Concurrency in C∀

by

Thierry Delisle

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

C∀ is a modern, non-object-oriented extension of the C programming language. This thesis serves as a definition and an implementation for the concurrency and parallelism C∀ offers. These features are created from scratch due to the lack of concurrency in ISO C. Lightweight threads are introduced into the language. In addition, monitors are introduced as a high-level tool for control-flow based synchronization and mutual-exclusion. The main contributions of this thesis are two-fold: it extends the existing semantics of monitors introduce by [37] to handle monitors in groups and also details the engineering effort needed to introduce these features as core language features. Indeed, these features are added with respect to expectations of C programmers, and integrate with the C∀ type-system and other language features.

Acknowledgements

I would like to thank my supervisor, Professor Peter Buhr, for his guidance through my degree as well as the editing of this document.

I would like to thank Professors Martin Karsten and Gregor Richards, for reading my thesis and providing helpful feedback.

Thanks to Aaron Moss, Rob Schluntz and Andrew Beach for their work on the C∀ project as well as all the discussions which have helped me concretize the ideas in this thesis.

Finally, I acknowledge that this has been possible thanks to the financial help offered by the David R. Cheriton School of Computer Science and the corporate partnership with Huawei Ltd.

Table of Contents

List of Tables

List of Figures

Listings

List of Acronyms

API Application Program Interface. 1, 7, 14, 30, 59

NUMA Non-Uniform Memory Access. 39

RAII Resource Acquisition Is Initialization. 14, 42

Chapter 1
Introduction

This thesis provides a minimal concurrency API that is simple, efficient and can be reused to build higher-level features. The simplest possible concurrency system is a thread and a lock but this low-level approach is hard to master. An easier approach for users is to support higher-level constructs as the basis of concurrency. Indeed, for highly productive concurrent programming, high-level approaches are much more popular [39]. Examples are task based, message passing and implicit threading. The high-level approach and its minimal API are tested in a dialect of C, called C∀. Furthermore, the proposed API doubles as an early definition of the C∀ language and library. This thesis also provides an implementation of the concurrency library for C∀ as well as all the required language features added to the source-to-source translator.

There are actually two problems that need to be solved in the design of concurrency for a programming language: which concurrency and which parallelism tools are available to the programmer. While these two concepts are often combined, they are in fact distinct, requiring different tools [18]. Concurrency tools need to handle mutual exclusion and synchronization, while parallelism tools are about performance, cost and resource utilization.

In the context of this thesis, a **thread** is a fundamental unit of execution that runs a sequence of code, generally on a program stack. Having multiple simultaneous threads gives rise to concurrency and generally requires some kind of locking mechanism to ensure proper execution. Correspondingly, **concurrency** is defined as the concepts and challenges that occur when multiple independent (sharing memory, timing dependencies, etc.) concurrent threads are introduced. Accordingly, **locking** (and by extension locks) are defined as a mechanism that prevents the progress of certain threads in order to avoid problems due to concurrency. Finally, in this thesis **parallelism** is distinct from concurrency and is defined as running multiple threads simultaneously. More precisely, parallelism implies *actual* simultaneous execution as opposed to concurrency which only requires *apparent* simultaneous execution. As such, parallelism is only observable in the differences in performance or, more generally, differences in timing.

Chapter 2

C∀ Overview

The following is a quick introduction to the C∀ language, specifically tailored to the features needed to support concurrency.

C∀ is an extension of ISO-C and therefore supports all of the same paradigms as C. It is a non-object-oriented system-language, meaning most of the major abstractions have either no runtime overhead or can be opted out easily. Like C, the basics of C∀ revolve around structures and routines, which are thin abstractions over machine code. The vast majority of the code produced by the C∀ translator respects memory layouts and calling conventions laid out by C. Interestingly, while C∀ is not an object-oriented language, lacking the concept of a receiver (e.g., `this`), it does have some notion of objects[1], most importantly construction and destruction of objects. Most of the following code examples can be found on the C∀ website [22].

2.1 References

Like C++, C∀ introduces rebind-able references providing multiple dereferencing as an alternative to pointers. In regards to concurrency, the semantic difference between pointers and references are not particularly relevant, but since this document uses mostly references, here is a quick overview of the semantics:

```
int x, *p1 = &x, **p2 = &p1, ***p3 = &p2,
    &r1 = x,     &&r2 = r1,   &&&r3 = r2;
***p3 = 3;                          //change x
r3    = 3;                          //change x, ***r3
**p3  = ...;                        //change p1
*p3   = ...;                        //change p2
int y, z, & ar[3] = {x, y, z};      //initialize array of references
typeof( ar[1]) p;                   //is int, referenced object type
typeof(&ar[1]) q;                   //is int &, reference type
sizeof( ar[1]) == sizeof(int);      //is true, referenced object size
sizeof(&ar[1]) == sizeof(int *);    //is true, reference size
```

The important take away from this code example is that a reference offers a handle to an object, much like a pointer, but which is automatically dereferenced for convenience.

[1] C defines the term objects as : "region of data storage in the execution environment, the contents of which can represent values" [20, 3.15]

2.2 Overloading

Another important feature of C∀ is function overloading as in Java and C++, where routines with the same name are selected based on the number and type of the arguments. As well, C∀ uses the return type as part of the selection criteria, as in Ada [56]. For routines with multiple parameters and returns, the selection is complex.

```
//selection based on type and number of parameters
void f(void);                   //(1)
void f(char);                   //(2)
void f(int, double);            //(3)
f();                            //select (1)
f('a');                         //select (2)
f(3, 5.2);                      //select (3)

//selection based on  type and number of returns
char   f(int);                  //(1)
double f(int);                  //(2)
char   c = f(3);                //select (1)
double d = f(4);                //select (2)
```

This feature is particularly important for concurrency since the runtime system relies on creating different types to represent concurrency objects. Therefore, overloading is necessary to prevent the need for long prefixes and other naming conventions that prevent name clashes. As seen in chapter 3, routine `main` is an example that benefits from overloading.

2.3 Operators

Overloading also extends to operators. The syntax for denoting operator-overloading is to name a routine with the symbol of the operator and question marks where the arguments of the operation appear, e.g.:

```
int ++? (int op);                       //unary prefix increment
int ?++ (int op);                       //unary postfix increment
int ?+? (int op1, int op2);             //binary plus
int ?<=?(int op1, int op2);             //binary less than
int ?=? (int & op1, int op2);           //binary assignment
int ?+=?(int & op1, int op2);           //binary plus-assignment

struct S {int i, j;};
S ?+?(S op1, S op2) {                   //add two structures
    return (S){op1.i + op2.i, op1.j + op2.j};
}
S s1 = {1, 2}, s2 = {2, 3}, s3;
s3 = s1 + s2;                           //compute sum: s3 == {2, 5}
```

While concurrency does not use operator overloading directly, this feature is more important as an introduction for the syntax of constructors.

2.4 Constructors/Destructors

Object lifetime is often a challenge in concurrency. C∀ uses the approach of giving concurrent meaning to object lifetime as a means of synchronization and/or mutual exclusion. Since C∀

relies heavily on the lifetime of objects, constructors and destructors is a core feature required for concurrency and parallelism. C∀ uses the following syntax for constructors and destructors:

```
struct S {
    size_t size;
    int * ia;
};
void ?{}(S & s, int asize) {     //constructor operator
    s.size = asize;              //initialize fields
    s.ia = calloc(size, sizeof(S));
}
void ^?{}(S & s) {               //destructor operator
    free(ia);                    //de-initialization fields
}
int main() {
    S x = {10}, y = {100};       //implicit calls: ?{}(x, 10), ?{}(y, 100)
    ...                          //use x and y
    ^x{};  ^y{};                 //explicit calls to de-initialize
    x{20};  y{200};              //explicit calls to reinitialize
    ...                          //reuse x and y
}                                //implicit calls: ^?{}(y), ^?{}(x)
```

The language guarantees that every object and all their fields are constructed. Like C++, construction of an object is automatically done on allocation and destruction of the object is done on deallocation. Allocation and deallocation can occur on the stack or on the heap.

```
{
    struct S s = {10};   //allocation, call constructor
    ...
}                        //deallocation, call destructor
struct S * s = new();    //allocation, call constructor
...
delete(s);               //deallocation, call destructor
```

Note that like C++, C∀ introduces `new` and `delete`, which behave like `malloc` and `free` in addition to constructing and destructing objects, after calling `malloc` and before calling `free`, respectively.

2.5 Parametric Polymorphism

Routines in C∀ can also be reused for multiple types. This capability is done using the **`forall`** clauses, which allow separately compiled routines to support generic usage over multiple types. For example, the following sum function works for any type that supports construction from 0 and addition:

```
//constraint type, 0 and +
forall(otype T | { void ?{}(T *, zero_t); T ?+?(T, T); })
T sum(T a[ ], size_t size) {
    T total = 0;                 //construct T from 0
    for(size_t i = 0; i < size; i++)
        total = total + a[i];    //select appropriate +
    return total;
}

S sa[5];
int i = sum(sa, 5);              //use S's 0 construction and +
```

Since writing constraints on types can become cumbersome for more constrained functions, C∀ also has the concept of traits. Traits are named collection of constraints that can be used both

instead and in addition to regular constraints:

```
trait summable( otype T ) {
    void ?{}(T *, zero_t);      //constructor from 0 literal
    T ?+?(T, T);                //assortment of additions
    T ?+=?(T *, T);
    T ++?(T *);
    T ?++(T *);
};
forall( otype T | summable(T) ) //use trait
T sum(T a[], size_t size);
```

Note that the type use for assertions can be either an `otype` or a `dtype`. Types declared as `otype` refer to "complete" objects, i.e., objects with a size, a default constructor, a copy constructor, a destructor and an assignment operator. Using `dtype`, on the other hand, has none of these assumptions but is extremely restrictive, it only guarantees the object is addressable.

2.6 with Clause/Statement

Since C∀ lacks the concept of a receiver, certain functions end up needing to repeat variable names often. To remove this inconvenience, C∀ provides the `with` statement, which opens an aggregate scope making its fields directly accessible (like Pascal).

```
struct S { int i, j; };
int mem(S & this) with (this)       //with clause
    i = 1;                          //this->i
    j = 2;                          //this->j
}
int foo() {
    struct S1 { ... } s1;
    struct S2 { ... } s2;
    with (s1)                       //with statement
    {
        //access fields of s1 without qualification
        with (s2)                   //nesting
        {
            //access fields of s1 and s2 without qualification
        }
    }
    with (s1, s2)                   //scopes open in parallel
    {
        //access fields of s1 and s2 without qualification
    }
}
```

For more information on C∀ see [17, 51, 22].

Chapter 3
Concurrency Basics

Before any detailed discussion of the concurrency and parallelism in C∀, it is important to describe the basics of concurrency and how they are expressed in C∀ user code.

3.1 Basics of concurrency

At its core, concurrency is based on having multiple call-stacks and scheduling among threads of execution executing on these stacks. Concurrency without parallelism only requires having multiple call stacks (or contexts) for a single thread of execution.

Execution with a single thread and multiple stacks where the thread is self-scheduling deterministically across the stacks is called coroutining. Execution with a single and multiple stacks but where the thread is scheduled by an oracle (non-deterministic from the thread's perspective) across the stacks is called concurrency.

Therefore, a minimal concurrency system can be achieved by creating coroutines (see Section 3.3), which instead of context-switching among each other, always ask an oracle where to context-switch next. While coroutines can execute on the caller's stack-frame, stack-full coroutines allow full generality and are sufficient as the basis for concurrency. The aforementioned oracle is a scheduler and the whole system now follows a cooperative threading-model (a.k.a., non-preemptive scheduling). The oracle/scheduler can either be a stack-less or stack-full entity and correspondingly require one or two context-switches to run a different coroutine. In any case, a subset of concurrency related challenges start to appear. For the complete set of concurrency challenges to occur, the only feature missing is preemption.

A scheduler introduces order of execution uncertainty, while preemption introduces uncertainty about where context switches occur. Mutual exclusion and synchronization are ways of limiting non-determinism in a concurrent system. Now it is important to understand that uncertainty is desirable; uncertainty can be used by runtime systems to significantly increase performance and is often the basis of giving a user the illusion that tasks are running in parallel. Optimal performance in concurrent applications is often obtained by having as much non-determinism as correctness allows.

3.2 C∀'s Thread Building Blocks

One of the important features that are missing in C is threading[1]. On modern architectures, a lack of threading is unacceptable [52, 53], and therefore modern programming languages must have the proper tools to allow users to write efficient concurrent programs to take advantage of parallelism. As an extension of C, C∀ needs to express these concepts in a way that is as natural as possible to programmers familiar with imperative languages. And being a system-level language means programmers expect to choose precisely which features they need and which cost they are willing to pay.

3.3 Coroutines: A Stepping Stone

While the main focus of this proposal is concurrency and parallelism, it is important to address coroutines, which are actually a significant building block of a concurrency system. **Coroutine**s are generalized routines which have predefined points where execution is suspended and can be resumed at a later time. Therefore, they need to deal with context switches and other context-management operations. This proposal includes coroutines both as an intermediate step for the implementation of threads, and a first-class feature of C∀. Furthermore, many design challenges of threads are at least partially present in designing coroutines, which makes the design effort that much more relevant. The core API of coroutines revolves around two features: independent call-stacks and `suspend`/`resume`.

A good example of a problem made easier with coroutines is generators, e.g., generating the Fibonacci sequence. This problem comes with the challenge of decoupling how a sequence is generated and how it is used. Listing 3.1 shows conventional approaches to writing generators in C. All three of these approach suffer from strong coupling. The left and centre approaches require that the generator have knowledge of how the sequence is used, while the rightmost approach requires holding internal state between calls on behalf of the generator and makes it much harder to handle corner cases like the Fibonacci seed.

Listing 3.1 is an example of a solution to the Fibonacci problem using C∀ coroutines, where the coroutine stack holds sufficient state for the next generation. This solution has the advantage of having very strong decoupling between how the sequence is generated and how it is used. Indeed, this version is as easy to use as the `fibonacci_state` solution, while the implementation is very similar to the `fibonacci_func` example.

Listing 3.2 shows the `Format` coroutine for restructuring text into groups of character blocks of fixed size. The example takes advantage of resuming coroutines in the constructor to simplify the code and highlights the idea that interesting control flow can occur in the constructor.

3.3.1 Construction

One important design challenge for implementing coroutines and threads (shown in section 3.4) is that the runtime system needs to run code after the user-constructor runs to connect the fully constructed object into the system. In the case of coroutines, this challenge is simpler since there

[1]While the C11 standard defines a "threads.h" header, it is minimal and defined as optional. As such, library support for threading is far from widespread. At the time of writing the thesis, neither `gcc` nor `clang` support "threads.h" in their respective standard libraries.

```c
//Using callbacks
void fibonacci_func(
  int n,
  void (*callback)(int)
) {
  int first = 0;
  int second = 1;
  int next, i;
  for(i = 0; i < n; i++)
  {
    if(i <= 1)
      next = i;
    else {
      next = f1 + f2;
      f1 = f2;
      f2 = next;
    }
    callback(next);
  }
}

int main() {
  void print_fib(int n) {
    printf("%d\n", n);
  }

  fibonacci_func(
    10, print_fib
  );



}
```

```c
//Using output array
void fibonacci_array(
  int n,
  int* array
) {
  int f1 = 0; int f2 = 1;
  int next, i;
  for(i = 0; i < n; i++)
  {
    if(i <= 1)
      next = i;
    else {
      next = f1 + f2;
      f1 = f2;
      f2 = next;
    }
    array[i] = next;
  }
}


int main() {
  int a[10];

  fibonacci_func(
    10, a
  );

  for(int i=0;i<10;i++){
    printf("%d\n", a[i]);
  }

}
```

```c
//Using external state
typedef struct {
  int f1, f2;
} Iterator_t;

int fibonacci_state(
  Iterator_t* it
) {
  int f;
  f = it->f1 + it->f2;
  it->f2 = it->f1;
  it->f1 = max(f,1);
  return f;
}




int main() {
  Iterator_t it={0,0};

  for(int i=0;i<10;i++){
    printf("%d\n",
      fibonacci_state(
        &it
      );
    );
  }

}
```

Table 3.1: Different implementations of a Fibonacci sequence generator in C.

is no non-determinism from preemption or scheduling. However, the underlying challenge remains the same for coroutines and threads.

