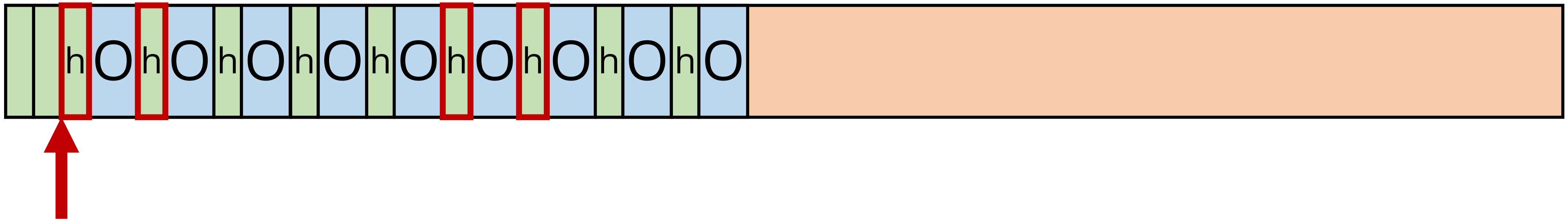
Lazy sweep

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

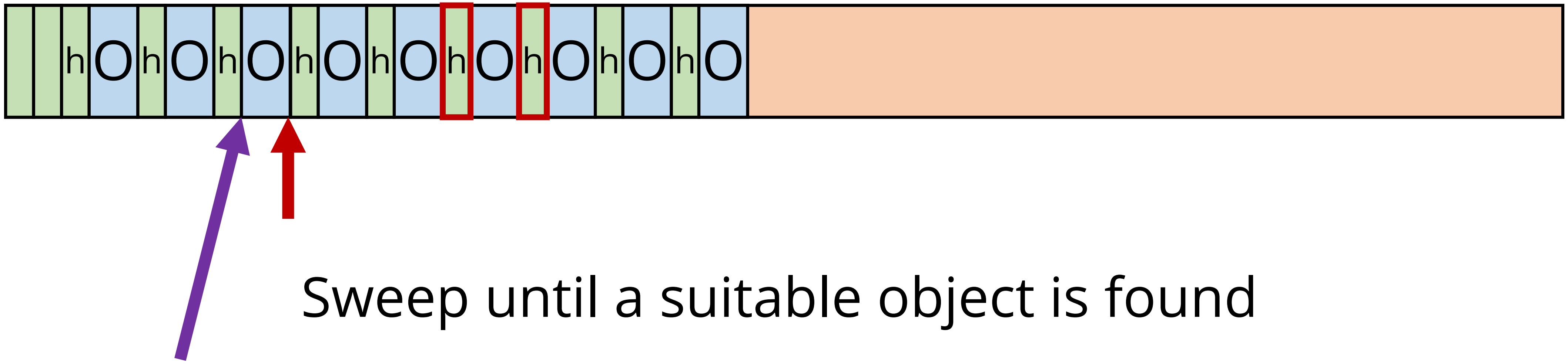
Lazy sweep



Sweep pointer maintained per pool

