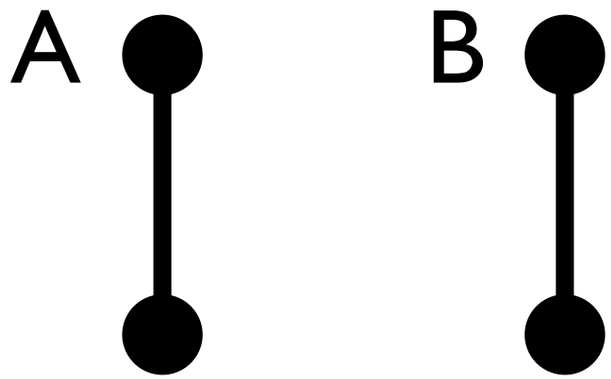
Reflective Parallel Programming

Nicholas D. Matsakis, Thomas R. Gross
ETH Zurich

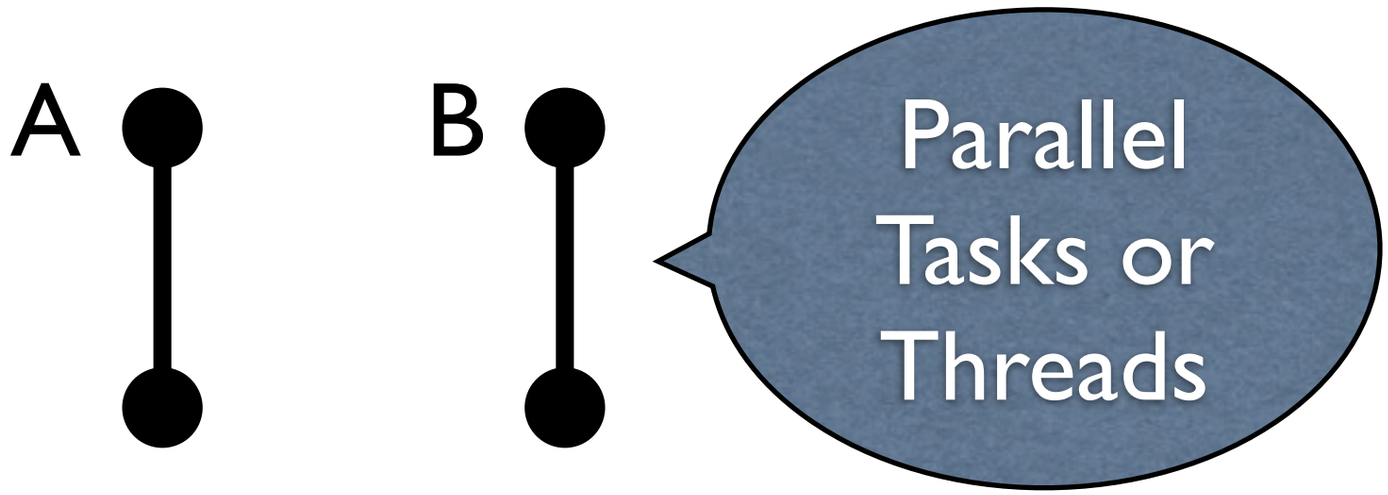
Reflective Parallelism

- *Reflection*: Ability for a program to reason about its own structure
- *Reflective Parallelism*: Ability for a program to reason about its own *schedule*.
- *Schedule*: the (partial) order in which parallel tasks execute.

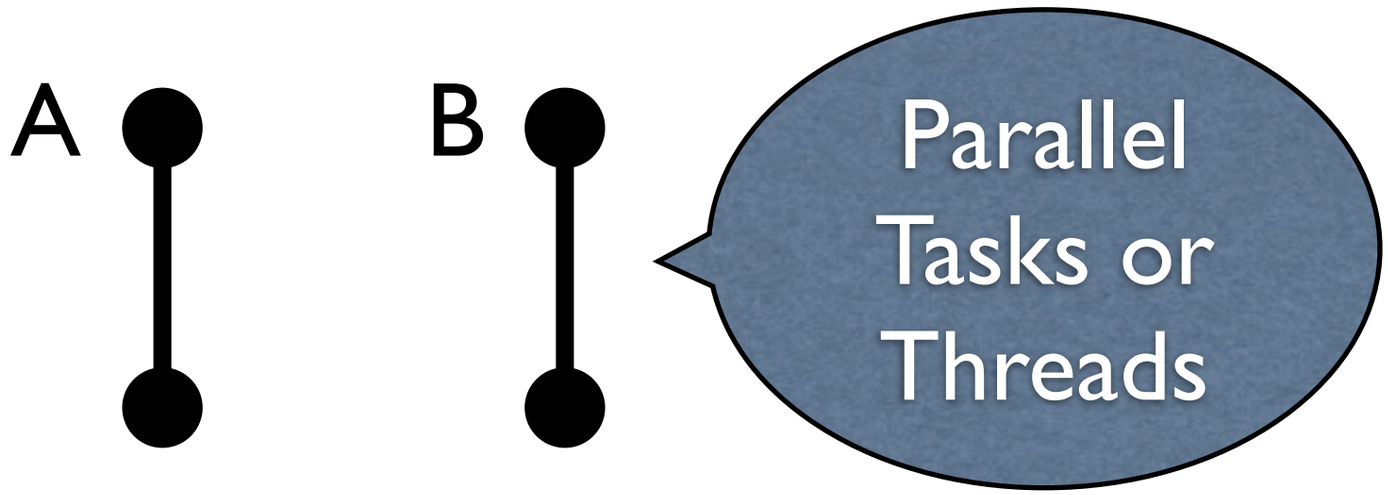
Reflection Example



Reflection Example

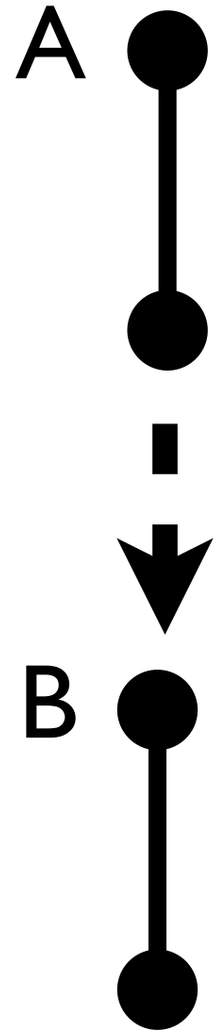


Reflection Example



Unordered tasks?

Reflection Example



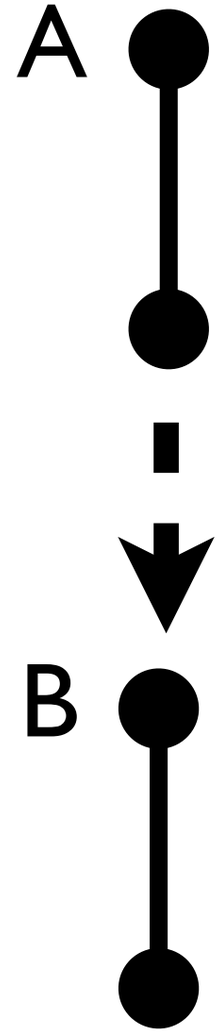
A must end before B starts?

Reflective Parallelism

- Reflective queries should return results that hold for all executions

- Reflection also allows interaction

  - Add scheduling constraints, etc.



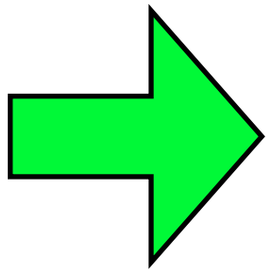
Static Evaluation

- When possible, should be able to analyze schedule statically.
- Only partial schedule known at compile time.

Applications

- Data-race detection
- Schedule visualization
- Testing frameworks
- ...and more

Outline



- What is reflective parallel programming?
- Why do we need a new model?
- Intervals model
- Example: Data-race detection

Outline

- What is reflective parallel programming?
- Why do we need a new model?
- Intervals model
- Example: Data-race detection

Traditional Threading

- Traditional APIs use operational primitives:
  - start, join a thread
  - wait for a signal, acquire a lock
- Program schedule not defined in advance
  - Can only query *after* execution!

Difficult to Analyze Statically

```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```

Difficult to Analyze Statically

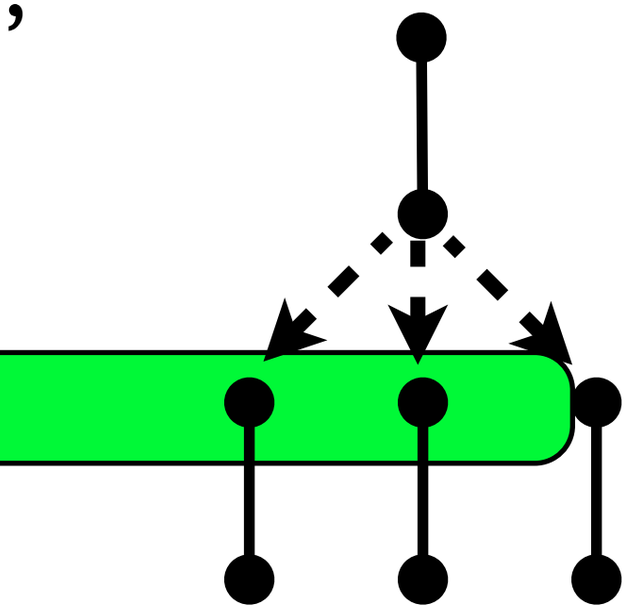
```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```

Difficult to Analyze Statically

```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```

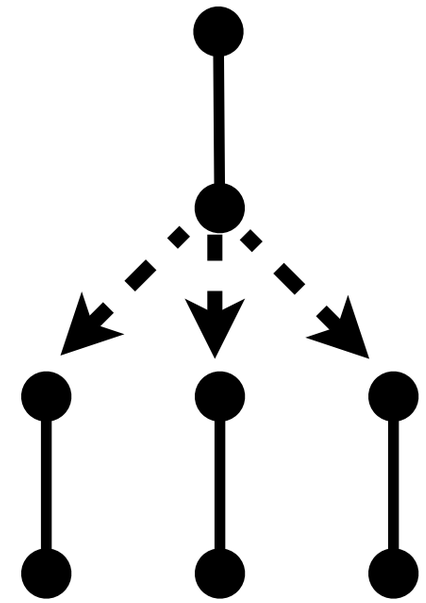
Difficult to Analyze Statically

```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```



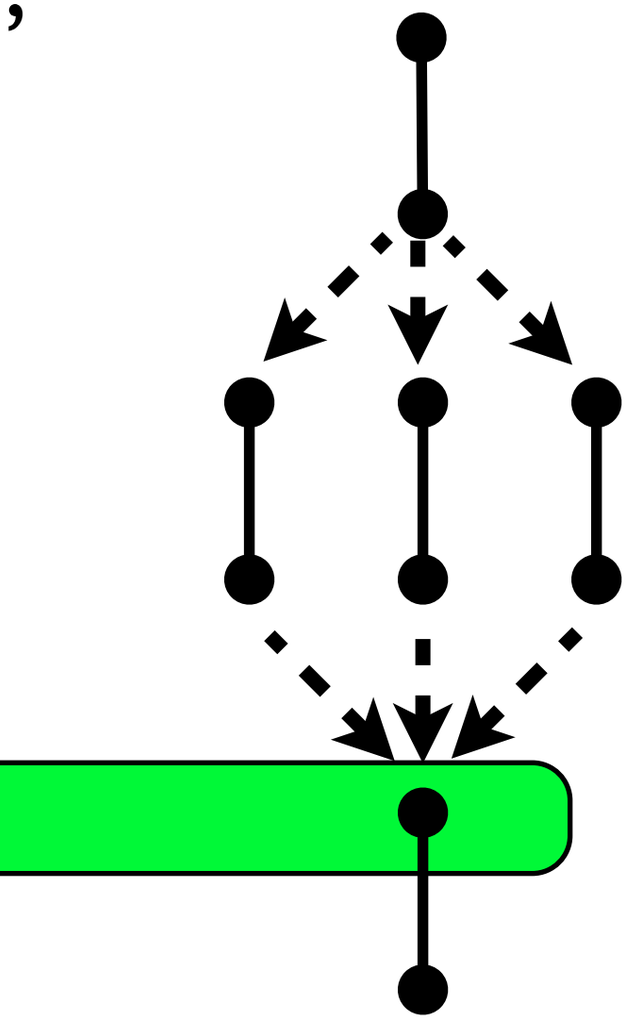
Difficult to Analyze Statically

```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```



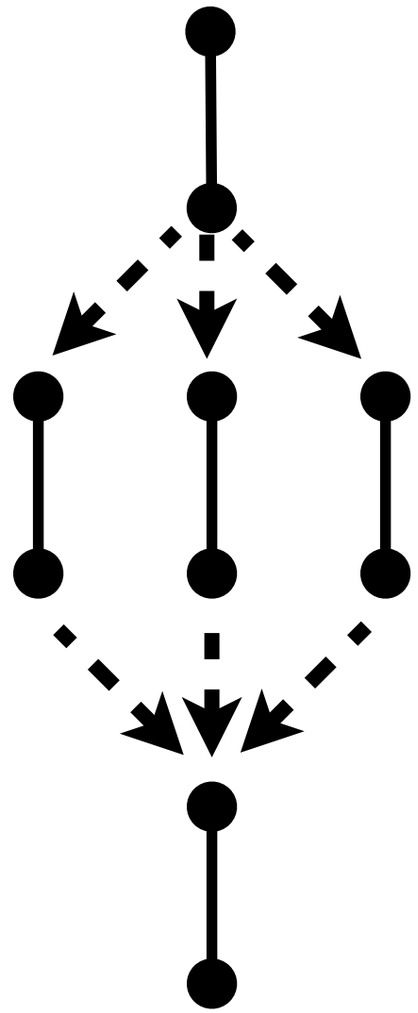
Difficult to Analyze Statically

