Moving GC

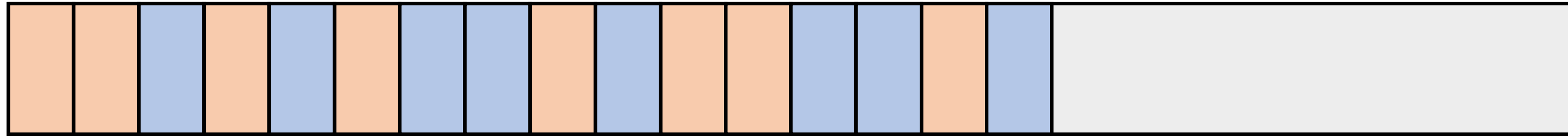
Review

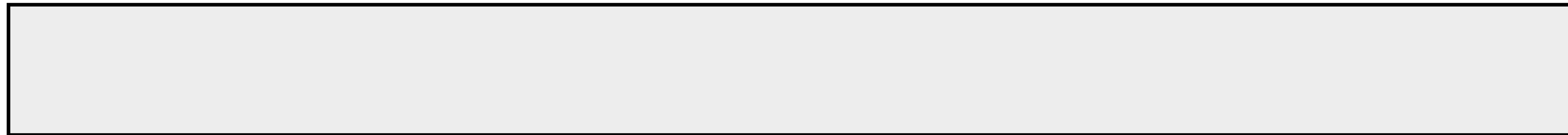
- Allocator owns pools
- Compiler controls roots
- Compiler informs allocator of roots, object types
- Trace references to find living objects

Mark and sweep

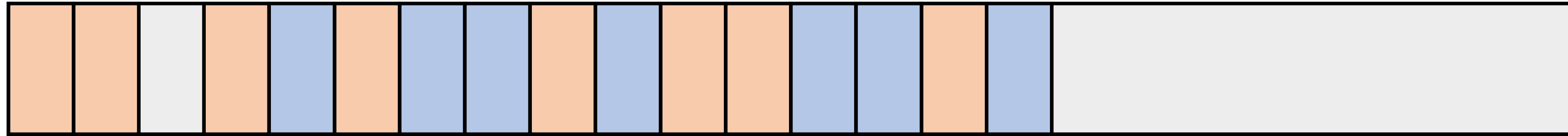
- Very natural map to reachability
- Two passes
- Prone to fragmentation

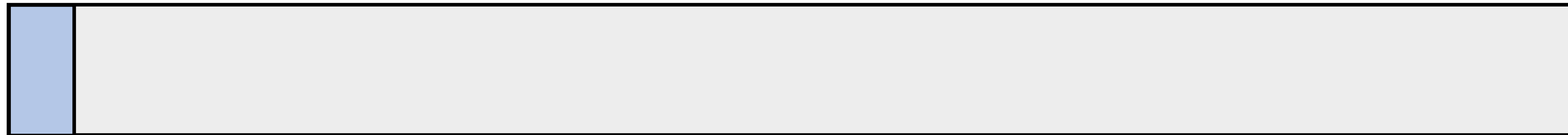
Semispace copying



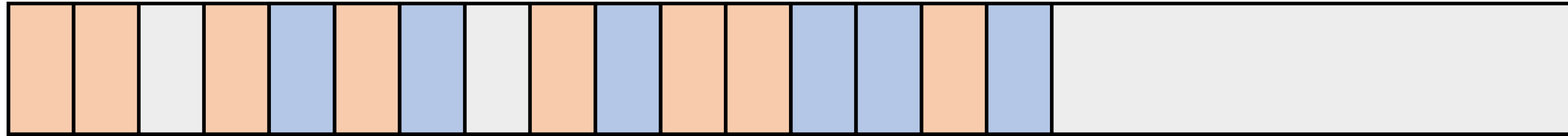


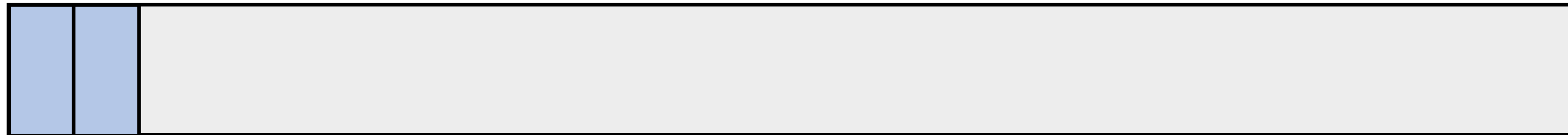
Semispace copying



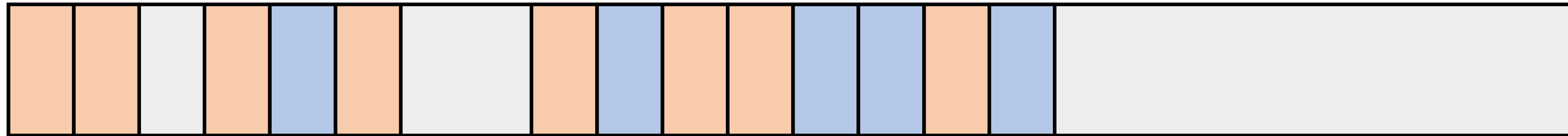


Semispace copying

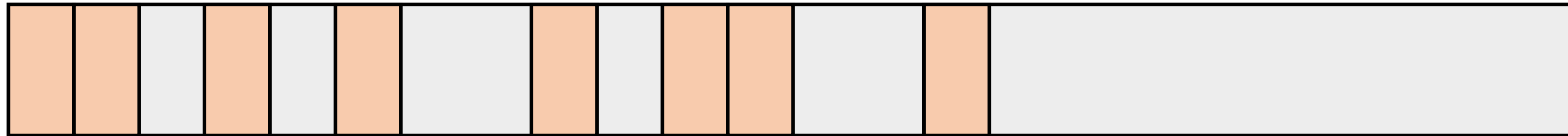




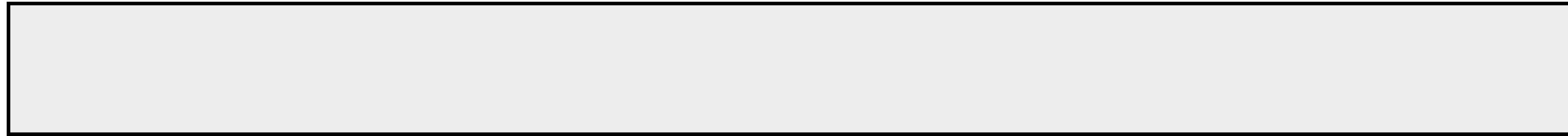
Semispace copying

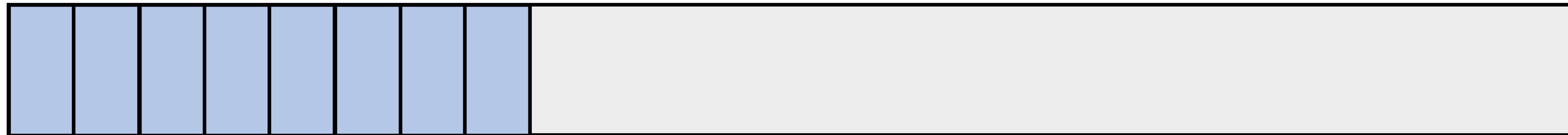


Semispace copying



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.

Forwarding

- Naïve:

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

Forwarding

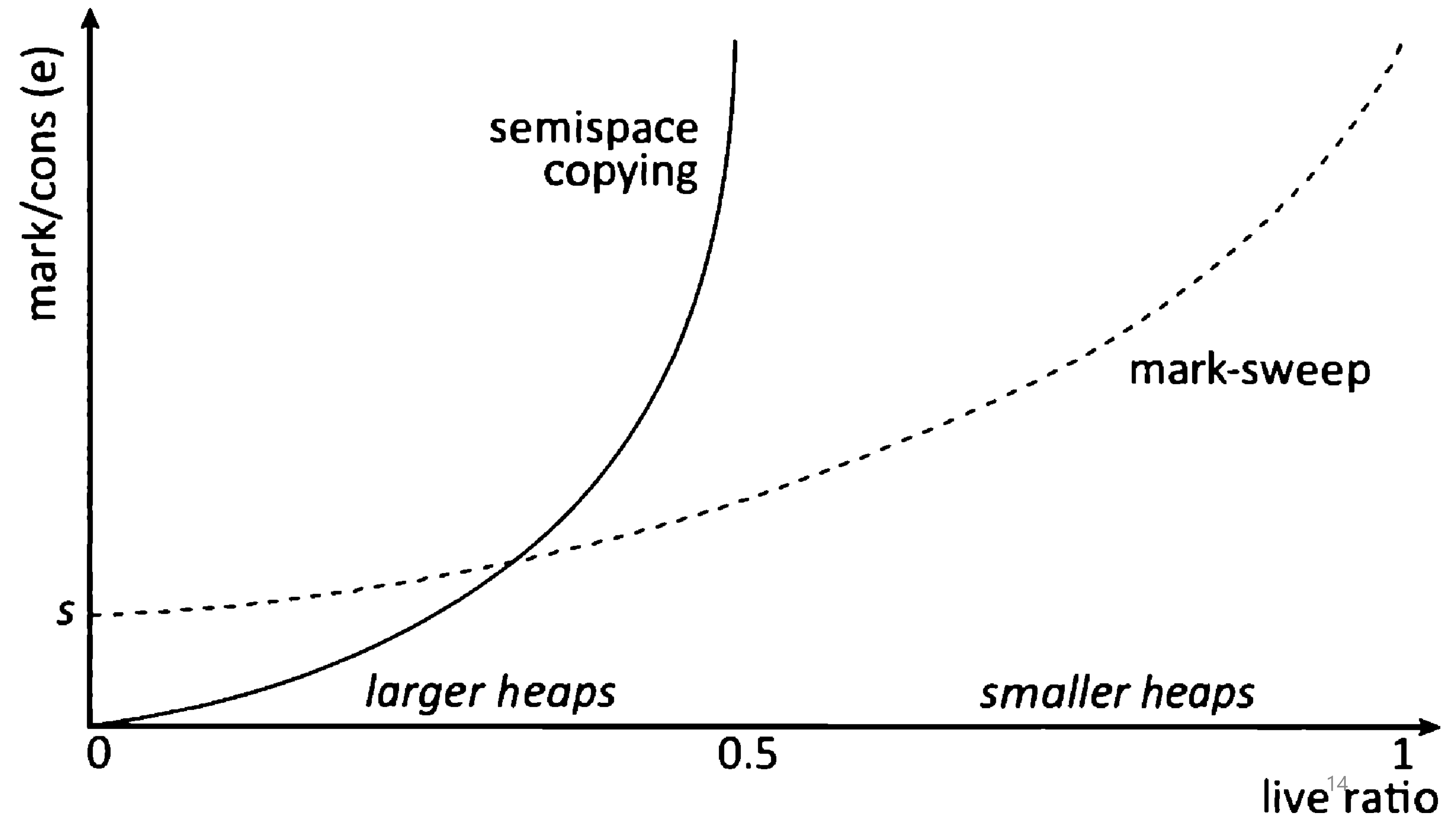
- Still have those extra bits!
- Once an object is forwarded, no longer need type info
- Careful: Pointer wrong in two ways

