Garbage Collector of XORP

史晓华
北京航空航天大学

2021-4-1

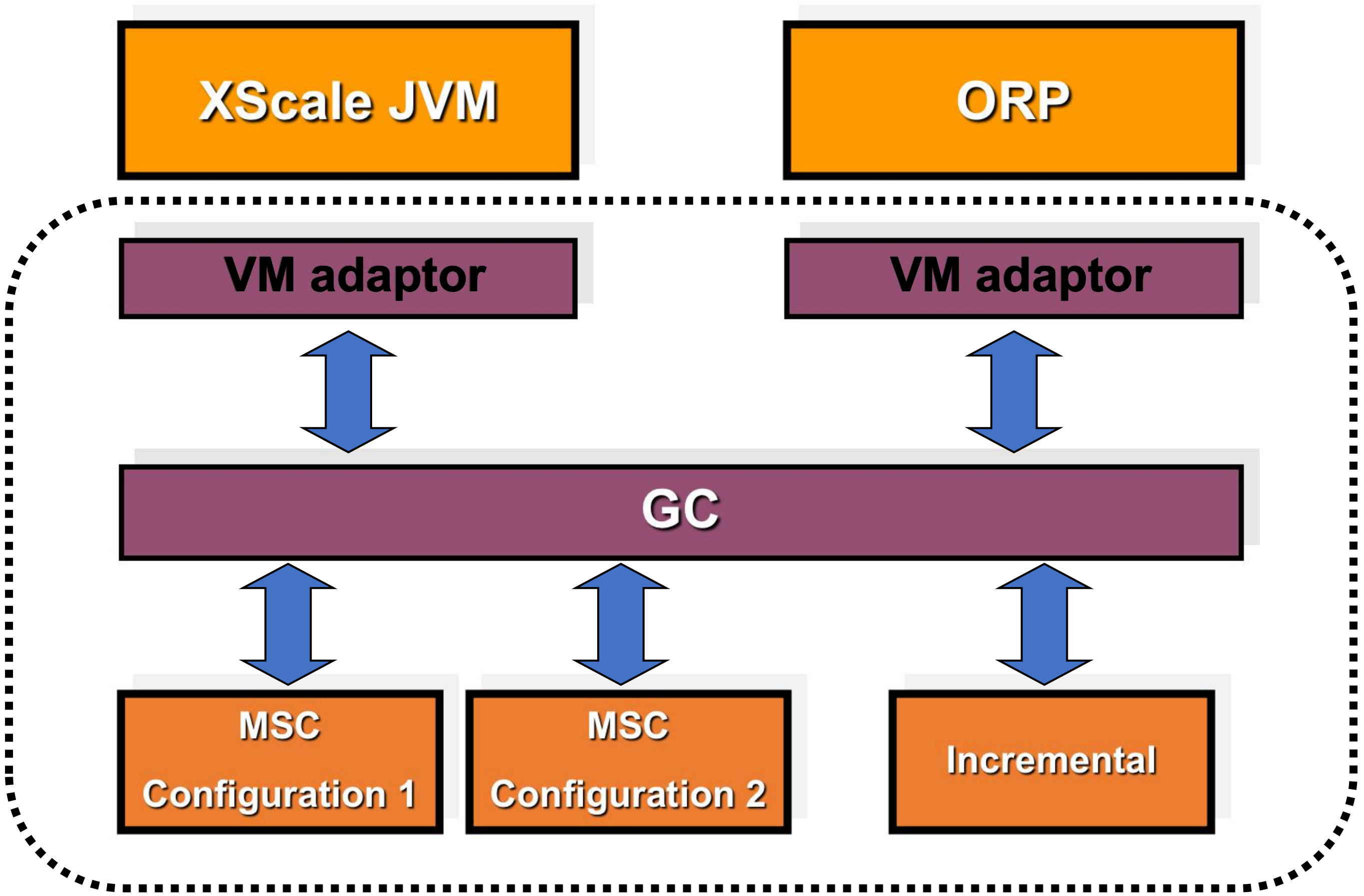
Design Considerations

- Small architecture
  - small and predictable memory footprint
  - everything in the heap
  - robust enough to handle partial failure
- Performance
  - fast allocation
  - friendly cache behavior
  - handle fragmentation if necessary
  - acceptable GC pause time
- Energy
  - GC controlled leakage energy optimization

Agenda

- Overview
- Infrastructure
- How it works?
    - allocation
    - reclamation
- Future works
- Summary

GC components



GC Overview

- Two-level allocation (chunks and blocks)
- Simple mark-sweep-compact GC
  - Optimize for small footprint
  - Compact data structures, minimize wastes, reuse if possible
  - Allocate and reclaim not only Java objects, but also VM objects, code
- Accurate GC
  - Corporation among VM/interpreter/JIT
- GC with well defined interfaces
  - Configurable and tailorable to different scales of applications
    - heap-chunk-block (e.g., #chunk=1, #block-per-chunk=1, regress to single block heap)
  - Adaptable to different VMs
- GC friendly to developers
  - Verbose, tracing, verification, and profiling support

Agenda

- Overview
- Infrastructure
- How it works?
  - allocation
  - reclamation
- Future works
- Summary

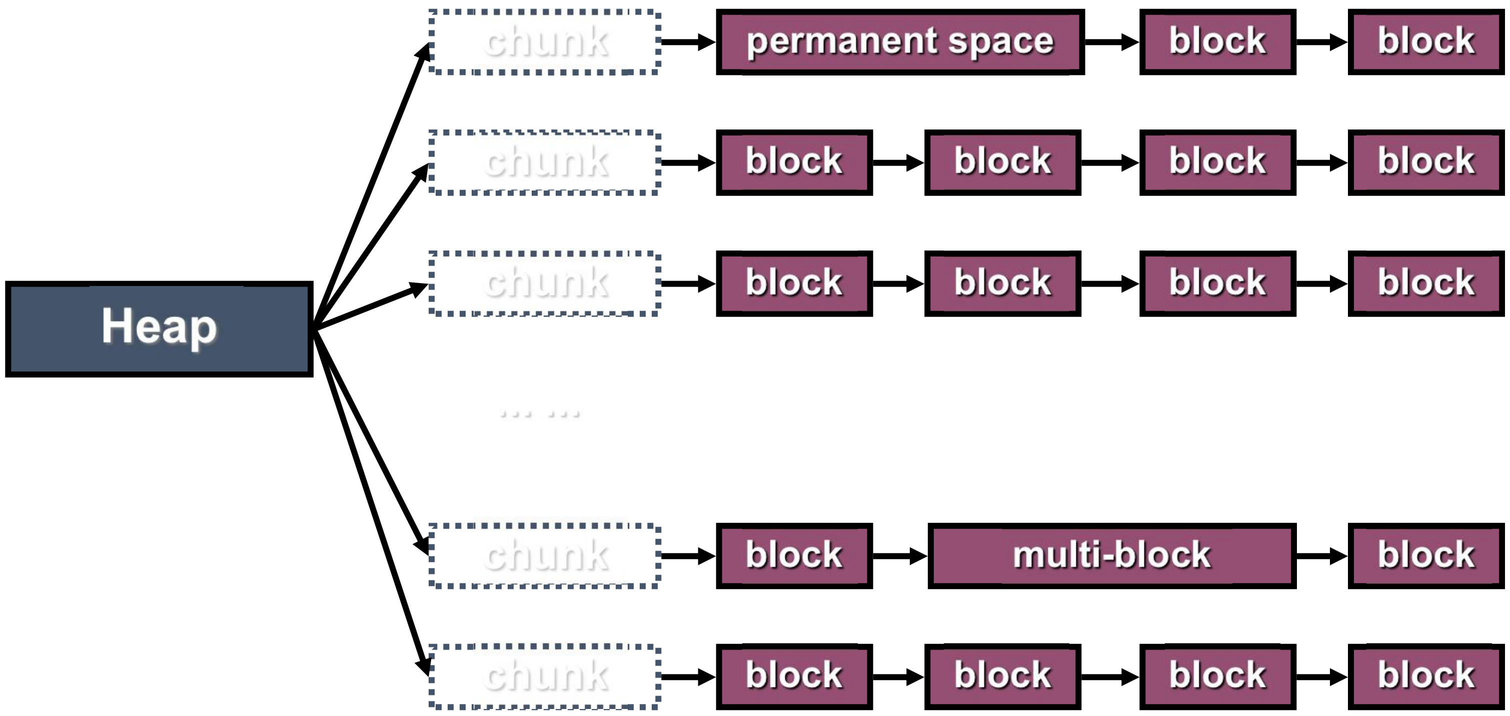
GC Small Architecture

- Small memory footprint
  - compact data structures, minimize wastes, reuse if possible
- Predictable memory usage – everything in heap
  - GC data structures, VM objects, Java objects, code etc.
- Flexible hierarchy: heap–chunk–block
  - configurable to different scales of application
    - e.g., #chunk=1, #block-per-chunk=1, regress to single block heap
- Componentized to modules
  - core GC module, tailorable GC module, VM adaptor

