Light Weight Threads and Their Communication Primitives

Tormod Hellen

December 13, 2014

Contents

Evaluation note

I want to be evaluated on both this report and the attached C code.

Part I
Introduction: The Indeterminism Problem

If you've ever written concurrent code, you might have noticed that you are pretty bad at writing that code. I certainly have. The nice thing about conventional code executing from top to bottom is that there aren't really any surprises. As my first programming teacher told me: if something goes wrong the bug is further up in the text file (accomodating code flow between functions etc.). This doesn't really work with concurrent code, you might have discovered. Concurrent code usually has the nasty feature of having indeterministic execution timing, presenting you with heisenbugs where even reproducing the bug is a challenge. One solution is to try to hold all the possible ways your threads might interact with each other in your head at once, but as the amount of threads doing things grow, this becomes unfeasible.

How to counter this onslaught on your sanity? One method is to keep the amount of threads working at once down, and using conventional semaphores this is the way you usually do it. But what if you want to use more threads, what then? Well, that's what most of this report is about. Some approaches hide and automatically recover from your faults, others make them more predictable so they're easier to find.

There are many models of concurrency awailable to the modern engineer. The classic semaphore and its relative - lock - are the most native to the way our processors look, with shared memory between thread hardware. Others, like the actor model, copy a distributed architecture with several computers, all with their own memory, processors and network connections. Others still, like channel-concurrency, are based on formal or mathematical theorems. These are the three main means of concurrent communication, though others have been introduced. We will look at some of them in this report.

Interestingly, parallel statements and synchronous execution do away with the indeterminism entirely, and are therefore handy when you can't be bothered with nondeterministic execution, though there are performance and flexibility costs to this.

1 Goals

The goals of this project is to:

1. Give a desciption of the paradigms relevant to safe concurrency, where safe is defined as lacking race conditions and making deadlocks easier to handle
2. Present the leading edge of implementations and other things that should influence a modern concurrency abstraction framework
3. Give an overview over how threads, channels etc. work in modern implementations
4. Propose a mechanism for safe concurrency in C
5. Prototype the solution mentioned

2 Problems, Complaints and Whining

Initially the idea was to make a lightweight threading library for C. Although this idea ended up sort-of-realized, a couple of limitations in C and POSIX presented themselves. For example, a POSIX thread's argument is the only way it can communicate and everything has to be crammed into this one struct. The result is a highly unpleasant nesting of structs, held together with pointers. Pointers, in turn, turn out to not be typechecked, and any significant refactoring resulted in unhelpful runtime errors. These problems, mutually reinforcing as they are, also reinforce the last problem: Mutexes as highest synchronization mechanism and the use of pointers to point to them. In the end, changing anything in the code became increasingly hard to reason about, making programming a nervous experience.

C11 has introduced a number of optional handy features, which would have made the code more portable, and generally better. Unfortunately, neither clang nor GCC have <stdatomic.h> and <thread.h> yet. The support for generics is also laclustre. C11 introduced a way to use C-macros to, at runtime, use the correct function for an arbitrary argument to a fictive function with generic argument. However, this requires the programmer to account for all the possible types people want to use in advance, and I assume a certain amount of work, at the very least, is involved if you want to have generic members of structs. Maybe using the lack of typechecking of pointers to structs would make this possible.

It also turned out to be hard to insulate the user from the complexities of the framework, though in retrospect an interface like that used by POSIX threads is probably possible. It's my fault, to be honest.

In short, working with a system language like C has its problems. For a funnier take, see [James Mickens: The Night Watch]

Part II
Background

3 Organization

This part is going to be relatively unorganized. There's no neat progression to concurrency techniques. Some compete and some are used for different things. I don't expect one of the approaches here to become completely dominant, but rather that a mixture of approaches will be used. While I try to keep the introductory material at towards the front and the more intricate material towards the back but some arbitrariness is unavoidable, I think.

4 Tl;DR

- *Semaphore*: C, C++, Java etc. implement this style of concurrency.
  - STM: Software Transactional Memory, or, for some architectures, simply Transactional Memory, is a kind of optimistic semaphore technique where changes are rolled back and attempted again in case of conflict. Coupled with language features that prevent the sharing of non-transactional memory between threads it is a convenient, but computationally costly technique.
- *Channel*: This is the CSP family and its derivatives. Go, Rust, Haskell and Occam have concurrency mechanisms based on channels. Focus on being deterministic and formally analyzable. Uses multi-sender, multi-receiver (first to read) synchronous channels.
- *Actor*: Erlang and Akka implement this model. Intuitively suited for making distributed systems, with a focus on fault tolerance rather than fault avoidance. Both implementations eschew sophisticated error recovery mechanisms and instead prefer crashing and rebooting errant processes, a philosophy called "let it crash".
- *Paralell statements*: If you have multiple actions that you know can be done in paralell without communicating with each other, then proceeding when all are done, then there's no reason you shouldn't. Rust and Akka offer this as futures, Occam has par blocks and LabView has paralell pipes. There's also parallel transformations of lists, which is particularly easy to do for the programmer. Often characterised as data-parallelism, as opposed to task-parallelism.
- Synchronous execution: Threads execute in synchrony, resulting in complete determinism. Esterel uses this.
- Lightweight threads that are less omputationally costly to deal with than OS threads, which in turn are less costly than processes. Details differ from implementation to implementation.

5 Not Included: Cray's Chapel

Most of what's covered in this report is pretty new, but at least somewhat mature. In order for programming languages to mature a lot of time is needed, and even the newest, not-out-of-beta-yet language here, Rust, is transitioning to being used in production right now. They also cover reasonably common scenarios: PC hardware, maybe with some distributed computing tossed in. Cray, a company making supercomputers, is cooperating with academics and other interested parties in making a new open source programming language for supercomputing: Chapel. Supercomputing has long been the domain of C/C++ and FORTRAN. Chapel is tasked with bringing supercomputer programming closer to the convenience of mainstream (Java, Python, Matlab etc.) while sacrificing no low-level control and providing as many parallel computation paradigms as possible. This could lead to exciting things in the future.

6 Limitations of Lightweight Threads

Rust is the newest and hottest language investigated in this project. It's developed by Mozilla as a replacement for C++. Rust *used to have* only lightweight threads[Rust heavyweight threads], but due to the many features desirable for threads in Rust (ability to call C code etc.) the lightweight threads ended up being as heavy to run as OS threads, and Rust's lightweight threads have been pushed to a library outside the standard library. This illustrates an important point: OS threads are not heavyweight just to irritate us, they are heavyweight because they have features lightweight threads don't. Tradeoffs are necessary. There is no free lunch. This is also true for the communication abstractions.

