Mark and sweep

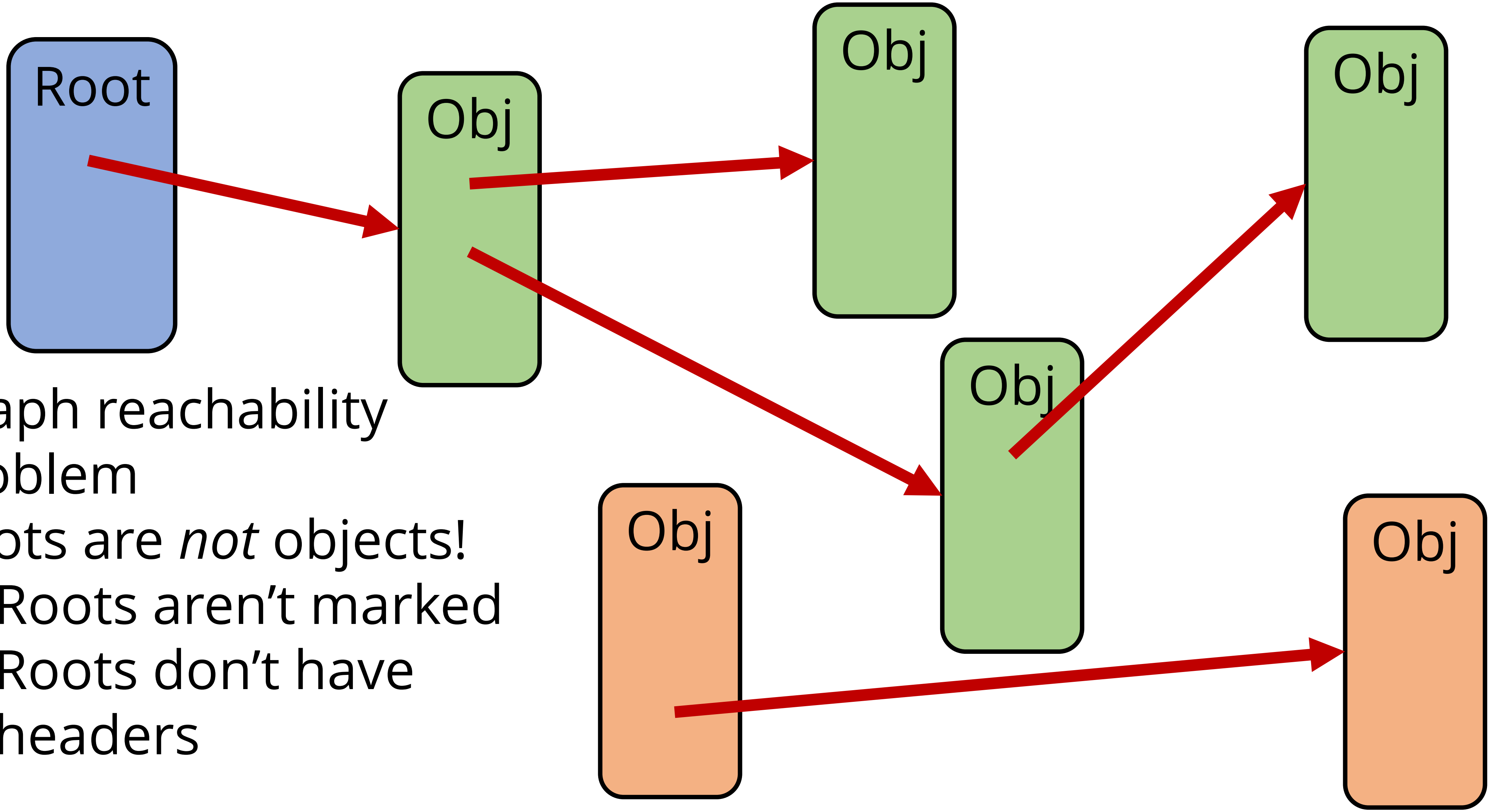
Review

- Memory manager: Allocation and revocation
- Revocation linked to allocation
- Scan heap for reachable objects, sweep to free unreachable ones

Review

- Mutator yields
- Collector decides when to collect
- Collector controls allocation
- “Stop the world”: Collector in complete control of heap

Marking



- Graph reachability problem
- Roots are *not* objects!
  - Roots aren't marked
  - Roots don't have headers

The mark algorithm (one version)

```
markPhase():
   worklist := new Queue
   foreach loc in roots:
      ref := *loc
      if ref != NULL and !marked(ref):
         mark(ref)
         worklist.push(ref)
         markWorklist()

markWorklist():
   while (ref := worklist.pop()):
      foreach loc in ref->header.descriptor->ptrs:
         child := *(ref+loc)
         if child != NULL and !marked(ref):
            mark(ref)
            worklist.push(child)
```

Root task very different from object task!

Scan order

- Presented algorithm:
  - Follows root pointers to completion before moving on to another root pointer
  - Is breadth-first for heap objects

This should make you angry!

Scan order

- Objects often form cliques
- Object cliques:
  - Are allocated around the same time
  - Mostly point at each other
  - Should be allocated near each other

Scan order: Address-first?

- We could sort worklist by ref address
- Time to sort usually overwhelms saved time scanning

Mark bit

- Without mark bit, graph reachability trace may never end!
- Mark bit can be in header…
- Or, can keep a side table
- If in header: Where to put the bit?

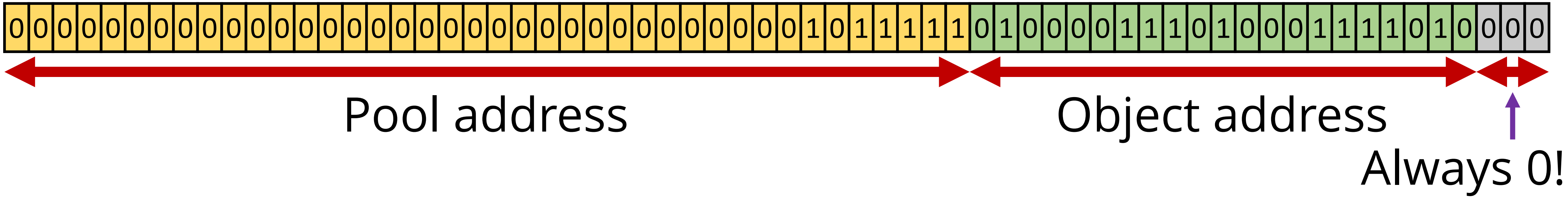
Mark bit

```
struct ObjectHeader {
   struct GCTypeInfo *typeInfo;
   char markBit;
};

void mark(struct ObjectHeader *hdr) {
   hdr->markBit = 1;
}

int isMarked(struct ObjectHeader *hdr) {
   return hdr->markBit;
}
```

How much larger are objects when this is added?

Bit-sneaky

- There are three[1] wasted bits in our header



- *All* pointers have this extra space!

[1] On 32-bit systems, two

Bit-sneaky C

```
struct ObjectHeader {
  struct GCTypeInfo *typeInfo;
};

void mark(struct ObjectHeader *hdr) {
  hdr->typeInfo = (struct GCTypeInfo *)
    ((size_t) hdr->typeInfo | 1);
}

int isMarked(struct ObjectHeader *hdr) {
  return (size_t) hdr->typeInfo & 1;
}
```

Worth it?

- If objects are small (hint: they are), every word counts
- Huge complication: Type info pointer is no longer valid!
- Must restore type info pointer later

Sweep

- Heap parsability is crucial!
- Consider heap parsability with:
  - Bump-pointer allocation
  - Free-list overallocation
  - Free object type/header

Sweep algorithm

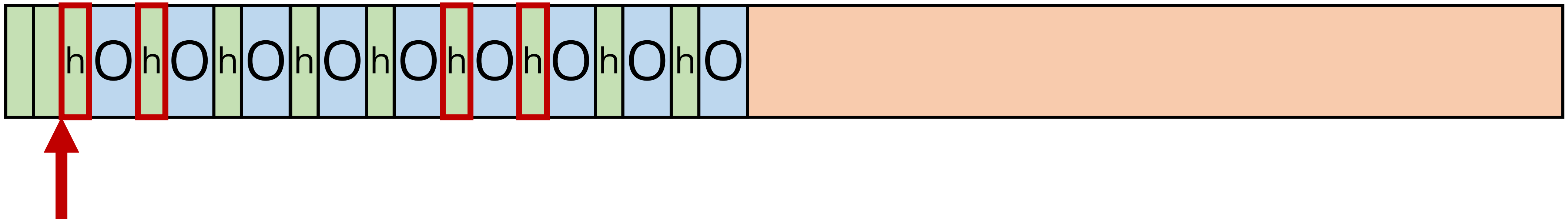
```
sweep():
  freeList := new FreeList
  foreach ref in heap:
    if marked(ref):
      unmark(ref)
    else:
      ref *:= new FreeObject
      freeList.push(ref)
```

