NTNU – Trondheim
Norwegian University of
Science and Technology

ProXC - A CSP-Inspired Concurrency Library for the C Programming Language

Edvard Severin Pettersen

Trondheim December 2016

PROJECT THESIS TTK4550 (15 SP)

Department of Engineering Cybernetics

Norwegian University of Science and Technology

Supervisor: Professor Sverre Hendseth

Co-supervisor: Øyvind Teig

Project Description

Create an abstraction of Communicating Sequential Processes (CSP) for C programs. This abstraction should be influenced by the CSP-based languages occam, XC and Go. Discuss limitations and uses.

The following features and constructs should be available in this abstraction:

- Lightweight processes
- Composite processes, including sequential and parallel process ordering
- Channels
- Alternation structure with conditional guards
- Alternation alternatives, including channel reads

A list of optional features if time and possibility allows it:

- Multi-core support
- Timers
- Alternation alternatives, including timeouts and skip

Assignment given: 26. August, 2016

Supervisor: Sverre Hendseth, ITK

Preface

This project thesis presents the results of the project work, affiliated with the course TTK4550 from Department of Engineering Cybertnertics (ITK) at the Norwegian University of Science and Technology (NTNU). The project work was carried out during the autumn semester in 2016, starting in August and ending in December.

The majority of the work has gone in the implementation of the library, especially how the coroutine context switching was implemented. I hope this thesis reflects this.

The reader is expected to have basic knowledge within programming and concurrency. Having experience with programming in C will help as well.

I'd like to thank both of my supervisors, Sverre Hendseth and Øyvind Teig, for great feedback and knowledge regarding concurrent system design and paradigms, and for enjoyable and interesting academic evenings at Solsiden. Also, extra thanks to Sverre for being patient and helping me with questions and advice regarding the writing process of this thesis.

Edvard Severin Pettersen

Trondheim, 2016-12-16

Abstract

This paper describes the design and implementation of ProXC, a *Communicating Sequential Processes* (CSP) inspired concurrency library for the C programming language. ProXC aims to enable C programs to use CSP abstractions, allowing a more expressive and safer design of concurrent systems. The choice of implementing a library was the result of an evaluation, assessing which approach to create the framework.

The CSP abstractions in ProXC are a collection of lightweight processes, which communicates via channels. Composite processes can be defined, where the execution ordering and synchronous and asynchronous execution are supported. Alternation construct is also possible, allowing a process to wait on multiple channel reads, with or without a timeout or a skip-able alternative. Process suspension is also possible.

ProXC is a single-core implementation, which emulates concurrency by implementing stackful coroutines. Each process is a coroutine, with an underlying run-time system taking care of scheduling, synchronization and resource handling between processes.

Limitations of ProXC is also discussed, explaining what ProXC can and can not do, and what in hindsight could have been done better during development of ProXC.

The abstractions ProXC provides is argued to be correct and expressive. By highlighting the library features and how these compare to the abstractions in the formal language CSP, ProXC is concluded to be a feature rich CSP abstraction framework for C programs. Furthermore, some simple benchmark comparisons between ProXC and other existing CSP based languages are conducted, giving ProXC promising results for concurrent throughput.

The source code for ProXC is publicly available at GitHub, released with the open-source MIT license.

Contents

Chapter 1

Introduction

The notion of concurrency in programming has existed since the 60's. Concurrency regarding computing means multiple computations that are executed in an overlapping fashion, giving the impression that multiple computations are advancing at the same time. This allows programmers to intuitively implement systems which express parallelism natively. Useful and powerful concepts such as semaphores, monitors and threads forms its basis on concurrency. However, new potential challenges such as deadlocks, livelocks and race conditions also comes with concurrency.

For the programmer, the use of concurrency in programs is an added mental overhead. This is mostly as a result of the error prone nature of concurrent programming, and how a small and trivial bug in a program can almost be impossible to find. As a result, many programmers choose other simpler paradigms to avoid this added overhead.

The introduction of *Communicating Sequential Processes* (CSP) by Tony Hoare [1] formed a whole new basis within concurrent computing. CSP is on its own a formal language, which describes concurrent systems by defining independent processes, communicating only through message-passing. Not only is CSP a powerful tool to describe concurrent systems, but is also a tool to analyze and check the correctness of concurrent systems.

For concurrent programming, CSP has proven to be a powerful tool. Programs using CSP abstractions follows a more stricter ruleset, creating more correct concurrent systems. This follows from CSP inherently being race condition free. CSP also allows concurrent system to be analyzed, such as proving specification refinement, as well as absence of deadlocks and live-

locks.

CSP has influenced the design of multiple programming languages, such as occam, XC and Go. Many of these programming languages have sadly never received the mainstream popularity it deserved, mostly from being proprietary or the lack of use of CSP in the industry. The exception here is Go, which has risen in popularity the last few years. This lack of mainstream CSP programming languages, both in use and availability, has resulted in the development of multiple libraries for more popular languages and compilers of existing CSP languages. This includes JCSP [2], C++CSP2 [3], SPoC [4], KRoC [5], and CCSP [6]. But as this list shows, there does not exist a fully fledged framework enabling CSP abstractions in C programs, that is both feature rich and mature for modern use.

This report introduces and describes the concurrency library ProXC, which enables the use of CSP abstraction in C programs. Chapter 1 briefly explains the motivation for using CSP abstractions, and the lack of frameworks for enabling CSP abstractions in C programs. Chapter 2 goes more in depth in related concepts and existing work in concurrency programming and the CSP paradigm. Chapter 3 assesses the different approaches for creating a CSP abstraction, and concludes with creating a library. Chapter 4 introduces ProXC the library, including the complete set of features, target platform and influences. Chapter 5 details the design and implementation of the library. Chapter 6 presents examples of code, showcasing the different library features. Chapter 7 briefly compares benchmarks of the library compared to other existing CSP languages. Chapter 8 discusses how the library compares as a CSP abstraction, limitations, and what should have been done differently during development. Chapter 9 discusses possible future work for this project. Chapter 10 concludes the report.

Chapter 2

Background

In this chapter a more in-depth explanation of the different topics introduced in the introduction chapter is presented. These topics cover the required background knowledge for the project, which plays an important role in determining on what kind of development tool is to be implemented, how it is to be implemented, as well as showing what already exists of current solutions and tools.

Each topic is presented on its own, and how it relates to the paper.

2.1 Concurrency vs. Parallelism

The notion of parallelism and concurrency in programs spawns from the limitations of sequential programs. All programming languages have different ways to express the structure and control flow of a program, but in the end all programs are transformed to machine code which the processor executes *sequentially*. That means for a single processor, only one instruction is executed at a time[1]. Concurrency however defines a program into multiple independent sequential programs, which in turn runs each sequential program in interleaving time periods. Even though only one instruction is executed at a time, this gives the impression that multiple sequential programs are advancing at the same time.

Parallelism in this sense refers to multiple programs running on multiple processors in parallel. This description might look familiar to how concurrency is described, but it is important

[1] This implies the processor is single-core

not to confuse these two terms together. Concurrency only serves as an abstraction for parallelism in a program, allowing the programmer to pretend multiple sequential programs are executed in parallel. This abstraction would be valid for both single and multi-core, while parallelism describes a condition which only happens on multi-core architectures. A more thorough and complete definition of concurrency and parallelism is described in Ben-Ari [7].

Concurrency and parallelism forms the very core of this project. Section 2.5 explains what CSP is, which forms its basis on concurrency. Understanding the underlying concepts of concurrency is important to understand what CSP entails.

2.2 Threading Models

As concurrency is a superb tool for programmers to abstract parallelism in their program, the implementation details of this is equally as important for this project. There are different ways to implement this, but almost all concurrency models implement some threading mechanism. When talking about threading mechanisms on *Operating Systems* (OS), one usually talks about two kinds of threads: user- and kernel-threads. As the names may imply, these threads operate in either user- or kernel-space. Three main models explain the different combinations of the two threads: user-, kernel-, and hybrid-threading models. Brown [3] goes further into details on these models, and a short summary is presented below from said article.

In short, the user-threading model implements a cooperative scheduled threading in user-space, and is called a `M:1` threading model. This model is running `M` user-threads on a single kernel-thread, where everything from scheduling and context switching is happening unbeknownst to the kernel-thread. This threading mechanism is not visible to the OS, and avoids the overhead related to context switches in kernel-threads. However, blocking calls are a challenge, as a single blocking call will block all user-threads.

Kernel-threading model is often directly supported in OS kernels, and is called a `1:1` threading model. Each kernel-thread is scheduled by the OS onto the the system's processors, which is often implemented as preemptive scheduling. Due to kernel-threads residing in kernel-space, context switches has a much larger overhead than user-threading. Blocking calls are however not a problem.

The Hybrid-threading model combines the two models, and is called a `M:N` threading model. This model is running multiple kernel-threads, with each kernel-thread running multiple user-threads. Blocking calls is still a problem, but can be mitigated through dispatching a single kernel-thread for the block user-thread, and continue the rest of the scheduling on a new kernel-thread.

2.3 Memory Layout of C Programs

The C programming language defines a memory layout of its programs. A total of 5 sections defines the memory layout.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap

The text segment contains the executable code of instructions. The initialized data segment contains global and static variables that are initialized by the programmer. The uninitialized data segment contains all zero initialized or non-initialized variables and data. The stack contains the program stack and stack frames, while the heap contains dynamic memory. The stack grows downwards, from higher to lower address, opposed to the heap which grows upwards, from lower to higher address.

The unmapped segment is a precursor to the stack, containing command-line arguments and environment variables sent as program arguments.

See Figure 2.1 for an outline of the memory layout.

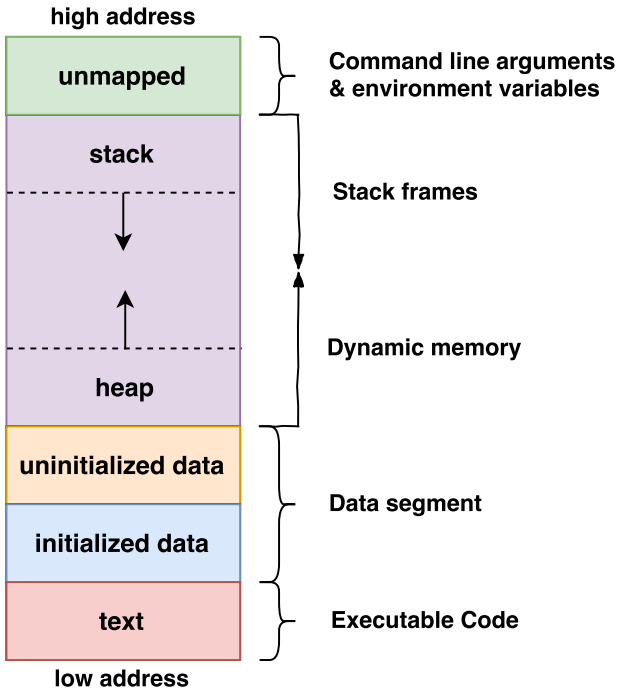

Figure 2.1: Memory layout of a C program

2.4 Stackful Coroutines

Coroutines are a generalized subroutines within a program, used for non-preemptive multitasking. This allows a single-threaded program to suspend and resume multiple executions, through predetermined scheduling points. There exists two types of coroutine implementations: stackful and stackless. As the name implies, stackful coroutines each has its own stack. While for stackless coroutines, the main stack frame is used by all coroutines.

Resuming and suspending stackful coroutines involves storing and swapping contexts of coroutines. Each coroutine has a context which represents the execution state of a coroutine at a given time. This execution state can be seen as a snapshot of the processor registers at a given place in the code execution. Suspending a coroutine is a matter of storing the current execution state in memory, and replace the execution state with another coroutine. Lastly, the program counter is replaced with the resuming coroutine. This procedure effectively suspends the initial coroutine, and resumes the other.

Knowing what to store in execution states requires knowledge of the processor state, which is architecture dependent. This is why coroutine implementations often cause portability issues.

The solution to this is looking at the calling convention for System V *Application Binary Interface* (ABI) [8], which is the convention almost all UNIX-like architectures follow.

Calling conventions describes, on a low-level, how subroutines receive parameters from their caller and how they return their result. This usually describes on register level which registers must be preserved by the callee, and which must not. Knowing this, a context structure and a context switching procedure can be outlined.

The context structure should contain the preserved registers specified by the calling convention and the program counter for a given coroutine. The context switch procedure must be a function call. This means all volatile variables remaining in non-preserved registers will be pushed to the stack before calling the procedure. Then, in the procedure, all preserved registers are stored in the context structure, along with the program counter. When resuming, the preserved registers are loaded from the context structure and lastly sets its own program counter. This will cause a jump to the other coroutine context, which resumes as if the context switching procedure call returned.

2.5 Communicating Sequential Processes

First introduced by Tony Hoare in 1978 [1], Communicating Sequential Programming (CSP) aimed to make concurrency and parallelism a much more convenient tool for programmers. Even though CSP in itself is a mathematical formal language, the concepts and structuring it introduces is very applicable as a programming model. The core concept of CSP is a composition of concurrent sequential processes, which strictly communicate by message passing via channels.

Despite CSP being an excellent model for describing concurrent systems, it has not gained much mainstream traction within programming communities and industries. As detailed in Ben-Ari [7], concurrent programming introduces new challenges such as race conditions, deadlocks and fairness. These challenges can for many programmers be a hurdle to create correct concurrent programs, shown in Ousterhout [9]. As a result of this they might end up opting in for other simpler programming models, and in turn miss out of CSP.

The lack of mainstream popularity does not mean however that there does not exist CSP im-

plementations. These implementations range from programming languages which implements the CSP model (Section 2.5.1), CSP libraries for more mainstream non-CSP programming languages (Section 2.5.2), and CSP language compilers (Section 2.5.3).

The CSP implementations range in varying popularity and use, and below is a summary of the most notable entries which relates to this project presented.

2.5.1 Programming Languages

Multiple programming languages implementing the CSP models in varying degree exists. Probably the most notable and truest implementation of the CSP model is occam. Other languages such as XC and Go is heavily influenced by the CSP model and occam. There exists other CSP languages such as Limbo [10], Joyce [11], and SuperPascal [12], however occam, XC and Go seems to be the most influential and used languages to date[2].

occam

Occam is a concurrent programming language which builds upon the CSP model. It was created by INMOS [13] and first appeared in 1983. Occam was initially developed as the native programming language for the Transputer, their microprocessor architecture highly specialized for real-time parallel computing [14].

Occam was initially only developed for the Transputer, making it hardware locked and inaccessible for everyone not using the Transputer. However, occam proved to be quite expressive and useful for concurrent programming, which spurred multiple ports and compilers of occam to other open architectures. Sadly, occam never received the appropriate recognition it deserved, and instead got its place in a niche market of real-time parallel computing with its highly specialized microprocessor Transputer.

All occam programs consists of processes. These processes can either be expressed as a single statements, or as a collection of multiple statements. What is very different in occam compared to other languages, is the native support for parallelism. The `PAR` keyword specifies a parallel execution between one-or-more processes. There is an equivalent keyword for se-

[2] This is a highly subjective view, based on the popularity and relevance in the industry

quential execution, called SEQ. All statements in occam must be explicitly specified if they are executed in sequence or parallel.

```
PAR                                SEQ
  p()                                x := 10
  q()                                x := x + 1
  z()                                y := x * x
```

Processes can be defined with the PROC keyword. The process can have zero-or-more arguments.

```
PROC system()
  SEQ
    i := 1
    i := i + 1
:
```

Processes are not allowed to share memory. All communication between processes is done through channels. Channels in occam are unbuffered and synchronized, which means both a sender and a receiver must be present on a channel to complete the transfer of data. Channels are also typed. Occam also requires disjointness on channels, meaning only one unique process can be a sender and a receiver on a given channel. In occam, the operator "!" is used to send on a channel, and "?" to receive on a channel.

```
INT x:
CHAN OF INT c:
PAR
  c ? x
  c ! 1
```

Occam also supports alternation, using the keyword ALT. This allows a process to wait on multiple channel reads at the same time. These are called alternatives, which can be guarded on a conditional boolean. This enables the alternative only if the conditional is true. A SKIP alternative may also be included, which as always available.

```
ALT
  a = b -> signal ? type:
    process(type)
  data ? value:
    check(value)
  SKIP
```

XC

XC is a concurrent programming language, which builds upon the concurrency and parallelism introduced in occam, and the syntax is based on C with new extensions and some minor restrictions. XC is the native programming language on the XCore processor architecture, created by XMOS [15]. XC first appeared in 2005.

XCore is a multi-core processor for embedded system, aimed at utilizing concurrency and parallelism natively. The only major implementation of XC is for the XCore architecture, making XC a hardware locked language such as occam was initially. Since there are no other major implementations of XC for other than XCore, it has more or less gained a niche market in embedded real-time parallel computing.

XC also has native support for parallelism. With the keyword `par` and a corresponding scope block of `{}`, all statements within the scope block are executed in parallel. This is somewhat equivalent of the keyword `PAR` in occam. XC does not need a sequential keyword, as this is the inherent behaviour in the C language.

```
par { f(); g(); }
```

Threads in XC is not allowed to share memory. Just as in occam, all communication between threads must go through channels. Channels in XC behaves almost the same as in occam, mainly being typed and one-to-one. The operators `:>` and `<:` are used to receive and send values over a channel, respectively.

```
chan c;
int x;
par {
    c <: 42;
    c :> x;
}
```

There are three types of channels, namely interfaces, normal channels, and streaming channels. Interfaces specify an interface between a client and a master thread, where the client can instantiate transactions through the interface. Interfaces are unbuffered and synchronized. Normal channels are used as a primitive way for threads to communicate, allowing to send and receive data. They are also unbuffered and synchronized. Streaming channels are a permanent connection between two threads, which are a buffered and asynchronous transfer of data.

