

TRASH TALK



EXPLORING THE MEMORY MANAGEMENT IN THE JVM

ABOUT ME.



Gerrit Grunwald | Developer Advocate | 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

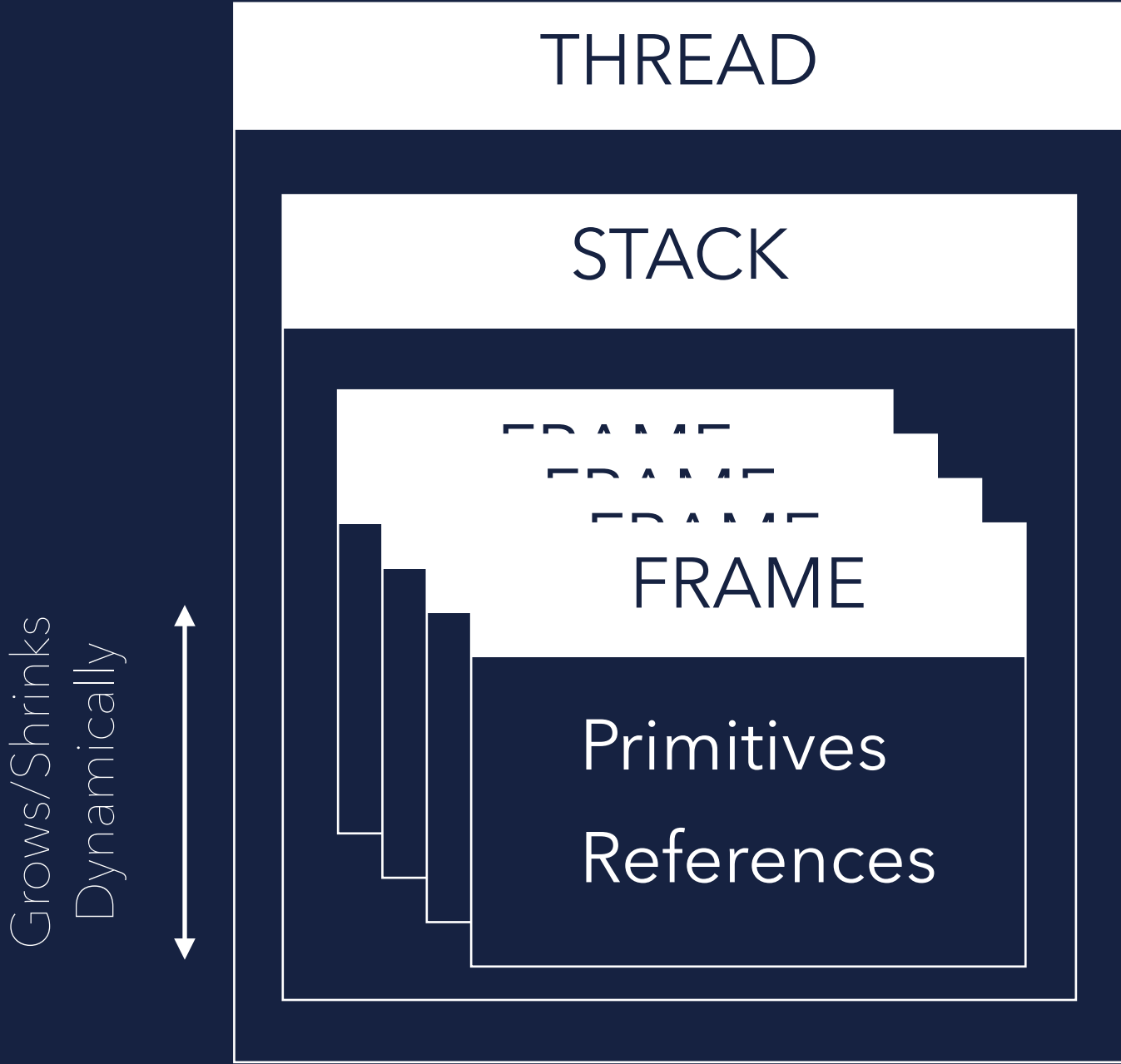
MEMORY MANAGEMENT

Why you should care...

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

MEMORY MANAGEMENT

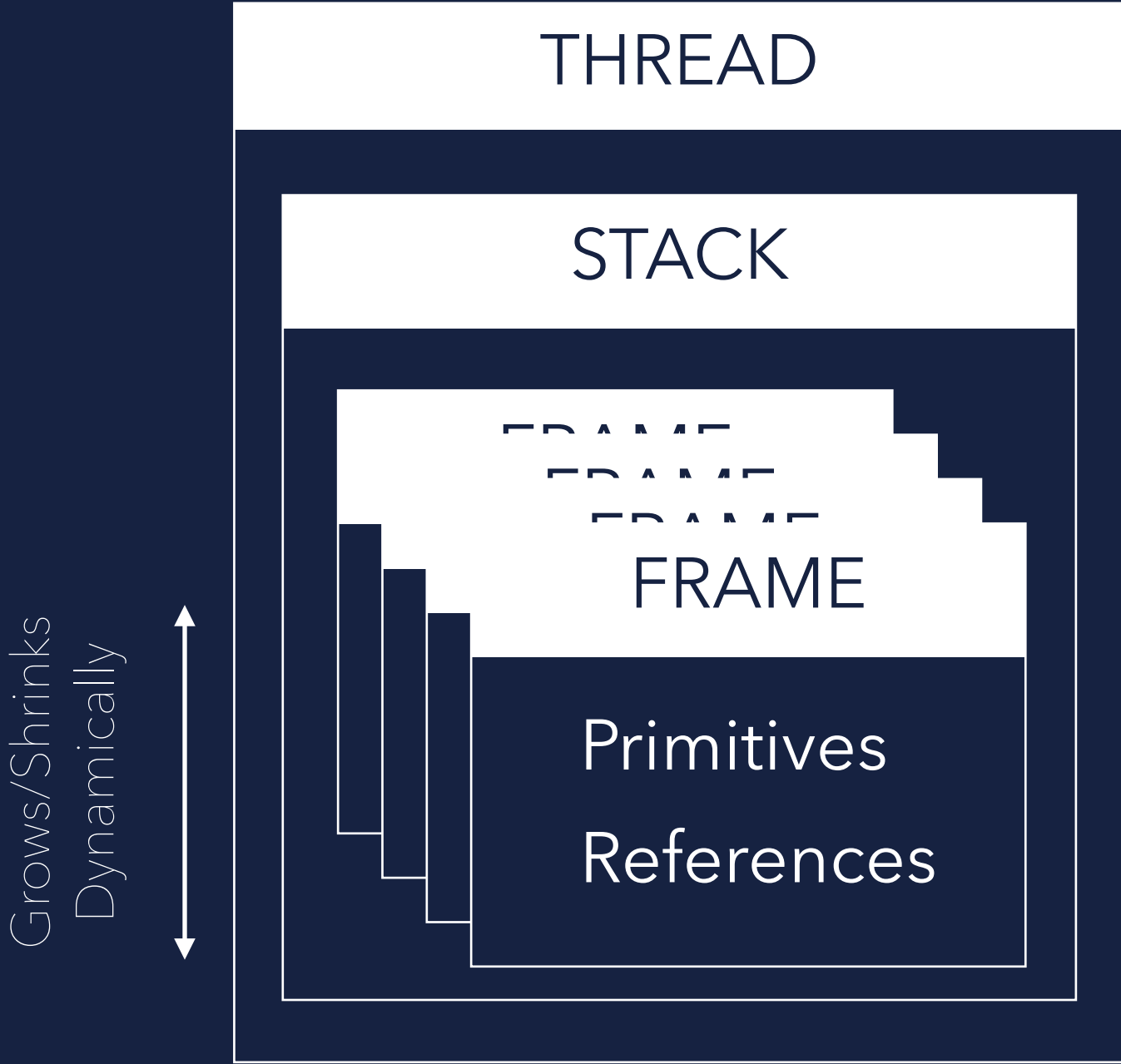
Stack, Heap and Metaspace



Local access -> thread safe

MEMORY MANAGEMENT

Stack, Heap and Metaspaces



Local access -> thread safe



Shared access -> Not thread safe
Needs Garbage Collection

MEMORY MANAGEMENT

Stack, Heap and Metaspaces



Local access -> thread safe

Shared access -> Not thread safe
Needs Garbage Collection

Contains info needed for
JVM to work with classes

MEMORY MANAGEMENT

Stack, Heap and Metaspaces



Local access -> thread safe

Shared access -> Not thread safe
Needs Garbage Collection

Contains info needed for JVM to work with classes

StackOverflowError

OutOfMemoryError

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  
  
}
```

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 for thread 1

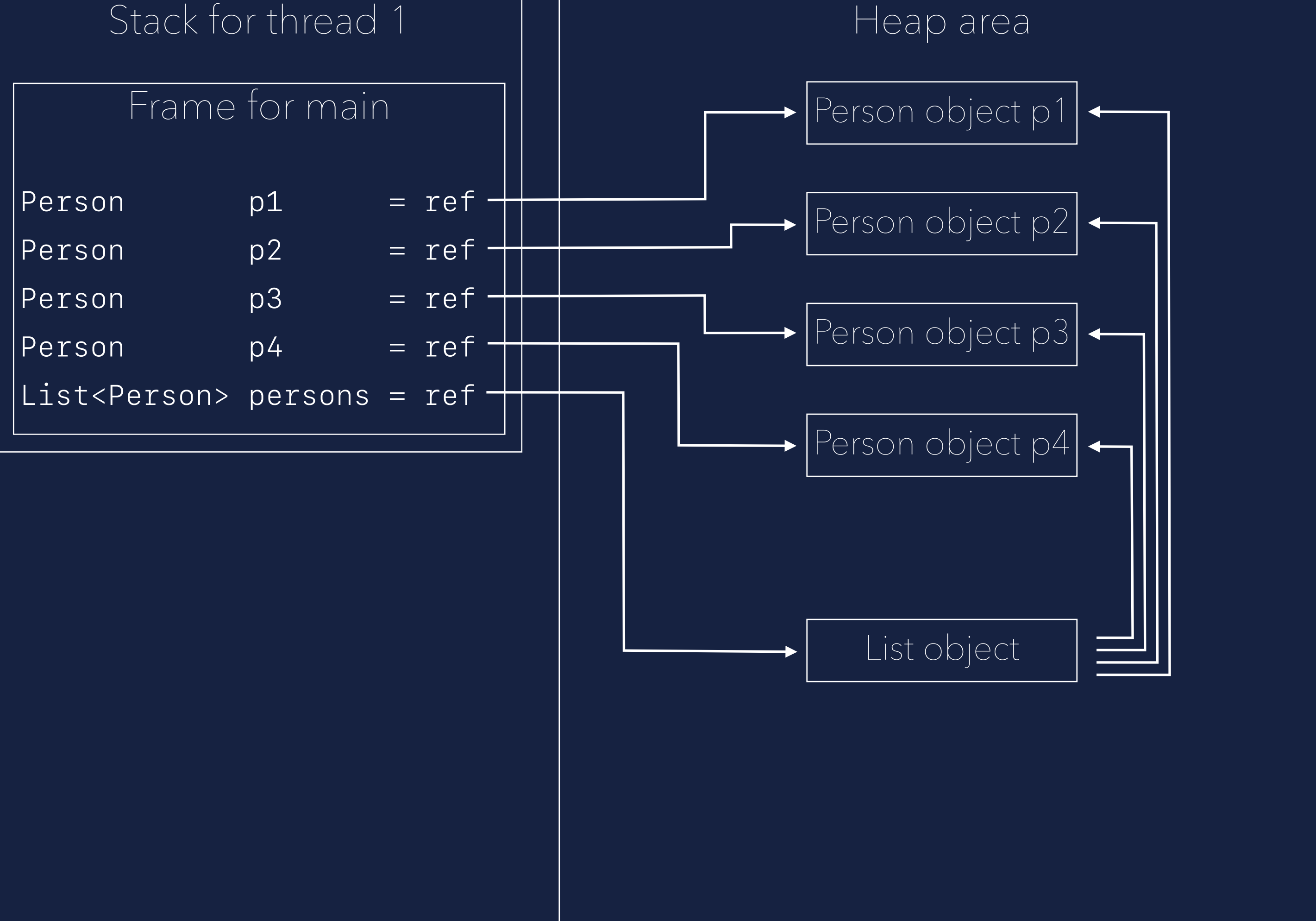
Frame for main

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

MEMORY MANAGEMENT

In the JVM...

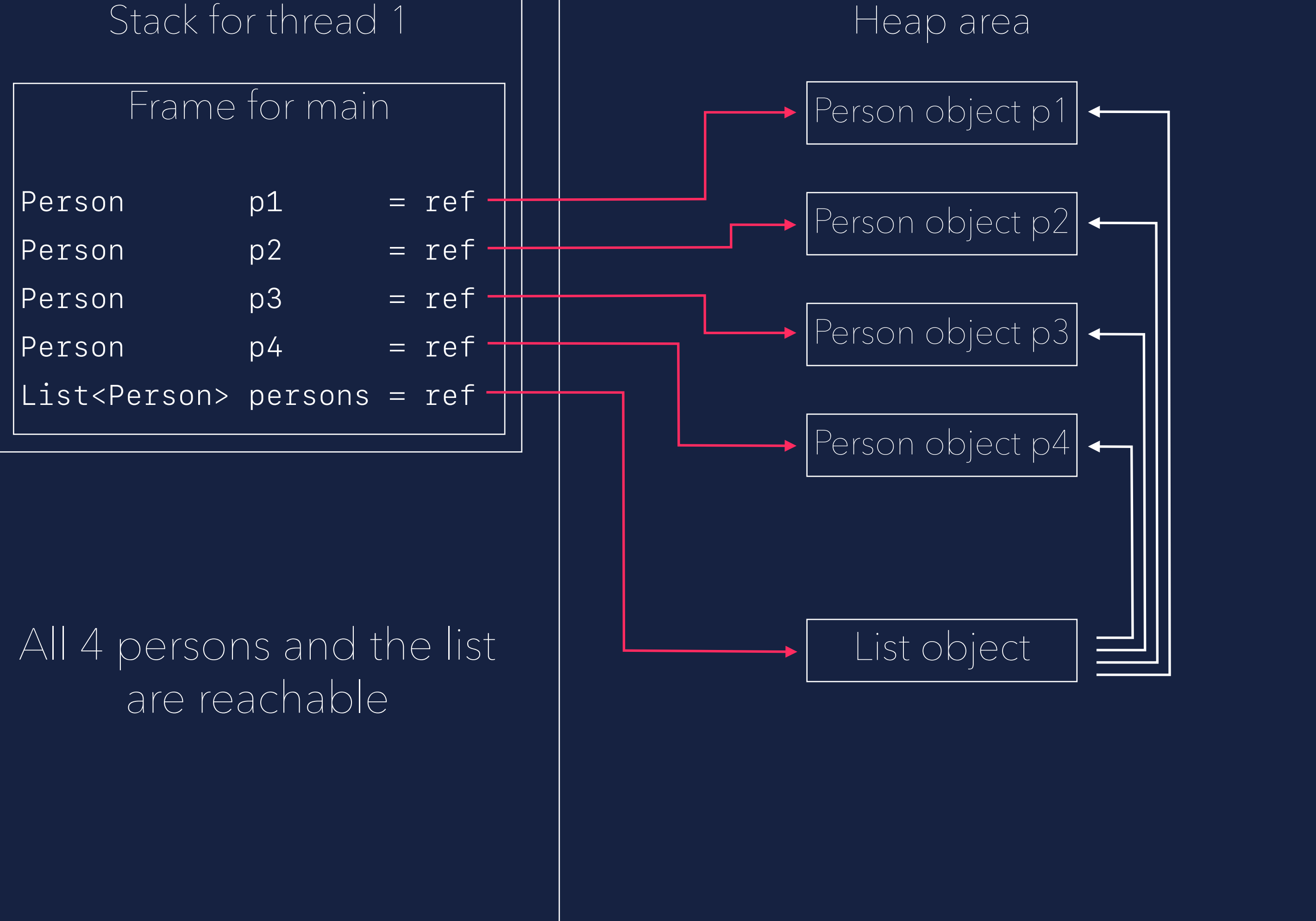
```
public static void main(String[] args) {  
  
    record Person(String name) {  
        @Override public String toString() { return name(); }  
    }  
  
    Person p1 = new Person("Gerrit");  
    Person p2 = new Person("Sandra");  
    Person p3 = new Person("Lilli");  
    Person p4 = new Person("Anton");  
  
    List<Person> persons = Arrays.asList(p1, p2, p3, p4);  
  
    System.out.println(p1); // -> Gerrit  
  
}
```



MEMORY MANAGEMENT

In the JVM...

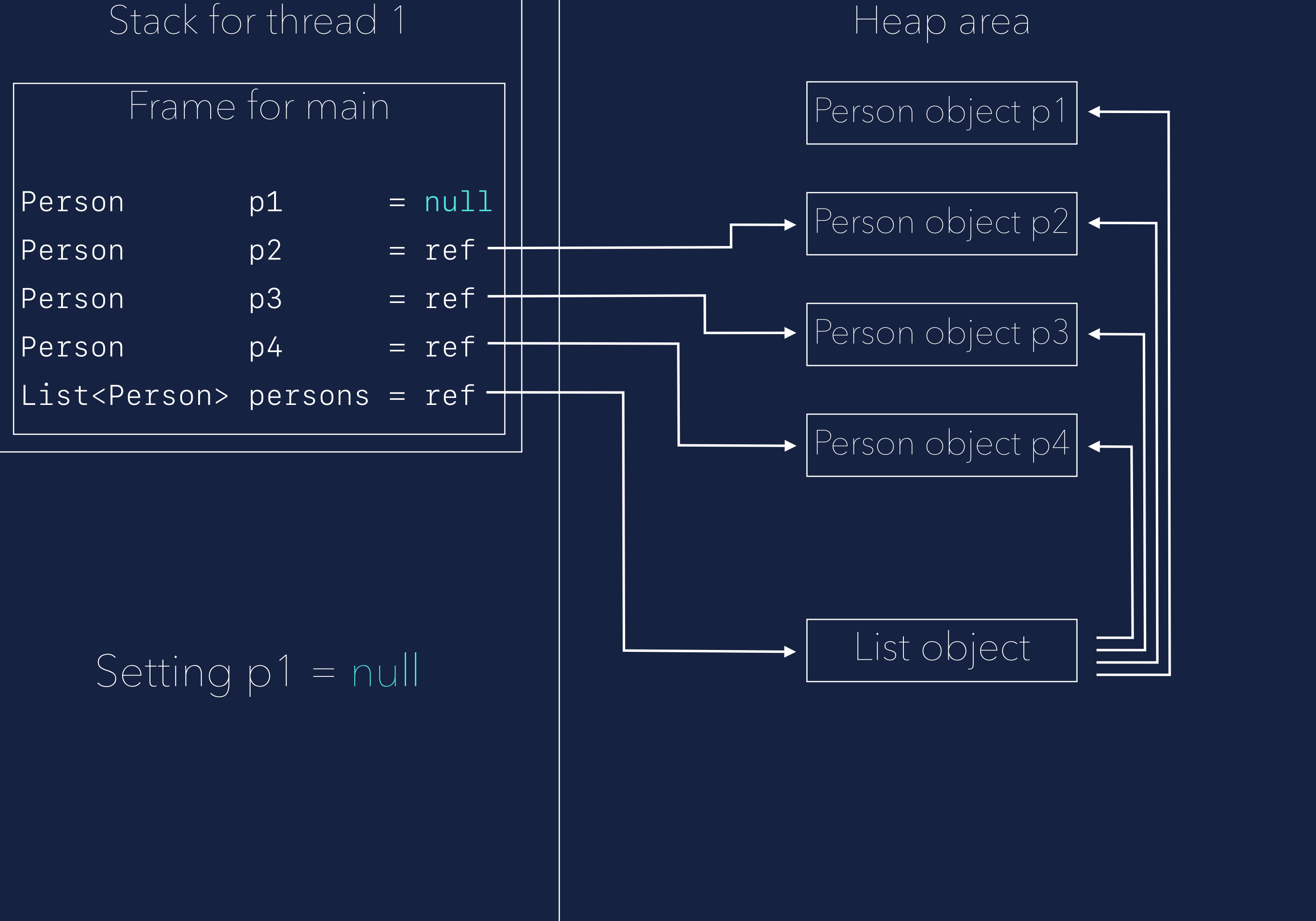
```
public static void main(String[] args) {  
  
    record Person(String name) {  
        @Override public String toString() { return name(); }  
    }  
  
    Person p1 = new Person("Gerrit");  
    Person p2 = new Person("Sandra");  
    Person p3 = new Person("Lilli");  
    Person p4 = new Person("Anton");  
  
    List<Person> persons = Arrays.asList(p1, p2, p3, p4);  
  
    System.out.println(p1); // -> Gerrit  
  
}
```



MEMORY MANAGEMENT

In the JVM...

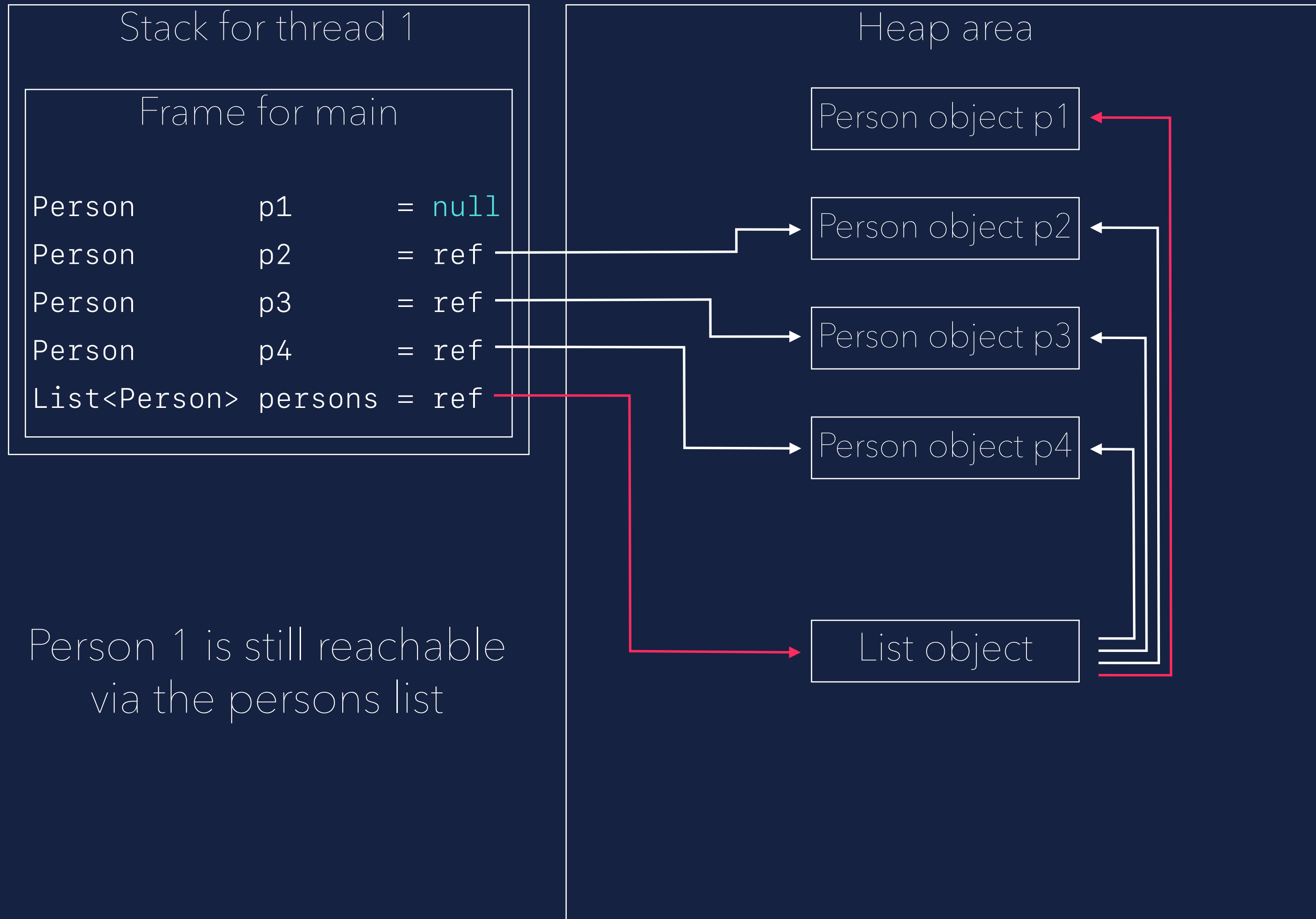
```
public static void main(String[] args) {  
  
    record Person(String name) {  
        @Override public String toString() { return name(); }  
    }  
  
    Person p1 = new Person("Gerrit");  
    Person p2 = new Person("Sandra");  
    Person p3 = new Person("Lilli");  
    Person p4 = new Person("Anton");  
  
    List<Person> persons = Arrays.asList(p1, p2, p3, p4);  
  
    System.out.println(p1); // -> Gerrit  
  
    p1 = null;  
  
