ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Supporting Parallelism in Operating Systems & Programming Languages

Garbage Collection

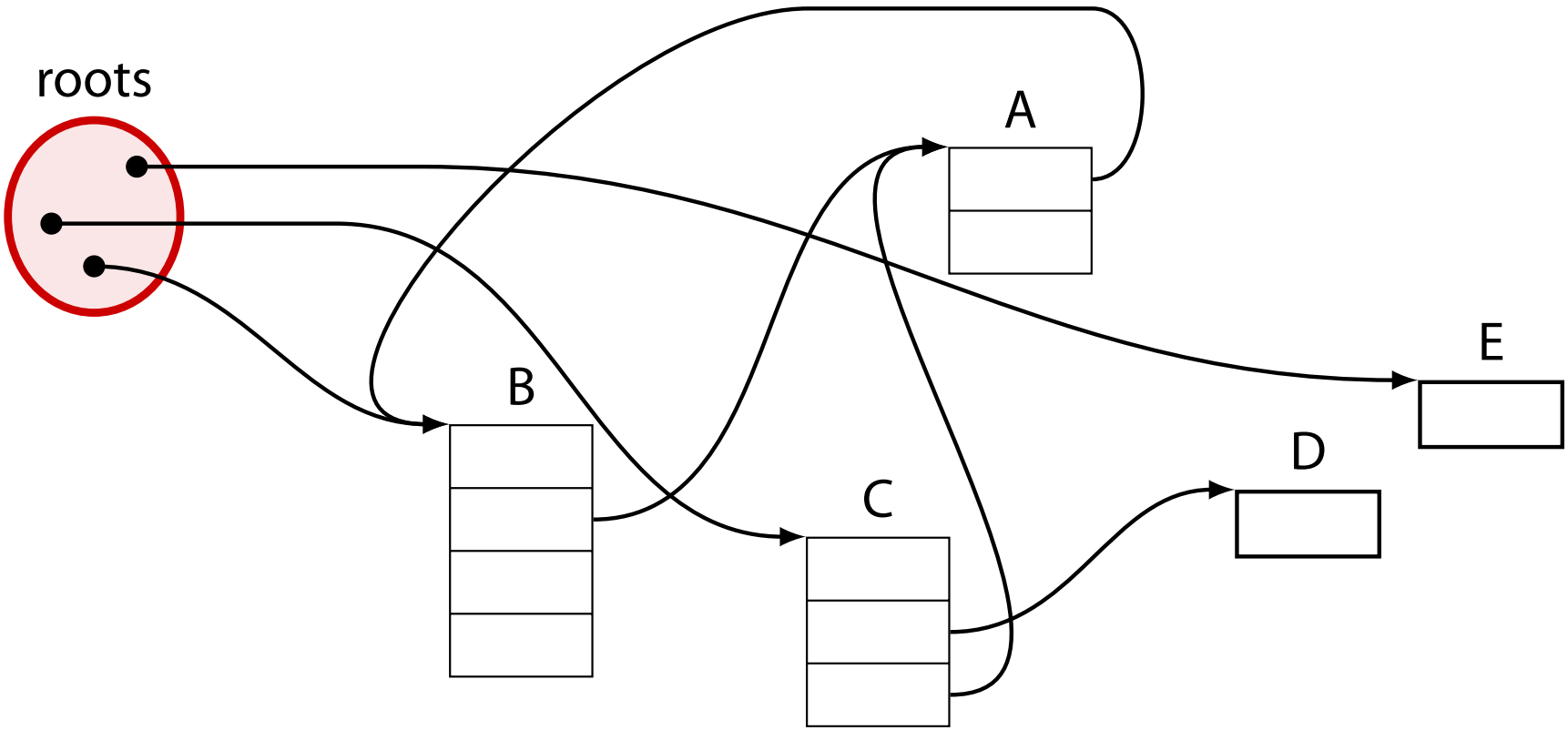
Kornilios Kourtis
<kornilios.kourtis@inf.ethz.ch>

Introduction

Parallel Garbage Collection

Concurrent Garbage Collection

Object graph



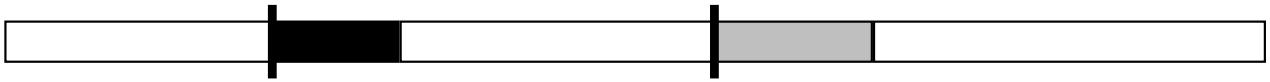
- root (e.g., stack variables)
- objects, reference fields

Terminology

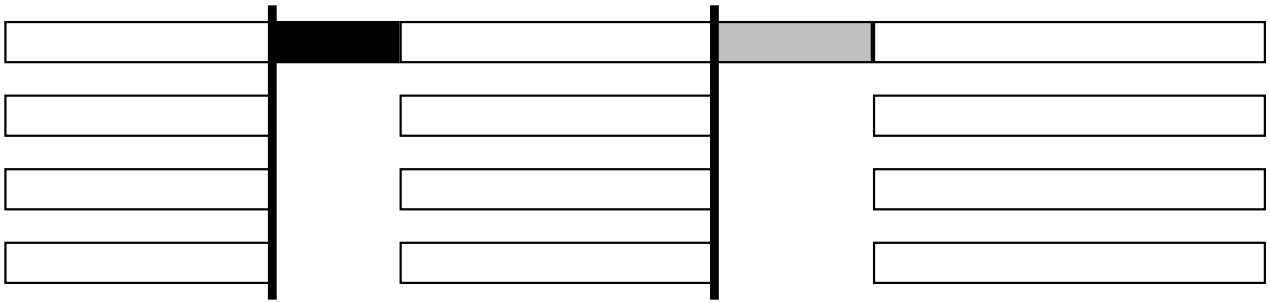
- Heap
- mutator
  - allocates objects
  - changes reference fields
  - changes root (add/remove/edit)
- collector
  - discovers and reclaims unreachable objects

Stop-the-world GC

one mutator thread:



multiple mutator threads:



- We assume stop-the-world for now

Tracing GC

- traverse object graph
- find unreachable objects
- reclaim them

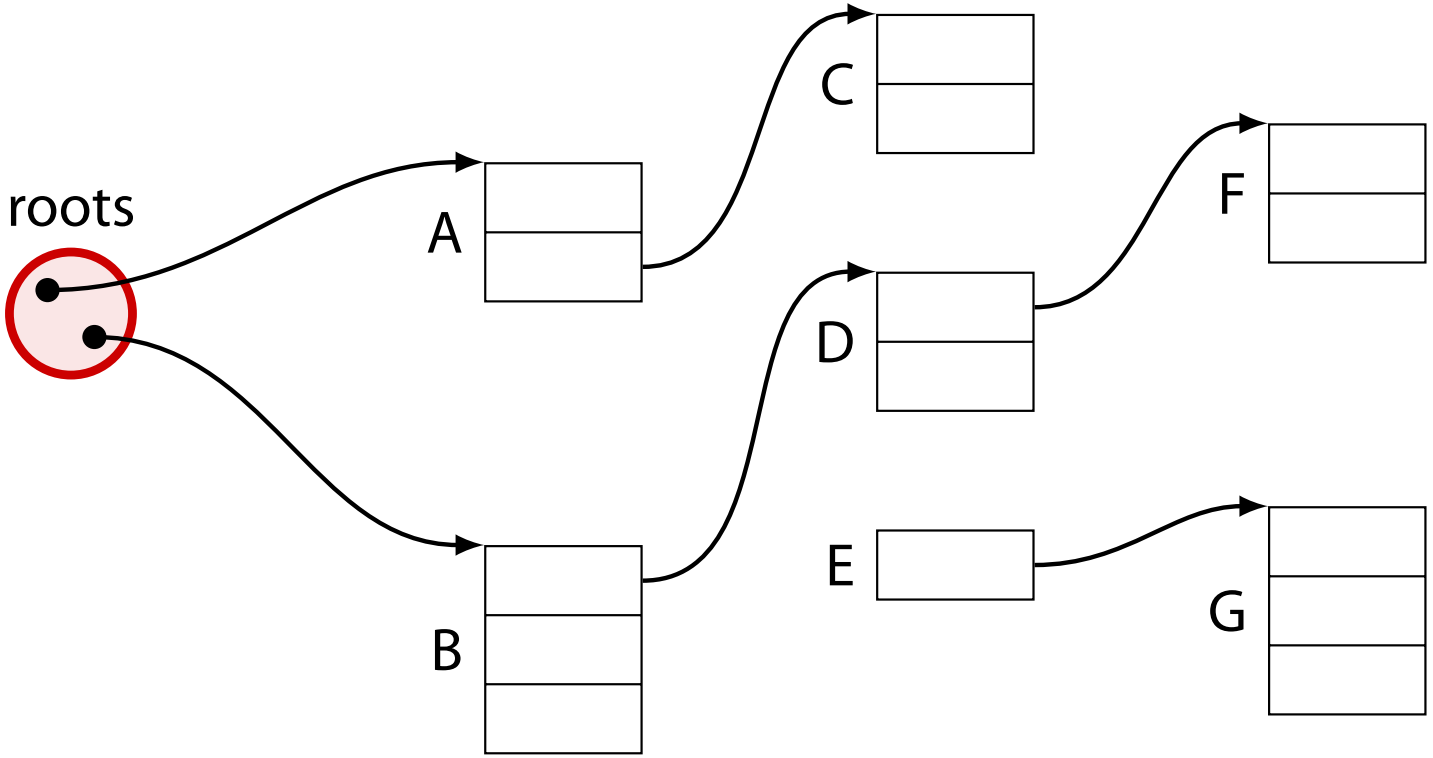
Algorithms:

- mark-sweep
- mark-compact
- copying

non-tracing GC:

