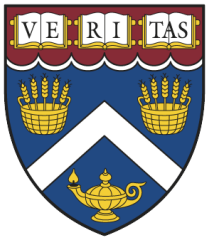
In the last week

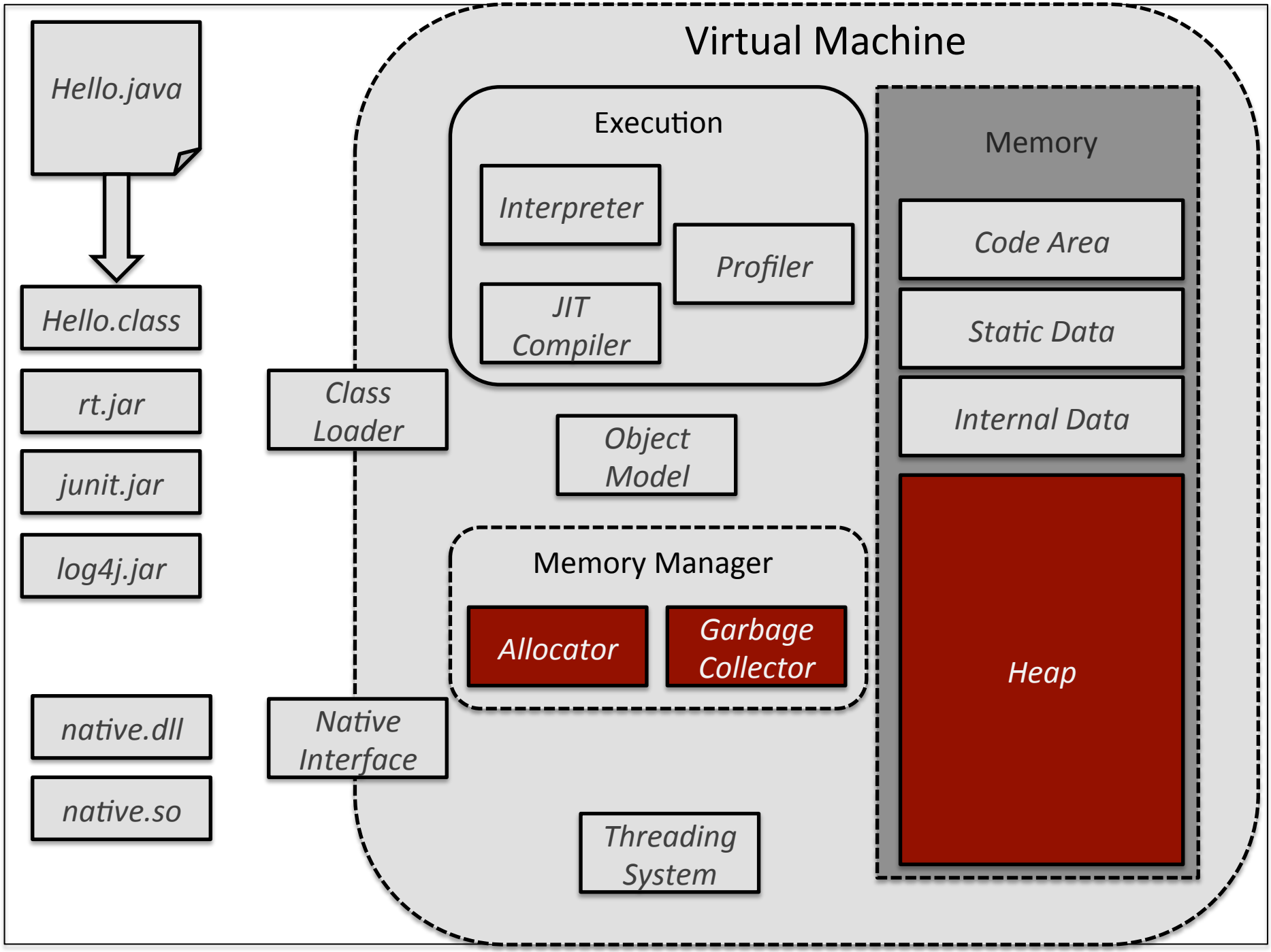
- Assignment 2 grades released
- Assignment 3 underway
  - Question about the heap and region sizes
  - Why isn't there a garbageCollect() method?

Forwarding Pointers

- Part of Assignment 3 involves relocating objects
  - Which means installing forwarding pointers
- Some confusion as to where to store the pointer
  - In C/C++ you would overwrite a header word
  - Set a bit to indicate that you'd installed the pointer
- Since we're using Java we can't set bits
  - Instead just store a `HeapPointer` to the object header
  - Use the `TypeDescriptor` word
  - Use `instanceof` to check if the word contains a pointer

Generational Garbage Collection




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

Principle of Locality

- Temporal locality
  - Once you use a value, you will likely use it again soon
  - Afterwards it may be a long time before using it again
- Spatial locality
  - Nearby objects tend to be used together
  - Extends to objects that reference one another

Copying Collection

- Take advantage of dynamic spatial locality
  - Breadth-first scan of the heap
  - Move objects based on their connectivity
- Costs vary based on live, not garbage, objects
  - Copy live objects out of the old space
  - Collect everything else en-masse

Copying Overheads

- Copy time overhead
  - Every live object must be copied at every GC
  - Long-lived objects carry the biggest overhead
    - Can also be the largest objects
- Copy reserve overhead
  - Semi space collector requires half the heap
  - Often that space is completely unused

Room For Improvement

- Copy objects as few times as possible
- Copy smaller areas of the heap
  - Copy reserve proportional to data being copied
- GC spends a lot of time tracing older objects
- Target objects that are more likely to be garbage
  - More garbage means fewer objects to copy

Generational Hypotheses

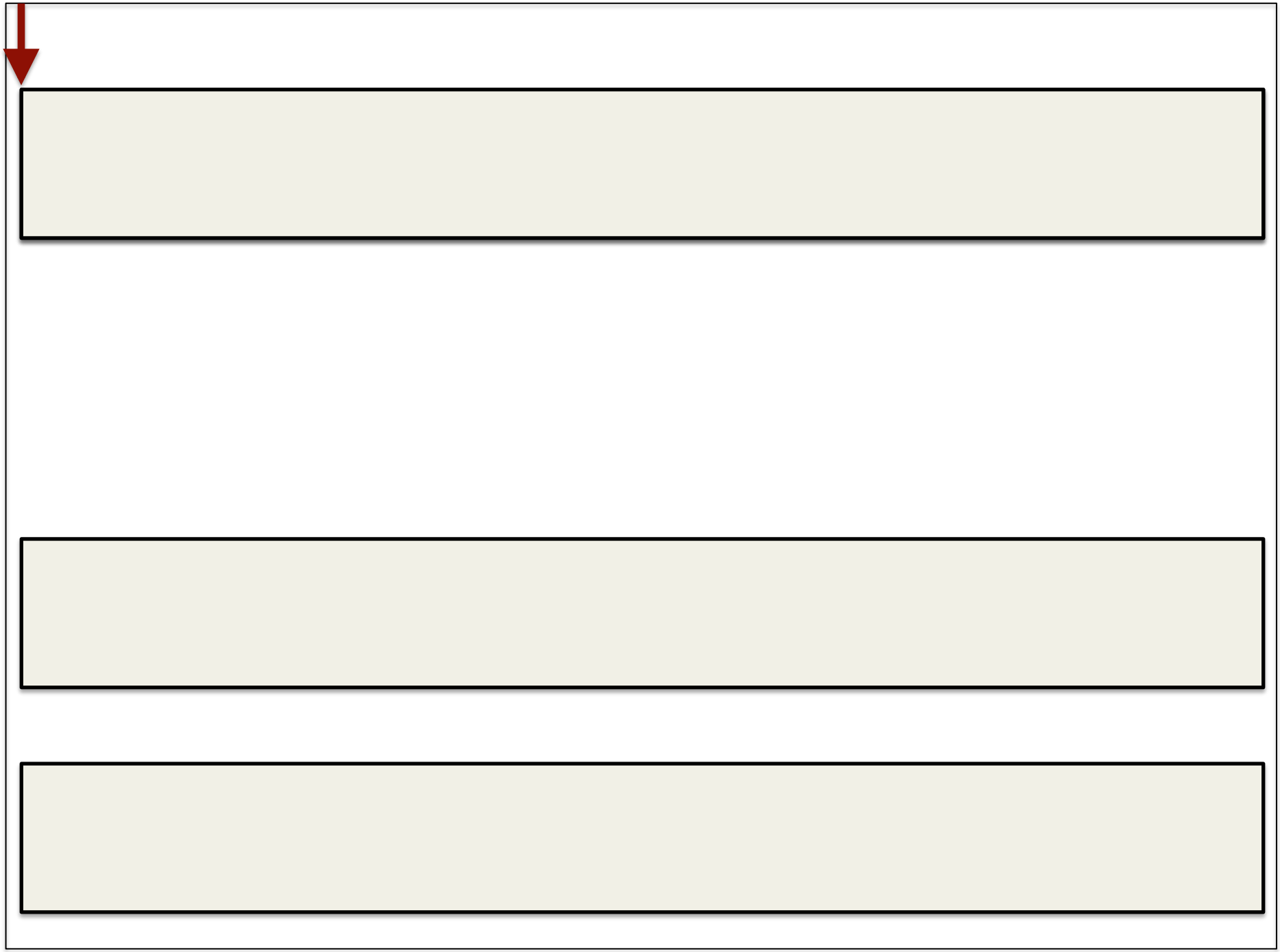
- Heuristics that suggest what is likely to be garbage
- Weak Generational Hypothesis
  - Most object become garbage shortly after allocation
  - Also known as high infant mortality
- Strong Generational Hypothesis
  - Older objects are more likely to survive a long time
  - Less well supported

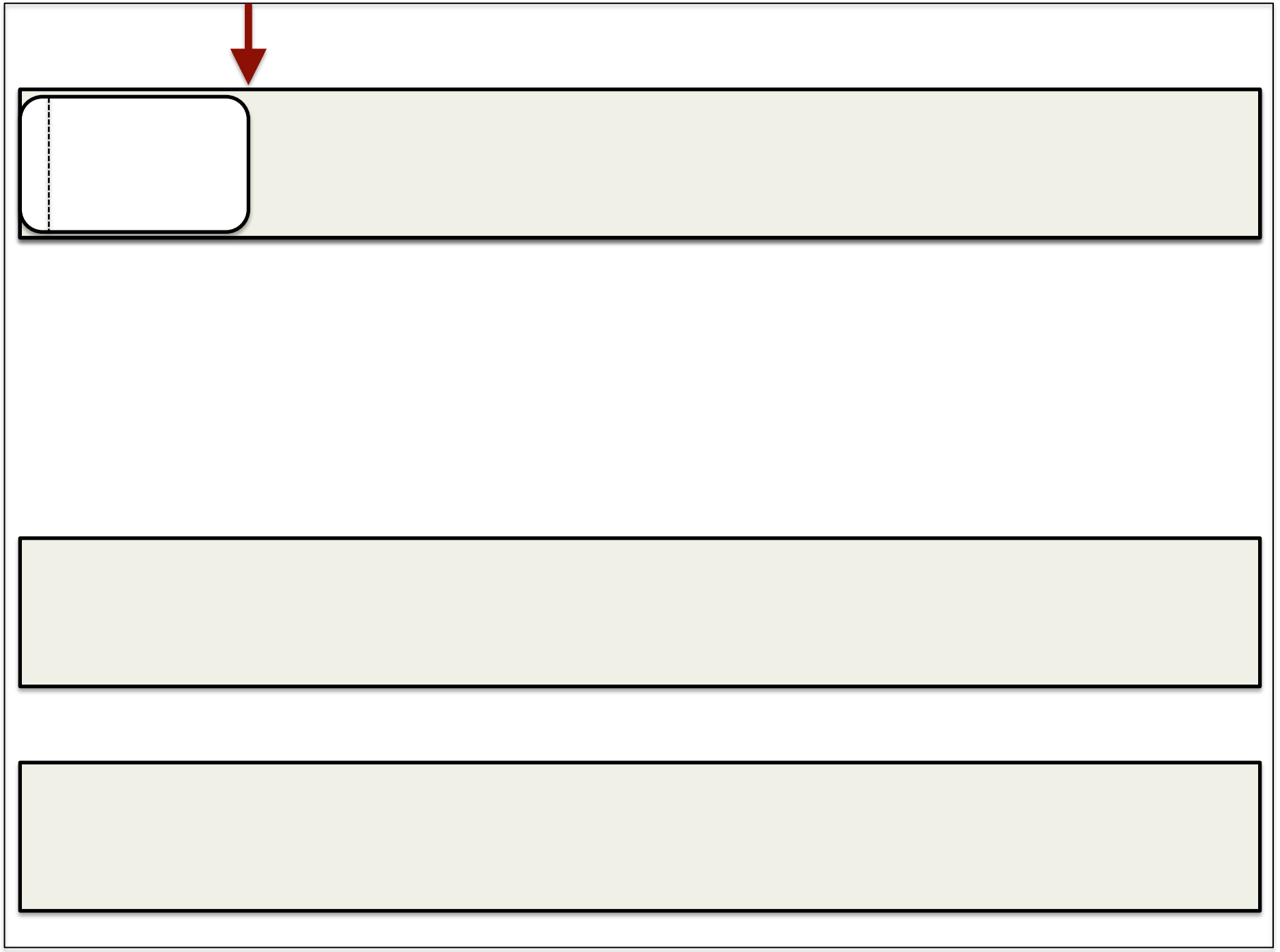
Generational Hypotheses

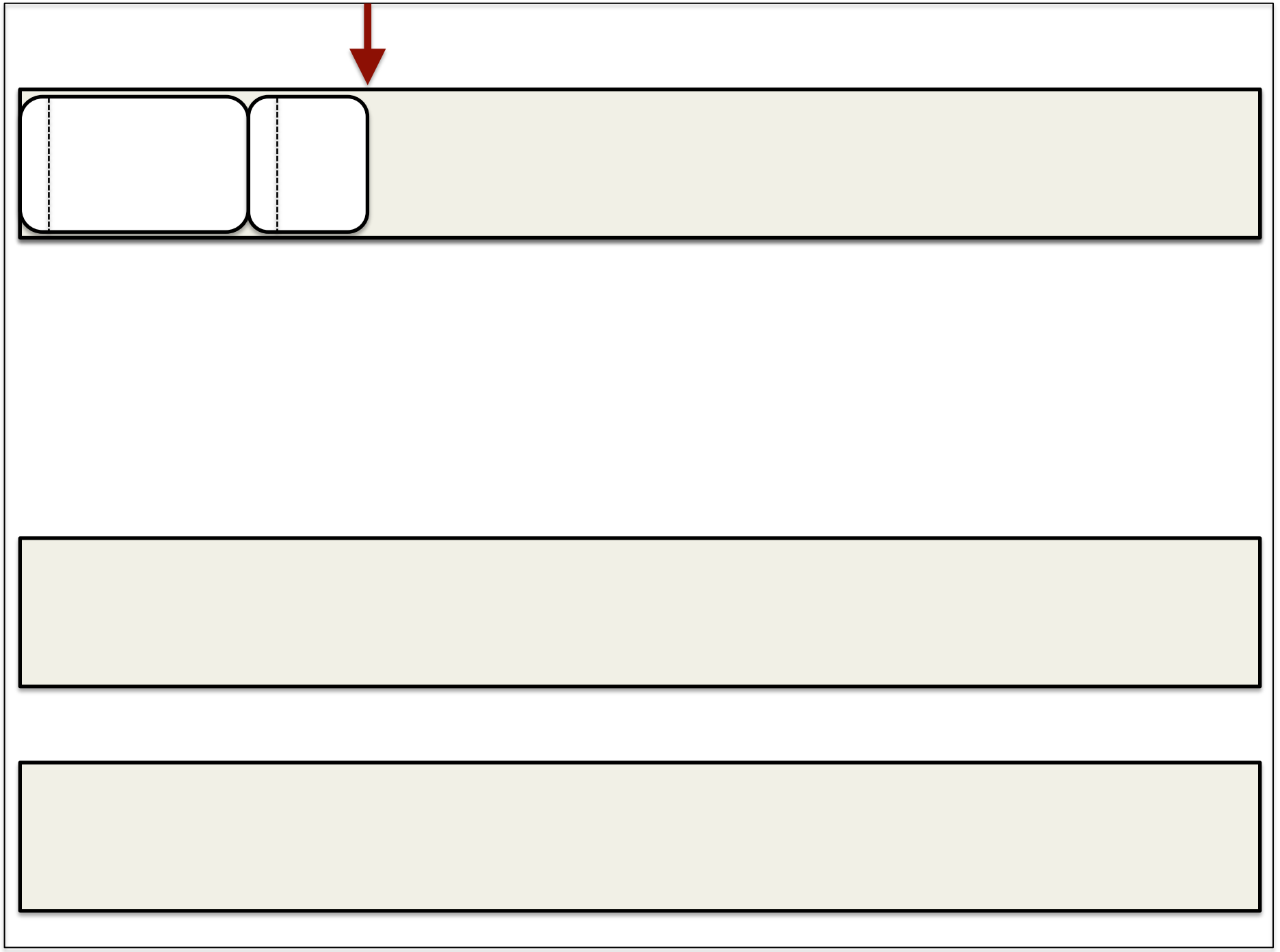
- First proposed in 1984
  - Consistently seen in practice
- Typically around 10% of objects survive
  - Varies by application
  - Frequently far fewer

