Lamport Clocks

- Distributed systems are inherently concurrent, asynchronous, and nondeterministic, so executing programs on multiple machines requires coordination
- Lamport introduce methods to define an ordering of events
- Want to create a partial ordering of events (instructions, message passing, or whatever)
- Define a happens before relation: \( a \rightarrow b \)
  - 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 \( a \rightarrow c \) (transitivity)

- Partial Order: Unordered events are concurrent
Logical Clocks

- **Clock Condition**: For any events $a, b$: if $a \rightarrow b$ then $C_{<a>} < C_{<b>}$
- **Hold if C1 and C2 are satisfied**:
  - **C1**: If $a$ and $b$ are events in Process $P_i$, and $a$ comes before $b$, then $C_{<a>} < C_{<b>}$
  - **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 $C_{<a>} < C_{<b>}$
- **Implementation**
  - **IR1**: Each process $P_i$ increments $C_i$ between any two successive events
  - **IR2a**: If event $a$ is the sending of a message $m$ by Process $P_i$, then the message $m$ contains a timestamp $T_m = C_{<a>}$
  - **IR2b**: Upon receiving a message $m$, process $P_j$ sets $C_j$ greater than or equal to its present value and greater than $T_m$.

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
  - $C_{<a>} < C_{<b>}$
  - $C_{<a>} = C_{<b>}$ and $P_i < P_j$

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 order events outside the system

Strong Clock Condition

- **Approach does not take into account external events**
- **Define new set of events**: $L$
- **Strong Clock Condition**: For any events $a, b$ in $L$:
  
  if $a \Rightarrow b$ then $C_{<a>} < C_{<b>}$
- **Achieve strong clock condition with physical clocks**
Physical Clocks

- **Run continuously**
  - $\exists k. k \ll 1 \cdot |dC_i(t)/dt - 1| < k$

- **PC1. Clocks must run at approximately the correct rate**
  - $|C_i(t) - C_j(t)| < \varepsilon$

- **PC2. Clocks must be synchronized**
  - $C_i(t) - C_j(t) > 0$

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

- **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 $(T_m + \mu_m)$

Eraser

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

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

- **What is the approach?**
  - Provide tool support to automatically verify synchronization

- **Dynamic data race detection tool**

- **Supports only lock-based synchronization**

- **Claim: Simpler, more efficient, and more thorough than approaches based on happens before**

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

- **Data Race**
  - More than 1 thread has read or write access to a variable without synchronization, and at least one is doing a write

Other 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
**Lockset Algorithm**

- **First version:** Enforces simple locking discipline
  - Each shared variable is protected by at least one lock
- **Problem:** Eraser doesn’t know which lock is for which variables
- **Solution:** Infer protection relation from execution history
- **Set** $C(v)$ of candidate locks for each shared variable $v$
  - Holds the locks that have protected a 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
  - $C(v) := C(v) \cap \text{locks\_held}(t)$
  - $C(v) = {} \rightarrow$ issue warning

**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 separately 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

**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 and Representing 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
- List of lock sets for each memory location inefficient
  - Use hash tables to avoid duplicate lock sets
- Shared variables represented by Shadow Words
  - 30 bits for lockset index (or thread ID in exclusive state)
  - 2 bits for state condition

Evaluation

- Effectiveness
  - Eraser more efficient than manual validation
- Sensitivity
  - Not sensitive to the number of threads
- Extension to detecting deadlocks possible

Problems

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

Current Status

- Helgrind implements the Lockset algorithm (current web page says it implements happens before)
- CheckSync implements Eraser for Java
- Microsoft was working on RaceTrack
- Intel Inspector - not clear what algorithm is used