Memory management

Nate Nystrom
University of Lugano

Credits

- Tim Teitelbaum, Cornell CS 412 slides, 2008
- Richard Jones and Rafael Lins, *Garbage Collection*
- Richard Jones, Antony Hosking and Eliot Moss, *The Garbage Collection Handbook*
- Matthias Hauswirth, SP slides, 2011

Outline

- Explicit memory management
- Automatic memory management
  - reference counting
  - mark and sweep
  - copying GC
  - concurrent/incremental GC
  - generational GC

Explicit memory management

Posix interface:

`void *malloc(size_t n)`

- allocate `n` bytes of storage on the heap and return its address

`void free(void *addr)`

- release storage allocated by `malloc` at address `addr`

User-level library manages heap, issues `brk` calls when necessary to grow the heap

C++: `new`/`delete` usually just call `malloc`/`free`

Explicit memory management – error-prone

Double deletes ("freed" twice)

- ```
  char* p = malloc(4096);
  free(p);
  free(p);
  ```

Freeing the wrong pointer

- ```
  char* p = malloc(4096);
  free(p+4);
  ```

Dangling pointers ("freed" too soon)

- ```
  char* p = malloc(4096);
  free(p);
  p[0] = 5;
  ```

Leaked objects ("freed" too late, or never)

- ```
  char* p = malloc(4096);
  // never free(p)
  ```

Problems with explicit memory mgmt

Makes modular programming more difficult

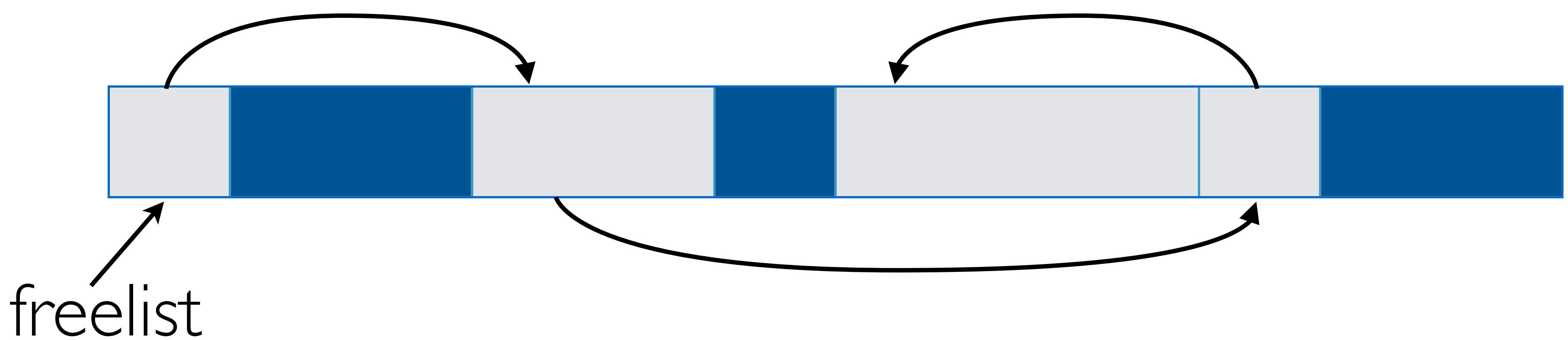
Every interface needs to agree on a contract

- Have to know what code "owns" a given object so that objects are deleted exactly once

Naive implementation

Blocks of unused memory stored in a freelist

- `malloc` finds unused block on freelist
- `free` puts block onto head of freelist



Simple, but:

external fragmentation = small free blocks scattered in the heap

- Cannot alloc a large block even if sum of all free blocks is enough

`malloc` can be O(|heap|)

Binning

Maintain freelists (bins) for different allocation sizes

- `bin(n)` is the freelist for chunks of size `n`

If chunks are all powers of 2 => **buddy system**

- `malloc`, `free` are O(log |heap|) worst case, O(1) in practice

The buddy system

`malloc(n)`

- round `n` up to nearest power of 2
- if no chunk of size `n` — i.e., `bin(n)` is empty
  - get chunk from `bin(2*n)`
  - split in half, return chunk of size `n`, add its buddy to `bin(n)`

`free()`

- add chunk of size `n` back to `bin(n)`
- if 2 buddies of size `n` are in `bin(n)`, coalesce and add chunk to `bin(2*n)`

Trades *external* for *internal fragmentation*

- Allocates larger chunks than needed because of rounding
- Typically 25% => no longer used in practice

Automatic memory management

Gives the programmer the illusion that they have infinite memory

Removes a huge class of bugs

Programmer doesn't have to think about it => huge boost in productivity

Automatic memory management

Techniques:

- regions
- reference counting
- garbage collection

Regions [Tofte-Talpin 1994]

- Allocate objects into regions with a fixed dynamic scope
- When region goes out of scope, free all objects in the region

```
r = newregion
var head: ListNode = null
for (i <- 1 to 1000) {
  newNode = allocInRegion[ListNode](r)
  newNode.next = head
  head = newNode
}
deleteregion r
```

- Used in some functional language implementations
  - Region inference finds where to insert **`newregion`** and **`deleteregion`** and infers in which region a given object should be allocated
  - Performance competitive with garbage collection

Reachability-based memory management

Want to delete objects if they won't be used again

- This is undecidable!
- So must be conservative
  - might still retain objects that won't be used again
  - but will not free objects that will be used again

Use **reachability** as an approximation of liveness:

- if there is no way to reach the object from globals, stack, registers, then object cannot be used again

Can determine reachability via:

- tracing => garbage collection
- reference counting

Reference counting

Idea:

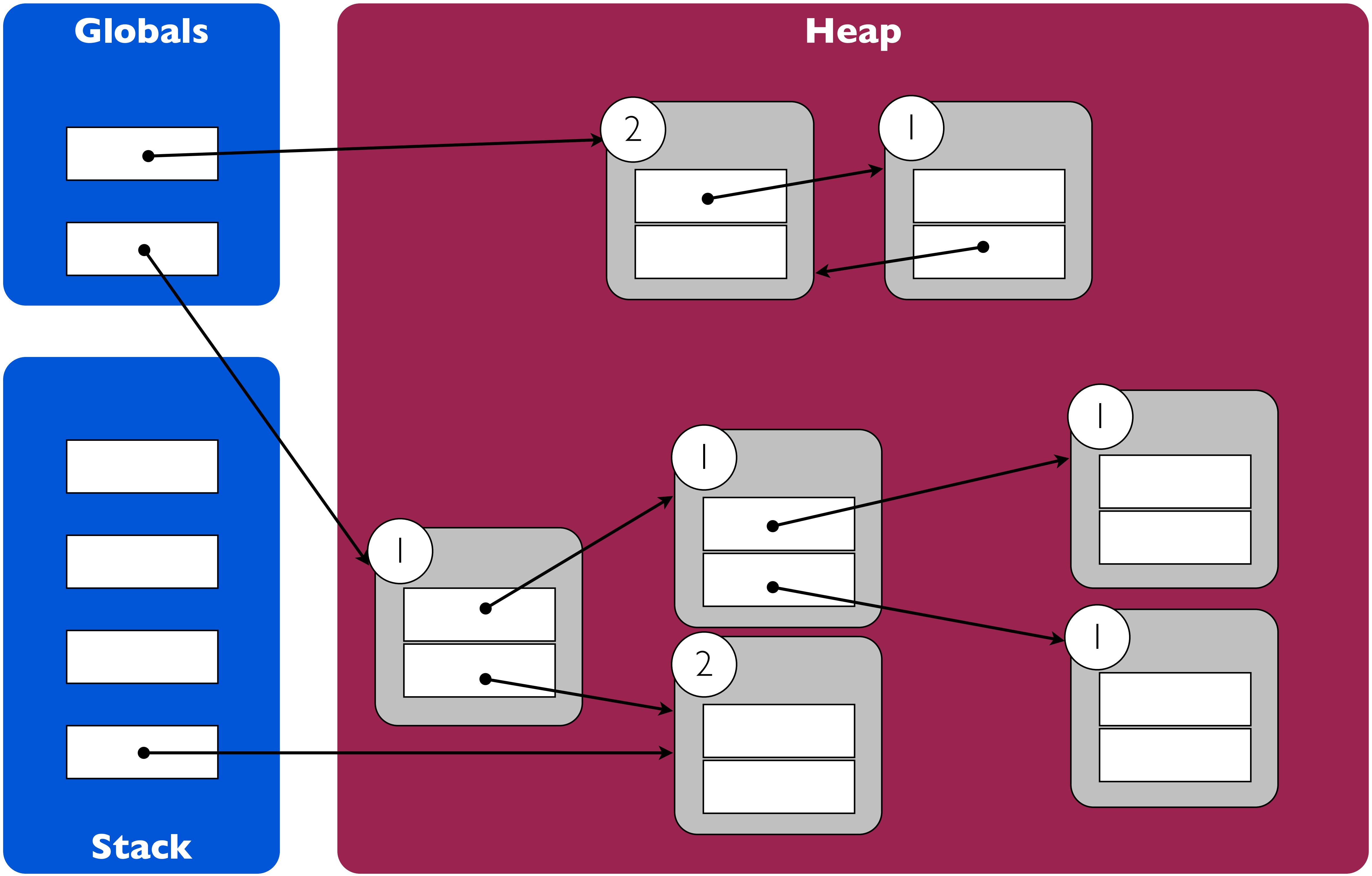
- associate a **reference count** with each object
- number of references (pointers) to the object

Keep track of reference counts

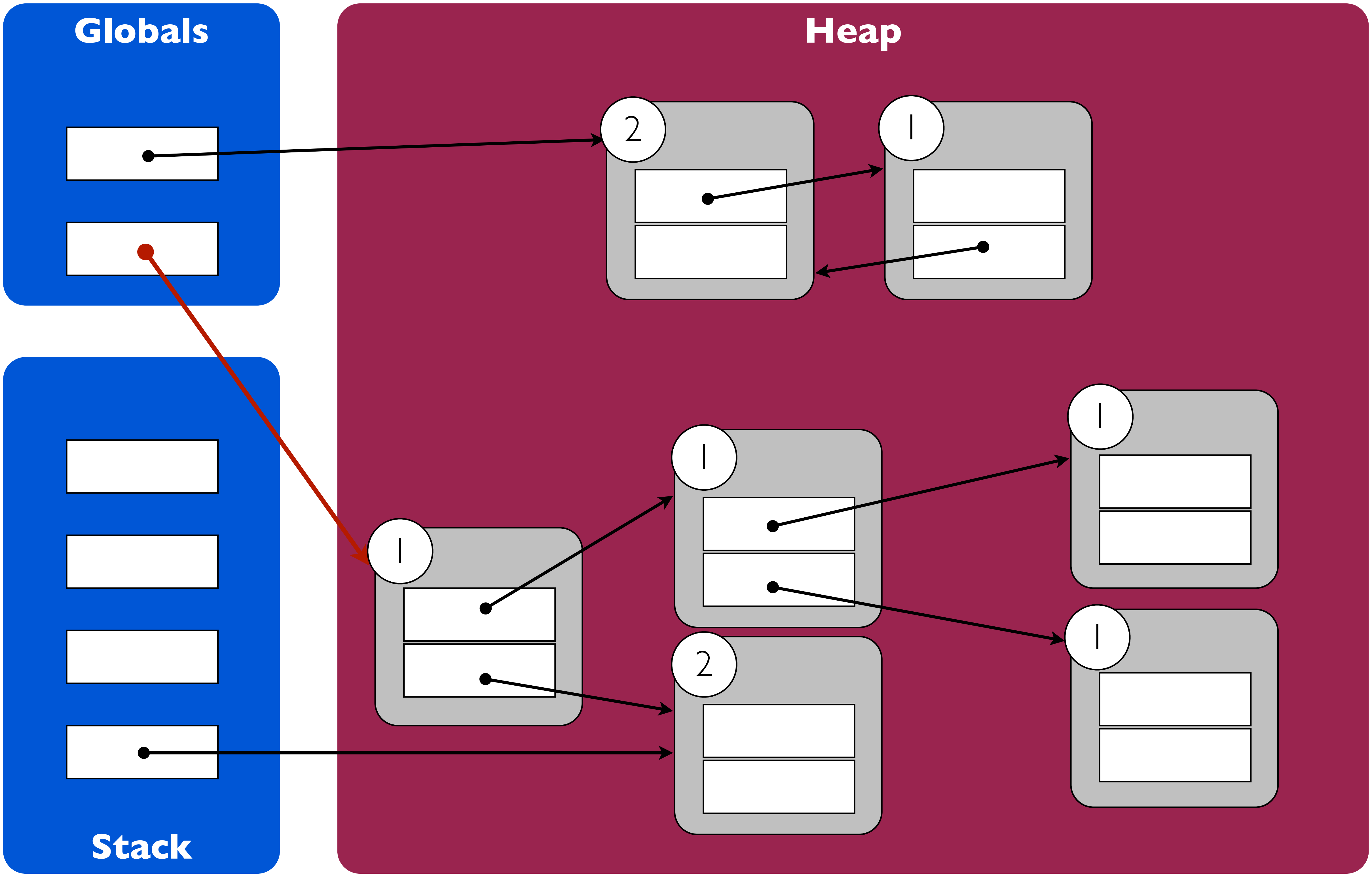
- For assignment `x = e`
  - decrement ref count for object referenced by `x` (if any)
  - increment ref count for object referenced by `e`
  - do the assignment