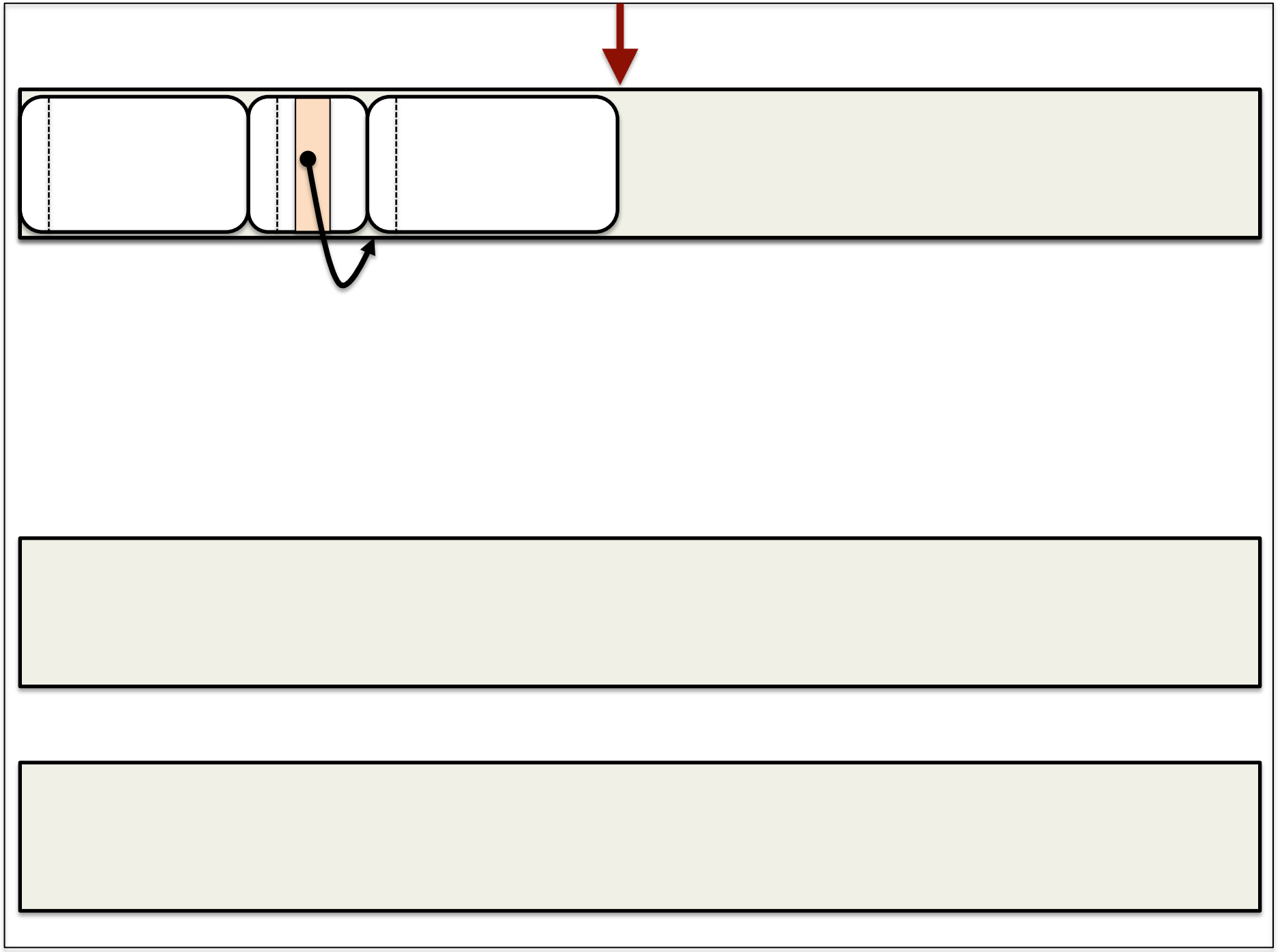
Generational Collectors

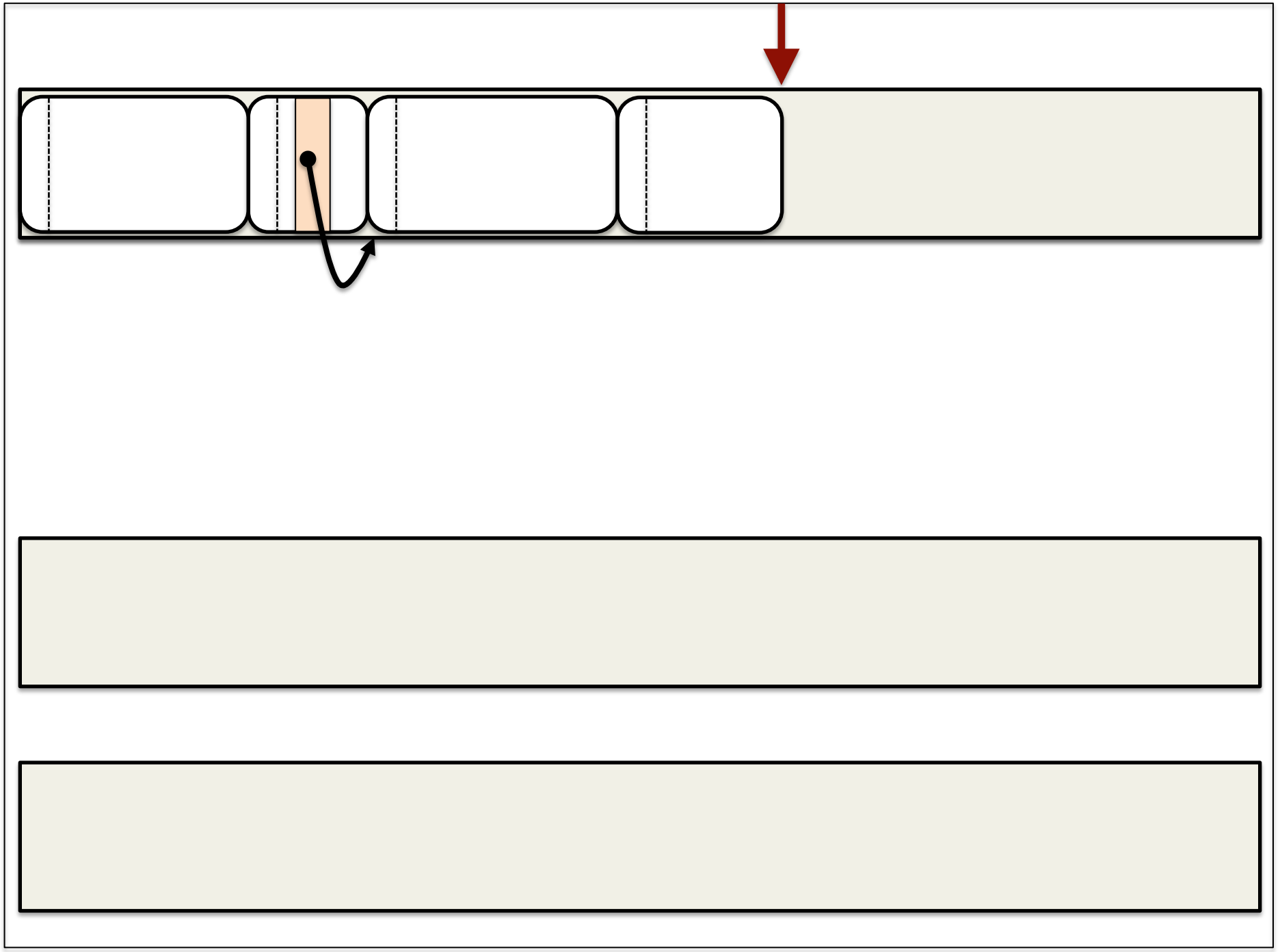
- Partition the heap into two regions
  - Regions may not be the same size
- Allocate to the first region
  - No flipping of semi-spaces
  - This area is the nursery – all allocation happens here
- During GC, move live objects to the second region
  - All object there have survived a GC
  - Known as the mature space