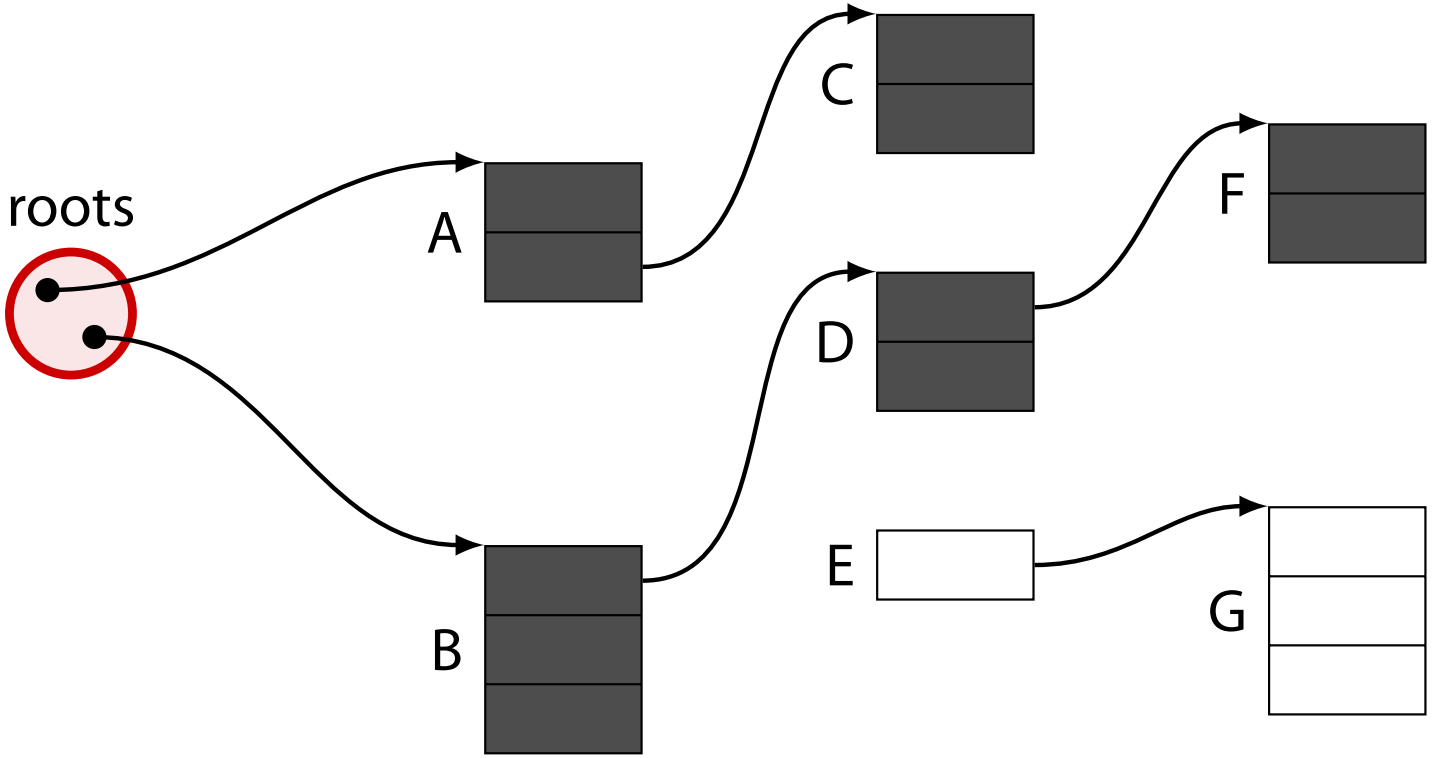
- reference counting

Reachability



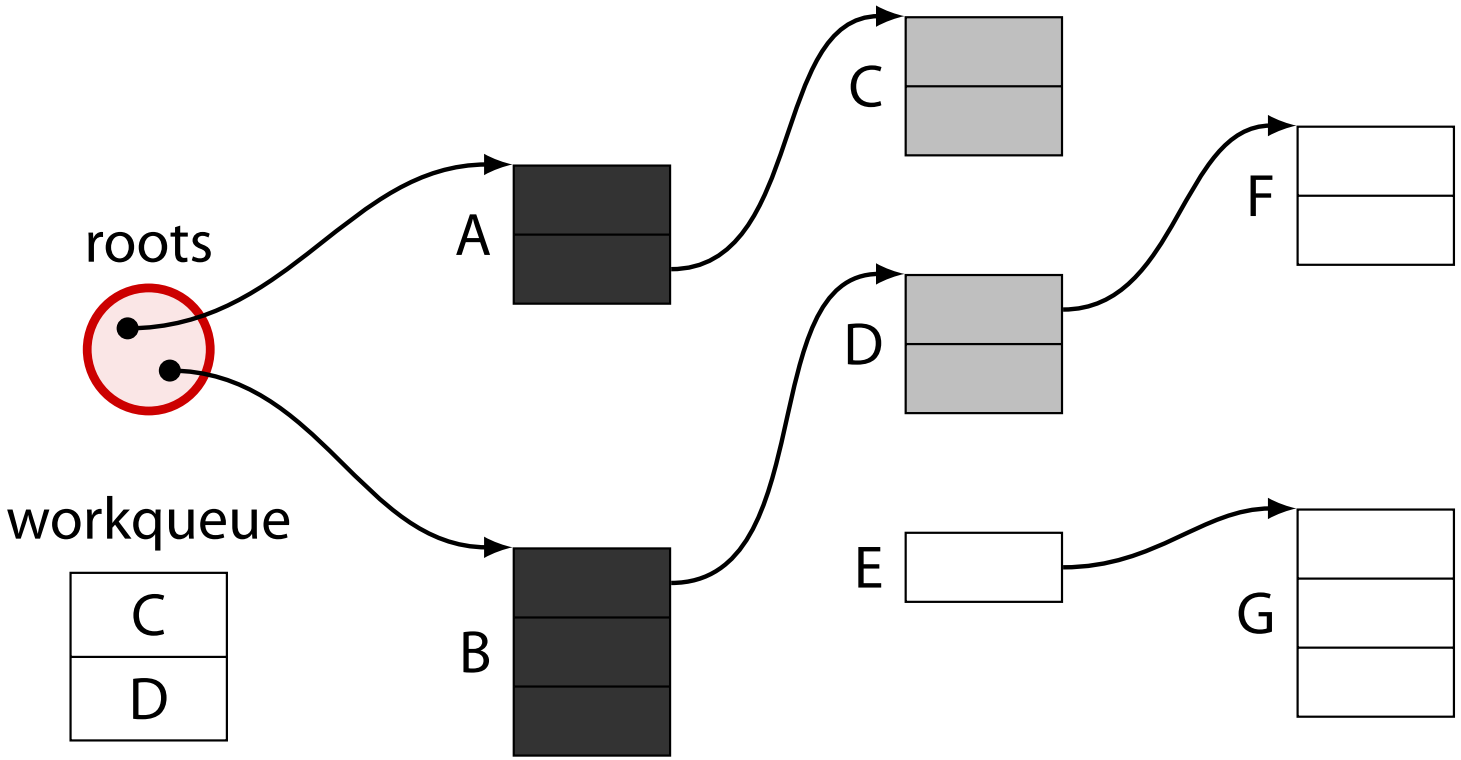
- rechability: path from mutator roots to object
- liveness: object will be accesses in the future (generally undecidable, found in compilers)

Reachability



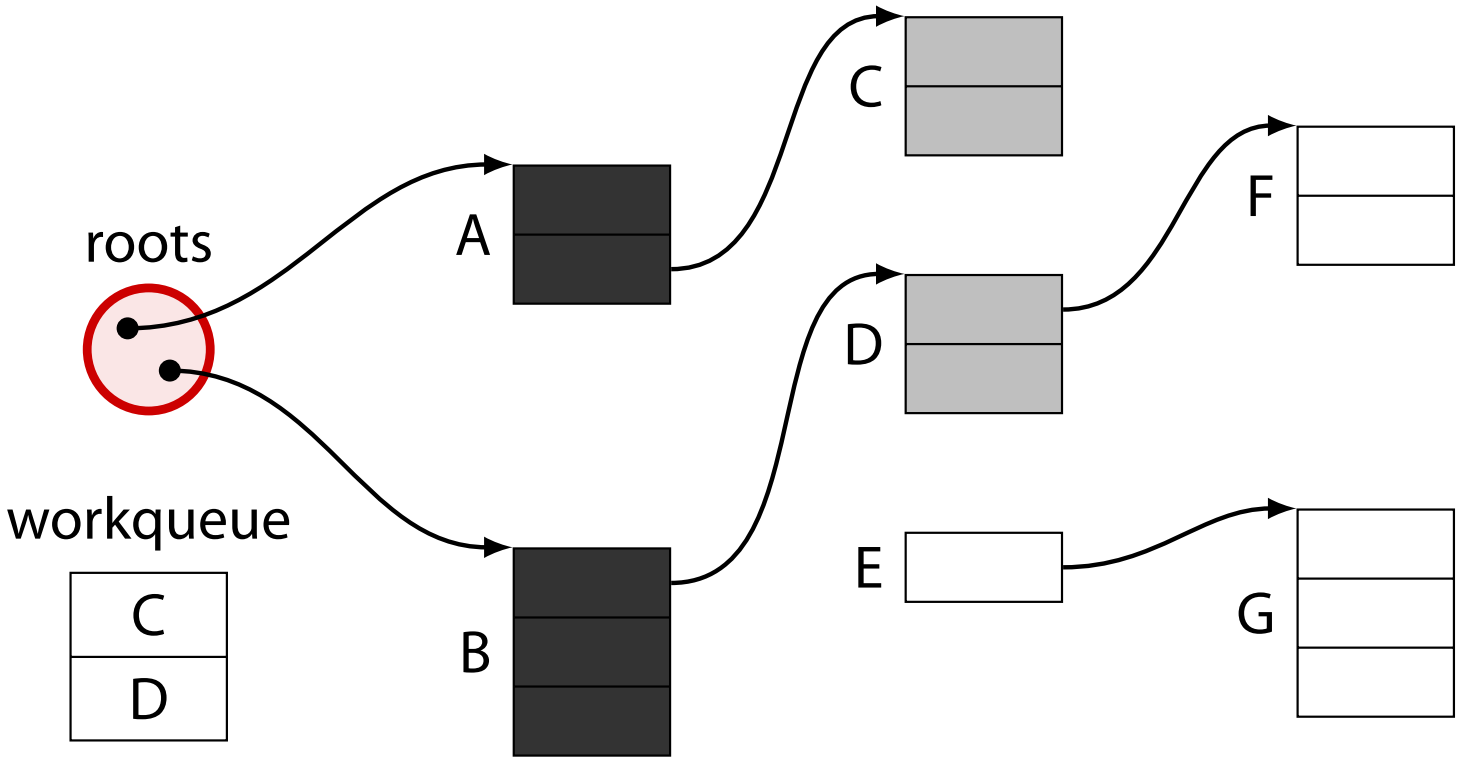
- rechability: path from mutator roots to object
- liveness: object will be accesses in the future (generally undecidable, found in compilers)