}
```



MEMORY MANAGEMENT

In the JVM...

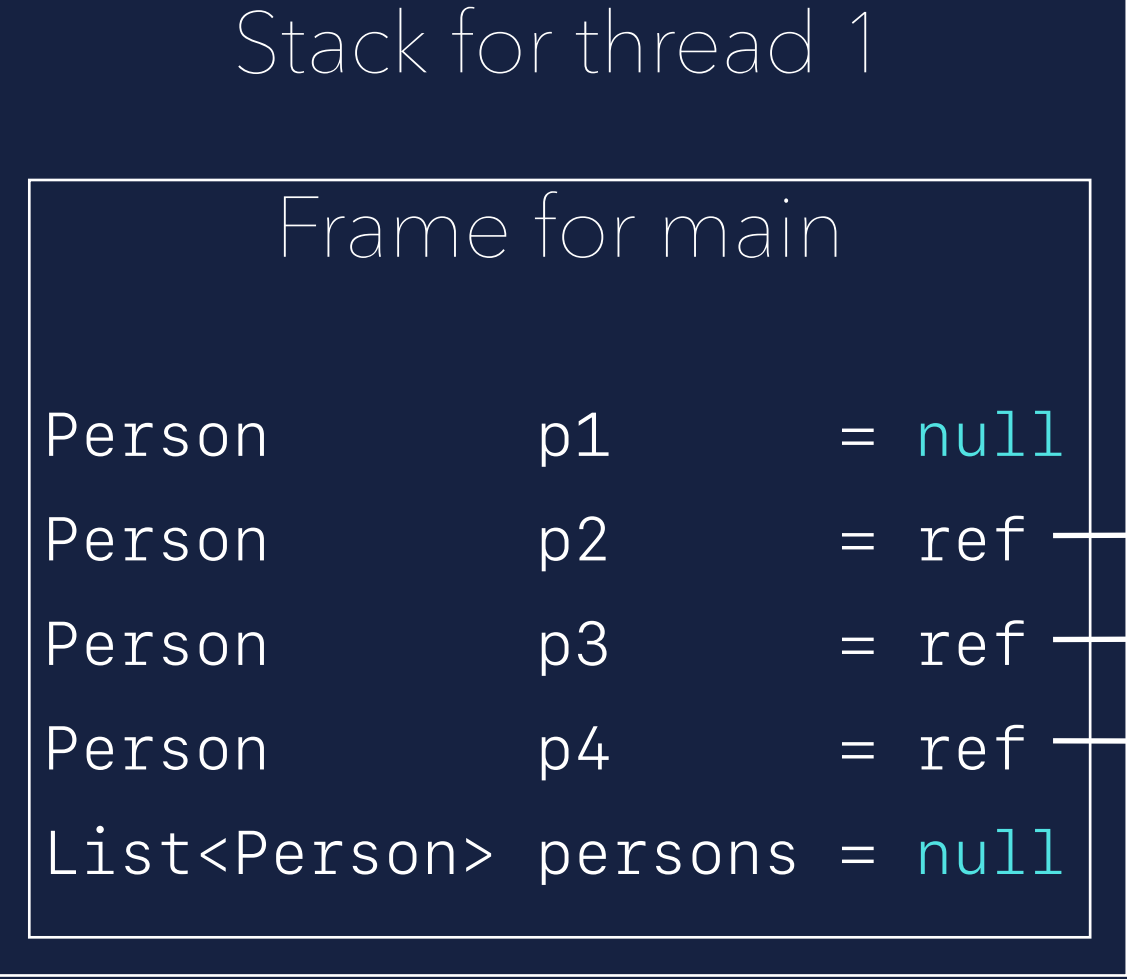
```
public static void main(String[] args) {  
  
    record Person(String name) {  
        @Override public String toString() { return name(); }  
    }  
  
    Person p1 = new Person("Gerrit");  
    Person p2 = new Person("Sandra");  
    Person p3 = new Person("Lilli");  
    Person p4 = new Person("Anton");  
  
    List<Person> persons = Arrays.asList(p1, p2, p3, p4);  
  
    System.out.println(p1); // -> Gerrit  
  
    p1 = null;  
  
    System.out.println(persons.get(0)); // -> Gerrit  
  
}
```



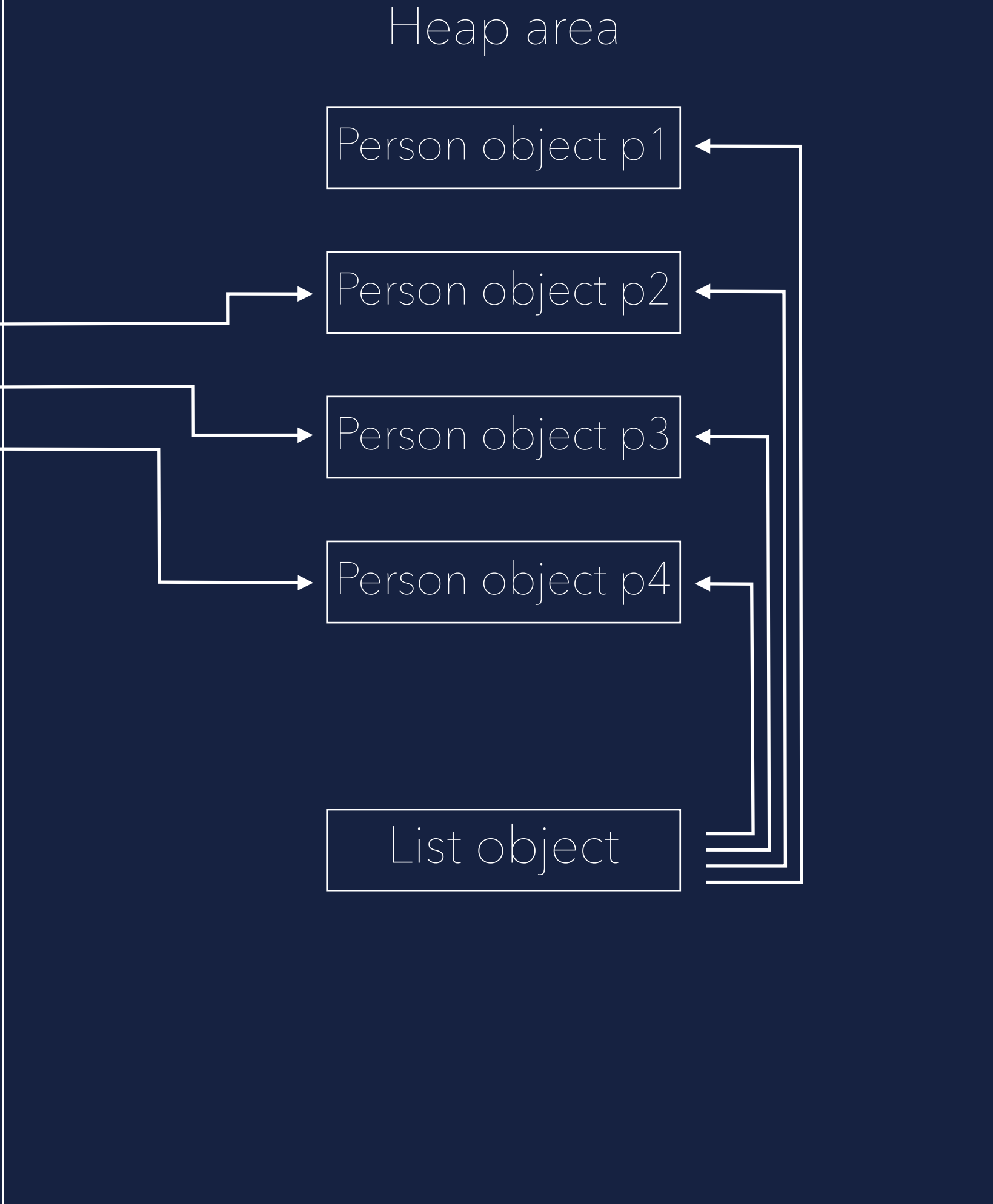
MEMORY MANAGEMENT

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;  
}
```



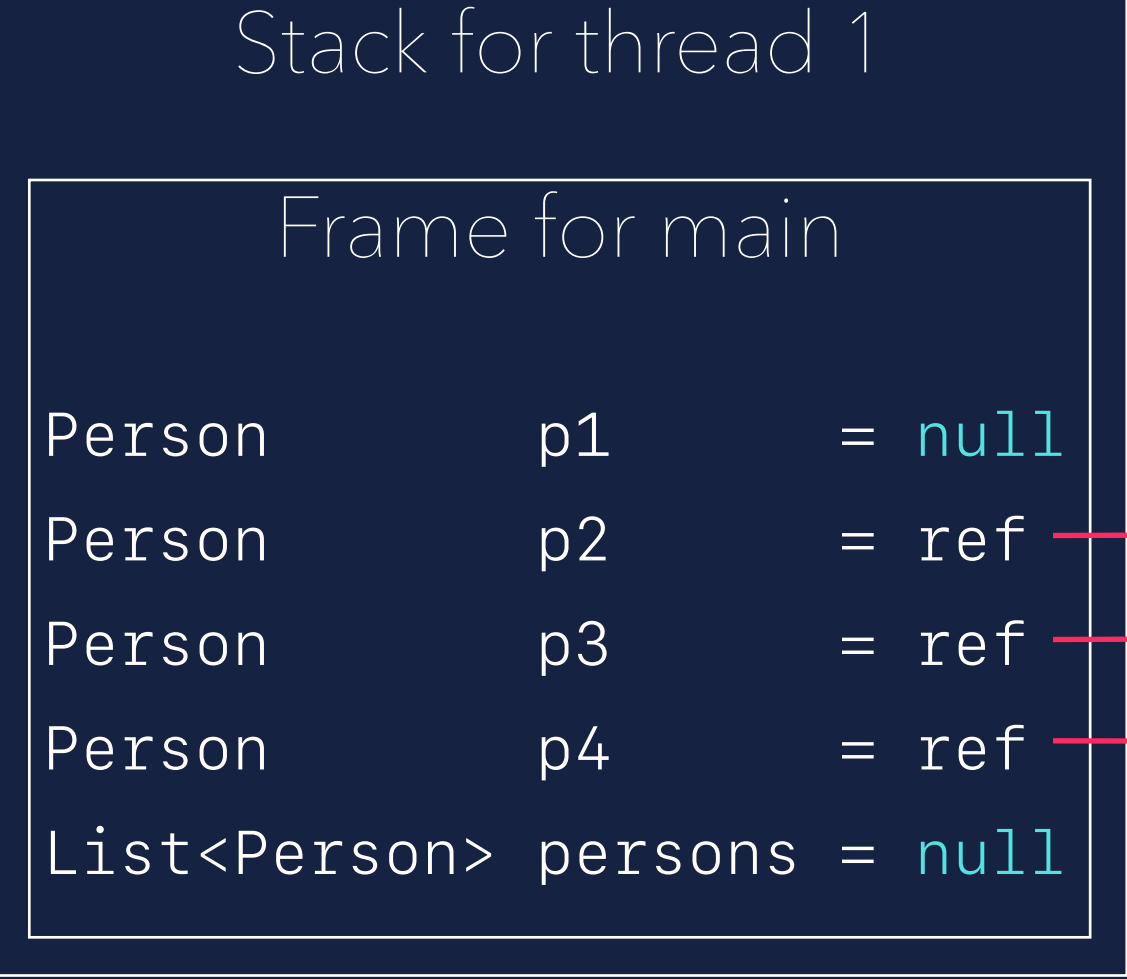
Setting persons = null



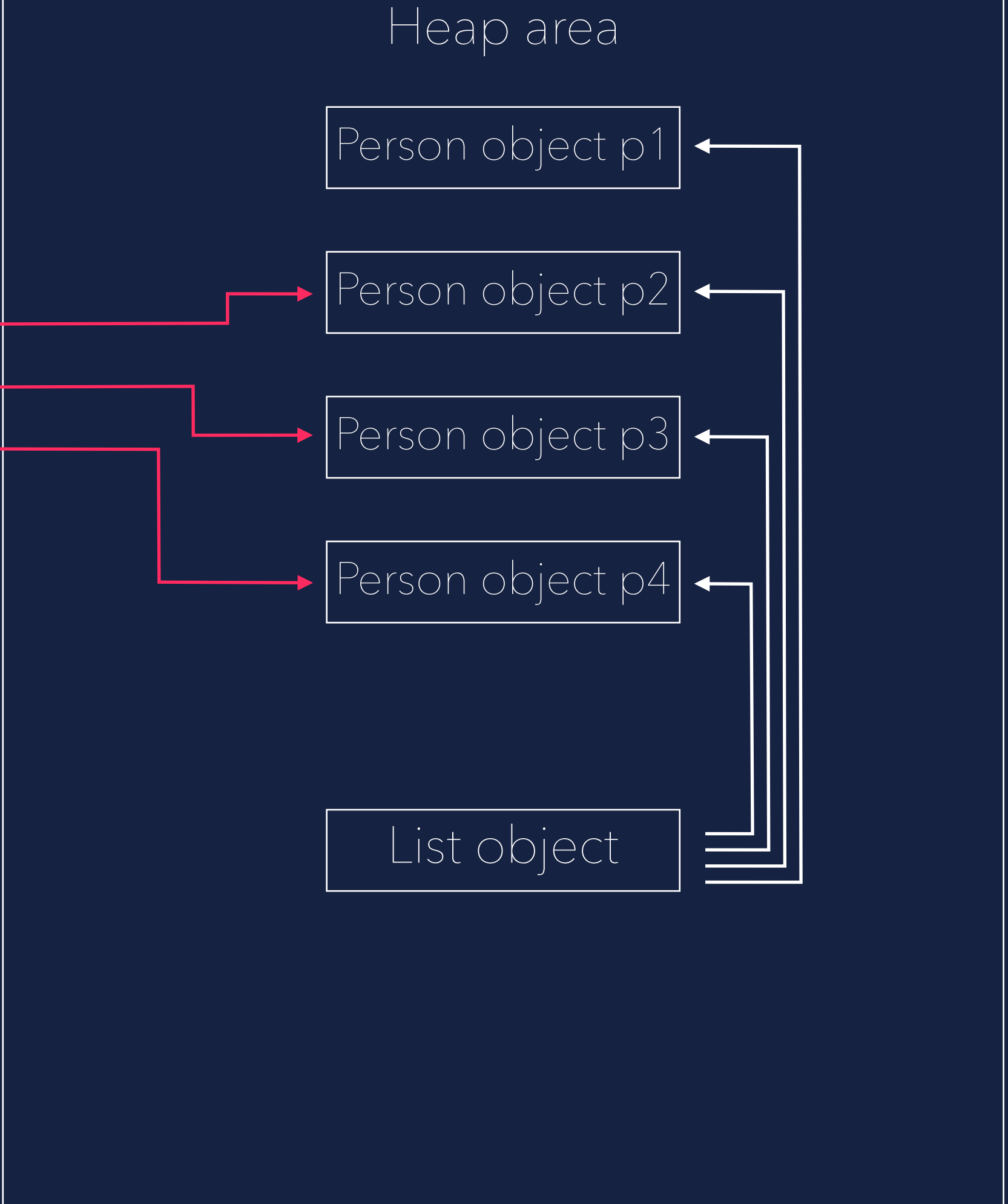
MEMORY MANAGEMENT

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  
  
    persons = null;  
}
```



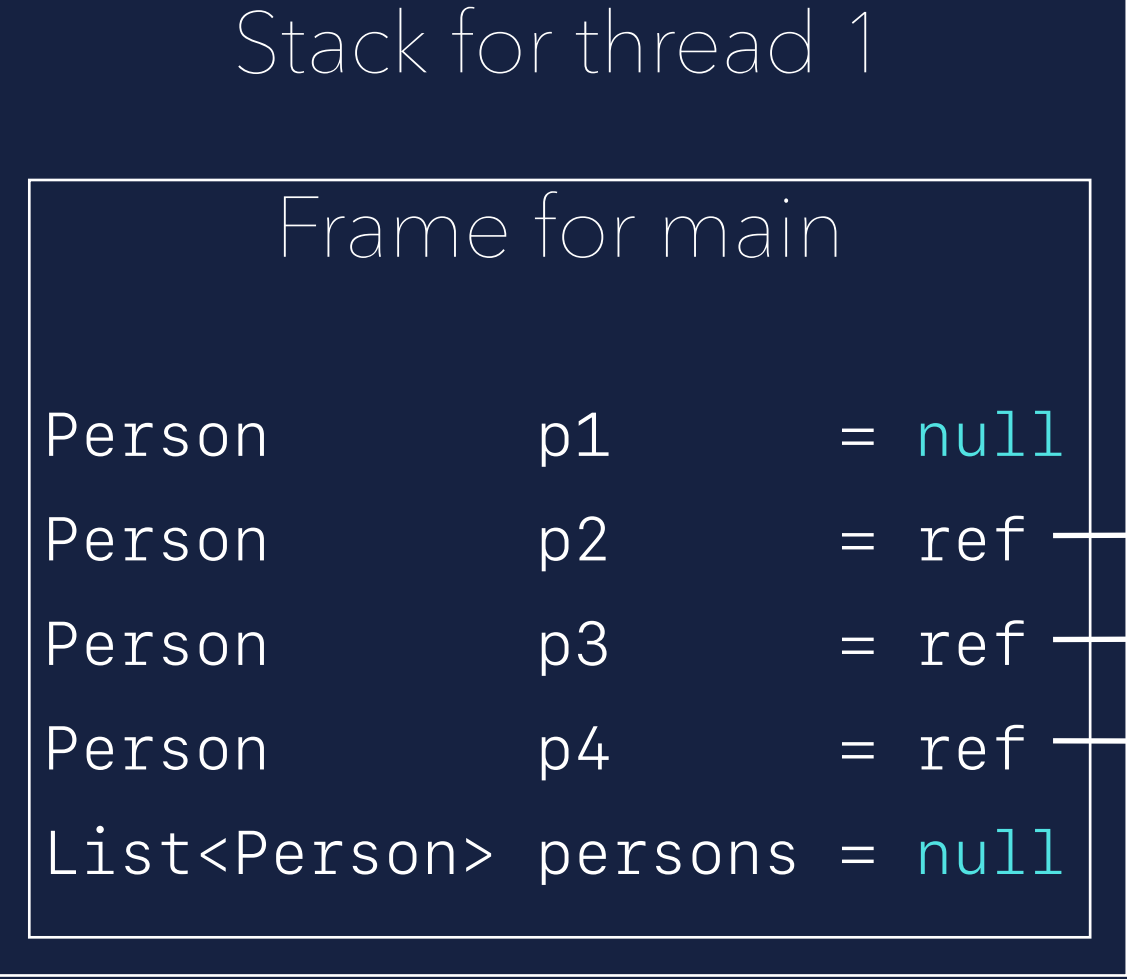
Only p2, p3 and p4 are reachable



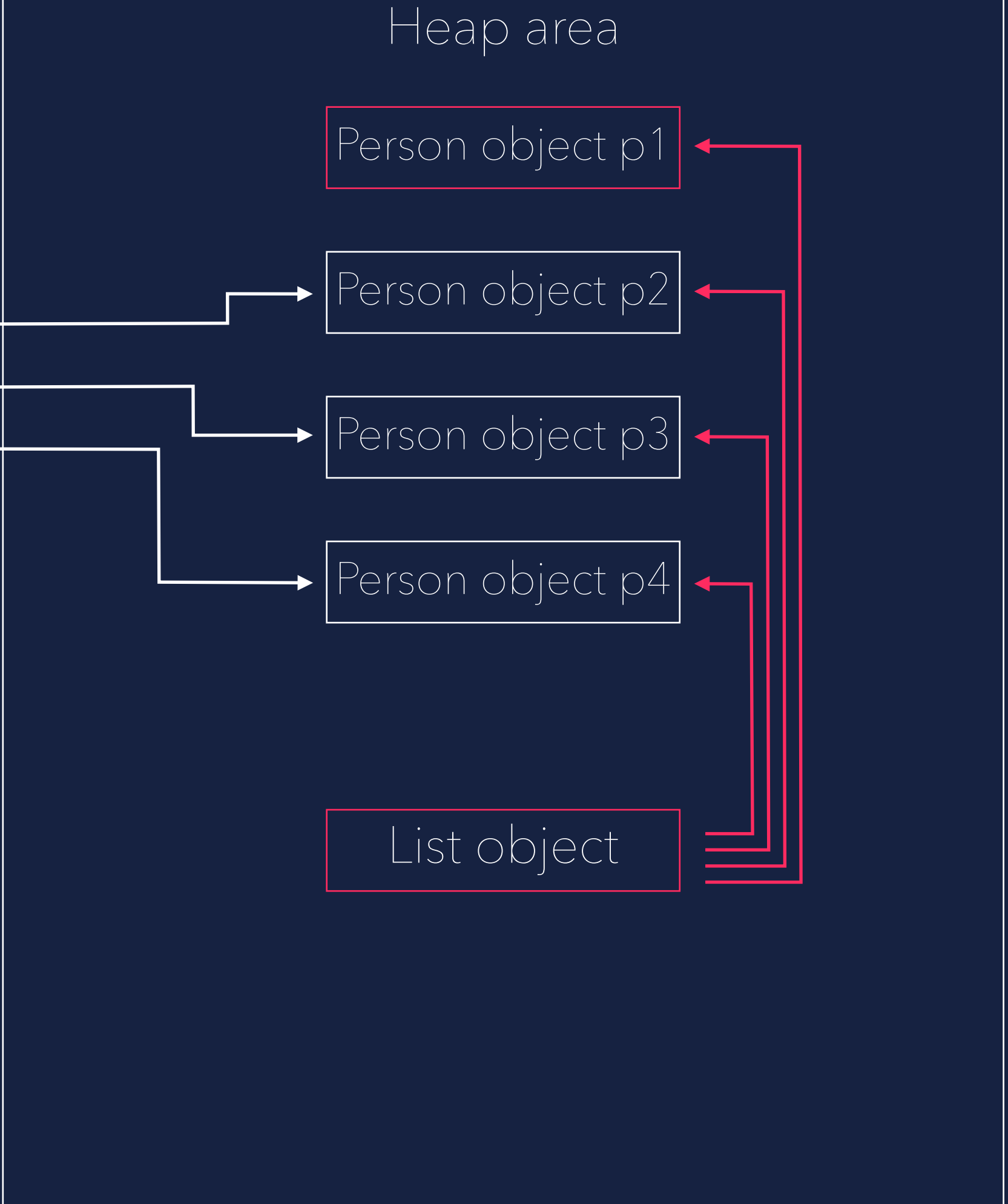
MEMORY MANAGEMENT

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;  
}
```



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

**CONSERVATIVE
AND
PRECISE**

GARBAGE COLLECTION



Conservative and Precise



Conservative does not fully identify all object references
(assumes any bit pattern in memory could be a reference, lead to more false positives)

GARBAGE COLLECTION

Conservative and Precise

-  Conservative does not fully identify all object references
(assumes any bit pattern in memory could be a reference, lead to more false positives)
-  Precise correctly identifies all references in an object
(needed in order to move objects)

PHASES

(precise collectors)

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

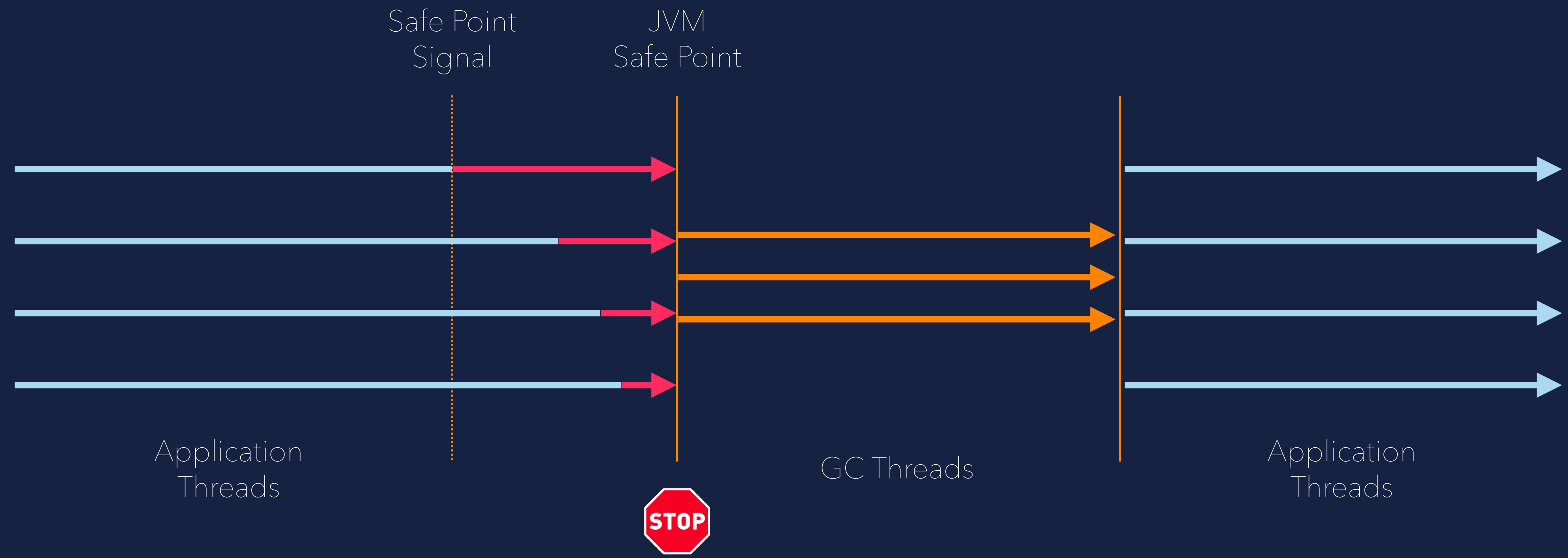
 Freeing
Reclaim resources held by dead objects