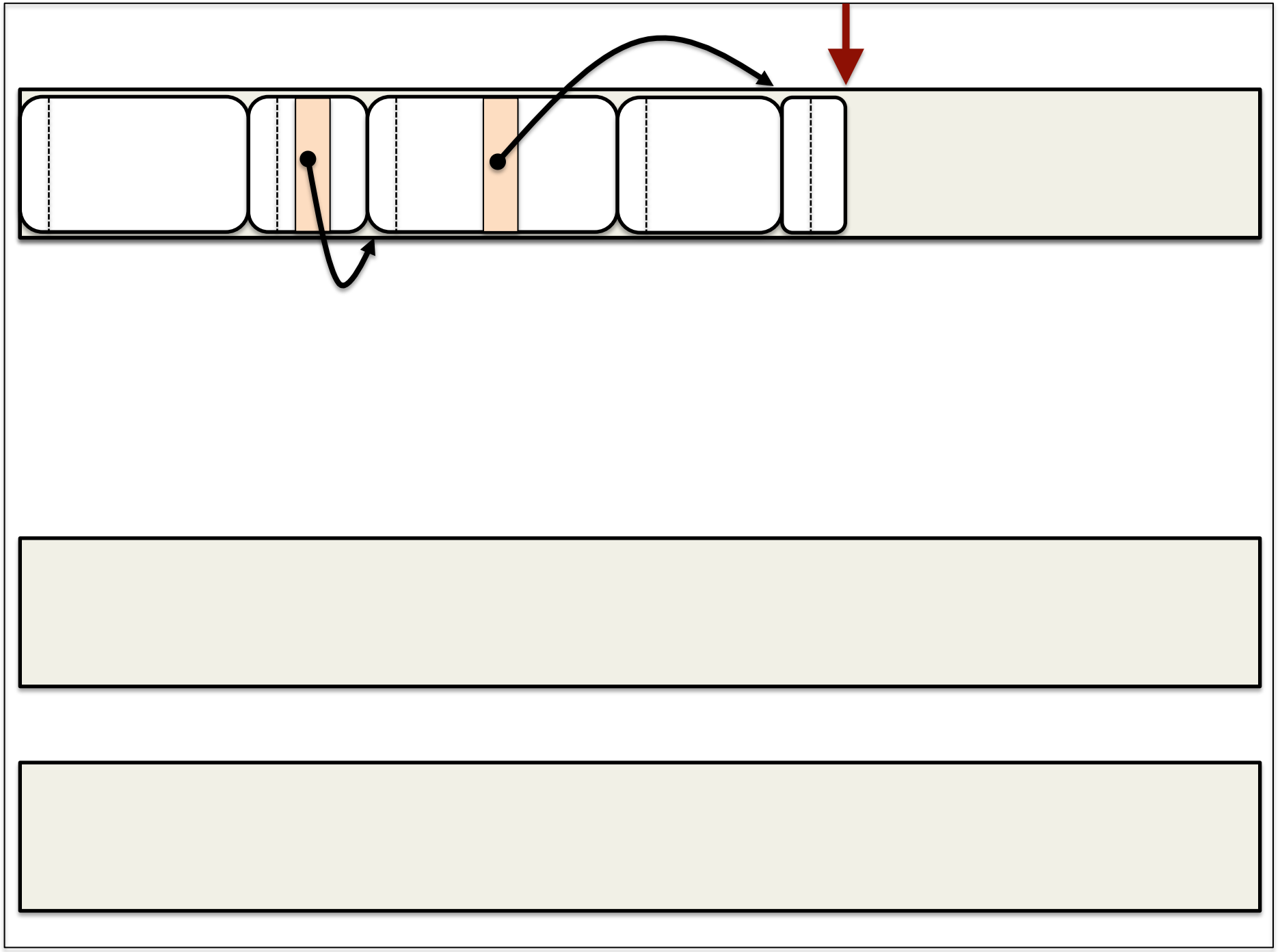


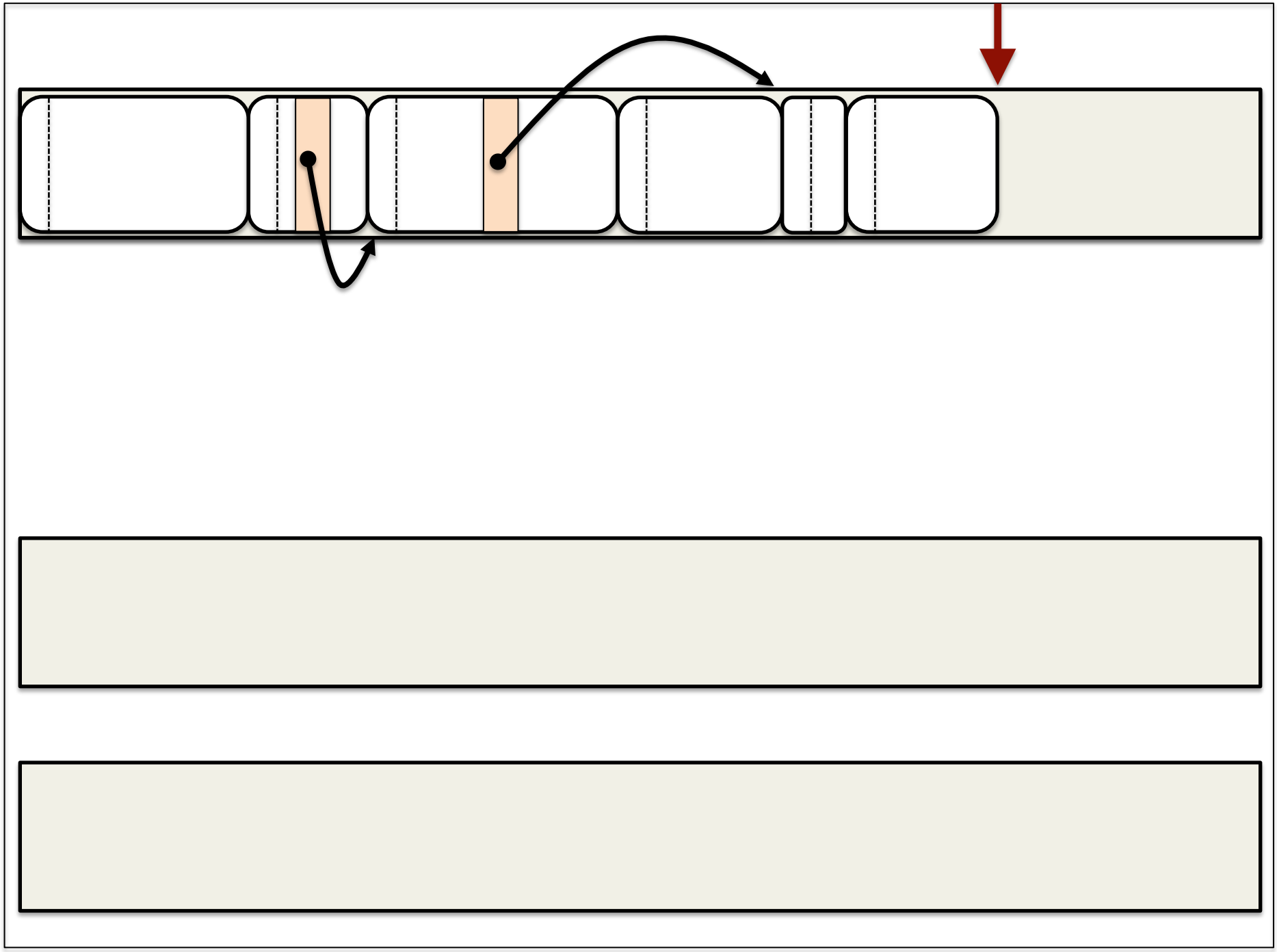


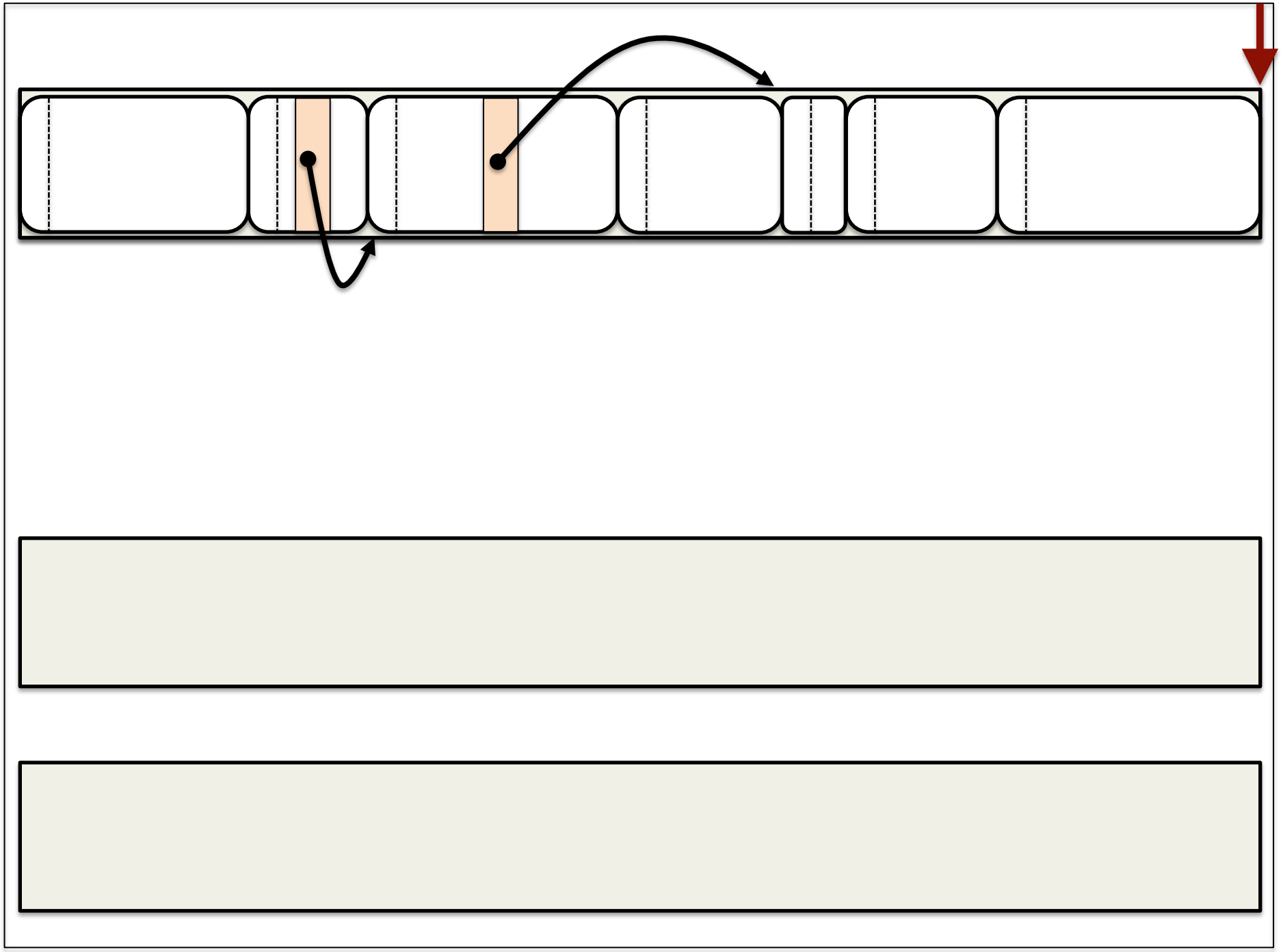


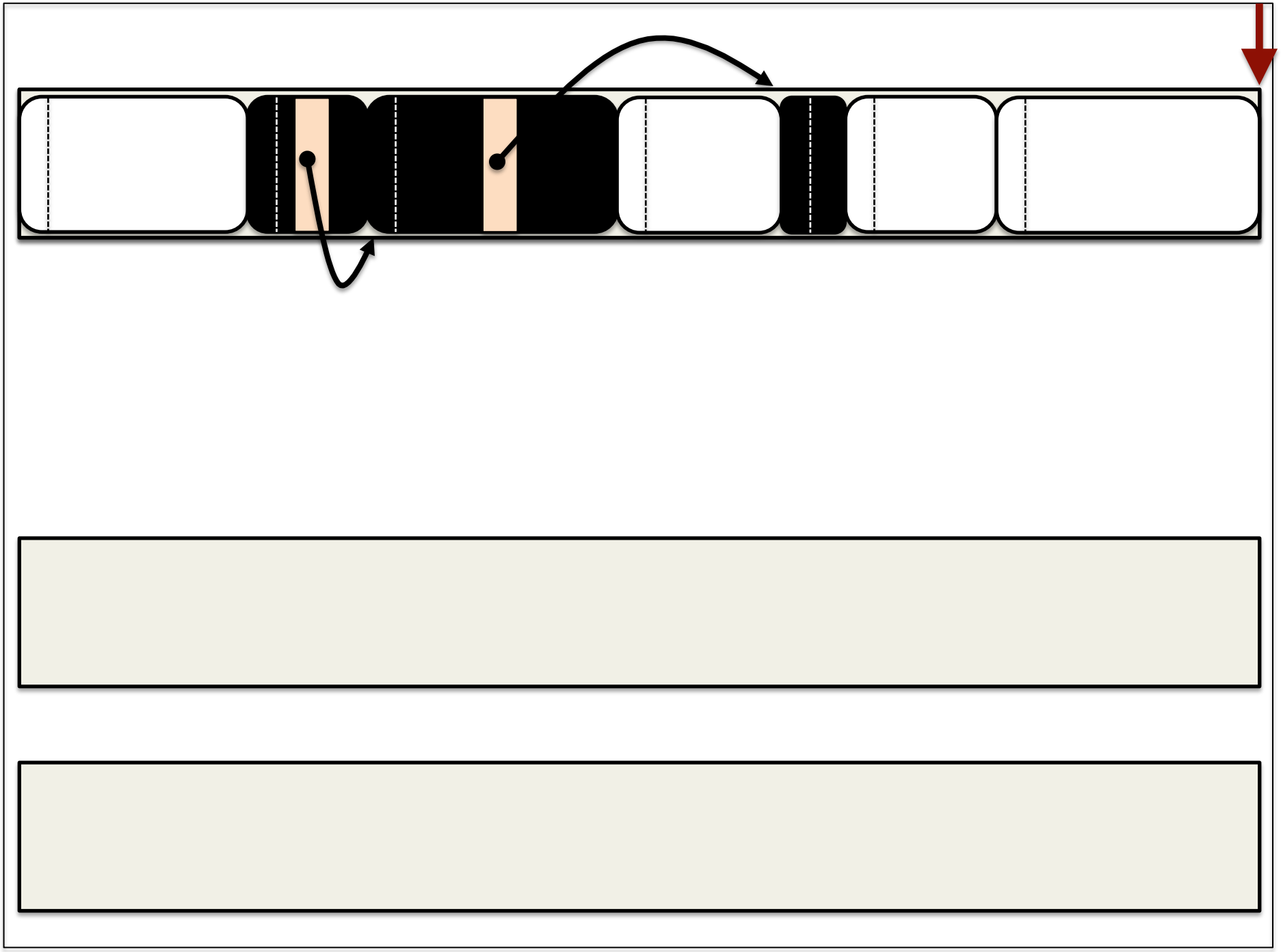


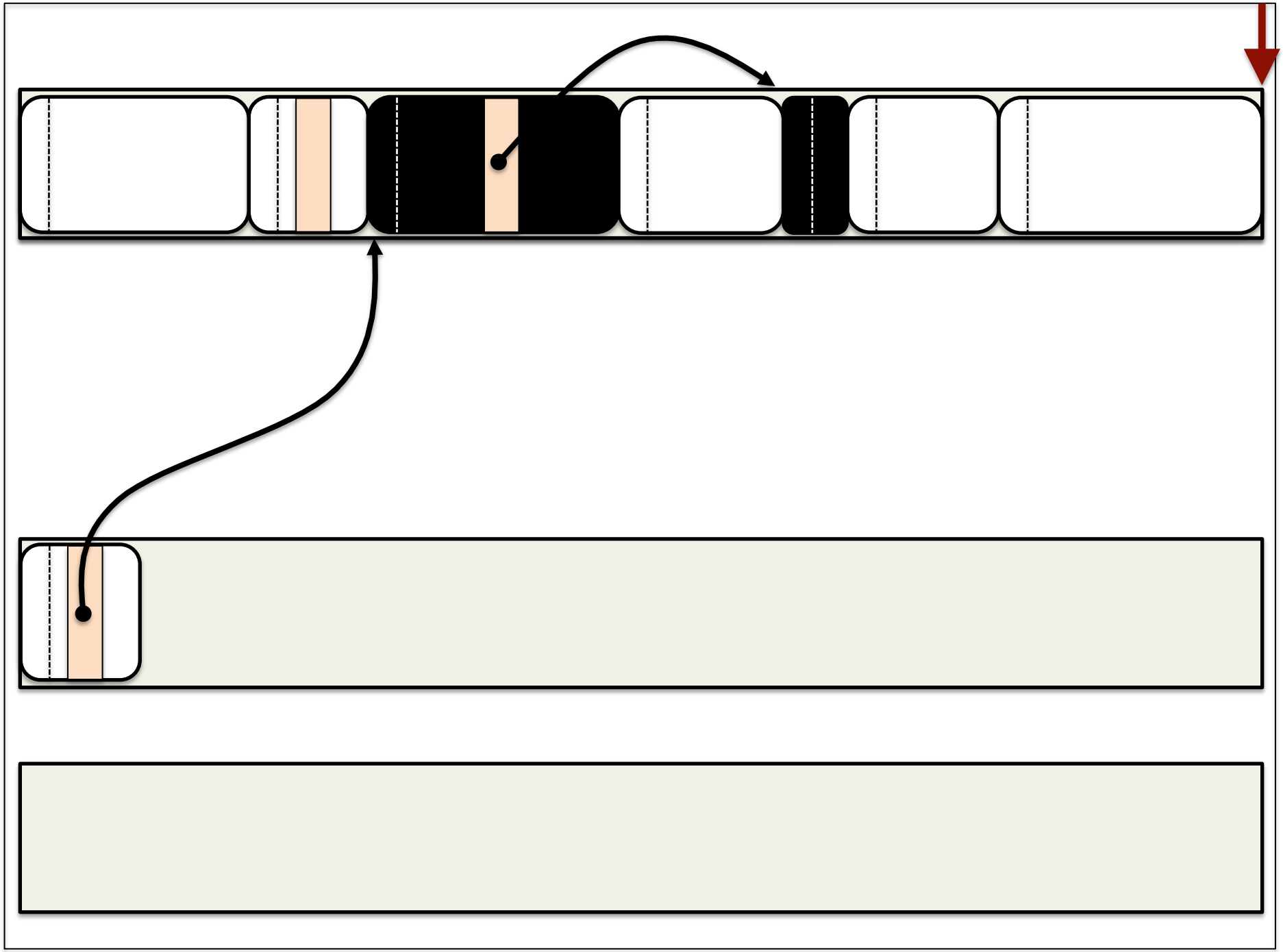


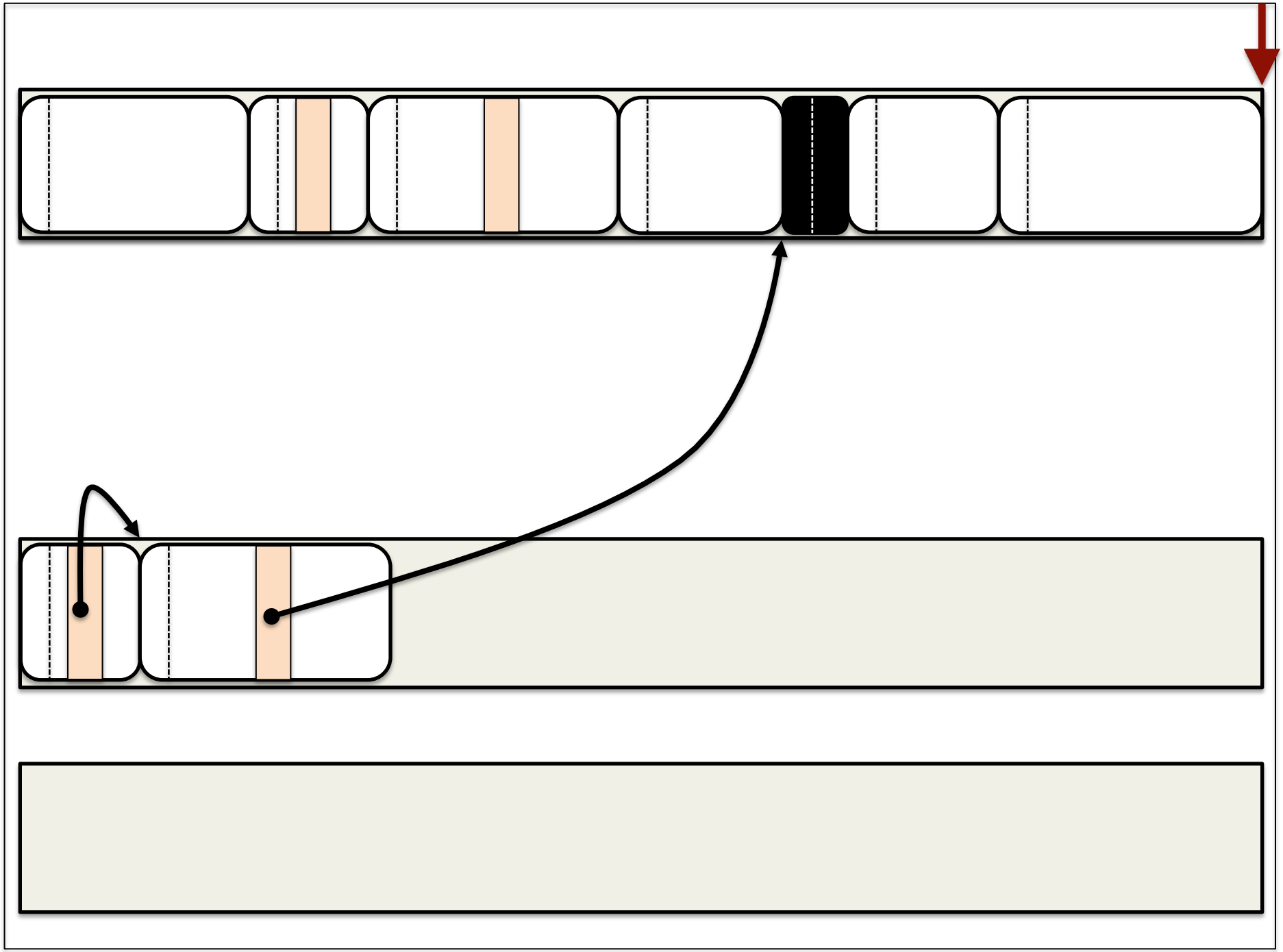


