Partitioning the Heap

Schedule

	M	W
Sept 14	Intro/Background	Basics/ideas
Sept 21	Allocation/layout	GGGGC
Sept 28	Mark/Sweep	Copying GC
Octo 5	Details	Ref C
Octo 12	Thanksgiving	Mark/Compact
Octo 19	Partitioning/Gen	Generational
Octo 26	Other part	Runtime
Nove 2	Final/weak	Conservative
Nove 9	Ownership	Regions etc
Nove 16	Adv topics	Adv topics
Nove 23	Presentations	Presentations
Nove 30	Presentations	Presentations

Review

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

Tradeoffs

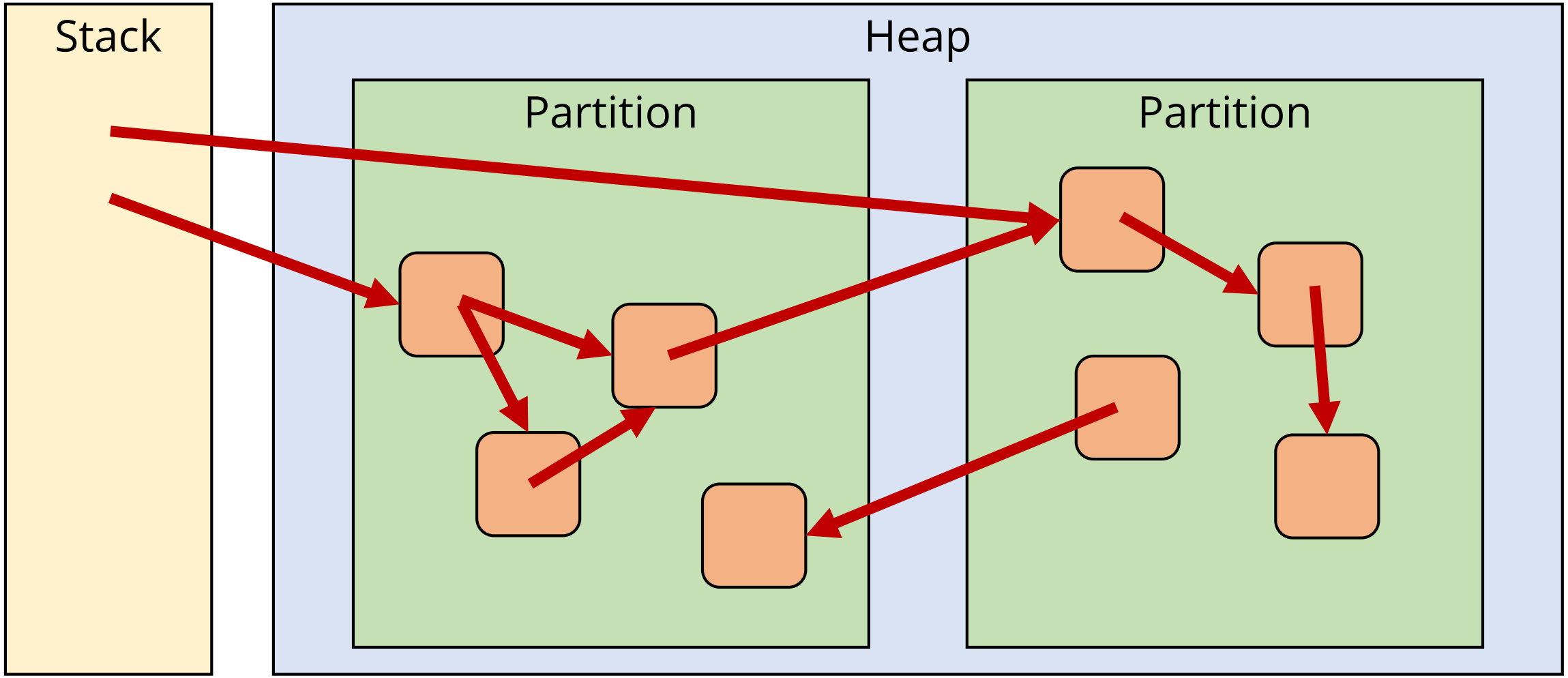
- Different strategies have different tradeoffs
- Mark-and-sweep: No moving, fragments
- Copying: Better locality, worst utilization
- Compacting: Good locality, very slow

