Understanding Low Latency JVM GCs

Jean-Philippe Bempel

@jpbempel

Understanding JVM GC: Advanced!

- Concurrent Marking
- Shenandoah
- Azul’s C4
- ZGC
- How to choose a GC algorithm?

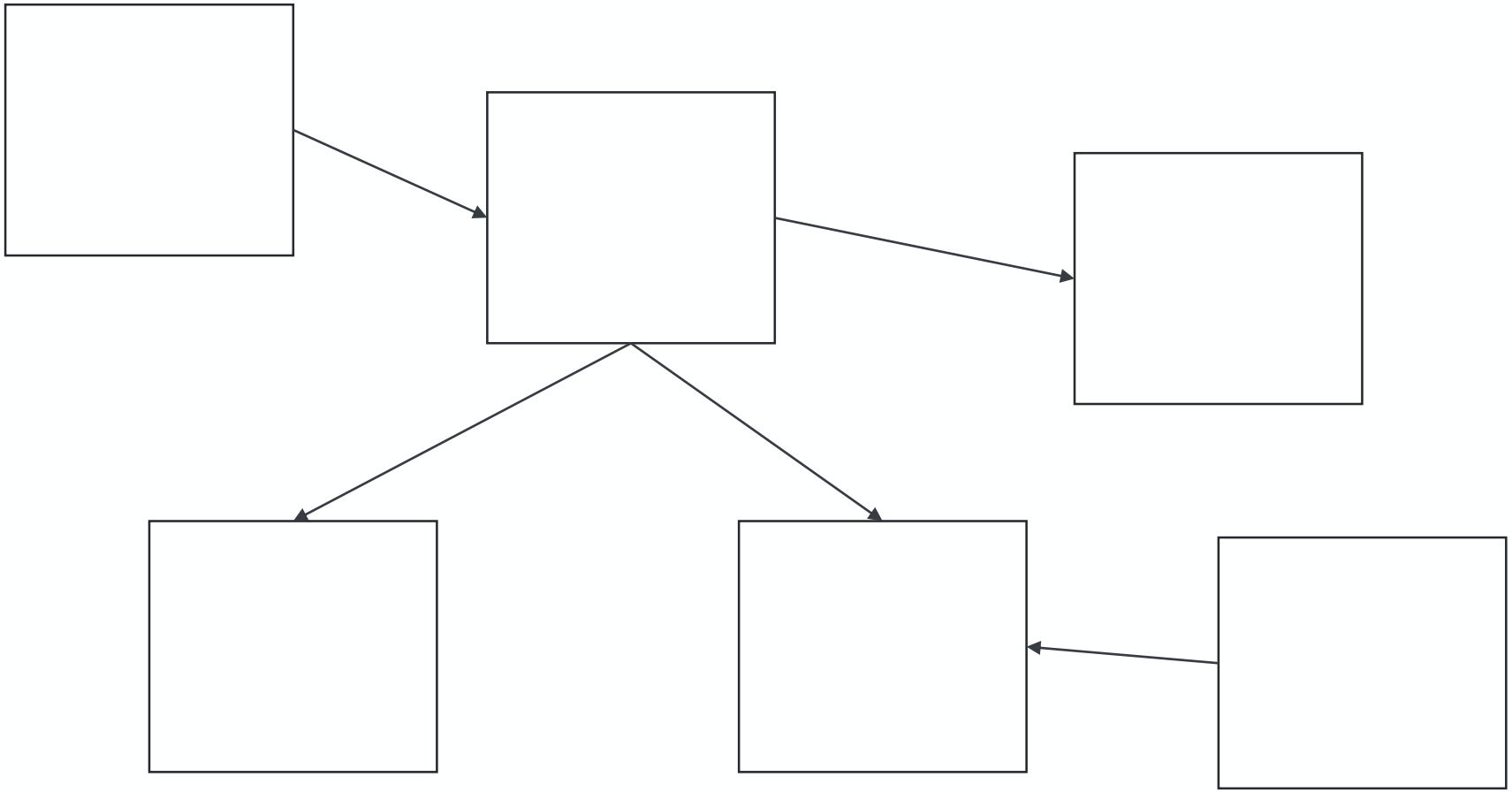
Concurrent Marking

criteo

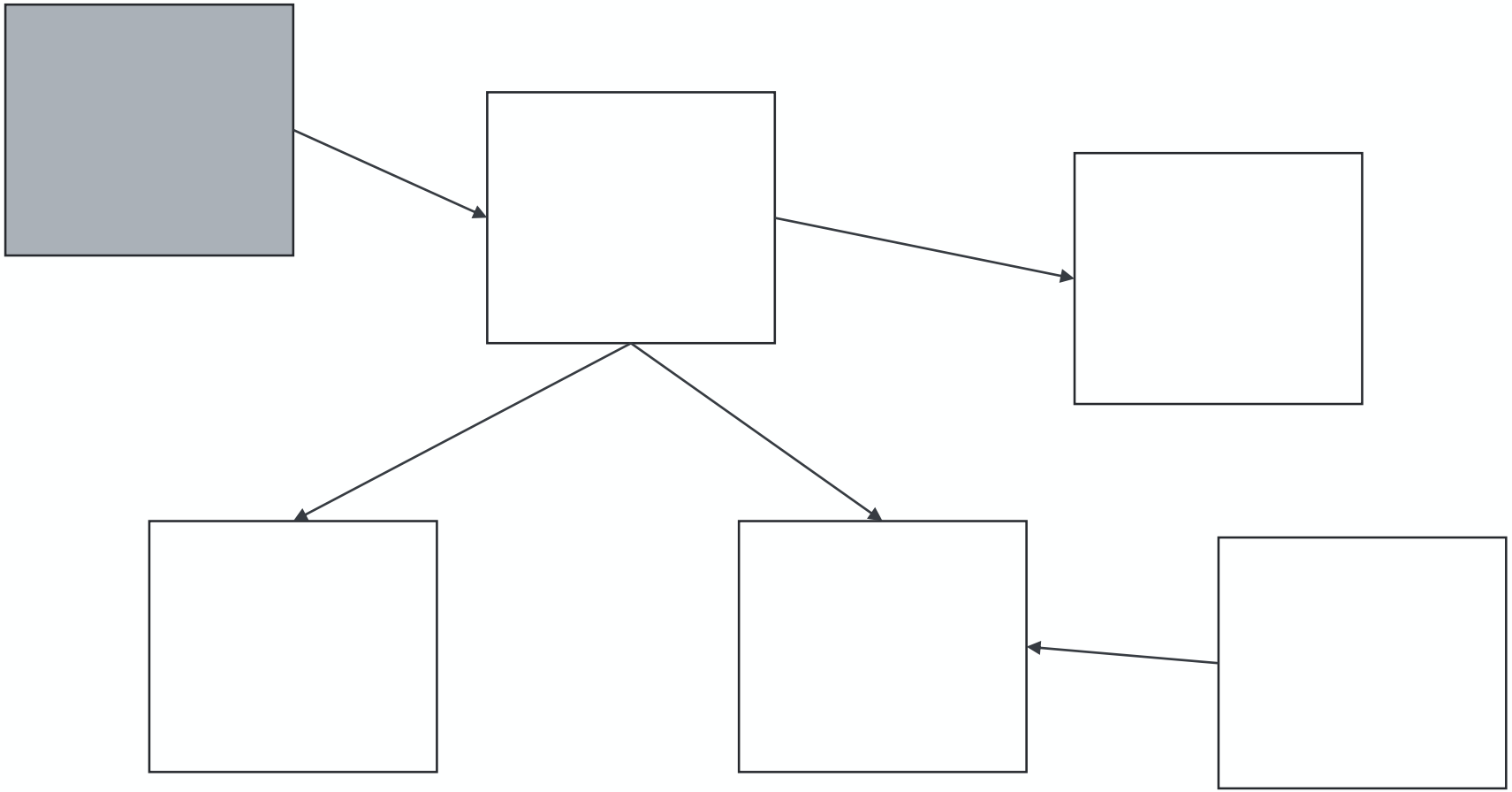
Concurrent Marking

- Used in CMS & G1 algorithms already and by all low latency GCs
- Try to mark the whole object graph concurrently with the application running
- Based on Tri-color abstraction & Snapshot-At-The-Beginning algorithm
  - Used by CMS, G1, Shenandoah

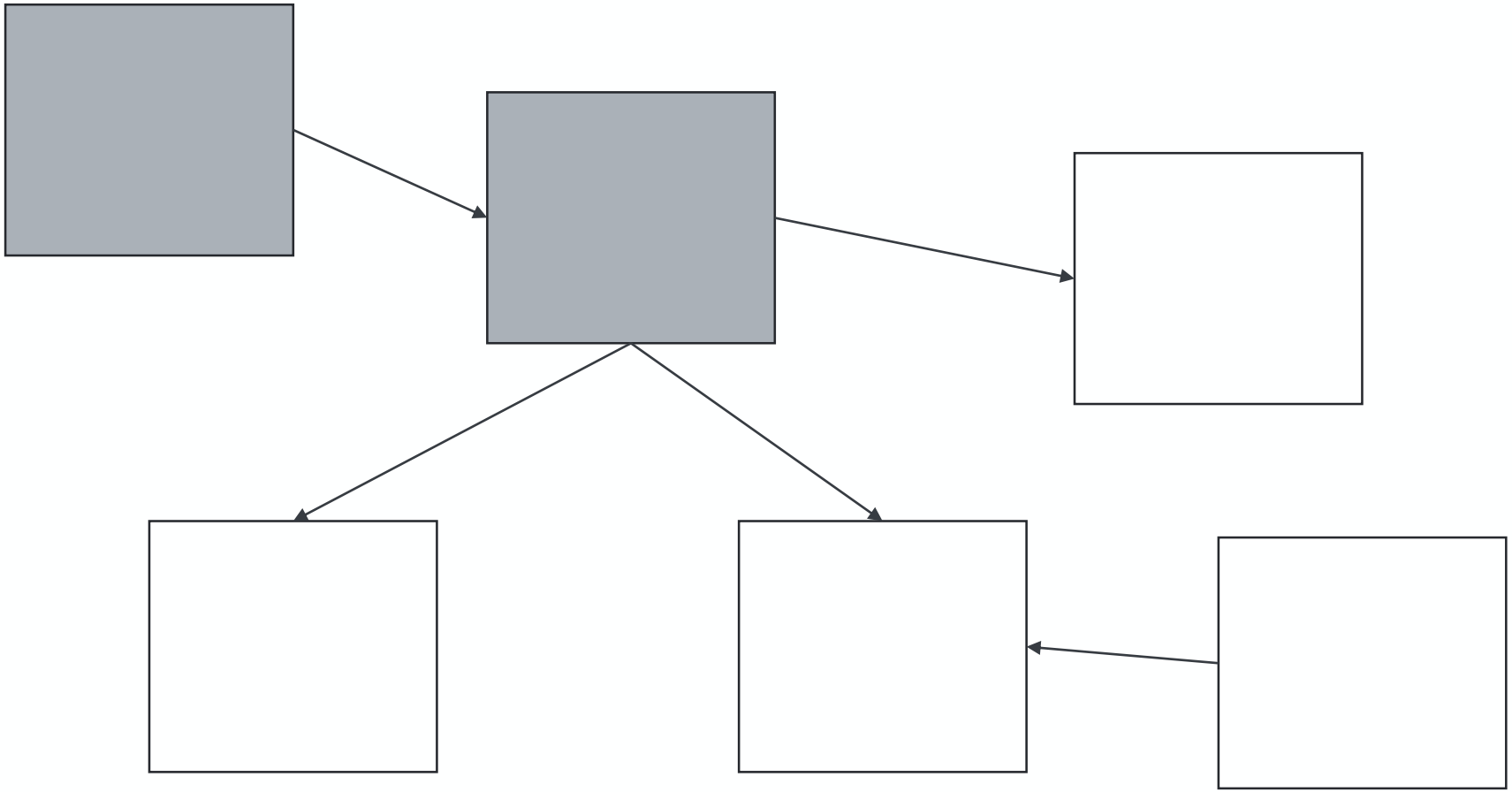
Concurrent Marking: Tri-Color Abstraction



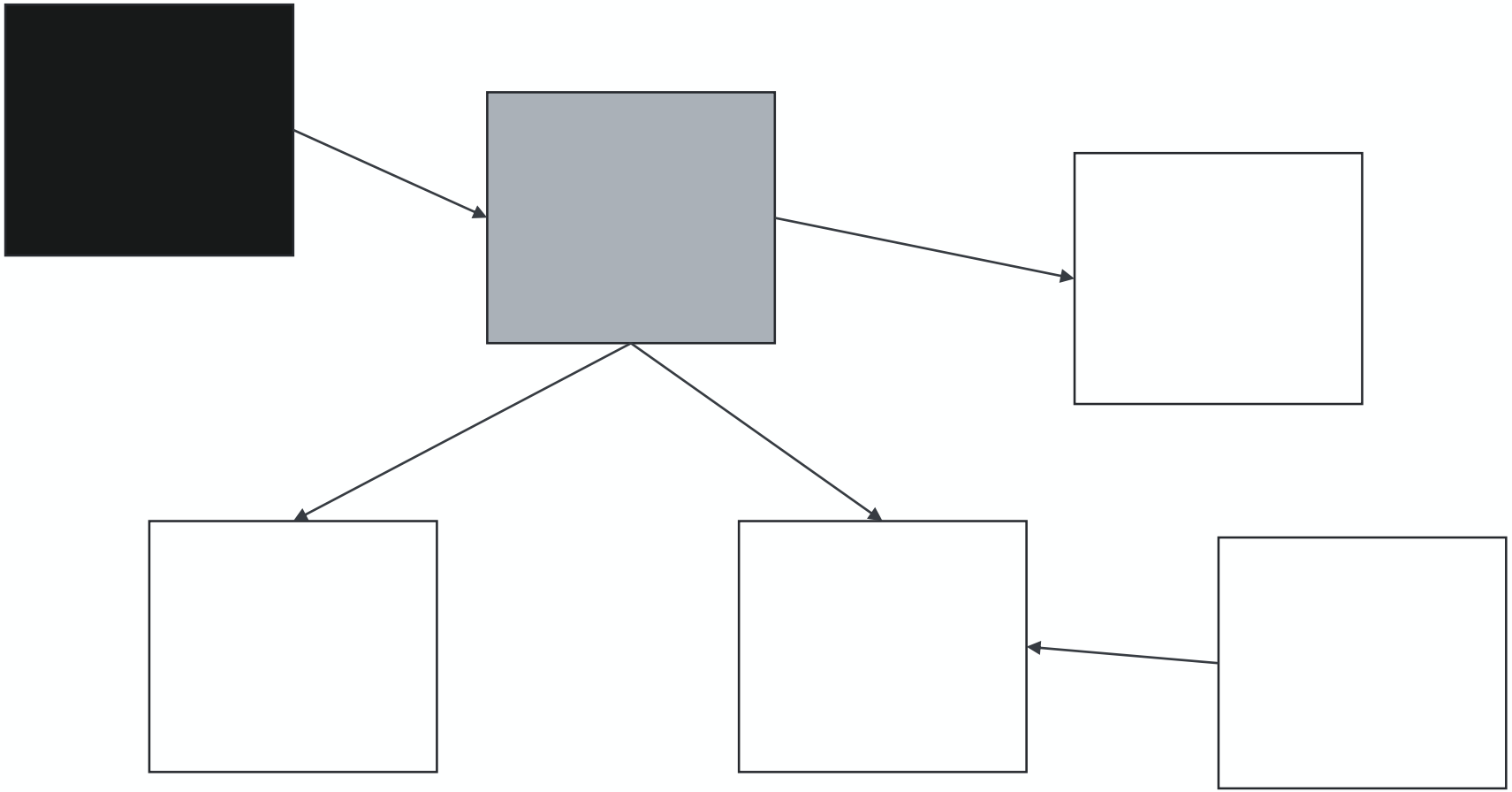
Concurrent Marking: Tri-Color Abstraction



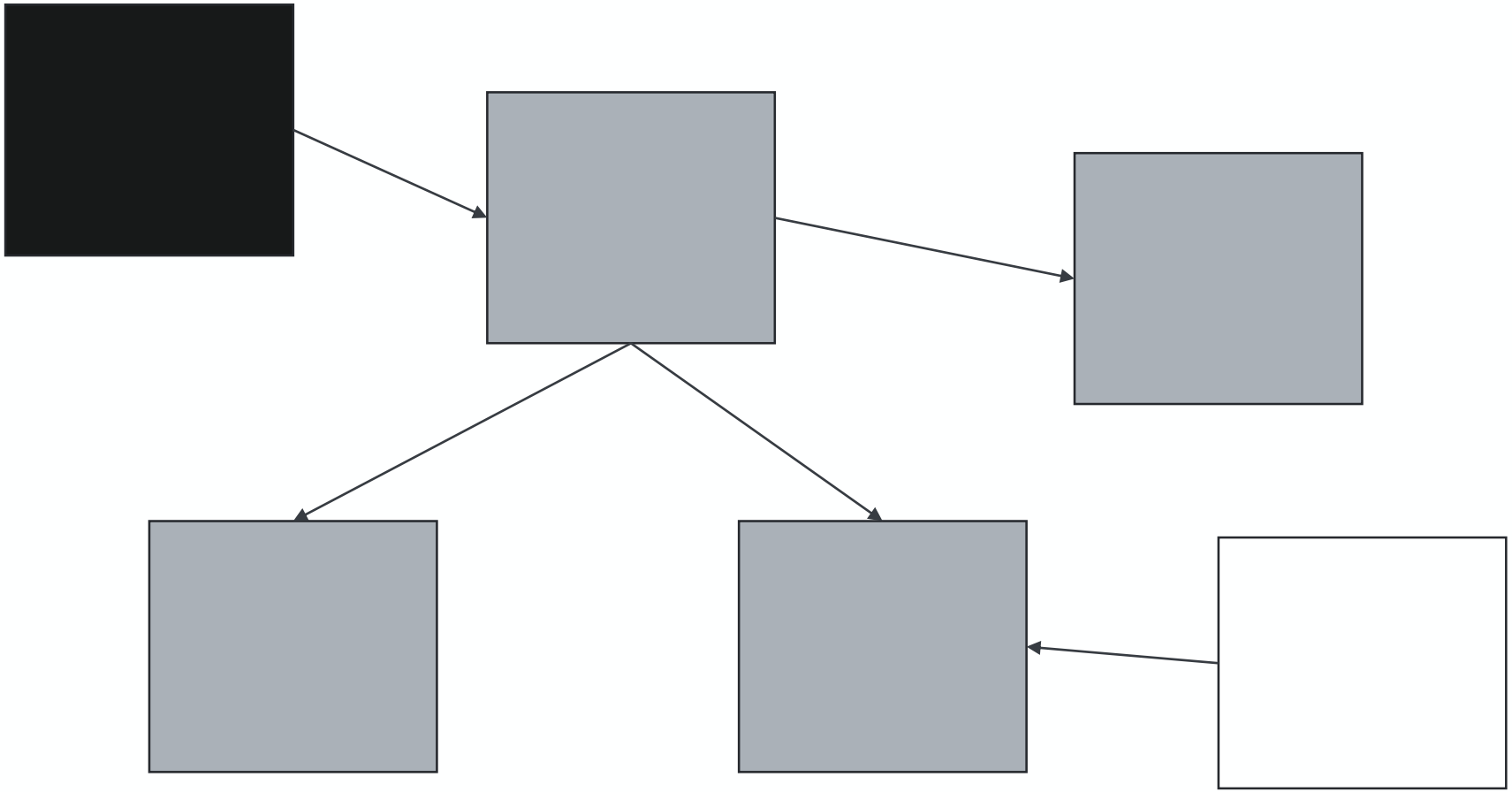
Concurrent Marking: Tri-Color Abstraction