Alternation is also supported in XC, using the keyword `select` with a corresponding scope block `{}`. Just like the switch construct, the scope block contains cases which consist of channel reads or channel writes. Each case can also be guarded by a conditional.

```
select {
    case enable_left => left :> v:
        out <: v;
        break;
    case enable_right => right :> v:
        out <: v;
        break;
}
```

Go

Go is a concurrent programming language, created and developed by Google [16], and first appeared in 2009. Go does not build itself entirely on the CSP model, but implements many features and ideas from CSP. This includes channels, parallel composition of asynchronous processes, and event handling on multiple channels with `select` [17].

Go started as a desire of a modern concurrent language, which was both fast and scalable, and supported networking and multiprocessing. Go has since its initial launch in 2009 risen in

popularity. As Go was not designed for a specific architecture, compared to occam and XC, Go became quite popular in the networking industry. As of writing of this paper, Go ranks as the 13th most popular programming language on the TIOBE Index [18].

The native support for parallelism in Go is through goroutines, which is their equivalent of coroutines. Goroutines are light-weight processes, which are spawned with the keyword `go`. Goroutines are asynchronous by nature, meaning the process which spawns a goroutine does not wait for it to finish, compared to occam and XC which does wait. Goroutines must be functions, and not single statements.

```
func main() {
    go a()
    go b()
    runtime.Goexit()
}
```

Channels are also supported in Go, but are not mandatory to use. There exists both buffered and unbuffered channels, and are type safe. Channels are created with the built-in function `make`, which takes the data type and the capacity of the channel as arguments. If the capacity is omitted, the channel is unbuffered, else it is buffered.

```
syncCh := make(chan int) // unbuffered
asyncCh := make(chan float64, 3) // buffered
```

Alternation construct is also supported in Go, with the same approach as XC. The keyword `select` is used with a corresponding scope block `{}`. In the scope block, multiple cases with channel read or writes are specified. A default case can also be supplied, that is chosen if none of the other cases are available at initialization. Go does not however support guarded cases natively.

```
writeCh := make(chan bool)
readCh := make(chan string)
select {
case word := <-readCh: fmt.Println("Channel read")
case writeCh <- true:  fmt.Println("Channel write")
}
```

2.5.2 Programming Libraries

With the lack of concurrent programming languages, CSP libraries have been developed for existing sequential, imperative programming languages. This includes libraries such as JCSP [2], C++CSP2 [3], PyCSP [19], and CSP.NET [20], however only JCSP and C++CSP2 will be summarized below as of relevance to this project.

C++CSP2

C++CSP2 [3] is a concurrency library for C++. The original library C++CSP [21] provided the same features as C++CSP2, but used only user-threads for its concurrency. C++CSP2 aimed to redesign the implementation of the library into a many-to-many threading model, which achieves proper utilizing of multi-core processors.

Much of the implementation and design regarding the CSP features and constructs are detailed in Brown and Welch [21], while the details regarding the multi-core support of the concurrency model are in Brown [3].

C++CSP2 provide CSP features such as processes, buffered/unbuffered typed one-directional channels, channel Poison for easily shutting down concurrent programs, time functions, alternative construct for expressing choice, barrier synchronization, and smart pointer Mobile to prevent aliasing. These features and more are explained in more detail in the documentation [22].

Processes are defined as classes, which inherit the `CSPProcess` class. The class must also implement the `run`, method, which is the procedure body of the process.

```
class WaitProcess : public CSProcess {
private:
    Time t;
protected:
    void run() {
        SleepFor(t);
    }
public:
    WaitProcess(const Time& _t) : t(_t)
    {}
};
```

Processes can be run in parallel or sequential order, just as in occam and XC. With the library procedures `InParallel` and `InSequence`, these process classes can then be executed in parallel and sequence, respectively. Execution orderings are also nestable.

```
Run(InSequence
    ( new WaitProcess(Seconds(3)) )
    ( new WaitProcess(Seconds(3)) )
);
Run(InParallel
    ( new WaitProcess(Seconds(3)) )
    ( new WaitProcess(Seconds(3)) )
);
```

Channels are unbuffered, synchronous and type safe. Four types of disjointed channels are supported: any-to-any, any-to-one, one-to-any and one-to-one. C++CSP2 explicitly operates on writing and reading ends of a channel, which are handed out from a created channel.

```
One2OneChannel<int> c;
Run(InParallel
    ( new IncreasingNumbers( c.writer() ) )
    ( new NumberPrinter( c.reader() ) )
);
```

The writing operand is << and reading operand is >>.

```
Chanin<int> in;
Chanout<int> out;
int total = 0;
int n;
in >> n;
total += n;
out << total;
```

One added feature in C++CSP2 which is not commonly implemented among CSP frameworks is channel poison. Poison is used to terminate concurrent programs gracefully.

Alternation is also supported in C++CSP2. Just as occam, C++CSP2 only supports channel reads as alternatives. The alternatives can not be guarded by a conditional boolean. It supports two types of selection, one fair selection and one priority selection. One distinction from other CSP frameworks is that an alternative selection does not complete the channel operation. This has to be done manually in the branched code.

```
AltChanin<Data> dataIn;
AltChanin<QuitSignal> quitIn;
list<Guard*> guards;
guards.push_back(quitIn.inputGuard());
guards.push_back(dataIn.inputGuard());
Alternative alt(guards);
switch (alt.priSelect()) {
case 0:
    quitIn >> qs;
    dataIn.poison();
    quitIn.poison();
    return;
case 1:
    dataIn >> data;
    // ... Handle data ...
    break;
}
```

JCSP

JCSP [2] is a concurrency library for Java. It offers the same features in C++CSP2 [3], building upon the CSP model. Compared to C++CSP2, JCSP focuses on more reliable and secure networking, and increases coding safety through promoting CSP specifiers to first-class entries. Other than that, they both provide a thorough CSP library. More information regarding the library and its features are found in the documentation [23].

JCSP follows the same conceptual model as C++CSP2. Processes are defined as classes, which implement the `run` procedure. All process classes inherit the `CSPProcess` class. Processes can be executed in sequence or parallel and are nestable.

Channels also support the four types of disjointness, and are unbuffered, synchronous and type safe. Channels are also explicitly operated on writing and readings ends, handed out from a created channel. Channel poison is also supported.

Alternation is as well a feature, with channel reads as alternatives. Guarded alternatives on conditional booleans are not supported.

2.5.3 Compilers and Run-Time Environments

As mentioned in Section 2.5.2, some CSP libraries have been developed as a consequence on the lack of concurrent programming languages implementing the CSP model. Taking this further, portable compilers and run-time environments have been developed on already existing, unavailable CSP languages, namely occam. Below are the most influential and used compilers and run-time environments summarized.

SPoC

The *Southampton's Portable occam Compiler* (SPoC) [4] is a compiler for occam, making occam programs portable on industry-standard platforms. It achieves this by translating occam source code to ANSI-C, and then using a native C compiler on the generated output. Because of this, SPoC calls itself an occam-to-C translator.

SPoC was the first major implementation of an occam compiler that was not for the Transputer processor architecture. This opened up the use of occam on other architectures, and prov-

ing occam's usefulness as a concurrent programming language.

SPoC works in 7 stages. In succession, lexer, parser, type checker, usage checker, occam-occam translation, C attribution, and C generation.

The lexer takes a source file and performs tokenization, which classifies string of input data into tokens. Any string and numeric constants is also transformed into binary format. The lexer also supports compiler directives of including files in the lexical analysis, as well as tokenization of prototype files.

The parser receives the output from the lexer. From the tokenized input, an abstract syntax tree is generated.

The type checker propagates various attributes around the inputted abstract syntax tree. This is used to detect any type violations in the code. The direction of channel usage is also checked to ensure that each channel has at most one reader and writer.

The usage checker monitors the usage and alias of both scalars and arrays. This is used to check that ranges are not exceeded, as well as parallel read/write and write/write errors between processes in `PAR`.

The occam-occam translator translates occam constructs which are difficult to translate to C. This include pushing processes in `PAR` and in replicated `PAR` into procedures, and `VALOF` blocks into functions. Further simplifications are done on the syntax tree if possible.

The C attribution and code generation is the largest component of SPoC, which takes the transformed abstract syntax tree and generates C code. The code generation consists of 5 main stages, which gradually builds up the static declarations, and the atomic sections of each procedure.

The run-time system of SPoC consists of a scheduler and routines which implement the occam concurrency and communication primitives. The essence is that processes are split into atomic sections, and the processes are statically declared along with its variable usage. The scheduler executes an atomic section at a time. Processes can be seen as stackless coroutines, as all processes and the scheduler uses the same stack. Since all variables are statically stored in the process structure, descheduling does not interfere with the process state.

KRoC

The *Kent Retargetable occam Compiler* (KRoC) [5] is an emulator of the transputer, acting as an architectural mapping, or an emulator, rather than a compiler.

KRoC works by taking the transputer assembly from a compiled occam program and emulating the transputer hardware. The transputer assembly is translated to the target architecture. The translated code is assembled and linked together with a small kernel, which is written in assembly.

The kernel provides functions used by the transputer microcode which are not supported on the targeted architecture. This concerns mainly process scheduling, timers, communications, and event handling.

CCSP

CCSP [6] is a portable run-time system that supports both occam and C, enabling occam-like programming in C. It also supports KRoC occam system for desktop platforms. CCSP aims to fulfill the shortcomings of KRoC, such as support for C and making debugging useful.

CCSP uses assembler macros in the C run-time code to perform flow control between processes and kernel procedures. To avoid stack manipulation, all C code has to be written in the same function using C labels as entry points. The kernel uses these C labels to build a jump table, which it schedules to and from processes. All variables are made static to avoid stack allocations. Parameters are realized by assembler macros which pop parameters from the stack into kernel variables. A return address is also necessary to be specified, as this is needed for returning back to the process when descheduled.

A context switch in CCSP explicitly labels descheduling points, either through some `yield()` function or communication process. To avoid saving lot of context, the kernel informs the compiler that all context will be lost over a CCSP kernel entry call. This minimizes the necessary context to save by the kernel.

Chapter 3

Evaluation of Project Approaches

There are several approaches for creating a CSP abstraction. The three main approaches, *programming language, library* and *run-time system* will be assessed. At the end of the assessment, an approach will be chosen for the project.

3.1 Assessment

The first option is to create a new programming language. This gives total freedom when it comes to design, specifications, and implementation. Two main approaches are outlined, either creating a new language, or creating a superset of an existing language. A superset in this context means the syntax and semantics is based on an existing language, and extending it with new constructs. An example of a new language is occam (Section 2.5.1), while an example of a superset is XC (Section 2.5.1). The programming language implementation has to be considered, usually between the choices of an interpreter, a compiler and a translator. The amount of work and thought that has to be put into designing and implementing a programming language is significant. The easiest approach, considering the amount of time available and competence level required, would be creating a superset of C and implementing a translator for the language. The translator could then translate source code to C-compliant code.

The second option is to create a library. Many successful CSP libraries have been created for other languages such as C++ and Java (Section 2.5.2), which allows reviewing what works and what not. A library has the advantage of being more available, and can be implemented directly

in the native programming language, making it more portable. The downside is the limitations of the language itself. One cannot often create new keywords or operators with libraries[3], and a CSP library could benefit from this.

The third option is to create a run-time system. This has the advantage of being much more flexible than libraries, allowing more direct control of the execution model in the program. Aspects such as task scheduling and resource management are possible to control in run-time systems, which are important to consider for a CSP library. Creating a run-time system is more demanding than a library, requiring more knowledge of the execution model of the programming language.

3.2 Conclusion of Assessment

For this project, with limited time available, creating a new programming language would most likely be too ambitious. The challenge of designing and implementing the programming language is not an easy task, and the result would probably end up as more of a proof of concept rather than a fully fledged framework. A new programming language is probably not something that would gain much popularity either. This does however rule out the possibility of custom syntax and operators, which would be very beneficial.

Creating a library does sort out the problem of availability, as C is one of the most popular and widespread languages still to date. This does limit the library to the limitations of C, which can be a challenge in code expressiveness and memory safety. A run-time system is also necessary for the library, as there needs to be some control of task scheduling in the library.

With this reasoning, the choice of a library with a run-time system for scheduling and resource handling is chosen for this project.

[3] at least not in C, without heavy use of macro magic

Chapter 4

ProXC - The Library

This chapter will go into further details on what basis ProXC is formed from. Details regarding library design and implementation are explained in Chapter 5.

4.1 Introduction to ProXC

ProXC is a CSP-inspired concurrency library for C. The library enables C programs to be decomposed into independent sequential processes, and allows the concurrent execution order of said processes to be specified. Communication and synchronization between processes can be strictly limited to message passing. Message passing is implemented as channels, and are synchronous, pseudo-typed, bi-directional, and any-to-any. Alternation between multiple alternatives is possible, and the alternatives can be guarded on a boolean condition. Alternatives consists of channel reads, timeouts and skip.

ProXC aims to achieve the following goals:

- Enabling CSP paradigms in C programs
- Acceptable performance
- Intuitive and modular API
- No use of macro magic in API
- Low memory footprint
- Low run-time overhead

The idea with ProXC is the ease of use and simple to incorporate in new and existing C pro-

grams. This means the library should not introduce a large number of keywords, macros and functions to be able to use the library. The amount of interactions with ProXC should be minimal, and the programmer should not know the implementation details of the introduced data types and functions. This is why no macro magic is used when interfacing with ProXC.

ProXC is *not* a hand-holding tool for programmers. All existing problems with C, such as NULL pointer dereferencing, memory leaks, pointer aliasing, and much more, is still possible when using ProXC. One can even disregard the CSP model and use global variables instead of message passing. ProXC will not complain. However, ProXC aims to ensure that when a program faults, the origin of the problem is not the library, but the programmer.

The name ProXC, pronounced "*prox-sea*", is an abbreviation of "*Programming XC*". The name is inspired by the CSP language XC (Section 2.5.1). Since XC is a CSP programming language, as well as a superset of C, ProXC makes a fitting name for a CSP library for C.

4.2 Target Platform

ProXC currently only supports 32-bit and 64-bit x86 Linux platforms. This lack of architecture support is mainly due to the implementation of context switching in user-threads, which requires hand-written assembly. Porting to other platforms should not be a difficult task, and is discussed in Section 8.2.4

4.3 Library Features

The complete set of features for ProXC is as follows:

- User-threaded lightweight processes
- Run-time system running a scheduler
- Composite processes of sequential and parallel execution sequence
- Synchronous or asynchronous execution of composite processes
- Bi-directional, pseudo-typed, any-to-any channels
- Alternation on multiple alternatives, consisting of channel reads, timeouts and skip

- Alternatives guarded by a boolean condition
- Implicit and explicit co-operative scheduling, through agreed scheduling points or a `YIELD` command
- Suspension of processes for a relative timeout, granularity of microseconds

4.4 Library API

The *Application Programming Interface* (API) for the library is presented below. The relevant code is found in the files `proxc.h` and `proxc.c`. Note that the header file `proxc.h` is the API header file, and is the only code the programmer interfaces with. The API header file can be found in Appendix A

- `START`: initializes library and run-time system. Takes a function which starts as main process. Must be called before any other library call.
- `EXIT`: exits the library. The run-time system will clean up any resources, and returns as if START call returned. Can be called from any process context.
- `ARGN`: returns the Nth argument for a process.
- `YIELD`: explicitly yields the running process to the scheduler.
- `SLEEP`: suspends the running process for a relative given amount of time. Granularity of microseconds.
- `PROC`: creates a process block for a composite process tree. Takes a function and zero-or-more function arguments.
- `PAR`: creates a parallel block for a composite process tree. Takes one-or-more composite process blocks and returns a parallel block.
- `SEQ`: creates a sequential block for a composite process tree. Takes one-or-more composite process blocks and returns a sequential block.
- `GO`: asynchronously spawns a composite process. Takes one composite process block.
- `RUN`: synchronously spawns a composite process. Takes one composite process block.
- `CHAN_GUARD`: creates a guarded channel read for an alternation process. Takes a conditional, a channel, variable to store data in, and size of the data, and returns a guarded

event.

- `TIME_GUARD`: creates a guarded timeout for an alternation process. Takes a conditional and a relative expiration time, and returns a guarded event.
- `SKIP_GUARD`: creates a guarded skip for an alternation process. Takes a conditional, and returns a guarded event.
- `ALT`: creates and executes an alternation process. Takes one-or-more guarded events, and returns the key for the synchronized event.
- `CHOPEN`: creates a channel of a given type. Takes the size of the data type, and returns a channel.
- `CHCLOSE`: closes a given channel.
- `CHWRITE`: writes data on a given channel. Takes a channel, the variable to be written and the size of the variable, and returns if the operation was a success.
- `CHREAD`: reads data on a given channel. Takes a channel, a variable to write to and the size of the variable, and returns if the operation was a success.

4.5 Influences

Some external influences from existing concurrent programming languages has affected the design choices. The primary source of influences comes from occam, XC and Go, all described in Section 2.5.1. The libraries C++CSP2 and JCSP has also influenced some design choices, all described in Section 2.5.2.

Occam directly influences the use of composite processes, as well as the keywords `PAR` and `SEC` which are also used to describe parallel and sequential processes. The keyword `RUN` comes from C++CSP2 and JCSP, which are used to synchronously execute a composite process. The keyword `GO` is influenced by Go, which uses `go` to asynchronously spawn single functions as goroutines. The keyword `PROC` is influenced by C++CSP2 and JCSP, which explicitly creates class instances as processes in a composite process definition.

Chapter 5

Design and Implementation

This chapter describes the design and implementation of the core library features of ProXC. The presented design and implementation are a single-core implementation.

The library will be written in C, with standard GNU99 dialect. Any details regarding the C programming language will not be explained. Refer to a C reference (e.g. Kernighan and Ritchie [24]) for more detailed descriptions.

Terminology

From here on, the term "*process*" means a single user-thread meant to run sequential C code from a C program. The term "*parent scheduler*" in the context of a process means the scheduler which the process is designated to, i.e. both reside on the same kernel-thread. The term "*running process*" corresponds to the process that is currently executed on a given kernel-thread, given that the scheduler has resumed a process. The term "*ownership*" in the context of a scheduler owning a process means the scheduler is a parent scheduler to the process. The term "*root process*" in the context of composite processes means the process that defined and spawned the composite process.

