



Automatic Memory Management for Java

David F. Bacon

IBM T.J. Watson Research Center

Three Synergistic Ideas

- ◆ Decoupled Object Model
- ◆ Memory Sandboxes
- ◆ Concurrent GC by Reference Counting

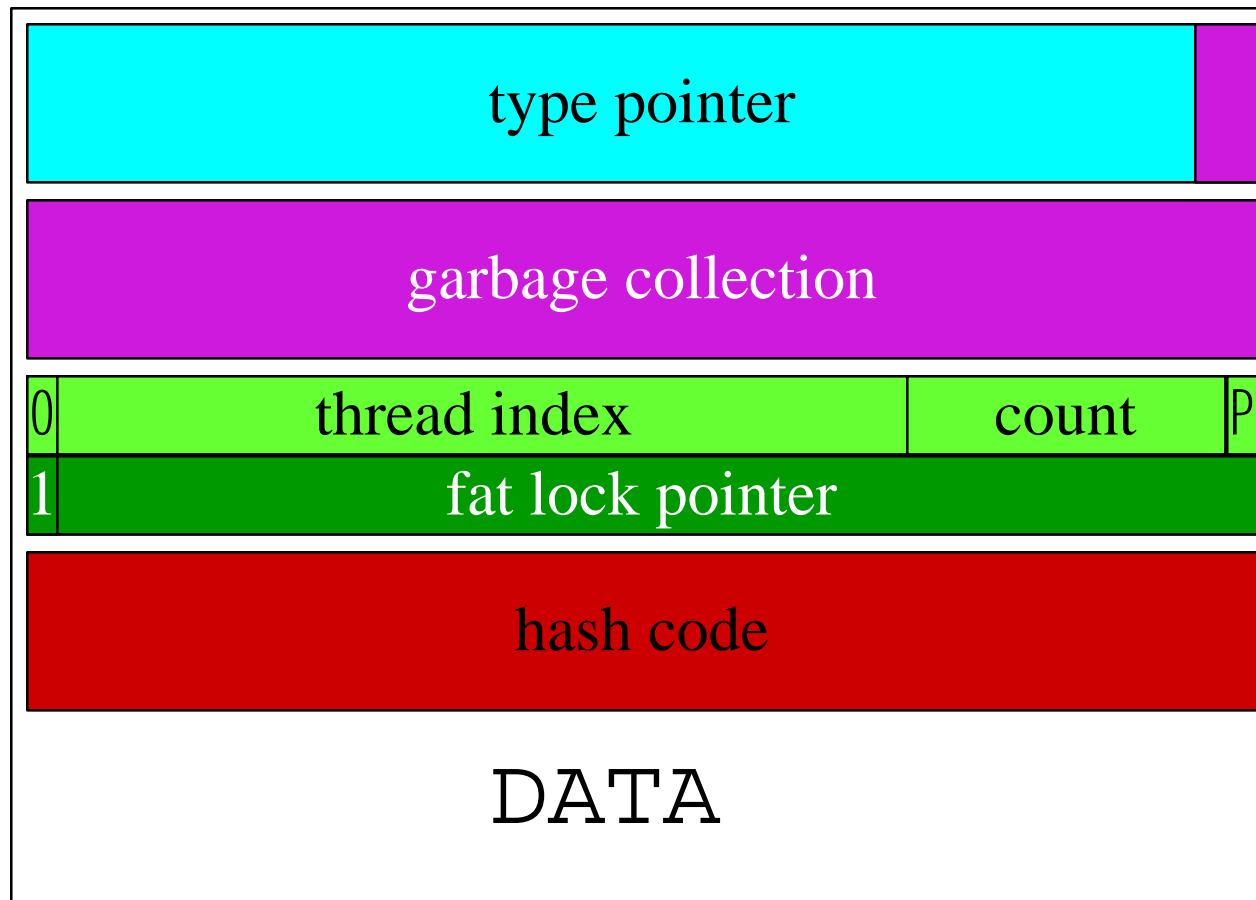
Decoupled Object Model

Synchronization and GC

- ◆ In advanced system, both are concurrent
- ◆ Portable atomic design:
 - requires separate words in object
 - but space overhead is onerous
- ◆ Solution
 - decouple
 - only consume space where needed

Decoupled Java Object Model

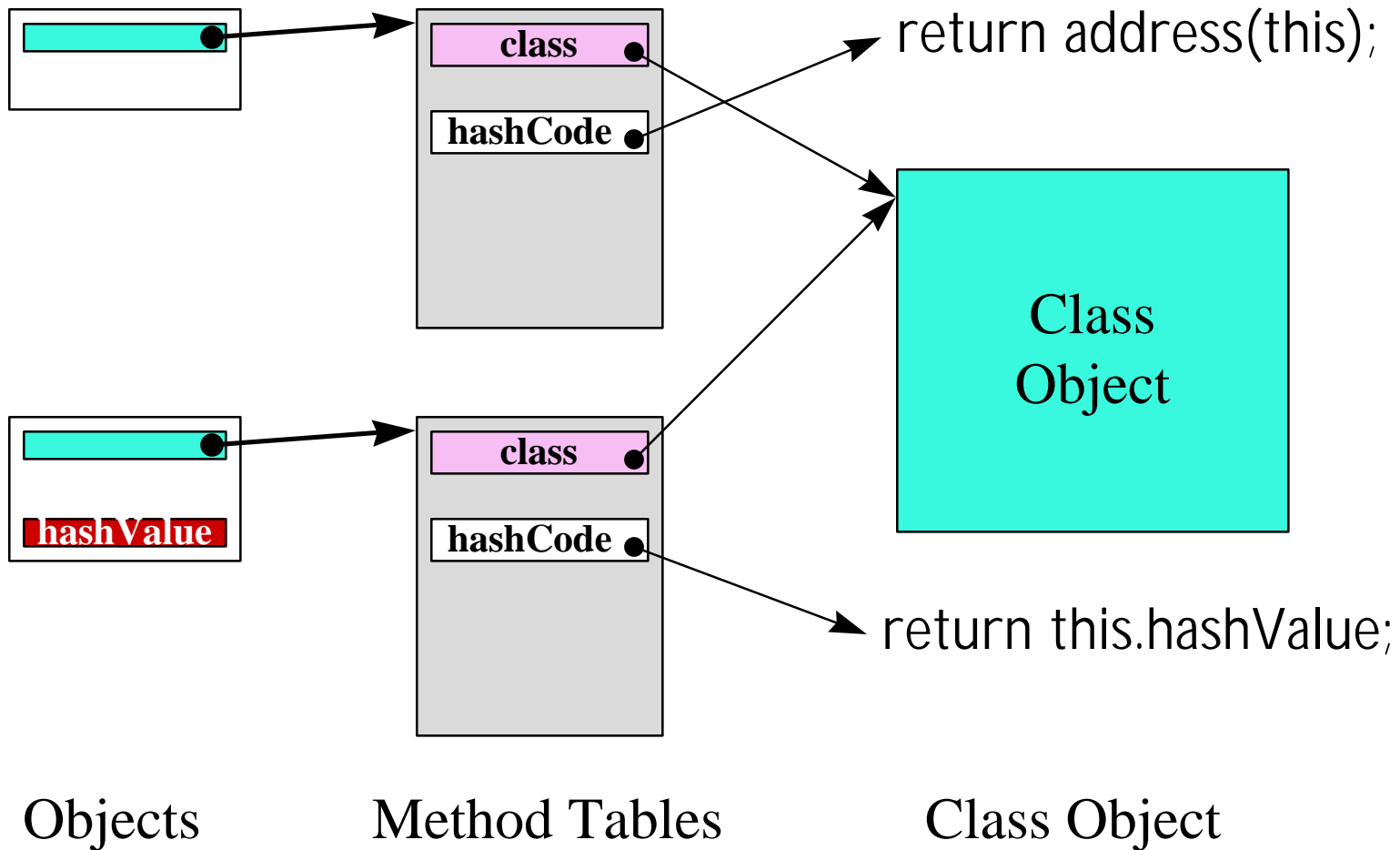
33222222222211111111110000000000
10987654321098765432109876543210



Optional Synchronization

- ◆ Introduce at fixed offset when class has
 - **synchronized** method
 - **synchronized (this)** statement
 - explicitly implements **Synchronizable** interface
- ◆ Synchronized methods run at top speed
- ◆ Synchronized blocks must check offset
 - if no synchronization word, use hash table

