COMP 322: Fundamentals of Parallel Programming

Lecture 34: Volatile Variables, Memory Consistency Models

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #34: Branching in SIMD code

Consider SIMD execution of the following pseudocode with 8 threads. Assume that each call to doWork(x) takes x units of time, and ignore all other costs. How long will this program take when executed on 8 GPU cores, taking into consideration the branching issues discussed in Slide 13?

```
1. int tx = threadIdx.x; // ranges from 0 to 7
2. if (tx % 2 = 0) {
3.    S1: doWork(1); // Computation S1 takes 1 unit of time
4. }
5. else {
6.    S2: doWork(2); // Computation S2 takes 2 units of time
7. }
```

Solution: 3 units of time (WORK=12, CPL=3)

Memory Visibility

- Basic question: if a memory location L is written by statement S1 in thread T1, when is that write guaranteed to be visible to a read of L in statement S2 of thread T2?
- HJ answer: whenever there is a directed path of edges from S1 in S2 in the computation graph
  - Computation graph edges are defined by semantics of parallel constructs: async, finish, async-await, futures, phasers, isolated, object-based isolation
- Java answer: whenever there is a "happens-before" relation between S1 and S2

  ==> Should we define "happens-before" using time or ordering?

  - Is there such a thing as universal global time?

Troublesome example

```
1. public class NoVisibility {
2.  private static boolean ready;
3.  private static int number;
4.
5.  private static class ReaderThread extends Thread {
6.    public void run() {
7.      while (!ready) Thread.yield()
8.      System.out.println(number)
9.    }
10.  }
11.
12.  public static void main(String[] args) {
13.    new ReaderThread().start();
14.    number = 42;
15.    ready = true;
16.  }
17. }
```

No happens-before ordering between main thread and ReaderThread
==> ReaderThread may loop forever OR may print 42 OR may print 0 !!

Volatile Variables available in Java

- Java provides a "light" form of synchronization/fence operations in the form of `volatile` variables (fields)
- Volatile variables guarantee visibility
  - —Reads and writes of volatile variables should be assumed to occur in isolated blocks
  - —Adds serialization edges to computation graph due to isolated read/write operations on same volatile variable
- Incrementing a volatile variable (++v) is <u>not thread-safe</u>
  - —Increment operation looks atomic, but isn't (read and write are two separate operations)
- Volatile variables are best suited for flags that have no dependencies e.g.,

```
volatile boolean asleep;
foo() { ... while (! asleep) ++sheep; ... }
```

— WARNING: In the absence of volatile declaration, the above code can legally be transformed to the following

```
boolean asleep;
foo(){ boolean temp=asleep; ... while (! temp) ++sheep; ... }
```

Troublesome example fixed with volatile declaration

```
1. public class NoVisibility {
2.  private static volatile boolean ready;
3.  private static volatile int number;
4.
5.  private static class ReaderThread extends Thread {
6.    public void run() {
7.      while (!ready) Thread.yield()
8.      System.out.println(number)
9.    }
10.  }
11.
12.  public static void main(String[] args) {
13.    new ReaderThread().start();
14.    number = 42;
15.    ready = true;
16.  }
17. }
```

Declaring number and ready as volatile ensures happens-before-edges: 14-->15-->7-->8, thereby ensuring that only 42 will be printed

Data Races are usually Errors, but not always

- Example of Data Race Error

```
1.  for ( p = first; p != null; p = p.next)
2.      async p.x = p.y + p.z;
3.  for ( p = first; p != null; p = p.next)
4.      sum += p.x;
```

- Example of intentional (benign) data race
  - Search algorithm that returns any match (need not be the first match)

