In the last two weeks

- Assignment 2 was due yesterday
  - Feedback survey now posted
- Reading on conservative GC
- Questions on allocation algorithms
  - How would you choose which to use

Assignment 2

- More conceptual than code intensive
  - Deeper dive into some of the issues from class
  - More open in terms of implementation
- Quite a lot of questions and comments
  - Not many on the same topic
- A couple of bugs showed up in SimpleJava
  - Inline code reversed method parameters
  - Leaky abstraction in the profiler interface

Conservative GC

- Managed languages have precise type information
  - Many other languages don't
  - Can add a conservative GC without compiler support
- Conservative GC works around imprecise types
  - If it looks like a pointer, treat it as one
  - Can lead to false retention in some cases

Garbage Collecting C++

- Useful when the language doesn't support GC
  - Some comparisons with reference counting
  - Benefits in large, multi-person projects
- Bohem-Demers-Weiser collector
  - Allows conservative collection in C/C++
  - How watertight is the abstraction?
- How expensive is type accuracy?

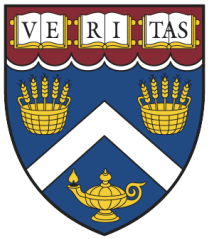
False Retention Rates

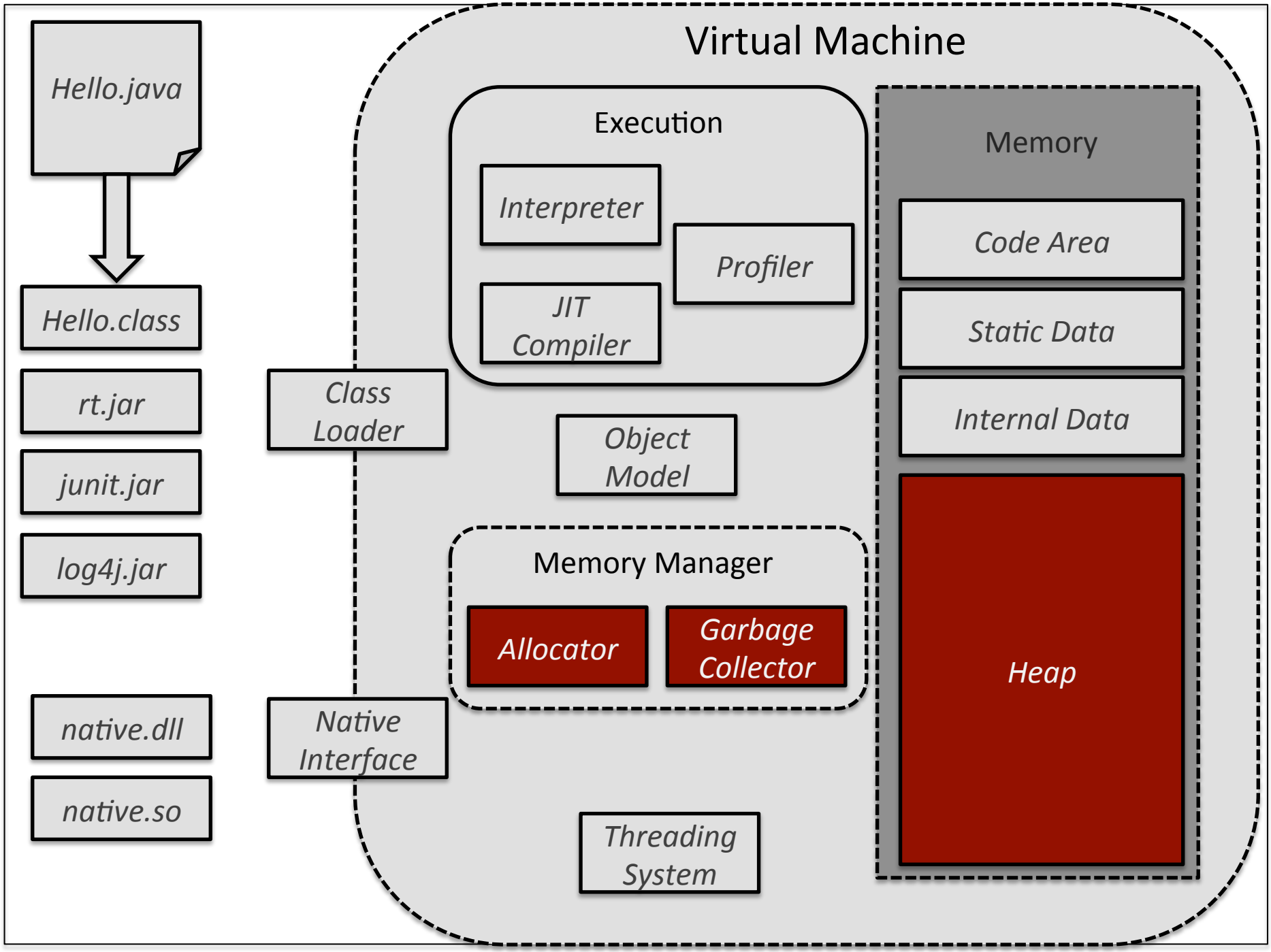
- Excess retention very low
  - On the order of 0.01% in Jikes RVM
- Does that add up over time?
- May still have to do some maintenance
  - Not good for a server environment

Target Applications

- Conservative GC good for small programs
  - Flash and Javascript programs tend to be small
  - Memory leaks don't have time to become serious
  - Is that still the case?
- The same languages tend to be user-facing
  - Pause times are important
  - Argument for reference counting?

Copying Garbage Collectors




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

Garbage Collection

- Data used by the application stored on the heap
  - Also some VM internal structures
- Objects are garbage when they're no longer used
  - Approximate using reachability
- Garbage collector frees unreachable objects

Reference Counting

- Keep track of all references to an object
- Some nice features
  - Incremental
  - Simple to implement and understand
  - Doesn't require runtime support
- One big problem
  - Garbage cycles

Stop The World Collectors

- Pause all mutator threads
- Do garbage collection work
- Restart mutator threads

- Can take a big-picture view of the heap
  - Know that it won't change during collection

- Leads to large pause times
- Useful when studying algorithms

Mark and Sweep Collection

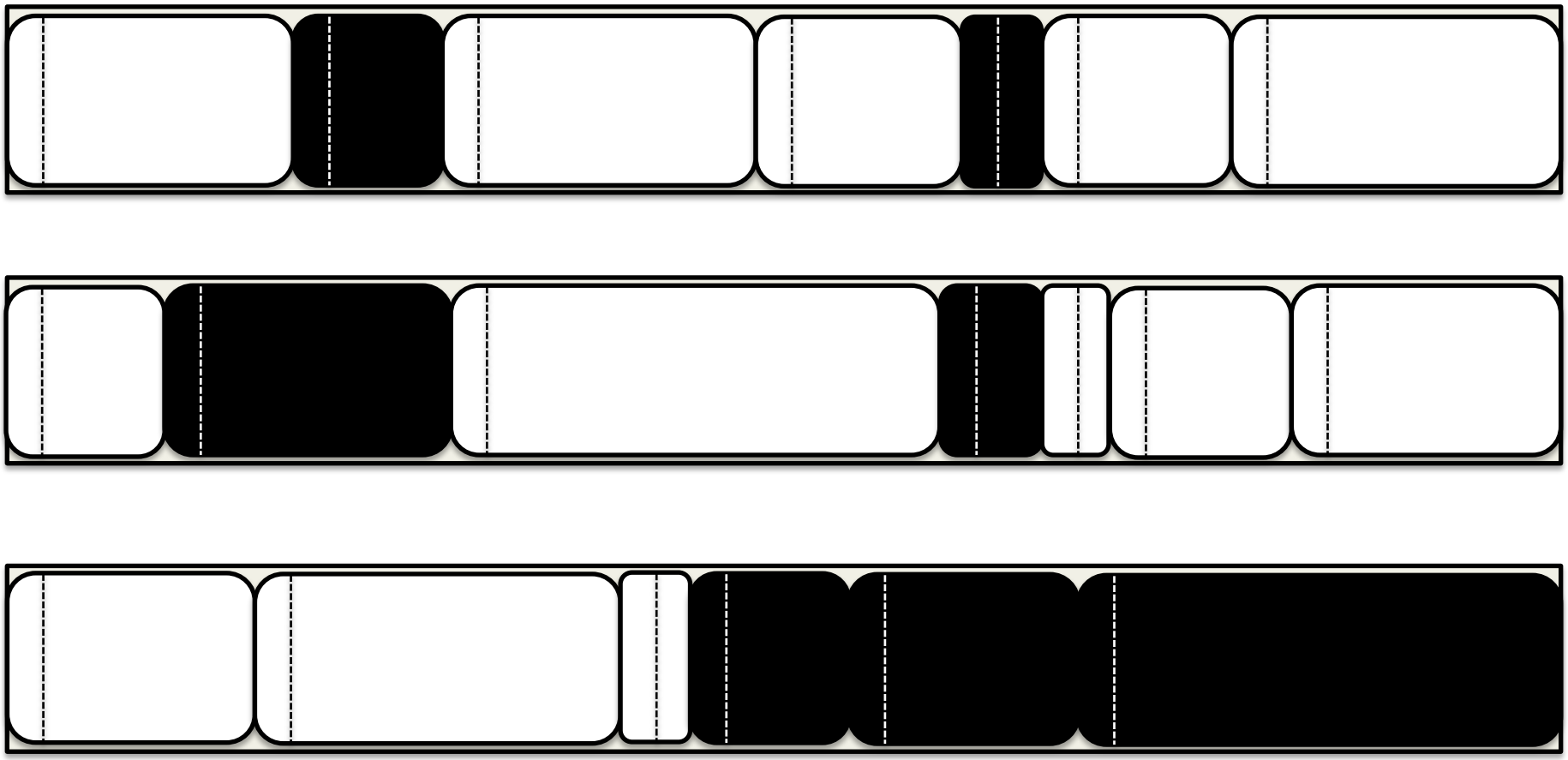
- Works out what objects are live
  - Traces the object graph from the system roots
  - Marks any objects that are reachable
- Scans linearly over the heap
- Returns unmarked objects to the free pool
- Introduces fragmentation to the heap

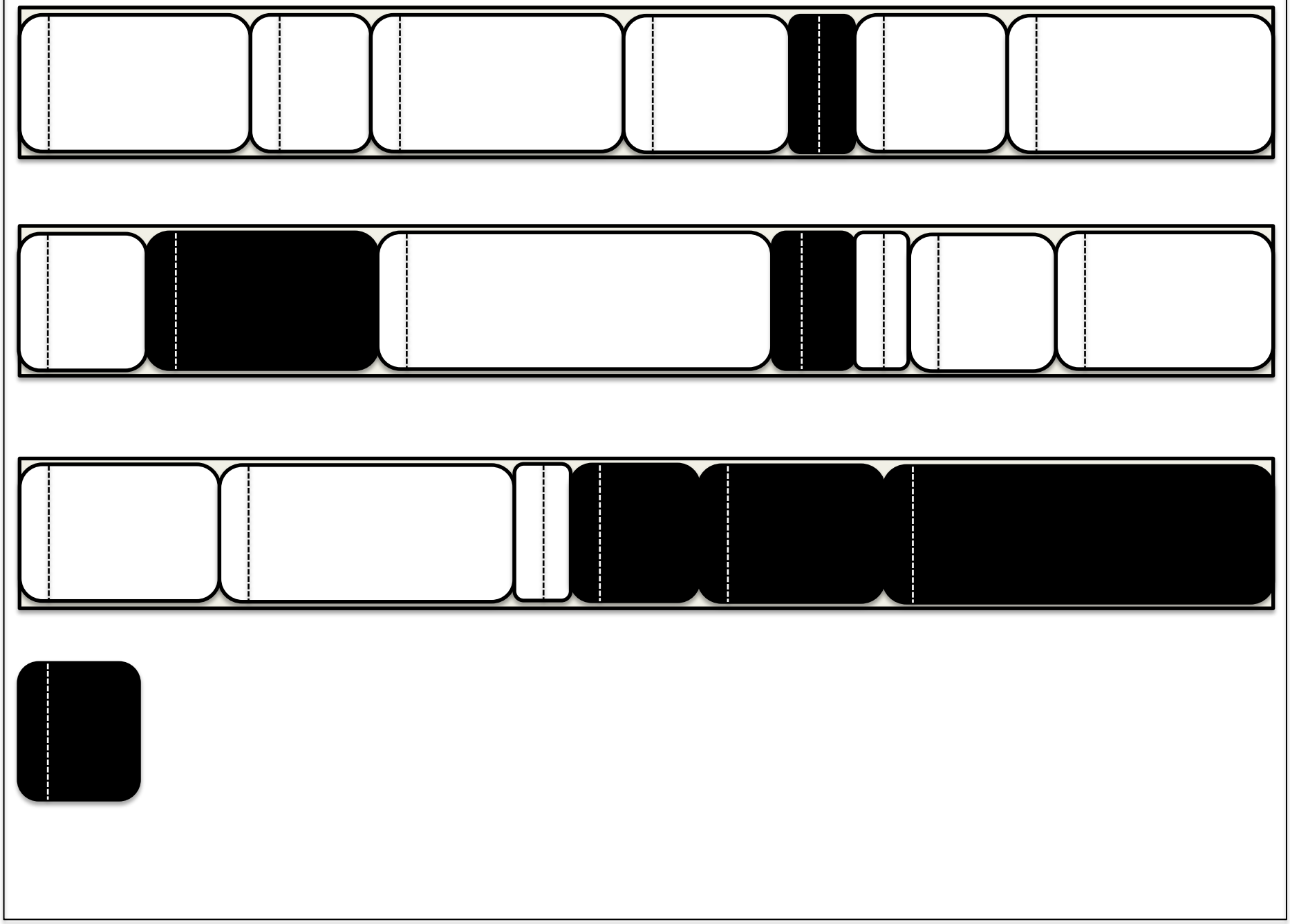
Fragmentation

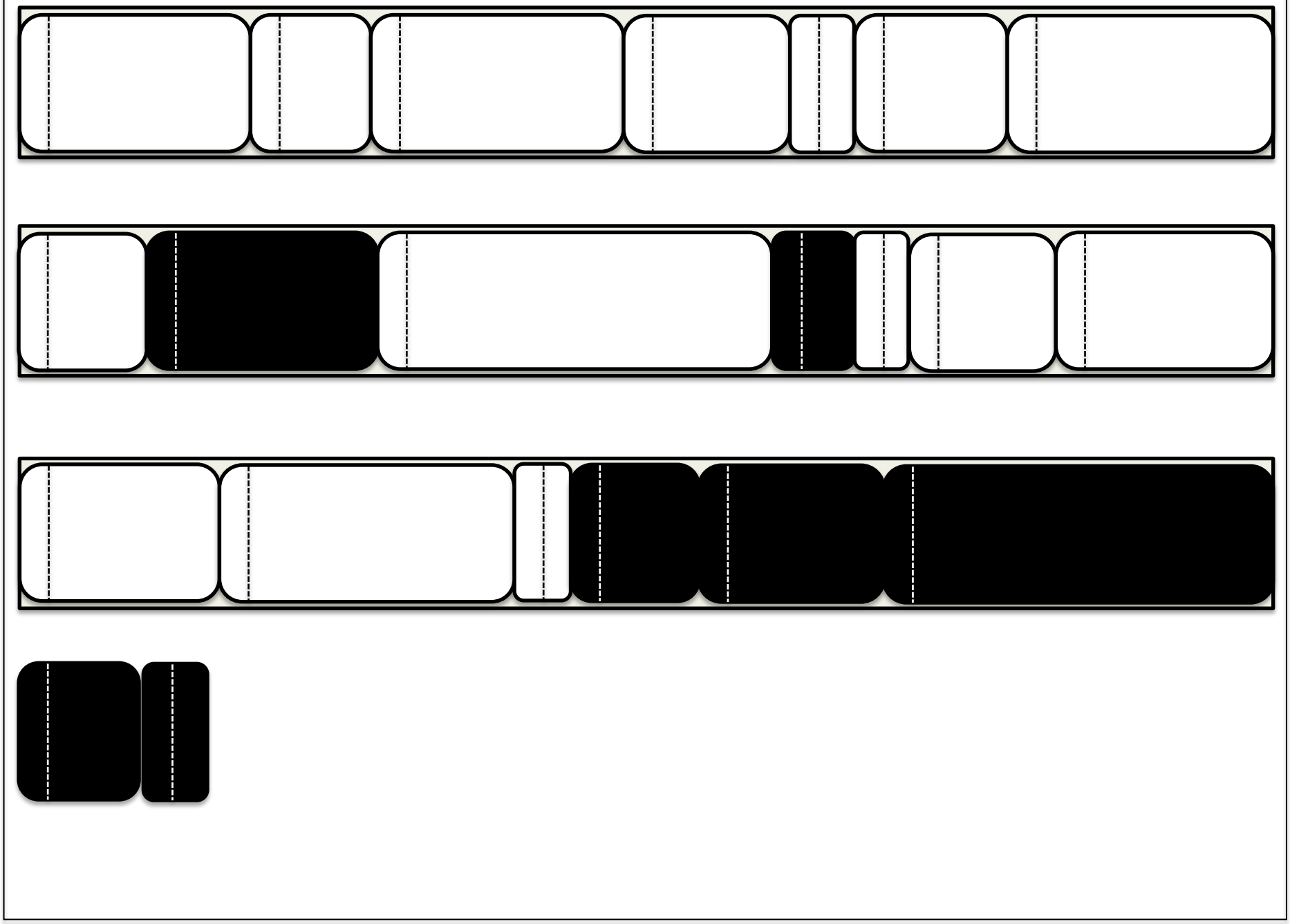
- Any collector that frees objects creates holes
  - Spaces where freed objects used to be
- Fragmentation can lead to wasted memory
  - There may not be an object of the correct size
- Allocation to fragmented memory is expensive
  - Need to figure out where to put objects
  - May not have a perfect fit

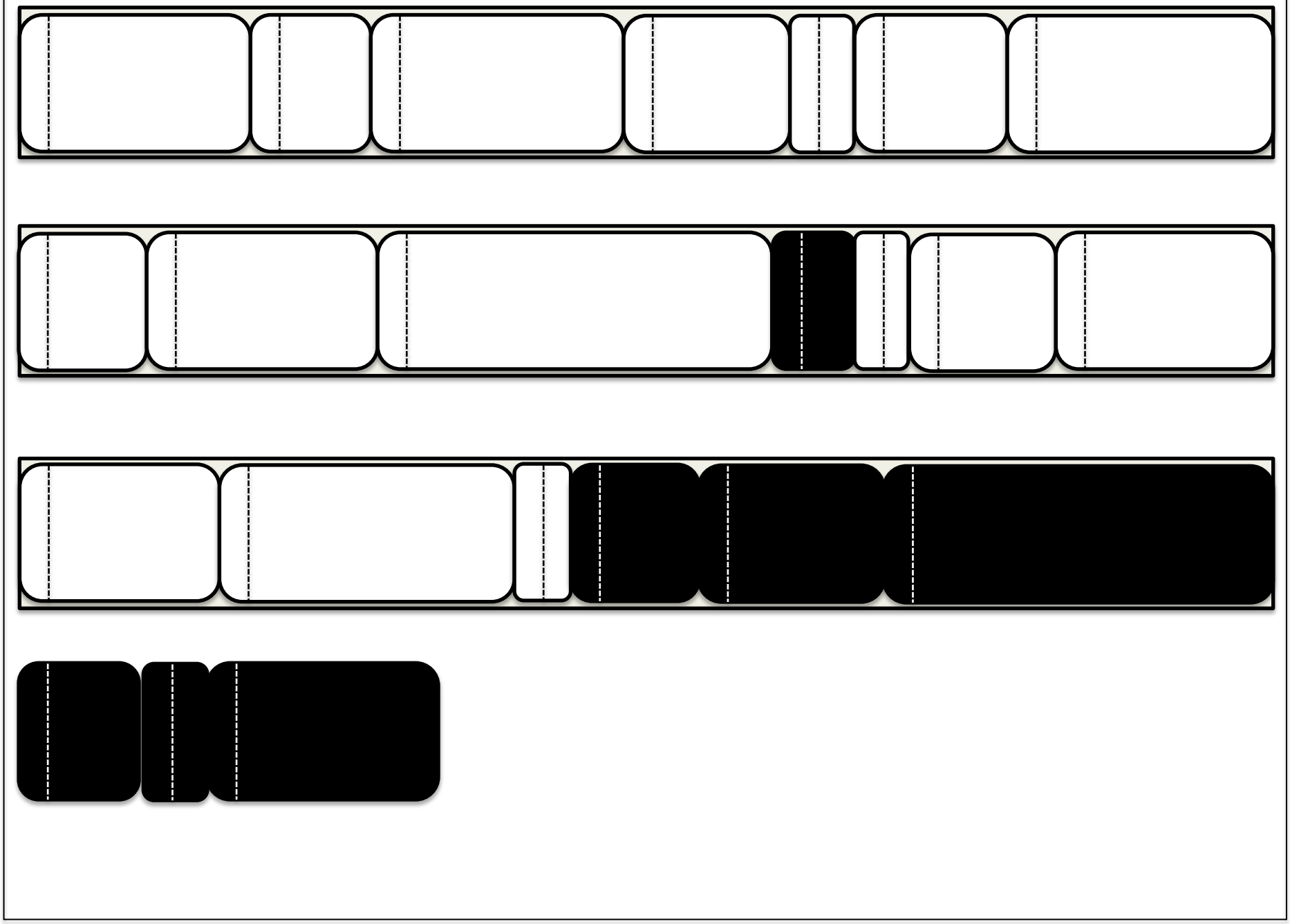
Eliminating Fragmentation

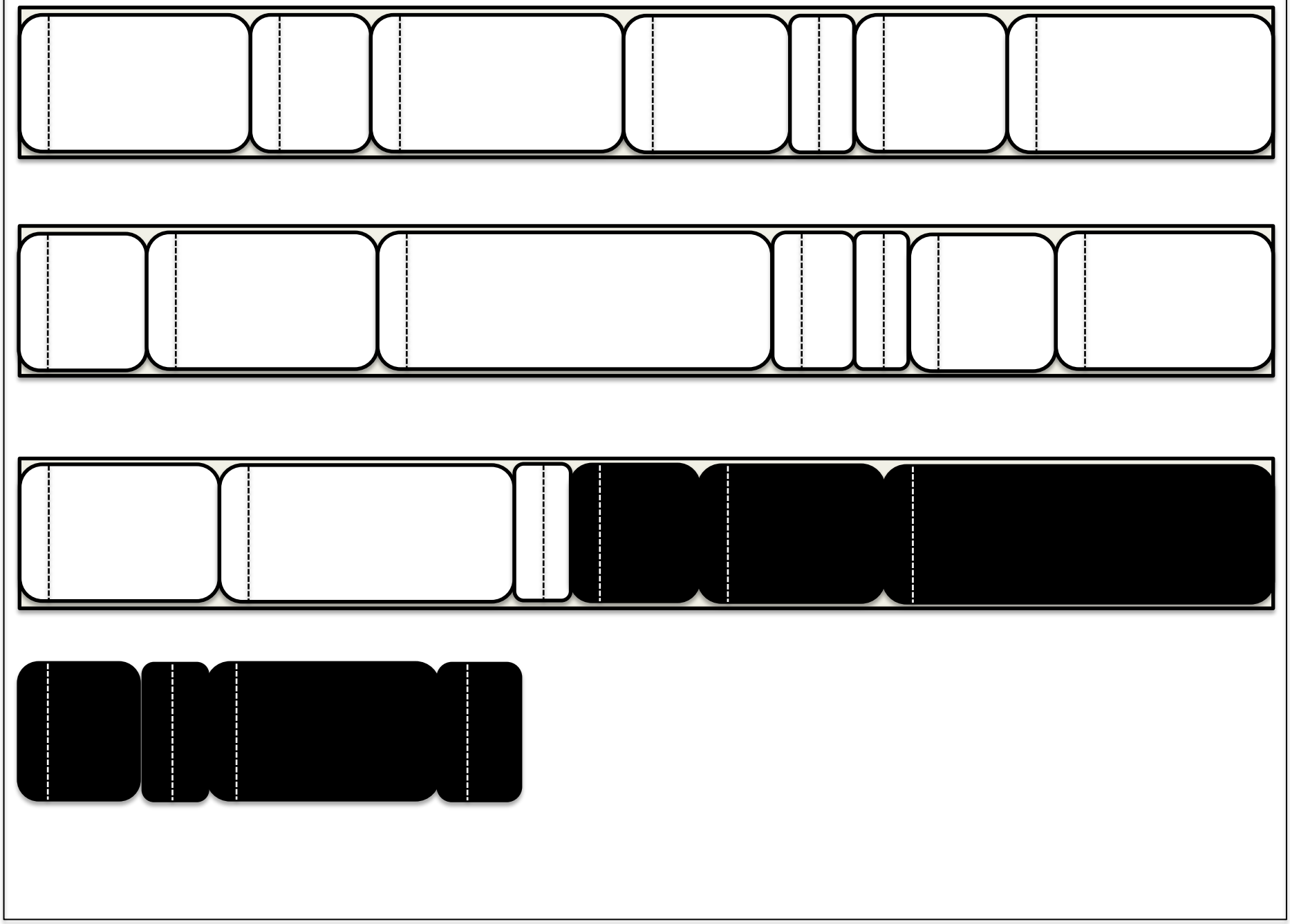
- Fragmentation is caused by freeing objects
- Garbage collector can compact the heap
  - Move all objects to a contiguous block
- MS and RC collectors free unreachable objects
- Copying collectors evacuate reachable objects
  - Everything left is garbage

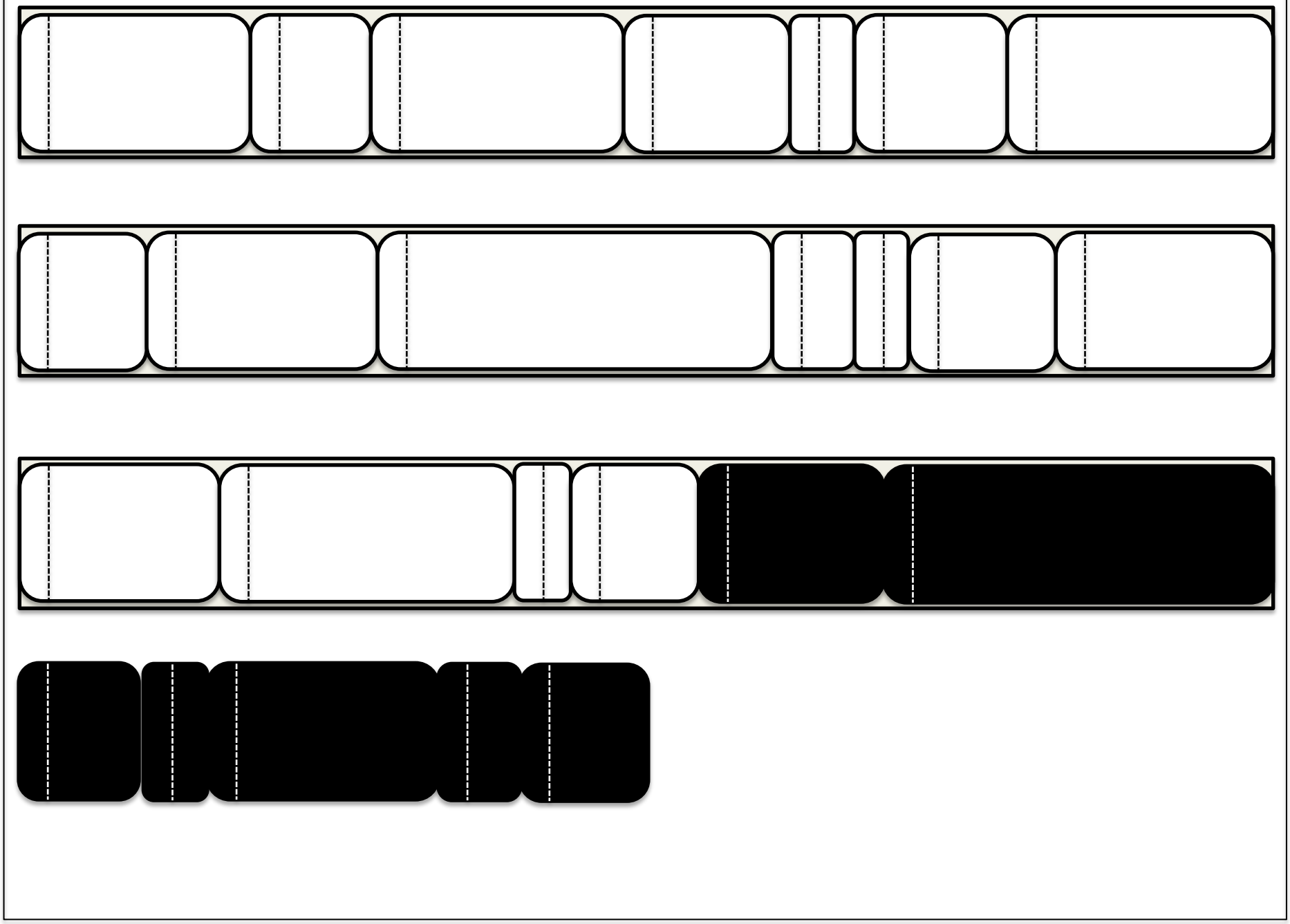


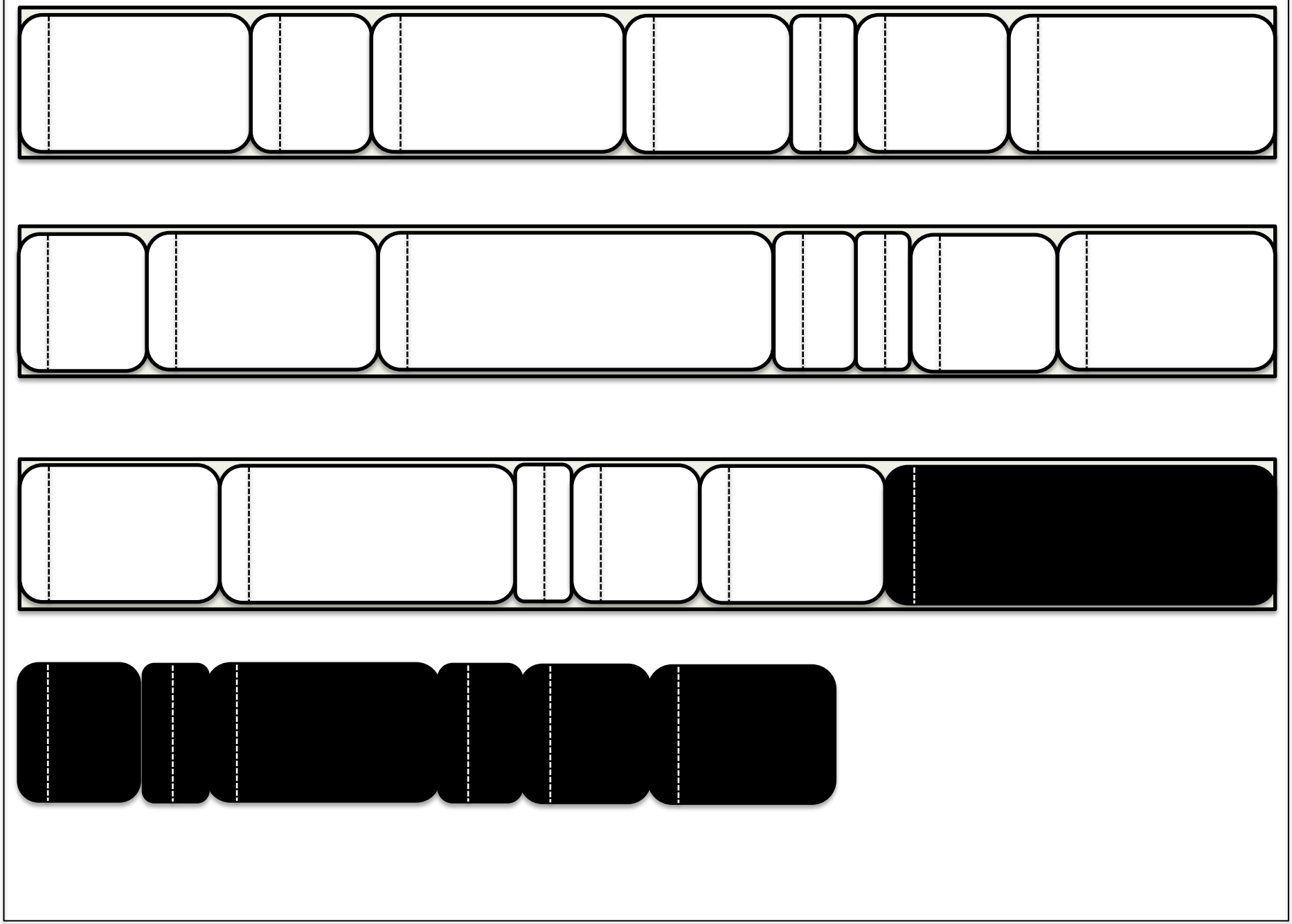


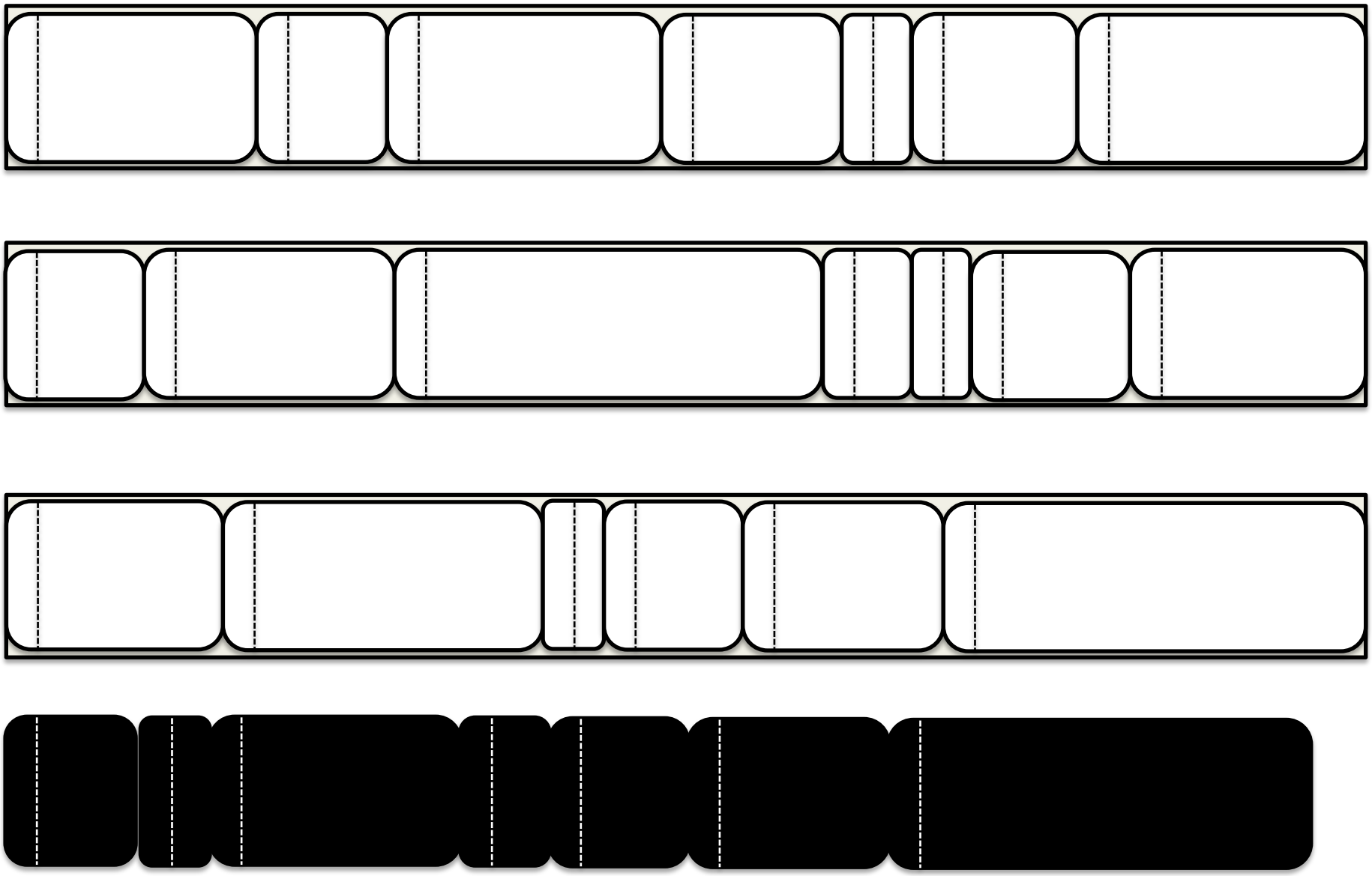


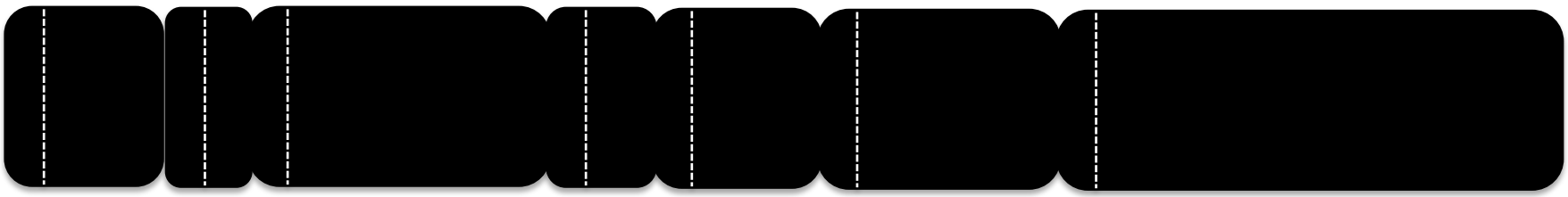










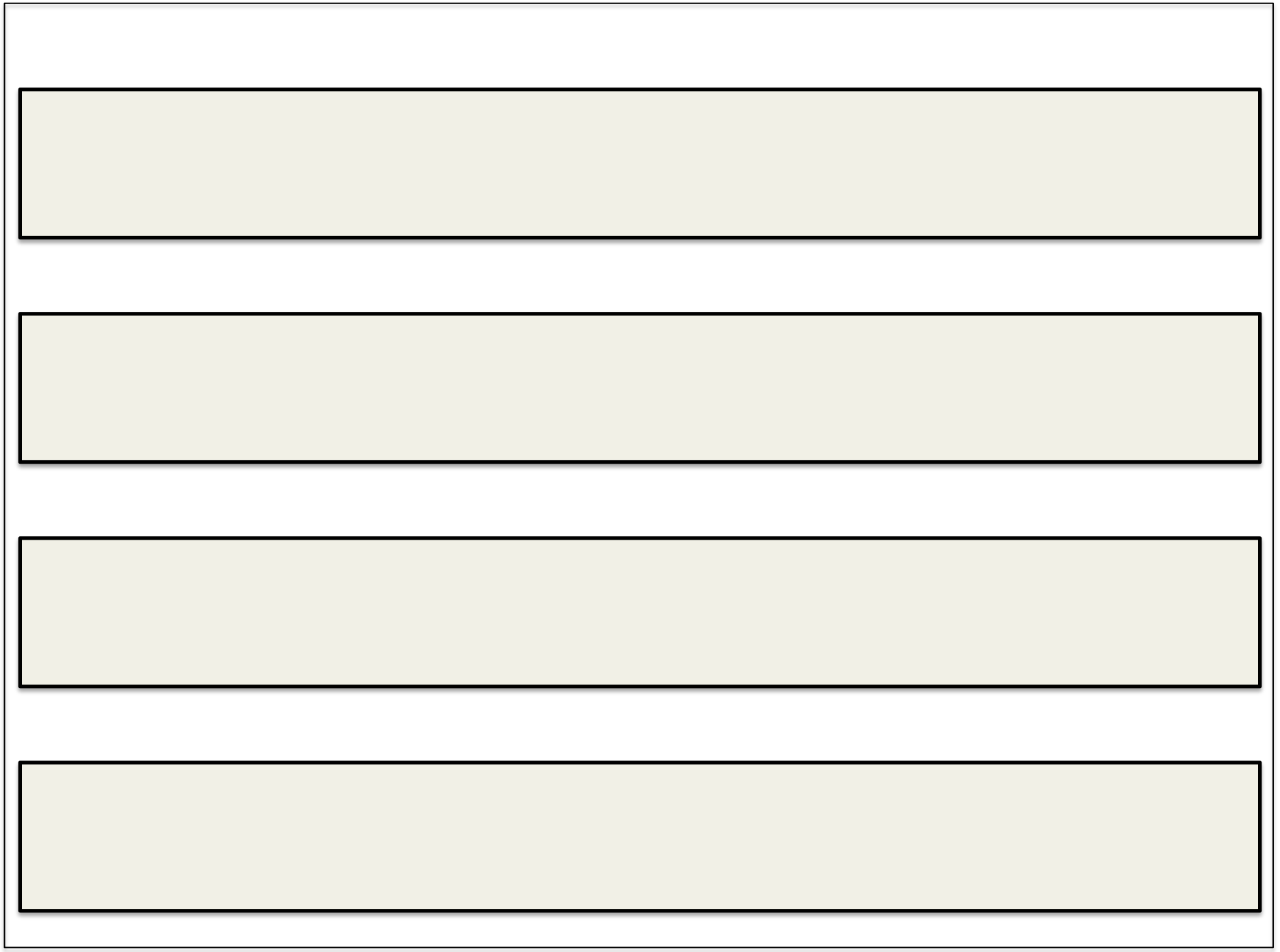


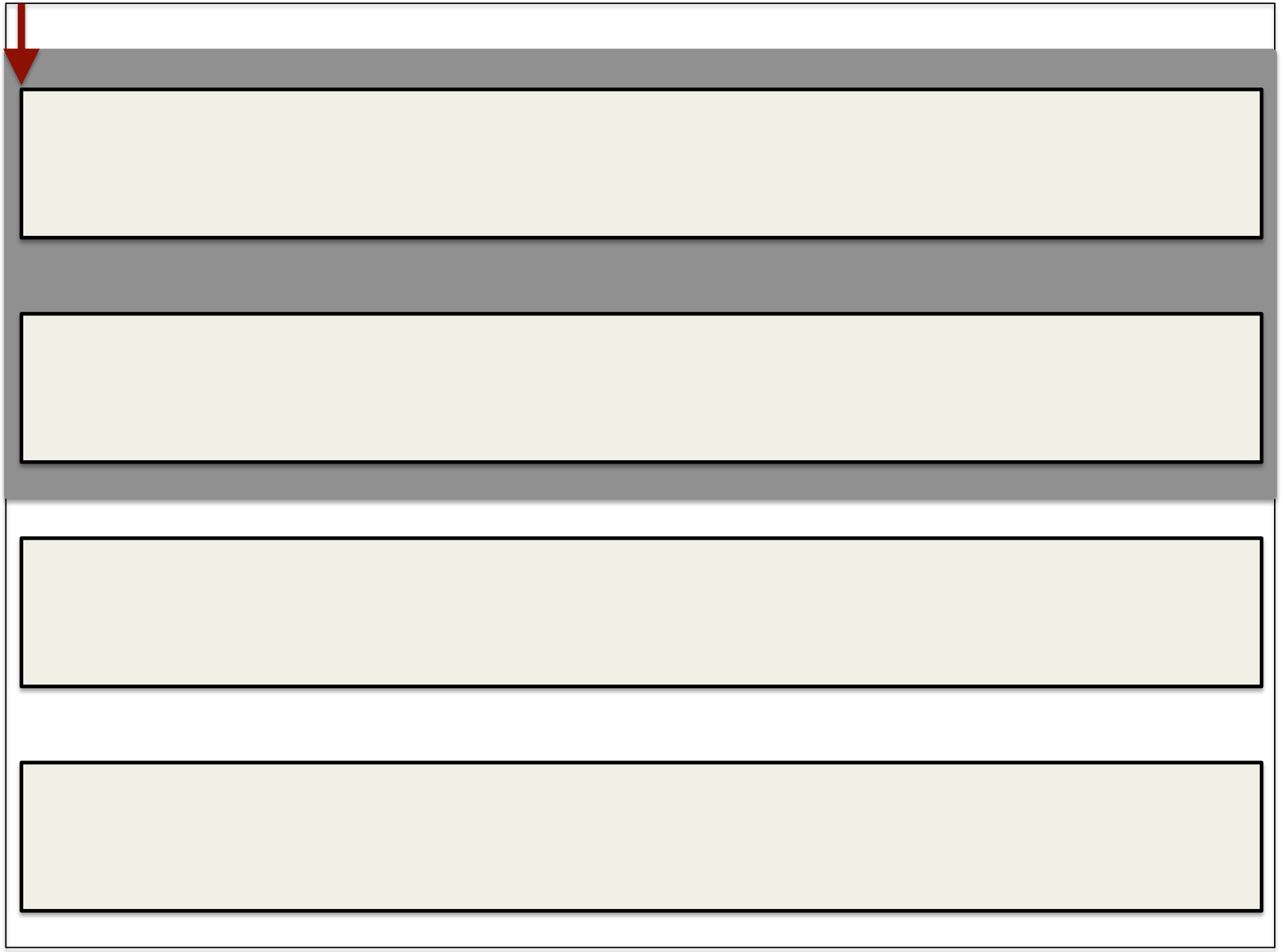
Copy Reserve

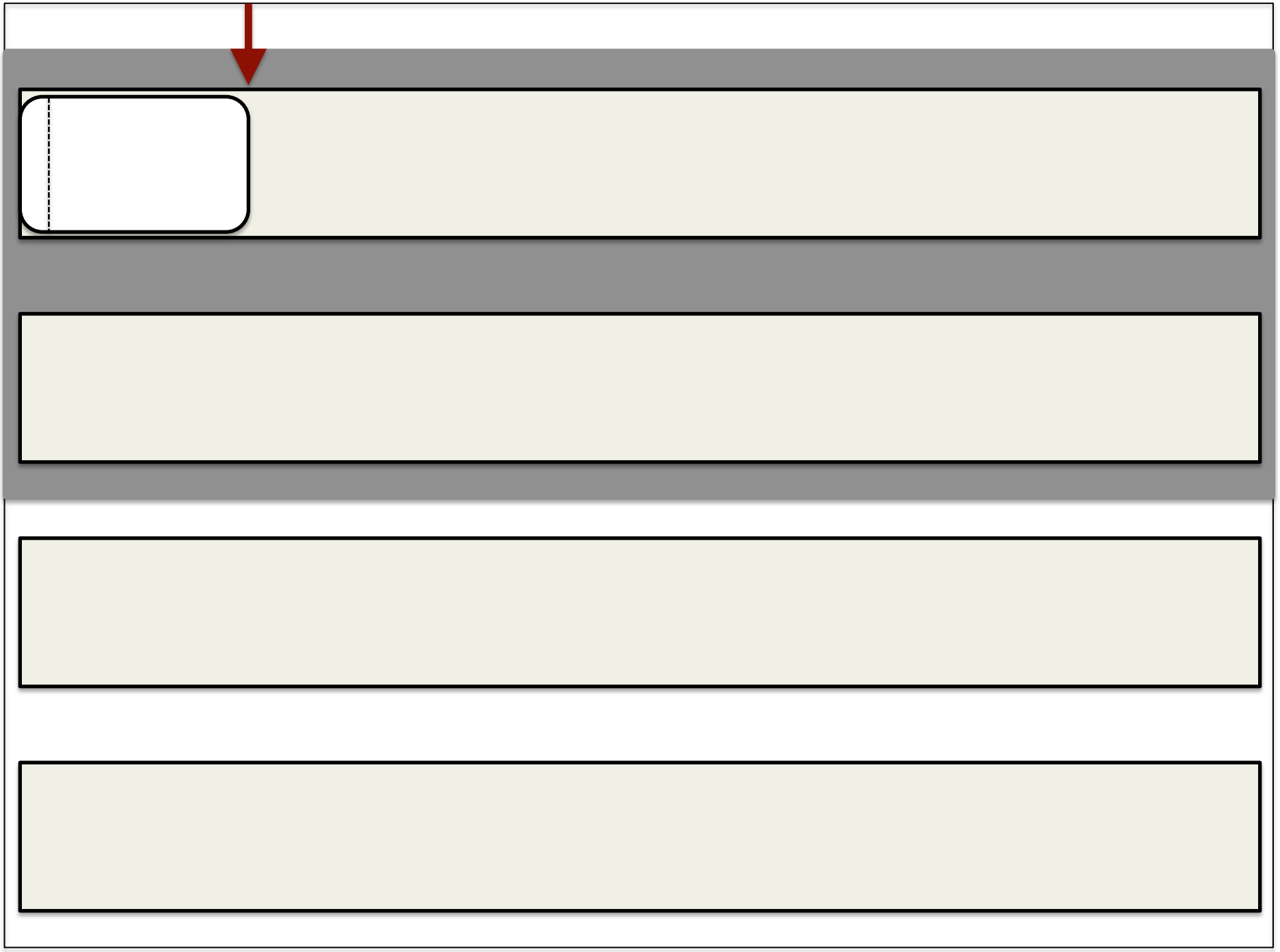
- Where do the copied objects go?
- We need to set aside space to copy them
  - In the worst case, all objects survive a collection
  - Need to set aside half the heap
  - We'll see later how to reduce this
- The extra space is called the copy reserve