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

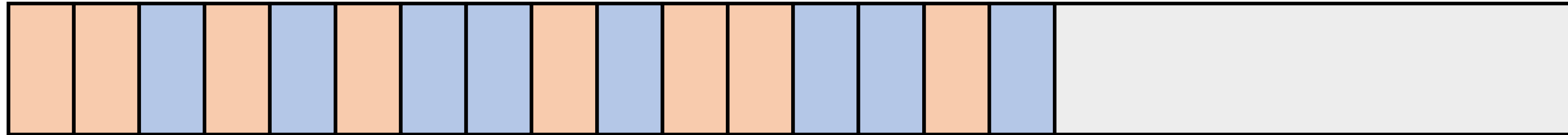
- Need double the space of mark&sweep
- Performance consideration:
  - Throughput
  - Latency
- More pools *always* better throughput

Mark and compact

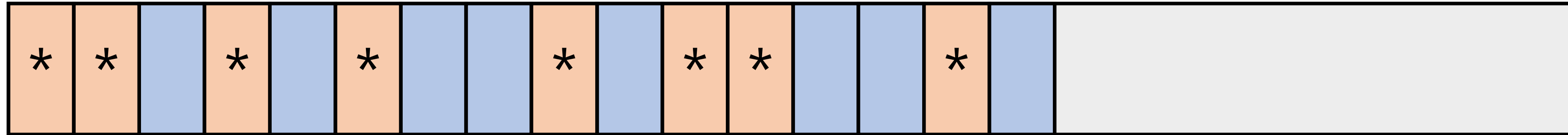
Review

- Roots → objects → reachable objects → discard unreachable
- Mark and sweep: Mark reachable objects, sweep unreachable
- Semispace copying: Copy reachable objects to second heap, ignore unreachable
- Reference counting: Continuously monitor reachability

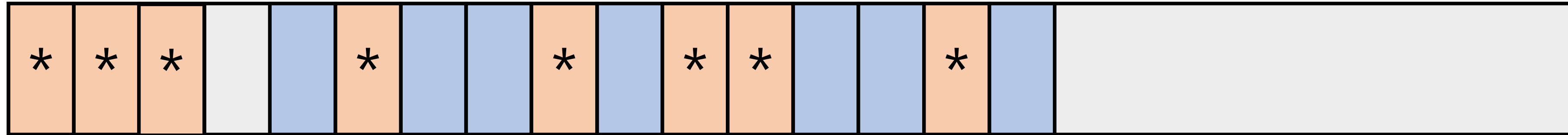
Mark and compact



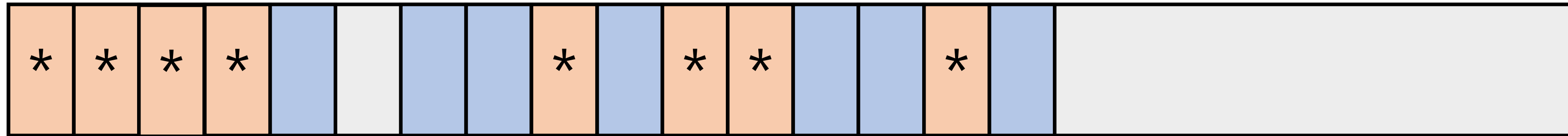
Mark and compact



Mark and compact



Mark and compact



Mark and compact

Mark and compact

Thoughts

- Like copying:
  - Simple allocation
  - No fragmentation
  - Objects move
- Like mark-and-sweep:
  - No wasted space
  - Must “sweep” full heap (to move)

Recall copying collection:

- During “mark”, knew forwarding addresses immediately
- Always possible to redirect pointers while marking

```
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
```

Quandary

- In copying collection, update pointers immediately
- In compacting, we must mark before compacting
- Don't know where to forward pointers until after mark phase: cannot mark-and-forward

Solution

- Three-pass compacting sweep (yuck!)
  - 1: Imaginary compaction to determine forwarding addresses
  - 2: Update references to forwarding addresses
  - 3: Compact

Fewer passes?

- We'll see how later, but…
- Compute locations + update references?
  - Pointer to later object in heap won't be updated
- Update references + compact?
  - Forwarding pointer typically in object header
  - Object header will be overwritten by a compacting object

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size

updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size

updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size

updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

Lisp2 algorithm

```
compact():
  computeLocations()
  updateReferences()
  relocate()

computeLocations(): (described per pool)
  scan := start
  free := start
  while scan < end
    if marked(scan):
      scan->header.fwd = free
      free += scan->header.size
    scan += scan->header.size
```

```
updateReferences():
  (update root references)
  foreach obj in heap:
    if (!marked(obj)) continue
    foreach loc in obj->header.typeInfo->ptrs:
      *(obj+loc) = (*(obj+loc))->header.fwd

relocate(): (described per pool)
  scan := start
  while scan < end:
    if isMarked(scan):
      memcpy(scan->header.fwd, scan, scan->header.size)
      unmark(scan->header.fwd)
    scan += scan->header.size
```

You will pay dearly

- Running time O(H)
  - Technically not worse than mark-and-sweep!
- Constant factor very high
  - One mark phase + three sweep phases
- Ever worth it? Maybe! For infrequent GC

Doing better

- Fundamental problem: Forwarding pointers stored with objects, thus overwritten
- So store them separately!
- … but where?

Side tables

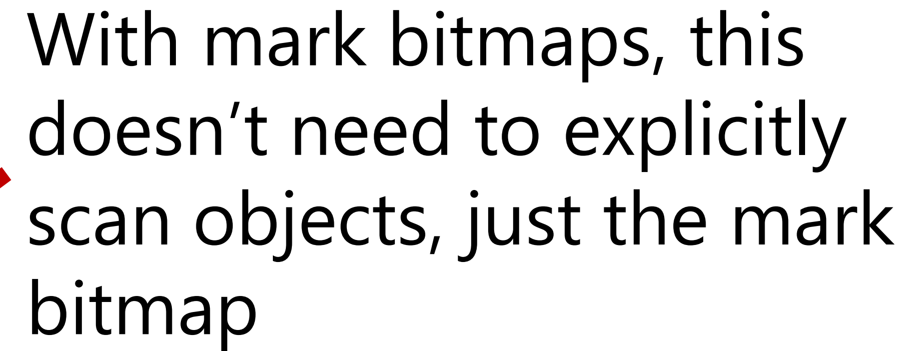
- Remember mark bitmaps?
- Store side-table for forwarding addresses
- Expensive to store forwarding per object, so divide pools into “blocks”
- Every object in a block moved by same amount
  - This means some objects unnecessarily stay alive

```
compact():
  computeLocations()
  relocate()

computeLocations():
  lastBlock := -1
  blockMarked := false
  free := start
  foreach obj in pool:
    if blockOf(obj) != lastBlock and blockMarked:
      forwards[lastBlock] = free
      free += WORDS_IN_BLOCK
      blockMarked := false
    if marked(obj):
      blockMarked := true

newAddress(old):
  block := blockOf(old)
  return forwards[block] + old%WORDS_IN_BLOCK

relocate():
  (update root references using newAddress)
  (use forwards table to move blocks)
  (update pointers in living objects using newAddress)
```



Problem?

- Heap parsability is painful:
  - Dead objects kept alive, but their descriptors may still be dead
  - Objects needn’t start at a block, so half-objects copied
- Still two passes (one of bitmaps only)

More advanced techniques

- Threading:
  - Remove forwarding pointers from object headers by reversing pointers during collection
- Break tables:
  - Keep forwarding info in free space between chunks of marked objects

Object lifetime

- Live objects at beginning of heap not copied
- Most objects die young
- Mark-compact will move long-lived objects to beginning of heap

Pseudo-generational

- Low heap is mostly immortal
- Start compaction part way through the heap: Only compact young objects
- Only compact full heap occasionally
- Saves time moving ancients

When to GC

- Mark-compact is *slow*
- Bigger heap slower, but not proportional
- GC as infrequently as possible
- H >>> L crucial
    - But unlike semispace, we can *use* H

Worth it?

- Next week we will talk about generational garbage collection
- “Young” generation frequently collected
- “Old” generation infrequently collected
- Mark-and-compact is most seen there
- (Copying + mark-and-compact = Java’s GC!)

Partitioning the Heap

Review

- Roots → objects → reachable objects → discard unreachable
- Moving vs. non-moving
- Copying vs. compacting
- Pause vs. live

Tradeoffs

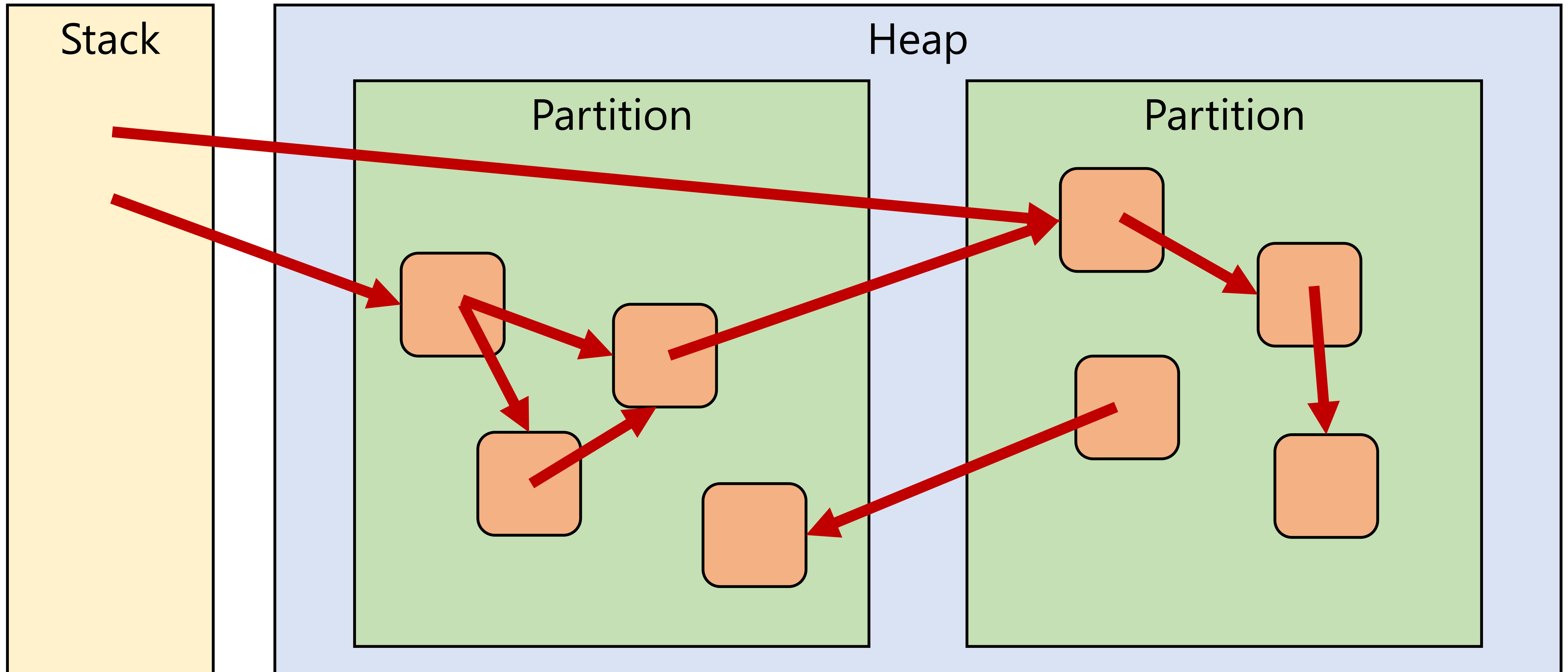
- Different strategies have different tradeoffs
- Mark-and-sweep: No moving, fragments
- Copying: Better locality, worst utilization
- Compacting: Good locality, very slow

Tradeoffs

- Best strategy to use depends on program
- Every program is different…

- Instead, use multiple strategies
- Choose “intelligently” per object

Partitioning



Paritioning

- Generally:
  - Objects in partitions share some property
  - Roots can point at any partition
  - Cross-partition references allowed
  - Partitions may hold dead cross-partition references
- May be false for some partitioning style

Why partition?