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

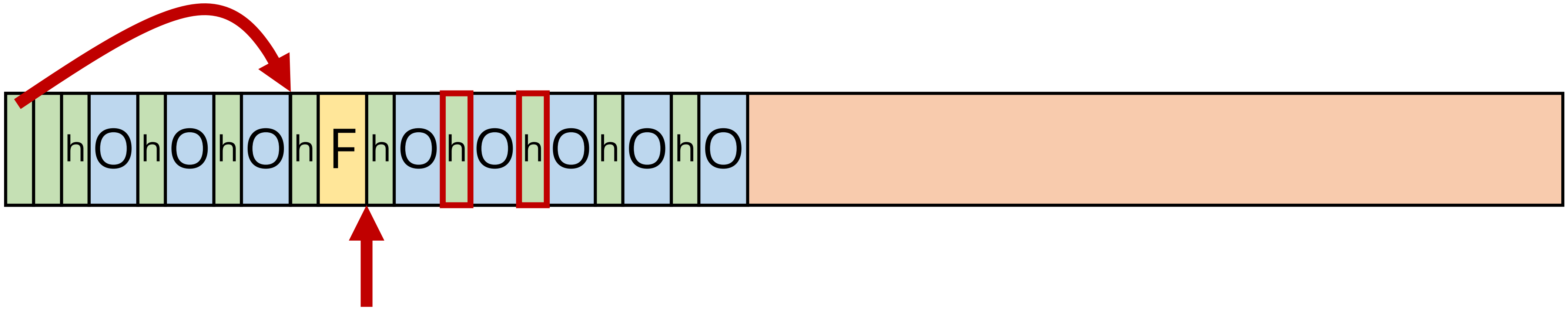
Lazy sweep



Sweep until a suitable object is found

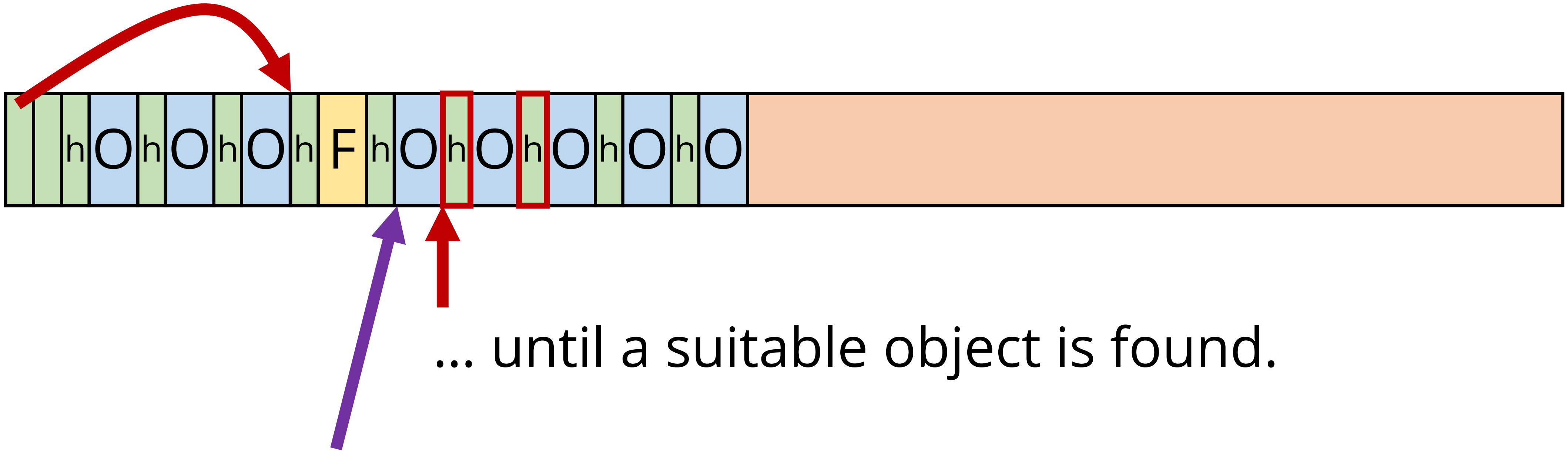