The runtime system needs to create the coroutine's stack and, more importantly, prepare it for the first resumption. The timing of the creation is non-trivial since users expect both to have fully constructed objects once execution enters the coroutine main and to be able to resume the coroutine from the constructor. There are several solutions to this problem but the chosen option effectively forces the design of the coroutine.

Furthermore, C∀ faces an extra challenge as polymorphic routines create invisible thunks when cast to non-polymorphic routines and these thunks have function scope. For example, the following code, while looking benign, can run into undefined behaviour because of thunks:

```c
//async: Runs function asynchronously on another thread
forall(otype T)
extern void async(void (*func)(T*), T* obj);

forall(otype T)
void noop(T*) {}
```

```
coroutine Fibonacci {
    int fn; //used for communication
};

void ?{}(Fibonacci& this) { //constructor
    this.fn = 0;
}

//main automatically called on first resume
void main(Fibonacci& this) with (this) {
    int fn1, fn2;       //retained between resumes
    fn  = 0;
    fn1 = fn;
    suspend(this);      //return to last resume

    fn  = 1;
    fn2 = fn1;
    fn1 = fn;
    suspend(this);      //return to last resume

    for ( ;; ) {
        fn  = fn1 + fn2;
        fn2 = fn1;
        fn1 = fn;
        suspend(this);  //return to last resume
    }
}

int next(Fibonacci& this) {
    resume(this); //transfer to last suspend
    return this.fn;
}

void main() { //regular program main
    Fibonacci f1, f2;
    for ( int i = 1; i <= 10; i += 1 ) {
        sout | next( f1 ) | next( f2 ) | endl;
    }
}
```

Listing 3.1: Implementation of Fibonacci using coroutines

```
void bar() {
    int a;
    async(noop, &a); //start thread running noop with argument a
}
```

The generated C code[2] creates a local thunk to hold type information:

```
extern void async(/* omitted */, void (*func)(void*), void* obj);

void noop(/* omitted */, void* obj){}

void bar(){
    int a;
    void _thunk0(int* _p0){
        /* omitted */
```

[2]Code trimmed down for brevity

```
//format characters into blocks of 4 and groups of 5 blocks per line
coroutine Format {
   char ch;                          //used for communication
   int g, b;                         //global because used in destructor
};

void  ?{}(Format& fmt) {
   resume( fmt );                    //prime (start) coroutine
}

void ^?{}(Format& fmt) with fmt {
   if ( fmt.g != 0 || fmt.b != 0 )
   sout | endl;
}

void main(Format& fmt) with fmt {
   for ( ;; ) {                      //for as many characters
      for(g = 0; g < 5; g++) {       //groups of 5 blocks
         for(b = 0; b < 4; fb++) {   //blocks of 4 characters
            suspend();
            sout | ch;               //print character
         }
         sout | "  ";                //print block separator
      }
      sout | endl;                   //print group separator
   }
}

void prt(Format & fmt, char ch) {
   fmt.ch = ch;
   resume(fmt);
}

int main() {
   Format fmt;
   char ch;
   Eof: for ( ;; ) {                 //read until end of file
      sin | ch;                      //read one character
      if(eof(sin)) break Eof;        //eof ?
      prt(fmt, ch);                  //push character for formatting
   }
}
```

Listing 3.2: Formatting text into lines of 5 blocks of 4 characters.

```
        noop(/* omitted */, _p0);
    }
    /* omitted */
    async(/* omitted */, ((void (*)(void*))(&_thunk0)), (&a));
}
```

The problem in this example is a storage management issue, the function pointer `_thunk0` is only valid until the end of the block, which limits the viable solutions because storing the function pointer for too long causes undefined behaviour; i.e., the stack-based thunk being destroyed before it can be used. This challenge is an extension of challenges that come with second-class routines. Indeed, GCC nested routines also have the limitation that nested routine cannot be passed outside of the declaration scope. The case of coroutines and threads is simply an extension of this problem

to multiple call stacks.

3.3.2 Alternative: Composition

One solution to this challenge is to use composition/containment, where coroutine fields are added to manage the coroutine.

```
struct Fibonacci {
    int fn; //used for communication
    coroutine c; //composition
};

void FibMain(void*) {
    //...
}

void ?{}(Fibonacci& this) {
    this.fn = 0;
    //Call constructor to initialize coroutine
    (this.c){myMain};
}
```

The downside of this approach is that users need to correctly construct the coroutine handle before using it. Like any other objects, the user must carefully choose construction order to prevent usage of objects not yet constructed. However, in the case of coroutines, users must also pass to the coroutine information about the coroutine main, like in the previous example. This opens the door for user errors and requires extra runtime storage to pass at runtime information that can be known statically.

3.3.3 Alternative: Reserved keyword

The next alternative is to use language support to annotate coroutines as follows:

```
coroutine Fibonacci {
    int fn; //used for communication
};
```

The `coroutine` keyword means the compiler can find and inject code where needed. The downside of this approach is that it makes coroutine a special case in the language. Users wanting to extend coroutines or build their own for various reasons can only do so in ways offered by the language. Furthermore, implementing coroutines without language supports also displays the power of the programming language used. While this is ultimately the option used for idiomatic C∀ code, coroutines and threads can still be constructed by users without using the language support. The reserved keywords are only present to improve ease of use for the common cases.

3.3.4 Alternative: Lambda Objects

For coroutines as for threads, many implementations are based on routine pointers or function objects [19, 42, 48, 44]. For example, Boost implements coroutines in terms of four functor object types:

```
asymmetric_coroutine<>::pull_type
asymmetric_coroutine<>::push_type
symmetric_coroutine<>::call_type
symmetric_coroutine<>::yield_type
```

Often, the canonical threading paradigm in languages is based on function pointers, `pthread` being one of the most well-known examples. The main problem of this approach is that the thread usage is limited to a generic handle that must otherwise be wrapped in a custom type. Since the custom type is simple to write in C∀ and solves several issues, added support for routine/lambda based coroutines adds very little.

A variation of this would be to use a simple function pointer in the same way `pthread` does for threads:

```
void foo( coroutine_t cid, void* arg ) {
    int* value = (int*)arg;
    //Coroutine body
}

int main() {
    int value = 0;
    coroutine_t cid = coroutine_create( &foo, (void*)&value );
    coroutine_resume( &cid );
}
```

This semantics is more common for thread interfaces but coroutines work equally well. As discussed in section 3.4, this approach is superseded by static approaches in terms of expressivity.

3.3.5 Alternative: Trait-Based Coroutines

Finally, the underlying approach, which is the one closest to C∀ idioms, is to use trait-based lazy coroutines. This approach defines a coroutine as anything that satisfies the trait `is_coroutine` (as defined below) and is used as a coroutine.

```
trait is_coroutine(dtype T) {
      void main(T& this);
      coroutine_desc* get_coroutine(T& this);
};

forall( dtype T | is_coroutine(T) ) void suspend(T&);
forall( dtype T | is_coroutine(T) ) void resume (T&);
```

This ensures that an object is not a coroutine until `resume` is called on the object. Correspondingly, any object that is passed to `resume` is a coroutine since it must satisfy the `is_coroutine` trait to compile. The advantage of this approach is that users can easily create different types of coroutines, for example, changing the memory layout of a coroutine is trivial when implementing the `get_coroutine` routine. The C∀ keyword `coroutine` simply has the effect of implementing the getter and forward declarations required for users to implement the main routine.

```
coroutine MyCoroutine {
   int someValue;
};
```

==

```
struct MyCoroutine {
   int someValue;
   coroutine_desc __cor;
};

static inline
coroutine_desc* get_coroutine(
   struct MyCoroutine& this
) {
   return &this.__cor;
}

void main(struct MyCoroutine* this);
```

The combination of these two approaches allows users new to coroutining and concurrency to have an easy and concise specification, while more advanced users have tighter control on memory layout and initialization.

3.4 Thread Interface

The basic building blocks of multithreading in C∀ are threads. Both user and kernel threads are supported, where user threads are the concurrency mechanism and kernel threads are the parallel mechanism. User threads offer a flexible and lightweight interface. A thread can be declared using a struct declaration `thread` as follows:

```
thread foo {};
```

As for coroutines, the keyword is a thin wrapper around a C∀ trait:

```
trait is_thread(dtype T) {
      void ^?{}(T & mutex this);
      void main(T & this);
      thread_desc* get_thread(T & this);
};
```

Obviously, for this thread implementation to be useful it must run some user code. Several other threading interfaces use a function-pointer representation as the interface of threads (for example C♯ [27] and Scala [28]). However, this proposal considers that statically tying a `main` routine to a thread supersedes this approach. Since the `main` routine is already a special routine in C∀ (where the program begins), it is a natural extension of the semantics to use overloading to declare mains for different threads (the normal main being the main of the initial thread). As such the `main` routine of a thread can be defined as

```
thread foo {};

void main(foo & this) {
    sout | "Hello World!" | endl;
}
```

In this example, threads of type `foo` start execution in the `void main(foo &)` routine, which prints `"Hello World!"`. While this thesis encourages this approach to enforce strongly typed programming, users may prefer to use the routine-based thread semantics for the sake of simplicity. With the static semantics it is trivial to write a thread type that takes a function pointer as a parameter and executes it on its stack asynchronously.

```
typedef void (*voidFunc)(int);

thread FuncRunner {
    voidFunc func;
    int arg;
};

void ?{}(FuncRunner & this, voidFunc inFunc, int arg) {
    this.func = inFunc;
    this.arg  = arg;
}

void main(FuncRunner & this) {
    //thread starts here and runs the function
    this.func( this.arg );
}
```

```
void hello(/*unused*/ int) {
    sout | "Hello World!" | endl;
}

int main() {
    FuncRunner f = {hello, 42};
    return 0?
}
```

A consequence of the strongly typed approach to main is that memory layout of parameters and return values to/from a thread are now explicitly specified in the API.

Of course, for threads to be useful, it must be possible to start and stop threads and wait for them to complete execution. While using an API such as `fork` and `join` is relatively common in the literature, such an interface is unnecessary. Indeed, the simplest approach is to use RAII principles and have threads `fork` after the constructor has completed and `join` before the destructor runs.

```
thread World;

void main(World & this) {
    sout | "World!" | endl;
}

void main() {
    World w;
    //Thread forks here

    //Printing "Hello " and "World!" are run concurrently
    sout | "Hello " | endl;

    //Implicit join at end of scope
}
```

This semantic has several advantages over explicit semantics: a thread is always started and stopped exactly once, users cannot make any programming errors, and it naturally scales to multiple threads meaning basic synchronization is very simple.

```
thread MyThread {
    //...
};

//main
void main(MyThread& this) {
    //...
}

void foo() {
    MyThread thrds[10];
    //Start 10 threads at the beginning of the scope

    DoStuff();

    //Wait for the 10 threads to finish
}
```

However, one of the drawbacks of this approach is that threads always form a tree where nodes must always outlive their children, i.e., they are always destroyed in the opposite order of construction because of C scoping rules. This restriction is relaxed by using dynamic allocation, so threads can outlive the scope in which they are created, much like dynamically allocating memory

lets objects outlive the scope in which they are created.

```
thread MyThread {
    //...
};

void main(MyThread& this) {
    //...
}

void foo() {
    MyThread* long_lived;
    {
        //Start a thread at the beginning of the scope
        MyThread short_lived;

        //create another thread that will outlive the thread in this scope
        long_lived = new MyThread;

        DoStuff();

        //Wait for the thread short_lived to finish
    }
    DoMoreStuff();

    //Now wait for the long_lived to finish
    delete long_lived;
}
```

Chapter 4
Concurrency

Several tools can be used to solve concurrency challenges. Since many of these challenges appear with the use of mutable shared state, some languages and libraries simply disallow mutable shared state (Erlang [29], Haskell [41], Akka (Scala) [45]). In these paradigms, interaction among concurrent objects relies on message passing [23, 31, 24] or other paradigms closely relate to networking concepts (channels [38, 34] for example). However, in languages that use routine calls as their core abstraction mechanism, these approaches force a clear distinction between concurrent and non-concurrent paradigms (i.e., message passing versus routine calls). This distinction in turn means that, in order to be effective, programmers need to learn two sets of design patterns. While this distinction can be hidden away in library code, effective use of the library still has to take both paradigms into account.

Approaches based on shared memory are more closely related to non-concurrent paradigms since they often rely on basic constructs like routine calls and shared objects. At the lowest level, concurrent paradigms are implemented as atomic operations and locks. Many such mechanisms have been proposed, including semaphores [26] and path expressions [21]. However, for productivity reasons it is desirable to have a higher-level construct be the core concurrency paradigm [39].

An approach that is worth mentioning because it is gaining in popularity is transactional memory [36]. While this approach is even pursued by system languages like C++ [43], the performance and feature set is currently too restrictive to be the main concurrency paradigm for system languages, which is why it was rejected as the core paradigm for concurrency in C$\forall$.

One of the most natural, elegant, and efficient mechanisms for synchronization and communication, especially for shared-memory systems, is the *monitor*. Monitors were first proposed by Brinch Hansen [13] and later described and extended by C.A.R. Hoare [37]. Many programming languages—e.g., Concurrent Pascal [14], Mesa [49], Modula [57], Turing [40], Modula-3 [12], NeWS [33], Emerald [50], μC++ [15] and Java [32]—provide monitors as explicit language constructs. In addition, operating-system kernels and device drivers have a monitor-like structure, although they often use lower-level primitives such as semaphores or locks to simulate monitors. For these reasons, this project proposes monitors as the core concurrency construct.

4.1 Basics

Non-determinism requires concurrent systems to offer support for mutual-exclusion and synchronization. Mutual-exclusion is the concept that only a fixed number of threads can access a critical section at any given time, where a critical section is a group of instructions on an associated portion

of data that requires the restricted access. On the other hand, synchronization enforces relative ordering of execution and synchronization tools provide numerous mechanisms to establish timing relationships among threads.

4.1.1 Mutual-Exclusion

As mentioned above, mutual-exclusion is the guarantee that only a fix number of threads can enter a critical section at once. However, many solutions exist for mutual exclusion, which vary in terms of performance, flexibility and ease of use. Methods range from low-level locks, which are fast and flexible but require significant attention to be correct, to higher-level concurrency techniques, which sacrifice some performance in order to improve ease of use. Ease of use comes by either guaranteeing some problems cannot occur (e.g., being deadlock free) or by offering a more explicit coupling between data and corresponding critical section. For example, the C++ `std::atomic<T>` offers an easy way to express mutual-exclusion on a restricted set of operations (e.g., reading/writing large types atomically). Another challenge with low-level locks is composability. Locks have restricted composability because it takes careful organizing for multiple locks to be used while preventing deadlocks. Easing composability is another feature higher-level mutual-exclusion mechanisms often offer.

4.1.2 Synchronization

As with mutual-exclusion, low-level synchronization primitives often offer good performance and good flexibility at the cost of ease of use. Again, higher-level mechanisms often simplify usage by adding either better coupling between synchronization and data (e.g., message passing) or offering a simpler solution to otherwise involved challenges. As mentioned above, synchronization can be expressed as guaranteeing that event X always happens before Y. Most of the time, synchronization happens within a critical section, where threads must acquire mutual-exclusion in a certain order. However, it may also be desirable to guarantee that event Z does not occur between X and Y. Not satisfying this property is called **barging**. For example, where event X tries to effect event Y but another thread acquires the critical section and emits Z before Y. The classic example is the thread that finishes using a resource and unblocks a thread waiting to use the resource, but the unblocked thread must compete to acquire the resource. Preventing or detecting barging is an involved challenge with low-level locks, which can be made much easier by higher-level constructs. This challenge is often split into two different methods, barging avoidance and barging prevention. Algorithms that use flag variables to detect barging threads are said to be using barging avoidance, while algorithms that baton-pass locks [10] between threads instead of releasing the locks are said to be using barging prevention.