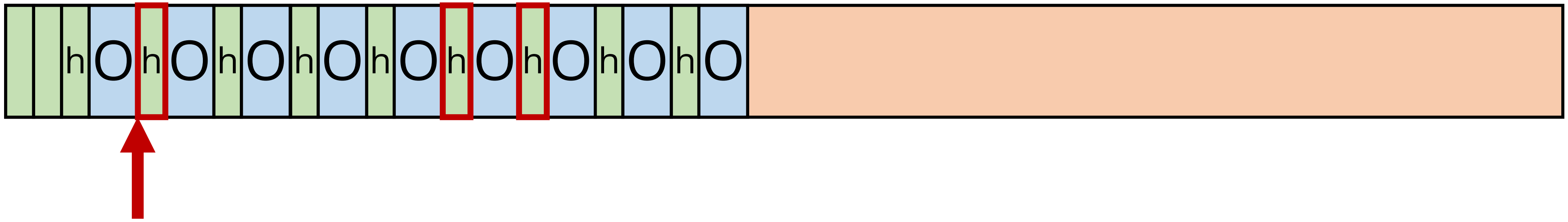
Discard old freelist

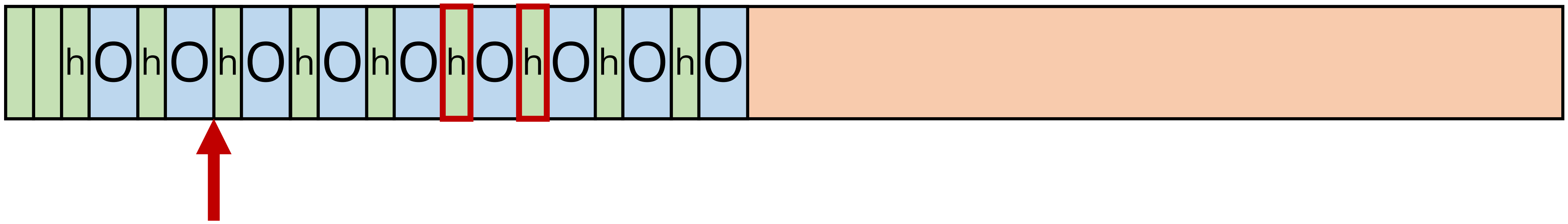
Must walk entire heap!

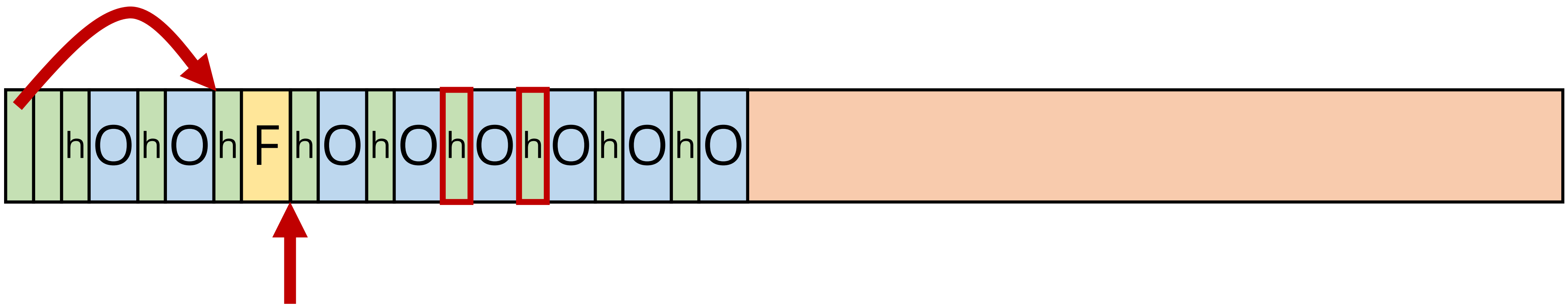
Perfect chance to unmark

Type of objects change in sweep

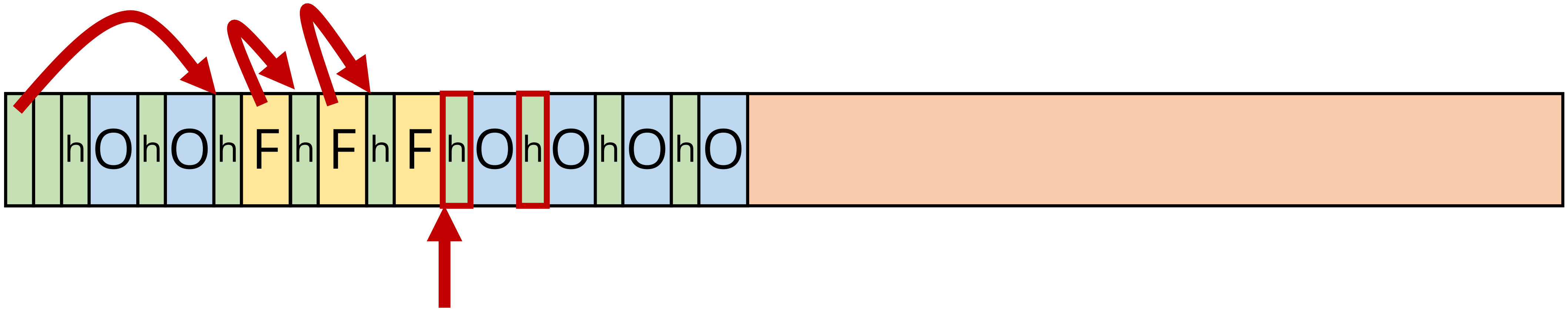

h O h O h O h O h O h O h O h O h O

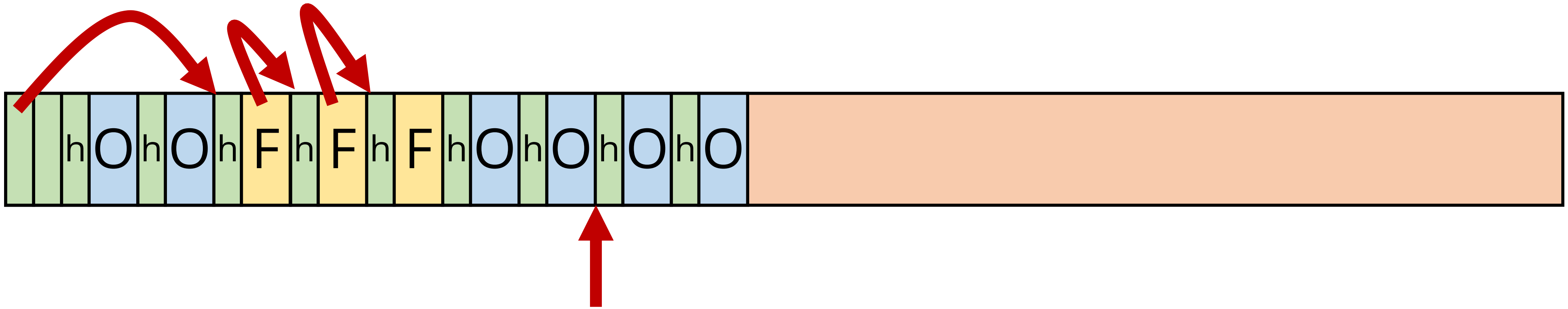

h O h O h O h O h O h O h O h O h O


h O h O h O h O h O h O h O h O h O

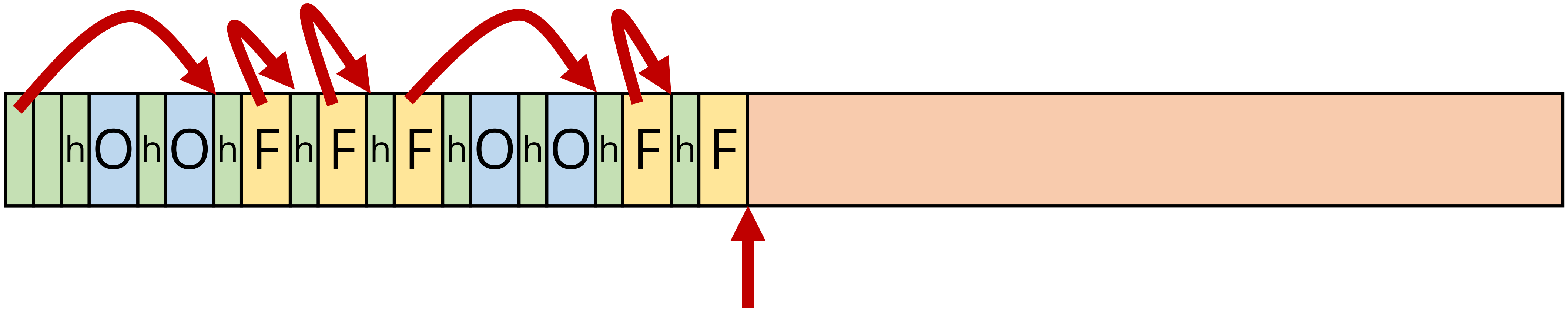


h O h O h F h O h O h O h O h O h O


h O h O h F h F h F h O h O h O h O



h O h O h F h F h F h O h O h O h O


h O h O h F h F h F h O h O h F h F

Performance

- Mark: O(L)
- Sweep:
- and-sweep: O(H)

Locality, locality, locality!