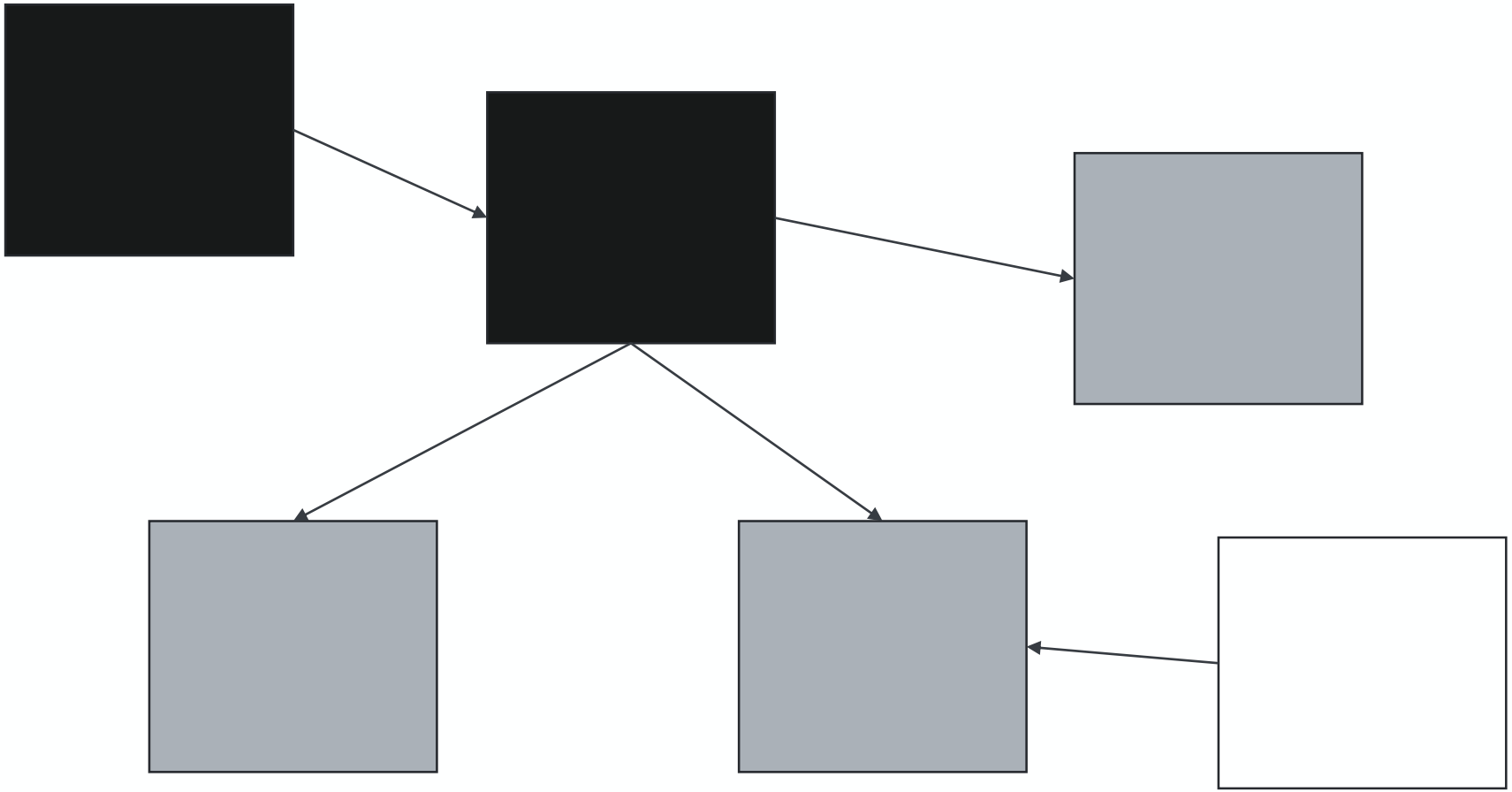
Concurrent Marking: Tri-Color Abstraction



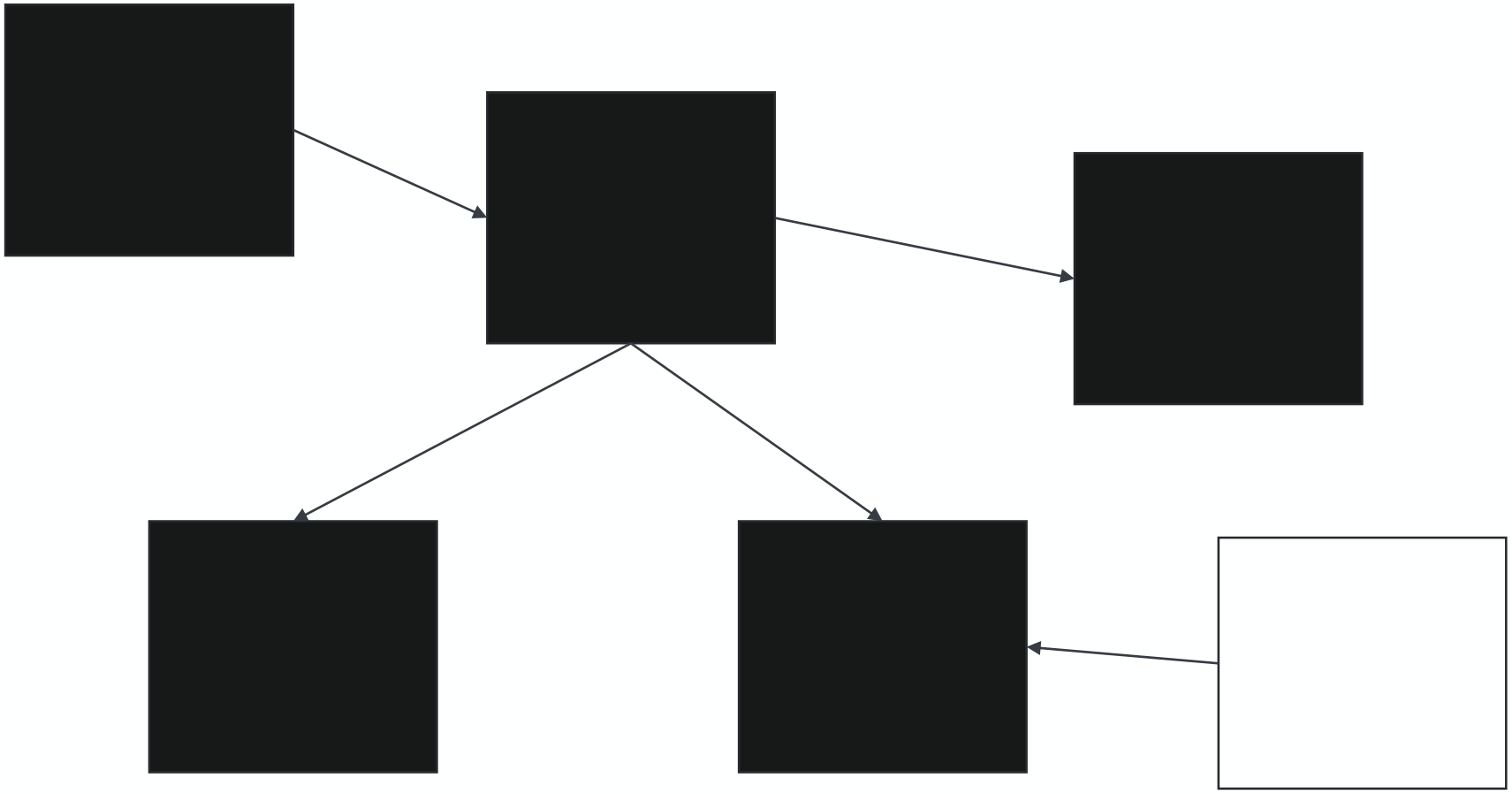
Concurrent Marking: Tri-Color Abstraction



Concurrent Marking: Tri-Color Abstraction



Concurrent Marking: Tri-Color Abstraction



Concurrent Marking: Issues

- New allocations during marking phase can be handled by:
  - Marking automatically object at allocation
  - Not considering new allocations for the current cycle

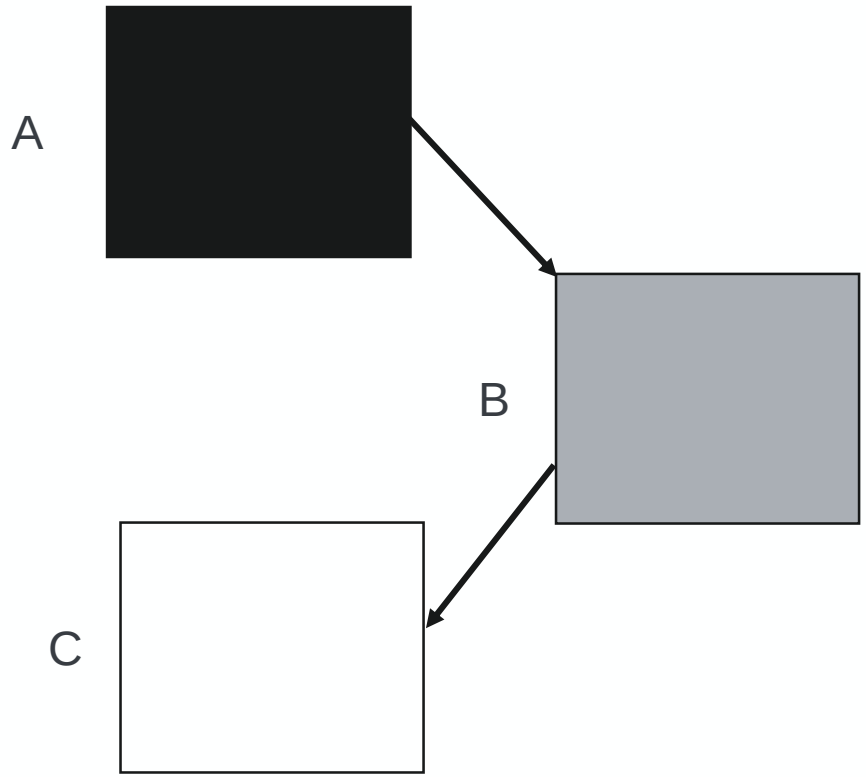
- Tri-Color abstraction provides 2 properties of missed object:

  1. The mutator stores a reference to a white object into a black object.
  2. All paths from any gray objects to that white object are destroyed.

http://www.memorymanagement.org/glossary/s.html#term-snapshot-at-the-beginning

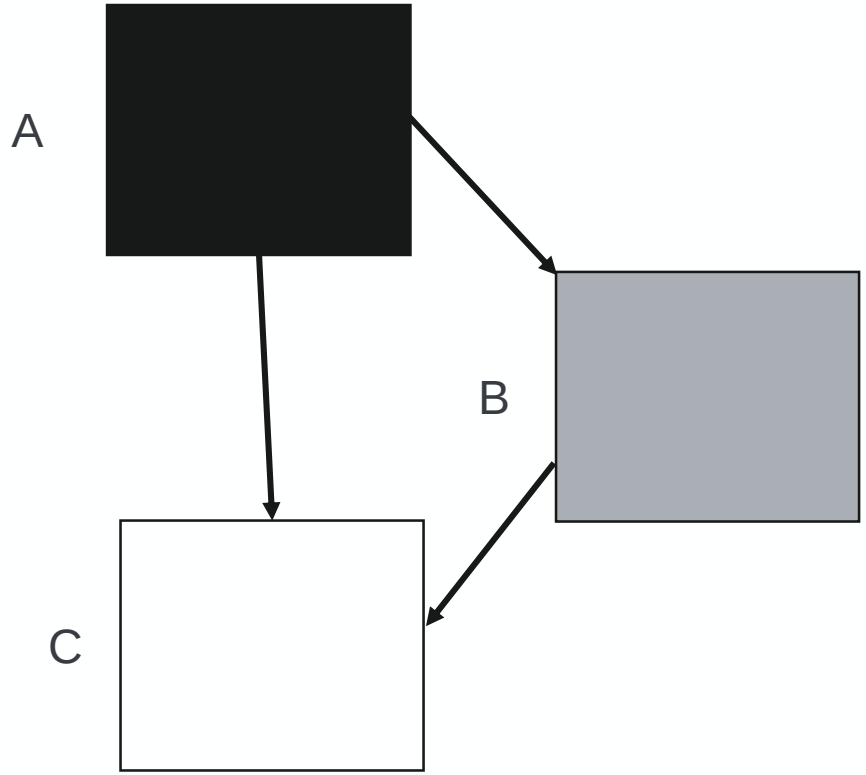
criteo

Concurrent Marking: Issues




```
A.field1 = C;
B.field2 = null;
```

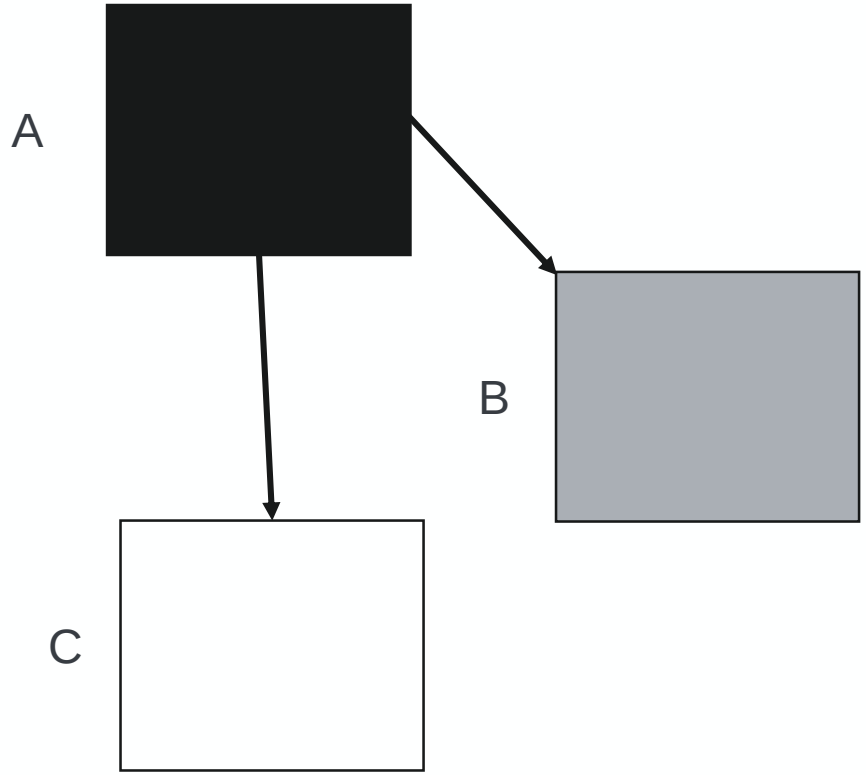
Concurrent Marking: Issues




```
A.field1 = C;
B.field2 = null;
```

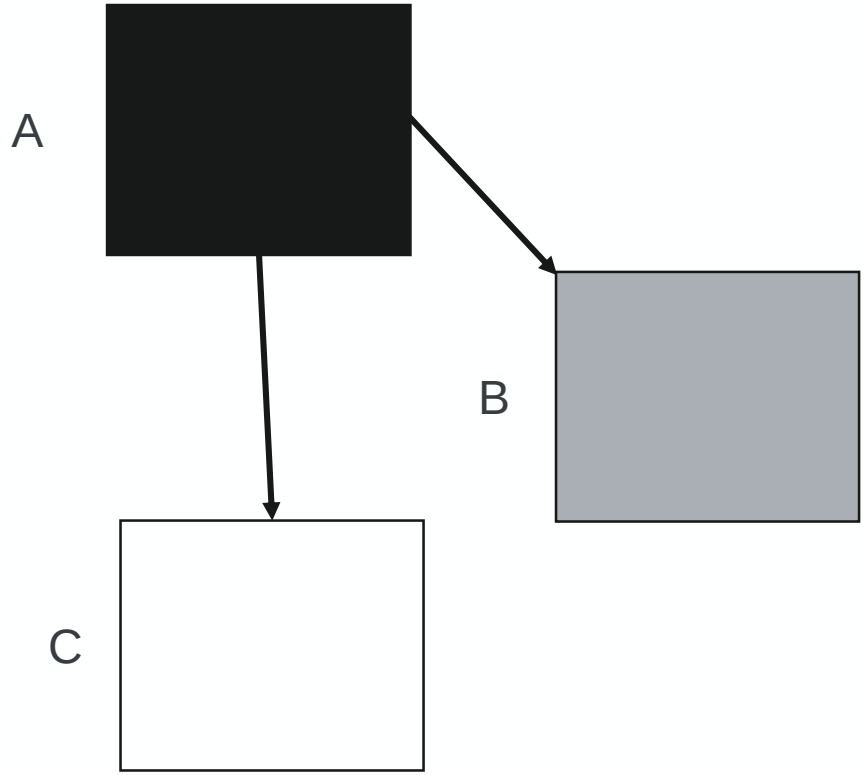

criteo

Concurrent Marking: Issues




```
A.field1 = C;
B.field2 = null;
```

Concurrent Marking: Issues




```
A.field1 = C;
B.field2 = null;
```

OOPS!

criteo

Concurrent Marking: Resolving misses

- 2 ways to ensure not missing any marking
  - Snapshot-At-The-Beginning
  - Incremental Update

- For SATB, Pre-Write Barriers, recording object for marking
  - Before a reference assignation (`X.f = Y`)