Memory Hierarchy

- Heap
  - A series of consecutive chunks
  - At the beginning of first chunk, there's a permanent space to place core GC data structures
- Chunk
  - A series of consecutive blocks as a low level allocation unit
  - Configurable chunk size (power of 2)
  - May map to memory banks
    - Free up chunks during GC → turn off corresponding memory banks
- Block
  - Configurable block size (power of 2)

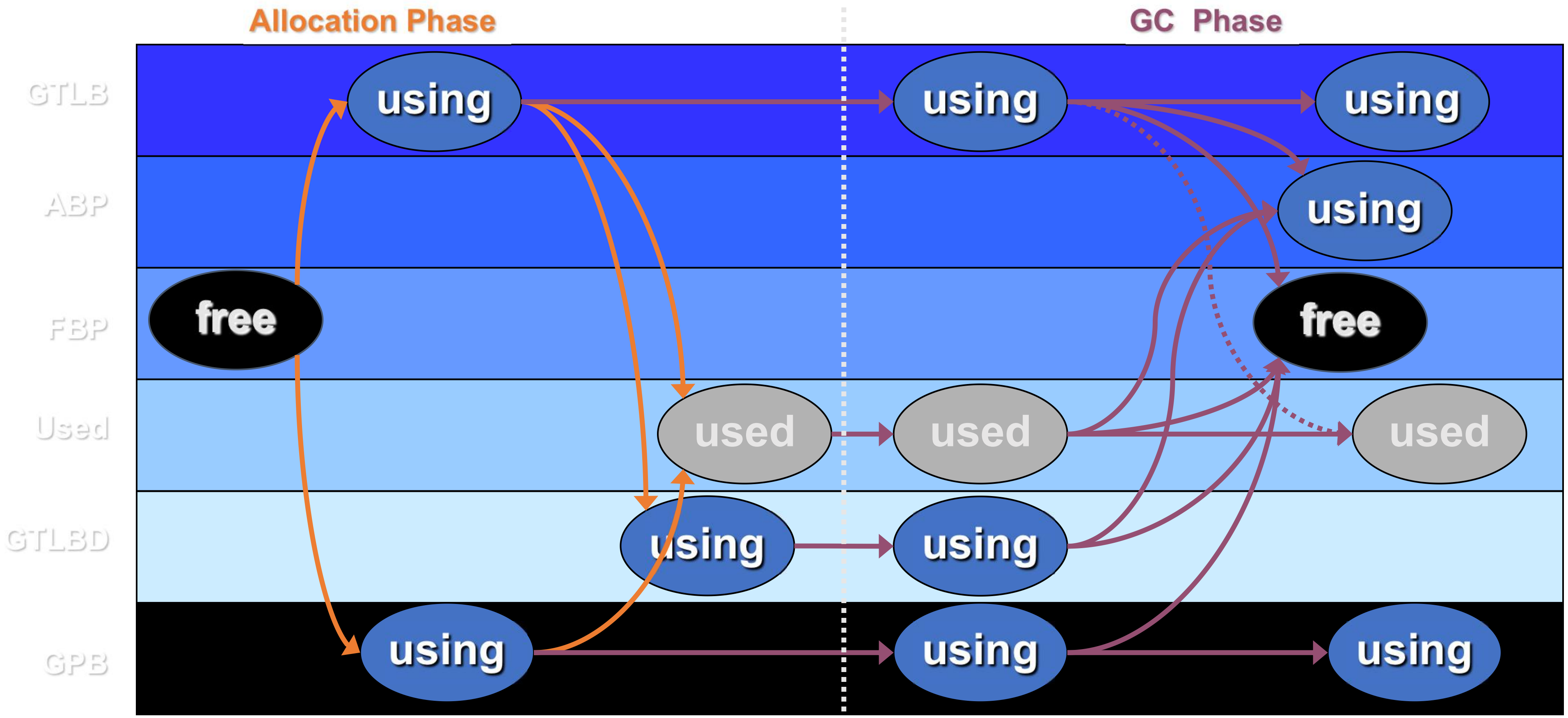
Physical Heap Layout



Logical Heap Layout

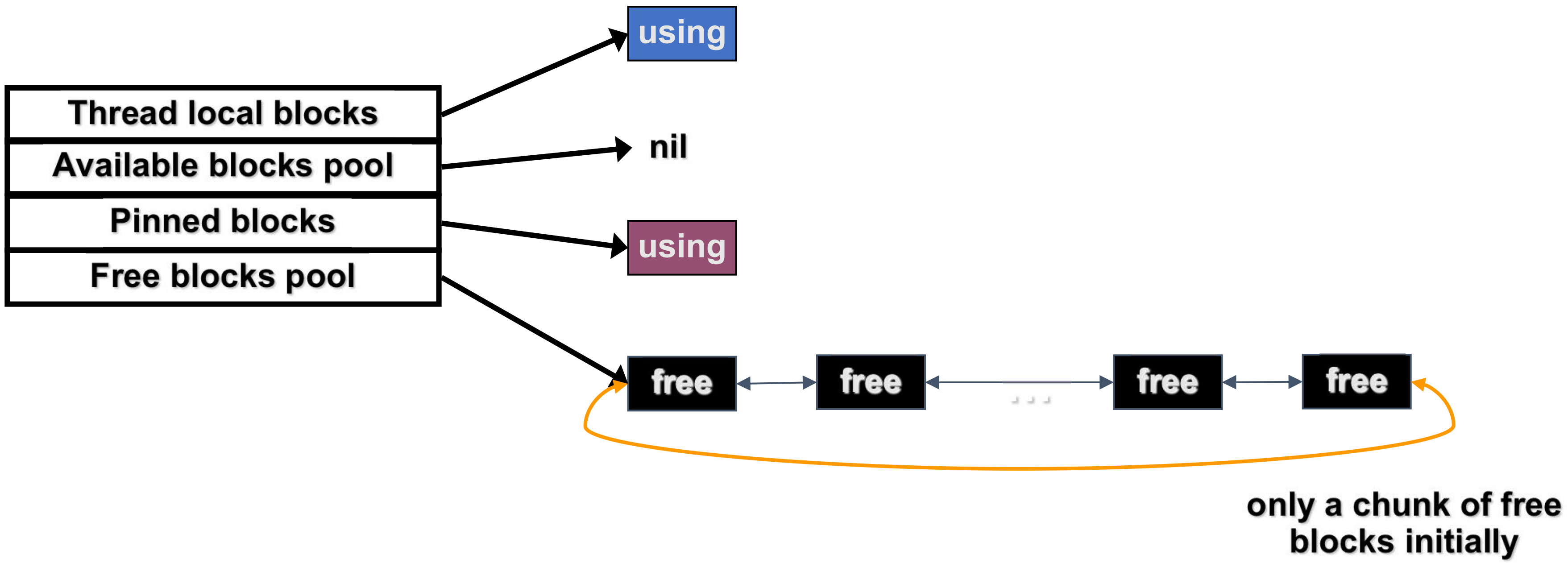
- Classify available blocks into categories (free space)
  - Pinned
    - Non—movable objects and large objects
  - Non—pinned
    - Thread local blocks
      - Lock-free allocation
    - Global blocks
      - Still in use
      - Must be assigned to thread local blocks before using its space for allocation
  - Free
- Used blocks (no free space)

