In search of …

verifiable…

Australian National University

DATA 61 CSIRO

veri~~fi~~able…

verified…

on-the-fly…

Concurrent Garbage Collectors

..on modern processors

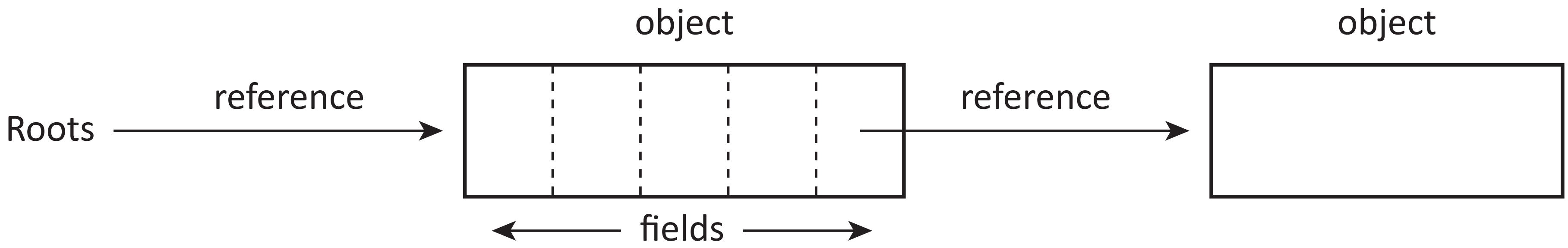
In search of verified on-the-fly Concurrent Garbage Collection on modern processors

Tony Hosking

Terminology

- The *heap* is a contiguous array of memory words *or* a set of discontiguous blocks of contiguous words
- A *cell* is a contiguous group of words that may be *allocated* or *free*, or *wasted* or *unusable*

Terminology



- An *object* is a *cell* allocated for use by the application, divided into *fields*
- A *reference* is either a pointer to a heap object or the distinguished value `null`
- A field may contain a *reference* or some other *scalar* non-reference value
- Program *roots* are the local/global variables of a program that hold references

Terminology

- The *mutator* executes application code, which *allocates* new objects and *mutates* references by modifying fields and roots
- Thus, objects can become disconnected from the roots, and become *unreachable*
- The *collector* executes garbage collection code, which discovers unreachable objects and *frees* them, allowing their storage to be *reallocated*

Example

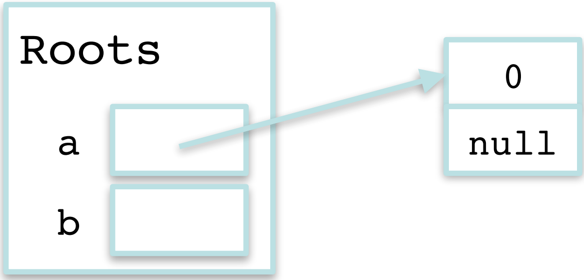
Roots

a	null
b	null

Mutator

```
type Cell = record
  value: int;
  next: ref Cell;
end;
var a, b: ref Cell := null;
a := new Cell(0, null);
b := new Cell(1, null);
a.next := b;
b.next := new Cell(2, null);
a := null;
```

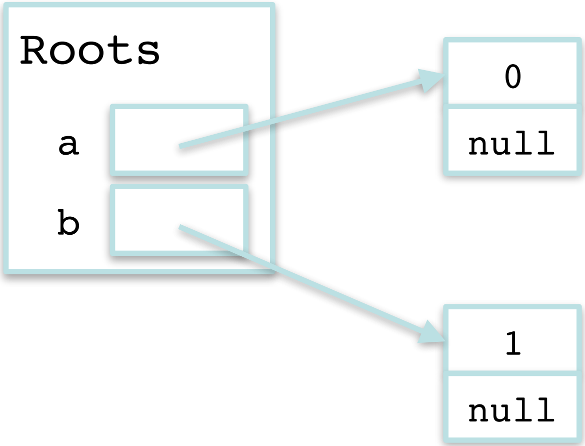
Example



Mutator

```
type Cell = record
  value: int;
  next: ref Cell;
end;
var a, b: ref Cell := null;
a := new Cell(0, null);
b := new Cell(1, null);
a.next := b;
b.next := new Cell(2, null);
a := null;
```

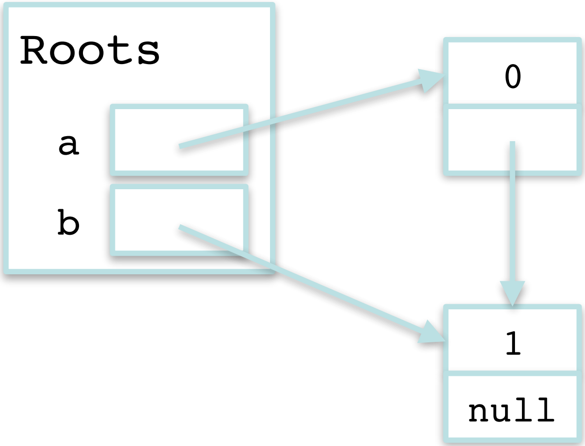
Example



Mutator

```
type Cell = record
  value: int;
  next: ref Cell;
end;
var a, b: ref Cell := null;
a := new Cell(0, null);
b := new Cell(1, null);
a.next := b;
b.next := new Cell(2, null);
a := null;
```

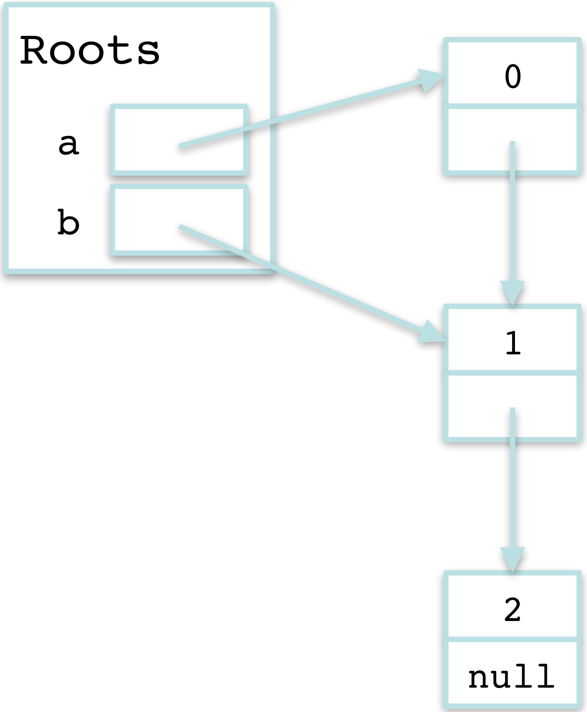
Example



Mutator

```
type Cell = record
  value: int;
  next: ref Cell;
end;
var a, b: ref Cell := null;
a := new Cell(0, null);
b := new Cell(1, null);
a.next := b;
b.next := new Cell(2, null);
a := null;
```

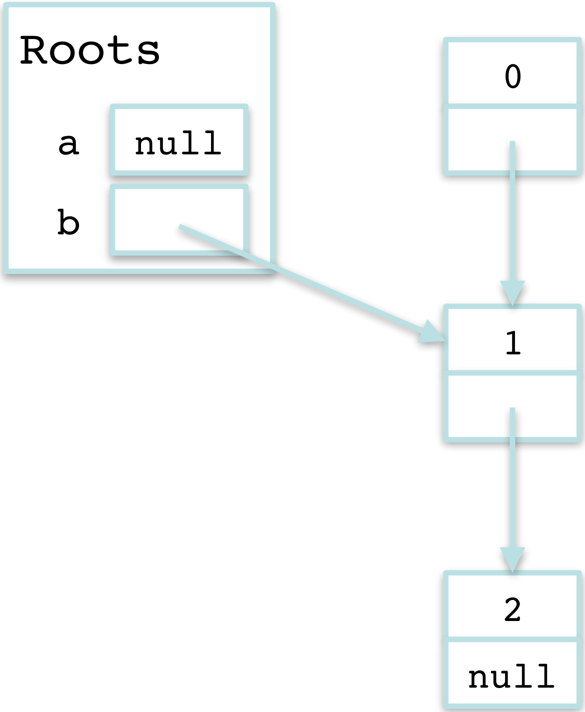
Example



Mutator

```
type Cell = record
  value: int;
  next: ref Cell;
end;
var a, b: ref Cell := null;
a := new Cell(0, null);
b := new Cell(1, null);
a.next := b;
b.next := new Cell(2, null);
a := null;
```

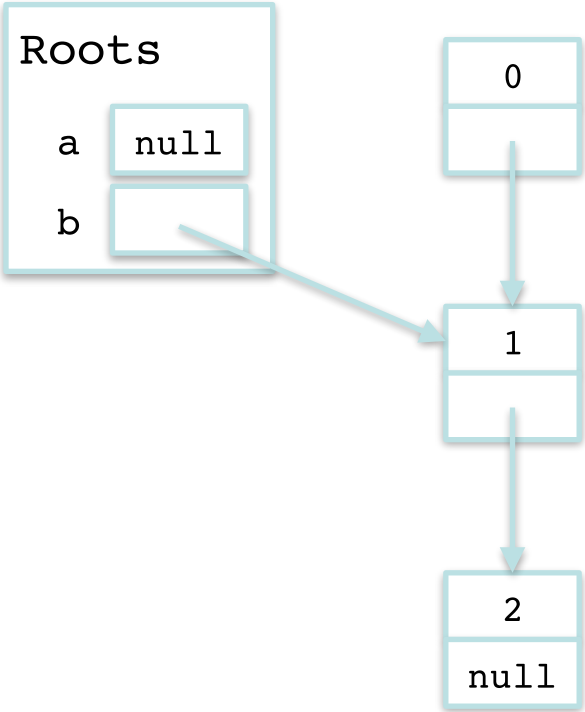
Example



Mutator

```
type Cell = record
  value: int;
  next: ref Cell;
end;
var a, b: ref Cell := null;
a := new Cell(0, null);
b := new Cell(1, null);
a.next := b;
b.next := new Cell(2, null);
a := null;
```

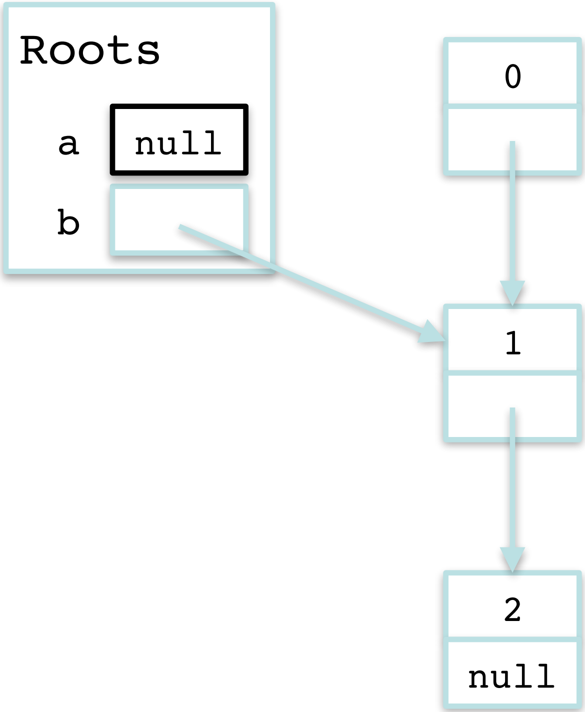
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

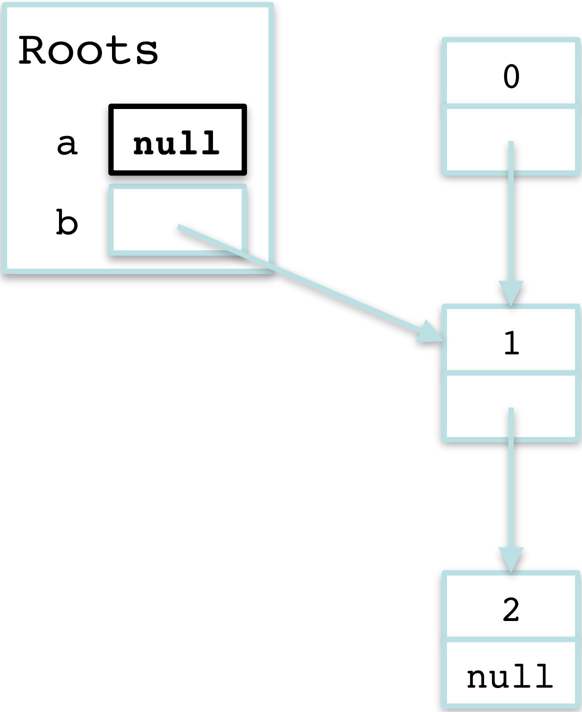
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

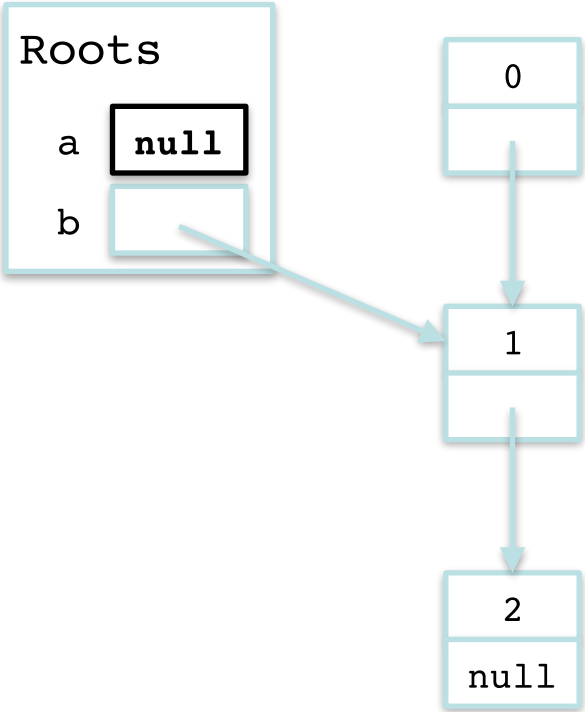
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

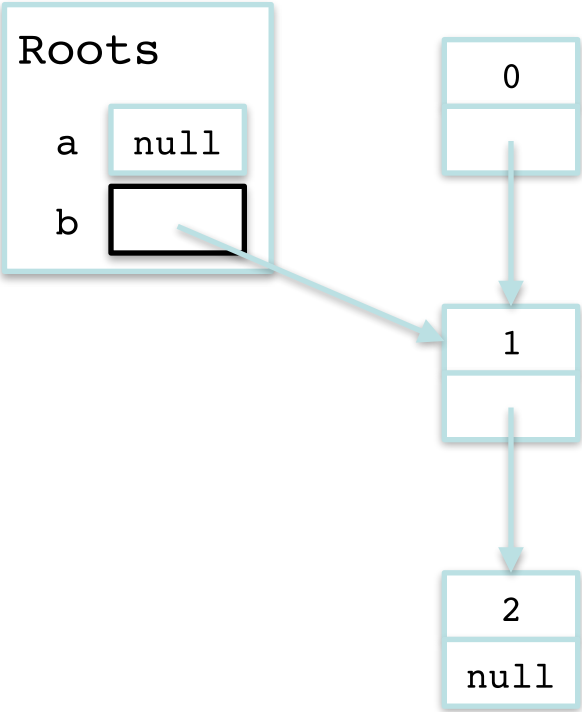
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

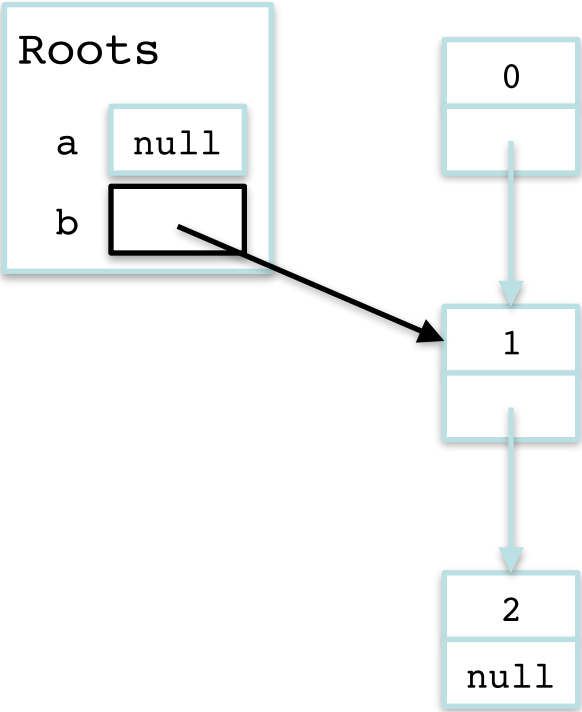
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

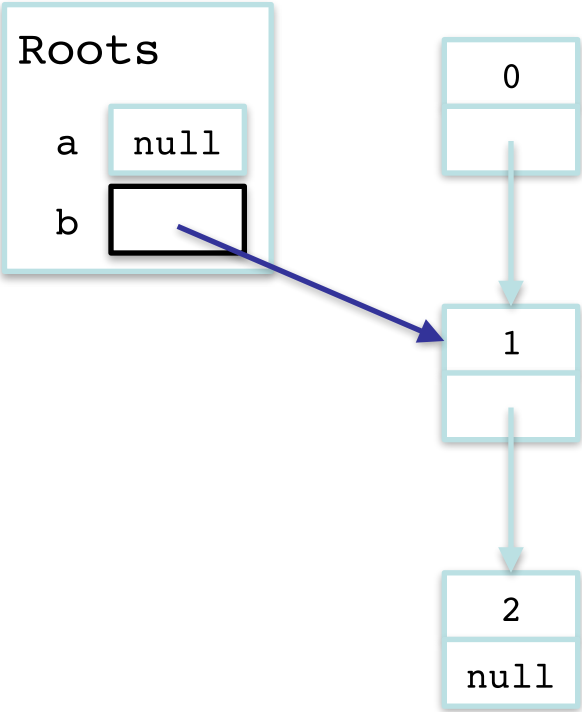
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

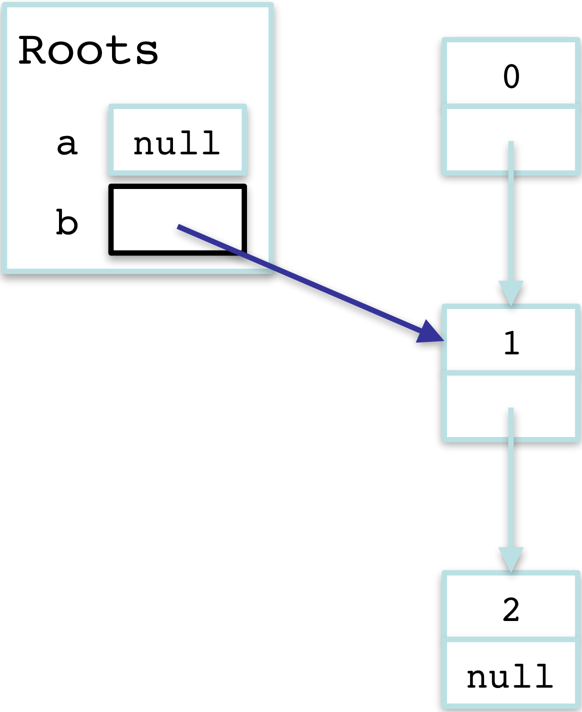
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

