COMP 322: Fundamentals of Parallel Programming

Lecture 5: Data Races, Determinism, Memory Consistency Models

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

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

Solution to Worksheet #4: how many processors should we use for ArraySum?

For ArraySum on P processors and input array size, S,

Speedup(S,P) = T(S,1)/T(S,P) = S/(S/P + $\log_2$(S))

- Question: For a given S, what value of P should we choose to obtain Efficiency(P) = 0.5? Recall that Efficiency(P) = 0.5 $\Rightarrow$ Speedup(S,P)/P = 0.5.

- Answer (derive value of P as a symbolic function of S):

  . . . $\Rightarrow$ P = S/$\log_2$(S)

- Check answer by observing that S/P = $\log_2$(S) $\Rightarrow$

  Speedup(S,P) = S/($\log_2$(S) + $\log_2$(S)) = S/(2*$\log_2$(S)) = P/2

Outline of Today's Lecture

- <u>**Data Races and Determinism**</u>
- **Memory Consistency Models**

- **Acknowledgments**
  - —COMP 322 Module 1 handout, Chapter 4
    - – https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

Parallel Programming Challenges

- **Correctness**
  - —New classes of bugs can arise in parallel programming, relative to sequential programming
    - – Data races, deadlock, nondeterminism
- **Performance**
  - —Performance of parallel program depends on underlying parallel system
    - – Language compiler and runtime system
    - – Processor structure and memory hierarchy
    - – Degree of parallelism in program vs. hardware
- **Portability**
  - —A buggy program that runs correctly on one system may not run correctly on another (or even when re-executed on the same system)
  - —A parallel program that performs well on one system may perform poorly on another

Bank Account Example

```
static int familyAccountBalance = 1000; // shared location
. . .
// parent deposits $100 to family account
async {
  parentAccountBalance = parentAccountBalance - 100;
  familyAccountBalance = familyAccountBalance + 100;
}
. . .
// student withdraws $100 from family account
async {
  familyAccountBalance = familyAccountBalance - 100;
  studentAccountBalance = studentAccountBalance + 100;
}
```

Question: What possible values can familyAccountBalance have after both async tasks have completed? Answer: 900, 1000, or 1100 !

Formal Definition of Data Races

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1 i.e., there is no path of dependence edges from S1 to S2 or from S2 to S1 in CG, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write. (L must be a shared location i.e., a static field, instance field, or array element.)

Data races are challenging because of

- Nondeterminism: different executions of the parallel program with the same input may result in different outputs.
- Debugging and Testing: it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program debugging and testing.

Examples of Data Races (Listing 5, Module 1 --- incorrect parallel versions)

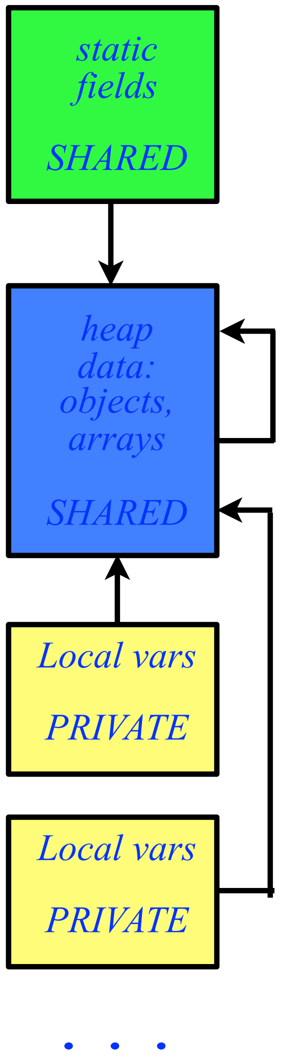
```
// Example 1: Incorrect parallel version
for (int i = 0; i < A.length; i++) async A[i] = B[i] + C[i];
System.out.println(A[0]);

// Example 2: Incorrect parallel version
p = first;
while ( p != null ) {
  async { p.x = p.y + p.z; }
  p = p.next;
}
System.out.println(first.x);
```

