomplished by adding a marker method that both message types can implement, and then requiring this method in an interface type. The method doesn't need to do anything, but its presence will signal to the type checker and the program that either individual message type can be used wherever the interface type is required.

```
type Message interface {
  isMessage()
}
```

Now we can add the code that implicitly implements this interface, by defining the `isMessage` method on each of them. The methods take no arguments, perform no work and return no results, so they can be written on a single line.

```
func (m ChatMessage) isMessage() {}
func (m QuitMessage) isMessage() {}
```

The code in the chat server can now be written in a slightly more general way. For example the `HandleMessage` function signature might be:

```
func HandleMessage(m Message)
```

The problem with this approach is that within the function that accepts a `Message`, it will only have access to the `isMessage` method (and then only if it is defined in the same package). In order to actually do anything with the message, the program will need to perform a type assertion at runtime. In Go, a type assertion is performed with the following syntax:

```
chatMsg, ok := message.(ChatMessage)
```

This code determines whether or not the `message` value is actually value of type `ChatMessage`. If so, the first return value is the same object with the correct type, and the second return value is the boolean value true. If the type assertion fails, the first return will be the zero value for the appropriate type, and the second return value will be the boolean value false. This is often enough for some programs, but in the case of our chat server we may have dozens of messages. For that, Go provides a special form of the switch statement called a type switch.

```
func HandleMessage(m Message) {
  switch m.(type) {
  case ChatMessage:
    // Handle the chat message
  case QuitMessage:
    // Handle the quit message
  default:
    // Handle any other message type
  }
}
```

A type switch is similar to a normal expression switch, only it requires a type assertion on the expression to the `type` type. Each case statement within the switch must match a valid type, and will be invoked if the type of the expression matches. If needed, the result of the type assertion can be assigned to a variable and then used in the case statements. The type of this value will match the branch that is executed.

Although interfaces provide a way to group different data structures together, they still require a bit of runtime checking in order to be handled correctly. Luckily the mechanisms that are provided are type-safe and have a predictable behaviour.

Interfaces are more conventionally used in Go to provide certain functionality to given clients. If some code requires an object that can be turned into some native representation as a string, then you can use the `fmt.Stringer` interface.

One of the more useful interfaces that new Go programmers encounter is the `io.Writer` interface. It provides an abstraction over things that can be written to, and are defined by the *io* package as follows:

```
type Writer interface {
  Write(p []byte) (n int, err error)
}
```

The documentation specifies the behaviour expected of any type that implements either of these methods. For example, the write method will attempt to write the bytes from `p` to the writer. The method returns the number of bytes that were actually written as a result of the call, along with any error that might have been encountered. This turns out to be really powerful, allowing code to be designed without regard for what is actually implementing the `io.Writer` interface. For example, the same package defines a function that can be used to write a string:

```
func WriteString(w Writer, s string) (n int, err error)
```

The `os.File` type and the `net.Conn` type both implement the `io.Writer` interface, representing files on the host file system and network connections. As a result, the code to write a string to either a file or a network connection is exactly the same. The writer is passed as the first argument, and the actual conversion of a string to a slice of bytes is performed behind the scenes by the `WriteString` function. Similar abstractions exist for things you can read from, using the `io.Reader` interface. There can be several different types with drastically different algorithms and behaviour that all implement such a common interface. Since the `compress/gzip` package defines a type that implements the appropriate interface, reading from a gzipped file from the file system is the same any other file.

Interfaces are compositional, meaning you can create a new interface that is the composition of two different interfaces. For example, the `io.ReadWriter` interface represents something that can both be read from and written to, such as a file or bidirectional network connection. This can be accomplished by listing any interfaces that are required to implement a new interface:

```
type ReadWriter interface {
  Reader
  Writer
}
```

Here, any type that implements both the `Reader` and the `Writer` interfaces can be used wherever a `ReadWriter` is required.

A.10 Inheritance through composition

Go does not have the traditional class-based hierarchy that is present in languages like Java. It does, however, support a form of inheritance that is similar to the interface composition we have already seen. A structured data type can contain an anonymous field of another type. This type of composition is typically called embedding. A data type with an embedded field will include the fields and method sets of the embedded field, with the language performing automatic delegation between the outer type and the embedded type.

This can be useful when creating wrappers around existing types that need to behave slightly differently. Consider an interface that contains three different methods:

```
type FooBarBazer interface {
  Foo(x int)
  Bar()
  Baz()
}
```

Say we want to create a new `FooBarBazer` from an existing one that will sometimes panic when the `Foo` method is called. We can create a new structure data type that will embed a `FooBarBazer`, and define a new `Foo` method.

```
type BrokenFooBarBaz struct {
  FooBarBazer
}

func (fbz BrokenFooBarBaz) Foo(x int) {
  if x == -1 {
    panic("Foo() called from a broken FooBarBaz")
  }
  fbz.FooBarBazer.Foo(x)
}
```

This new method can even invoke the original, by selecting using the name of the embedded type. The advantage of this sort of composition is that the wrapping type does not need to define wrapper methods around each of the methods that are present in the embedded type. Embedded fields work in the same way, with any conflicts between outer fields shadowing those in the embedded type (but with those still being accessible using the type name).

