Mutual Exclusion

CS 475, Fall 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
Review: Processes vs Threads

• Context Switching
  • Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
  • When switching processes, all of that data needs to get flushed out (by the OS)
  • Threads share the same address space: no need to do this switch
Review: Threads in Java

```java
public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            //This code will now run in a new thread
            System.out.println("Hello from the thread!");
        }
    });
    t.start();
    System.out.println("Hello from main!");
    t.join();
}
```

What is the output of this code?

This is a race condition

#1 Hello from the thread!
   Hello from main!

#2 Hello from main!
   Hello from the thread!
Producer/Consumer

• Alice and Bob can’t meet
  – Each has restraining order on other
  – So he puts food in the pond
  – And later, she releases the pets

• Avoid
  – Releasing pets when there’s no food
  – Putting out food if uneaten food remains
To post a message

WASH THE CAR

whew
Let’s send another message
Uh-Oh

SELL THE CAR

OK
Readers/Writers

- Devise a protocol so that
  - Writer writes one letter at a time
  - Reader reads one letter at a time
  - Reader sees
    - Old message or new message
    - No mixed messages
Readers/Writers (continued)

• Easy with mutual exclusion
• But mutual exclusion requires waiting
  – One waits for the other
  – Everyone executes sequentially
Amdahl's Law

• Identifies performance gains from adding additional cores to an application that has both serial and parallel components
• $S$ is serial portion
• $N$ processing cores
• That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
• As $N$ approaches infinity, speedup approaches $1 / S$
• Serial portion of an application has disproportionate effect on performance gained by adding additional cores
Today

• Today we will try to formalize our understanding of mutual exclusion
• We will also use the opportunity to show you how to argue about and prove various properties in an asynchronous concurrent setting
• Reading: H&S 2.1-2.3
• Note: HW1 posted: https://www.jonbell.net/gmu-cs-475-fall-2019/homework-1/
Why is Concurrent Programming so Hard?

• Try preparing a seven-course banquet
  - By yourself
  - With one friend
  - With twenty-seven friends …

• Before we can talk about programs
  - Need a language
  - Describing time and concurrency
“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)
Threads

- 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
Concurreny

• Thread A
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

\[ a_0 \quad B_0 \quad b_1 \]

\[ a_0 \quad A_0 \quad a_1 \]

\[ \text{time} \]
Intervals may be Disjoint
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”
- Informally,
  - Alice can’t look at the billboard while Bob is changing the letters
• 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?
• 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 $A \rightarrow C$
• Funny thing: $A \rightarrow B$ & $B \rightarrow A$ might both be false!
Partial Orders
(you may know this already)

• 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
(you may know this already)

• Also
  – Irreflexive
  – Antisymmetric
  – Transitive

• Except that for every distinct A, B,
  – Either $A \rightarrow B \text{ or } B \rightarrow A$
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}
Locks (Mutual Exclusion)

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

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

acquire lock
Locks (Mutual Exclusion)

public interface Lock {
    public void lock();  // acquire lock
    public void unlock(); // release lock
}
Using Locks

```java
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;
    }
}
```
Using Locks

```java
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;
    }
}
```
Using Locks

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

Release lock (no matter what)
Using Locks

```java
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;
    }
}
```

Critical section
Mutual Exclusion, Formally

- Let $CS_i^k$ be thread $i$’s $k$-th critical section execution
Mutual Exclusion, Formally

- Let $\text{CS}_i^k$ be thread i’s k-th critical section execution
- And $\text{CS}_j^m$ be thread j’s m-th critical section execution
Mutual Exclusion, Formally

- Let $\text{CS}_i^k$ be thread i’s k-th critical section execution
- And $\text{CS}_j^m$ be thread j’s m-th critical section execution
- Then either
  - or

Mutual Exclusion, Formally

- Let $\text{CS}_i^k$ be thread i’s k-th critical section execution
- And $\text{CS}_j^m$ be thread j’s m-th execution
- Then either
  - or

$\text{CS}_i^k \rightarrow \text{CS}_j^m$
Mutual Exclusion, Formally

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

Aka: it is guaranteed that one critical section happens before the other (NOT concurrently)
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
Locking in Java

• Most locks are *reentrant*: if you hold it, and ask for it again, you don’t have to wait (because you already have it)

• Basic primitives:
  • `synchronized{}`
  • We will come back to in next few weeks

• Plus…
  • Lock API… `lock.lock()`, `lock.unlock()`
  • The *preferred* way
How to Implement a Lock?

• Note - we will ONLY discuss how to implement mutual exclusion locks for two threads
• $n$-Thread solutions exist too, they just take somewhat longer to explain (see book if curious)
• Note - It’s unlikely you will ever implement a mutual exclusion lock. But, they provide a great example to understand how to write concurrent algorithms
Peterson’s Algorithm

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

Where i is the index of the current thread, j is the index of the other thread
```
Peterson’s Algorithm

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

Where $i$ is the index of the current thread, $j$ is the index of the other thread

Announce I’m interested
Peterson’s Algorithm

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

Where \(i\) is the index of the current thread, \(j\) is the index of the other thread.
Peterson’s Algorithm

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

Where i is the index of the current thread, j is the index of the other thread
Peterson’s Algorithm

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

Announce I’m interested
Defer to other
Wait while other interested & I’m the victim
No longer interested

Where \( i \) is the index of the current thread, \( j \) is the index of the other thread.
Peterson’s Alg: Mutual Exclusion

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

- If thread 1 in critical section,
  - flag[1]=true,
  - victim = 0

- If thread 0 in critical section,
  - flag[0]=true,
  - victim = 1

Cannot both be true, hence yes: it is safe!
Peterson’s Alg: Deadlock Free

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

- Thread blocked
  - only at `while` loop
  - only if it is the victim

- One or the other must not be the victim
Peterson’s Alg: Starvation Free

• Thread $i$ blocked only if $j$ repeatedly re-enters so that
  \[\text{flag}[j] == \text{true and victim} == i\]

• 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;
}
```
Introducing HW1

This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
Peterson’s Algorithm

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>i=1</td>
</tr>
<tr>
<td>j=1</td>
<td>j=0</td>
</tr>
</tbody>
</table>

Victim:
flag[0]:
flag[1]:

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

public void unlock() {
    flag[i] = false;
}
```
Peterson’s Algorithm

Thread 0 wants lock, Thread 1 doesn’t have or want it

Thread 0 wants lock, Thread 1 has it already

Thread 1 wants lock, Thread 0 wants it also

Thread 0

Thread 1