- Usually: Use different GC schemes
- Often: GC only some partition(s) to reduce GC pause time
- Sometimes: Different allocation schemes, fragmentation avoidance, etc.
    - Segregated blocks!

Elephant in the room

- #1 partitioning scheme is generational GC
- Generational = partition by age
- We’ll get there, but others first

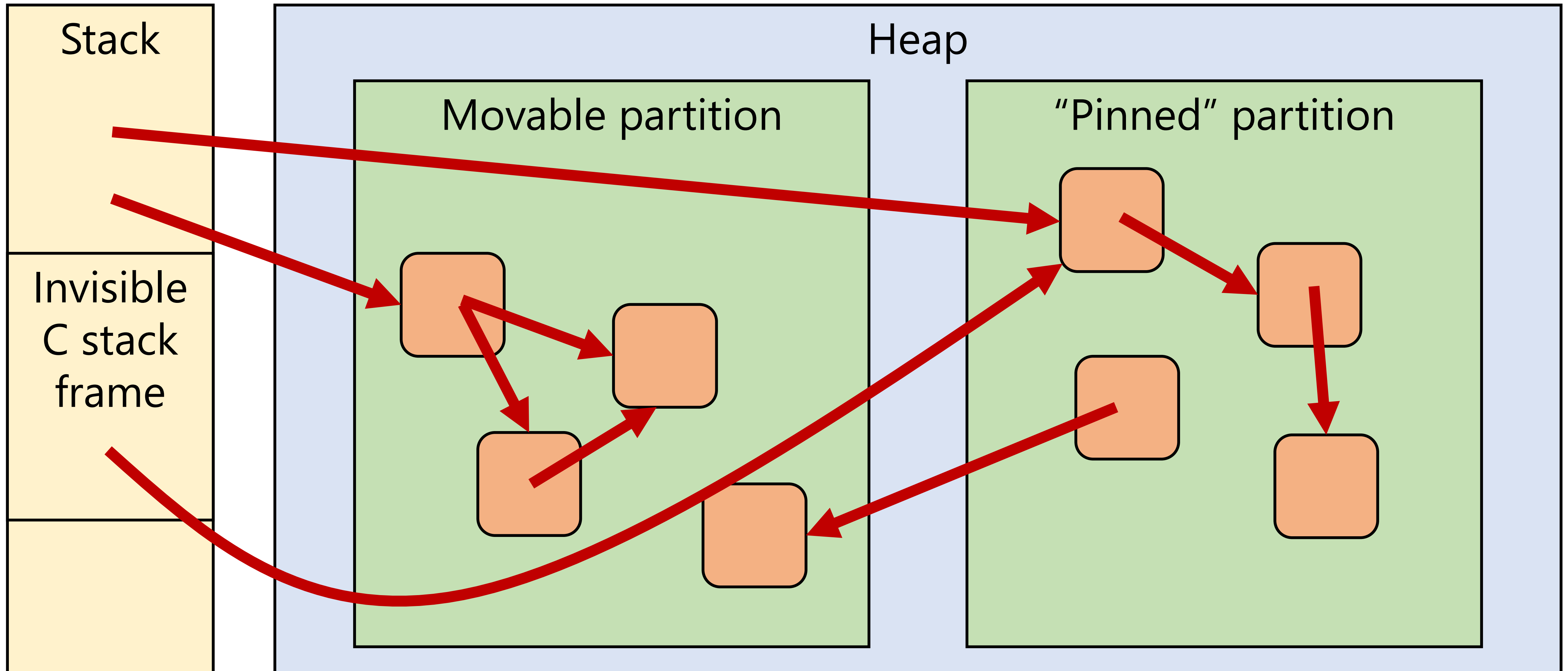
Mobility

- Moving objects reduces fragmentation
- Moving objects means references must change: Burden on compiler to handle references properly
- “Normal” C code not so nice

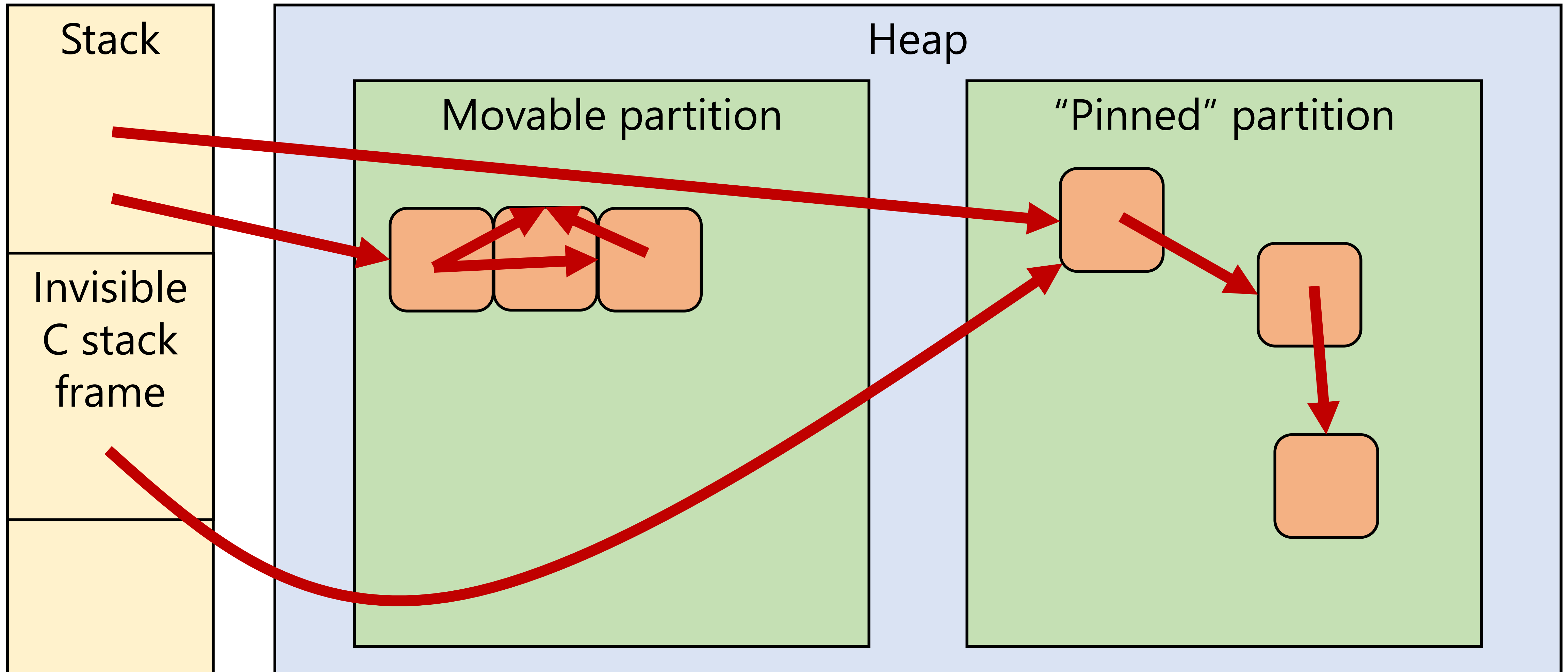
Mobility

- Non-moving: Must have *at least one* reference
- Java communicating with C:
  - Keep a reference in Java’s roots
  - Give reference to C (GC unaware)
  - When C is done, discard Java root ref