Shared and Private data in Java's Storage Model (Recap)

Java's storage model contains three memory regions:

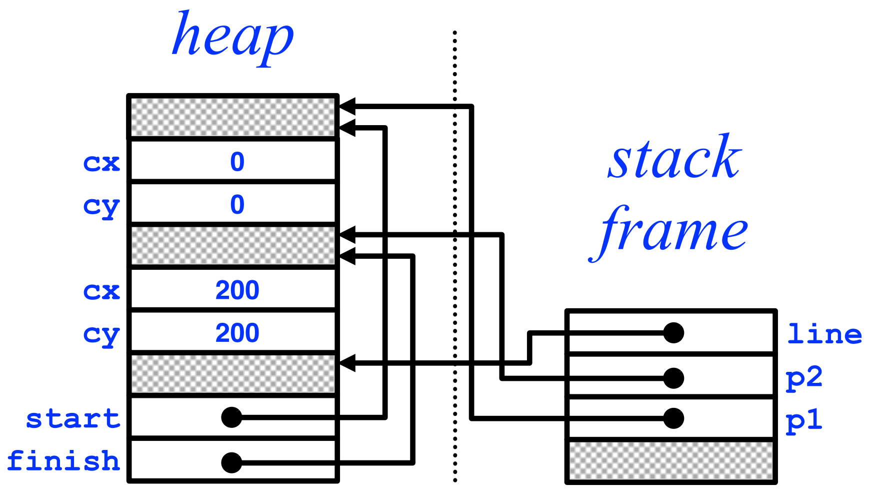
1. **Static Data**: region of memory reserved for variables that are not allocated or destroyed during a class' lifetime, such as static fields.
   - Static fields can be shared among threads/tasks
2. **Heap Data**: region of memory for dynamically allocated objects and arrays (created by "new").
   - Heap data can be shared among threads/tasks
3. **Stack Data**: Each time you call a method, Java allocates a new block of memory called a stack frame to hold its local variables
   - Local variables are private to a given thread/task

All references (pointers) must point to heap data --- no references can point to static or stack data



Example of Stack-to-Heap and Heap-to-Heap Pointers

```
public void run() {
   Point p1 = new Point(0, 0);
   Point p2 = new Point(200, 200);
   Line line = new Line(p1, p2); // Heap-stack diagram is for this stmt
}

public class Line {
   public Line(Point p1,
               Point p2) {
      start = p1;
      finish = p2;
   }
   . . .
   private Point start;
   private Point finish;
}

public class Point {
   public Point(int x, int y) {
      cx = x;
      cy = y;
   }
     . . .
   private int cx;
   private int cy;
}
```



