TRASHTALK

Serial
Parallel
CMS
Epsilon
G1
ZGC
Shenandoah
C4

EXPLORING THE MEMORY MANAGEMENT IN THE JVM

ABOUT ME.

Gerrit Grunwald | Developer Advocate | Azul

𝕏 @hansolo_

azul

MEMORY MANAGEMENT

IN THE JVM...

IS AUTOMATIC RIGHT...?

SO...WHY CARE...?

MEMORY MANAGEMENT

Why you should care…

🗑️ Impact on application performance

MEMORY MANAGEMENT

Why you should care…

- Impact on application performance
- Impact on application responsiveness

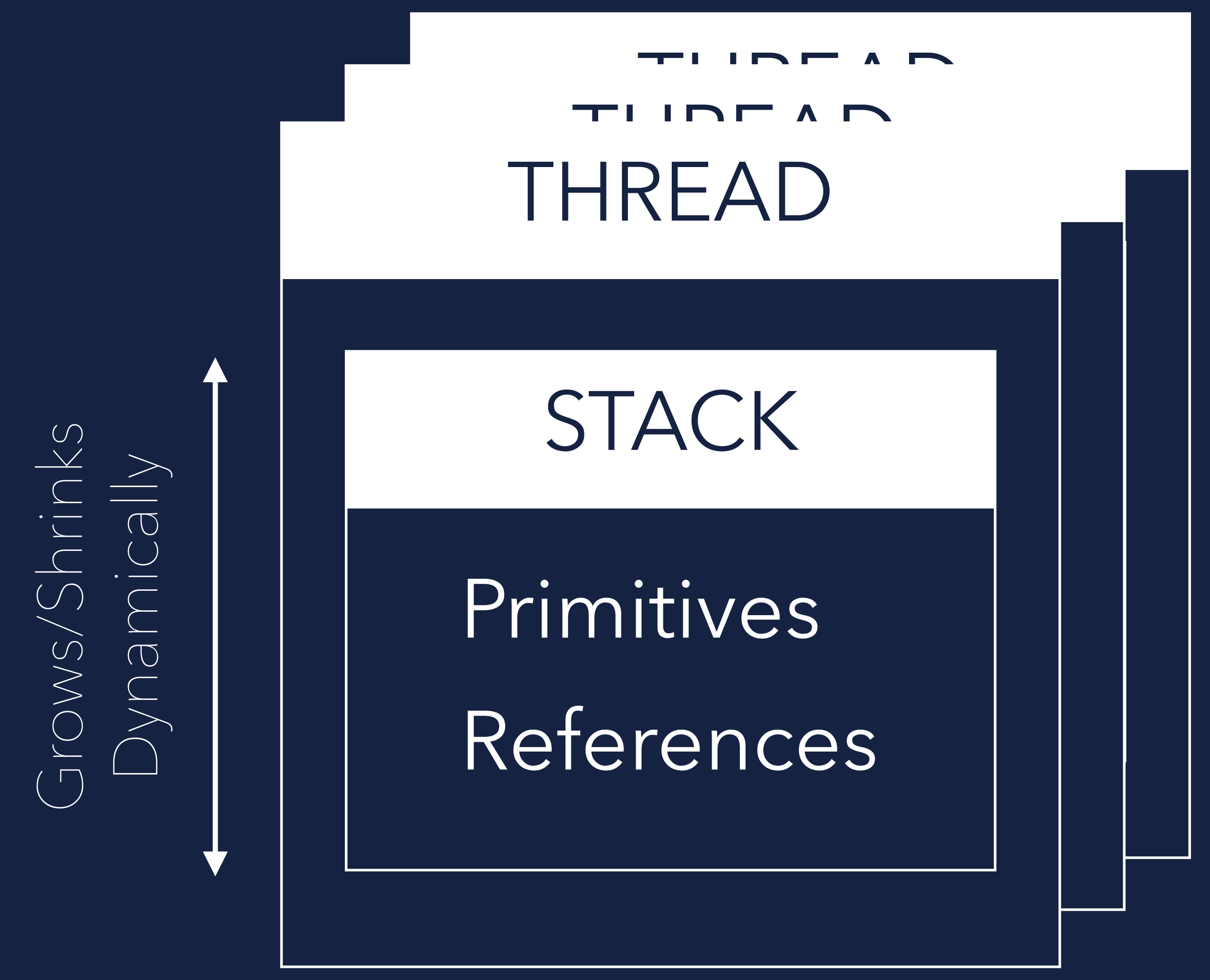
MEMORY MANAGEMENT

Why you should care...

- Impact on application performance
- Impact on application responsiveness
- Impact on system requirements

MEMORY MANAGEMENT

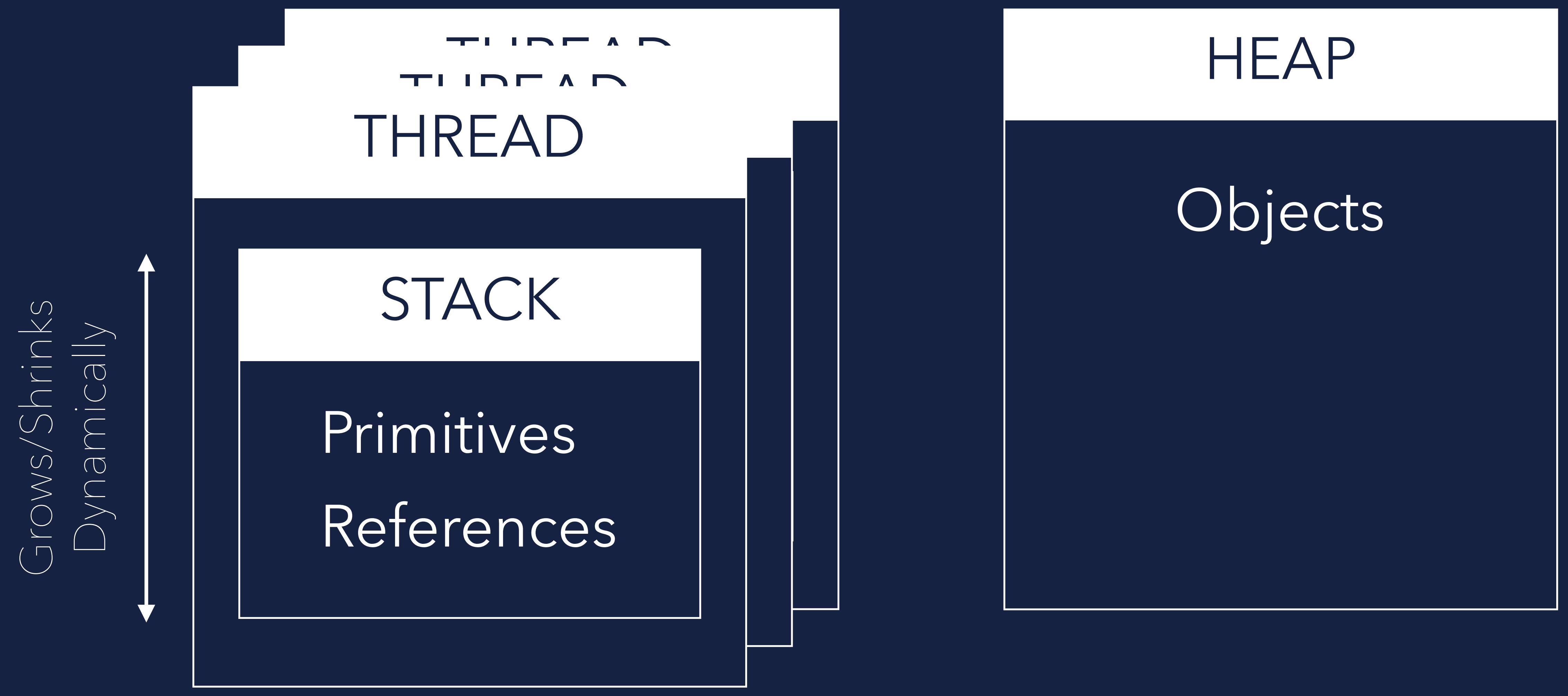
Stack, Heap and Metaspace



Local access -> thread safe

MEMORY MANAGEMENT

Stack, Heap and Metaspace



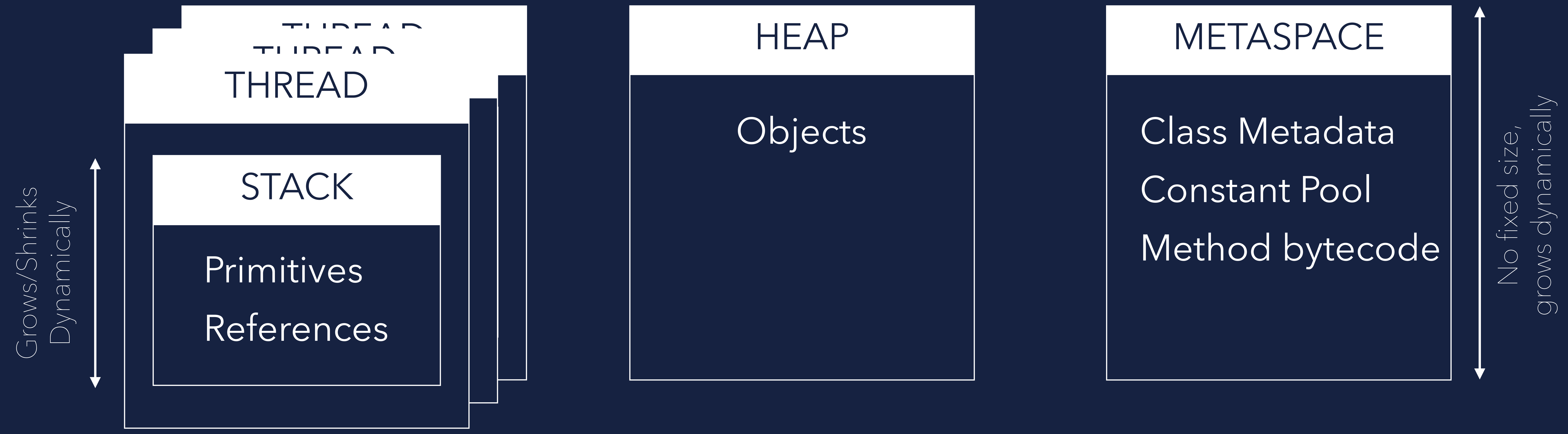
Local access -> thread safe

Shared access -> Not thread safe

Needs Garbage Collection

MEMORY MANAGEMENT

Stack, Heap and Metaspace



THREAD

THREAD

THREAD

STACK

Primitives

References

Grows/Shrinks Dynamically

HEAP

Objects

METASPACE

Class Metadata

Constant Pool

Method bytecode

No fixed size, grows dynamically

Local access -> thread safe

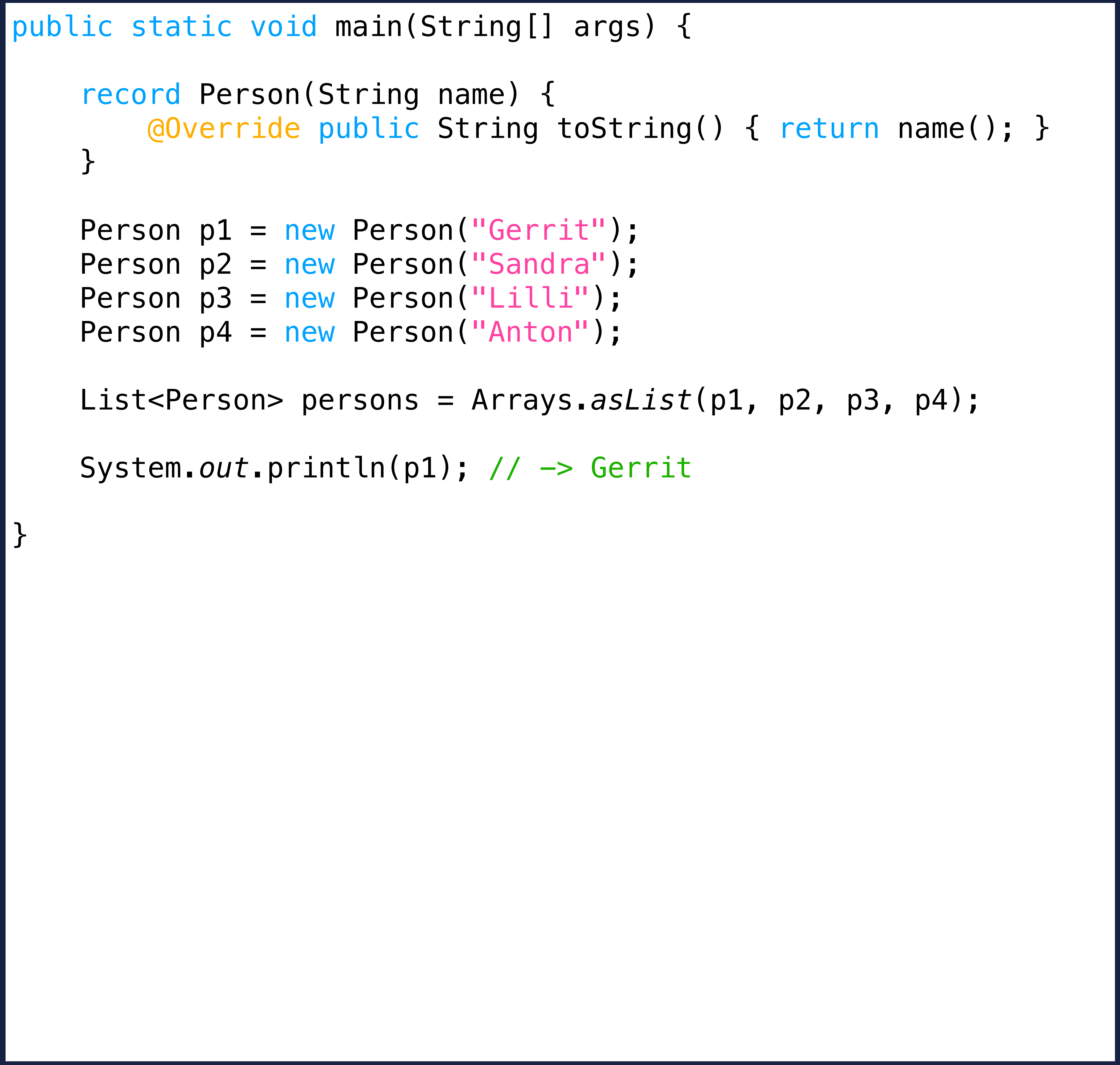
Shared access -> Not thread safe

Needs Garbage Collection

Contains info needed for JVM to work with classes

MEMORY MANAGEMENT

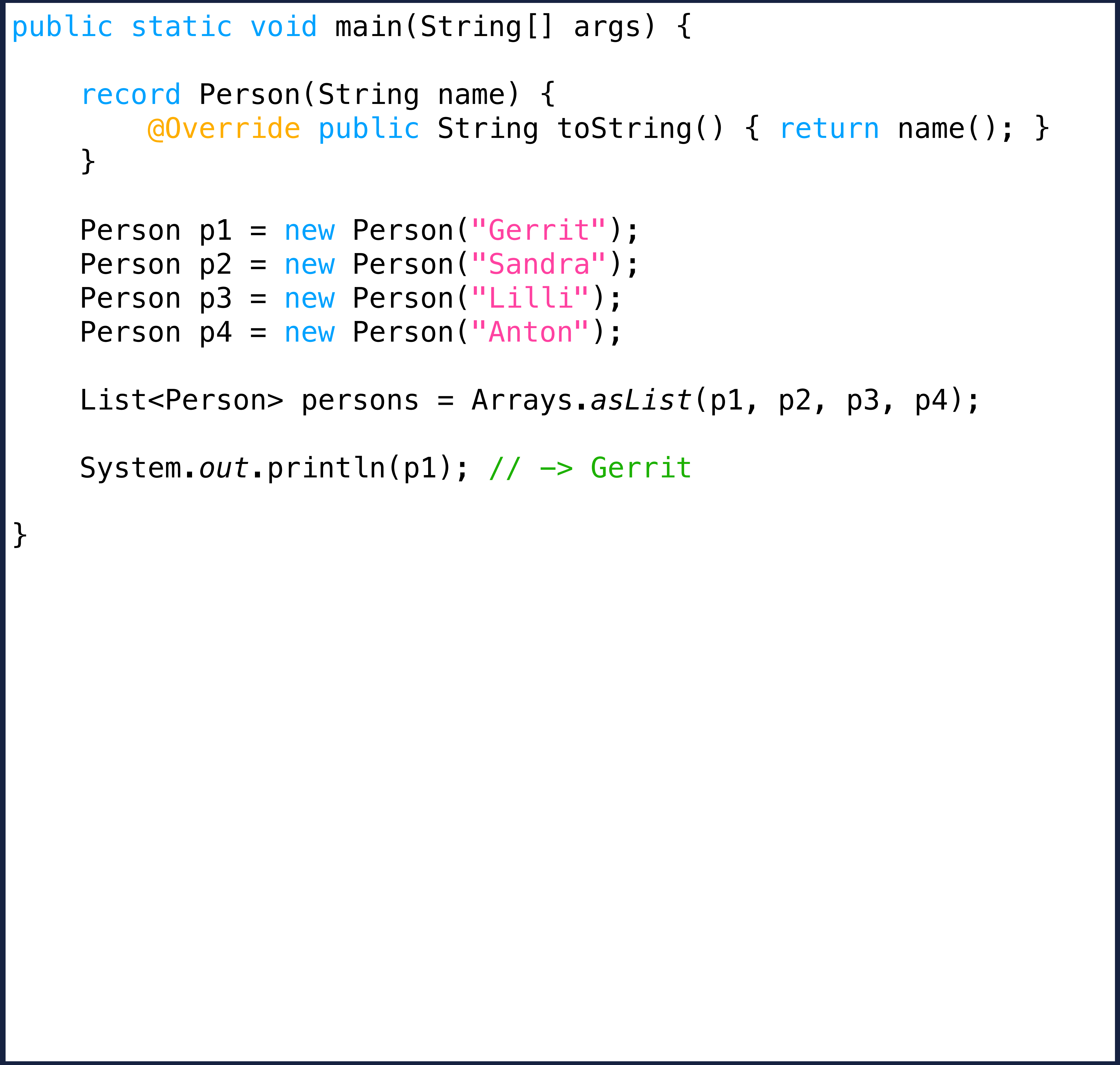
In the JVM...



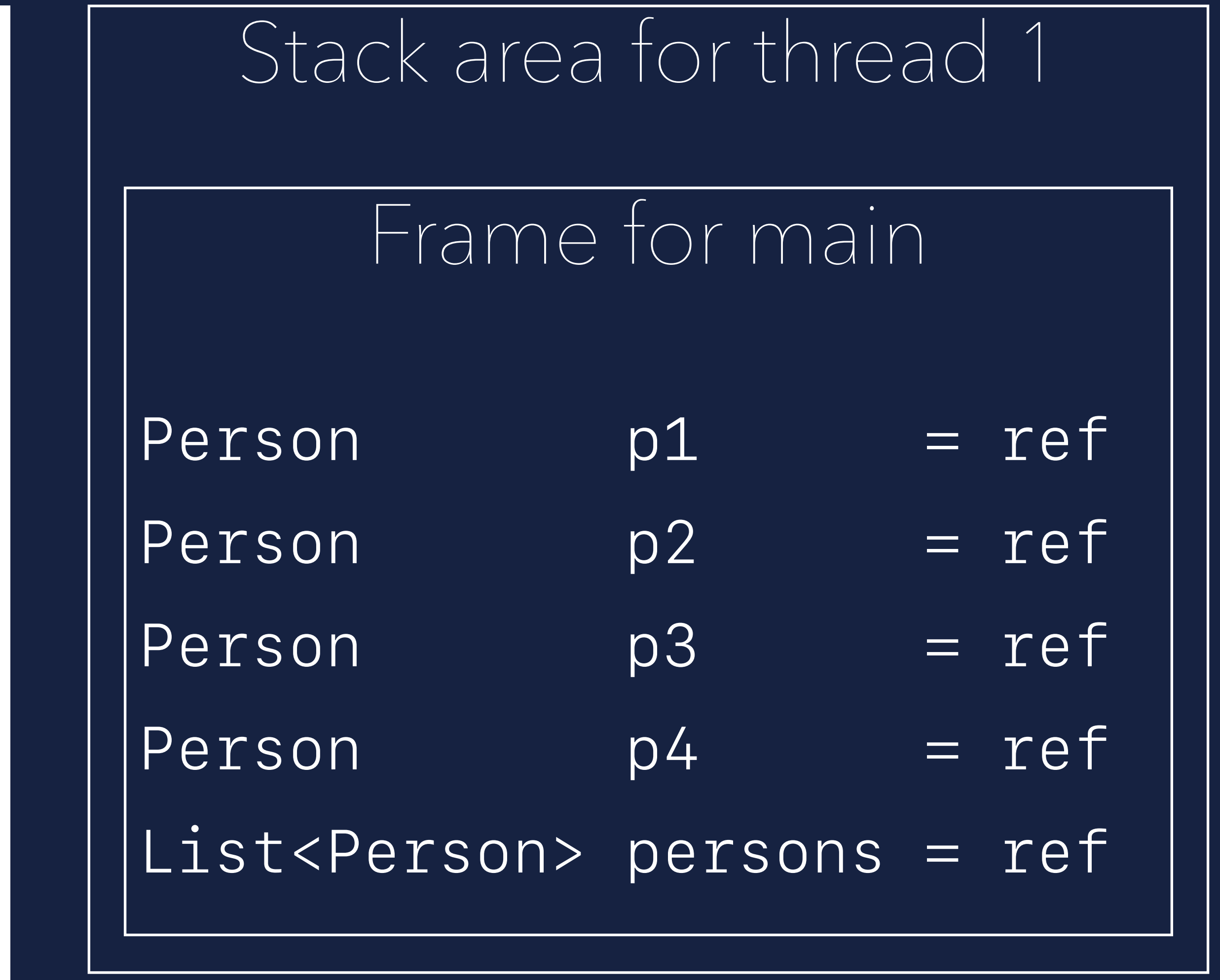
```java
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

}
```

MEMORY MANAGEMENT

In the JVM...



```java
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

}
```



Stack area for thread 1

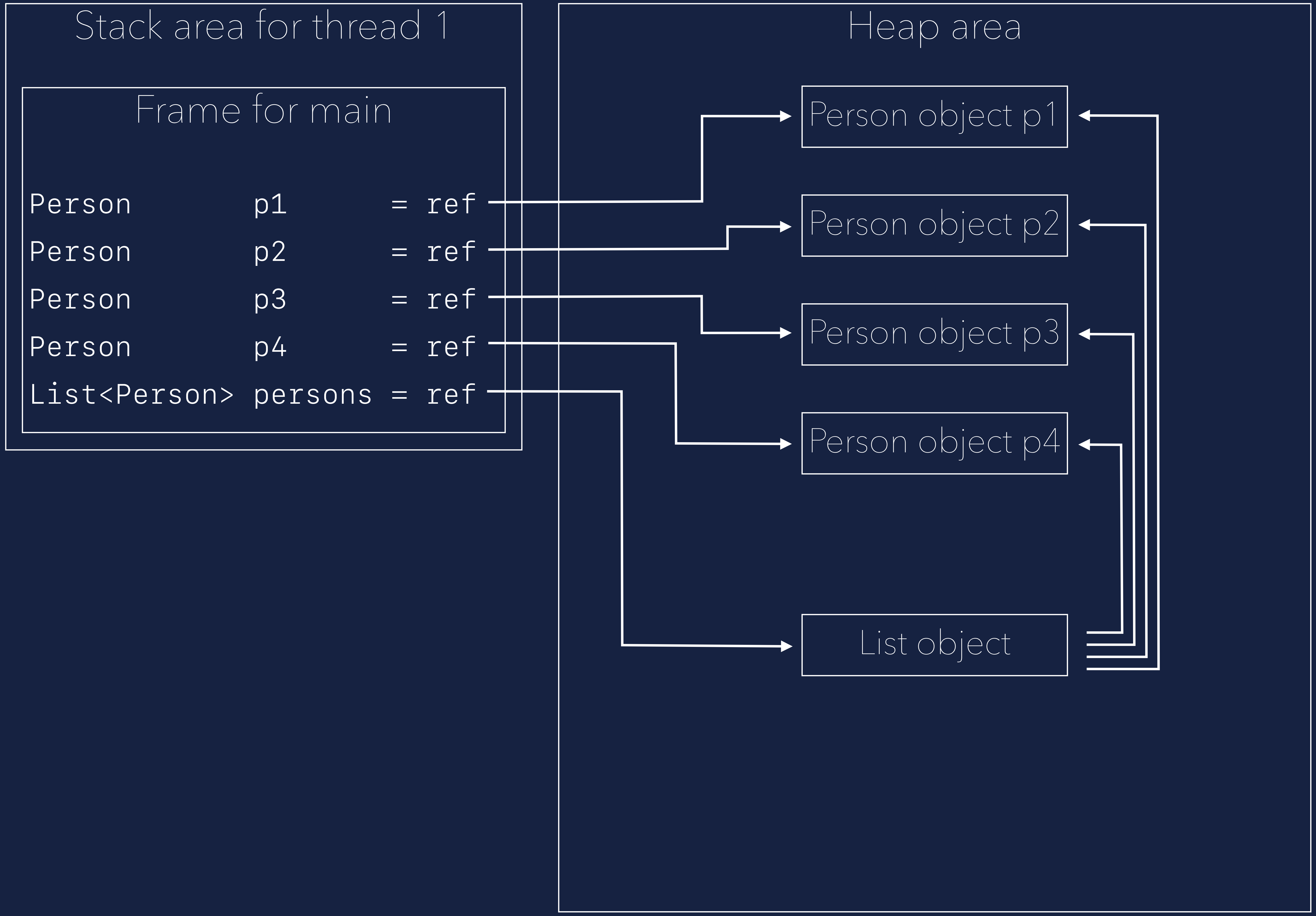
Frame for main

```
Person       p1      = ref
Person       p2      = ref
Person       p3      = ref
Person       p4      = ref
List<Person> persons = ref
```

MEMORY MANAGEMENT

In the JVM...

```
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

}
```



Stack area for thread 1

Frame for main

```
Person        p1      = ref
Person        p2      = ref
Person        p3      = ref
Person        p4      = ref
List<Person>  persons = ref
```

Heap area

Person object p1

Person object p2

Person object p3

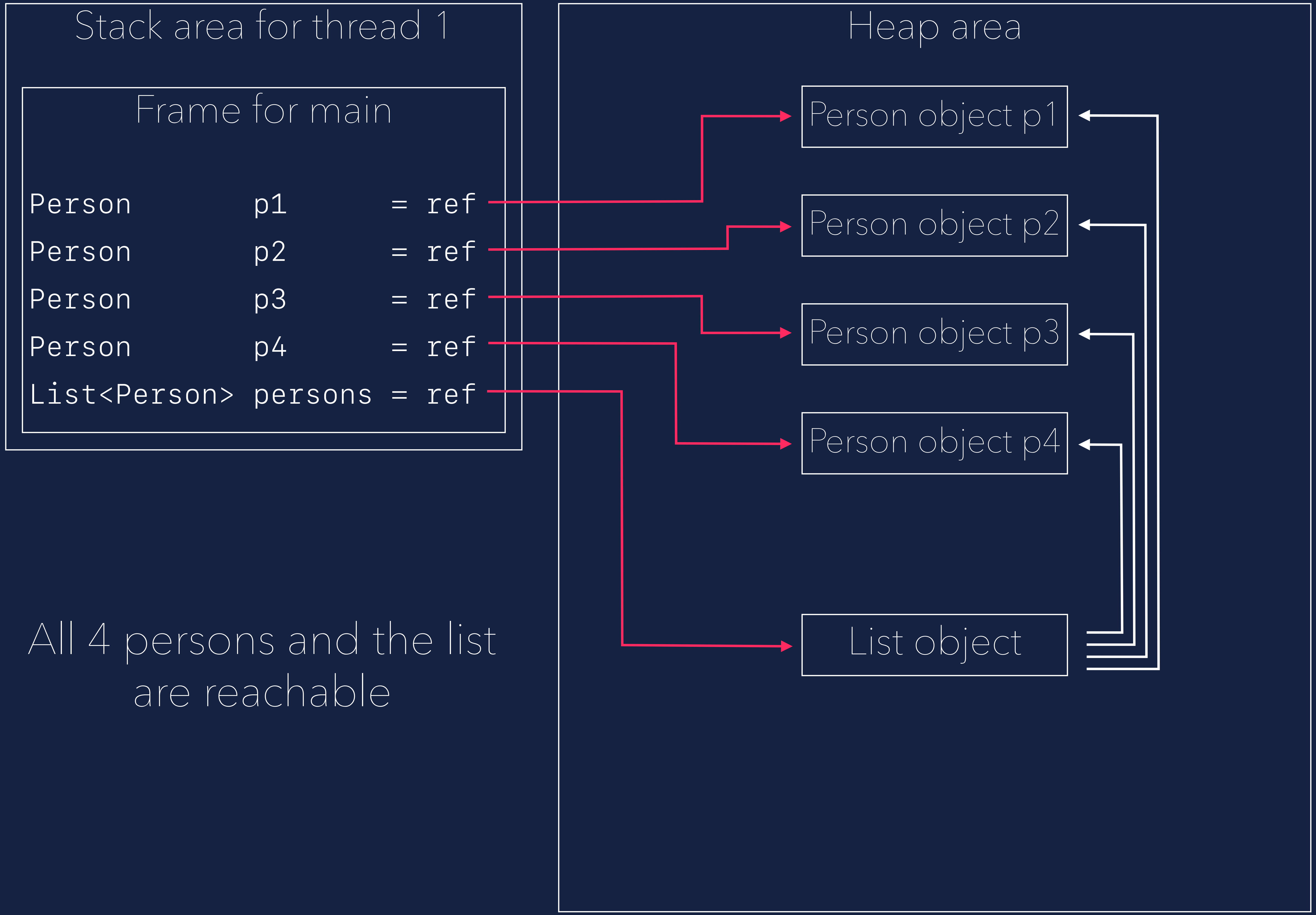
Person object p4

List object

MEMORY MANAGEMENT

In the JVM...

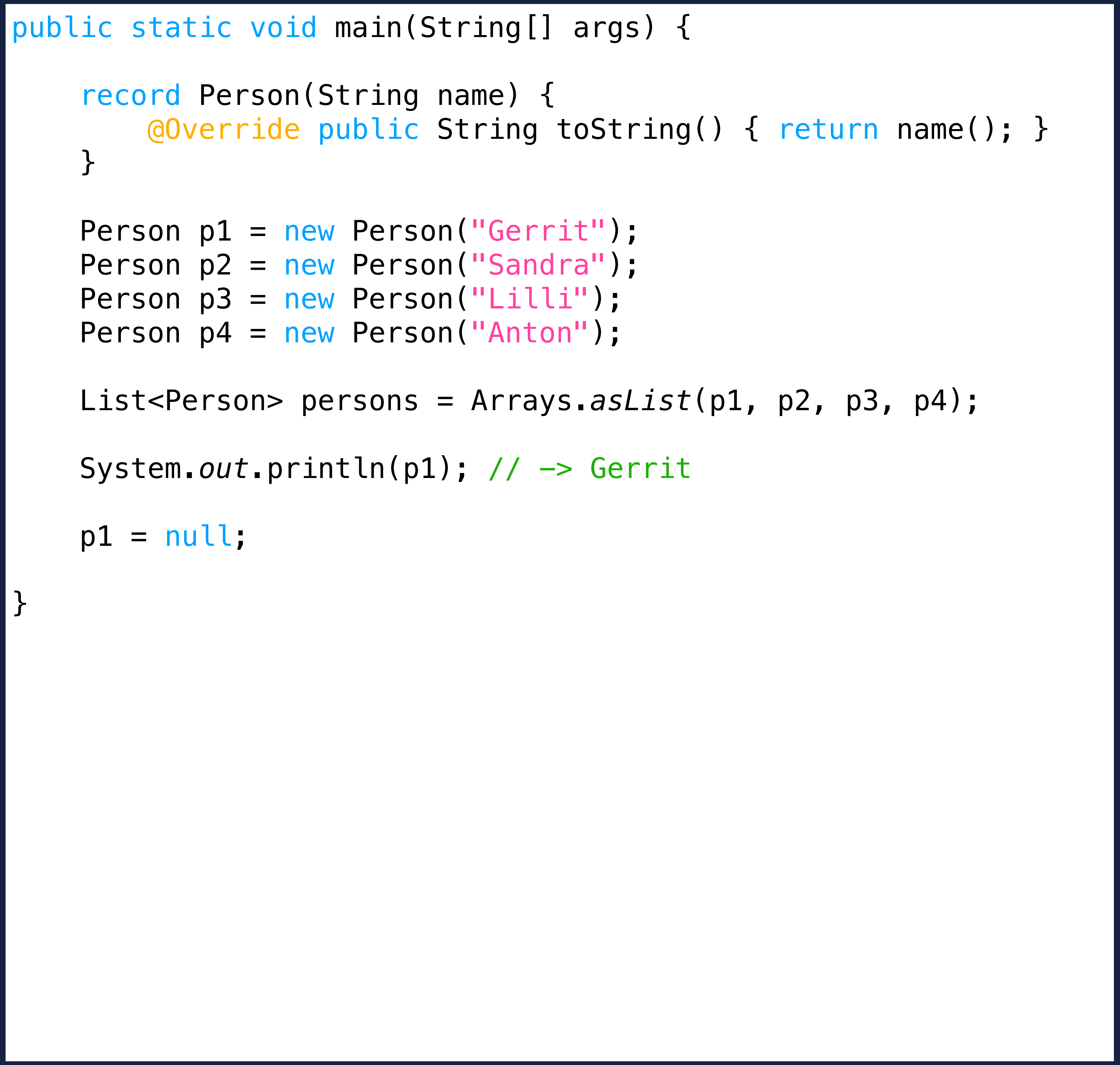
```java
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

}
```



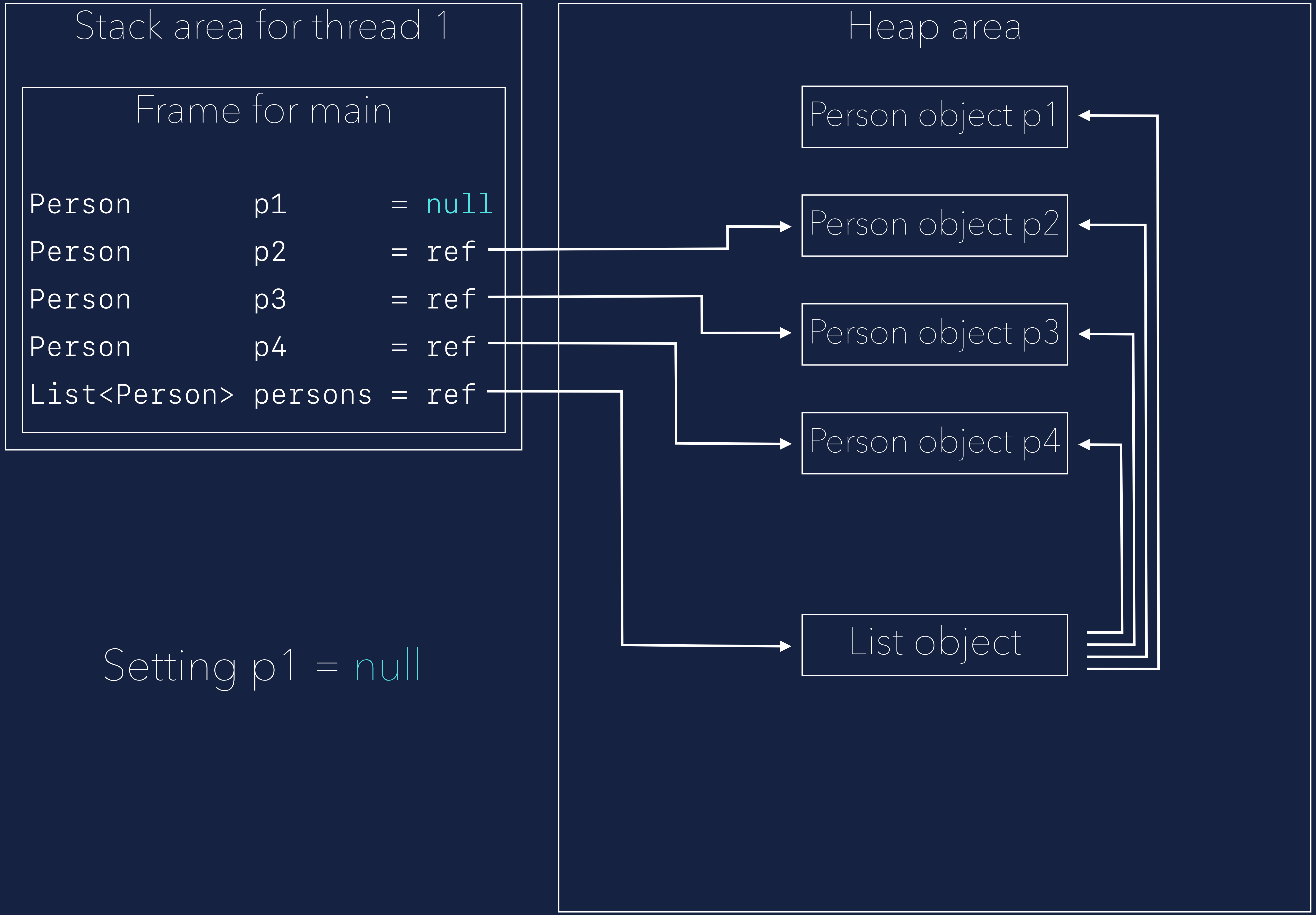
All 4 persons and the list are reachable

MEMORY MANAGEMENT

In the JVM...



```java
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

    p1 = null;

}
```

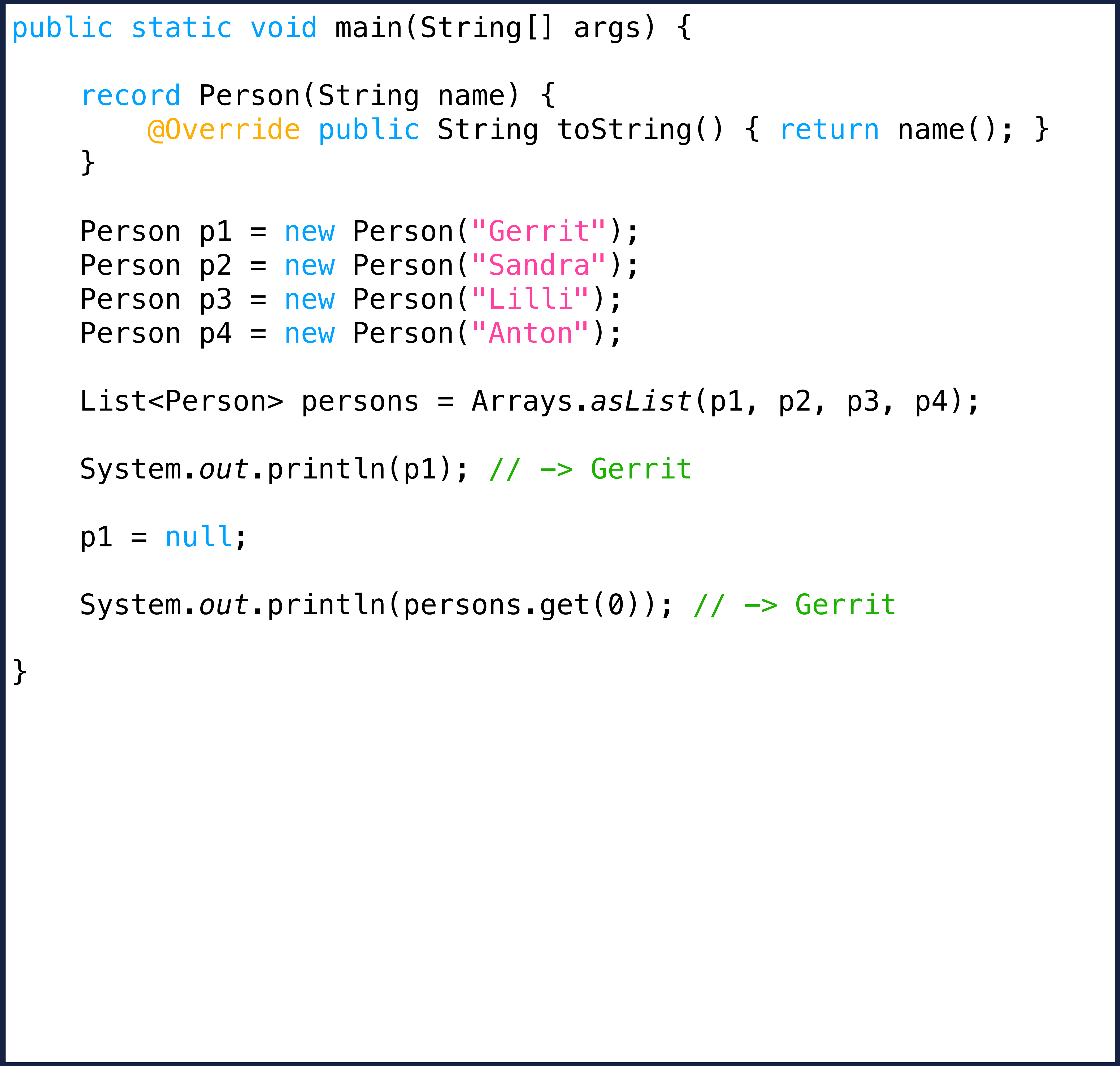


```
Person         p1      = null
Person         p2      = ref
Person         p3      = ref
Person         p4      = ref
List<Person>   persons = ref
```

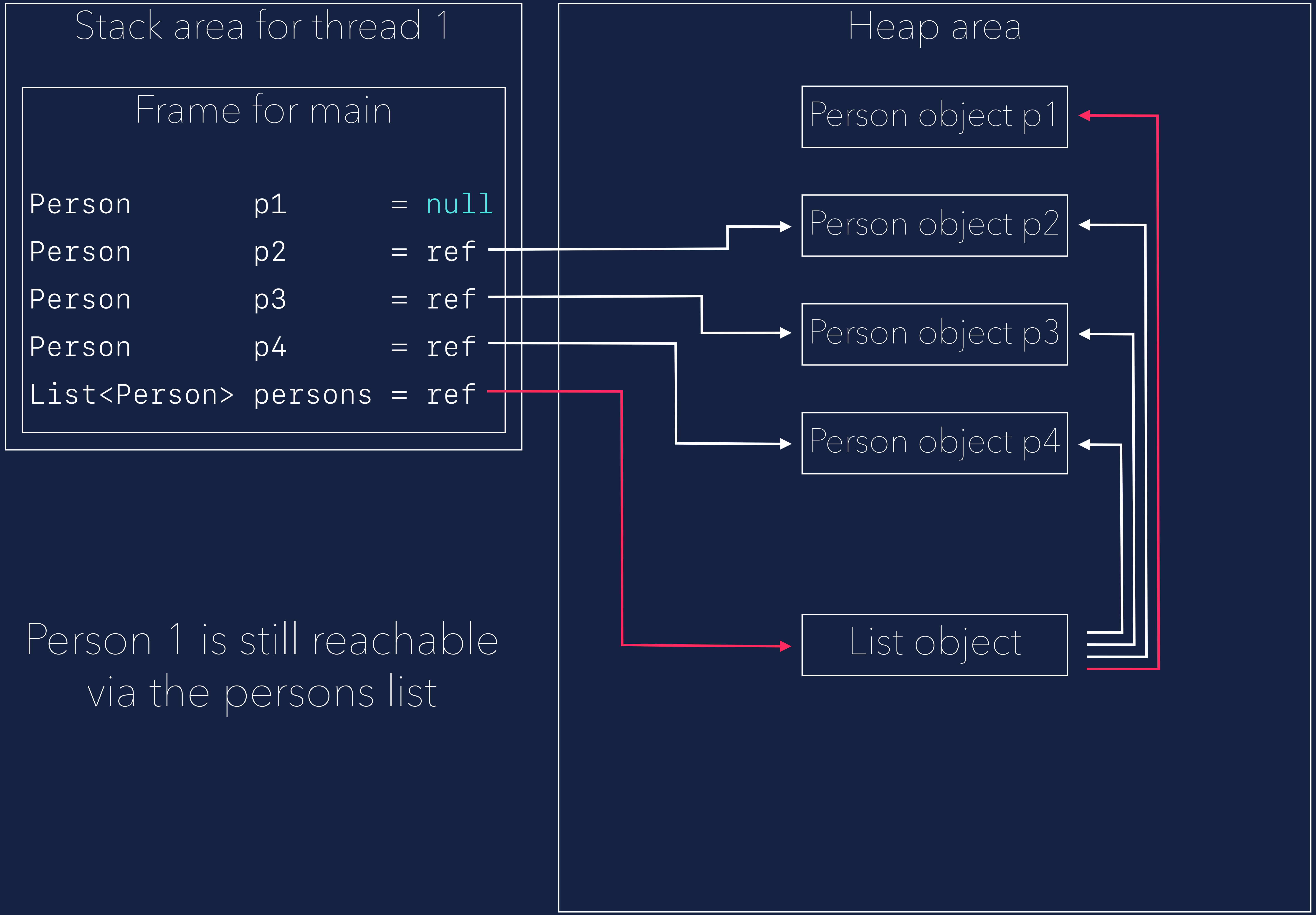
Setting p1 = null

MEMORY MANAGEMENT

In the JVM…



```
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

    p1 = null;

    System.out.println(persons.get(0)); // -> Gerrit

}
```



Stack area for thread 1

Frame for main

```
Person        p1       = null
Person        p2       = ref
Person        p3       = ref
Person        p4       = ref
List<Person>  persons  = ref
```

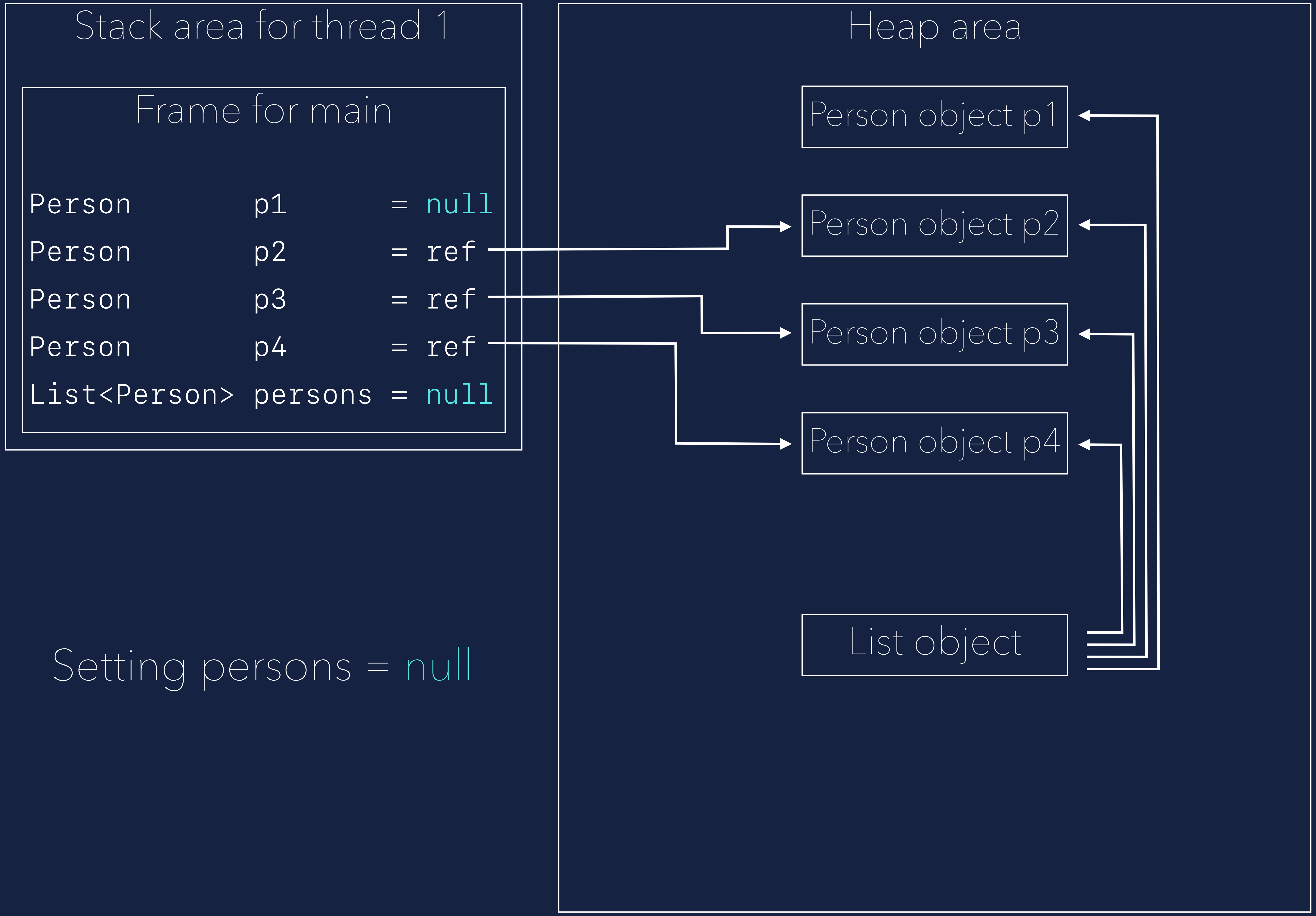
Heap area

Person 1 is still reachable via the persons list

MEMORY MANAGEMENT

In the JVM...

```
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

    p1 = null;

    System.out.println(persons.get(0)); // -> Gerrit

    persons = null;
}
```

