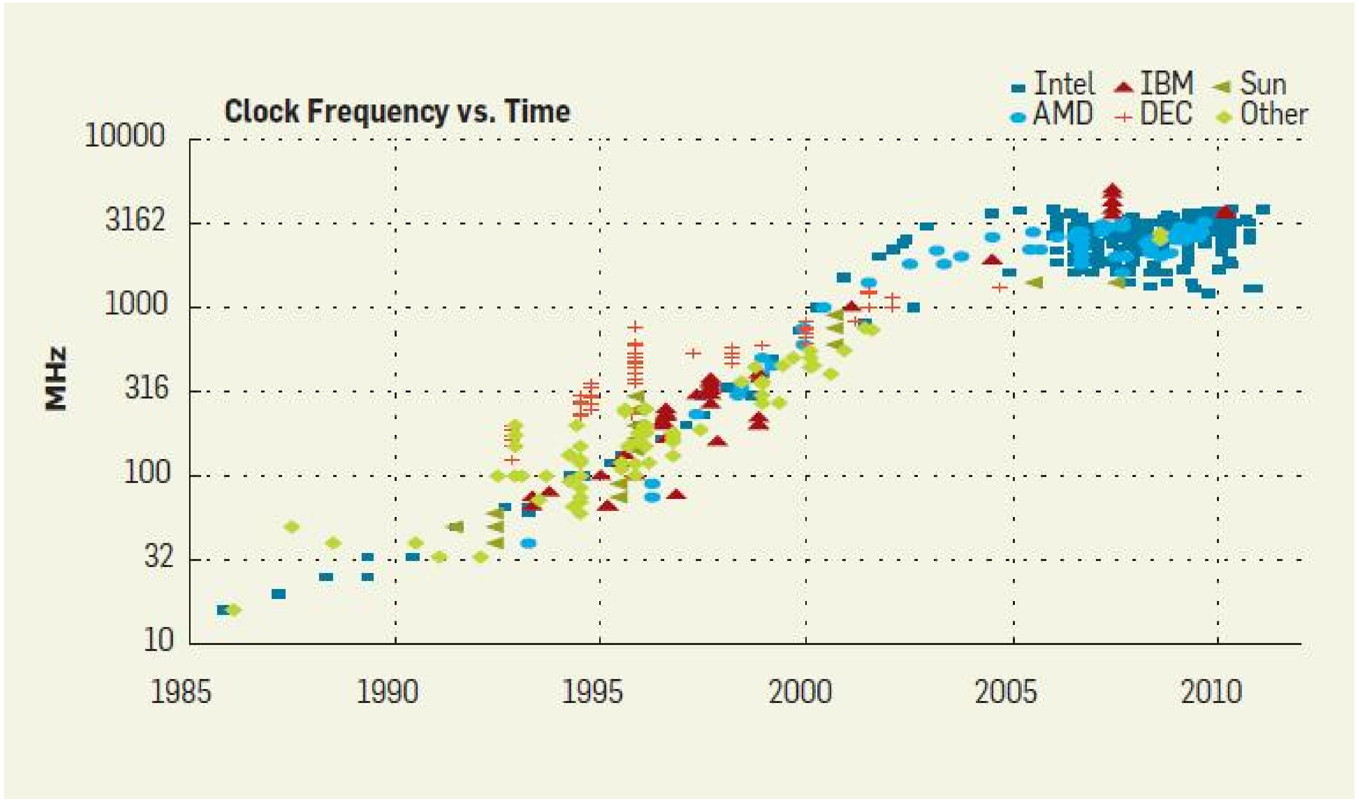
Lock-Free Fixpoint Algorithms

Jesper Öqvist
PhD Student
Lund University, Sweden

The Rise of Multicore



CACM,
CPU DB: Recording Microprocessor History

The Rise of Multicore

Clock speeds stagnated around 3 GHz in 2005.

Performance gains now come from parallelism.

Current generation CPUs

- AMD Ryzen Threadripper: 8-16 cores
- Intel Core i9: 10-18 cores

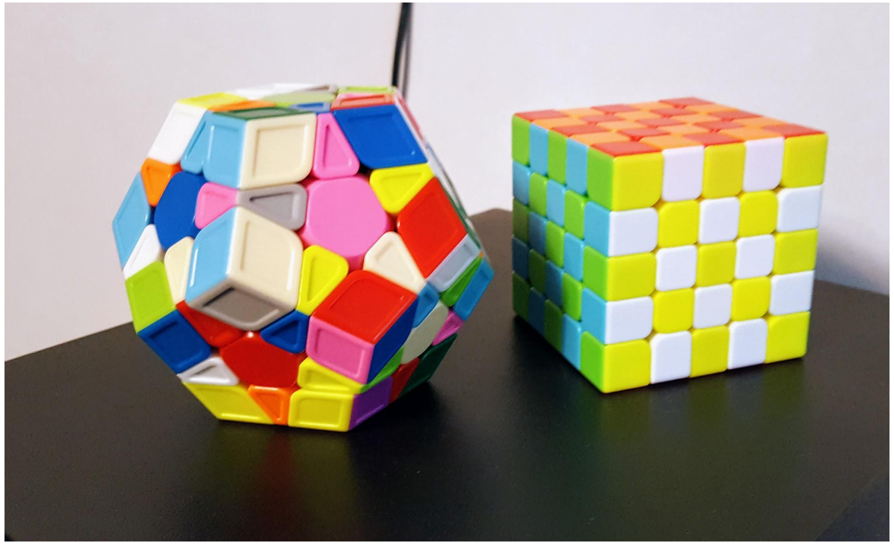
Parallelization

Parallel algorithms are

- complicated,
- hard to develop, and,
- very difficult to ensure correct.

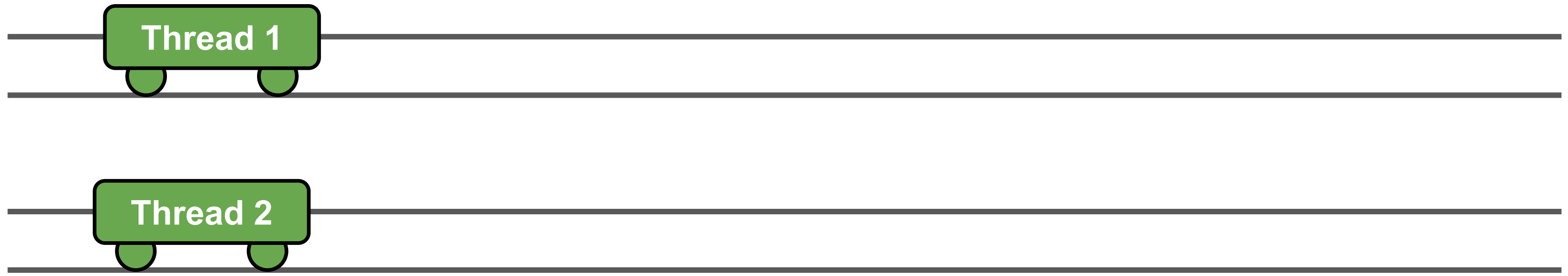
I like problems.

Hopefully you do, too!



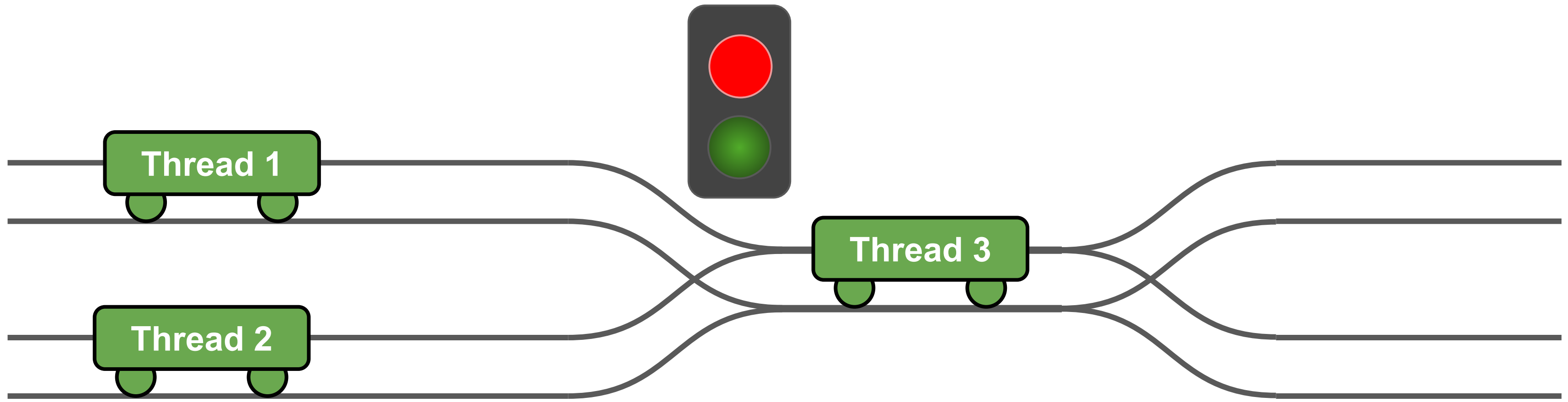
Parallel Programming

We can view threads as trains:



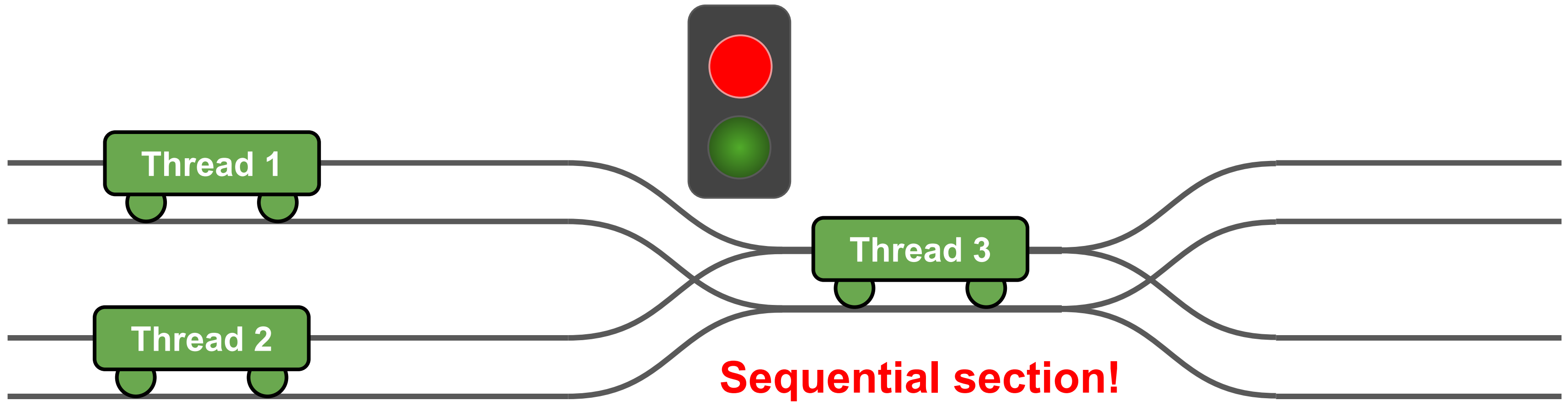
The computation they perform is the length of track covered.

Parallel Programming



Synchronization happens at intersections.

Parallel Programming



Locks force sequential execution.

Intermission: Amdahl’s Law

The potential speedup of parallelizing depends on the fraction of the work that can be parallelized.