When reference count hits 0, object is unreachable => free it

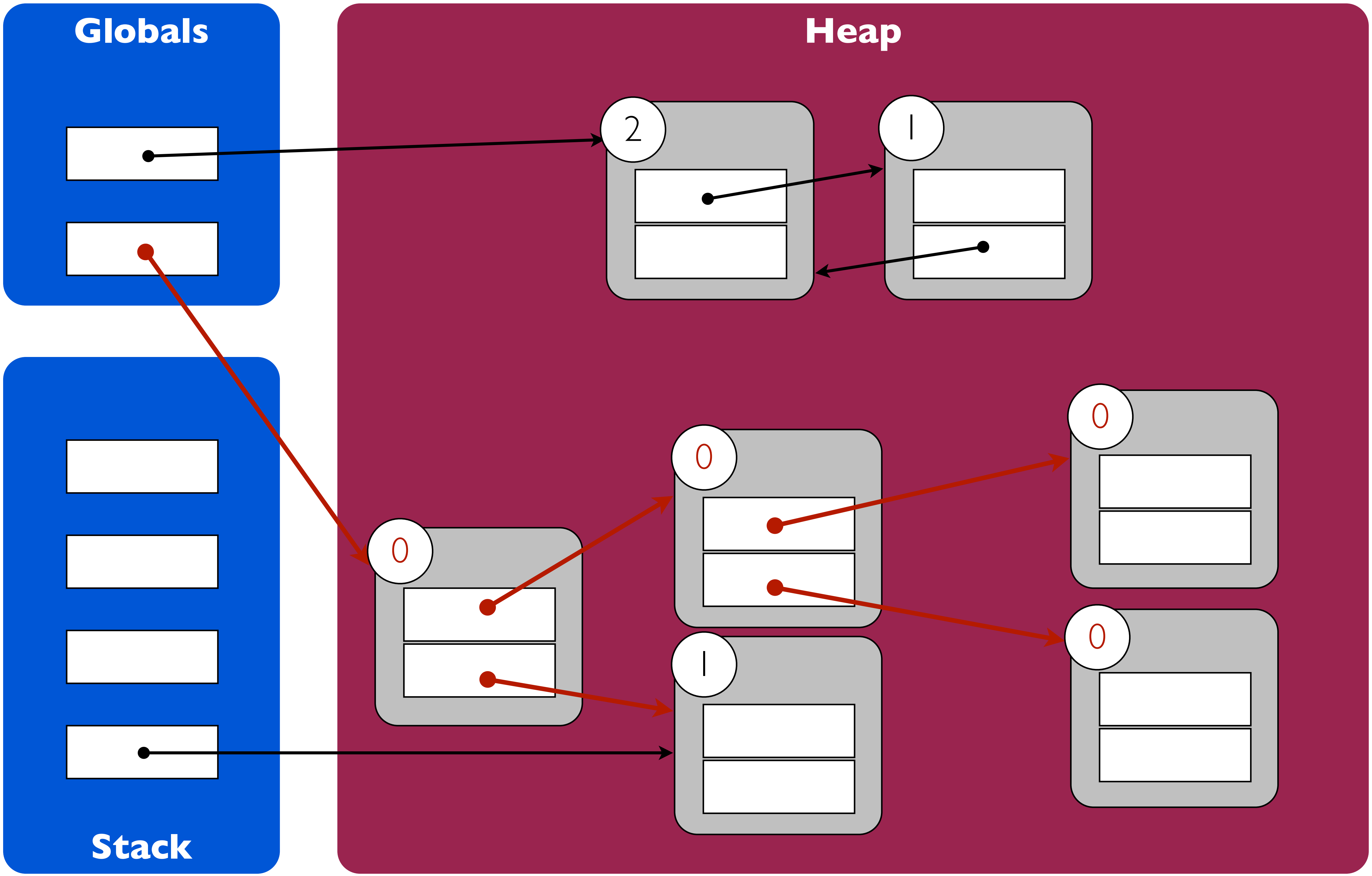
Reference counting



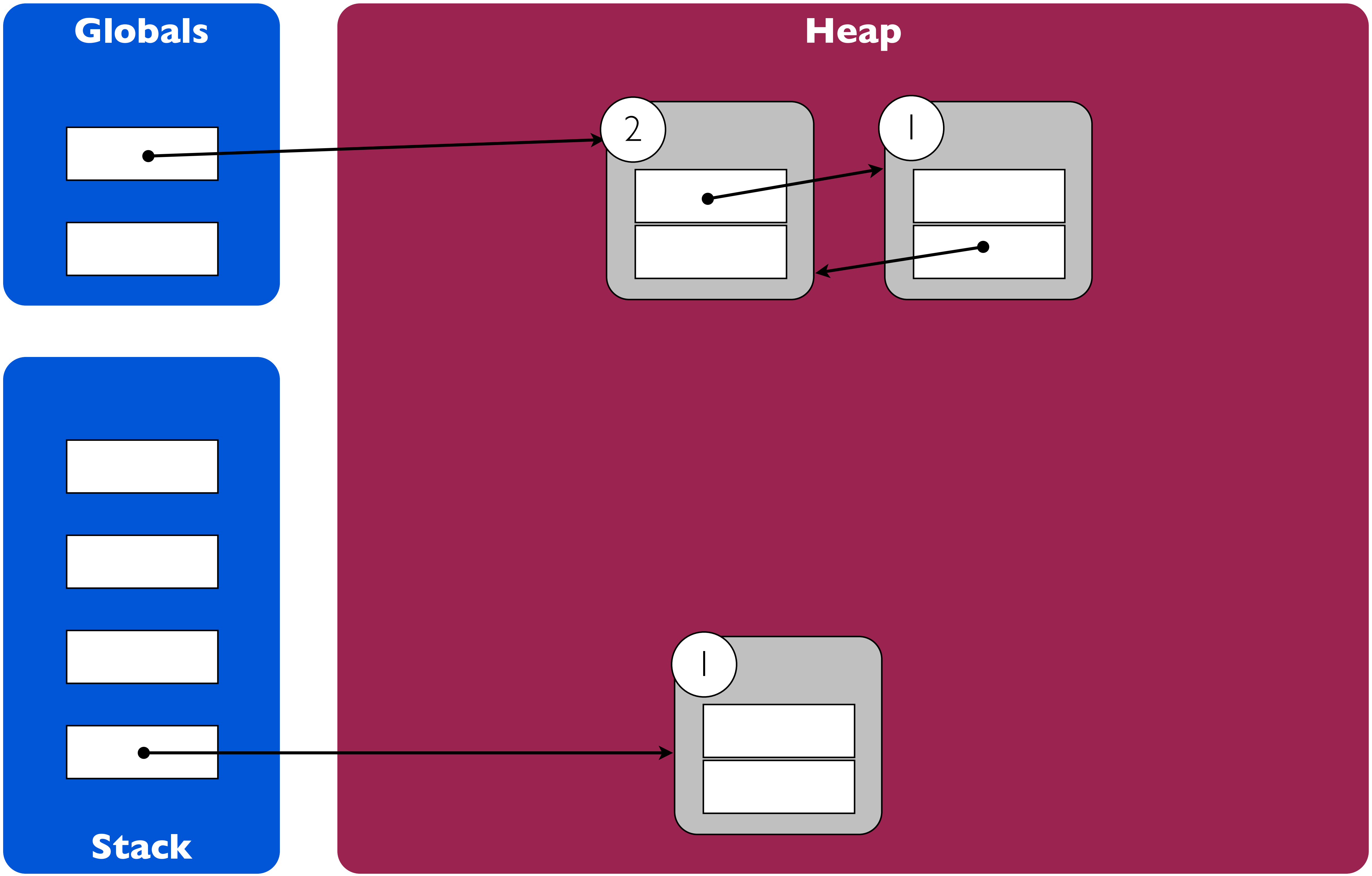
Reference counting



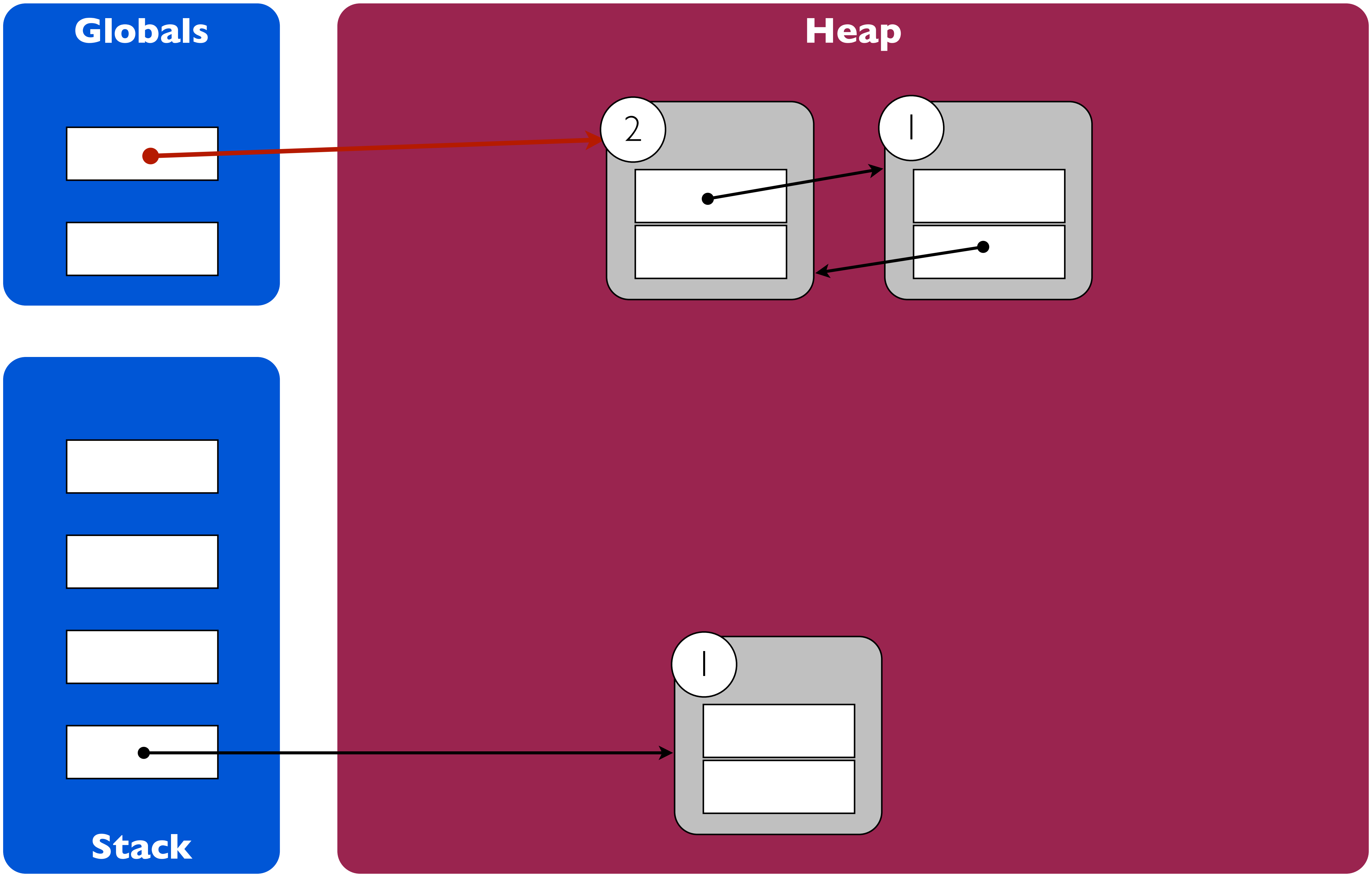
Reference counting



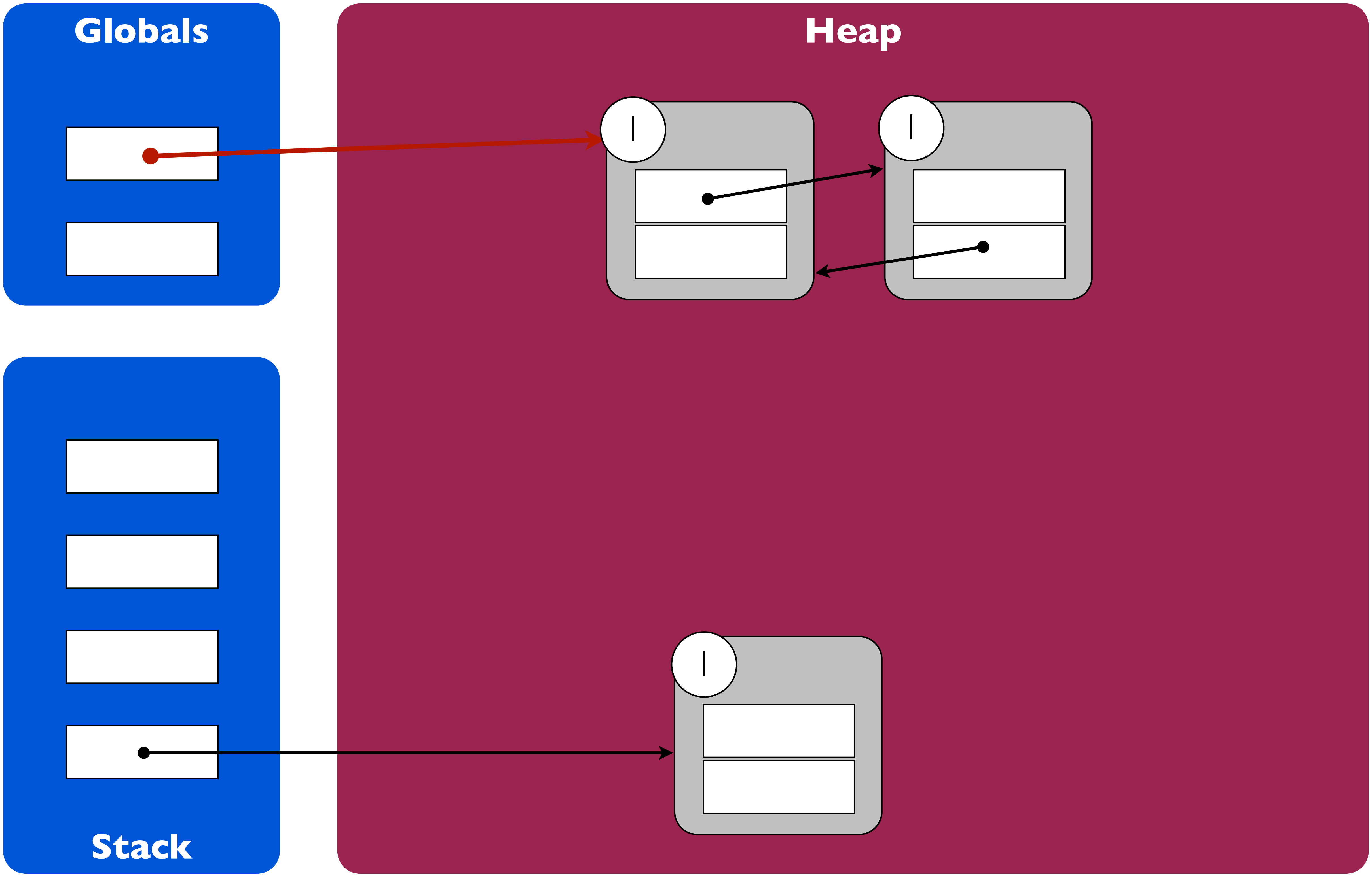
Reference counting



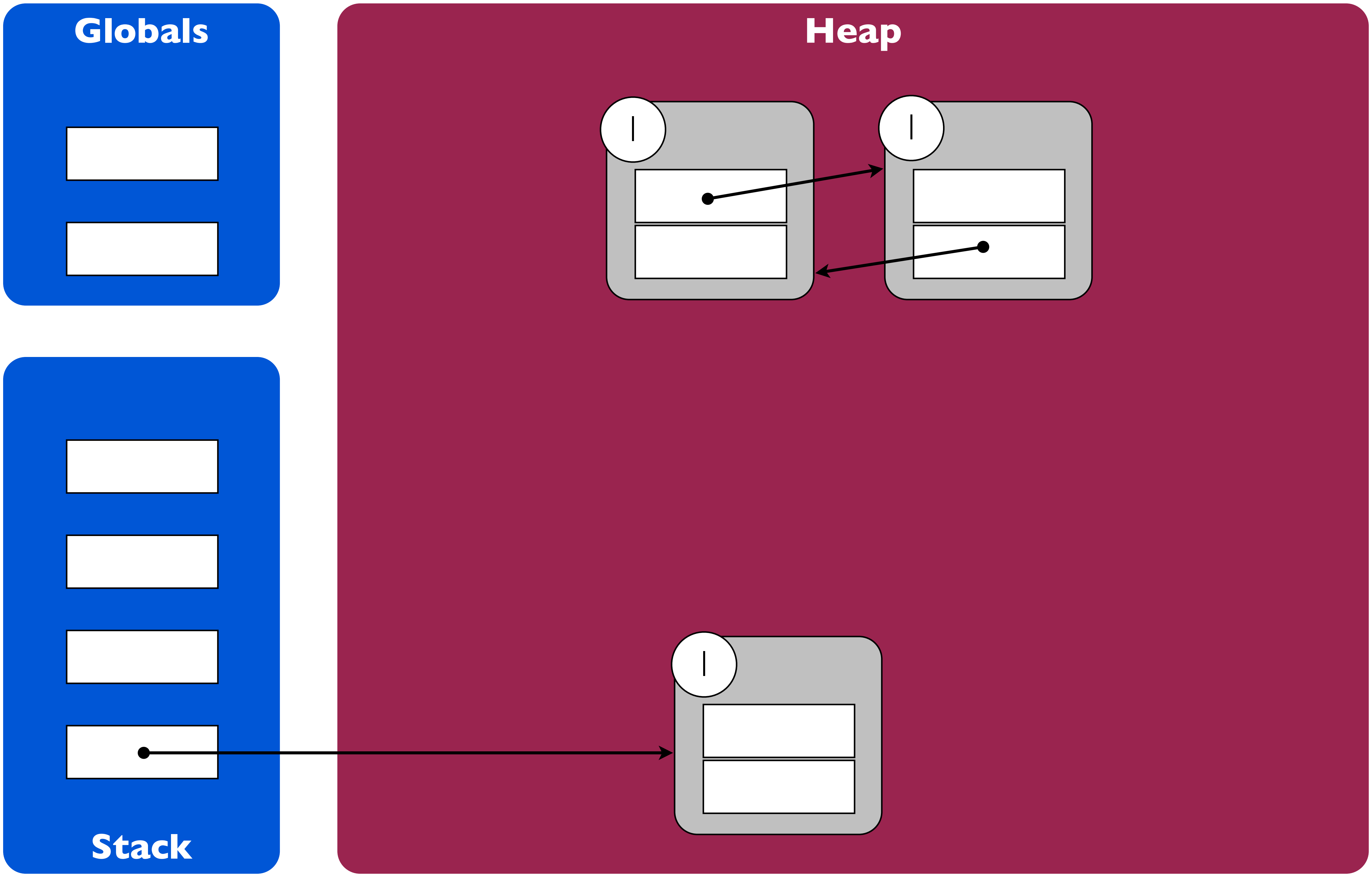
Reference counting



Reference counting



Reference counting



Problem: performance

Consider assignment: `x.f = y`

Without ref counts, one store instruction:

```
[tx + f_offset] = ty
```

Problem: performance

With reference counts:

```
t1 = [tx + f_offset]       ; load the old value of x.f
tc = [t1 + rc_offset]      ; load the ref count of old value
tc = tc - 1                ; decrement
[t1 + rc_offset] = tc      ; store the new reference count
bnz tc, do_not_free        ; check if count is 0
reclaim_object(t1)         ; if so, reclaim the object
do_not_free:
tc = [ty + rc_offset]      ; load the ref count of y
tc = tc + 1                ; increment
[ty + rc_offset] = tc      ; store the new reference count
[tx + f_offset] = ty       ; store the new value
```