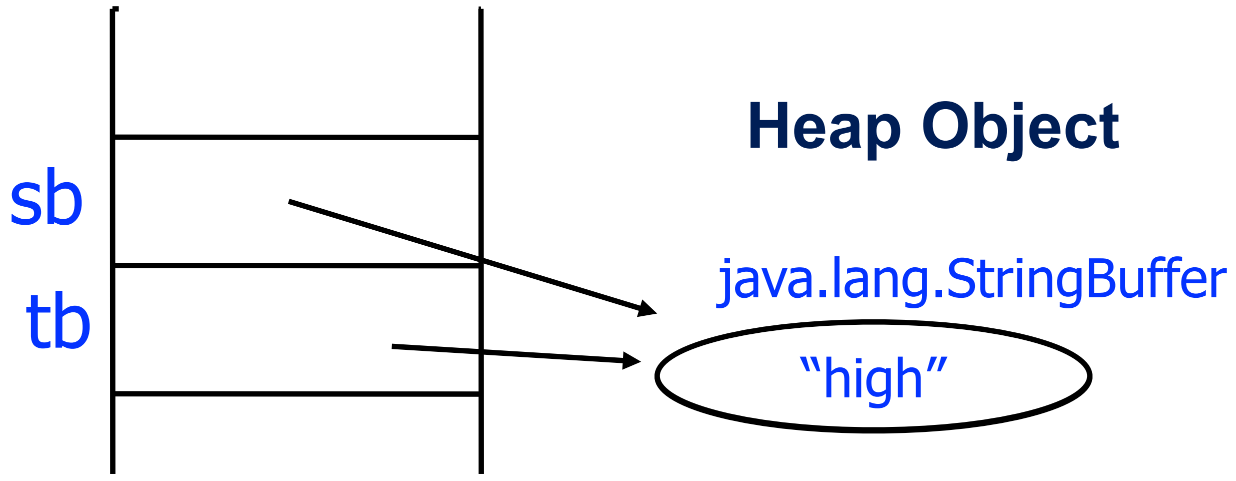
Mutability

- **If an object is modified, all references to the object see the new value**

```
StringBuffer sb = new ("hi");
StringBuffer tb = sb;
tb.append ("gh");
```

Stack Frame

Heap Object



Parameter Passing in Java

- **Call-by-value: All parameters in Java are passed by value. The method receives a copy of the parameter, not the caller's local variable.**
- **Parameters can only contain primitives and references. Copying a reference (pointer) does not make a copy of the object pointed to.**
- **Caller and callee methods can communicate in the following ways**
    - **—Parameters: callee receives a parameter from the caller in the form of a local variable (stack data)**
    - **—Return values: callee can return a single value as a local variable for the caller (stack data)**
    - **—Caller and callee can both read/write the same static fields (static data)**
    - **—Caller and callee can both read/write the same objects and arrays (heap data)**

These data sharing rules apply to tasks as well

Five Observations related to Data Races

1. Immutability property: there cannot be a data race on shared immutable data.
   - A location, L, is immutable if it is only written during initialization, and can only be read after initialization. In this case, no read can potentially execute in parallel with the write.

- Parallel programming tip: use immutable objects and arrays to avoid data races
  - Will require making copies of objects and arrays for updates
  - Copying overhead may be prohibitive in some cases, but acceptable in others

- Example with java.lang.String

```
1. finish {
2.   String s1 = "XYZ";
3.   async { String s2 = s1.toLowerCase(); ... }
4.   System.out.println(s1);
5. }
```

Observations

2. <u>Single-task ownership property</u>: there cannot be a data race on a location that is only read or written by a single task.
   - Define: step S in computation graph CG "owns" location L if S performs a read or write access on L. If step S belongs to Task T, we can also say that Task T owns L when executing S.
   - Consider a location L that is only owned by steps that belong to the same task, T. Since all steps in Task T must be connected by *continue* edges in CG, all reads and writes to L must be ordered by the dependences in CG. Therefore, no data race is possible on location L.

- Parallel programming tip: if an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task.
  - Will require making copies when sharing the object or array with other tasks.

Example of Single-task ownership with Copying

- If an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task.
  - Entails making copies when sharing the object with other tasks.
  - As with Immutability, copying overhead may be prohibitive in some cases, but acceptable in others.
- Example

```
finish { // Task T1 owns A
  int[] A = new int[n]; // ... initialize array A ...
  // create a copy of array A in B
  int[] B = new int[A.length]; System.arraycopy(A,0,B,0,A.length);
  async { // Task T2 owns B
    int sum = computeSum(B,0,B.length-1);// Modifies B as in ArraySum1
    System.out.println("sum = " + sum);
  }
  // ... update Array A ...
  System.out.println(Arrays.toString(A)); //printed by task T1
}
```

Observations (contd)

3. <u>Ownership-transfer property:</u> there cannot be a data race on a location if all steps that read or write it are totally ordered in CG (i.e., if the steps belong to a single directed path)
    - Think of the ownership of L being ``transferred" from one step to another, even across task boundaries, as execution follows the path of dependence edges in the total order.