This object is returned to mutator

Lazy sweep



Unsuitable objects added to free-list during allocation

Lazy sweep



... 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

Copying GC

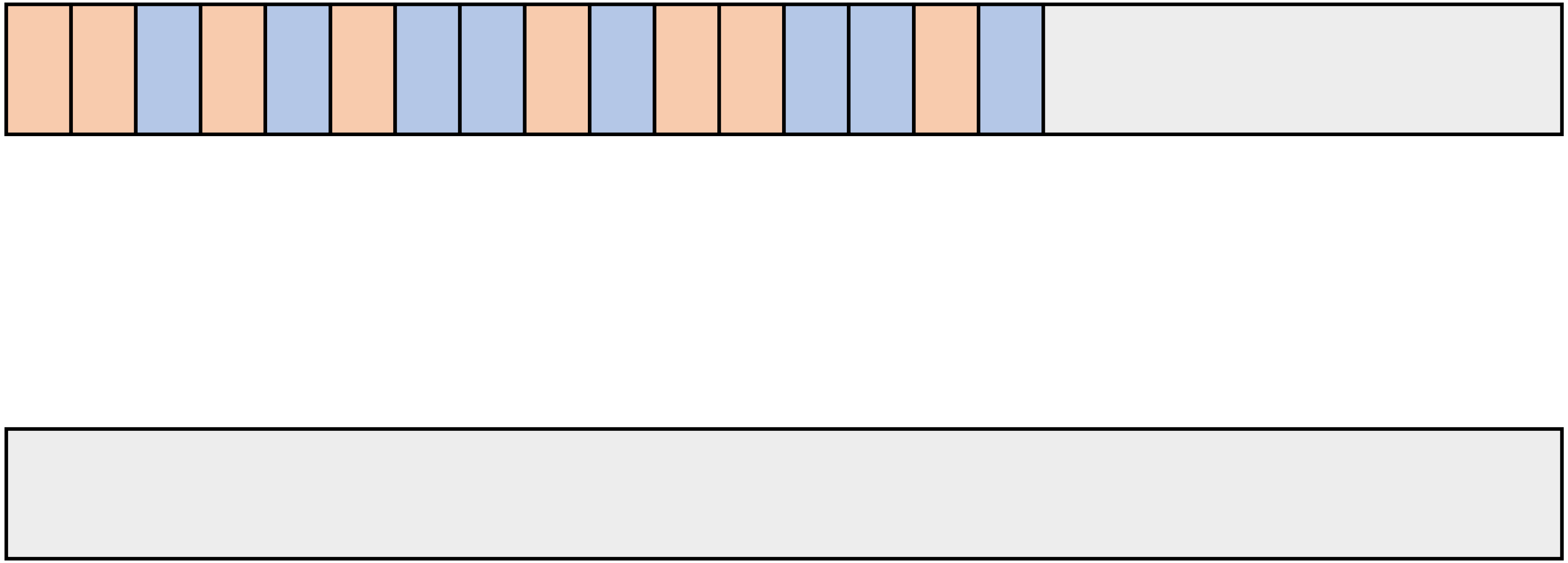
Review

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

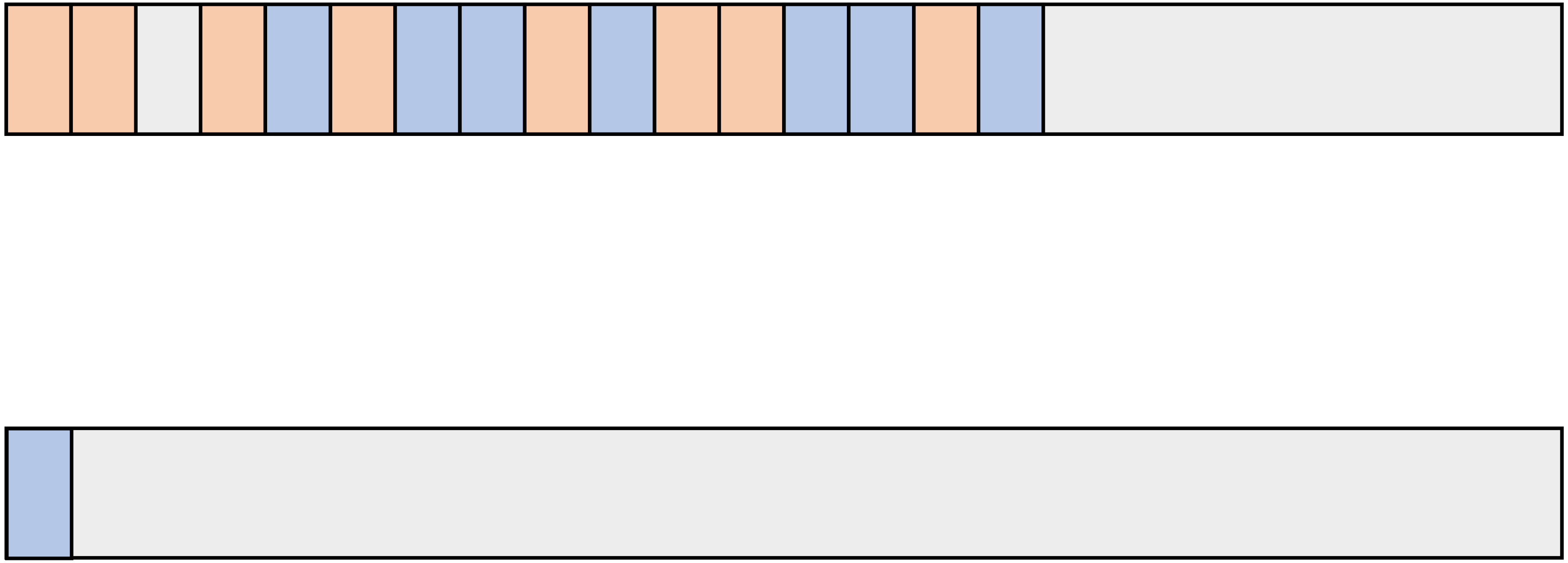
Mark and sweep