Appendix B

File Server Models

B.1 Sequential Model

```
module Sequential(MAX_PROC, MAX_PROC_FILES, MAX_FD)
    -- @param MAX_PROC, Total number of processes to model
    -- @param MAX_PROC_FILES, Maximum number of open files per process
    -- @param MAX_FD, Maximum number of open file desciptors in system

    exports

    ------------------------------------------------------------------------------
    -- Channels and data types
    ------------------------------------------------------------------------------
    Pid = {0 .. MAX_PROC-1}  -- set of user processes
    Fd = {0 .. MAX_FD-1}     -- set of file descriptors

    datatype Syscall = Fopen | Fclose | Fread | Fwrite | Chdir
    datatype Calltype = Call | CallArg.Fd | Ret | RetVal.Fd

    channel syscall : Pid.Syscall.Calltype

    -- The events for a given set of process IDs
    syscall_events(pids) =
        {| syscall.pid.Fopen.Call, syscall.pid.Fopen.RetVal,
           syscall.pid.Fclose.CallArg, syscall.pid.Fclose.Ret,
           syscall.pid.Chdir.Call, syscall.pid.Chdir.Ret,
           syscall.pid.Fread.CallArg, syscall.pid.Fread.Ret,
           syscall.pid.Fwrite.CallArg, syscall.pid.Fwrite.Ret
         | pid <- pids |}

    ------------------------------------------------------------------------------
    -- Filesystem specification
    ------------------------------------------------------------------------------
    FSServer = let
        Ready(open) =
            -- open a file
            card(open) < MAX_FD & syscall?pid.Fopen.Call ->
            syscall.pid.Fopen.RetVal?fd:diff(Fd, open) -> Ready(union(open, {fd}))
            -- close an open file
         [] syscall?pid.Fclose!CallArg?fd:open ->
            syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))
            -- read from an open file
         [] syscall?pid.Fread!CallArg?fd:open -> syscall.pid.Fread.Ret ->
```

```
            Ready(open)
            -- write to an open file
        [] syscall?pid.Fwrite!CallArg?fd:open -> syscall.pid.Fwrite.Ret ->
            Ready(open)
            -- change working directory
        [] syscall?pid.Chdir.Call -> syscall.pid.Chdir.Ret -> Ready(open)
    within Ready({})

    ------------------------------------------------------------------------
    -- File descriptor regulator, ensuring proper behaviour
    ------------------------------------------------------------------------
    FDRegulator(pid, open_fds) =
        card(open_fds) < MAX_PROC_FILES &
        syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd ->
        FDRegulator(pid, union(open_fds, {fd}))
     [] syscall.pid.Fclose.CallArg?fd:open_fds -> syscall.pid.Fclose.Ret ->
        FDRegulator(pid, diff(open_fds, {fd}))
     [] syscall.pid.Fread.CallArg?fd:open_fds -> syscall.pid.Fread.Ret ->
        FDRegulator(pid, open_fds)
     [] syscall.pid.Fwrite.CallArg?fd:open_fds -> syscall.pid.Fwrite.Ret ->
        FDRegulator(pid, open_fds)
     [] syscall.pid.Chdir.Call -> syscall.pid.Chdir.Ret ->
        FDRegulator(pid, open_fds)

    FDRegulators = (|| pid:Pid @ [syscall_events({pid})] FDRegulator(pid, {}))
    Filesystem = FSServer [| syscall_events(Pid) |] FDRegulators

    ------------------------------------------------------------------------
    -- A process that opens two files and copies between them
    ------------------------------------------------------------------------
    OpenCopyClose(pid) = let
        Open =
            syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd1 ->
            syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd2 -> Copy(fd1, fd2)
        Copy(fd1, fd2) =
            syscall.pid.Fread.CallArg.fd1 -> syscall.pid.Fread.Ret ->
            syscall.pid.Fwrite.CallArg.fd2 -> syscall.pid.Fwrite.Ret -> Close(fd1, fd2)
        Close(fd1, fd2) =
            (syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret ->
            syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret -> Open)
         [] (syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret ->
            syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret -> Open)
    within Open

    -- Only allow one system call at a time
    SeqLimit = syscall?pid?call?calltype -> syscall.pid!call?rettype -> SeqLimit

    -- Actions taken by other processes
    OtherProcs = CHAOS(syscall_events(diff(Pid, {0})))

    ------------------------------------------------------------------------
    -- Use non-determinism to allow any file to be opened
    ------------------------------------------------------------------------
    OpenAny(pid) = syscall.pid.Fopen.RetVal$fd1 -> OpenAny(pid)
    CopySpec = OpenCopyClose(0) [| {| syscall.0.Fopen.RetVal |} |] OpenAny(0)

    SeqSpec = (CopySpec ||| OtherProcs) [| {| syscall |} |] SeqLimit
    SeqFilesystem = Filesystem [| {| syscall |} |] SeqLimit
    syscall_0_events = {| syscall.0 |}
endmodule
```

B.1.1 Assertions: Sanity check

```
include "01-sequential.csp"

instance FS = Sequential(6, 2, 3)
-- Note: this instance has MAX_FD = 6, MAX_PROC_FILES = 2 and
-- MAX_PROC = 3, which is the reverse order to the parameters
-- of Sequential. This is due to a current parser bug in FDR3.

assert FS::SeqLimit :[deadlock free [FD]]
assert FS::Filesystem :[deadlock free [FD]]
assert FS::CopySpec :[deadlock free [FD]]
```

B.1.2 Assertions: Normal

```
include "01-sequential.csp"

instance FS = Sequential(6, 2, 3)
-- Note: this instance has MAX_FD = 6, MAX_PROC_FILES = 2 and
-- MAX_PROC = 3, which is the reverse order to the parameters
-- of Sequential. This is due to a current parser bug in FDR3.

SeqSpec = FS::SeqSpec
Filesystem = FS::Filesystem
OpenCopyClose = FS::OpenCopyClose
syscall_0_events = FS::syscall_0_events

assert SeqSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))

NonSeqSpec = (FS::CopySpec ||| FS::OtherProcs)
assert not NonSeqSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))
```

B.1.3 Assertions: Not enough file descriptors

```
include "01-sequential.csp"

instance FS = Sequential(5, 2, 3)
-- Note: this instance has MAX_FD = 5, MAX_PROC_FILES = 2 and
-- MAX_PROC = 3, which is the reverse order to the parameters
-- of Sequential. This is due to a current parser bug in FDR3.

SeqSpec = FS::SeqSpec
Filesystem = FS::Filesystem
OpenCopyClose = FS::OpenCopyClose
syscall_0_events = FS::syscall_0_events

assert not SeqSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))

NonSeqSpec = (FS::CopySpec ||| FS::OtherProcs)
assert not NonSeqSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))
```

