COMP 322: Fundamentals of Parallel Programming

Lecture 7: Data Races, Functional & Structural Determinism

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

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

Worksheet #6 solution: Why must Future References be declared as final?

Consider the pseudocode on the right with futures declared as non-final static fields. Is there a possible execution in which a deadlock situation may occur between tasks T1 and T2 with this code (with each task waiting on the other due to get() operations)? Explain why or why not.

Yes, a deadlock can occur when future f1 does f2.get() and future f2 does f1.get().

WARNING: such "spin" loops are an example of bad parallel programming practice in application code. Their semantics depends on the "memory model". In the Java memory model, there's no guarantee that the above spin loops will ever terminate.

```
static future f1=null;
static future f2=null;

void main(String[] args) {
  f1 = async {return a1();};
  f2 = async {return a2();};
}

int a1() { // Task T1
  while (f2 == null); // spin loop
  return f2.get(); //T1 waits for T2
}

int a2() { // Task T2
  while (f1 == null); // spin loop
  return f1.get(); //T2 waits for T1
}
```

deadlock

Why should Future References be declared as final?

Now consider a modified version of the above code in which futures are declared as final local variables (which is permitted in HJ). Can you add get() operations to methods a1() and a2() to create a deadlock between tasks T1 and T2 with this code? Explain why or why not.

No, the final declarations make it impossible for future f1's task (T1) to receive a reference to f2.

Will your answer be different if f1 and f2 are final fields in objects or final static fields?

No.

```
final future f1 =
    async {return a1();};
 final future f2 =
    async {return a2(f1);};
}

int a1() {
// Task T1 cannot receive a
// reference to f2
 . . .
}

int a2(futuref1) {
// Task T2 can refer
// to f1 but that won't cause
// a deadlock.
... f1.get() ...
}
```

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

What happens if we forget a finish?

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
async { // Task T0 computes sum of lower half of array
  for(int i=0; i < X.length/2; i++)
    sum1 += X[i];
}
async { // Task T1 computes sum of upper half of array
  for(int i=X.length/2; i < X.length; i++)
    sum2 += X[i];
}
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
```

Data race between accesses of sum1 in async and in main program

Formal Definition of Data Races

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.)

- A program is *data-race-free* it cannot exhibit a data race for any input
- Above definition includes all “potential” data races i.e., it’s considered a data race even if S1 and S2 execute on the same processor

Four 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
  - NOTE: future values are also immutable

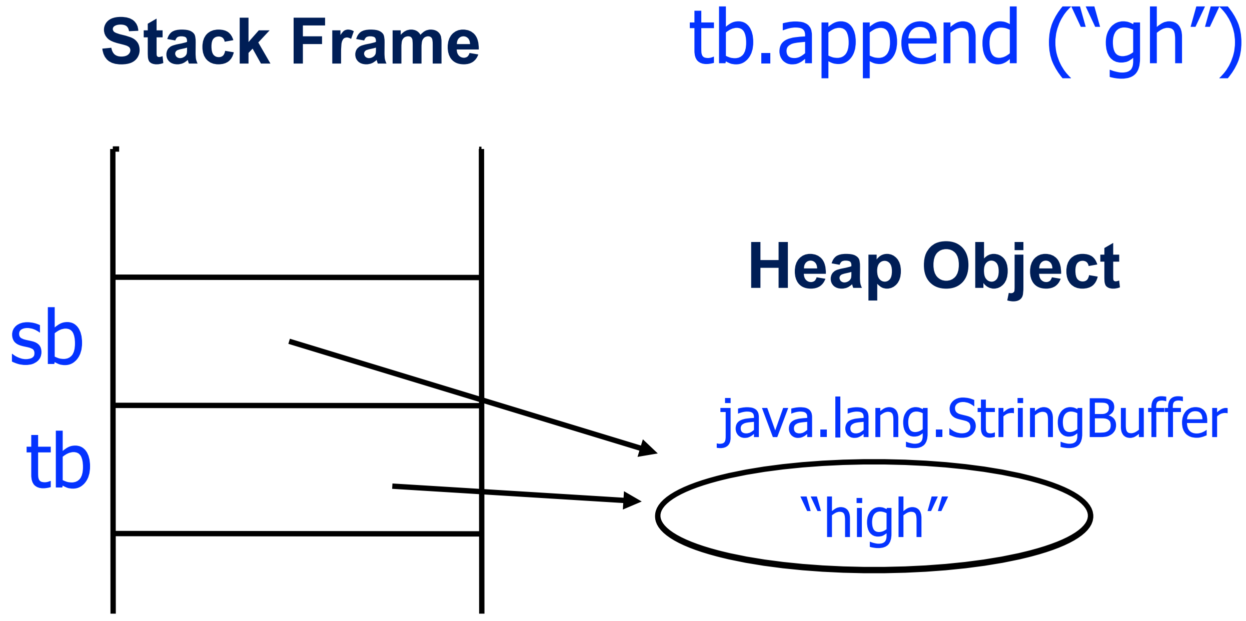
- Example with java.lang.String

