CSCI 322
Principles of Concurrent Programming

Filip Jagodzinski
Announcements

- Homework #1 has been posted
  - Due via “canvas” in 1 week
  - Architecture review

- Second lab this Friday
  - Threads
  - More “Linux refresher”
From last time ...

\[ d = 3 \]
\[ n = 4 \]
\[ \text{# of cycles needed : 6} \]

Q: What is the formula for calculating the number of cycles needed to complete \( n \) instructions for a balanced \( d \) stage pipeline?

\[ d = 4 \]
\[ n = 5 \]
\[ \text{# of cycles needed : 8} \]
From last time ...

\[ \text{cycles} = d + n - 1 \]

- \( d = 3 \)
- \( n = 4 \)
- \# of cycles needed: 6

- \( d = 4 \)
- \( n = 5 \)
- \# of cycles needed: 8
From last time ...

Balancing a pipeline’s stages most often will decrease duration and number of stalls, and throughput is

A. Increase
B. Decrease

A. Not affected
B. Increased
C. Decreased
Balancing a pipeline’s stages most often will decrease duration and number of stalls, and throughput is not affected.
From last time ...

**Thread** : the smallest sequence of instructions that can be independently scheduled and managed by the operating system (scheduler)

When more than 1 thread is running, synchronization is important (we’ve already seen this in the data hazard example illustrated previously)

**Serialization** : Event A must happen before Event B

**Mutual Exclusion** : Events A and B must NOT happen at the same time

**Concurrent** : ?
Synchronization
Synchronization – the Lunch example

Scenario: You and Bob live in different cities
You want to know who ate lunch first

Q: How do you find out if you or Bob ate lunch first?
Synchronization – the Lunch example

Scenario: You and Bob live in different cities
You want to know who ate lunch first

Q: How do you find out if you or Bob ate lunch first?

A. Call Bob at 3pm, and ask, “Heya Bob, when did you eat lunch?”
B. Call Bob at 8am, and ask him, “Heya Bob, when will you eat lunch?”
C. Both of the above will guarantee you find out for certain if you and Bob lunch at the same time
D. Neither Choices A nor B can guarantee that you and Bob eat lunch at the same time
Synchronization – the Lunch example

Scenario: You and Bob live in different cities
You want to know who ate lunch first

Q: How do you find out if you or Bob ate lunch first?

A. Call Bob at 3pm, and ask, “Heya Bob, when did you eat lunch?”
B. Call Bob at 8am, and ask him, “Heya Bob, when will you eat lunch?”
C. Both of the above will guarantee you find out for certain if you and Bob lunch at the same time
D. Neither Choices A nor B can guarantee that you and Bob eat lunch at the same time

Q: Why is A not the answer?
Synchronization – the Lunch example

Scenario: You and Bob live in different cities
You want to know who ate lunch first

Q: How do you find out if you or Bob ate lunch first?

A. Call Bob at 3pm, and ask, “Heya Bob, when did you eat lunch?”
B. Call Bob at 8am, and ask him, “Heya Bob, when will you eat lunch?”
C. Both of the above will guarantee you find out for certain if you and Bob lunch at the same time
D. Neither Choices A nor B can guarantee that you and Bob eat lunch at the same time

Q: Why is A not the answer?
If you called and ask at what time he ate, you cannot know if yours and Bob’s clocks are synchronized, so even if you ate at 12:16pm, and he said, “12:16pm,” you are still unsure
Scenario: You and Bob live in different cities
You want to know who ate lunch first

Q: Assuming Bob is willing to follow instructions, how can you guarantee that you and Bob eat lunch at the same time?
Synchronization – the Lunch example

Scenario: You and Bob live in different cities
You want to know who ate lunch first

Q: Assuming Bob is willing to follow instructions, how can you guarantee that you and Bob eat lunch at the same time?

You instruct Bob to NOT starting eating lunch until you call. Hence, you send a message.

Message passing is the foundation of many solutions to synchronizing multiple instructions streams.
Synchronization – the Lunch example

You
• Eat breakfast
• Work
• Eat Lunch
• Call Bob

Bob
• Eat breakfast
• Wait for a call
• Eat Lunch

Your thread of execution

Bob’s thread of execution
Because programs are executed sequentially (starting with the first instruction and proceeding through to the last), we know the ORDER of events for both your and Bob’s threads.

You and Bob ate lunch **Sequentially** (in the same order of events)
Synchronization – the Lunch example

You
- Eat breakfast
- Work
- Eat Lunch
- Call Bob

Bob
- Eat breakfast
- Wait for a call
- Eat Lunch

Your thread of execution
Bob’s thread of execution

The issue is that there is no way to compare events from different threads ... who ate breakfast first?
Synchronization – the Lunch example

You
• Eat breakfast
• Work
• Eat Lunch
• Call Bob

Bob
• Eat breakfast
• Wait for a call
• Eat Lunch

Your thread of execution

Bob’s thread of execution

The issue is that there is no way to compare events from different threads ... who ate breakfast first?