- SATB barrier is only active when Marking is on (global state)

```
if (SATB_WriteBarrier) {
  if (X.f != null)
     SATB_enqueue(X.f);
}
```

```
cmp     BYTE PTR [r15+0x30],0x0
jne     0x000002965edc62e5
[...]
mov     r11d,DWORD PTR [rbp+0x74]
test    r11d,r11d
je      0x000002965edc6253
mov     r10,QWORD PTR [r15+0x38]
mov     rcx,r11
shl     rcx,0x3
test    r10,r10
je      0x000002965edc6318
mov     r11,QWORD PTR [r15+0x48]
mov     QWORD PTR [r11+r10*1-0x8],rcx
add     r10,0xfffffffffffffff8
mov     QWORD PTR [r15+0x38],r10
jmp     0x000002965edc6253
mov     rdx,r15
movabs r10,0x7ffac2febc50
call    r10
jmp     0x000002965edc6253
```


criteo

Shenandoah

criteo

Shenandoah GC

- Non-generational (still option for partial collection)
- Region based
- Use Read Barrier: Brooks pointer
- Self-Healing
  - Cooperation between mutator threads & GC threads
  - Only for concurrent compaction
- Mostly based on G1 but with concurrent compaction

Shenandoah Phases

- Initial Marking (STW)
- Concurrent Marking
- Final Remark (STW)
- Concurrent Cleanup
- Concurrent Evacuation
- Init Update References (STW)
- Concurrent Update References
- Final Update References (STW)
- Concurrent Cleanup

Concurrent Marking

- SATB-style (like G1)
- 2 STW pauses for Initial Mark & Final Remark
- Conditional Write Barrier
  - To deal with concurrent modification of object graph

Concurrent Evacuation

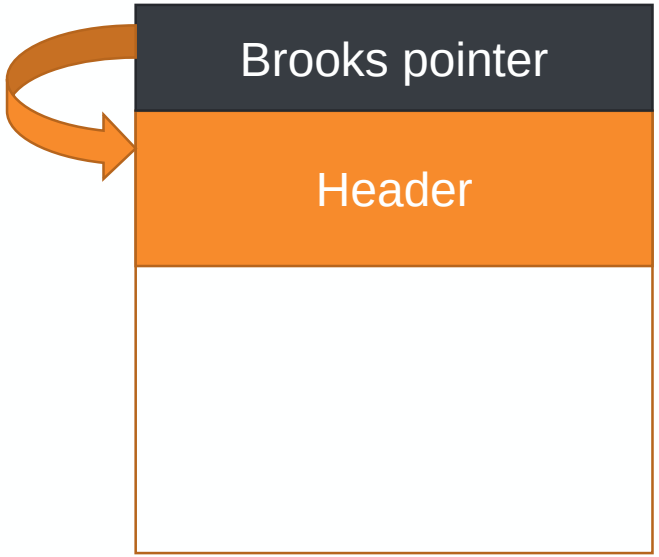
- Same principle than G1:
  - Build CollectionSet with Garbage First!
  - Evacuate to new regions to release the region for reuse

- Concurrent Evacuation done with the help of:
  - 1 Read Barrier : Brooks pointer
  - 4 Write Barriers

- Barriers help to keep the to-space invariant:
  - All Writes are made into an object in to-space

Brooks pointers

- All objects have an additional forwarding pointer
  - Placed before the regular object



- Dereference the forwarding pointer for each access
  - Memory footprint overhead
  - Throughput overhead

```
mov     r13,QWORD PTR [r12+r14*8-0x8]
```


criteo

Write Barriers

- Any writes (even primitives) to from-space object needs to be protected

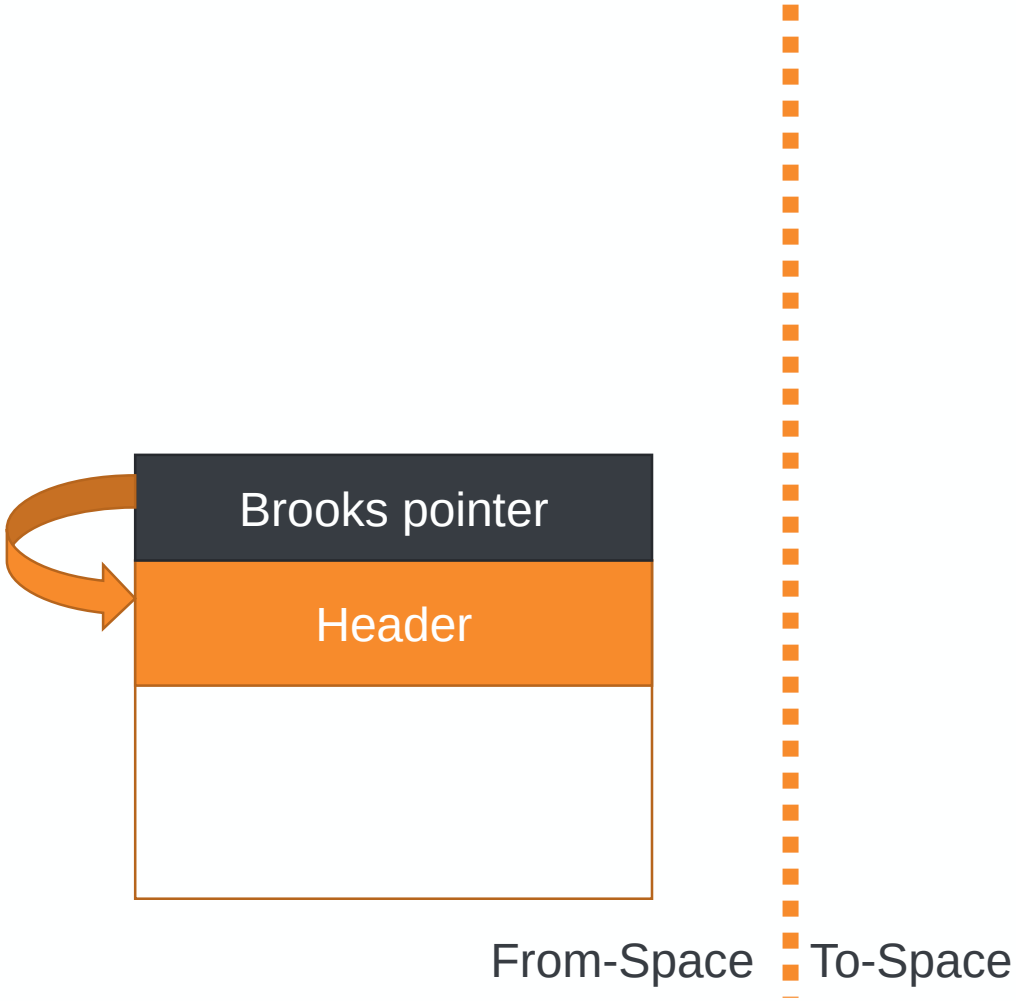
```
if (evacInProgress
  && inCollectionSet(obj)
  && notCopyYet(obj)) {
    evacuateObject(obj)
}
```

```
test   BYTE PTR [r15+0x3c0],0x2
jne    0x00000000281bcbc
[...]
mov    r10d,DWORD PTR [r13+0xc]
test   r10d,r10d
je     0x00000000281bc2b
mov    r11,QWORD PTR [r15+0x360]
mov    rcx,r10
shl    rcx,0x3
test   r11,r11
je     0x00000000281bd0d
[...]
mov    rdx,r15
movabs r10,0x62d1f660
call   r10
jmp    0x00000000281bc2b
```

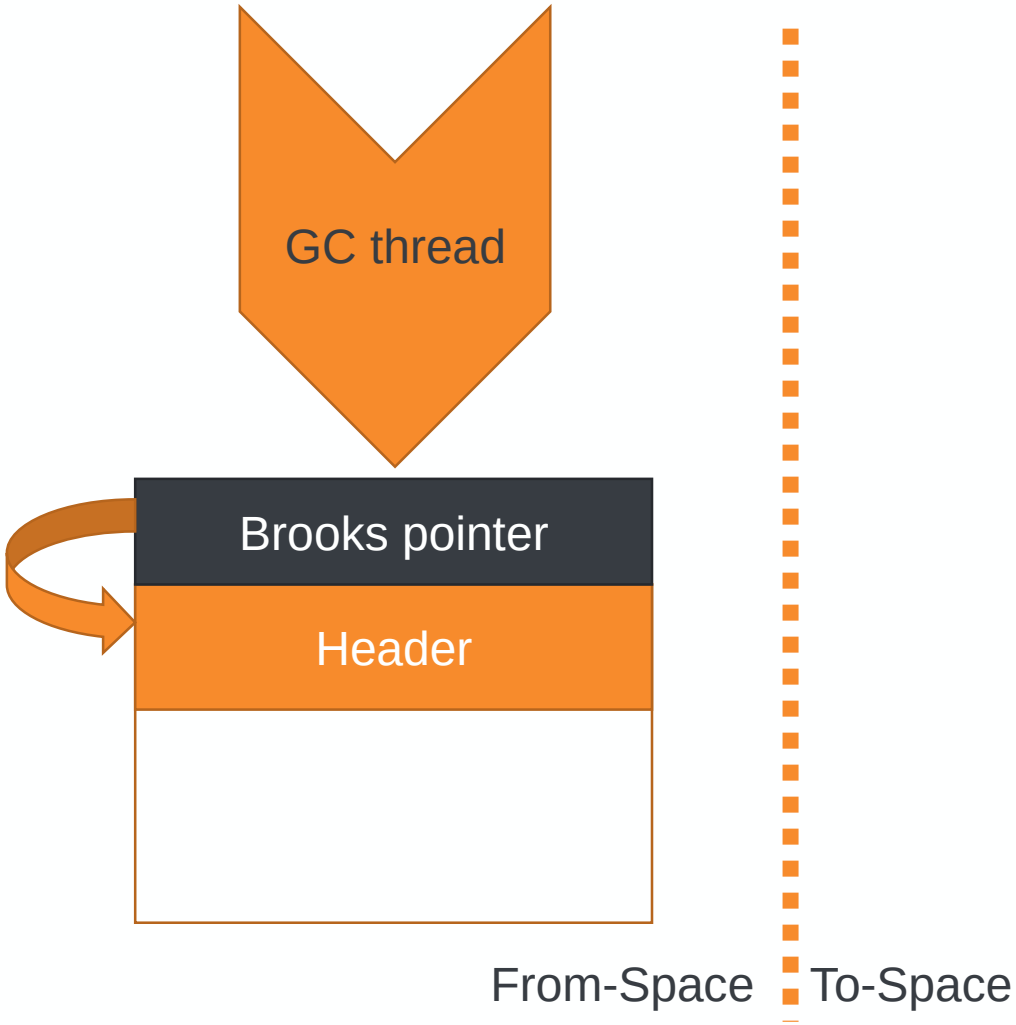
- Exotic barriers:
  - acmp (pointer comparison)
  - CAS
  - clone