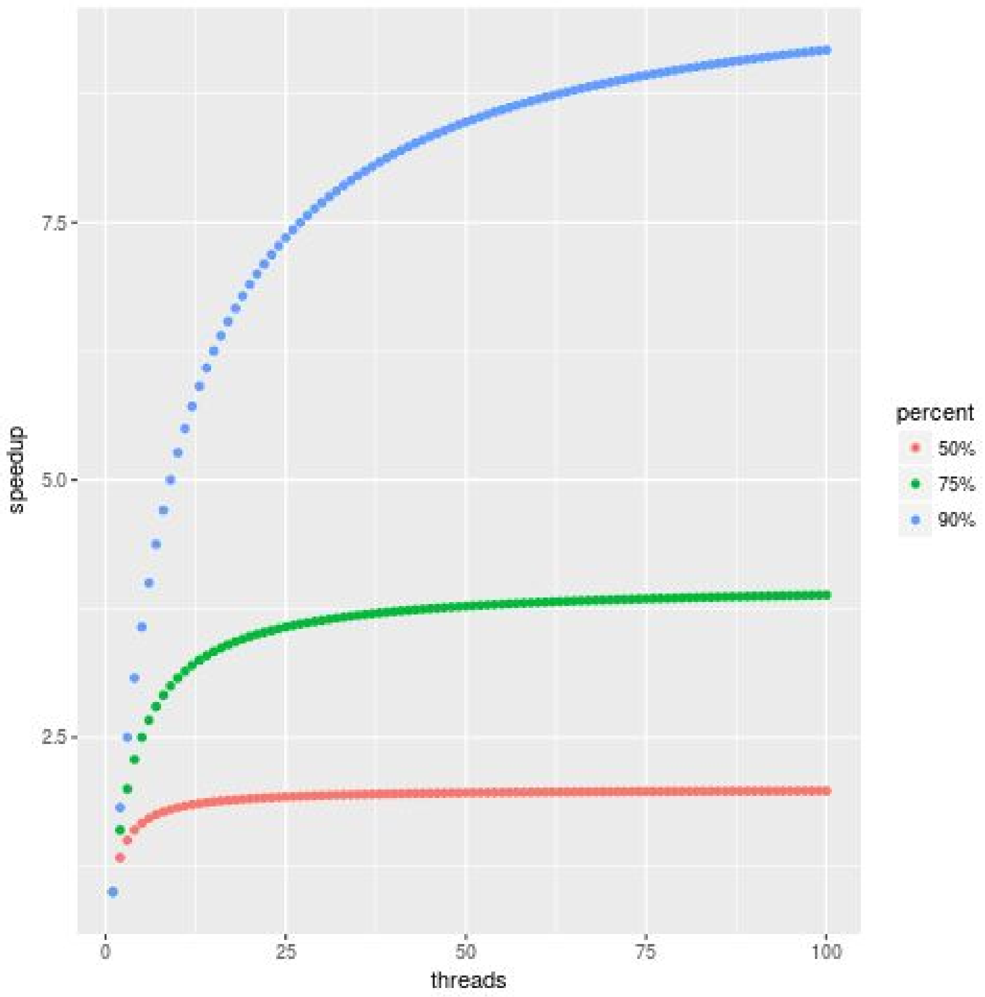
Amdahl’s law:

Speedup S from n-way parallelization:

$S = 1 / (1 - p + p/n)$

Where p is the fraction of work that can be executed in parallel.

Amdahl's Law



If 1/10th of the work is sequential, speedup limit is 10x.

Lock-Freedom

The goal of lock-free programming is to remove sequential bottlenecks.

☞ Absence of locks is not absence of synchronization/ordering.

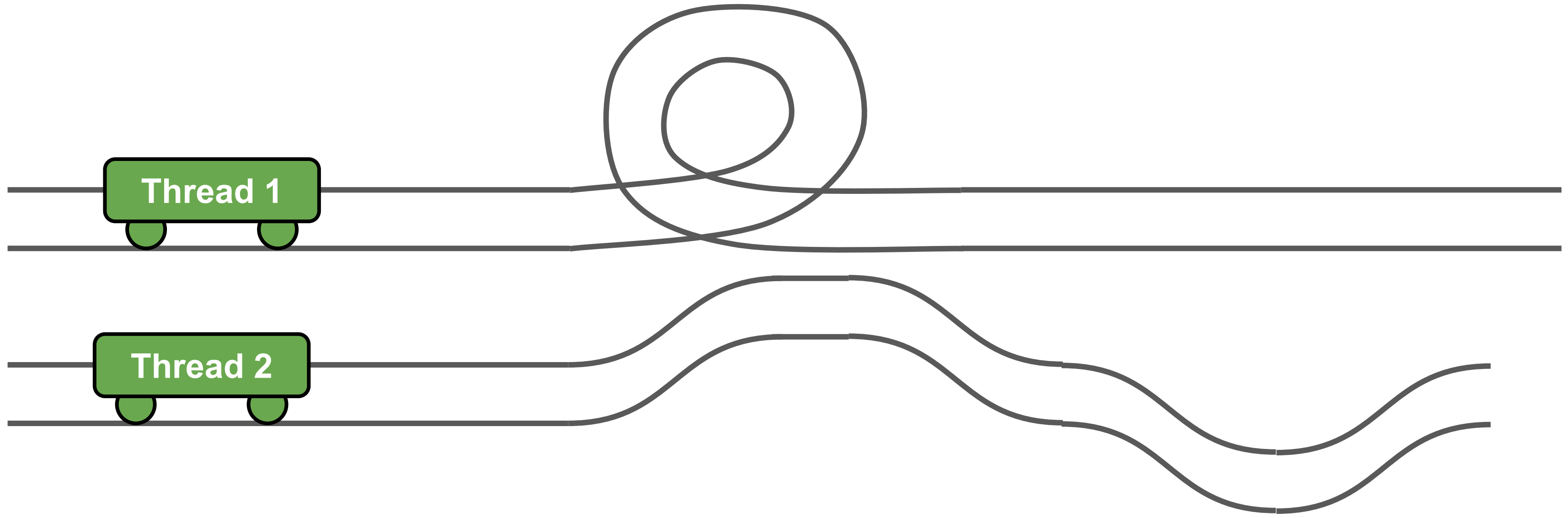
Lock-free algorithms:

- Can use retry mechanisms.
- Can get stuck indefinitely due to unfair scheduling.

Wait-free algorithms:

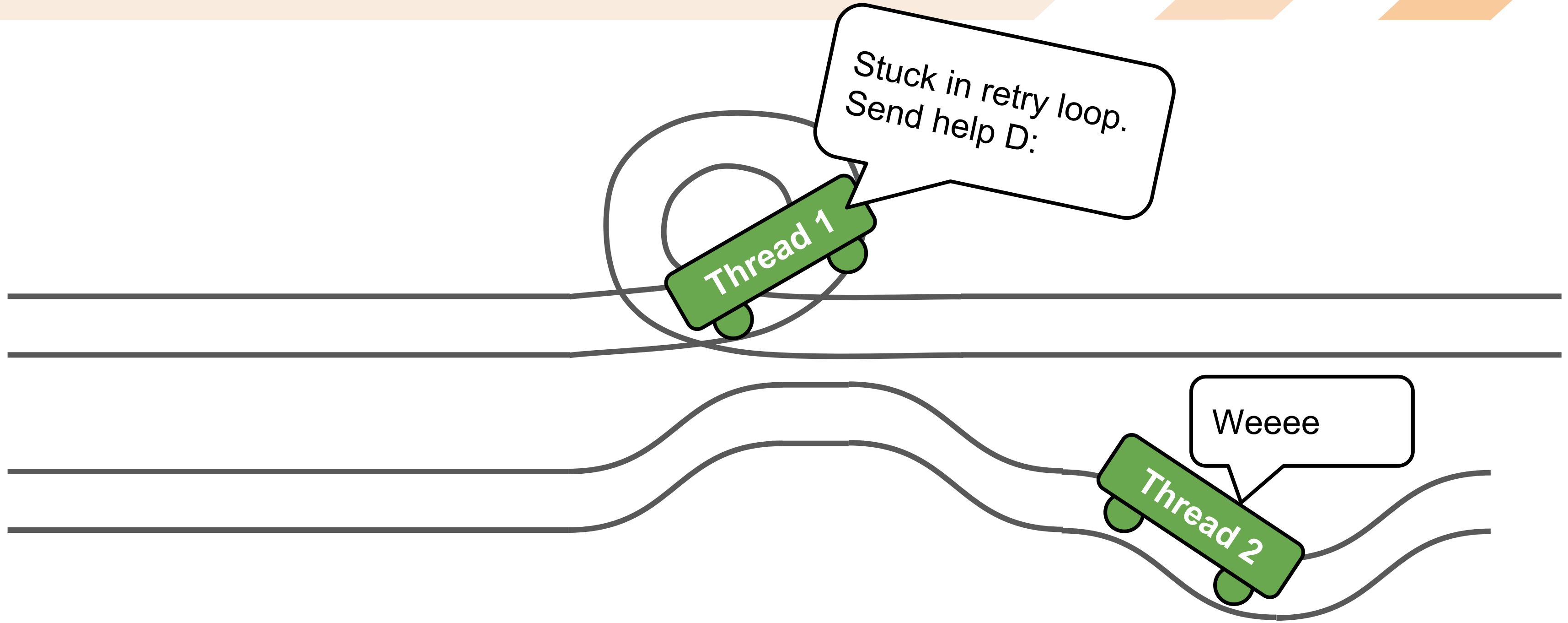
- Don’t use unlimited retries.

Lock-Freedom



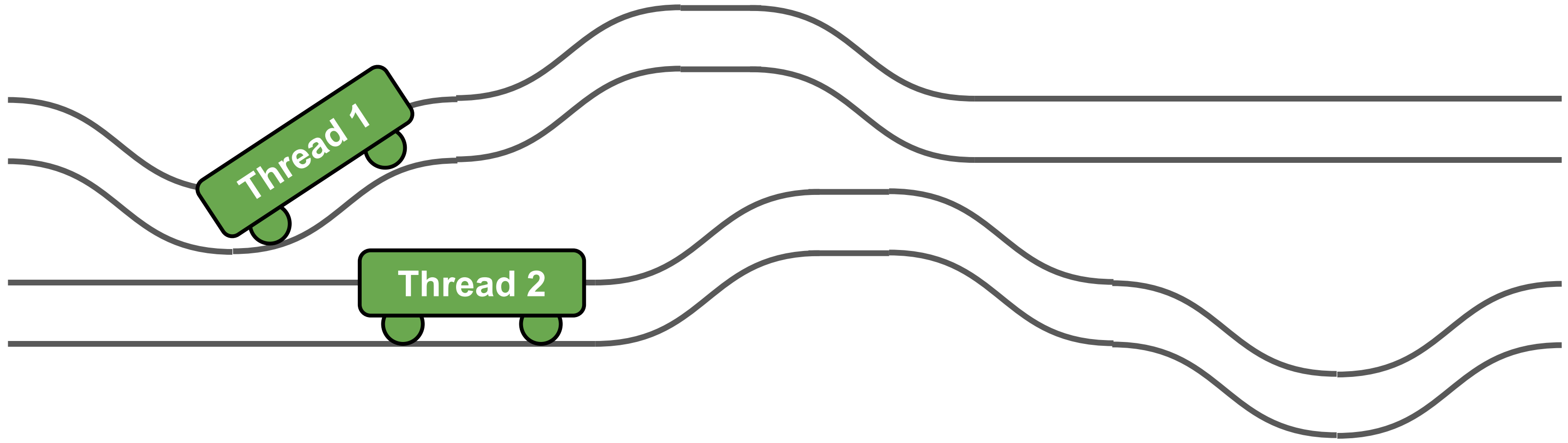
Lock-free: at least one thread guaranteed to complete.

Lock-Freedom



Lock-free: at least one thread guaranteed to complete.

Wait-Freedom



Wait-free: all threads guaranteed to complete!

Parallel Programming

How do we deal with parallelization in our code?

- High-level model of computation.
- High-level synchronization abstractions.

Reasoning about how hardware works is useful, but we use abstractions.
The exact details are too complicated, and change between hardware.

I will talk about the Java Memory Model.
All details do not apply to other languages!

Low- vs. High-Level

What you see is not what you get:

```
a = 3;
b = a * 2;
```

This can be optimized to

```
b = 6;
```

(This is a very obvious example.
Much less obvious reorderings are possible!)

Reordering

Instruction reordering is done by **compiler** and **processor**.

```
X = 3;
Y = 4;
```

Writes to memory can be reordered,
observed in the wrong order in different thread!

Reordering

Which states can T2 observe?

Thread 1:

```
X = 3;
Y = 4;
```

Thread 2:

```
print(Y);
print(X);
```

Reordering - Case 1

Which states can T2 observe?

Thread 1:

```
X = 3;    ✘
Y = 4;    ✘
```

Thread 2:

```
print(Y);
print(X);
```

Output:

```
0
0
```

Reordering - Case 2

Which states can T2 observe?

Thread 1:

```
X = 3;   ✓
Y = 4;   ✗
```

Thread 2:

```
print(Y);
print(X);
```

Output:

```
0
3
```

Reordering - Case 3

Which states can T2 observe?

Thread 1:

```
X = 3;    ✓
Y = 4;    ✓
```

Thread 2:

```
print(Y);
print(X);
```

Output:

```
4
3
```

Reordering - Case 4

Which states can T2 observe?

Thread 1:		**Thread 2:**
X = 3;	✗	print(Y);
Y = 4;	✓	print(X);

Output:

```
4
0
```

Reordering - Case 4

Which states can T2 observe?

Thread 1:

```
X = 3;    ✘
Y = 4;    ✔
```

Thread 2:

```
print(Y);
print(X);
```

Output:

```
4
0
```

Many other surprising cases here:
https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Reordering - Case 4

Which states can T2 observe?