```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```



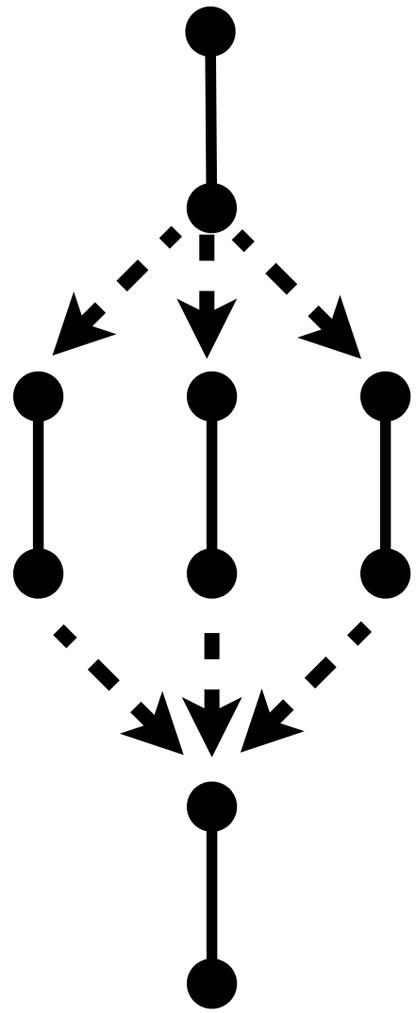
Difficult to Analyze Statically

```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```



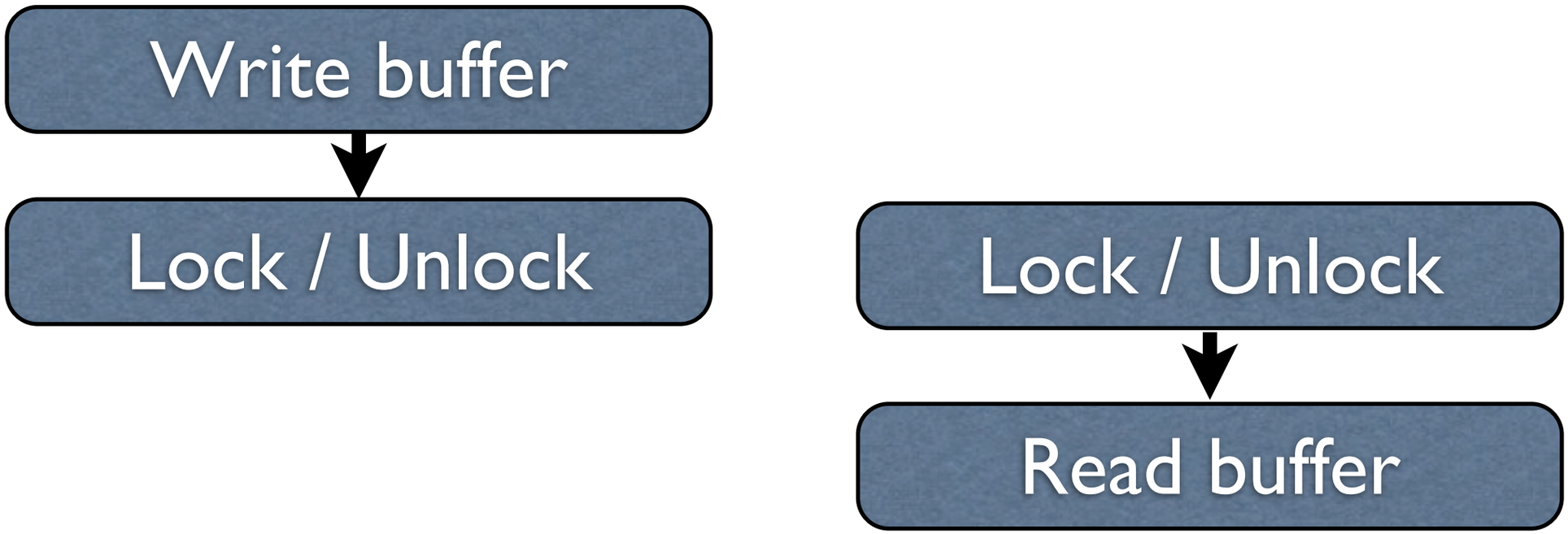
Difficult to Analyze Statically

```
Thread[] threads = new Thread[N];

for(int i = 0; i < N; i++) {
  threads[i] = new Thread(...);
  threads[i].start();
}

for(int i = 0; i < N; i++)
  threads[i].join();
```

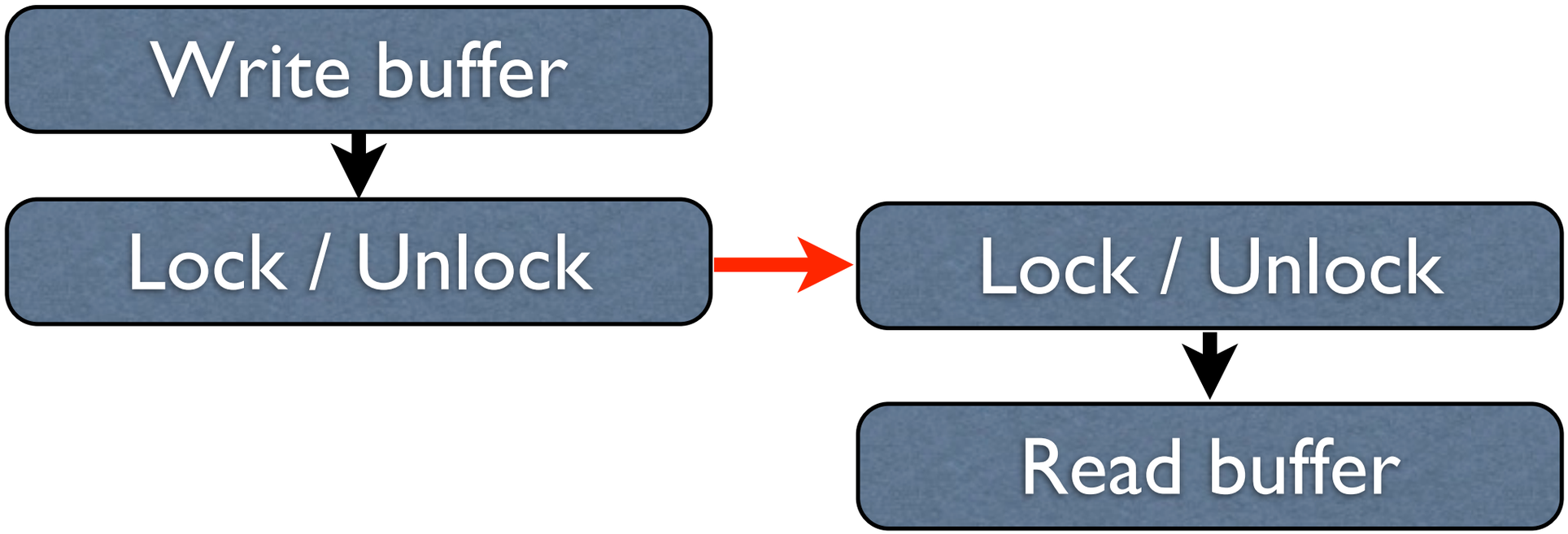


Have all threads been joined?

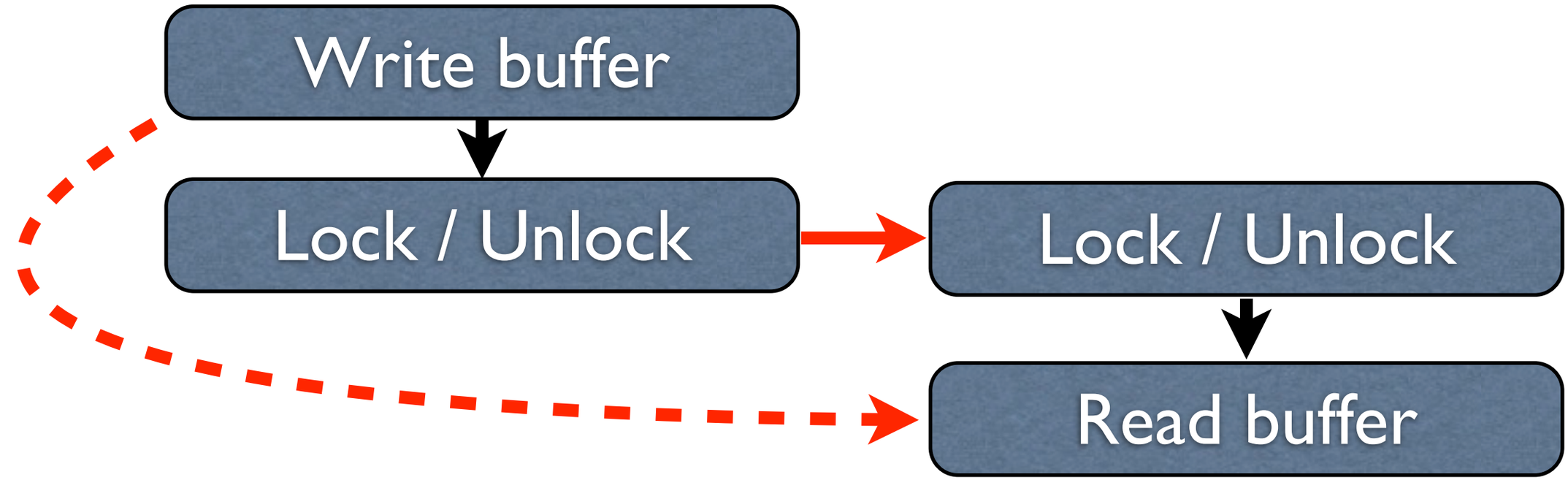
Reverse Engineering is Risky



Reverse Engineering is Risky

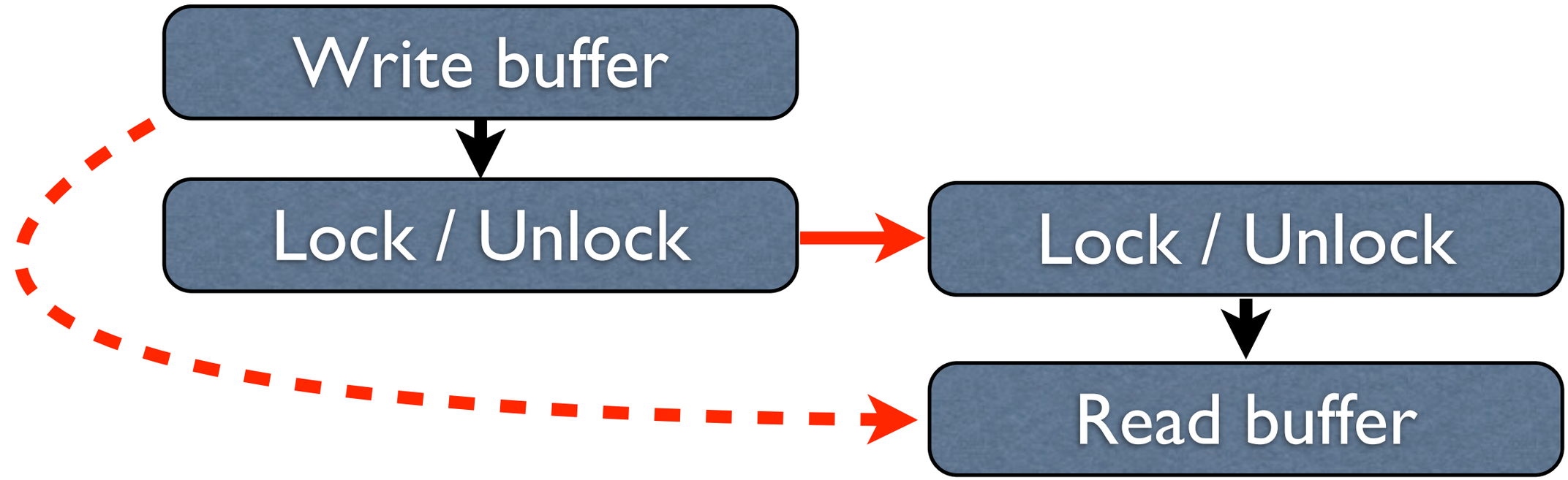


Reverse Engineering is Risky



Observed: Wr happened before Rd

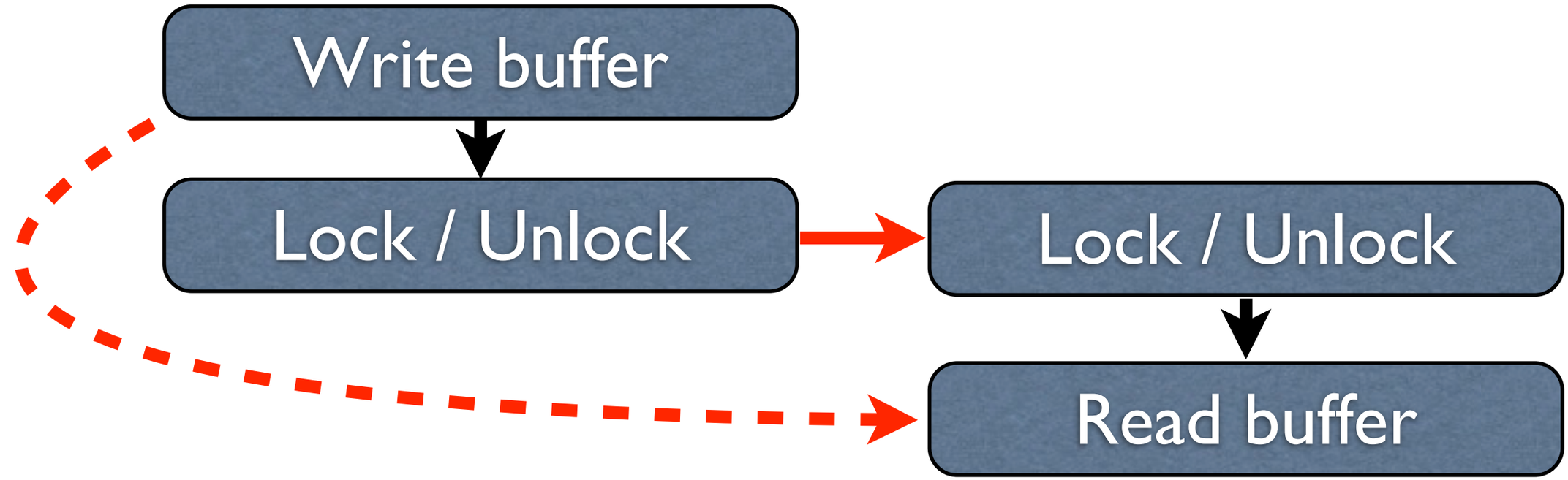
Reverse Engineering is Risky



Observed: Wr happened before Rd

Conclusion: Wr happens before Rd?

Reverse Engineering is Risky



Observed: Wr happened before Rd

Conclusion: Wr happens before Rd?

Past performance is no guarantee of future results.

Summary

- Traditional model unsuitable for reflection
- Cannot know schedule in advance
- Difficult to analyze statically
- Can draw false conclusions

Outline

- What is reflective parallel programming?
- Why do we need a new model?
- Intervals model
- Example: Data-race detection

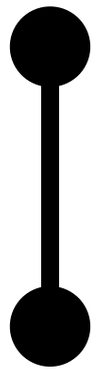
Outline

- What is reflective parallel programming?
- Why do we need a new model?
- Intervals model
- Example: Data-race detection

Intervals Approach

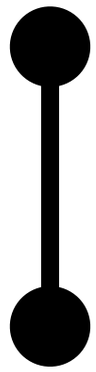
- Schedule is a first-class entity
- Users builds desired schedule through declarative methods
- Runtime executes simultaneously
- Schedule can be queried during execution

Schedule Model



Intervals
represent
asynchronous tasks
or group of tasks.

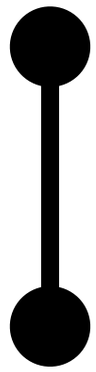
Schedule Model



Intervals
represent
asynchronous tasks
or group of tasks.