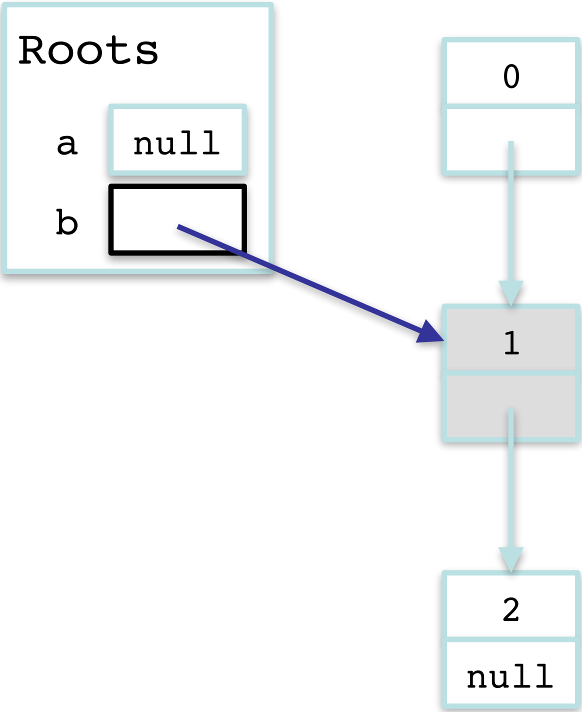
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

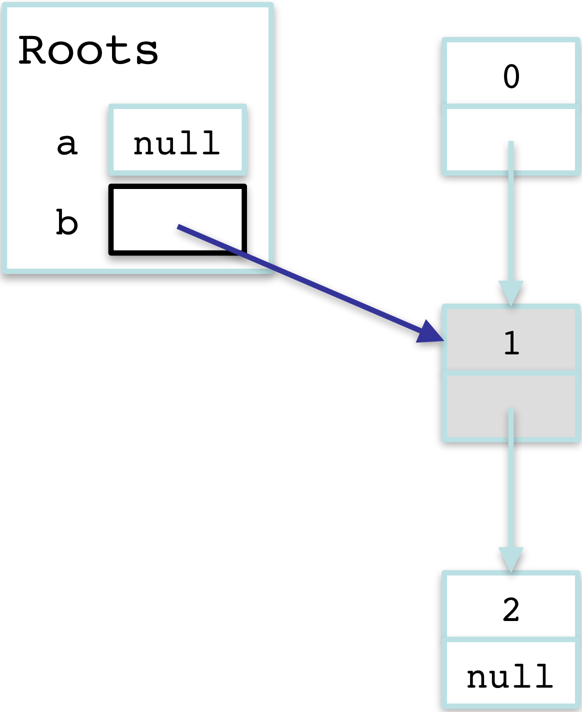
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

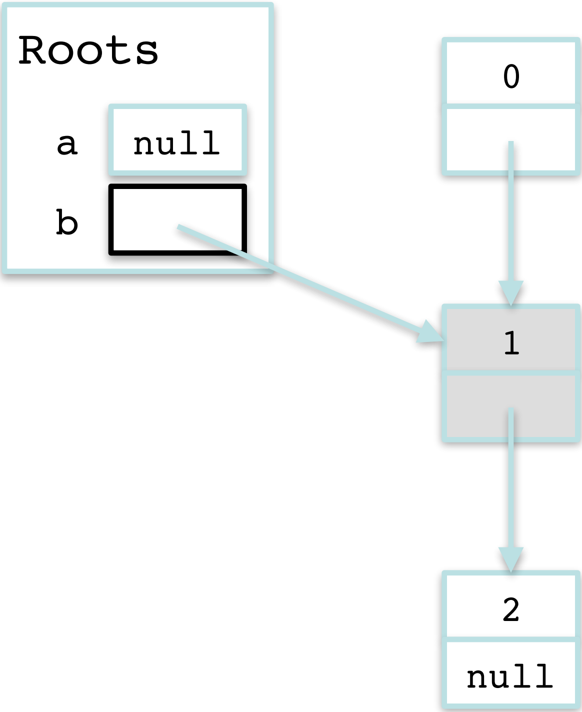
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

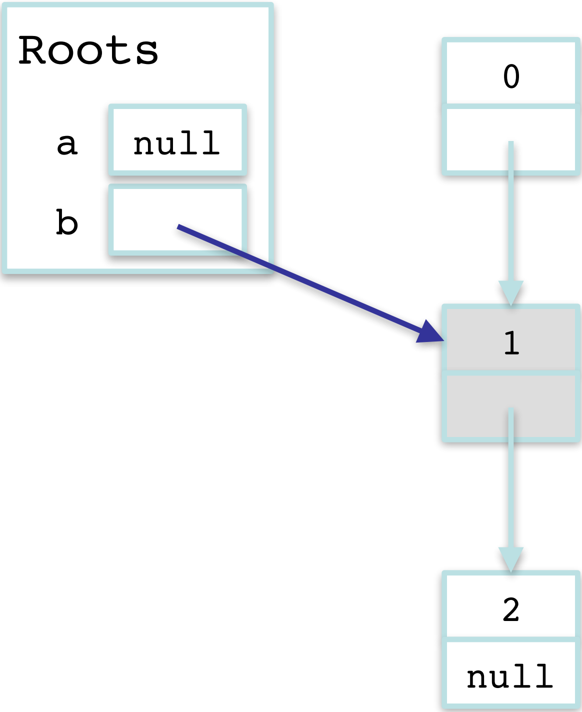
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

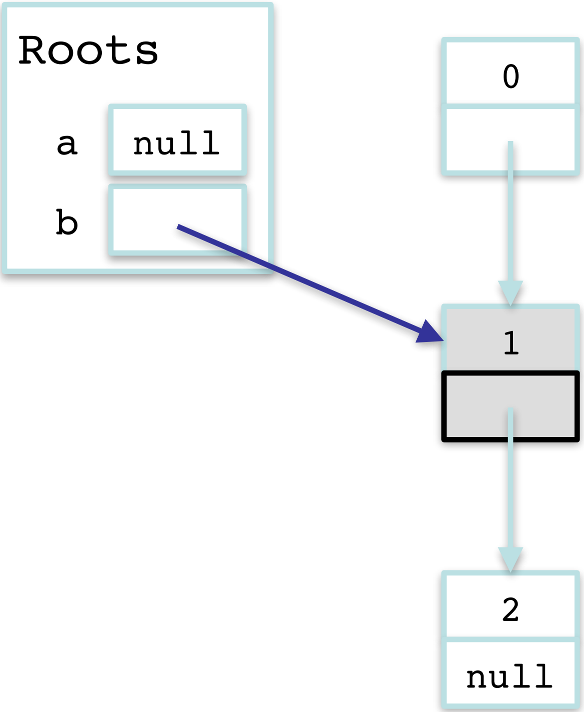
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

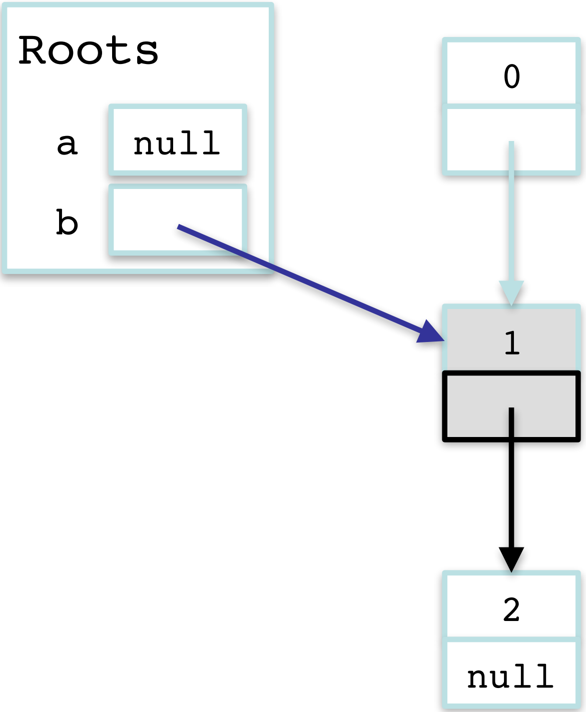
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

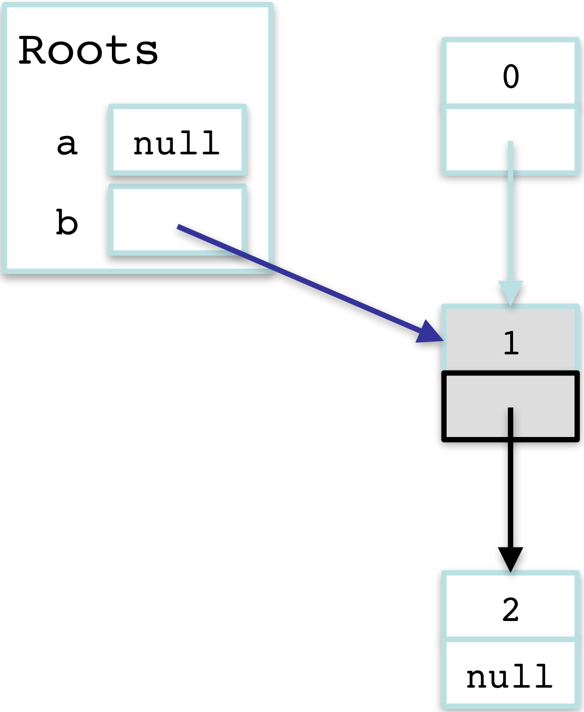
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

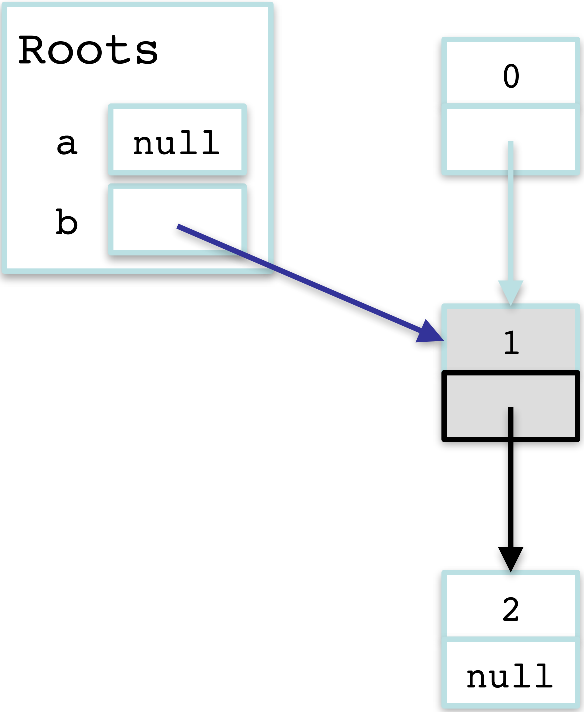
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

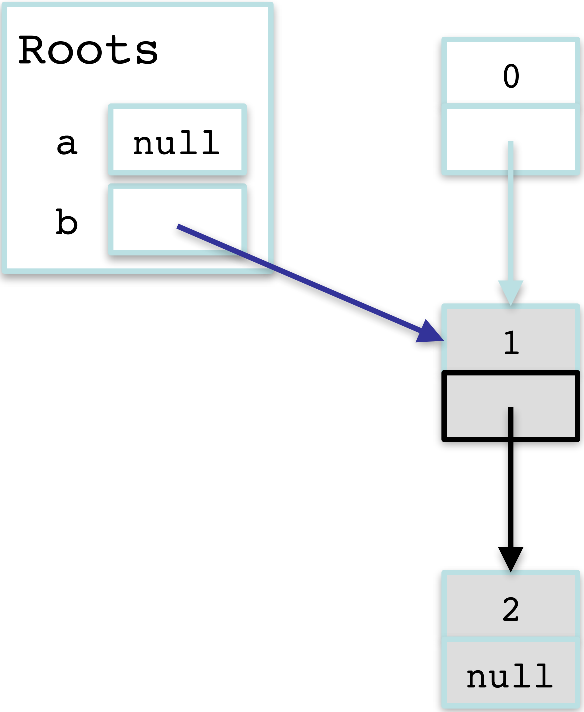
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

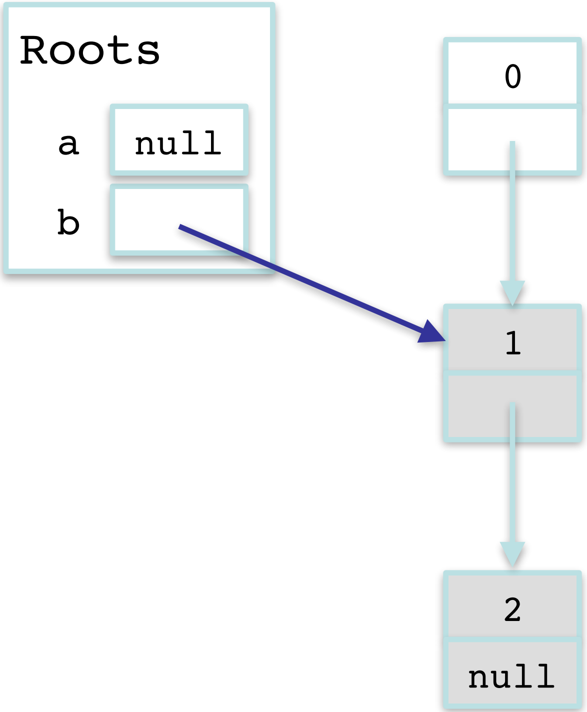
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

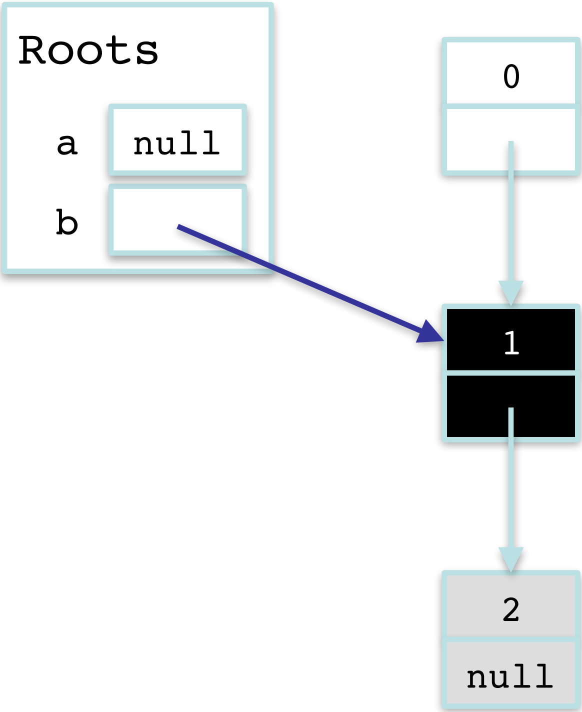
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

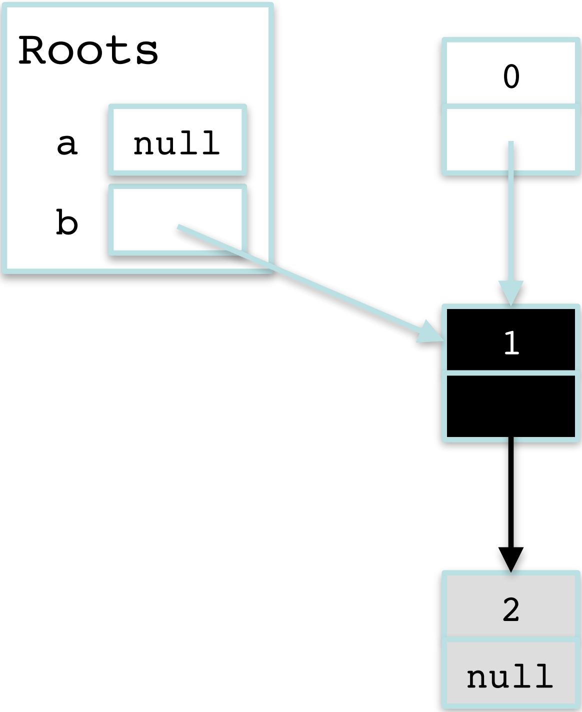
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

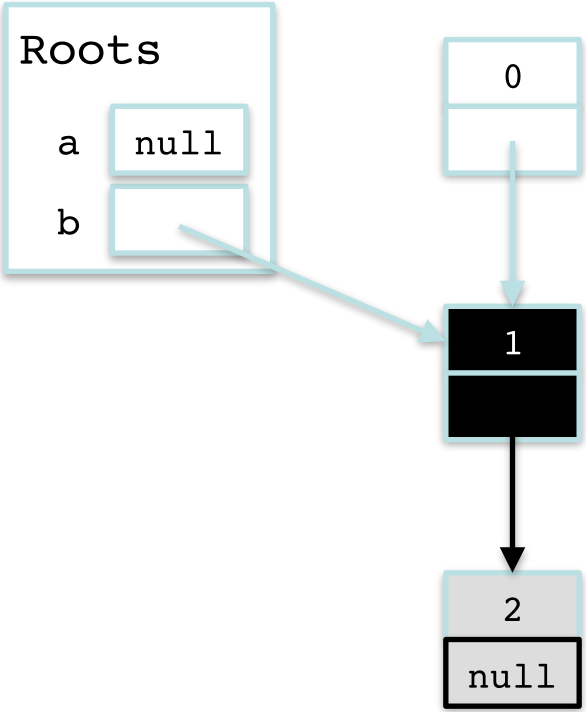
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

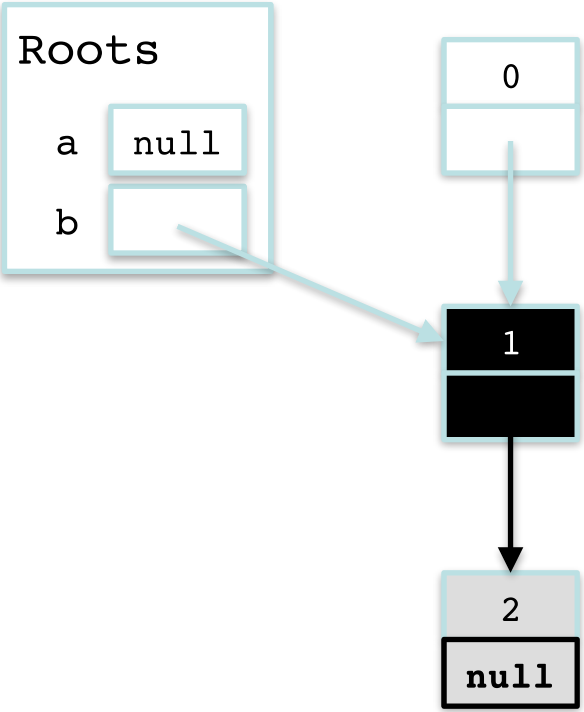
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

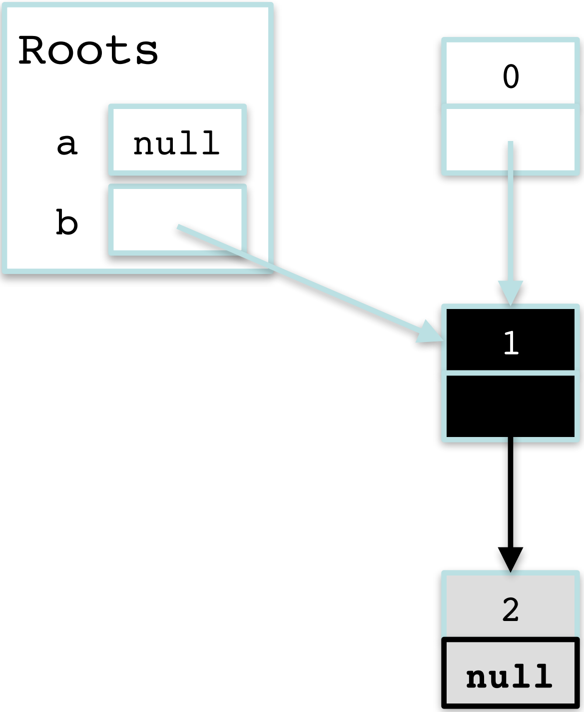
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