criteo

Concurrent Copy: GC thread

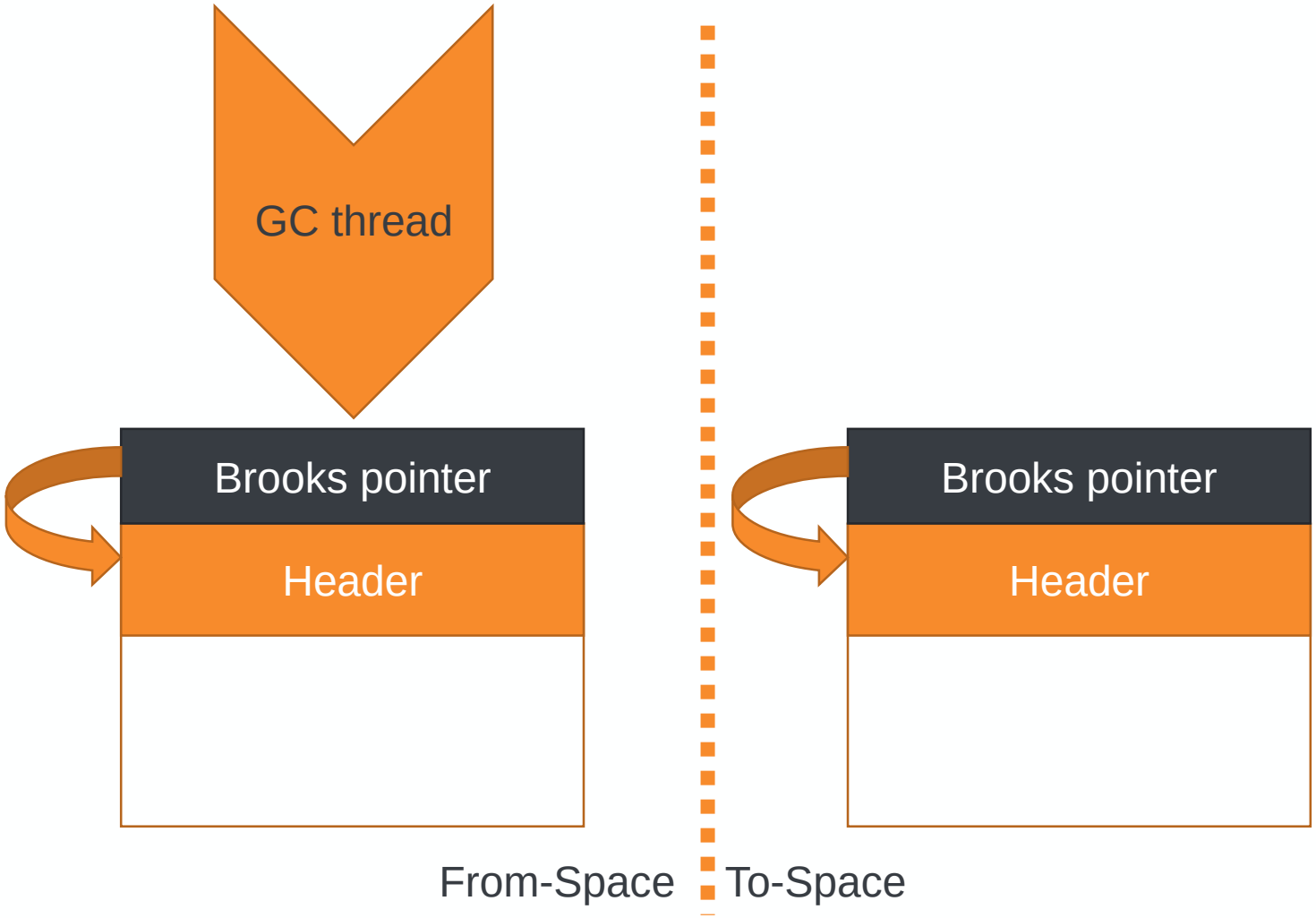


Concurrent Copy: GC thread



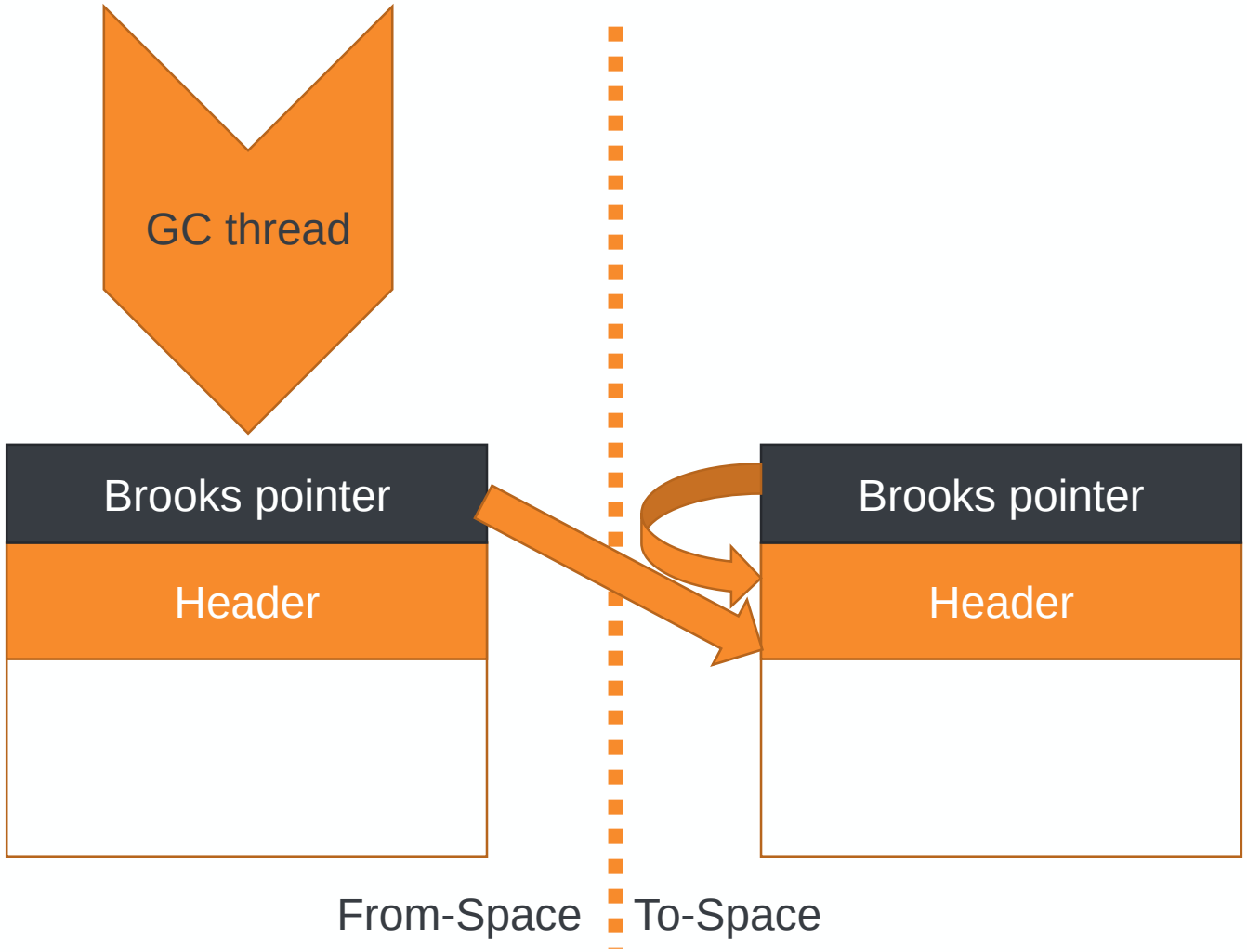
criteo

Concurrent Copy: GC thread



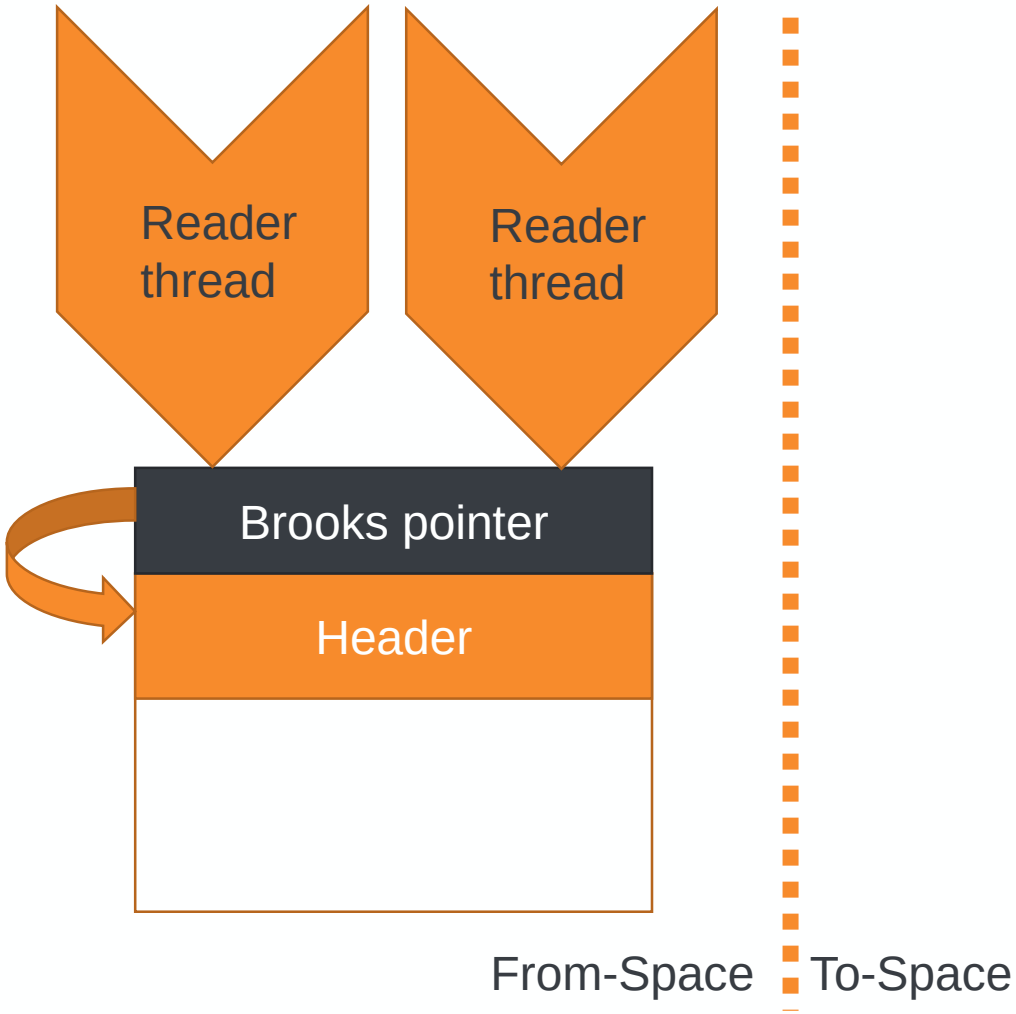
criteo

Concurrent Copy: GC thread



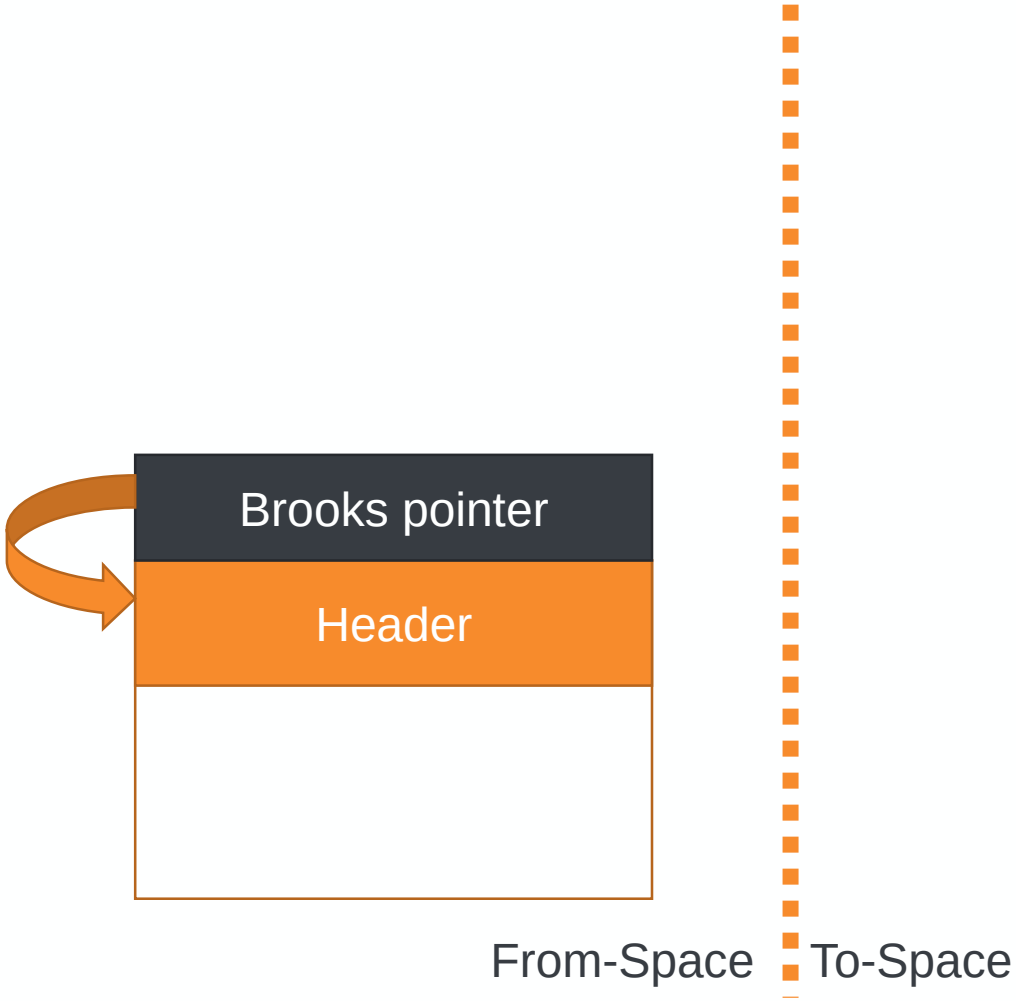
criteo

Concurrent Copy: Reader threads

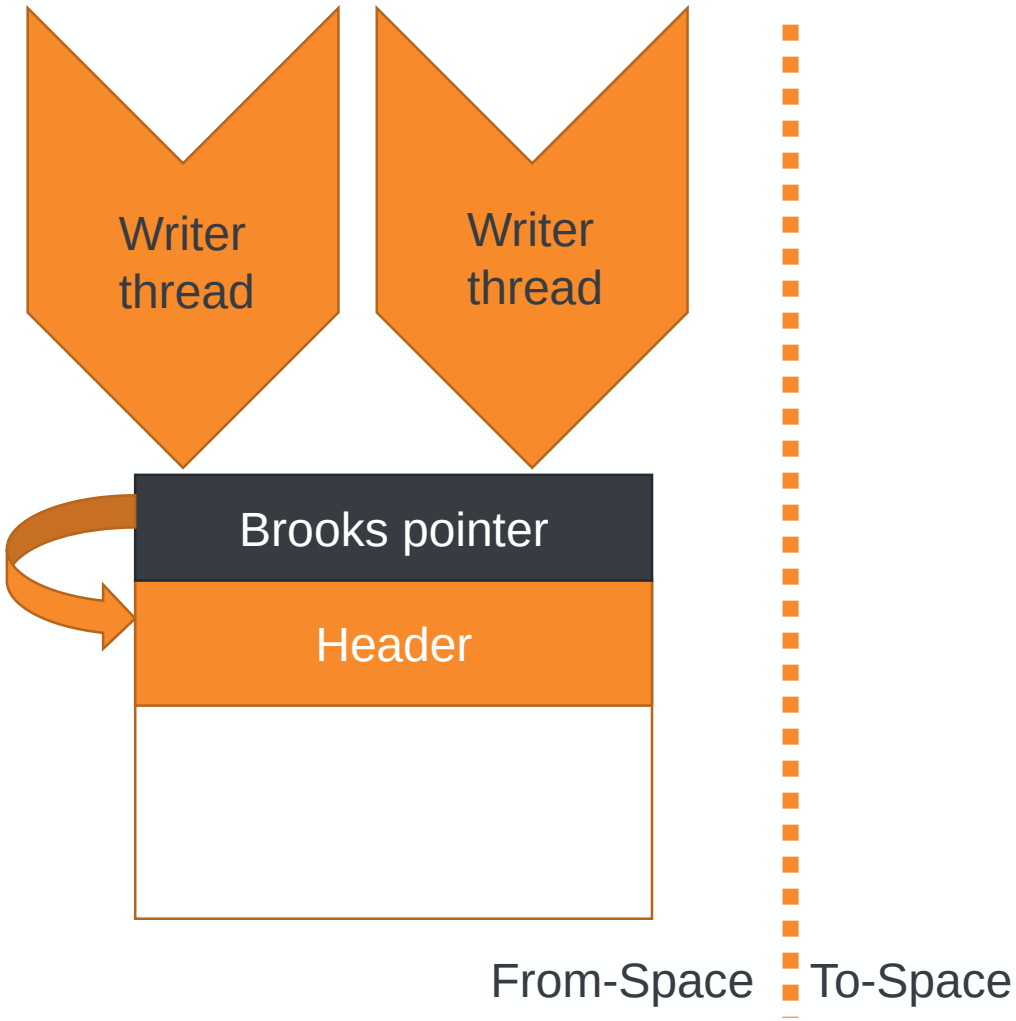


criteo

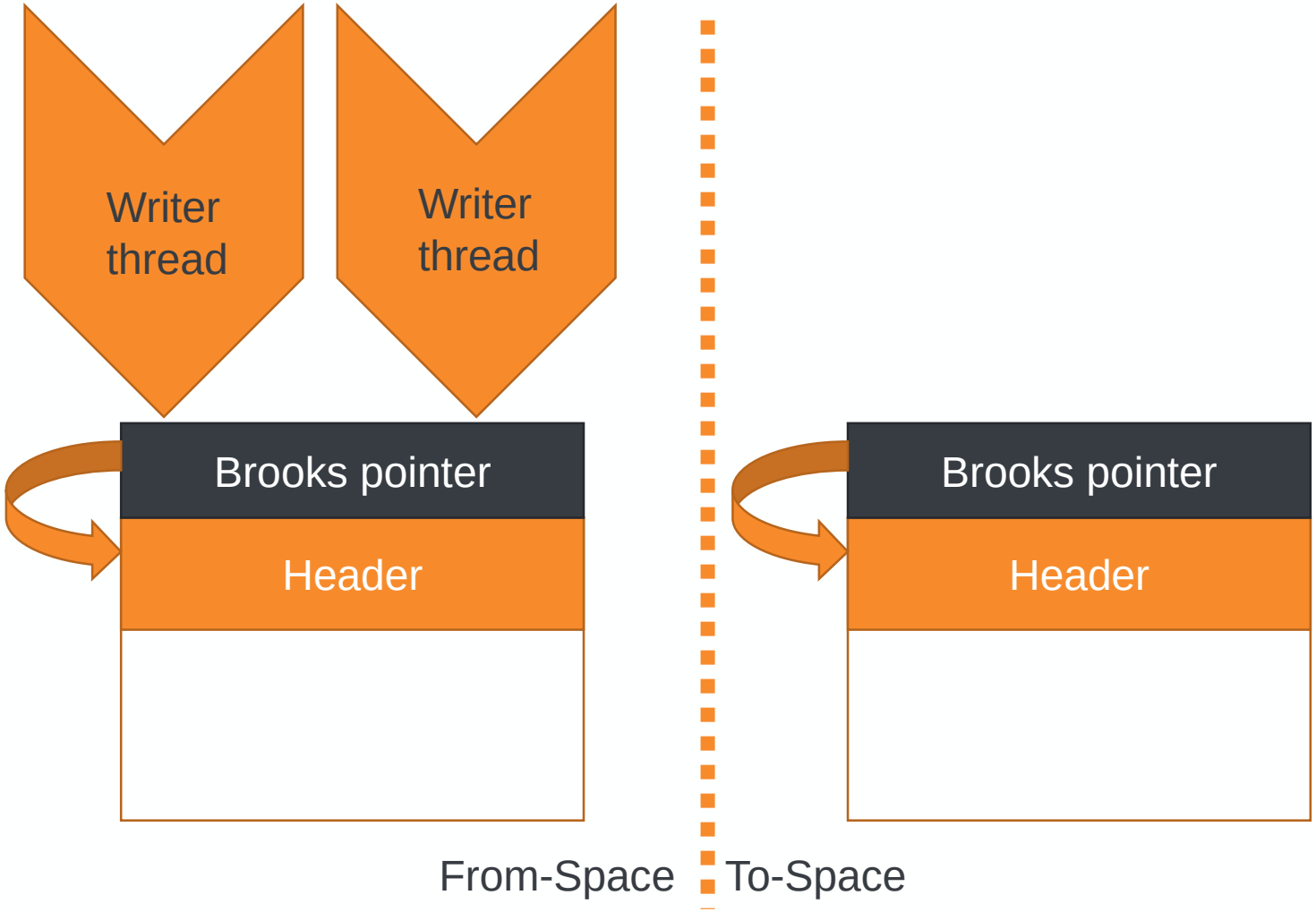
Concurrent Copy: Writer threads



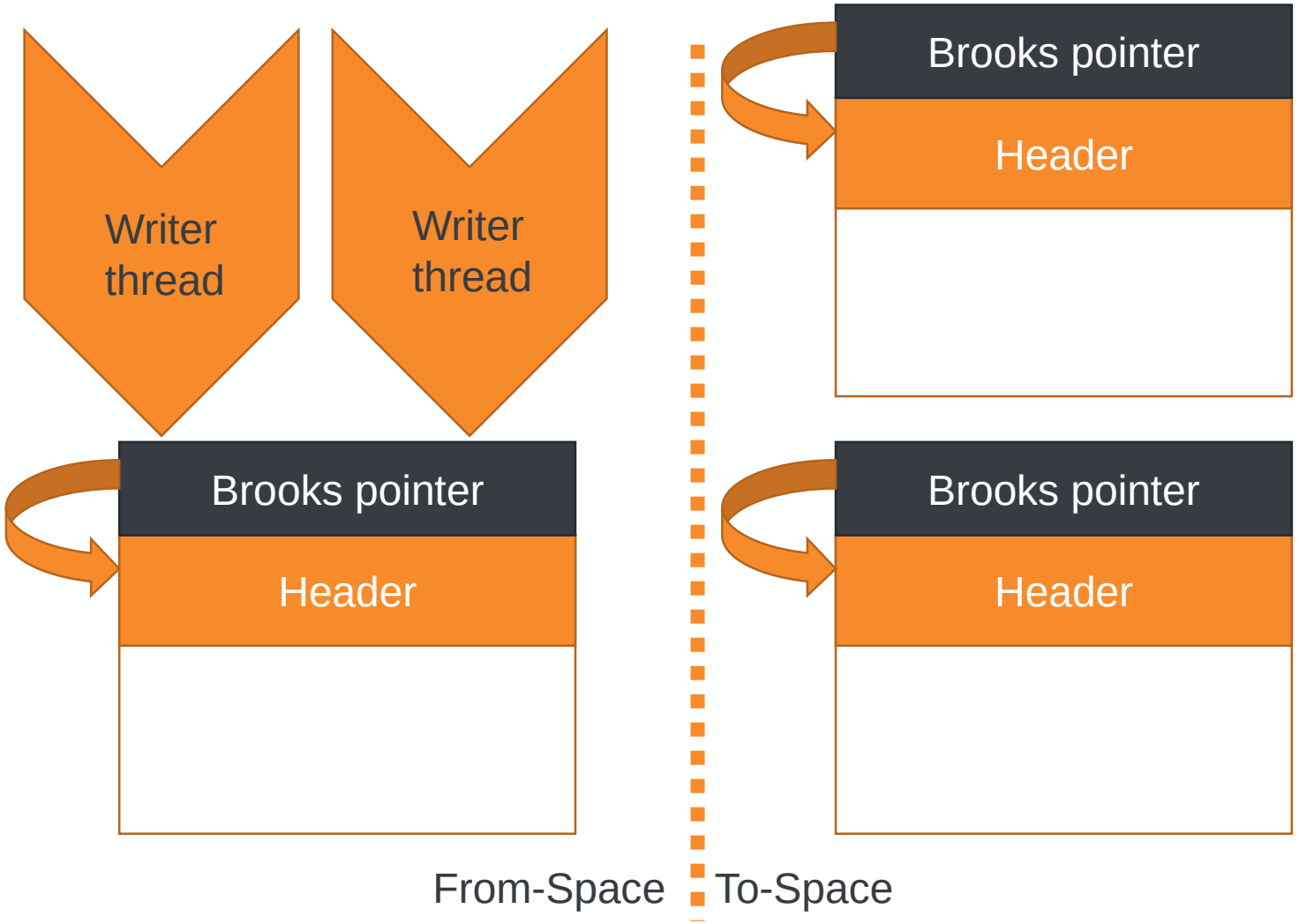
Concurrent Copy: Writer threads



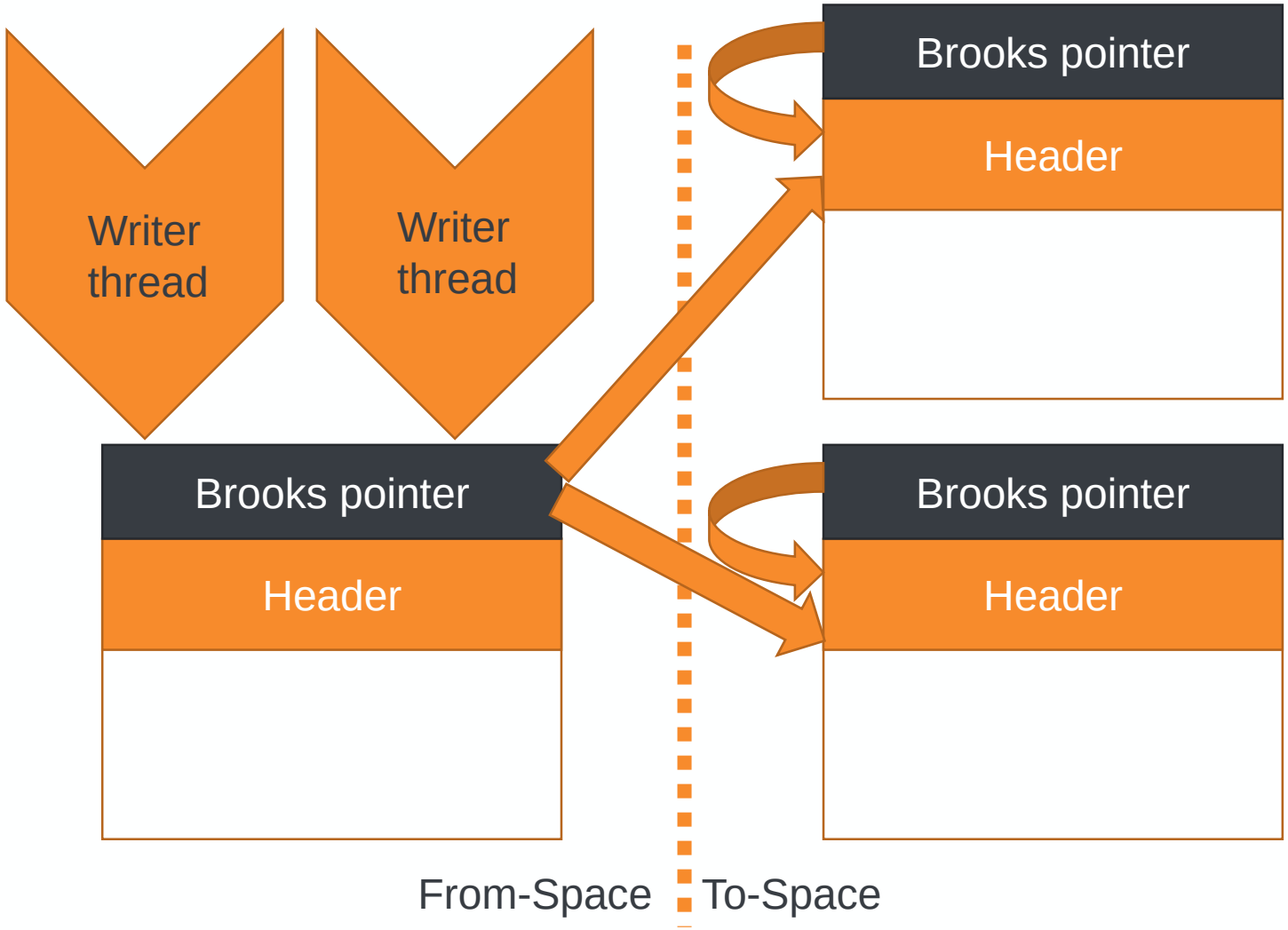
Concurrent Copy: Writer threads



Concurrent Copy: Writer threads

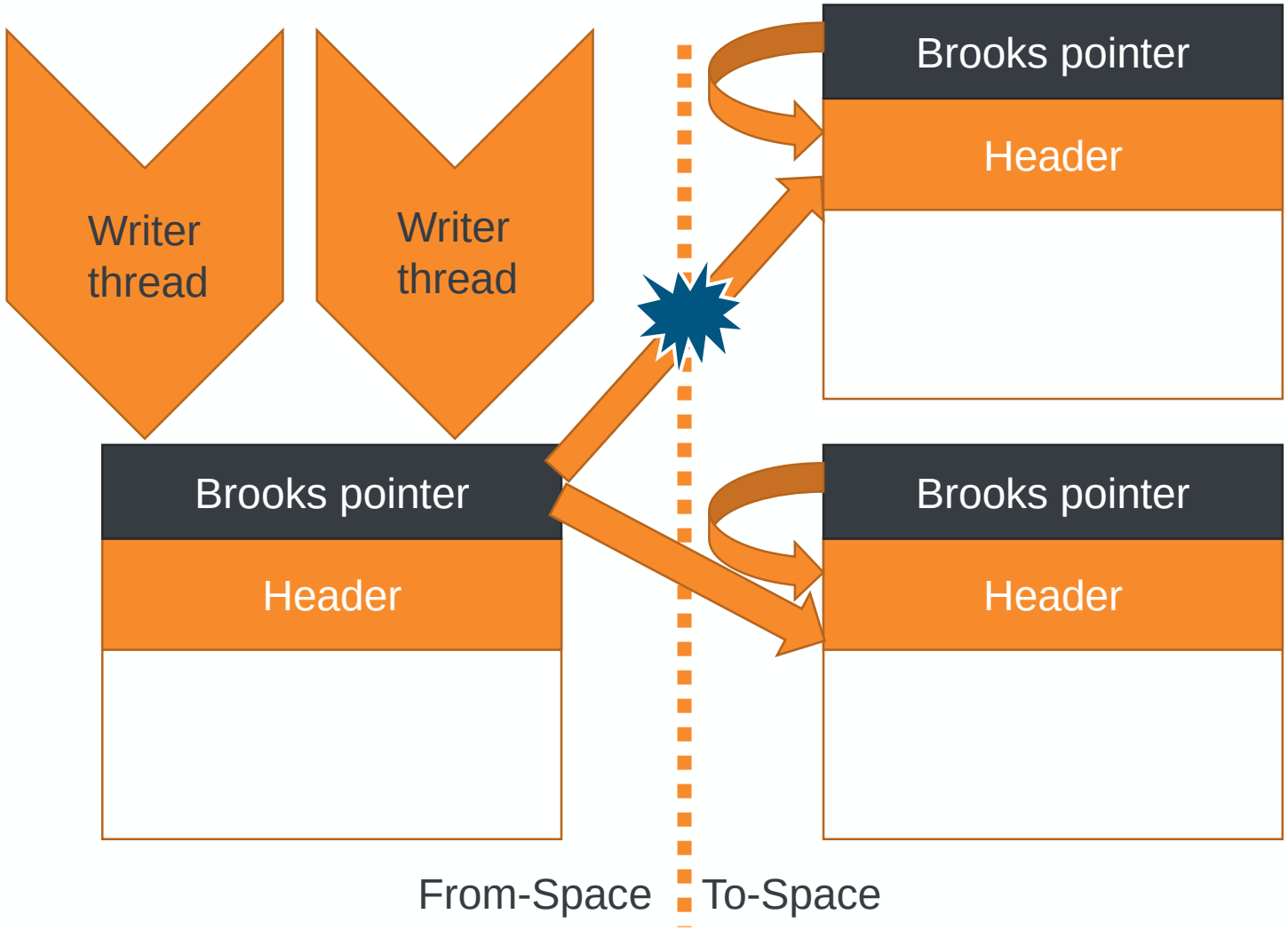


Concurrent Copy: Writer threads

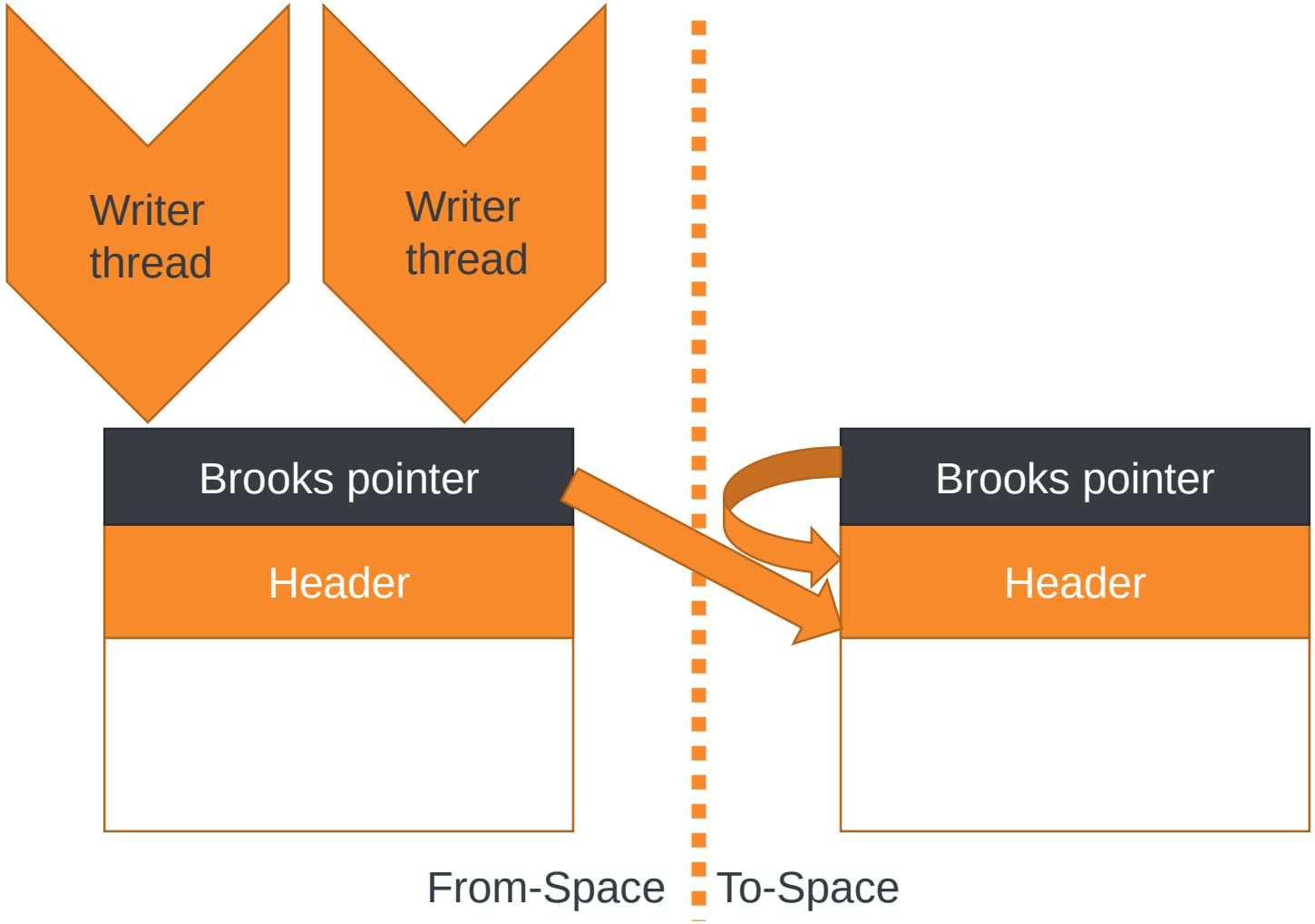


criteo

Concurrent Copy: Writer threads



Concurrent Copy: Writer threads



criteo

Extreme cases

- Late memory release
  - Only happens when all refs updated (Concurrent Cleanup phase)
- Allocations can overrun the GC
- Failure modes:
  - Pacing
  - Degenerated GC
  - FullGC

criteo

Shenandoah 2.0?

- Since JDK13, notable changes introduced

- Load Reference Barrier
  - Baker-style barrier (Relocation)
  - Evaluated at reference load time

- Eliminating forward pointer word
  - Store forward information into Mark word
  - Remove Brooks pointer

Azul's C4

criteo

Continuously Concurrent Compacting Collector

- Generational (young & old)
- Region based (pages)
- Use Read Barrier: Loaded Value Barrier
- Self-Healing
  - Cooperation between mutator threads & GC threads