Tracing: tricolour abstraction



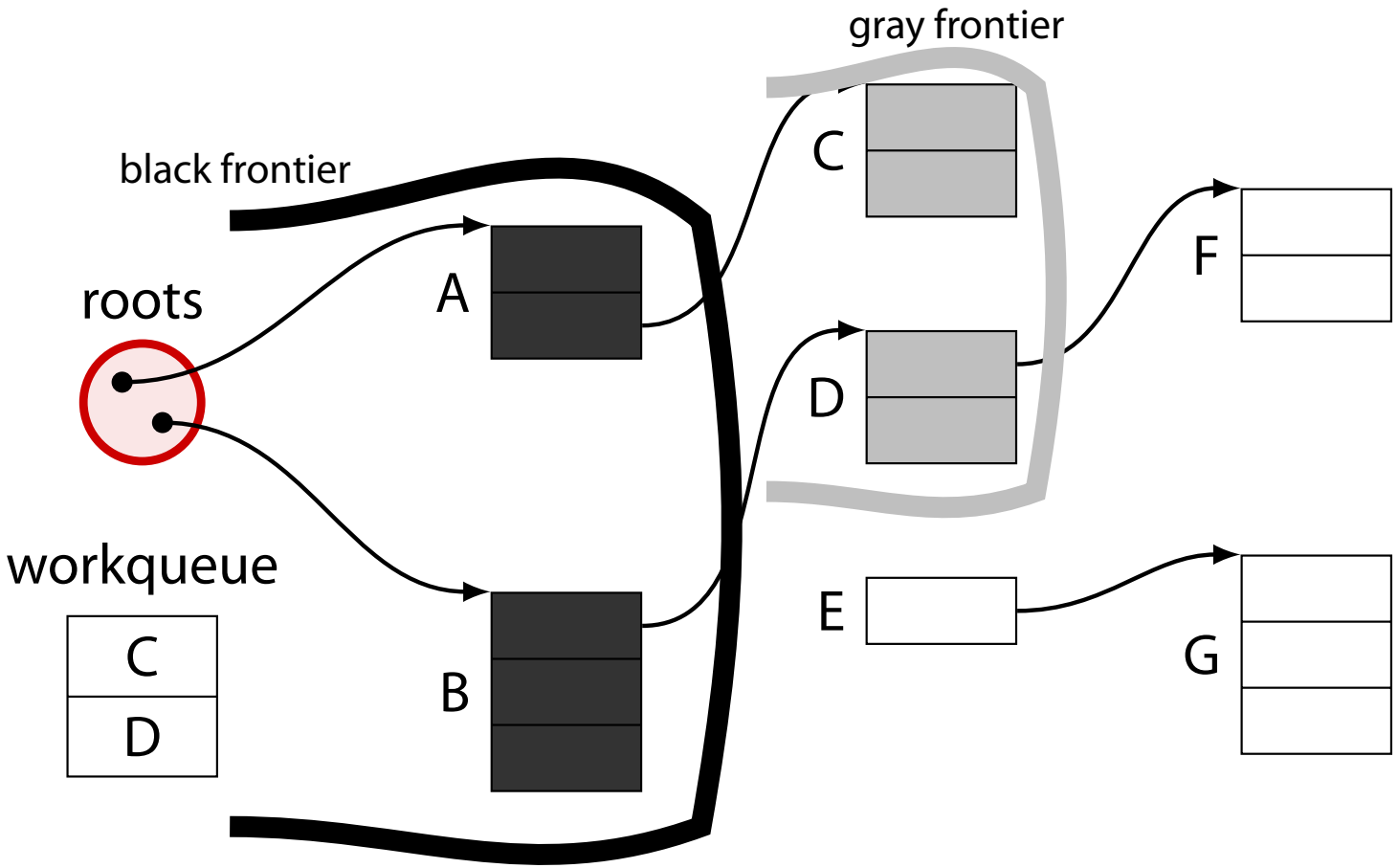
- why do we record state on objects?

Tracing: tricolour abstraction



- why do we record state on objects? circles!

Tracing: tricolour abstraction



- why do we record state on objects? circles!

Mark-sweep GC

Mark

- mark roots
- traverse object graph starting from roots

Sweep

- iterate all objects
- if object is marked, unmarked it
- else, free object

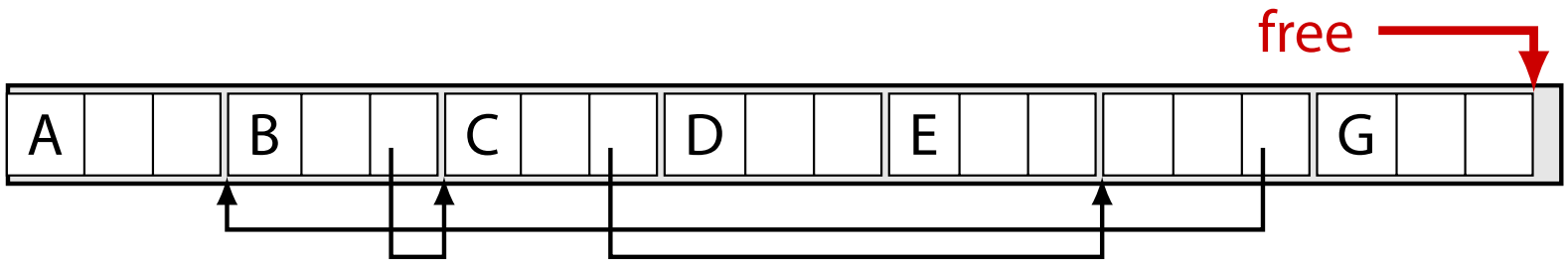
Mark-Compact GC

- allocation: bump a pointer
- GC: mark, compact data and update references

LISP2: marking + 3 phases

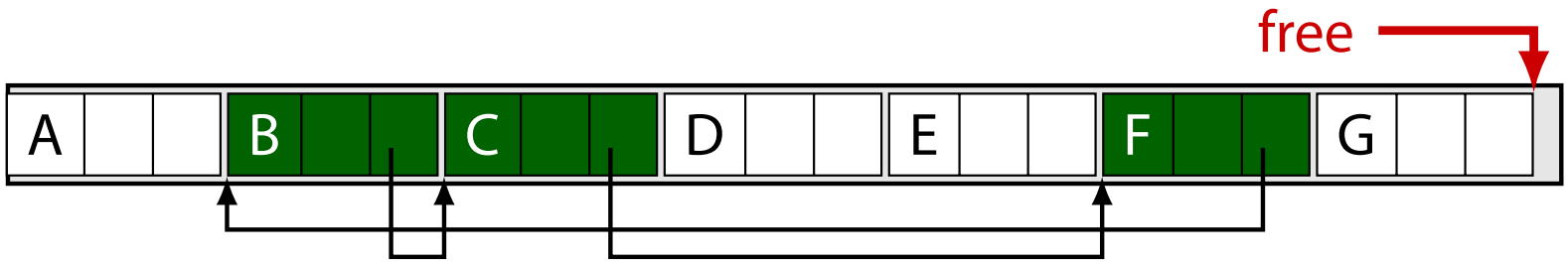
- compute forwarding addresses
- update pointers
- relocate blocks

LISP2 Mark-Compact GC: Example



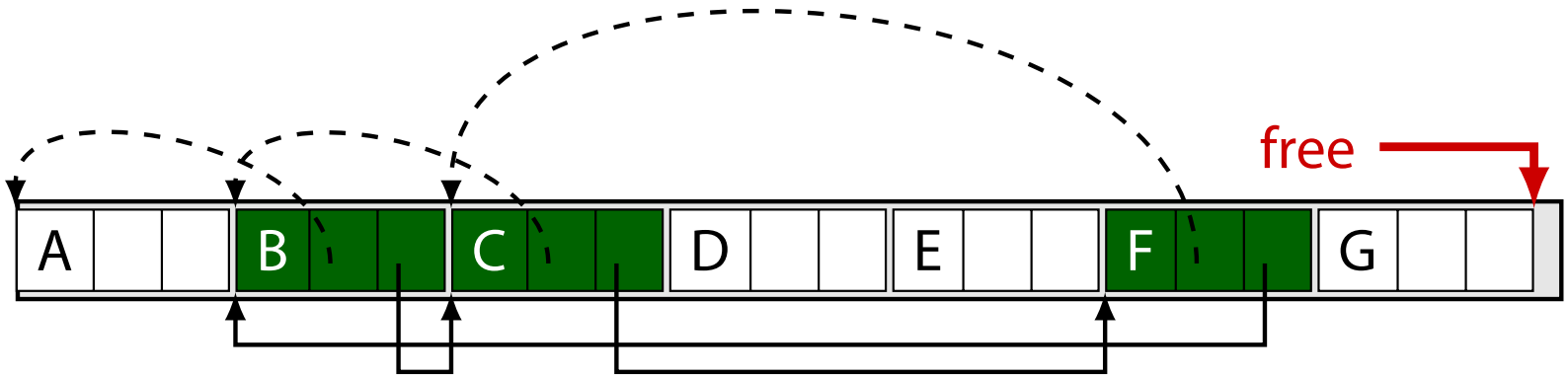
- allocation failed

LISP2 Mark-Compact GC: Example



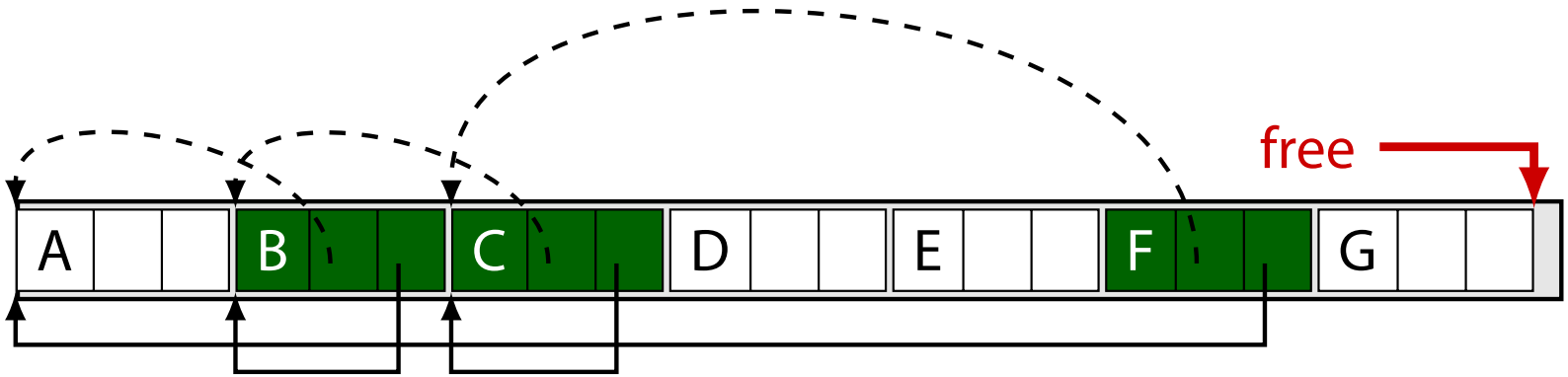
- allocation failed
- mark

LISP2 Mark-Compact GC: Example



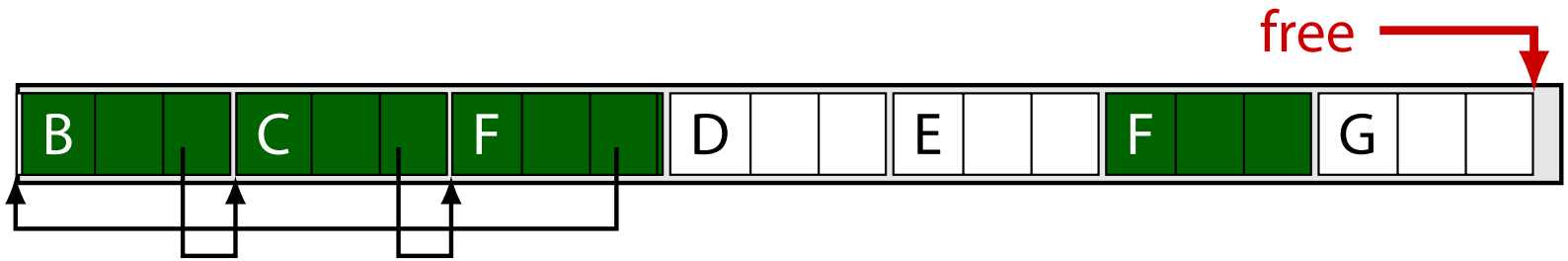
- allocation failed
- mark
- compute forwarding addresses

