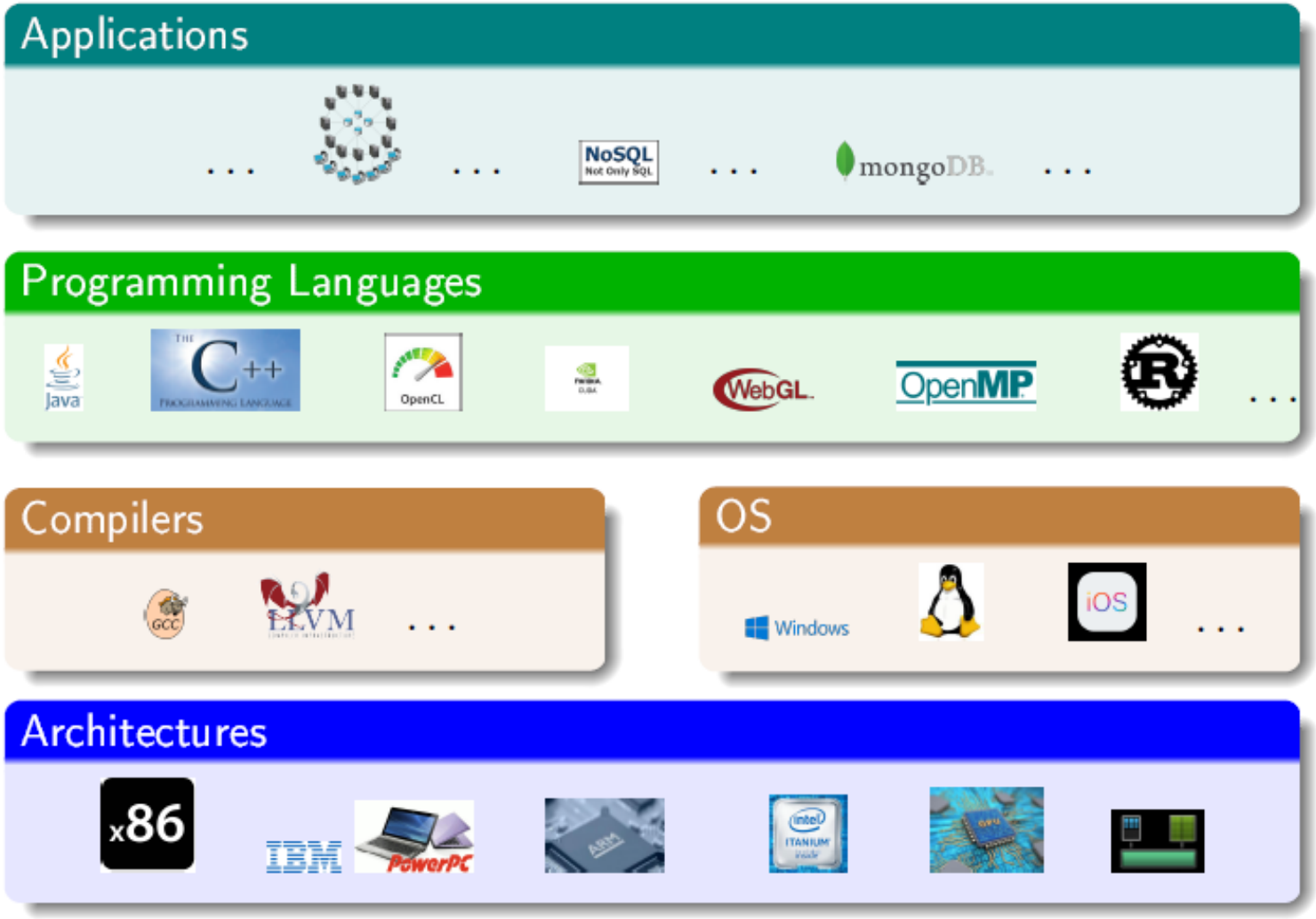
# Selected Topics in Concurrency
# COL 869

## Relaxed Memory Concurrency

# Concurrent Computing



# Use of Concurrency

**Pros.**

- Concurrency improves performance

**Cons.**

- Concurrent programming is hard
- May result in tricky bugs in programs
- Requires careful analysis

# Concurrent Programming

- Program with multiple threads
- Shared memory access

# Concurrent Programming

- Program with multiple threads
- Shared memory access

Given a concurrent program and an outcome, is it a correct outcome?

# Concurrent Programming

- Program with multiple threads
- Shared memory access

Given a concurrent program and an outcome, is it a correct outcome?

How does the program execute?