 Compaction
Periodically relocate live objects

**STOPPING THE
WORLD**

STOPPING THE WORLD RL D

Halt of all application threads



COLLECTORS

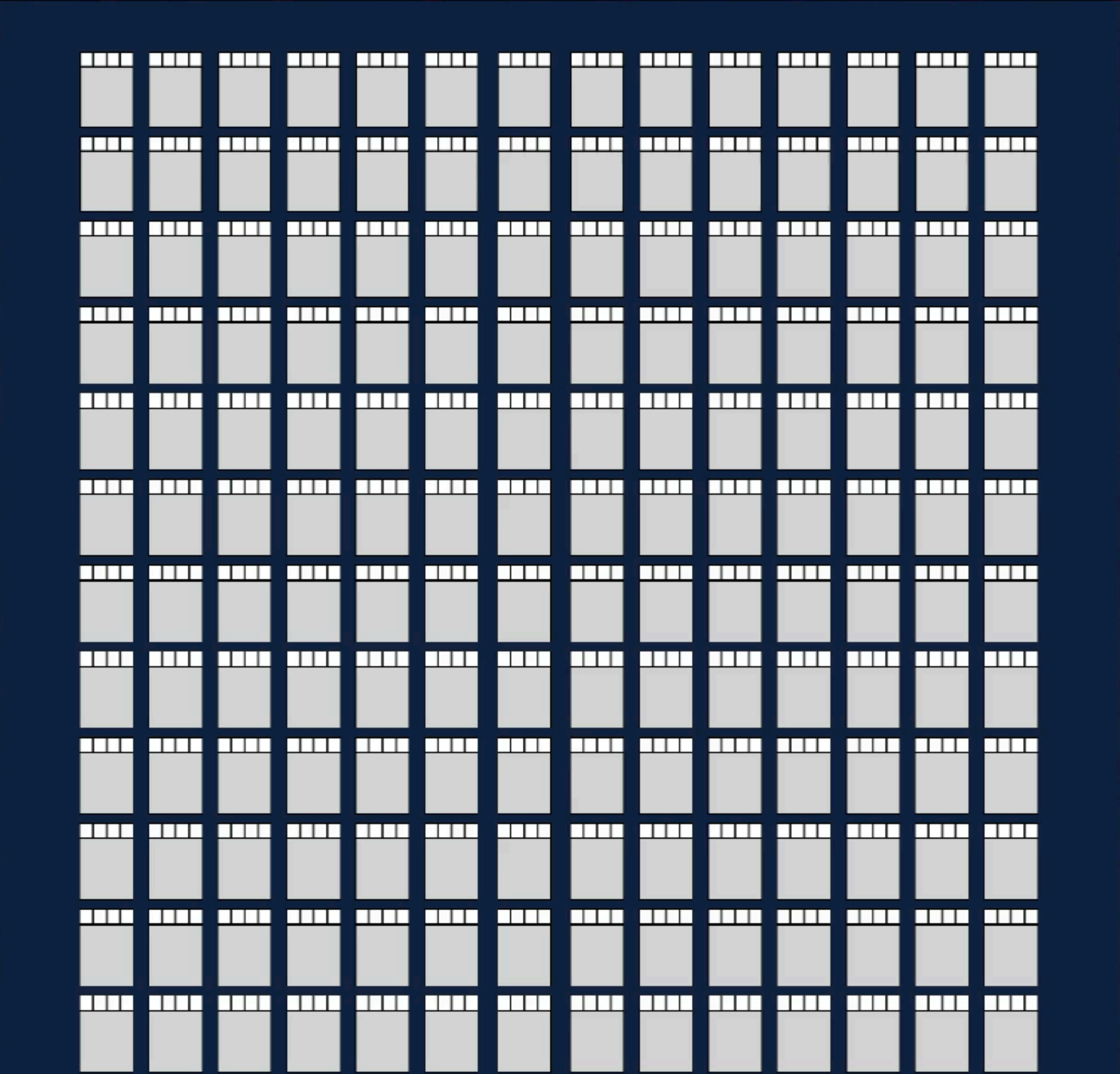
NONMOVING COLLECTOR

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
- Referenced Cell (survived 1 GC)



Fragmentation

Heap

MOVING COLLECTORS

Compacting Collector & Copy Collector

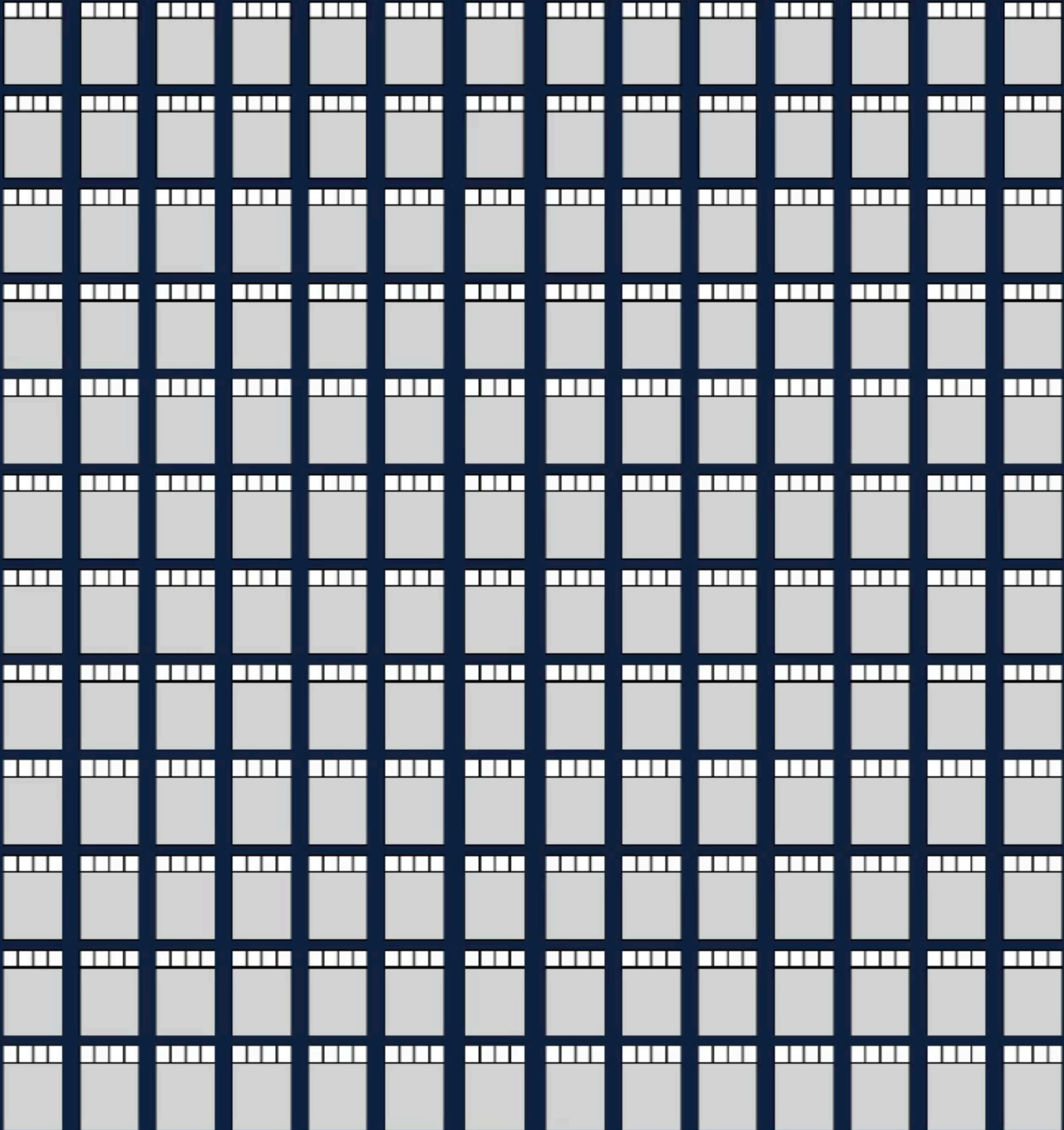
COMPACTING COLLECTOR

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



Heap

- Free Cell
- Referenced Cell
- Dereferenced Cell
- Marked Cell
- Referenced Cell (survived 1 GC)

Easy allocation through "Bump the pointer" technique



Headroom
20-50%

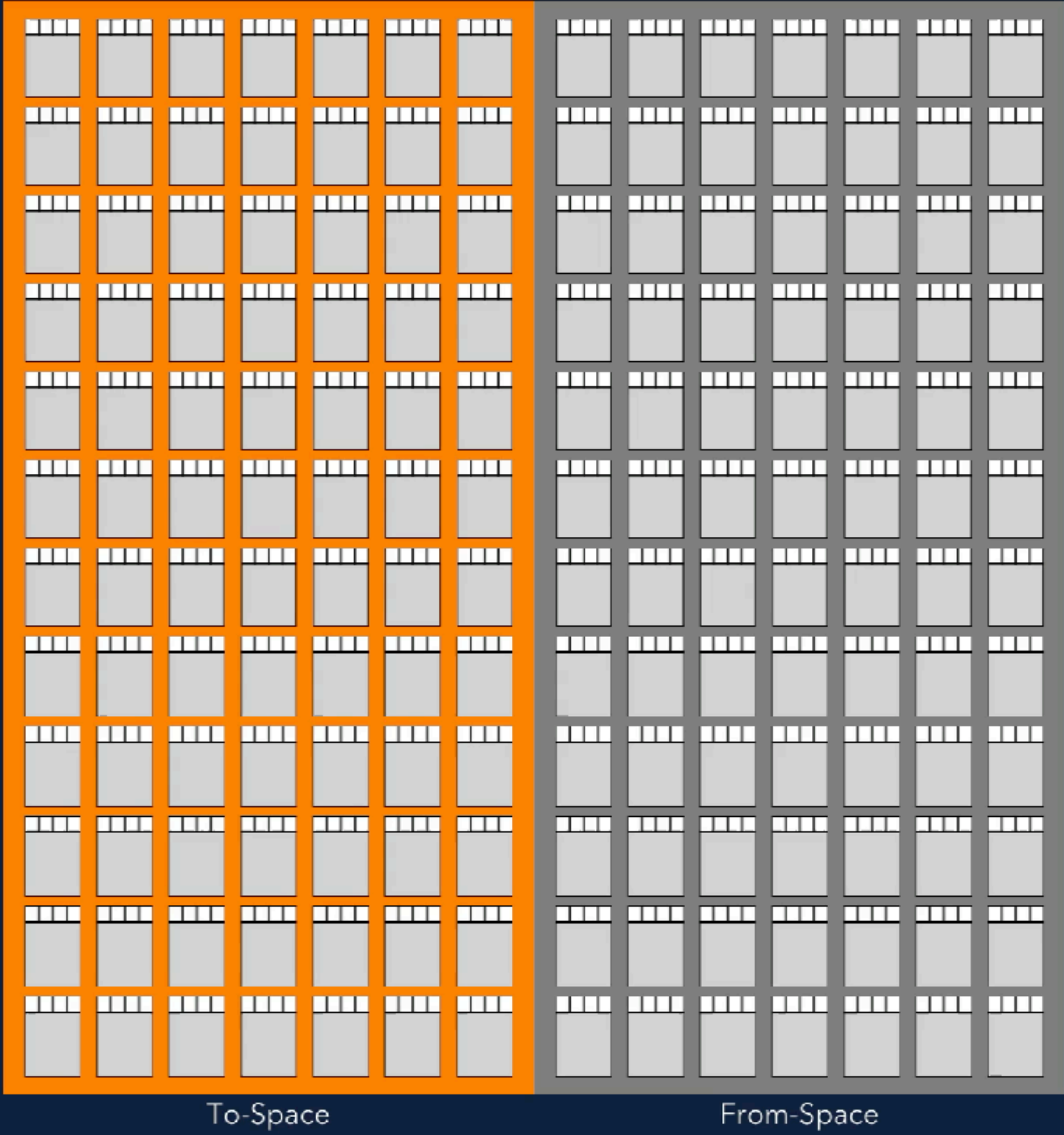
COPY COLLECTOR

Mark & Copy

COPY COLLECTOR

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

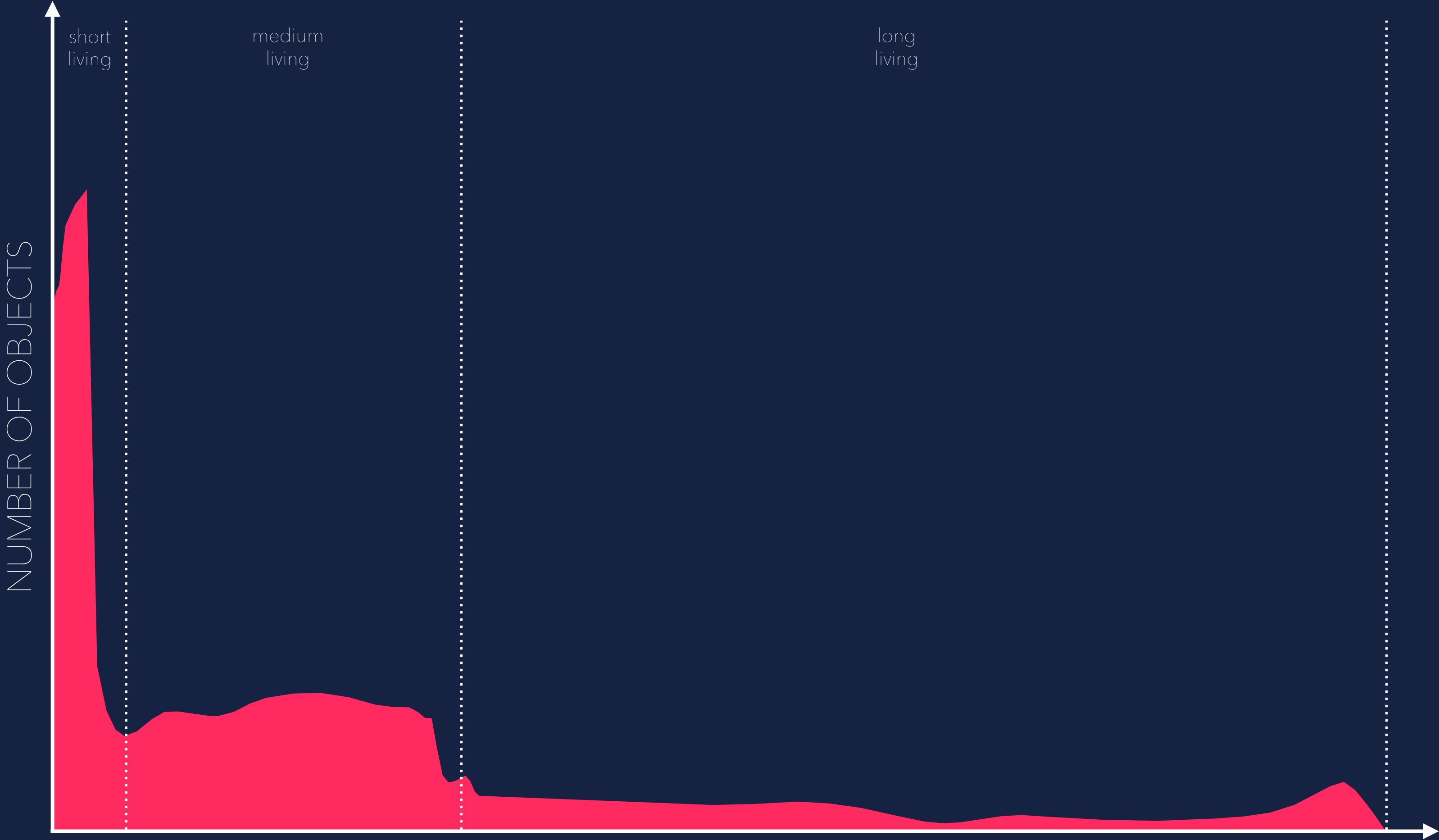

Long living
objects and twice
as much memory

GENERATIONAL COLLECTOR

Generational Mark & Compact

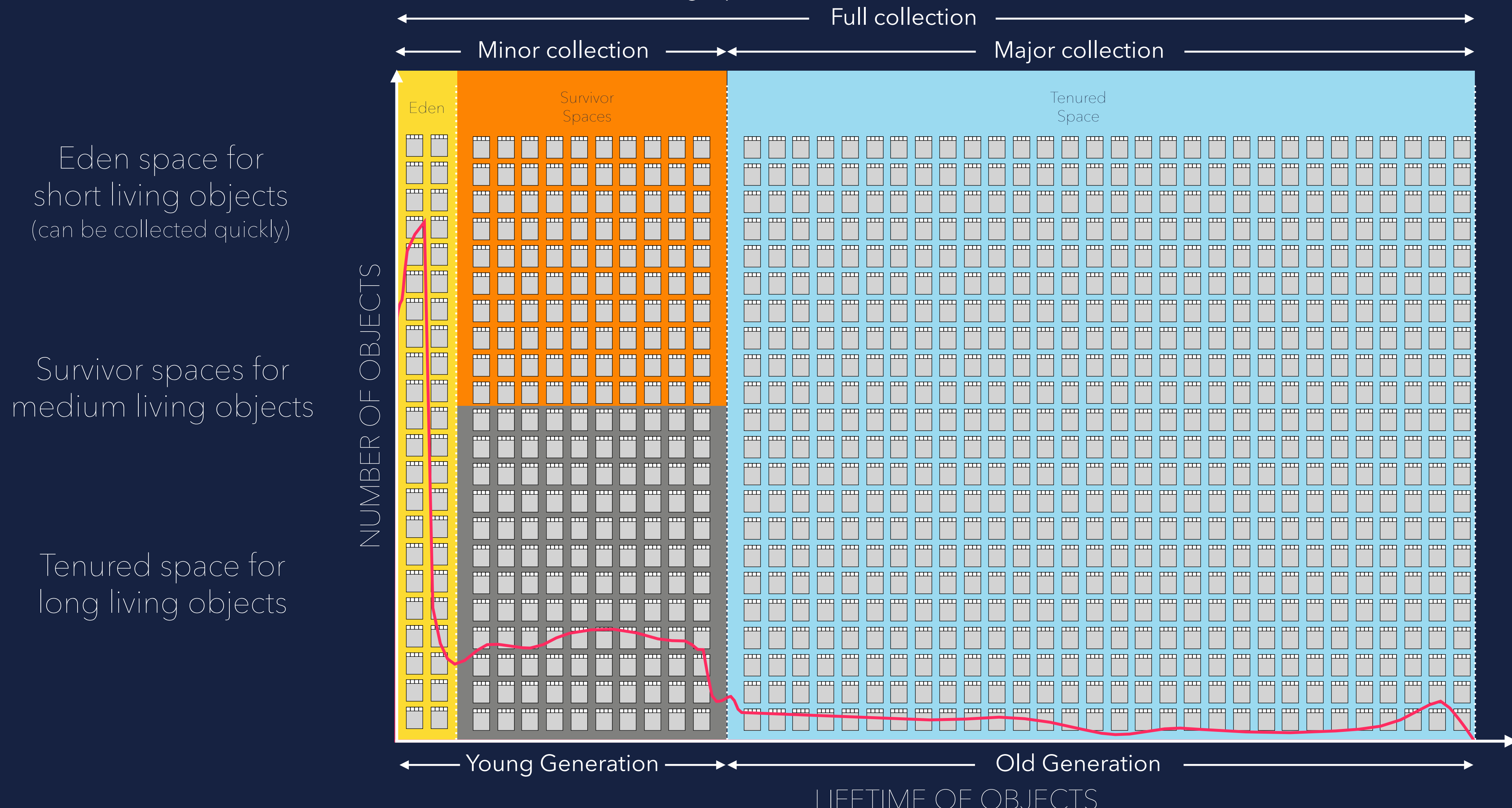
GENERATIONAL COLLECTOR

Weak Generational Hypothesis (Most objects die young)



GENERATIONAL COLLECTOR

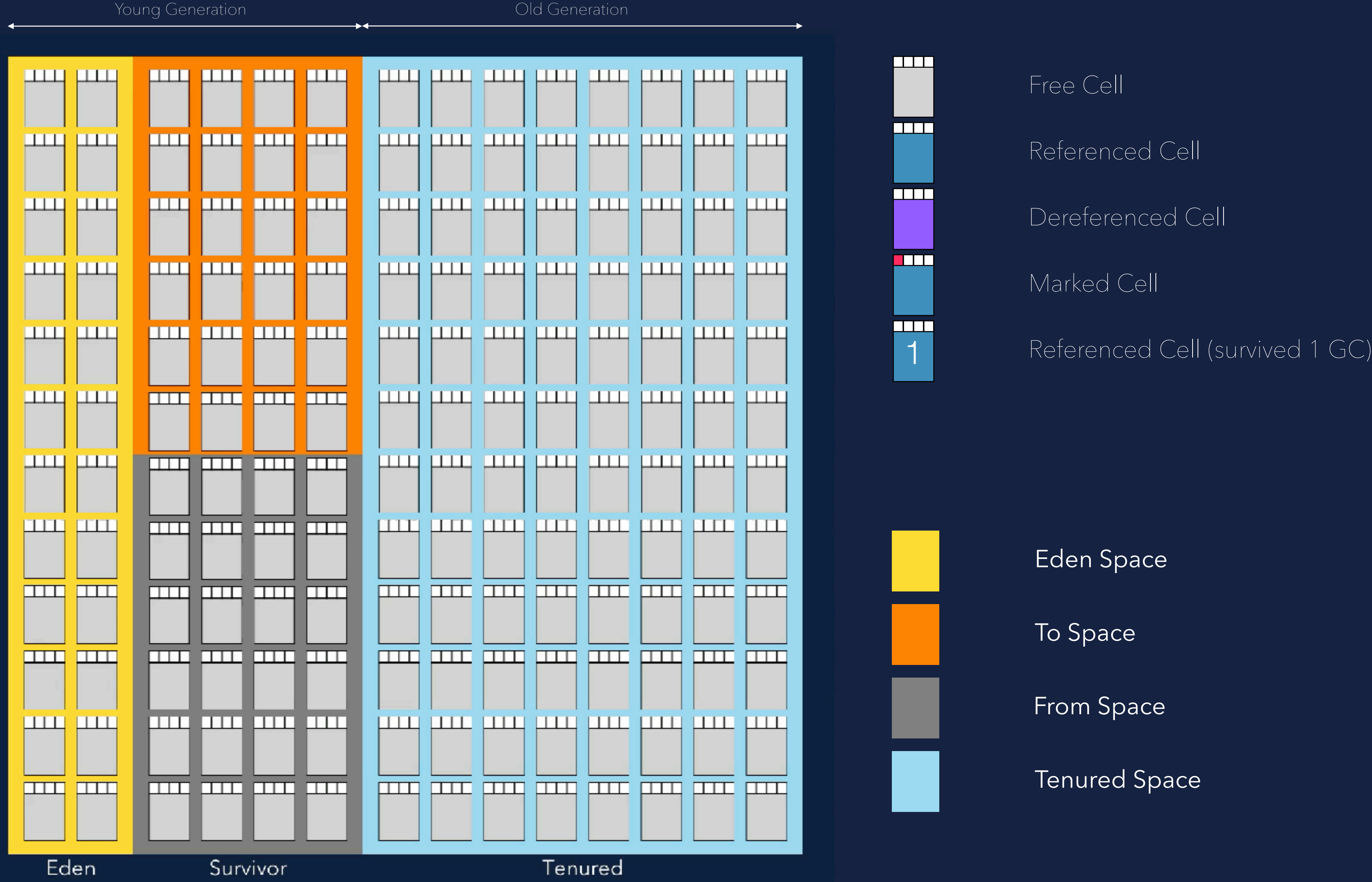
Weak Generational Hypothesis (Most objects die young)