LISP2 Mark-Compact GC: Example



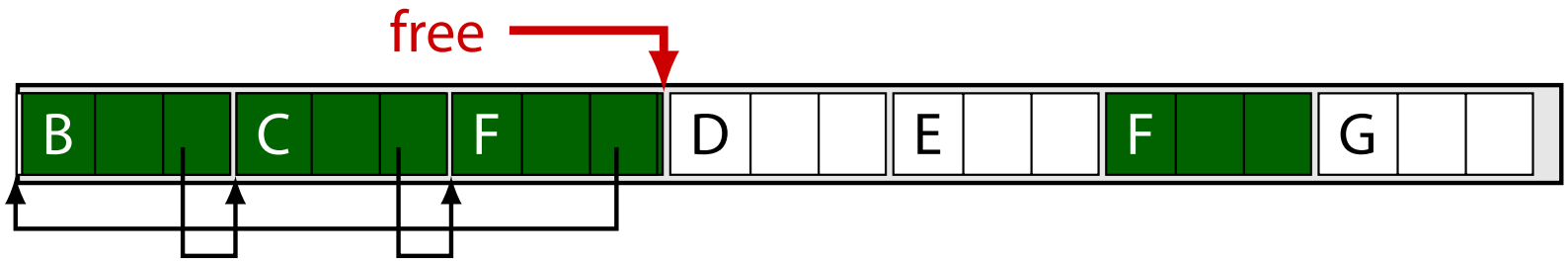
- allocation failed
- mark
- compute forwarding addresses
- update pointers

LISP2 Mark-Compact GC: Example



- allocation failed
- mark
- compute forwarding addresses
- update pointers
- relocate blocks

LISP2 Mark-Compact GC: Example



- allocation failed
- mark
- compute forwarding addresses
- update pointers
- relocate blocks
- move pointer to last object

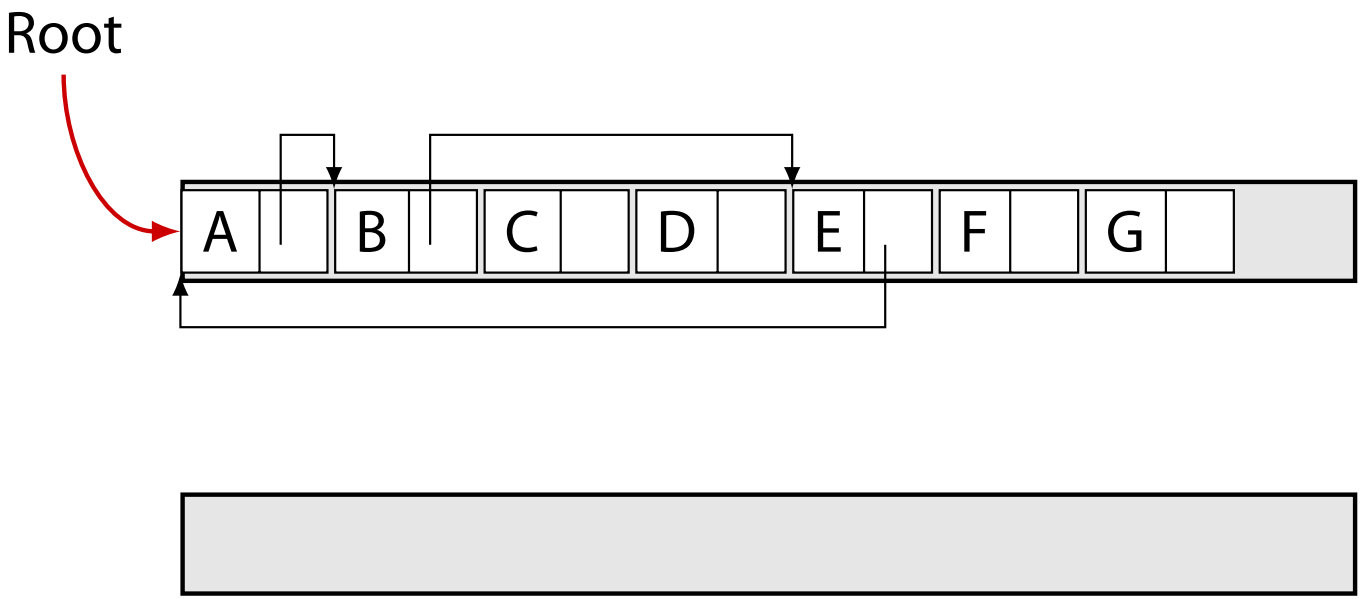
Copying GC

- space is divided in two semispaces:
  - *fromspace*
  - *tospace*
- allocation: bump a pointer in *fromspace*
- GC: copy live objects from *fromspace* to *tospace*
- flip semispaces

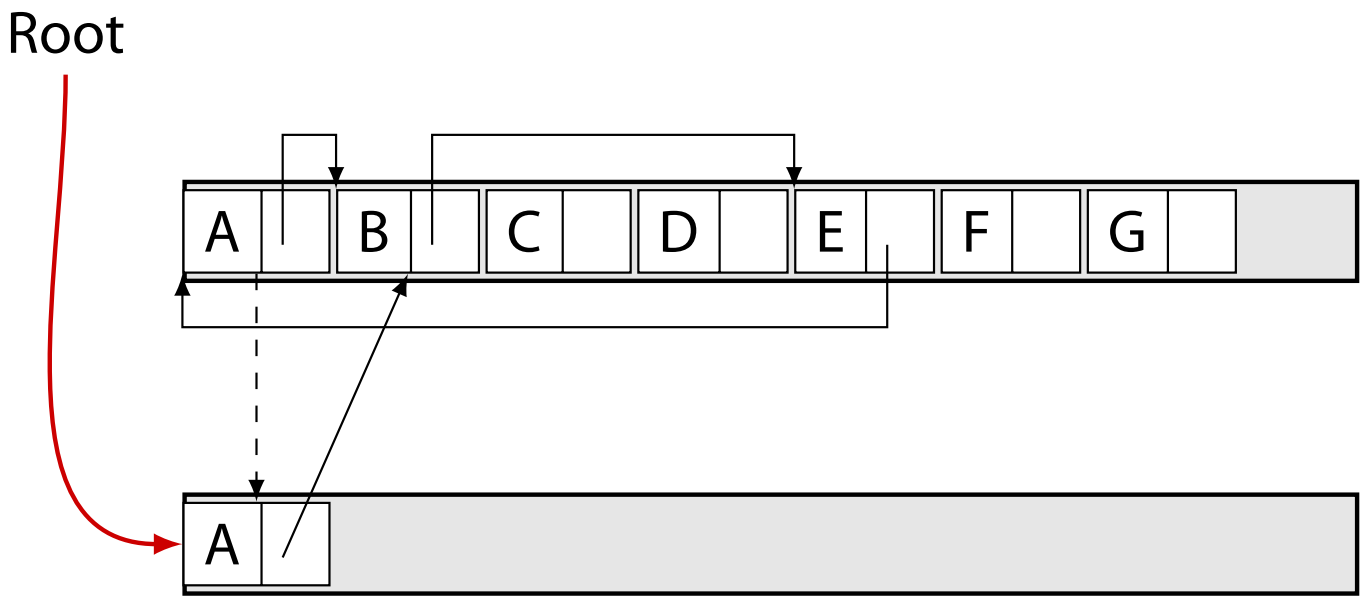
Copying GC

- space is divided in two semispaces:
  - *fromspace*
  - *tospace*
- allocation: bump a pointer in *fromspace*
- GC: copy live objects from *fromspace* to *tospace*
- flip semispaces
- need to update pointers from old to new objects
  - forward pointers