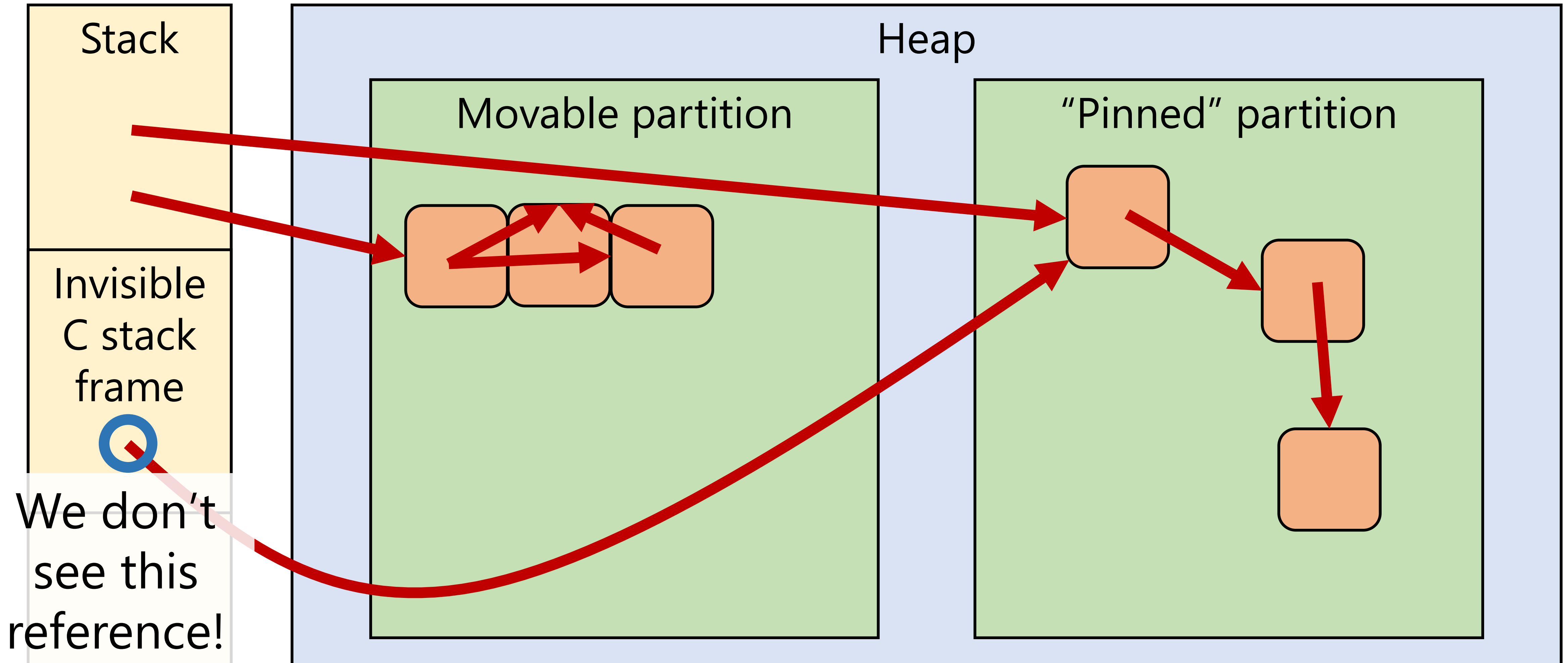
Partitioning for mobility



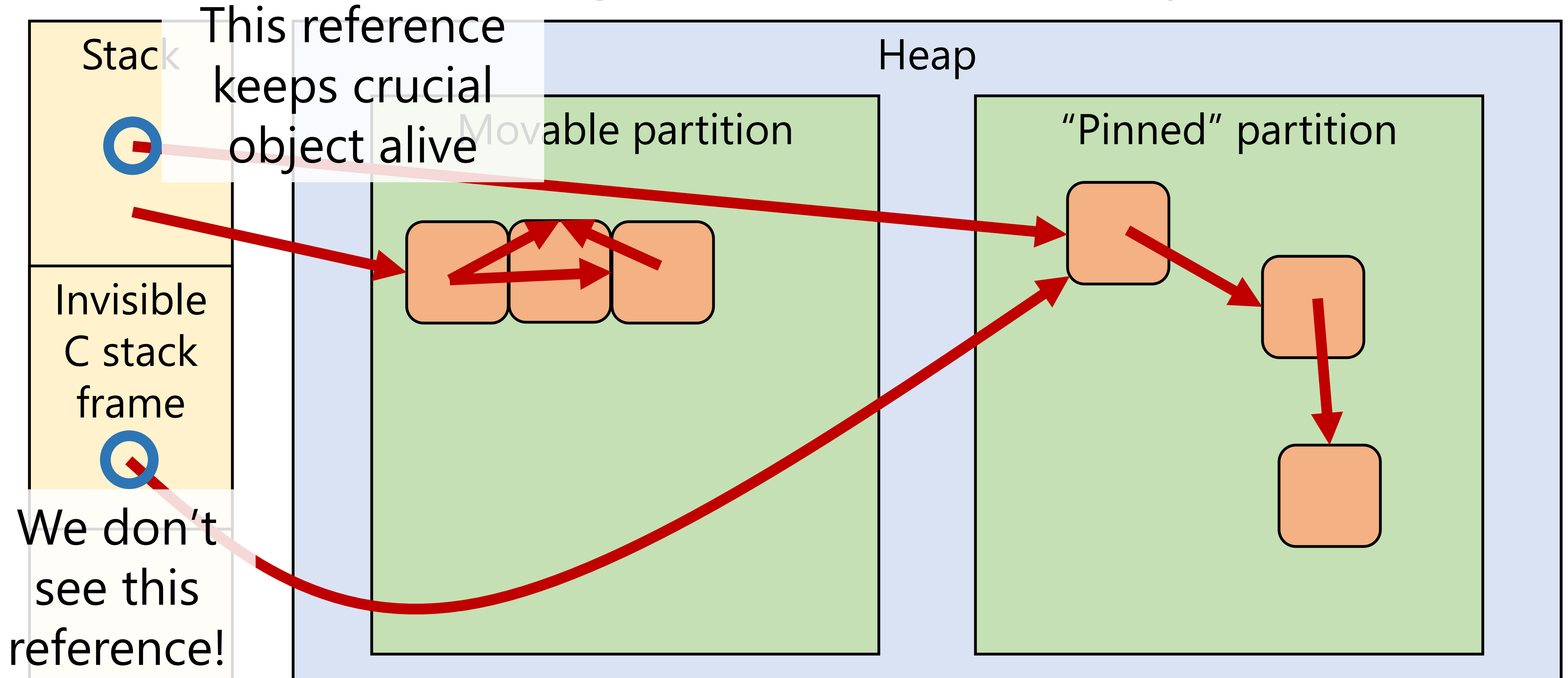
Partitioning for mobility



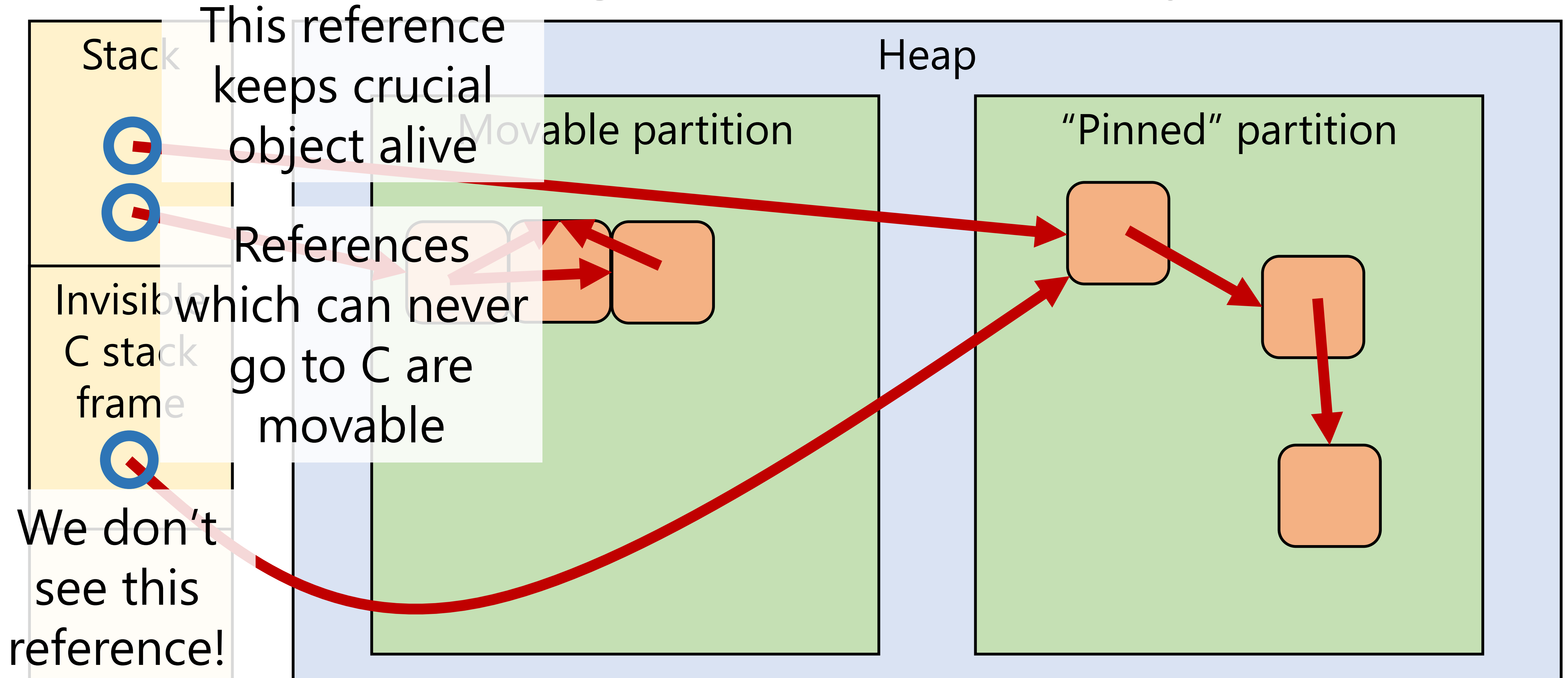
Partitioning for mobility



Partitioning for mobility



Partitioning for mobility



Mobility necessities

- Must know all objects which *might* be immobile (e.g. only certain types go to C)
- Assure visible root reference stays alive
- Immobile heap mark-and-sweep
- Must GC whole heap, not one partition

GC'ing with partitions

- Every GC strategy has a mark-like phase
  - Collectively these are called "tracing"
- This phase broadly similar in each GC
- When scanning object, determine which kind of tracing based on which partition

Partitioning algorithm sketch

```
trace():
  worklist := new Queue()
  (add roots to worklist)
  while loc := worklist.pop():
    obj := *loc
    if obj is in M&S or compacting partition:
      marked := mark(obj)
    else if obj is in copying partition:
      obj, marked := copy(obj)
    if !marked:
      (add obj's references to worklist)

sweep():
  mandsSweep()
  compactingSweeps()
```

Distinguishing partitions

- Partitions are separate sets of pools
- Pool remembers which partition it's in (typically pool header)

Distinguishing partitions

- Partitions are separate sets of pools
- Pool remembers which partition it's in (typically pool header)

"Pool mask": 0xFFFF0000

```
(0x0104B0C8 & 0xFFFF0000) == 0x01040000
(struct Pool *) ((size_t)  p & POOL_MASK)
```

Distinguishing partitions

- Partitions are separate sets of pools
- Pool remembers which partition it's in (typically pool header)

Align pools to get nice pool mask

"Pool mask": 0xFFFF0000

```
(0x0104B0C8 & 0xFFFF0000) == 0x01040000
(struct Pool *) ((size_t)  p & POOL_MASK)
```

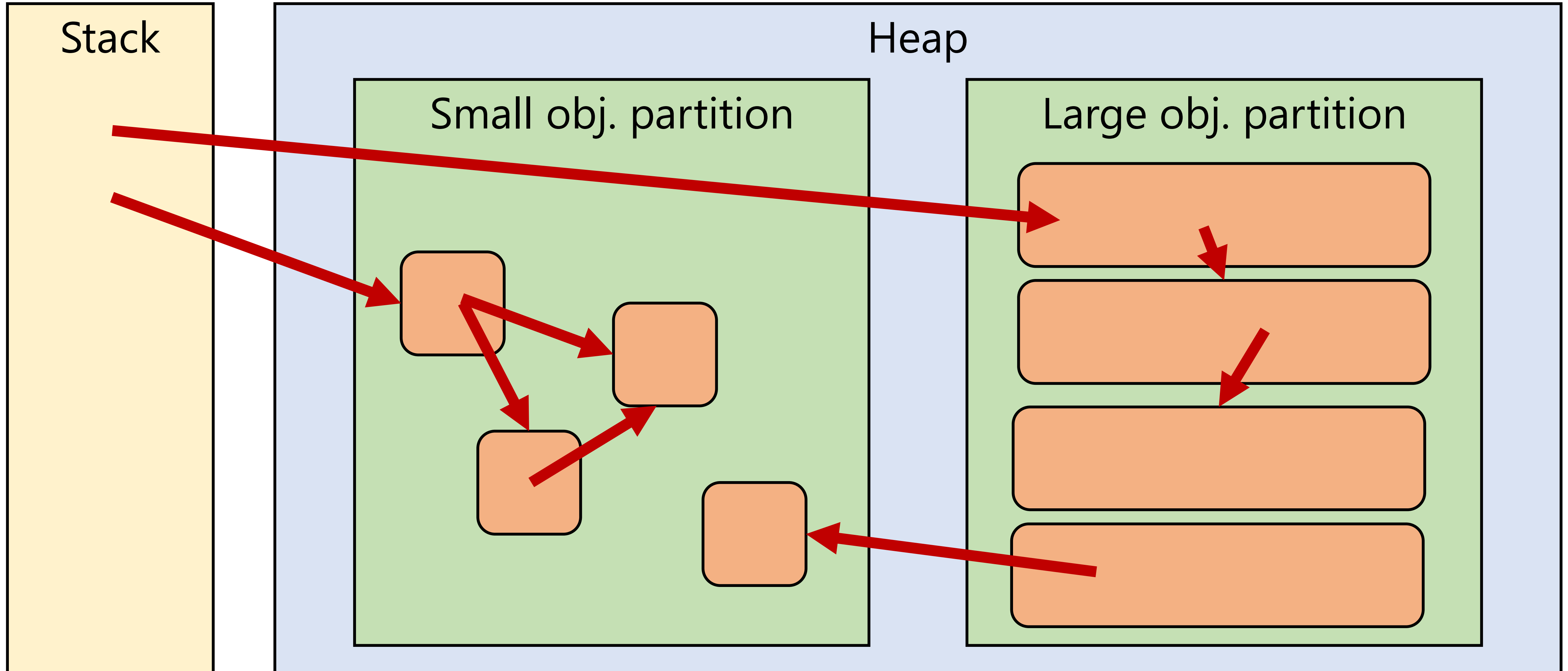
Breather

- Partition to use best of multiple strategies
- Partitions just part of heap: References from roots and other partitions
- Algorithm mixes by checking partition
- Partition by mobility for C