Hash Code Compression

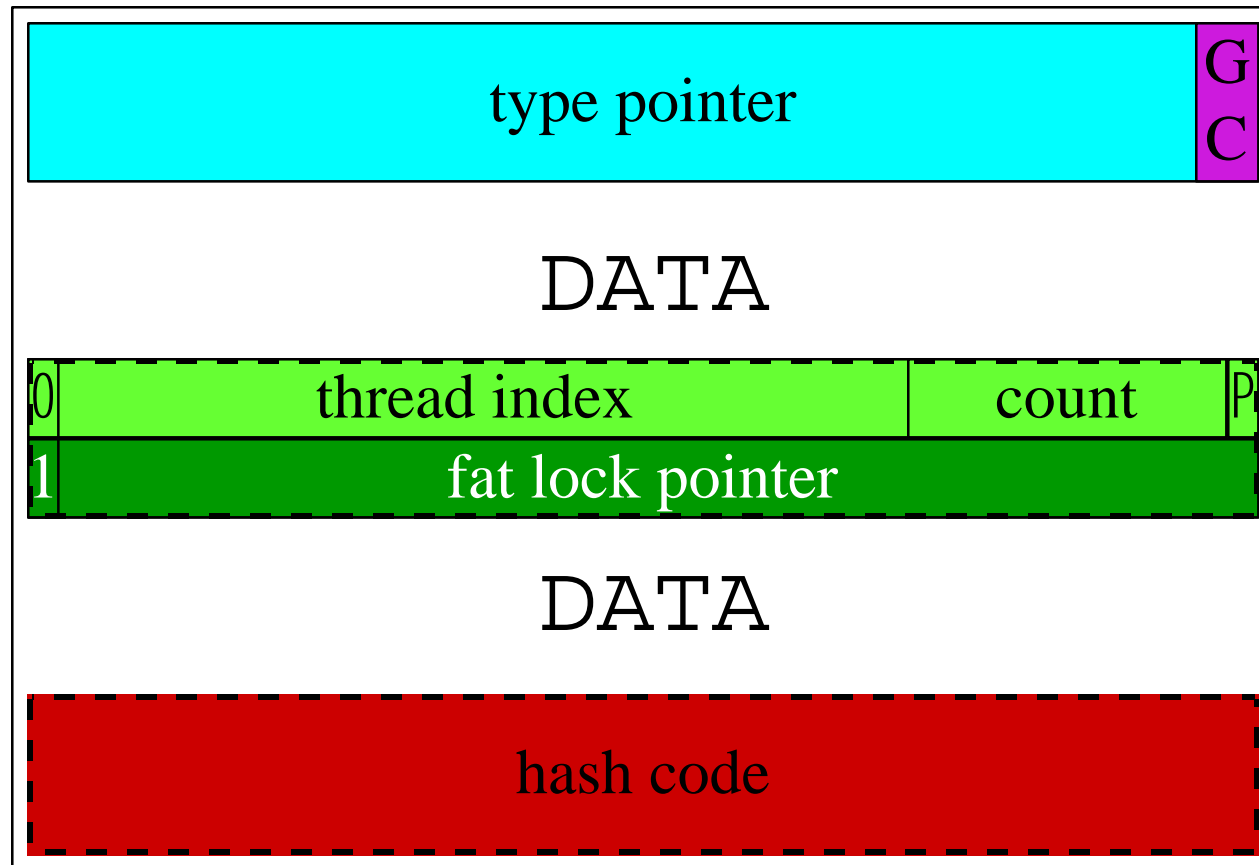


Simple Garbage Collector

- ◆ Stop-the-world
- ◆ Mark-and-sweep or mark-and-compact
- ◆ Only need two bits per object for markings
 - Use low 2 bits in type pointer

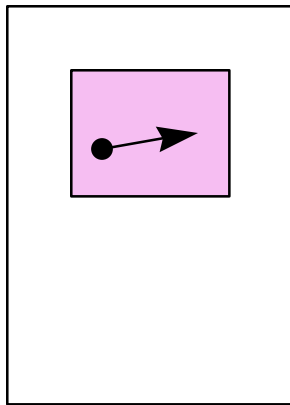
Object Model for Handheld

33222222222211111111110000000000
10987654321098765432109876543210

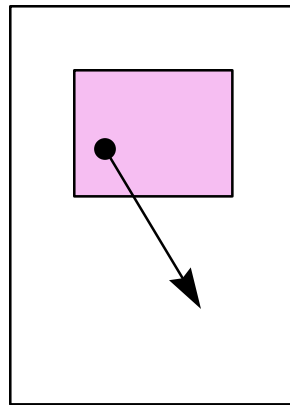


Memory Sandboxes

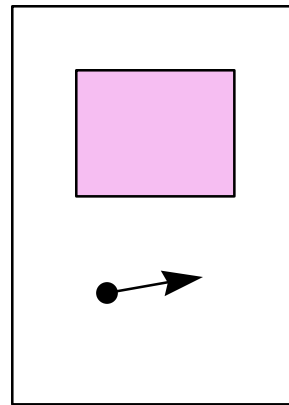
Sandbox Principle



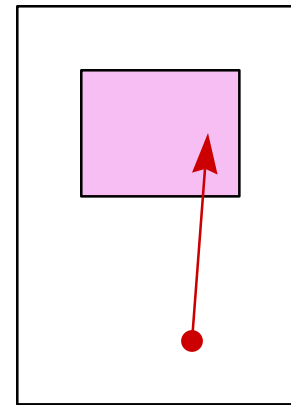
OK



OK



OK



Error!

*You may write indeed of all pointers in the memory.
Nevertheless of the pointers of the knowledge outside
of the sandbox you are not to write, or you shall surely die.*

Gen 2:16-17

Sandboxing

- ◆ Sandbox \Leftrightarrow Generational GC
 - throw exception, don't record pointer in table
 - check is only 2-3 instructions in common case
- ◆ Modify bytecode implementations
 - *putfield*
 - *putstatic*
 - *aastore*

Consequences of Sandboxing

- ◆ Target: medium-grain computations
- ◆ Use “handheld” object model in sandbox
- ◆ When method exits, release sandbox
 - in normal case, no GC required at all
- ◆ Can omit synchronization as well

Stack allocated, no GC, no sync, one word overhead per object
Speed and Compactness of C++
Semantics of Java

Sandbox Class

```
public class Sandbox implements Runnable {  
  
    private final Runnable object;  
  
    public Sandbox(Runnable object) {  
        this.object = object;  
    }  
  
    private native void runInternal();  
  
    public void run() {  
        runInternal();  
    }  
}
```

Issues

- ◆ “Unsandboxing” class needed:
 - return to global space from within sandbox
- ◆ Code stability issue with libraries:
 - “sandboxability” should be part of interface
- ◆ Implementation underway in JDK 1.1.6
 - immediate benefit for WebSphere

Concurrent Garbage Collection with Reference Counting

Reference Counting

- ◆ GC technology of choice for:
 - Operating systems
 - » Bell Labs Inferno, DEC Firefly, etc.
 - Distributed systems
- ◆ Advantages
 - locality
 - incrementality
 - scalability