You and Bob ate lunch **Concurrently** (order undetermined)
Synchronization – the Lunch example

You
• Eat breakfast
• Work
• Eat Lunch
• Call Bob

Bob
• Eat breakfast
• Wait for a call
• Eat Lunch

Your thread of execution

Bob’s thread of execution

The issue is that there is no way to compare events from different threads ... who ate breakfast first?

Two events are concurrent IF you cannot tell by looking at them which happens (or will happen) first.
Synchronization – the Lunch example

You
- Eat breakfast
- Work
- Eat Lunch
- Call Bob

Bob
- Eat breakfast
- Wait for a call
- Eat Lunch

Your thread of execution

Bob’s thread of execution

The issue is that there is no way to compare events from different threads ... who ate breakfast first?

By passing a message from one thread to the next (the phone call), both threads are synchronized.
Synchronization – the Lunch example

You
• y1: Eat breakfast
• y2: Work
• y3: Eat Lunch
• y4: Call Bob

Bob
• b1: Eat breakfast
• b2: Wait for a call
• b3: Eat Lunch

“label” each event in both execution threads

Q: Assuming y4 is “matched” to b2, does that change whether you and Bob eat breakfast concurrently?
Synchronization – the Lunch example

You
- 7am: Eat breakfast
- 8am: Work
- 9am: Eat Lunch
- 10am: Call Bob

Bob
- 7am: Eat breakfast
- 10am: Wait for a call
- 11am: Eat Lunch

“label” each event in both execution threads

Q: Assuming y4 is “matched” to b2, does that change whether you and Bob eat breakfast concurrently? No

Just because the threads are synchronized at “10am,” you could have eaten breakfast at 6am or 5am, so eating breakfast still happens concurrently.
Q: In the above example, what is the motivation for sending a “message” from one thread (you) to another (Bob)?

Abstractly ... two “threads” of execution (you and Bob) are synchronized so that there “you” eat lunch before Bob

Q: On a single core computer, how can the above two threads complete/execute differently from one invocation to another?
In the above scenario, we ONLY care about you eating lunch before Bob eats his. Either you or Bob can eat breakfast first.

Q: What are the possible orderings of i1 through i7 such that i4 happens BEFORE i7?
Instruction Scheduling

You
- i1: Eat breakfast
- i2: Work
- i3: Eat Lunch
- i4: Call Bob

Bob
- i5: Eat breakfast
- i6: Wait for a call
- i7: Eat Lunch

Order A : i1, i5, i2, i6, i7, i3, i4
Order B : i1, i2, i5, i6, i7, i3, i4
Order C : i1, i2, i5, i3, i4, i6, i7
Order D : i1, i2, i3, i4, i5, i6, i7

In the above scenario, we ONLY care about you eating lunch before Bob eats his. Either you or Bob can eat breakfast first.

Q: What are the possible orderings of i1 through i7 such that i4 happens BEFORE i7?

Order A : i1, i5, i2, i6, i7, i3, i4
Order B : i1, i2, i5, i6, i7, i3, i4
Order C : i1, i2, i5, i3, i4, i6, i7
Order D : i1, i2, i3, i4, i5, i6, i7

...
In the above scenario, we ONLY care about you eating lunch before Bob eats his. Either you or Bob can eat breakfast first.

Q: What are the possible orderings of i1 through i7 such that i4 happens BEFORE i7?
A pipeline allows “more” instructions to be executed “concurrently” ....
Instruction Scheduling

A pipeline allows “more” instructions to be executed “concurrently” ....

A multiscalar architecture duplicates the pipeline units
A pipeline allows “more” instructions to be executed “concurrently” ....

A multiscalar architecture duplicates the pipeline units

Duplicating the register file and/or controls units allows a computer to thread (multithreading)

(much more on this later on in the quarter)
public class SillyMath{
    public static void main (String args[]){
        int i=3, j=46, k=4, l=76, prod, sum;
        double sRoot;
        prod = i * j;
        sum = k + l;
        sRoot = Math.pow((double)76,0.3);
    }
}
public class SillyMath{
    public static void main (String args[]){
        int i=3, j=46, k=4, l=76, prod, sum;
        double sRoot;
        prod = i * j;
        sum = k + l;
        sRoot = Math.pow((double)76,0.3);
    }
}
### Instruction Scheduling

```
public class SillyMath{
    public static void main (String args[]){
        int i=3, j=46, k=4, l=76, prod, sum;
        double sRoot;
        prod = i * j;
        sum = k + l;
        sRoot = Math.pow((double)76,0.3);
    }
}
```
With Threading
Synchronization ...

You: eat breakfast, work, eat lunch, call Bob

Bob: eat breakfast, wait for call, eat lunch

Lunch synchronization by message passing ...

This ensured the order of who eats lunch first/second
Scenario: You and Bob are bitter enemies (no sending of messages), AND you do NOT want to be in the lunch room at the same time ... how can this be achieved?

You: eat breakfast, work, eat lunch, call Bob

Bob: eat breakfast, wait for call, eat lunch
Synchronization ...

Keep a conch shell in the office ... ONLY if you have the conch shell can you go into the lunch room

You: eat breakfast, work, eat lunch, call Bob

Bob: eat breakfast, wait for call, eat lunch
Synchronization ...