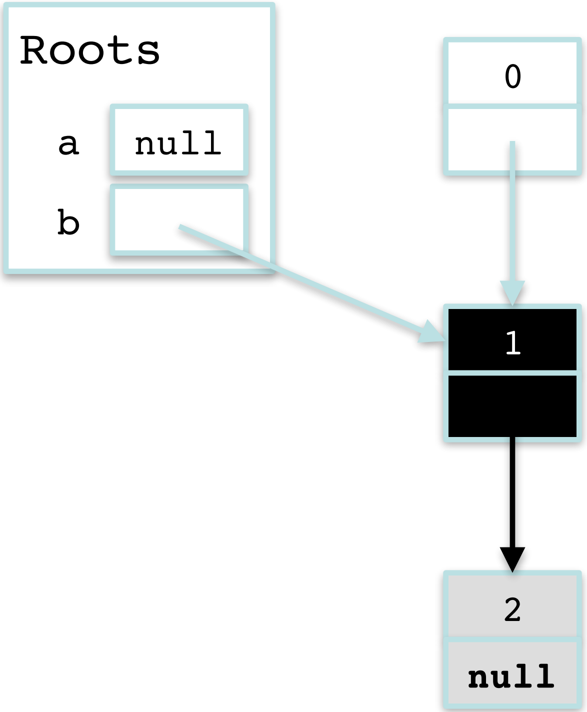
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

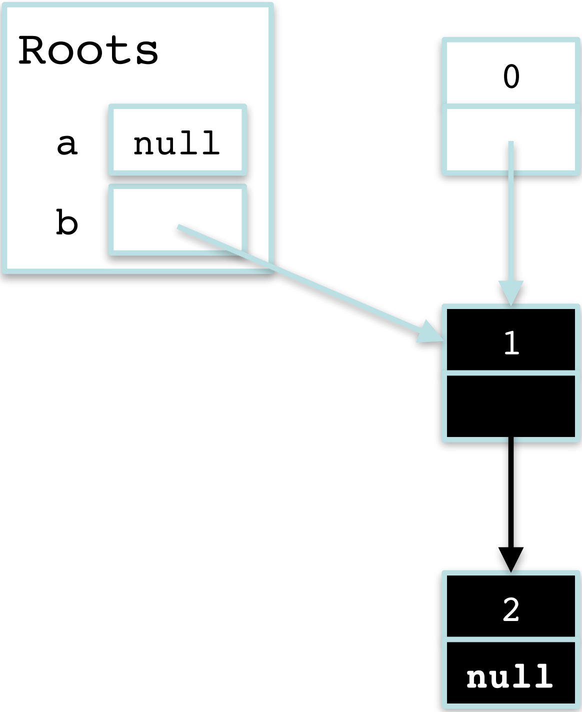
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

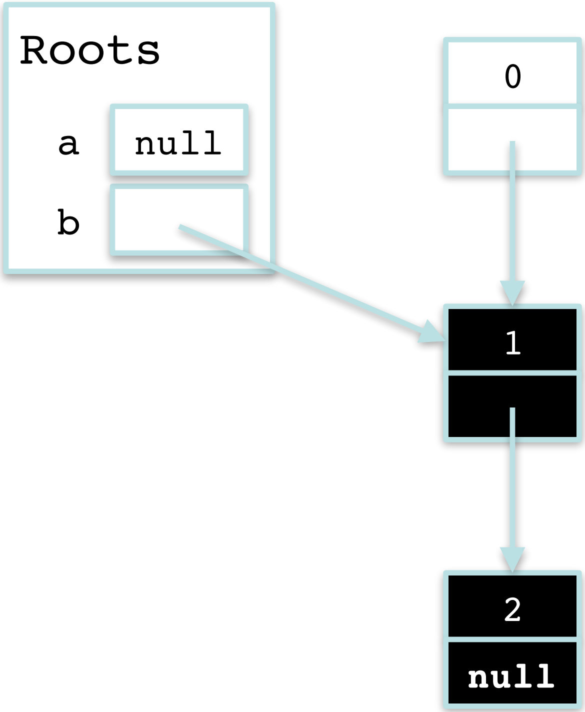
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

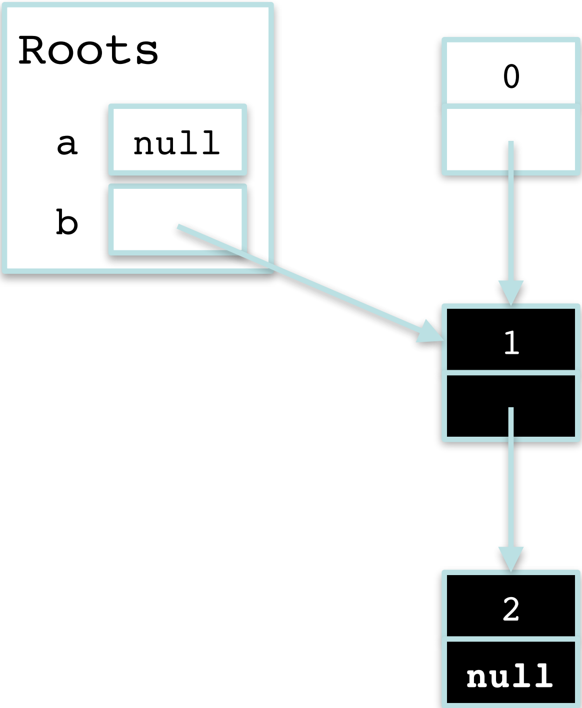
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

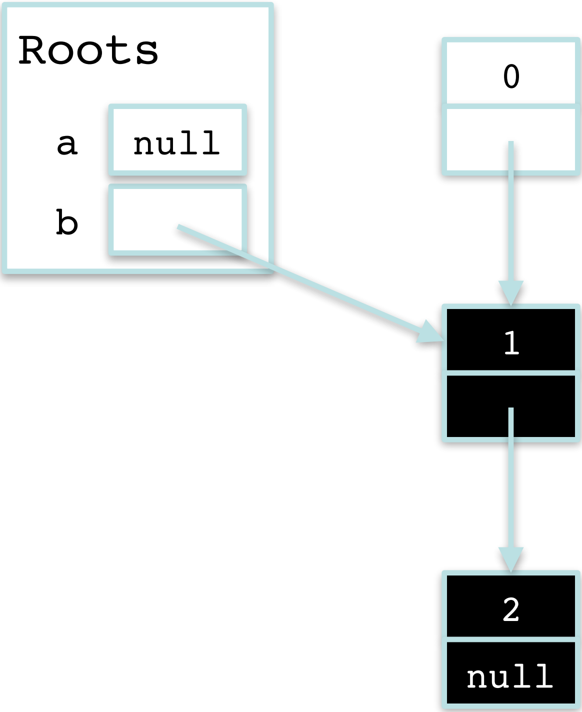
Example



Collector: tri-colour marking

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

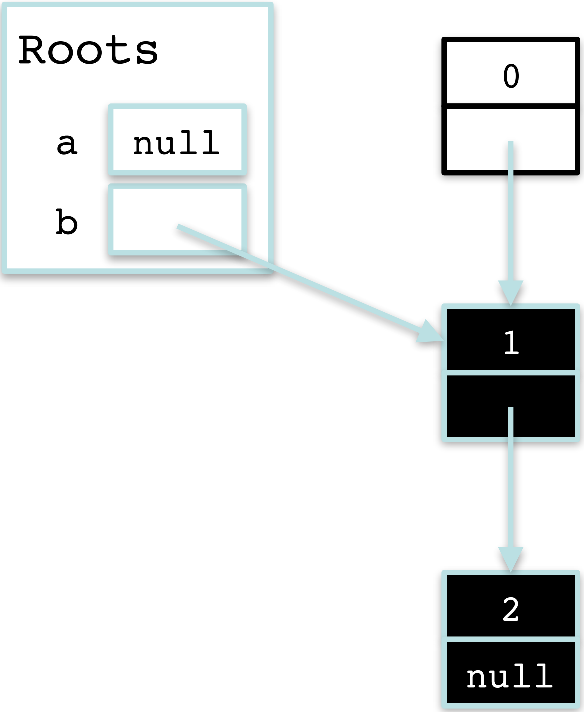
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

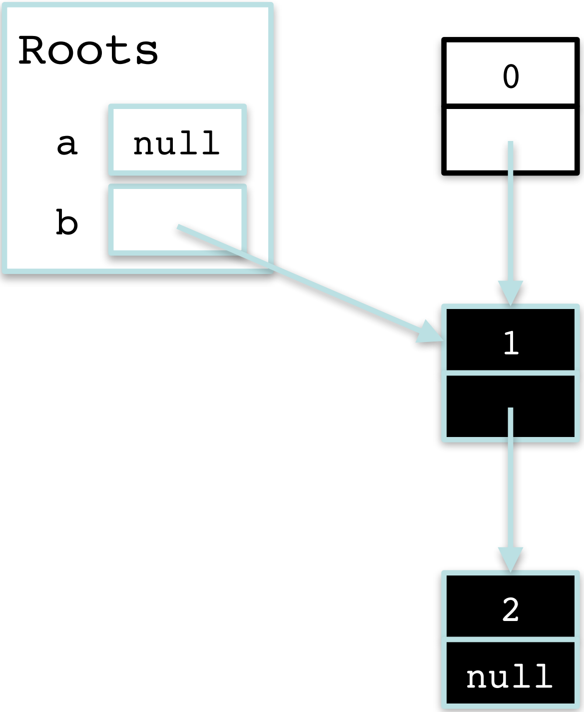
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

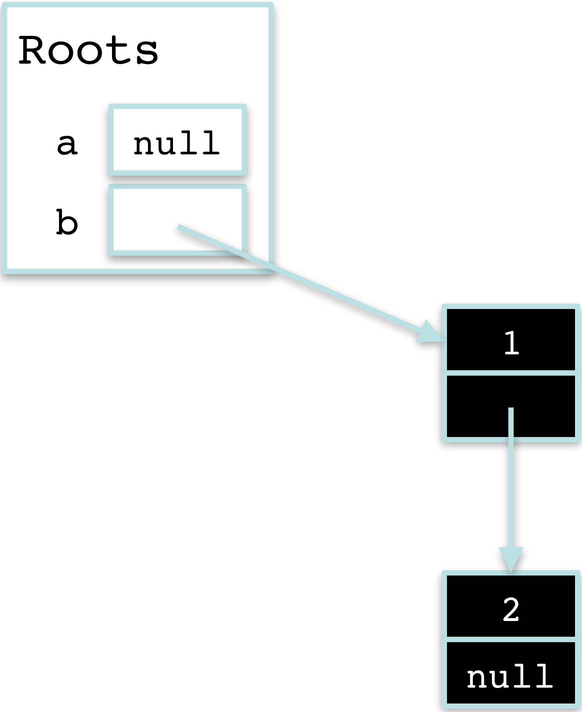
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

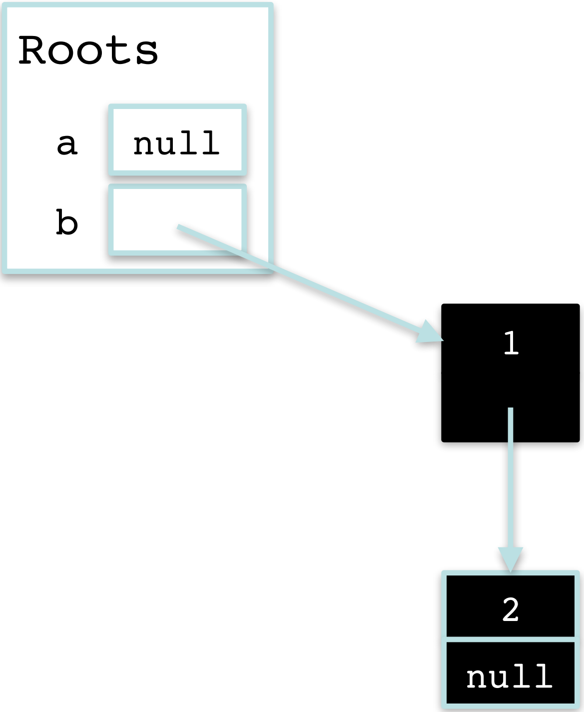
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

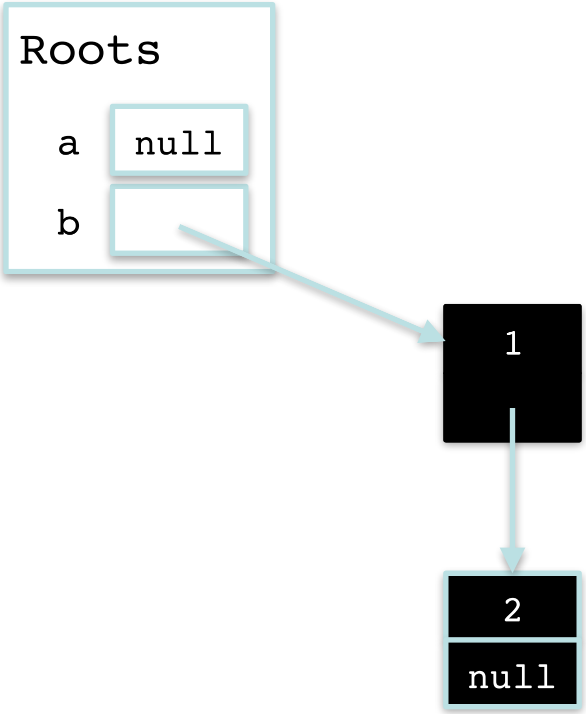
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

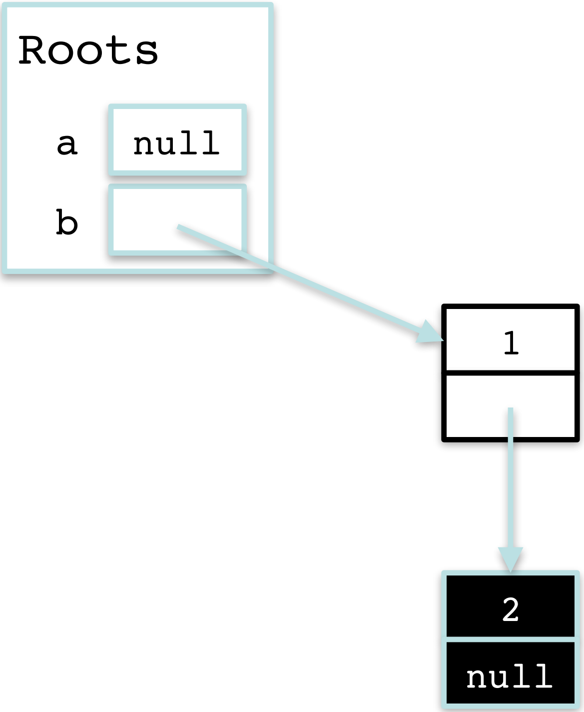
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

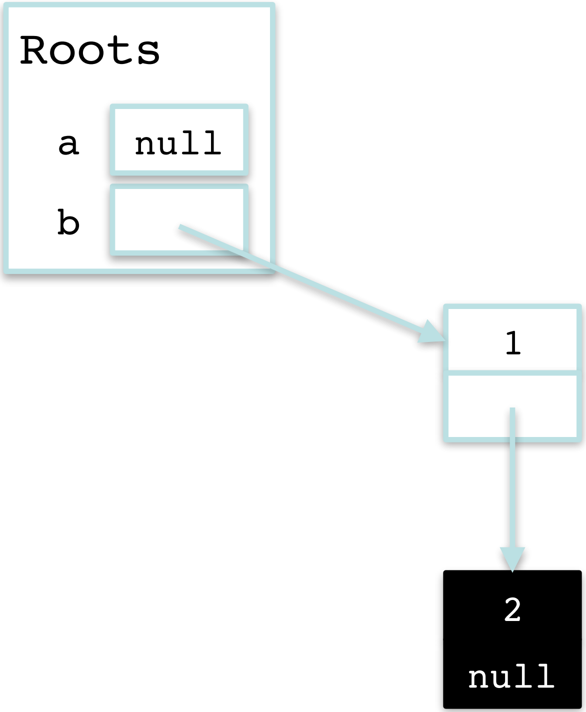
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

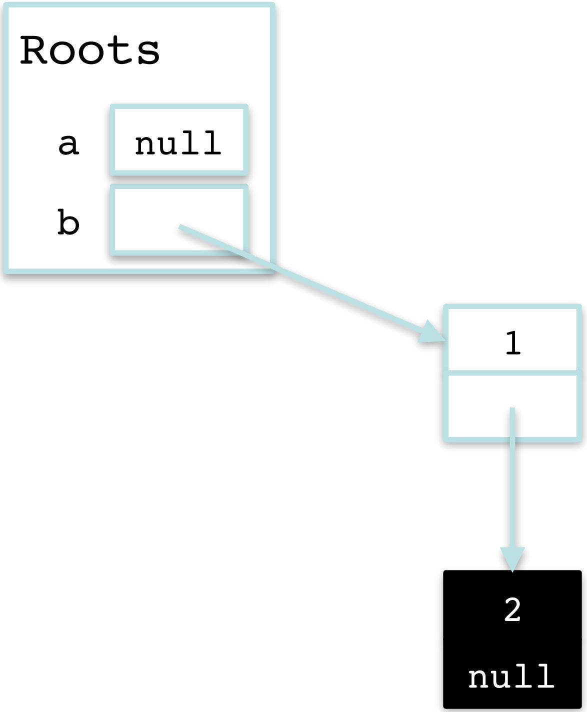
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

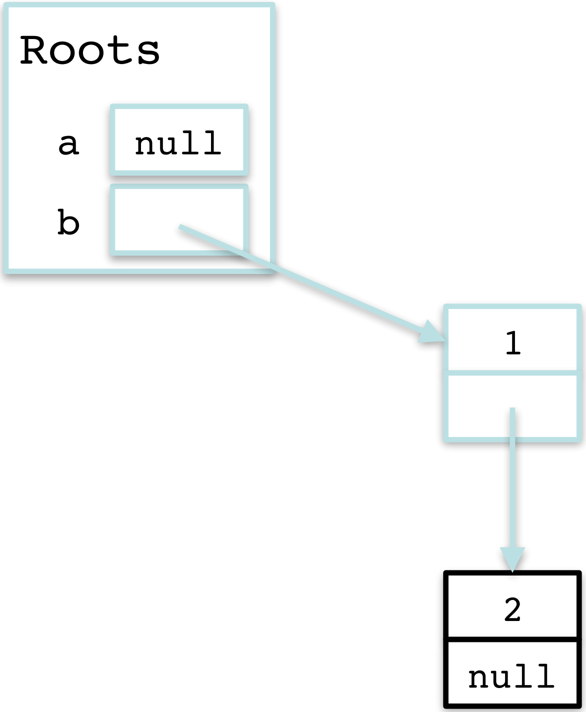
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

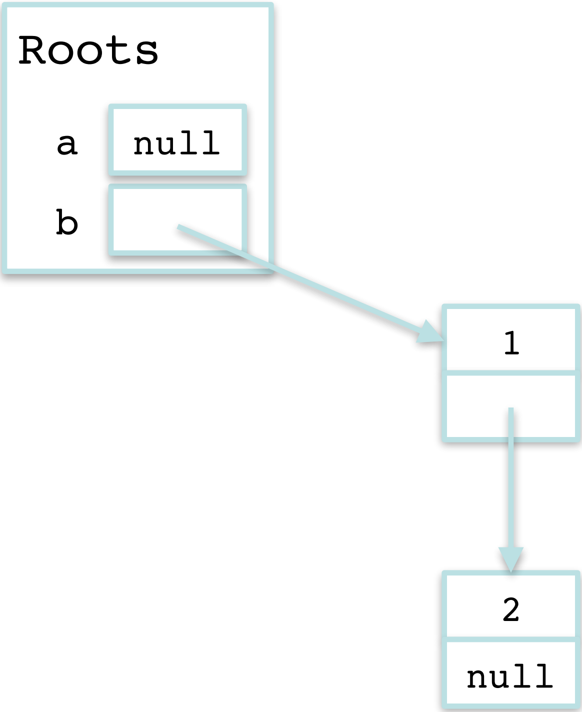
Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

Example



Collector: sweeping

```
for each cell in Heap
  if isWhite(cell)
    free(cell)
  else
    setWhite(cell)
```

Correctness Properties

Safety:

The collector never reclaims *live* objects.