B.2 Dual Servers Model

```
module DualServer(MAX_PROC, MAX_PROC_FILES, MAX_FD)
    -- @param MAX_PROC, Total number of processes to model
    -- @param MAX_PROC_FILES, Maximum number of open files per process
```

```
-- @param MAX_FD, Maximum number of open file desciptors in system

exports

-------------------------------------------------------------------------------
-- Channels and data types
-------------------------------------------------------------------------------
Pid = {0..MAX_PROC-1}
Fd = {0..MAX_FD-1}

datatype Syscall = Fopen | Fclose | Fread | Fwrite | Chdir
datatype Calltype = Call | CallArg.Fd | Ret | RetVal.Fd
channel syscall : Pid.Syscall.Calltype

datatype Intcall = FopenIO | FcloseIO
channel intcall : Pid.Intcall.Calltype

fsio_events =
    {| intcall.pid.FopenIO.Call, intcall.pid.FopenIO.RetVal,
       intcall.pid.FcloseIO.CallArg, intcall.pid.FcloseIO.Ret
     | pid <- Pid |}

-- The events for a given set of process IDs
syscall_events(pids) =
    {| syscall.pid.Fopen.Call, syscall.pid.Fopen.RetVal,
       syscall.pid.Fclose.CallArg, syscall.pid.Fclose.Ret,
       syscall.pid.Chdir.Call, syscall.pid.Chdir.Ret,
       syscall.pid.Fread.CallArg, syscall.pid.Fread.Ret,
       syscall.pid.Fwrite.CallArg, syscall.pid.Fwrite.Ret
     | pid <- pids |}

-------------------------------------------------------------------------------
-- Filesystem specification
-------------------------------------------------------------------------------
FSServer = let
    Ready(open) =
        -- open a file
        card(open) < MAX_FD & syscall?pid.Fopen.Call ->
        intcall.pid.FopenIO.Call -> intcall.pid.FopenIO.RetVal?fd ->
        syscall.pid.Fopen.RetVal.fd -> Ready(union(open, {fd}))
        -- close an open file
     [] syscall?pid.Fclose!CallArg?fd:open ->
        intcall.pid.FcloseIO.CallArg.fd -> intcall.pid.FcloseIO.Ret ->
        syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))
        -- change working directory
     [] syscall?pid.Chdir.Call -> syscall.pid.Chdir.Ret -> Ready(open)
    within Ready({})

-------------------------------------------------------------------------------
-- I/O server specification
-------------------------------------------------------------------------------

IOServer = let
    Ready(open) =
        -- handle an open file request
        intcall?pid.FopenIO.Call -> intcall.pid!FopenIO.RetVal?fd:diff(Fd, open) ->
        Ready(union(open, {fd}))
        -- close an open file
     [] intcall?pid.FcloseIO!CallArg?fd:open -> intcall.pid.FcloseIO.Ret ->
        Ready(diff(open, {fd}))
```

```
        -- read from an open file
     [] syscall?pid.Fread!CallArg?fd:open -> syscall.pid.Fread.Ret -> Ready(open)
        -- write to an open file
     [] syscall?pid.Fwrite!CallArg?fd:open -> syscall.pid.Fwrite.Ret -> Ready(open)
within Ready({})

--------------------------------------------------------------------------------
-- File descriptor regulator, ensuring proper behaviour
--------------------------------------------------------------------------------
FDRegulator(pid, open_fds) =
    card(open_fds) < MAX_PROC_FILES &
    syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd ->
    FDRegulator(pid, union(open_fds, {fd}))
 [] syscall.pid.Fclose.CallArg?fd:open_fds -> syscall.pid.Fclose.Ret ->
    FDRegulator(pid, diff(open_fds, {fd}))
 [] syscall.pid.Fread.CallArg?fd:open_fds -> syscall.pid.Fread.Ret ->
    FDRegulator(pid, open_fds)
 [] syscall.pid.Fwrite.CallArg?fd:open_fds -> syscall.pid.Fwrite.Ret ->
    FDRegulator(pid, open_fds)
 [] syscall.pid.Chdir.Call -> syscall.pid.Chdir.Ret ->
    FDRegulator(pid, open_fds)

FDRegulators = (|| pid:Pid @ [syscall_events({pid})] FDRegulator(pid, {}))

-- The two main servers synchronize only on the messages between them
Servers = (FSServer [| fsio_events |] IOServer) \ fsio_events
Filesystem = Servers [| syscall_events(Pid) |] FDRegulators

--------------------------------------------------------------------------------
-- A process that opens two files and copies between them
--------------------------------------------------------------------------------
OpenCopyClose(pid) = let
    Open =
        syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd1 ->
        syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd2 -> Copy(fd1, fd2)
    Copy(fd1, fd2) =
        syscall.pid.Fread.CallArg.fd1 -> syscall.pid.Fread.Ret ->
        syscall.pid.Fwrite.CallArg.fd2 -> syscall.pid.Fwrite.Ret -> Close(fd1, fd2)
    Close(fd1, fd2) =
        (syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret ->
        syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret -> Open)
     [] (syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret ->
        syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret -> Open)
within Open

-- Only allow one system call at a time
SeqLimit = syscall?pid?call?calltype -> syscall.pid!call?rettype -> SeqLimit

-- Actions taken by other processes
OtherProcs = CHAOS(syscall_events(diff(Pid, {0})))

channel syncio
ConcLimit = let
    FS = syscall?pid.Chdir.Call -> syscall.pid.Chdir.Ret -> FS
      [] syscall?pid.Fopen.Call -> syncio -> syscall.pid.Fopen.RetVal?fd -> FS
      [] syscall?pid.Fclose!CallArg?fd -> syncio -> syscall.pid.Fclose.Ret -> FS

    IO = syscall?pid.Fread!CallArg?fd -> syscall.pid.Fread.Ret -> IO
      [] syscall?pid.Fwrite!CallArg?fd -> syscall.pid.Fwrite.Ret -> IO
      [] syncio -> IO
```

```
    within (FS [| {| syncio |} |] IO) \ {| syncio |}

    -- A specification that allows concurrency
    ConcSpec = (CopySpec ||| OtherProcs) [| {| syscall |} |] ConcLimit
    ------------------------------------------------------------------------
    -- Use non-determinism to allow any file to be opened
    ------------------------------------------------------------------------
    OpenAny(pid) = syscall.pid.Fopen.RetVal$fd1 -> OpenAny(pid)
    CopySpec = OpenCopyClose(0) [| {| syscall.0.Fopen.RetVal |} |] OpenAny(0)

    SeqSpec = (CopySpec ||| OtherProcs) [| {| syscall |} |] SeqLimit
    SeqFilesystem = Filesystem [| {| syscall |} |] SeqLimit
    syscall_0_events = {| syscall.0 |}

    ConcFilesystem = Filesystem [| {| syscall |} |] ConcLimit
endmodule
```

