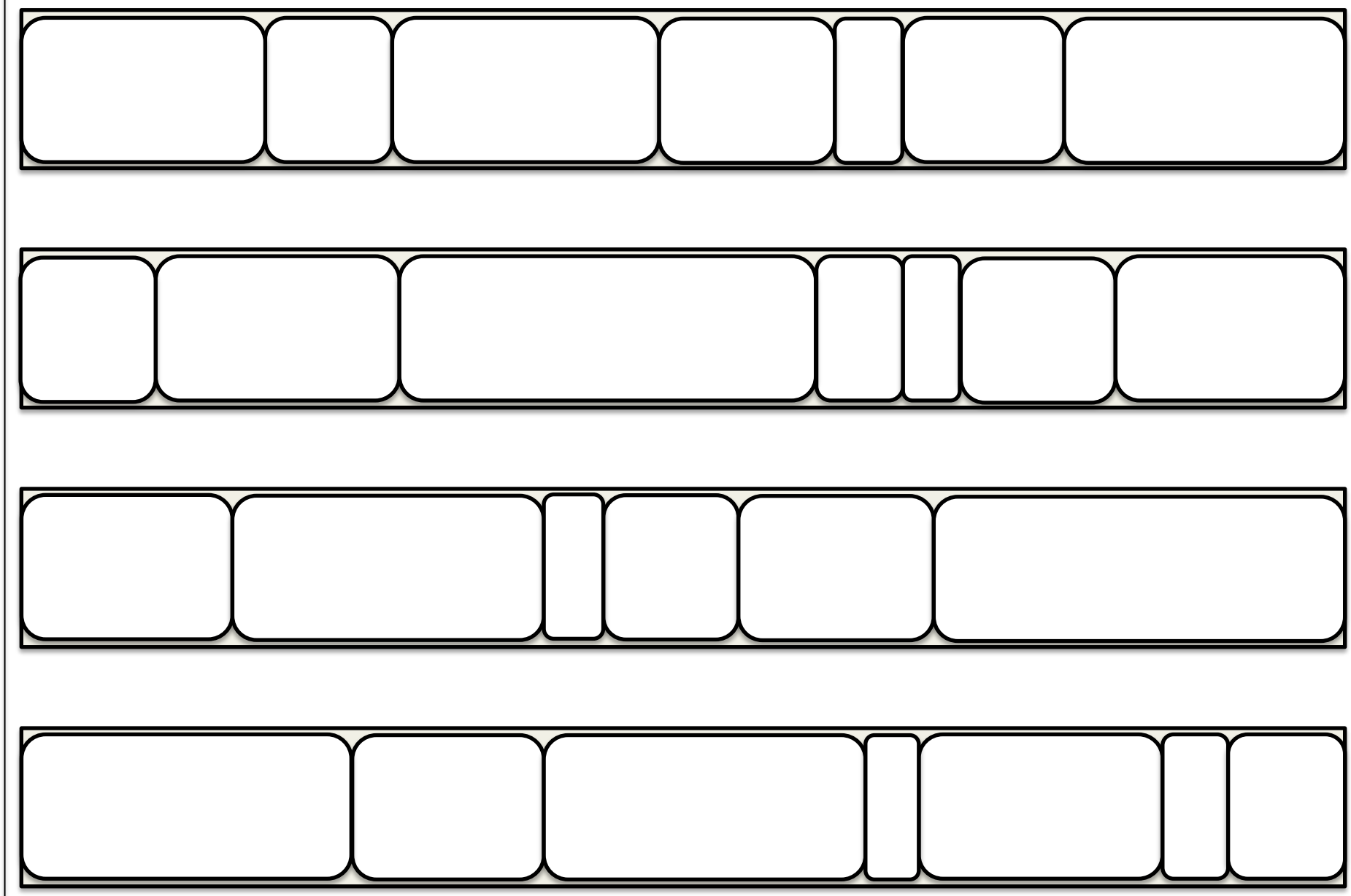
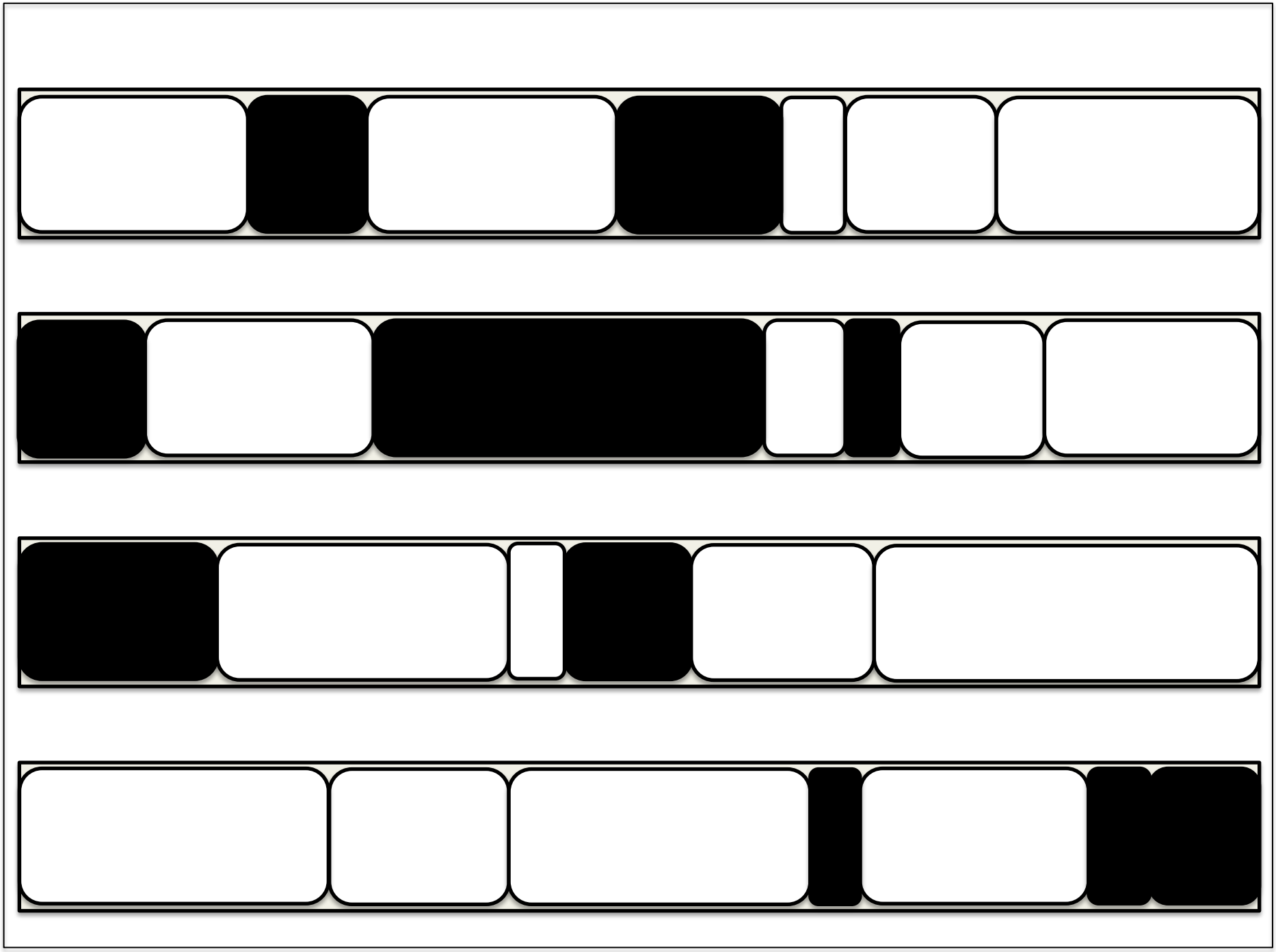
In the Last Week

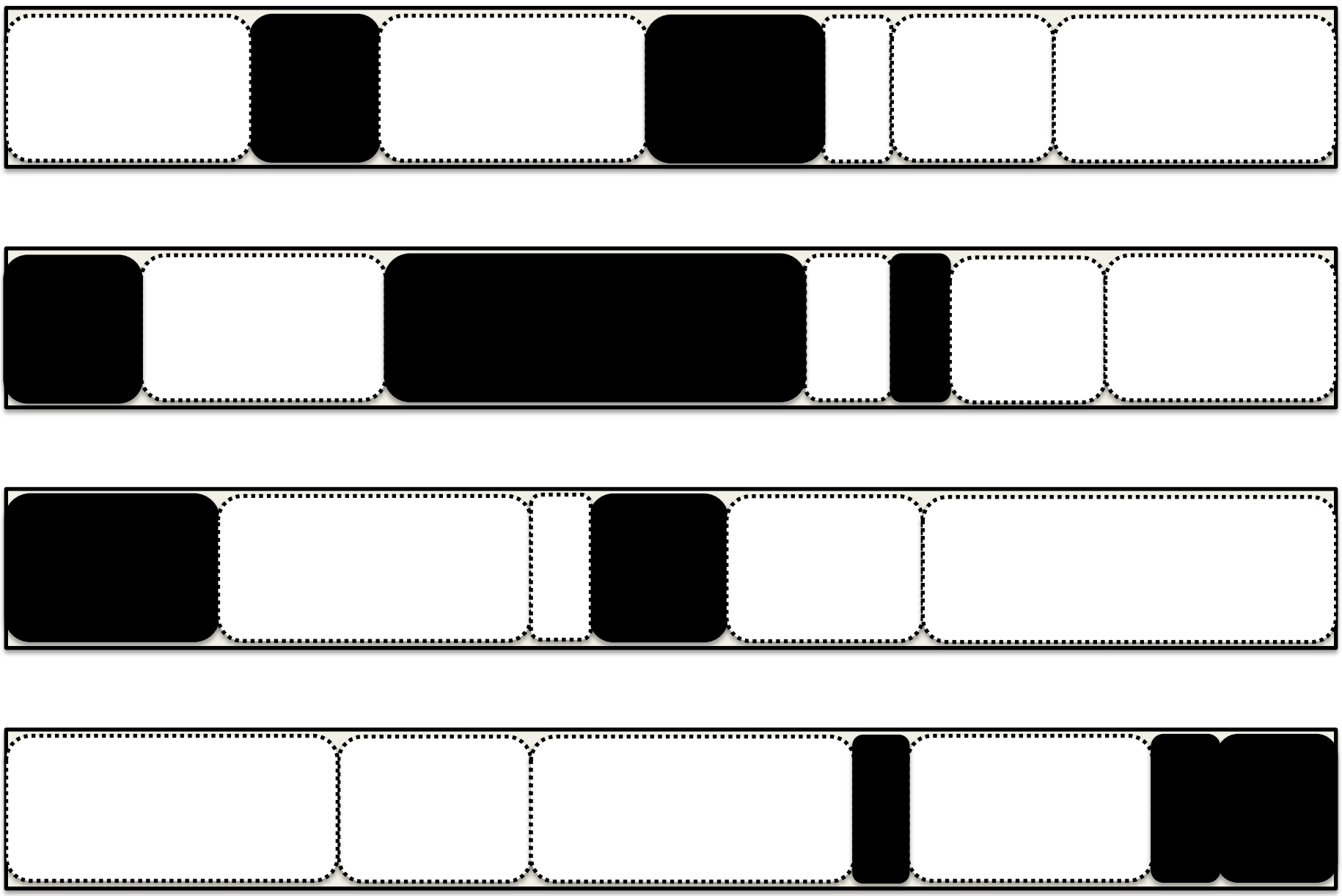
- Fairly quiet week
  - A few more suggestions for special topics
  - Some questions on the final project
- Going beyond the final project specification
- Objects and holes in the non-contiguous allocator

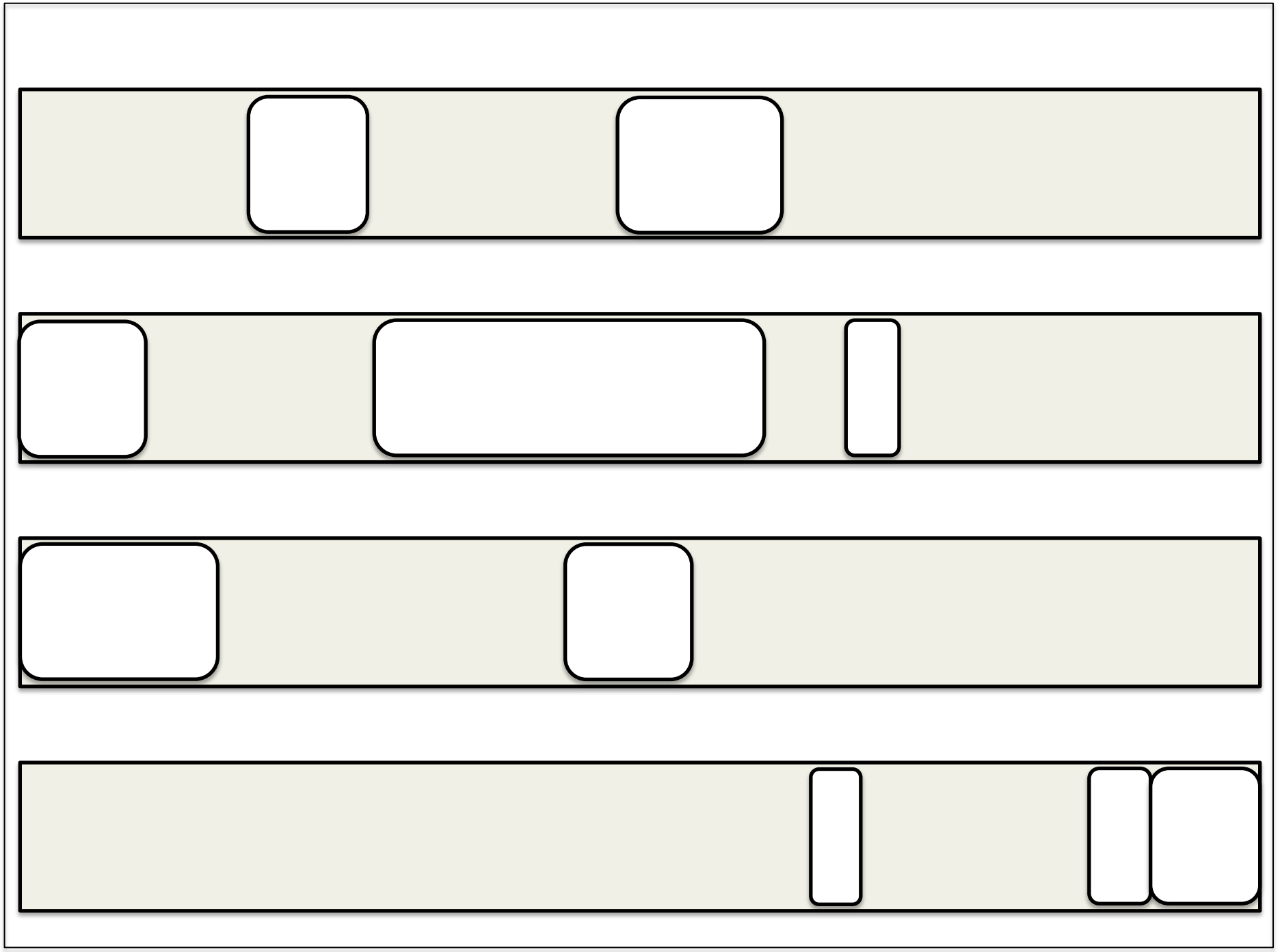
Objects and Holes

- Non-moving collectors create fragmentation
  - Solution is to compact the heap
- Over time fragmentation gets worse
  - Need to be able to fill in the gaps between objects
  - Recall the various allocation algorithms
- It can be useful to think of holes as a separate item
  - List of free locations to allocate objects





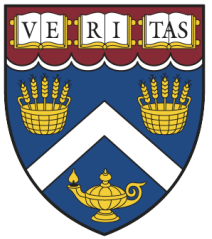


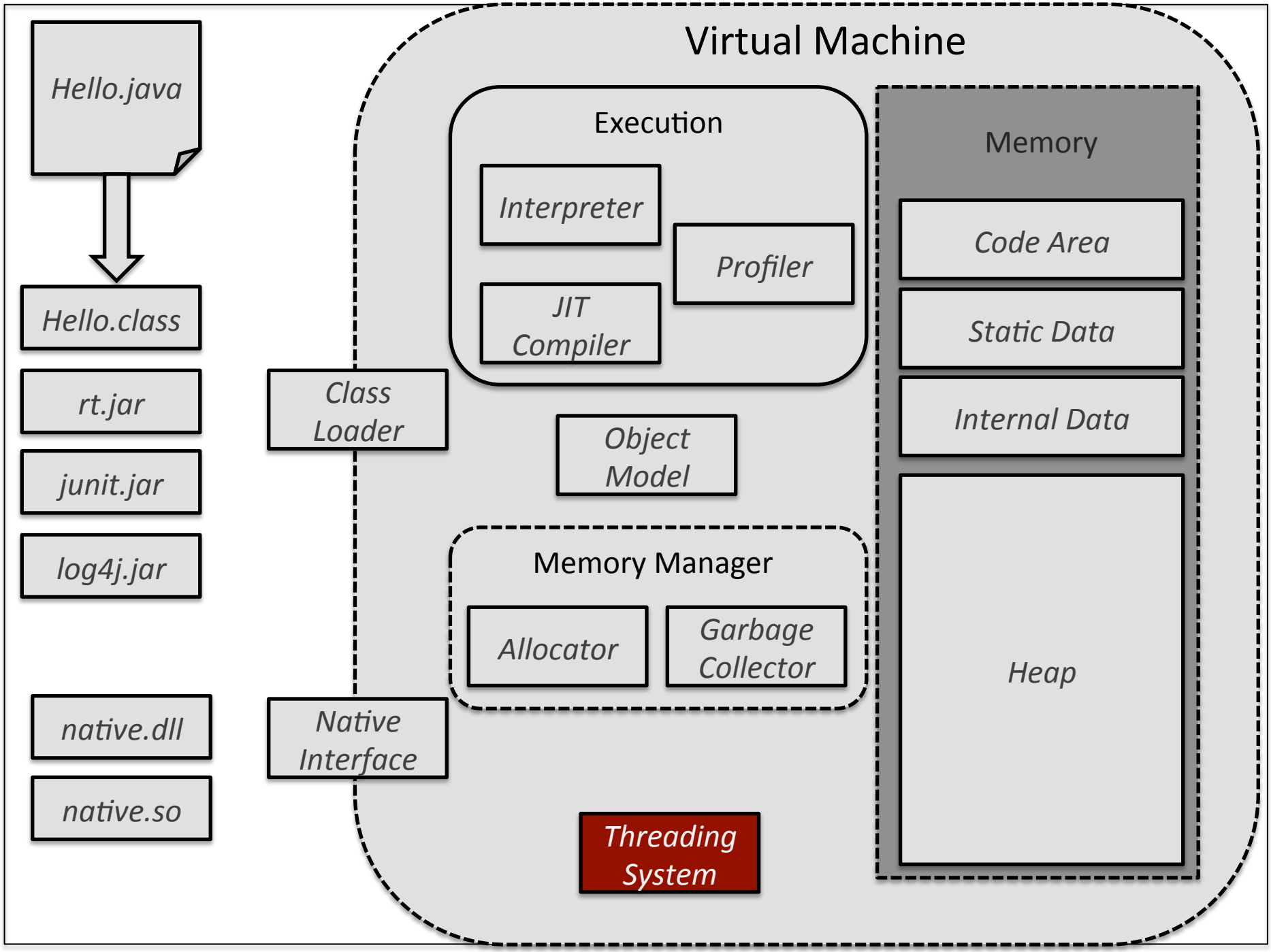


Tracking Space

- It is not necessary to track holes separately
  - You can just track objects allocated on the heap
  - The holes can be derived from the objects
- Allocation is much faster with a free list
  - Just scan through the list to find first fit
  - How real-world allocators are implemented
- Maintaining list of holes adds complication

VM Threads




Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Memory Fences

- Processor-level synchronization instruction
  - One of the most expensive in the ISA
  - Most processors have finer-grained instructions
- Guarantees memory ordering before and after
  - Use fences to synchronize on monitor operations
- Flushes cached versions of variables

Volatile

- `volatile` variables are never cached locally
  - All writes conceptually go to main memory
  - All threads see the same value immediately
- Useful as lightweight synchronization
  - Fewer features than full monitors
  - May perform better than critical sections
- Some semantic traps for the unwary
  - Incrementing a variable in two threads really two actions

Concurrency and Parallelism

- Two terms often used synonymously
  - Useful to maintain a distinction
- Concurrent threads interact with others
  - Must synchronize between threads
- Parallel threads have limited or no interaction
  - Very little synchronization
  - Best performance advantage

Daemon Threads

- Threads in the VM fall into two groups
  - Daemon threads normally belong to the VM
  - Non-daemon threads belong to the application
  - Applications can turn threads into daemons
- Daemon threads don't normally terminate
  - The VM runs until all non-daemon threads end
  - Daemon threads abandoned at shut-down
  - No cleanup operations happen

Internal Daemon Threads

- Periodic task thread
  - Implements scheduled events and timer interrupts
- Signal dispatcher thread
  - Monitors for OS-level signals
- Compilation and garbage collection
- `VMThread`
  - Main dispatcher for system operations

VMThread

- Single instance in a VM
  - Runs background tasks separately from Java threads
- VM tasks are added to a synchronized queue
  - `VMThread` waits for a change to the queue
  - Does safepointing based on the operation
- Single VM thread simplifies implementation
  - Don't need to worry about concurrency

VMThread

- Around fifty possible tasks
  - Some heavyweight VM operations
  - Many of them handlers for rare cases
- Four modes for tasks to run
  - Safepoint/no safepoint
  - Blocking/non blocking

VMThread Operations

- Logging and reporting
- Verification
- Deoptimize code
- Redefine classes
- Compute stack traces
- Revoke biased locks

Compiler Threads

- Compilation can happen in parallel
  - Doesn't block the Java threads
  - Interpreter can continue to execute a method
- Most code replacement doesn't require a safepoint
  - Use the compiled code for the next execution
- On Stack Replacement is a heavier operation
  - Task scheduled for the VMThread

Memory Model and JIT Compiler

- Java concurrency has implications for JIT
  - Limits code reordering
  - Limits memory prefetching
- `final` fields treated specially
  - Initial write in constructor comes before object is shared
  - Could be reordered in inlined code
  - Afterwards, don't have to reload the final field

Profiler Sampling

- Some VMs use a thread for their profiler
  - Periodic thread samples stack state
  - Builds up map of hot methods
- Lower overhead than measuring every call
  - Background thread doesn't block Java thread
- Results can be less accurate
  - More likely to miss short methods

Multi-Thread Memory Management

- Garbage collectors take advantage of threads
  - Can run far faster than single-threaded GC
  - Implementation is more complex
  - Consider trade-off between throughput and pauses
- Allocation introduces contention
- GC can be split across multiple threads
  - Add more phases to the garbage collector algorithm
  - Threads do different work at different points

Parallel and Concurrent GC

- Significant distinction in garbage collection
- Parallel GC splits the collection between threads
  - Still stop all running Java threads
- Concurrent GC interleaves with the mutator
  - Implementation can be very complex
- Modern garbage collectors are both
  - Concurrent phases interact with mutators
  - Parallel phases speed up critical sections

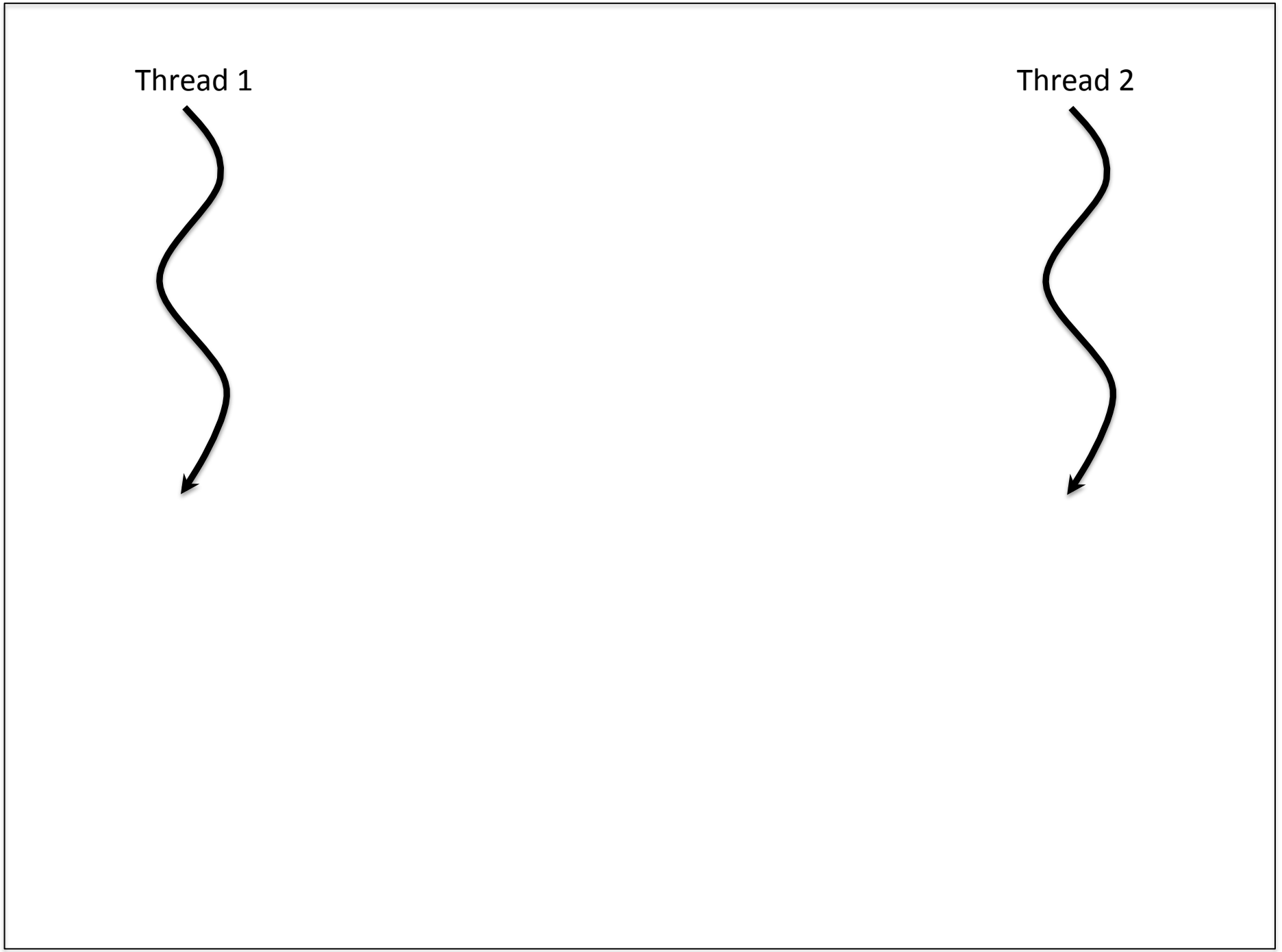
Parallel GC Algorithms

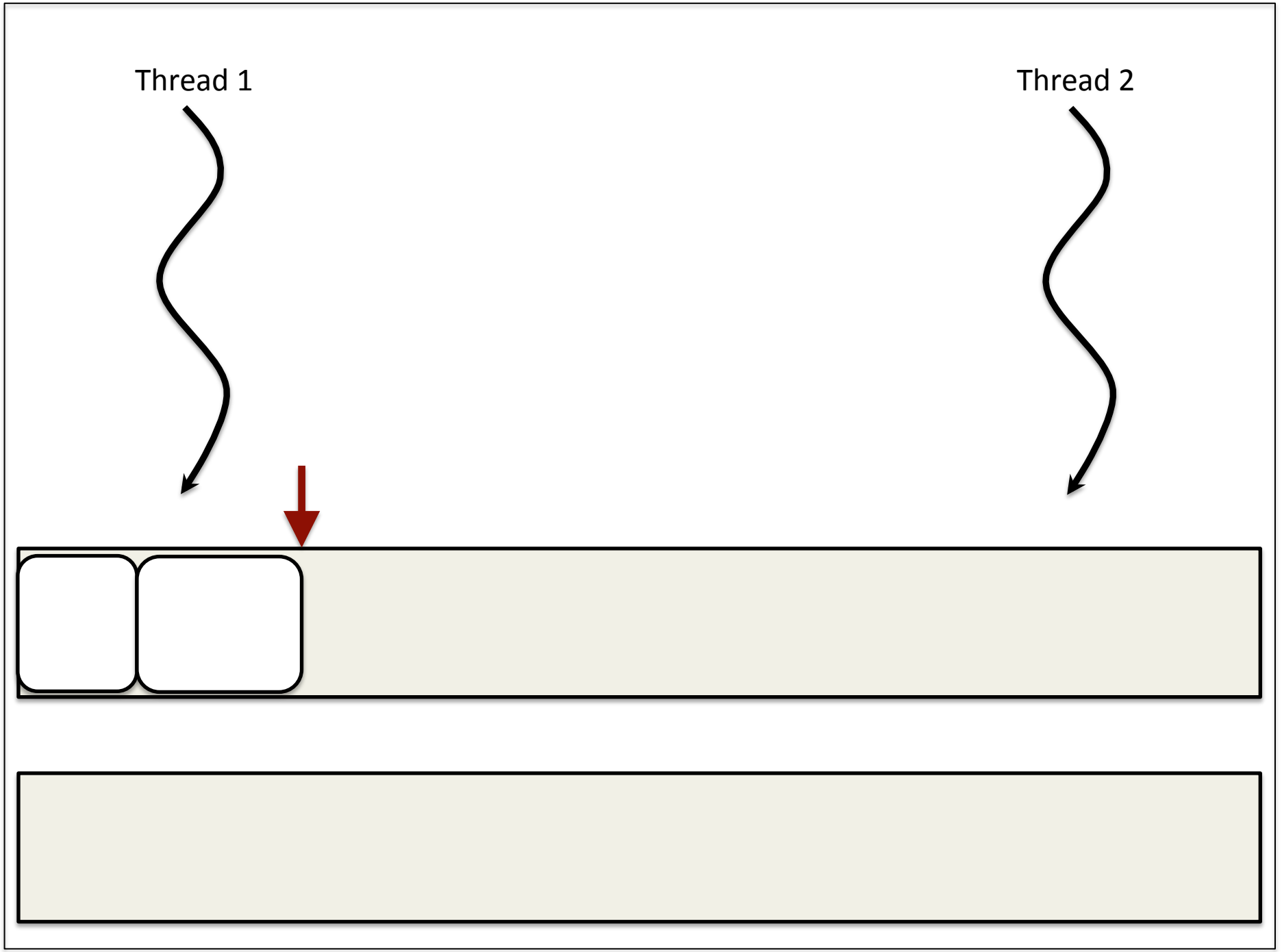
- Most effective when there is little coordination
  - Allow majority of work to happen independently
  - Scalability limited by the number of threads
- Many algorithms use work stealing
  - Each thread maintains a list of tasks
  - When a thread's list is empty, look to other thread's lists
- Work stealing can introduce contention
  - Work queues are a vital place for performance
  - Can minimize by stealing from the back of the queue

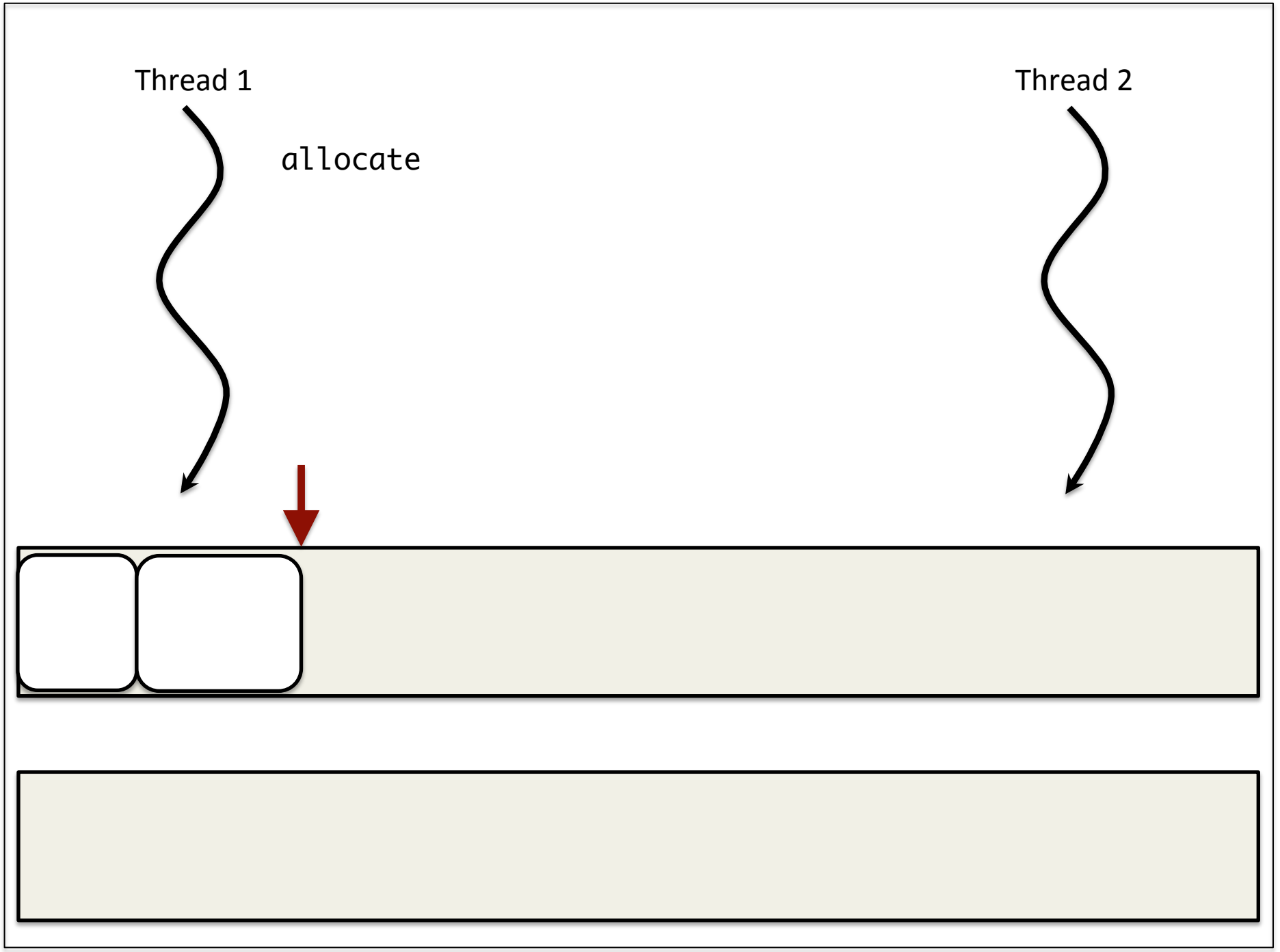
Allocation by Concurrent Threads

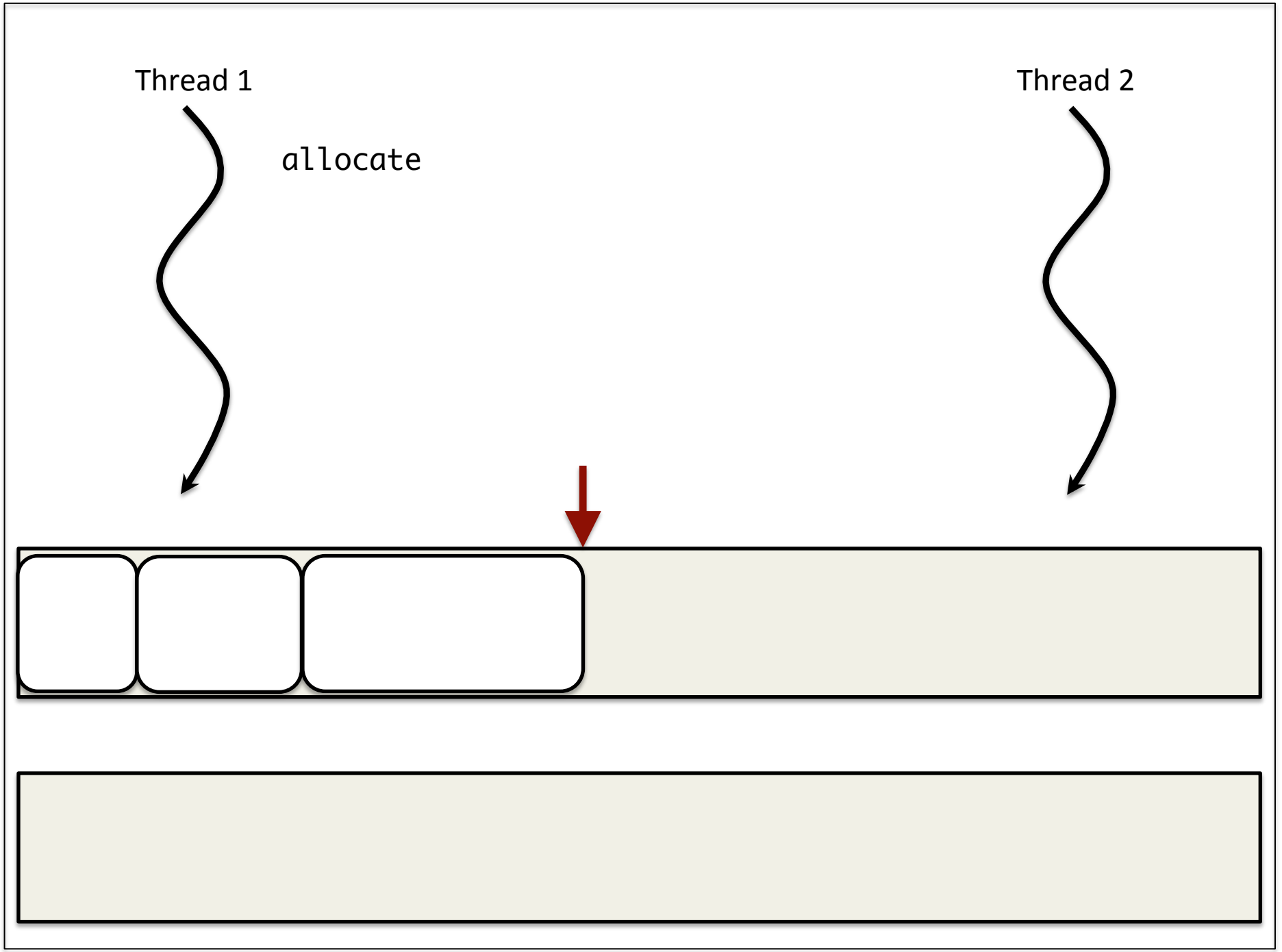
- So far allocation has happened in the nursery
  - Bump pointer algorithm

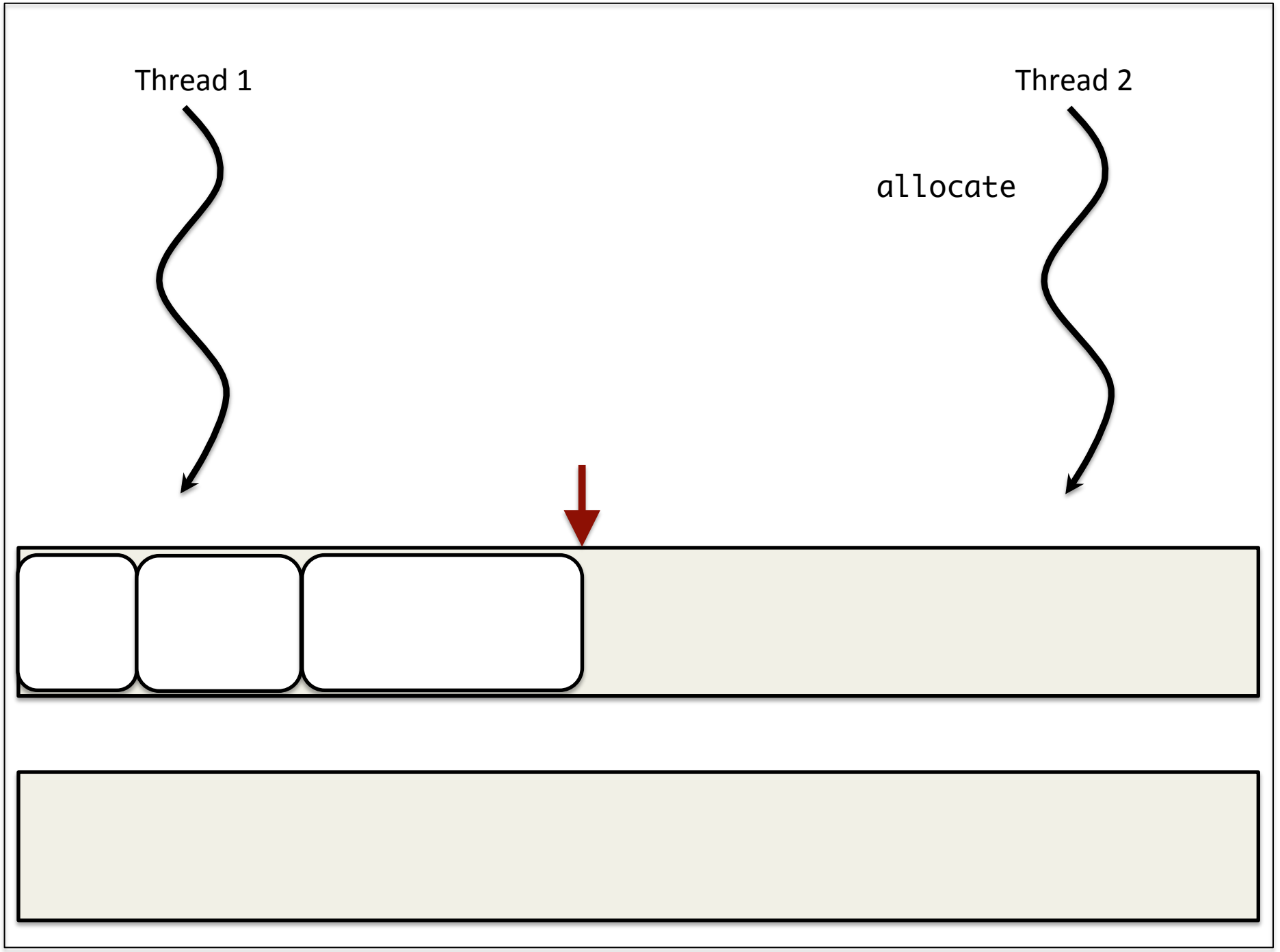
- Allocation is fast
  - Load the current allocation pointer
  - Add the size of the object to allocate
  - Compare with the limit pointer
  - Update the allocation pointer