Reference Counting Problems

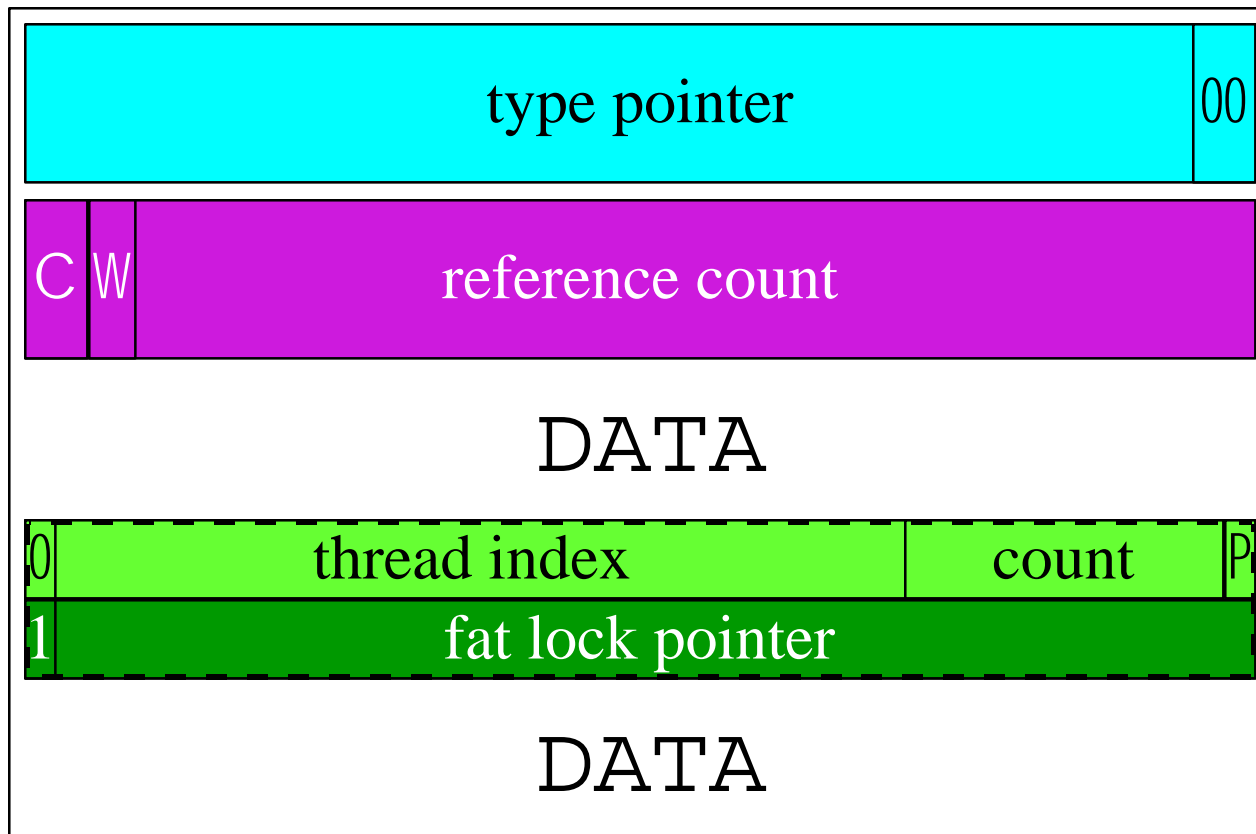
1. space overhead for reference counts
2. cost of reference counting heap updates
3. cost of reference counting stack loads
4. concurrency control on reference counts
5. cycle collection needed anyway

Approach

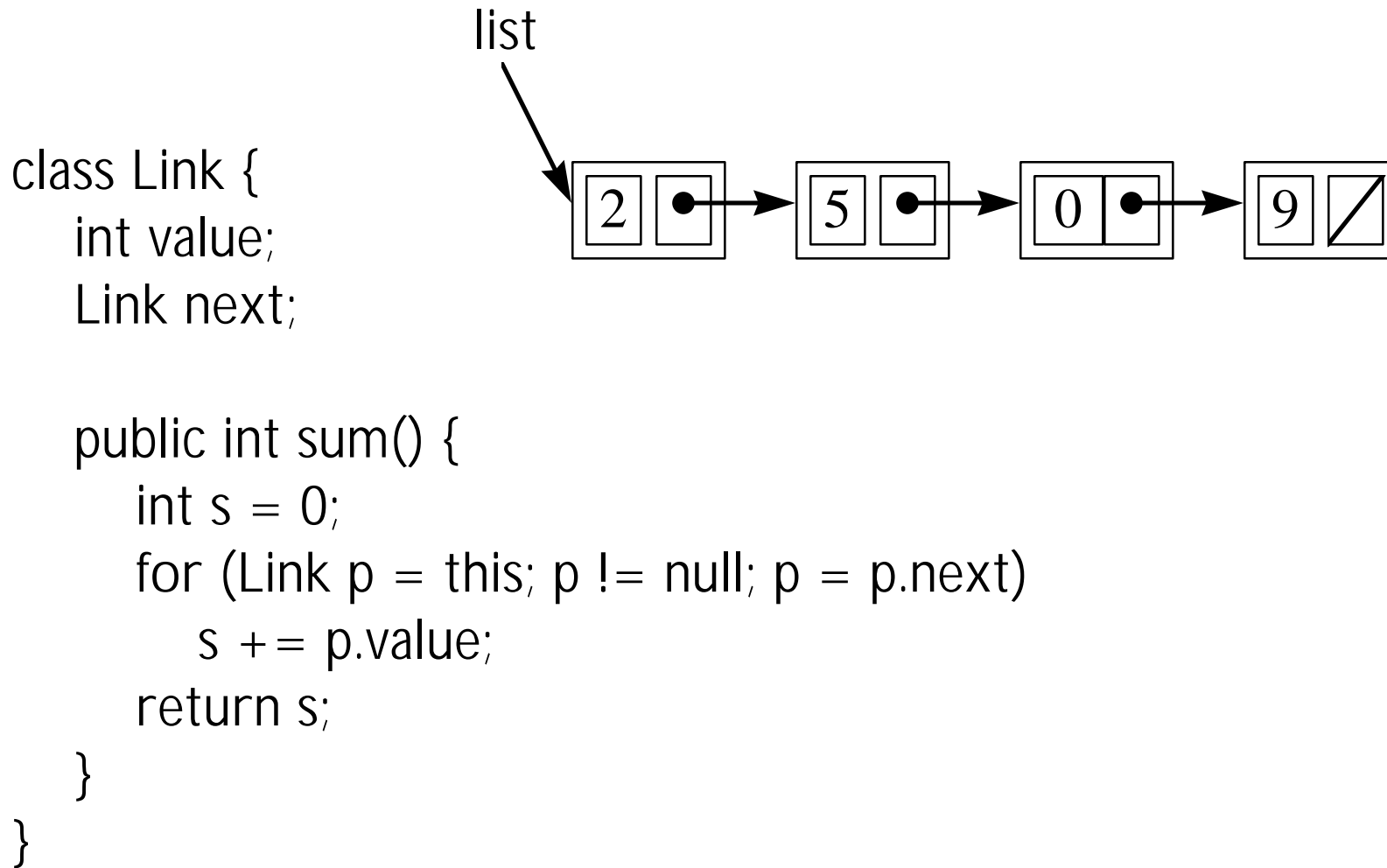
- ◆ Use pure reference counting
 - no hybrid schemes
- ◆ Coalescing, but not copying, collector
- ◆ Start with uniprocessor version
 - evolve for concurrency

Object Model

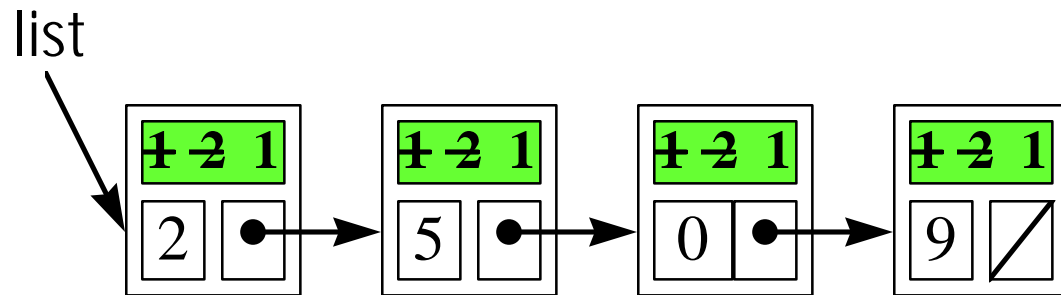
33222222222211111111110000000000
10987654321098765432109876543210