B.2.1 Assertions: Sanity check

```
include "02-dualserver.csp"

instance FS = DualServer(6, 2, 3)

assert FS::SeqLimit :[deadlock free [FD]]
assert FS::ConcLimit :[deadlock free [FD]]
assert FS::Filesystem :[deadlock free [FD]]
assert FS::ConcFilesystem :[deadlock free [FD]]
assert FS::CopySpec :[deadlock free [FD]]
```

B.2.2 Assertions: Normal

```
include "02-dualserver.csp"

instance FS = DualServer(6, 2, 3)
-- Note: this instance has MAX_FD = 6, MAX_PROC_FILES = 2 and
-- MAX_PROC = 3, which is the reverse order to the parameters
-- of Sequential. This is due to a current parser bug in FDR3.

CopySpec = FS::CopySpec
OtherProcs = FS::OtherProcs
SeqLimit = FS::SeqLimit
ConcLimit = FS::ConcLimit
Filesystem = FS::Filesystem
OpenCopyClose = FS::OpenCopyClose
SeqSpec = FS::SeqSpec
ConcSpec = FS::ConcSpec
syscall = FS::syscall
syscall_0_events = {| syscall.0 |}
SeqFilesystem = Filesystem [| {| syscall |} |] SeqLimit

assert SeqSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0))

assert ConcSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))

assert not SeqSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))

assert SeqSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0))
```

```
assert not ConcSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0))
```

B.2.3 Assertions: Not enough file descriptors

```
include "02-dualserver.csp"

instance FS = DualServer(5, 2, 3)
-- Note: this instance has MAX_FD = 5, MAX_PROC_FILES = 2 and
-- MAX_PROC = 3, which is the reverse order to the parameters
-- of Sequential. This is due to a current parser bug in FDR3.

CopySpec = FS::CopySpec
OtherProcs = FS::OtherProcs
SeqLimit = FS::SeqLimit
ConcLimit = FS::ConcLimit
Filesystem = FS::Filesystem
OpenCopyClose = FS::OpenCopyClose
SeqSpec = FS::SeqSpec
ConcSpec = FS::ConcSpec
SeqFilesystem = FS::SeqFilesystem
syscall = FS::syscall
syscall_0_events = {| syscall.0 |}

assert not SeqSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0))

assert not ConcSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))

assert not ConcSpec [FD= (Filesystem [| syscall_0_events |] OpenCopyClose(0))
```