- Pauseless algorithm but implementation requires safepoints
- Pauses are most of the time < 1ms

LVB

- Baker-style based Barrier
  - move objects through forwarding addresses stored aside
  - Applied at load time, not when dereferencing
  - Fused marking & relocation state
- Ensure C4 invariants:
  - Marked Through the current cycle
  - Not relocated
- If not => Self-healing process to correct it
  - Mark object
  - Relocate & correct reference
- Checked for each reference loads
  - Benefits from JIT optimization for caching loaded value (registers)

criteo

LVB

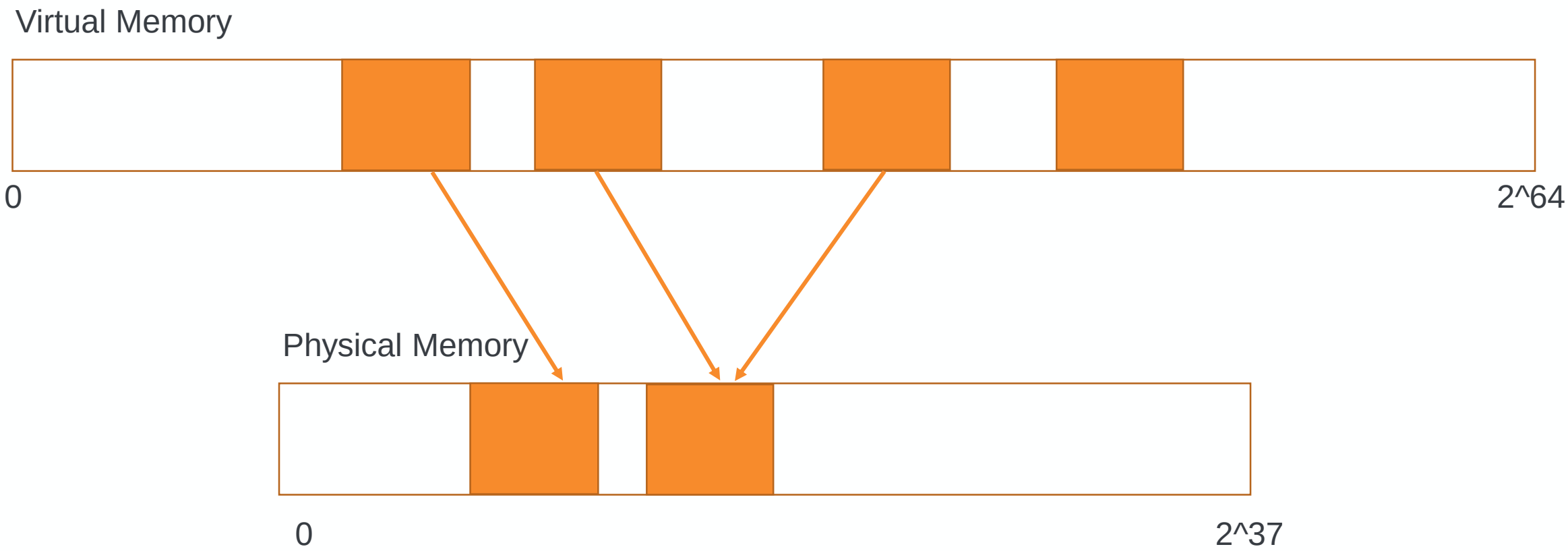
- States of objects stored inside reference address => Colored pointers
  - NMT bit
  - Remapped/Generation

- Checked against a global expected value during the GC cycle
  - Thread local, almost always L1 cache hits
  - Register

```
test    r9, rax
jne     0x3001443b
mov     r10d, dword ptr [rax + 8]
```

- Unmasking reference addresses:
  - Linux Kernel module, aliasing
  - memfd, multi-mapping

Virtual Memory vs Physical Memory



criteo

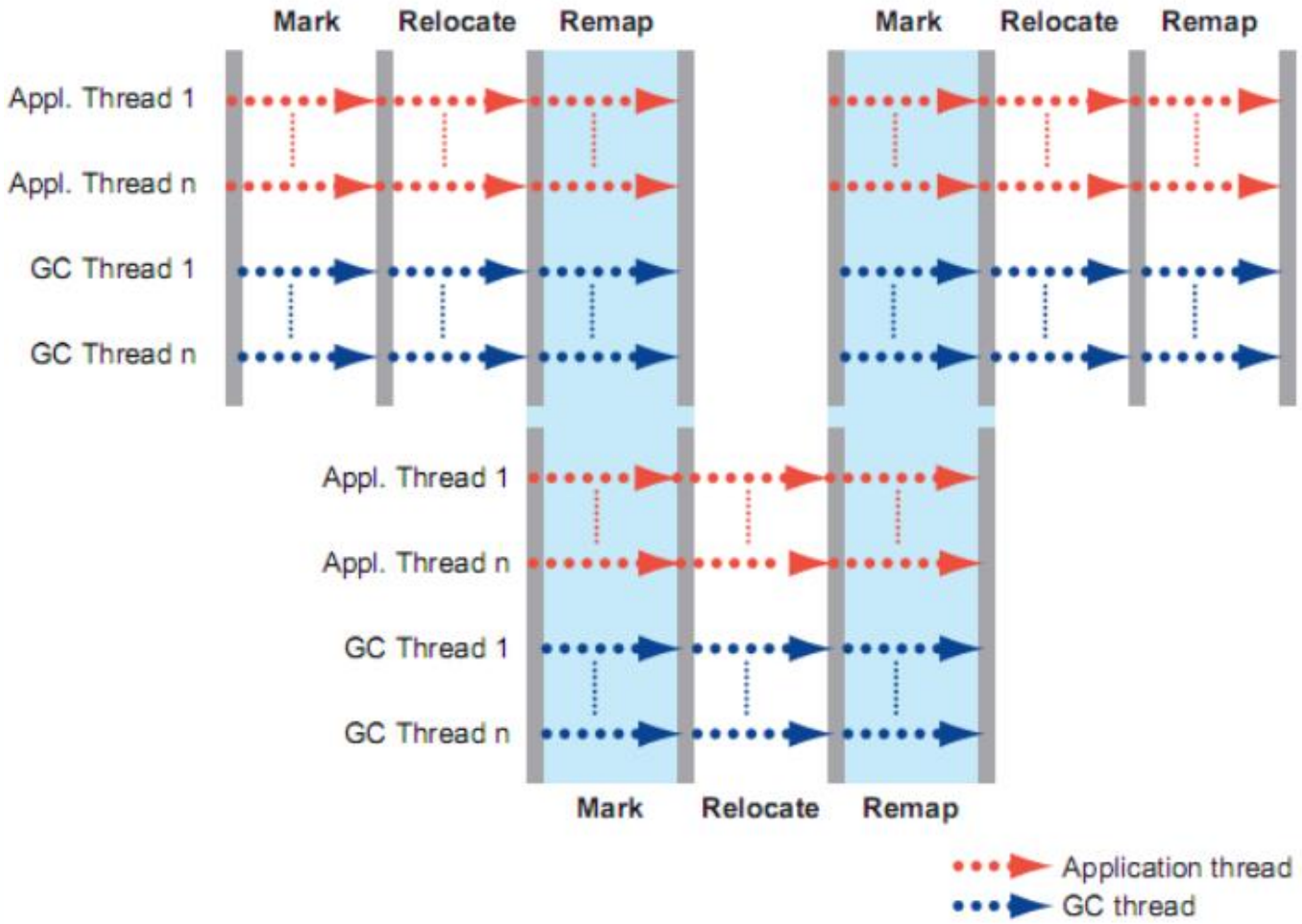
C4 Phases

- All phases are fully parallel & concurrent

- No "rush" to finish phases
  - No constraint about STW pause to be short
  - Physical memory released quickly in relocation phase
  - Can be reused for new allocations
  - Plenty of virtual space vs physical memory

C4 Phases

- Mark
  - Marking all objects in graph
- Relocation
  - Moving objects to release pages
- Remap
  - Fixup references in object graph
  - Folded with next mark cycle

