Introduction to Concurrency

Adapted by BCP from lectures by Maurice Herlihy at Brown

From the New York Times ...

SAN FRANCISCO, May 7. 2004 - Intel said on Friday that it was scrapping its development of two microprocessors, a move that is a shift in the company's business strategy.

Moore's Law

On Your Desktop: The Uniprocessor

In the Enterprise: The Shared Memory Multiprocessor (SMP)

Your New Desktop: The Multicore processor (CMP)

© 2007 Herlihy and Shavit
Multicores Are Here

- “Intel ups ante with 4-core chip. New microprocessor, due this year, will be faster, use less electricity…” [San Fran Chronicle]
- “AMD will launch a dual-core version of its Opteron server processor at an event in New York on April 21.” [PC World]
- “Sun’s Niagara…will have eight cores, each core capable of running 4 threads in parallel, for 32 concurrently running threads.” [The Inquirer]

Why do we care?

- Time no longer cures software bloat
  - The “free ride” is over
- When you double the work your program is doing…
  - …you can’t just wait 6 months for it to run the same speed again!
  - Your software must somehow exploit twice as much concurrency

Traditional Scaling Process

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1.8x</th>
<th>3.6x</th>
<th>7x</th>
</tr>
</thead>
<tbody>
<tr>
<td>User code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traditional Uniprocessor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time: Moore’s law

Multicore Scaling Process

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1.8x</th>
<th>3.6x</th>
<th>7x</th>
</tr>
</thead>
<tbody>
<tr>
<td>User code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicore</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unfortunately, not so simple...

Real-World Scaling Process

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1.8x</th>
<th>2x</th>
<th>2.9x</th>
</tr>
</thead>
<tbody>
<tr>
<td>User code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicore</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parallelization and Synchronization require great care...

Sequential Computation

thread

memory
Concurrent Computation

Asynchrony

Model Summary

• Multiple threads
  - Sometimes called processes
• Single shared memory
• Unpredictable asynchronous delays

Road Map

• Today: background on concurrency
• Monday: semantics of Haskell’s basic concurrency primitives (threads/MVars)
• Wednesday: thread programming
• Following week: Software Transactional Memory (STM)

Concurrency Jargon

• Hardware
  - Processors
• Software
  - Threads, processes
• Sometimes OK to confuse them, sometimes not.

Parallel Primality Testing

• Challenge
  - Print primes from 1 to \(10^{10}\)
• Given
  - Ten-processor multiprocessor
  - One thread per processor
• Goal
  - Get ten-fold speedup (or close)
Load Balancing

Idea:
- Split the work evenly
- Each thread tests range of $10^9$

Procedure for Thread $i$

```java
void primePrint {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*10^9+1, j<(i+1)*10^9; j++) {
        if (isPrime(j))
            print(j);
    }
}
```

Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need dynamic load balancing

Shared Counter

```
int counter = new Counter(1);
void primePrint {
    long j = 0;
    while (j < 10^10) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```
Procedure for Thread i

Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 10^{10}) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}

Where Things Reside

Counter Implementation

public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}

OK for single thread, not for concurrent threads (i.e., not "thread safe")
What It Means

```java
public class Counter {
    private long value;
    public long getAndIncrement() {
        return value++;
    }
}
```

What It Means

```java
public class Counter {
    private long value;
    public long getAndIncrement() {
        return value++;
        temp = value;
        value = value + 1;
        return temp;
    }
}
```

Not so good...

<table>
<thead>
<tr>
<th>Value...</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is this problem inherent?

If we could only glue reads and writes...

Challenge

```java
public class Counter {
    private long value;
    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}
```

Challenge

Make these steps atomic (indivisible)
**Hardware Solution**

```java
public class Counter {
    private long value;
    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}
```

**An Aside: Java™**

```java
public class Counter {
    private long value;
    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

**An Aside: Java™**

Haskell uses slightly different primitives to achieve the same effect.

**Synchronized block**

**Mutual Exclusion, or “Alice & Bob share a pond”**

**Alice has a pet**
Bob has a pet

The Problem

The pets don't get along

Formalizing the Problem

• Two types of formal properties in asynchronous computation:
  - Safety Properties
    • Nothing bad happens ever
  - Liveness Properties
    • Something good happens eventually

Formalizing our Problem

• Mutual Exclusion
  - Both pets never in pond simultaneously
  This is a safety property
• No Deadlock
  - if only one wants in, it gets in
  - if both want in, one gets in
  This is a liveness property

Simple Protocol

• Idea
  - Just look at the pond, see if it is empty, and release pet if so
• Gotcha
  - Both look at the same instant
  - Both release pets
  - Bad thing happens in pond

Telephone Protocol

• Idea
  - Bob calls Alice (or vice-versa)
• Gotcha
  - Alice in shower when Bob calls
  - Bob recharging phone battery when Alice calls
  - Alice out shopping for pet food when Bob calls...
**Patient Telephone Protocol**

- **Idea**
  - Bob calls Alice (or vice-versa) and lets phone ring until Alice answers

- **Gotcha**
  - Alice goes on vacation for a month...