Transition Between Groups



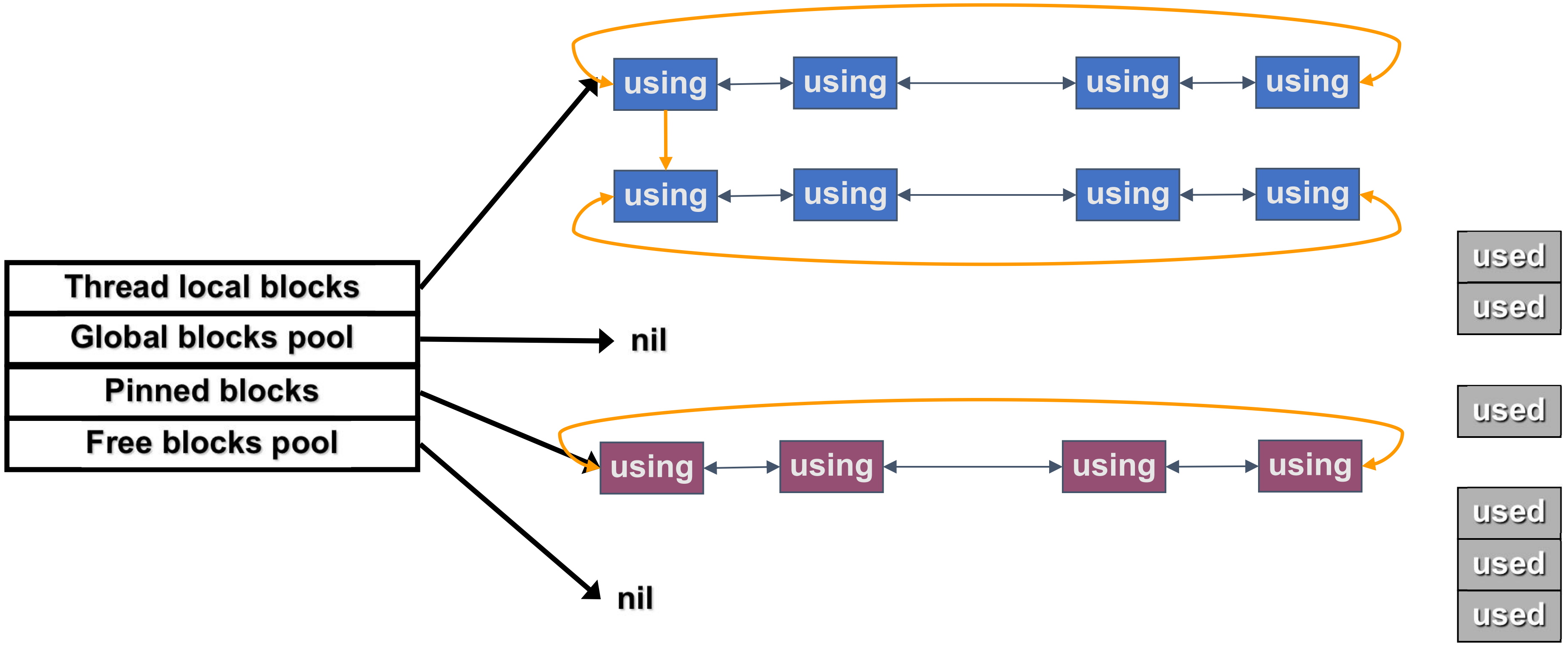
GTLB: Groups of Thread-local Blocks
GPB: Group of Pinned Blocks
FBP: Free Blocks Pool
ABP: Available Blocks Pool
GTLBD: Groups of Thread-local Blocks of Dead threads