Reference counting

- Advantage
  - Automatic (no programmer errors)
  - Frees dead objects immediately
  - Easy to implement
- Drawbacks
  - Still some pause times (lumpy deletion)
  - Space overhead: need count in object header
    - typically ~20%
  - Pervasive run-time overhead
  - Can't deal with cycles

Idea: backup tracing

To handle cycles, use a **backup tracing collector**

- In most cases, can reclaim objects immediately when ref count goes to 0
- Need to GC less often than if there were no ref counting

Can use just a few bits for the reference count

- **Sticky counters**: when ref count hits maximum (say 3), don't decrement it again
  - let backup collector recompute counts or reclaim
- Extreme: use 1 bit (object is either shared or not)

Garbage collection

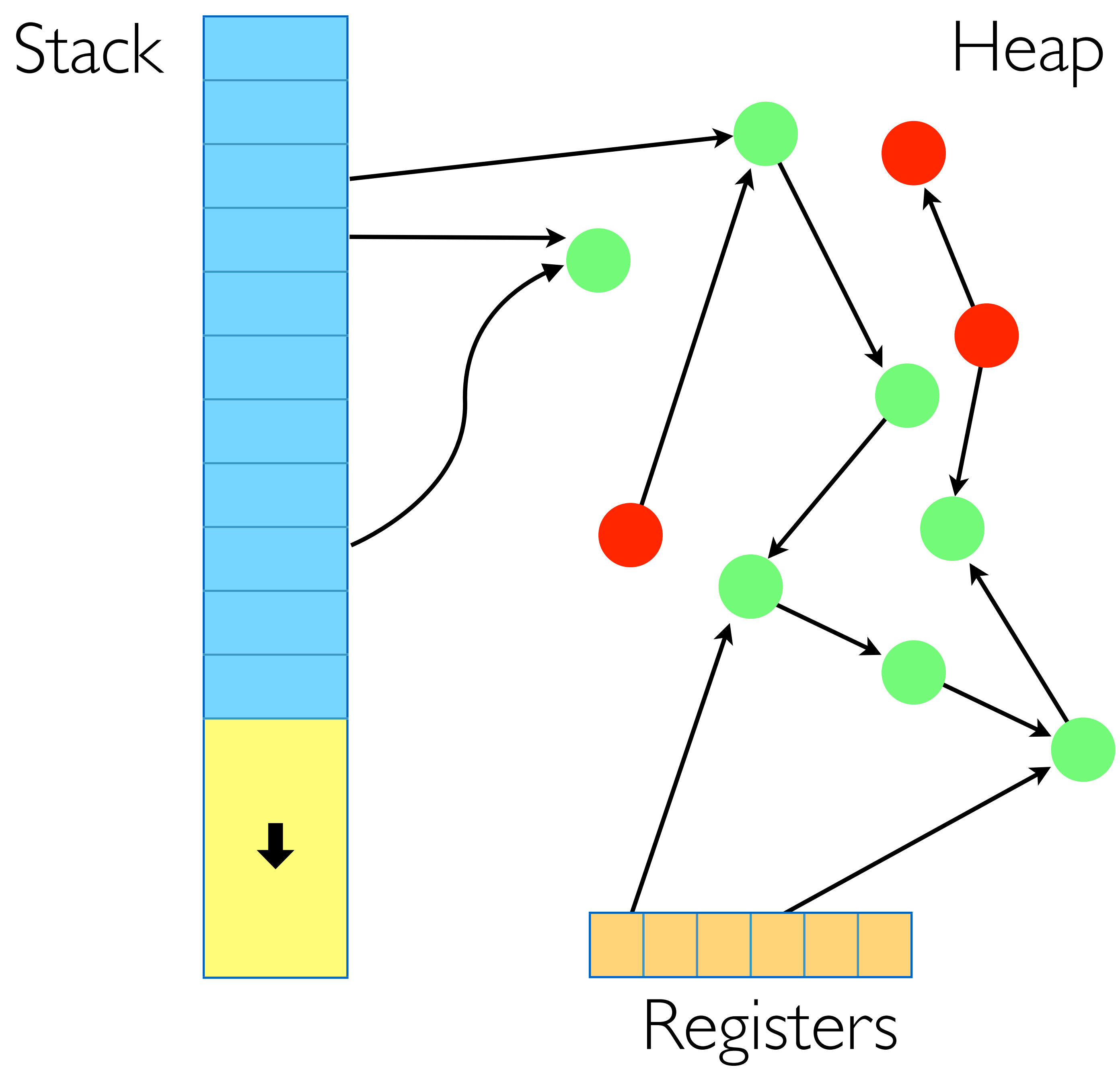
Three popular techniques

- mark-sweep
- mark-compact
- copying

Object graph

Stack, registers, globals are **roots** of the object graph

Anything reachable from the roots is **live**, all else is **garbage**



Abstraction

Useful to ignore the actual content of the object graph

Just treat it like a graph problem: identify unreachable nodes in the graph

The program is the **mutator**: it changes the graph

Mark-sweep GC

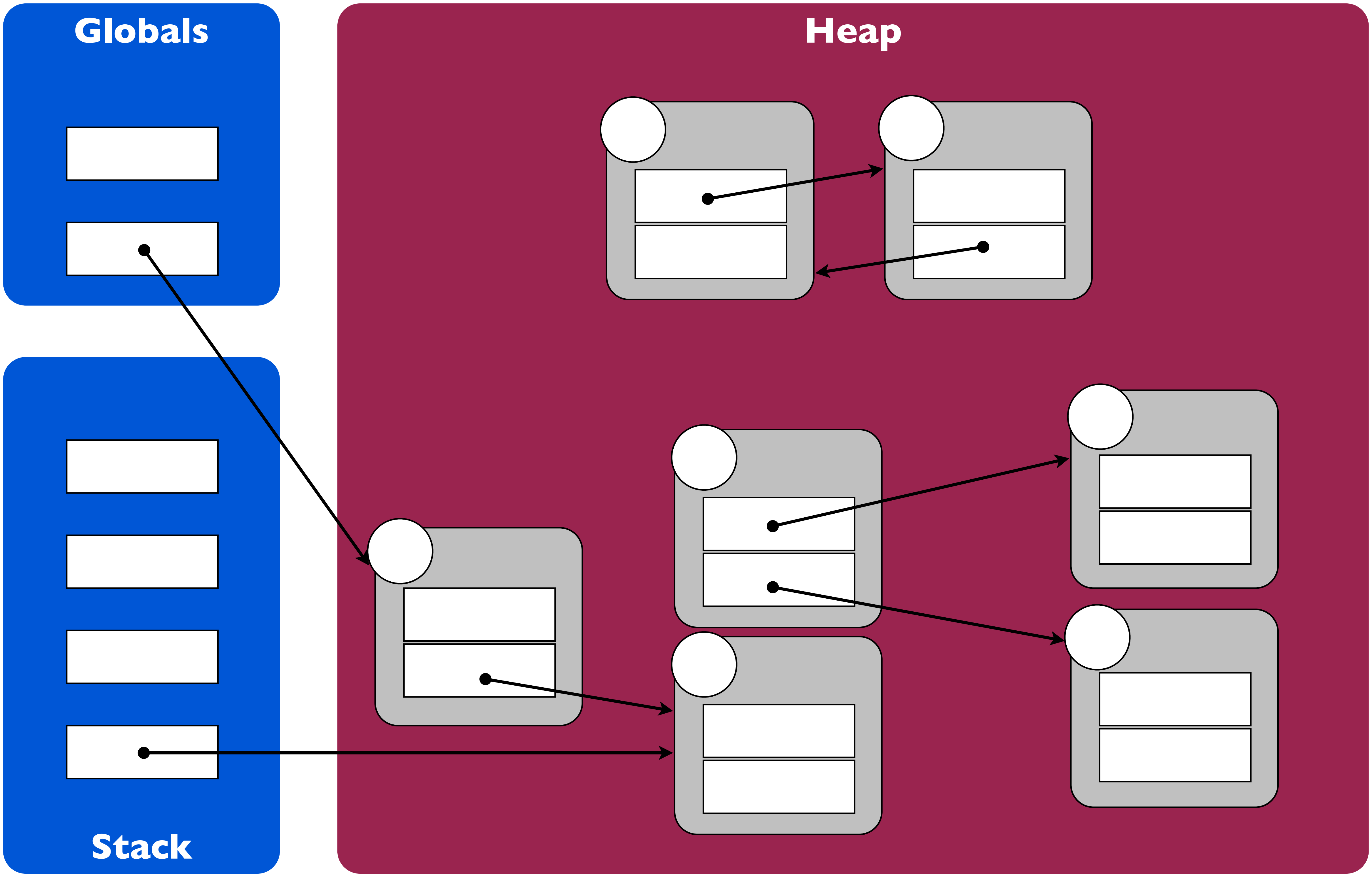
The classic algorithm

Two phases:

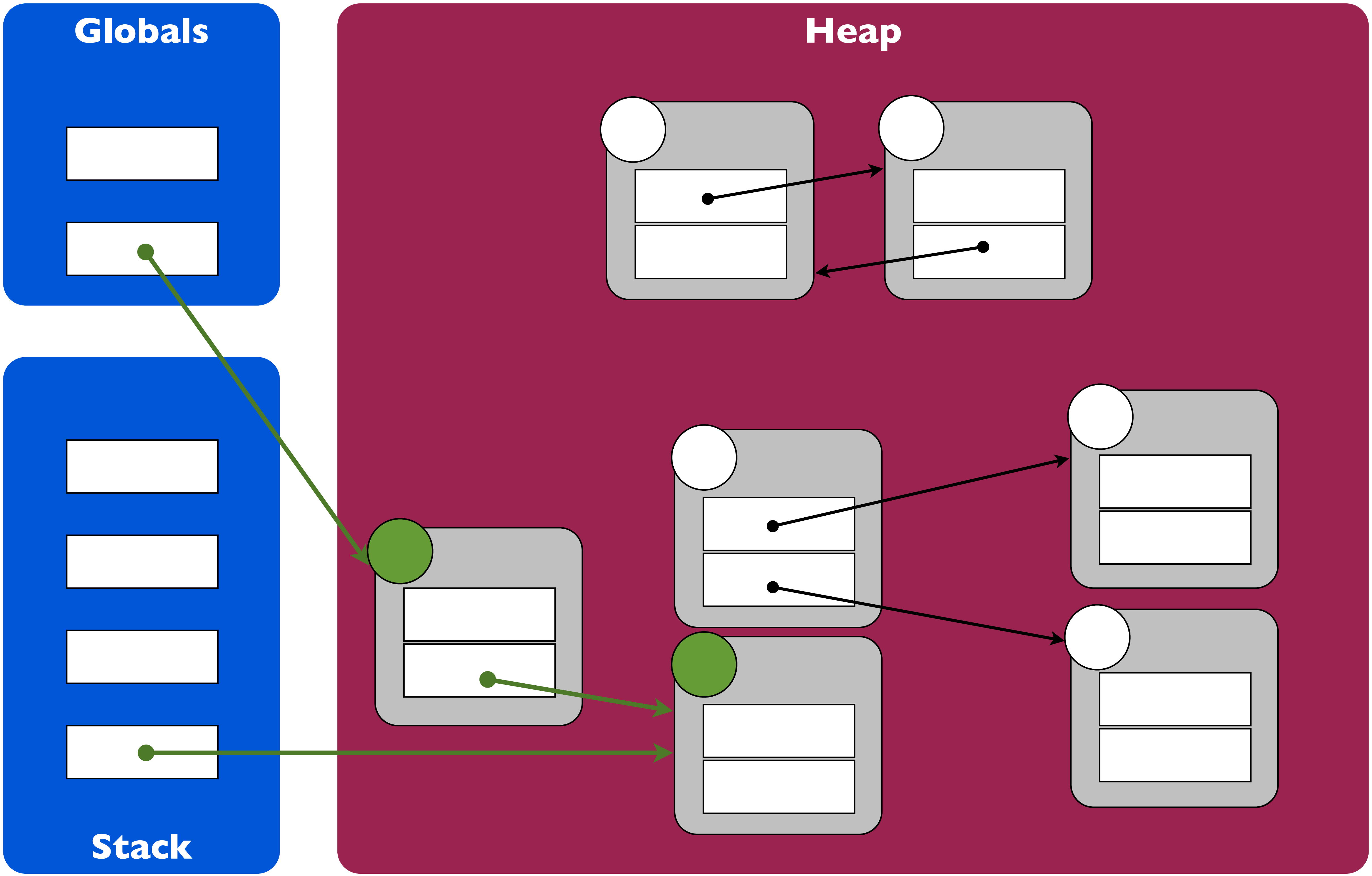
- Mark phase
  - start from roots, **trace** object graph, marking every object reached
- Sweep phase
  - iterate through all objects in the heap
  - reclaim unmarked objects
  - clear marks

  - optional: compact live objects in heap (called mark-compact)

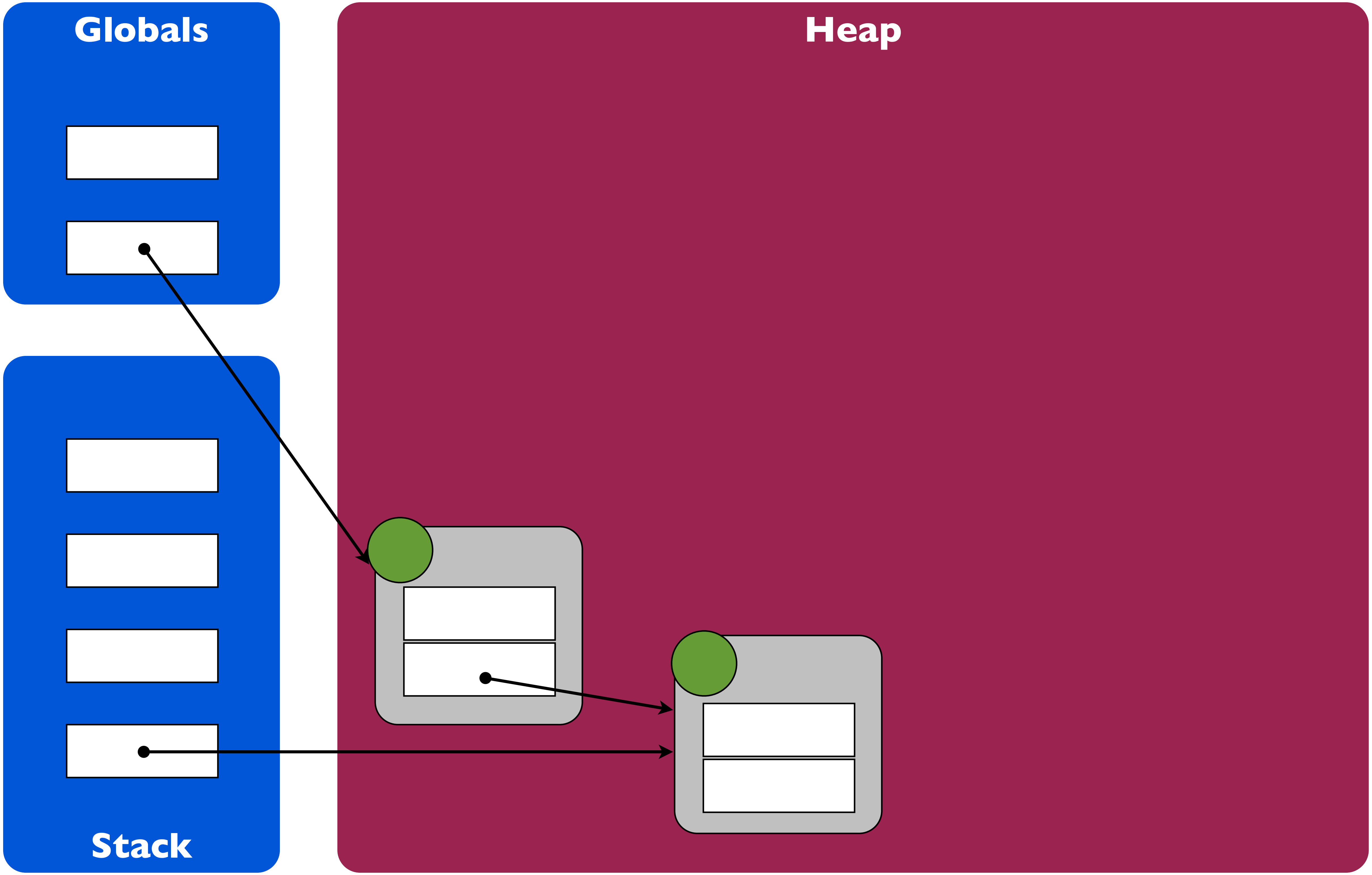
Mark & Sweep GC



Mark & Sweep GC



Mark & Sweep GC



Mark phase

Implemented as depth-first search of object graph

Natural recursive implementation

```
for each ref p in rootSet:
    mark(p)

mark(p) {
    if (*p marked) return
    mark *p
    for each reference-type field x in *p:
        mark(p->x)
}
```

Mark phase

```
stack = new Stack()
for each ref p in rootSet:
    stack.push(p)

while (! stack.empty) {
    p = stack.pop
    if (*p is marked) continue
    mark *p
    for each reference-type field x in *p:
        stack.push(p->x)
}
```

Mark phase

Question: what happens when we try to mark a long linked list while explicitly maintaining a stack?