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

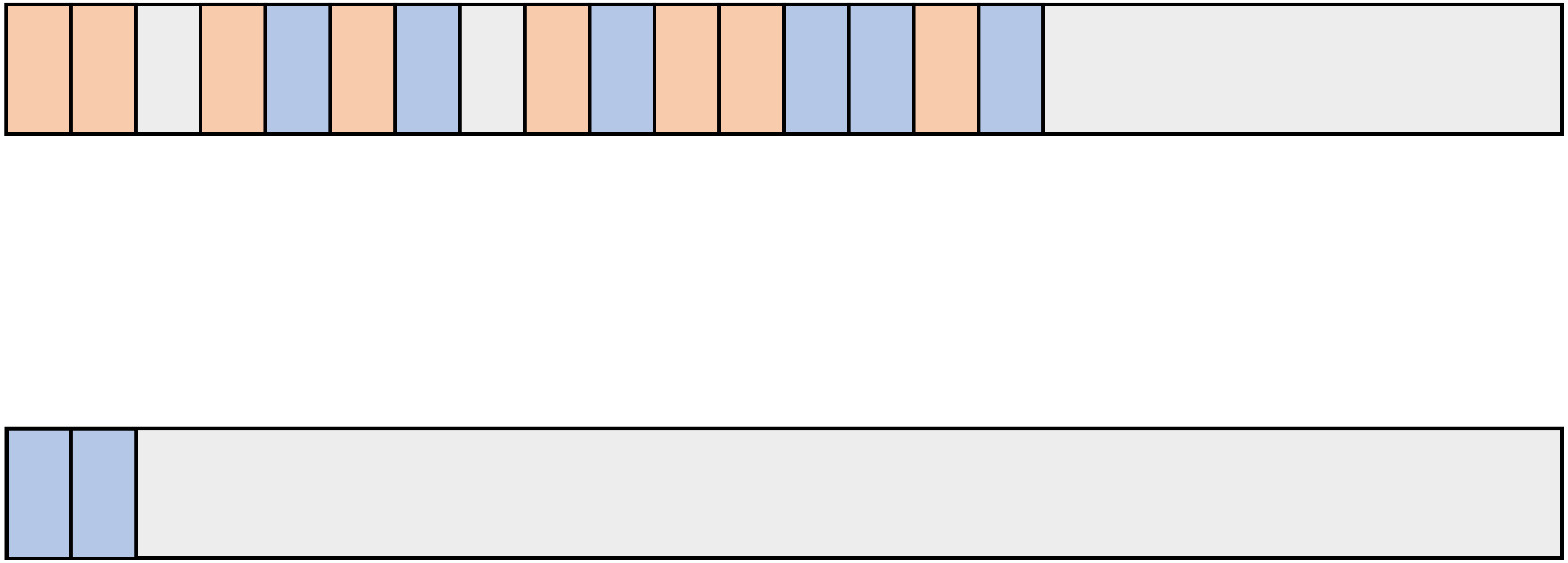
Semispace copying



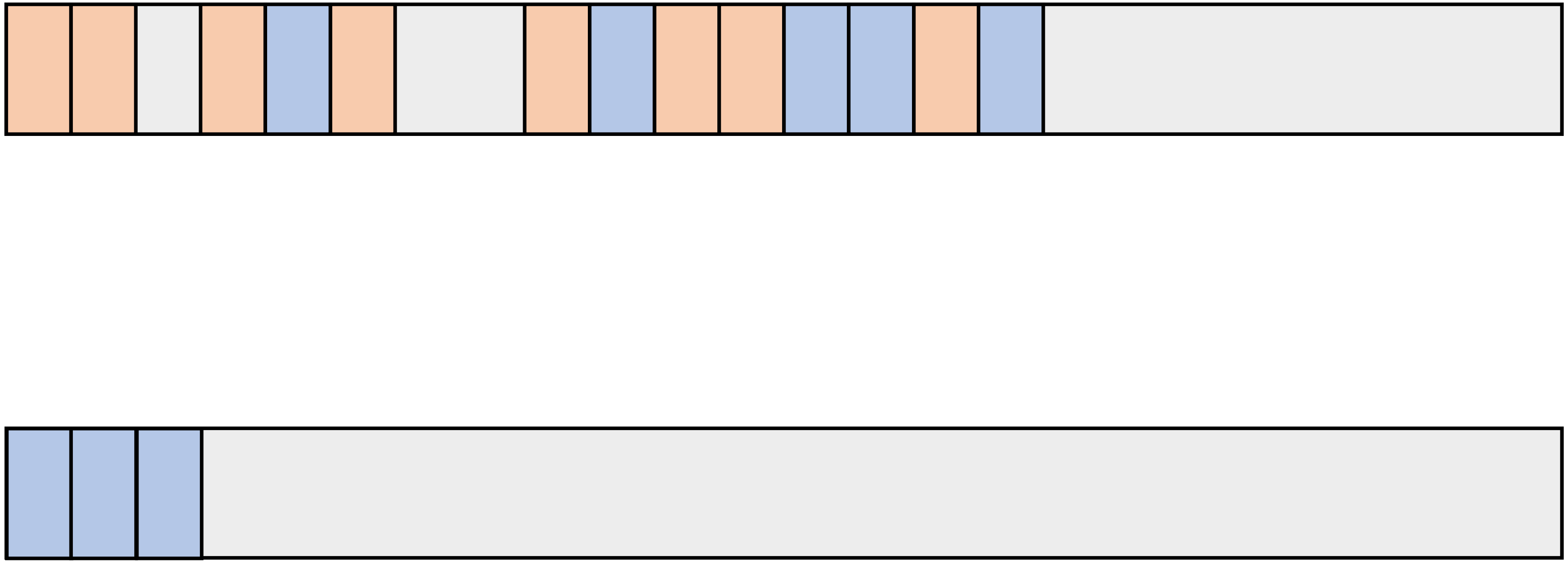
Semispace copying