Problem 3: Stack Loads



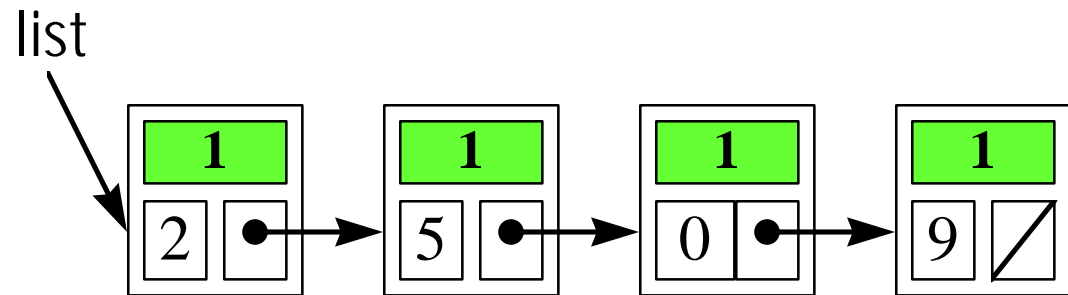
Simple Implementation of RC



```
public int sum() {  
    int s = 0;  
    for (Link p = this, inc(p); p != null; dec(p), p = p.next, if (p) inc(p))  
        s += p.value;  
    return s;  
}
```

Fix: Only Update RC at Safe Points

```
public int sum() {  
    int s = 0;  
    Link p = this;  
    do {  
        for (int i = 0; p != null && i < 256; i++, p = p.next)  
            s += p.value;  
        if (p) { inc(p); safePoint(); dec(p); }  
    } while (p != null);  
    return s;  
}
```



Methods Calls and **new** Operations

```
public Link summary() {  
    Link q = this.next;  
    Link p = new Link();  
    Link r = p;  
    r.value = q.sum();  
    return p;  
}
```



```
public Link summary() {  
    Link q = this.next;  
    inc(q);  
    safePoint();  
    Link p = new Link();  
    Link r = p;  
    inc(r);  
    safePoint();  
    p.value = q.sum();  
    dec(q);  
    dec(r);  
    return p;  
}
```

Solution: Lazy Reference Counting

```
public Link summary() {  
    Link q = this.next;  
    inc(q);  
    safePoint();  
    Link p = new Link();  
    Link r = p;  
    inc(r);  
    safePoint();  
    p.value = q.sum();  
    dec(q);  
    dec(r);  
    return p;  
}
```



```
public Link summary() {  
    Link q = this.next;  
    optionalSafePoint(q);  
    Link p = new Link();  
    Link r = p;  
    optionalSafePoint(q,r);  
    p.value = q.sum();  
    return p;  
}
```

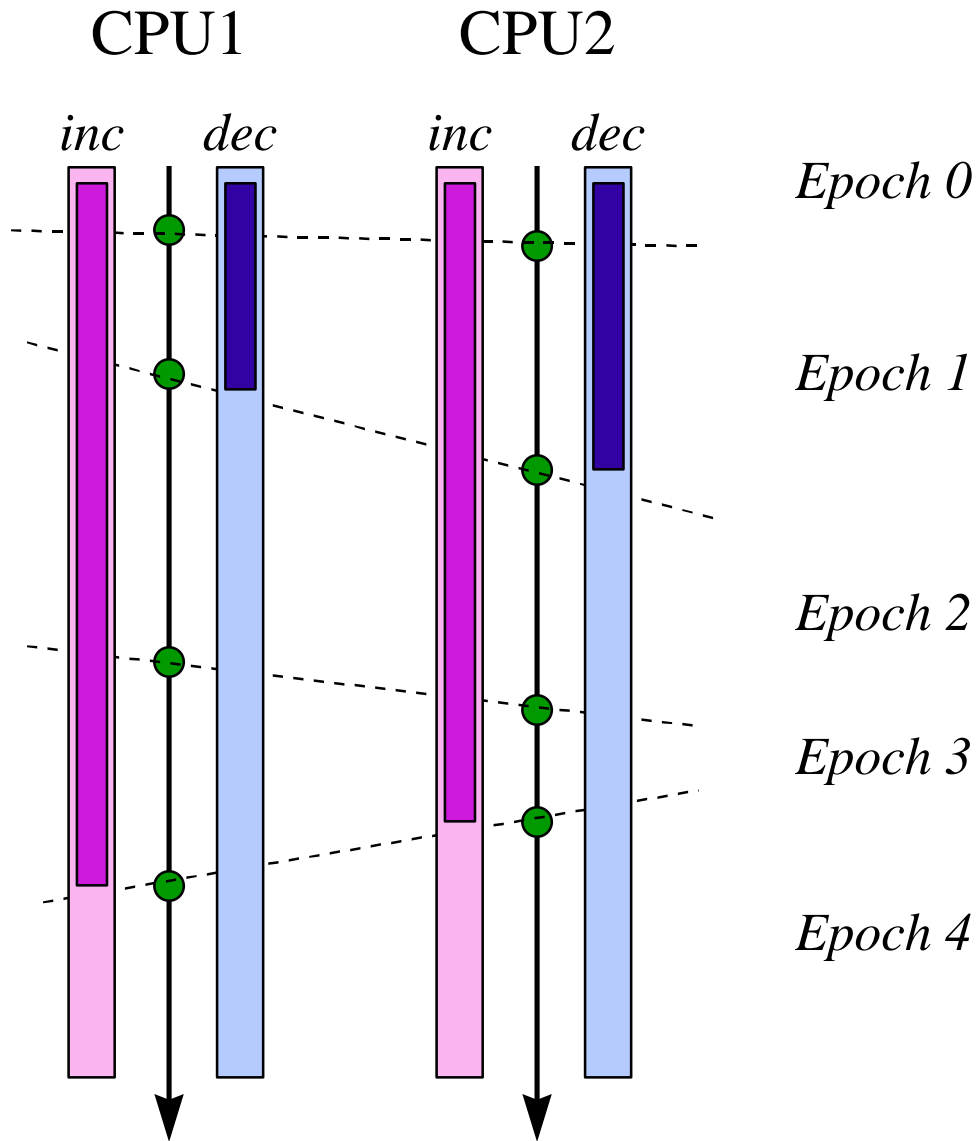
Safe Point Analysis

- ◆ Unbounded loops
 - Strip-mine
- ◆ Method calls
 - create thunks to *inc* before/*dec* after call
 - evaluate thunks lazily
- ◆ Calls to **new**
 - hoist or strip-mine availability checks
 - no safe point if GC not required; be lazy

Problem 4: Concurrency

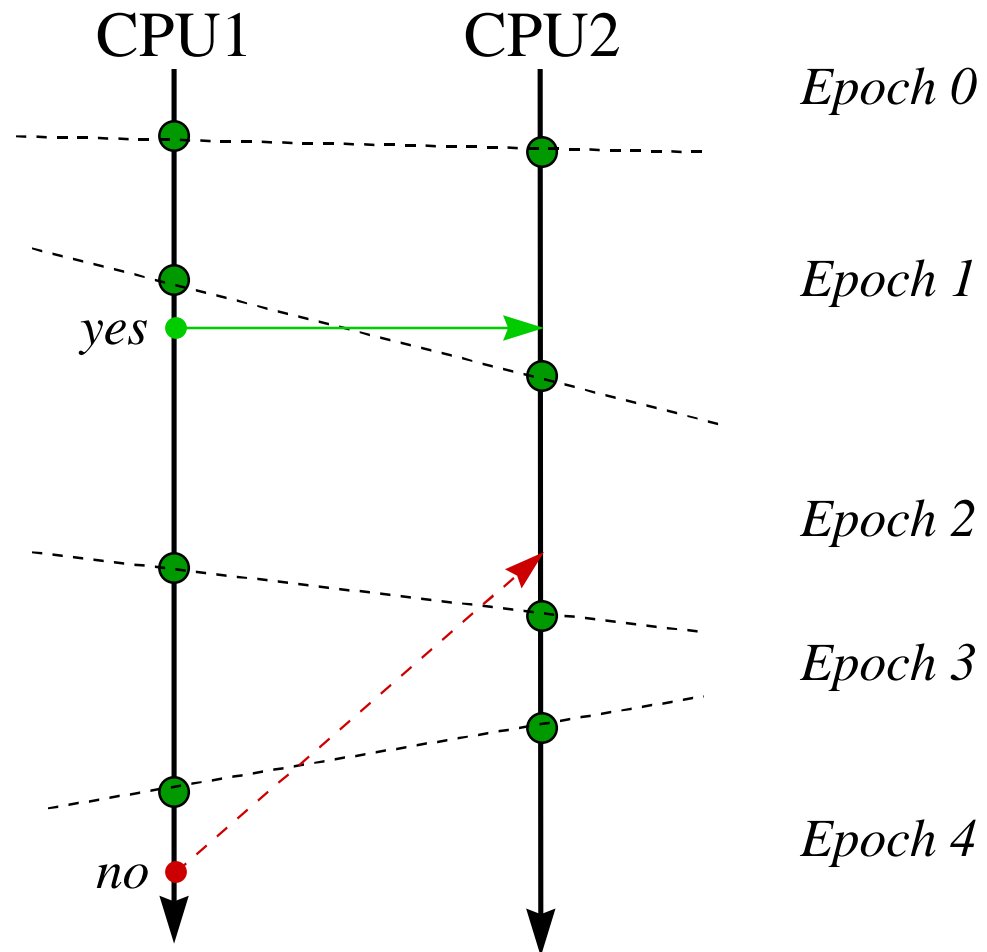
- ◆ Defer reference count updates to safe points
- ◆ Per-CPU buffer of *inc's* and *dec's*
- ◆ Global reference count lock
 - amortize synchronization across many updates
 - can partition for increased scalability
- ◆ Note: pointer writes must be atomic
 - but not synchronized
- ◆ Ragged barriers w/deferred *dec's*