4.2 Monitors

A **monitor** is a set of routines that ensure mutual-exclusion when accessing shared state. More precisely, a monitor is a programming technique that associates mutual-exclusion to routine scopes, as opposed to mutex locks, where mutual-exclusion is defined by lock/release calls independently of any scoping of the calling routine. This strong association eases readability and maintainability, at the cost of flexibility. Note that both monitors and mutex locks, require an abstract handle to

identify them. This concept is generally associated with object-oriented languages like Java [32] or μC++ [16] but does not strictly require OO semantics. The only requirement is the ability to declare a handle to a shared object and a set of routines that act on it:

```
typedef /*some monitor type*/ monitor;
int f(monitor & m);

int main() {
    monitor m;  //Handle m
    f(m);       //Routine using handle
}
```

4.2.1 Call Semantics

The above monitor example displays some of the intrinsic characteristics. First, it is necessary to use pass-by-reference over pass-by-value for monitor routines. This semantics is important, because at their core, monitors are implicit mutual-exclusion objects (locks), and these objects cannot be copied. Therefore, monitors are non-copy-able objects (`dtype`).

Another aspect to consider is when a monitor acquires its mutual exclusion. For example, a monitor may need to be passed through multiple helper routines that do not acquire the monitor mutual-exclusion on entry. Passthrough can occur for generic helper routines (`swap`, `sort`, etc.) or specific helper routines like the following to implement an atomic counter:

```
monitor counter_t { /*...see section 4.2.3...*/ };

void ?{}(counter_t & nomutex this); //constructor
size_t ++?(counter_t & mutex this); //increment

//need for mutex is platform dependent
void ?{}(size_t * this, counter_t & mutex cnt); //conversion
```

This counter is used as follows:

```
//shared counter
counter_t cnt1, cnt2;

//multiple threads access counter
thread 1 : cnt1++; cnt2++;
thread 2 : cnt1++; cnt2++;
thread 3 : cnt1++; cnt2++;
    ...
thread N : cnt1++; cnt2++;
```

Notice how the counter is used without any explicit synchronization and yet supports thread-safe semantics for both reading and writing, which is similar in usage to the C++ template `std::atomic`.

Here, the constructor (`?{}`) uses the `nomutex` keyword to signify that it does not acquire the monitor mutual-exclusion when constructing. This semantics is because an object not yet constructed should never be shared and therefore does not require mutual exclusion. Furthermore, it allows the implementation greater freedom when it initializes the monitor locking. The prefix increment operator uses `mutex` to protect the incrementing process from race conditions. Finally, there is a conversion operator from `counter_t` to `size_t`. This conversion may or may not require the `mutex` keyword depending on whether or not reading a `size_t` is an atomic operation.

For maximum usability, monitors use multiple-acquisition semantics, which means a single thread can acquire the same monitor multiple times without deadlock. For example, listing 4.1

```
monitor printer { ... };
struct tree {
    tree * left, right;
    char * value;
};
void print(printer & mutex p, char * v);

void print(printer & mutex p, tree * t) {
    print(p, t->value);
    print(p, t->left );
    print(p, t->right);
}
```

Listing 4.1: Recursive printing algorithm using multiple-acquisition.

uses recursion and multiple-acquisition to print values inside a binary tree.

Having both `mutex` and `nomutex` keywords can be redundant, depending on the meaning of a routine having neither of these keywords. For example, it is reasonable that it should default to the safest option (`mutex`) when given a routine without qualifiers `void foo(counter_t & this)`, whereas assuming `nomutex` is unsafe and may cause subtle errors. On the other hand, `nomutex` is the "normal" parameter behaviour, it effectively states explicitly that "this routine is not special". Another alternative is making exactly one of these keywords mandatory, which provides the same semantics but without the ambiguity of supporting routines with neither keyword. Mandatory keywords would also have the added benefit of being self-documented but at the cost of extra typing. While there are several benefits to mandatory keywords, they do bring a few challenges. Mandatory keywords in C∀ would imply that the compiler must know without doubt whether or not a parameter is a monitor or not. Since C∀ relies heavily on traits as an abstraction mechanism, the distinction between a type that is a monitor and a type that looks like a monitor can become blurred. For this reason, C∀ only has the `mutex` keyword and uses no keyword to mean `nomutex`.

The next semantic decision is to establish when `mutex` may be used as a type qualifier. Consider the following declarations:

```
int f1(monitor & mutex m);
int f2(const monitor & mutex m);
int f3(monitor ** mutex m);
int f4(monitor * mutex m []);
int f5(graph(monitor *) & mutex m);
```

The problem is to identify which object(s) should be acquired. Furthermore, each object needs to be acquired only once. In the case of simple routines like `f1` and `f2` it is easy to identify an exhaustive list of objects to acquire on entry. Adding indirections (`f3`) still allows the compiler and programmer to identify which object is acquired. However, adding in arrays (`f4`) makes it much harder. Array lengths are not necessarily known in C, and even then, making sure objects are only acquired once becomes none-trivial. This problem can be extended to absurd limits like `f5`, which uses a graph of monitors. To make the issue tractable, this project imposes the requirement that a routine may only acquire one monitor per parameter and it must be the type of the parameter with at most one level of indirection (ignoring potential qualifiers). Also note that while routine `f3` can be supported, meaning that monitor `**m` is acquired, passing an array to this routine would be type-safe and yet result in undefined behaviour because only the first element of the array is acquired. However, this ambiguity is part of the C type-system with respects to arrays. For this

reason, `mutex` is disallowed in the context where arrays may be passed:

```
int f1(monitor & mutex m);     //Okay : recommended case
int f2(monitor * mutex m);     //Not Okay : Could be an array
int f3(monitor mutex m []);   //Not Okay : Array of unknown length
int f4(monitor ** mutex m);    //Not Okay : Could be an array
int f5(monitor * mutex m []); //Not Okay : Array of unknown length
```

Note that not all array functions are actually distinct in the type system. However, even if the code generation could tell the difference, the extra information is still not sufficient to extend meaningfully the monitor call semantic.

Unlike object-oriented monitors, where calling a mutex member *implicitly* acquires mutual-exclusion of the receiver object, C∀ uses an explicit mechanism to specify the object that acquires mutual-exclusion. A consequence of this approach is that it extends naturally to multi-monitor calls.

```
int f(MonitorA & mutex a, MonitorB & mutex b);

MonitorA a;
MonitorB b;
f(a,b);
```

While OO monitors could be extended with a mutex qualifier for multiple-monitor calls, no example of this feature could be found. The capability to acquire multiple locks before entering a critical section is called *bulk-acquiring*. In practice, writing multi-locking routines that do not lead to deadlocks is tricky. Having language support for such a feature is therefore a significant asset for C∀. In the case presented above, C∀ guarantees that the order of acquisition is consistent across calls to different routines using the same monitors as arguments. This consistent ordering means acquiring multiple monitors is safe from deadlock when using bulk-acquiring. However, users can still force the acquiring order. For example, notice which routines use `mutex`/`nomutex` and how this affects acquiring order:

```
void foo(A& mutex a, B& mutex b) { //acquire a & b
    ...
}

void bar(A& mutex a, B& /*nomutex*/ b) { //acquire a
    ... foo(a, b); ... //acquire b
}

void baz(A& /*nomutex*/ a, B& mutex b) { //acquire b
    ... foo(a, b); ... //acquire a
}
```

The multiple-acquisition monitor lock allows a monitor lock to be acquired by both `bar` or `baz` and acquired again in `foo`. In the calls to `bar` and `baz` the monitors are acquired in opposite order.

However, such use leads to lock acquiring order problems. In the example above, the user uses implicit ordering in the case of function `foo` but explicit ordering in the case of `bar` and `baz`. This subtle difference means that calling these routines concurrently may lead to deadlock and is therefore undefined behaviour. As shown [47], solving this problem requires:

1. Dynamically tracking the monitor-call order.
2. Implement rollback semantics.

While the first requirement is already a significant constraint on the system, implementing a general

function call	`mutex` statement

`monitor M {};`
`void foo( M & mutex m1, M & mutex m2 ) {`
`  // critical section`
`}`
 	
`void bar( M & m1, M & m2 ) {`
`  foo( m1, m2 );`
`}`
	`monitor M {};`
	`void bar( M & m1, M & m2 ) {`
	`  mutex(m1, m2) {`
	`  // critical section`
	`  }`
	`}`

Table 4.1: Regular call semantics vs. `mutex` statement

rollback semantics in a C-like language is still prohibitively complex [25]. In C∀, users simply need to be careful when acquiring multiple monitors at the same time or only use bulk-acquiring of all the monitors. While C∀ provides only a partial solution, most systems provide no solution and the C∀ partial solution handles many useful cases.

For example, multiple-acquisition and bulk-acquiring can be used together in interesting ways:

```
monitor bank { ... };

void deposit( bank & mutex b, int deposit );

void transfer( bank & mutex mybank, bank & mutex yourbank, int me2you) {
    deposit( mybank, -me2you );
    deposit( yourbank, me2you );
}
```

This example shows a trivial solution to the bank-account transfer problem [11]. Without multiple-acquisition and bulk-acquiring, the solution to this problem is much more involved and requires careful engineering.

4.2.2 `mutex` statement

The call semantics discussed above have one software engineering issue: only a routine can acquire the mutual-exclusion of a set of monitor. C∀ offers the `mutex` statement to work around the need for unnecessary names, avoiding a major software engineering problem [30]. Table 4.1 shows an example of the `mutex` statement, which introduces a new scope in which the mutual-exclusion of a set of monitor is acquired. Beyond naming, the `mutex` statement has no semantic difference from a routine call with `mutex` parameters.

4.2.3 Data semantics

Once the call semantics are established, the next step is to establish data semantics. Indeed, until now a monitor is used simply as a generic handle but in most cases monitors contain shared data. This data should be intrinsic to the monitor declaration to prevent any accidental use of data without its appropriate protection. For example, here is a complete version of the counter shown in section 4.2.1:

```
monitor counter_t {
    int value;
```

```
};

void ?{}(counter_t & this) {
    this.cnt = 0;
}

int ?++(counter_t & mutex this) {
    return ++this.value;
}

//need for mutex is platform dependent here
void ?{}(int * this, counter_t & mutex cnt) {
    *this = (int)cnt;
}
```

Like threads and coroutines, monitors are defined in terms of traits with some additional language support in the form of the `monitor` keyword. The monitor trait is:

```
trait is_monitor(dtype T) {
    monitor_desc * get_monitor( T & );
    void ^?{}( T & mutex );
};
```

Note that the destructor of a monitor must be a `mutex` routine to prevent deallocation while a thread is accessing the monitor. As with any object, calls to a monitor, using `mutex` or otherwise, is undefined behaviour after the destructor has run.

4.3 Internal Scheduling

In addition to mutual exclusion, the monitors at the core of C∀'s concurrency can also be used to achieve synchronization. With monitors, this capability is generally achieved with internal or external scheduling as in [37]. With **scheduling** loosely defined as deciding which thread acquires the critical section next, **internal scheduling** means making the decision from inside the critical section (i.e., with access to the shared state), while **external scheduling** means making the decision when entering the critical section (i.e., without access to the shared state). Since internal scheduling within a single monitor is mostly a solved problem, this thesis concentrates on extending internal scheduling to multiple monitors. Indeed, like the bulk-acquiring semantics, internal scheduling extends to multiple monitors in a way that is natural to the user but requires additional complexity on the implementation side.

First, here is a simple example of internal scheduling:

```
monitor A {
    condition e;
}

void foo(A& mutex a1, A& mutex a2) {
    ...
    //Wait for cooperation from bar()
    wait(a1.e);
    ...
}

void bar(A& mutex a1, A& mutex a2) {
    //Provide cooperation for foo()
    ...
    //Unblock foo
```

```
    signal(a1.e);
}
```

There are two details to note here. First, `signal` is a delayed operation; it only unblocks the waiting thread when it reaches the end of the critical section. This semantics is needed to respect mutual-exclusion, i.e., the signaller and signalled thread cannot be in the monitor simultaneously. The alternative is to return immediately after the call to `signal`, which is significantly more restrictive. Second, in CV, while it is common to store a `condition` as a field of the monitor, a `condition` variable can be stored/created independently of a monitor. Here routine `foo` waits for the `signal` from `bar` before making further progress, ensuring a basic ordering.

An important aspect of the implementation is that CV does not allow barging, which means that once function `bar` releases the monitor, `foo` is guaranteed to be the next thread to acquire the monitor (unless some other thread waited on the same condition). This guarantee offers the benefit of not having to loop around waits to recheck that a condition is met. The main reason CV offers this guarantee is that users can easily introduce barging if it becomes a necessity but adding barging prevention or barging avoidance is more involved without language support. Supporting barging prevention as well as extending internal scheduling to multiple monitors is the main source of complexity in the design and implementation of CV concurrency.

4.3.1 Internal Scheduling - Multi-Monitor

It is easy to understand the problem of multi-monitor scheduling using a series of pseudo-code examples. Note that for simplicity in the following snippets of pseudo-code, waiting and signalling is done using an implicit condition variable, like Java built-in monitors. Indeed, `wait` statements always use the implicit condition variable as parameters and explicitly name the monitors (A and B) associated with the condition. Note that in CV, condition variables are tied to a *group* of monitors on first use (called branding), which means that using internal scheduling with distinct sets of monitors requires one condition variable per set of monitors. The example below shows the simple case of having two threads (one for each column) and a single monitor A.

thread 1

```
acquire A
    wait A
release A
```

thread 2

```
acquire A
    signal A
release A
```

One thread acquires before waiting (atomically blocking and releasing A) and the other acquires before signalling. It is important to note here that both `wait` and `signal` must be called with the proper monitor(s) already acquired. This semantic is a logical requirement for barging prevention.

A direct extension of the previous example is a bulk-acquiring version:

```
acquire A & B
    wait A & B
release A & B
```

```
acquire A & B
    signal A & B
release A & B
```

This version uses bulk-acquiring (denoted using the & symbol), but the presence of multiple monitors does not add a particularly new meaning. Synchronization happens between the two threads in exactly the same way and order. The only difference is that mutual exclusion covers a group of

monitors. On the implementation side, handling multiple monitors does add a degree of complexity as the next few examples demonstrate.

While deadlock issues can occur when nesting monitors, these issues are only a symptom of the fact that locks, and by extension monitors, are not perfectly composable. For monitors, a well-known deadlock problem is the Nested Monitor Problem [47], which occurs when a `wait` is made by a thread that holds more than one monitor. For example, the following pseudo-code runs into the nested-monitor problem:

```
acquire A
    acquire B
        wait B
    release B
release A
```

```
acquire A
    acquire B
        signal B
    release B
release A
```

The `wait` only releases monitor `B` so the signalling thread cannot acquire monitor `A` to get to the `signal`. Attempting release of all acquired monitors at the `wait` introduces a different set of problems, such as releasing monitor `C`, which has nothing to do with the `signal`.

However, for monitors as for locks, it is possible to write a program using nesting without encountering any problems if nesting is done correctly. For example, the next pseudo-code snippet acquires monitors A then B before waiting, while only acquiring B when signalling, effectively avoiding the Nested Monitor Problem [47].

```
acquire A
    acquire B
        wait B
    release B
release A
```

```
acquire B
    signal B
release B
```

However, this simple refactoring may not be possible, forcing more complex restructuring.

4.3.2 Internal Scheduling - In Depth

A larger example is presented to show complex issues for bulk-acquiring and its implementation options are analyzed. Listing 4.2 shows an example where bulk-acquiring adds a significant layer of complexity to the internal signalling semantics, and listing 4.3 shows the corresponding C∀ code to implement the pseudo-code in listing 4.2. For the purpose of translating the given pseudo-code into C∀-code, any method of introducing a monitor is acceptable, e.g., `mutex` parameters, global variables, pointer parameters, or using locals with the `mutex` statement.