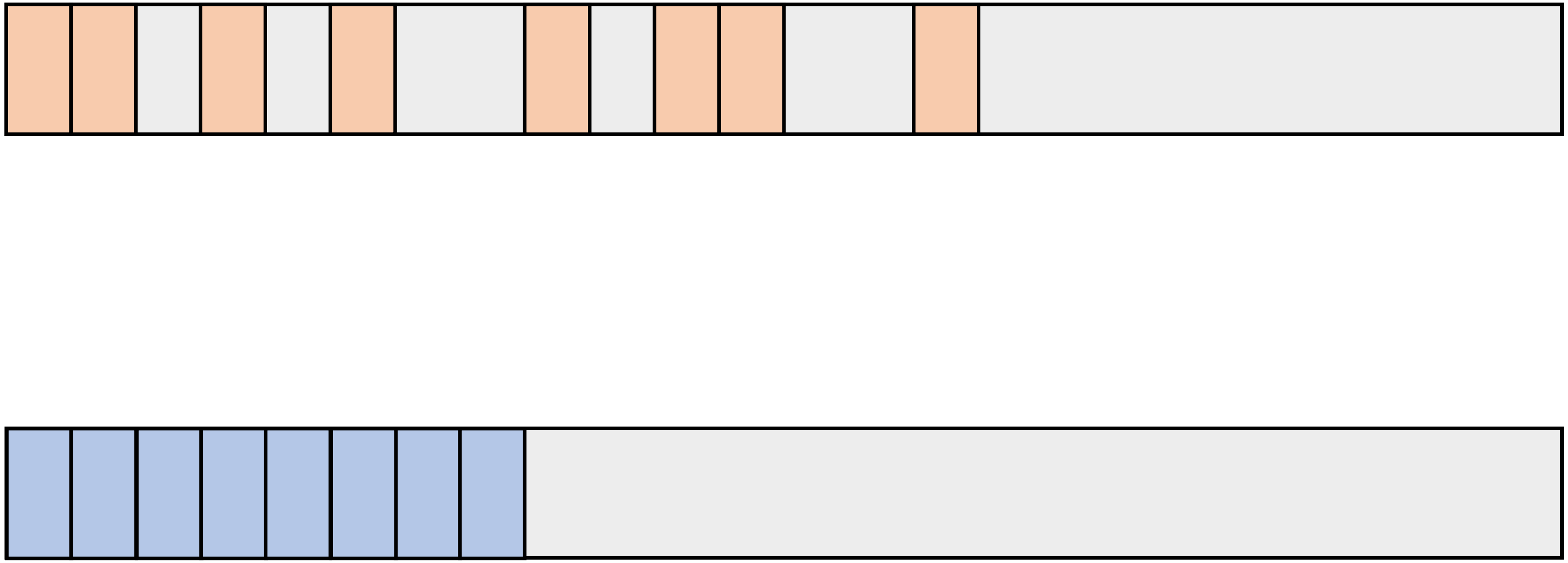
Semispace copying



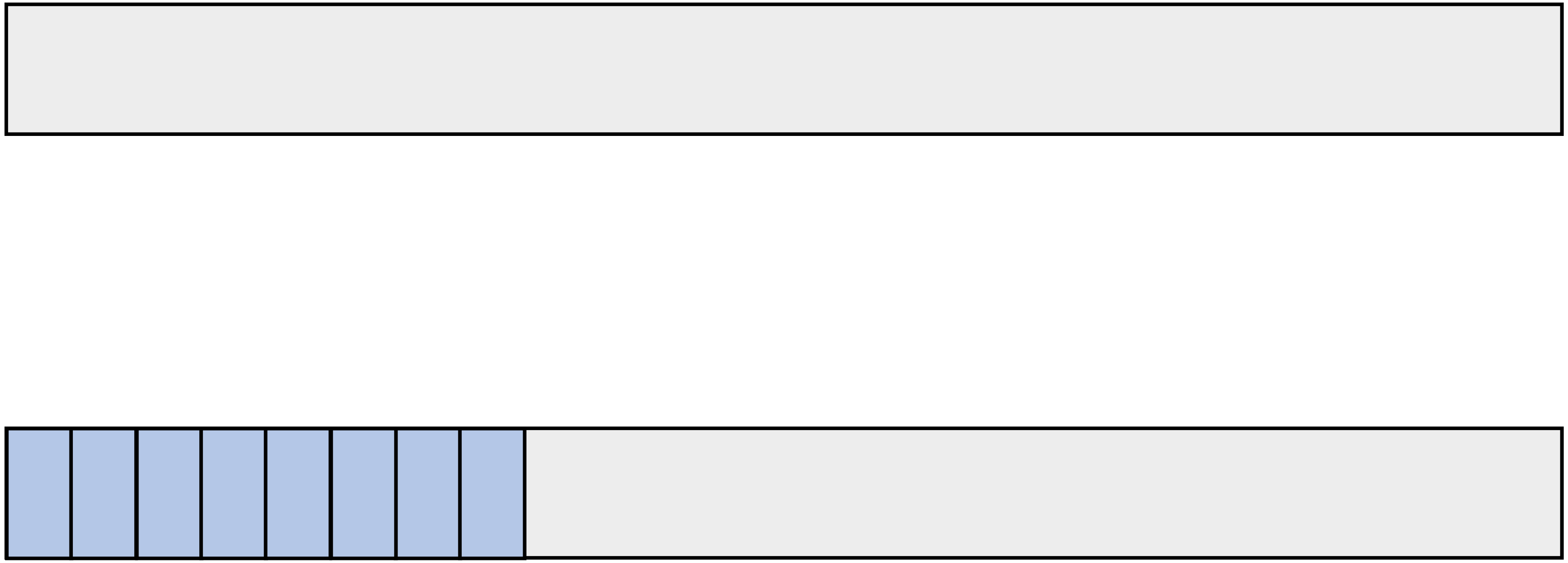
Semispace copying



Semispace copying



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