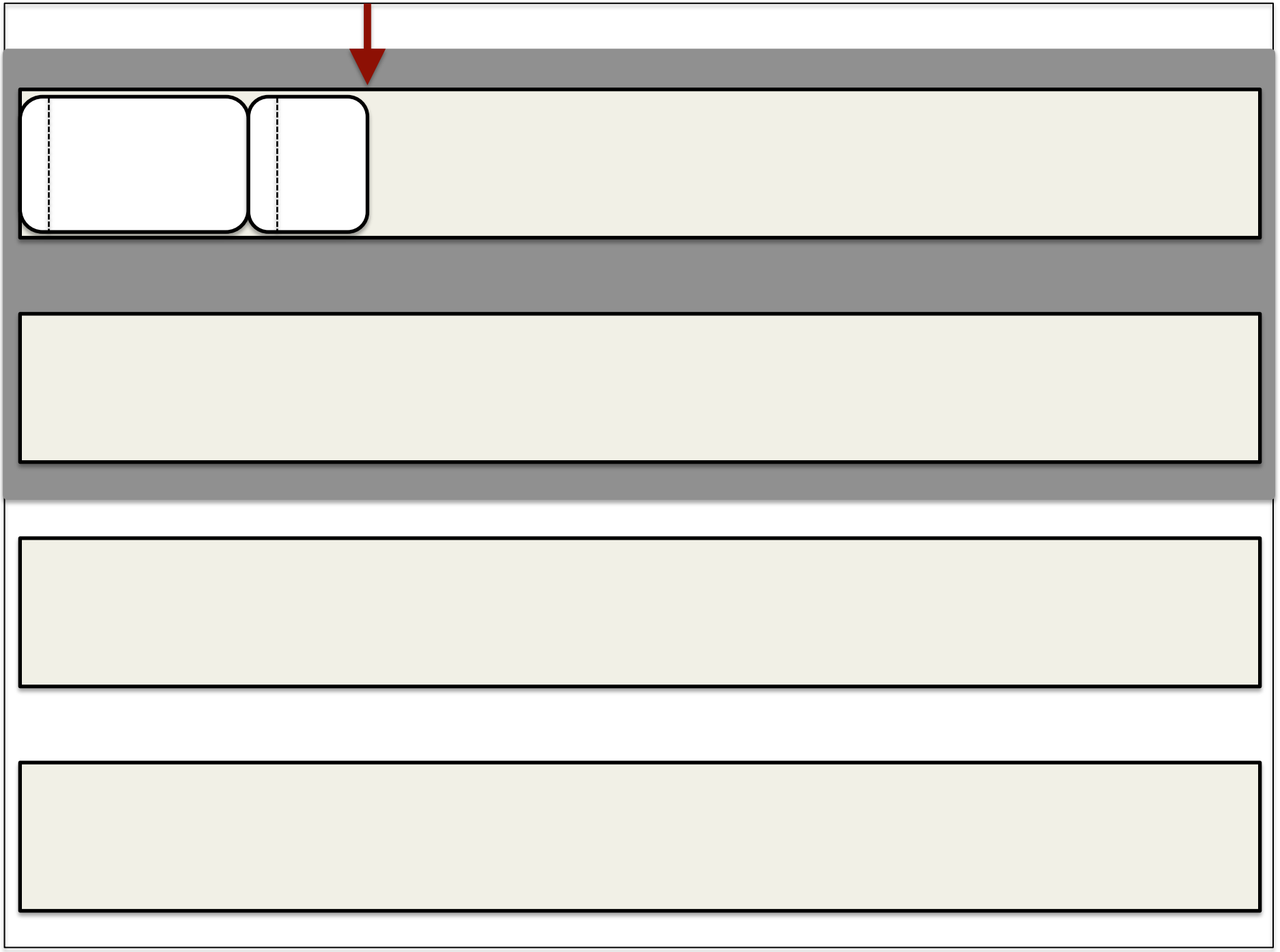
Semi-Space Collectors

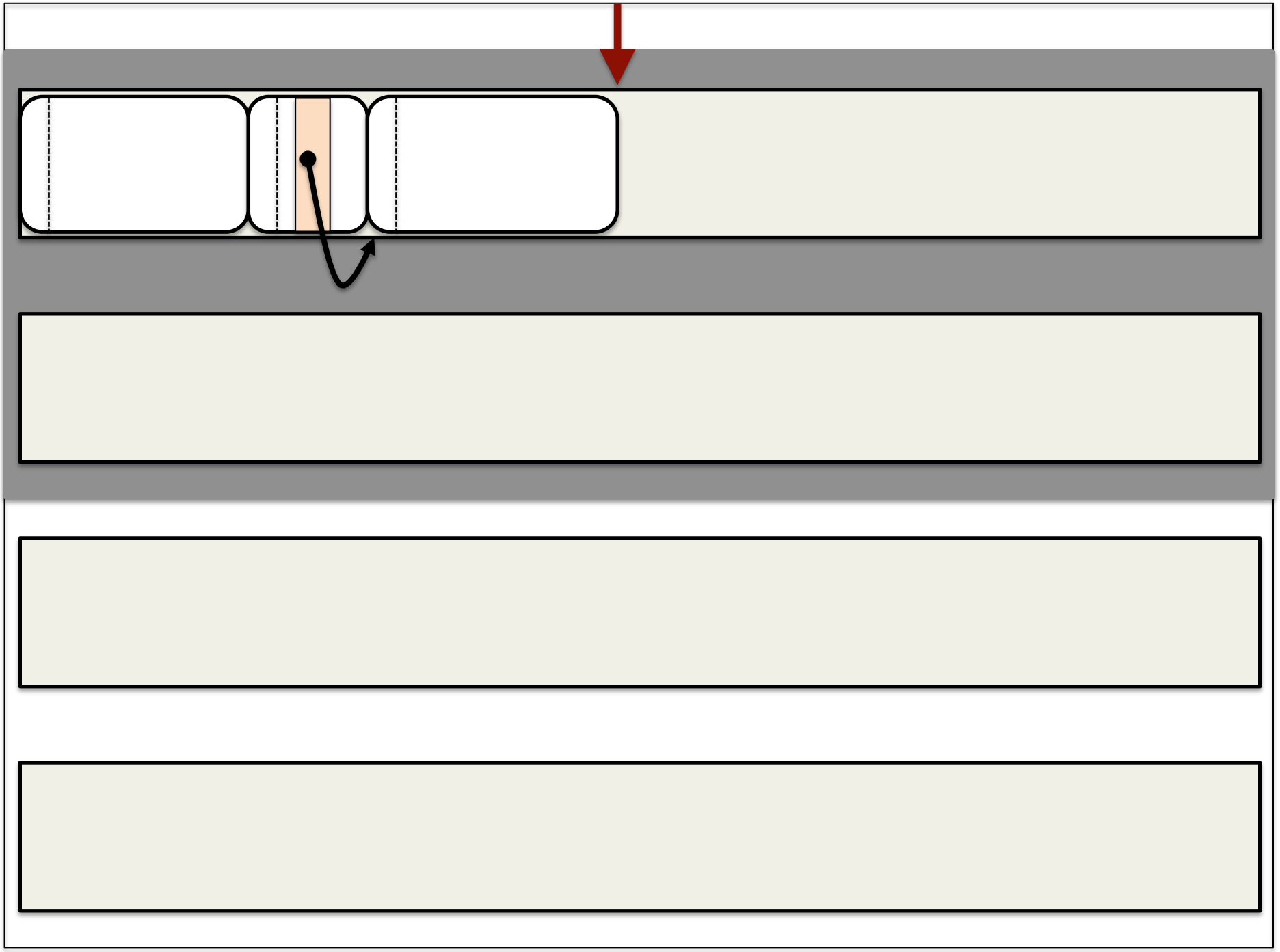
- Simplest form of copying collector
  - Divide the heap into two
  - Allocate to the first half
- Collector triggered when the heap is half full
  - Trace the heap to find reachable objects
  - Copy all reachable objects to the second half
- Allocation continues in the second semi-space

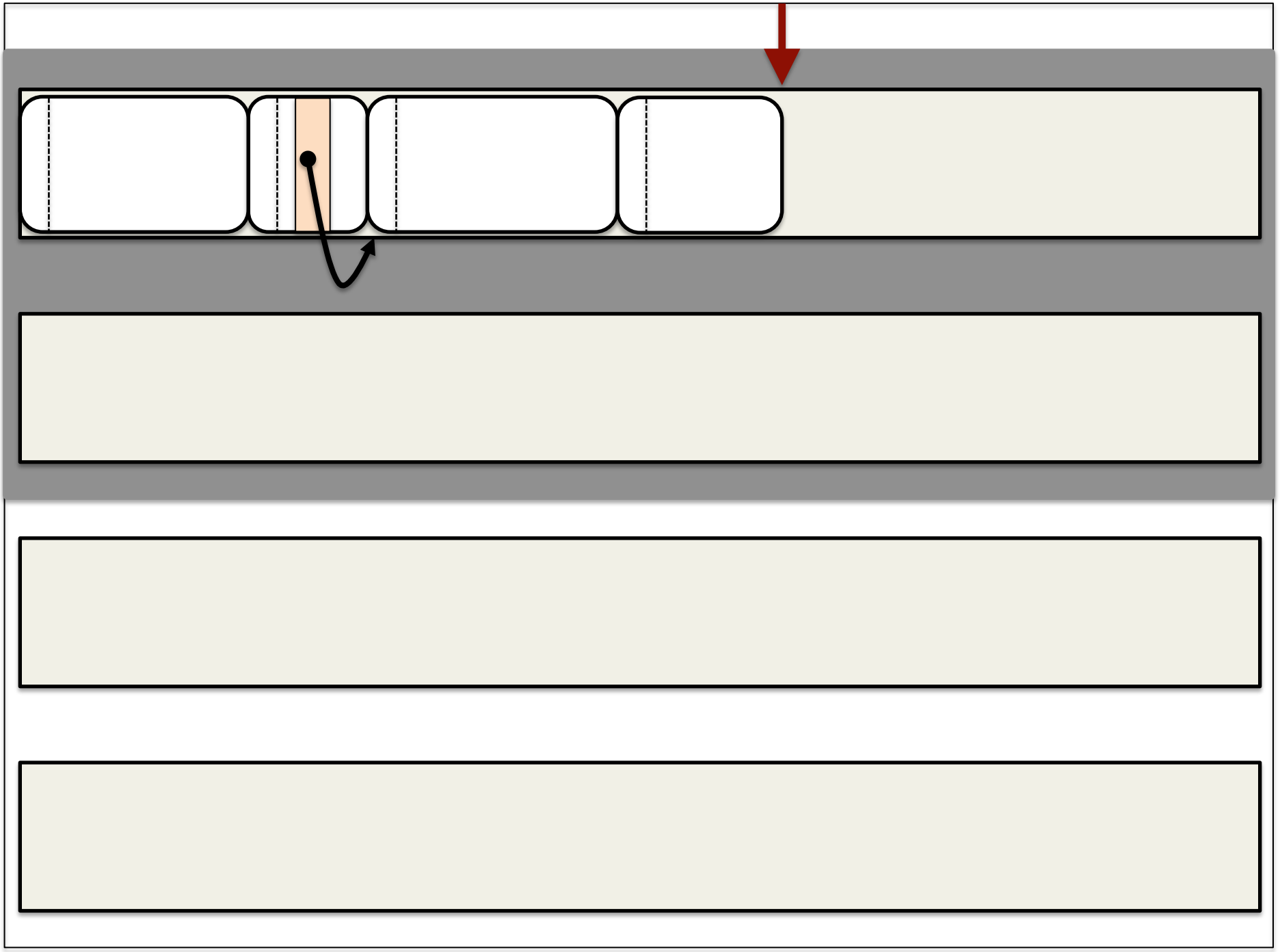


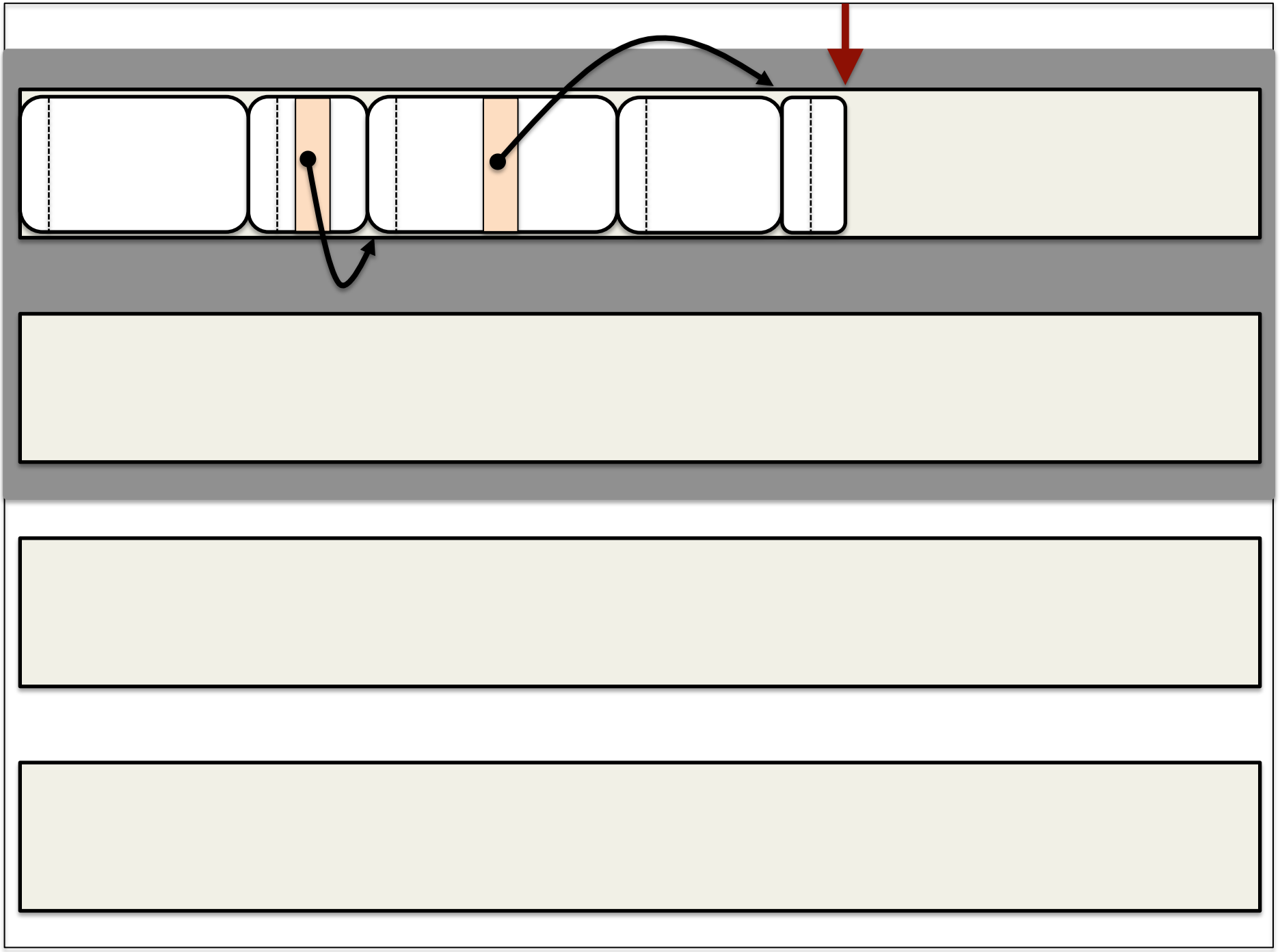


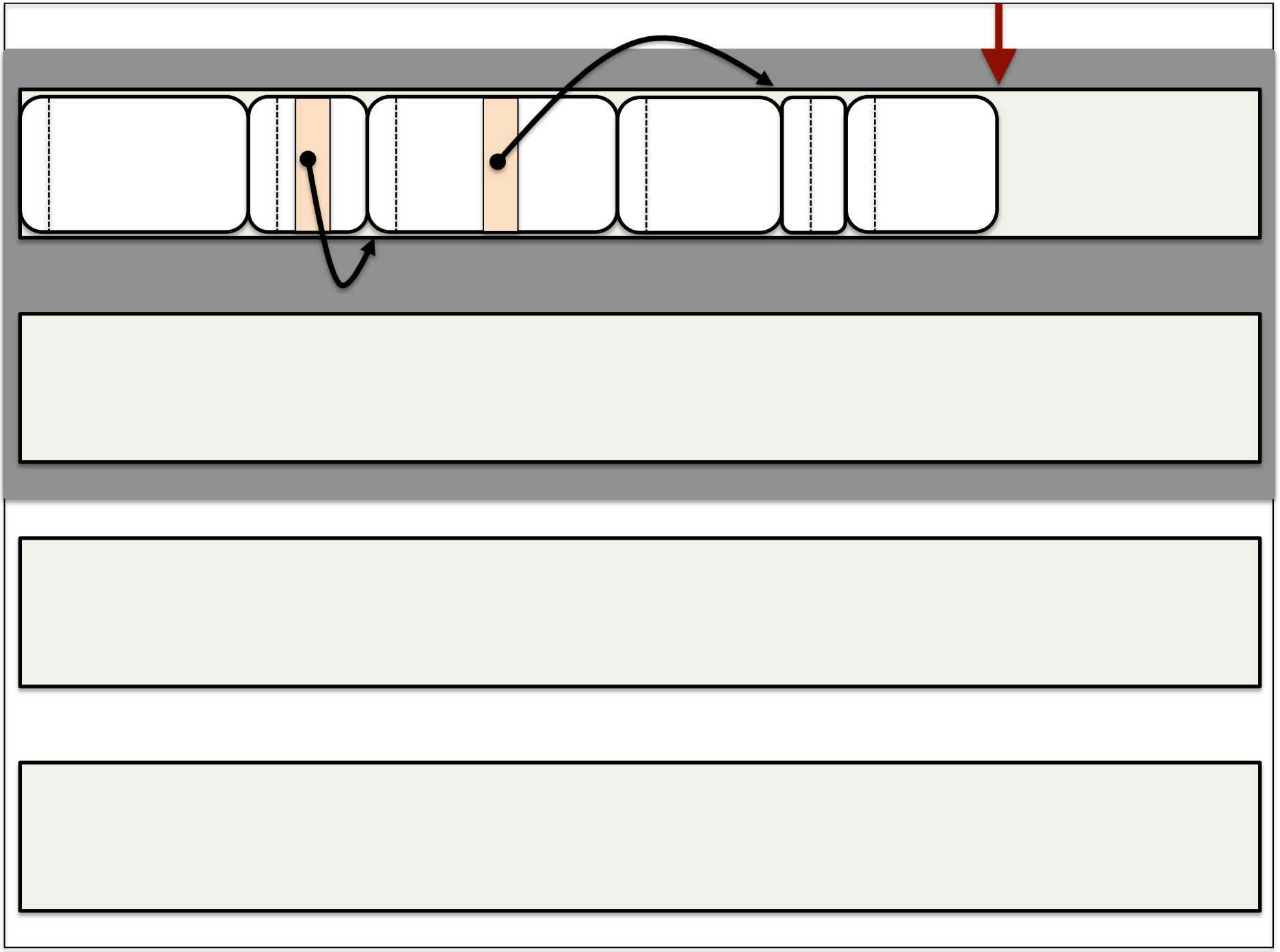


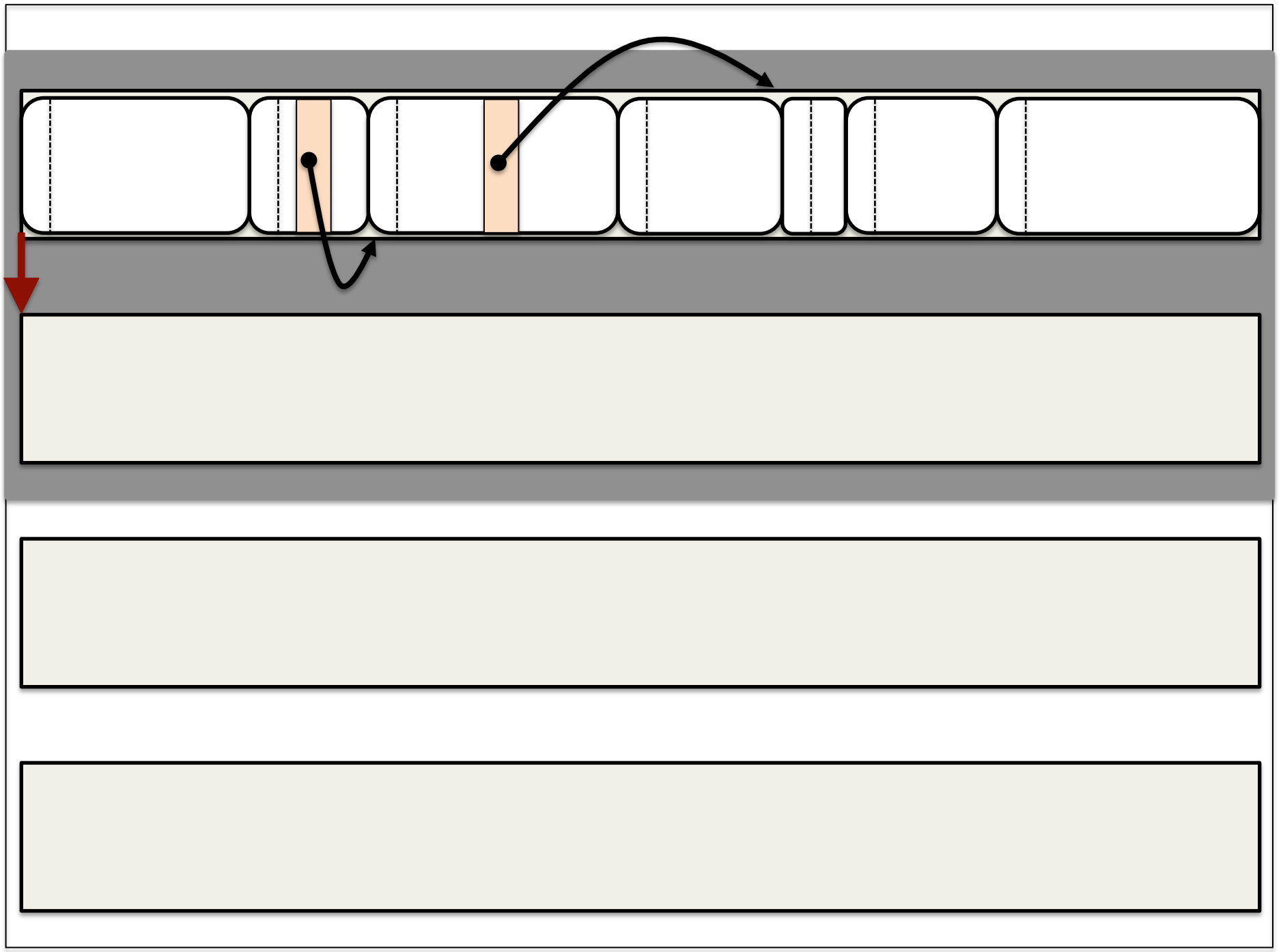










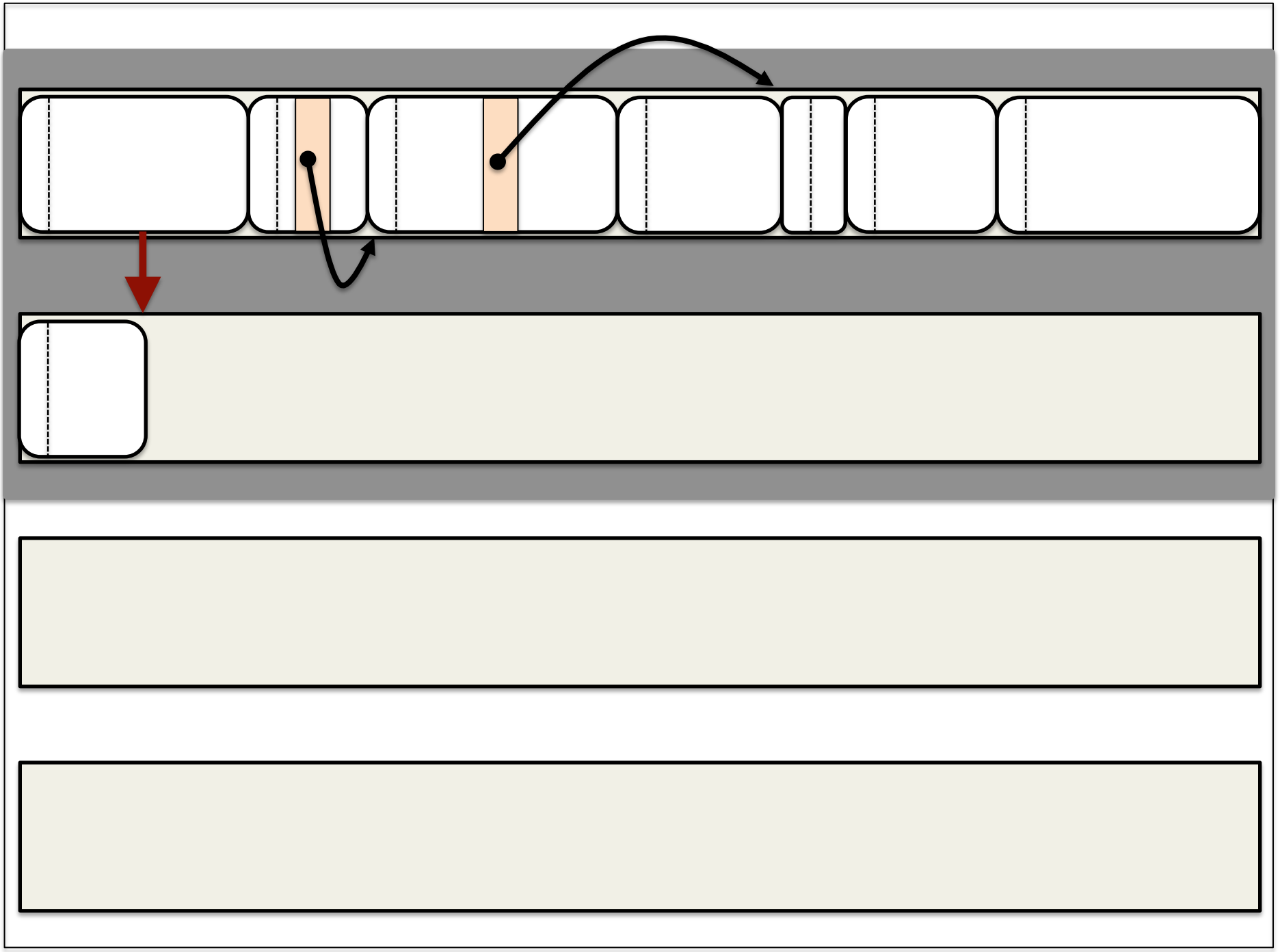


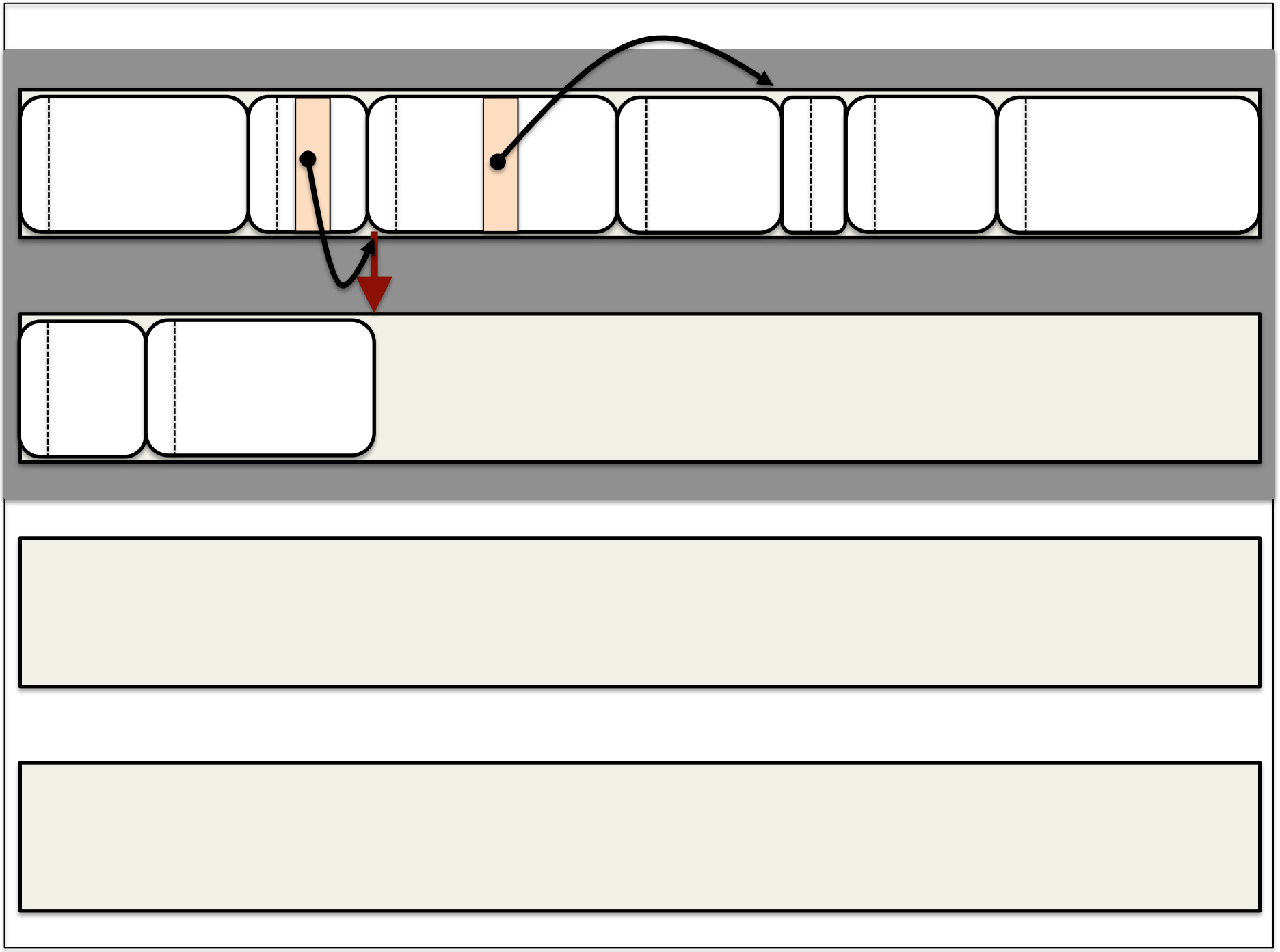
Bump Pointer Allocation

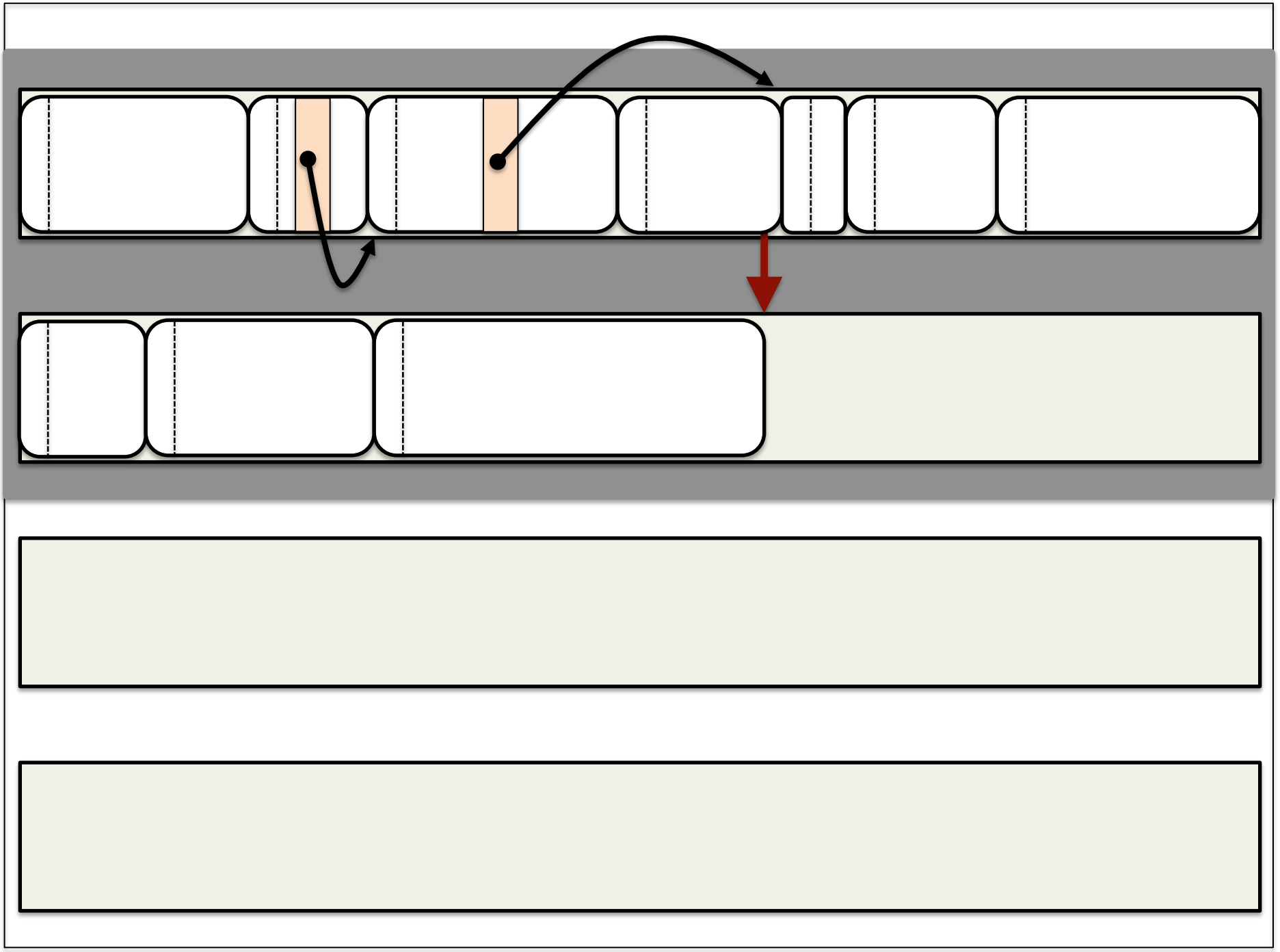
- Each object allocated directly after the previous
  - Only works if there is no fragmentation
- Straightforward algorithm
  - No heuristics as in previous allocators
- Allocation is very cheap