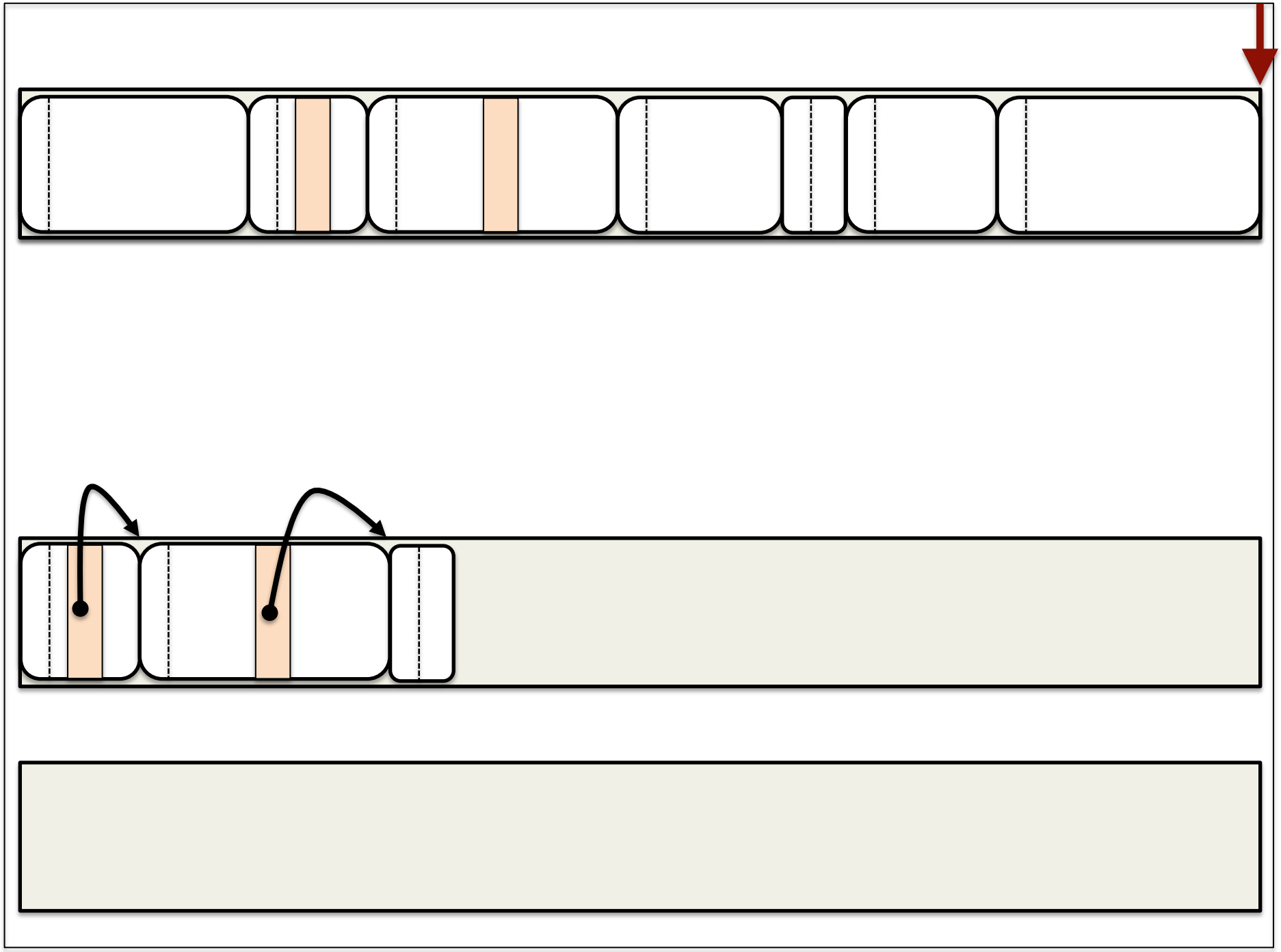


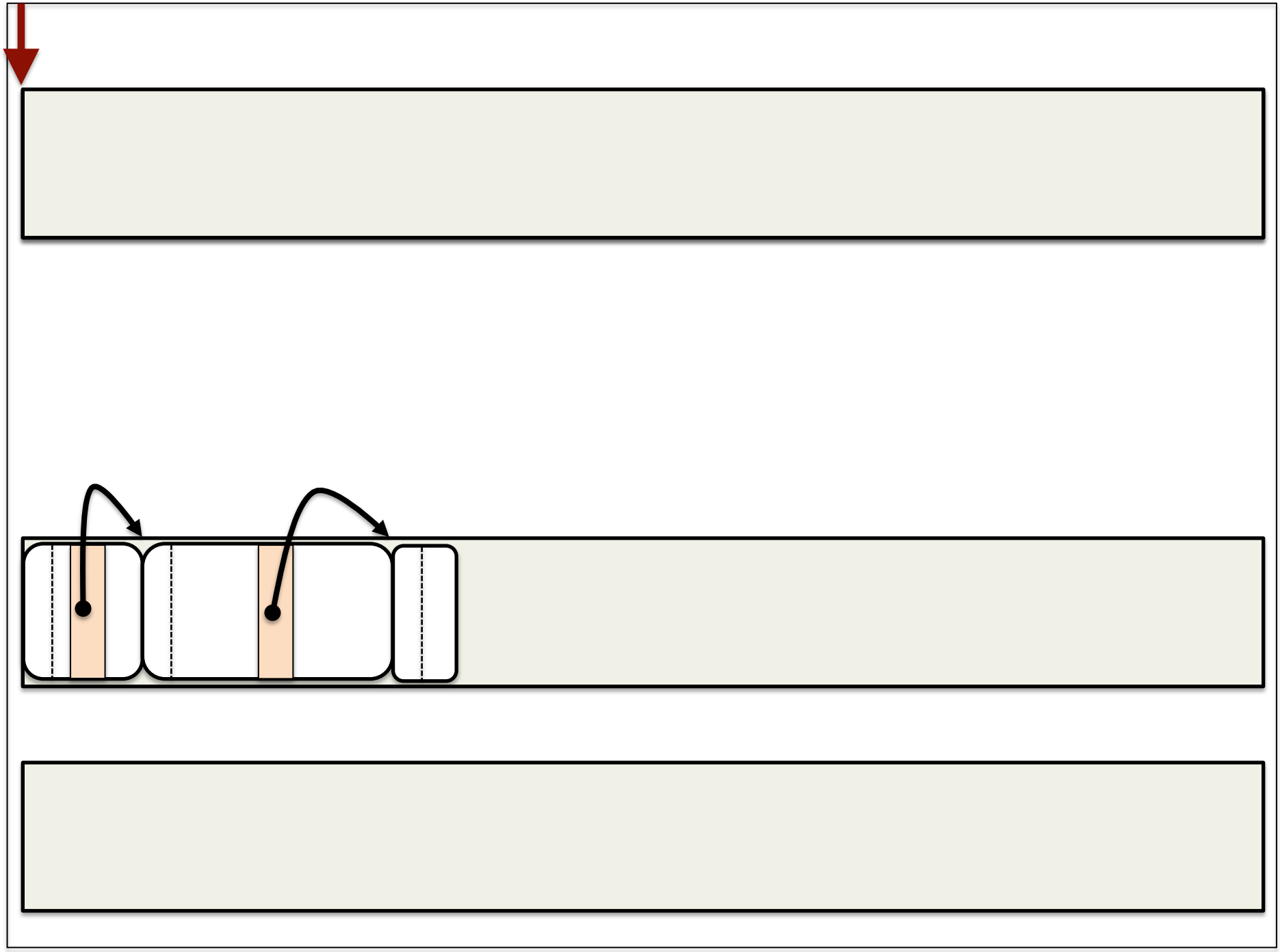






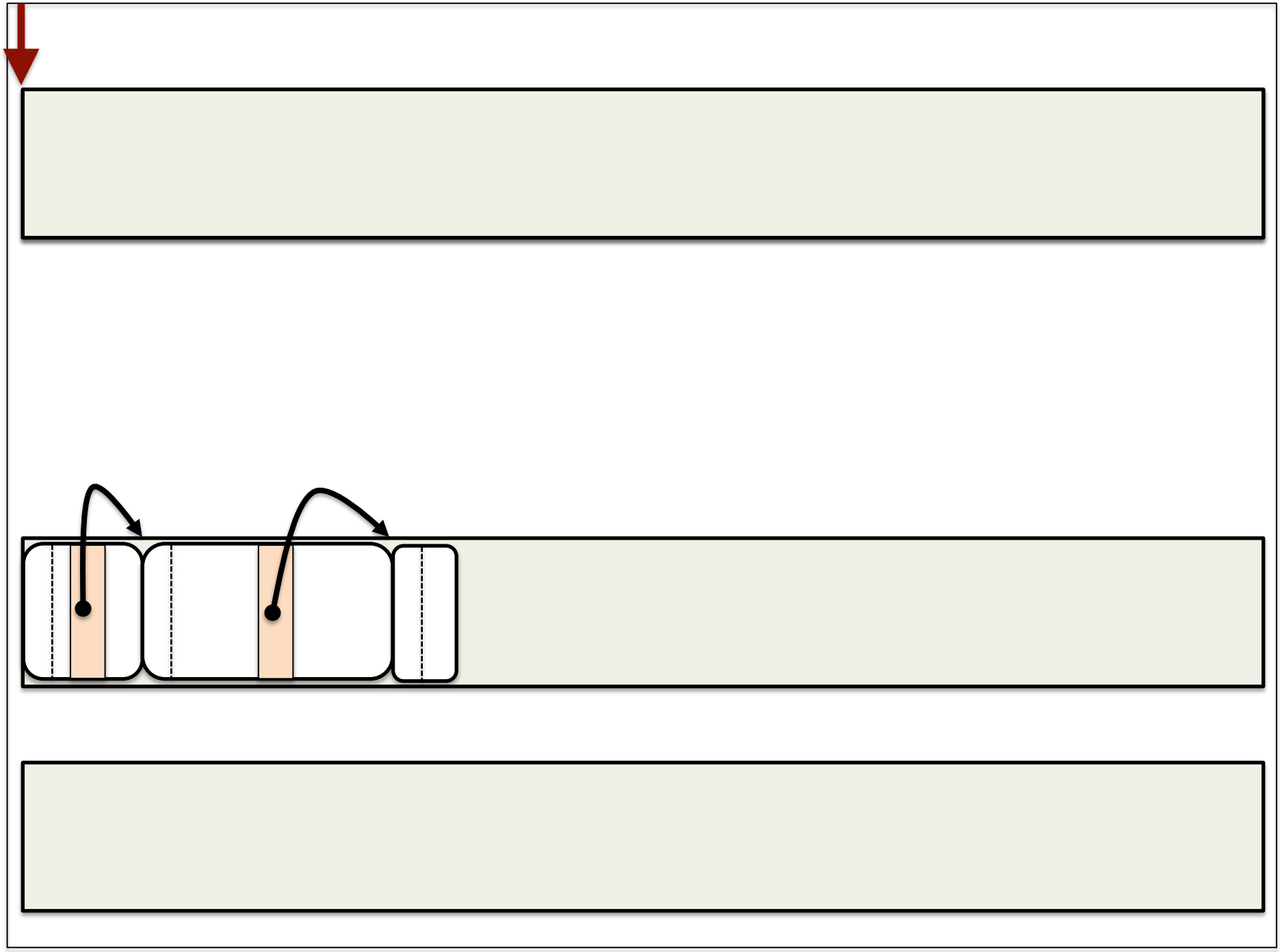


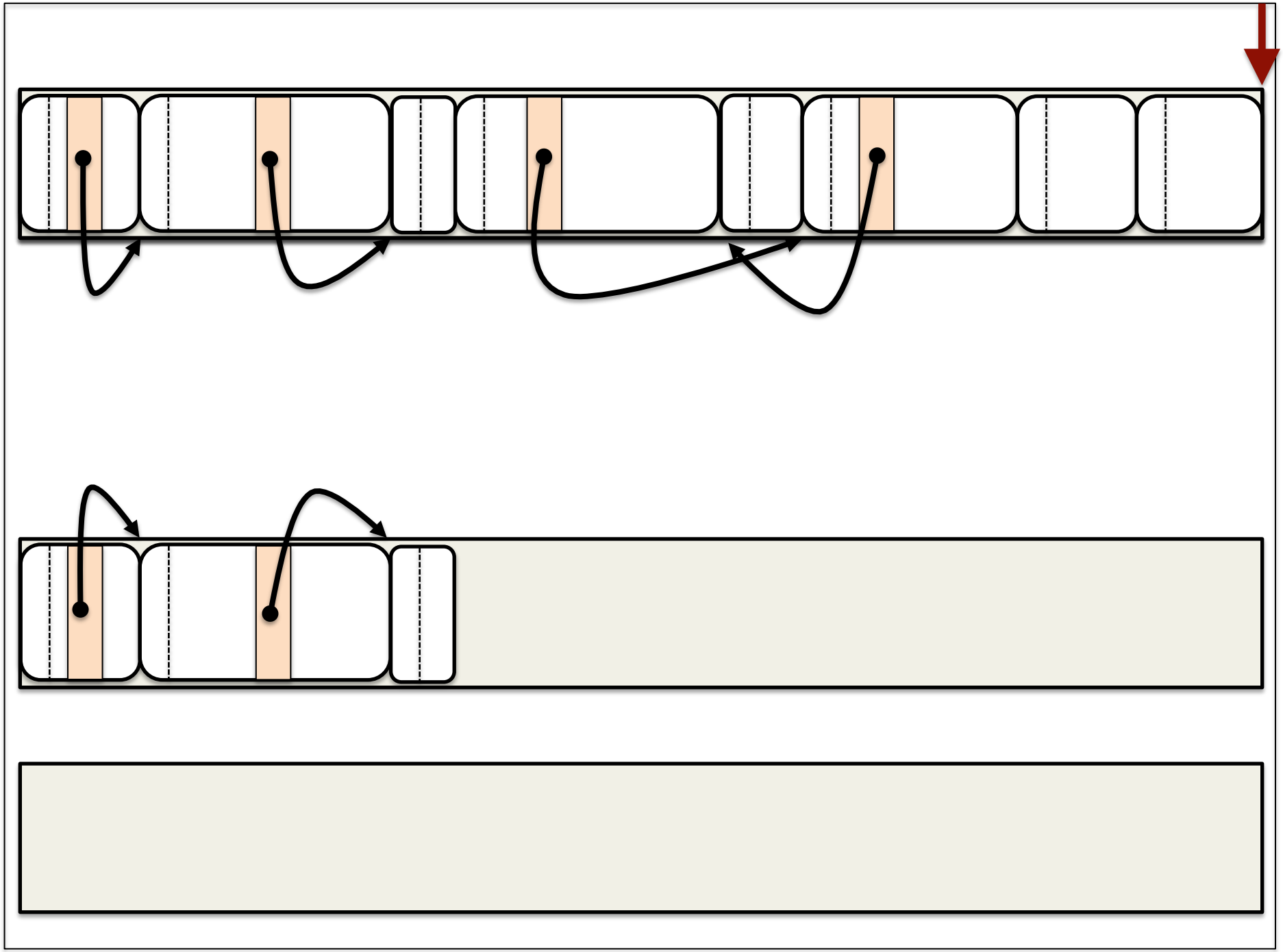


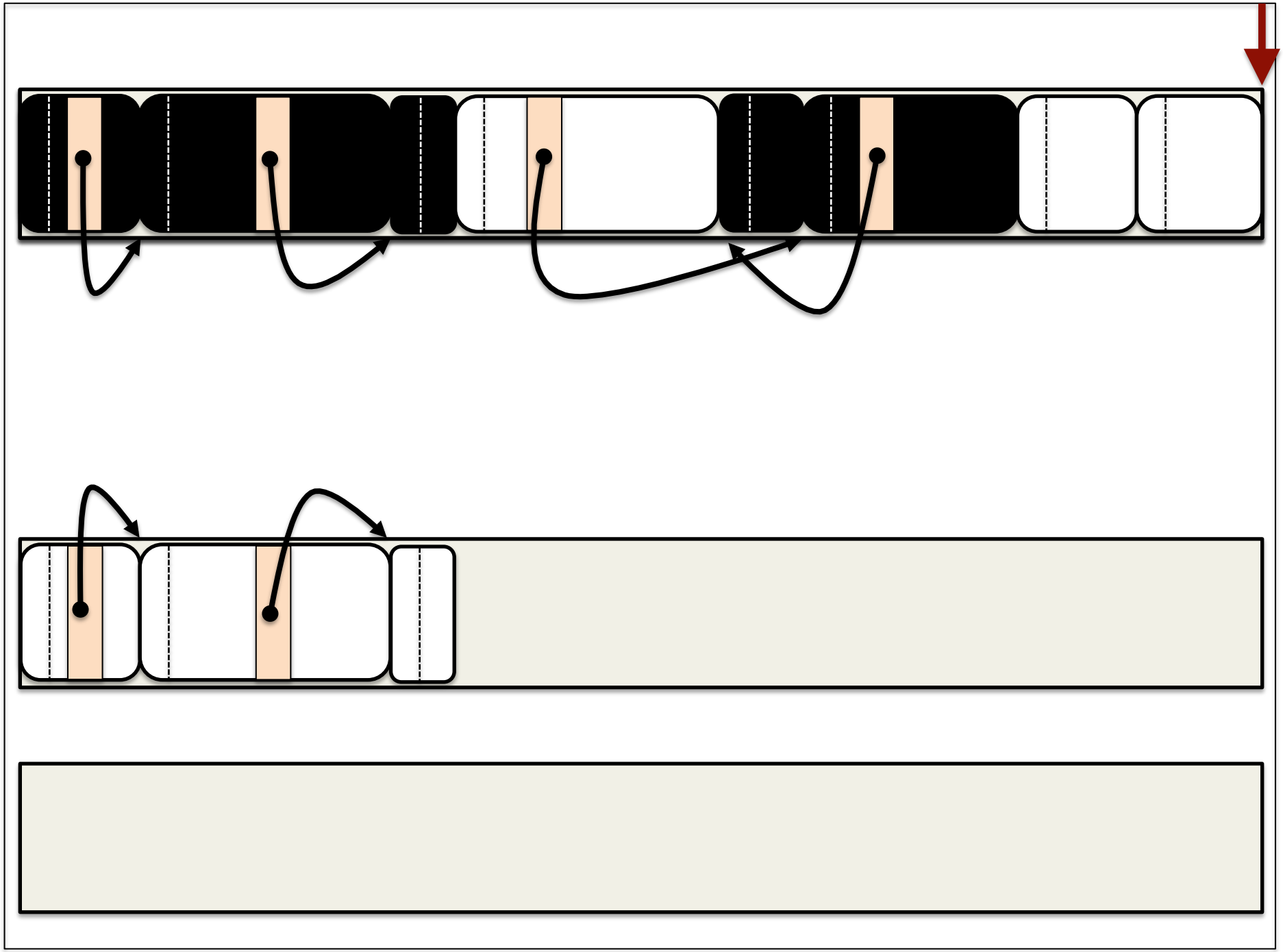


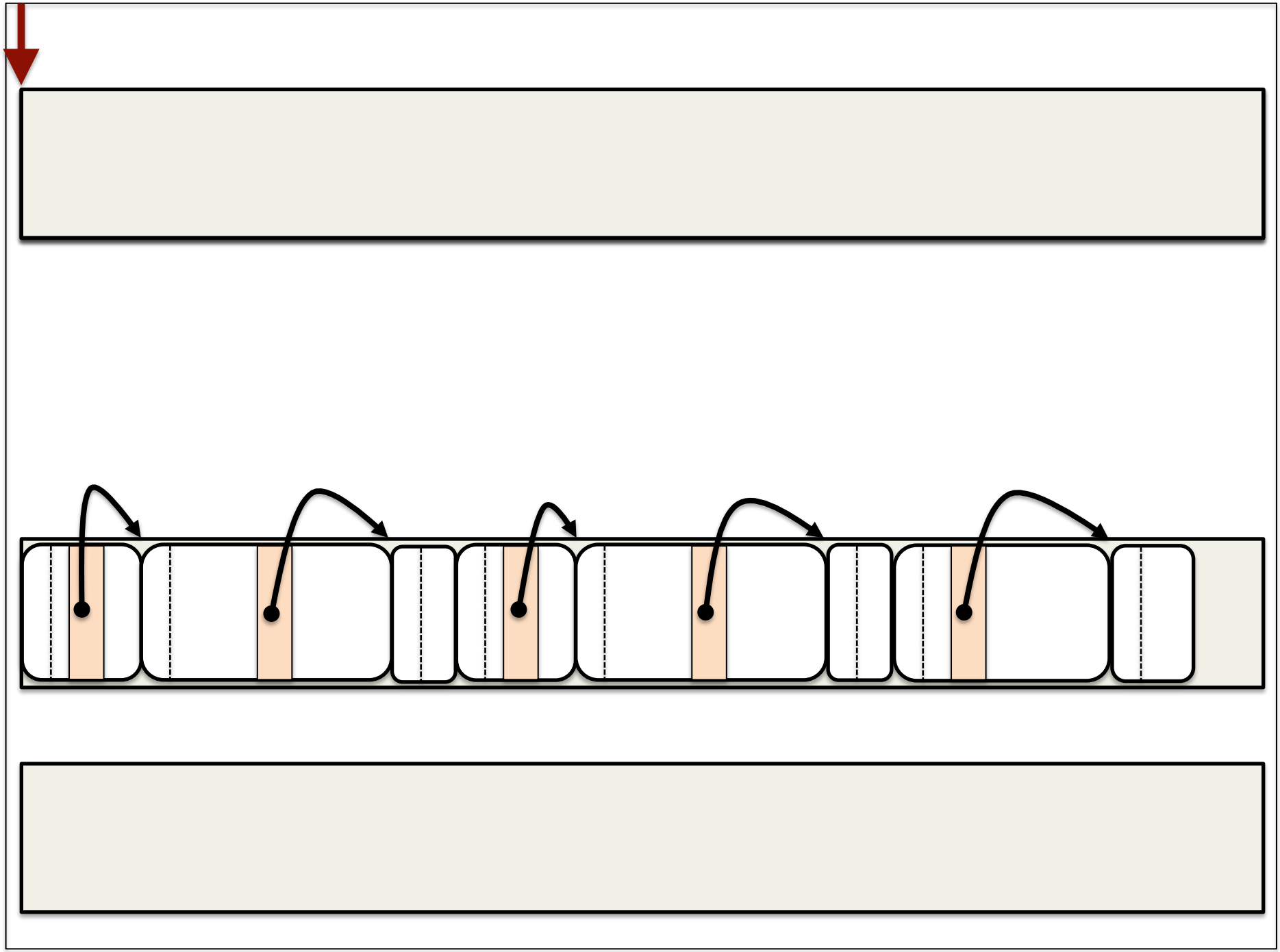
Minor Collections

