Garbage Collection: Overview, Techniques, Successes

Tony Printezis
tony.printezis@sun.com

Sun Microsystems Laboratories
MS BUR02-311
1 Network Drive
Burlington, MA 01803
USA

Who Am I?

- Tony Printezis
  - Member Of Technical Staff, JTech Group,
  - Sun Microsystems Laboratories, East, MA
- Previously
  - Faculty Member, Dept of Computing Science,
  - University of Glasgow, Scotland
- Working on GC for $\sim$5 years
  - wrote first version of mostly-concurrent GC

Overview

- Introduction / GC Benefits
- Simple GC Techniques
- Incremental GC Techniques
- Generational GC Techniques
- GC in the Java HotSpot™ Virtual Machine
- GC Issues in the Real World
- Conclusions

Garbage Collection

- Traditional Explicit De-Allocation (C/C++)
  - Programmer allocates memory (`new`/`malloc`)
  - Programmer also has to de-allocate it (`delete`/`free`)
- *Automatic Memory Management* (aka GC)
  - Programmer allocates memory (`new`)
  - GC reclaims all unused memory

GC Brief History

- GC has existed since the 1960s!
  - LISP
- Functional / O-O / Logic languages
  - ML, Haskell, SmallTalk, etc.
- Conservative GCs
  - C and C++
- Mainstream (finally!) in the late 1990s
  - the Java™ programming language, then C#

GC Benefits

- **No dangling references**
  - (wrongly de-allocated memory)
- **No memory leaks**
  - (unused, not de-allocated memory)
- **Greater programmer productivity**
  - no need to de-allocate memory
  - simplified team work
  - simplified APIs

The Java Language Benefits GC Too

- "Chicken and Egg" problem
  - no good GC $\rightarrow$ no applications use it
  - no applications to test $\rightarrow$ no improved GC
- The Java language is *great* for GC research
  - large amounts of industrial-strength code
  - several industrial-strength JVMs
  - industry-standard benchmarks
  - academia / industry both interested

Programmers and GC

Three categories of programmers:

- ✔ Learned to program using garbage collection; really hate explicit de-allocation
- ✔ Learned to program using explicit de-allocation; migrated smoothly to garbage collection *(me!)*
- ✘ Learned to program using explicit de-allocation; really hate garbage collection

Memory Costs and GC

- Why do we need GC anyway?
  - memory these days is *really cheap* ($\sim$\$200/GB)
  - 64-bit address space is *really huge*
  - can't we keep adding more memory?
- One "real-world" application allocates
  - 20MB/sec, 1.2GB/min, 70.3GB/hour
  - translates to $\sim$\$14,000/hour (quite expensive!)
  - but, it would still take $>$27,000 years to fill up the 64-bit address space though!

Overview

- Introduction / GC Benefits
- ➜ Simple GC Techniques
- Incremental GC Techniques
- Generational GC Techniques
- GC in the Java HotSpot™ Virtual Machine
- GC Issues in the Real World
- Conclusions

Simple GC Techniques

- General Concepts
- Indirect Techniques
  - Mark-Sweep
  - Mark-Compact
  - Copying
- Direct Techniques
  - Reference Counting

Object

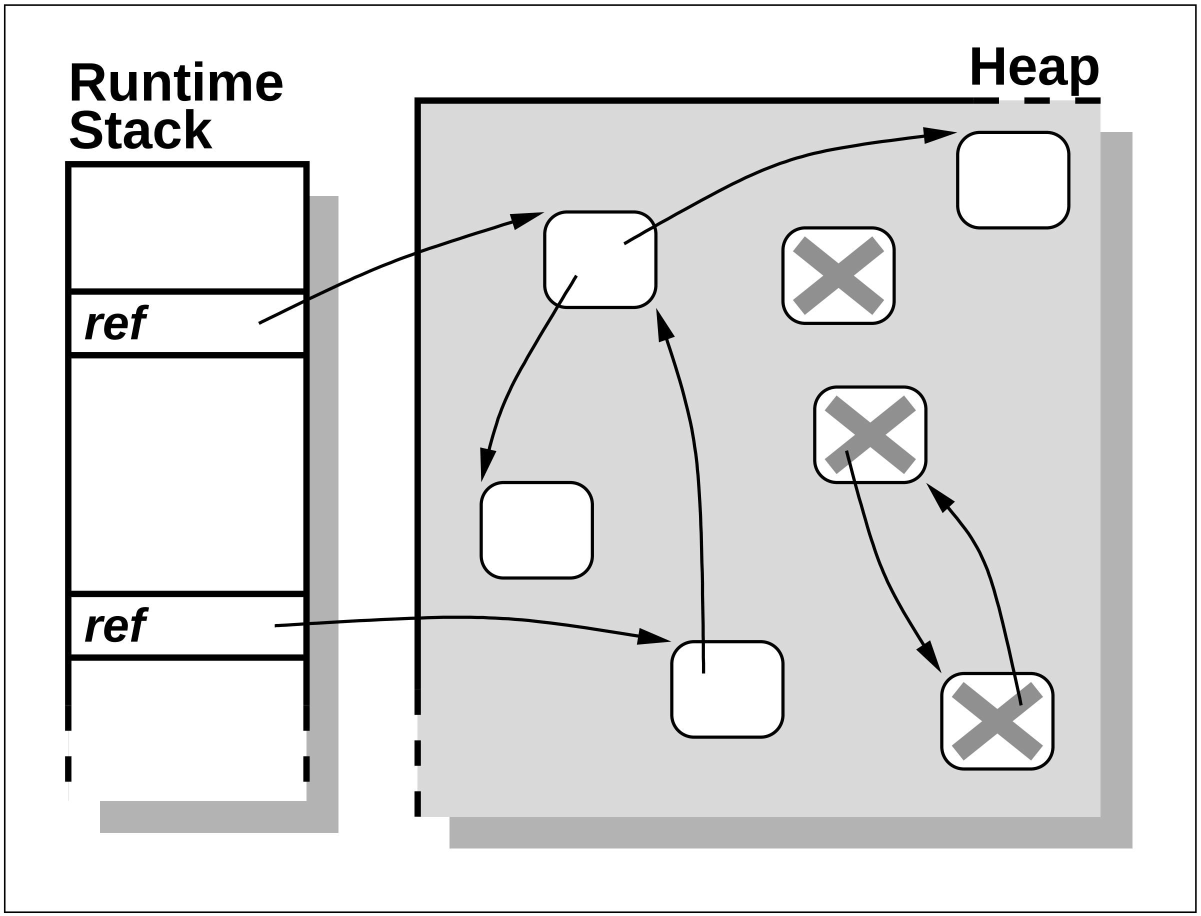
"A container, with a well-defined structure, of one or more fields, some of which can contain references."

- not only full-fledged objects, with encapsulation and inheritance in the context of object-oriented programming,
  - e.g. instances in the Java language, C++
- but also any kind of structured data records.
  - e.g. structs in C

Reachability

- *Roots*
  - memory locations that are *live* by default
  - e.g. runtime stack locations, static fields
  - *Root Objects*
    - objects directly reachable from the roots
- *Live Objects*
  - all objects transitively reachable from the roots
- *Garbage Objects*
  - all other objects

Reachability Example



GC Phases

- A GC has two main phases
  - *Identification* of garbage objects
  - *Reclamation* of garbage objects
- They are either distinct. . .
  - Mark-Sweep, Mark-Compact
- . . . or interleaved
  - Copying, Reference Counting

Fundamental GC Property

“When an object becomes garbage, it stays garbage.”

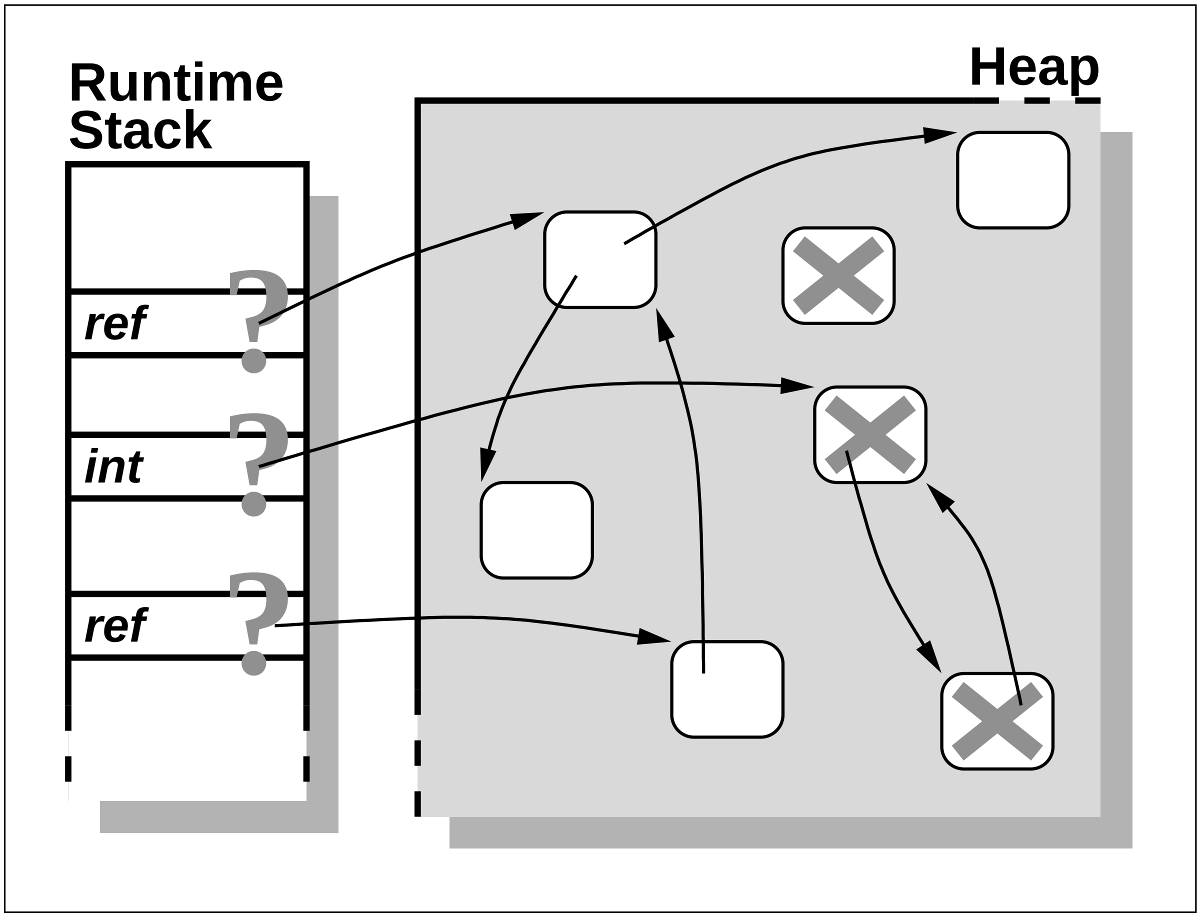
Exact (or Accurate) GC

- Can tell which memory locations contain obj refs
  - on stacks, objects, classes (statics)
- Allows object relocation
  - will have to update all refs to it
- Accurate liveness information
  - exactly all live objs marked
- Very flexible, but not free!
- Most JVMs have exact GCs (e.g. HotSpot JVM)

Conservative GC

- Can’t tell which memory locations contain obj refs
- Assume that what looks like an obj ref is an obj ref
  - can’t always relocate objects
  - object liveness information is conservative
- Easier to implement than exact
- GCs for C/C++
- Some JVMs have semi-conservative GCs

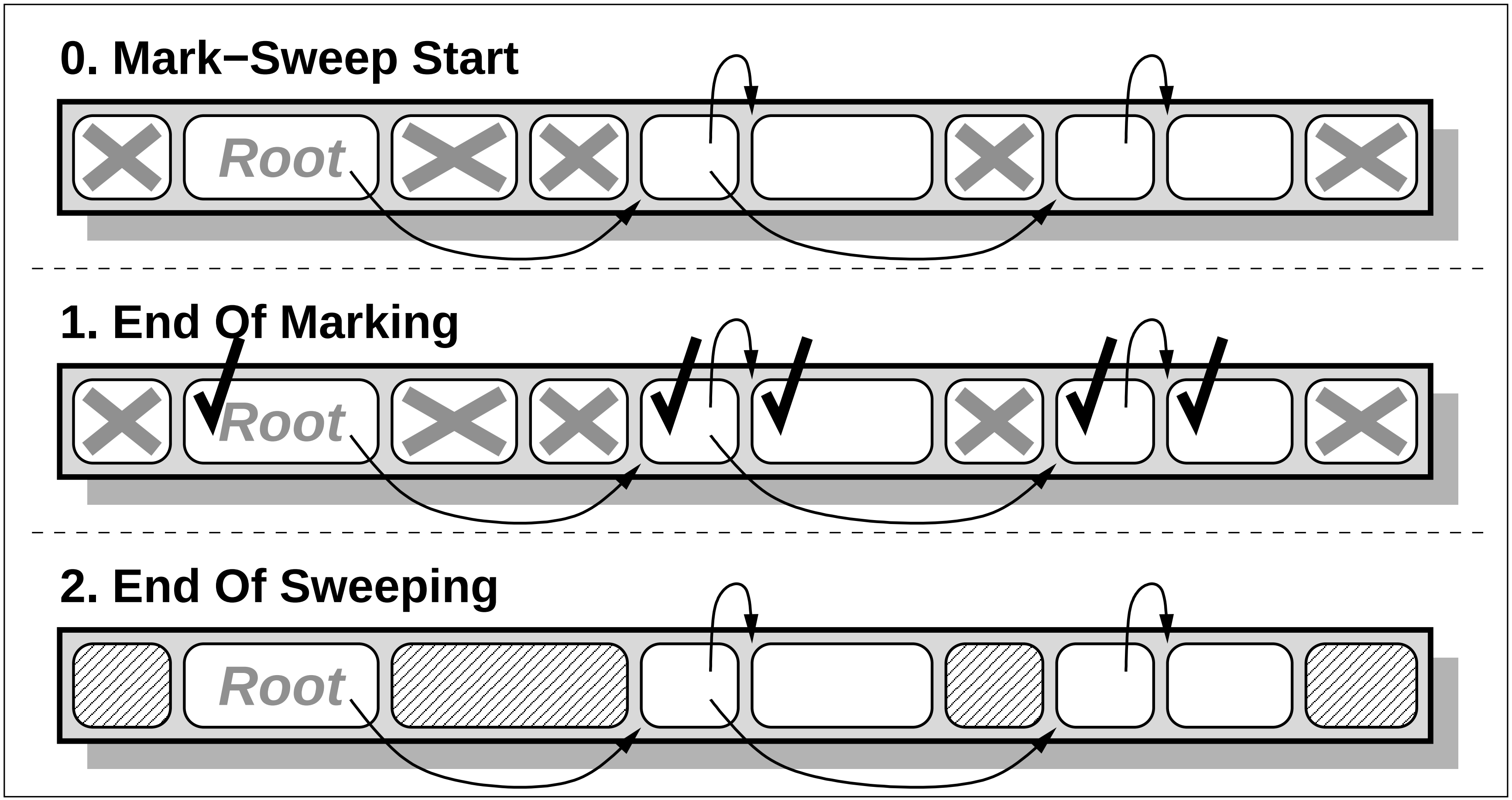
Conservative GC Example



Mark-Sweep

- Identify all live objects
  - marking phase
- Sweep over heap
  - de-allocate all garbage objects in-place
- Allocation
  - keep track of where free space is
    - e.g. free lists, bitmaps
  - reuse free space to satisfy allocation requests