```
Interval a = interval {
   ...
};
```

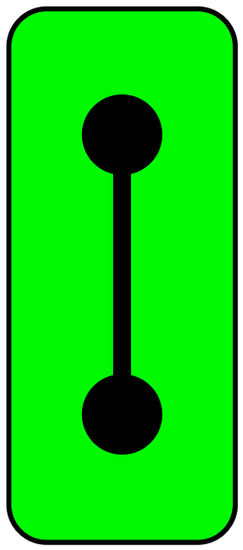
Schedule Model



Intervals
represent
asynchronous tasks
or group of tasks.

```
Interval a = interval {
    ...
};
```

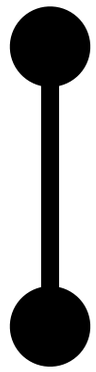
Schedule Model



Intervals
represent
asynchronous tasks
or group of tasks.

```
Interval a = interval {
    ...
};
```

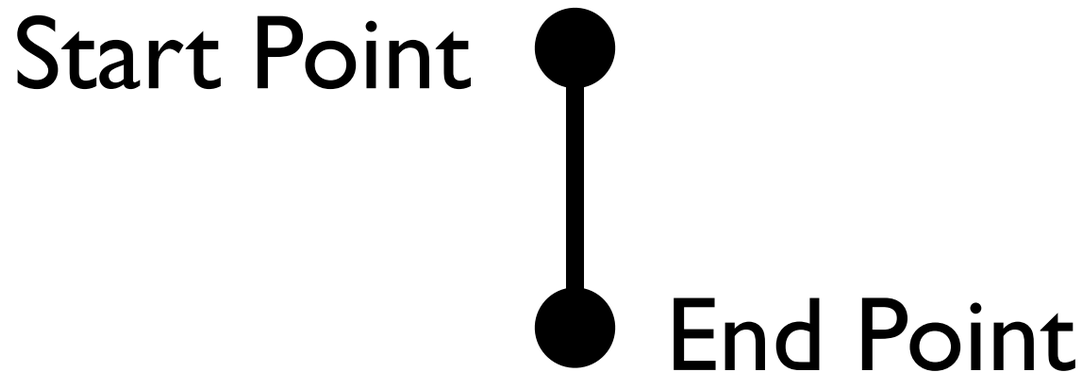
Schedule Model



Intervals
represent
asynchronous tasks
or group of tasks.

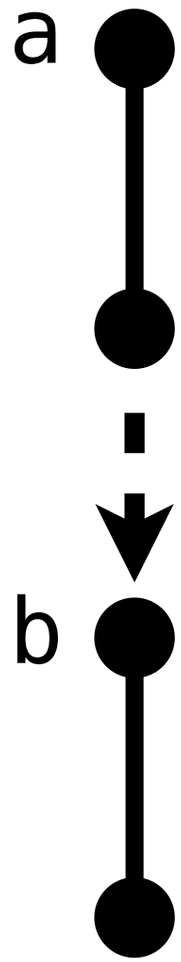
```
Interval a = interval {
   ...
};
```

Schedule Model



Points represent the moments in time when the interval begins or ends execution.

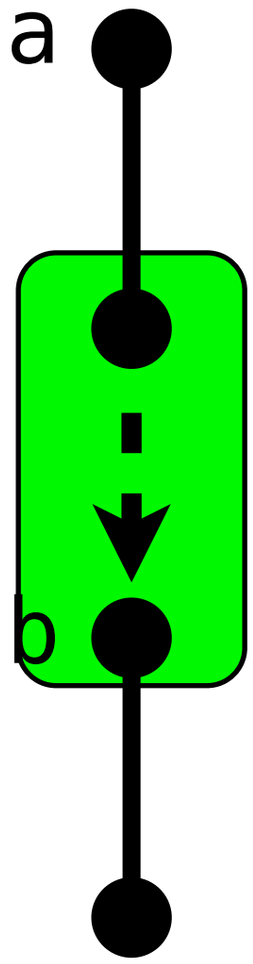
Schedule Model



Happens-Before Edges
partially order points.

```
a.end.addHb(b.start);
```

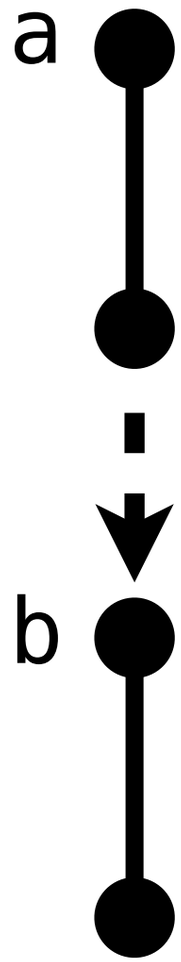
Schedule Model



Happens-Before Edges
partially order points.

```
a.end.addHb(b.start);
```

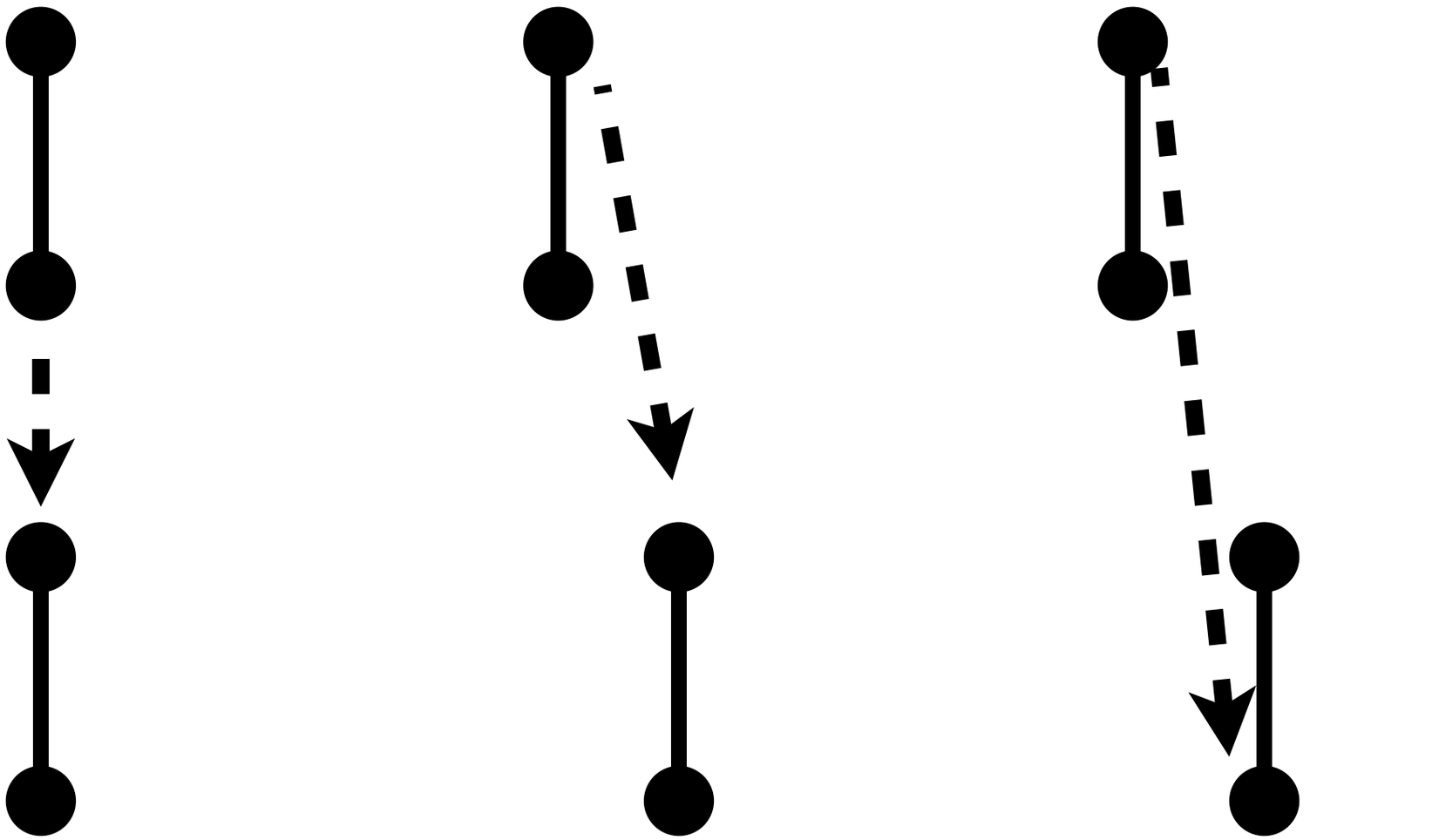
Schedule Model



Happens-Before Edges
partially order points.

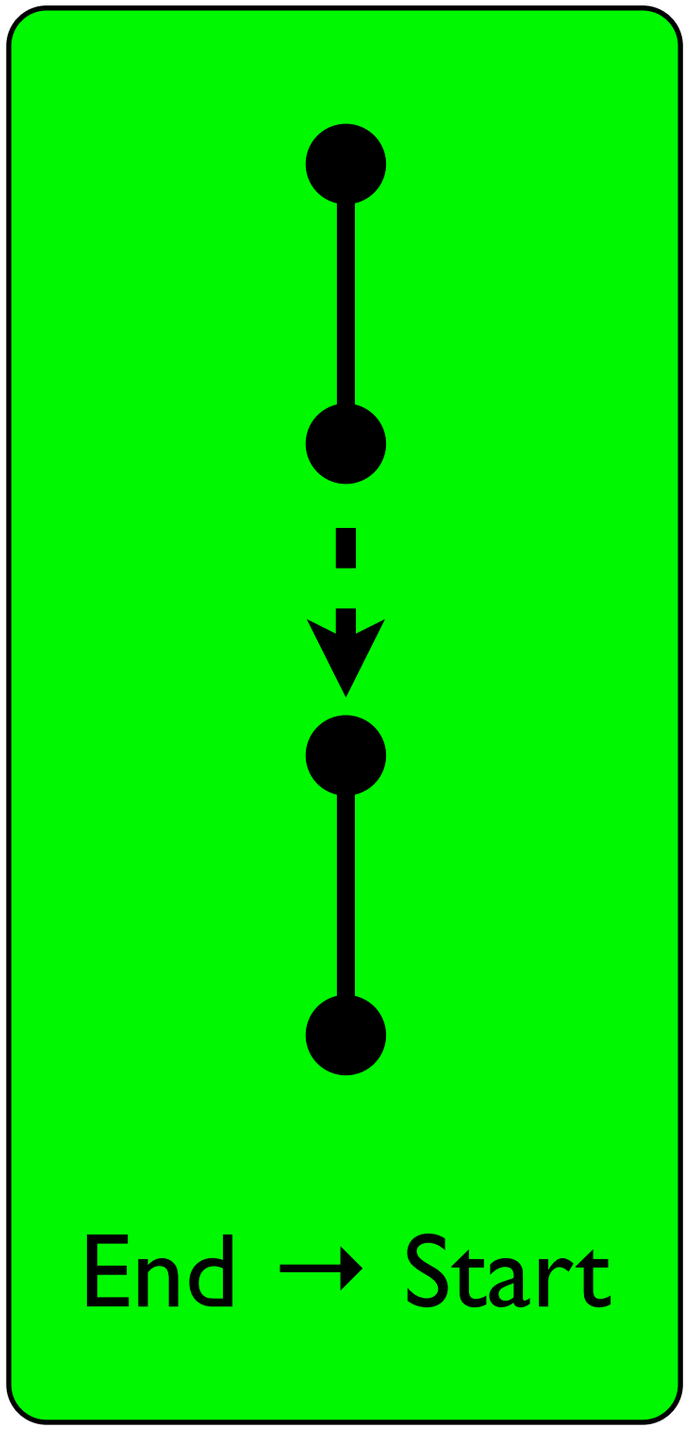
```
a.end.addHb(b.start);
```

Schedule Model



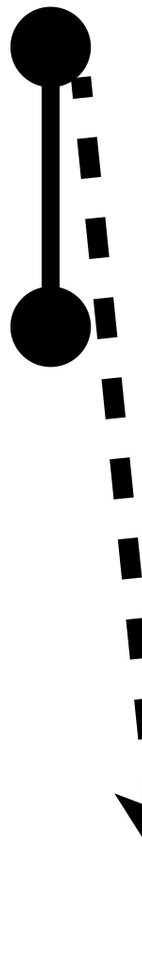
End → Start Start → Start Start → End

Schedule Model



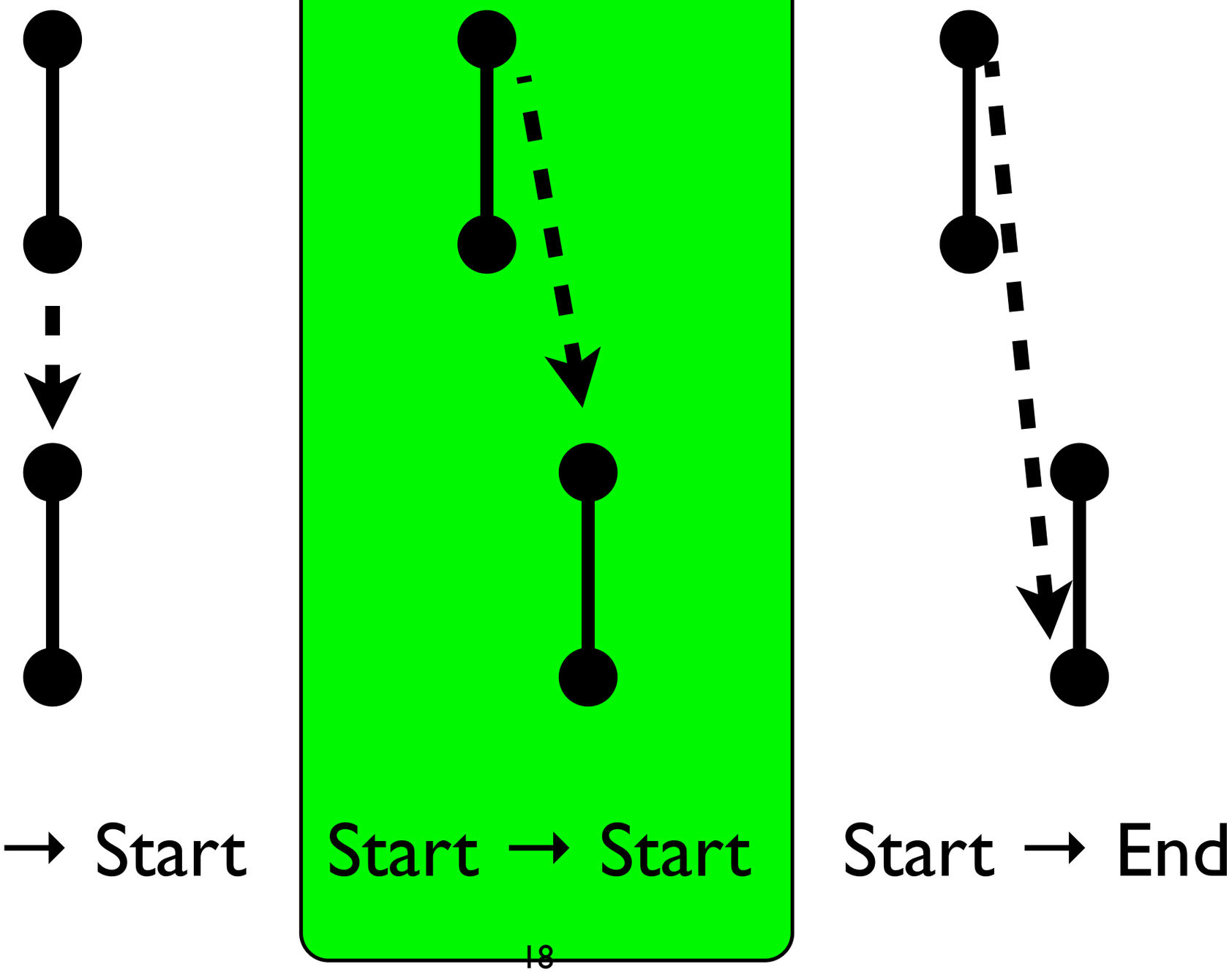
End → Start

Start → Start

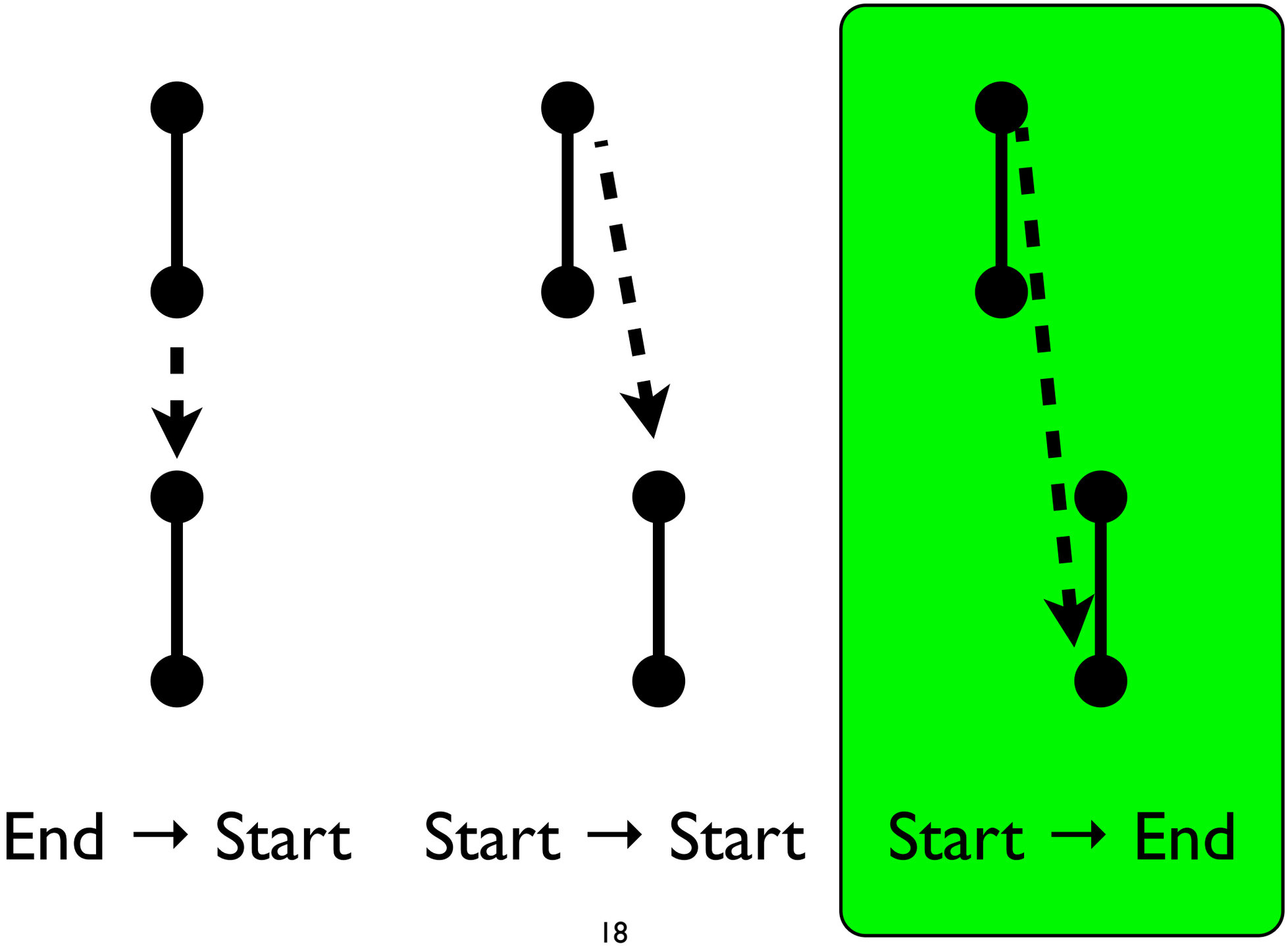


Start → End

Schedule Model



Schedule Model

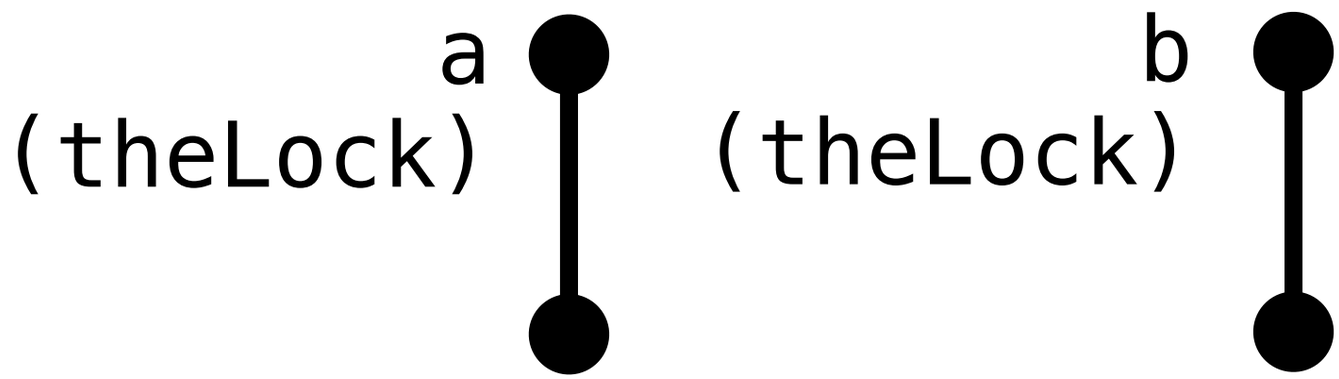


Schedule Model

Locks allow intervals to be sequential but unordered.