The complexity begins at code sections 4 and 8 in listing 4.2, which are where the existing semantics of internal scheduling needs to be extended for multiple monitors. The root of the problem is that bulk-acquiring is used in a context where one of the monitors is already acquired, which is why it is important to define the behaviour of the previous pseudo-code. When the signaller thread reaches the location where it should "release `A & B`" (listing 4.2 line 16), it must actually transfer ownership of monitor `B` to the waiting thread. This ownership transfer is required in order to prevent barging into `B` by another thread, since both the signalling and signalled threads still need monitor `A`. There are three options:

Waiting thread

```
1  acquire A
2      //Code Section 1
3      acquire A & B
4          //Code Section 2
5          wait A & B
6          //Code Section 3
7      release A & B
8      //Code Section 4
9  release A
```

Signalling thread

```
10 acquire A
11     //Code Section 5
12     acquire A & B
13         //Code Section 6
14         signal A & B
15         //Code Section 7
16     release A & B
17     //Code Section 8
18 release A
```

Listing 4.2: Internal scheduling with bulk-acquiring

```
monitor A a;
monitor B b;
condition c;
```

Waiting thread

```
mutex(a) {
    //Code Section 1
    mutex(a, b) {
        //Code Section 2
        wait(c);
        //Code Section 3
    }
    //Code Section 4
}
```

Signalling thread

```
mutex(a) {
    //Code Section 5
    mutex(a, b) {
        //Code Section 6
        signal(c);
        //Code Section 7
    }
    //Code Section 8
}
```

Listing 4.3: Equivalent C∀ code for listing 4.2

Waiter

```
1  acquire A
2      acquire A & B
3          wait A & B
4      release A & B
5  release A
```

Signaller

```
6  acquire A
7      acquire A & B
8          signal A & B
9      release A & B
10     //Secretly keep B here
11 release A
12 //Wakeup waiter and transfer A & B
```

Listing 4.4: Listing 4.2, with delayed signalling comments

Delaying Signals

The obvious solution to the problem of multi-monitor scheduling is to keep ownership of all locks until the last lock is ready to be transferred. It can be argued that that moment is when the last lock is no longer needed, because this semantics fits most closely to the behaviour of single-monitor scheduling. This solution has the main benefit of transferring ownership of groups of monitors, which simplifies the semantics from multiple objects to a single group of objects,

effectively making the existing single-monitor semantic viable by simply changing monitors to monitor groups. This solution releases the monitors once every monitor in a group can be released. However, since some monitors are never released (e.g., the monitor of a thread), this interpretation means a group might never be released. A more interesting interpretation is to transfer the group until all its monitors are released, which means the group is not passed further and a thread can retain its locks.

However, listing 4.4 shows this solution can become much more complicated depending on what is executed while secretly holding B at line 10, while avoiding the need to transfer ownership of a subset of the condition monitors. Listing 4.5 shows a slightly different example where a third thread is waiting on monitor `A`, using a different condition variable. Because the third thread is signalled when secretly holding `B`, the goal becomes unreachable. Depending on the order of signals (listing 4.5 line 8 and 10) two cases can happen:

Case 1: thread α goes first. In this case, the problem is that monitor `A` needs to be passed to thread β when thread α is done with it.

Case 2: thread β goes first. In this case, the problem is that monitor `B` needs to be retained and passed to thread α along with monitor `A`, which can be done directly or possibly using thread β as an intermediate.

Note that ordering is not determined by a race condition but by whether signalled threads are enqueued in FIFO or FILO order. However, regardless of the answer, users can move line 10 before line 8 and get the reverse effect for listing 4.5.

In both cases, the threads need to be able to distinguish, on a per monitor basis, which ones need to be released and which ones need to be transferred, which means knowing when to release a group becomes complex and inefficient (see next section) and therefore effectively precludes this approach.

Dependency graphs

In listing 4.2, there is a solution that satisfies both barging prevention and mutual exclusion. If ownership of both monitors is transferred to the waiter when the signaller releases `A & B` and then the waiter transfers back ownership of `A` back to the signaller when it releases it, then the problem is solved (`B` is no longer in use at this point). Dynamically finding the correct order is therefore the second possible solution. The problem is effectively resolving a dependency graph of ownership requirements. Here even the simplest of code snippets requires two transfers and has a super-linear complexity. This complexity can be seen in listing 4.6, which is just a direct extension to three monitors, requires at least three ownership transfer and has multiple solutions. Furthermore, the presence of multiple solutions for ownership transfer can cause deadlock problems if a specific solution is not consistently picked; In the same way that multiple lock acquiring order can cause deadlocks.

Given the three threads example in listing 4.5, figure 4.1 shows the corresponding dependency graph that results, where every node is a statement of one of the three threads, and the arrows the dependency of that statement (e.g., $\alpha 1$ must happen before $\alpha 2$). The extra challenge is that this dependency graph is effectively post-mortem, but the runtime system needs to be able to build and

Thread α

```
1  acquire A
2      acquire A & B
3          wait A & B
4      release A & B
5  release A
```

Thread γ

```
6  acquire A
7      acquire A & B
8          signal A & B
9      release A & B
10     signal A
11 release A
```

Thread β

```
12 acquire A
13     wait A
14 release A
```

Listing 4.5: Pseudo-code for the three thread example.

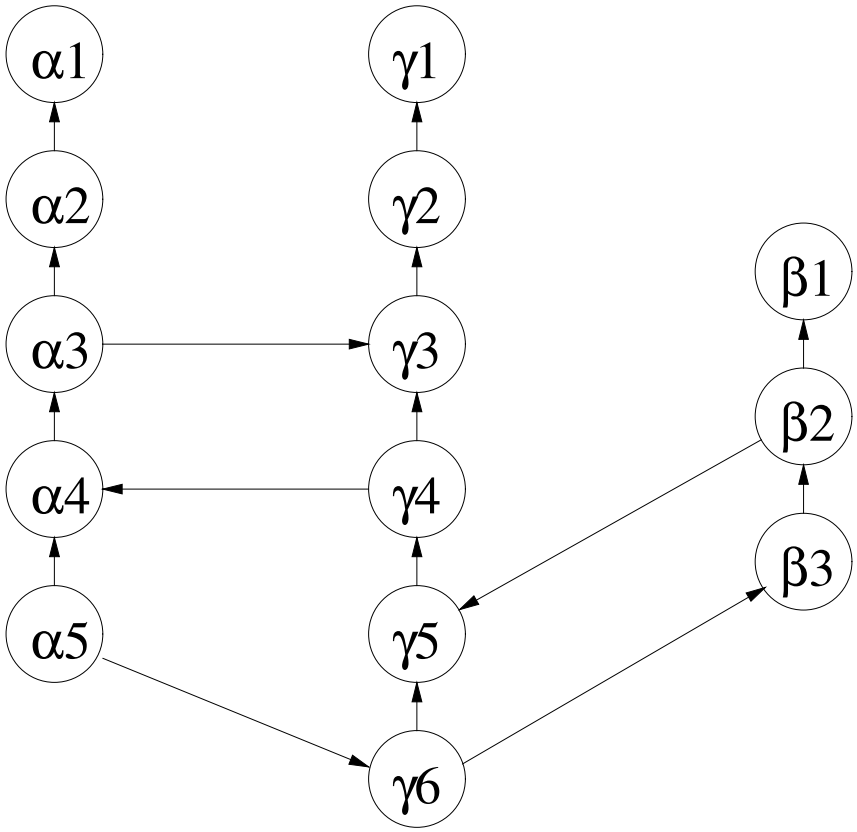

Figure 4.1: Dependency graph of the statements in listing 4.5

```
acquire A
    acquire B
        acquire C
            wait A & B & C
        release C
    release B
release A
```

```
acquire A
    acquire B
        acquire C
            signal A & B & C
        release C
    release B
release A
```

Listing 4.6: Extension to three monitors of listing 4.2

solve these graphs as the dependencies unfold. Resolving dependency graphs being a complex and expensive endeavour, this solution is not the preferred one.

Partial Signalling

Finally, the solution that is chosen for C∀ is to use partial signalling. Again using listing 4.2, the partial signalling solution transfers ownership of monitor `B` at lines 14 to the waiter but does not wake the waiting thread since it is still using monitor `A`. Only when it reaches line 18 does it actually wake up the waiting thread. This solution has the benefit that complexity is encapsulated into only two actions: passing monitors to the next owner when they should be released and conditionally waking threads if all conditions are met. This solution has a much simpler implementation than a dependency graph solving algorithms, which is why it was chosen. Furthermore, after being fully implemented, this solution does not appear to have any significant downsides.

Using partial signalling, listing 4.5 can be solved easily:

- When thread γ reaches line 9 it transfers monitor `B` to thread α and continues to hold monitor `A`.
- When thread γ reaches line 11 it transfers monitor `A` to thread β and wakes it up.
- When thread β reaches line 14 it transfers monitor `A` to thread α and wakes it up.

4.3.3 Signalling: Now or Later

An important note is that, until now, signalling a monitor was a delayed operation. The ownership of the monitor is transferred only when the monitor would have otherwise been released, not at the point of the `signal` statement. However, in some cases, it may be more convenient for users to immediately transfer ownership to the thread that is waiting for cooperation, which is achieved using the `signal_block` routine.

The example in table 4.2 highlights the difference in behaviour. As mentioned, `signal` only transfers ownership once the current critical section exits; this behaviour requires additional synchronization when a two-way handshake is needed. To avoid this explicit synchronization, the `condition` type offers the `signal_block` routine, which handles the two-way handshake as shown in the example. This feature removes the need for a second condition variables and simplifies programming. Like every other monitor semantic, `signal_block` uses barging prevention, which means mutual-exclusion is baton-passed both on the front end and the back end of the call to `signal_block`, meaning no other thread can acquire the monitor either before or after the call.

4.4 External scheduling

An alternative to internal scheduling is external scheduling (see Table 4.3). This method is more constrained and explicit, which helps users reduce the non-deterministic nature of concurrency. Indeed, as the following examples demonstrate, external scheduling allows users to wait for events from other threads without the concern of unrelated events occurring. External scheduling can generally be done either in terms of control flow (e.g., Ada with `accept`, μC++ with `_Accept`) or in terms of data (e.g., Go with channels). Of course, both of these paradigms have their own strengths and weaknesses, but for this project, control-flow semantics was chosen to stay consistent with the rest of the languages semantics. Two challenges specific to C∀ arise when

signal

```
monitor DatingService
{
    //compatibility codes
    enum{ CCodes = 20 };

    int girlPhoneNo
    int boyPhoneNo;
};

condition girls[CCodes];
condition boys [CCodes];
condition exchange;

int girl(int phoneNo, int ccode)
{
    //no compatible boy ?
    if(empty(boys[ccode]))
    {
        //wait for boy
        wait(girls[ccode]);

        //make phone number available
        girlPhoneNo = phoneNo;

        //wake boy from chair
        signal(exchange);
    }
    else
    {
        //make phone number available
        girlPhoneNo = phoneNo;

        //wake boy
        signal(boys[ccode]);

        //sit in chair
        wait(exchange);
    }
    return boyPhoneNo;
}

int boy(int phoneNo, int ccode)
{
    //same as above
    //with boy/girl interchanged
}
```

signal_block

```
monitor DatingService
{
    //compatibility codes
    enum{ CCodes = 20 };

    int girlPhoneNo;
    int boyPhoneNo;
};

condition girls[CCodes];
condition boys [CCodes];
//exchange is not needed

int girl(int phoneNo, int ccode)
{
    //no compatible boy ?
    if(empty(boys[ccode]))
    {
        //wait for boy
        wait(girls[ccode]);

        //make phone number available
        girlPhoneNo = phoneNo;

        //wake boy from chair
        signal(exchange);
    }
    else
    {
        //make phone number available
        girlPhoneNo = phoneNo;

        //wake boy
        signal_block(boys[ccode]);

        //second handshake unnecessary

    }
    return boyPhoneNo;
}

int boy(int phoneNo, int ccode)
{
    //same as above
    //with boy/girl interchanged
}
```

Table 4.2: Dating service example using `signal` and `signal_block`.

Internal Scheduling	External Scheduling	Go

`_Monitor Semaphore {`
`  condition c;`
`  bool inUse;`
`public:`
`  void P() {`
`  if(inUse)`
`  wait(c);`
`  inUse = true;`
`  }`
`  void V() {`
`  inUse = false;`
`  signal(c);`
`  }`
`}`
	`_Monitor Semaphore {`
	`  bool inUse;`
	`public:`
	`  void P() {`
	`  if(inUse)`
	`  _Accept(V);`
	`  inUse = true;`
	`  }`
	`  void V() {`
	`  inUse = false;`
	`  }`
	`}`
		`type MySem struct {`
		`  inUse bool`
		`  c  chan bool`
		`}`
		
		`// acquire`
		`func (s MySem) P() {`
		`  if s.inUse {`
		`  select {`
		`  case <-s.c:`
		`  }`
		`  }`
		`  s.inUse = true`
		`}`
		
		`// release`
		`func (s MySem) V() {`
		`  s.inUse = false`
		
		`  //This actually deadlocks`
		`  //when single thread`
		`  s.c <- false`
		`}`

Table 4.3: Different forms of scheduling.

trying to add external scheduling with loose object definitions and multiple-monitor routines. The previous example shows a simple use `_Accept` versus `wait`/`signal` and its advantages. Note that while other languages often use `accept`/`select` as the core external scheduling keyword, CA uses `waitfor` to prevent name collisions with existing socket APIs.

For the `P` member above using internal scheduling, the call to `wait` only guarantees that `V` is the last routine to access the monitor, allowing a third routine, say `isInUse()`, acquire mutual exclusion several times while routine `P` is waiting. On the other hand, external scheduling guarantees that while routine `P` is waiting, no other routine than `V` can acquire the monitor.

4.4.1 Loose Object Definitions

In μC++, a monitor class declaration includes an exhaustive list of monitor operations. Since CA is not object oriented, monitors become both more difficult to implement and less clear for a user:

```
monitor A {};

void f(A & mutex a);
void g(A & mutex a) {
    waitfor(f); //Obvious which f() to wait for
}

void f(A & mutex a, int); //New different F added in scope
void h(A & mutex a) {
    waitfor(f); //Less obvious which f() to wait for
}
```

Furthermore, external scheduling is an example where implementation constraints become visible from the interface. Here is the pseudo-code for the entering phase of a monitor:

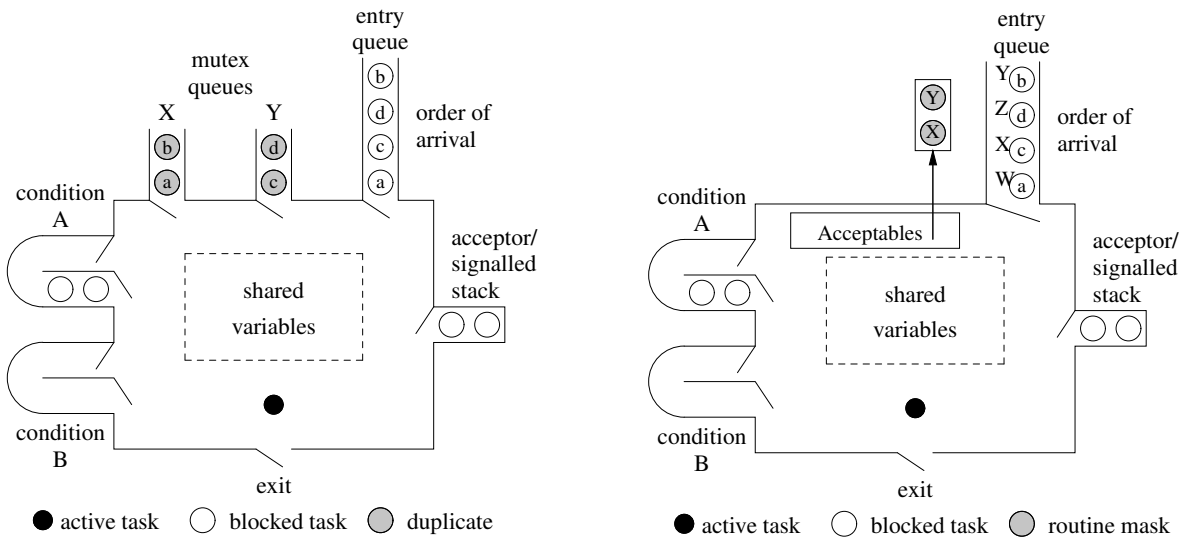

Figure 4.2: External Scheduling Monitor

```
if monitor is free
    enter
elif already own the monitor
    continue
elif monitor accepts me
    enter
else
    block
```

For the first two conditions, it is easy to implement a check that can evaluate the condition in a few instructions. However, a fast check for `monitor accepts me` is much harder to implement depending on the constraints put on the monitors. Indeed, monitors are often expressed as an entry queue and some acceptor queue as in Figure 4.2(a).