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

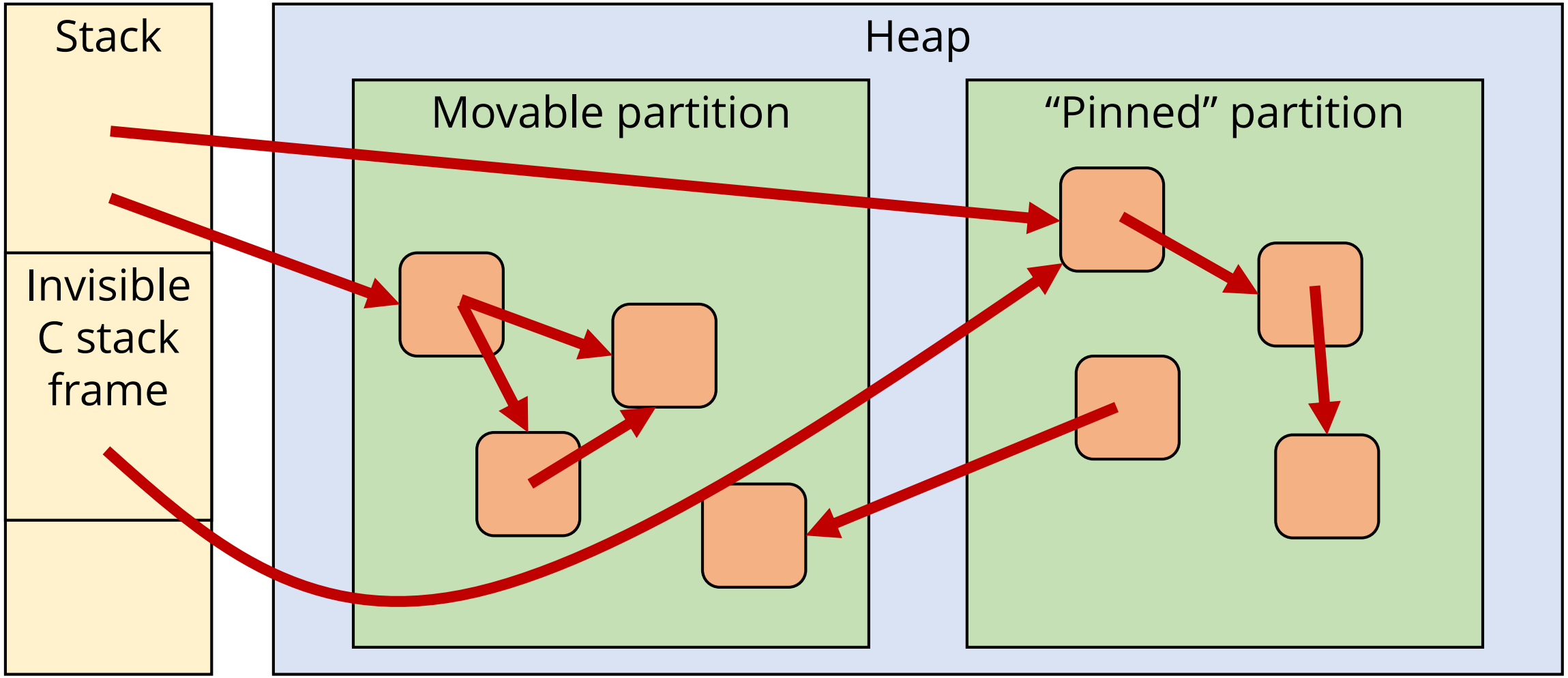
Mobility

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

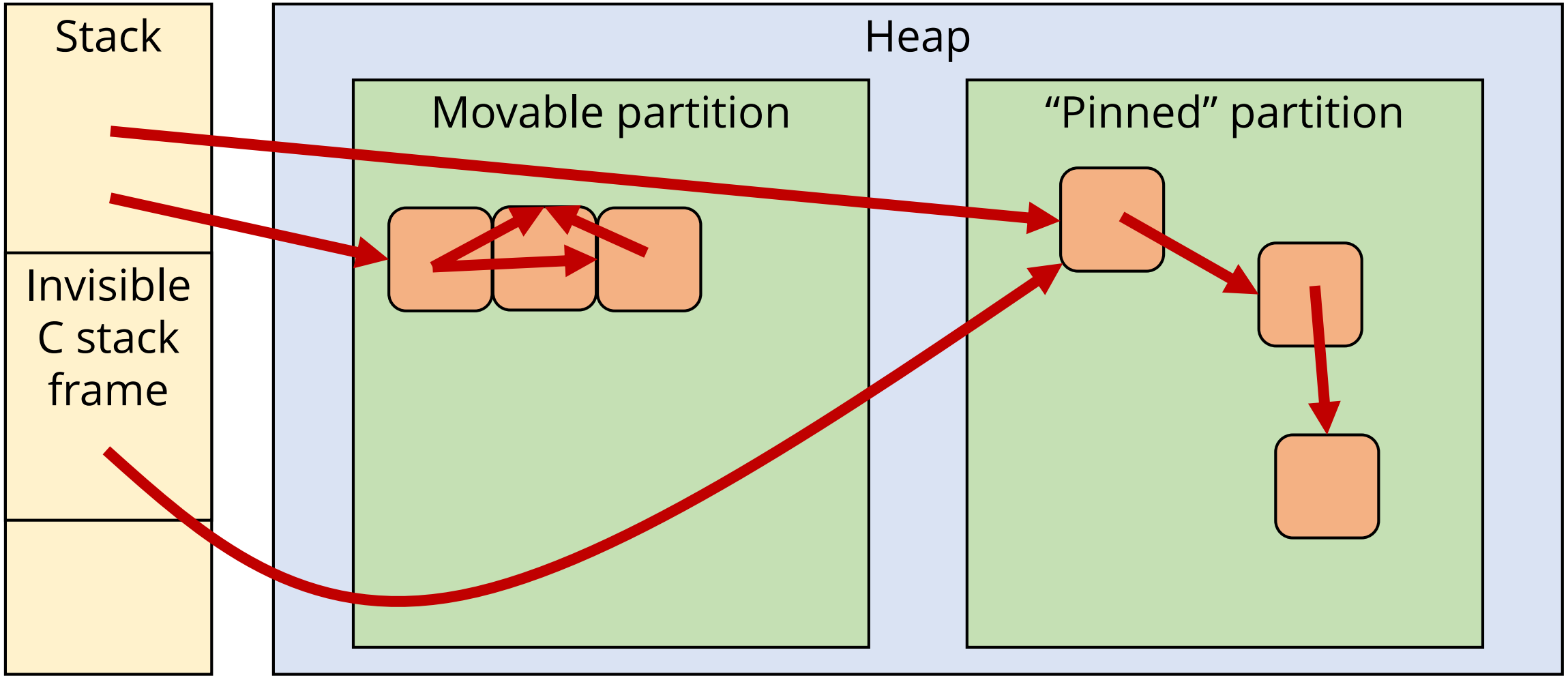
Mobility

- 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