4
0
Global Epoch *Included CPU's*



Why It Works

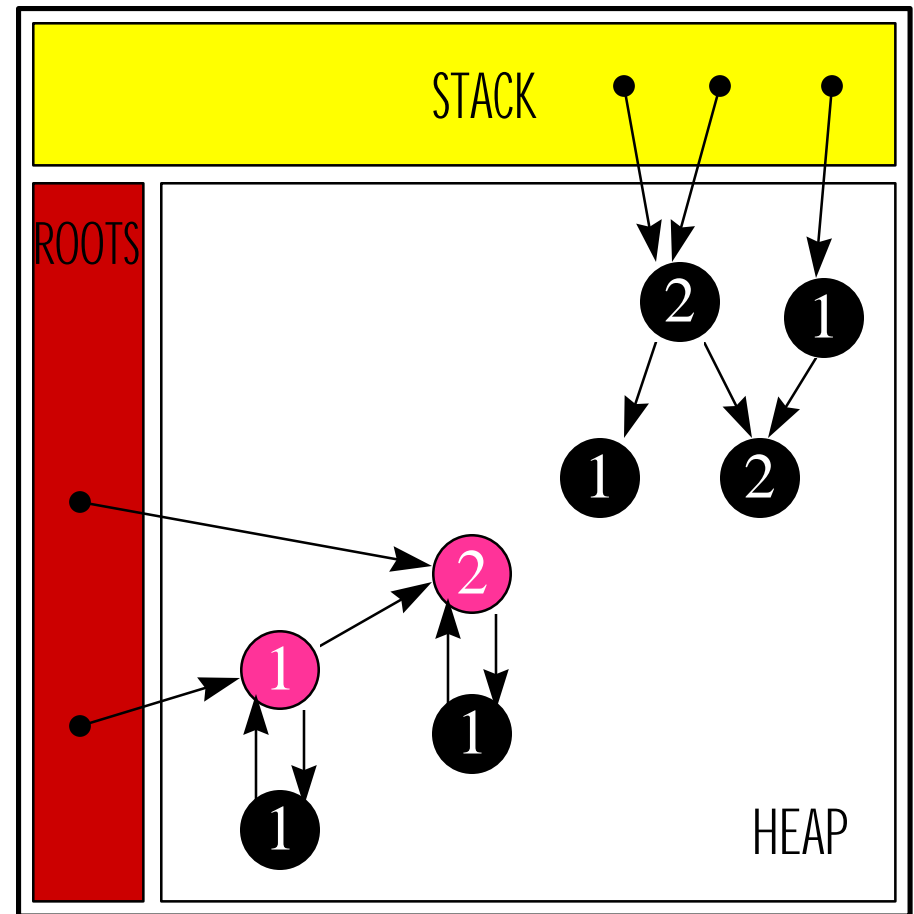
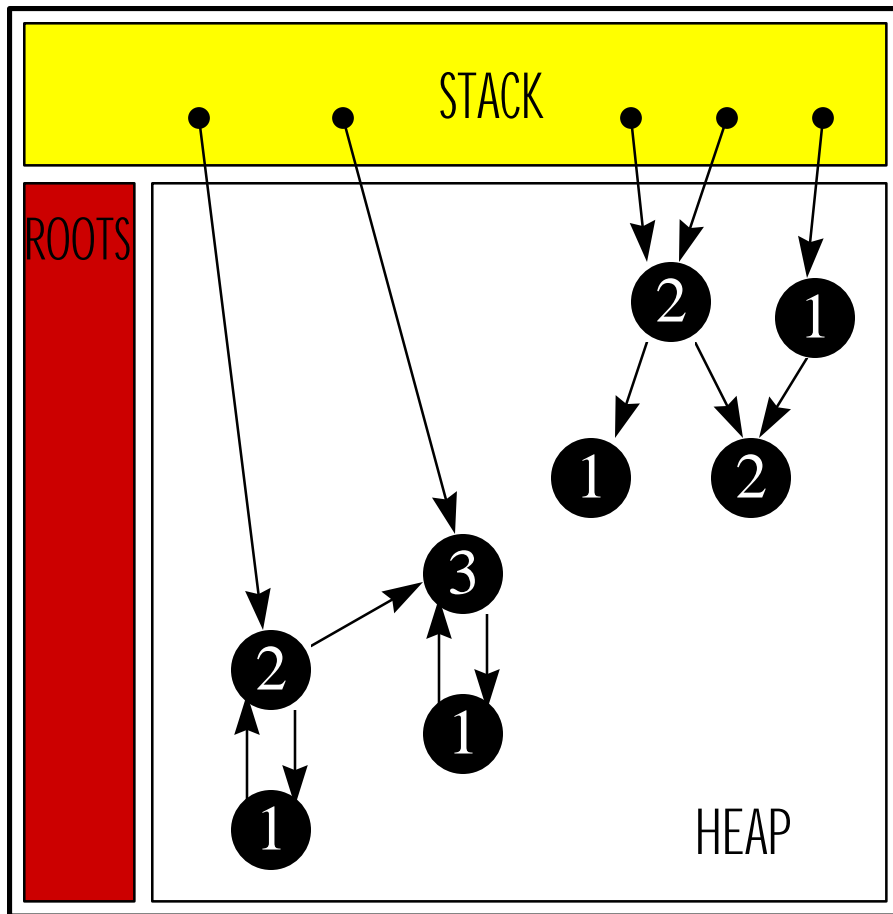
- ◆ Causality can flow backwards across one epoch
- ◆ But not two!
- ◆ Two epochs are equal to a barrier synchronization