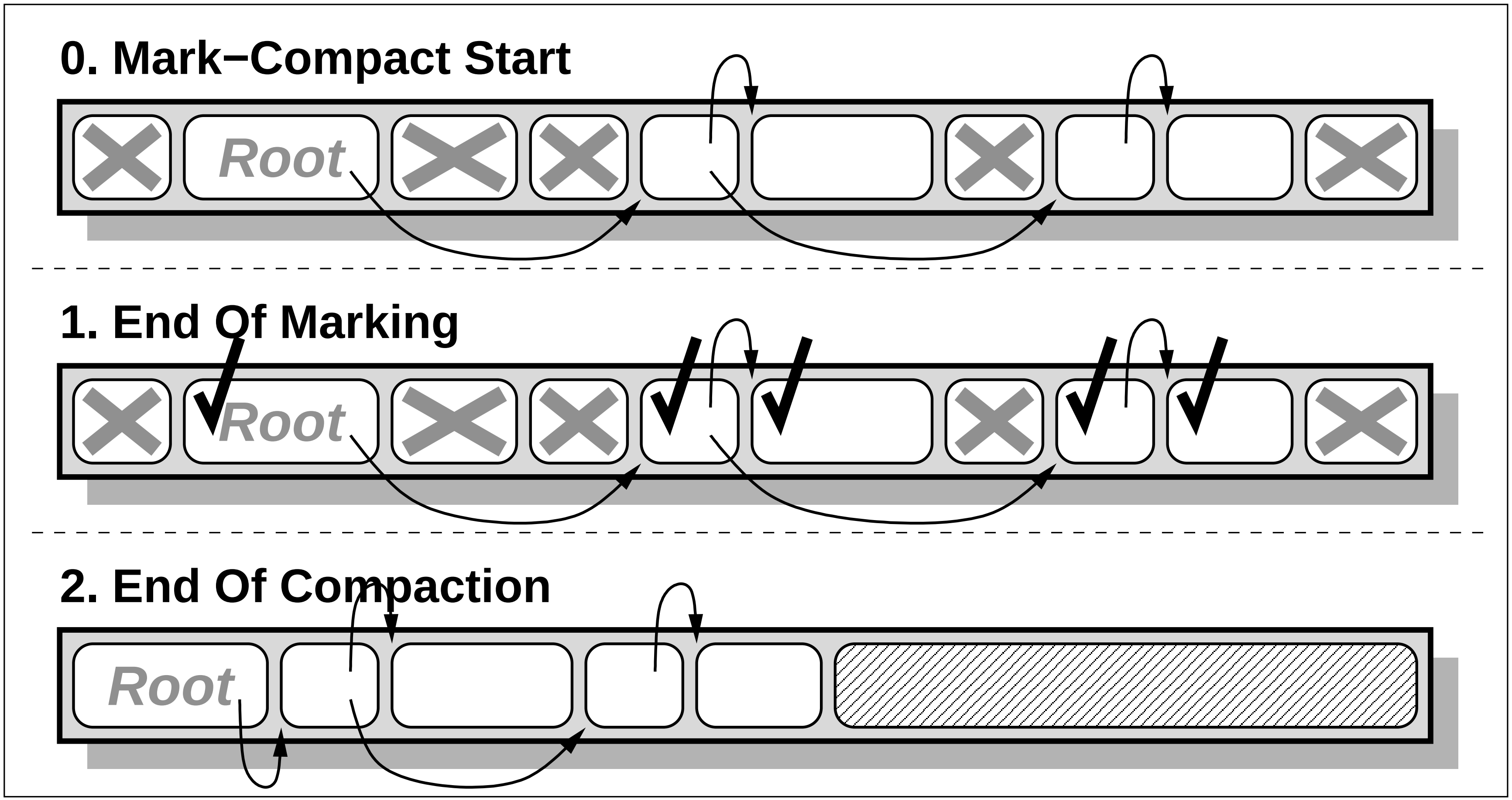
Mark-Sweep Example



Mark-Compact

- Identify all live objects
  - same as Mark-Sweep
- Sweep over heap
  - slide all live objects towards start of the heap
  - create single free chunk at the end of the heap
  - need to patch references as objects move
- Allocation
  - fast *“bump a pointer and check”*

Mark-Compact Example



Mark-Sweep vs. Mark-Compact

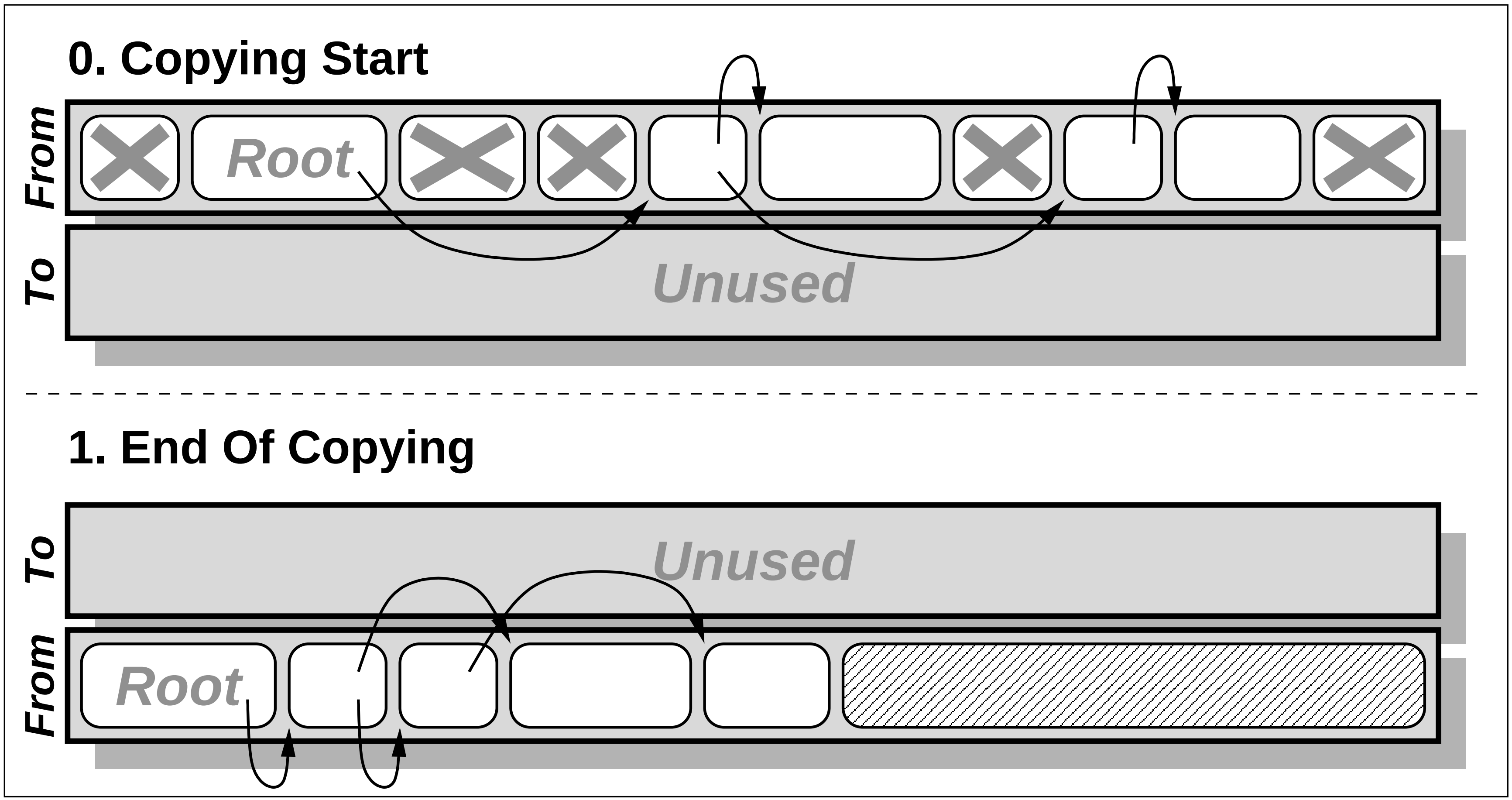
- Performance
  - compaction adds 2–2.5$\times$ overhead
- Complexity
  - Mark-Sweep: well-tuned free lists
  - Mark-Compact: compaction
- Fragmentation
  - Mark-Sweep suffers from it
  - Mark-Compact eliminates it

Sun microsystems

Copying GC

- Heap split into two equal-sized areas
  - from-space and to-space
- Mutator allocates/modifies objects in from-space
- GC visits transitive closure of live objects. . .
  - . . . and copies them to to-space
  - objects contiguously allocated in to-space
  - identification / copying interleaved
- Spaces swap rôles after GC

Copying GC Example



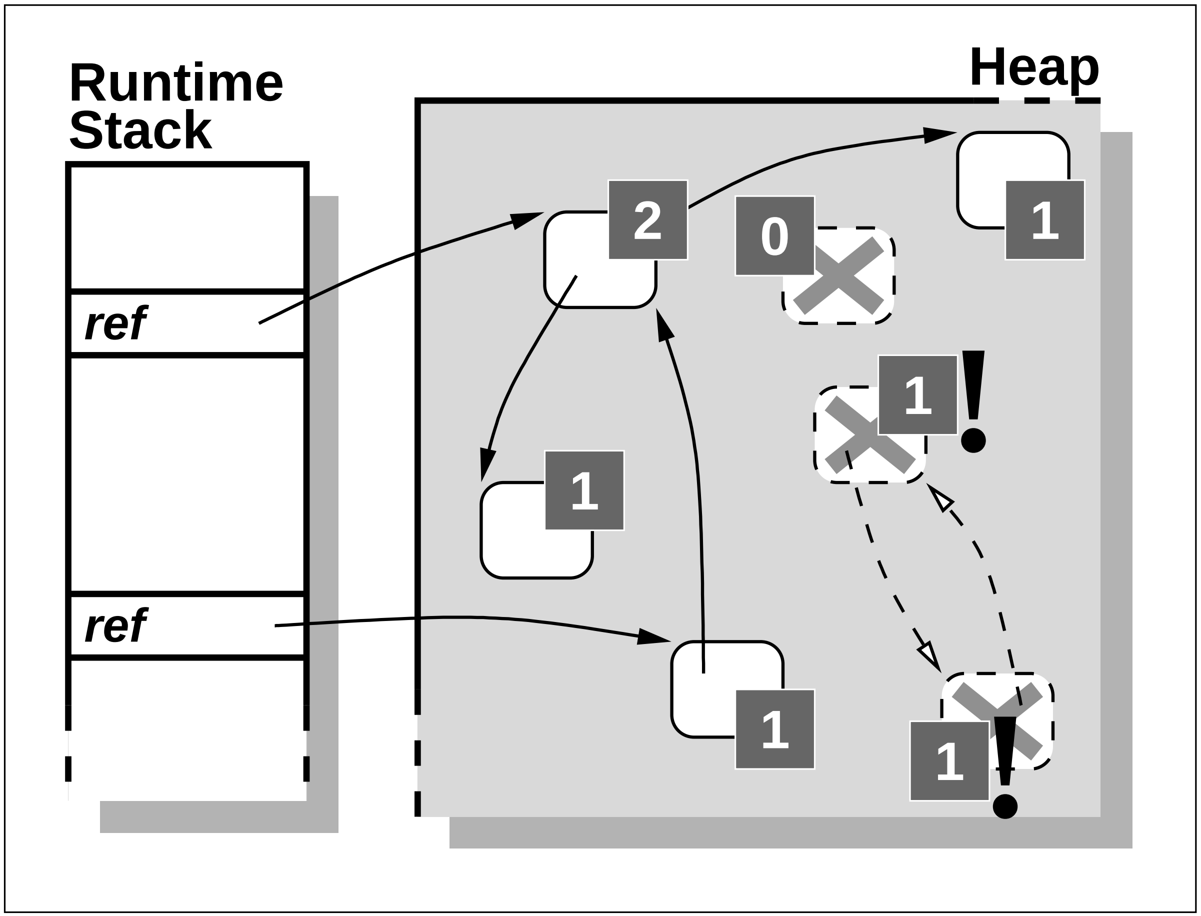
Copying GC Performance

- ✔ Copying GC is *very* fast. . .
  - . . . provided the percentage of live objects is low
  - it only visits live objects
- ✔ Compaction
  - no fragmentation and fast allocation
- ✘ Doubles space requirements though
  - only from-space used to store live objects
  - impractical for very large heaps

Reference Counting

- Keep a reference count field per object
  - increase ref count
    - when reference to that object created
  - decrease ref count
    - when reference to that object dropped
- De-allocate objects with zero ref count
  - also scan them and decrease ref counts
- Need to track all reference updates

Reference Counting Example



Reference Counting Performance

- ✘ Incomplete
  - garbage cycles
- ✘ Incremental, but *not always!*
  - last reference to the root of large data structure
- ✘ Multi-threaded issues
  - safe ref count maintenance
- ✘ Extra space requirements
- ✔ Does *not* need to visit all objects!

Advanced Reference Counting

- Background cyclic GC
- 2-bit ref counts
  - if ref count is 3, assume object live
    - don’t decrease it after that
  - when object garbage, cyclic GC will find it
- Per-thread ref count update buffers
  - process them when convenient
  - schedule de-allocations when convenient

Overview

- Introduction / GC Benefits
- Simple GC Techniques
- ➜ Incremental GC Techniques
- Generational GC Techniques
- GC in the Java HotSpot™ Virtual Machine
- GC Issues in the Real World
- Conclusions

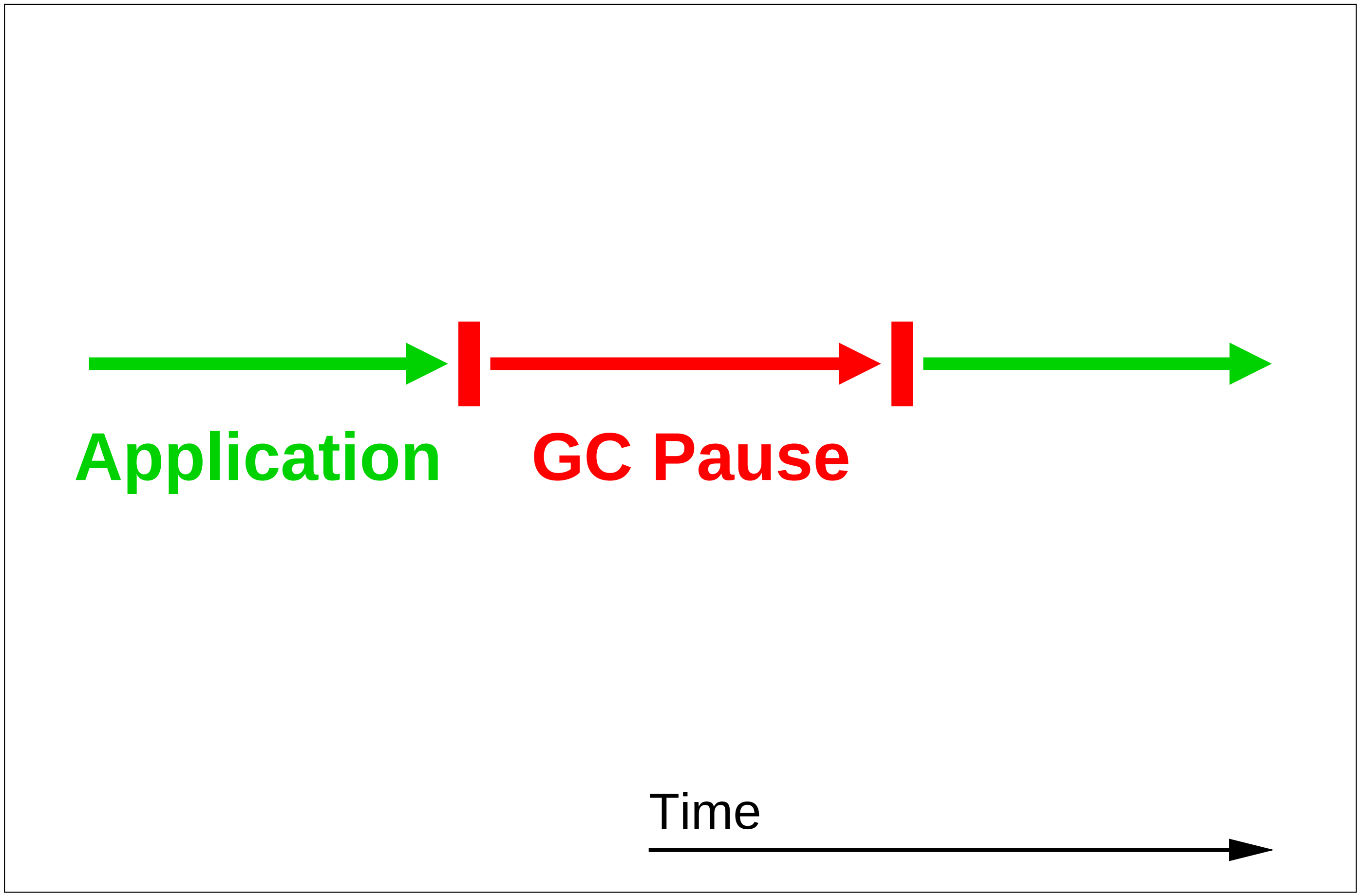
Incremental GC Techniques

- Tricolor Marking
- Boehm's Mostly-Concurrent GC
- Baker's Copying GC

Stop-The-World GC

- Mutator (application) threads stopped during GC
  - object graph frozen
  - consistent liveness information
- Heap inconsistent during object moves
  - move objects safely when mutator stopped