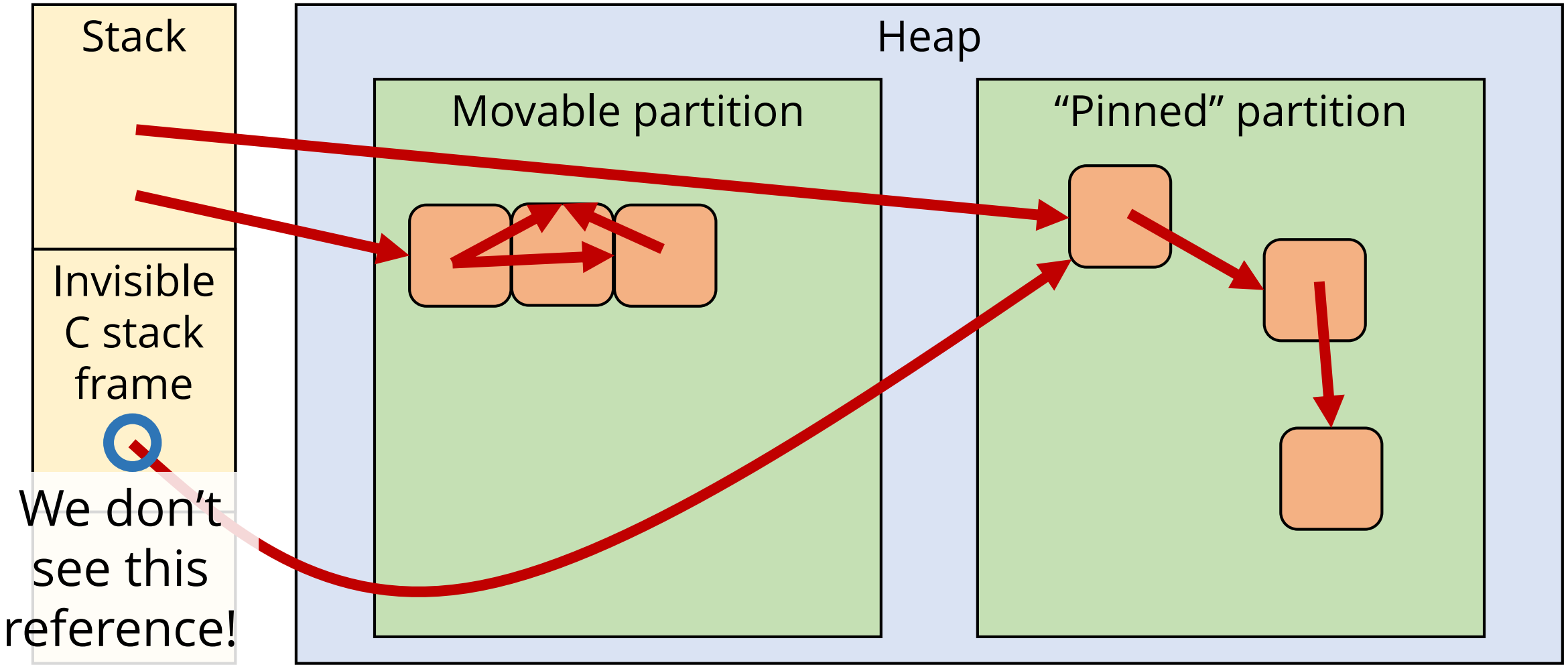
Partitioning for mobility



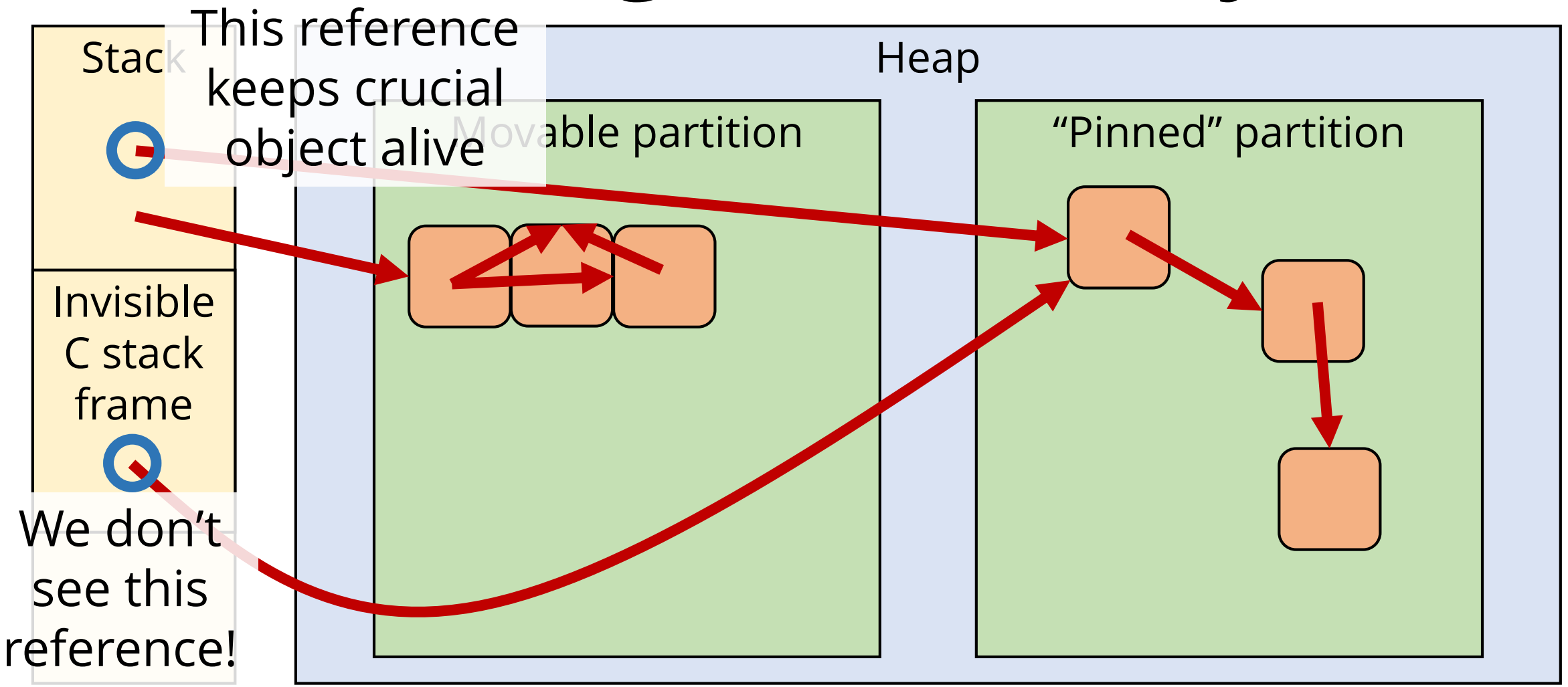
Partitioning for mobility



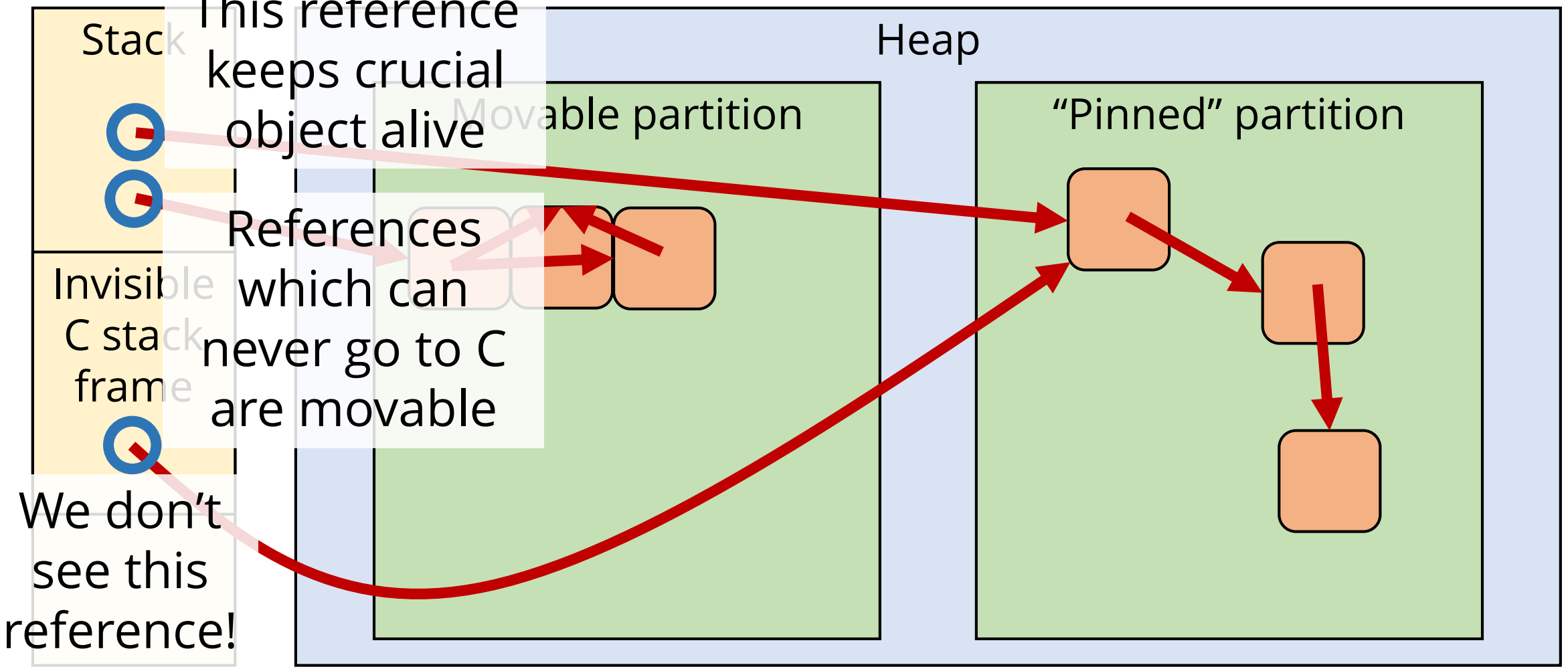
Partitioning for mobility



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)
```