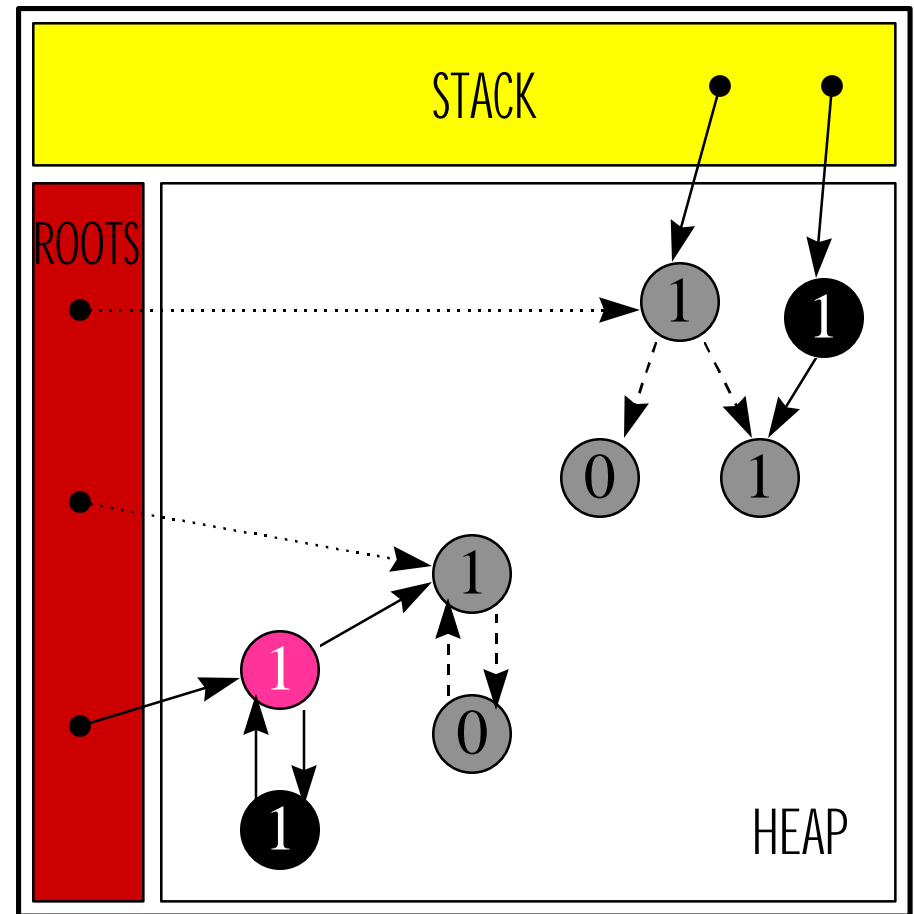
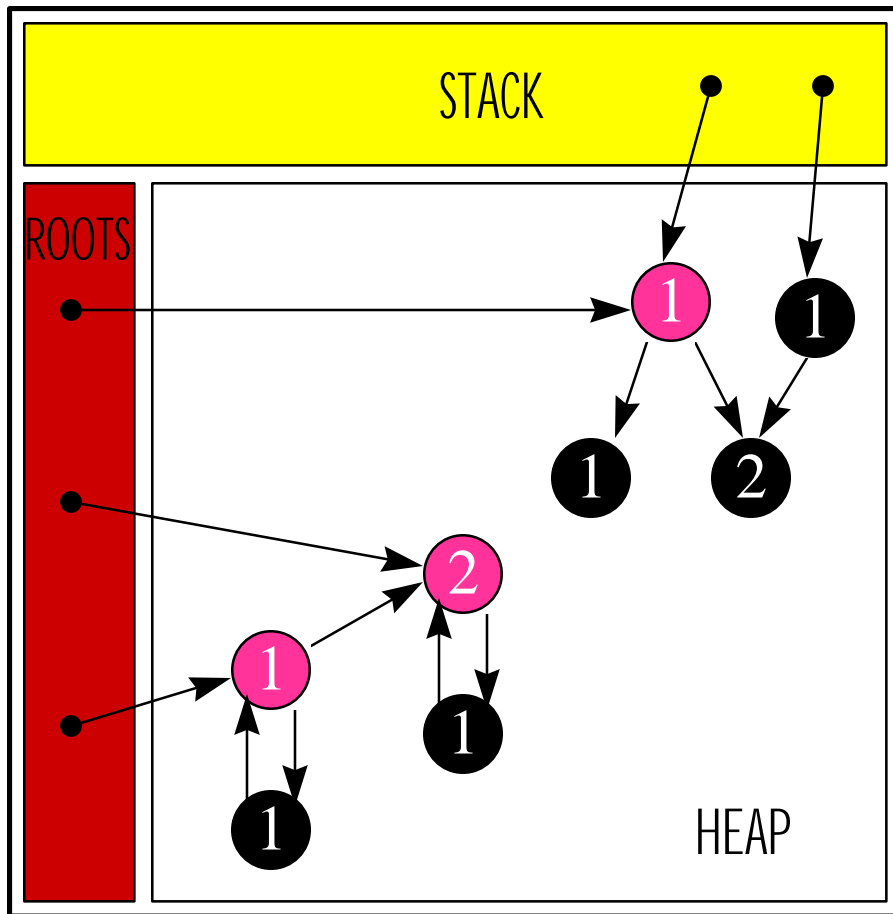
Problem 5: Cycle Collection

- ◆ Not needed for inherently acyclic types
- ◆ Maintain set of possible roots of garbage:
 - when pointer to cyclic type *dec's* to > 0
 - cycle must have count > 1 somewhere
- ◆ Periodically sweep possible roots
 - do DFS from root
 - subtract reference counts due to edges
 - if root count becomes 0, cycle found
 - otherwise, restore counts

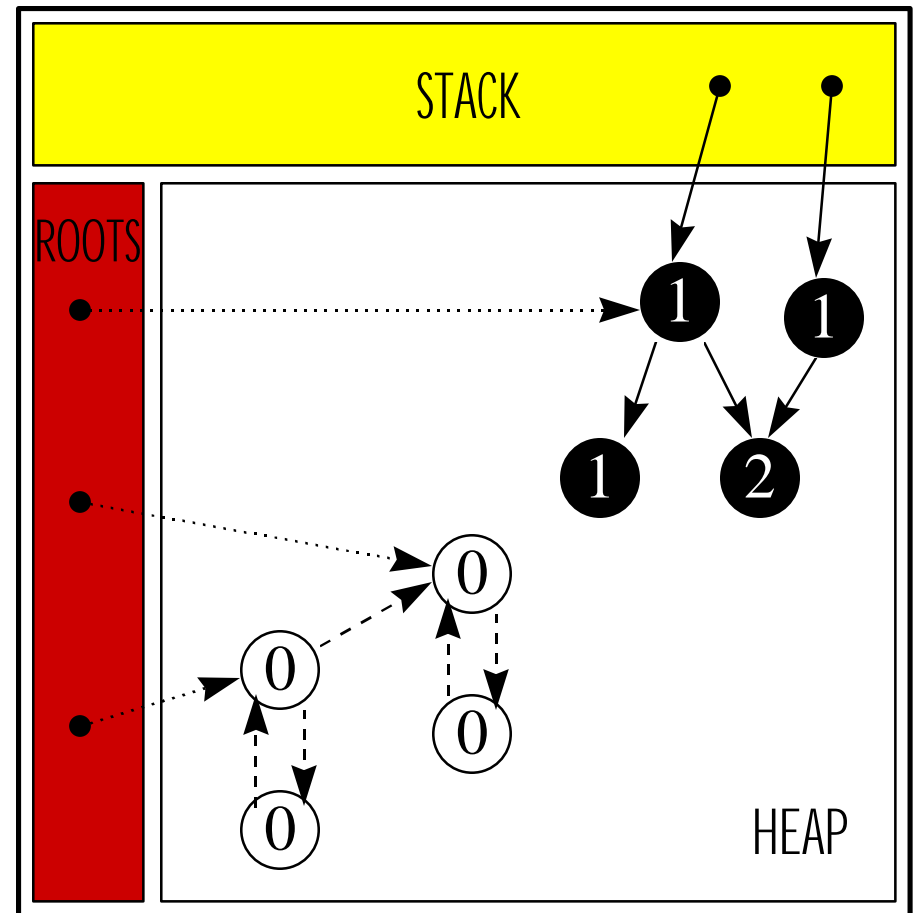
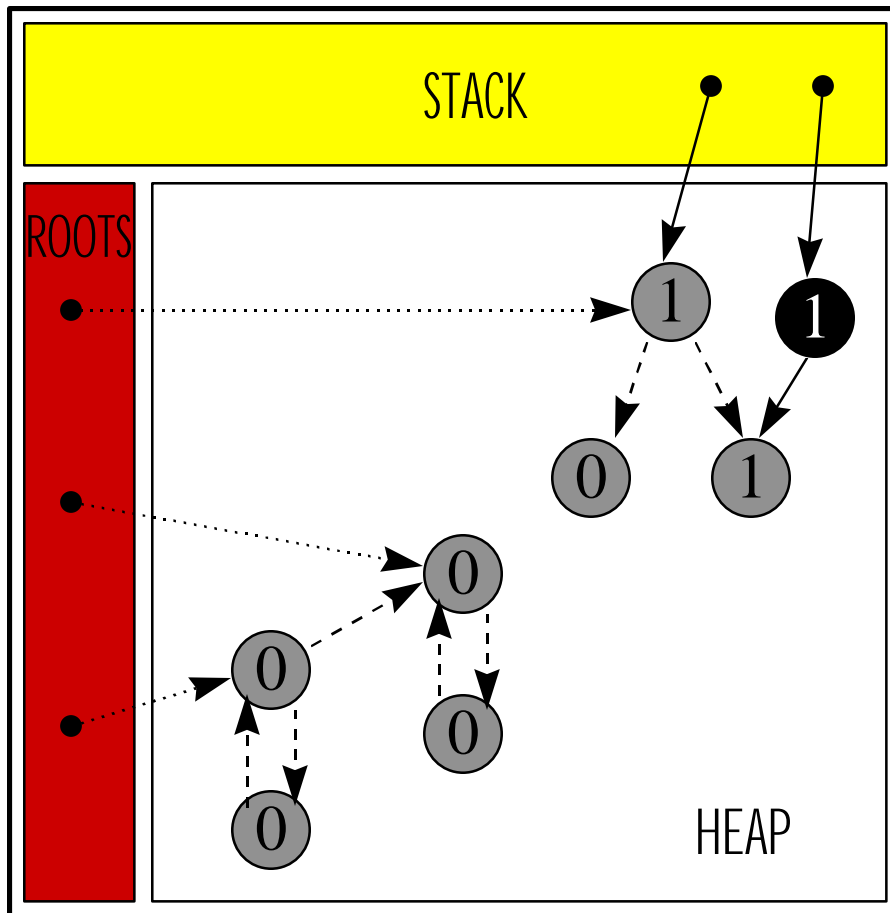
Adding Objects to the Root Set