Let there be no mistake: In switching from OS threads with semaphores to any of the higher abstractions we sacrifice flexibility and usually performance for the sake of safety and ease of programming. That's the disclaimer. Now for the good part: The end result can still be more performant and do more complicated things because the safety and ease of use lets us program less conservatively. It's a bit like going from a low-level language to a high-level one.

7 Lightweight Threads and Communication

Lightweight threads and Communication mechanisms are different things not really intimately linked. It's quite possible to have one without the other. However, the motivation that lead to both is the same: In the multicore present, bug-free concurrent code must become easier to write. Since they share motivations, seeing them together is the norm. If your aim is to create a better way to write concurrent code, why only go half way, after all? And frankly, wanting lightweight threads so you can debug even more threads running with semaphores doesn't make sense to me. In fact that sounds like a mixture of masochism, code golf and performance art.

Lightweight threads have influenced the design of modern Communication mechanisms. Since having lots of lightweight threads and switching between them is cheap, blocking calls also become cheap. Consequently, both the actor- and CSP-model can afford to have their threads mostly sitting around waiting for messages.

8 The Nature of a Lightweight Thread

Process > OS Thread > Lightweight Thread OS threads are actually themselves lightweight when compared to another popular concurrency abstraction: processes. All processes run concurrently anyway, and moving information between them with inter-process communication (example: localhost) you suddenly have a heavyweight actor model: All that's old is new again, it seems. My point is that the quest for more efficient means of concurrency is hardly new, though it has acquired unusual urgency as single-core processor performance gains are slowing down.

What does a lightweight thread look like? Well, the point of a lightweight thread is to be light - it can't have all the things an OS thread does. A POSIX thread holds the following things in memory[llnl.gov pthreads]:

- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data such as the current execution stack.

Collectively, these things are called a thread context. A context switch is when a processor core pauses, switches its registers, points its stack pointer and program counter at the right memory adresses etc. and starts executing again. So lightweight threads, in order to be lightweight, have to sustain themselves on less than a thread context and/or have faster context switches, which are two goals with overlapping means. There are many ways from here:

- Go's goroutines have no signals, but they do have a stack. The stack starts small and, like a vector, can be expanded by allocating a bigger piece of memory somewhere else and deallocating the original stack space. By having a stack, goroutines are a bit like normal threads as they can be resumed after a pause. Go's goroutines are implemented like coroutines where inserting yield and resume is done by the compiler.
- Akka's actors have no stack. How do they work then? The Java (OS) thread runs the actor's receive function until completion and then the stack is thrown away as the Java thread moves on to another actor. Clever and simple, but it requires a different way of programming than we're used to. As a result of this, Akka offers 2.5 million actors per GB of heap memory.

9 The Conflict Between Fault Tolerance and Verification/Debugging

Formal verification is desireable in many safety-critical programs. Unfortunate side-effects of fault-tolerant systems, for example actor systems, is that they

1. make the program's behaviour more complex
2. hide faults from the developer at runtime.

Both are problems when we want to ensure correctness or debug the program at runtime. The first problem can be partly solved by using frameworks or languages where the added complexities can be assumed bug free, though you still need to superficially understand how they work. The second problem can be partly solved by using extensive logging. These techniques can make debugging such a complex system less hard than it would otherwise be, but the formal analysis of such a system can still be an unsolveable problem due to the amount of states the software can be in. Consider the queues of messages actors have.

Their content, even their types, are hard to predict at compile time. The sequence messages arrive in is nondeterministic and, Heisenberg-like, is changed if you try to observe it.

It is, in sum, unlikely that an implementation made using actor- or other fault-tolerant techniques can be verified directly, however using these models can help create systems that will resist non-software faults. Also, these tools can usually be used to emulate verifiable systems and although that's not as good as using a tool with built-in verifiable abstractions, it will usually be better than constructing your own using semaphores and heavy threads.

10 Immutability

If an object or value is immutable it means it can't be changed. Depending on whom you ask, this might mean that only the object itself need be unchangeable, or it could mean that the object and all the objects it references are unchangeable. To avoid the ambiguity, terms like "deeply immutable" and "invariant" can be used for the latter. Objects or values that are deeply immutable can be shared between threads without fear of adverse effects, which make this property very attractive. Functional languages and languages with functional influences like Haskell, Erlang, Scala, Clojure and Rust place heavy emphasis on immutability.

11 Coroutines: Single-thread Interleaved Execution

Coroutines are a superclass of subroutines, what we ordinarliy call functions, and are mostly written the same way. While a function is called and then returns, with a clear caller-callee relationship, coroutines can yield and receive execution as many times as desired during its lifetime. Functions, to contrast, "resume" at the beginning and yield at the end. Coroutines are so called because they do co-operative scheduling - yielding is entirely voluntary and predictable on the part of the coroutine. Coroutines were first done in Assembly by Melvin Conway in 1958, and it's easy to understand why - yielding in assembly can be done by simply jumping into another "coroutine". Actually, Assembly usually has no concept of functions, let alone coroutines, so you just jump to somewhere else in program memory.

11.1 Disadvantages

Coroutines have co-operative scheduling, which is cheap compared to pre-emptive but that means we're reliant on the programmer for parallelism. Any coroutine can hog procesing capacity for as long as it likes. Also, real threads don't share stack, so state held in stack somehow needs to be synchronised across threads.

11.2 Problems in C

I've decided to provide you with a basic example of the mechanisms used. Here we have two functions, co1 and co2, pass control between one another, saving each other's positions on the stack in global environment variables. The variables need to be global, because jumping to a function higher up on the stack will mess up any variable, even const pointers, held on the stack in that function. It's probably the same for all functions higher up on the stack, but I haven't tested. That gave me some weird bug behaviour that was pretty hard to figure out. It's worth pointing out that the behaviour of longjumping to a function that is not the caller of the current function is *undefined*; this is abuse of longjmp, but it is not unprecedented and does work. An alternative would be using Boost.context from the Boost library for C++, if that can be made to work.

```
1  #include <stdio.h>
2  #include <setjmp.h>
3
4  static jmp_buf environment1;
5  static jmp_buf environment2;
6
7  __attribute__ ((noreturn)) static void co2()
8  {
9    int VAR = 5;
10   printf("VAR = %d\n", VAR);
11
12   printf("In function co2 place 1\n");
13
14   int ret = setjmp(environment2);
15   if (ret == 0)
16   {
17     longjmp(environment1, 1);
18   }
19   printf("In function co2 place 2\n");
```

```
20    ret = setjmp(environment2);
21    if (ret == 0)
22    {
23      longjmp(environment1, 1);
24    }
25    printf("In function co2 place 3\n");
26    printf("VAR = %d\n", VAR);
27    longjmp(environment1, 1);
28  }
29
30  static void co1()
31  {
32    printf("In function co1 place 1\n");
33
34    int ret = setjmp(environment1);
35    if (ret == 0)
36    {
37      co2();
38    }
39    printf("In function co1 place 2\n");
40
41    ret = setjmp(environment1);
42    if (ret == 0)
43    {
44      longjmp(environment2, 1);
45    }
46    printf("In function co1 place 3\n");
47    ret = setjmp(environment1);
48    if (ret == 0)
49    {
50      longjmp(environment2, 1);
51    }
52    return;
53  }
54
55  int main()
56  {
57      printf("Hello World\n");
58      co1();
59      return 0;
60  }
```