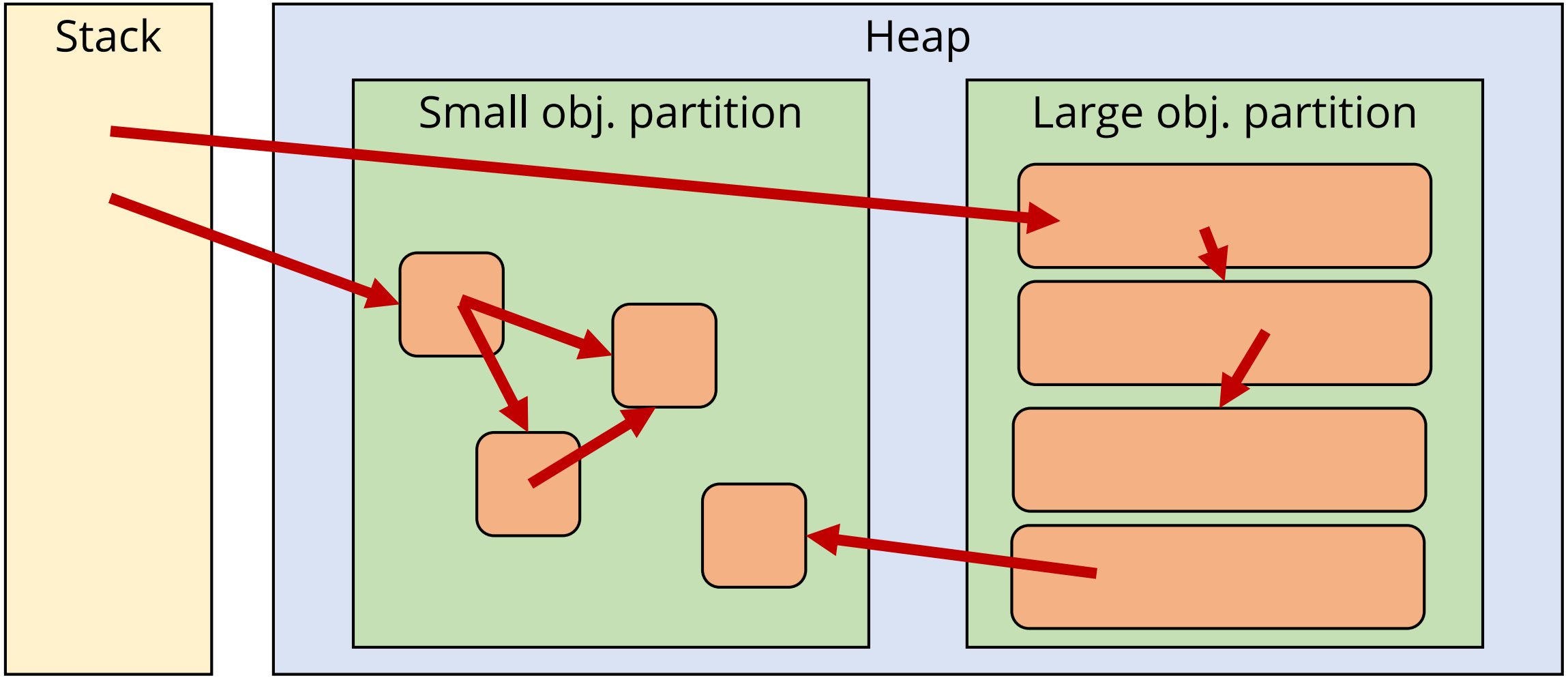
Breather

- 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

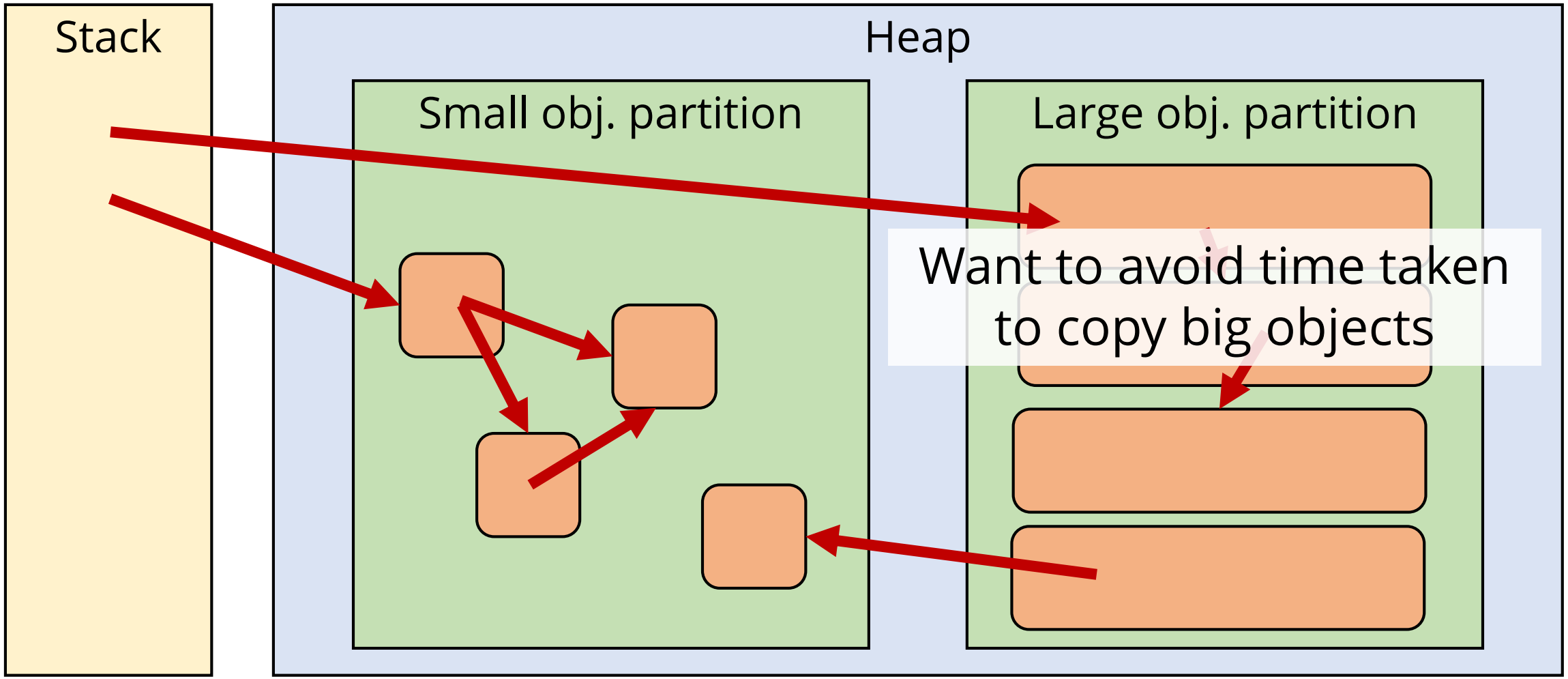
Partition by size

- 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

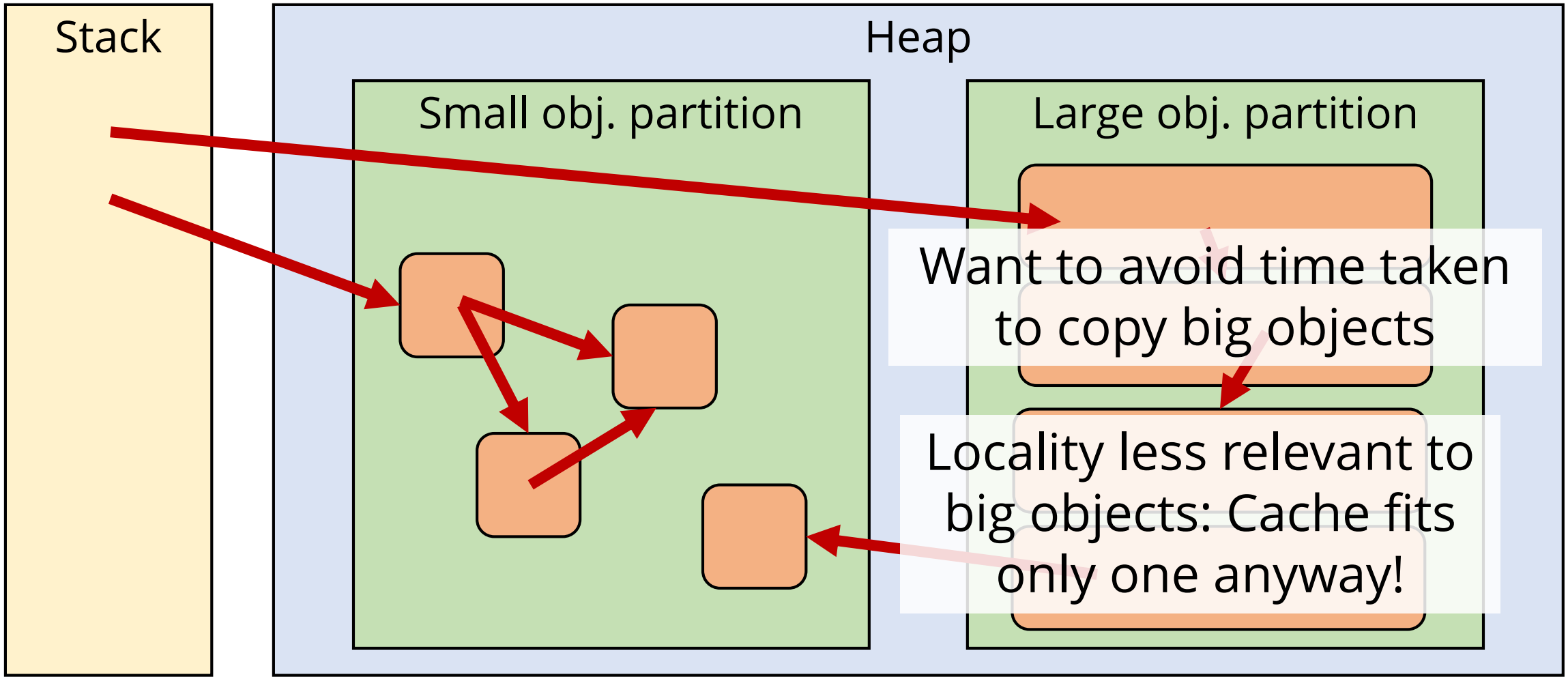
Partitioning for size