```
Lock lock = context.newLock();
```

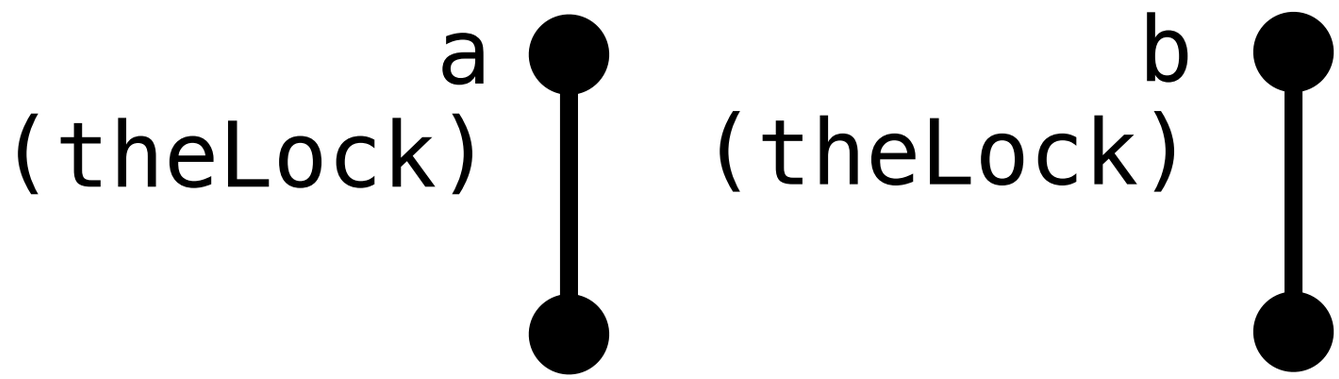
Schedule Model



Intervals may hold lock(s) for their duration.

```
a.addLock(theLock);
b.addLock(theLock);
```

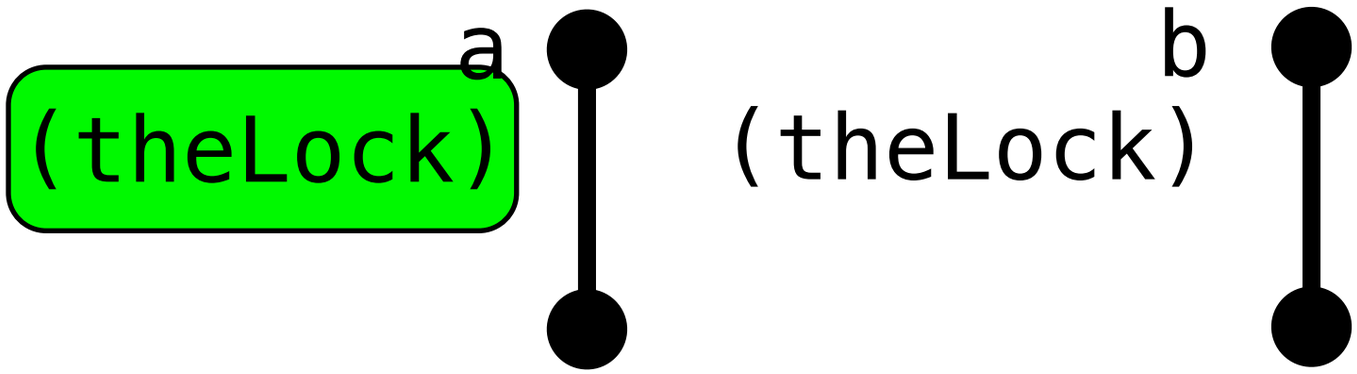
Schedule Model



Intervals may hold lock(s) for their duration.

```
a.addLock(theLock);
b.addLock(theLock);
```

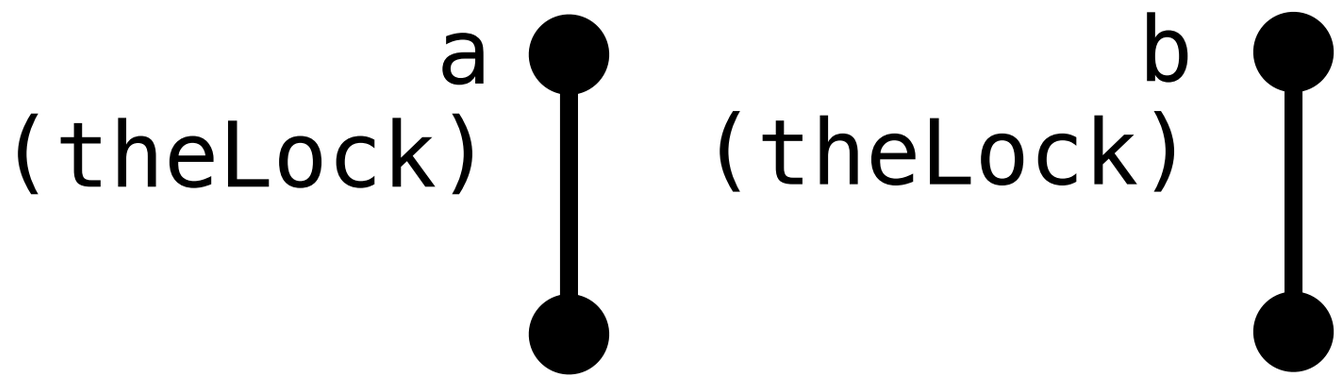
Schedule Model



Intervals may hold lock(s) for their duration.

```
a.addLock(theLock);
b.addLock(theLock);
```

Schedule Model



Intervals may hold lock(s) for their duration.

```
a.addLock(theLock);
b.addLock(theLock);
```

Schedule Model

```
Interval inter = ...;

// add edges, locks

inter.ready();
```

Invoked by creator of `inter` when initial dependencies have been added.

Schedule Model

```
Interval inter = ...;

// add edges, locks

inter.ready();
```

Invoked by creator of `inter` when initial dependencies have been added.

Schedule Model

```
Interval inter = ...;

// add edges, locks

inter.ready();
```

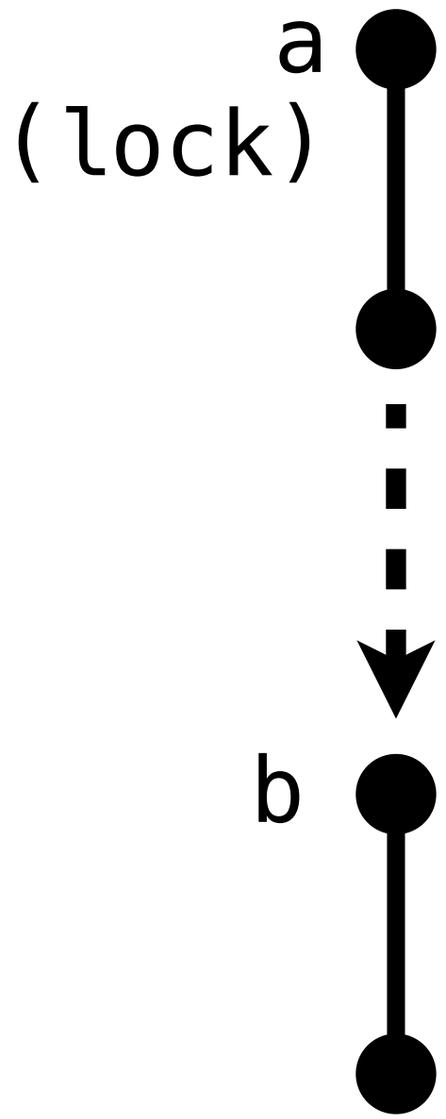
Invoked by creator of `inter` when initial dependencies have been added.

Summary

- Schedule Model
  - *Intervals* represent tasks
  - *Points* represents moments in time
  - *Happens-before edges* order points
  - *Locks* permit mutual exclusion of tasks

Querying the Schedule

```
a.end.hb(b.start)?

a.locks(lock)?

b.end.hb(a.start)?