Thread 1:

```
X = 3;    ✗
Y = 4;    ✓
```

Thread 2:

```
print(Y);
print(X);
```

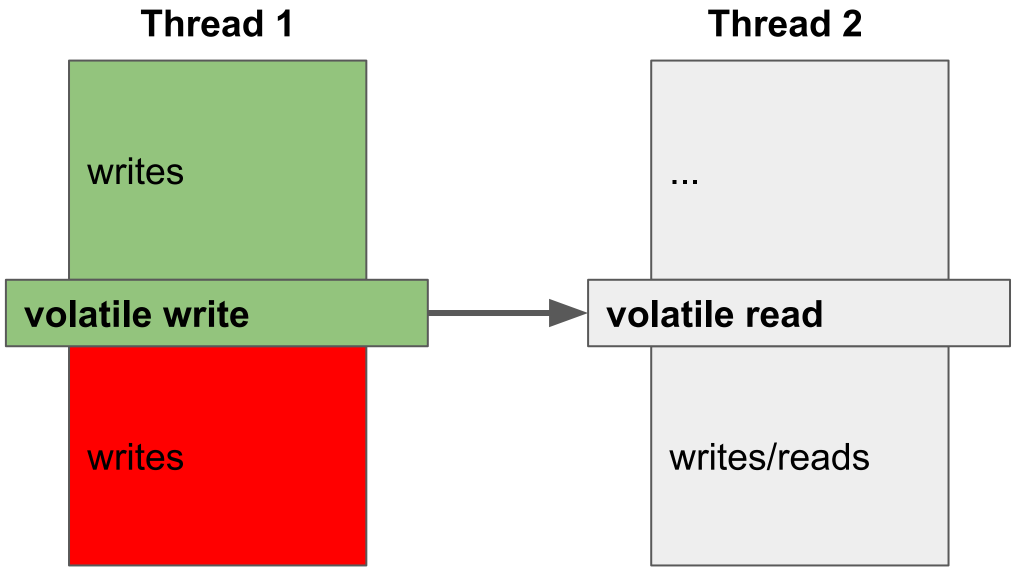
Output:

```
4
0
```

If **Y** is **volatile**,
this case is not possible!

Volatile

1. Reading a volatile observes some previous write.
2. All previous writes from the observed write are visible.

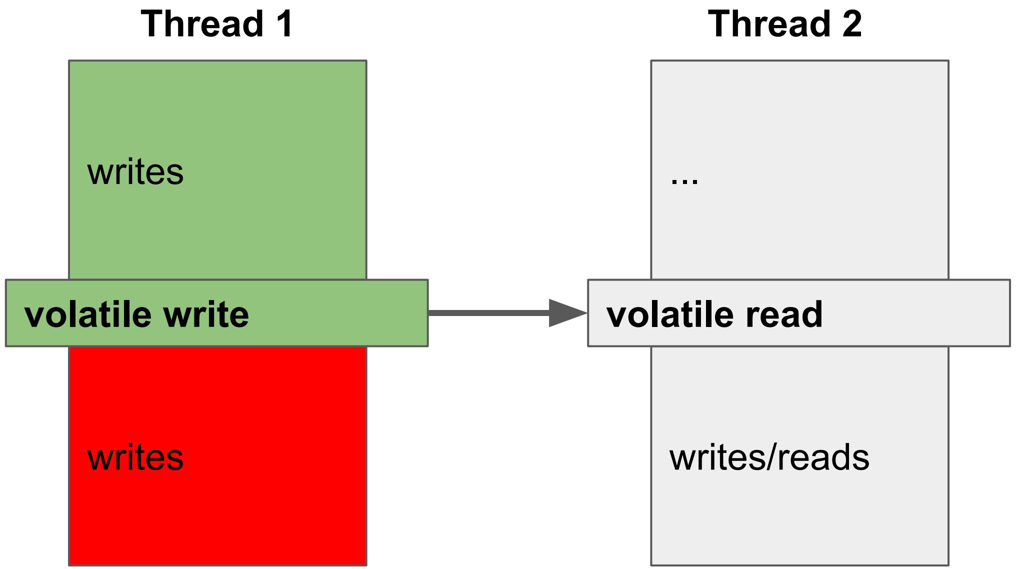


Volatile

We can think of volatile read as lock acquisition,

…

and volatile write as lock release.



Volatile

Some problems can be solved with **volatile** flags:

```
volatile bool done = false;
int value;

int compute() {
    if (done) {
        return value;
    }
    value = 123;
    done = true;
    return value;
}
```

Volatile

Some problems can be solved with **volatile** flags:

```
volatile bool done = false;
int value;

int compute() {
    if (done) {
        return value;
    }
    value = 123;  // Multiple threads can enter here!
    done = true;
    return value;
}
```

Volatile

Volatile is a useful building block for ensuring safe publication!

Volatile is not strong enough to solve test-and-act problems that need mutual exclusion.

Other synchronization primitives like Compare-And-Set are used instead.

Fixpoint Functions

A fixpoint function **f(x)** is a function that has a fixed point **x**:

$$x = f(x)$$

Typically f(x) is defined in terms of f(x). It is **recursive**. Can be mutually recursive with other function(s).

Fixpoint Functions

A fixpoint function **f(x)** is a function that has a fixed point **x**:

$$x = f(x)$$

How can we find the fixed point **x**?

1. Start with some value.
2. Repeatedly apply **f** until the value is fixed.

Successive Approximation

```
int x = 0, old;
while (true) {
        old = x;
        x = f(x);
        if (x == old) break;
}
```

Initial value

Recompute

Change test

Fixpoint Functions

Fixpoint functions can be computed using

- **Change flag**: run the loop as long as a change is detected.
- **Worklist**: add nodes to worklist to propagate changes.

Liveness

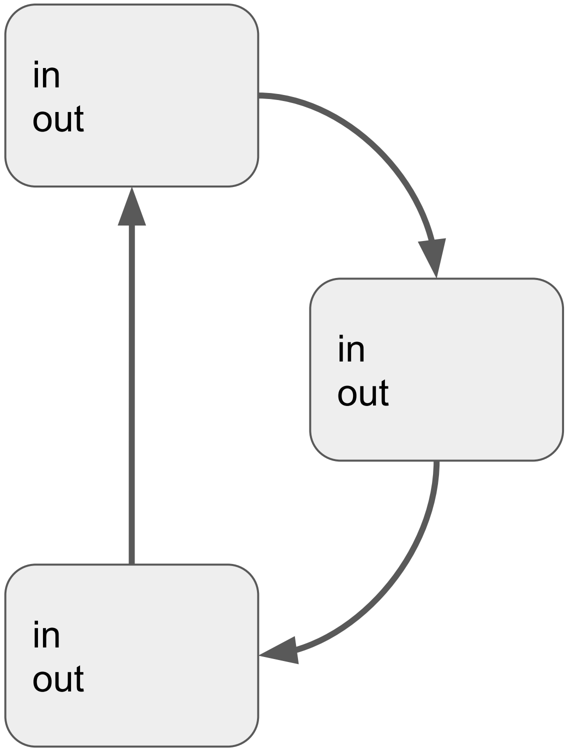
Computing liveness is a fixpoint problem.

$$in = (out - def) \cup use$$

$$out = \bigcup_{s = succ} s.in$$

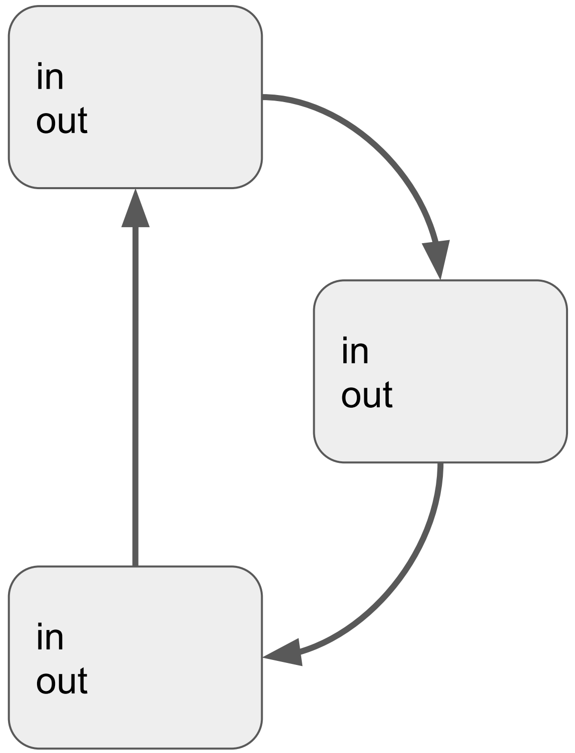
These equations are circularly defined!

Lock-based implementation could deadlock!



Liveness

The liveness algorithm used in this course is worklist based.



Sequential Liveness

Main loop:

```
while (!worklist.isEmpty()) {
        Vertex v = worklist.pop();
        v.compute(worklist);
}
```

```
class Vertex {
  BitSet in, out, use, def;

  void compute(List<Vertex> worklist) {}
}
```

Sequential Liveness

```
for (Vertex v : succ)
    out.or(v.in);
old = in;
in = new BitSet();
in.or(out);
in.andNot(def);
in.or(use);
if (!in.equals(old)) {
    for (Vertex v : pred)
        worklist.addLast(v);
}
```

Sequential Liveness

```
for (Vertex v : succ)
    out.or(v.in);
old = in;
in = new BitSet();
in.or(out);
in.andNot(def);
in.or(use);
if (!in.equals(old)) {
    for (Vertex v : pred)
        worklist.addLast(v);
}
```

Compute out set

Compute in set

Change test

Propagate change

Parallel Liveness

Parallelizing this sequential algorithm is not easy.

Most of your lab solutions are probably faulty - they worked by chance.

Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```

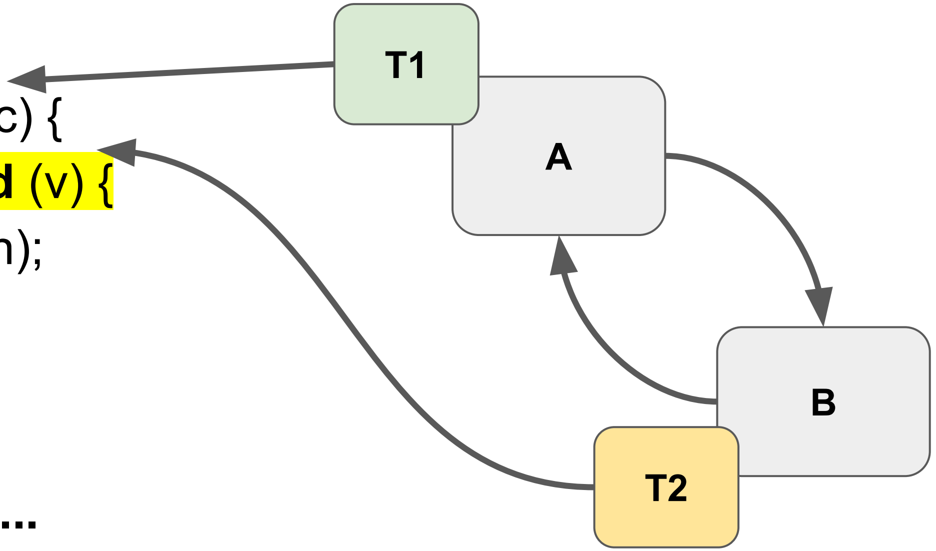
Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```

Warning: circular dependency!

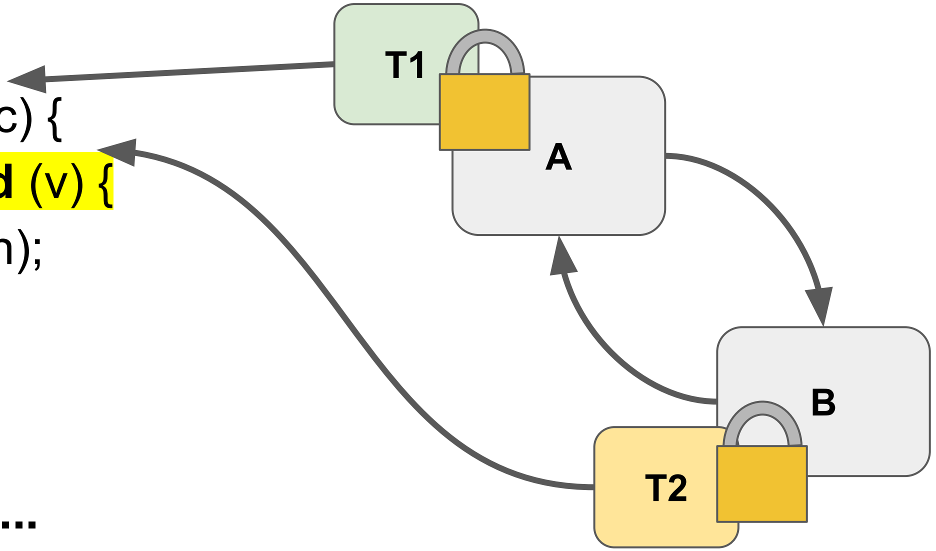
Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```



Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```



Deadlock!