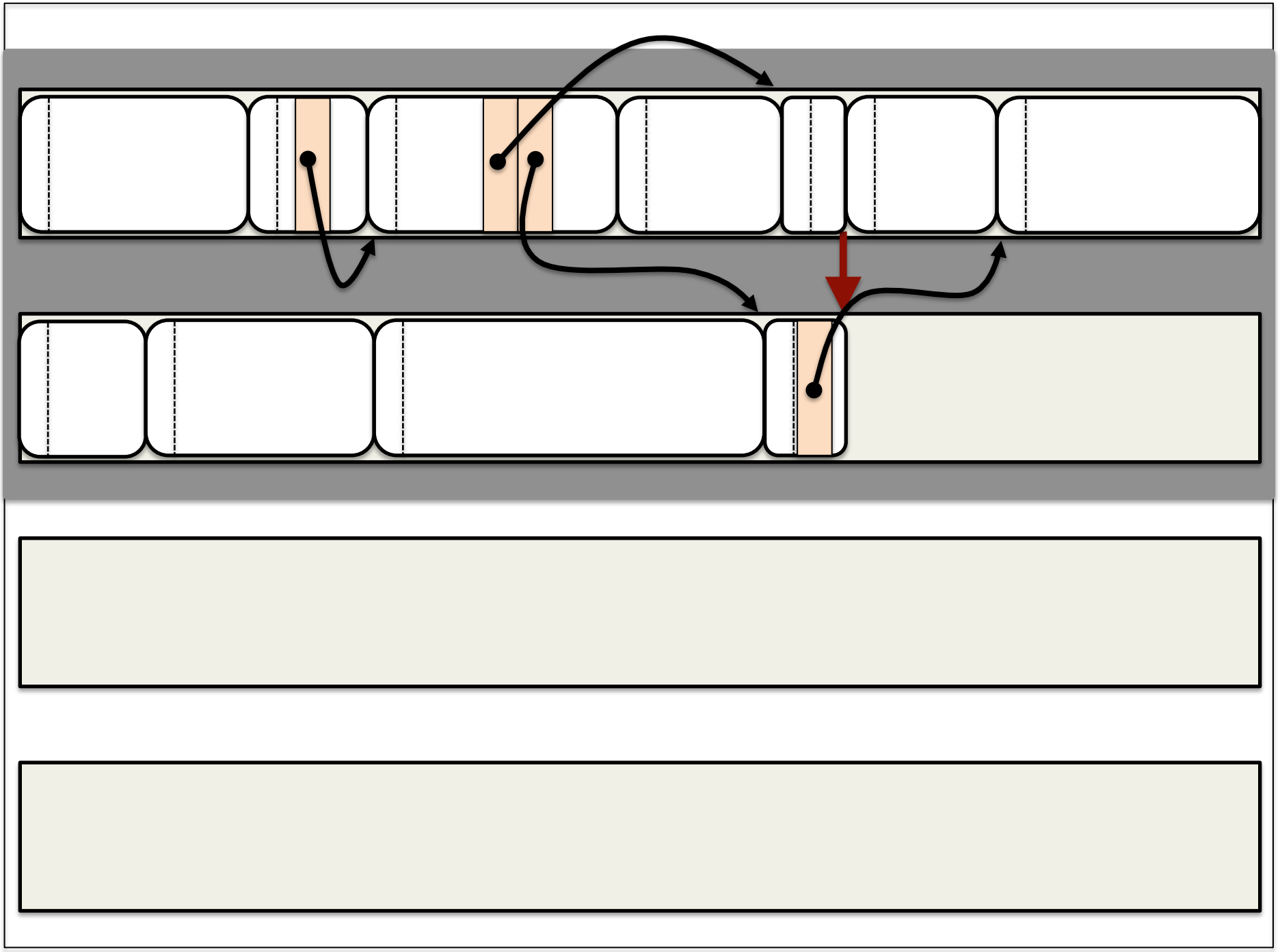
Bump Pointer Characteristics

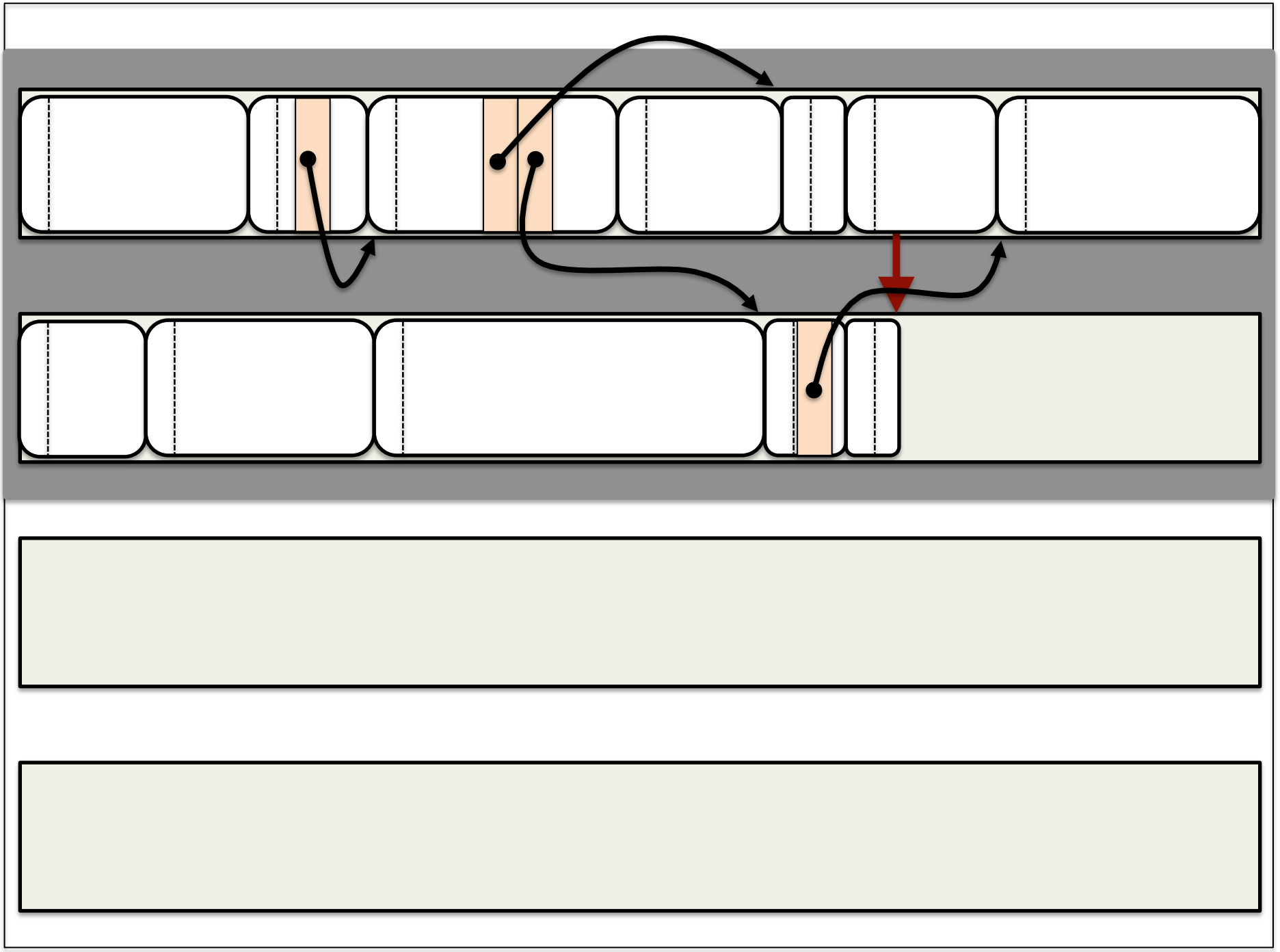
- Fast path
  - Taken when there is enough memory available
  - Allocation is an increment and comparison
  - Easily inlined and optimized
- Slow path
  - Taken when comparison fails
  - Get new memory region from heap
  - Trigger garbage collection if none available

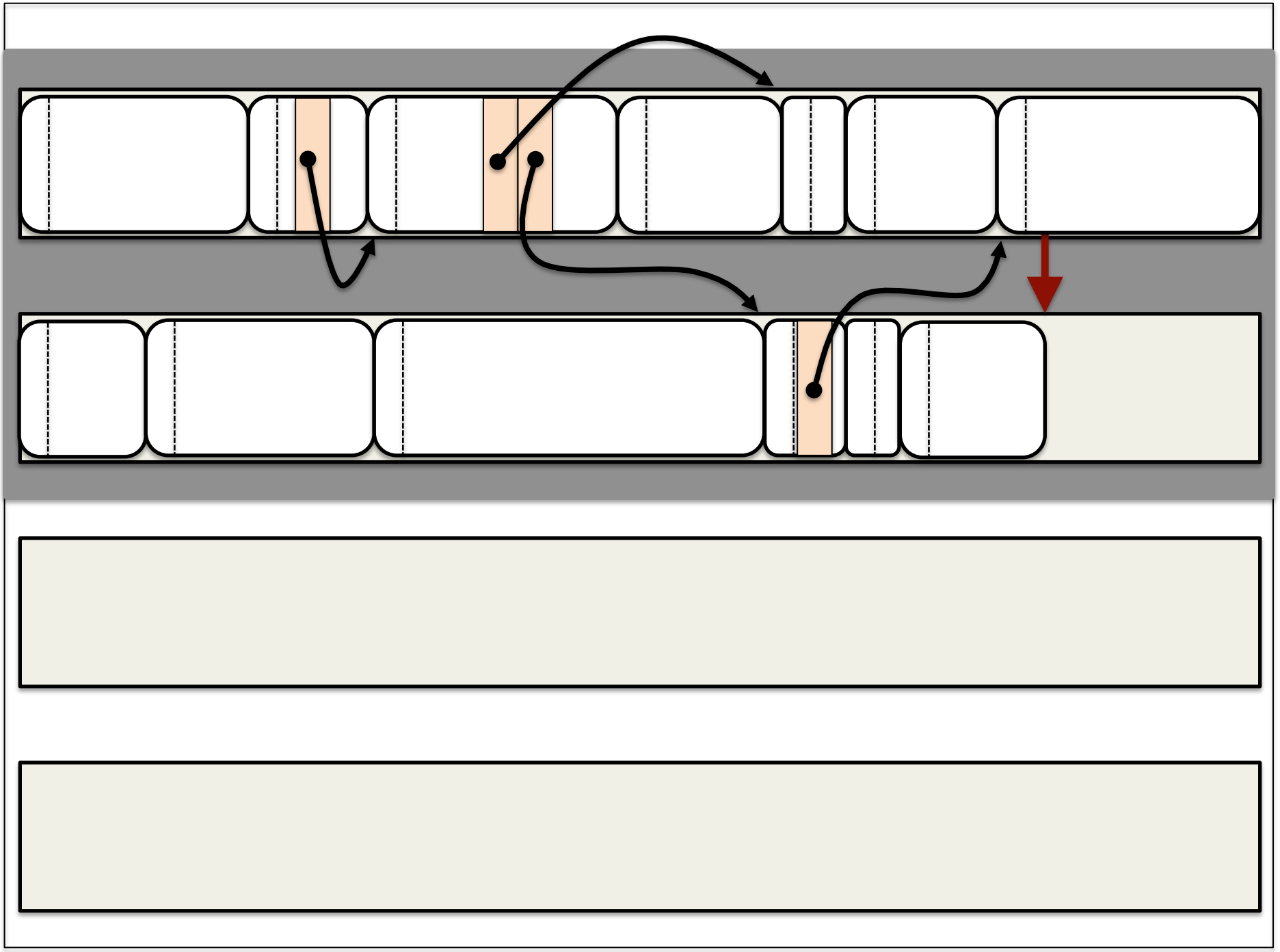


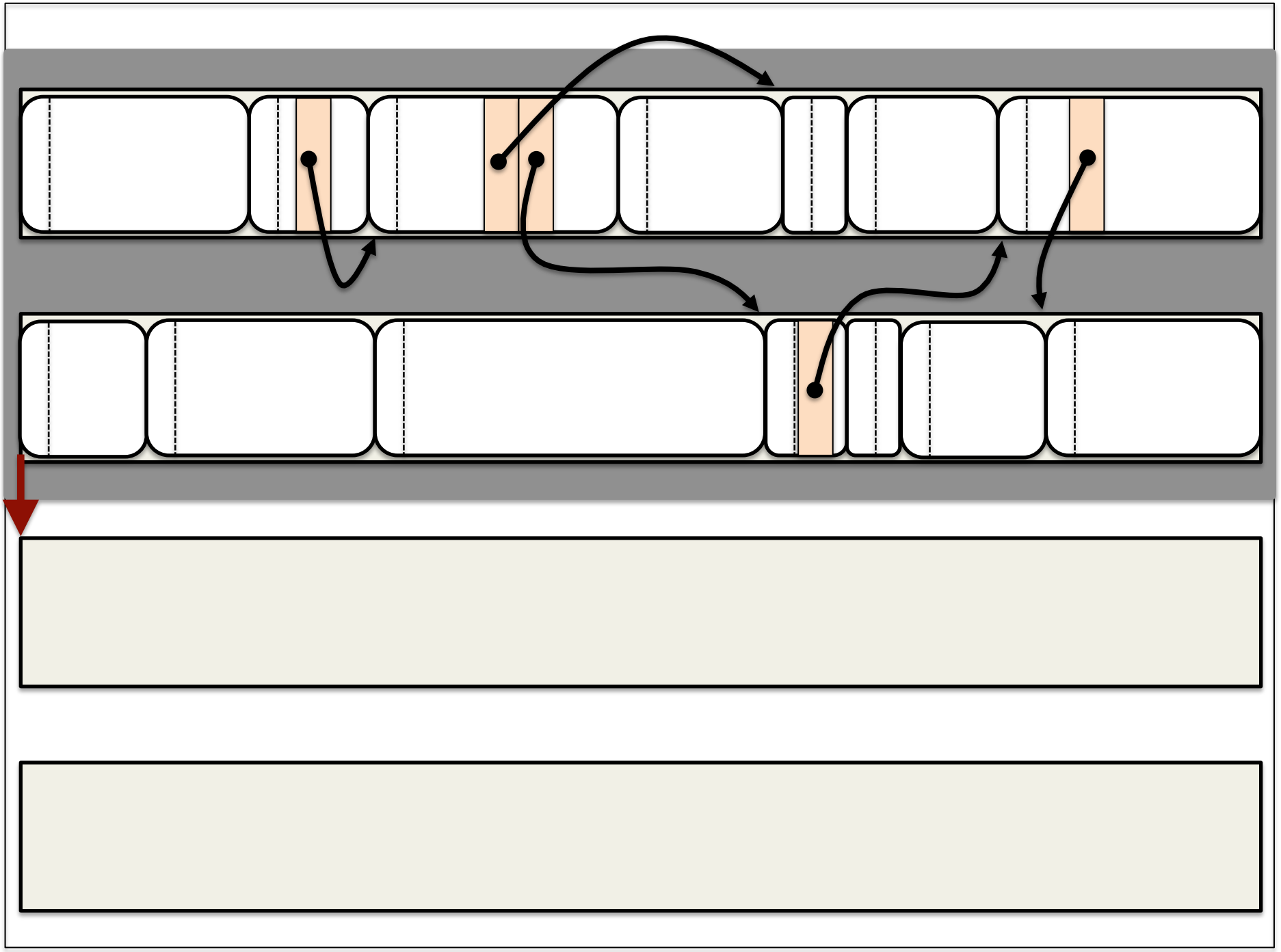






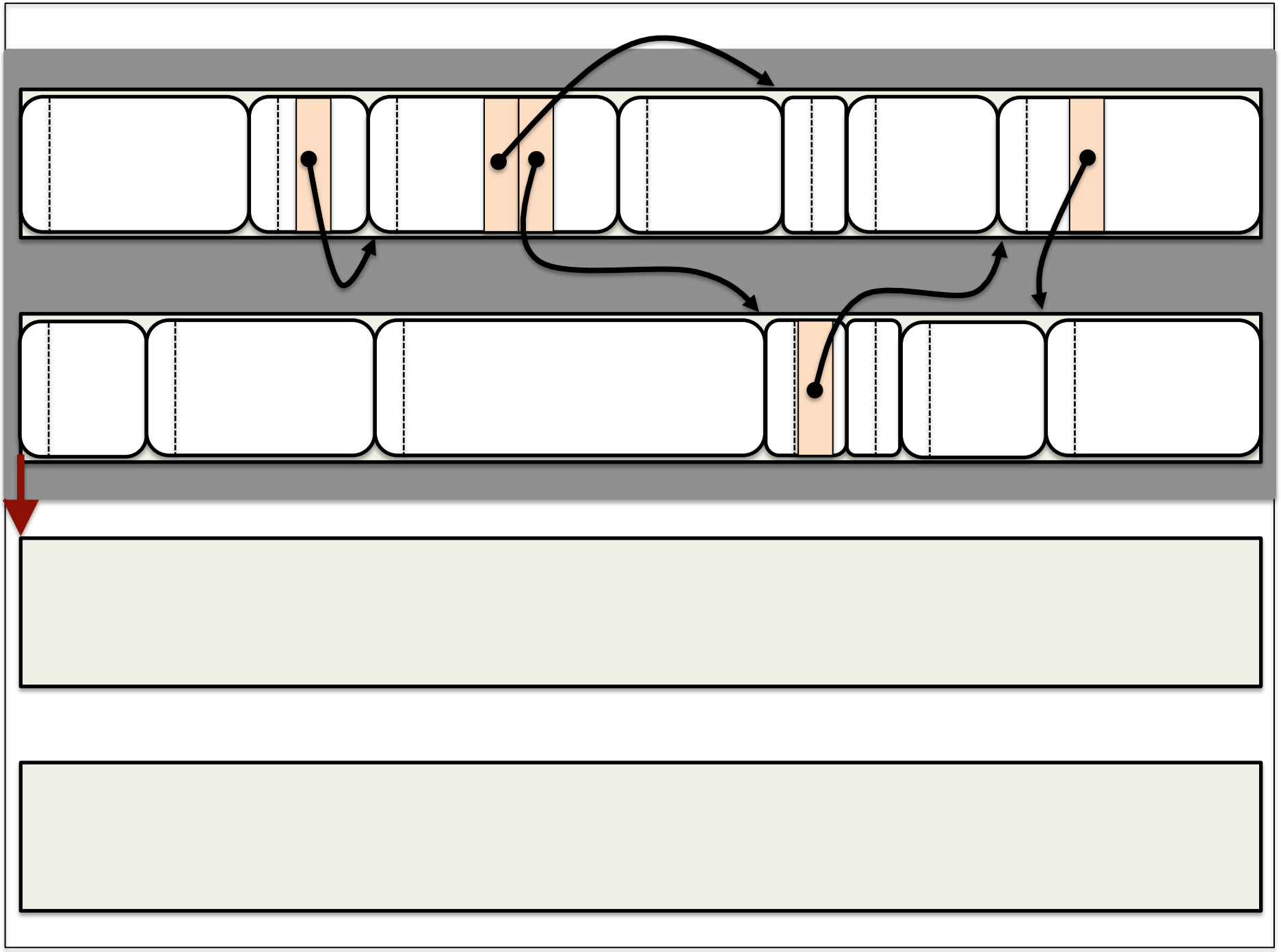


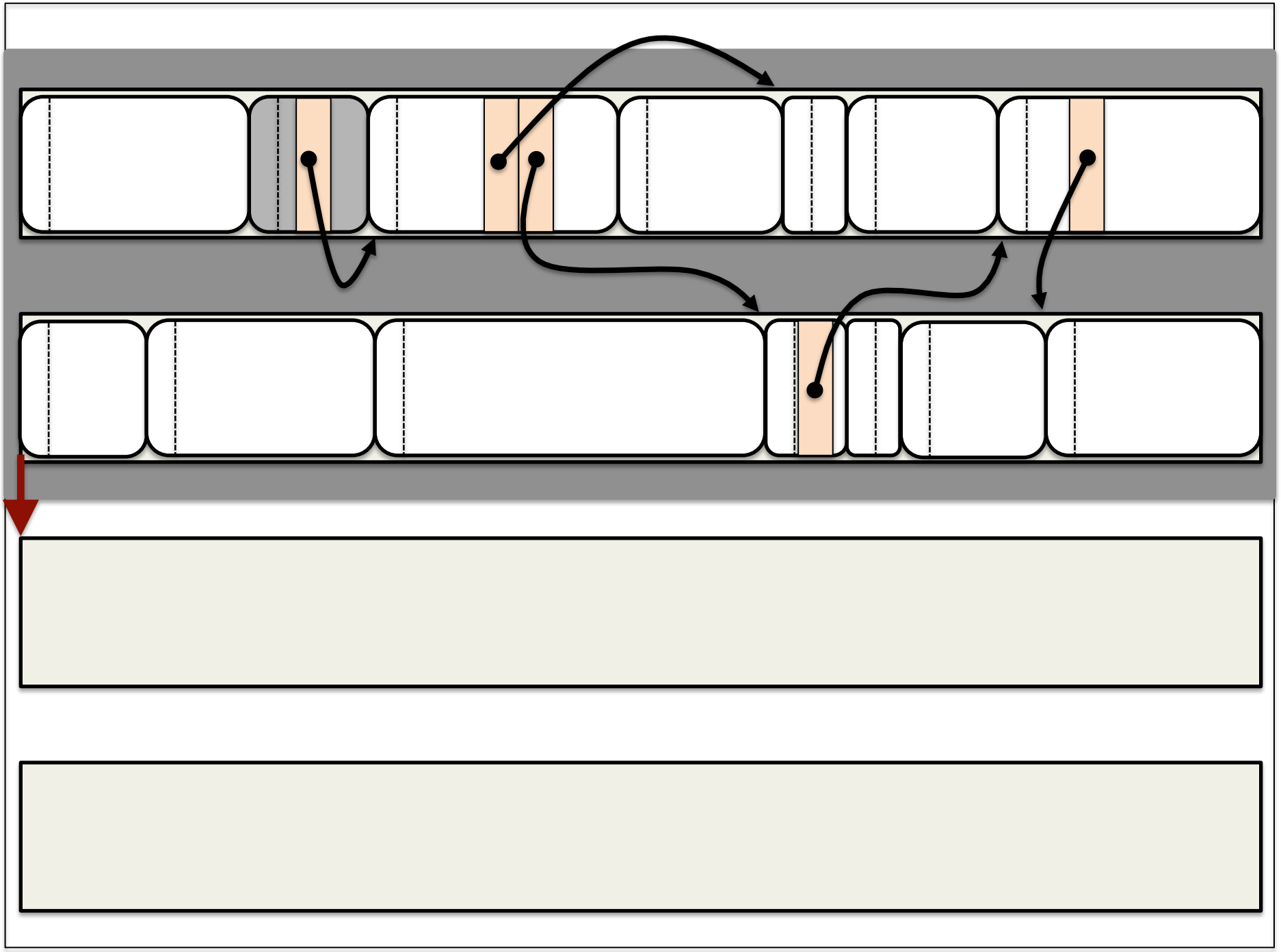


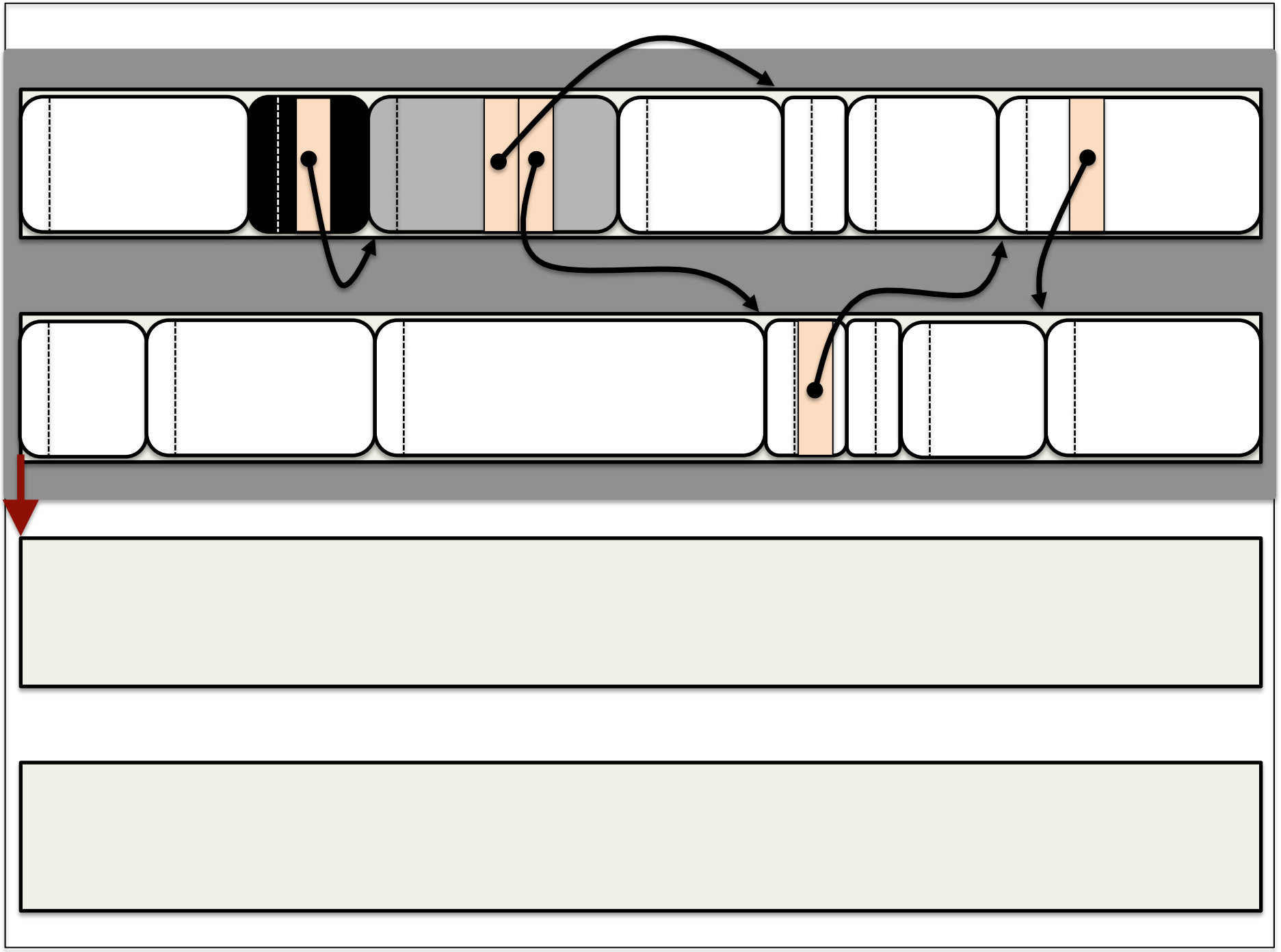


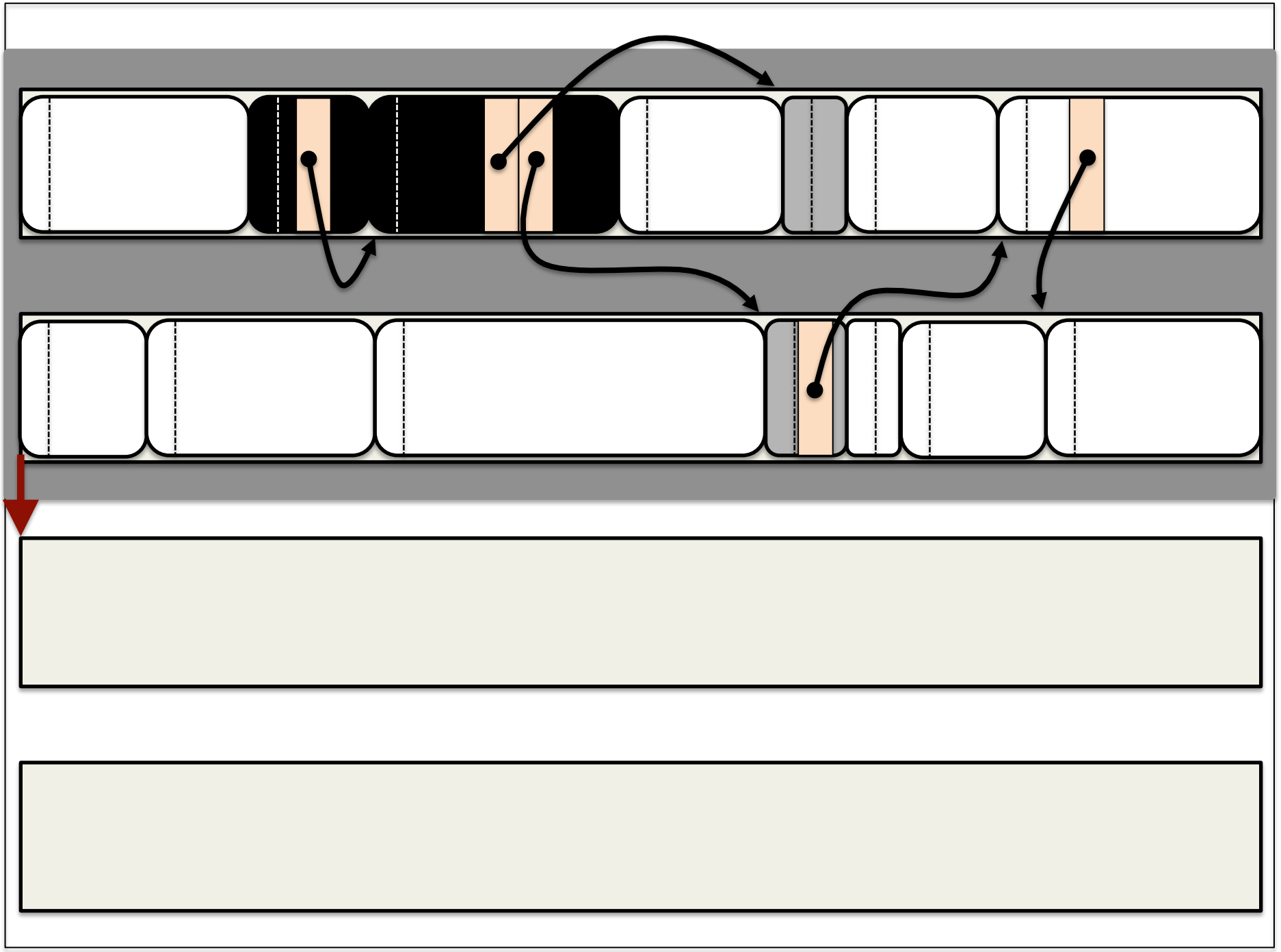
Tracing

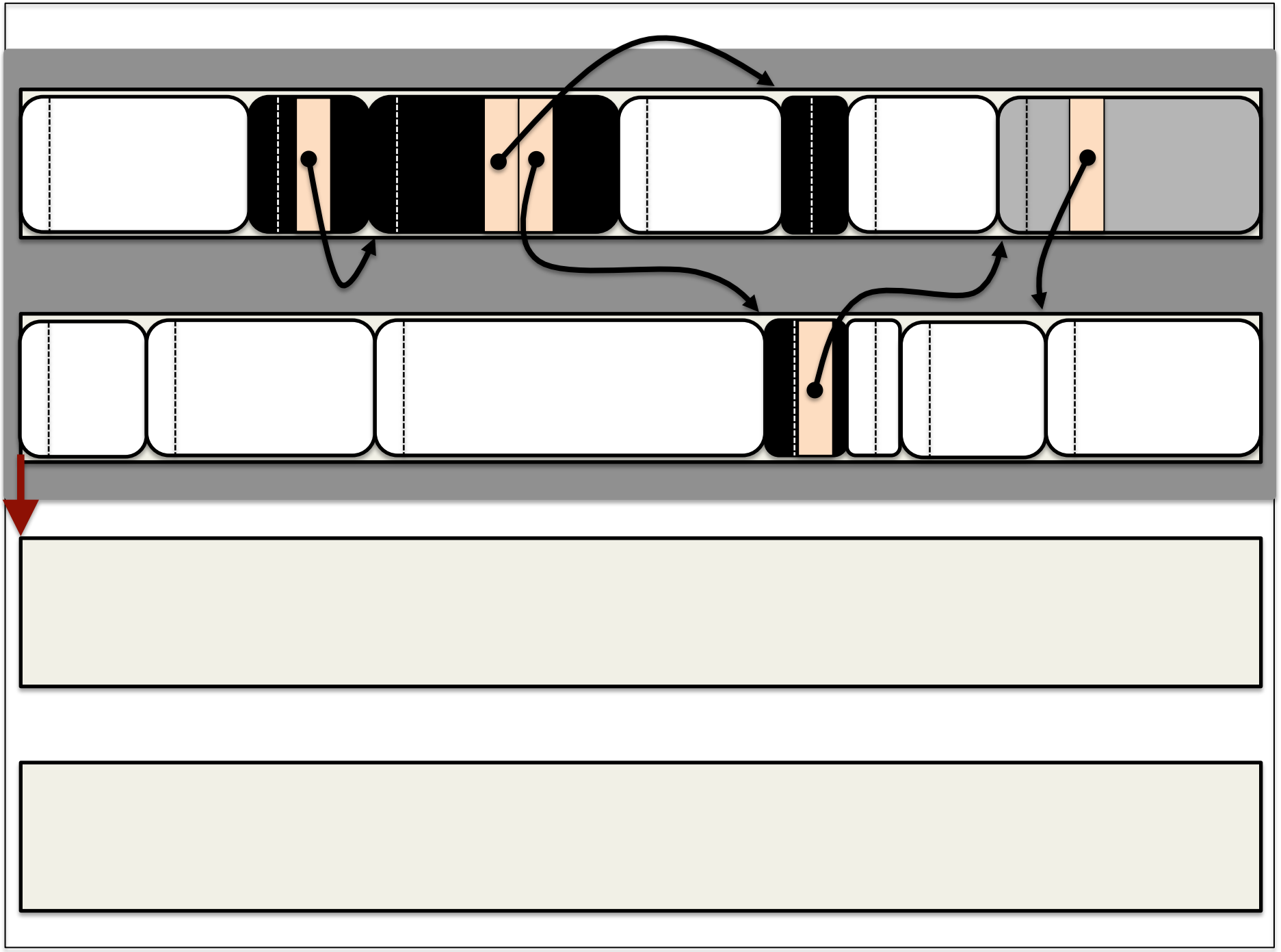
- Trace process the same as in mark and sweep
  - We'll see a more specialized algorithm later
- Start with the system roots
- Use tricolor algorithm to find all live objects
- Result is that reachable objects are black