b.locks(lock)?
```



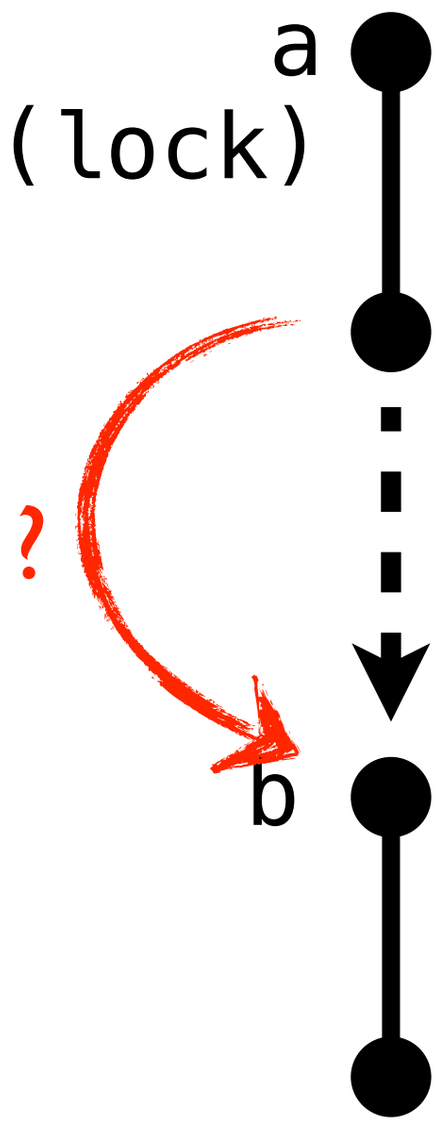
Querying the Schedule

`a.end.hb(b.start)?`

`a.locks(lock)?`

`b.end.hb(a.start)?`

`b.locks(lock)?`



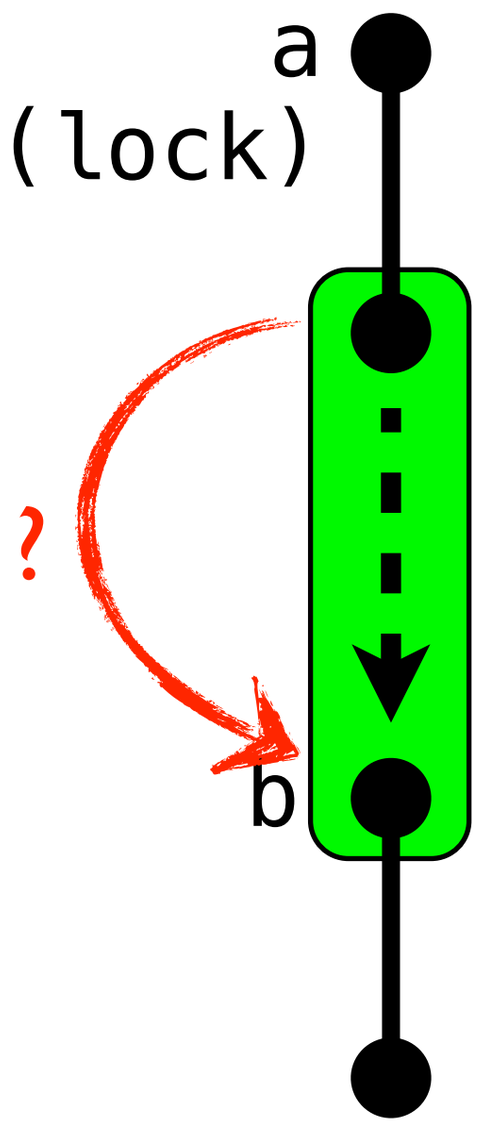
Querying the Schedule

`a.end.hb(b.start)?`

`a.locks(lock)?`

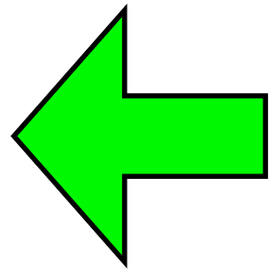
`b.end.hb(a.start)?`

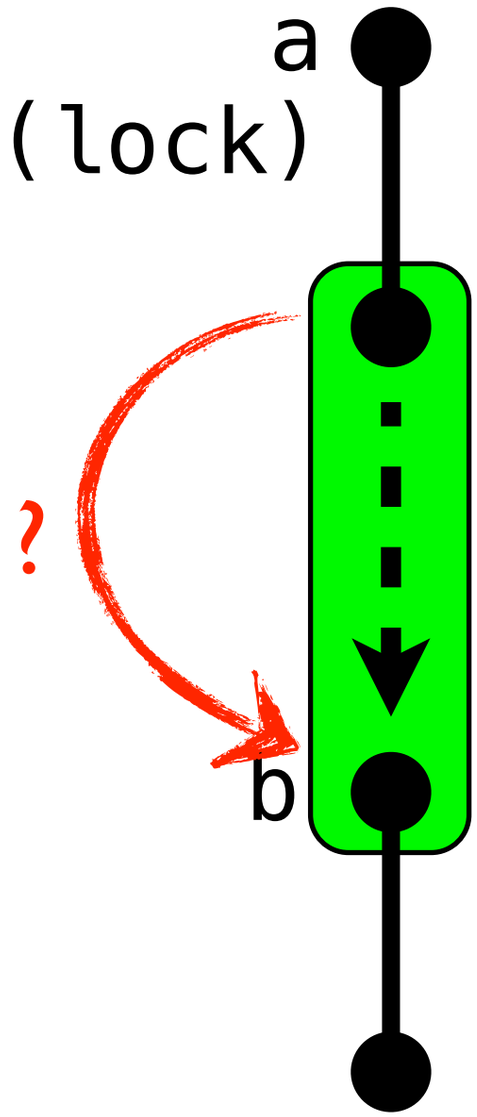
`b.locks(lock)?`



Querying the Schedule

```
a.end.hb(b.start)?
      true
```





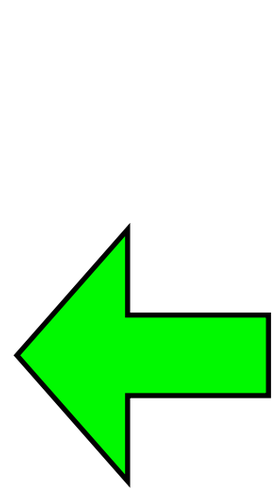

```
a.locks(lock)?
```

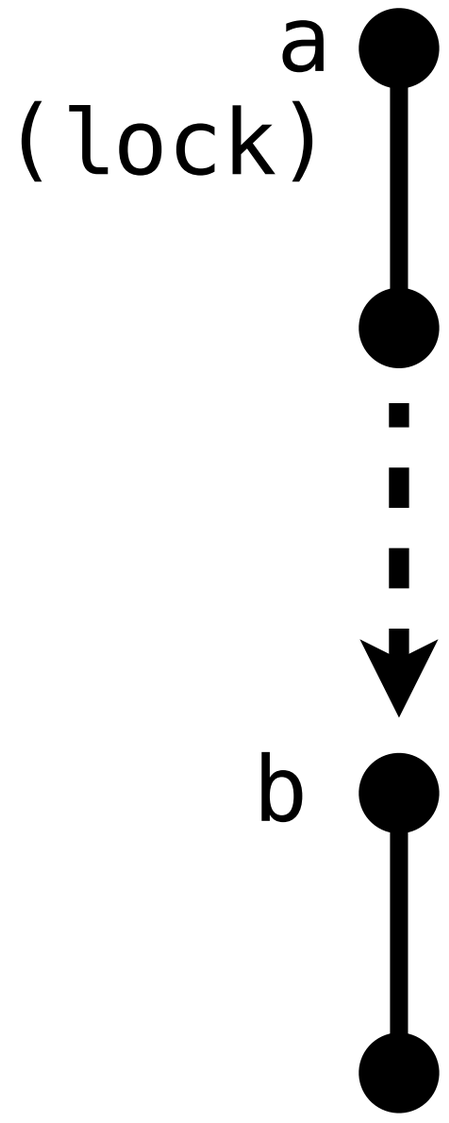
```
b.end.hb(a.start)?
```

```
b.locks(lock)?
```

Querying the Schedule

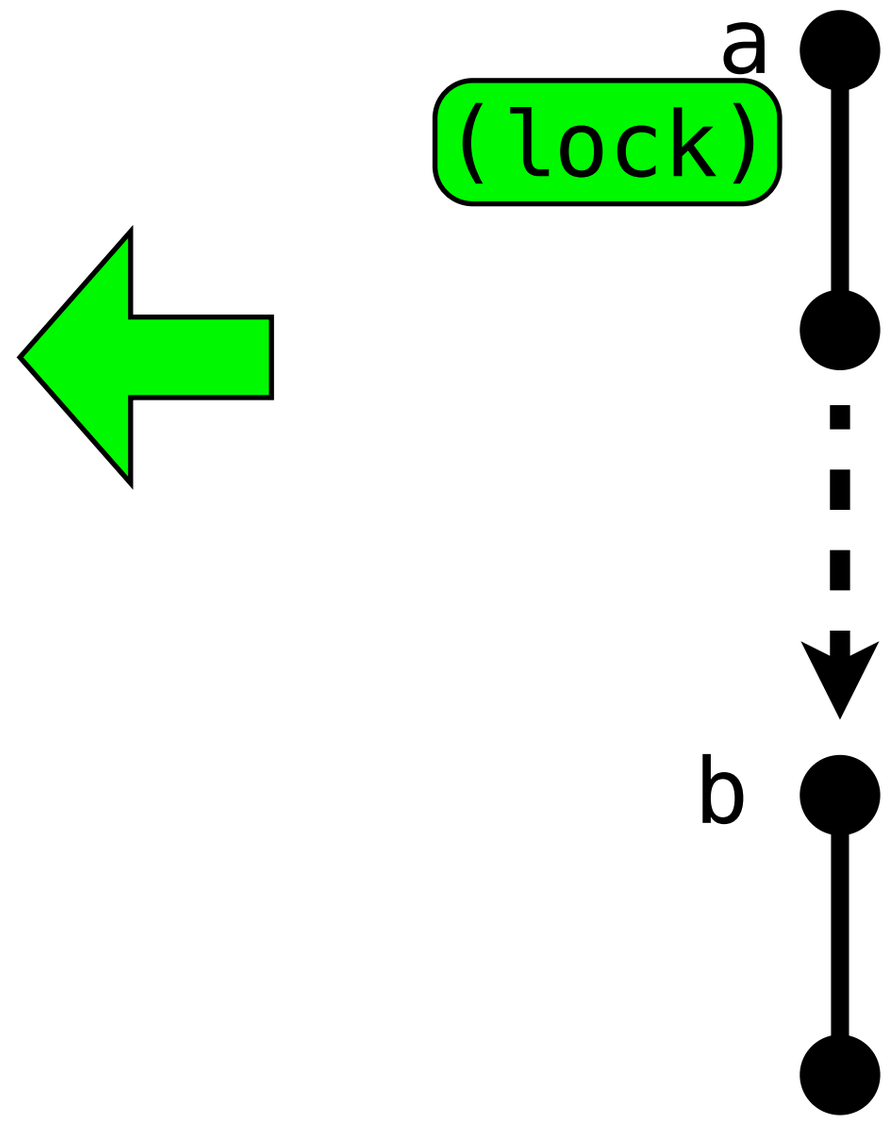
```
a.end.hb(b.start)?
     true

a.locks(lock)?

b.end.hb(a.start)?

b.locks(lock)?
```





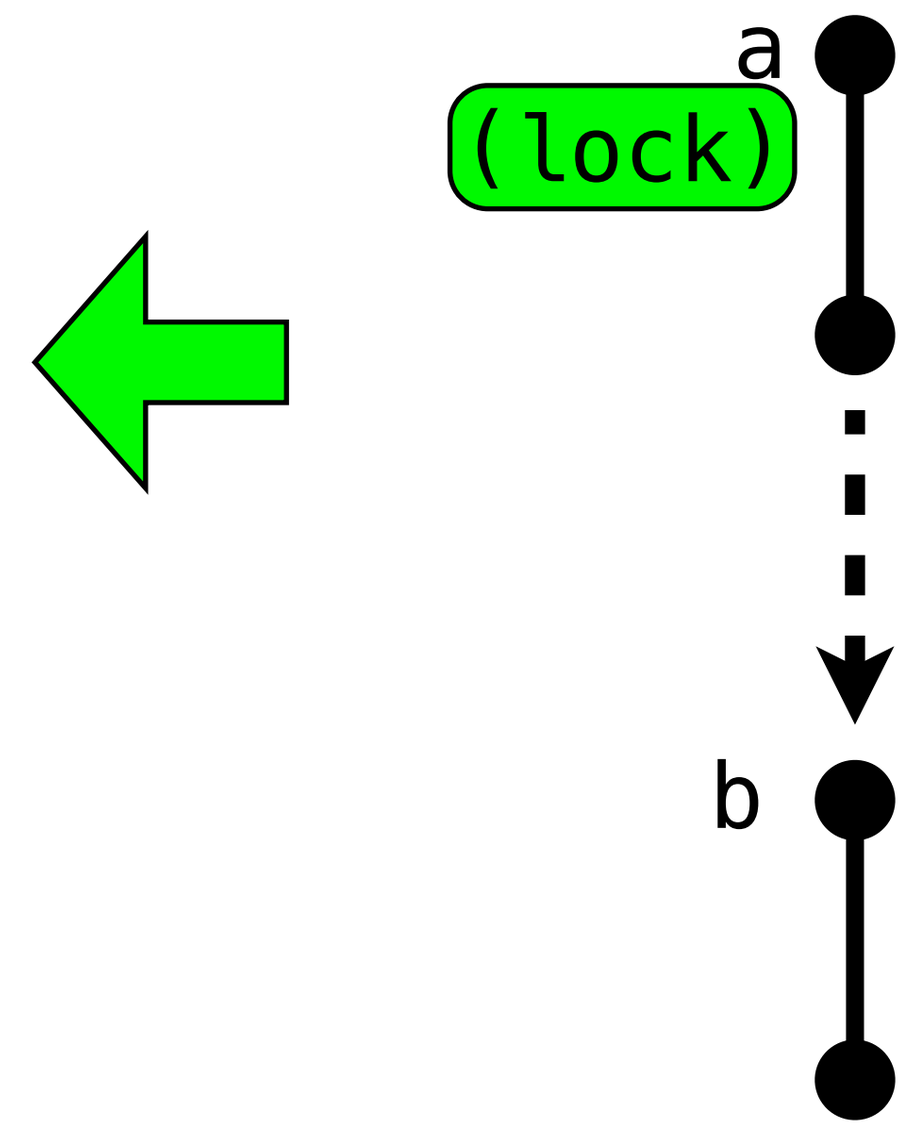
Querying the Schedule

```
a.end.hb(b.start)?
     true

a.locks(lock)?

b.end.hb(a.start)?