Logical Heap Layout
— At the beginning of execution



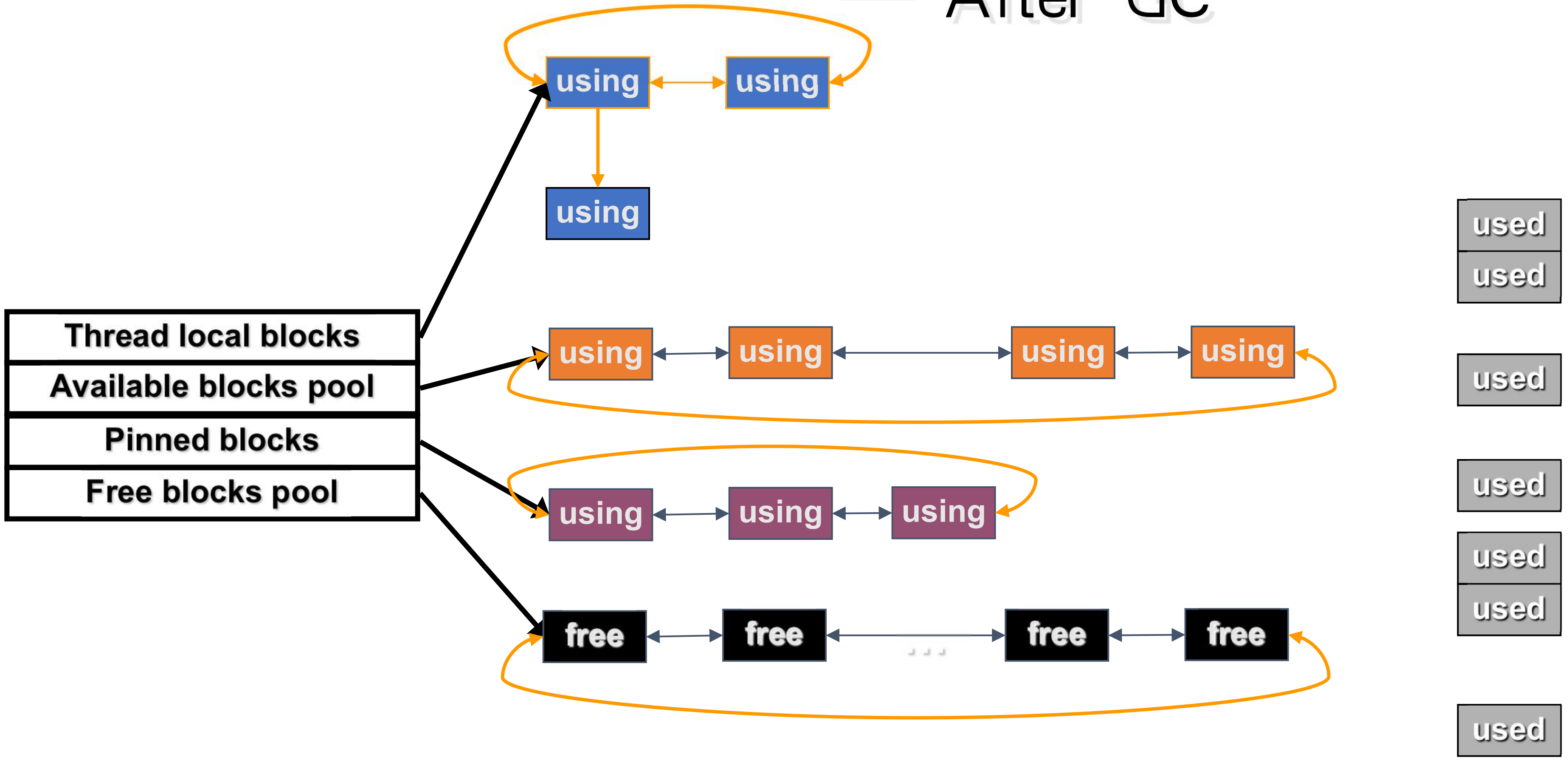
Logical Heap Layout

— Before GC



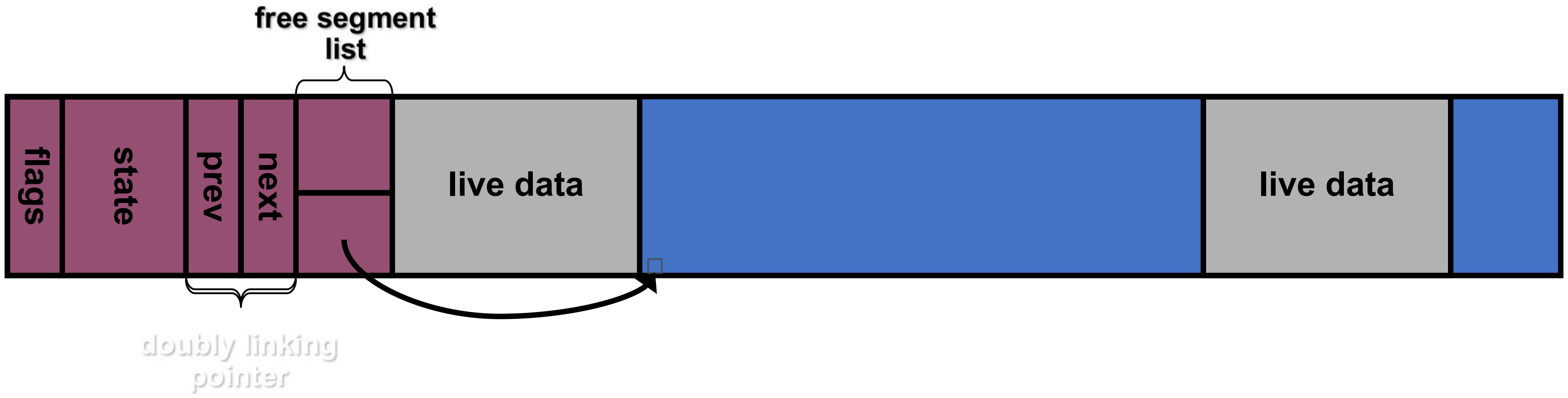
Logical Heap Layout

— After GC

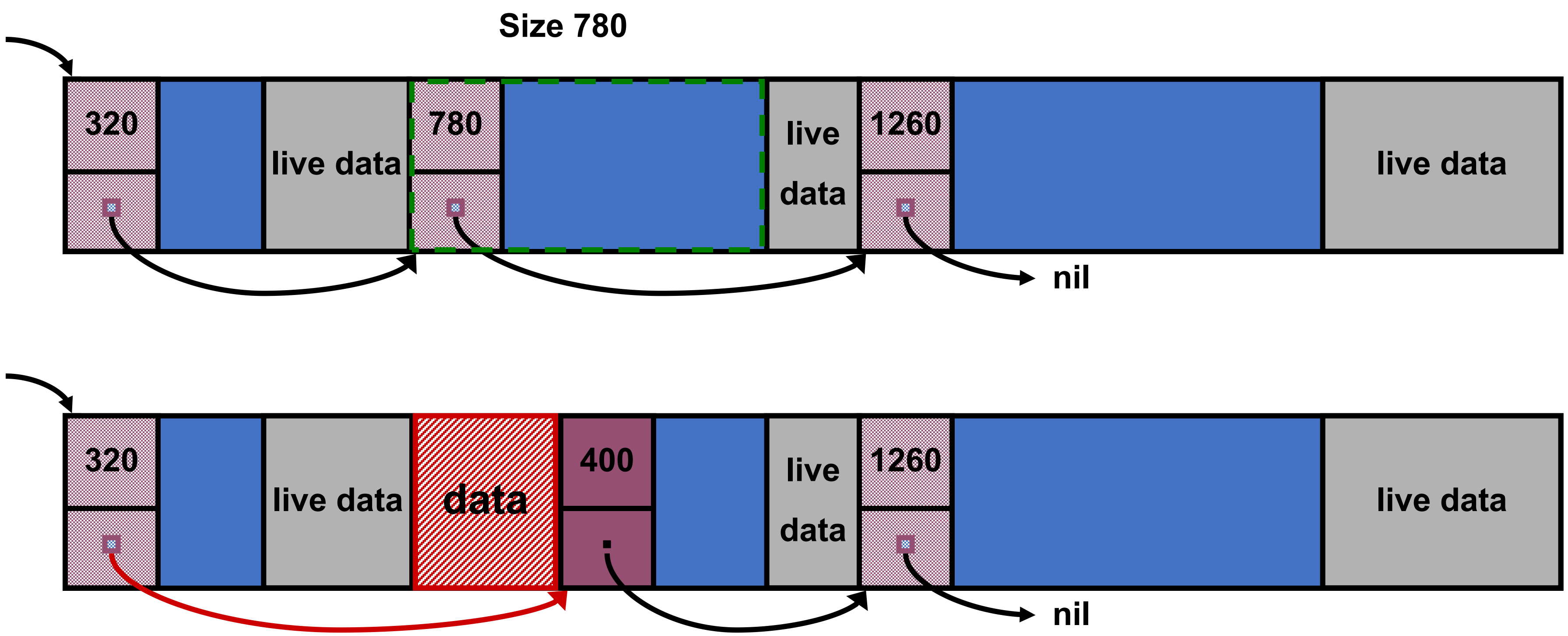


Block Layout

- Block Info
  - block flag and state, link pointers, etc.
  - mark bits array
- Block data
  - linked free segments, two schemes:

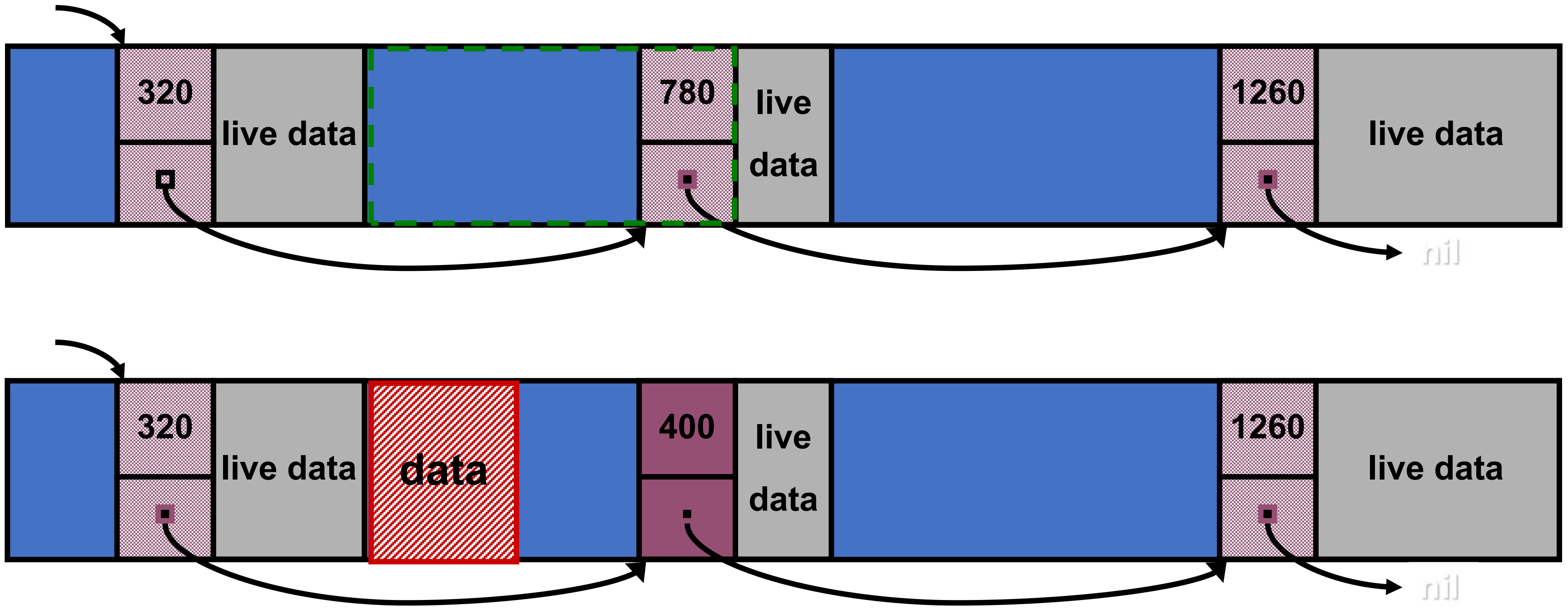


Block data (1)



Size 780
320
live data
780
live
data
1260
live data
nil
320
live data
data
400
live
data
1260
live data
nil

Block data (2)



320
live data
780
live
data
1260
live data
nil
320
live data
data
400
live
data
1260
live data
nil

Locate object's block info

- Block info is at the beginning of block
  - Given the object address, you can get block info through a single bit mask
- For large objects that need multiple consecutive blocks to accommodate
  - "multi-block" is allocated and linked into group of pinned blocks
  - Only the first block of a "multi-block" has block info
  - Large objects are placed into "multi-block" right aligned, and the leading space can still be used for other small pinned objects
  - For each object in "multi-block", you can still get block info through the same bit mask

GC Utilities

- Verbose information of coarse granularity
- Tracing GC activities at finer granularities
  - log into trace file
  - feed trace file into GC to help debugging
- Verifier
  - Verify temporary data like root sets, mark stack
  - verify heap/chunk/block data at any snapshot
- Profiler
  - collect and print statistical information of heap/chunk/block
  - track type and size distribution of heap objects, identify prolific types

Agenda

- Overview
- Infrastructure
- How it works?
  - allocation
  - reclamation
- Future works
- Summary