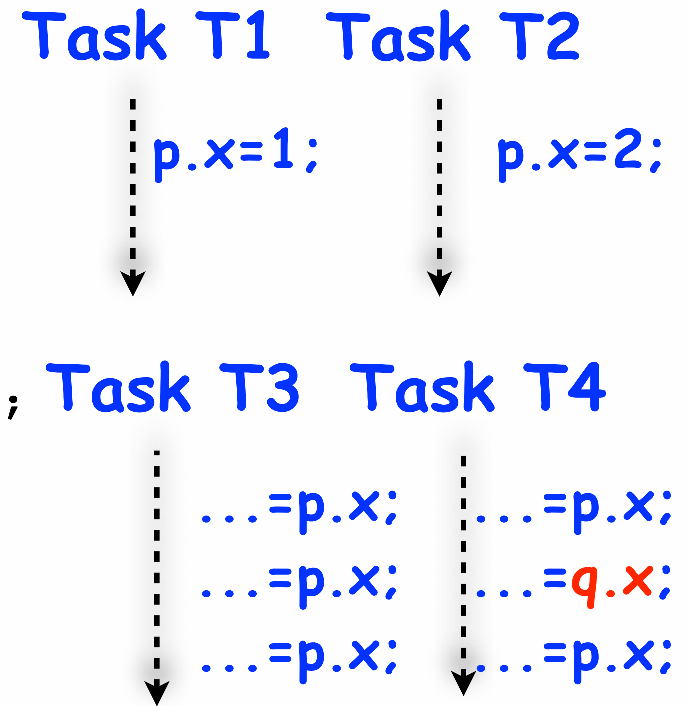
```
1.  static int index = -1; // static field
2.  . . .
3.  finish for (int i = 0; i <= N - M; i++) async {
4.    for (j = 0; j < M; j++)
5.      if (text[i+j] != pattern[j]) break;
6.    if (j == M) index = i;             // found at offset i
7.  }
```

- In both cases, the semantics of data races still needs to be fully specified

Semantics of Data Races

<u>Example HJ program:</u>

```
1. p.x = 0; q = p;
2. async p.x = 1; // Task T1
3. async p.x = 2; // Task T2
4. async { // Task T3
5.   System.out.println("First read = " + p.x);
6.   System.out.println("Second read = " + p.x);
7.   System.out.println("Third read = " + p.x)
8. }
9. async { // Task T4
10.  System.out.println("First read = " + p.x);
11.  System.out.println("Second read = " + q.x);
12.  System.out.println("Third read = " + p.x);
13.}
```



Can the following values be printed by tasks T3 & T4?

T3: 0, 0, 0
T4: 1, 2, 1

Program Order != Reality, for Racy Programs

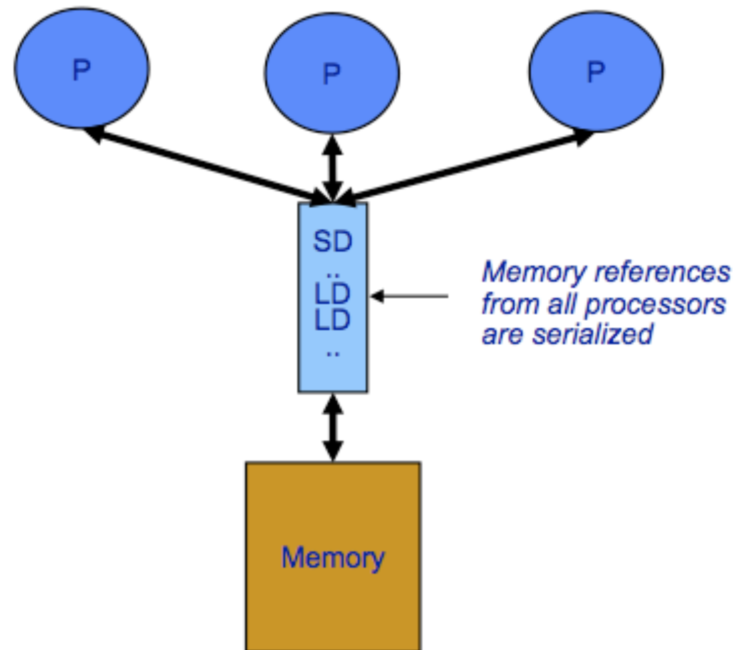
- Programmer's view:
  - —Everything happens in the order I indicate through the code statements that I write
- Reality (JVM/compiler & hardware processor):
  - —Everything happens in whatever order yields best performance, so long as the program(mer) can't tell the difference
- For data-race-free programs
  - —Program order can't be distinguished from actual order
- For "racy" programs
  - —Different tasks can see different actions in memory
    - At different times
    - In different orders