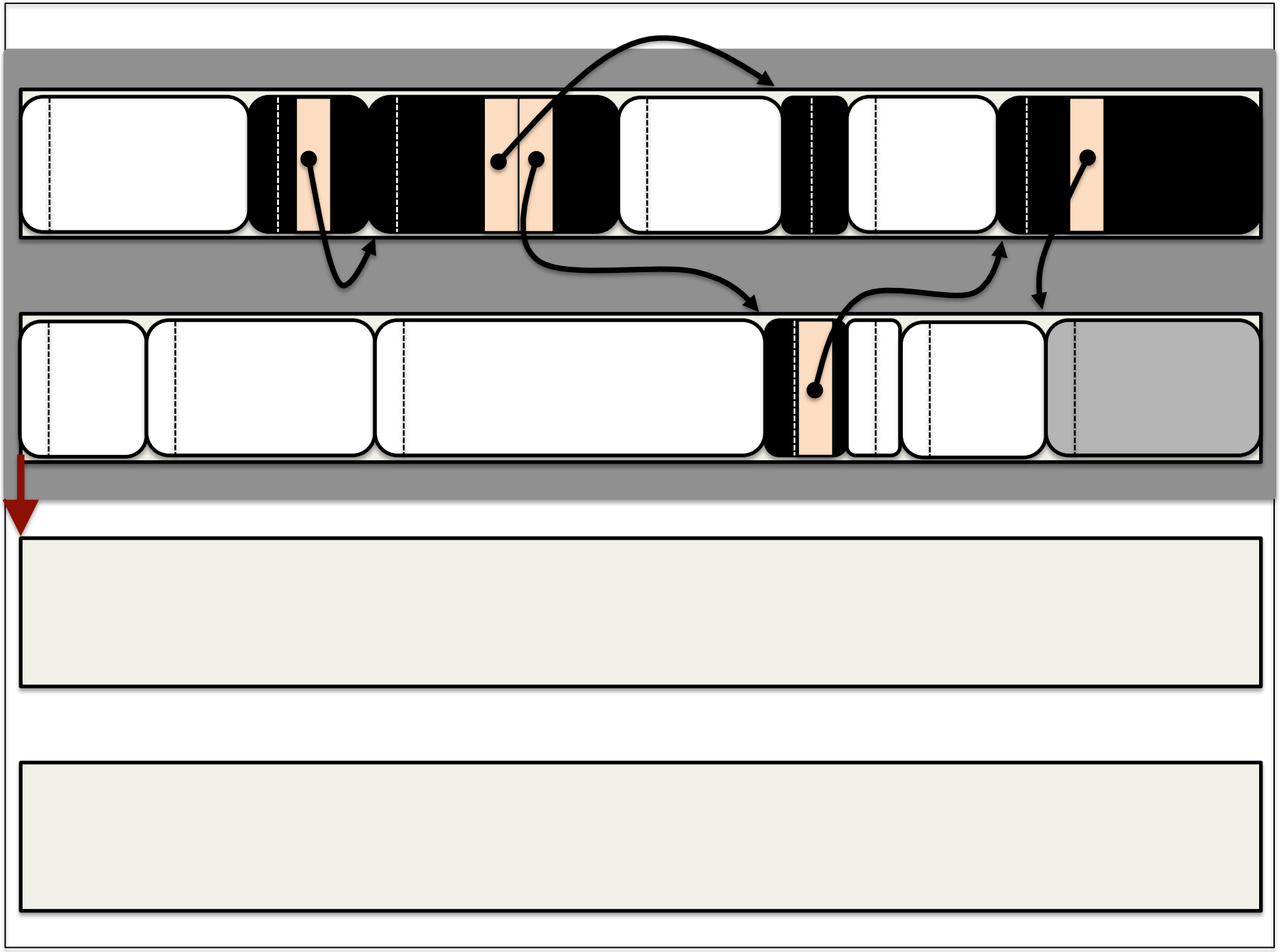


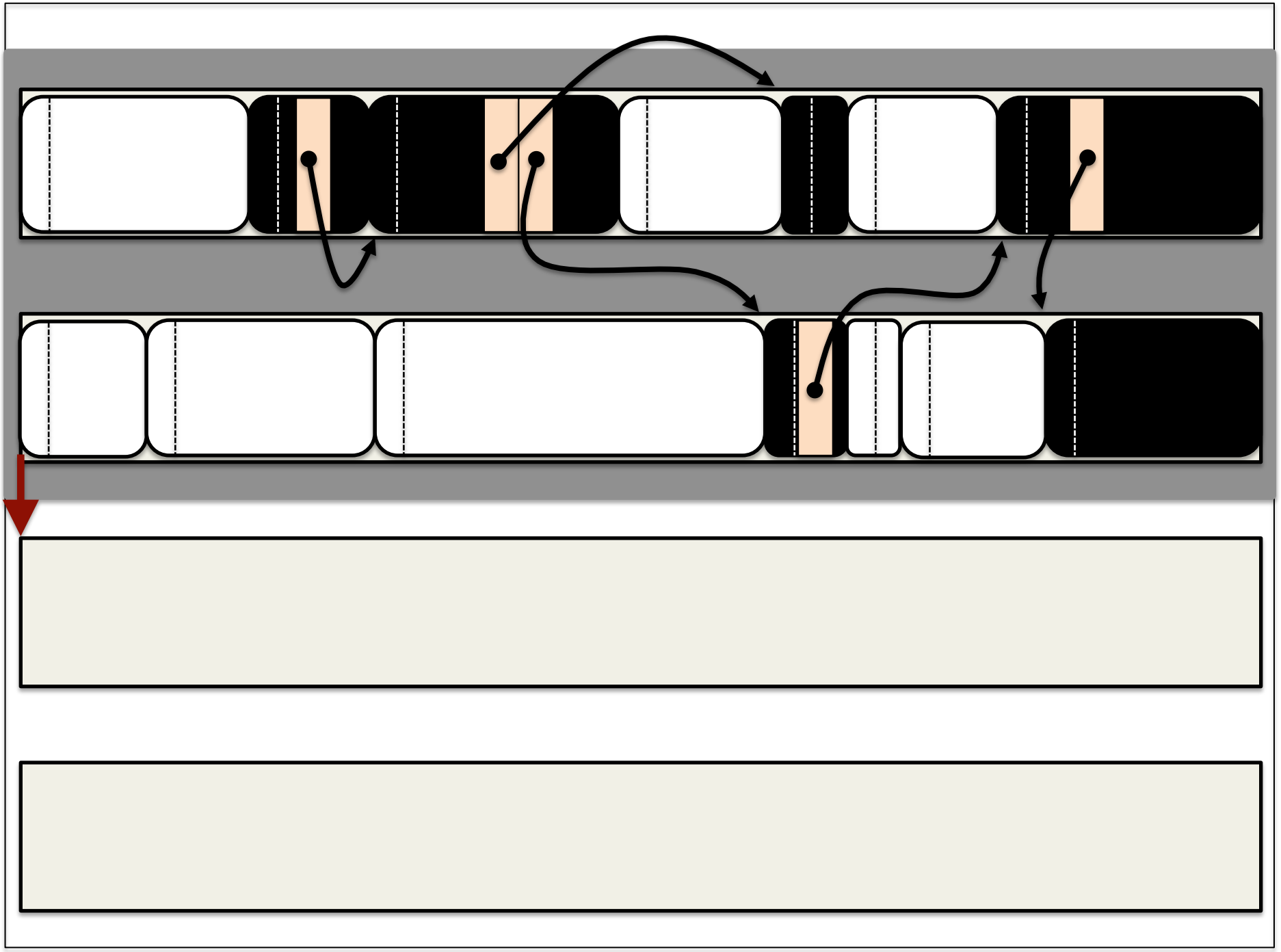






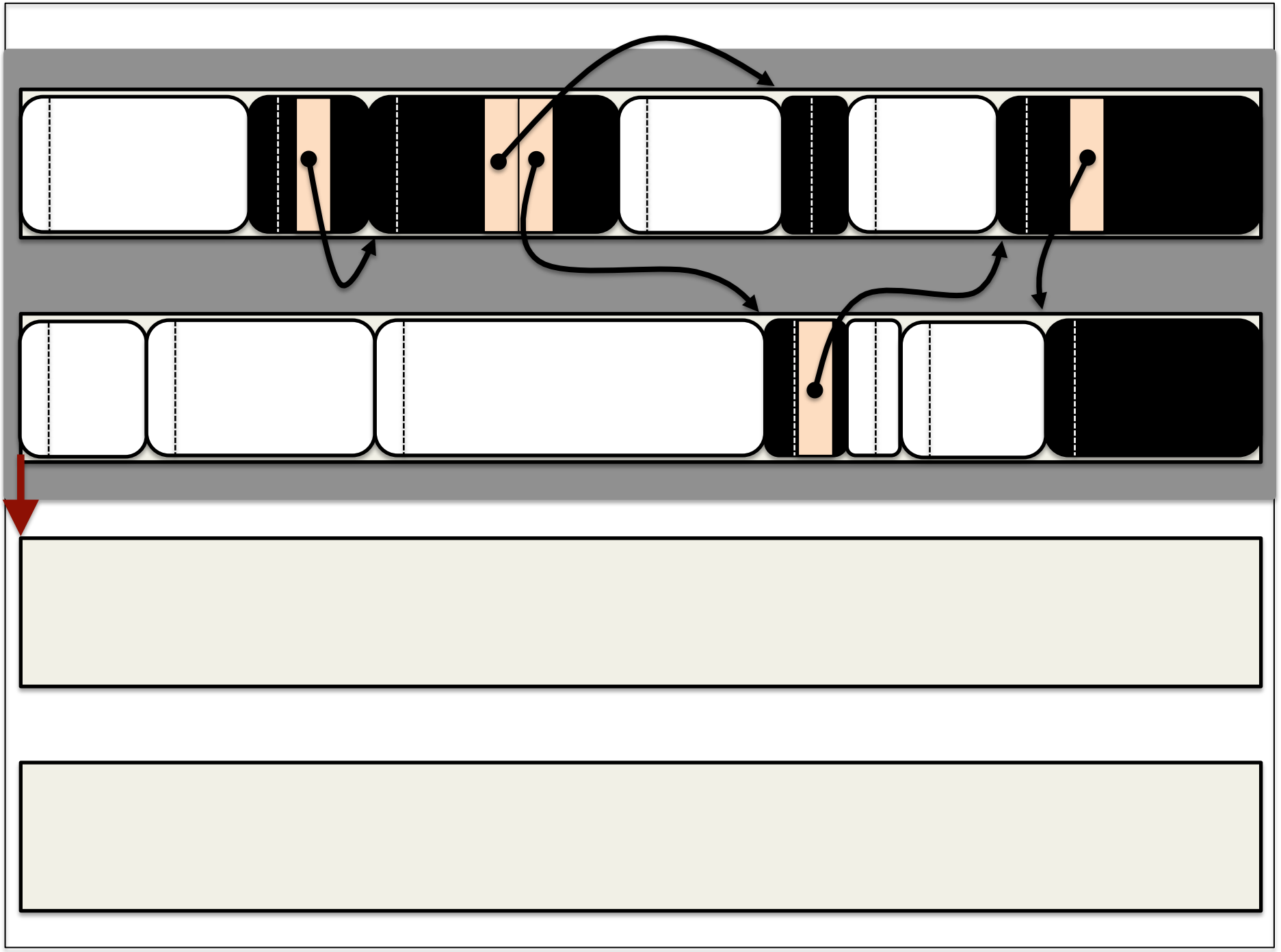


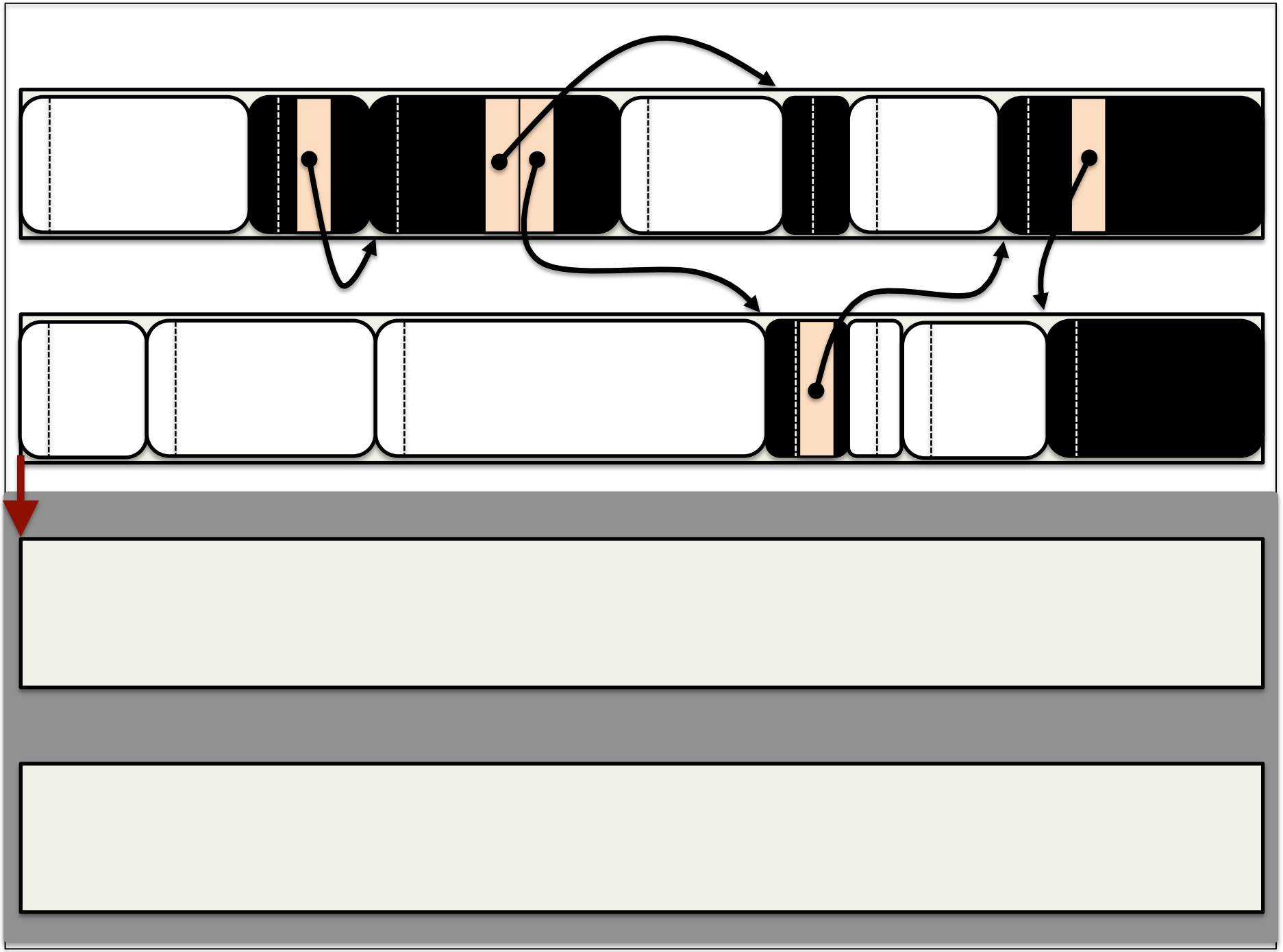


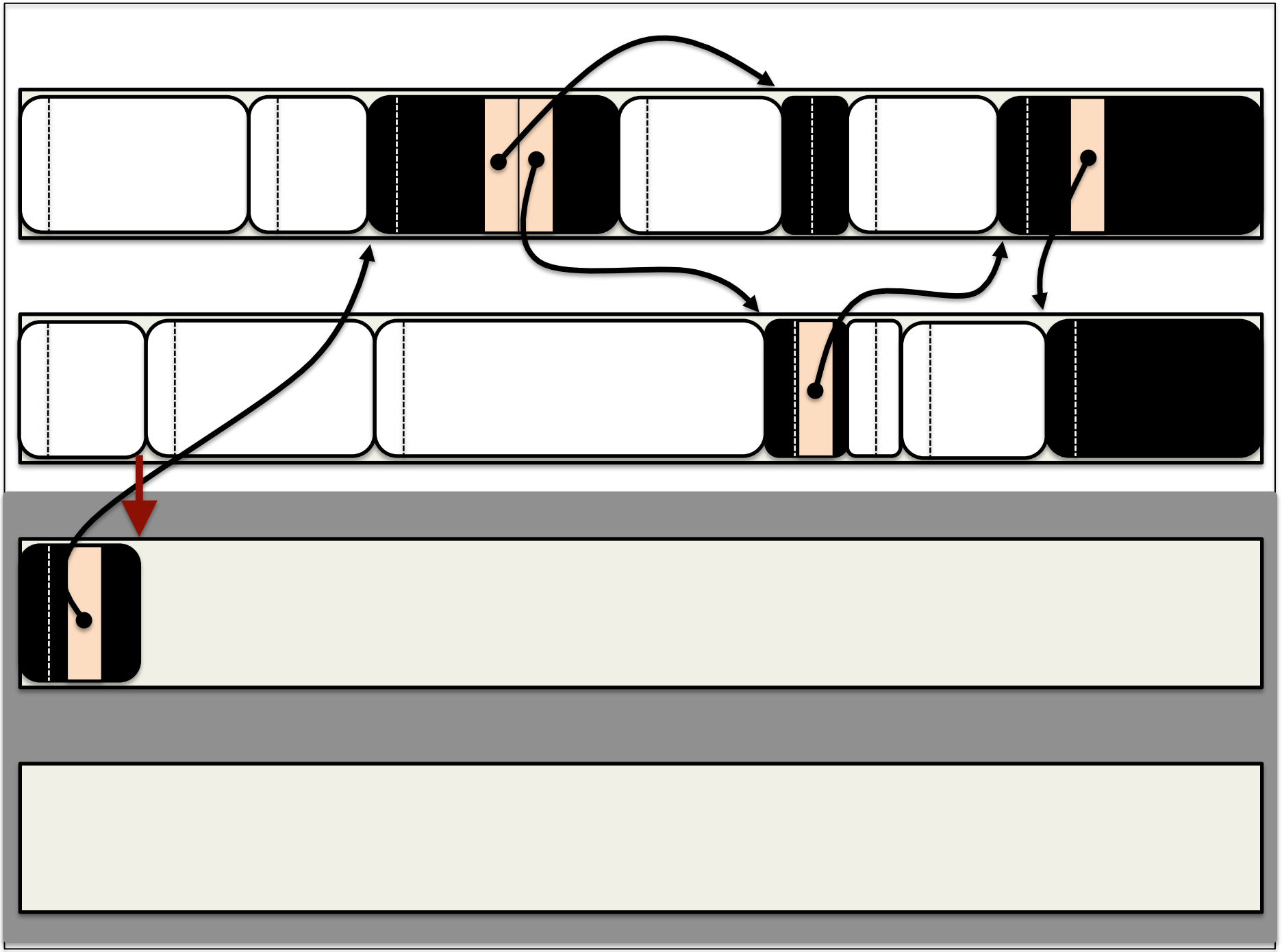


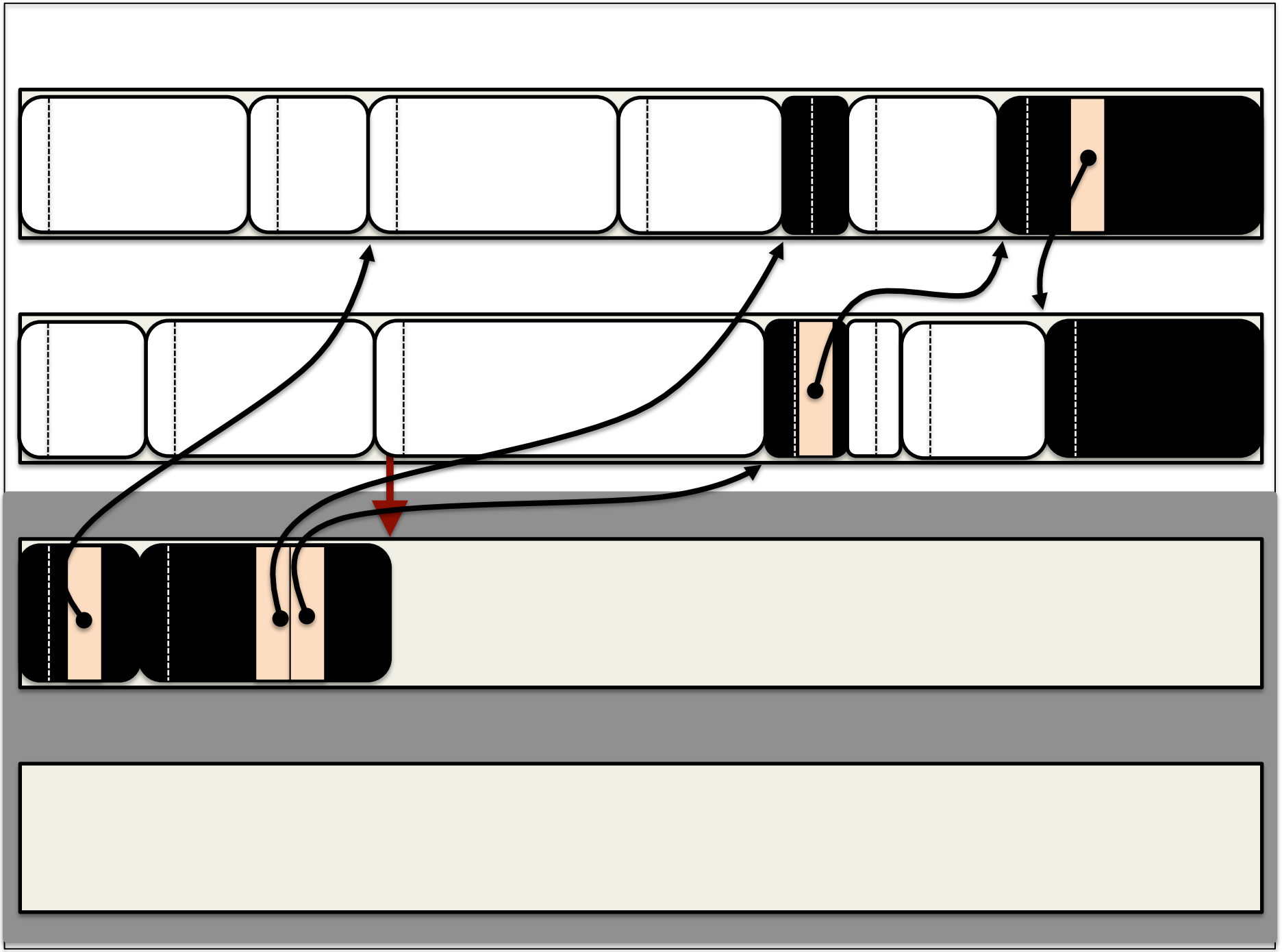
Old and New Space

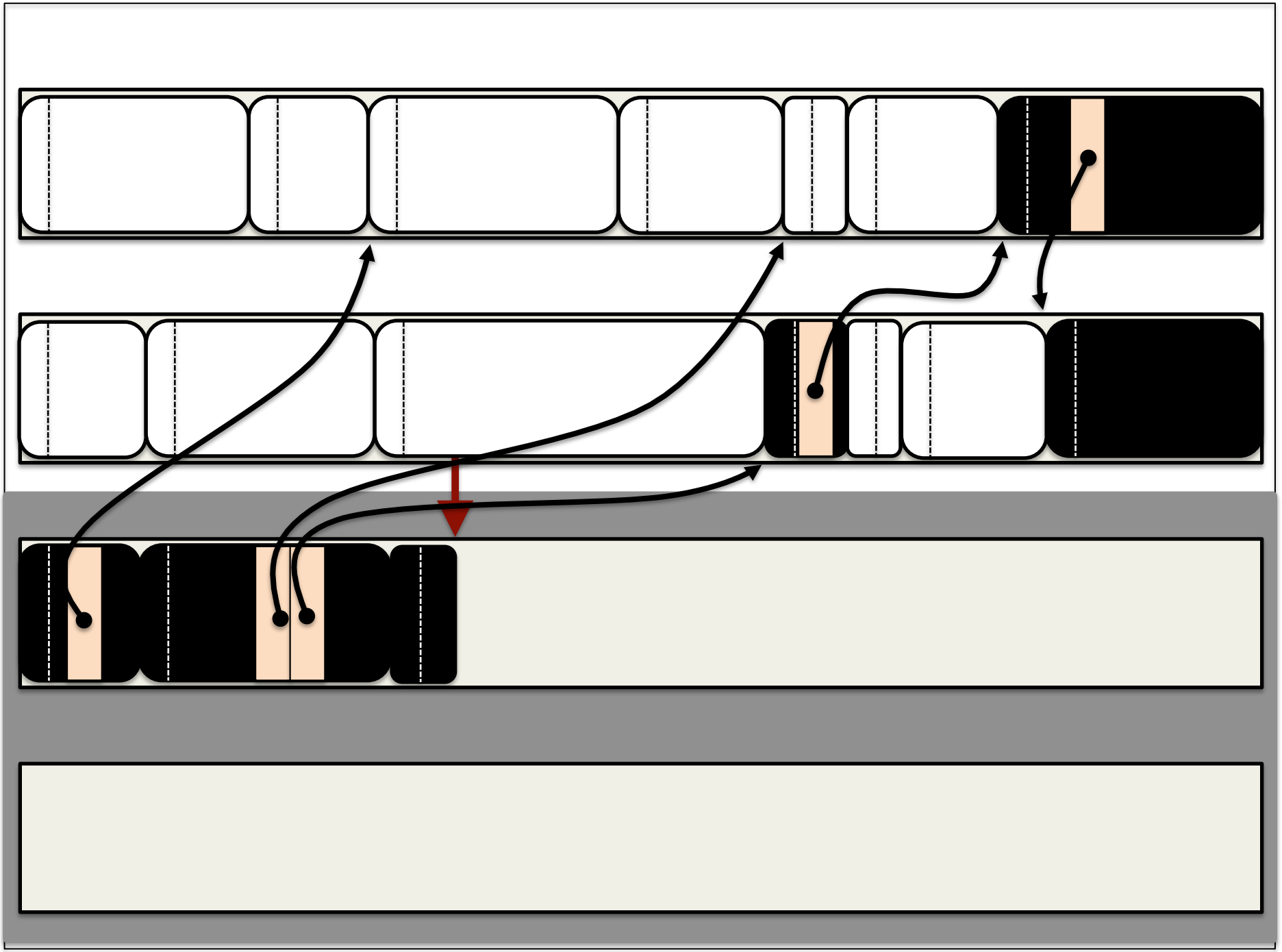
- Copy all live objects from one space to the other
- Spaces referred to by different names
  - Old space or From space
  - New space or To space
- Which one is used depends on convention
  - Both mean the same thing

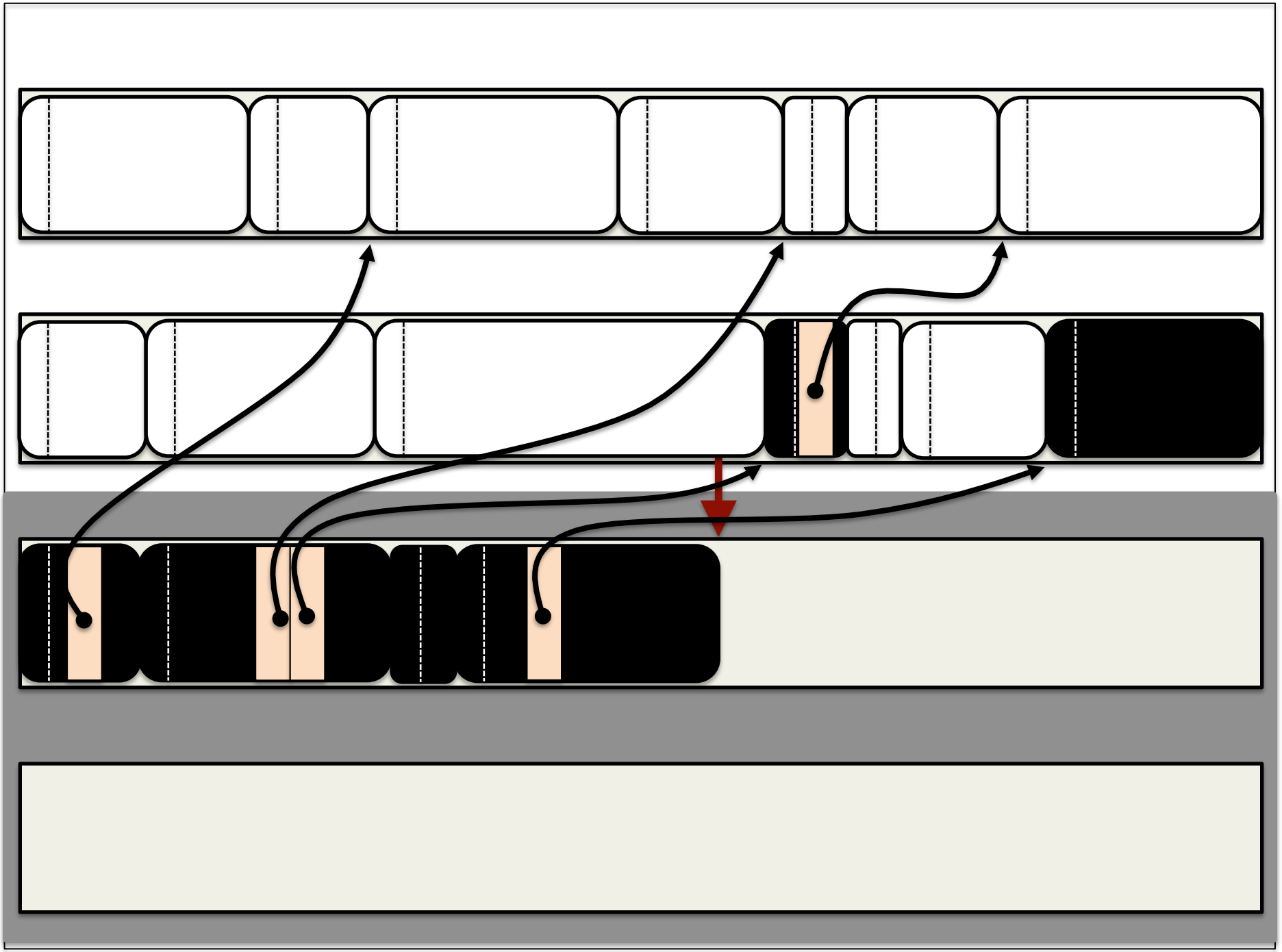


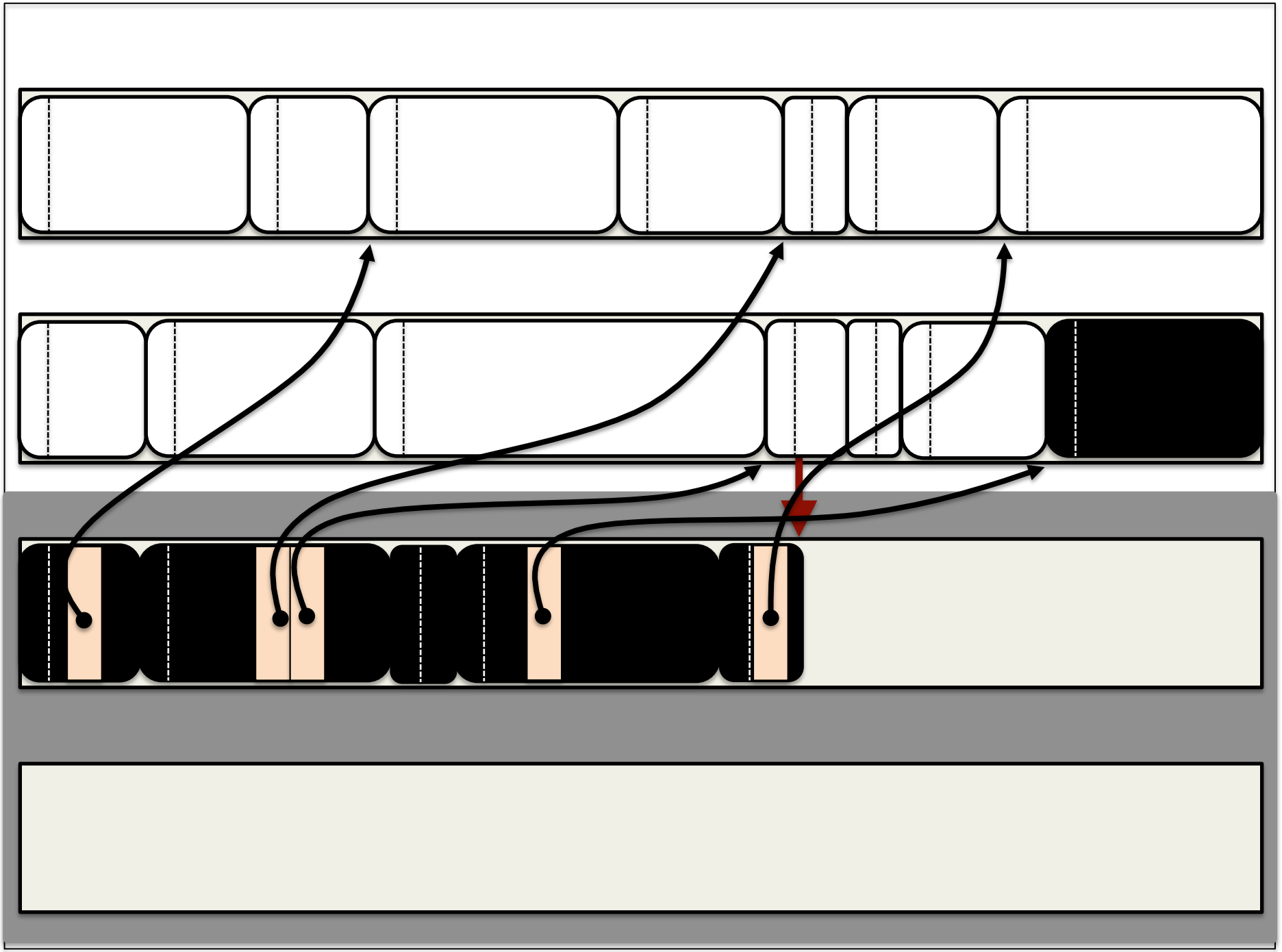


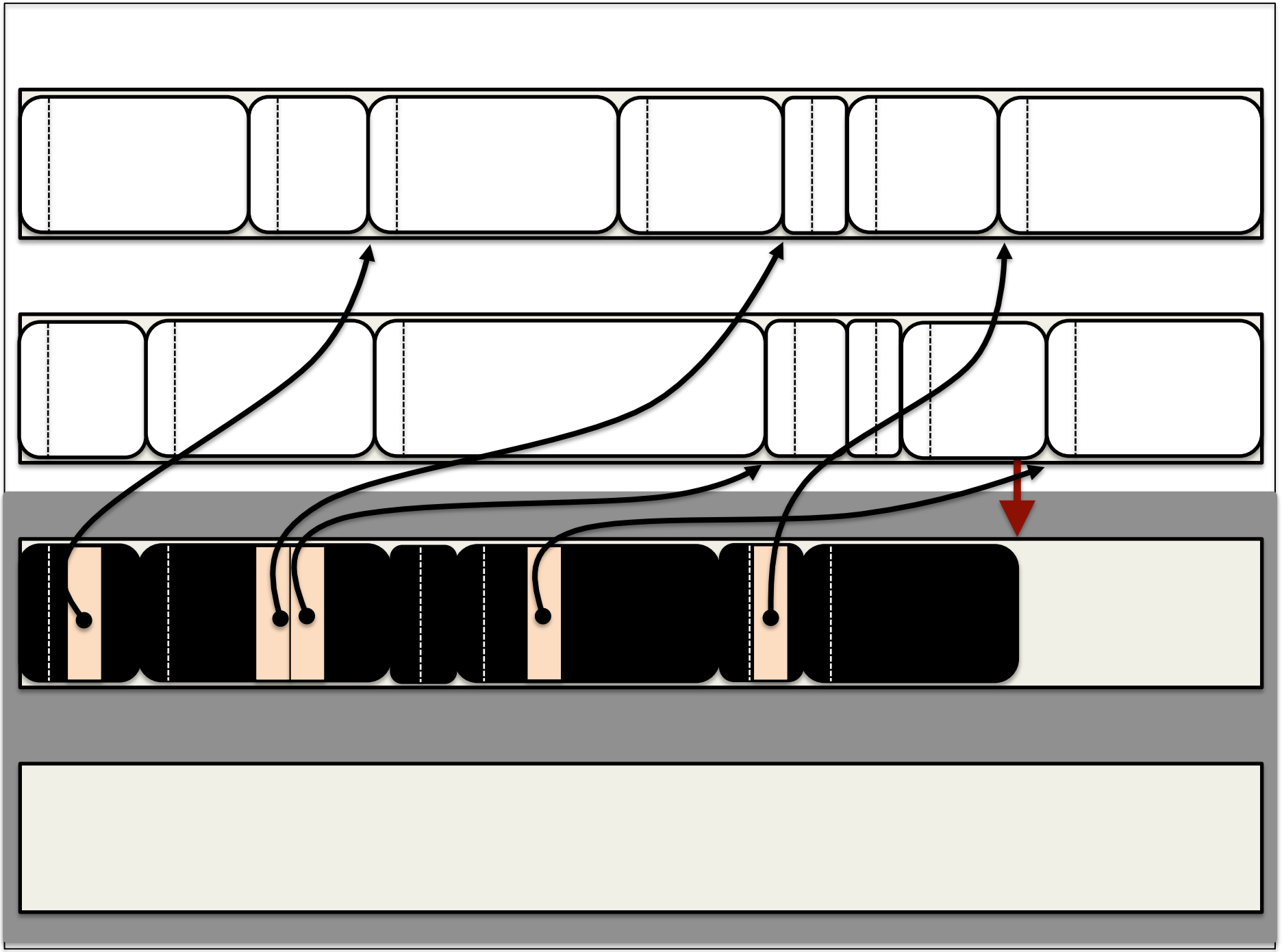












Updating References

- Copy into new space is bit-for-bit
  - Generally use memcpy
  - Pointers are just patterns of bits
- Copied object refers to old space
  - But old space is garbage
  - Need to have one canonical version of the object
- We have to update references when we copy

Handles

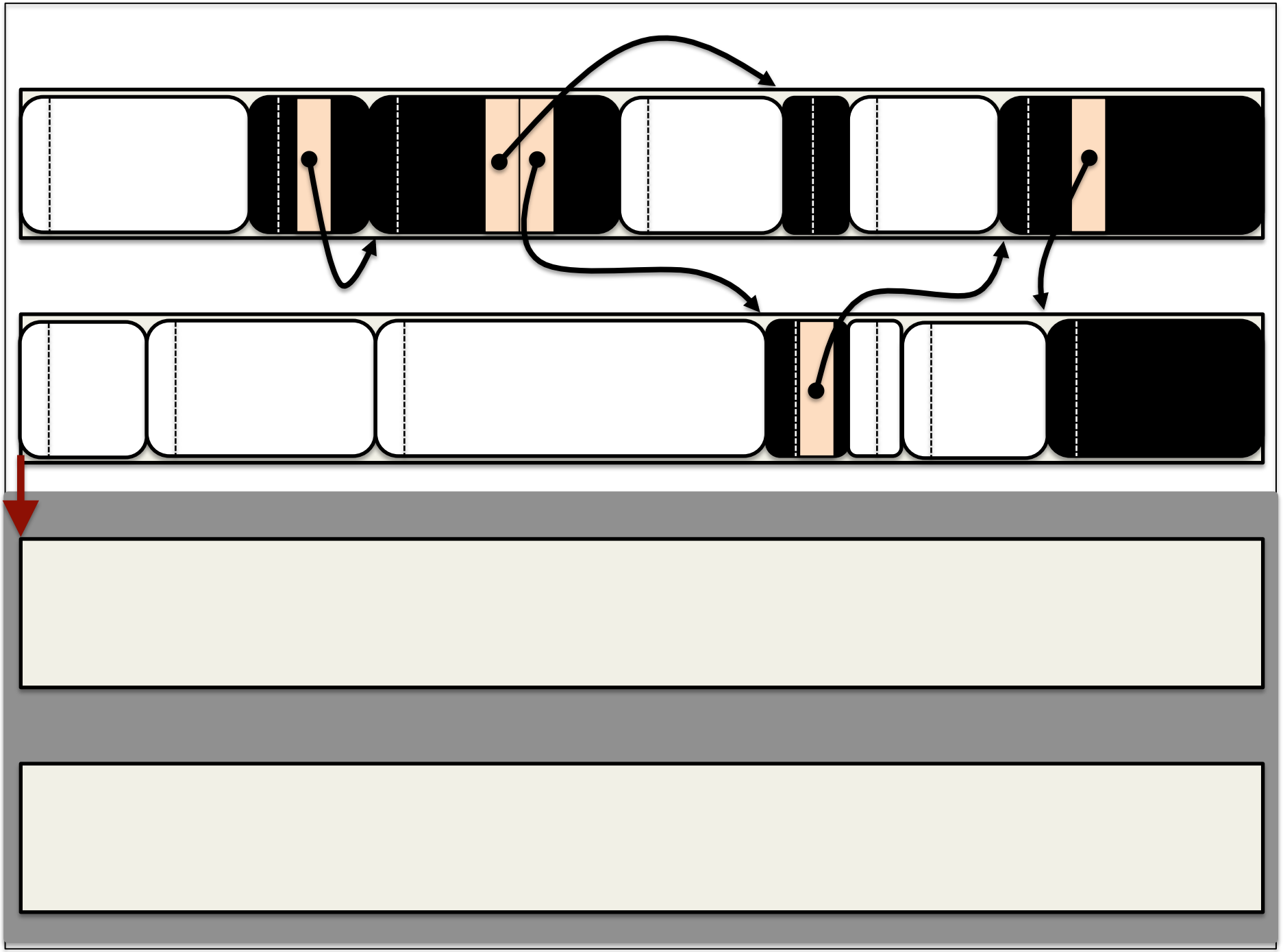
- Remember the two ways of storing metadata
  - Handles
  - Object headers
- Handles give us a single reference to an object
  - All other references go through the handle
- Solves the forwarding pointer problem
- Still other issues with handles

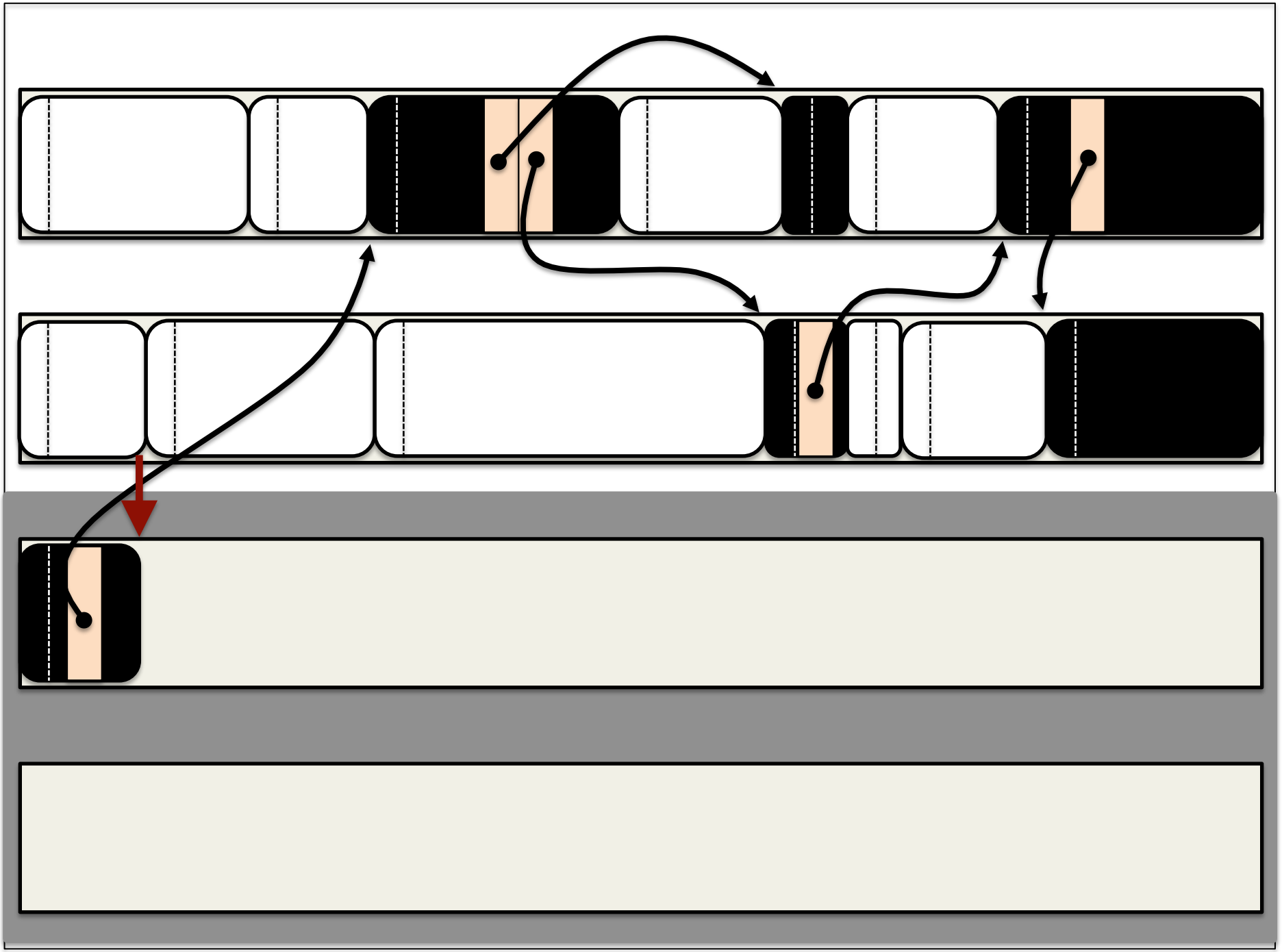
Forwarding Pointers

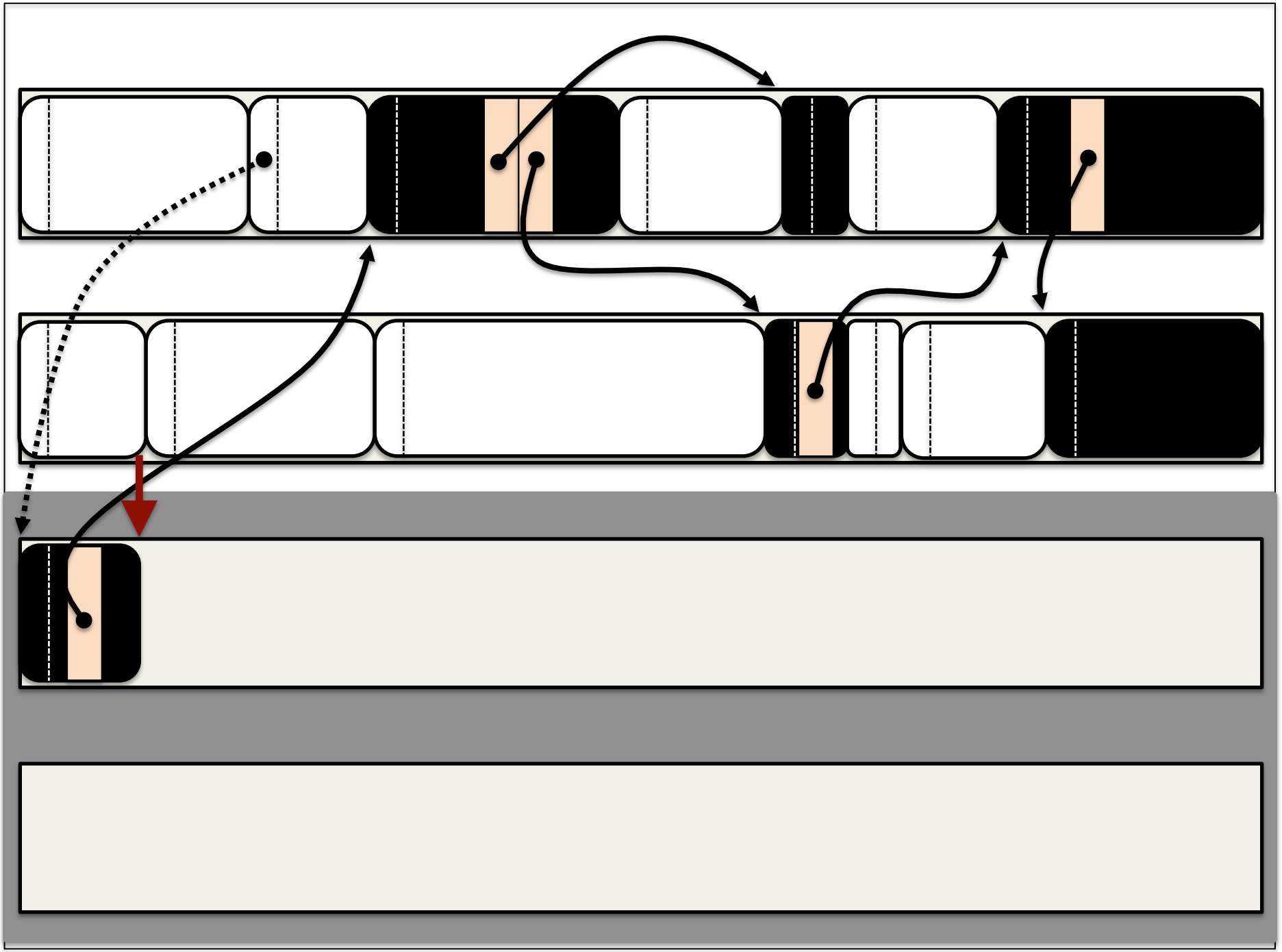
- We can't update all references when we move
  - Object graph is directional
  - We don't know where all the references are
- Instead we track where the object went
  - Fix up pointers in another pass
- Need some per-object metadata for pointer

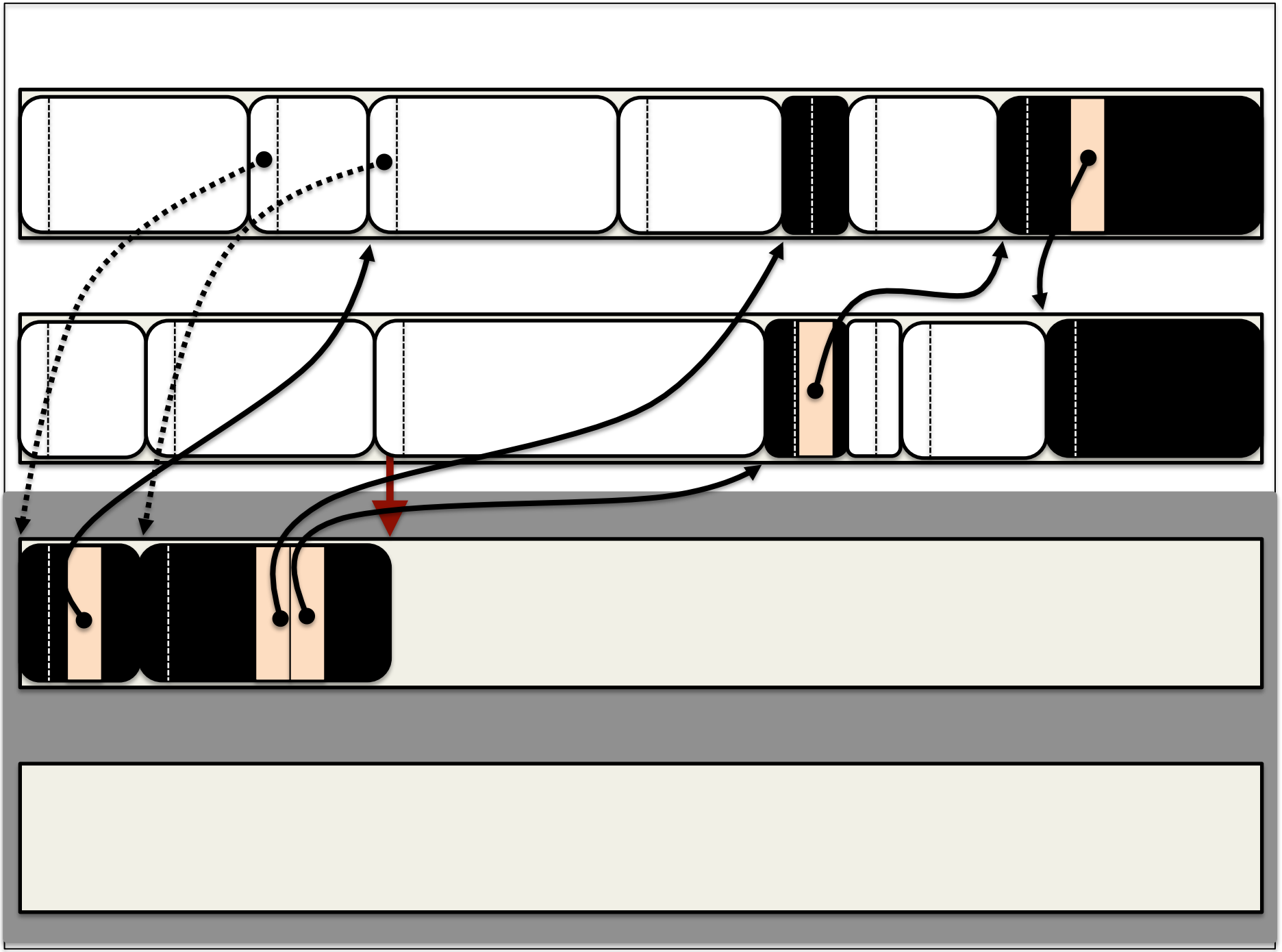
Forwarding Pointers

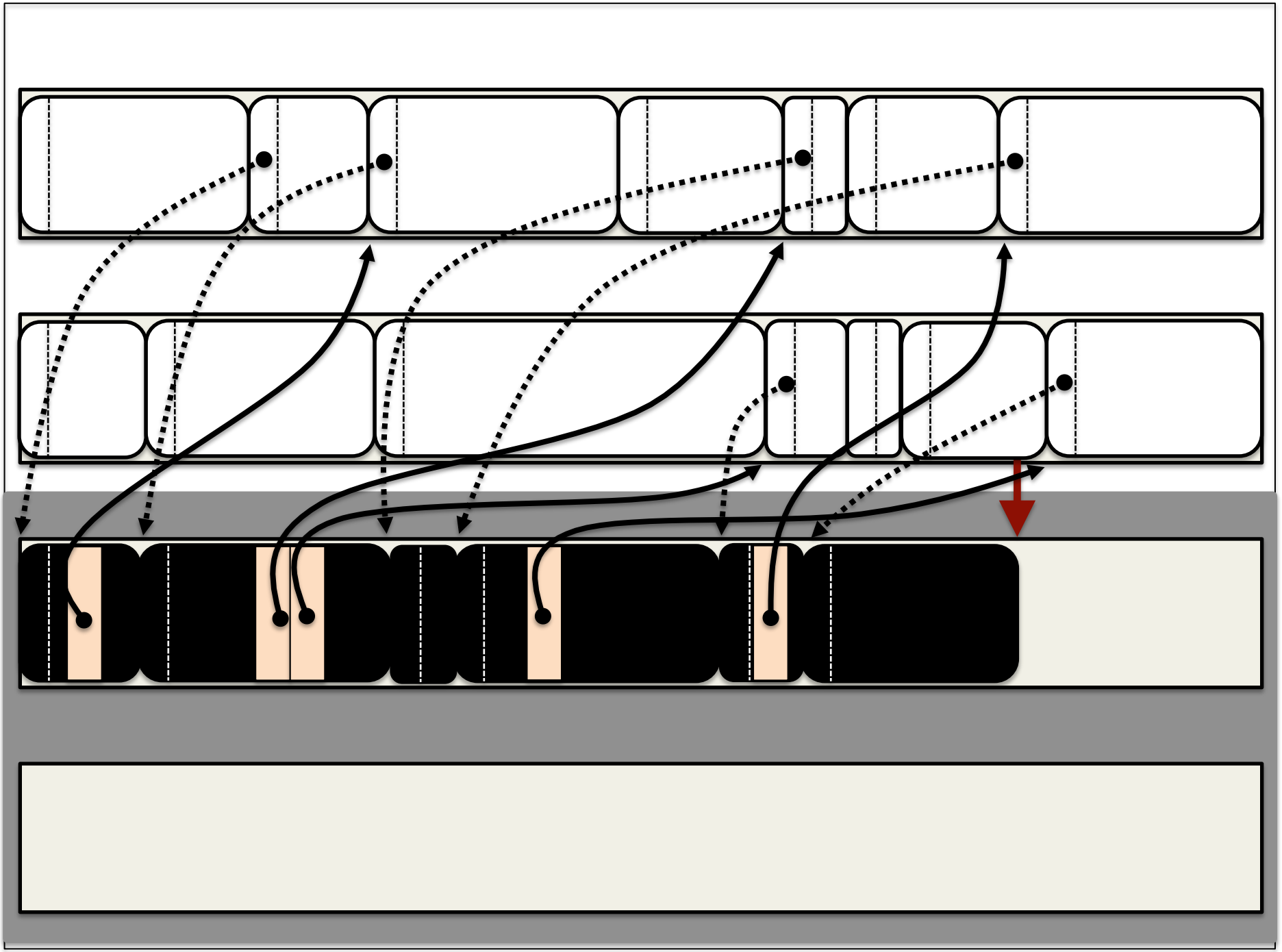
- We know that the old version is invalid
  - All objects in the old space are garbage
- We can overwrite all or part of that object
  - Install a forwarding pointer
  - Generally use a word in the header
- The GC has to recognize a forwarding pointer
  - Steal the low-order bit to use as a flag