5.1 Run-Time System

The run-time system of ProXC forms the foundation for how the library can implement a CSP model in C. Some threading model has to be implemented when implementing a CSP model

for programming, which is discussed in Section 2.1, 2.2 and 2.5. The choice of threading model is an important choice regarding performance, and is analysed in Brown [3, chapter 1]. The paper reasons that the context switch time between threads is the major factor in performance, and argues that the hybrid model would be the best choice for the CSP library. For a multi-core implementation, a hybrid model would be the preferred choice. However, as this is a single-core implementation, a user-thread model will be used in this implementation.

The core of the run-time system is the scheduler, which controls the flow of execution from a collection of processes. In the threading model, each kernel-thread has a permanent corresponding scheduler, scheduling multiple processes on said kernel-thread. However, it is important to understand that the scheduler itself is a user-thread on the kernel-thread. Each process is a user-thread. The kernel-thread executes only one user-thread at the time, and the scheduler decides which process is to be resumed by context switching. The context switch only switches between the scheduler and a process context, or vice versa.

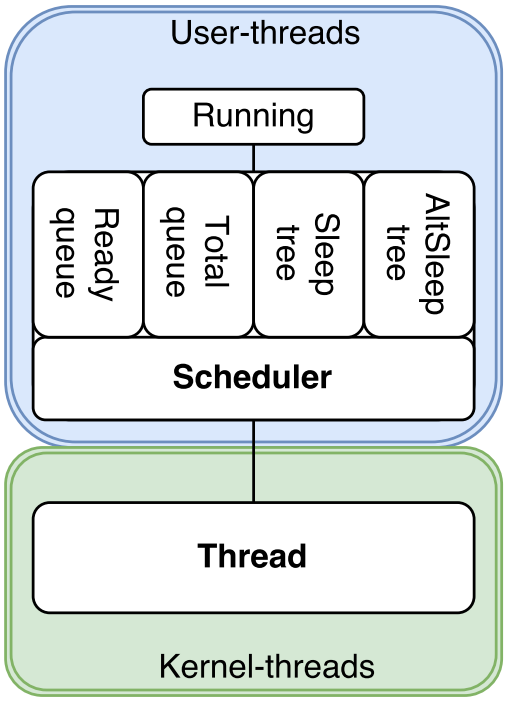

Figure 5.1: Overview of the run-time system, with N online processor cores

When the `START` procedure is called, the run-time system creates and initializes the main process, specified by the programmer. Afterwards, the scheduler is created and initialized. The main process is registered in the scheduler, and the main loop of the scheduler is activated. The

scheduler main loop will continue running until one of two thins happen: the main process returns, or the `EXIT` procedure is called.

5.1.1 Data Structures

Two types of data structures are used by the run-time system: queues and trees. These data structures are not supported in the C standard library, and has to either be implemented or use an existing third-party implementation.

For this project, the BSD libc implementations of queues and trees, `sys/queue.h` [25] and `sys/tree.h` [26], is used respectively. `sys/queue.h` implements four types of queues: singly-linked lists, singly-linked tail queues, lists, and tail queues. `sys/tree.h` implements two types of trees: red-black trees and splay trees.

These implementations are header-only, dependency free, requires no dynamic allocations, and are type-safe. This is highly beneficial for the development of the run-time system. No dynamic allocations will also lower the run-time overhead.

Tail-queues, hereafter called tailq, and red-black trees, hereafter called rb-tree, will be used in the scheduler implementation.

5.1.2 Thread-Local Storage

A *thread-local storage* (TSD) variable is used in multithreaded programs where a single global variable would be inappropriate. An example of such situations is where a global variable is used to store some internal state of the program. Having multiple threads accessing this global variable could overwrite a previously written value from another thread. TSD solves this by letting the variable be global, but exists only once per thread.

TSD is used by the run-time system to retrieve the scheduler link at any given context, as well as what the current running process is. Picture it like this: whenever a process makes a call to the library, the run-time system is invoked. The run-time system has no context on which process or which scheduler the call is made from. Using the TSD, the run-time system finds which scheduler belongs to this kernel-thread, and in turn which process is currently running. With these links, the library call can be processed.

Since this is a single-core implementation, a simple global variable would achieve the same results as a TSD. However, it would not work for a multi-processor implementation. So TSD is used for a cleaner, scalable and portable implementation. TSD does however introduce some slight overhead, but is negligible for this project.

TSD is implemented using `pthread_key_t` type, which is a POSIX implementation. The TSD type `pthread_key_t` stores a single variable of type `void*`. The TSD variable is initialized by the run-time system at start up.

Together with the TSD variable, the internal procedure `scheduler_self` returns a pointer the corresponding scheduler struct for the given kernel-thread. `scheduler_self` is only used internally by the run-time system, and is invisible to the programmer. The POSIX method `pthread_getspecific()` is used to retrieve the actual TSD variable.

Listing 5.1: Procedure to find the scheduler for a given kernel-trhead

```
1 Scheduler* scheduler_self(void) {
2     static pthread_key_t key_scheduler;
3     return pthread_getspecific(key_scheduler);
4 }
```

`scheduler_self` can also be used to find the current running process struct, as the scheduler always records which process is currently running. It is called `process_self`, and is also an internal procedure for the run-time system, just as the `scheduler_self` procedure.

Listing 5.2: Procedure to find the current running process

```
1 Process* process_self(void) {
2     return scheduler_self()->running_process;
3 }
```

5.1.3 Stackful Coroutines

User-threads are implemented as coroutines. Each user-thread is its own coroutines. There are multiple ways to implement coroutines. Most known approach is using the library routines `setjmp` and `longjmp` to provide non-local flow between different call frames on the stack. This is however not a portable solution, as jumping down the stack relies on undefined behaviour. The C standard does not specify wheter deallocated stack frames is retained in memory when

jumping down the stack. This would break on architectures where deallocated stack frames are deallocated when jumping down the stack.

Another approach is using a duff's device [27], splitting up a process into atomic instructions between scheduling points. This is both portable and does not rely on undefined behaviour. However, the procedure of splitting up and defining the atomic instructions can be difficult to implement. This has usually in existing libraries been implemented using macro magic, which is not favorable for this implementation.

The third approach is stackful coroutines, which is used in this implementation. This allows for fast context switches between coroutines, as they are usually implemented in a few assembly instructions. However, the stack has in most implementations fixed size, which makes stack overflow a potential problem. This is discussed in more detail in Section 8.2.2.

Coroutine Implementation

All relevant code for coroutine implementation are found in files `context.h` and `context.c`.

Each coroutine has its own stack and a context. The stack acts as the stack frame of the coroutine execution, and is either allocated on the stack (of the main process) or on the heap. The context is a snapshot of the processor registers at a given execution time, and is stored in a C-struct. The snapshot consists of registers that must be preserved by the callee and the program counter. See Section 2.4 for a more detailed description for why.

When created, the stack is allocated on the heap. The stack pointer is positioned at the top of the allocated memory, and is 16-byte aligned. The return address is stored on the stack. The context structure is initialized, and the process function pointer, stack base pointer and stack pointer is stored in the corresponding registers.

A context switch from a coroutine *c1* to a coroutine *c2* does the following: *c1* invokes a context switch from its own context to the context of *c2*. A snapshot of the processor registers are stored in *c1*'s context struct, and the snapshot stored in *c2*'s context struct is loaded into the processor registers. The program counter is the last register to be loaded, as this triggers the continuation of *c2*. From now on, the processor is executing coroutine *c2*.

Currently, context switching is only implemented for x86_64 and i386 platforms. The context struct for i386 and x86_64 is in Listing 5.3 and 5.4, respectively. Note that each register is stored

as 32-bit or 64-bit variables, since i386 is a 32-bit platform and x84_64 is a 64-bit platform, and the registers are of such width.

Listing 5.3: Context struct for i386

```
1  struct Context {
2      // preserved registers
3      uint32_t ebx;
4      uint32_t esi;
5      uint32_t edi;
6      uint32_t ebp;
7      uint32_t esp;
8      // program counter
9      uint32_t eip;
10 };
```

Listing 5.4: Context struct for x86_64

```
1  struct Context {
2      // preserved registers
3      uint64_t rbx;
4      uint64_t rsp;
5      uint64_t rbp;
6      uint64_t r12;
7      uint64_t r13;
8      uint64_t r14;
9      uint64_t r15;
10     // program counter
11     uint64_t rip;
12 };
```

The context switching procedure has the following C function prototype.

```
void context_switch(Context *from, Context *to);
```

The context switching procedure is however implemented in inline assembly. Direct access to processor registers is not supported in native C. By insert inline assembly into the C program, one can directly access the registers. See Appendix B for full implementation.

5.1.4 Yielding

Yielding is used to transfer the flow of control from a running process back to the scheduler. To achieve this, the context of both the running process and the scheduler must be available. Getting either the scheduler pointer or the process pointer is enough, as the scheduler and the process has a pointer to each other.

For the internal process, calling the `scheduler_self` would be sufficient. However, there is an overhead penalty of acquiring the TSD variable. The yielding procedure is a frequently called procedure by the run-time system, and the overhead would quickly accumulate. A simple optimization would be passing the process pointer if the caller already has access to it, and if not

a NULL pointer is passed.

See Listing 5.5 for both internal and external procedure implementation.

Listing 5.5: Internal and external yielding procedure

```
1  // Internal procedure
2  void process_yield(Process *process) {
3      Scheduler *scheduler = NULL;
4      if (process == NULL) { // Process struct is not known
5          scheduler = scheduler_self();
6          process = scheduler->current_process;
7      } else { // Process struct is known
8          scheduler = process->scheduler;
9      }
10     context_switch(&process->context, &scheduler->context);
11 }
12 // External procedure
13 void YIELD(void) {
14     process_yield(NULL);
15 }
```

Note that the external procedure has to call the yielding procedure with NULL as argument, since the process pointer is not available. The internal process will then find what the process pointer is.

5.1.5 Descheduling Points

The ability to express concurrency relies on running processes giving the control of execution back, or yielding, to the scheduler at regular intervals, since the scheduler does not use preemption to regain control. Relinquishing control is done through designated descheduling points, either implicitly defined in library calls, or explicitly defined through the YIELD command.

The following library calls and programming contexts will result in a yield:

- YIELD
- EXIT
- SLEEP
- RUN
- Processes returning

The following may result in a yield:

- `GO`
- `ALT`
- `CHWRITE`
- `CHREAD`

The following will not result in a yield:

- `ARGN`
- `PROC`
- `PAR`
- `SEQ`
- `CHAN_GUARD`
- `TIME_GUARD`
- `SKIP_GUARD`
- `CHOPEN`
- `CHCLOSE`
- Normal function calls
- Normal functions returning

5.2 Scheduler

The scheduler keeps track of the execution state of a kernel-thread. The scheduler knows which process is the main process and keeps track of the current process running.

As mentioned above, the scheduler runs in its own user-thread. It is however important to state that the scheduler does not need to allocate a stack for its user-thread, as it uses the user stack in the kernel-thread as its own local stack. This is not the case for processes, which is explained in Section 5.3.

Two types of data structures are used in the scheduler. It has two queues: a ready queue and a total queue. It also has two trees: a sleep tree and a alternating guard sleep tree. The ready queue is a queue of processes that are ready to be executed. Processes can be added to the ready queue by any other process, or the scheduler itself. The total queue is the total collection of all processes that are present in the scheduler. The sleep tree is a minimal ordered tree of sleeping processes, where the root node of the tree is the process with the smallest time left sleeping. The alternation sleep tree is the same as the sleep tree, where it contains alternating processes.

5.2.1 Scheduler Phases

The schedulers main loop has three phases: prologue, context switch, and epilogue. Each phase corresponds to what it does before, under and after a context switch to a process.

In prologue the scheduler does a termination test, checks if any processes are available, wakes up any timed out processes, and finds next process to resume. The termination test succeeds if either the total queue is empty, or the exit flag is set. Checking the ready queue ensures there are any processes available, and if not, suspends the kernel-thread until a process becomes ready. A process becomes ready by either timing out in the sleep trees, or is added to the ready queue by an another scheduler. When a process is ready, the scheduler checks the sleep trees and adds any processes that have timed out to the ready queue. Lastly, it finds which process from the ready queue to resume.

After prologue, the scheduler now has a process to resume execution to. First, the process is registered as the running process in the scheduler. Registering the process is used to retrieve the running process link, described in Section 5.1.2. Then, the actual context switch to the process happens. From here, the flow of execution in the kernel-thread jumps from the scheduler to the context of the process. Whenever the process is descheduled, either through an implicit or explicit descheduling point, the flow of execution is returned to the scheduler, and it resumes as if the function call returned.

In epilogue, the state of the returned process is checked and appropriately handled. Currently only the return states of an unaltered process state or an ended process state is handled. The process is added back to the ready queue, or the process is cleaned up, respectively. If the ended process is the main process, the exit flag in the scheduler is set. Lastly, the running process information is reset, and loops back to prologue.

In Listing 5.6 is the pseudocode of the following phases shown.

It is important to note that when entering epilogue, the state of the returned process is changed in the context of the process. So even though the scheduler sets the state of the process to *Running* in prologue, the process state can and will be changed when execution is returned to the scheduler.

Listing 5.6: Pseudocode for scheduler main loop

```
1  void scheduler() {
2      // Phase 1: Prologue
3      while ( termination_test() ) {
4          wait_for_readyQ_if_empty();
5          check_timedout_processes();
6          next_process = find_next_process();
7
8          // Phase 2: Context switch
9          register_running_process(next_process);
10         context_switch(next_process);
11
12         // Phase 3: Epilogue
13         check_state(next_process);
14         reset_running_process();
15     }
16     // Here the scheduler exits, and the run-time takes over
17 }
```

5.2.2 Scheduler Implementation

All relevant code for scheduler implementation is found in the following files `scheduler.h` and `scheduler.c`.

The scheduler is realized by a scheduler struct with method functions which operates on a given scheduler struct. The scheduler struct type is shown in Listing 5.7.

The scheduler constructor is called `scheduler_create`. When created, the scheduler struct is allocated on the heap. The default stack size for the coroutines is determined, the page size of the memory management unit is stored, and the exit flag is nilled out. The scheduler struct pointer is then registered in the TLS variable. The context of the scheduler and the two tailqs and rb-trees are initialized.

The scheduler destructor is called `scheduler_free`. When cleaned up, the total queue is iterated over and calls the cleanup procedure for each process. Lastly, the scheduler struct pointer is freed up.

Listing 5.7: Scheduler struct type

```
1  typedef struct {
2      Context context;
3      size_t  stack_size;
4      size_t  page_size;
5      int     is_exit;
6      Process *main_process;
7      Process *running_process;
8      // queues
9      struct ProcessQ totalQ;
10     struct ProcessQ readyQ;
11     // trees
12     struct ProcessRB sleepRB;
13     struct PRocessRB altsleepRB;
14 } Scheduler;
```

When checking the ready queue and no processes are ready, the kernel-thread is suspended by sleeping the minimal expiration time in either sleep or alternation sleep rb-tree. The wake-up procedure, which checks both sleep rb-trees for expired timeouts, are straight forward. However, some slight care has to be taken with the alternation sleep tree. Whenever a sleeping alternation process has expired, the scheduler must check if the process has not already been added to the ready queue by another process. The scheduling policy is currently *First In, First Out* (FIFO) for the ready queue.

Registering the current process is done by setting the next process as the current process, and setting the state of that process to *Running*. After this, the process is resumed by context switching. When the context switch returns, one additional action is made. A procedure checking the stack usage of the process will advise the kernel of memory usage if the stack usage exceeds a certain limit. This reduces the total memory footprint of the run-time system. See Listing 5.8 for reference.

Listing 5.8: Scheduler registering and context switch to running process

```
1 next_process->state = PROCESS_RUNNING;
2 scheduler->current_process = next_process;
3 context_switch(&scheduler->context, &next_process->context);
4 scheduler->current_process = NULL;
5 memory_advise(next_process->stack);
```

During epilogue, the state of the process is checked. This reflects on why the process yielded back to the scheduler. If the process state is *Ready* or *Running*, the process is added back to the ready queue. If the state is *Ended*, the termination condition is checked, the composition tree is re-parsed if the process is part of one, and lastly the process struct is freed. See Listing 5.9 for reference.

Listing 5.9: Handling of process state in epilogue

```
1  switch(next_process->state) {
2  case PROC_RUNNING:
3  case PROC_READY: // fallthrough
4      scheduler_addreadyQ(scheduler, next_process);
5      break;
6  case PROC_ENDED:
7      // Determine termination status
8      scheduler->is_exit = scheduler->is_exit
9                          || (next_process == scheduler->main_process);
10     //
11     composite_parse(next_process->composite_block);
12     process_free(next_process);
13     break;
14 case PROC_WAIT:
15 case PROC_SLEEP: // fallthrough
16     // Do nothing
17     break;
18 case ERROR:
19     process_error(next_process);
20     break;
21 }
```

5.3 Processes

A process is a sequential execution of C code. In ProXC, and consequently in C, the sequential code is written as a normal function. The process keeps track of which function to execute and its arguments, the coroutine associated with the process, and the state of the process. The process does also express some notion of ownership, as it acknowledges the parent scheduler is the only scheduler it is assigned to.

The processes and the scheduler are both executed as user-threads. This means each user-thread has its own stack and context, represented as a coroutine.

5.3.1 Process States

The process state describes the current state of the process after a yield. In the run-time system, the process state is used for a couple of things: informing the scheduler the reason for why the process returned control, and knowing when the process ended. The complete finite state machine of all possible process states and transitions are shown in Figure 5.2.

