COMP 322: Parallel and Concurrent Programming

Lecture 20: Critical Sections, Isolated Construct

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Formal Definition of Data Races (Recap)

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:

- 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
- Both S1 and S2 read or write L, and at least one of the accesses is a write.

However, there are many cases in practice when two tasks may legitimately need to perform conflicting accesses to shared locations without incurring data races

- How should conflicting accesses be handled in general, when outcome may be nondeterministic?

Focus of the “Concurrency” part of the course (nondeterministic parallelism)

Conflicting accesses --- need for “mutual exclusion”

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   { // start of desired mutual exclusion region
     this.prev.next = this.next;
     this.next.prev = this.prev;
   } // end of desired mutual exclusion region
   . . . // remaining code in delete() that does not need mutual exclusion
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final DoublyLinkedListNode L) {
   finish(() -> {
      DoublyLinkedListNode second = L.next;
      DoublyLinkedListNode third = second.next;
      async(() -> { second.delete(); });
      async(() -> { third.delete(); }); // conflicts with previous async
 });
}
```

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."

  http://en.wikipedia.org/wiki/Critical_section
- Also known as the "Monitor Concurrency Pattern"

“Global” isolated construct

- `isolated (() -> <body> );`
- Isolated construct identifies a critical section
- Two tasks executing isolated constructs are guaranteed to perform them in mutual exclusion
  - Isolation guarantee only applies to (isolated, isolated) pairs of constructs, not to (isolated, non-isolated) pairs of constructs
- Isolated constructs may be nested
  - An inner isolated construct is redundant
- Blocking parallel constructs are forbidden inside isolated constructs
  - Isolated constructs must not contain any parallel constructs that perform a blocking operation
    - finish, future get, next
  - Non-blocking task (async task, future task, data-drive task) creation is permitted, but isolation guarantee only applies to the creation of the task, not to its execution
- Isolated constructs can never cause a deadlock
  - Other techniques for enforcing mutual exclusion (e.g., locks) could lead to a deadlock, if used incorrectly
- “Global isolated” construct is semantically equivalent to a “***global lock***”

Use of isolated to fix previous example with conflicting accesses

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(() -> { // start of desired mutual exclusion region
     this.prev.next = this.next;
     this.next.prev = this.prev;
   }); // end of desired mutual exclusion region
   . . . // other code in delete() that does not need mutual exclusion
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); }); // conflicts with previous async
 });
}
```

Global Isolated

Compute the CPL for this program with a global isolated construct.

```
finish(() -> {
    for (int i = 0; i < 5; i++) {
      async(() -> {
        doWork(2);
        isolated(() -> { doWork(1); });
        doWork(2);
      }); // async
    } // for
 }); // finish
```