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



REMEMBERED SET

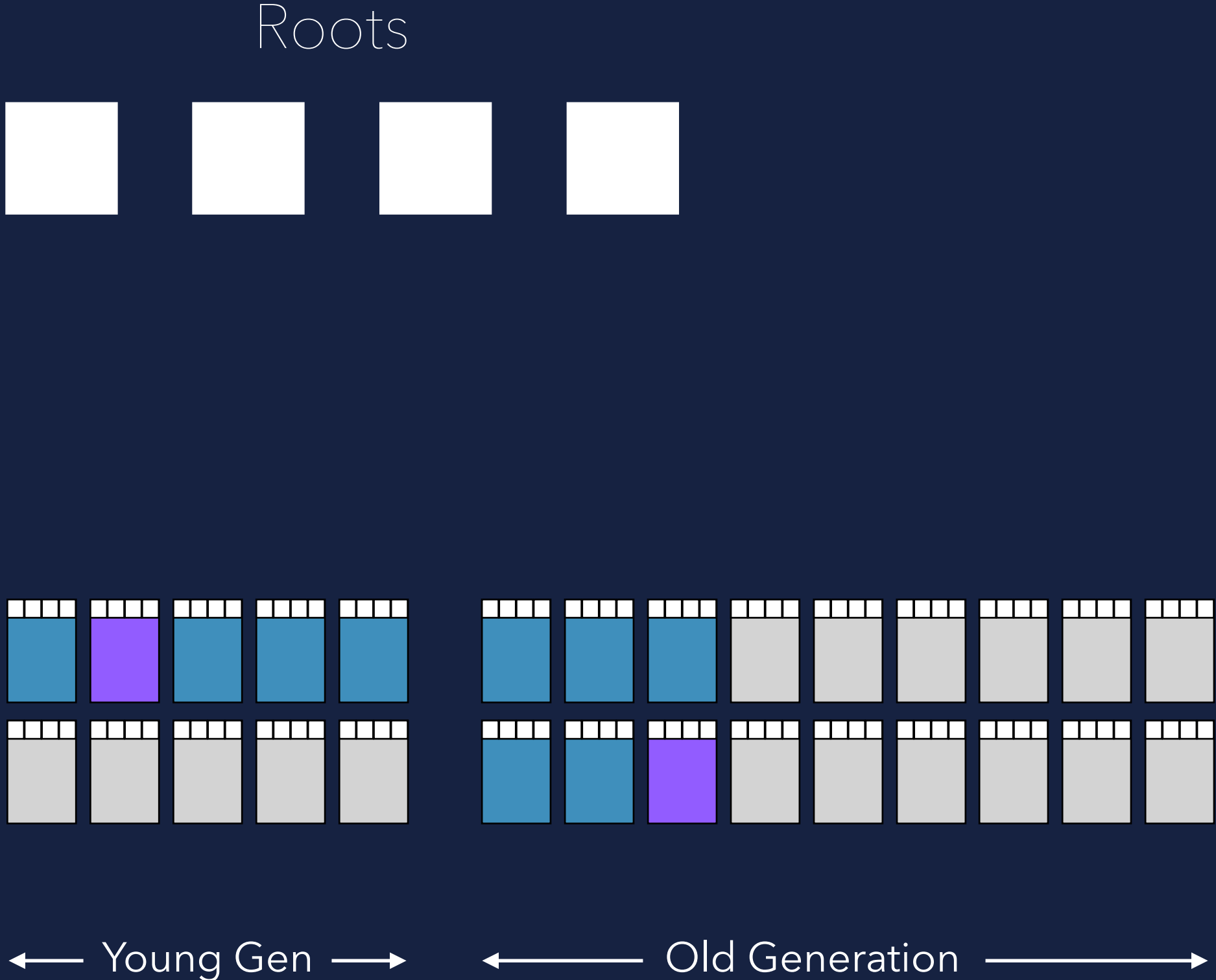
Intergenerational References

REMEMBERED SET

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

REMEMBERED SET

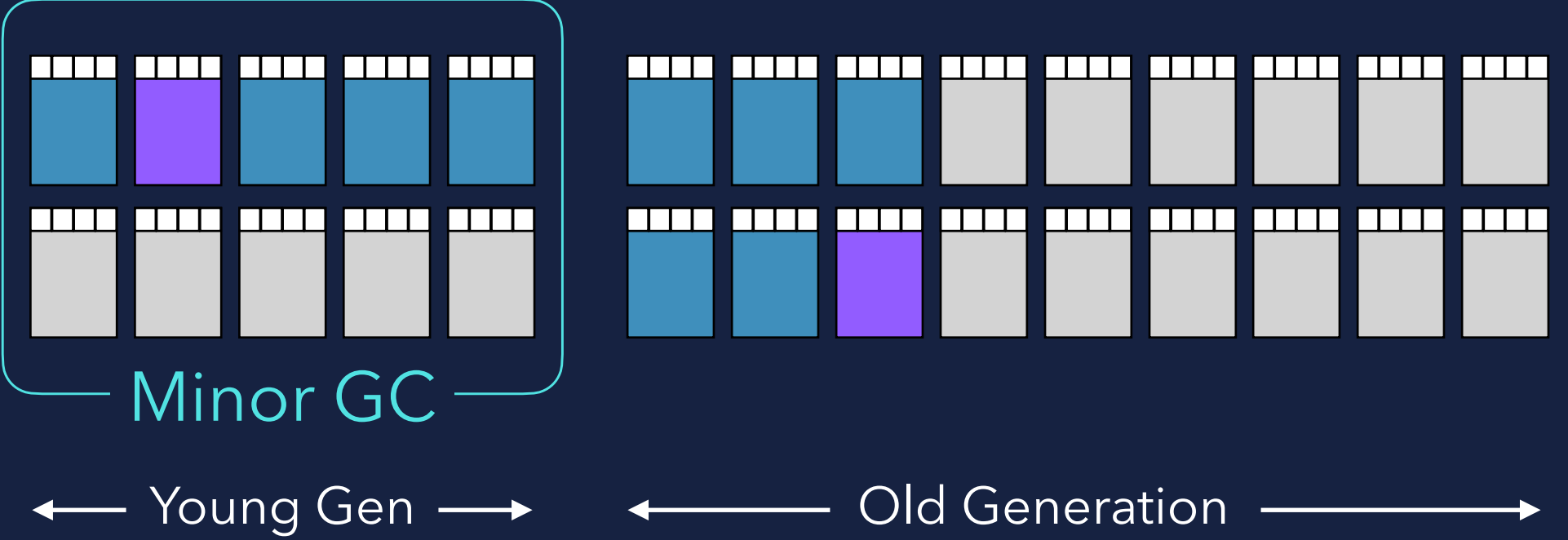
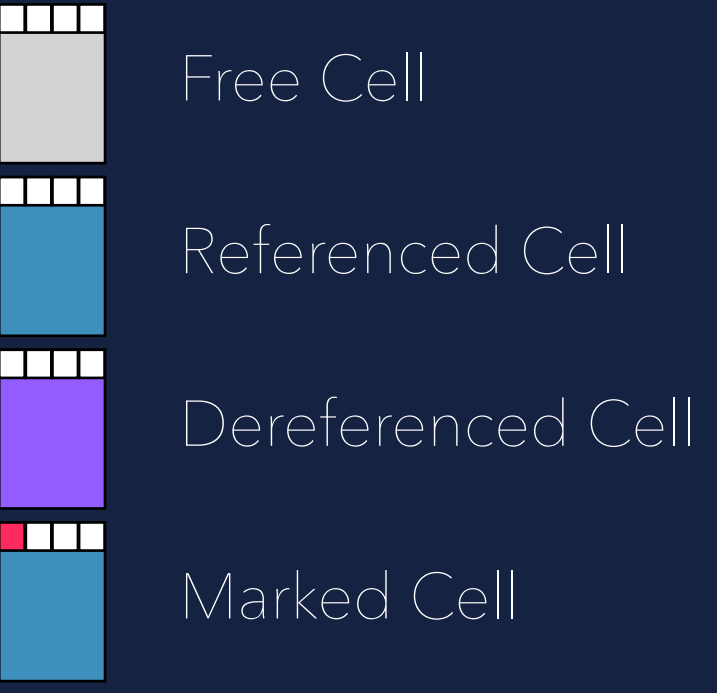
Also known as Card Table



- Free Cell
- Referenced Cell
- Dereferenced Cell
- Marked Cell

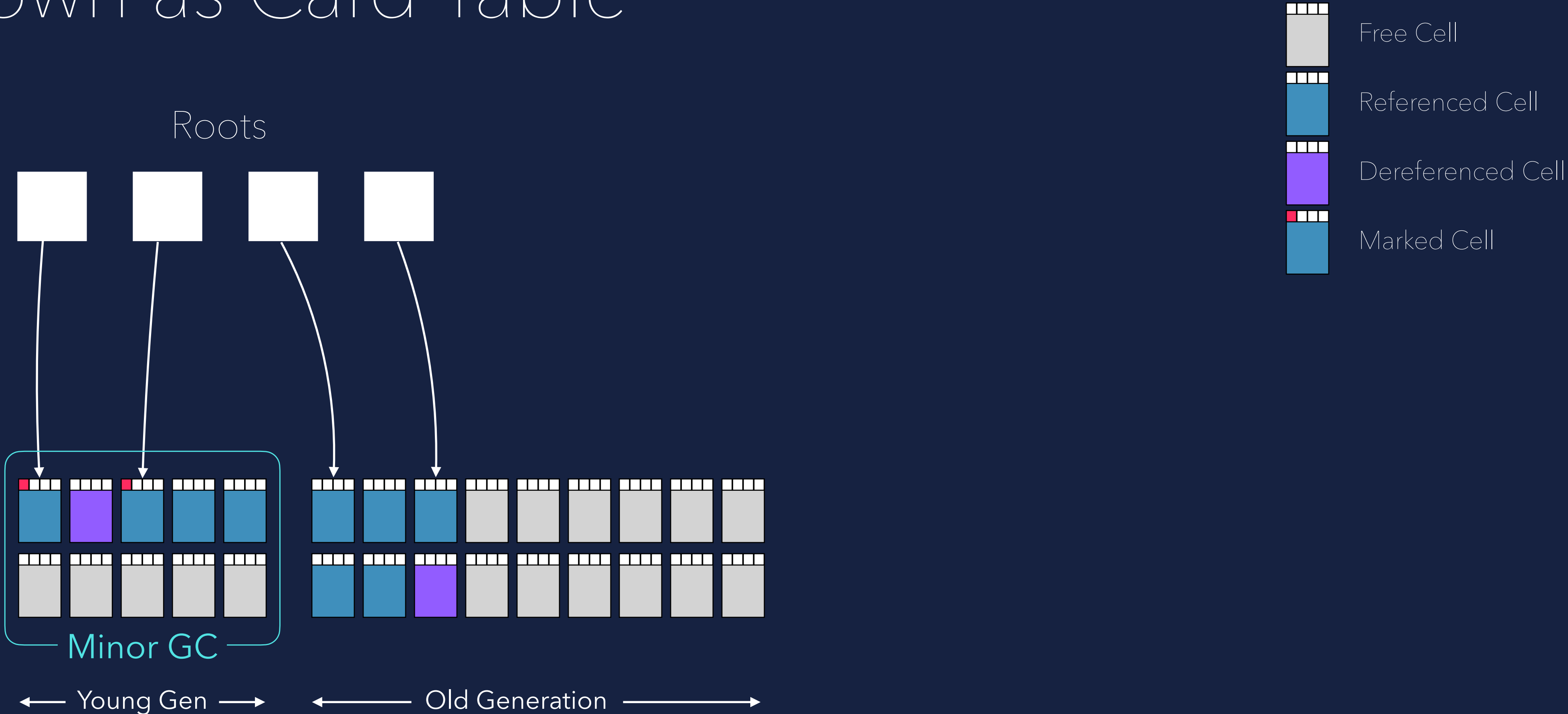
REMEMBERED SET

Also known as Card Table



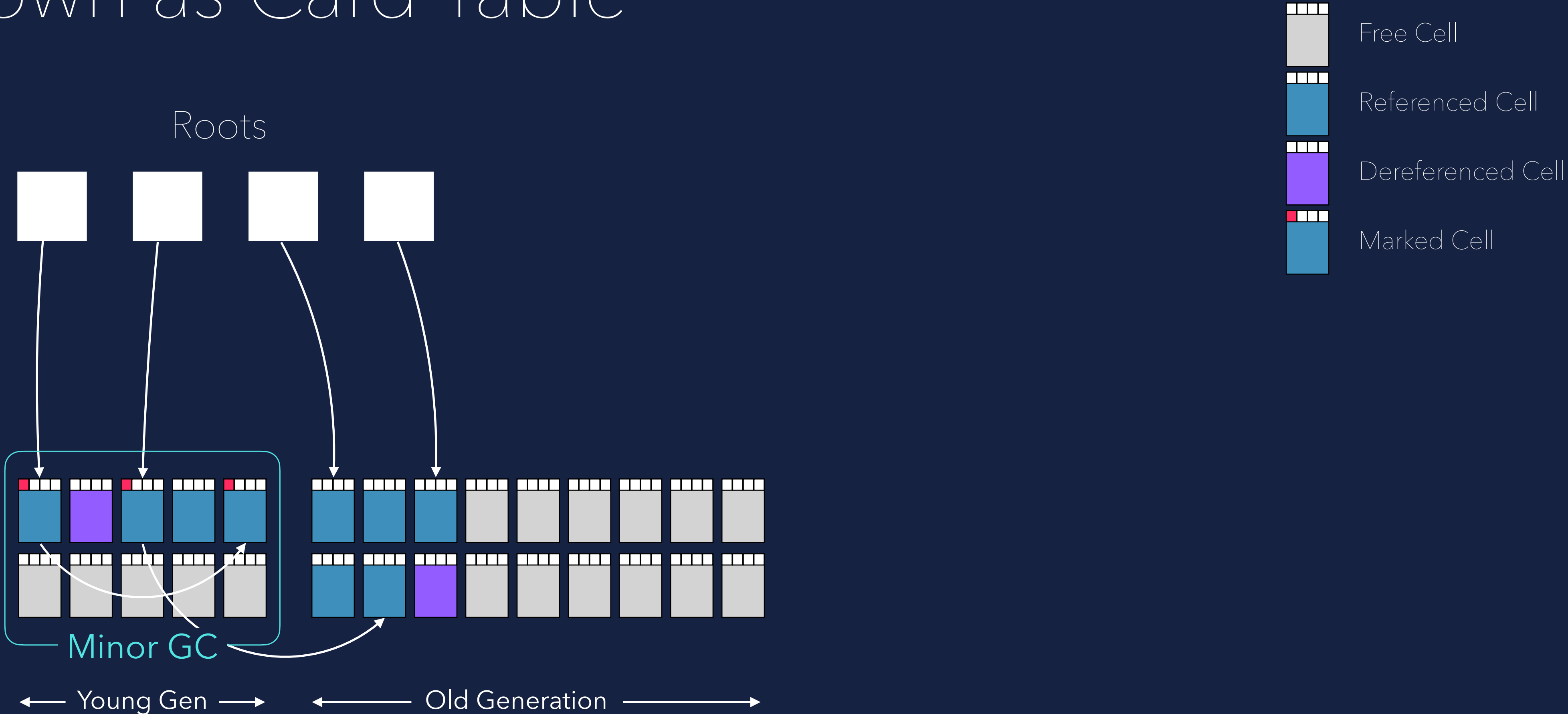
REMEMBERED SET

Also known as Card Table



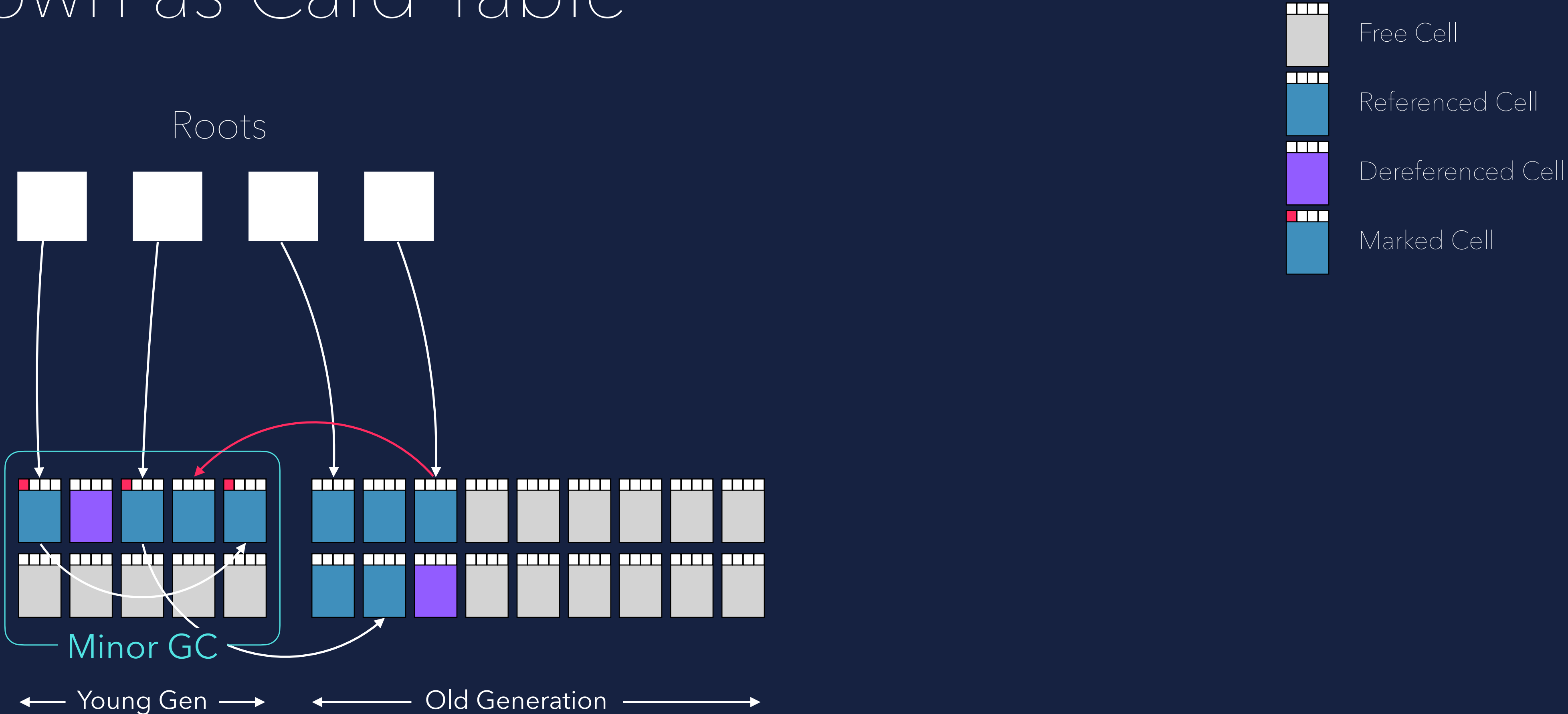
REMEMBERED SET

Also known as Card Table



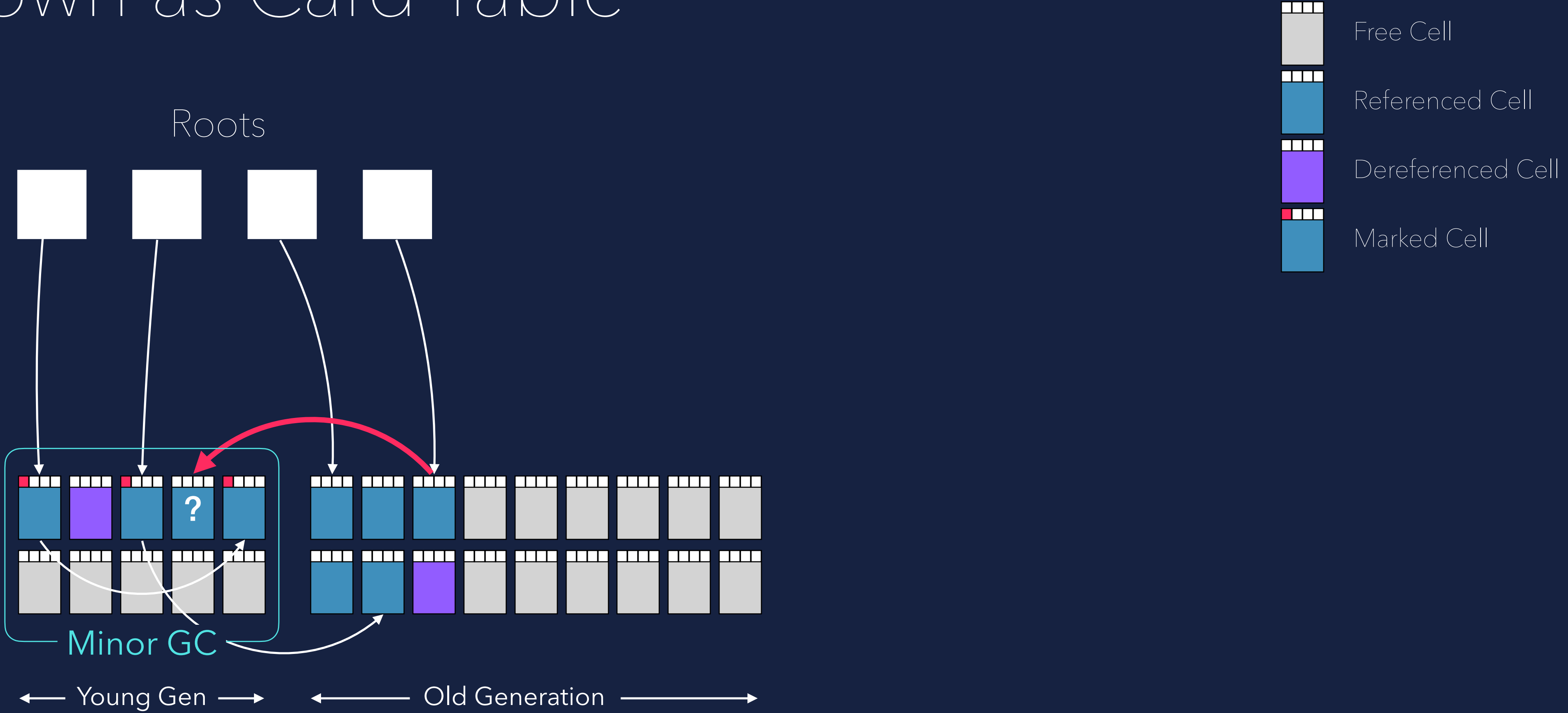
REMEMBERED SET

Also known as Card Table



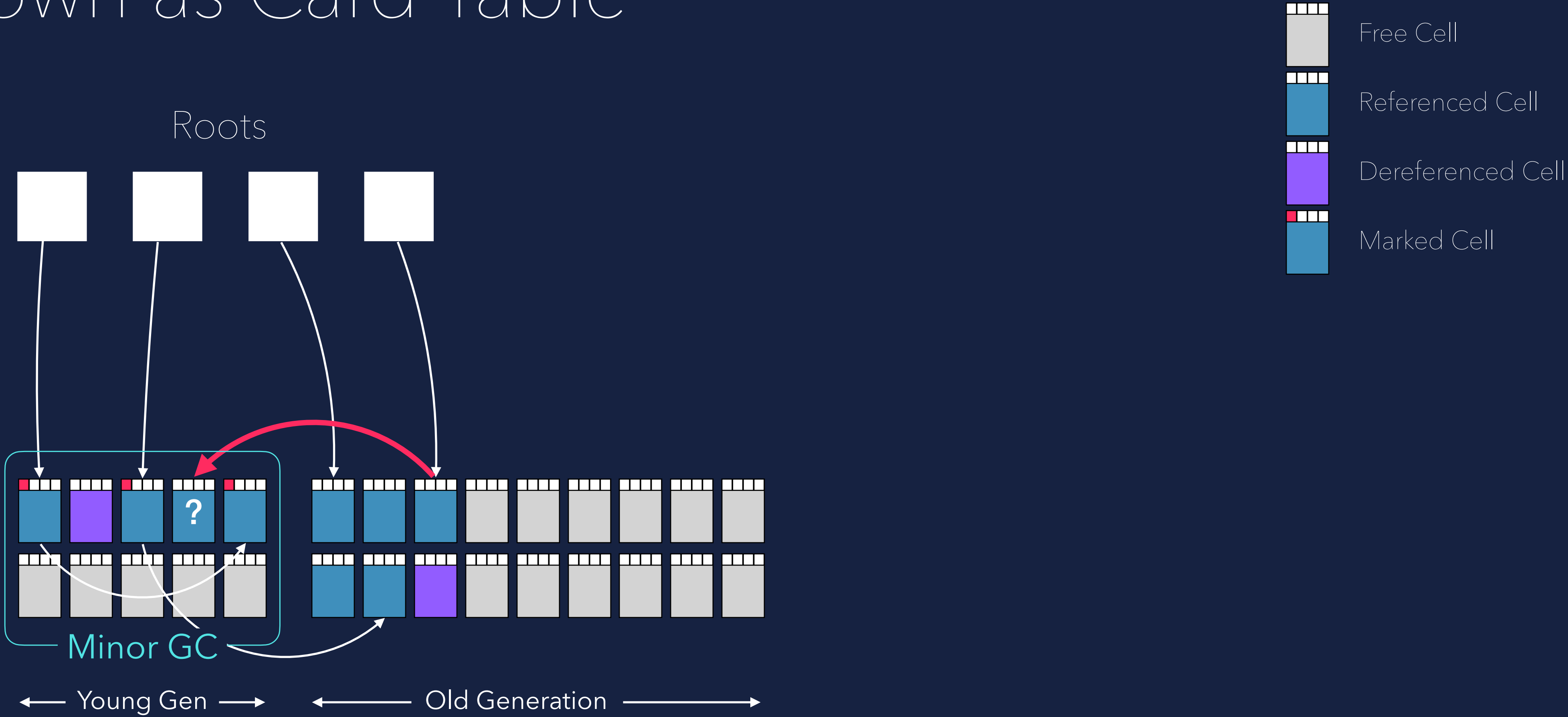
REMEMBERED SET

Also known as Card Table



REMEMBERED SET

Also known as Card Table

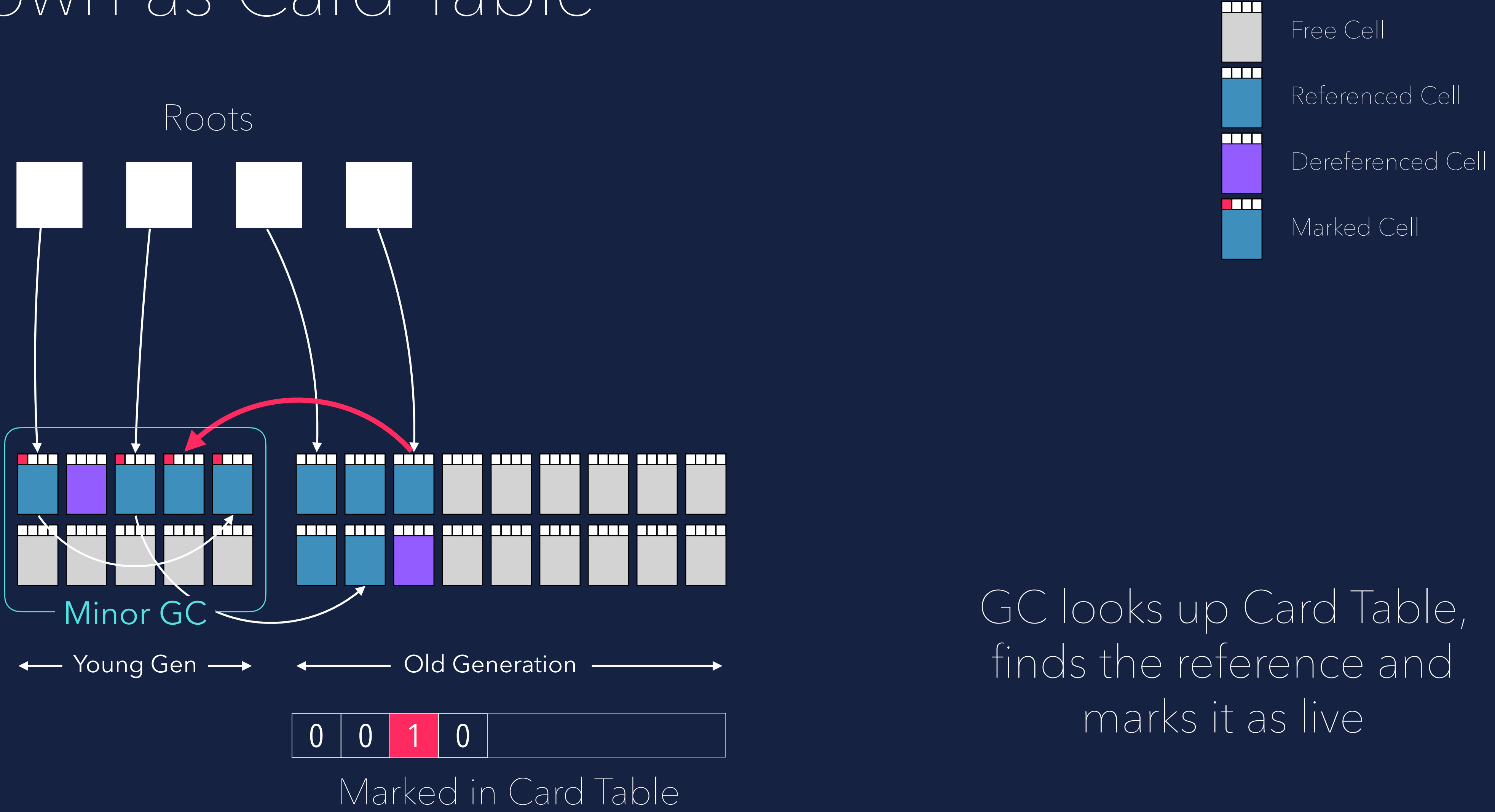


0	0	1	0
---	---	---	---

Marked in Card Table

REMEMBERED SET

Also known as Card Table



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

CONCURRENT COLLECTION ?