The output is as follows:

```
1  Hello World
2  In function co1 place 1
3  VAR = 5
4  In function co2 place 1
5  In function co1 place 2
6  In function co2 place 2
7  In function co1 place 3
8  In function co2 place 3
9  VAR = 0
```

Interestingly, the integer variable VAR changes value without ever having been touched since creation in the source code. Not only can we not synchronise stack state between real threads, we can't even keep it within one real thread. As explained, coroutines can be done with jumps, but stack variables are not conserved. At least not without using costly system calls or complicated assembly voodoo, which most lightweight thread libraries for C seems to use. This means that the normal way of programming will have to be constrained a bit - all variables will have to be held on the heap or in global variables.

11.3 How it may be used to create general-purpose lightweight threads

Coroutines as hereto discussed might make some code more readable (especially consumer-producer relationships), but it does not help us create lightweight threads of the kind we want. But what if the coroutine didn't yield to another coroutine but to a scheduler? Indeed, this is how goroutines work in Go, although Go hides so much of the manual scheduling work that its authors decided to exchange c for g in the name to avoid confusion.

A possible way to use coroutines to achieve lightweight threads would be to have each lightweight thread be a function pointer and a struct held on heap where state can be saved. The function yields each time it awaits a message and when it is done. When a function yields, the executing thread goes back to the scheduler and starts or resumes another function. Messages are received through a special value in the struct.

12 Semaphores

Semaphores, mutexes and locks operate mostly the same. They are atomic variables that the threads can use to negotiate access to values they want to mutate. Unfortunately the negotiation has to be written by the programmer and this process is notoriously difficult. The solution, as with manual memory management, is to be very very careful. Concurrent threads usually do not persist during the entire program, but are created for specific purposes and ended as soon as possible to keep complexity as low as possible.

Simple example of the related mutex concept:

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #include <unistd.h>
7
8  #define ITERATIONS 1000000
9
10 pthread_mutex_t mutex;
11
12 static void *increment(void *arg)
13 {
14   long *counter = arg;
15   for(long i=0; i<ITERATIONS; i++)
16   {
17     pthread_mutex_lock(&mutex);
18     *counter = *counter+1;
19     pthread_mutex_unlock(&mutex);
20   }
21   return 0;
22 }
23
24 static void *decrement(void *arg)
25 {
26   long *counter = arg;
27   for(long i=0; i<ITERATIONS; i++)
28   {
29     pthread_mutex_lock(&mutex);
30     *counter = *counter-1;
31     pthread_mutex_unlock(&mutex);
32   }
33   return 0;
34 }
35
36 int main(void)
37 {
38   printf("Hello World\n");
39   long *counter = malloc(sizeof(long));
40   *counter = 0;
```

```
41    printf("in main before threads counter=%ld\n", *counter);
42
43
44    pthread_t *threads = malloc(2*sizeof(pthread_t));
45    pthread_mutex_t *mutex;
46
47    int rc;
48    rc = pthread_create(&threads[0], NULL, increment, counter);
49    assert(0 == rc);
50    rc = pthread_create(&threads[1], NULL, decrement, counter);
51    assert(0 == rc);
52
53    for (int i=0; i<2; ++i)
54    {
55      // block until thread i completes
56      rc = pthread_join(threads[i], NULL);
57      printf("Thread number %d is complete\n", i);
58      assert(0 == rc);
59    }
60
61
62    printf("in main after threads counter=%ld\n", *counter);
63    return 0;
64  }
```

If you're anything like me you don't like this very much.

13 Software Transactional Memory

Software Transactional Memory, STM for short, is a very convenient way to get communication between threads going. "You there!", one can imagine the salesman go, "You there programming with semaphores! Wouldn't it be great if you could just make the semaphores disappear? With some STM you can!" Gosh, sounds great, doesn't it? So how does STM work? Well, STM is, as the name implies, transactional. Hang on:

1. You lock, copy (to C1) and unlock the variable you want to perform a calculation on (O).
2. You perform the calculation
3. If the calculation involves writing to the original variable, then you write the result to a new copy (C2)
4. Now you lock the original variable (O), see if it is equal to C1. If O==C1, then the result from the calculation is still valid. If O!=C1 then you have to go back to step 1 (you never changed the original, so you simply delete C1 and C2).
5. If the calculation involved writing to the original variable (O), then you assign the second copy to the original (O=C2)
6. Now you have to unlock O again.

Of course it's done a bit more smartly than this in serious implementations, but it looks costly doesn't it? Unfortunately it is, but it's expected to improve a lot when we get hardware implementations. Well, actually we have hardware implementations, but not the sort that are targeted by mainstream compilers. A developer wishing to use Intel's (unfortunately buggy and off by default) TSX instructions will have to get dirty, probably with assembly, as these instructions were first available with the Haswell generation. If you're the sort whose processors come from Oracle or IBM, then your ecosystem might have had more time to nicely abstract these things away for you.

STM implementations do not have guarantees pleasing to the ear of a real-time engineer. Typically, operations will be done eventually, in arbitrary sequence. Unless something smart is done I don't see anything preventing starvation of a thread doing a particularly lengthy operation on an STM variable when others are doing short ones.

Whatever your langauge is, someone's probably made an STM implementation for it, but Clojure and Haskell are notable for their communities preferring them as their primary means of inter-thread communication.

Here's an example in Haskell (shamelessly taken from [Haskell STM Example]):

```
1  module Main where
2  import Control.Monad
3  import Control.Concurrent
```

```
import Control.Concurrent.STM

main = do shared <- atomically $ newTVar 0
          before <- atomRead shared
          putStrLn $ "Before: " ++ show before
          forkIO $ 25 `timesDo` (dispVar shared >> milliSleep 20)
          forkIO $ 10 `timesDo` (appV ((+) 2) shared >> milliSleep 50)
          forkIO $ 20 `timesDo` (appV pred shared >> milliSleep 25)
          milliSleep 800
          after <- atomRead shared
          putStrLn $ "After: " ++ show after
 where timesDo = replicateM_
       milliSleep = threadDelay . (*) 1000

atomRead = atomically . readTVar
dispVar x = atomRead x >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

This program runs for 800 milliseconds and has three threads. A transactional integer is made. Thread A prints the integer every 20 milliseconds, thread B adds two to the integer every 50 milliseconds and thread C subtracts one from the integer every 25 milliseconds. After 800 milliseconds the integer is 0 again. At least that's what I think it does.