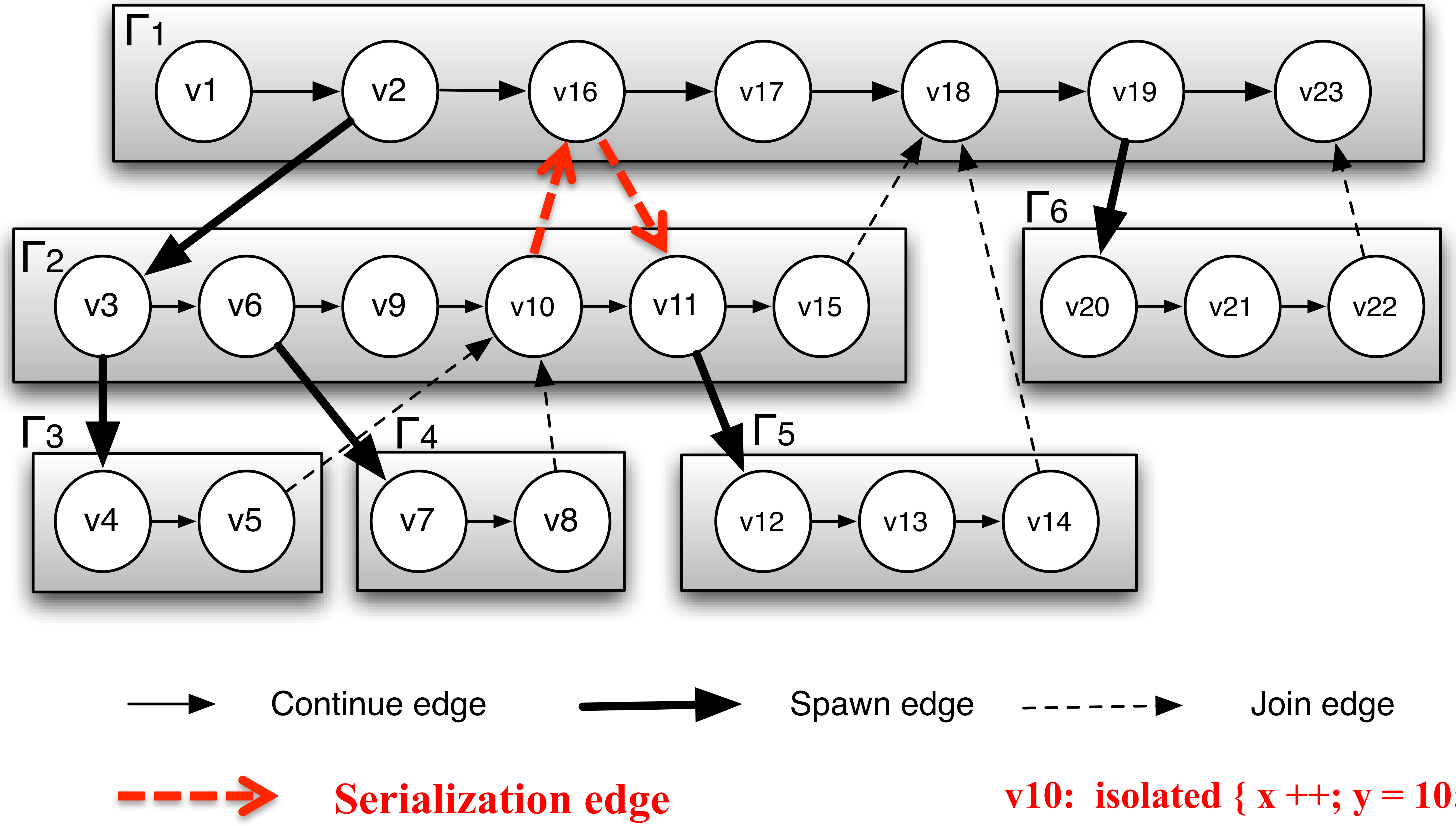
Object-based isolated construct

- `isolated (Object participant1, () → <body> );`
- `isolated (Object participant1, Object participant2, () → <body> );`
- `isolated (Object participant1, Object participant2, Object participant3, () → <body> );`
- `isolated (Object[] participants, () → <body> );`
- Allows for finer-grained control of critical sections
- Two isolated constructs that have an empty intersection of participant objects do not interfere
- When nesting (still redundant), the inner isolated participant object set has to be a subset of the outer one
- Deadlock guarantee still applies
  - Implementation makes sure the objects are acquired in a global order
  - Object-based isolated construct is ***not*** semantically the same as per-object locking
- Serialization edges are only added between isolated steps with at least one common object (non-empty intersection of object lists)
- Standard isolated is equivalent to "isolated(*)" by default i.e., isolation across all objects
- Related concept: Java Synchronized blocks and methods

Serialized Computation Graph for Isolated Constructs

- Model each instance of an isolated construct as a distinct step (node) in the CG.
- Need to reason about the *order* in which interfering isolated constructs are executed
  - Complicated because the order of isolated constructs may vary from execution to execution
- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated constructs.
  - 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 prior "interfering" isolated step, S
    - Two "global isolated" constructs always interfere with each other
    - Interference of "object-based isolated" constructs depends on intersection of object sets
    - Serialization edge is not needed if S and S' are already ordered in CG
  - An SCG represents a set of schedules in which all interfering isolated constructs execute in the same order.