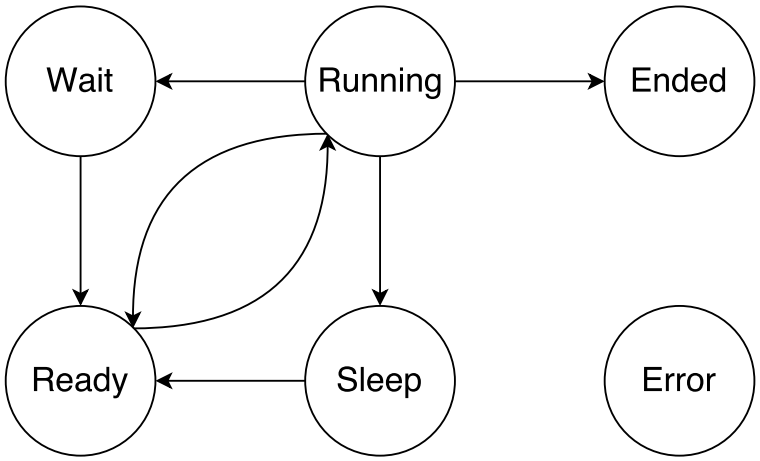

Figure 5.2: Finite state machine of process states and transitions

The process states describes the following:

- **Ready state**: the process is in a scheduler's ready queue and is ready to be resumed.
- **Running state**: the process is currently being executed on a kernel-thread, and the scheduler is waiting for the process to return control.

- **Ended state**: the process has reached the end of its sequential code, and can safely be cleaned up.
- **Error state**: the process has entered an unrecoverable state.
- **Wait state**: the process is waiting for an another process to release it. This can happen during synchronization points such as channel communication and alternation without a timeout.
- **Sleep state**: the process is suspended for a given amount of time, and is waiting for time out. This happens during explicit suspension or alternation with a timeout.

A couple of constraints regarding the process states and transitions needs to be addressed. When a process is created, the process state is initially set to *Ready*. Only the parent scheduler can transition the process state from *Ready* to *Running*. This transition transfers the flow of control from the parent scheduler to the running process. Only one process can be in the *Running* state at the time on a given scheduler. This should be logical, given that only one user-thread can be executed at a time on a kernel-thread. The transition from *Running* to either *Ready*, *Ended*, *Wait* or *Sleep* can only be done by the running process itself. This transition also follows that the flow of control is transferred back to the parent scheduler. The transition from *Wait* or *Sleep* to *Ready* is either done by the parent scheduler or by other running processes. The transition to *Error* can happen from anywhere at any time, and is used by the run-time system to detect unrecoverable process states.

5.3.2 Process Implementation

All relevant code for scheduler implementation is found in the following files `proc.h` and `proc.c`.

A process is realized by a process struct with method functions which operates on a given process struct. The process struct type is shown in Listing 5.10. The scheduler related node members is not directly used by the process struct, but by the data structure methods.

Listing 5.10: Process struct type

```
typedef void (*ProcessFxn)(void);
struct Process {
    Scheduler *scheduler;
    enum {
        PROC_ERROR = 0,
        PROC_READY,
        PROC_RUNNING,
        PROC_ENDED,
        PROC_SLEEP,
        PROC_WAIT
    } state;
    // Fxn and args
    ProcessFxn fxn;
    struct {
        size_t num;
        void   **ptr;
    } args;
    // Coroutine related
    Context context;
    struct {
        size_t size;
        size_t used;
        void   *ptr;
    } stack;
    // Process sleep metric
    uint64_t sleep_usec;
    // Composite process related
    Composite *composite_block;
    // Scheduler tailq and rb-tree nodes
    TAILQ_ENTRY(Process) totalQ_node;
    TAILQ_ENTRY(Process) readyQ_node;
    RB_ENTRY(Process) sleepRB_node;
    RB_ENTRY(Process) altsleepRB_node;
};
```

The process constructor is called `process_create`, and takes a function pointer as an argument. The function pointer is the function the process is to execute. Only the function pointer is set, not the arguments. When created, a coroutine is allocated and initialized. The state is set to *Ready*, and the scheduler pointer is set to the appointed scheduler through `scheduler_self` method. The rest of the struct members, except the tailq and rb-tree nodes, are zero-initialized. Lastly, the process struct is inserted in the total queue of the scheduler. This gives the ownership of the process to the scheduler.

The process destructor is called `process_free`. The process is removed from the total queue of the scheduler, and the stack and argument array is freed. Lastly, the process struct is freed.

The process execution flow does not start from the specified function pointer. Instead, the process execution starts at an internal library function `process_mainfxn`. This procedure appropriately calls the function pointer, and sets the process state to *Ended* and yields when the process function returns. The process main function is shown in Listing 5.11. This approach of abstracting the function call away, at Line 2 in Listing 5.11, constrains the number of function arguments to a fixed number if the arguments were to be passed directly in the function call.

Listing 5.11: Process main function

```
1 void process_mainfxn(Process *process) {
2     process->fxn();
3     process->state = PROC_ENDED;
4     process_yield(process);
5     // This is never reached
6 }
```

Allowing arbitrary number of function arguments is much more expressive than a fixed number, which is why the arguments are rather stored as an argument array in the process struct, seen at Line 16 in Listing 5.10. This argument array can then be accessed through the API function `ARGN`, see Listing 5.12. This does however constrain the type of arguments to only pointers.

Listing 5.12: ARGN API function

```
1 void* ARGN(size_t n) {
2     Process *process = process_self();
3     return (n < process->args.num)
4         ? process->args.ptr[n]
5         : NULL;
6 }
```

The added flexibility of having arbitrary number of arguments does, in my opinion, outweigh the added overhead of storing and accessing the arguments through API calls.

Since function arguments are optional, the argument array is populated by a separate method `process_setargs`. It takes a variadic list of void pointers, allocates an argument array, and copies over the pointers from the variadic list to the argument array. An example of how this is used by the run-time system when creating a process is shown in Listing 5.13.

Listing 5.13: Process creation example example

```
1  void process_setargs(Process *process, va_list args);
2  // Takes function arguments as a variadic list
3  Process* example_create_process(ProcessFxn fxn, ...){
4      Process *process;
5      process_create(&process, fxn);
6      va_list args;
7      va_start(args, fxn);
8      process_setargs(process, args);
9      va_end(args);
10     return process;
11 }
```

This is why the process constructor does not allocate and populate the argument array, while the destructor frees the argument array.

5.4 Composite Processes

Processes are responsible for defining and spawning new processes, which is done in a process context during process execution. Five new keywords are introduced to be able to express com-

posite processes: `PROC`, `SEQ`, `PAR`, `GO`, and `RUN`.

5.4.1 Composite Process Representation

A composite process can be described as a node tree. At the root node of the tree is an execution block, `GO` or `RUN`. Every leaf node of the tree is a `PROC` block. Branch nodes consists of execution order blocks, `SEQ` or `PAR`, and are nestable. The behavior of each block is specified as follows:

- `PROC`: defines a *new process*. Takes a function and zero-or-more function arguments, and returns a leaf node.
- `SEQ`: defines an *execution order*. Takes one-or-more leaf or branch nodes, and returns a branch node. The execution order of these child nodes is specified to be executed *sequentially*, from left to right.
- `PAR`: defines an *execution order*. Operates just as a `SEQ` block, however it specifies the execution order of the child nodes to be *parallel*.
- `GO`: defines the *type of execution*. Takes one leaf or branch node, and specifies the type of execution to be *asynchronous*. The root process **DOES NOT** wait for the composite process to finish.
- `RUN`: defines the *type of execution*. Operates just as a `GO` block, however it specifies the type of execution to be *synchronous*. The root process **DOES** wait for the composite process to finish.

Figure 5.3 visually represents each composite block. Remember that the `SEQ` and `PAR` blocks can have one-or-more child nodes, even though the figure specifies three.

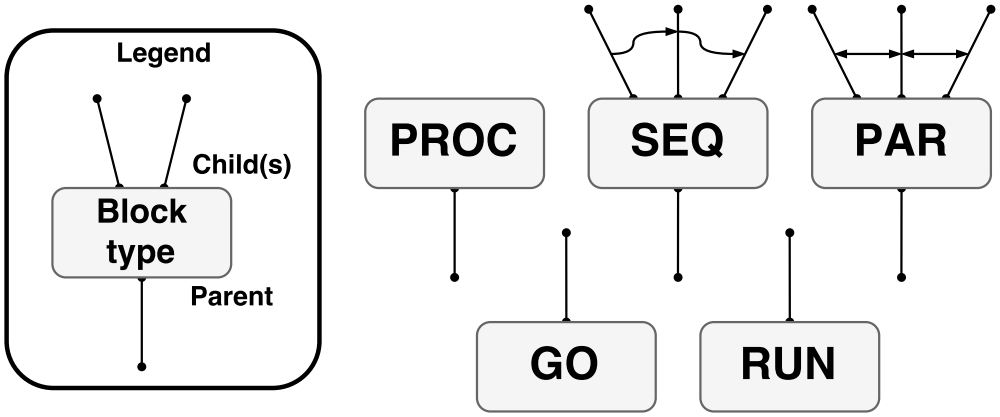

Figure 5.3: Building blocks for composite processes

The combination of these building blocks can form any composite process. What is important to take away from this is that a composite process must always contain one, and only one, root node of either `GO` or `RUN`, and always contain one-or-more leaf nodes of `PROC`.

Branch nodes, `SEQ` or `PAR`, with only one child does not alter the composite process behaviour. This becomes evident when we define a composite process of only one `PROC`, where it does not matter if multiple branch nodes of `SEQ` or `PAR` are present. This means branch nodes with one child are optional and redundant. See Figure 5.4 for reference.

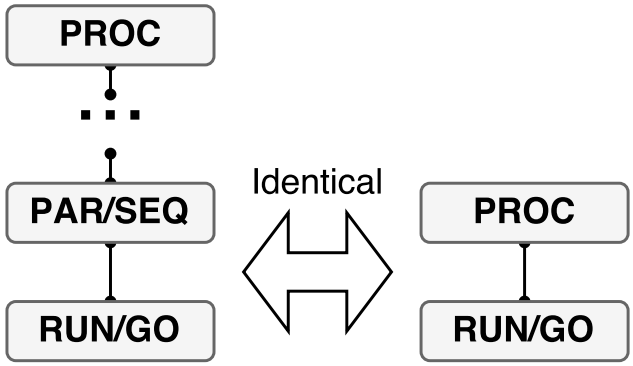

Figure 5.4: Redundancy of branch nodes with one child

A composite process with a single process does not need a branch node. However, a composite process with two-or-more `PROC` cannot be expressed without at least one-or-more branch nodes.

5.4.2 Composite Process Execution

When a process defines a composite process, the run-time system parses and creates a composite process tree. The running process is marked as the root process of the composite process tree. The composite process tree is then traversed by the run-time system, described by the pseudo-code in Listing 5.14.

Listing 5.14: Pseudocode for the composite process tree traversal algorithm

```
1  // Called with the child node of the root node
2  int traversal(node) {
3      switch (node.type) {
4      case PROC: process = create_process(node.fxn_and_args);
5                 reschedule(process);
6                 return Ok;
7      case SEQ: // fallthrough PAR
8      case PAR: if (node.num_childs < 1) return Error;
9                if (node.num_childs == 1) {
10                   node.parent.this_node = node.child;
11                   return traversal(node.child);
12               }
13               node.childs_left = node.num_childs;
14               if (node.type == SEQ)
15                   return traversal(node.leftmost_child);
16               for_each(child in node.childs)
17                   if (traversal(child) != Ok) return Error;
18               return Ok;
19     }
20     return Error;
21 }
```

After the composite process tree is traversed, all initial processes are created and added to the ready queue in the scheduler. One thing to remark about the algorithm is that process creation is lazily evaluated [28], which means the processes are created when needed. This is entirely the consequence of the `SEQ` block behavior.

Whether the execution type is specified to be asynchronous or synchronous, the root process continues execution or transitions process state from *Ready* to *Wait*, respectively. For the

synchronous case, the root node transitions from *Wait* to *Ready* when all PROC in the composite process tree has ended.

However, one also has to take into consideration how the execution order of the composite process tree behaves when a PROC ends. The pseudocode in Listing 5.15 shows how this is processed by the run-time system.

Listing 5.15: Pseudocode for the composite process tree PROC ended algorithm

```
1  // Called with the ended PROC
2  int proc_end(node) {
3      while (true) {
4          switch (node.type) {
5          // fallthrough on GO
6          case RUN:  reschedule(node.root_process);
7          case GO:   return Ok;
8          case PROC: node = node.parent;
9                     continue;
10         case SEQ: // fallthrough
11         case PAR: if (--node.childs_left == 0) {
12                       node = node.parent;
13                       continue;
14                   }
15                   return (node.type == SEQ)
16                       ? traversal(node.next_in_sequence)
17                       : Ok;
18         }
19         return Error;
20     }
21 }
```

The algorithm starts at the ended PROC and backtracks down the tree until hitting either the root node or a branch node with remaining children left. Whenever entering a branch node, decrement the number of children left for the given branch node. A resulting value of zero indicates all children of the given branch has ended, and backtrack to the parent of the branch node. A resulting non-zero value indicates there are still children of the given branch node that are unfinished. For a PAR block this means the remaining children are dispatched and will in-

voke the algorithm when ended. For a SEQ block this means there is a sequence of one-or-more children remaining to be dispatched, for which the travel algorithm is called on the next child in the sequence. This is described on the lines 10 to 17 in Listing 5.15.

The rescheduling of the root process in a synchronous composite process is described on line 6 in Listing 5.15.

5.4.3 Composite Process Implementation

All relevant code for scheduler implementation is found in the following files `csp.h` and `csp.c`.

Composite processes are implemented as node trees. The challenge is how the composite tree blocks are implemented. One approach is to create a "god" struct for the block type, containing all data necessary for each block type. Another more modular approach is to create a struct for each composite tree block type, and one generic block type. The generic block is the interface, which is type casted to the corresponding composite tree block type. The latter approach is used in this implementation.

Listing 5.16: Header struct type

```
1  typedef struct {
2      enum {
3          PROC_BLOCK,
4          PAR_BLOCK,
5          SEQ_BLOCK
6      } type;
7      Block *parent;
8      int is_root;
9      Process *root_process;
10     // node tree related
11     TAILQ_ENTRY(Block) node;
12 } Header;
```

Five structs are defined: a Header, Block, ProcBlock, ParBlock, and SeqBlock. Go and Run is not used as they are optimized away, explained later. The header struct defines the common interface between the generic block and the typed blocks. Listing 5.16 shows the header struct. The type of the block is set in the header struct, the parent of the block, some variables for

finding the root of the composite process tree, and a pointer to an optional root process. The node variable is for Block queues in PAR and SEQ blocks.

All structs Block, ProcBlock, ParBlock and SeqBlock must have the header struct as a member variable, and it must be the first member variable. With the header struct member, composite process procedures and block queues can work on the generic block struct rather than one for each block type. To determine the block type, inspect the header member. Then reinterpret cast the block pointer to the correct block type through type punning. Listing 5.17,5.18,5.19,5.20 shows the different block types.

Listing 5.17: Generic Block struct type

```
1 typedef struct {
2     Header header;
3 } Block;
```

Listing 5.18: PROC Block struct type

```
1 typedef struct {
2     Header header;
3     Process *process;
4 } ProcBlock;
```

Listing 5.19: PAR Block struct type

```
1 typedef struct {
2     Header header;
3     struct {
4         size_t num;
5         struct BlockQ Q;
6     } childs;
7 } ParBlock;
```

Listing 5.20: SEQ Block struct type

```
1 typedef struct {
2     Header header;
3     struct {
4         size_t num;
5         Block *current;
6         struct BlockQ Q;
7     } childs;
8 } SeqBlock;
```

Type punning is done through union casting. In other words, a pointer is casted to a union of the source and destination type, and reading from the destination type yields the reinterpreted pointer. This is the correct way to achieve type punning without breaking strict aliasing. A convenience macro can be made to easily achieve type punning, shown in Listing 5.21. Note that union casting and __typeof__ is a GNU C extension.

Listing 5.21: Type punning through union cast

```
1 #define BLOCK_CAST(block, destType) \
2     (((union {__typeof__(block) src; destType dst;})block).dst)
```

Listing 5.22 shows how type punning is used to convert from a generic block struct to the given typed block, and back from typed block to generic block type. The union casting is used anytime when the run-time system casts a pointer either way between a generic block and a typed block.

Listing 5.22: Type punning example

```
1  Block* type_punning_example(Block *block) {
2      switch (block->type) {
3      case PROC_BLOC: {
4          ProcBlock *proc_block = BLOCK_CAST(block, ProcBlock*);
5          // ... work ...
6          return BLOCK_CAST(proc_block, Block*);
7      }
8      case PAR_BLOCK: {
9          ParBlock *par_block = BLOCK_CAST(block, ParBlock*);
10         // ... work ...
11         return BLOCK_CAST(par_block, Block*);
12     }
13     case SEQ_BLOCK: {
14         ProcBlock *seq_block = BLOCK_CAST(block, SeqBlock*);
15         // ... work ...
16         return BLOCK_CAST(seq_block, Block*);
17     }
18 }
```

Defining a composite tree process is done through the API calls `PROC`, `PAR`, `SEQ`, `GO`, and `RUN`. The function prototypes for the API calls are shown in Listing 5.23. Both `PAR` and `SEQ` has a single child block as first argument, because a non-variadic argument must be specified before the variadic list. Also, if the variadic list is empty, the given block can be optimized away as a result of the redundancy identity shown in Figure 5.4.

Listing 5.23: Composite process API prototypes

```
1 Block* PROC(ProcessFxn fxn, ...);
2 Block* PAR(Block *first_child, ...);
3 Block* SEQ(Block *first_child, ...);
4 void GO(Block *root);
5 void RUN(Block *root);
```

Currently, the API calls `PROC`, `PAR` and `SEQ` allocates a typed block on the heap. `GO` or `RUN` does not however allocate any typed block on the heap. `PROC` also creates a process with the given function pointer and arguments, and sets the process pointer in the typed block to the created process. `PAR` and `SEQ`, if two or more children given, inserts each child in the block queue and records the number of children. `SEQ` initializes the current block pointer to the first child.