T1 locked A.
T2 locked B and tries to lock A.

Lock-based Liveness (2)

```
listed = false;
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```

Lock-based Liveness (2)

T1

```
listed = false;
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
...
```

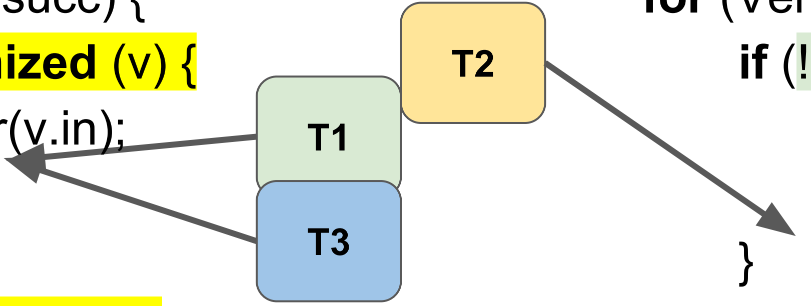
T2

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```

Lock-based Liveness (2)

```
listed = false;
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```



Data race!

T1 and T3 try to modify the same out set. Bad things happen.

Lock-based Liveness (3)

```
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
listed = false;
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```

Lock-based Liveness (3)

```
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
listed = false;
...
```

T1

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```

T2

Lock-based Liveness (3)

```
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
listed = false;
...
```

T1

listed is
true

T2

```
                              ls(old)) {
                              (Vertex v : pred) {
                                if (!pred.listed) {
                                    worklist.add(v);
                                    pred.listed = true;
                                }
                            }
                        }
```

T1 updated the vertex based on old state.

Missed update!

How to catch concurrency errors?

It can work fine on your machine even if you have a broken algorithm.

Running multiple times helps, but guarantees nothing!

Valgrind gives no guarantees.

It does not help to run more threads!

More threads can reduce the failure rate.

How to catch concurrency errors?

Think about your invariants and try to disprove them!

This leads to insight into possible flaws.

Invariants

Invariants for the liveness algorithm:

- A node must be updated it is in, or will be added to, the worklist.
- Only one thread works on a single node at a time.
- A thread should not hold multiple locks at the same time.

Try to disprove these to gain insight!

Try to remove invariants if possible.

Can you redesign the algorithm to remove an invariant,
or make it easier to ensure that it holds?

Compare And Set

Atomically test-and-set a variable:

```
AtomicReference r = new AtomicReference(null);
r.compareAndSet(null, foo);
```

Only succeeds if the value of **r** is **null**.

Atomically updates **r** to the **foo** reference.

Lock-Free Liveness

Main idea:

Take a snapshot of the state of the current vertex.

Only commit state update if no other thread changed the state.

Lock-Free Liveness

```
AtomicReference<BitSet> in;
while (true) {
        BitSet oldIn = in.get();
        ...
        if (!out.compareAndSet(oldOut, newOut)) {
                // Someone else changed the out set.
                continue;
        }
        ...
}
```

Take snapshot

Compute

Try update

Restart

Lock-Free Liveness

Rinse and repeat for the in set!

Attribute Grammars

Attributes are an abstraction used in compilers.

Compilation is split up into **attributes** - essentially, small functions.

Attributes are declarative and pure (no side effects).

Attribute Grammars

Attributes are automatically scheduled by an attribute evaluator.

Attribute evaluation can be freely reordered.

Attribute Grammars

JastAdd is a metacompilation tool:

A tool for building compilers.

I recently implemented concurrent attribute evaluation in JastAdd.

The project was funded by a 2015 Google Faculty Research Award.

Concurrent attributes have been used to parallelize a Java compiler, with a resulting 2x speedup!

Attribute Liveness

Circular attributes are used to compute fixpoint functions!

For example, liveness can be computed using attributes!

It’s the most compact way, counting code size.

Not the most efficient, but a nice demonstration of the abstraction power of attributes!

Attribute Liveness

```
syn BitSet Vertex.out() {
  BitSet out = new BitSet();
  for (Vertex s : getSucc())
    out.or(s.in());
  return out;
}
```

```
syn BitSet Vertex.in() circular [new BitSet()] {
  BitSet in = new BitSet();
  in.or(out());
  in.andNot(getDef());
  in.or(getUse());
  return in;
}
```

Attribute Liveness

```
syn BitSet Vertex.out() {
  BitSet out = new BitSet();
  for (Vertex s : getSucc())
    out.or(s.in());
  return out;
}
```

Fixpoint function.

```
syn BitSet Vertex.in() circular [new BitSet()] {
  BitSet in = new BitSet();
  in.or(out());
  in.andNot(getDef());
  in.or(getUse());
  return in;
}
```

Initial value.

Attribute Liveness

Compute liveness for all vertices:

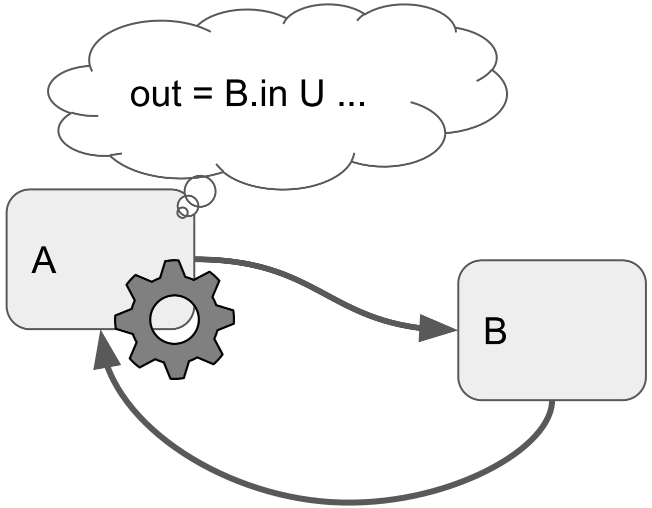
for (Vertex v : vertices) v.in();

This computes all in/out sets because **in** is recursively defined, and uses **out**.

Attribute Evaluation

How does it work?

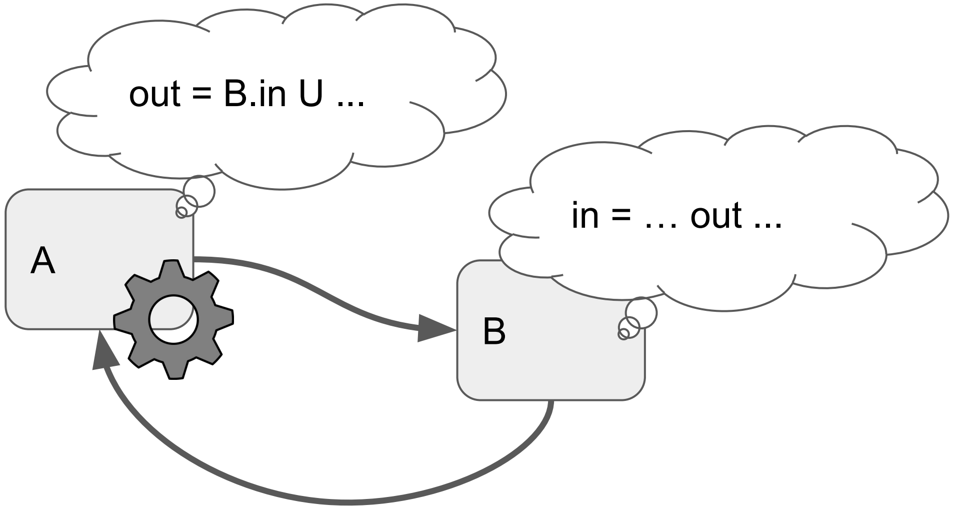
Recursive dynamic computation of the attribute function:



Attribute Evaluation

How does it work?

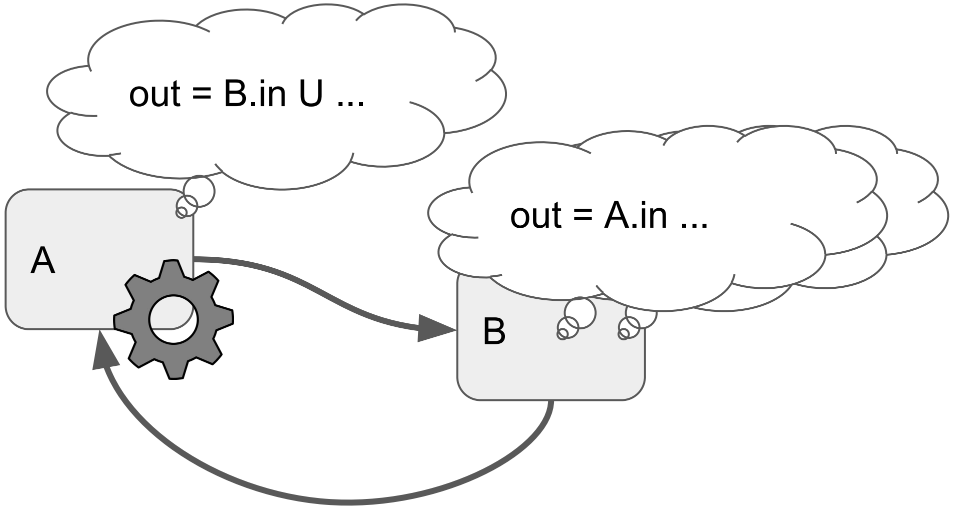
Recursive dynamic computation of the attribute function:



Attribute Evaluation

How does it work?

Recursive dynamic computation of the attribute function:



Circular Attribute Evaluator

Global state:

CYCLE	- Keeps track of the current iteration in the fixpoint loop.
CHANGE	- Global change flag. Fixpoint iteration is done when it remains false.

Each vertex:

in_value	- Current approximation.
done	- Flag indicating if the attribute is fully computed.
visited	- Tracks the last iteration the attribute value was computed in. Used as a visit flag.

Circular Attribute Evaluator

```
BitSet Vertex.in() {
    if (done) return in_value;          // Cache check.

    if (CYCLE == 0) {
        // Case 1: Start fixpoint evaluation!
    } else if (visited != CYCLE) {
        // Case 2: Compute value.
    } else {
        // Case 3: Reuse current value.
    }
    return in_value;
}
```

Circular Attribute Evaluator

```
// Case 1: Start fixpoint evaluation!
do {
    CYCLE += 1;
    CHANGE = false;
    visited = CYCLE;
    next = in_compute();
    if (!next.equals(in_value)) {
        in_value = next;
        CHANGE = true;
    }
} while (CHANGE);
done = true;
```

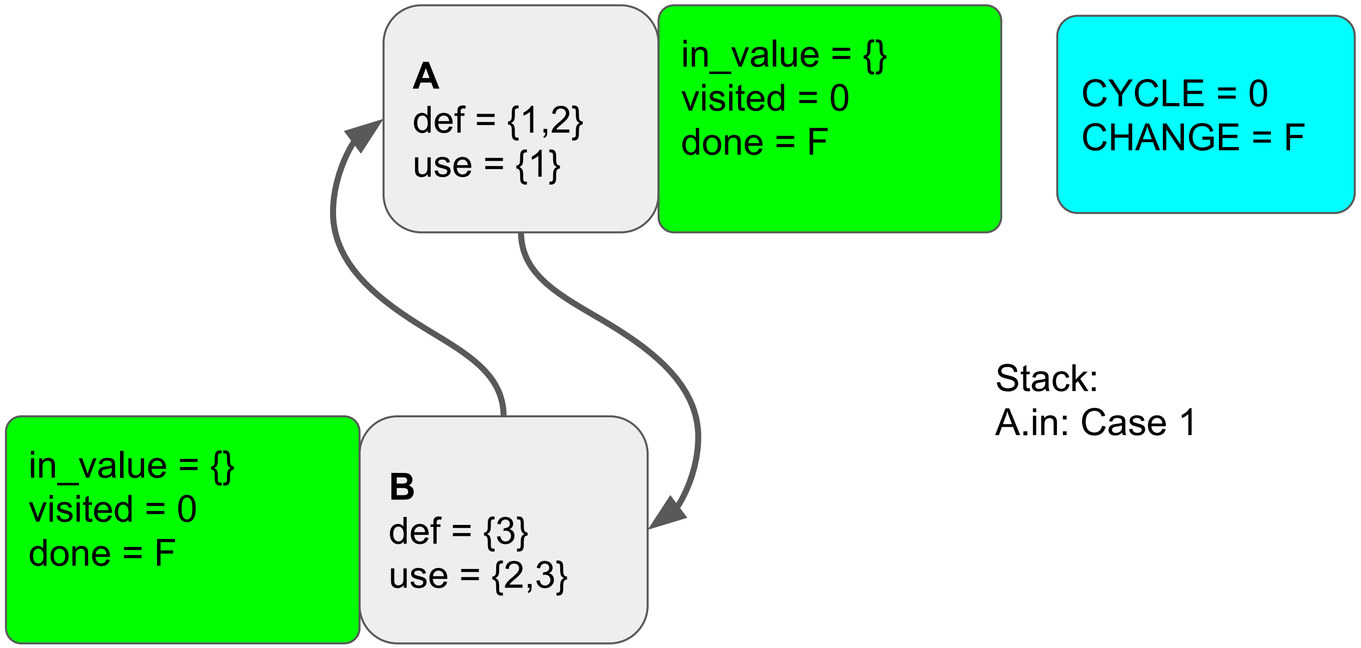
Circular Attribute Evaluator