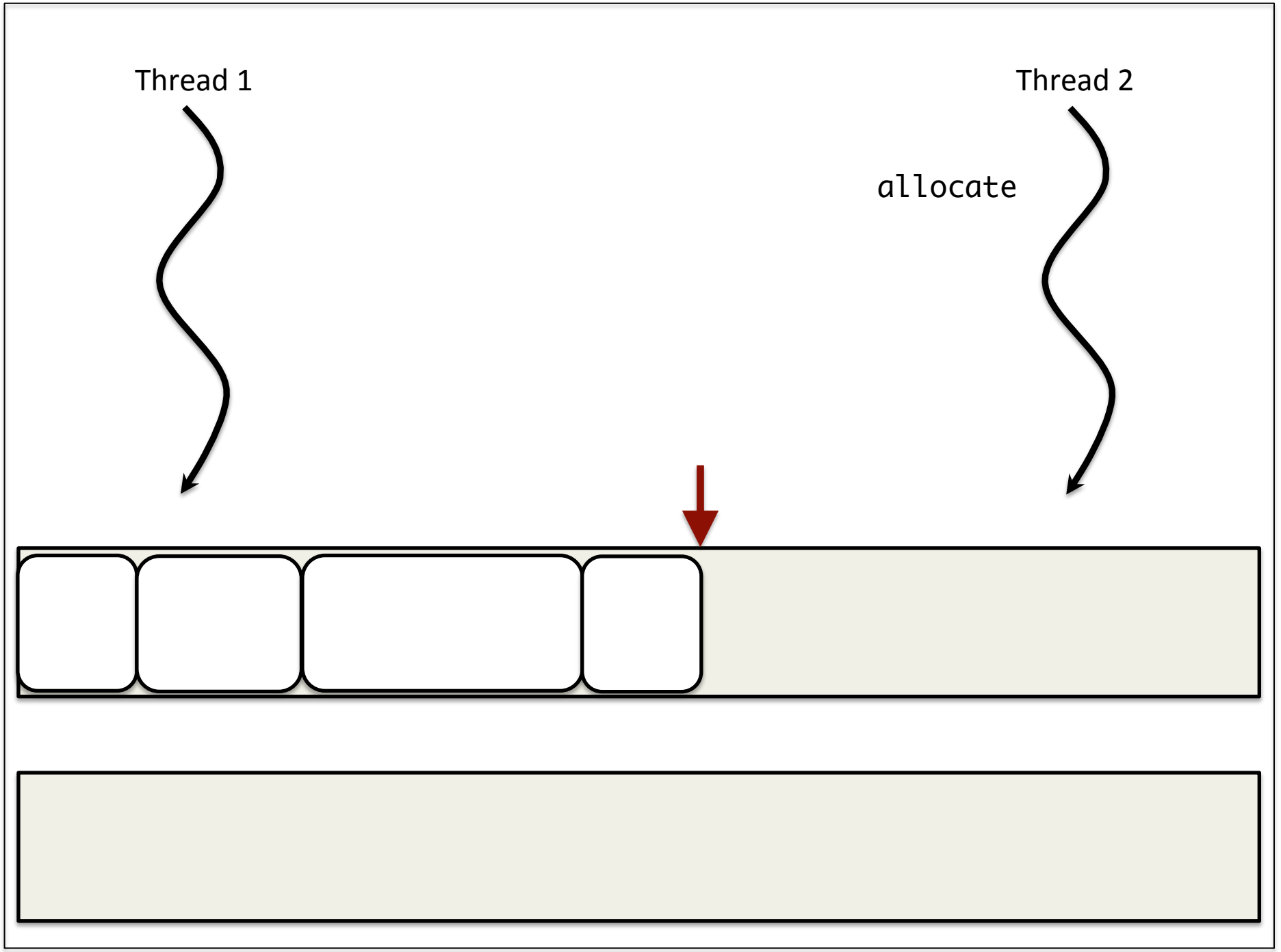

Thread 1
Thread 2

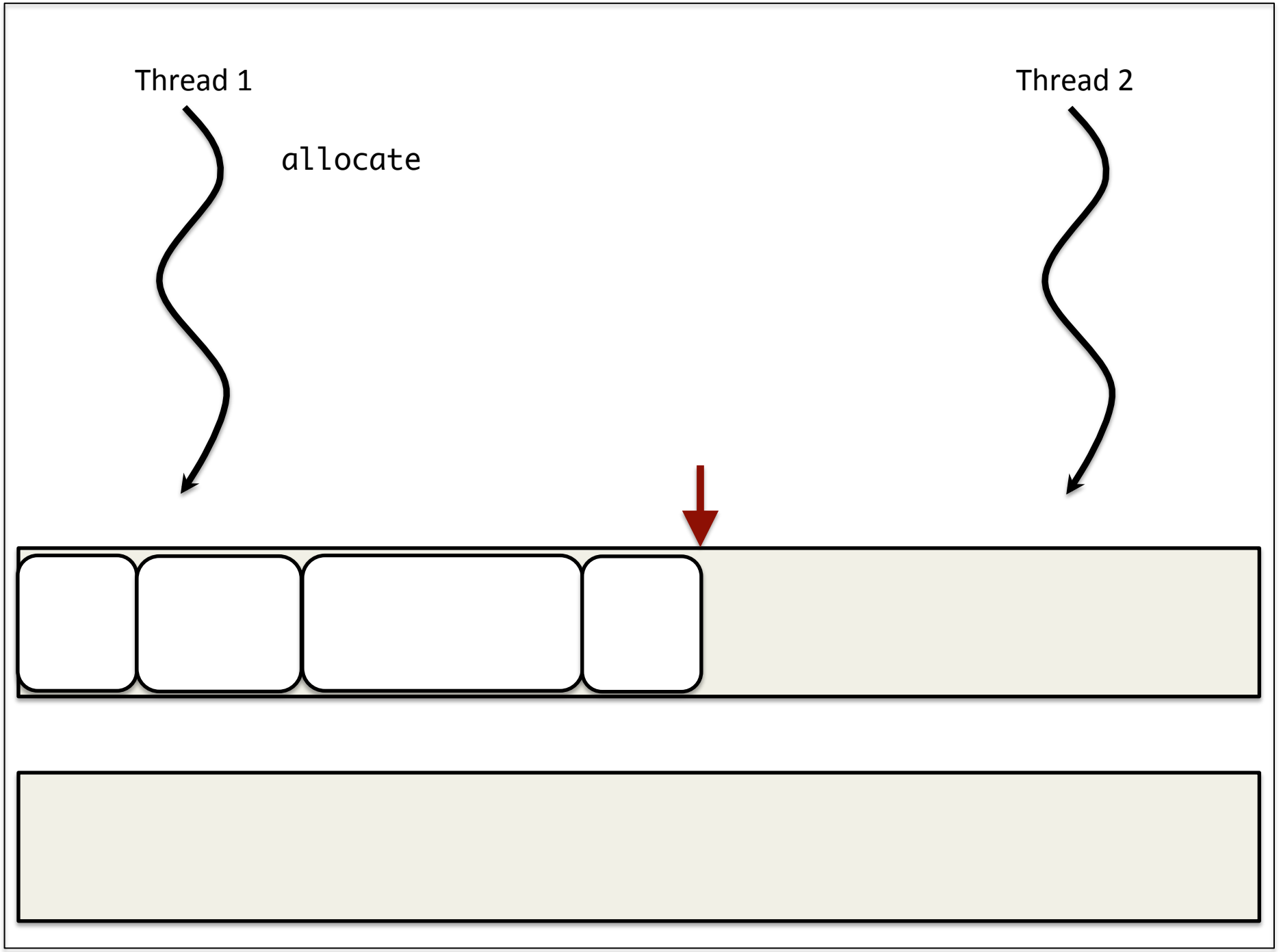

Thread 1
Thread 2

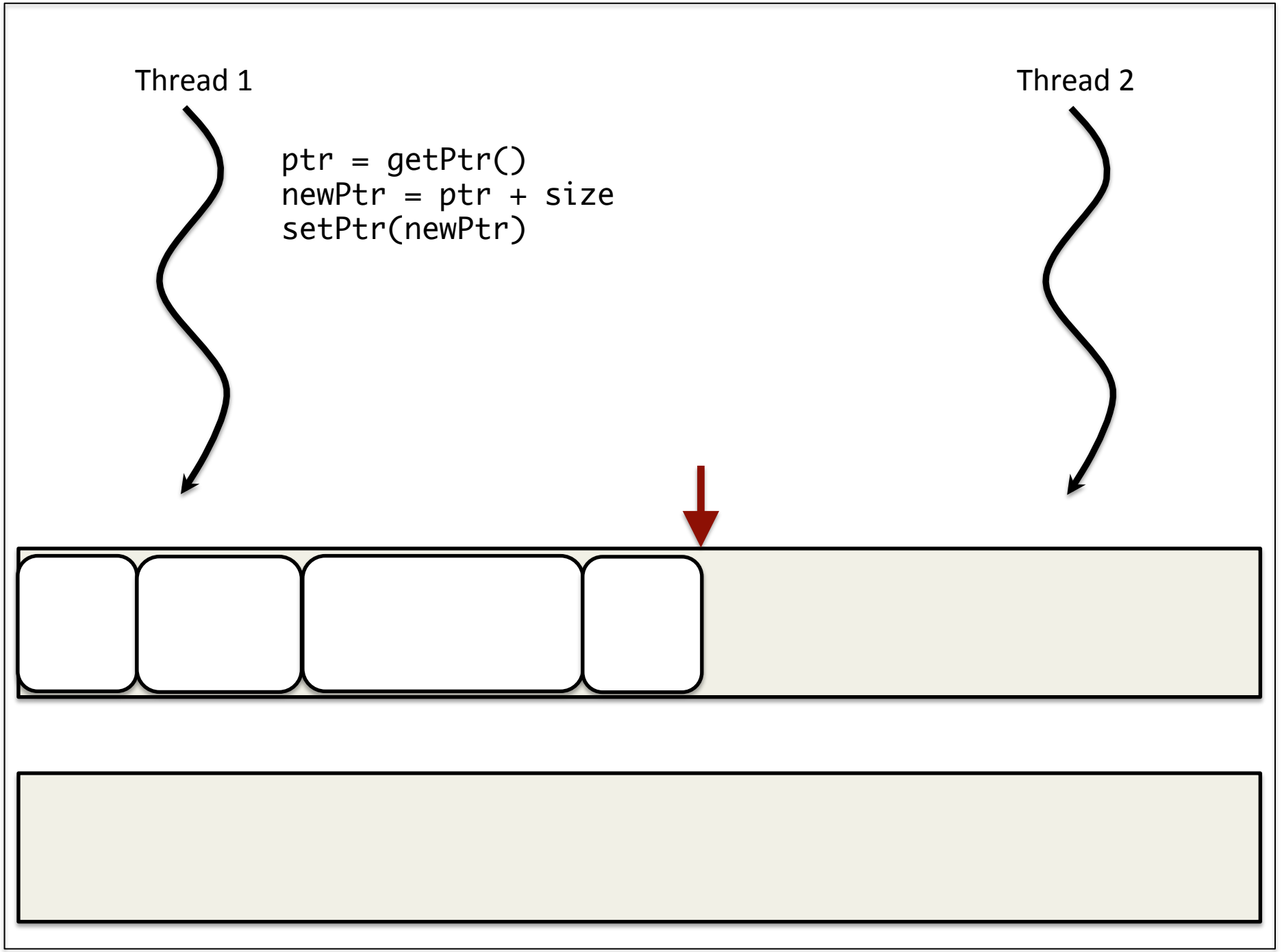

Thread 1
allocate
Thread 2

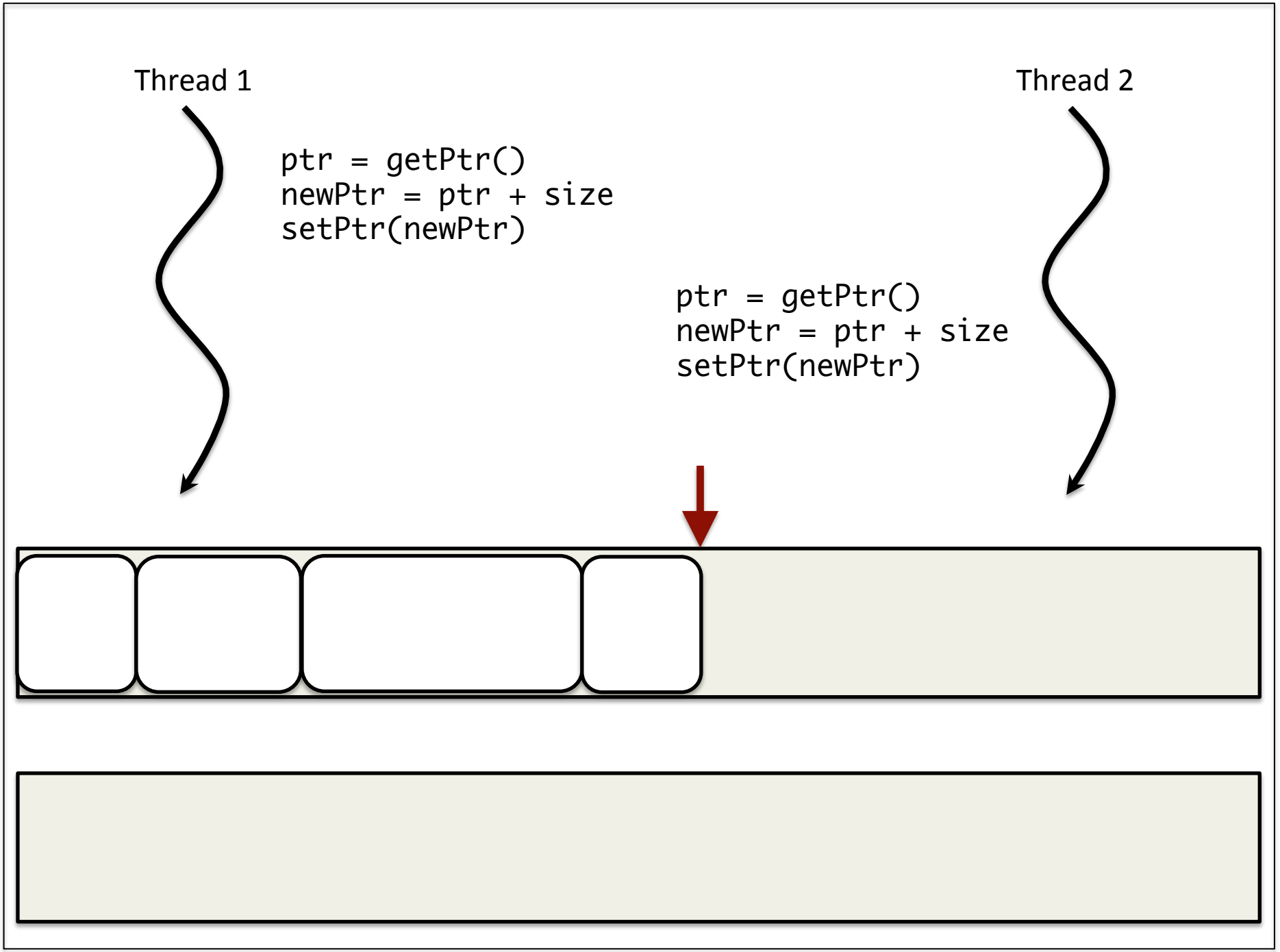

Thread 1
allocate
Thread 2

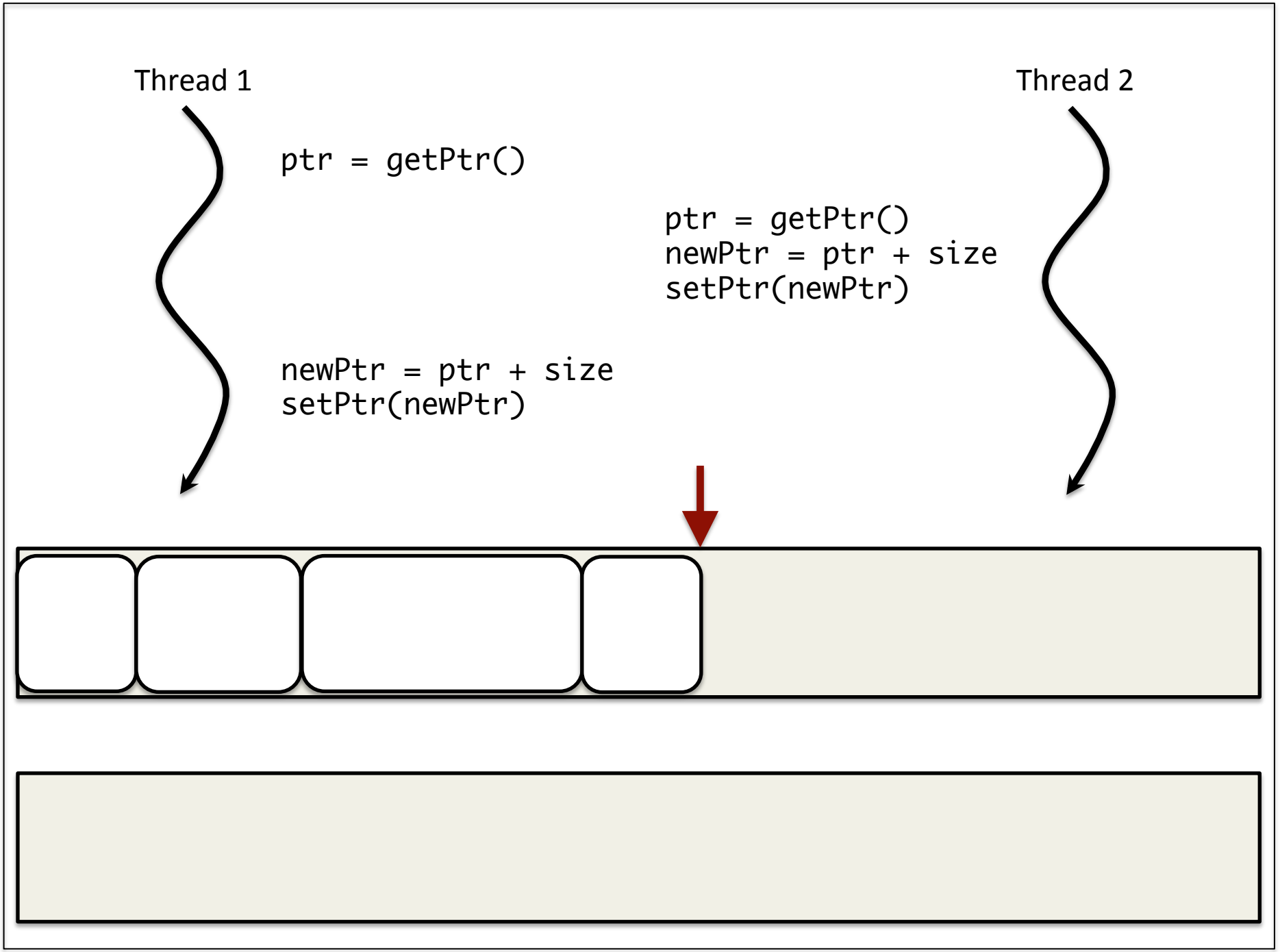

Thread 1
Thread 2
allocate

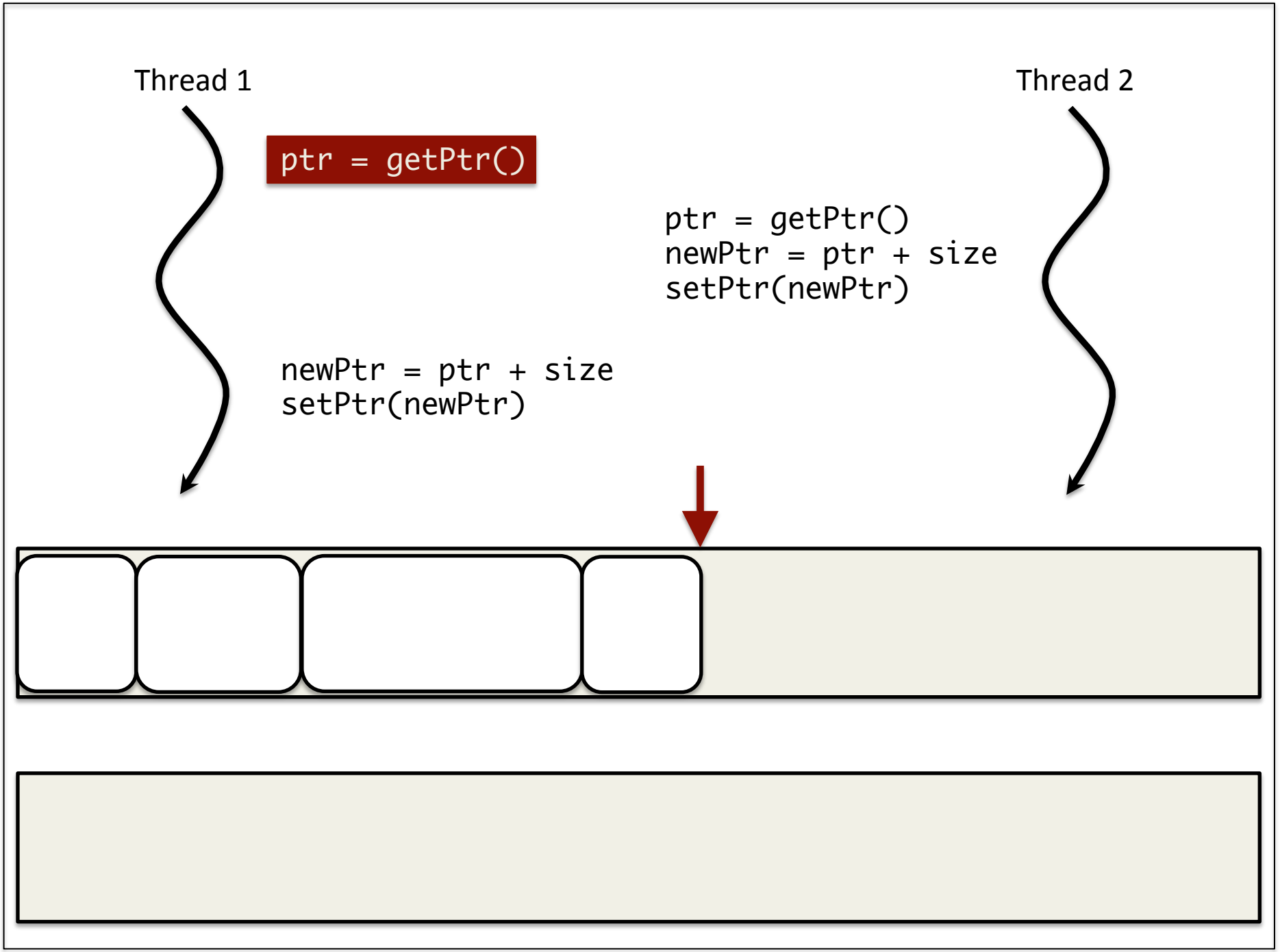

Thread 1
Thread 2
allocate

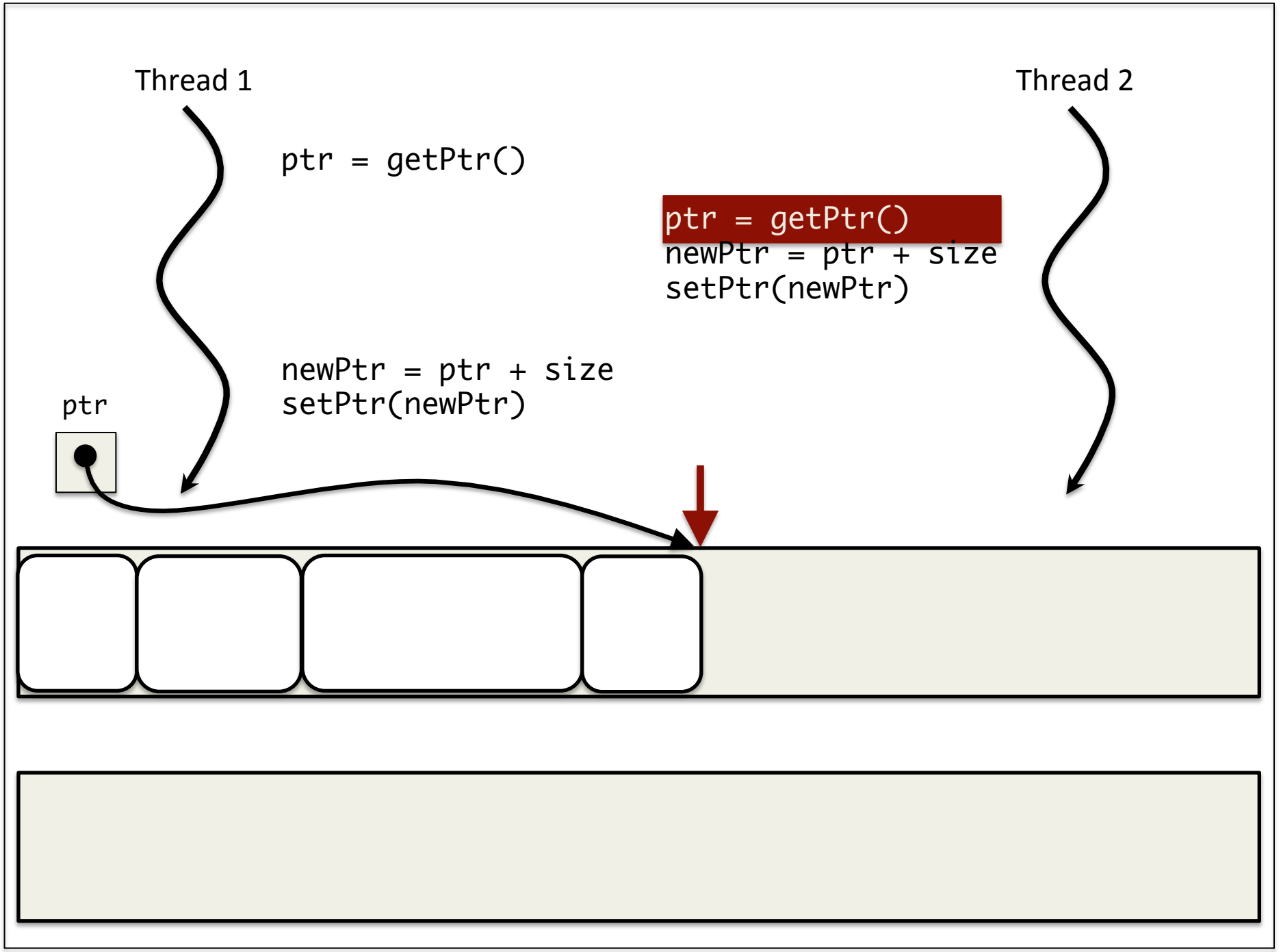

Thread 1
allocate
Thread 2

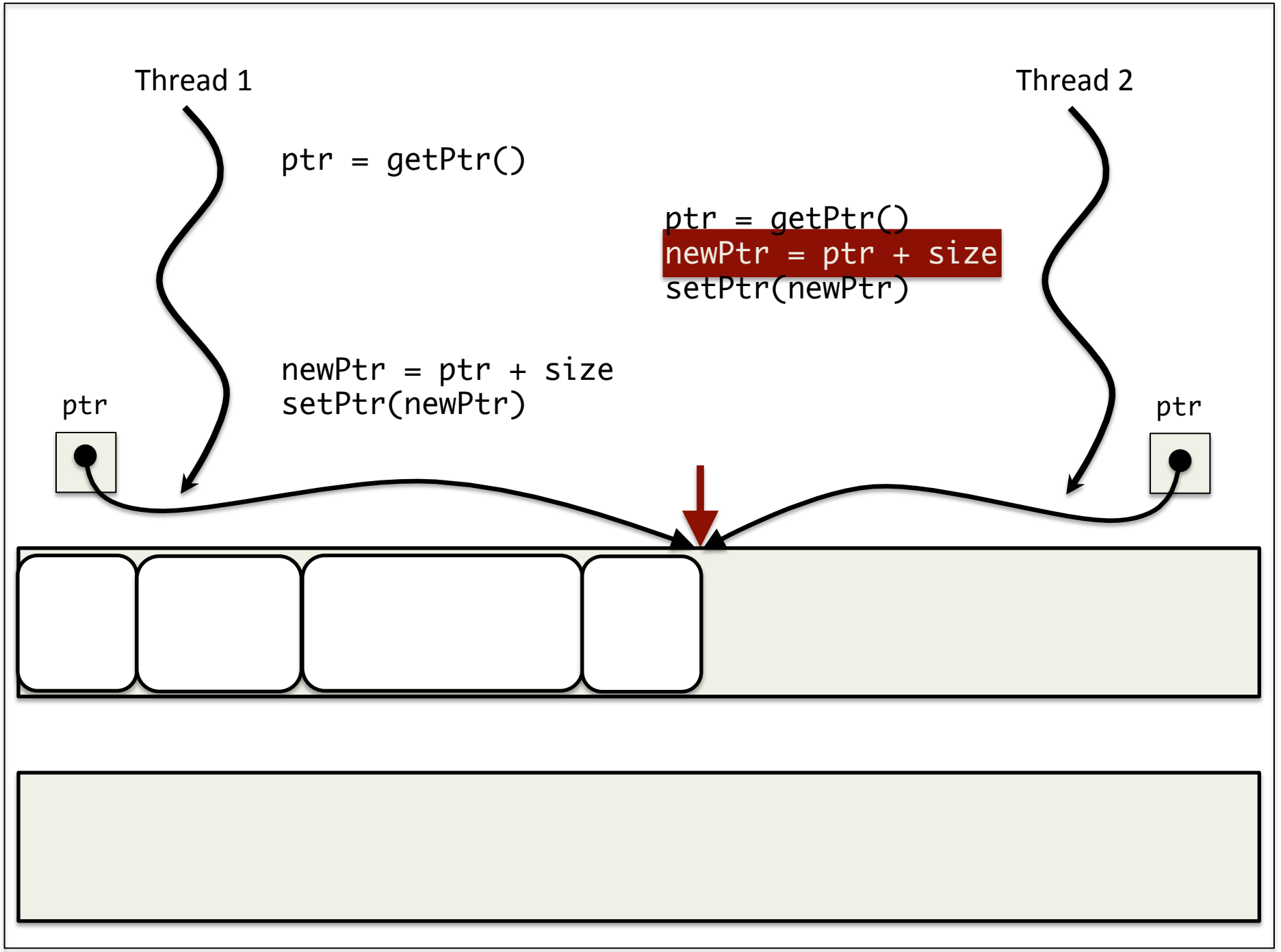

Thread 1
Thread 2
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)

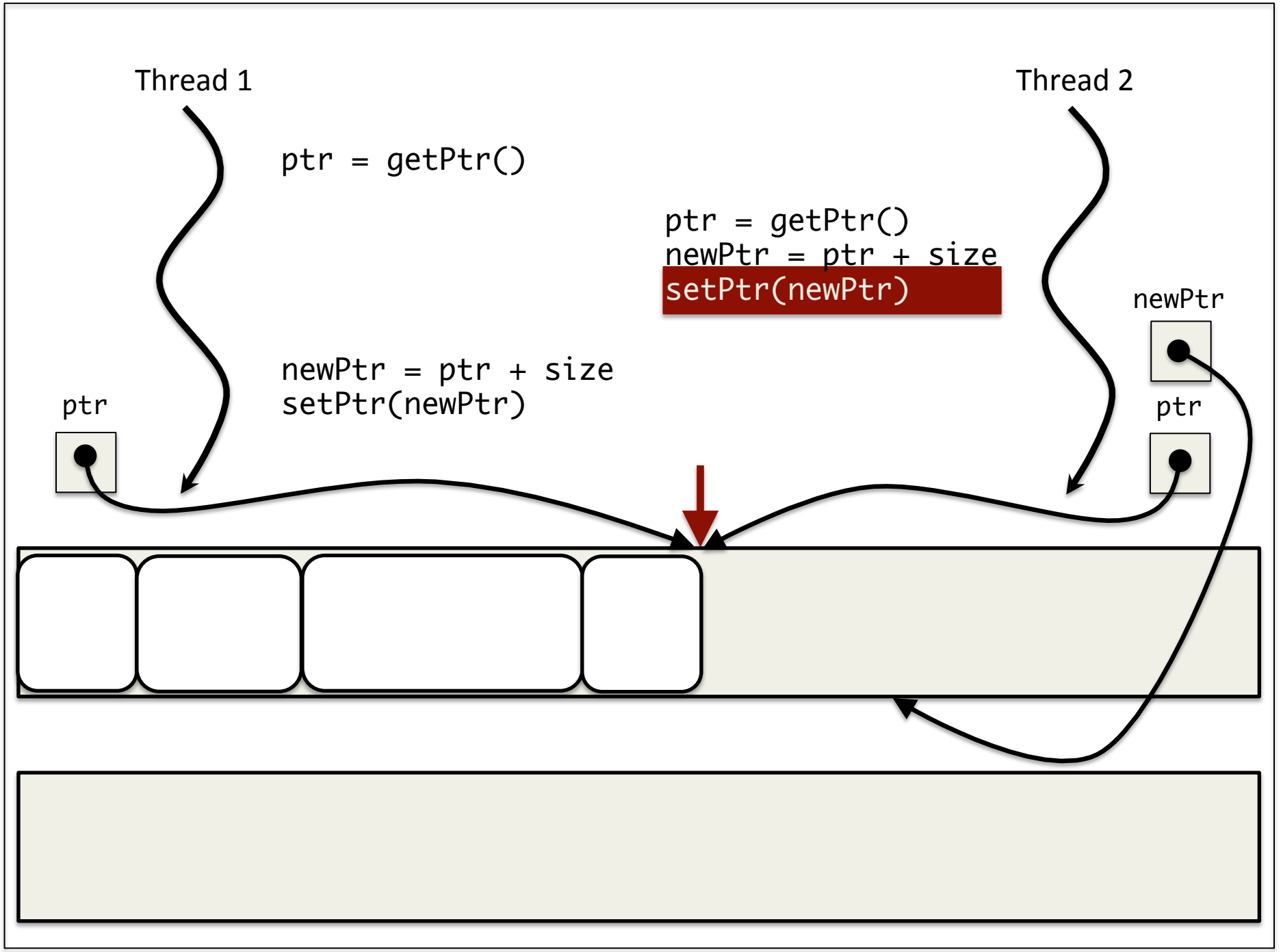

Thread 1
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
Thread 2
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)

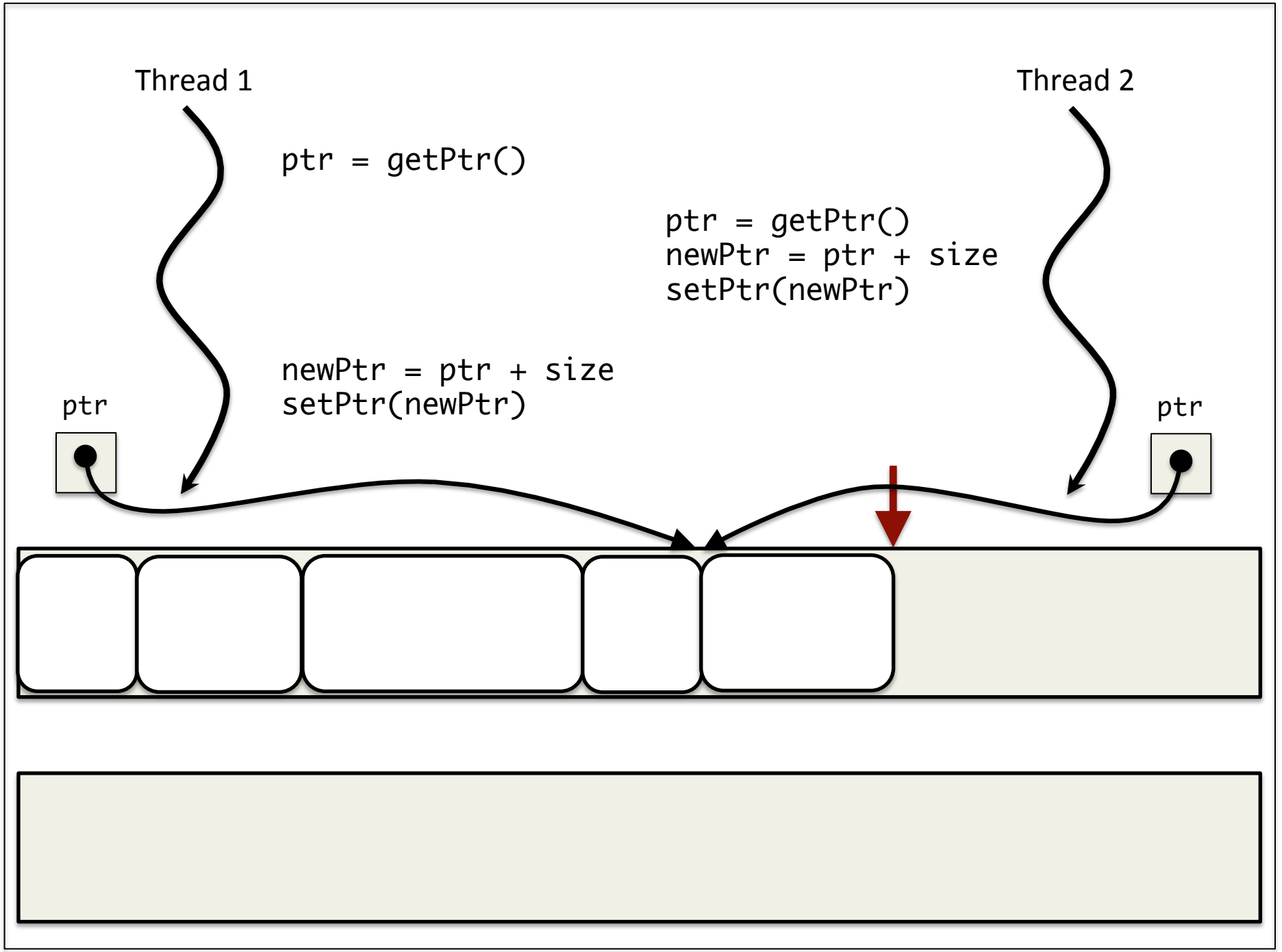

Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)

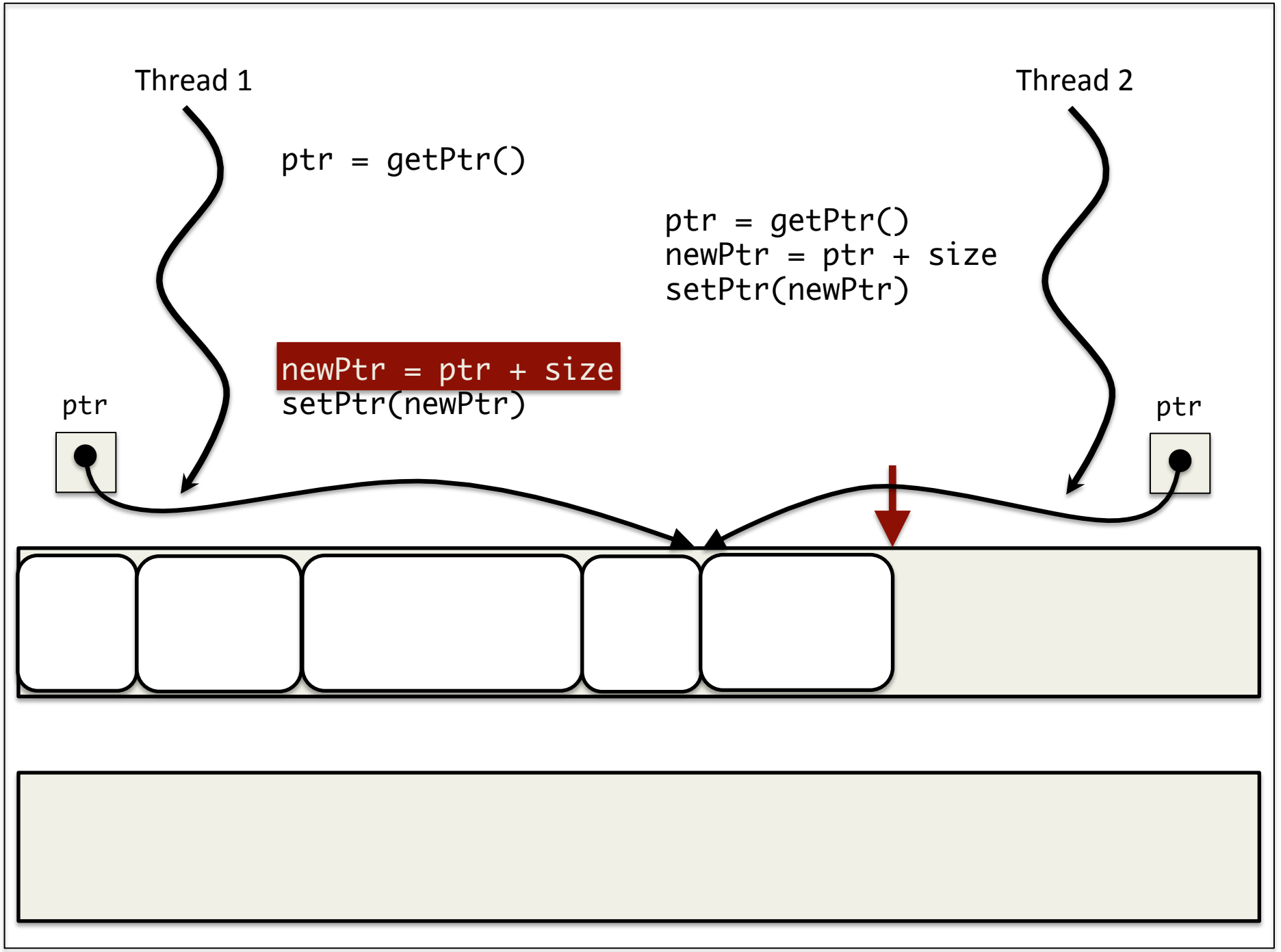

Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)

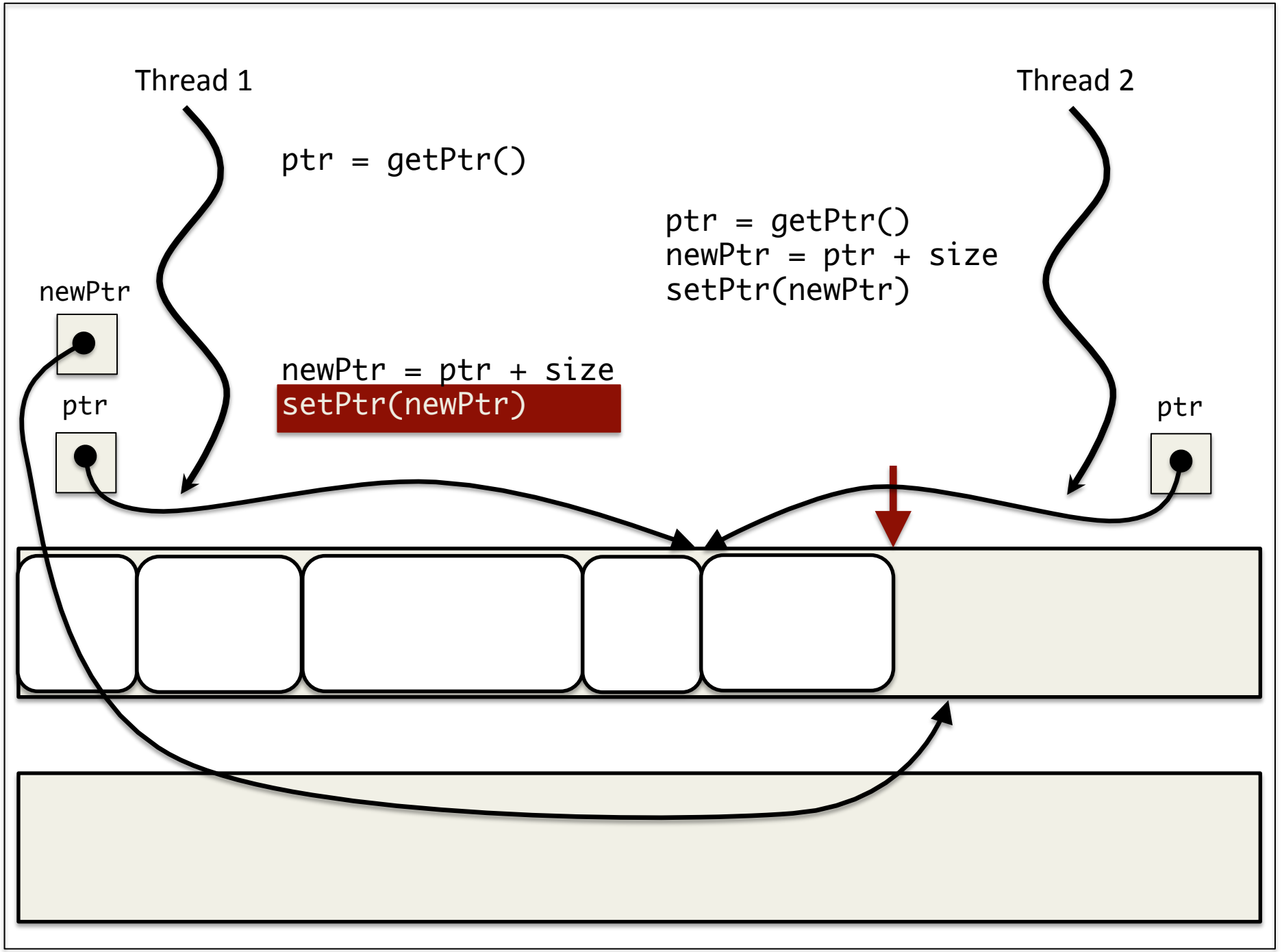

Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)
ptr

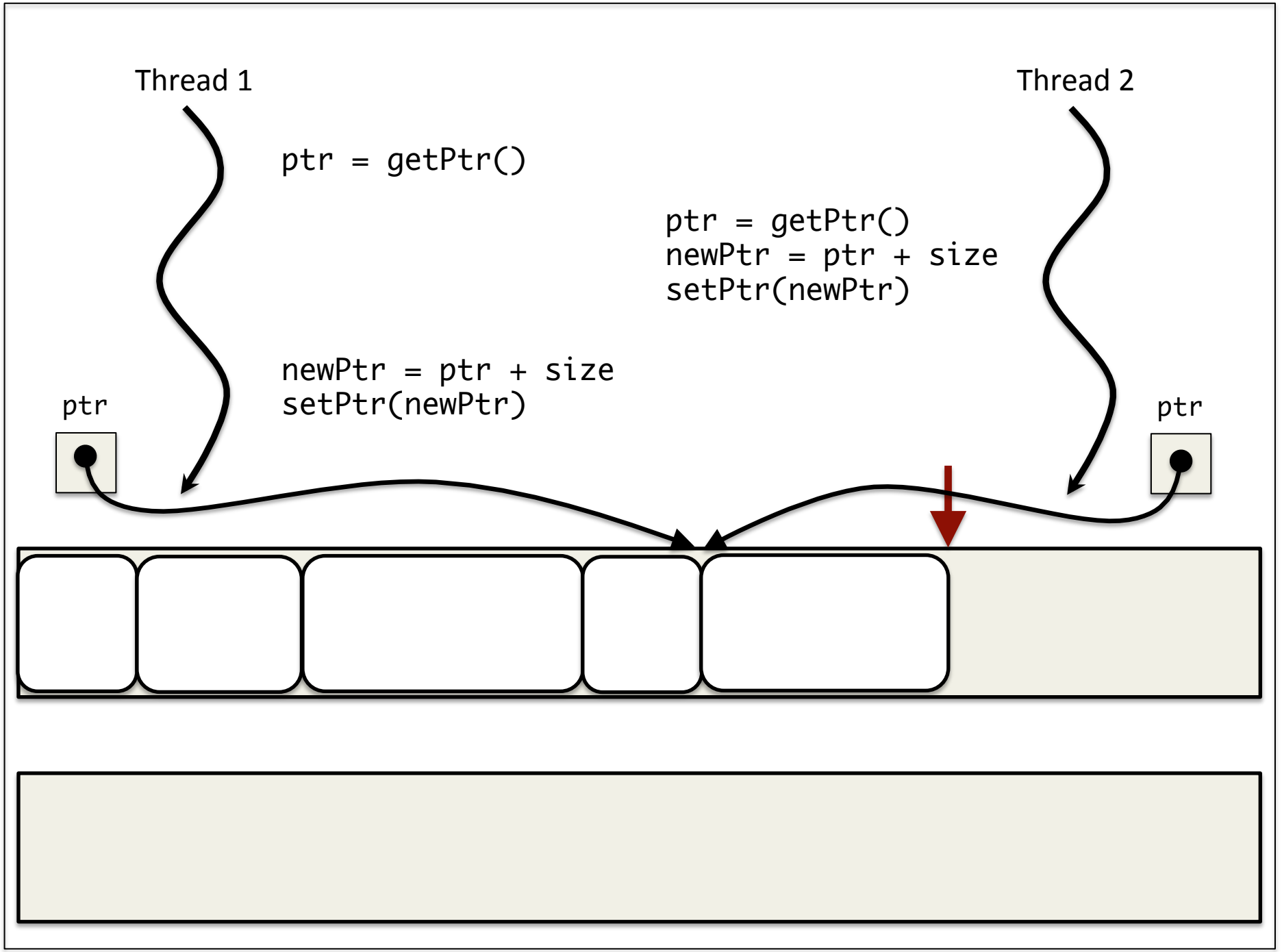

Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)
ptr
ptr


Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr
ptr
newPtr = ptr + size
setPtr(newPtr)
ptr


Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)
ptr
ptr


Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)
ptr
ptr

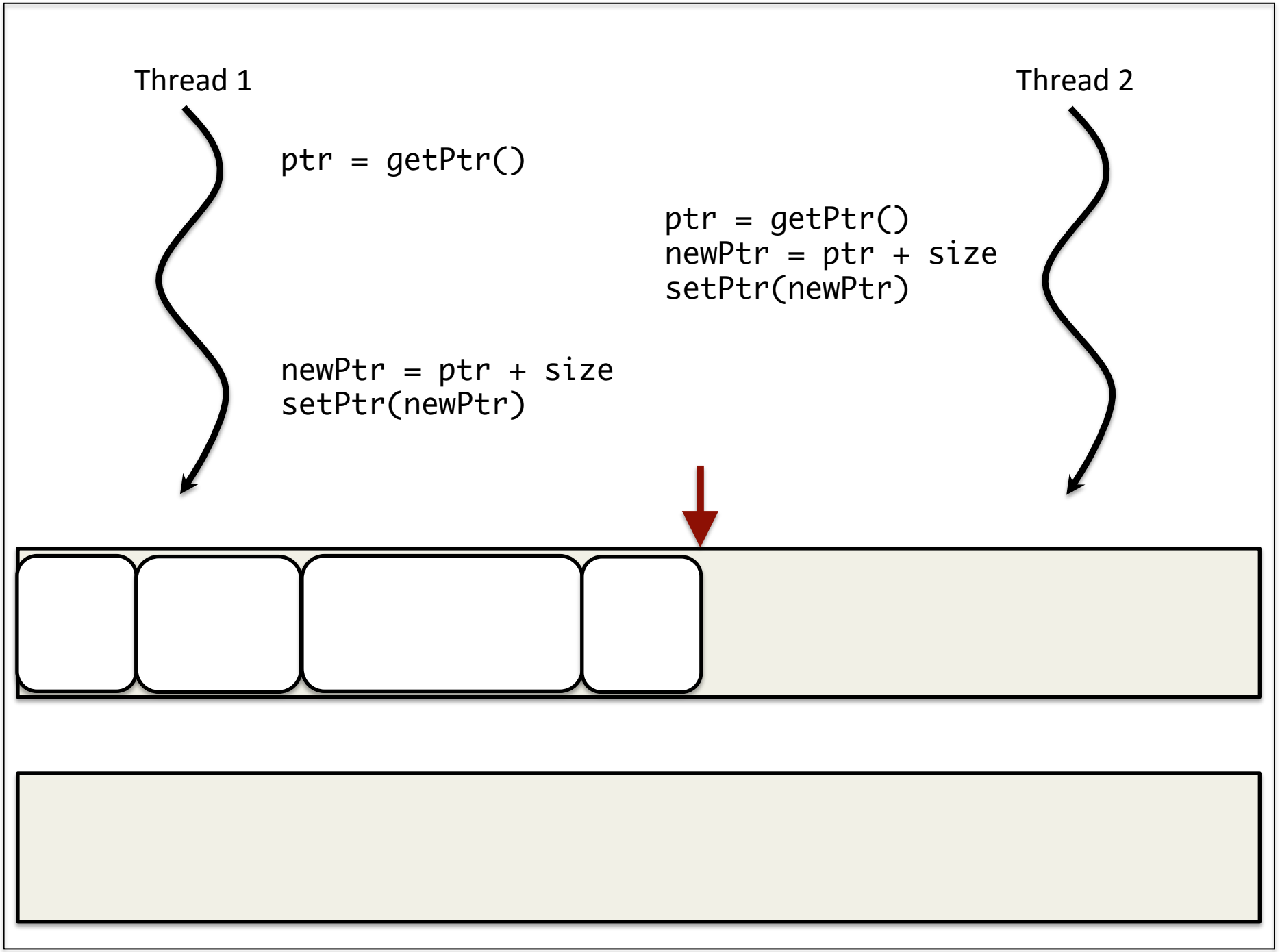

Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr
ptr
newPtr = ptr + size
setPtr(newPtr)
ptr

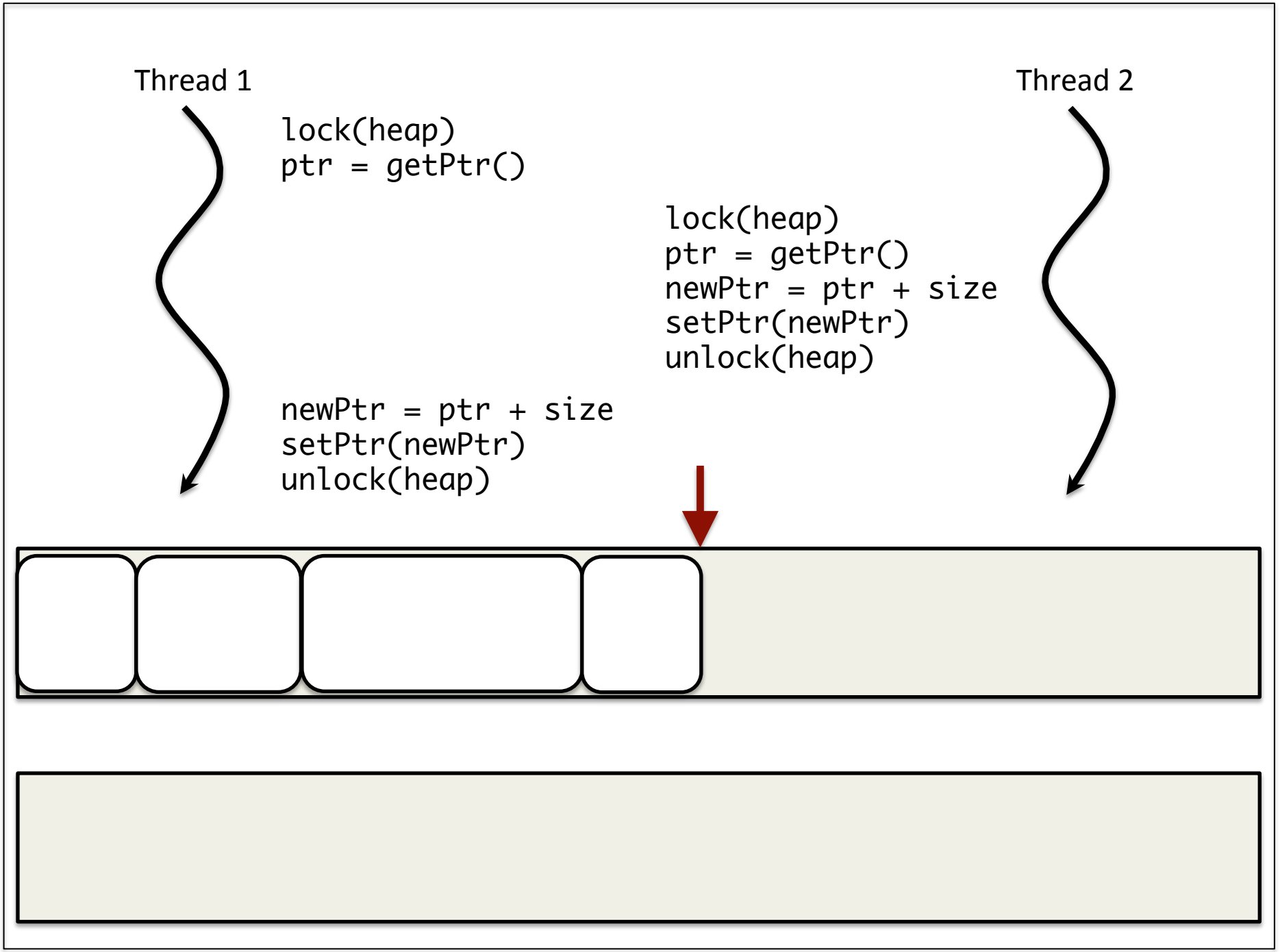

Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)
ptr
ptr

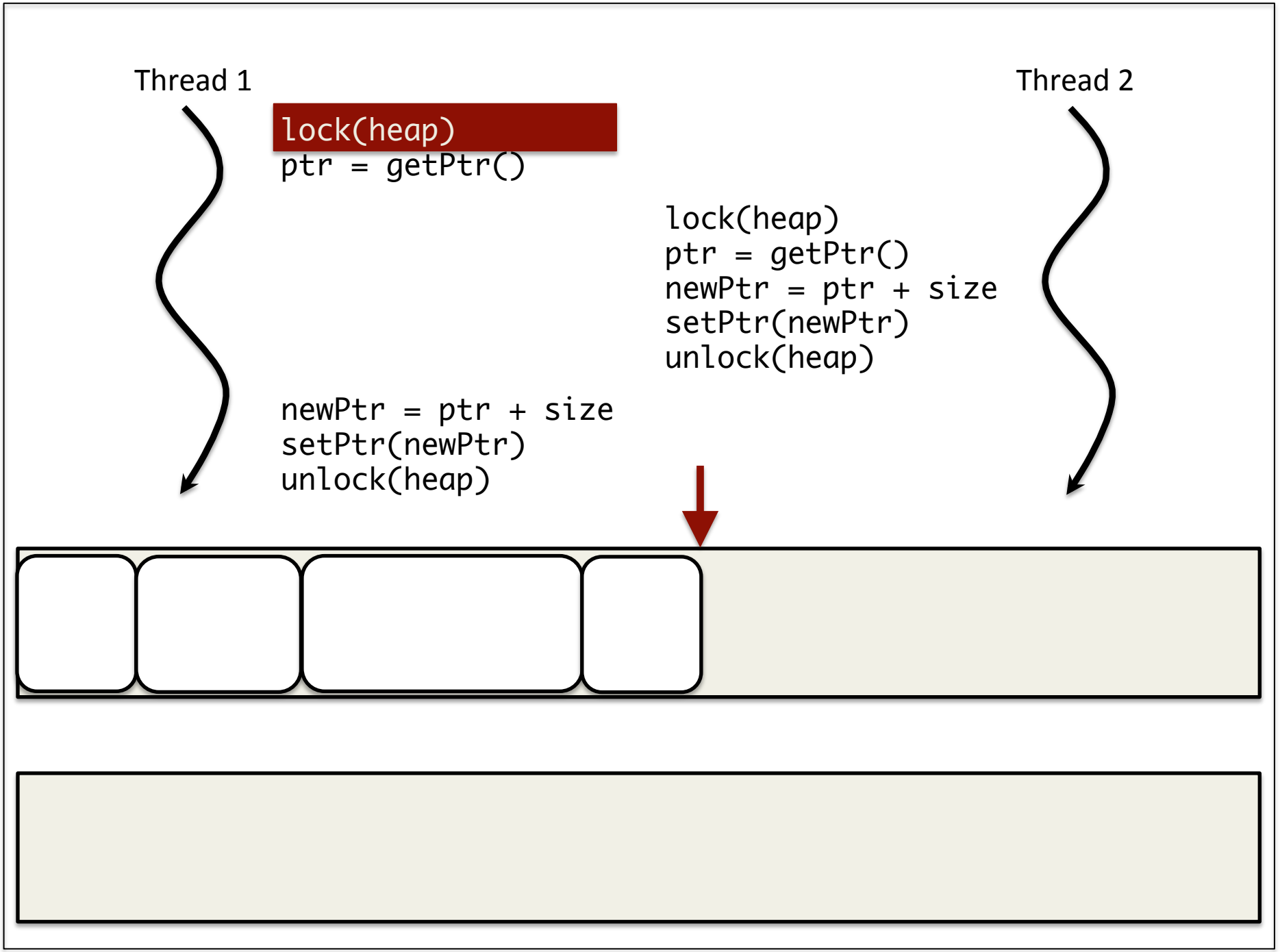
Solving Allocation Races

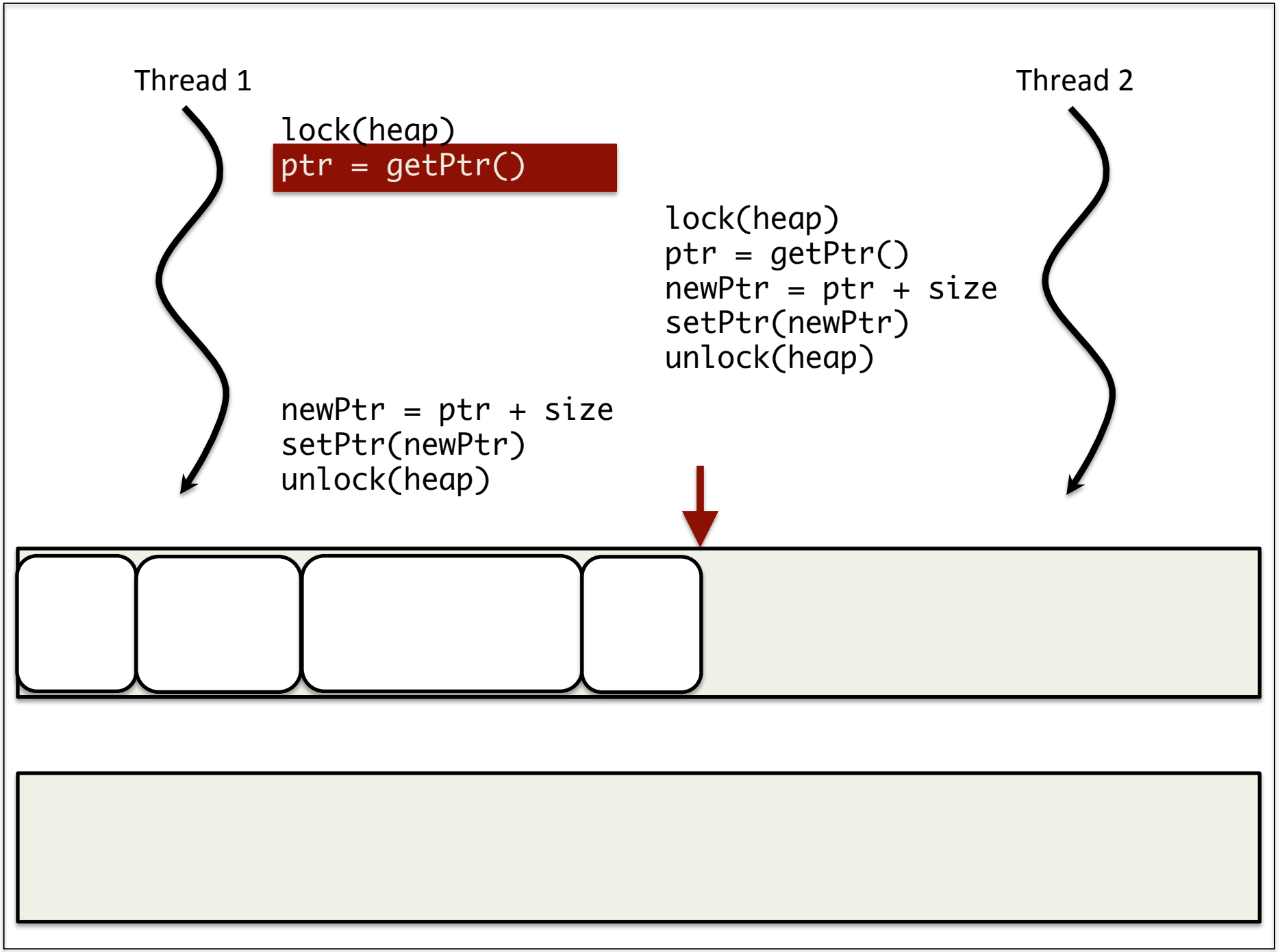
- We saw last week that this is a data race
  - Two threads accessing the same memory location
  - No synchronization in between

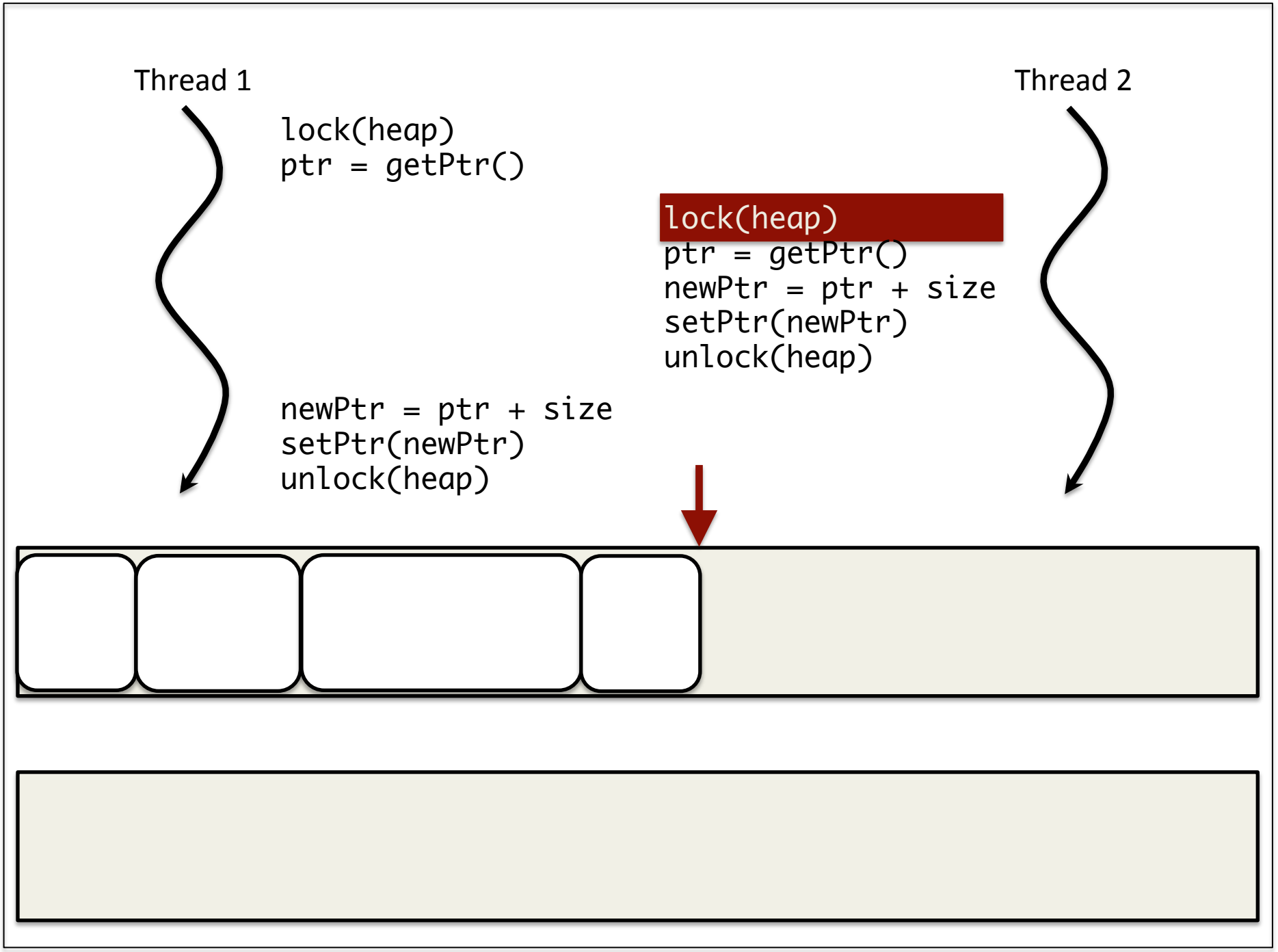
- Standard solution would be to synchronize
  - Acquire some shared lock before allocation
  - Use atomic instructions

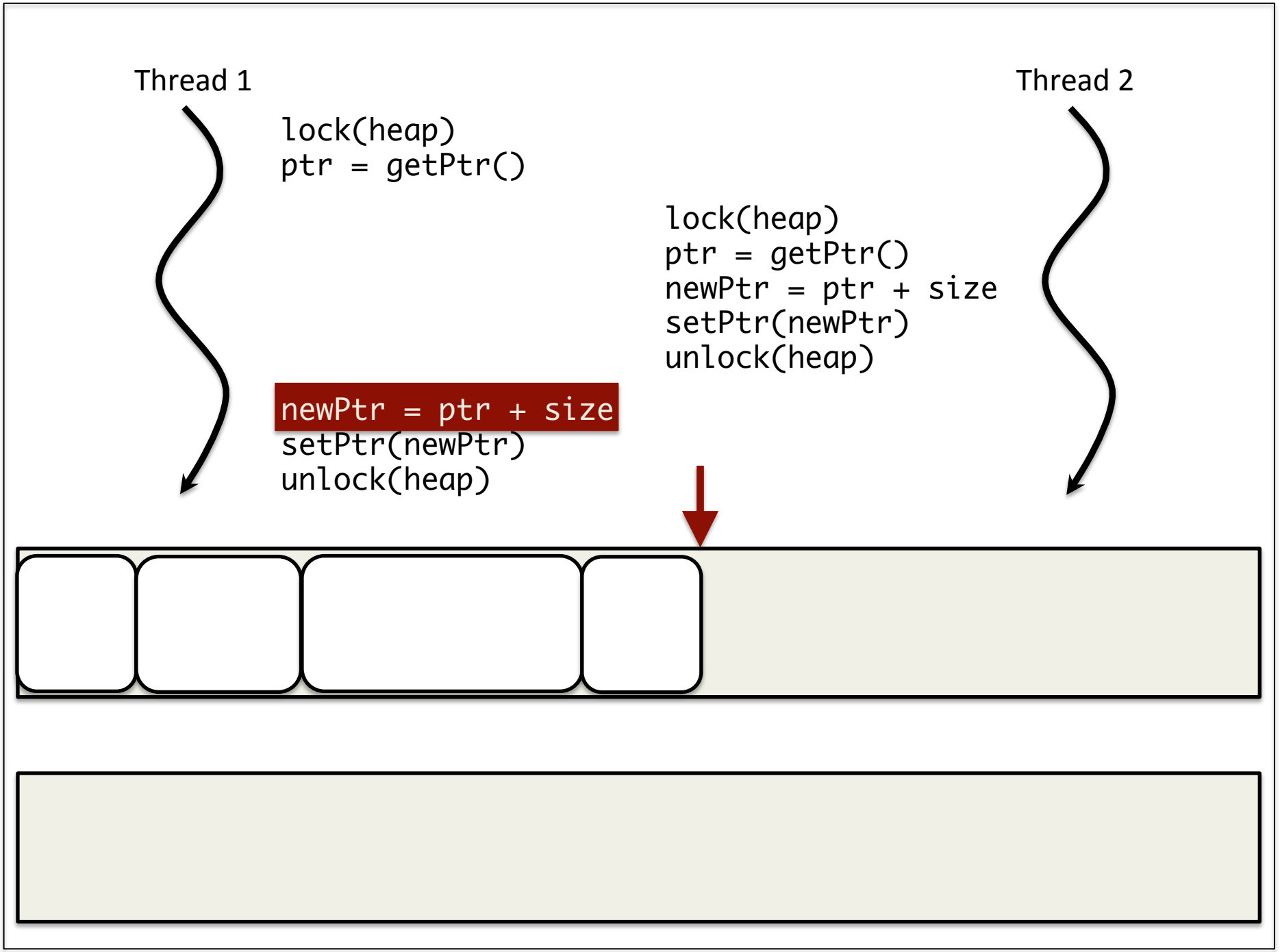

Thread 1
Thread 2
ptr = getPtr()
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
newPtr = ptr + size
setPtr(newPtr)

