Lecture 7: Mutual Exclusion
2/16/12

slides adapted from The Art of Multiprocessor Programming, Herlihy and Shavit
Time

- “Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external.” (I. Newton, 1689)

- “Time is, like, Nature’s way of making sure that everything doesn’t happen all at once.” (Anonymous, circa 1968)
Events

- An event $a_0$ of thread A is
  - Instantaneous
  - No simultaneous events (break ties)
A thread $A$ is (formally) a sequence $a_0, a_1, \ldots$ of events

- “Trace” model
- Notation: $a_0 \rightarrow a_1$ indicates order
Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …
Threads are State Machines

Events are transitions
States

- **Thread State**
  - Program counter
  - Local variables

- **System state**
  - Object fields (shared variables)
  - Union of thread states
Concurrency

- Thread A
- Thread B
Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)
Intervals

- An *interval* $A_0 = (a_0, a_1)$ is
  - Time between events $a_0$ and $a_1$
Intervals may Overlap

CS390C: Principles of Concurrency and Parallelism
Intervals may be Disjoint

CS390C: Principles of Concurrency and Parallelism
Precedence

Interval $A_0$ precedes interval $B_0$
Precedence

- Notation: $A_0 \rightarrow B_0$
- Formally,
  - End event of $A_0$ before start event of $B_0$
  - Also called “happens before” or “precedes”
Precedence Ordering

- Remark: $A_0 \rightarrow B_0$ is just like saying
  - 1066 AD $\rightarrow$ 1492 AD,
  - Middle Ages $\rightarrow$ Renaissance,
- Oh wait,
  - what about this week vs this month?
Precedence Ordering

- Never true that $A \Rightarrow A$
- If $A \Rightarrow B$ then not true that $B \Rightarrow A$
- If $A \Rightarrow B$ & $B \Rightarrow C$ then $A \Rightarrow C$
- Clearly: $A \Rightarrow B$ & $B \Rightarrow A$ might both be false!
Partial Orders
(review)

- **Irreflexive:**
  - Never true that \( A \rightarrow A \)

- **Antisymmetric:**
  - If \( A \rightarrow B \) then not true that \( B \rightarrow A \)

- **Transitive:**
  - If \( A \rightarrow B \) & \( B \rightarrow C \) then \( A \rightarrow C \)
Total Orders
(review)

- Also
  - Irreflexive
  - Antisymmetric
  - Transitive

- Except that for every distinct A, B,
  - Either A \(\rightarrow\) B or B \(\rightarrow\) A
Implementing a Counter

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp  = value;
        value = temp + 1;
        return temp;
    }
}
```

Make these steps *indivisible* using locks.
Locks (Mutual Exclusion)

```java
public interface Lock {
    public void lock();
    public void unlock();
}
```
Locks (Mutual Exclusion)

```java
public interface Lock {
    public void lock();
    public void unlock();
}
```

- acquire lock
- release lock
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = value + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
Mutual Exclusion

- Let $CS_i^k$ be thread i’s k-th critical section execution.
- And $CS_j^m$ be j’s m-th execution.
- Then either $CS_i^k \rightarrow CS_j^m$ or $CS_j^m \rightarrow CS_i^k$. 

CS390C: Principles of Concurrency and Parallelism

Wednesday, February 15, 12
Deadlock-Free

- If some thread calls `lock()`
  - And never returns
  - Then other threads must complete `lock()` and `unlock()` calls infinitely often
- System as a whole makes progress
  - Even if individuals starve
Starvation-Free

- If some thread calls lock()
  - It will eventually return
- Individual threads make progress
class ... implements Lock {
    ...
    // thread-local index, 0 or 1
    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        ...
    }
}
Two-Thread Conventions

```java
class ... implements Lock {
    ...
    // thread-local index, 0 or 1
    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        ...
    }
}
```

Henceforth: i is current thread, j is other thread
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];
    public void lock() {
        flag[i] = true;
        while (flag[j]) {} 
    }
}
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];
    public void lock() {
        flag[i] = true;
        while (flag[j]){}
    }
}
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];
    public void lock() {
        flag[i] = true;
        while (flag[j]) {} 
    }
}
class LockOne implements Lock {
  private boolean[] flag = new boolean[2];
  public void lock() {
    flag[i] = true;
    while (flag[j]) {
    }
  }
}

Wait for other flag to become false
LockOne Satisfies Mutual Exclusion

- Assume $CS_A^j$ overlaps $CS_B^k$
- Consider each thread's last ($j$-th and $k$-th) read and write in the lock() method before entering
- Derive a contradiction
Deadlock Freedom

- **LockOne Fails deadlock-freedom**
  - Concurrent execution can deadlock