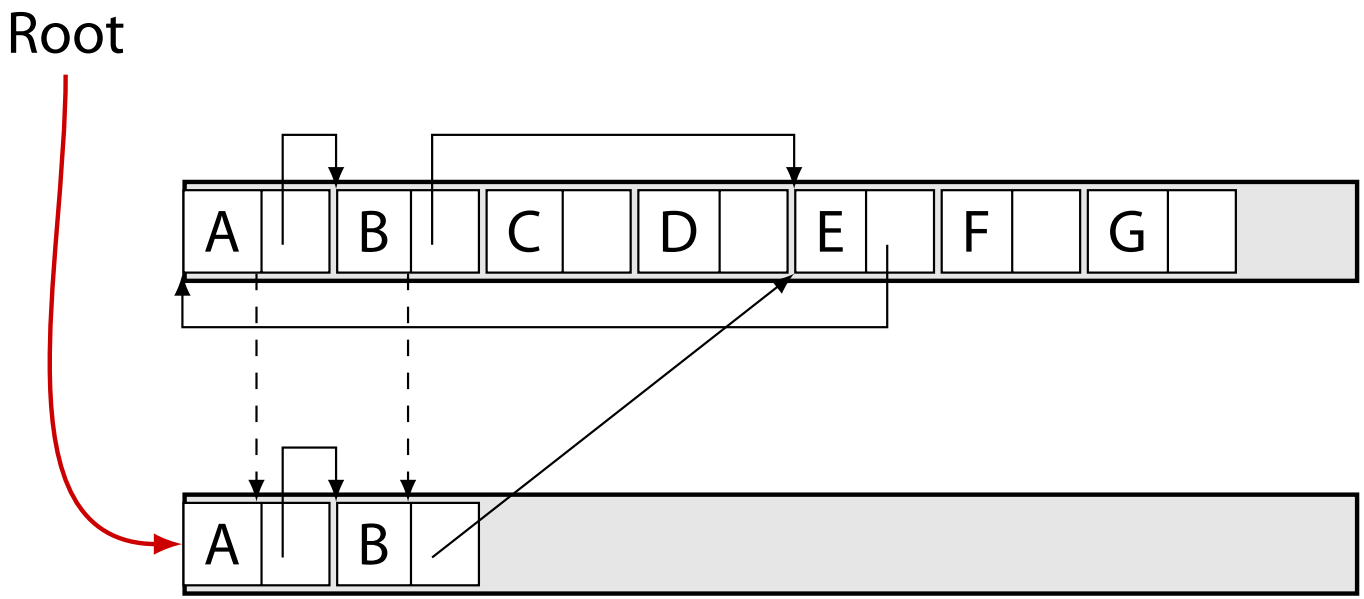
Copying GC: Example



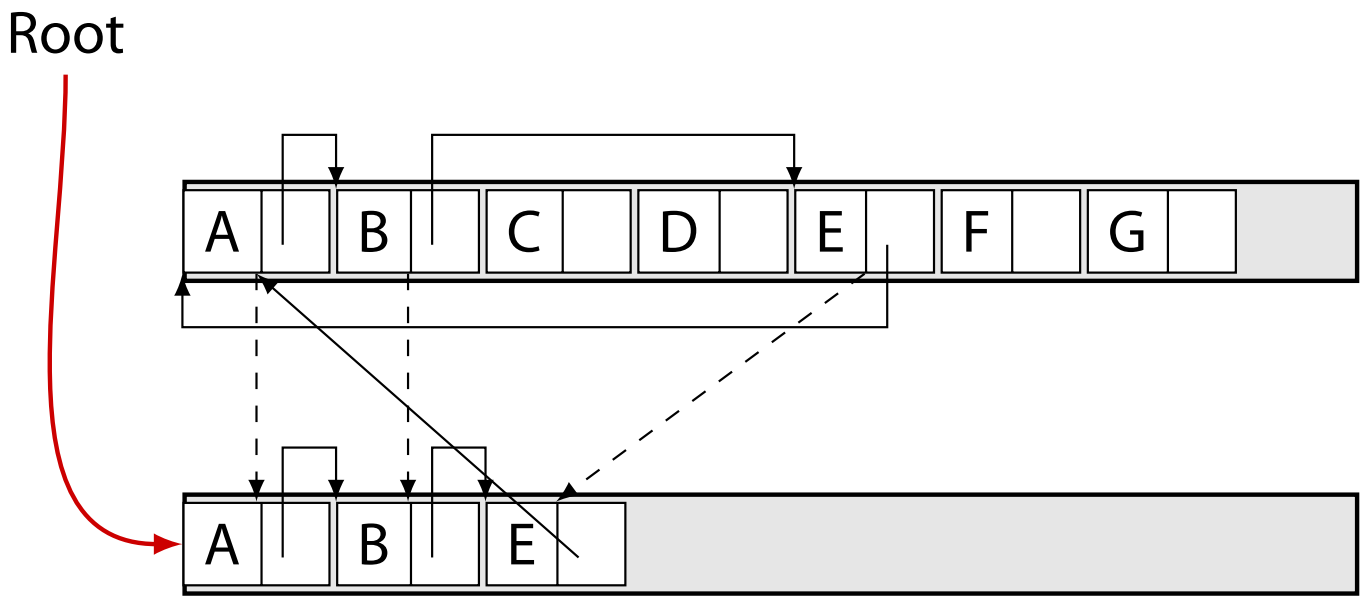
Copying GC: Example



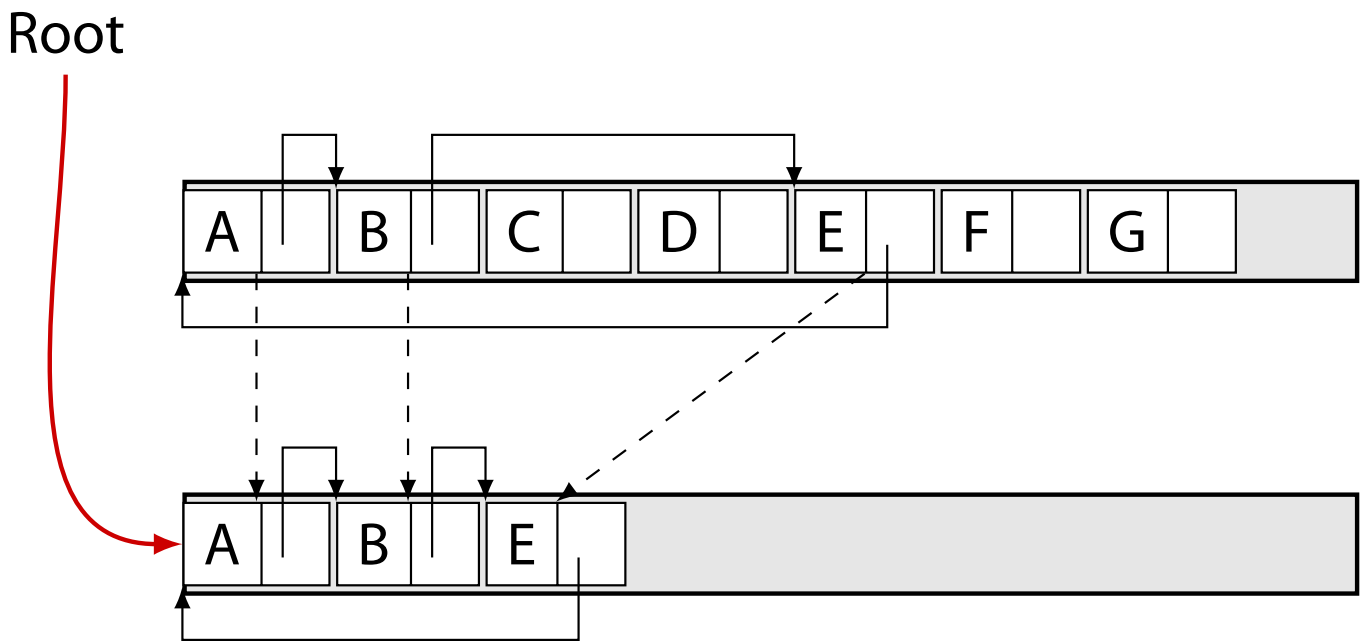
Copying GC: Example



Copying GC: Example



Copying GC: Example



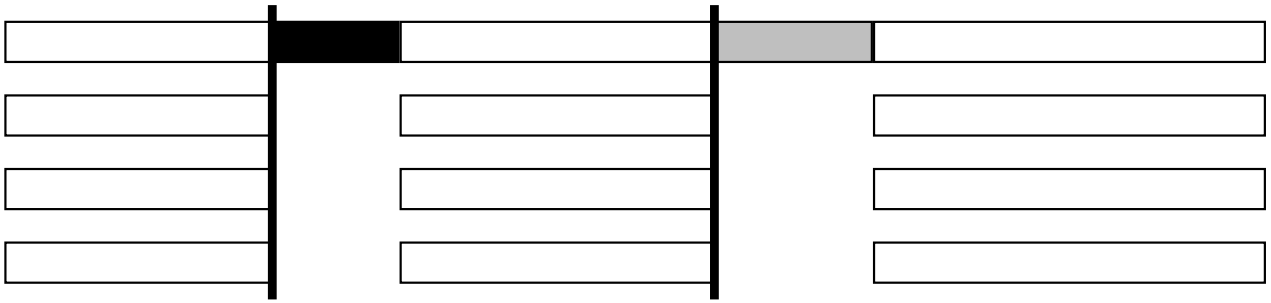
Introduction

Parallel Garbage Collection

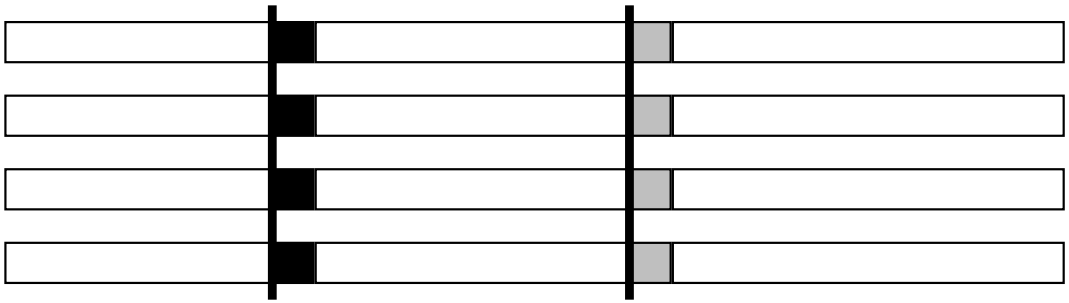
Concurrent Garbage Collection

Stop-the-world parallel GC

single-threaded GC:



parallel GC:



Parallel Marking

[Flood et al. 2001]

What:

- start from mutator roots, traverse object graph, mark objects

Parallel Marking

[Flood et al. 2001]

What:

- start from mutator roots, traverse object graph, mark objects

How:

- statically over-partition roots into groups
- assign GC threads dynamically to groups
- work stealing queues
  - outgoing references found in object are pushed into (local) queue
- fixed-size queues with overflow handling
- termination: status bit

Parallel semispaces

[Flood et al. 2001]

- Copying GC
- used for young generation

Parallel semispaces

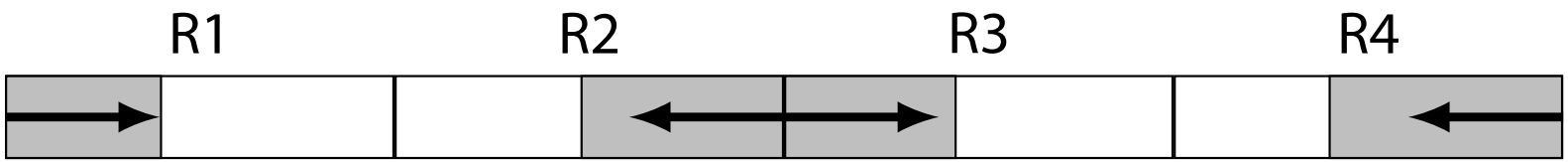
[Flood et al. 2001]

- Copying GC
- used for young generation
- working queues
- each thread operates on its own *tospace* (LAB: local allocation buffer)
- when a new object is met:
  - optimistically allocate space in LAB
  - CAS to update forward pointer

Parallel mark-compact

[Flood et al. 2001]

- used for old generation
- parallel version of Lisp2 algorithm (mark + 3 phases)
- over-partitioning
- compaction: each thread has its own region
  - alternate directions to avoid fragmentation