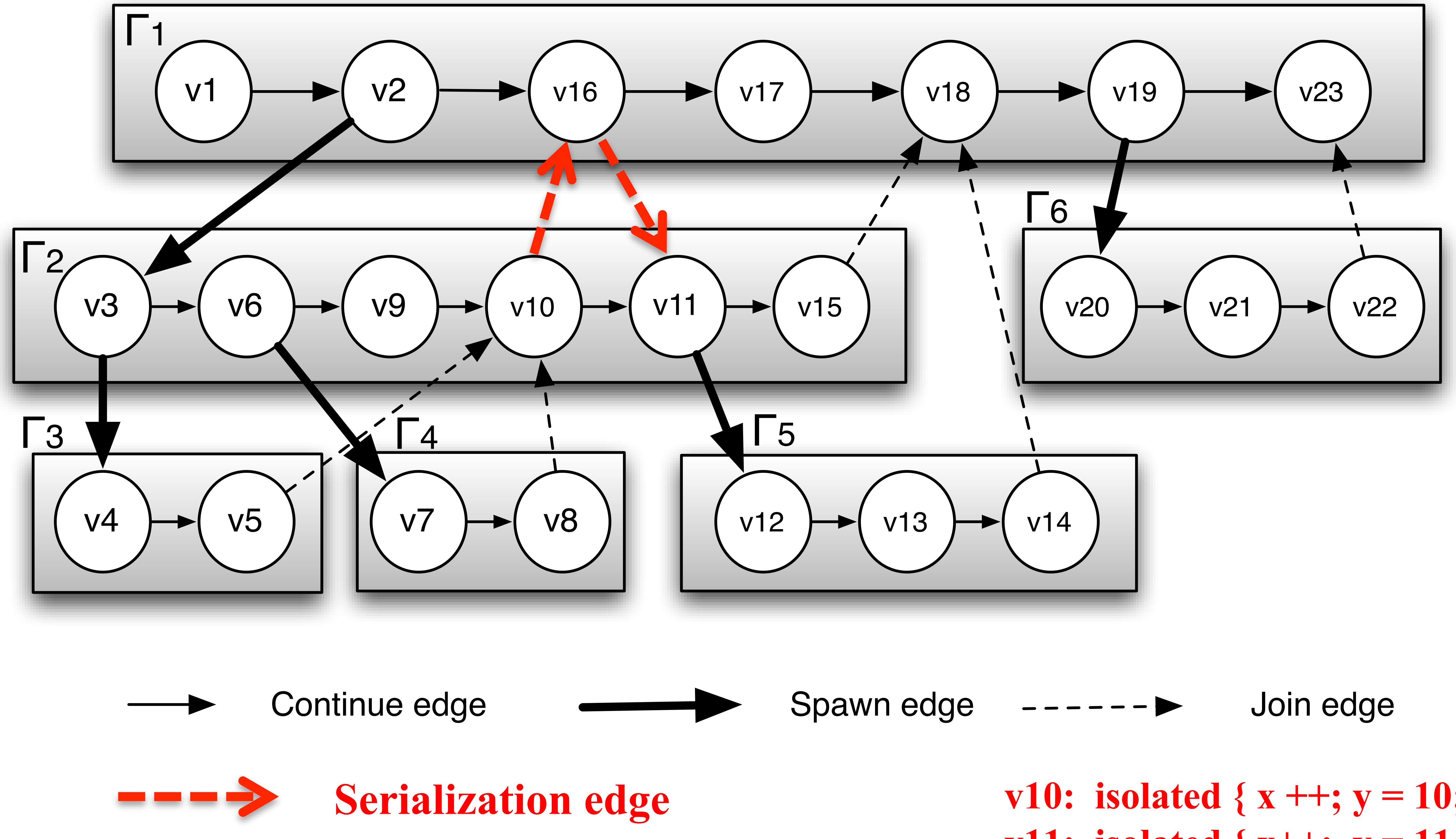
Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order