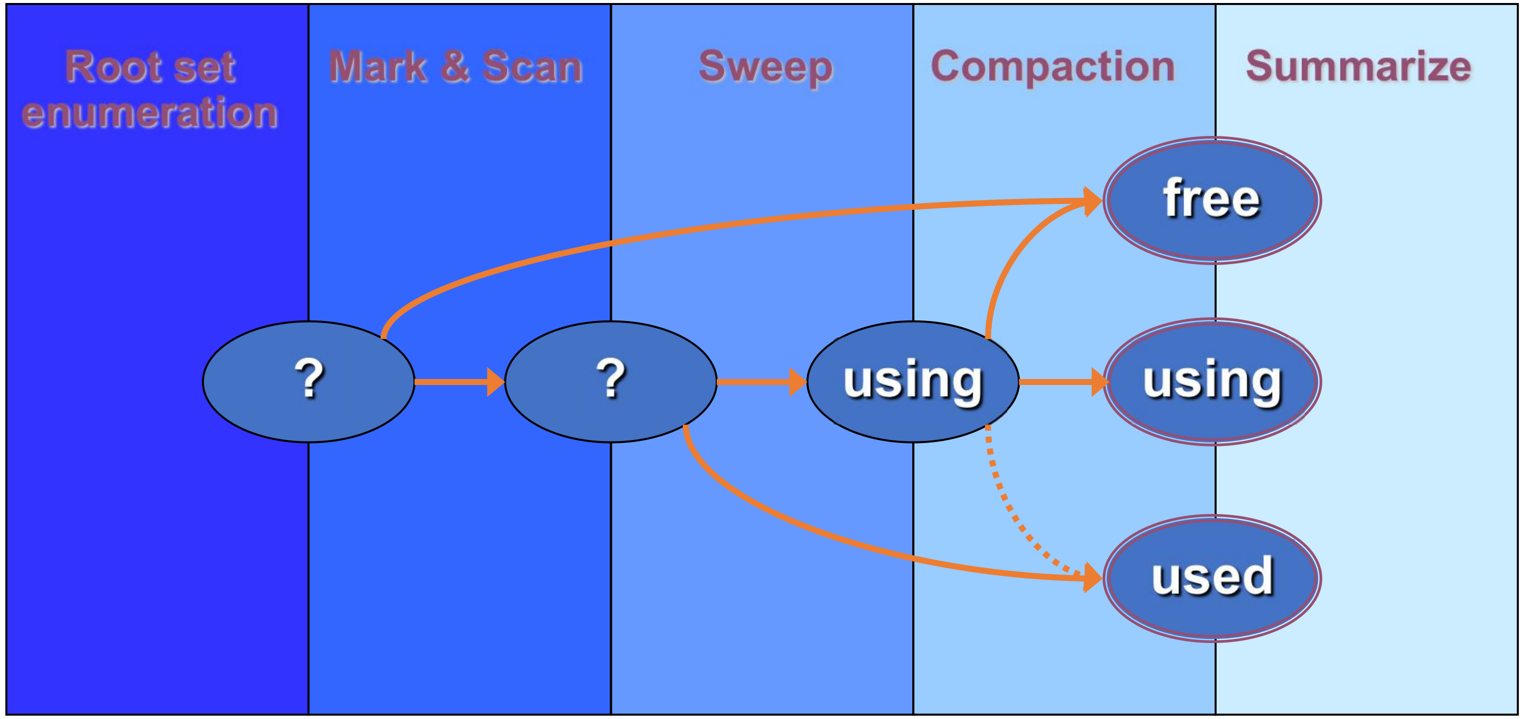
Allocation

- Fast and slow path
  - Fast lock-free allocation that never invokes GC
  - Slow allocation if fast allocation fails
- Allocate objects with special constraints via slow path
  - pinned objects and large objects
  - VM/Java objects with finalizers, weak/soft/phantom references, etc. in future
- Refine policy that
  - avoid GC if possible
  - balance the block distribution between normal blocks and pinned blocks, and between groups of thread local blocks (no starvation)

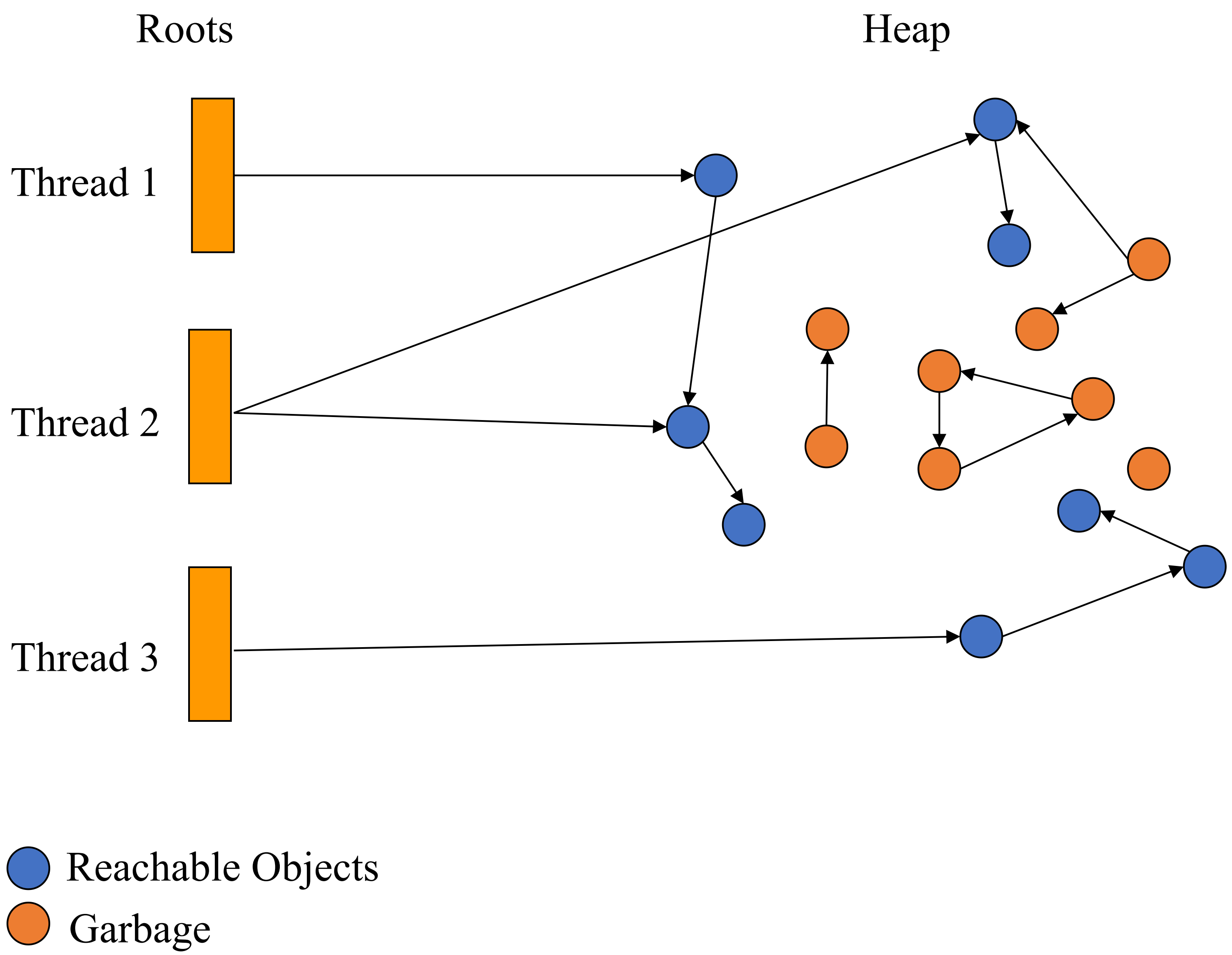
Heap Reclamation

1. Prepare heap reclamation (reset states)
2. Call on VM to enumerate root set
3. Select compacting chunks
4. Mark and Scan
5. Sweep
6. Incremental compaction
   - redirect pointers into compacted regions
7. Summarize heap reclamation (finalize heap states)