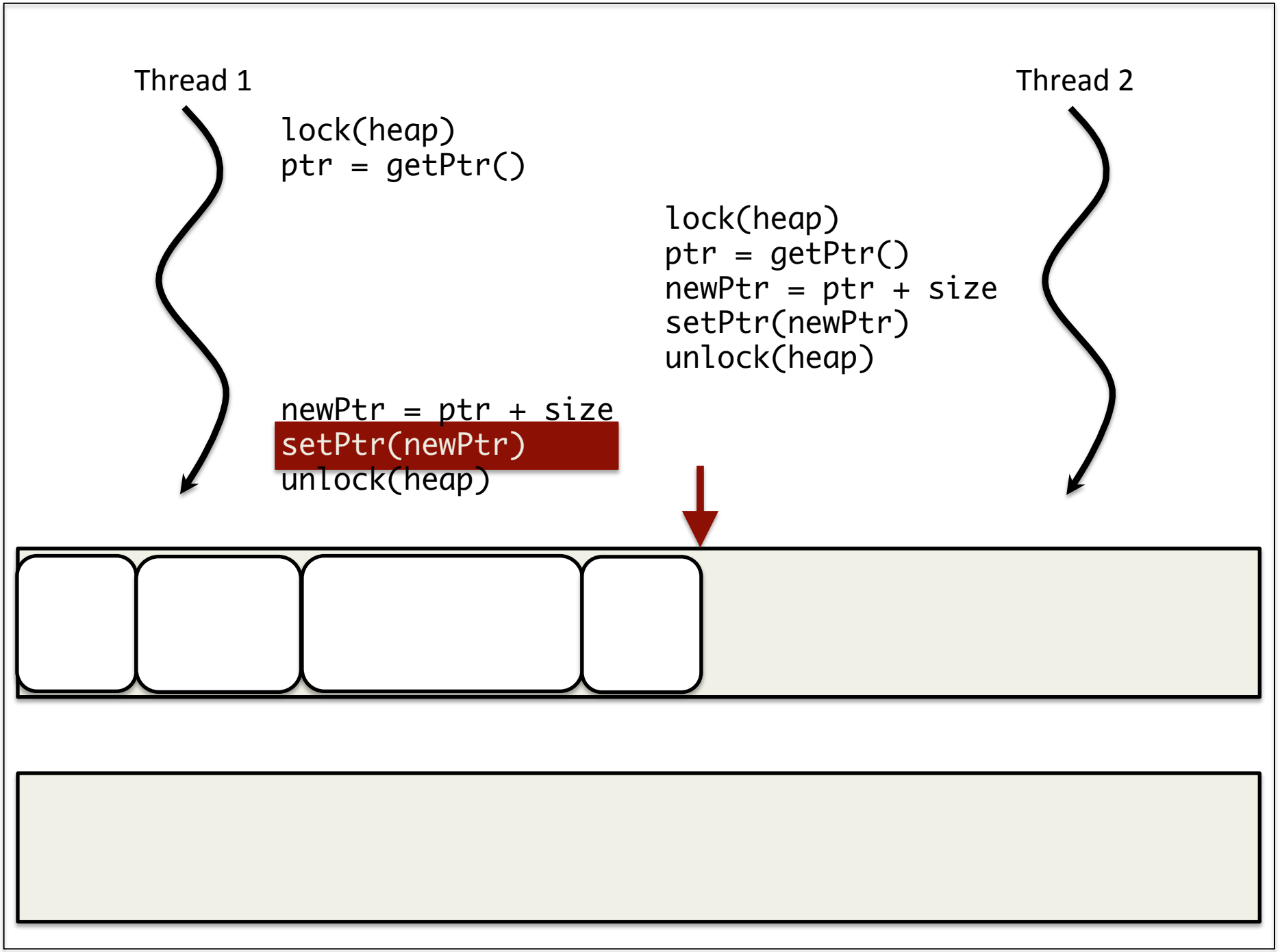

Thread 1
Thread 2
lock(heap)
ptr = getPtr()
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

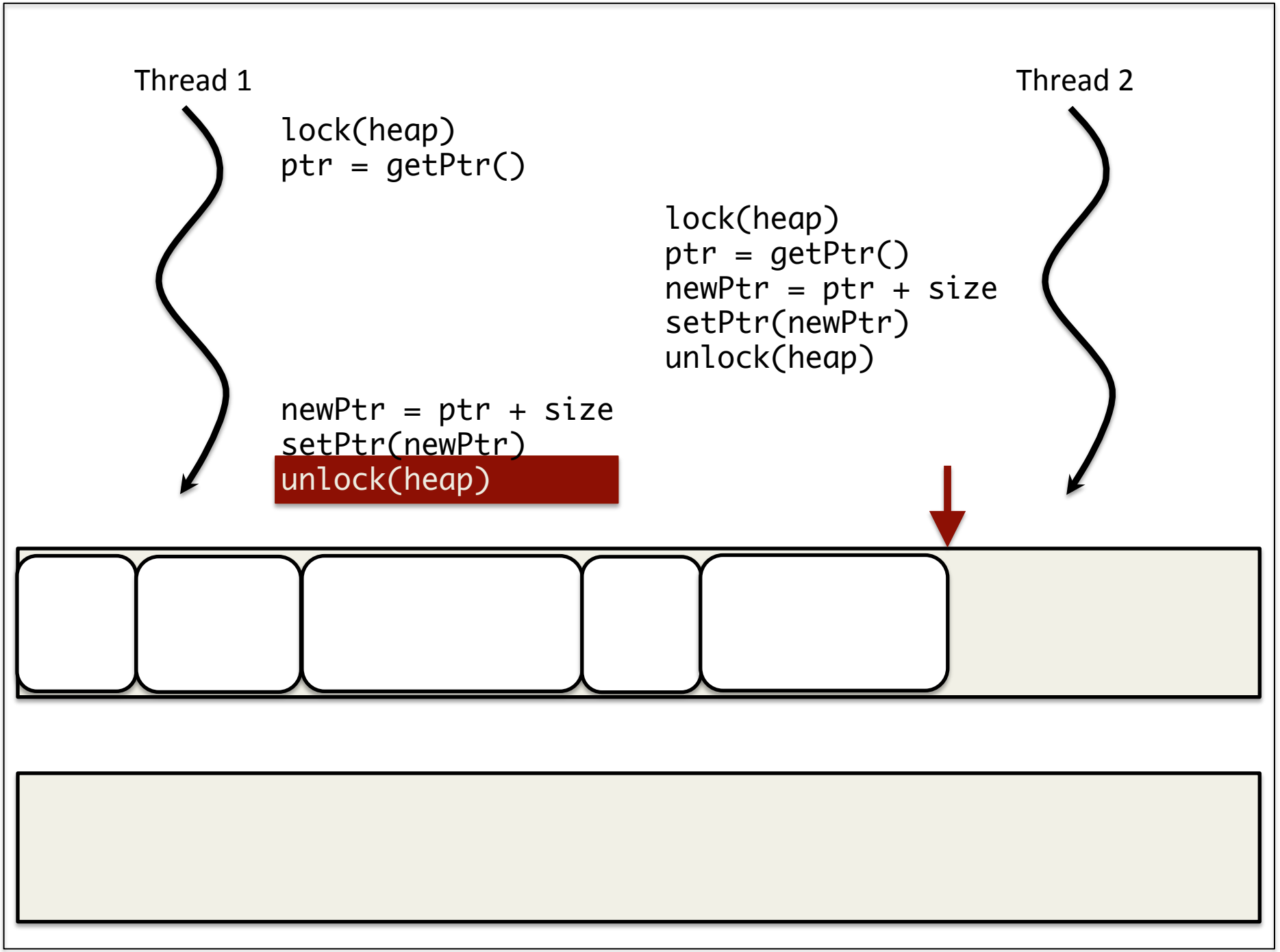

Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

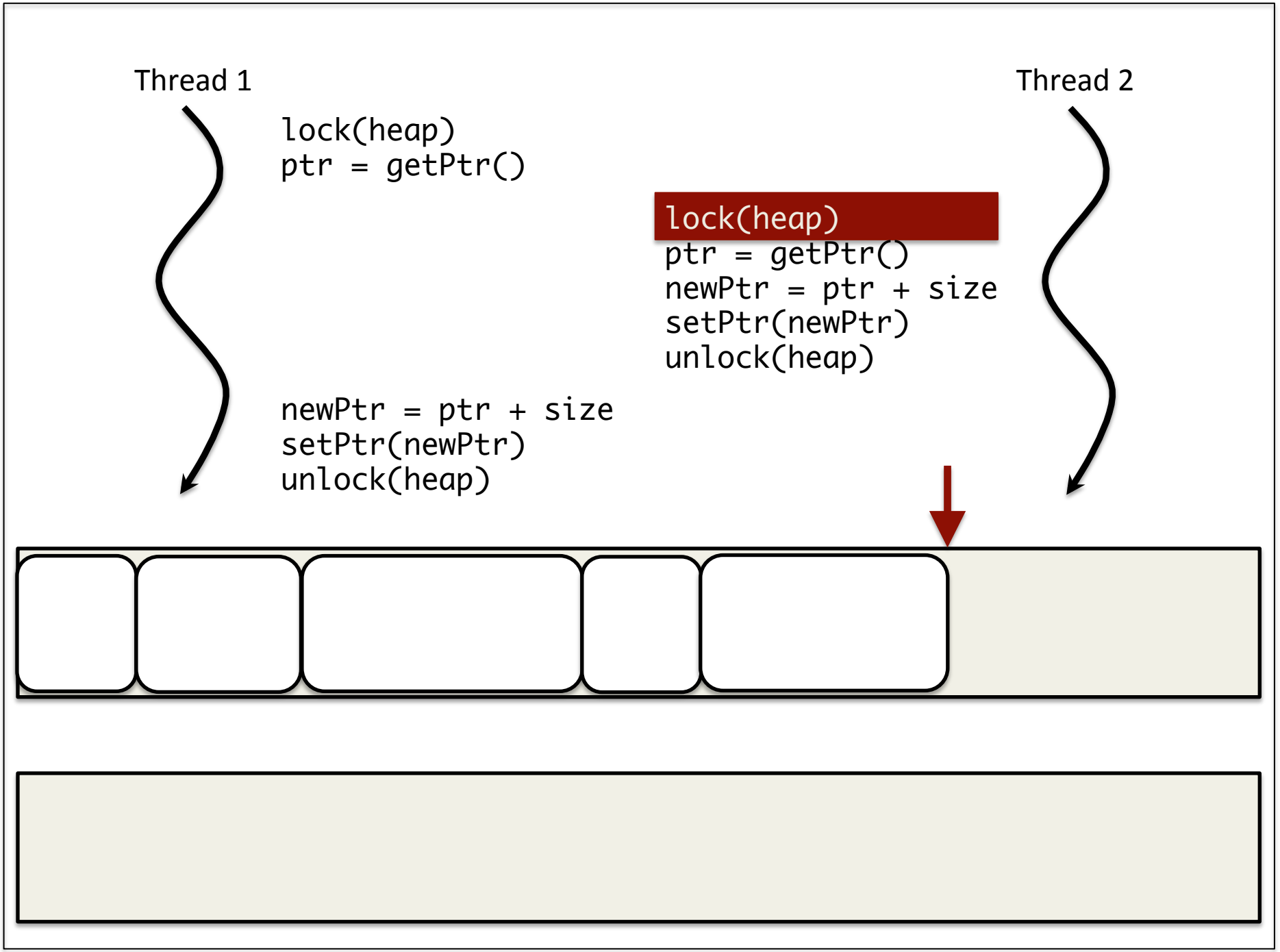

Thread 1
Thread 2
lock(heap)
ptr = getPtr()
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

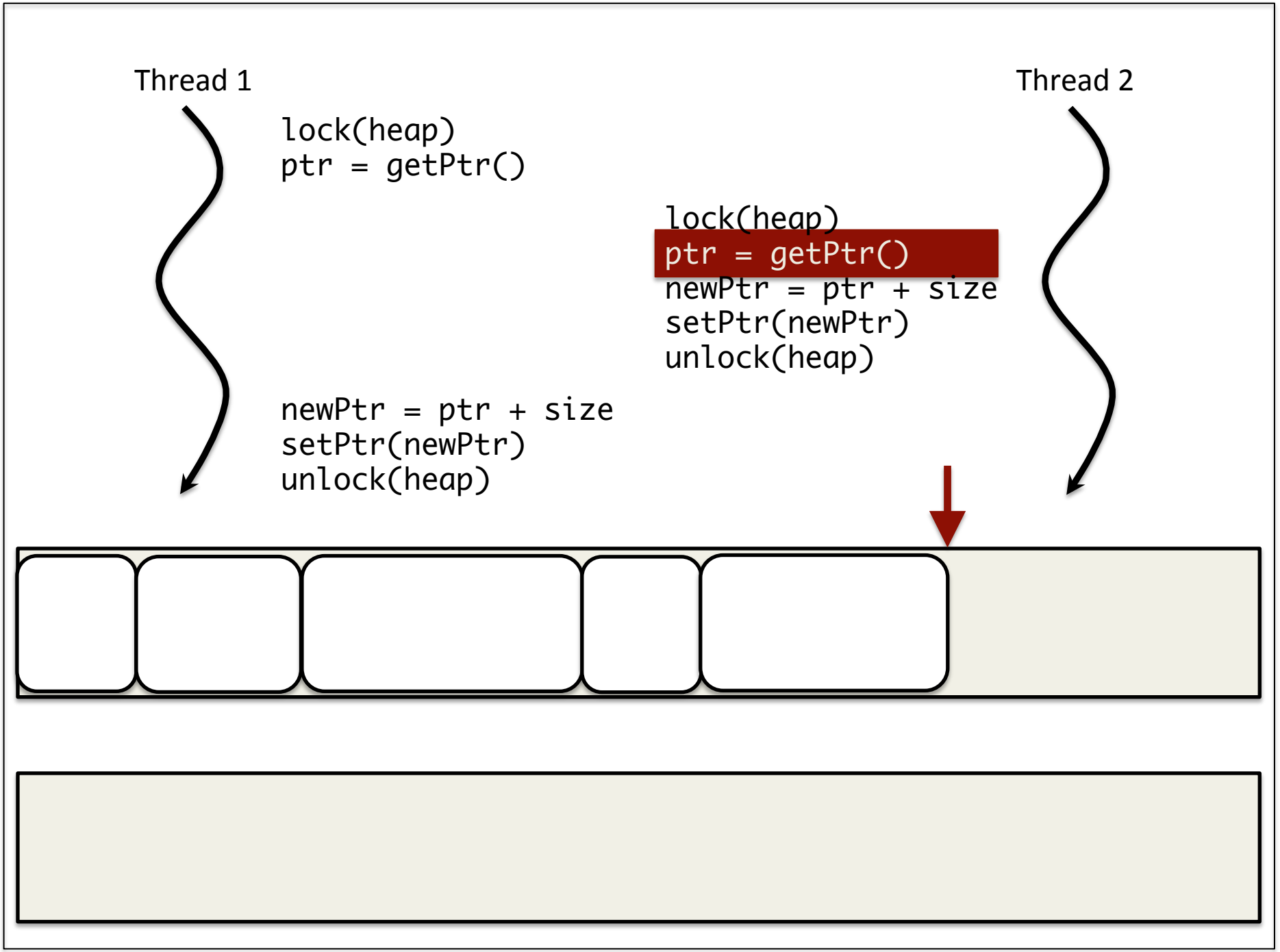

Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2

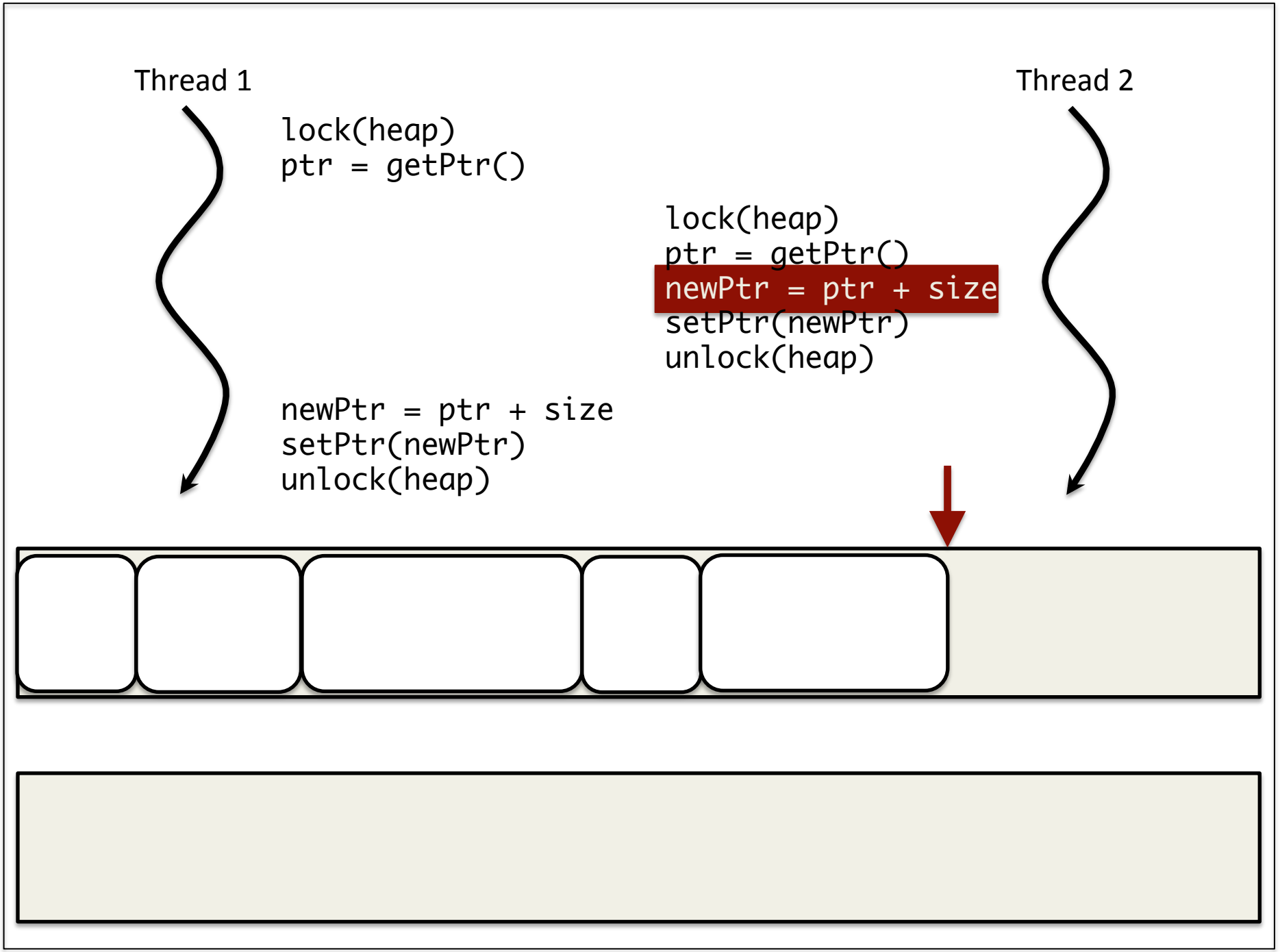

Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

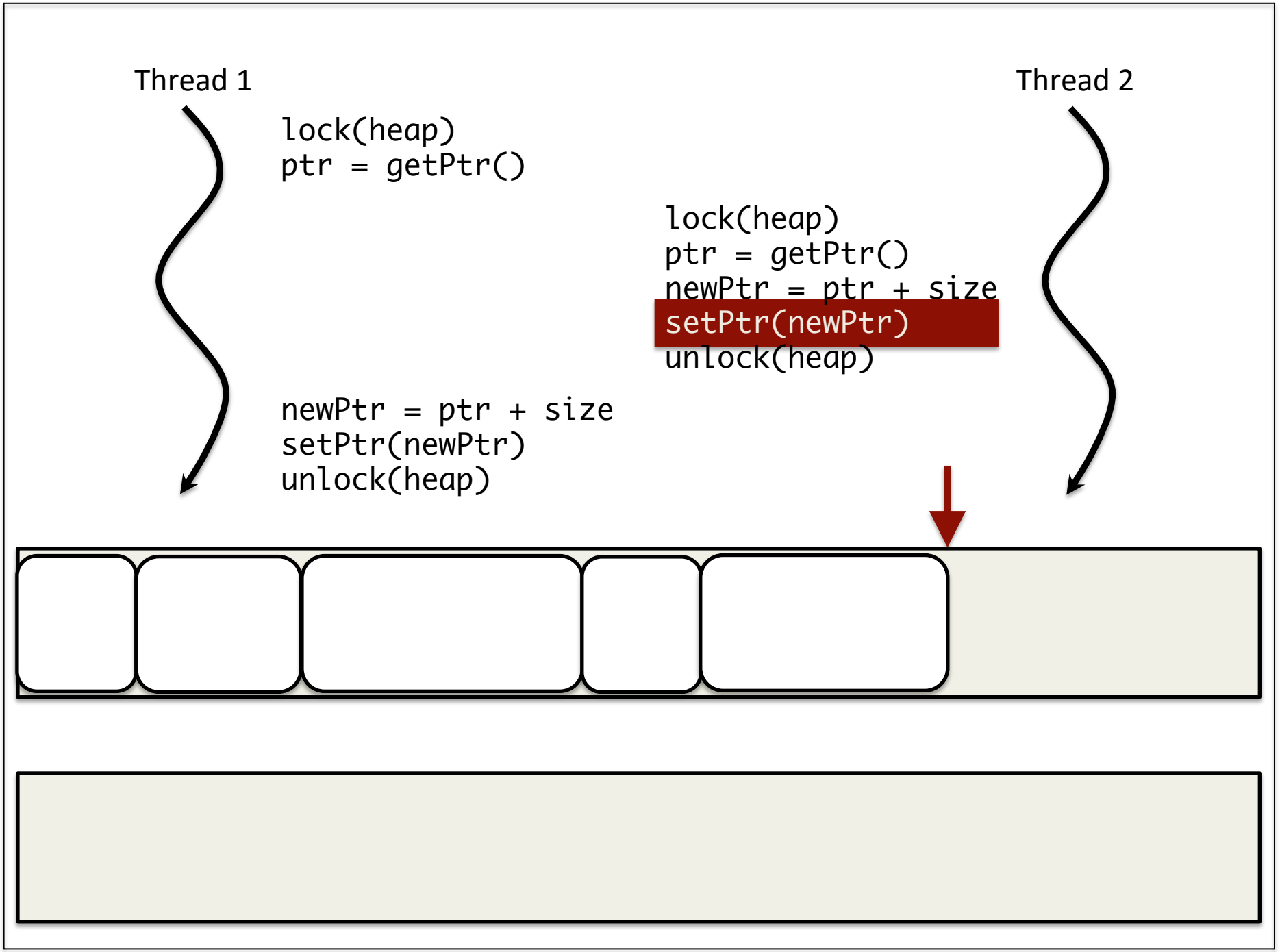

Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

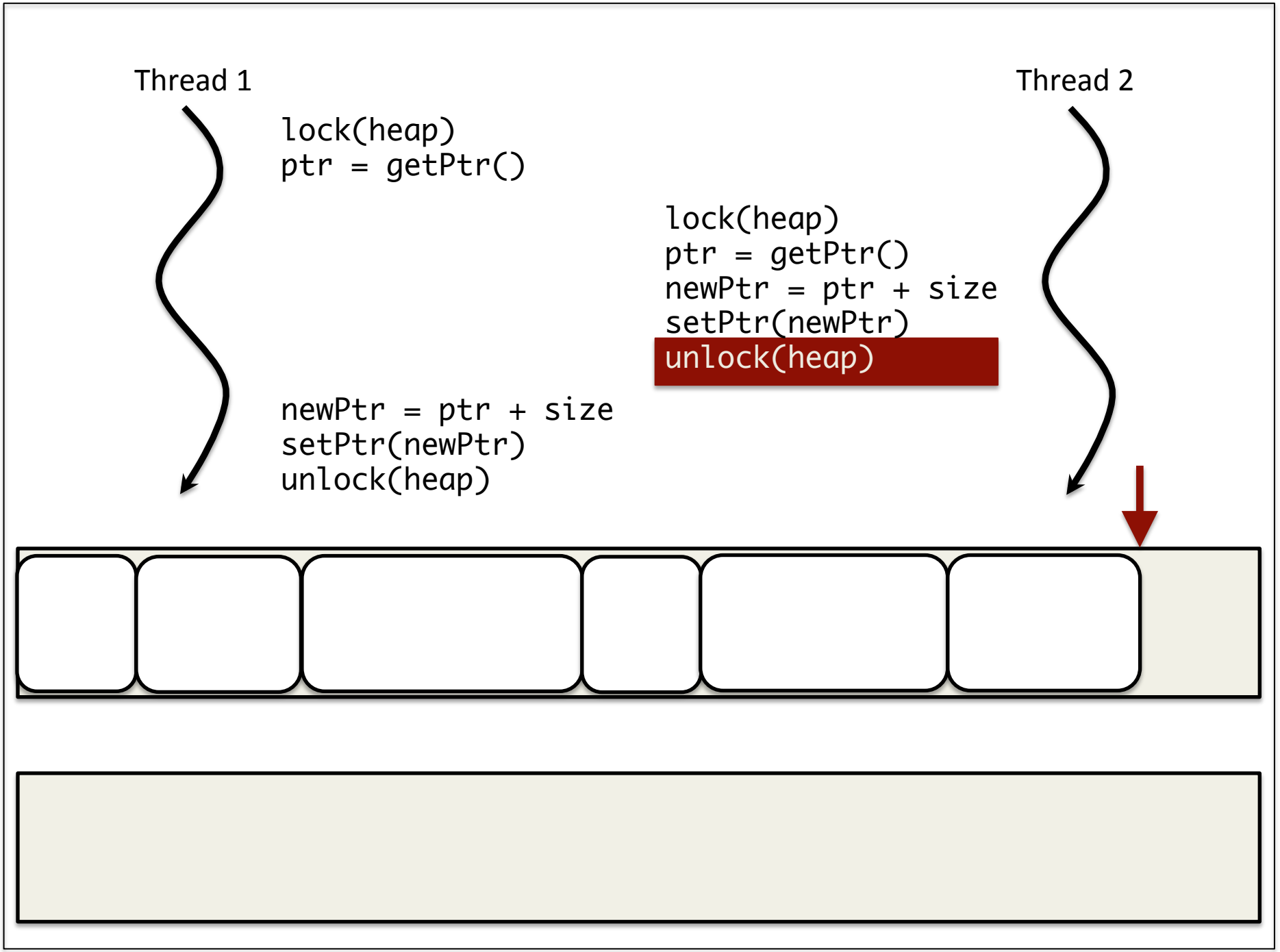

Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)


Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)


Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

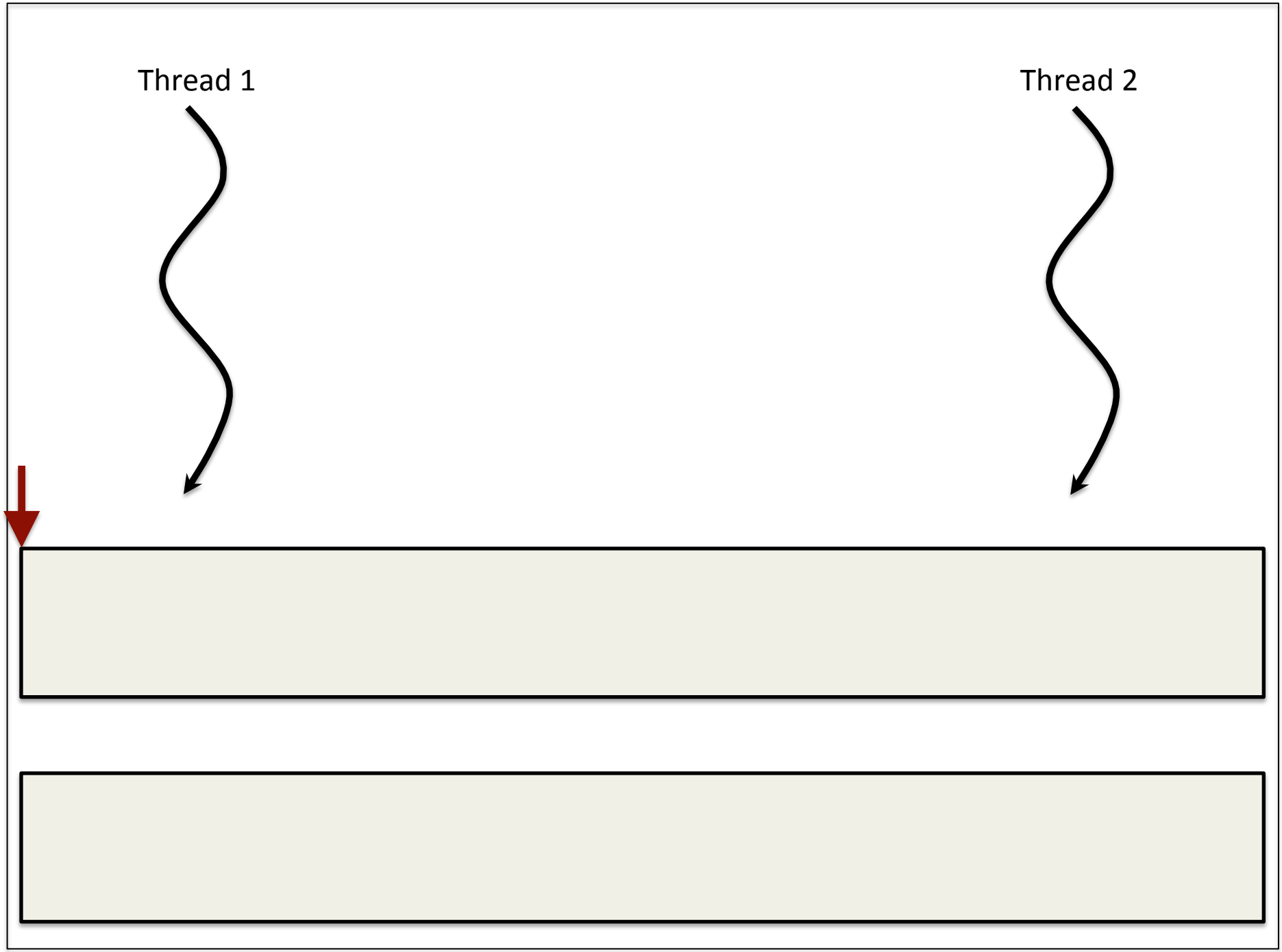

Thread 1
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
Thread 2
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

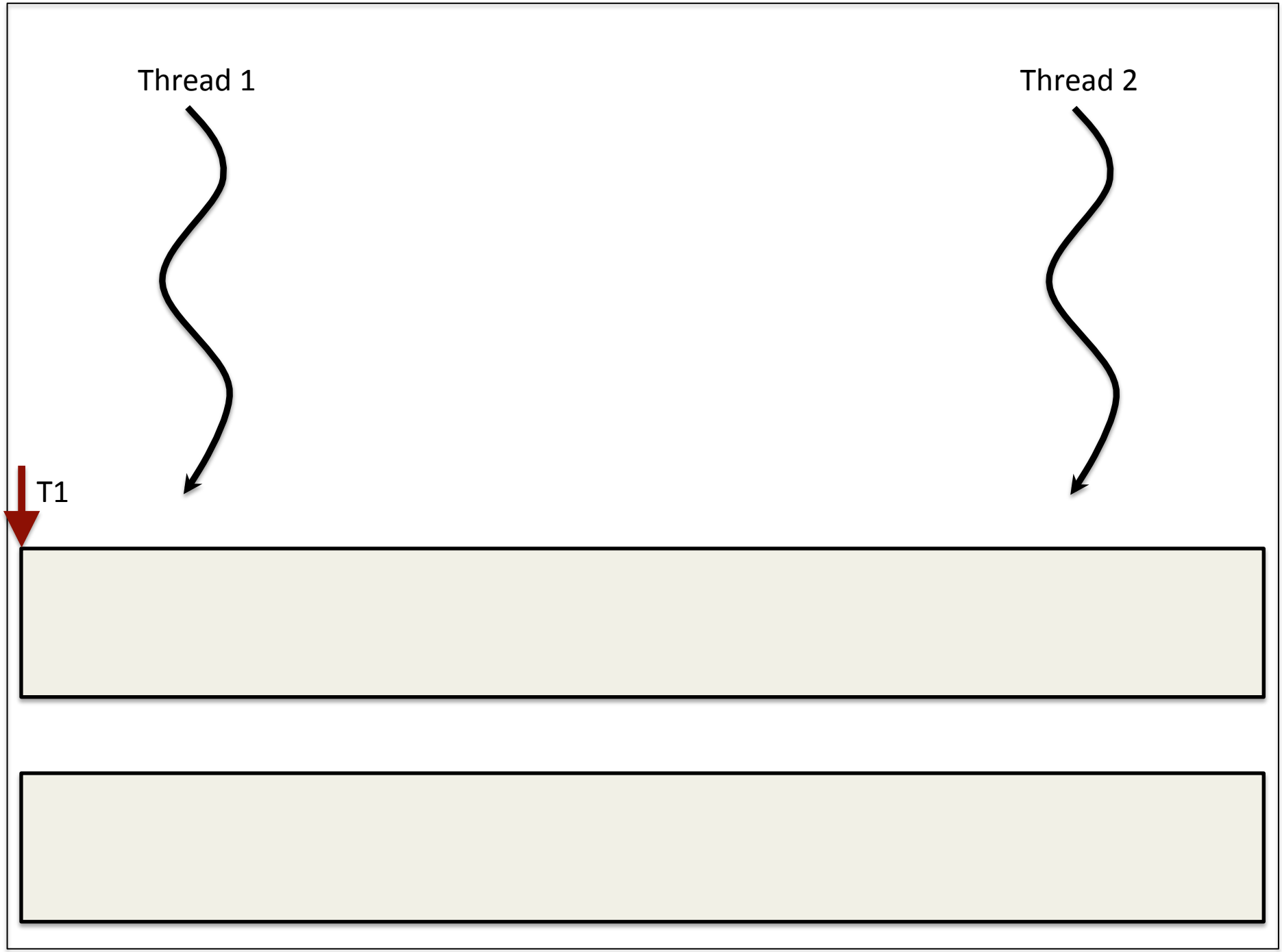

Thread 1
Thread 2
lock(heap)
ptr = getPtr()
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

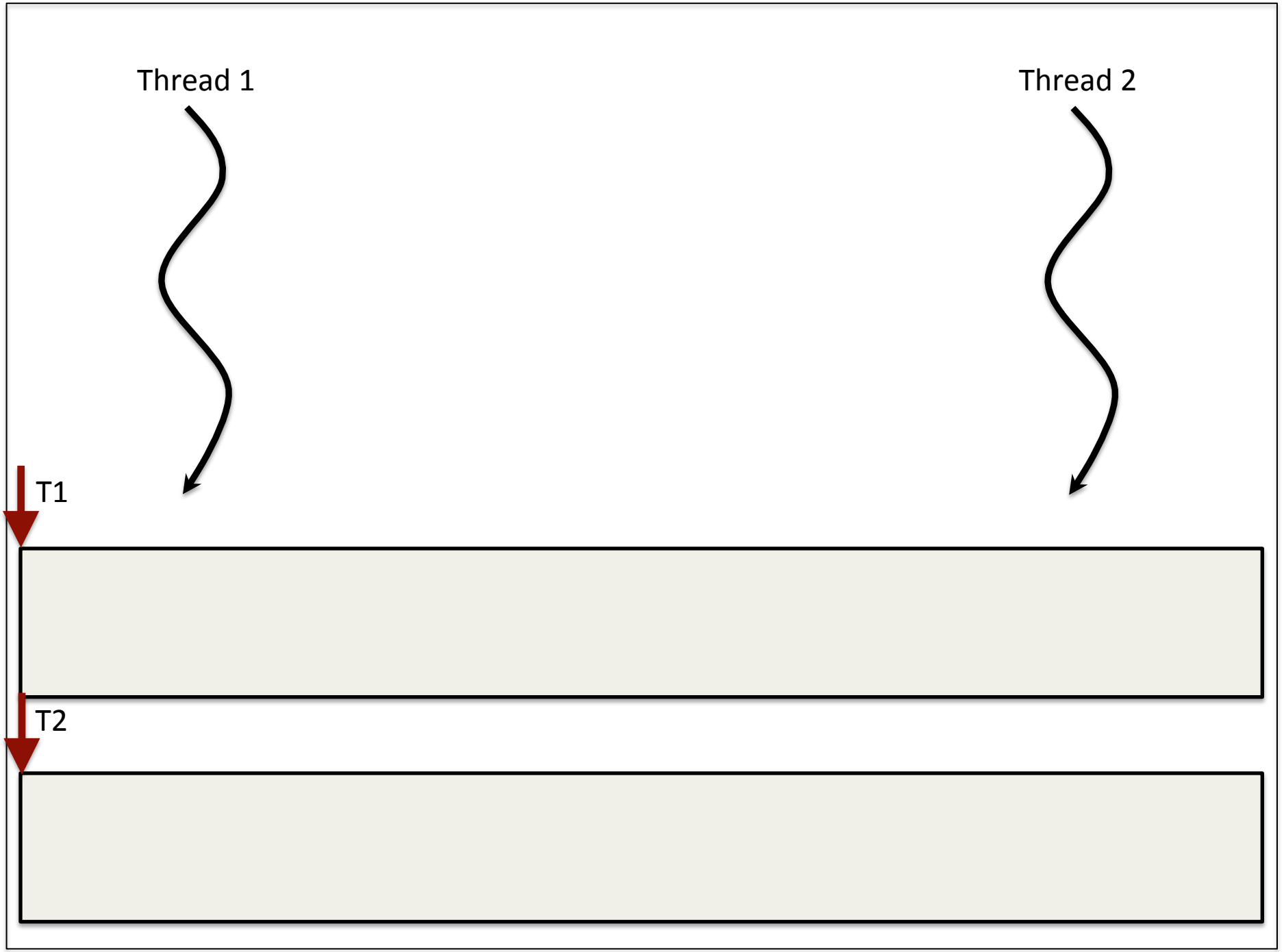

Thread 1
Thread 2
lock(heap)
ptr = getPtr()
lock(heap)
ptr = getPtr()
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)
newPtr = ptr + size
setPtr(newPtr)
unlock(heap)

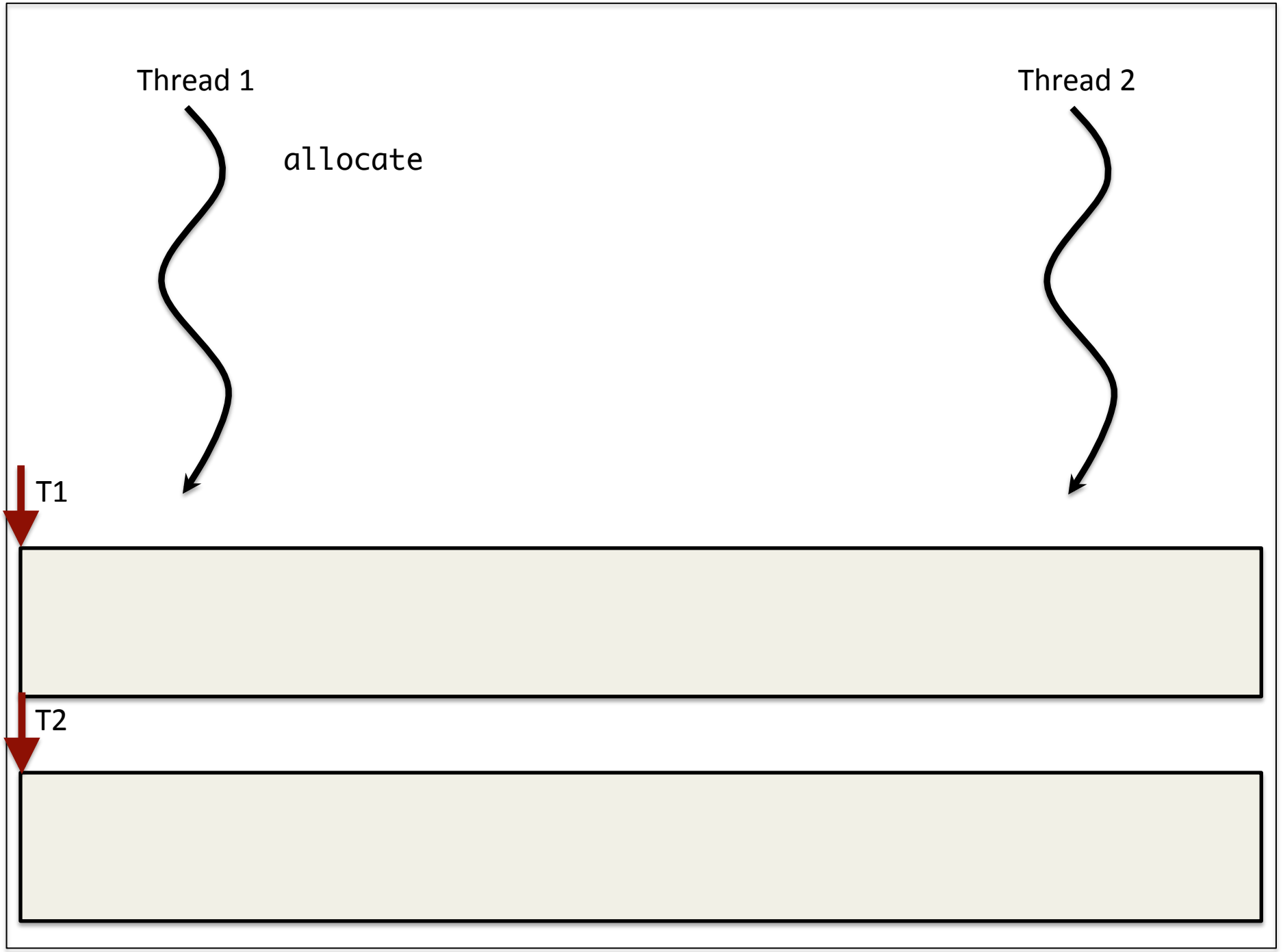
Contention

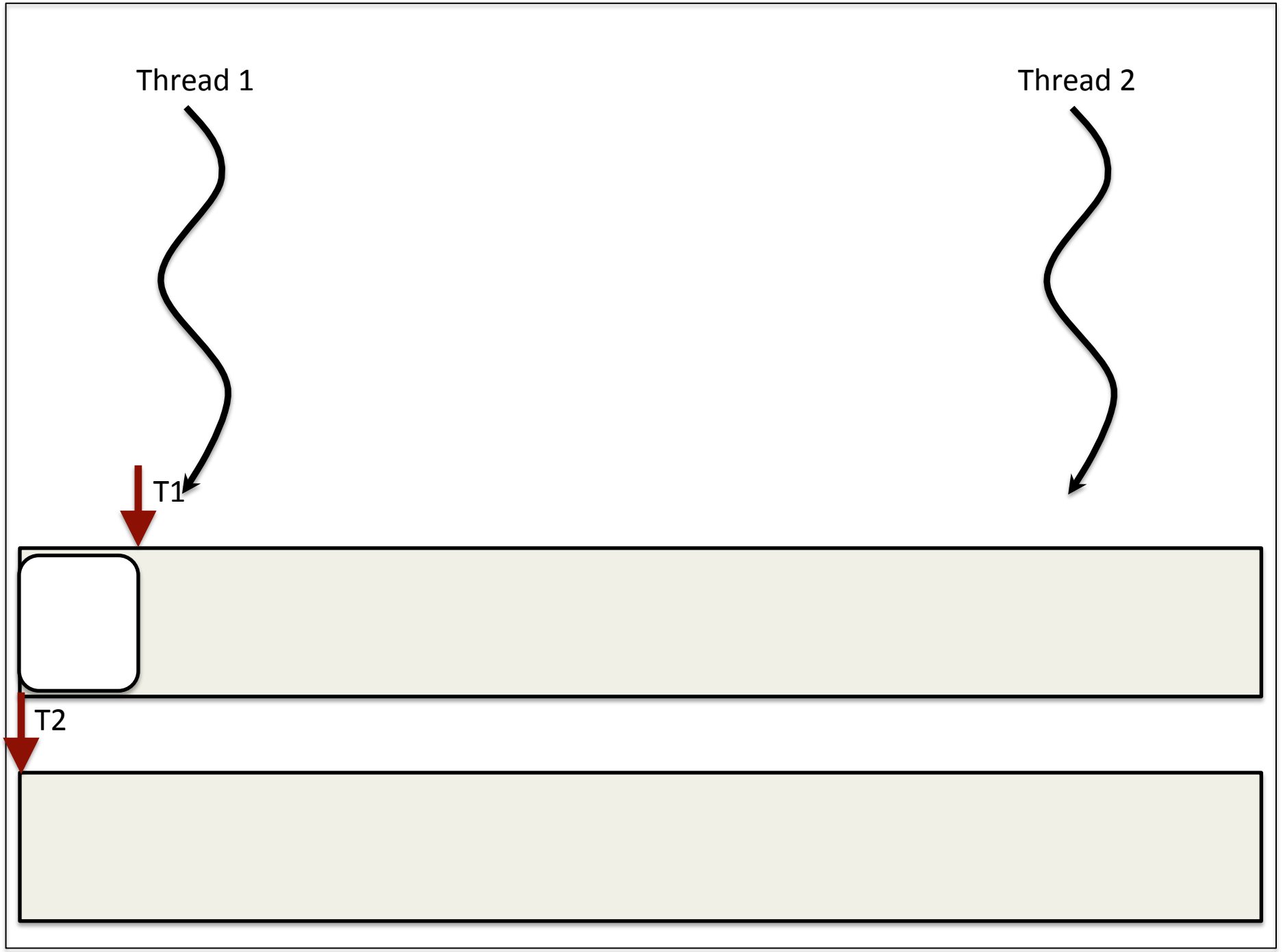
- Allocation is a very common operation
  - Particularly in managed environments
  - Developers assume allocation is easy
- Contention on the allocator is bad
  - Particularly as the number of threads increases
  - Single bottleneck for the whole system
- Better to allocate without synchronization