More feasibly: the collector never reclaims *reachable* objects.

Liveness:

The collector eventually completes is collection cycle.

Safety Invariants

At the end of each iteration of the mark loop there are no references from black to white objects.

Thus, any *reachable* white object must be reachable from a grey object.

Thus, all *reachable* white objects will eventually be marked.

```
for each field in Roots
  root := *field
  if root ≠ null && isWhite(root)
    setGrey(root)
    mark()
mark():
  while ∃ source in Grey
    for each field in Pointers(source)
      dest := *field
      if dest ≠ null && isWhite(dest)
        setGrey(dest)
    setBlack(source)
```

INCREMENTAL AND CONCURRENT COLLECTION

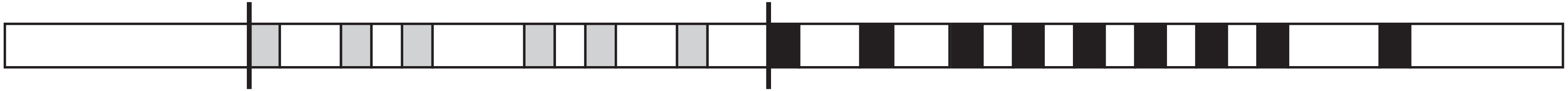
Pauses

The duration of the collection cycle is proportional to the number of marked objects:

- users may notice pauses in interactive applications
- transactions may time out forcing retry
- real-time tasks may miss deadlines

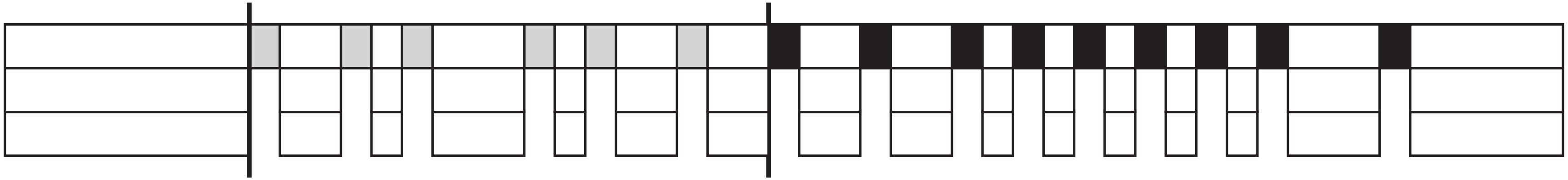
One way to reduce pauses is to use *incremental* or *concurrent collection.*

Uniprocessor incremental collection



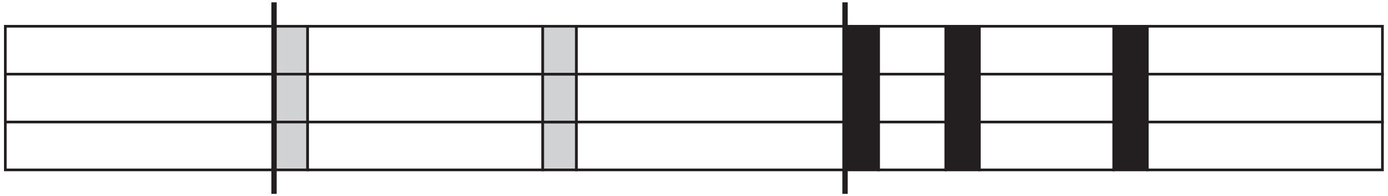
Collector *interleaves* with mutator at defined *safe points* such as allocation sites

Multiprocessor incremental collection



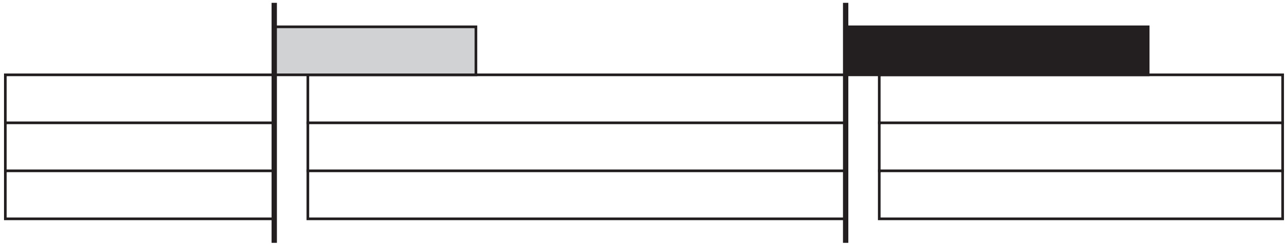
Collector interleaves with mutator threads

Parallel incremental collection



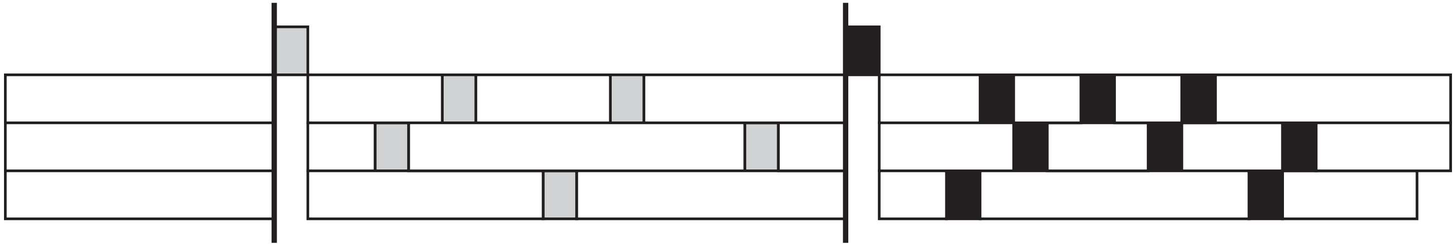
Collector threads interleave with mutator threads

Mostly-concurrent collection



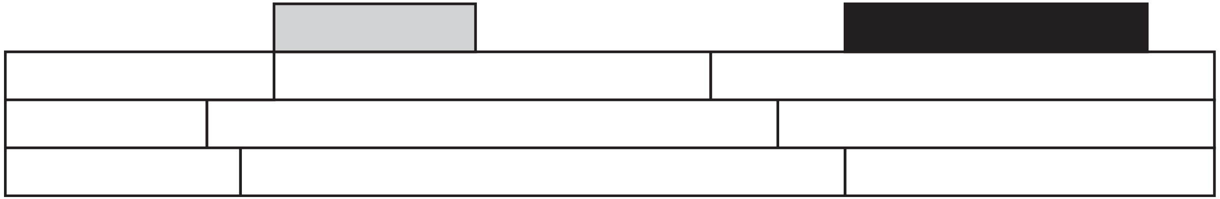
Mutator threads pause together briefly at the beginning of each collection cycle, then run concurrently with the collector

Mostly-concurrent incremental collection



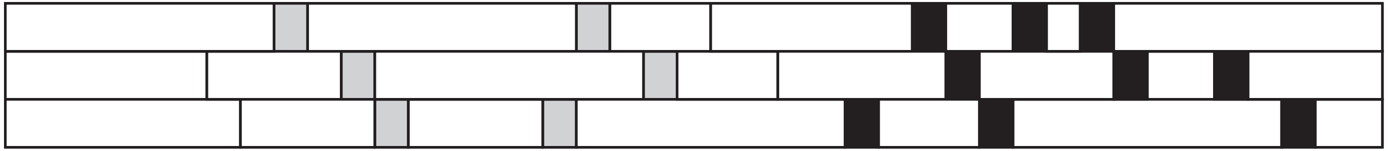
Mutator threads pause together briefly at the beginning of each collection cycle, then interleave with the collector

On-the-fly collection



No *stopping-the-world*: mutator threads separately *synchronize* with the collector at the beginning of each collection cycle, then run concurrently with the collector

On-the-fly incremental collection



No *stopping-the-world*: mutator threads separately *synchronize* with the collector at the beginning of each collection cycle, then interleave with the collector

What can go wrong for concurrent GC?

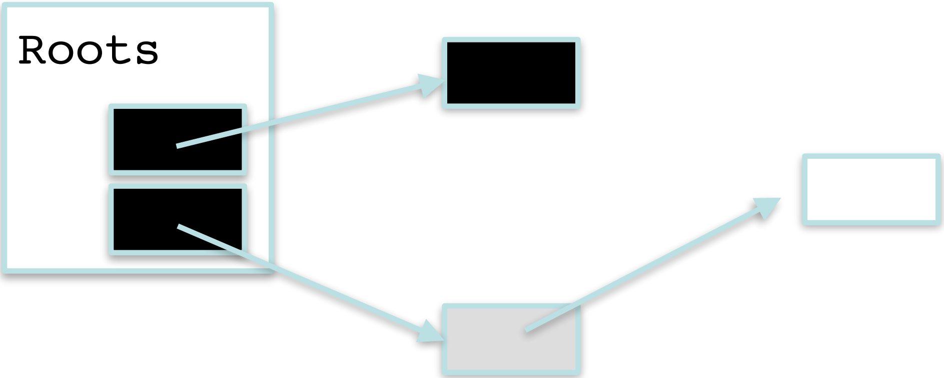
- It’s OK for the mutator to modify objects ahead of the wavefront (grey or reachable white objects) whose fields are still to be scanned
- It’s not OK for the mutator to insert a white pointer into a black object without letting the collector know

The lost object problem

Condition 1: the mutator stores a pointer to a white object into a black object; and

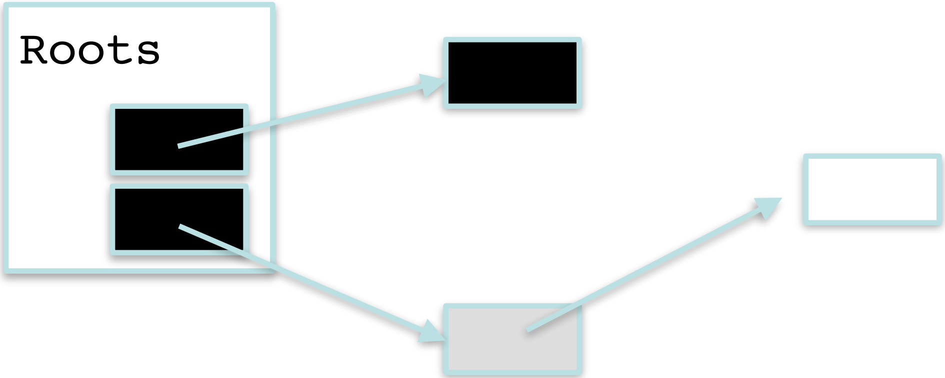
Condition 2: all paths from any grey objects to that white object are destroyed

Example 1: direct



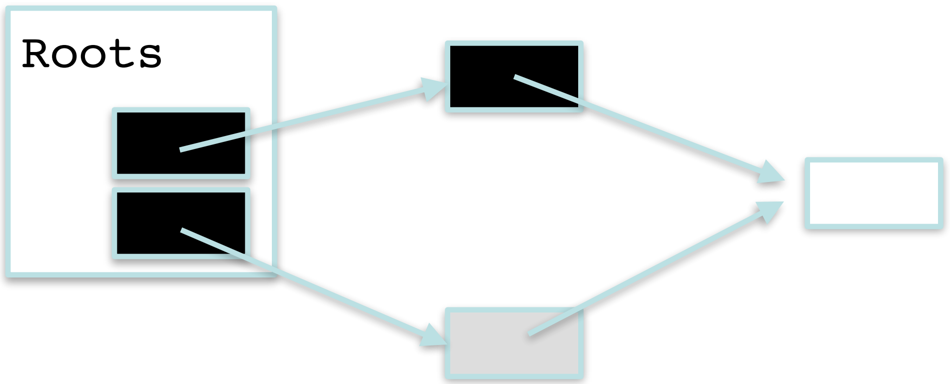
Hiding a reachable white object by dropping a direct link from grey

Example 1: direct



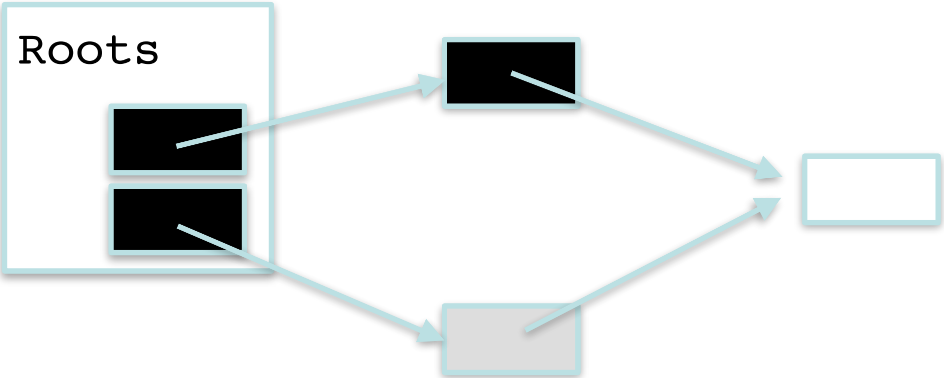
Hiding a reachable white object by dropping a direct link from grey

Example 1: direct



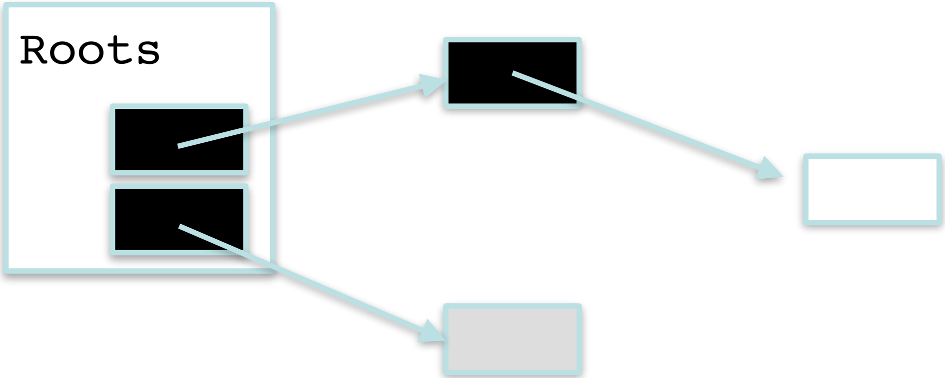
Hiding a reachable white object by dropping a direct link from grey

Example 1: direct



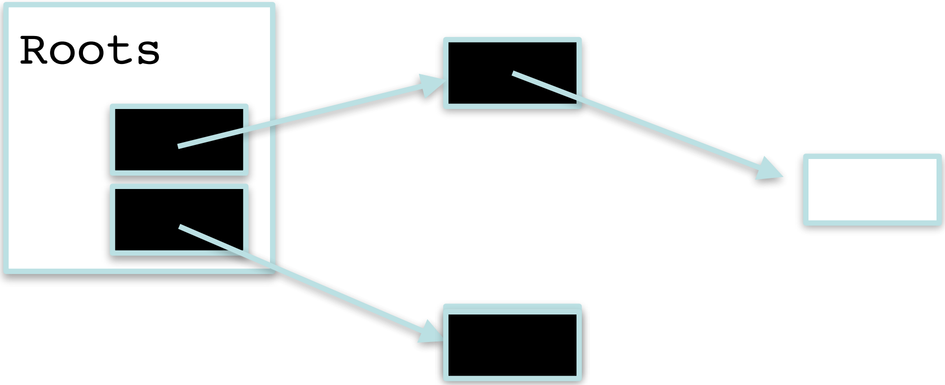
Hiding a reachable white object by dropping a direct link from grey

Example 1: direct



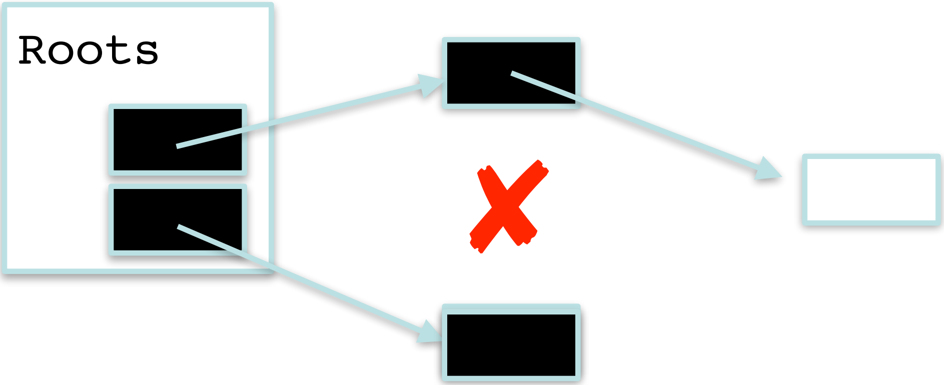
Hiding a reachable white object by dropping a direct link from grey

Example 1: direct



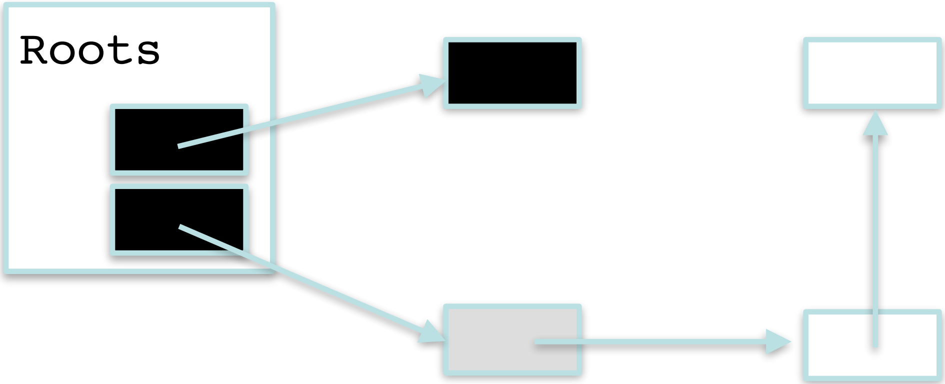
Hiding a reachable white object by dropping a direct link from grey

Example 1: direct



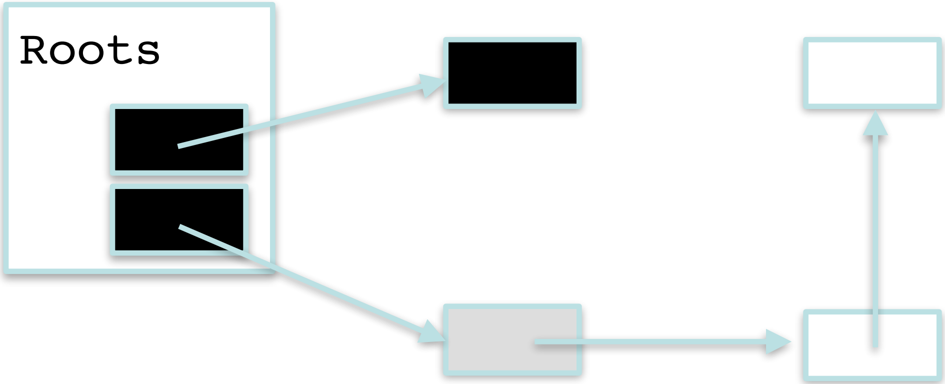
Hiding a reachable white object by dropping a direct link from grey