Serial App / Serial GC



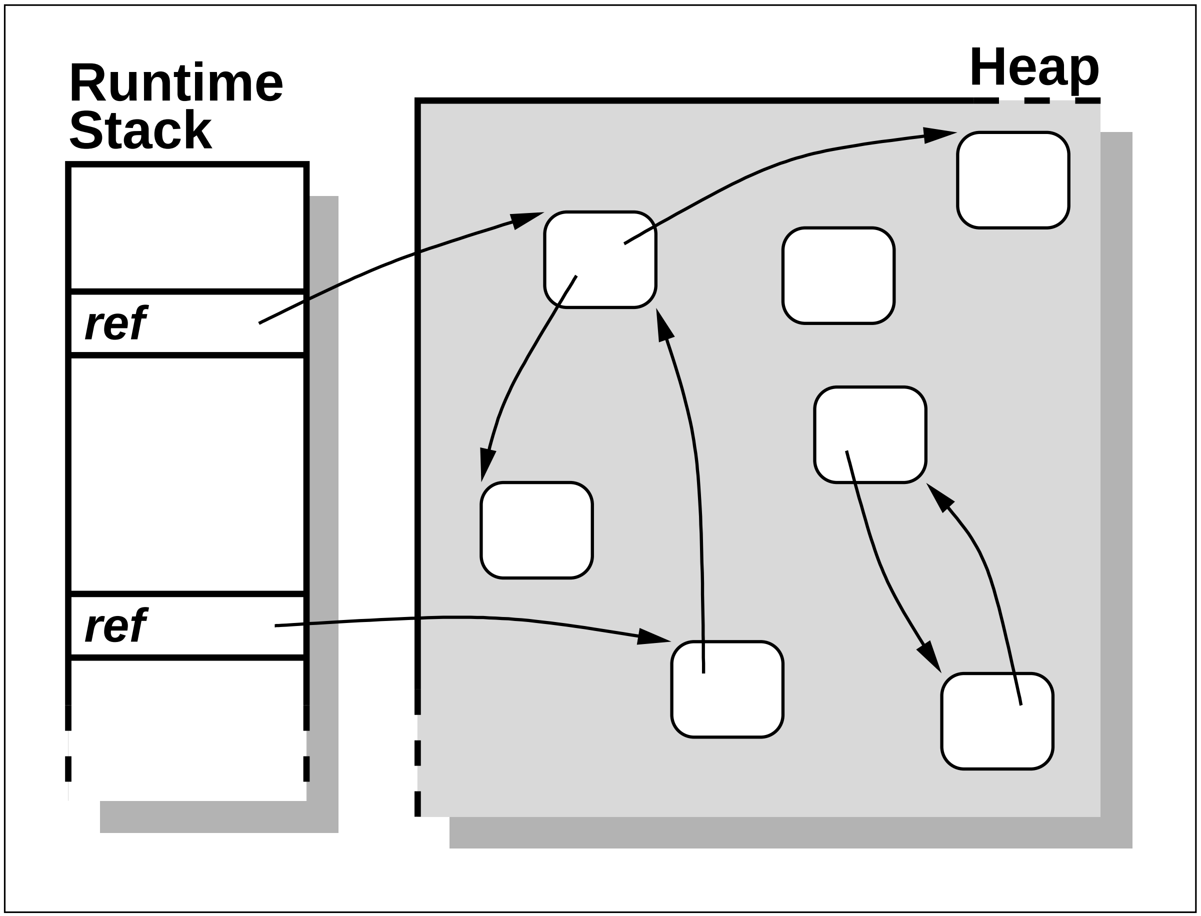
Tricolor Marking

- Invariant during liveness identification
- An object can have one of three colors
  - *White*: not marked
  - *Gray*: marked, its children not yet marked
  - *Black*: marked, all its children marked

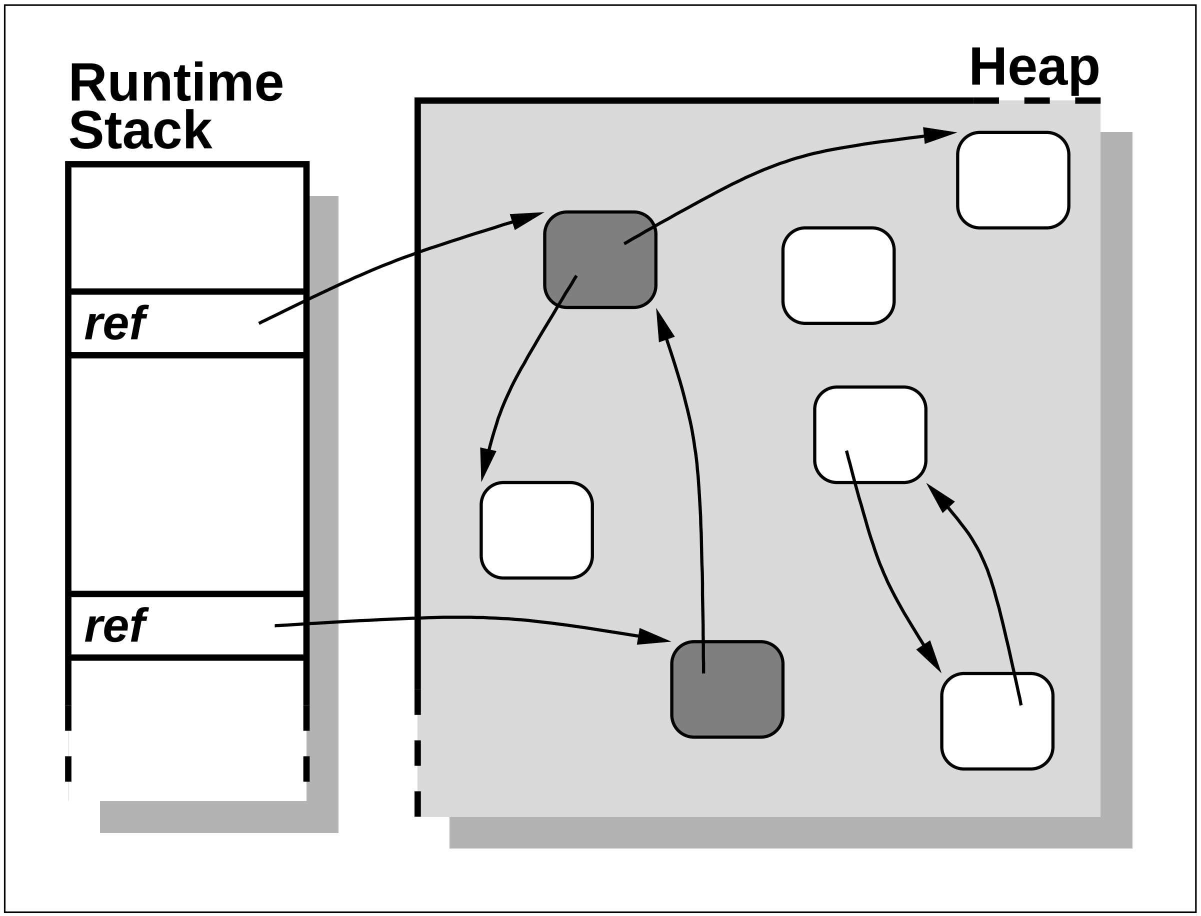
Tricolor Marking

- Start with all objects white
- Mark roots grey
- While there are gray objects
  - pick a gray object
  - mark its children gray, then mark it black
- When done, all white objects are garbage