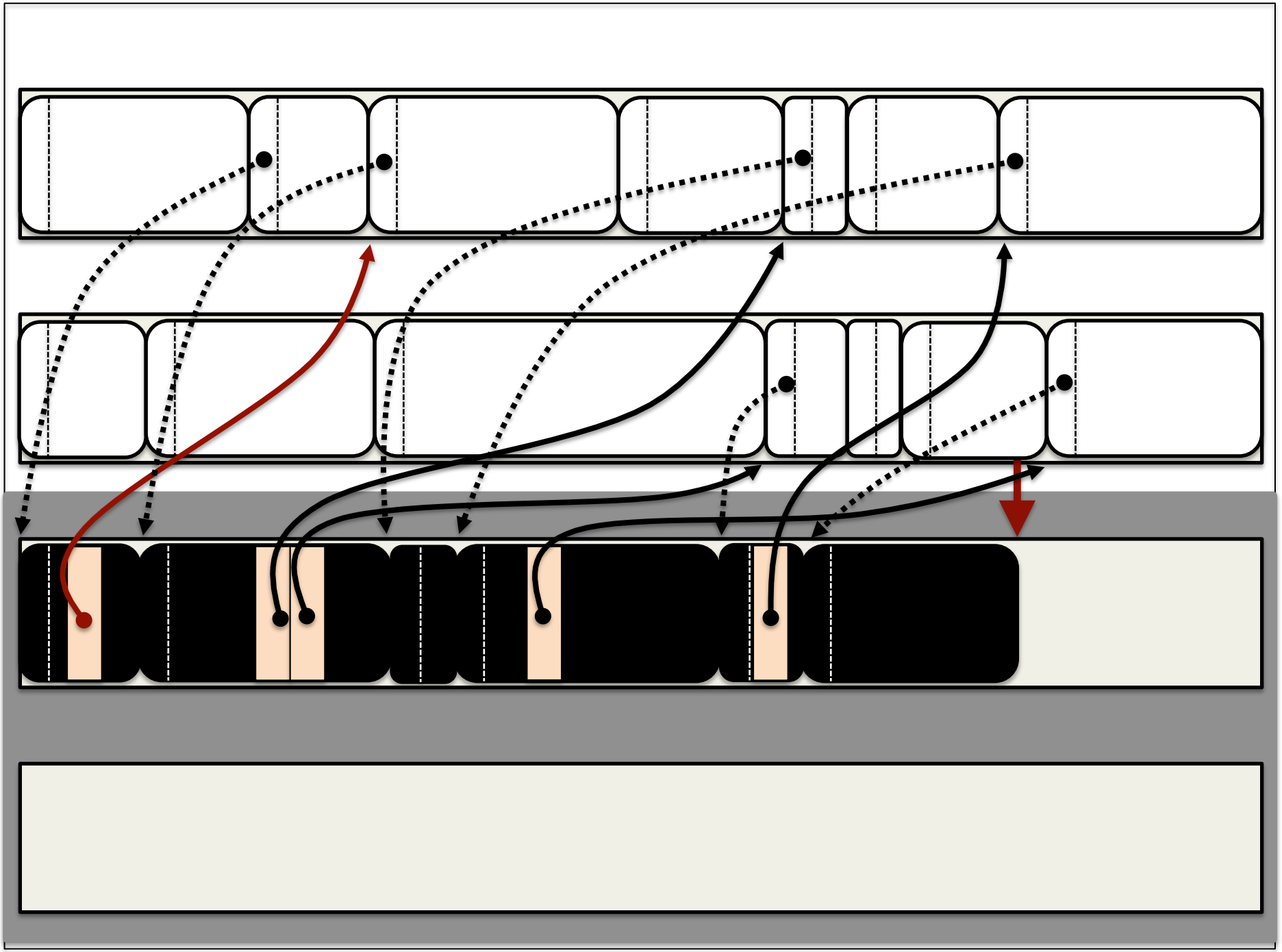


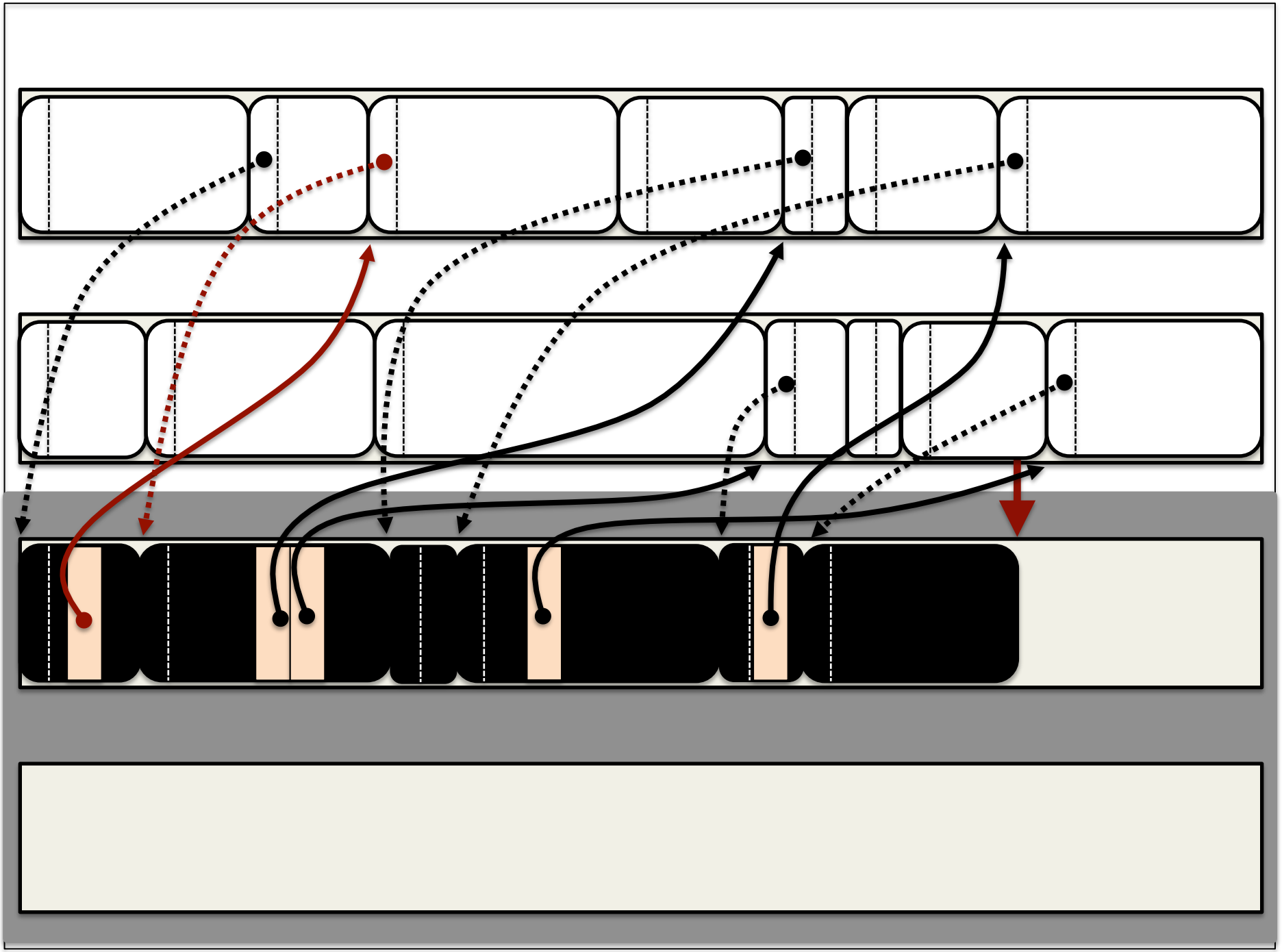


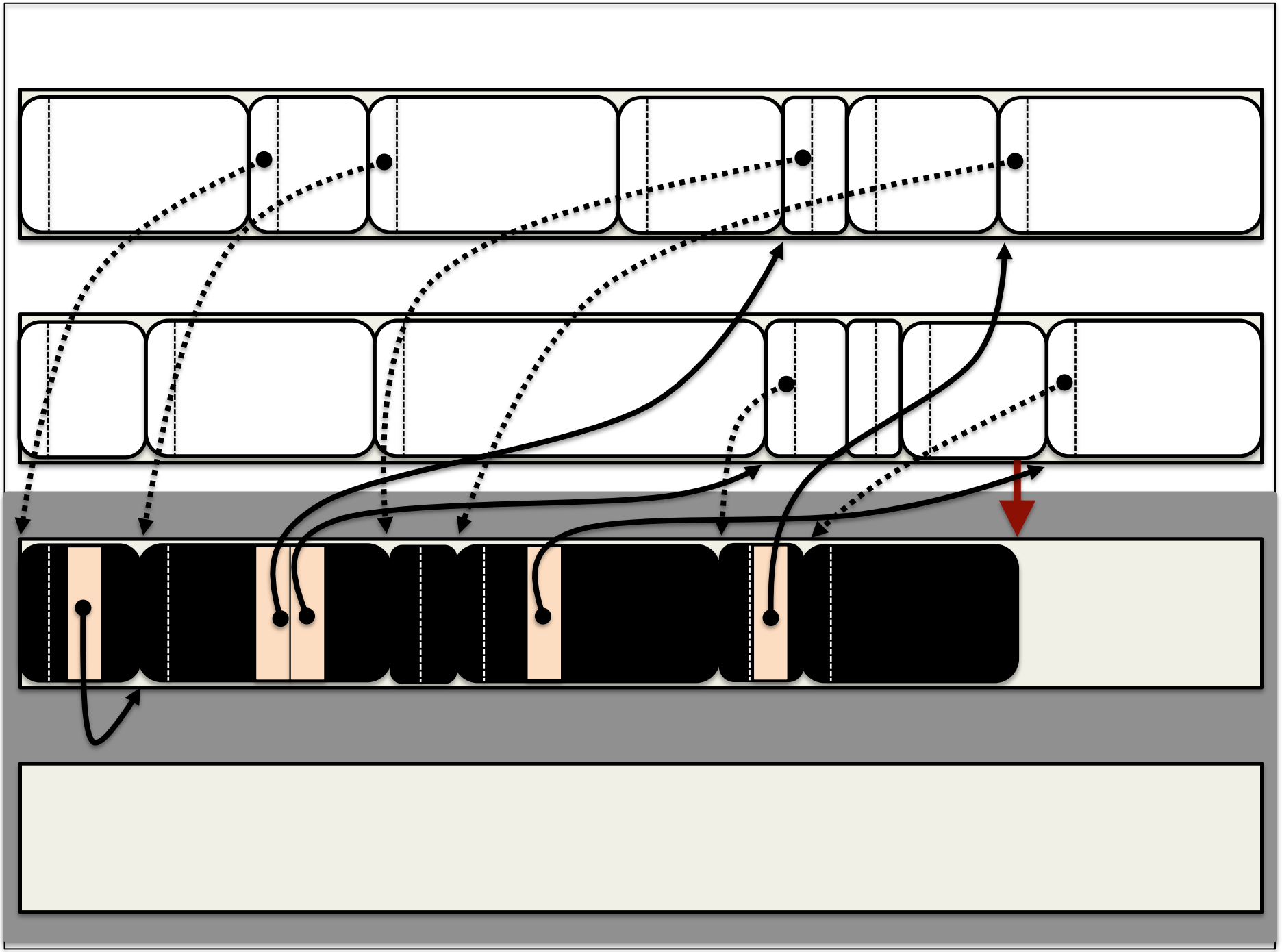


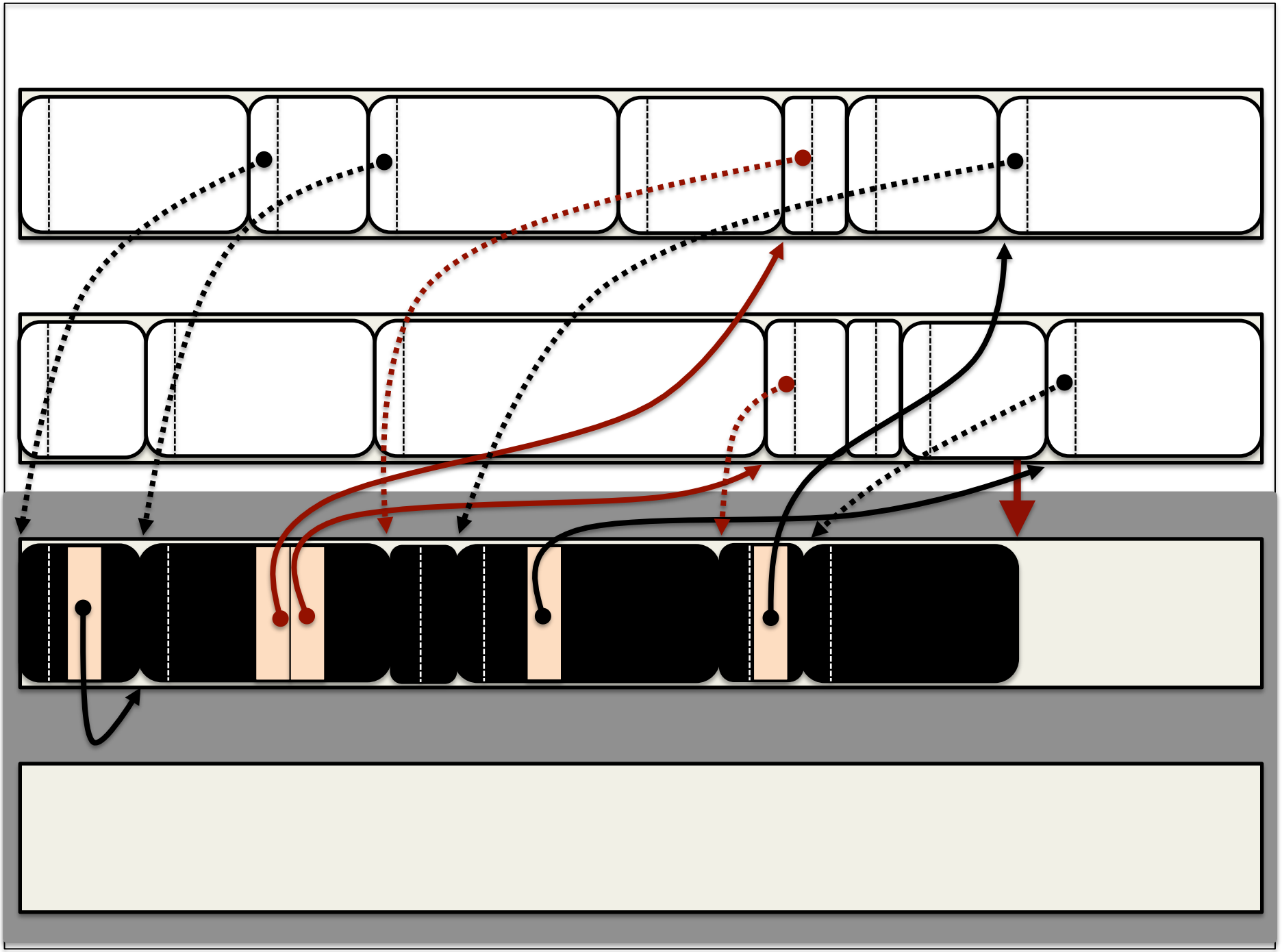


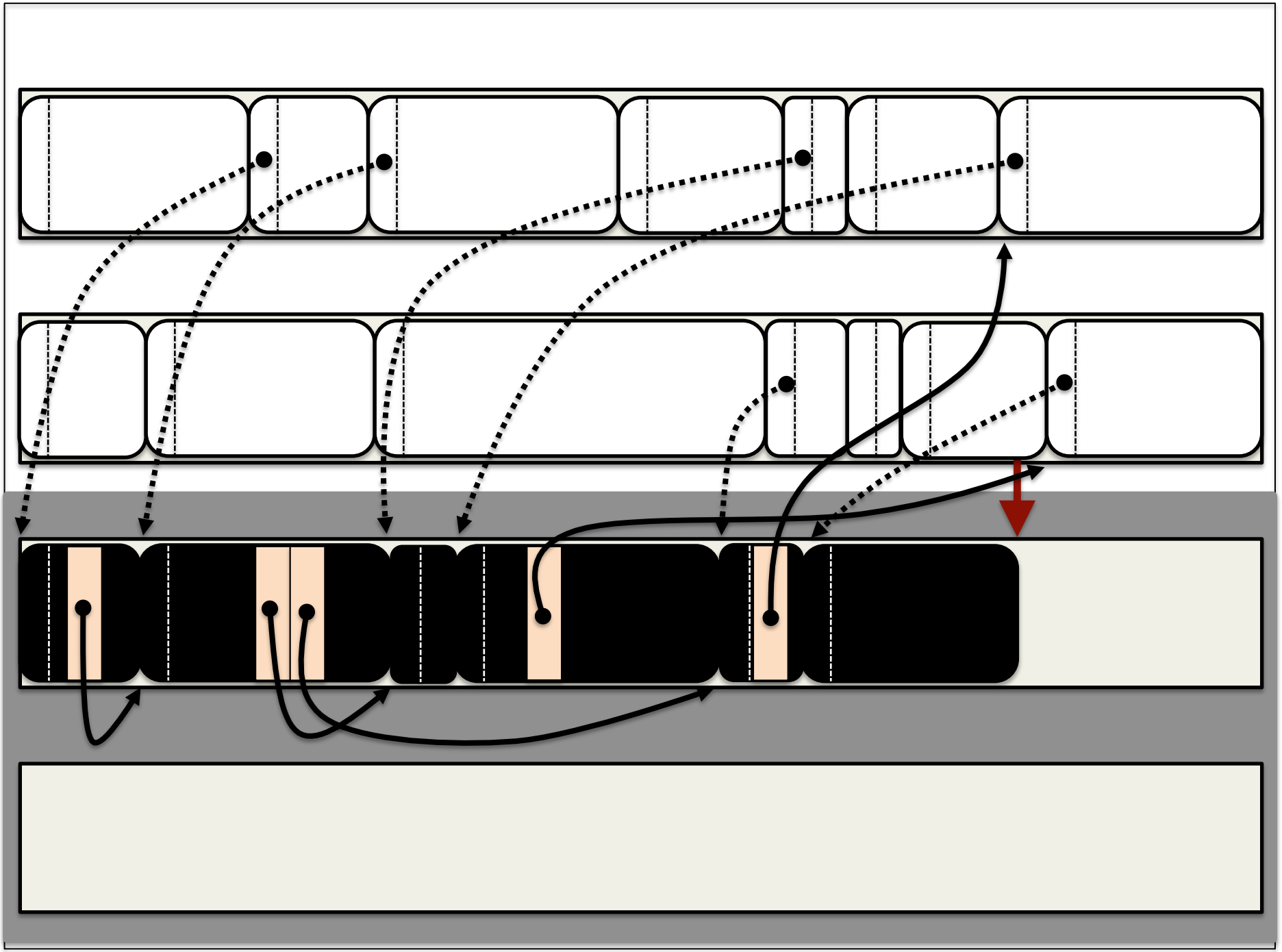


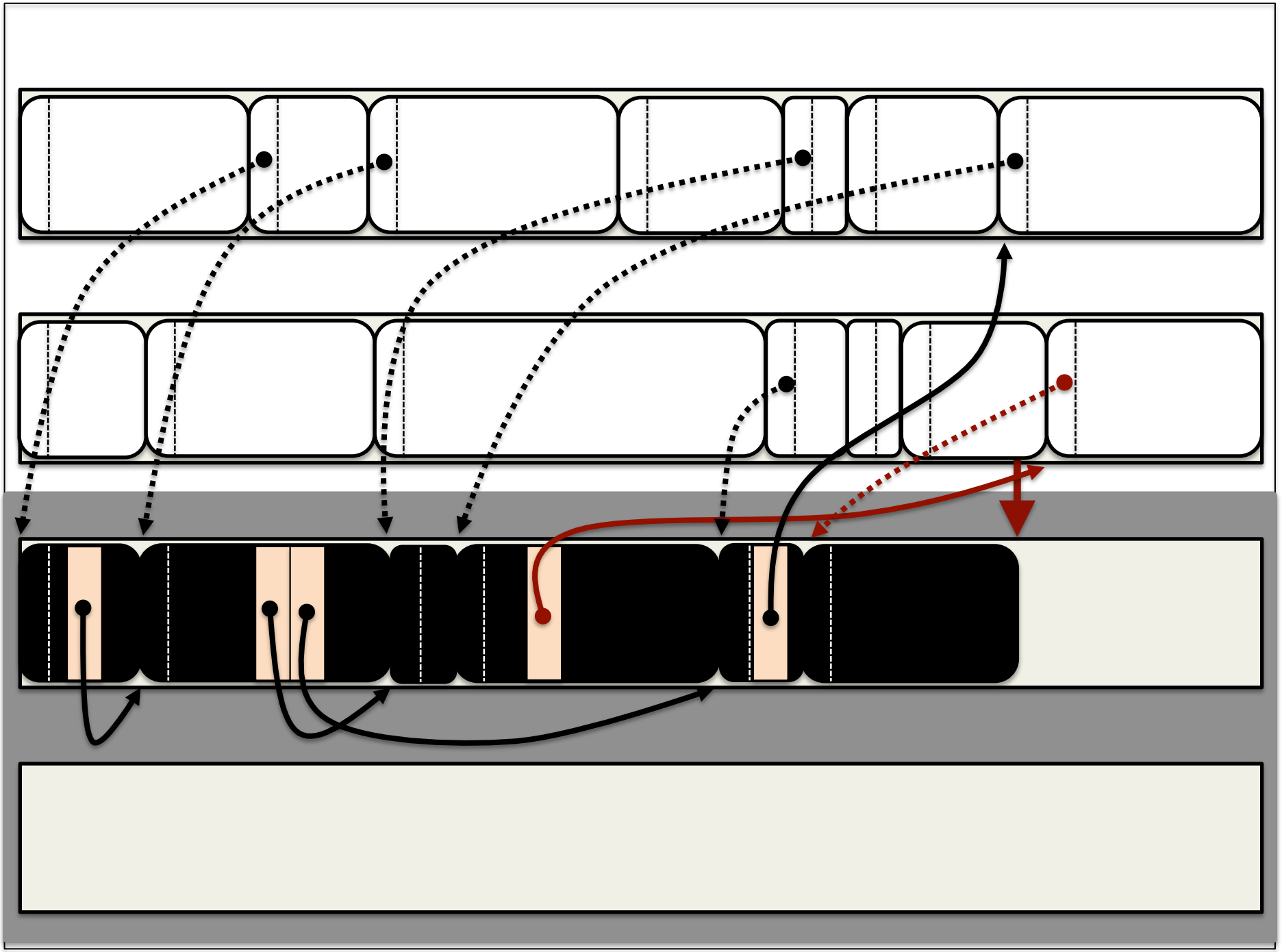


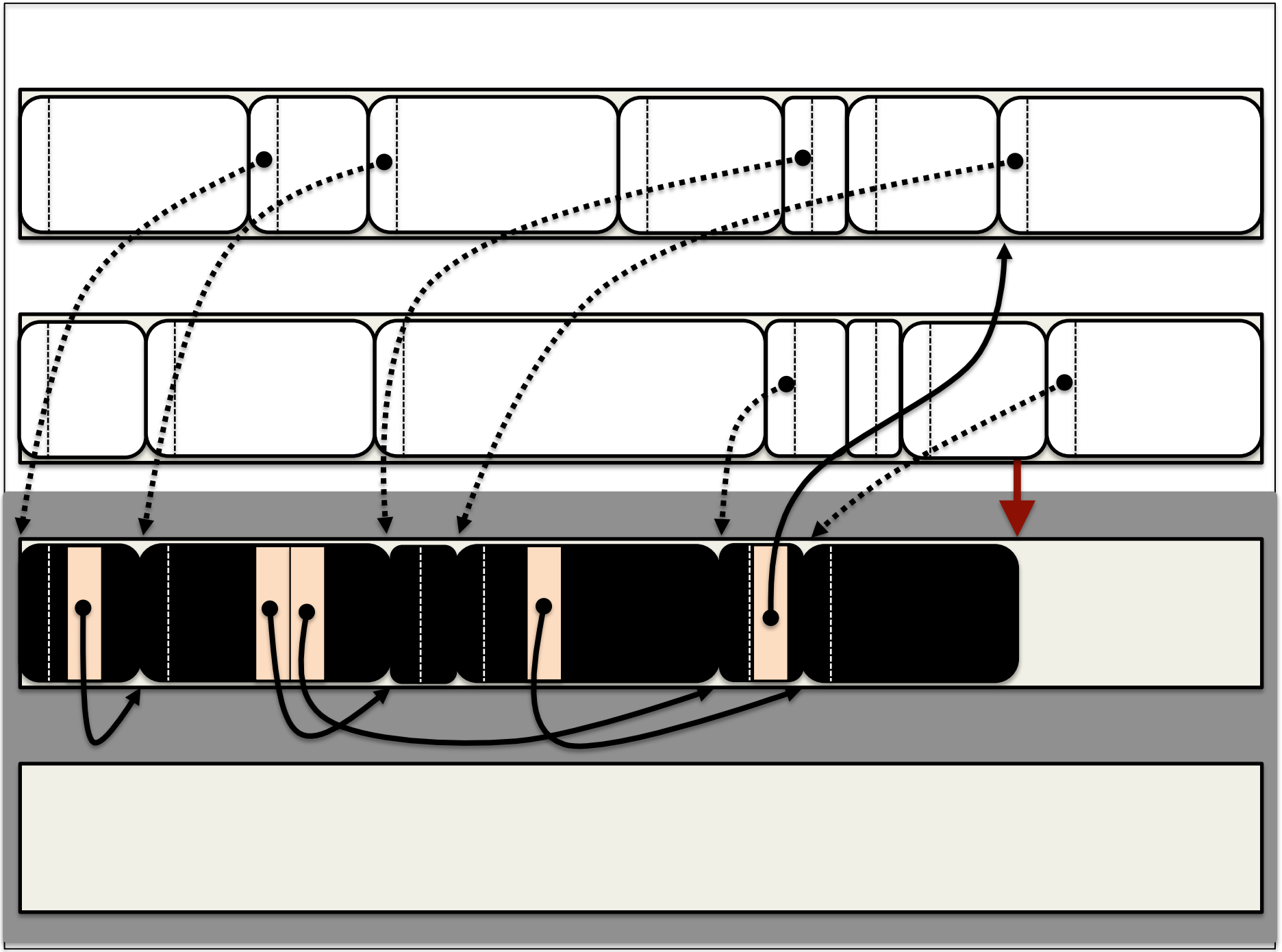


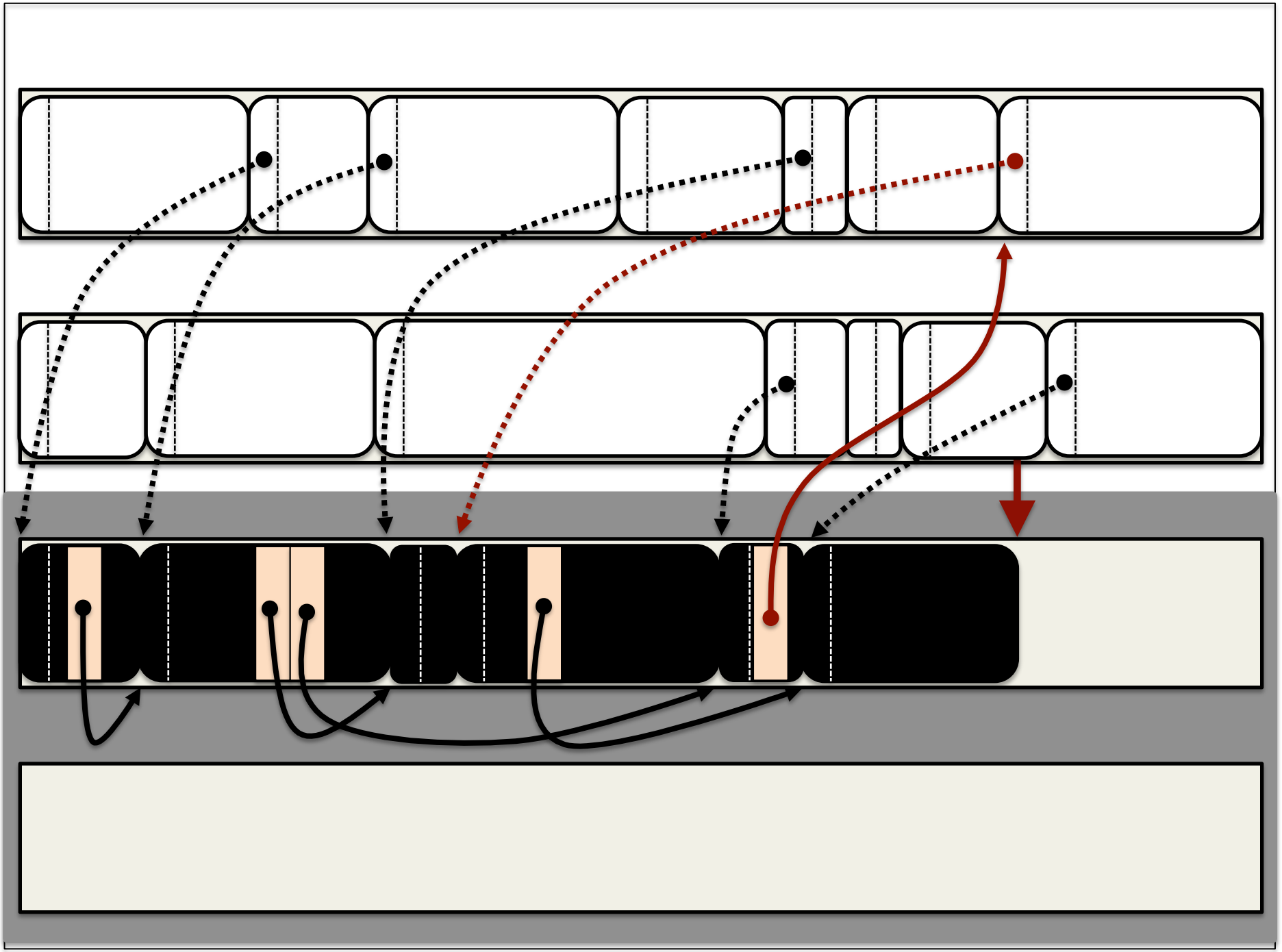


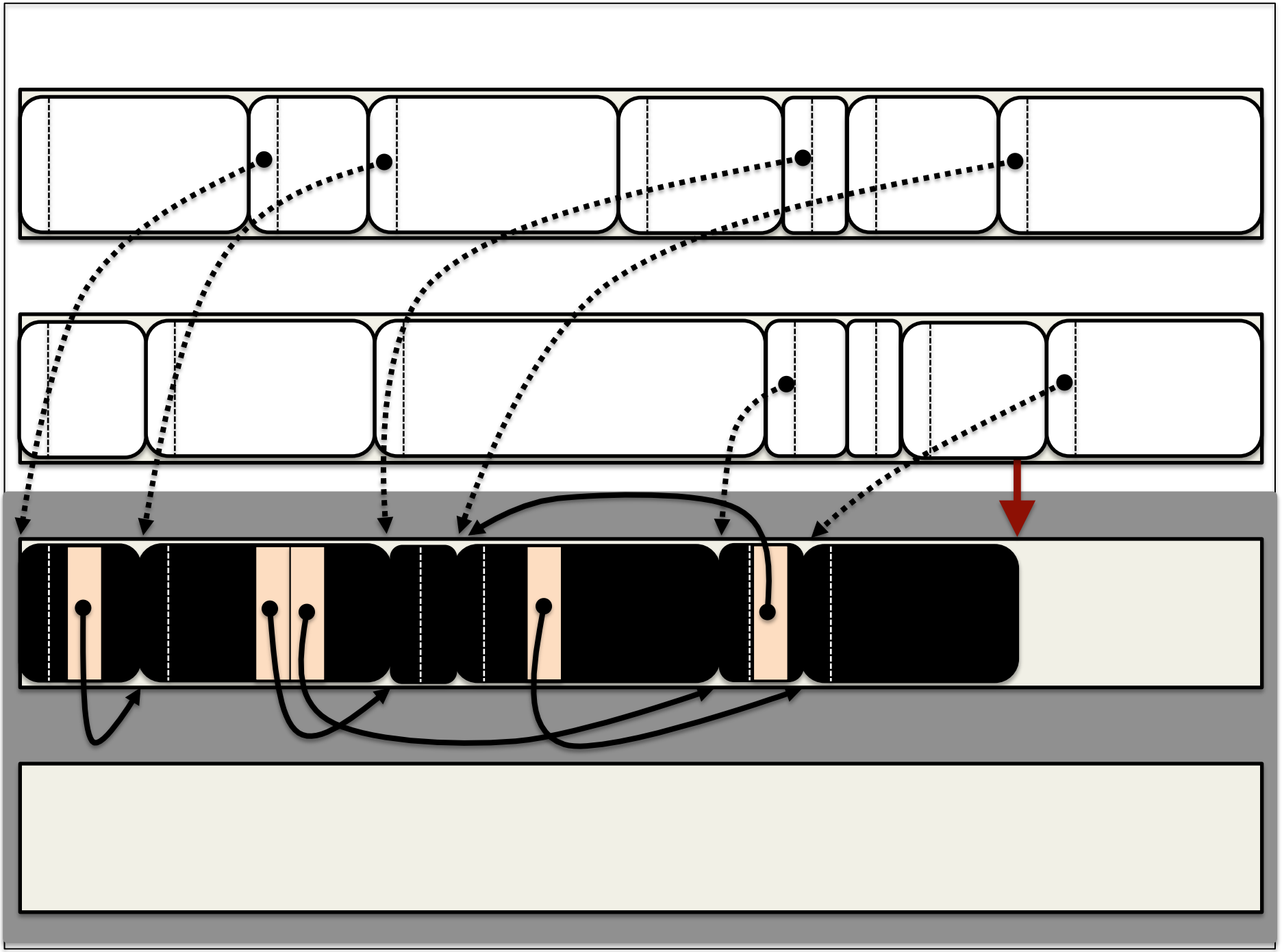






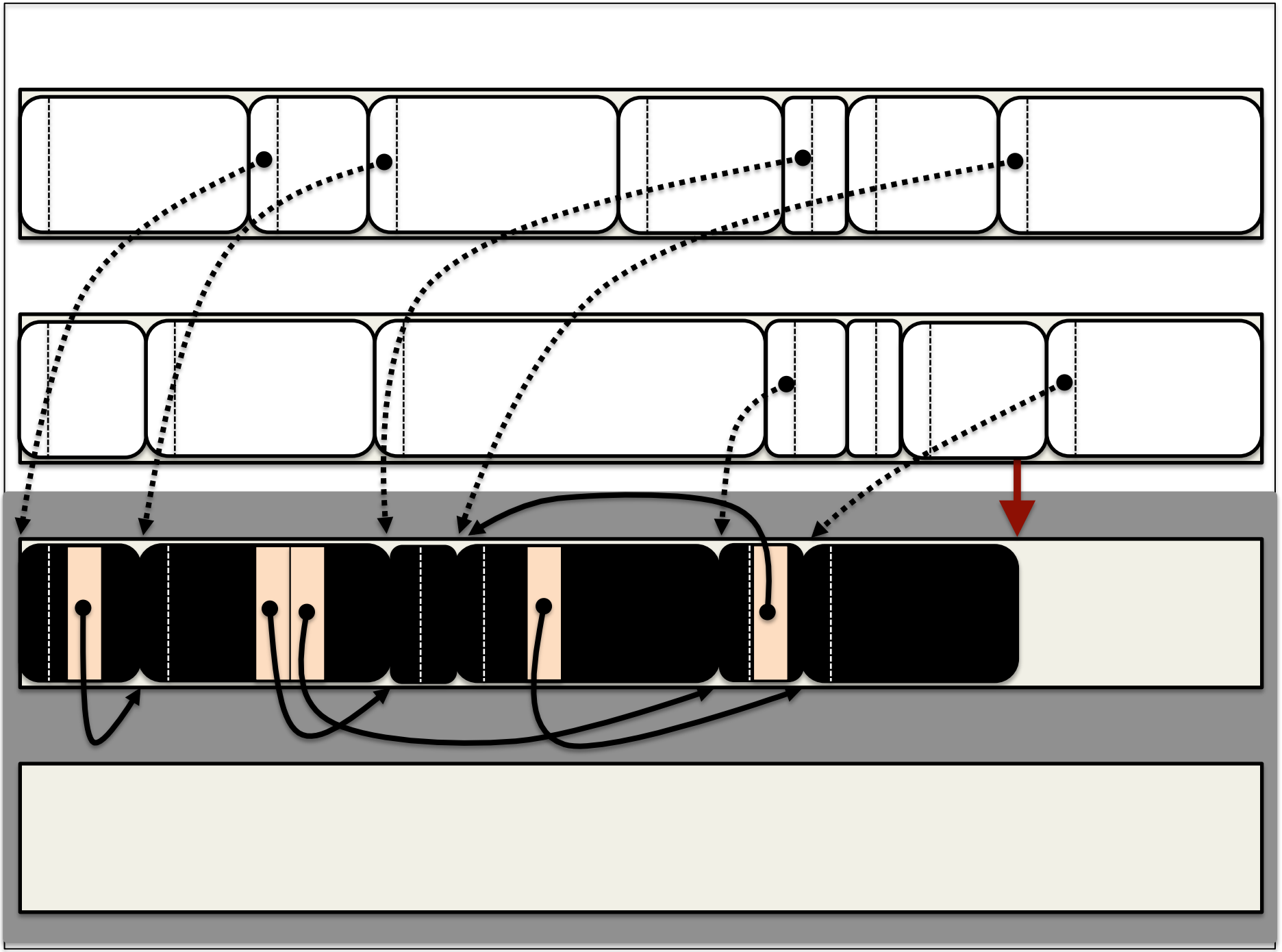


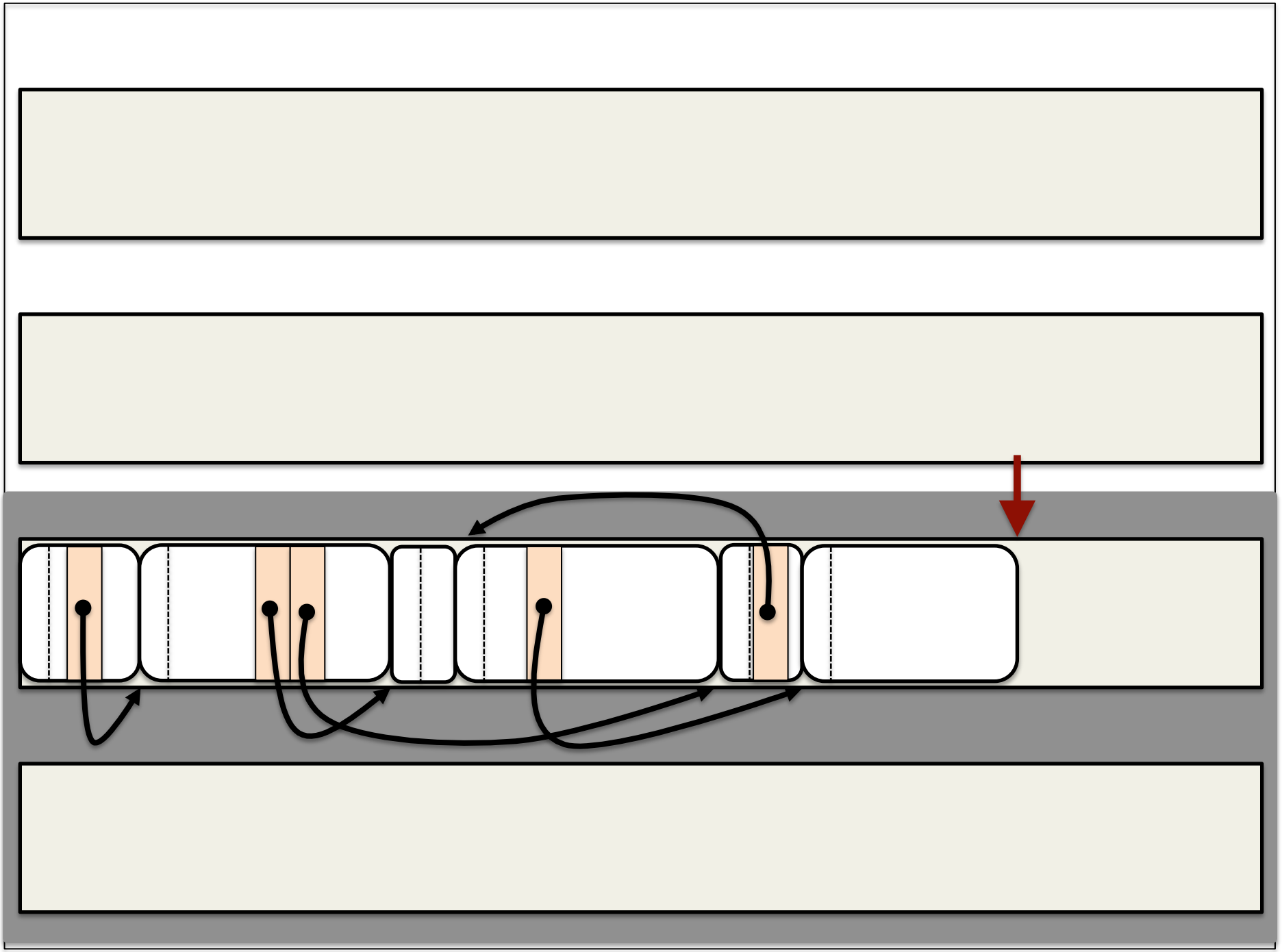


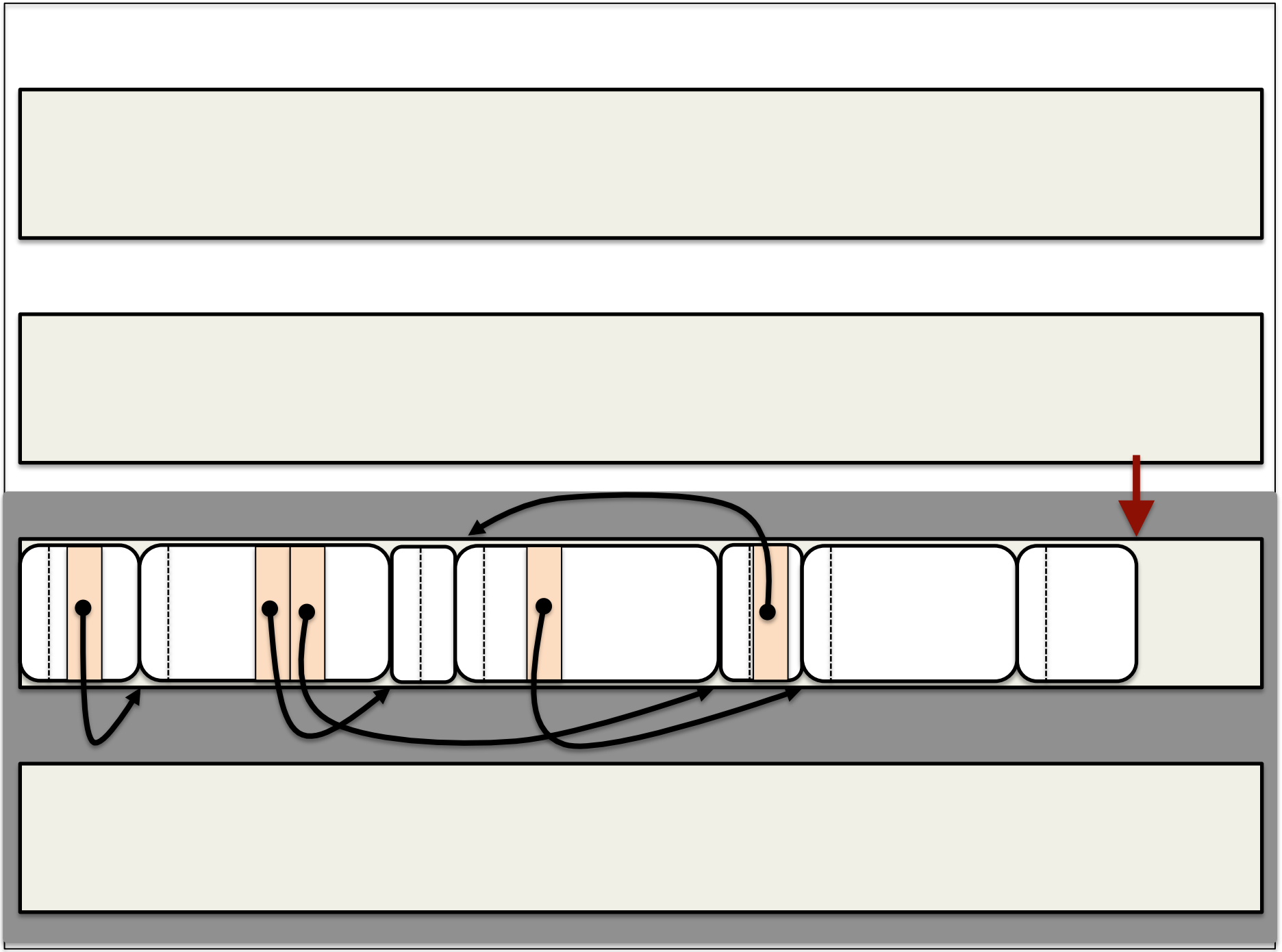


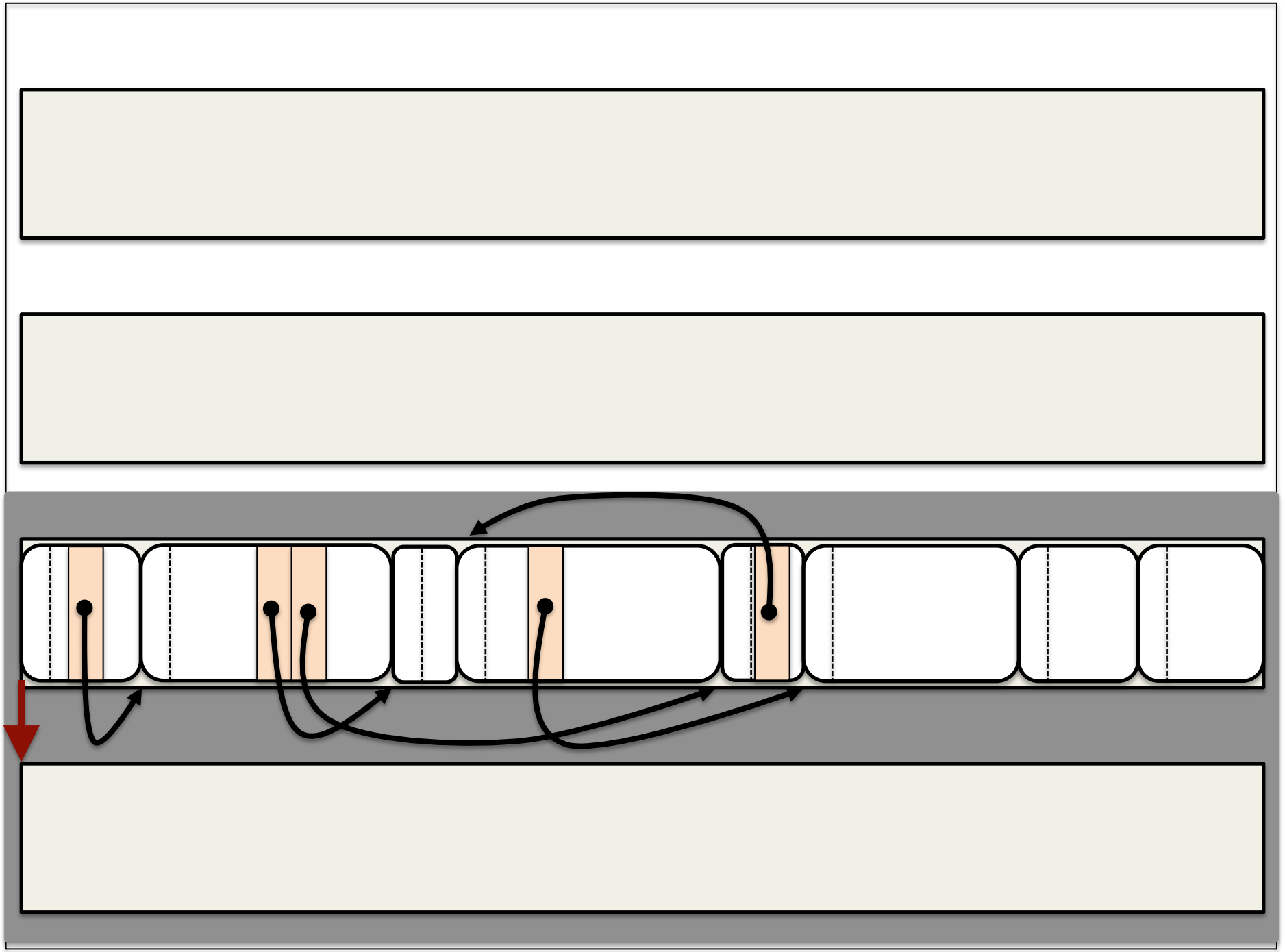
Finishing Up

- Remember to update the roots as well
  - There can be no references to the old space
- The old space can now be reused
  - Reset the bump pointer to the start of the space
- Allocation continues to the new space









Copying Collectors

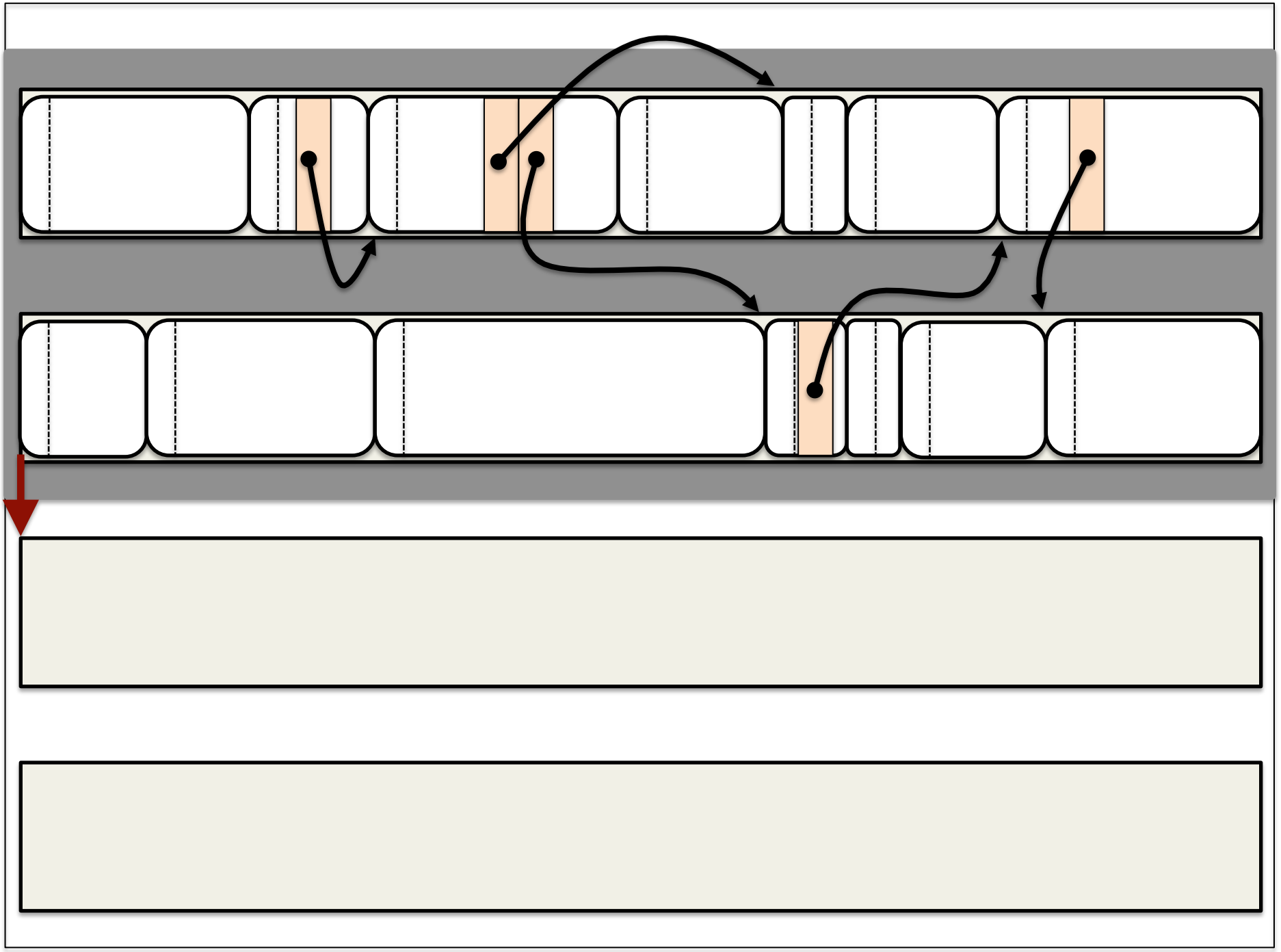
- Moving objects gives us some nice features
  - No fragmentation
  - Much better cache behavior
- Comes with some large overheads
  - Copying can be expensive
  - Wastes half of the heap
- We'll see how to reduce these over time