# Question

Given a program and an outcome, is it a correct outcome?

Depends on

*Program execution*

# Example

X=0

| | |
|---|---|
| X =X+1; | X=X+1; |

**What is the final value of X ?**

# Example

X=0

| | |
|---|---|
| X =X+1; | X =X+1; |

**What is the final value of X ?**

**Desired : X=2**

# Execution

X=0

```
a=X;  // 0
a = a+1;
X = a;
```

```
b=X;
b = b+1;
X = b;
```

# Execution

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X;
b = b+1;
X = b;
```

# Execution

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X; // 1
b = b+1;
X = b;
```

# Execution

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X; // 1
b = b+1;
X = b;  // 2
```

**Output:** X = 2

# Another Execution

X=0

| | |
|---|---|
| a=X; // 0 | b=X; |
| a = a+1; | b = b+1; |
| X = a; | X = b; |

# Another Execution

X=0

| | |
|---|---|
| a=X; // 0 | b=X; // 0 |
| a = a+1; | b = b+1; |
| X = a; | X = b; |

# Another Execution

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X; // 0
b = b+1;
X = b;
```

# Another Execution

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X; // 0
b = b+1;
X = b;  // 1
```

**Output:** X = 1

# Example

X=0

| | |
|---|---|
| X= X+1; | X=X+1; |

**What is the final value of X ?**

**Desired : X=2**

***Reality :*** **X = 1 / 2**

# What about This Execution?

X=0

```
a=X;  // 0
a = a+1;
X = a;
```

```
b=X;
b = b+1;
X = b;
```

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X;
b = b+1;
X = b;
```

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X; // 0
b = b+1;
X = b;
```

X=0

```
a=X;  // 0
a = a+1;
X = a; // 1
```

```
b=X; // 0
b = b+1;
X = b;  // 1
```

# Understanding Concurrency

Key Question:

*How does concurrent programs execute?*

*Given an execution of a program, is it correct?*

Answer may vary across platforms.

# *How does concurrent programs execute?*

*"…the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* ~ Leslie Lamport

# *How does concurrent programs execute?*

*"…the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* ~ Leslie Lamport

Sequential consistency (SC)

# *How does concurrent programs execute?*

*"…the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."* ~ Leslie Lamport

Sequential consistency (SC) ==> Interleaving Execution

# Earlier Example ➔ Interleaving Execution

X=0

| | |
|---|---|
| a=X; // 0 | b=X; |
| a = a+1; | b = b+1; |
| X = a; | X = b; |

X=0

| | |
|---|---|
| a=X; // 0 | b=X; // 0 |
| a = a+1; | b = b+1; |
| X = a; | X = b; |

X=0

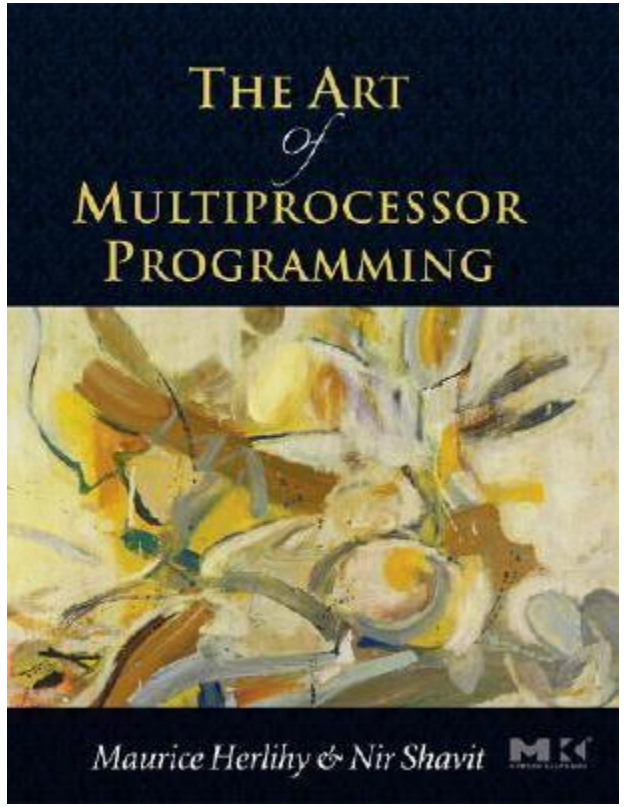
| | |
|---|---|
| a=X; // 0 | b=X; // 0 |
| a = a+1; | b = b+1; |
| X = a; // 1 | X = b; |

X=0

| | |
|---|---|
| a=X; // 0 | b=X; // 0 |
| a = a+1; | b = b+1; |
| X = a; // 1 | X = b; // 1 |

# Sequential Consistency

Foundation of many concurrent algorithms



# Example: Dekker's Algorithm

wants_to_enter[0] ← false, wants_to_enter[1] ← false, turn ← 0  // or 1