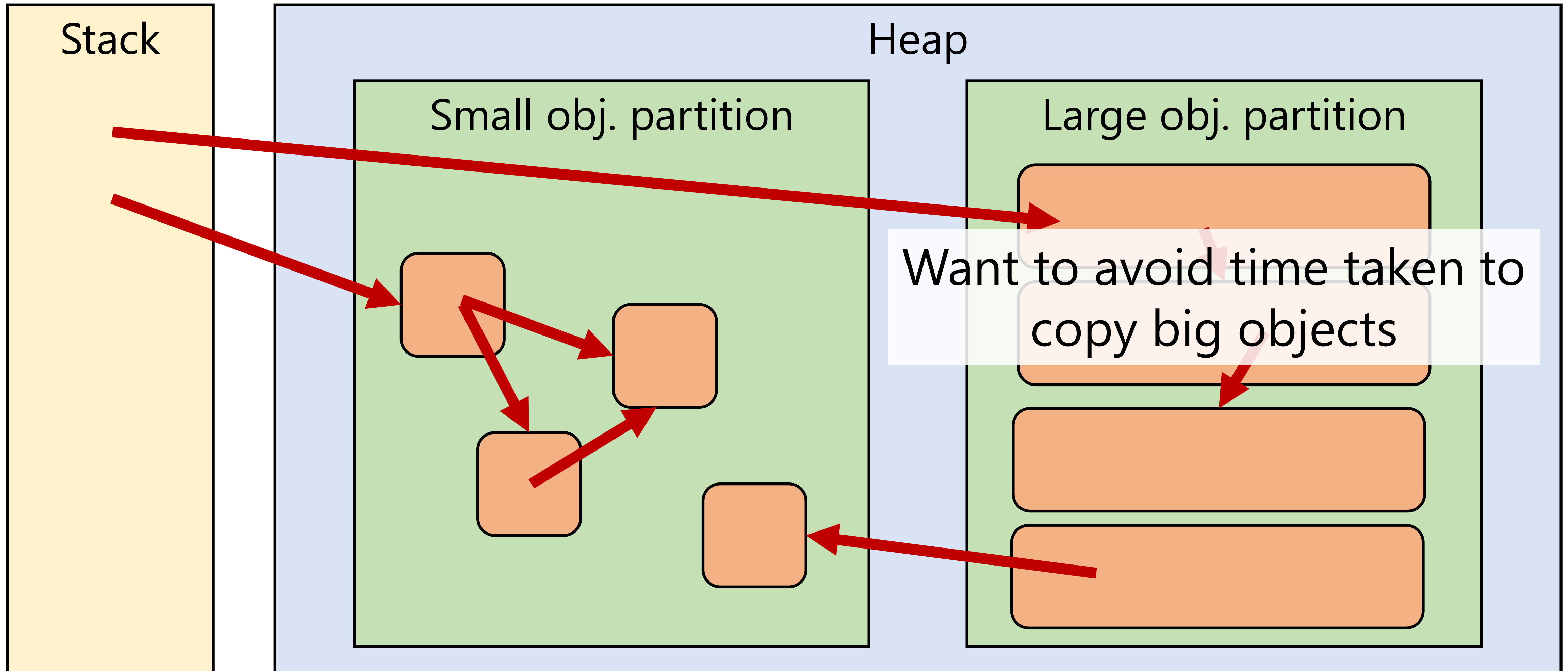
Partition by size

- Segregated blocks: Different pools for different object sizes, no fragmentation
- Copying: No fragmentation + improved locality, must copy objects
- Mix them:
  - Copy smallest objects
  - M&S + segregated blocks for larger
  - M&S + regular free-list for largest

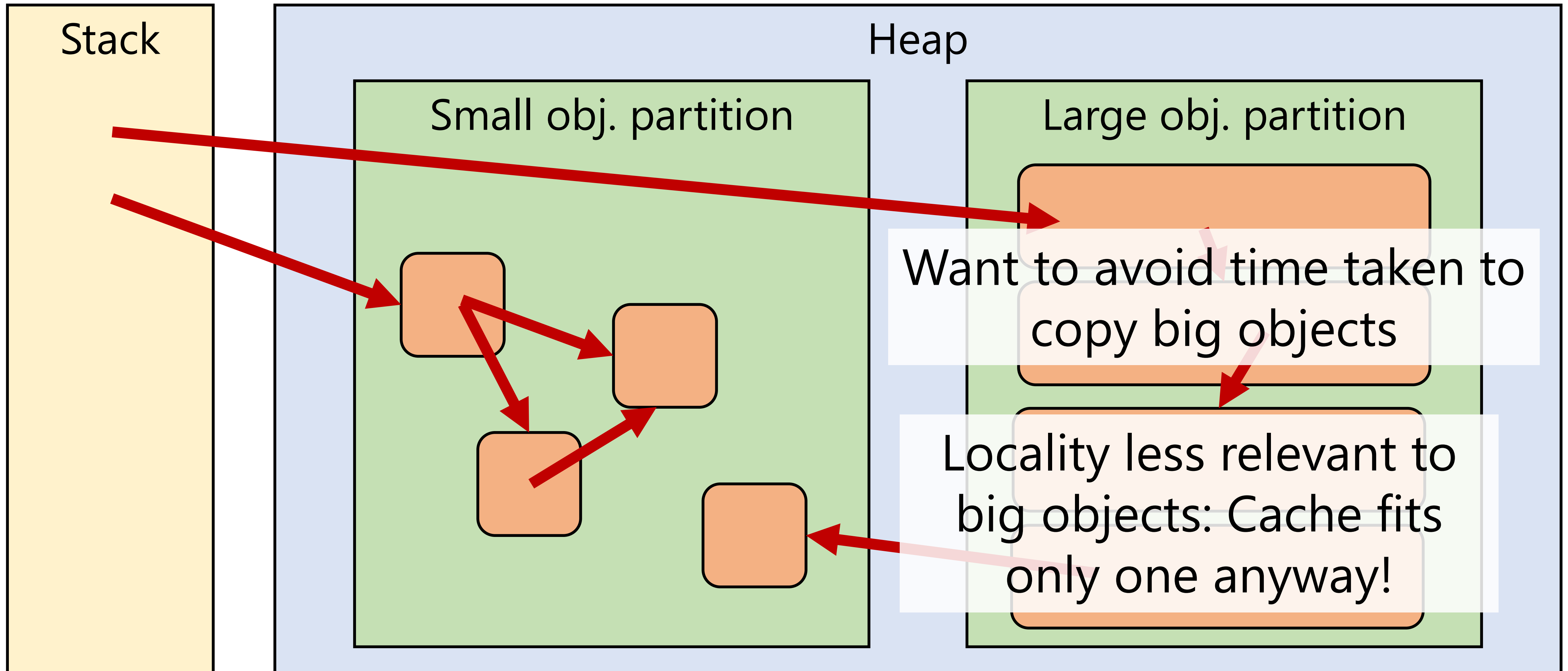
Partitioning for size



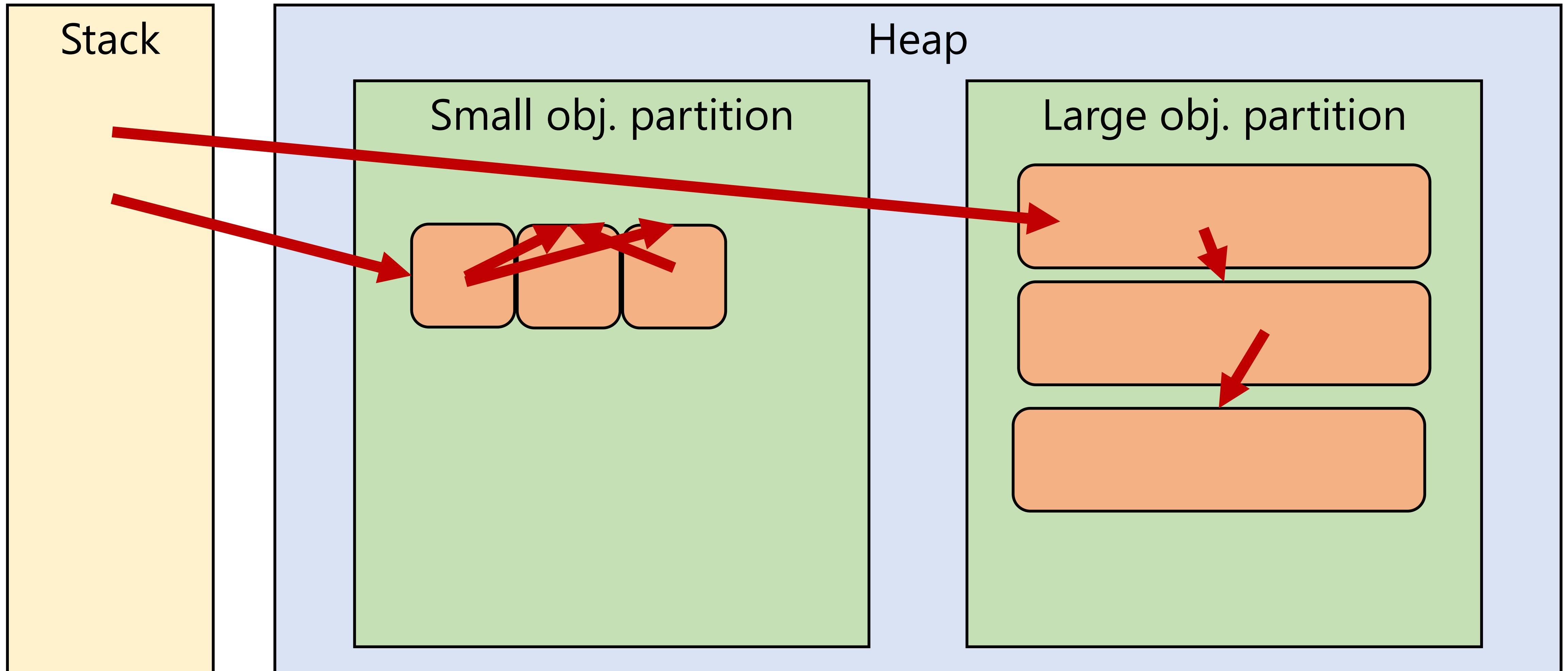
Partitioning for size



Partitioning for size



Partitioning for size



Partition by size

- Natural extension of segregated blocks
- One size per pool [copy pool(s) flexible]
- Too-large objects have own pool(s)
- No fragmentation except last pool(set)
- Other benefits: Fast allocation, full-heap sweep only on large objects (faster)