There are other alternatives to these pictures, but in the case of the left picture, implementing a fast accept check is relatively easy. Restricted to a fixed number of mutex members, N, the accept check reduces to updating a bitmask when the acceptor queue changes, a check that executes in a single instruction even with a fairly large number (e.g., 128) of mutex members. This approach requires a unique dense ordering of routines with an upper-bound and that ordering must be consistent across translation units. For OO languages these constraints are common, since objects only offer adding member routines consistently across translation units via inheritance. However, in C$\forall$ users can extend objects with mutex routines that are only visible in certain translation unit. This means that establishing a program-wide dense-ordering among mutex routines can only be done in the program linking phase, and still could have issues when using dynamically shared objects.

The alternative is to alter the implementation as in Figure 4.2(b). Here, the mutex routine called is associated with a thread on the entry queue while a list of acceptable routines is kept separate. Generating a mask dynamically means that the storage for the mask information can vary between calls to `waitfor`, allowing for more flexibility and extensions. Storing an array of accepted function pointers replaces the single instruction bitmask comparison with dereferencing a pointer followed by a linear search. Furthermore, supporting nested external scheduling (e.g.,

```
monitor M {};
void foo( M & mutex a ) {}
void bar( M & mutex b ) {
    //Nested in the waitfor(bar, c) call
    waitfor(foo, b);
}
void baz( M & mutex c ) {
    waitfor(bar, c);
}
```

Listing 4.7: Example of nested external scheduling

listing 4.7) may now require additional searches for the `waitfor` statement to check if a routine is already queued.

Note that in the right picture, tasks need to always keep track of the monitors associated with mutex routines, and the routine mask needs to have both a function pointer and a set of monitors, as is discussed in the next section. These details are omitted from the picture for the sake of simplicity.

At this point, a decision must be made between flexibility and performance. Many design decisions in C∀ achieve both flexibility and performance, for example polymorphic routines add significant flexibility but inlining them means the optimizer can easily remove any runtime cost. Here, however, the cost of flexibility cannot be trivially removed. In the end, the most flexible approach has been chosen since it allows users to write programs that would otherwise be hard to write. This decision is based on the assumption that writing fast but inflexible locks is closer to a solved problem than writing locks that are as flexible as external scheduling in C∀.

4.4.2 Multi-Monitor Scheduling

External scheduling, like internal scheduling, becomes significantly more complex when introducing multi-monitor syntax. Even in the simplest possible case, some new semantics needs to be established:

```
monitor M {};

void f(M & mutex a);

void g(M & mutex b, M & mutex c) {
    waitfor(f); //two monitors M => unknown which to pass to f(M & mutex)
}
```

The obvious solution is to specify the correct monitor as follows:

```
monitor M {};

void f(M & mutex a);

void g(M & mutex a, M & mutex b) {
    //wait for call to f with argument b
    waitfor(f, b);
}
```

This syntax is unambiguous. Both locks are acquired and kept by `g`. When routine `f` is called, the lock for monitor `b` is temporarily transferred from `g` to `f` (while `g` still holds lock `a`). This behaviour

can be extended to the multi-monitor `waitfor` statement as follows.

```
monitor M {};

void f(M & mutex a, M & mutex b);

void g(M & mutex a, M & mutex b) {
    //wait for call to f with arguments a and b
    waitfor(f, a, b);
}
```

Note that the set of monitors passed to the `waitfor` statement must be entirely contained in the set of monitors already acquired in the routine. `waitfor` used in any other context is undefined behaviour.

An important behaviour to note is when a set of monitors only match partially:

```
mutex struct A {};

mutex struct B {};

void g(A & mutex a, B & mutex b) {
    waitfor(f, a, b);
}

A a1, a2;
B b;

void foo() {
    g(a1, b); //block on accept
}

void bar() {
    f(a2, b); //fulfill cooperation
}
```

While the equivalent can happen when using internal scheduling, the fact that conditions are specific to a set of monitors means that users have to use two different condition variables. In both cases, partially matching monitor sets does not wakeup the waiting thread. It is also important to note that in the case of external scheduling the order of parameters is irrelevant; `waitfor(f,a,b)` and `waitfor(f,b,a)` are indistinguishable waiting condition.

4.4.3 `waitfor` Semantics

Syntactically, the `waitfor` statement takes a function identifier and a set of monitors. While the set of monitors can be any list of expressions, the function name is more restricted because the compiler validates at compile time the validity of the function type and the parameters used with the `waitfor` statement. It checks that the set of monitors passed in matches the requirements for a function call. Listing 4.8 shows various usages of the waitfor statement and which are acceptable. The choice of the function type is made ignoring any non-`mutex` parameter. One limitation of the current implementation is that it does not handle overloading, but overloading is possible.

Finally, for added flexibility, C∀ supports constructing a complex `waitfor` statement using the `or`, `timeout` and `else`. Indeed, multiple `waitfor` clauses can be chained together using `or`; this chain forms a single statement that uses baton pass to any function that fits one of the function+monitor set passed in. To enable users to tell which accepted function executed, `waitfor`s are followed by a statement (including the null statement `;`) or a compound statement, which is

```
monitor A{};
monitor B{};

void f1( A & mutex );
void f2( A & mutex, B & mutex );
void f3( A & mutex, int );
void f4( A & mutex, int );
void f4( A & mutex, double );

void foo( A & mutex a1, A & mutex a2, B & mutex b1, B & b2 ) {
    A * ap = & a1;
    void (*fp)( A & mutex ) = f1;

    waitfor(f1, a1);      //Correct : 1 monitor case
    waitfor(f2, a1, b1); //Correct : 2 monitor case
    waitfor(f3, a1);      //Correct : non-mutex arguments are ignored
    waitfor(f1, *ap);     //Correct : expression as argument

    waitfor(f1, a1, b1); //Incorrect : Too many mutex arguments
    waitfor(f2, a1);      //Incorrect : Too few mutex arguments
    waitfor(f2, a1, a2); //Incorrect : Mutex arguments don't match
    waitfor(f1, 1);       //Incorrect : 1 not a mutex argument
    waitfor(f9, a1);      //Incorrect : f9 function does not exist
    waitfor(*fp, a1 );    //Incorrect : fp not an identifier
    waitfor(f4, a1);      //Incorrect : f4 ambiguous

    waitfor(f2, a1, b2); //Undefined behaviour : b2 not mutex
}
```

Listing 4.8: Various correct and incorrect uses of the waitfor statement

executed after the clause is triggered. A `waitfor` chain can also be followed by a `timeout`, to signify an upper bound on the wait, or an `else`, to signify that the call should be non-blocking, which checks for a matching function call already arrived and otherwise continues. Any and all of these clauses can be preceded by a `when` condition to dynamically toggle the accept clauses on or off based on some current state. Listing 4.9 demonstrates several complex masks and some incorrect ones.

4.4.4 Waiting For The Destructor

An interesting use for the `waitfor` statement is destructor semantics. Indeed, the `waitfor` statement can accept any `mutex` routine, which includes the destructor (see section 4.2.3). However, with the semantics discussed until now, waiting for the destructor does not make any sense, since using an object after its destructor is called is undefined behaviour. The simplest approach is to disallow `waitfor` on a destructor. However, a more expressive approach is to flip ordering of execution when waiting for the destructor, meaning that waiting for the destructor allows the destructor to run after the current `mutex` routine, similarly to how a condition is signalled. For example, listing 4.10 shows an example of an executor with an infinite loop, which waits for the destructor to break out of this loop. Switching the semantic meaning introduces an idiomatic way to terminate a task and/or wait for its termination via destruction.

```
monitor A{};

void f1( A & mutex );
void f2( A & mutex );

void foo( A & mutex a, bool b, int t ) {
    //Correct : blocking case
    waitfor(f1, a);

    //Correct : block with statement
    waitfor(f1, a) {
        sout | "f1" | endl;
    }

    //Correct : block waiting for f1 or f2
    waitfor(f1, a) {
        sout | "f1" | endl;
    } or waitfor(f2, a) {
        sout | "f2" | endl;
    }

    //Correct : non-blocking case
    waitfor(f1, a); or else;

    //Correct : non-blocking case
    waitfor(f1, a) {
        sout | "blocked" | endl;
    } or else {
        sout | "didn't block" | endl;
    }

    //Correct : block at most 10 seconds
    waitfor(f1, a) {
        sout | "blocked" | endl;
    } or timeout( 10`s) {
        sout | "didn't block" | endl;
    }

    //Correct : block only if b == true
    //if b == false, don't even make the call
    when(b) waitfor(f1, a);

    //Correct : block only if b == true
    //if b == false, make non-blocking call
    waitfor(f1, a); or when(!b) else;

    //Correct : block only of t > 1
    waitfor(f1, a); or when(t > 1) timeout(t); or else;

    //Incorrect : timeout clause is dead code
    waitfor(f1, a); or timeout(t); or else;

    //Incorrect : order must be
    //waitfor [or waitfor... [or timeout] [or else]]
    timeout(t); or waitfor(f1, a); or else;
}
```

Listing 4.9: Various correct and incorrect uses of the or, else, and timeout clause around a waitfor statement

```
monitor Executer {};
struct  Action;

void ^?{}   (Executer & mutex this);
void execute(Executer & mutex this, const Action & );
void run    (Executer & mutex this) {
    while(true) {
           waitfor(execute, this);
        or waitfor(^?{}   , this) {
            break;
        }
    }
}
```

Listing 4.10: Example of an executor which executes action in series until the destructor is called.

Chapter 5
Parallelism

Historically, computer performance was about processor speeds and instruction counts. However, with heat dissipation being a direct consequence of speed increase, parallelism has become the new source for increased performance [52, 53]. In this decade, it is no longer reasonable to create a high-performance application without caring about parallelism. Indeed, parallelism is an important aspect of performance and more specifically throughput and hardware utilization. The lowest-level approach of parallelism is to use kernel-level threads in combination with semantics like `fork`, `join`, etc. However, since these have significant costs and limitations, kernel-level threads are now mostly used as an implementation tool rather than a user oriented one. There are several alternatives to solve these issues that all have strengths and weaknesses. While there are many variations of the presented paradigms, most of these variations do not actually change the guarantees or the semantics, they simply move costs in order to achieve better performance for certain workloads.

5.1 Paradigms

5.1.1 User-Level Threads

A direct improvement on the kernel-level thread approach is to use user-level threads. These threads offer most of the same features that the operating system already provides but can be used on a much larger scale. This approach is the most powerful solution as it allows all the features of multithreading, while removing several of the more expensive costs of kernel threads. The downside is that almost none of the low-level threading problems are hidden; users still have to think about data races, deadlocks and synchronization issues. These issues can be somewhat alleviated by a concurrency toolkit with strong guarantees, but the parallelism toolkit offers very little to reduce complexity in itself.

Examples of languages that support user-level threads are Erlang [29] and μC++ [16].

5.1.2 Fibers : User-Level Threads Without Preemption

A popular variant of user-level threads is what is often referred to as fibers. However, fibers do not present meaningful semantic differences with user-level threads. The significant difference between user-level threads and fibers is the lack of preemption in the latter. Advocates of fibers list their high performance and ease of implementation as major strengths, but the performance difference between user-level threads and fibers is controversial, and the ease of implementation,

while true, is a weak argument in the context of language design. Therefore this proposal largely ignores fibers.

An example of a language that uses fibers is Go [34]

5.1.3 Jobs and Thread Pools

An approach on the opposite end of the spectrum is to base parallelism on thread-pools. Indeed, thread-pools offer limited flexibility but at the benefit of a simpler user interface. In thread-pool based systems, users express parallelism as units of work, called jobs, and a dependency graph (either explicit or implicit) that ties them together. This approach means users need not worry about concurrency but significantly limit the interaction that can occur among jobs. Indeed, any job that blocks also block the underlying worker, which effectively means the CPU utilization, and therefore throughput, suffers noticeably. It can be argued that a solution to this problem is to use more workers than available cores. However, unless the number of jobs and the number of workers are comparable, having a significant number of blocked jobs always results in idles cores.

The gold standard of this implementation is Intel's TBB library [54].

5.1.4 Paradigm Performance

While the choice between the three paradigms listed above may have significant performance implications, it is difficult to pin down the performance implications of choosing a model at the language level. Indeed, in many situations one of these paradigms may show better performance but it all strongly depends on the workload. Having a large amount of mostly independent units of work to execute almost guarantees equivalent performance across paradigms and that the thread-pool-based system has the best efficiency thanks to the lower memory overhead (i.e., no thread stack per job). However, interactions among jobs can easily exacerbate contention. User-level threads allow fine-grain context switching, which results in better resource utilization, but a context switch is more expensive and the extra control means users need to tweak more variables to get the desired performance. Finally, if the units of uninterrupted work are large, enough the paradigm choice is largely amortized by the actual work done.

5.2 The C∀ Kernel : Processors, Clusters and Threads

A cluster is a group of kernel-level threads executed in isolation. User-level threads are scheduled on the kernel-level threads of a given cluster, allowing organization between user-level threads and kernel-level threads. It is important that kernel-level threads belonging to a same clusters have homogeneous settings, otherwise migrating a user-level thread from one kernel-level thread to the other can cause issues. A cluster also offers a pluggable scheduler that can optimize the workload generated by the user-level threads.

Clusters have not been fully implemented in the context of this thesis. Currently C∀ only supports one cluster, the initial one.

5.2.1 Future Work: Machine Setup

While this was not done in the context of this thesis, another important aspect of clusters is affinity. While many common desktop and laptop PCs have homogeneous CPUs, other devices often have more heterogeneous setups. For example, a system using NUMA configurations may benefit from users being able to tie clusters and/or kernel threads to certain CPU cores. OS support for CPU affinity is now common [55, 7, 3, 4, 1], which means it is both possible and desirable for C∀ to offer an abstraction mechanism for portable CPU affinity.

5.2.2 Paradigms

Given these building blocks, it is possible to reproduce all three of the popular paradigms. Indeed, user-level threads is the default paradigm in C∀. However, disabling preemption on the cluster means threads effectively become fibers. Since several clusters with different scheduling policy can coexist in the same application, this allows fibers and user-level threads to coexist in the runtime of an application. Finally, it is possible to build executors for thread pools from user-level threads or fibers, which includes specialized jobs like actors [9].

Chapter 6
Behind the Scenes

There are several challenges specific to C∀ when implementing concurrency. These challenges are a direct result of bulk-acquiring and loose object definitions. These two constraints are the root cause of most design decisions in the implementation. Furthermore, to avoid contention from dynamically allocating memory in a concurrent environment, the internal-scheduling design is (almost) entirely free of mallocs. This approach avoids the chicken and egg problem [58] of having a memory allocator that relies on the threading system and a threading system that relies on the runtime. This extra goal means that memory management is a constant concern in the design of the system.