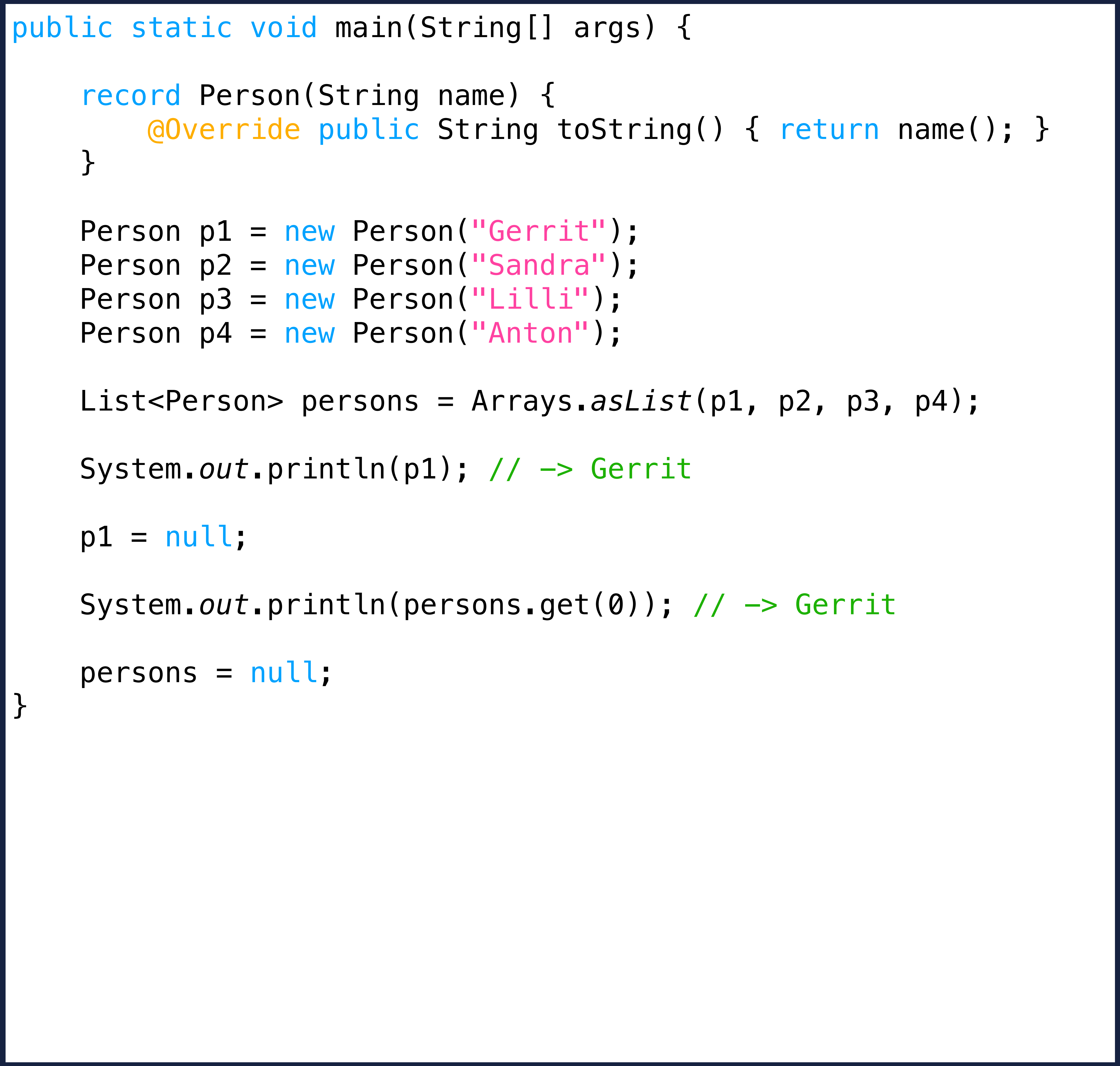


```
Person         p1       = null
Person         p2       = ref
Person         p3       = ref
Person         p4       = ref
List<Person>   persons  = null
```

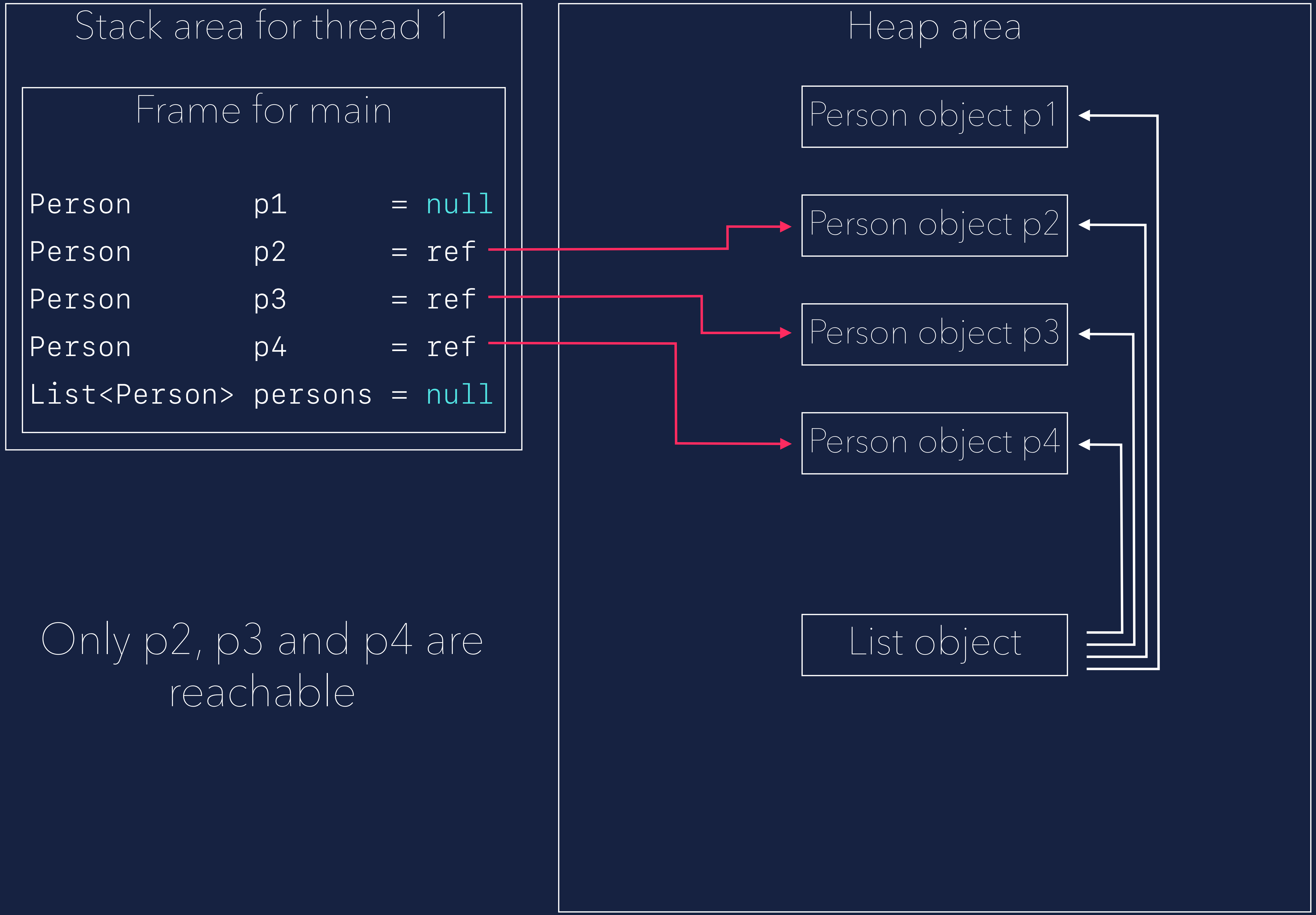
Setting persons = null

MEMORY MANAGEMENT

In the JVM...



```java
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

    p1 = null;

    System.out.println(persons.get(0)); // -> Gerrit

    persons = null;
}
```



Stack area for thread 1

Frame for main

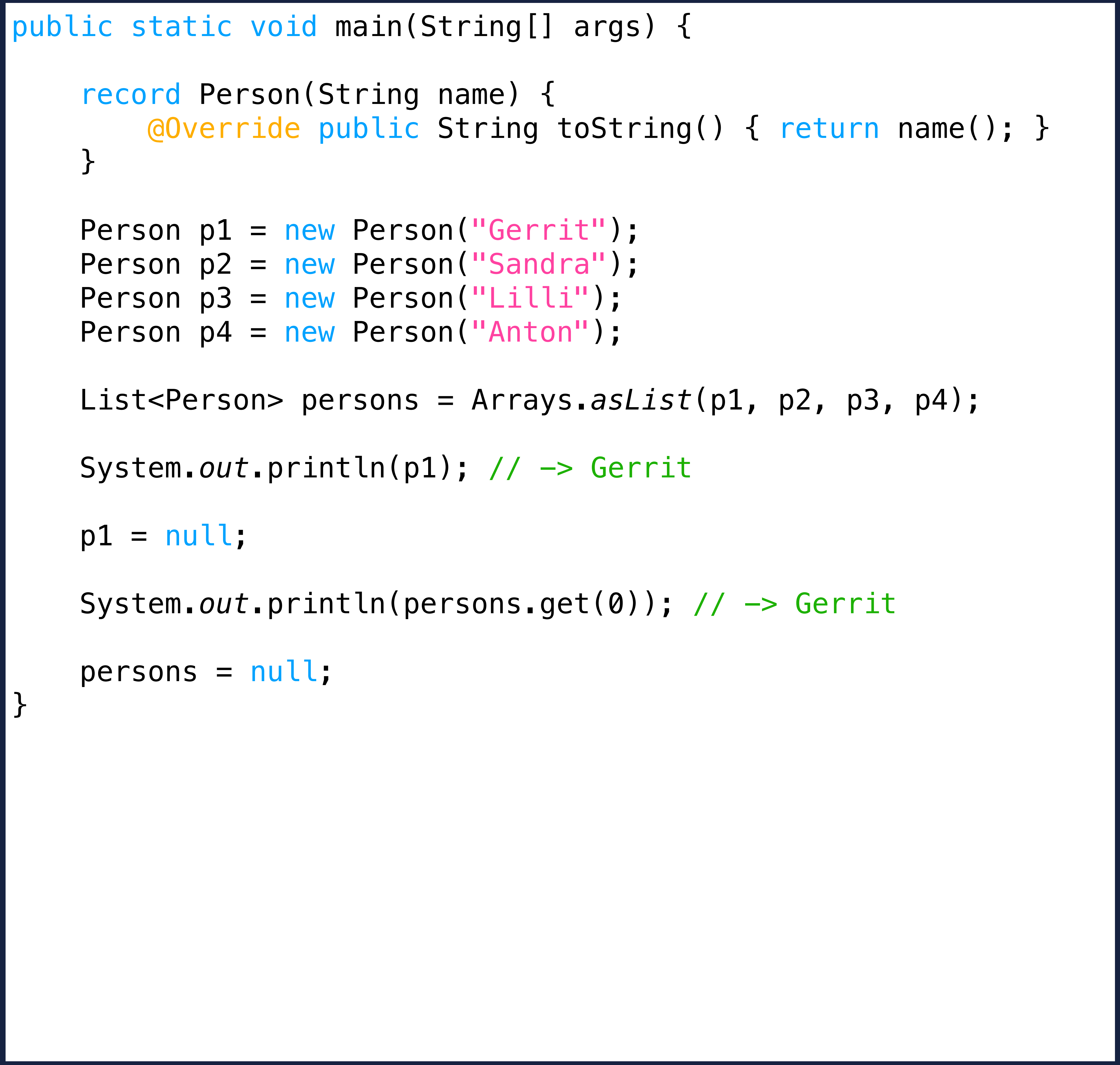
```
Person        p1      = null
Person        p2      = ref
Person        p3      = ref
Person        p4      = ref
List<Person>  persons = null
```

Heap area

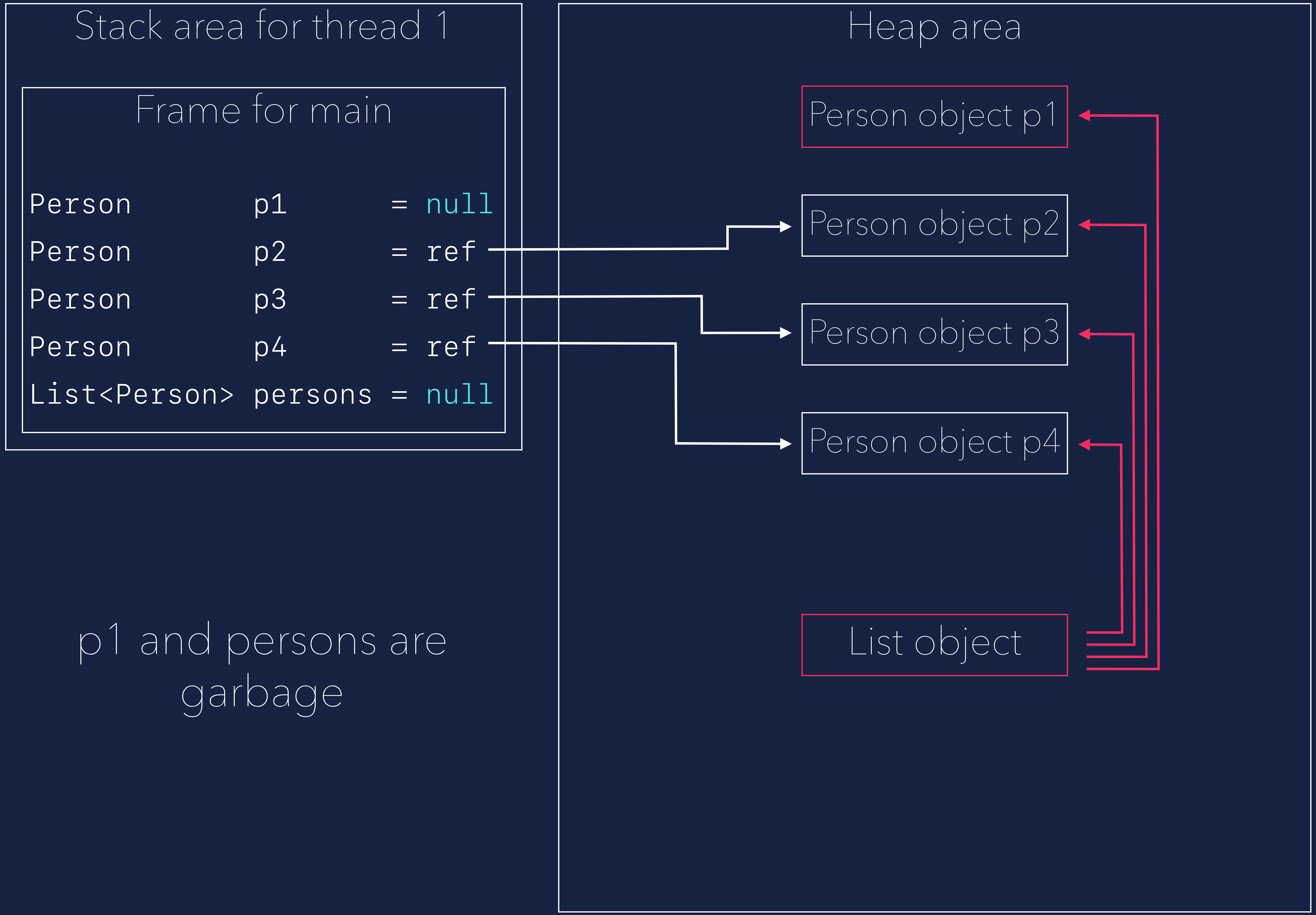
Only p2, p3 and p4 are reachable

MEMORY MANAGEMENT

In the JVM...



```
public static void main(String[] args) {

    record Person(String name) {
        @Override public String toString() { return name(); }
    }

    Person p1 = new Person("Gerrit");
    Person p2 = new Person("Sandra");
    Person p3 = new Person("Lilli");
    Person p4 = new Person("Anton");

    List<Person> persons = Arrays.asList(p1, p2, p3, p4);

    System.out.println(p1); // -> Gerrit

    p1 = null;

    System.out.println(persons.get(0)); // -> Gerrit

    persons = null;
}
```



```
Person        p1       = null
Person        p2       = ref
Person        p3       = ref
Person        p4       = ref
List<Person>  persons  = null
```

p1 and persons are garbage

HOW TO GET RID OF IT...?

GARBAGE COLLECTION

GARBAGE COLLECTION

What is it...

🗑️ Form of automatic memory management

GARBAGE COLLECTION

What is it...

- Form of automatic memory management
- Identifies and reclaims no longer used memory

GARBAGE COLLECTION

What is it…

- Form of automatic memory management
- Identifies and reclaims no longer used memory
- Ensures efficient memory utilisation

GARBAGE COLLECTION

What is it...

- Form of automatic memory management
- Identifies and reclaims no longer used memory
- Ensures efficient memory utilisation
- Frees user from managing the memory manually

PHASES

GARBAGE COLLECTION

Phases (precise collectors)

- Tracing
  Identify live objects on the heap

GARBAGE COLLECTION

Phases (precise collectors)

- ♻ Tracing
  Identify live objects on the heap
- ♻ Freeing
  Reclaim resources held by dead objects

GARBAGE COLLECTION

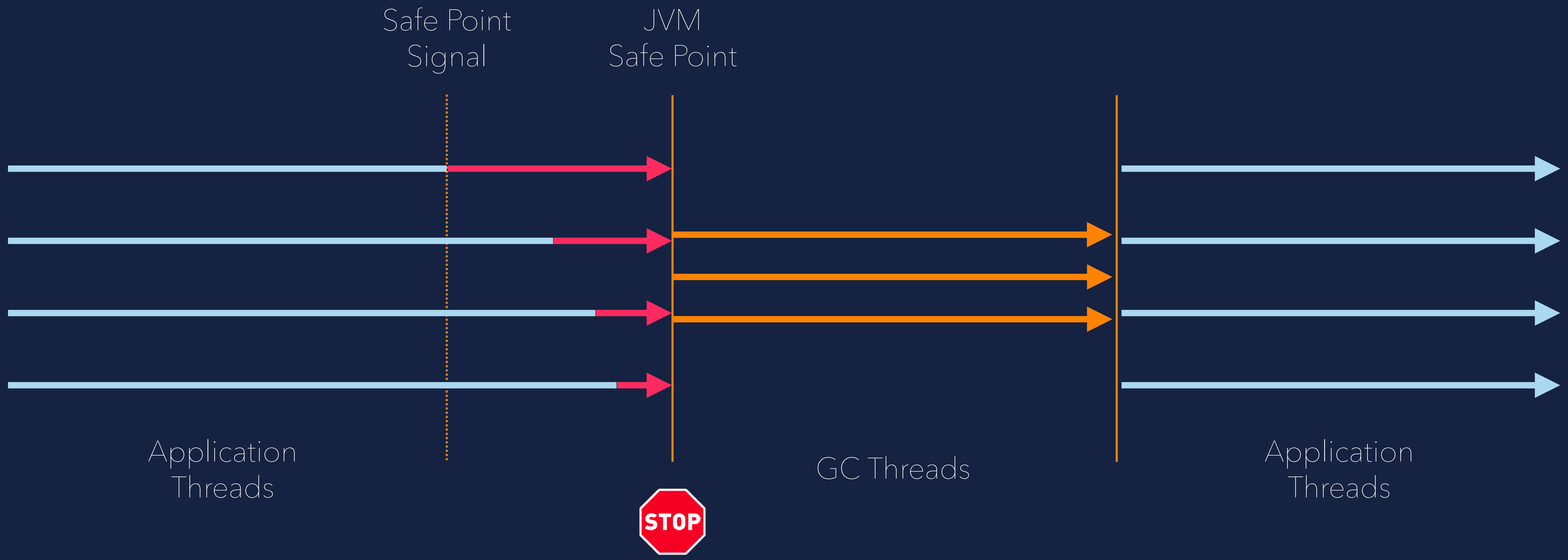
Phases (precise collectors)

- Tracing
  Identify live objects on the heap
- Freeing
  Reclaim resources held by dead objects
- Compaction
  Periodically relocate live objects

STOPPING THE W🛑RLD

STOPPING THE W STOP RLD

Halt of all application threads



COLLECTORS

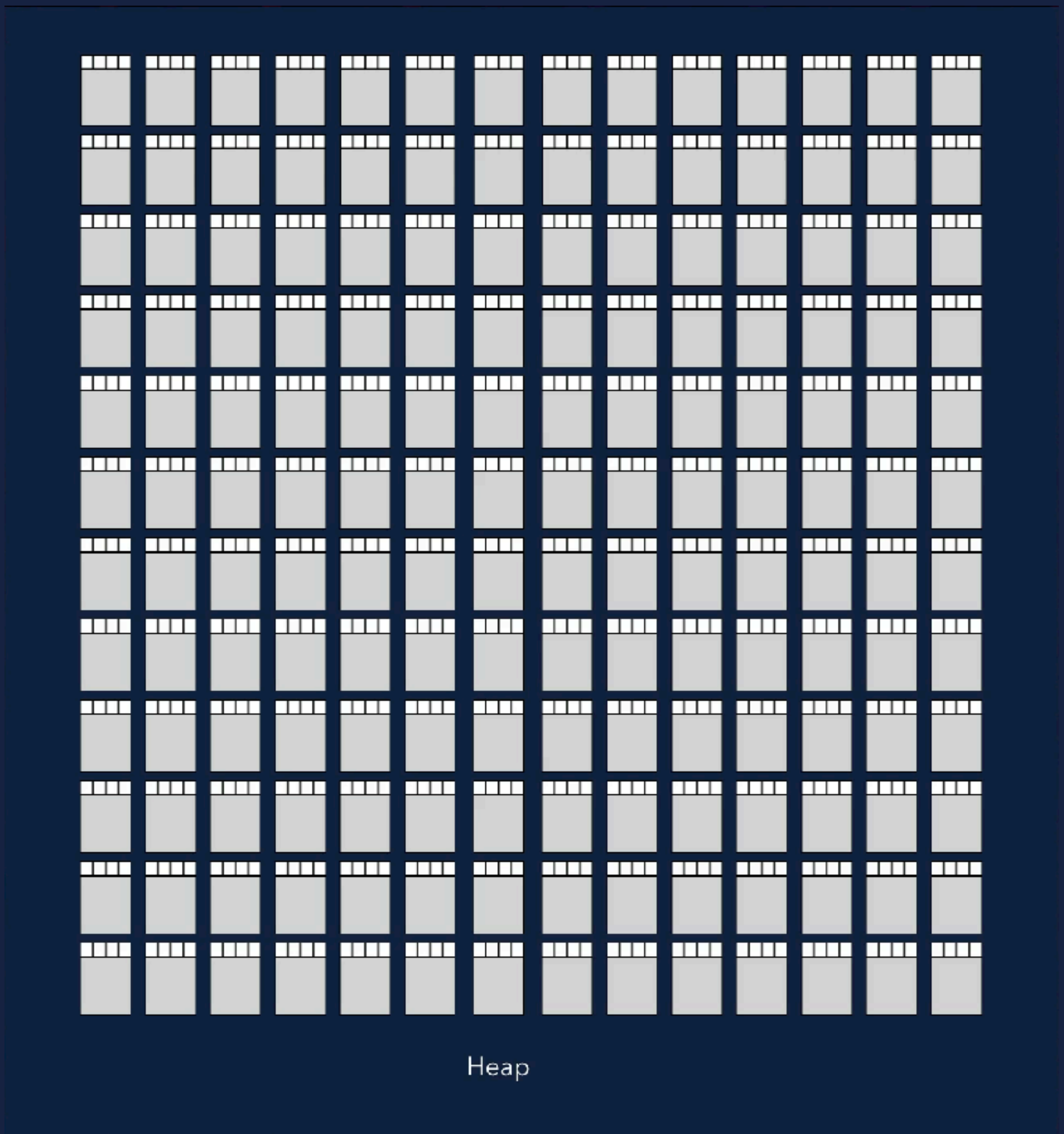
NON MOVING COLLECTOR

Mark & Sweep

NON MOVING COLLECTOR

Demo

1. Mutator allocates cells in Heap
2. Heap is out of memory -> GC
3. Mark all live cells
4. Free all dead cells
5. Unmark all live cells
6. Resume Mutator



- ⬜ Free Cell
- 🟦 Referenced Cell
- 🟪 Dereferenced Cell
- 🟦 Marked Cell
- 1 Referenced Cell (survived 1 GC)

⚠ Fragmentation

MOVING COLLECTOR

Compacting Collector & Copy Collector

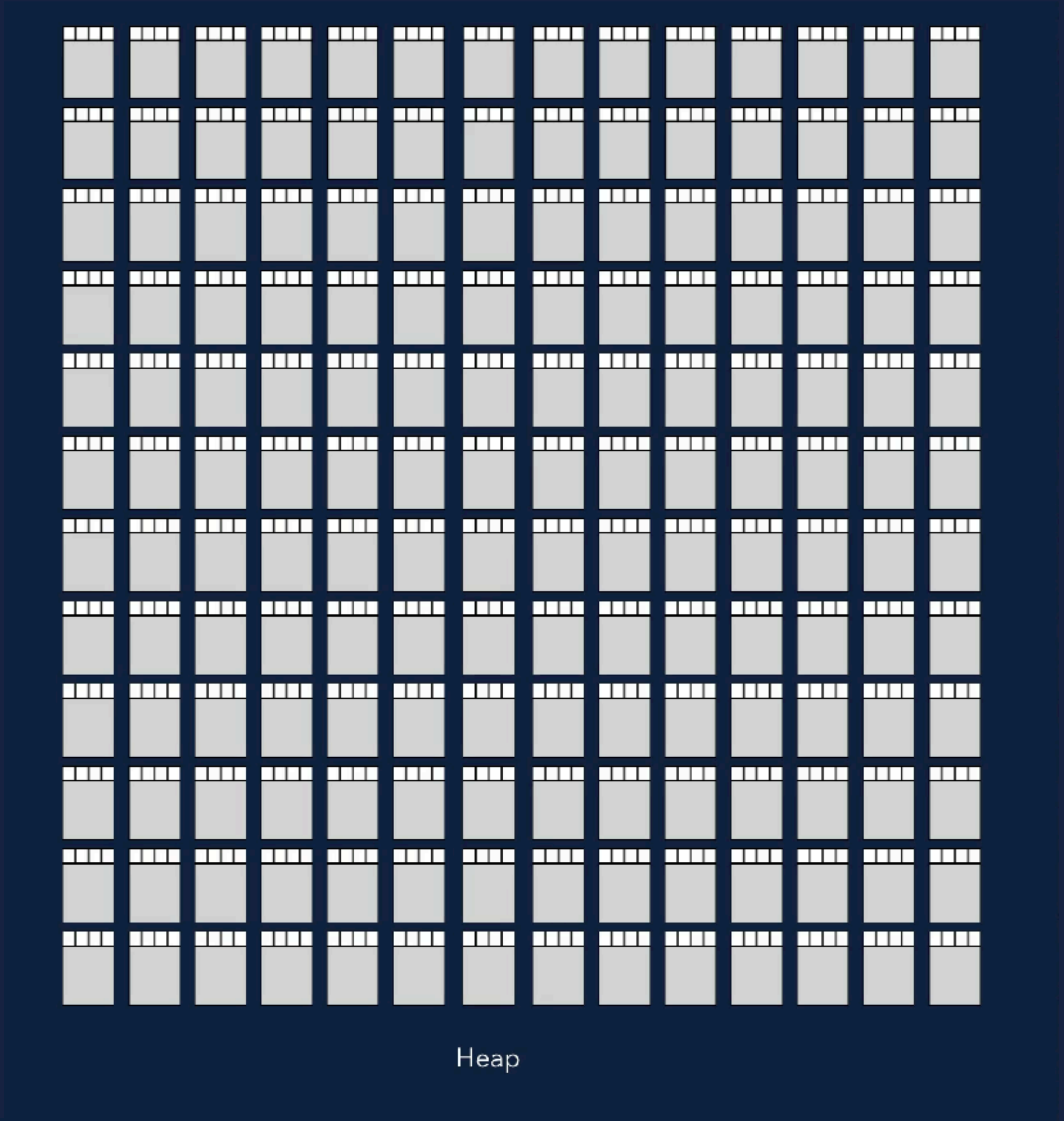
COMPACTING COLLECTOR

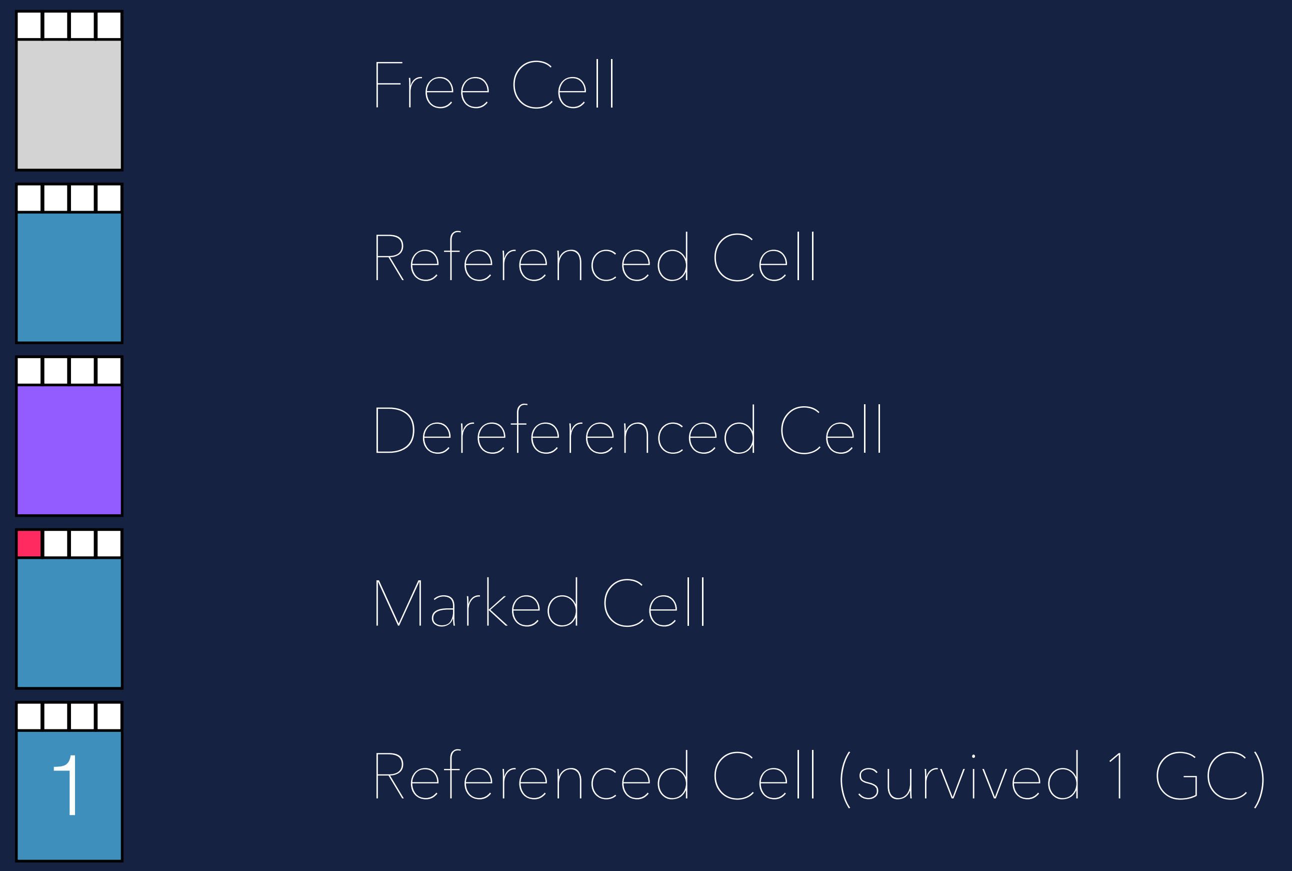
Mark & Compact

COMPACTING COLLECTOR

Demo

1. Mutator allocates cells in Heap
2. Heap is out of memory -> GC
3. Mark all live cells
4. Free all dead cells
5. Unmark all live cells
6. Compact all live cells
7. Resume Mutator





⚠ Headroom 20-50%

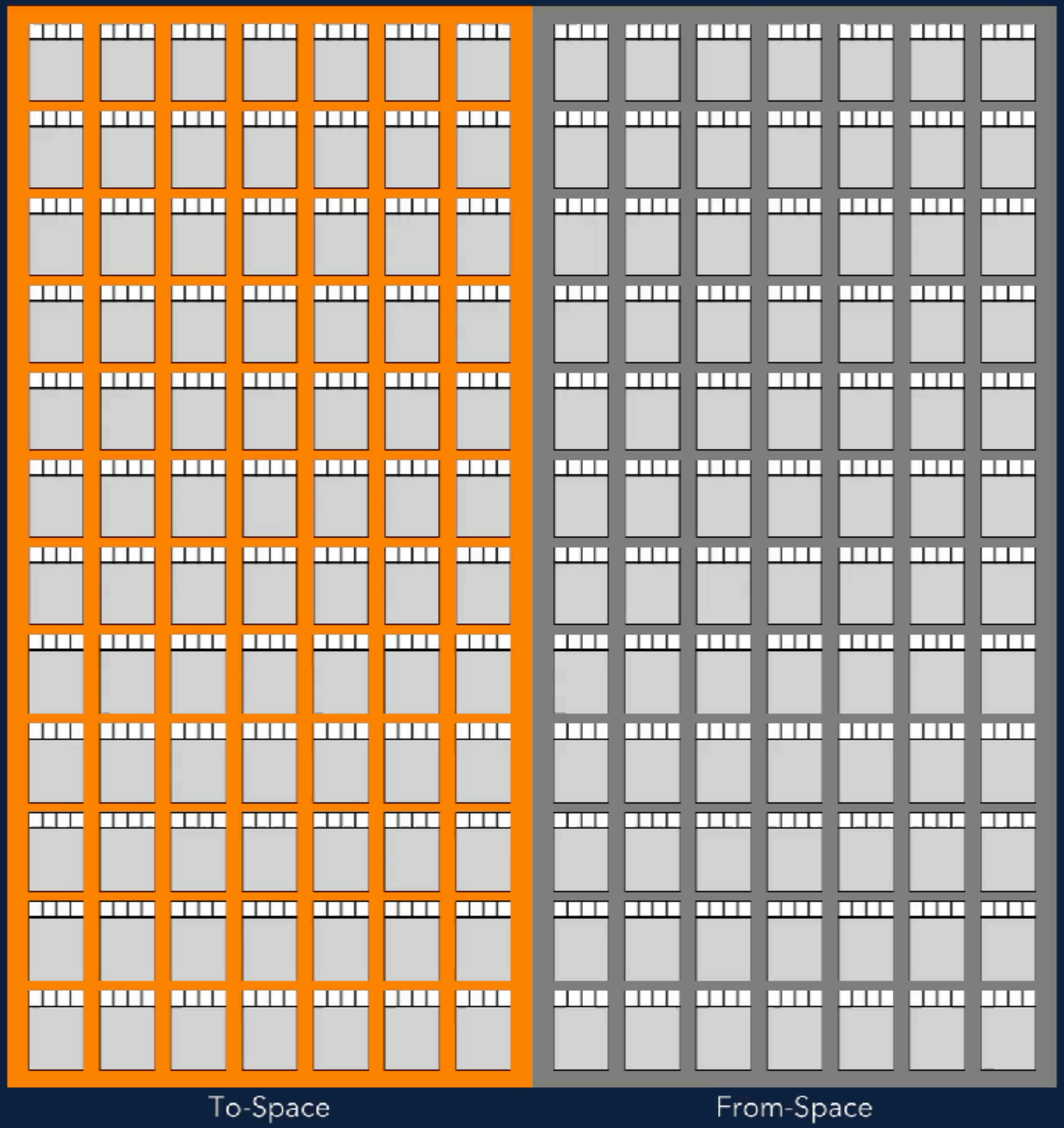
COPY COLLECTOR

Mark & Copy

COPY COLL≡CTOR

Demo

1. Allocating in ToSpace
2. ToSpace is out of memory -> GC
3. Toggle To- and FromSpace
4. Mark live cells in FromSpace
5. Copy live cells to ToSpace
6. Free all cells in FromSpace
7. Resume Mutator



Free Cell

Referenced Cell

Dereferenced Cell

Marked Cell

Referenced Cell (survived 1 GC)

🟧 To Space

⬜ From Space

⚠

Long living objects and twice as much memory

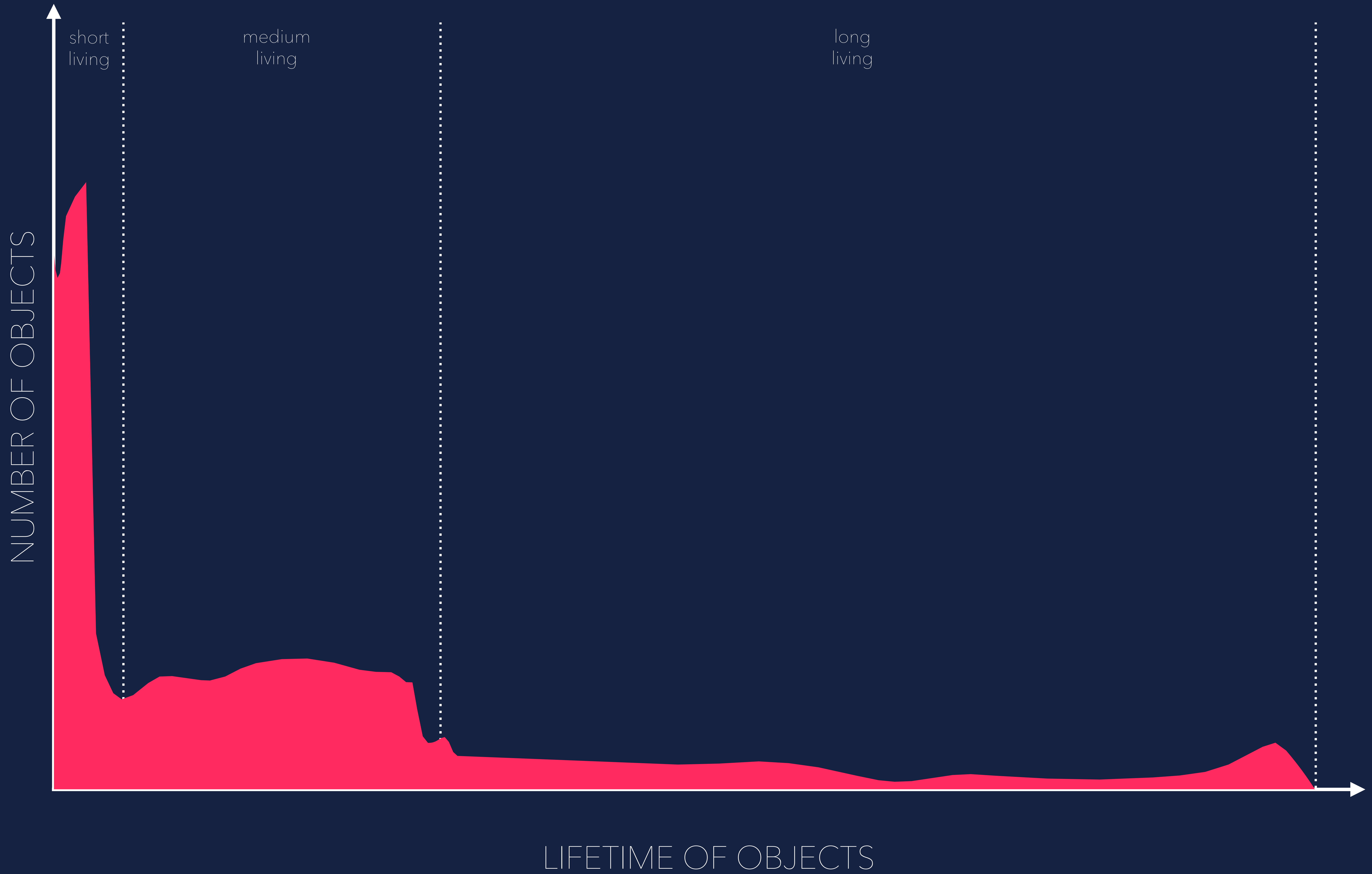
azul

GENERATIONAL COLLECTOR

Generational Mark & Compact

GENERATIONAL COLLECTOR

Weak Generational Hypothesis (Most objects die young)



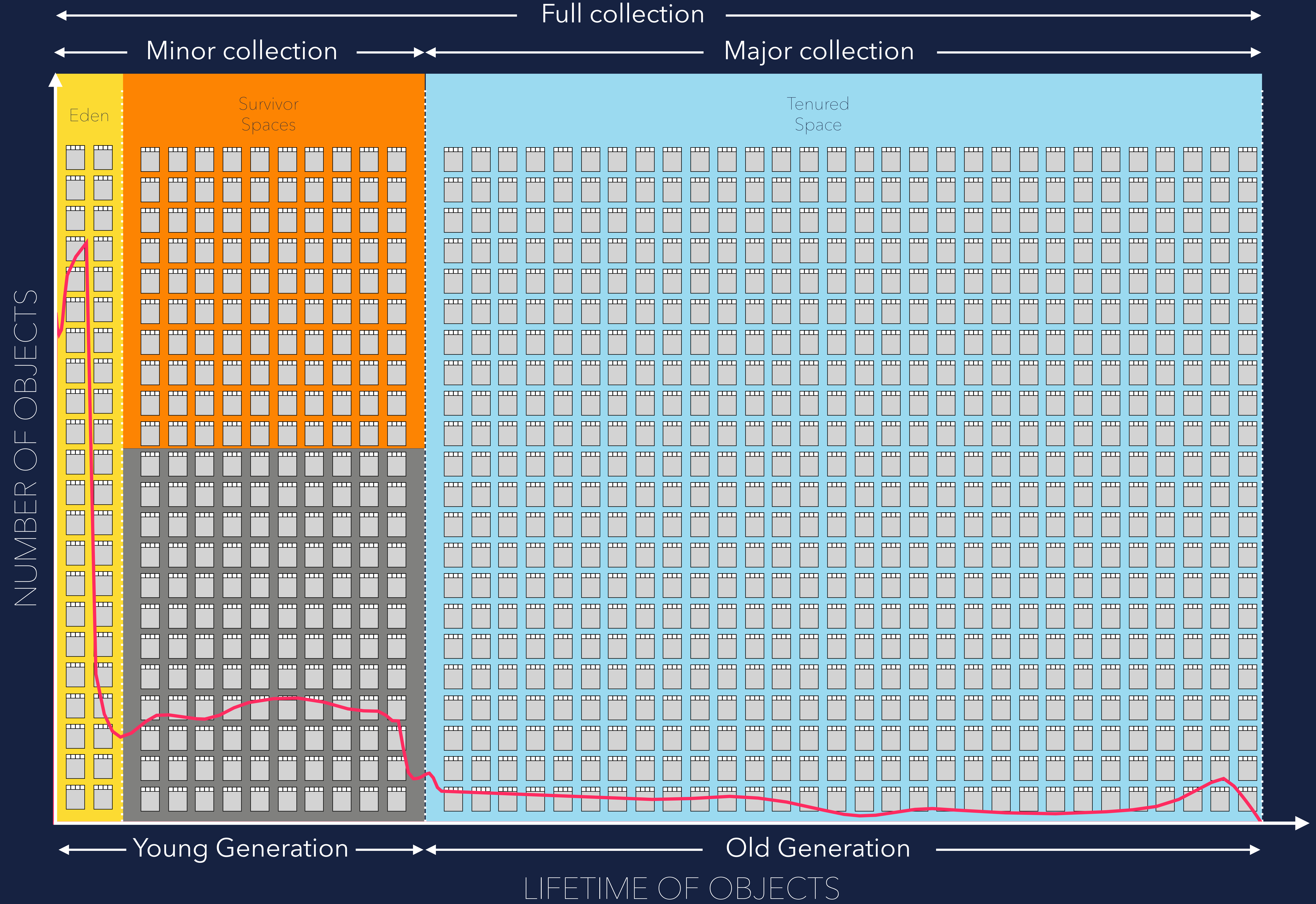
GENERATIONAL COLLECTOR

Weak Generational Hypothesis (Most objects die young)

Eden space for short living objects

Survivor spaces for medium living objects

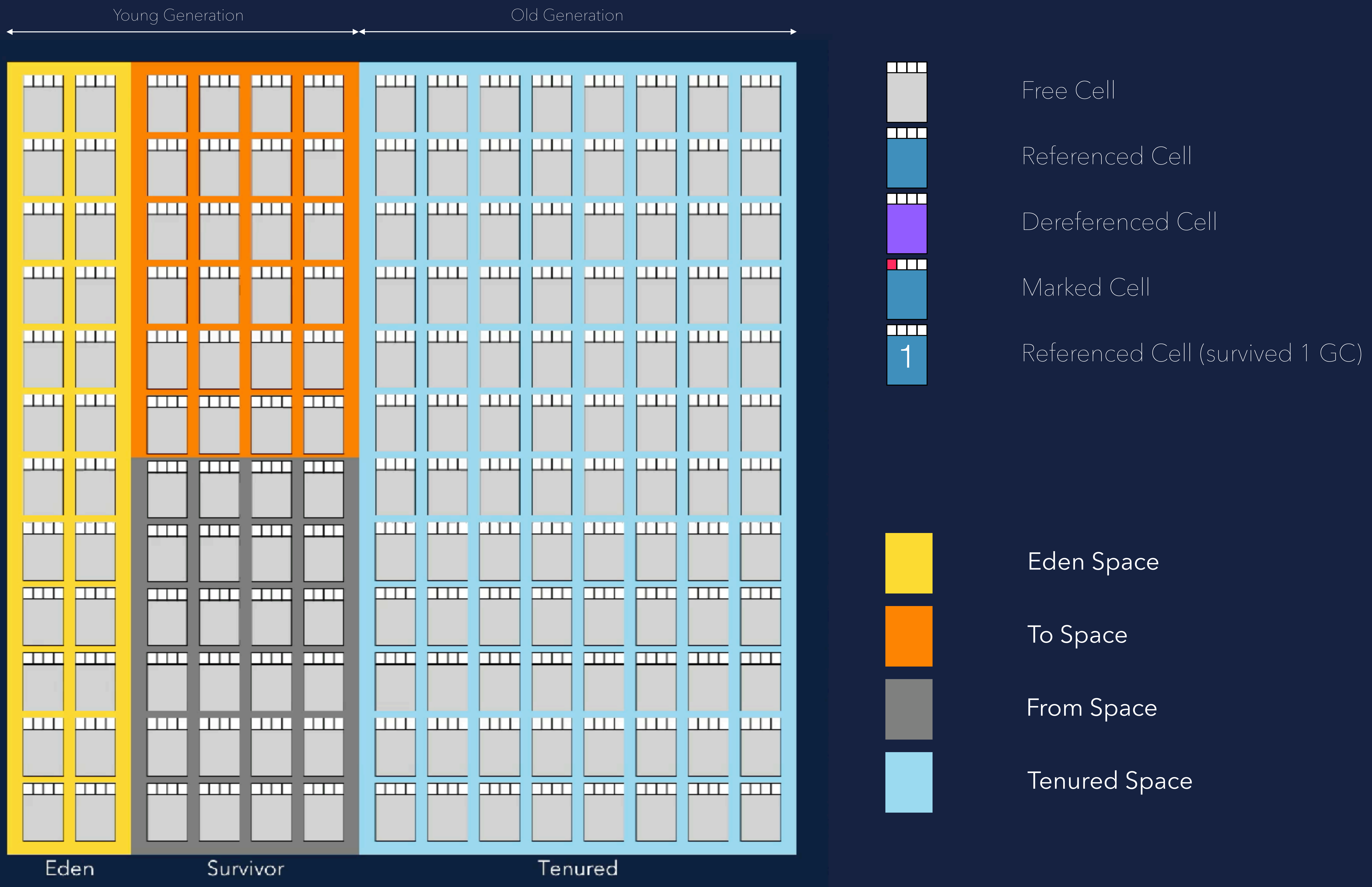
Tenured space for long living objects



GENERATIONAL COLLECTOR

Demo

1. Mutator allocates cells in Eden
2. Eden is out of memory -> GC
3. Toggle To- and FromSpace
4. Copy all live cells from FromSpace to ToSpace
5. Copy all live cells from Eden to ToSpace
6. Promote live cells from FromSpace to TenuredSpace
7. Free all dead cells
8. Resume Mutator

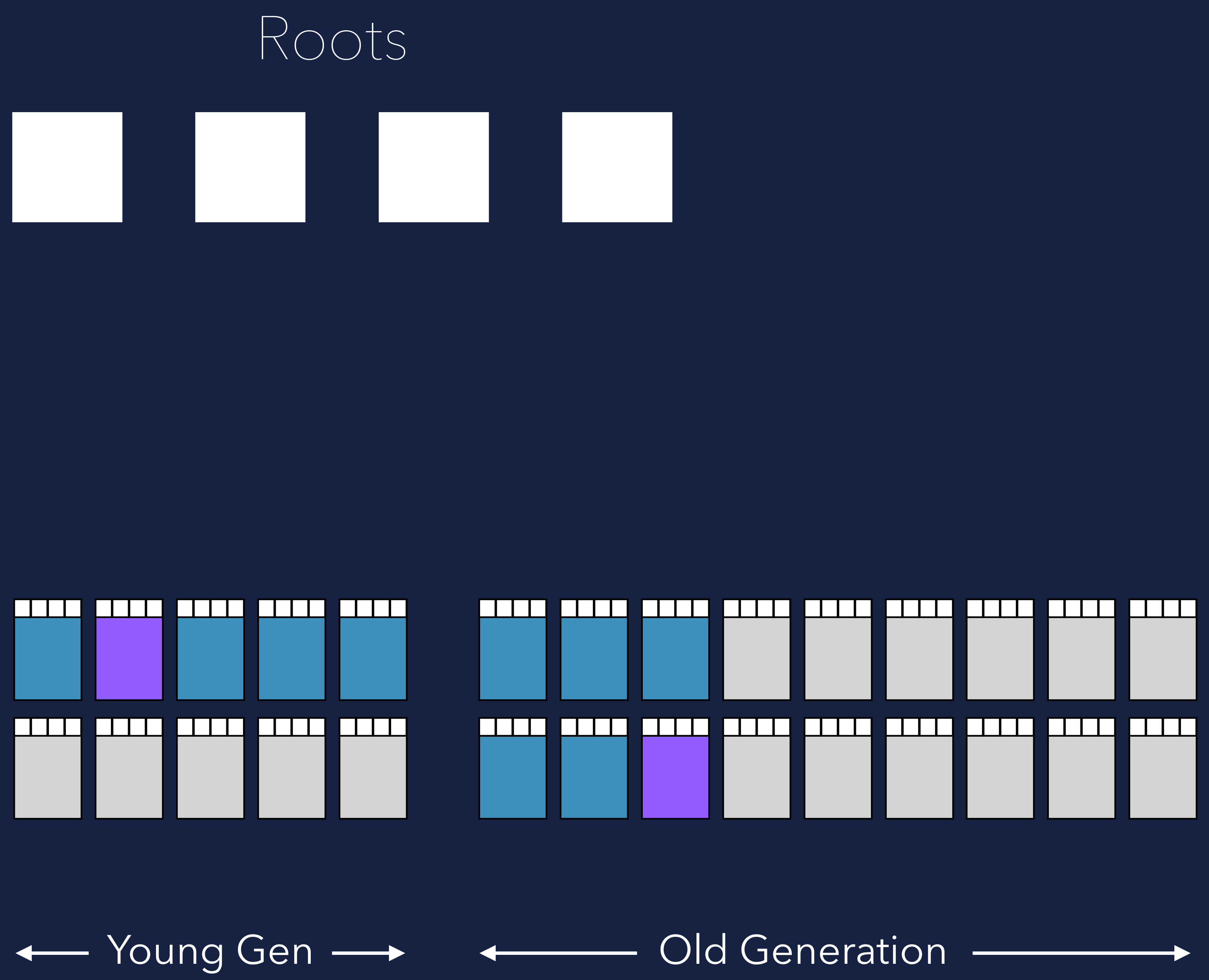


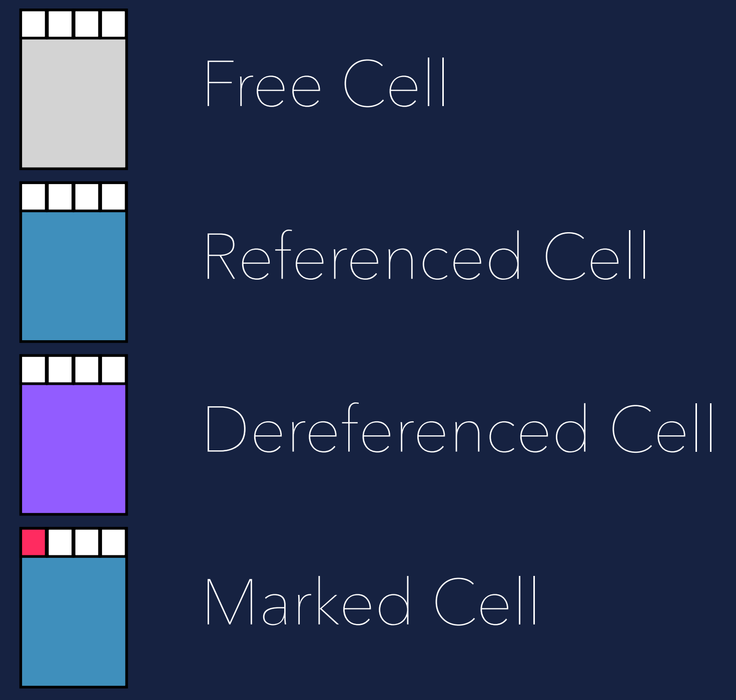
GENERATIONAL COLLECTOR

How to do a minor collection with references from old to young generation...?