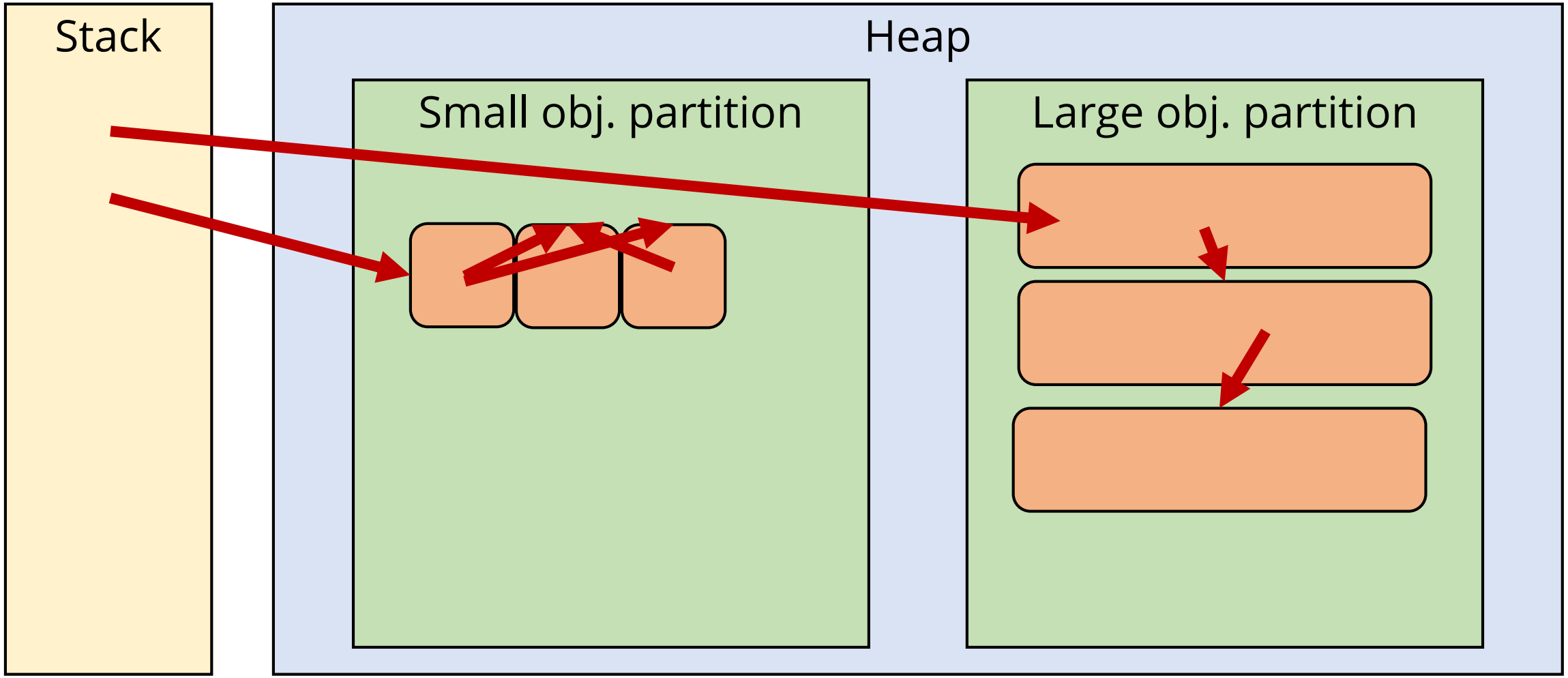
Partitioning for size



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

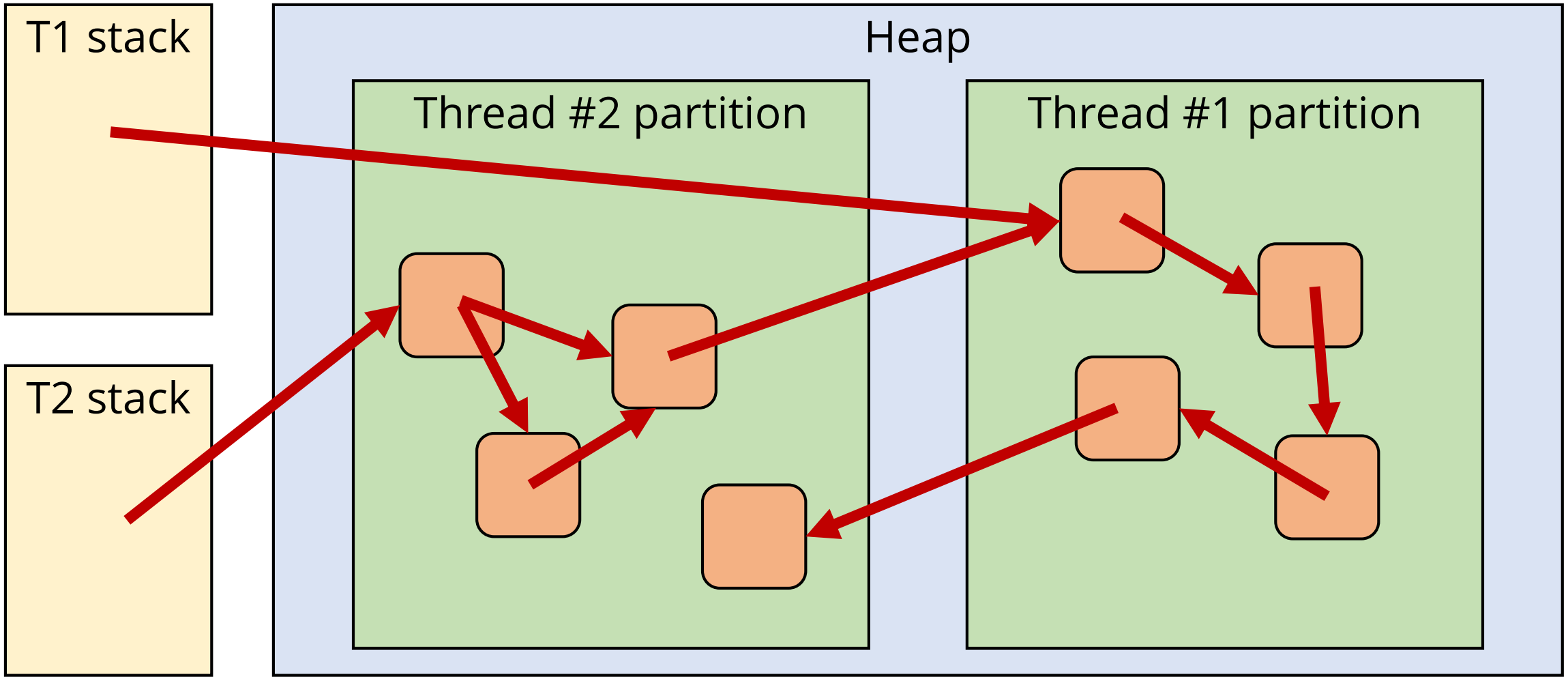
Partitioning by thread

- 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?

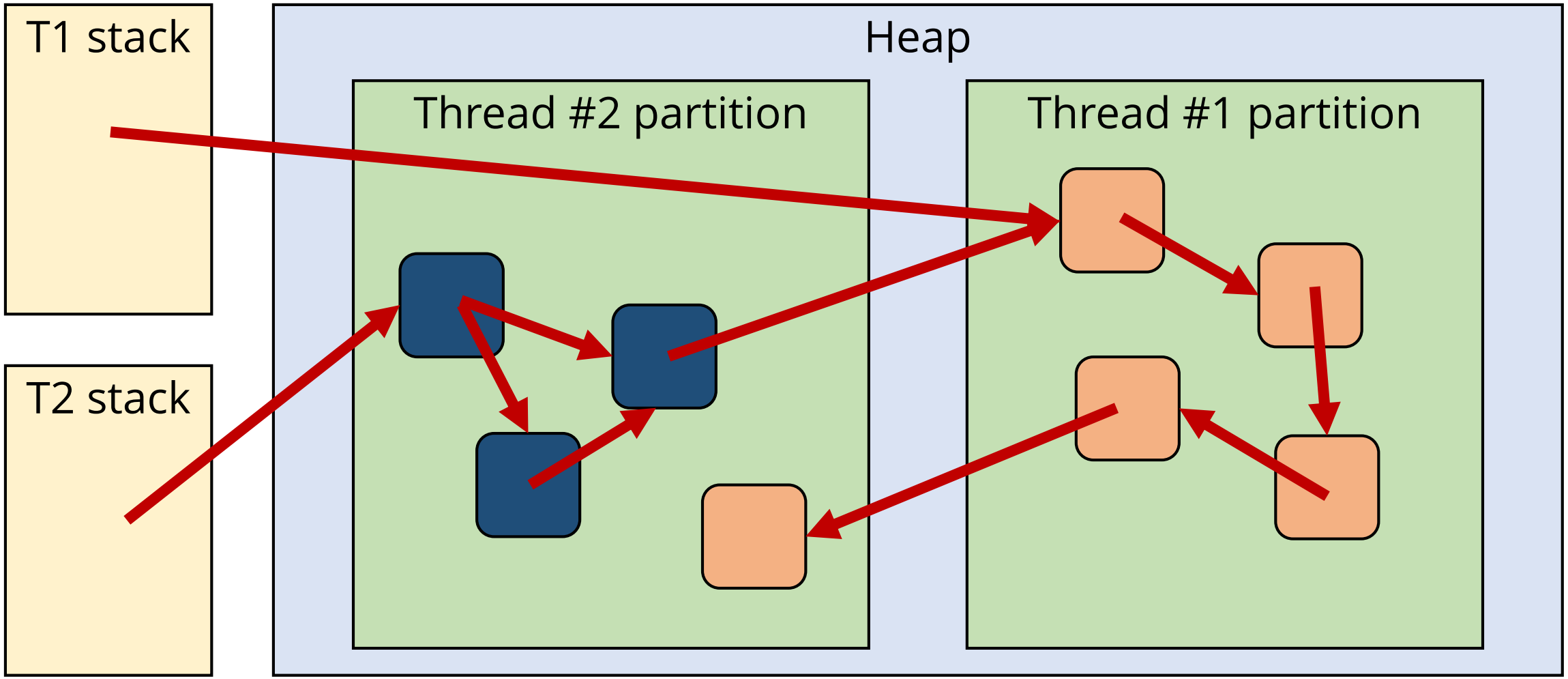