Here's a Clojure example illustrating STM, and this one I've actually written myself:

```
(def x (ref 1))

(defn increment [i]
  (if (> i 0)
    (
      (dosync
        (alter x inc)
      )
      (Thread/sleep 1)
      (increment (- i 1))
    )
  )
)

(defn decrement [i]
  (if (> i 0)
    (
      (dosync
        (alter x dec)
      )
      (Thread/sleep 1)
      (decrement (- i 1))
    )
  )
)

(defn printref [i]
  (if (> i 0)
    (
      (dosync
        (println (format "in printref %d" @x))
      )
      (Thread/sleep 1)
      (printref (- i 1))
    )
  )
)

```

```
(future
  (increment 10)
)

(future
  (printref 15)
)

(future
  (decrement 10)
)
```

Handling large STM variables My explanation of STM above has a problem in it: What if the STM variable is really really large? In the real world, STM algorithms do not copy, but use logs of the specific thing they read and write: A read-set and a write-set. Copies or records will necessarily have to be made, but if you only read and write to element 23438 in the array then whether the other variables have been changed during operation doesn't matter. Not only does this improve memory usage, but other threads can work concurrently on other parts of the array without much conflict.

14 Channels

A channel is a communication primitive between threads. A channel, as used in Go, Rust and CSP, can have several senders and recipients. Each message can typically only be received once, so multiple receivers is not a broadcast mechanism, but rather a work division mechanism.

Channels as a method for communication are analogous to the concurrency in CSP, and convenient to verify and do formal analysis upon. The most known languages impementing them are Rust and Go.

Rust uses channels, futures and immutable copies. Channels are multi-sender, multi-receiver affairs, CSP-style:

```
fn main()
{
        let (tx, rx) = channel();
        let txclone = tx.clone();

        //proc denotes an anonymous function with function body being the stuff behind "proc
            ()"
        spawn(proc() tx.send("message"));
        spawn(proc() txclone.send("another message"));
        spawn(proc() println!("{:s}, {:s}" ,rx.recv(), rx.recv()) );
}
```

When sending a message over a channel in Rust, the data in the message will be duplicated. To improve efficiency, multiple tasks can share an object as if on a shared heap, provided that the object is immutable. You might have noticed that I cloned the transmitter. Why? Rust has no garbage collection, but still automatic memory management, which means that the moment at which an object can be deallocated from the heap need to be apparent at compile time. Therefore, two threads can't have references to the same heap object, in this case the transmitter. In Rust parlance, a thread owns the object and two threads can't own the same object.

Go takes a more imperative approach, and accidental data sharing is easier to do. Here we're doing the exact same thing in Go.

```
package main

import "fmt"

func main() {
  messages := make(chan string)
  go func() { messages <- "message" }()
  go func() { messages <- "another message" }()
  msg := <-messages
  msg2 := <-messages
  fmt.Println(msg + ", " + msg2)
}
```

Since Go has garbage collection, there's no need for cloning.

15 Actors

There are many implementations of the Actor model, but only two that I know of that matter: Erlang and Akka.

15.1 Akka on Scala

Scala, a language, originally had its own message passing library, but this has been deprecated in favor of the Akka library, the essentials of which is included as standard in all recent distributions of Scala. The Akka actor library is an improvement on the Scala actor library which in turn is based on the Erlang actor model. Akka is also available to Java programmers.

It's used like this:

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.ActorSystem
4  import akka.actor.Props
5  import scala.sys
6  import java.lang.Thread
7  //Type declarations look like this: "nameOfObject:NameOfClass" with or without space.
8  class DemoActor(printstr:String) extends Actor
9  {
10   def receive =
11   {
12     case other: ActorRef =>
13     {
14       println(printstr)
15       other ! self
16     }
17     case _       => println("unknown message")
18   }
19 }
20
21 object Main extends App
22 {
23   val system = ActorSystem("DemoSystem")
24   val aActor = system.actorOf(Props(classOf[DemoActor], "a"))
25   val bActor = system.actorOf(Props(classOf[DemoActor], "b"))
26   aActor ! bActor
27   Thread.sleep(3)    //let actors run for 3 milliseconds
28   scala.sys.exit()
29 }
```

The output of this is a sequence of type "a b a b a b" with each letter on a line of its own. The precise number of a's and b's printed to the console varies, but in general you don't have to scroll if your console is maximised. In another example, creating a million Akka actors in Scala takes about 11 to 12 seconds on the lab computers.

Synchronous messaging is neither enforced nor recommended by the Akka developers, but you can use bounded and blocking mailboxes. A bounded and blocking mailbox will block the sender of a message if the receiver's message queue is full but the minimum capacity is 1, so true synchrony is not achieved. To achieve true synchrony you need to manually use Await for each time you send a message. The reason it is not recommended is that new classes of bugs surface, something that might be advantageous to a life-critical application programmer, but not to the vast majority of programmers.

Akka actors interface neatly with other concurrency abstractions in Akka. More in 19.

Akka actors have many interesting properties. For example, actor references are network aware, so one of our DemoActors could receive a message with an ActorRef to an actor on another continent and it would still respond correctly using the Akka Remote Protocol over TCP. Akka, unlike Erlang, enforces supervision in a similar manner as offered by Erlang's OTP.

Akka actor reference examples[Akka actorRefs]:

```
"akka://my-sys/user/service-a/worker1"                   // purely local
"akka.tcp://my-sys@host.example.com:5678/user/service-b" // remote
```

Akka offers configurable message dispatchers, offering control over the number of underlying threads and number of messages an actor can process before the underlying thread jumps to the next actor. The default dispatcher uses a fork-join method to run the actors, but you can also use thread-pool or your own custom dispatcher.

Akka actors send messages as references by default, and unfortunately neither Java nor Scala enforces immutability of the underlying objects. To work around this, Akka provides optional deep copying of all messages.

15.2 Erlang

Both Erlang and OTP were developed at Ericsson, and today are used for critical infrastructure worth multiples of billions, like WhatsApp. Insert snarky remark about capitalism here. OTP stands for Open Telephony Platform, an archaic name not reflecting current use.

For Erlang, the actor model is the beginning and the end. Erlang is structured around them; each actor has its own heap, which means that everything sent is physically copied. Erlang is also an eraly functional programming language, and has no loops. State is held as recursion arguments as the actors main function calls itself at the end of processing a message (or don't, if you want to terminate the actor).

Erlang can use something called selective receives to prioritize certain kinds of messages.

