Time, Clocks, and the Ordering of Events in a Distributed System

by Leslie Lamport

Introduction

• What is the problem?
  – Distributed systems are inherently concurrent, asynchronous, and nondeterministic
  – Executing programs on multiple machines requires coordination

• Who cares?
  – Developers and users of distributed systems

• What is the approach?
  – Introduce concept to define a ordering of events
Application Examples

• Distributed mutual exclusion

• Causal or total ordering on events

• Atomic distributed operations

• Etc.

Basics

• Basics
  – System: collection of processes
  – Process: sequence of events (with a priori total ordering)
  – Event: Execution of a subprogram/single instruction, message send and receive

• Introduce partial ordering

• Define *happens before* relation: \( a \rightarrow b \)

• Intuition for \( a \rightarrow b \)
  – event a happened before event b
  – event a can causally affect event b
Happens Before Relation

1. If a and b are events in the same process, and a comes before b, then \( a \rightarrow b \)

2. If a is sending of a message by one process and b is the receipt of the same message by another process, then \( a \rightarrow b \).

3. If \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \).

- Partial Order: Unordered events are concurrent

Happens-before explained graphically

![Diagram showing happens-before relation rules 1, 2, and 3]
Logical Clocks

- **Clock Condition:** For any events a, b: if a $\rightarrow$ b then $C\langle a \rangle < C\langle b \rangle$

- Holds if C1 and C2 are satisfied:
  - C1. If a and b are events in Process Pi, and a comes before b, then $C_i(a) < C_i(b)$
  - C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i(a) < C_j(b)$

- **Implementation**
  - IR1. Each process Pi increments $C_i$ between any two successive events
  - IR2a. If event a is the sending of a message m by Process Pi, then the message m contains a timestamp $T_m = C_j(a)$.
  - IR2b. Upon receiving a message m, process P_j sets $C_j$ greater than or equal to its present value and greater than $T_m$.

Clock Ticks

- if a $\rightarrow$ b then $C\langle a \rangle < C\langle b \rangle$

- $C\langle a \rangle < C\langle b \rangle$ doesn’t imply a $\rightarrow$ b
Total Ordering

• Partial ordering not always enough

• Prioritize processes $P_i < P_j$

• Total ordering $a \Rightarrow b$:

  If $a$ is in $P_i$ and $b$ is in $P_j$, then $a \Rightarrow b$ iff
  - $C_i(a) < C_j(b)$
  - $C_i(a) = C_j(b)$ and $P_i < P_j$

Example: Mutual Exclusion Problem

• Set of processes accessing a shared resource

• Conditions:
  I. A process that has been granted the resource must release it before it can be granted to another process
  II. The requests must be granted in the order they are made
  III. If every process eventually releases the resource, then every request will be eventually granted
Negative Example

- P0 is scheduler
- Events:
  1. P1 request resource
  2. P2 requests resource
  3. P0 grants resource to P2
- Violation of II

Scheduling Example

- Request Tm:Pi
  - Pj sends request to all processes Px
  - Pj puts request on its queue
  - Px puts request on its queue
  - Px sends time stamped acknowledgement
- Release Tm:Pi
  - Pj removes all requests with Tm from its queue
  - Pj sends release to all processes Px
  - Px removes any Tm:Pj requests from its queue
- Grant Tm:Pi
  - Pi:Tm is ordered before others queue
  - Pi has received message from all processes with timestamp later Tm
Logical Clocks

- Issues with physical clocks (clock drift, etc.)
- For many purposes, it is sufficient to know the order in which events occurred
- BUT: Logical clocks cannot be used to cover events outside the system

Strong Clock Condition

- Approach does not take into account external events
- Define new set of events \( \mathcal{L} \)
- \textit{Strong Clock Condition}: For any events \( a, b \) in \( \mathcal{L} \):
  \[ \text{if } a \Rightarrow b \text{ then } C(a) < C(b) \]
- Achieve strong clock condition with physical clocks
Physical Clocks

- Run continuously

- PC1. Clocks must run at approximately the correct rate
  - $\exists k << 1, |dC_i(t)/dt - 1| < k$

- PC2. Clocks must be synchronized
  - $|C_i(t) - C_j(t)| < \varepsilon$

- Minimum message delay $\mu$
  - $C_i(t+\mu) - C_j(t) > 0$

Satisfying Strong Clock Condition

- IR1. Each event occurs at a precise instant

- IR2.
  - If $P_i$ sends a message $m$ at physical time $t$, then $m$ contains a timestamp $T_m = C_i(t)$.
  - Upon receiving a message $m$ at time $t'$, process $P_j$ sets $C_j(t')$ equal to the maximum of $C_j(t')$ and $(T_m + \mu m)$
Applications

- Distributed Systems
  - BOINC

- Network
  - Peer to Peer

- Operating Systems
  - Interrupt coordination

- Databases
  - Access control

- Detecting Race Conditions

Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson
Introduction

• What is the problem?
  – Implementing multi-threaded programs is difficult and error prone

  ![What's Wrong With Threads?](image)

• Who cares?
  – Developers (and users) of multi-threaded systems

• What is the approach?
  – Provide tool support to automatically verify synchronization