Partition by kind

- “Kind” may have many meanings:
  - Type (e.g. language type annotations, mutability)
  - GC-relevant category (e.g. references vs. no references)
  - Runtime properties (e.g. owner, trust, source)
  - Memory properties (e.g. alignment, executability)

Partitioning by executability

- JIT compilers generate code at runtime
- That code can die
- That code must be executable
- Making whole heap executable is a *very bad idea*™
- Place executable "objects" on own executable heap

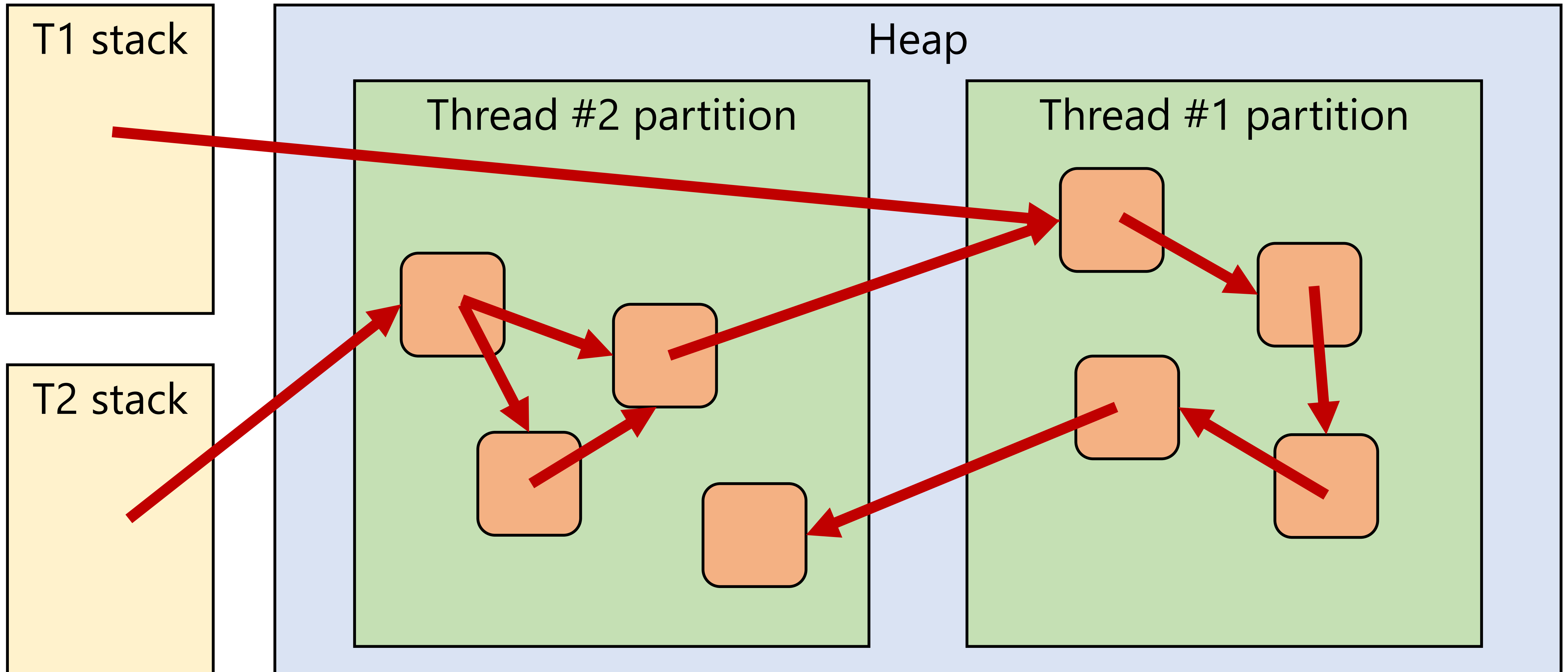
Partitioning by thread

- Threads cause problems:
  - Allocation contention
  - Stop-the-world (all threads must yield)
  - Let’s not even talk about reference counting
- Can partitioning by threads solve them?

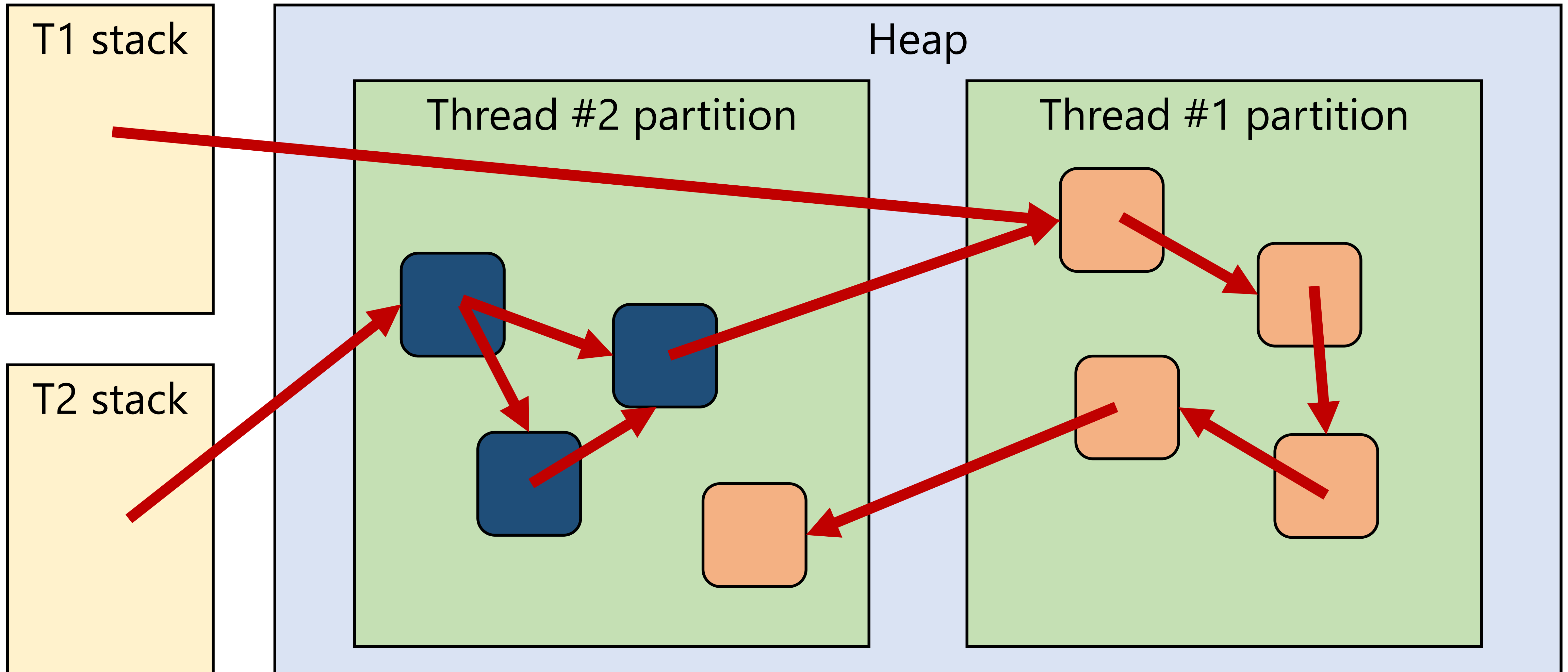
Partitioning by thread

- Each thread gets own partition
- Collect just one thread!