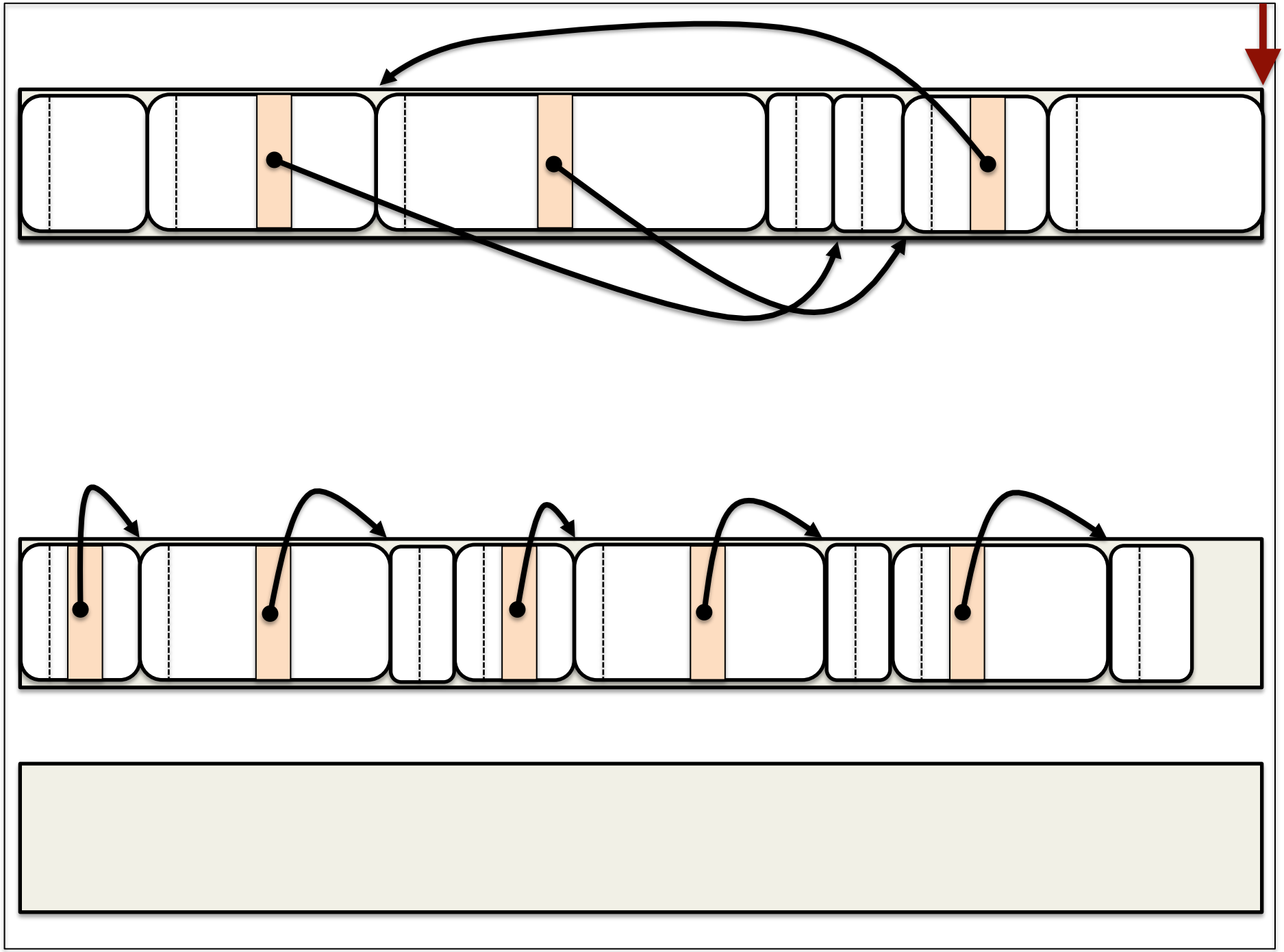
- Minor collections focus only on the nursery
  - Mature space is only used as a destination
- Promote all live objects to the mature space
- Reset bump pointer in the nursery
  - New allocation starts again in the same region