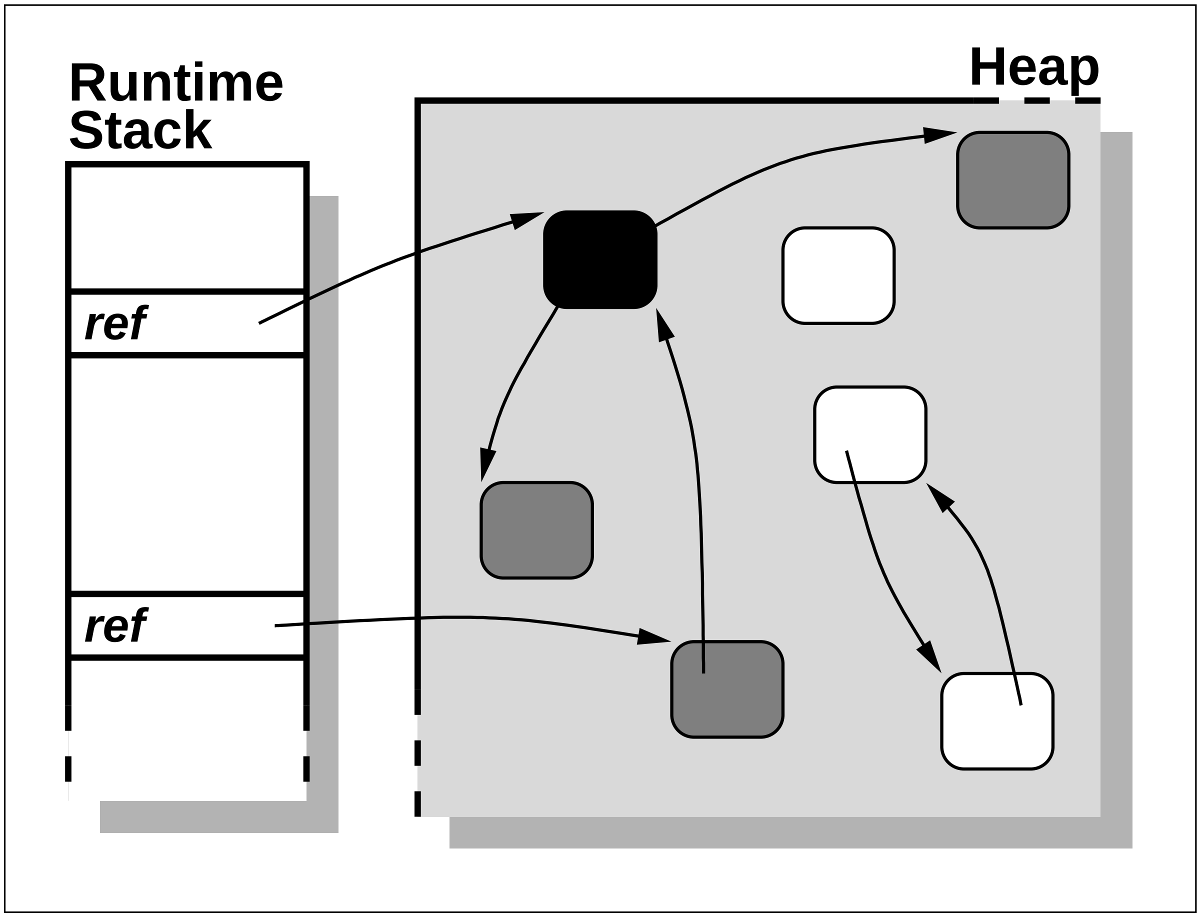
Tricolor Marking Example



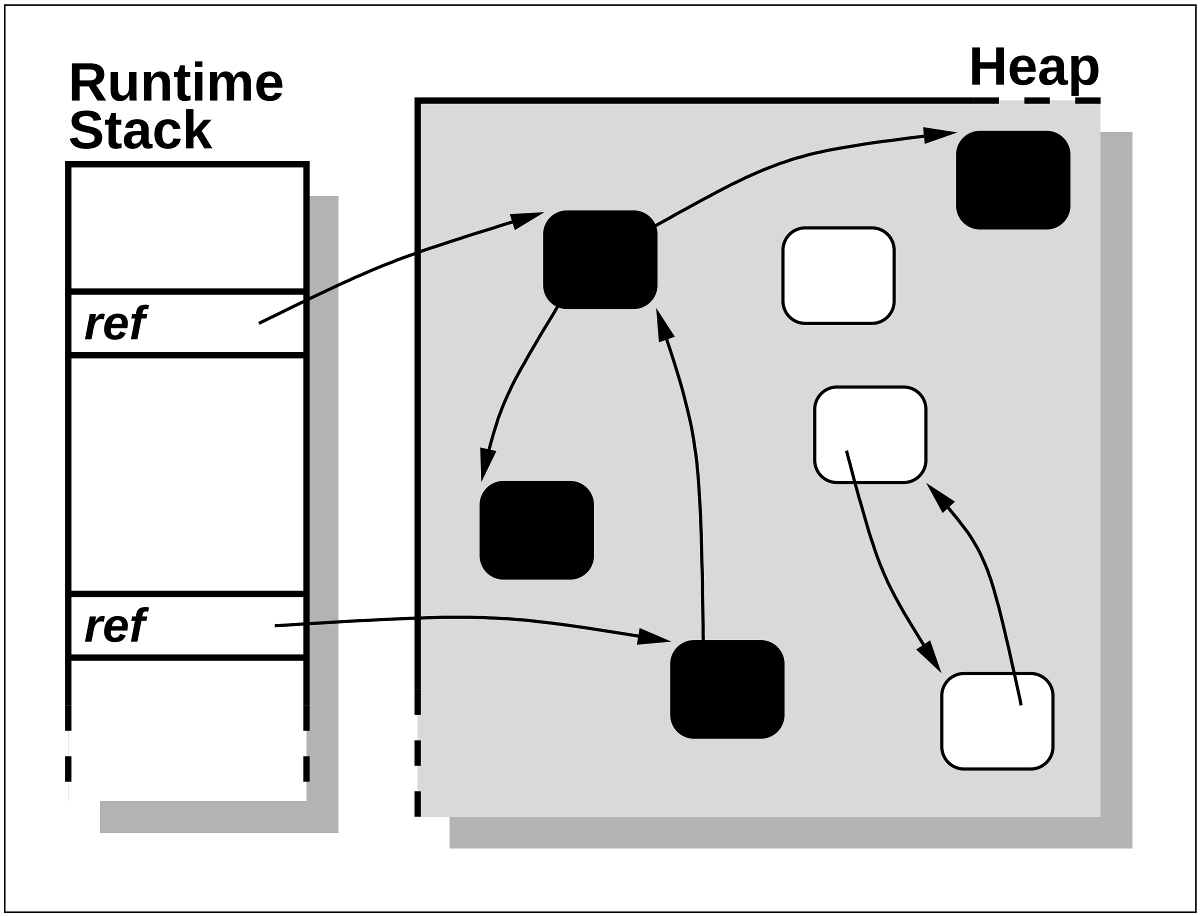
Tricolor Marking Example



Tricolor Marking Example



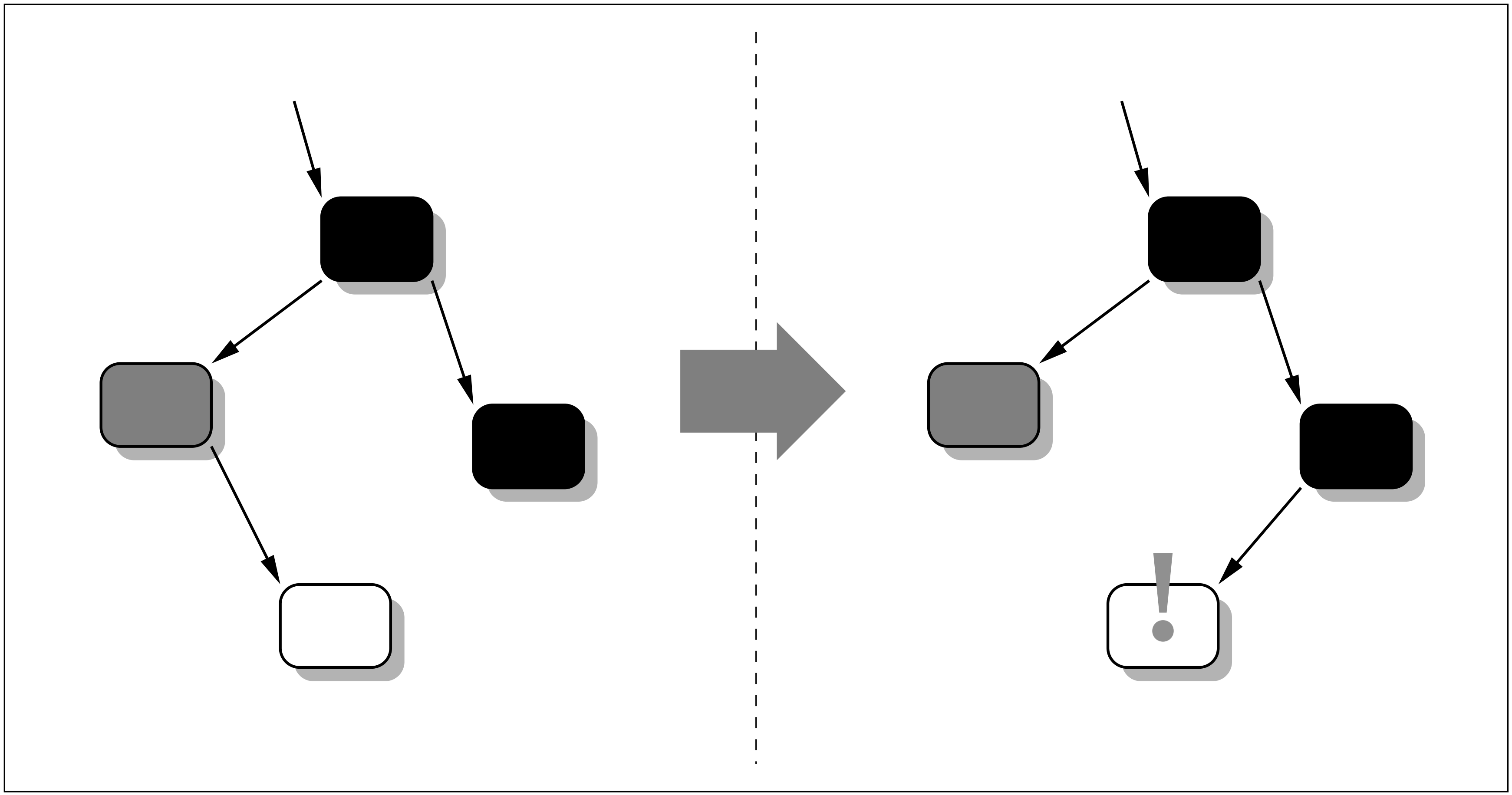
Tricolor Marking Example



Incremental GC

- Perform GC while the mutator is running
- Liveness identification is more challenging
  - mutator is changing the object graph
  - GC might not visit some live objects
  - need to synchronise GC with mutator
- The scenario we need to avoid
  - *a black object pointing to a white object*

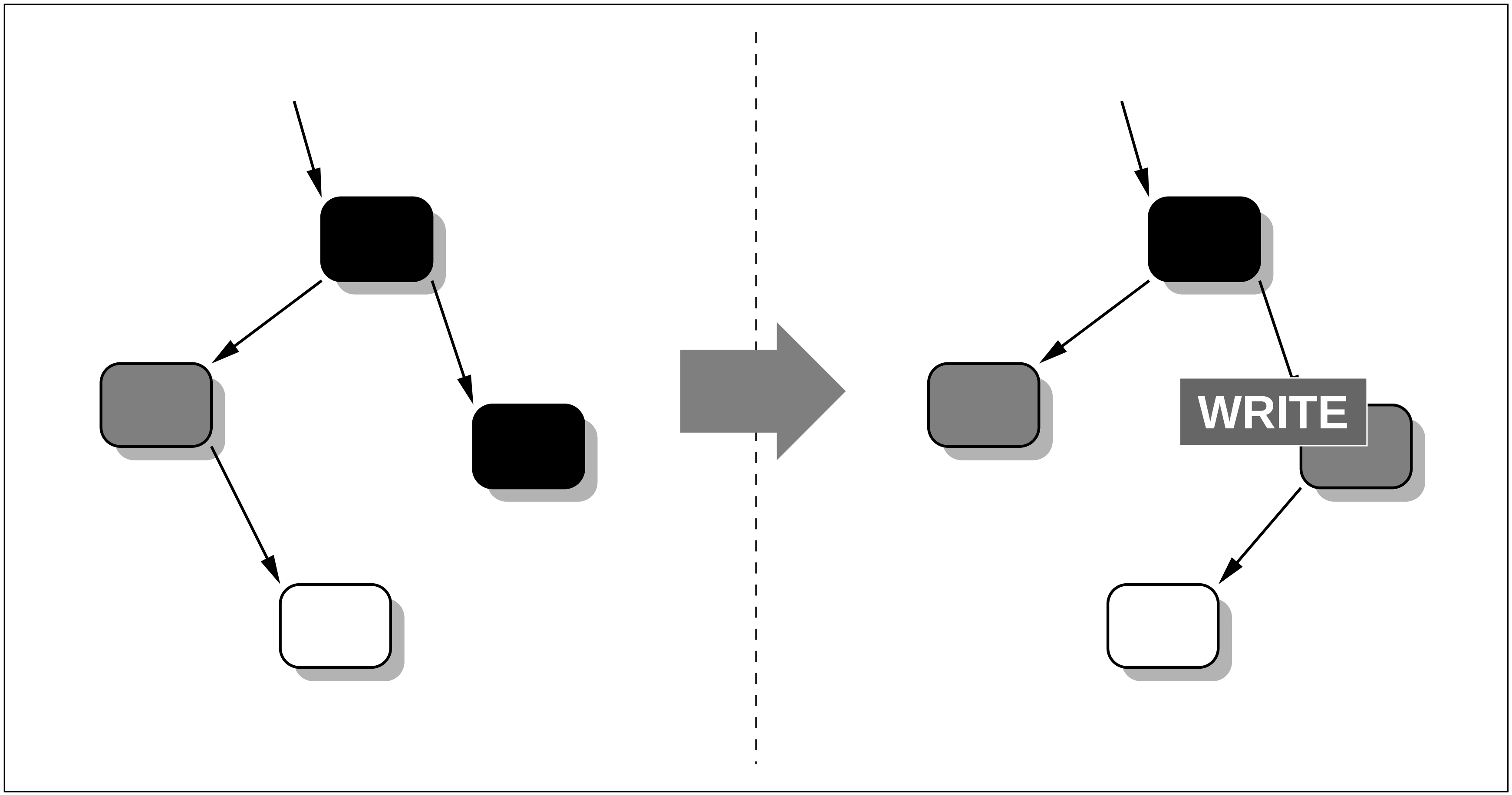
Tricolor Invariant Violation



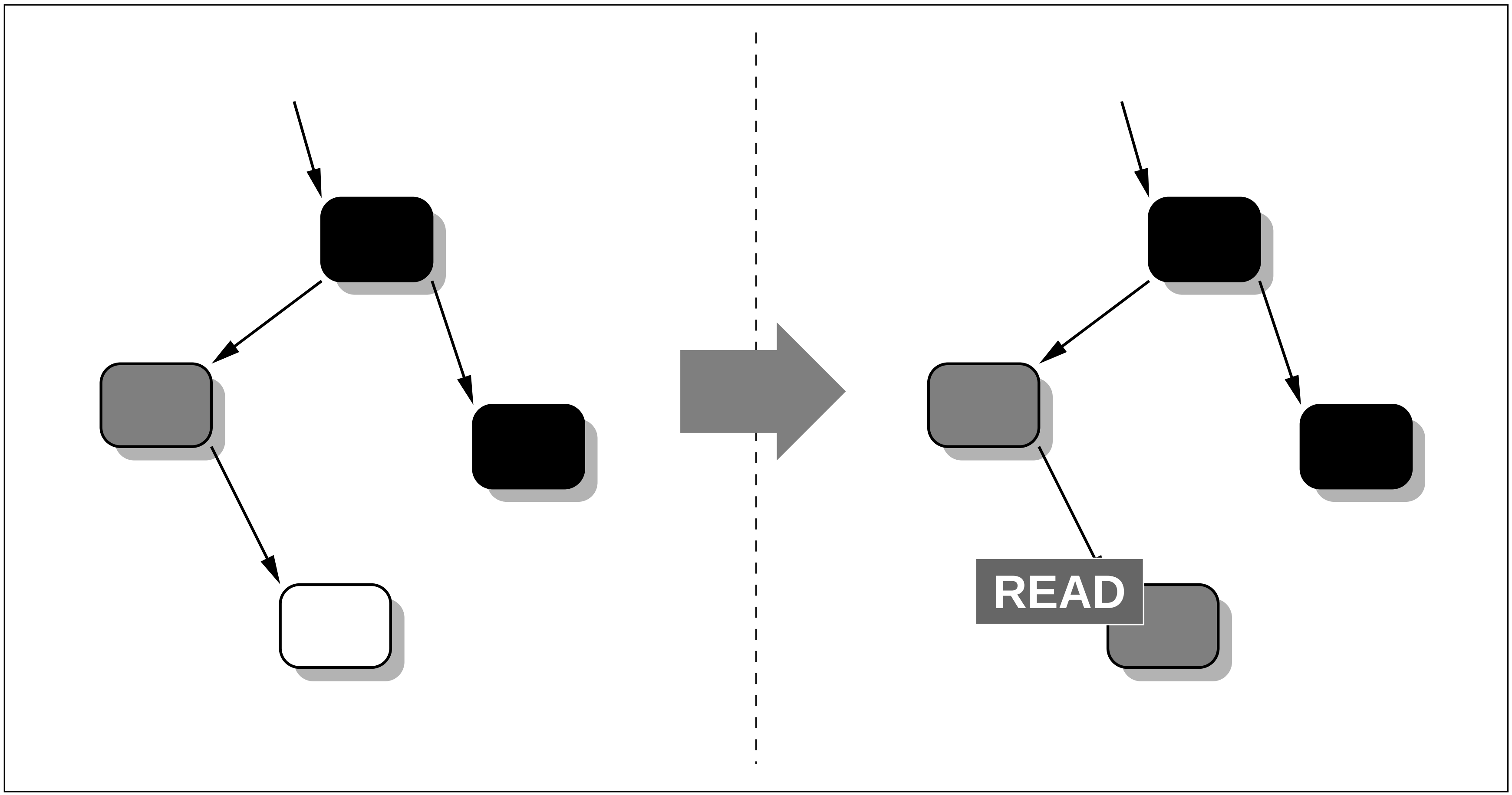
Mutator/GC Synchronisation

- To prevent the violation we can either:
- *Track writes to black objects*
  - with a write barrier
    - executed upon all reference field updates
  - mark them gray
- *Track reads from white objects*
  - with a read barrier
    - executed upon all reference field reads
  - mark them gray

Write Barrier



Read Barrier

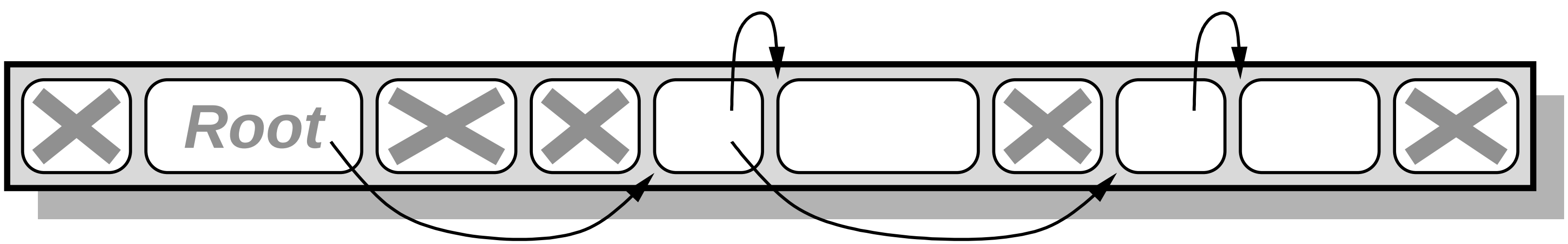


Boehm's Mostly-Concurrent GC

- Single-space Incremental Mark-Sweep GC
- GC operation
  - first stop-the-world and "checkpoint" roots
  - incrementally, mark live objects from roots
    - keep track of modified reference fields
    - write barrier (modified black objects $\rightarrow$ gray)
  - stop-the-world again and remark
    - mark from modified reference fields
  - incrementally, de-allocate garbage objects