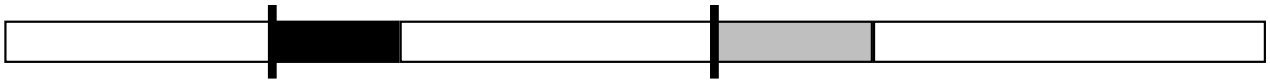
Introduction

Parallel Garbage Collection

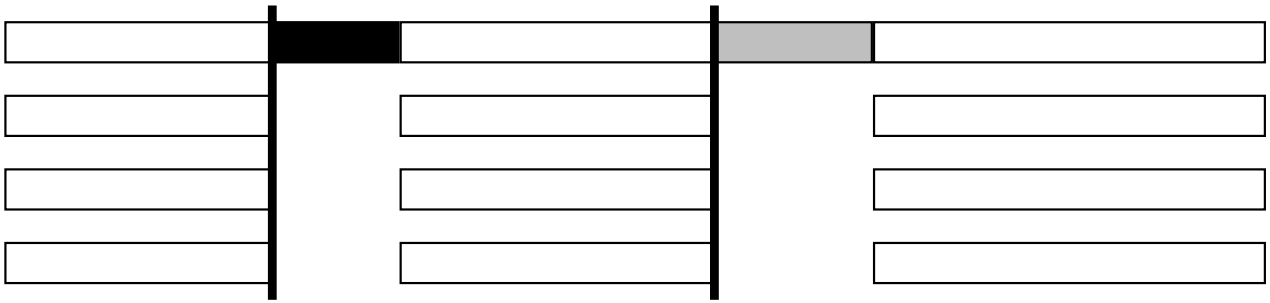
Concurrent Garbage Collection

Recap: Stop-the-world GC

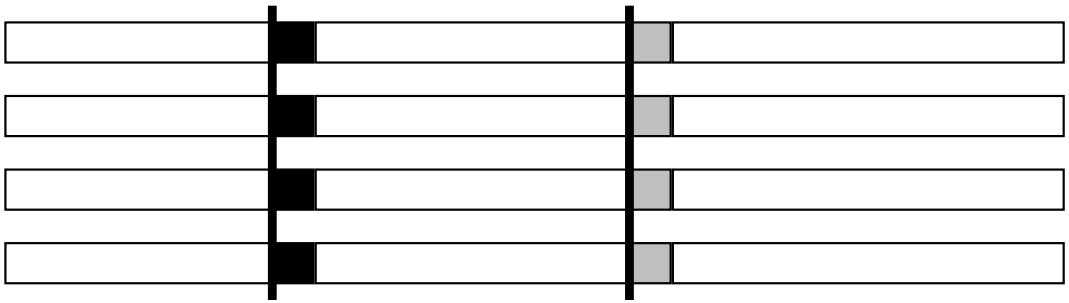
one mutator thread:



multiple mutator threads:

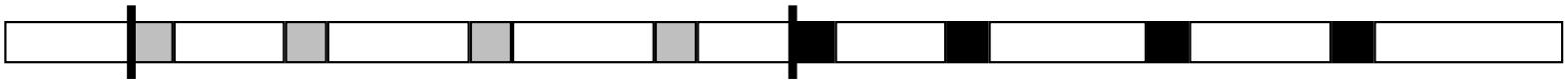


parallel GC:

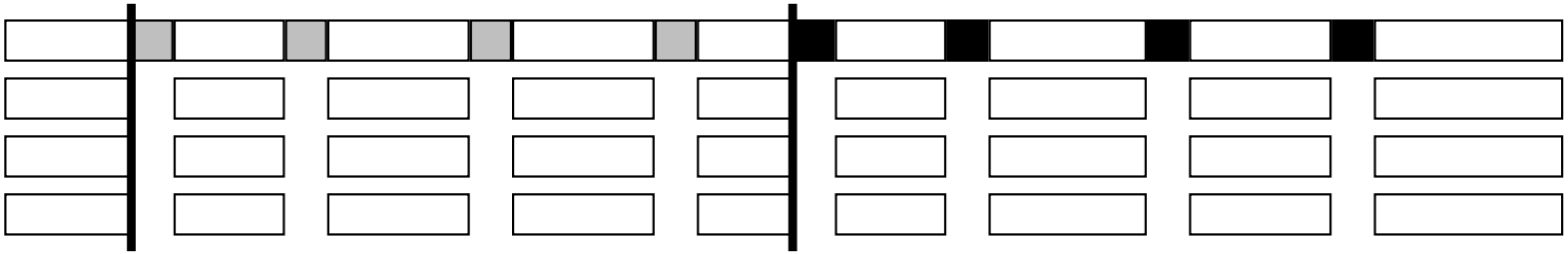


Incremental GC

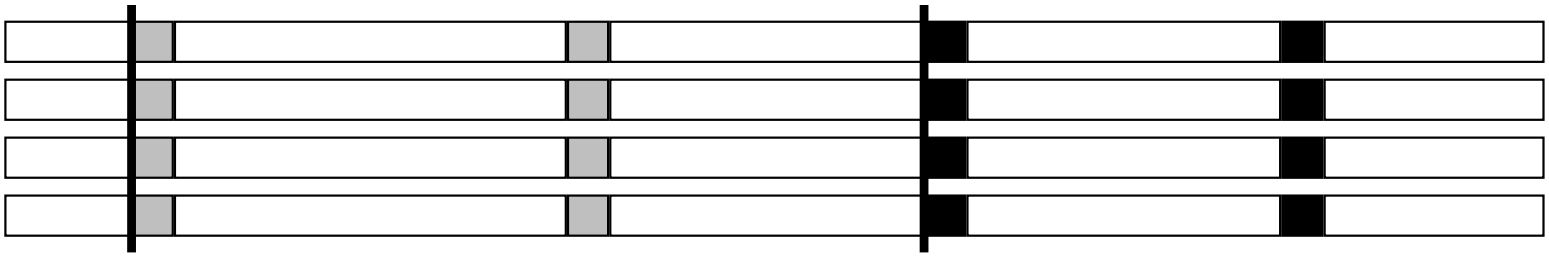
single mutator thread:



multiple mutator threads:

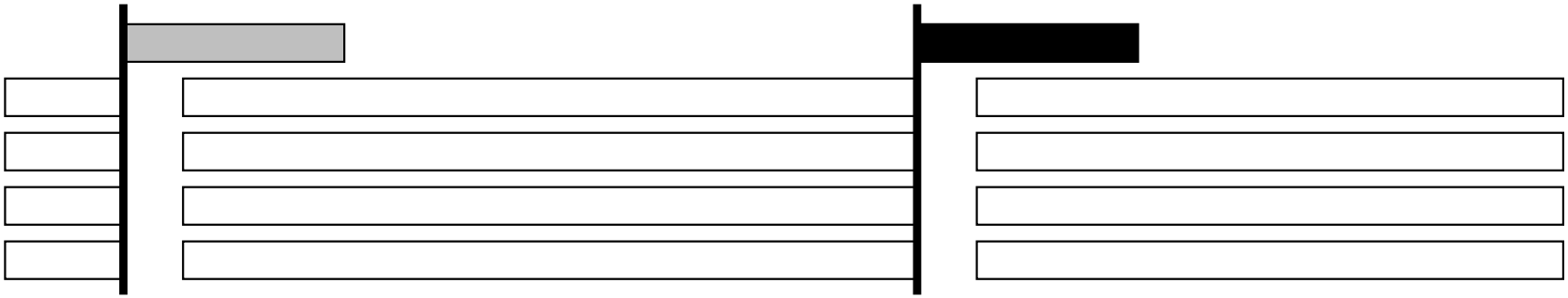


parallel:

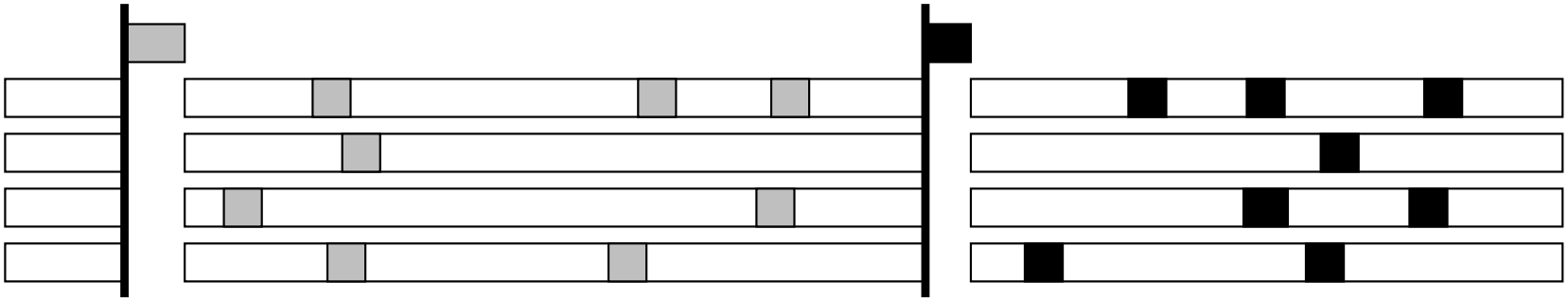


Mostly-concurrent GC

Mostly-concurrent GC:



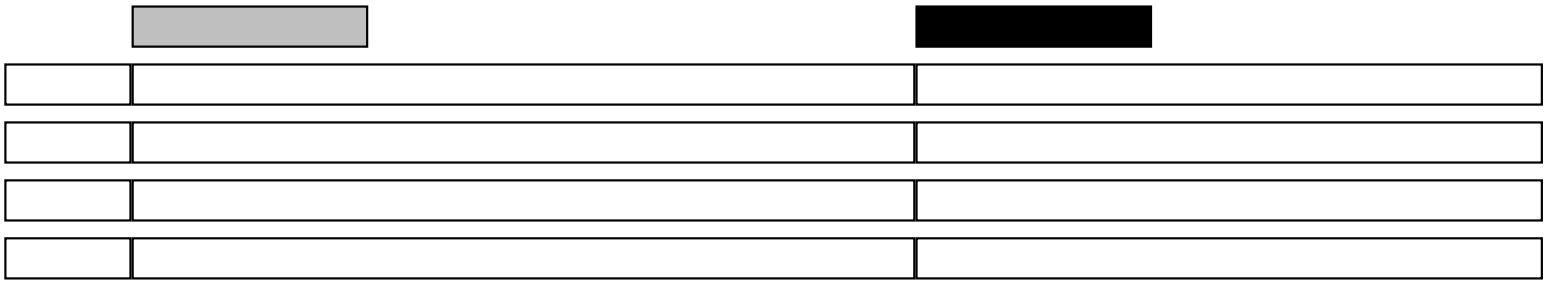
Mostly-concurrent incremental GC:



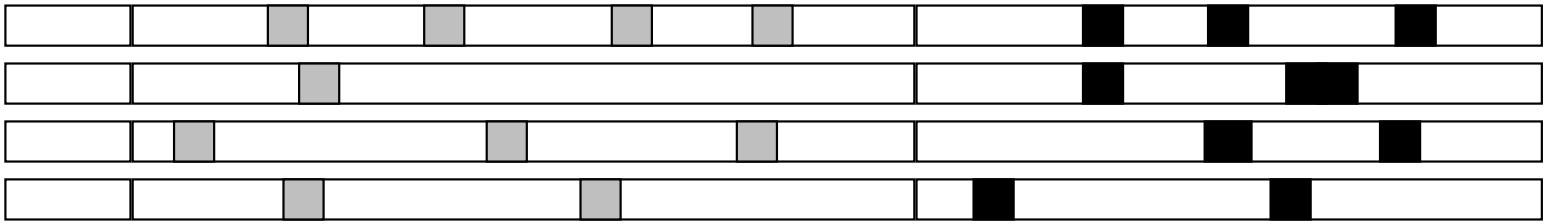
- stops mutators (e.g., at the start of the cycle)
  - but not for the whole collection cycle
- e.g., for obtaining stack roots