b.locks(lock)?
```



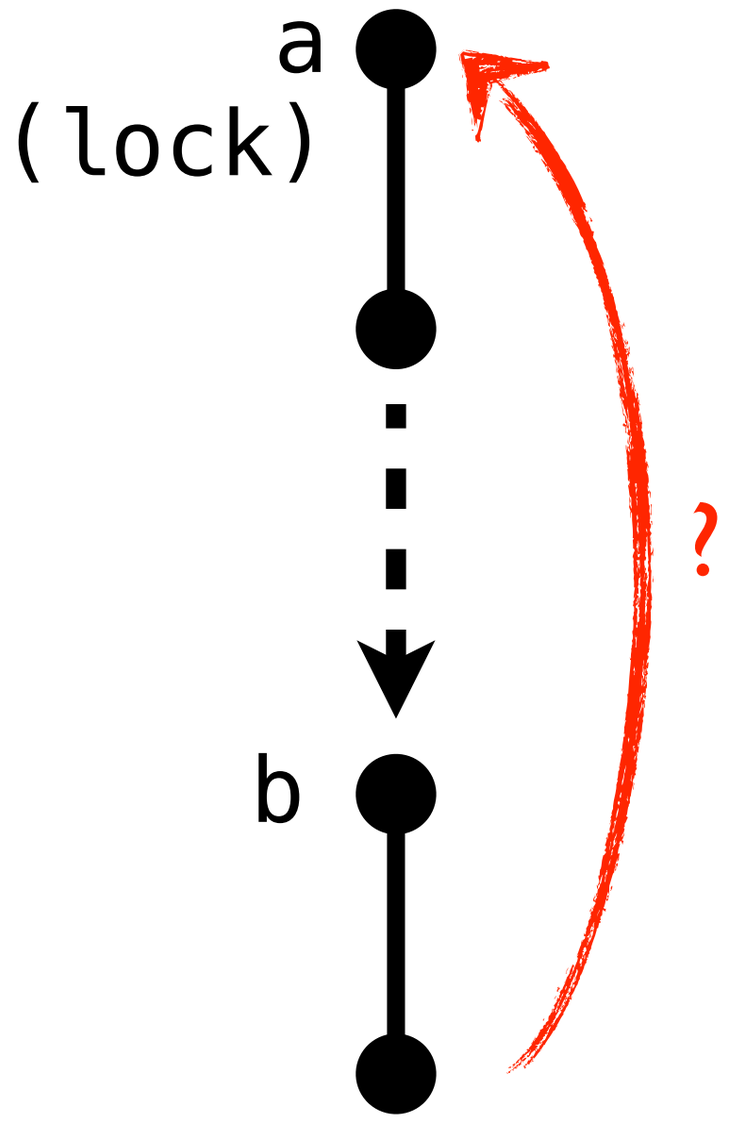
Querying the Schedule

```
a.end.hb(b.start)?
     true

a.locks(lock)?
     true

b.end.hb(a.start)?

b.locks(lock)?
```



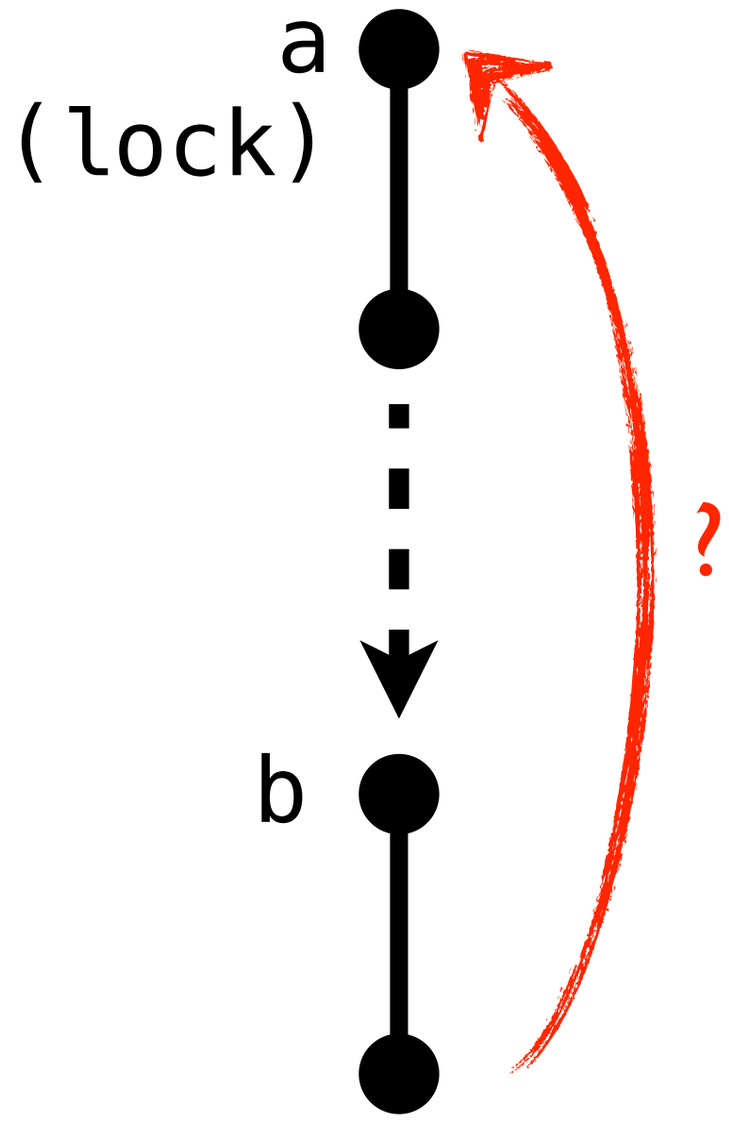
Querying the Schedule

```
a.end.hb(b.start)?
     true

a.locks(lock)?
     true

b.end.hb(a.start)?

b.locks(lock)?
```



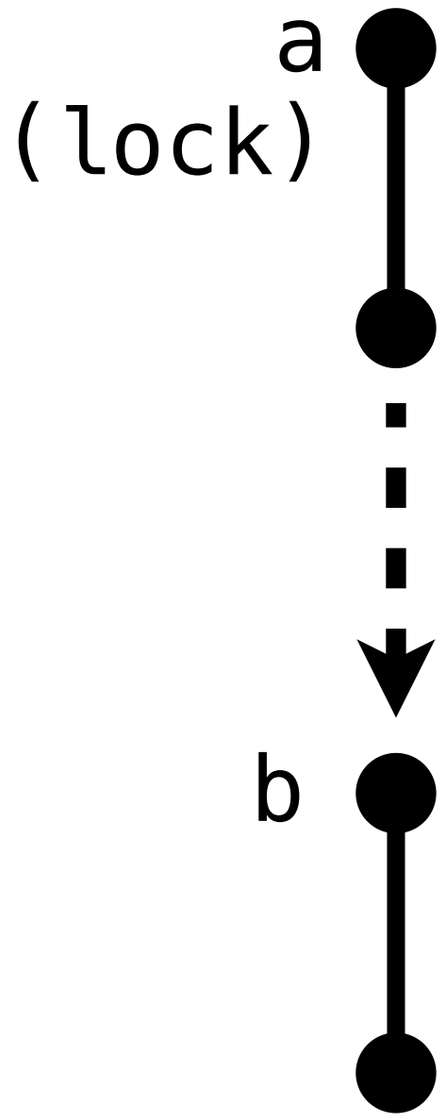
Querying the Schedule

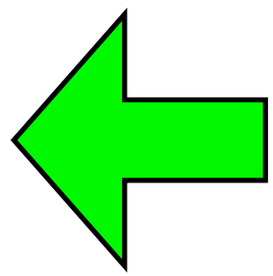
```
a.end.hb(b.start)?
     true

a.locks(lock)?
     true

b.end.hb(a.start)?
     false

b.locks(lock)?
```



Querying the Schedule

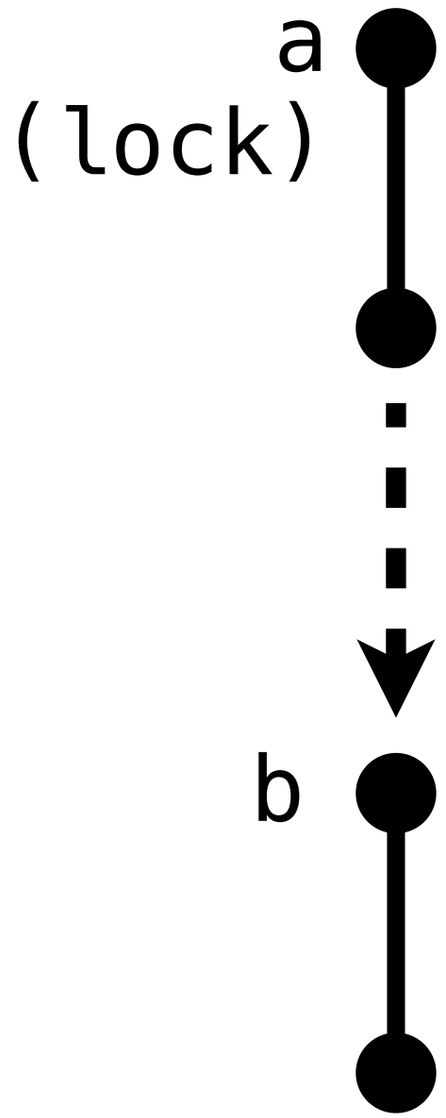
```
a.end.hb(b.start)?
     true

a.locks(lock)?
     true

b.end.hb(a.start)?
     false

b.locks(lock)?
```





Querying the Schedule

```
a.end.hb(b.start)?
    true

a.locks(lock)?
    true

b.end.hb(a.start)?
    false

b.locks(lock)?
    false
```



Monotonicity

- Edges and locks can only be added, not removed
- Necessary for static analysis:
  - Compiler knows that edges and locks it sees cannot be removed at runtime

Outline

- What is reflective parallel programming?
- Why do we need a new model?
- Intervals model
- Example: Data-race detection

Outline

- What is reflective parallel programming?
- Why do we need a new model?
- Intervals model
- Example: Data-race detection

Data Race Detection

- Key Idea:
  - User defines conditions in which a field can be accessed
  - Use the reflective API to determine whether conditions are met

Locked Fields

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Locked Fields

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Locked Fields

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Locked Fields

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Locked Fields

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Dynamic Checking

```
void setString() {
  assert current.locks(theLock);
  theString = "...";
}
```

Dynamic Checking

```
void setString() {
    assert current.locks(theLock);
    theString = "...";
}
```

Dynamic Checking

```
void setString() {
   assert current.locks(theLock);
   theString = "...";
}
```

Dynamic Checking

```
void setString() {
  assert current.locks(theLock);
  theString = "...";
}
```

Static Checking

```
void staticCheck() {
  Interval x = interval {
    assert current.locks(theLock);
    theString = "...";
  }
  x.addLock(theLock);
  x.ready();
}
```

Static Checking

```
void staticCheck() {
   Interval x = interval {
      assert current.locks(theLock);
      theString = "...";
   }
   x.addLock(theLock);
   x.ready();
}
```

Static Checking

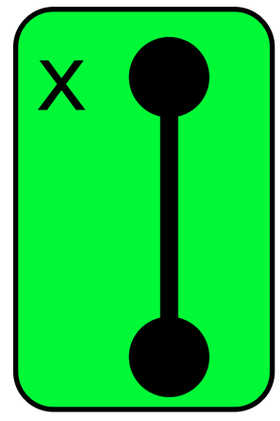
```
void staticCheck() {
   Interval x = interval {
      assert current.locks(theLock);
      theString = "...";
   }
   x.addLock(theLock);
   x.ready();
}
```

Static Checking

```
void staticCheck() {
   Interval x = interval {
      assert current.locks(theLock);
      theString = "...";
   }
   x.addLock(theLock);
   x.ready();
}
```

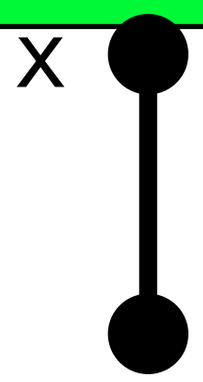
Static Checking

```
void staticCheck() {
   Interval x = interval {
      assert current.locks(theLock);
      theString = "...";
   }
   x.addLock(theLock);
   x.ready();
}
```



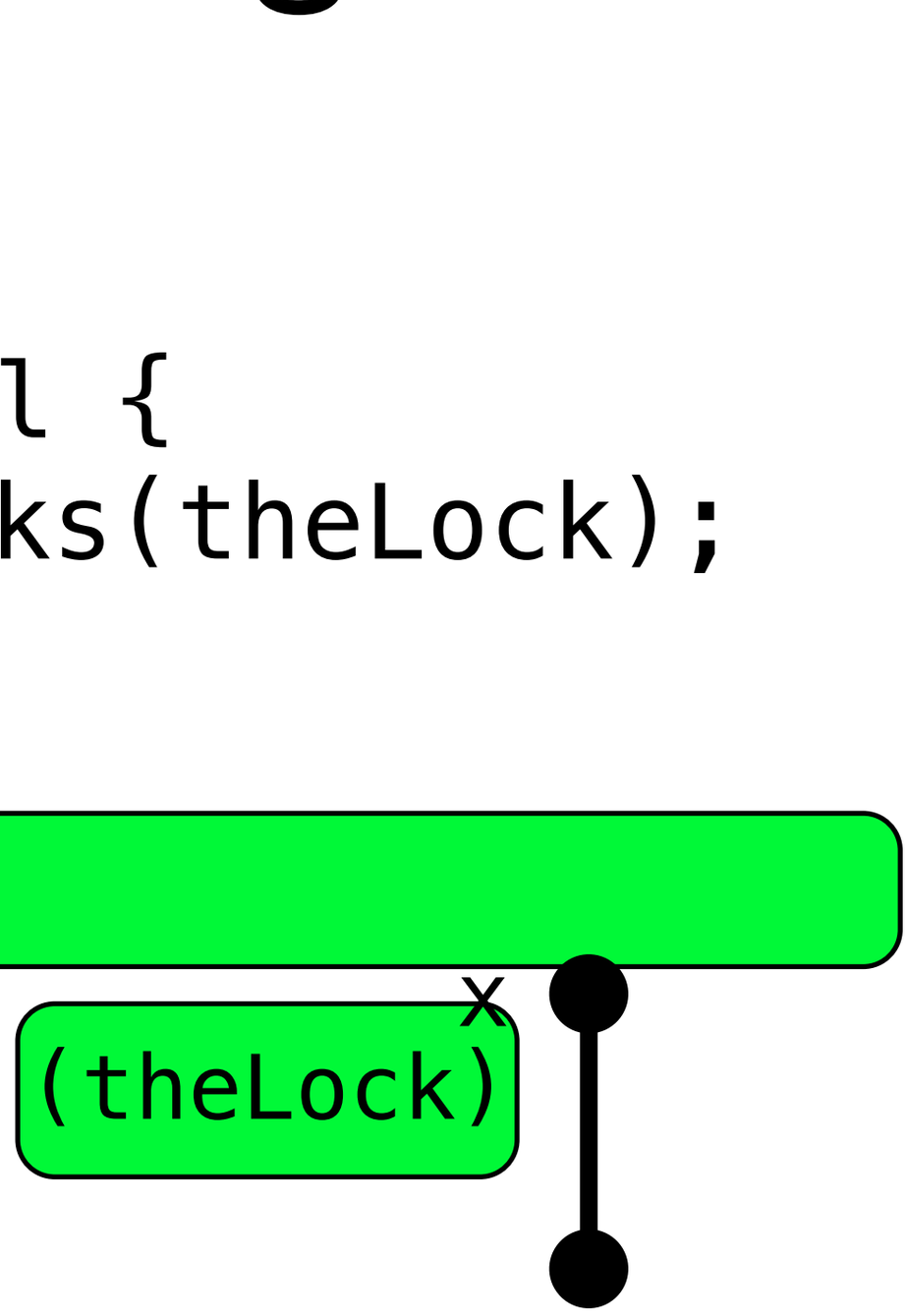
Static Checking