Example 2: indirect



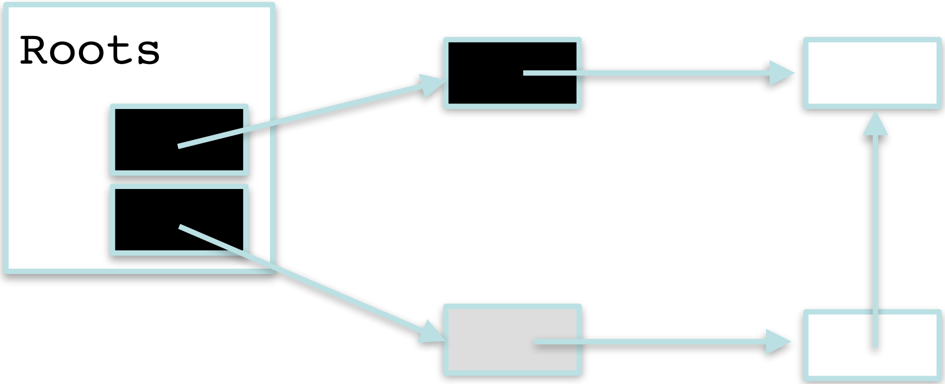
Hiding a transitively reachable white object by breaking an indirect chain from grey

Example 2: indirect



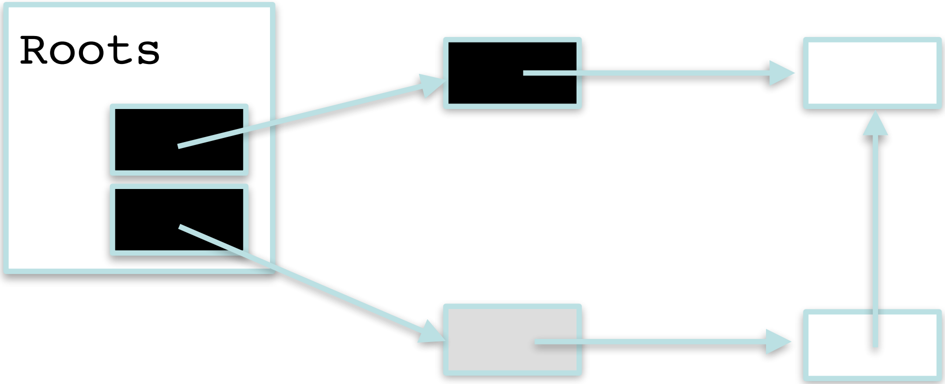
Hiding a transitively reachable white object by breaking an indirect chain from grey

Example 2: indirect



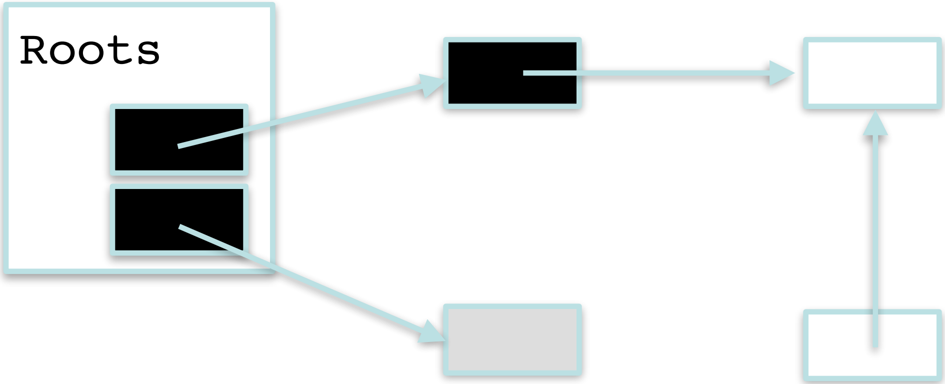
Hiding a transitively reachable white object by breaking an indirect chain from grey

Example 2: indirect



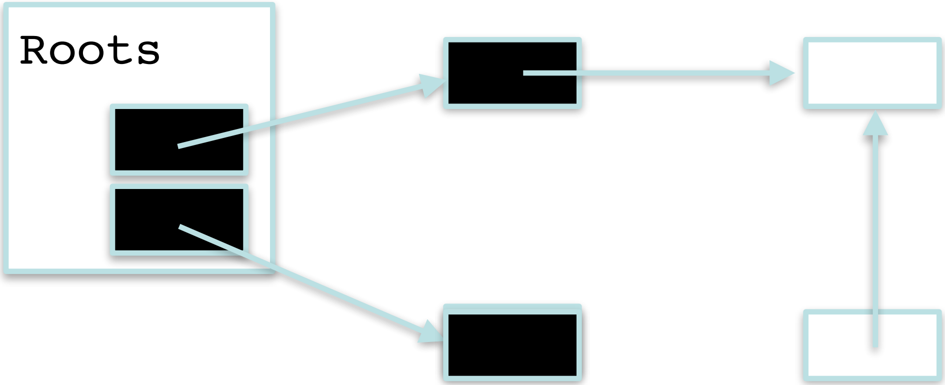
Hiding a transitively reachable white object by breaking an indirect chain from grey

Example 2: indirect



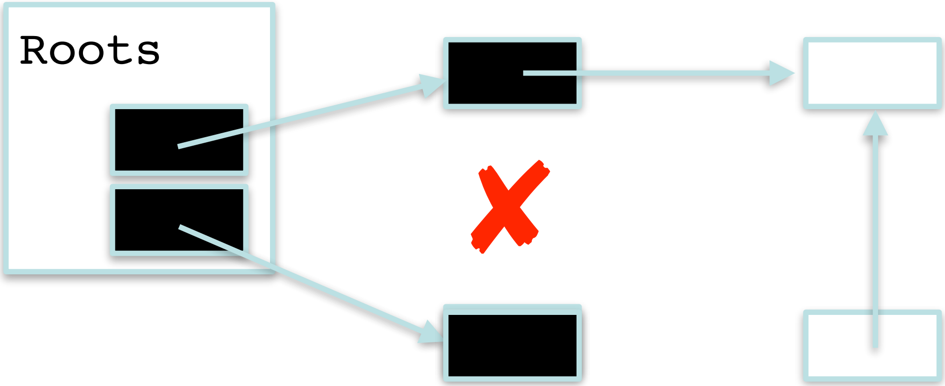
Hiding a transitively reachable white object by breaking an indirect chain from grey

Example 2: indirect



Hiding a transitively reachable white object by breaking an indirect chain from grey

Example 2: indirect



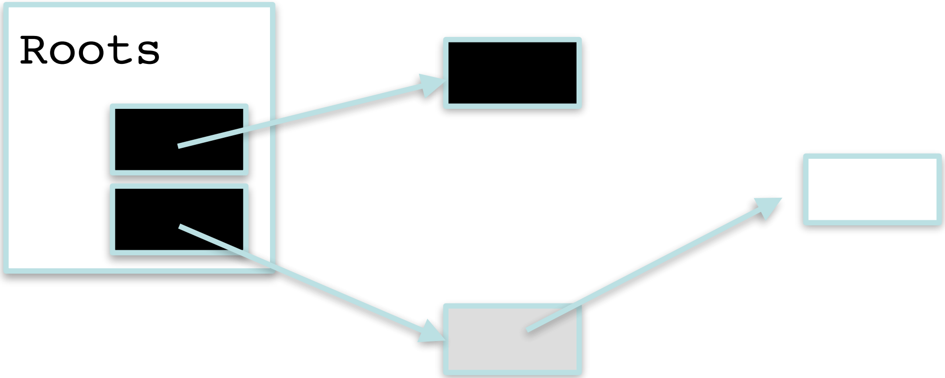
Hiding a transitively reachable white object by breaking an indirect chain from grey

The strong tricolor invariant

Preventing Condition 1:

Ensure there are no pointers from black objects to white objects

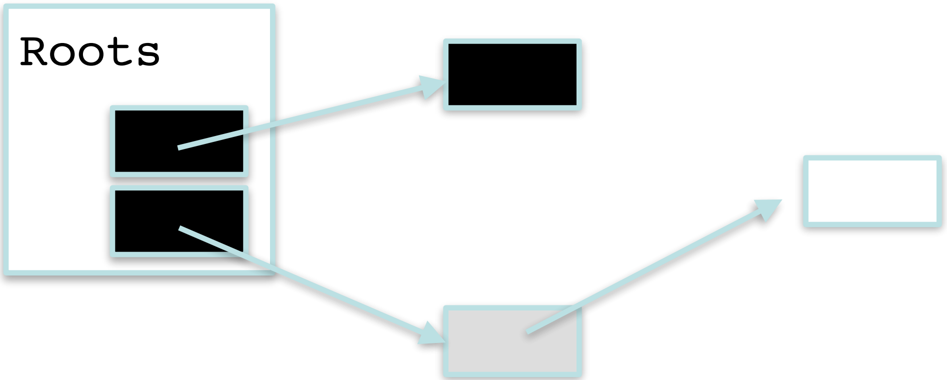
Insertion barrier [Steele]



Retreat the wavefront: grey the source

Mutator is allowed to have white roots

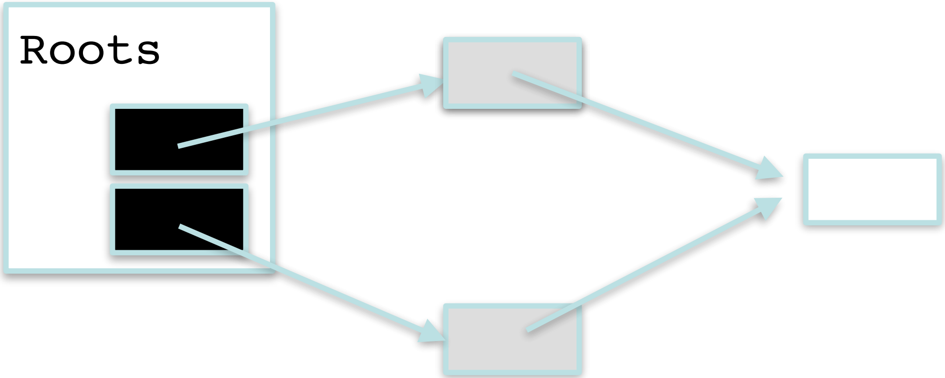
Insertion barrier [Steele]



Retreat the wavefront: grey the source

Mutator is allowed to have white roots

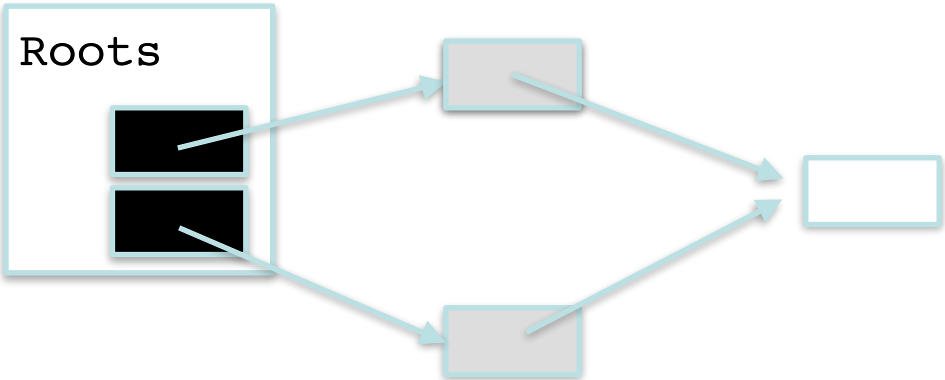
Insertion barrier [Steele]



Retreat the wavefront: grey the source

Mutator is allowed to have white roots

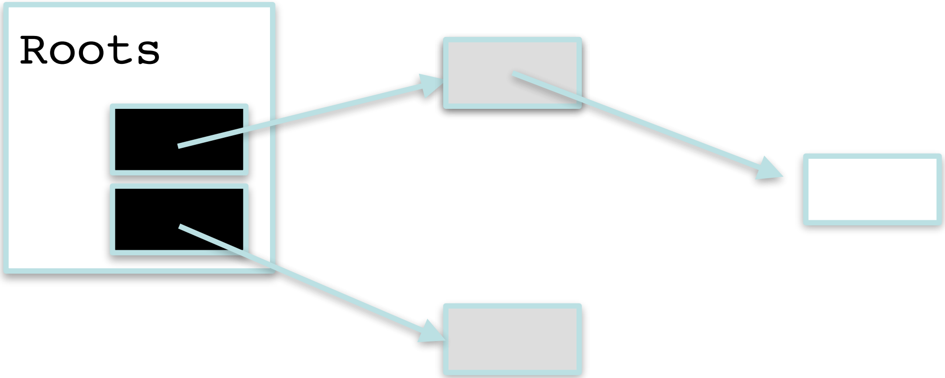
Insertion barrier [Steele]



Retreat the wavefront: grey the source

Mutator is allowed to have white roots

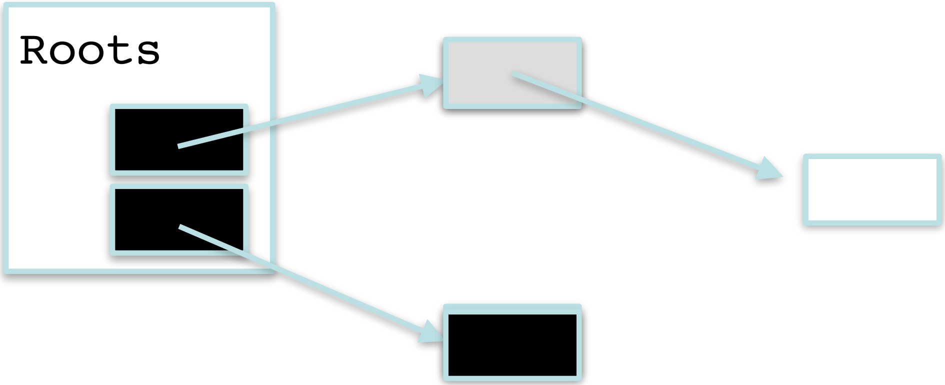
Insertion barrier [Steele]



Retreat the wavefront: grey the source

Mutator is allowed to have white roots

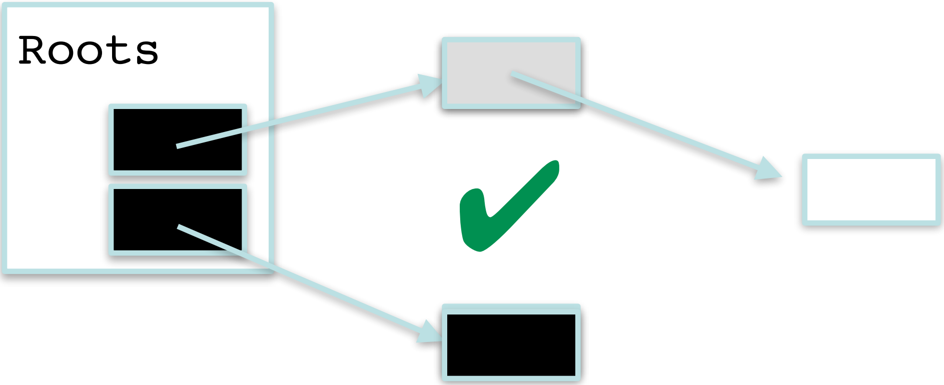
Insertion barrier [Steele]



Retreat the wavefront: grey the source

Mutator is allowed to have white roots

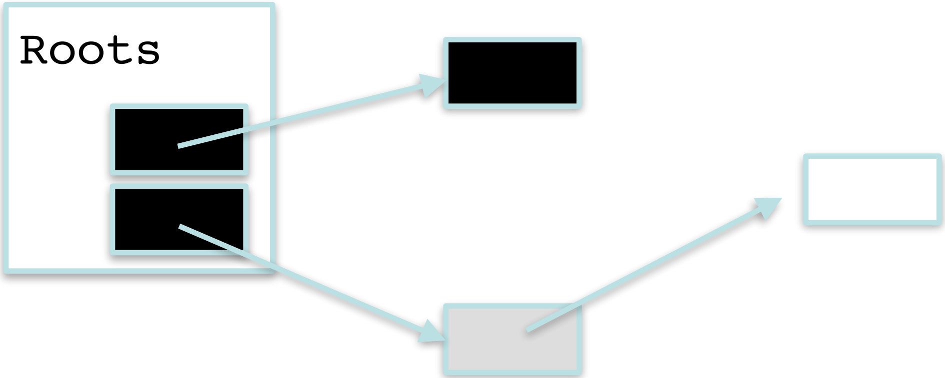
Insertion barrier [Steele]



Retreat the wavefront: grey the source

Mutator is allowed to have white roots

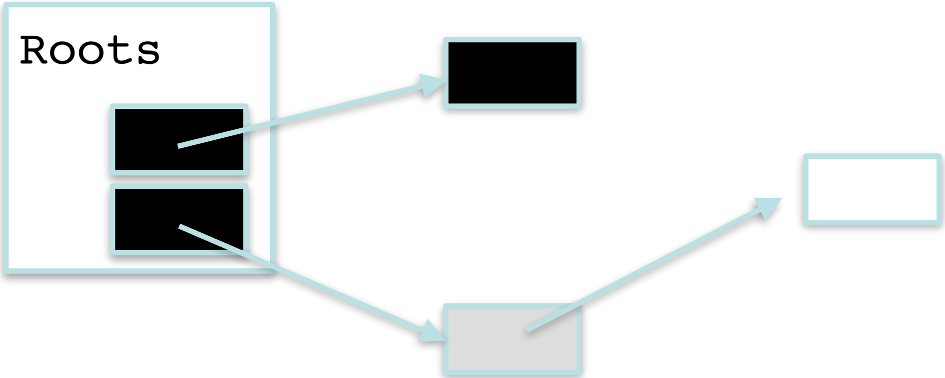
Insertion barrier [Dijkstra]



Advance the wavefront: shade the target

Mutator is allowed to have white roots

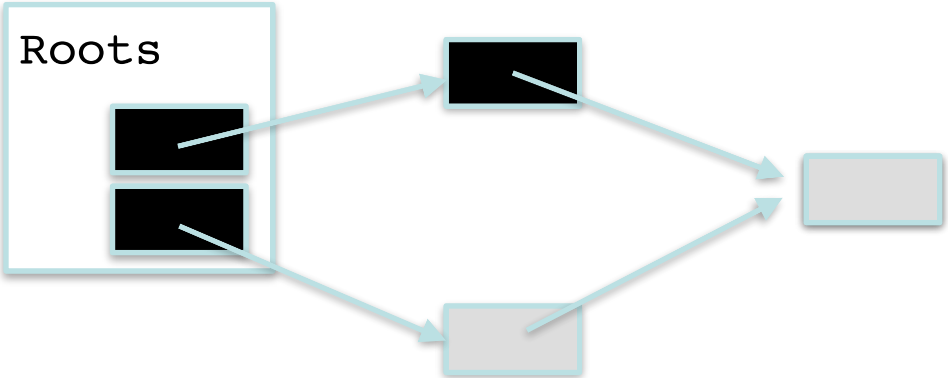
Insertion barrier [Dijkstra]



Advance the wavefront: shade the target

Mutator is allowed to have white roots

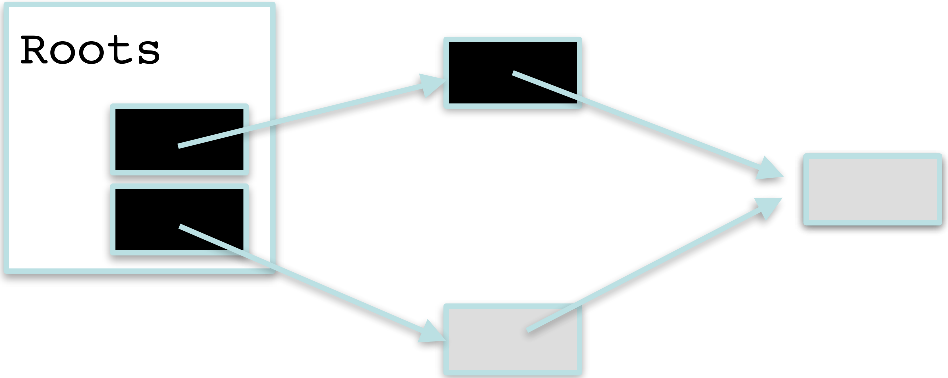
Insertion barrier [Dijkstra]