Memory Consistency Models

- A memory consistency model, or <u>memory model</u>, is the part of a programming specification that defines what write values a read may observe
  - — For data-race-free programs, all memory models are identical since each read can observe exactly one write value
  - ⇒ *if you only write data-race-free programs, you don't have to worry about memory models!*
- Question: why do different memory models have different rules for data races?
- Answer: because different memory models are useful at different levels of software
  - — Sequential Consistency (SC)
    - – Useful for implementing low-level synchronization primitives e.g., operating system services
  - — Java Memory Model (JMM)
    - – Useful for implementing task schedulers e.g., HJ runtime
  - — Habanero Java Memory Model (HJMM)
    - – Useful for specifying semantics at application task level e.g., HJ programs
    - – Derived from past work on "Location Consistency" memory model

HJMM
JMM
SC

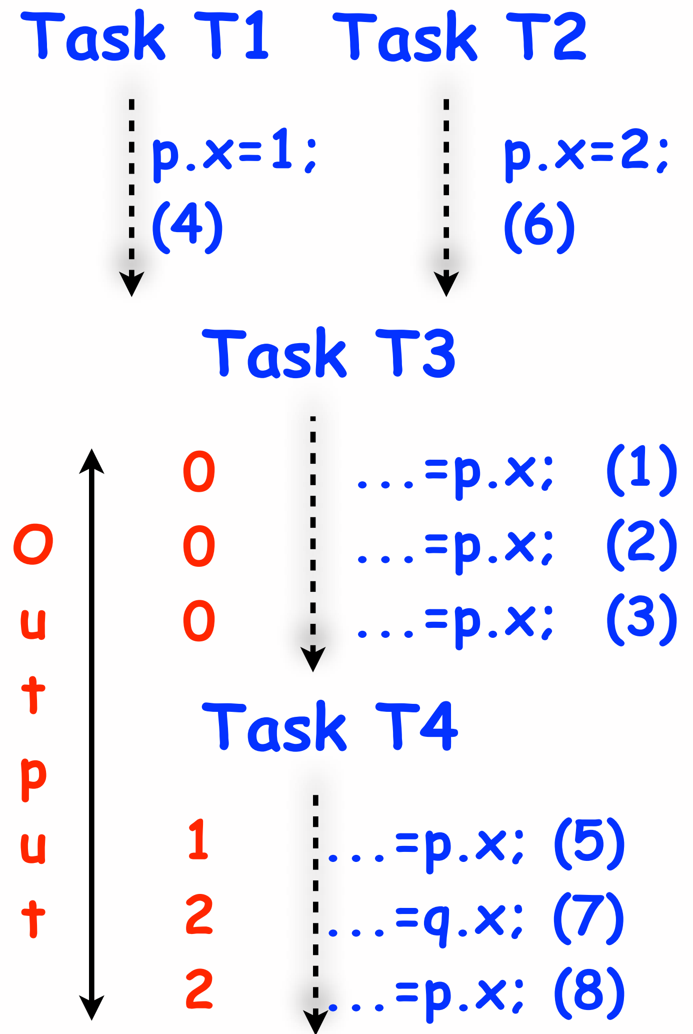
Sequential Consistency Memory Model



[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