The main memory concern for concurrency is queues. All blocking operations are made by parking threads onto queues and all queues are designed with intrusive nodes, where each node has pre-allocated link fields for chaining, to avoid the need for memory allocation. Since several concurrency operations can use an unbound amount of memory (depending on bulk-acquiring), statically defining information in the intrusive fields of threads is insufficient.The only way to use a variable amount of memory without requiring memory allocation is to pre-allocate large buffers of memory eagerly and store the information in these buffers. Conveniently, the call stack fits that description and is easy to use, which is why it is used heavily in the implementation of internal scheduling, particularly variable-length arrays. Since stack allocation is based on scopes, the first step of the implementation is to identify the scopes that are available to store the information, and which of these can have a variable-length array. The threads and the condition both have a fixed amount of memory, while `mutex` routines and blocking calls allow for an unbound amount, within the stack size.

Note that since the major contributions of this thesis are extending monitor semantics to bulk-acquiring and loose object definitions, any challenges that are not resulting of these characteristics of C∀ are considered as solved problems and therefore not discussed.

6.1 Mutex Routines

The first step towards the monitor implementation is simple `mutex` routines. In the single monitor case, mutual-exclusion is done using the entry/exit procedure in listing 6.1. The entry/exit procedures do not have to be extended to support multiple monitors. Indeed it is sufficient to enter/leave monitors one-by-one as long as the order is correct to prevent deadlock [35]. In C∀, ordering of monitor acquisition relies on memory ordering. This approach is sufficient because all objects are guaranteed to have distinct non-overlapping memory layouts and mutual-exclusion for a monitor

Entry

```
if monitor is free
    enter
elif already own the monitor
    continue
else
    block
increment recursions
```

Exit

```
decrement recursion
if recursion == 0
    if entry queue not empty
        wake-up thread
```

Listing 6.1: Initial entry and exit routine for monitors

is only defined for its lifetime, meaning that destroying a monitor while it is acquired is undefined behaviour. When a mutex call is made, the concerned monitors are aggregated into a variable-length pointer array and sorted based on pointer values. This array persists for the entire duration of the mutual-exclusion and its ordering reused extensively.

6.1.1 Details: Interaction with polymorphism

Depending on the choice of semantics for when monitor locks are acquired, interaction between monitors and CV's concept of polymorphism can be more complex to support. However, it is shown that entry-point locking solves most of the issues.

First of all, interaction between `otype` polymorphism (see Section 2.5) and monitors is impossible since monitors do not support copying. Therefore, the main question is how to support `dtype` polymorphism. It is important to present the difference between the two acquiring options: callsite-lockings and entry-point locking, i.e., acquiring the monitors before making a mutex routine-call or as the first operation of the mutex routine-call. For example:

Mutex call	callsite-locking pseudo-code	entry-point-locking pseudo-code

`void foo(monitor& mutex a){`
 	
`  //Do Work`
`  //...`
 	
`}`
 	
`void main() {`
`  monitor a;`
 	
`  foo(a);`
 	
`}`
	`foo(& a) {`
	
	`  //Do Work`
	`  //...`
	
	`}`
	
	`main() {`
	`  monitor a;`
	`  acquire(a);`
	`  foo(a);`
	`  release(a);`
	`}`
		`foo(& a) {`
		`  acquire(a);`
		`  //Do Work`
		`  //...`
		`  release(a);`
		`}`
		
		`main() {`
		`  monitor a;`
		
		`  foo(a);`
		
		`}`

Table 6.1: Call-site vs entry-point locking for mutex calls

Note the `mutex` keyword relies on the type system, which means that in cases where a generic monitor-routine is desired, writing the mutex routine is possible with the proper trait, e.g.:

```
// Incorrect: T may not be monitor
forall(dtype T)
void foo(T * mutex t);

// Correct: this function only works on monitors (any monitor)
forall(dtype T | is_monitor(T))
void bar(T * mutex t));
```

Both entry point and callsite-locking are feasible implementations. The current C∀ implementation uses entry-point locking because it requires less work when using Resource Acquisition Is Initialization (RAII), effectively transferring the burden of implementation to object construction/destruction. It is harder to use RAII for call-site locking, as it does not necessarily have an existing scope that matches exactly the scope of the mutual exclusion, i.e., the function body. For example, the monitor call can appear in the middle of an expression. Furthermore, entry-point locking requires less code generation since any useful routine is called multiple times but there is only one entry point for many call sites.

6.2 Threading

Figure 6.1 shows a high-level picture if the C∀ runtime system in regards to concurrency. Each component of the picture is explained in detail in the flowing sections.

6.2.1 Processors

Parallelism in C∀ is built around using processors to specify how much parallelism is desired. C∀ processors are object wrappers around kernel threads, specifically `pthread`s in the current implementation of C∀. Indeed, any parallelism must go through operating-system libraries. However, user-level threads are still the main source of concurrency, processors are simply the underlying source of parallelism. Indeed, processor kernel-level threads simply fetch a user-level thread from the scheduler and run it; they are effectively executers for user-threads. The main benefit of this approach is that it offers a well-defined boundary between kernel code and user code, for example, kernel thread quiescing, scheduling and interrupt handling. Processors internally use coroutines to take advantage of the existing context-switching semantics.

6.2.2 Stack Management

One of the challenges of this system is to reduce the footprint as much as possible. Specifically, all `pthread`s created also have a stack created with them, which should be used as much as possible. Normally, coroutines also create their own stack to run on, however, in the case of the coroutines used for processors, these coroutines run directly on the kernel-level thread stack, effectively stealing the processor stack. The exception to this rule is the Main Processor, i.e., the initial kernel-level thread that is given to any program. In order to respect C user expectations, the stack of the initial kernel thread, the main stack of the program, is used by the main user thread rather than the main processor, which can grow very large.

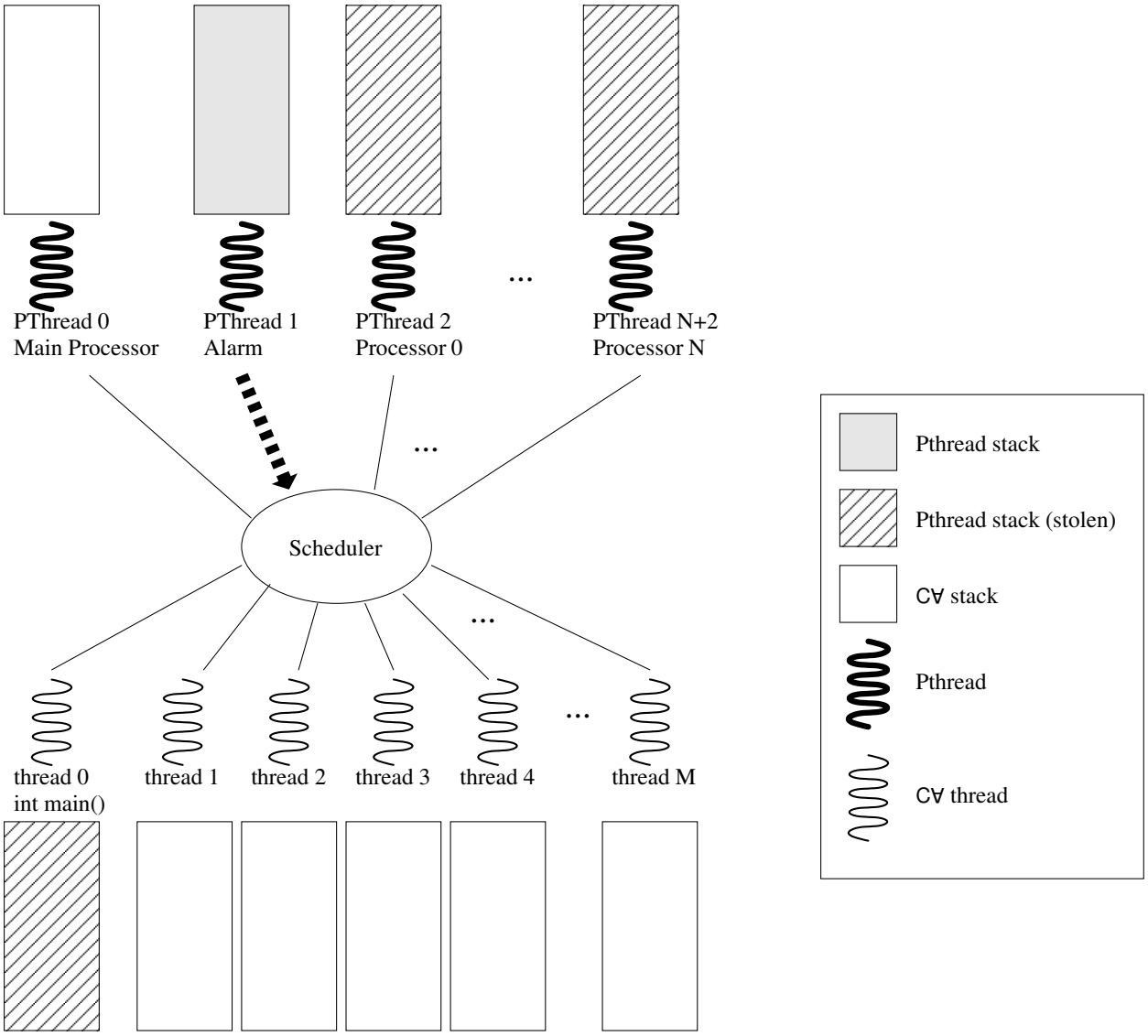

Figure 6.1: Overview of the entire system

6.2.3 Context Switching

As mentioned in section 3.3, coroutines are a stepping stone for implementing threading, because they share the same mechanism for context-switching between different stacks. To improve performance and simplicity, context-switching is implemented using the following assumption: all context-switches happen inside a specific function call. This assumption means that the context-switch only has to copy the callee-saved registers onto the stack and then switch the stack registers with the ones of the target coroutine/thread. Note that the instruction pointer can be left untouched since the context-switch is always inside the same function. Threads, however, do not context-switch between each other directly. They context-switch to the scheduler. This method is called a 2-step context-switch and has the advantage of having a clear distinction between user code and the kernel where scheduling and other system operations happen. Obviously, this doubles the context-switch

cost because threads must context-switch to an intermediate stack. The alternative 1-step context-switch uses the stack of the "from" thread to schedule and then context-switches directly to the "to" thread. However, the performance of the 2-step context-switch is still superior to a `pthread_yield` (see section 8). Additionally, for users in need for optimal performance, it is important to note that having a 2-step context-switch as the default does not prevent C∀ from offering a 1-step context-switch (akin to the Microsoft `SwitchToFiber` [8] routine). This option is not currently present in C∀, but the changes required to add it are strictly additive.

6.2.4 Preemption

Finally, an important aspect for any complete threading system is preemption. As mentioned in chapter 3, preemption introduces an extra degree of uncertainty, which enables users to have multiple threads interleave transparently, rather than having to cooperate among threads for proper scheduling and CPU distribution. Indeed, preemption is desirable because it adds a degree of isolation among threads. In a fully cooperative system, any thread that runs a long loop can starve other threads, while in a preemptive system, starvation can still occur but it does not rely on every thread having to yield or block on a regular basis, which reduces significantly a programmer burden. Obviously, preemption is not optimal for every workload. However any preemptive system can become a cooperative system by making the time slices extremely large. Therefore, C∀ uses a preemptive threading system.

Preemption in C∀[1] is based on kernel timers, which are used to run a discrete-event simulation. Every processor keeps track of the current time and registers an expiration time with the preemption system. When the preemption system receives a change in preemption, it inserts the time in a sorted order and sets a kernel timer for the closest one, effectively stepping through preemption events on each signal sent by the timer. These timers use the Linux signal `SIGALRM`, which is delivered to the process rather than the kernel-thread. This results in an implementation problem, because when delivering signals to a process, the kernel can deliver the signal to any kernel thread for which the signal is not blocked, i.e.:

> A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses an arbitrary thread to which to deliver the signal.
> SIGNAL(7) - Linux Programmer's Manual

For the sake of simplicity, and in order to prevent the case of having two threads receiving alarms simultaneously, C∀ programs block the `SIGALRM` signal on every kernel thread except one.

Now because of how involuntary context-switches are handled, the kernel thread handling `SIGALRM` cannot also be a processor thread. Hence, involuntary context-switching is done by sending signal `SIGUSR1` to the corresponding processor and having the thread yield from inside the signal handler. This approach effectively context-switches away from the signal handler back to the kernel and the signal handler frame is eventually unwound when the thread is scheduled again. As a result, a signal handler can start on one kernel thread and terminate on a second kernel thread (but the same user thread). It is important to note that signal handlers save and restore signal

[1]Note that the implementation of preemption is strongly tied with the underlying threading system. For this reason, only the Linux implementation is cover, C∀ does not run on Windows at the time of writting

masks because user-thread migration can cause a signal mask to migrate from one kernel thread to another. This behaviour is only a problem if all kernel threads, among which a user thread can migrate, differ in terms of signal masks[2]. However, since the kernel thread handling preemption requires a different signal mask, executing user threads on the kernel-alarm thread can cause deadlocks. For this reason, the alarm thread is in a tight loop around a system call to `sigwaitinfo`, requiring very little CPU time for preemption. One final detail about the alarm thread is how to wake it when additional communication is required (e.g., on thread termination). This unblocking is also done using `SIGALRM`, but sent through the `pthread_sigqueue`. Indeed, `sigwait` can differentiate signals sent from `pthread_sigqueue` from signals sent from alarms or the kernel.

6.2.5 Scheduler

Finally, an aspect that was not mentioned yet is the scheduling algorithm. Currently, the CV scheduler uses a single ready queue for all processors, which is the simplest approach to scheduling. Further discussion on scheduling is present in section 9.1.2.

6.3 Internal Scheduling

The following figure is the traditional illustration of a monitor (repeated from page 31 for convenience):

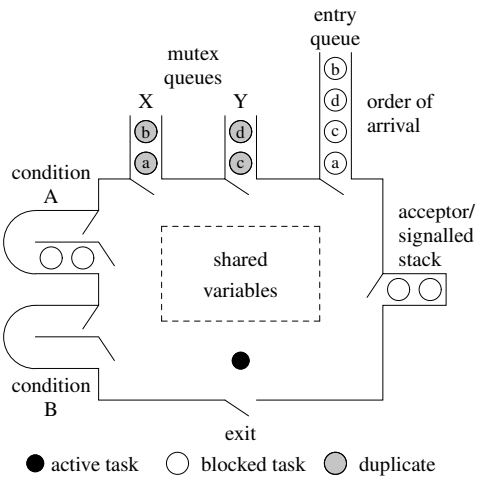

Figure 6.2: Traditional illustration of a monitor

This picture has several components, the two most important being the entry queue and the AS-stack. The entry queue is an (almost) FIFO list where threads waiting to enter are parked, while the acceptor/signaller (AS) stack is a FILO list used for threads that have been signalled or otherwise marked as running next.

For CV, this picture does not have support for blocking multiple monitors on a single condition. To support bulk-acquiring two changes to this picture are required. First, it is no longer helpful to

[2]Sadly, official POSIX documentation is silent on what distinguishes "async-signal-safe" functions from other functions.

attach the condition to *a single* monitor. Secondly, the thread waiting on the condition has to be separated across multiple monitors, seen in figure 6.3.