GENERATIONAL COLLECTOR

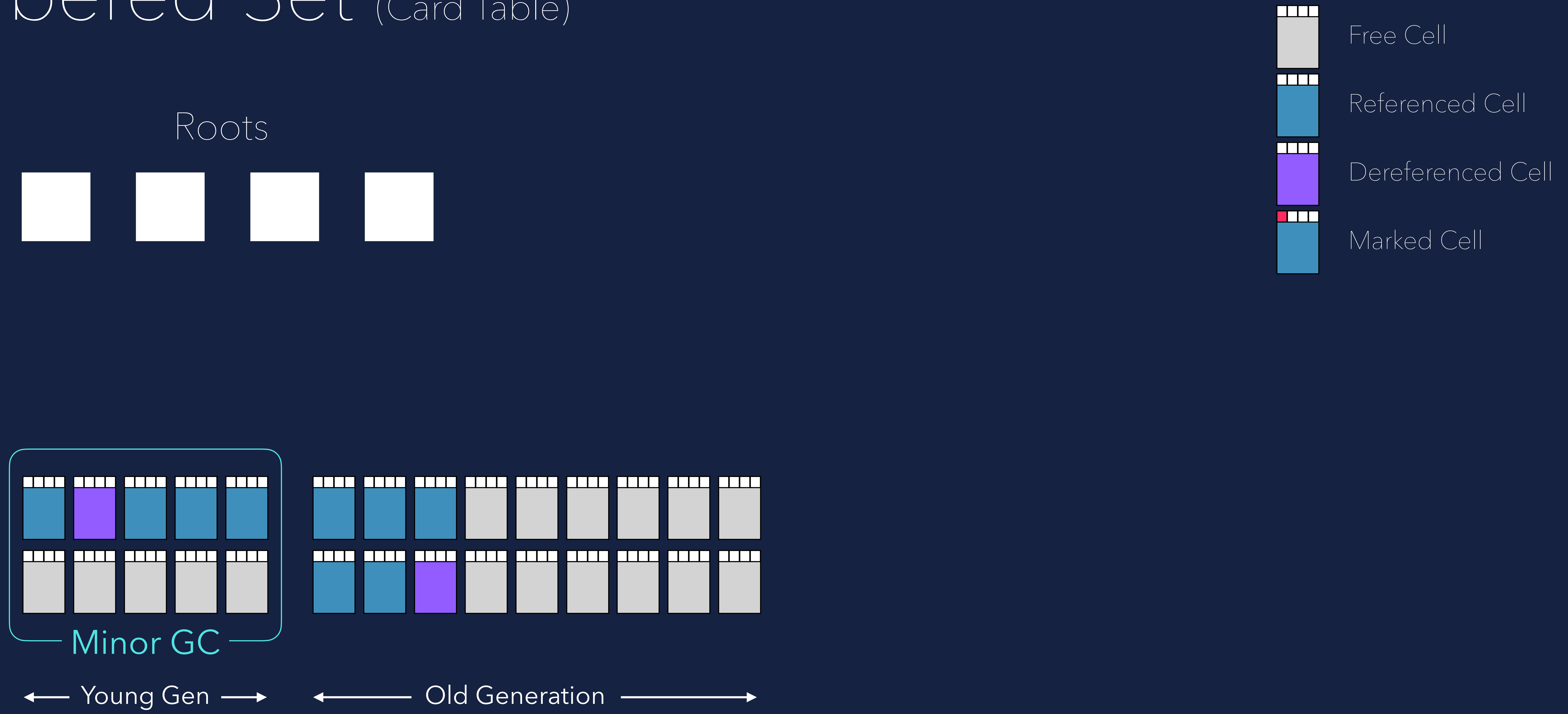
Remembered Set (Card Table)





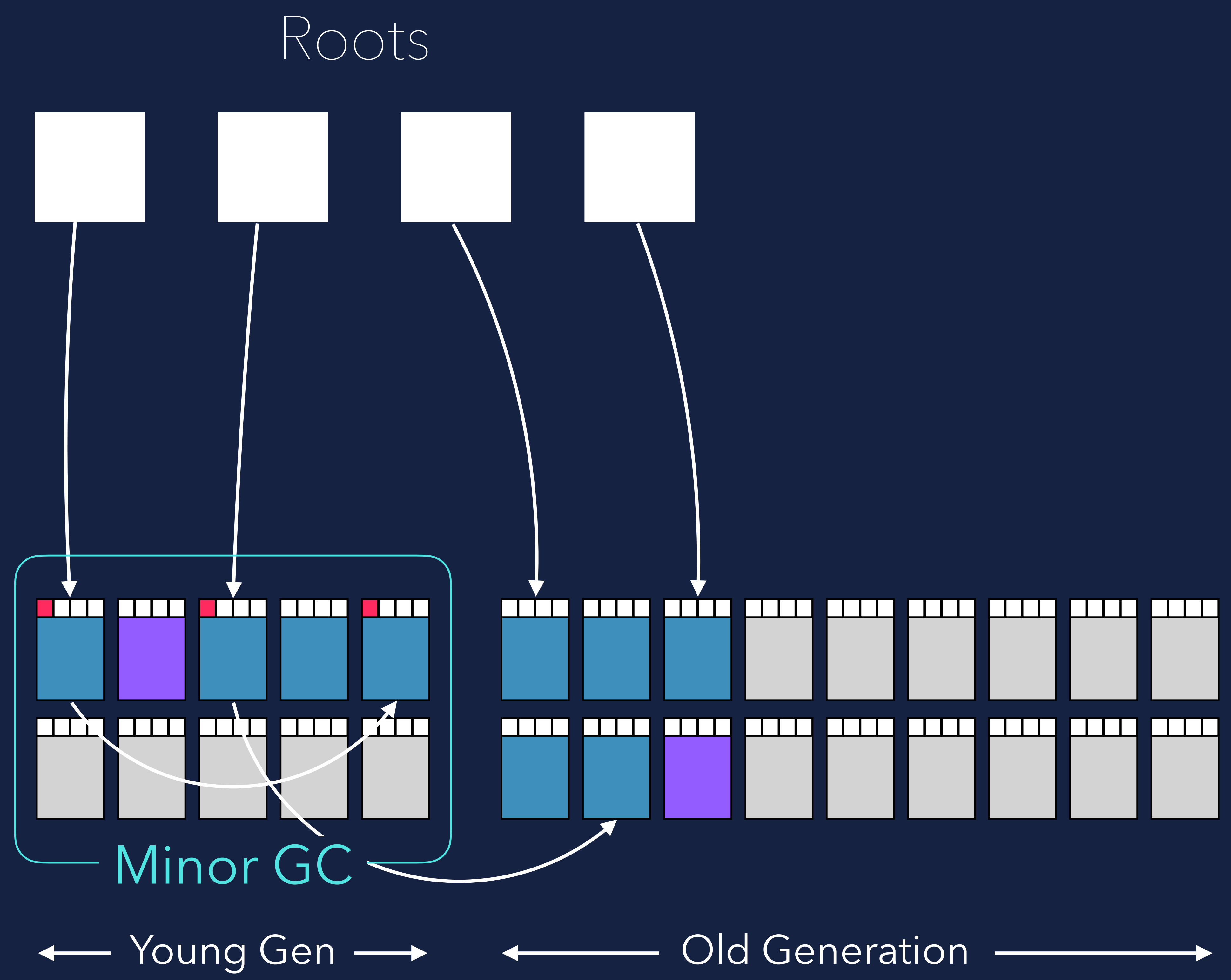
GENERATIONAL COLLECTOR

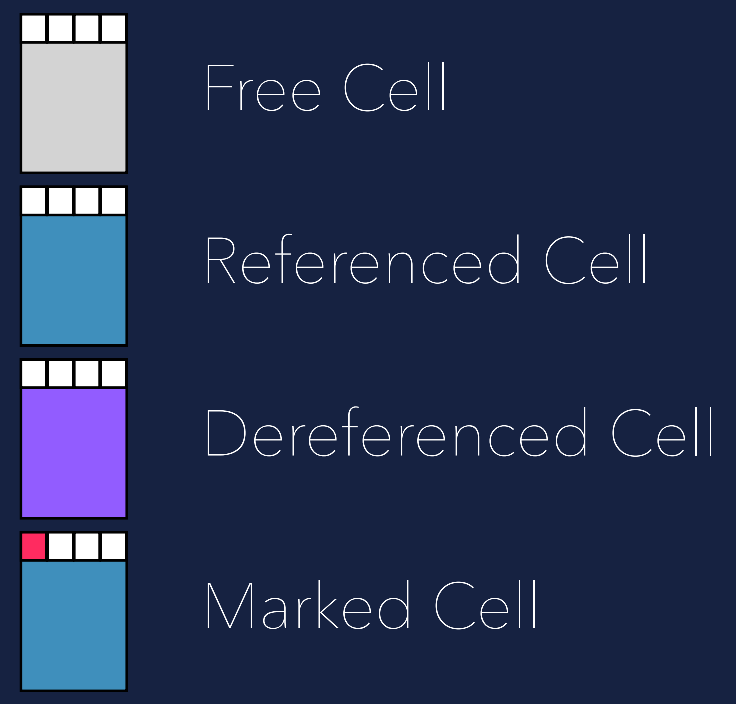
Remembered Set (Card Table)



GENERATIONAL COLLECTOR

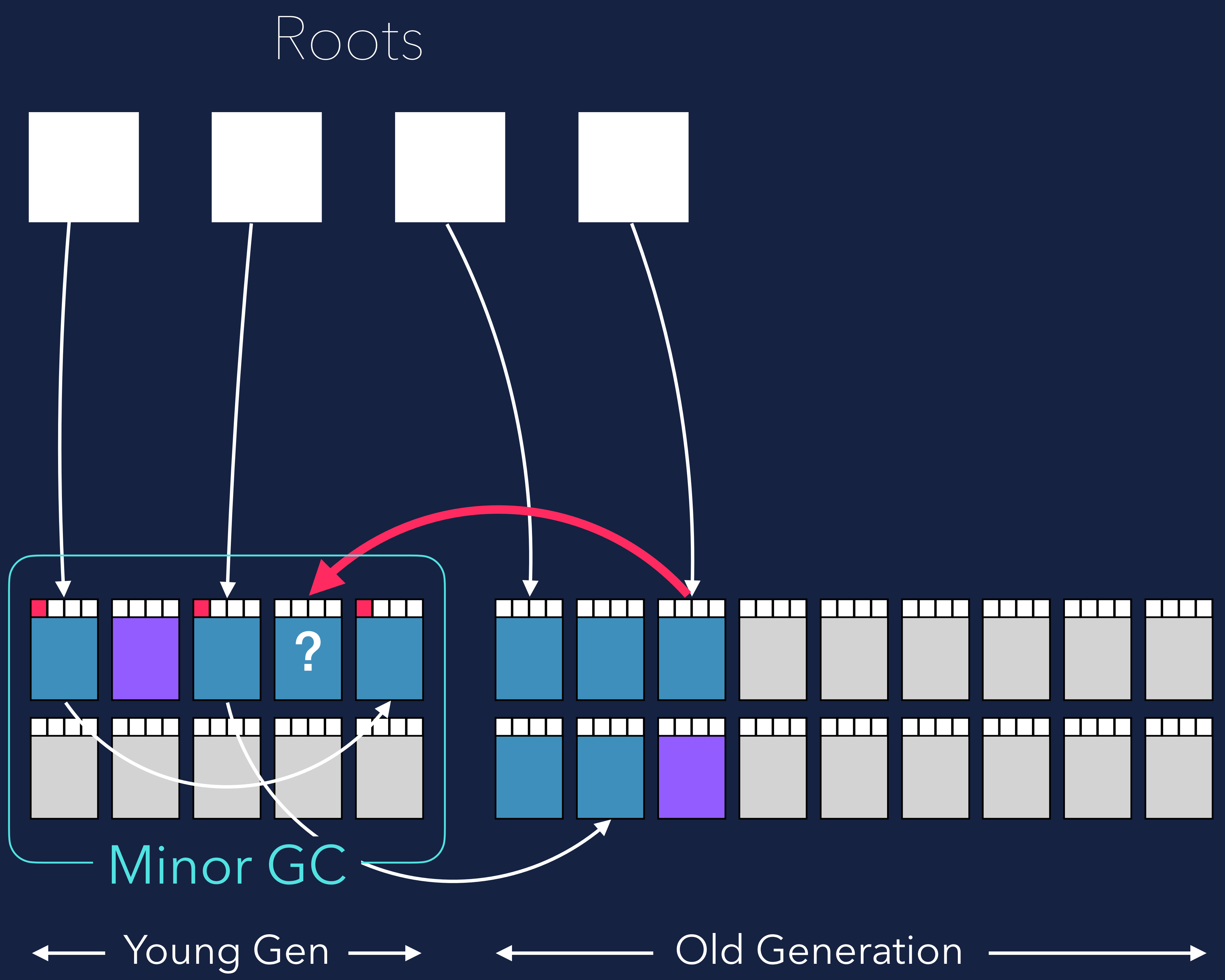
Remembered Set (Card Table)

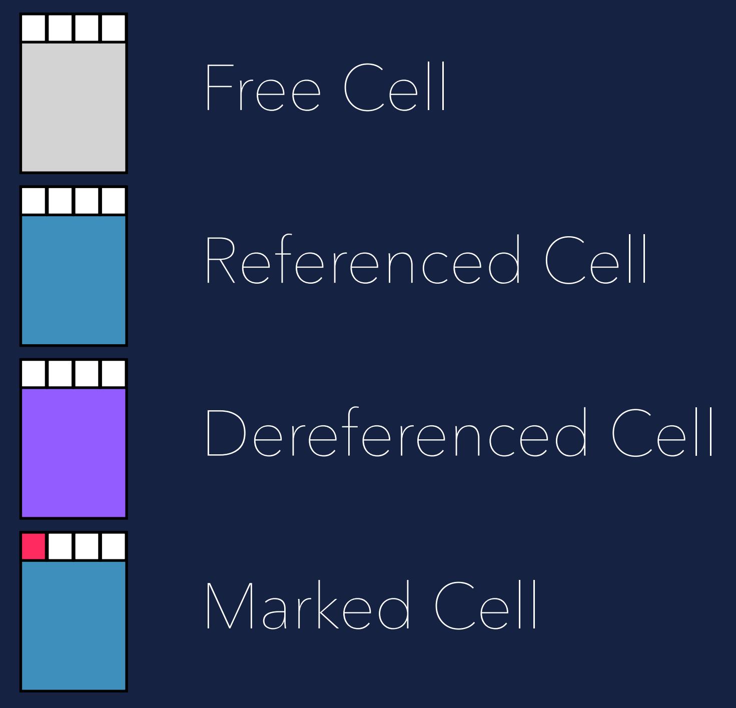




GENERATIONAL COLLECTOR

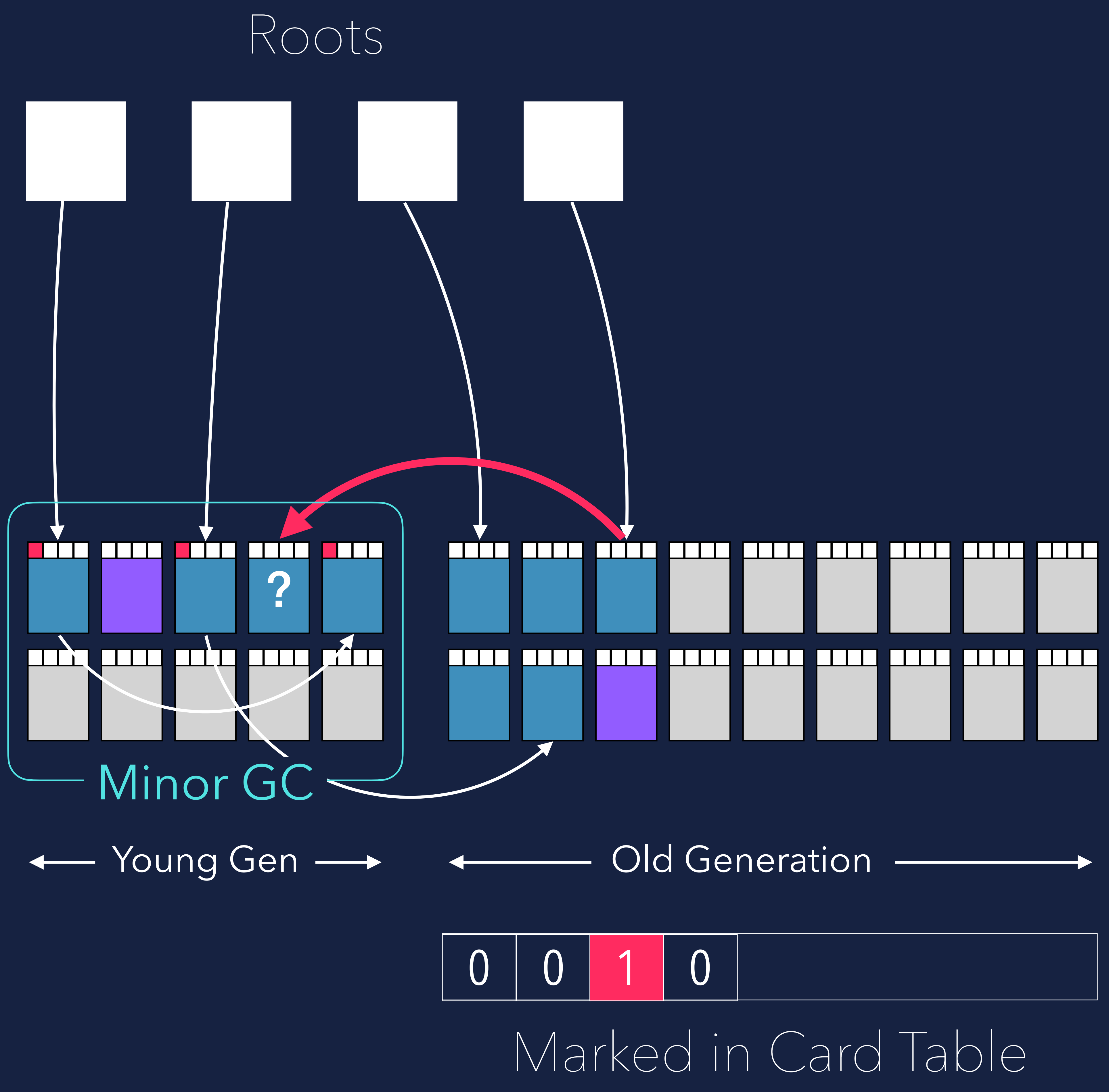
Remembered Set (Card Table)

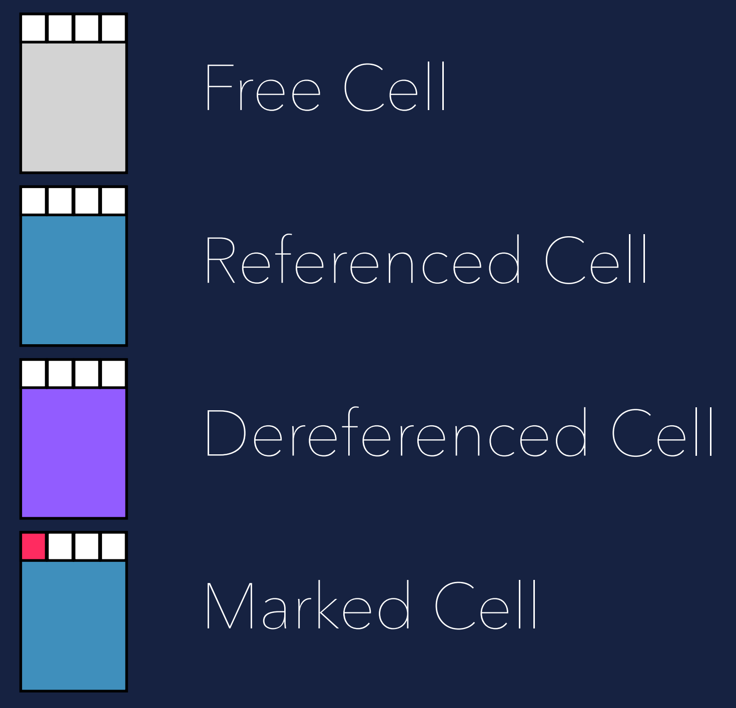




GENERATIONAL COLLECTOR

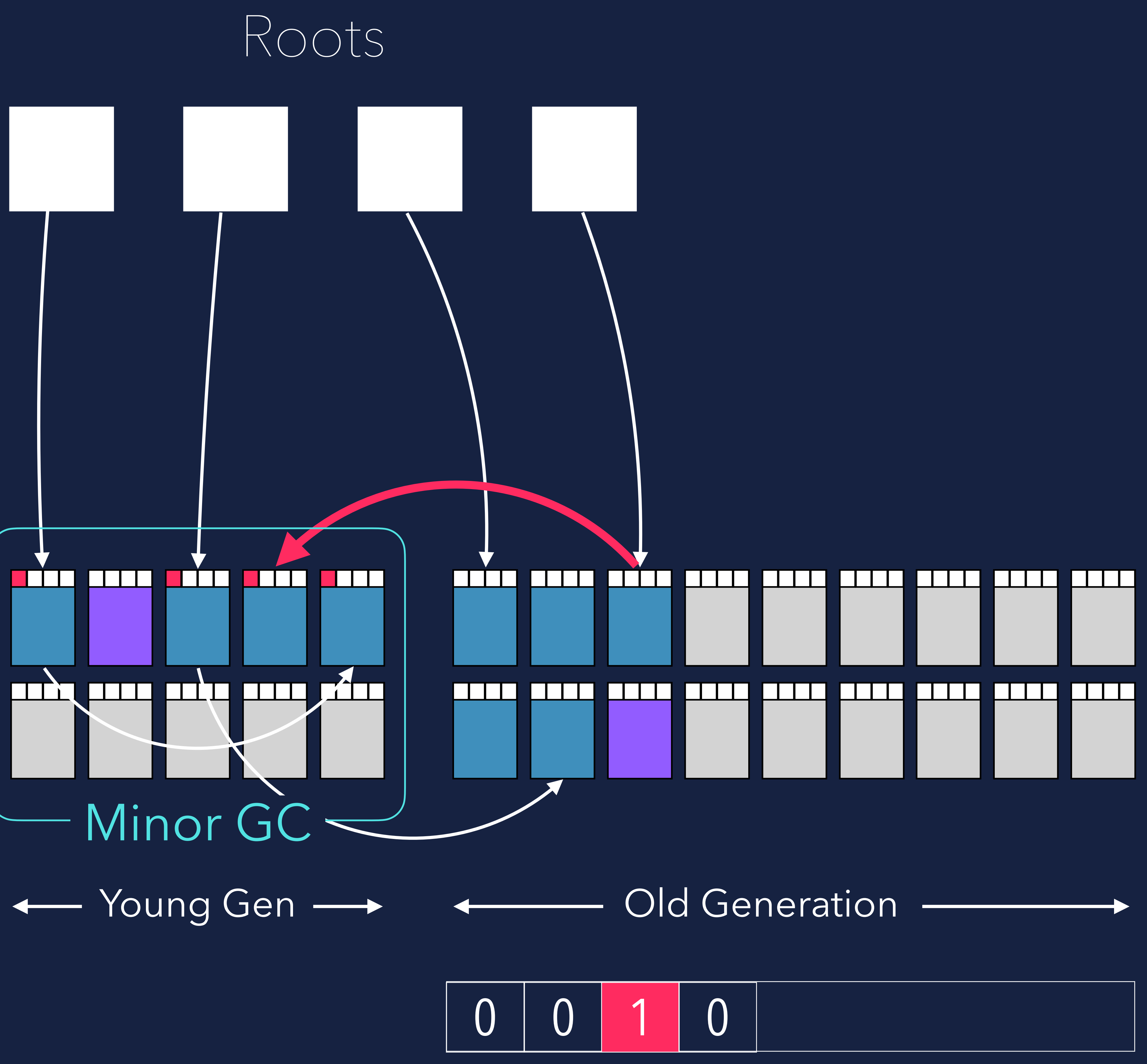
Remembered Set (Card Table)





GENERATIONAL COLLECTOR

Remembered Set (Card Table)



0	0	1	0	

Marked in Card Table

- ⬜ Free Cell
- 🟦 Referenced Cell
- 🟪 Dereferenced Cell
- 🟥 Marked Cell

GC looks up Card Table, finds the reference and marks it as live

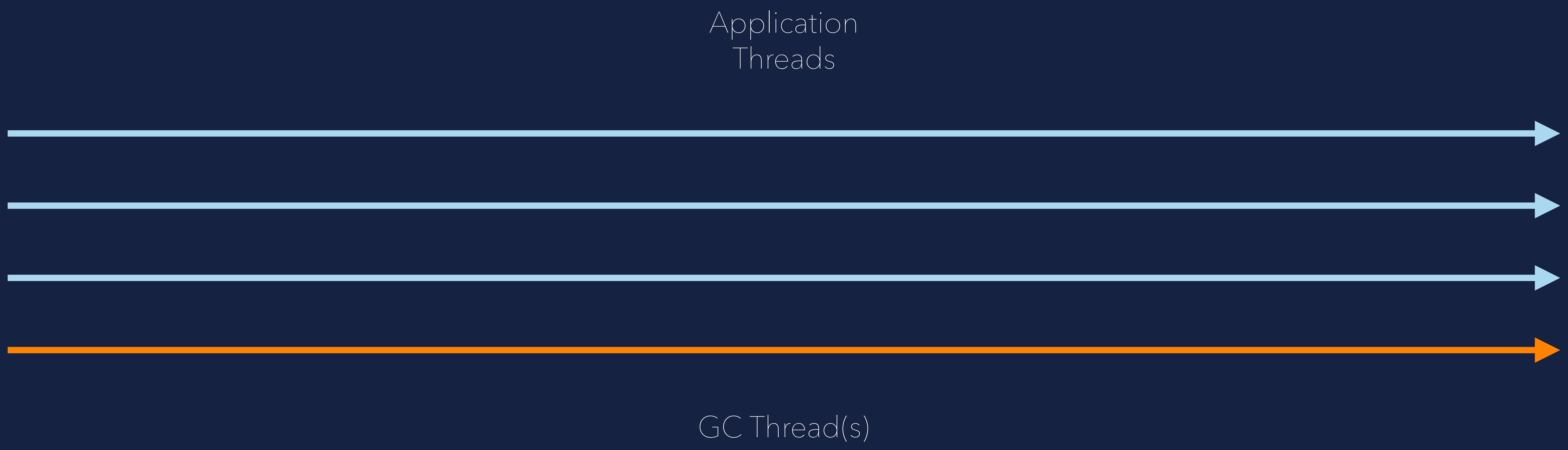
CONCURRENT COLLECTION ?

CONCURRENCY IS HARD...

STOP THE W🛑RLD

CONCURRENT COLLECTION

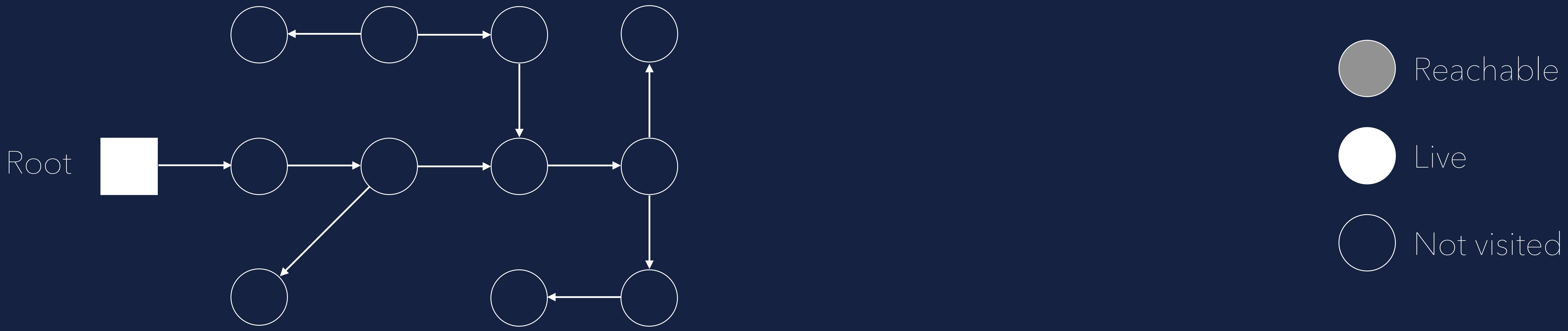
Application and GC running concurrently



CONCURRENT MARKING

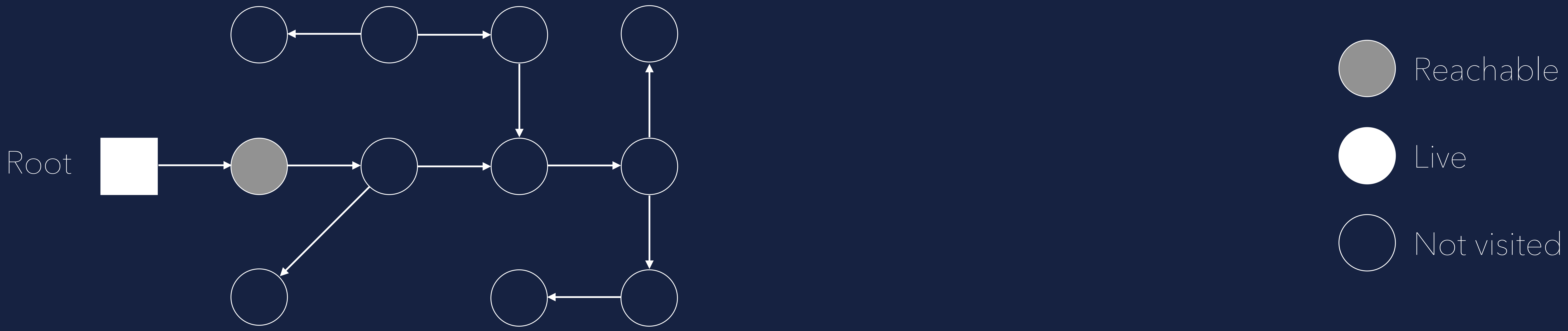
CONCURRENCY IS HARD...

Concurrent Marking



CONCURRENCY IS HARD...

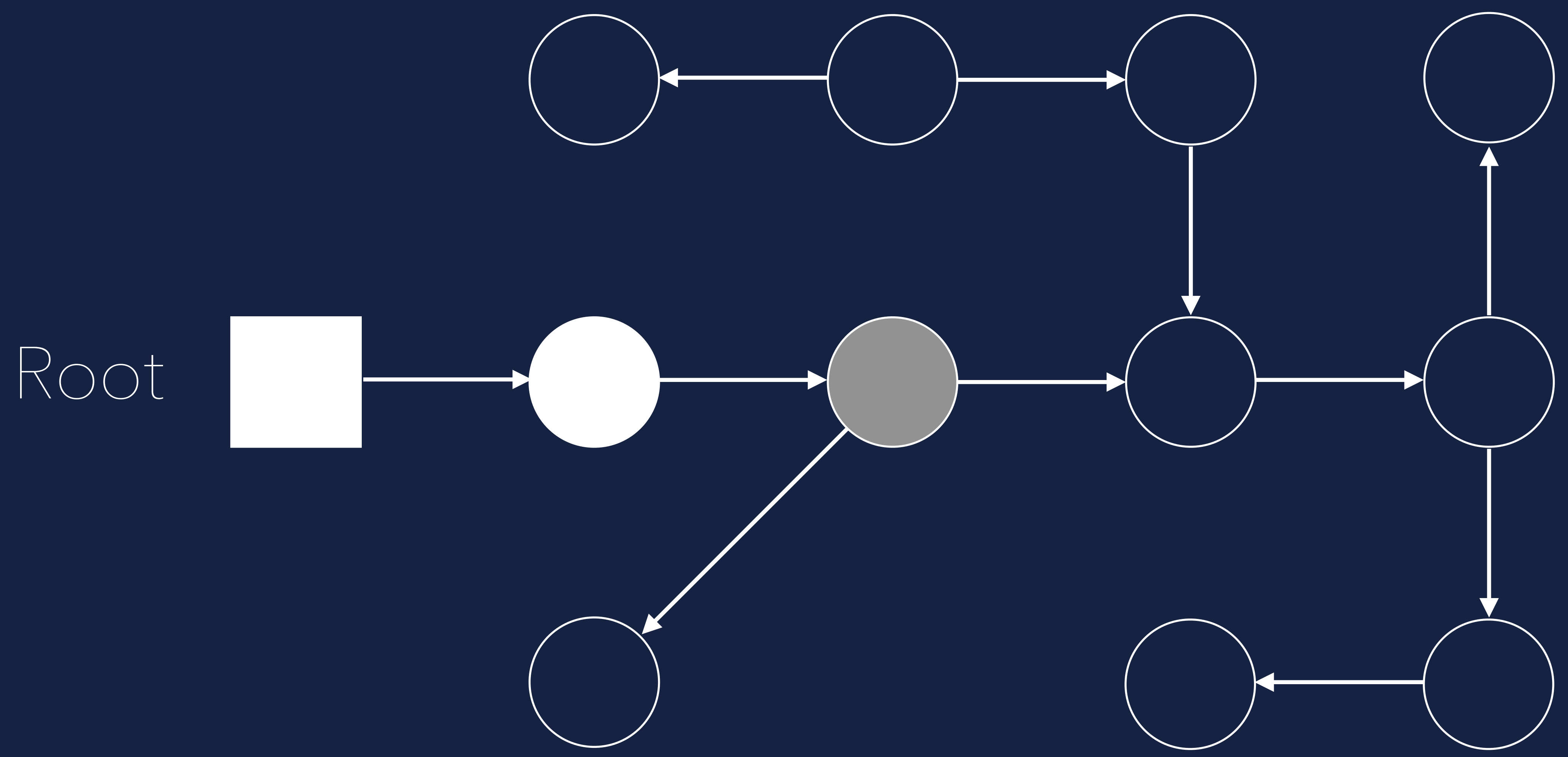
Concurrent Marking



Collector starts marking objects

CONCURRENCY IS HARD...

Concurrent Marking



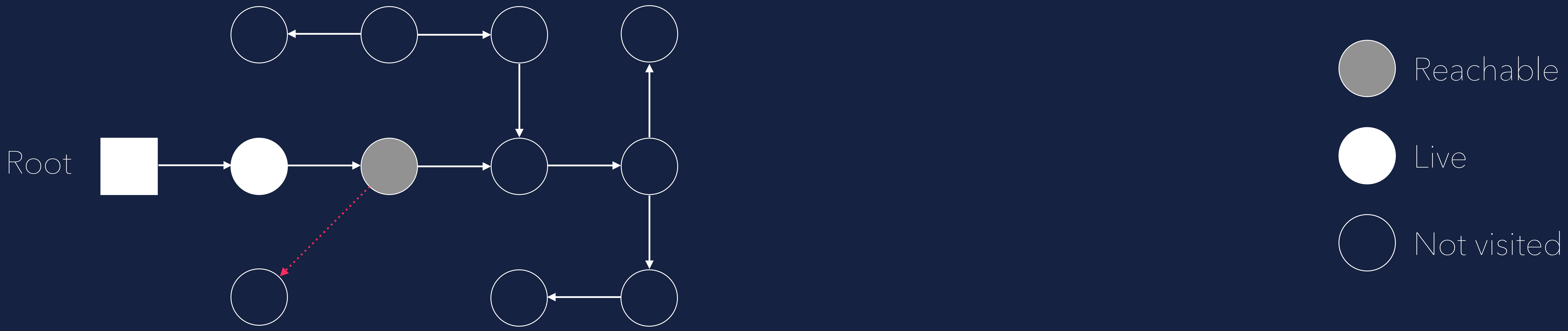
● Reachable

○ Live

○ Not visited

CONCURRENCY IS HARD...

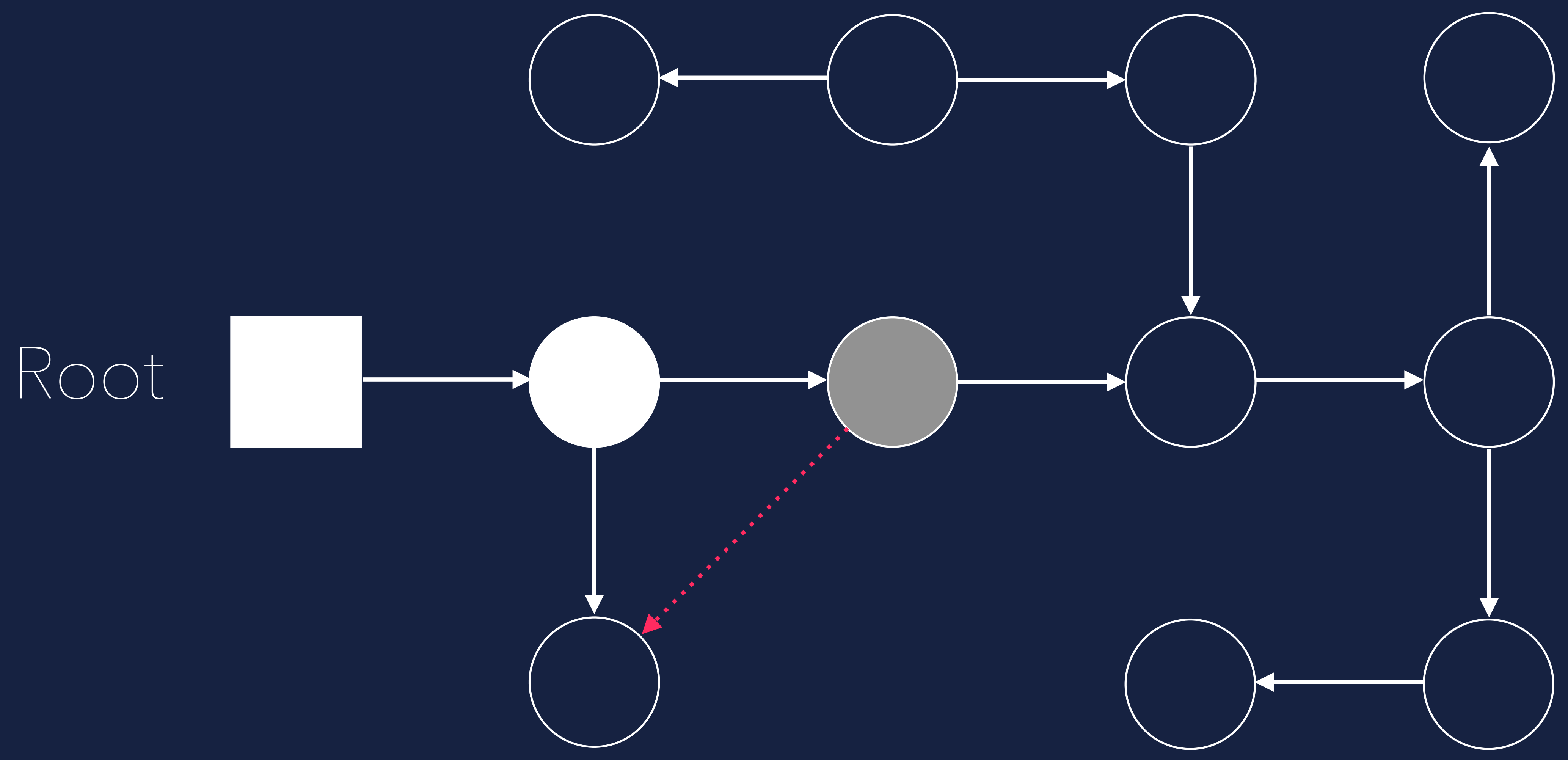
Concurrent Marking



Mutator removes reference and creates a new one from an already visited cell !

CONCURRENCY IS HARD...

Concurrent Marking



● Reachable

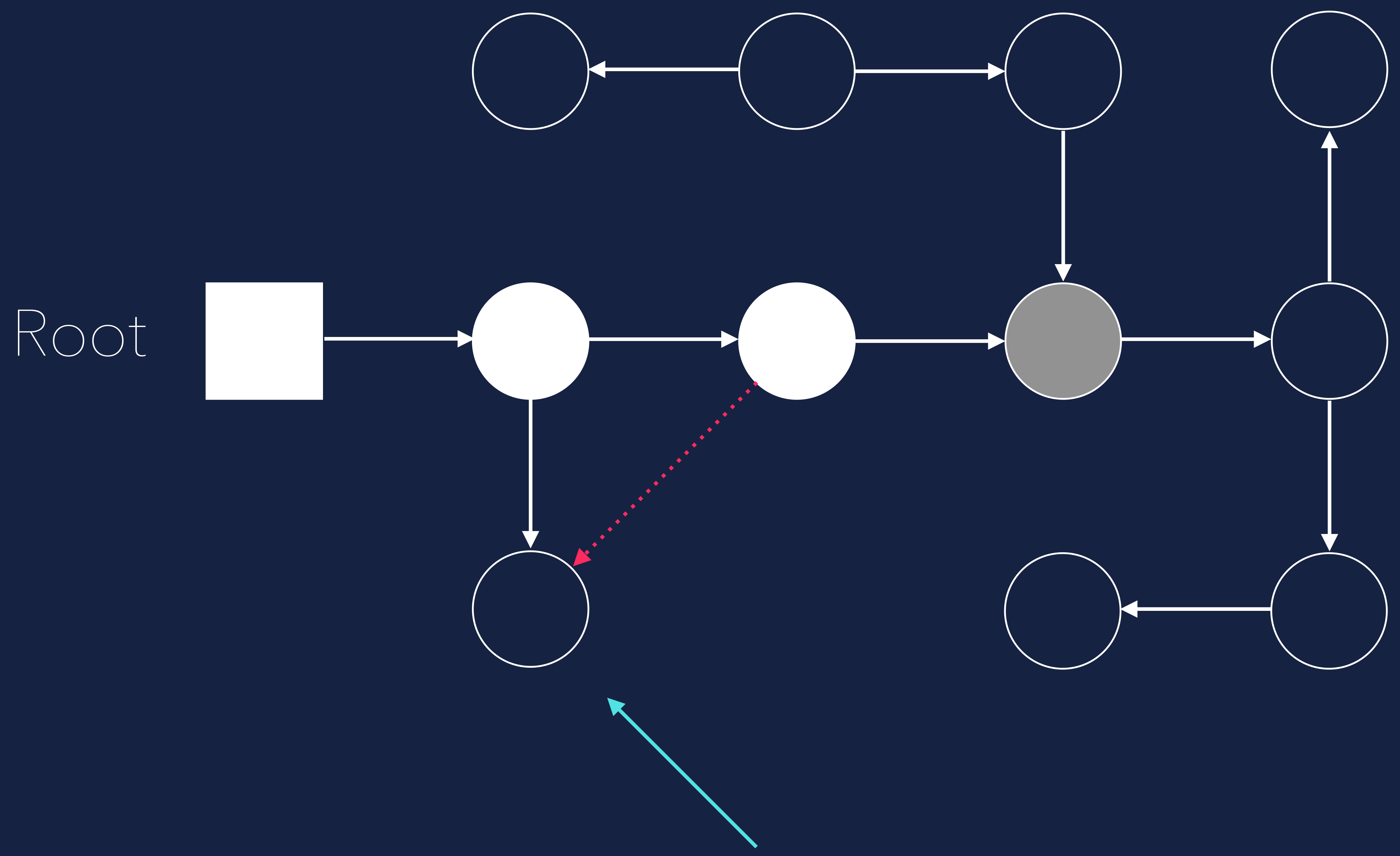
○ Live

○ Not visited

Mutator removes reference and creates a new one from an already visited cell !

CONCURRENCY IS HARD...

Concurrent Marking



⚠

● Reachable

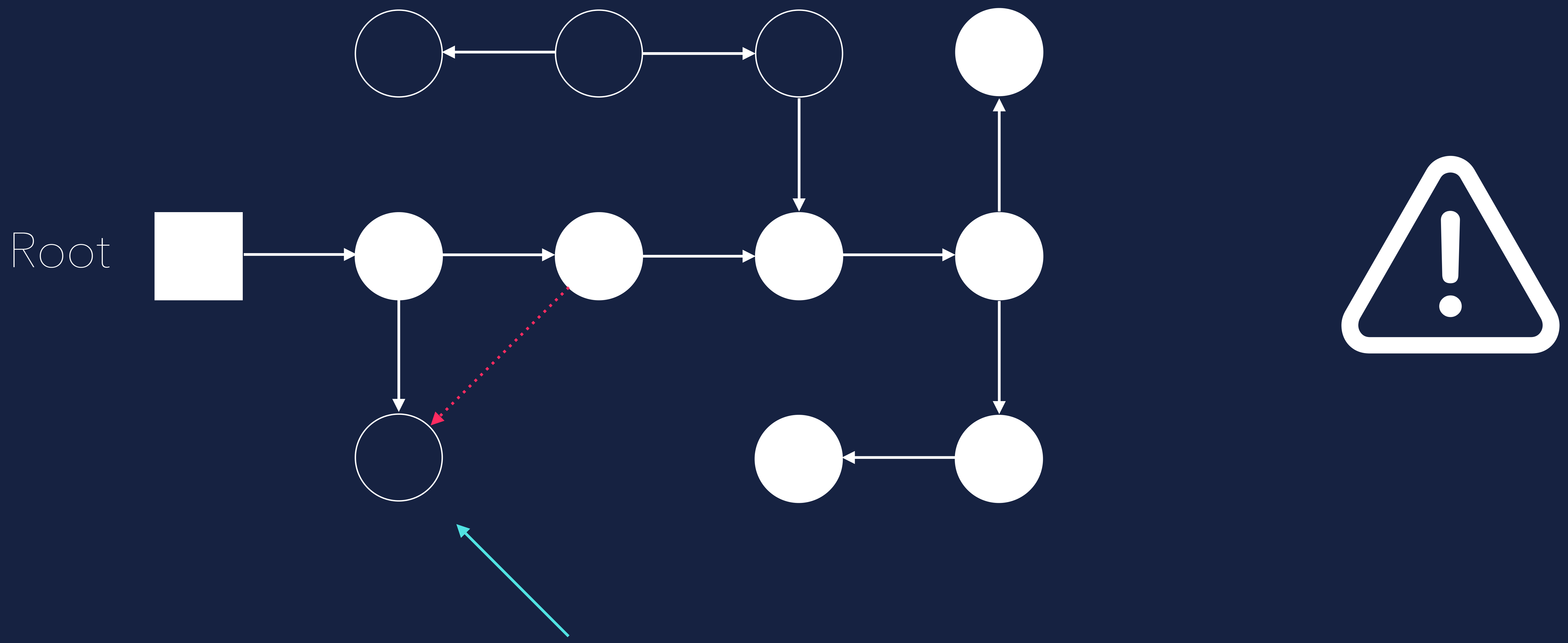
○ Live

◯ Not visited

Won't be detected by the Garbage Collector !