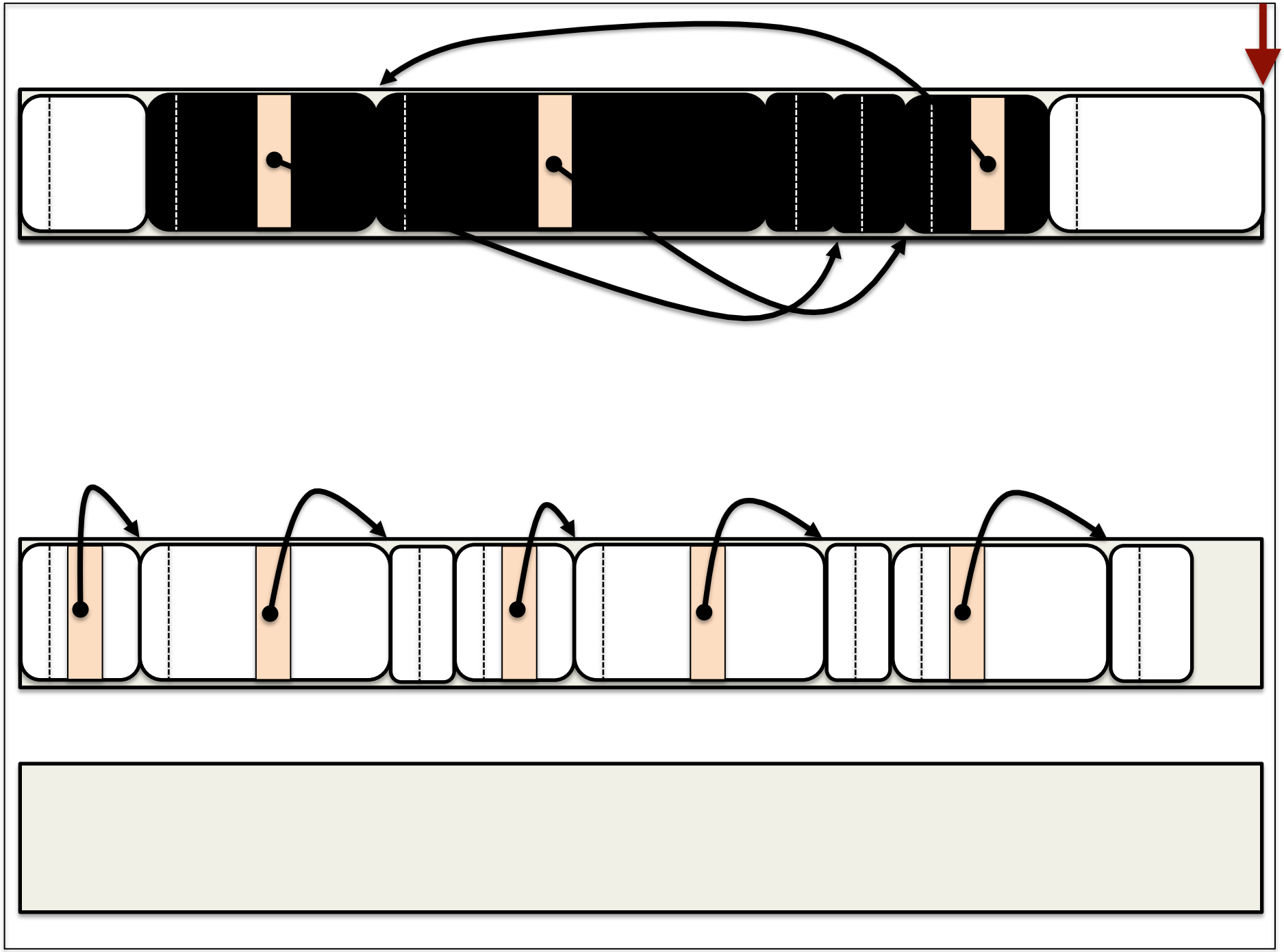


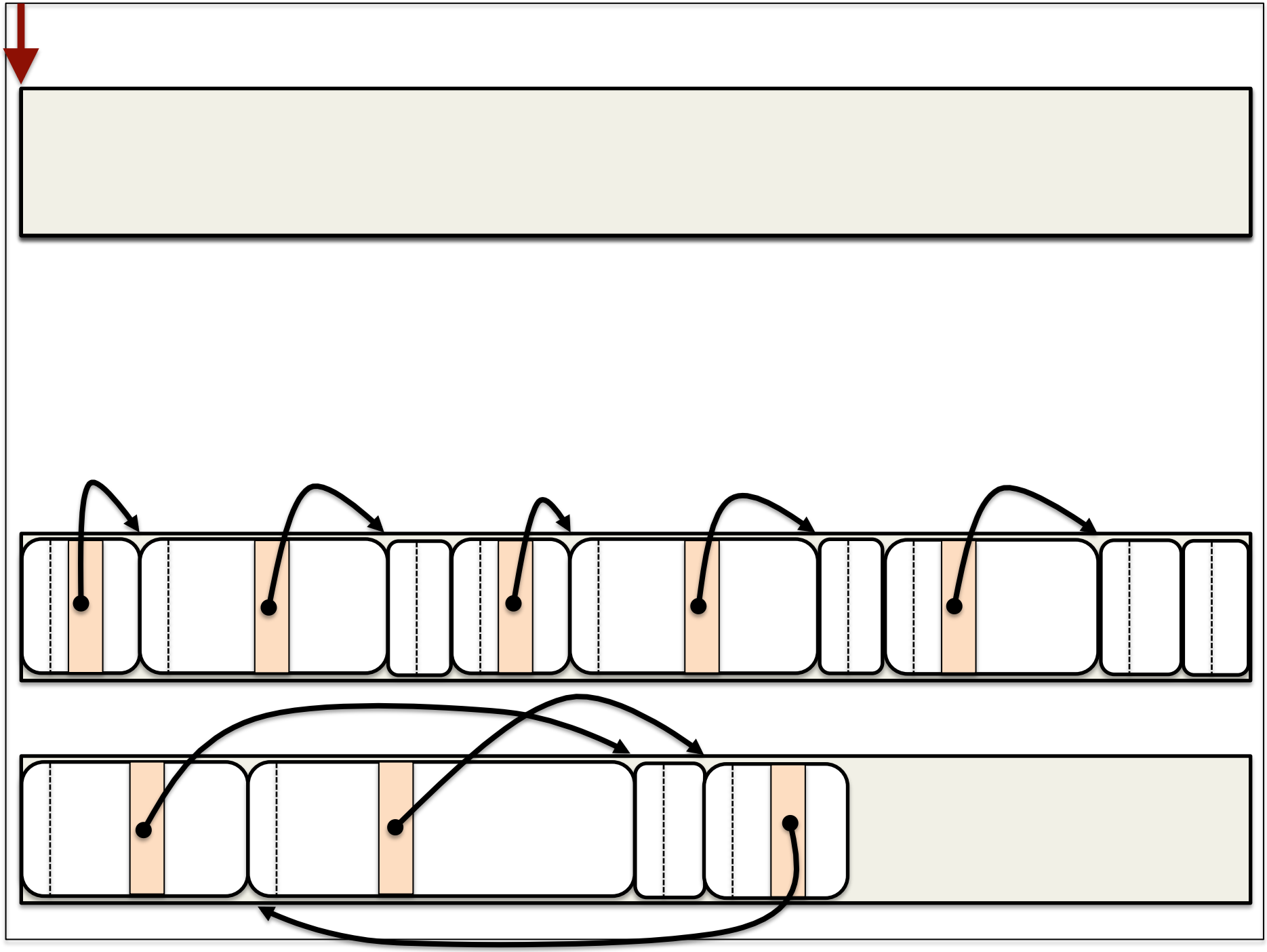






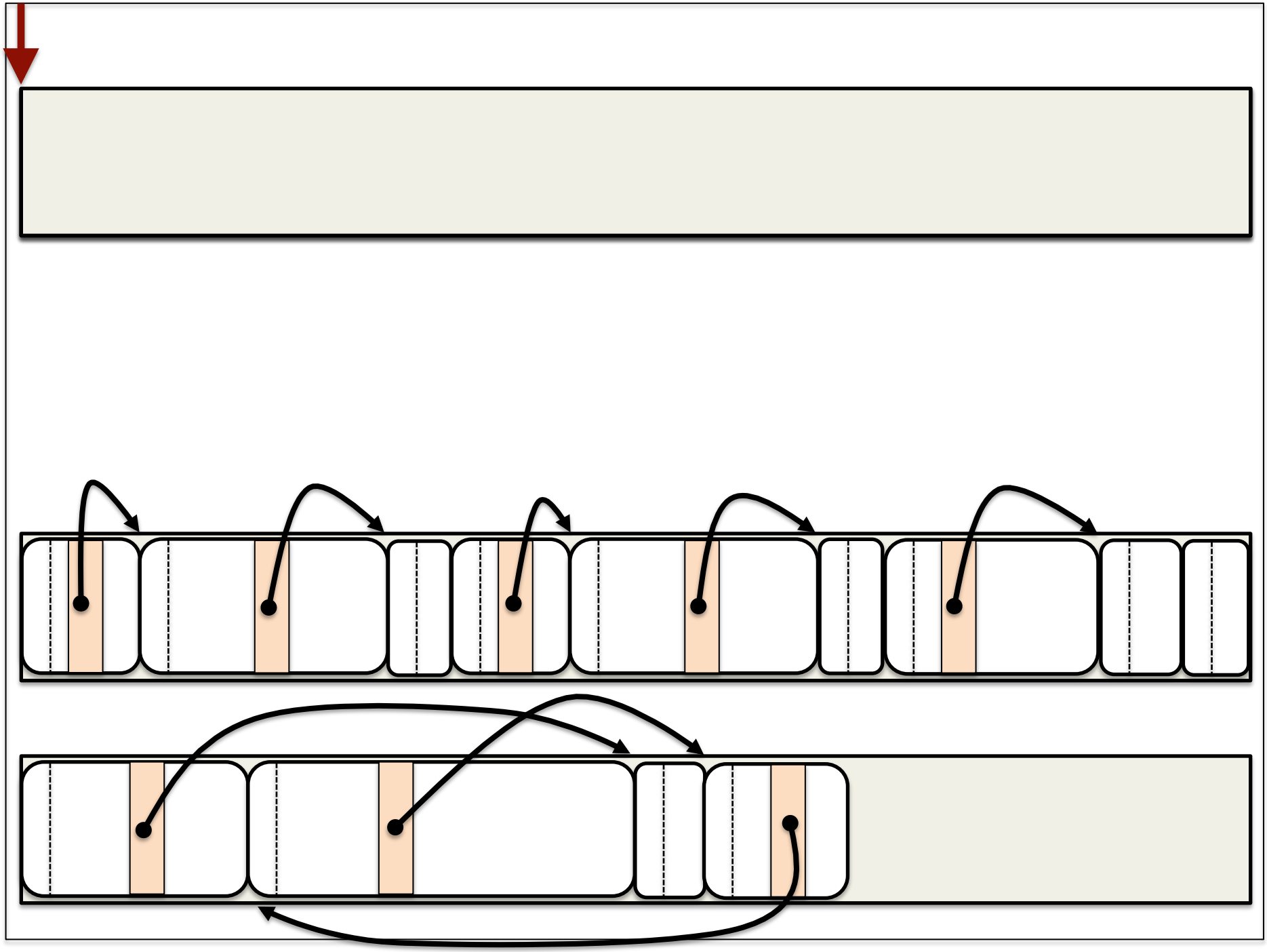


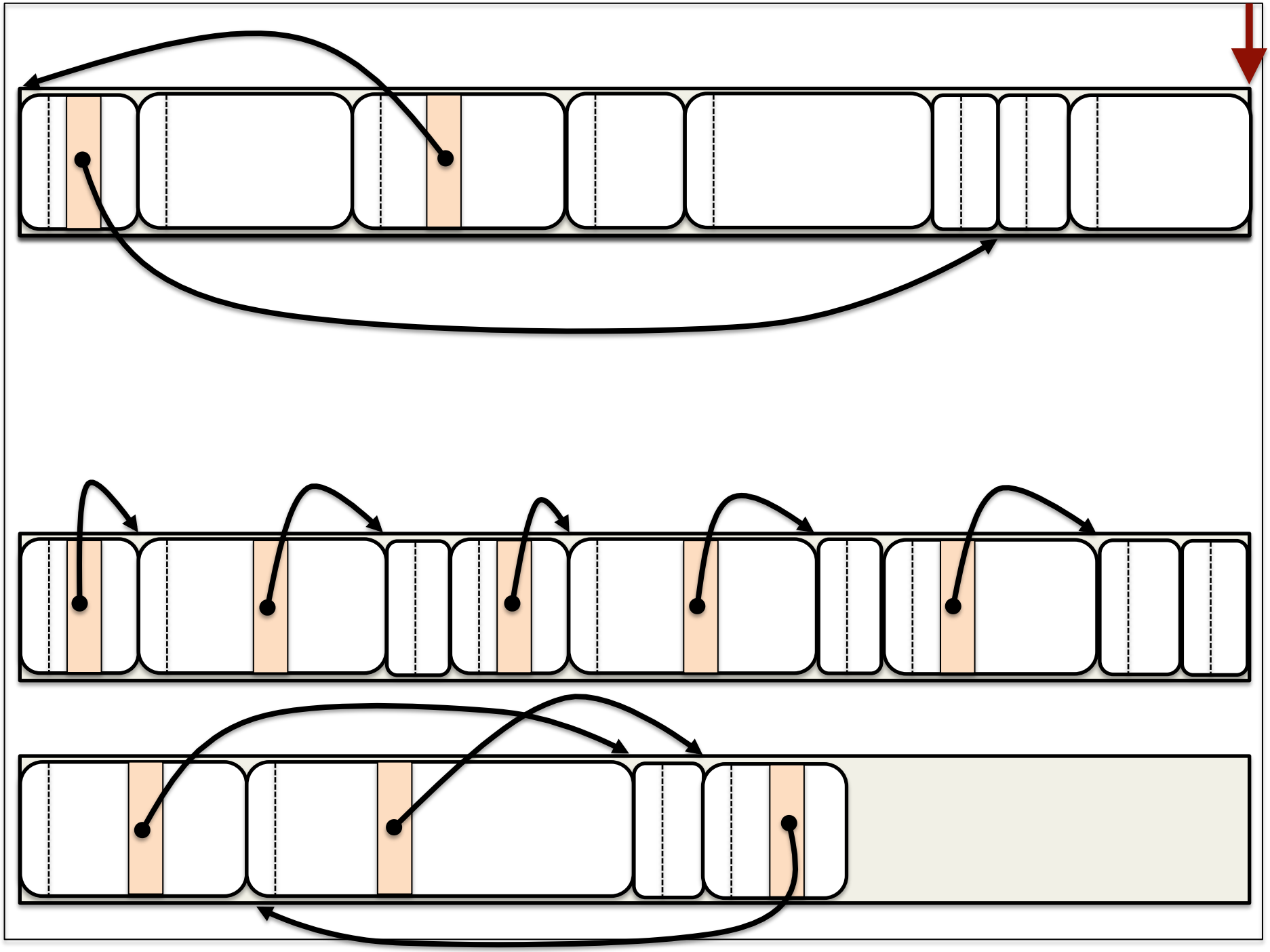


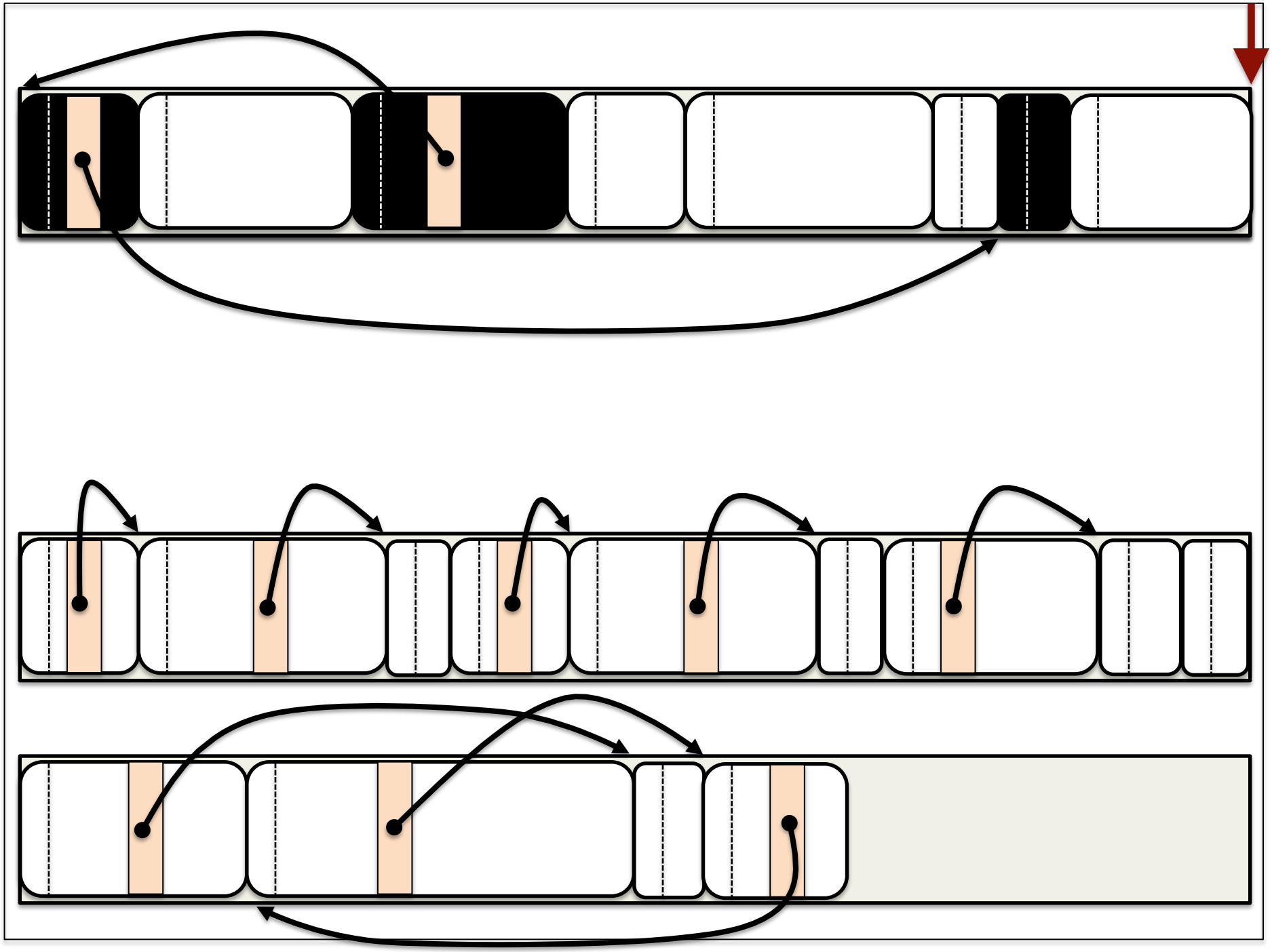


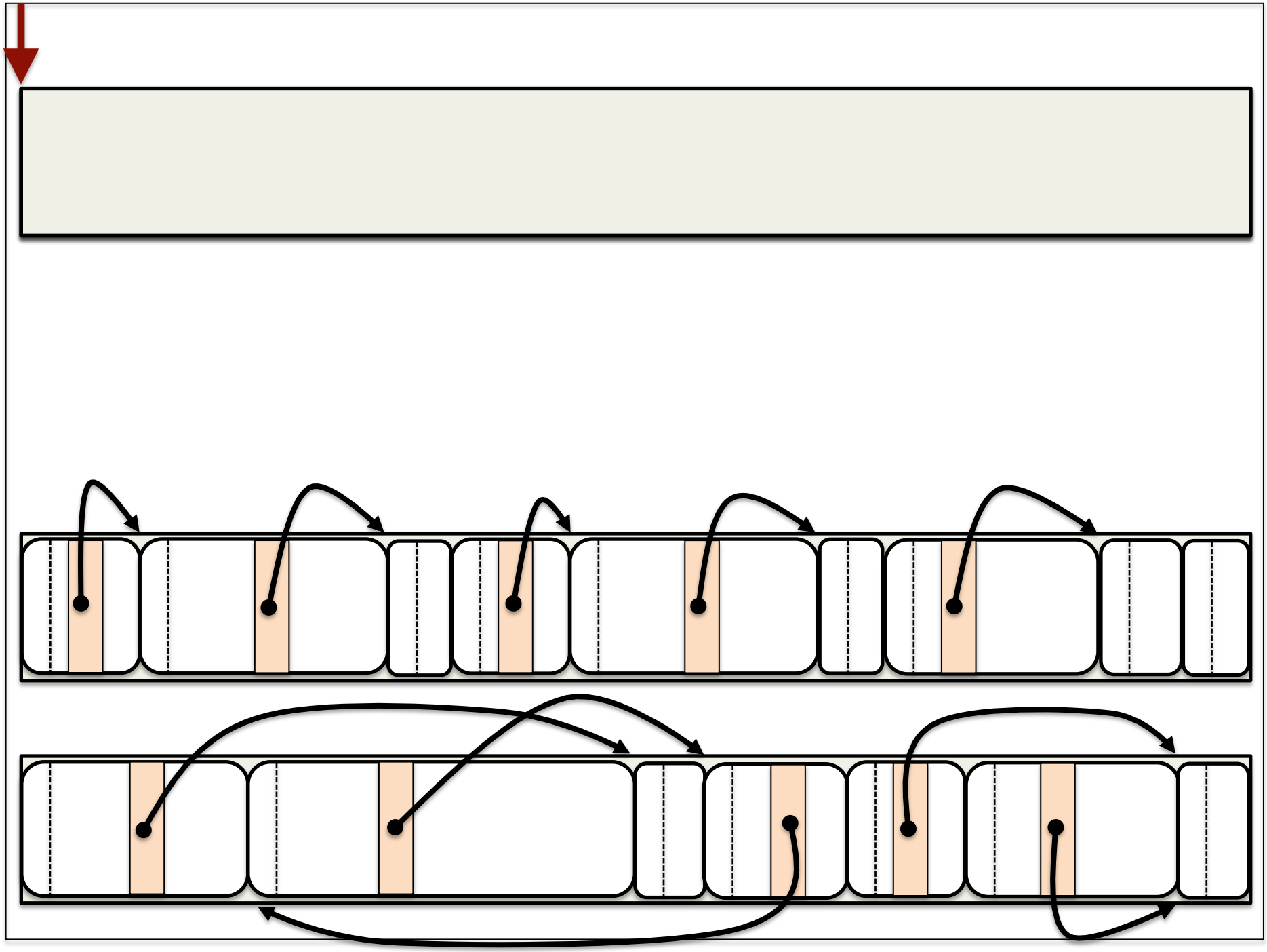
Minor Collections

- Minor collections are very fast
  - Only looking at a small part of the heap
  - Most objects in the nursery are garbage
- Mitigates the drawbacks to copying
  - Maximizes garbage to live ratio
  - Don't copy long-lived objects many times