CONCURRENENCY

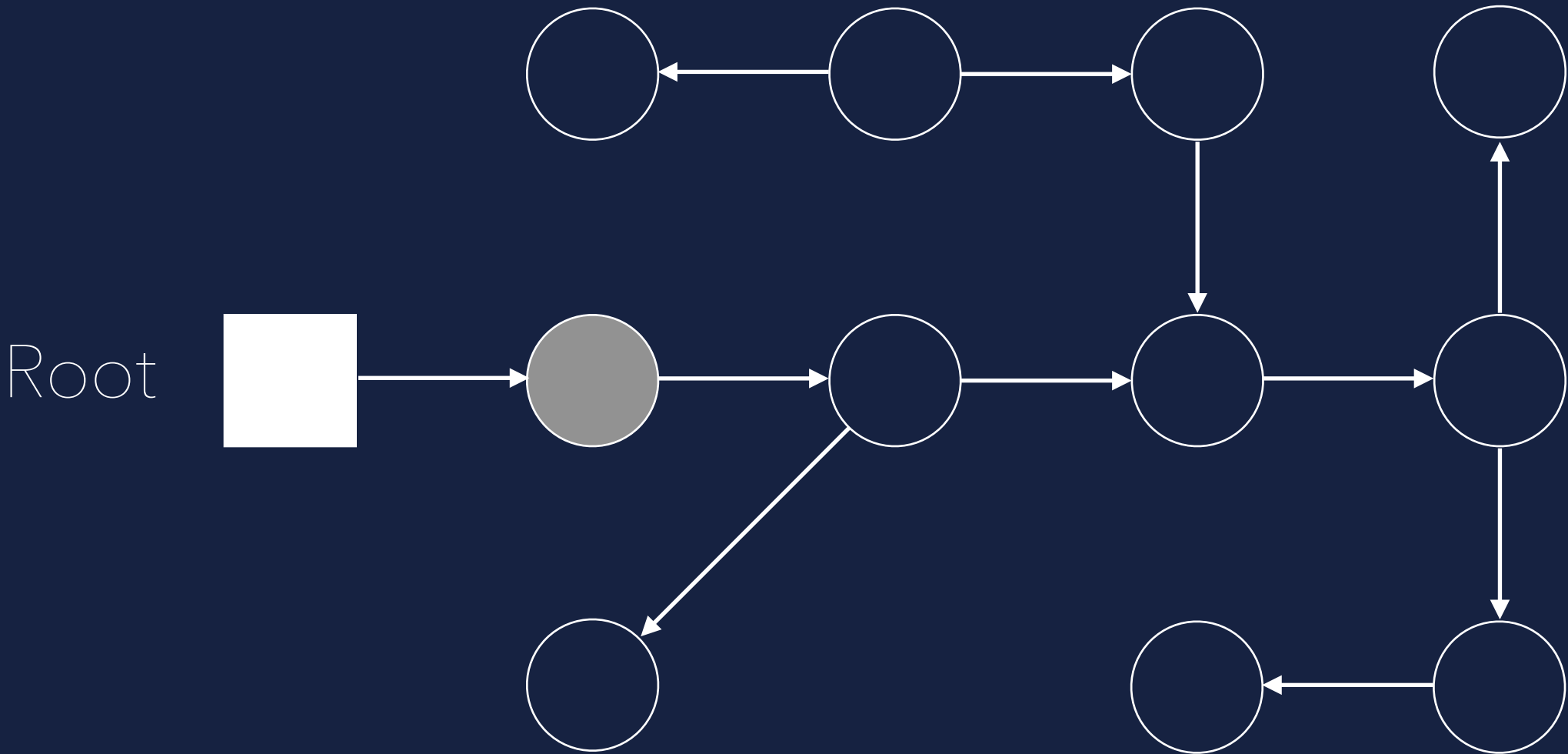
IS

HARD...

CONCURRENT MARKING

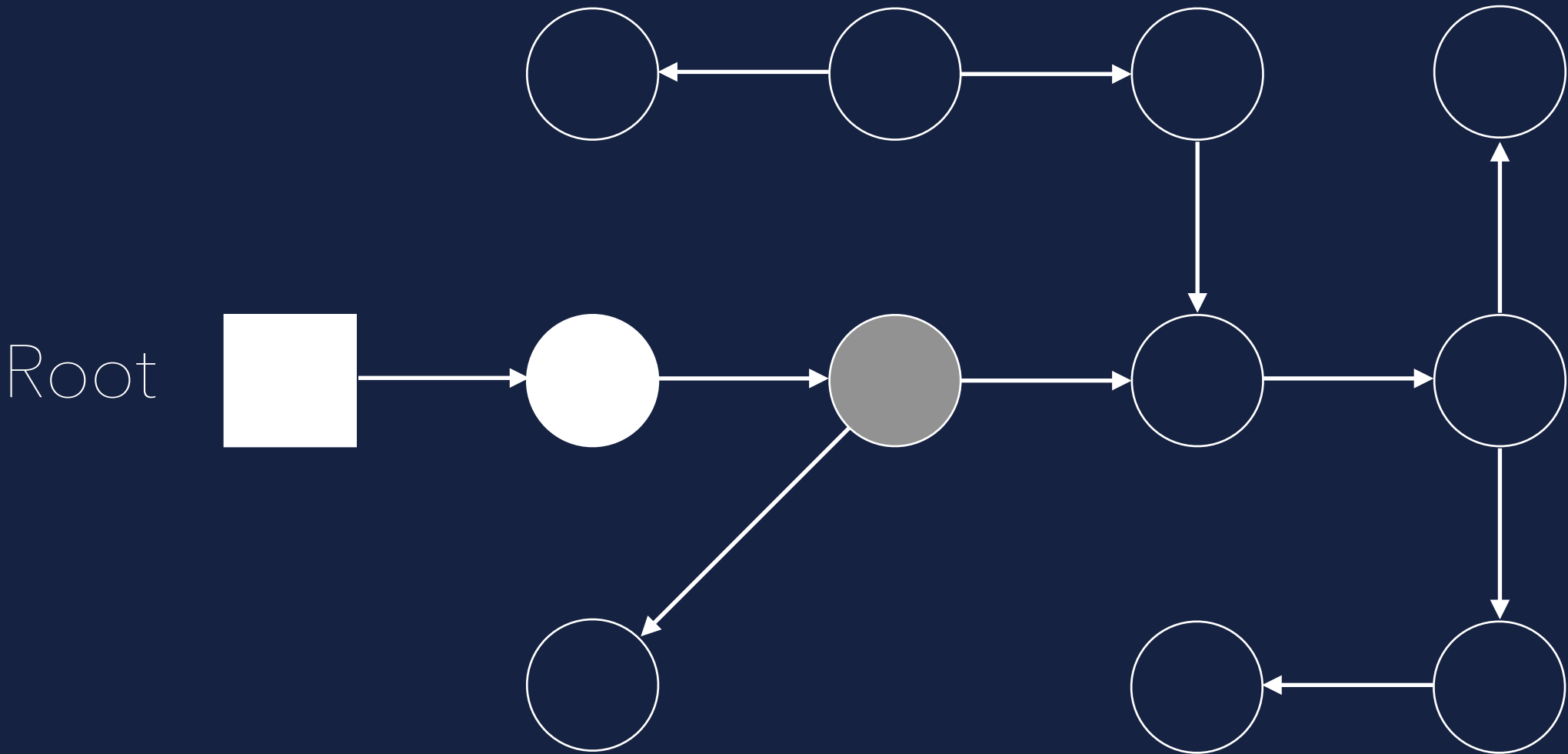
CONCURRENCY IS HARD...

Stop the World marking



CONCURRENCY IS HARD...

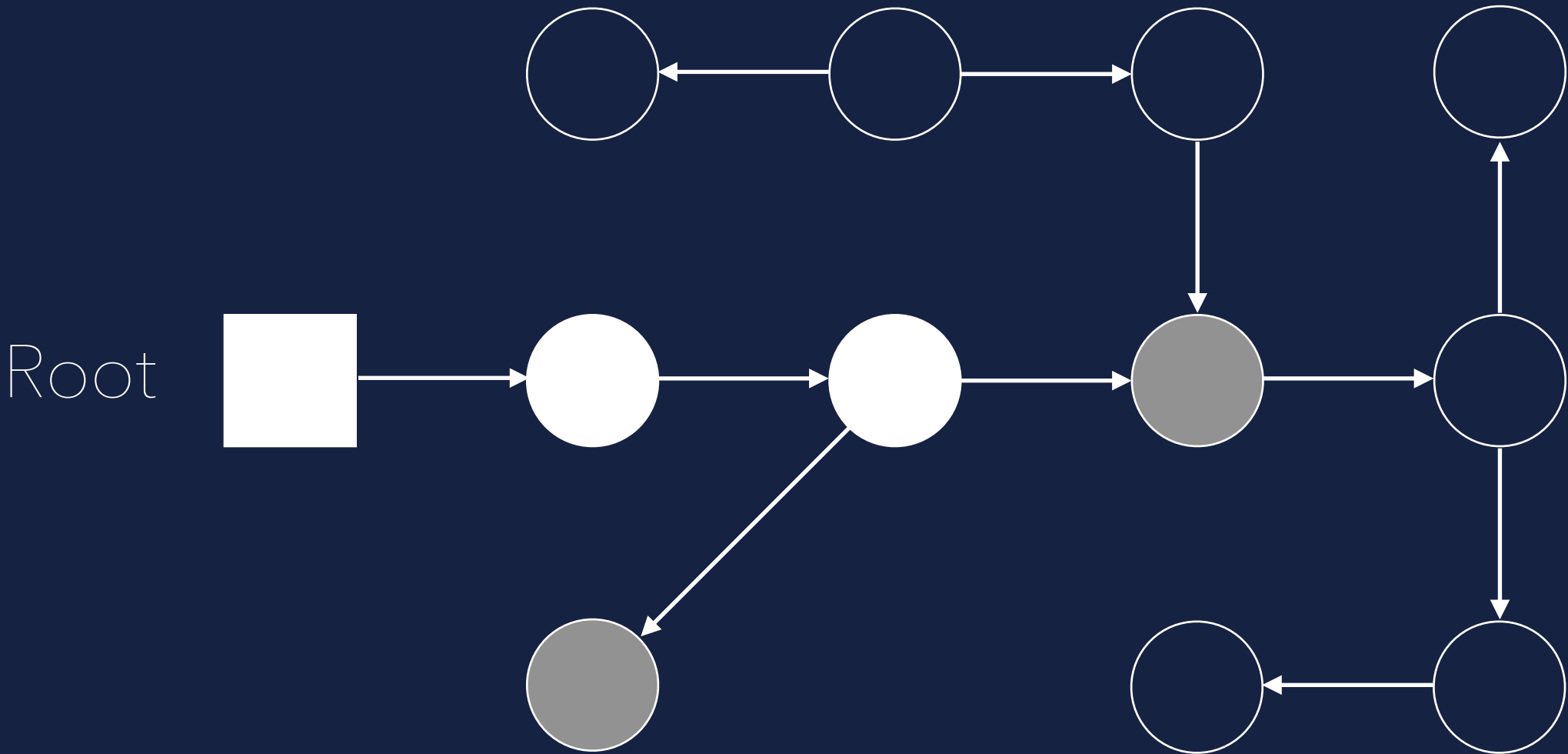
Stop the World marking



- Reachable
- Live
- Not visited

CONCURRENCY IS HARD...

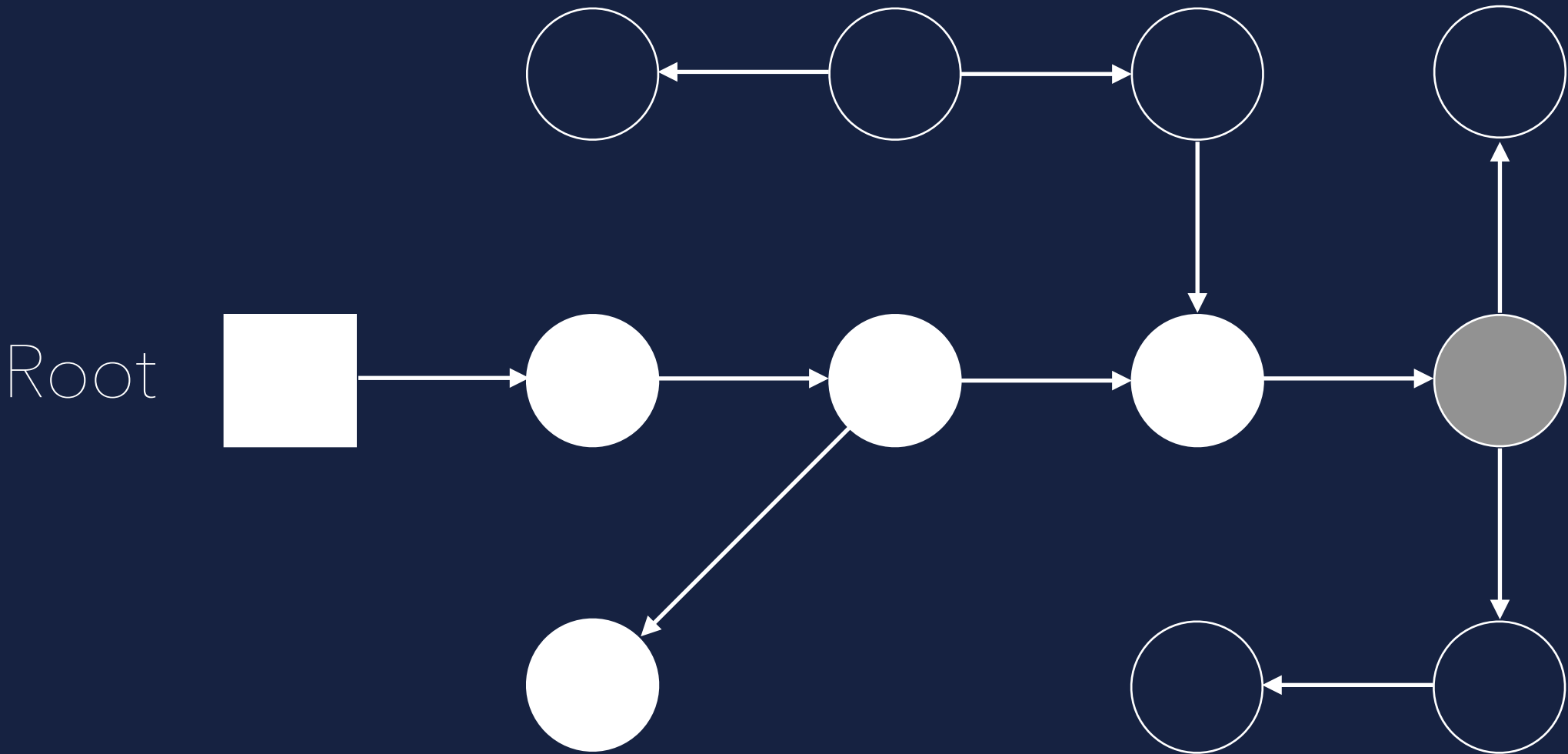
Stop the World marking






- Reachable
- Live
- Not visited

CONCURRENCY IS HARD...

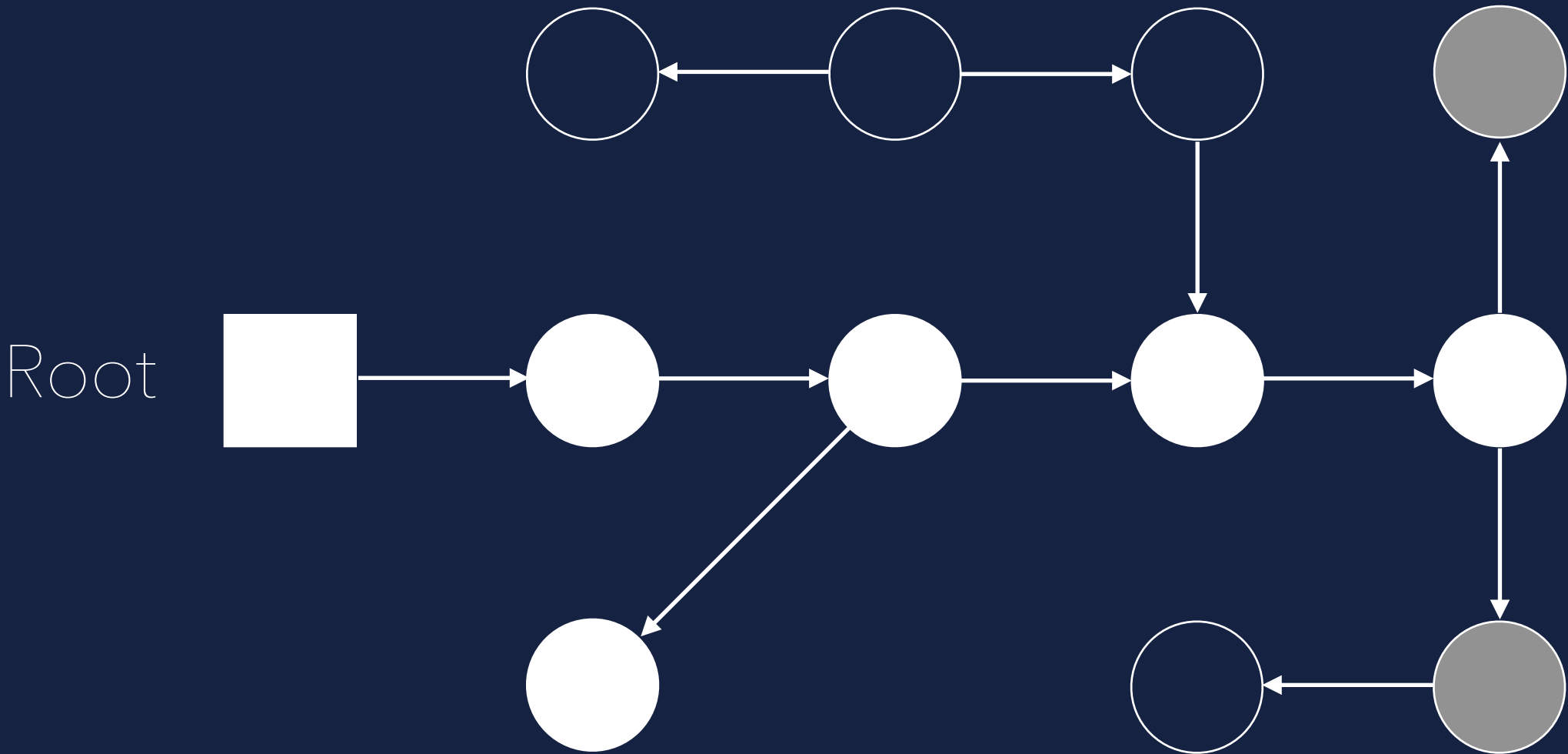
Stop the World marking






-  Reachable
-  Live
-  Not visited

CONCURRENCY IS HARD...

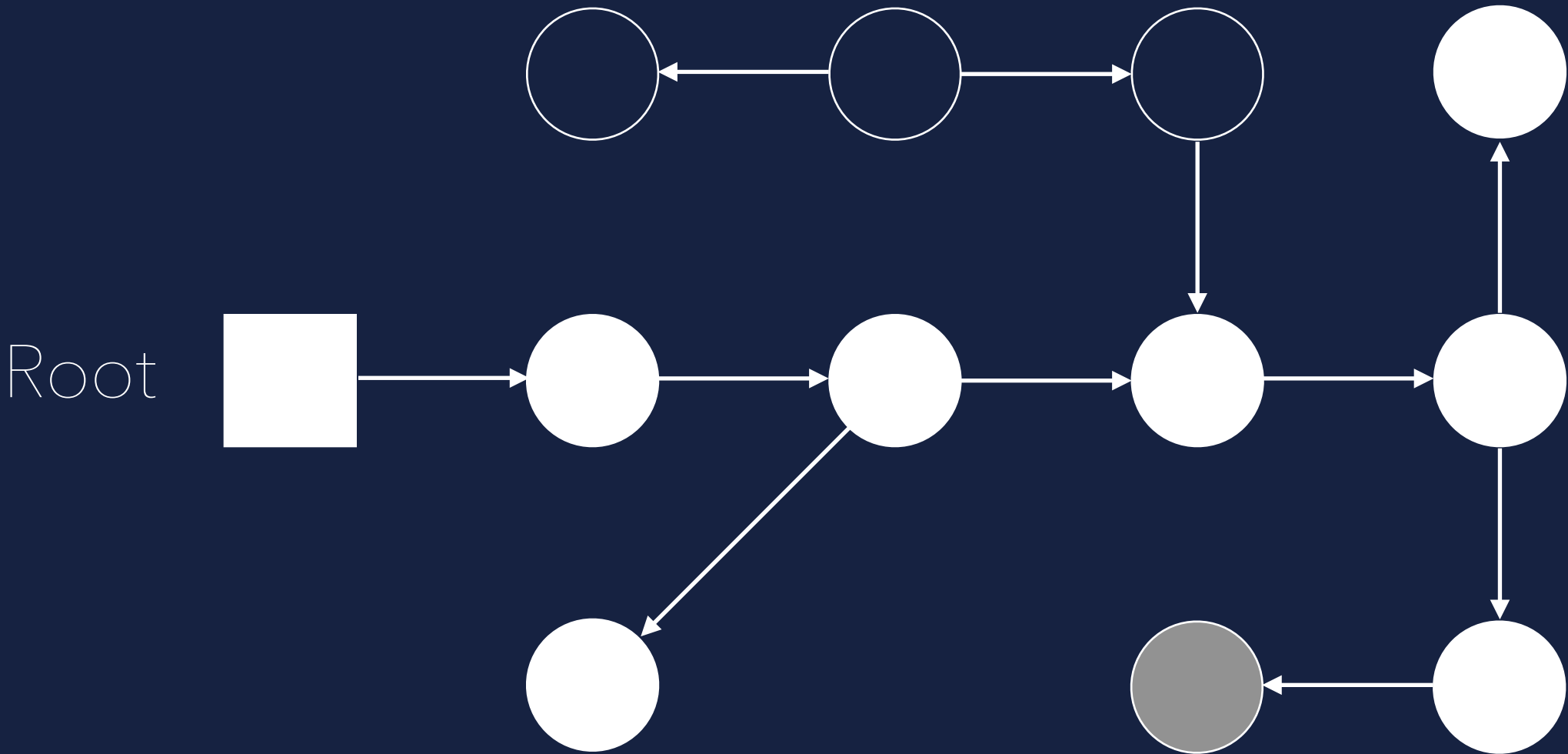
Stop the World marking



-  Reachable
-  Live
-  Not visited

CONCURRENCY IS HARD...

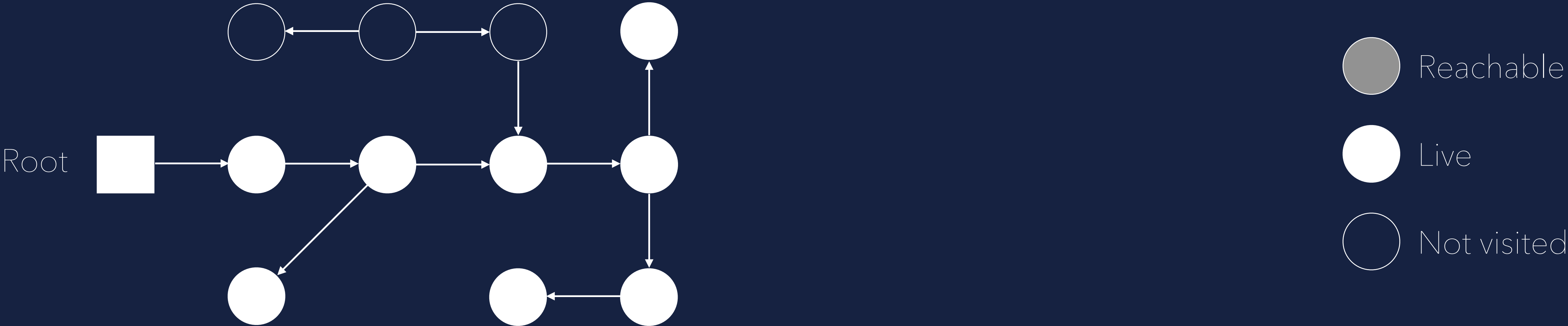
Stop the World marking



- Reachable
- Live
- Not visited

CONCURRENCY IS HARD...

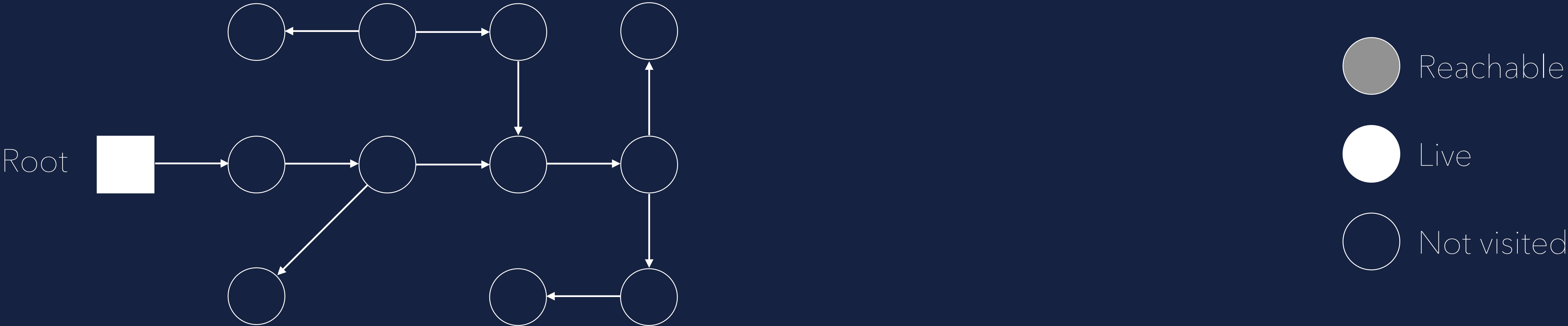
Stop the World marking



All reachable objects are marked live

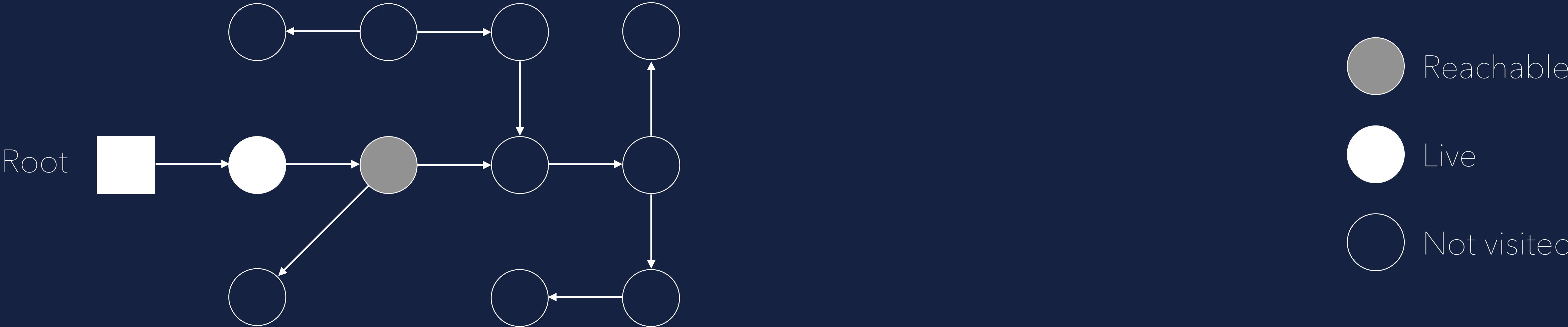
CONCURRENCY IS HARD...