On-the-fly GC

On-the-fly GC:



On-the-fly incremental GC:

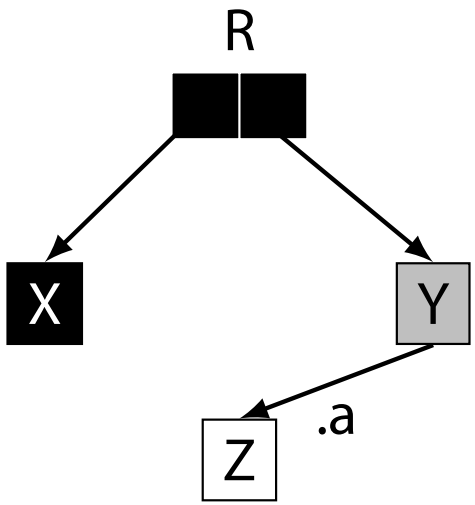


Concurrent GC issues

- Collector can't assume state remains unchanged
- Correctness:
  - *safe:* retain all reachable objects
  - *liveness:* the collection phase should termninate
  - floating garbage should eventually be collected
- tricolour abstraction
  - changes between black and white wavefronts may break collector assumptions

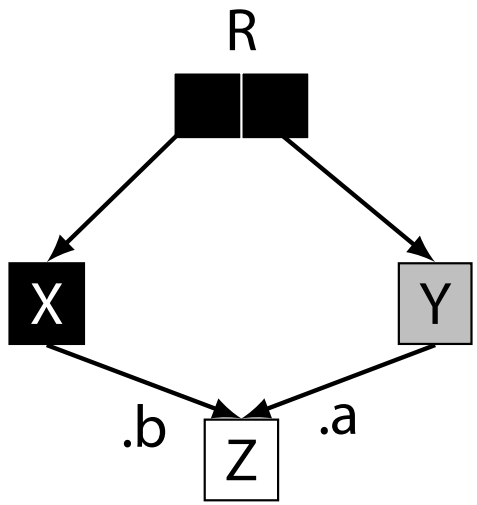
The lost object problem

(example #1)



The lost object problem

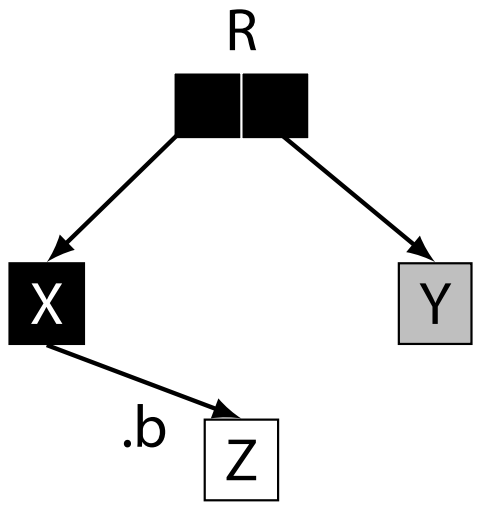
(example #1)



- `X.b = Y.a`

The lost object problem

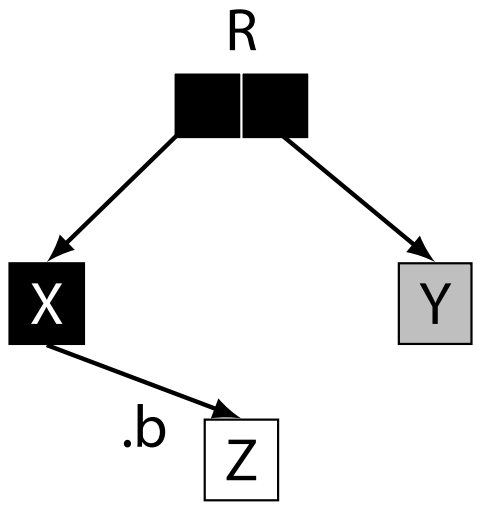
(example #1)



- `X.b = Y.a`
- `del Y.a`

The lost object problem

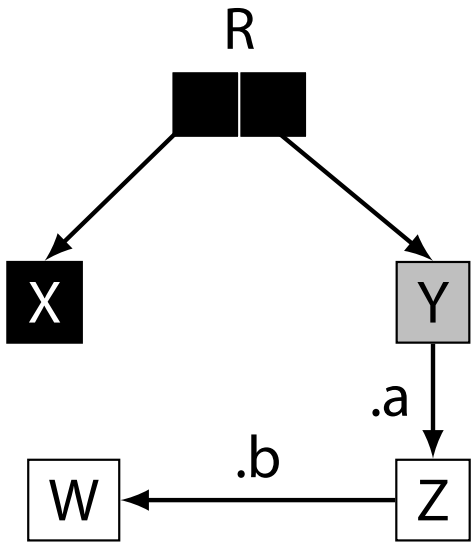
(example #1)



- `X.b = Y.a`
- `del Y.a`
- collector can't locate Z

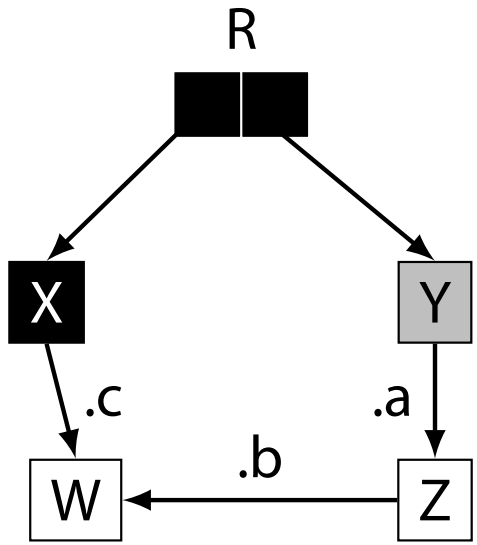
The lost object problem

(example #2)



The lost object problem

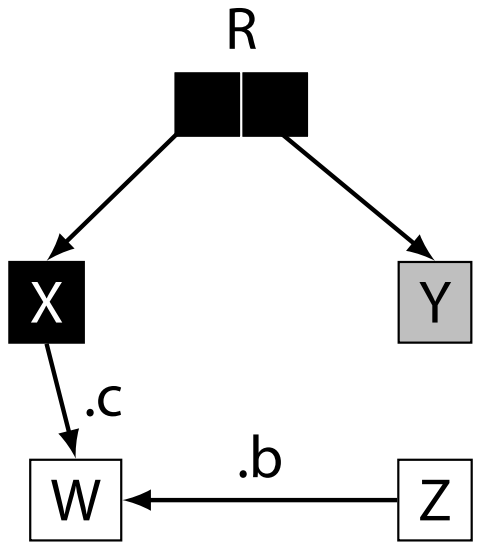
(example #2)



- `X.c = (Y.a).b`

The lost object problem

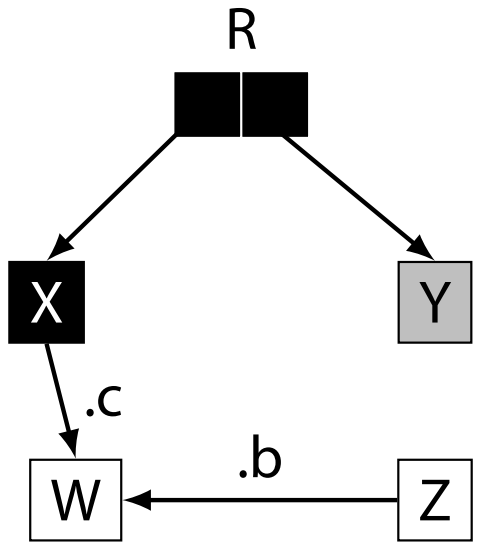
(example #2)



- `X.c = (Y.a).b`
- `del Y.a`

The lost object problem

(example #2)



- `X.c = (Y.a).b`
- `del Y.a`
- collector can't locate `Z,W`

Conditions for losing objects

Examples:

- #1: lost object was directly reachable from a gray object
- #2: lost object was transitively reachable from a gray object

Conditions for losing objects:

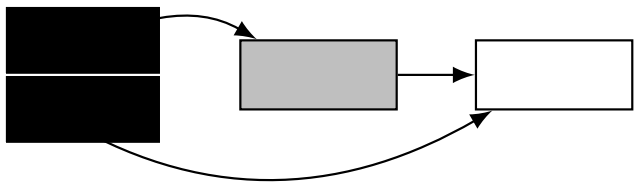
- mutator stores a pointer to a white object into a black object
- all paths from gray objects to that white object are destroyed

Correctness invariants: weak invariant

weak invariant

All white objects pointed to by a black object are *gray protected*: reachable from a gray object

- directly
- indirectly via a chain of white objects
- works for non-copying collectors, because the pointer remains valid

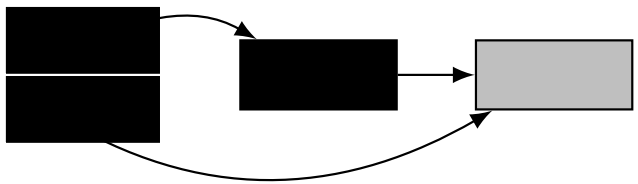


Correctness invariants: weak invariant

weak invariant

All white objects pointed to by a black object are *gray protected*: reachable from a gray object

- directly
- indirectly via a chain of white objects
- works for non-copying collectors, because the pointer remains valid