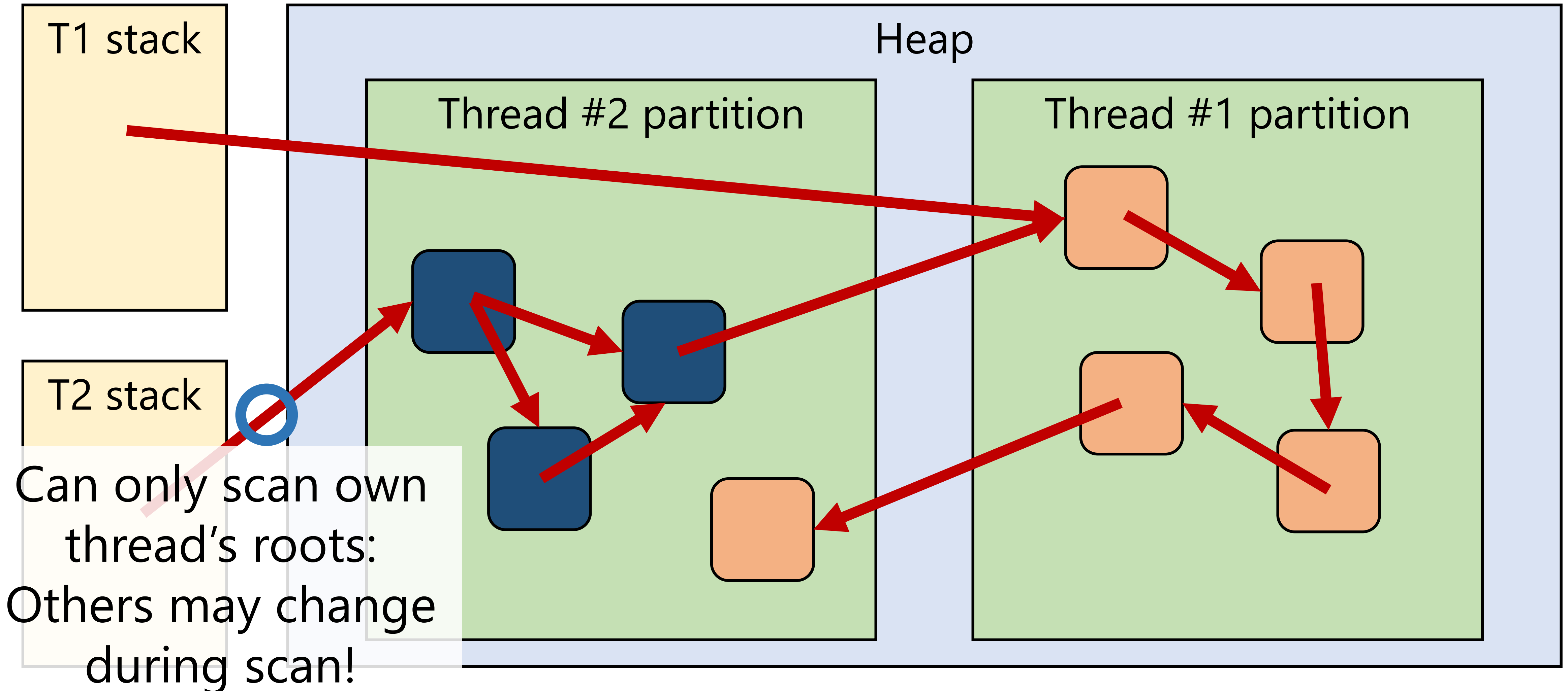
Partitioning by thread



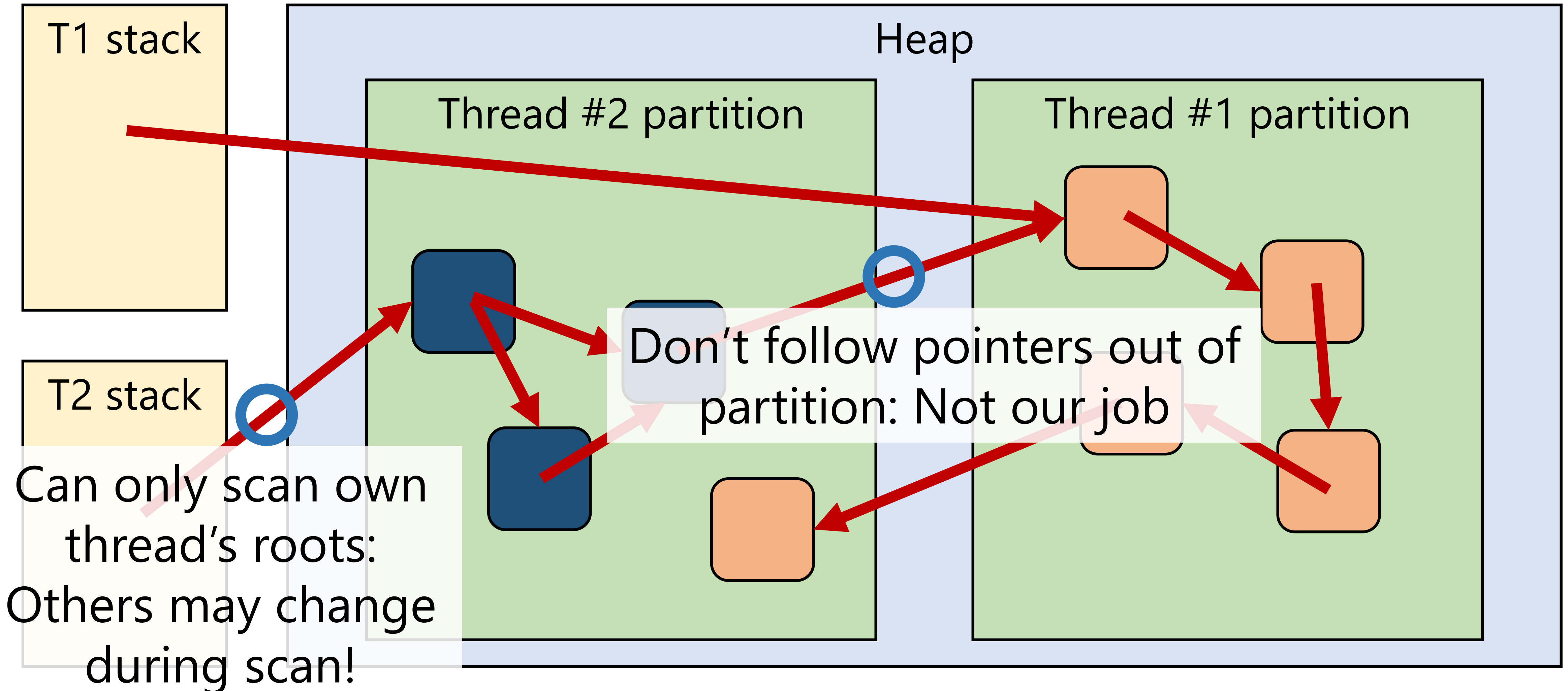
Partitioning by thread



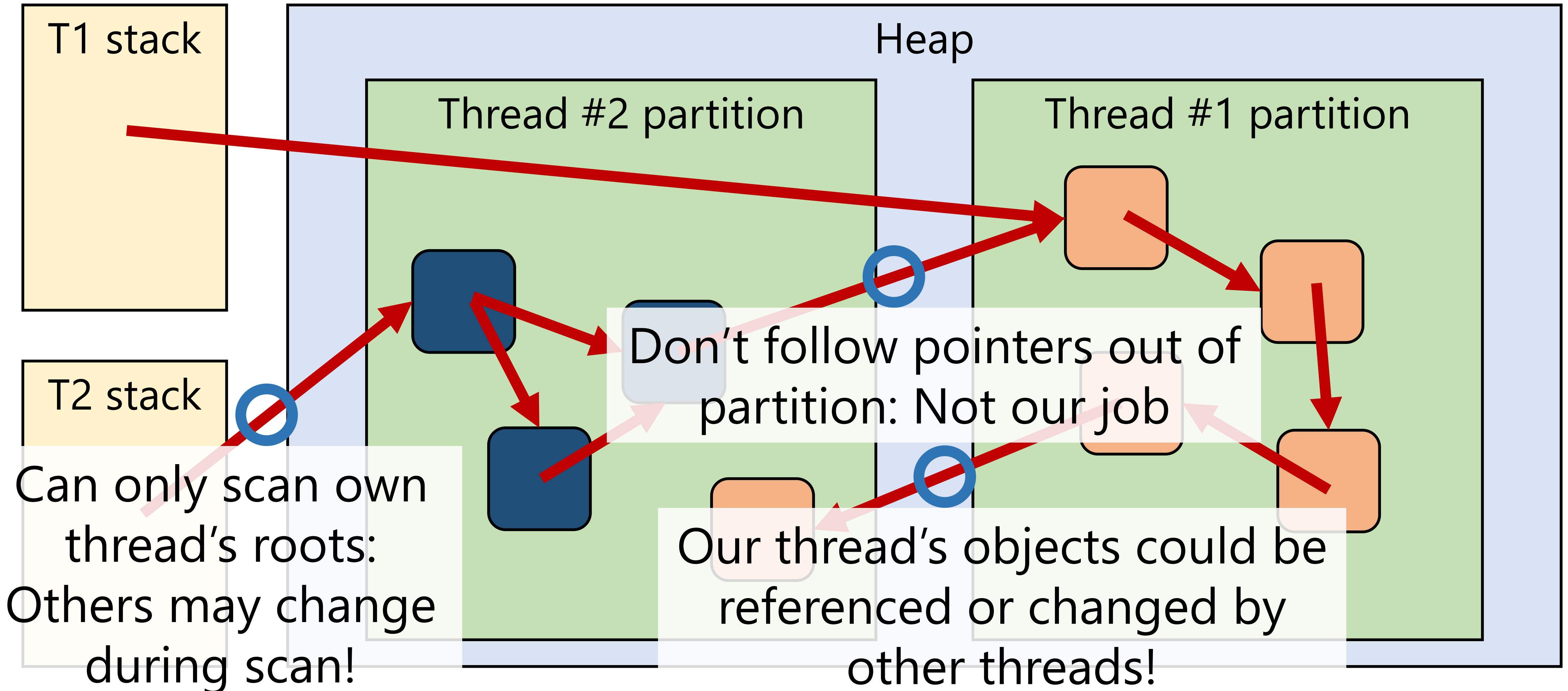
Partitioning by thread



Partitioning by thread



Partitioning by thread

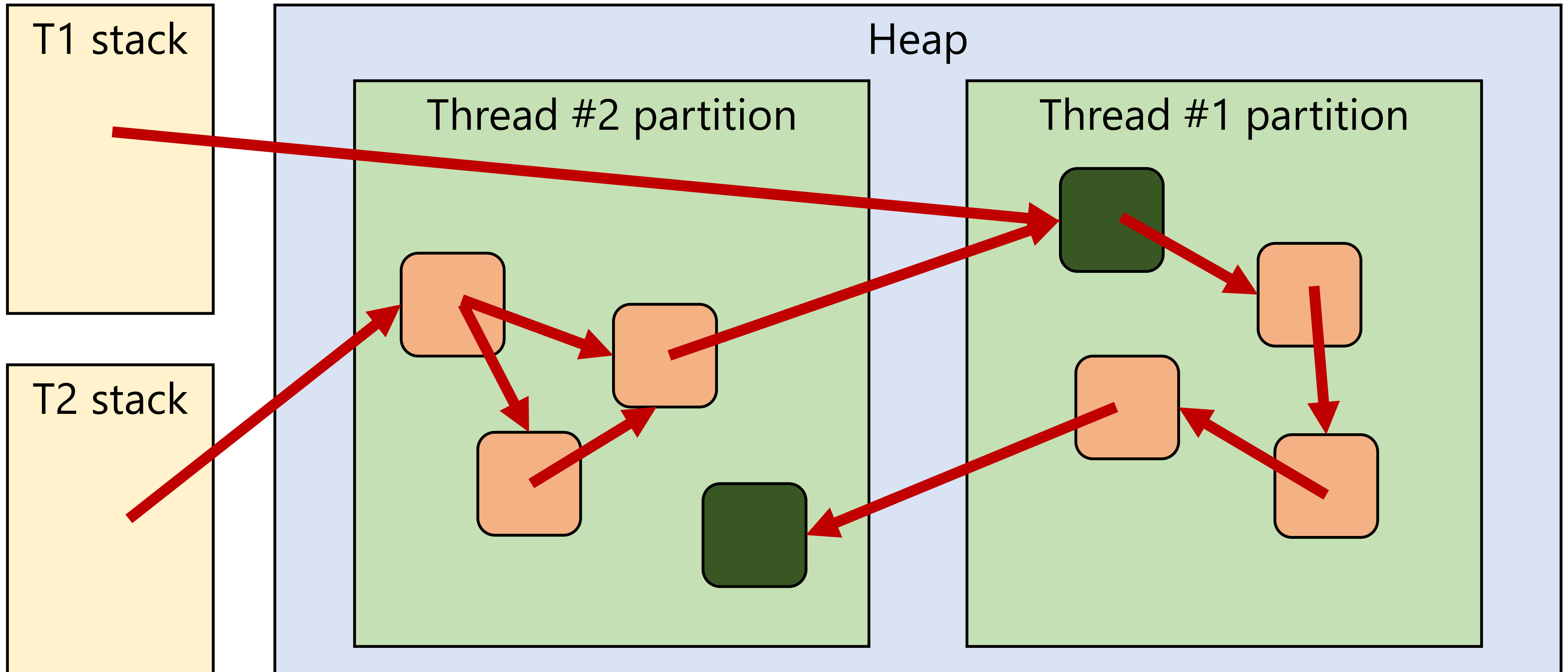


Inter-thread madness

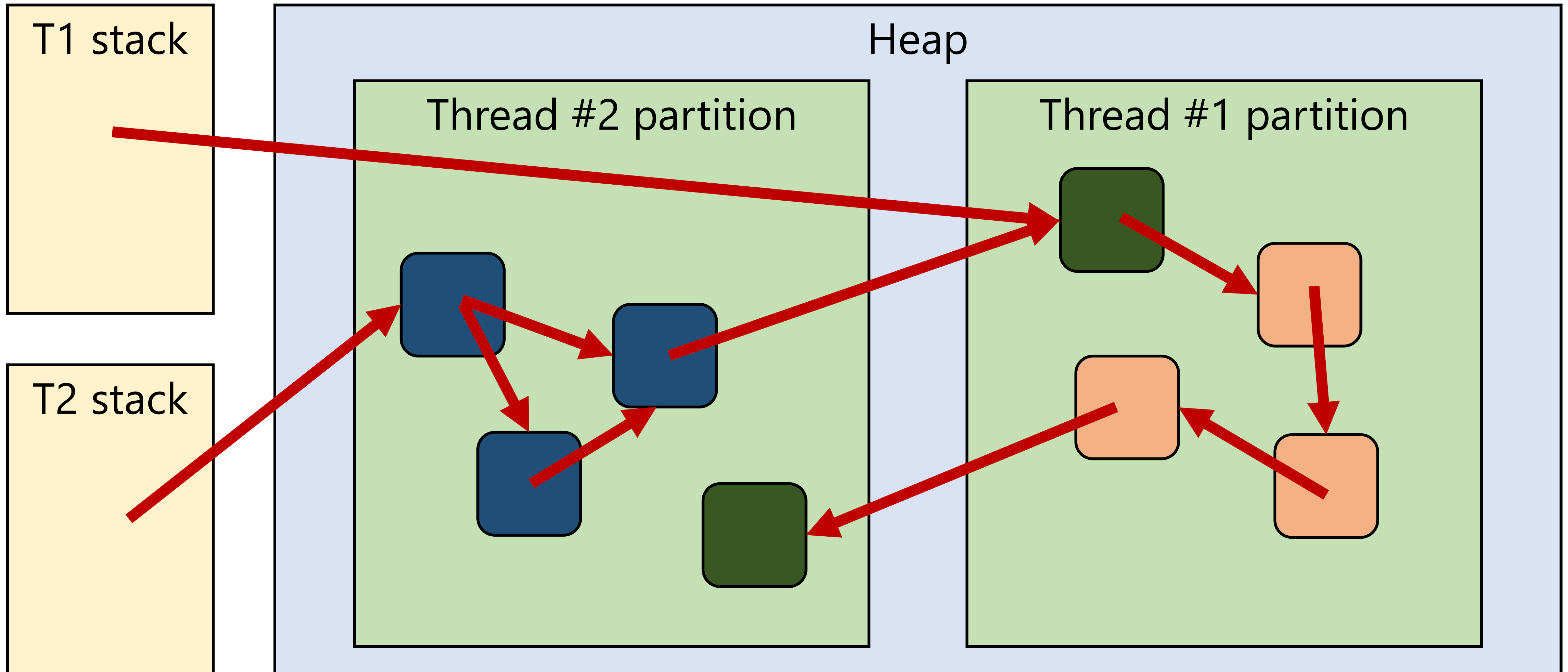
- Inter-thread links/modification causing problems
- Write barriers to the rescue!

```
write(loc, obj):
  if threadOf(loc) != threadOf(obj) or
     threadOf(loc) != myThread():
    markAsInterthread(obj)
  *loc := obj
```

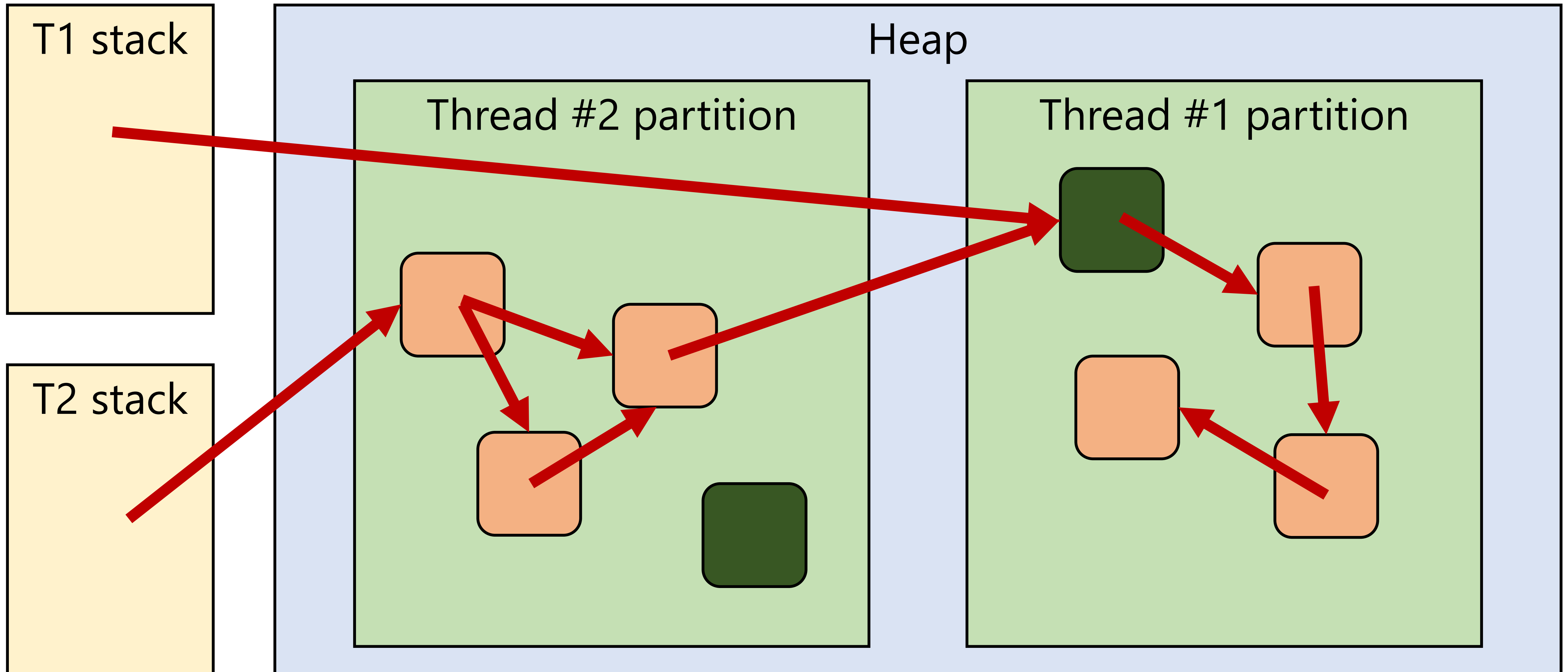
Partitioning by thread



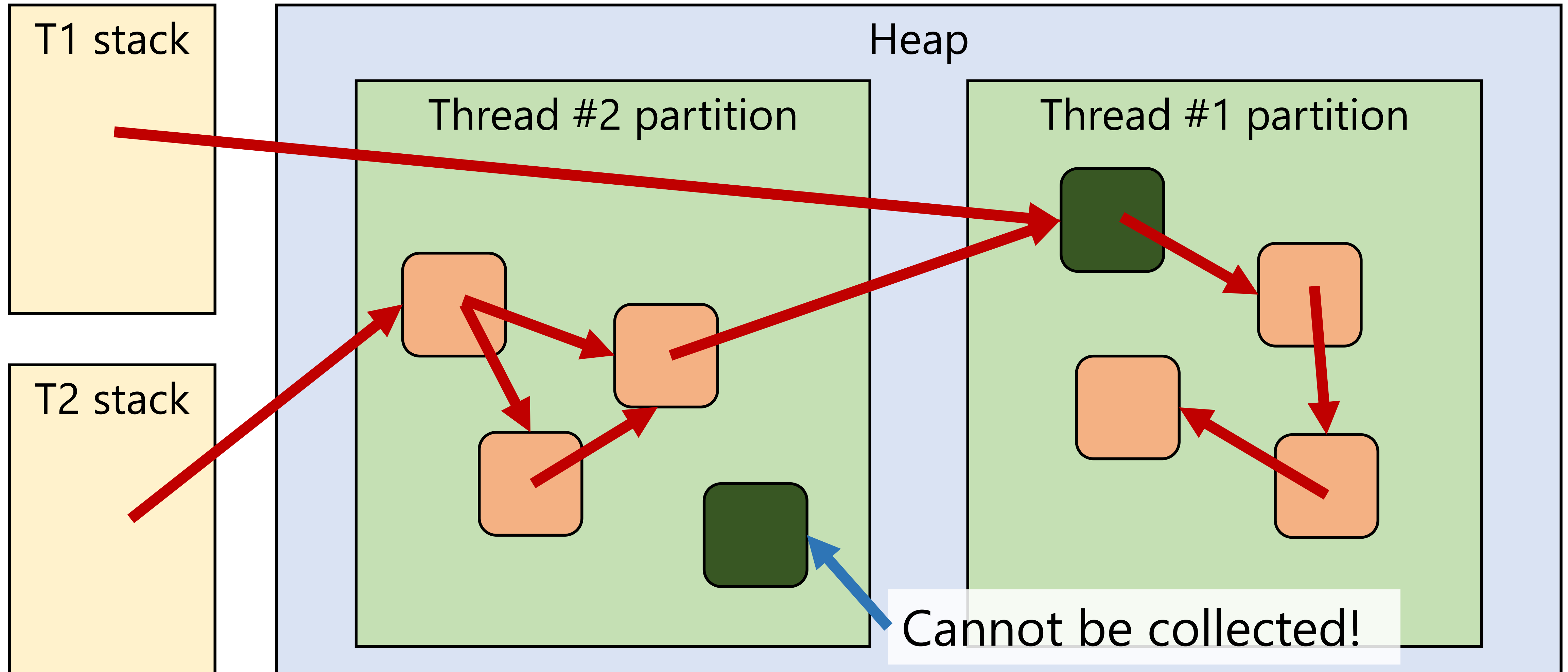
Partitioning by thread



Partitioning by thread



Partitioning by thread



Inter-thread objects

- Inter-thread mark is a long-term mark
- Any object with inter-thread references cannot be collected in partial GC
- Still need occasional full GC to collect inter-thread objects
- Maybe move objects to other threads

Thread-local allocation

- Without thread partitioning, allocation must lock
- Big lock, big contention!
- Partition per thread: No locking
- Partitioning by thread for allocation worthwhile even without per-thread GC

When/what to GC

- With flat GC: When all pools full
- With partitioned GC: When a partition is full
- How to decide when to do a full GC?
  - Depends on partitioning scheme…
  - Threads: When large portion of objects are inter-thread marked

Partitioning by age

- “Young” partition and “old” partition
- Called “generations”
- Objects allocated in young partition
- Move to old partition if they survive
- Usually collect only young (most objects die young)

Partitioning by age

- “Young” partition and “old” partition
- Called “generations”
- Objects allocated in young partition
- Move to old partition if they survive
- Usually collect only young (most objects die young)

Generational garbage collection!