Concurrent Marking



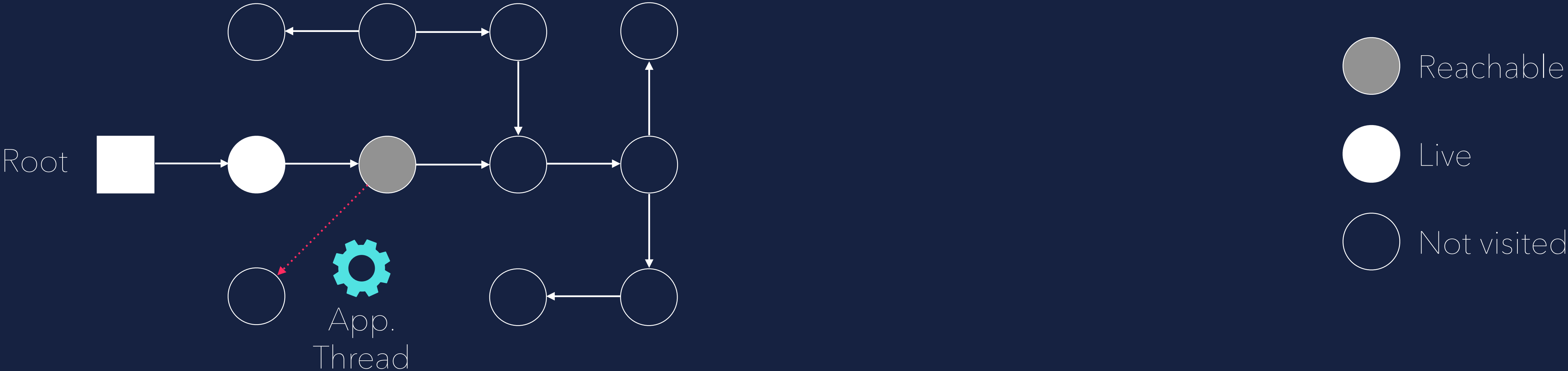
CONCURRENCY IS HARD...

Concurrent Marking



CONCURRENCY IS HARD...

Concurrent Marking



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

CONCURRENCY IS HARD...

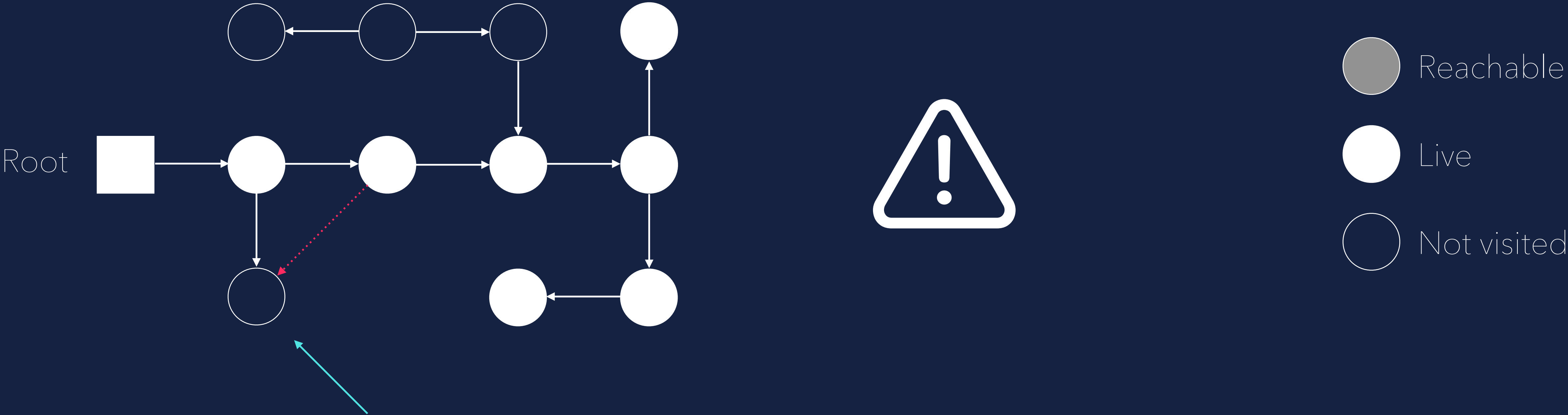
Concurrent Marking



Won't be detected by the Garbage Collector !

CONCURRENCY IS HARD...

Concurrent Marking



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

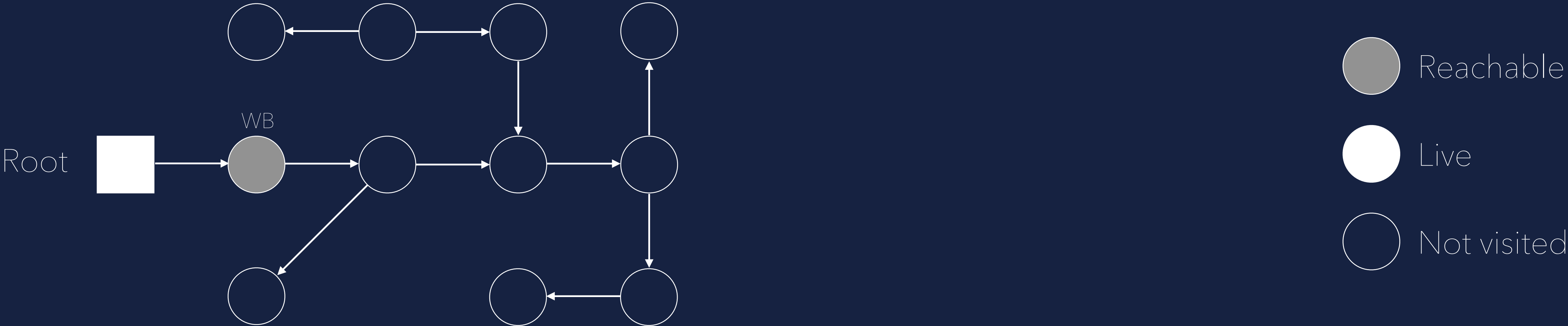
BARRIERS

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 Remembered 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 be very efficient)

CONCURRENCY IS HARD...

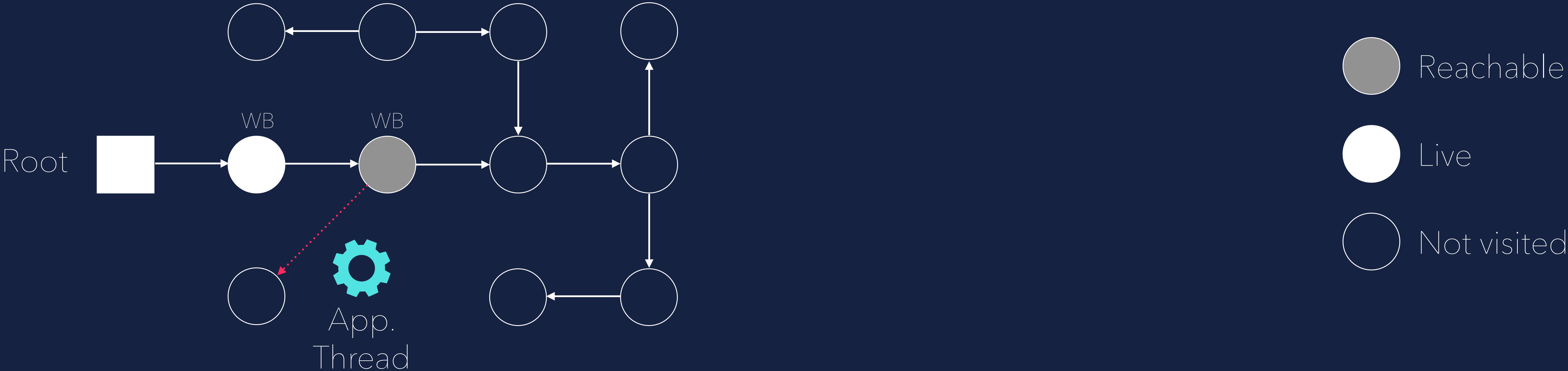
Concurrent Marking using Write Barriers



Collector starts marking objects

CONCURRENCY IS HARD...

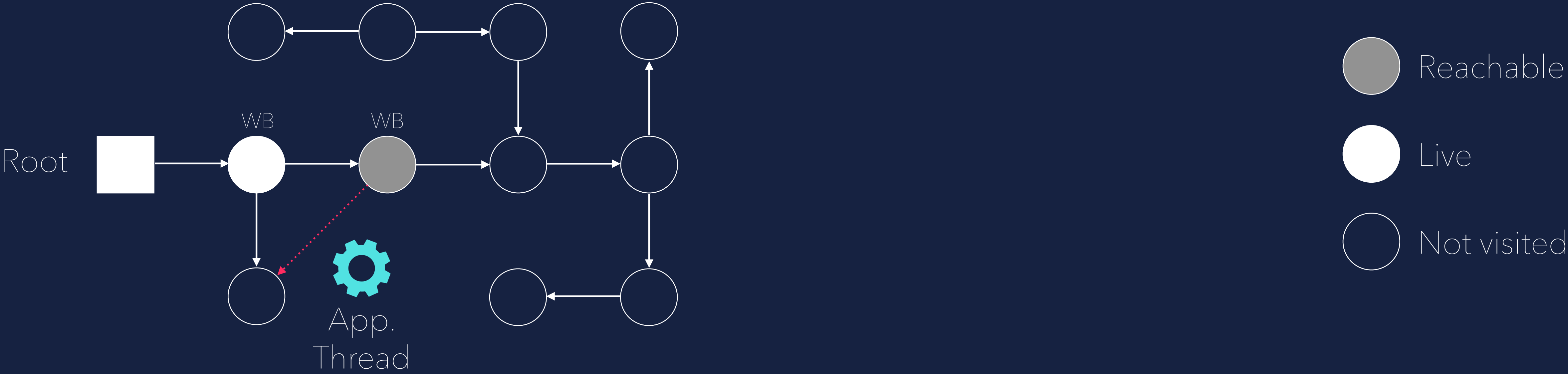
Concurrent Marking using Write Barriers



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

CONCURRENCY IS HARD...

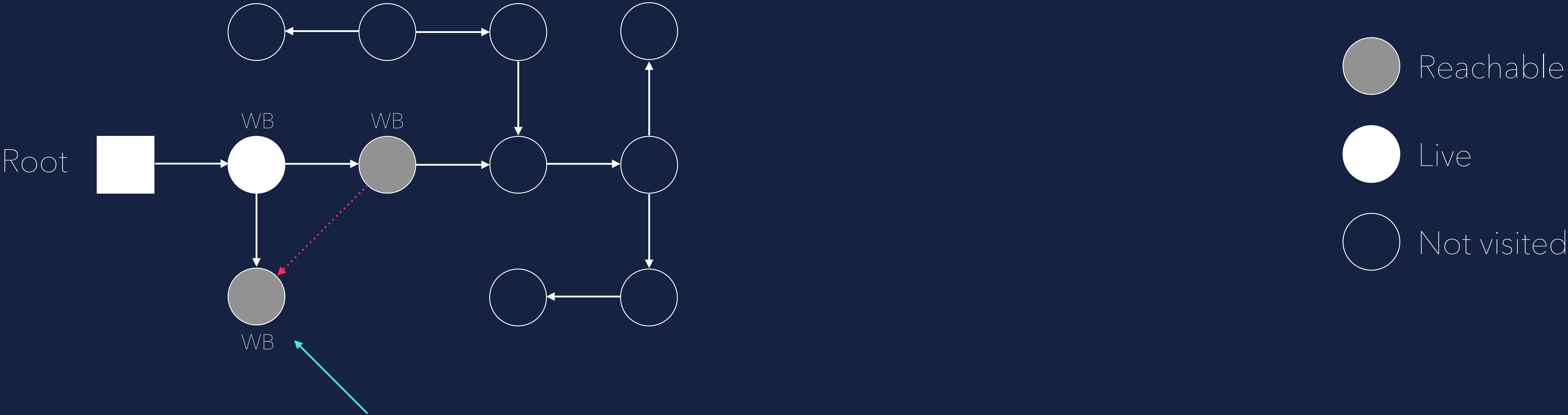
Concurrent Marking using Write Barriers



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

CONCURRENCY IS HARD...

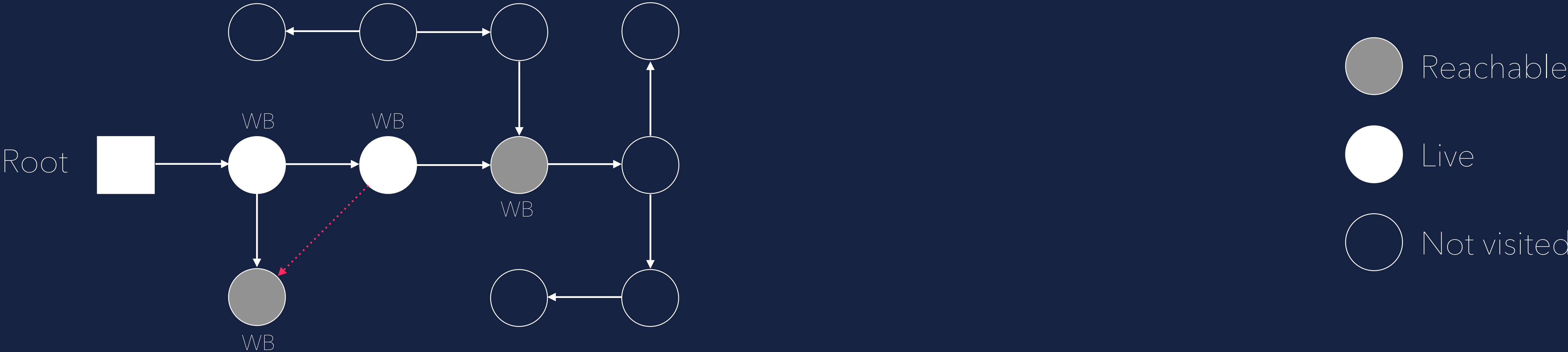
Concurrent Marking using Write Barriers



Removed references will be marked as reachable by Write Barrier

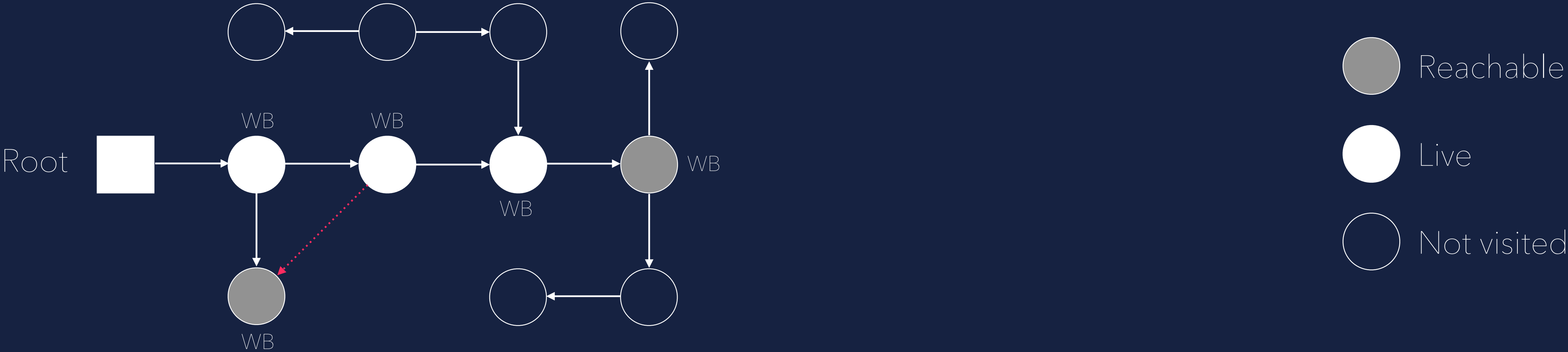
CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



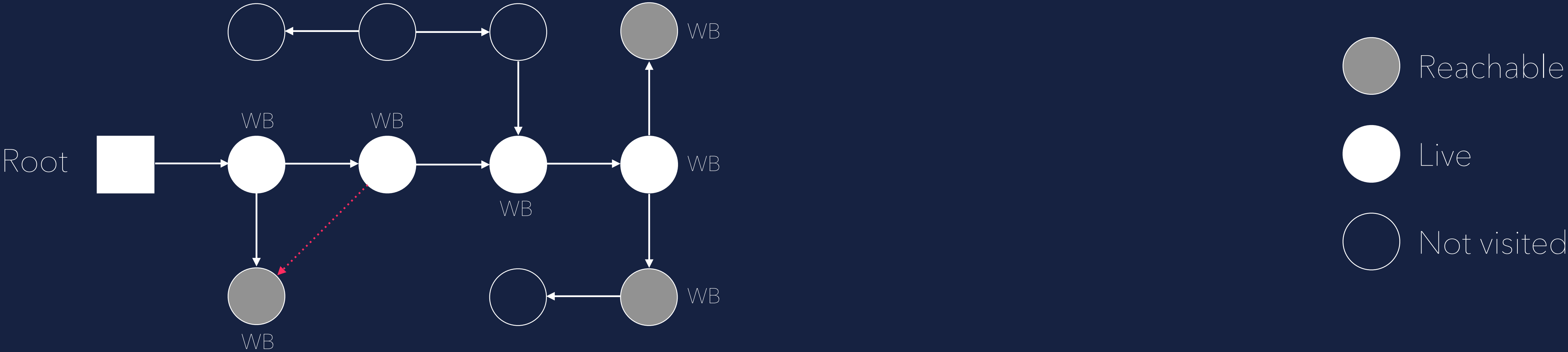
CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



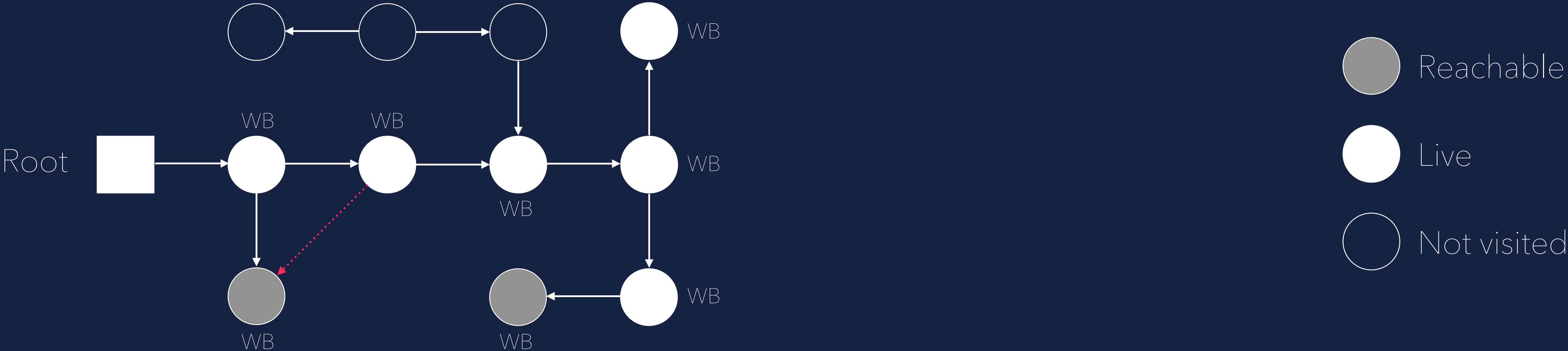
CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



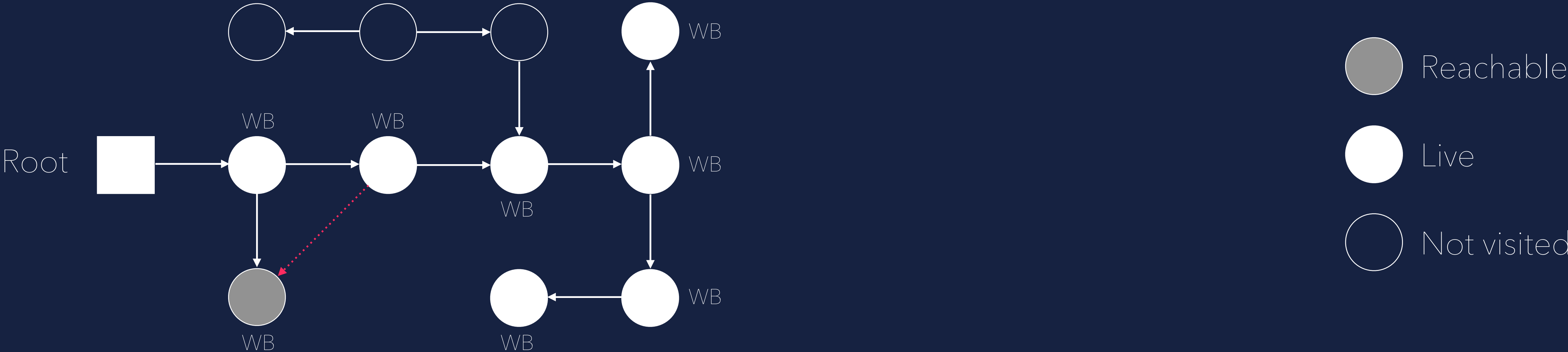
CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



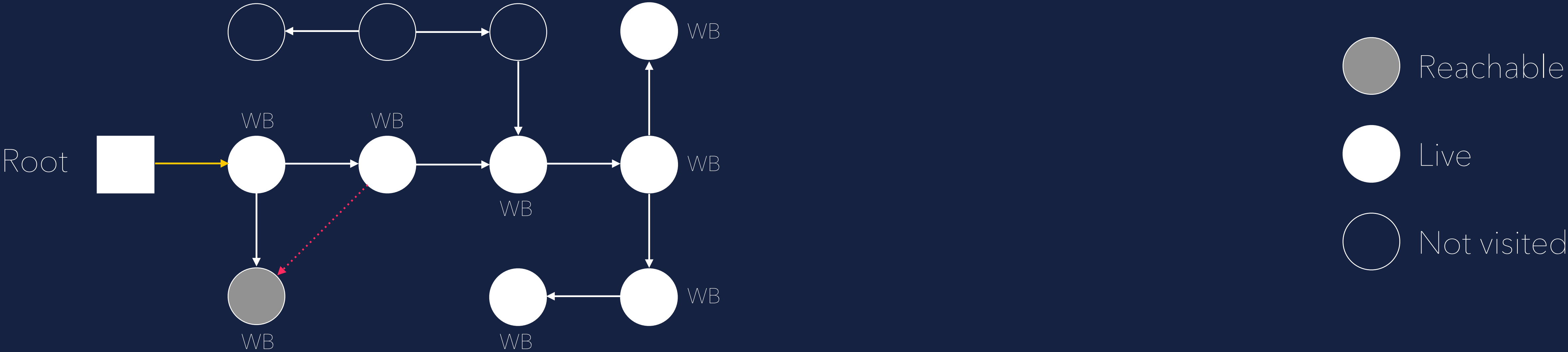
CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers



CONCURRENCY IS HARD...