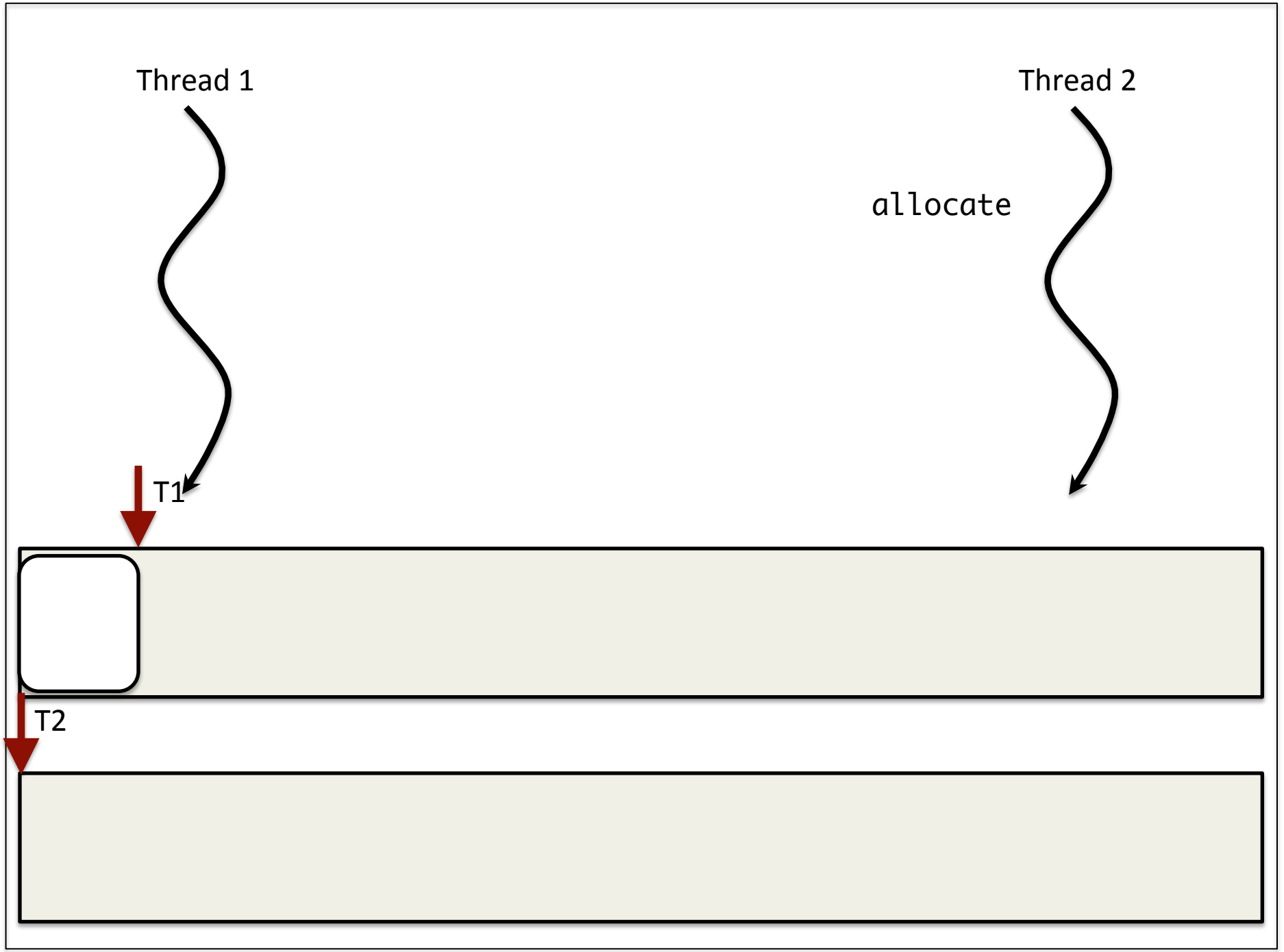

Thread 1
Thread 2

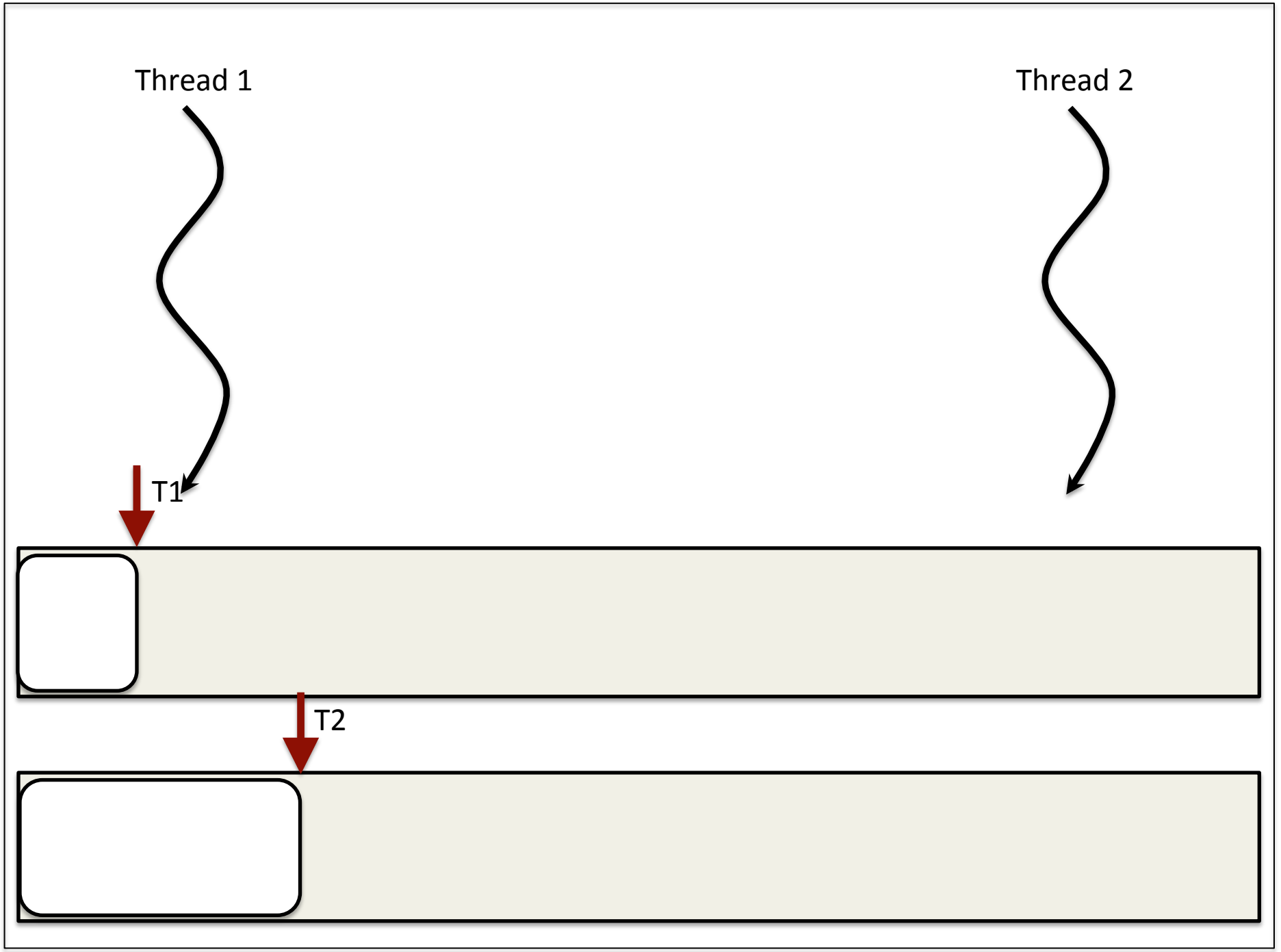

Thread 1
Thread 2
T1

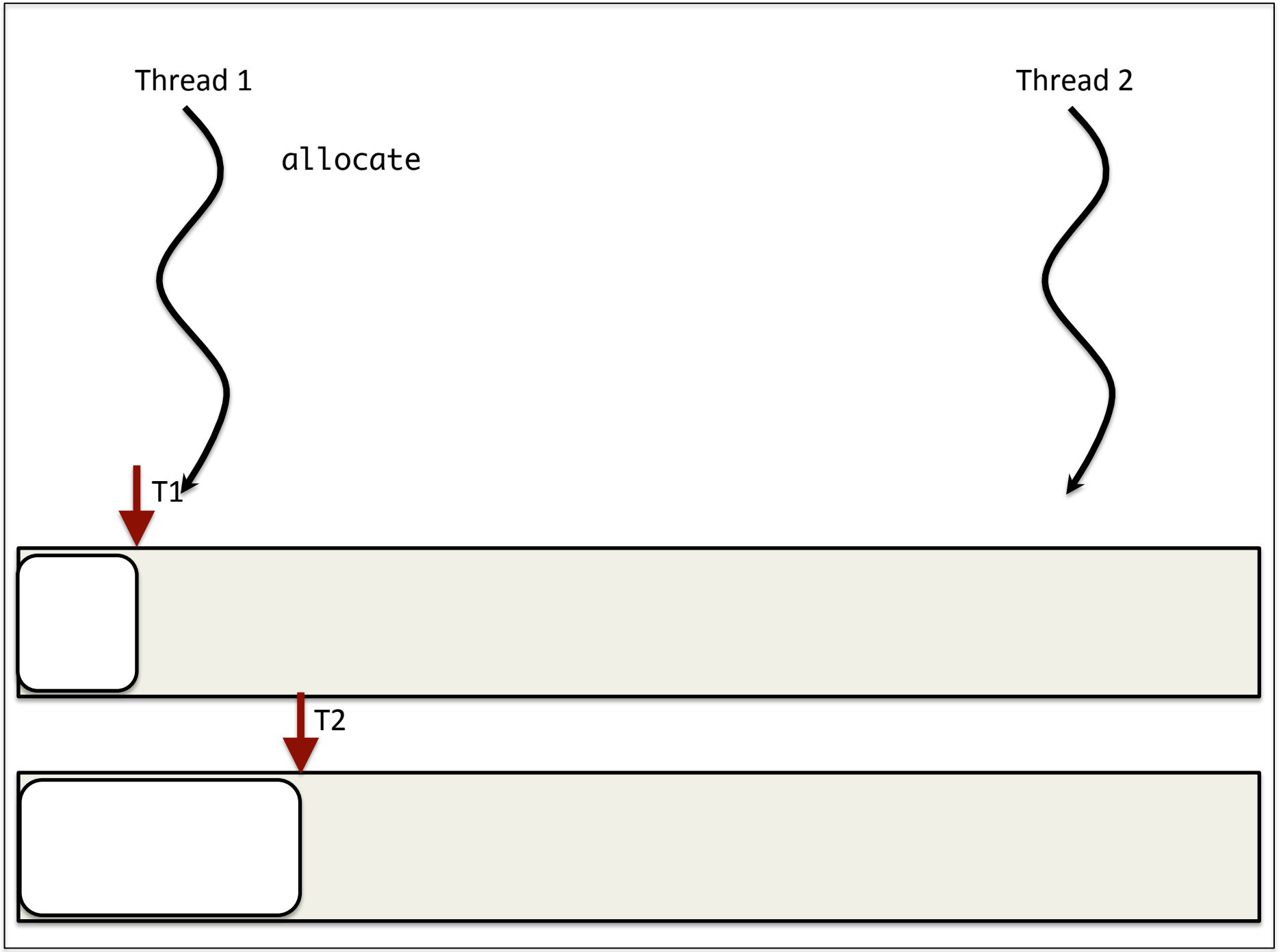

Thread 1
Thread 2
T1
T2

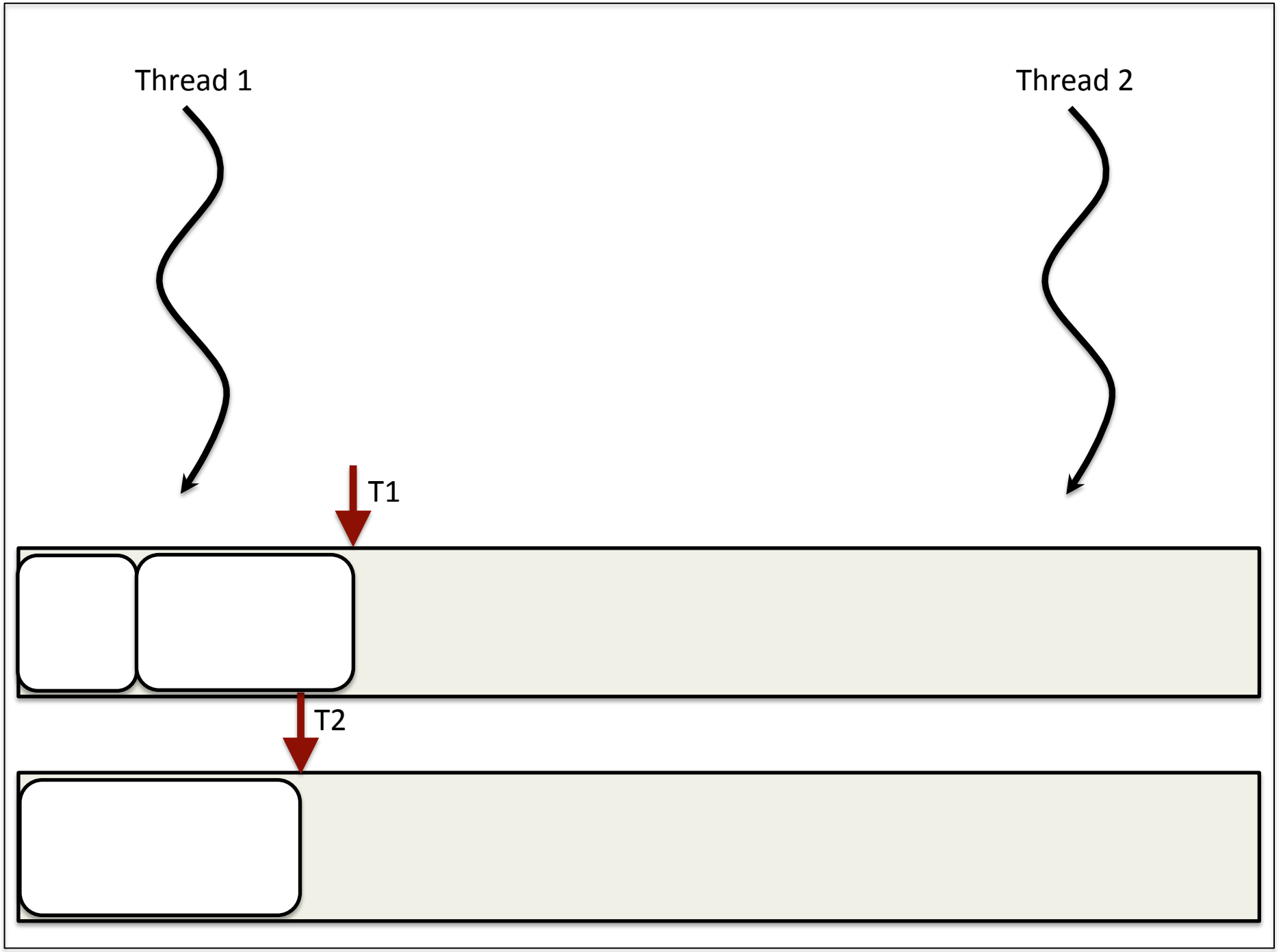

Thread 1
allocate
Thread 2
T1
T2

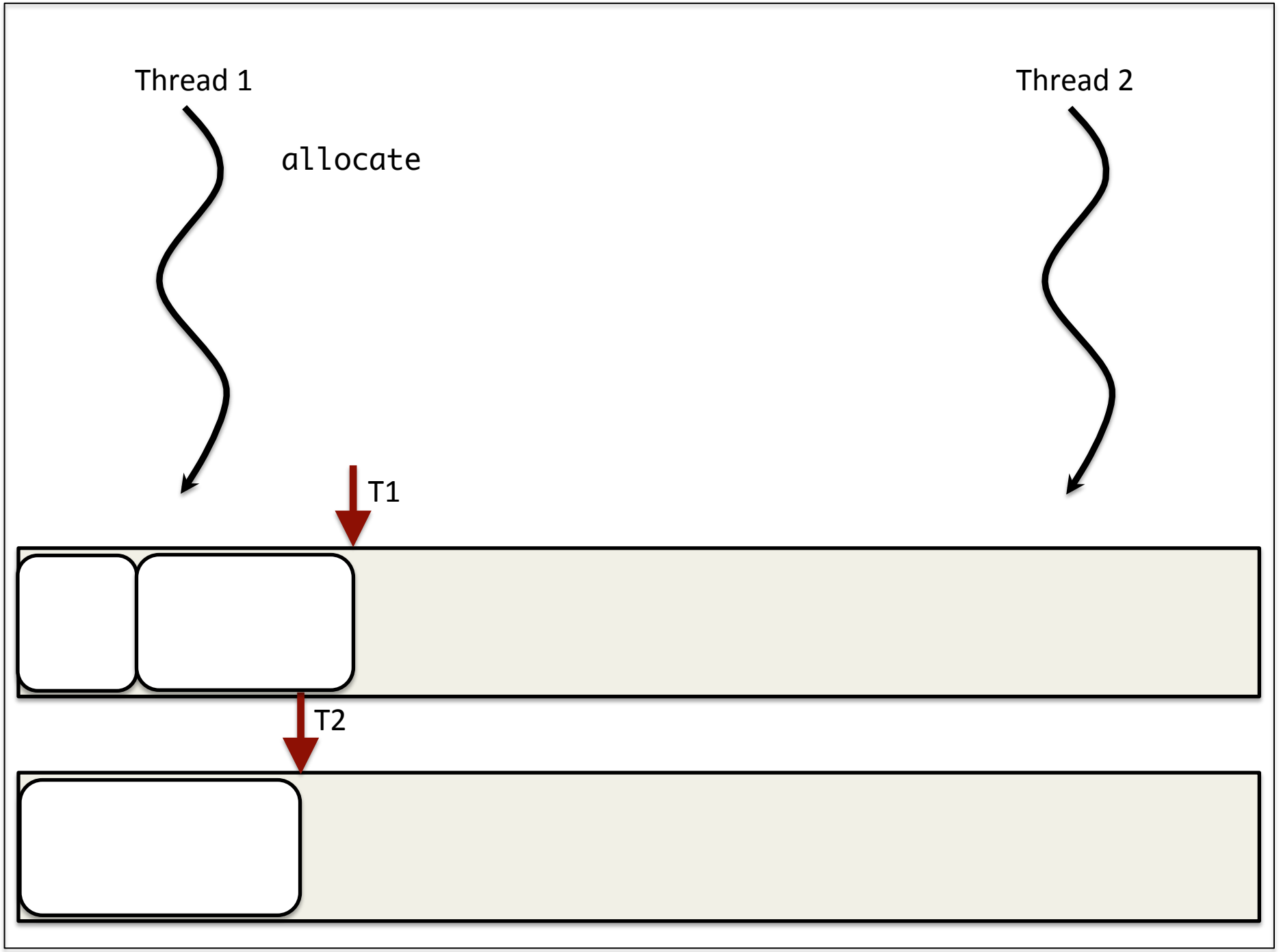

Thread 1
Thread 2
T1
T2

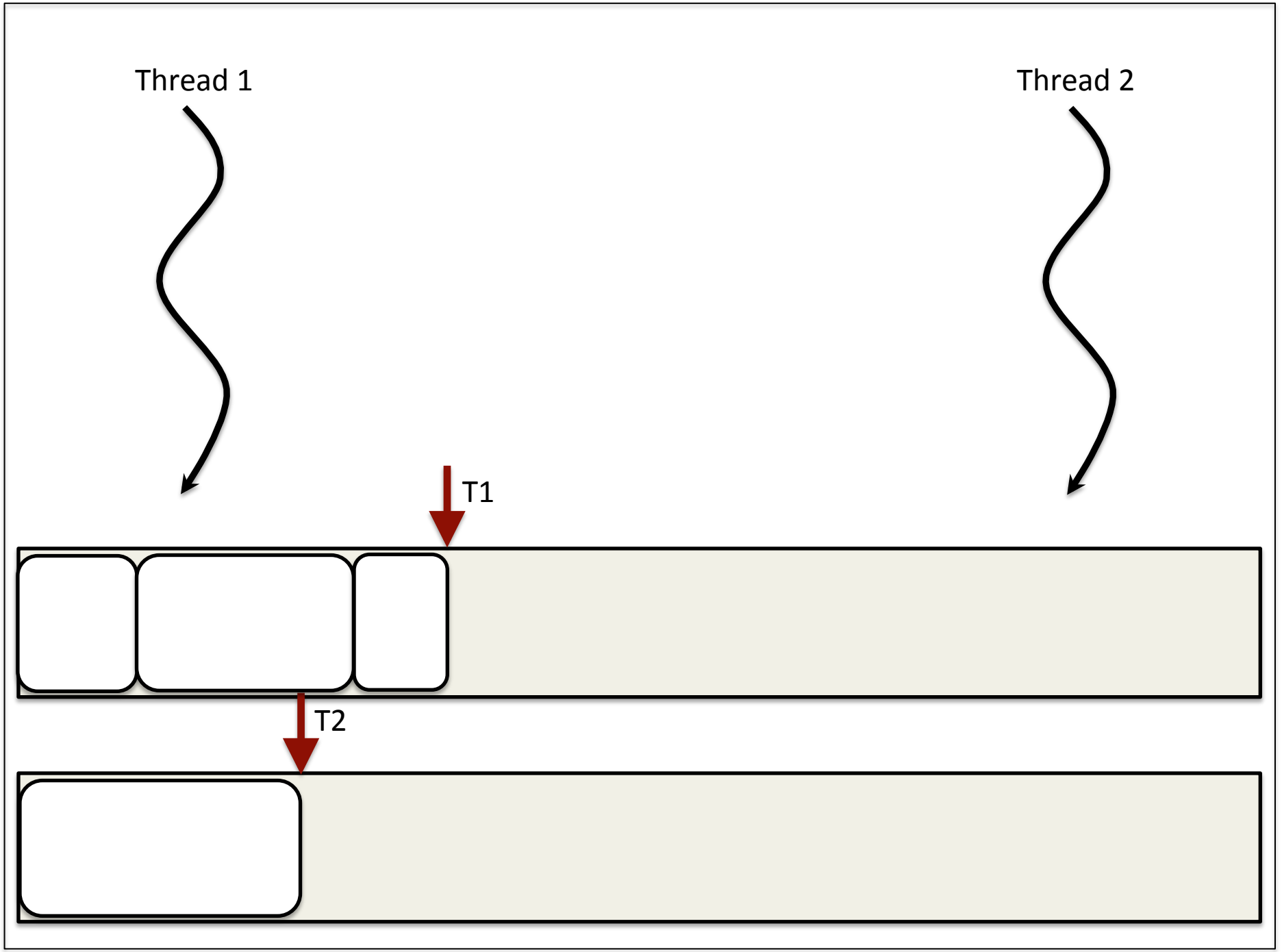

Thread 1
Thread 2
allocate
T1
T2


Thread 1
Thread 2
T1
T2


Thread 1
allocate
Thread 2
T1
T2


Thread 1
Thread 2
T1
T2


Thread 1
allocate
Thread 2
T1
T2


Thread 1
Thread 2
T1
T2

Thread-Local Allocation

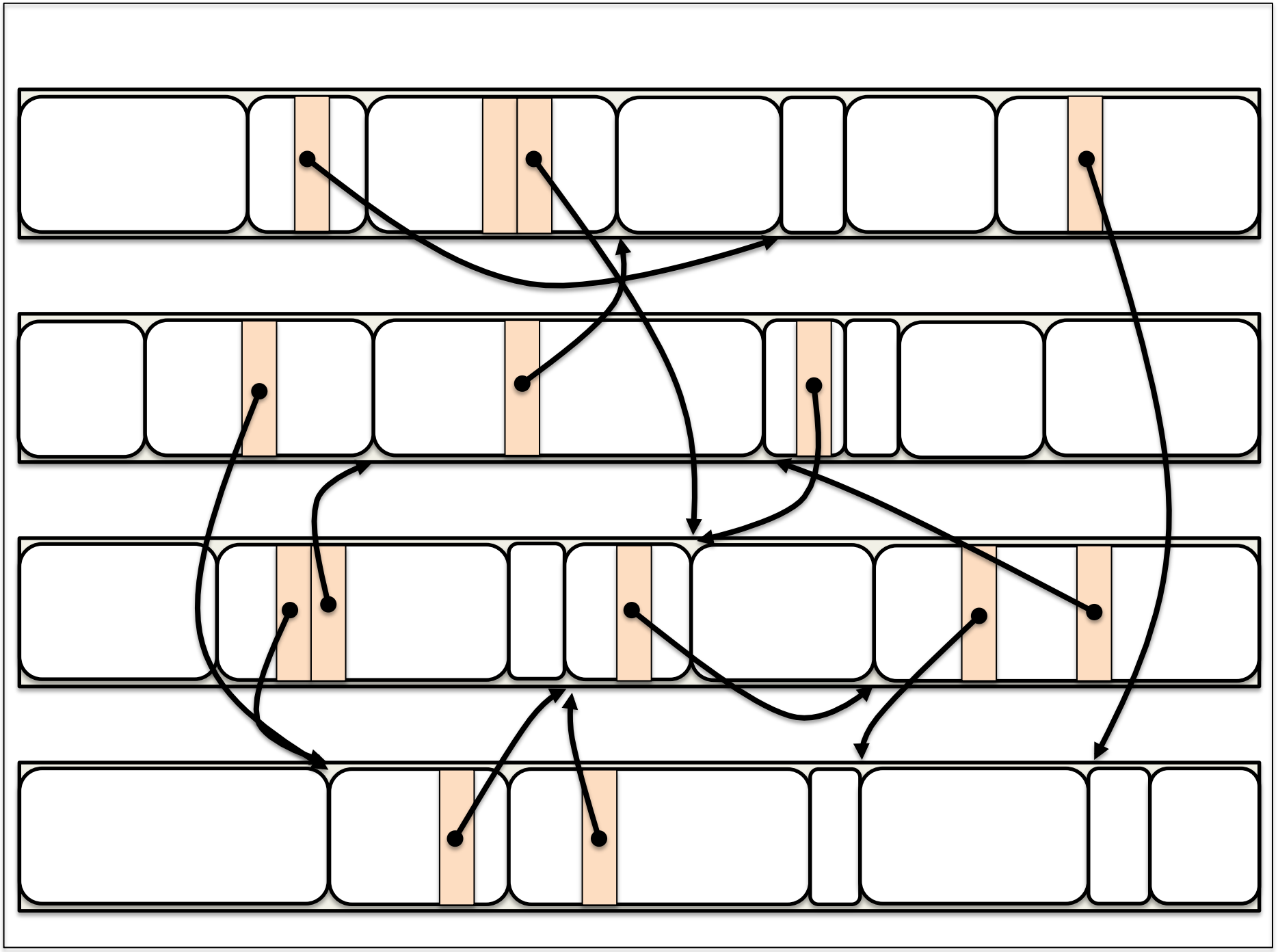
- The nursery is divided into smaller regions
  - Thread Local Allocation Buffer (TLAB)
- Each thread has its own TLAB for allocation
  - Store the buffer details in thread-local storage
  - No contention for the allocator
- Nursery maintains a set of available TLABs
  - Threads grab a new one when needed
  - Contention on the TLAB store
  - Garbage collect when there are none left

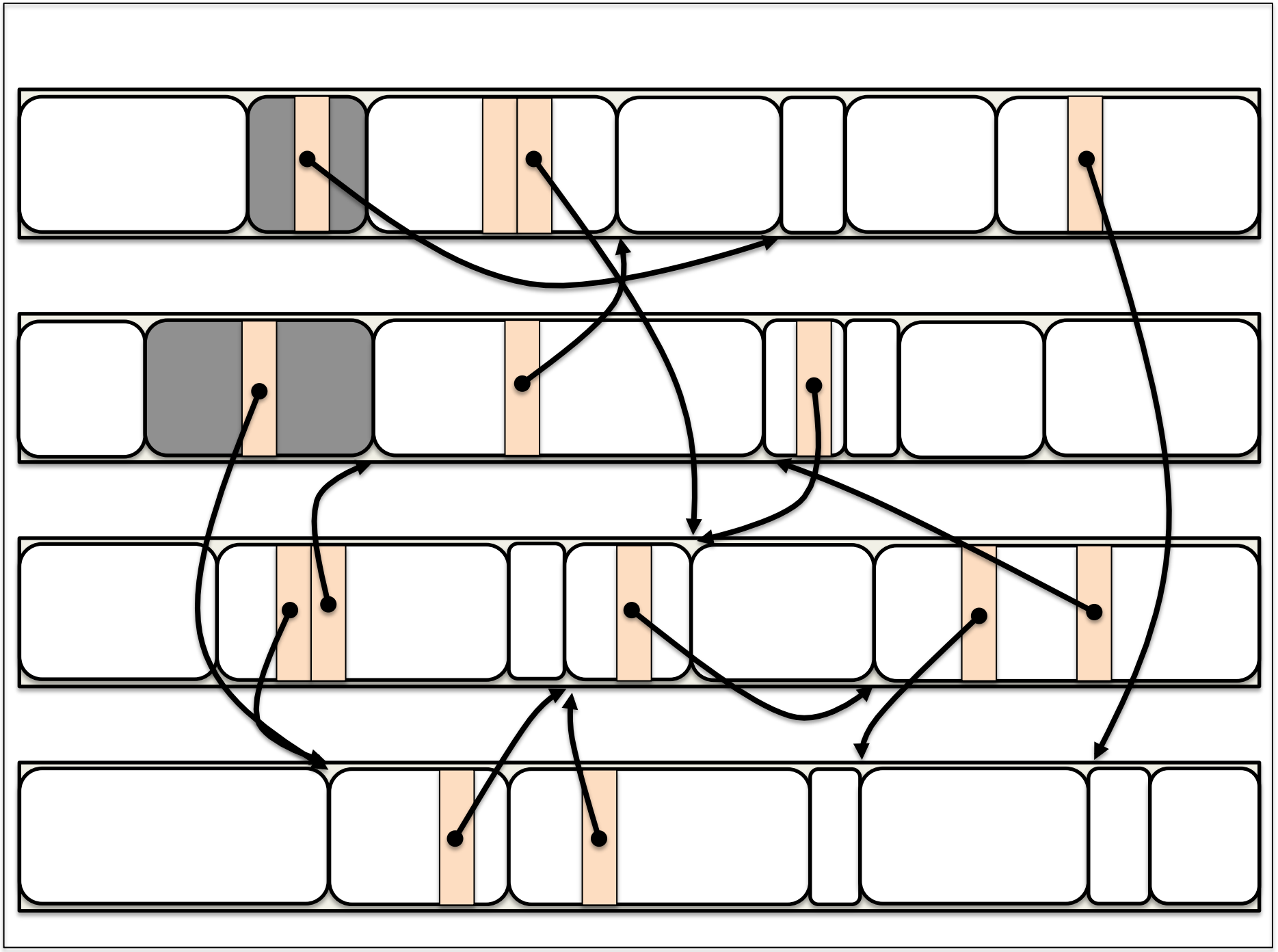
Concurrent Marking

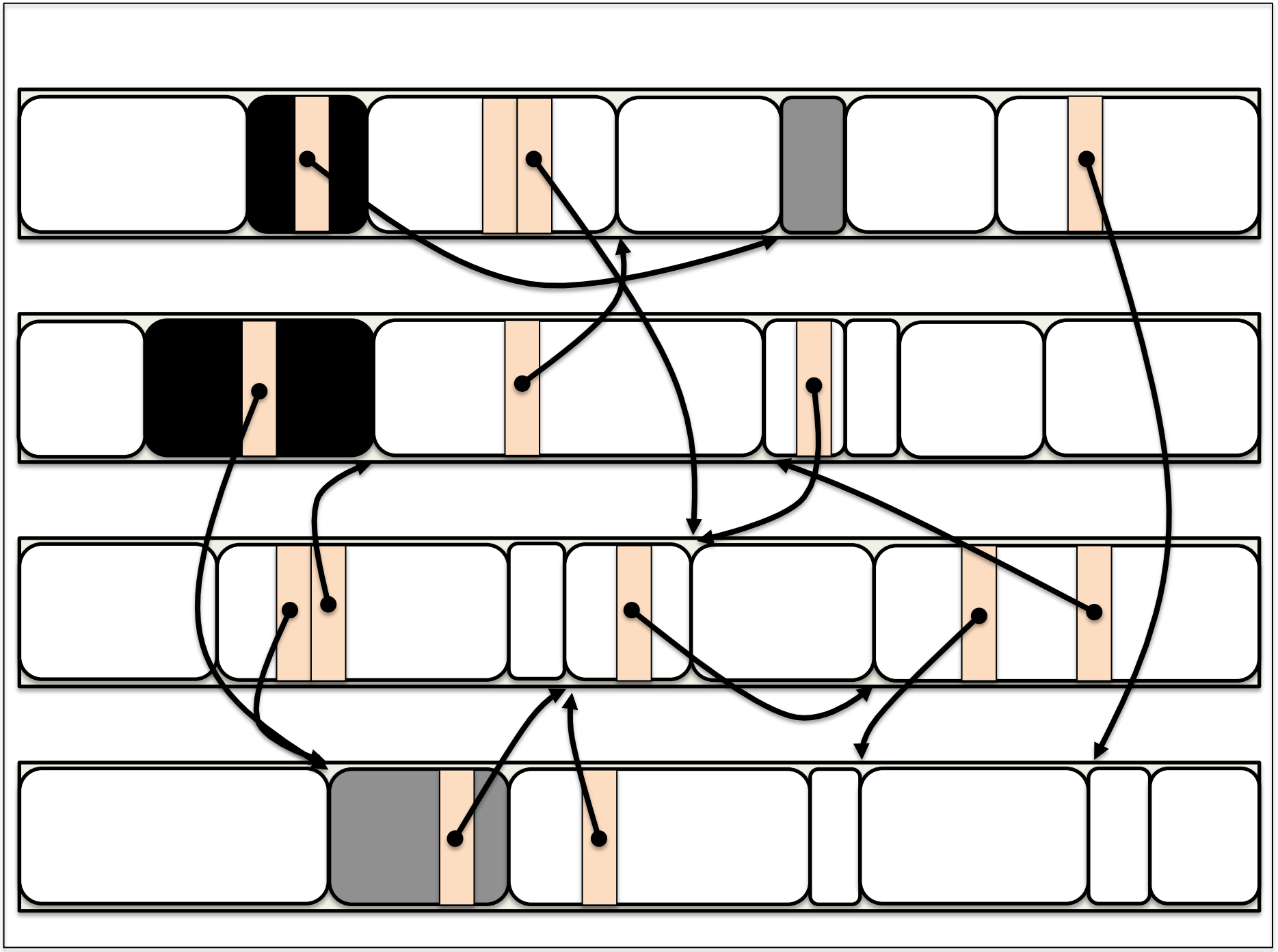
- Marking a large region can be expensive
  - Even with remembered sets
  - Complex write barriers come with other overheads
- Marking can largely be performed concurrently
  - Background threads mark as the mutators run
  - Garbage objects don't become live again
- What happens if the object graph changes?

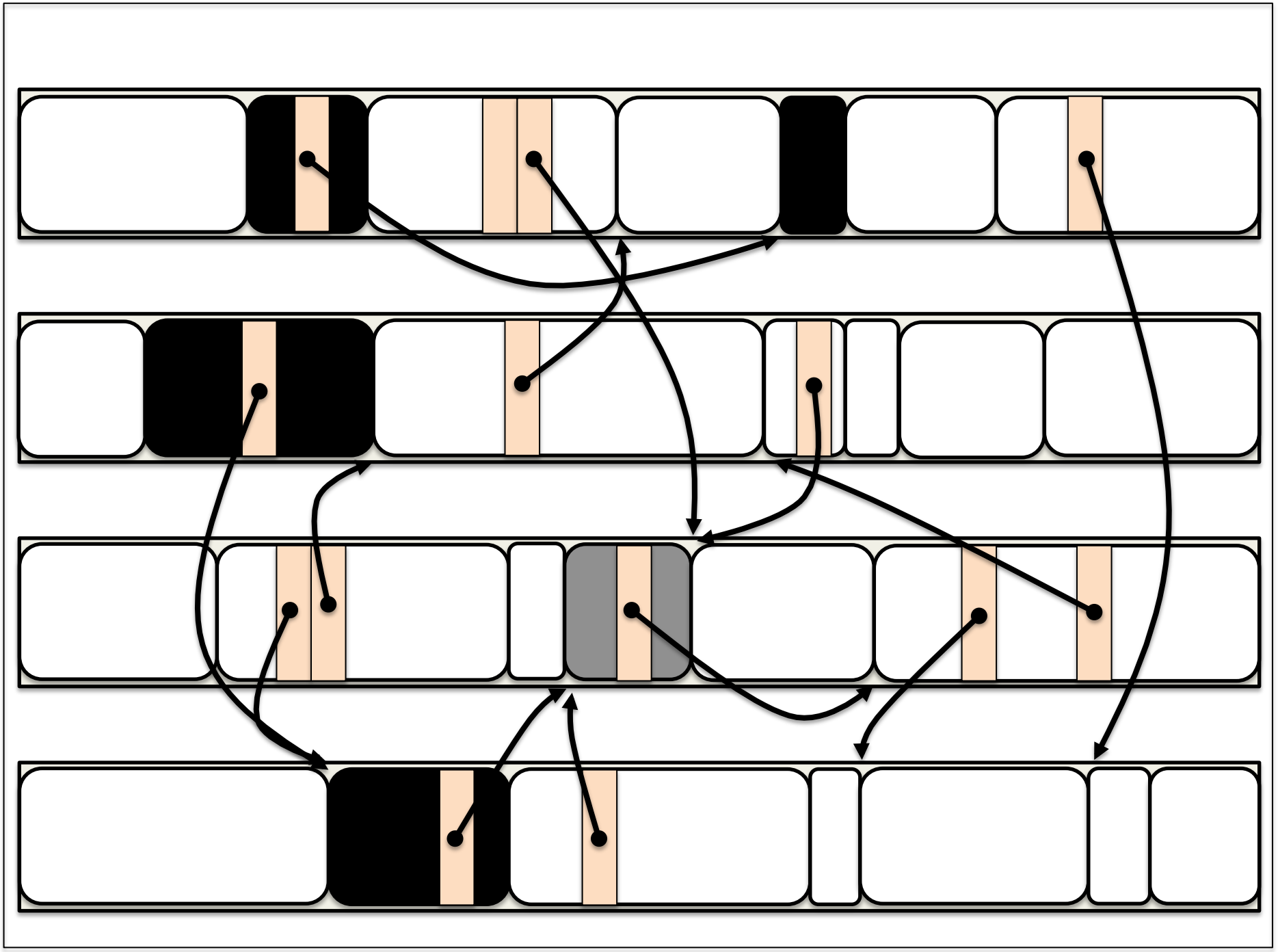
Tricolor Marking

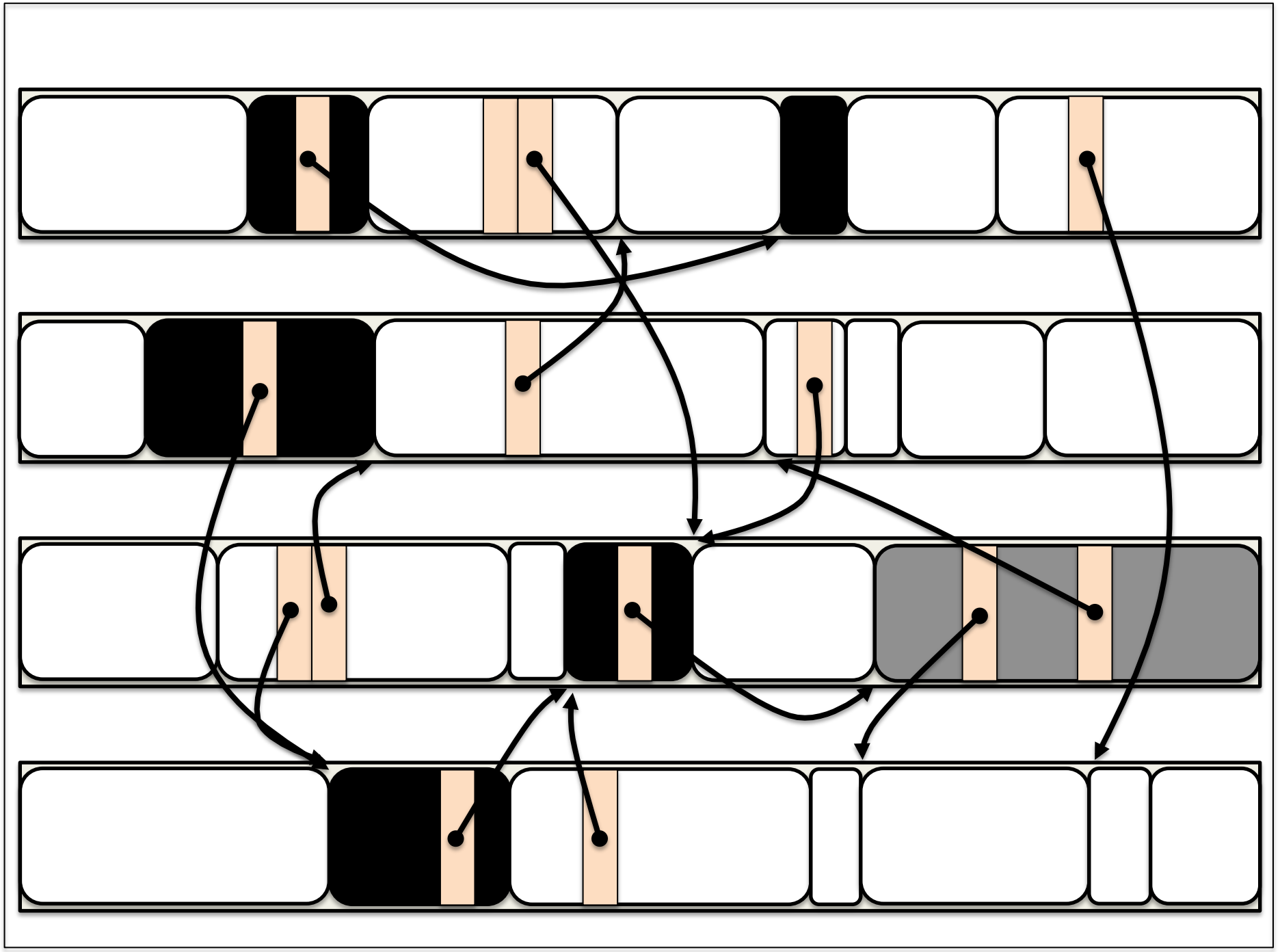
- Convenient way to think about marking
- Objects exist in three possible states
  - White
  - Gray
  - Black
- Maintain invariants between colors
  - Black objects can't point to white
  - Objects cannot go from black or gray to white

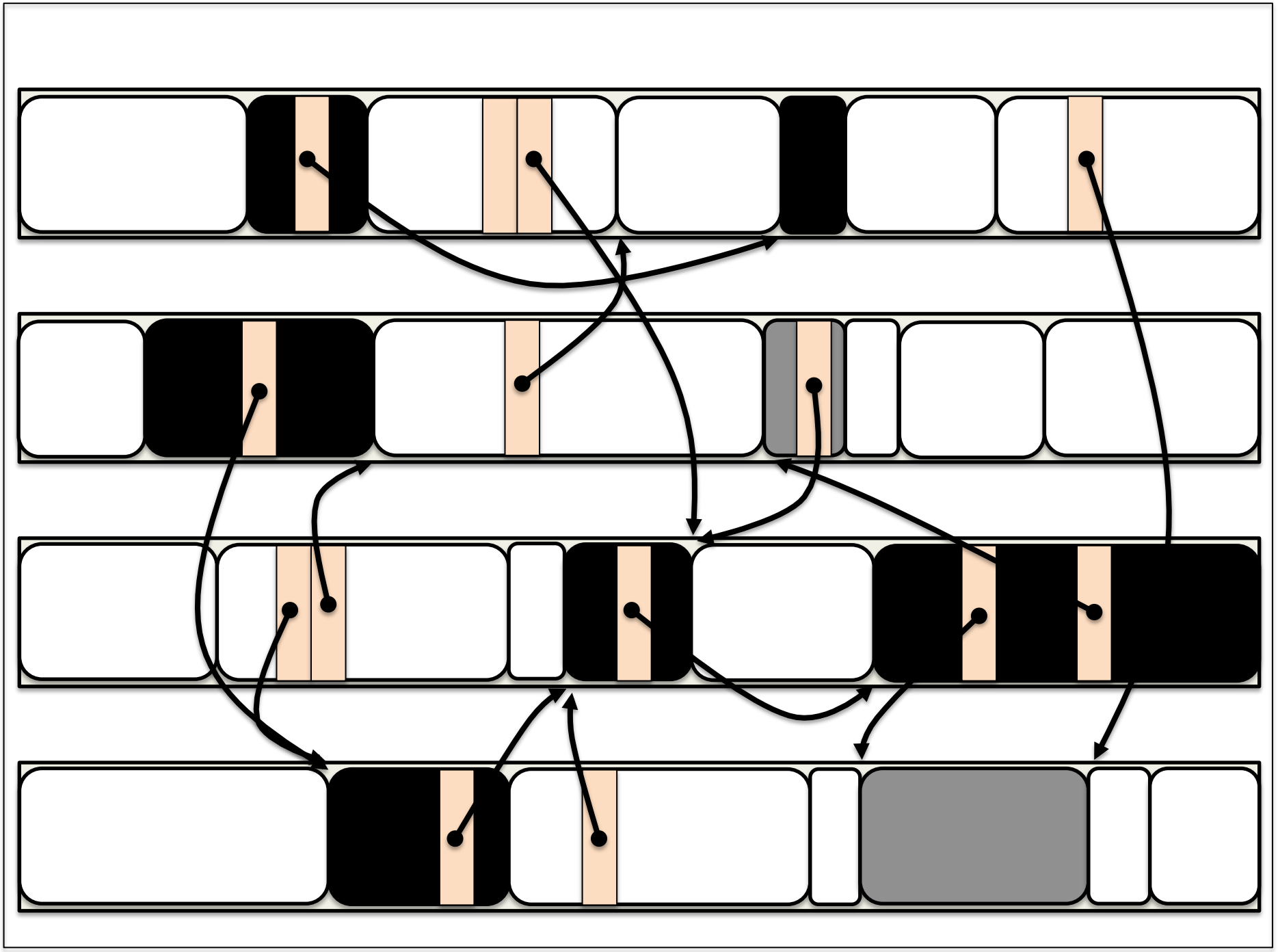


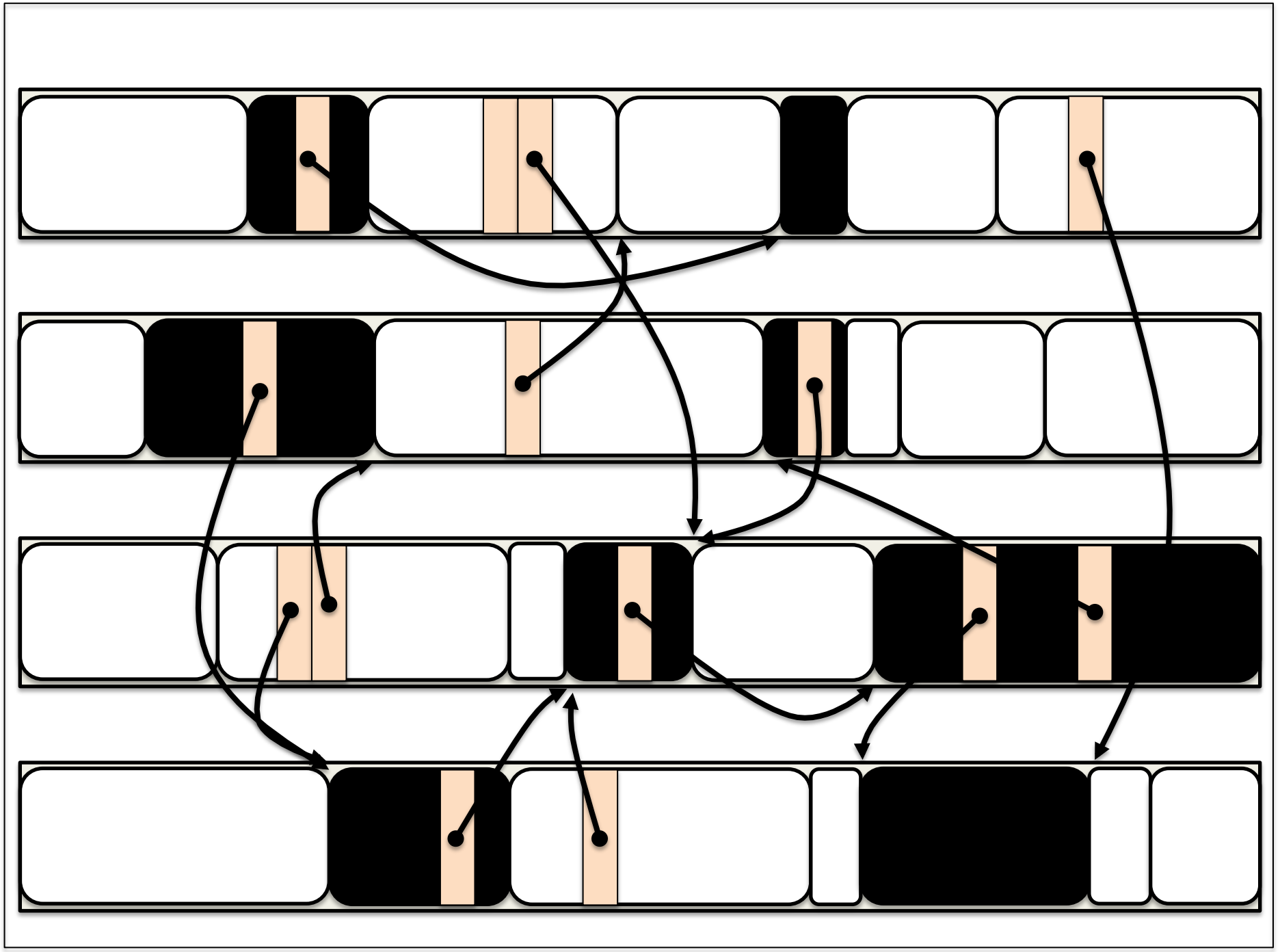






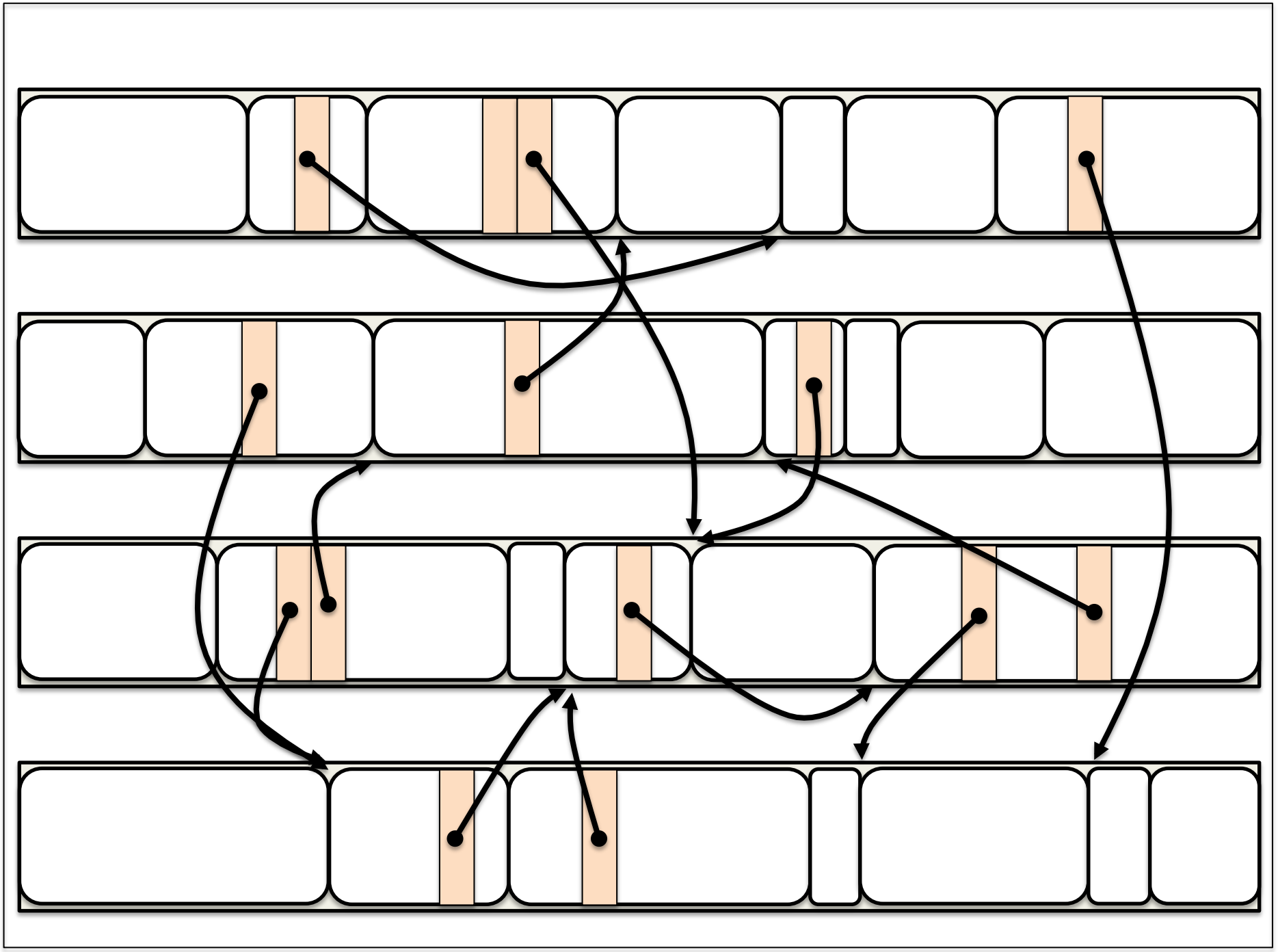


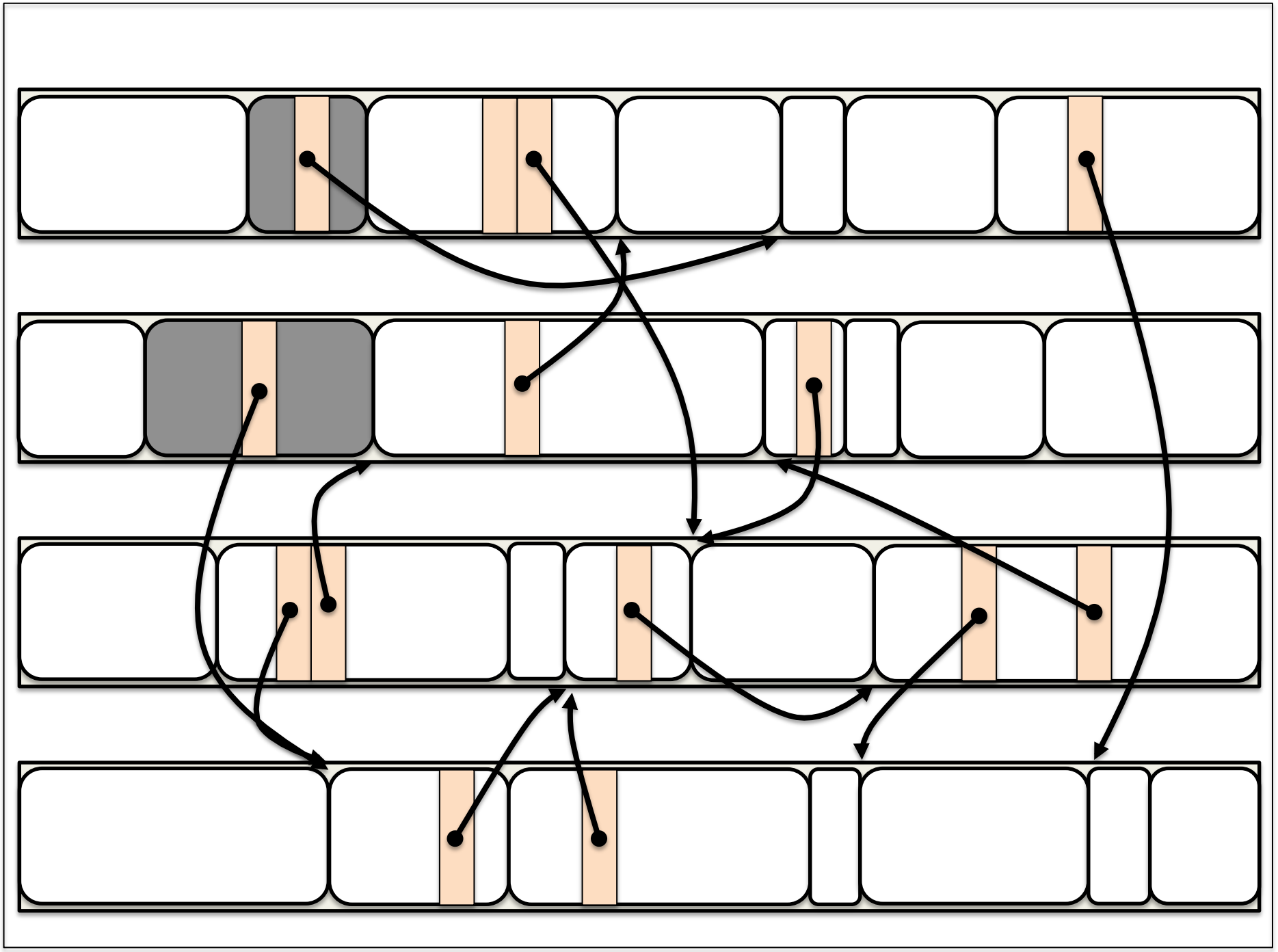


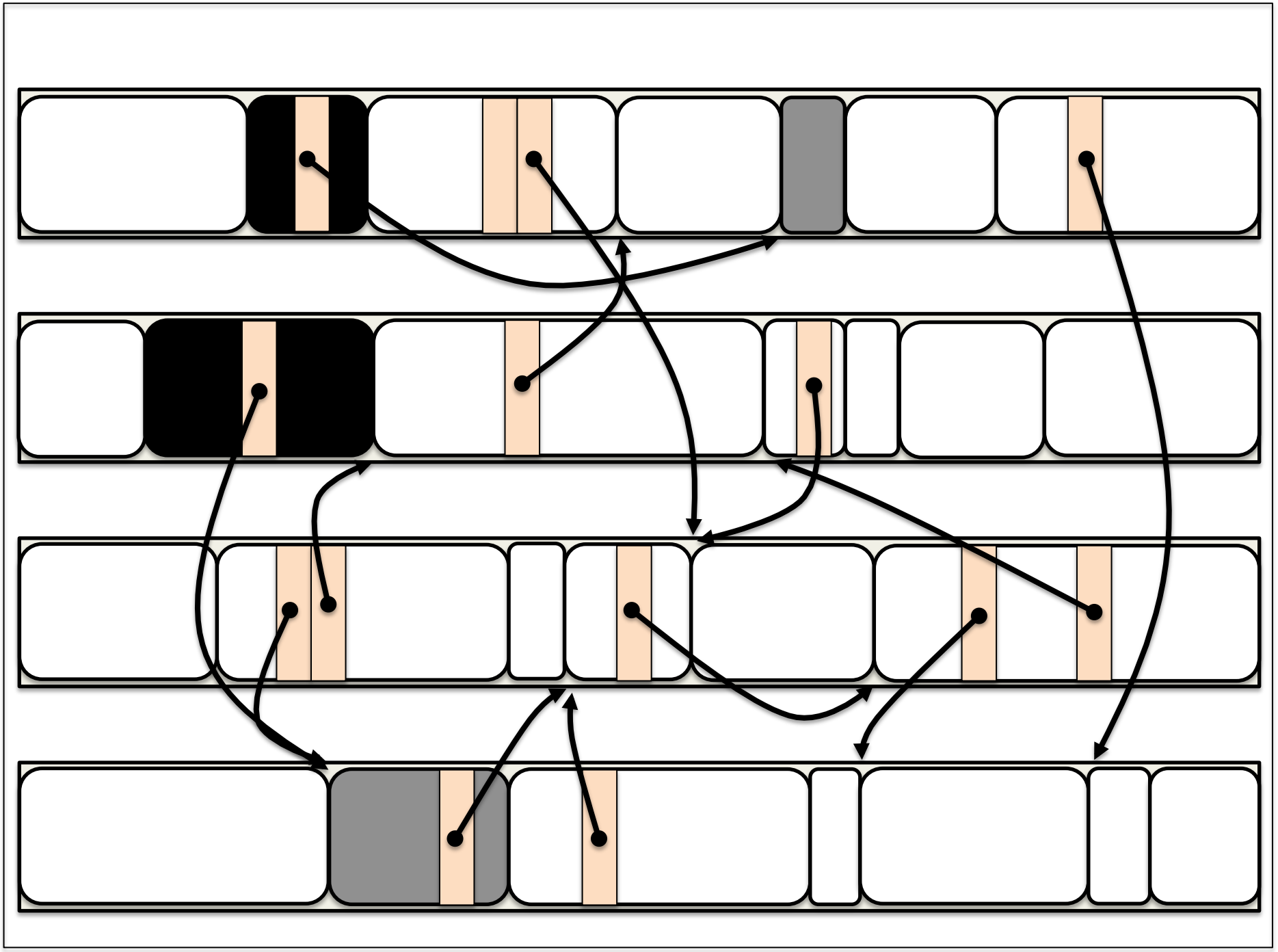


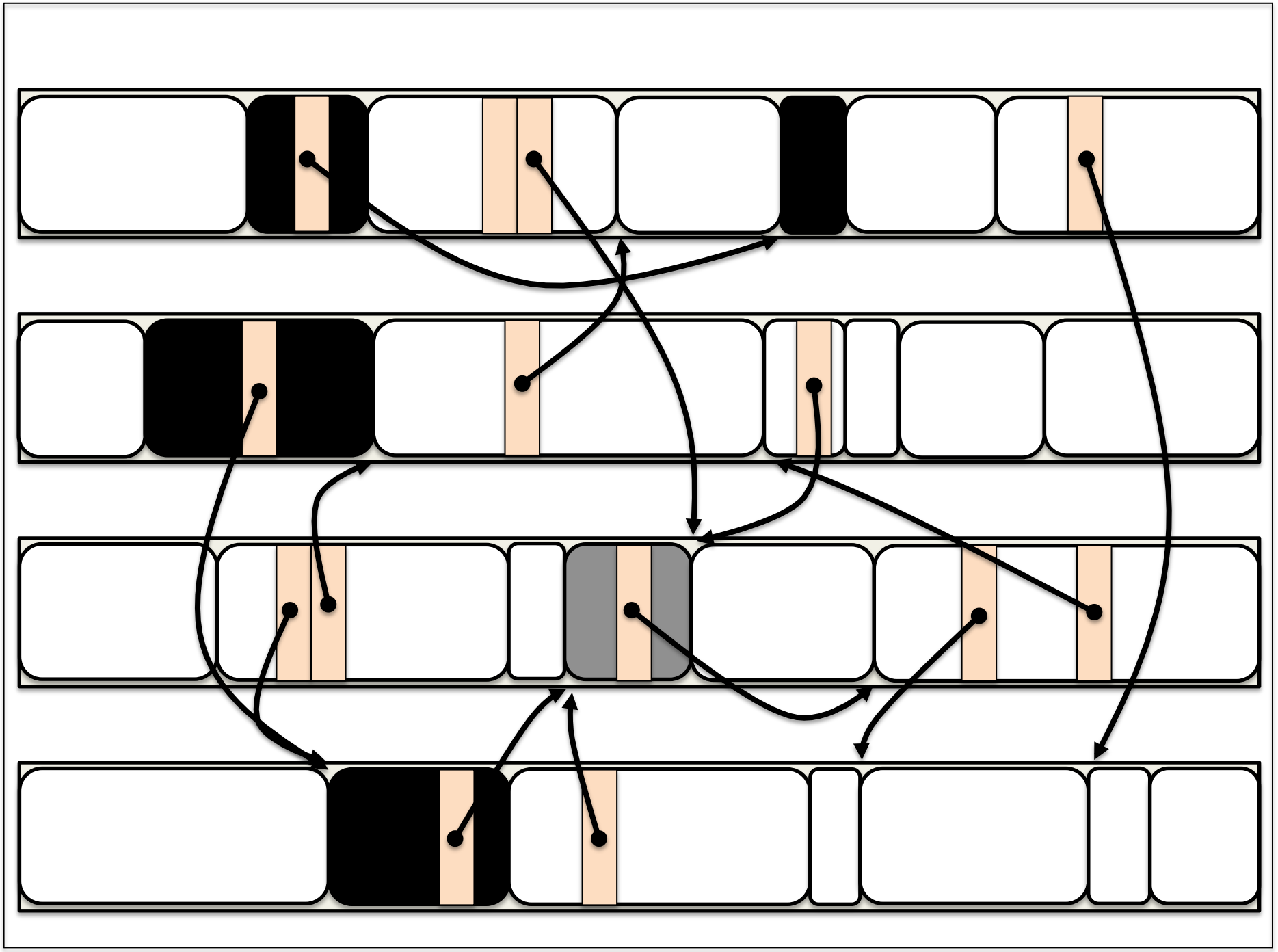
Snapshot At The Beginning

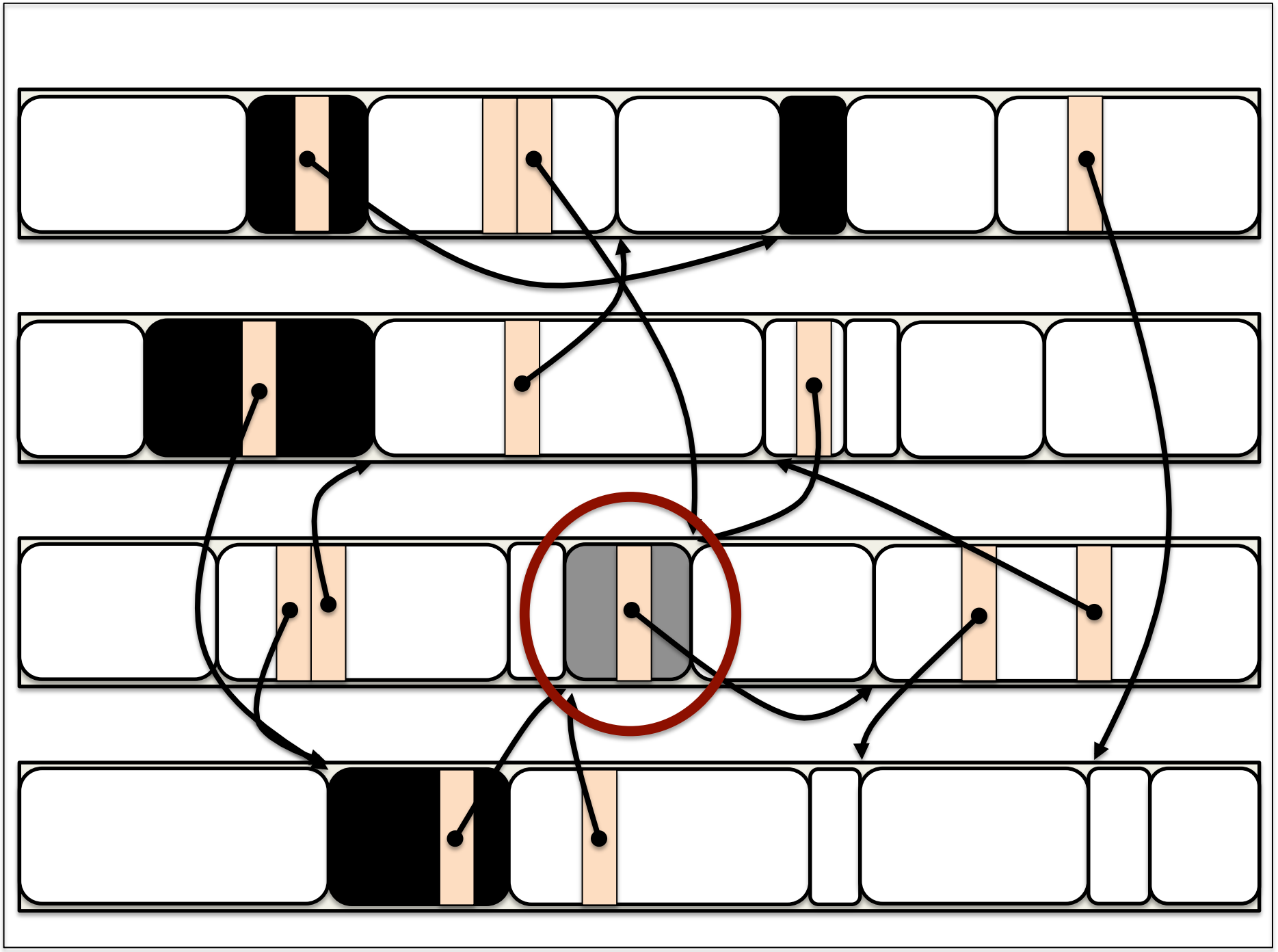
- GC requires that all live objects are marked
  - Does not require that all garbage is unmarked
- Concurrent algorithm works from a point in time
  - Mark all objects that were live at that point
  - Plus all newly-allocated objects
- Implies that all live objects are now marked
  - As well as newly-allocated garbage