Bit-swapping

- Can avoid cost of clearing bits by swapping meaning:
  - In first collection, 0 = unreachable, 1 = reachable,
  - in second collection, 1 = unreachable, 0 = reachable, etc.
- Must remember to allocate with correct mark!

Improving mark

- Depth-first vs. breadth-first vs. address-ordered
- Bitmapped mark
- Other tricks beyond scope of course

Bitmapped mark

- Connected to bitmap free-list:
  - Bitmap at beginning of pool
  - Clear bitmap before marking
  - One bit per word
  - If object is alive, mark its words in bitmap
  - Use as bitmap free-list during allocation
- With bit-swapping, *no sweep*

Improving sweep

- It's not so bad (locality!)
- Improve by:
  - Even better cache behavior,
  - concurrent/lazy sweeping, or
  - O(1) sweep

Sweep cache behavior

- Stride of sweep always object size
- CPUs prefetch
- Object size varies
- Segregated blocks: Object size constant, perfect prefetch

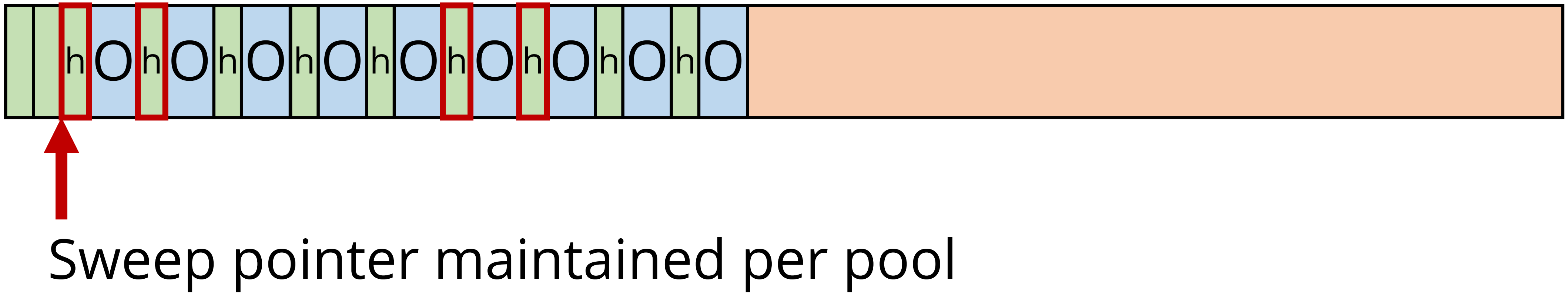
Concurrent sweep

- Mutator will never touch unmarked objects
- Sweep in a separate thread
- Must be careful about allocation/sweep races!

Lazy sweep

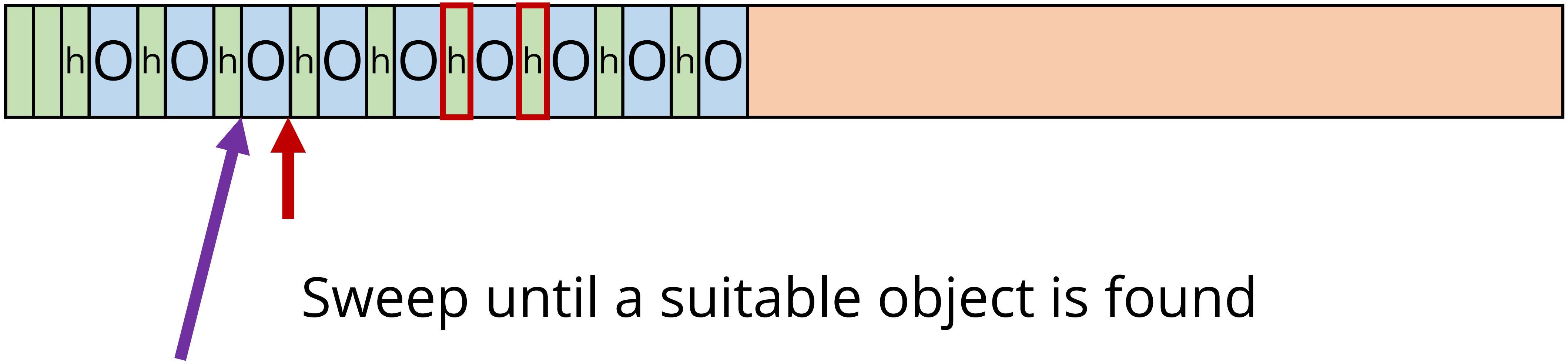
- Sweep during allocation
- If free-list is empty, sweep until sufficient free object is found
- Insufficient objects added to free-list

Lazy sweep



When allocating, if free-list is empty or has no suitable objects…

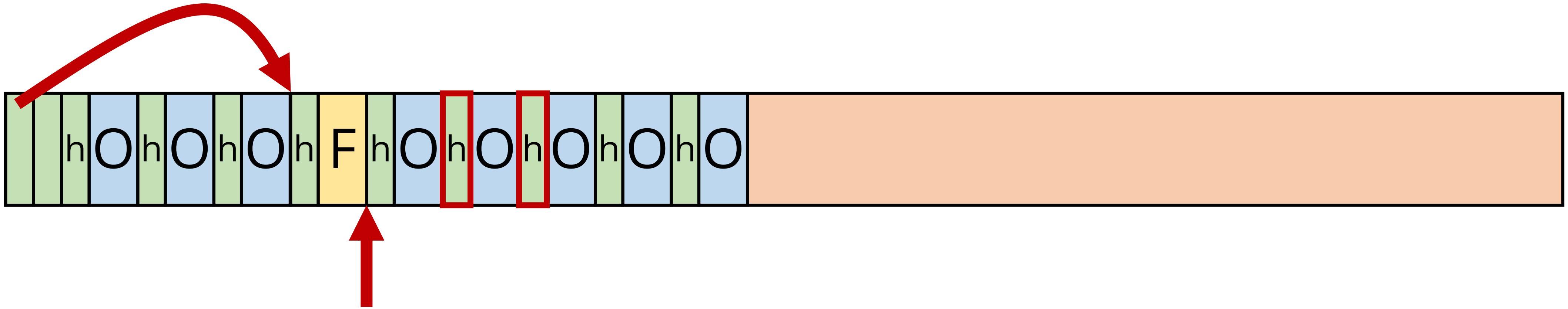
Lazy sweep



Sweep until a suitable object is found

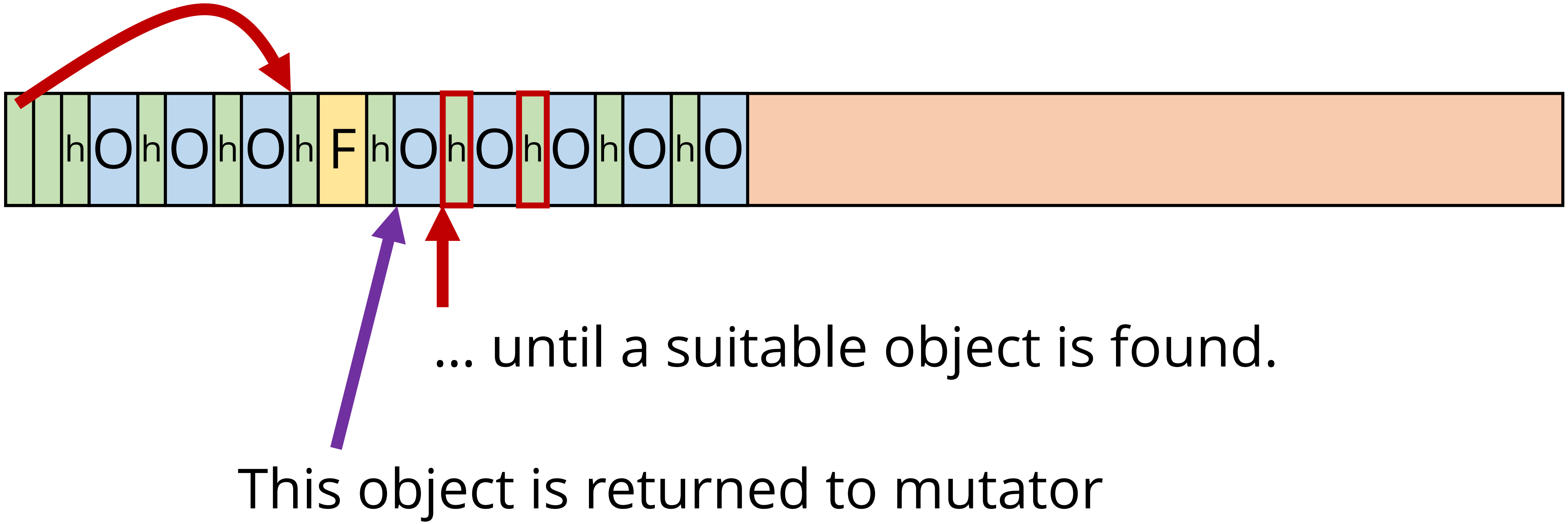
This object is returned to mutator

Lazy sweep



Unsuitable objects added to free-list during allocation

Lazy sweep


h
O
h
O
h
O
h
F
h
O
h
O
h
O
h
O
h
O
... until a suitable object is found.
This object is returned to mutator