```
wants_to_enter[0] ← true
 while (wants_to_enter[1]) {
   if (turn ≠ 0) {
     wants_to_enter[0] ← false
     while (turn ≠ 0) { // busy wait }
     wants_to_enter[0] ← true
   }}
 // critical section
 ...
 turn ← 1
 wants_to_enter[0] ← false
 // remainder section
```

```
wants_to_enter[1] ← true
 while (wants_to_enter[0]) {
   if (turn ≠ 1) {
     wants_to_enter[1] ← false
     while (turn ≠ 1) { // busy wait }
     wants_to_enter[1] ← true
   }}
 // critical section
 ...
 turn ← 0
 wants_to_enter[1] ← false
 // remainder section
```

# Example: Dekker's Algorithm

wants_to_enter[0] ← false, wants_to_enter[1] ← false, turn ← 0  // or 1

```
wants_to_enter[0] ← true
 while (wants_to_enter[1]) {…}
 // critical section
```

```
wants_to_enter[1] ← true
 while (wants_to_enter[0]) {…}
 // critical section
```

Ensures mutual exclusion

# Representative Example: Store Buffer (SB)

X=Y=0

| | |
|---|---|
| X = 1; | Y = 1; |
| a = Y; | b = X; |

# Program: Store Buffer (SB)

X=Y=0

| | |
|---|---|
| X = 1; | Y = 1; |
| a = Y; | b = X; |

**Outcome:**

a=0, b=1

# Program: Store Buffer (SB)

X=Y=0

| | |
|---|---|
| X = 1; | Y = 1; |
| a = Y; | b = X; |

**Outcome:**

a=0, b=1

a=1, b=1

# Program: Store Buffer (SB)

X=Y=0

| | |
|---|---|
| X = 1; | Y = 1; |
| a = Y; | b = X; |

**Outcome:**

a=0, b=1

a=1, b=1

a=1, b=0

# Program: Store Buffer (SB)

X=Y=0

| | |
|---|---|
| X = 1; | Y = 1; |
| a = Y; | b = X; |

*Let's execute the program…*

# Program: Store Buffer (SB)

X=Y=0

| | |
|---|---|
| X = 1; | Y = 1; |
| a = Y; | b = X; |

**Outcome:**

a=0, b=1

a=1, b=1

a=1, b=0

a=0, b=0

# Program: Store Buffer (SB)

X=Y=0

| | |
|---|---|
| X = 1; | Y = 1; |
| a = Y; | b = X; |

a=0, b=0