```
// Case 2: Compute value.
visited = CYCLE;
next = in_compute();
if (!next.equals(in_value)) {
        in_value = next;
        CHANGE = true;
}
```

Circular Attribute Evaluator

```
// Case 3: Reuse current value.
return in_value;
```

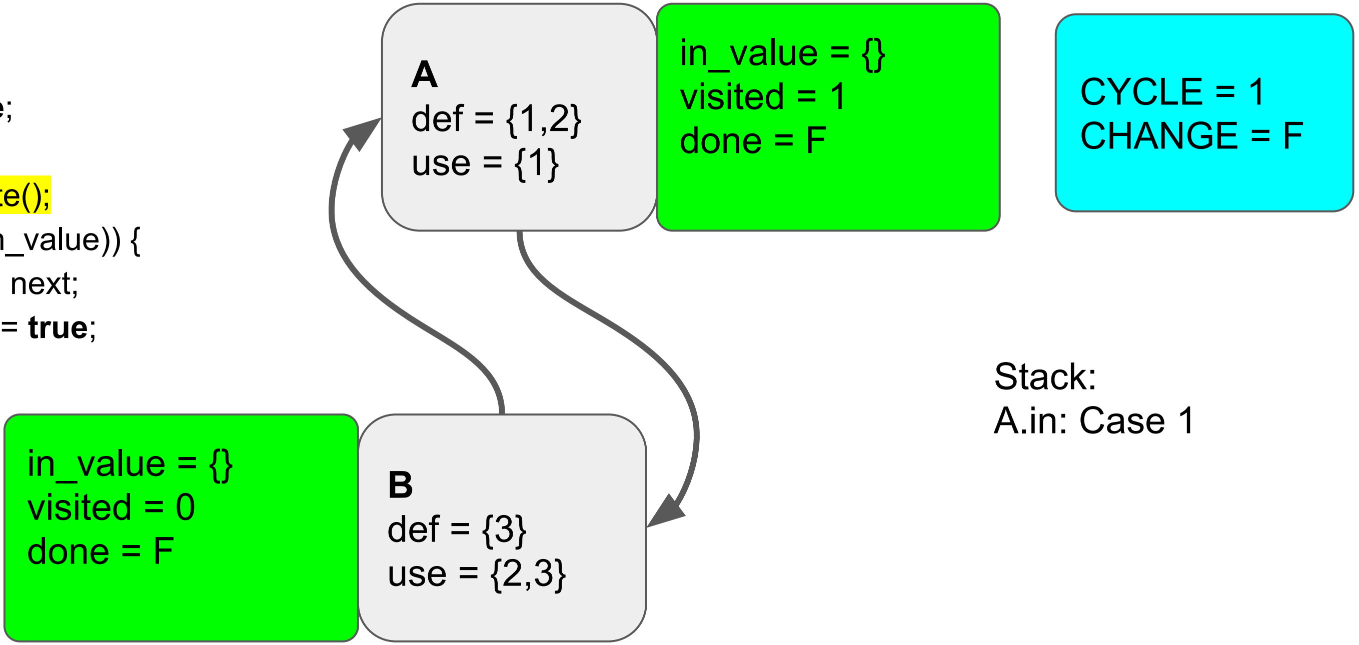
Attribute Evaluation



Stack:
A.in: Case 1

Attribute Evaluation

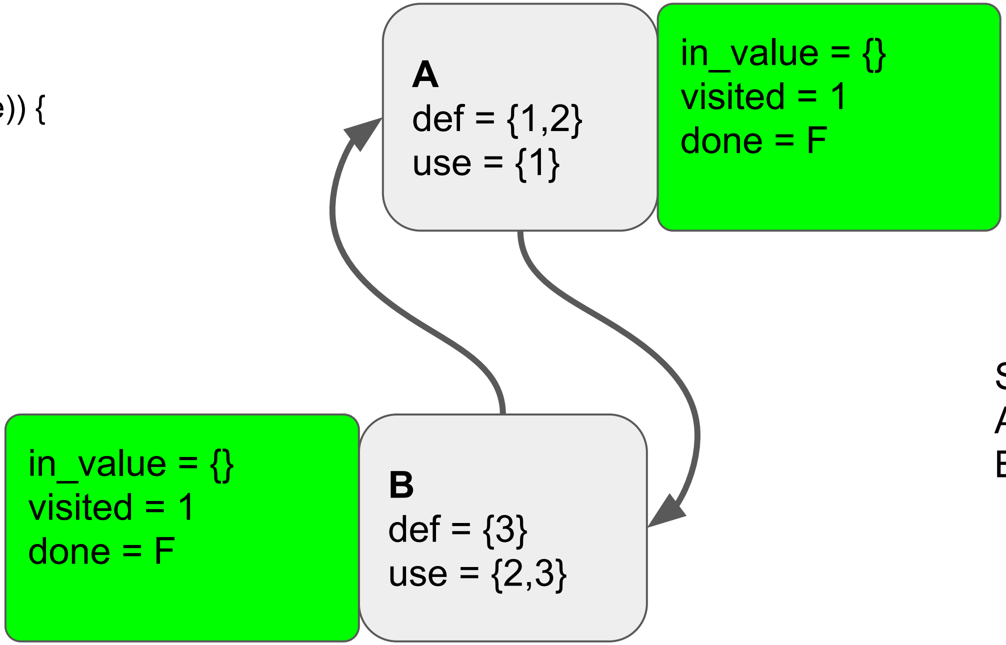
```
// Case 1: Start fixpoint evaluation!
do {
        CYCLE += 1;
        CHANGE = false;
        visited = CYCLE;
        next = in_compute();
        if (!next.equals(in_value)) {
                in_value = next;
                CHANGE = true;
        }
} while (CHANGE);
done = true;
```



Stack:
A.in: Case 1

Attribute Evaluation

```
// Case 2: Compute value.
visited = CYCLE;
next = in_compute();
if (!next.equals(in_value)) {
        in_value = next;
        CHANGE = true;
}
```

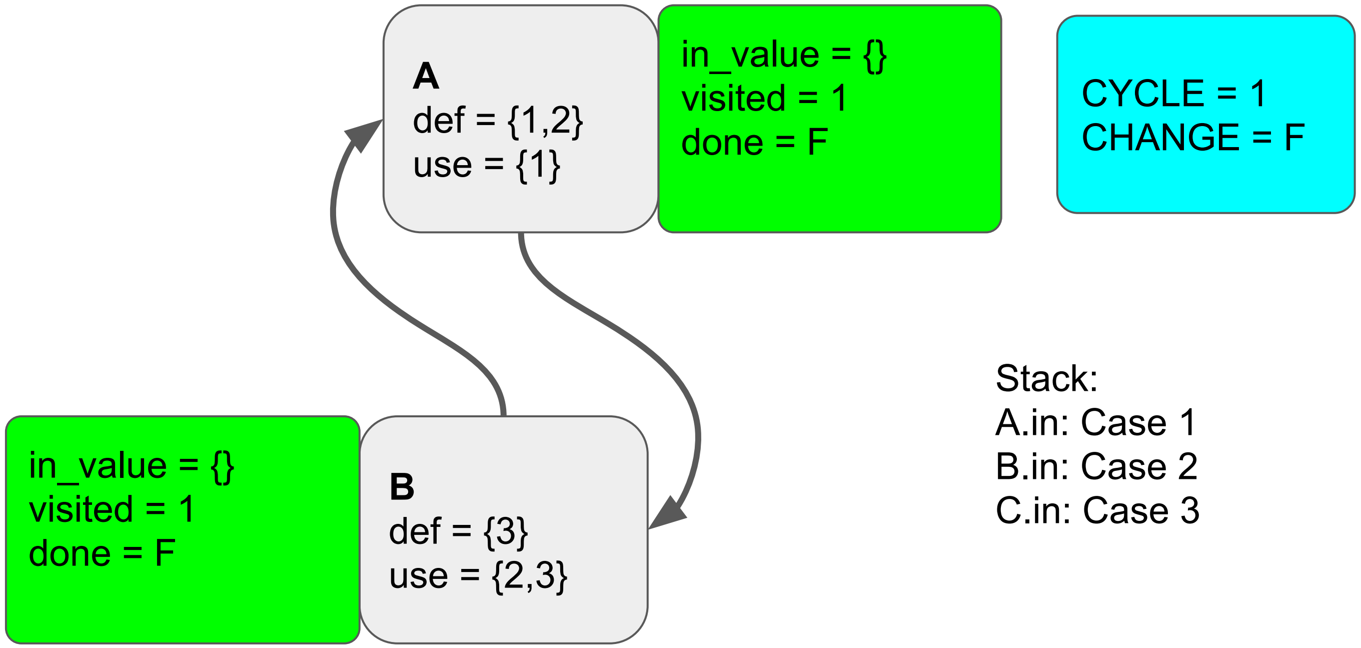


CYCLE = 1
CHANGE = F

Stack:
A.in: Case 1
B.in: Case 2

Attribute Evaluation

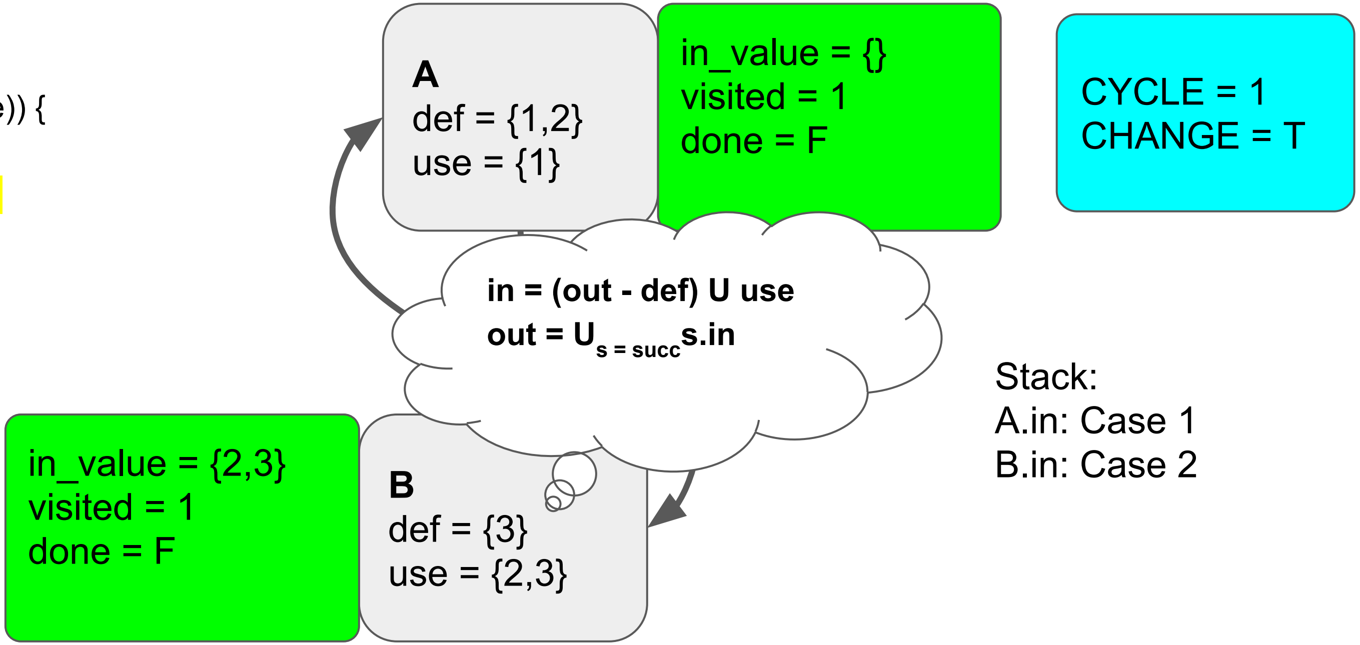
```
// Case 3: Reuse current value.
return in_value;
```



Stack:
A.in: Case 1
B.in: Case 2
C.in: Case 3

Attribute Evaluation

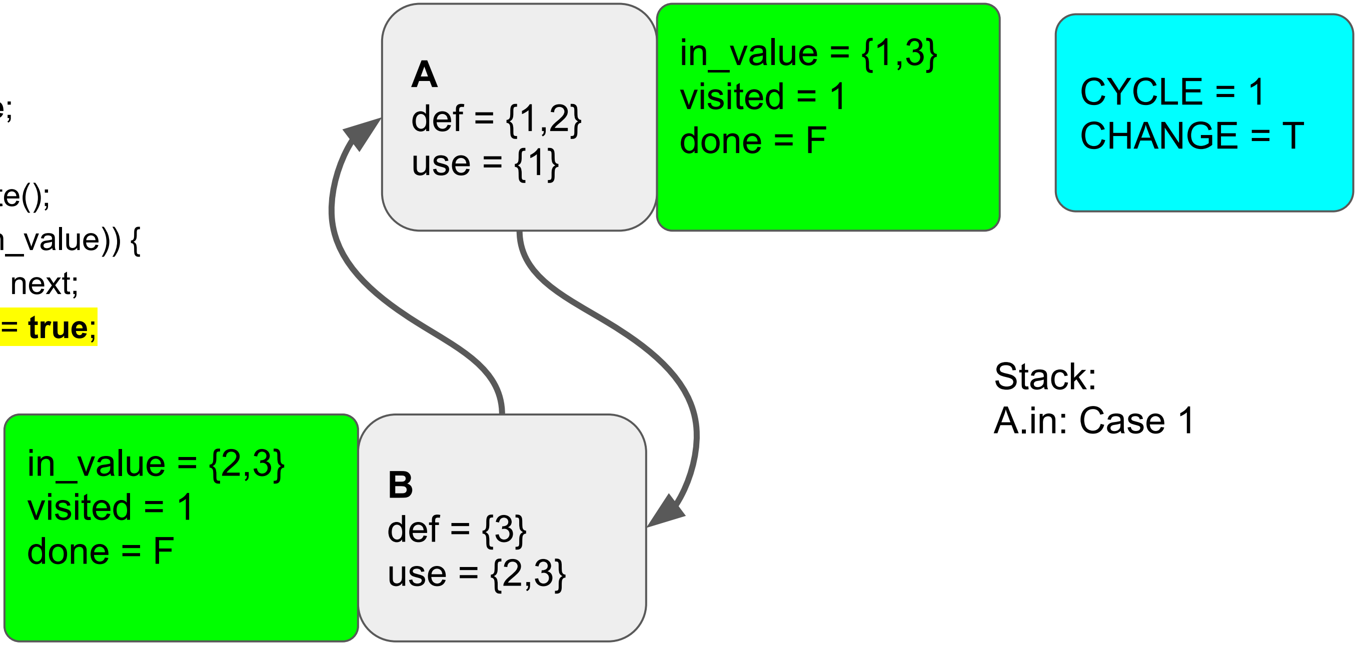
```
// Case 2: Compute value.
visited = CYCLE;
next = in_compute();
if (!next.equals(in_value)) {
        in_value = next;
        CHANGE = true;
}
```



Stack:
A.in: Case 1
B.in: Case 2

Attribute Evaluation

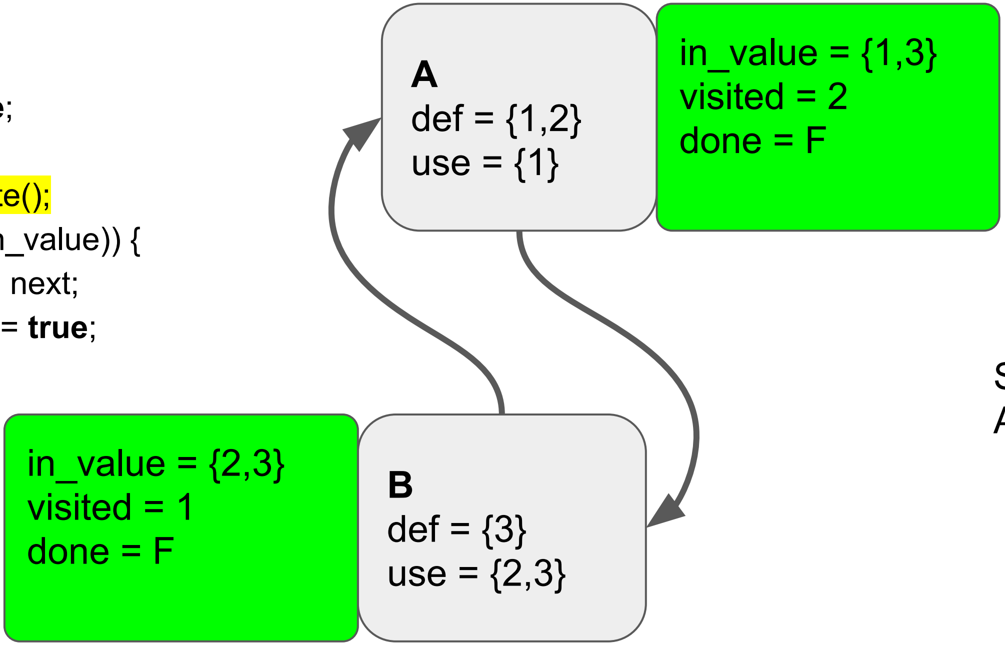
```
// Case 1: Start fixpoint evaluation!
do {
        CYCLE += 1;
        CHANGE = false;
        visited = CYCLE;
        next = in_compute();
        if (!next.equals(in_value)) {
                in_value = next;
                CHANGE = true;
        }
} while (CHANGE);
done = true;
```



Stack:
A.in: Case 1

Attribute Evaluation