CONCURRENCY IS HARD...

Concurrent Marking



- ● Reachable
- ● Live
- ○ Not visited

Won't be detected by the Garbage Collector !

???

BARRIERS TO THE RESCUE

BARRIERS

Read / Write Barriers

- Mechanisms to execute memory management code when a read/write on some object takes place

BARRIERS

Read / Write Barriers

- Mechanisms to execute memory management code when a read/write on some object takes place

- Used to keep track of inter-generational references.
  (references from old generation to young generation, the so called Rembered Set)

BARRIERS

Read / Write Barriers

- Mechanisms to execute memory management code when a read/write on some object takes place

- Used to keep track of inter-generational references.
  (references from old generation to young generation, the so called Rembered Set)

- Used to synchronize action between mutator and collector
  (allocation concurrent to collection)

BARRIERS

Read / Write Barriers

- Mechanisms to execute memory management code when a read/write on some object takes place

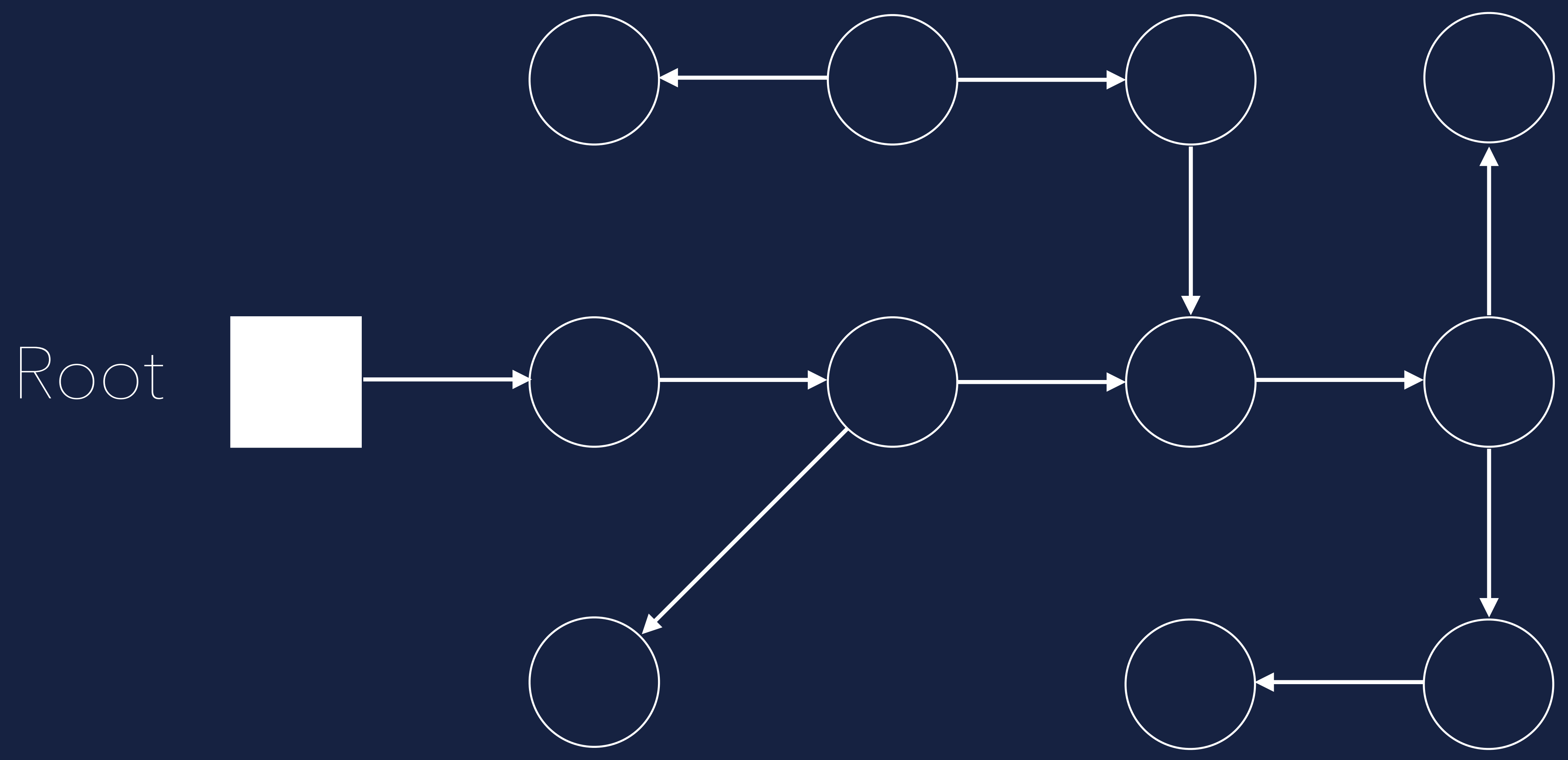
- Used to keep track of inter-generational references.
  (references from old generation to young generation, the so called Rembered Set)

- Used to synchronize action between mutator and collector
  (allocation concurrent to collection)

- Read Barriers are usually more expensive
  (reads 75% to writes 25% -> Read Barriers must very efficient)

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



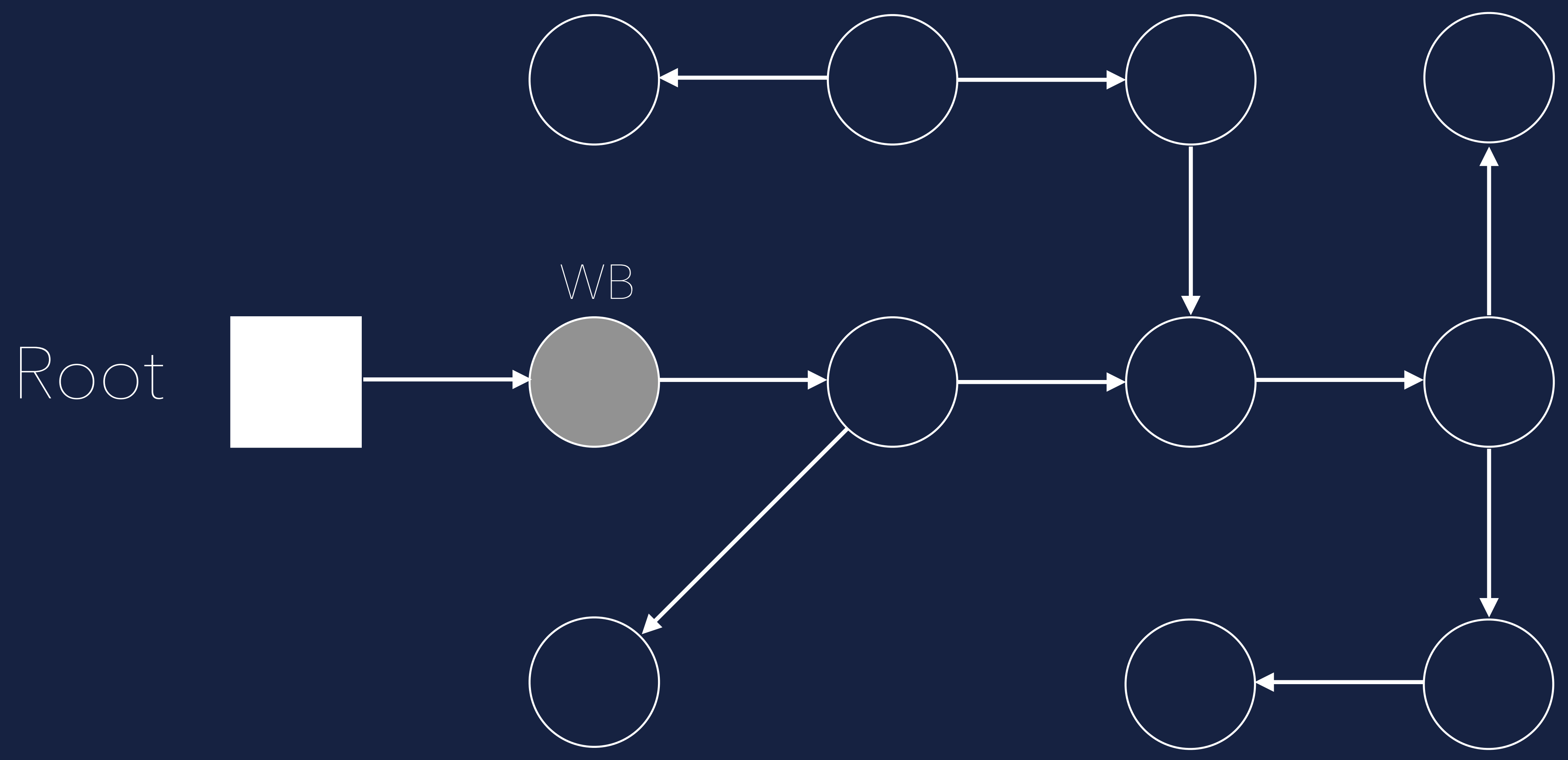
● Reachable

● Live

○ Not visited

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers

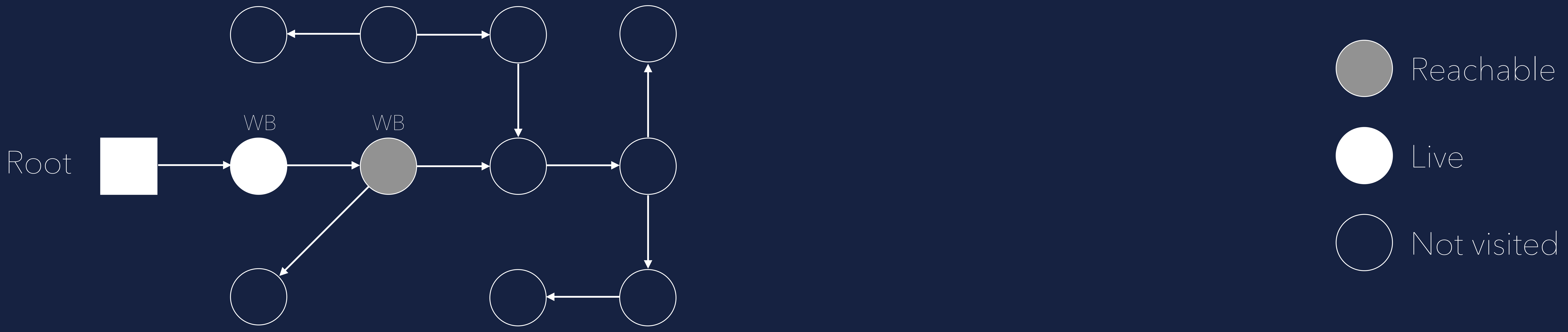


- ● Reachable
- ● Live
- ○ Not visited

Collector starts marking objects

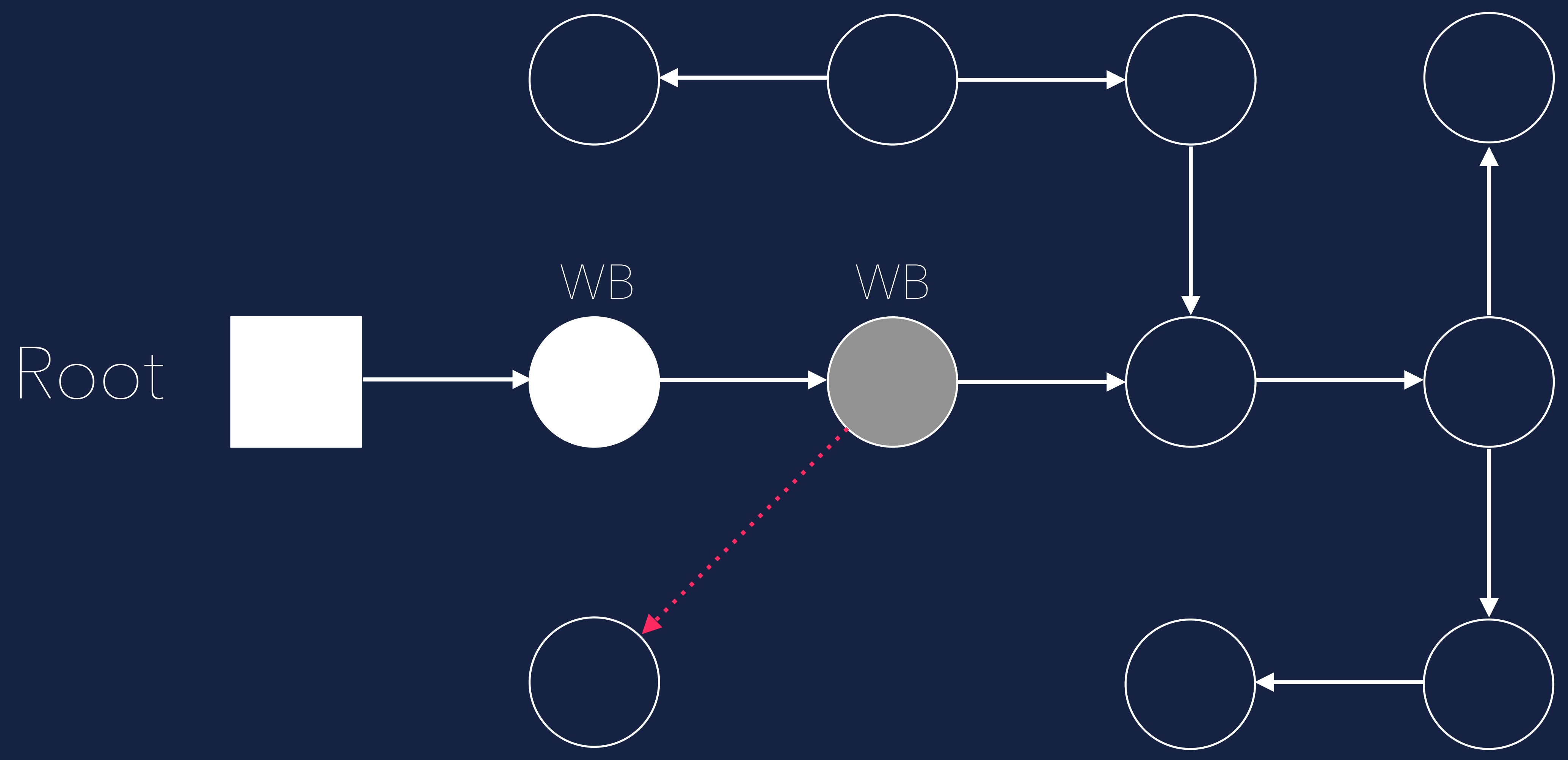
CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



● Reachable

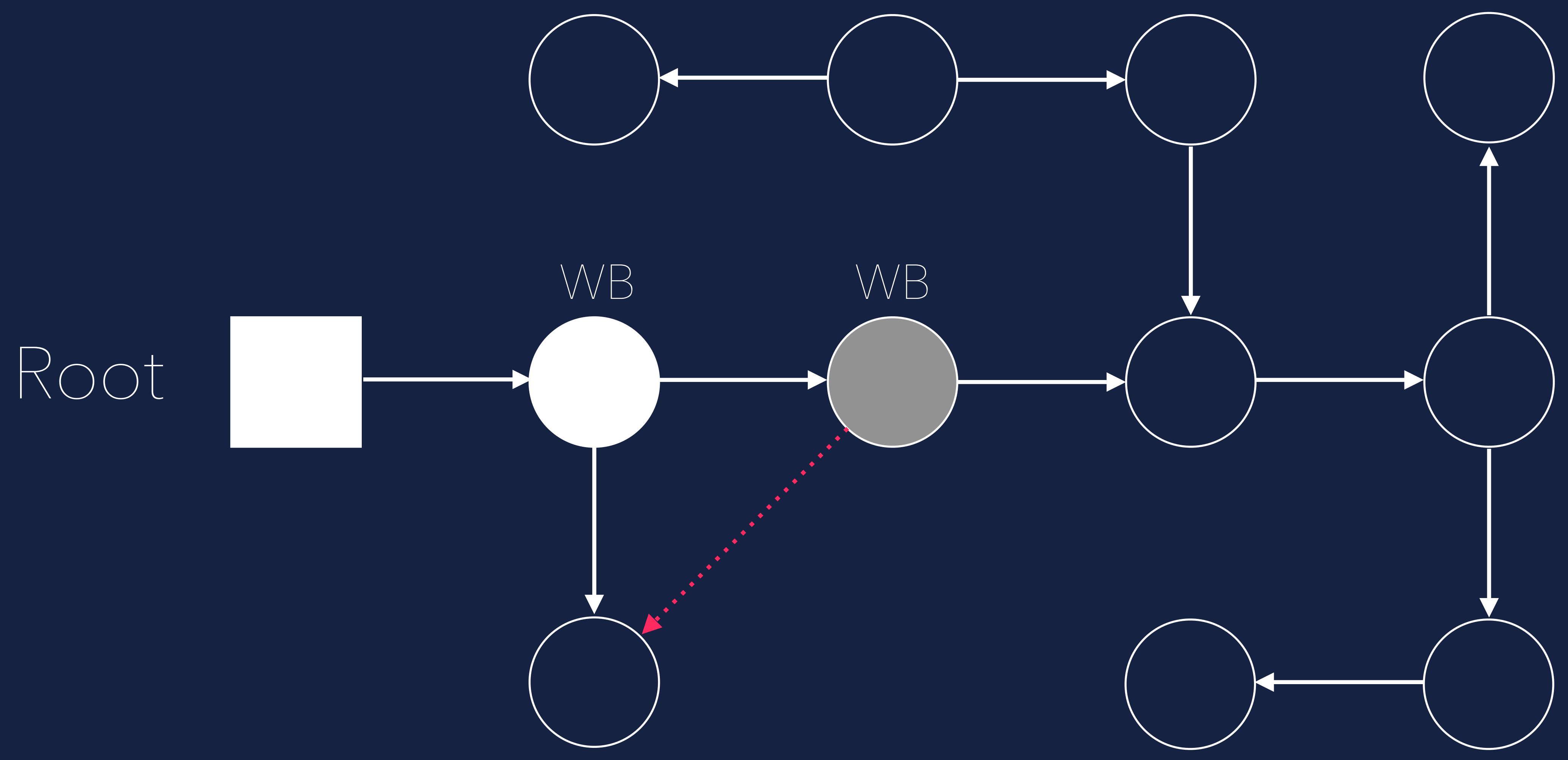
● Live

○ Not visited

Mutator hits write barrier and removes reference and adds a new one

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



● Reachable

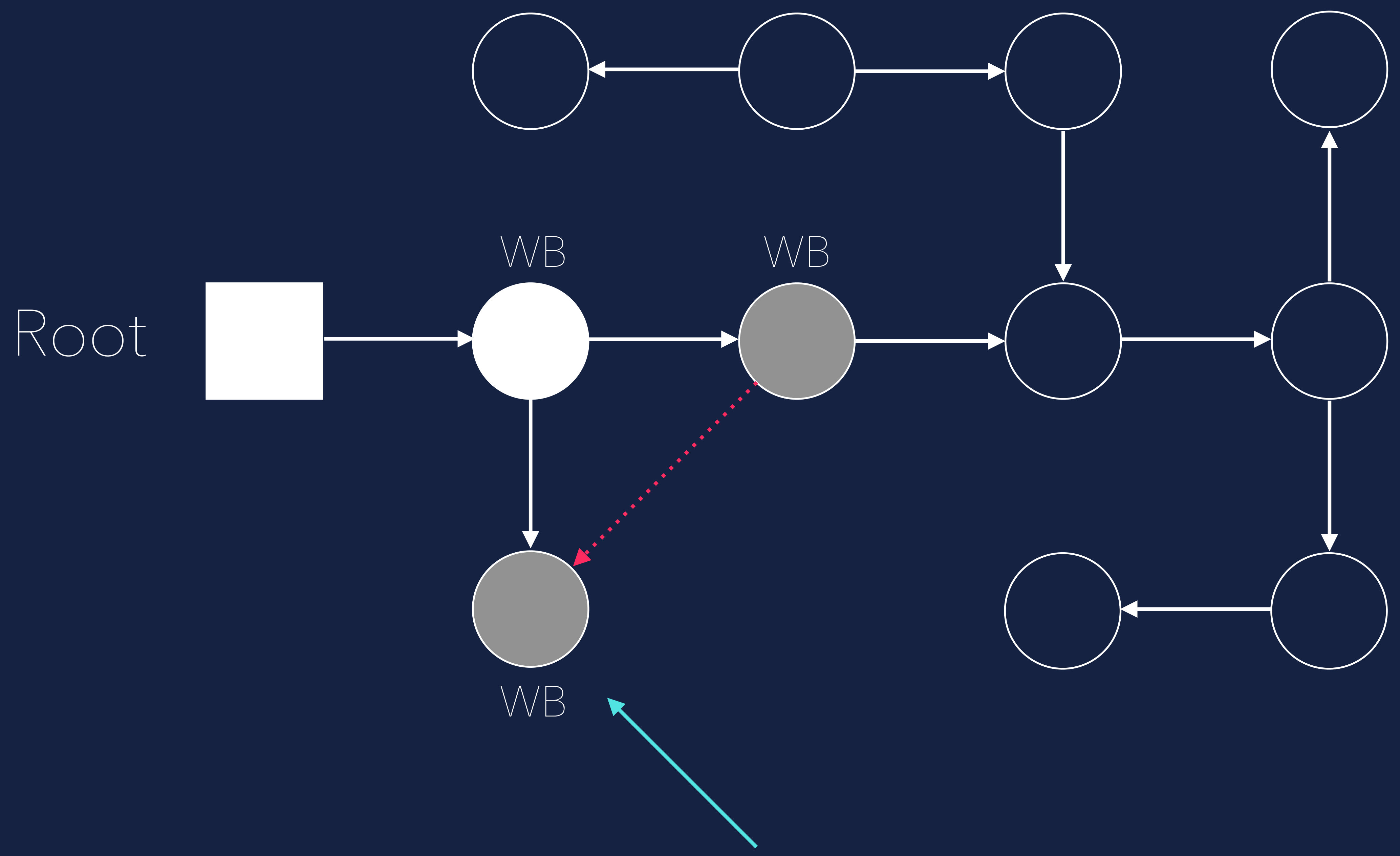
● Live

○ Not visited

Mutator hits write barrier and removes reference and adds a new one

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



● Reachable

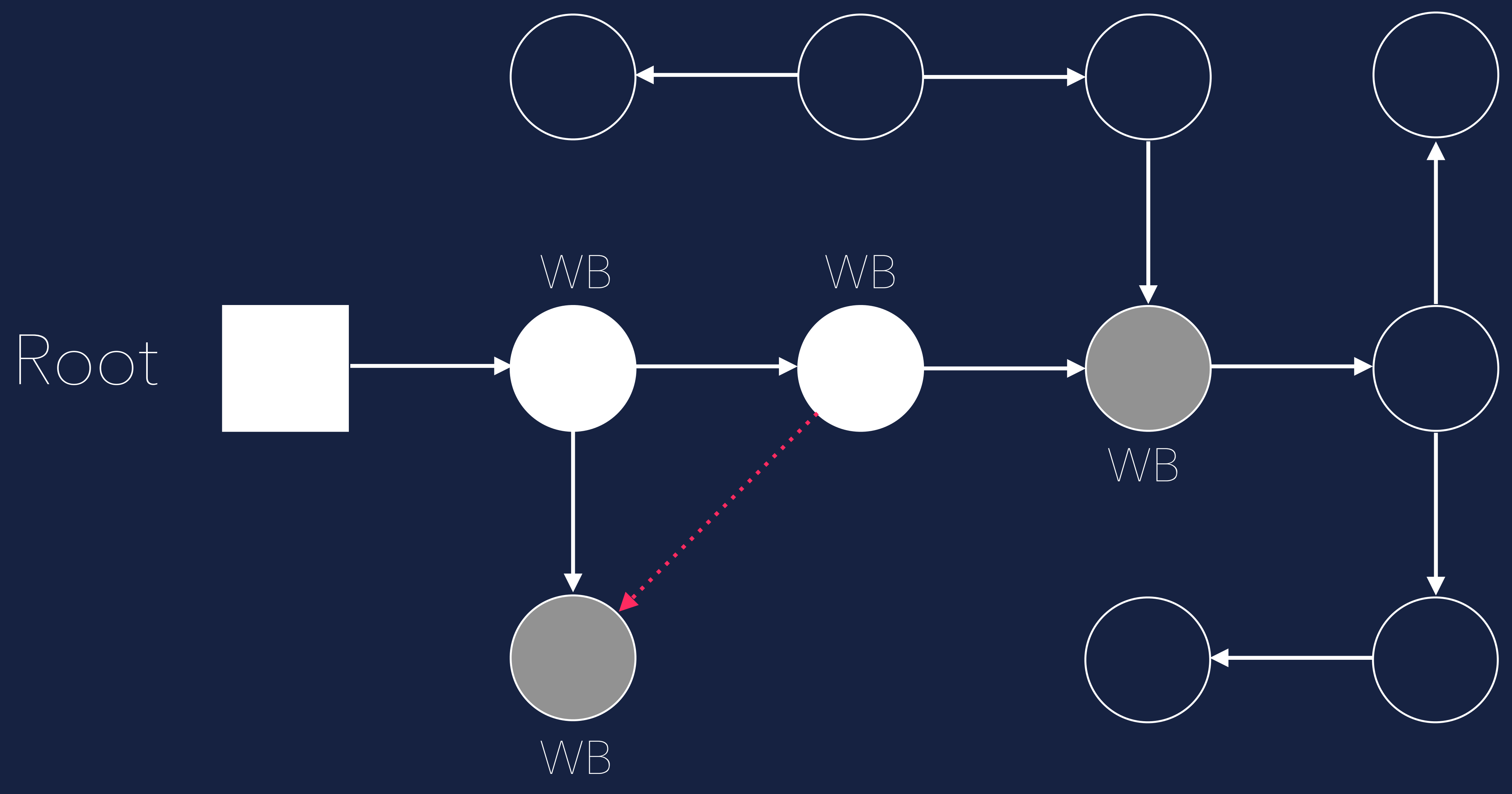
● Live

○ Not visited

Removed references will be marked as reachable by Write Barrier

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



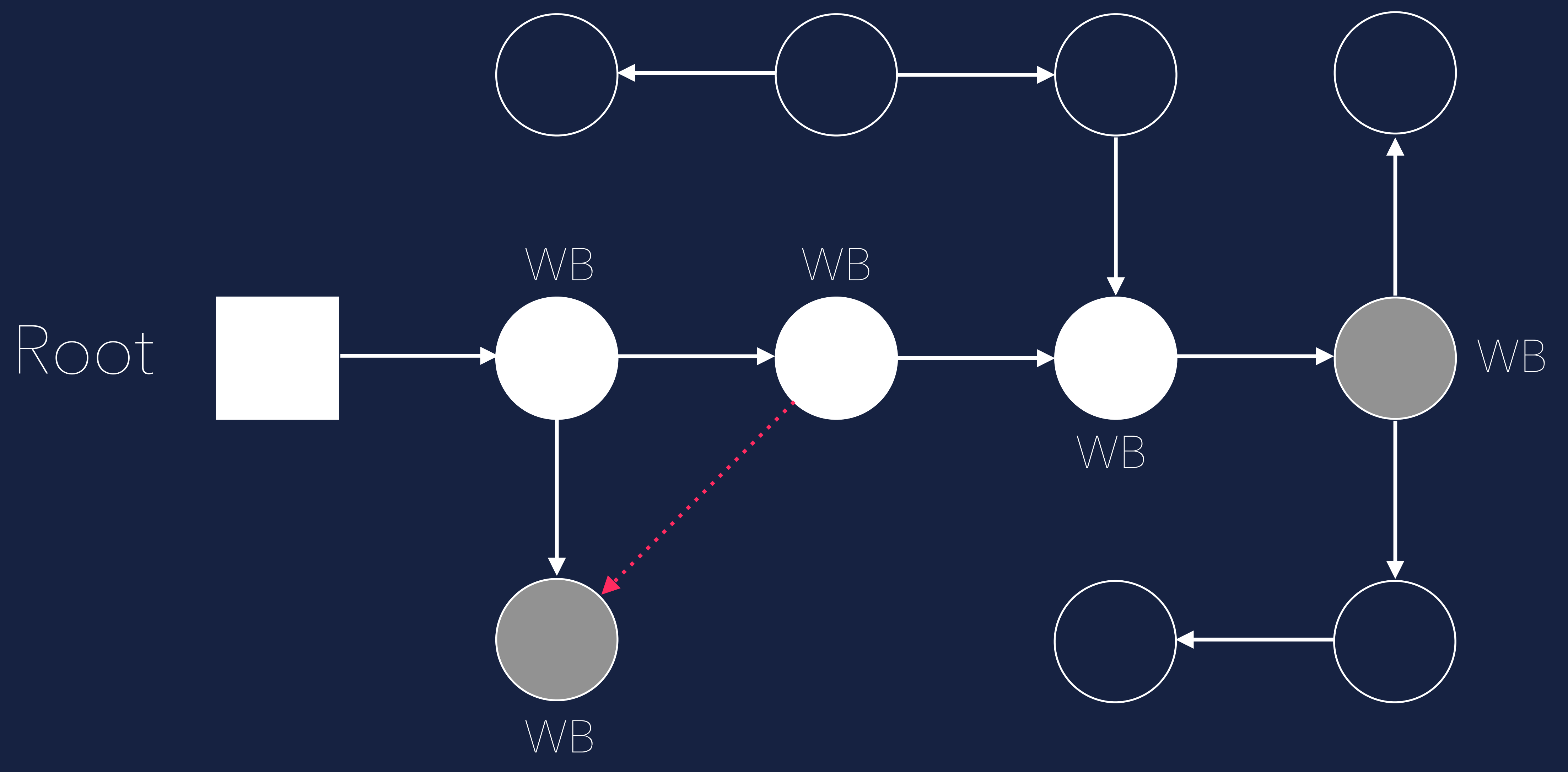
● Reachable

● Live

○ Not visited

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



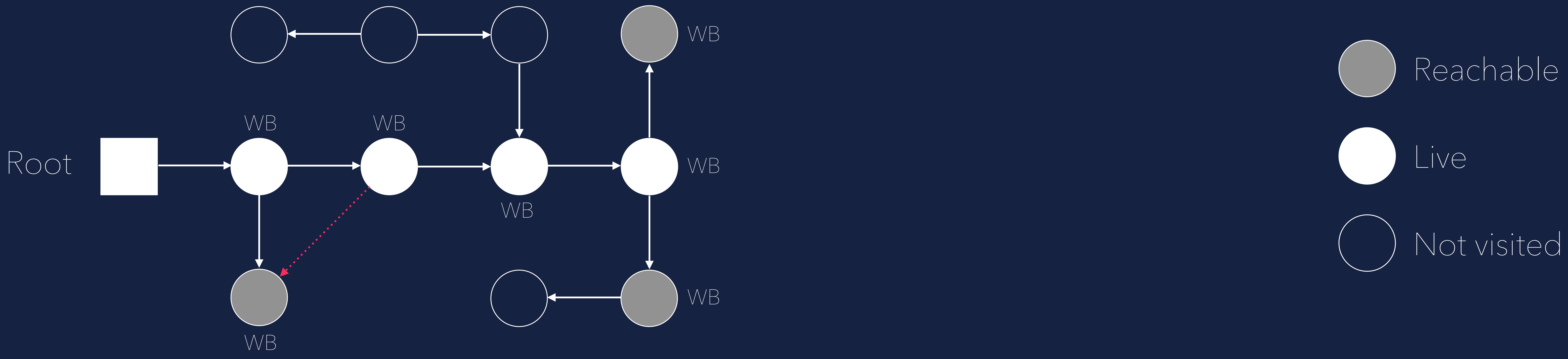
● Reachable

○ Live

○ Not visited

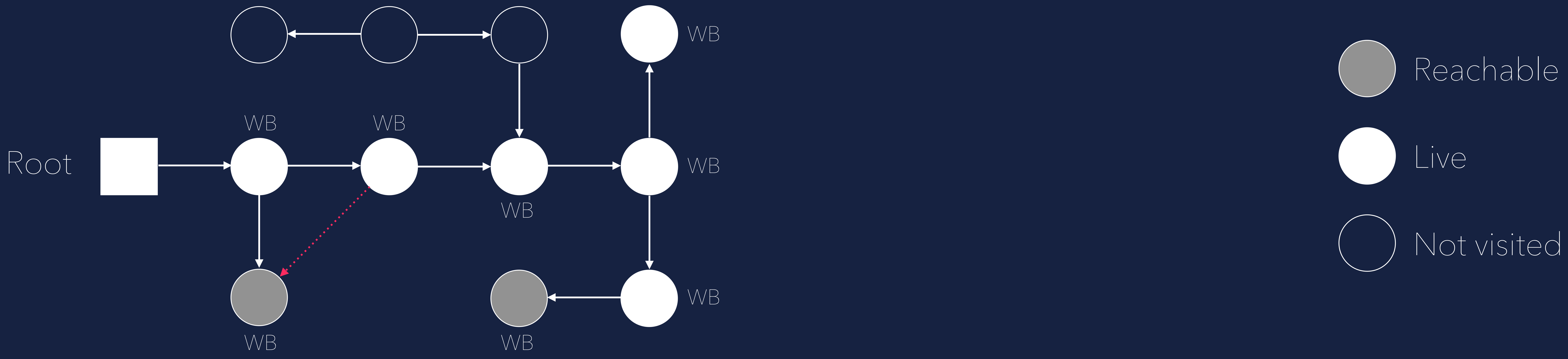
CONCURRENCY IS HARD…

Concurrent Marking using Write Barriers



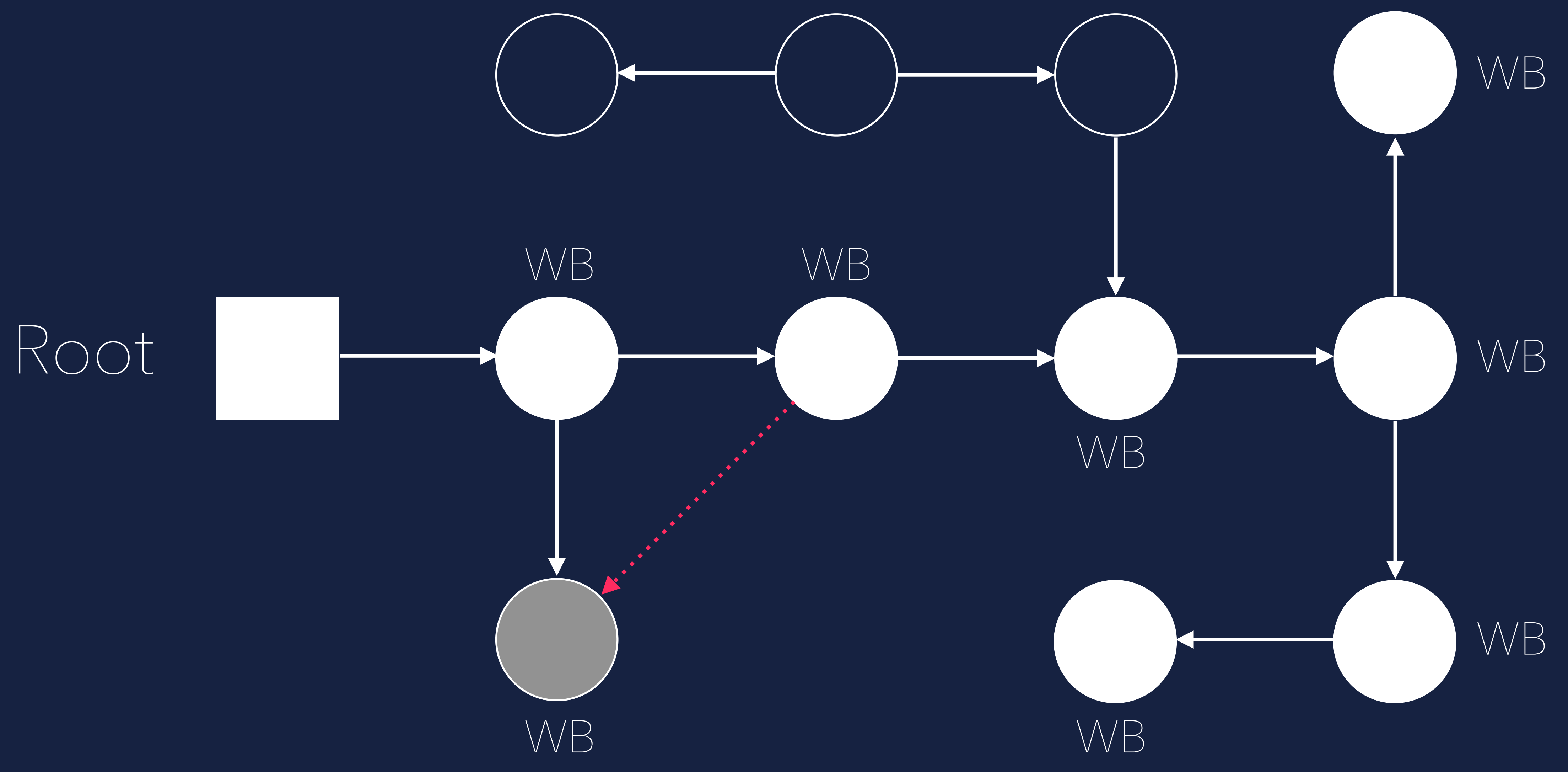
CONCURRENCY IS HARD...

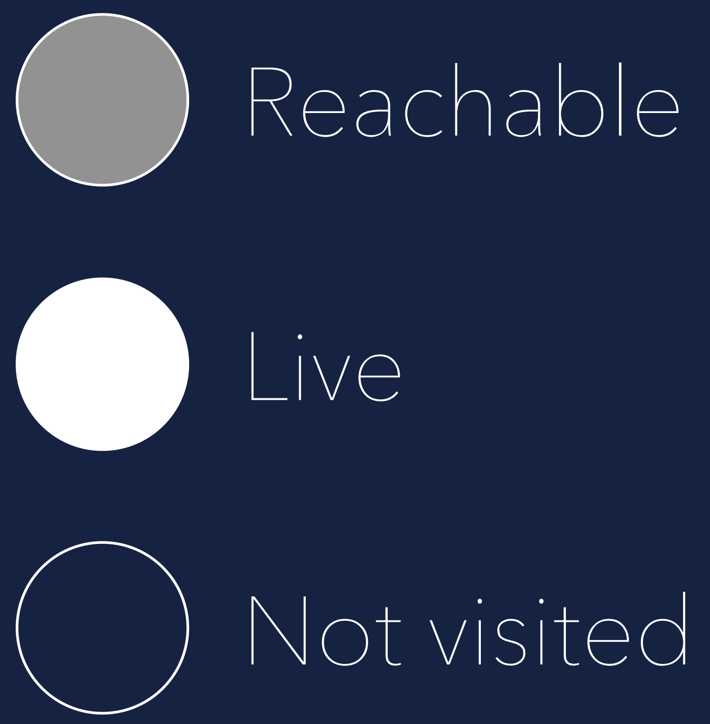
Concurrent Marking using Write Barriers



CONCURRENCY IS HARD...