Major Collection

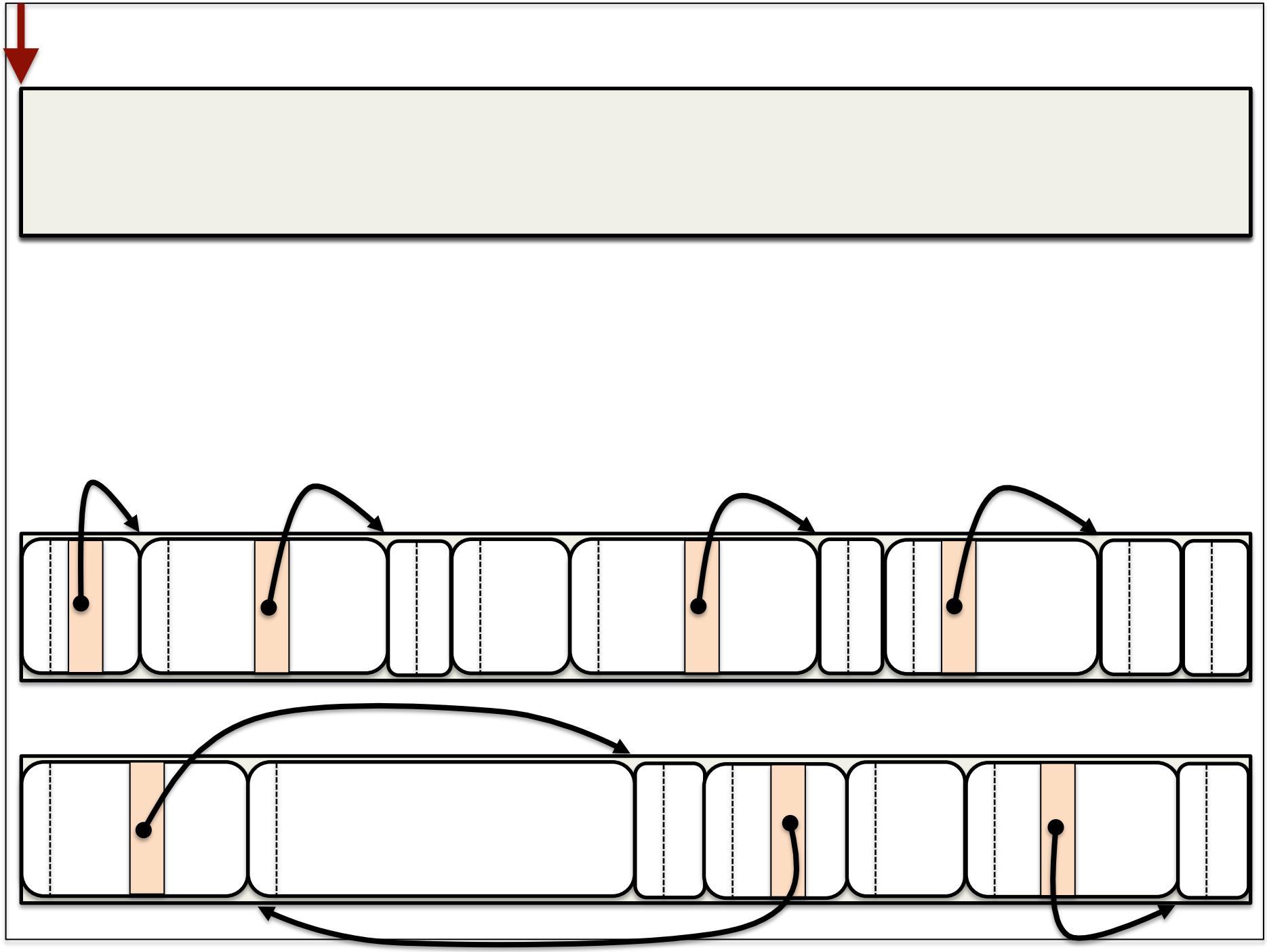
- Eventually the mature space fills up
- We trigger a major collection to free space
  - Collect the whole heap
- Same performance impact as a regular GC
  - Much less frequent than minor collections

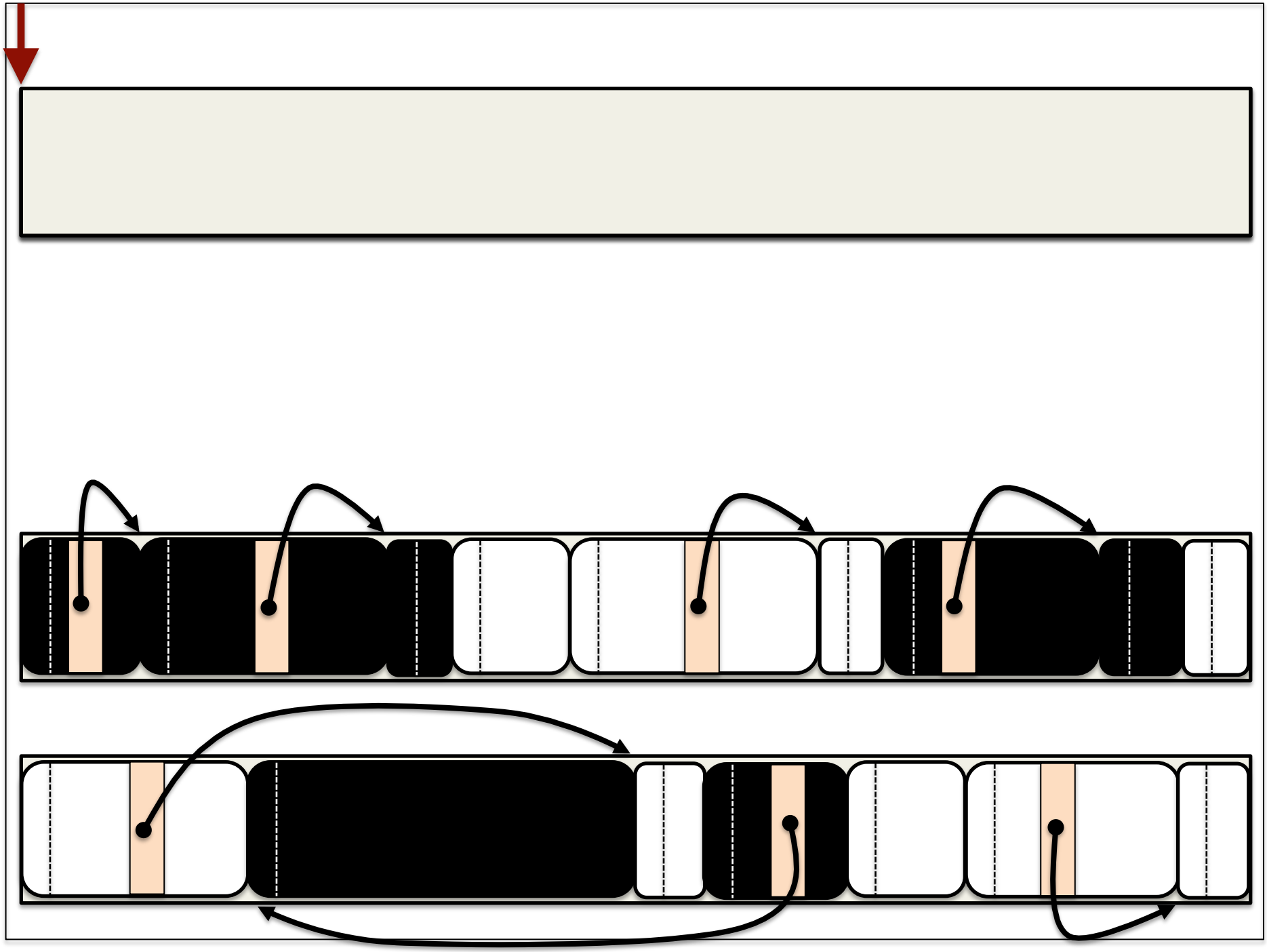
Triggering a Major Collection

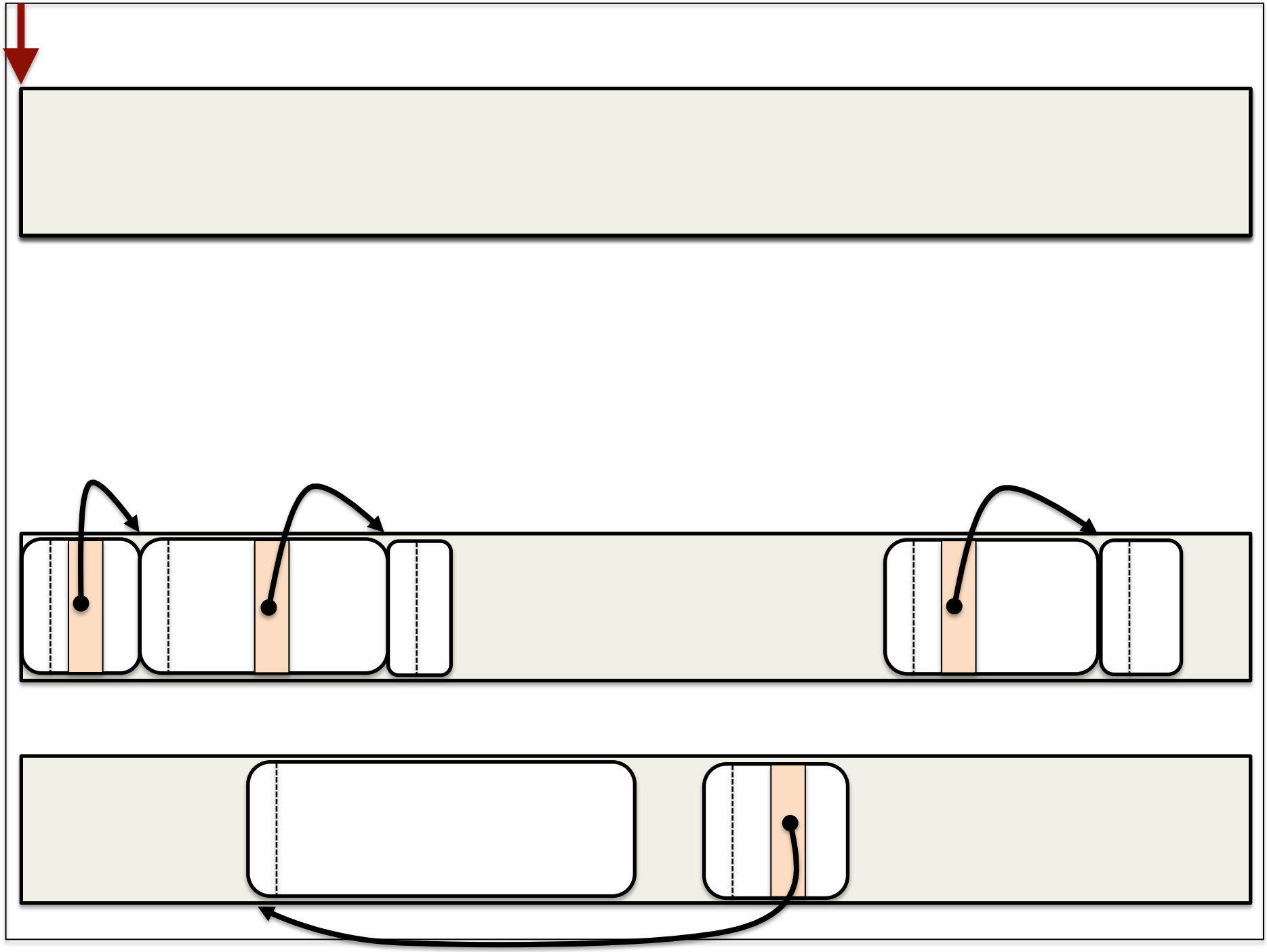
- GC algorithm triggers major collection
  - When no space on nursery promotion
  - Based on heap occupancy
  - When the user calls `System.gc()`
- Trade-off complexity for time
  - Frequent major collections more expensive
  - Switching from minor to major has overhead

Per-Region Management

- The nursery always uses copying collection
  - Need to evacuate objects
  - Want to use bump-pointer allocation
- Mature space can use a different algorithm
  - May not want to use a copy reserve
  - Allocation happens less frequently
  - May have a high live to garbage ratio
    - If strong hypothesis is correct