B.3 Internal Servers Model

```
module InternalServer(MAX_PROC, MAX_PROC_FILES, MAX_FD)
    -- @param MAX_PROC, Total number of processes to model
    -- @param MAX_PROC_FILES, Maximum number of open files per process
    -- @param MAX_FD, Maximum number of open file desciptors in system

    exports

    ------------------------------------------------------------------------------
    -- Channels and data types
    ------------------------------------------------------------------------------
    Pid = {0..MAX_PROC-1}
    Fd = {0..MAX_FD-1}

    datatype Syscall = Fopen | Fclose | Fread | Fwrite | Chdir
    datatype Calltype = Call | CallArg.Fd | Ret | RetVal.Fd
    datatype Intcall = FopenIO | FcloseIO | GetBlock | PutBlock |
                       GetInode | PutInode | AllocInode | FreeInode |
                       AllocBlock | FreeBlock

    channel syscall : Pid.Syscall.Calltype
    channel intcall : Pid.Intcall.Calltype

    fsio_events =
        {| intcall.pid.FopenIO.Call, intcall.pid.FopenIO.RetVal,
           intcall.pid.FcloseIO.CallArg, intcall.pid.FcloseIO.Ret
```

```
     | pid <- Pid |}
-- The events for a given set of process IDs
syscall_events(pids) =
    {| syscall.pid.Fopen.Call, syscall.pid.Fopen.RetVal,
       syscall.pid.Fclose.CallArg, syscall.pid.Fclose.Ret,
       syscall.pid.Chdir.Call, syscall.pid.Chdir.Ret,
       syscall.pid.Fread.CallArg, syscall.pid.Fread.Ret,
       syscall.pid.Fwrite.CallArg, syscall.pid.Fwrite.Ret
     | pid <- pids |}

channel lookup_recurse
channel devio_recurse

bcache_events = {| intcall.pid.call | pid <- Pid, call <- {GetBlock, PutBlock} |}
icache_events = {| intcall.pid.call | pid <- Pid, call <- {GetInode, PutInode} |}
alloc_events = {| intcall.pid.call | pid <- Pid, call <- {AllocBlock, FreeBlock,
                                                          AllocInode, FreeInode} |}
inode_alloc_events = union(alloc_events, icache_events)
recurse_events = {| lookup_recurse, devio_recurse |}

-------------------------------------------------------------------------------
-- Block cache
-------------------------------------------------------------------------------
channel devio : Pid.Calltype
BlockCache = let
    devio_events = {| devio |}
    Device = devio?pid.Call -> devio.pid.Ret -> Device
    Cache = intcall?pid.GetBlock.Call -> devio.pid.Call -> devio.pid.Ret ->
            intcall.pid.GetBlock.Ret -> Cache
         [] intcall?pid.PutBlock.Call -> devio.pid.Call -> devio.pid.Ret ->
            intcall.pid.PutBlock.Ret -> Cache
    within (Cache [| devio_events |] Device) \ devio_events

InodeCache =
    -- Fetch an inode
    intcall?pid.GetInode.Call -> intcall.pid.GetBlock.Call ->
    intcall.pid.GetBlock.Ret -> intcall.pid.PutBlock.Call ->
    intcall.pid.PutBlock.Ret -> intcall.pid.GetInode.Ret -> InodeCache
    -- Release an inode
 [] intcall?pid.PutInode.Call -> intcall.pid.GetBlock.Call ->
    intcall.pid.GetBlock.Ret -> intcall.pid.PutBlock.Call ->
    intcall.pid.PutBlock.Ret -> intcall.pid.PutInode.Ret -> InodeCache

AllocServer = let
    Ready = intcall?pid.AllocBlock.Call -> DoBlockIO(pid, AllocBlock)
         [] intcall?pid.FreeBlock.Call -> DoBlockIO(pid, FreeBlock)
         [] intcall?pid.AllocInode.Call -> DoBlockIO(pid, AllocInode)
         [] intcall?pid.FreeInode.Call -> DoBlockIO(pid, FreeInode)
    DoBlockIO(pid, calltype) =
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        intcall.pid.calltype.Ret -> Ready
    within Ready

InodeAlloc = InodeCache ||| AllocServer

-------------------------------------------------------------------------------
-- Filesystem specification
-------------------------------------------------------------------------------
FSServer = let
```

```
    Ready(open) =
        -- open a file
        card(open) < MAX_FD & syscall?pid.Fopen.Call -> Lookup(open, pid, Opening)
        -- close an open file
     [] syscall?pid.Fclose!CallArg?fd:open -> Closing(open, pid, fd)
        -- change working directory
     [] syscall?pid.Chdir.Call -> Lookup(open, pid, ChangingDir)

    Lookup(open, pid, P) =
        -- Path is found in this directory
        intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get dirp
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret -> -- lookup path
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret -> P(open, pid)
        -- Recurse to another directory level
    |~| intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get dirp
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret -> -- lookup path
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- put dirp
        lookup_recurse -> Lookup(open, pid, P)

    Opening(open, pid) =
        -- opening an existing file
        intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get file inode
        intcall.pid.FopenIO.Call -> intcall.pid.FopenIO.RetVal?fd ->
        syscall.pid.Fopen.RetVal.fd -> Ready(union(open, {fd}))
        -- create a new file
    |~| intcall.pid.AllocInode.Call -> intcall.pid.AllocInode.Ret -> -- alloc inode
        intcall.pid.GetInode.Call -> intcall.pid.GetInode.Ret -> -- get file inode
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret -> -- add to directory
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- put dirp
        intcall.pid.FopenIO.Call -> intcall.pid.FopenIO.RetVal?fd ->
        syscall.pid.Fopen.RetVal.fd -> Ready(union(open, {fd}))

    Closing(open, pid, fd) =
        -- close an open file still in hierarchy
        intcall.pid.FcloseIO.CallArg.fd -> intcall.pid.FcloseIO.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- return inode
        syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))
        -- close and delete a file
    |~| intcall.pid.FcloseIO.CallArg.fd -> intcall.pid.FcloseIO.Ret ->
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret ->   -- return inode
        intcall.pid.FreeInode.Call -> intcall.pid.FreeInode.Ret -> -- free inode
        syscall.pid.Fclose.Ret -> Ready(diff(open, {fd}))

    ChangingDir(open, pid) =
        intcall.pid.PutInode.Call -> intcall.pid.PutInode.Ret -> -- return curdir
        syscall.pid.Chdir.Ret -> Ready(open)

    within Ready({})

-------------------------------------------------------------------------------
-- I/O server specification
-------------------------------------------------------------------------------

IOServer = let
    Ready(open) =
        -- handle an open file request
        intcall?pid.FopenIO.Call ->
        intcall.pid!FopenIO.RetVal?fd:diff(Fd, open) -> Ready(union(open, {fd}))
```

```
        -- close an open file