Object Mobility

- For copying to work, all objects must be mobile
- Any pinned objects create holes in the heap
  - Bump-pointer allocation can't use the holes
  - Scanning algorithms become much harder
  - Basic assumption is that heap is contiguous
- Imposes limitations throughout the VM
  - Particularly when working with native code

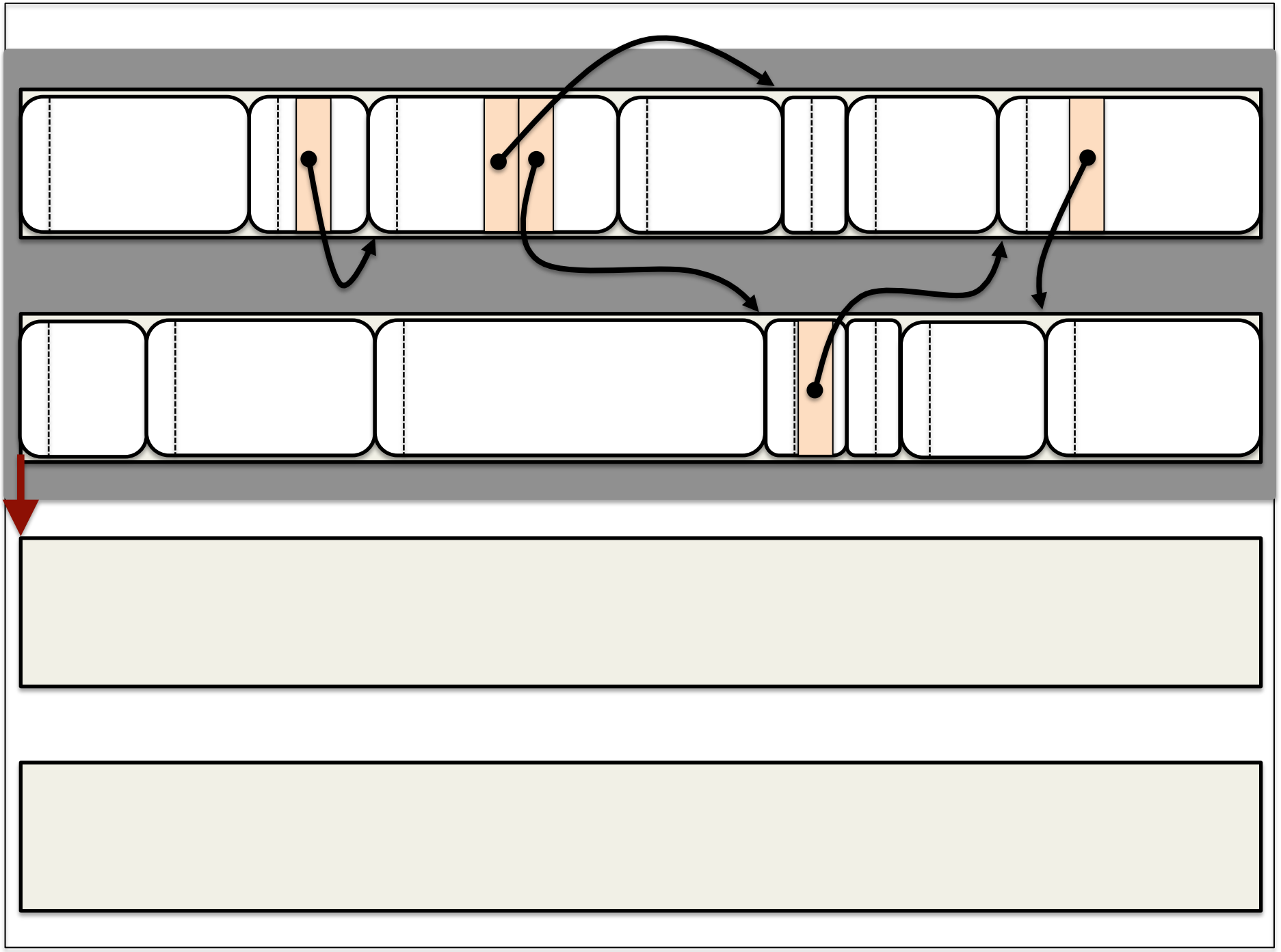
Cheney's Algorithm

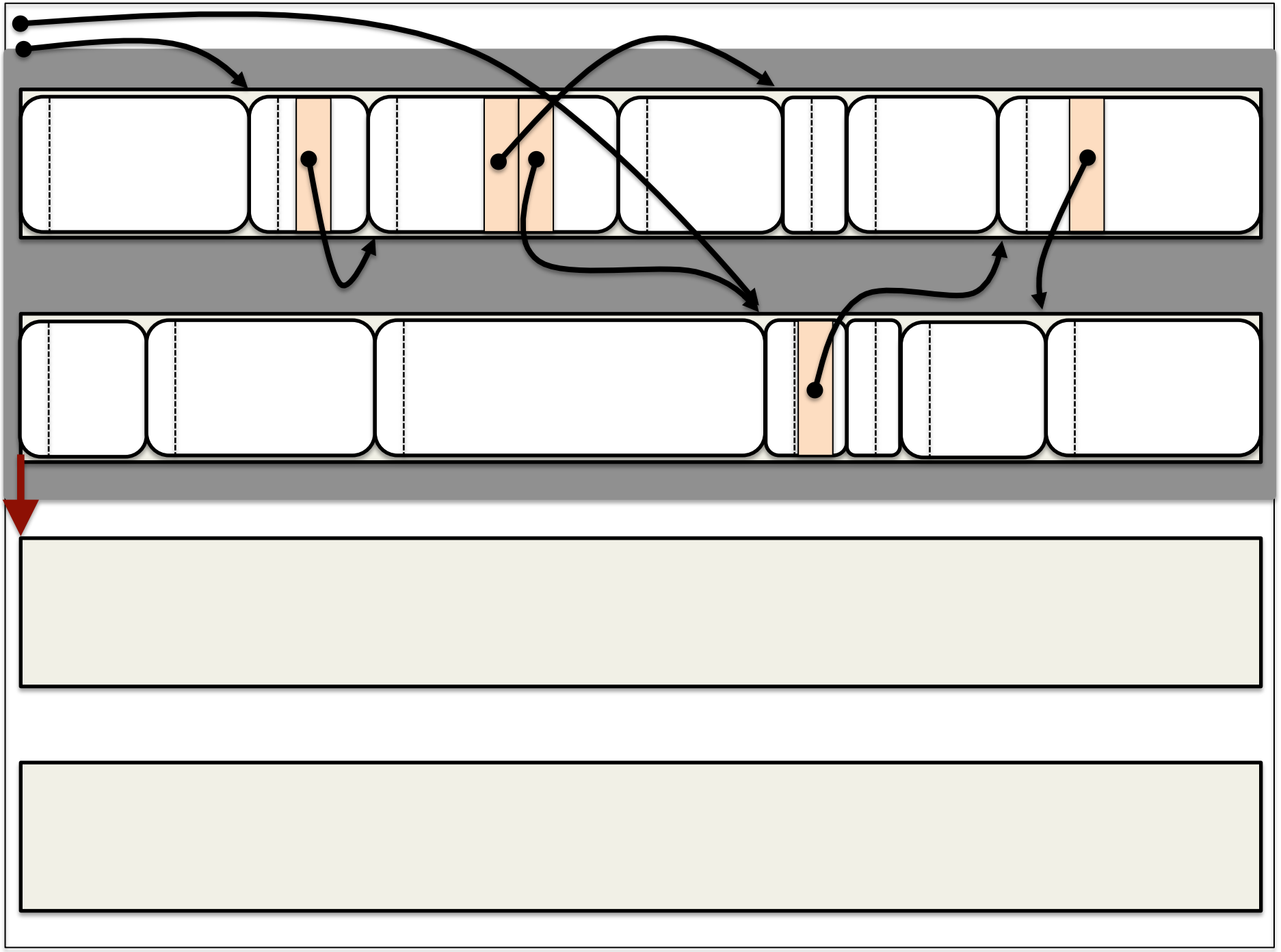
- Proposed in 1970
  - Tends to be the canonical copying algorithm
- Uses a single pass of the heap
  - Merges marking and copying phases
  - Updates references as it goes
- You will implement this algorithm in SimpleJava
  - Assignment 3 is now available

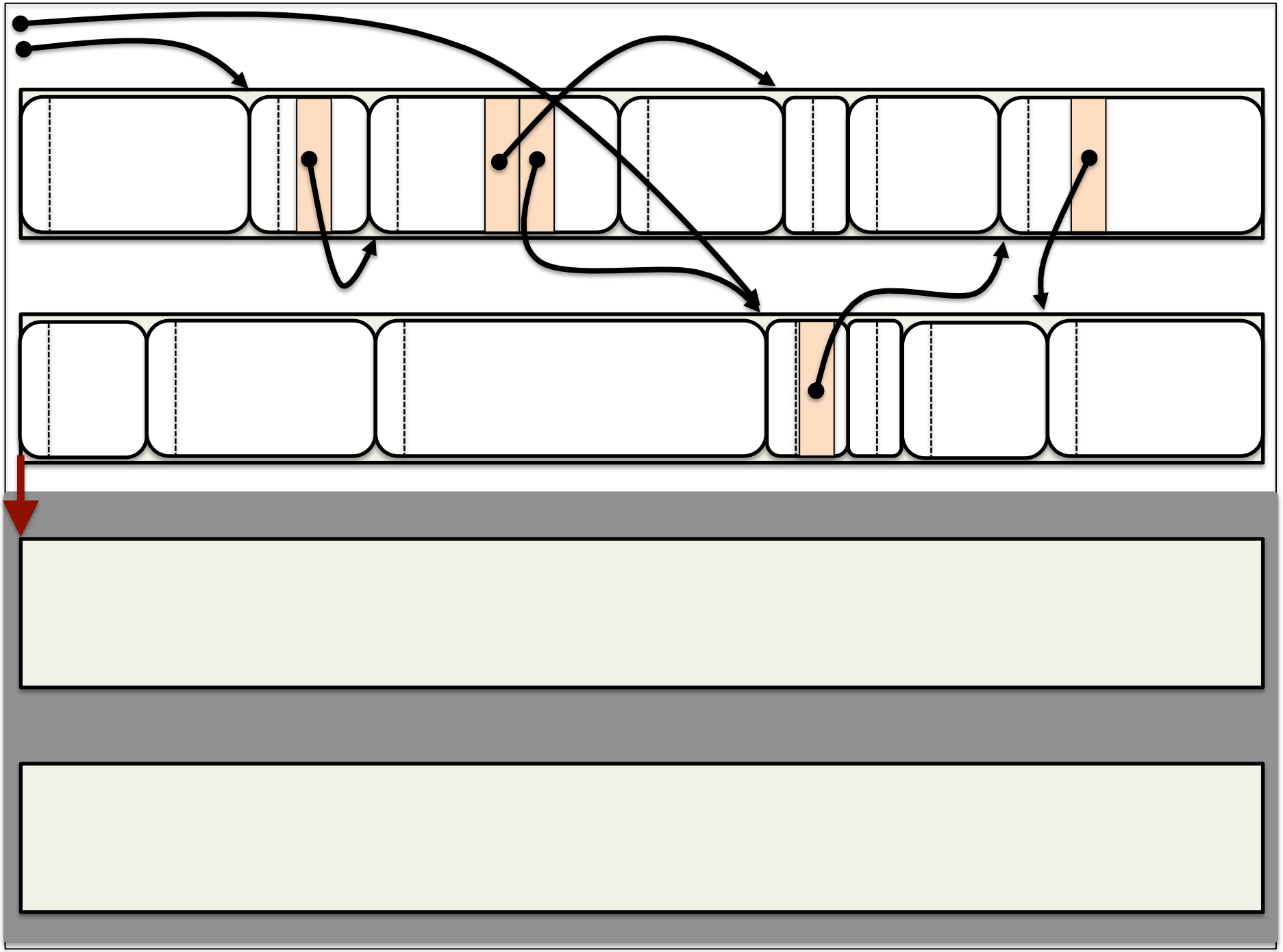


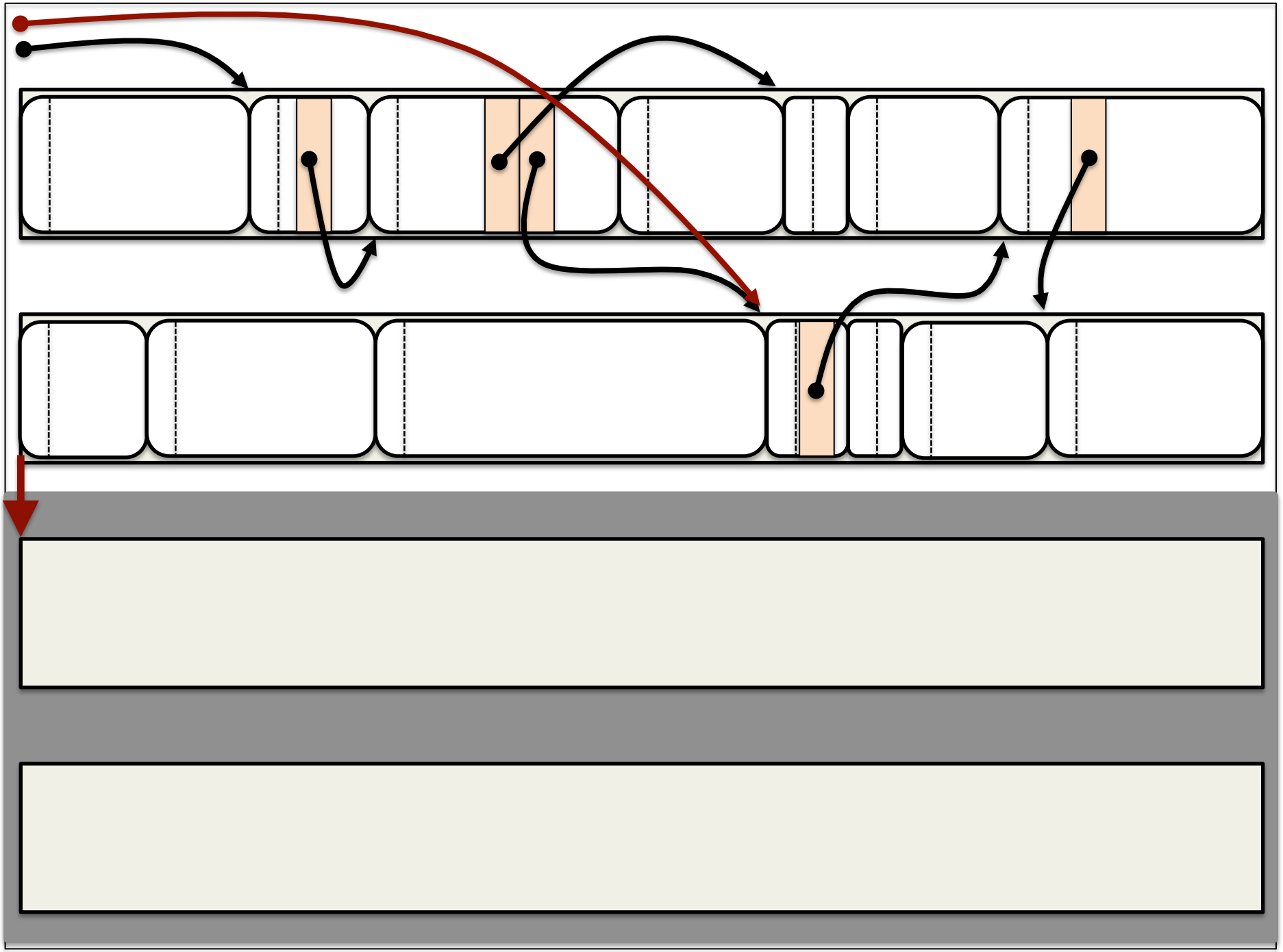
Root Scanning

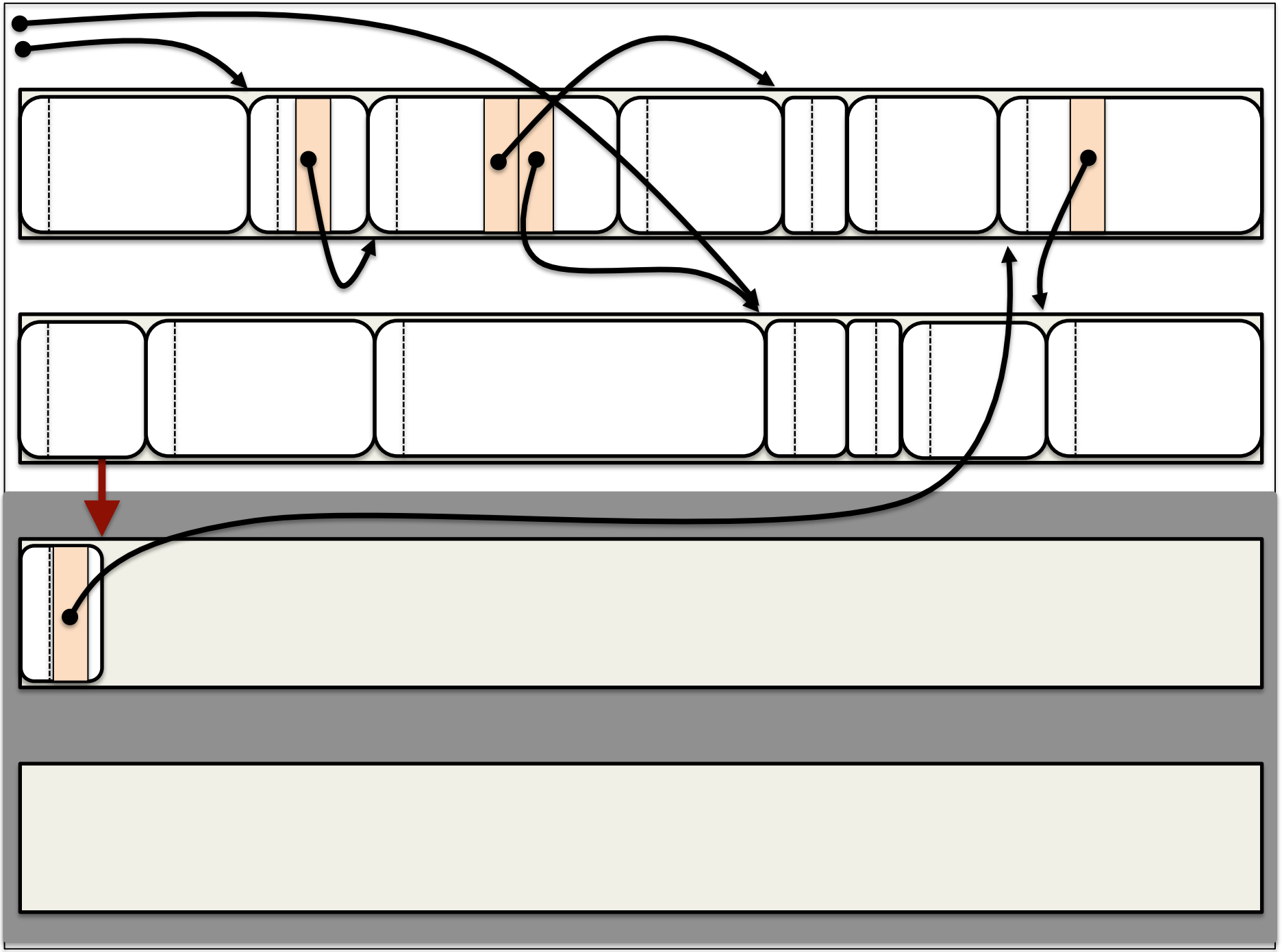
- First step of any tracing algorithm
  - Copy all objects pointed to by a root
- Root set of a VM is implementation dependent
- SimpleJava has three root locations
  - Thread stack and local references
  - Class static references
  - String intern table