- Parallel programming tip:
    - If an object or array needs to be written multiple times after initialization and also accessed by multiple tasks, then try and ensure that all the steps that read or write a location L in the object/ array are totally ordered by dependences in CG.
        - Ownership transfer is even necessary to support single-task ownership. In the previous example, since Task T1 initializes array B as a copy of array A, T1 is the original owner of A. The ownership of B is then transferred from T1 to T2 when Task T2 is created.

Observations (contd)

4. <u>Local-variable ownership property:</u> there cannot be a data race on a local variable.
   - If L is a local variable, it can only be written by the task in which it is declared (L's owner). The *copy-in* semantics for local variables ensures that the value of the local variable is copied on async creation thus guaranteeing that there is no race condition between the read access in the descendant task and the write access in L's owner.

- **Parallel programming tip:**
   - You do not need to worry about data races on local variables, since they are not possible. However, local variables in Java are restricted to contain primitive data types (such as int) and references to objects and arrays. In the case of object/array references, be aware that there may be a data race on the underlying object even if there is no data race on the local variable that refers to (points to) the object.

Relating Data Races and Determinism

- A parallel program is said to be deterministic with respect to its inputs if it always computes the same answer when given the same inputs.
- Structural Determinism Property
  - —If a parallel program is written using the constructs in Module 1 and is guaranteed to be race-free, then it must be deterministic with respect to its inputs. The final computation graph is also guaranteed to be the same for all executions of the program with the same inputs.
- Constructs introduced in Module 1 (“Deterministic Shared-Memory Parallelism”) include async, finish, finish accumulators, futures, data-driven tasks (async await), forall, barriers, phasers, and phaser accumulators.
  - —The notable exceptions are critical sections, isolated statements, and actors, all of which will be covered in Module 2 (“Nondeterministic Shared-Memory Parallelism”)

Worksheet #5: Data Races and Determinism

Name 1: ____________________ Name 2: ____________________

Consider a modified String Search program that returns true if any occurrence is found, rather than the count of all occurrences:

```
static boolean found = false; // static field
. . .
finish for (int i = 0; i <= N - M; i++)
  async {
    for (j = 0; j < M; j++)
      if (text[i+j] != pattern[j]) break;
    if (j == M) found = true; // found at offset i
  } // finish-for-async
```

<u>Question:</u> Does this program have a data race? Is it deterministic? Why?

(Use the space below this slide for your answer)

Outline of Today's Lecture

- **Data Races and Determinism**
- **<u>Memory Consistency Models</u>**

- **Acknowledgments**
  - —COMP 322 Module 1 handout, Chapter 4
    - – https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

Data Races are usually Errors, but not always

- **Example of Data Race Error**

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

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

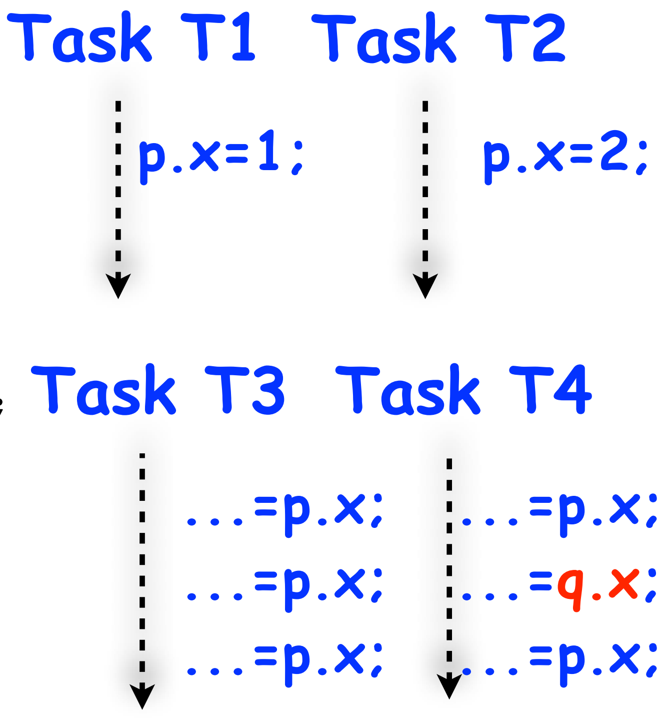
```
static int index = -1; // static field
. . .
finish for (int i = 0; i <= N - M; i++) async {
  for (j = 0; j < M; j++)
    if (text[i+j] != pattern[j]) break;
  if (j == M) index = i;               // found at offset i
}
```