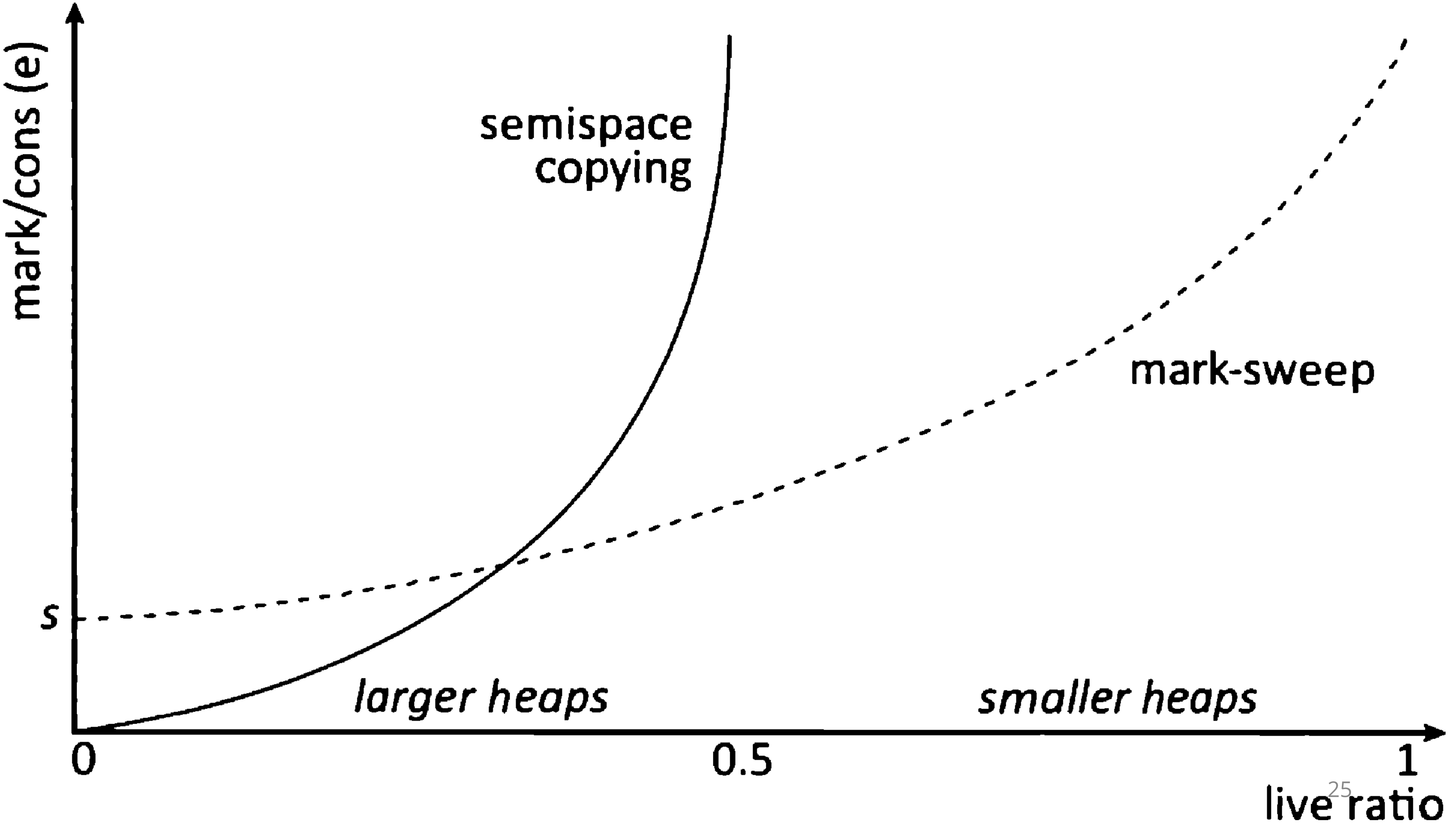
- Can we predict the best way to arrange objects?
- No. NP-complete even with access pattern oracle.

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