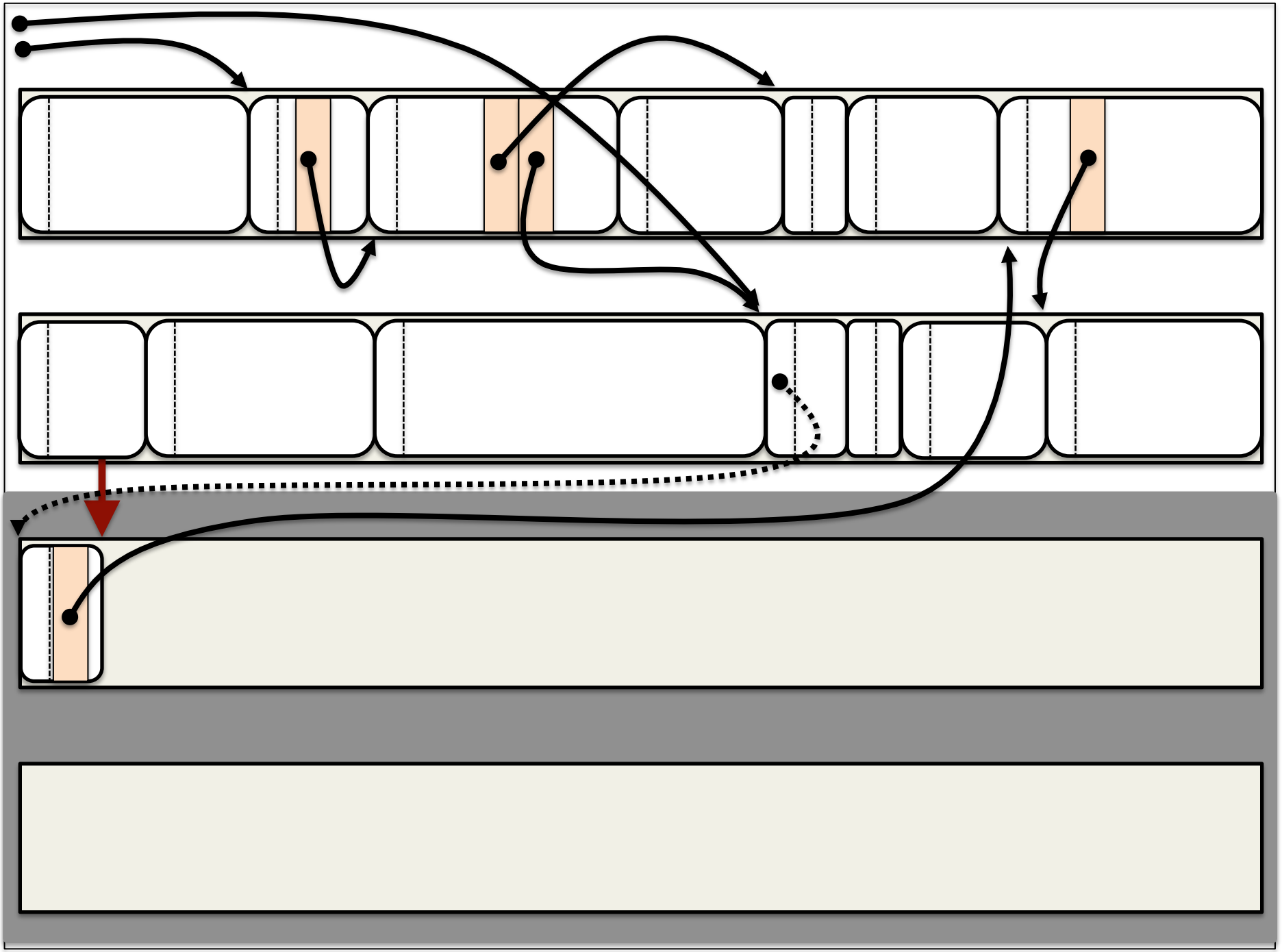


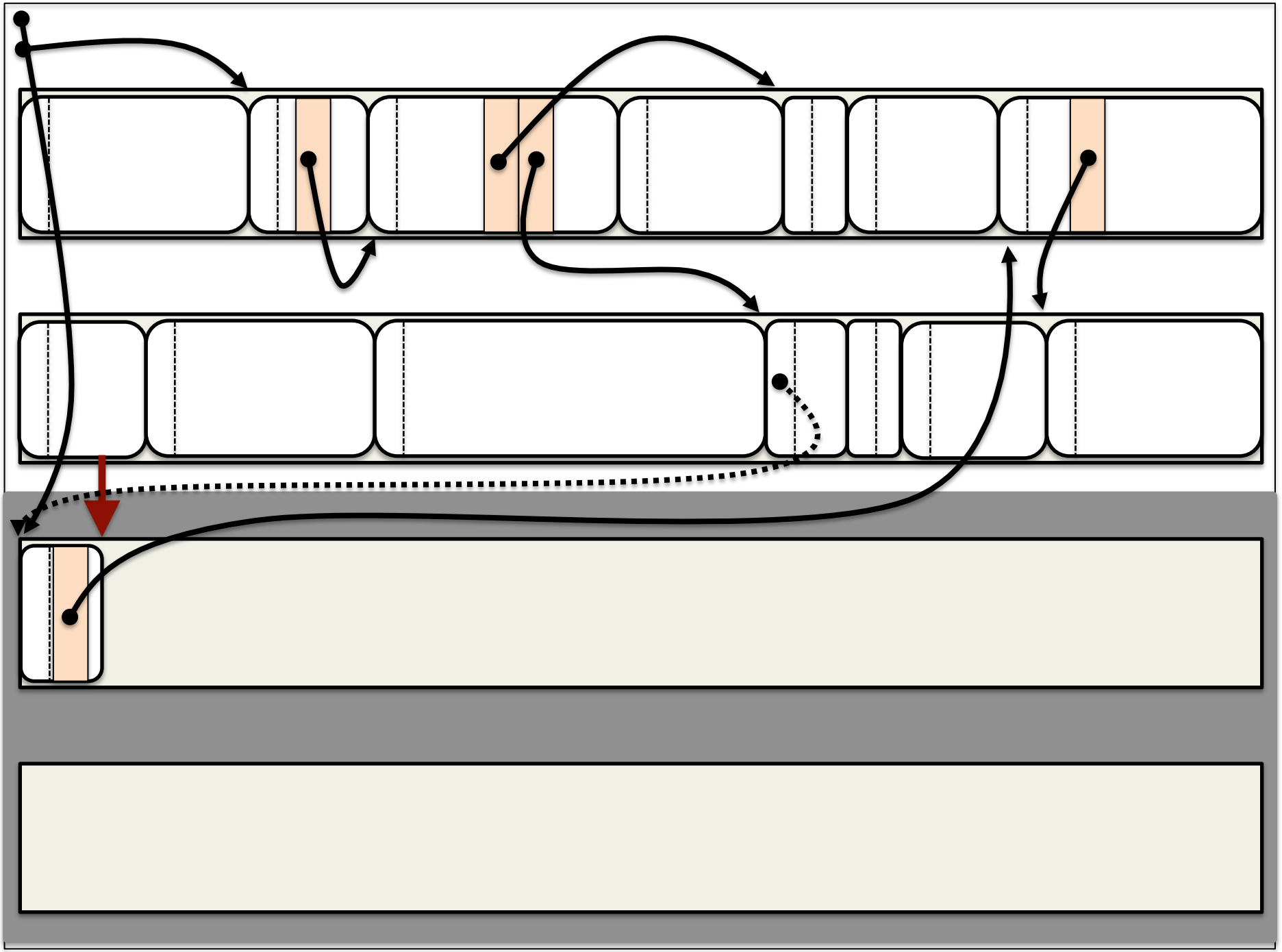


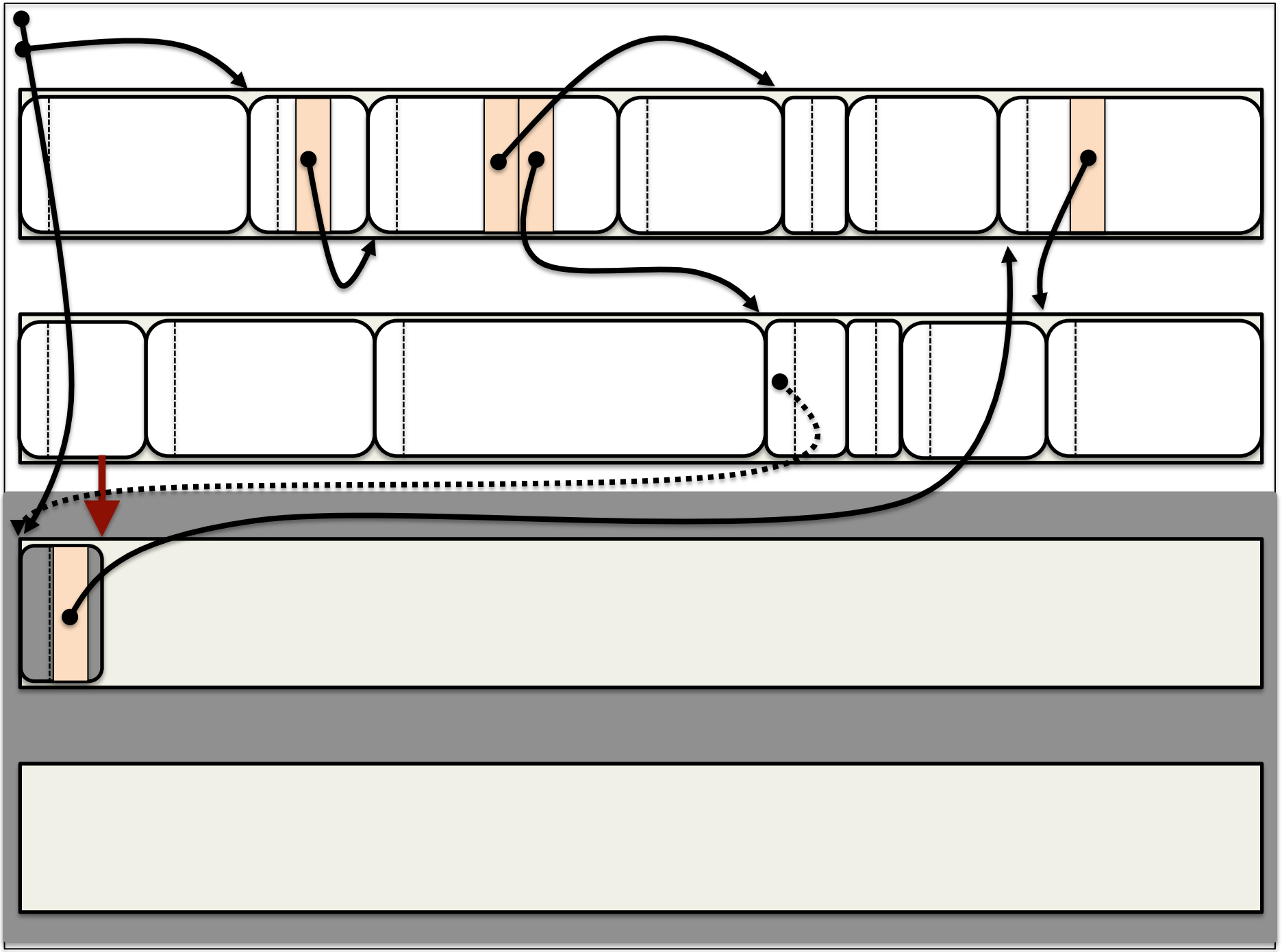


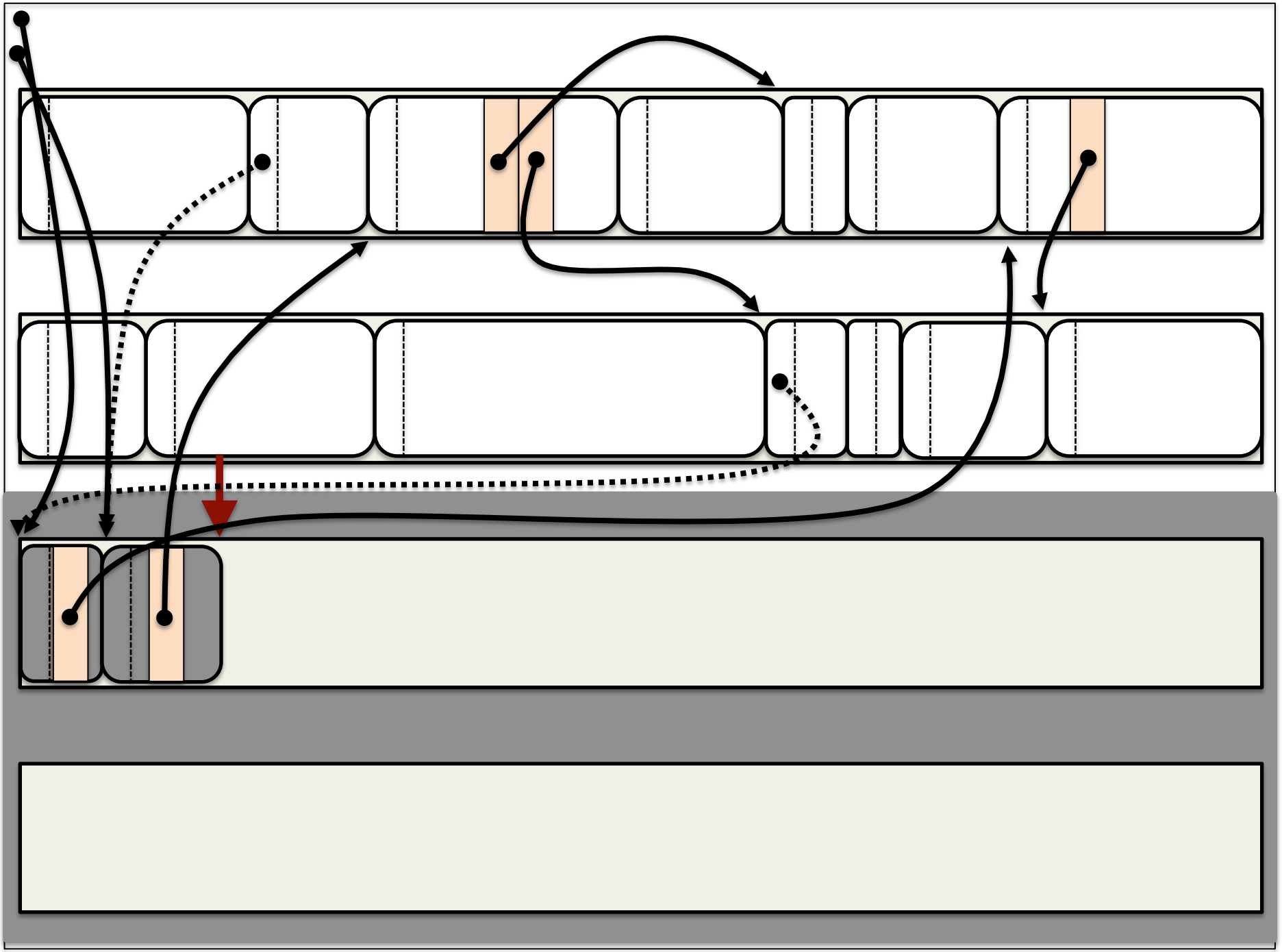






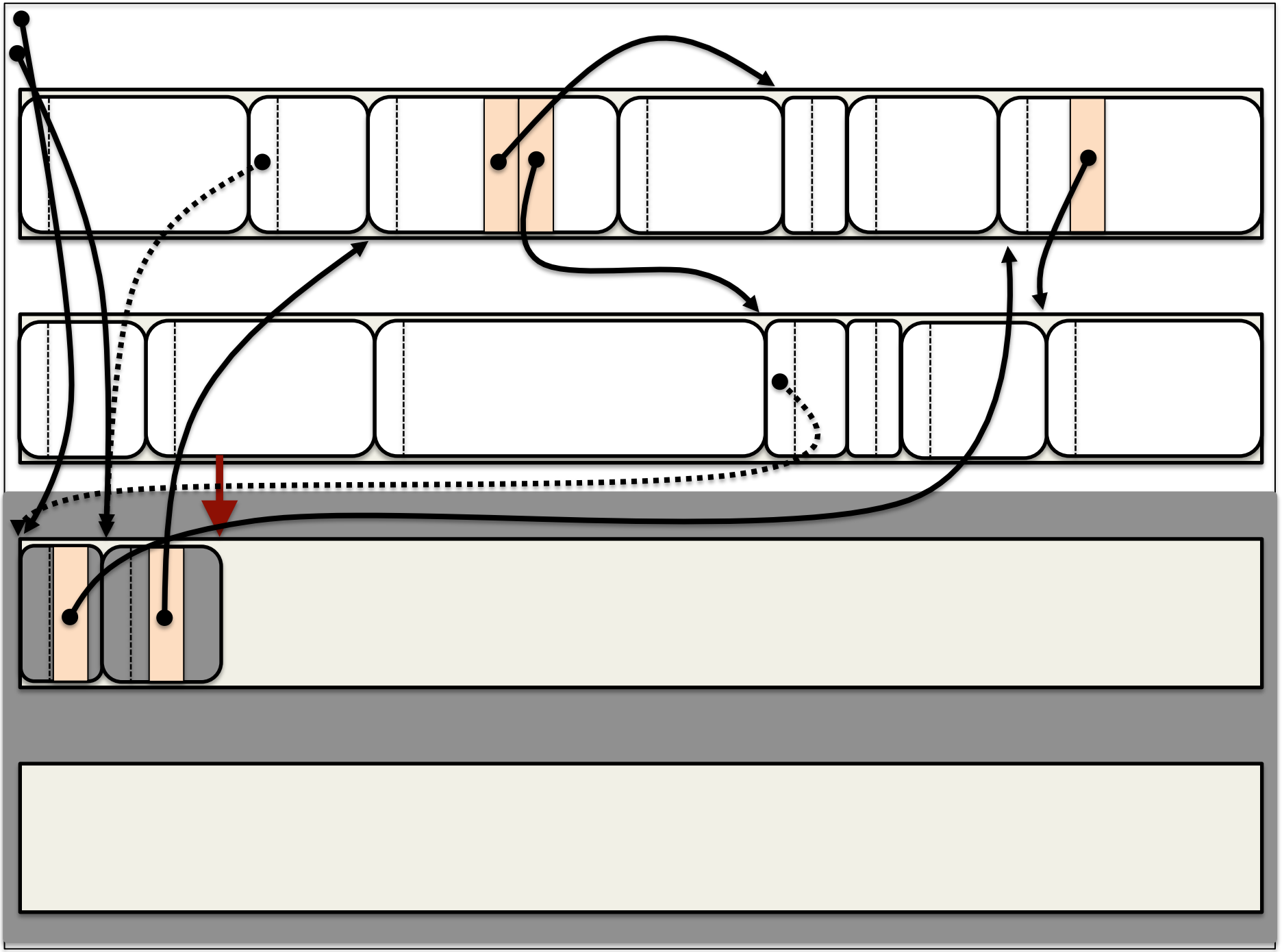


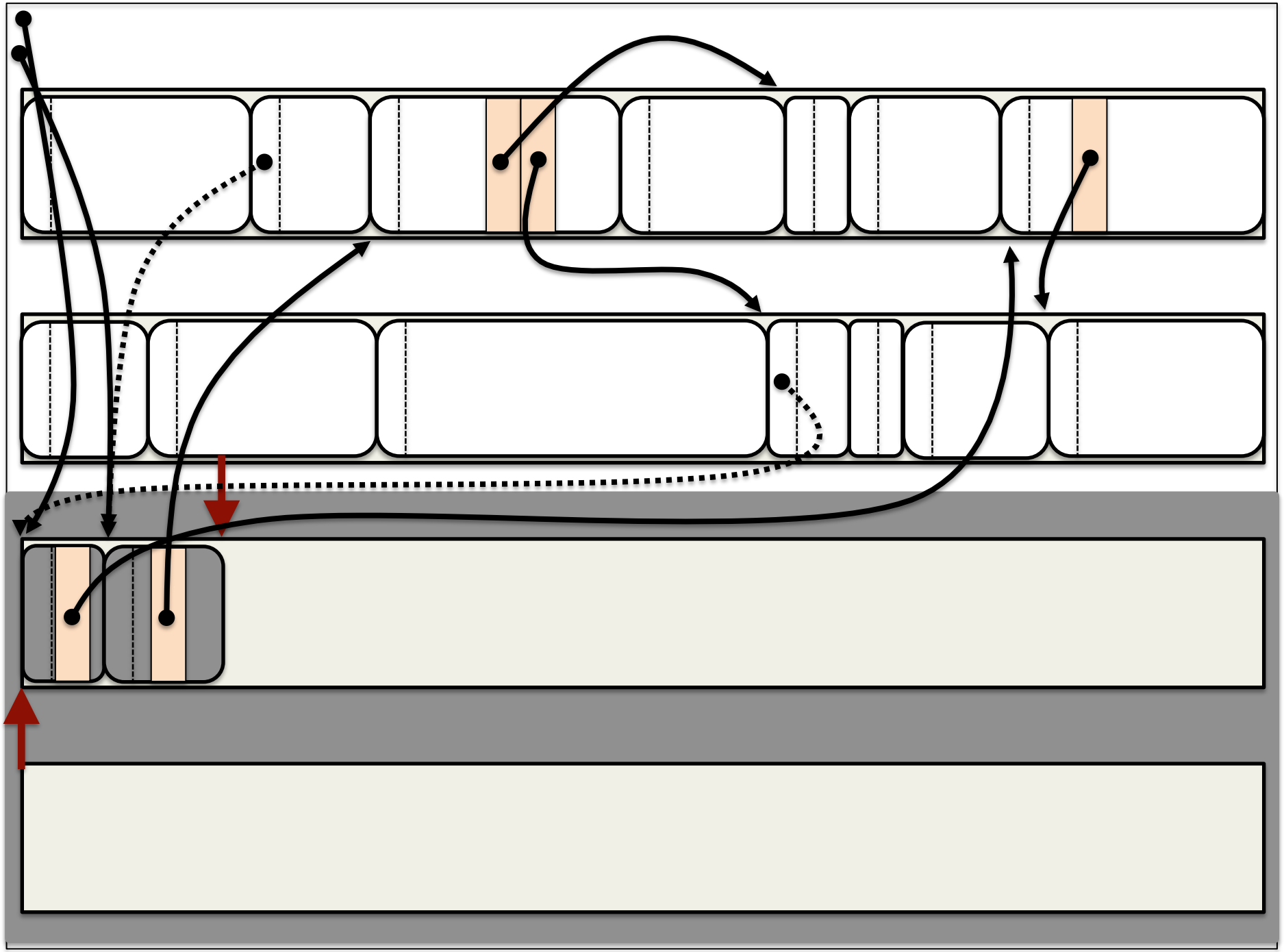


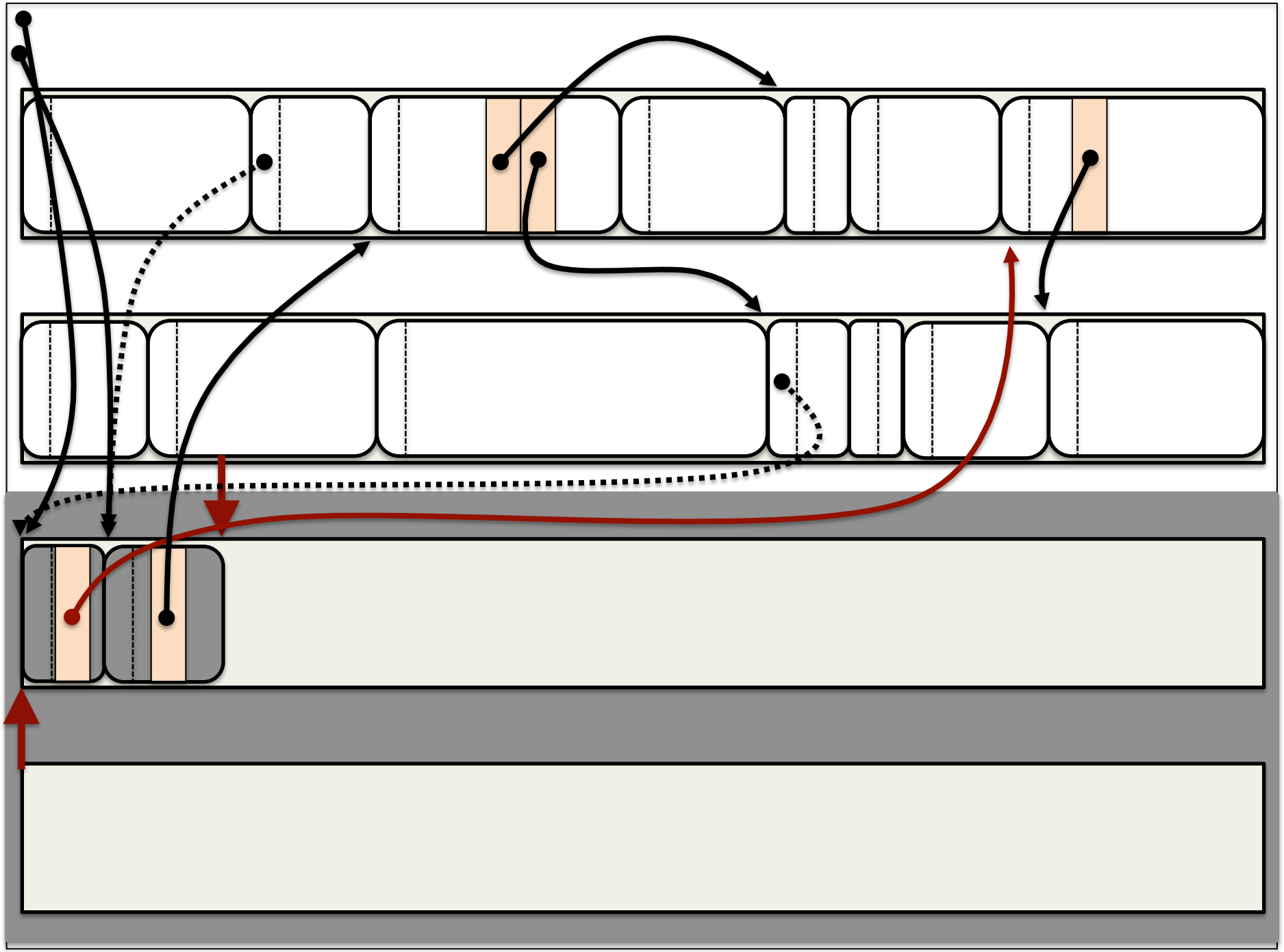


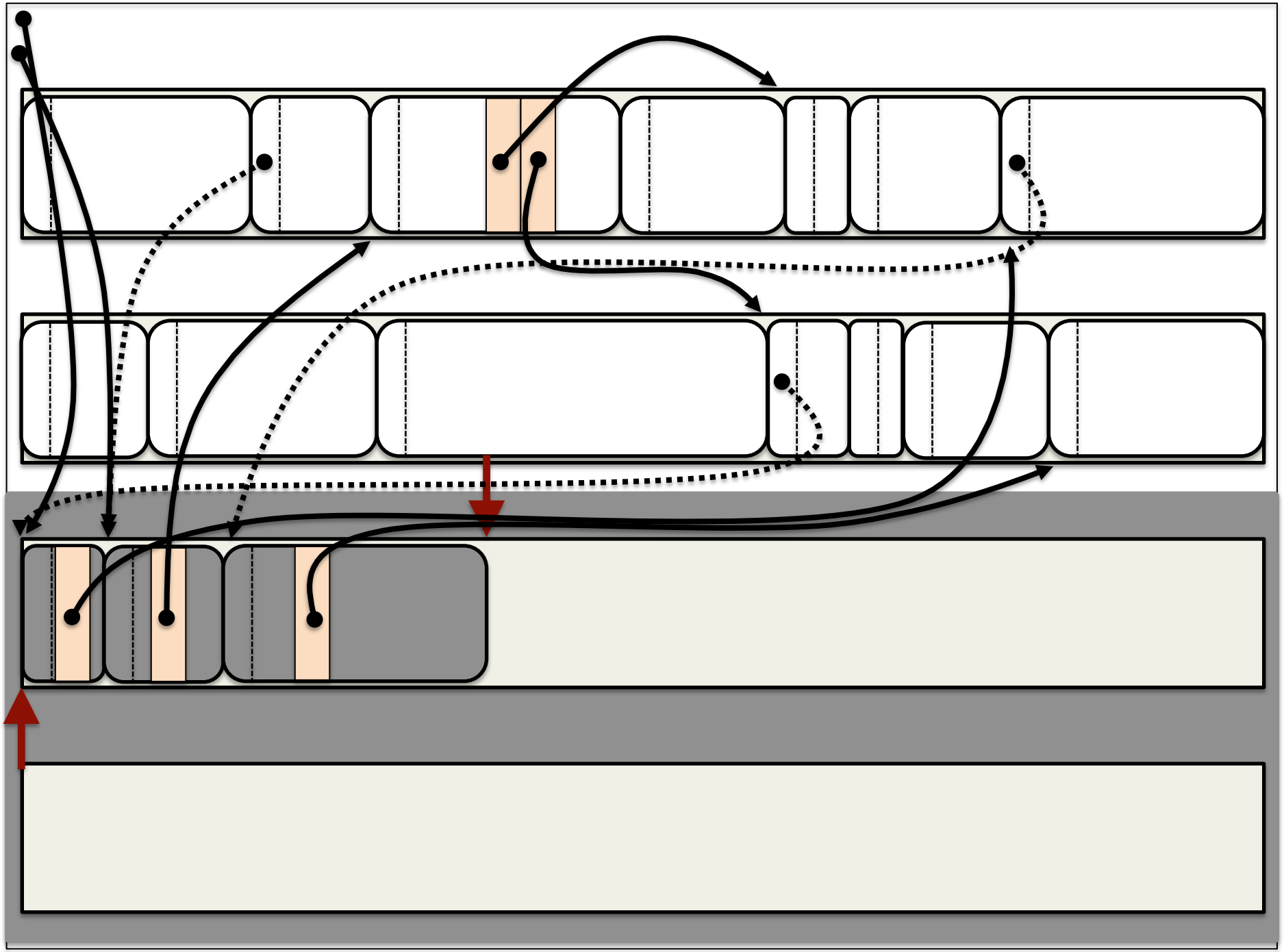
Two Finger Algorithm

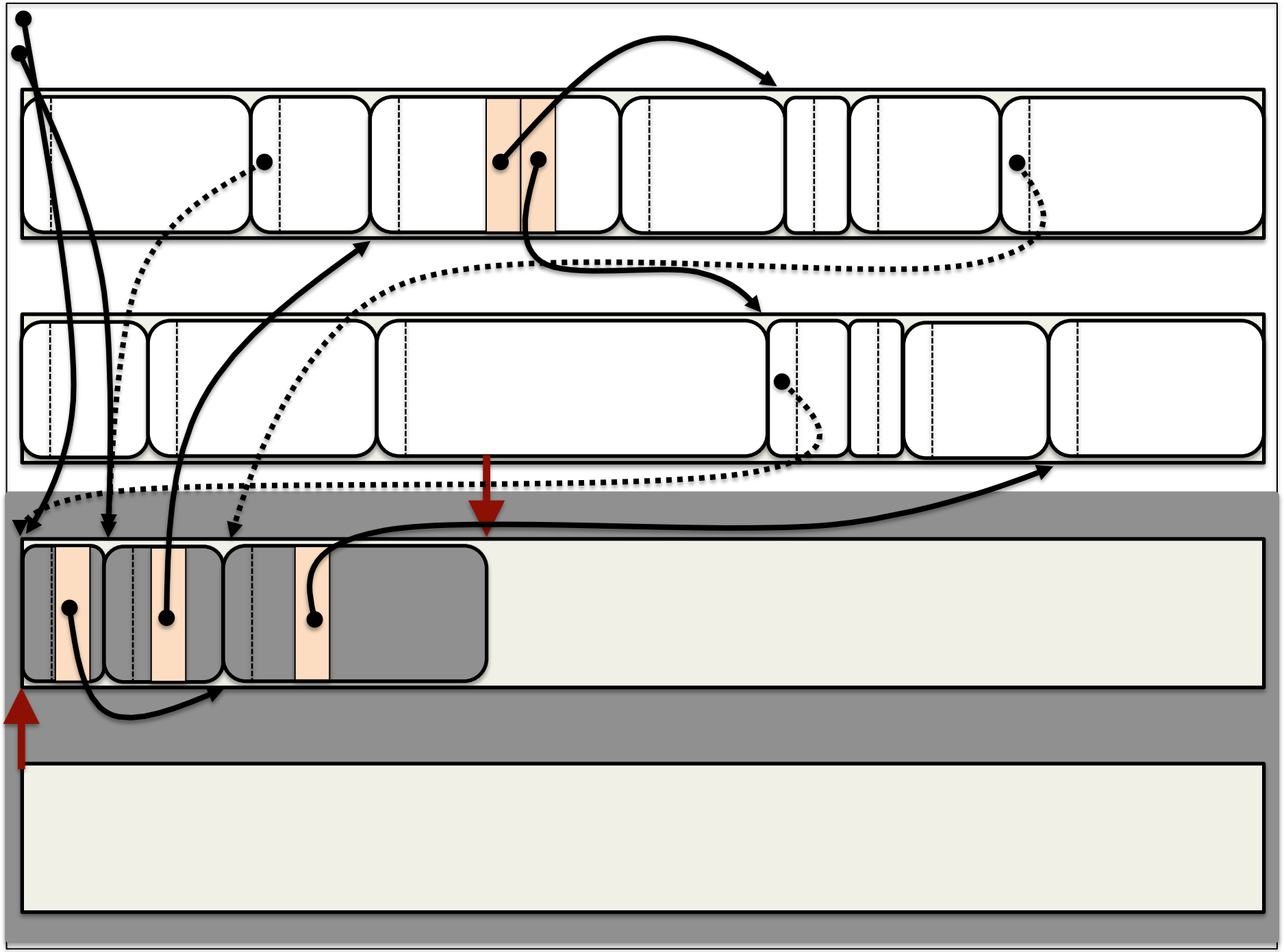
- Cheney's algorithm doesn't use a gray stack
- Objects are copied immediately
- We can use the new space as an implicit stack
  - Note that objects are allocated sequentially
  - Scan objects in the same order
- We're done when the two pointers meet