Both API calls `GO` and `RUN` receives a root block, which the root flag of the block is set to true. `GO` traverses the resulting composite process tree, starting at the root block. When the traversal returns, the API call returns as well, and the running process continues execution. `RUN` sets the root process pointer to the running process, and traverses the resulting composite process tree, starting at the root block. When the traversal returns, the running process yields with the process state set to *Wait*. The running process is resumed when the root block finishes.

The traversal implementation is very straightforward, following the pseudocode in Listing 5.14. The only difference is process creation happens in the API call `PROC`, rather in the traversal procedure. This has to do with the simplifications of the variadic list handling.

The `PROC` ended implementation follows the pseudocode in Listing 5.15 with some slight modifications. The `RUN` and `GO` cases are removed, as they are simplified to a root flag. Whenever either `PROC`, `SEQ` or `PAR` block, a `block_done` variable is set to true. This is checked at the end of the switch case, and if the current block is the root block, the root process is rescheduled if the composite tree is synchronous.

An added cleanup procedure must be called when the root block finishes, as the entire tree is allocated on the heap. This is a simple tree traversal, freeing up each block in a post-order fashion.

5.5 Timers

Timers are used to suspend a process for a given amount of time. This is used in two situations: in a normal process execution, or in an alternation process.

In a normal process execution, whenever a timeout is specified, the process is suspended. The run-time system places the process in the sleep tree based on the specified timeout period. The process is resumed by the run-time system when the timeout period is expired.

In an alternation process, the timeout is specified as a guarded alternative. When the alternative is activated, the alternation process is placed in the alternation sleep tree based on the specified timeout period. One of two things happens: either the timeout period is expired and the run-time system registers the timeout alternative in the alternation available set. If the alternation process is still suspended, resume it. Or, the alternation process is resumed before the timeout expiration, and the timeout alternative is deactivated, removing it from the alternation sleep tree.

5.5.1 Timer Implementation

Timers is realized through the sleep trees in the scheduler. Invoking a timer on a process is done through the API call `SLEEP`, which takes an expiration time in microseconds as an argument. If the expiration time is zero, the process yields with no additional changes. If non-zero, the process registers the expiration time in the process struct, and adds itself to the sleep tree in the scheduler. Then, yielding with the state *Sleep*. When the time expires, the scheduler will reschedule the sleeping process.

For alternation constructs, the timeout guard registers the process in the alternation sleep tree. The same functionality applies here for normal timers. However, if the alternation construct completes the selection process before the timeout occurs, the alternation process removes itself from the alternation sleep tree.

5.6 Channels

Channels form the means of interprocess communication and synchronization, following the message passing model. There are mainly three aspects of channel design that must be considered, namely synchronous or asynchronous communication, channel disjointness, and type-safe data transfer.

5.6.1 Synchronous and Unbuffered vs Asynchronous and Buffered

Coming from the CSP model, channels are implemented as unbuffered and synchronous, and are sometimes called a synchronous rendezvous. This is used as a one-way transfer of data between a receiver and a sender. Both the receiver and sender must be available on the channel for the transfer to occur. If one of the ends are not present, the other end must wait until both ends are "ready".

The channel design follows the CSP model, being unbuffered and synchronous.

5.6.2 Channel Disjointness

Some channels have a notion of disjointness [see 15, chapter 3.3.1], setting restrictions on how many processes are allowed to use a channel, and how many unique senders or receivers a channel can have. These designs are often denoted as a one-to-one, any-to-one, one-to-any, or any-to-any channels, following the *#Senders*-to-*#Receivers* scheme.

Creating four version of the channel design, one for each of the four combinations of *one* and *any*, might turn out to be a flexible approach. This allows the programmer to handpick the channel type for certain use cases, i.e. one-to-any for multiple readers and one writer, etc..

One might argue "Why not just have an any-to-any design, and cover all use cases?". Enforcing disjointness increases the expressiveness of the abstraction, allowing the programmer to create more correct programs that reflect a truer behaviour of the system. As well, this also gives the implementation some assumptions allowing better optimizations for each design.

However, for the sake of simplicity and as a result of the limitations of C, only the any-to-any version will be implemented.

5.6.3 Type-Safety

Channels are also used to transfer data between processes. Type errors can be an issue in a data transfer, where the type of data sent over a channel does not match the type of data expected to be received.

An example can be a sender and a receiver on a channel with no type-safety. Sender sends a message of type A, which has a 4 byte size representation. Receiver B expects to receive a message of type B, which has a 2 byte size representation, and allocates the appropriate memory. When the transfer of data occurs, the channel writes the 4 bytes of data to the memory allocated only for 2 bytes. 2 bytes are written past the allocated memory, and memory corruption has occurred. See illustration in Figure 5.5 for reference. This illustrates the dangers of not enforcing type-safety on channels.

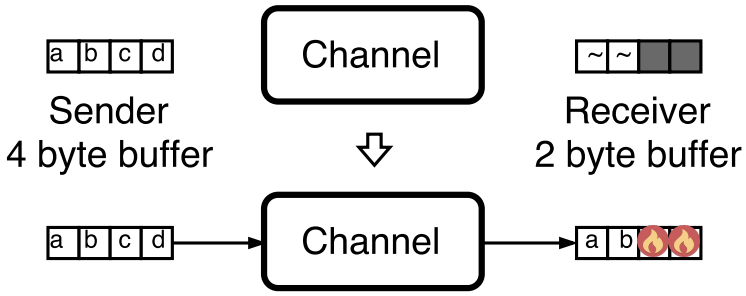

Figure 5.5: Memory corruption from conflicting types between sender and receiver

Type-safe channels enforce only transfer of messages of only one type. In turn, this enforces the sender and receiver to agree on the type of the message. Type safety should be favorable, as type errors should be treated as bugs in a program. Channels should therefore be as type-safe as possible.

C is however known for not being very type-safe. The language enforces type-safety to some degree, but is easily circumventable through type casting to pointer types. A common source of type errors in C programs is pointers casted from and to void pointers, `void*`. Void pointers are used for memory with an unspecified type, which is convenient for implementation of generic interfaces. This is most likely how channels will implement data transfer, being compatible with any type. Knowing this, complete type-safe channels in C will be impossible without code bloat

and breaking encapsulation[4]. A compromise is size-safe channels. Instead of ensuring the type matches, the size representation of the type must match on both sides. This should not require much overhead, and at least avoids memory corruption, illustrated in Figure 5.5.

5.6.4 Channel Implementation

All relevant code for channel implementation is found in the following files `chan.h` and `chan.c`.

This channel implementation implements a synchronous, unbuffered, size-safe channels. However, disjointed channels proved to be to difficult to implement, which is why only any-to-any channels are used. This is explained more later.

Channels is realized by a channel struct and a channel end struct. A channel struct represents a created channel, while a channel end represents a process waiting on a channel. The two struct are shown in Listing 5.24 and 5.25.

Listing 5.24: Channel type struct

```
1 typedef struct {
2     size_t data_size;
3     struct ChanEndQ endQ;
4     struct ChanEndQ altQ;
5 } Chan;
```

Listing 5.25: Channel end type struct

```
1  typedef struct {
2      enum {
3          CHAN_WRITER,
4          CHAN_READER,
5          CHAN_ALTER,
6      } type;
7      void *data;
8      Chan *chan;
9      Process *process;
10     AltGuard *guard;
11     TAILQ_ENTRY(ChanEnd) node;
12 } ChanEnd;
```

The channel constructor is called `chan_create`, and takes the size of the data type as an argument. The channel struct is allocated on the heap. The data size is stored, and the two channel end queues are initialized. The API call `CHOPEN` calls the channel constructor and returns the channel struct pointer.

The channel destructor is called `chan_free`, and frees the channel struct. The API call

[4]This is possible, but requires heavy use of macros and leaking implementation details to the programmer

`CHCLOSE` calls the channel destructor.

The channel struct contains two channel end queues, called hereafter end queue and altend queue. These queues are used when processes has to wait on a channel for a process to be available on the other end. Two queues are used instead of one, as alternation waiting is separated from normal waiting. The end queue can contain both writers and readers, but not at the same time. The altend queue only contains alternation processes trying to read the channel.

Three operations can be done on a channel struct: channel write, channel read and channel alternation read. All these three operations take a channel struct pointer, a data pointer, and the size of the data as arguments. There are as well two additional internal operations for alternation, namely alternation enabling and disabling of a channel.

A channel read does the following: the channel end queue of the channel is checked. If the queue is non-empty and contains writers, the first channel end is popped from the end queue. The data transfer is completed, and the writer process is rescheduled. The reading process resumes execution. If the end queue is empty or contains readers, a channel end struct is allocated on the stack and initialized. The channel end is inserted at the end of the end queue of the channel, and the reader process yields with the state *Wait*. When the process is resumed, the channel operation is already completed. The API call `CHREAD` calls the channel read operation, and returns whether the write was successful or not.

A channel write does the following: the altend queue is checked before the end queue. If it contains an alternation process, the writer tries to accept the alternation alternative. If it is accepted, the data transfer is completed, and the alternation process is rescheduled. If this fails, the end queue is checked. If it is non-empty and contains readers, the first channel end is popped from the end queue. The data transfer is completed, and the reader process is rescheduled. The writer process resumes execution. If the end queue is empty or contains writers, a channel end struct is allocated on the stack and initialized. The channel end is inserted at the end of the end queue of the channel, and the reader process yields with the state *Wait*. When the process is resumed, the channel operation is already completed. The API call `CHWRITE` calls the channel write operation, and returns whether the write was successful or not.

A channel alternation read consists of multiple stages. First, the alternation process enables the channel for alternation synchronization. The end queue is checked, and if it contains writ-

ers, the channel is marked as available. If empty or contains readers, the alternation process is inserted in the altend queue. The case where the channel is available, the alternation process completes the data transfer with the channel alternation read operation. This pops the first writer from the end queue, the data transfer is completed, and the waiting writer process is rescheduled. Lastly, regardless if the alternation process or a writer process completed the channel operation, the alternation process disables the channel for alternation synchronization. This removes the channel end from the altend queue. The channel alternation read operation is done internally by the run-time system.

Transfer of data is optimized depending on the data size. If the data size is either 1, 2, 4, or 8 bytes, data is transferred as integer assignments. If data size is zero, nothing happens. Every other data size is transferred with a normal `memcpy` call. See Listing 5.26 on how this is done.

Listing 5.26: Channel data transfer

```
1  void channel_datatransfer(void *dst, void *src, size_t size) {
2      switch (size) {
3      case 0:    /* do nothing */                    break;
4      case 1:   *(uint8_t *)dst =  *(uint8_t *)src; break;
5      case 2:  *(uint16_t *)dst = *(uint16_t *)src; break;
6      case 4:  *(uint32_t *)dst = *(uint32_t *)src; break;
7      case 8:  *(uint64_t *)dst = *(uint64_t *)src; break;
8      default: memcpy(dst, src, size); break;
9      }
10 }
```

This implementation only implements an any-to-any channel. Implementing one-to-one, any-to-one, or one-to-any channels proved to be too difficult. There was no possibility of enforcing this during compile-time, as C lacks the language features for this. Run-time checks ere possible, but created too much overhead and were too error-prone to be any useful. So for the sake of code simplicity, only implementing an any-to-any channel was favored.

5.7 Alternation

Alternation is used to wait on multiple alternatives, selecting only one available alternative to execute. Each alternative can be supplemented with a corresponding code execution. Optionally, alternatives are guarded by a boolean condition, which is evaluated at the initialization of the alternation process. Only alternatives for which the guard condition is true is enabled for selection. Alternatives consist of the following: channel reads, timeouts and skip. The alternation process is a synchronous process, meaning the alternating process is waiting until an alternative is available and selected.

Alternation is used together with the switch construct in C. Alternation takes in a set of ordered, alternatives, and returns the number, hereafter called key, corresponding to the alternative that was selected. Keys are incrementally ordered, starting at zero on the first alternative. When an alternative is selected and a key is returned, the operation corresponding to the alternative is already executed and branches to the matching switch case.

Listing 5.27 displays pseudocode of the alternation setup, with three guarded alternatives of the three possible alternatives. Note that the code section for each guard is optional, and can be completely left out.

Listing 5.27: Pseudocode of alternation setup

```
switch (Alternation(
    Guard(condition,  // Guard 0, enabled when `condition` is true
        ChanRead(chan, data, sizeof(data)),
    Guard(true,       // Guard 1, always enabled
        Timeout(time)),
    Guard(false,      // Guard 2, never enabled
        Skip())
)) {
case 0: // Code for Guard 0
case 1: // Code for Guard 1
case 2: // Code for Guard 2
}
```

5.7.1 Alternation Process Stages

The alternation process consists of three stages: activation, selection, and deactivation. It has one set of available alternatives, initialized to empty. Key is initialized to 0.

The first stage, activation, checks if the guards are enabled and the corresponding alternatives are available. First, iterating over every guard and checking the guard condition. If the condition is true, the alternative is activated. The alternative is also given a corresponding key. If the condition is false, the alternative is ignored. The key is incremented for each check regardless if activated or not. This is to match each guards with the case numbers. When an alternative is activated, it is checked if available. If available, it is registered in the set of available alternatives.

The second stage, selection, selects which alternative to synchronize on from the set of available alternatives. After the first stage, one of two cases apply: either some alternatives were already available during activation, or non were. In other words, the set of available alternatives is either non-empty or empty. If non-empty, an alternative is selected and moves on to the third stage. If empty, the alternation process is suspended and waits for an alternative to be available. When the alternation process is resumed, the set of available alternative is now populated by one or more alternatives. An alternative is selected and moves on to the third stage. More details about selection in Section 5.7.4.

In the third stage, deactivation, every alternative is iterated over and deactivated if the guard conditional is true.

5.7.2 Alternatives

The three alternatives, when enabled, operates as follows:

- **Channel Read**: specifies a channel read on a given channel, and stores the message in a given variable. The alternative is available when a sender is on the channel.
- **Timeout**: specifies a timeout for the selection process. If the alternation process has not selected an alternative within the specified time, the timeout guard is selected and the alternating process is resumed.
- **Skip**: is always available. In other words, if no other alternatives are available at the alternation initialization, the skip alternative is automatically selected.

It might be tempting to suggest that a skip alternative makes the alternation process asynchronous, but this is not the case. Whenever the alternation selects the skip alternative as a result of no other available alternatives, it simply synchronizes on the skip alternative.

Alternation can consist of multiple enabled channel read alternatives. Only one enabled timeout and skip alternative will be registered. If multiple timeout alternatives are enabled, the timeout alternative with the smallest expiration time will be used. If multiple skip alternatives are enabled, the first skip alternative will be used.

An alternative is initially deactivated. During the alternation process, if the condition of the guard is true, the alternative is activated. Activating an alternative registers the alternation process' interest in synchronizing with the alternative. Deactivating an alternative unregisters this interest.

An alternative which is not available when activated has the added responsibility of resuming the alternation process when becoming available. It should only resume the alternation process if it is still suspended, as multiple alternatives might become available before the alternation resumes execution.

5.7.3 Alternative Selection

During the selection stage the set of available alternatives is populated with one or more members. The following algorithm chooses an alternative from that set:

1. If set only contains one alternative, choose that alternative and goto 4.
2. If the set contains a skip alternative, remove it from the set.
3. Uniform pesudo-randomly choose one alternative from the remaining set.
4. Complete the associated action with the alternative.
5. Return the key from the chosen alternative.

If the selected alternative is a channel read, the data transfer completes and both processes on each end resume. If the selected alternative is a timeout or skip, no additional actions are made and the alternation process continues.

5.7.4 Fairness and Determinism

The use of a pseudo-random choice of multiple available alternatives makes it fair and non-deterministic. The reasoning for this policy being fair is subtle. If it were not fair, starvation would occur when arbitrating between multiple available alternatives. This is however only possible within a finite amount of time, as the arbitration is uniform pseudo-random. Eventually, all alternatives will be chosen. This is the same design and reasoning the Go programming language uses [see 17, Select statements].

This does make the choice non-deterministic, as it is a pseudo-random choice. The discussion between fairness and determinism is an age old debate within the concurrent system design, and for all practical reasons the alternation construct favors fairness over determinism.

5.7.5 Alternation Implementation

All relevant code for channel implementation is found in the following files `alt.h` and `alt.c`.

Alternation is realized by an alternation struct for the alternation process, and a guard struct for each guarded alternative. Listing 5.28 and 5.29 shows the struct definition for the alternation type and guard type, respectively.

The alternation constructor is called `alternation_init`, and takes a pre-allocated alternation struct as an argument. The alternation struct is allocated on the stack. The constructor initializes the struct members and the guard queue, and saves the running process pointer. This is used to reschedule the alternation process if it suspends itself waiting.

The alternation destructor is called `alternation_cleanup`, and takes an alternation struct as an argument. It iterates over the guard queue, calling the guard destructor for each guard. The available guard array is freed as well, as it is allocated on the heap.

Listing 5.28: Alternation type struct

```
typedef struct {
    int key_count;
    int is_accepted;
    Guard *winner;
    struct {
        int num;
        Guard **guards;
    } ready;
    struct {
        size_t num;
        struct GuardQ Q;
    } guards;
    // skip and time guards
    // only exists one of
    Guard *guard_skip;
    Guard *guard_time;
    // alternation process
    Process *process;
} Alternation;
```

Listing 5.29: Guard type struct

```
typedef struct {
    enum {
        GUARD_SKIP,
        GUARD_TIME,
        GUARD_CHAN
    } type;
    int key;
    Alternation *alternation;
    TAILQ_ENTRY(Guard) Qnode;
    RB_ENTRY(Guard) RBnode;
    // Timer Guard
    uint64_t  usec;
    // Chan Guard
    Chan *chan;
    ChanEnd ch_end;
    struct {
        void *ptr;
        size_t size;
    } data;
    int in_chan;
} Guard;
```

The guard constructor is called `alternation_guardcreate`, and is used for all of the three types of guards. The guard struct is allocated on the heap. It takes a guard type, expiration time in microseconds, channel struct pointer, data pointer, and the size of the data. The API call `CHAN_GUARD` calls the constructor with the `GUARD_CHAN` type and the channel related arguments. Note that the channel end struct is allocated alongside the guard struct. The API call `TIME_GUARD` calls the constructor with the `GUARD_TIME` type and the expiration time. Lastly, the API call `SKIP_GUARD` calls the constructor with only the `GUARD_SKIP` type.