     [] intcall?pid.FcloseIO!CallArg?fd:open -> intcall.pid.FcloseIO.Ret ->
        Ready(diff(open, {fd}))
        -- read from an open file
     [] syscall?pid.Fread!CallArg?fd:open -> MultiBlockRead(open, pid)
        -- write to an open file
     [] syscall?pid.Fwrite!CallArg?fd:open -> MultiBlockWrite(open, pid)

    MultiBlockRead(open, pid) =
        -- more blocks to read
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockRead(open, pid)
        -- done reading
    |~| intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        syscall.pid.Fread.Ret -> Ready(open)

    MultiBlockWrite(open, pid) =
        -- update an existing block
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockWrite(open, pid)
        -- allocate and write to a new block
    |~| intcall.pid.AllocBlock.Call -> intcall.pid.AllocBlock.Ret ->
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockWrite(open, pid)
        -- block is truncate, free
    |~| intcall.pid.FreeBlock.Call -> intcall.pid.FreeBlock.Ret ->
        intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        devio_recurse -> MultiBlockWrite(open, pid)
        -- write to a block and finish
    |~| intcall.pid.GetBlock.Call -> intcall.pid.GetBlock.Ret ->
        intcall.pid.PutBlock.Call -> intcall.pid.PutBlock.Ret ->
        syscall.pid.Fwrite.Ret -> Ready(open)
within Ready({})

-- The two main servers synchronize only on the messages between them
Servers = FSServer [| fsio_events |] IOServer

--------------------------------------------------------------------------------
-- File descriptor regulator, ensuring proper behaviour
--------------------------------------------------------------------------------
FDRegulator(pid, open_fds) =
    card(open_fds) < MAX_PROC_FILES &
    syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd ->
    FDRegulator(pid, union(open_fds, {fd}))
 [] syscall.pid.Fclose.CallArg?fd:open_fds -> syscall.pid.Fclose.Ret ->
    FDRegulator(pid, diff(open_fds, {fd}))
 [] syscall.pid.Fread.CallArg?fd:open_fds -> syscall.pid.Fread.Ret ->
    FDRegulator(pid, open_fds)
 [] syscall.pid.Fwrite.CallArg?fd:open_fds -> syscall.pid.Fwrite.Ret ->
    FDRegulator(pid, open_fds)
 [] syscall.pid.Chdir.Call -> syscall.pid.Chdir.Ret ->
    FDRegulator(pid, open_fds)

FDRegulators = (|| pid:Pid @ [syscall_events({pid})] FDRegulator(pid, {}))
```

```
-------------------------------------------------------------------------------
-- File and IO servers make calls to the inode cache and allocation table
-------------------------------------------------------------------------------
-- The servers with file descriptor restrictions
ServersWithRules = Servers [| syscall_events(Pid) |] FDRegulators
-- File system accessing inode and allocation servers
AllServers= (ServersWithRules [| inode_alloc_events |] InodeAlloc)
Filesystem = AllServers [| bcache_events |] BlockCache

-------------------------------------------------------------------------------
-- A process that opens two files and copies between them
-------------------------------------------------------------------------------
OpenCopyClose(pid) = let
    Open =
        syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd1 ->
        syscall.pid.Fopen.Call -> syscall.pid.Fopen.RetVal?fd2 -> Copy(fd1, fd2)
    Copy(fd1, fd2) =
        syscall.pid.Fread.CallArg.fd1 -> syscall.pid.Fread.Ret ->
        syscall.pid.Fwrite.CallArg.fd2 -> syscall.pid.Fwrite.Ret -> Close(fd1, fd2)
    Close(fd1, fd2) =
        (syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret ->
        syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret -> Open)
     [] (syscall.pid.Fclose.CallArg.fd2 -> syscall.pid.Fclose.Ret ->
        syscall.pid.Fclose.CallArg.fd1 -> syscall.pid.Fclose.Ret -> Open)
within Open

-------------------------------------------------------------------------------
-- Use non-determinism to allow any file to be opened
-------------------------------------------------------------------------------
OpenAny(pid) = syscall.pid.Fopen.RetVal$fd1 -> OpenAny(pid)

-------------------------------------------------------------------------------
-- Regulator that forces serialisation of system calls
-------------------------------------------------------------------------------
SeqLimit = syscall?pid?call?calltype -> syscall.pid!call?rettype -> SeqLimit

-------------------------------------------------------------------------------
-- Regulator that models separate FS and IO servers
-------------------------------------------------------------------------------
channel fsiocall
ConcLimit = let
    FS = syscall?pid.Chdir.Call -> syscall.pid.Chdir.Ret -> FS
      [] syscall?pid.Fopen.Call ->
         fsiocall ->
         syscall.pid.Fopen.RetVal?fd -> FS
      [] syscall?pid.Fclose!CallArg?fd ->
         fsiocall ->
         syscall.pid.Fclose.Ret -> FS

    IO = syscall?pid.Fread!CallArg?fd -> syscall.pid.Fread.Ret -> IO
      [] syscall?pid.Fwrite!CallArg?fd -> syscall.pid.Fwrite.Ret -> IO
      [] fsiocall -> IO

within (FS [| {| fsiocall |} |] IO) \ {| fsiocall |}

-------------------------------------------------------------------------------
-- Specification that is aware of lookup/devio recursion
-------------------------------------------------------------------------------
FSCanRecurse = let
    Ready = syscall.0.Fopen.Call -> OpenLookup
```

```
            [] syscall.0.Fclose.CallArg?fd -> syscall.0.Fclose.Ret -> Ready
            [] syscall.0.Chdir.Call -> ChdirLookup
            [] syscall.0.Fread.CallArg?fd -> DevIO(Fread)
            [] syscall.0.Fwrite.CallArg?fd -> DevIO(Fwrite)
        OpenLookup = syscall.0.Fopen.RetVal?fd -> Ready
                |~| lookup_recurse -> OpenLookup
        ChdirLookup = syscall.0.Chdir.Ret -> Ready
                |~| lookup_recurse -> ChdirLookup
        DevIO(call) = syscall.0.call.Ret -> Ready
                |~| devio_recurse -> DevIO(call)
    within Ready

    -- Actions that aren't taken directly by the specification of process 0
    OtherProcs = CHAOS(Union({
        recurse_events,
        syscall_events(diff(Pid, {0}))
    }))

    CopySpec = OpenCopyClose(0) [| {| syscall.0.Fopen.RetVal |} |] OpenAny(0)

    Spec = (CopySpec [| {| syscall |} |] FSCanRecurse)
    SpecWithAny = Spec ||| OtherProcs

    SeqSpec = SpecWithAny [| {| syscall |} |] SeqLimit
    SeqFilesystem = Filesystem [| {| syscall |} |] SeqLimit

    ConcSpec = SpecWithAny [| {| syscall |} |] ConcLimit
    ConcFilesystem = Filesystem [| {| syscall |} |] ConcLimit

    spec_hidden = Union({
        {| intcall |}
    })
endmodule
```