- **Lesson**
  - Need to be able to leave persistent messages (like writing, not speaking)

**Can Protocol**

- **Idea**
  - Cans on Alice's windowsill
  - Strings lead to Bob's house
  - Bob pulls strings, knocks over cans

- **Gotcha**
  - Cans cannot be reused
  - Bob runs out of cans

**Bob conveys a bit**

**Can Protocol**

**Flag Protocol**
Alice’s Protocol (roughly)

- Raise flag
- Wait until Bob’s flag is down
- Unleash pet
- Lower flag when pet returns

Bob’s Protocol (roughly)

- Raise flag
- Wait until Alice’s flag is down
- Unleash pet
- Lower flag when pet returns

Bob’s Protocol (2nd try)

- Raise flag
- While Alice’s flag is up
  - Lower flag
  - Wait for Alice’s flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

The Flag Principle

- Raise the flag
- Look at other’s flag
- Flag Principle:
  - If each raises and looks, then
  - Last to look must see both flags up

danger!
Proof of Mutual Exclusion

- Assume both pets in pond
  - Derive a contradiction
  - By reasoning backwards
- Consider the last time Alice and Bob each looked before letting the pets in
- Without loss of generality assume Alice was the last to look...

Proof

- Bob last raised flag
- Alice last raised her flag
- Bob’s last look
- Alice’s last look
- QED

Alice must have seen Bob’s Flag. A Contradiction

Proof of No Deadlock

- If only one pet wants in, it gets in.

Proof of No Deadlock

- If only one pet wants in, it gets in.
- Deadlock requires both continually trying to get in.
- If Bob sees Alice’s flag, he gives her priority (a gentleman...)

Remarks

- Protocol is unfair
  - Bob’s pet might never get in
- Protocol uses waiting
  - If Bob is eaten by his pet, Alice’s pet might never get in
Moral of Story

• Mutual Exclusion cannot be solved by
  - transient communication (cell phones)
  - interrupts (cans)
• It can be solved by
  - one-bit shared variables
  - that can be read or written

The Fable Continues

• Alice and Bob fall in love & marry

The Fable Continues

• Alice and Bob fall in love & marry
  • Then they fall out of love & divorce
    - She gets the pets
    - He has to feed them
• Leading to a new coordination problem: Producer-Consumer

Bob Puts Food in the Pond

Alice releases her pets to Feed

mmm...
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

Surprise Solution

Bob puts food in Pond

Alice Releases Pets

Bob knocks over Can
Alice Resets Can when Pets are Fed

Pseudocode

```java
while (true) {
    while (can.isUp()) {
        pet.release();
        pet.recapture();
        can.reset();
    }
}
```

Alice's code

Pseudocode

```java
while (true) {
    while (can.isDown()) {
        pond.stockWithFood();
        can.knockOver();
    }
}
```

Bob's code

Correctness

- **Mutual Exclusion**
  - Pets and Bob never together in pond

- **No Starvation**
  - If Bob always willing to feed and pets always famished, then pets eat infinitely often.

- **Producer/Consumer**
  - Pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.

Waiting

- Both solutions use waiting
  - `while (mumble) {}`
- Waiting is problematic
  - If one participant is delayed, so is everyone else!
  - But delays are common & unpredictable
The Fable drags on …
• Bob and Alice still have issues

The Fable drags on …
• Bob and Alice still have issues
• So they need to communicate

The Fable drags on …
• Bob and Alice still have issues
• So they need to communicate
• So they agree to use billboards …

Billboards are Large

Write One Letter at a Time …

To post a message
Let's send another message

SELL LAVA LAMPS

Uh-Oh

SELL THE CAR

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
- Remarkably
  - We can solve R/W without mutual exclusion

Why do we care?

- We want as much of the code as possible to execute concurrently (in parallel)
- A larger sequential part implies reduced performance
- Amdahl's law: this relation is not linear...

Amdahl's Law

\[
\text{Speedup} = \frac{\text{OldExecutionTime}}{\text{NewExecutionTime}}
\]

...of computation given \( n \) CPUs instead of 1
Amdahl's Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]

Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = \frac{1}{1 - 0.6 + \frac{0.6}{10}} = 2.17
\]

Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = \frac{1}{1 - 0.8 + \frac{0.8}{10}} = 3.57
\]

Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}
\]

Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}
\]

The Moral

- The small % of a program that is hard to parallelize may have a large impact on overall speedup.