COSC441 Concurrent Programming

Time, Happens-Before, Logical Clocks

Richard A. O'Keefe

August 2, 2016

Newtonian time

"Absolute, true, and mathematical time, in and of itself and of its own nature, without reference to anything external, flows uniformly and by another name is called duration. Relative, apparent, and common time is any sensible and external measure (precise or imprecise) of duration by means of motion; such a measure — for example, an hour, a day, a month, a year — is commonly used instead of true time." — Newton, *Principia Mathematica.*

Unpacking Newton

- Time is *universal*
- Time is *regular*
- Time is *unbranching*
- Time is *continuous* (Archimedean)

Modelling with Time (1)

The natural way to model systems given such a view of time is as systems of coupled partial differential equations.

Parting with Newton: discrete time

- Digital computers are (made from analogue circuits pretending quite successfully to be) *discrete*-valued systems making state transitions at *discrete* points in time.
- Time is *discrete*, not continuous; like the integers, not the reals.

Modelling with Time (2)

- The natural way to model computer systems is as deterministic finite state automata augmented with inputs, outputs, and some sort of memory.

$$
\begin{aligned}
s(0) &= \text{initial state} \\
m(0) &= \text{initial memory} \\
s(t+1) &= f(s(t), m(t), i(t)) \\
m(t+1) &= g(s(t), m(t), i(t)) \\
o(t+1) &= h(s(t), m(t), i(t))
\end{aligned}
$$

- We can specify and reason about such systems using a *temporal logic*

Temporal Logic

- ϕ : ϕ is true at t
- $X\phi$: ϕ is true at $t+1$
- $F\phi$: ϕ is true at $t+n$ for some $n \geq 0$
- $G\phi$: ϕ is true at $t+n$ for all $n \geq 0$
- $\phi \mathcal{U} \psi$: there is an $n \geq 0$ such that ψ is true at $t+n$ and ϕ is true at all $t+m$ where $0 \leq m < n$.

Parting with Newton: nondeterminism

- VLSI systems are actually quantum devices.
- Modern Pentiums have a True Random Number Generator in hardware (based on thermal noise) accessed using the RDRAND and RDSEED instructions.
- We might want to model the environment of a system as unpredictable.
- This treats time as *branching* in the future ("the Trousers of Time").

Modelling with Time (3)

- Now deal with *nondeterministic* automata
- Temporal logic now quantifies over *paths* in the tree of time as well as *levels*.
- Look up CTL and CTL*
- Model checkers often use LTL, CTL, or some other selection from CTL*.

Parting from Newton: nonuniformity

- Computer clocks are a “sensible and external measure of duration by means of [electronic] motion”. They have always been subject to drift.
- In order to save electrical power, modern CPUs reduce the clock frequency when there is little to do and increase it when there is much to do — frequency scaling.
- So the number of CPU clock pulses per second changes often.
- Not an issue for discrete modelling and temporal reasoning.

The Story so Far

For single-processor systems,

- Time is *universal* (within a program)
- Time is **irregular**
- Time is **branching**
- Time is **discrete**

Relativistic Time

- Time passes at different rates for different observers
- Passing at high speed, two observers each think the *other's* clock is running slow
- Even the order of events may be perceived differently by different observers
- For details, see *The Feynman Lectures on Physics*

Goodbye, Newton

- In a distributed program,
- or even a multicore program,
- time is no longer universal.
- Different “places” have their own rubber clocks.

Rebuilding time

Instead of thinking of time as a *coördinate*, think of time as a network of *relations between events*, reflecting our basic ideas of *causality* and *information flow*

Events

An *event* is a thing that happens in a single instant at a single "place". Examples include

- Entry to/exit from a procedure
- Creation/death of a thread
- Sending/receiving a message (like an interrupt)
- Writing/reading a variable
- Acquiring/releasing a lock

Places

For our purposes, a thread is a place.
Events in a thread occur in the order specified by the program.

A partial order, even in one place

Consider $f(g(), h())$ in C.

- $g \uparrow \rightarrow g \downarrow \rightarrow f \uparrow$
- $h \uparrow \rightarrow h \downarrow \rightarrow f \uparrow$
- $f \uparrow \rightarrow f \downarrow$
- but NO order between $g \uparrow$ and $h \uparrow$ or $g \downarrow$ and $h \downarrow$; the compiler is allowed to inline and interleave the $g()$ and $h()$ calls.

Just for clarity

I'm not saying that the execution of a single-threaded C program is nondeterministic. (It *is*, I'm just not *saying* that.) The elementary events in a process will take place in *some* definite order. What I'm saying is that the language definition does not *fully specify* that order.

Lamport's Paper

The "happened-before" relation was defined by Leslie Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, CACM 21.7, July 1978.
"A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process."

Lamport's Definition

The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying ...

1. If a and b are events in the same thread and a comes before b, then $a \rightarrow b$.
2. If a is the sending of a message by one thread and b is the receipt of the same message ..., then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.
4. $a \not\rightarrow a$ (so $\rightarrow$ is a partial $<$ not a partial $\leq$).

Extending Lamport's Definition

- Lamport was giving a *minimal* definition. Sending and receiving a message was chosen as a typical *causal* connection.
- A lock must be released by the thread that owns it before it can be acquired by another.
- C11 and C++11 and Java and Go use happens-before with memory management events to define the effects of locks and atomic variable operations.

What does it mean for us?

- If we want $a \longrightarrow c$, we have to *make* that so by setting up a series of $a \longrightarrow \cdots \longrightarrow c$ causal connections.
- If we *don't* have such a series we can make no judgements about order.

An example

- (1) Process P sends message A to process Q then
- (2) Process P sends message B to process Q.
- (3) Process Q receives message A.
- (4) Process Q receives message B.
- We *can* conclude that (1) $\rightarrow$ (2) and (1) $\rightarrow$ (3) and (2) $\rightarrow$ (4).
- We *can't* conclude that (3) $\rightarrow$ (4).

That Example continued

- With UDP, out of order receipt really happens.
- With TCP, it doesn't.
- We need new events: (process) sends (message) through (channel) and (process) receives (message) through (channel) where events on the same channel are serialised like events in the same process.

Logical clocks

A *logical clock* for a thread p is a function C_p that assigns a number $C_p(e)$ to every event e that occurs in p, such that if $a \rightarrow b$ then $C_p(a) < C_p(b)$.
Given a system of logical clocks, define $C(e)$ to be $C_p(e)$ where e occurs in p.

But but but

- Didn't we just finish overthrowing Newton?
- This isn't Newtonian time. It's not even uniquely defined. But any partial order can be embedded in a total order and any total order can be numbered.
- *Clock Condition*: if $a \to b$ then $C(a) < C(b)$, **but** $C(a) < C(b)$ does **not** imply $a \to b$.

Getting started

- Each thread p maintains a *counter* C_p.
- If a happens in p, associate C_p with a, a happens, C_p is increased.
- If a is sending message m to q, send (m, C_p) instead.
- If a is receiving message m from q, we get (m, C_q) instead, set $C_p > \max(C_p, C_q)$.

Why?

If we want to say things like “requests must be served in the order in which they are made”, we have to agree on what that order is.
If we want to say “keep the most recent version”, we have to agree on what “recent” means.
Both of these need a total order.