Light Weight Threads and Their Synchronization Primitives

Tormod Hellen

December 8, 2014

Contents

Evaluation note

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

1 memo to self: paste code with edit->paste special->plain text

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

2 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.

2.1 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 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.

- 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.

3 Limitations

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, 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 synchronization 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.

4 Lightweight threads and synchronization

Lightweight threads and synchronization mechanisms are different things not really intimately linked. It's' quite possible to have one without

5 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 piece of memory somewhere else. By having a stack, goroutines are a bit like normal threads as they can be resumed after a pause.
- Akka's actors have no stack. How do they work then? The Java 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. See? Tradeoffs.

6 CSP Theory

This part is strictly optional. 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.

7 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. hides fault 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 analysability of such a system can still be an unsolveable problem due to the amount of states the software can be in.

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 may be better than constructing your own using semaphores and heavy threads.

8 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.

9 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.
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.

There's an awful lot of implementations of STM, though Clojure and Haskell are notable for using them as their primary means of synchronization.

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

```
module Main where
import Control.Monad
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.

10 Channels

Channels as a method for communication are analogous to the concurrency in CSP, and convenient to verify and do formal analysis upon.

11 Actors

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

11.1 Scala with Akka

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.

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!

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.

11.2 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.

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.
```

12 Synchronous Execution

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

12.1 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 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.

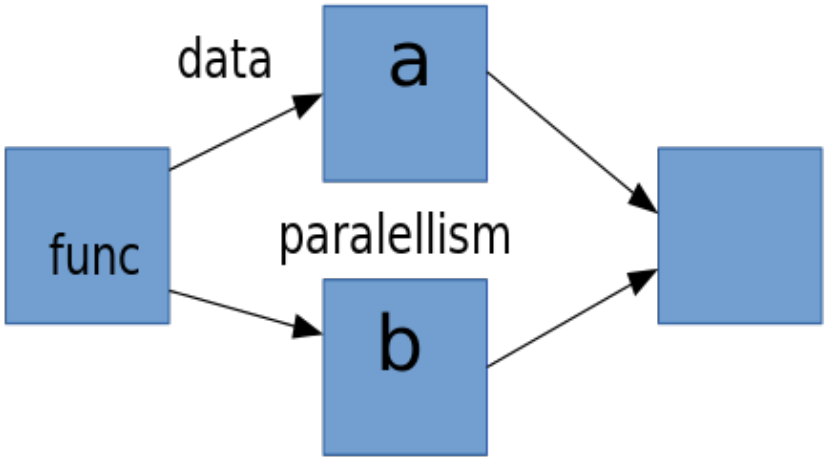
Esterel is not a real-time language but since it is completely deterministic, simple testing should suffice for investigating time characteristics.

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.

13 Parallel Statemens

13.1 LabView

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) 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:

```
1 SEQ
```

```
2       do stuff
3       PAR
4               a
5               b
6       do more stuff
```

http://www.ni.com/white-paper/6422/en/
http://zone.ni.com/reference/en-XX/help/370622K-01/lvrtbestpractices/rt_priorities/

14 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:

14.1 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.

14.2 Fork-join dispatching

Juicy stuff. This is what Go and Akka uses by default.

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.

References

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

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

Index