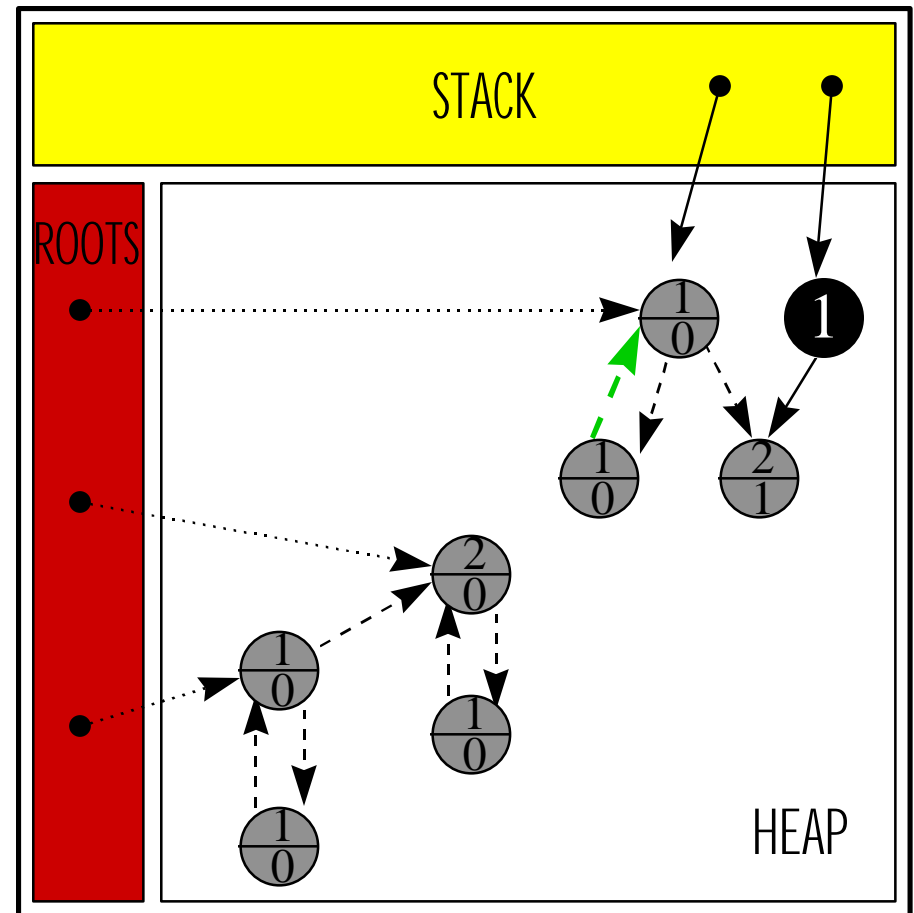
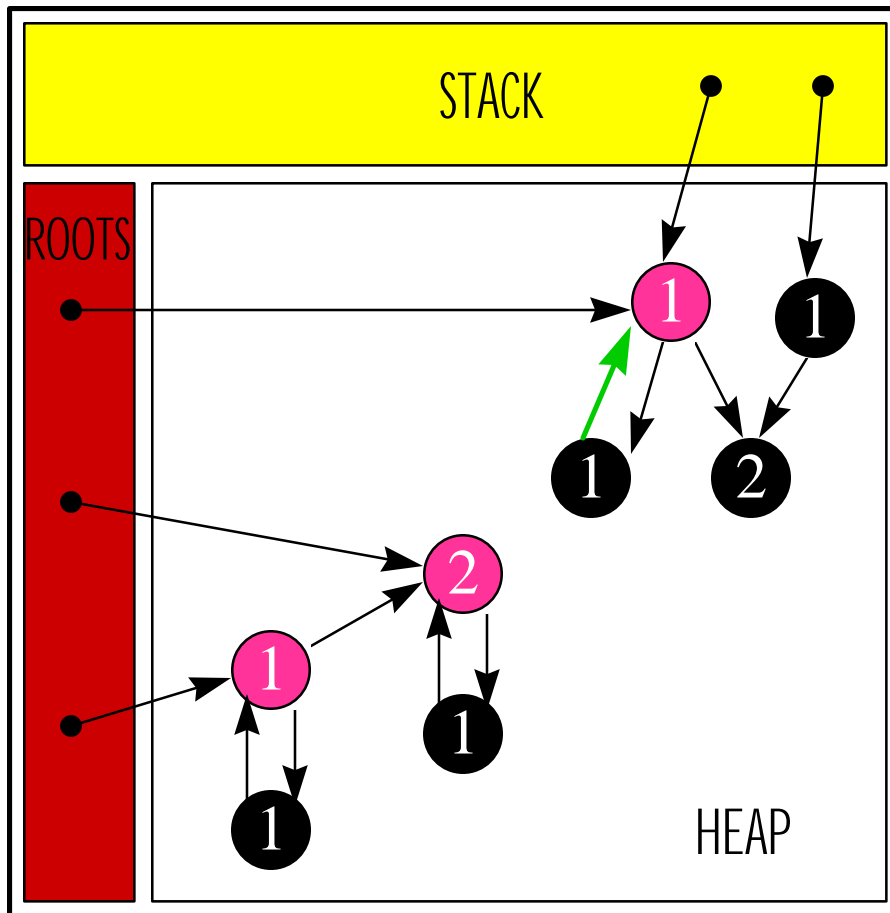
Cycle Collection: Marking Phase