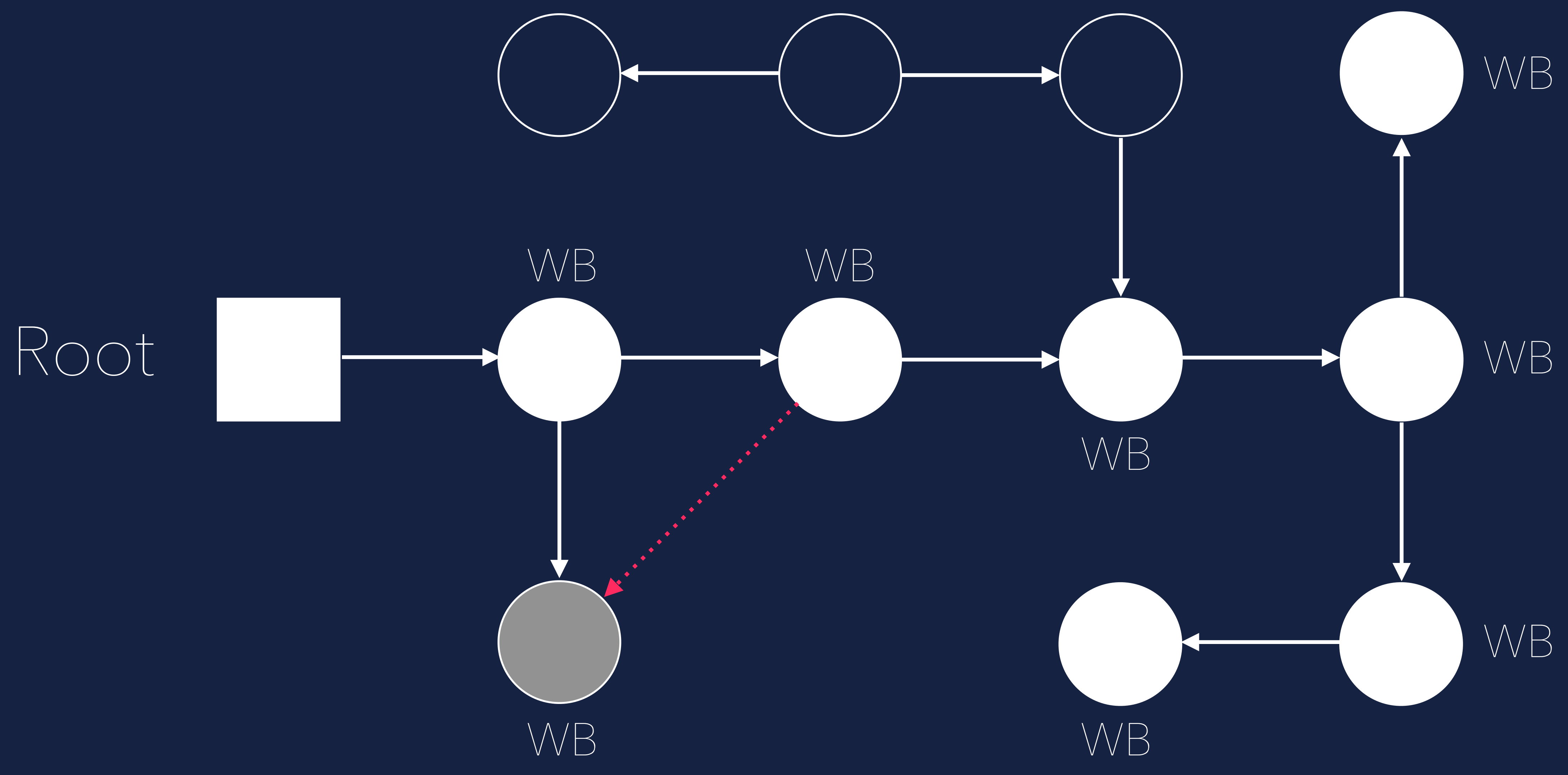
Concurrent Marking using Write Barriers

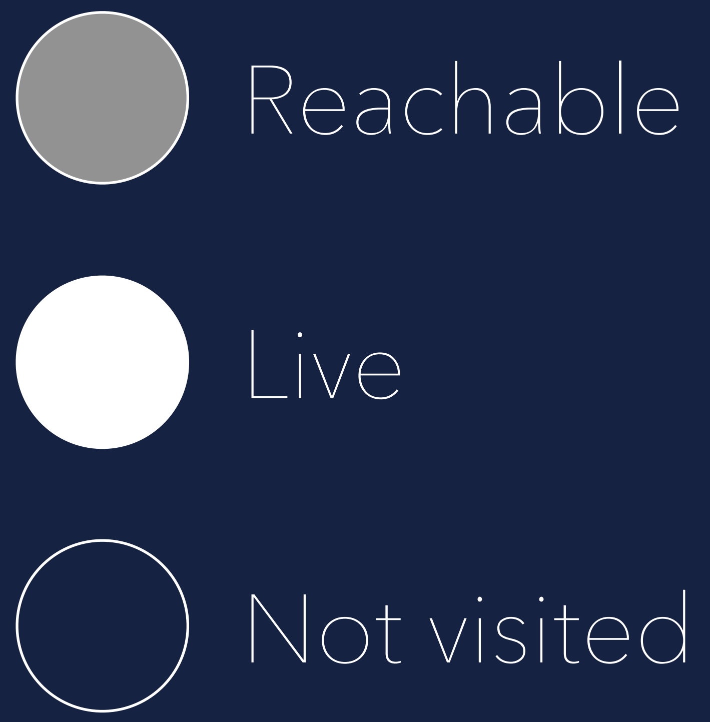




CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers

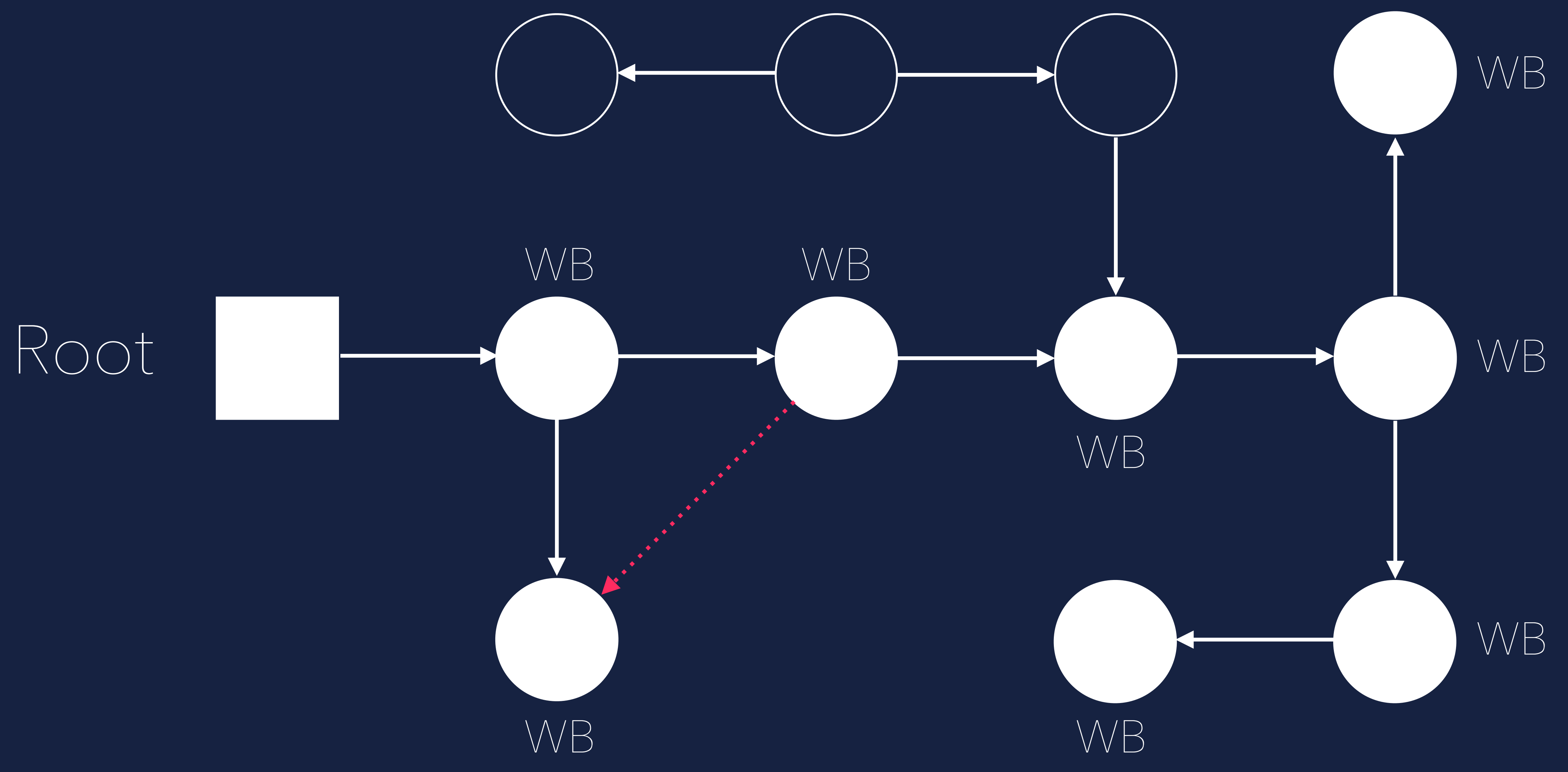


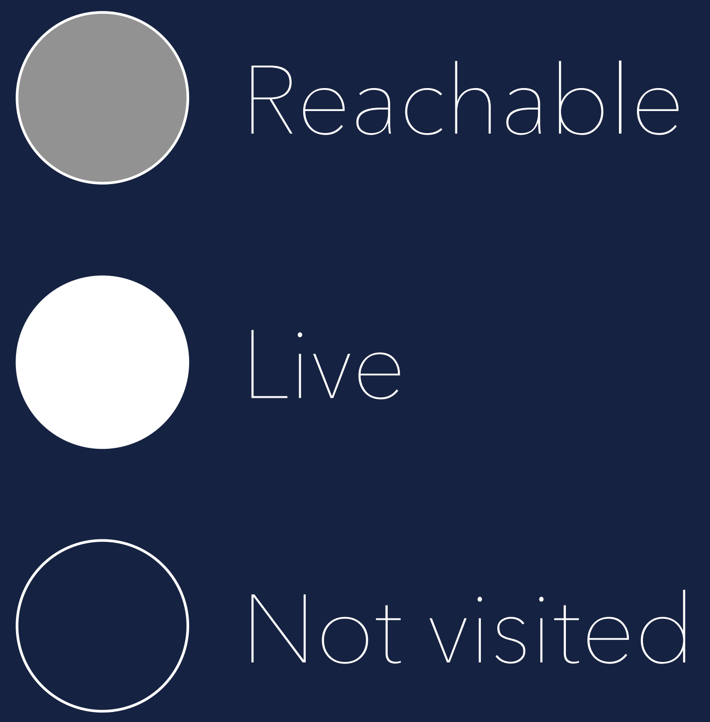


In Re-Mark phase, in between marked references will be marked as live

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers

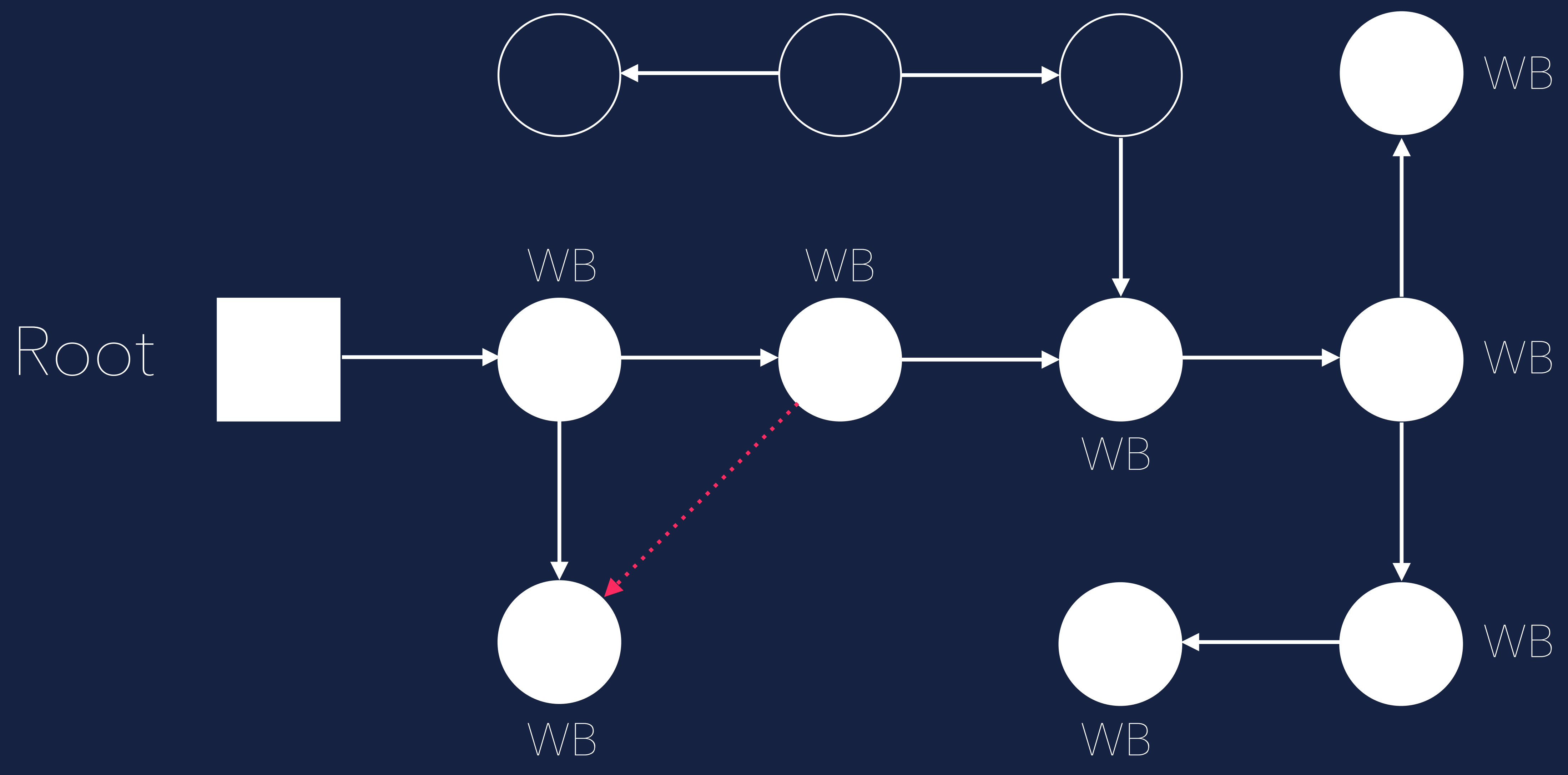




In Re-Mark phase, in between marked references will be marked as live

CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



● Reachable

● Live

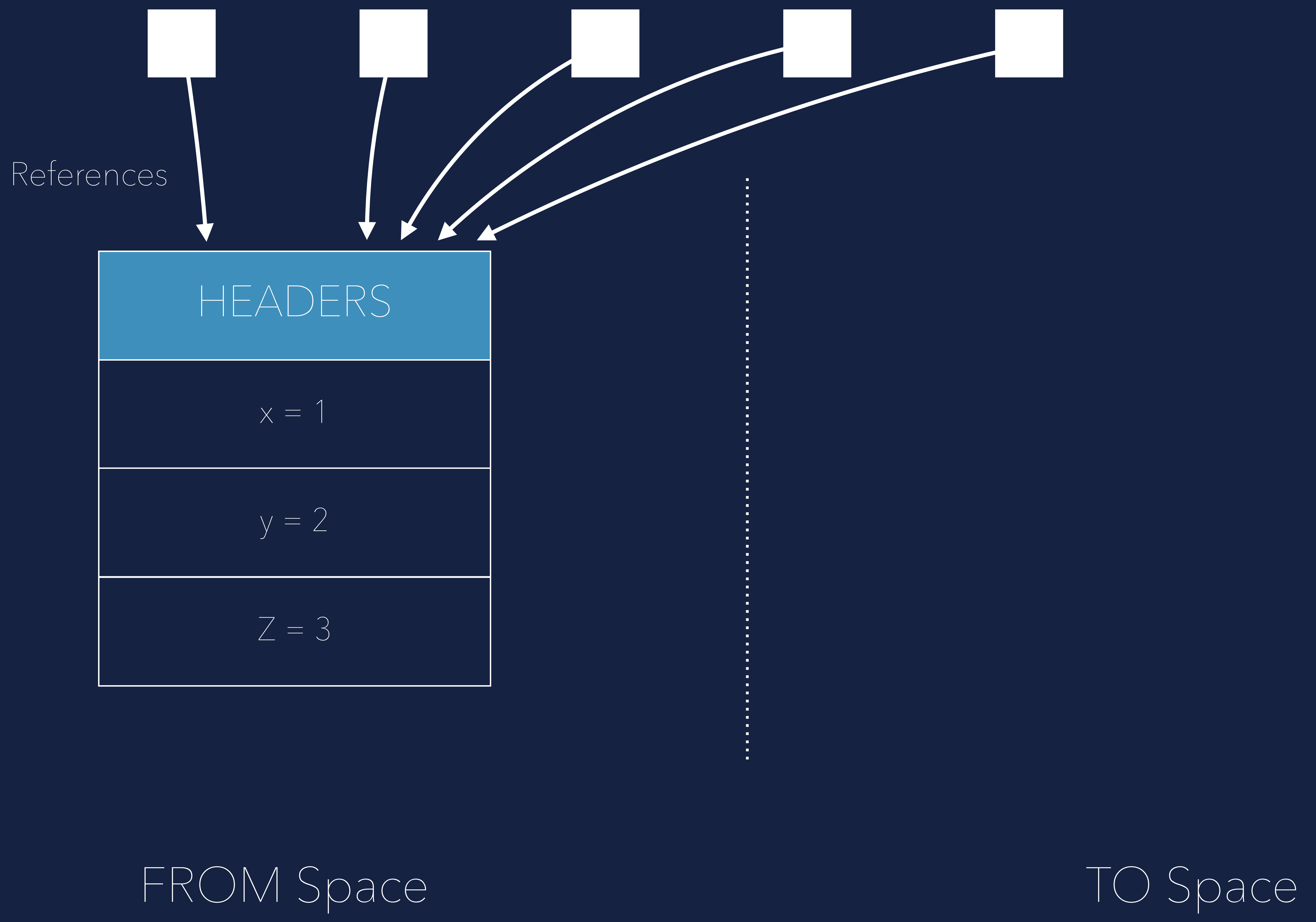
○ Not visited

"Snapshot at the beginning"

CONCURRENT COPYING

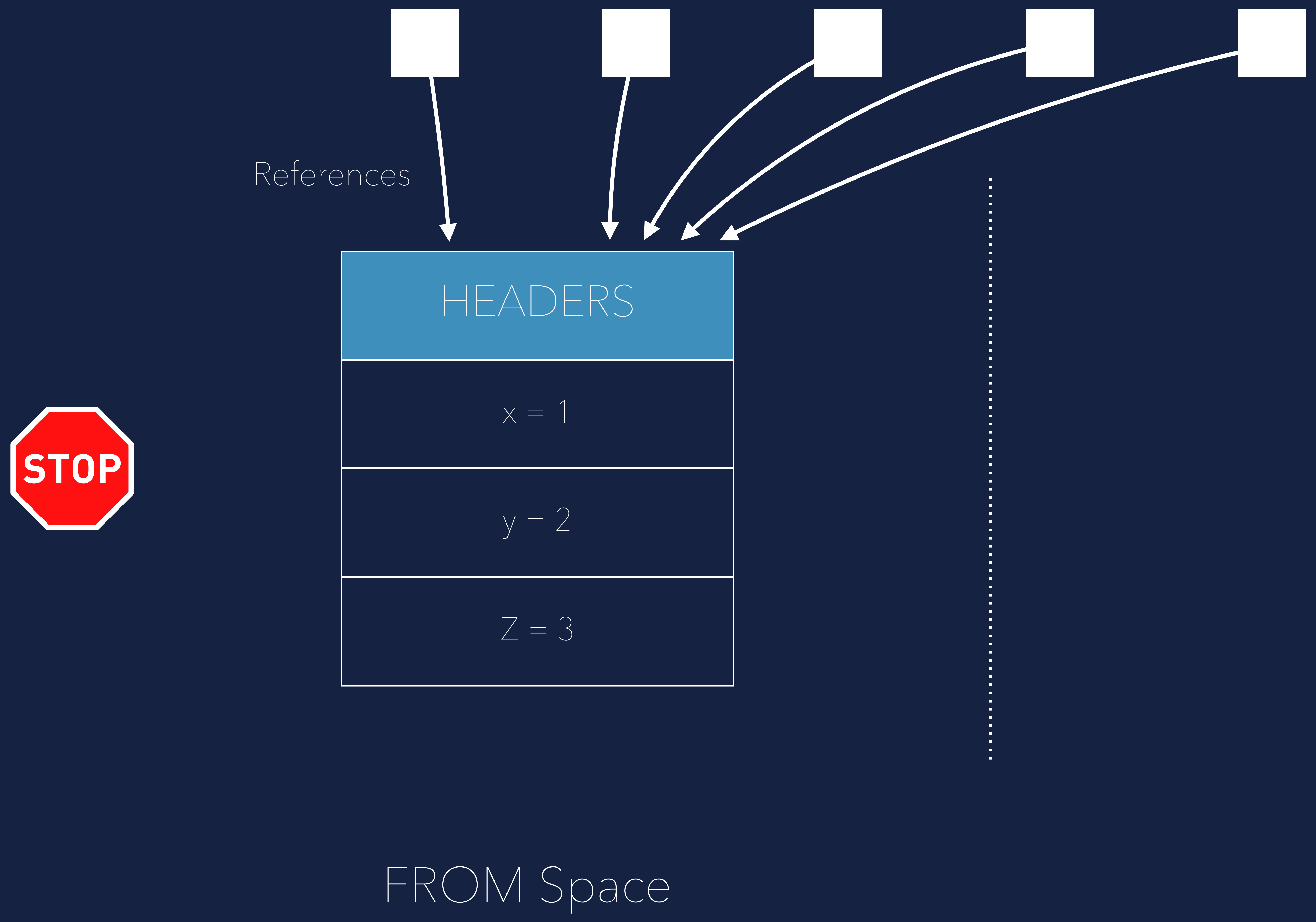
CONCURRENCY IS HARD…

Stop the world copying



CONCURRENCY IS HARD…

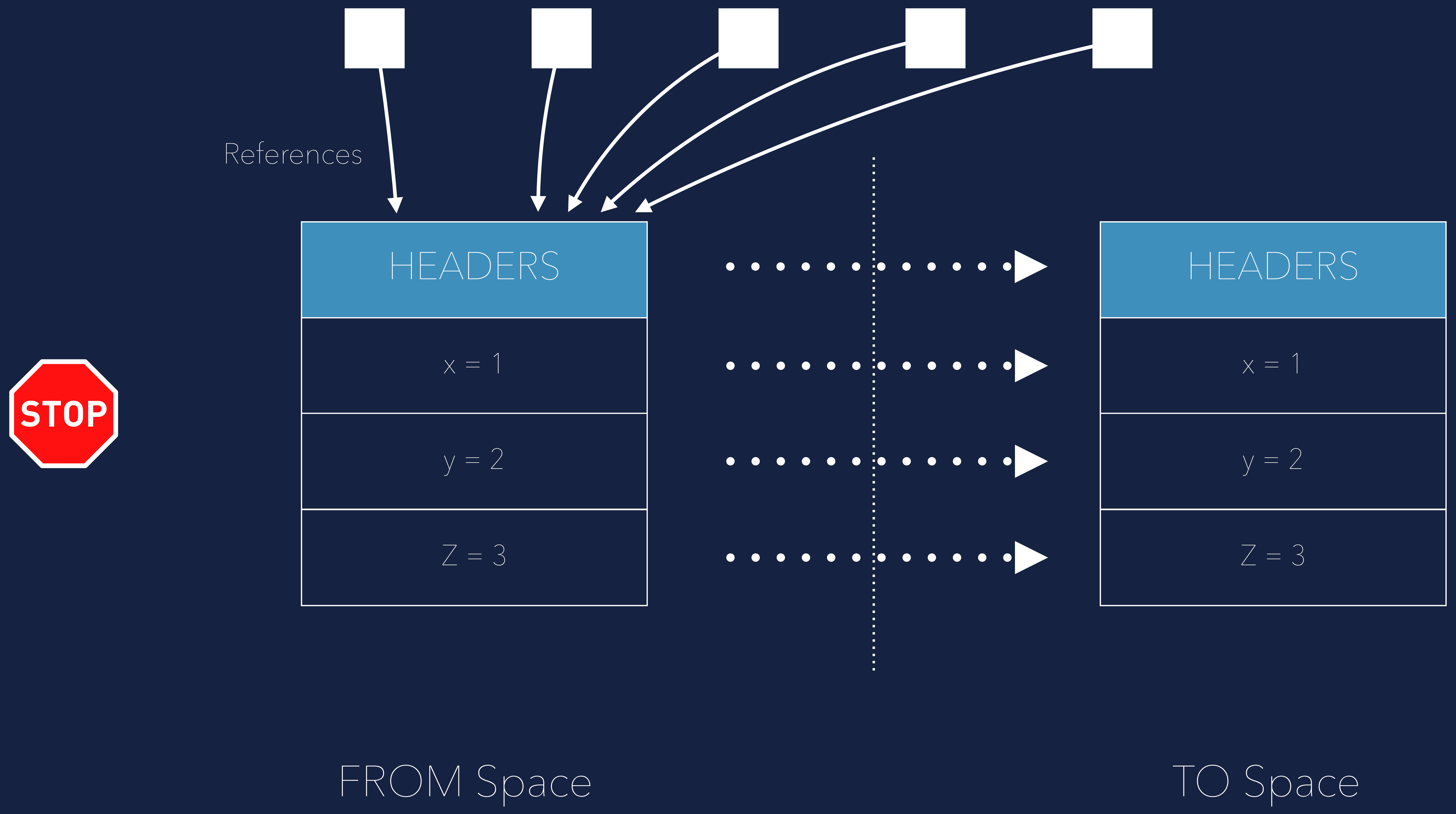
Stop the world copying



Stop the World
(the Mutator)

CONCURRENCY IS HARD...

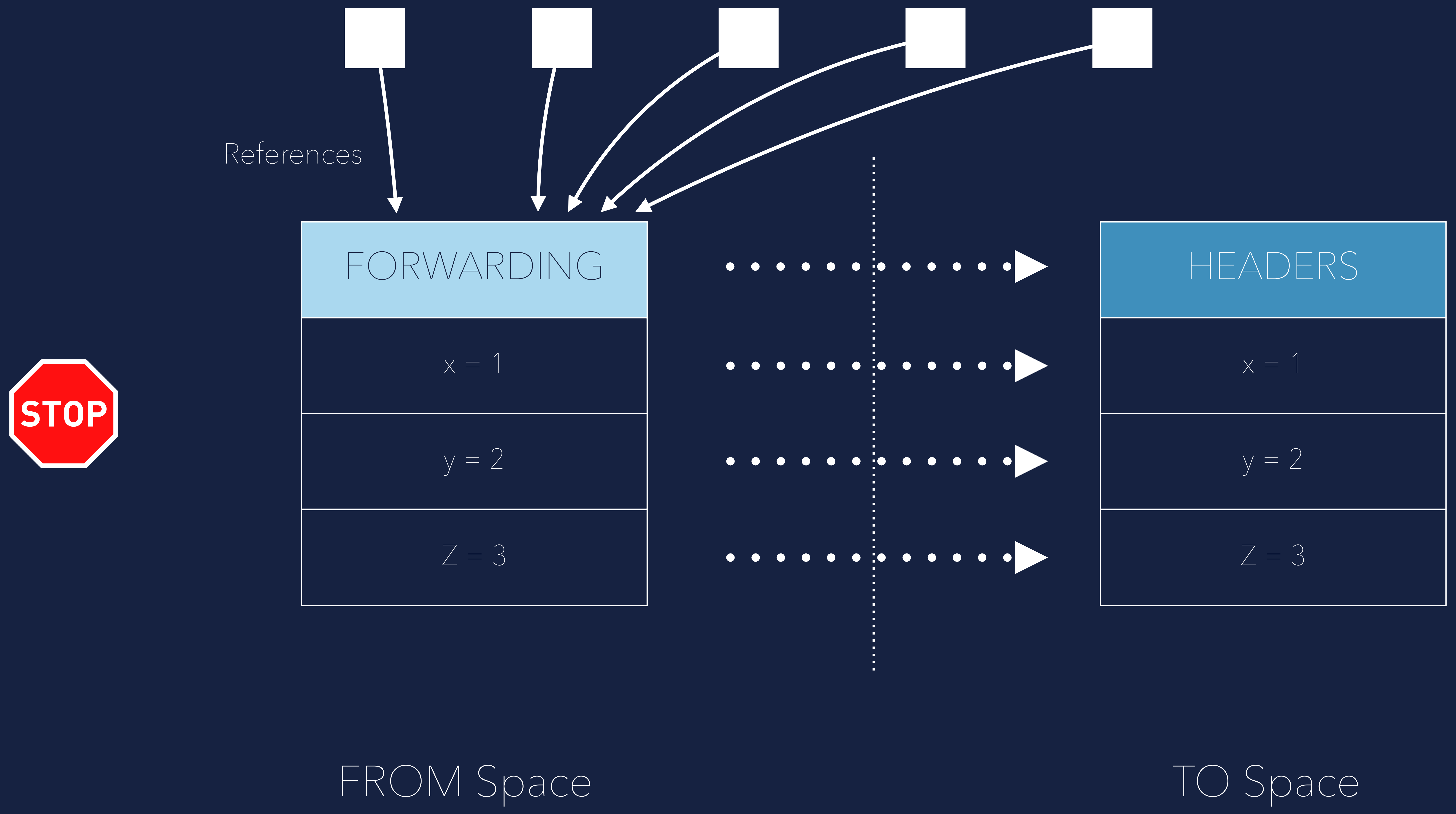
Stop the world copying



Copy the Object
(Create forwarding pointer)

CONCURRENCY IS HARD...

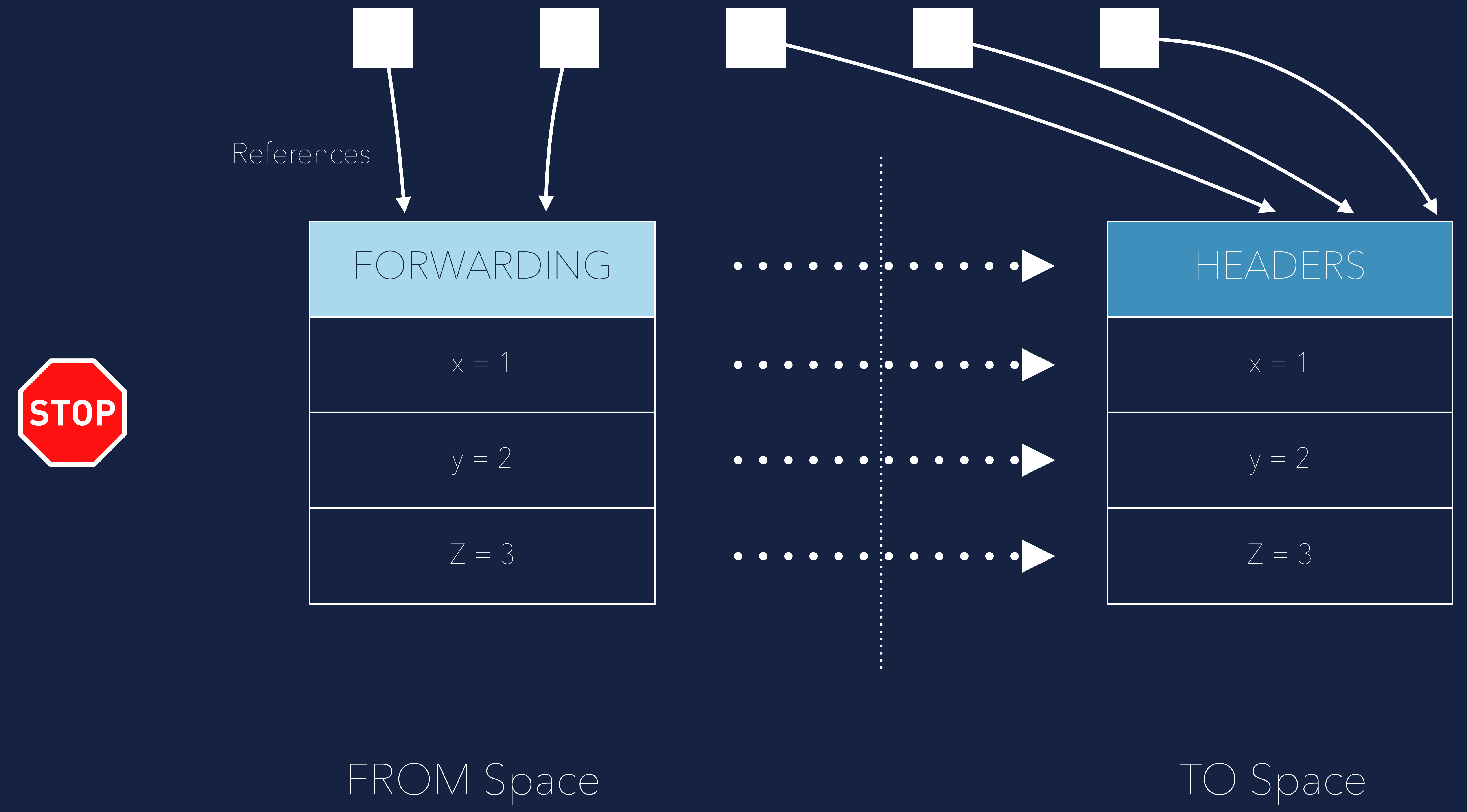
Stop the world copying



Update all references
(Save the pointer that fowards the copy)

CONCURRENCY IS HARD...

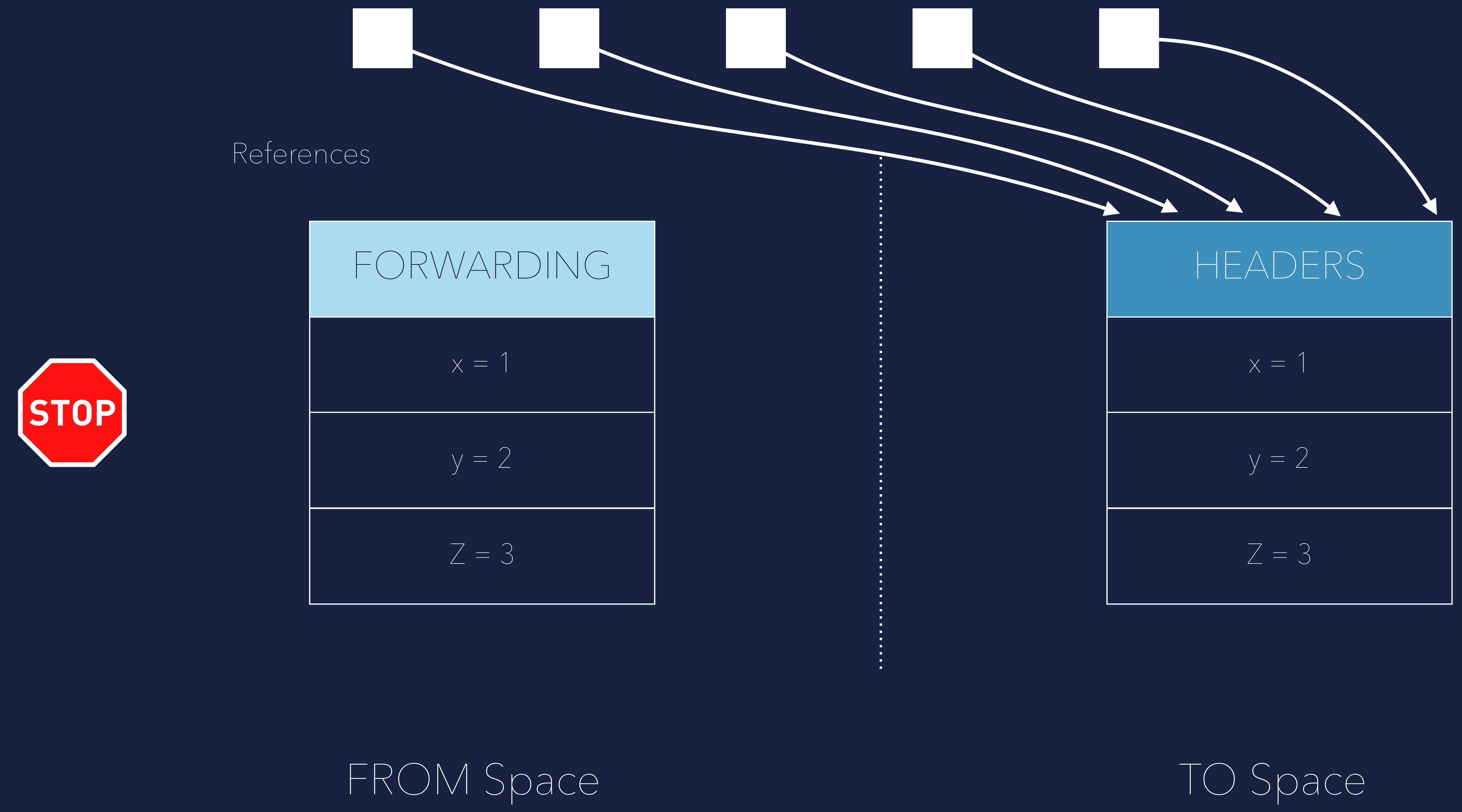
Stop the world copying



Update all references
(Walk the heap and replace all references with forwarding pointer to new location)

CONCURRENCY IS HARD...

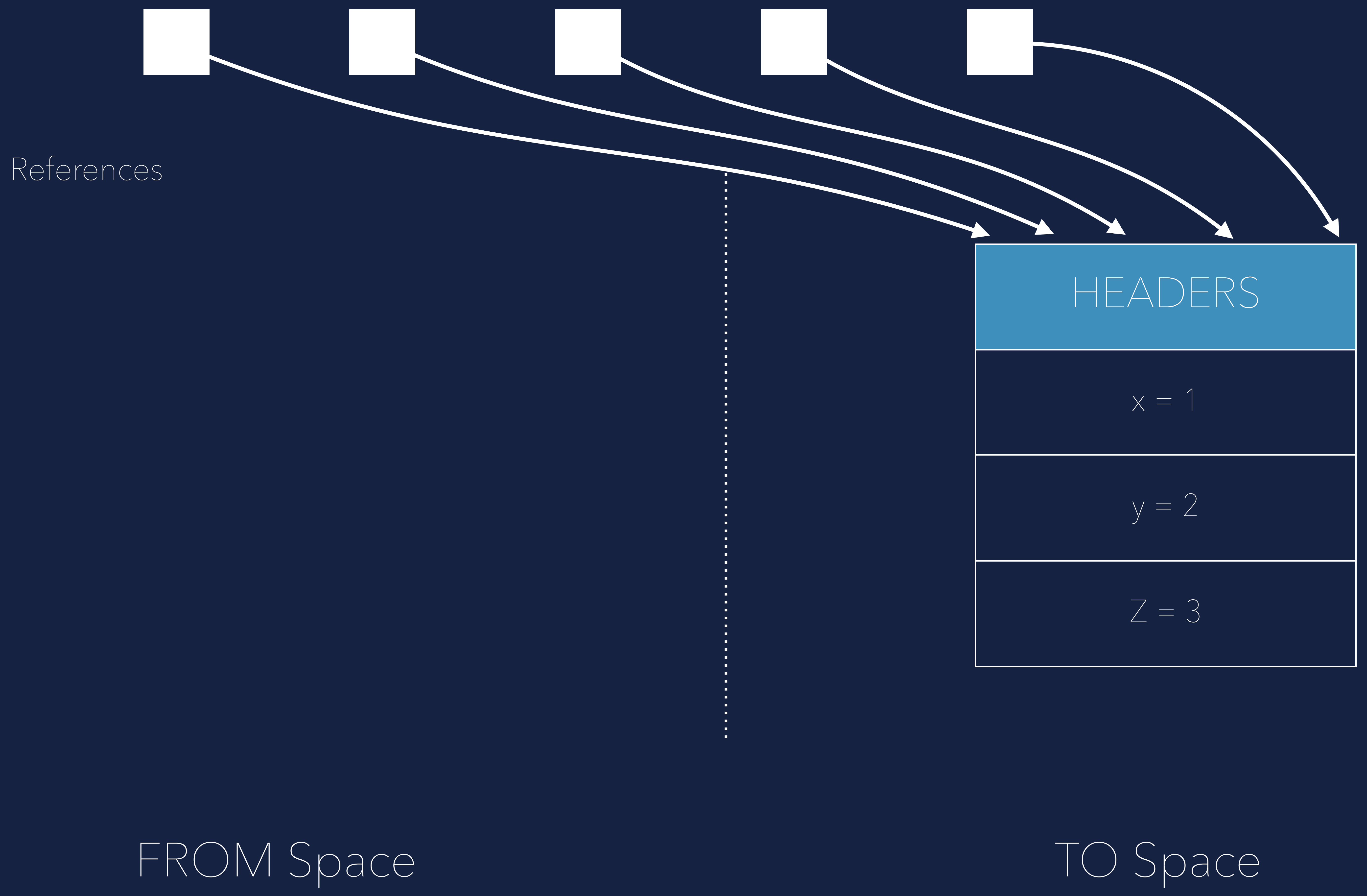
Stop the world copying



Update all references
(Walk the heap and replace all references with forwarding pointer to new location)

CONCURRENCY IS HARD...

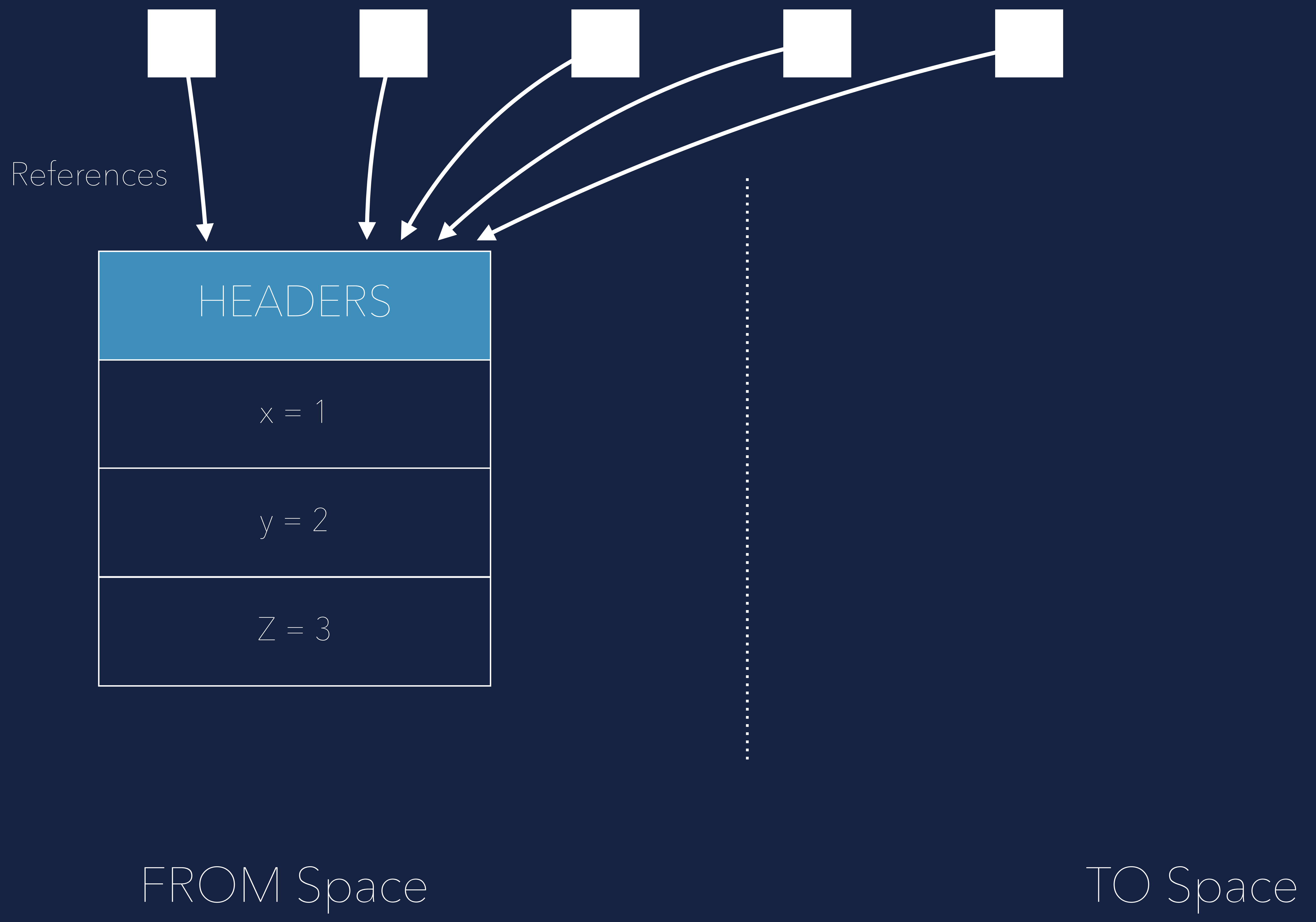
Stop the world copying



Remove old objects and continue running the Mutator

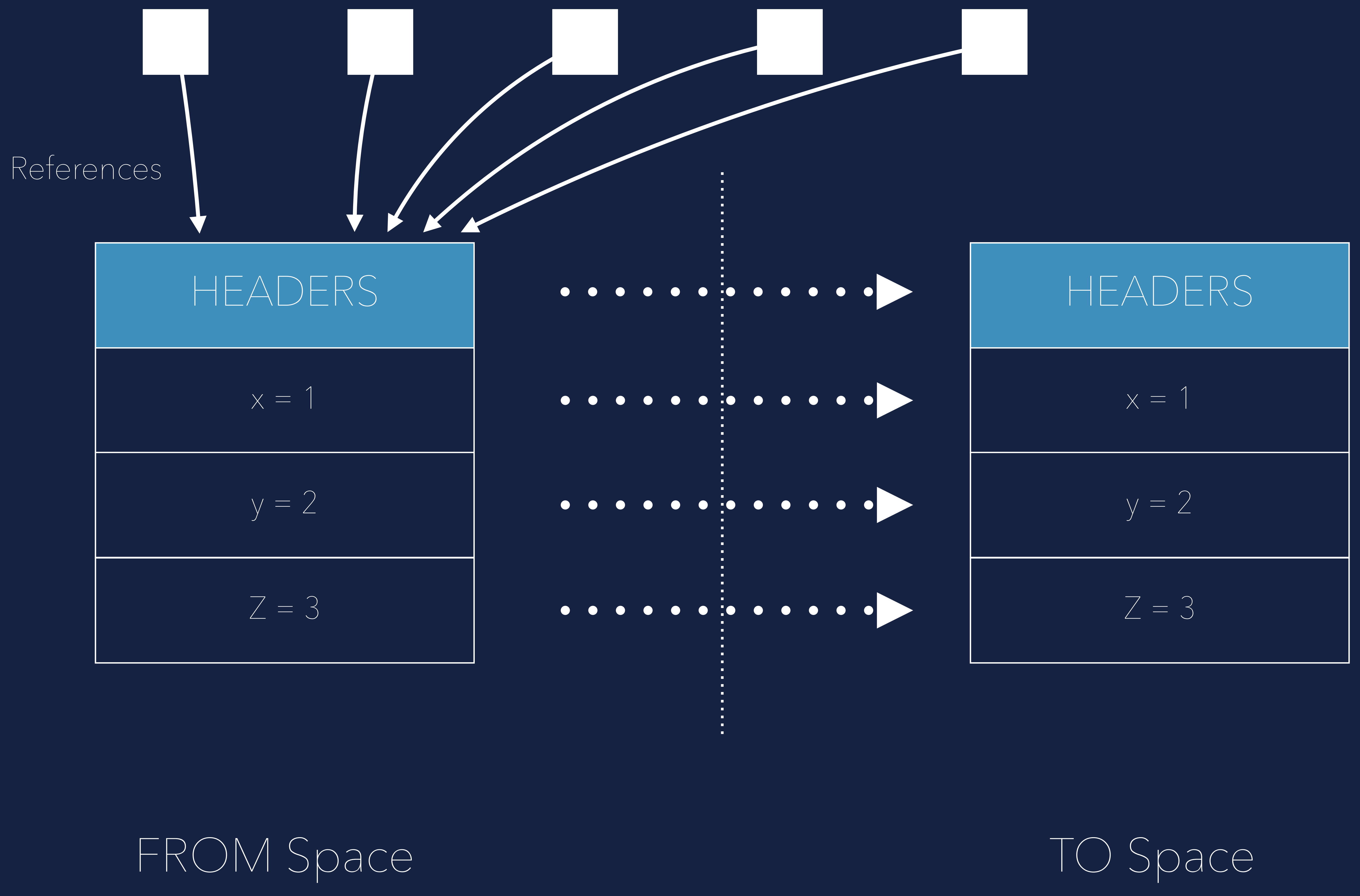
CONCURRENCY IS HARD...

Concurrent copying



CONCURRENCY IS HARD...

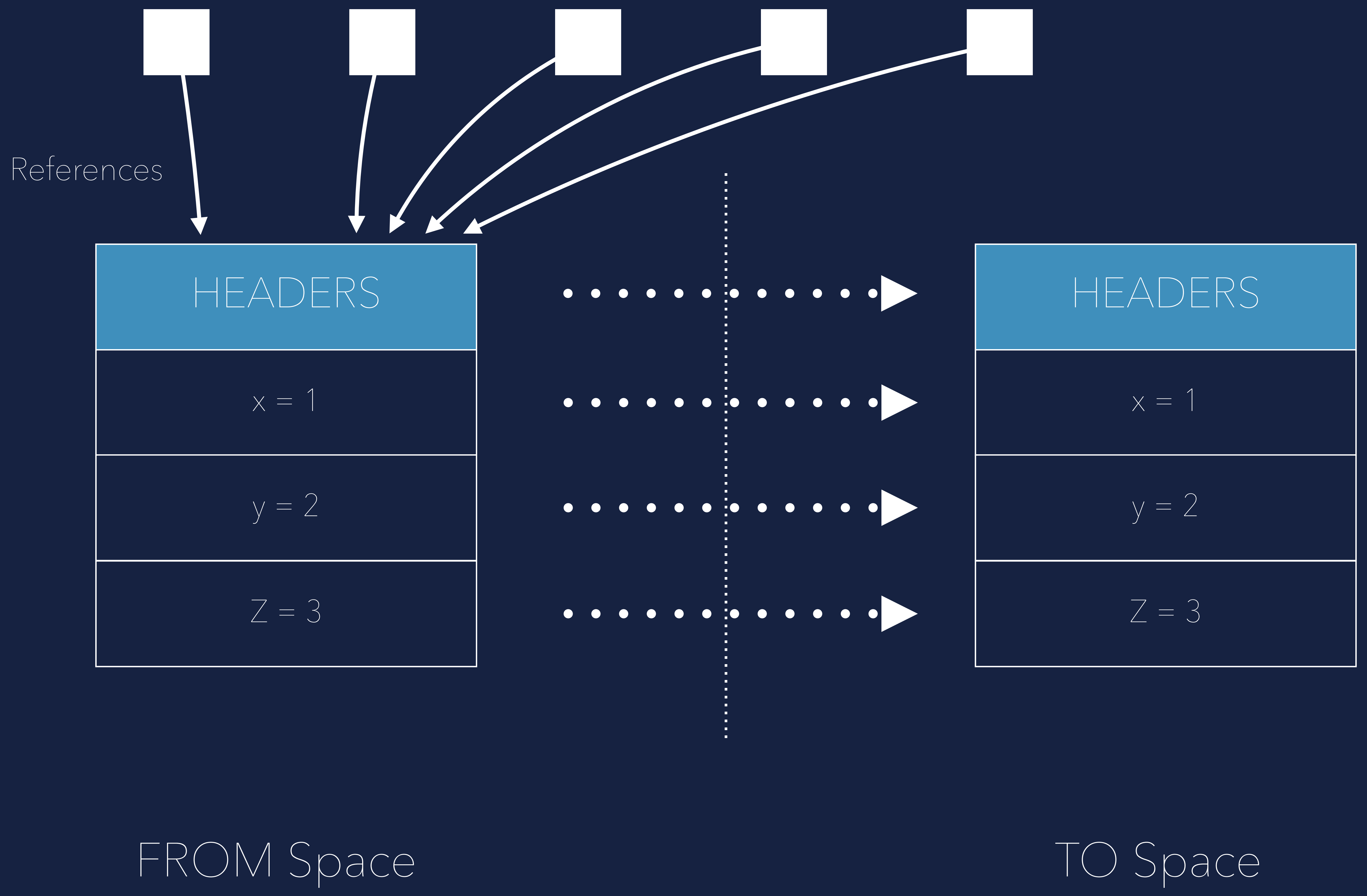
Concurrent copying



Copy the Object

CONCURRENCY IS HARD...

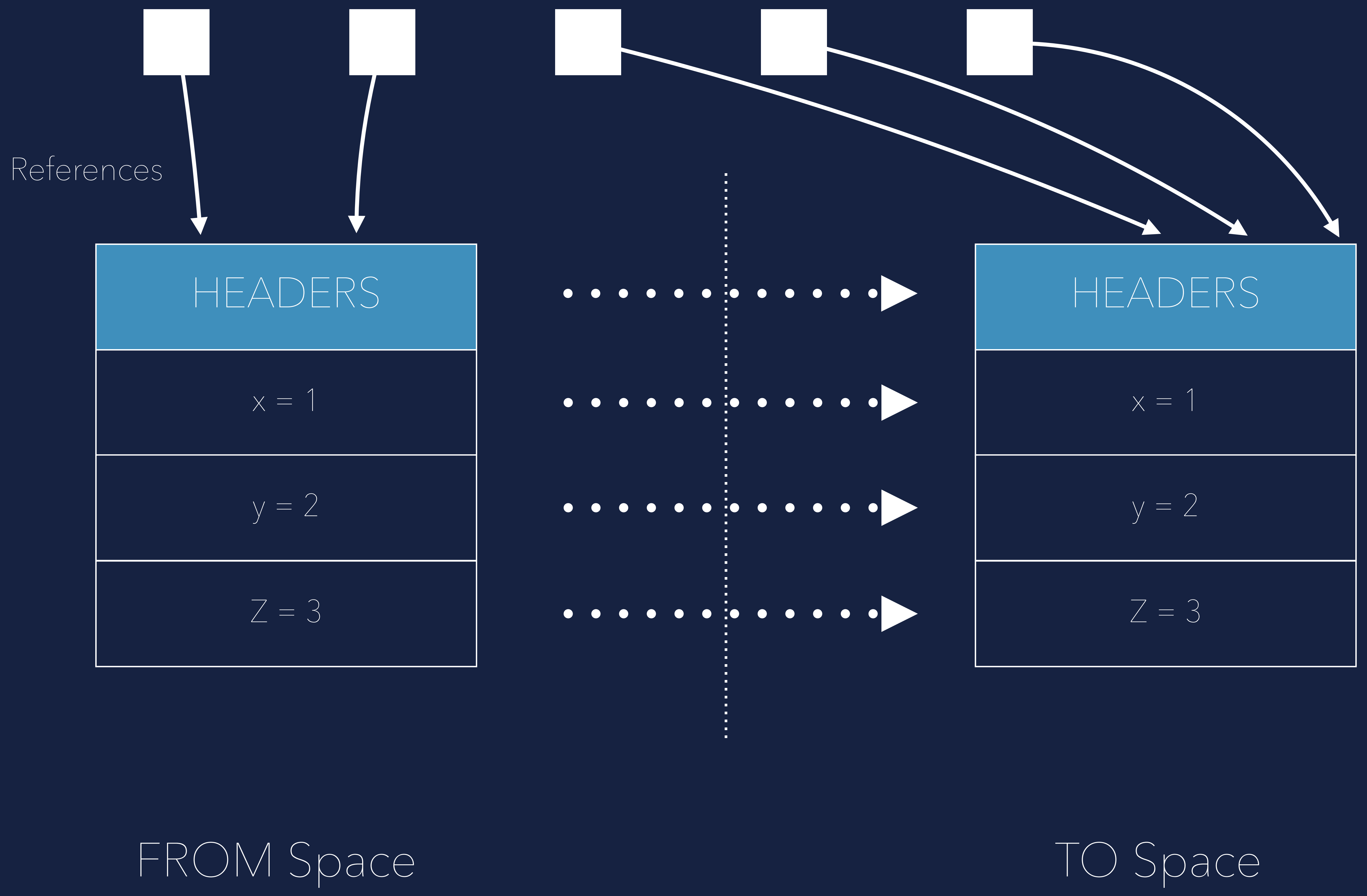
Concurrent copying



Now both Objects are reachable !

CONCURRENCY IS HARD...

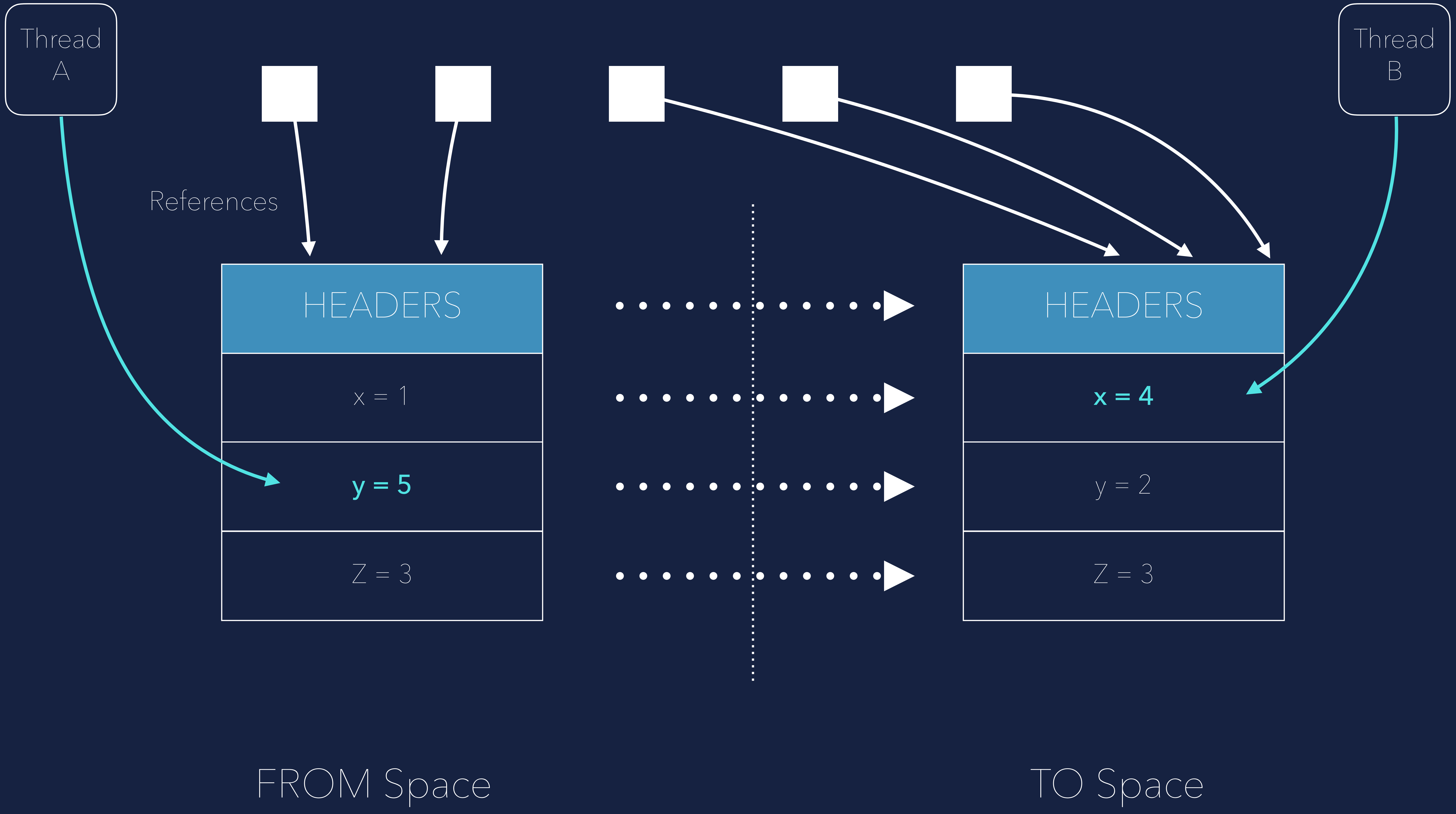
Concurrent copying



Now both Objects are reachable !
And can be accessed in parallel by different Threads.