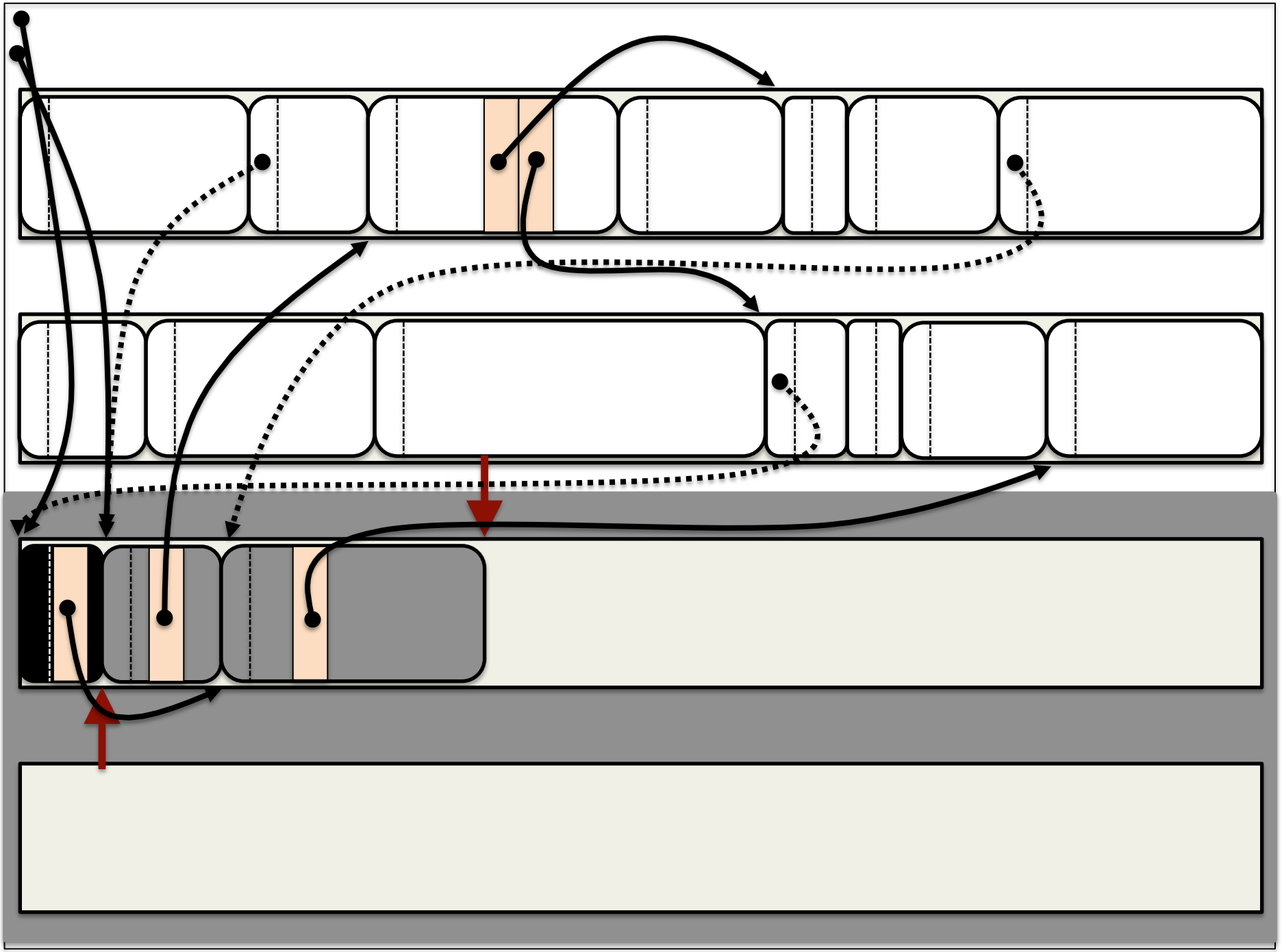


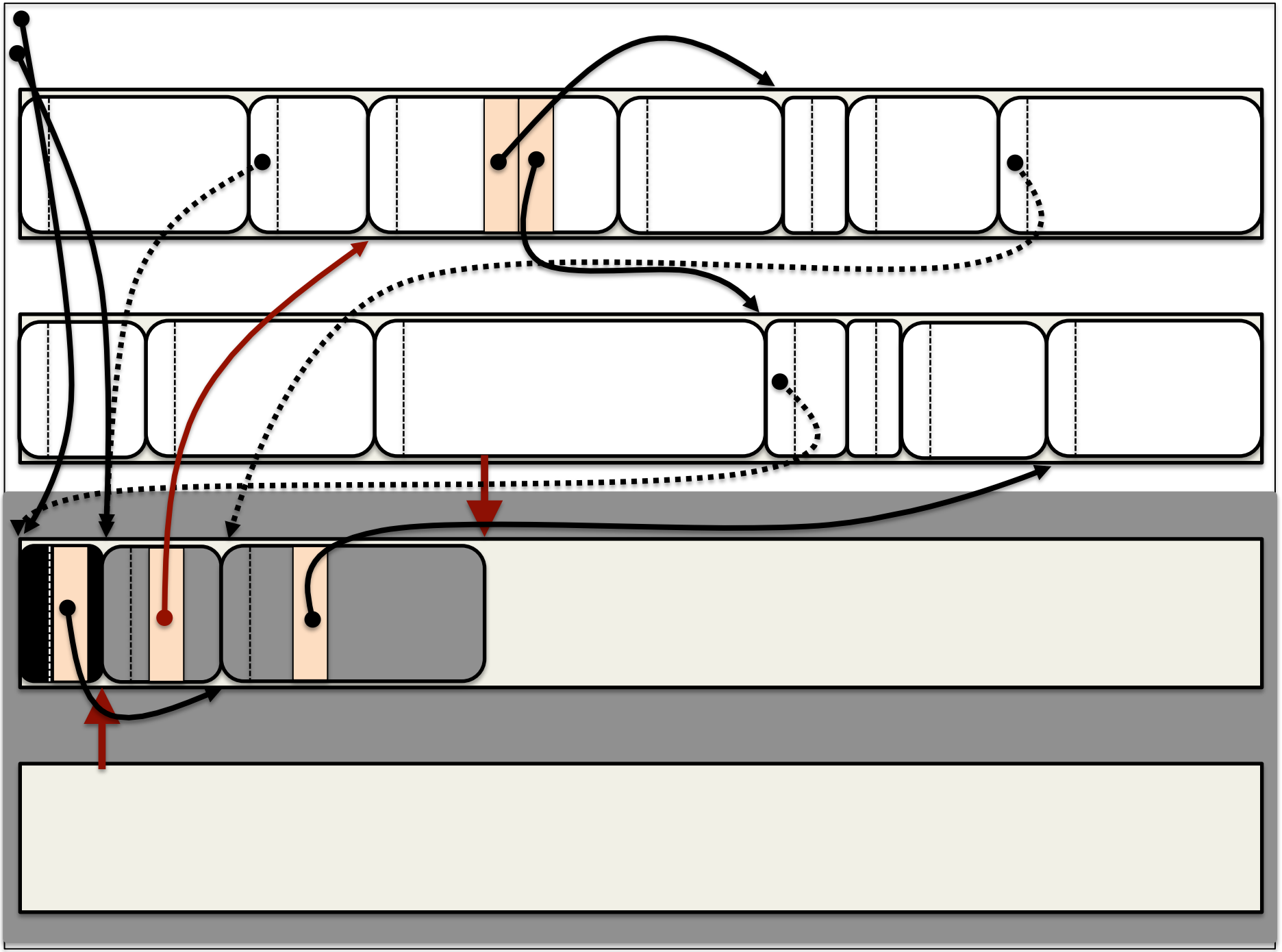


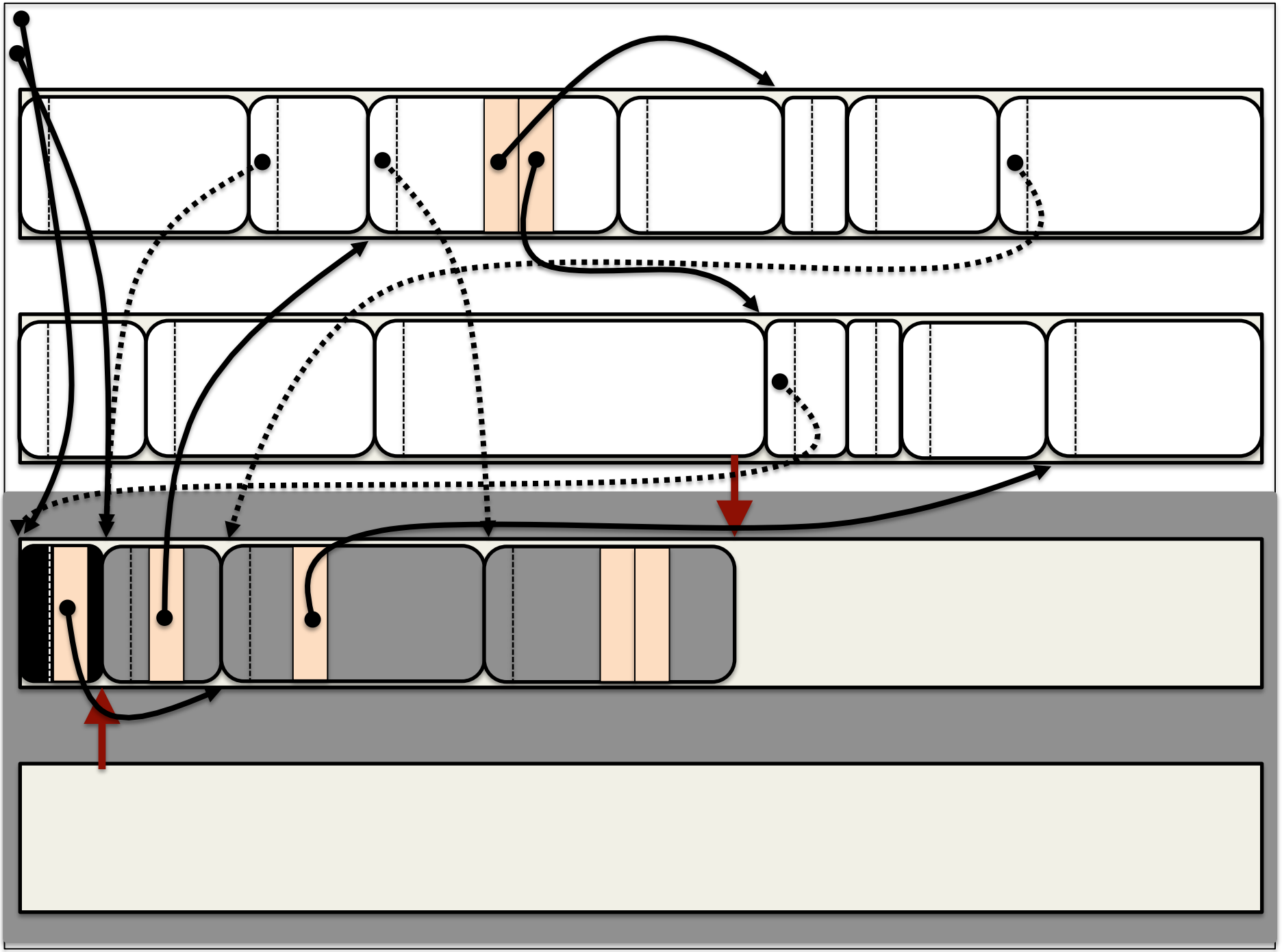


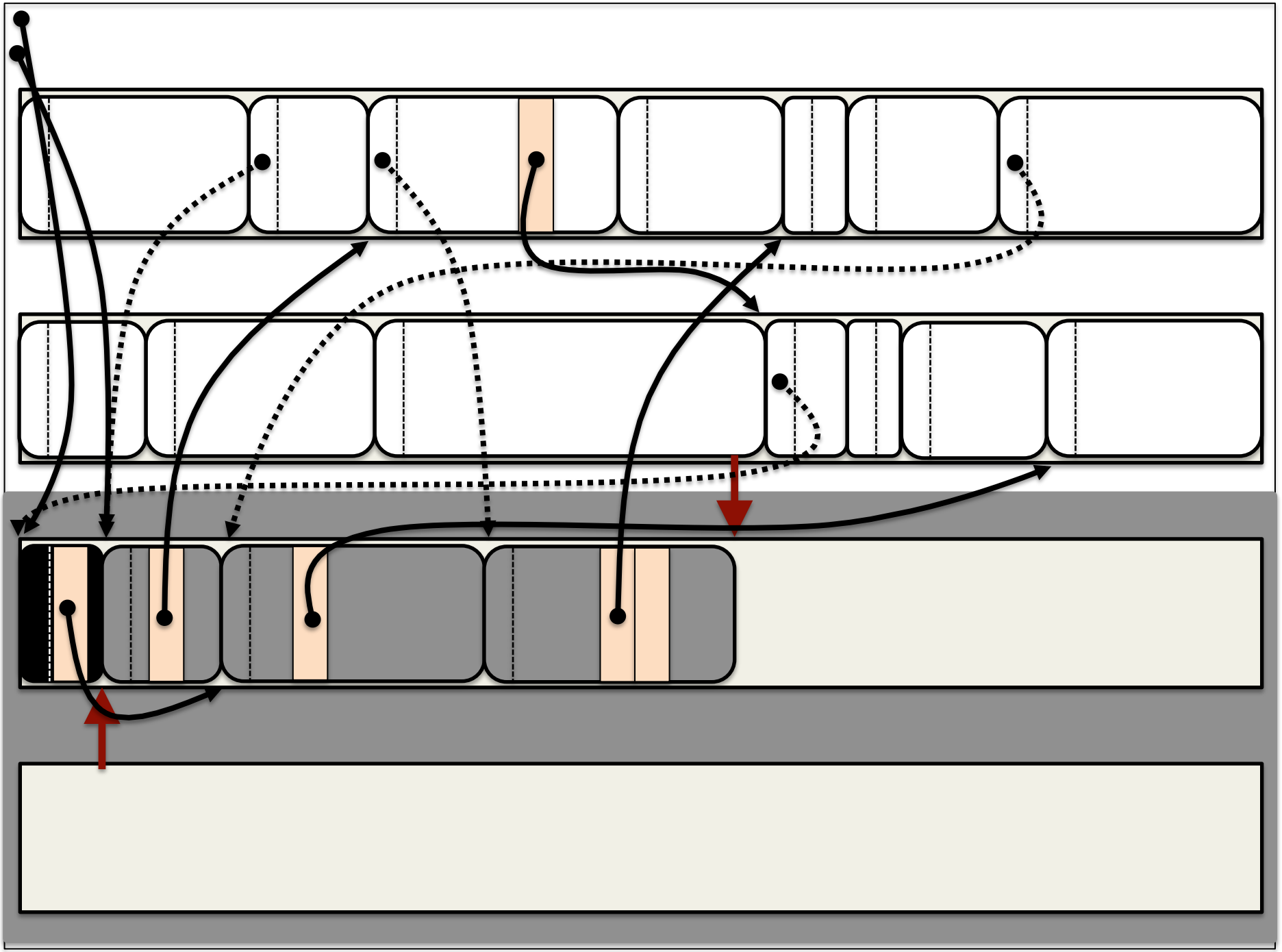


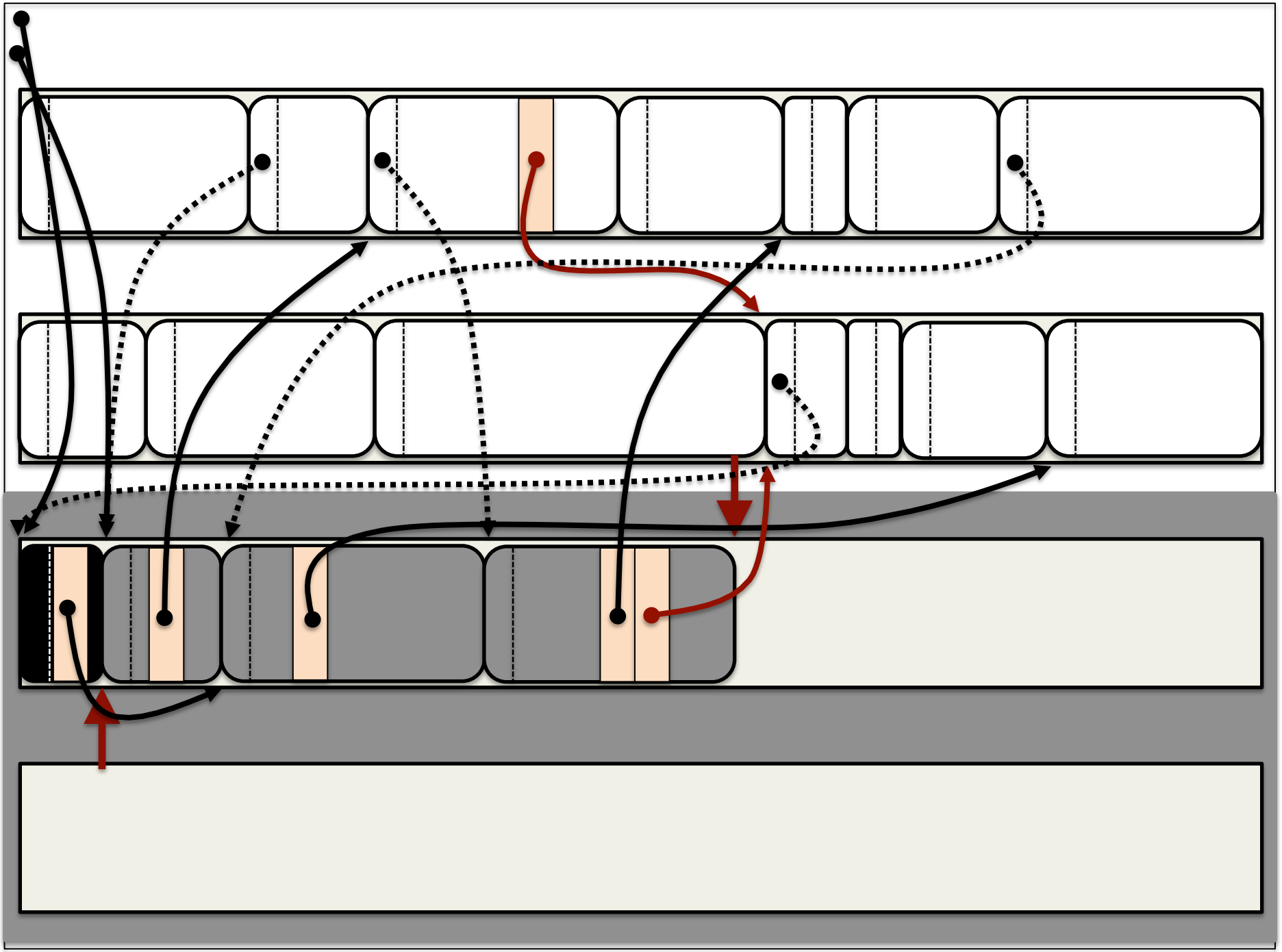


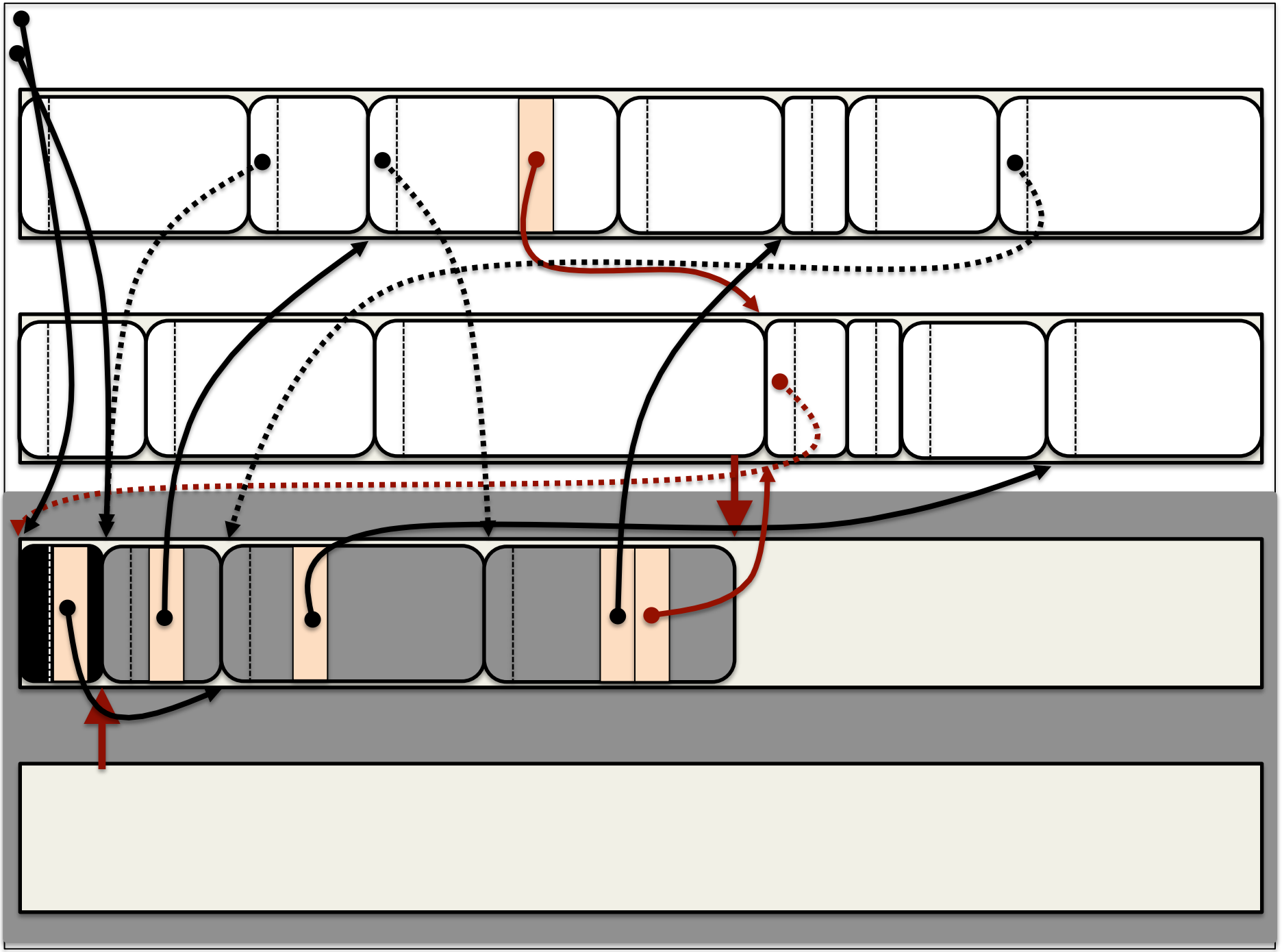


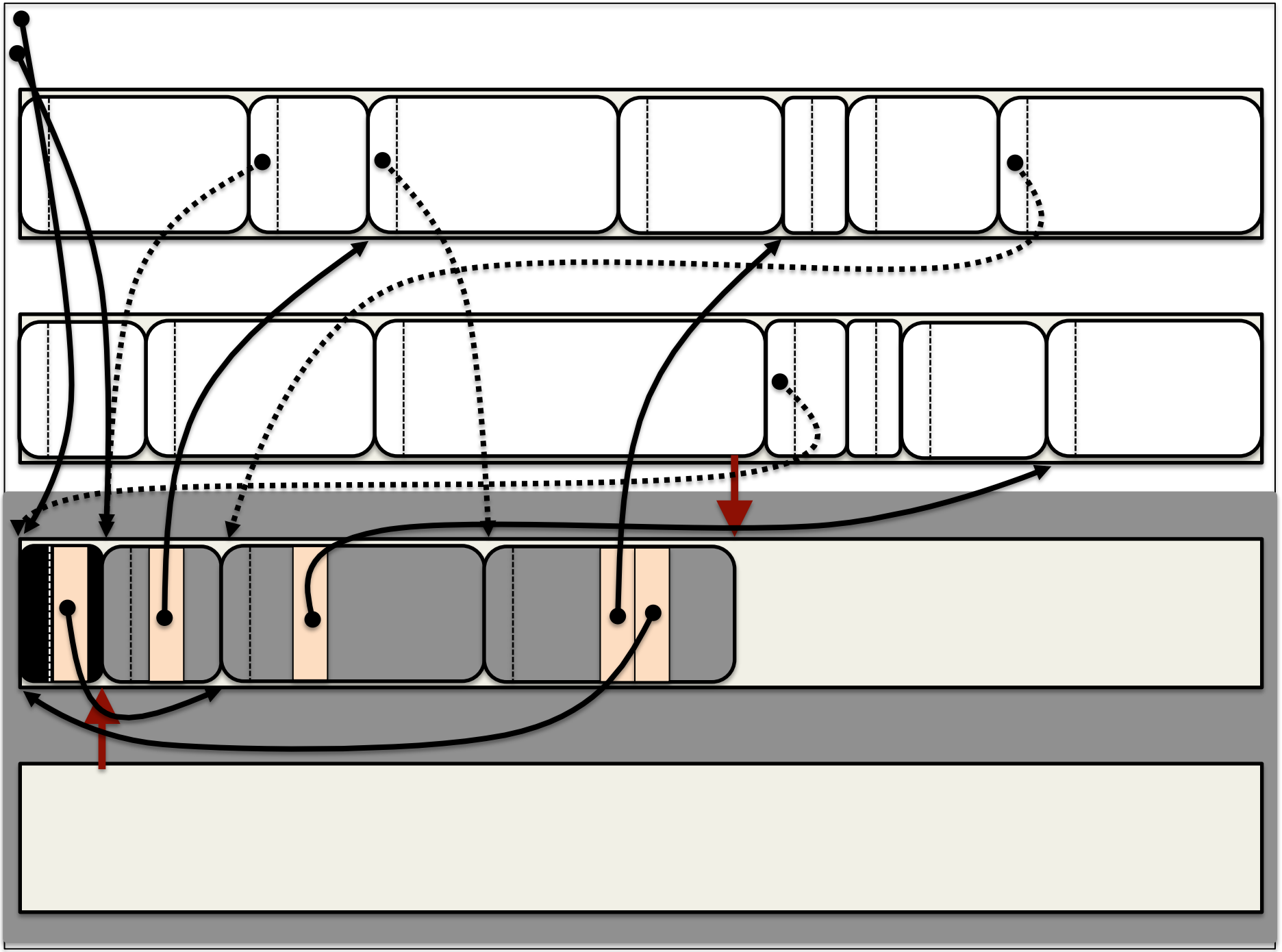


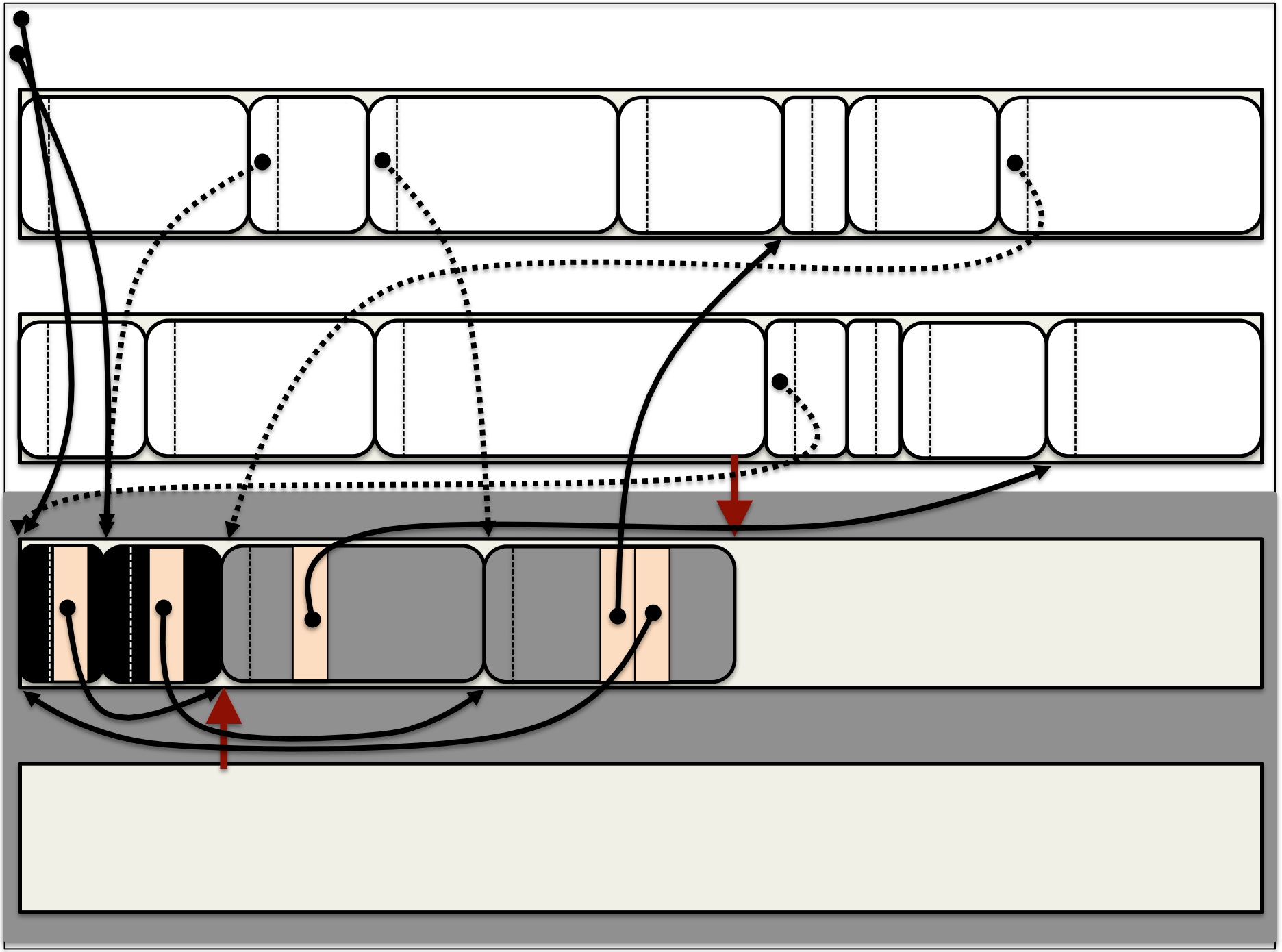


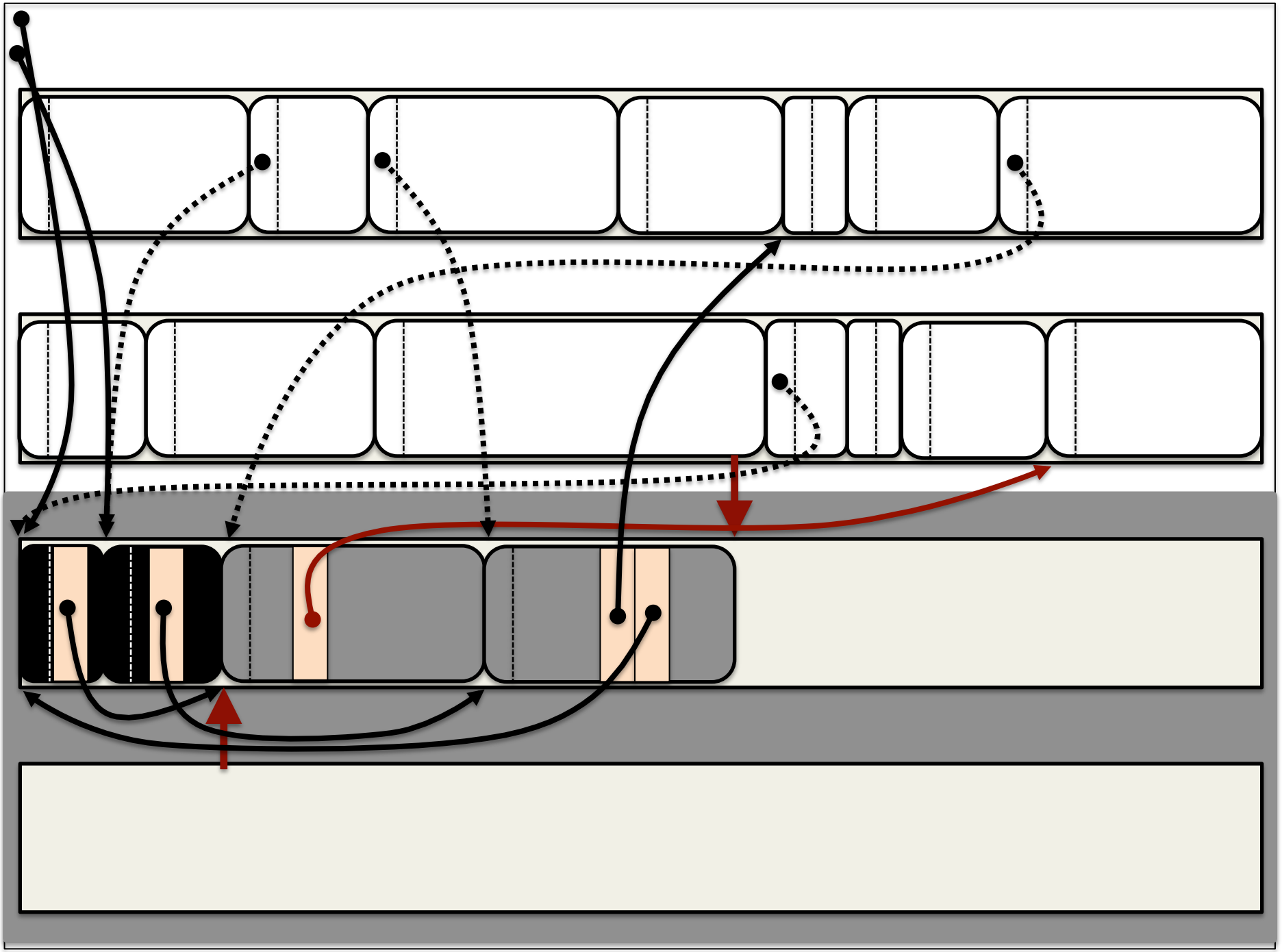


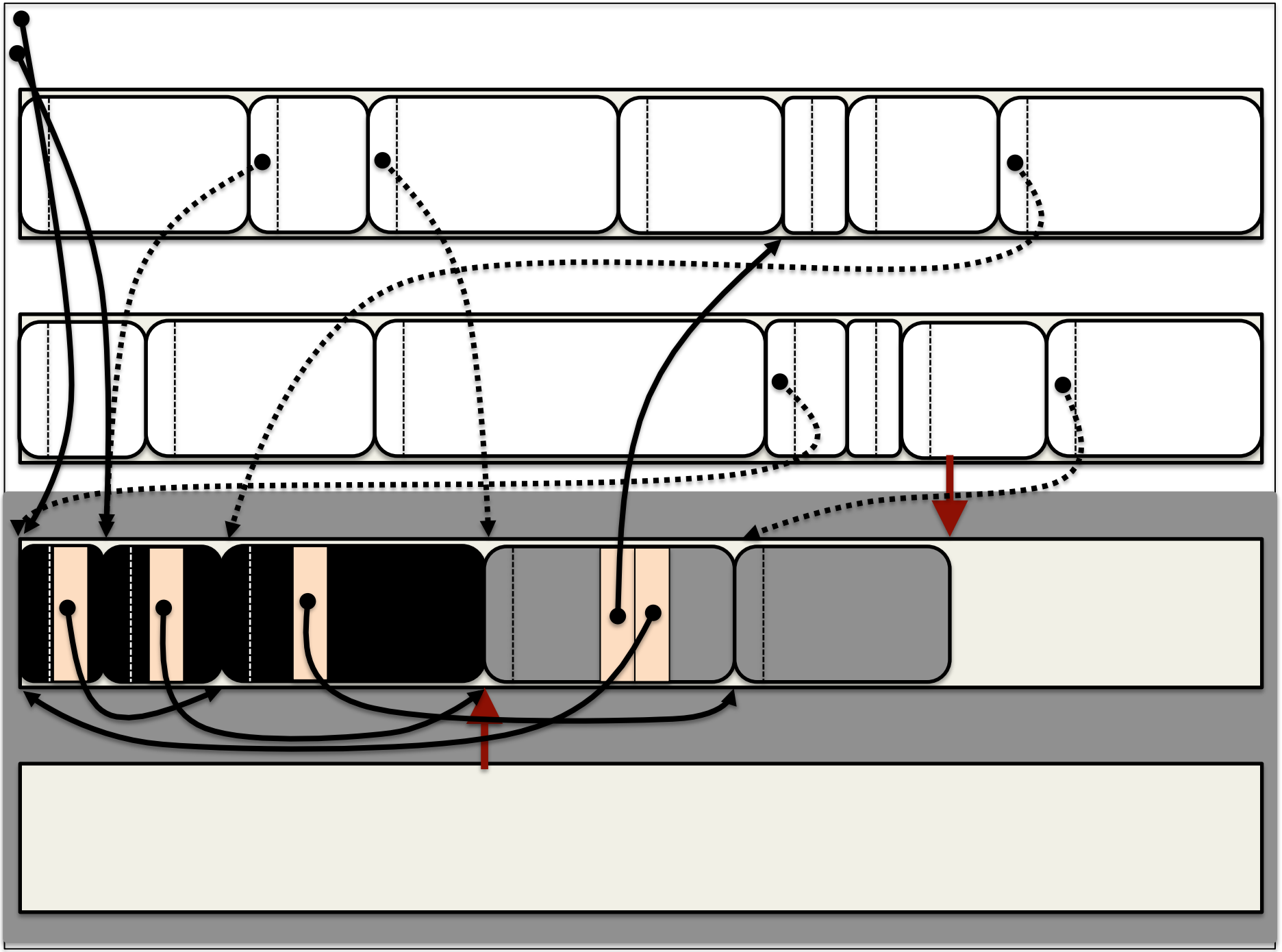










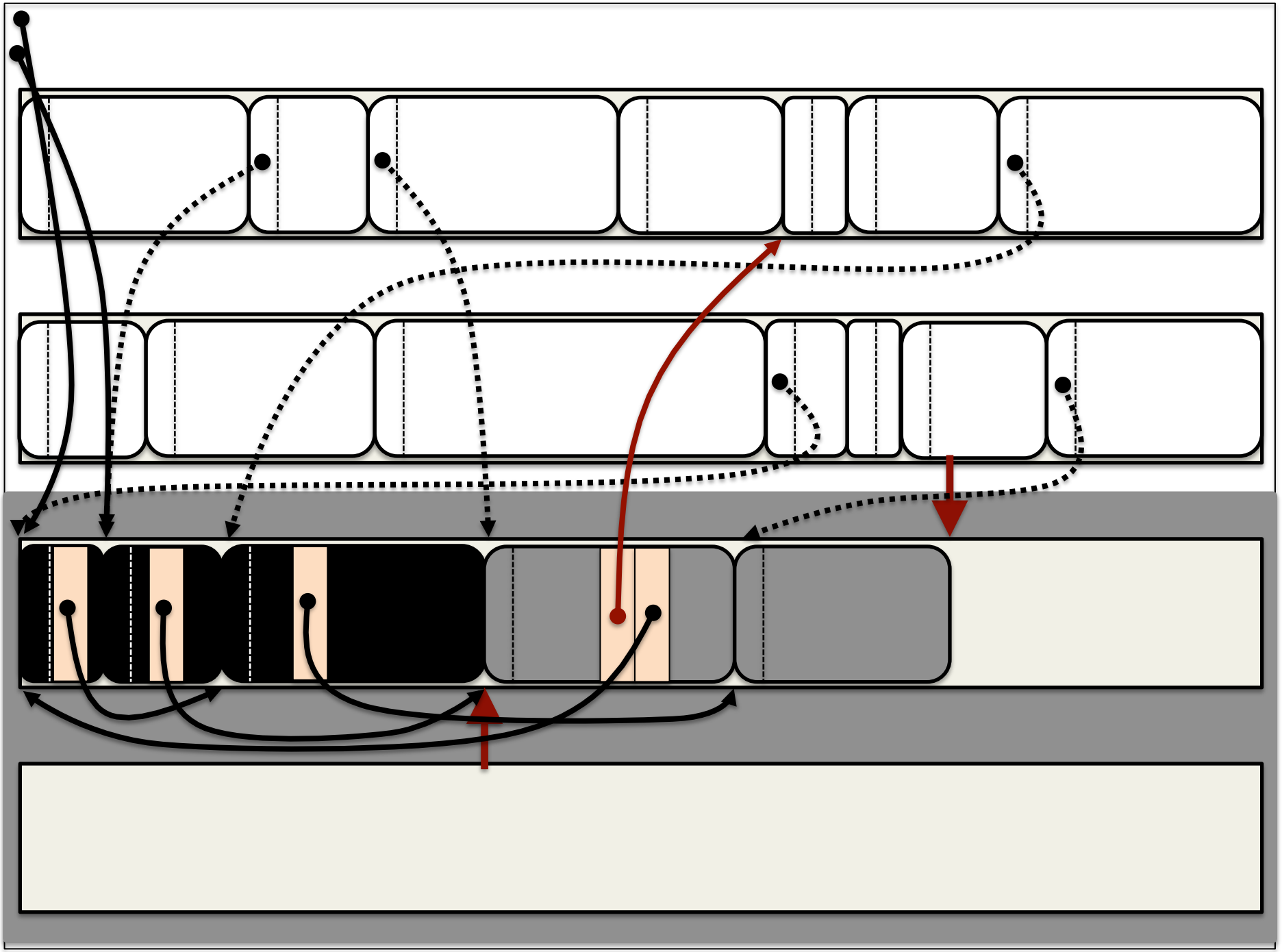


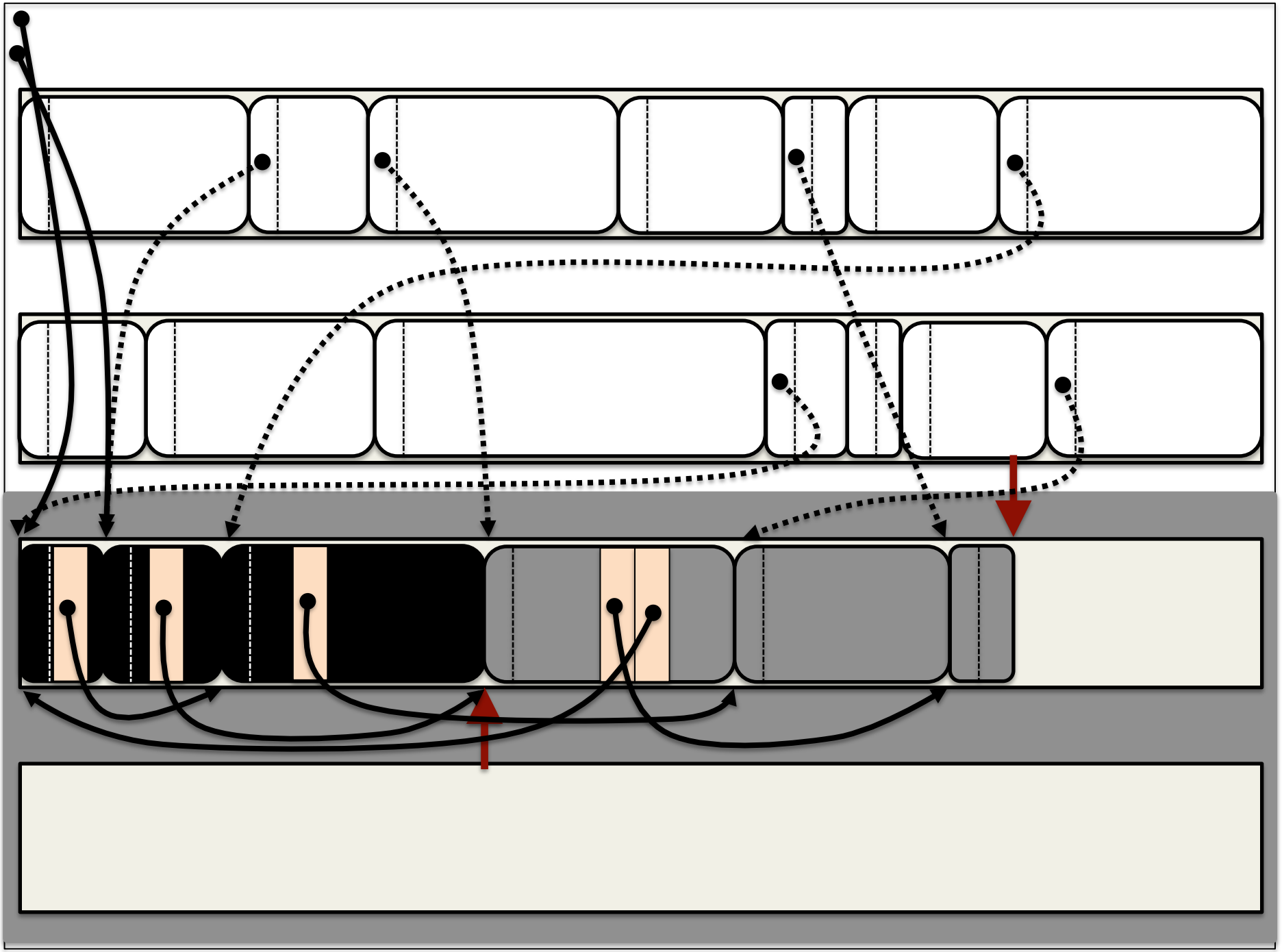
Updating References

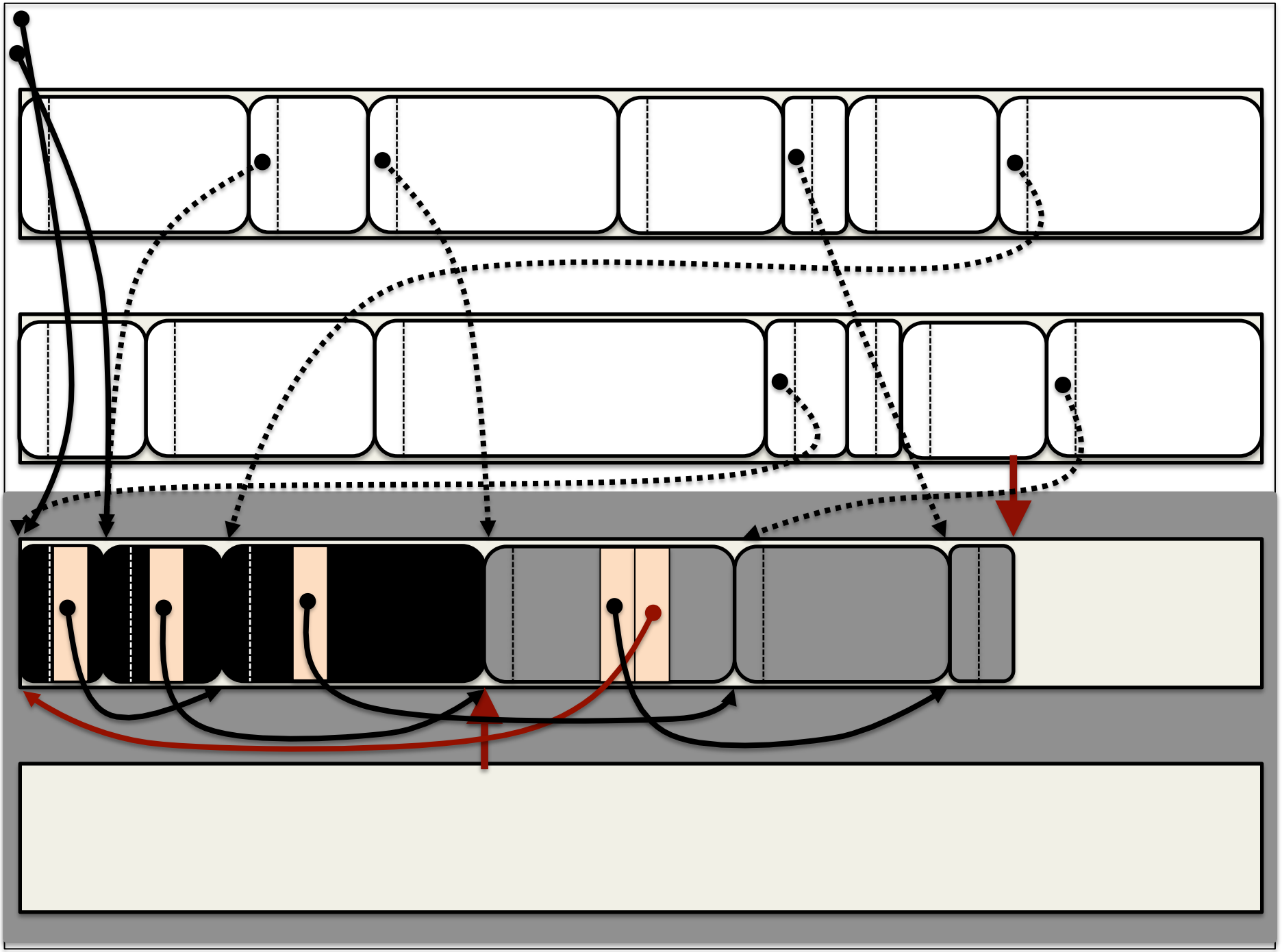
- There can be no references from new to old
  - Part of marking an object black
- Two cases when scanning an object

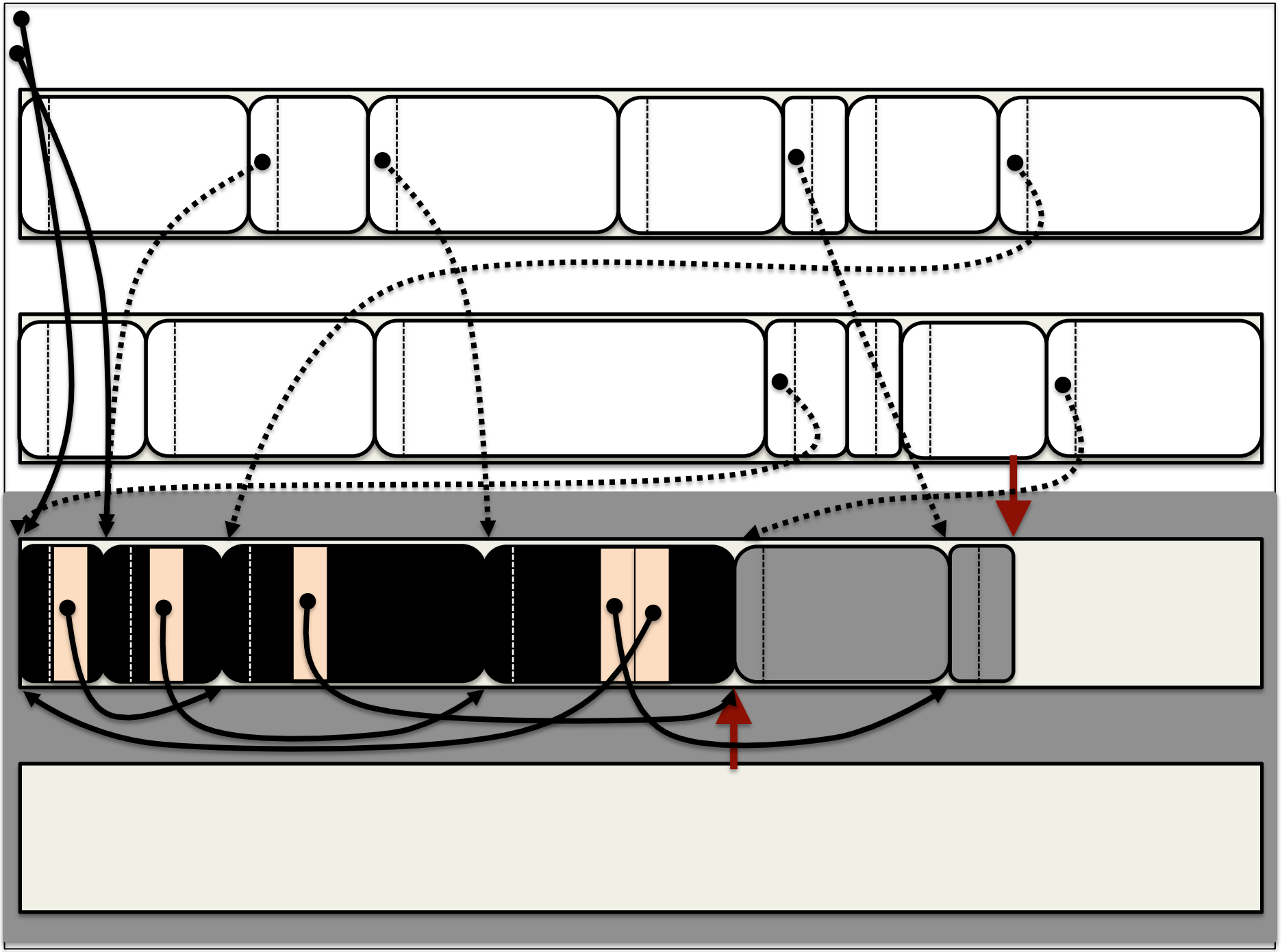
- The object has not been copied
  - Copy to the new space
  - Update the reference

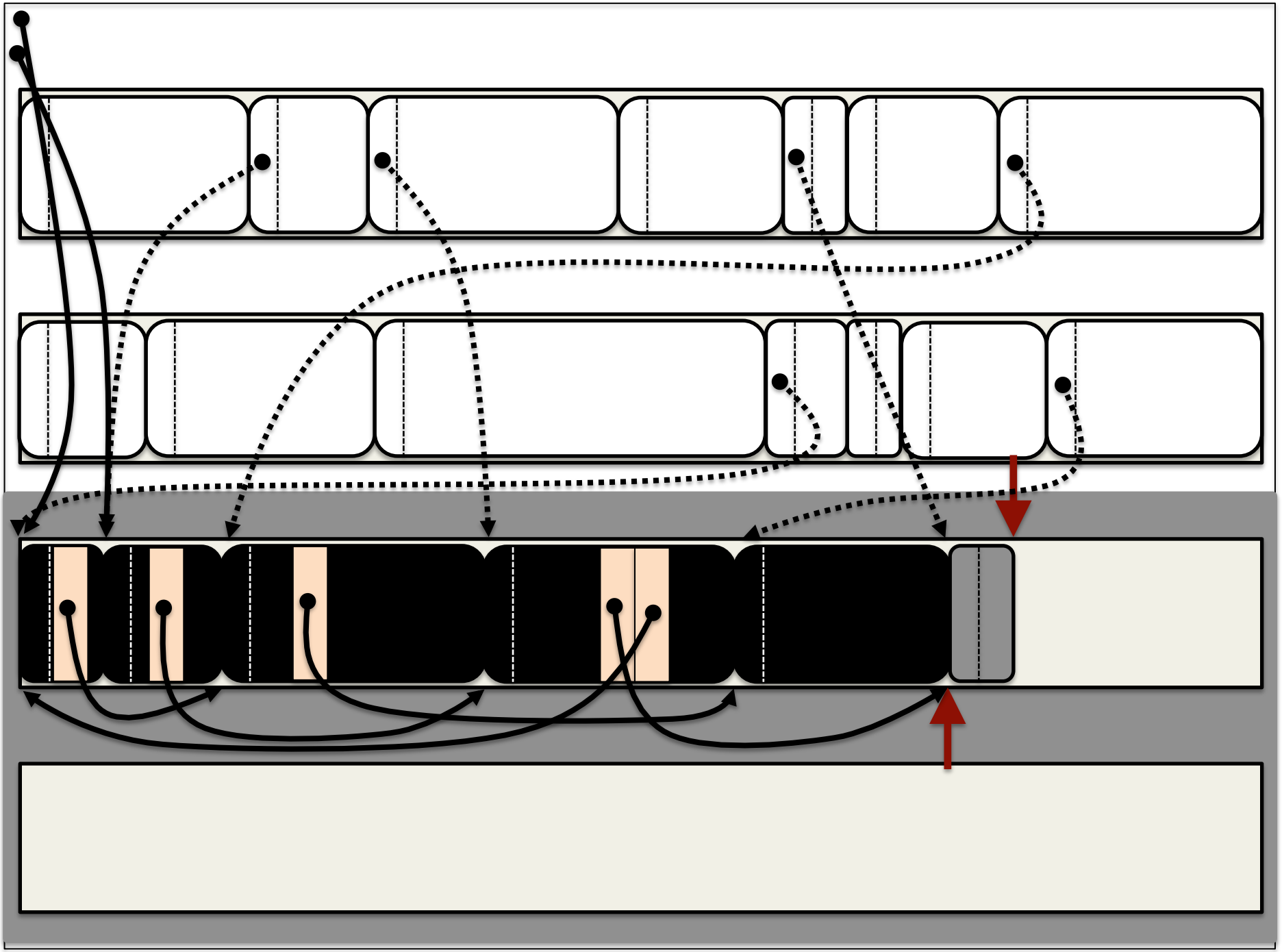
- The object has already been moved
  - Use the forwarding pointer to update the reference

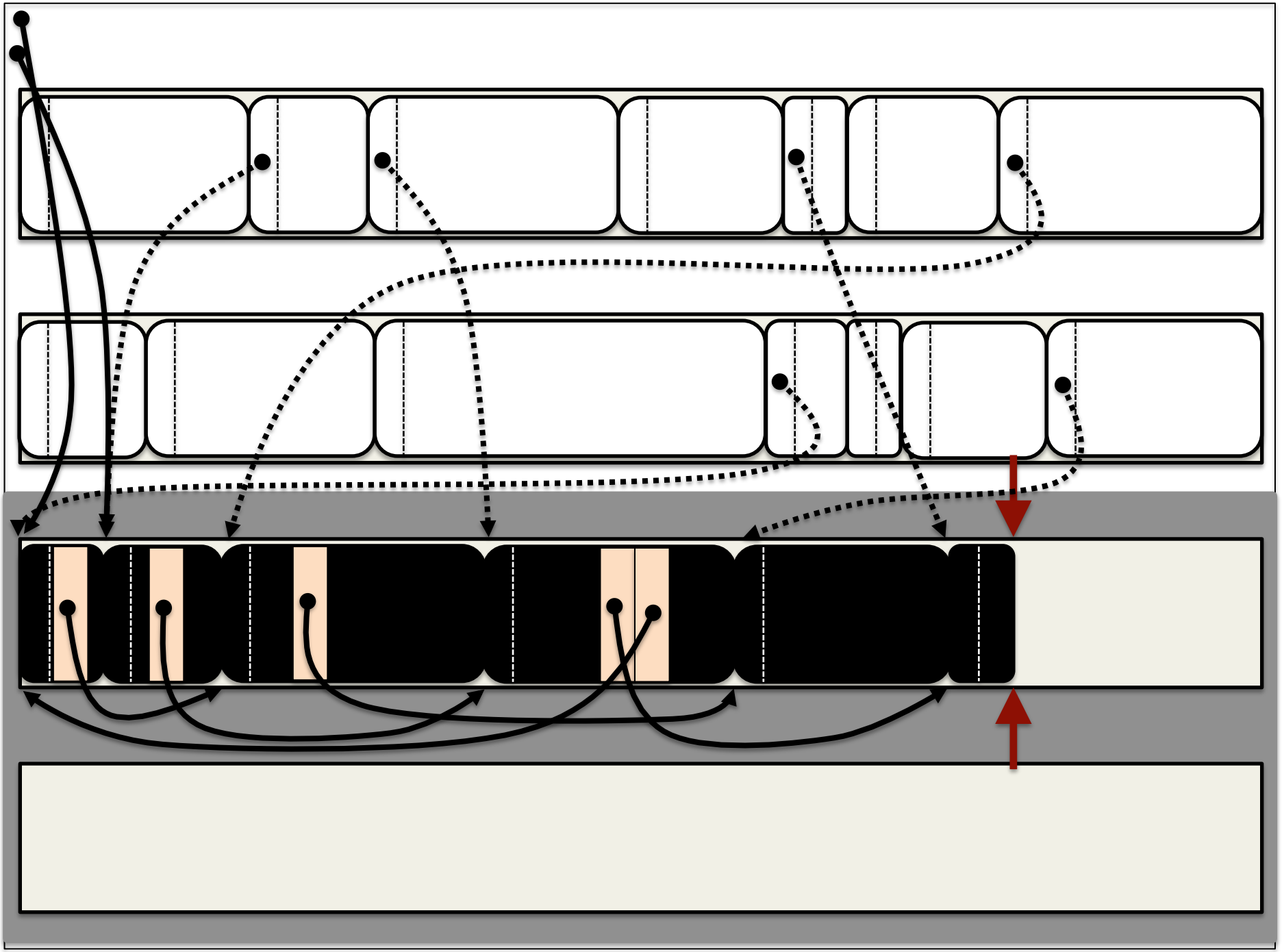












Finishing Up

- Algorithm ends when the finger pointers meet
  - All gray objects have been scanned
- No per-object finishing processes
  - New-space objects are not marked
  - All references point to the new space
  - Forwarding pointers only exist in garbage objects
- Continue allocating in the new space

Assignment 3

- Implement Cheney's algorithm
  - Bump-pointer allocation
  - Semi-Space heap layout
  - Stop the world thread management
  - Single pass mark and copy
- Implement the `MemoryManager` interface
  - Implementation details are up to you
  - Following the algorithm correctly matters
  - Grading will look at the state of the VM after GC

SimpleJava Memory Management

- Interfaces under `vm/memory`
  - `Heap`
  - `MemoryManager`
  - `Region`
- Implementations in `impl/memory`
  - Implementations of heap tracking
  - Two possible memory managers

Allocation

- Allocation handled by the `Region`
  - Currently supports `BumpPointerAllocator`
- Look at the existing allocation code
  - `InfiniteMemoryManager.java`
- Look at `BumpPointerAllocator.java`
  - You shouldn't need to change the implementation
  - Could help while debugging

Heap Class

- Java doesn't give direct pointer access
  - You now know why
- A memory manager needs to access pointers

- In SimpleJava we simulate the heap
  - In production we could replace with raw memory access
  - Remember JIT intrinsics

- The heap is a logical construct
  - Maintains a mapping of pointers to objects

Heap Layout

- The heap is divided into regions
  - Areas of memory to manage independently
- At one extreme there is a single region
  - `InfiniteMemoryManager`
- At the other there can be many
  - We'll talk about this next week
- For your work there will be two

Heap Operations

- Carve off a region from the heap
  - Current implementation assumes bump pointer
  - This may change for the final project

- `memcpy`
  - Copy an object from one location to another
  - Makes an exact duplicate of the object

Object Builder

- Creates correctly structured objects
  - Look at the code in `NewInst` and `NewArray`
- Handles the heap management
- Only way that you should create new objects
  - You may copy objects with `Heap.memcpy`

Roots

- Thread stack and local variables
  - Accessed via `VmThread` interface
- Static fields
  - Accessed via `VmClassLoader`
- String intern table
  - Accessed via `ObjectBuilder`

Pointers to Pointers

- Need to update root locations
- Two options
  - Trace through stack, local, static and intern table
  - Use pointers to pointers
- Look at the `ReferenceLocation` interface

OutOfMemoryError

- A GC-based system can still run out of memory
  - All objects survive the collection
  - Not enough space freed up for an allocation request
- The VM spec calls for an `OutOfMemoryError`
  - SimpleJava doesn't yet throw errors internally
  - Exception handling will be in Assignment 4
- Throw a VmInternalError
  - Terminates the VM

Writing a Garbage Collector

- Notoriously tricky to debug
  - Errors show up long after the bug
  - Bugs can be non-deterministic
- Better to focus on getting it right than fixing later
  - Think of the GC as a graph algorithm
  - Make sure that you modify the graph correctly
- Test and verify

Heap Verification

- Unit tests let you examine the heap closely
  - Don't need to set up mutators to do the right thing
  - It's how I will grade the assignment
- Consider adding a verification phase
- Check at the end of GC for invariants
  - No pointers to the old space
  - Re-do trace and count reachable objects
  - All roots properly updated