Sequential Consistency (SC) Memory Model

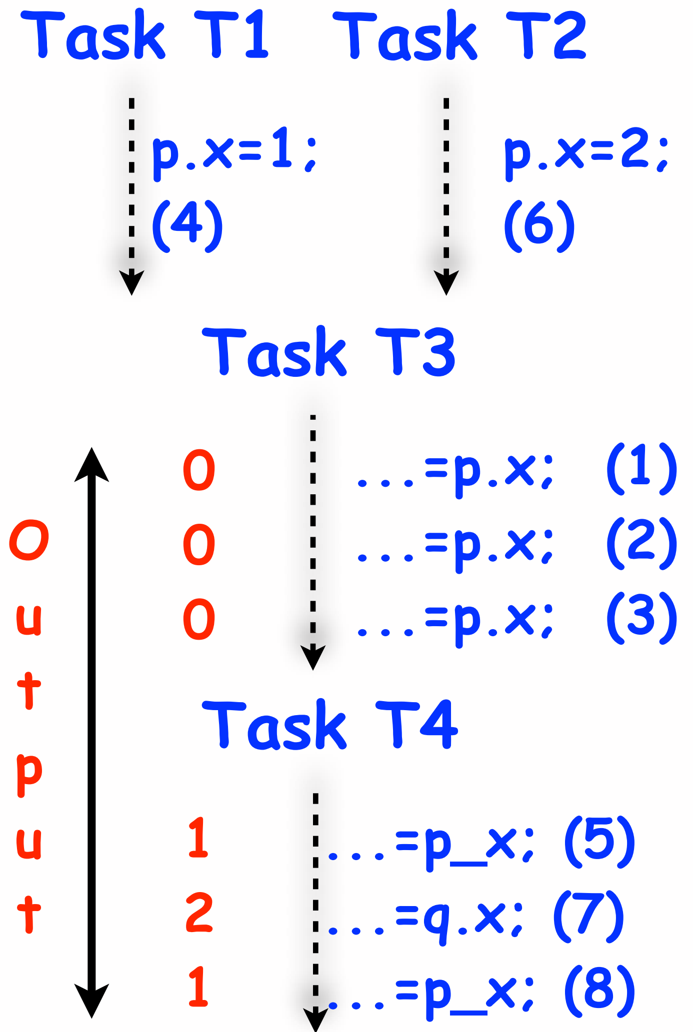
- **SC constrains all memory operations across all tasks**
  - Write → Read
  - Write → Write
  - Read → Read
  - Read → Write
- **Simple model for reasoning about data races at the hardware level, but may lead to counter-intuitive behavior at the application level e.g.,**
  - A programmer may perform modular code transformations for software engineering reasons without realizing that they are changing the program's semantics



Consider a “reasonable” code transformation performed by a programmer

<u>Example HJ program:</u>

```
1.  p.x = 0; q = p;
2.  async p.x = 1; // Task T1
3.  async p.x = 2; // Task T2
4.  async { // Task T3
5.    System.out.println("First read = " + p.x);
6.    System.out.println("Second read = " + p.x);
7.    System.out.println("Third read = " + p.x)
8.  }
9.  async { // Task T4
10.   // Assume programmer doesn't know that p=q
11.   int p_x = p.x;
12.   System.out.println("First read = " + p_x);
13.   System.out.println("Second read = " + q.x);
14.   System.out.println("Third read = " + p_x);
15. }
```