Advance the wavefront: shade the target

Mutator is allowed to have white roots

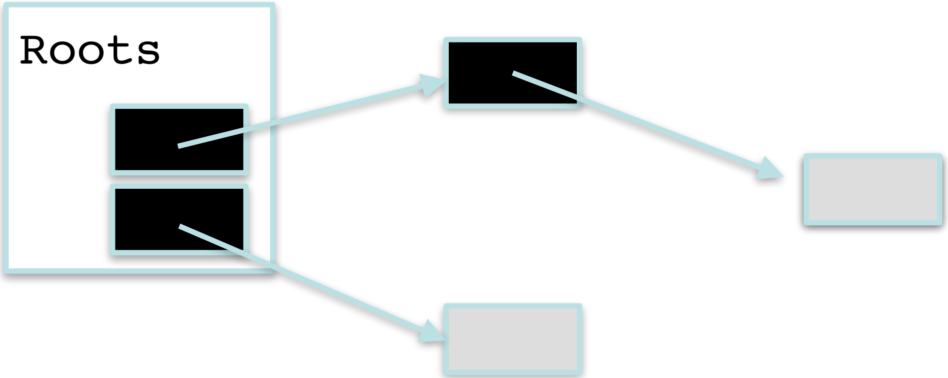
Insertion barrier [Dijkstra]



Advance the wavefront: shade the target

Mutator is allowed to have white roots

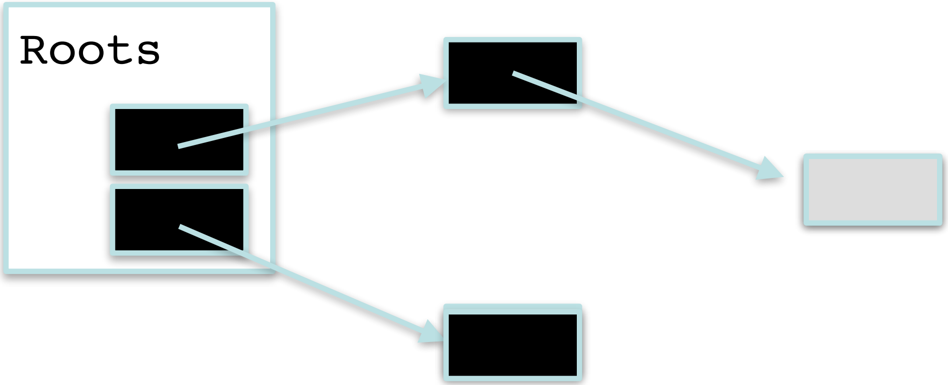
Insertion barrier [Dijkstra]



Advance the wavefront: shade the target

Mutator is allowed to have white roots

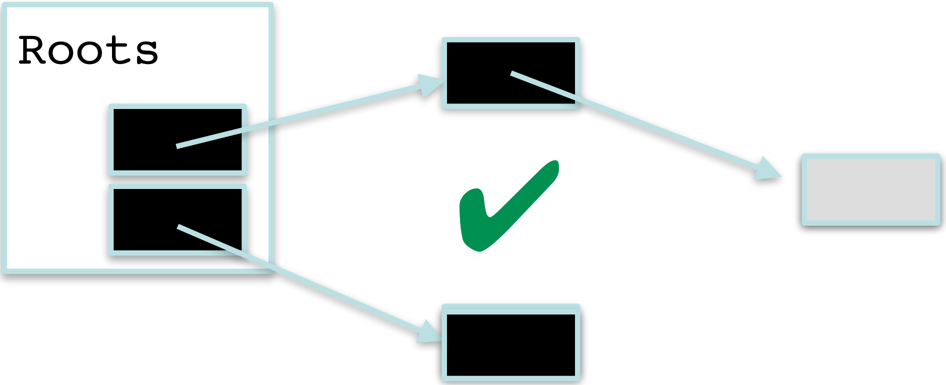
Insertion barrier [Dijkstra]



Advance the wavefront: shade the target

Mutator is allowed to have white roots

Insertion barrier [Dijkstra]



Advance the wavefront: shade the target

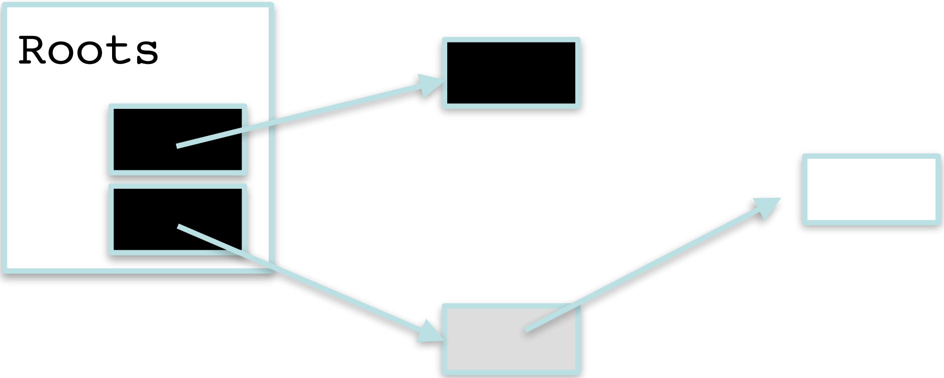
Mutator is allowed to have white roots

The weak tricolor invariant

Preventing Condition 1:

Every white object pointed to by a black object must be grey protected: that is, reachable from some grey object directly or indirectly via a chain of white objects

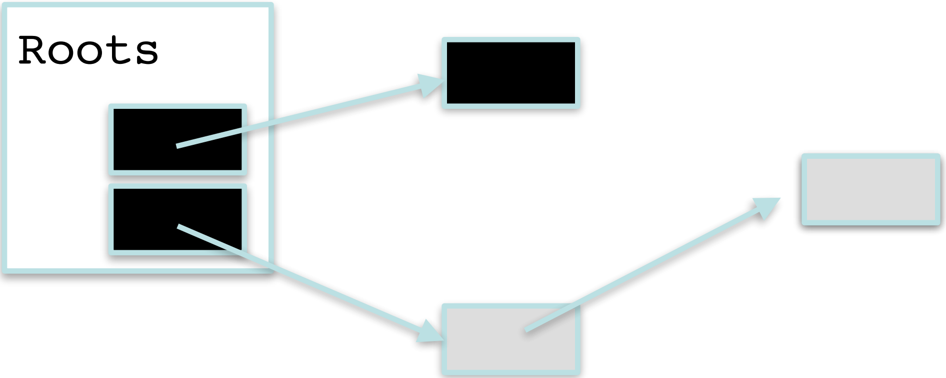
Read barrier [Baker]



Advance the wavefront: shade the target on load

Mutator is prevented from holding white roots

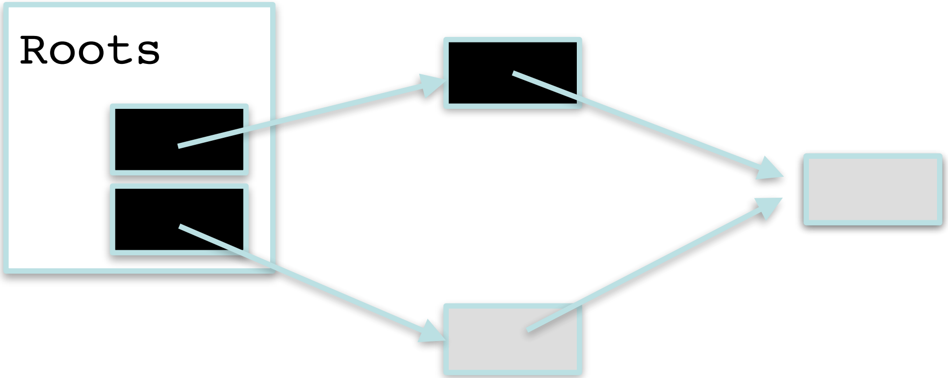
Read barrier [Baker]



Advance the wavefront: shade the target on load

Mutator is prevented from holding white roots

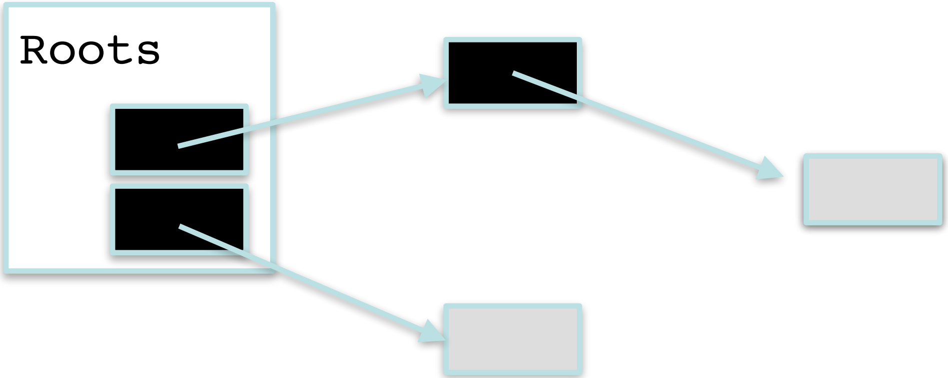
Read barrier [Baker]



Advance the wavefront: shade the target on load

Mutator is prevented from holding white roots

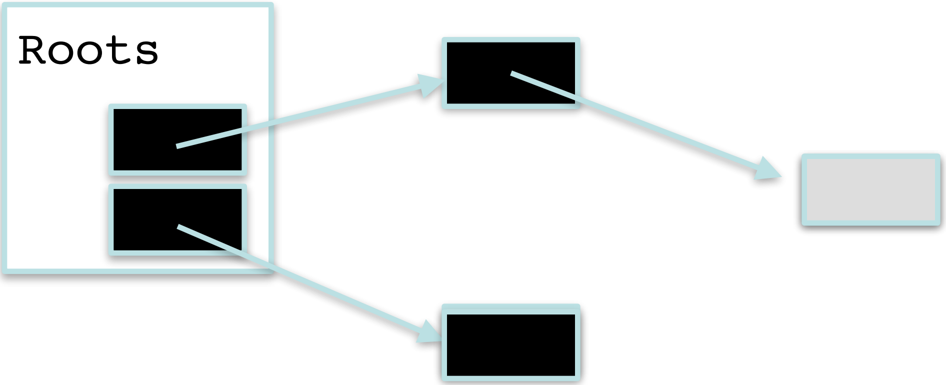
Read barrier [Baker]



Advance the wavefront: shade the target on load

Mutator is prevented from holding white roots

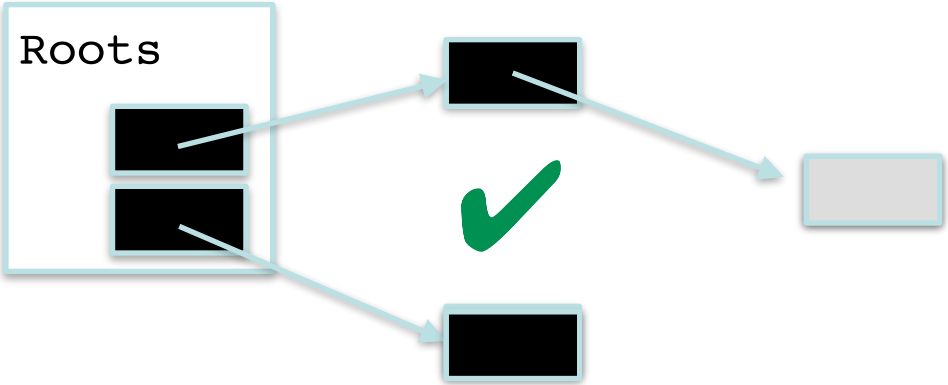
Read barrier [Baker]



Advance the wavefront: shade the target on load

Mutator is prevented from holding white roots

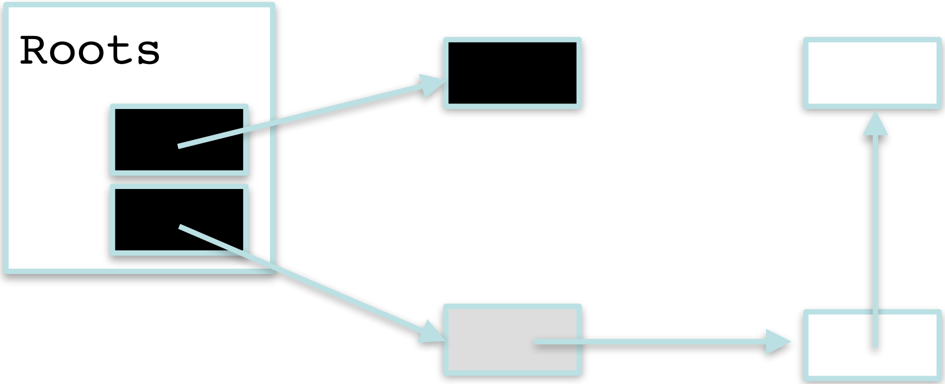
Read barrier [Baker]



Advance the wavefront: shade the target on load

Mutator is prevented from holding white roots

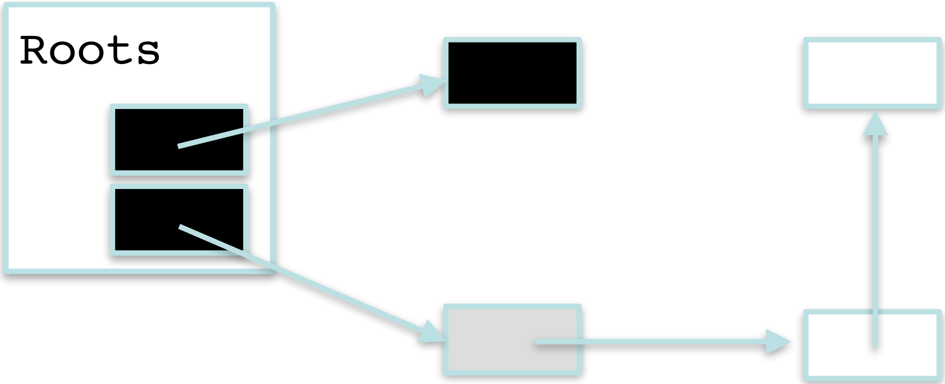
Deletion barrier [Abraham & Patel; Yuasa]



Advance the wavefront: shade the deleted target

Weak invariant allows mutator to hold white roots

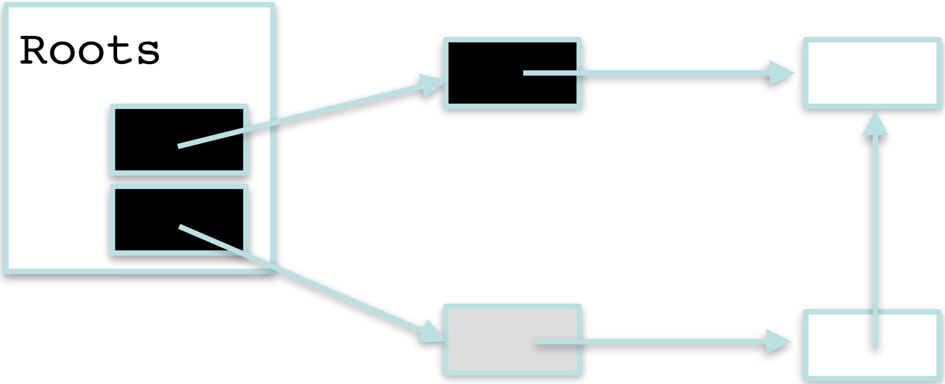
Deletion barrier [Abraham & Patel; Yuasa]



Advance the wavefront: shade the deleted target

Weak invariant allows mutator to hold white roots

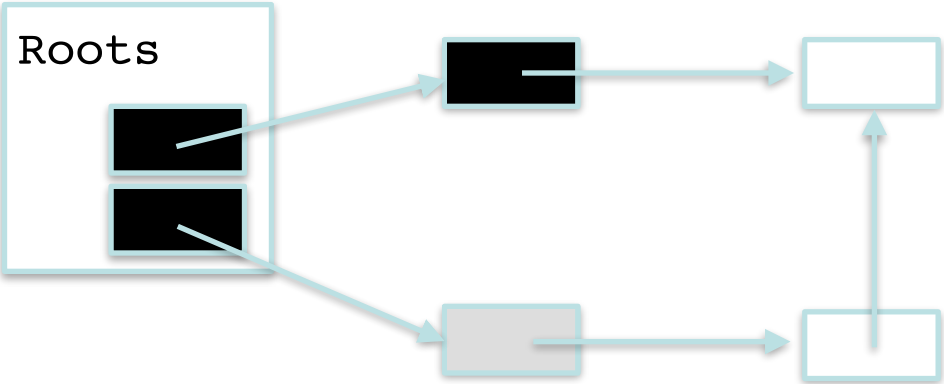
Deletion barrier [Abraham & Patel; Yuasa]



Advance the wavefront: shade the deleted target

Weak invariant allows mutator to hold white roots

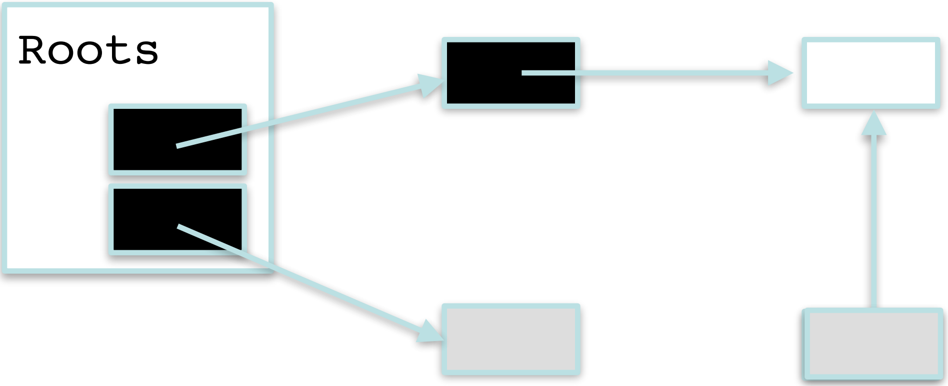
Deletion barrier [Abraham & Patel; Yuasa]



Advance the wavefront: shade the deleted target

Weak invariant allows mutator to hold white roots

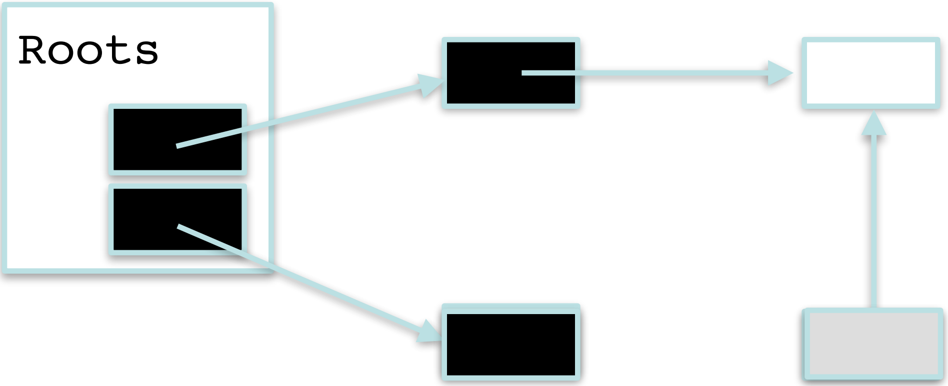
Deletion barrier [Abraham & Patel; Yuasa]