Lazy sweep performance

- Throughput
- Responsiveness
- Latency
- Resource utilization
- Fairness

O(1) sweep

- Walking the heap is O(H)
- Appending lists is O(1)
- Keep “allocated list”
- Mark by moving to new list

When to GC

- Must GC if:
  - Free-list is empty,
  - no free space in any pool, and
  - OS cannot give any more space.
- Should GC far more often than that

When to GC

- Typical strategy is to GC when:
  - An allocation is made that cannot be satisfied without requesting a new pool, or
  - traversing free-list is becoming expensive.
- Requires active monitoring

Free-list monitoring

- Depends on free-list type
- For, e.g., first-fits list, count number of hops during allocation
- Frequent many-hop allocations = fragmentation

When to GC

- If every full pool leads to GC, no new pools allocated
- Must allocate new pools when collection leaves pools mostly full

Pool space

- To gauge used pool space, simply sum size of all reachable objects
- Due to fragmentation, free pool space is not a perfect indicator of available space
- Might use free-list monitoring too

Summary

- Mark-and-sweep is exactly how it sounds
- Sweep seems expensive but has great locality
- Optimizations can reduce or eliminate sweep

Moving GC

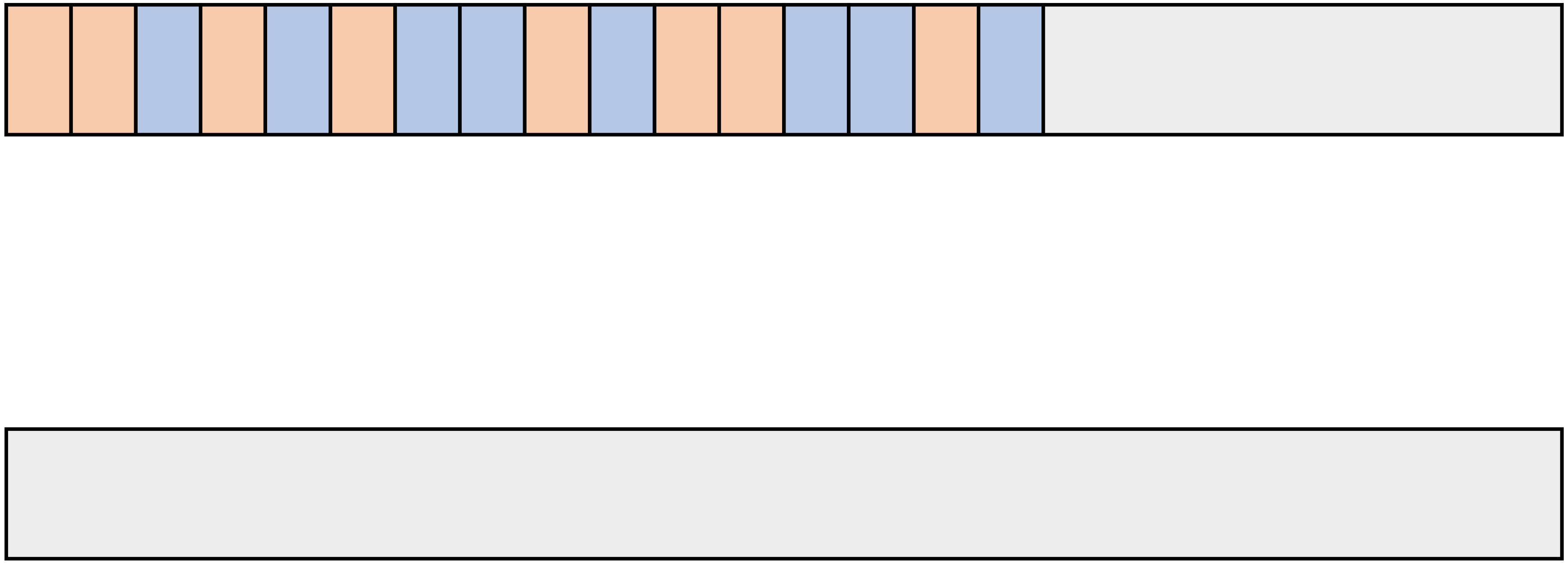
Review

- Allocator owns pools
- Compiler controls roots
- Compiler informs allocator of roots, object types
- Trace references to find living objects

Mark and sweep

- Very natural map to reachability
- Two passes
- Prone to fragmentation

Semispace copying



Semispace copying

- “fromspace” and “tospace”
- After moving from fromspace to tospace, no reachable objects in fromspace
- Swap from/to space for new collection
- No sweep, free-lists, fragmentation

Implications

- Isn’t moving objects expensive?
  - L <<< H
- Must update all references
- Must never copy twice
- Can only use half of heap (allocate in tospace)

```
collect():
  fromspace, tospace := tospace, fromspace
  worklist := new Queue
  foreach loc in roots:
    process(loc)
  while (ref := worklist.pop()):
    scan(ref)

scan(ref):
  foreach loc in ref->header.descriptor->ptrs:
    process(ref+loc)

process(loc):
  fromRef := *loc
  if fromRef != NULL:
    *loc := forward(fromRef)

forward(fromRef):
  if alreadyMoved(fromRef):
    return forwardingAddress(fromRef)
  toRef := (allocate in tospace)
  memcpy(toRef, fromRef, fromRef->header.size)
  setForwardingAddress(fromRef, toRef)
  worklist.push(toRef)
  return toRef
```

Queue?

- Yet again, algorithm shown is queue
- Object cliques still real
- Stack actually *improves* locality of object cliques!

Even better moving

- Can we predict the best way to arrange objects?
- No. NP-complete even with access pattern oracle.

Forwarding

- Naïve:

```
struct ObjectHeader {
  struct TypeInfo *typeInfo;
  void *forward;
};
```

Forwarding

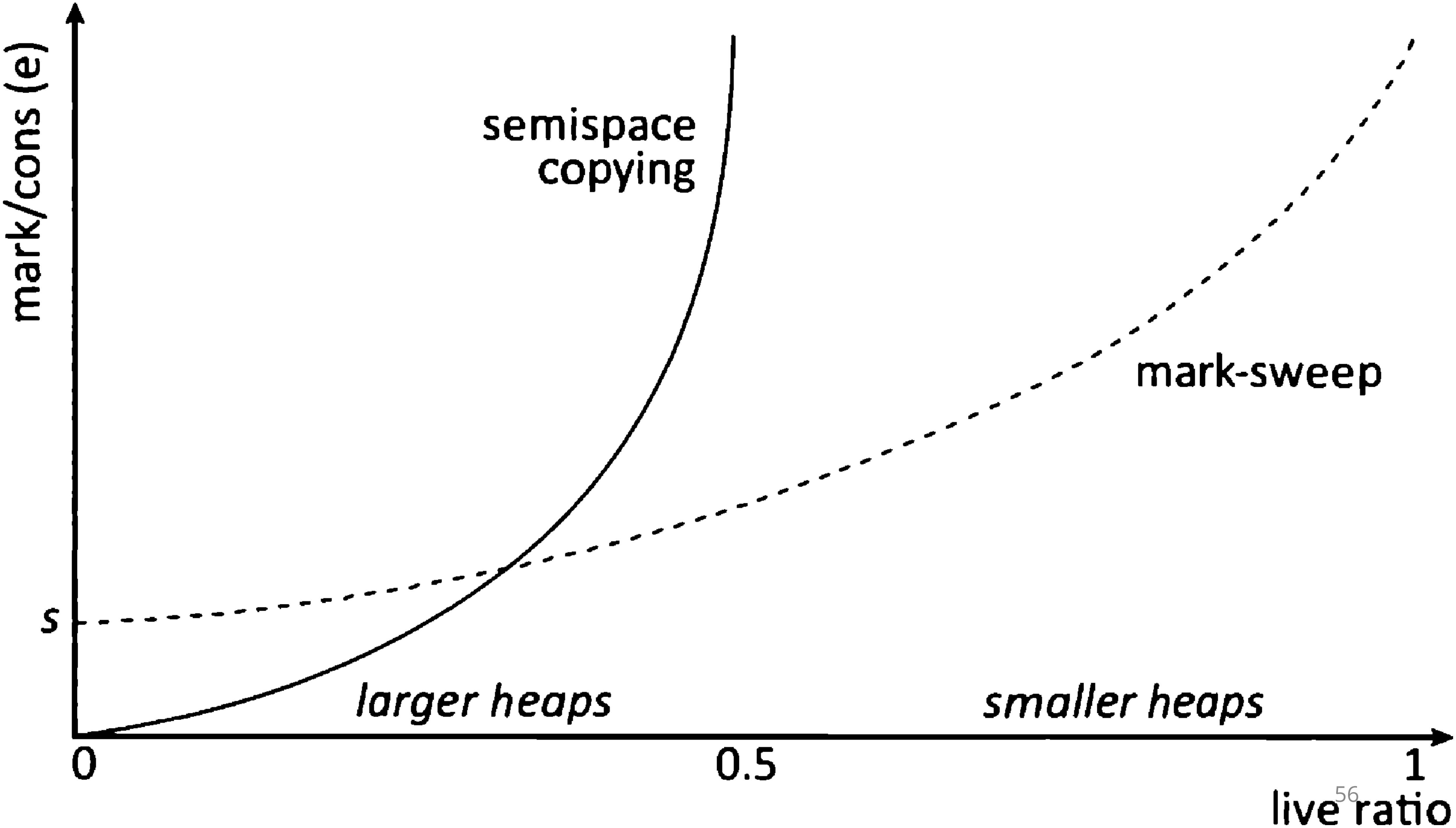
- Still have those extra bits!
- Once an object is forwarded, no longer need type info
- Careful: Pointer wrong in two ways