Concurrent Marking using Write Barriers

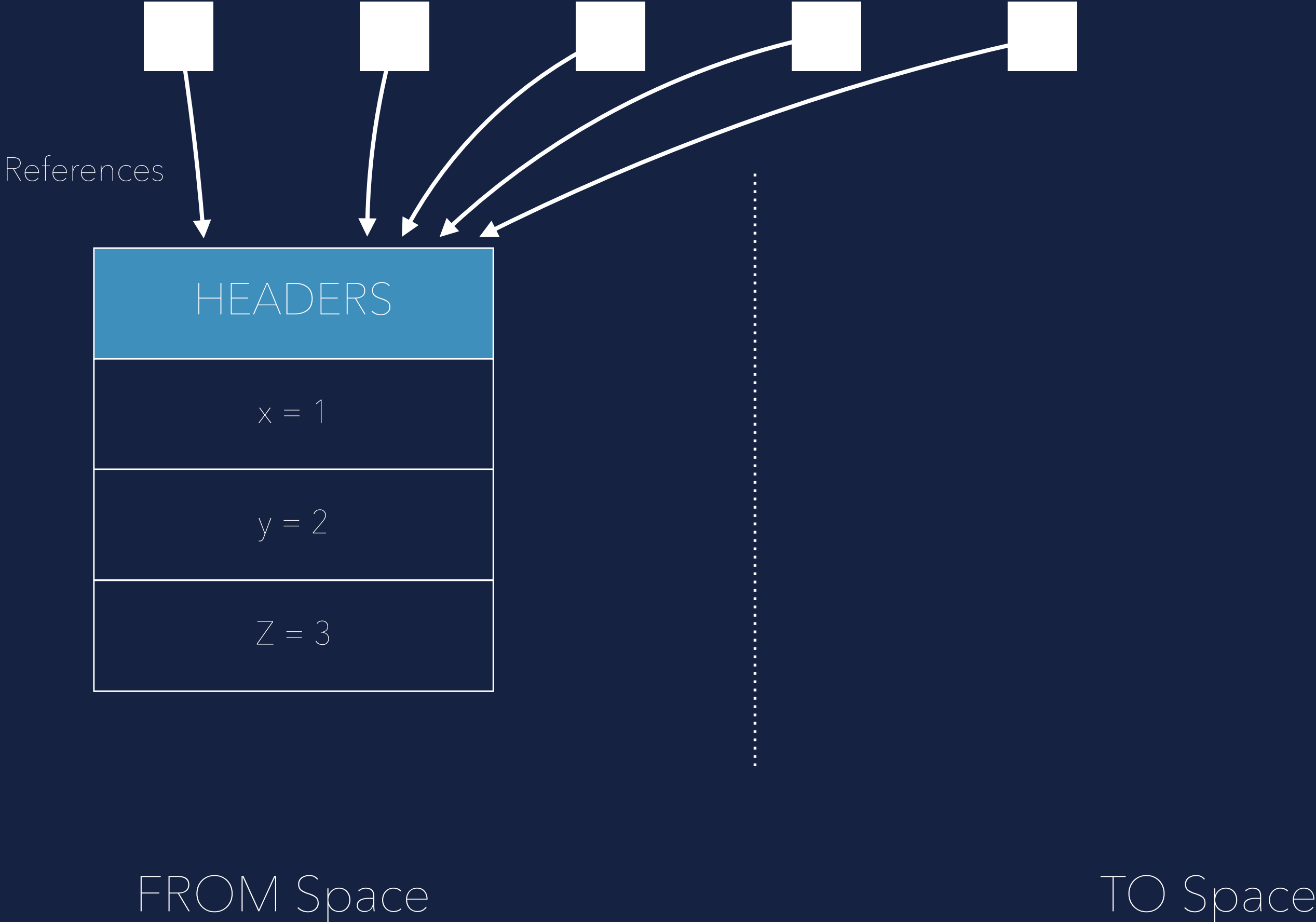


In the **Re-Marking** phase, in between marked references will be marked as live

CONCURRENT COPYING

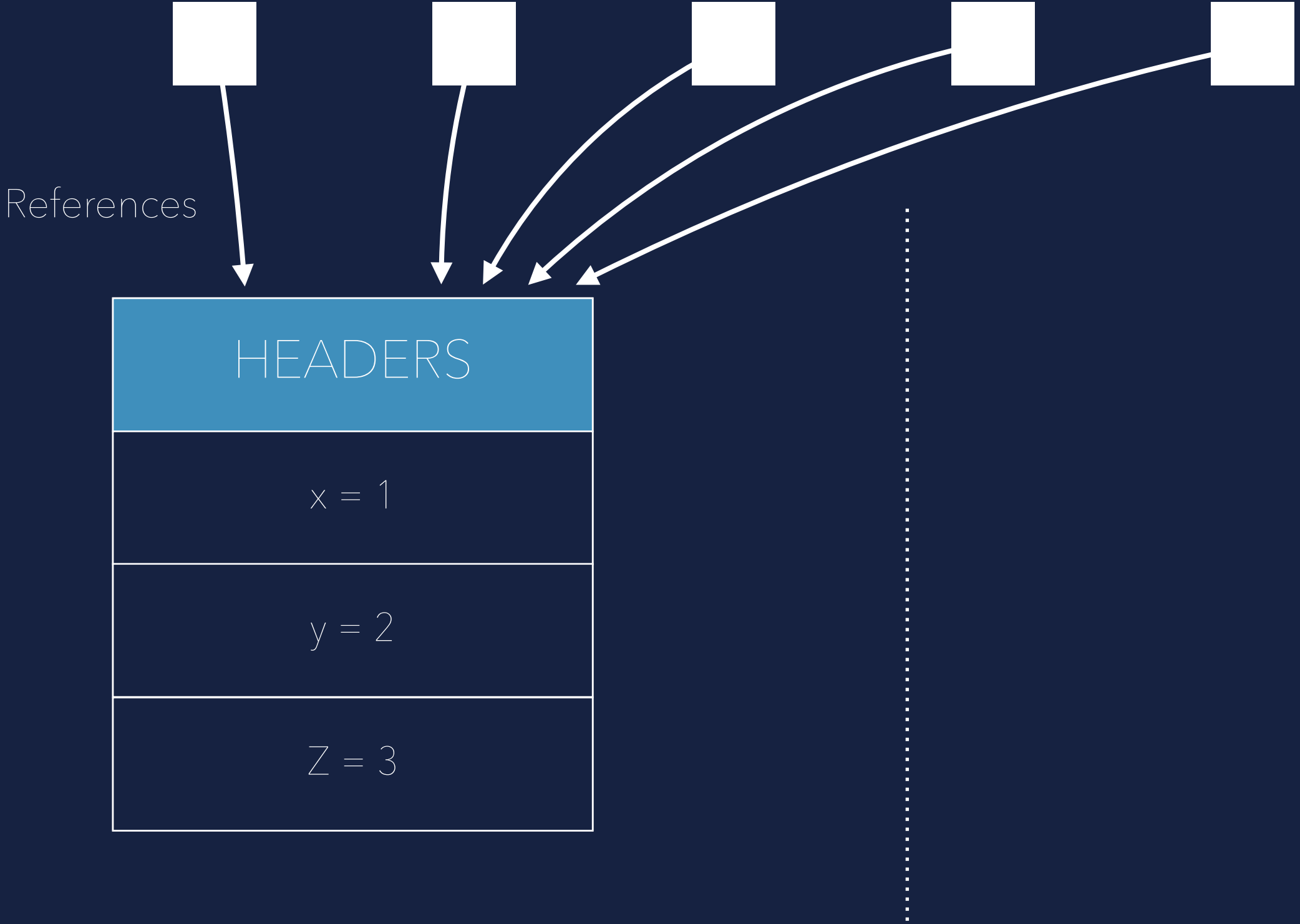
CONCURRENCY IS HARD...

Stop the world copying



CONCURRENCY IS HARD...

Stop the world copying



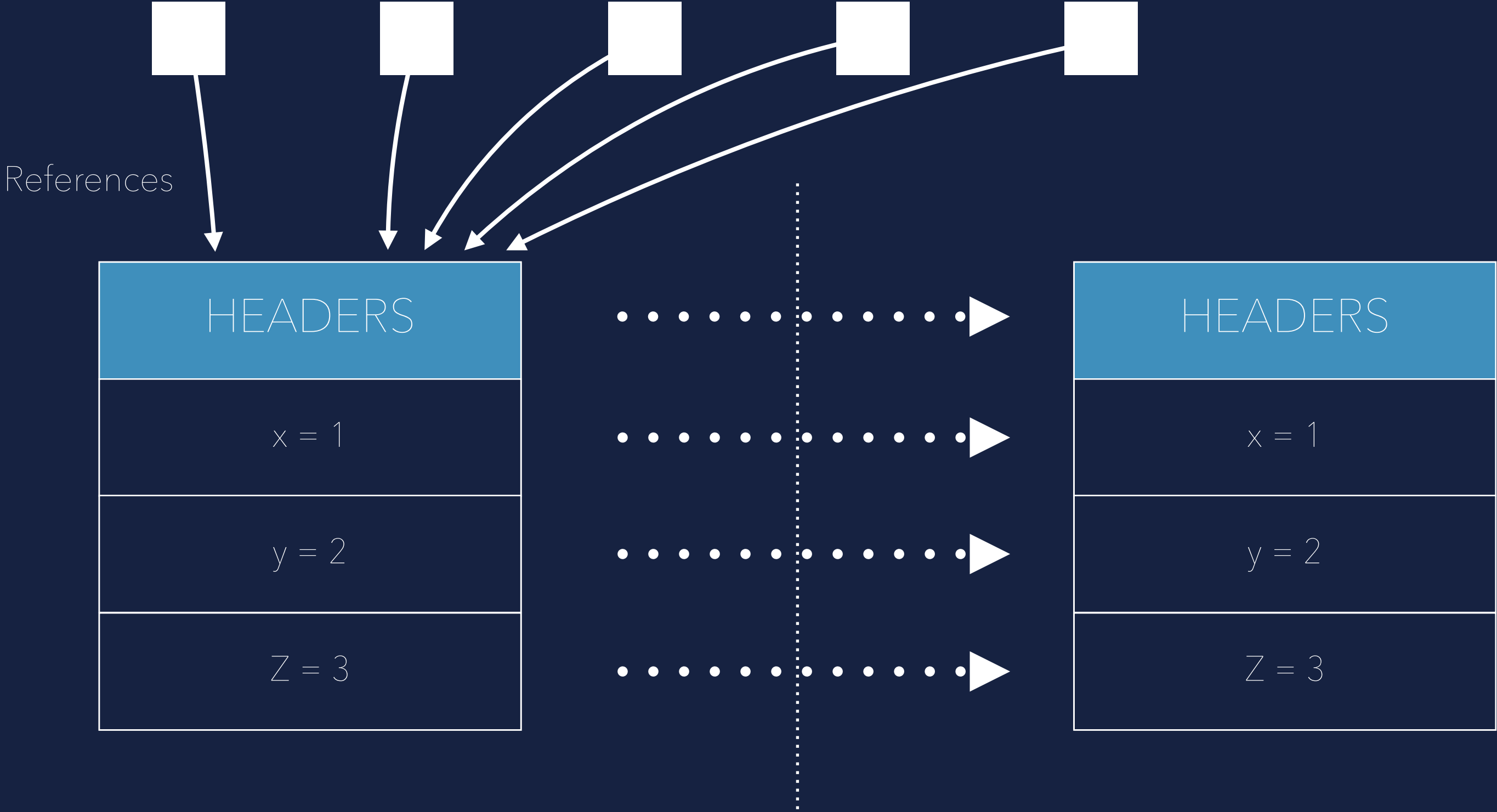
Stop the World
(the Mutator)

FROM Space

TO Space

CONCURRENCY IS HARD...

Stop the world copying



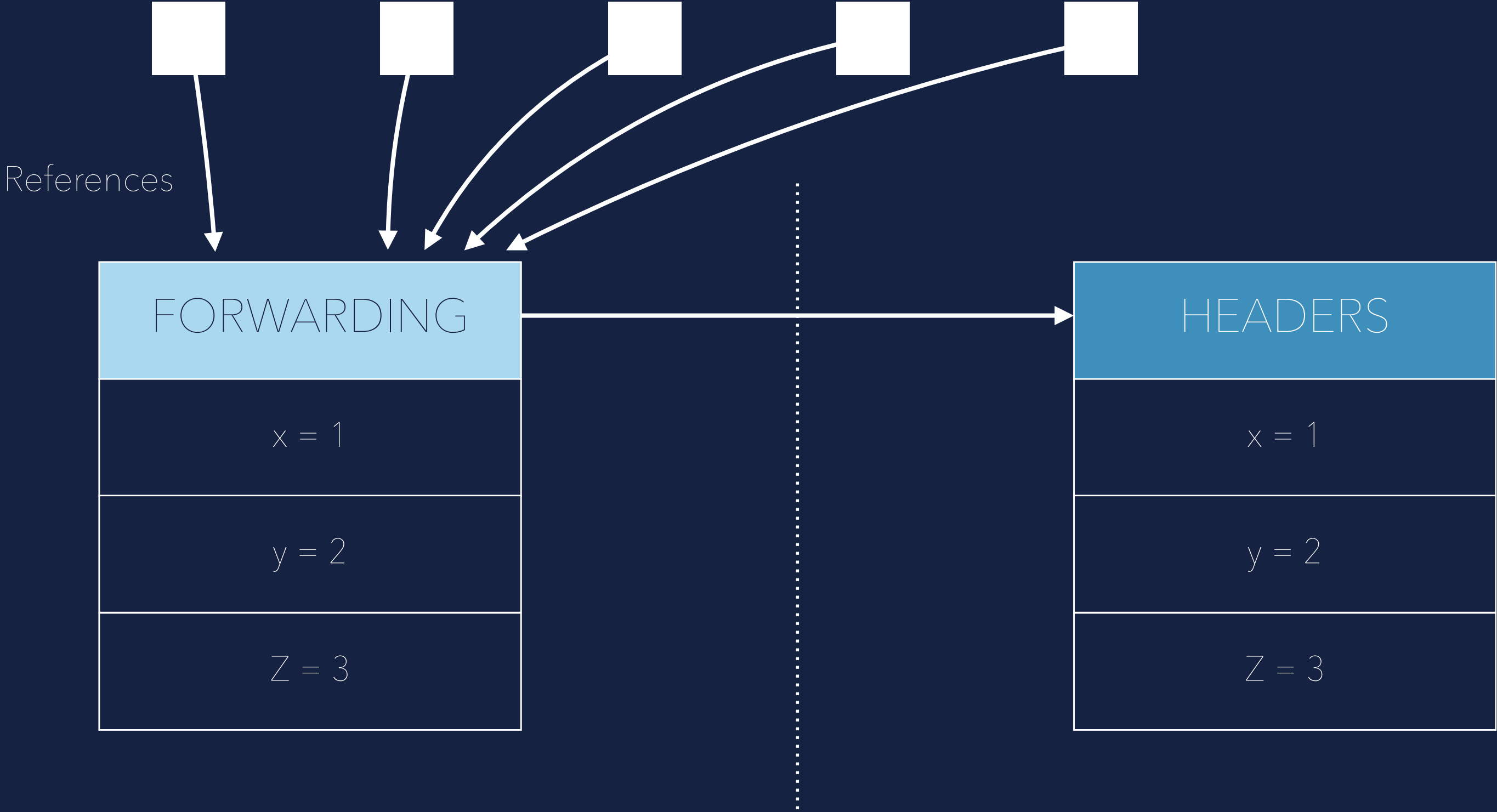
FROM Space

TO Space

Copy the Object
(Create forwarding pointer)

CONCURRENCY IS HARD...

Stop the world copying



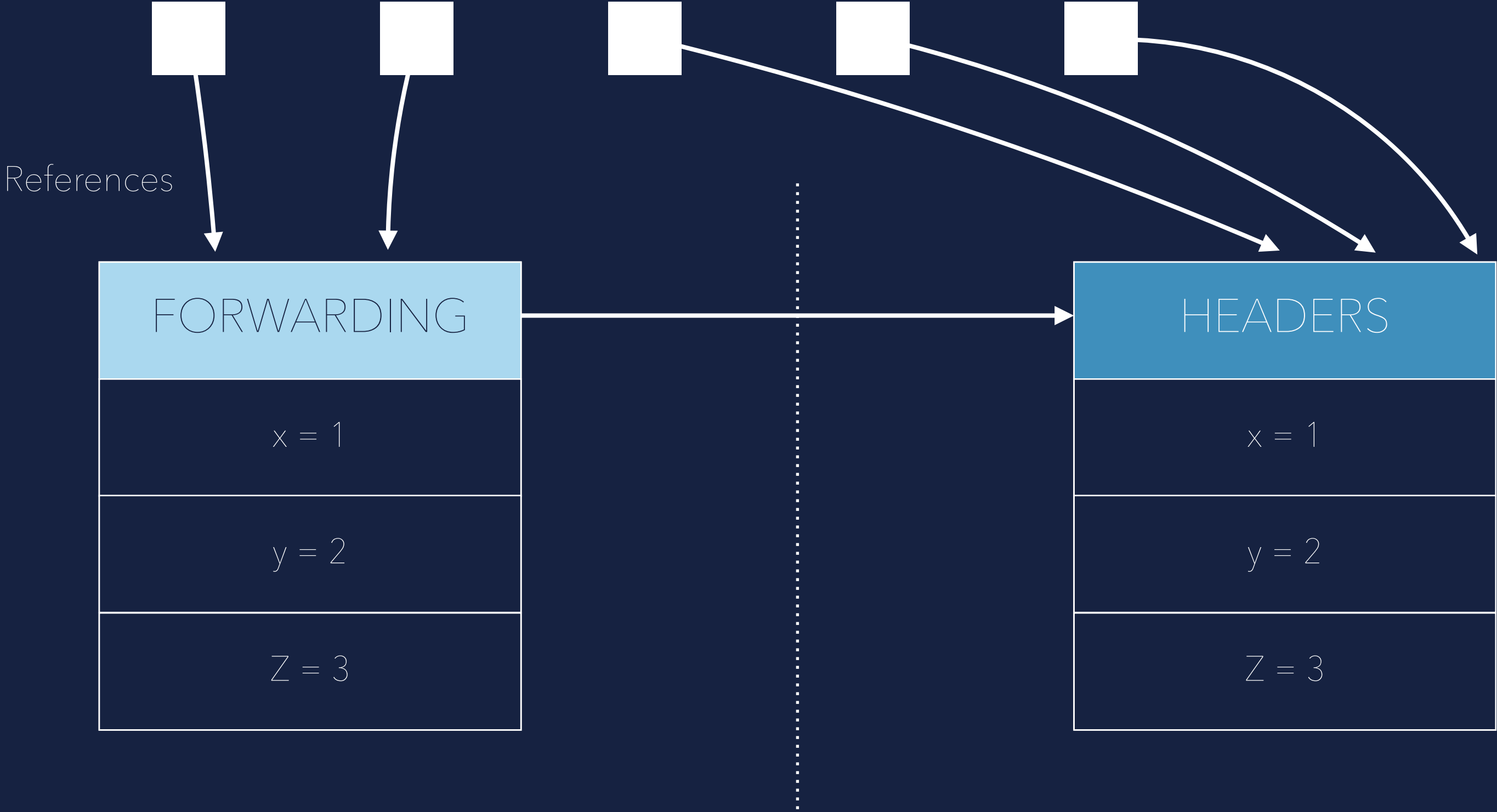
Update all references
(Save the pointer that forwards the copy)

FROM Space

TO Space

CONCURRENCY IS HARD...

Stop the world copying



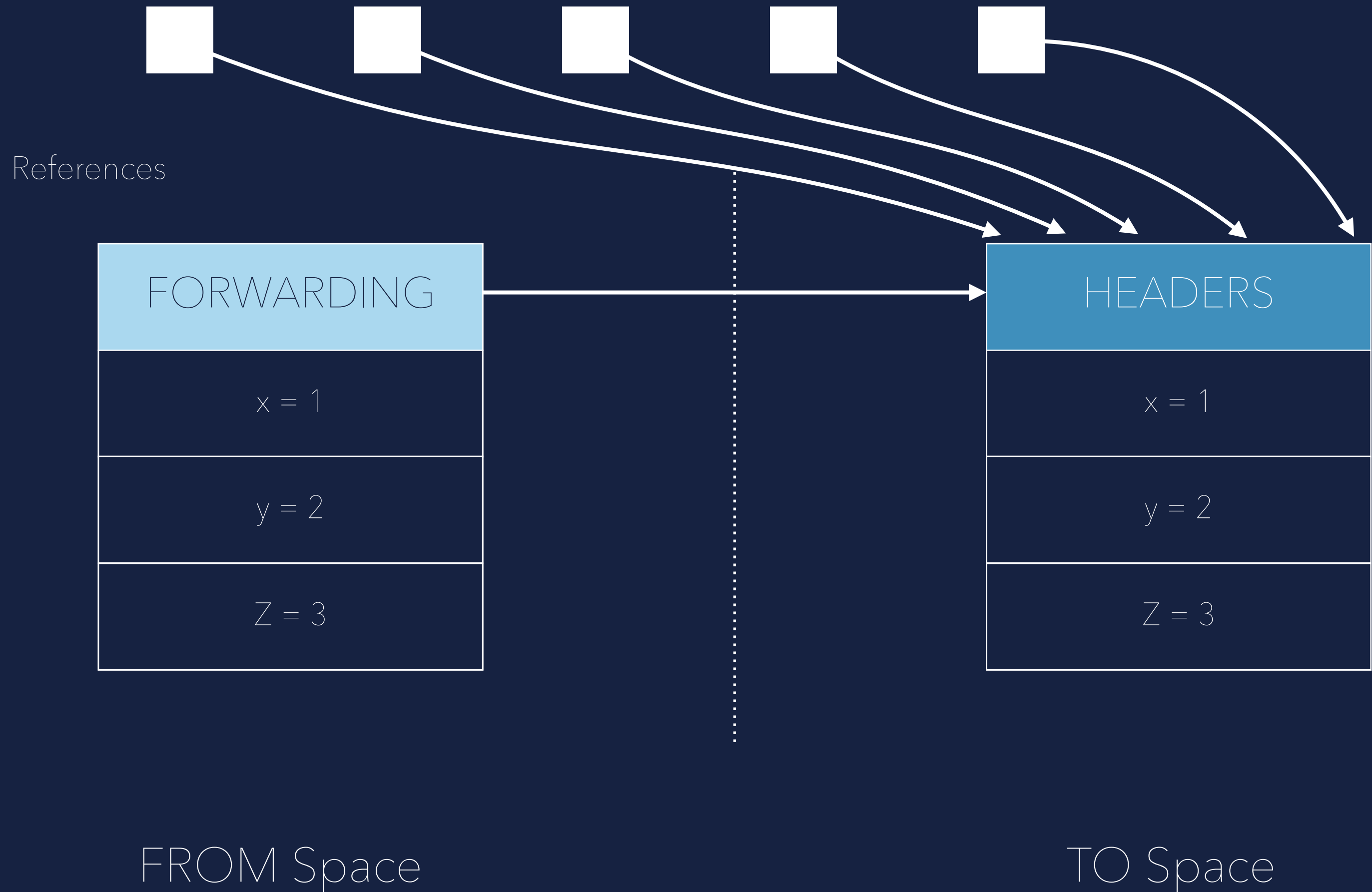
FROM Space

TO Space

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

CONCURRENCY IS HARD...

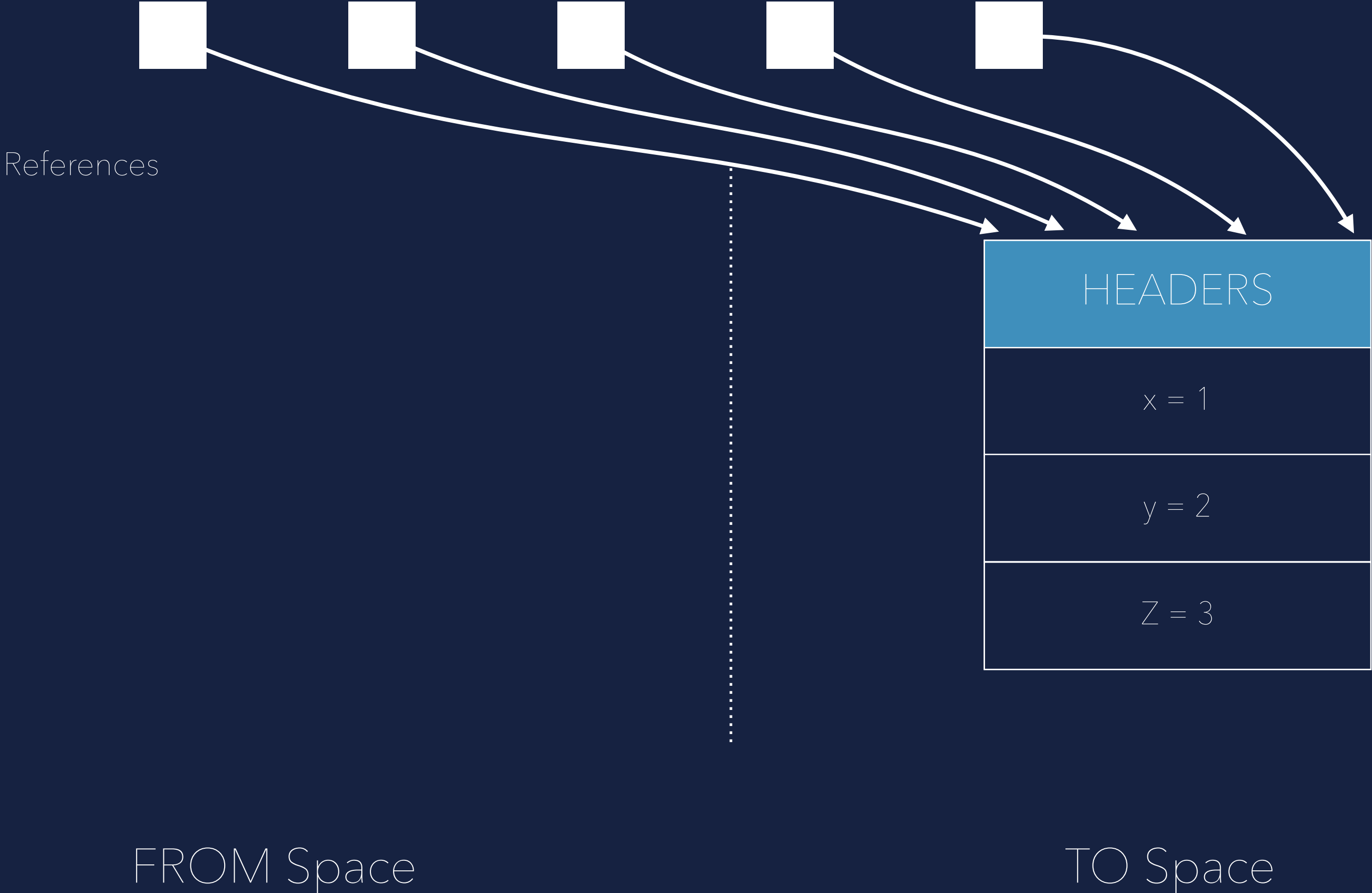
Stop the world copying



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

CONCURRENCY IS HARD...