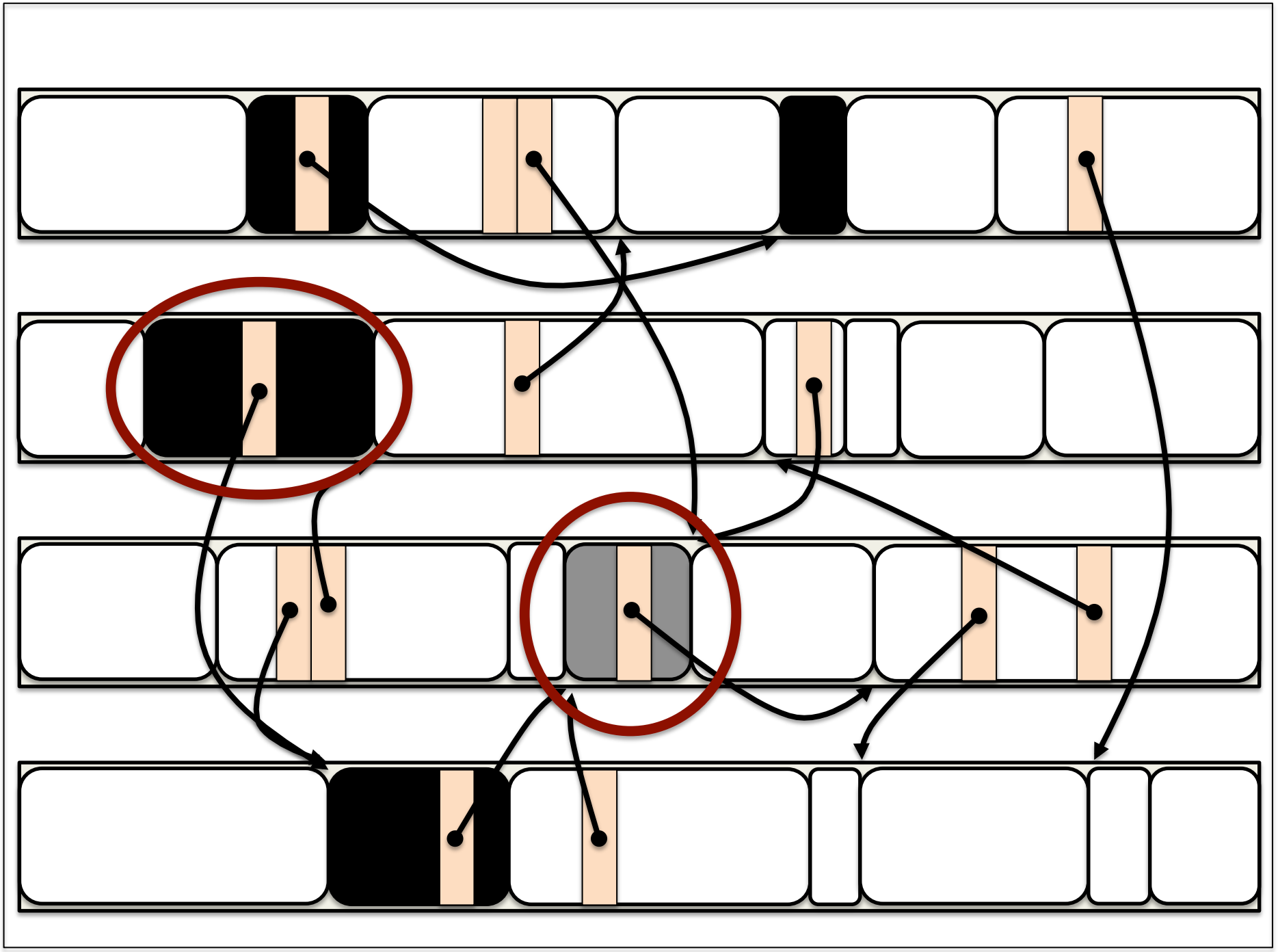


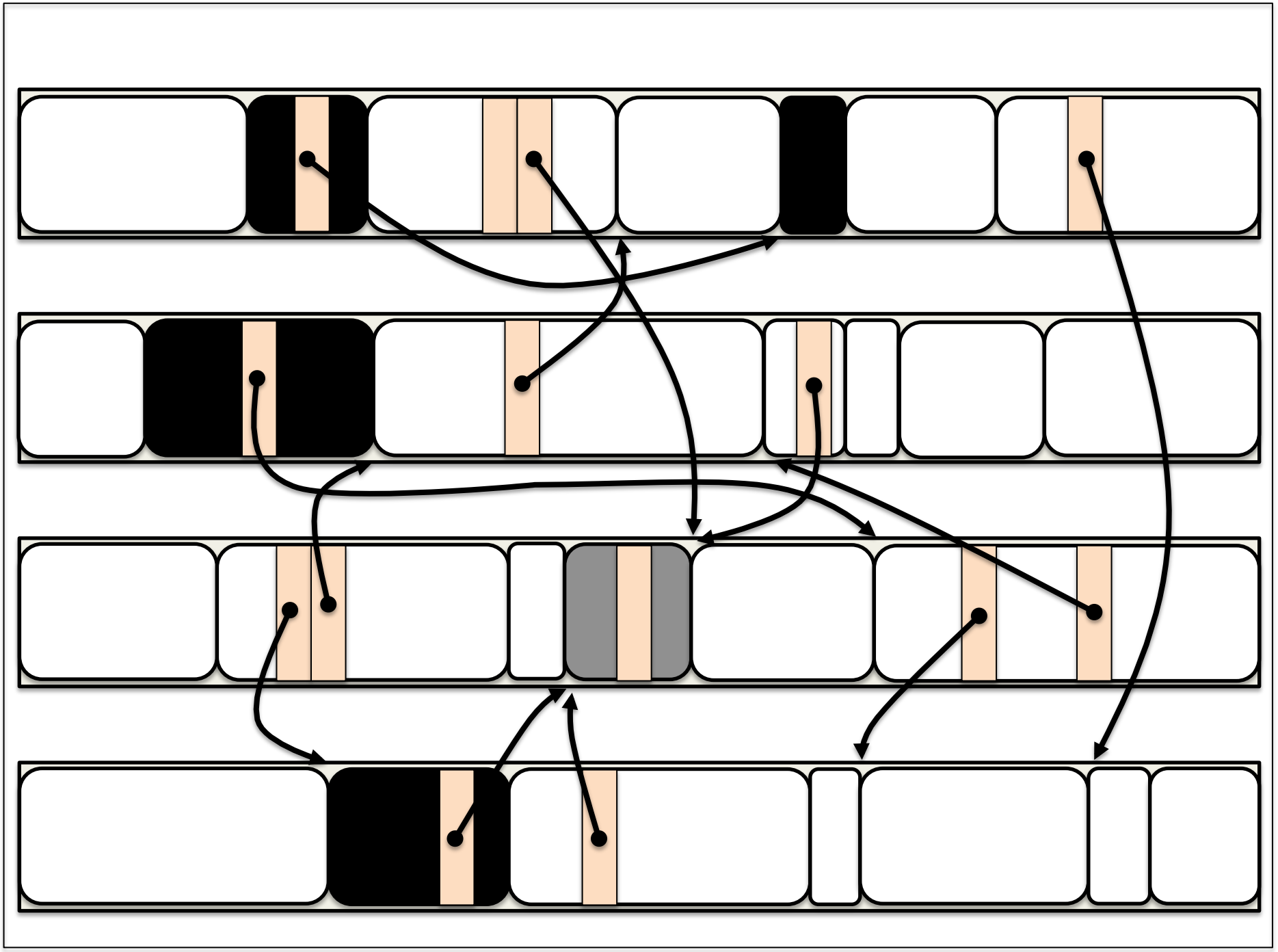


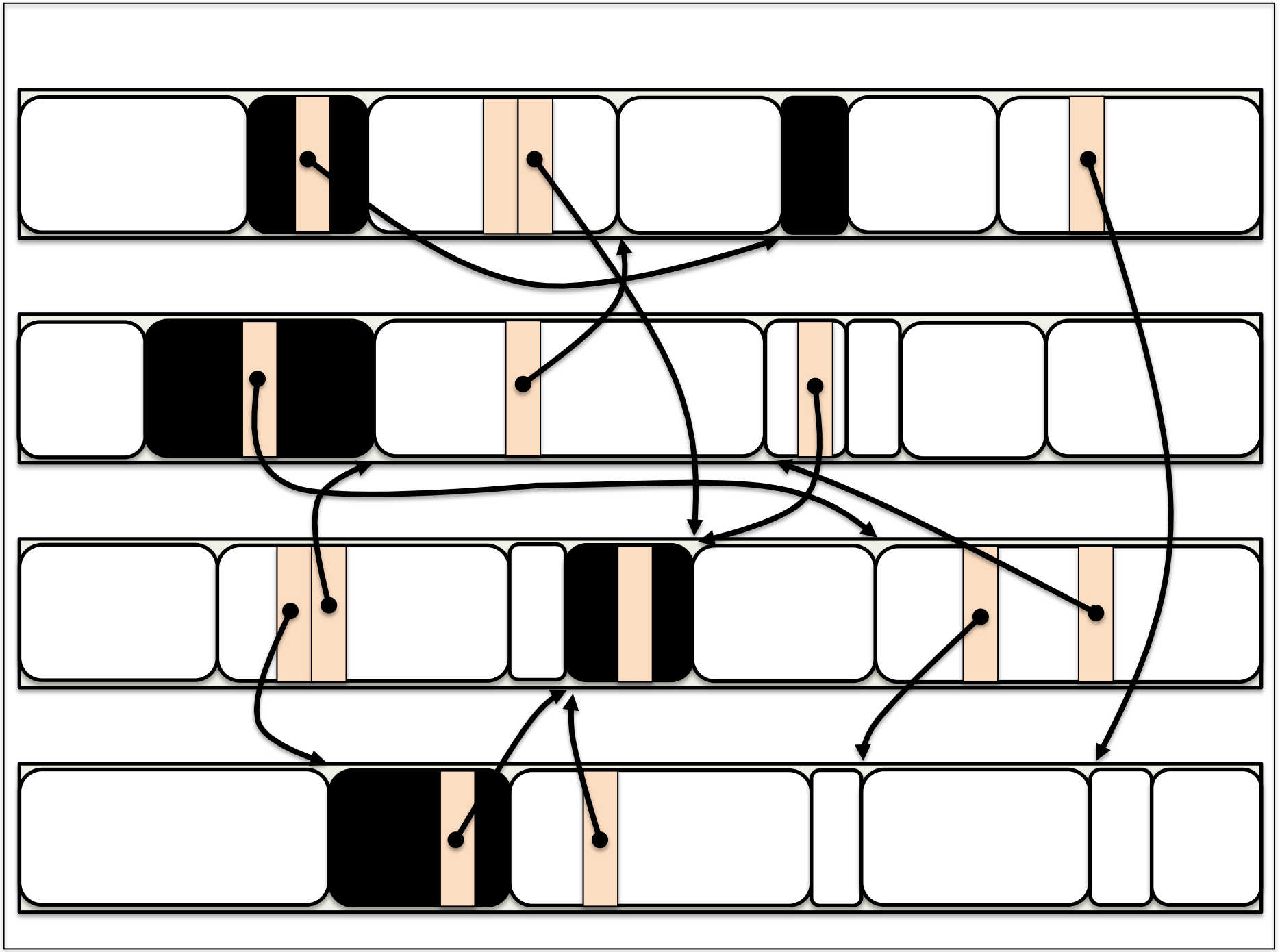


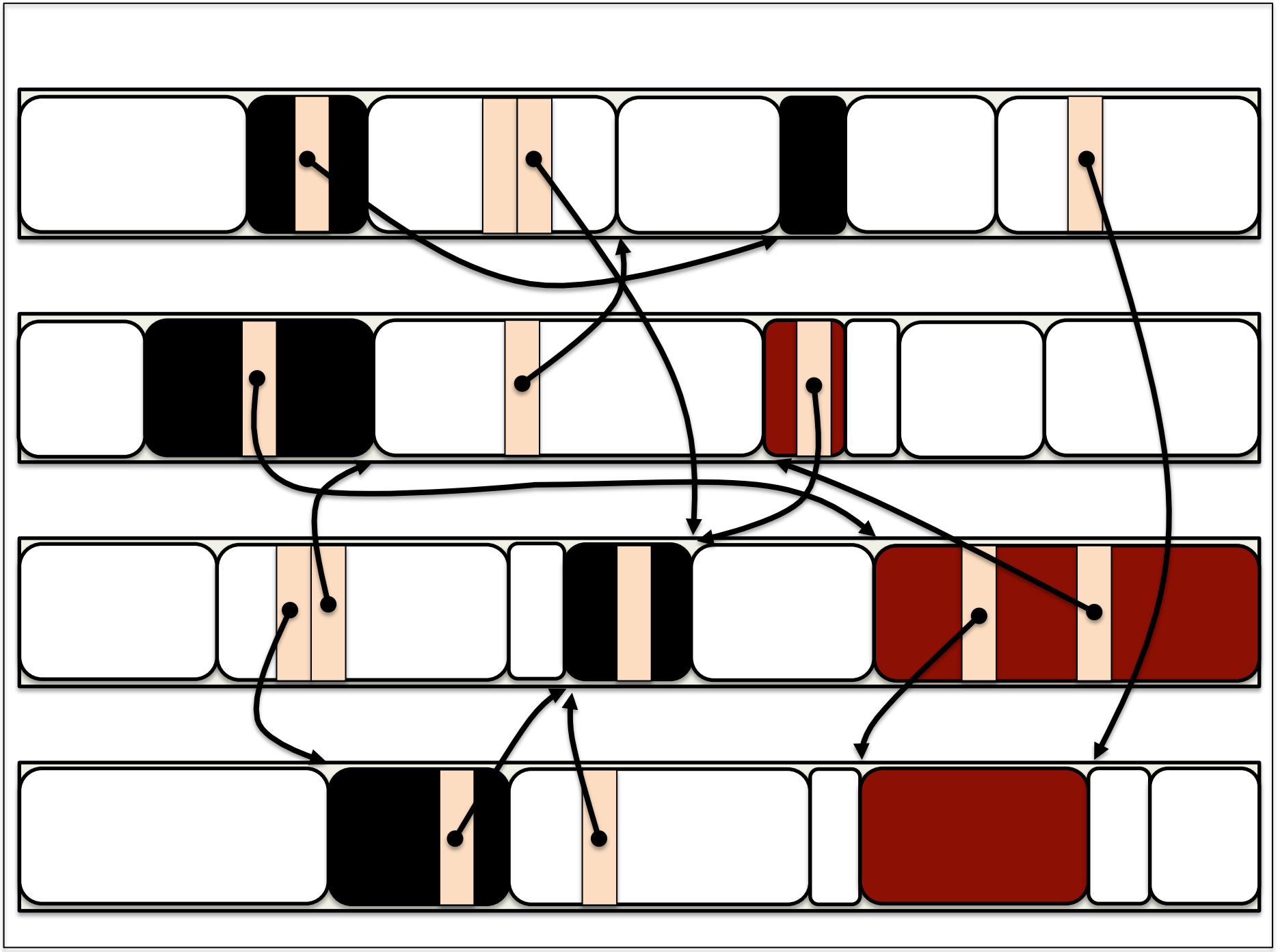










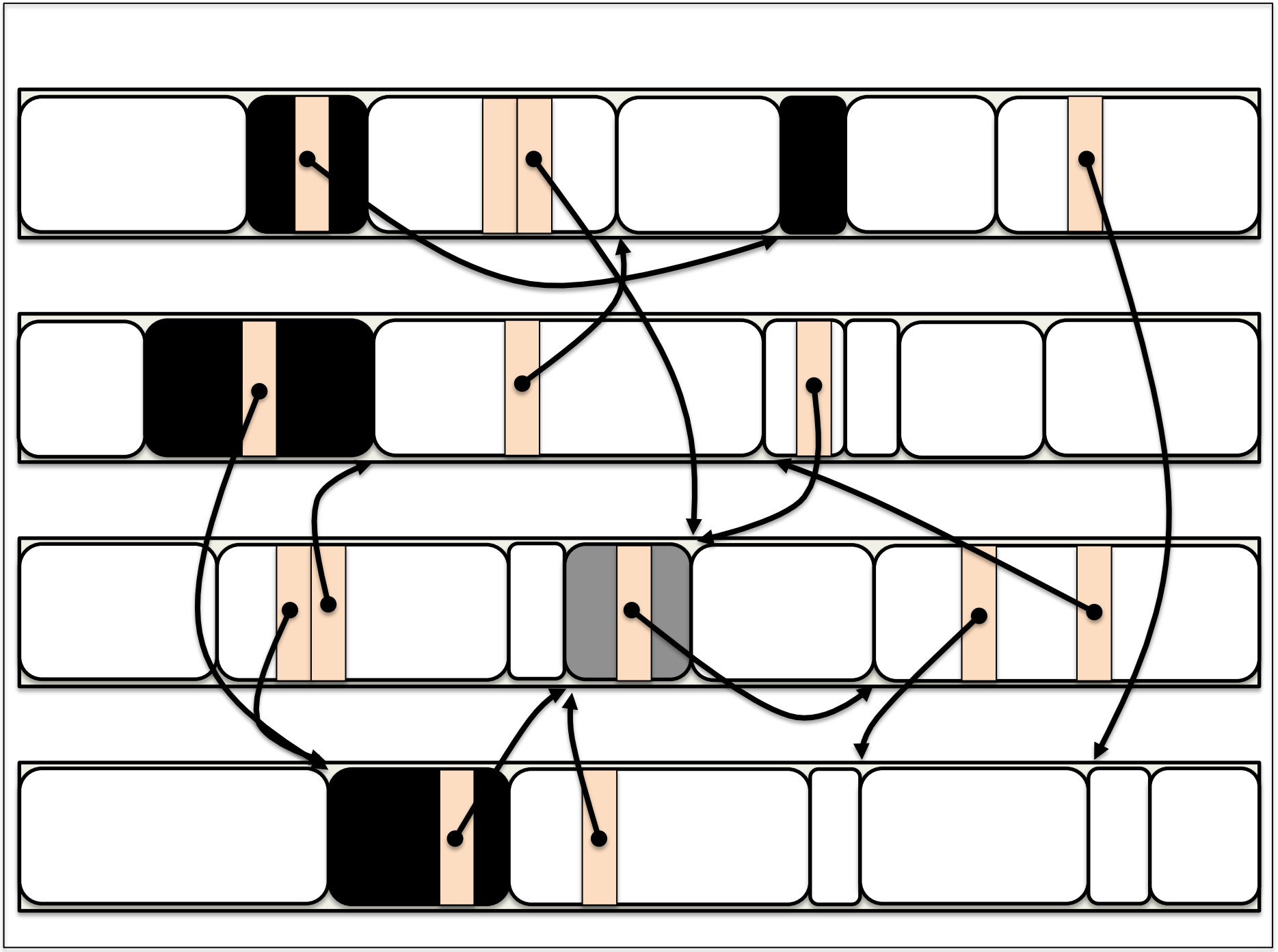


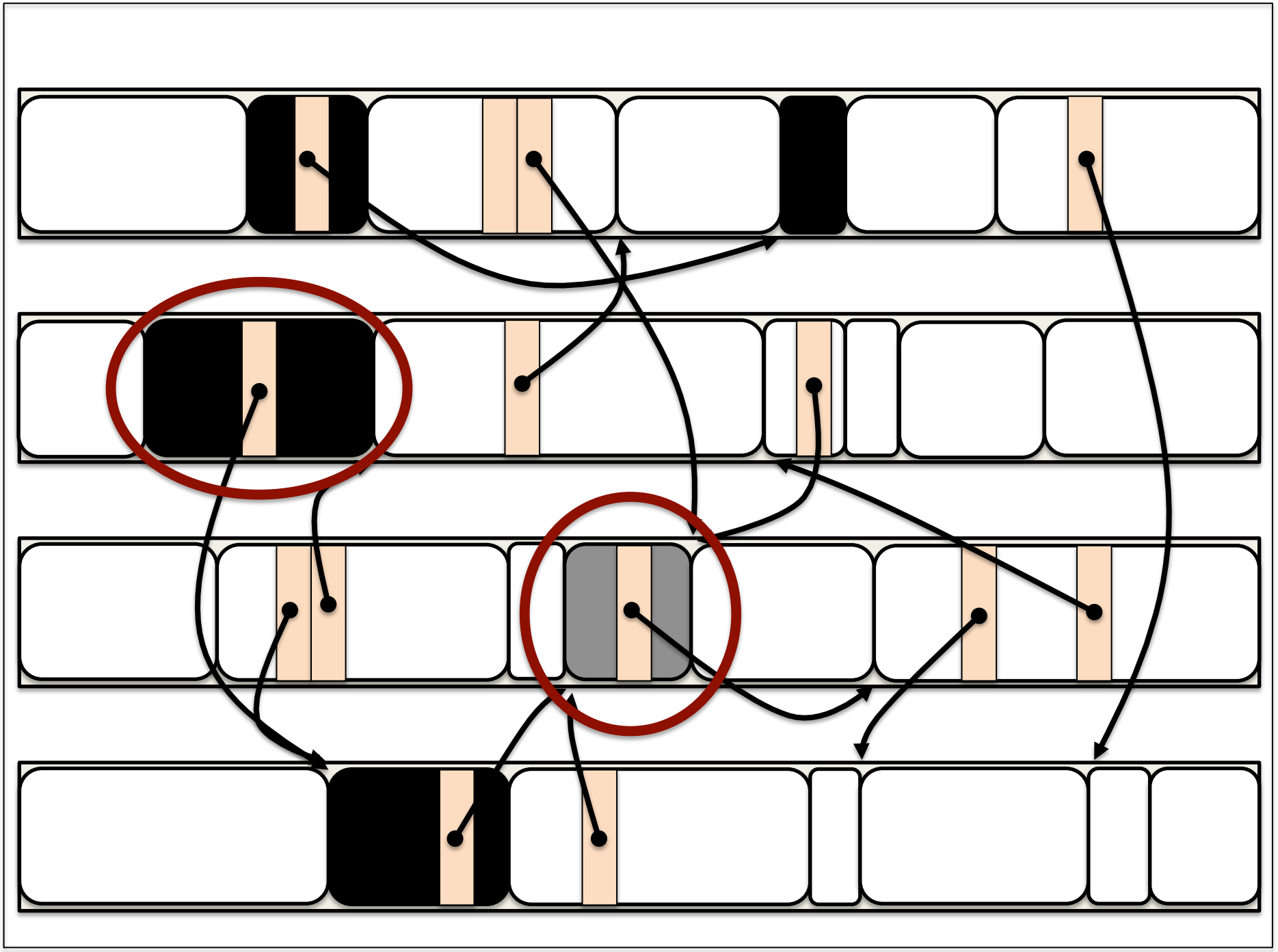
Tracking All Objects

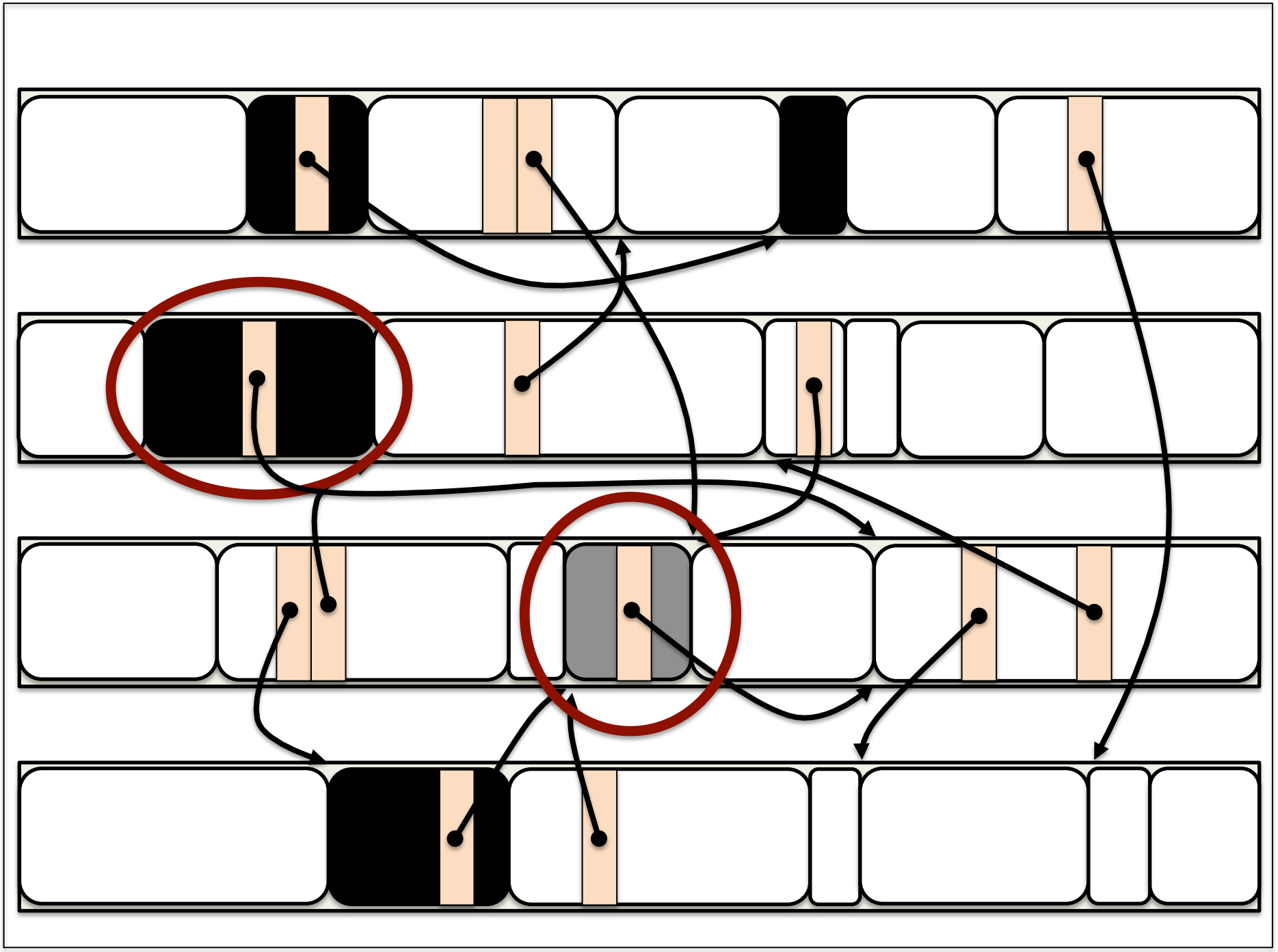
- We can lose an object if:
  - A white object reference is stored in a black object
  - All paths from gray objects to white are removed
- We can consider stacks to be black in this case

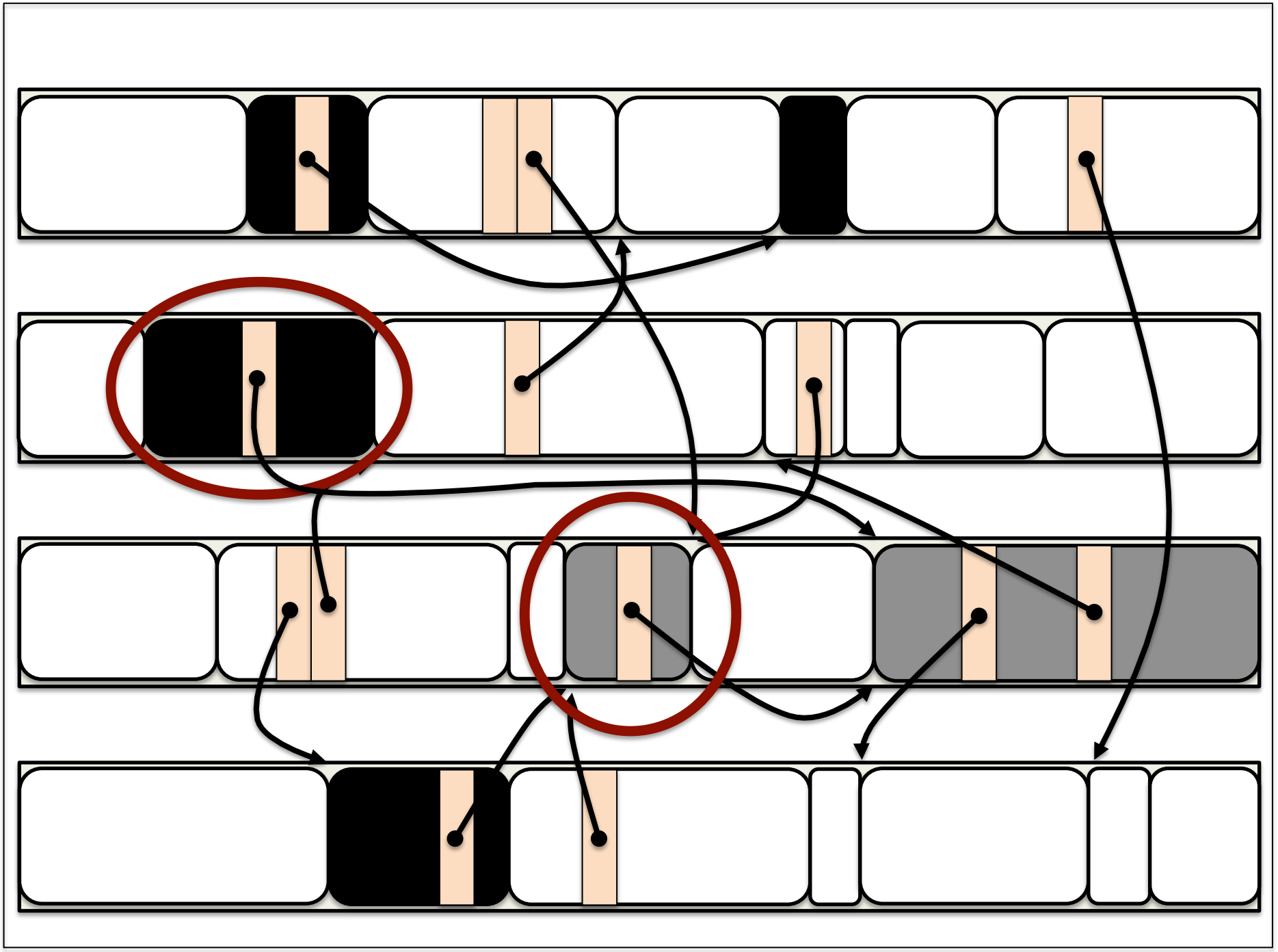
- Fix this by intercepting writes to gray objects
  - Check if the target is white
  - If so, set it to be gray

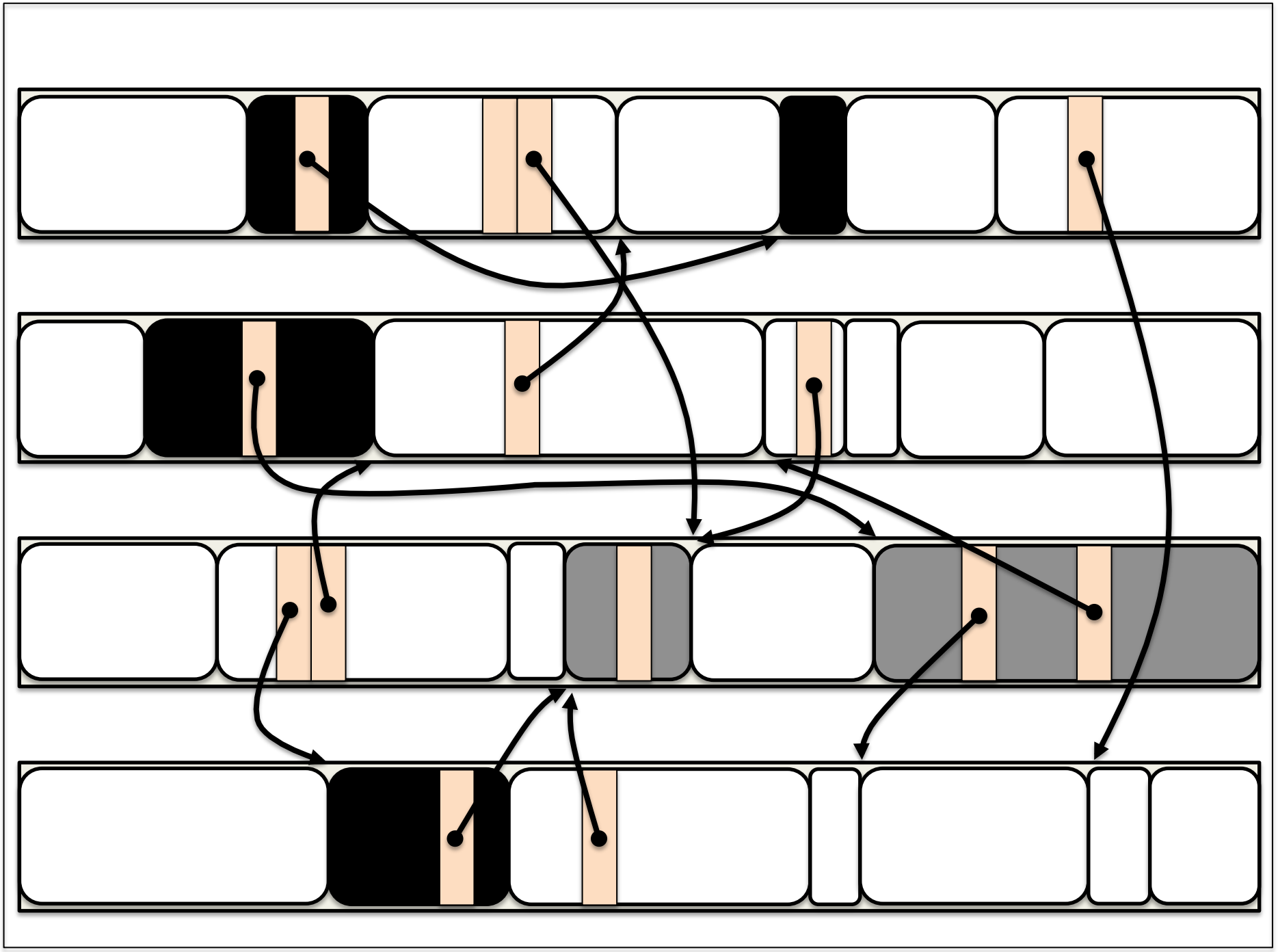
- Implement using a write barrier

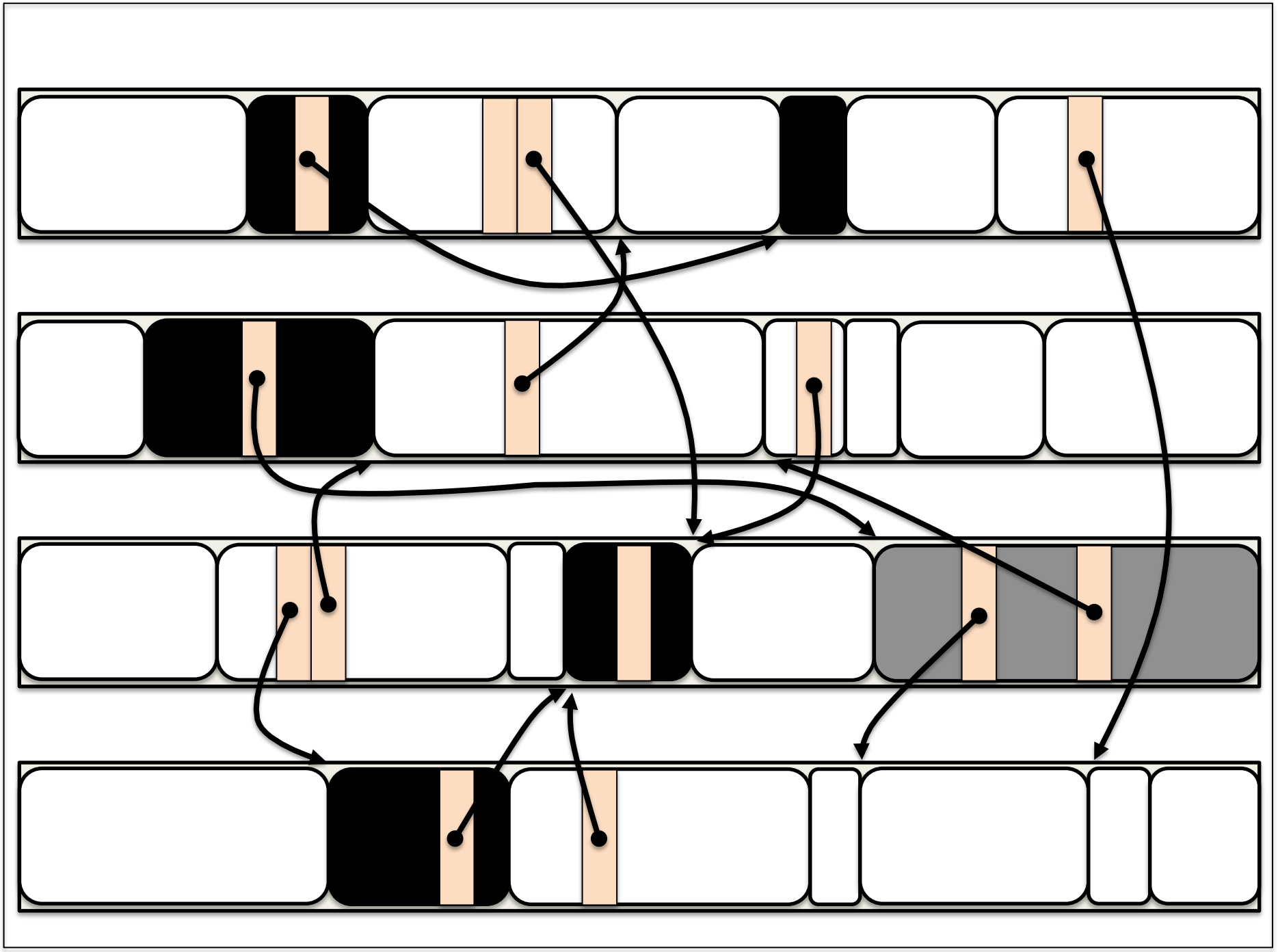


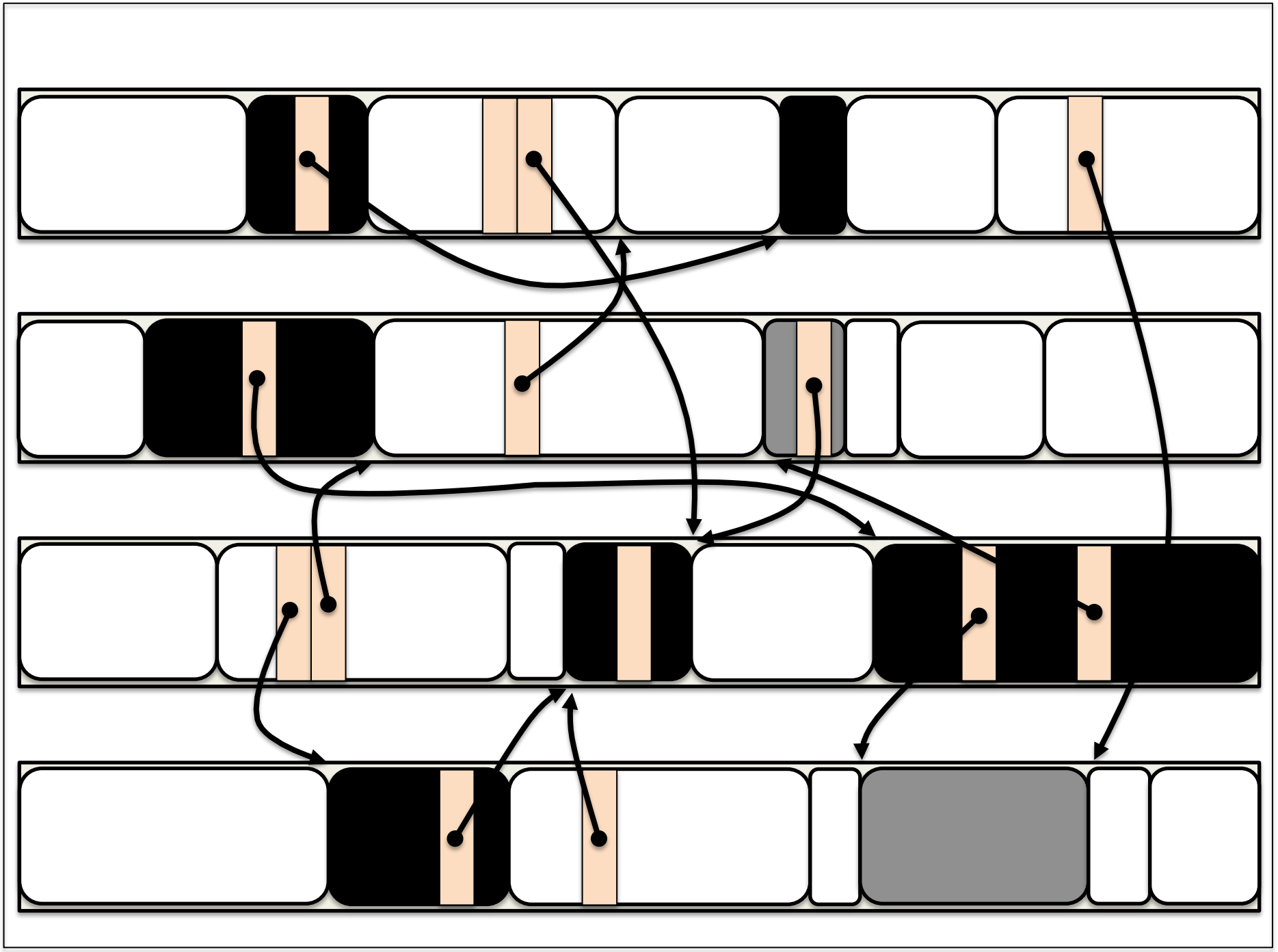


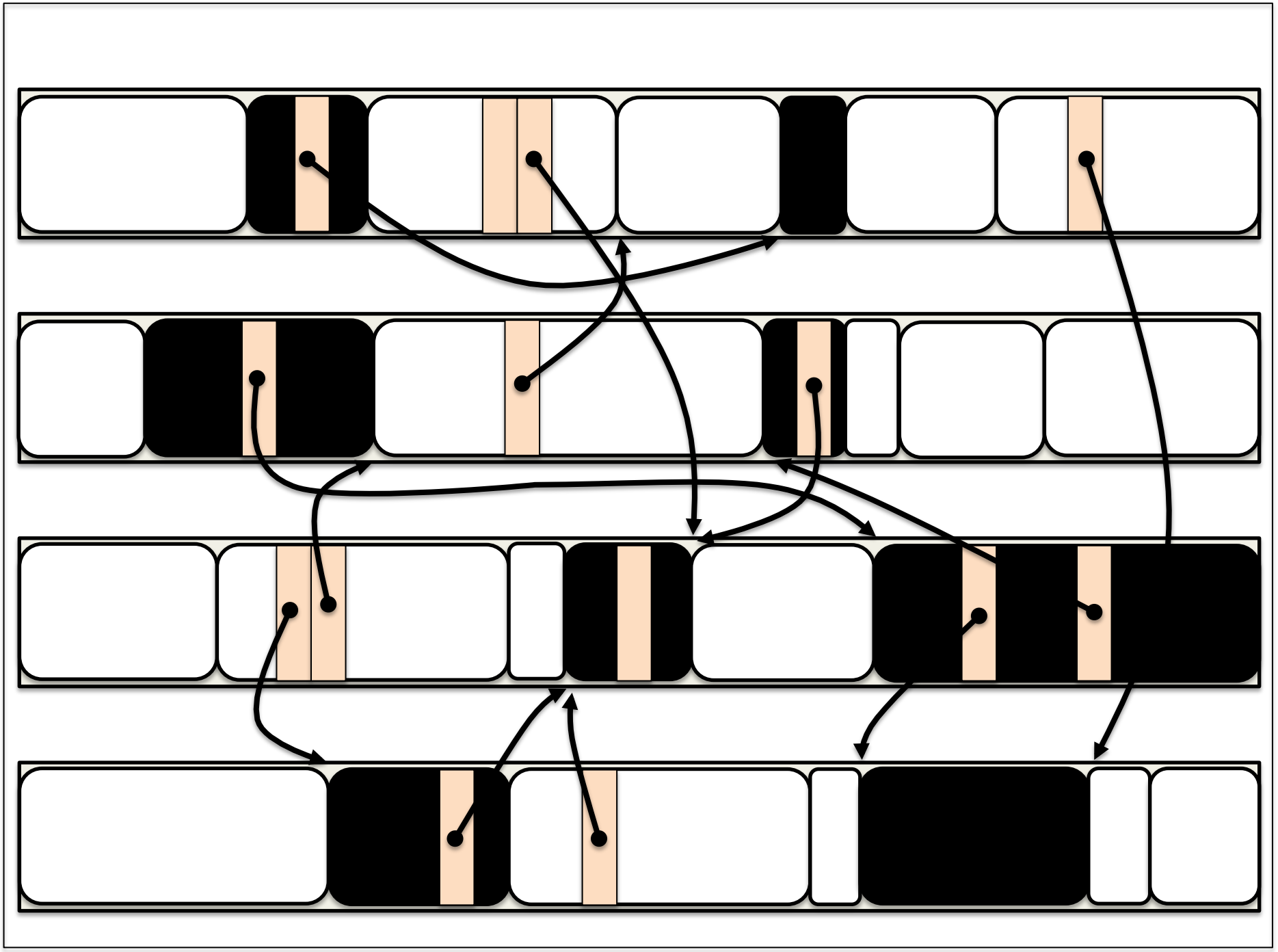












SATB Write Barrier

- If the source object is gray
  - And the target object is white
    - Add the target object to the gray stack

Allocating Objects

- We have three choices for newly-allocated objects
  - Allocate white
  - Allocate gray
  - Allocate black
- New objects contain no references
  - Recall the default values for fields
  - They are already scanned
- For this algorithm, we allocate objects as black

Snapshot At The Beginning

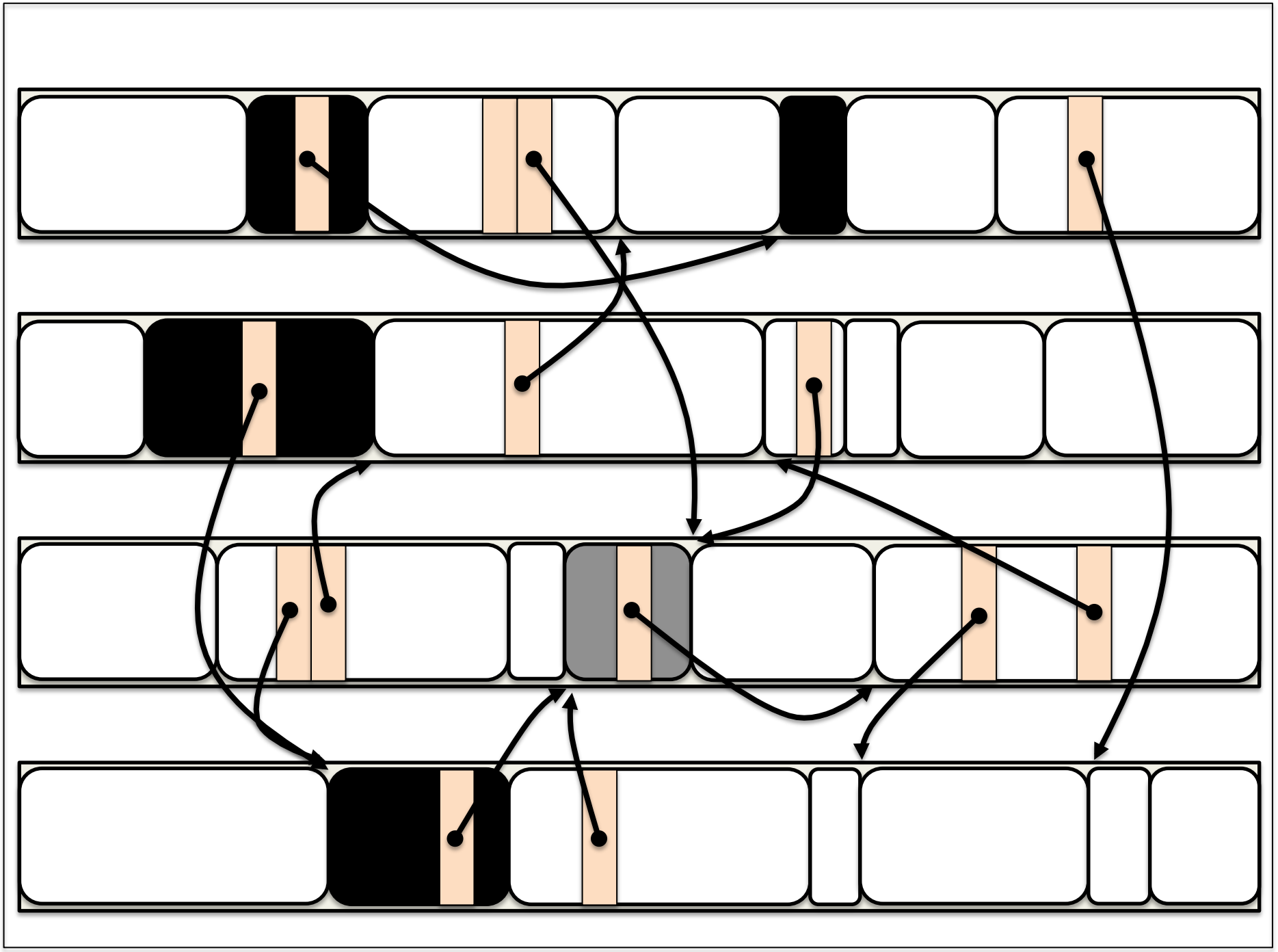
- Concurrent algorithm
  - Very short initial pause to scan the root set
  - All other work happens concurrently
- Limited amount of scanning
  - Once an object is black, it never changes
- Conservative
  - Retains all newly-allocated objects
  - Reduces benefit of generational collection

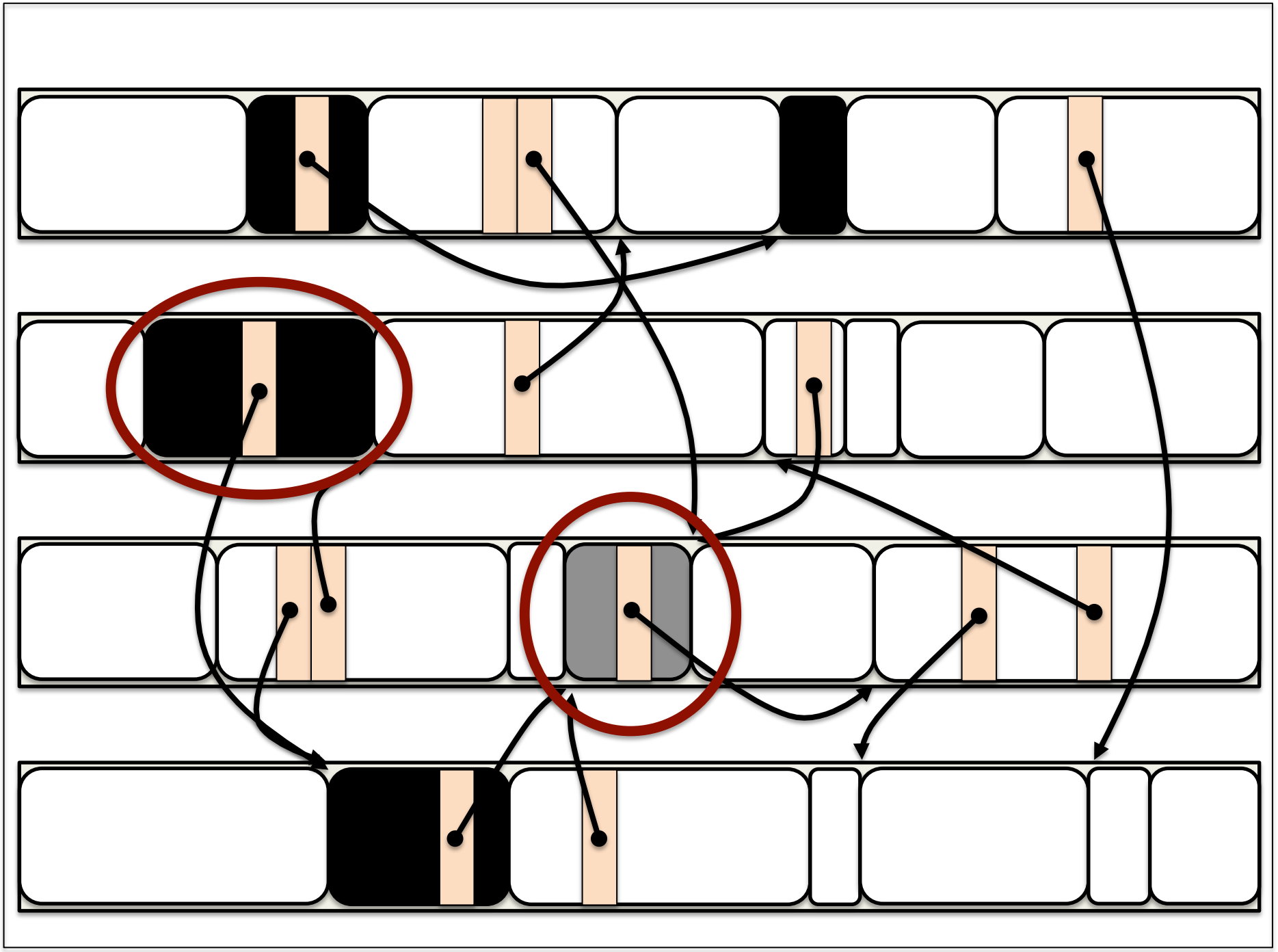
Incremental Update

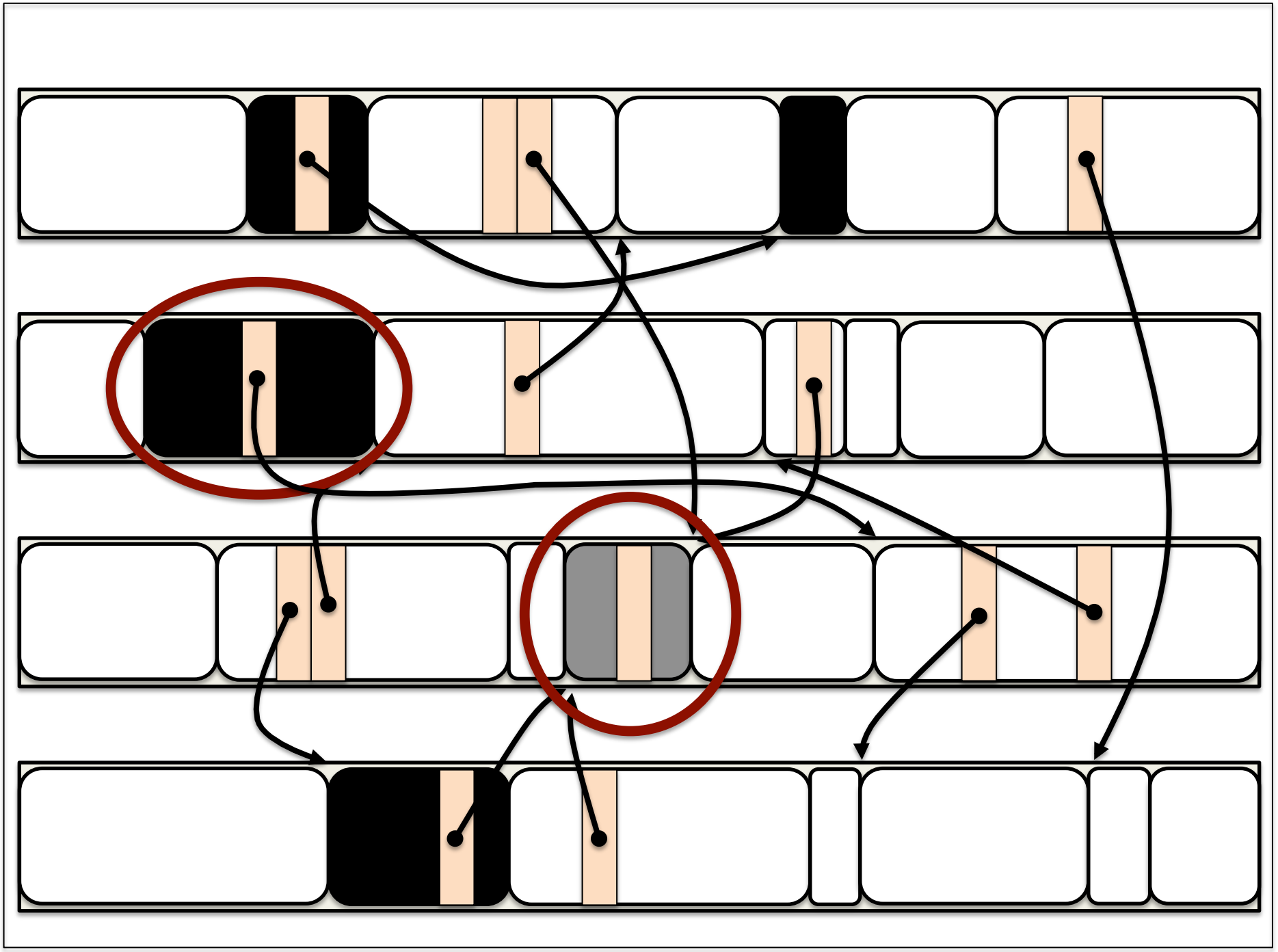
- Reduces the conservatism of SATB
- Takes the opposite approach
  - Tries to mark objects alive at the end of the phase
  - New objects that are unreachable should not be marked
- Heuristic approach – may mark some garbage
  - Generally far less than SATB