Mark phase

Question: what happens when we try to mark a long linked list while explicitly maintaining a stack?

Very deep recursion => **stack overflow!**

Need to preallocate sufficient space for the stack

Mark phase

Can we do marking with **no** space overhead?

Mark phase

Can we do marking with **no** space overhead?

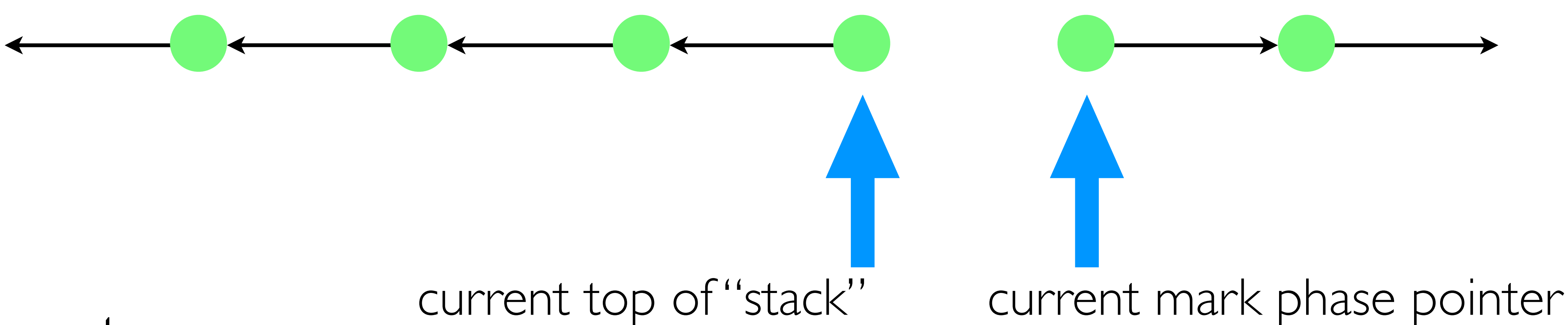
Yes!

(otherwise, I probably wouldn't have asked)

Deutsch-Waite-Schorr algorithm

Idea:

- reverse the pointers after following them – no stack needed!
- need a few bits per object to record which field to follow on next backtrack – can steal the low bits of the pointers



Disadvantage:

- objects are broken while being traversed
- mutator must be halted during mark phase
- => no concurrency allowed

Where to store the mark bits?

Can use bit vector to record marks on the side

- Advantage: don't have to touch (i.e., pollute the cache with) objects during sweep phase

Or store a mark bit in the object header

- add another word
- or use a bit of the dispatch table pointer
  - pointers aligned 4 have 2 free bits
  - need to mask off bits on method dispatch

Mark & Sweep GC

- Advantages
  - Automatic
  - Handles cycles!
- Drawbacks
  - Allocation cost
  - Collection cost (stop-the-world)
    - Free unreachable objects
  - Fragmentation

Cost of mark-sweep

Accesses all memory in use by the program

Mark phase reads only live (reachable) data

Sweep phase reads all of the data (live + garbage)

=> run time proportional to total amount of data!

=> can cause long program pauses!

What’s a pointer?

Root set consists of registers, stack slots, globals

To determine the root set, we need meta-information

- For each instruction in the program, which registers and stack slots point to the heap?
- Stored in so-called “GC maps”

Optimization

- Only have GC map for instructions where thread has to be able to stop for GC (GC safe points)
  - Loop back-edges (to bound waiting time)
  - Call sites
  - Allocation sites

Conservative collectors

Allocated storage contains both pointers and non-pointers

Is 22,022,592 an integer or an address?

Conservative collection:

- assume values are pointers unless they can't be (not in the range of the heap)
- safe, but not precise: treat non-pointers as pointers
- unsafe: treat pointers are non-pointers (might free some reachable objects)

requires no language support, no GC maps ==> works for C!

Boehm-Demers-Weiser collector

AKA Boehm-Weiser or BDW

http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Conservative mark-sweep GC for C, C++

Drop-in replacement for malloc

- `malloc = GC_malloc`
- `free =` no-op
- On Linux: `LD_PRELOAD=/usr/local/lib/libgc.so ./a.out`

Can also be used as a leak detector

Copying collection

Idea: use two heaps

- one in use by the program
- one sits idle until GC needs it

GC algorithm

- copy all live objects from active heap ("from-space") to the inactive heap ("to-space")
- dead objects are left in the from-space
- heaps then switch roles

Issue: must rewrite references between objects

The following algorithm is due to C.J. Cheney 1970

Cheney algorithm

Treat the to-space as a queue

Not this Cheney

Initialize the queue:

- Copy all objects referenced directly from roots to the to-space
- Leave a forwarding pointer in place of the old object

Dequeue an object

- Scan its pointer fields, copying uncopied children to the queue

When end of queue reached, flip spaces

Cheney algorithm

Treat the to-space as a queue

Initialize the queue:

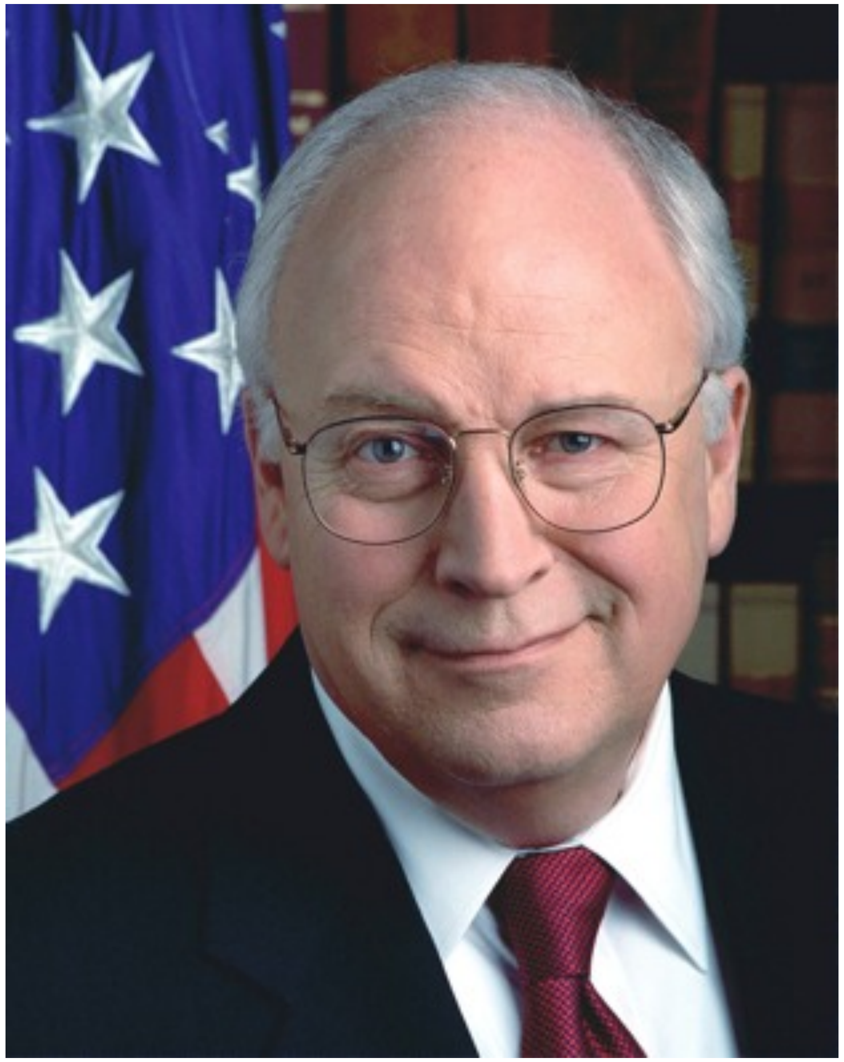
- Copy all objects referenced directly from roots to the to-space
- Leave a forwarding pointer in place of the old object

Dequeue an object

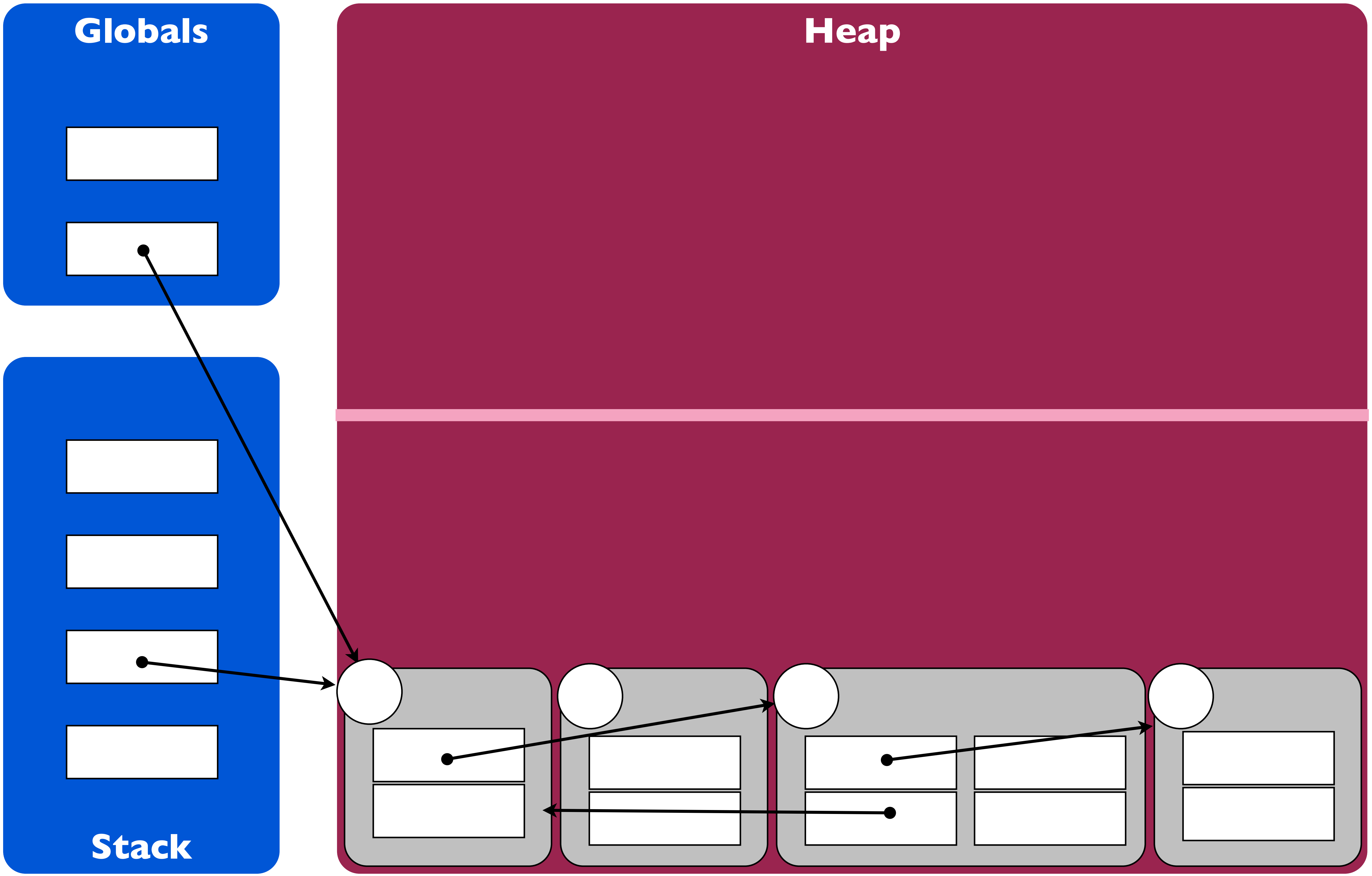
- Scan its pointer fields, copying uncopied children to the queue