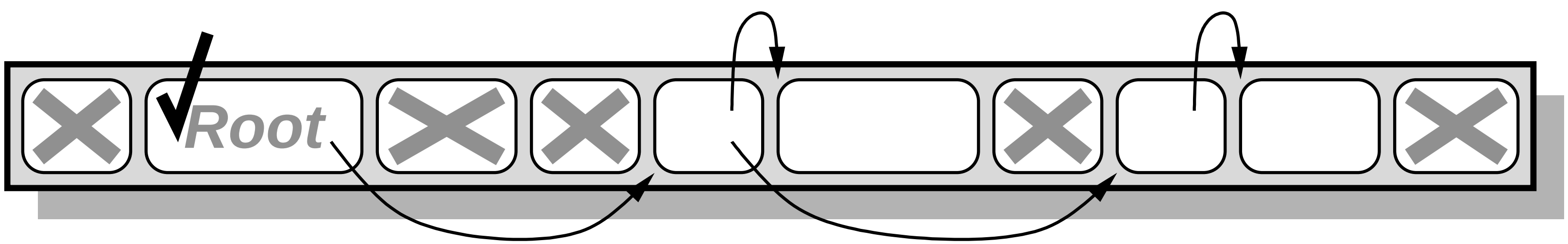
Mostly-Concurrent GC Example

0. Start of GC Cycle



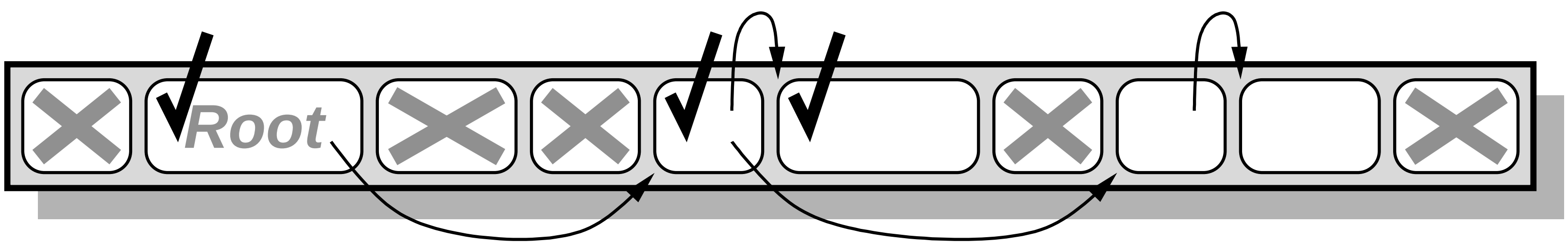
Mostly-Concurrent GC Example

1. Stop−The−World Root Checkpointing



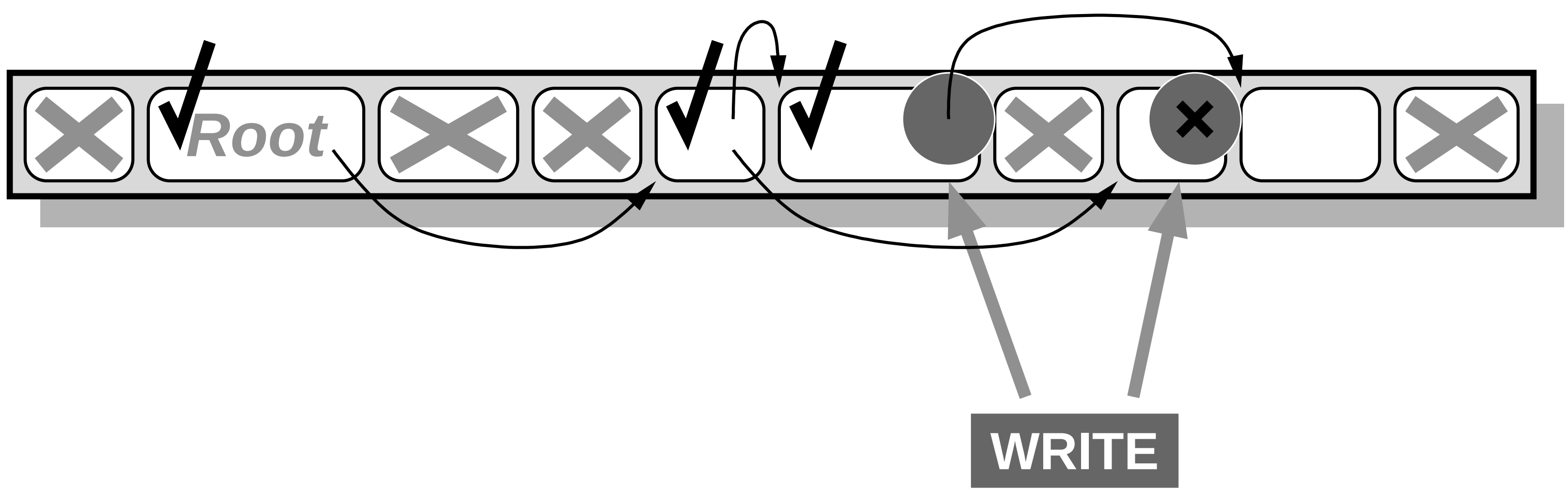
Mostly-Concurrent GC Example

2. Start of Incremental Marking Phase



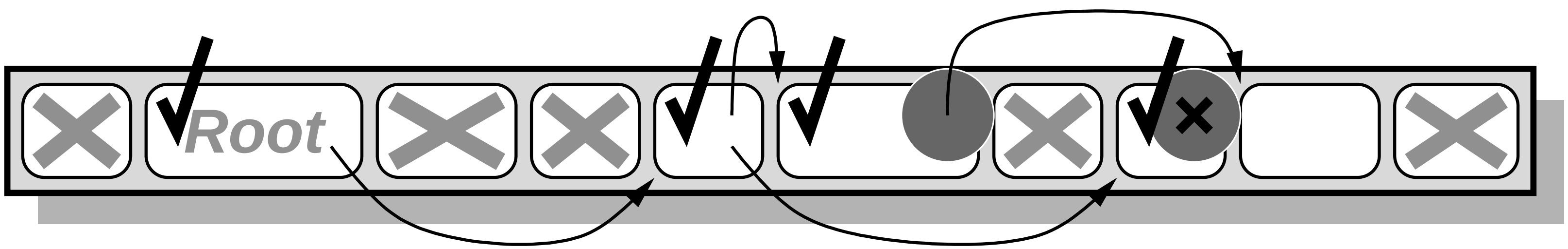
Mostly-Concurrent GC Example

3. Mutator Writes



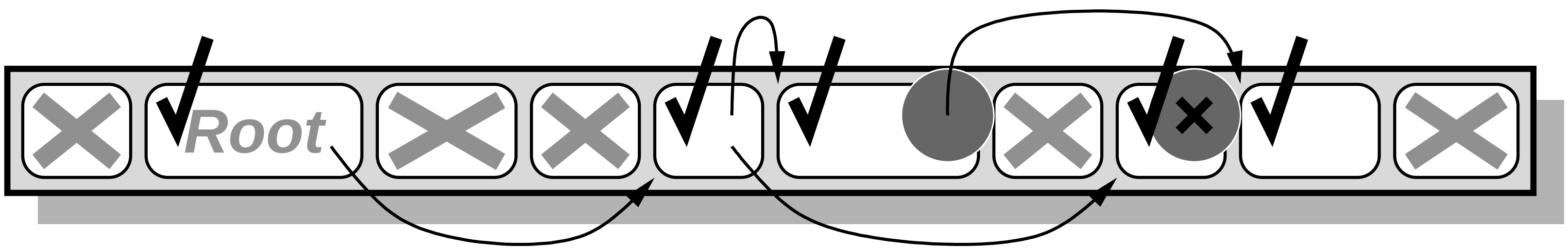
Mostly-Concurrent GC Example

4. End of Incremental Marking Phase



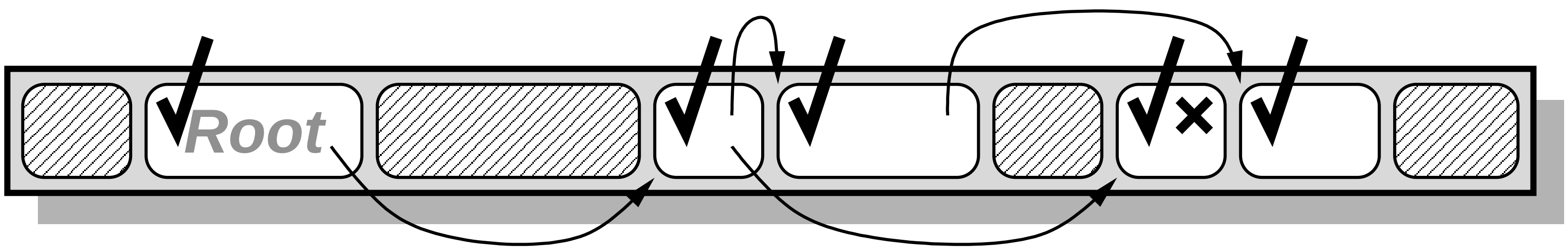
Mostly-Concurrent GC Example

5. Stop−The−World Remark



Mostly-Concurrent GC Example

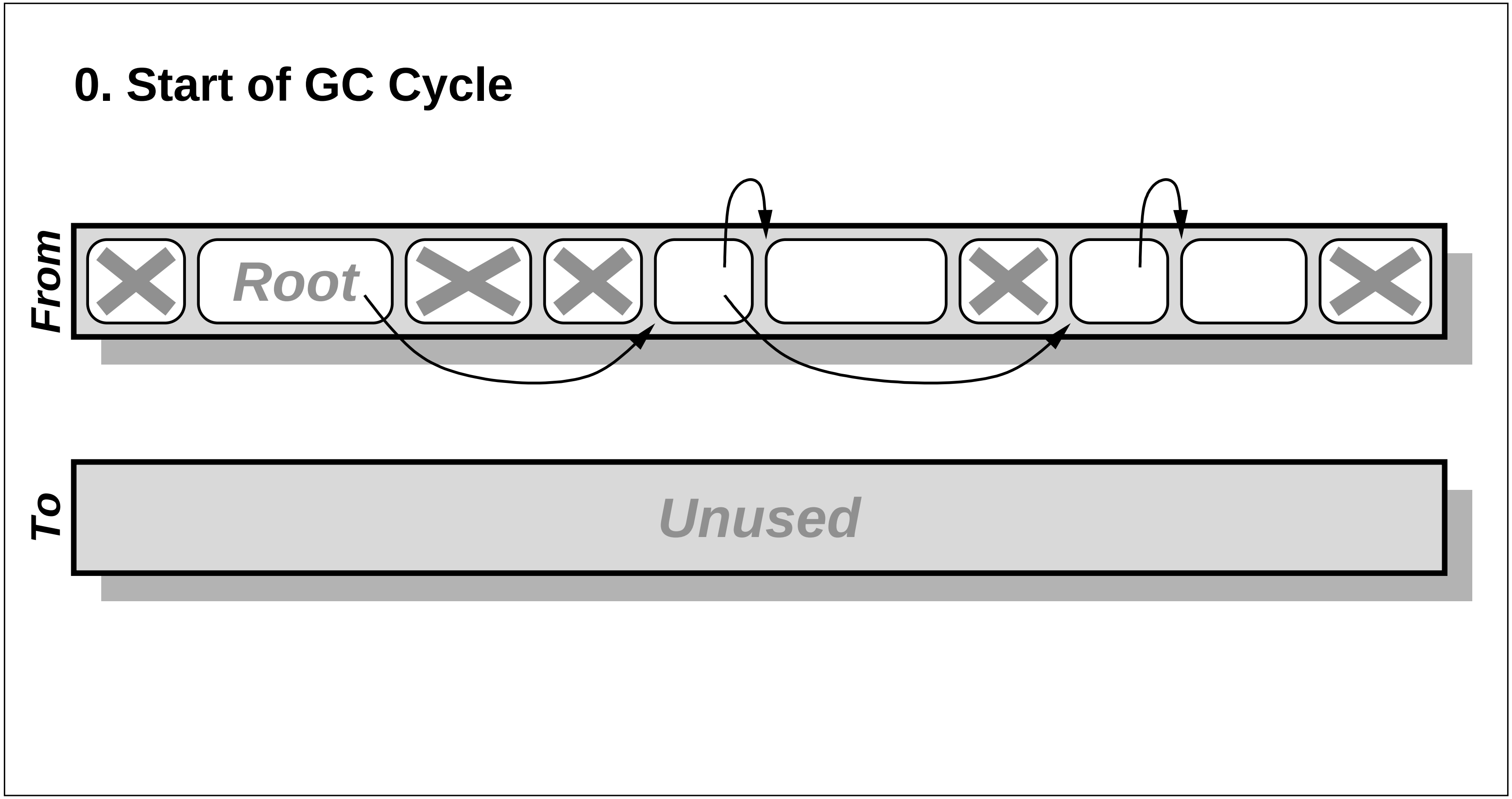
6. Incremental Sweeping Phase



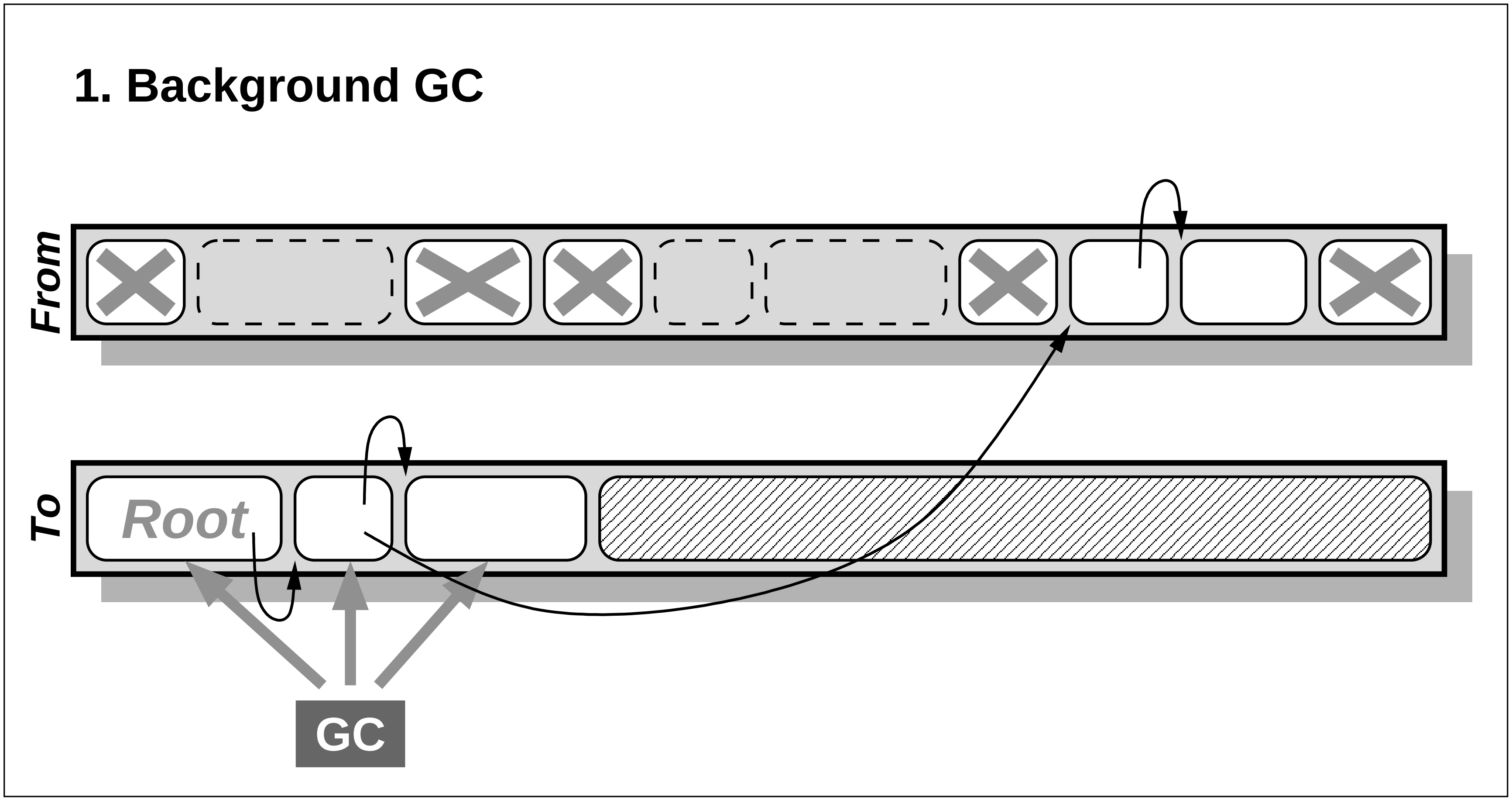
Baker's Incremental Copying GC

- Two-space Incremental Copying GC
- Mutator only accesses objects in *to-space*
- GC operation
  - background GC copies objects to to-space
  - when mutator is about to access an object in from-space
    - the object is first copied to to-space
    - read barrier (read white objects $\longrightarrow$ gray)
  - when all objects copied, swap spaces