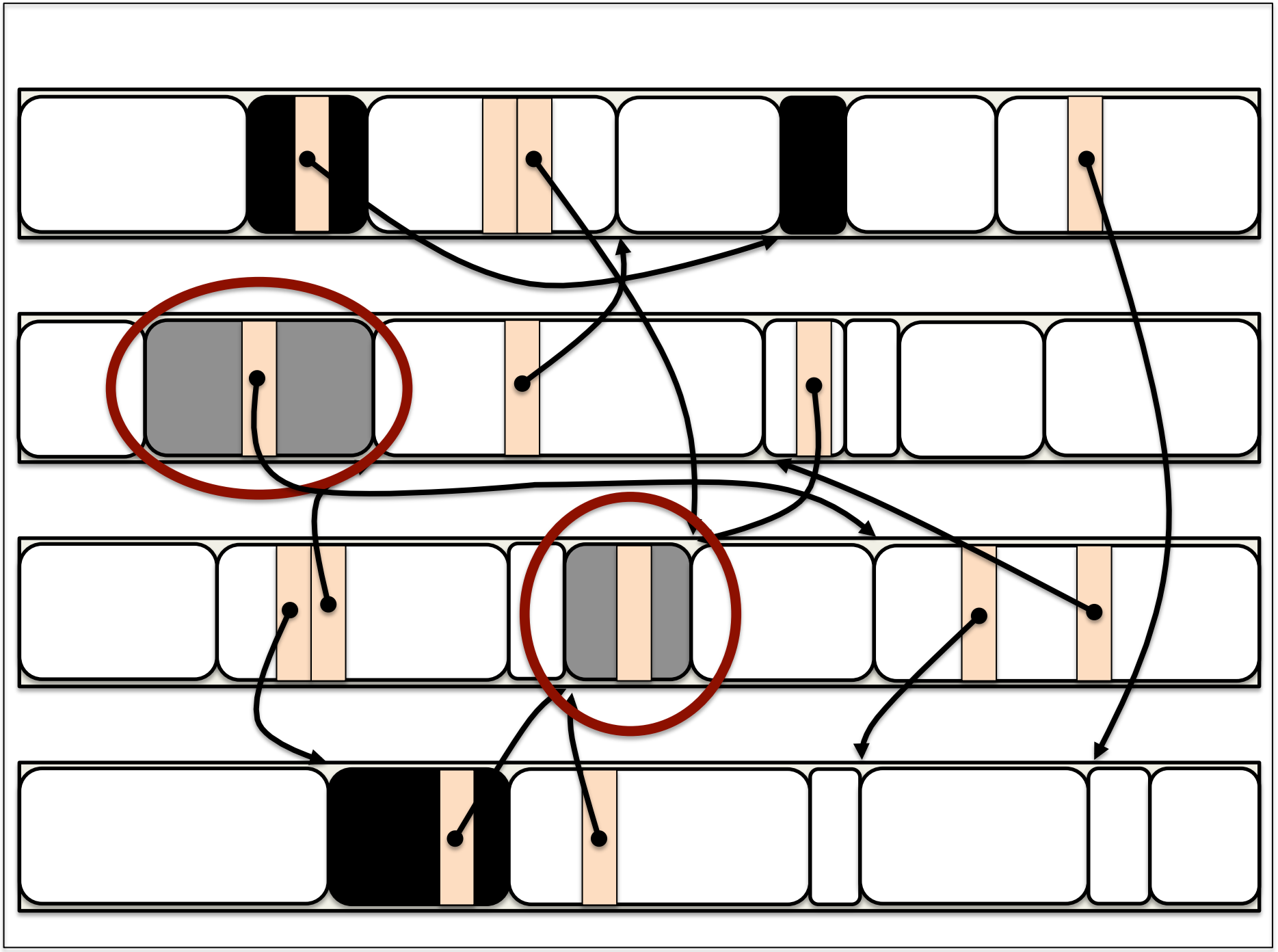
Incremental Update

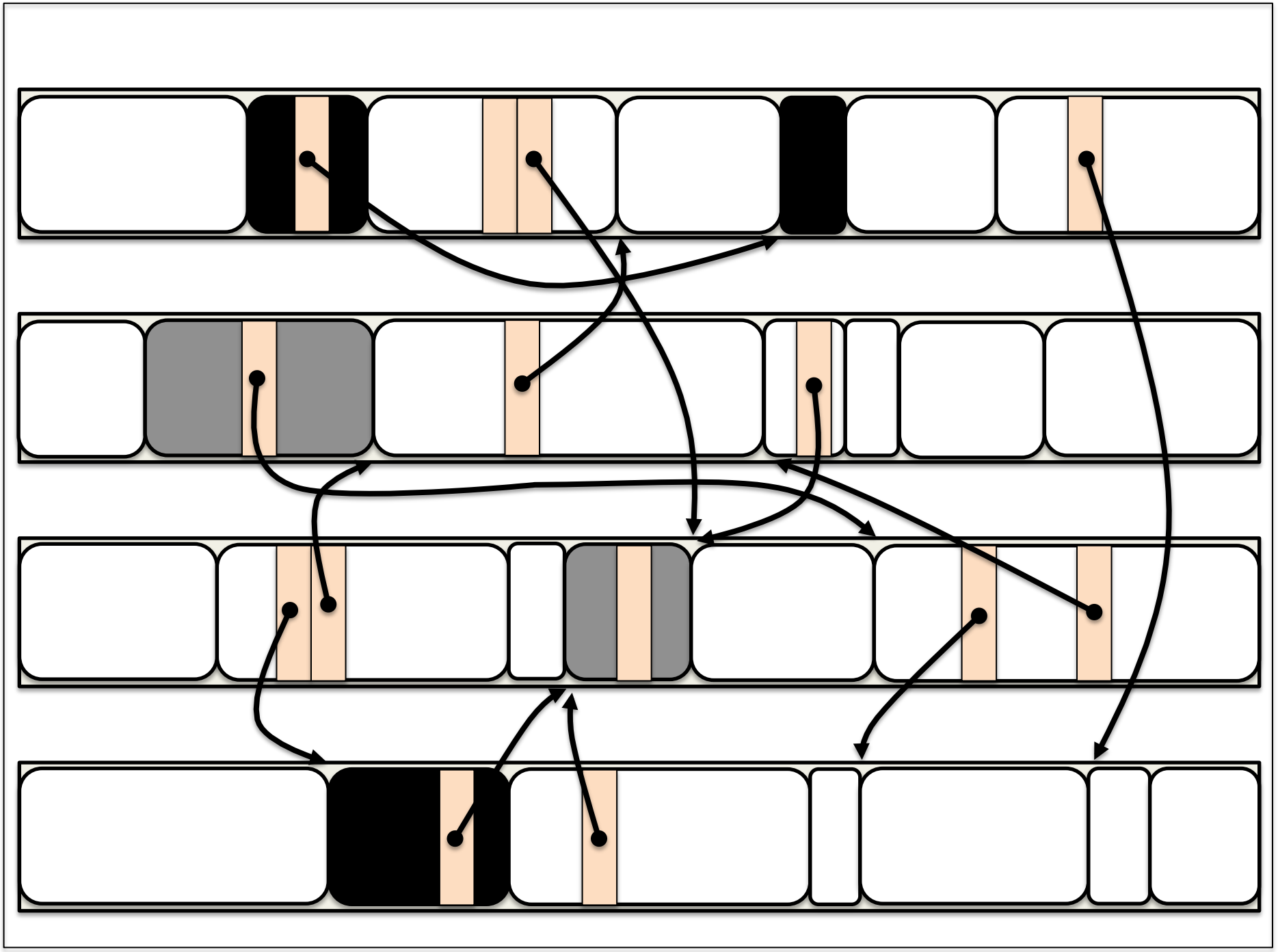
- Maintains the same invariant
  - Black objects must not point to white objects
- Captures writes to black objects
  - If the target is gray, do nothing
  - If the target is white, turn the black object gray
- Eventually the black object is re-scanned
  - The reference to the white object is located
  - If the reference has been overwritten, it is not marked

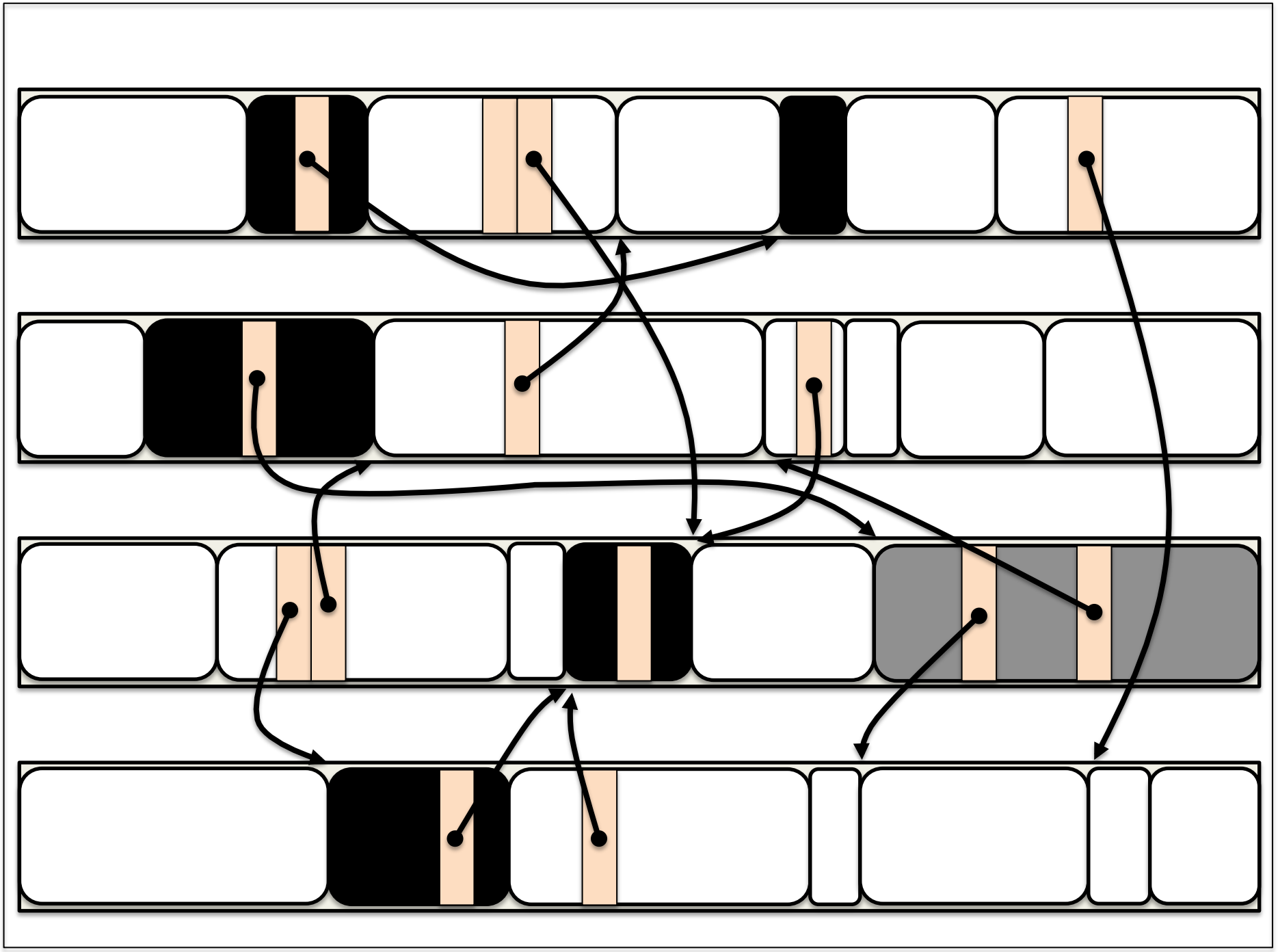


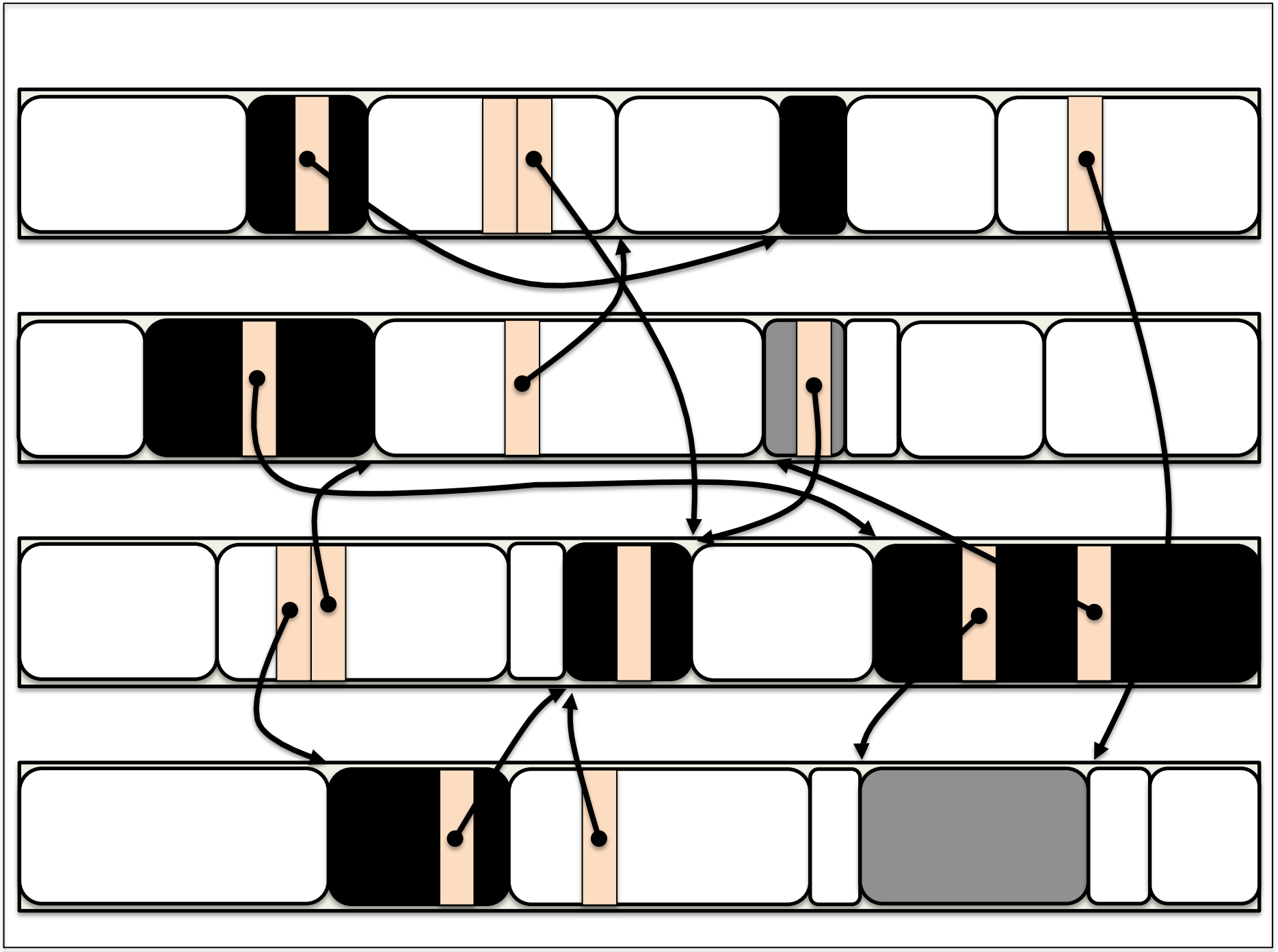


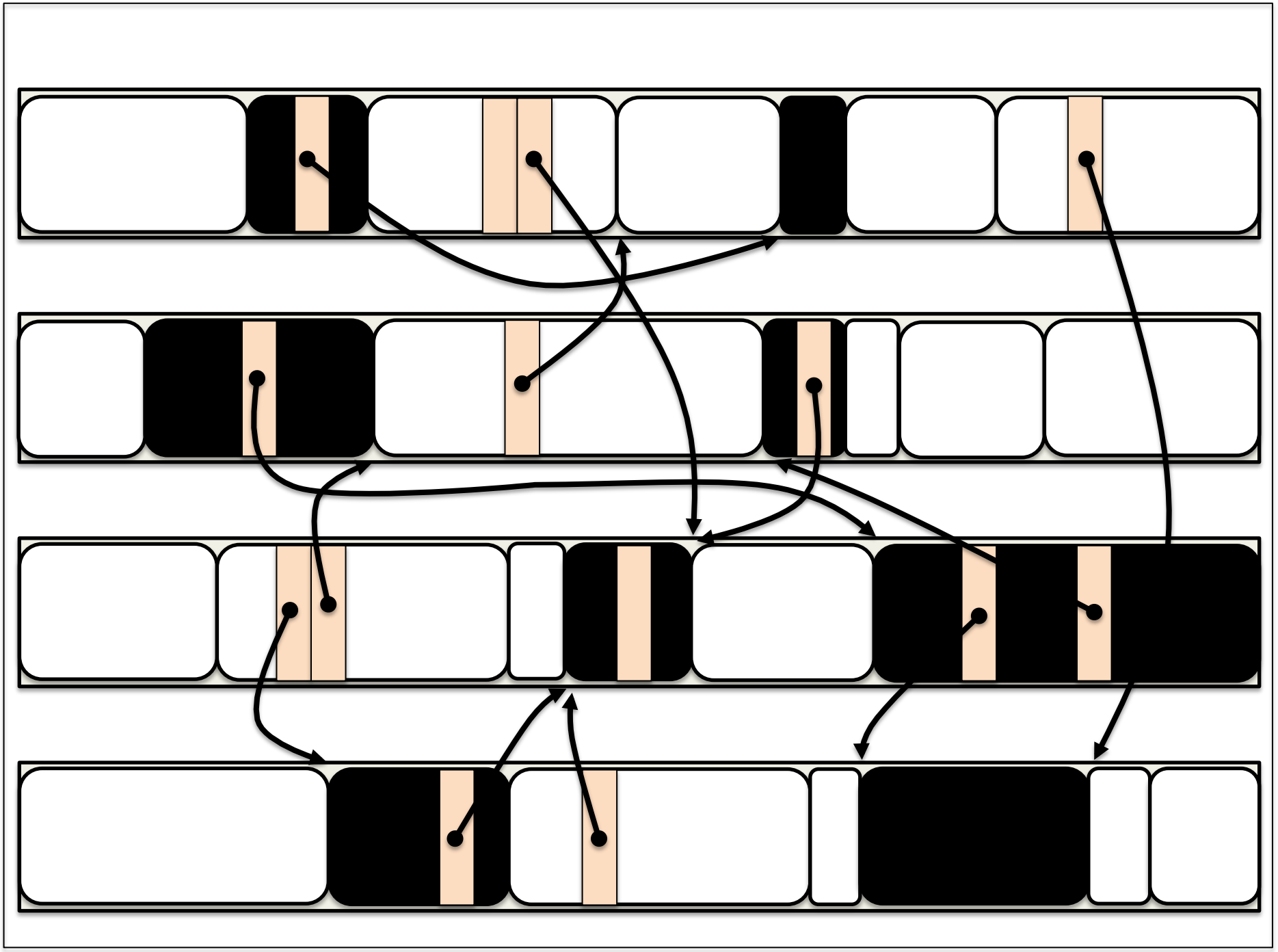












Incremental Update Barrier

- If the target object is white
  - And the destination object is black
    - Add the destination to the gray stack

Allocation

- Objects in Incremental Update allocated white
  - Object marked if it is stored in a black object
  - If object is never reachable, it is never marked
- Has potential to skip a lot of garbage
  - Most objects are never referred to from the heap
- Concurrent mark maintain generational behavior

Remark Phase

- Must re-scan roots when the mark is over
  - Write barrier tracks updates to the heap
  - No write barrier on stack or local writes
  - Only reference to a white object may be in a root
- Algorithm is iterative
  - A single object may turn gray and black many times
  - Remark phase may trigger more changes
- Algorithm is guaranteed to terminate
  - Citation available on request

Incremental Update

- Less conservative than Snapshot At The Beginning
  - Allocating white retains less garbage
  - White pointers may be overwritten before scanning
- Implementation is considerably trickier
  - Need to iterate over root scanning
- Mark phase may last longer
  - Write barriers must be active for longer

Concurrent Write Barriers

- Concurrent marking barriers have overhead
  - Need to check the mark state of source and target
- More expensive than basic generational barrier
  - Runs at the same time as the mutator
  - Must be active for all threads, not just GC
- Want to be able to turn off barriers at times
  - Use the expensive barrier only during marking

Phased Write Barrier

- If the source object is gray
  - And the target object is white
    - Add the target object to the gray stack

Phased Write Barrier

- If the system is currently marking
  - And the source object is gray
    - And the target object is white
      - Add the target object to the gray stack

Changing Write Barrier State

- We know from last week that is can be a race
  - The GC state is changed by one thread
  - Another thread doesn't see the update
- State change requires a safepoint
  - Safepoint ensures that there is a memory fence
  - Each thread gets the latest value of the state variable
- Safepoint operation is very lightweight
  - Don't need to synchronize with all other threads

Incremental Marking

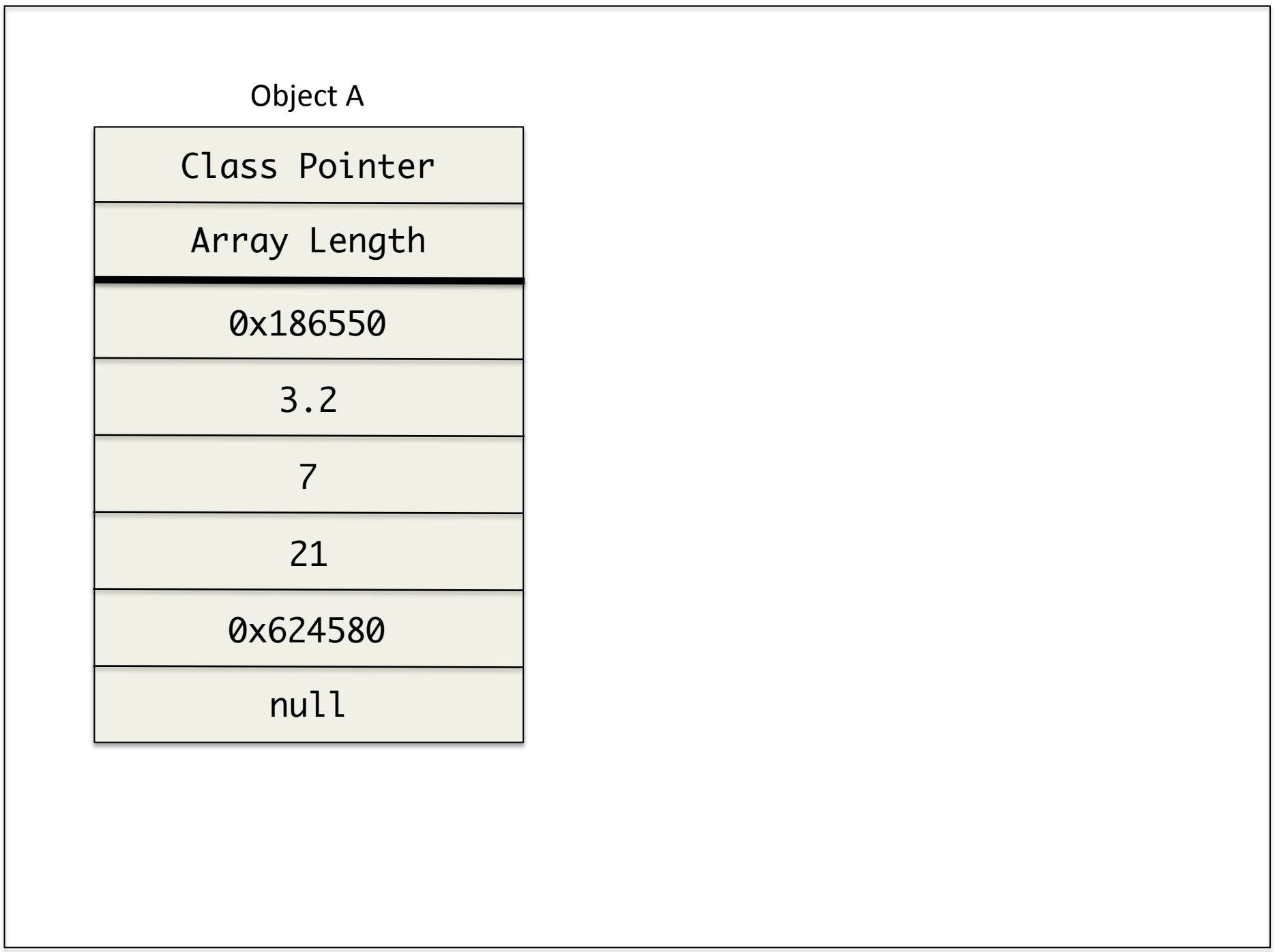
- SATB lets us mark an evolving object graph
  - Nothing requires that the algorithm be concurrent
- We can easily change concurrent to incremental
  - Do a little bit of work every so often
  - Ideal time is during allocation
- Spread the work out without adding threads

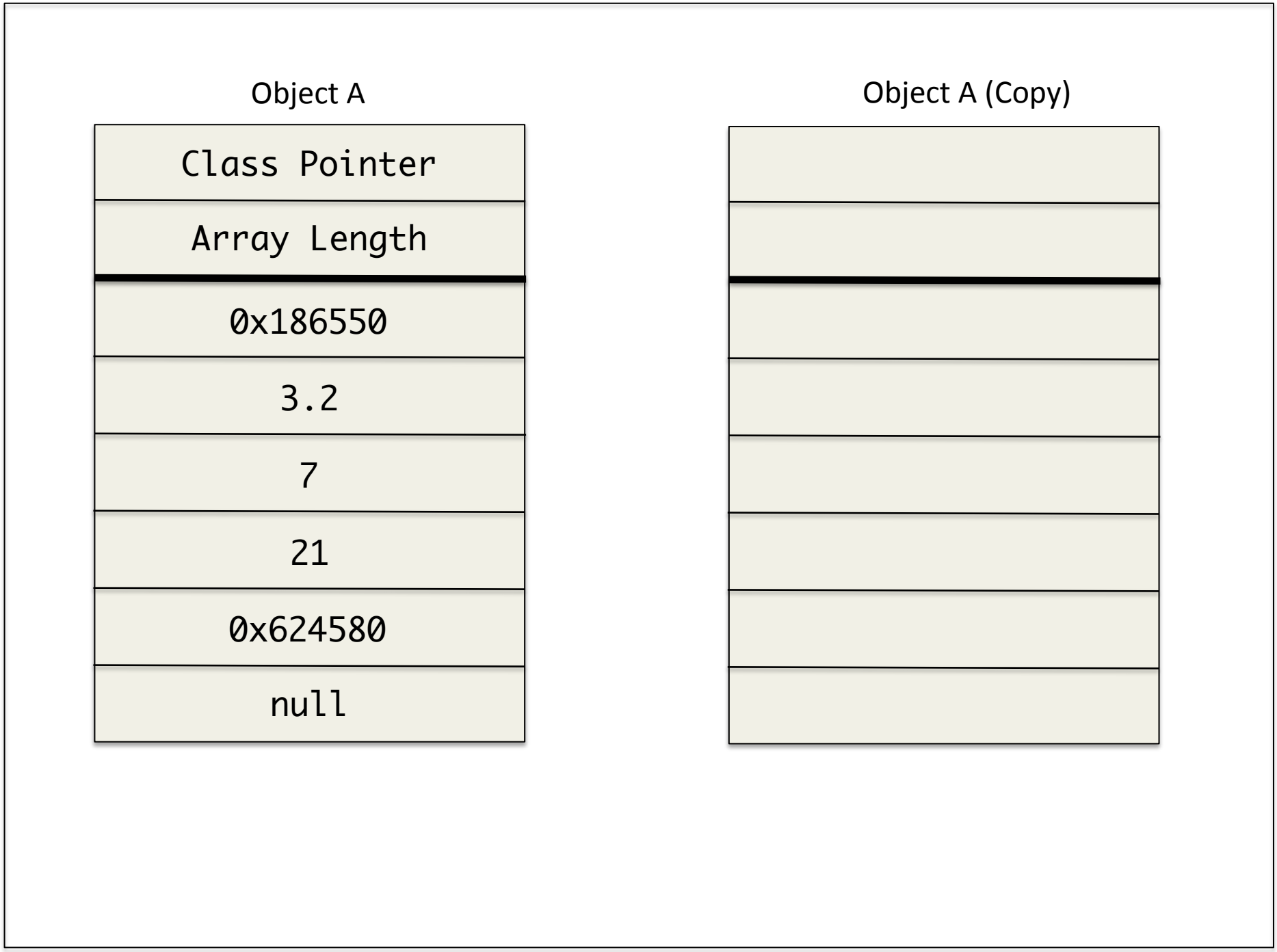
Concurrent Mark Sweep

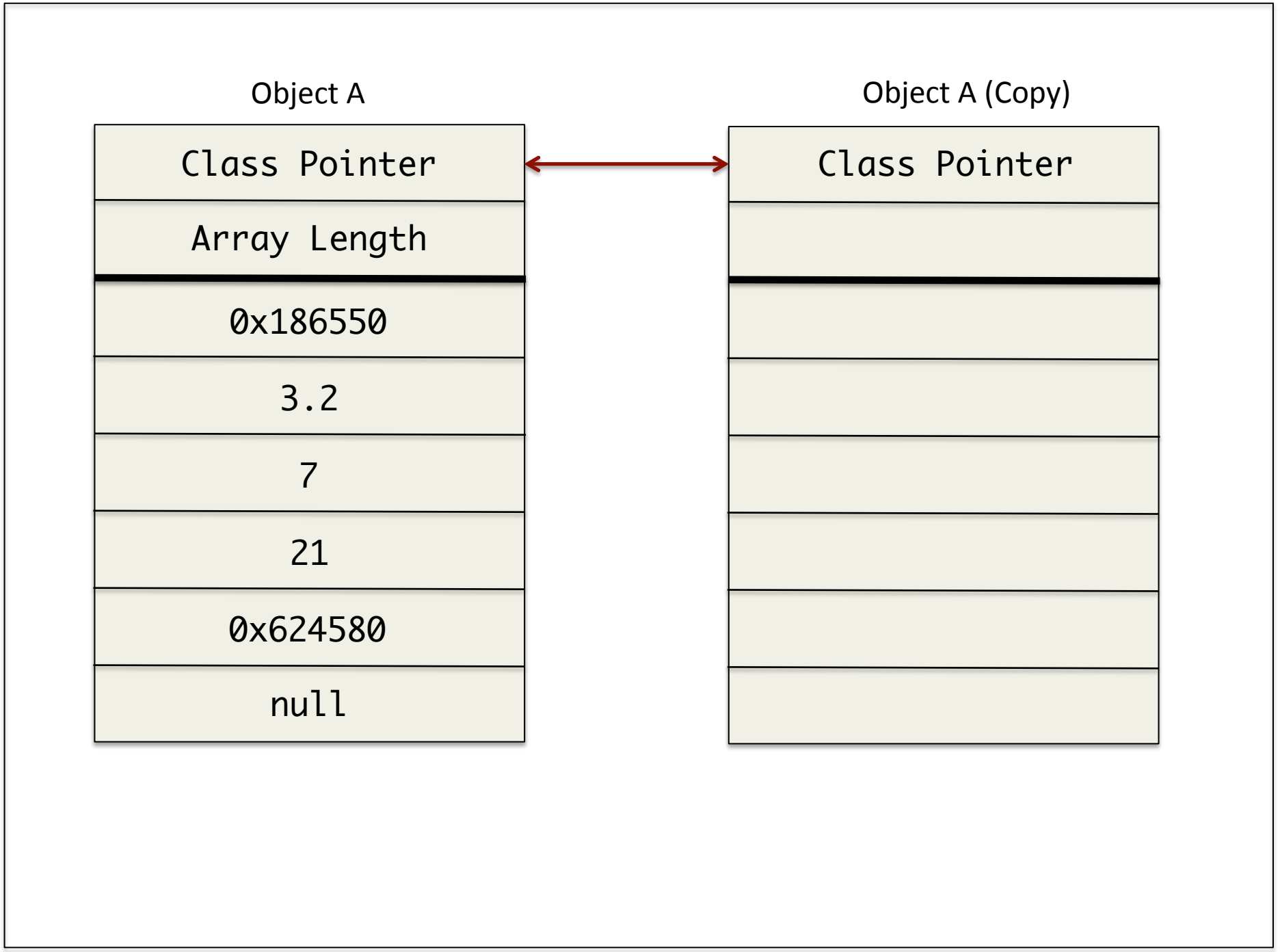
- Previous iteration of Hotspot's GC
  - Still in wide production use
- Composed from the primitives that we've now seen
  - Generational algorithm with Mark/Sweep mature space
  - Incremental update mark phase
  - Parallel sweep phase
- Sweeping can be completely parallel
  - By the time an object is swept, there are no references
  - No way for the mutator to be affected

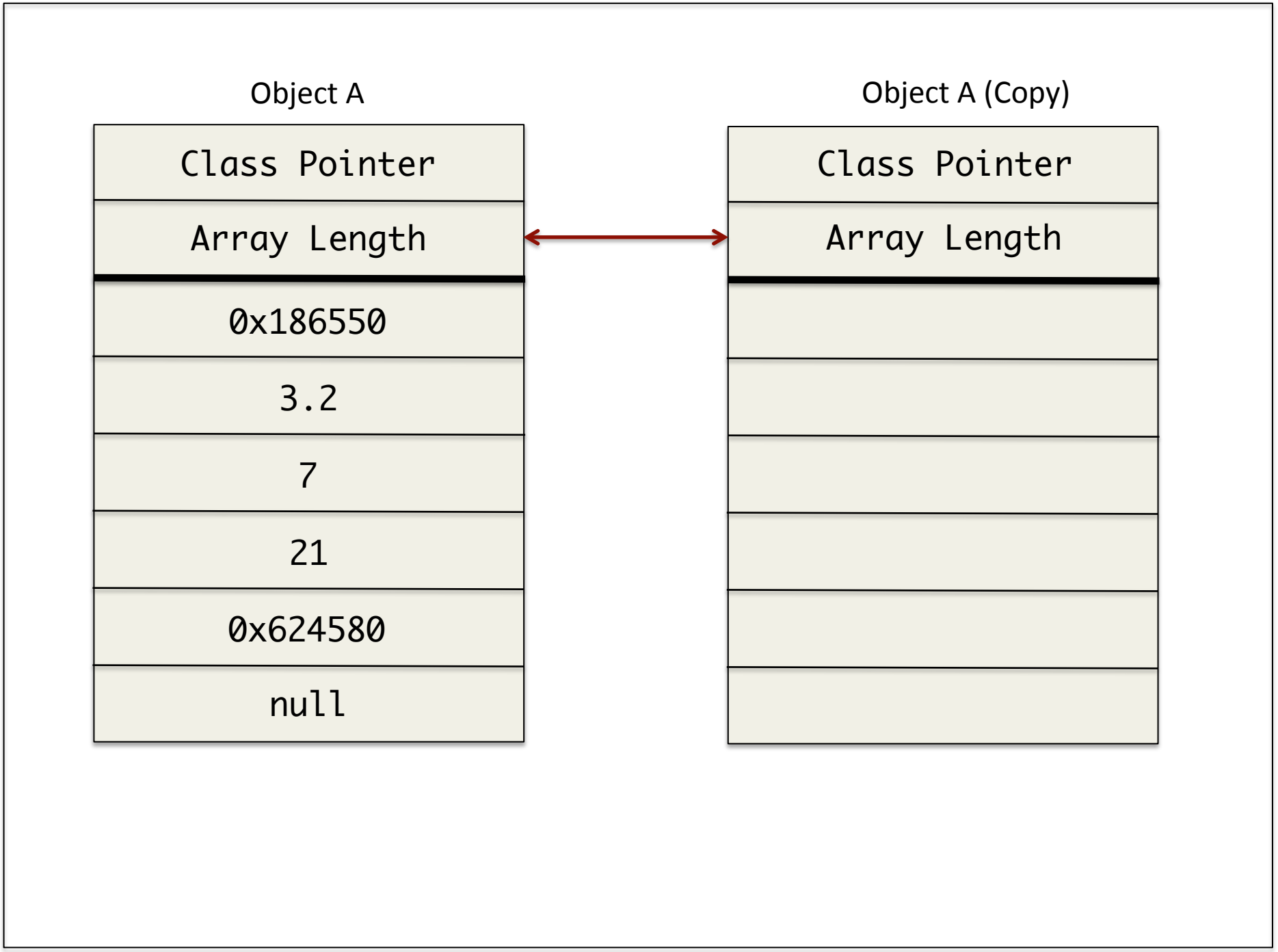
Concurrent Copying

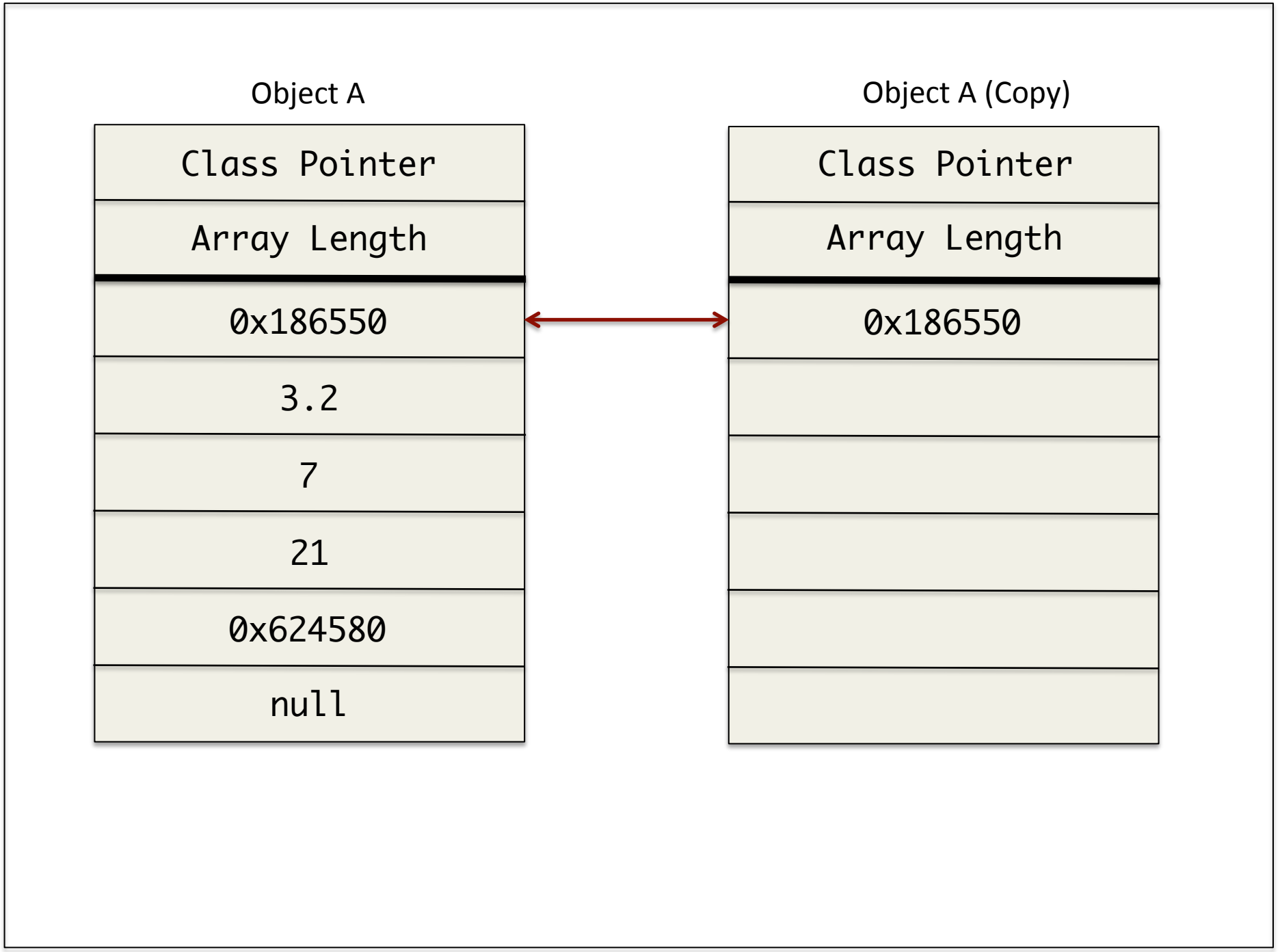
- So far all copying has happened during GC
  - All mutator threads are stopped
  - We know that the heap is in a bad state
  - We don't really care
- Copy phase can be slow
  - Minimize by copying smaller areas
  - Leads to more pauses, albeit shorter ones
- Ideally we could copy objects concurrently

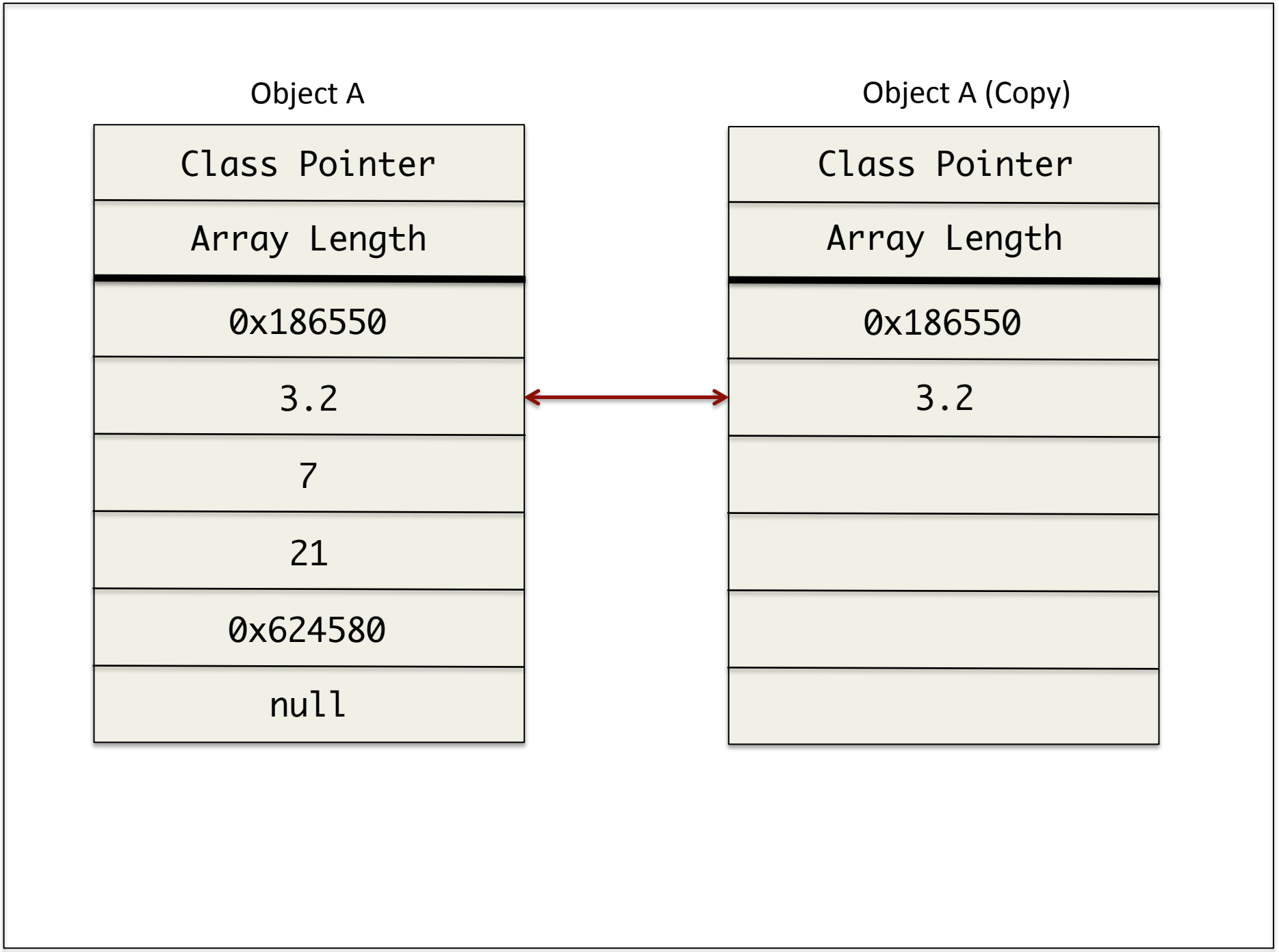

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null

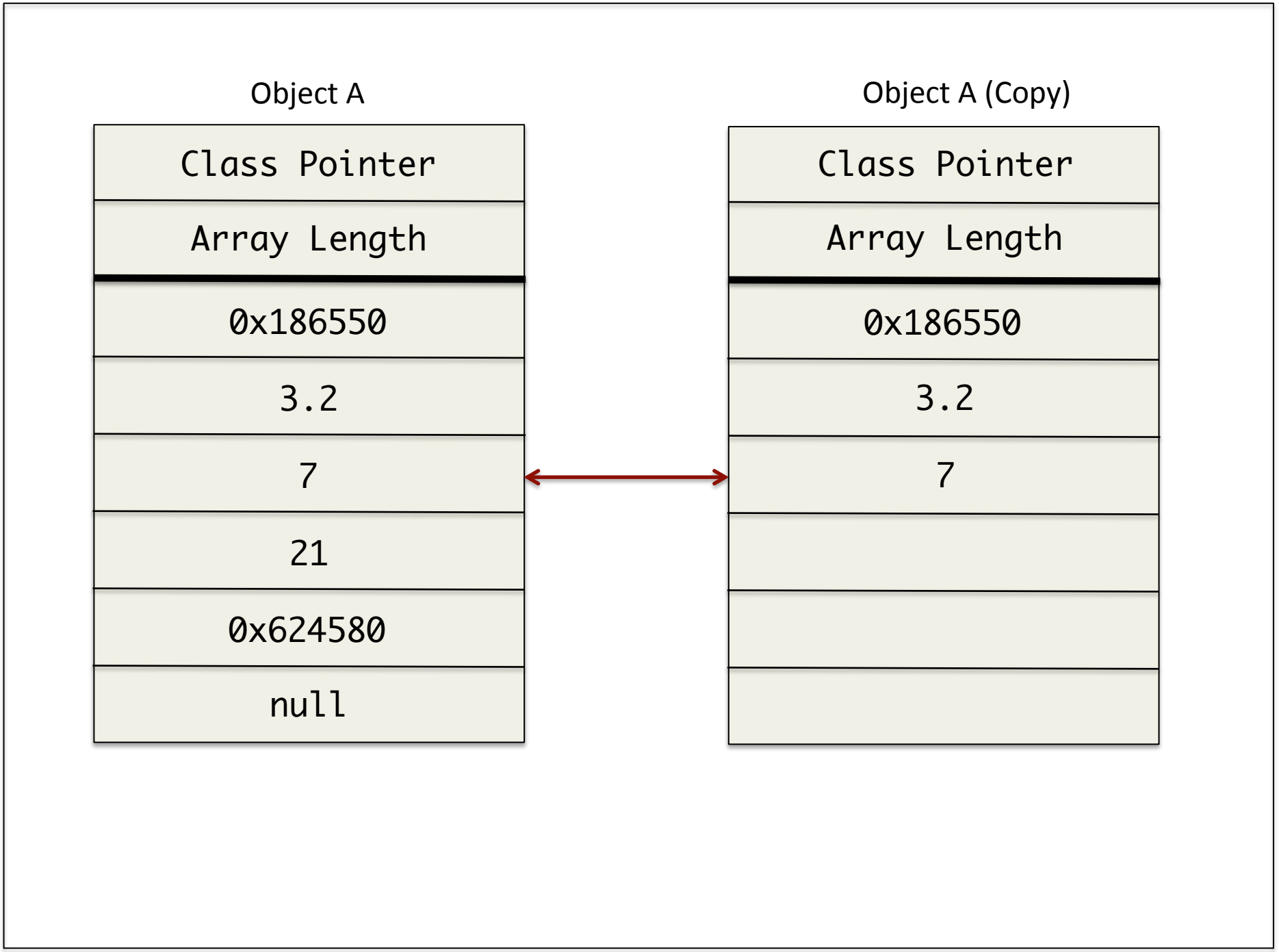

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)