Partitioning by thread

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

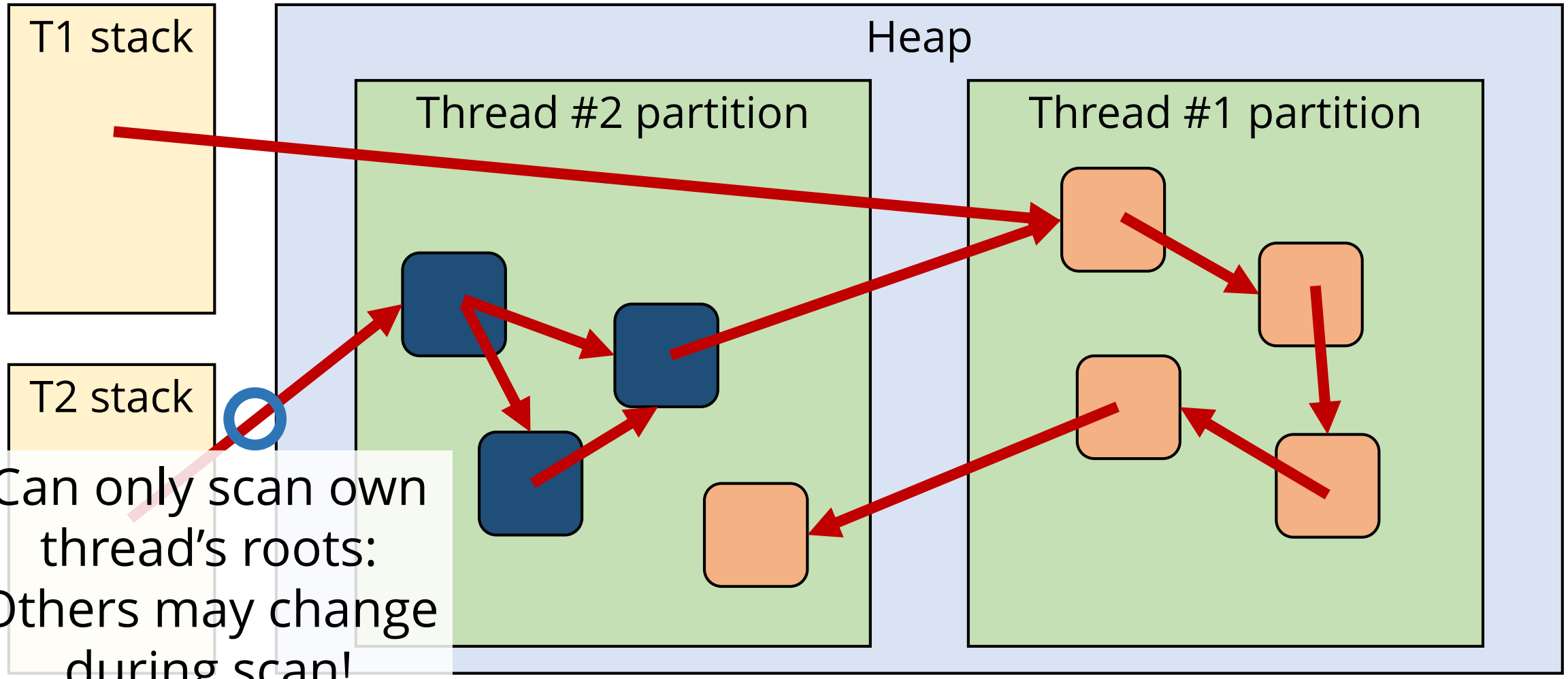
Partitioning by thread



Partitioning by thread

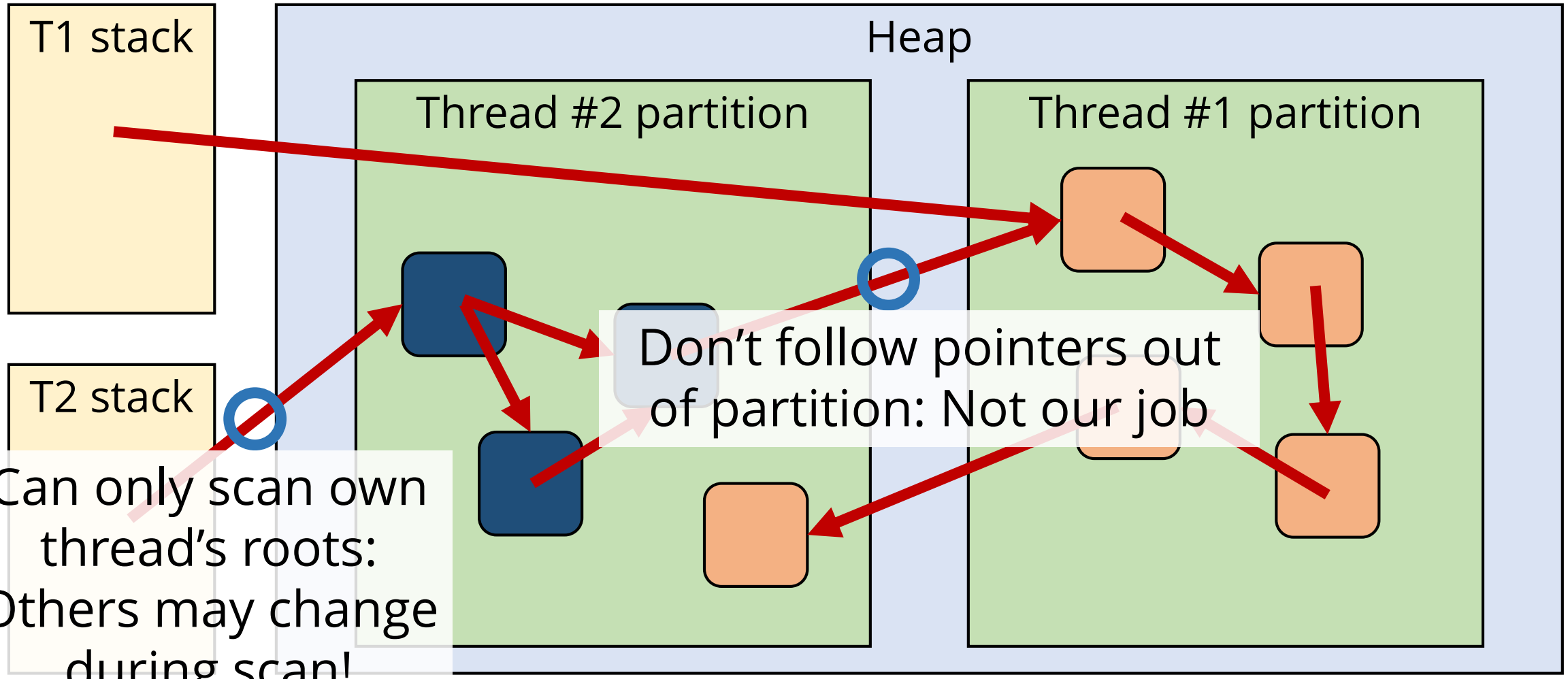


Partitioning by thread

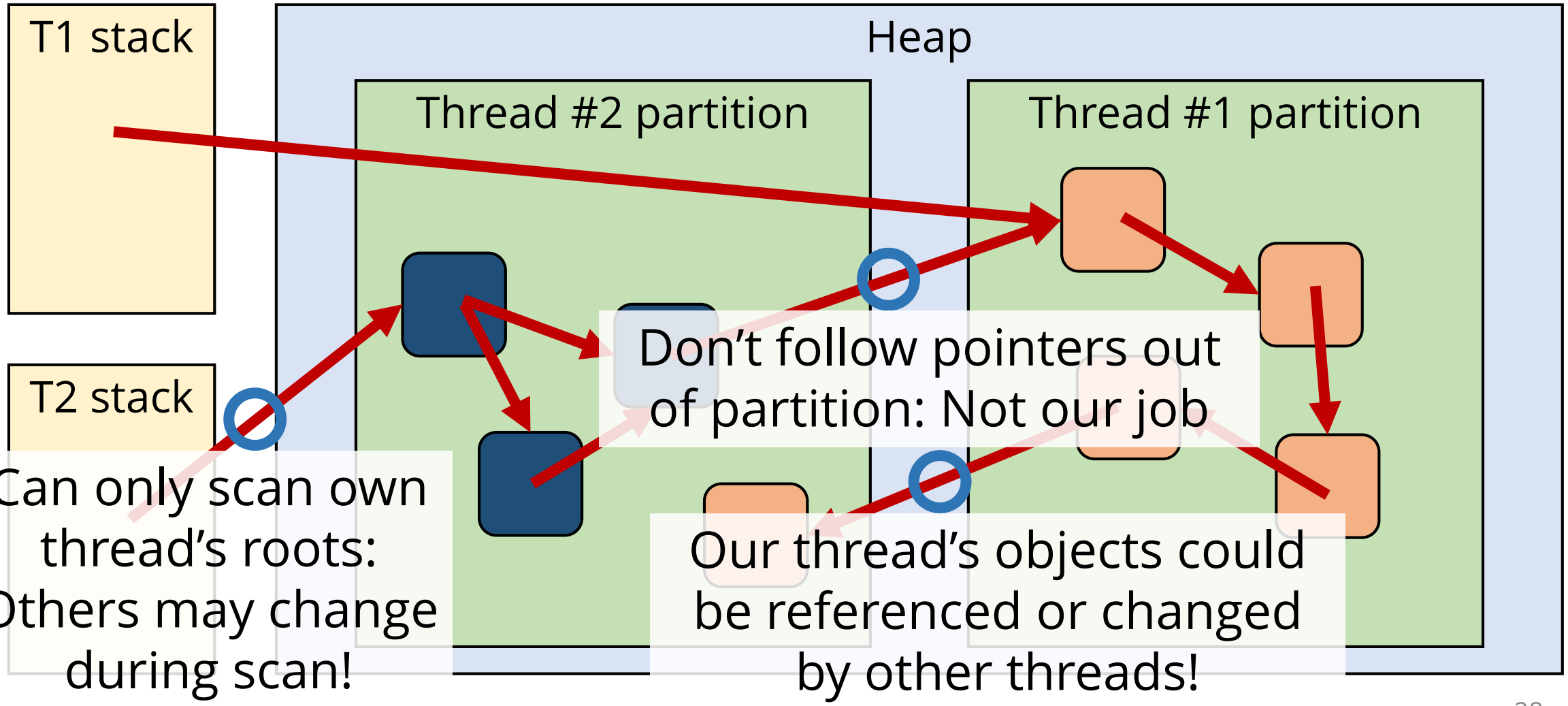