CONCURRENCY IS HARD...

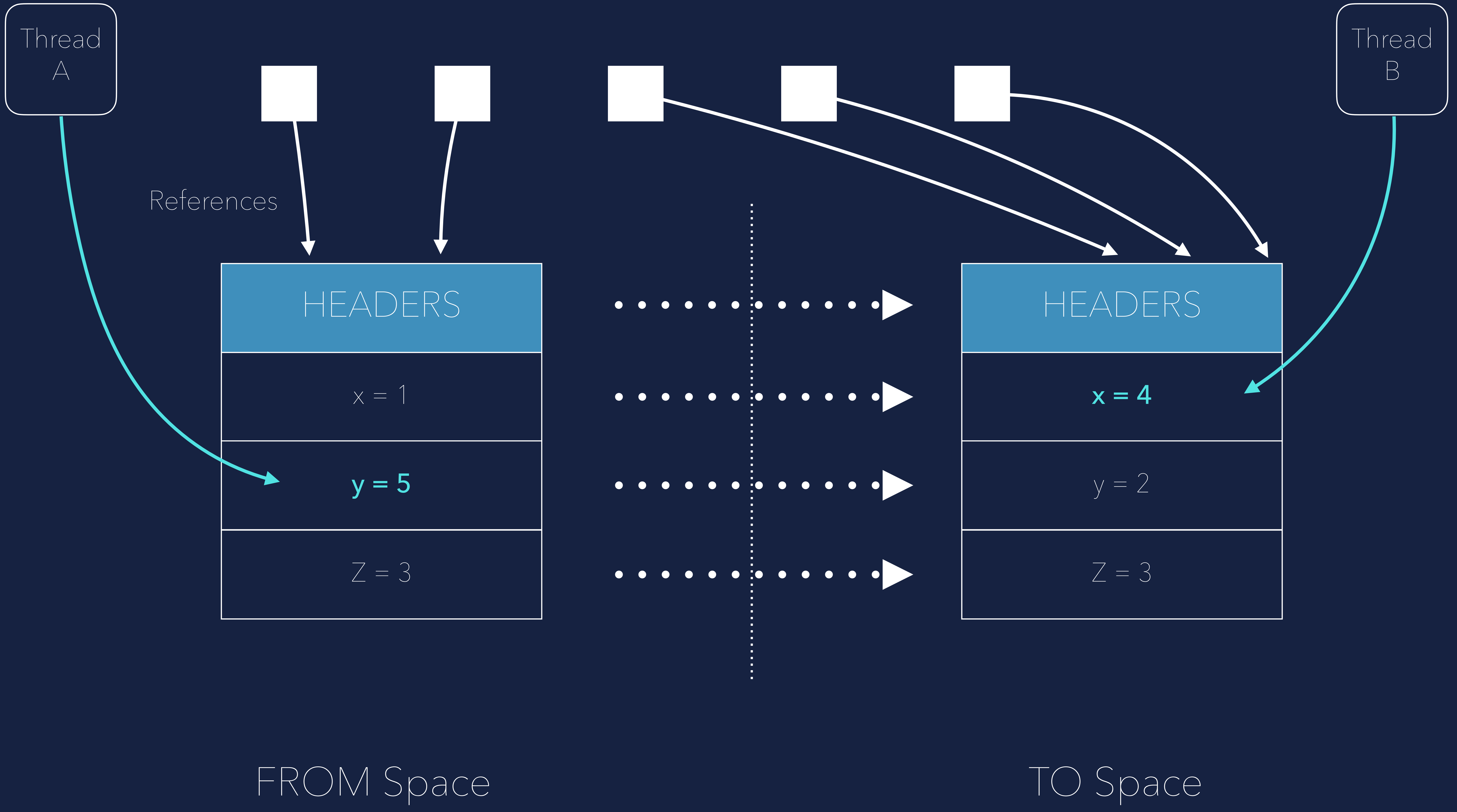
Concurrent copying



Threads can write to both Objects !

CONCURRENCY IS HARD…

Concurrent copying

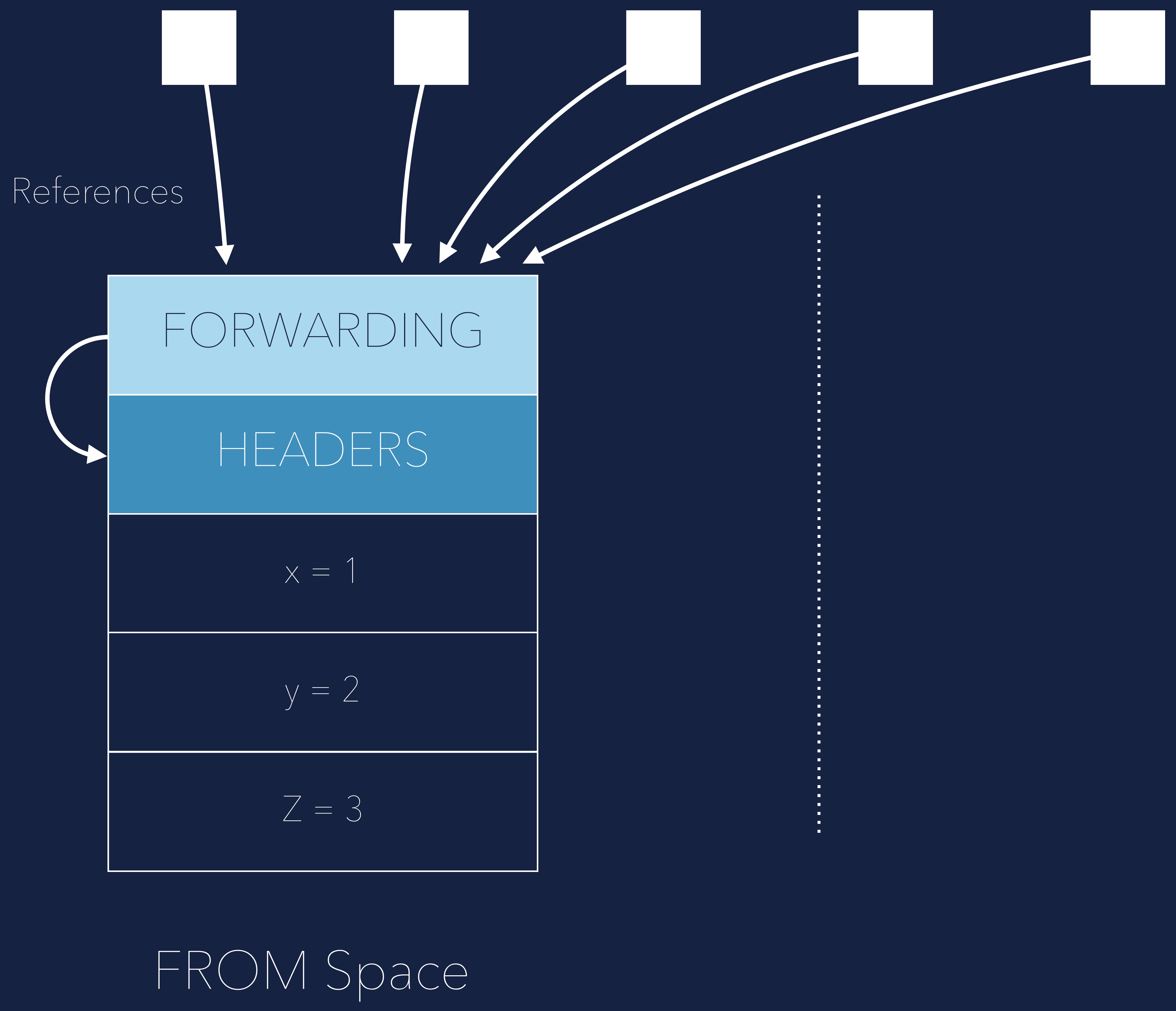


Threads can write to both Objects !

Which copy is correct ?

CONCURRENCY IS HARD...

Concurrent copying

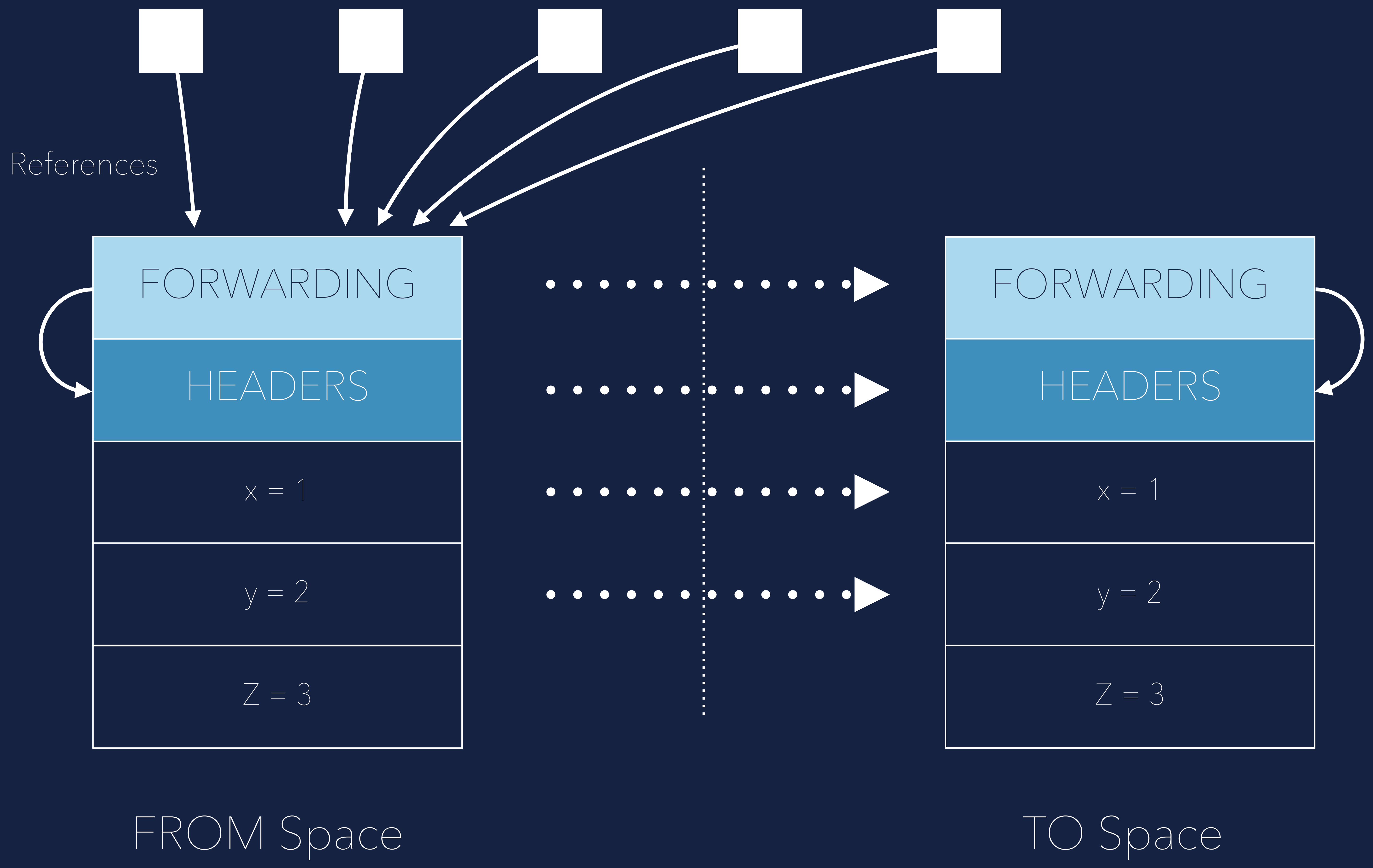


Solution could be a Brooks Pointer

(Initially points to the Object itself)

CONCURRENCY IS HARD...

Concurrent copying

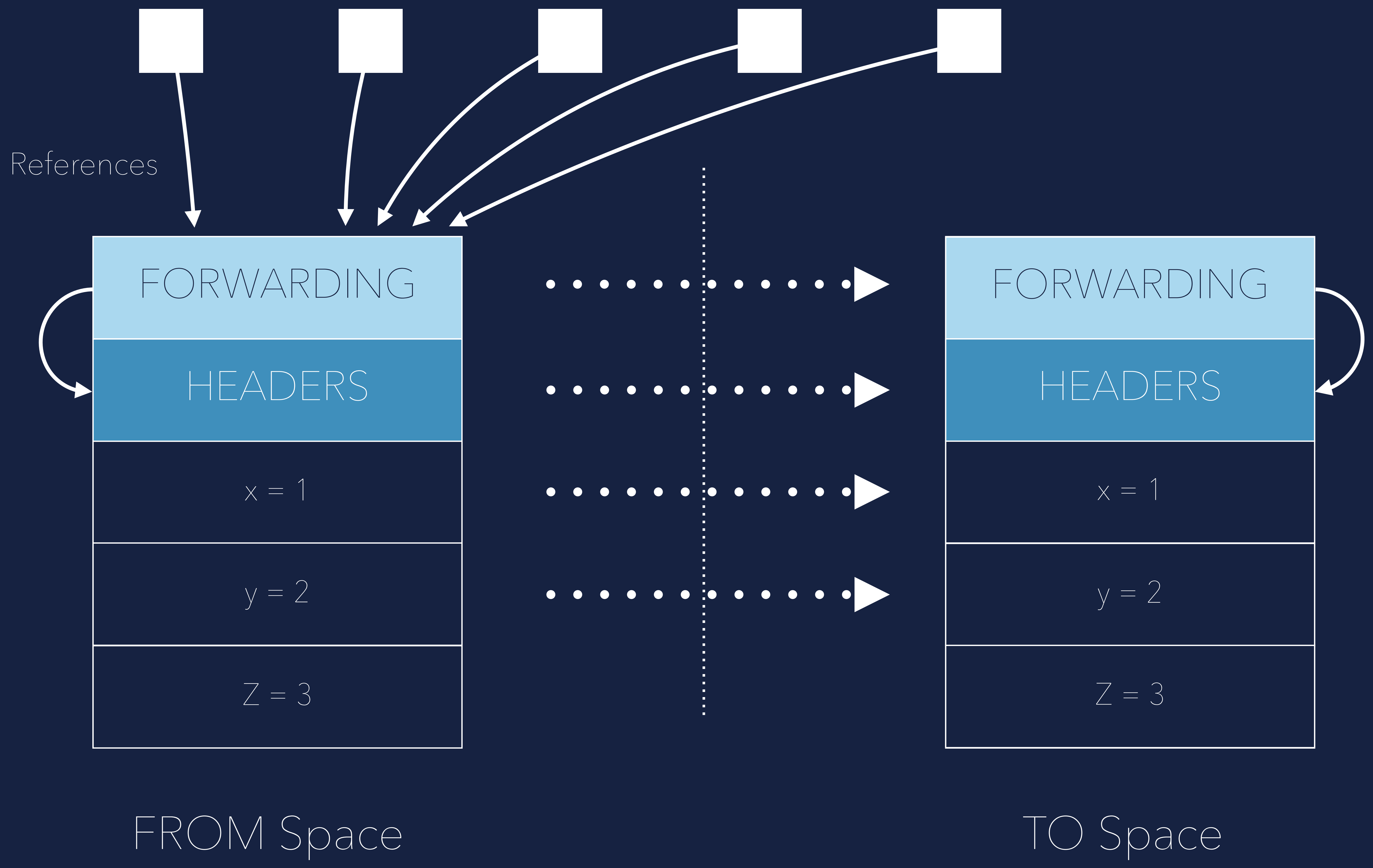


Copy the Object

(Init fowarding pointer to itself)

CONCURRENCY IS HARD...

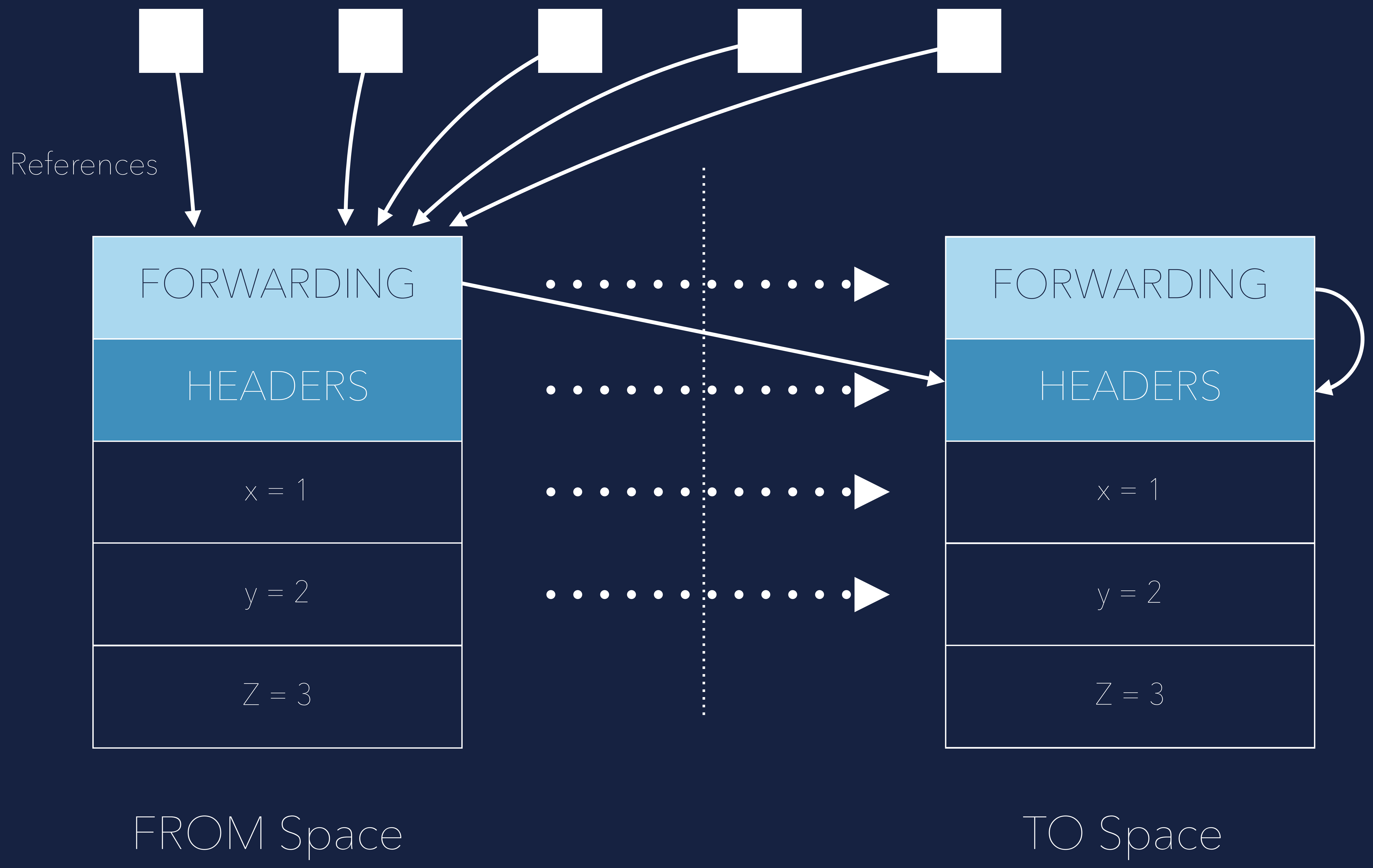
Concurrent copying



Nobody knows about copy

CONCURRENCY IS HARD...

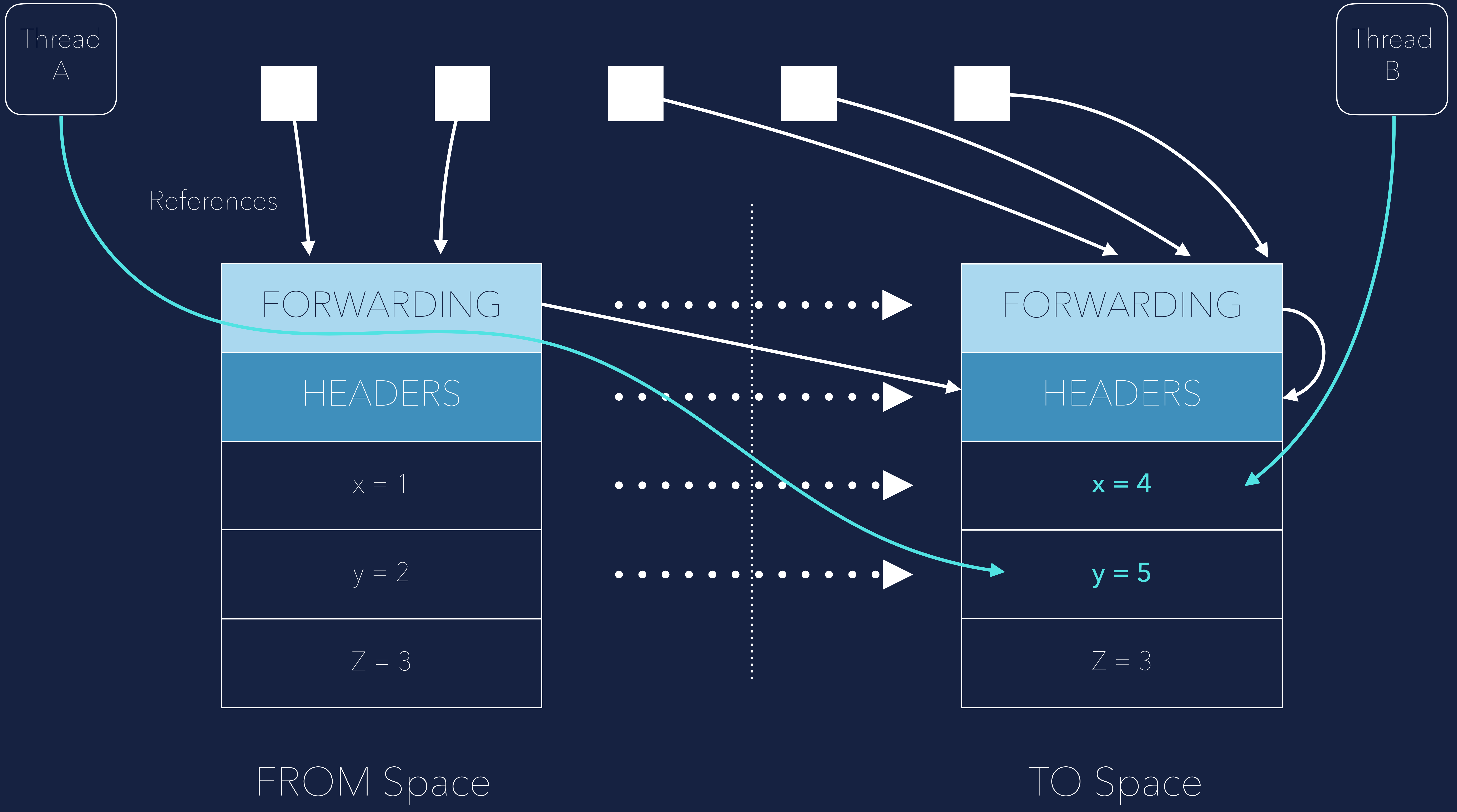
Concurrent copying



Install forwarding pointer
of original object
to new copy

CONCURRENCY IS HARD...

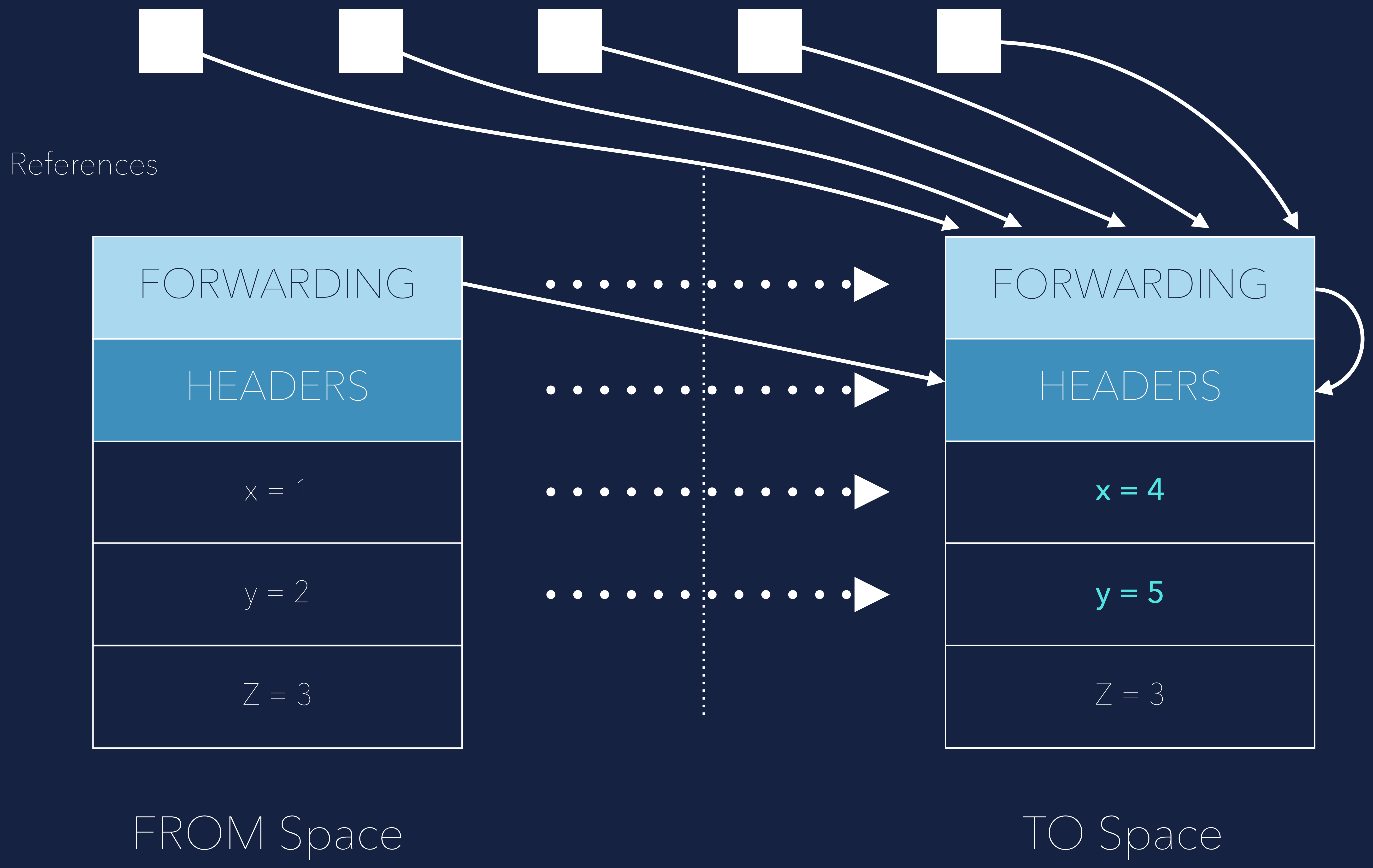
Concurrent copying



Threads now will always find the right object

CONCURRENCY IS HARD...

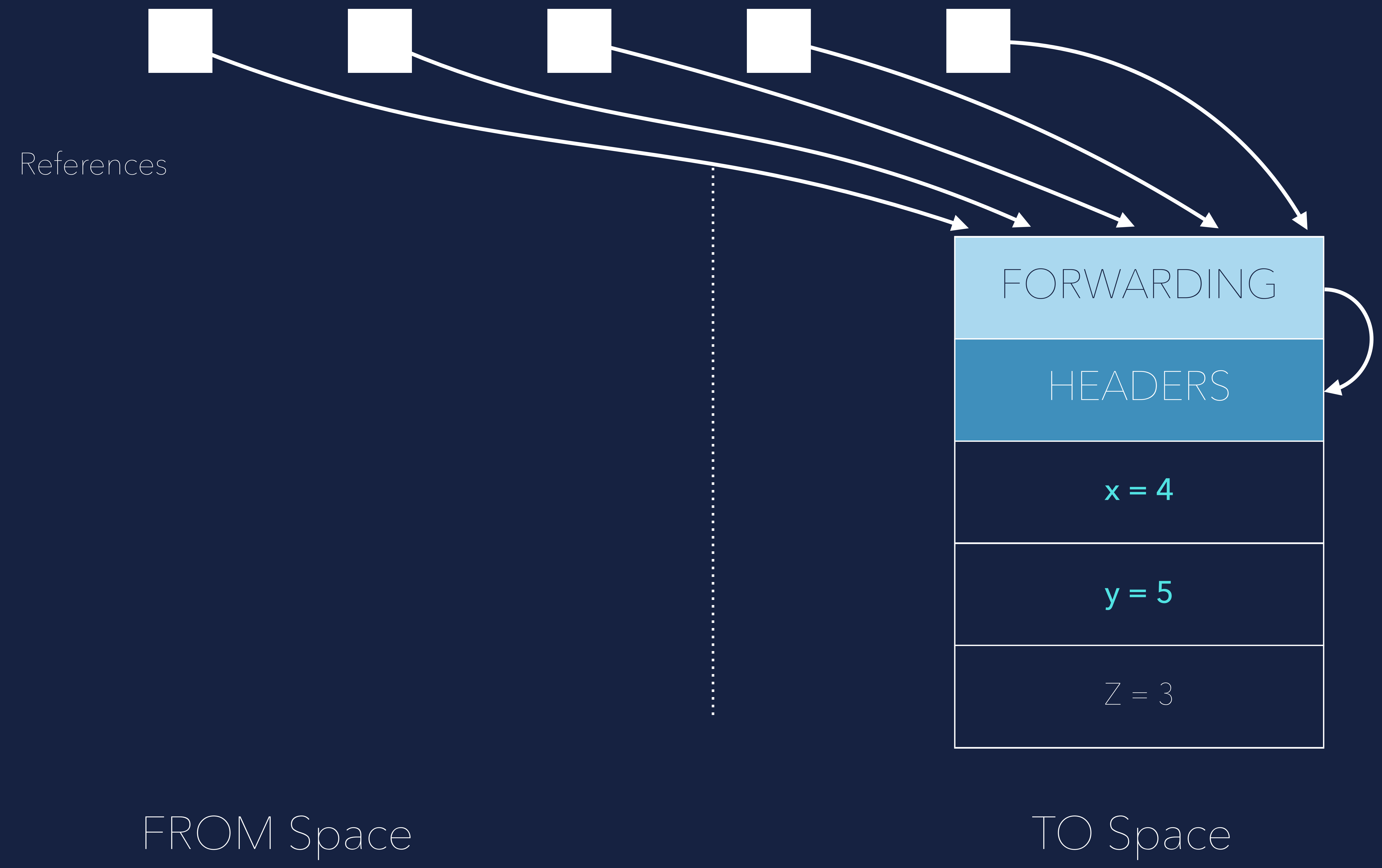
Concurrent copying



All references are updated

CONCURRENCY IS HARD...

Concurrent copying



Remove the old object

COLLECTORS IN THE JVM

SERIAL

SERIAL

AVAILABILITY	ALL JDK'S
PARALLEL	NO
CONCURRENT	NO
GENERATIONAL	YES
HEAP SIZE	SMALL - MEDIUM
PAUSE TIMES	LONGER
THROUGHPUT	LOW
LATENCY	HIGHER
CPU OVERHEAD	LOW (1-5%)

CHOOSE WHEN

- Single core systems with small heap (<4GB)
- No pause time requirements

BEST SUITED FOR

- Single threaded applications
- Development environments
- Microservices on small nodes

OS SUPPORT	🐧	⊞	🍎

JVM SWITCH

```
> java -XX:+UseSerialGC
```

SERIAL

NOTES

- Automatically selected if only a single processor is available
- Automatically selected if the avail. memory less than 1792 MB
- Young Generation algorithm: Copy Collector
- Old Generation algorithm: Mark Sweep Compact

PARALLEL

PARALLEL

AVAILABILITY	ALL JDK'S
PARALLEL	YES
CONCURRENT	NO
GENERATIONAL	YES
HEAP SIZE	MEDIUM - LARGE
PAUSE TIMES	MODERATE
THROUGHPUT	HIGH
LATENCY	LOWER
CPU OVERHEAD	MODERATE (5-10%)

CHOOSE WHEN

- Multi-core systems with small heap (<4GB)
- Peak performance is needed without pause time requirements

BEST SUITED FOR

- Batch processing
- Scientific computing
- Data analysis

OS SUPPORT

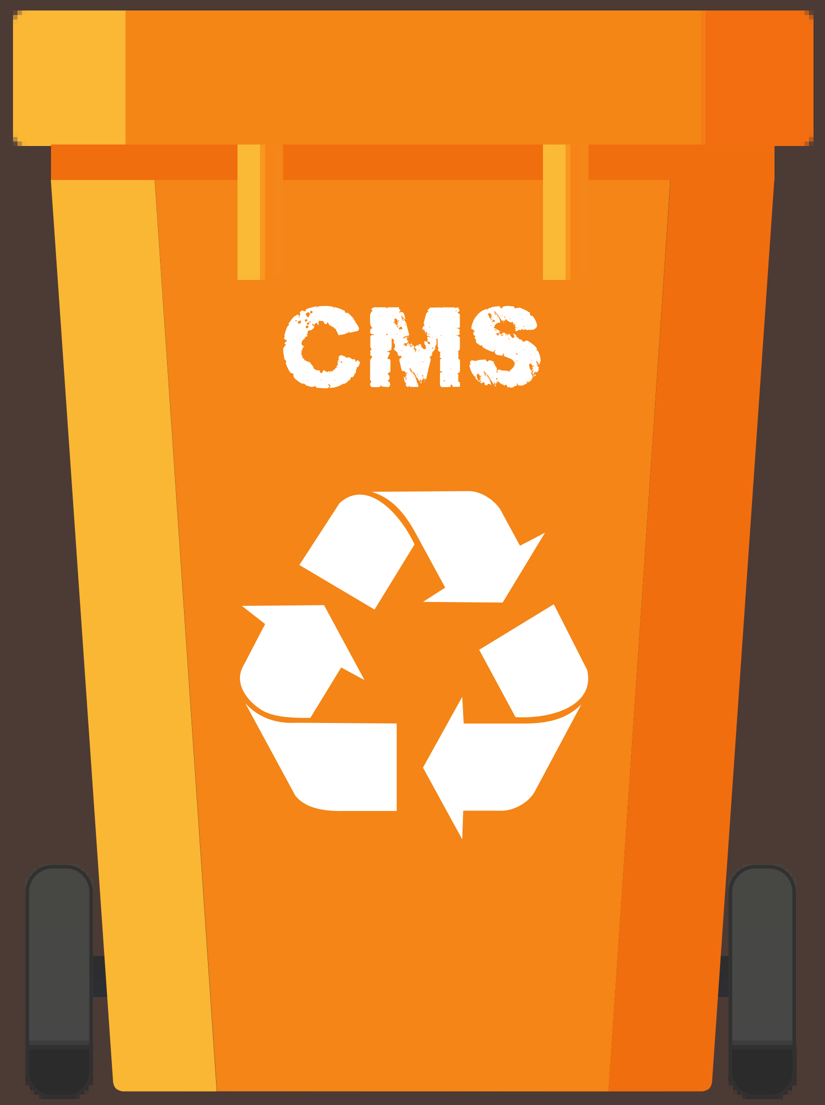
JVM SWITCH

```
> java -XX:+UseParallelOldGC
```

PARALLEL

NOTES

- Default garbage collector from JDK 5 to JDK 7
- Young Generation algorithm: Copy Collector
- Old Generation algorithm: Mark Sweep Compact



CMS

Concurrent Mark and Sweep

DEPRECATED

CMS

AVAILABILITY	JDK 1.4 - 13
PARALLEL	YES
CONCURRENT	PARTIALLY
GENERATIONAL	YES
HEAP SIZE	MEDIUM - LARGE
PAUSE TIMES	MODERATE
THROUGHPUT	MODERATE
LATENCY	MODERATE
CPU OVERHEAD	MODERATE (5-15%)

CHOOSE WHEN

- Response time is more important than throughput
- Pause time must be kept shorter than 1 sec

BEST SUITED FOR

- Web applications
- Mediums sized enterprise systems

OS SUPPORT	

JVM SWITCH

```
> java -XX:+UseConcMarkSweepGC
```

CMS

NOTES

- Deprecated as of JDK 9
- Removed from JDK 14
- Young Generation algorithm: Copy Collector
- Old Generation algorithm: Concurrent Mark and Sweep
- Full GC algorithm: Mark Sweep Compact

G1

Garbage First

Heap-Layout

Region size 1 - 32 MB

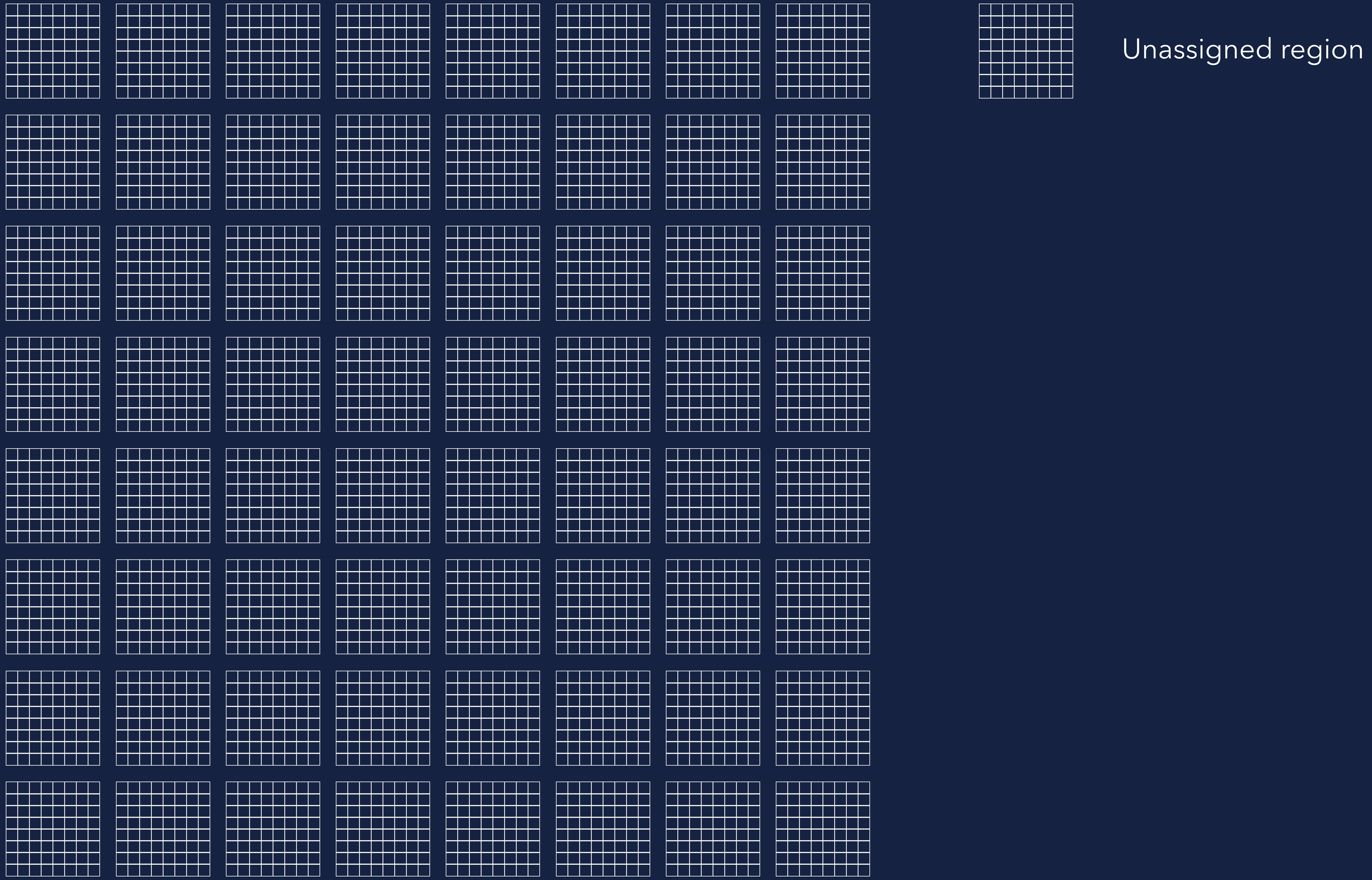
Max no. of region <= 2048

Heap		Region
< 4 GB	-	1 MB
< 8 GB	-	2 MB
< 16 GB	-	4 MB
< 32 GB	-	8 MB
< 64 GB	-	16 MB
> 64 GB	-	32 MB

Example 8GB Heap:

8 GB Heap = 8192 MB

8192 MB / 2048 = 4 MB region size



Unassigned region

G1

Heap-Layout

Region size 1 - 32 MB

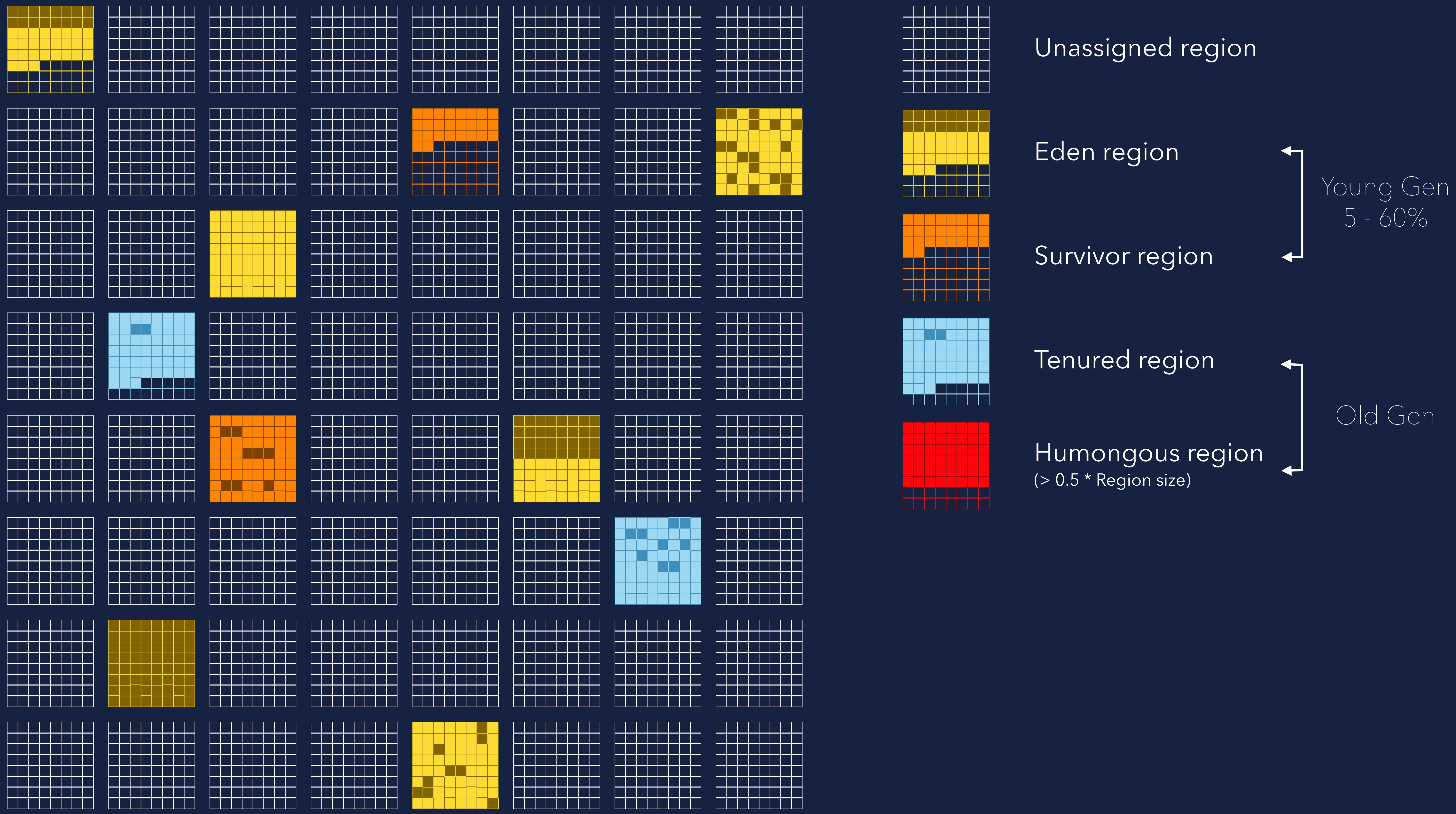
Max no. of region <= 2048

Heap	Region
< 4 GB	1 MB
< 8 GB	2 MB
< 16 GB	4 MB
< 32 GB	8 MB
< 64 GB	16 MB
> 64 GB	32 MB

Example 8GB Heap:

8 GB Heap = 8192 MB

8192 MB / 2048 = 4 MB region size



Unassigned region

Eden region

Survivor region

Young Gen 5 - 60%

Tenured region

Humongous region (> 0.5 * Region size)