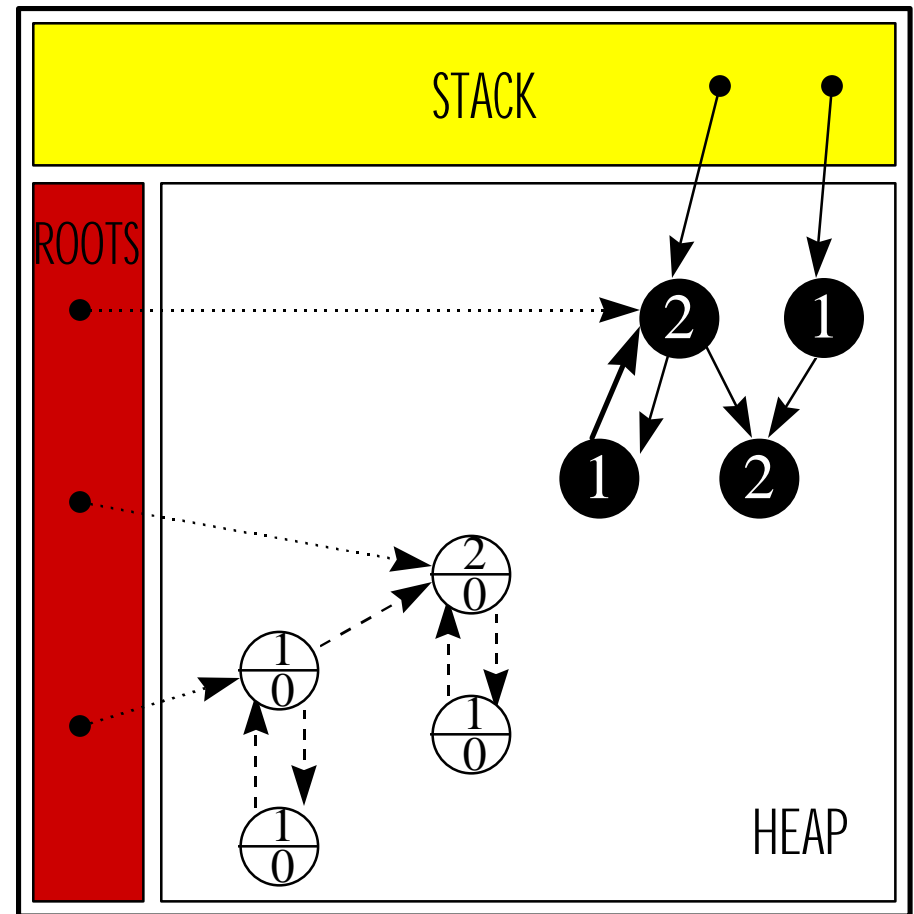
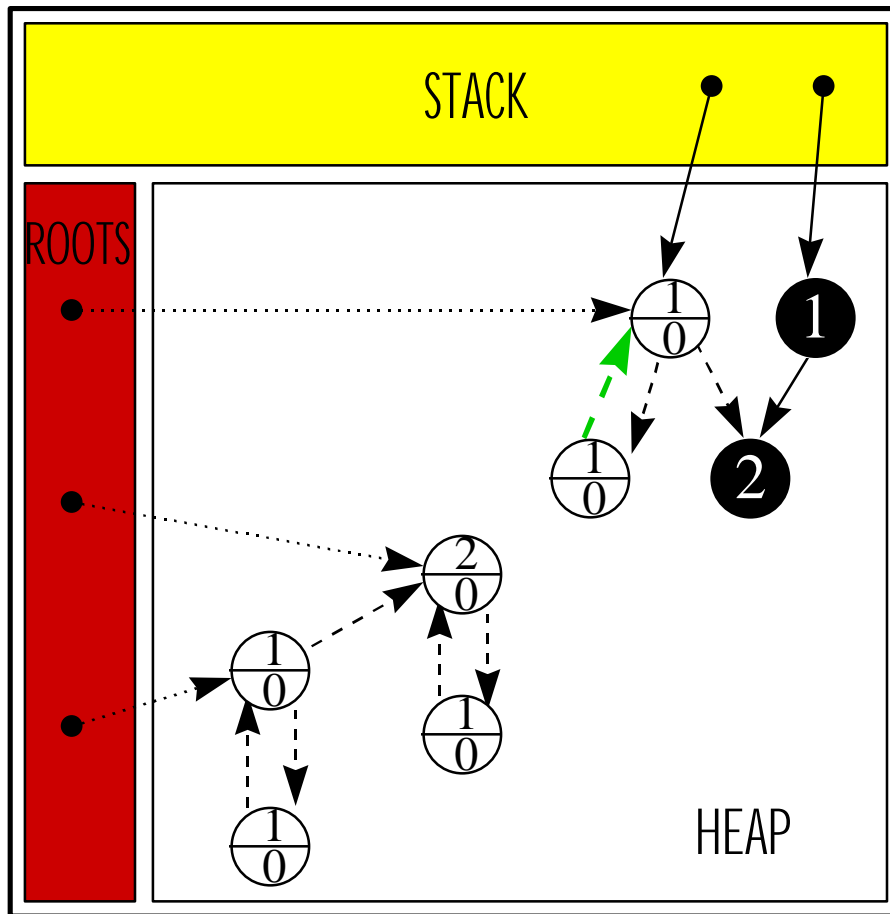
Cycle Collection: Finding Garbage



Problem: Concurrent Mutator

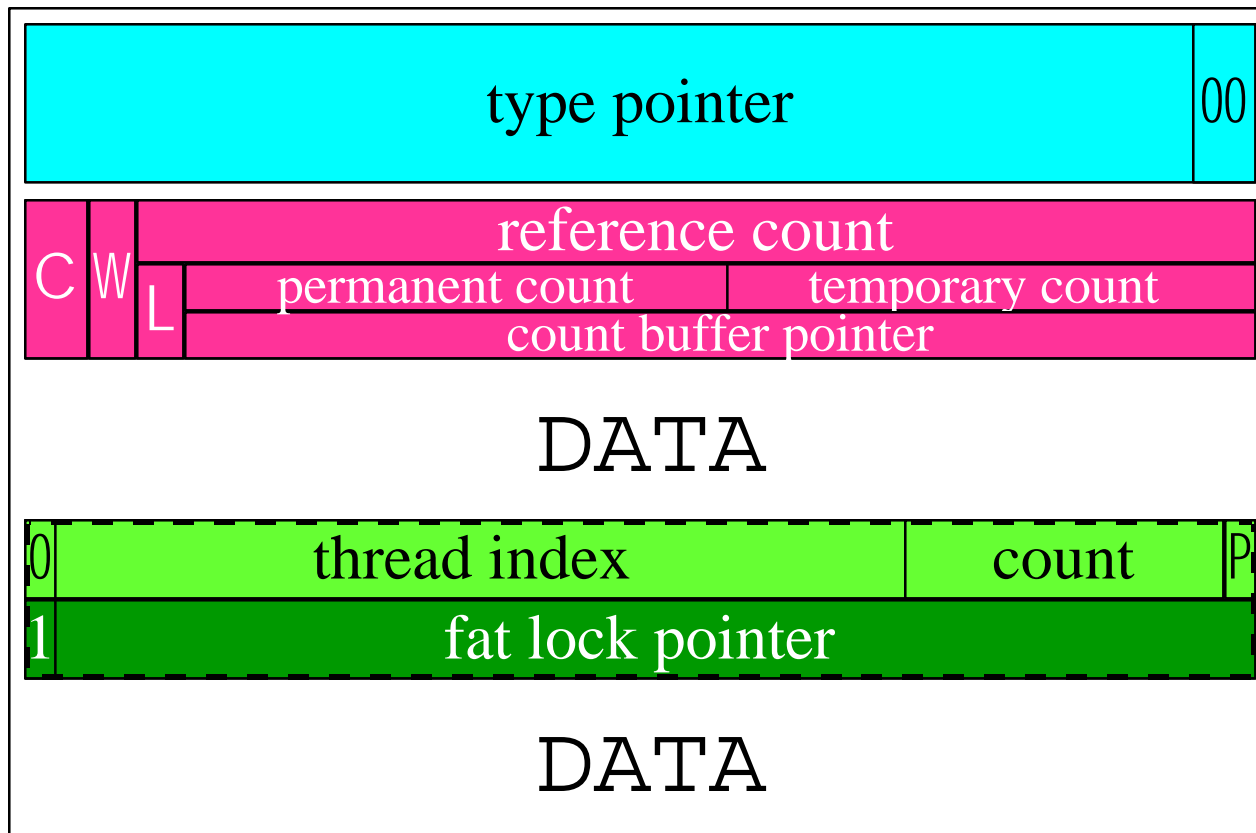


Fix: Defer Collection to Barrier



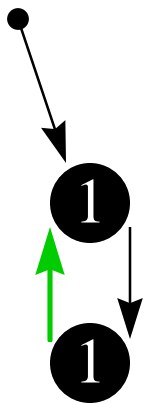
Revised Object Model

33222222222211111111110000000000
 10987654321098765432109876543210



An Interesting Non-Problem

- ◆ Safe point optimizations eliminate *dec*'s
- ◆ Will we miss *dec*'s that introduce roots? No!
 - cycles are entirely in the heap
 - heap *inc*'s and *dec*'s are always processed
 - when cycle created, "closing" node has $RC > 0$
 - extra edge results in *inc* of closing node, so $RC > 1$
 - can't be garbage until RC decreases



Properties of Concurrent RC GC

- ◆ Incremental
- ◆ Concurrent
 - Very loosely synchronized
- ◆ Scalable
- ◆ Non-Stop
 - Cycle detection can be interleaved
- ◆ Mostly only reference counts heap updates

Synergy: a Java fit for Servers

- ◆ "Transactions" run w/very low overhead
 - No garbage collection
 - No synchronization
 - One-word headers
- ◆ **Shared, Long-lived Processes are Robust**
 - Incremental nature of reference counting
 - Concurrent, non-stop collection
- ◆