```
void staticCheck() {
  Interval x = interval {
    assert current.locks(theLock);
    theString = "...";
  }
  x.addLock(theLock);
  x.ready();
}
```



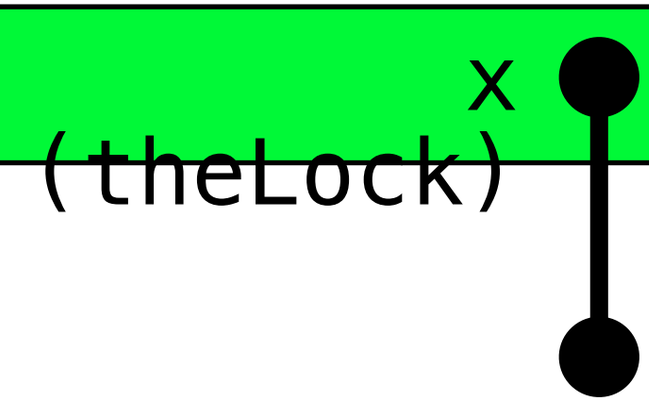
Static Checking

```
void staticCheck() {
  Interval x = interval {
    assert current.locks(theLock);
    theString = "...";
  }
  x.addLock(theLock);
  x.ready();
}
```



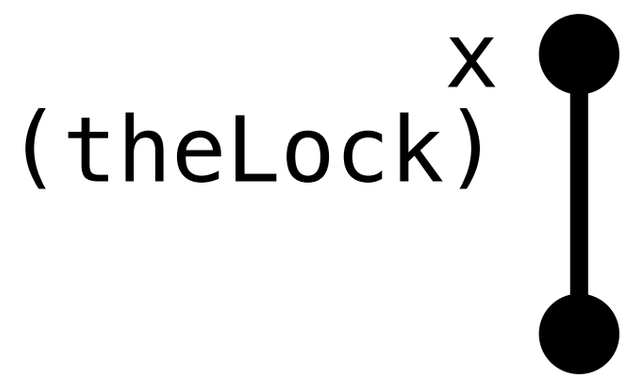
Static Checking

```
void staticCheck() {
  Interval x = interval {
    assert current.locks(theLock);
    theString = "...";
  }
  x.addLock(theLock);
  x.ready();
}
```



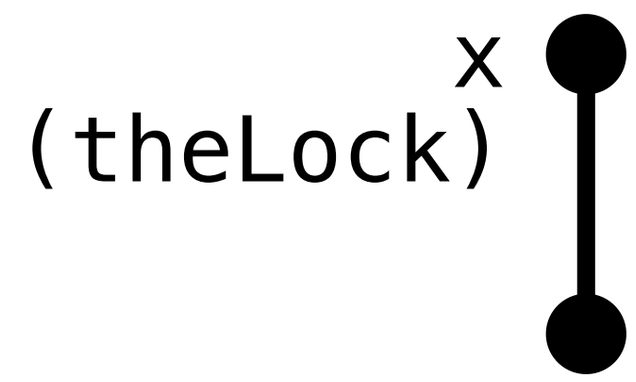
Static Checking

```
void staticCheck() {
   Interval x = interval {
      assert current.locks(theLock);
      theString = "...";
   }
   x.addLock(theLock);
   x.ready();
}
```



Static Checking

```
void staticCheck() {
  Interval x = interval {
    assert current.locks(theLock);
    theString = "...";
  }
  x.addLock(theLock);
  x.ready();
}
```



Guard Objects

- Our compiler automatically enforces these kind of checks using **guard objects**
- Guard object defines methods that check each read and write for validity
- When possible, checks are performed statically

Guard Object Annotations

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Guard Object Annotations

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Guard Object Annotations

```
class TheClass {

    final Lock theLock;

    @GuardedBy(theLock)
    String theString;

    ...
}
```

Custom Guards

- Example Conditions
  - Written only by one interval
  - Dynamic monitoring
  - Lock only on writes, not reads
  - Select lock dynamically

Summary

- User defines conditions to access a field by writing code against the reflective API
- Compiler runs checks statically if possible
- Runtime can run checks with live schedule

Related Work

- Smalltalk
  - Reflective objects for stack frames, etc
  - Debuggers and other tools require no special support from VM
  - Traditional threading model
- More in the paper

Conclusion

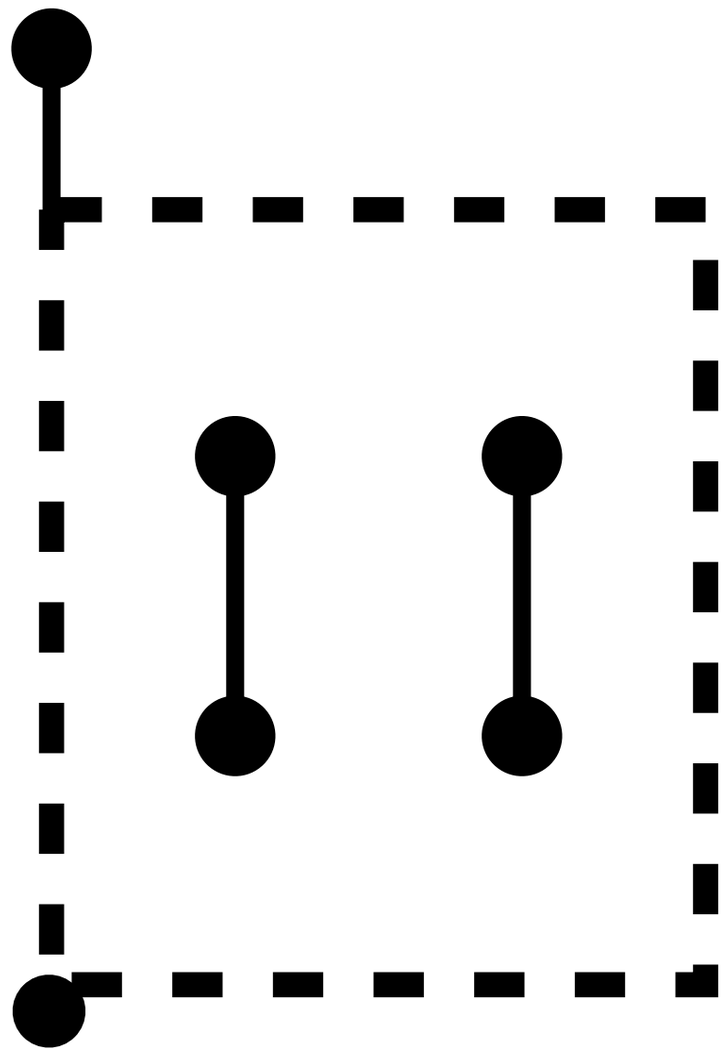
- Reflective parallelism empowers users:
  - Custom tools for safety checking and monitoring
- Reflective parallelism as foundation for static analysis:
  - Seamless integration of static and dynamic checks

Thank You

- Intervals library is available for download:
  - http://intervals.inf.ethz.ch

Spare Slides

Schedule Model



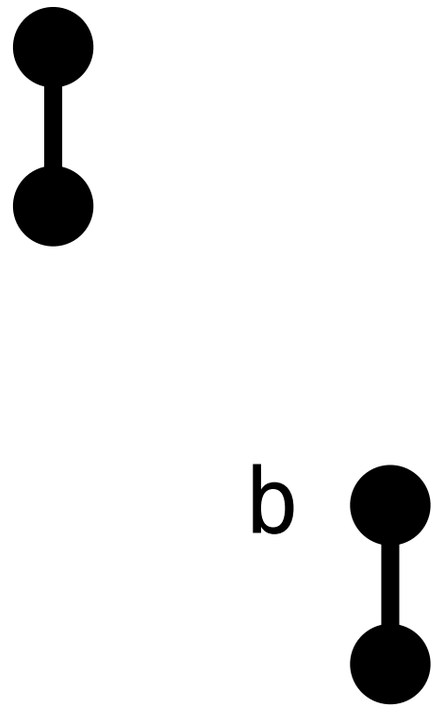
Hierarchical Structure

Illegal Additions

- Schedule is being built and executed simultaneously
- Certain additions are illegal

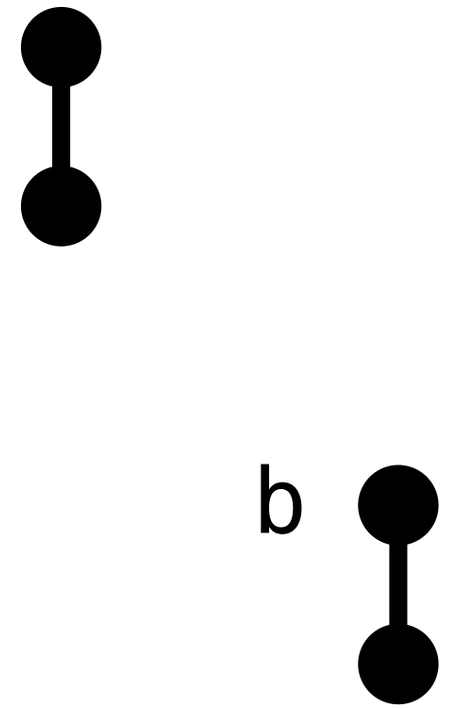
Adding Edges

```
void method(
    Interval a,
    Interval b)
{
    a.end.addHb(b.start);
}
```



Adding Edges

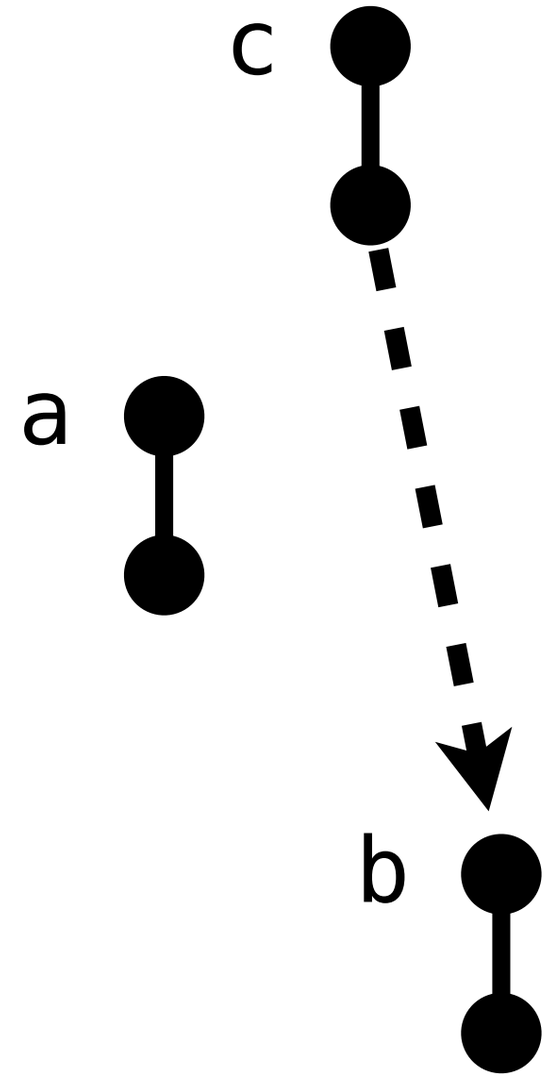
```
void method(
    Interval a,
    Interval b)
{
    a.end.addHb(b.start);
}
```



What if b had already begun execution?

Adding Edges

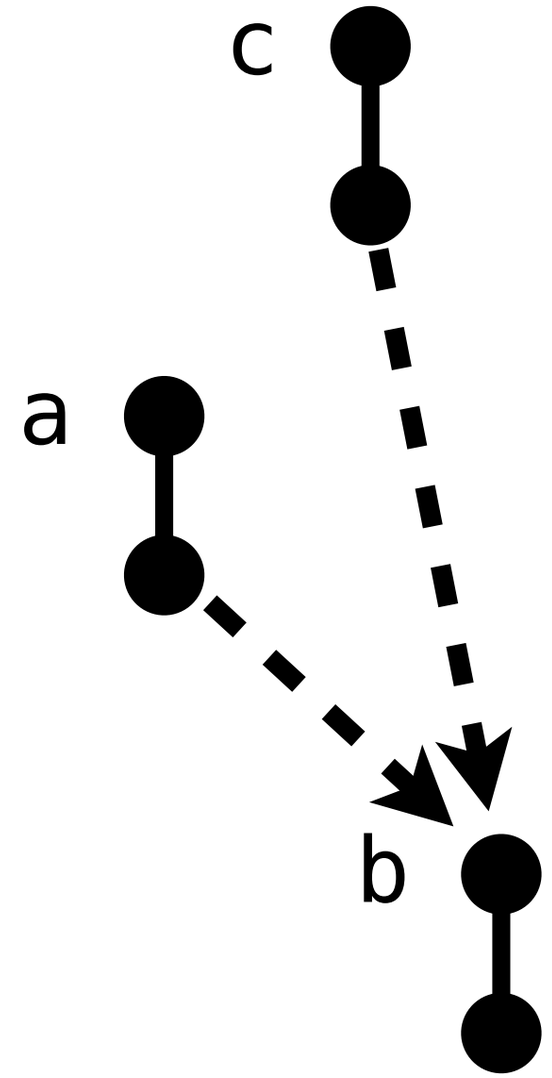
```
void method(
    Interval a,
    Interval b)
{
    a.end.addHb(b.start);
}
```



If method is part of c, b cannot have started.

Adding Edges

```
void method(
    Interval a,
    Interval b)
{
    a.end.addHb(b.start);
}
```



If method is part of c, b cannot have started.

Point to Point

```
// Signal this thread is done
sync[id]++;

// Wait for neighbors;
while(sync[id-1] < sync[id])
   ;
while(sync[id+1] < sync[id])
   ;
```

Point to Point

```
    // Signal this thread is done
    sync[id]++;

    // Wait for neighbors;
    while(sync[id-1] < sync[id])
        ;
    while(sync[id+1] < sync[id])
        ;
```

Point to Point

```
// Signal this thread is done
sync[id]++;

// Wait for neighbors;
while(sync[id-1] < sync[id])
   ;
while(sync[id+1] < sync[id])
   ;
```

Point to Point

```
// Signal this thread is done
sync[id]++;

// Wait for neighbors;
while(sync[id-1] < sync[id])
    ;
while(sync[id+1] < sync[id])
    ;
```

Point to Point

```
// Signal this thread is done
sync[id]++;