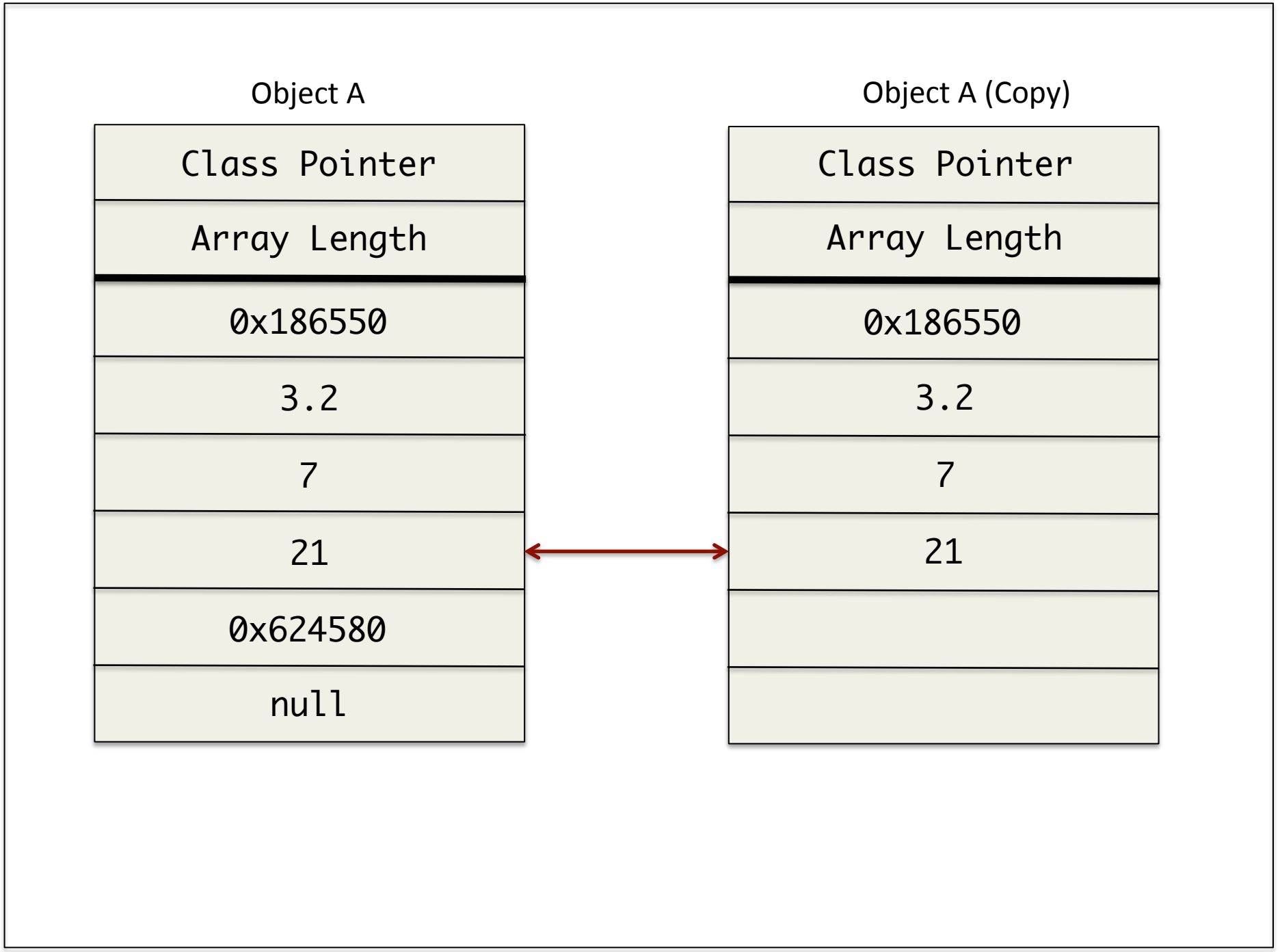

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer

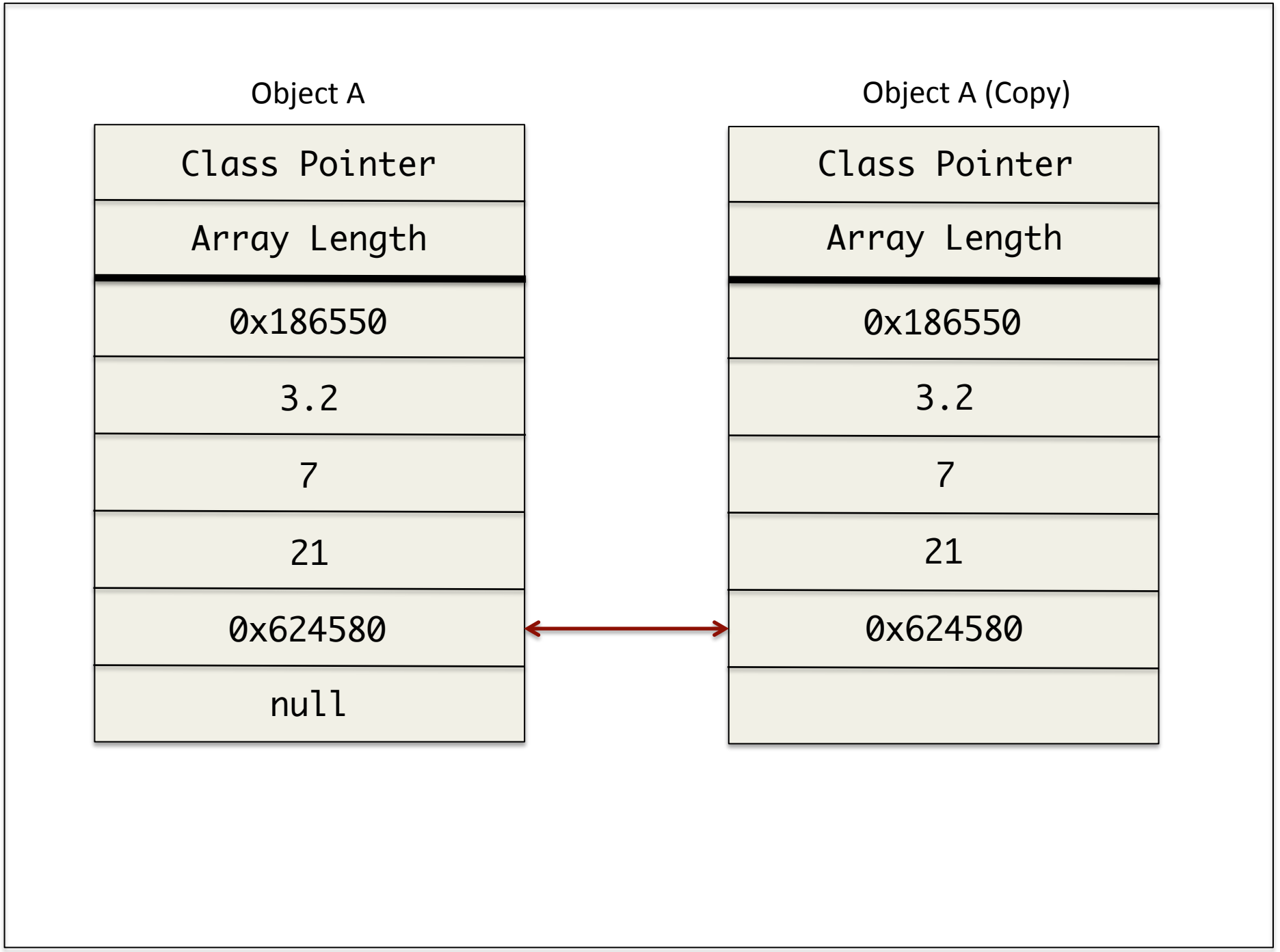

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length

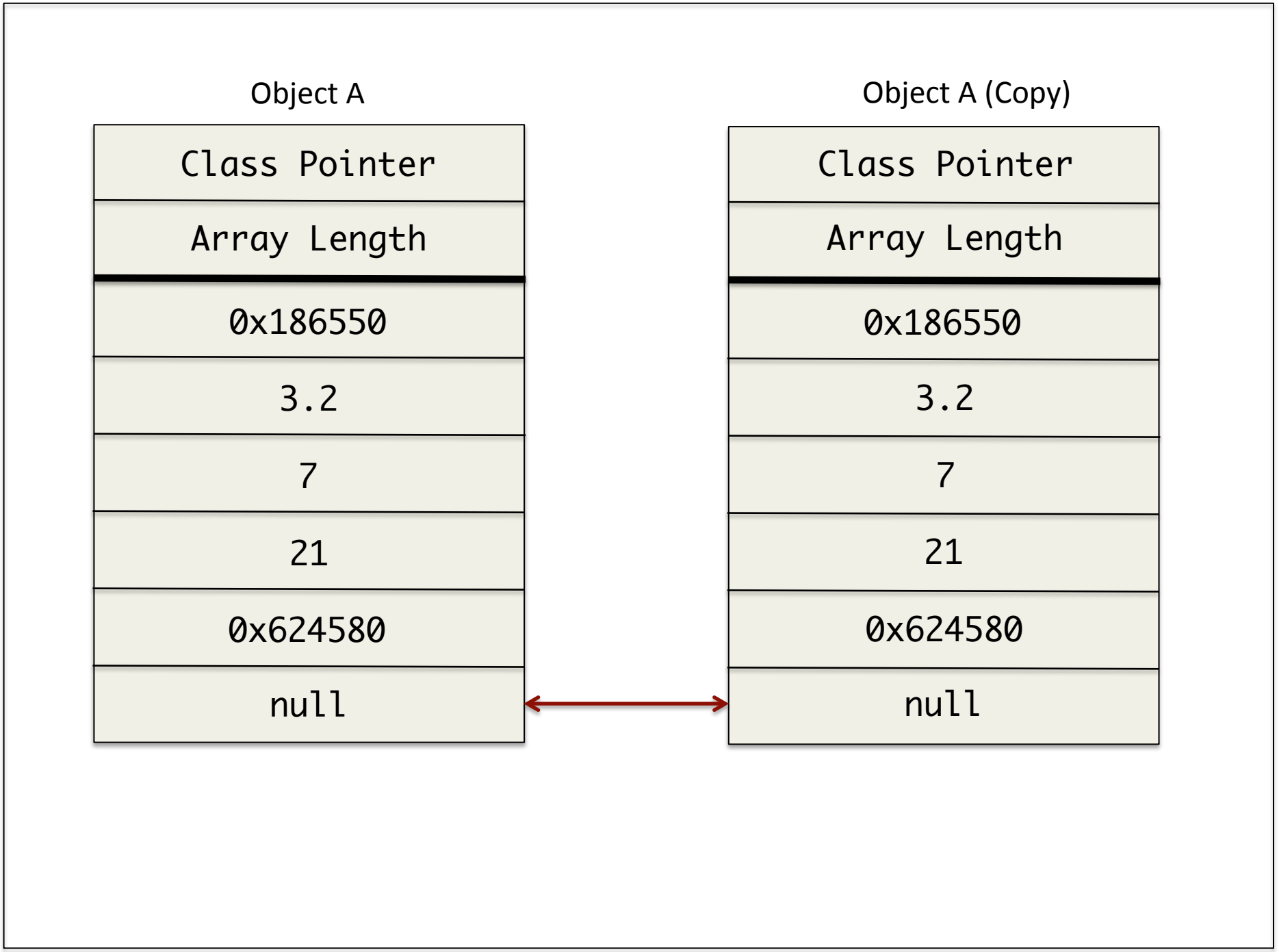

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550

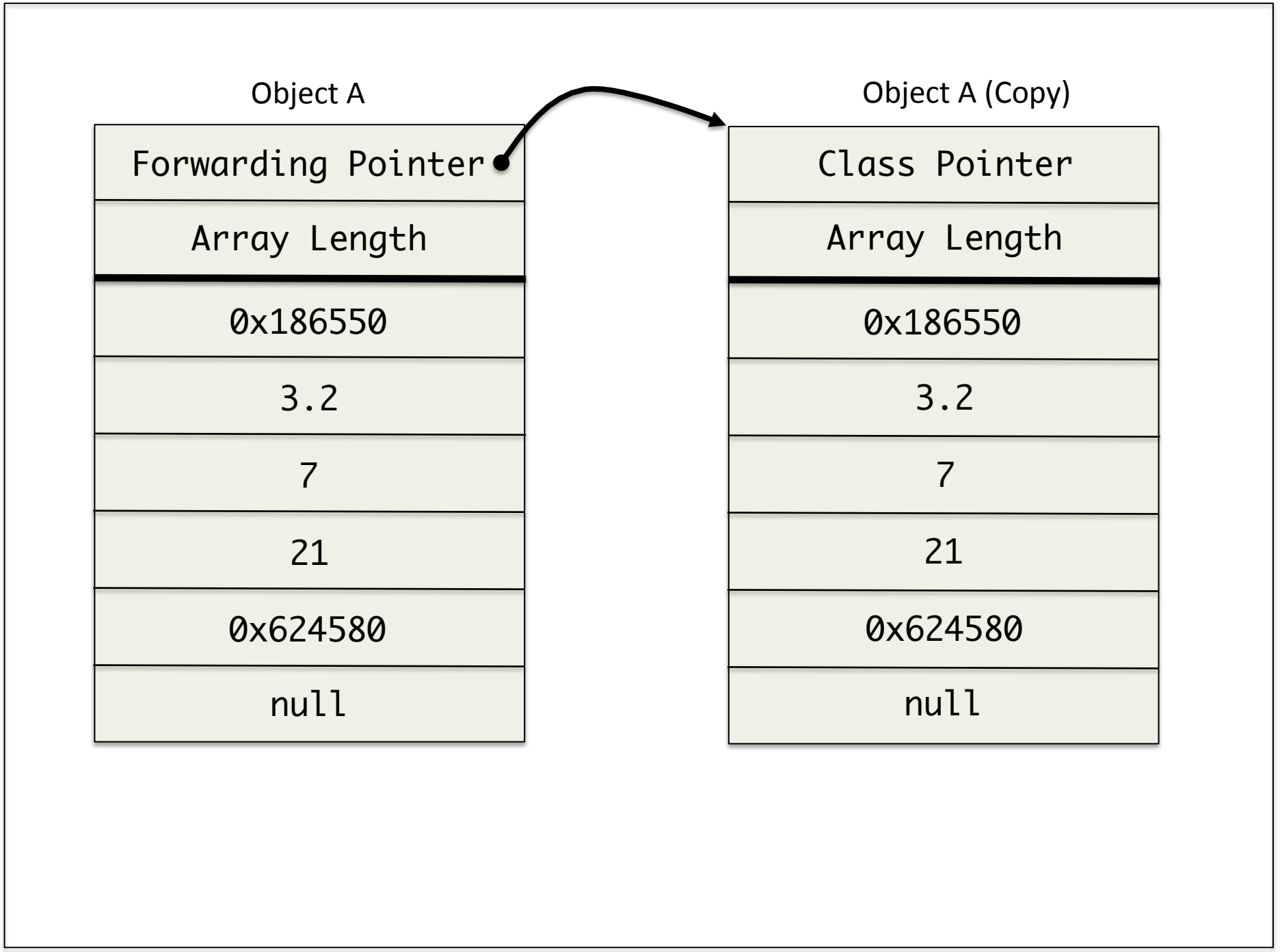

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2

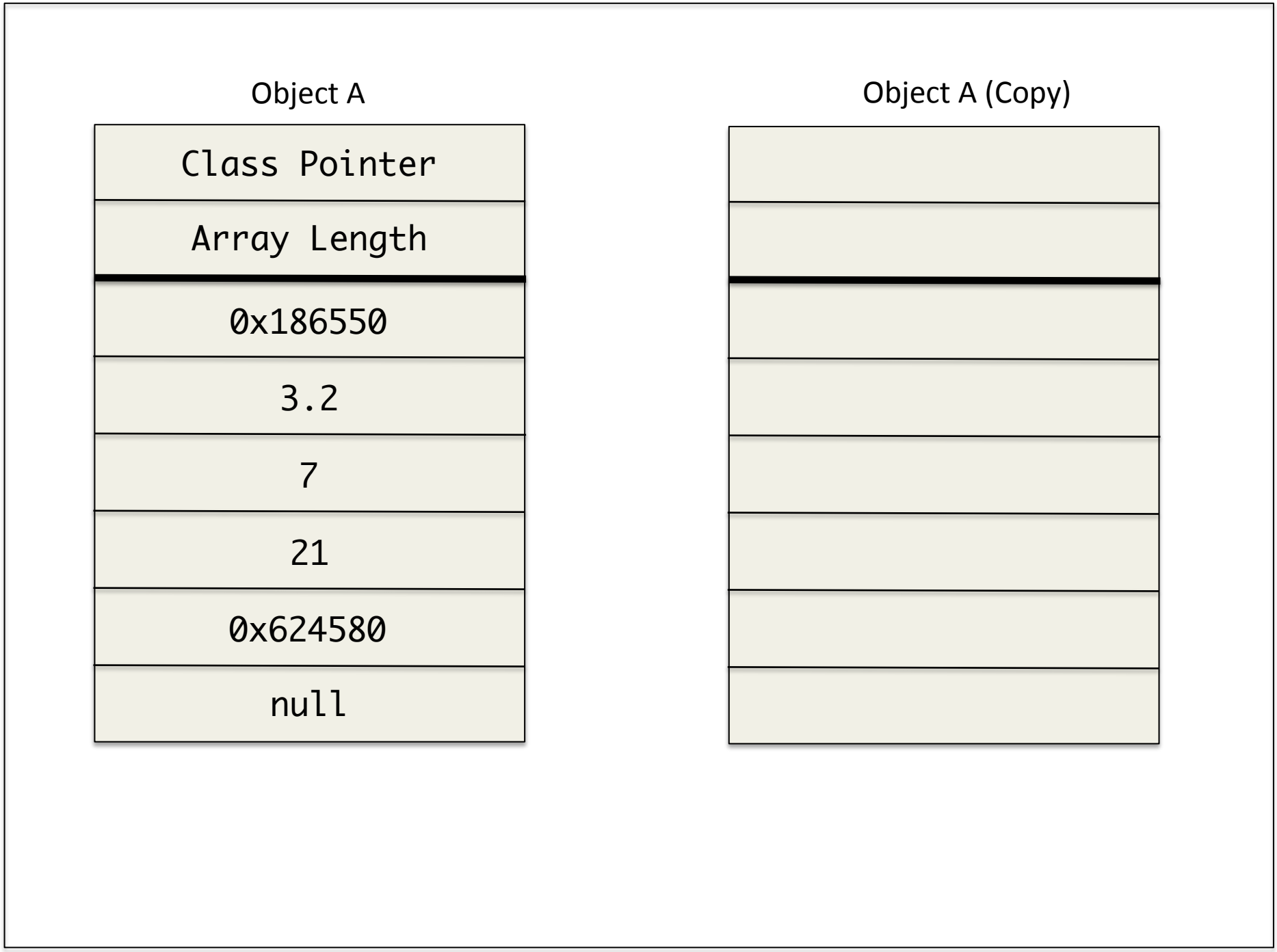

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7

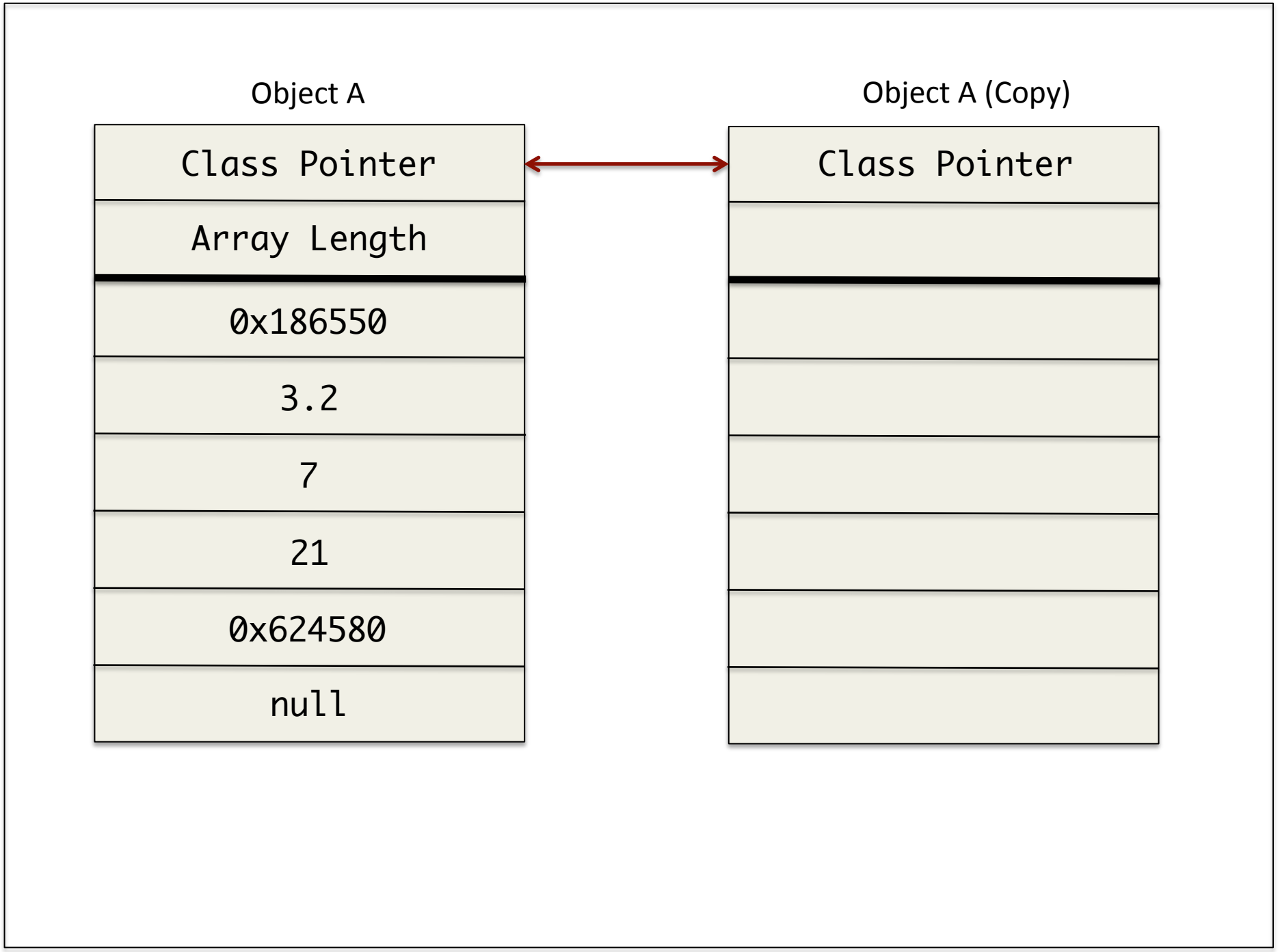

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21

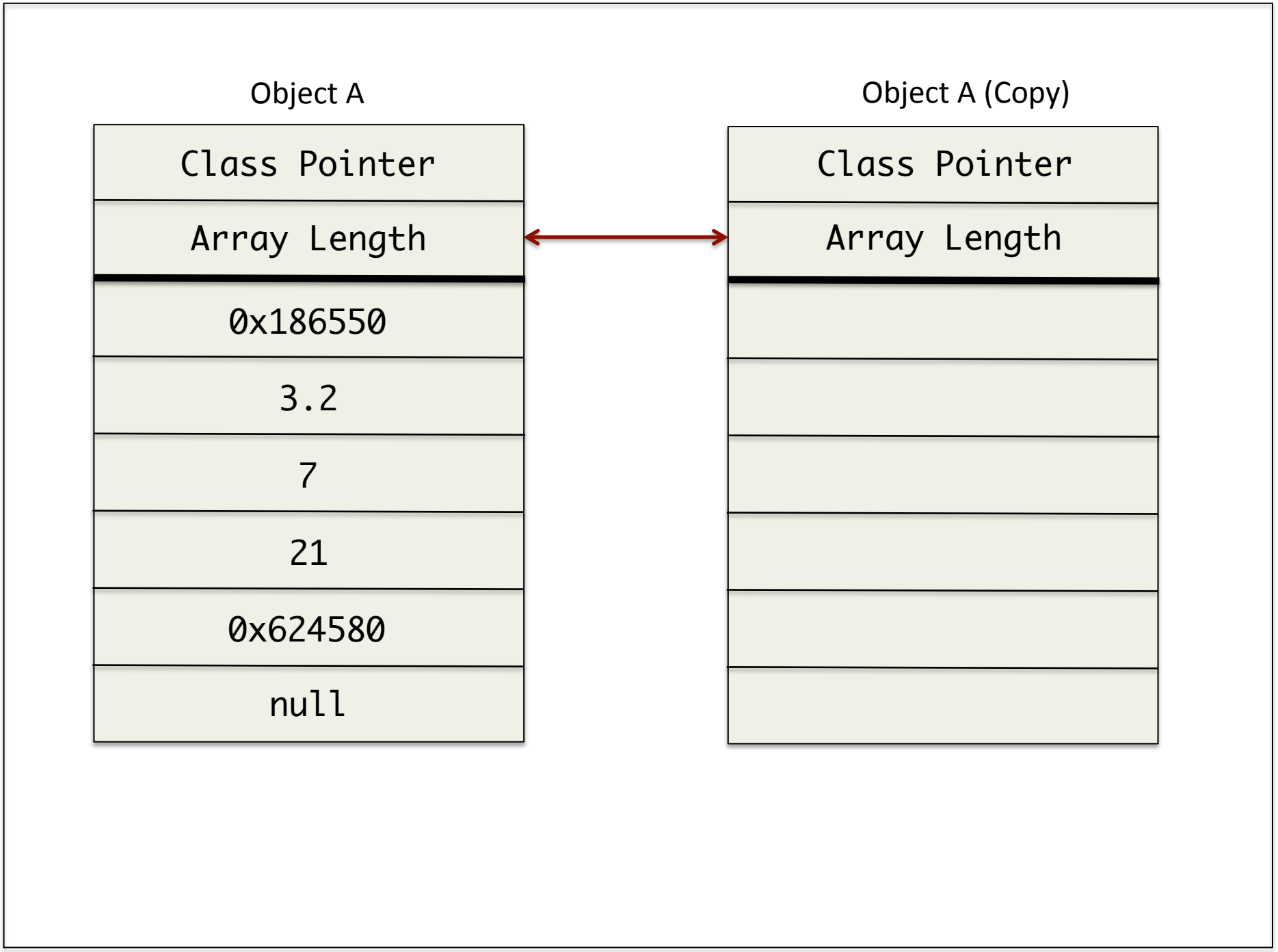

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580

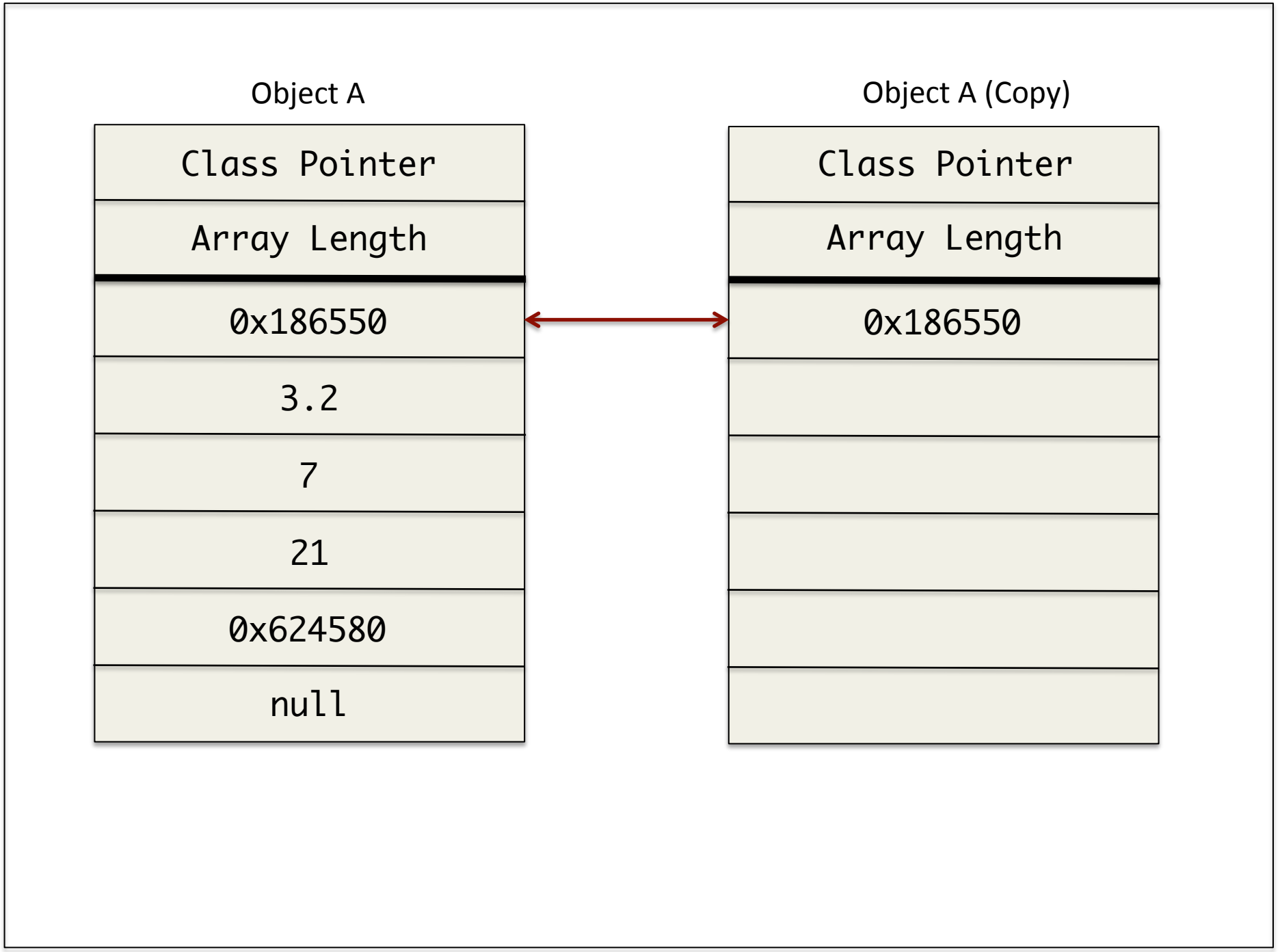

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null

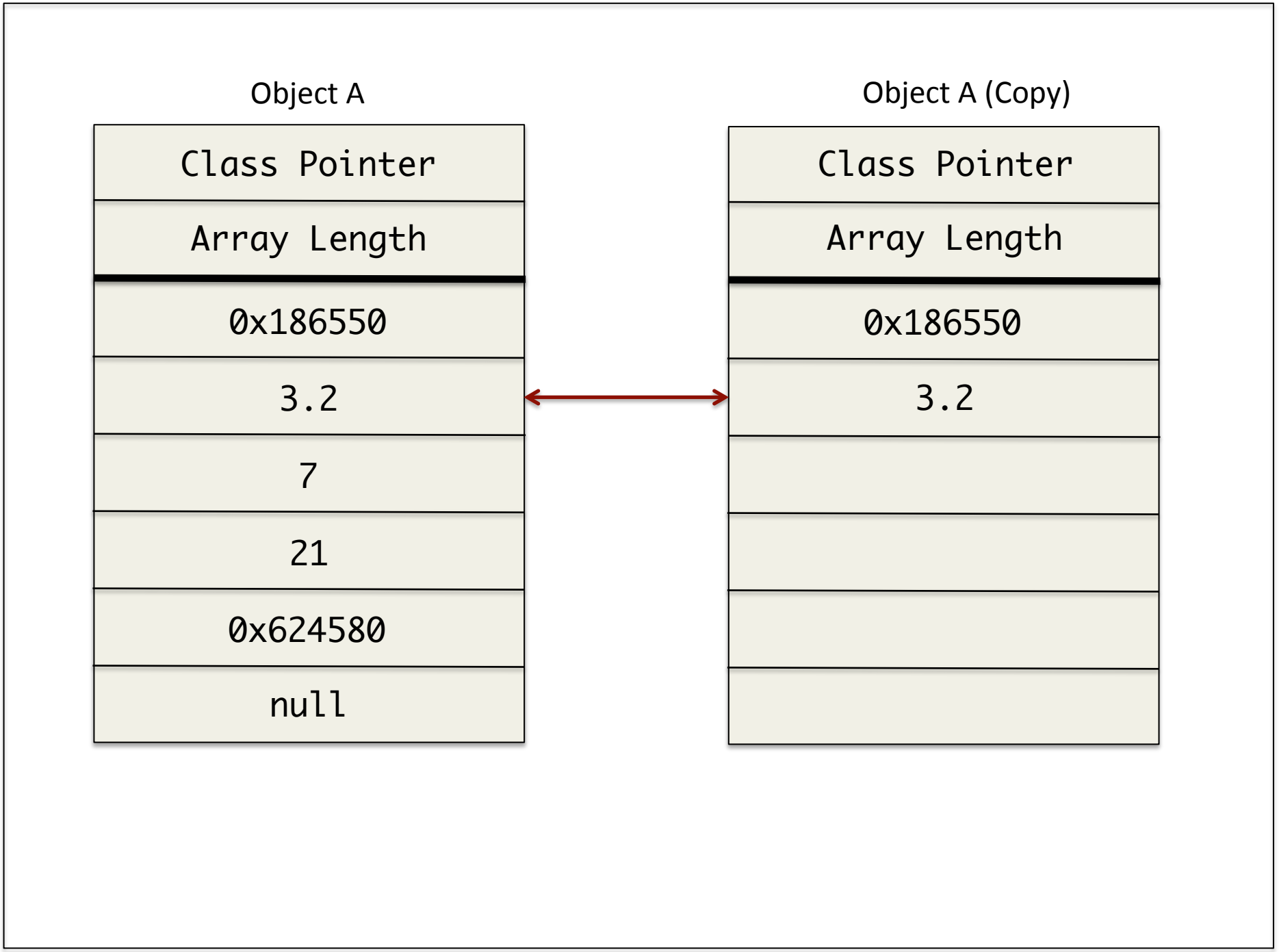

Object A
Forwarding Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null

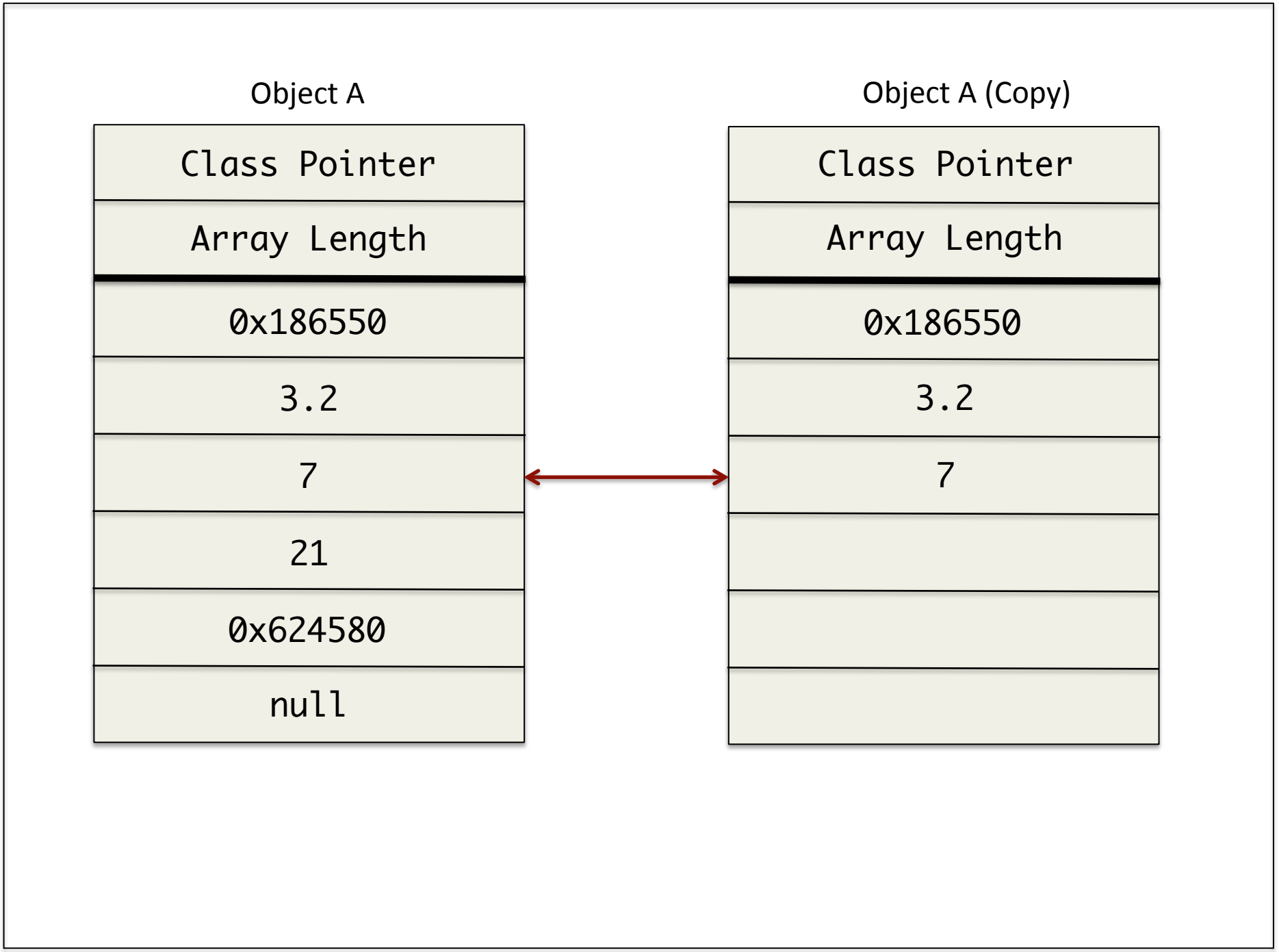

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)

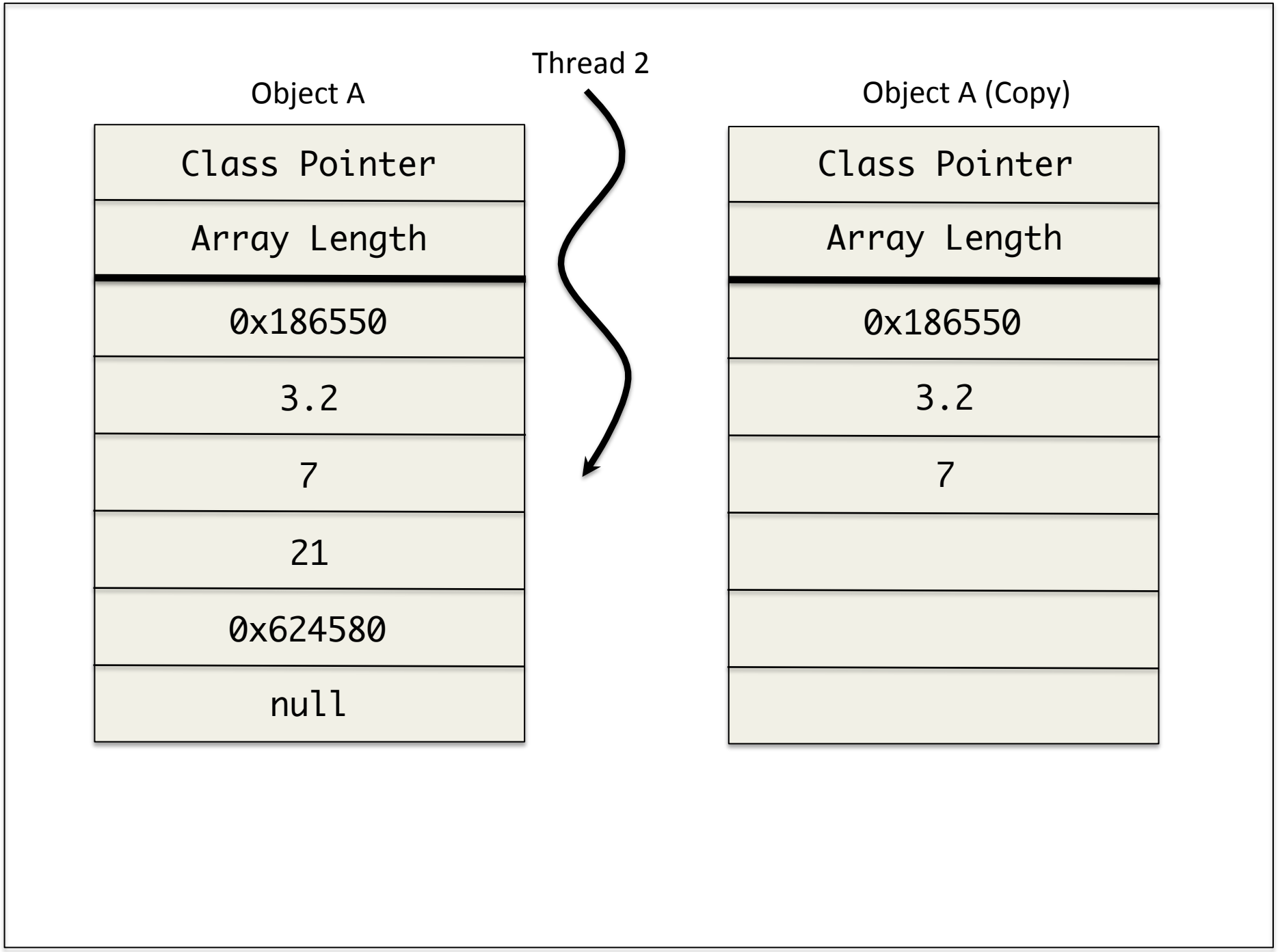

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer

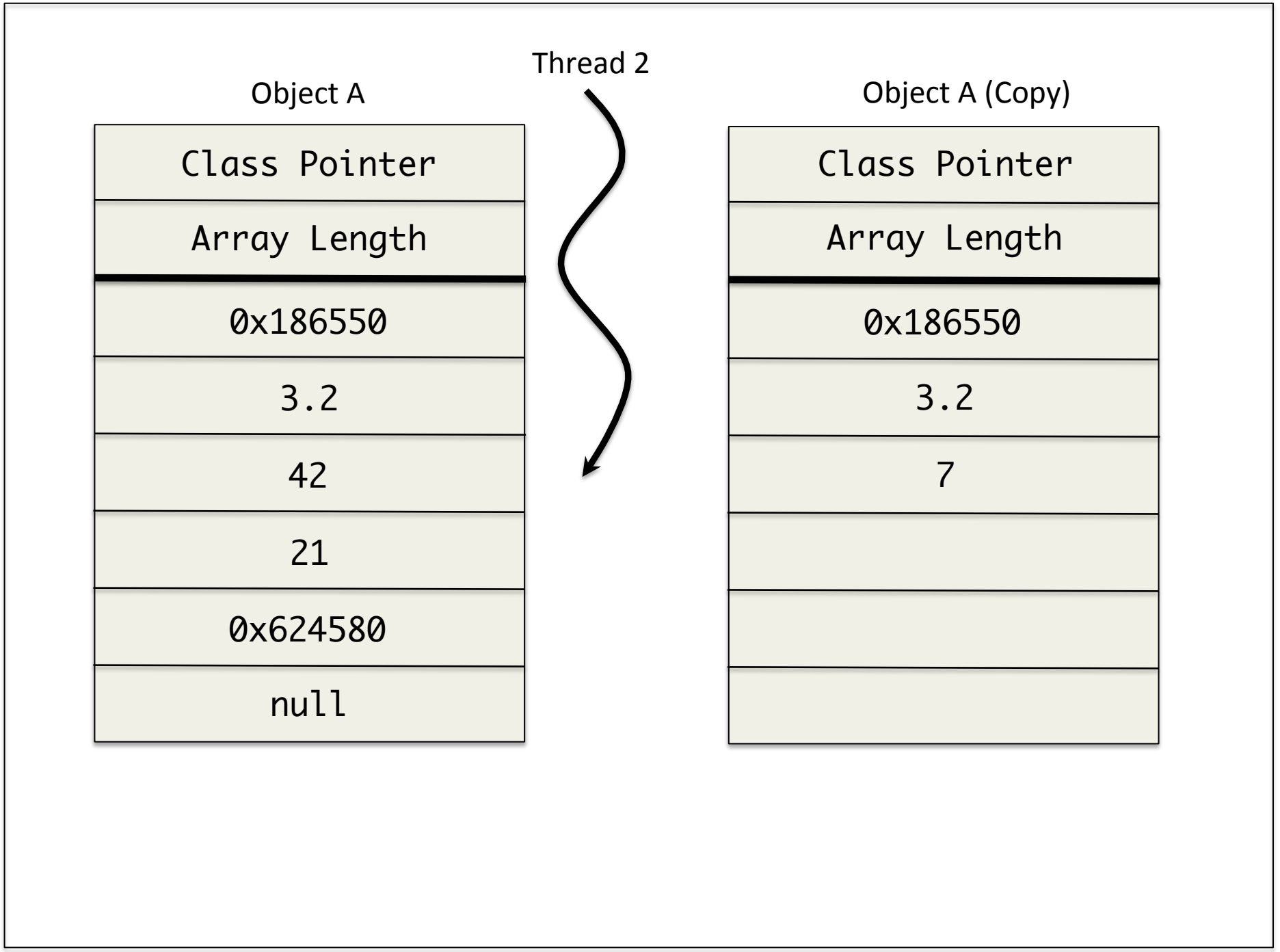

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length

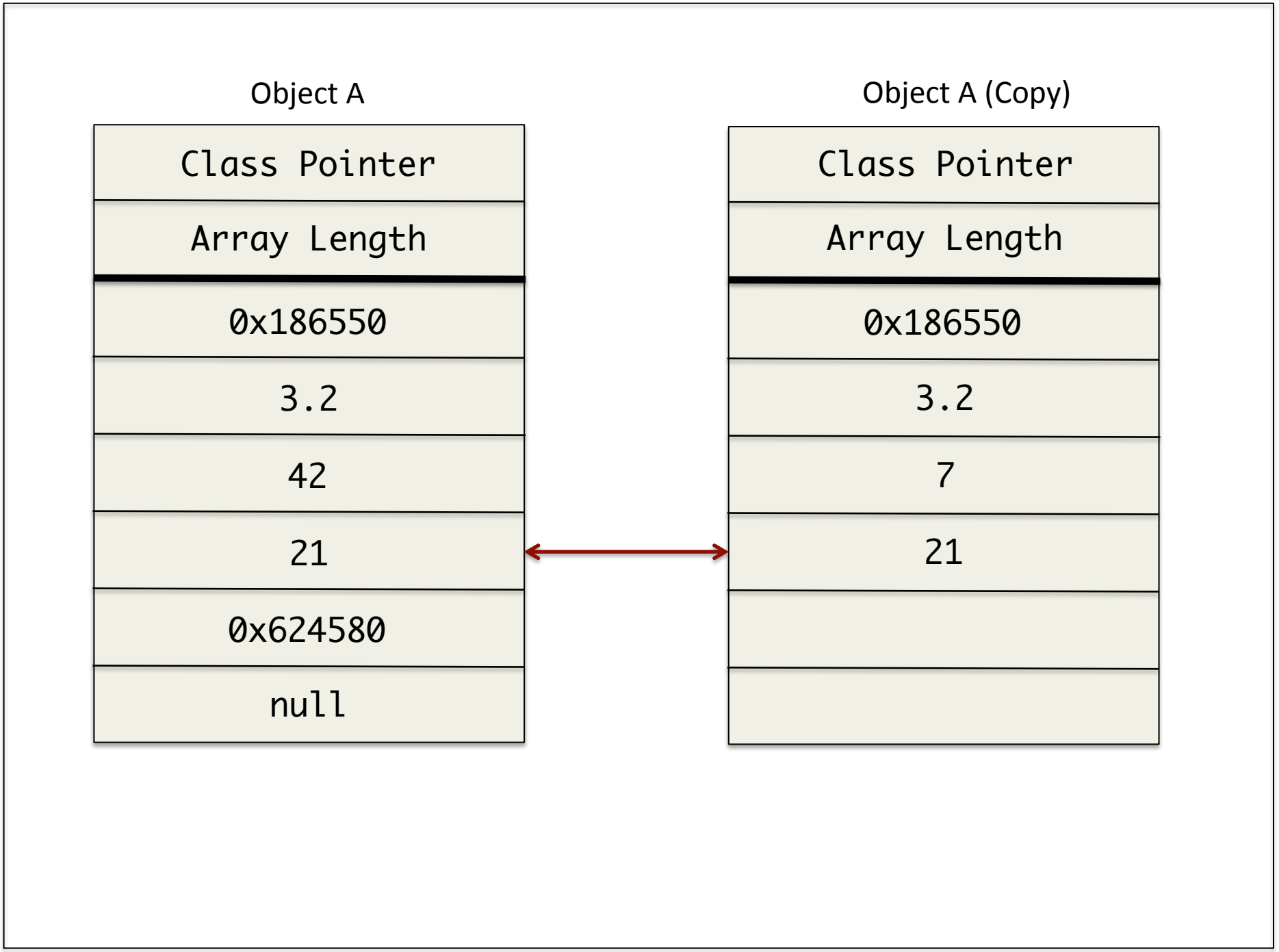

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550

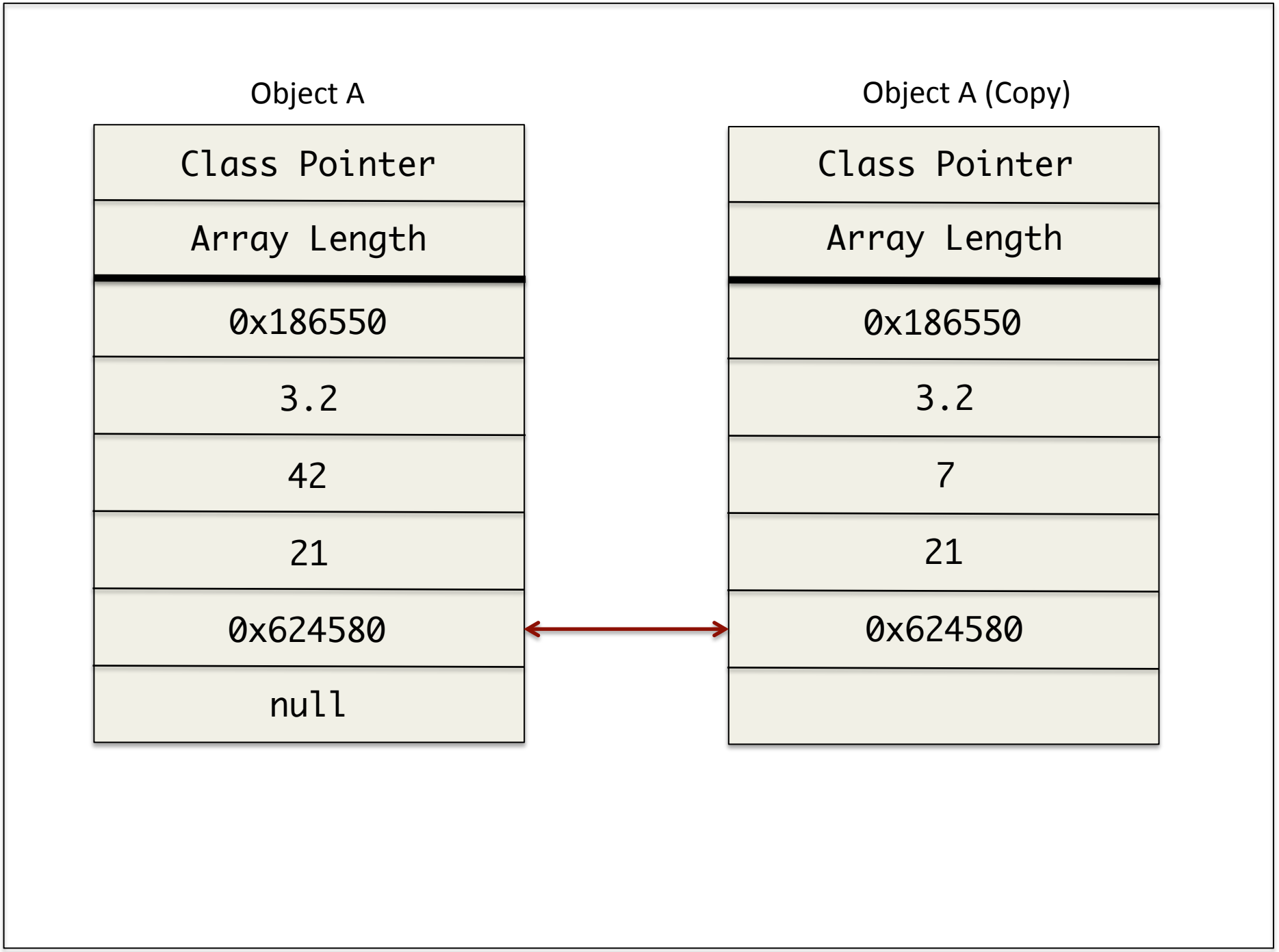

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2

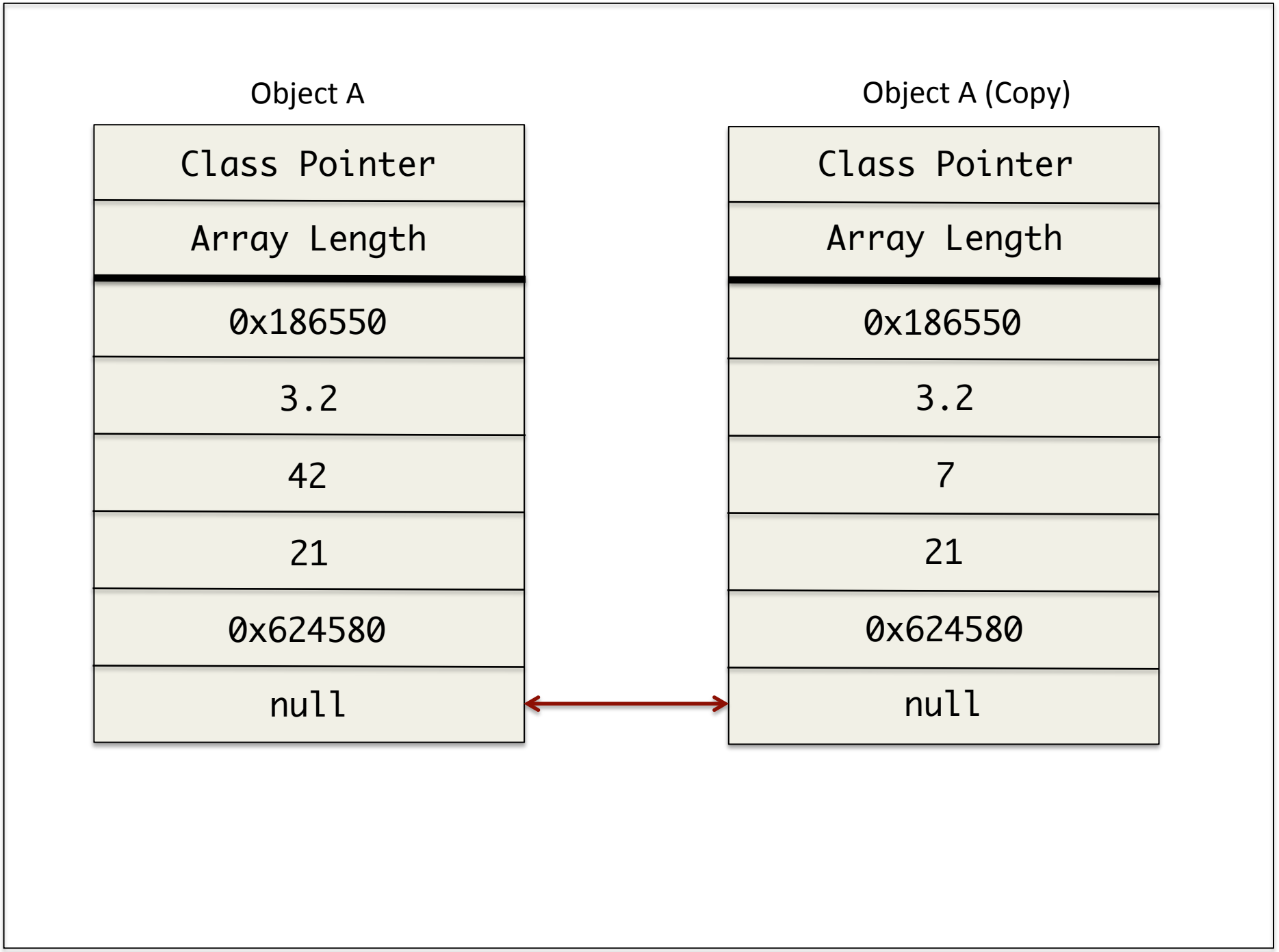

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7

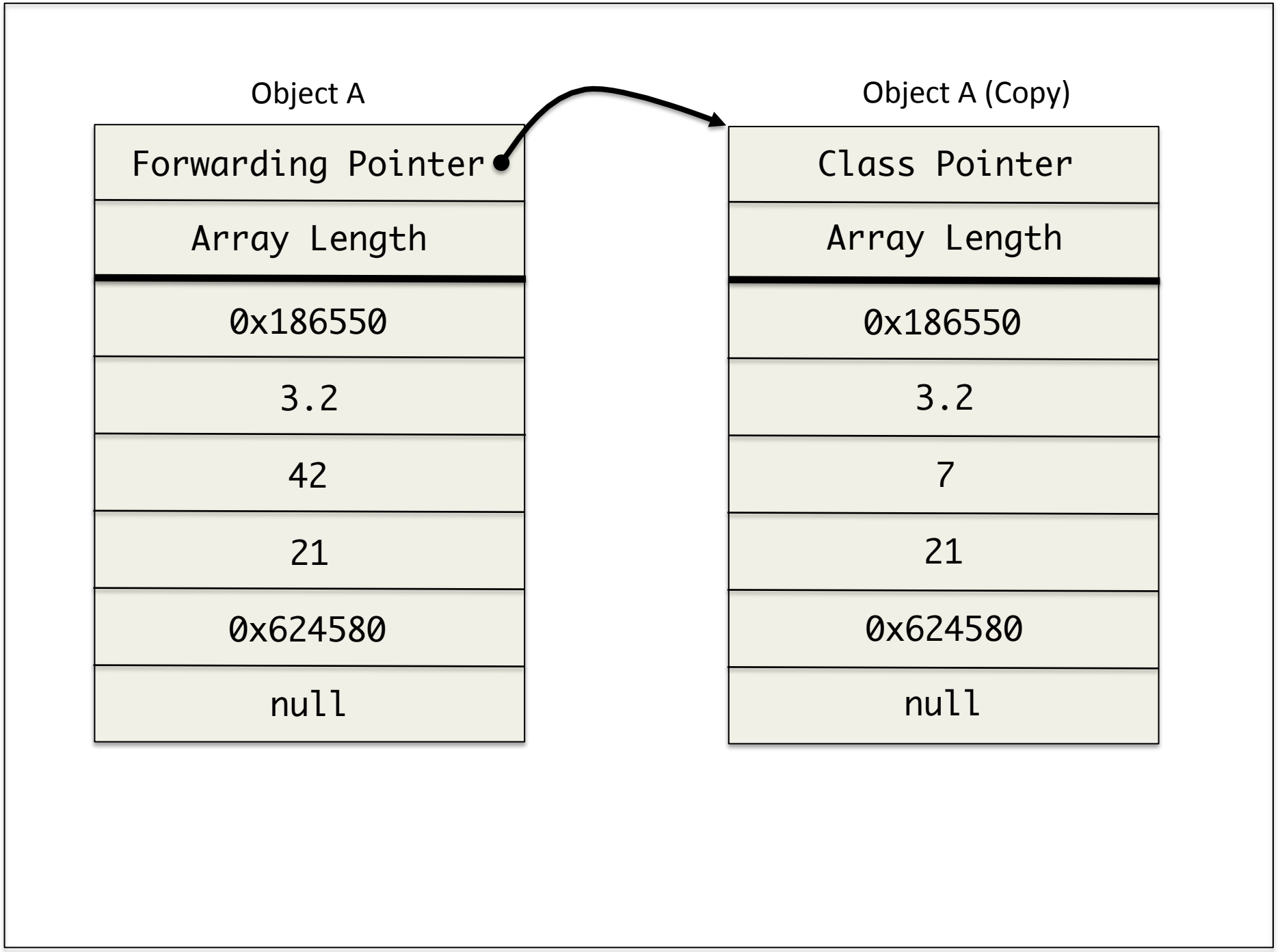

Thread 2
Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7


Thread 2
Object A
Class Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7


Object A
Class Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21


Object A
Class Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580


Object A
Class Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null

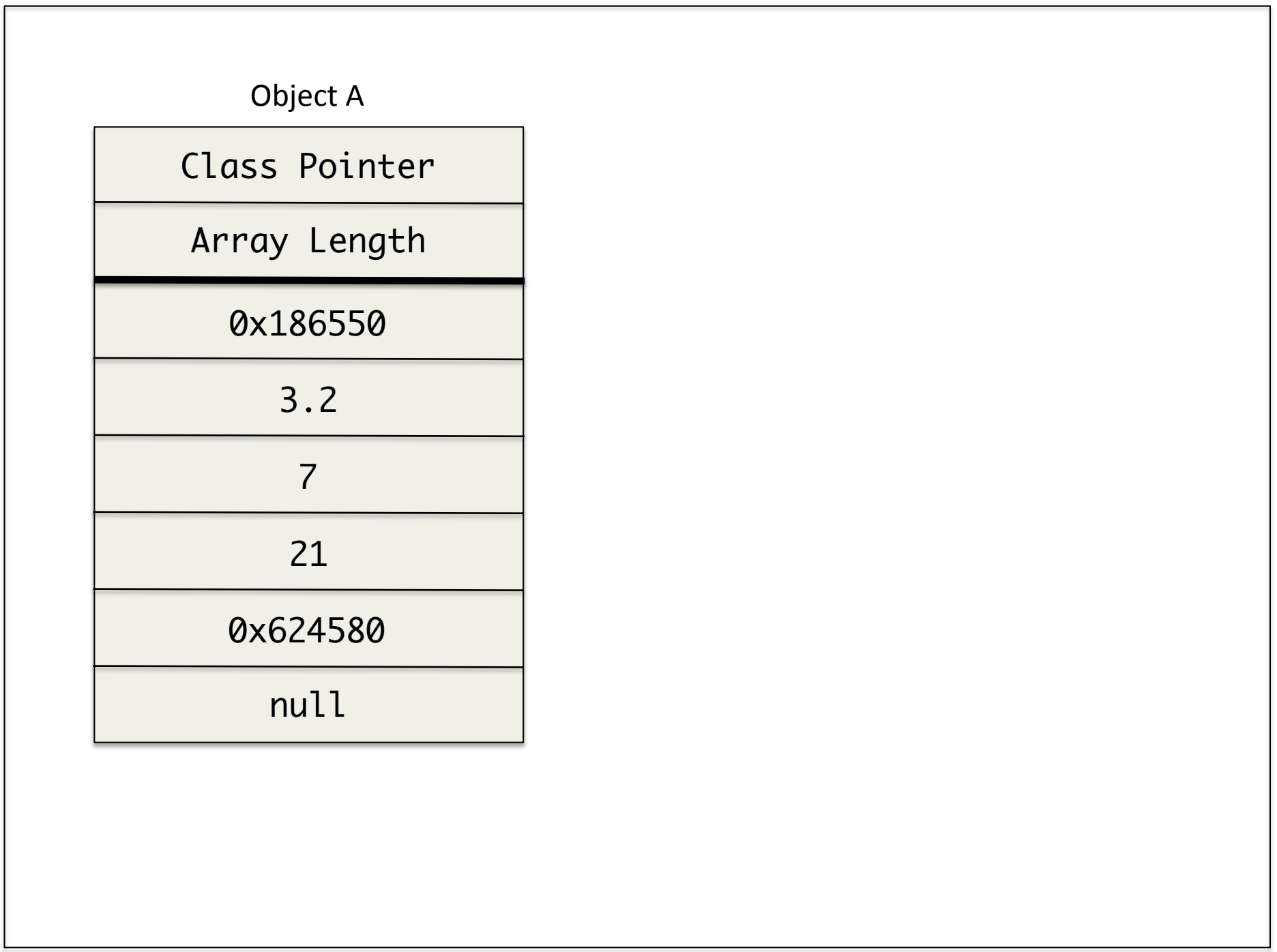

Object A
Forwarding Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null

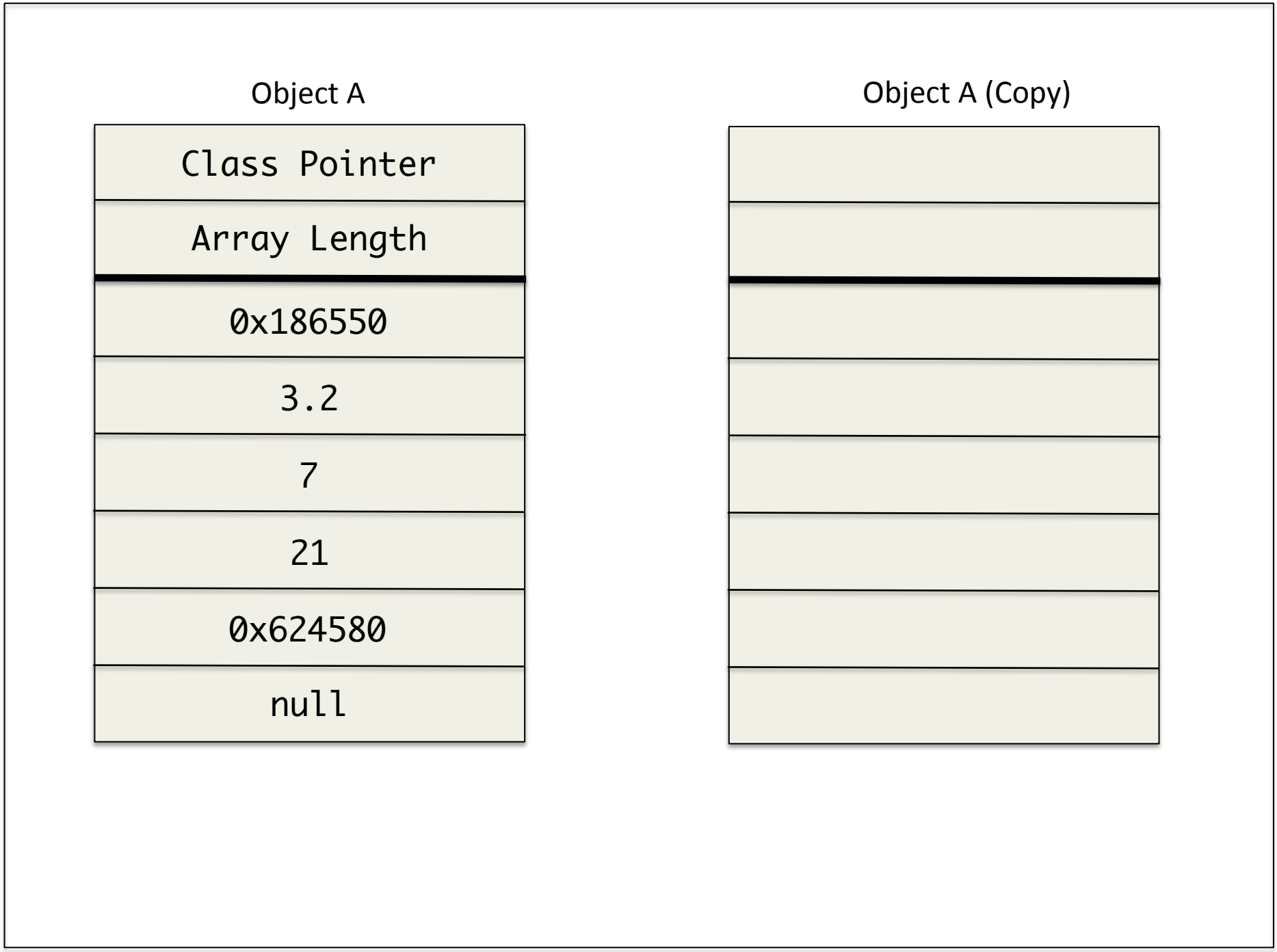
Concurrent Copying

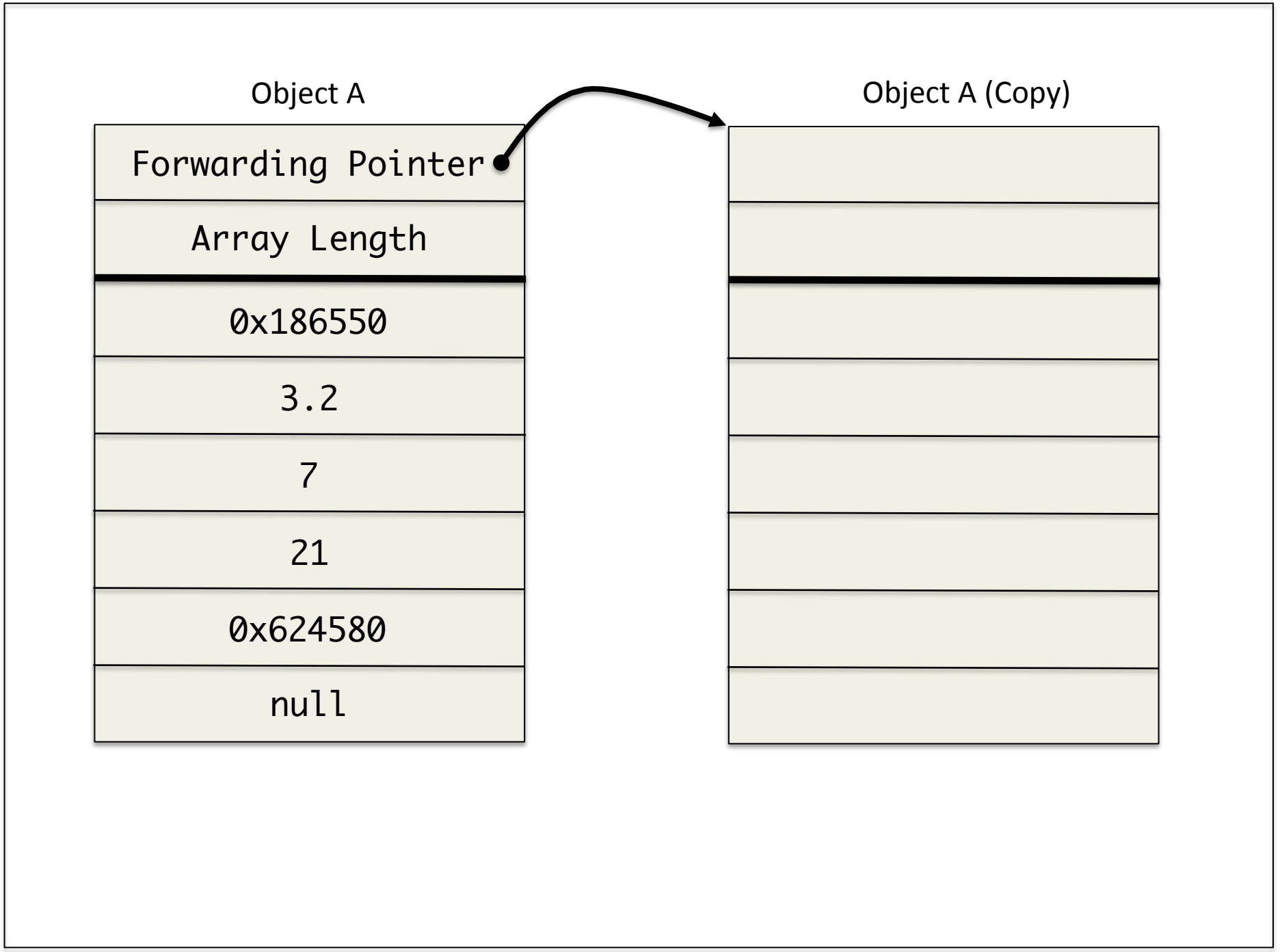
- Lost update problem makes concurrent copy hard
  - Particularly in light of the Java memory model
- One option would be to lock each object
  - GC and mutator threads contend for access
  - Set up contention manager to favor the mutator
- Some other options available

