Acknowledgments for Today’s Lecture

- Lecture 10 handout
Introduction

• For the programming constructs async, finish, future, get, forall, the following situation was defined to be a data race error
  —when two accesses on the same shared location can potentially execute in parallel such that at least one access is a write.

• However, there are many cases in practice when two tasks may legitimately need to perform conflicting accesses to shared locations.
Example of two tasks performing conflicting accesses

1. class DoublyLinkedList {
2.   DoublyLinkedList prev, next;
3.   . . .
4.   void delete() {
5.       isolated { // start of mutual exclusion region (critical section)
6.           if (this.prev != null) this.prev.next = this.next;
7.           if (this.next != null) this.next.prev = this.prev
8.       } // end of mutual exclusion region (critical section)
9.   . . .
10.  }
11.  . . .
12.}
13. . . .
14. static void deleteTwoNodes(DoublyLinkedList L) {
15.     finish {
16.         async L.delete();
17.         async L.next.delete();
18.     }
19.}
How to enforce mutual exclusion?

- The predominant approach to ensure mutual exclusion proposed many years ago is to enclose the code region in a critical section.

  — “In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.”
HJ isolated statement

isolated <body>

• Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  — Two instances of isolated statements, ⟨stmt1⟩ and ⟨stmt2⟩, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.
  ➔ Weak isolation guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances

• Isolated statements may be nested (redundant)

• Isolated statements must not contain any other parallel statement: async, finish, get, forall

• In case of exception, all updates performed by <body> before throwing the exception will be observable after exiting <body>
How small or big should an isolated statement be?

- Too small $\Rightarrow$ may lose invariants desired from mutual exclusion
- Too big $\Rightarrow$ limits parallelism

- Observation: no combination of finish, async, get, forall and isolated constructs can create a deadlock cycle among tasks.
Serialized Computation Graph for Isolated Statements

- Model each instance of an isolated statement as a distinct step (node) in the CG.

- Need to reason about the order in which interfering isolated statements are executed
  - complicated because the order may vary from execution to execution

- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated statements.
  - SCG consists of a CG with additional serialization edges.
  - Each time an isolated step, $S'$, is executed, we add a serialization edge from $S$ to $S'$ for each isolated step, $S$, that has already executed such that $S$ and $S'$ have interfering accesses.
  - An SCG represents a set of executions in which all interfering isolated statements execute in the same order.
Example of Serialized Computation Graph with Serialization Edges

\[ v_{10}: \text{isolated} \left\{ x++; y = 10; \right\} \]
\[ v_{11}: \text{isolated} \left\{ x++; y = 11; \right\} \]
\[ v_{16}: \text{isolated} \left\{ x++; y = 16; \right\} \]
1. class V {
2.     V[] neighbors; // adjacency list for input graph
3.     V parent;       // output value of parent in spanning tree
4.     boolean tryLabeling(V n) {
5.         isolated if (parent == null) parent=n;
6.         return parent == n;
7.     } // tryLabeling
8.     void compute() {
9.         for (int i=0; i<neighbors.length; i++) {
10.            V child = neighbors[i];
11.            if (child.tryLabeling(this))
12.                async child.compute(); //escaping async
13.         }
14.     } // compute
15.} // class V
16...
17. root.parent = root; // Use self-cycle to identify root
18. finish root.compute();
19...
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 $S_1$ and $S_2$ in computation graph $CG$ such that:

1. $S_1$ does not depend on $S_2$ and $S_2$ does not depend on $S_1$ i.e., there is no path of dependence edges from $S_1$ to $S_2$ or from $S_2$ to $S_1$ in $CG$, and
2. Both $S_1$ and $S_2$ read or write $L$, and at least one of the accesses is a write.

Apply above definition to an SCG