Advance the wavefront: shade the deleted target

Weak invariant allows mutator to hold white roots

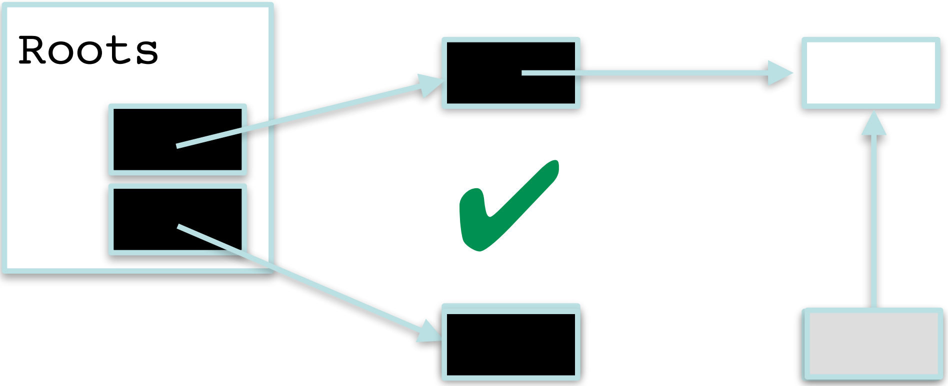
Deletion barrier [Abraham & Patel; Yuasa]



Advance the wavefront: shade the deleted target

Weak invariant allows mutator to hold white roots

Deletion barrier [Abraham & Patel; Yuasa]



Advance the wavefront: shade the deleted target

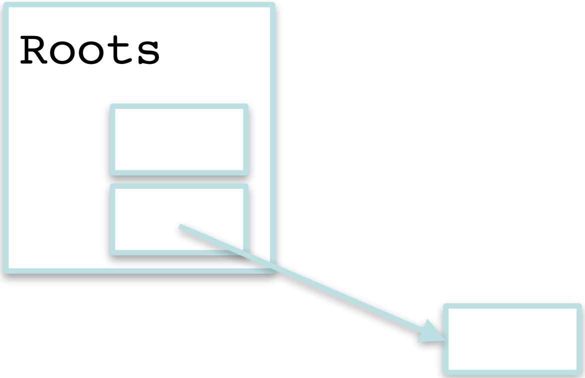
Weak invariant allows mutator to hold white roots

Termination

Eventually all live objects are black

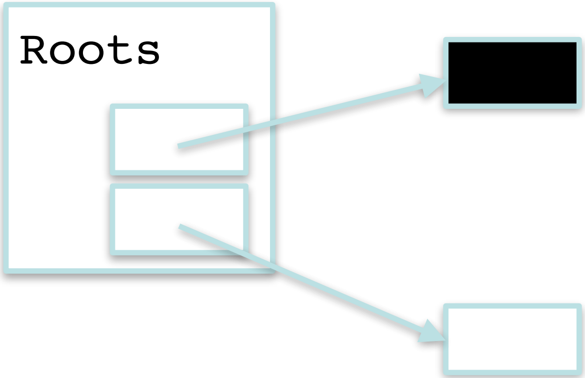
- Marking only adds to the wavefront
- Mutator barriers only add to the wavefront
- Allocating black speeds termination
- Deletion barrier avoids need to re-scan mutator roots
    - mutator can safely acquire white roots after snapshot
    - otherwise, must repeat marking from mutator roots until no white roots remain

Allocating black



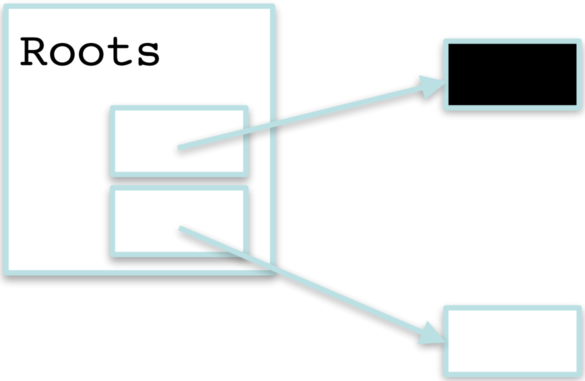
Deletion barrier is insufficient when allocating black.

Allocating black



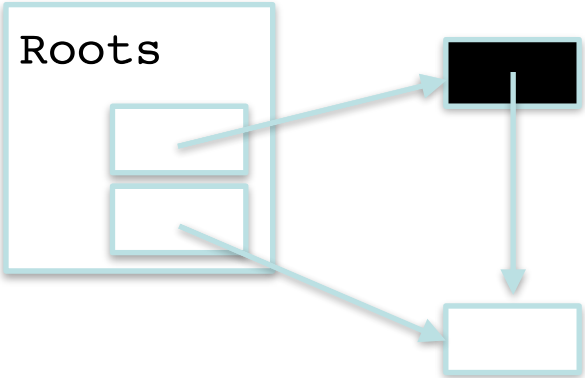
Deletion barrier is insufficient when allocating black.

Allocating black



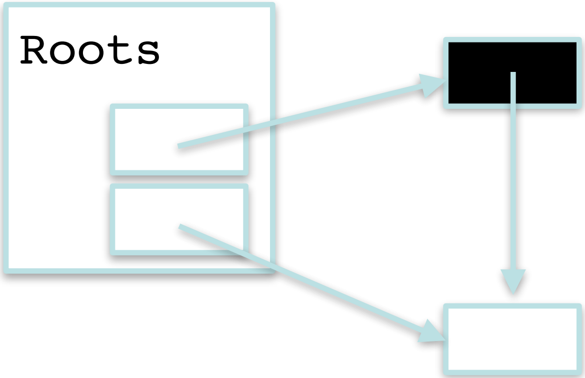
Deletion barrier is insufficient when allocating black.

Allocating black



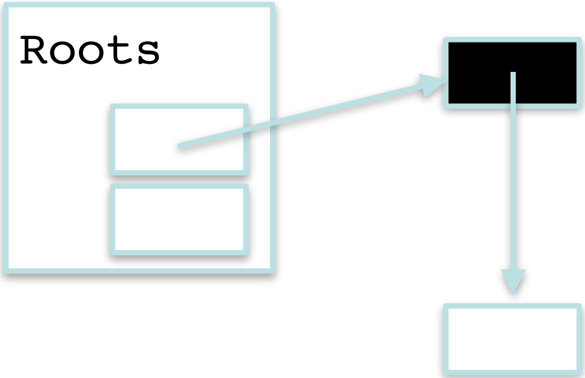
Deletion barrier is insufficient when allocating black.

Allocating black



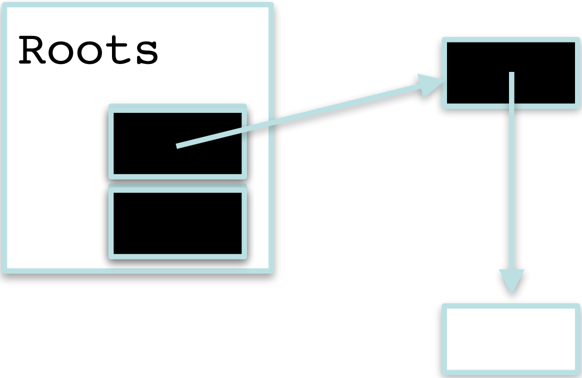
Deletion barrier is insufficient when allocating black.

Allocating black



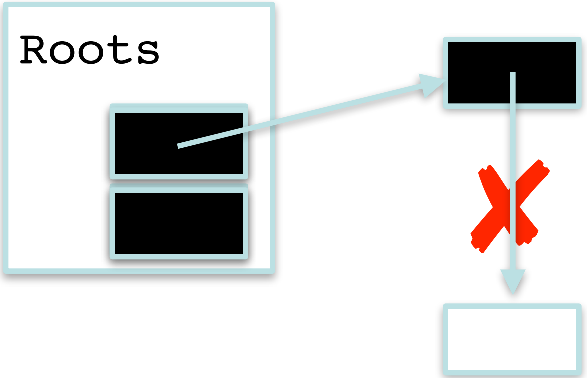
Deletion barrier is insufficient when allocating black.

Allocating black



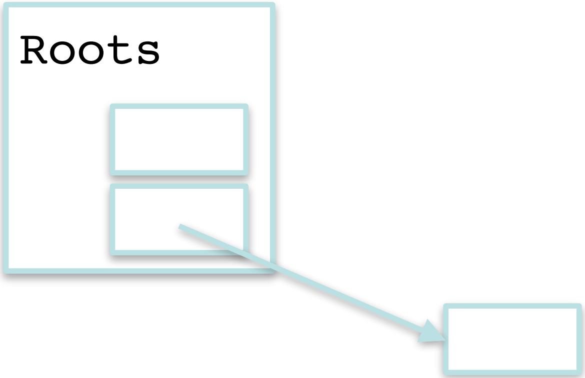
Deletion barrier is insufficient when allocating black.

Allocating black



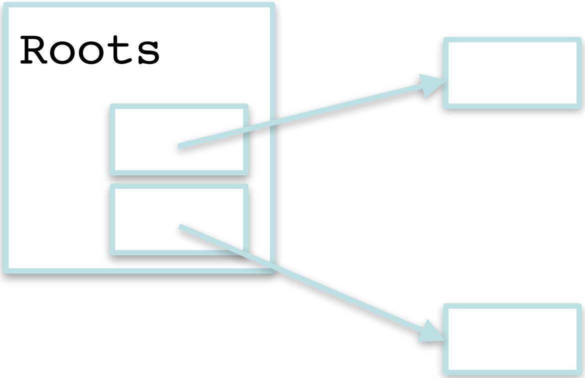
Deletion barrier is insufficient when allocating black.

Allocating white



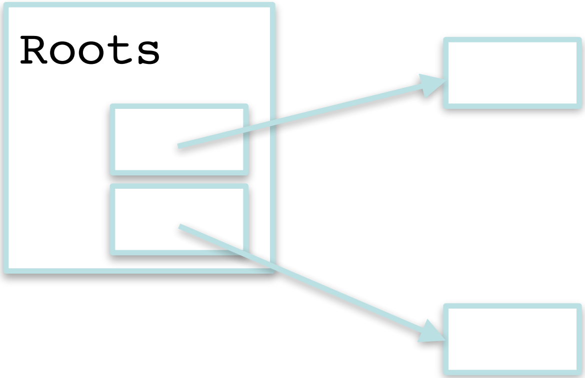
Deletion barrier suffices when allocating white.

Allocating white



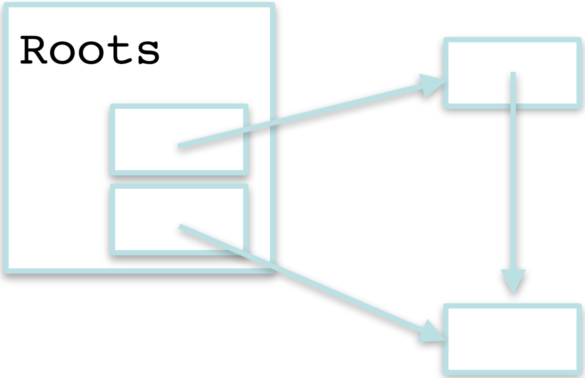
Deletion barrier suffices when allocating white.

Allocating white



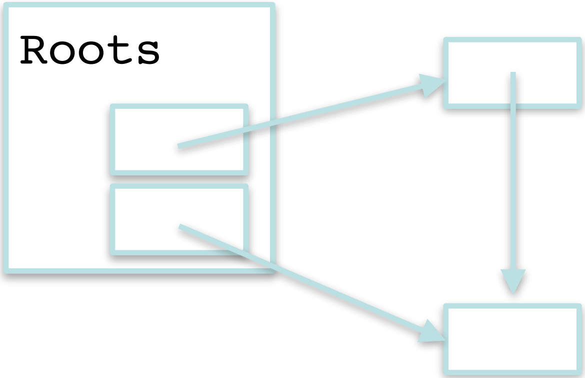
Deletion barrier suffices when allocating white.

Allocating white



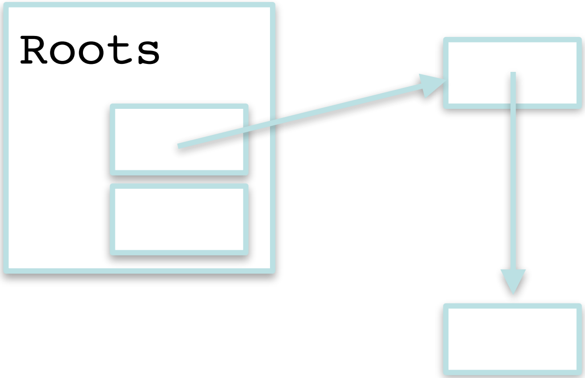
Deletion barrier suffices when allocating white.

Allocating white



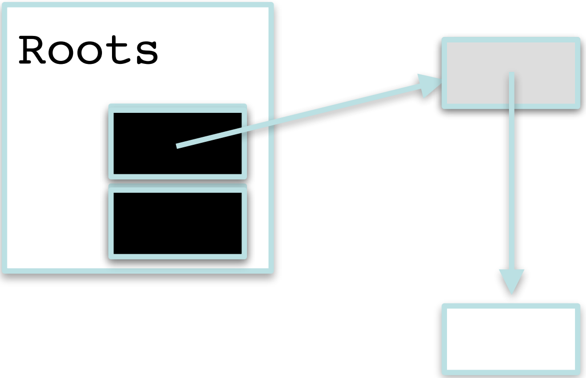
Deletion barrier suffices when allocating white.

Allocating white



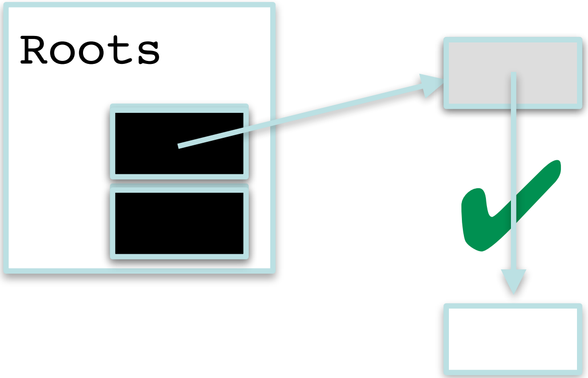
Deletion barrier suffices when allocating white.

Allocating white



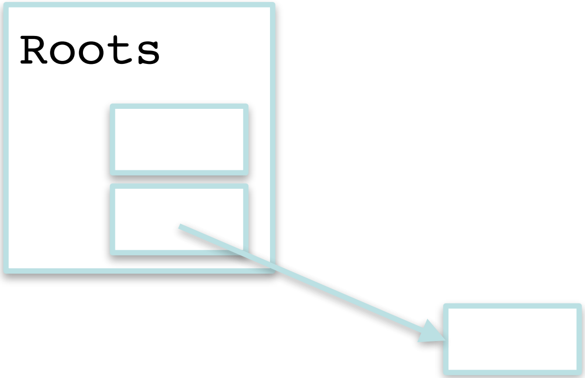
Deletion barrier suffices when allocating white.

Allocating white



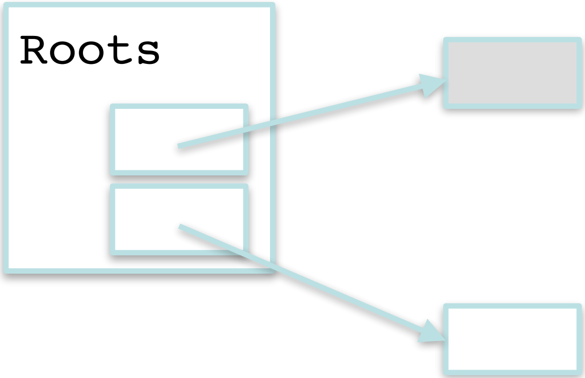
Deletion barrier suffices when allocating white.

Allocating grey



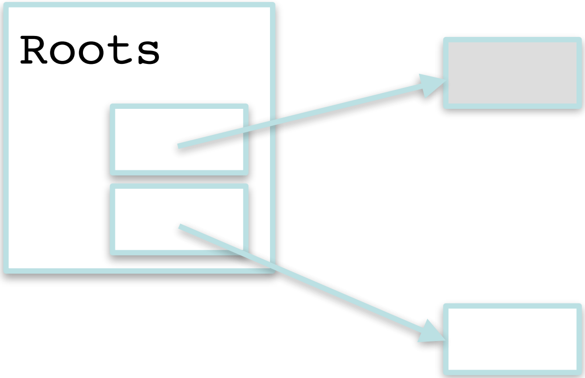
Deletion barrier suffices when allocating white/grey.

Allocating grey



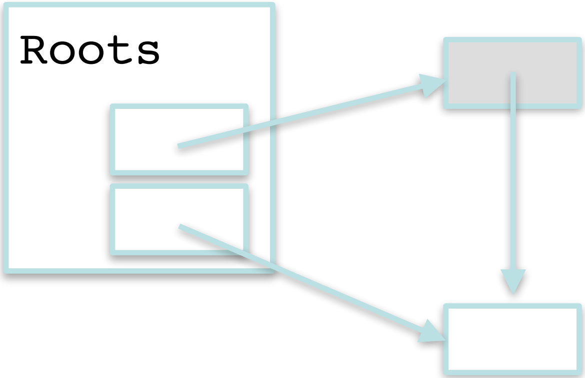
Deletion barrier suffices when allocating white/grey.

Allocating grey



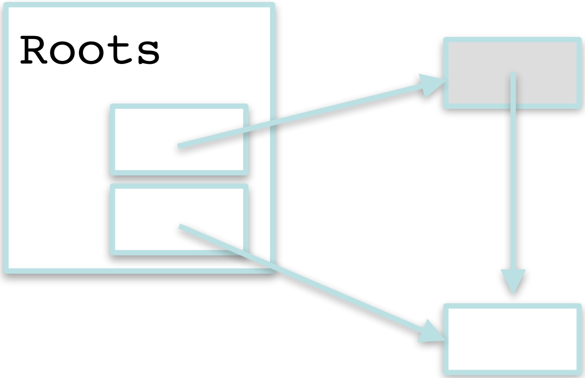
Deletion barrier suffices when allocating white/grey.

Allocating grey



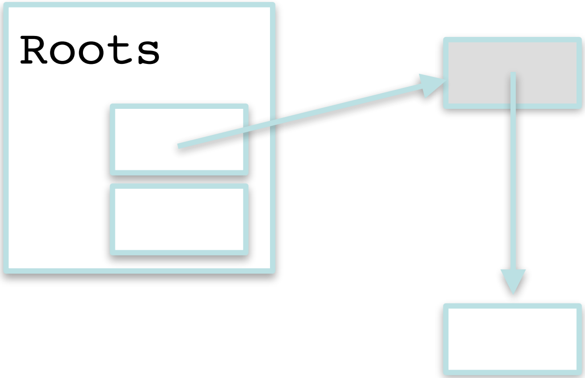
Deletion barrier suffices when allocating white/grey.

Allocating grey



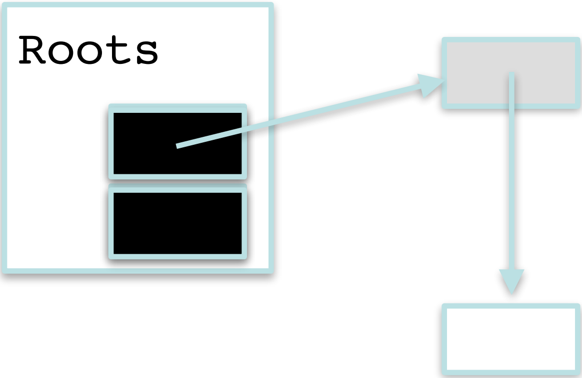
Deletion barrier suffices when allocating white/grey.