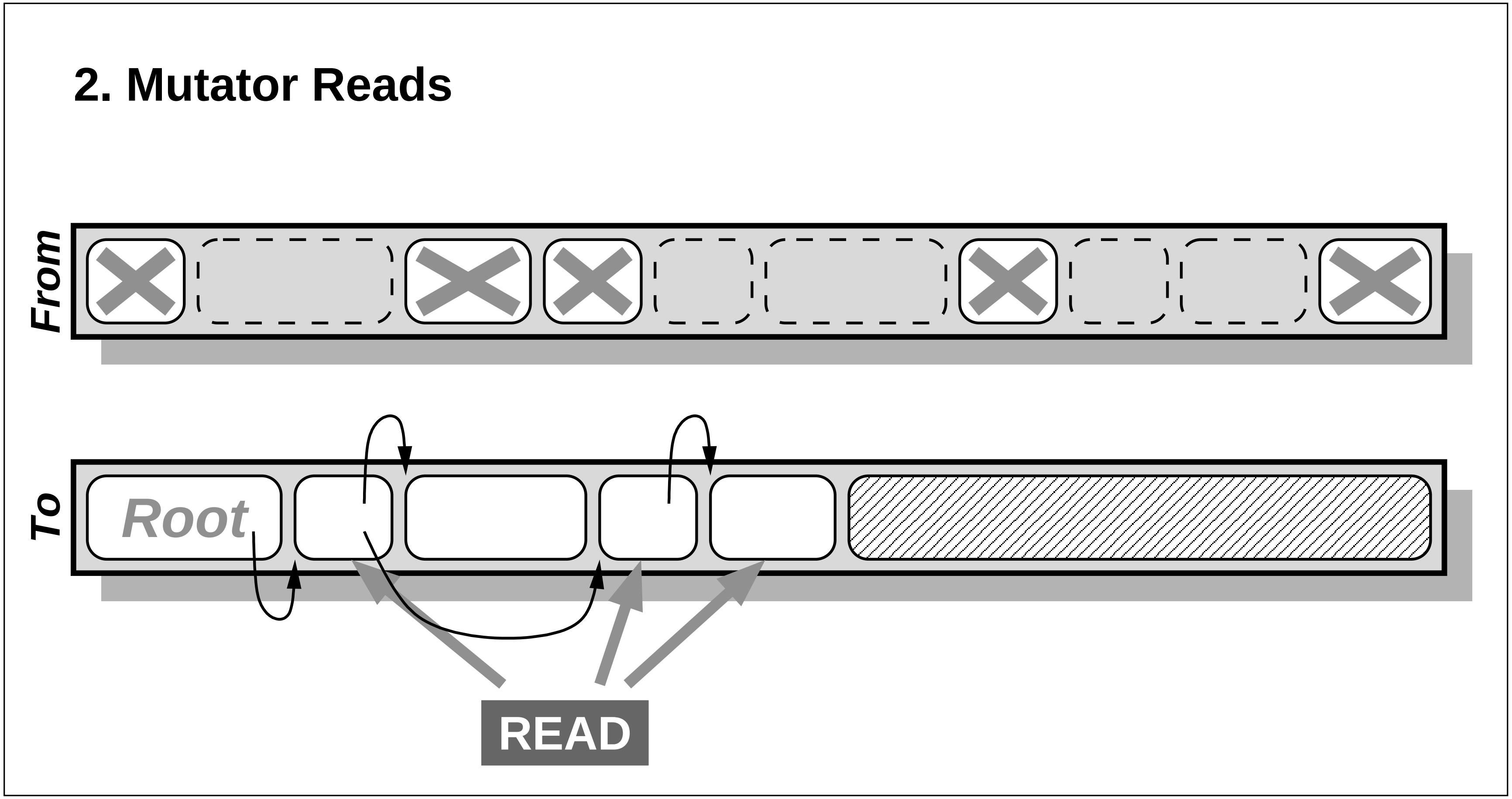
Incremental Copying GC Example



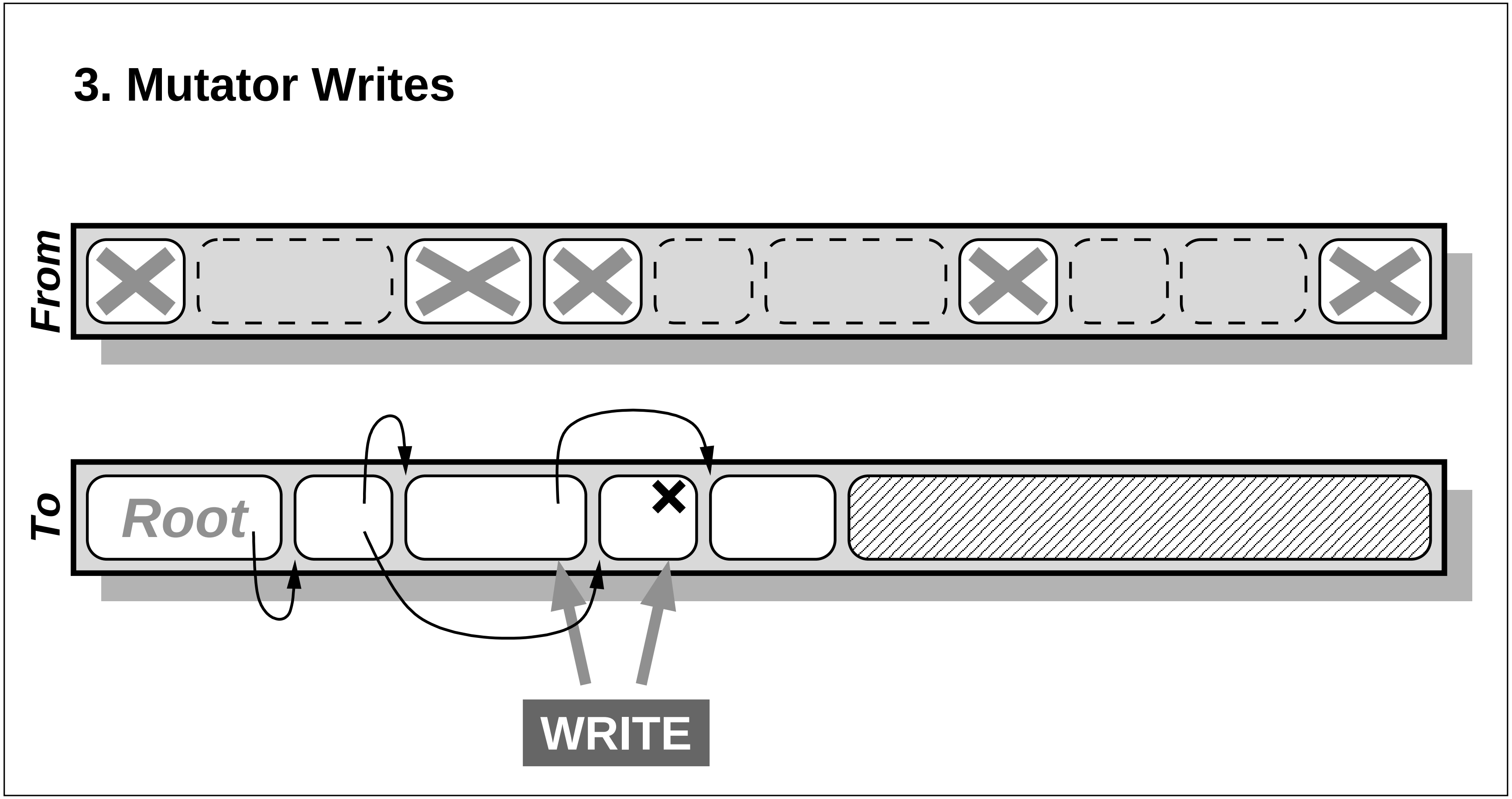
Incremental Copying GC Example



Incremental Copying GC Example

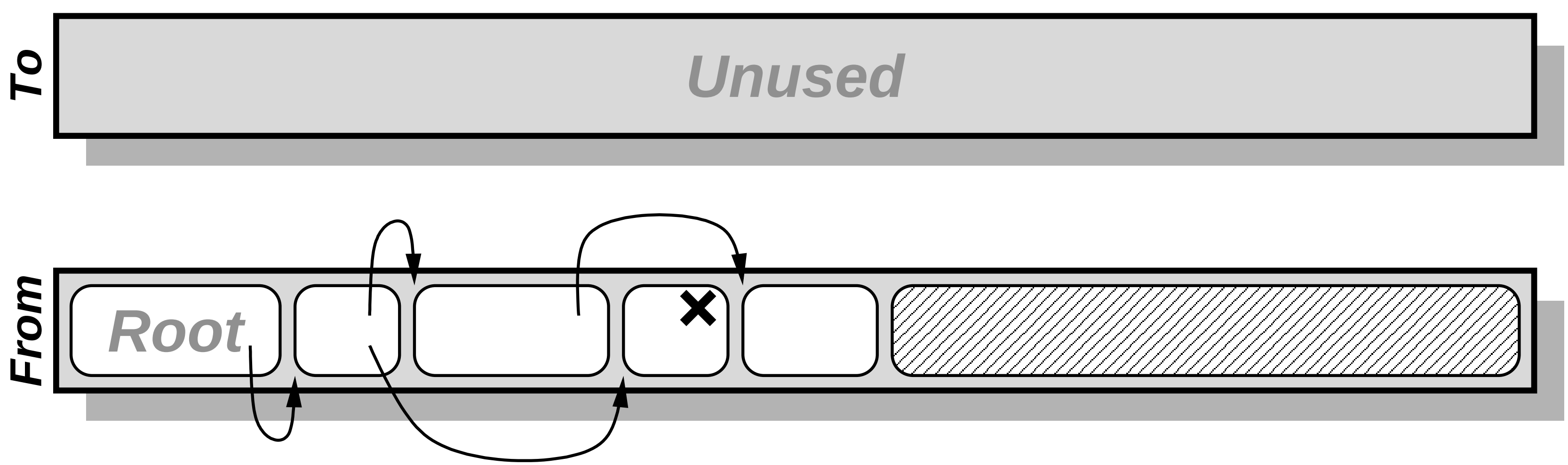


Incremental Copying GC Example



Incremental Copying GC Example

4. End of GC Cycle – Swap Spaces



Incremental GC Summary

- Pause Times
  - shorter than stop-the-world, usually!
- Memory Requirements
  - greater
  - floating garbage
- Total GC Time
  - $2\times$ or more
- Complex, hard to debug

Overview

- Introduction / GC Benefits
- Simple GC Techniques
- Incremental GC Techniques
- ➜ Generational GC Techniques
- GC in the Java HotSpot™ Virtual Machine
- GC Issues in the Real World
- Conclusions

Two Interesting Observations

- *Weak Generational Hypothesis*
  - “Most objects will die young.”
  - “Few refs in old objects point to young objects.”
  - e.g. ML, SmallTalk, mostly true for Java programs
- Can take advantage of this
  - the GC mostly concentrates on young objects
  - get more bang for your buck

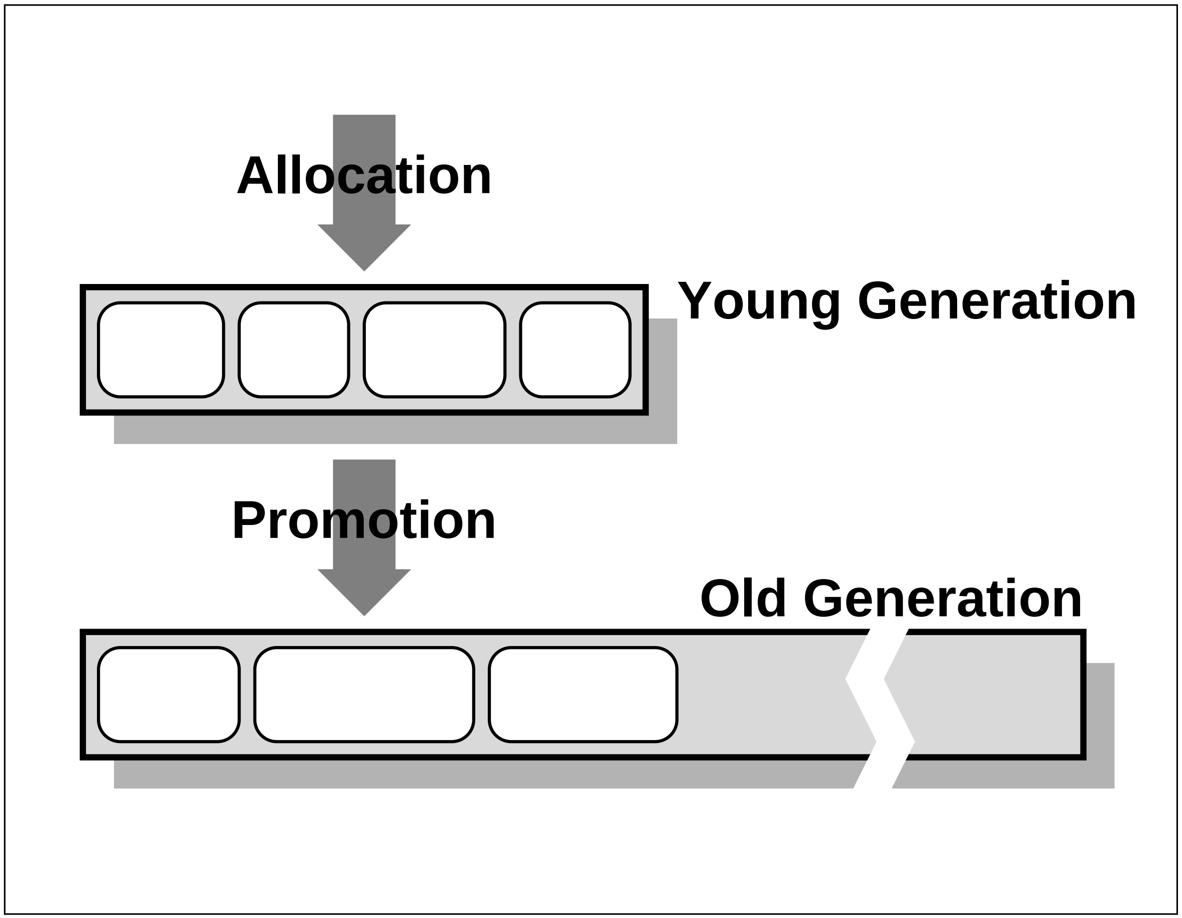
Generational GC

- Heap split into separate physical areas
  - generations
  - objects grouped according to age
  - N youngest generations GCed independently
- Need to track references
  - from older to younger generations
  - these are assumed to be infrequent!

Two-Generation GC

- Typically
  - two generations
  - young generation smaller than old generation
- *Minor Collection*
  - young generation collection, fast, frequent
- *Major Collection*
  - old generation collection, slow, infrequent

Generational GC — Promotion



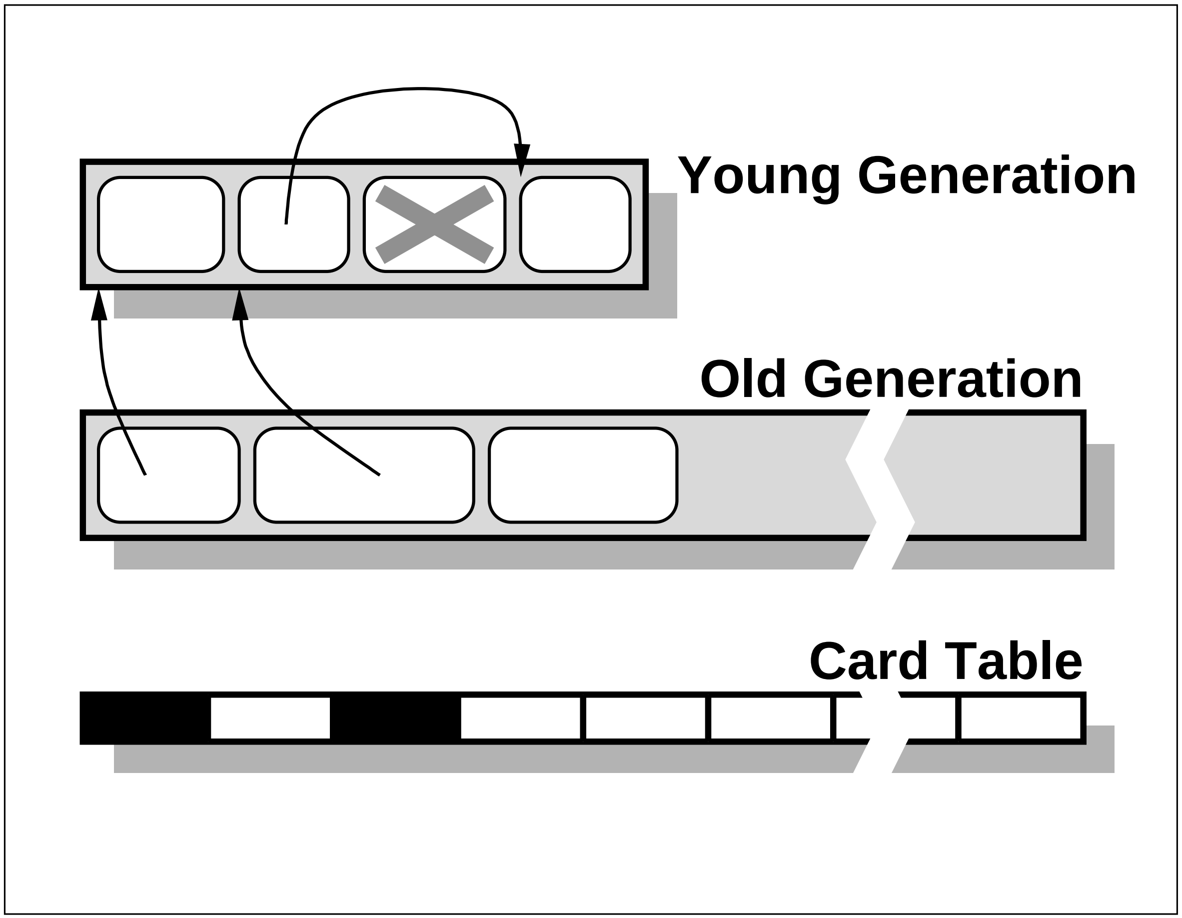
Reference Tracking

- *Write Barrier*
  - filtering / non-filtering
    - keeps track of all updated fields, or
    - only the ones that have old-to-young refs
- Most widely-used data structures
  - *Card Table*
  - *Remembered Set*

Card Table

- Heap split into small regions (cards)
  - an array (card table) has one word per card
  - cards: 0.5K–2K, *clean* / *dirty*
  - upon a reference field update
    - write barrier sets card to dirty
    - write barrier: 2–3 native instructions
- Need to scan *all* fields in *all* dirty cards
- Works well in multi-threaded environments