The guard destructor is called `alternation_guardfree`, and frees the guard struct.

The alternation process starts with defining different guards with the following API calls `SKIP_GUARD`, `TIME_GUARD` and `CHAN_GUARD`. Each call evaluates the boolean condition supplied as an argument. If true, calls the corresponding guard constructor and returns the guard struct

pointer. If false, a `NULL` pointer is returned.

The list of guards is stored as a variadic list and sent as arguments to the API call `ALT`. The alternation struct is allocated on the stack, and initialized with the alternation constructor. Then, the variadic list is iterated over, adding each guard to the alternation struct. A `NULL` pointer guard is skipped, but the key value is incremented regardless. After evaluating and adding each guard, the selection procedure is called. Following the alternative selection algorithm in Section 5.7.3, some slight additions are used. The available guards array is allocated as a guard struct pointer array, equal to the number of enabled guards. This is dynamically allocated on the heap. The chosen guard is stored in the `winner` member in the alternation struct. For the uniform pseudo-random function, the library function `rand` is used.

Chapter 6

Examples of Usage

This chapter shows the basics of ProXC and how to use this in a C program. Note that this is a very simple introduction, and only covers the basic.

Prerequisites

To use ProXC in a C program, include the header file `proxc.h` in your C-files,

```
#include <proxc.h>
```

and during linking add the linker flag `-lproxc`. See Appendix A for the API header file `proxc.h`. From here on, ProXC can be used.

Starting and Exiting

Everything starts with the `START` call. This is usually called from `main` function. A function pointer is supplied with the `START` call, which will start as the main process in the ProXC. When the main process returns, ProXC will exit as well, and the `START` call in `main` will return.

Listing 6.1: Hello World in ProXC

```
1 void process(void) {
2     printf("Hello World!");
3 }
4 int main(int argc, char *argv[]) {
5     START(process);
6 }
```

A process, which is not the main process, can exit the library with the call `EXIT`. This will stop the library, and the `START` call in `main` will return as if the main process returned.

Process Creation

All process functions must have the function type signature of

```
void process(void);
```

A new process is created with the `PROC` call, which takes a function pointer to a process function, and zero-or-more function arguments for the process. Arguments can only be of pointer types. Within the new process, arguments are retrieved with the call `ARGN`, supplied with the argument number. Arguments are zero-indexed.

Process Execution

A new process can be executed synchronously, with the call `RUN`. This means the spawning process will wait for the new process to finish executing before resuming.

First, we define a simple process called `printer`, which takes a one argument of type string, and prints out this string.

Listing 6.2: Hello World with multiple processes

```
1  void printer(void) {
2      char *msg = ARGN(0);
3      printf("%s", msg);
4  }
```

Then we let a generic process spawn the `printer` process twice, with a synchronous execution. With two separate calls, the string "Hello World!" is printed out as a result of the process spawning.

Listing 6.3: Hello World with multiple processes

```
1  void process(void) {
2      RUN( PROC(printer, "Hello ") );
3      RUN( PROC(printer, "World!\n") );
4  }
```

There is also asynchronous execution, with the call `GO`. This means the spawning process will not wait for the new process to finish executing, and will resume execution right after spawning if possible. Listing 6.4 will alternate between outputting the asynchronous and synchronous message.

Listing 6.4: Asynchronous and synchronous execution

```
1  void never_stop(void) {
2      char *msg = ARGN(0);
3      for (;;) {
4          printf("%s", msg);
5          YIELD();
6      }
7  }
8  void process(void) {
9      GO( PROC(never_stop, "I'm asynchronous!\n") );
10     RUN( PROC(never_stop, "And I'm synchronous!\n") );
11 }
```

Yielding

Note that the process `never_stop` in Listing 6.4 called `YIELD` in the infinite loop. `YIELD` is used to give back control to the run-time system. ProXC relies on cooperative scheduling, and infinite loops without descheduling points will cause stop the library from regaining control. This is why `never_stop` calls `YIELD`, or only one of the messages would print.

Yielding can also be used by a process to occasionally give back control if during a data-intensive computing over a prolonged time.

Composite processes

It is cumbersome to execute one and one process at the time. Instead, one can define a composite process, which contains multiple processes and their execution order. Defining a sequential execution of processes is used with the call `SEQ`.

Listing 6.5: Sequential composite process execution

```
1 void process(void) {
2     RUN(SEQ(
3             PROC(printer, "Hello "),
4             PROC(printer, "World!\n")
5     ));
6 }
```

There is an equivalent call for parallel execution, `PAR`. Note that the code in Listing 6.6 will always print *Chicken* first, every time. This has to do that ProXC is a single-core implementation, which means `PAR` only simulates parallel execution. The order in which processes are defined in a parallel execution will affect which processes are executed first.

Listing 6.6: Parallel composite process execution

```
1  void process(void) {
2      RUN(SEQ(
3              PROC(printer, "Which came first? "),
4              PAR(
5                  PROC(printer, "Chicken\n"),
6                  PROC(printer, "Egg\n")
7      )));
8  }
```

SEQ and PAR can be nested, making intricate and expressive composite trees possible. Listing 6.7 outputs the sequence of numbers 1, 3, 4, 2, 5, 6.

Listing 6.7: Nested composite process

```
1  void process(void) {
2      RUN(PAR(
3              PAR(
4                  SEQ( PROC(printer, "1\n"), PROC(printer, "2\n") ),
5                  PROC(printer, "3\n")
6              ),
7              SEQ(
8                  PROC(printer, "4\n"),
9                  PAR( PROC(printer, "5\n"), PROC(printer, "6\n") )
10     )));
11 }
```

Channels

Independent processes are not very useful for any meaningful computation. Communication between processes is realized through channels. Channels also serve as synchronization points between processes.

Channels are created with the call CHOPEN, and the data size width of the channel is specified with the call. A channel pointer is returned from the call. This channel pointer can be supplied as arguments when creating composite processes.

Reading and writing on the channel pointer is done through the calls `CHREAD` and `CHWRITE`, respectively. The channel pointer is specified, alongside the data pointer and size.

As the channel is dynamically allocated on the heap, call `CHCLOSE` to close and cleanup the given channel. Note that this is up to the programmer to do, and is not cared for by the library.

In Listing 6.8, two processes are trying to write a char pointer to the same channel, with one process reading once from the same channel. The two processes is executed asynchronously in parallel, with each assigned to either *chicken* or *egg*. Before writing to the channel, they are randomly sleeping between 1 to 500 milliseconds. When a channel read is successful, either *chicken* or *egg* is printed out.

Listing 6.8: Channel communication and synchronization

```
1  static const char * chicken = "chicken", *egg = "egg";
2  void first(void) {
3      Chan *ch = ARGN(0);
4      char *entity = ARGN(1);
5      int msec = 1 + (rand() % 500);
6      SLEEP(MSEC(msec));
7      CHWRITE(ch, &entity, char*);
8  }
9  void process(void) {
10     Chan *ch = CHOPEN(char*);
11     GO(PAR(
12             PROC(first, ch, chicken),
13             PROC(first, ch, egg)
14         )
15     );
16     char *winner = NULL;
17     CHREAD(ch, &winner, char*);
18     printf("Winner is %s\n", winner);
19     CHCLOSE(ch);
20 }
```

Process Suspension

Another API call was used in Listing 6.8, namely `SLEEP`. This allows a process to be suspended for a given amount of time, without blocking the entire library. If you were to use the normal system call `sleep`, the entire program would be suspended, as it suspends the kernel-thread.

The suspension time sent to `SLEEP` has a granularity of microseconds. Three additional convenience macros, `SEC`, `MSEC` and `USEC`, is available to easily convert seconds, milliseconds or microseconds to the granularity of the suspension time.

Alternation

The last construct to introduce is the alternation construct. This allows a process to wait for multiple channel reads at the same time, called events. The events are guarded by a conditional bool, which is only enabled when the condition is true. The alternation construct is by default synchronous, meaning the process alternating waiting for the first available event to be available, and then completing said event. When an event is completed, the alternation construct branches to a corresponding code branch, individual for each event.

The alternation construct is invoked by the call `ALT`, which take one-or-more guarded events. For the channel read, the guarded event is created with a `CHAN_GUARD`. The alternation construct is used together with a switch block, which represents the individual code block for each event.

Listing 6.9: Alternation construct setup with channel read guards

```
1  void process(void) {
2      // ...
3      switch (ALT(
4          // Guard 0
5          CHAN_GUARD(x < y, // enabled when x < y equals true
6              inCh, &signal, int),
7          // Guard 1
8          CHAN_GUARD(1, // always enabled
9              dataCh, &data, long),
10         // Guard 2
11         CHAN_GUARD(0, // never enabled
```

```
12              exitCh, &status, int)
13      ) {
14      case 0: // Code for Guard 0
15      case 1: // Code for Guard 1
16      case 2: // Code for Guard 2
17      }
18  }
```

The alternation process also supports timeout events, created with a `TIME_GUARD`. For a given expiration time, with a granularity of microseconds, the timeout event will become available after expiration. This allows an alternation process to only wait for a given time.

Listing 6.10: Alternation construct with timeout

```
1   void process(void) {
2       // ...
3       switch (ALT(
4           CHAN_GUARD(x < y,
5               inCh, &signal, int),
6           TIME_GUARD(1,
7               MSEC(500))
8       ) {
9       case 0: // Code for channel read guard
10      case 1: // Code for timeout guard
11      }
12  }
```

Lastly, the alternation process supports skip events, created with a `SKIP_GUARD`. This allows the alternation process to select the skip event, if no other events are already available at initialization.

Listing 6.11: Alternation construct with skip

```
1  void process(void) {
2      // ...
3      switch (ALT(
4          CHAN_GUARD(x < y,
5              inCh, &data, int),
6          CHAN_GUARD(1,
7              signalCh, &signal, int),
8          SKIP_GUARD(1)
9      ) {
10     case 0: // Code for inCh channel read
11     case 1: // Code for signalCh channel read
12     case 2: // Code for skip guard
13     }
14 }
```

Chapter 7

Performance

This chapter will try to quantify the performance of ProXC by comparing ProXC programs to equal implementations in occam and Go. For compiling ProXC, occam and Go programs, GCC, the SPoC compiler and the official Go compiler gc will be used, respectively. The native 64-bit version of GCC and gc is used, but SPoC was compiled for 32-byte as it does not support 64-bit.

These benchmarks should not be taken as the de-facto metric of usefulness for the library. Rather, it should be an indication on how to perform against other implementations. What really should matter is the library's ability to provide a good CSP abstraction. Either way for completeness sake, benchmarking is provided.

All code for the different benchmark tests can be found in Appendix C.

7.1 Benchmark Setup

All benchmarks used in this chapter is done on the same machine; an HP® Folio 13 Notebook PC with an Intel® Core™ i5-2467M 1.60GHz processor, 4GiB DDR3 RAM, running Ubuntu GNOME 3.18.5 Linux® 64-bit as operating system.

HP Folio is a registered trademark of Hewlett-Packard Development Company, Intel Core is a registered trademark of Intel Corporation, Linux is the registered trademark of Linus Torvalds in the U.S. and other countries, Ubuntu is a registered trademark of Canonical Ltd.

7.2 Benchmark Tests

The benchmarks aim to quantify performance regarding context switches, channel operations, and some general concurrency based data-intensive computing. Each benchmark timing is presented in real-time, and is the average result of 1000 runs. Hopefully, any timing related interruptions that could affect the results will be averaged out.

ProXC and SPoC are single-core implementations. To make the results comparable, Go was restricted to one core with `runtime.GOMAXPROCS(1)`.

7.2.1 Context Switch Test

Benchmarking context switching times can be very difficult to get an accurate measure, so a more rough approach is used to get a more ballpark figure on the context switch times. This test is based on the test done in Brown [29].

This benchmark tests a for-loop iterating 1 million times. First, the raw instruction speed of the for-loop is timed. A single process is spawned running the for-loop, timing the total execution time.

```
for (uint64_t i = 0; i < 1000000; ++i) { /* no-op */ }
```

The results for each language is shown in Table 7.1.

Language/ Compiler	Time per loop iteration (nanoseconds)
occam / SPoC 1.50	3
Go / gc 1.6.2	1
ProXC / GCC 5.4.0	1

Table 7.1: Instruction speed results

To no surprise, ProXC achieves a running time of 1 nanosecond for each loop iteration. An empty for-loop in C is hard to outperform. Interestingly, Go matches this speed with no problem, still with the added run-time overhead. SPoC should have matched ProXC, as it generates standard C code.

Next is to context switch in each loop iteration instead a no-op.

```
for (uint64_t i = 0; i < 1000000; ++i) { yield(); }
```

Now, x number of processes are executed in parallel, instead of only one, running the for-loop with a yield in each loop iteration. For $x = 2$ and $x = 10$ is timed. The results for each x is shown in Table 7.2

Language/ Compiler	Time per loop iteration $x = 2$ (nanoseconds)	Time per loop iteration $x = 10$ (nanoseconds)
occam / SPoC 1.50	150	638
Go / gc 1.6.2	329	1639
ProXC / GCC 5.4.0	130	664

Table 7.2: Context switch results

The benchmarking results in Table 7.2 is the time for one loop iteration for all x processes, not for one single process. In other words, the time result is calculated by $t/10^6$, not $t/(x \times 10^6)$. For a more intuitive representation of the time results, a ratio comparisons between the languages themselves are shown in Table 7.3. The first column is the ratio between the sequential time in Table 7.1 and $x = 2$, and the second column is the ratio between $x = 10$ and $x = 2$, to 2 decimal points. If the languages had an ideal system with no overhead, the ratios would be 2.00 and 5.00 respectively.

Language/ Compiler	$(x = 2)$/Sequential Ratio, 2 d.p.	$(x = 10)/(x = 2)$ Ratio, 2 d.p.
occam / SPoC 1.50	50.00	4.25
Go / gc 1.6.2	329.00	4.98
ProXC / GCC 5.4.0	130.00	5.10

Table 7.3: Running time ratios

What the results show in Table 7.3 is the significance of introducing context switching does to the sequential code. For ProXC, it was a 65 times increased running time compared to an ideal concurrent system. However, both Go and SPoC does not run ideally either, both getting an increased running time of 165 and 25 times more than ideal. Despite this, all languages has a linear increase in running times, when it comes a linear increase in processes. ProXC does however have a unusual ratio of 5.10, meaning a better increase in running time than ideal. Reasons for this might be the operating system helping with caching memory allocations for stack use.

Some naive calculations is used to get a rough estimate of context switch times. The results

for $x = 2$ in Table 7.2, divided by 2, is subtracted by the sequential results in Table 7.1. This yields the results in Table 7.4.

Language/ Compiler	Rough estimate context switch time (nanoseconds)
occam / SPoC 1.50	72
Go / gc 1.6.2	164
ProXC / GCC 5.4.0	64

Table 7.4: Context switch time estimate

This shows that ProXC manages to context switch faster than both SPoC and Go, which is promising results!

7.2.2 Commstime Test

The commstime test is a pseudo-standard benchmark for testing concurrent channel communication in a concurrent system [30]. Three processes, `prefix`, `delta` and `successor`, are connected in a loop by channels and produces all natural numbers by sending a variable through the loop, incrementing each iteration. A fourth process, `consumer`, consumes a number for each iteration in the loop. For each run, the time results are the average time over 1 million numbers consumed, with a warmup run of 1000 iterations.

Language/ Compiler	Time per commstime loop iteration (nanoseconds)
occam / SPoC 1.50	231
Go / gc 1.6.2	1585
ProXC / GCC 5.4.0	451

Table 7.5: Commstime results

occam/SPoC and ProXC outperforms Go in this test. It seems like there is a bigger overhead around channel communication in Go than in occam/SPoC and ProXC. ProXC shows it can handle communication-heavy systems.

7.2.3 Concurrent Prime Sieve

Lastly, a more diverse benchmark is the concurrent prime sieve. This was popularized as an elegant piece of concurrent code in Go [31]. The equivalent code is implemented in occam and ProXC.

The concurrent sieve is a daisy-chained set of processes, which each represent a prime number. At the start of the chain is a generator, which outputs all natural numbers and sends them through the daisy-chain. When a process first receives a number, this number becomes that process' prime number. All other numbers it recieves will be tested for divisibility against the prime number. If yes, then discard, else send it further down the daisy-chain. At the end of the daisy-chain is the final output.

The sieve is set to calculate the 4000th prime, which runs 500 times. The resulting time is the average over all runs to calculate each prime, in other words the average of the calculated $t/(4000)$.

This benchmark tries to measure more generalized performance regarding many processes, channel operations and concurrency system overhead.

Table 7.2.3 shows the results.

Language/ Compiler	Time per prime (microseconds)
occam / SPoC 1.50	153
Go / gc 1.6.2	1068
ProXC / GCC 5.4.0	633

Table 7.6: Concurrent prime sieve results

SPoC comes out as the winner from this test, having an almost five times faster running time per prime than ProXC. This could mostly be a result of ProXC using stackfull coroutines, while SPoC only simulates coroutines, which preallocates all needed memory at program startup. Go uses around 1 millisecond per prime, which probably is the added overhead of the virtual machine and garbage collector.

Chapter 8

Discussion

This chapter discusses the how the abstractions ProXC delivers compares to the CSP model, and what limitations the current implementation faces. Changes in design and implementation which became apparent too late in the project is also taken up for discussion.

8.1 Adaptation of CSP Abstractions

The conceptual driving force of ProXC is to abstract C programs into the CSP paradigm. There are multiple ways to accomplish this, and none of them are the de-facto answer. In order to "measure" how successful ProXC is, is to see how correct and useful the abstractions ProXC provides are compared to the CSP model.

At the core of the library, ProXC consists of lightweight process which communicate with message passing via channels. Complex and expressive process spawning and execution is expressed with composite processes, and processes can arbitrate on multiple channel reads at the same time with alternation constructs.