B.3.1 Assertions: Sanity check

```
include "04-internal.csp"

instance FS = InternalServer(4, 2, 2)

assert FS::SeqLimit :[deadlock free [FD]]
assert FS::ConcLimit :[deadlock free [FD]]
assert FS::Filesystem :[deadlock free [FD]]
assert FS::ConcFilesystem :[deadlock free [FD]]
assert FS::Spec :[deadlock free [FD]]
```

B.3.2 Assertions: Normal

```
include "04-internal.csp"

instance FS = InternalServer(4, 2, 2)
CopySpec = FS::CopySpec
OtherProcs = FS::OtherProcs
syscall = FS::syscall
SeqLimit = FS::SeqLimit
Filesystem = FS::Filesystem
OpenCopyClose = FS::OpenCopyClose
```

```
--SeqSpec = (CopySpec ||| OtherProcs) [| {| syscall |} |] SeqLimit
--SeqFilesystem = Filesystem [| {| syscall |} |] SeqLimit

intcall = FS::intcall
spec_hidden = {| intcall |}

SeqSpec = FS::SeqSpec
SeqFilesystem = FS::SeqFilesystem
syscall_0_events = {| syscall.0 |}

assert SeqSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0)) \ spec_hidden

ConcSpec = FS::ConcSpec
ConcFilesystem = FS::ConcFilesystem

assert ConcSpec [FD= (ConcFilesystem [| syscall_0_events |] OpenCopyClose(0)) \ spec_hidden
```

B.3.3 Assertions: Not enough file descriptors

```
include "04-internal.csp"

instance FS = InternalServer(3, 2, 2)
CopySpec = FS::CopySpec
OtherProcs = FS::OtherProcs
syscall = FS::syscall
SeqLimit = FS::SeqLimit
Filesystem = FS::Filesystem
OpenCopyClose = FS::OpenCopyClose

--SeqSpec = (CopySpec ||| OtherProcs) [| {| syscall |} |] SeqLimit
--SeqFilesystem = Filesystem [| {| syscall |} |] SeqLimit

spec_hidden = FS::spec_hidden

SeqSpec = FS::SeqSpec
SeqFilesystem = FS::SeqFilesystem
syscall_0_events = {| syscall.0 |}

assert not SeqSpec [FD= (SeqFilesystem [| syscall_0_events |] OpenCopyClose(0)) \ spec_hidden

ConcSpec = FS::ConcSpec
ConcFilesystem = FS::ConcFilesystem

assert not ConcSpec [FD= (ConcFilesystem [| syscall_0_events |] OpenCopyClose(0)) \ spec_hidden
```

References

[1] Docker. http://docker.io.

[2] Groupcache. https://github.com/golang/groupcache.

[3] Gstreamer: Open Source Multimedia Framework. http://gstreamer.freedesktop.org/.

[4] Node.js: Event-Driven Network Application Framework. http://nodejs.org/.

[5] Prophetic Petroglyphs. http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html.

[6] Vitess: Scaling MySQL Databases for the Web. http://code.google.com/p/vitess/.

[7] World occam Transputer User Group (WoTUG). http://wotug.org/about.shtml.

[8] Lighttpd Web Server. http://www.lighttpd.net/, August 2010.

[9] Yaws - Yet Another Webserver. http://yaws.hyber.org/, September 2010.

[10] Go Authors. FAQ - The Go Programming Language.

[11] Go Authors. Go 1.3 Release Notes.

[12] Go Authors. The Go Programming Language Specification.

[13] H C Baker Jr and C Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12(8):55–59, 1977.

[14] F R M Barnes and P H Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk

Muller, editors, *Communicating Process Architectures 2001*, number 59 in Concurrent Systems Engineering Series, pages 243–264. IOS Press, Amsterdam, The Netherlands, September 2001.

[15] F R M Barnes and P H Welch. occam-pi: blending the best of CSP and the pi-calculus, 2006.

[16] Fred Barnes. occwserv: An occam Web-Server. In J F Broenink and G H Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 251–268, Amsterdam, The Netherlands, September 2003. IOS Press.

[17] Frederick R M Barnes and Carl G Ritson. Checking Process-oriented Operating System Behaviour Using CSP and Refinement. *SIGOPS Oper. Syst. Rev.*, 43(4):45–49, January 2010.

[18] T Berners-Lee, R Fielding, and H Frystyk. RFC1945: Hypertext Transfer Protocol–HTTP/1.0. *RFC Editor United States*, 1996.

[19] Rick D Beton. libcsp - a Building mechanism for CSP Communication and Synchronisation in Multithreaded C Programs. In Peter H Welch and Andrè W P Bakkers, editors, *Communicating Process Architectures 2000*, pages 239–250, September 2000.

[20] John Markus Bjørndalen, Brian Vinter, and Otto J Anshus. PyCSP - Communicating Sequential Processes for Python. In Alistair A McEwan, Steve Schneider, Wilson Ifill, and Peter H Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, July 2007.

[21] Eric Bonnici and Peter H Welch. Mobile Processes, Mobile Channels and Dynamic Systems. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 182–196. IEEE Press, 2009.

[22] N C C Brown and P H Welch. An introduction to the Kent C++ CSP Library. *Communicating Process Architectures*, 61:139–156, 2003.

[23] Neil C C Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H Welch, S Stepney, F A C Polack, Frederick R M Barnes, Alistair A McEwan, G S Stiles, Jan F Broenink, and Adam T Sampson, editors, *Communicating Process Architectures 2008*, pages 67–83, September 2008.

[24] Neil C C Brown and Peter H Welch. An Introduction to the Kent C++CSP Library. In Jan F Broenink and Gerald H Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, September 2003.

[25] Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.

[26] Brad L Cox. The Object Oriented Pre-compiler: Programming Smalltalk 80 Methods in C Language. *SIGPLAN Not.*, 18(1):15–22, January 1983.

[27] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 186–189, New York, NY, USA, 2002. ACM.

[28] Ole-Johan Dahl. *SIMULA 67 Common Base Language, (Norwegian Computing Center. Publication)*. 1968.

[29] C David and Peter H Welch. The Kent retargetable occam compiler. In *Parallel Processing Developments: WoTUG-19: Proceedings of the 19th World Occam and Transputer User Group Technical Meeting, 31st March-3rd April 1996, Nottingham, UK*, volume 47, page 143. IOS Press, 1996.