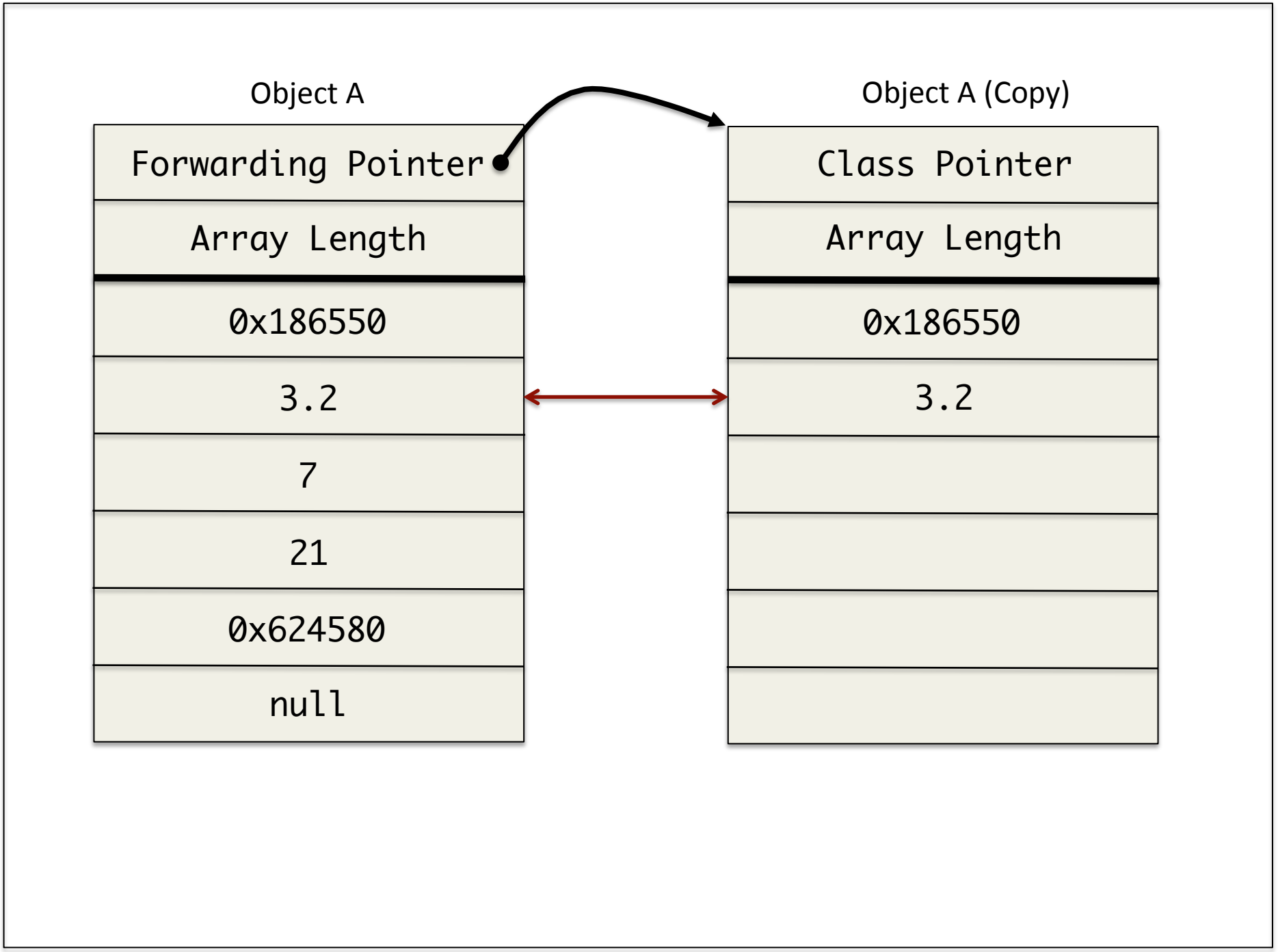
Sapphire

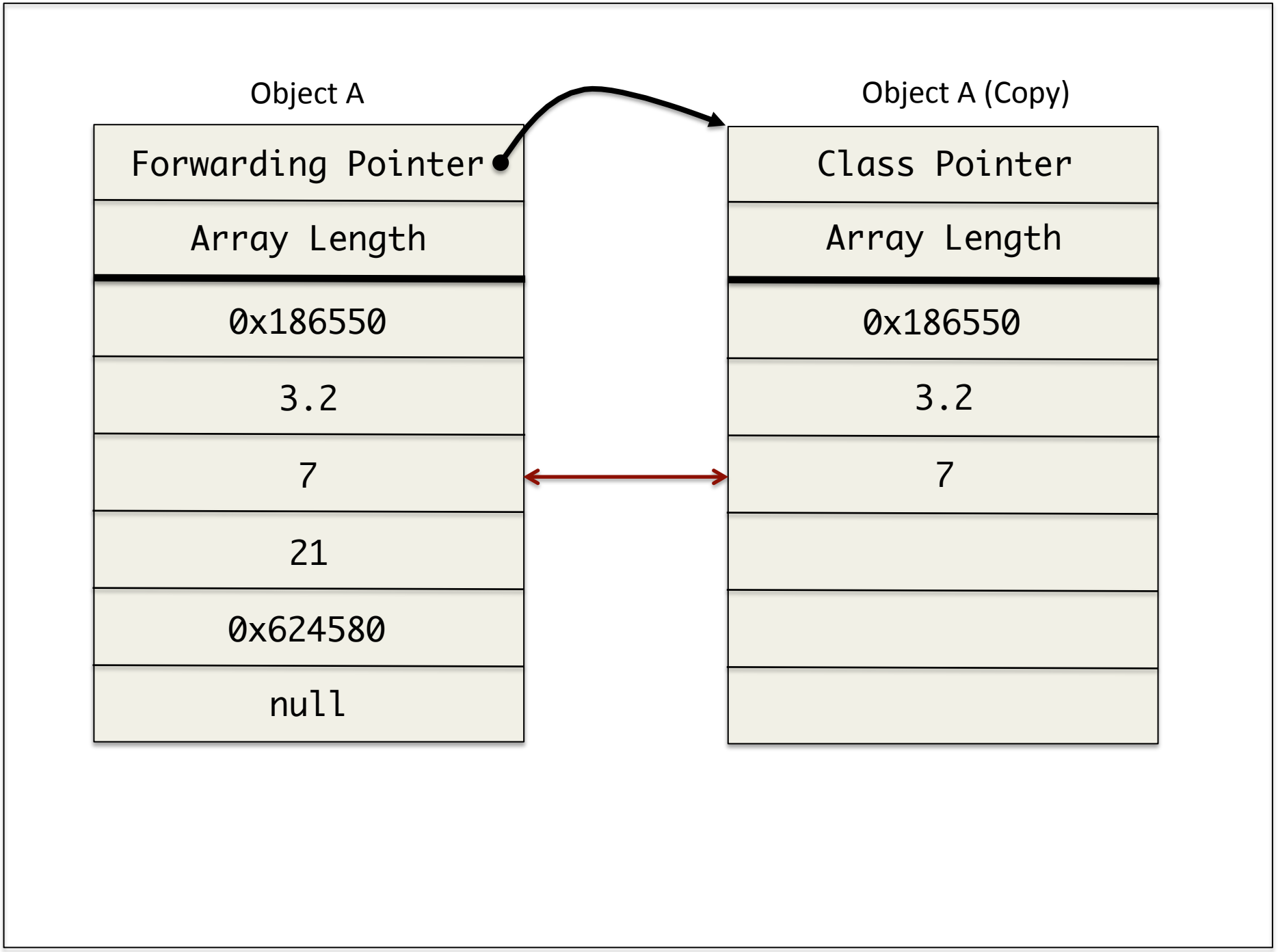
- The Sapphire collector is split into three phases
  - Mark, copy, flip
- Creates a shell for each live object prior to copy
  - Installs forwarding pointer to the shell
- Installs write barrier during copy phase
  - Barrier installed on all writes (not just pointers)
  - Mutator writes to both old and new versions

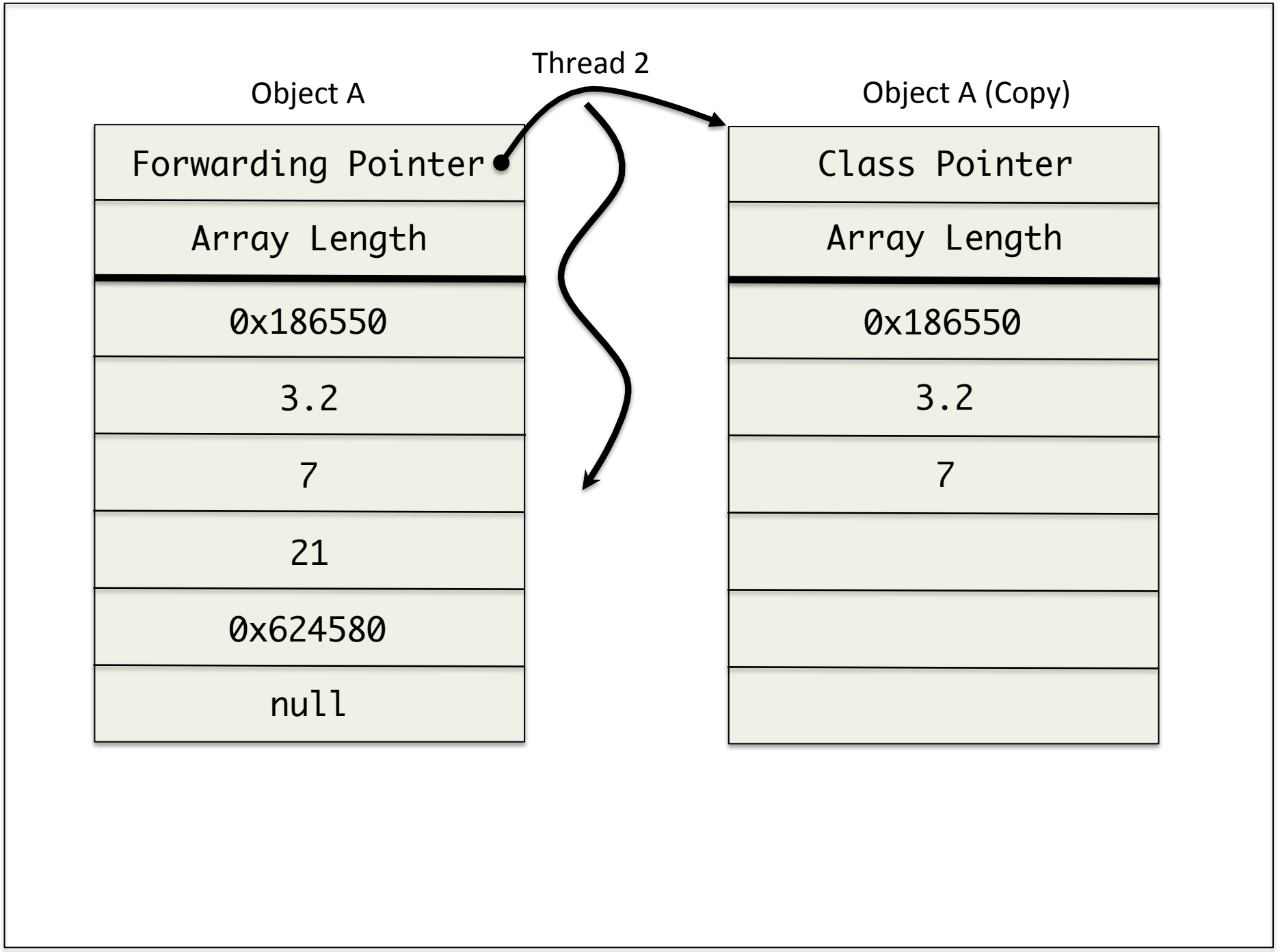

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null

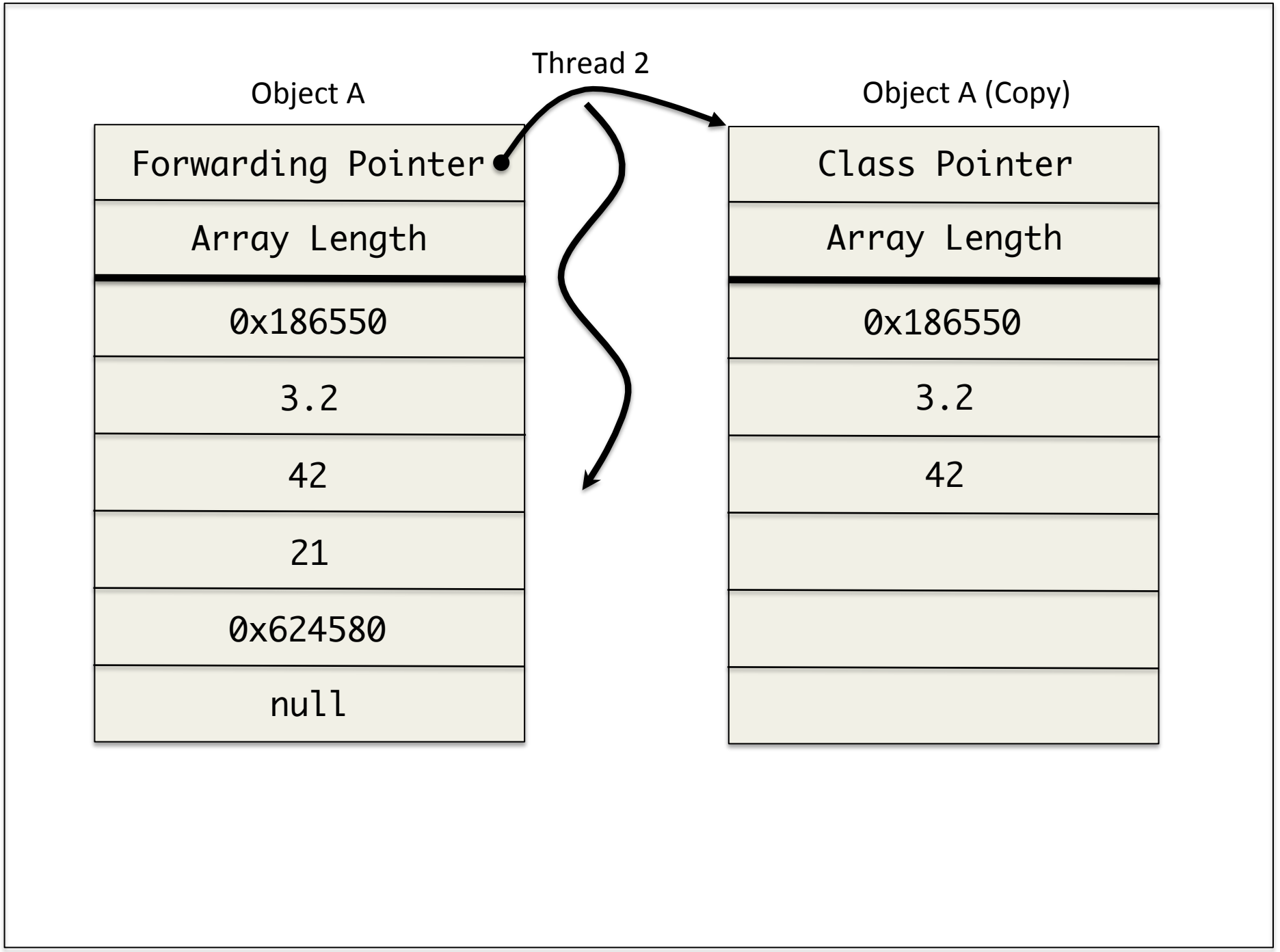

Object A
Class Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)

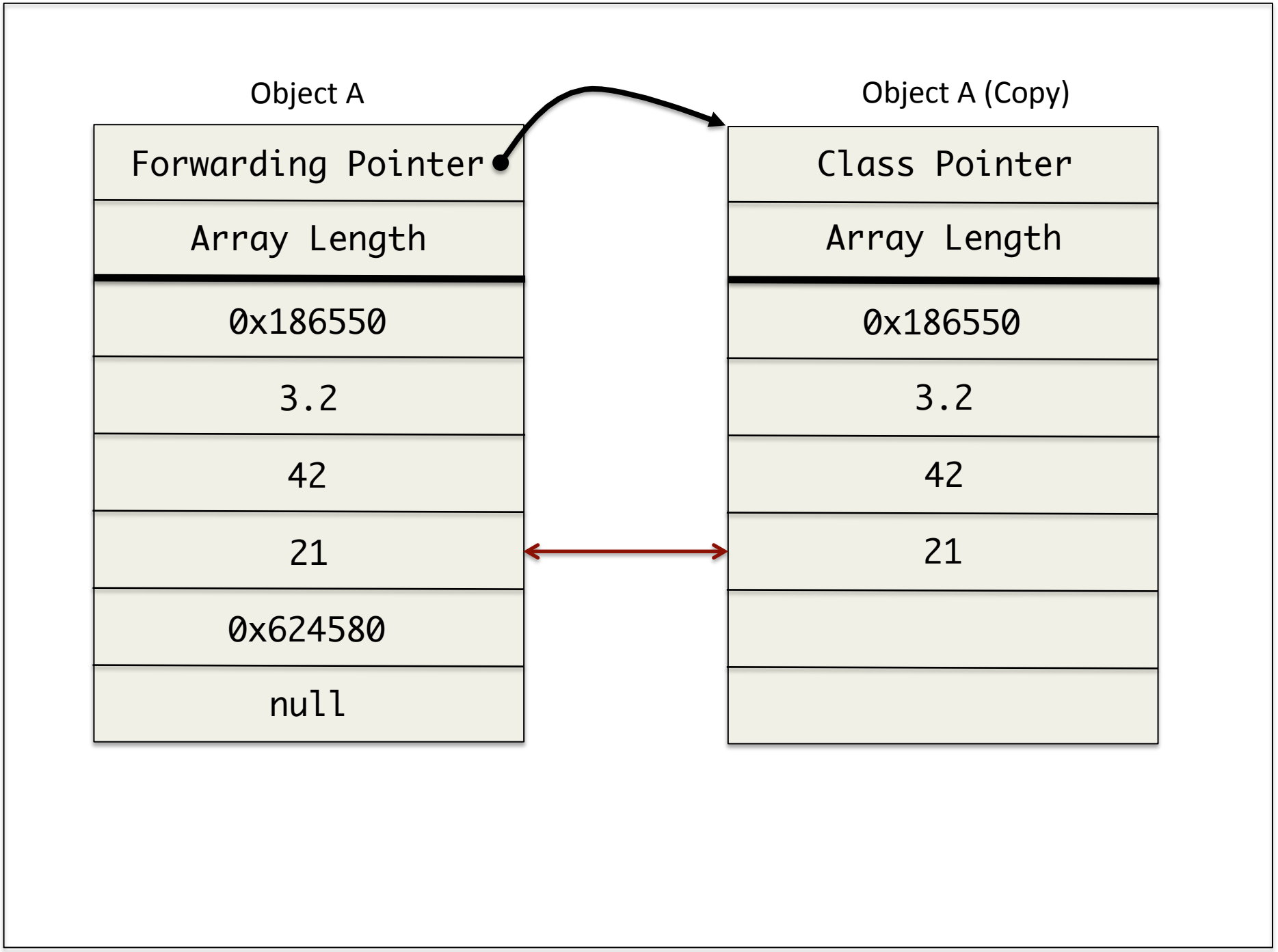

Object A
Forwarding Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)

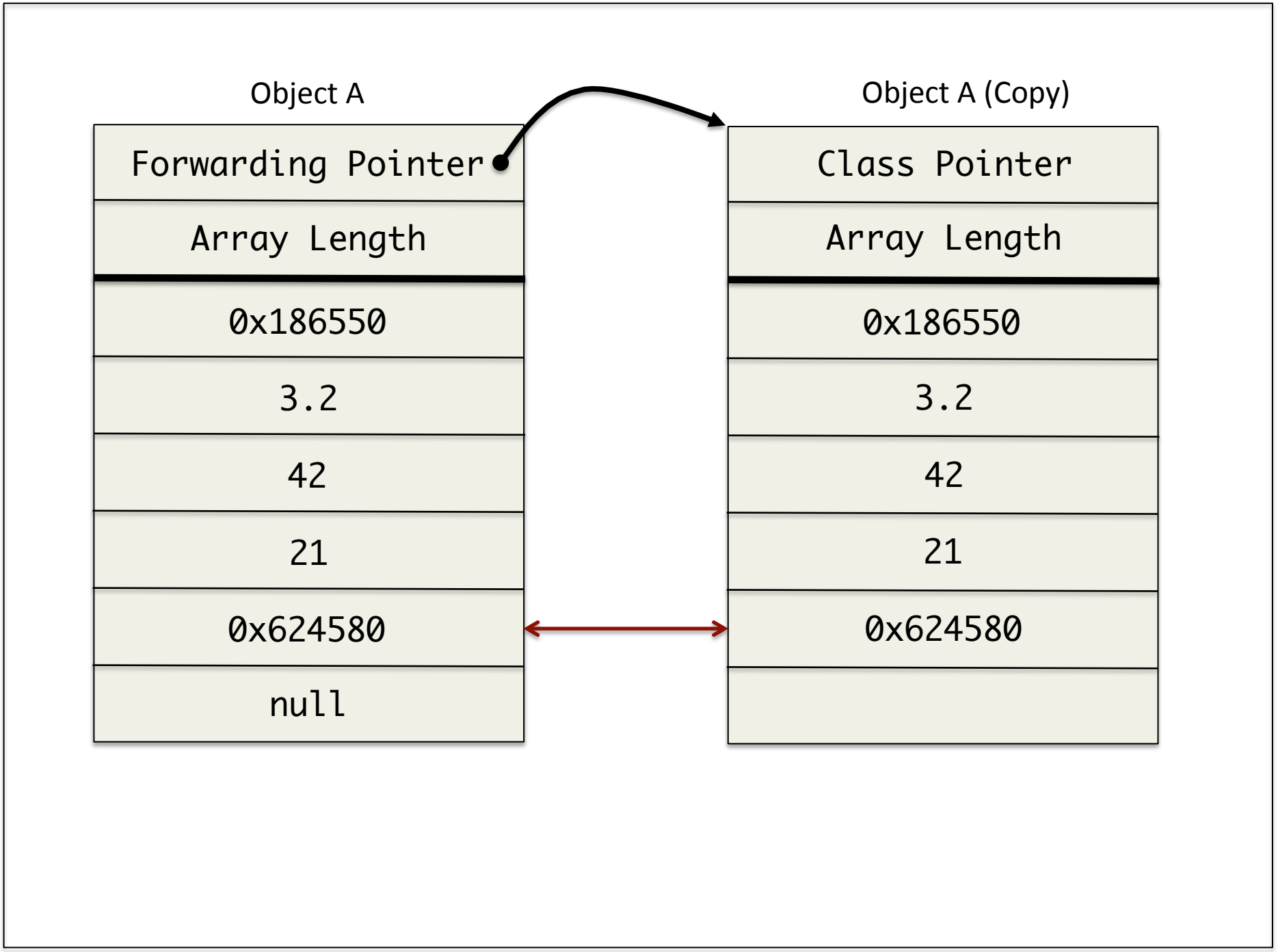

Object A
Forwarding Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2

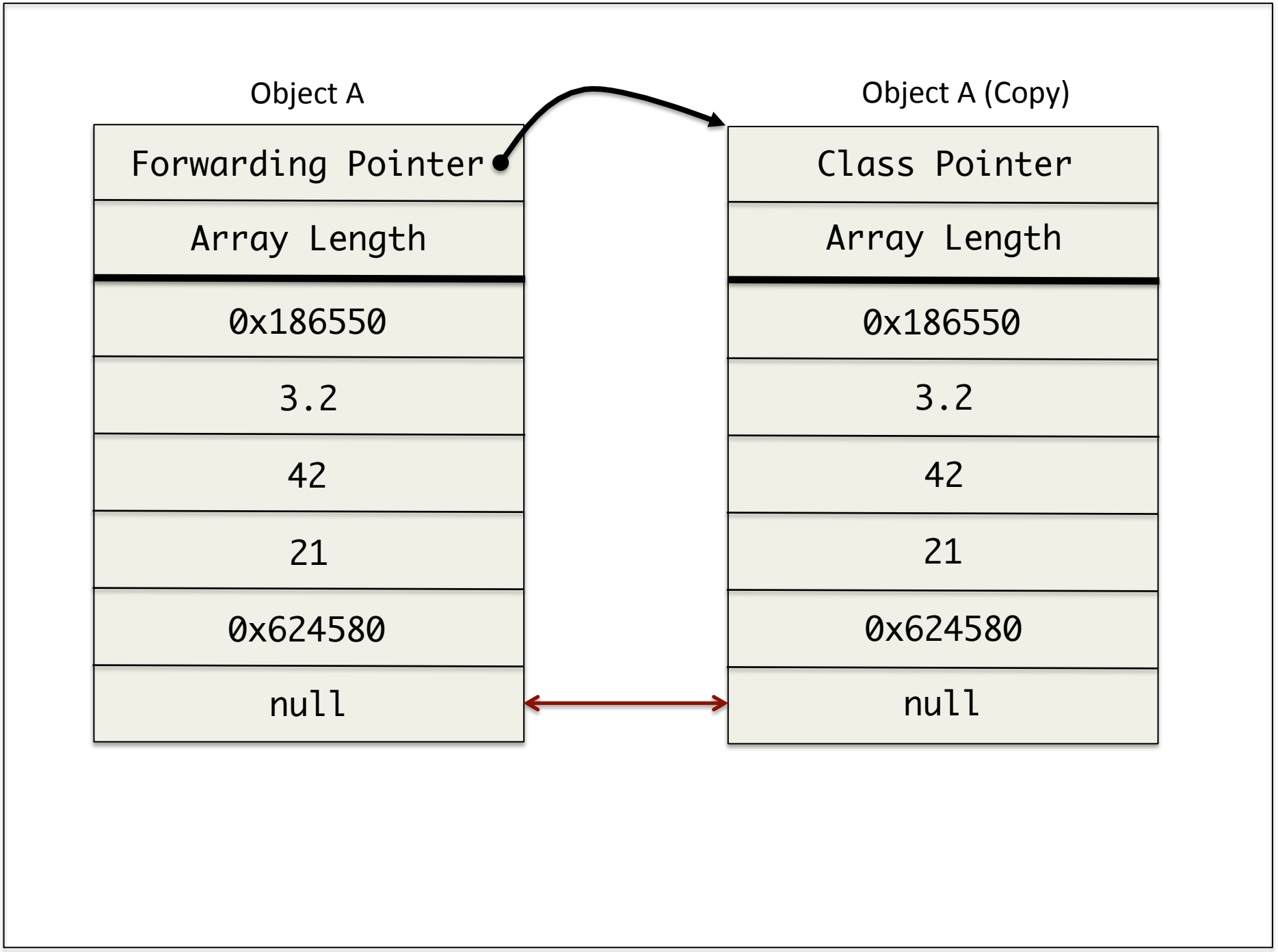

Object A
Forwarding Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
7


Thread 2
Object A
Object A (Copy)
Forwarding Pointer
Array Length
0x186550
3.2
7
21
0x624580
null
Class Pointer
Array Length
0x186550
3.2
7


Thread 2
Object A
Object A (Copy)
Forwarding Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Class Pointer
Array Length
0x186550
3.2
42


Object A
Forwarding Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
42
21


Object A
Forwarding Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
42
21
0x624580


Object A
Forwarding Pointer
Array Length
0x186550
3.2
42
21
0x624580
null
Object A (Copy)
Class Pointer
Array Length
0x186550
3.2
42
21
0x624580
null

Sapphire Pros and Cons

- Enables lock-free concurrent copying
  - Synchronization between threads use atomics
  - Much lower overhead than locks
- Mutator ends up doing a lot more work
  - Follows forwarding pointers
  - Double writes
  - Complex write barrier

Concurrent Copying

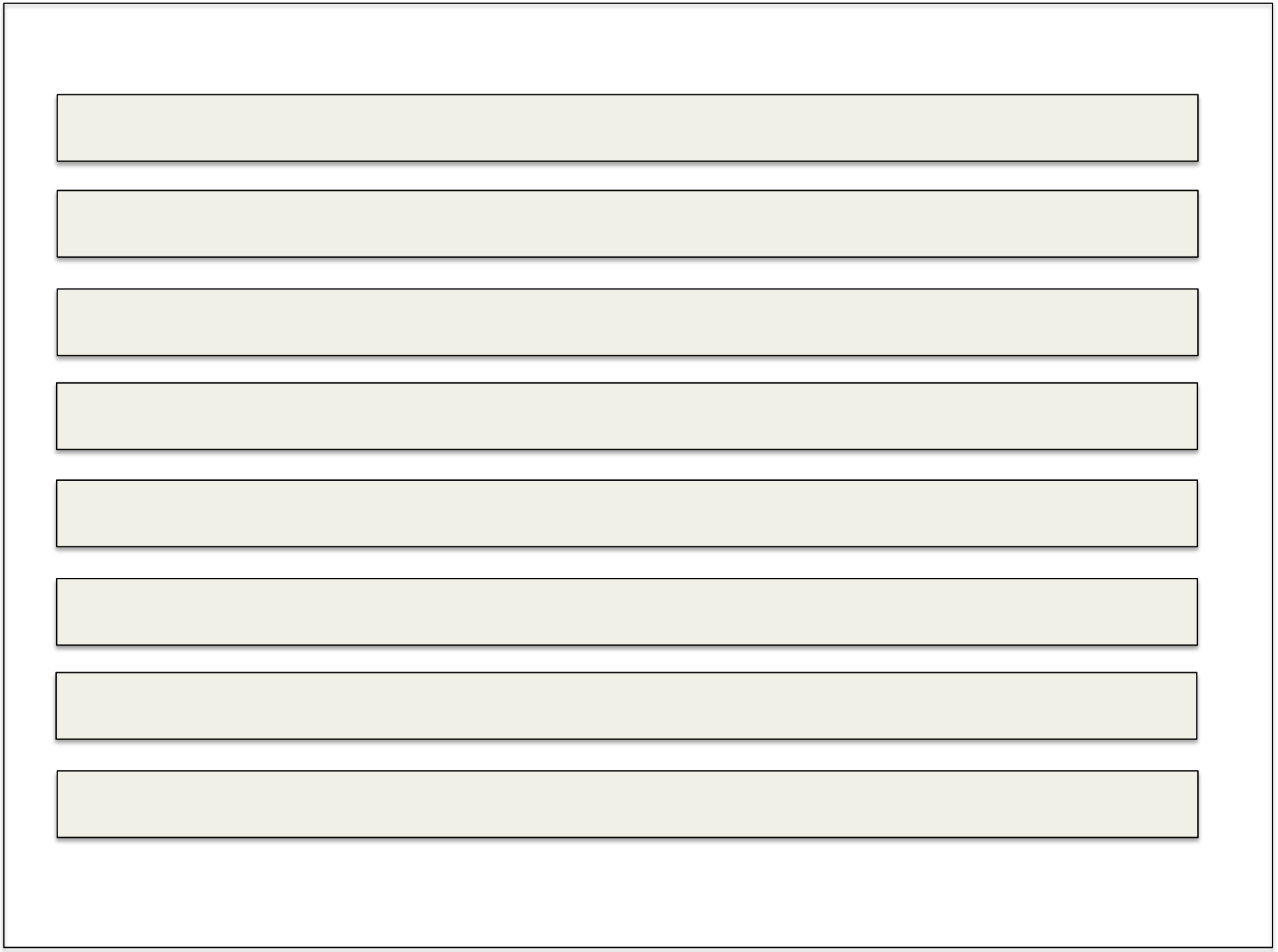
- Generally, the overheads aren't worth it
  - Increased write barrier complexity
  - More synchronization operations bad for the cache
- Most algorithms stop the world to copy
  - Limit each pause to a small region of the heap
  - Target regions likely to have more garbage
- Copying can be highly parallelized

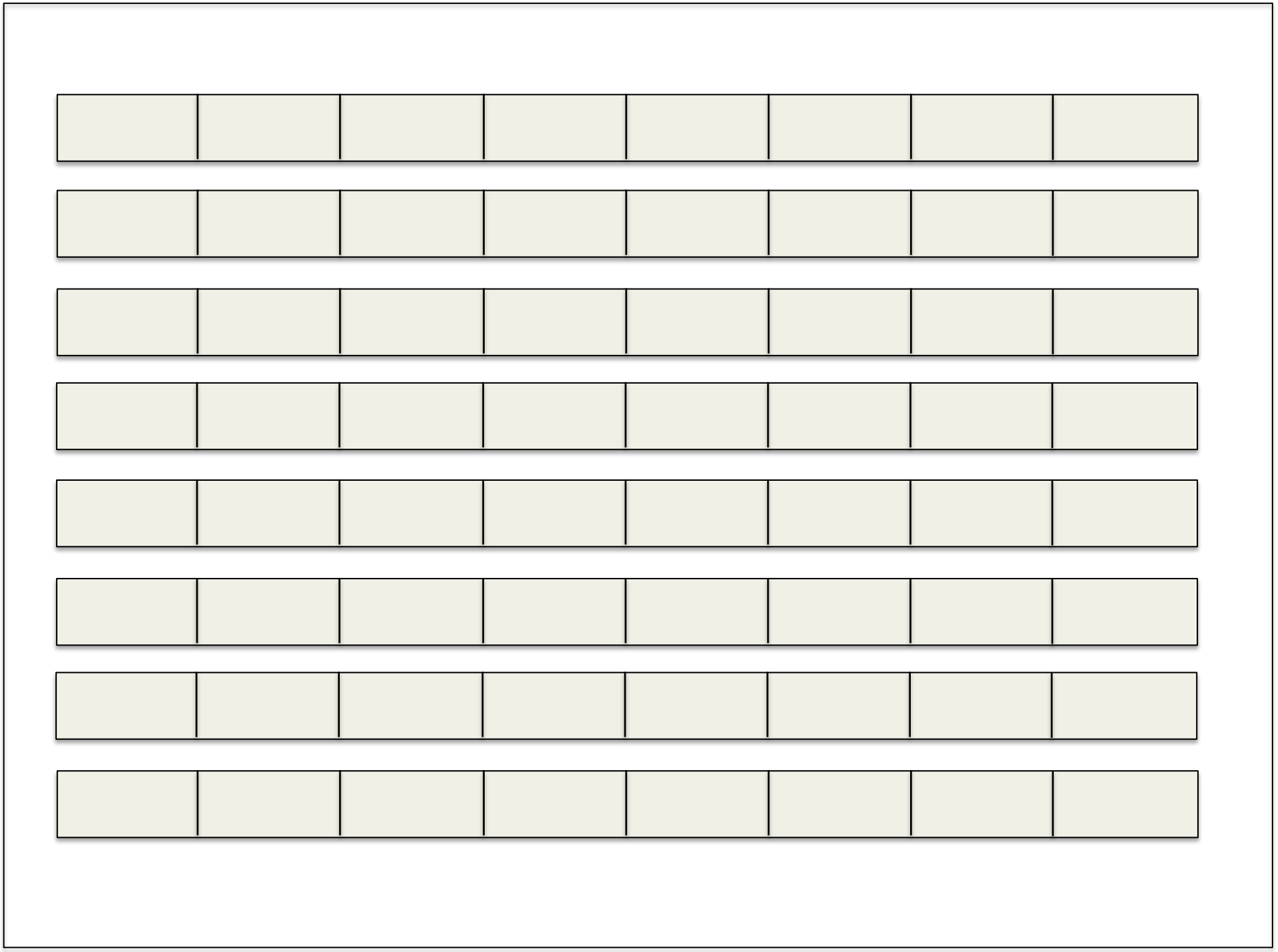
Garbage First Algorithm

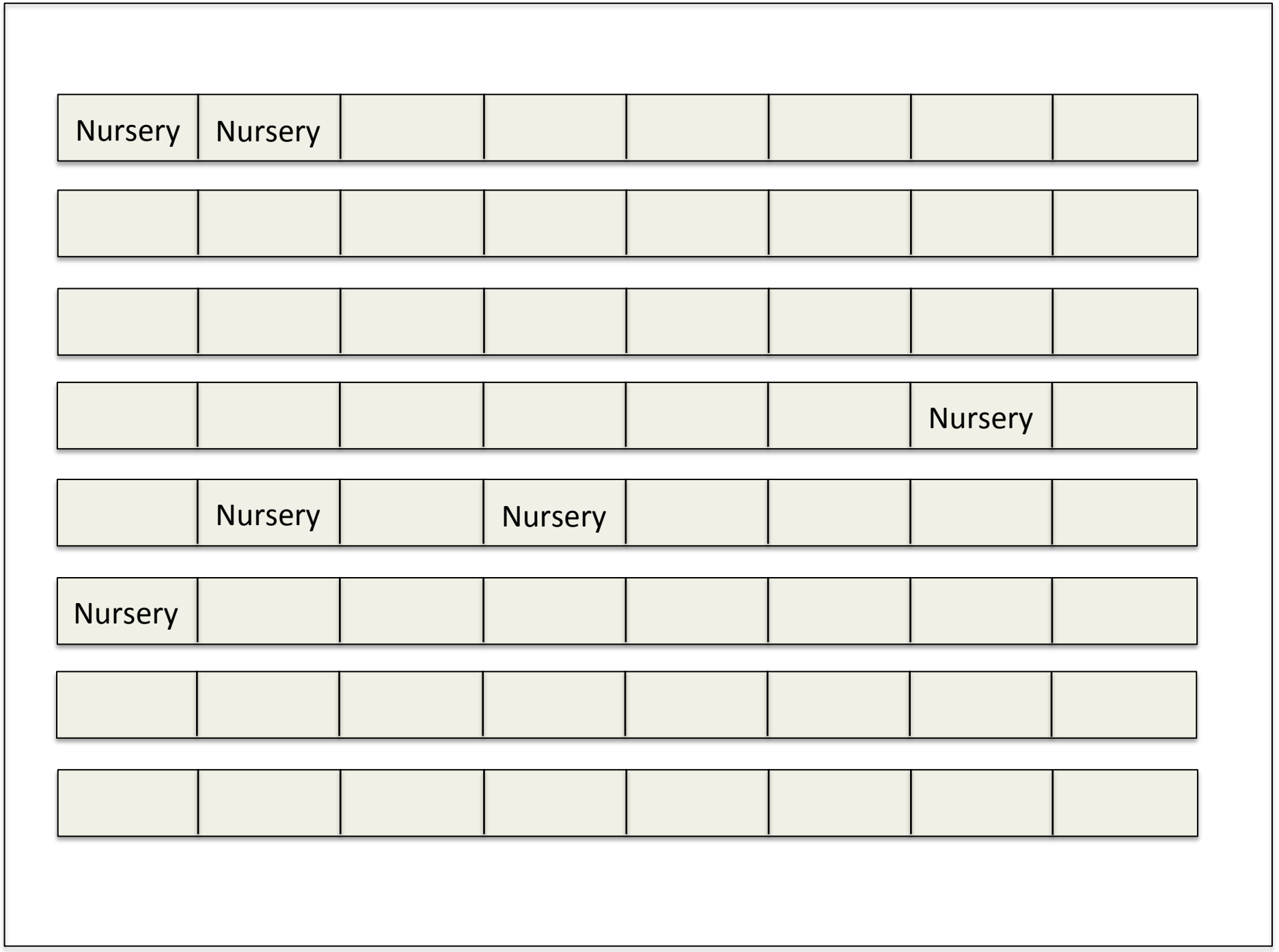
- Latest iteration of the Hotspot collector
  - Not enabled by default
  - Targets pause times over throughput
  - Many server applications want the inverse
- GC is parallel and concurrent
  - Copies objects to compact the heap
  - Pauses are predictable and tunable
  - Offers soft real-time without guarantees

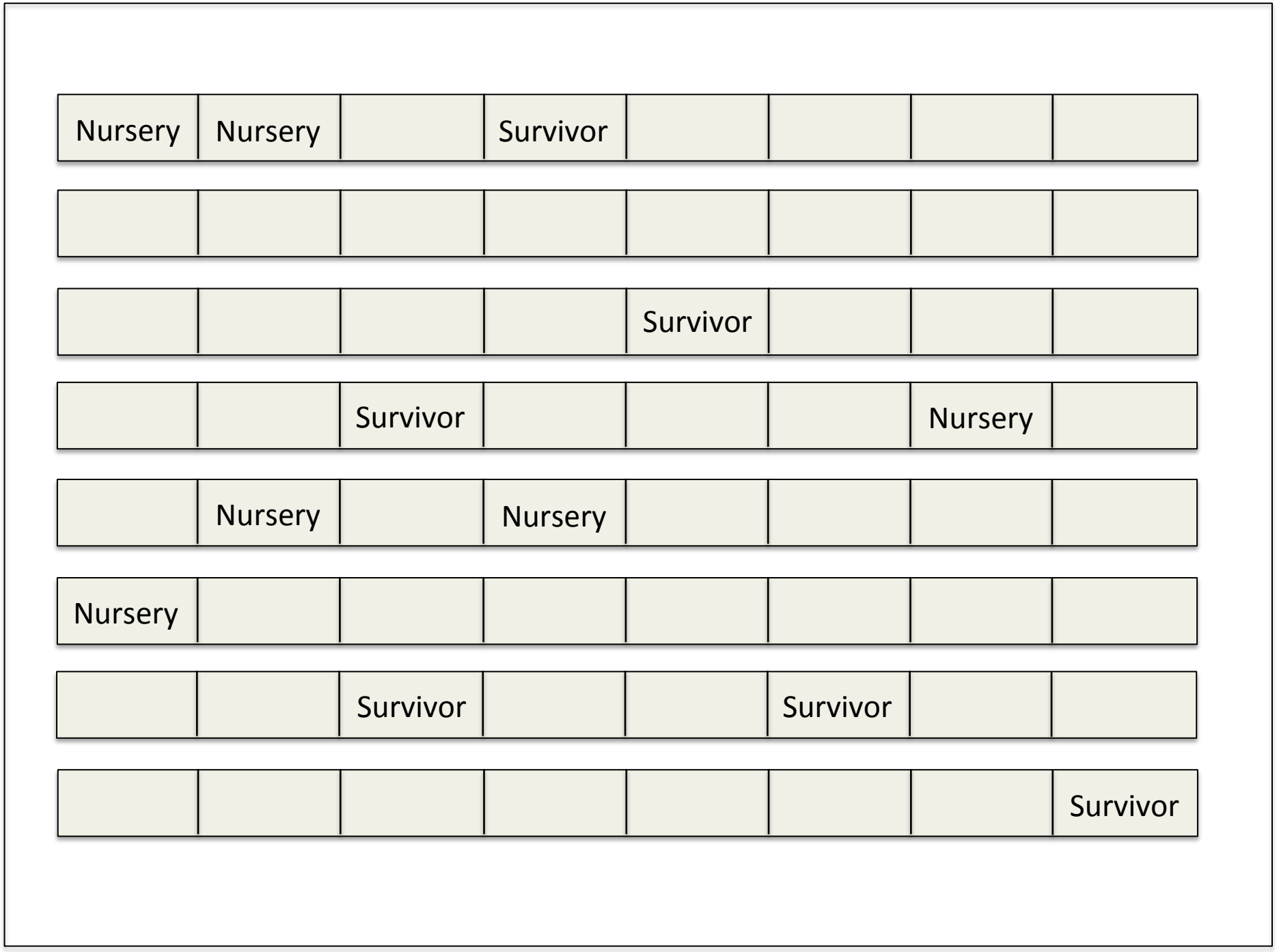
G1 Regions

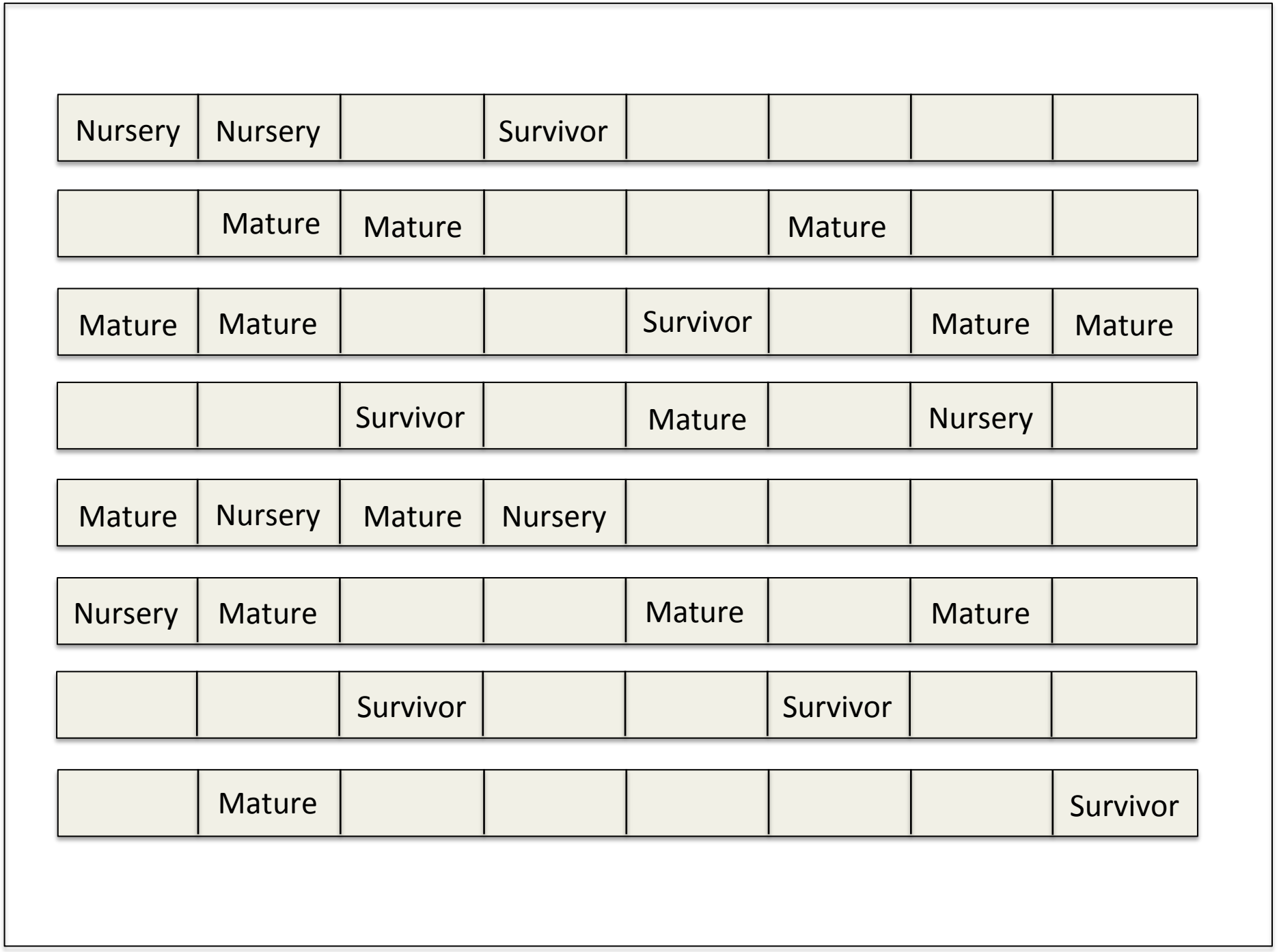
- Heap is split into equally-sized regions
  - Region sizes range from 1-32 Mb
  - Aim to have no more than 2048 regions
  - More tuning to enable more than 64 Gb heap
- Young generation made up of two block types
  - Nursery (or Eden)
  - Survivor set
- Very large objects given a region to themselves
  - Multiple regions can coalesce if necessary

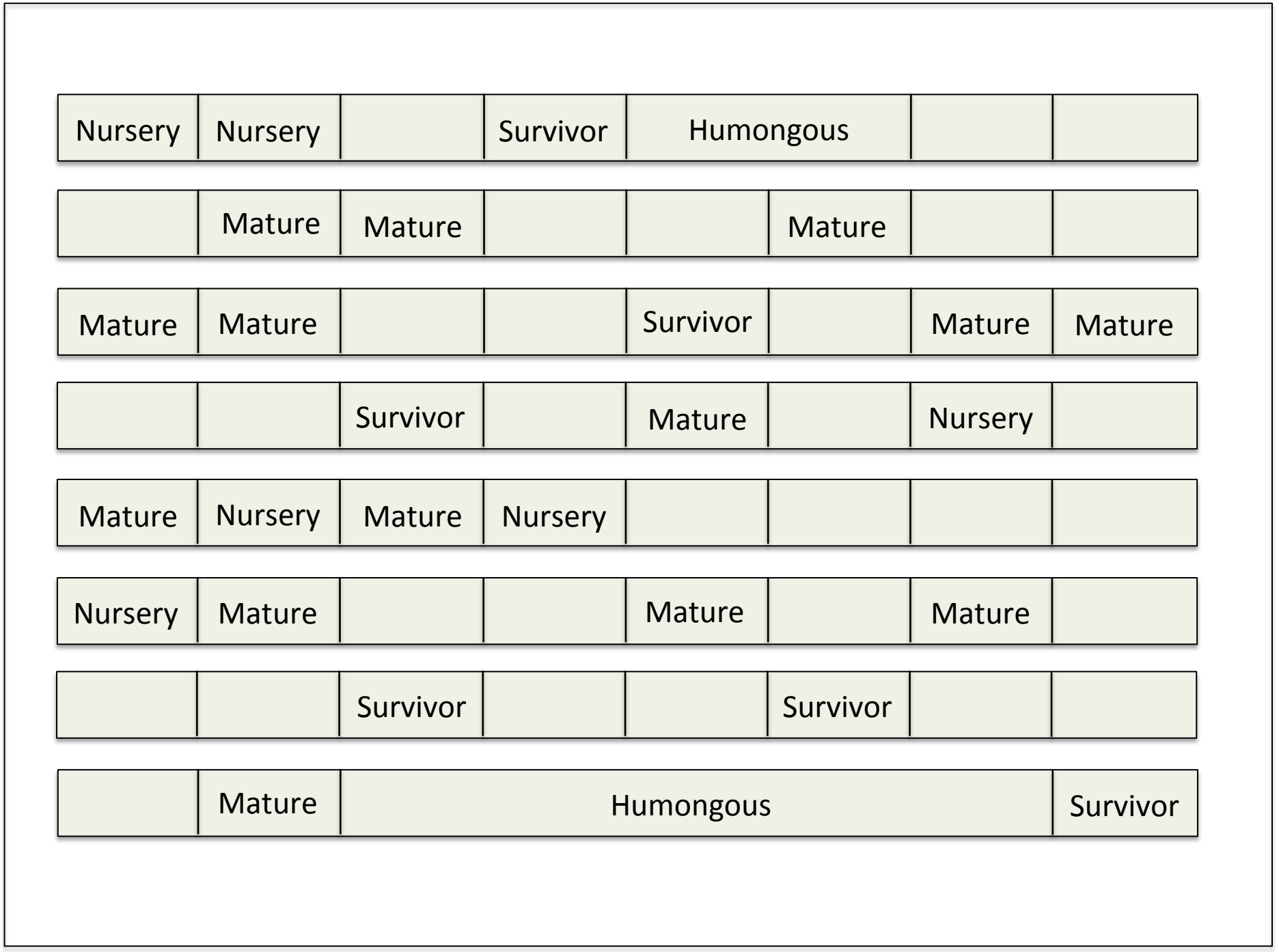





Nursery
Nursery
Nursery
Nursery
Nursery
Nursery
Nursery


Nursery
Nursery
Survivor
Survivor
Survivor
Nursery
Nursery
Nursery
Survivor
Survivor
Survivor


Nursery Nursery Survivor
Mature Mature Mature
Mature Mature Survivor Mature Mature
Survivor Mature Nursery
Mature Nursery Mature Nursery
Nursery Mature Mature Mature
Survivor Survivor
Mature Survivor


Nursery Nursery Survivor Humongous
Mature Mature Mature
Mature Mature Survivor Mature Mature
Survivor Mature Nursery
Mature Nursery Mature Nursery
Nursery Mature Mature Mature
Survivor Survivor
Mature Humongous Survivor

Remembered Sets

- Maintain remembered sets for each region
  - Track references to that region from outside
  - Don't track internal references
- Maintain set using a write barrier
  - Use block alignment to generate region ID
  - Simple shift operation
- Remembered sets give estimate of live data

Minor GC

- Parallel stop-the-world copying collection
- Promote all live young objects
  - Move objects from nursery to the survivor regions
  - Move objects from the survivor to the mature
- Don't evacuate all young objects at once
  - Determine nursery regions by pause time
  - Determine survivor regions by garbage ratio estimates

Concurrent Marking

- Initial marking to scan the root set
  - Requires that threads be stopped
  - Piggy-backs on a minor GC pause
- Concurrent marking across the heap
  - Uses incremental update
- Remark phase to complete incremental update
  - Stops the world

Cleanup Phase

- After mark phase we know all live objects
  - Remembered sets aggregate the information
- We can explicitly target regions with more garbage
  - Minimize copy overhead
  - Reclaim more garbage in a bounded time
- In many cases a region is entirely garbage
  - Remember the train algorithm

Evacuation

- Final part of major collection
- Iterate over regions selected for copying
  - Copy all live objects to an empty region
  - Highly parallel
  - Uses a form of TLAB to minimize contention
- Number of regions copied can be tuned
  - Trade-off throughput for pause time