```
// Case 1: Start fixpoint evaluation!
do {
        CYCLE += 1;
        CHANGE = false;
        visited = CYCLE;
        next = in_compute();
        if (!next.equals(in_value)) {
                in_value = next;
                CHANGE = true;
        }
} while (CHANGE);
done = true;
```

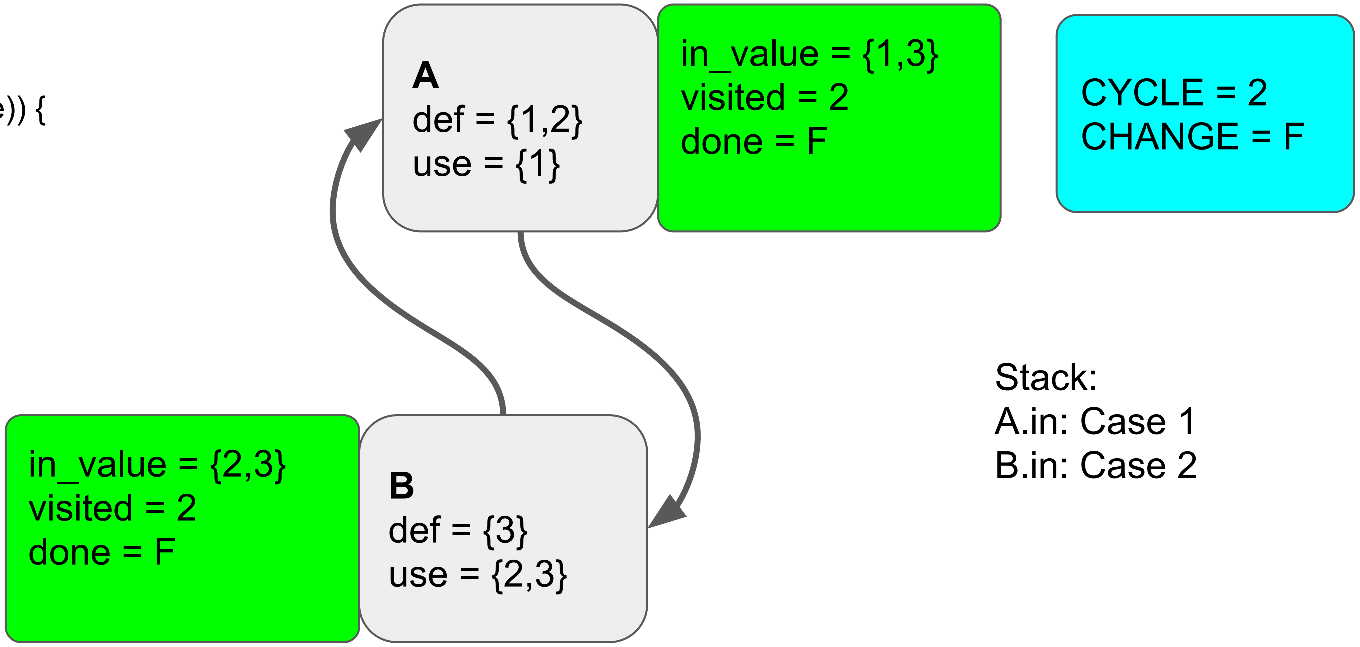


CYCLE = 2
CHANGE = F

Stack:
A.in: Case 1

Attribute Evaluation

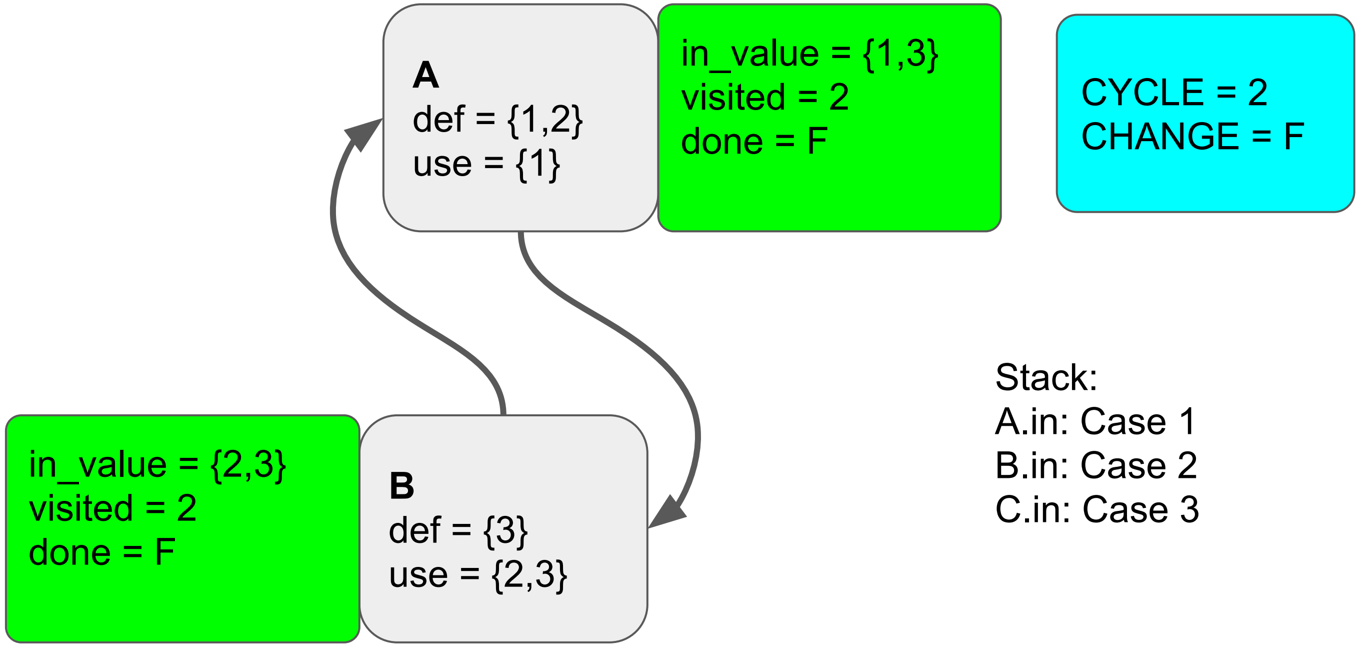
```
// Case 2: Compute value.
visited = CYCLE;
next = in_compute();
if (!next.equals(in_value)) {
        in_value = next;
        CHANGE = true;
}
```



Stack:
A.in: Case 1
B.in: Case 2

Attribute Evaluation

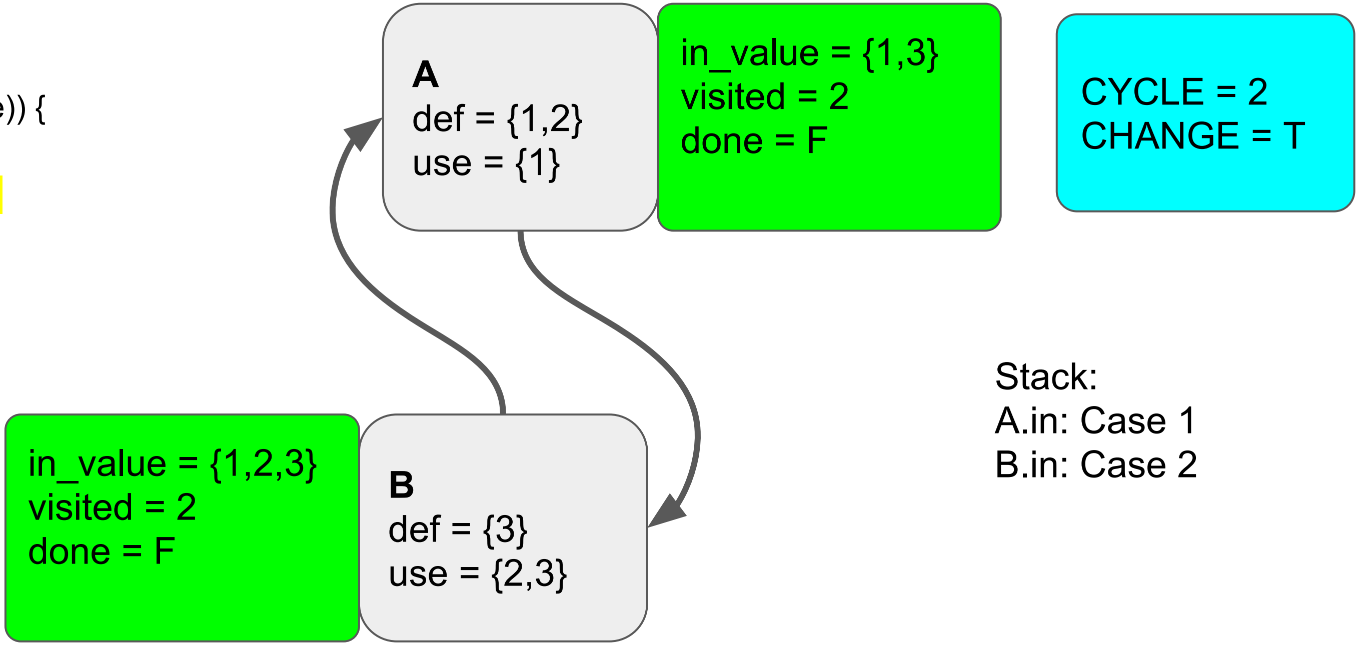
```
// Case 3: Reuse current value.
return in_value;
```



Stack:
A.in: Case 1
B.in: Case 2
C.in: Case 3

Attribute Evaluation

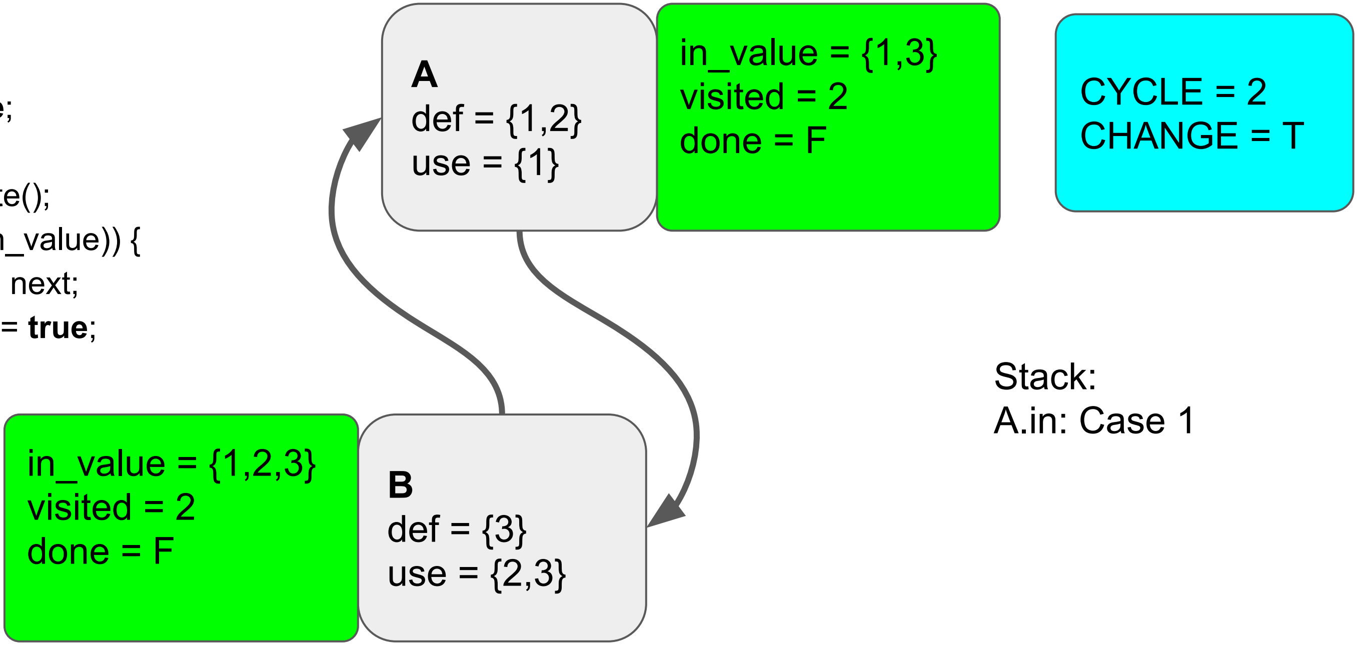
```
// Case 2: Compute value.
visited = CYCLE;
next = in_compute();
if (!next.equals(in_value)) {
        in_value = next;
        CHANGE = true;
}
```



Stack:
A.in: Case 1
B.in: Case 2

Attribute Evaluation

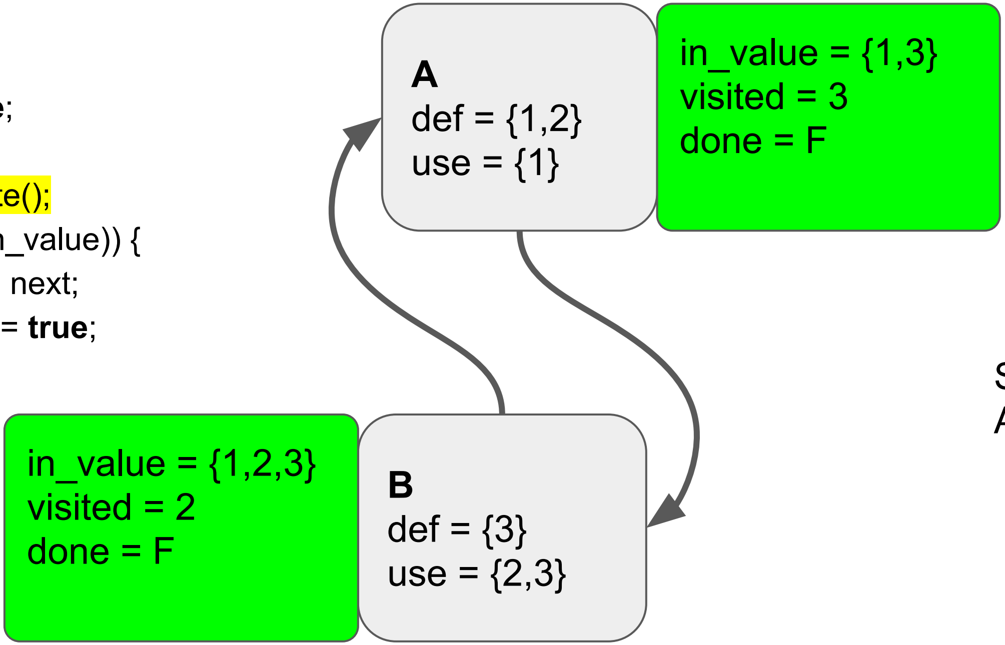
```
// Case 1: Start fixpoint evaluation!
do {
        CYCLE += 1;
        CHANGE = false;
        visited = CYCLE;
        next = in_compute();
        if (!next.equals(in_value)) {
                in_value = next;
                CHANGE = true;
        }
} while (CHANGE);
done = true;
```



Stack:
A.in: Case 1

Attribute Evaluation