```
1  -module(kitchen).
2  -compile(export_all).
3
4  fridge2(FoodList) ->
5          receive
6                  {From, {store, Food}} ->
7                          From ! {self(), ok},
8                          fridge2([Food|FoodList]);
9                  {From, {take, Food}} ->
10                         case lists:member(Food, FoodList) of
11                                 true ->
12                                         From ! {self(), {ok, Food}},
13                                         fridge2(lists:delete(Food, FoodList));
14                                 false ->
15                                         From ! {self(), not_found},
16                                         fridge2(FoodList)
17                                 end;
18                 terminate ->
19                         ok
20         end.
21
22 store(Pid, Food) ->
23         Pid ! {self(), {store, Food}},
24         receive
25                 {Pid, Msg} -> Msg
26         end.
27
28 take(Pid, Food) ->
29         Pid ! {self(), {take, Food}},
30         receive
31                 {Pid, Msg} -> Msg
32         end.
```

16 Synchronous Execution

I'v'e only found one language that does this, and that is:

16.1 Esterel

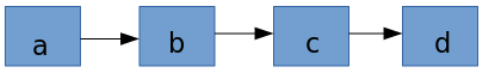
Esterel may be the coolest implementation discussed here. It has very limited expressive power and is not freely available, but it has *exciting* properties: Threads in Esterel *execute in synchronous time*. Put differently, Esterel has a global clock that threads march in lockstep to! A thread has a loop and one cycle of that loop is ompleted per tick of the global clock. Esterel is completely

deterministic; neither dynamic memory nor spawning of processes are supported. Signals are broadcast, and threads await and send signals. A signal is either precent in a cycle or it is not - the time of broadcast is abstracted away. Programs in Esterel are deterministic finite state machines.

Esterel is not a real-time language but since it is completely deterministic, simple testing should suffice for investigating time characteristics.

Esterel is hard to get hold of as a private individual, is a niche language and therefore we won't explore it in more detail.

17 Pipelining



If you have a constantly incoming stream of data coming in to the leftmost block in the picture, you have an opportunity to pipeline. This concept should be familiar to you if you've ever studied processor design, where pipelining means you piece up computations into small parts that can be done after each other. If you've got a lot of similar computations coming in after each other then you can do each step in parallel. Think of how an assembly line doesn't necessarily make any given car faster than manual assembly, but since each step is done in parallel you get figures like "3 cars per minute".

That's the concept here as well. If you place each step on its own processor core, you have an assembly line for computation. LabView uses this[LabView parallelism].

18 Parallel Statements

Parallel statements are very attractive, because they can make our code parallel without introducing indeterminism. The catch is that the statements to be parallelised cannot communicate with each other. Keep in mind, though, that the overhead of managing threads might outweigh the benefits of parallelism if the tasks you want done are sufficiently simple and few. Some tools and languages, like Occam, treat your parallel instructions as guidelines where this tradeoff is outsourced to the compiler. Others, like Scala, does exactly what you tell them, and there you should not use parallel statements unless your workload is heavy enough.

18.1 Occam's PAR (without messages)

Unfortunately it proved troublesome to get one of the Occam compilers going - working with dead languages can be frustrating. Fortunately, many of the facilities offered in Occam can be replicated in other languages, for example by using Futures and Actors.

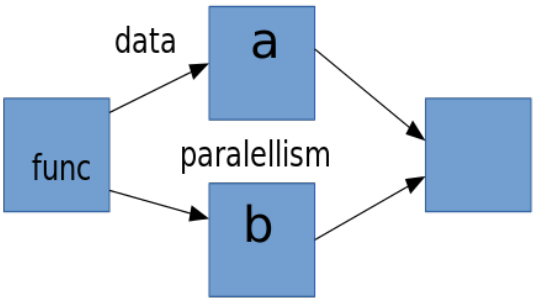
For example this (admittedly completely paper-programmed) Occam snippet...

```
#INCLUDE "hostio.inc"
#USE "hostio.lib"
PROC Main(CHAN OF SP fs, ts)
        SEQ
                PAR
                        x = function(1)
                        y = function(2)
                z = x+y
                so.write.string.int(fs, ts, z, 0)
                so.exit(fs,ts,sps.success)
:
```

...has x and y be computed in parallel and then combined into z. The same thingcan be done with futures, and in 18.3 we have a corresponding example in Scala. Of course a function inside a PAR can send messages to another inside the same PAR, while it is not unthinkable that other parallel statement constructs in other languages can do the same, it is usually not the intended use.

18.2 LabView's parallel arrows

LabView is a proprietary graphical programming language, and as such a bit different from the other languages discussed here. LabView uses parallel statements.



I had problems getting hold of labview screnshots with an appropriate license. Anyway, the basic concept can be easily eplained even with limited fidelity. Data flows along the arrows and the blocks hold functions that outputs new data. As the dataflows branch, you can see the potential for paralellism. Block a and block b are completely independent and LabView can (and will)[LabView parallelism] therefore execute them in paralell. In the end it all gets fed into a sink, for example a controller or a print to memory or screen.

An Occam programmer may express the sequence pictured above like this:

```
SEQ
        do stuff
        PAR
                a
                b
        do more stuff
```

18.3 Futures

A future is an [insert value here]-value. It represents the result of a computation that may or may not have finished yet. It's a feature for requesting a computation and getting the results (including side effects) later. Here's a Rust example of the use of a Future:

```
use std::sync::Future;

fn main()
{
        fn fib(n: u64) -> u64
        {
                // lengthy computation returning an uint
                12586269025
        }
        let mut delayed_fib = Future::spawn(proc() fib(50));
        println!("fib(50) = {}", delayed_fib.get());
}
```

This Scala example does the same as the Occam one in 18.1, just with futures:

```
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.language.postfixOps

object occ extends App
{
        def function(i: Int) = i*i
        val x = Future {function(1)}
        val y = Future {function(2)}

        val z = for
        {
                xc <- x
                yc <- y
```

```
        } yield xc + yc

        val zval = Await.result(z, 0 nanos)
        println(zval)
}
```

18.4 Parallel List Transformation

Many languages support doing calculations on elements in a list in parallel. This is a very simple way to achieve parallelism; no undeterminism or other concerns. Here's how it's done in Scala:

```
val a = List(1,2,3,4,5,6,7,8,9)
val b = a.par.map(x=>2*x).toList
//b is List(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

While Haskell does the same like this:

```
import Control.Parallel.Strategies
let a = [1,2,3,4,5,6,7,8,9]
let b = parMap rpar (*2) a
//b is [2,4,6,8,10,12,14,16,18]
```

It is noteworthy that rpar makes this evaluation lazy, a sort of future if you will, in that the computation returns immediately and sequential computing afterwards will only wait when b is needed. Neat.