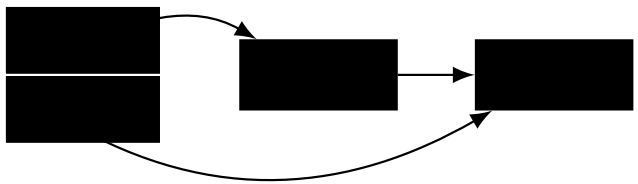


Correctness invariants: weak invariant

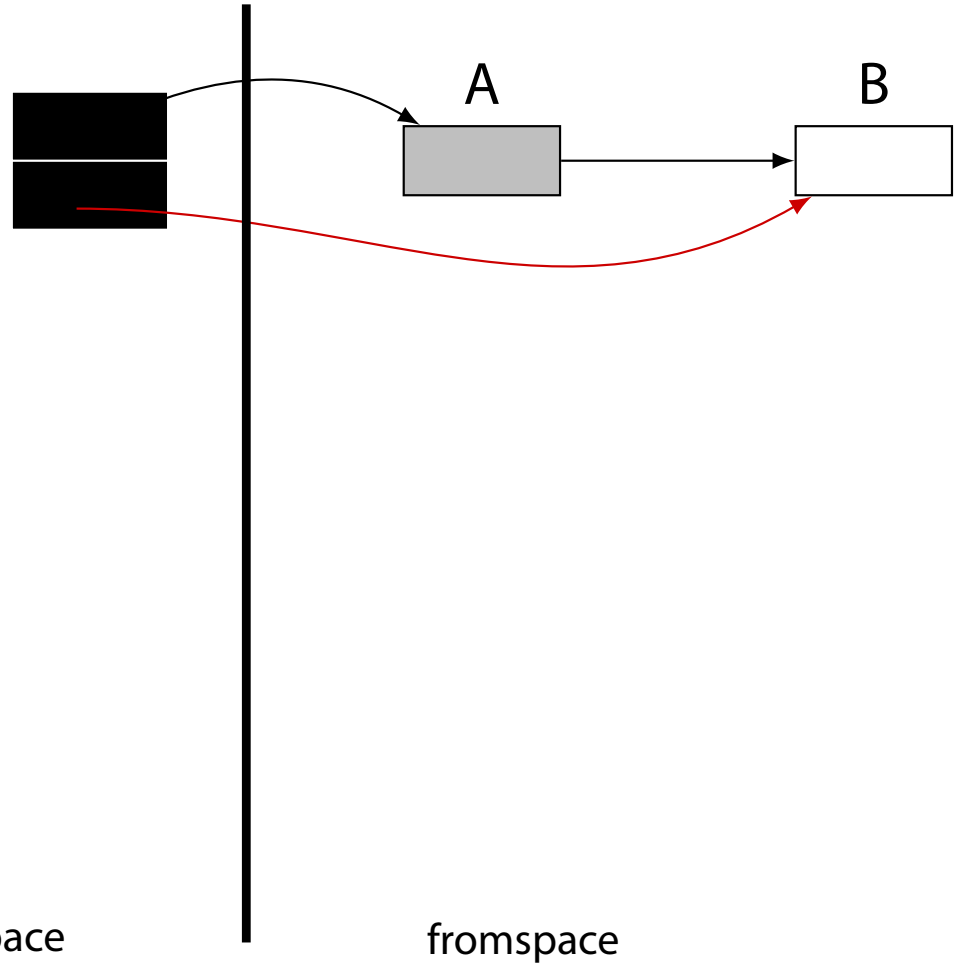
weak invariant

All white objects pointed to by a black object are *gray protected*: reachable from a gray object

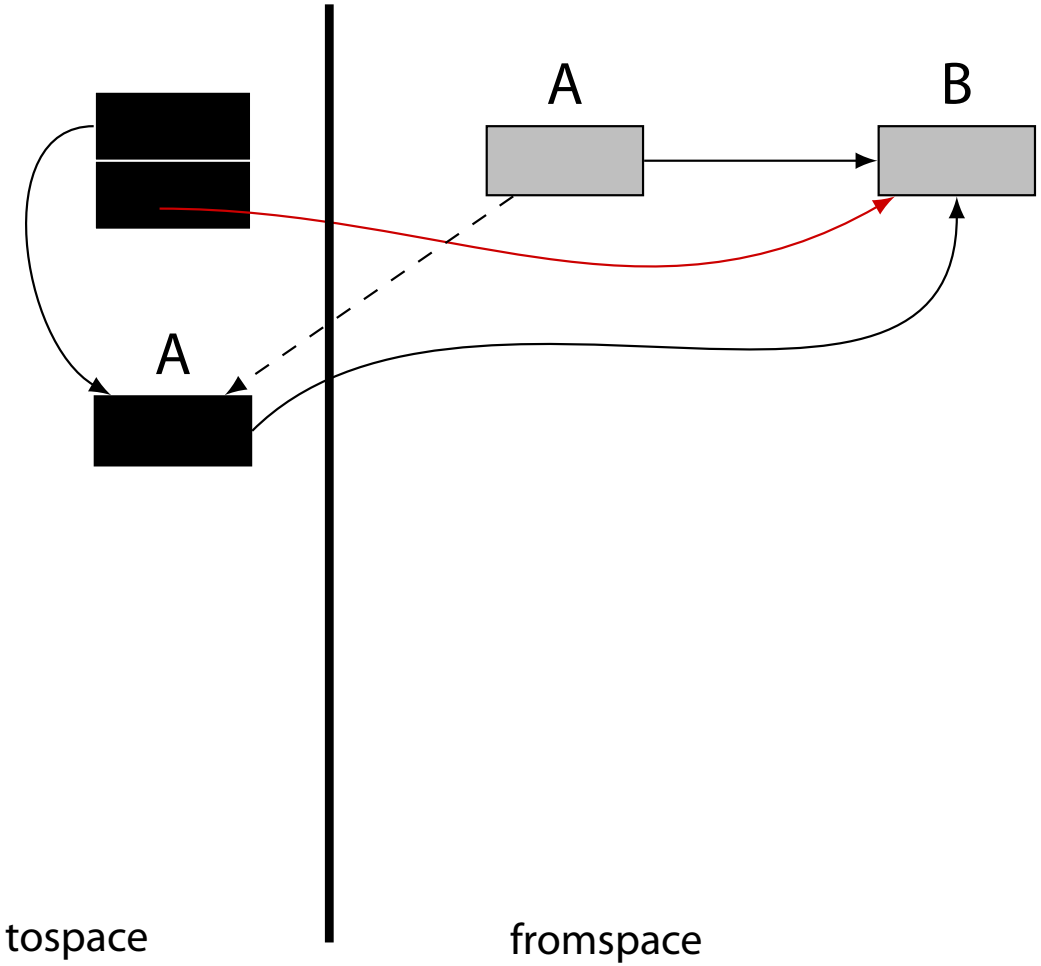
- directly
- indirectly via a chain of white objects
- works for non-copying collectors, because the pointer remains valid



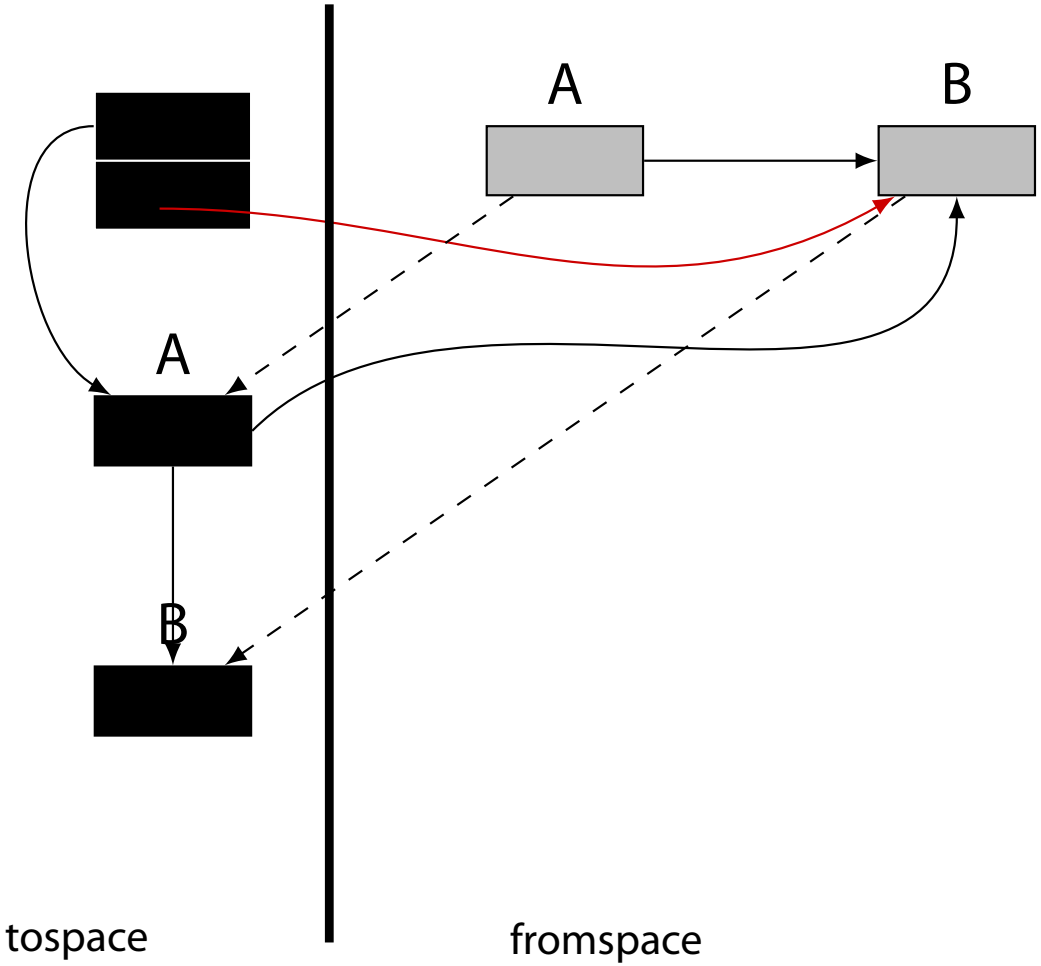
weak invariant on copying collectors



weak invariant on copying collectors



weak invariant on copying collectors



strong invariant for copying collectors

strong invariant:
No pointers from black objects to white objects

- implies the weak variant
- can be used for both copying and non-copying collectors

Enforcing invariants: barriers

- Detect READs or WRITEs to *interesting* pointers
- relevant information: source, target, old target
- overload READ/WRITE operations
- in GC literature: (read or write) barriers
- tradeoff: precission vs speed
- VM dirty bit to detect writes
- card tables

Incremental update solution

(preserve strong invariant)

- detect insertion of white pointers into black objects (write barrier)
- maintain invariant

  Example:

```
Write(src, field, ref):
    src.field = ref
    if isBlack(src) && isWhite(ref):
        revert(src) // turn src from black to gray
```

- locking/checks overhead can be amortized by sacrificing precision

Snapshot-at-the-beginning

(preserve weak invariant)

- detect mutator deletions of W pointers from W/G objects
- speculate that a pointer exists behind the wavefront, and keep object
- requires snapshotting the mutator at the beginning of the collection cycle
  - detect all its roots and make them black (black mutator)
  - make sure that mutator does not hold white objects

Thread-local GC

- avoid synchronization by collecting on a thread-private heap
- co-exist with global heap for shared objects
- shared objects cannot point to thread-private objects

- enforcing invariant:
  - write barriers
    e.g., by globalising transitive closure of private heap pointer
  - statically using pointer analysis
  - immutable objects
    (e.g., original erlang messaging with copies)

References

- The Garbage Collection Handbook
  Jones et al.
  2012
- Parallel Garbage Collection for Shared Memory Multiprocessors
  Flood et al.
  2001