How correct CSP programs can you get with these features? The abstractions ProXC provides allows the programmer to structure code into independent processes. These independent processes, if enforced by the programmer, can only communicate through channels. This in itself is the core of CSP paradigm.

Process parallelism is realized through composite processes, which is fully inspired by occam. This way of structuring concurrency through sequential and parallel execution ordering

is equal to that of the follow and parallel operator in CSP, respectively. ProXC also introduces asynchronous execution, which corresponds to the interleaving operator in CSP.

The alternation structure compares to the deterministic choice on multiple channel reads. Guarded alternatives allows the alternative to be enabled or disabled, limiting the deterministic choice depending on internal states. The skip alternative is always available for the deterministic choice, while the timeout alternative becomes available after a given expiration time. This, of course, presumes they are enabled.

Suspension of processes, which is the equivalent of sleeping in ProXC, is not necessarily a direct CSP construct, as the original CSP model ignores time. However, process suspension can be useful in use cases where timing requirements are needed.

ProXC does not however support abstractions such as pipes, subordinate processes, or shared processes. Despite this, the already supported abstractions in ProXC serves as a sufficiently powerful tool kit for creating CPS abstractions in C programs. Seeing that the core abstractions is present in occam and that occam is a well renowned abstraction of CSP, ProXC should in its initial state be more than capable of doing the same job.

8.2 Shortcomings and Limitations

When it comes to correct usage and functionality, ProXC gives the necessary tools to express CSP abstractions in C programs. However, it is not perfect. Some considerable shortcomings and limitations are discussed in this section.

8.2.1 Enforcing Incorrect Usage

ProXC provides a framework for CSP abstractions, which allows a programmer to write their C program in the CSP paradigm. However, ProXC is not omnipotent. It only serves as a potential framework for the programmer to use. Compared to occam and XC, where the framework is embedded in the language, the programmer is enforced to conform to the CSP paradigm. Using ProXC, the programmer can do whatever the C language allows, which most of the time is not conforming to the CSP paradigm. That is why ProXC does not enforce incorrect or bad usage, and it is up to the programmer to enforce proper usage to attain correct CSP abstractions.

8.2.2 Fixed Stack

Each process, as a coroutine, has an individual stack. This stack currently has a fixed size at 8 KiB. Stack overflows is entirely possible with such a small stack size. Causing a stack overflow is undefined behaviour, and in best case segmentation faults the program. This consequently discourages large stack allocations and recursive function calls in processes.

8.2.3 Simple Scheduler Policy

As of now, the scheduler policy for ready processes is a straightforward FIFO queue. Since ProXC does not deal with deadlines and no considerations are taken for blocking calls and multi-core support, a simple FIFO queue policy is sufficient. This could however prove to be insufficient if deadlines and time constraints were to be implemented.

8.2.4 Portability

One major weakness of the current implementation of ProXC is how context switching between coroutines is implemented. Since it is implemented in handwritten assembly, makes ProXC naturaly not portable for architectures where the context switching is not written for. As of writing this, context switching for i386 and x86_64 architectures (32/64-bit) is implemented.

The current solution for circumventing this portability issue is to use the `makecontext` [32] library for non-implemented architectures, which is a portable context switching library for System V-like environments. This library is however 3~4 times slower than the handwritten solution, which is why this library is only used for portability reasons.

8.3 Design and Implementation Improvements

Some design and implementation choices became apparent that they were not ideal during the later stages in this project. In hindsight, if I were to undergo this project again, the different choices presented below would be done differently.

Simplified Composite Tree Representation

Composite processes do not necessarily need to be represented as a node tree. A more clever algorithm could work out the process dependency for the execution ordering. What this means is instead of creating a node tree, a direct linked list between execution ordering of processes could be made. This could be represented for each process as a pointer to the next processes to execute when the current one ends. This would reduce the amount of dynamic allocated memory, as each node member is allocated on the heap.

Extensive use of Heap Allocations

The run-time system allocates a considerable amount of memory on the heap. Some of these allocations, such as the scheduler struct, composite process tree nodes for synchronous trees, and the alternation struct and the corresponding guards, could all be allocated on the stack, as these structs have a fixed lifetime. The scheduler struct would not affect the stack size, as it uses the kernel-thread's stack. The alternation structs and its related allocations could however motivate for a larger default stack for the processes, as this would be directly allocated on their stack. This, among other things, could reduce the total amount of dynamic allocated memory footprint.

Static Declarations of Composite Processes

Static declarations of composite processes, alternation constructs and channels may be statically defined and allocated by the run-time system during compilation. This could however require some use of macro magic and leaking of implementation details, and mostly relies on techniques exploiting the limitations of the C programming language.

Timer Based Suspension

Suspended processes, either by sleeping or alternation sleeping, is currently implemented as a manual check by the scheduler each iteration. This creates unnecessary overhead for the scheduler. A better implementation would be to incorporate POSIX timers, which would asynchronously signal the scheduler when a process has expired.

Unecessary Complex Data Structures

Some data structures used in this implementation proved to later be redundant. For instance, as a result of the FIFO scheduler policy, a simple linked list would be sufficient compared to a double ended queue.

Implementation Programming Language

Lastly, writing the run-time system in C is not obligatory. If this were to be redone, I would probably write it in a more type and memory-safe language, such as C++ or Rust, and create API bindings for C programs.

Chapter 9

Future Work

Some optional features that was planned for the library were not implemented because time proved too short. In this chapter these features are discussed, and how the library can benefit from this.

The features discussed in this chapter are highly relevant for the master thesis, beginning January 2017. The most prominent feature is multi-core support and portability, which will be the main focus for the master thesis.

9.1 Multi-Core support

The most prominent feature that could provide great increase in performance and flexibility would be multi-core support. As Brown [3] argues, taking advantage of the potential parallelism in multicore is the future of concurrency programming.

Multi-core support was one of the features that were planned for at the beginning of this project. But, it was quickly determined the amount of work required for the implementation were too much. This is why a single-core support was chosen.

The design for the multi-core support was however made. In short terms, the library spawns a kernel-thread for each online processor core. Each core runs its own scheduler, with its own set of processes to schedule. Spawned processes are distributed through distributed work stealing among schedulers. All access by the scheduler and run-time system to structures that is subjected to race conditions is guarded by atomic constructs. The multi-core algorithms for the

channel and alternation constructs is a modified version of those described in Brown [3].

9.2 Blocking IO and System Calls

Interfacing blocking calls, such as IO operations or system calls, will cause the entire kernel-thread to block. It is possible to mitigate this by detaching the blocking user-thread to its own kernel-thread, and continue operating on the other kernel-thread. When the blocking user-thread resumes, it is added back to the scheduler.

Implementing should be trivial to some degree, as this involves adding extra logic for detaching and attaching user-threads to schedulers. However, further testing is required to test for correctness.

9.3 Portable Coroutines

Since the coroutine implementation requires handwritten assembly, only support for `i386` and `x86_64` was implemented. Full support for multiple architectures is sought after in libraries, which is why portable coroutines would be desired.

There are two options: either implementing it, or using a third party library for coroutines. Implementing it gives total control of necessary features, but requires extensive work and testing. Using an existing third party coroutine library would remove the need for extensive testing, but could require modifications for appropriate use.

9.4 Replicators

Replicators is used in occam and XC to specify ranges of components for both `SEQ`, `PAR` and `ALT` constructs. This allows the programmer to specify finite ranges of processes in composite processes, and ranges of alternatives in alternation constructs.

```
SEQ i = 0 FOR n
PAR i = 0 FOR n
ALT i = 0 FOR n
```

This is a very expressive and powerful semantic, which is very useful for reusing code. Note that a replicator behaves like a for-loop, specifying a range which the composite process or alternation ranges over to generate each process or alternative.

ProXC does not support replicators, but there is in my opinion no reason not to. Implementing replicators should not be very difficult. An appropriate API must be designed, and further implement the logic where the run-time system iteratively generates the processes and alternatives.

9.5 More Complex Scheduler Policy

Currently, a simple FIFO queue is used as the scheduler policy for ready processes. For this simple framework, this policy is good enough. However, if in the future more complex features such as replicators, multi-core support, block calls and etc. is implemented, a more complex scheduler policy might be necessary.

Another factor is how deterministic do we want the library to be. As of now, determinism is not the main focus. But if this changes, a more complex scheduler policy might be needed.

9.6 Stack Reusability, Flexibility and Safety

Stack usage among processes is the most resource heavy component in the run-time system. A single stack is allocated each time a process is spawned, and deallocated when the process has ended. For many cheap and short lived processes, this procedure of allocating and deallocating stacks each time is wasteful. A smarter system of caching already allocated stacks and reusing them for other processes would better utilize stack allocations.

Additionally, the stacks are fixed size. The majority of processes, stack usage is insignificant. However, for processes that require extensive use of the stack, stack overflow is a real issue. A growable stack is possible by having a buffer zone above the stack. When the run-time system detects the stack pointer of the process has accessed the buffer zone, the stack is reallocated to a much larger stack size. Finding the best initial fixed and resizeable stack size would need experimenting on different values.

Lastly, safety of stack overflow should also be considered important to implement. A stack overflow in the current system has no guarantees to be detected or cause an error. For all practical purposes, it is undefined behaviour. This could be implemented by having a small region above the buffer zone, which was mentioned above. This buffer zone would be marked as a memory protected area. When a process accesses the buffer zone, a stack overflow has occurred. The kernel will generate a segmentation fault, crashing the program. This does require kernel support.

Chapter 10

Conclusion

ProXC is a concurrency library for the C programming language. It allows programmers to use Communicating Sequential Processes (CSP) abstractions in their code, equivalent to what occam and XC achieves. This enables programmers to write concurrent code in C with the safety and expressiveness of CSP, either by wrapping existing code with this library, or using this library from scratch.

This paper has assessed the different approaches for creating a CSP abstraction framework for C, which concluded in implementing a library. This library was named ProXC and implemented by the author. The full feature list for ProXC was decided, and from there, a design and API was created. Based on the design, a single-core implementation was detailed, which was written in C. Example of usage was presented, and a few benchmarks were conducted to compare the framework to other existing languages. The presented benchmarks showed great potential for concurrent throughput.

This work should prove helpful for creating CSP abstraction frameworks in other non-CSP languages. The details regarding user-thread implementation could in itself be very helpful for other coroutine implementations.

Not all planned features made it into the library, mostly because of time shortage. The most prominent feature not implemented was multi-core support. This was a sought after feature for taking advantage of the ever increasing parallelism in multi-core architectures. Other concurrent constructs such as barriers and mobiles could be advantageous for ProXC. Lastly, more support for native context switching on other architectures should be favorable.

10.1 Availability

Source code for ProXC is available at GitHub [33], with an open-source MIT license. Any inquires about code or questions in general can reach the author by mail `edvard.pettersen@gmail.com`.

Appendix A

API header

Listing A.1: API header file for ProXC

```
1  #ifndef PROXC_H__
2  #define PROXC_H__
3
4  #include <stddef.h>
5  #include <stdint.h>
6
7  #define PROXC_NULL  ((void *)-1)
8
9  typedef void (*ProcFxn)(void);
10
11 typedef struct Chan Chan;
12 typedef struct Builder Builder;
13 typedef struct Guard Guard;
14
15 void proxc_start(ProcFxn fxn);
16 void proxc_exit(void);
17
18 void* proxc_argn(size_t n);
19 void  proxc_yield(void);
20
21 void  proxc_sleep(uint64_t usec);
22
```

```
23 Builder* proxc_proc(ProcFxn, ...);
24 Builder* proxc_par(int, ...);
25 Builder* proxc_seq(int, ...);
26
27 int proxc_go(Builder *root);
28 int proxc_run(Builder *root);
29
30 Chan* proxc_chopen(size_t size);
31 void  proxc_chclose(Chan *chan);
32 int   proxc_chwrite(Chan *chan, void *data, size_t size);
33 int   proxc_chread(Chan *chan, void *data, size_t size);
34
35 Guard* proxc_guardchan(int cond, Chan* chan, void *out, size_t size);
36 Guard* proxc_guardtime(int cond, uint64_t usec);
37 Guard* proxc_guardskip(int cond);
38 int    proxc_alt(int, ...);
39
40 #ifndef PROXC_NO_MACRO
41
42 #   define START(fxn)  proxc_start(fxn)
43 #   define EXIT()      proxc_exit()
44
45 #   define ARGN(index)  proxc_argn(index)
46 #   define YIELD()      proxc_yield()
47
48 #   define SEC(sec)     MSEC(1000ULL * (uint64_t)(sec))
49 #   define MSEC(msec)   USEC(1000ULL * (uint64_t)(msec))
50 #   define USEC(usec)   ((uint64_t)(usec))
51 #   define SLEEP(usec)  proxc_sleep((uint64_t)(usec))
52
53 #   define PROC(...)  proxc_proc(__VA_ARGS__, PROXC_NULL)
54 #   define PAR(...)   proxc_par(0, __VA_ARGS__, PROXC_NULL)
55 #   define SEQ(...)   proxc_seq(0, __VA_ARGS__, PROXC_NULL)
56
57 #   define GO(build)   proxc_go(build)
58 #   define RUN(build)  proxc_run(build)
```

```
59
60 #   define CHOPEN(type)   proxc_chopen(sizeof(type))
61 #   define CHCLOSE(chan)  proxc_chclose(chan)
62 #   define CHWRITE(chan, data, type)
63                            proxc_chwrite(chan, data, sizeof(type))
64 #   define CHREAD(chan, data, type)
65                            proxc_chread(chan, data, sizeof(type))
66
67 #   define CHAN_GUARD(cond, ch, out, type)
68                 proxc_guardchan(cond, ch, out, sizeof(type))
69 #   define TIME_GUARD(cond, usec)
70                 proxc_guardtime(cond, usec)
71 #   define SKIP_GUARD(cond)
72                 proxc_guardskip(cond)
73 #   define ALT(...)  proxc_alt(0, __VA_ARGS__, PROXC_NULL)
74
75 #endif /* PROXC_NO_MACRO */
76
77 #endif /* PROXC_H__ */
```

Appendix B

Context Switch Assembly

This appendix contains the assembly code used for context switching implementation.

Listing B.1: Context switch assembly for i386

```
1  ; void context_switch(Context *from, Context *to);
2      .globl context_switch
3      .type context_switch,@function
4  context_switch:
5      movl    0x04(%esp), %edx    ; from = %edx
6  ; Store current snapshot in from-context
7      movl    %ebx,   0x00(%edx) ; %ebx
8      movl    %esi,   0x04(%edx) ; %esi
9      movl    %edi,   0x08(%edx) ; %edi
10     movl    %ebp,   0x0C(%edx) ; %ebp
11     movl    %esp,   0x10(%edx) ; %esp
12
13     movl    (%esp), %eax       ; %eip
14     movl    %eax,   0x14(%edx)
15
16     movl    0x08(%esp), %edx    ; to = %edx
17 ; Load to-context snapshot
18     movl    0x00(%edx), %ebx    ; %ebx
19     movl    0x04(%edx), %esi    ; %esi
20     movl    0x08(%edx), %edi    ; %edi
21     movl    0x0C(%edx), %ebp    ; %ebp
```

```
22      movl    0x10(%edx), %esp    ; %esp
23
24      movl    0x14(%edx), %eax    ; %eip
25      movl    %eax,   (%esp)
26      ret
27  .size context_switch, .-context_switch
```

Listing B.2: Context switch assembly for x86_64

```
1  ; void context_switch(Context *from, Context *to);
2     .globl context_switch
3     .type context_switch,@function
4  context_switch:
5  ; from = %rdi, to = %rsi
6  ; Store current snapshot in from-context
7       movq    %rbx,   0x00(%rdi)  ; %rbx
8       movq    %rsp,   0x08(%rdi)  ; %rsp
9       movq    %rbp,   0x10(%rdi)  ; %rbp
10      movq    %r12,   0x18(%rdi)  ; %r12
11      movq    %r13,   0x20(%rdi)  ; %r13
12      movq    %r14,   0x28(%rdi)  ; %r14
13      movq    %r15,   0x30(%rdi)  ; %r15
14
15      movq    (%rsp), %rax        ; %rip
16      movq    %rax,   0x38(%rdi)
17 ; Load to-context snapshot
18      movq    0x00(%rsi), %rbx    ; %rbx
19      movq    0x08(%rsi), %rsp    ; %rsp
20      movq    0x10(%rsi), %rbp    ; %rbp
21      movq    0x18(%rsi), %r12    ; %r12
22      movq    0x20(%rsi), %r13    ; %r13
23      movq    0x28(%rsi), %r14    ; %r14
24      movq    0x30(%rsi), %r15    ; %r15
25
26      movq    0x38(%rsi), %rax    ; %rip
27      movq    %rax,   (%rsp)
```

```
28      retq
29  .size context_switch, .-context_switch
```

Appendix C

Benchmark Code

Context Switch Test

ProXC

Listing C.1: Context switch code for ProXC

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <time.h>
4  #include <proxc.h>
5  #define X      10
6  #define ITERS  1000000UL
7  #define RUNS   1000
8  void worker(void) {
9      for (uint64_t iter = 0; iter < ITERS; ++iter) {
10         YIELD();
11         //__asm__("");
12     }
13 }
14 void bench(void) {
15     printf("ProXC context bench\n");
16     printf("X: %d\n", X);
17     printf("ITERS: %lu\n", ITERS);
18     printf("RUNS: %d\n", RUNS);
```

```
19      double summa = 0.0;
20      for (int run = 0; run < RUNS; run++) {
21          clock_t start, stop;
22          start = clock();
23          RUN(PAR(
24              PROC(worker),
25              PROC(worker),
26              PROC(worker),
27              PROC(worker),
28              PROC(worker),
29              PROC(worker),
30              PROC(worker),
31              PROC(worker),
32              PROC(worker),
33              PROC(worker)
34          ));
35          stop = clock();
36          double elapsed_ns = (double)(stop - start);
37          elapsed_ns *= 1000.0 * 1000.0 * 1000.0 / CLOCKS_PER_SEC;
38          double loop_time = elapsed_ns / ITERS;
39          summa += loop_time;
40      }
41      printf("average time per loop iter: %fns\n", summa / RUNS);
42  }
43  int main(void) {
44      proxc_start(bench);
45  }
```

Go

Listing C.2: Context switch code for Go