Allocation

- To Hell with free-lists!
- Bump-pointer is fast and sufficient
- No overallocation, fragmentation, coalescence, complex data structures…

When to collect

- Half as much active heap
- Double resource utilization, or
- collect twice as often


mark/cons (e)
semispace
copying
mark-sweep
s
larger heaps
smaller heaps
0
0.5
1
live ratio

When to GC?

- Typically: When tospace is full
- GC takes O(L)
- (L is a constant for most programs)

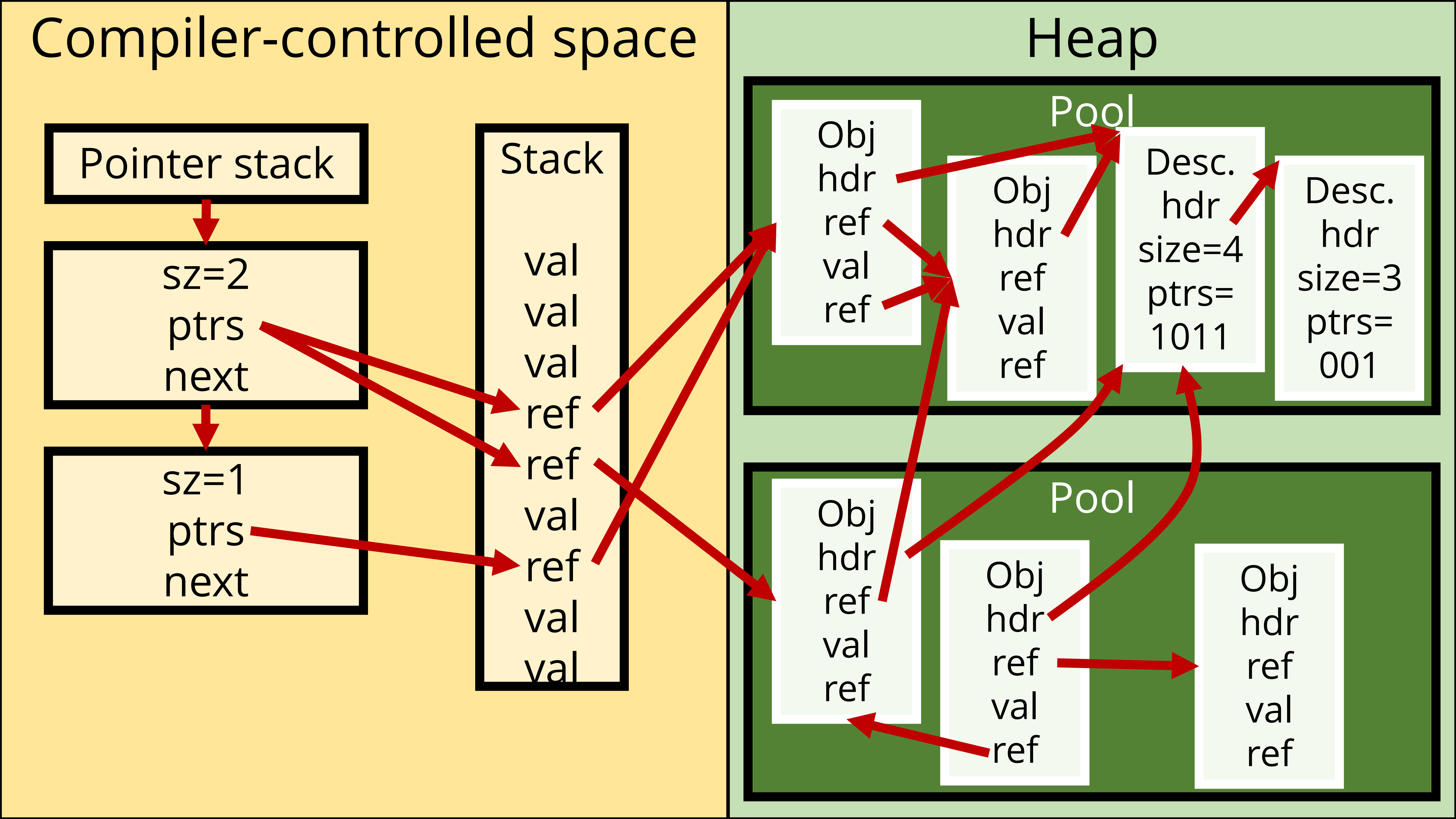
Allocating pools

- Must keep two sets of pools
- Always allocate in both!
- Tospace “mirrors” fromspace, but don’t need individual frompools and topools