When end of queue reached, flip spaces

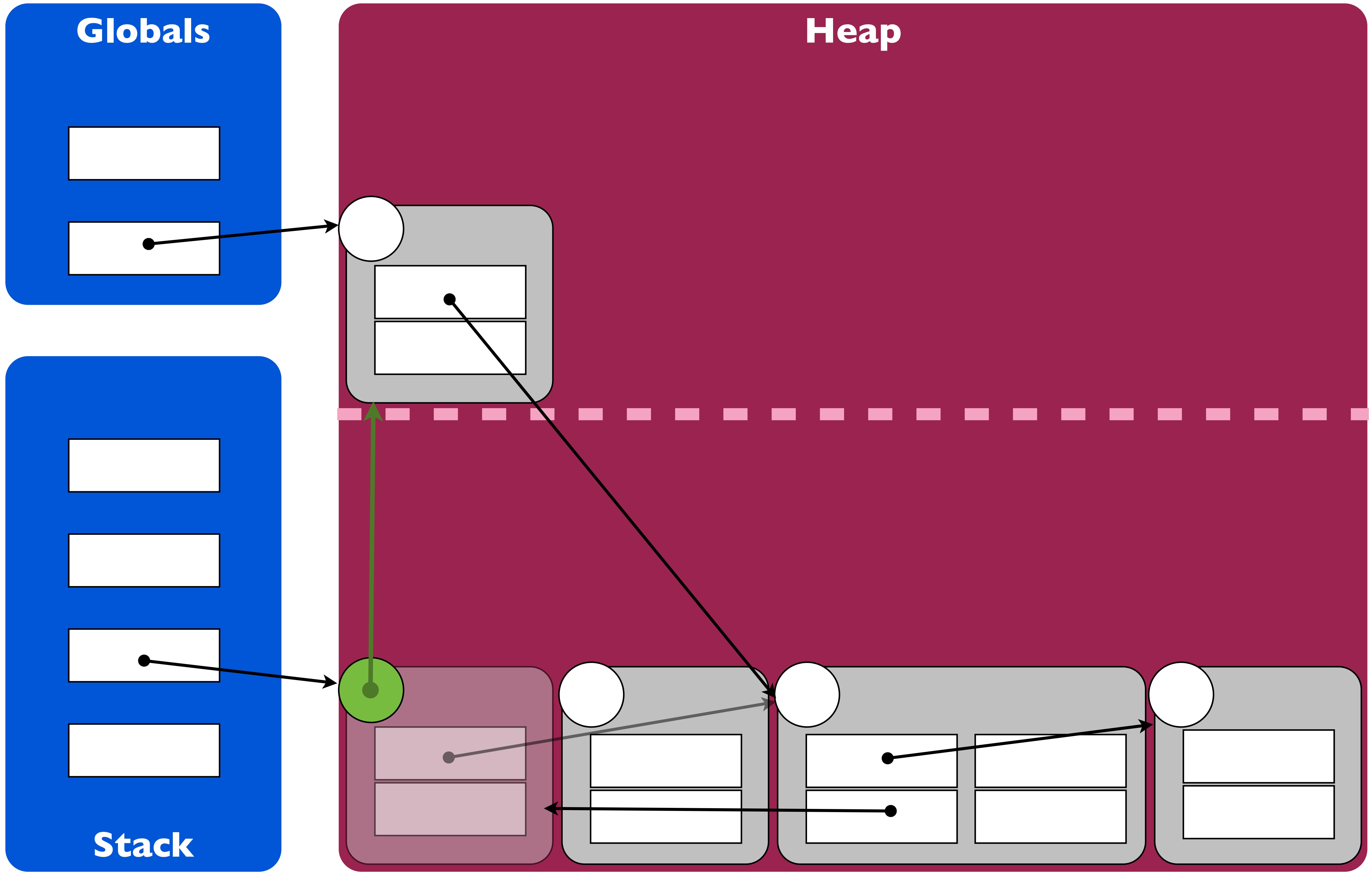
Not this Cheney



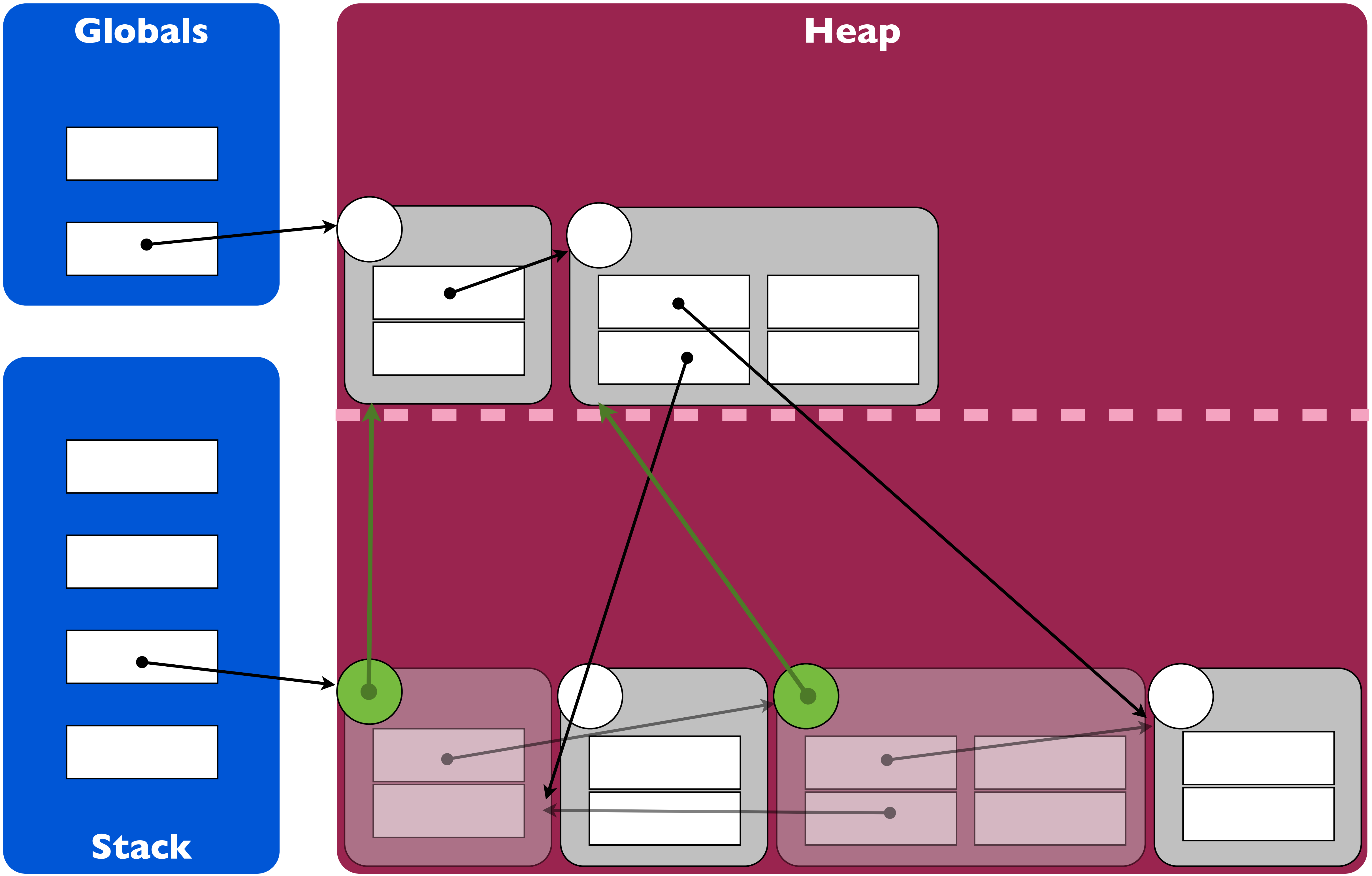
Copying GC



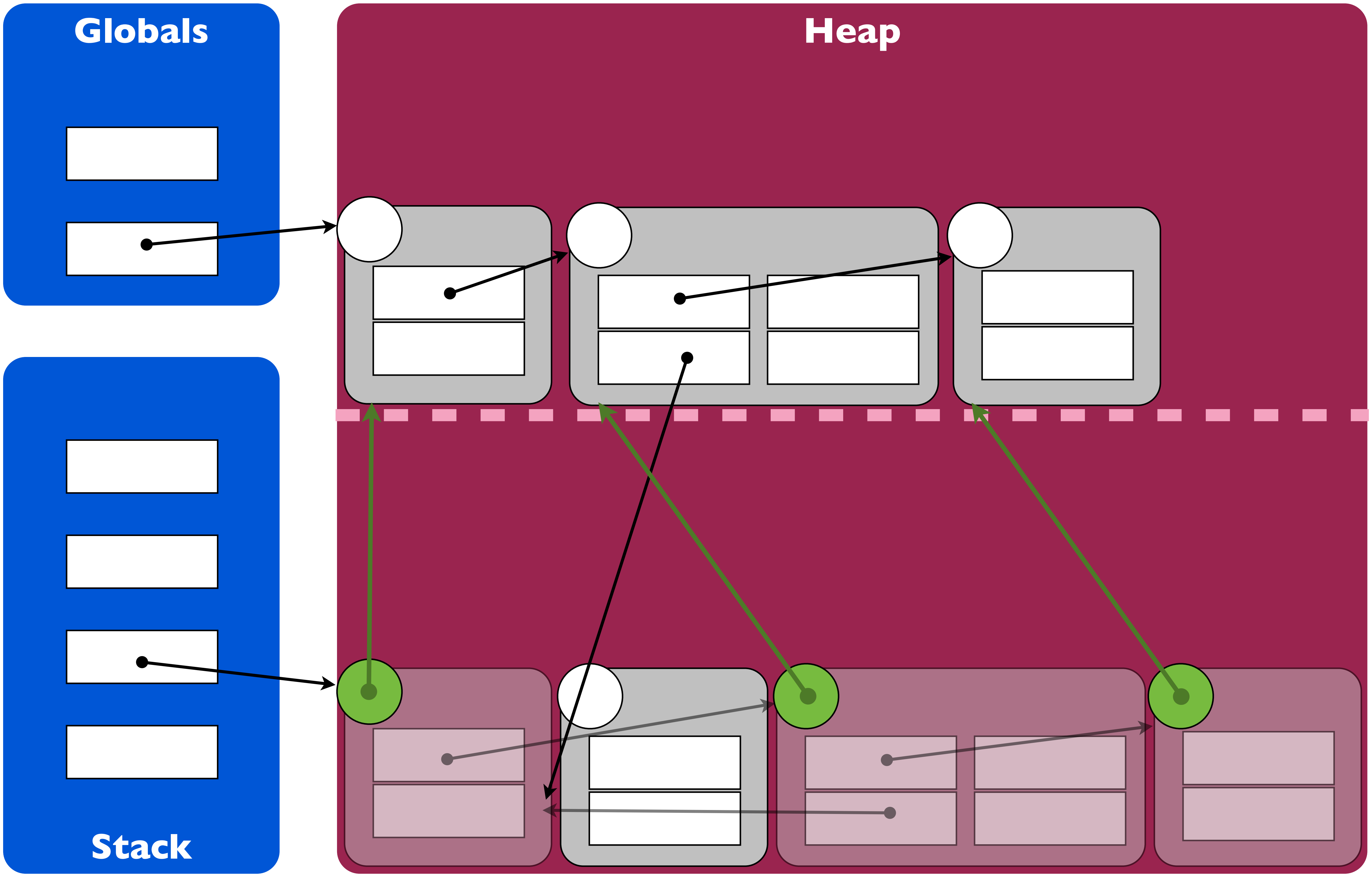
Copying GC



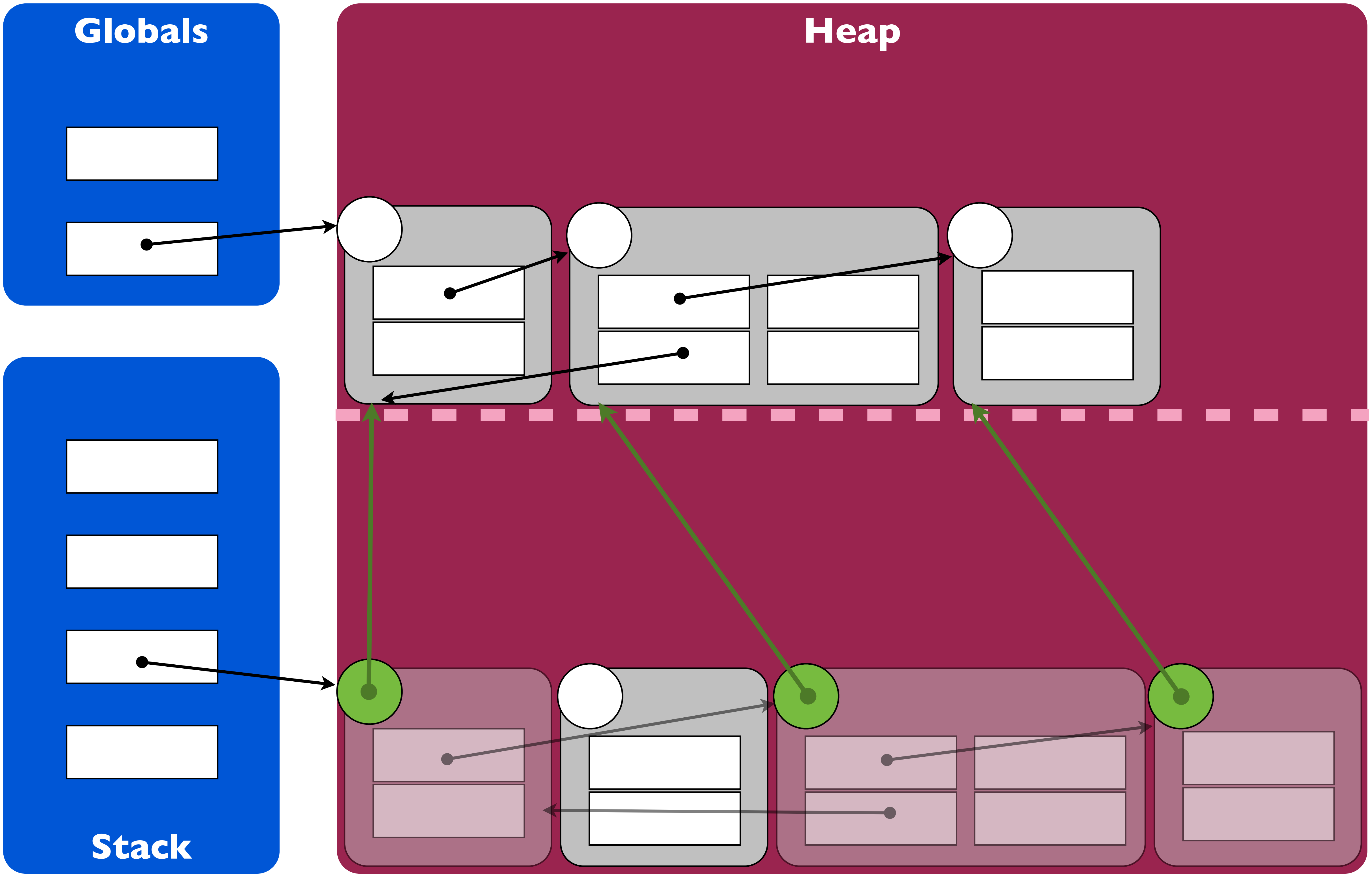
Copying GC



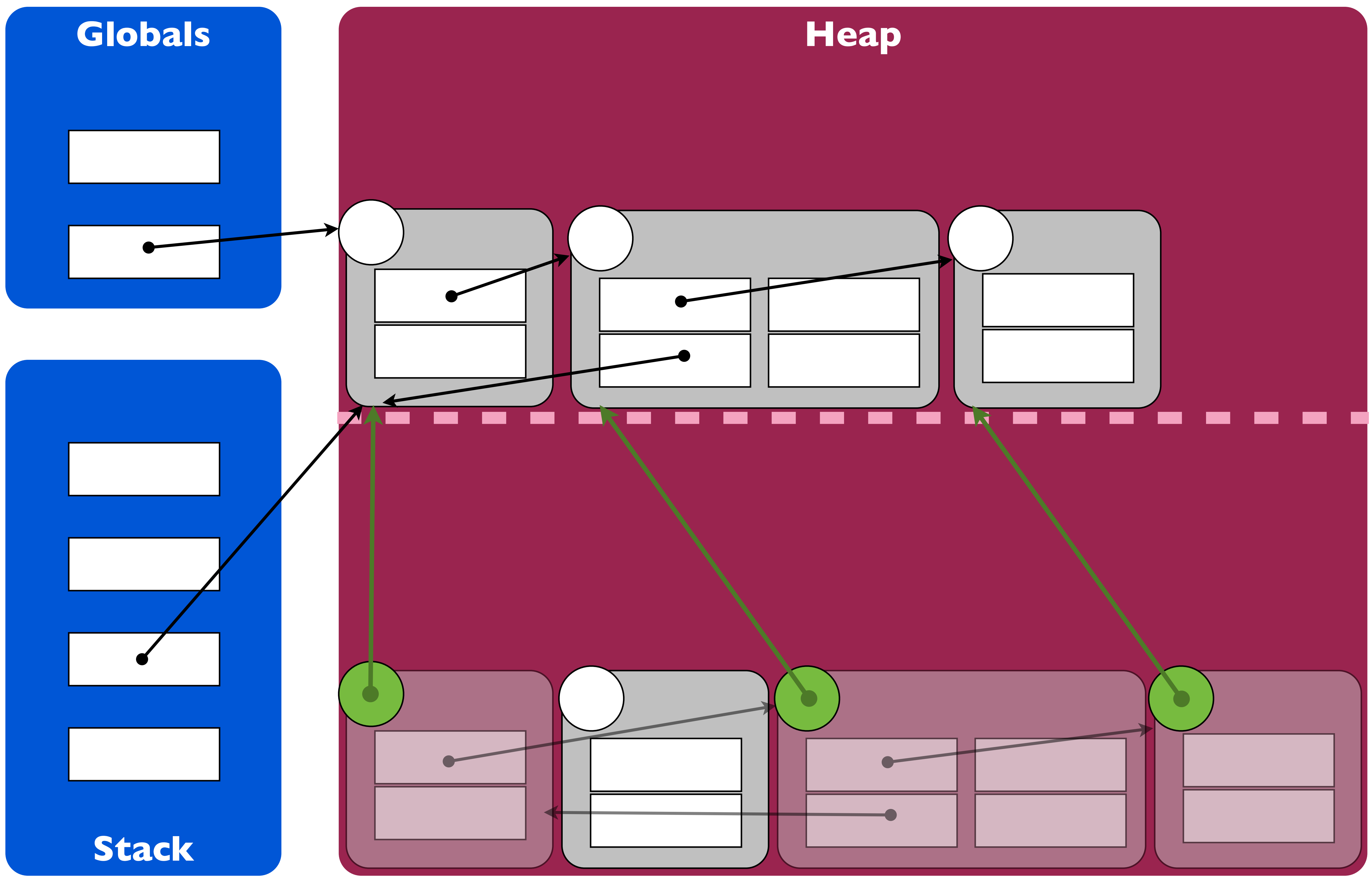
Copying GC



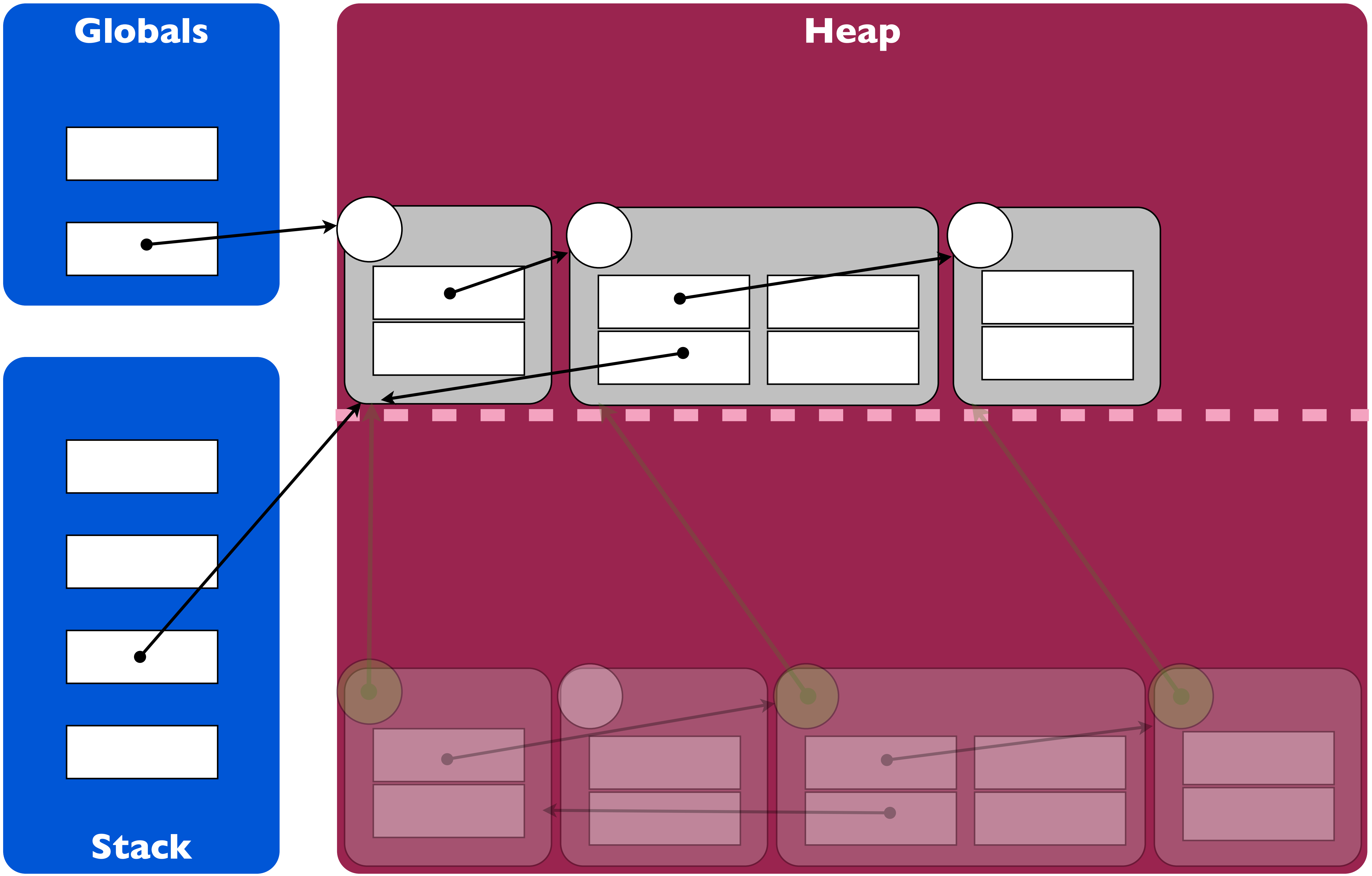
Copying GC



Copying GC



Copying GC



Benefits

Simple, no stack space needed

Run time proportional to number of live objects

Automatically compacts, eliminating fragmentation

Bump pointer allocation

- malloc(n) implemented as tail = tail + n

Implementation issues

Precise pointer information required

- difficult to use on languages like C
  - (but there are "mostly copying" GC algorithms)

Uses twice as much memory

