CMSC 714
Lecture 15
Lamport Clocks and Race Conditions

Alan Sussman
(with thanks to Chris Ackermann)
Notes

• **OpenMP projects**
  • Keon working on them, but no results until later this week or early next week

• **Research project questions?**

• **Don’t forget to send questions about readings when you are assigned**
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 $C_{a} < C_{b}$

• Holds if C1 and C2 are satisfied:
  • C1. If $a$ and $b$ are events in Process $P_i$, 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 $P_i$ 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 $T_m = C_{i,a}$.
  • 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

• $C_i<a> < C_j<b>$
• $C_i<a> = C_j<b>$ 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
• Approach does not take into account external events

• Define new set of events $\mathcal{E}$

• **Strong Clock Condition**: For any events $a, b$ in $\mathcal{E}$:

  \[
  \text{if } a \Rightarrow b \text{ then } C<a> < C<b>
  \]

• Achieve strong clock condition with physical clocks
Physical Clocks

• Run continuously

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

• 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 \( (T_m + \mu_m) \)
Race Conditions

• 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
Data Races

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

• **Static race detection**
  • Analyze the program code, so does not require that the program execute
  • Difficult analysis, if *sound* (does not produce false negatives) tends to produce many false positives (lack of *completeness*)
  • Getting both soundness and completeness is undecidable
Data Races (cont.)

• **Dynamic race detection**
  • Analyze the events from a single program execution to determine the occurrence of a race condition in one program execution
  • Can be sound and complete, but only for that execution
  • Want to have the *single input, single execution* (SISE) property, so that a single execution instance is sufficient to determine the existence of a data race for a given input.
  • Two basic kinds – based on happens-before (HB) relation (Lamport), and based on locksets (e.g., Eraser algorithm)
HB-based Dynamic Race Detection

• Inefficient since large amount of information is required

• Basic idea has 3 parts:
  • track the HB-relation within each thread
  • keep an access history as a sequence of logical timestamps for each shared resource (variable or memory location)
  • validate that, for every resource, critical accesses are ordered by the HB-relation

• While the analysis can be sound and complete, the article shows that with a more general notion of data races, the HB-based analysis does not report all possible data races so is not sound wrt that definition
Lockset-based Detection

• Targeted at programs that use *critical sections* as their primary synchronization model

• Validates that a program execution adheres to a programming policy, called a *locking discipline*
  
  • E.g., threads that access a common memory location must hold a mutual exclusion lock when performing the access

• Compliance with the locking discipline implies that executions don’t have a data race

• Validation can be done with static or dynamic analysis, or both
Lockset-based algorithm

• Each thread tracks at run-time the set of locks it currently holds – i.e. via a shadow location for each variable that holds the current lockset.
• On the first access to a shared variable, the shadow memory is initialized with the lockset of the current thread.
• On subsequent accesses, the lockset in shadow memory is updated by intersecting it with the lockset of the accessing thread.
• If the intersection is empty and the variable has been accessed by different threads, a potential data race is reported.
• Lockset-based detection is sound, and has the SISE property.
• Detection is incomplete, since accesses that violate the locking discipline may be ordered by other means of synchronization – so can get false positives.
Static Data Race Detection

- **Pragmatic methods** look for deviations from common programming practice
  - Examples include FindBugs for Java from UMD, RacerX for large OS codes

- **Methods based on dataflow analysis**
  - May-happen-in-parallel analysis (MHP) to compute the may-happen-in-parallel relation among statements in different threads
  - Inter-process precedence graph for determining anomalies in programs with post-wait synchronization

- **Type-based methods**
  - To model and express data protection and locking policies in data and method declarations

- **Model checking**
  - To explore every possible control flow-path and variable value assignment for undesired program behavior
  - Since that is computationally intractable, models of the data and program are explored