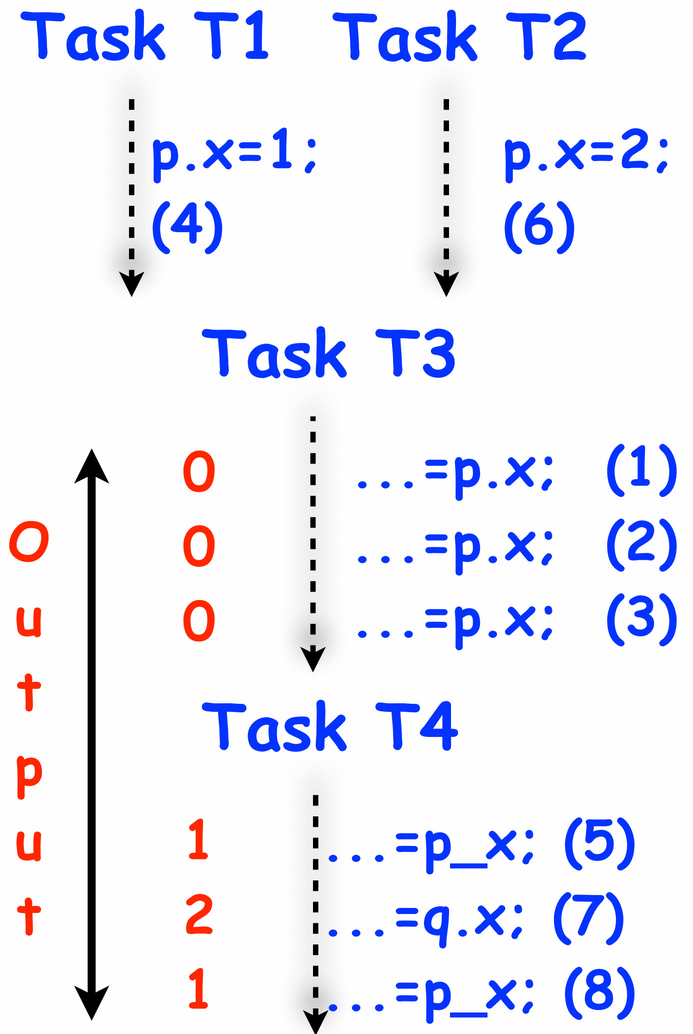


Consider a "reasonable" code transformation performed by a programmer

Example HJ program:

```
1. p.x = 0; q = p;
2. async p.x = 1; // Task T1
3. async p.x = 2
4. async {
5.    Sy                                        x);
6.    Sy                                        x);
7.    Syst                                      .x)
8. }
9. async { // Task T4
10.   // Assume programmer doesn't know that p=q
11.   int p_x = p.x;
12.   System.out.println("First read = " + p_x);
13.   System.out.println("Second read = " + q.x);
14.   System.out.println("Third read = " + p_x);
15.}
```

This reasonable code transformation resulted in an illegal output, under the SC model!



The Java Memory Model (JMM) and the Habanero-Java Memory Model (HJMM)

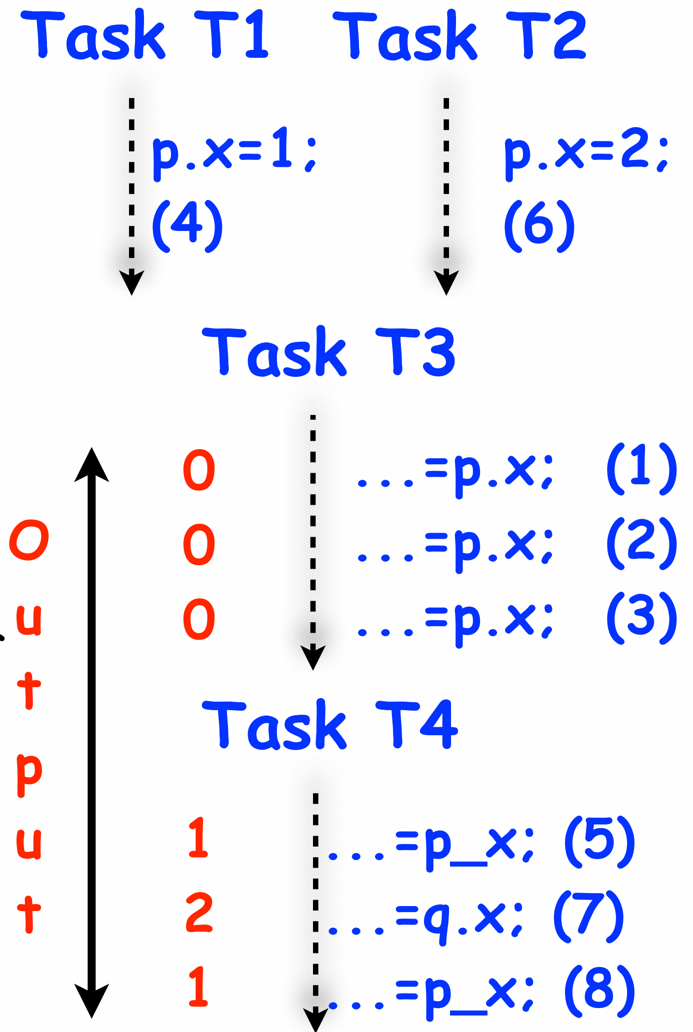
- **Conceptually simple:**
  - Every time a variable is written, the value is added to the set of "most recent writes" to the variable
  - A read of a variable is allowed to return ANY value from this set
- **The JMM defines the rules by which values in the set are removed**
  - By using ordering relationships ("happens-before") similar to the Computation Graph to determine when a value must be overwritten
- **HJMM has weaker ordering rules for HJ's "isolated" statements, compared to Java's "synchronized" blocks**
- **Programmer's goal: through proper use of synchronization**
  - Ensure the absence of data races, in which case this set will never contain more than one value and SC, JMM, HJMM will all have the same semantics