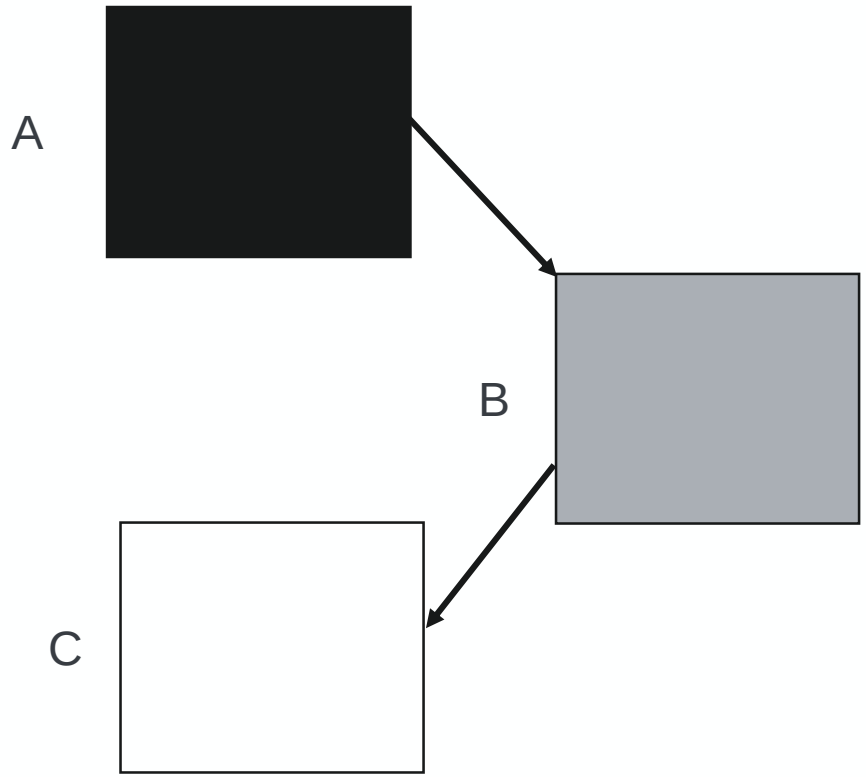


criteo

Mark Phase

- Precise Wavefront Marking
- Single pass
- No final mark/remark
- Self-Healing: Mark object that are not marked for the current cycle

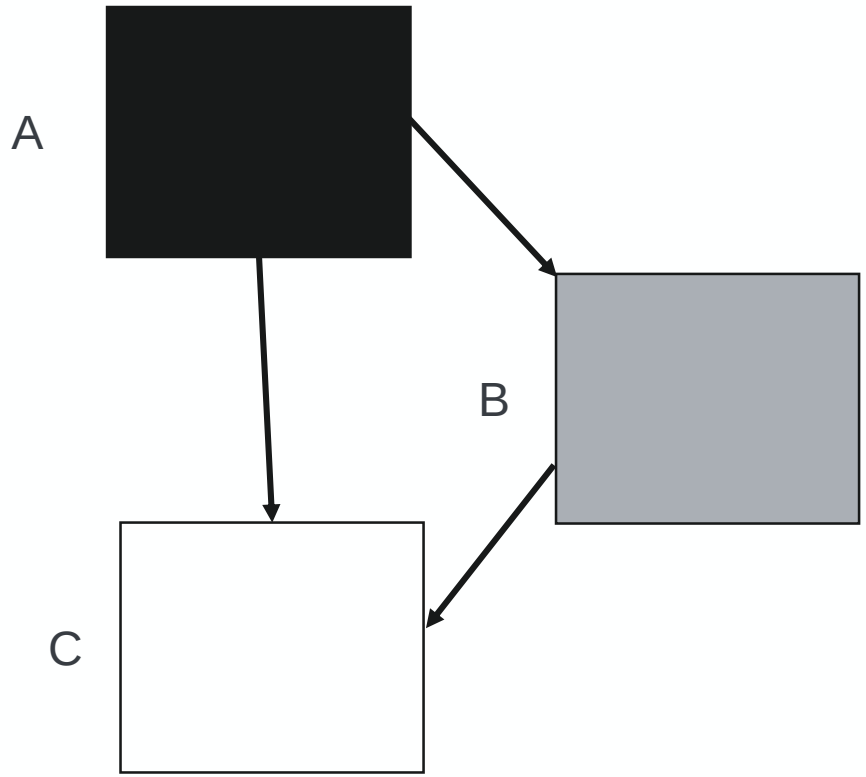
Mark Phase: Concurrent Modification




```
A.field1 = C;
B.field2 = null;
```


criteo

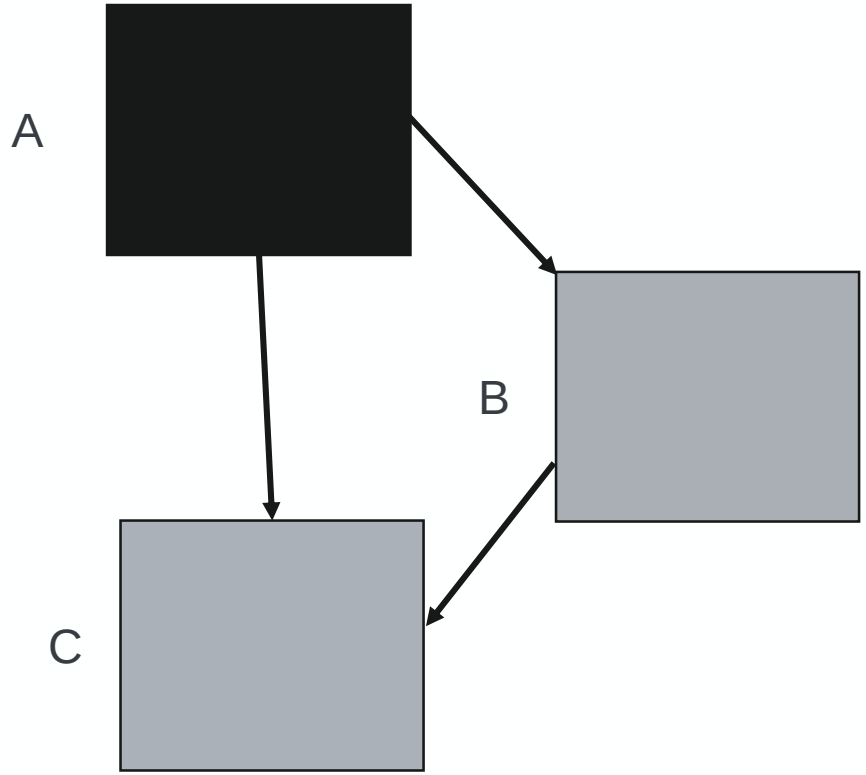
Mark Phase: Concurrent Modification




```
A.field1 = C;
B.field2 = null;
```


criteo

Mark Phase: Concurrent Modification

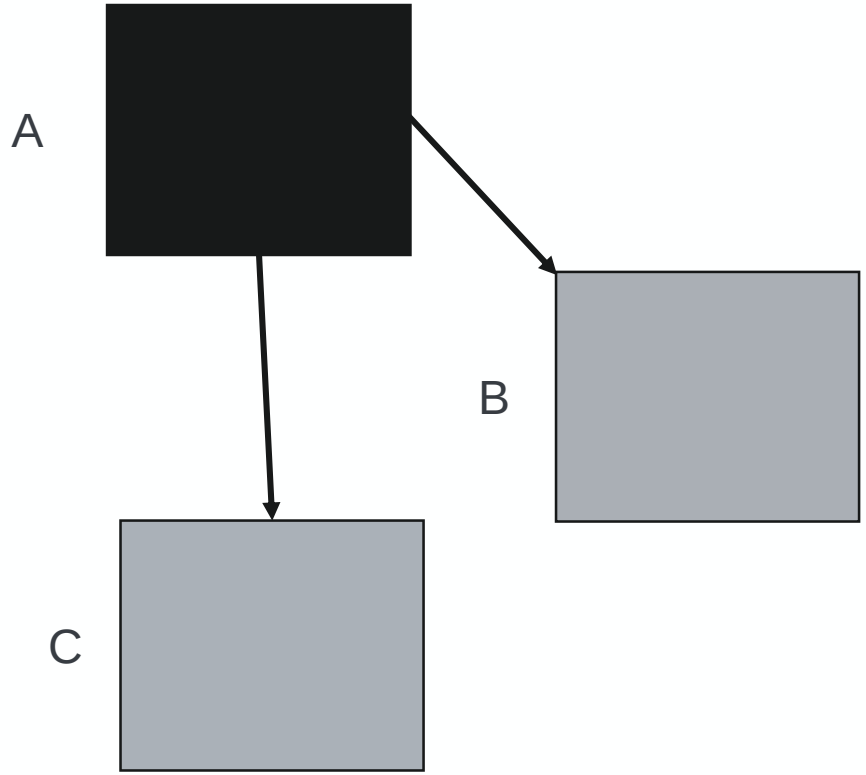


LVB

```
A.field1 = C;
B.field2 = null;
```


criteo

Mark Phase: Concurrent Modification



LVB

```
A.field1 = C;
B.field2 = null;
```


criteo

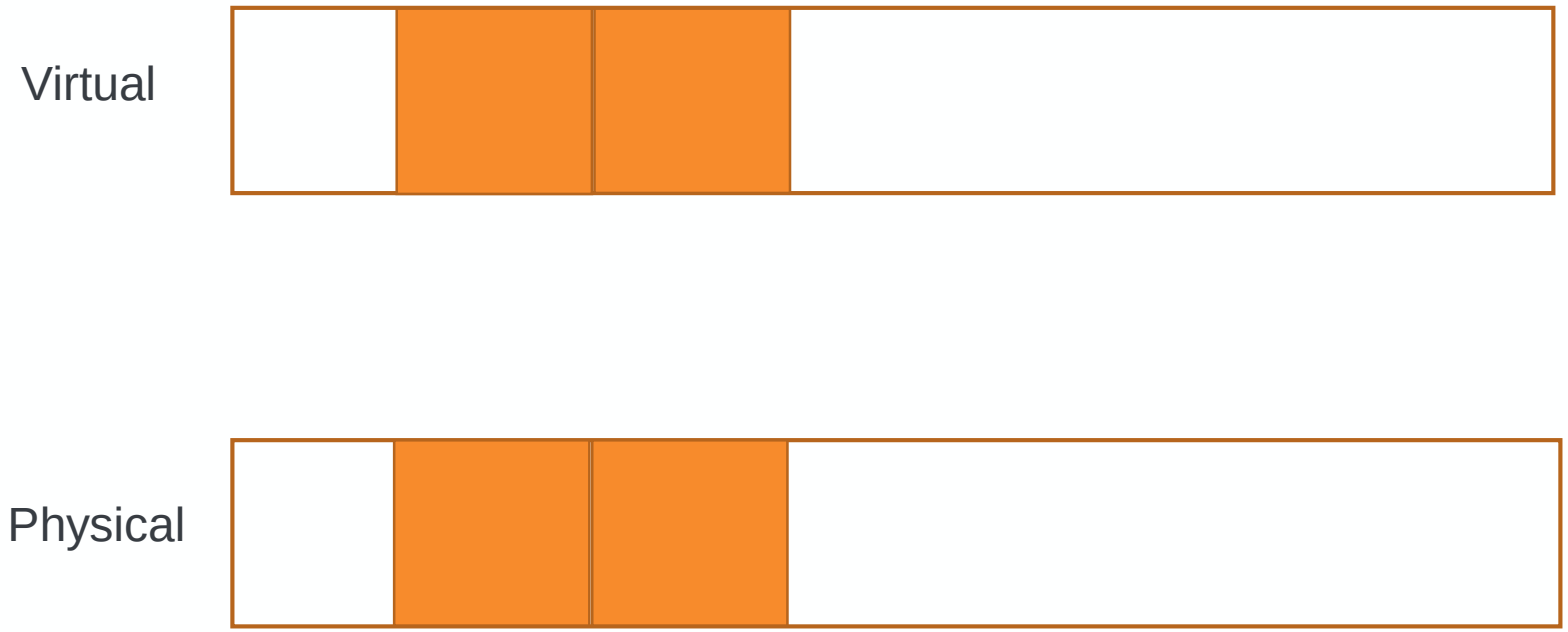
Mark Phase

- Scanning roots (Static var, Thread stacks, register, JNI handles)
  - GC threads scans stalled threads
  - Running threads scans their own stack stopping **individually** at Safepoint
- Scanning object graph like a parallel collector
- Newly allocated objects into new pages, not considered for reclaim (relocation)
- For each page, summing live data bytes, used to select page to reclaim

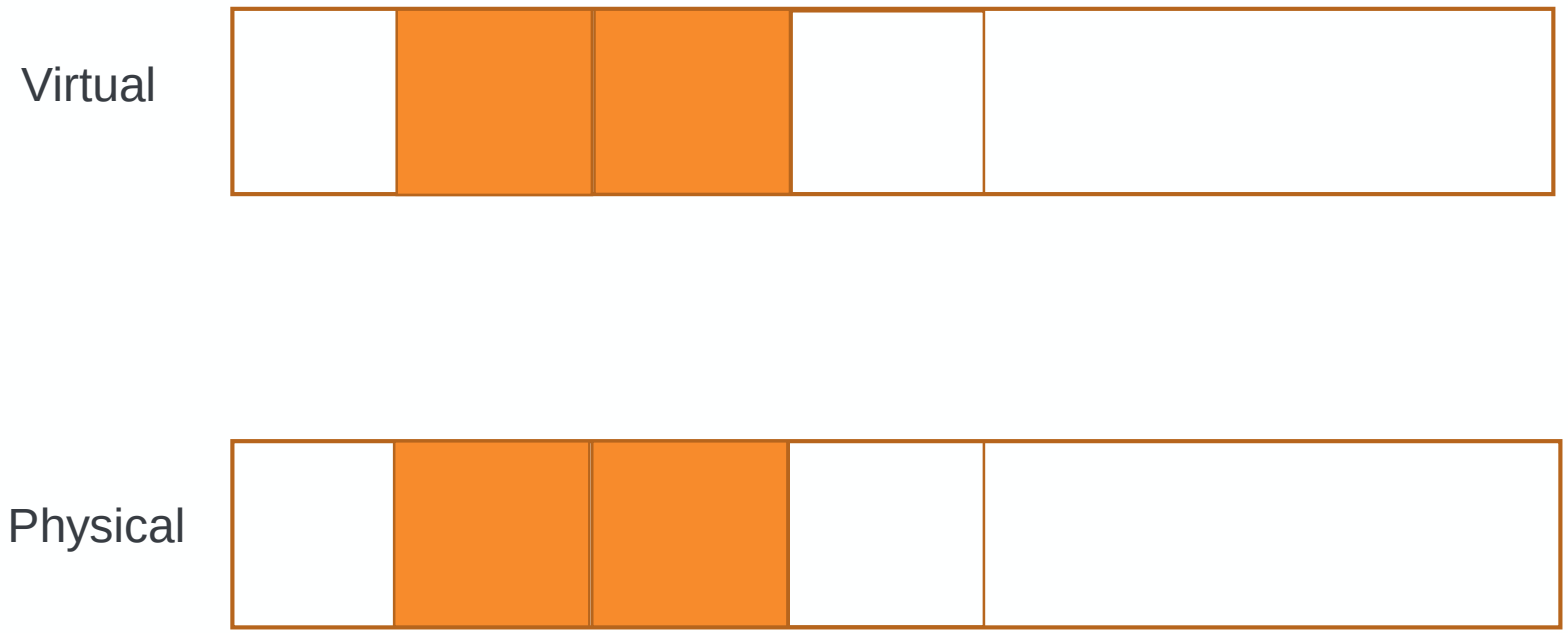
Relocation Phase

- Select pages with the greatest number of dead objects (garbage first!)
- Protect page selected from being accessed by mutators thread
- Move objects to new allocated pages
- Build side arrays (off heap tables) for forwarding information
- Self-Healing: As protected, LVB will trigger a trap to:
  - Copy object to the new location if not done
  - Use forward pointer to fix the reference