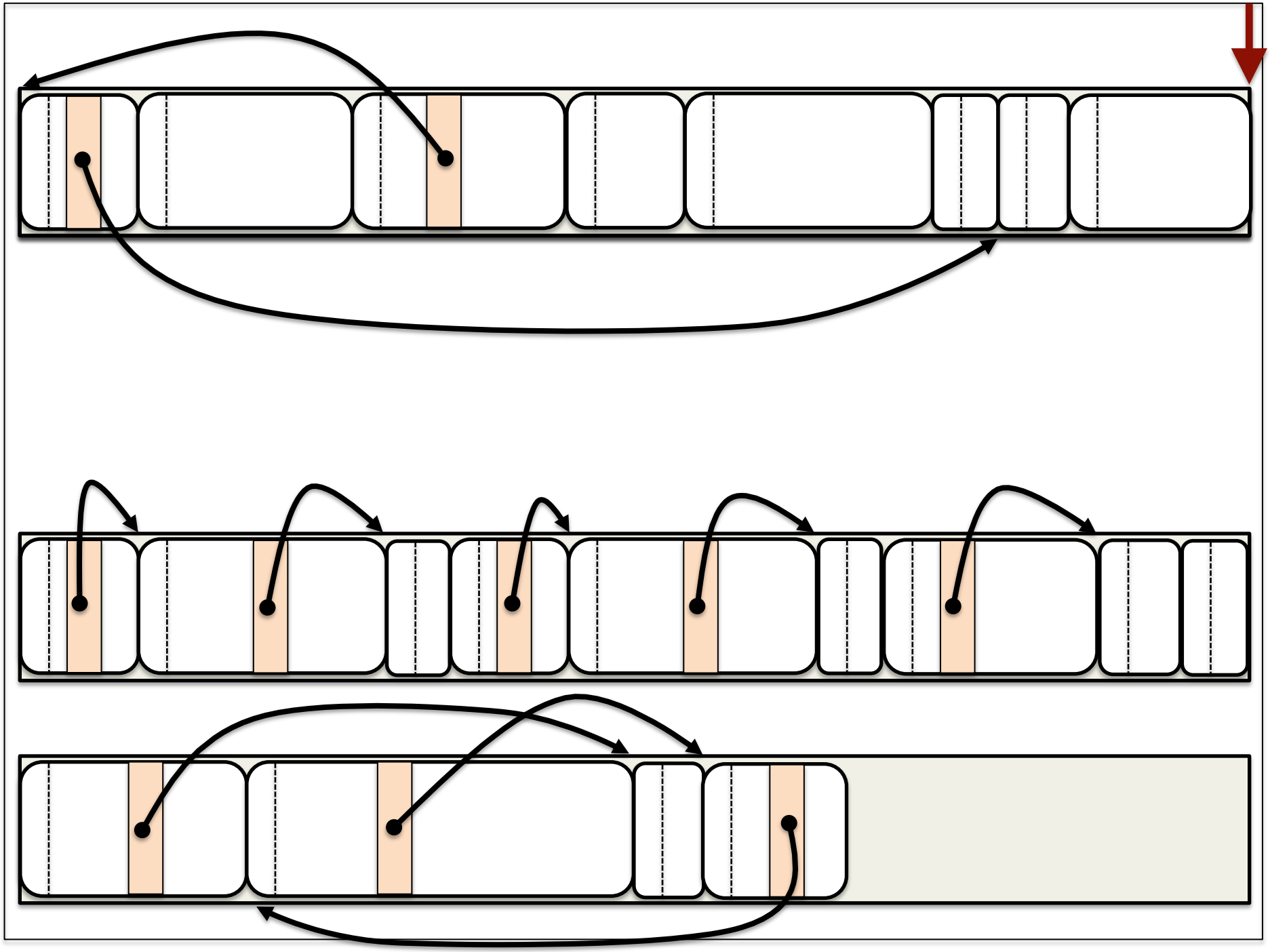


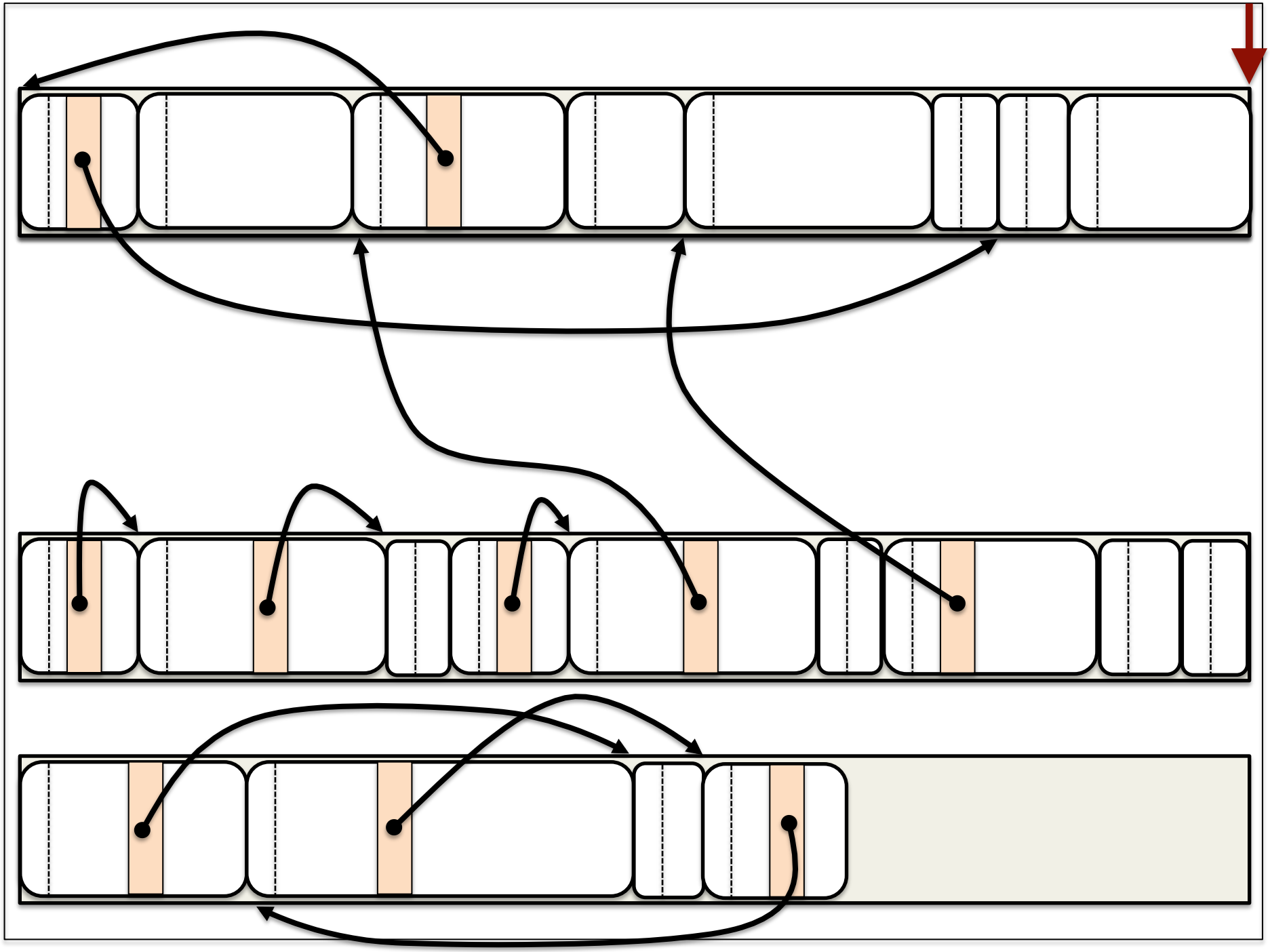
Mature Space Management

- Lots of variation in how to collect mature space
  - Mark/Sweep
  - Semi-Space
  - Mark/Compact
  - Various hybrids
- Tuning can depend on application characteristics
- Concurrent Mark Sweep (CMS) used in Hotspot

Reference Directions

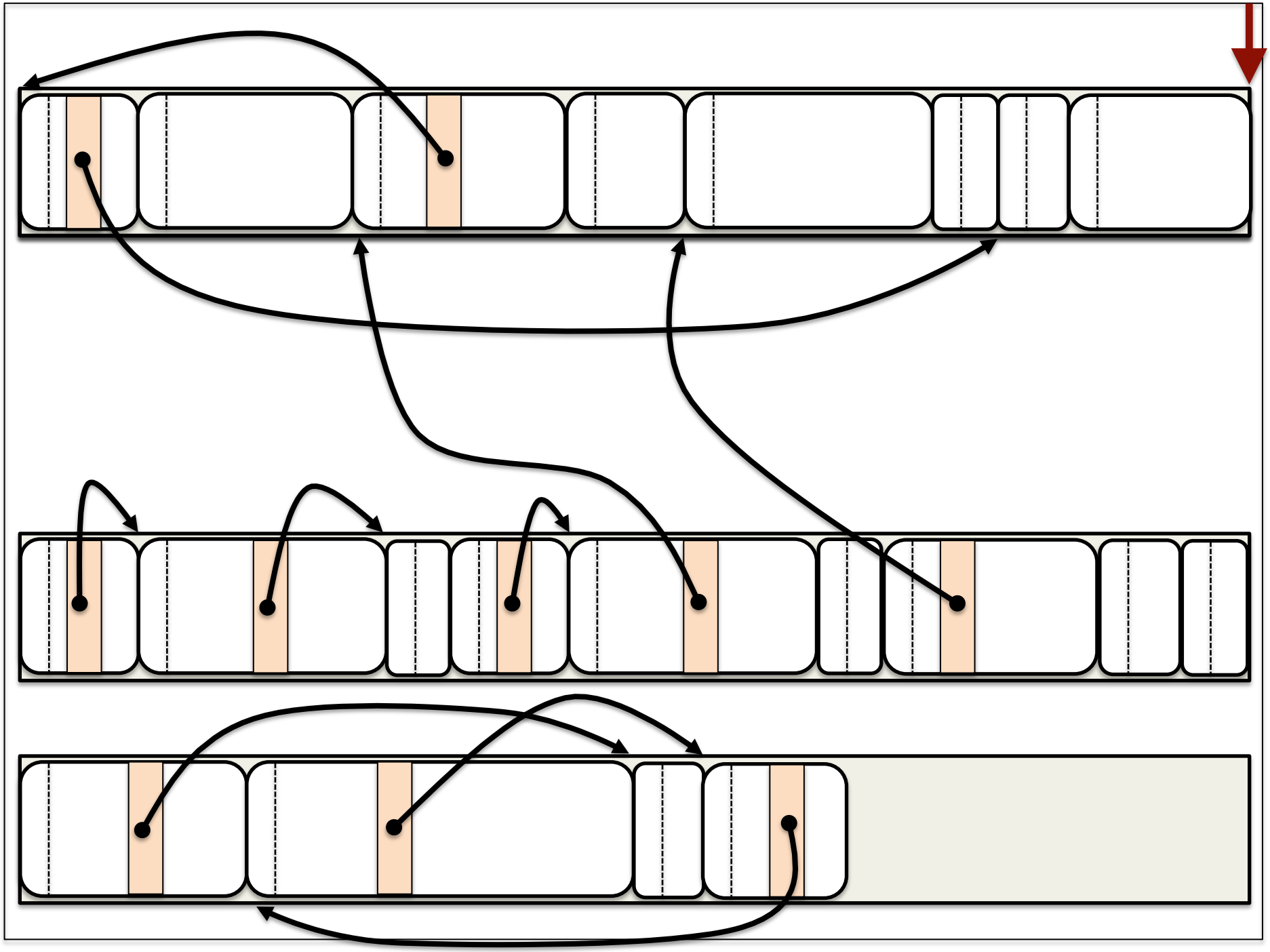
- So far we've assumed immutable objects
  - Can't change an object's contents after allocation
  - Reasonable assumption in some languages
- Mutable objects introduce a new problem
  - Mature objects can point to the nursery
- Mature objects could keep nursery objects live
  - Need to scan mature space for minor collection

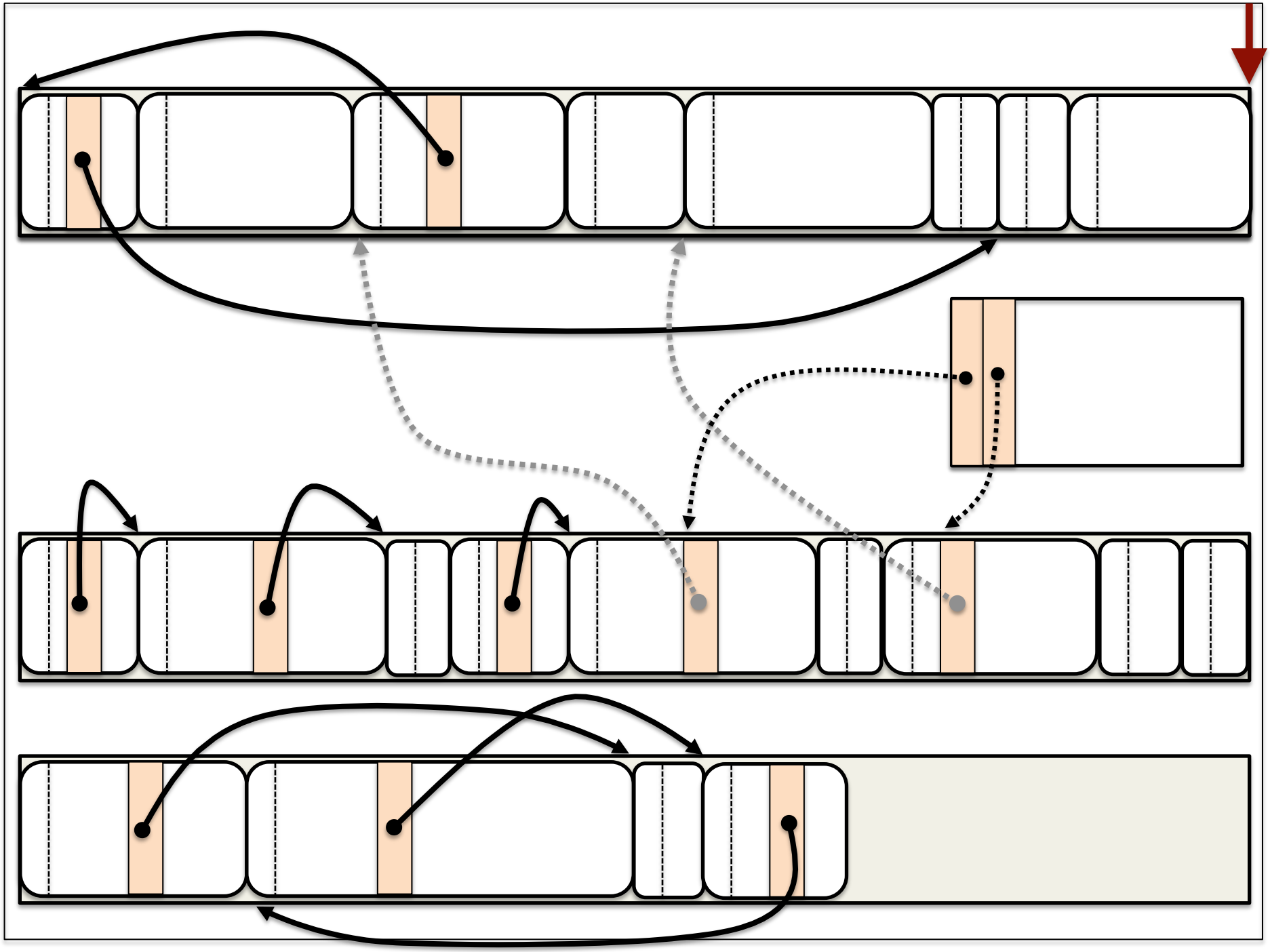




Remembered Sets

- Don't want to scan mature space every time
  - Lose much of the benefit of generational GC
- Track objects that may point to the nursery
  - Generally found to be uncommon
- Use that set as roots for minor GC
- Recall the write barrier concept from RC





Generational Write Barrier

- Insert small piece of code on reference writes
  - Maintain the remembered set
- Need only store references from old to new
  - We'll be scanning the nursery anyway
- Barrier can be heavily optimized
  - Still a source of some overhead
  - Hard to measure – somewhere around 2-3%

Generational Write Barrier

- If the source pointer is in the mature space
  - And the target pointer is in the nursery
    - Update the remembered set

Optimizing the Barrier

- There are two comparisons in the barrier
  - Is the source in the mature space?
  - Is the target in the nursery?

- We know the boundary between the regions
  - Checking for location just a comparison

Heap Layout

- The layout of the heap can help performance
  - Align regions on power of two boundaries
  - Gives a natural tag to regions
  - Find region by masking bits
- Nursery location can speed up allocation
  - Put the nursery at the end of the heap
  - Don't need to do the bump pointer comparison
  - Allocation ends at page fault

Fast and Slow Paths

- We can use heuristics to speed up the barrier
  - Assume that most pointer updates are in the nursery
  - If so, the first comparison normally fails
- Only inline the first comparison
  - If it succeeds, jump to another method
- Performance gain depends on the architecture
  - Size of the instruction cache
  - Effectiveness of branch prediction

Multiple Generations

- Can use more than two generations
  - Maintain some third space to promote from mature
  - Should cluster very old and long-lived objects
- Tends to have diminishing returns
- Limits options for mature space manager
  - Copying overhead eventually dominates

Pretenuring

- We can identify some long-lived objects
  - Static analysis of the code
  - Historical profiling information
  - Heuristics (large arrays)
- We can allocate those directly to mature space
  - Eliminate the copying overhead
  - Can play to mature collector's strengths

Permanent Generation

- Heap region for long-lived structures
  - Class data
  - Pretenured objects
  - Very large objects
- Something of a misnomer
  - Objects may turn out to be garbage
  - May be collected if memory is tight

Nursery Size

- Don't want nursery to be too small
  - Frequent minor collections
  - Objects need time to become garbage
- Don't want nursery to be too large
  - Minor collections become slow
  - More objects copied triggering major collections

Tuning Nursery Size

- Optimal size for throughput is large
  - Approximately half the heap
  - Does the least amount of work overall
- Optimal for responsiveness is much smaller
  - Sacrifice total work for short pauses
  - May need to experiment to find the best size
  - Could tune adaptively