v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order

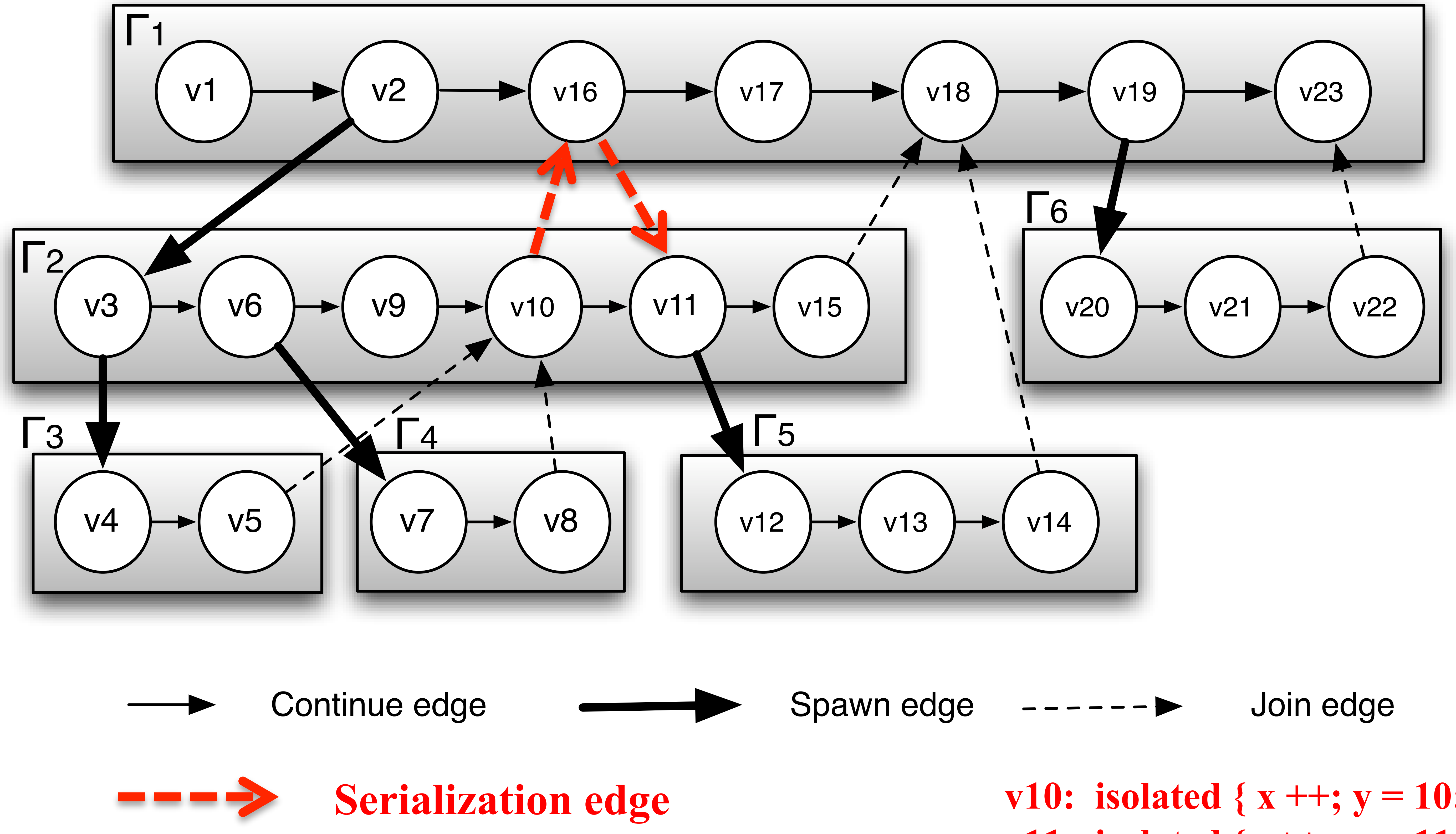
Data race definition can be applied to Serialized Computation Graphs (SCGs) just like regular CGs



v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

Example of Serialized Computation Graph with Serialization Edges for v10-v16-v11 order

Data race definition can be applied to Serialized Computation Graphs (SCGs) just like regular CGs



v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

Have to consider all possible orderings of interfering isolated constructs to establish data race freedom!

DoublyLinkedListNode with Object-Based Isolation

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(?, ?,…, () -> { // object-based isolation
     this.prev.next = this.next;
     this.next.prev = this.prev;
   });
   . . .
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); });
 });
}
```

DoublyLinkedListNode with Object-Based Isolation

```
class DoublyLinkedListNode {
 DoublyLinkedListNode prev, next;
 . . .
 void delete() {
   isolated(this.prev, this, this.next, () -> { // object-based isolation
     this.prev.next = this.next;
     this.next.prev = this.prev;
   });
   . . .
 }
} // DoublyLinkedListNode
. . .
static void deleteTwoNodes(final DoublyLinkedListNode L) {
 finish(() -> {
   DoublyLinkedListNode second = L.next;
   DoublyLinkedListNode third = second.next;
   async(() -> { second.delete(); });
   async(() -> { third.delete(); });
 });
}
```

Pros and Cons of Object-Based Isolation

- Pros
  - Increases parallelism relative to critical section approach
  - Simpler approach than “locks” (which we will learn later)
  - Deadlock-freedom property is still guaranteed

- Cons
  - Programmer needs to worry about getting the participant objects right
  - Participant objects can only be specified at start of the isolated construct
  - Large participant object arrays can contribute to large overheads

Summary

- Concurrent access to shared data is sometimes unavoidable
- Global isolated construct guarantees deadlock-free and race-free access to shared data, but may be too restricting
- Object-based isolation still guarantees deadlock-free and race-free access to shared data, but requires more programmer involvement
- If you mix isolation with non-isolated access to shared data, you ***still*** have to reason about data races in your computation graph
- To prove race-freedom, you have to consider ***all*** legal orderings of isolated constructs