Old Gen

G1

Heap-Layout

Region size 1 - 32 MB

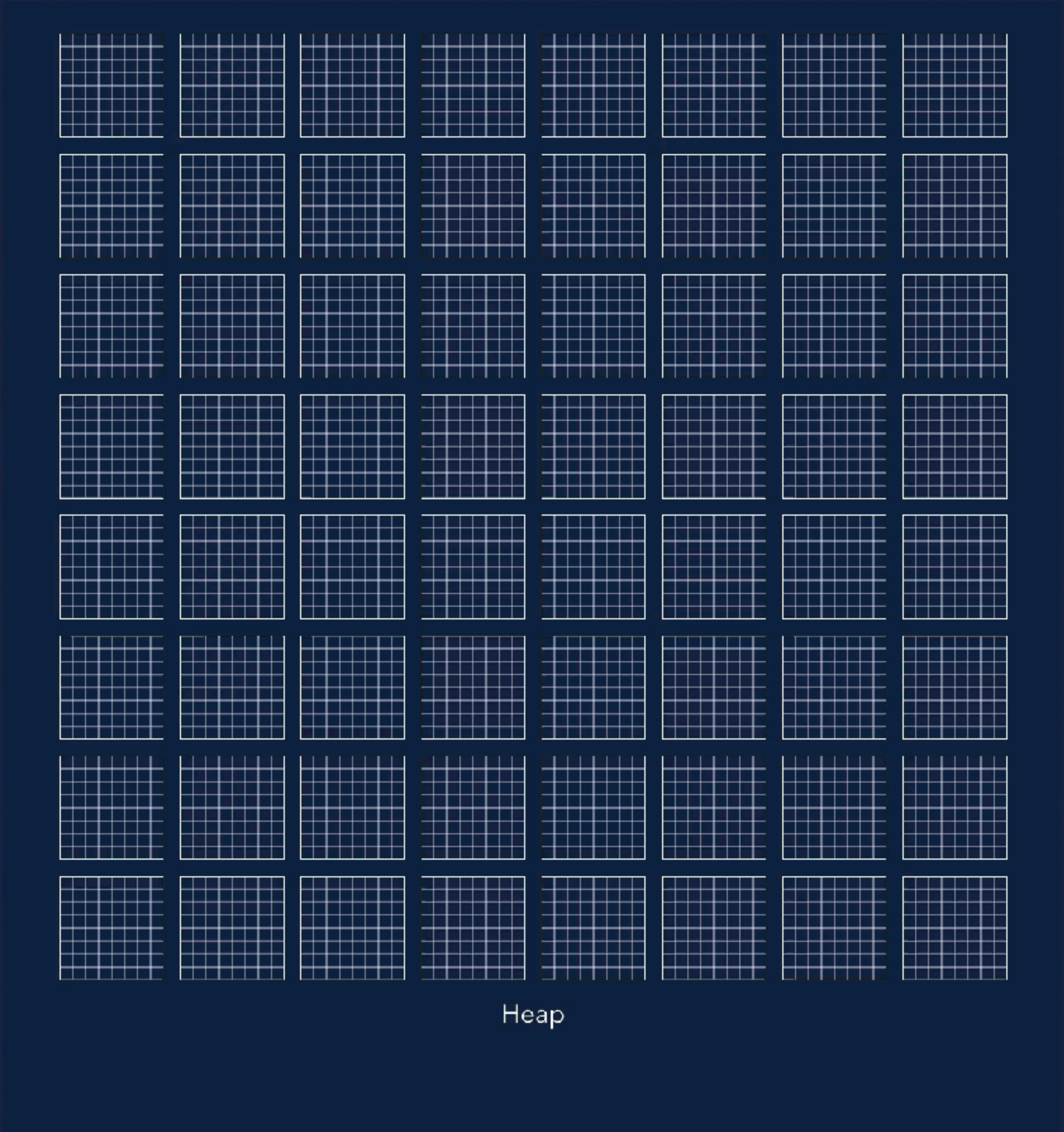
Max no. of region <= 2048

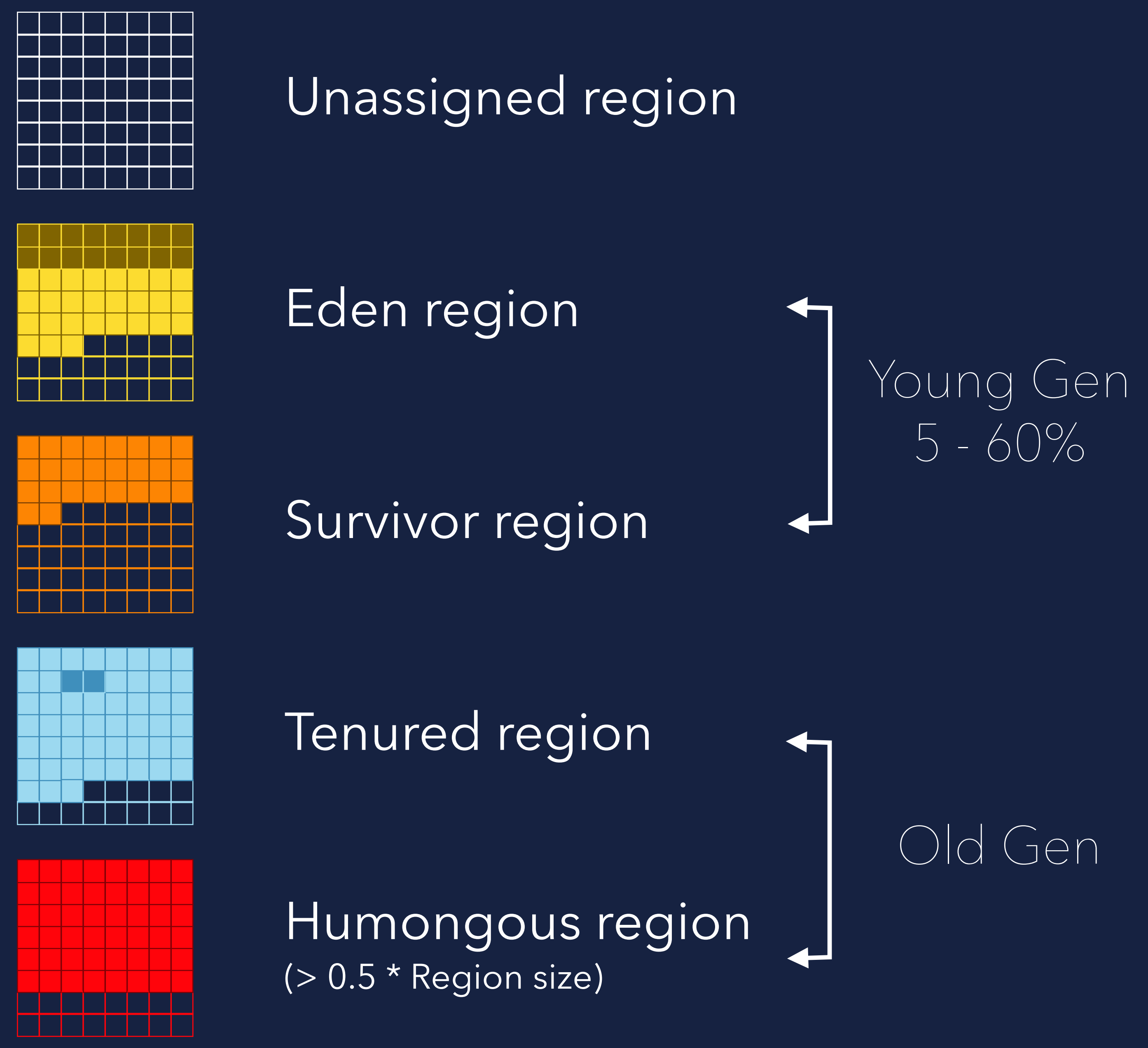
Heap		Region
< 4 GB	-	1 MB
< 8 GB	-	2 MB
< 16 GB	-	4 MB
< 32 GB	-	8 MB
< 64 GB	-	16 MB
> 64 GB	-	32 MB

Example 8GB Heap:

8 GB Heap = 8192 MB

8192 MB / 2048 = 4 MB region size





Example:
6 Eden Regions
3 Survivor Regions

2 Regions with most garbage will be collected/promoted

azul

G1

AVAILABILITY	JDK 7U4+
PARALLEL	YES
CONCURRENT	PARTIALLY
GENERATIONAL	YES
HEAP SIZE	MEDIUM - LARGE
PAUSE TIMES	SHORT - MEDIUM
THROUGHPUT	HIGH
LATENCY	LOWER
CPU OVERHEAD	MODERATE (5-15%)

CHOOSE WHEN

- Response time is more important than throughput
- Pause time must be kept shorter than 1 sec

BEST SUITED FOR

- Mixed workloads
- Large sized enterprise systems
- Responsive in medium to large heaps

OS SUPPORT 🐧 ⊞

JVM SWITCH

```
> java -XX:+UseG1GC
```

G1

NOTES

- Default collector from JDK 9 onwards
- Young Generation algorithm: Evacuating Collector (Mark and Compact)
- Old Generation algorithm: Concurrent Mark and Compact
- Full GC algorithm: Mark and Compact

EPSILON

EPSILON

AVAILABILITY	JDK 11+
PARALLEL	-
CONCURRENT	-
GENERATIONAL	-
HEAP SIZE	-
PAUSE TIMES	-
THROUGHPUT	-
LATENCY	-
CPU OVERHEAD	VERY LOW

CHOOSE WHEN

- Testing performance or memory pressure
- Highest performance is needed and nearly no garbage is created

BEST SUITED FOR

- Extremely short lived jobs
- Last drop latency improvements
- Last drop throughput improvements

OS SUPPORT	🐧	⊞	🍎

JVM SWITCH

```
> java -XX:+UseEpsilonGC
```

EPSILON

NOTES

- Only in builds of OpenJDK

SHENANDOAH

SHENANDOAH

AVAILABILITY	JDK 11.0.9+
PARALLEL	YES
CONCURRENT	FULLY
GENERATIONAL	NO
HEAP SIZE	MEDIUM - LARGE
PAUSE TIMES	SHORT
THROUGHPUT	VERY HIGH
LATENCY	VERY LOW
CPU OVERHEAD	MODERATE (10-20%)

CHOOSE WHEN

- Response time is a high priority
- Using a very large heap (100GB+)
- Predictable response times needed

BEST SUITED FOR

- Latency sensitive applications
- Large scale systems
- Highly concurrent applications

OS SUPPORT	🐧	⊞	

JVM SWITCH

```
> java -XX:+UseShenandoahGC
```

SHENANDOAH

NOTES

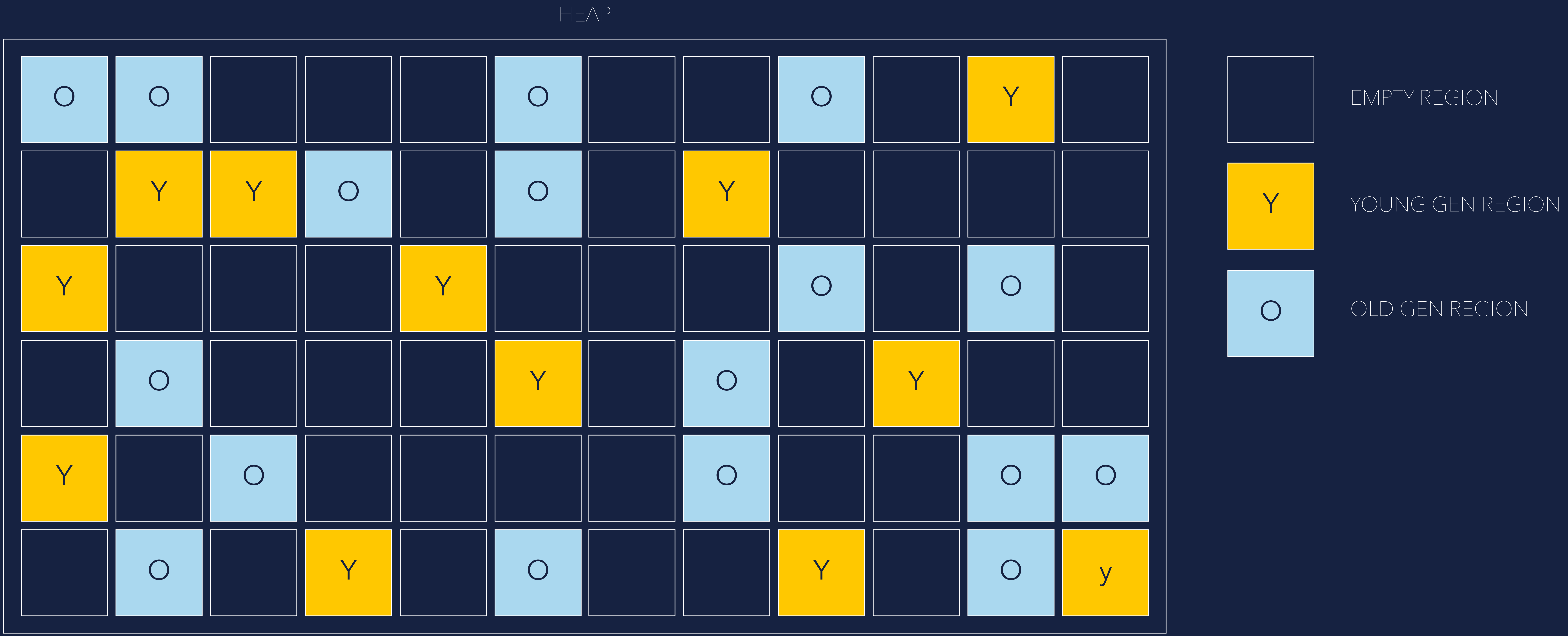
- Not available in Oracle JDK
- A bit reduced throughput due to concurrent GC
- Makes use of new barrier concept, load reference barrier

ZGC

Zero Garbage Collector

ZGC

Heap-Layout



HEAP

O	O				O			O		Y	
	Y	Y	O		O		Y				
Y				Y				O		O	
	O				Y		O		Y		
Y		O					O			O	O
	O		Y		O			Y		O	y

□ EMPTY REGION

Y YOUNG GEN REGION

O OLD GEN REGION

ZGC

AVAILABILITY	JDK 15 / 21+
PARALLEL	YES
CONCURRENT	FULLY
GENERATIONAL	NO / YES
HEAP SIZE	LARGE
PAUSE TIMES	SHORT
THROUGHPUT	VERY HIGH
LATENCY	VERY LOW
CPU OVERHEAD	MODERATE (10-20%)

CHOOSE WHEN

- Response time is a high priority
- Using a very large heap (100GB+)
- Predictable response times needed

BEST SUITED FOR

- Low latency sensitive applications
- Large scale systems
- Highly concurrent applications

OS SUPPORT

JVM SWITCH

```
> java -XX:+UseZGC -XX:+ZGenerational*
```

* Not needed in the future, because generational ZGC will become the default

azul

ZGC

NOTES

- Will become the default collector in the future
- Non-generational version will be deprecated

C4

Concurrent Continues Compacting Collector

C4

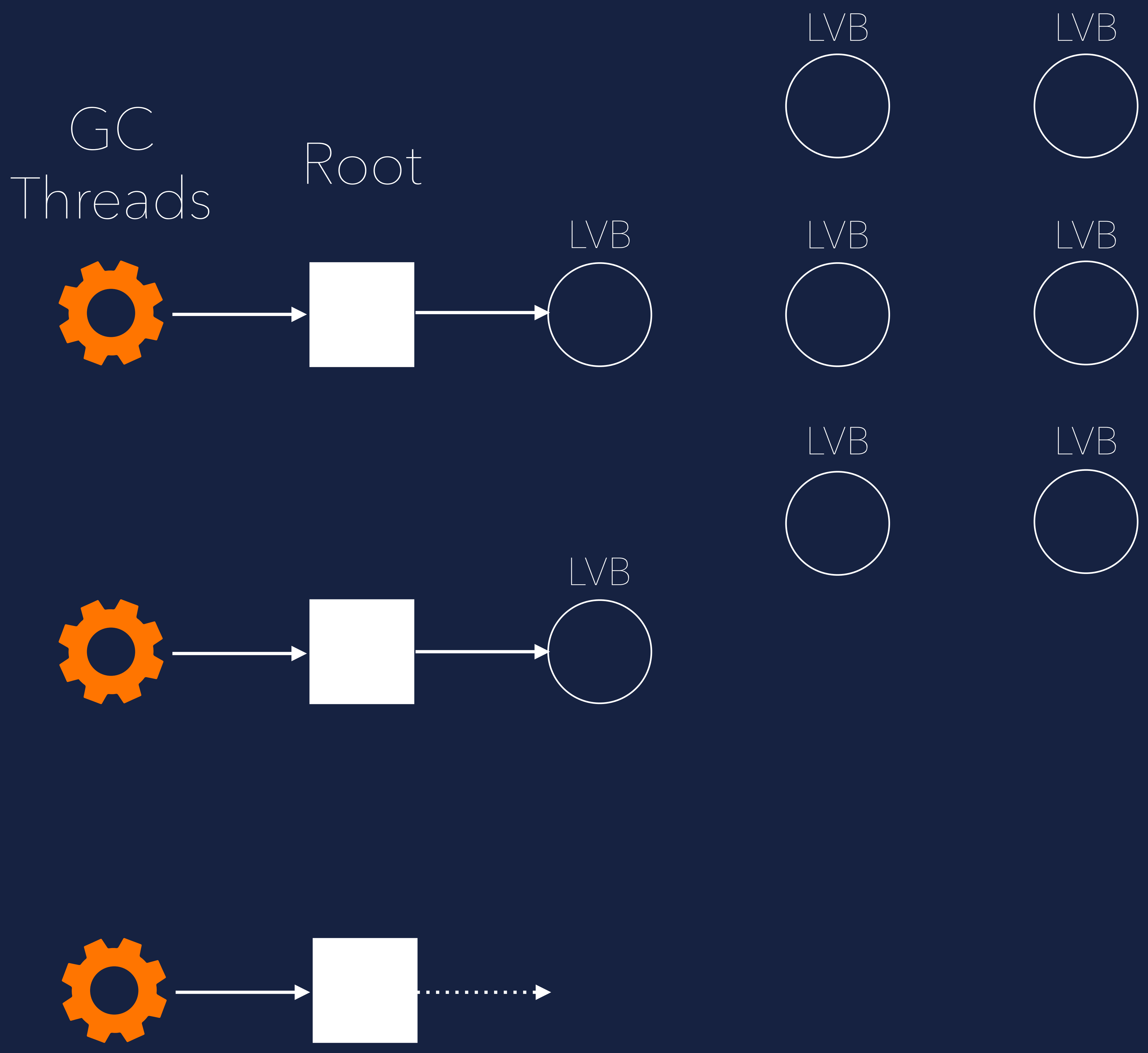
NOTES

- Part of Azul Zing JVM
- Makes use of Loaded Value Barrier everywhere
  (Test + Jump which only takes 1 cpu cycle -> very fast)
- Best performance by using Transparent Huge Pages
  (Normal page size 4kB, THP size 2MB)

MARKING PHASE

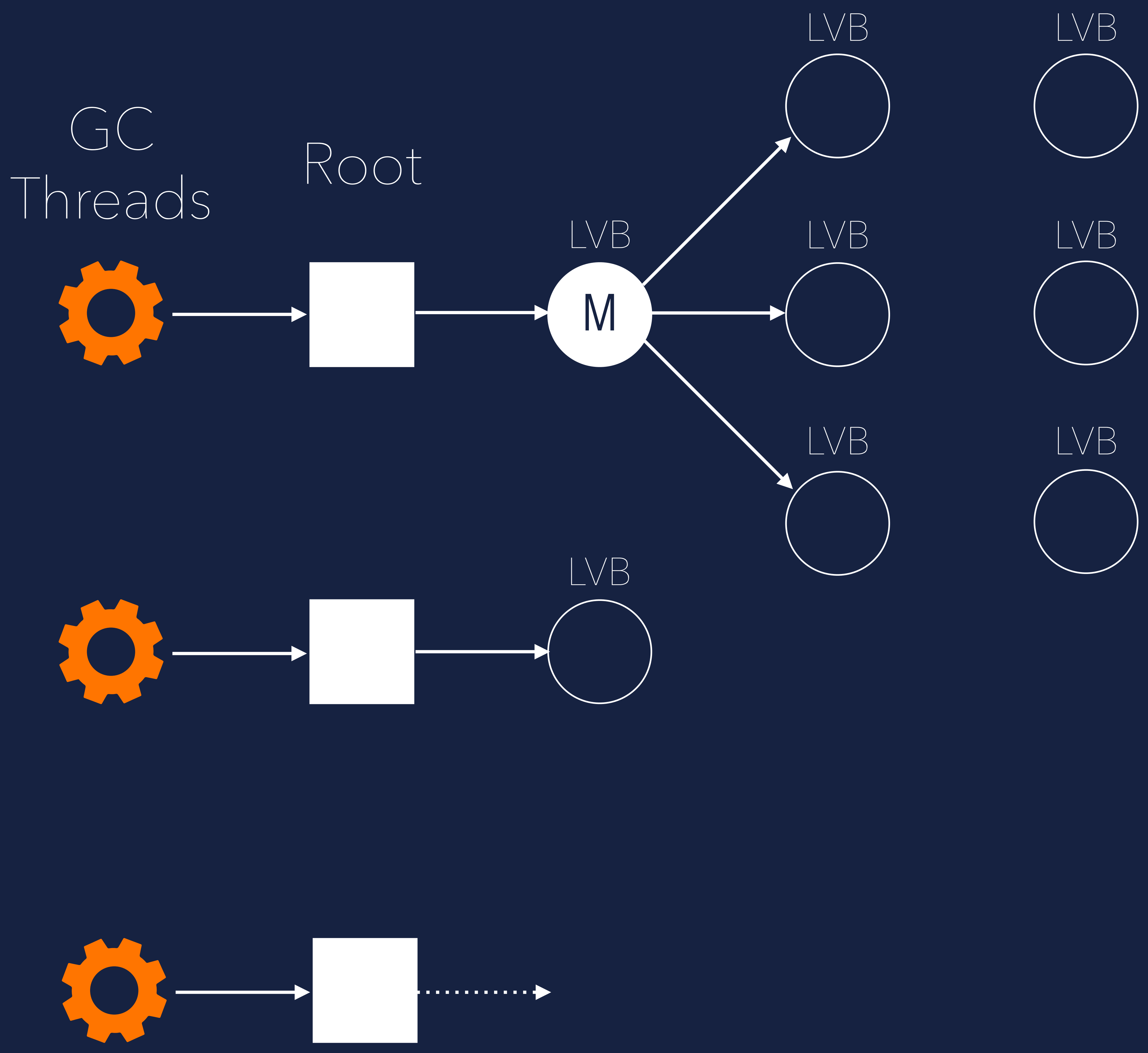
C4

Marking Phase



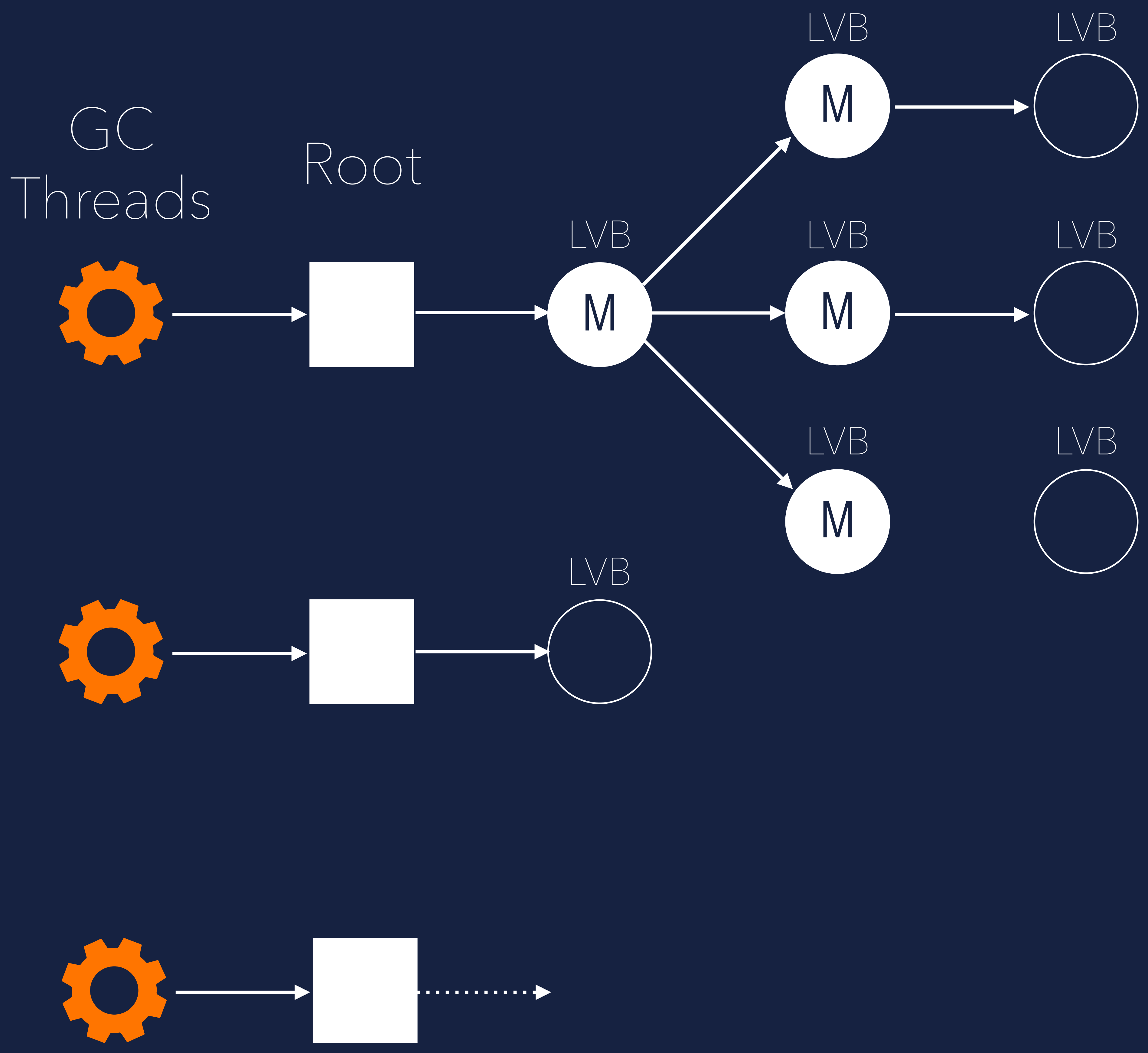
azul

Marking Phase



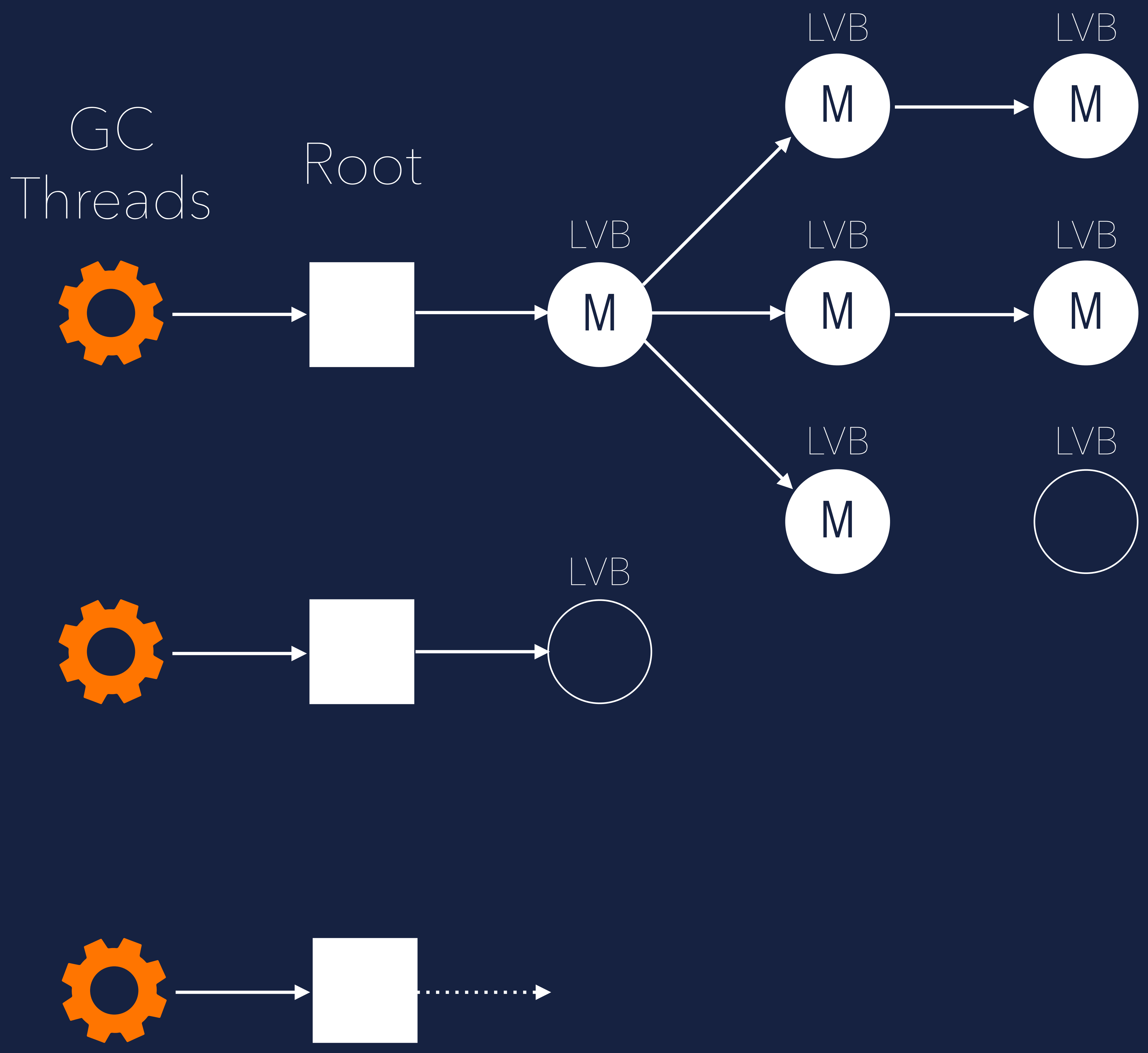
C4

Marking Phase



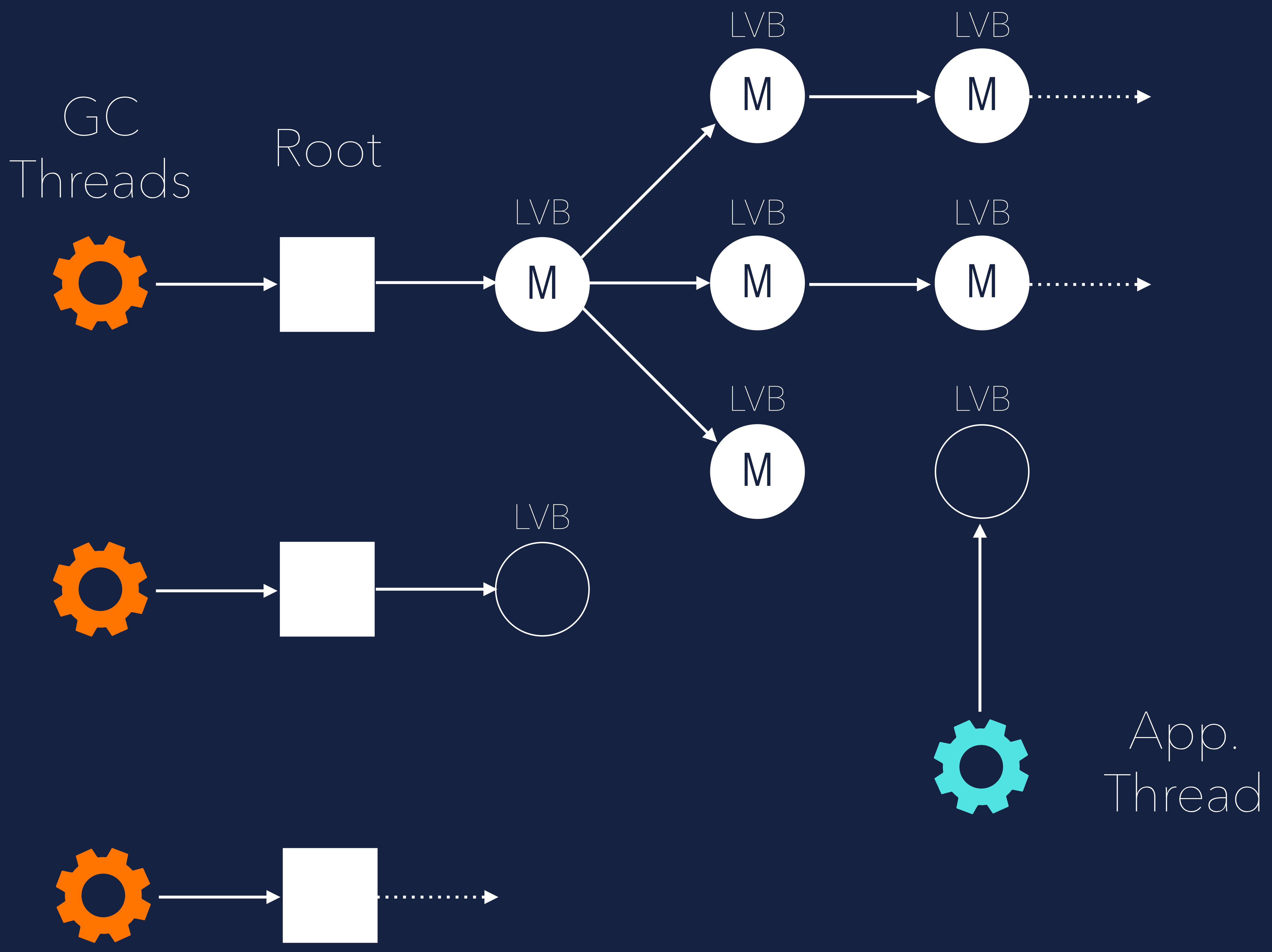
C4

Marking Phase



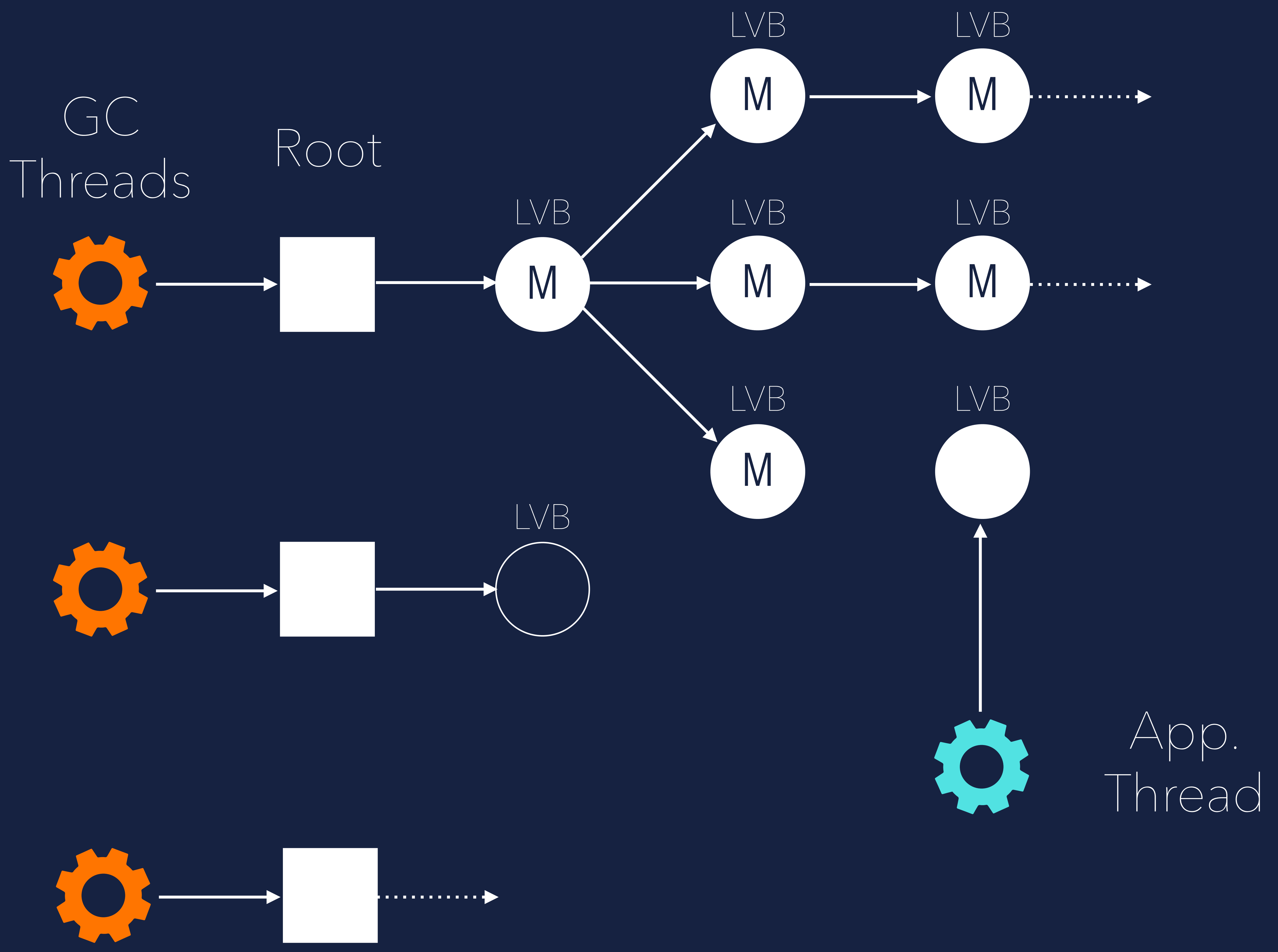
C4

Marking Phase



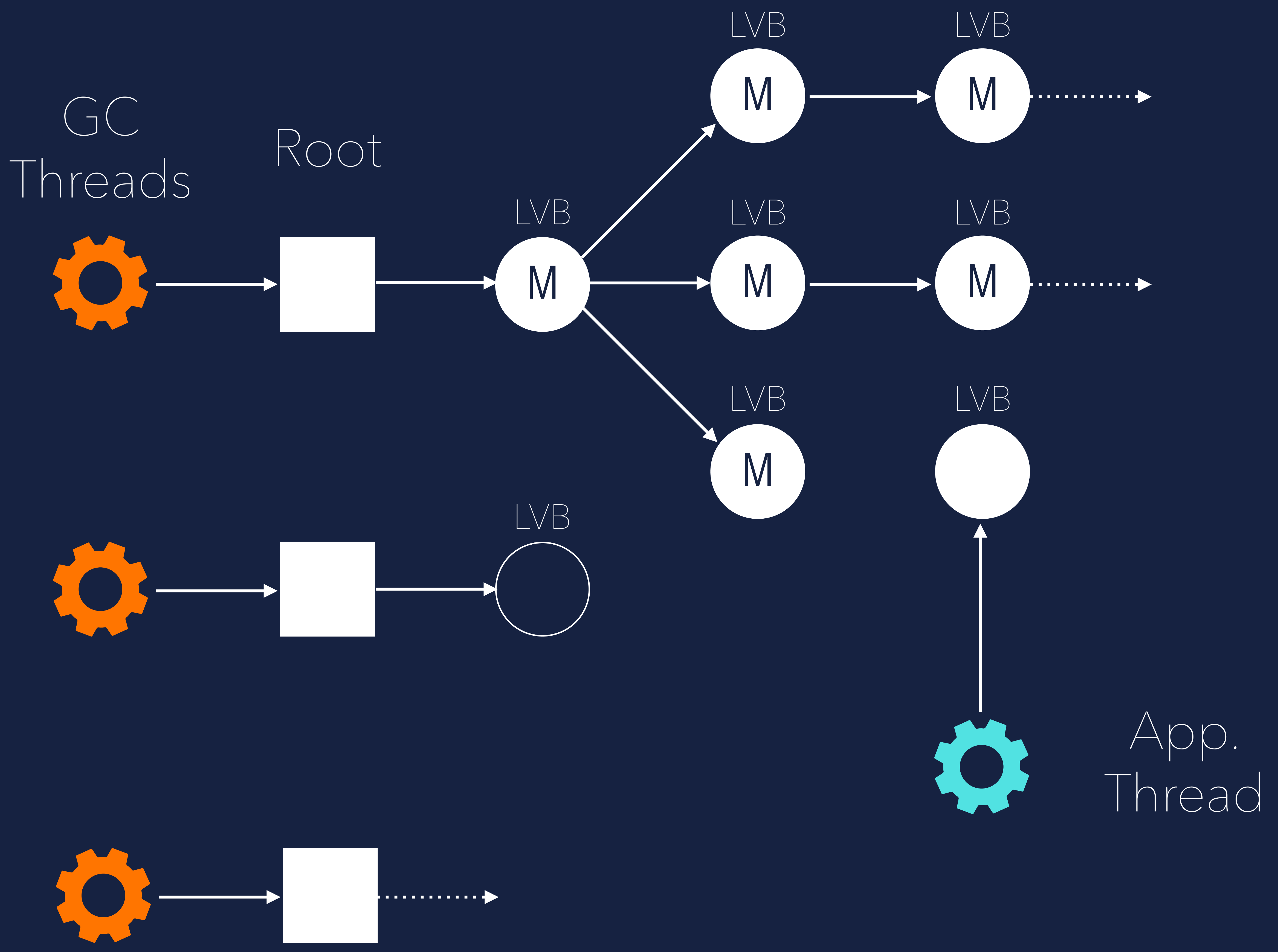
C4

Marking Phase



Trigger LVB

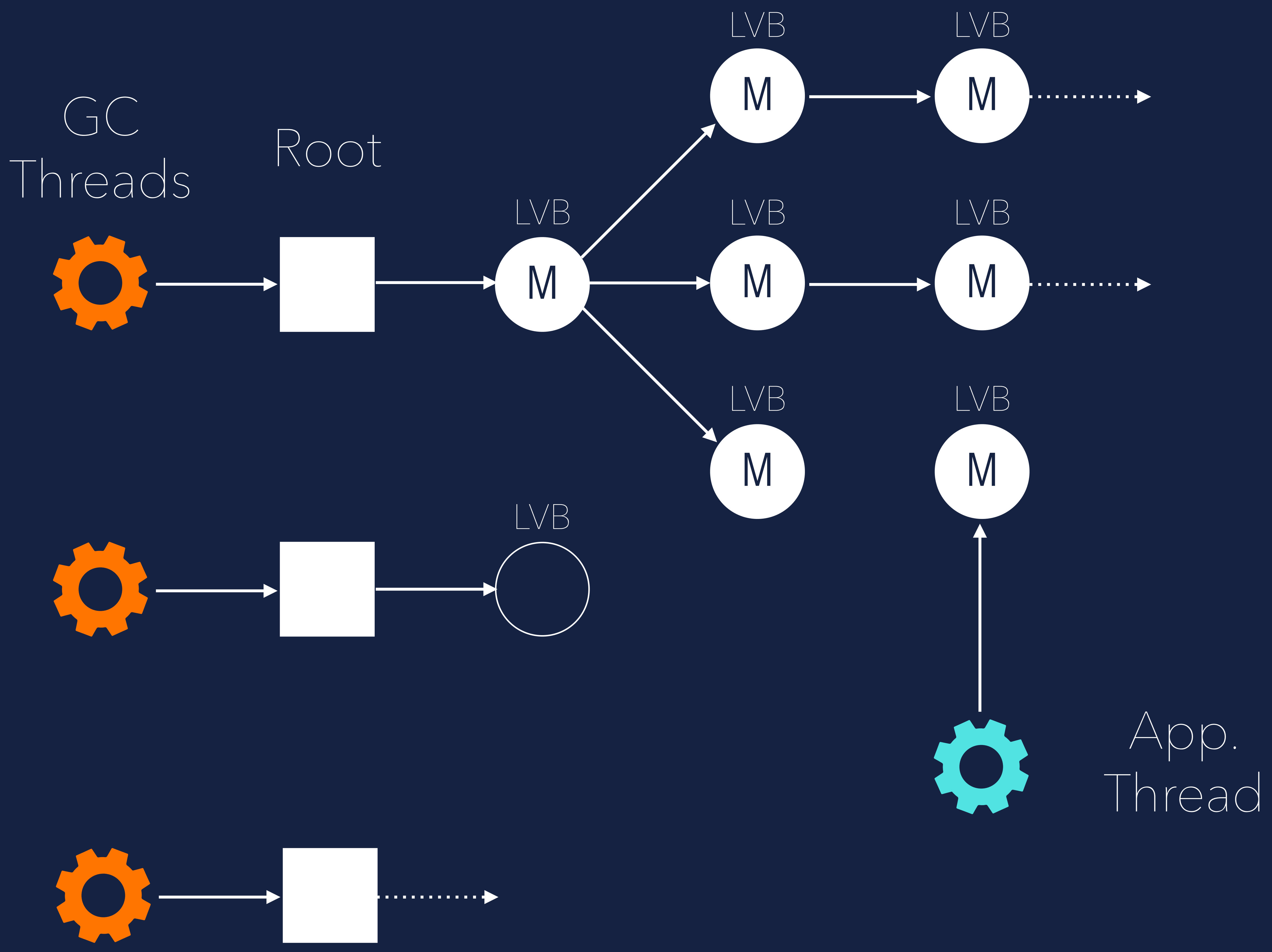
Marking Phase



Test+Jump

C4

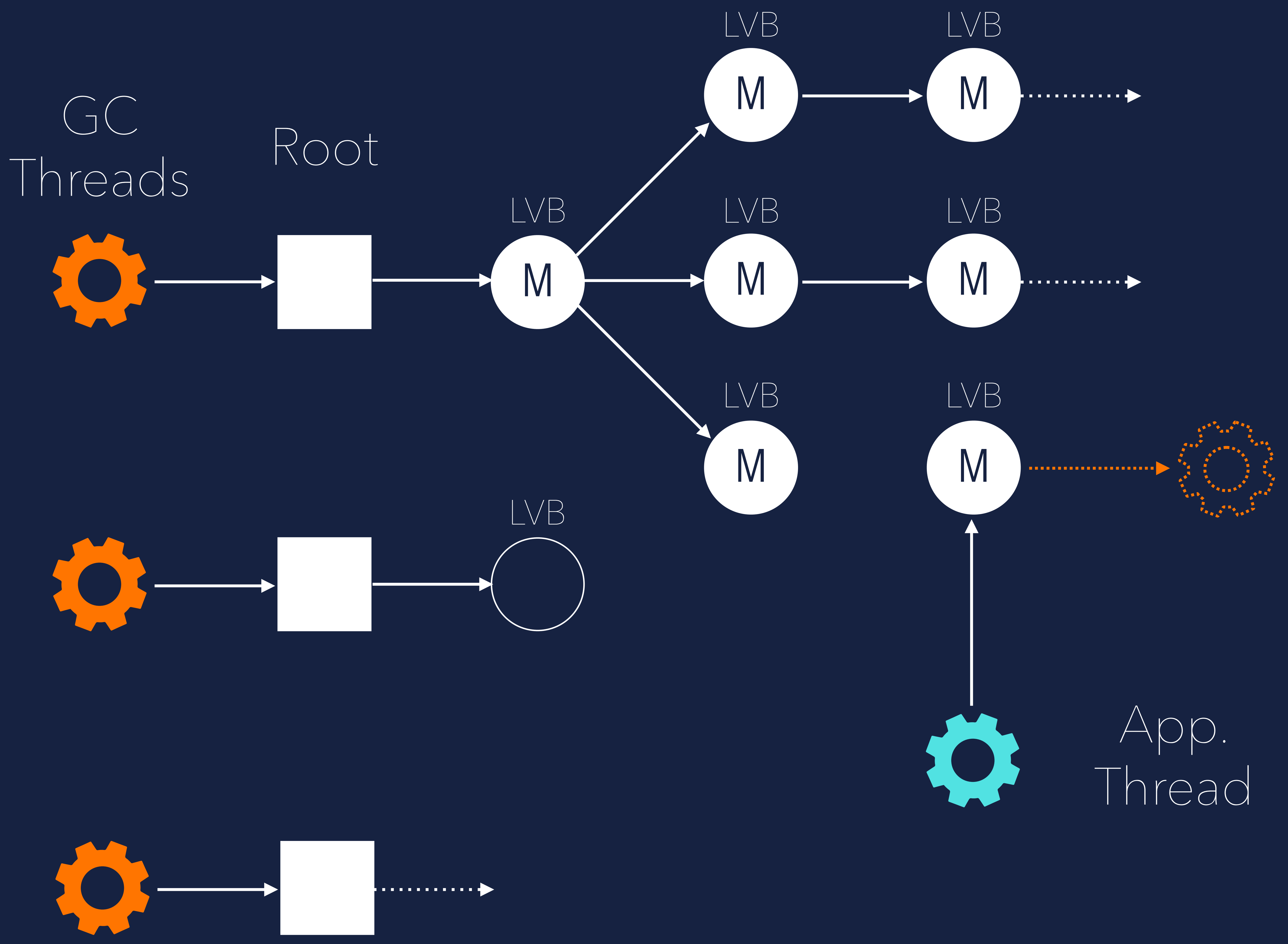
Marking Phase



Mark

C4

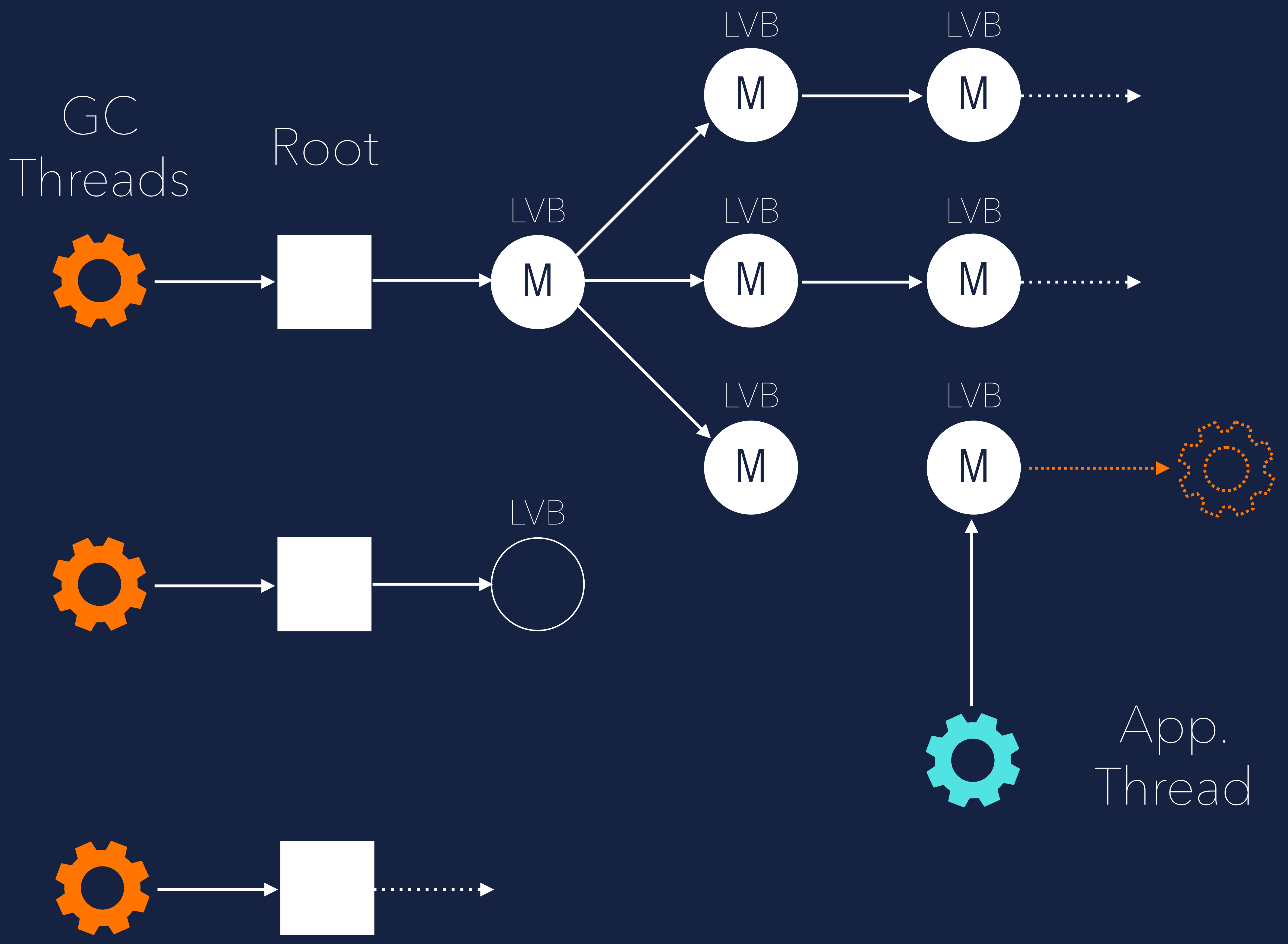
Marking Phase



Hand over
to GC

C4

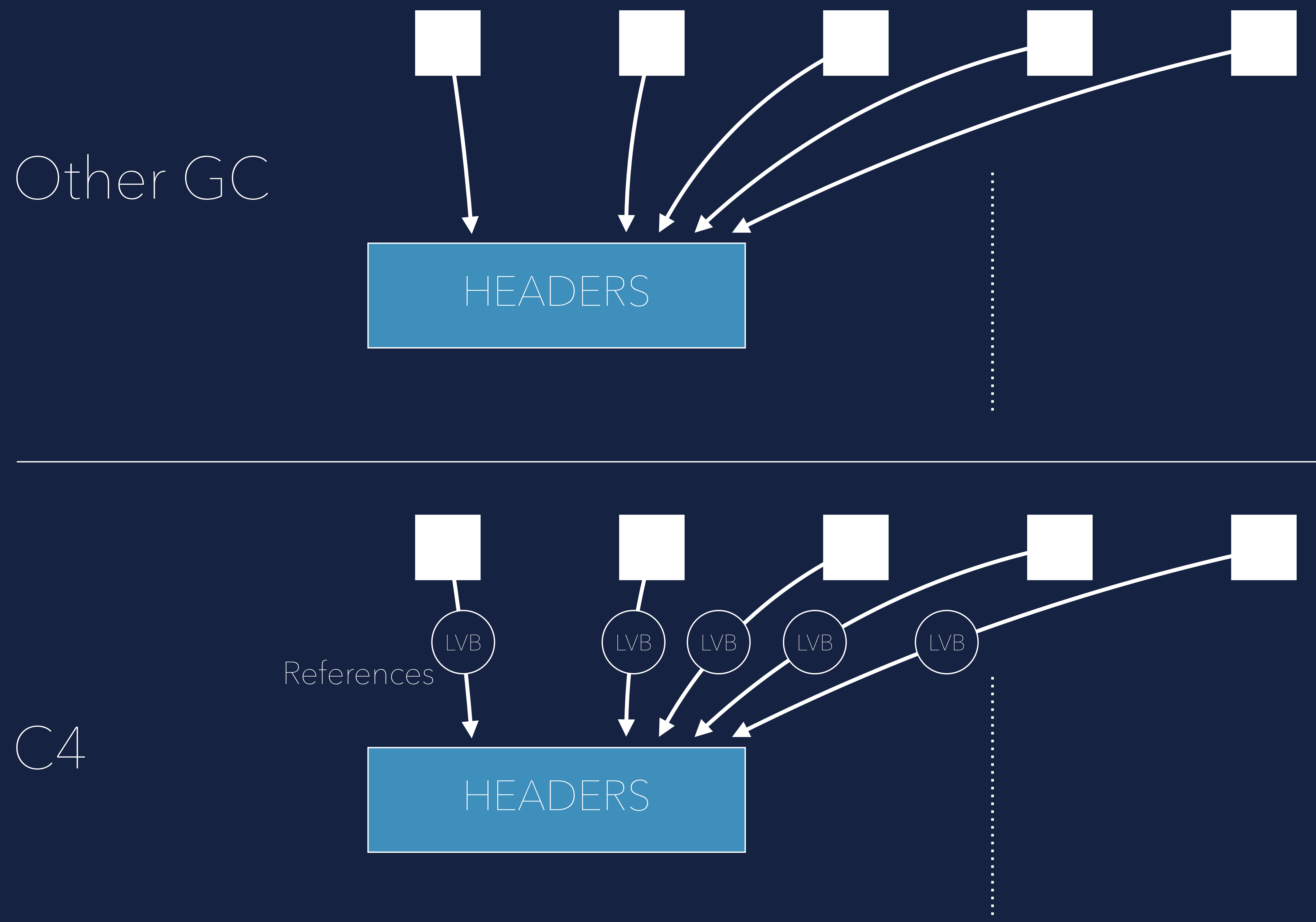
Marking Phase



No need to mark again by GC !