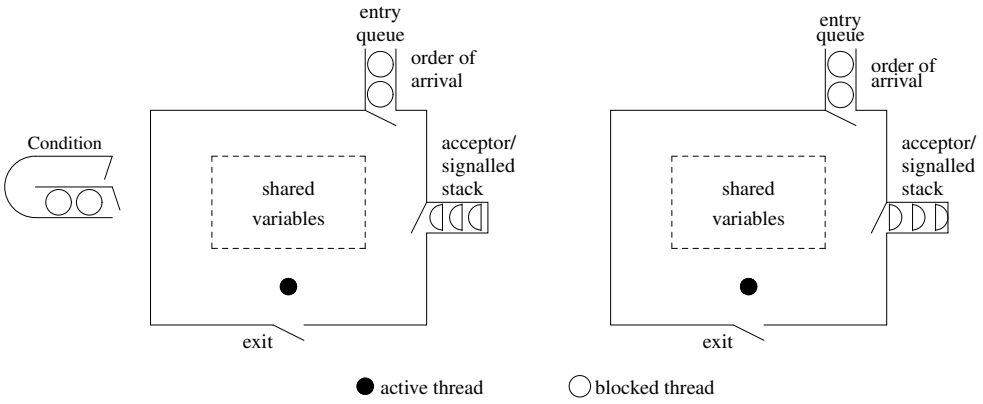

Figure 6.3: Illustration of C∀ Monitor

This picture and the proper entry and leave algorithms (see listing 6.2) is the fundamental implementation of internal scheduling. Note that when a thread is moved from the condition to the AS-stack, it is conceptually split into N pieces, where N is the number of monitors specified in the parameter list. The thread is woken up when all the pieces have popped from the AS-stacks and made active. In this picture, the threads are split into halves but this is only because there are two monitors. For a specific signalling operation every monitor needs a piece of thread on its AS-stack.

The solution discussed in 4.3 can be seen in the exit routine of listing 6.2. Basically, the solution boils down to having a separate data structure for the condition queue and the AS-stack, and unconditionally transferring ownership of the monitors but only unblocking the thread when the last monitor has transferred ownership. This solution is deadlock safe as well as preventing any potential barging. The data structures used for the AS-stack are reused extensively for external scheduling, but in the case of internal scheduling, the data is allocated using variable-length arrays on the call stack of the `wait` and `signal_block` routines.

Entry

```
if monitor is free
    enter
elif already own the monitor
    continue
else
    block
increment recursion
```

Exit

```
decrement recursion
if recursion == 0
    if signal_stack not empty
        set_owner to thread
        if all monitors ready
            wake-up thread

    if entry queue not empty
        wake-up thread
```

Listing 6.2: Entry and exit routine for monitors with internal scheduling

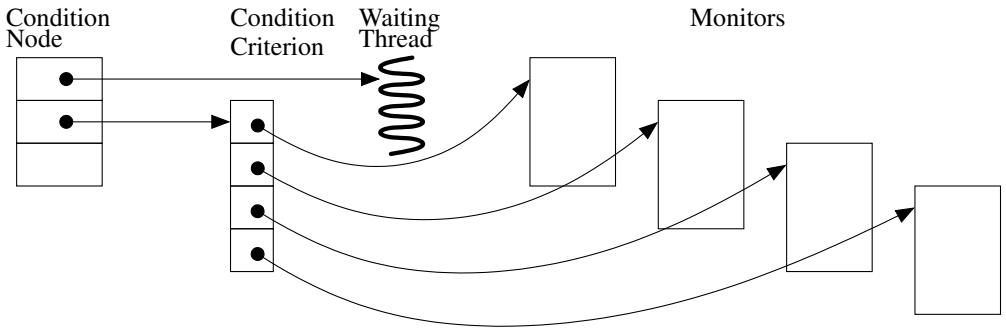

Figure 6.4: Data structures involved in internal/external scheduling

Figure 6.4 shows a high-level representation of these data structures. The main idea behind them is that, a thread cannot contain an arbitrary number of intrusive "next" pointers for linking onto monitors. The `condition node` is the data structure that is queued onto a condition variable and, when signalled, the condition queue is popped and each `condition criterion` is moved to the AS-stack. Once all the criteria have been popped from their respective AS-stacks, the thread is woken up, which is what is shown in listing 6.2.

6.4 External Scheduling

Similarly to internal scheduling, external scheduling for multiple monitors relies on the idea that waiting-thread queues are no longer specific to a single monitor, as mentioned in section 4.4. For internal scheduling, these queues are part of condition variables, which are still unique for a given scheduling operation (i.e., no signal statement uses multiple conditions). However, in the case of external scheduling, there is no equivalent object which is associated with `waitfor` statements. This absence means the queues holding the waiting threads must be stored inside at least one of the monitors that is acquired. These monitors being the only objects that have sufficient lifetime and are available on both sides of the `waitfor` statement. This requires an algorithm to choose which monitor holds the relevant queue. It is also important that said algorithm be independent of the order in which users list parameters. The proposed algorithm is to fall back on monitor lock ordering (sorting by address) and specify that the monitor that is acquired first is the one with the relevant waiting queue. This assumes that the lock acquiring order is static for the lifetime of all concerned objects but that is a reasonable constraint.

This algorithm choice has two consequences:

- The queue of the monitor with the lowest address is no longer a true FIFO queue because threads can be moved to the front of the queue. These queues need to contain a set of monitors for each of the waiting threads. Therefore, another thread whose set contains the same lowest address monitor but different lower priority monitors may arrive first but enter the critical section after a thread with the correct pairing.
- The queue of the lowest priority monitor is both required and potentially unused. Indeed, since it is not known at compile time which monitor is the monitor which has the lowest address, every monitor needs to have the correct queues even though it is possible that some queues go unused for the entire duration of the program, for example if a monitor is only used in a specific

Entry

```
if monitor is free
    enter
elif already own the monitor
    continue
elif matches waitfor mask
    push criteria to AS-stack
    continue
else
    block
increment recursion
```

Exit

```
decrement recursion
if recursion == 0
    if signal_stack not empty
        set_owner to thread
        if all monitors ready
            wake-up thread
        endif
    endif

    if entry queue not empty
        wake-up thread
    endif
```

Listing 6.3: Entry and exit routine for monitors with internal scheduling and external scheduling

pair.

Therefore, the following modifications need to be made to support external scheduling:

- The threads waiting on the entry queue need to keep track of which routine they are trying to enter, and using which set of monitors. The `mutex` routine already has all the required information on its stack, so the thread only needs to keep a pointer to that information.
- The monitors need to keep a mask of acceptable routines. This mask contains for each acceptable routine, a routine pointer and an array of monitors to go with it. It also needs storage to keep track of which routine was accepted. Since this information is not specific to any monitor, the monitors actually contain a pointer to an integer on the stack of the waiting thread. Note that if a thread has acquired two monitors but executes a `waitfor` with only one monitor as a parameter, setting the mask of acceptable routines to both monitors will not cause any problems since the extra monitor will not change ownership regardless. This becomes relevant when `when` clauses affect the number of monitors passed to a `waitfor` statement.
- The entry/exit routines need to be updated as shown in listing 6.3.

6.4.1 External Scheduling - Destructors

Finally, to support the ordering inversion of destructors, the code generation needs to be modified to use a special entry routine. This routine is needed because of the storage requirements of the call order inversion. Indeed, when waiting for the destructors, storage is needed for the waiting context and the lifetime of said storage needs to outlive the waiting operation it is needed for. For regular `waitfor` statements, the call stack of the routine itself matches this requirement but it is no longer the case when waiting for the destructor since it is pushed on to the AS-stack for later. The `waitfor` semantics can then be adjusted correspondingly, as seen in listing 6.4

Destructor Entry

```
if monitor is free
    enter
elif already own the monitor
    increment recursion
    return
create wait context
if matches waitfor mask
    reset mask
    push self to AS-stack
    baton pass
else
    wait
increment recursion
```

Waitfor

```
if matching thread is already there
    if found destructor
        push destructor to AS-stack
        unlock all monitors
    else
        push self to AS-stack
        baton pass
    endif
    return
endif
if non-blocking
    Unlock all monitors
    Return
endif

push self to AS-stack
set waitfor mask
block
return
```

Listing 6.4: Pseudo code for the `waitfor` routine and the `mutex` entry routine for destructors

Chapter 7
Putting It All Together

7.1 Threads As Monitors

As it was subtly alluded in section 3.4, `thread`s in C∀ are in fact monitors, which means that all monitor features are available when using threads. For example, here is a very simple two thread pipeline that could be used for a simulator of a game engine:

```
// Visualization declaration
thread Renderer {} renderer;
Frame * simulate( Simulator & this );

// Simulation declaration
thread Simulator{} simulator;
void render( Renderer & this );

// Blocking call used as communication
void draw( Renderer & mutex this, Frame * frame );

// Simulation loop
void main( Simulator & this ) {
    while( true ) {
        Frame * frame = simulate( this );
        draw( renderer, frame );
    }
}

// Rendering loop
void main( Renderer & this ) {
    while( true ) {
        waitfor( draw, this );
        render( this );
    }
}
```

Listing 7.1: Toy simulator using `thread`s and `monitor`s.

One of the obvious complaints of the previous code snippet (other than its toy-like simplicity) is that it does not handle exit conditions and just goes on forever. Luckily, the monitor semantics can also be used to clearly enforce a shutdown order in a concise manner:

```
// Visualization declaration
thread Renderer {} renderer;
Frame * simulate( Simulator & this );

// Simulation declaration
thread Simulator{} simulator;
void render( Renderer & this );

// Blocking call used as communication
void draw( Renderer & mutex this, Frame * frame );

// Simulation loop
void main( Simulator & this ) {
    while( true ) {
        Frame * frame = simulate( this );
        draw( renderer, frame );

        // Exit main loop after the last frame
        if( frame->is_last ) break;
    }
}

// Rendering loop
void main( Renderer & this ) {
    while( true ) {
           waitfor( draw, this );
        or waitfor( ^?{}, this ) {
            // Add an exit condition
            break;
        }

        render( this );
    }
}

// Call destructor for simulator once simulator finishes
// Call destructor for renderer to signify shutdown
```

Listing 7.2: Same toy simulator with proper termination condition.

7.2 Fibers & Threads

As mentioned in section 6.2.4, C∀ uses preemptive threads by default but can use fibers on demand. Currently, using fibers is done by adding the following line of code to the program :

```
unsigned int default_preemption() {
    return 0;
}
```

This function is called by the kernel to fetch the default preemption rate, where 0 signifies an infinite time-slice, i.e., no preemption. However, once clusters are fully implemented, it will be possible to create fibers and user-level threads in the same system, as in listing 7.3

```
//Cluster forward declaration
struct cluster;

//Processor forward declaration
struct processor;

//Construct clusters with a preemption rate
void ?{}(cluster& this, unsigned int rate);
//Construct processor and add it to cluster
void ?{}(processor& this, cluster& cluster);
//Construct thread and schedule it on cluster
void ?{}(thread& this, cluster& cluster);

//Declare two clusters
cluster thread_cluster = { 10`ms };         //Preempt every 10 ms
cluster fibers_cluster = { 0 };             //Never preempt

//Construct 4 processors
processor processors[4] = {
    //2 for the thread cluster
    thread_cluster;
    thread_cluster;
    //2 for the fibers cluster
    fibers_cluster;
    fibers_cluster;
};

//Declares thread
thread UThread {};
void ?{}(UThread& this) {
    //Construct underlying thread to automatically
    //be scheduled on the thread cluster
    (this){ thread_cluster }
}

void main(UThread & this);

//Declares fibers
thread Fiber {};
void ?{}(Fiber& this) {
    //Construct underlying thread to automatically
    //be scheduled on the fiber cluster
    (this.__thread){ fibers_cluster }
}

void main(Fiber & this);
```

Listing 7.3: Using fibers and user-level threads side-by-side in C∀

Chapter 8
Performance Results

8.1 Machine Setup

Table 8.1 shows the characteristics of the machine used to run the benchmarks. All tests were made on this machine.

Architecture	x86_64	NUMA node(s)	8
CPU op-mode(s)	32-bit, 64-bit	Model name	AMD Opteron™Processor 6380
Byte Order	Little Endian	CPU Freq	2.5GHz
CPU(s)	64	L1d cache	16 KiB
Thread(s) per core	2	L1i cache	64 KiB
Core(s) per socket	8	L2 cache	2048 KiB
Socket(s)	4	L3 cache	6144 KiB
Operating system	Ubuntu 16.04.3 LTS	Kernel	Linux 4.4-97-generic
Compiler	GCC 6.3	Translator	CFA 1
Java version	OpenJDK-9	Go version	1.9.2

Table 8.1: Machine setup used for the tests

8.2 Micro Benchmarks

All benchmarks are run using the same harness to produce the results, seen as the `BENCH()` macro in the following examples. This macro uses the following logic to benchmark the code:

```
#define BENCH(run, result) \
    before = gettime(); \
    run; \
    after  = gettime(); \
    result = (after - before) / N;
```

The method used to get time is `clock_gettime(CLOCK_THREAD_CPUTIME_ID);`. Each benchmark is using many iterations of a simple call to measure the cost of the call. The specific number of iterations depends on the specific benchmark.

C∀ Coroutines

```
coroutine GreatSuspender {};
void main(GreatSuspender& this) {
    while(true) { suspend(); }
}
int main() {
    GreatSuspender s;
    resume(s);
    BENCH(
        for(size_t i=0; i<n; i++) {
            resume(s);
        },
        result
    )
    printf("%llu\n", result);
}
```

C∀ Threads

```
int main() {


    BENCH(
        for(size_t i=0; i<n; i++) {
            yield();
        },
        result
    )
    printf("%llu\n", result);
}
```

Listing 8.1: C∀ benchmark code used to measure context-switches for coroutines and threads.

	Median	Average	Standard Deviation
Kernel Thread	241.5	243.86	5.08
C∀ Coroutine	38	38	0
C∀ Thread	103	102.96	2.96
μC++ Coroutine	46	45.86	0.35
μC++ Thread	98	99.11	1.42
Goroutine	150	149.96	3.16
Java Thread	289	290.68	8.72

Table 8.2: Context Switch comparison. All numbers are in nanoseconds(ns)

8.2.1 Context-Switching

The first interesting benchmark is to measure how long context-switches take. The simplest approach to do this is to yield on a thread, which executes a 2-step context switch. Yielding causes the thread to context-switch to the scheduler and back, more precisely: from the user-level thread to the kernel-level thread then from the kernel-level thread back to the same user-level thread (or a different one in the general case). In order to make the comparison fair, coroutines also execute a 2-step context-switch by resuming another coroutine which does nothing but suspending in a tight loop, which is a resume/suspend cycle instead of a yield. Listing 8.1 shows the code for coroutines and threads with the results in table 8.2. All omitted tests are functionally identical to one of these tests. The difference between coroutines and threads can be attributed to the cost of scheduling.

8.2.2 Mutual-Exclusion

The next interesting benchmark is to measure the overhead to enter/leave a critical-section. For monitors, the simplest approach is to measure how long it takes to enter and leave a monitor routine. Listing 8.2 shows the code for C∀. To put the results in context, the cost of entering a non-

```
monitor M {};
void __attribute__((noinline)) call( M & mutex m /*, m2, m3, m4*/ ) {}

int main() {
    M m/*, m2, m3, m4*/;
    BENCH(
        for(size_t i=0; i<n; i++) {
            call(m/*, m2, m3, m4*/);
        },
        result
    )
    printf("%llu\n", result);
}
```

Listing 8.2: C∀ benchmark code used to measure mutex routines.

	Median	Average	Standard Deviation
C routine	2	2	0
FetchAdd + FetchSub	26	26	0
Pthreads Mutex Lock	31	31.86	0.99
μC++ `monitor` member routine	30	30	0
C∀ `mutex` routine, 1 argument	41	41.57	0.9
C∀ `mutex` routine, 2 argument	76	76.96	1.57
C∀ `mutex` routine, 4 argument	145	146.68	3.85
Java synchronized routine	27	28.57	2.6

Table 8.3: Mutex routine comparison. All numbers are in nanoseconds(ns)

inline function and the cost of acquiring and releasing a `pthread_mutex` lock is also measured. The results can be shown in table 8.3.

8.2.3 Internal Scheduling

The internal-scheduling benchmark measures the cost of waiting on and signalling a condition variable. Listing 8.3 shows the code for C∀, with results table 8.4. As with all other benchmarks, all omitted tests are functionally identical to one of these tests.