```c
flag[i] = true;  flag[j] = true;
while (flag[j]){}  while (flag[i]){}
```
public class LockTwo implements Lock {
    private int victim;
    public void lock() {
        victim = i;
        while (victim == i) {};
    }

    public void unlock() {}
}
LockTwo Claims

- Satisfies mutual exclusion
  - If thread i in CS
  - Then \texttt{victim} == j
  - Cannot be both 0 and 1
- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

```java
public void LockTwo() {
    victim = i;
    while (victim == i) {];//
}
```
Peterson's Algorithm

public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {
    }
}
public void unlock() {
    flag[i] = false;
}
Deadlock Free

public void lock() {
    ...
    while (flag[j] && victim == i) {};

● Thread blocked
  - only at `while` loop
  - only if other’s flag is true
  - only if it is the `victim`
● Solo: other’s flag is false
● Both: one or the other not the victim
Starvation Free

- Thread i blocked only if j repeatedly re-enters so that flag[j] == true and victim == i
- When j re-enters
  - it sets victim to j.
  - So i gets in

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}

public void unlock() {
    flag[i] = false;
}
```
Bakery Algorithm: Generalizing to n Threads

- Provides First-Come-First-Served
  - fairness
  - locks have two parts:
    - doorway: bounded number of steps
    - waiting: potentially unbounded number of steps
  - whenever a thread A finishes its doorway before thread B starts its doorway, A cannot be overtaken by B

- How?
  - Take a “number”
  - Wait until lower numbers have been served

- Lexicographic order
  - \((a,i) > (b,j)\)
    - If \(a > b\), or \(a = b\) and \(i > j\)
Bakery Algorithm

class Bakery implements Lock {
    boolean[] flag;
    Label[] label;
    public Bakery (int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }
    ...
}
Bakery Algorithm

class Bakery implements Lock {
    boolean[] flag;
    Label[] label;
    
    public Bakery (int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }
    ...
}
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k]
                && (label[i],i) > (label[k],k));
    }
}
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k] && (label[i], i) > (label[k], k));
    }
}
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k] && (label[i],i) > (label[k],k));
    }

I’m interested
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1])+1;
        while (∃k flag[k]
            && (label[i],i) > (label[k],k));
    }

Take increasing label (read labels in some arbitrary order)
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k]
            && (label[i], i) > (label[k], k));
    }

Bakery Algorithm

class Bakery implements Lock {
    boolean flag[n];
    int label[n];

    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], …, label[n-1]) + 1;
        while (∃k flag[k] && (label[i], i) > (label[k], k));
    }
}

Someone is interested …

… whose (label, i) in lexicographic order is lower
class Bakery implements Lock {

    ...

    public void unlock() {
        flag[i] = false;
    }

}
Bakery Algorithm

class Bakery implements Lock {
    ... 
    public void unlock() {
        flag[i] = false;
    }
}

No longer interested
class Bakery implements Lock {

    ...

    public void unlock() {
        flag[i] = false;
    }
}

labels are always increasing
Timestamps

- Label variable is really a timestamp
- Need ability to
  - Read others’ timestamps
  - Compare them
  - Generate a later timestamp
- Can we do this without overflow?
The Good News

- One can construct a
  - Wait-free (no mutual exclusion)
  - Concurrent
  - Timestamping system
  - That never overflows
The Good News

- One can construct a
  - Wait-free (no mutual exclusion)
  - Concurrent
  - Timestamping system
  - That never overflows

This part is hard
Deep Philosophical Question

○ The Bakery Algorithm is
  - Succinct,
  - Elegant, and
  - Fair.

○ Q: So why isn’t it practical?

○ A: Well, you have to read $N$ distinct variables
Shared Memory

- Shared read/write memory locations called Registers (historical reasons)
- Come in different flavors
  - Multi-Reader-Single-Writer (Flag[])
  - Multi-Reader-Multi-Writer (Victim[])
  - Not that interesting: SRMW and SRSW
Bad News Theorem

At least $N$ MRMW multi-reader/multi-writer registers are needed to solve deadlock-free mutual exclusion.

(So multiple writers don’t help)
Theorem (For 2 Threads)

Theorem: Deadlock-free mutual exclusion for 2 threads requires at least 2 multi-reader multi-writer registers

Proof: assume one register suffices and derive a contradiction
Two Thread Execution

- Threads run, reading and writing R
- Deadlock free so at least one gets in

CS390C: Principles of Concurrency and Parallelism
In any protocol B has to write to the register before entering CS, so stop it just before
Proof: Assume Cover of 1

A runs, possibly writes to the register, enters CS
Proof: Assume Cover of 1

B Runs, first obliterating any trace of A, then also enters the critical section.
Proof: Assume Cover of 1

B Runs, first obliterating any trace of A, then also enters the critical section
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work

- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.

- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.