Block State Transition During GC



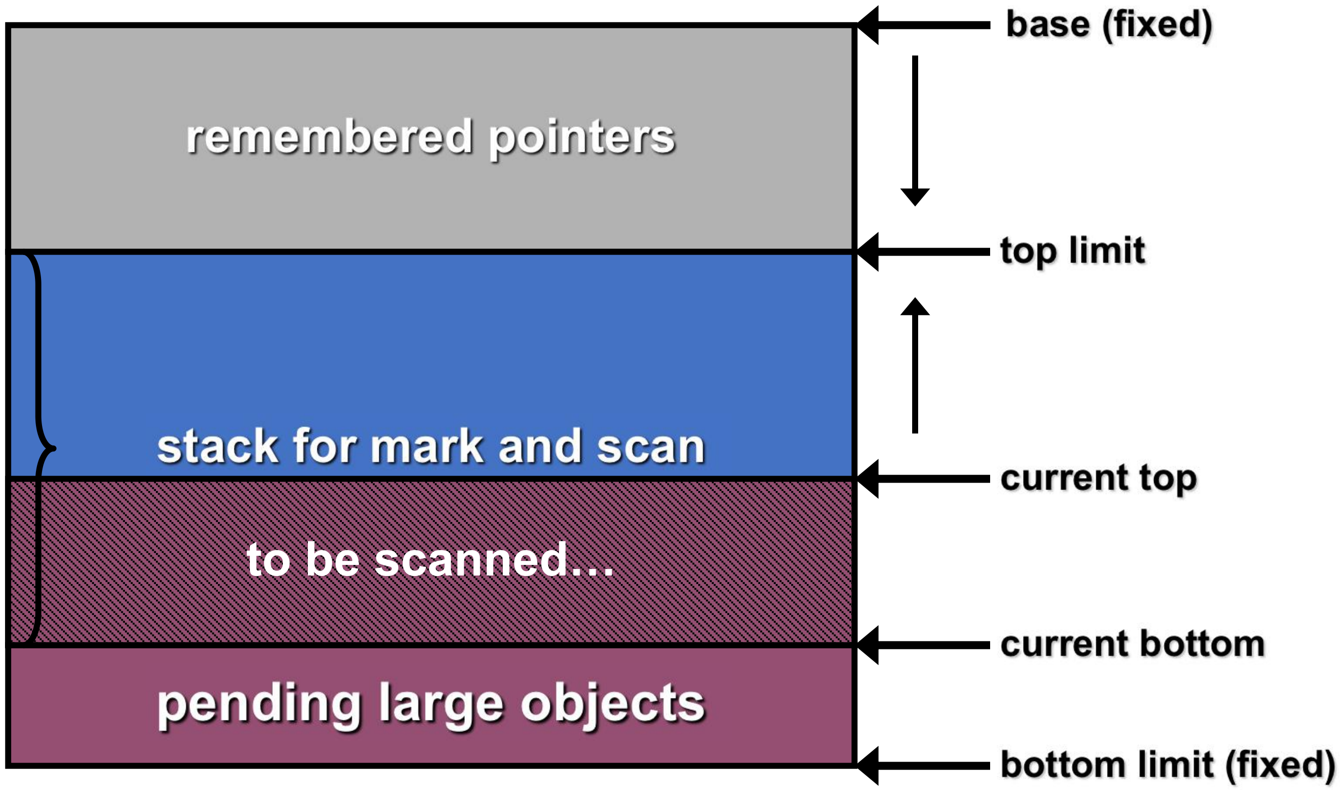
How Mark/Sweep Works



Mark and Scan

- Mark table per block
  - Use external mark bit to ease object header design
- Allocate mark stack in heap permanent space on the fly
  - Enumerate root sets directly into mark stack
  - Mark blocks "dirty" if they have live objects being marked
    - ease sweeping for these blocks that should be free
  - Remember pointers into to—be—compacted regions
    - Record the pointers in mark stack
    - Mark chunks/blocks "remembered" if they have such pointers and demarcate "remembered regions"
  - Minimize mark stack depth
    - Traverse object graph by node, not by edge
    - Handle those objects with large amount of references specially

Mark Stack



remembered pointers
stack for mark and scan
to be scanned...
pending large objects
base (fixed)
top limit
current top
current bottom
bottom limit (fixed)

Sweep

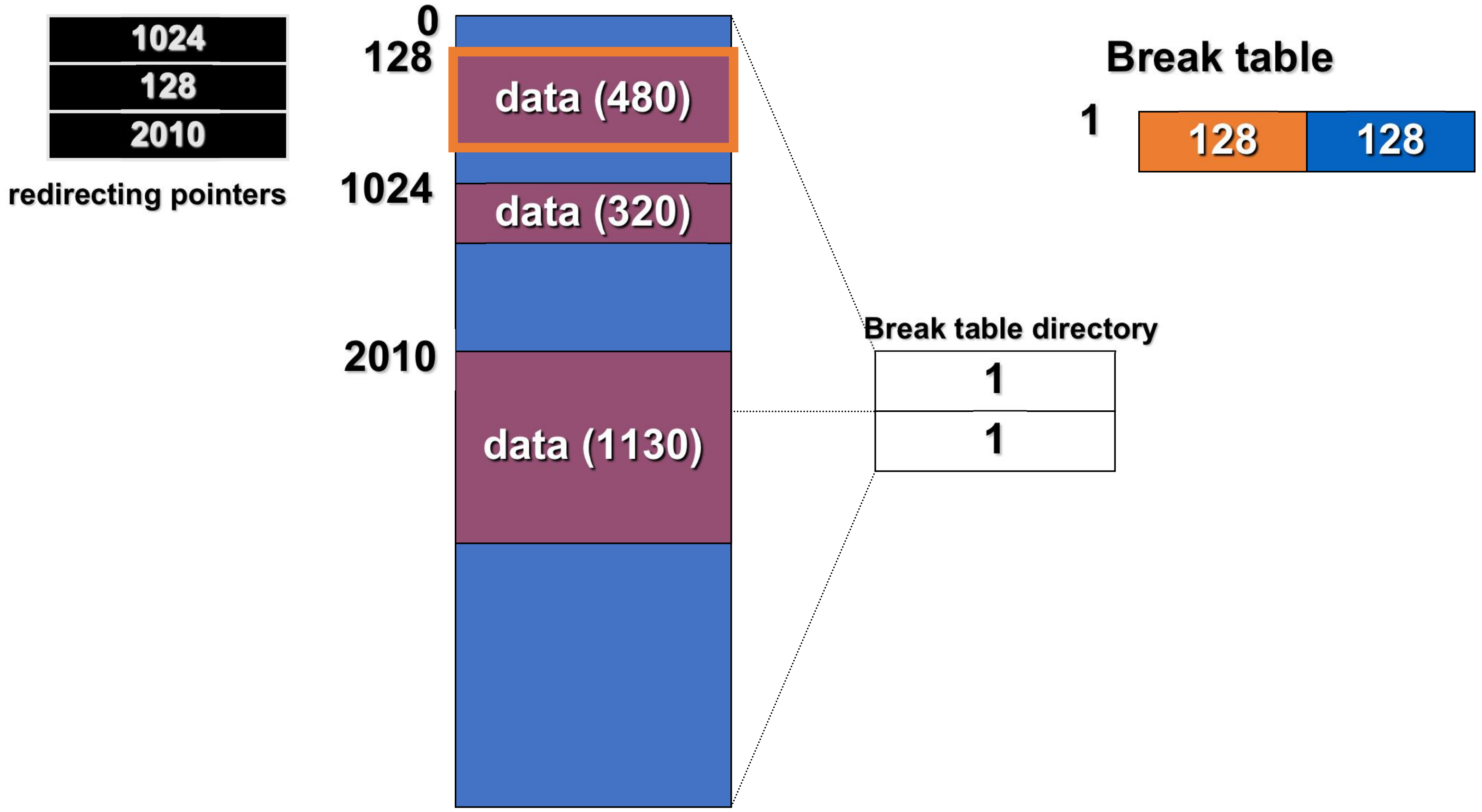
- Only sweep if the block is marked as “dirty”
- Block-scope one pass sweeping
  - scan mark table (bits manipulation)
  - generate linked free segments
  - clearing free segment eagerly for simplicity

Compaction

- Cross block compaction
  - select compaction candidates heuristically
  - incremental compaction to reduce GC pause time
  - apply optimized break-table based compaction block by block
    - sliding compaction that maintains allocation order
- Redirect remembered pointers/regions to new positions in compacted regions

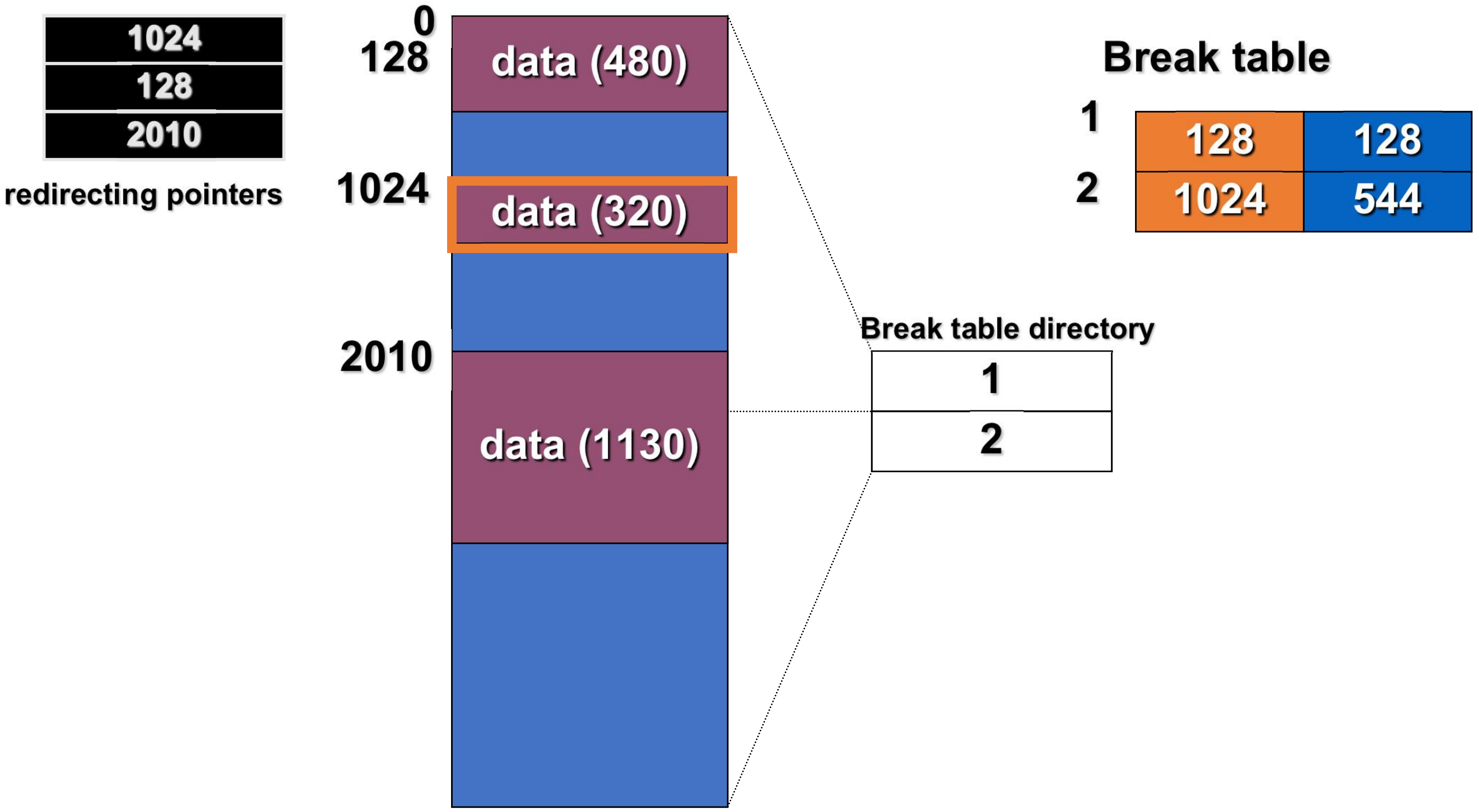
How Compaction works?