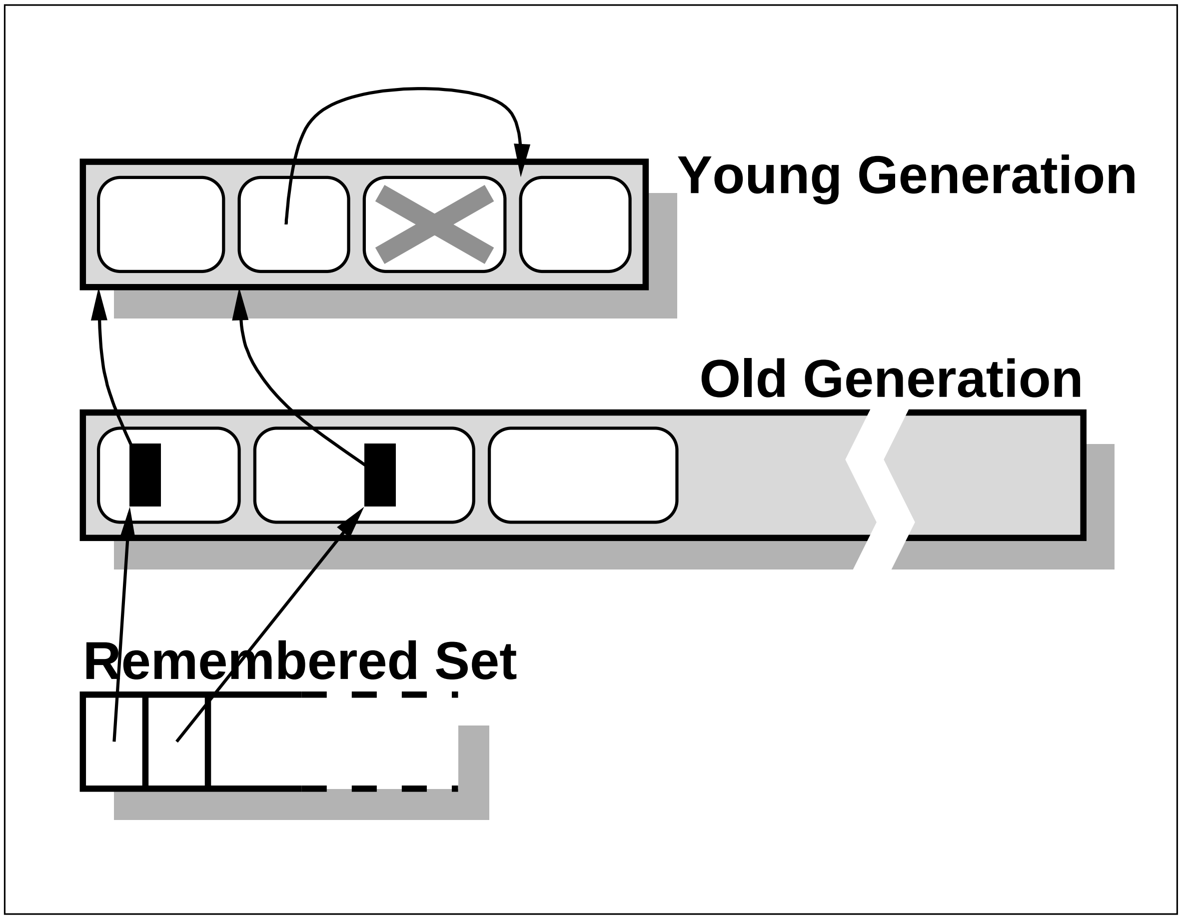
Card Table Example



Remembered Set

- Maintain a list of updated locations
  - upon a reference field update
    - write barrier adds field location to a buffer
    - can filter unwanted entries, if needed
- More accurate then card table
  - heavier-weight write barrier
  - multi-threaded issues
    - per-thread buffers

Remembered Set Example



Generational GC Summary

- Can combine different GC techniques
- Typical configuration
  - Copying GC in young generation
  - Mark-Compact GC in old generation, or
  - an Incremental GC in old generation
  - *best of both worlds!*
- More code / memory and write barrier impact . . .
- . . . but collection more effective, in most cases!

Overview

- Introduction / GC Benefits
- Simple GC Techniques
- Incremental GC Techniques
- Generational GC Techniques

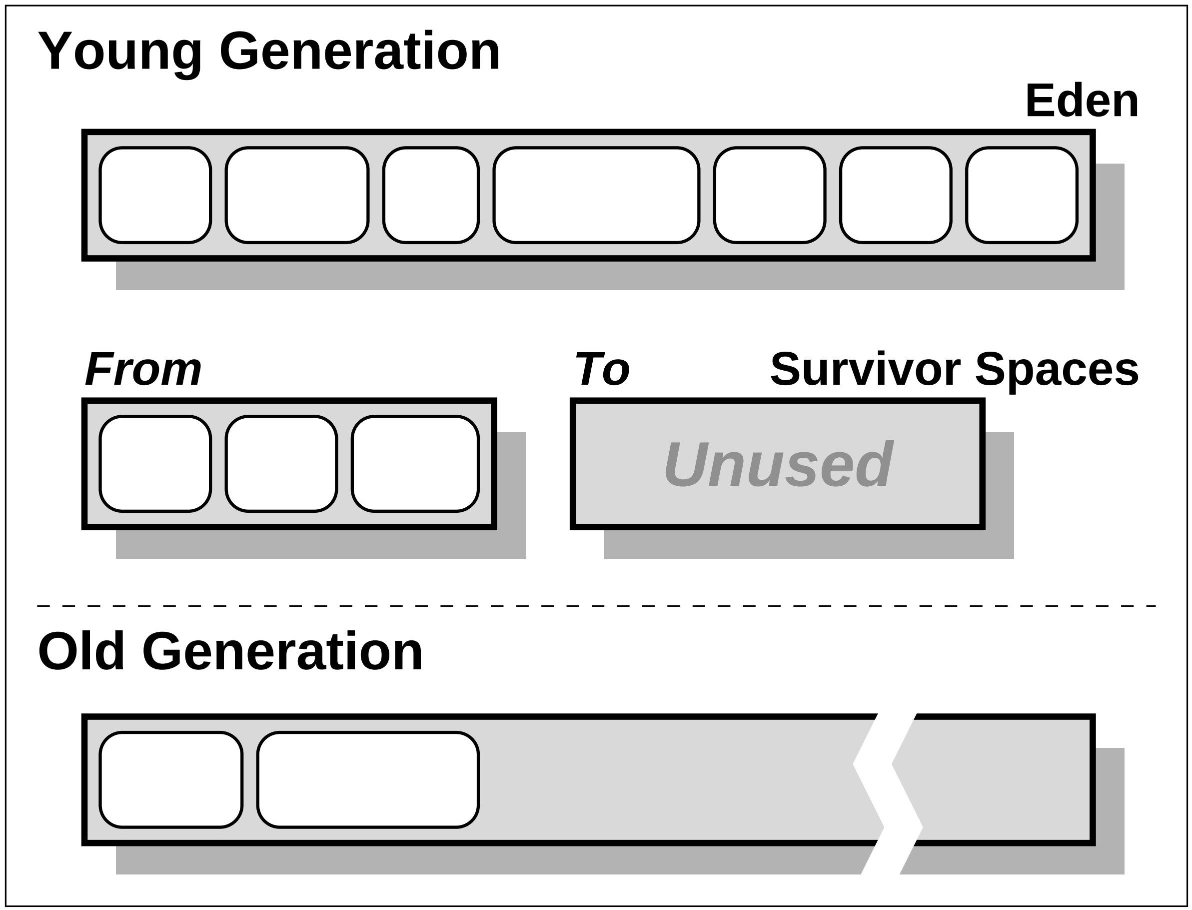
➜ GC in the Java HotSpot™ Virtual Machine

- GC Issues in the Real World
- Conclusions

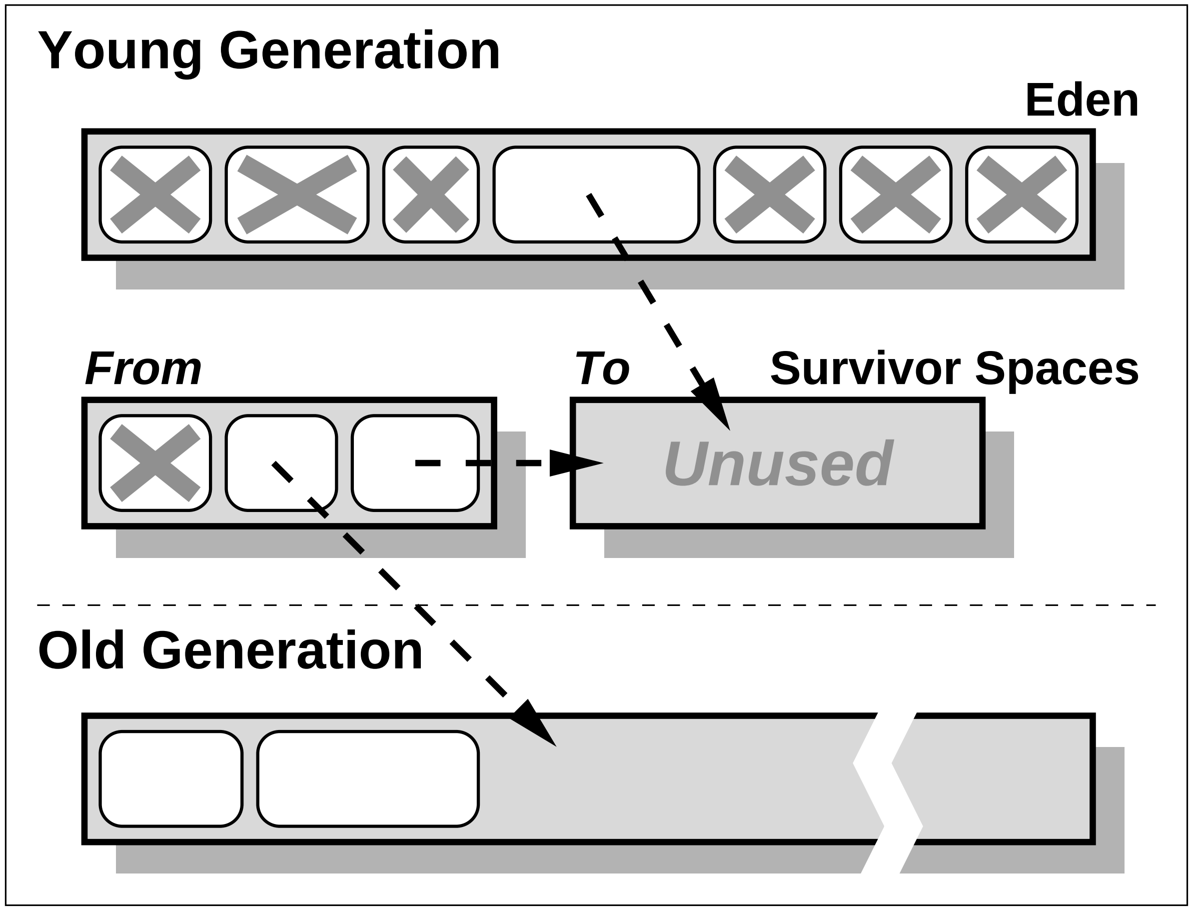
GC in the HotSpot JVM

- Exact
- Generational with Card Table
- Very, very, very fast allocation
- Stop-The-World and Mostly-Concurrent GCs
- Serial and Parallel GCs

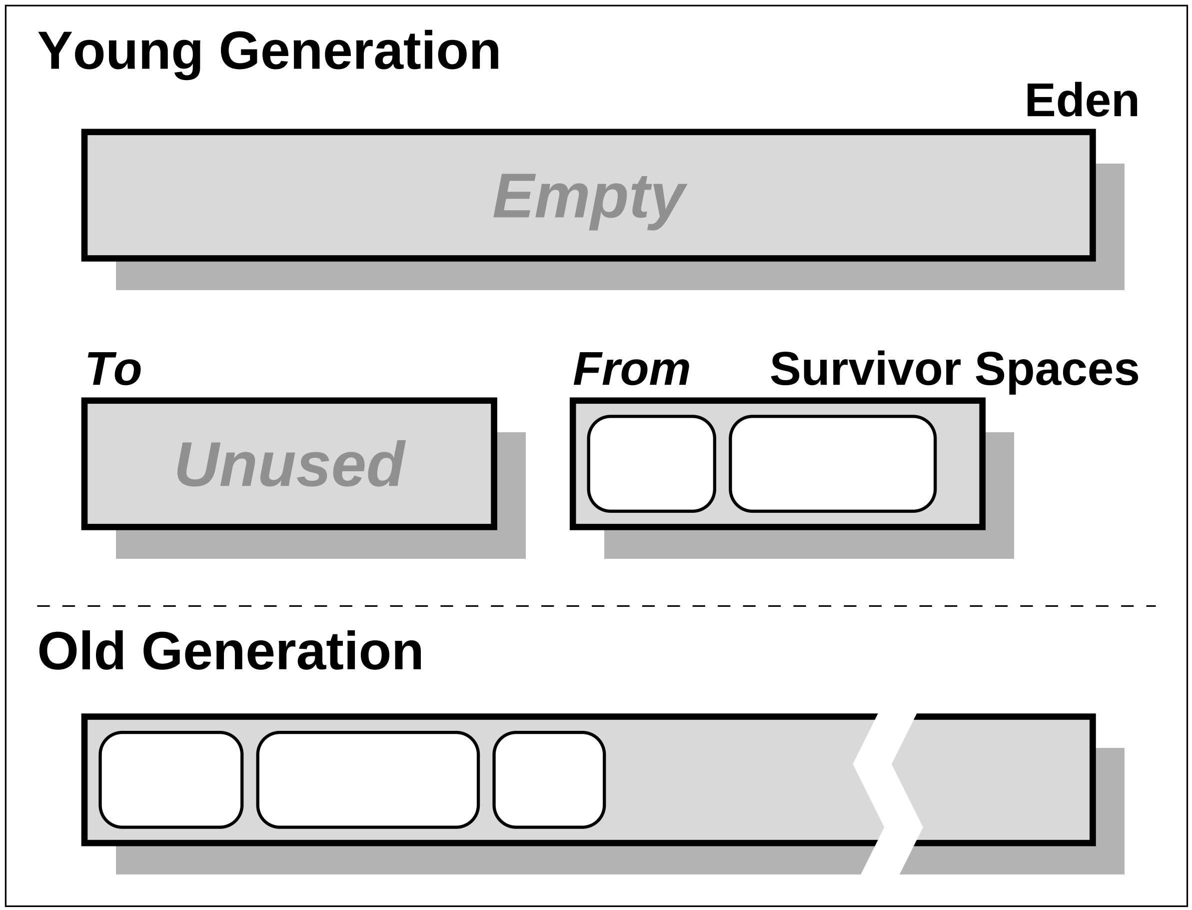
Generations in the HotSpot JVM



Before Minor GC



After Minor GC



Fast Allocation

- Eden always empty after minor GC
  - can allocate very fast into it (bump-a-pointer)
  - allocation code inlined by the JIT
  - `new Object()` is about 10 native instructions
- Multi-threaded allocation
  - thread-local allocation buffers in eden
  - no locking for most allocations!
- *Fast allocation is enabled by GC!*

