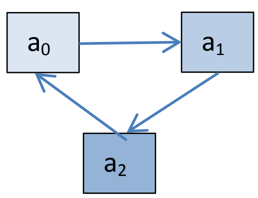
Quinn Damerell
CS390C – Concurrently and Parallelism
Notes for 2/23/2012 Lecture 7

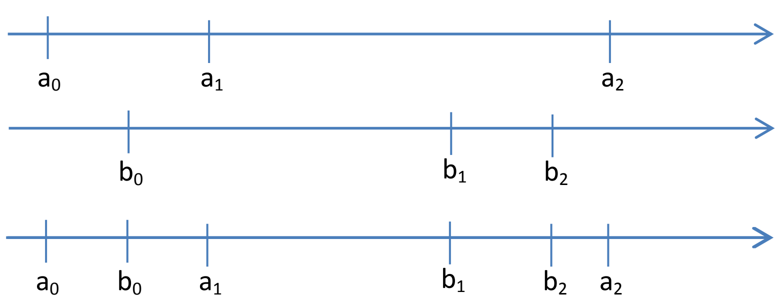
Mutual Exclusion

1) Goal
   a) Locking is essential for protecting shared data in programs, however locking can be a large source of slow down
   b) Thus we need to understand how these locks work and are implemented so we can use fully understand the most efficient ways of using them.
2) Some Background
   a) Thinking in timelines
   b) Say a we have a timeline that preforms events in a sequential order

      i) Here A are the event being performed in a liner order
      ii) A notation of $A_0 \rightarrow A_1$ indicates order
         (1) A_0 "happens before" A_1
   c) Thinking in State Machines
      i) We can also think of this concept as a state machine, where every event is a node and the transitions between the nodes are the connected paths.



      ii) Each node is an action preformed, and the arrows between them are the transition states.
   d) Concurrency
      i) To achieve concurrency, we simply need to create two separate times threads, and merge them in to one.



      ii) Here we can see an example of two time threads becoming one
      iii) Notice they are Interleaved, but not necessarily independent

e) Intervals
   i) Problem: How do we make a consistent unit of time among all threads?
      (1) We don't want one to be thinking in a unit of weeks, and another to be in a until of months
      (2) We create intervals
         (a) Interval $A_0 = (a_0, a_1)$
            (i) Interval A_0 is simply the time between a_0 and a_1
      (3) Now that both A and B have intervals we can put them tighter on a timeline.
      (4) Note Intervals can overlap or be disjoint
   ii) Our goal is to ensure that will any interval is happing, no other interval will start, end, or be running with it. If we can do this, we can ensure that the program is correct.

f) Precedence
   i) We want to run multiple things at the same time, but with no overlap in any intervals
   ii) We want to think of the problem as sequential, but have the solution run in parallel
   iii) Precedence
      (1) $A_0 \rightarrow B_0$
         (a) A will run before B, formally a_1 ends before b_1 (where a and b are events in A and B)
   iv) An important note is that threads still make think of "their" intervals differently. For example one thread might set any array of 1 million ints to 0, and anther thread might just want to read the 16th int. The large thread might define it's interval to hold the entire array while it does it operation, but this is not fair to the small thread that just wants to read the 16th element.
   v) Precedence Rules
      (1) Irreflexive
         (a) A ➔ A is never true
      (2) Antisymmetric
         (a) If A ➔ B, then B ➔ A can never happen
      (3) Transitive
         (a) If A ➔ B and B ➔ C, then A ➔ C
      (4) A ➔ B and B ➔ A might both be false
         (a) If A and B overlap

g) Notes:
   i) One thread always has total order
   ii) Multiple threads may not have total order

3) Implementing a Counter
   a) Goal
      i) Now that we have some fundamental concepts and terms defined, how do we use this knowledge to build locks ourselves? And how can we prove that they are always correct?
      ii) Correctness of Locking
         (1) If a program is correct: Any interval will never overlap another interval
      iii) If we want to ensure mutual exclusion, intervals must form a total ordering.

b) Section Goal
   i) Build a lock without hardware, library, and language support
c) Example of where we need a lock (see lecture 7; slide 19 for code)
   i) This for a single thread, but multiple threads can enter the critical section at the same time and mess up the shared variable
      (1) Example
         (a) Thread one comes in and reads value (let's say 1) into temp, gets swapped out
         (b) Thread two comes in and read value (1 again) into temp, and sets value to be temp +1 (so temp = 2), gets swapped out
         (c) Thread one continues and sets value to be temp + 1 (also temp = 2) returns temp
         (d) Thread two continues and returns temp as 2
         (e) oops
d) How do we fix this?
   i) Define Mutual Exclusion
      (1) Let CS^k_j be a critical section
      (2) Let CS^m_i be a different critical section
      (3) $CS^k_j \rightarrow CS^m_i$ OR $CS^m_i \rightarrow CS^k_j$
         (a) If we can prove this always holds true, we have correct
         (b) This means our program will only allow one thread to be in the critical code at a time
   ii) Define Deadlock-free
      (1) If a thread calls lock and never returns, we must make it so other threads will not get stuck waiting
      (2) The system as a whole makes progress
   iii) Define Starvation-Free
      (1) If some thread calls lock, it will return eventually.
      (2) Then all individual threads can make progress
      (3) The critical section intervals must be finite
e) Creating A Lock
   i) We need to guarantee mutual exclusion, and no deadlock
   ii) For consistency we define:
      (1) Thread i – The thread already in the critical section
      (2) Thread j – The thread that wants to get in

Cont. on next page

iii) Attempt One (See lecture 7; slide 30)
   (1) The Idea
      (a) Thread I comes in, sets his flag to true
      (b) While the j flag is true thread one keep checking, when it is false thread one "has the lock" and will return
      (c) Thread one needs to set its flag to false (when it is done) before thread 2 can get the lock.
   (2) The Good
      (a) This lock does satisfied mutual exclusion
         (i) The flag sets its flag to true (meaning it is in the critical section) and then waits for the other to be false
         (ii) Can be proved by contradiction
   (3) The Bad
      (a) This does not prevent deadlock
      (b) Example
         (i) Thread i comes in and sets its flag to true, get swapped
         (ii) Thread j comes in and sets its flag to true, spins on the while because i is true, gets swapped
         (iii) Thread i spins on the while because j is set to true
         (iv) Continues forever
iv) Attempt Two (See lecture 7; slide 34)
   (1) The idea:
      (a) Our problem with the last implementation was that we had two "bits" (two flags) to control one critical section
      (b) Try this one:
         (i) Thread i comes in and sets its self to be the victim, then spins on the while until it doesn't equal the victim any longer, gets swapped out
         (ii) Thread j comes in and sets its self to be the victim, then spins on the while until it doesn't equal the victim any longer, gets swapped out
         (iii) Resume Thread i, it check the while loop and is no longer the victim so it gets the lock and returns.
         (iv) They will keep handing off until they are done, but the last one to call the lock method will be stuck forever.
   (2) The Good:
      (a) This is does satisfy mutual exclusion
         (i) If thread I is in CS
         (ii) The victim == j
         (iii) Cannot be both 0 and 1
   (3) The Bad
      (a) Fails at being deadlock free
         (i) Sequential execution deadlocks
         (ii) Concurrent execution does not until the end.