— 1st pass

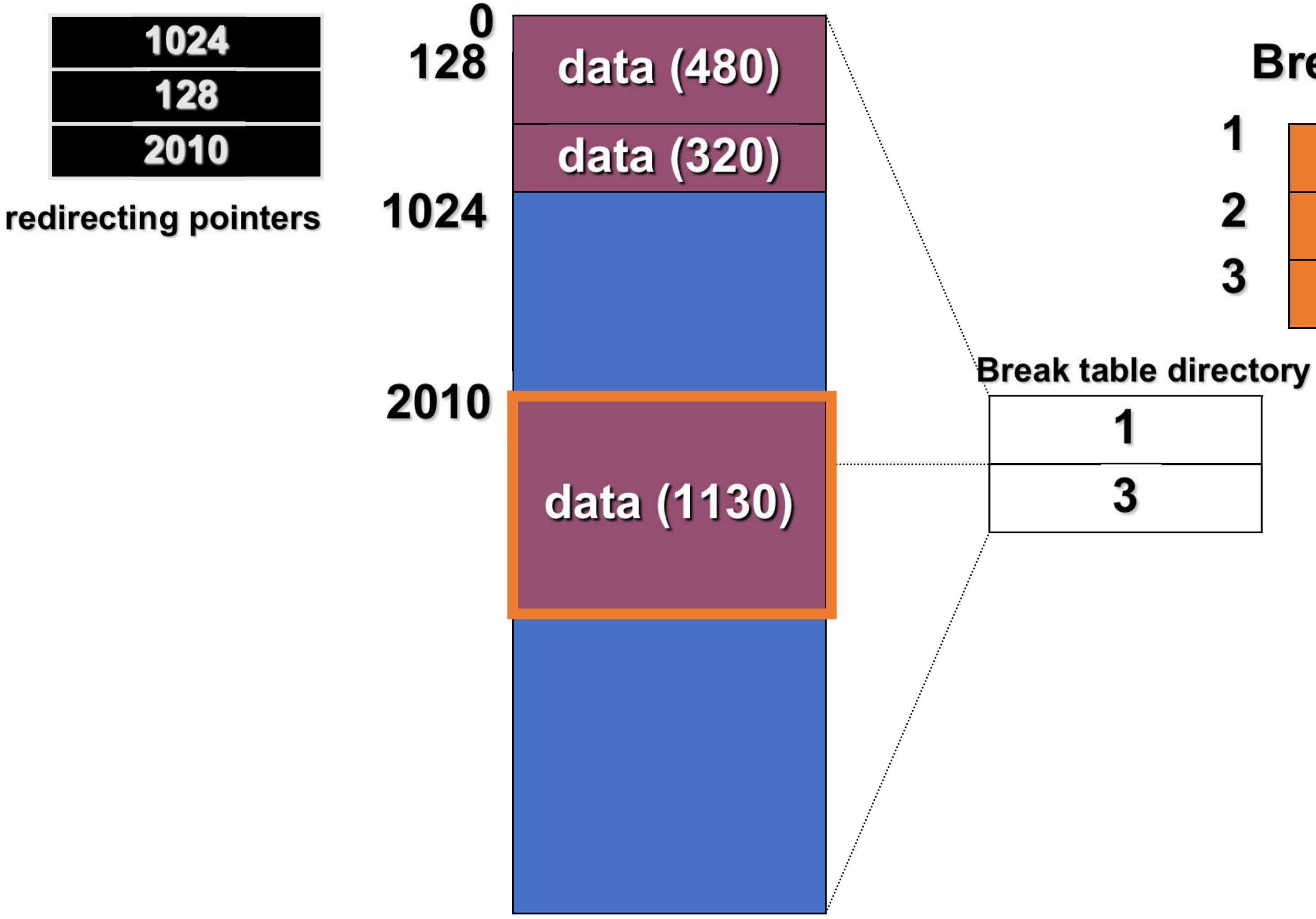


How Compaction works?