Can only scan own thread's roots: Others may change during scan!

Partitioning by thread



Partitioning by thread

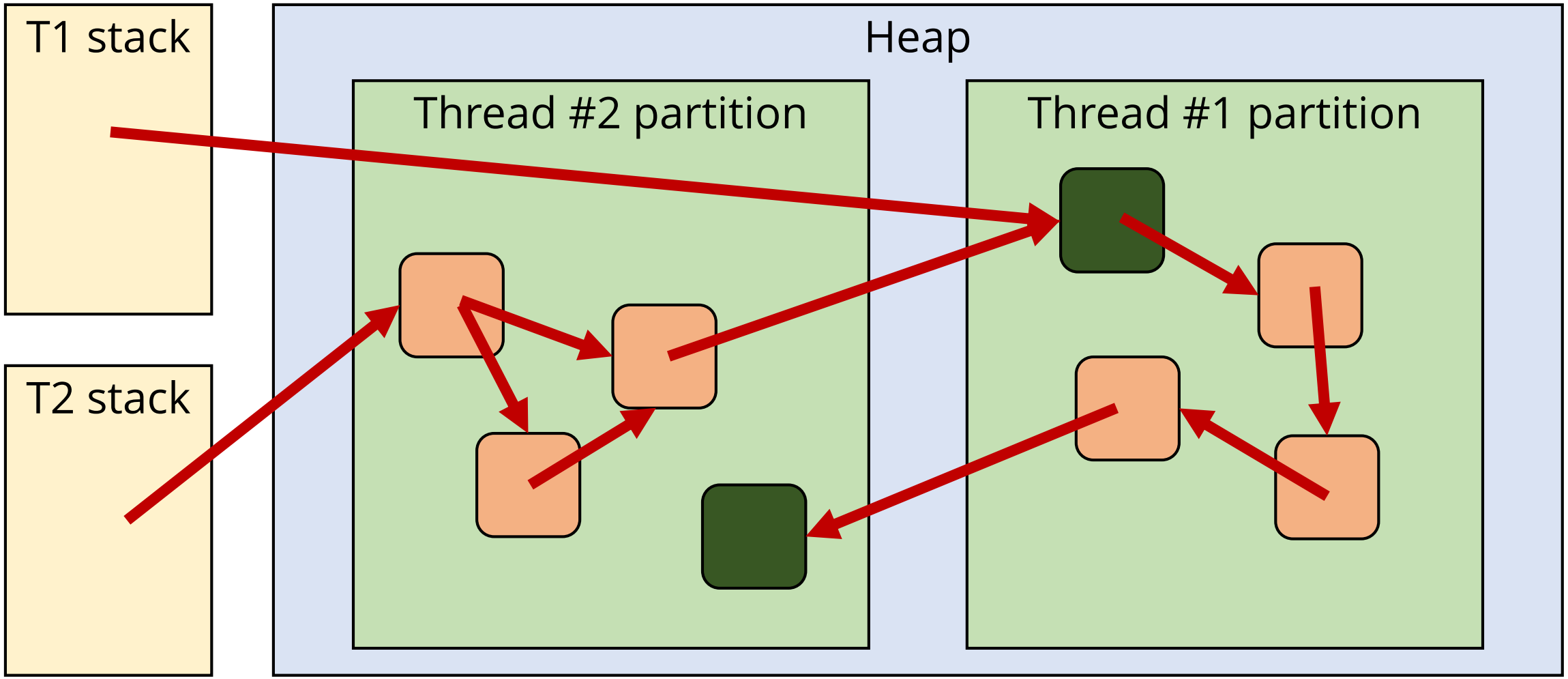


Inter-thread madness

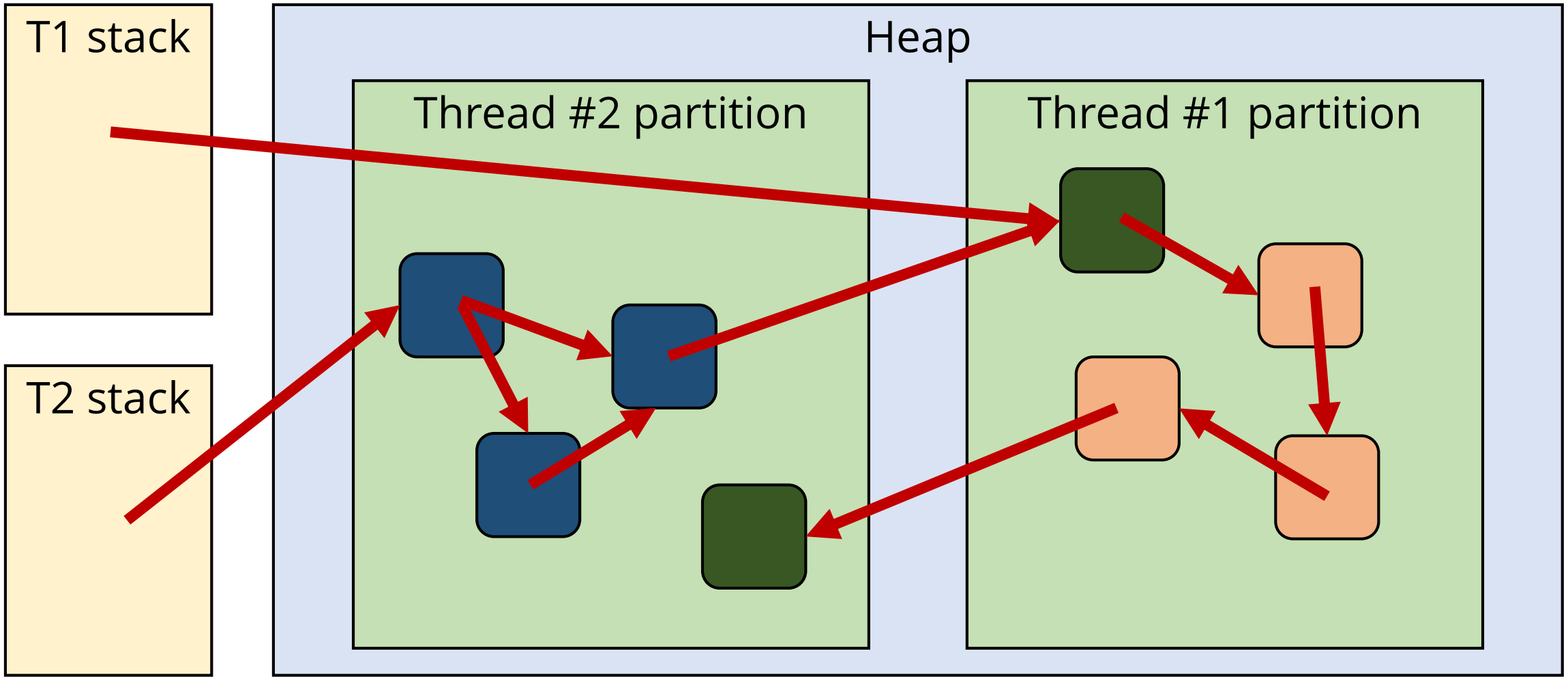
- Inter-thread links/modification causing problems