Code Transformation Example

Example HJ program:

```
1.  p.x = 0; q = p;
2.  async p.x = 1; // Task T1
3.  async p.x = 2
4.  async {
5.     Sy                                   x);
6.     Sy                                   x);
7.     Syst                                 .x)
8.  }
9.  async { // Task T4
10.    // Assume programmer doesn't know that p=q
11.    int p_x = p.x;
12.    System.out.println("First read = " + p_x);
13.    System.out.println("Second read = " + q.x);
14.    System.out.println("Third read = " + p_x);
15. }
```

This output is legal under the JMM and HJMM!

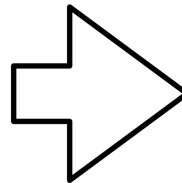


Semantics-Preserving Code Transformations in Sequential Programs

- **A Code Transformation is said to be semantics-preserving if the transformed program, P', exhibits the same Input-Output behavior as the original program, P**
- **For <u>sequential</u> programs, many local transformations are guaranteed to be semantics-preserving regardless of the context**
  - **e.g., replacing the second access of an object field or array element by a local variable containing the result of the first access, if there are no possible updates between the two accesses**

P

```
1.  static void foo(T p, T q) {
2.    System.out.println(p.x);
3.    System.out.println(q.x);
4.    System.out.println(p.x);
5.  }
```



P'

```
1.  static void foo(T p, T q) {
2.    int xLocal = p.x;
3.    System.out.println(xLocal);
4.    System.out.println(q.x);
5.    System.out.println(xLocal);
6.  }
```

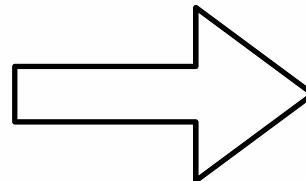
Semantics-Preserving Code Transformations in Parallel Programs

- <u>Question:</u> What should we expect if we perform a Code Transformation on a sequential region of a parallel program, if the transformation is known to be semantics-preserving for sequential programs?
- <u>Answer:</u> The transformation should be semantics-preserving for the parallel program if there are no data races. Otherwise, it depends on the memory model!

P

```
1.   p.x = 0; q = p;
2.   async p.x = 1;
3.   async p.x = 2;
4.   async foo(p, p);
5.   async foo(p, q);
6.   . . .
7.   static void foo(T p, T q) {
8.     System.out.println(p.x);
9.     System.out.println(q.x);
10.    System.out.println(p.x);
11.  }
```

Is this a legal transformation?



It may result in the following output:
0 0 0
1 2 1

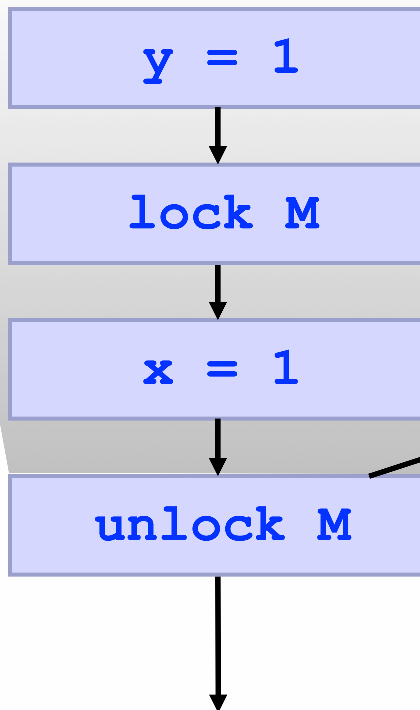
P'

```
1.   p.x = 0; q = p;
2.   async p.x = 1;
3.   async p.x = 2;
4.   async foo(p, p);
5.   async foo(p, q);
6.   . . .
7.   static void foo(T p, T q) {
8.     int xLocal = p.x
9.     System.out.println(xLocal);
10.    System.out.println(q.x);
11.    System.out.println(xLocal);
12.  }
```