// Wait for neighbors;
while(sync[id-1] < sync[id])
   ;
while(sync[id+1] < sync[id])
   ;
```

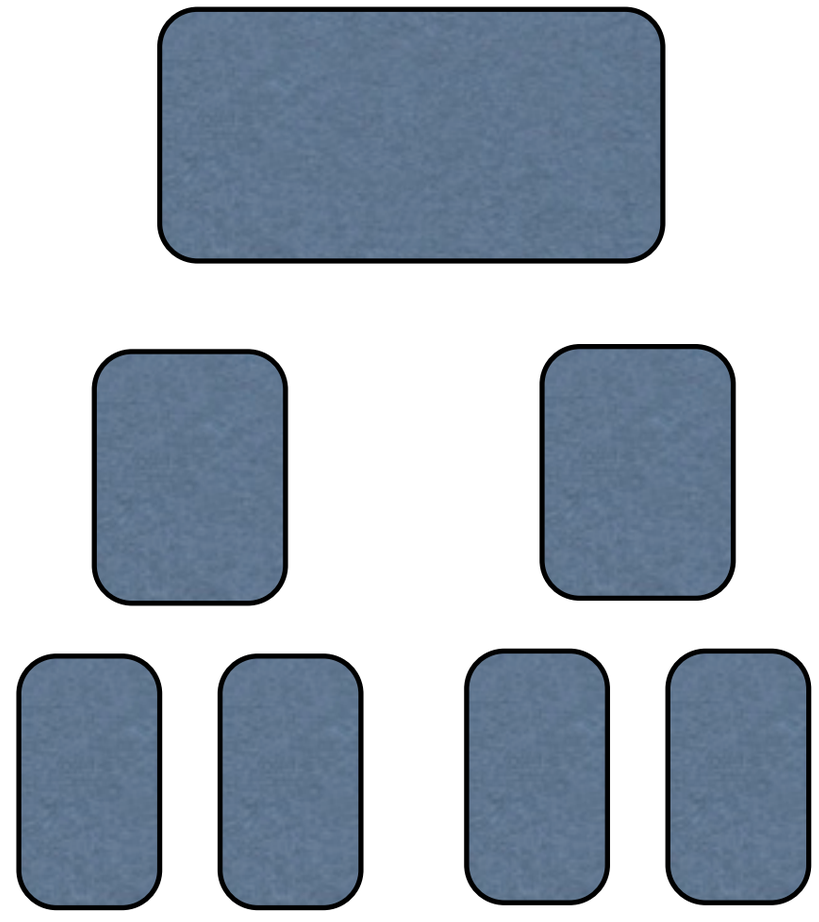
Point to Point

```
// Signal this thread is done
sync[id]++;

// Wait for neighbors;
while(sync[id-1] < sync[id])
   ;
while(sync[id+1] < sync[id])
   ;
```

Complex patterns are even worse.

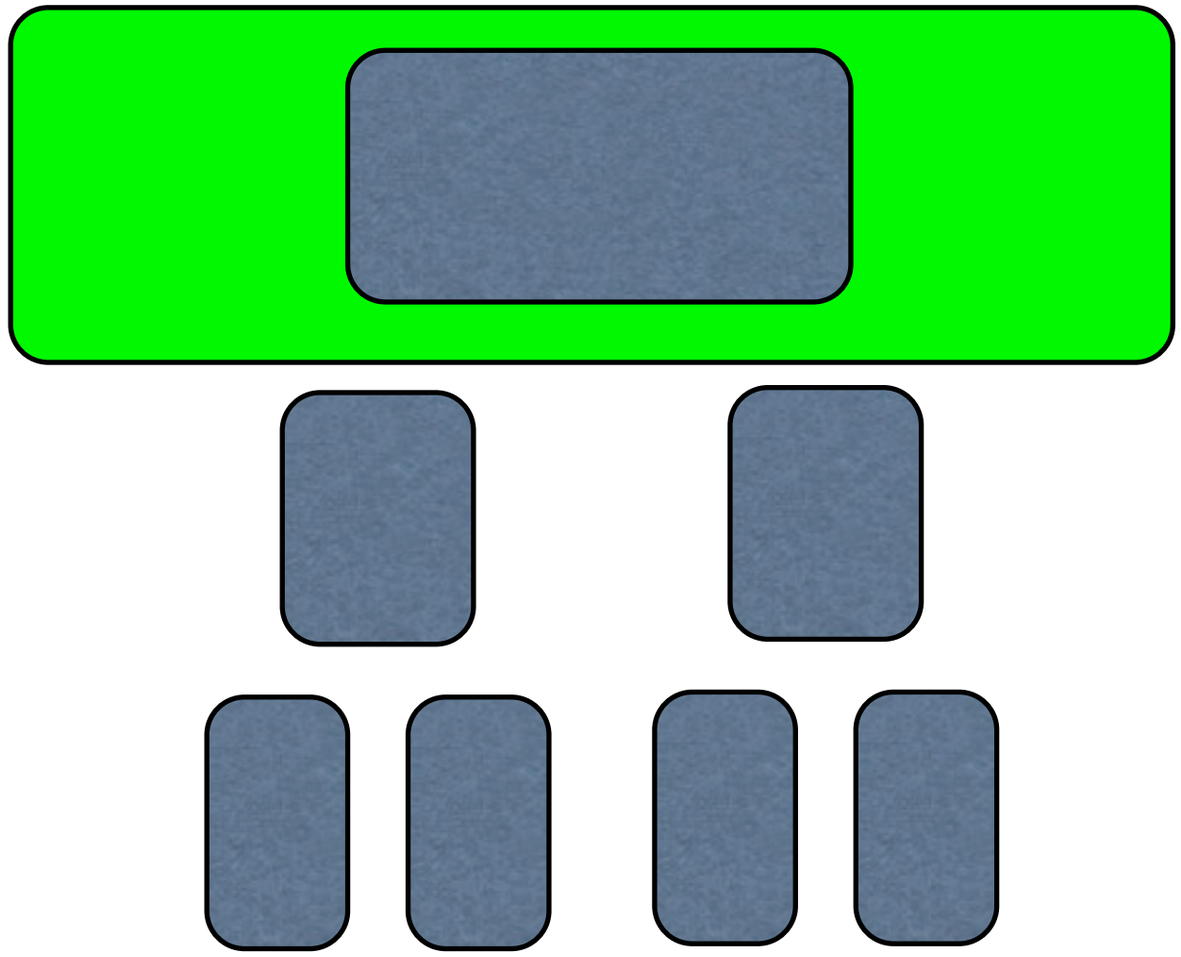
Quake Lock

- Game map represented as tree
- Lock depends on location in volume tree



Quake Lock

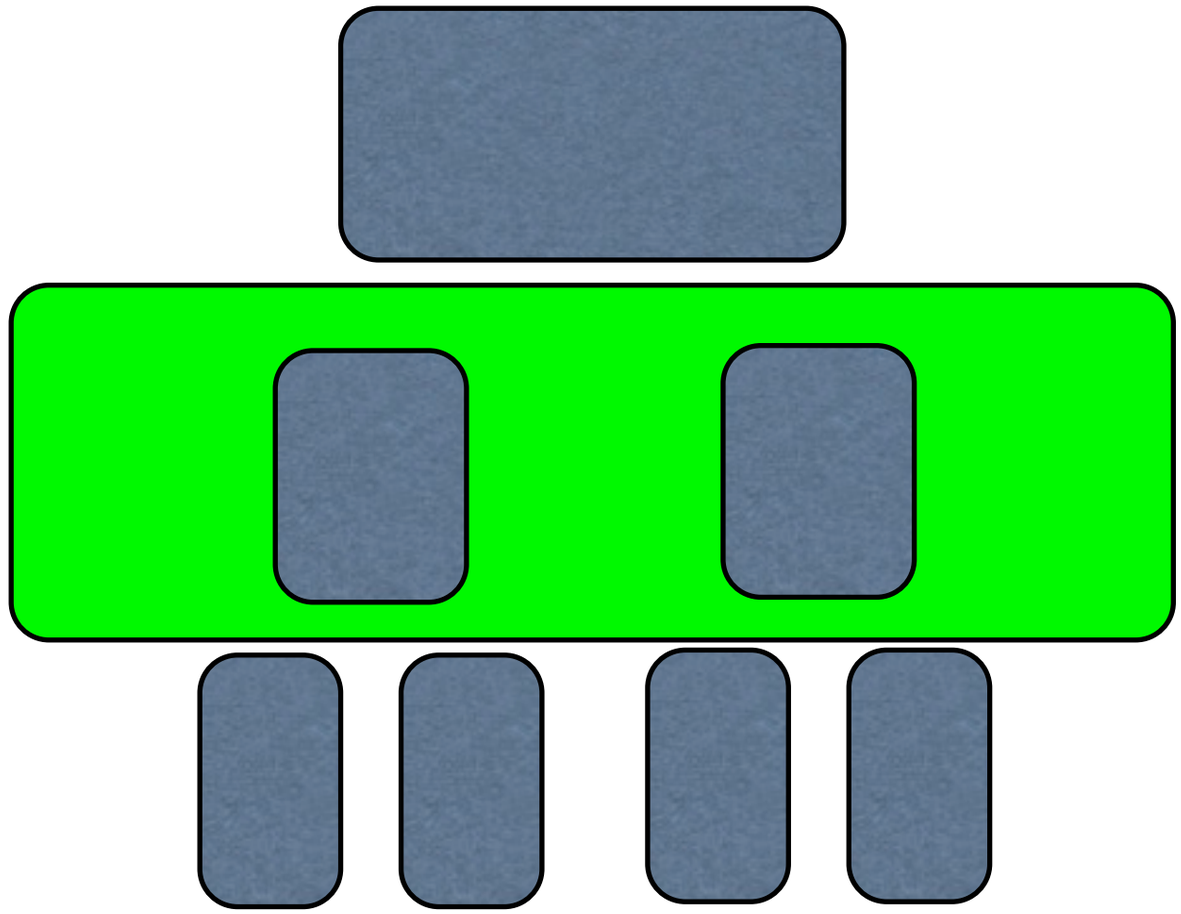
- Game map represented as tree
- Lock depends on location in volume tree



Quake Lock

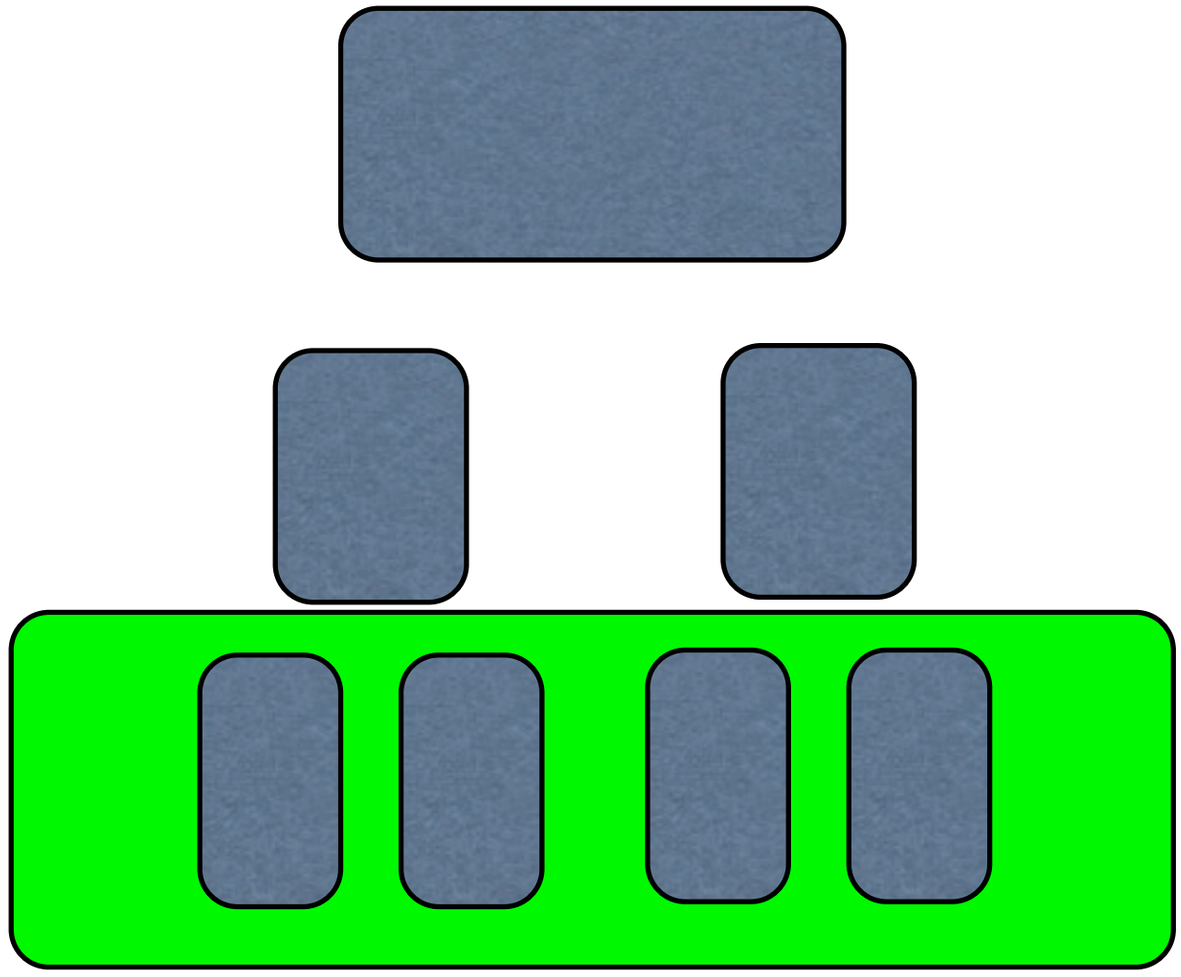
- Game map represented as tree

- Lock depends on location in volume tree



Quake Lock

- Game map represented as tree
- Lock depends on location in volume tree



Mirrors

- Mirrors allow many implementations
  - Compile-time approximations / previews
  - Programs on a different machine

Guard Interface

```
class Guard {

  void checkRead(Interval inter);

  void checkWrite(Interval inter);

}
```

Guard Interface

```
class Guard {

   void checkRead(Interval inter);

   void checkWrite(Interval inter);

}
```

Guard Interface

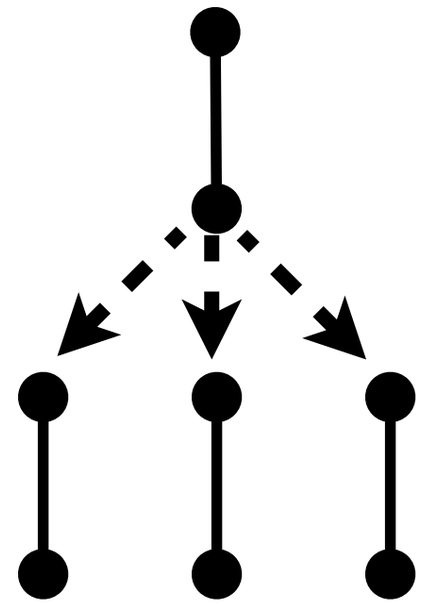
```
class Guard {

  void checkRead(Interval inter);

  void checkWrite(Interval inter);

}
```

Reflective Parallelism with Intervals

- Query and manipulate program schedule both statically and during execution
- "Roll your own" data-race detectors and other tools

Reflective Parallelism with Intervals

- Query and manipulate program schedule both statically and during execution
- "Roll your own" data-race detectors and other tools



Reflective Parallelism with Intervals

- Query and manipulate program schedule both statically and during execution
- "Roll your own" data-race detectors and other tools