- but: its *virtual* memory
- still: might be inappropriate to use copying GC in embedded systems

Copying GC

- Advantages
  - Fast allocation (bump pointer, like stack)
  - Fast free (no cost)
  - No fragmentation
  - Improves locality!
- Drawbacks
  - Collection cost (stop-the-world)
    - Copy reachable objects
  - Only half the heap available

Generational GC

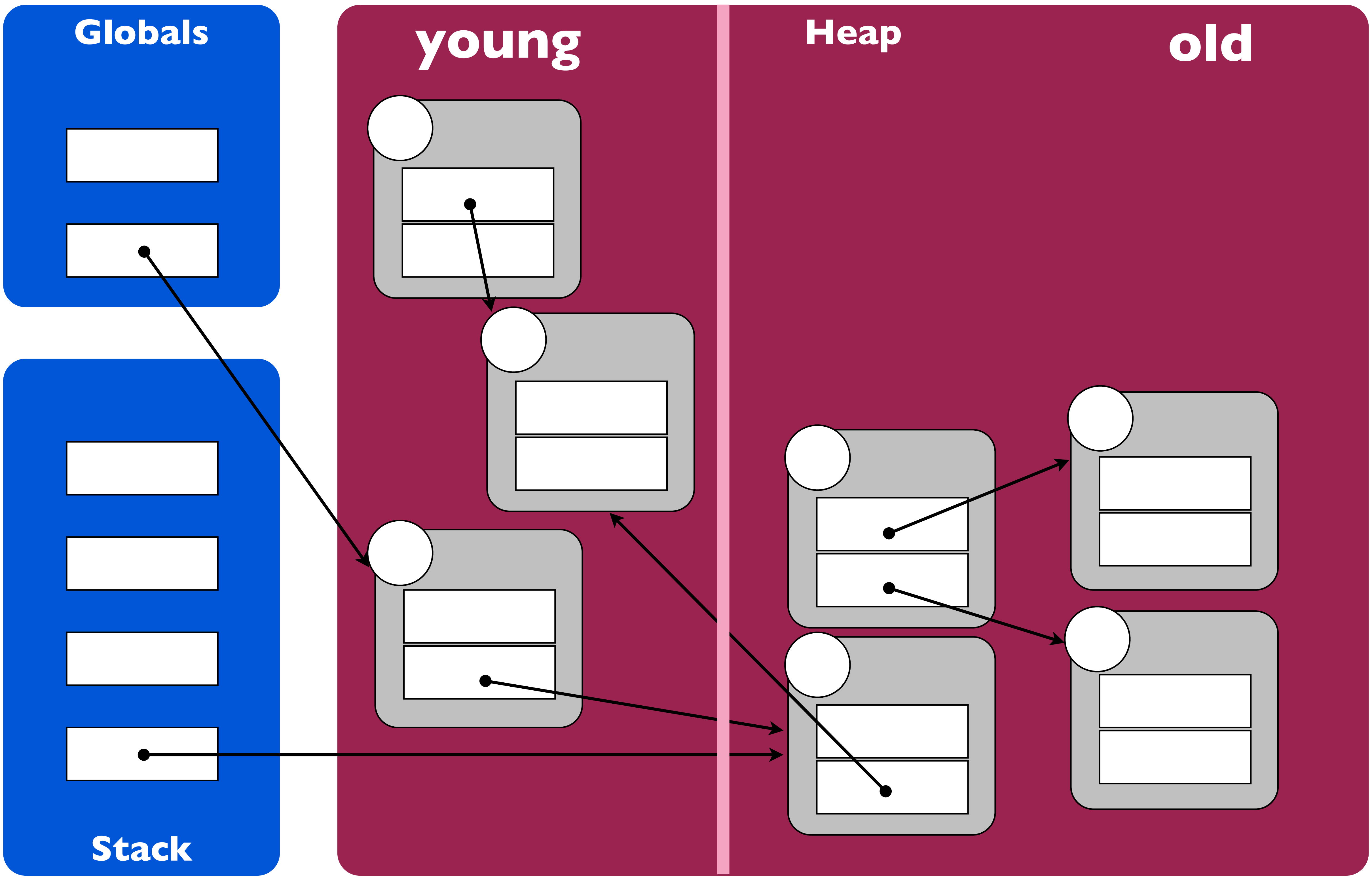
Observation:

- if an object has been reachable for a long time, it is likely to remain so
  - globals, objects referenced from main function
- most objects die young

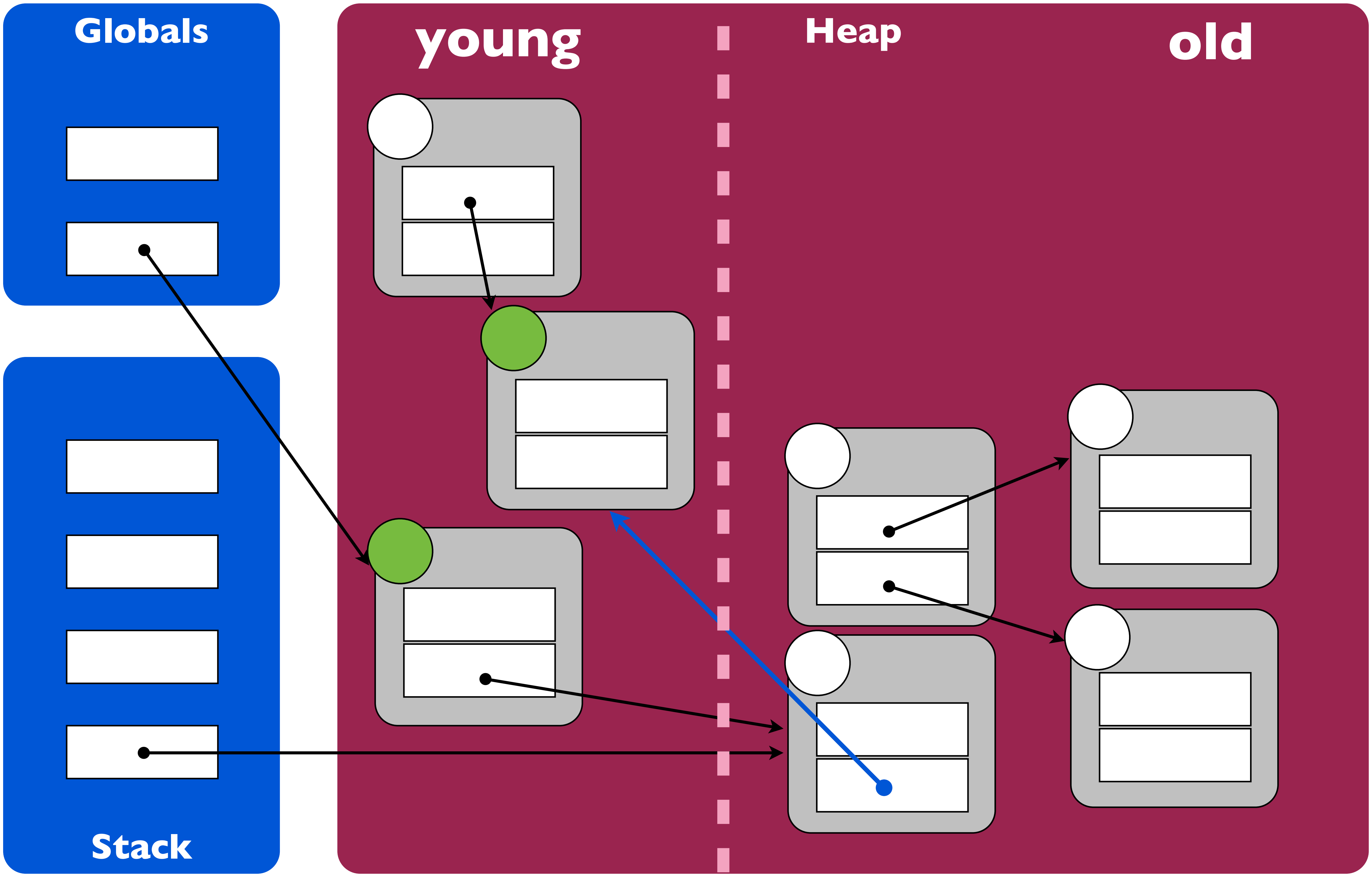
```
String toString() {
  String s = "";
  for (x : this)
    s += x;
  return s;
}
```

In a long-running system, mark-sweep, copying collection wastes time by scanning/copying older objects