```
1  package main
2  import (
3      "fmt"
```

```
4       "runtime"
5       "sync"
6       "time"
7   )
8   const (
9       X      = 10
10      ITERS  = 1000000
11      RUNS   = 1000
12  )
13  func worker(wg *sync.WaitGroup) {
14      for i := 0; i < ITERS; i++ {
15          runtime.Gosched()
16      }
17      wg.Done()
18  }
19  func main() {
20      runtime.GOMAXPROCS(1)
21      fmt.Println("Go context bench")
22      fmt.Printf("X: %d\n", X)
23      fmt.Printf("ITERS: %d\n", ITERS)
24      fmt.Printf("RUNS: %d\n", RUNS)
25      var summa float64 = 0.0
26      var wg sync.WaitGroup
27      for run := 0; run < RUNS; run++ {
28          wg.Add(X)
29          start := time.Now()
30          for x := 0; x < X; x++ {
31              go worker(&wg)
32          }
33          wg.Wait()
34          elapsed := float64(time.Since(start).Nanoseconds())
35          loop_time := elapsed / ITERS
36          summa += loop_time
37      }
38      fmt.Println("")
39      fmt.Println("Results:")
```

```
40      fmt.Printf("average time per loop iter: %fns\n", summa/RUNS)
41  }
```

occam

Listing C.3: Context switch code for occam

```
1   #INCLUDE "hostio.inc"
2   #USE "hostio.lib"
3   #H #include <time.h>
4   PROC worker(VAL INT iters)
5     INT iter :
6     SEQ iter = 0 FOR iters
7       PAR -- hack to force SPoC a context switch
8         SKIP
9   :
10  PROC bench(CHAN OF SP fs,ts, []INT mem)
11    VAL x     IS 1 :
12    VAL iters IS 1000000 :
13    VAL runs  IS 1000 :
14    INT start, stop, elapsed :
15    REAL64 summa, iter :
16    SEQ
17      #C printf("occam ctx switch\n");
18      #C printf("X: %d\n", $x);
19      #C printf("ITERS: %d\n", $iters);
20      #C printf("RUNS: %d\n", $runs);
21      summa := 0.0(REAL64)
22      SEQ run = 0 FOR runs
23        SEQ
24          #C $start = clock();
25          PAR i = 0 FOR x
26            SEQ iter = 0 FOR iters
27              SKIP
28          #C $stop = clock();
```

```
29          #C $elapsed = (double)($stop - $start) * 1000.0 * 1000.0 * 1000.0↩
                / CLOCKS_PER_SEC;
30          #C $summa += $elapsed / $iters;
31      #C printf("Average: %fns\n", $summa / $runs);
32  :
```

Commstime Test

ProXC

Listing C.4: Commstime code for ProXC

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <time.h>
5  #include <proxc.h>
6  #define WARMUP  1000
7  #define REPEAT  1000000
8  #define RUNS    1000
9  void prefix(void) {
10     Chan *a = ARGN(0), *b = ARGN(1);
11     int value = 0;
12     for(;;) {
13         CHWRITE(a, &value, int);
14         CHREAD(b, &value, int);
15     }
16 }
17 void delta(void) {
18     Chan *a = ARGN(0), *c = ARGN(1), *d = ARGN(2);
19     int value;
20     for (;;) {
21         CHREAD(a, &value, int);
22         CHWRITE(d, &value, int);
23         CHWRITE(c, &value, int);
```

```
24      }
25  }
26  void successor(void) {
27      Chan *b = ARGN(0), *c = ARGN(1);
28      int value;
29      for (;;) {
30          CHREAD(c, &value, int);
31          value++;
32          CHWRITE(b, &value, int);
33      }
34  }
35  void consumer(void) {
36      Chan *d = ARGN(0);
37      clock_t start, stop;
38      double summa = 0.0, elapsed;
39      int x;
40      printf("ProXC commstime\n");
41      printf("WARMUP: %d\n", WARMUP);
42      printf("REPEAT: %d\n", REPEAT);
43      printf("RUNS: %d\n", RUNS);
44      for (int j = 0; j < RUNS; j++) {
45          for (int i = 0; i < WARMUP; i++ ) {
46              CHREAD(d, &x, int);
47          }
48          start = clock();
49          for (int i = 0; i < REPEAT; i++ ) {
50              CHREAD(d, &x, int);
51          }
52          stop = clock();
53          elapsed = (double)(stop - start);
54          elapsed *= 1000.0 * 1000.0 * 1000.0 / CLOCKS_PER_SEC;
55          summa += elapsed / REPEAT;
56      }
57      printf("average: %fns\n", summa / RUNS);
58  }
59  void commstime(void) {
```

```
60      Chan *a = CHOPEN(int);
61      Chan *b = CHOPEN(int);
62      Chan *c = CHOPEN(int);
63      Chan *d = CHOPEN(int);
64      GO(PAR(
65              PROC(prefix, a, b),
66              PROC(delta, a, c, d),
67              PROC(successor, b, c)
68          )
69      );
70      RUN( PROC(consumer, d) );
71  }
72  int main(void) {
73      proxc_start(commstime);
74      return 0;
75  }
```

Go

Listing C.5: Commstime code for Go

```
1   package main
2   import (
3       "fmt"
4       "runtime"
5       "time"
6   )
7   const (
8       WARMUP = 1000
9       REPEAT = 1000000
10      RUNS   = 100
11  )
12  func prefix(a, b chan int) {
13      value := 0
14      for {
```

```
15          a <- value
16          value = <-b
17      }
18  }
19  func delta(a, c, d chan int) {
20      var value int
21      for {
22          value = <-a
23          d <- value
24          c <- value
25      }
26  }
27  func successor(b, c chan int) {
28      var value int
29      for {
30          value = <-c
31          value++
32          b <- value
33      }
34  }
35  func main() {
36      runtime.GOMAXPROCS(1)
37      fmt.Println("Go commstime")
38      fmt.Printf("WARMUP: %d\n", WARMUP)
39      fmt.Printf("REPEAT: %d\n", REPEAT)
40      fmt.Printf("RUNS: %d\n", RUNS)
41      a := make(chan int)
42      b := make(chan int)
43      c := make(chan int)
44      d := make(chan int)
45      go prefix(a, b)
46      go delta(a, c, d)
47      go successor(b, c)
48      summa := 0.0
49      for run := 0; run < RUNS; run++ {
50          for i := 0; i < WARMUP; i++ {
```

```
51              <-d
52          }
53          start := time.Now()
54          for i := 0; i < REPEAT; i++ {
55              <-d
56          }
57          elapsed := float64(time.Since(start).Nanoseconds())
58          summa += elapsed / REPEAT
59      }
60      fmt.Printf("Average: %fns\n", summa/RUNS)
61  }
```

occam

Listing C.6: Commstime code for occam

```
1  #INCLUDE "hostio.inc"
2  #USE "hostio.lib"
3  #H #include <time.h>
4  PROC prefix(CHAN OF INT a, b)
5    INT value :
6    SEQ
7      value := 0
8      WHILE TRUE
9        SEQ
10         a ! value
11         b ? value
12 :
13 PROC delta(CHAN OF INT a, c, d)
14   INT value :
15   WHILE TRUE
16     SEQ
17       a ? value
18       d ! value
19       c ! value
```

```
:
PROC successor(CHAN OF INT b, c)
  INT value :
  WHILE TRUE
    SEQ
      c ? value
      b ! value + 1
:
PROC commstime(CHAN OF SP fs,ts, []INT mem)
  CHAN OF INT a, b, c, d :
  VAL warmup IS 1000 :
  VAL repeat IS 1000000 :
  VAL runs   IS 1000 :
  INT start, stop :
  REAL64 summa, elapsed :
  SEQ
    #C printf("occam commstime\n");
    #C printf("WARMUP: %d\n", $warmup);
    #C printf("REPEAT: %d\n", $repeat);
    #C printf("RUNS: %d\n", $runs);
    PAR
      prefix(a, b)
      delta(a, c, d)
      successor(b, c)
      INT x :
      SEQ
        SEQ run = 0 FOR runs
          SEQ
            SEQ i = 0 FOR warmup
              d ? x
            #C $start = clock();
            SEQ i = 0 FOR repeat
              d ? x
            #C $stop = clock();
            #C $elapsed = (double)($stop - $start);
            #C $elapsed *= 1000.0 * 1000.0 * 1000.0 / CLOCKS_PER_SEC;
```

```
56              #C $summa += $elapsed / $repeat;
57          #C printf("Average: %f\n", $summa / $runs);
58          so.exit(fs,ts,sps.success)
59  :
```

Concurrent Prime Sieve Test

ProXC

Listing C.7: Concurrent prime sieve code for ProXC

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <proxc.h>
4  void generate(void) {
5      Chan *chlong = ARGN(0);
6      long i = 2;
7      for (;;) {
8          CHWRITE(chlong, &i, long);
9          i++;
10     }
11 }
12 void filter(void) {
13     Chan *in_chlong = ARGN(0);
14     Chan *out_chlong = ARGN(1);
15     long prime = (long)(long *)ARGN(2);
16     long i;
17     for (;;) {
18         CHREAD(in_chlong, &i, long);
19         if (i % prime != 0) {
20             CHWRITE(out_chlong, &i, long);
21         }
22     }
23 }
24 #define PRIME 4000
```

```
25 static Chan *chs[PRIME+1];
26 void sieve(void) {
27     chs[0] = CHOPEN(long);
28     Chan *chlong = chs[0];
29     long prime = 0;
30     long start, stop;
31     double elapsed;
32     start = clock();
33     GO(PROC(generate, chlong));
34     for (long i = 0; i < PRIME; i++) {
35         CHREAD(chlong, &prime, long);
36         chs[i+1] = CHOPEN(long);
37         Chan *new_chlong = chs[i+1];
38         GO(PROC(filter, chlong, new_chlong, (void *)prime));
39         chlong = new_chlong;
40     }
41     stop = clock();
42     elapsed = (double)(stop - start);
43     elapsed *= 1000.0 * 1000.0 / CLOCKS_PER_SEC;
44     printf("%.2f\n", elapsed / PRIME);
45 }
46 int main(void) {
47     proxc_start(sieve);
48 }
```

Go

Listing C.8: Concurrent prime sieve code for Go

```
1 package main
2 import (
3     "fmt"
4     "runtime"
5     "time"
6 )
```

```
7  func Generate(ch chan<- int) {
8      for i := 2; ; i++ {
9          ch <- i
10     }
11 }
12 func Filter(in <-chan int, out chan<- int, prime int) {
13     for {
14         i := <-in // Receive value from 'in'.
15         if i%prime != 0 {
16             out <- i // Send 'i' to 'out'.
17         }
18     }
19 }
20 const (
21     PRIME = 4000
22 )
23 func main() {
24     runtime.GOMAXPROCS(1)
25     ch := make(chan int)
26     var prime int
27     start := time.Now()
28     go Generate(ch)
29     for i := 0; i < PRIME; i++ {
30         prime = <-ch
31         ch1 := make(chan int)
32         go Filter(ch, ch1, prime)
33         ch = ch1
34     }
35     elapsed := float64(time.Since(start).Nanoseconds())
36     fmt.Printf("%.2f\n", elapsed/1000.0/PRIME)
37 }
```

occam

Listing C.9: Concurrent prime sieve code for occam

```
#INCLUDE "hostio.inc"
#USE "hostio.lib"
#H #include <time.h>
PROC generate(CHAN OF INT ch)
  INT i :
  SEQ
    i := 2
    WHILE TRUE
      SEQ
        ch ! i
        i := i + 1
:
PROC filter(CHAN OF INT in, out)
  INT i, prime :
  SEQ
    in ? prime
    WHILE TRUE
      SEQ
        in ? i
        IF
          NOT ((i \ prime) = 0)
            out ! i
          TRUE
            SKIP
:
PROC bench(CHAN OF SP fs,ts, []INT mem)
  VAL numPrime IS 10000 :
  [numPrime]CHAN OF INT chs :
  INT prime :
  INT start, stop :
  REAL64 elapsed :
  SEQ
    #C $start = clock();
```

```
34      PAR
35        generate(chs[0])
36        PAR i = 0 FOR numPrime-1
37          filter(chs[i], chs[i+1])
38        SEQ
39          chs[numPrime-1] ? prime
40          #C $stop = clock();
41          #C $elapsed = (double)($stop - $start) * 1000.0 * 1000.0 / ↩
                CLOCKS_PER_SEC;
42          #C printf("%d: %d\n", $numPrime, $prime);
43          #C printf("%.2f\n", $elapsed / $numPrime);
44          so.exit(fs,ts,sps.success)
45  :
```

Appendix D

Acronyms

ABI	Application Binary Interface
API	Application Programming Interface
CSP	Communicating Sequential Processes
FIFO	First In, First Out
KRoC	Kent Retargetable occam Compiler
OS	Operating System
SPoC	Southampton's Portable occam Compiler
TSD	Thread-Local Storage

Bibliography

[1] C. A. R. Hoare and Robert Ashenhurst. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL http://delivery.acm.org/10.1145/360000/359585/p666-hoare.pdf?ip=129.241.154.110&id=359585&acc=ACTIVE%20SERVICE&key=CDADA77FFDD8BE08%2E5386D6A7D247483C%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&CFID=835533675&CFTOKEN=35625395&__acm__=1473328188_f71b4c32a0689f575f8a07fa974ecf0c.

[2] Peter H. Welch, Neil C. C. Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Integrating and Extending JCSP. In *CPA*, volume 65, pages 349–370, 2007.

[3] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, jul 2007. ISBN 978-1586037673.

[4] Mark Wykes, Mark Debbage, Sean Hill, and Denis Nicole. Southampton's Portable occam Compiler (SPoC). In *Progress in Transputer and occam Research: WoTUG-17: Proceedings of the 17th World occam and Transputer User Group Technical Meeting, 10th-13st April 1994, Bristol, UK*, volume 38, page 40. IOS Press, 1994. ISBN 9051991630.

[5] 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. ISBN 9051992610.

[6] J. Moores. CCSP - A Portable CSP-based Run-time System Supporting C and occam. *Archi-*

tectures, Languages and Techniques for Concurrent Systems, 57:147–168, 1999. ISSN 1383-7575.

[7] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Pearson Education, 2006. ISBN 032131283X.

[8] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[9] John Ousterhout. Why Threads are a Bad Idea (for Most Purposes). In *Presentation Given at the 1996 USENIX Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996.

[10] Dennis M. Ritchie. The Limbo Programming Language. *Inferno Programmer's Manual*, 2, 1997.

[11] Per B. Hansen. *Joyce — A Programming Language for Distributed Systems*, pages 464–492. Springer, 1987.

[12] Per B. Hansen. *SuperPascal: A Publication Language for Parallel Scientific Computing*, pages 495–524. Springer, 1994.

[13] David May. Occam. *ACM Sigplan Notices*, 18(4):69–79, 1983. ISSN 0362-1340.

[14] Iann M. Barron. The Transputer. *The Microprocessor and its Application*, pages 343–57, 1978.

[15] Douglas Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009. ISBN 1907361030.

[16] Ivo Balbaert. *The Way to Go: A Thorough Introduction to the Go Programming Language*. IUniverse, 2012. ISBN 1469769166.

[17] Go Team. The Go Programming Language Specification. Report, Technical Report `https://golang.org/ref/spec`, Google Inc, 2009.

[18] TIOBE Index. `http://www.tiobe.com/tiobe-index/go/`. Accessed: 2016-11-27.

[19] John M. Bjørndalen, Brian Vinter, and Otto J. Anshus. PyCSP - Communicating Sequential Processes for Python. In *CPA*, pages 229–248, 2007.

[20] Alex A. Lehmberg and Martin N. Olsen. An Introduction to CSP.NET. In *CPA*, pages 13–30, 2006.

[21] Neil C. C. Brown and Peter H. Welch. An Introduction to the Kent C++ CSP Library. In J. F. Broenink and G. H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, sep 2003. ISBN 978-1-58603-381-1.

[22] C++CSP2 Documentation. `https://www.cs.kent.ac.uk/projects/ofa/c++csp/doc/index.html`. Accessed: 2016-11-27.

[23] JCSP Documentation. `https://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/`. Accessed: 2016-11-27.

[24] Brian W Kernighan and Dennis M Ritchie. The C Programming Language. 2006.

[25] queue(3) - FreeBSD man page. `https://www.freebsd.org/cgi/man.cgi?query=queue&sektion=3`, . Accessed: 2016-12-08.

[26] tree(3) - FreeBSD man page. `https://www.freebsd.org/cgi/man.cgi?query=tree&sektion=3`, . Accessed: 2016-12-08.

[27] Simon Tatham. Coroutines in C. `http://www.linuxhowtos.org/C_C++/coroutines.pdf`, 2000. Accessed: 2016-12-08.

[28] Sergio Antoy. Lazy Evaluation in Logic. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 371–382. Springer, 1991.

[29] Neil Brown. Rain: A New Concurrent Process-Oriented Programming Language. In *CPA*, pages 237–251, 2006.

[30] MA Roger. A Reconfigurable Host lnterconnection Scheme for Occam-based Field Programmable Gate Arrays. In *Communicating Process Architectures 2001: WoTUG-24: Proceedings of the 24th World Occam and Transputer User Group Technical Meeting, 16-19 September 2001, Bristol, United Kingdom*, volume 59, page 179. IOS Press, 2001.

[31] Concurrent Prime Sieve in Go. `https://play.golang.org/p/9U22NfrXeq`. Accessed: 2016-12-14.

[32] makecontext(3) - Linux man page. https://linux.die.net/man/3/makecontext, . Accessed: 2016-12-14.

[33] Edvard S. Pettersen. ProXC - A CSP-Inspired Concurrency Library for C. https://github.com/edvardsp/libproxc, 2016. Accessed: 2016-12-15.