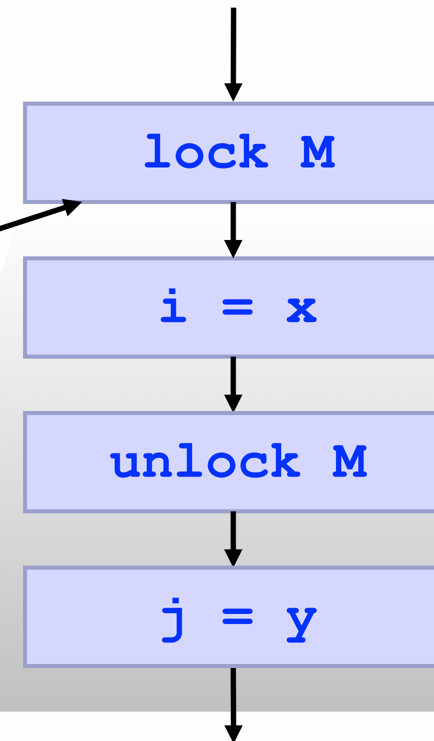
==> *Code transformation is legal for JMM & HJMM, but not for SC !*

When are actions visible and ordered with other Threads in the JMM?

Thread 1



Thread 2



Everything before the unlock is visible to everything after the matching lock in the JMM

lock/unlock operations can come from synchronized statement or from explicit calls to locking libraries

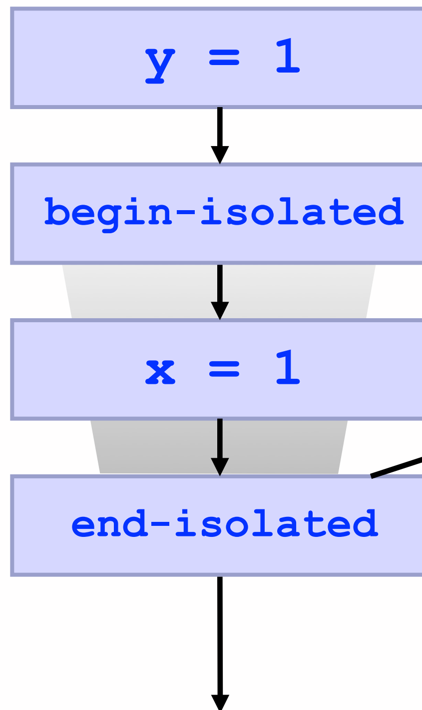
Troublesome example fixed with empty synchronized statements instead of volatile (JMM)

```
1. public class NoVisibility {
2.  private static boolean ready;
3.  private static int number;
4.  private static final Object a = new Object();
5.
6.  private static class ReaderThread extends Thread {
7.    public void run() {
8.      synchronized(a){}
9.      while (!ready) { Thread.yield(); synchronized(a){} }
10.     System.out.println(number);
11.    }
12.  }
13.
14.  public static void main(String[] args) {
15.    new ReaderThread().start();
16.    number = 42;
17.    ready = true; synchronized(a){}
18.  }
19. }
```

Empty synchronized statement is NOT a no-op in Java. It acts as a memory "fence".

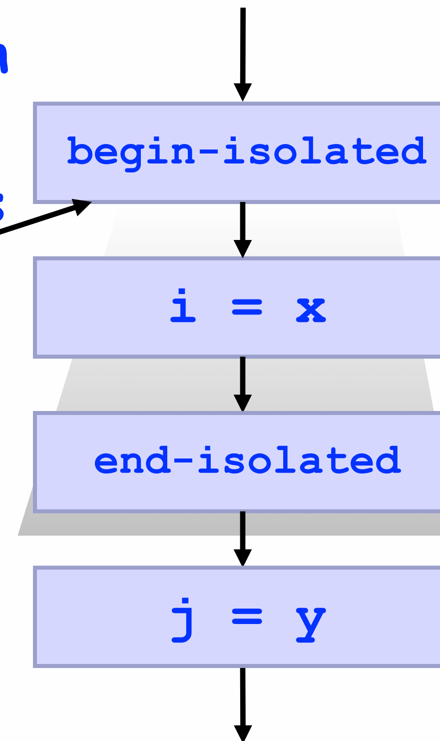
When are actions visible and ordered with other Threads in the HJMM?