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

Semantics of Data Races

Example HJ program:

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x)
}
async { // Task T4
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p.x);
}
```



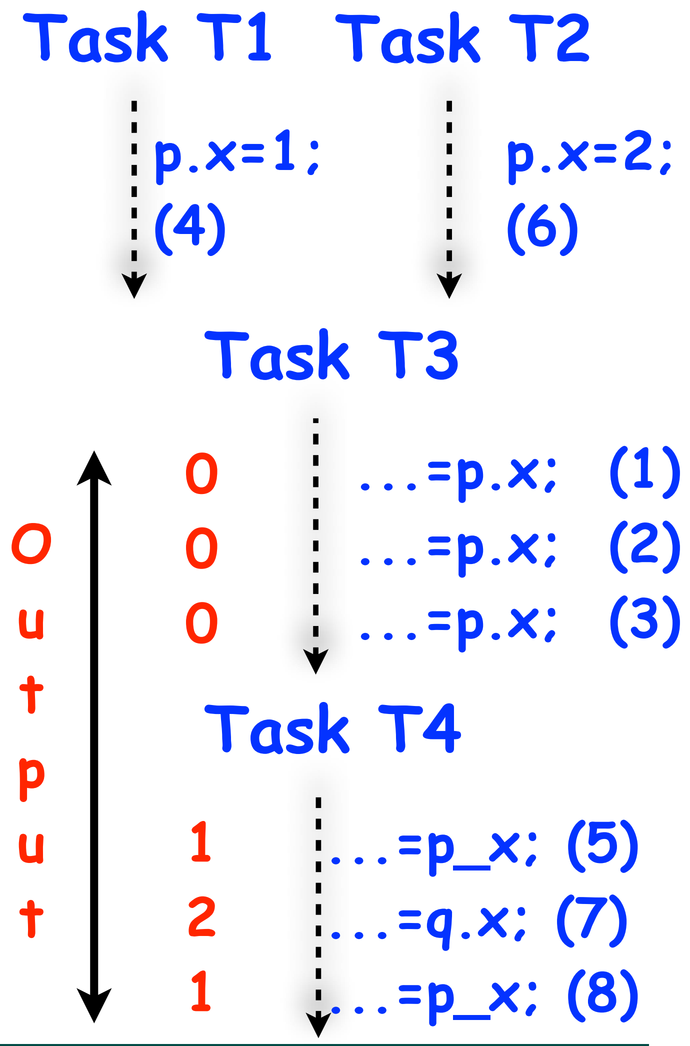
Can the following values be printed by tasks T3 & T4?

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

Consider a “reasonable” code transformation performed by a programmer: is it legal according to SC?

Example HJ program:

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x)
}
async { // Task T4
  // Assume programmer doesn’t know that p=q
  int p_x = p.x;
  System.out.println("First read = " + p_x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p_x);
}
```



Memory Consistency Models

- A memory consistency model, or memory model, is the part of a programming language specification that defines what write values a read may see in the presence of data races.
- We will briefly introduce three memory models, and discuss them in more detail later in the course
  - —Sequential Consistency (SC)
    - – Suitable for specifying semantics at the hardware and OS levels
  - —Java Memory Model (JMM)
    - – Suitable for specifying semantics at worker thread level
  - —Habanero Java Memory Model (HJMM)
    - – Suitable for specifying semantics at application task level