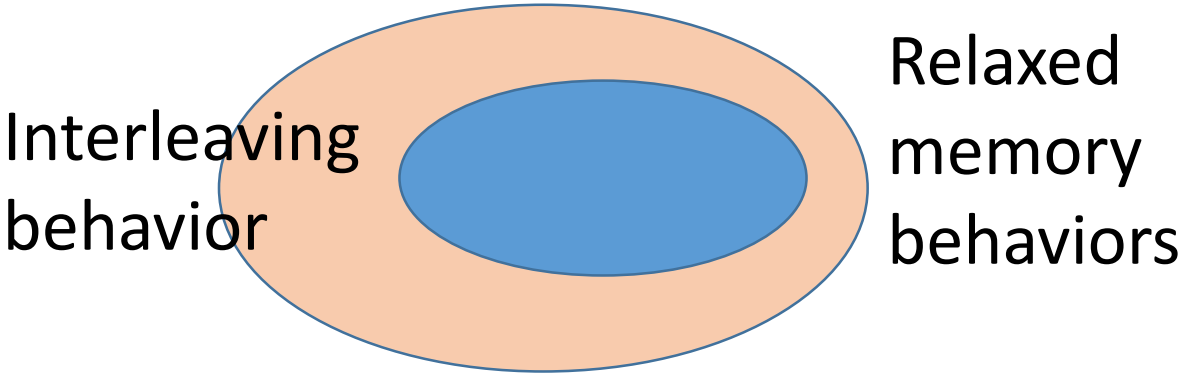
Cannot be explained by thread interleaving.

*Relaxed memory concurrency*

# Relaxed Memory Concurrency

Traditionally: Concurrency = thread interleaving

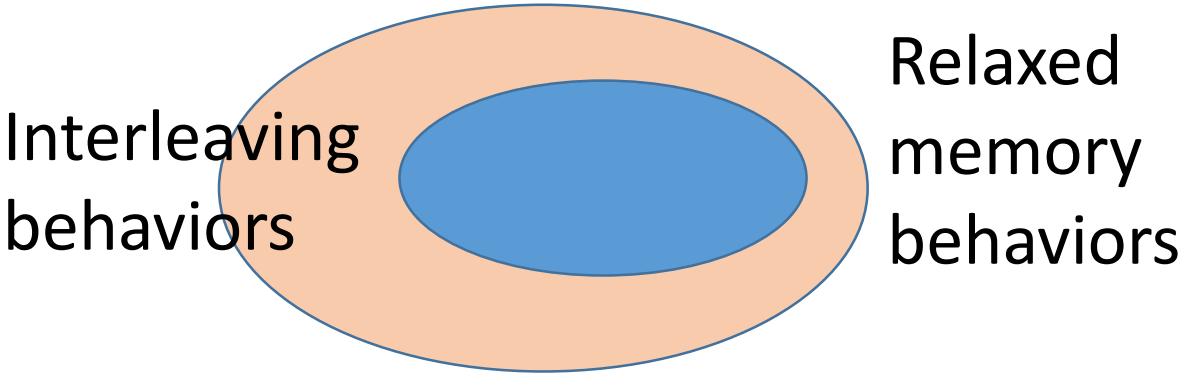
Reality: more behaviors than thread interleaving



# Relaxed Memory Concurrency

Traditionally: Concurrency = thread interleaving

Reality: more behaviors than thread interleaving




**Reasons:**

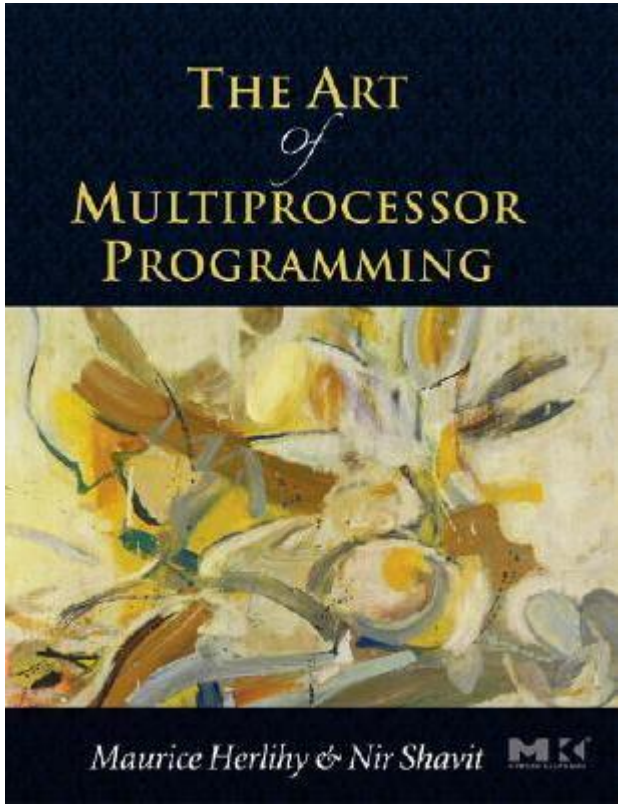
Compiler: reorder instructions

Hardware:

- Out of order execution
- Data movement/communication is not instantaneous

# Is it a problem?

Programmability: *Sequential consistency assumption may not hold !*



# Is it a problem?

Programmability: *Sequential consistency assumption may not hold !*

Tools: *Verification, testing, analysis tools assume interleaving*

# Is it a problem?

Programmability: *Sequential consistency assumption may not hold !*

Tools: *Verification, testing, analysis tools assume interleaving*

Compilers: *Not sure about correct transformations*

- *Suboptimal compilation*
- *compiler bugs*

# How to address the Problem?

Programmability: *Sequential consistency assumption may not hold !*

Tools: *Verification, testing, analysis tools assume interleaving*

Compilers: *Not sure about correct transformations*
- *Suboptimal compilation*
- *compiler bugs*

Study **relaxed memory concurrency**

***This course*** ☺

# Content

Semantics

- Programming languages (C/C++)
- Architectures (x86, ARM, PowerPC)

Compiler Correctness

- Compiler optimizations
- Mappings to architectures

Program Correctness

- Data-race-freedom properties

# Course Logistics

Visit:
http://www.cse.iitd.ac.in/~soham/COL869/page.html