This type of concurrency is actually not too hard to achieve in C. Yours truly have made an admittedly array[int]->array[int] attempt:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>

struct ThreadArgument
{
  int numIterations;
  int *iterationsStart;
  int (*function)(int);
};

void *singlethreaditerator(void *arg) //struct ThreadArgument* argument
{
  printf("Entered singlethreaditerator\n");
  struct ThreadArgument argument = *(struct ThreadArgument*)arg;
  for(int i=0; i<argument.numIterations; i++)
  {
    *(argument.iterationsStart+i) = argument.function(*(argument.iterationsStart+i));
  }
  return NULL;
}

void intparmap(int array[], int arrayLength, int (*function)(int) )
{
  int numofcpus = sysconf(_SC_NPROCESSORS_ONLN);
  int numofthreads = numofcpus;
  pthread_t threads[numofthreads];
  int thread_args[numofthreads];
  int tasksPerThread = arrayLength/numofthreads;
  printf("tasksPerThread = %d\n", tasksPerThread);
  struct ThreadArgument threadsarguments[numofthreads];

```

```
35    for (int i=0; i<(numofthreads-1); i++) //handle first n-1 threads
36    {
37      printf("intparmap\n");
38      threadsarguments[i].iterationsStart = &array[i*tasksPerThread];
39      threadsarguments[i].numIterations = tasksPerThread;
40      printf("intparmap\n");
41      threadsarguments[i].function = function;
42    }
43
44    threadsarguments[numofthreads-1].iterationsStart = &array[(numofthreads-1)*tasksPerThread
          ];
45    threadsarguments[numofthreads-1].numIterations = arrayLength-(numofthreads-1)*
          tasksPerThread;
46    threadsarguments[numofthreads-1].function = function;
47
48    int rc;
49    for (int i=0; i<(numofthreads); i++) //start threads
50    {
51      printf("Creating thread %d\n", i);
52      rc = pthread_create(&threads[i], NULL, singlethreaditerator, (void *) &threadsarguments[
            i]);
53      assert(0 == rc);
54    }
55
56    //wait for threads
57    for (int i=0; i<numofthreads; ++i)
58    {
59      // block until thread i completes
60      rc = pthread_join(threads[i], NULL);
61      printf("Thread %d is complete\n", i);
62      assert(0 == rc);
63    }
64  }
65
66  int square(int x)
67  {
68    return x*x;
69  }
70
71  int main()
72  {
73    printf("Hello World\n");
74    int num = 1000;
75
76    int array[num];
77    for(int i=0; i<num; i++)
78    {
79      array[i] = i;
80    }
81    intparmap(array, num, square);
82
83    for(int i=0; i<num; i++)
84    {
85      printf("%d\n" , array[i]);
86    }
87
88  }
```

This might not be the best way to do it, but it is fairly transparent to the end user. The code here transforms any array of integers by changing each element with any int->int function. Notice that since there's no need to share information among

threads, there's no need for mutexes, semaphores or other such mechanisms.

Scala's parallel statements are actually classes with parallel methods, so-called parallel collections. There are many ways these can be used and you can map, fold, filter and foreach on them in parallel. As all of these functions require input functions to work, the types of functions you use are important. The docs warn:

- Side-effecting operations can lead to non-determinism
- Non-associative operations lead to non-determinism

Having experienced something like this myself my advice would be to not have references to parallel collections, but rather immediately transform them to sequential ones, or at least not pass parallel collections between functions. It's easy to forget that the collections you are working with are not deterministic for certain operations. Languages like Haskell, where side-effects are more controlled, have an advantage here, as side-effecting operations are discouraged.

Hardware implementations There are some hardware implementations of parallel statements, like SIMD and OpenCL. SIMD (Single Instruction, Multiple Data) is a kind of processor instruction that allows a processor to do exactly what the name states very efficiently. On x86 platforms AVX (Advanced Vector eXtensions) is the name of the instructions for doing this.

In the same vein, GPUs are very good at taking a lot of floating point numbers and doing the same operation on all of them. Given that a modern AMD GPU has close to 3000 processing units and runs at 1GHz it's given that certain operations are going to be faster on a GPU than on a CPU. This has had a huge impact on many fields, for example in Artificial Intelligence it was believed that neural networks was inferior to many other techniques. Turns out that when you train the network on a GPU you can afford bigger neural networks and this improves performance so much that lately this has become the best AI technique for many fields. If you've heard some hype about "deep learning", that's basically just big neural networks made possible by GPUs and some algorithmic insights that I don't know particularly much about. There are two ways to interface with GPUs: OpenCL and CUDA, of which CUDA is only for Nvidia cards.

19 Actors, Futures and STM, Together.

In Akka, futures are used by default to avoid blocking when waiting for a response, effectively allowing an actor to process several messages concurrently, if it needs to. This dramatically increases the complexity required to produce a deadlock, and does away with the classical "A waits on B and B waits on A" example entirely. For instance this program...

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.ActorSystem
4  import akka.actor.Props
5  import akka.agent.Agent
6  import scala.sys
7  import java.lang.Thread
8  import scala.collection.mutable.ArrayBuffer
9  import scala.util.Random
10 import akka.util.Timeout
11 import scala.concurrent._
12 import scala.concurrent.ExecutionContext.Implicits.global
13 import scala.language.postfixOps
14 import akka.pattern.ask
15 import scala.util.{Failure, Success}
16
17 class FutureSTMActor(printstr:String, agent:Agent[String]) extends Actor {
18   var othermessagesanswered:Int = 0
19   def receive =
20   {
21     case other: ActorRef =>
22     {
23       val response = other.ask("please reply")(50000) //the 50000 here is the timeout. It's
             required and I just chose a very large value
24       response onComplete                                                    //executed
             when response arrives or timeout
25       {
26         case Success(result) => println("success: " + result); println(printstr + ": " + "
             STM Int says main thread in iteration: " + agent.get)
```

```
27          case Failure(failure) => println(failure)
28        }
29        othermessagesanswered += 1
30        println(printstr + ": " + othermessagesanswered.toString())
31      }
32      case "please reply" =>
33      {
34        sender() ! "this is a reply"
35      }
36      case s: String =>
37      {
38        println("got string: " + s)
39      }
40    }
41  }
42
43  object Main extends App
44  {
45          val agent = Agent("0"*10)
46          val system = ActorSystem("DemoSystem")
47      val aActor = system.actorOf(Props(classOf[FutureSTMActor], "a", agent))
48      val bActor = system.actorOf(Props(classOf[FutureSTMActor], "b", agent))
49      for (x <- 0 to 10)
50      {
51        println("m: " + x.toString)
52        agent send ((x.toString) * 10)
53        println("m: " + agent.get.toString)
54        Future{ aActor ! bActor }
55        Future{ bActor ! aActor }
56        if(x%1==0) { Thread.sleep(2) }
57      }
58      Thread.sleep(1000)
59      scala.sys.exit()
60  }
```

... will not deadlock. Moreover, the two actors will not even be in synchrony. Since they wait for each other with futures, they're not really waiting at all!