Eraser

• Dynamic race detection tool

• Supports only lock based synchronization

• Claim: Simpler, more efficient, and more thorough than approaches based on happens-before
Some Definitions

- **Thread**
  - Independent execution stream with a shared state.
  - Processes do not share the same state

- **Lock**
  - Synchronization object used for mutual exclusion
  - Only the owner of a lock may release it (not like semaphores)

- **Data Race**
  - One of two threads have write access to variable without synchronization

Related Approaches

- **Monitors by Hoare**
  - Do not account for dynamically allocated data

- **Static race detection**
  - Need many test cases to produce reliable results

- **Race detection based on Happens Before**
  - Inefficient since large amount of information is required
Happens Before Problem

- Case I
  - no data race in y

- Case II
  - y is accessed by both threads simultaneously

- Reliability depends on test cases!

---

Lockset Algorithm

- First version: Enforces simple locking discipline
  - Each shared variable is accessed by at least one lock

- Problem: Eraser doesn’t know which lock is for which variables

- Solution: Infer protection relation from execution history
CA3  Explain analysis in this example
Chris, 10/21/2007
Lockset Algorithm cont’d

- Set $C(v)$ of candidate locks for each shared variable $v$
  - Holds the locks that have protected variable during execution

- Intuition:
  - Every time a thread $t$ accesses a shared variable $v$ it must hold at least one lock $l$.

- Algorithm:
  - Initialize $C(v)$ with all locks
  - $C(v) := C(v) \cap \text{locks}_\text{held}(t)$
  - $C(v) = \emptyset \rightarrow$ issue warning

```
y := y+1
lock(mu)
v := v+1
unlock(mu)
y := y+1
```

locks$_\text{held}(P0)$ locks$_\text{held}(P1)$ $C(v)$ $C(y)$ $P0$ $P1$

| $\emptyset$ | $\emptyset$ | {mu} | {mu} | y := y+1 |
| $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | lock(mu) |
| {mu} | $\emptyset$ | $\emptyset$ | $\emptyset$ | v := v+1 |
| {mu} | $\emptyset$ | {mu} | $\emptyset$ | unlock(mu) |
| $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $\emptyset$ | {mu} | $\emptyset$ | $\emptyset$ | lock(mu) |
| $\emptyset$ | {mu} | $\emptyset$ | $\emptyset$ | v := v+1 |
| $\emptyset$ | {mu} | {mu} | $\emptyset$ | unlock(mu) |
| $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | y := y+1 |
CA2  Insert steps for updating lock_held sets
Chris, 10/21/2007
Improvements

- Relax locking discipline

- Initialization: Shared variables initialized w/o holding lock
  - Algorithm “pauses” until variable is accessed by a second thread

- Read-shared data: Variables written during init only and read-only thereafter
  - No races are reported until a second thread writes to variable

- Read-write locks: Multiple readers can access a shared variable but only one writer at a time.
  - Keep track of write locks

States of Memory Locations

- Virgin:
  - New data, not referenced

- Exclusive
  - Accessed by one thread

- Shared
  - One write and multiple read accesses

- Shared-Modified
  - Multiple write accesses
CA1

Explain this further

Chris, 10/21/2007
Implementation

• Developed for DIGITAL Unix OS
  – now known as Tru64 UNIX (by HP)

• Input: Unmodified program binary

• Output: Instrumented binary that is functionally identical
  but includes calls to Eraser

• Race report:
  – file + line
  – list of stack frames
  – thread ID, memory address, type of access

Maintaining Lock Sets

• To maintain C(v)
  – Instrumented each call to storage allocator to init C(v) for
    dynamically allocated data
  – Instrument each load/store instruction

• To maintain lock_held(t)
  – Instrument each lock acquire/release (+ initialize/finalize)

• Each 32-bit word on heap or global data is possible
  shared variable
Representing Lock Sets

- List of lock sets for each memory location inefficient
  - Use hash tables to avoid duplicate lock sets

- Shared variables represented by *Shadow Words*
  - 30 bit for lockset index (or thread ID in exclusive state)
  - 2 bit for state condition

Experiences

- SPIN OS
  - Locks for certain interrupt levels
  - Extended Eraser to accommodate for interrupts

- AltaVista: 30 KLOC in C
  - 100 locks, 250 locksets

- Vesta Cache Server: 30 KLOC in C++, 10 threads
  - 26 locks, 70 locksets

- Petal: 25 KLOC in C, 64 threads

- Undergraduate Coursework
Results

• Detected a total of 3 serious data races

• Faults positives:
  – Semaphore synchronization leads to false positives
  – Benign races (to increase performance) cause fault positives
  – Individual list elements

Evaluation

• Effectiveness
  – Eraser more efficient than manual validation

• Sensitivity
  – Not sensitive to the number of threads

• Extension to detecting dead locks possible
Criticism

• Slows down program by a factor of 10 to 30
• High number of false positives
• Removing false positives might be time consuming

Current Status

• Helgrind implements the Lockset algorithm

• CheckSync implements Eraser for Java

• Sun’s data race detection tool: Sun Studio Express

• Microsoft is working on RaceTrack
any questions?