```
// Case 1: Start fixpoint evaluation!
do {
        CYCLE += 1;
        CHANGE = false;
        visited = CYCLE;
        next = in_compute();
        if (!next.equals(in_value)) {
                in_value = next;
                CHANGE = true;
        }
} while (CHANGE);
done = true;
```

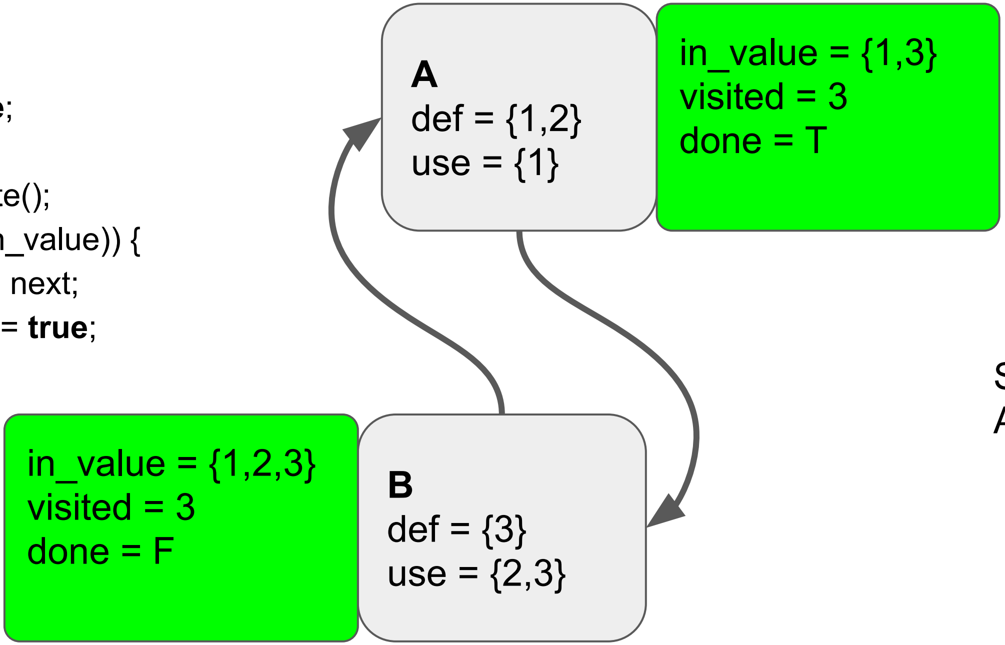


CYCLE = 3
CHANGE = F

Stack:
A.in: Case 1

Attribute Evaluation

```
// Case 1: Start fixpoint evaluation!
do {
        CYCLE += 1;
        CHANGE = false;
        visited = CYCLE;
        next = in_compute();
        if (!next.equals(in_value)) {
                in_value = next;
                CHANGE = true;
        }
} while (CHANGE);
done = true;
```



CYCLE = 3
CHANGE = F

Stack:
A.in: Case 1

Concurrent Evaluation

Challenges in making the circular attribute evaluator concurrent:

Attribute approximations need to be safely shared between threads.

Can’t use locks - would lead to deadlock in in circular eval!

Concurrent Evaluation

Changes in the concurrent algorithm:

Each thread has their own thread-local CHANGE and CYCLE states.

The per-attribute visit information is stored in a thread-local map.

Attribute value and done flags are combined into a tuple and stored using AtomicReference.

Thread-Local State

In Java, you can use ThreadLocal to store persistent thread-local data.

Thread-local state of the concurrent evaluator:

- **tls.iter** - maps attributes to iteration indices
- **tls.cycle** - current iteration index
- **tls.change** - change flag

Global State

Each vertex has the following global state:

```
CircularAttributeValue in_value;

class CircularAttributeValue {
    volatile bool done = false;
    AtomicReference value = new AtomicReference(NIL);
}
```

NIL is an invalid attribute value.

Concurrent Cache Check

First test if it was already computed:

```
if (in_value.done) {
    // done is volatile, ensuring safe publication:
    return in_value.get();
}
```

If not, compute the attribute. Does not matter if another thread just finished computing the attribute, we'll just get the same value!

Concurrent Compute

To compute the attribute value, each thread first reads the current value:

```
prev = in_value.value.get();

next = in_compute();

if (!next.equals(prev)) {
      in_value.value.compareAndSet(prev, next);
      tls.change = true;
}
```

Take snapshot

Compute

Change test

Try update

Lock-Freedom

This concurrent algorithm is lock-free and wait-free!

If a thread fails to update an attribute value, it will just the other thread’s result going forward. (No unlimited retries)

The concurrent algorithm works for any fixpoint function, not just this liveness example.

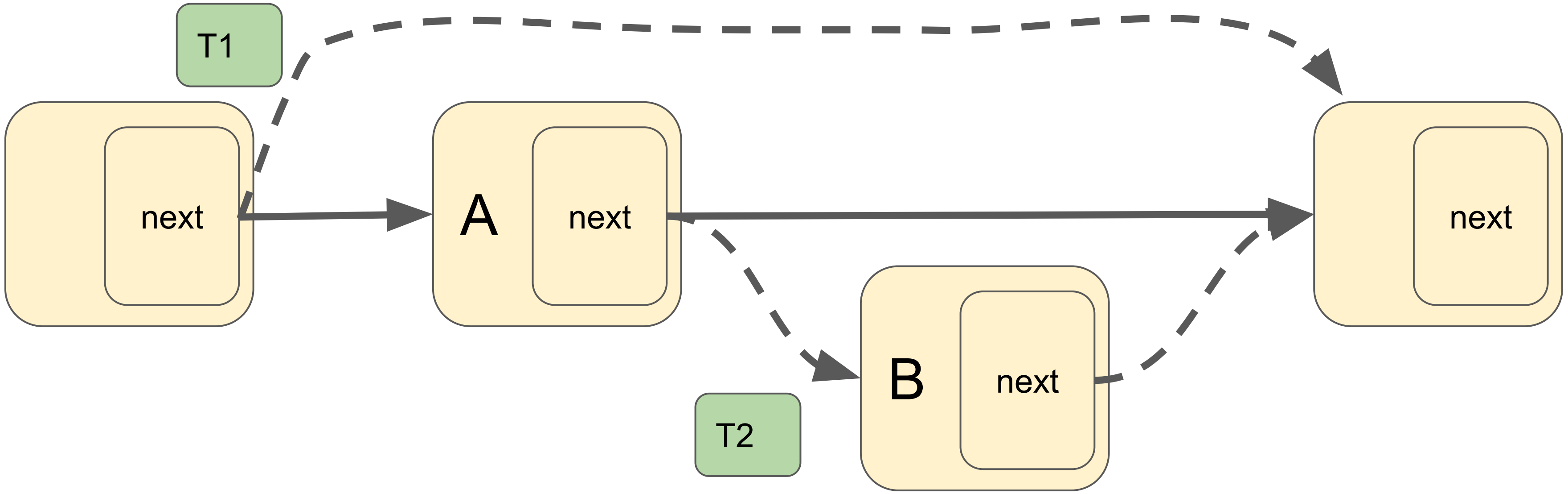
Other Topics

Other topics to discuss if there is time:

- Lock-free linked lists
- Skiplists
- Universal wait-free construction

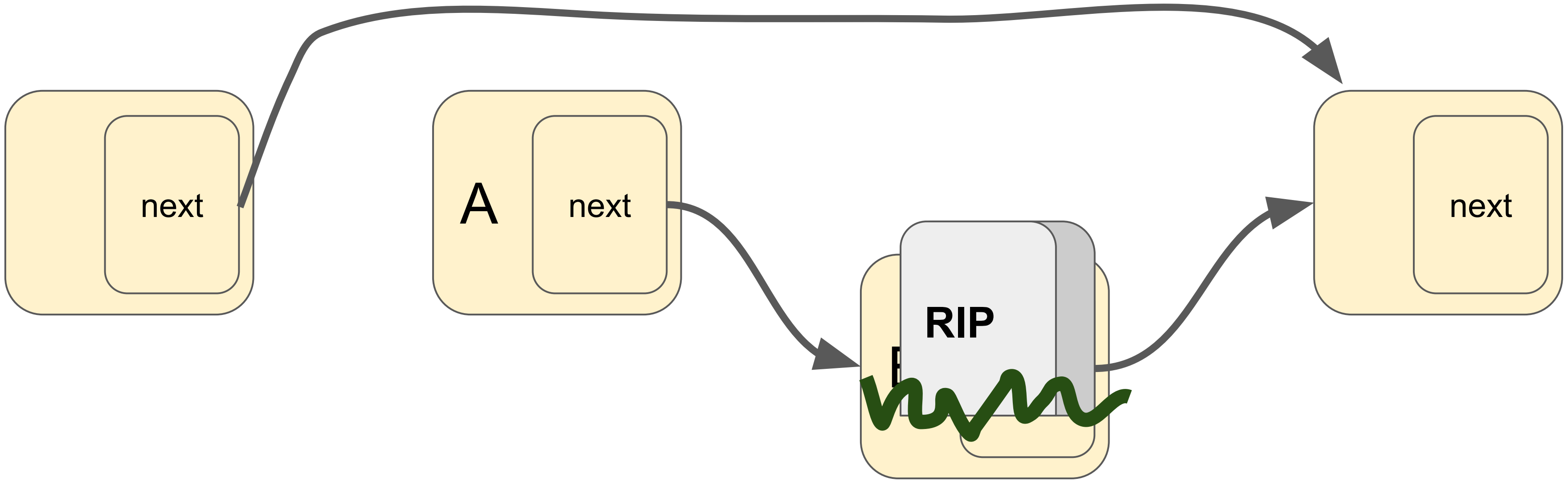
Concurrent Linked List

Thread 1 deletes node A, thread 2 deletes node B:



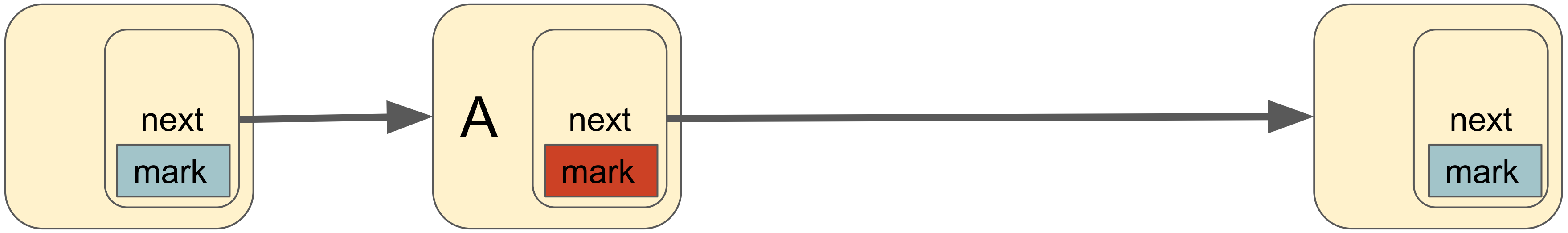
Concurrent Linked List

Result: node B was lost!



Lock-Free Linked List

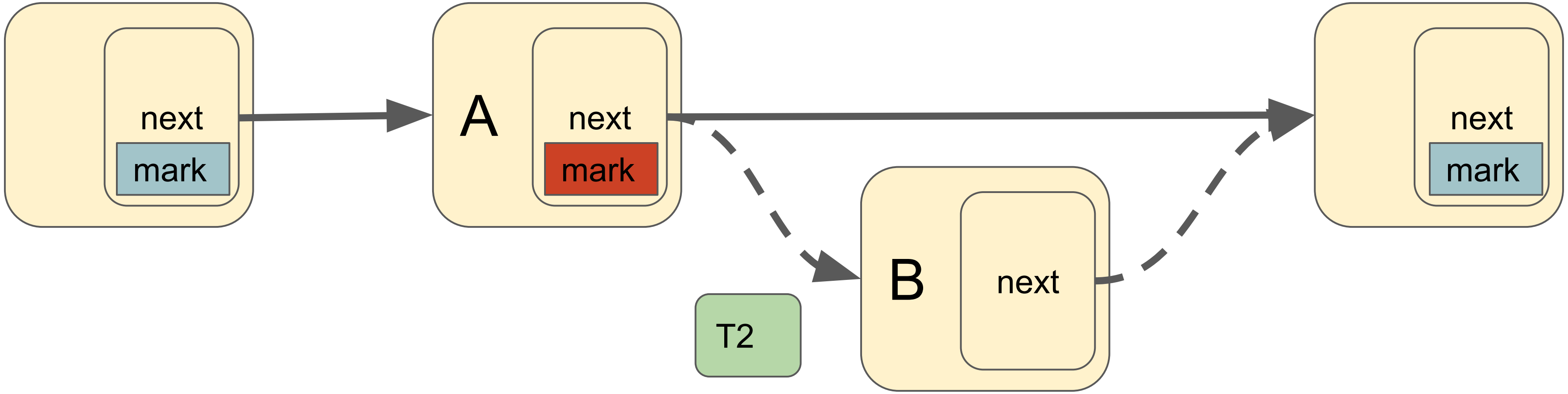
Solution: add a mark field to logically delete nodes.



Marked nodes are treated as-if deleted from the list.

Lock-Free Linked List

Solution: add a mark field to logically delete nodes.

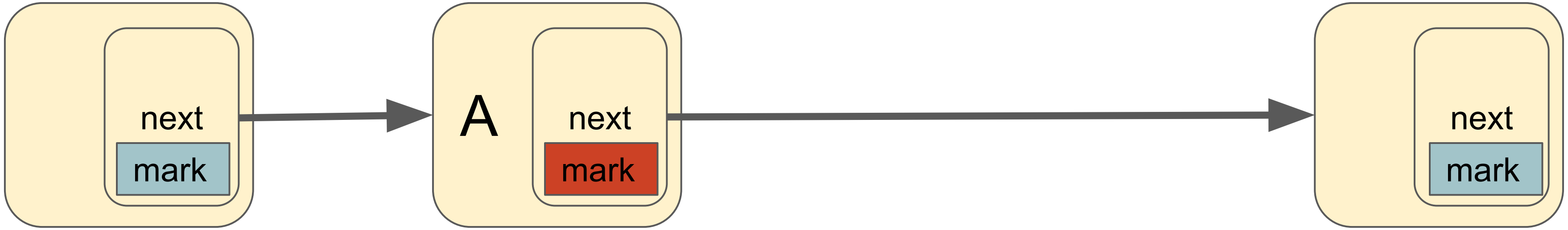


T2

Can’t insert here because
predecessor was marked!

Lock-Free Linked List

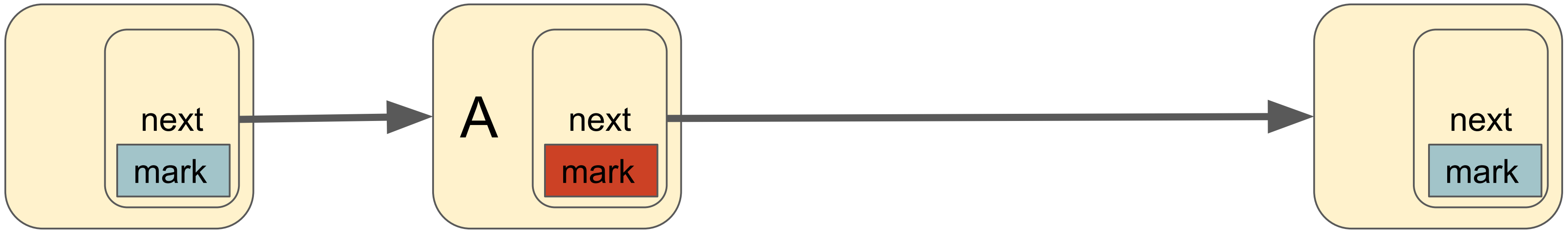
Physical deletion is done separately.



Need to update **next** and **mark** atomically...

Lock-Free Linked List

Physical deletion is done separately.



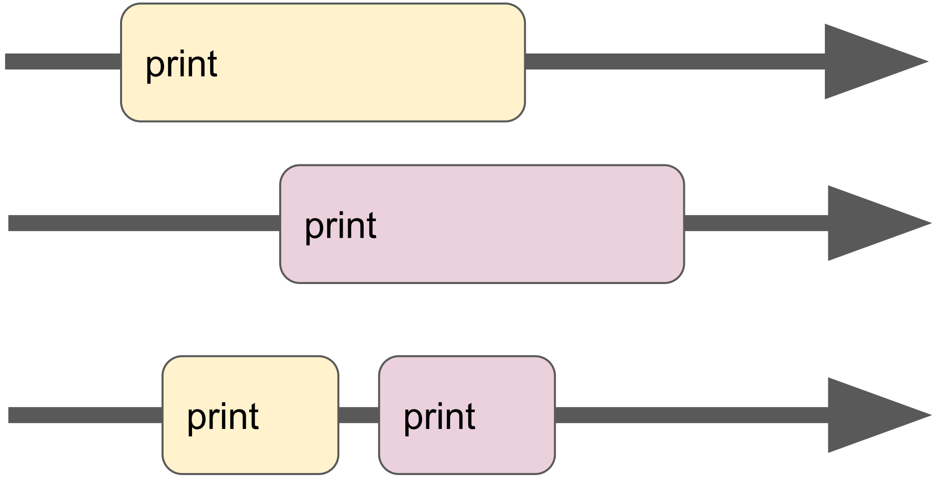
Need to update **next** and **mark** atomically…

Use tuple or **AtomicMarkedReference**

Then we can use CAS to atomically update the next pointer and the mark.

Linearizability

Overlapping linearizable functions calls work as if they took effect in a sequential order:



Linearization:

Thank you for your attention!