Thread 1



Everything within the first isolated region is visible to everything in the second isolated region, in the HJMM

Thread 2



Empty isolated statements are no-ops in HJ

```
1. public class NoVisibility {
2.  private static boolean ready;
3.  private static int number;
4.
5.  private static class ReaderThread extends Thread {
6.    public void run() {
7.      isolated{}
8.      while (!ready) { Thread.yield(); isolated{} }
9.      System.out.println(number);
10.    }
11.  }
12.
13.  public static void main(String[] args) {
14.    new ReaderThread().start();
15.    number = 42;
16.    ready = true; isolated {}
17.  }
18. }
```

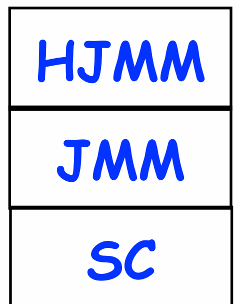
Empty isolated statement is a no-op in HJ. ReaderThread may loop forever OR may print 42 OR may print 0.

Better to use explicit synchronization in HJ instead

```
1. public class NoVisibility {
2.   private static boolean ready;
3.   private static int number;
4.   private static DataDrivenFuture<Boolean>
5.       readyDDF = new DataDrivenFuture<Boolean>();
6.
7.   public static void main(String[] args) {
8.     async await(readyDDF){ System.out.println(number); }
9.     number = 42;
10.    readyDDF.put(true);
11.  }
12. }
```

Summary of Memory Model Discussion

- Memory model specifies rules for what write values can be seen by reads in the presence of data races
  - In the absence of data races, program semantics specifies exactly one write for each read
- A local code transformation performed on a sequential code region may be semantics-preserving for sequential programs, but not necessarily for parallel programs
  - Stronger memory models (e.g., SC) are more restrictive about permissible read sets than weaker memory models (e.g., JMM, HJMM), and thus more restrictive about allowing transformations
- Different memory models are appropriate for different levels of the software stack
  - e.g., SC at the OS/HW level, JMM at the thread level, HJMM at the task level



Worksheet #35: Double Checked Locking Idiom in Java

Name: ______________________ Netid: ______________________

Consider two threads calling the getHelper() method in parallel:

1) Can you construct a possible data race if they call the unoptimized version of getHelper() in lines 3-8?

2) Can you construct a possible data race if they call the optimized version of getHelper() in lines 12-21?

3) How will your answer to 2) change if the helper field in line 11 was declared as volatile?

COMP 322, Spring 2014 (V.Sarkar)

Worksheet #35 (contd)

```
1. class Foo { //unoptimized version
2.    private Helper helper; // Singleton pattern
3.    public synchronized Helper getHelper() {
4.        if (helper == null) {
5.            helper = new Helper();
6.        }
7.        return helper;
8.    }
9.    . . .

10.class Foo { //Optimized version
11.    private Helper helper; // Singleton pattern
12.    public Helper getHelper() {
13.        if (helper == null) {
14.            synchronized(this) {
15.                if (helper == null) {
16.                    helper = new Helper();
17.                }
18.            }
19.        }
20.        return helper;
21.    }
22.    . . .
```

Announcements

- Graded midterms can be picked up from Melissa Cisneros in Duncan Hall room 3122 (mcisnero@rice.edu)
- Homework 5 due by 11:55pm on Monday, April 21st
  - Send email to comp322-staff@rice.edu if you plan to use slip days
- Homework 6 assigned today
  - Written-only, no programming assignments
  - Due by 11:55pm on April 25th, penalty-free extension till May 2nd
- No lab next week
- April 25th is last day of classes
  - Exam 2 will be handed out on April 25th
    - Take-home exam, due by May 2nd