Relocation Phase

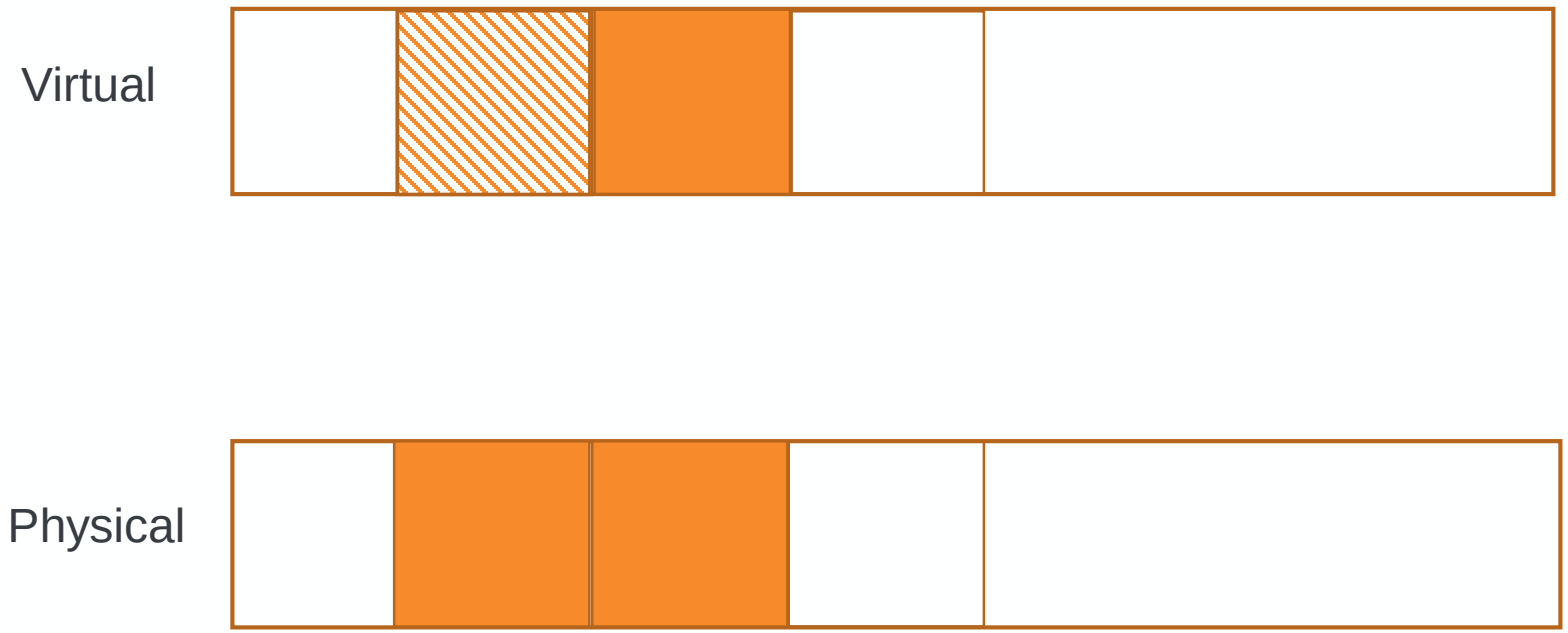


Relocation Phase

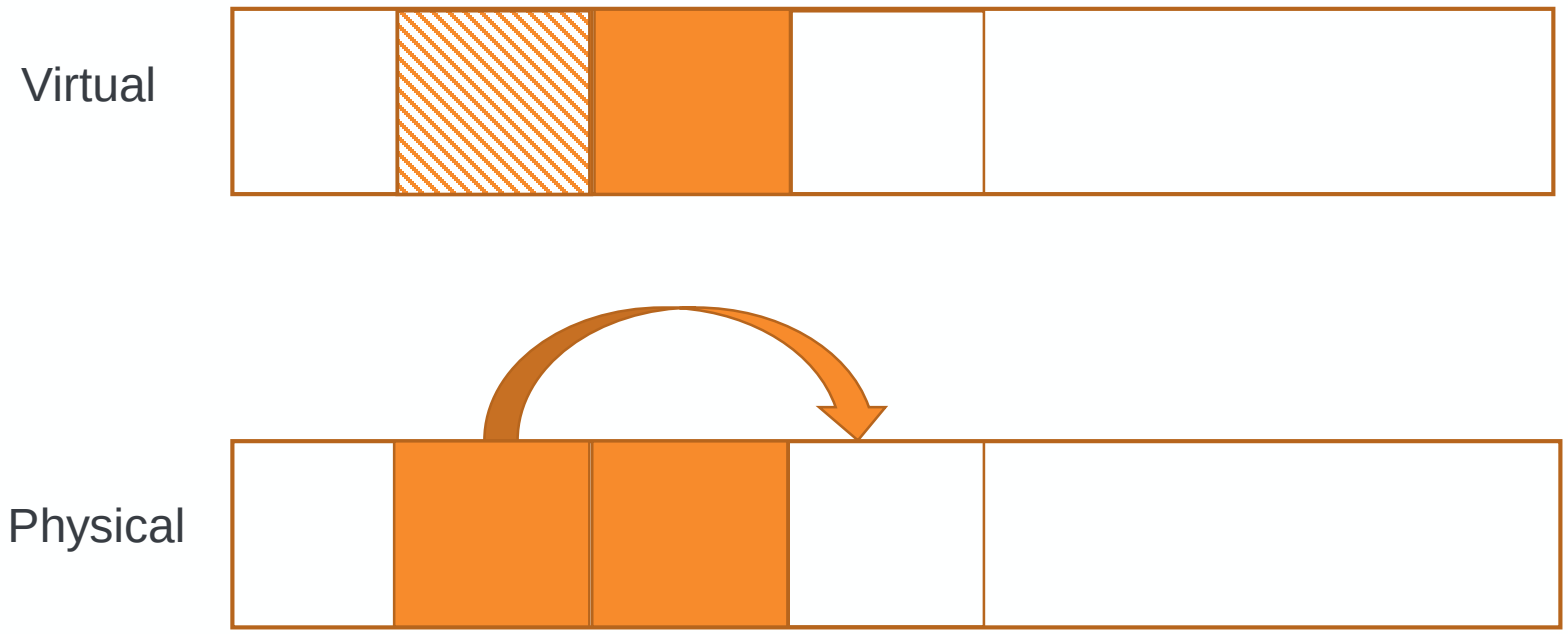


criteo

Relocation Phase

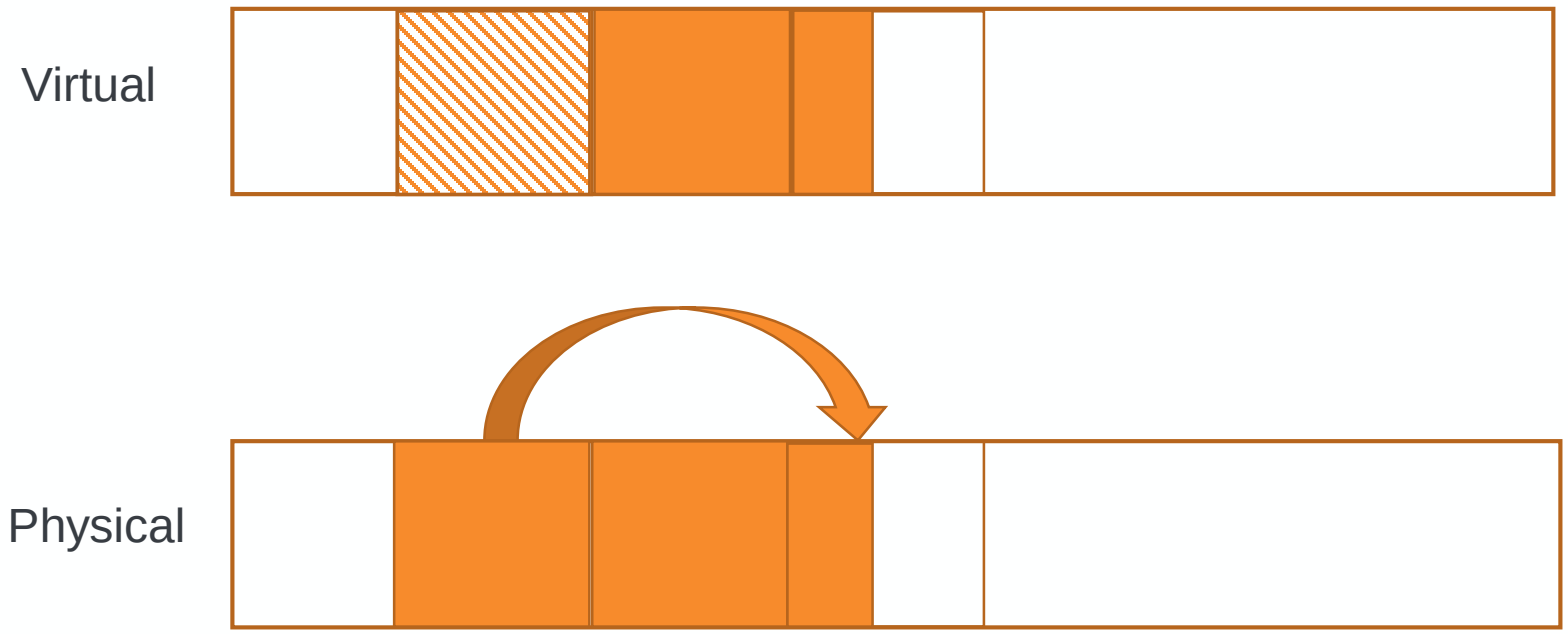


Relocation Phase



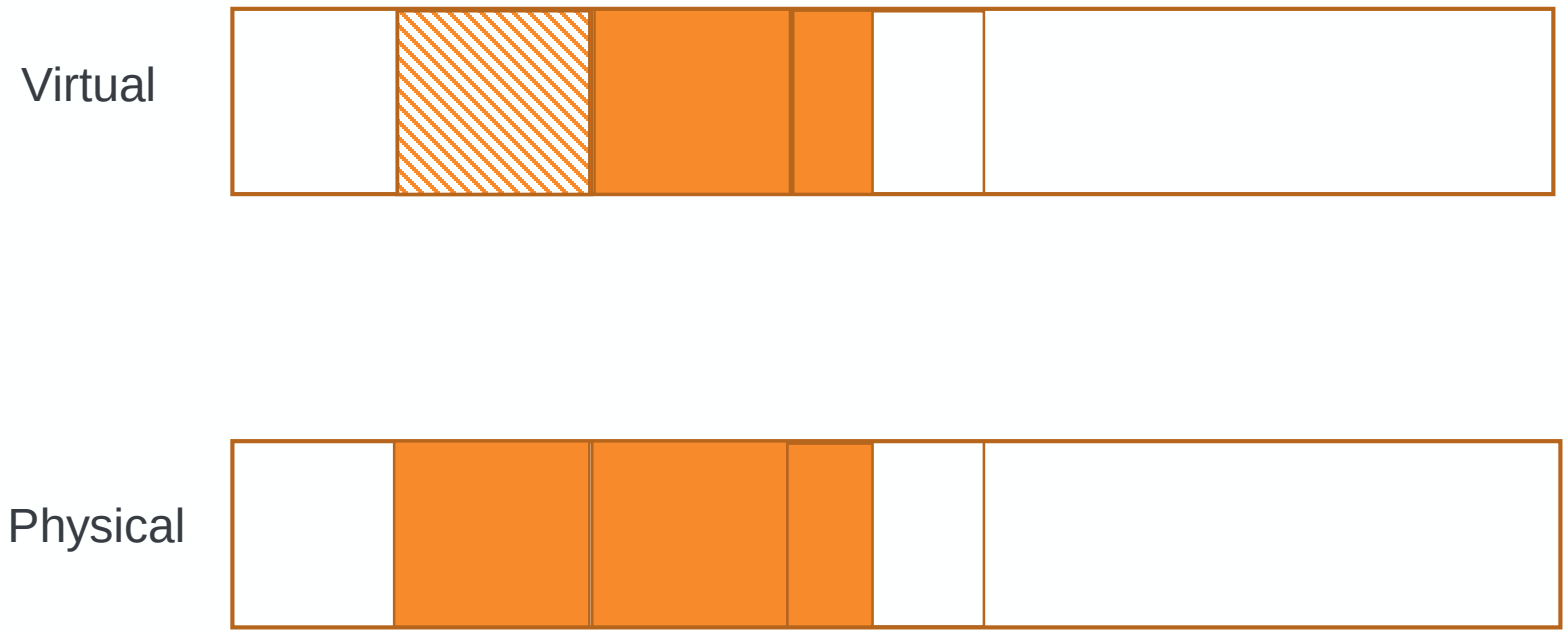
criteo

Relocation Phase



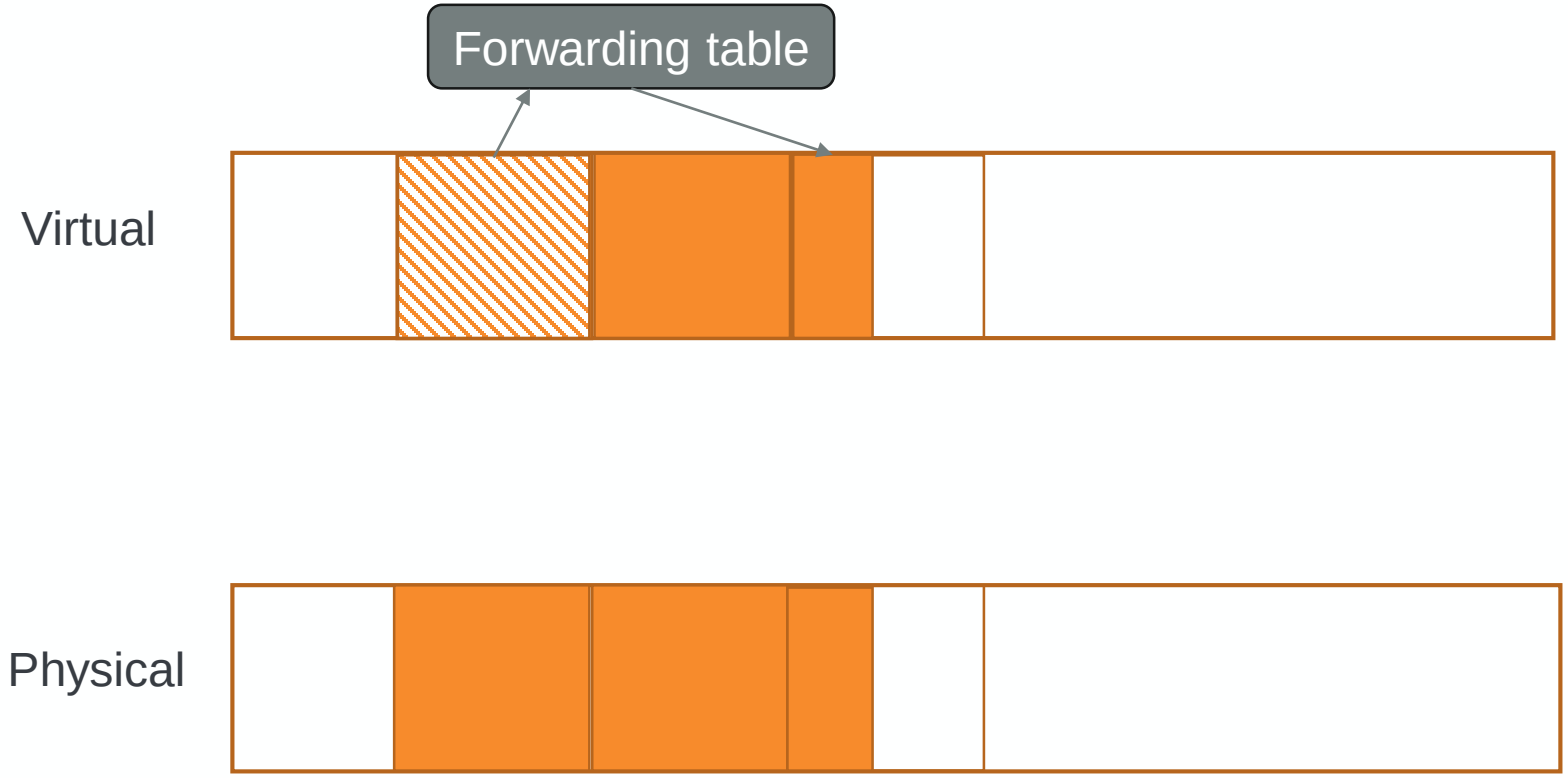
criteo

Relocation Phase



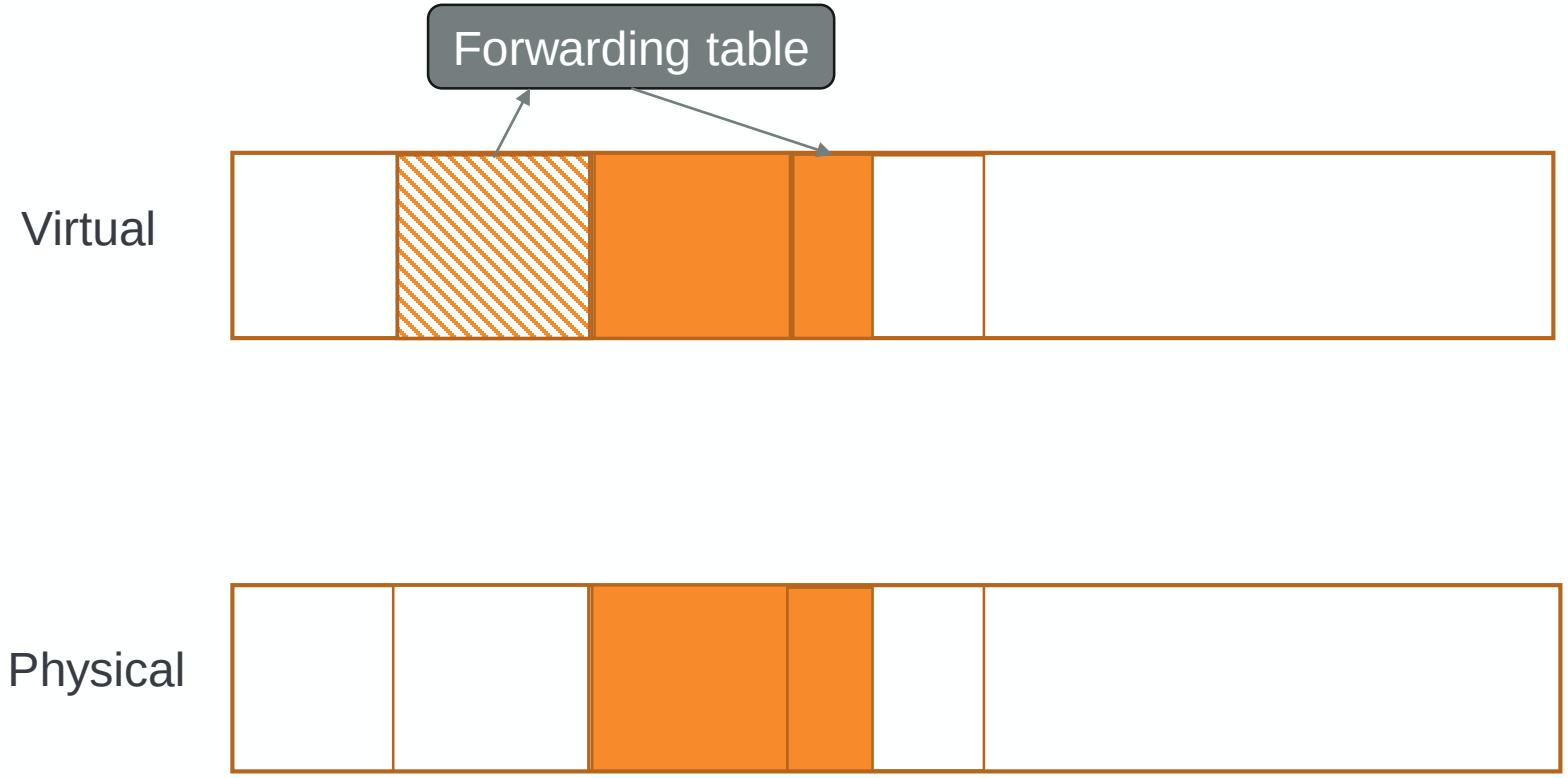
criteo

Relocation Phase



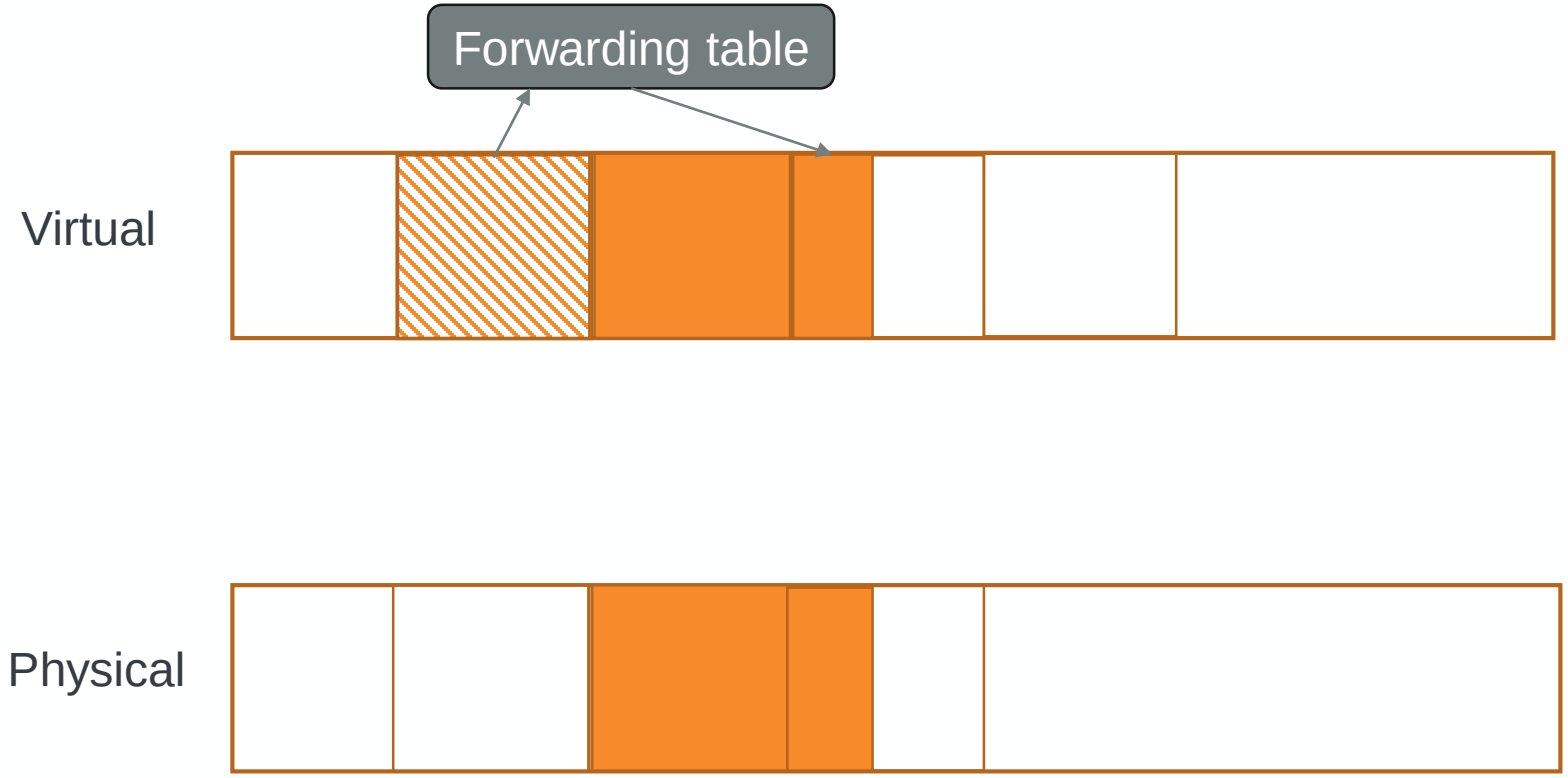
criteo

Relocation Phase



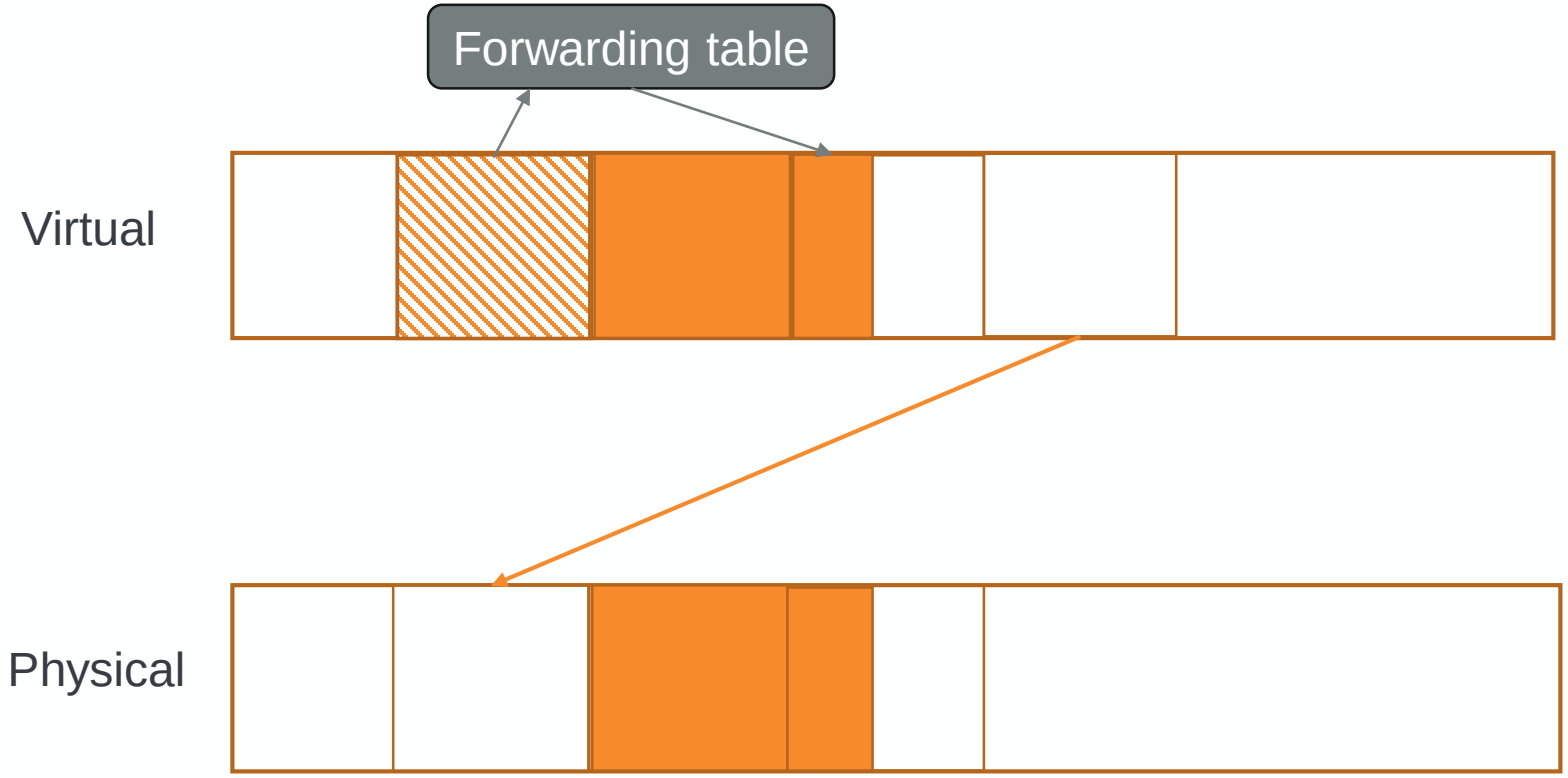
criteo

Relocation Phase



criteo

Relocation Phase



criteo

Relocation Phase

- Few chances mutators stall on accessing a ref as processing mostly dead pages

- Once object copy done, physical memory is released (Quick Release)
  - Can be immediately reused (remapped) to satisfy new allocations

- Pages evacuated are still mapped & protected to help remap phase
  - Cannot be released until all objects are remapped
  - Not a problem as we have a **huge** virtual address space

Remap Phase

- Traverse Object Graph and fixup references
- Execute LVB barrier for each object
- Self-Healing: fixup references using forward information
- As we traverse again, mark for the next phase
  - Mark & Remap phases are folded!

Remap – Kernel module

- Algorithm requires a sustainable rate or remapping operations

- Linux limitations:
  - TLB invalidation
  - Only 4KB pages can be remapped
  - Single threaded remapping (write lock)

- Kernel module implements API for the Zing JVM to increase significantly the remapping rate

- Implements also virtual address aliasing for addressing objects with metadata

Generational

- Young & Old collections done by same algorithm and can be concurrent
- Size of the generation are dynamically adjusted
- Card Marking with write barrier (Stored Value Barrier)
- Old collection is based on young-to-old roots generated by previous young cycle
- Young collection will perform card scanning per page
  - hold an eventual concurrent Old collection per page scanned

C4 @ Criteo

- Used by Hadoop Name Node
- 580GB Heap
- Very hard to tune with G1
- No issue so far regarding GC since production roll out (Oct 2017)

criteo

Z GC

criteo

Z GC

- Non generational
- Region based (zPages, dynamically sized)
- Concurrent Marking, Compaction, Ref processing
- Use Colored Pointers & Read/Load Barrier

```
mov     r10,QWORD PTR [r11+0xb0]
test    QWORD PTR [r15+0x20],r10
jne     0x00007f9594cc54b5
```

- Self-Healing
  - Cooperation between mutator threads & GC threads
- Experimental in JDK 11 (`-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`)

criteo

Z GC

Cliff Click
@cliff_click

Abonné

Ok, quick read of ZGC and it basically screams "GPGC done by Oracle". Can anybody closer to the truth fill in any details?

Traduire le Tweet

22:41 - 27 sept. 2018

5 Retweets **12** J'aime

2 5 12

criteo

Z GC phases:

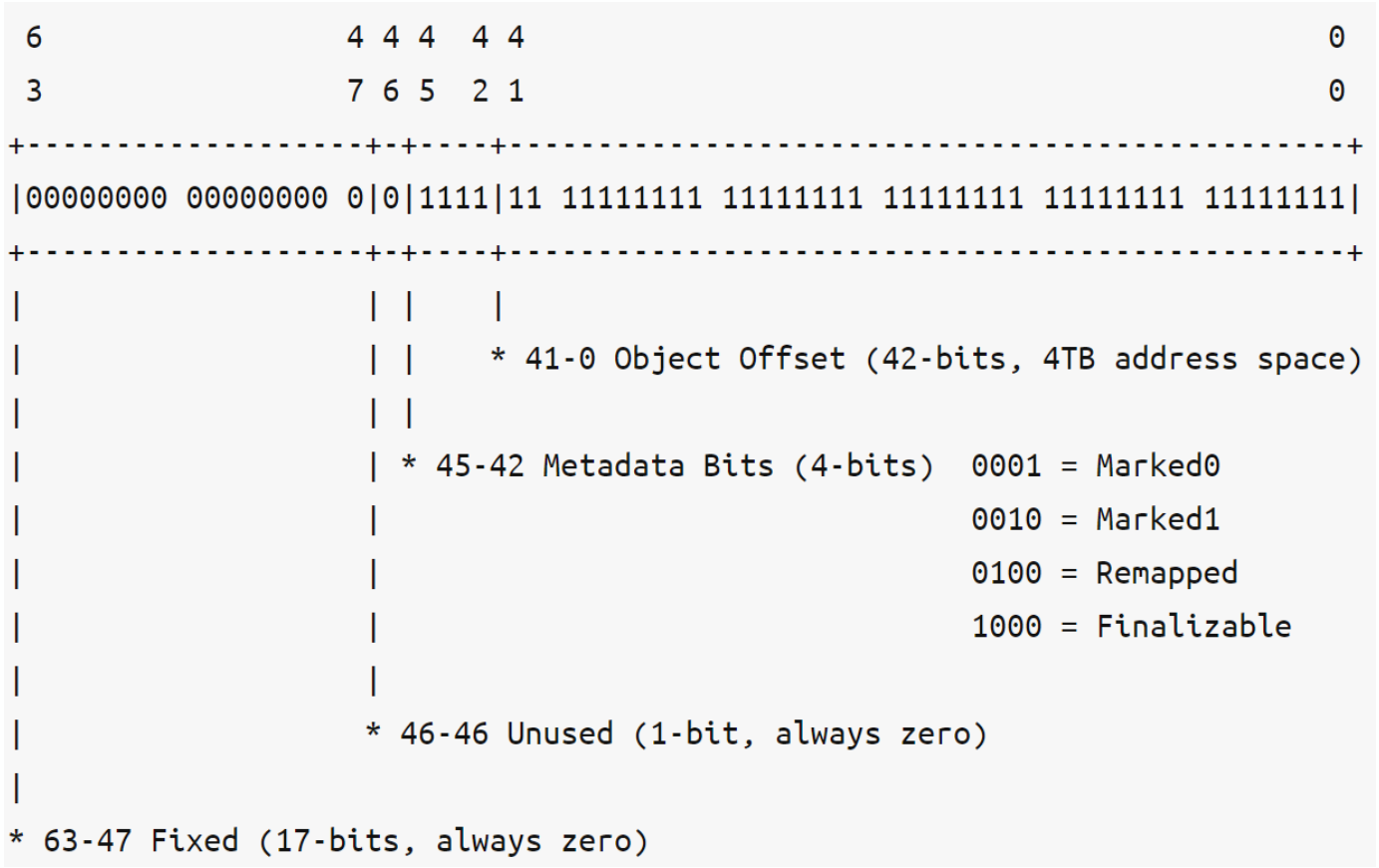
- Initial Mark (STW)
- Concurrent Mark/Remap
- Final Mark (STW)