Now, you think we may be covered, but the problem with messages is one of synchronization. How do you make sure that an entire system of actors all have the same value, for example money in your bank account? One way to do it is to use STM, as we here do, synchronizing the current iteration number in the main thread's loop across actors using Agents, which are inspired by Clojure's STM Agents. As expected, the STM Agent is synchronized through the entire run, even though nothing else among the agents are.

An interesting problem occured when I tested this. Sometimes the number in the shared string would go downwards. Why? Turns out sometimes the transactions are enacted out of order, and the STM variable is set to a late value after a newer one. This isn't a feature that's universal to STM, but rather this particular implementation; usually updates from a single thread arrive in-order. A way to get around this is to increment the value instead of assigning to it, like so:

```
1  agent send (_ + 1)
2  //or, if you want to be consistent in your anonymous function syntax:
3  agent send (x=>x+1)
```

This is an important reminder that even STM does not let you write concurrent code as if it was single-threaded.

20 Scheduling Lightweight Threads

If you are goin to run N lightweight threads on M OS threads, you'll have to spread the tasks over, or dispatch them to, the OS threads somehow. What I'm saying is that you have to write a scheduler. As someone who has done that while writing this: Oy vey, woe be you. It's not a particularly easy task, but some smart people have thought about it before you and I have:

20.1 Thread-pool dispatching

The tasks to be run are divided into groups of assumed equal load and given to the real threads for execution. This method is divided into first a divide and then a conquer phase. If one group of tasks turns out to take longer than another the real thread with the easier task group is left idling.

20.2 Fork-join dispatching

Juicy stuff. This is what Go and Akka uses by default, and with good reason. In actor or channel concurrency most lightweight threads sit around waiting for messages at any given time. That's fine as long as the lightweight threads doing work are spread out roughly equally over the OS threads. But what if they aren't? What if all the lightweight threads doing work at any given time are on a single OS thread? Well, then you've written a multithreaded program where a only one thread at a time does any work. Fortunately, this dispatching method fixes the problem when applied correctly. So read on.

Fork-join is similar to thread-pool, but uses something called work stealing. With work stealing, a thread that finishes its tasks early can "steal" tasks from a thread that's still busy.To avoid overworking the scheduling problem tasks are grouped in groups, which makes the following thread a bit hard to understand. What happens is that the tasks are grouped in a single group and pushed on a stack. Each real thread can then pop a group of tasks and either split the group and push the resulting groups or execute the group. In general, real threads will only execute very small groups, splitting groups instead if they're too large. Task groups should be considered small enough when the expected overhead of worrying about their size is larger than the expected cost of a single group being too large. The benefit of this method over thread-pool is that the divide and conquer phases are interleaved. There are a lot more task groups than threads, so a misjudgment of the complexity of a single task group should have smaller consequence.

The "work stealing" term comes from an alternative implementation where the N real threads split the tasks into N task groups which each real thread finally splits and places on its own work stack. Once a real thread is done with its own task groups it will try to steal task groups from other real thread's stacks. If there's no work to find there it will look to an input queue of work common to all N real threads.

The first variant is handy if the tasks are messages to be processed, but if the tasks are the lightweight threads then you need the work stealing from the second method.

21 Conclusion

The popular approaches taken to concurrency in modern languages can be roughly divided into three: The actor model, channels and software transactional memory. The three approaches come from different mindsets and priorities. Users of the actor model often use it to create distributed, parallel, fault-tolerant systems. Users of CSP often want lower-level concurrency mechanisms that are easier to reason about and more predictable. STM and parallel statements work neatly with both paradigms.

The differences between CSP and actor model

- CSP threads are anonymous, actors are named.
- CSP communication is synchronous, actor communication is asynchronous.
- CSP communication is multi-sender, multi-receiver, actor communication is one-to-one.

Part III
Proposal

A recurring theme in discussing these topics online seems to be a need for predictability. POSIX threads are not predictable, but are or may be perceived as more predictable than lightweight threads with lots of stuff happening underneath in a non-obvious manner.

I propose an implementation that is as simple and deterministic as possible, providing guarantees like "all threads get to run a decent amout". Exposed functionality should be something like:

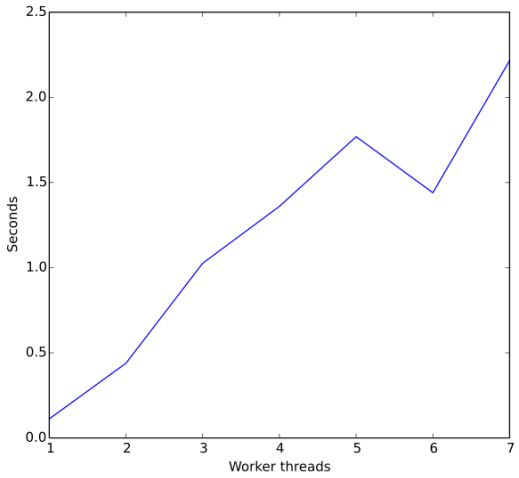
1. Parmap, a very simple mapping of one array to another of type $arrayA[x] = f(arrayB[x])$ for all elements x. Executed in parallel.
2. Futures, parallel computation
3. Lightweight threads with channels.

Part IV
Prototype

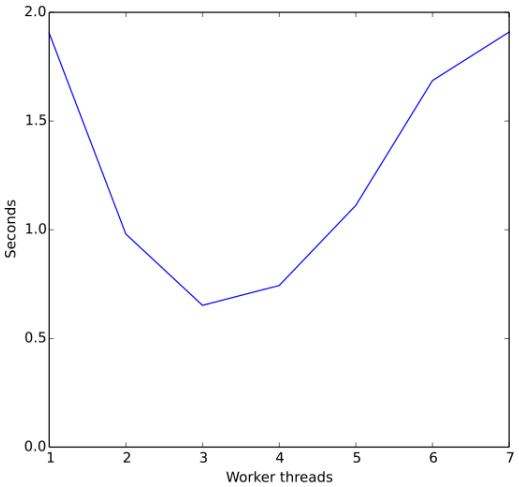
A prototype for parmap can be seen in 18.4. We'll now look at the prototype for lightweight threads. I haven't made a prototype for futures.

