Light Weight Threads

Tormod Hellen

October 31, 2014

Contents

Part I
Goals

The goals of this project is to:

1. Give a desciption of the paradigms relevant to safe concurrency
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. are implemented in modern implementations
4. Propose a solution for concurrency in C
5. Prototype the solution mentioned

Part II
Background

1 Introduction

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 concurrency, though others have been introduced. We will look at some of them here.

1.1 Tl;DR

- *Semaphore*: C, C++, Java etc. implement this style of concurrency.
- *Channel*: This is the CSP family and its derivatives. Go, Rust 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”.

- Pipelining: LabView uses this. Each thread has one or more pipelines to and from other threads. Uses one-way, single sender, single receiver message passing called pipes.
- Synchronous execution: Threads execute in synchrony, resulting in complete determinism. Esterel uses this.

2 CSP Theory

The Communicating Sequential Processes theory describes a way for parallel computation to be done in which the traditional shared-memory errors like race conditions are not possible. CSP introduces processes and events.

3 Scala with Akka

> "We believe that writing correct concurrent, fault-tolerant and scalable applications is too hard. Most of the time it's because we are using the wrong tools and the wrong level of abstraction. Akka is here to change that. Using the Actor Model we raise the abstraction level and provide a better platform to build scalable, resilient and responsive applications—see the Reactive Manifesto for more details. For fault-tolerance we adopt the "let it crash" model which the telecom industry has used with great success to build applications that self-heal and systems that never stop. Actors also provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications." - The Akka documentation

Scala originally had its own message passing library, but this has been deprecated in favor of the Akka library, 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.

3.1 Use

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.

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 scala.sys
6  import java.lang.Thread
7  import scala.collection.mutable.ArrayBuffer
8  import scala.util.Random
9  import akka.util.Timeout
10 import scala.concurrent._
11 import scala.concurrent.ExecutionContext.Implicits.global
12 import scala.language.postfixOps
13 import akka.pattern.ask import scala.util.{Failure,
       Success}
14
15 //Type declarations look like this: "nameOfObject:
       NameOfClass" with or without space.
16 class LockingActor(printstr:String) extends Actor
17 {
18         var othermessagesanswered:Int = 0
19         def receive =
20         {
21                 case other: ActorRef =>
22                 {
23                         val response = other.ask("please␣
                            reply")(50000)     //ask for
                            response with timeout of 50
                            seconds
24                         response onComplete
25                         {
26                                 case Success(result) =>
                                    println("success:␣" +
                                    result)
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         }
37 }
38
39 object Main extends App
40 {
```

```
41      val system = ActorSystem("DemoSystem")
42          val aActor = system.actorOf(Props(classOf[
                LockingActor], "a"))
43          val bActor = system.actorOf(Props(classOf[
                LockingActor], "b"))
44          for (x <- 0 to 100000)
45          {
46                  println(x)
47                  Future{ aActor ! bActor }
48                  Future{ bActor ! aActor }
49          }
50  }
```

... 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!

3.2 Implementation

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 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.

4 Occam

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 using other for example Futures and Actors in other languages.

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

```
1  #INCLUDE "hostio.inc"
2  #USE "hostio.lib"
3  PROC Main(CHAN OF SP fs, ts)
4          SEQ
5                  PAR
```

```
6                               x = function(1)
7                               y = function(2)
8                       z = x+y
9                       so.write.string.int(fs, ts, z, 0)
10                      so.exit(fs,ts,sps.success)
11  :
```

... can be replicated in Scala like this using futures:

```
1  import scala.concurrent._
2  import scala.concurrent.ExecutionContext.Implicits.global
3  import scala.concurrent.duration._
4  import scala.language.postfixOps
5
6  object occ extends App
7  {
8          def function(i: Int) = i*i
9          val x = Future {function(1)}
10         val y = Future {function(2)}
11
12         val z = for
13         {
14                 xc <- x
15                 yc <- y
16         } yield xc + yc
17
18         val zval = Await.result(z, 0 nanos)
19         println(zval)
20 }
```

Obviously, this is a good deal more cumbersome than the Occam syntax and probably has more overhead, but the general idea of "do these things in parallel and then combine them" is alive and well, though modern replacements tend to be less streamlined for this purpose and have additional bells and whistles.

5 Erlang with OTP

> "Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance. [...] OTP is set of Erlang libraries and design principles providing middle-ware to develop these systems. It includes its own distributed database, applications to interface towards other languages, debugging and release handling tools." - Erlang home page

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.

5.1 Use

copy-on-send
    state as recursion argument
    flushing (timeout 0)

```
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.
```

5.2 Implementation

6 Go

> "Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language." - Go documentation

6.1 Use

6.2 Implementation

7 Rust

> "Rust provides safe concurrency through a combination of lightweight, memory-isolated tasks and message passing. [...] Rust tasks are not the same as traditional threads: rather, they are considered green threads, lightweight units of execution that the Rust runtime schedules cooperatively onto a small number of operating system threads. On a multi-core system Rust tasks will be scheduled in parallel by default. Because tasks are significantly cheaper to create than traditional threads, Rust can create hundreds of thousands of concurrent tasks on a typical 32-bit system. In general, all Rust code executes inside a task, including the main function." - Rust documentation

Rust is the newest and hottest language to be investigated here. Developed by Mozilla as a replacement for C++, its 1.0 spec is not even stable yet, but optional garbage collection and embeddability makes it very relevant for the embedded engineer of the near future. Semicolons mandatory.

7.1 Use

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

```
1  fn main()
2  {
3          let (tx, rx) = channel();
```

```
4           let txclone = tx.clone();
5
6           //proc denotes an anonymous function with
                function body being the stuff behind "proc()"
7           spawn(proc() tx.send("message"));
8           spawn(proc() txclone.send("another message"));
9           spawn(proc() println!("{:s}, {:s}" ,rx.recv(), rx
                .recv()) );
10  }
```

Above, an example of how a two-sender single-receiver program is written. Now, the future:

```
1   use std::sync::Future;
2
3   fn main()
4   {
5           fn fib(n: u64) -> u64
6           {
7                   // lengthy computation returning an uint
8                   12586269025
9           }
10          let mut delayed_fib = Future::spawn(proc() fib
                (50));
11          println!("fib(50) = {}", delayed_fib.get());
12  }
```

7.2 Implementation

When sending a message over a channel, 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.

8 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 diferently, 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.

9 Simulink

data transfer

10 Modelica

11 LabView

LabView uses pipelining.

12 Existing C message-passing implementations

- https://github.com/tylertreat/chan
- libthread
- CSP for C

13 Concepts

13.1 Thread Pool

A pool of real threads, POSIX or otherwise, each thread able to be executed directly on the processor. The notion of "real" thread may be extended to more abstract concurrency concepts like Java threads, since they map directly to OS threads. This pool of real threads are then used to execute lightweight or virtual threads. These lightweight threads are in reality only tasks for the real threads to execute.

13.2 Dispatching

How work tasks are given to the real threads.

13.3 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.

13.4 Fork-join dispatching

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. 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 task 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.

13.5 Future

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. For examples, see 4 and 7.1.

13.6 Channel

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.

13.7 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.

13.8 Pipe

A pipe is a single sender, single receiver, one-way and asynchronous communications channel between threads.

13.9 Coroutine

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 doing a 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.

13.9.1 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.

13.9.2 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
```

Observe that 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.

13.9.3 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.

14 Conclusion

The popular approaches taken to concurrency in modern languages can be roughly divided into two: The actor model and CSP. The two 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.

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 for C

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.

15 How to implement lightweight threads

Index

References

[1] http://doc.rust-lang.org/0.11.0/guide-tasks.html