Allocating grey



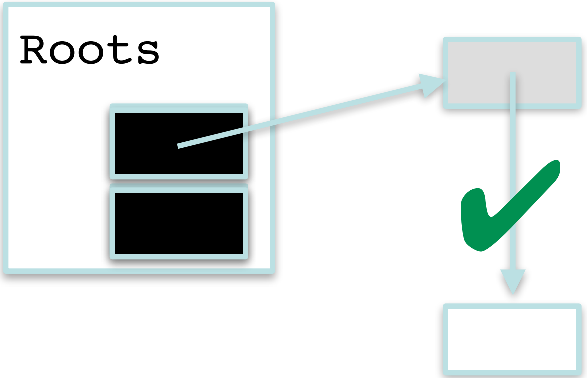
Deletion barrier suffices when allocating white/grey.

Allocating grey



Deletion barrier suffices when allocating white/grey.

Allocating grey



Deletion barrier suffices when allocating white/grey.

Safety

Arguments about safety of on-the-fly collectors trade on the weak or strong tricolor invariants enforced by the barriers.

BUT…

Can we really believe these arguments for real?

Can we prove safety for an efficient production algorithm that uses lightweight synchronization on x86 relaxed memory?

Segue from PLDI'10 to PLDI'15 and beyond

PROVING SAFETY

The bones of Schism: CMR [PLDI'10]

Concurrent mark-region GC

- Concurrent
- On-the-fly
- Wait-free constant-time heap access
- Mark-region allocator (linear-time for small objects or no fragmentation)
- Good throughput

CMR data structures

- `o.flag` field indicates if object `o` is marked
- `o.next` field is used to log `o` to a worklist
- `o.len` field object `o` is marked
- `W` is the collector worklist
- `W[t]` is a private worklist for mutator `t`

Mutator primitives

- `Read(`o`,`i`)`: load slot i from object o
- `Write(`o`,`i`,`v`)`: store v into slot i of o
- `Alloc(`n`,`v`)`: return a reference to a new object having n fields initialized to v
- Collector-mutator *handshakes* in between, never in the middle

CMR Write

```
Write(o,i,v):
 w := o[i]
 mark(w,W[self])
 mark(v,W[self])
 o[i] := v
```

CMR Alloc

```
Alloc(n,v):
  o := allocRaw(n)
  o.flag := fA
  o.next := null
  o.len := n
  for each l in Refs(o)
    *l := v
  return o
```

CMR Collector

```
loop
    phase ← Idle
    handshake t in T nop
    f_M ← !f_M
    handshake t in T nop
    phase ← Init
    handshake t in T nop
    phase ← Mark
    f_A ← f_M
    handshake t in T nop
    handshake t in T
        for each l in Roots(t)
            mark(*l, W[t])
        atomic transfer(W[t], W)
    while W.head ≠ null
        while W.head ≠ null
            s ← dequeue(W)
            shadeBlock(s,s.len)
            for each l in Refs(s)
                mark(*l, W)
        handshake t in T
            atomic transfer(W[t],
            W)
    phase ← Sweep
    sweepBlocksAndLines()
```

CMR Collector

```
loop
    phase ← Idle
    handshake t in T nop
    f_M  ← !f_M
    handshake t in T nop
    phase ← Init
    handshake t in T nop
    phase ← Mark
    f_A ← f_M
    handshake t in T nop
    handshake t in T
        for each l in Roots(t)
            mark(*l, W[t])
        atomic transfer(W[t], W)
    while W.head ≠ null
        while W.head ≠ null
            s ← dequeue(W)
            shadeBlock(s,s.len)
            for each l in Refs(s)
                mark(*l, W)
        handshake t in T
            atomic transfer(W[t],
            W)
    phase ← Sweep
    sweepBlocksAndLines()
```

Idle: no marking

CMR Collector

```
loop
    phase ← Idle
    handshake t in T nop
    f_M ← !f_M
    handshake t in T nop
    phase ← Init
    handshake t in T nop
    phase ← Mark
    f_A ← f_M
    handshake t in T nop
    handshake t in T
        for each l in Roots(t)
            mark(*l, W[t])
        atomic transfer(W[t], W)
    while W.head ≠ null
        while W.head ≠ null
            s ← dequeue(W)
            shadeBlock(s,s.len)
            for each l in Refs(s)
                mark(*l, W)
        handshake t in T
            atomic transfer(W[t],
            W)
    phase ← Sweep
    sweepBlocksAndLines()
```

CMR Collector

```
loop
    phase ← Idle
    handshake t in T nop
    fM ← !fM
    handshake t in T nop
    phase ← Init
    handshake t in T nop
    phase ← Mark
    fA ← fM
    handshake t in T nop
    handshake t in T
        for each l in Roots(t)
            mark(*l, W[t])
        atomic transfer(W[t], W)
    while W.head ≠ null
        while W.head ≠ null
            s ← dequeue(W)
            shadeBlock(s, s.len)
            for each l in Refs(s)
                mark(*l, W)
        handshake t in T
            atomic transfer(W[t],
            W)
    phase ← Sweep
    sweepBlocksAndLines()
```

Init: Store marks

CMR Collector

```
loop
    phase ← Idle
    handshake t in T nop
    fM  ← !fM
    handshake t in T nop
    phase ← Init
    handshake t in T nop
    phase ← Mark
    fA ← fM
    handshake t in T nop
    handshake t in T
        for each l in Roots(t)
            mark(*l, W[t])
        atomic transfer(W[t], W)
    while W.head ≠ null
        while W.head ≠ null
            s ← dequeue(W)
            shadeBlock(s,s.len)
            for each l in Refs(s)
                mark(*l, W)
        handshake t in T
            atomic transfer(W[t],
            W)
    phase ← Sweep
    sweepBlocksAndLines()
```

CMR Collector

```
loop
      phase ← Idle
      handshake t in T nop
      fM  ← !fM
      handshake t in T nop
      phase ← Init
      handshake t in T nop
      phase ← Mark
      fA ← fM
      handshake t in T nop
      handshake t in T
          for each l in Roots(t)
              mark(*l, W[t])
          atomic transfer(W[t], W)
      while W.head ≠ null
          while W.head ≠ null
              s ← dequeue(W)
              shadeBlock(s,s.len)
              for each l in Refs(s)
                  mark(*l, W)
          handshake t in T
              atomic transfer(W[t],
              W)
      phase ← Sweep
      sweepBlocksAndLines()
```

Mark: Store &
Alloc mark

CMR Collector

```
loop
    phase ← Idle
    handshake t in T nop
    fM ← !fM
    handshake t in T nop
    phase ← Init
    handshake t in T nop
    phase ← Mark
    fA ← fM
    handshake t in T nop
    handshake t in T
        for each l in Roots(t)
            mark(*l, W[t])
        atomic transfer(W[t], W)
    while W.head ≠ null
        while W.head ≠ null
            s ← dequeue(W)
            shadeBlock(s,s.len)
            for each l in Refs(s)
                mark(*l, W)
        handshake t in T
            atomic transfer(W[t],
            W)
    phase ← Sweep
    sweepBlocksAndLines()
```

CMR Collector

```
loop
      phase ← Idle
      handshake t in T nop
      fM ← !fM
      handshake t in T nop
      phase ← Init
      handshake t in T nop
      phase ← Mark
      fA ← fM
      handshake t in T nop
      handshake t in T
          for each l in Roots(t)
              mark(*l, W[t])
          atomic transfer(W[t], W)
      while W.head ≠ null
          while W.head ≠ null
              s ← dequeue(W)
              shadeBlock(s,s.len)
              for each l in Refs(s)
                  mark(*l, W)
          handshake t in T
              atomic transfer(W[t], W)
      phase ← Sweep
      sweepBlocksAndLines()
```

Sweep: Alloc marks

CMR Collector

```
loop
    phase ← Idle
    handshake t in T nop
    f_M  ← !f_M
    handshake t in T nop
    phase ← Init
    handshake t in T nop
    phase ← Mark
    f_A ← f_M
    handshake t in T nop
    handshake t in T
        for each l in Roots(t)
            mark(*l, W[t])
        atomic transfer(W[t], W)
    while W.head ≠ null
        while W.head ≠ null
            s ← dequeue(W)
            shadeBlock(s,s.len)
            for each l in Refs(s)
                mark(*l, W)
        handshake t in T
            atomic transfer(W[t],
            W)
    phase ← Sweep
    sweepBlocksAndLines()
```

CMR Mark

```
mark(o,W):
  if o.flag ≠ fM
    if phase ≠ Idle
      if CAS(&o.flag, !fM → fM) = !fM
        o.next := W.head
        if W.tail = null
          W.tail := o
        W.head := o
```

CMR Transfer

```
transfer(W1, W2):
 if W1.head ≠ null
  W1.tail.next := W2.head
  W2.head := W1.head
  W1.head := null
  W1.tail := null
```

CMR Dequeue

```
dequeue(W):
  o ← W.head
  if o = null
    return null
  W.head ← o.next
  if W.tail = o
    W.tail = null
  return o
```

Relaxing Safely: Verified On-the-Fly Garbage Collection for x86-TSO

Peter Gammie[1] Tony Hosking[2] Kai Engelhardt[3]

[1]ex NICTA Australia

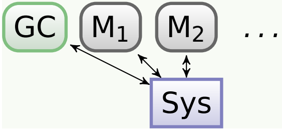
[2]Purdue University while on leave at NICTA

[3]UNSW and NICTA Australia

Tony's Goal

Verify a highly concurrent, on-the-fly garbage collector wrt a non-sequentially consistent memory model

Our headline result



$$\models \Box(\forall r.\text{reachable } r \rightarrow \text{valid\_ref } r)$$

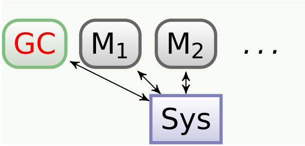
For Sys $\simeq$ TS0, *there is always an object at every reference reachable from a mutator root*

Main challenges

- the system is **highly concurrent**
- memory is not **sequentially consistent**
- mutators may not be **data-race free**
- **convince** of our proof technique/invariants

The Schism Garbage Collector

as described at PLDI'10

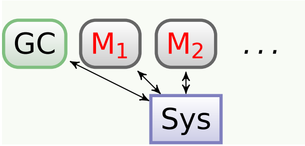


Concurrent, on-the-fly, mark-sweep

- Does **not stop the world**
  - wait-free for the mutators up to allocation
- Takes a **snapshot**
- Races with the mutators to mark reachable objects
- Objects are built from cache-line-sized chunks
  - representation bakes in GC space overhead
  - $\Rightarrow$ fragmentation tolerant without relocation

Claim: has **predictable real-time performance**

Folklore: GC requires mutator cooperation



Mutators use **write barriers**
unless the GC is idle

- the **insertion barrier** prevents mutators from hiding references from the collector
- the **deletion barrier** accelerates termination

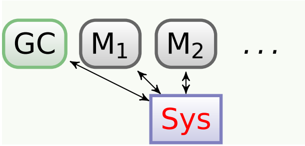
Objects are allocated. . .

- **white** when GC is **idle**
  - minimizes **floating garbage**
- **black** when GC is **not idle**
  - accelerates termination

Grey objects track pending GC work

Total Store Order (x86-TSO)

x86-TSO has **one write queue per core** and **consults these queues for memory reads**



Some instructions (CAS) **are an atomic sequence memory operations**, so there is also a **bus lock**

x86-TSO is **not sequentially consistent**
. . . there can be **data races**

- The model has been **validated against recent x86 hardware** by Peter Sewell et al.

Data races

Starting in a state where $x = y = 0$:

P_0	P_1
$x \leftarrow 1$	$y \leftarrow 1$
$r_1 \leftarrow y$	$r_2 \leftarrow x$

Can observe $r_1 = r_2 = 0$

a **data race**

Data races

Starting in a state where $x = y = 0$:

P_0	P_1
$x \leftarrow 1$	$y \leftarrow 1$
`MFENCE`	`MFENCE`
$r_1 \leftarrow y$	$r_2 \leftarrow x$

`MFENCE` waits until the core's TSO buffer is empty

Now **cannot** observe $r_1 = r_2 = 0$

We have sequential consistency!

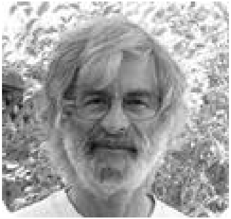
. . . with significantly degraded performance

The promise of many reduction theorems

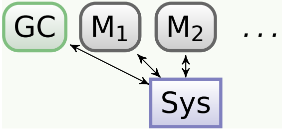
If there are **no data races**, then we can use **classical techniques** for sequentially consistent memory

Problem: our mutators need not be **DRF**

Solution: resort to general techniques for concurrency



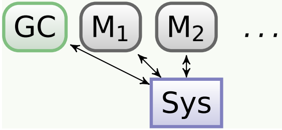
So much for the model... what about the assertion?



$$\models \Box(\forall r.\,\text{reachable}\ r \rightarrow \text{valid\_ref}\ r)$$

For Sys $\simeq$ TS0*, there is always an object at every reference reachable from a mutator root*

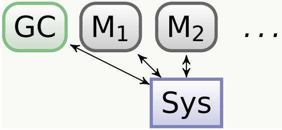
So much for the model... what about the assertion?



$$\models \Box(\forall r.\,\mathrm{reachable}\ r \longrightarrow \mathrm{valid\_ref}\ r)$$

For Sys $\simeq$ TS0, *there is always an object at every reference reachable from a mutator root*

So much for the model... what about the assertion?

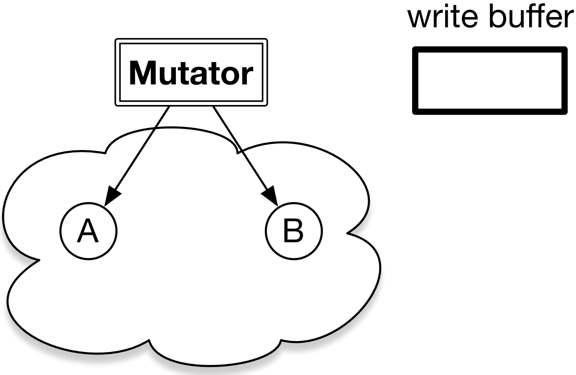


$$\models \Box(\forall r.\,\text{reachable } r \longrightarrow \text{valid\_ref } r)$$

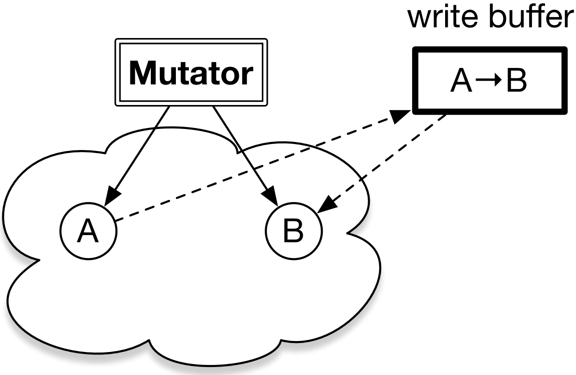
For Sys $\simeq$ TSO, *there is always an object at every reference reachable from a mutator root*

What does TSO do to reachability?

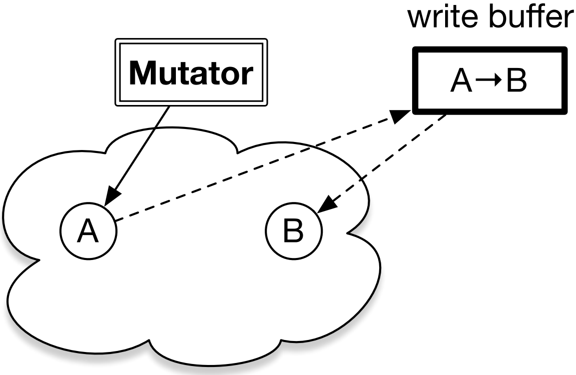
TSO and reachability, breathlessly



TSO and reachability, breathlessly

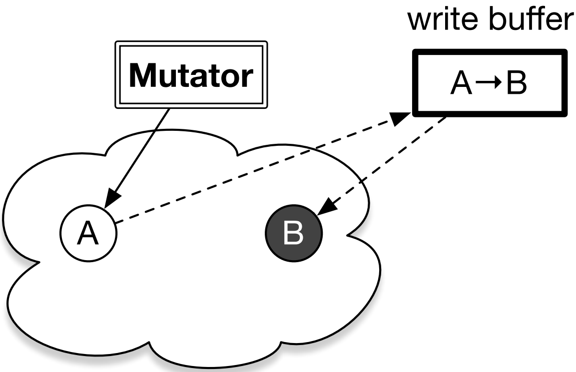


TSO and reachability, breathlessly



The pending write is the only witness to B's reachability

TSO and reachability, breathlessly

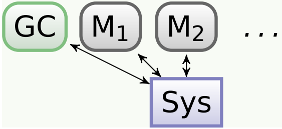


. . . but the **insertion barrier has already greyed B!**

We treat grey and TSO refs as **extra mutator roots**

. . . and **all paths go via the heap**

So much for the result. . . how did we express all that?



$\models \Box(\forall r.\,\text{reachable}\ r \to \text{valid\_ref}\ r)$

For Sys $\simeq$ TS0*, there is always an object at every reference reachable from a mutator root*

“You can’t always write a chord ugly enough to say what you want to say, so sometimes you have to rely on a giraffe filled with whipped cream.”

The framework: CIMP

- Simple imperative language (`IMP`) + synchronous message passing
- Flat, top-level parallel composition.
- Each process has local control and data states

  There is **no shared global state**
- Amenable to **state-based invariant reasoning**

Enables **separation** of programs, invariants and proofs

Constructing the GC invariants

Tony was confident that the GC satisfied the **standard mark/sweep tricolour invariants**

. . . but before we made it to those lofty realms, we first:

- Defined a system-wide **“program counter”** that encodes the GC’s phase structure
- **Exploited DRF** where we could, and sliced the program wrt non-DRF variables when we had to
- Developed **fine-grained assertions** around the **object marking operations**

. . . in higher-order logic

Isabelle’s parallelism for low-latency invariant search!

Are you advocating this for arbitrary programs?

No

This was about discovering invariants for a particular non-trivial program on x86-TSO

Why didn’t you develop or use. . .

- a suitable reduction theorem?
- something compositional, like rely/guarantee?
- separation logic and ownership?
- shared variables and Owicki/Gries?
- communication-closed layers?
- a more abstract model and refinement?
- some other formalism such as I/O automata?
- proof outlines?
- a modelchecker?
- . . . your favoured technique?

Why didn't you develop or use...

- a suitable reduction theorem?
- something compositional, like rely/guarantee?
- separation logic and ownership?
- shared variables and Owicki/Gries?
- communication-closed layers?
- a more abstract model and refinement?
- some other formalism such as I/O automata?
- proof outlines?
- a modelchecker?
- ... your favoured technique?

Largely because (almost) none of these would have eased invariant discovery.

OK, so where did you cheat?

Minor:

- grey is not subject to TSO
- underlying memory blocks are not modelled
- ragged safepoints are abstract
- each mutator + GC runs on its own core

⇒ We leave these to an atomicity refinement technique.

Major:

- `alloc` and `free` are global and atomic.

Remaining and future work

- A defensible treatment of allocation
- Connection with something more executable
  - x86 instruction semantics + refinement
- Recast in a compositional framework
  - extra invariants: the heap is not mutilated, ...

A plausible liveness result may prove elusive.

Come and talk to us if any of that interests you!

Concluding remarks

- Reasoning about data-racy programs on x86-TSO is (sometimes) not too bad... but that might be because the collector is carefully constructed
- To scale one certainly wishes to exploit the general absence of data races
- The x86-TSO model is useful for thinking about correctness but not performance
  - How do we provide analytic WCET bounds for this putatively real-time collector?
- ARM/POWER are more complex