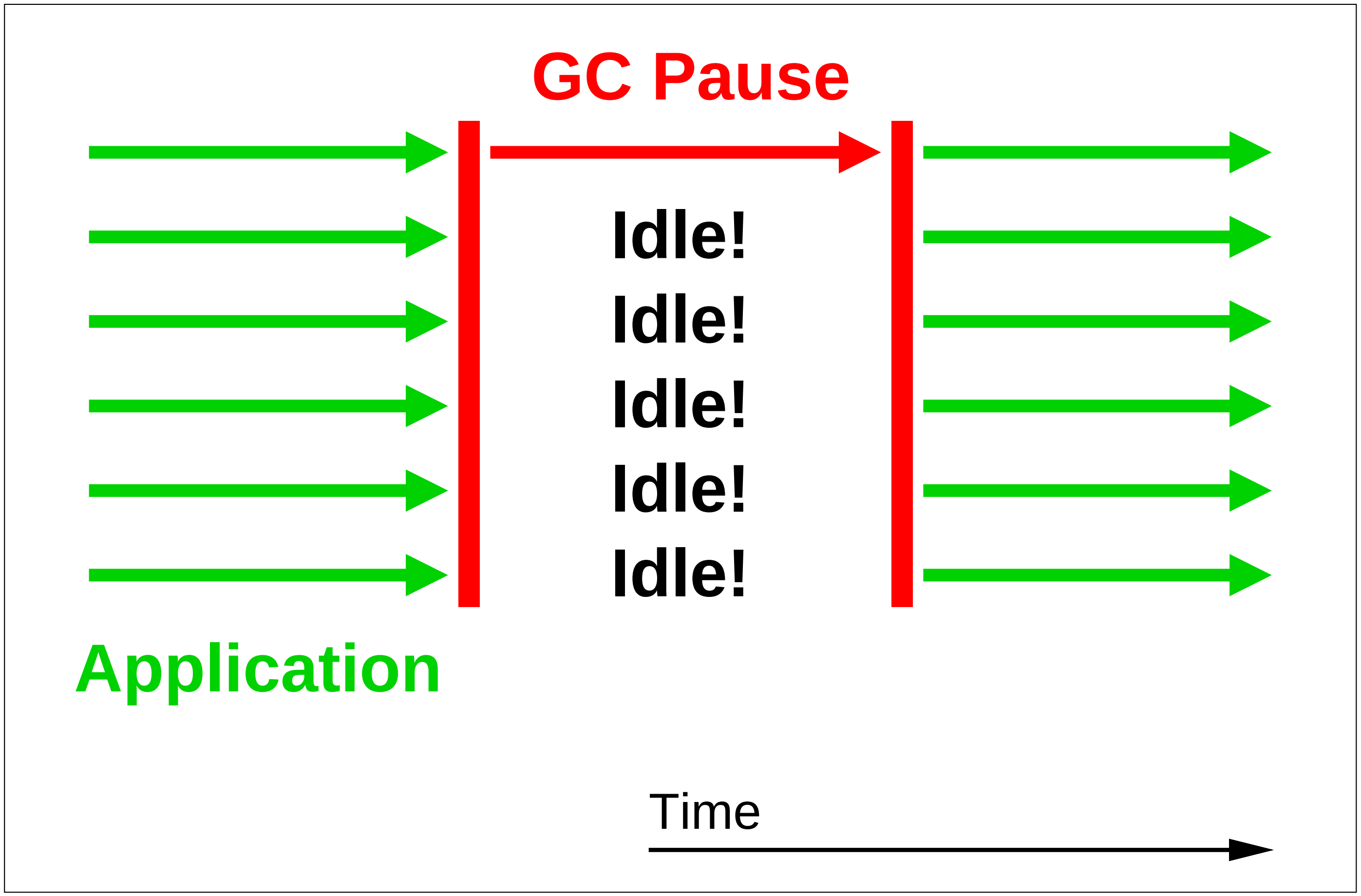
DEMO
Default GC

Java Applications

Workstation

Server

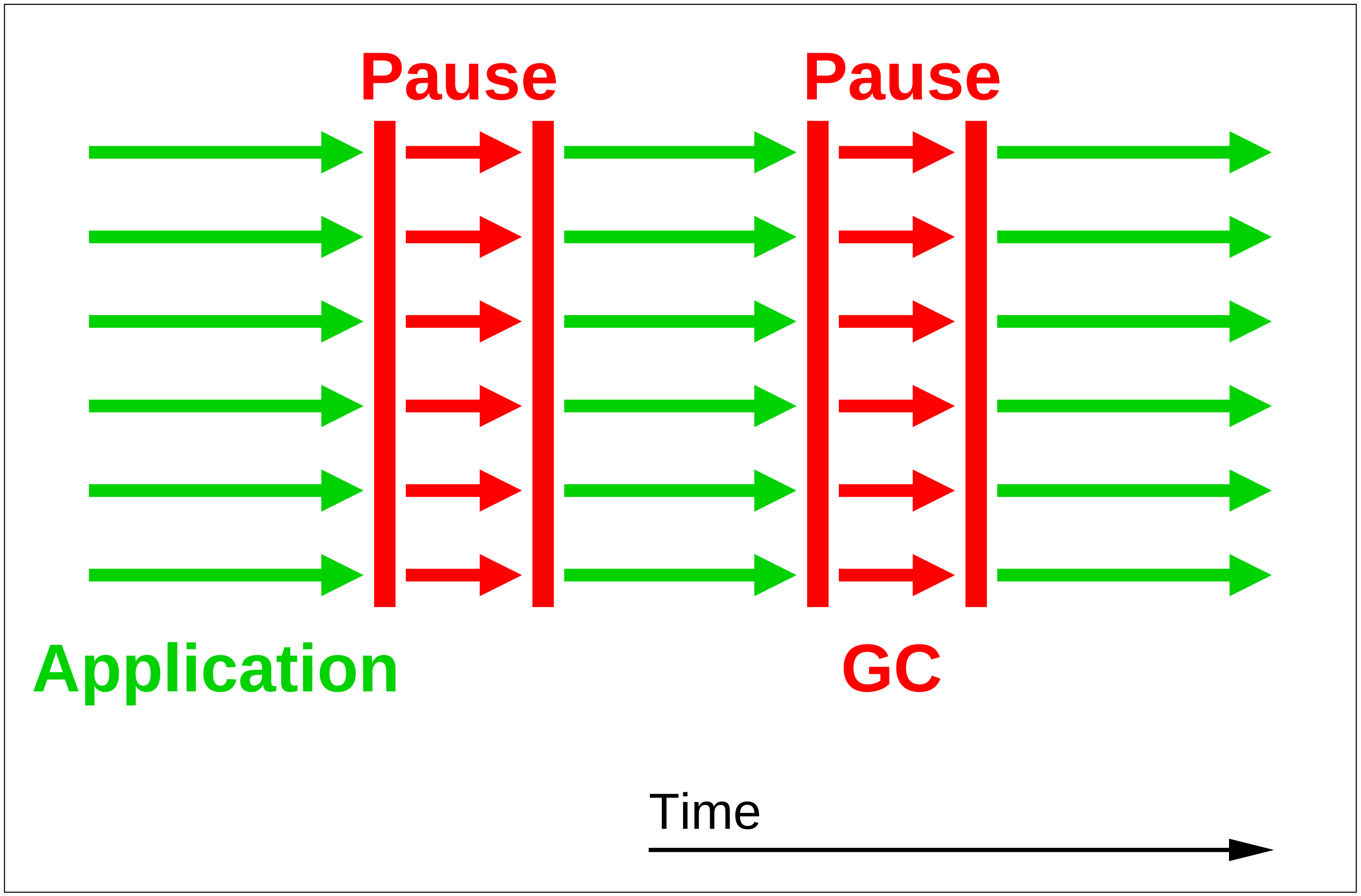
Parallel App / Serial GC



Parallel GC

- Parallel GC for young generation
  - take advantage of multiple CPUs
  - improves throughput and pause times
  - load balancing
  - allows for a larger young generation
- Old generation still done serially
- Customer quote
  "The best JVM enhancement I've seen in years!"

Parallel App / Parallel GC



DEMO
Parallel GC

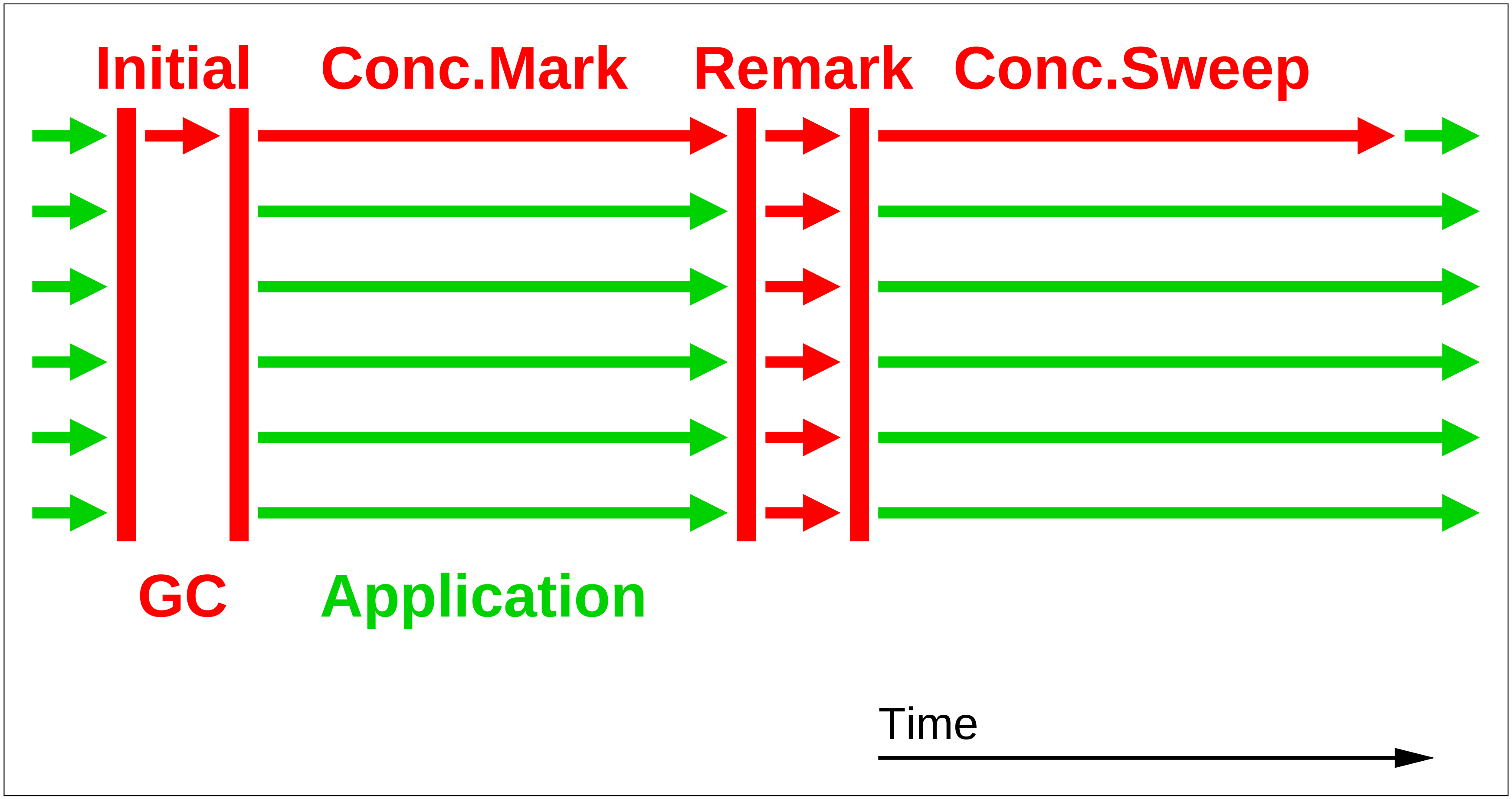
Pause-Time Issues

- ✔ *Most* GC pauses are short
  - minor collections
  - parallel young generation
- ✘ *Few* GC pauses can be long
  - major collections
  - serial
  - mainly depend on heap size
- ✘ Some applications cannot tolerate long pauses

Mostly-Concurrent GC

- Mostly-Concurrent GC for old generation
  - aka Concurrent Mark-Sweep or CMS
  - bulk of GC work concurrent (short pauses)
  - no compaction
    - in-place de-allocation
    - slower promotion, fragmentation
- Parallel young GC by default, if CPUs available

Parallel App / CMS



DEMO

Mostly-Concurrent GC

CMS Achievements

- CMS achieves
  - low GC pause times
    - 200ms–250ms possible
  - on several GB heaps
  - in combination with Parallel Young GC
- It has been successfully deployed
  - server-style telecommunications applications
- Works best for $>$1 CPU, large heaps

Overview

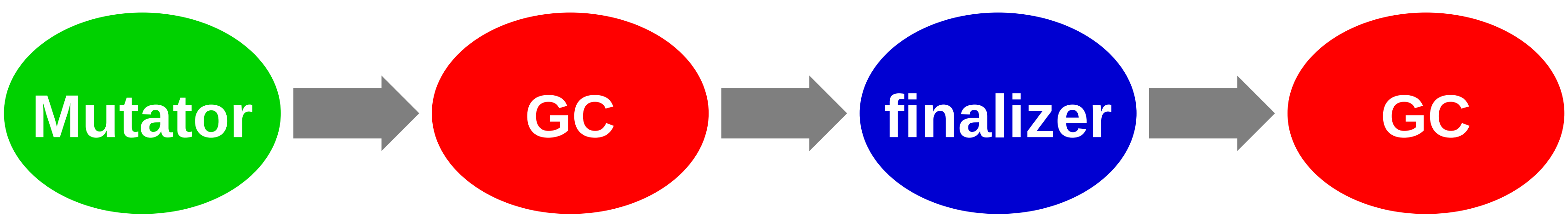
- Introduction / GC Benefits
- Simple GC Techniques
- Incremental GC Techniques
- Generational GC Techniques
- GC in the Java HotSpot™ Virtual Machine
- ➜ GC Issues in the Real World
- Conclusions

Finalization

- Cleanup hook for external resources
  - file descriptors
  - native GUI state
- Usage:
  - override `protected void finalize()`
  - at some *unspecified* time after object has become unreachable
  - `finalize()` *might* be invoked

How Does It Work?

1. An instance is registered when allocated,
2. is enqueued when it becomes unreachable,
3. has its `finalize()` method invoked,
4. becomes unreachable again,
5. then, finally, has its storage reclaimed.



Finalization Impact

- ✘ Execution speed
  - slower allocation
  - finalizer thread affects scheduling
- ✘ Heap size
  - memory retained longer
- ✘ Collection pauses
  - longer
  - discovery and queuing

Finalization Suggestions

- Use for cleanup of *external* resources
- Limit the number of finalizable objects
- Reorganise classes
  - finalizable object holds no extra data
- Beware when extending finalizable classes
  - in standard libraries (e.g. GUI elements)
- Use one of the `java.lang.ref` reference objects instead

Object Pools

- Manual memory management
  - allocation *serialised*
  - current JVMs support fast, parallel allocation
- Data is kept artificially alive
  - adds pressure on garbage collector
- Breaks down abstract data types
  - who is responsible for the instances?
- Use if object initialization is *really* expensive

Object Pool Example

```
class Node {
    private static Node head = null; private Node next;
    public static synchronized Node allocate() {
        if (head == null) return new Node();
        Node result = head; head = head. next; return result;
    }
    public static synchronized void free(Node n) {
        n. next = head; head = n;
    }
    …
}
```

Real Customer Problem

- Object pools never truncated
  - peak live data ∼300MB
  - average live data ∼100MB
- Problem
  - other garbage generated from libraries
  - GCs less frequent, but dealt with 300MB
- Solution
  - removed object pools; application ran faster!

Avoid Frequent Bad Habits

- Size heap appropriately
  - maximum should be larger than working set
  - leave room for the system to adapt
- Avoid `java.lang.System.gc()`
  - especially when using CMS!
- *Consider* setting references to `null` early
  - large objects in particular

GCs Have Bad Habits Too. . .

- ✔ The *great* thing about using a GC is that

  It does everything automatically, behind your back!

- ✘ The *bad* thing about using a GC is that

GCs Have Bad Habits Too. . .

- ✔ The *great* thing about using a GC is that

  It does everything automatically, behind your back!

- ✘ The *bad* thing about using a GC is that

  It does everything automatically, behind your back!

Problem Hunting

- Most of the time
  - GC helps the programmer avoid problems
- When things do go wrong though
  - problems very hard to track down
  - lack of feedback from the GC
    - everything is automatic, remember?
  - e.g. the return of the memory leaks

Tools

- *Needed*
  - due to the implicit nature of GC
- Current ones too expensive to use in deployment environments
- JVMTI
  - Java Virtual Machine Tool Interface
  - for development and monitoring tools
  - JSR 169

Overview

- Introduction / GC Benefits
- Simple GC Techniques
- Incremental GC Techniques
- Generational GC Techniques
- GC in the Java HotSpot™ Virtual Machine
- GC Issues in the Real World

➜ Conclusions

Conclusions

- Several types of GC
  - serial, parallel, concurrent, . . .
  - each suited to a subset of applications
  - the HotSpot JVM provides choices
  - choose the one appropriate for *you*
- GC simplifies Java programs
  - but developers should learn how to use it!

Bibliography

R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996. With a chapter on Distributed Garbage Collection by R. Lins.

P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the First International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St Malo, France, September 1992. Springer-Verlag.

QUESTIONS?

tony.printezis@sun.com