	Median	Average	Standard Deviation
Pthreads Condition Variable	5902.5	6093.29	714.78
μC++ `signal`	322	323	3.36
C∀ `signal`, 1 `monitor`	352.5	353.11	3.66
C∀ `signal`, 2 `monitor`	430	430.29	8.97
C∀ `signal`, 4 `monitor`	594.5	606.57	18.33
Java `notify`	13 831.5	15 698.21	4782.3

Table 8.4: Internal scheduling comparison. All numbers are in nanoseconds(ns)

```
volatile int go = 0;
condition c;
monitor M {};
M m1;

void __attribute__((noinline)) do_call( M & mutex a1 ) { signal(c); }

thread T {};
void ^?{}( T & mutex this ) {}
void main( T & this ) {
    while(go == 0) { yield(); }
    while(go == 1) { do_call(m1); }
}
int  __attribute__((noinline)) do_wait( M & mutex a1 ) {
    go = 1;
    BENCH(
        for(size_t i=0; i<n; i++) {
            wait(c);
        },
        result
    )
    printf("%llu\n", result);
    go = 0;
    return 0;
}
int main() {
    T t;
    return do_wait(m1);
}
```

Listing 8.3: Benchmark code for internal scheduling

	Median	Average	Standard Deviation
μC++ `Accept`	350	350.61	3.11
C∀ `waitfor`, 1 `monitor`	358.5	358.36	3.82
C∀ `waitfor`, 2 `monitor`	422	426.79	7.95
C∀ `waitfor`, 4 `monitor`	579.5	585.46	11.25

Table 8.5: External scheduling comparison. All numbers are in nanoseconds(ns)

8.2.4 External Scheduling

The Internal scheduling benchmark measures the cost of the `waitfor` statement (`_Accept` in μC++). Listing 8.4 shows the code for C∀, with results in table 8.5. As with all other benchmarks, all omitted tests are functionally identical to one of these tests.

8.2.5 Object Creation

Finally, the last benchmark measures the cost of creation for concurrent objects. Listing 8.5 shows the code for `pthreads` and C∀ threads, with results shown in table 8.6. As with all other benchmarks, all omitted tests are functionally identical to one of these tests. The only note here is that the call stacks of C∀ coroutines are lazily created, therefore without priming the coroutine, the

```
volatile int go = 0;
monitor M {};
M m1;
thread T {};

void __attribute__((noinline)) do_call( M & mutex a1 ) {}

void ^?{}( T & mutex this ) {}
void main( T & this ) {
    while(go == 0) { yield(); }
    while(go == 1) { do_call(m1); }
}
int  __attribute__((noinline)) do_wait( M & mutex a1 ) {
    go = 1;
    BENCH(
        for(size_t i=0; i<n; i++) {
            waitfor(call, a1);
        },
        result
    )
    printf("%llu\n", result);
    go = 0;
    return 0;
}
int main() {
    T t;
    return do_wait(m1);
}
```

Listing 8.4: Benchmark code for external scheduling

	Median	Average	Standard Deviation
Pthreads	26996	26984.71	156.6
C∀ Coroutine Lazy	6	5.71	0.45
C∀ Coroutine Eager	708	706.68	4.82
C∀ Thread	1173.5	1176.18	15.18
μC++ Coroutine	109	107.46	1.74
μC++ Thread	526	530.89	9.73
Goroutine	2520.5	2530.93	61.56
Java Thread	91114.5	92272.79	961.58

Table 8.6: Creation comparison. All numbers are in nanoseconds(ns).

creation cost is very low.

pthread

```
int main() {
    BENCH(
        for(size_t i=0; i<n; i++) {
            pthread_t thread;
            if(pthread_create(&thread,NULL,foo,NULL)<0) {
                perror( "failure" );
                return 1;
            }

            if(pthread_join(thread, NULL)<0) {
                perror( "failure" );
                return 1;
            }
        },
        result
    )
    printf("%llu\n", result);
}
```

C∀ Threads

```
int main() {
    BENCH(
        for(size_t i=0; i<n; i++) {
            MyThread m;
        },
        result
    )
    printf("%llu\n", result);
}
```

Listing 8.5: Benchmark code for pthreads and C∀ to measure object creation

Chapter 9
Conclusion

This thesis has achieved a minimal concurrency API that is simple, efficient and usable as the basis for higher-level features. The approach presented is based on a lightweight thread-system for parallelism, which sits on top of clusters of processors. This M:N model is judged to be both more efficient and allow more flexibility for users. Furthermore, this document introduces monitors as the main concurrency tool for users. This thesis also offers a novel approach allowing multiple monitors to be accessed simultaneously without running into the Nested Monitor Problem [47]. It also offers a full implementation of the concurrency runtime written entirely in C∀, effectively the largest C∀ code base to date.

9.1 Future Work

9.1.1 Performance

This thesis presents a first implementation of the C∀ concurrency runtime. Therefore, there is still significant work to improve performance. Many of the data structures and algorithms may change in the future to more efficient versions. For example, the number of monitors in a single bulk-acquiring is only bound by the stack size, this is probably unnecessarily generous. It may be possible that limiting the number helps increase performance. However, it is not obvious that the benefit would be significant.

9.1.2 Flexible Scheduling

An important part of concurrency is scheduling. Different scheduling algorithms can affect performance (both in terms of average and variation). However, no single scheduler is optimal for all workloads and therefore there is value in being able to change the scheduler for given programs. One solution is to offer various tweaking options to users, allowing the scheduler to be adjusted to the requirements of the workload. However, in order to be truly flexible, it would be interesting to allow users to add arbitrary data and arbitrary scheduling algorithms. For example, a web server could attach Type-of-Service information to threads and have a "ToS aware" scheduling algorithm tailored to this specific web server. This path of flexible schedulers will be explored for C∀.

9.1.3 Non-Blocking I/O

While most of the parallelism tools are aimed at data parallelism and control-flow parallelism, many modern workloads are not bound on computation but on IO operations, a common case being web servers and XaaS (anything as a service). These types of workloads often require significant engineering around amortizing costs of blocking IO operations. At its core, non-blocking I/O is an operating system level feature that allows queuing IO operations (e.g., network operations) and registering for notifications instead of waiting for requests to complete. In this context, the role of the language makes Non-Blocking IO easily available and with low overhead. The current trend is to use asynchronous programming using tools like callbacks and/or futures and promises, which can be seen in frameworks like Node.js [5] for JavaScript, Spring MVC [6] for Java and Django [2] for Python. However, while these are valid solutions, they lead to code that is harder to read and maintain because it is much less linear.

9.1.4 Other Concurrency Tools

While monitors offer a flexible and powerful concurrent core for C∀, other concurrency tools are also necessary for a complete multi-paradigm concurrency package. Examples of such tools can include simple locks and condition variables, futures and promises [46], executors and actors. These additional features are useful when monitors offer a level of abstraction that is inadequate for certain tasks.

9.1.5 Implicit Threading

Simpler applications can benefit greatly from having implicit parallelism. That is, parallelism that does not rely on the user to write concurrency. This type of parallelism can be achieved both at the language level and at the library level. The canonical example of implicit parallelism is parallel for loops, which are the simplest example of a divide and conquer algorithms [16]. Table 9.1 shows three different code examples that accomplish point-wise sums of large arrays. Note that none of these examples explicitly declare any concurrency or parallelism objects.

Implicit parallelism is a restrictive solution and therefore has its limitations. However, it is a quick and simple approach to parallelism, which may very well be sufficient for smaller applications and reduces the amount of boilerplate needed to start benefiting from parallelism in modern CPUs.

Sequential

```
void big_sum(
   int* a, int* b,
   int* o,
   size_t len)
{
   for(
      int i = 0;
      i < len;
      ++i )
   {
      o[i]=a[i]+b[i];
   }
}



int* a[10000];
int* b[10000];
int* c[10000];
//... fill in a & b
big_sum(a,b,c,10000);
```

Library Parallel

```
void big_sum(
   int* a, int* b,
   int* o,
   size_t len)
{
   range ar(a, a+len);
   range br(b, b+len);
   range or(o, o+len);
   parfor( ai, bi, oi,
   [](   int* ai,
      int* bi,
      int* oi)
   {
      oi=ai+bi;
   });
}


int* a[10000];
int* b[10000];
int* c[10000];
//... fill in a & b
big_sum(a,b,c,10000);
```

Language Parallel

```
void big_sum(
   int* a, int* b,
   int* o,
   size_t len)
{
   parfor (ai,bi,oi)
       in (a, b, o )
   {
      oi = ai + bi;
   }
}



int* a[10000];
int* b[10000];
int* c[10000];
//... fill in a & b
big_sum(a,b,c,10000);
```

Table 9.1: For loop to sum numbers: Sequential, using library parallelism and language parallelism.

Bibliography

[1] *Affinity API Release Notes for OS X v10.5*. 39

[2] Django. https://www.djangoproject.com/. 60

[3] *FreeBSD General Commands Manual - CPUSET(1)*. 39

[4] *NetBSD Library Functions Manual - AFFINITY(3)*. 39

[5] Node.js. https://nodejs.org/en/. 60

[6] Spring Web MVC. https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html. 60

[7] *Windows (vs.85) - SetThreadAffinityMask function*. 39

[8] *Windows (vs.85) - SwitchToFiber function*. 44

[9] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, 1986. 39

[10] Gregory R. Andrews. A method for solving synronization problems. *Science of Computer Programming*, 13(4):1–21, December 1989. 17

[11] Bank account transfer problem. 2010. 21

[12] Andrew Birrell, Mark R. Brown, Luca Cardelli, Jim Donahue, Lucille Glassman, John Gutag, Jim Harning, Bill Kalsow, Roy Levin, and Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, 1991. 16

[13] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, 1973. 16

[14] Per Brinch Hansen. The programming language concurrent pascal. *IEEE Trans. Softw. Eng.*, 2:199–206, June 1975. 16

[15] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the object-oriented language C++. *Softw. Pract. Exp.*, 22(2):137–172, February 1992. 16

[16] Peter A. Buhr. *Understanding Control Flow: Concurrent Programming using μC++*. Springer, Switzerland, 2016. 18, 37, 60

[17] Peter A. Buhr, Glen Ditchfield, David Till, and Charles R. Zarnke. C$\forall$ users guide, version 0.1. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, October 2001. http://plg.uwaterloo.ca/~cforall/cfa.ps. 5

[18] Peter A. Buhr and Ashif S. Harji. Concurrent urban legends. *Concurrency Comput. Pract. Exp.*, 17(9):1133–1172, August 2005. 1

[19] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley, Boston, 1997. 11

[20] C11. *American National Standard Information technology – Programming Languages – C*. International Standard ISO/IEC 9899-2011[2012], http://www.iso.org, 2012. 2

[21] R. H. Campbell and A. N. Habermann. *The Specification of Process Synchronization by Path Expressions*, volume 16 of *Lecture Notes in Computer Science*. Springer, 1974. 16

[22] C∀. *C∀ Programmming Language*. https://plg.uwaterloo.ca/~cforall. 2, 5

[23] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. American Elsevier, 1982. 16

[24] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988. 16

[25] Dave Dice, Yossi Lev, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'10, pages 325–334, New York, NY, USA, 2010. ACM. 21

[26] E. W. Dijkstra. The structure of the "THE"–multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968. 16

[27] ECMA International Standardizing Information and Communication Systems. *C# Language Specification, Standard ECMA-334*, 4th edition, June 2006. 13

[28] École Polytechnique Fédérale de Lausanne. *Scala Language Specification, Version 2.11*, 2016. http://www.scala-lang.org/files/archive/spec/2.11. 13

[29] Erlang AB. *Erlang/OTP System Documentation 8.1*, September 2016. http://erlang.org/doc/-pdf/otp-system-documentation.pdf. 16, 37

[30] Martin Fowler. Twohardthings. https://martinfowler.com/bliki/TwoHardThings.html, 2009. 21

[31] W. Morven Gentleman. Using the harmony operating system. Technical Report 24685, National Research Council of Canada, Ottawa, Canada, May 1985. 16

[32] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, 2nd edition, 2000. 16, 18

[33] James Gosling, David S. H. Rosenthal, and Richelle J. Arden. *The NeWS Book*. Springer-Verlag, 1989. 16

[34] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, 2009. http://golang.org/ref/spec. 16, 38

[35] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84, 1968. 40

[36] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. 16

[37] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. iii, 16, 22

[38] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. 16

[39] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K. Hollingsworth, and Marvin V. Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. 1, 16

[40] R. C. Holt and J. R. Cordy. The turing programming language. *Communications of the ACM*, 31(12):1410–1423, December 1988. 16

[41] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *SIGPLAN Not.*, 27(5):T1–53, May 1992. 16

[42] International Standard ISO/IEC 14882:2014 (E), http://www.iso.org. *Programming Languages – C++*, 4th edition, 2014. 11

[43] International Standard ISO/IEC TS 19841:2015, http://www.iso.org. *Technical Specification for C++ Extensions for Transactional Memory*, 2015. 16

[44] Oliver Kowalke. Boost coroutine library, 2015. http://www.boost.org/doc/libs/1_61_0/libs/-coroutine/doc/html/index.html [Accessed September 2016]. 11

[45] Lightbend Inc. *Akka Scala Documentation, Release 2.4.11*, September 2016. http://-doc.akka.io/docs/akka/2.4/AkkaScala.pdf. 16

[46] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, July 1988. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. 60

[47] Andrew Lister. The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7, July 1977. 20, 24, 59

[48] Microsoft Corporation. *Microsoft Visual C++ .NET Language Reference*, 2002. Microsoft Press, Redmond, Washington, U.S.A. 11

[49] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, April 1979. 16

[50] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Softw. Pract. Exp.*, 21(1):91–118, January 1991. 16

[51] Rob Schluntz. Resource management and tuples in cforall. Master's thesis, University of Waterloo, 2017. https://uwspace.uwaterloo.ca/handle/10012/11830. 5

[52] Herb Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal : Software Tools for the Professional Programmer*, 30(3):16–22, March 2005. 7, 37

[53] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005. 7, 37

[54] Intel thread building blocks. https://www.threadingbuildingblocks.org/. 38

[55] *Linux man page - sched_setaffinity(2)*. 39

[56] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Springer, New York. 3

[57] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer, New York, 4th edition, 1988. 16

[58] Doug Zongker. Chicken chicken chicken: Chicken chicken. 2006. 40

Glossary

bulk-acquiring Implicitly acquiring several monitors when entering a monitor. x, 20–25, 31, 40, 45, 59

callsite-locking Locking done by the calling routine. With this technique, a routine calling a monitor routine aquires the monitor *before* making the call to the actuall routine. 41, 42

cluster A group of kernel-level thread executed in isolation.

Synonyms : None. 38, 39

entry-point-locking Locking done by the called routine. With this technique, a monitor routine called by another routine aquires the monitor *after* entering the routine body but prior to any other code. 41

fiber Fibers are non-preemptive user-level threads. They share most of the caracteristics of user-level threads except that they cannot be preempted by another fiber.

Synonyms : Tasks. 37, 39

job Unit of work, often sent to a thread pool or worker pool to be executed. Has neither its own stack nor its own thread of execution.

Synonyms : Tasks. 38

kernel-level thread Threads created and managed inside kernel-space. Each thread has its own stack and its own thread of execution. Kernel-level threads are owned, managed and scheduled by the underlying operating system.

Synonyms : OS threads, Hardware threads, Physical threads. 37, 38, 42, 54, 66

multiple-acquisition Any locking technique that allows a single thread to acquire the same lock multiple times. x, 18–21

preemption Involuntary context switch imposed on threads at a specified rate.

Synonyms : None. 37, 39

thread User level threads that are the default in C∀. Generally declared using the `thread` keyword.

Synonyms : None. 13, 39

thread-pool Group of homogeneuous threads that loop executing units of works after another.

Synonyms : 38

user-level thread Threads created and managed inside user-space. Each thread has its own stack and its own thread of execution. User-level threads are invisible to the underlying operating system.

Synonyms : User threads, Lightweight threads, Green threads, Virtual threads, Tasks. x, 37–39, 42, 51, 52, 54