— 1st pass



How Compaction works? — 1st pass

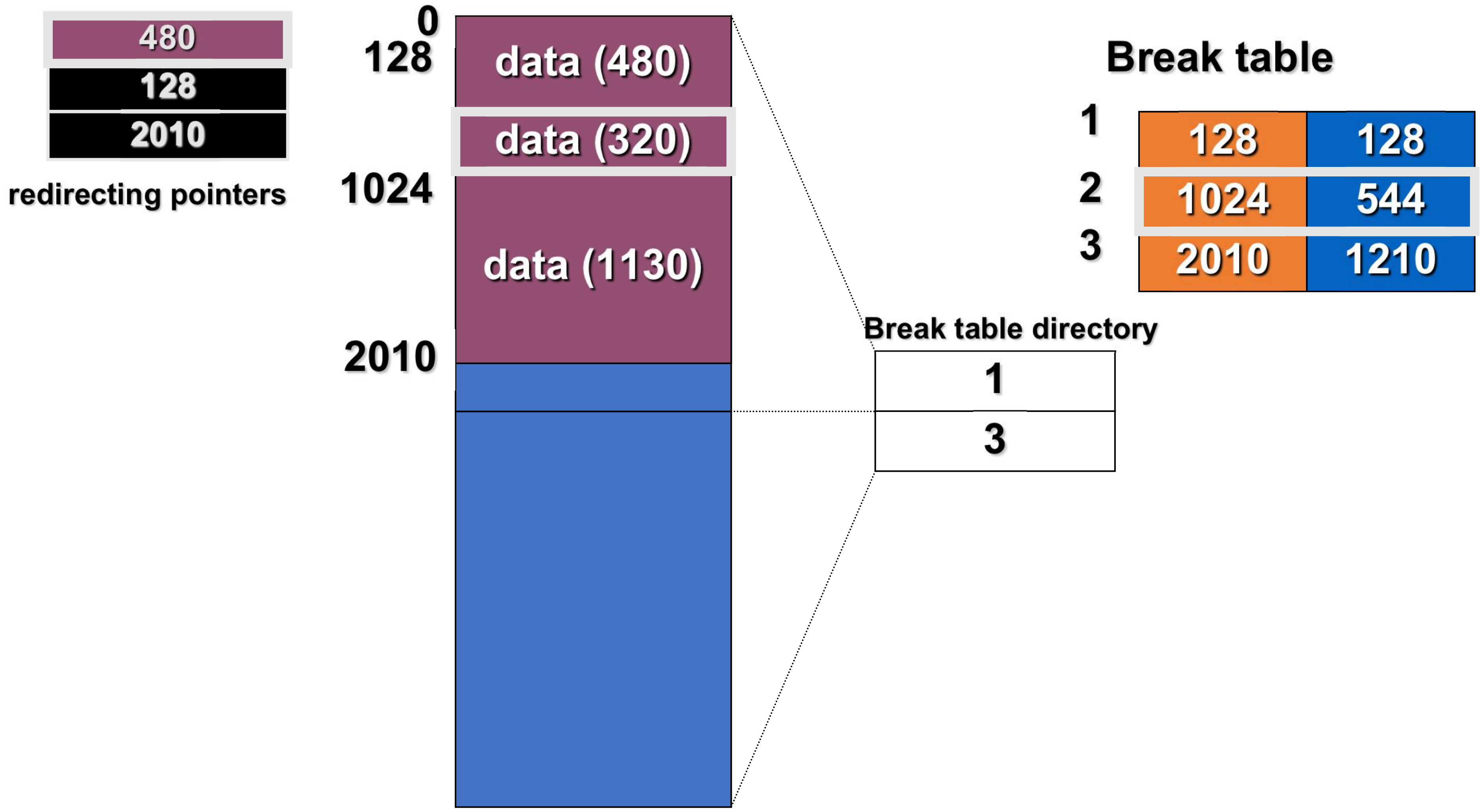


Break table

1	128	128
2	1024	544
3	2010	1210

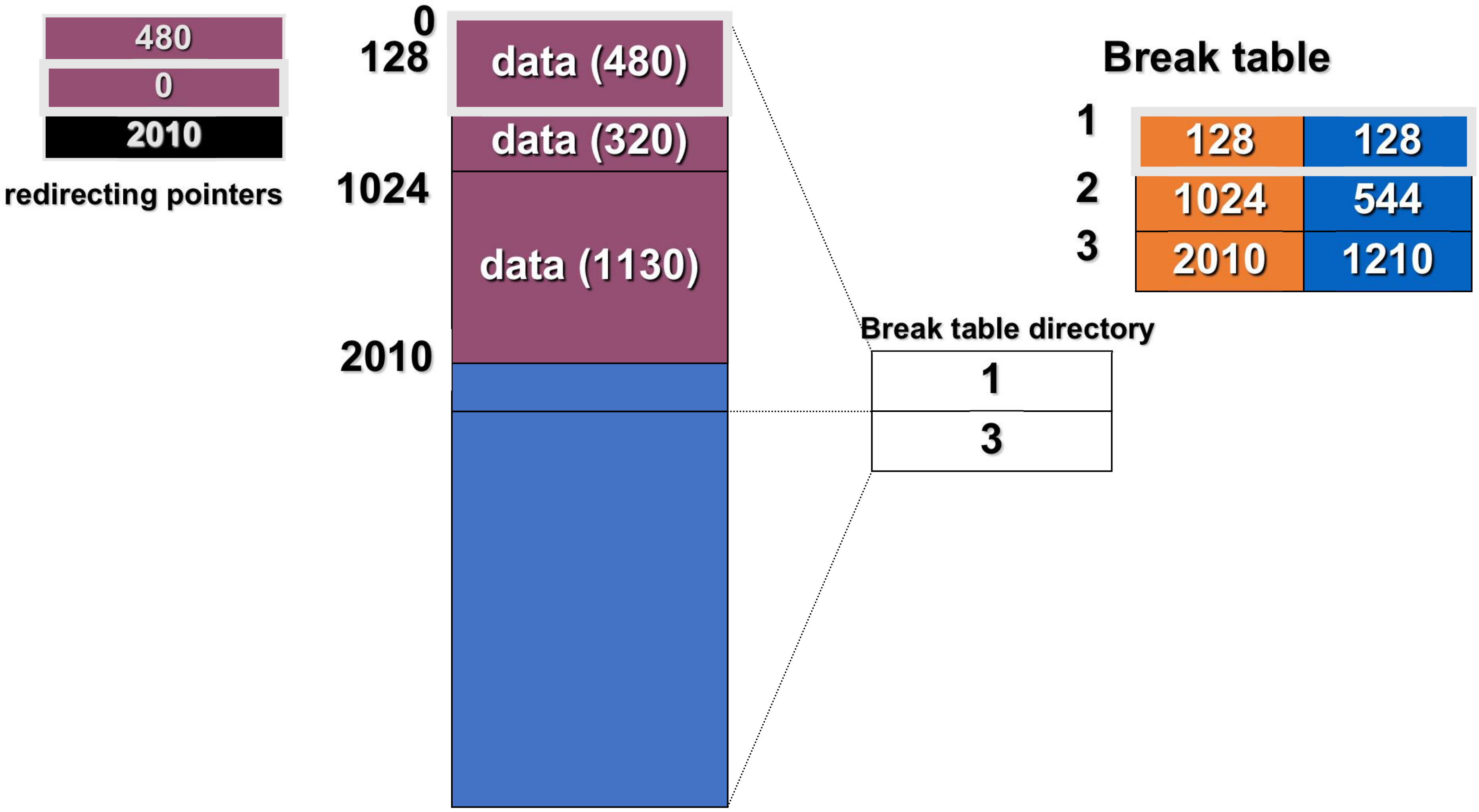
How Compaction works?

— 2nd pass: redirecting pointers



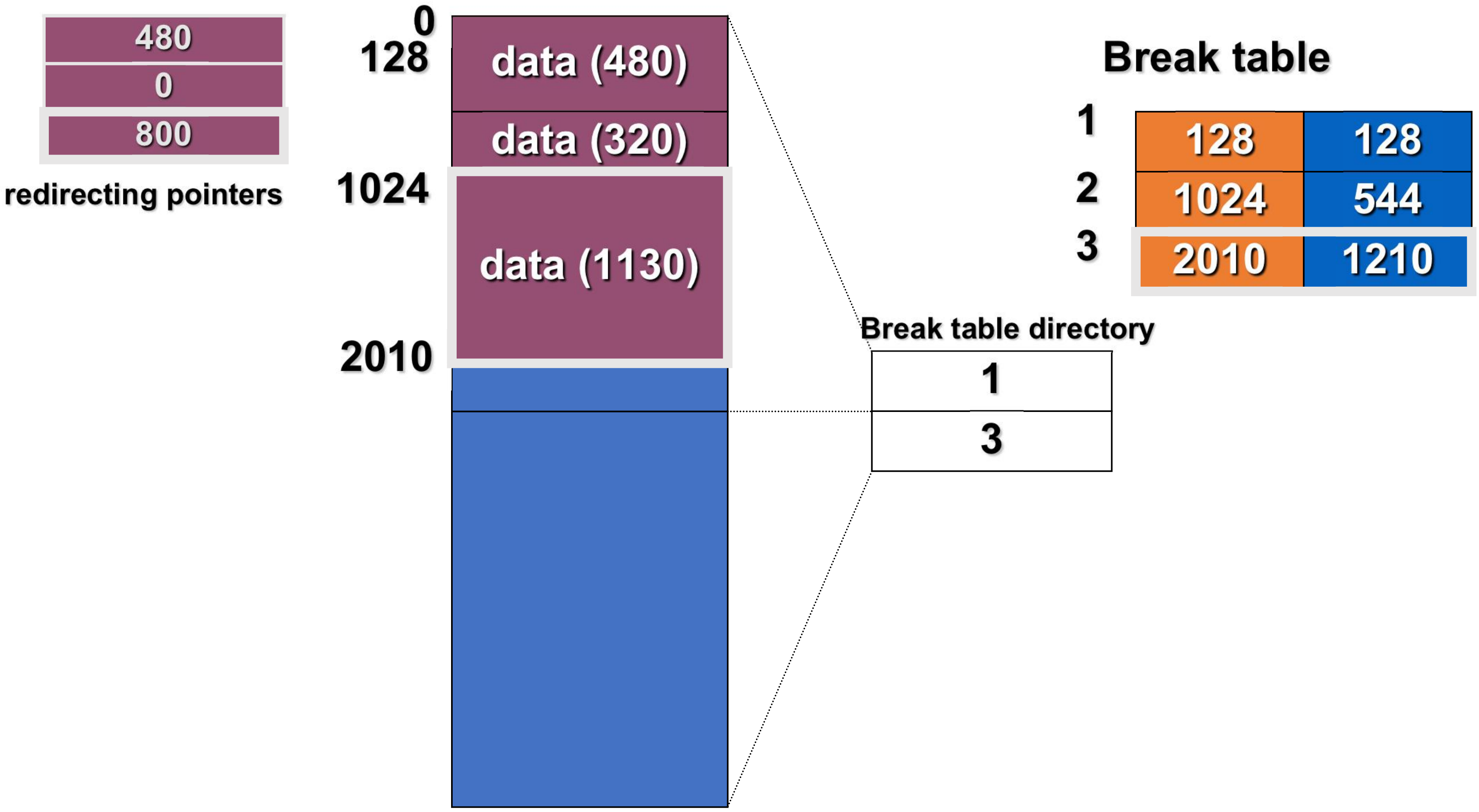
How Compaction works?

— 2nd pass: redirecting pointers



How Compaction works?

— 2nd pass: redirecting pointers



Space for Compaction Data Structure

- Break table
  - find space of table size in permanent space and free segments in previous blocks, instead of free segments in this block, so rolling or sorting can be avoided
- Break table directory
  - reuse this block's mark table space
- Remembered pointers
  - store in permanent space
  - for those demarcated region in "remembered" blocks, need one pass scanning