22 Scheduler

Of paramount importance in implementing lightweight threads is the scheduler. Both Go and Akka use a work-stealing scheduler and this seems to be the industry standard. Because it's simple and I'm not very good at C programming I made a simpler scheduler in which all worker threads compete for all lightweight threads all the time. For lightweights that don't do much the result is catastrophic as worker threads compete for mutexes:



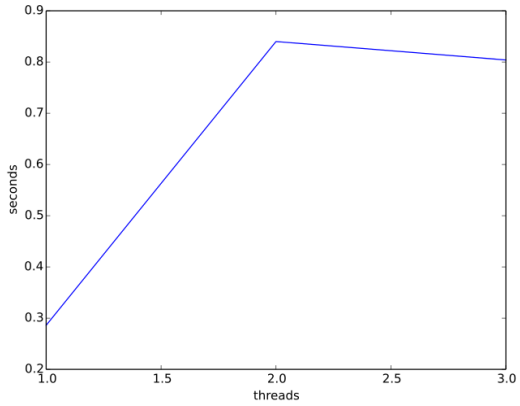
For lightweight threads that do a fair amount of work the result is a lot better:



Note that the time given on the y axis can not be compared across graphs, as the amount of repetitions is different. There are three lightweight threads, two of which sends messages to each other. That's why we speedup when moving from two to three workers is so lacklustre and performance only improves until we have three threads. The perfect amount of worker threads seems best determined by the programmer, who knows how the light threads will communicate, and thus what the real amount of paralellism is. That said, thread-pooling or work-stealing schedulers deal close to optimally with surplus threads.

An important point can be observed from this: You can make your code slower by throwing more threads at it! Even a good scheduler still has a lot of overhead and that overhead may take more time than the actual work, if that amount of work is small enough.

Smarter scheduling with thread-pool dispatching This is the final scheduler with very light load on the lightweight threads:



Again, I wouldn't compare seconds across graphs, because I have been very incosistent about number of iterations etc. Unfortunately, the final scheduler is unable to cope with more than one OS thread per lightweight thread, so that's why the threads don't go to seven here. Fortunately that situation makes absolutely no sense for thread-pool schedulers, as the superfluous threads would have nothing to do. In the discussion about fork-join dispatching I mentioned that it solved the problem of bad labor division among the OS threads. Here we can see such a problem play out as time goes down from two threads to three. In lightweight threads that do almost no work the majority of the time is spent in the scheduler. It's therefore better to just have as few threads as possible to make the scheduling go as fast as possible. The only reason I can see for the performance going up as we move from two to three is that the lightweight threads are badly handed out. If you're looking at the source right now, I'm thinking the grouping is [[foo], [baz, lol]] when a better grouping would be [[foo, baz], [lol]] as lol is independent of foo and baz and the latter two cannot run at the same time. With a work-stealing scheduler this would have solved itself pretty quickly.

From two to three threads we also see mutex contention being a much lesser problem than earlier. It's basically gone, in fact, as this is the goal of thread-pool scheduling.

23 The Structure of the Prototype. Or, a brief Documentation

All debug printouts left intact so you can see what's going on more easily.

The prototype has 3 coroutines that prints messages and otherwise yield, resume etc. Their functionality is held in the functions printfoo, printbaz and printlol. They all use a Duff's Device-inspired switch-case to handle yield and resume.

- printfoo prints foo and sends messages to printbaz
- printbaz prints baz and receives messages from foo
- printlol prints lol and goes in a loop, randomly yielding.

The coroutines have 7 functions to handle bookkeeping when yielding. These are retsend, retrecv, rettryrecv, retcont, retloop, retfin and retsleep. All, like retcont, do continuation bookkeeping, except retfin, which doesn't need to.

- retsend yields and instructs the scheduler to place a specified message on the specified channel. Scheduler resumes coroutine when message sent.
- retrecv yields and instructs the scheduler to resume when a message has been received on the specified channel; blocking receive.
- rettryrecv yields and instructs the scheduler to resume the coroutine, receiving a message on the specified channel if one is available; non-blocking receive.
- retcont yields and instructs the scheduler to resume at the specified continuation.
- retloop yields and instructs the scheduler to resume at the start of the function the coroutine is executing.
- retfin yields and instructs the scheduler to never resume.
- retsleep yields and instructs the scheduler to only resume after a specified amount of time.

rettryrecv and retsleep are untested but may work. They're mostly there for illustration purposes.

There are five structs:

- Datastruct holds the data the routines do work on. One per coroutine.

- Comstruct holds communication information; what channel the thread is communicating over and the message it is sending or receiving. One per coroutine.
- Sysstruct holds instructions for the scheduler, like whether the coroutine is sending, receiving or is ready for continuation and the place it is ready for continuation. One per coroutine.
- ThreadArgument holds the POSIX thread creation arguments. It houses mutexes and pointers to all other structs as well as the threadid and the amount of coroutines the thread is responsible for. One per POSIX thread.
- Channel holds messages and state of a channel. One per channel.

What remains are three functions: scheduler, coroutines and main.

- scheduler is the scheduler. It's a huge mess, but then again, it's a *scheduler*. The POSIX thread returns to this function when a coroutine yields. It then determines what to do next. Passing messages over channels is also done here in order to keep the mutexes out of user functions. The grand theme of what happens in the scheduler is that the OS thread goes over its coroutines in a loop, treating them according to their waitstate. If a thread is seen as hogging or blocking shared resources it will usleep a bit so as to not do exactly that.
- coroutines stuff away the work of spawning the POSIX threads and everything related. It builds the ThreadArguments and creates the mutexes.
- main is a mix of what the user of the prototype should do and what a user should definitely not have to do but has to do anyway.

Nowhere in the prototype is memory ever deallocated. Since memory isn't being allocated over the runtime of a program I don't see this as a serious problem.

References

[llnl.gov pthreads] https://computing.llnl.gov/tutorials/pthreads/

[Haskell STM Example] https://www.haskell.org/haskellwiki/Simple_STM_example

[James Mickens: The Night Watch] http://research.microsoft.com/en-us/people/mickens/thenightwatch.pdf

[Akka actorRefs] http://doc.akka.io/docs/akka/snapshot/general/addressing.html

[Scala parcollections] http://docs.scala-lang.org/overviews/parallel-collections/overview.html

[Rust heavyweight threads] https://mail.mozilla.org/pipermail/rust-dev/2013-December/007565.html

[LabView parallelism] Labhttp://www.ni.com/white-paper/6099/en/

Index