[30] E De Vries, R Plasmeijer, and D Abrahamson. Uniqueness typing simplified. *Implementation and Application of Functional Languages*, pages 201–218, 2008.

[31] Rust Project Developers. Rust language. `http://www.rust-lang.org/`, 2012.

[32] Andrea Fazzi. GoSpeccy - An evolving ZX Spectrum 48k Emulator.

[33] D P Friedman and D S Wise. *The Impact of Applicative Programming on Multiprocessing*, pages 263–272. Indiana University, Computer Science Department, 1976.

[34] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[35] OPEN GROUP and Others. The Single UNIX Specification, Version 2. *System Interface & Headers (XSH)*, (5), 1997.

[36] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009.

[37] C A R Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21:666–677, 1978.

[38] C A R Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[39] M A Jackson. *Principles of program design*, volume 123790506. Academic Press, Inc., 1975.

[40] Christian L Jacobsen, Frederick R M Barnes, and Brian Vinter. RMoX: A raw-metal occam Experiment. In Jan F Broenink and Gerald H Hilderink, editors, *Communicating Process Architectures 2003*, pages 269–288, September 2003.

[41] H Jifeng, I Page, and J Bowen. Towards a provably correct hardware implementation of Occam. *Correct Hardware Design and Verification Methods*, pages 214–225, 1993.

[42] Maxwell Krohn, Eddie Kohler, and M Frans Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 7:1—-7:14, Berkeley, CA, USA, 2007. USENIX Association.

[43] Hugh C Lauer and Roger M Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

[44] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 189–199, New York, NY, USA, 2007. ACM.

[45] J Lions. A Commentary on the Sixth Edition UNIX Operating System. *Department of Computer Science, The University of New South Wales*, 1977.

[46] Jeremy M R Martin and Peter H Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4):215–232, 1997.

[47] M. Douglas McIlroy. Coroutines. *Bell Labs Technical Report*, 1968.

[48] R Milner. *Communicating and mobile systems: the π-calculus*. Cambridge Univ Pr, 1999.

[49] Chris Nevison. Teaching Distributed and Parallel Computing with Java and CSP. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0:484, 2001.

[50] University of Oxford. CSP-M Manual. `http://www.cs.ox.ac.uk/projects/fdr/manual/cspm.html`.

[51] J Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*. Citeseer, 1996.

[52] I Page and W Luk. Compiling Occam into field-programmable gate arrays. *FPGAs*, pages 271–284, 1991.

[53] R M A Peel and B M Cook. Occam on Field Programmable Gate Arrays-Fast Prototyping of Parallel Embedded Systems. In *the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTAâĂŹ2000)*, pages 2523–2529, 2000.

[54] Rob Pike. Lexical Scanning in Go, 2011.

[55] F A C Polack, P S Andrews, and A T Sampson. The engineering of concurrent simulations of complex systems. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 217–224. IEEE, 2009.

[56] POSIX.1-2008. The Open Group Base Specifications. Also published as IEEE Std 1003.1-2008, July 2008.

[57] Ian C Pyle. *ADA Programming Language*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[58] Carl G Ritson and Peter H Welch. A Process-Oriented Architecture for Complex System Modelling. *Concurrency and Computation: Practice and Experience*, 22:182–196, 2010.

[59] D Robinson and K Coar. The common gateway interface (CGI) version 1.1. Technical report, RFC 3875, October 2004.

[60] A W Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[61] A T Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, PhD thesis, University of Kent, 2010.

[62] Adam T Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.

[63] Adam T Sampson, John Markus Bjrndalen, and Paul S Andrews. Birds on the Wall: Distributing a Process-Oriented Simulation. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 182–196. IEEE Press, 2009.

[64] SGS-Thomson Microelectronics Limited. occam 2.1 Reference Manual. 1995.

[65] Bernhard HC Sputh and Alastair R Allen. JCSP-Poison: Safe termination of CSP process networks. In Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood, editors, *Communicating Process Architectures 2005: WoTUG-28: Proceedings of the 28th WoTUG Technical Meeting, 18-21 September 2005, Technische Universiteit Eindhoven, The Netherlands*, volume 63, page 71. Ios PressInc, 2005.

[66] Bernard Sufrin. Communicating Scala Objects. In Peter H Welch, S Stepney, F A C Polack, Frederick R M Barnes, Alistair A McEwan, G S Stiles, Jan F Broenink, and Adam T Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, September 2008.

[67] Alexandre Boulgakov A W Roscoe Thomas Gibson-Robinson Philip Armstrong. *Failures Divergences Refinement (FDR) Version 3*, 2013.

[68] R Von Behren, J Condit, and E Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, page 4. USENIX Association, 2003.

[69] Rob Von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio. In *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, volume 37, page 268, New York, New York, USA, October 2003. ACM Press.

[70] P Wadler. Linear types can change the world. In M Broy and C B Jones, editors, *Programming concepts and methods: proceedings of the IFIP Working*

Group 2.2/2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990, pages 561–581. North-Holland, 1990.

[71] P H Welch. Process Oriented Design for Java–Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. Citeseer.

[72] P H Welch. Graceful Termination–Graceful Resetting. *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, 1989.

[73] P H Welch and F R M Barnes. Communicating Mobile Processes: introducing occam-pi. In A E Abdallah, C B Jones, and J W Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[74] P H Welch, N C C Brown, J Moores, K Chalmers, and B Sputh. Integrating and Extending JCSP. In Steve Schneider, Alistair A McEwan, Wilson Ifill, and Peter H Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 182–196, Amsterdam, The Netherlands, 2007. WoTUG, IOS.

[75] Peter H Welch. Communicating Sequential Processes for Java (JCSP). `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[76] Peter H Welch, Kurt Wallnau, Adam T Sampson, and Mark Klein. To Boldly Go: an occam-π mission to engineer emergence. *Natural Computing*, pages 1–27, April 2012.

[77] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[78] James Whitehead II. Webpipes: A Compositional Web Sever Toolkit. `http://github.com/jnwhiteh/webpipes`.