```
finish {
  String s1 = "XYZ";
  async { String s2 = s1.toLowerCase(); ... }
  System.out.println(s1);
}
```

Example of a Mutable Object

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

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



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. Local-variable ownership property: 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 "*implicitly final*" semantics for accessing outer local variables ensures that there is no race condition between the read access in the child task and the write access in L's owner (parent task).

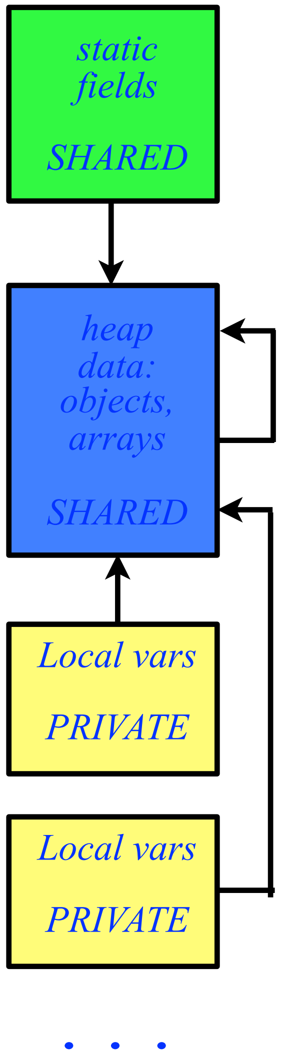
- Parallel programming tip:
  - You do not need to worry about data races on local variables, since they are not possible. However, local variables in Java are restricted to contain primitive data types (such as int) and references to objects and arrays. In the case of object/array references, be aware 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.

Recap of Java's Storage Model

Java's storage model contains three memory regions:

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

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



Functional vs. Structural Determinism

- A parallel program is said to be *functionally deterministic* if it always computes the same answer when given the same input
- A parallel program is said to be *structurally deterministic* if it always produces the same computation graph when given the same input
- Race-Free Determinism
  - —If a parallel program is written using the constructs learned so far (finish, async, futures) and is known to be race-free, *then it must be both functionally deterministic and structurally deterministic*

V1: Functional + Structural Determinism (No data race)

```
// Count all occurrences
a = new ACCUM
finish(a) 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) a.put(1);          // found
 }
print a.get();
```

V2: Functional + Structural Determinism (Benign data race)

```
// Existence of an occurrence
found = false
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;
 }
 print found
```

V3: Functional Nondeterminism + Structural Determinism

```
// Index of an occurrence
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 i
}
```

V4: Functionally Deterministic + Structurally Nondeterministic

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

V5: Functionally Nondeterministic + Structurally Nondeterministic

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

A Classification of Parallel Programs

Data Race Free?	Functionally Deterministic?	Structurally Deterministic?	Example: String Search variation
Yes	Yes	Yes	Count of all occurrences
No	Yes	Yes	Existence of an occurrence
No	No	Yes	Index of any occurrence
No	Yes	No	“Eureka” extension for existence of an occurrence: do not create more async tasks after occurrence is found
No	No	No	“Eureka” extension for index of an occurrence: do not create more async tasks after occurrence is found

Data-Race-Free Determinism Property implies that it is not possible to write an HJ program with Yes in column 1, and No in column 2 or column 3 (when only using Module 1 constructs)