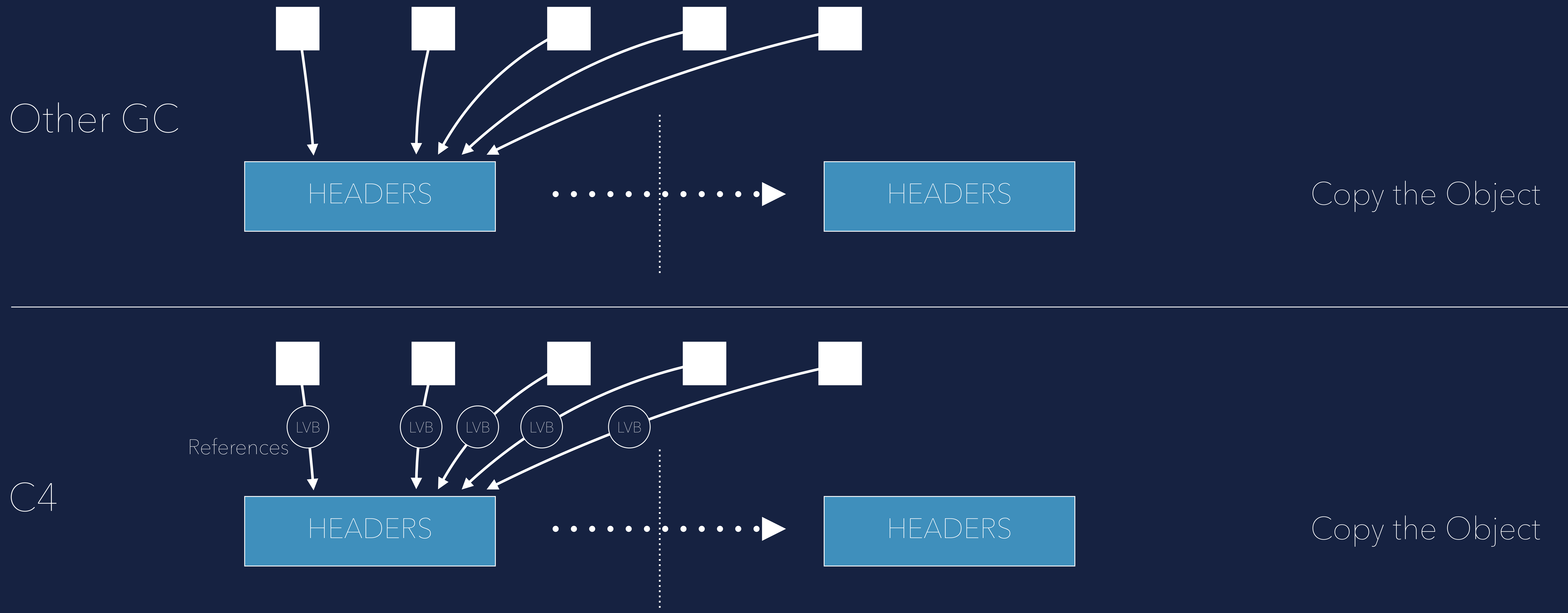
QUICK RELEASE

Remapping phase with quick release



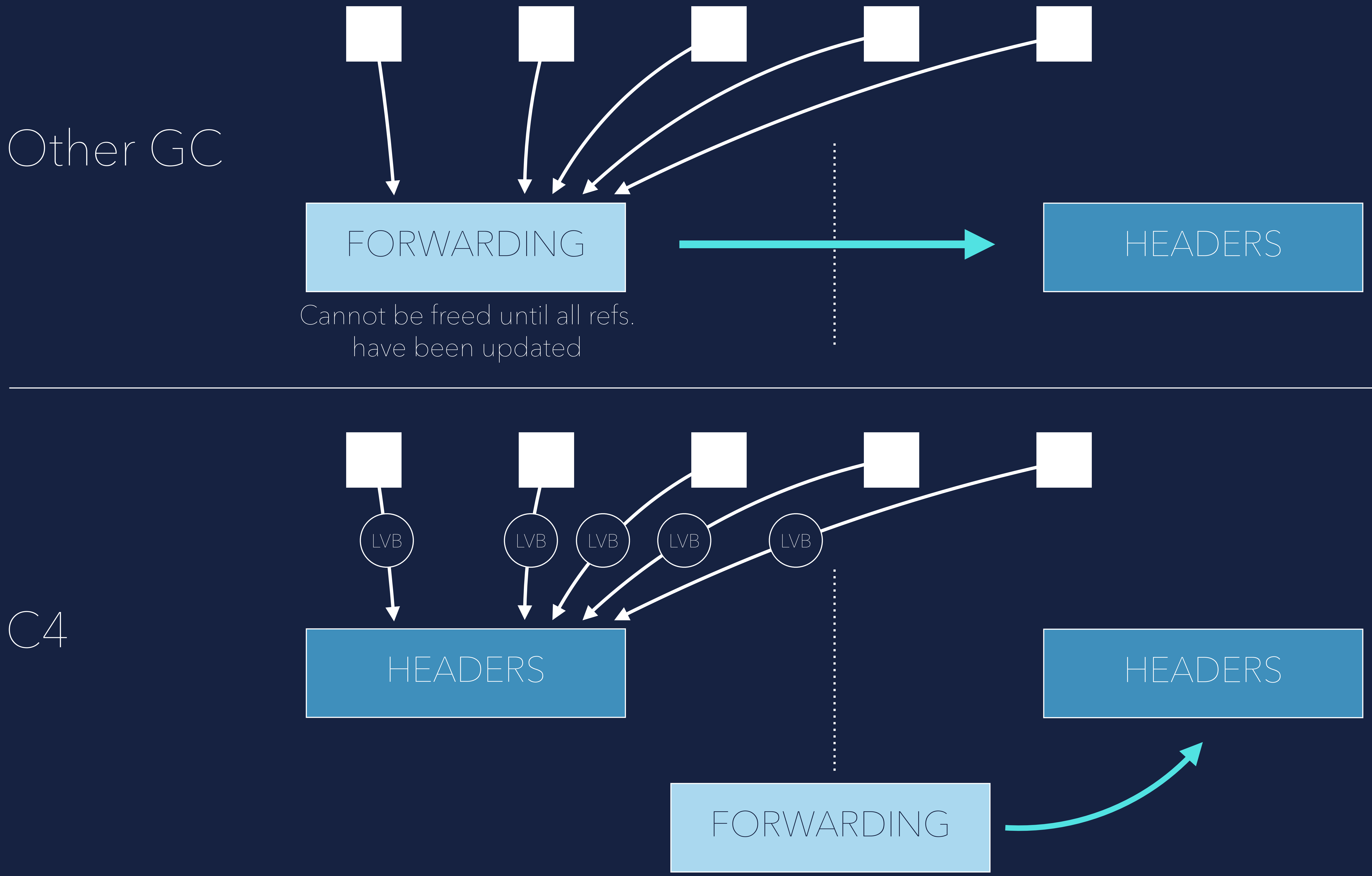
C4

Remapping phase with quick release



C4

Remapping phase with quick release



Cannot be freed until all refs. have been updated

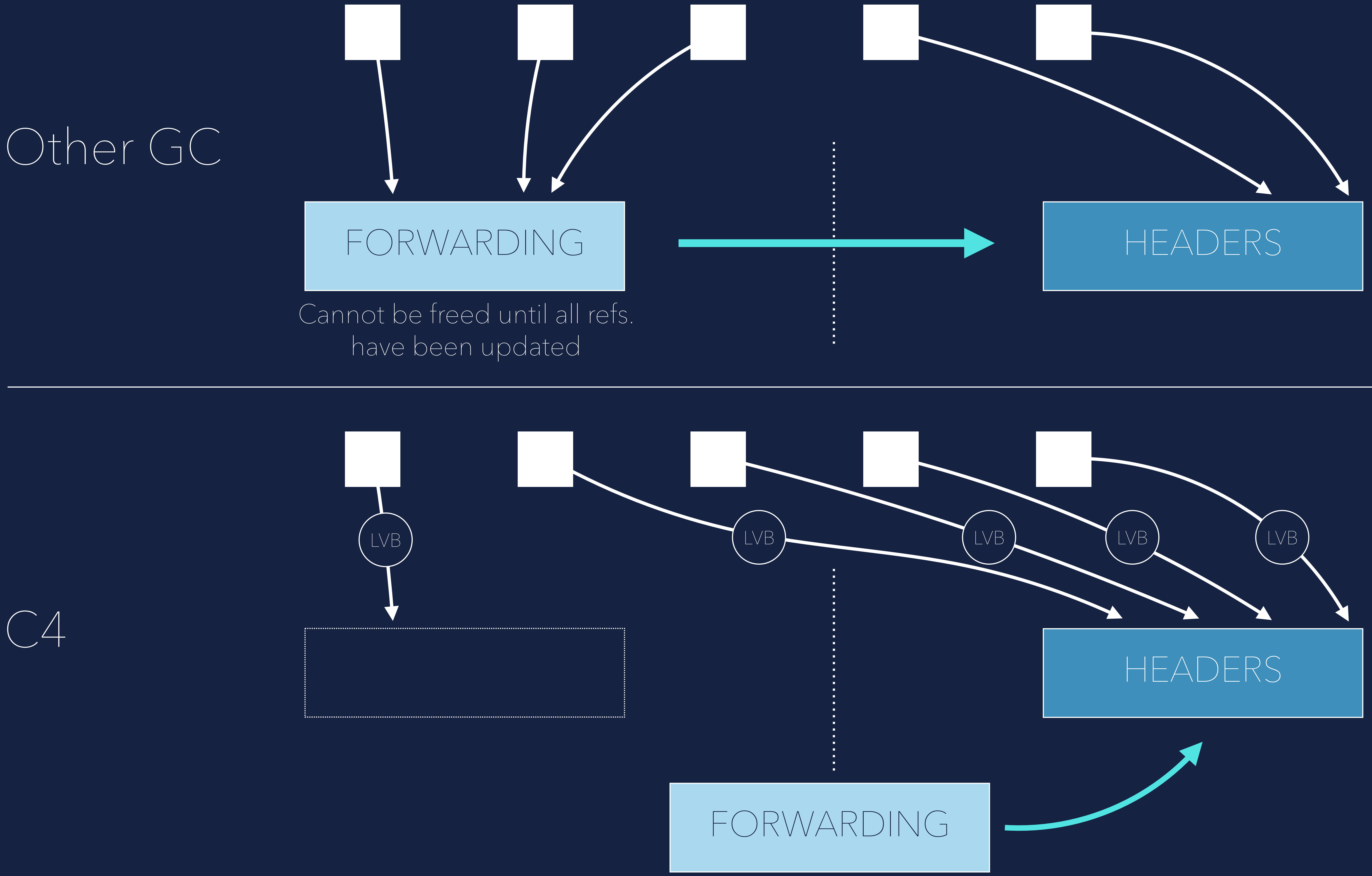
Save fowarding info to Object header

Save forwarding info to Off Heap page

azul

C4

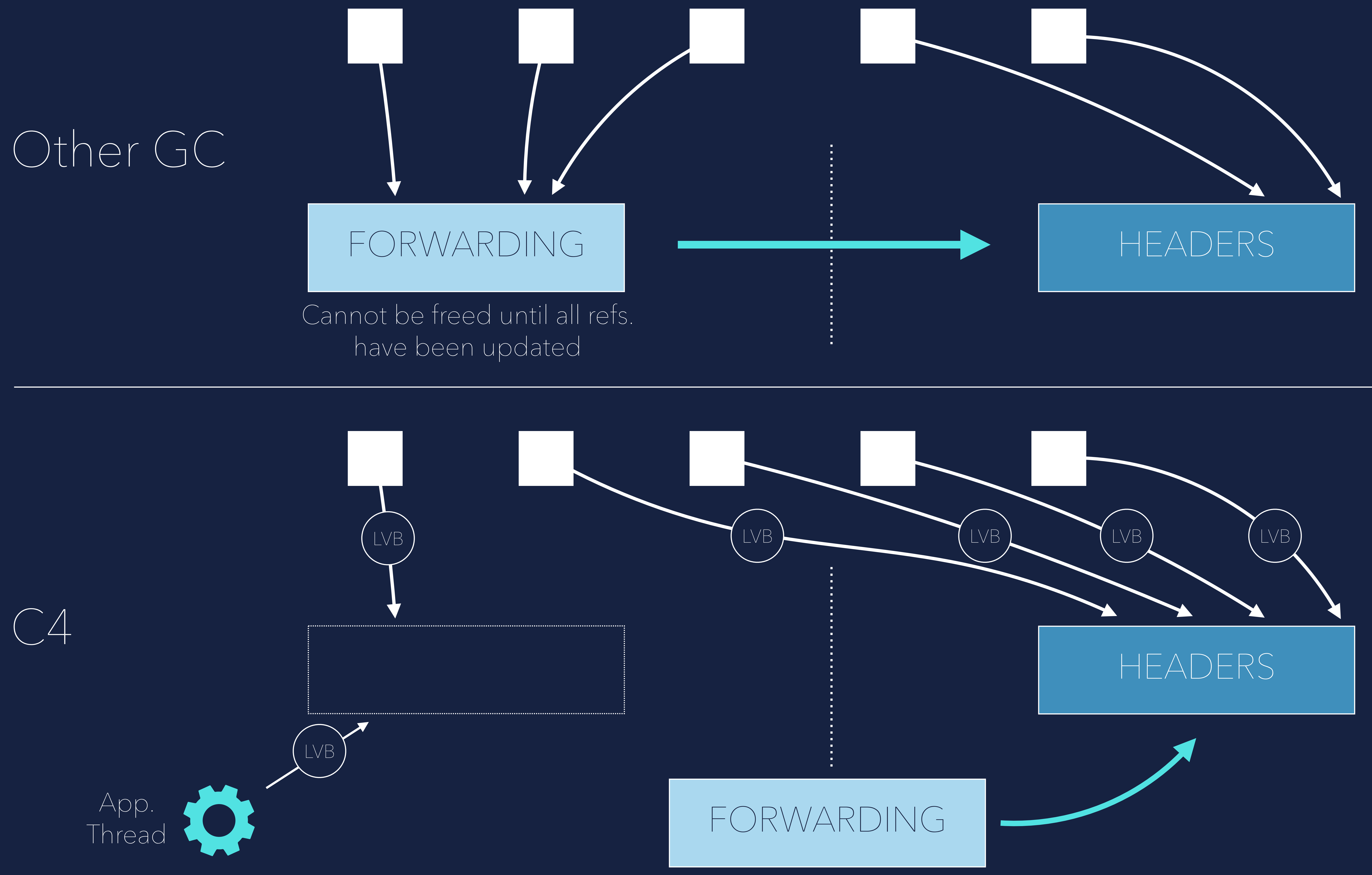
Remapping phase with quick release



Update all references
(Collector walks the heap and replaces all references to new location)

Original reference is freed, and Collector updates references
(No pressure because LVB is self healing)

Remapping phase with quick release

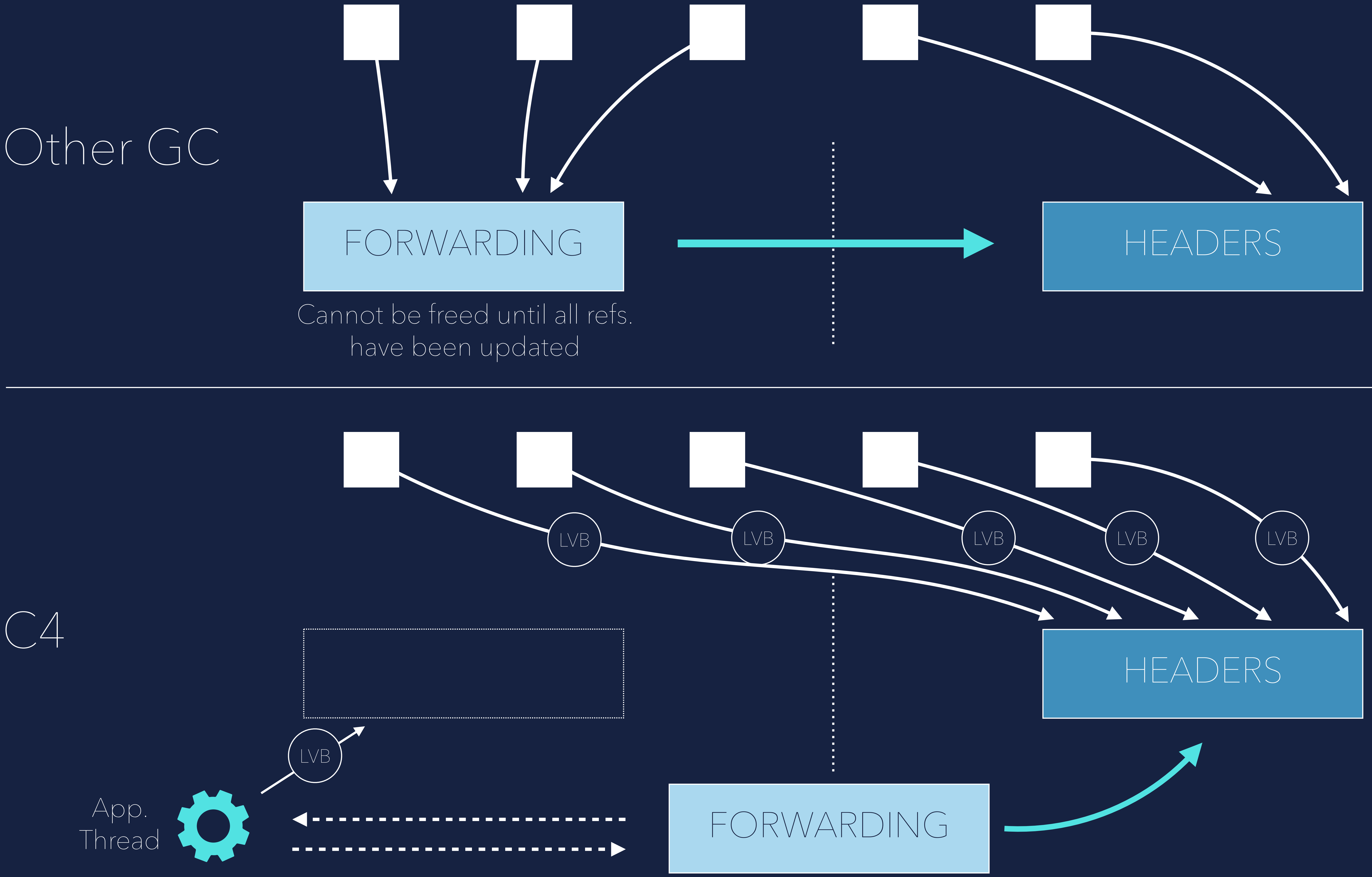


Update all references
(Collector walks the heap and replaces all references to new location)

App Thread triggers LVB
(Test + Jump)

C4

Remapping phase with quick release



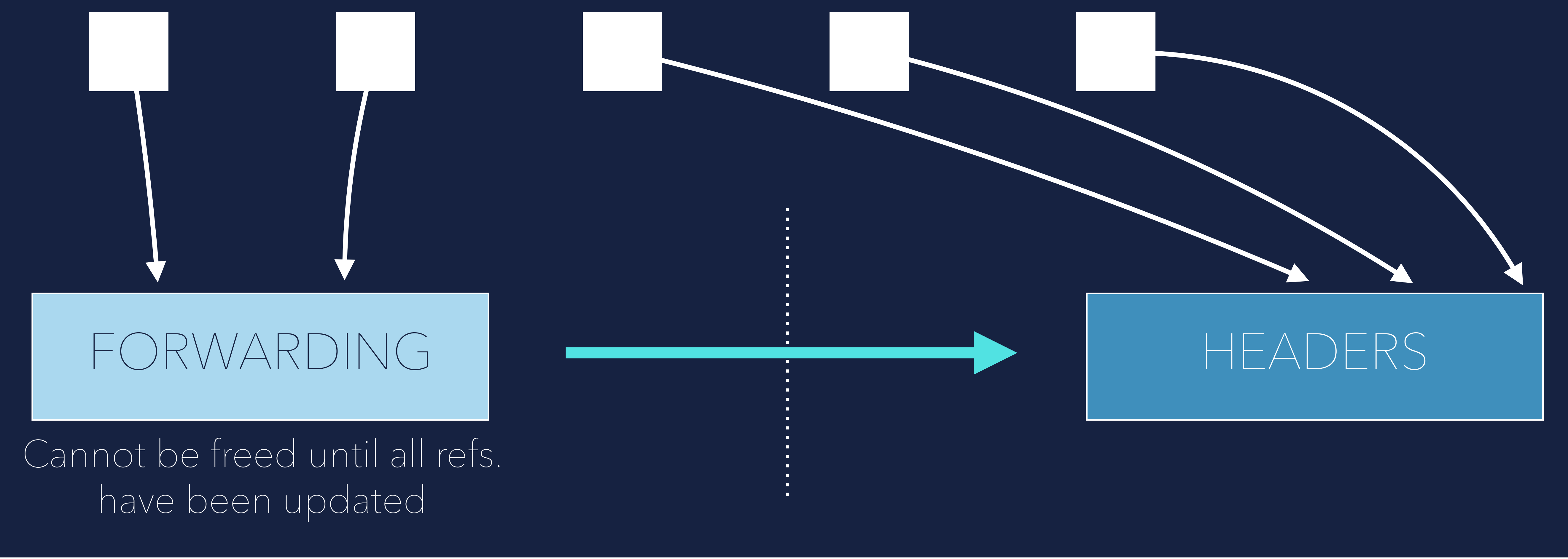
Cannot be freed until all refs. have been updated

Update all references
(Collector walks the heap and replaces all references to new location)

App Thread updates ref
(Using the info from the off heap page)

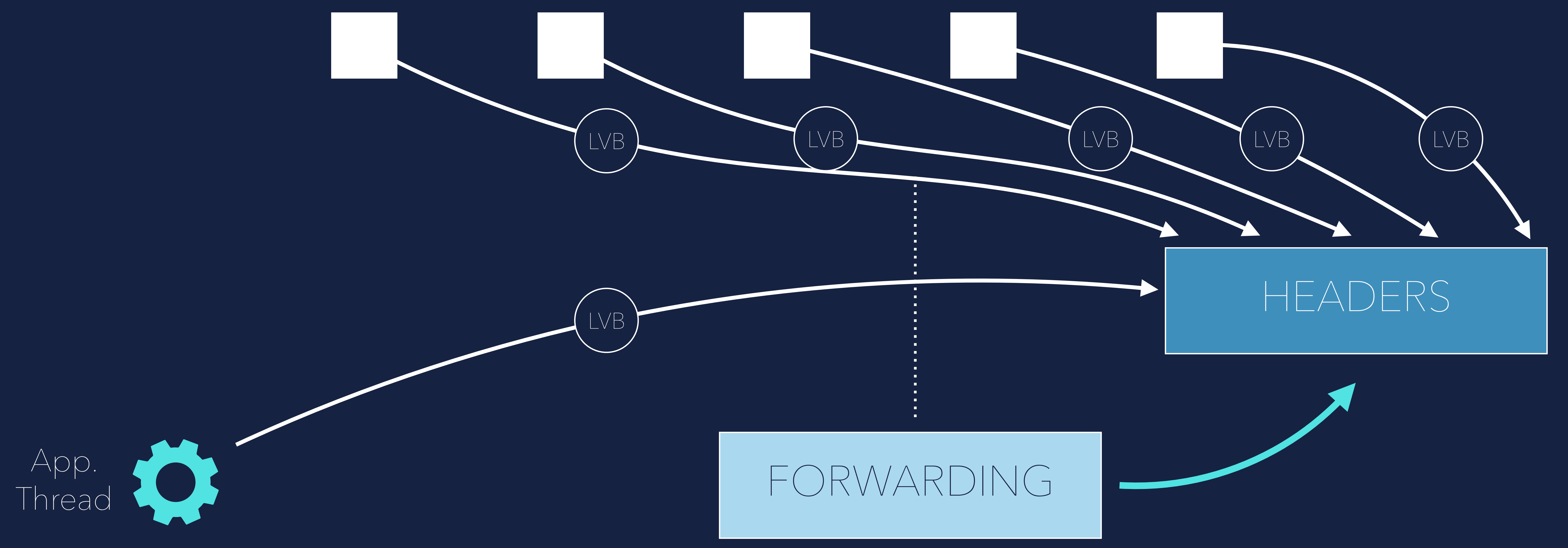
C4

Remapping phase with quick release



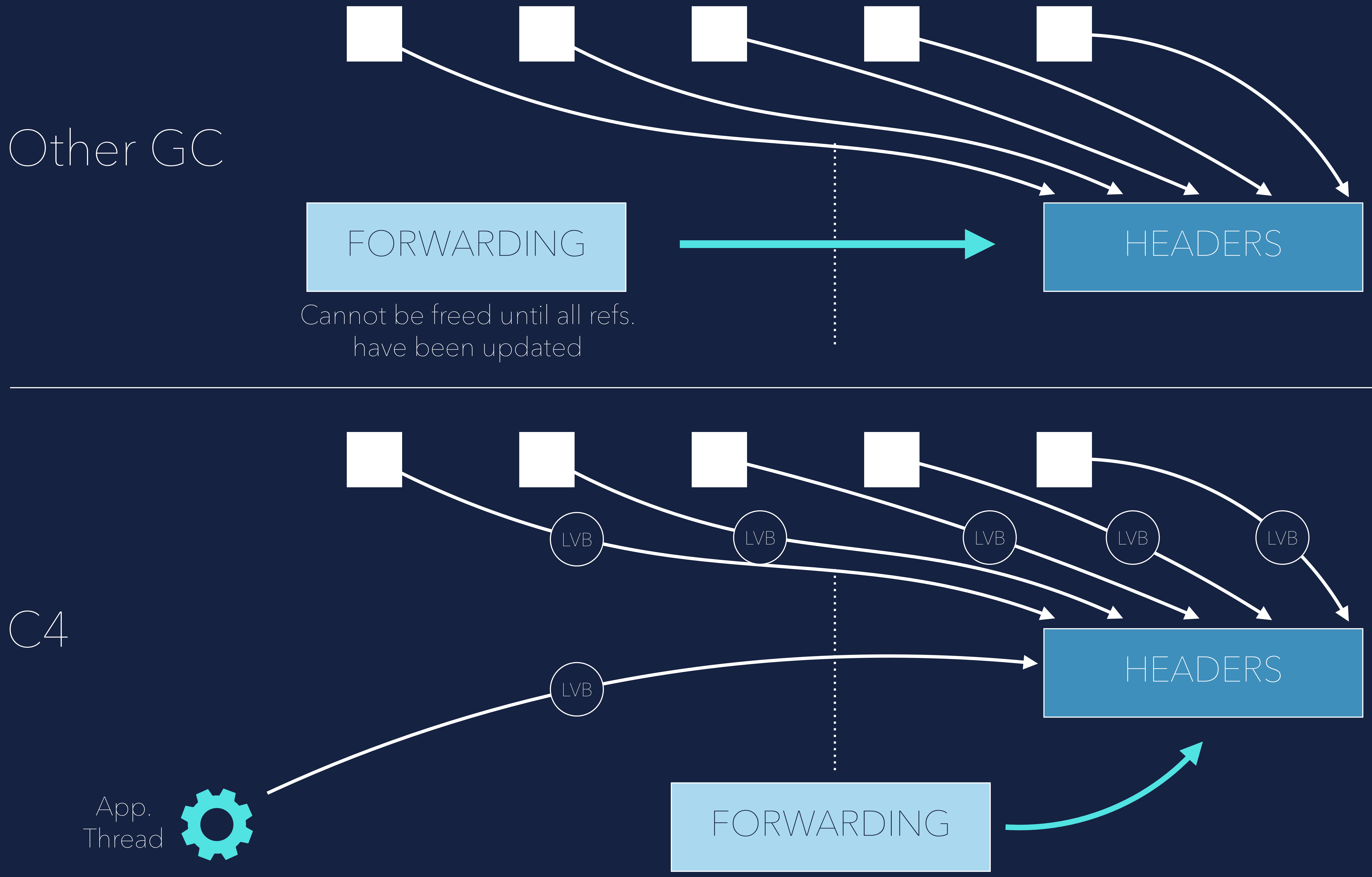
Cannot be freed until all refs. have been updated

Update all references
(Collector walks the heap and replaces all references to new location)



Self healing through LVB

Remapping phase with quick release



Cannot be freed until all refs. have been updated

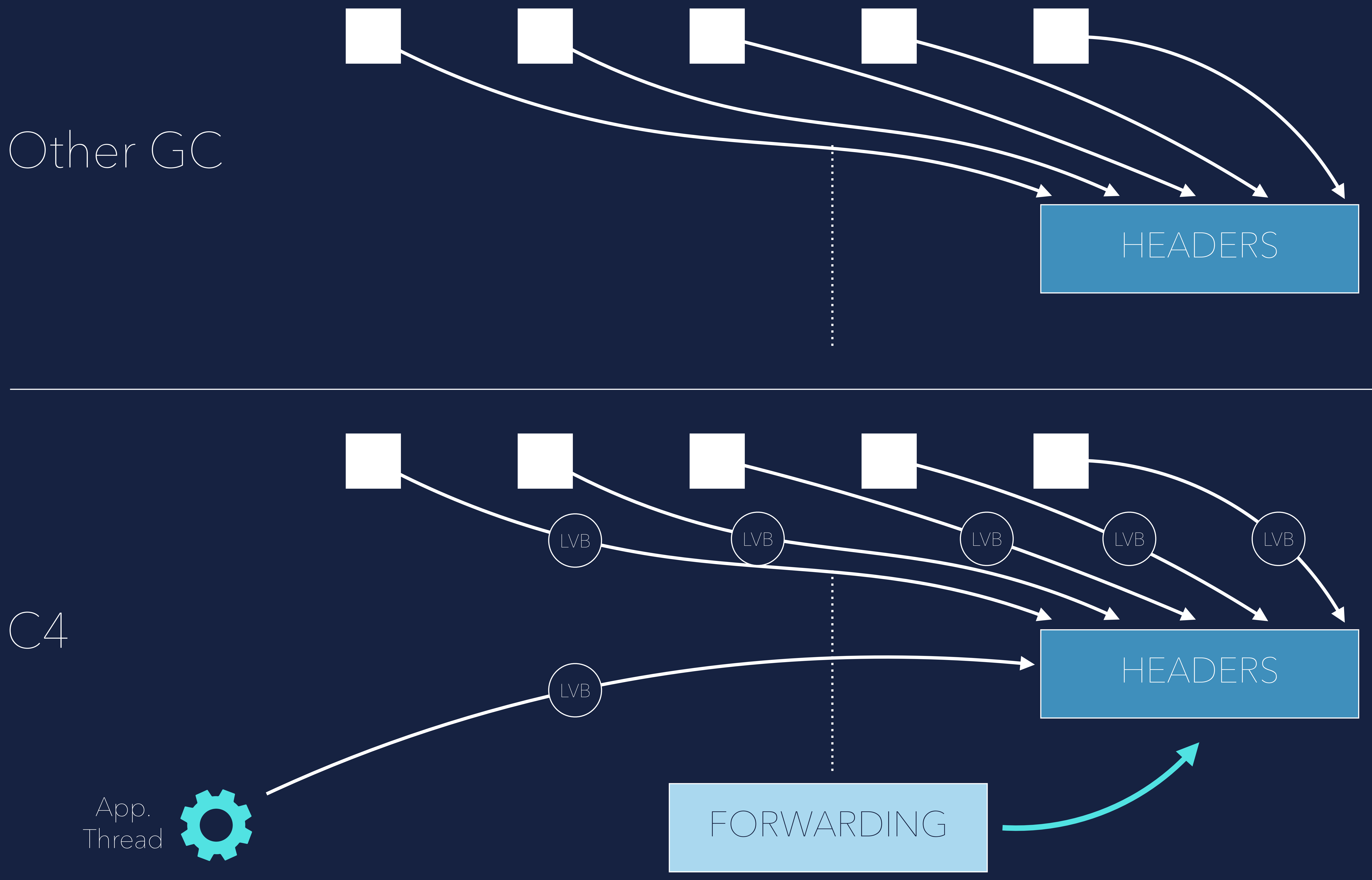
Update all references

(Walk the heap and replace all references with forwarding pointer to new location)

Memory can directly be freed

(ReadBarrier takes care about updating refs.)

Remapping phase with quick release



Remove original object

Off Heap page enables Quick Release

C4

AVAILABILITY	AZUL ZING JVM
PARALLEL	YES
CONCURRENT	FULLY
GENERATIONAL	YES
HEAP SIZE	LARGE
PAUSE TIMES	SHORT
THROUGHPUT	VERY HIGH
LATENCY	VERY LOW
CPU OVERHEAD	MODERATE (10-20%)

CHOOSE WHEN

- Response time is a high priority
- Using a very large heap (100GB+)
- Predictable response times needed

BEST SUITED FOR

- Low latency sensitive applications
- Large scale systems
- Highly concurrent applications

OS SUPPORT	🐧		

JVM SWITCH

```
> –
```

C4

NOTES

- Only available in Azul Zing JVM
- No performance overhead because of faster Falcon compiler

WHICH ØNE...?

WHICH ONE...?

Essential Criteria

🗑 Throughput

Percentage of total time spent in application vs. memory allocation and garbage collection

WHICH ONE...?

Essential Criteria

- ♻ Throughput

  Percentage of total time spent in application vs. memory allocation and garbage collection

- ♻ Latency

  Application responsiveness, affected by gc pauses

WHICH ONE...?

Essential Criteria

- Throughput

  Percentage of total time spent in application vs. memory allocation and garbage collection

- Latency

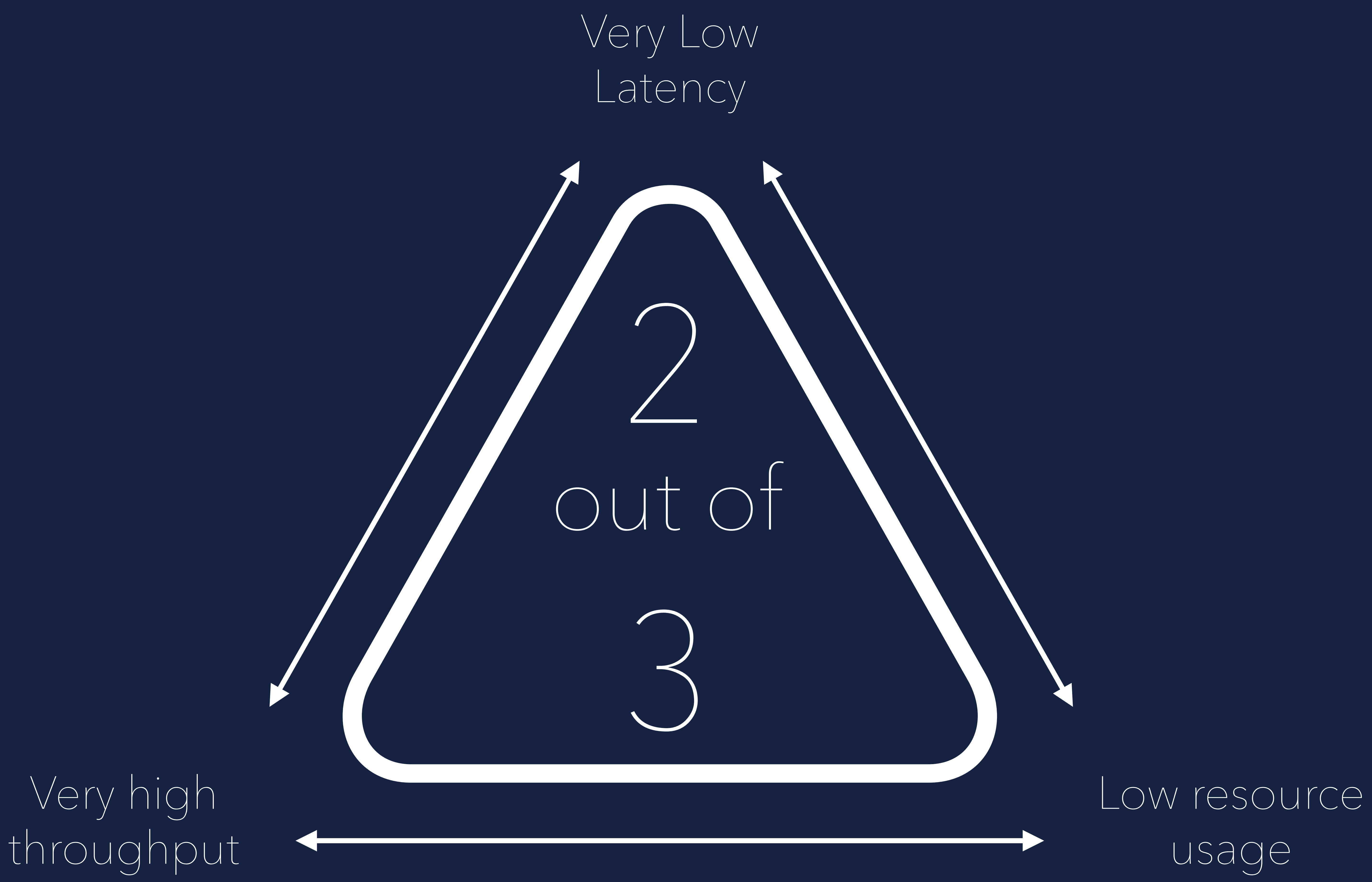
  Application responsiveness, affected by gc pauses

- Resource usage

  The working set of a process, measured in pages and cache lines

WHICH ONE…?

Essential Criteria



Very Low Latency

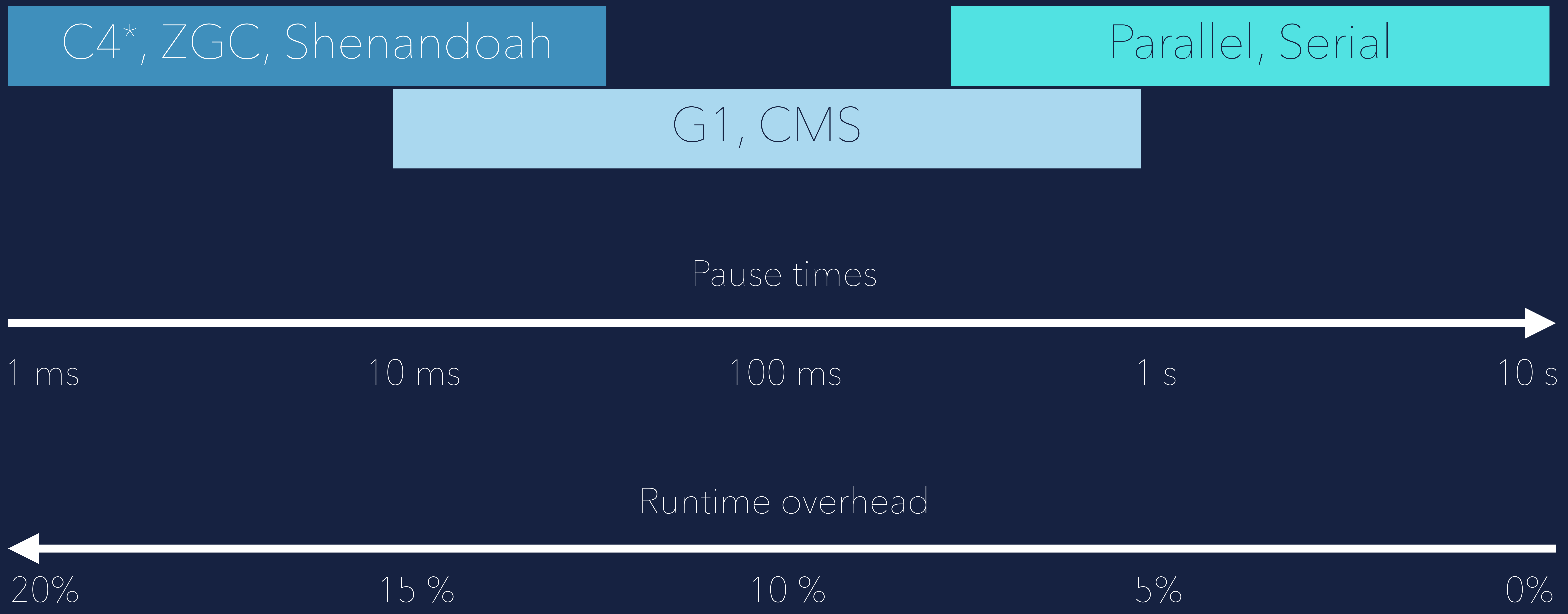
2 out of 3

Very high throughput

Low resource usage

WHICH ONE...?

Choose dependent on your workload



C4*, ZGC, Shenandoah

Parallel, Serial

G1, CMS

Pause times

1 ms

10 ms

100 ms

1 s

10 s

Runtime overhead

20%

15 %

10 %

5%

0%

* C4 has less overhead due to faster Falcon compiler

OVERVIEW

OVERVIEW

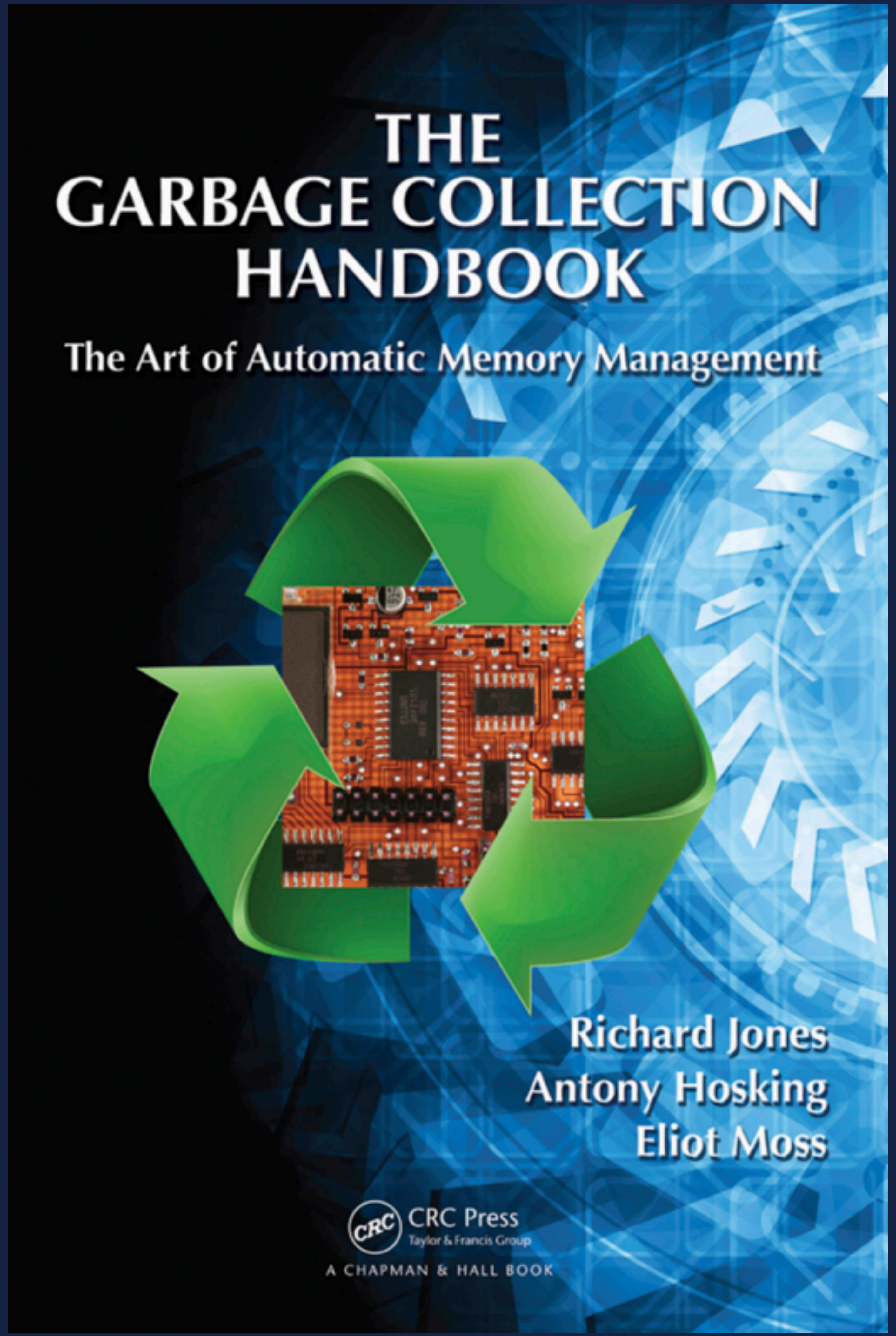
	Serial GC	Parallel GC	CMS GC	G1	Epsilon	Shenandoah	ZGC	C4
	Serial	Parallel	CMS	G1	Epsilon	Shenandoah	ZGC	C4
Availability	ALL JDK's	ALL JDK's	JDK 1.4-13	JDK 7u4+	JDK 11+	JDK 11.0.9+	JDK15 / 21+	Azul Prime
Parallel	NO	YES	YES	YES	╳	YES	YES	YES
Concurrent	NO	NO	PARTIALLY	PARTIALLY	╳	FULLY	FULLY	FULLY
Generational	YES	YES	YES	YES	╳	NO	NO / YES	YES
Heap Size	SMALL - MEDIUM	MEDIUM - LARGE	MEDIUM - LARGE	MEDIUM - LARGE	╳	LARGE	VERY LARGE	VERY LARGE
Pause Times	LONGER	MODERATE	MODERATE	SHORT - MEDIUM	╳	VERY SHORT (<10ms)	VERY SHORT (<1ms)	VERY SHORT (<1ms)
Throughput	LOW	HIGH	MODERATE	HIGH	╳	VERY HIGH	VERY HIGH	VERY HIGH
Latency	HIGHER	LOWER	MODERATE	LOWER	╳	VERY LOW	VERY LOW	VERY LOW
Performance	LOWER	HIGHER	MODERATE	HIGHER	VERY HIGH	VERY HIGH	VERY HIGH	VERY HIGH
CPU Overhead	LOW	LOWER	MODERATE	MODERATE	VERY LOW	LOW - MODERATE	LOW - MODERATE	LOW - MODERATE
Tail latency	HIGH	HIGH	HIGH	HIGH	╳	MODERATE	LOW	LOW

WANNA KNOW MORE ?

azul

WANNA KNOW MORE ?

R. Jones et al. "The Garbage Collection Handbook". Chapman & Hall/CRC, 2012



azul

THANK YOU

azul