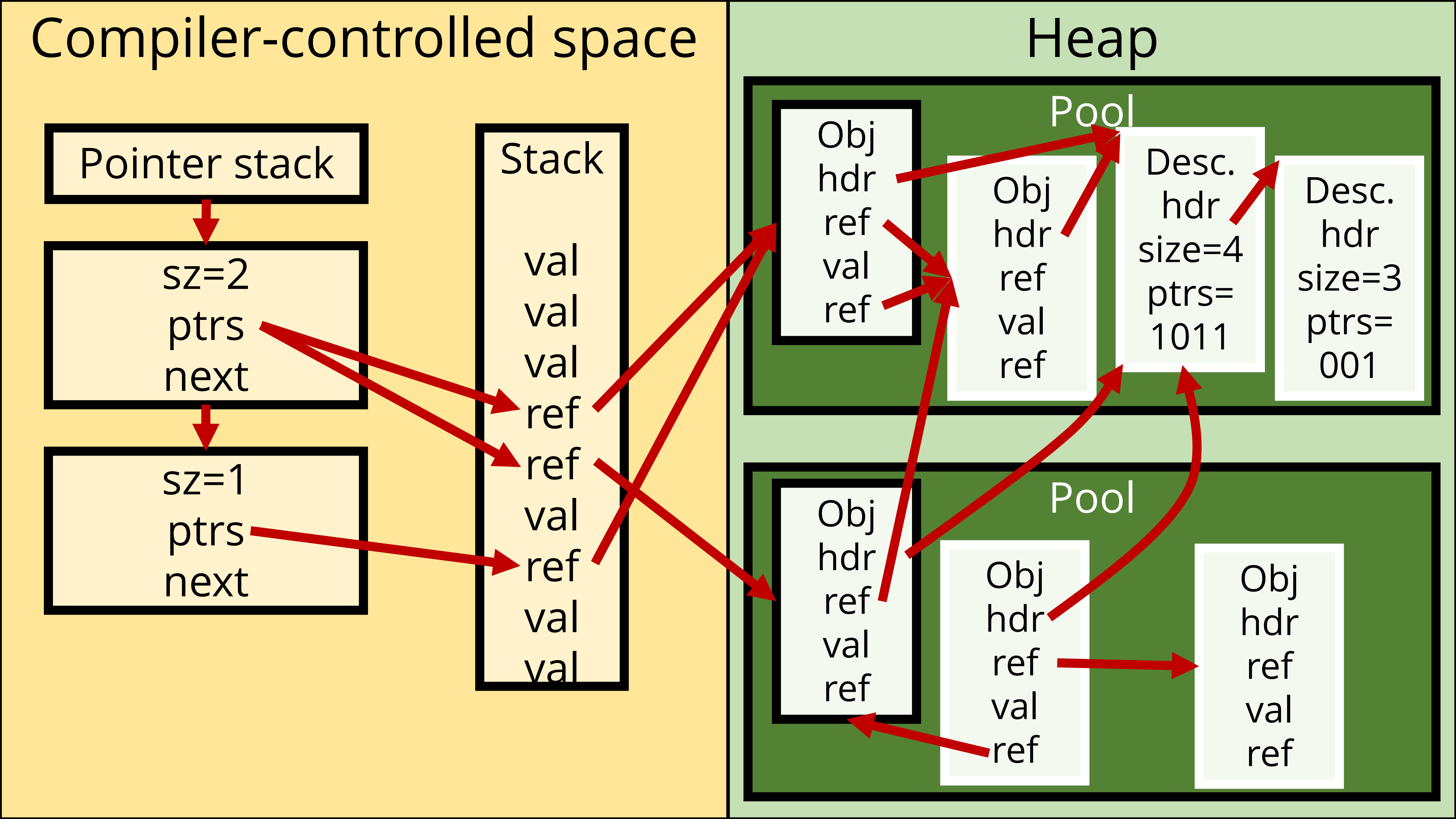
When to allocate pools

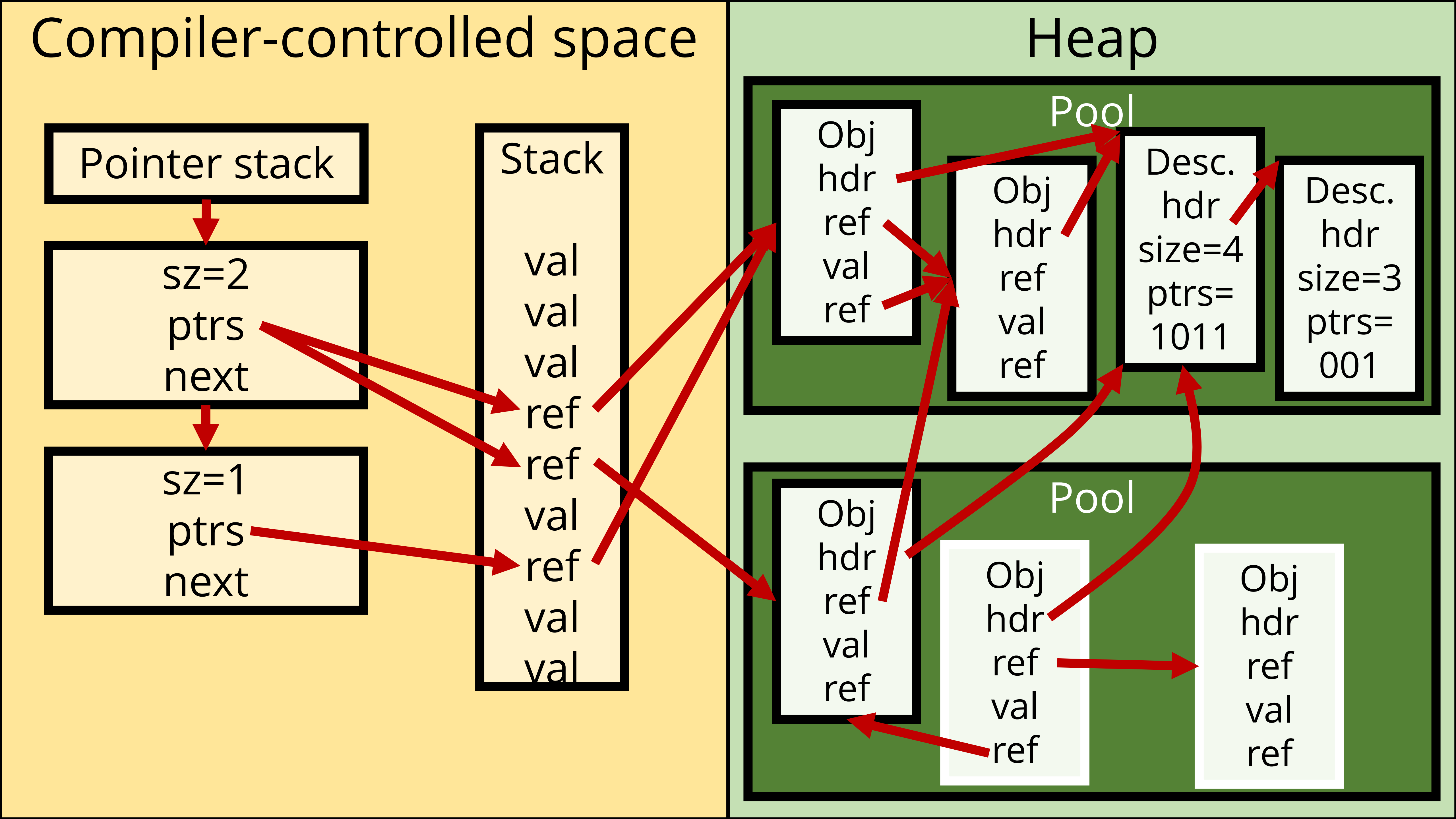
- Need double the space of mark&sweep
- Performance consideration:
  - Throughput
  - Latency
- More pools *always* better throughput

The Devil is in the Details

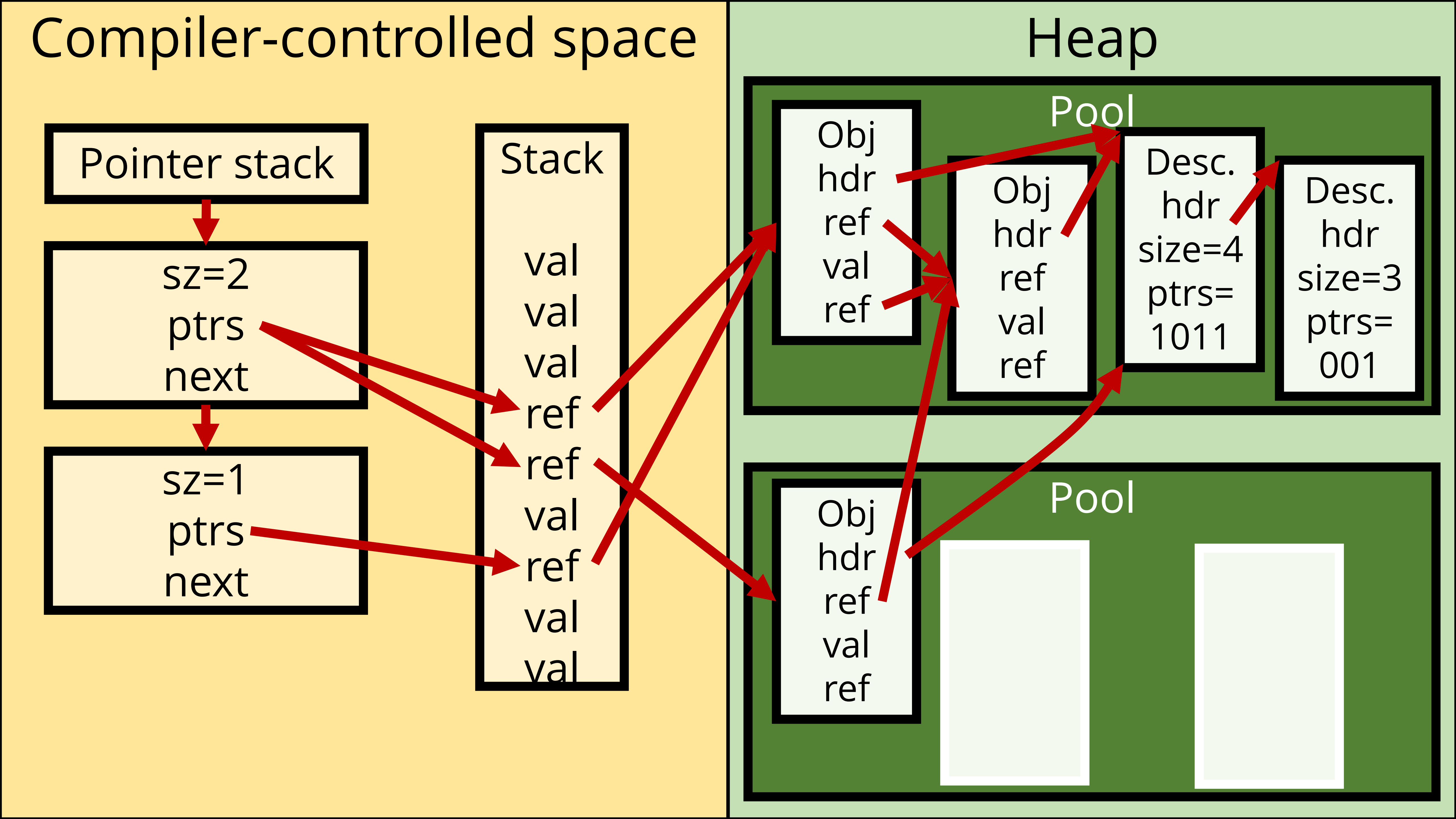


Compiler-controlled space
Heap
Pointer stack
sz=2
ptrs
next
sz=1
ptrs
next
Stack
val
val
val
ref
ref
val
ref
val
val
Pool
Obj hdr ref val ref
Obj hdr ref val ref
Desc. hdr size=4 ptrs= 1011
Desc. hdr size=3 ptrs= 001
Pool
Obj hdr ref val ref
Obj hdr ref val ref
Obj hdr ref val ref


Compiler-controlled space
Heap
Pointer stack
sz=2
ptrs
next
sz=1
ptrs
next
Stack
val
val
val
ref
ref
val
ref
val
val
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Desc.
hdr
size=4
ptrs=
1011
Desc.
hdr
size=3
ptrs=
001
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref



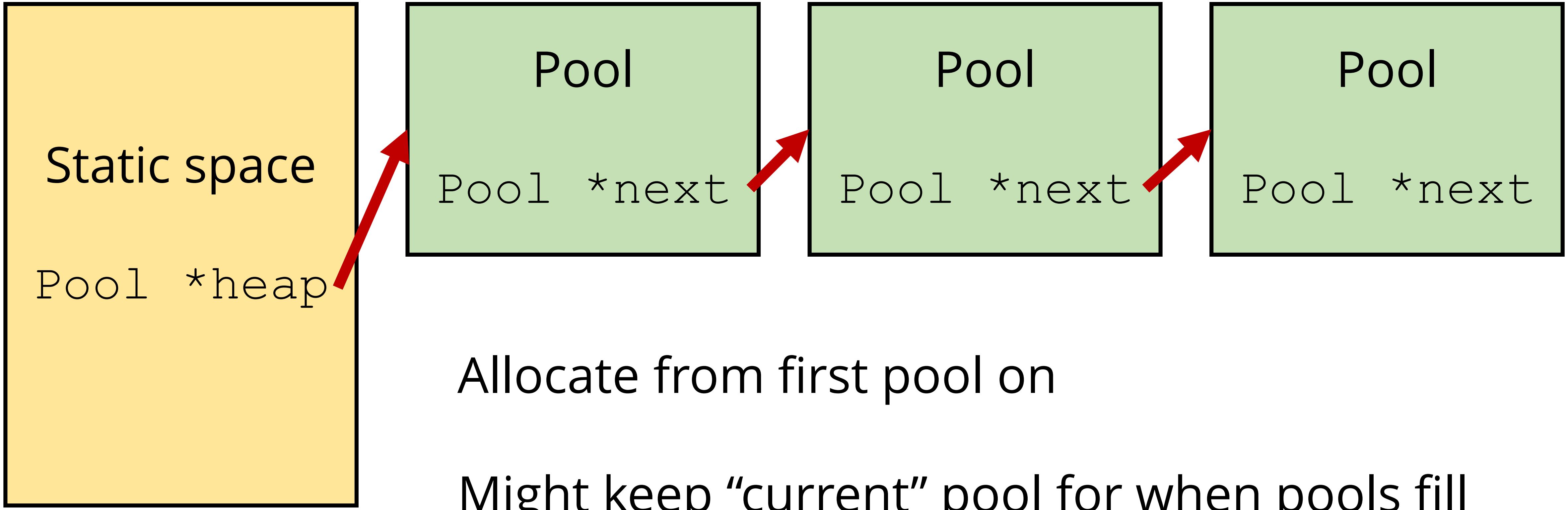
Compiler-controlled space
Heap
Pointer stack
sz=2
ptrs
next
sz=1
ptrs
next
Stack
val
val
val
ref
ref
val
ref
val
val
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Desc.
hdr
size=4
ptrs=
1011
Desc.
hdr
size=3
ptrs=
001
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref


Compiler-controlled space
Heap
Pointer stack
sz=2
ptrs
next
sz=1
ptrs
next
Stack
val
val
val
ref
ref
val
ref
val
val
Pool
Obj
hdr
ref
val
ref
Obj
hdr
ref
val
ref
Desc.
hdr
size=4
ptrs=
1011
Desc.
hdr
size=3
ptrs=
001
Pool
Obj
hdr
ref
val
ref

Heap

- OS is dumb: Gives you some pages
- GC maintains pools
- “Heap” is all pools
- GC must keep track

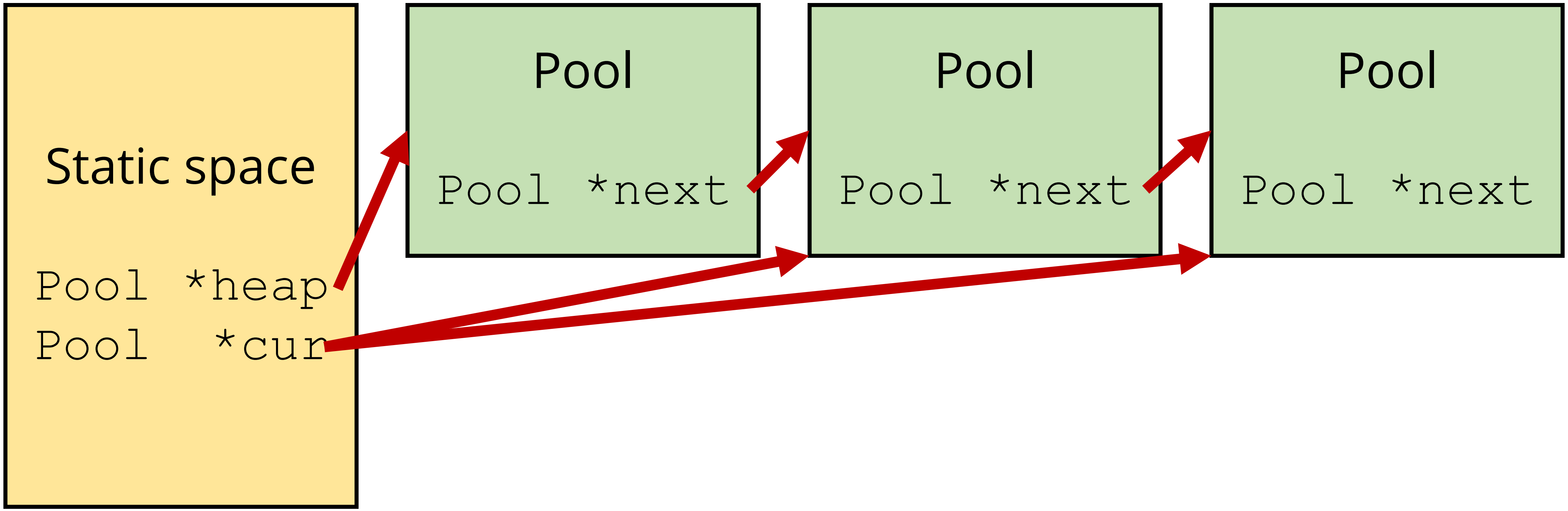
Keeping pools



Allocate from first pool on

Might keep “current” pool for when pools fill

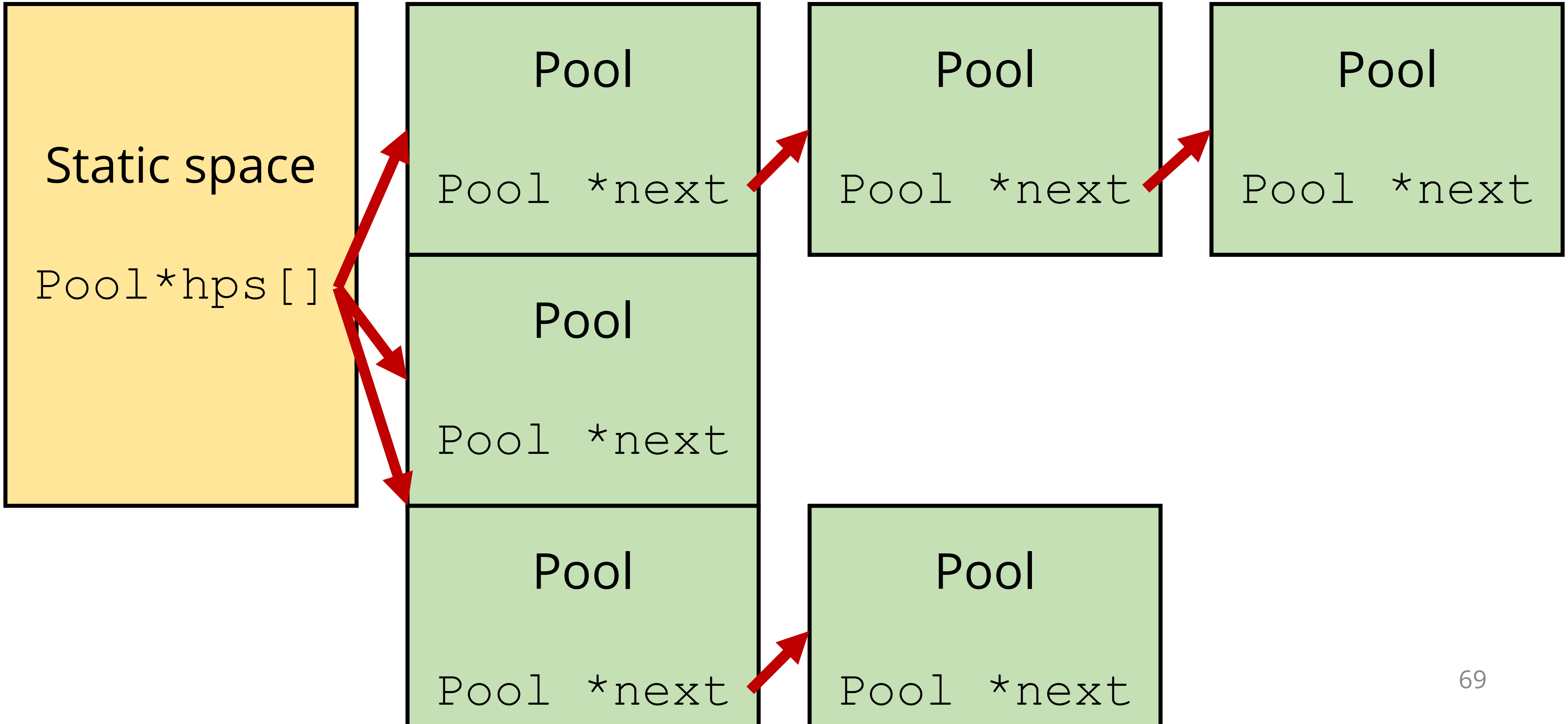
Keeping pools



Segregated blocks

- With segregated blocks, pools have fixed-sized objects
- No reason to mingle dissimilar pools