- Write barriers to the rescue!

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

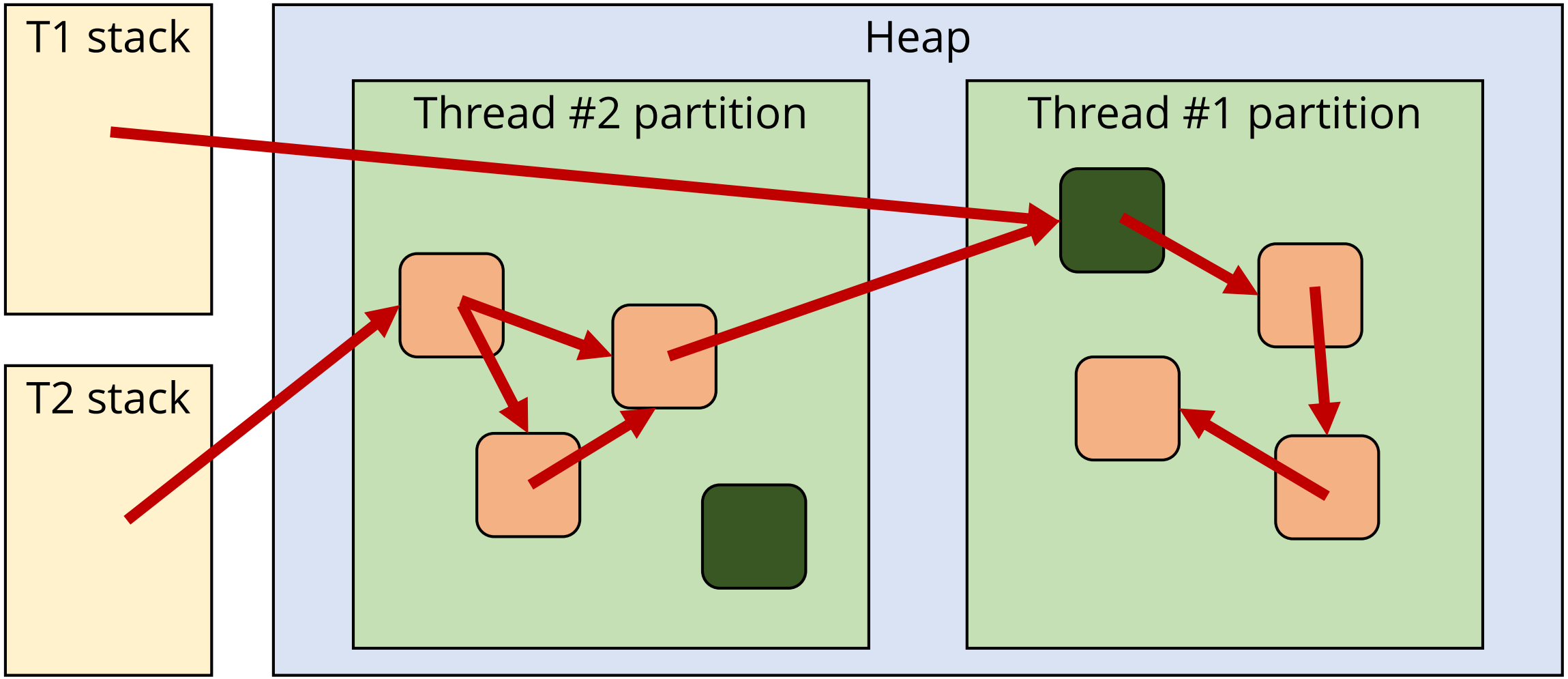
Partitioning by thread



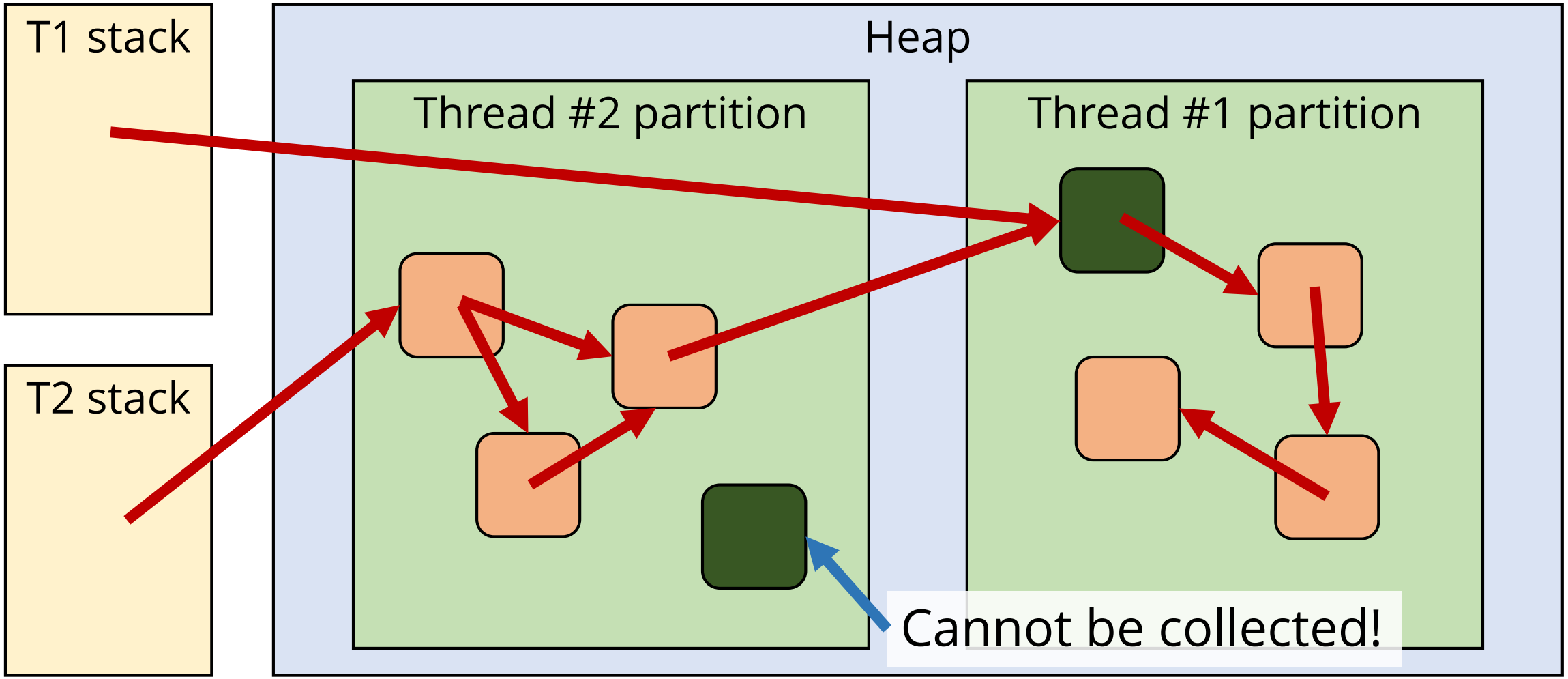
Partitioning by thread



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!