- Concurrent Prepare for Relocation
- Start Relocate (STW)
- Concurrent Relocate

Colored Pointers

- Store metadata in unused bits of reference address

- 42 bits for addressing (4TB)
  - 44 bits (16TB) for JDK 13

- 4 bits for metadata
  - Marked0
  - Marked1
  - Remapped
  - Finalizable



Multi-Mapping

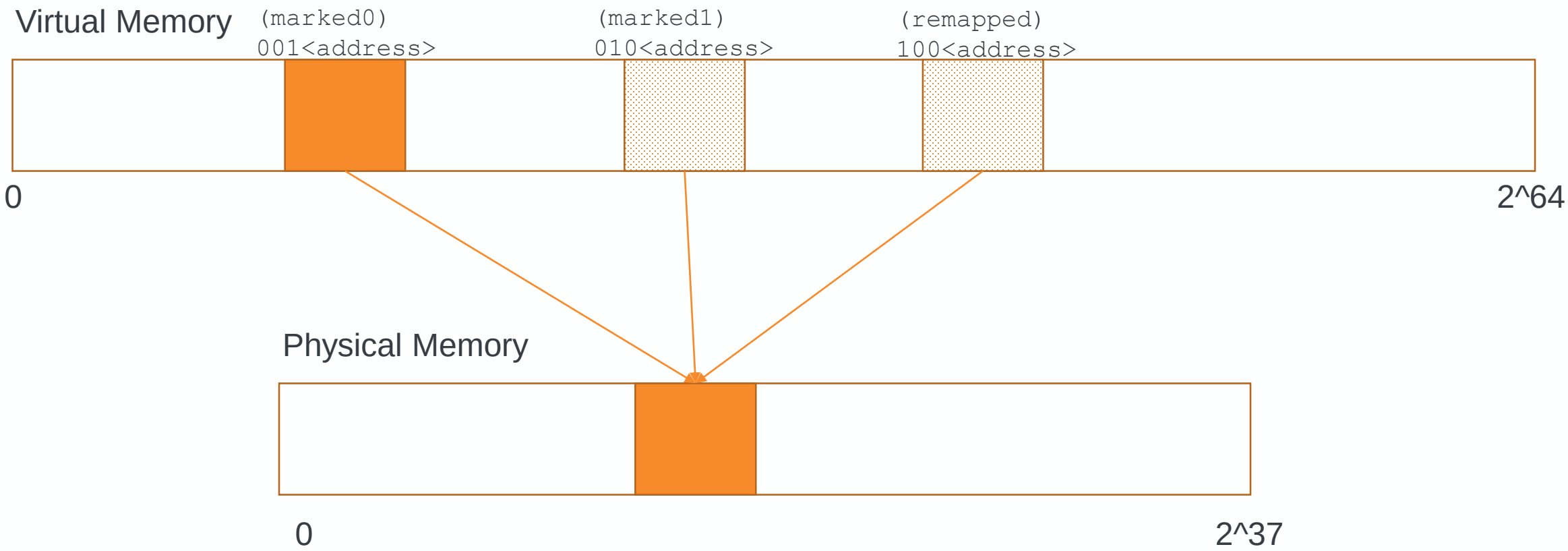
- Colored pointers needs to be unmasked for dereferencing
  - Some HW support masking (SPARC, Aarch64))
  - On linux/windows, overhead if done with classical instructions

- Only one view is active at any point

- Plenty of Virtual Space

criteo

Multi-Mapping



criteo

Memory Usage

Hi there,

I'm trying ZGC on trivial workloads, and I have a question about footprint. The workload runs with -Xms16g -Xms16g, but the RSS figures are at least 3x larger

```
VmPeak: 18256721392 kB
VmSize: 18256721392 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:  50729036 kB
VmRSS:  50729036 kB
RssAnon:          369700 kB
RssFile:           27688 kB
RssShmem:       50331648 kB
```

Is this because ZGC maps the same physical space with multiple virtual mappings? Or is it a bug?

Thanks,
-Aleksey

criteo

Page Allocations

- Pages are multiple of 2MB

- 3 different groups
  - Small: 2MB pages with object size <= 256KB
  - Medium: 32MB pages with object size <= 4MB
  - Large: 2MB pages, objects span over multiple of them

- Objects in Large group are meant to not to be relocated (too expensive)

Difference between C4 & Z GC

- Unmasking ref addresses
  - C4: Kernel module aliasing
  - Z: Multi-mapping or HW support

- Pages & Relocation
  - C4:
    - Page are fixed (2MB)
    - relocation for large objects by remapping
  - Z:
    - zPages are dynamic, a zPage can be 100MB large
    - No relocation for large objects

How to choose a GC algorithm

criteo

Low latency GCs

- You have to run on Windows
  - Shenandoah
- Battlefield tested GC (maturity)
  - C4
  - Shenandoah
- Minimizing any kind of JVM pauses
  - C4
  - Z
- You don’t want pay for it:
  - Shenandoah
  - Z

References

References GC Basics

- Java Garbage Collection distilled by Martin Thompson
- The Java GC mini book
- Oracle's white paper on JVM memory management & GC
- What differences JVM makes by Nitsan Wakart
- Memory Management Reference
- IBM Pause-Less GC

References Shenandoah

- Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK
- Shenandoah: The Garbage Collector That Could by Aleksey Shipilev
- Shenandoah GC Wiki
- Load Reference Barriers by Roman Kennke
- Eliminating forward pointer word by Roman Kennke

References C4

- The Pauseless GC algorithm (2005)
- C4: Continuously Concurrent Compacting Collector (2011)
- Azul GC in Detail by Charles Humble
- 2010 version source code

References ZGC

- ZGC - Low Latency GC for OpenJDK by Per Liden
- Java's new Z Garbage Collector (ZGC) is very exciting by Richard Warburton
- A first look into ZGC by Dominik Inführ
- Architectural Comparison with C4/Pauseless
- ZGC Heap Size and RSS counters

Thank You!

@jpbempel

criteo