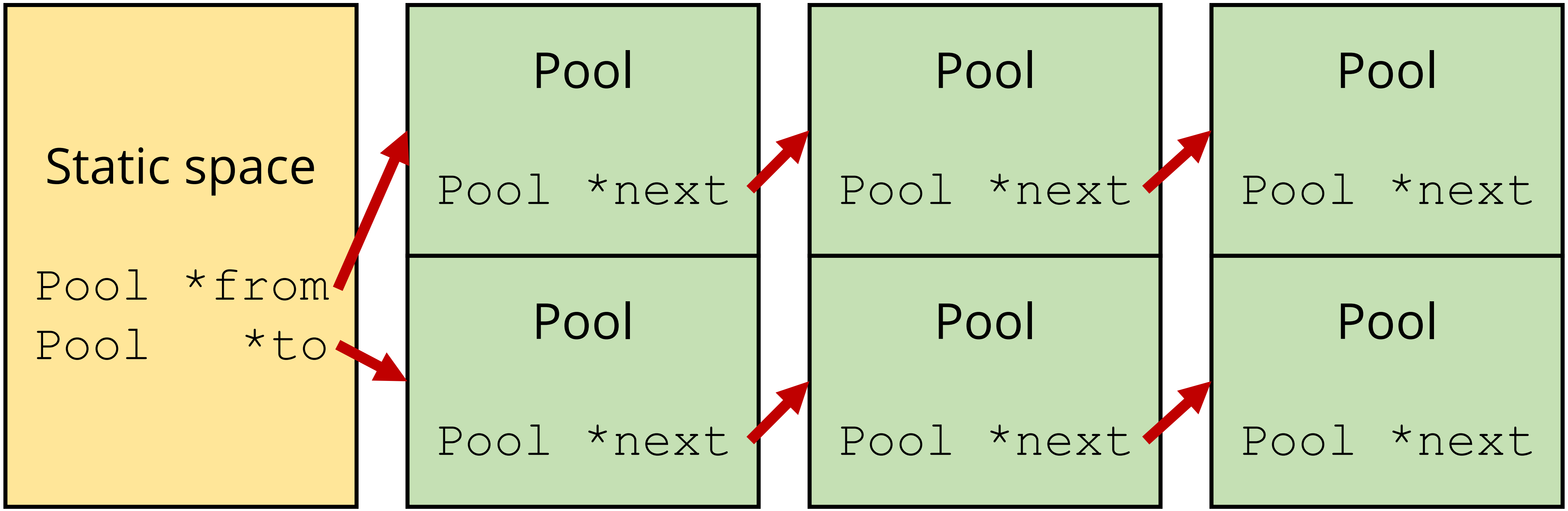
Pools w/ segregated blocks



Pools w/ semispace copying

- Need fromspace and tospace
- Pool “spaces” are non-intersecting, equal size

Keeping pools



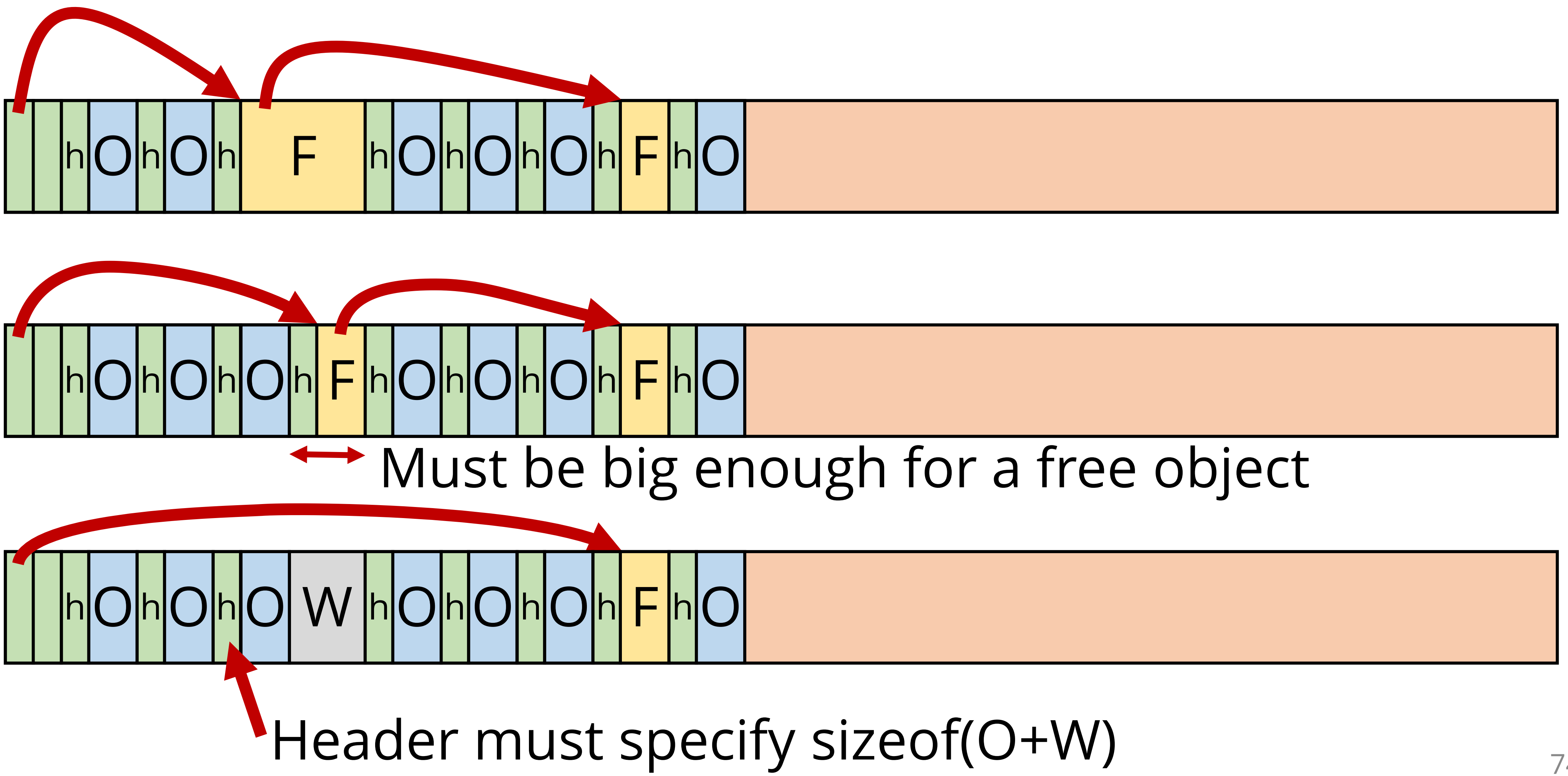
Free-lists

- Global or per-pool?
- Global: Thread contention (not an issue for now)
- Per-pool:
  - Go through every pool every allocation? Or
  - Accept lost space after large allocations?

Free-list order

- Mark-and-sweep makes address-ordered free-list
- Pools aren’t necessarily address-ordered
- Should they be?

Splitting vs overallocation



Overallocating

- Can be avoided:
  - Bitmapped-fits
  - Allocation granule ≥ size of free object
  - Non-free-list allocation
- Let’s think about headers…

Overallocating

```
struct ObjectHeader {
  struct GCTypeInfo *typeInfo;
};

struct GCTypeInfo {
  size_t size;
  unsigned long pointerMap;
};
```

Cannot change per object (typeInfo)

Does not represent overallocated size (size)

Objects

- GC only knows:
  - Size
  - Location of references
- Both are in descriptor, also a GC object!
- Must make sure to keep object descriptors alive

Objects

- Mutator is assumed correct
- References always point to heap, pointer stack is correct, etc
- Mutator wrong → crash

Sizes and optimal configuration

- Several important metrics
  - L = size of live objs
  - H = size of heap
  - D = size of dead
- L mostly static
- Most objects die young
- H=L*3 typical, H=L*5 often ideal

So wasteful!

- If (H >>> L), I’m wasting space!
- Problem of fairness
  - Can solve with IPC
- Memory is cheap
- Time is expensive

Tradeoffs

- You choose H, but not L
- H >>> L:
  - Less frequent GC
  - Mark-and-sweep: More time spent in GC (latency)
- H ≈ L:
  - Very frequent GC