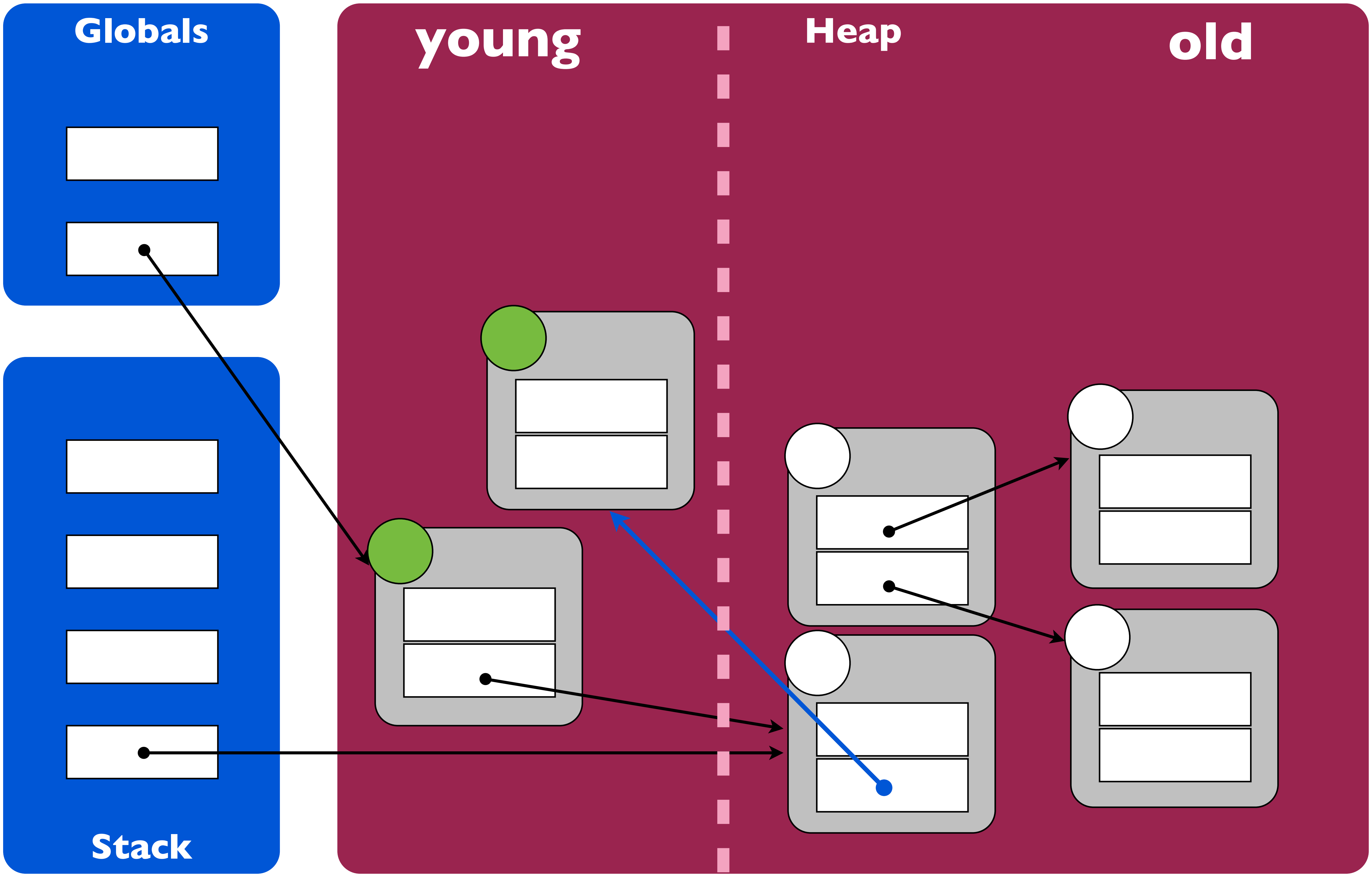
Generational GC



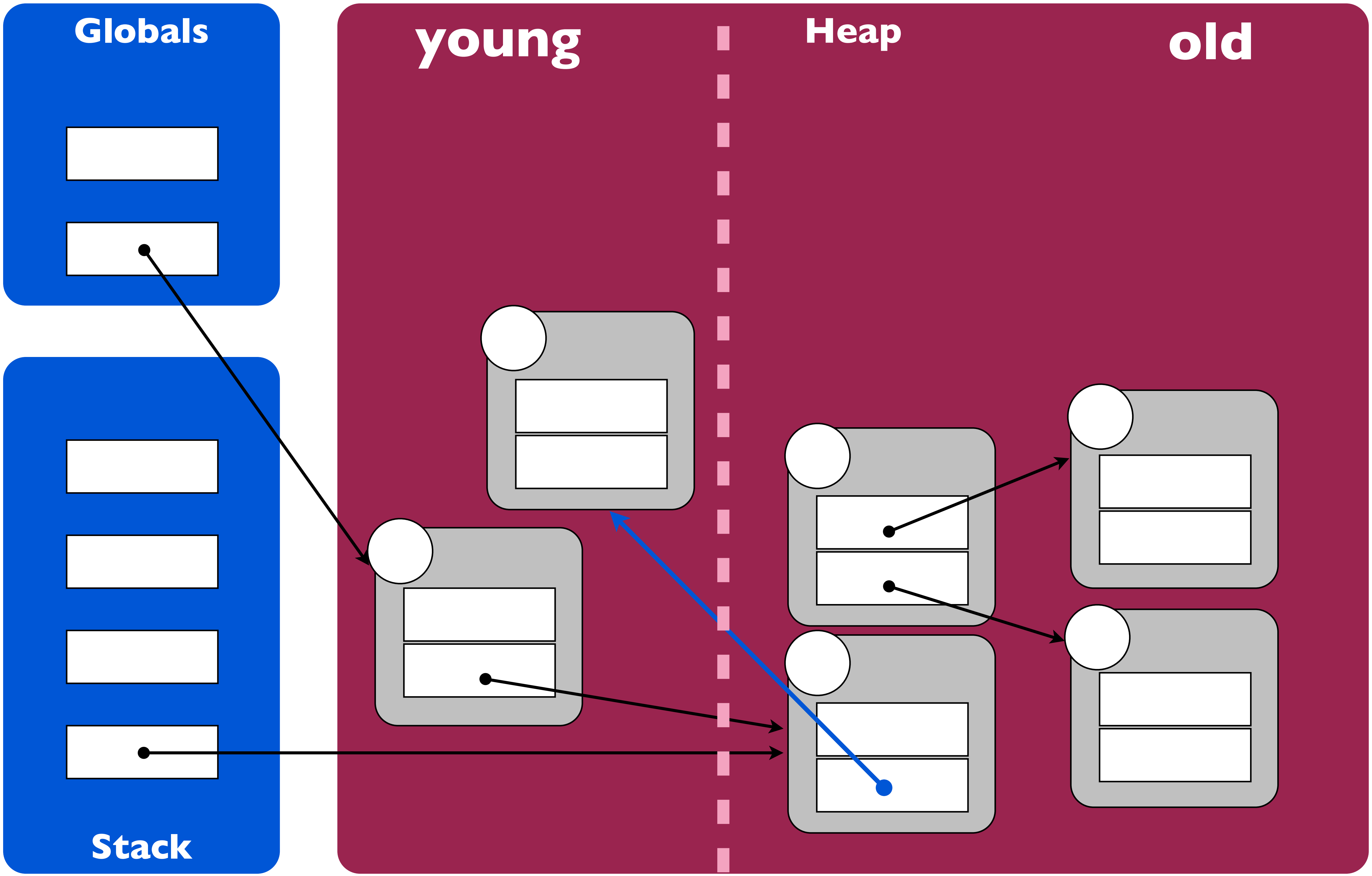
Generational GC



Generational GC



Generational GC



Generational GC

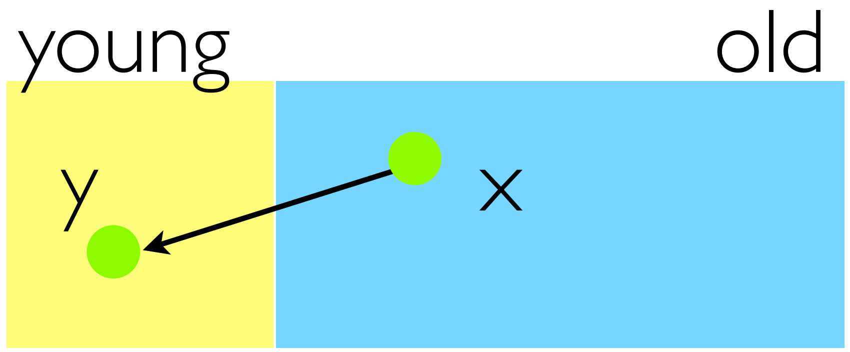
- Advantage
  - Most objects die young:
    few get tenured to old generation:
    can mostly only collect young generation
  - Young generation (nursery) can be much smaller and that old generation – less wasted space
- Disadvantage
  - Write barrier (slows down mutator!)
    Track all writes to reference fields

Exercise

Construct a program that behaves poorly with generational GC

Construct a program that behaves well with generational GC

Roots of the nursery?



Tenured objects might point to new objects

How to avoid scanning them all?

Roots of the nursery?

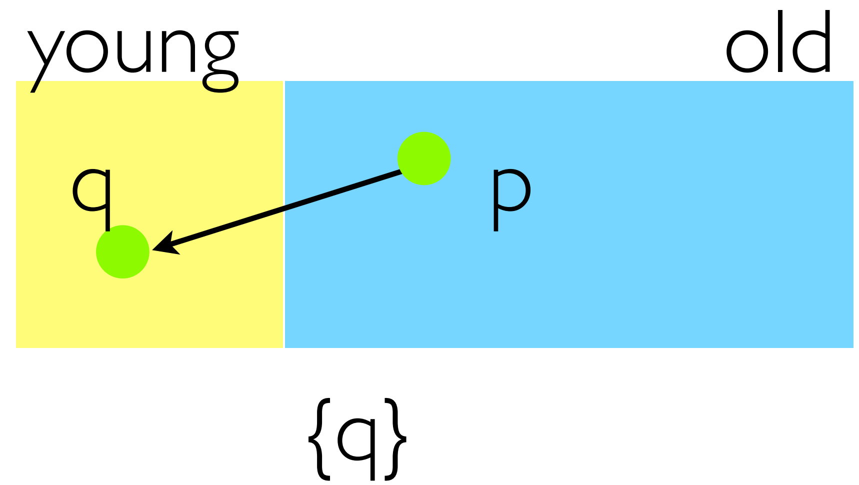
In practice, few tenured objects point to new objects

- unusual for an object to point to a newer object
- can only happen if older object is modified long after creation to point to a new object

Keeping track of pointers from old generation to new

- remembered sets
- card marking

Remembered sets



Want to identify:

- p.f = q
- p is tenured, y is not

Write barrier

- When storing a pointer (q) into a field (f) of an old object (p), record the pointer q in a **remembered set**

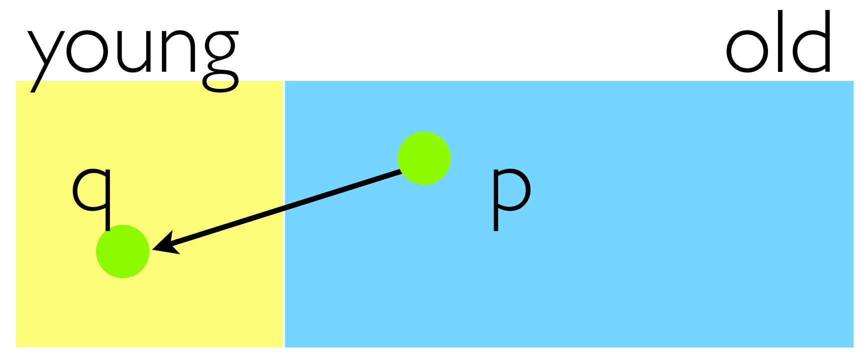
Root set of young generation

- now stack + registers + globals + remembered set

Card marking

Want to identify:

- p.f = q
- p is tenured, y is not



Divide memory into cards of 2^k words (say k = 7..9)

Maintain a bit vector with one bit per card

Write barrier

- When storing a pointer (q) into a field (f) of an old object (p), **mark the card** containing the object p

Root set of young generation

- stack + registers + globals + pointers in marked cards

Pros and cons

Advantage of card marking over remembered sets:

- faster, simpler write barrier

Disadvantage:

- less precise – all pointers on a marked card treated as roots for young gen, not just cards in remembered set
- Smaller cards: more precise, but card table takes more space

Incremental GC

GC might have to “stop the world”

- pause mutator while collecting
- can be unacceptable for interactive applications or real-time applications
- emacs: "garbage collecting..." status message

Incremental GC

- interleave GC and mutator
- GC a bit, run mutator a bit, GC a bit, ...

Concurrent GC

- Run GC in a separate thread in parallel with mutator

Modern incremental GC = very fast

GC pause times

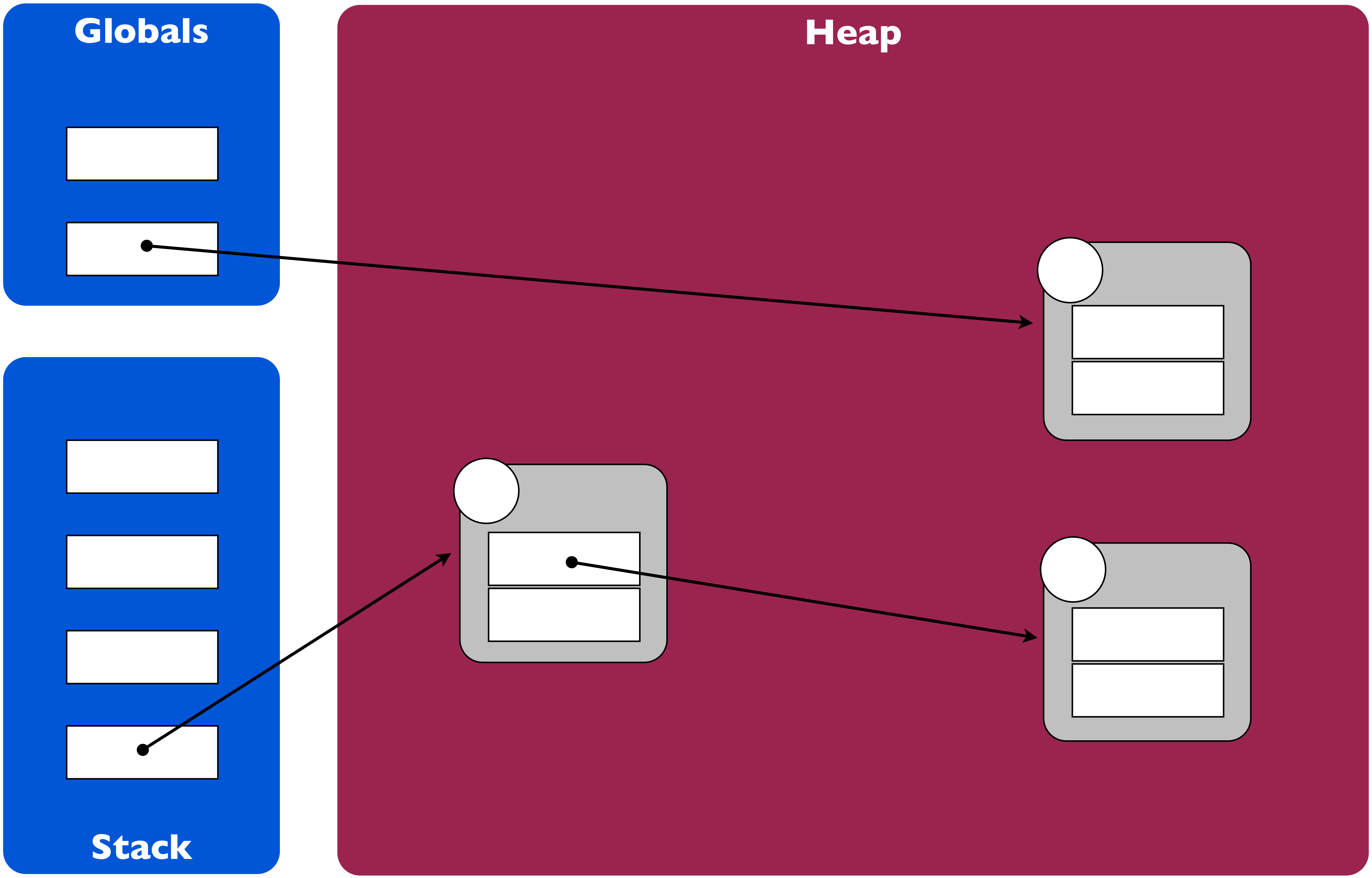
Azul:

- 100GB heap
- pause times < 10 ms

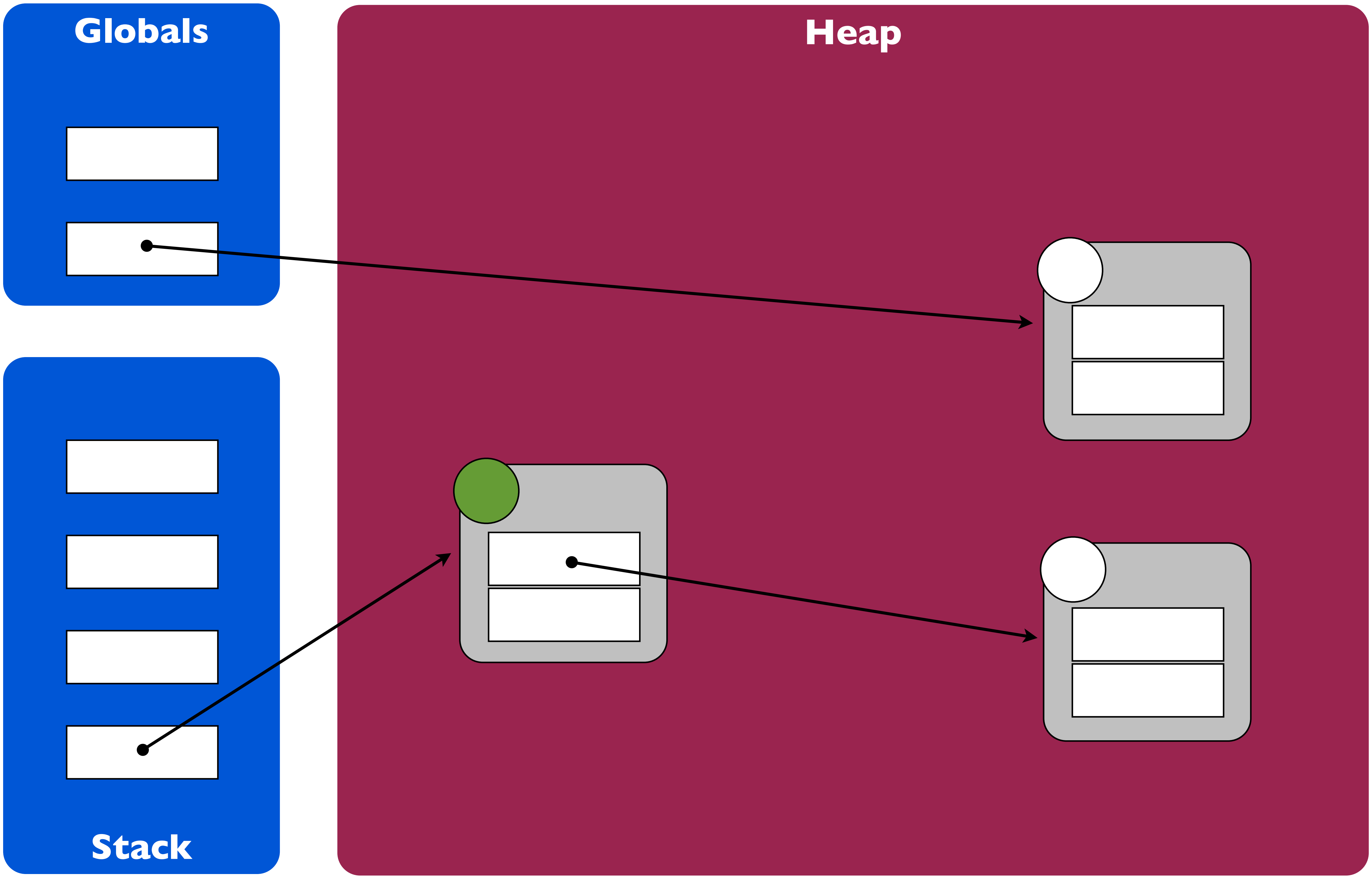
IBM:

- 100s of MB heap
- pause times < 10 microseconds

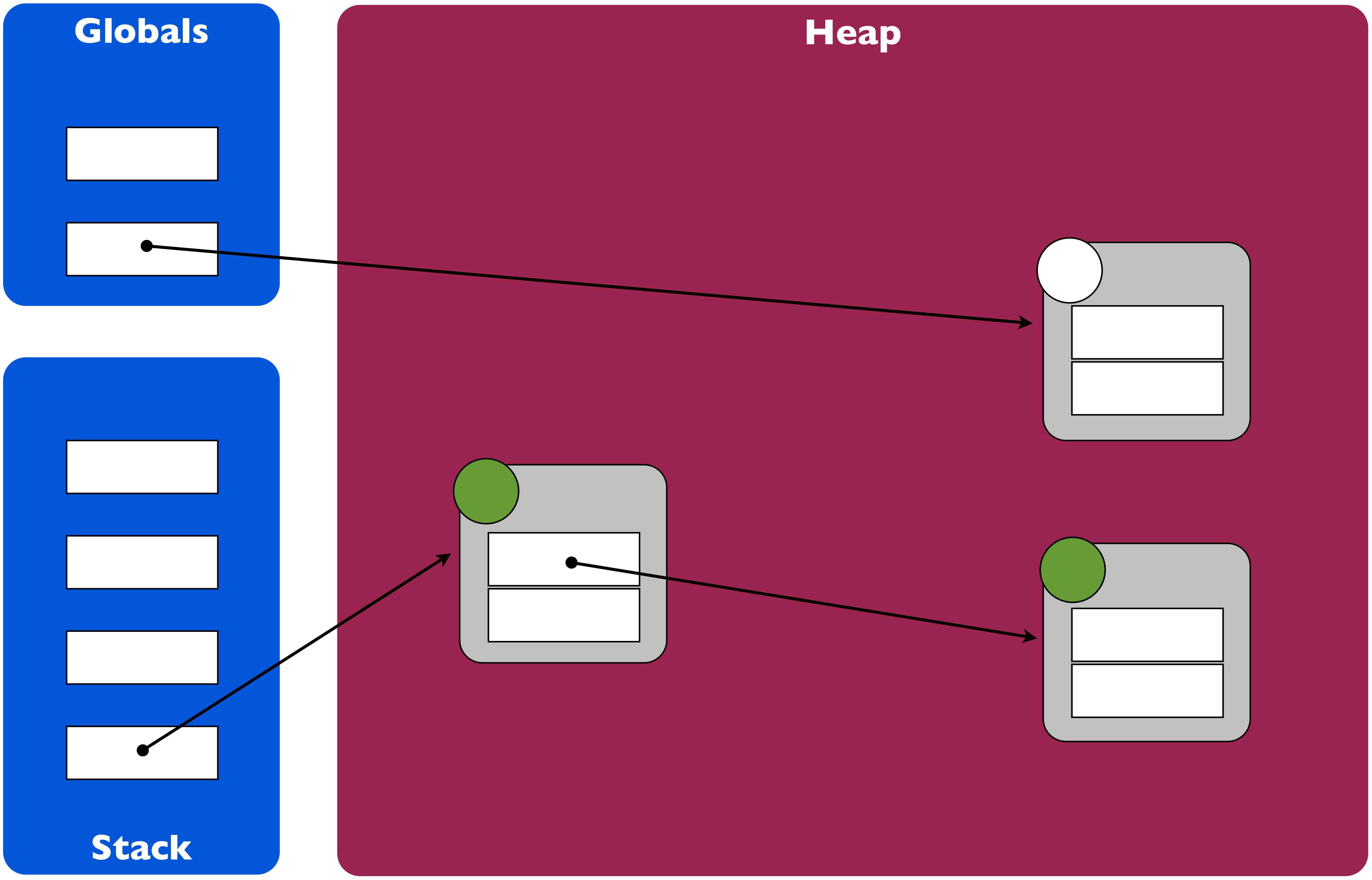
Concurrent GC problem



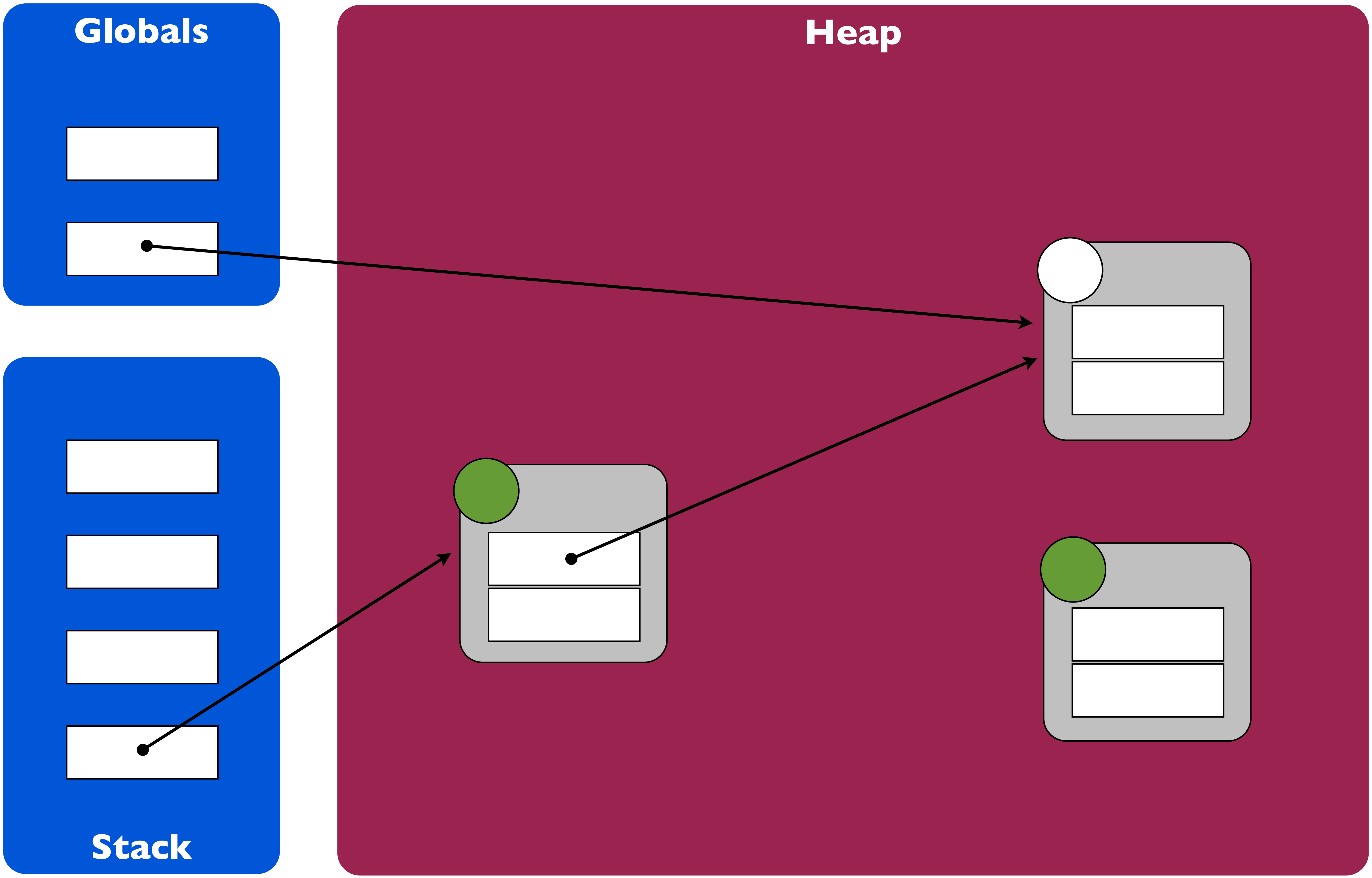
Concurrent GC problem



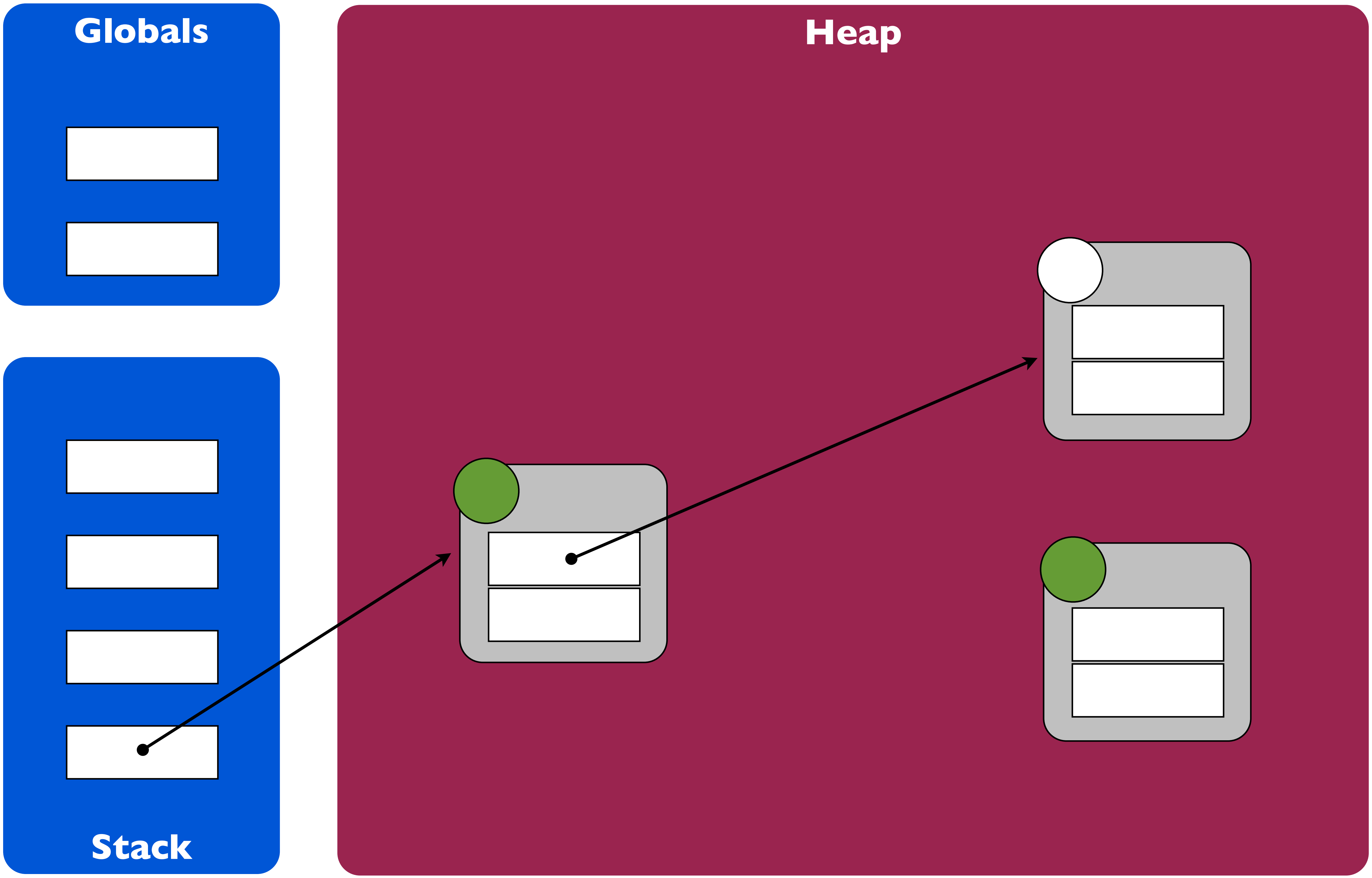
Concurrent GC problem



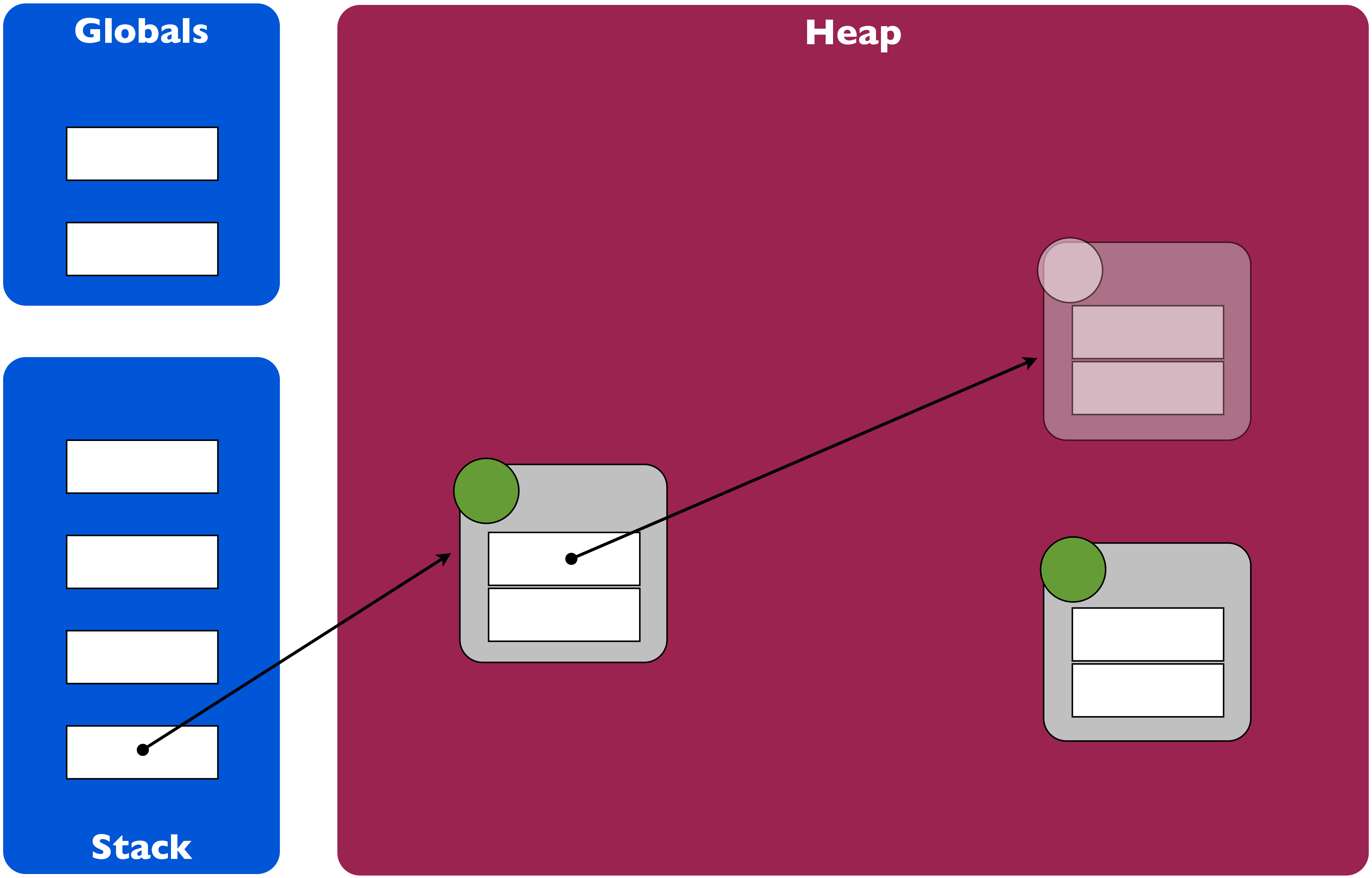
Concurrent GC problem



Concurrent GC problem



Concurrent GC problem



Performance

Conventional wisdom

GC is worse then `malloc` because...

- extra processing
- poor cache performance
- bad page locality
- increased footprint (delayed reclamation)

Conventional wisdom

GC improves performance by...

- faster allocation
  (fast path inlining & bump pointer allocation)
- better cache performance
  (object reordering)
- improved page locality
  (heap compaction)

Reality

Matthew Hertz, Emery Berger, Quantifying the Performance of Garbage Collection vs. Explicit Memory Management, 2007

Best collector performs as well as or better than malloc

- up to 10% faster on some benchmarks

... but uses more memory

- at least twice
- sometimes 5x

GC good if:

- system has a lot of RAM

GC bad if:

- limited RAM
- competition for physical memory
- RAM relied upon for performance
  - in-memory databases, search engines, ...

Object pooling

- manage your own freelists
- usually a bad idea: overhead much more than GC overhead

Marking values null to free early

- good idea (if careful)

Summary

GC simplifies interfaces

Reduces memory errors

Performance often as good as or better than malloc/free