You (y)
- Eat breakfast
- Retrieve conch shell
- Eat Lunch
- Put conch shell back

Bob (b)
- Eat breakfast
- Take conch shell
- Eat Lunch
- Put conch shell back
Synchronization ...

You (y)
- Eat breakfast
- Retrieve conch shell
- Eat Lunch
- Put conch shell back

Bob (b)
- Eat breakfast
- Take conch shell
- Eat Lunch
- Put conch shell back

Breakfast : done concurrently
Lunch : done _________
You (y)
- Eat breakfast
- Retrieve conch shell
- Eat Lunch
- Put conch shell back

Bob (b)
- Eat breakfast
- Take conch shell
- Eat Lunch
- Put conch shell back

Breakfast: done concurrently
Lunch: done concurrently, also
Synchronization ...

You (y)
- Eat breakfast
- Retrieve conch shell
- Eat Lunch
- Put conch shell back

Bob (b)
- Eat breakfast
- Take conch shell
- Eat Lunch
- Put conch shell back

Goal: enforce mutual exclusion so that only one person at any time can be eating lunch

Q: How might you enforce the “conch” shell approach?
Shared Variables

You (y)
- Eat breakfast
- Retrieve conch shell
- Eat Lunch
- Put conch shell back

Bob (b)
- Eat breakfast
- Take conch shell
- Eat Lunch
- Put conch shell back

Goal: enforce mutual exclusion so that only one person at any time can be eating lunch

Conch = \{0,1,2,3,4,...\}

Q: How might you enforce the “conch” shell approach?
Shared Variables – concurrent write

Thread A

i1: x=5
i2: print x

Thread B

i3: x=7
<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>i1: x=5</strong></td>
<td><strong>i3: x=7</strong></td>
</tr>
<tr>
<td><strong>i2: print x</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Q:** What path (scheduling sequence) prints a 5 and sets the final value to 5?

**Q:** What path (scheduling sequence) prints a 7 and sets the final value to 7?
Shared Variables – concurrent write

Q: What path (scheduling sequence) prints a 5 and sets the final value to 5?

Q: What path (scheduling sequence) prints a 7 and sets the final value to 7?
Q: What is the difference between an update and a write?

Thread A

i1: x=5
i2: print x

These are examples of concurrent writes

Thread B

i3: x=7
Q: What is the difference between an update and a write?

Thread A

```
i1: x=5
i2: print x
```

Thread B

```
i3: x=7
```

These are examples of concurrent writes

```
i1: count = count + 1
i2: count = count +1
```
Shared Variables – concurrent updates

Thread A

i1: x=5
i2: print x

These are examples of concurrent **writes**

Thread B

i3: x=7

**Q: What is the difference between an update and a write?**

i1: count = count + 1

**An update** reads a value in a variable, computes a new value based on the “old” value, and writes the new value to the variable

i2: count = count +1

**Q: What is a synchronization error that might arise among updates for multiple threads?**
Shared Variables – concurrent updates

Remember that “executing” an instruction involves multiple pipeline steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.
Shared Variables – concurrent updates

Thread A

i1: count = count + 1

a1: load count
a2: add 1
a3: store count

Thread B

i2: count = count + 1

b1: load count
b2: add 1
b3: store count

Remember that “executing” an instruction involves multiple pipeline steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

Assume initial value of count = 4

Q: What is final value of count if execution order is a1 < b1 < a2 < b2 < a3 < b3?
Shared Variables – concurrent updates

Thread A

i1: count = count + 1
a1: load count
a2: add 1
a3: store count

Register/ALU “view”

Thread B

i2: count = count + 1
b1: load count
b2: add 1
b3: store count

Remember that “executing” an instruction involves multiple pipeline steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

Assume initial value of count = 4

Q: What execution order do we need to impose so thread A’s “update” does not interfere with thread B’s update?
Shared Variables – concurrent updates

Thread A

\[
\text{i1: count} = \text{count} + 1
\]

\[
\begin{align*}
\text{a1: load count} \\
\text{a2: add 1} \\
\text{a3: store count}
\end{align*}
\]

Thread B

\[
\text{i2: count} = \text{count} + 1
\]

\[
\begin{align*}
\text{b1: load count} \\
\text{b2: add 1} \\
\text{b3: store count}
\end{align*}
\]

Register/ALU “view”

Assume initial value of count = 4

Q: What execution order do we need to impose so thread A’s “update” does not interfere with thread B’s update?

Sample “solution” : a1 < a2 < a3 < b1 < b2 < b3

Sample “solution” : b1 < b2 < b3 < a1 < a2 < a3

Q: What similarities do these two “solutions” share?
Remember that “executing” an instruction involves multiple pipeline steps, including loading registers, loading ALUs, executing ALUs, fetching results from ALU, etc.

An operation or instruction stream that CANNOT be interrupted is called atomic.

Sample “solution” : a1 < a2 < a3 < b1 < b2 < b3
Sample “solution” : b1 < b2 < b3 < a1 < a2 < a3
Up Next ...

Semaphores