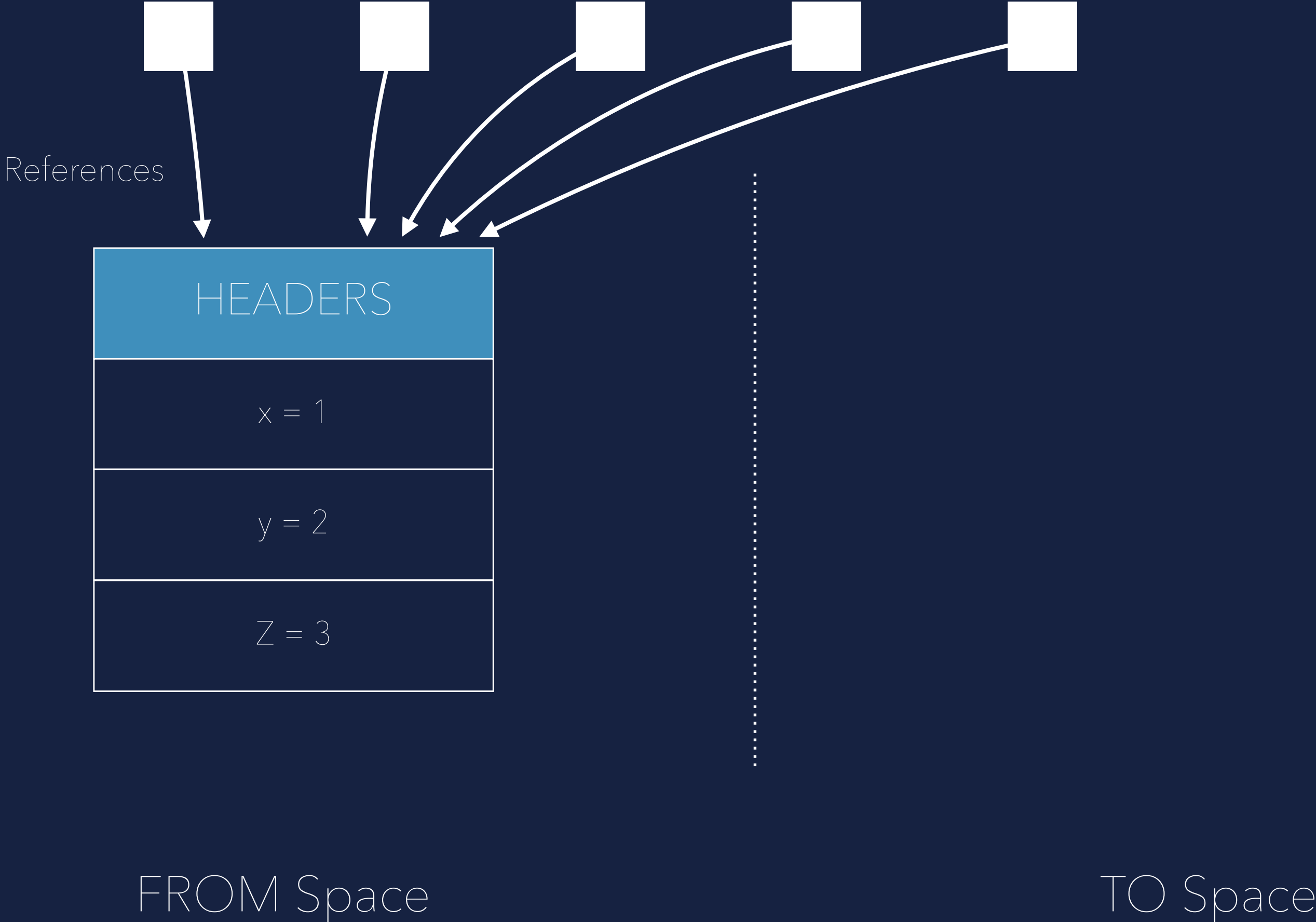
Stop the world copying



Remove old objects and continue running the Mutator

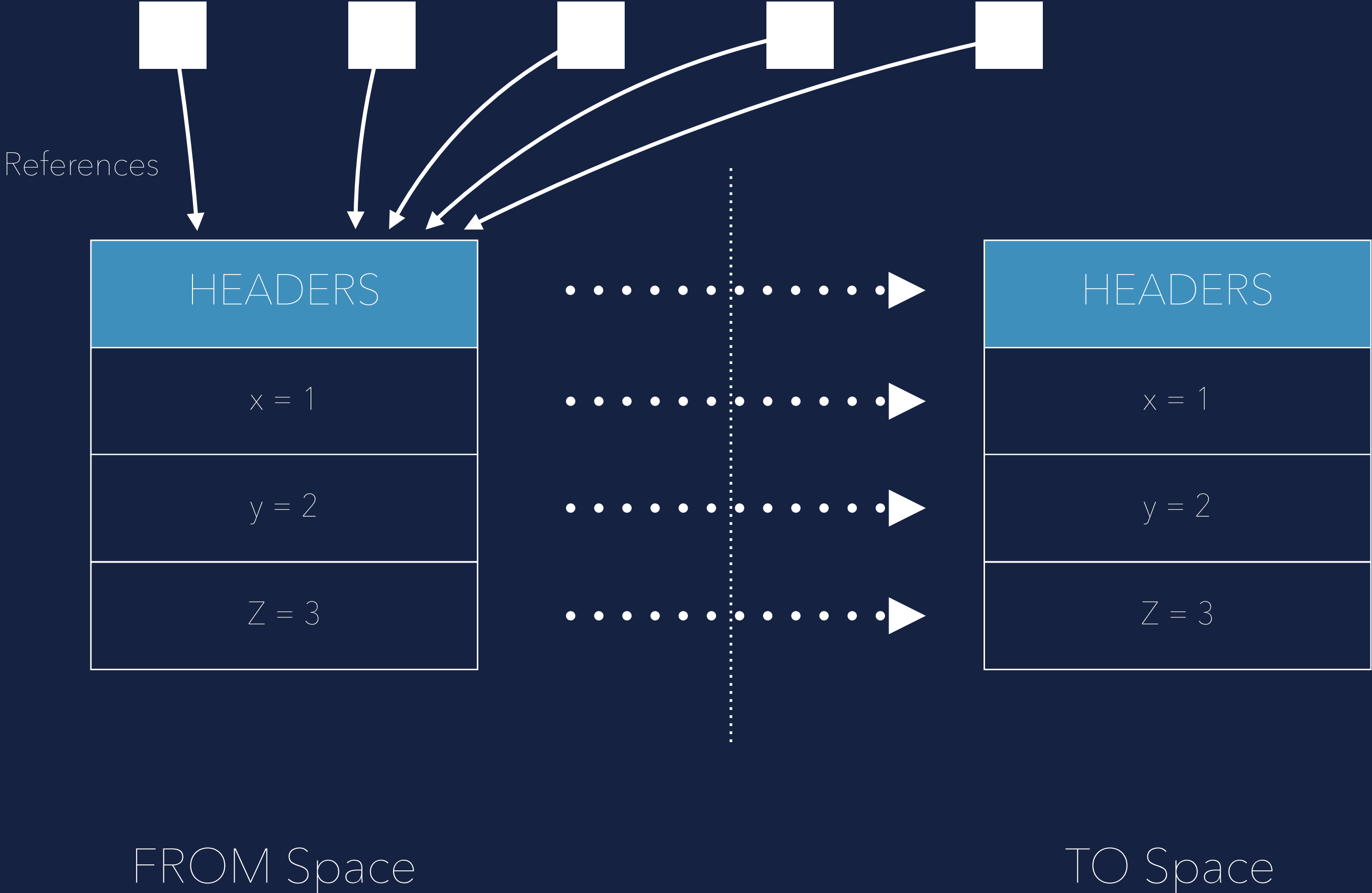
CONCURRENCY IS HARD...

Concurrent copying



CONCURRENCY IS HARD...

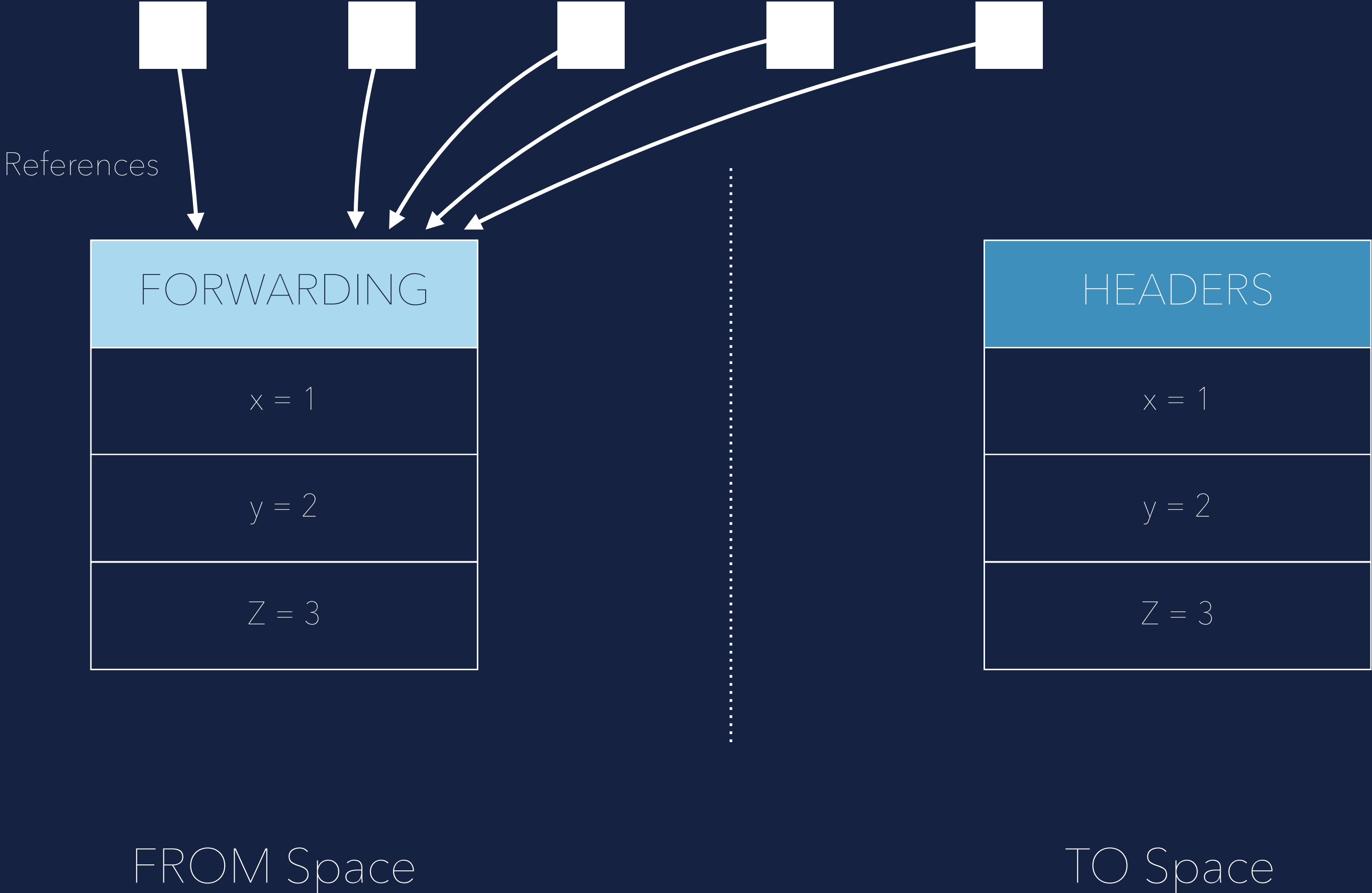
Concurrent copying



While copying
the Object...

CONCURRENCY IS HARD...

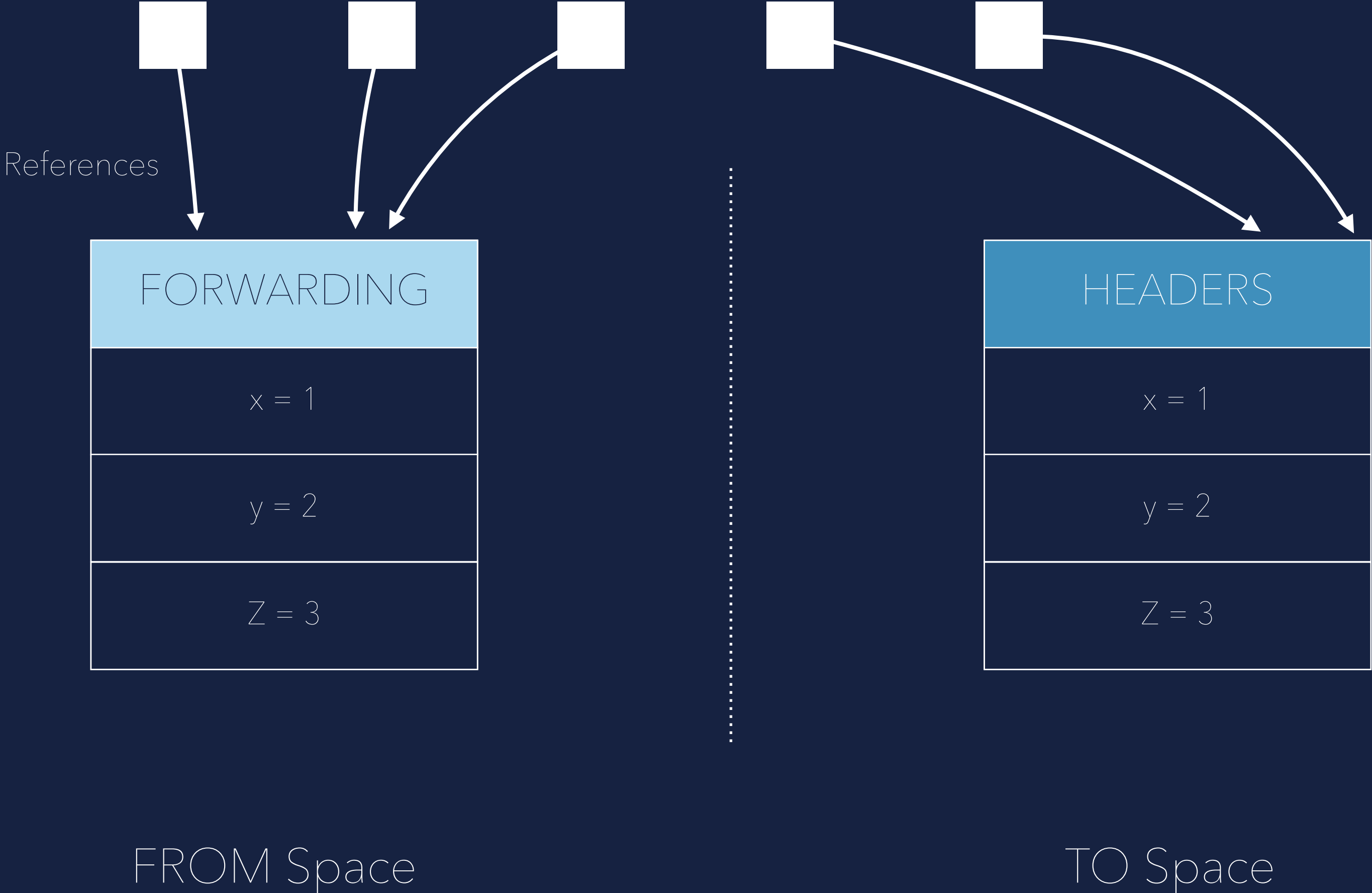
Concurrent copying



While copying
the Object...

CONCURRENCY IS HARD...

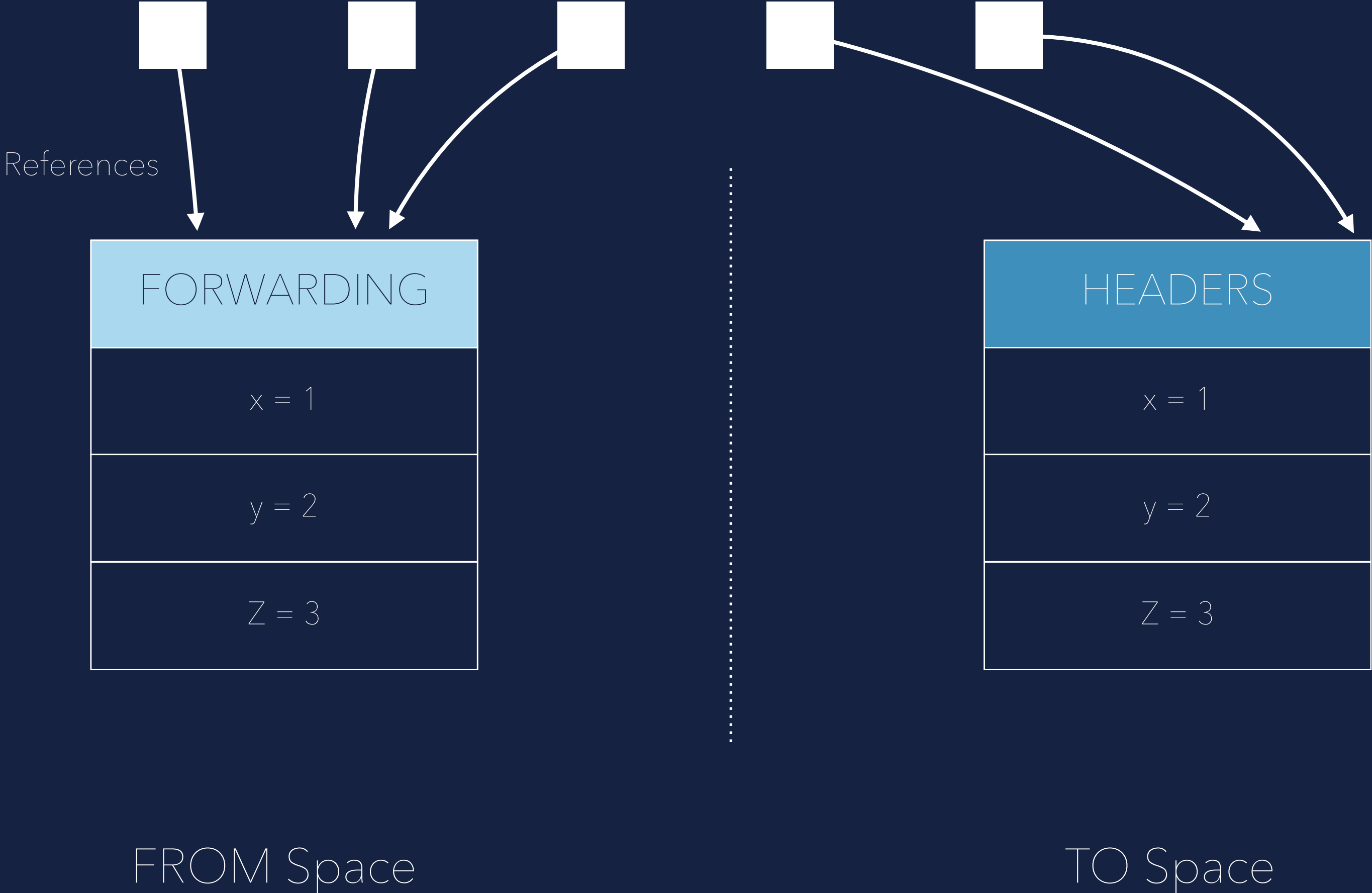
Concurrent copying



...when updating the references...

CONCURRENCY IS HARD...

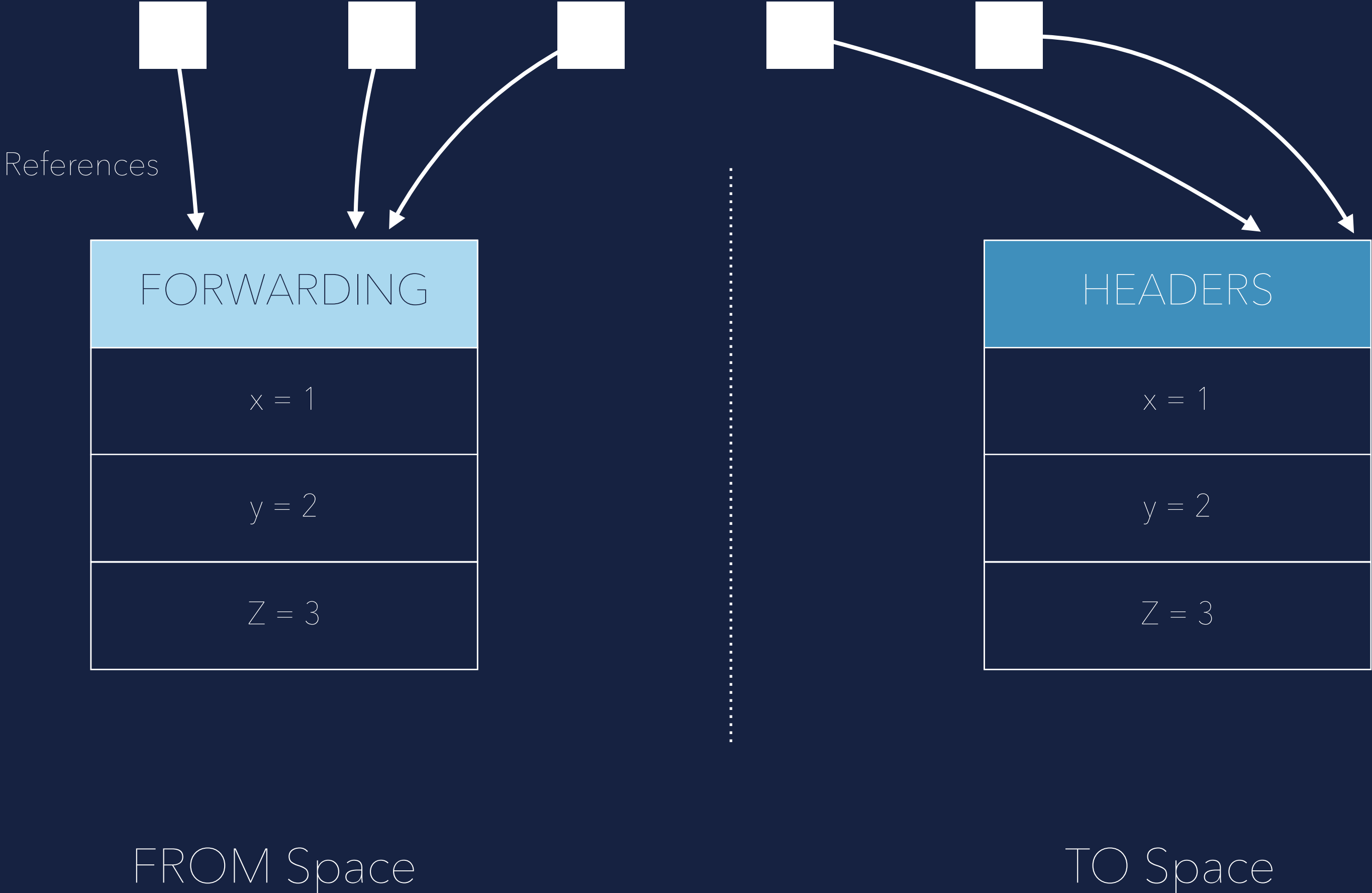
Concurrent copying



...both Objects are reachable !

CONCURRENCY IS HARD...

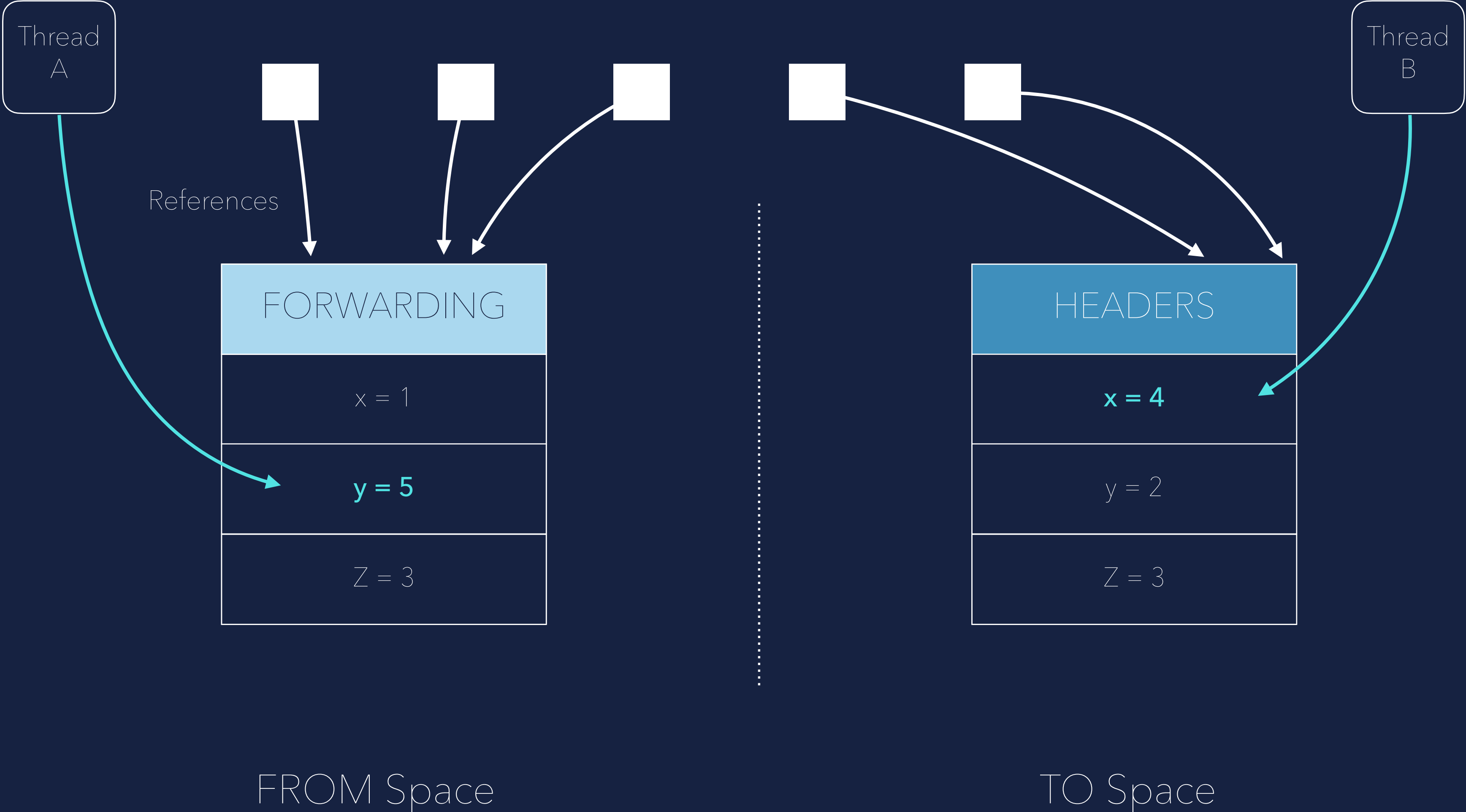
Concurrent copying



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

CONCURRENCY IS HARD...

Concurrent copying

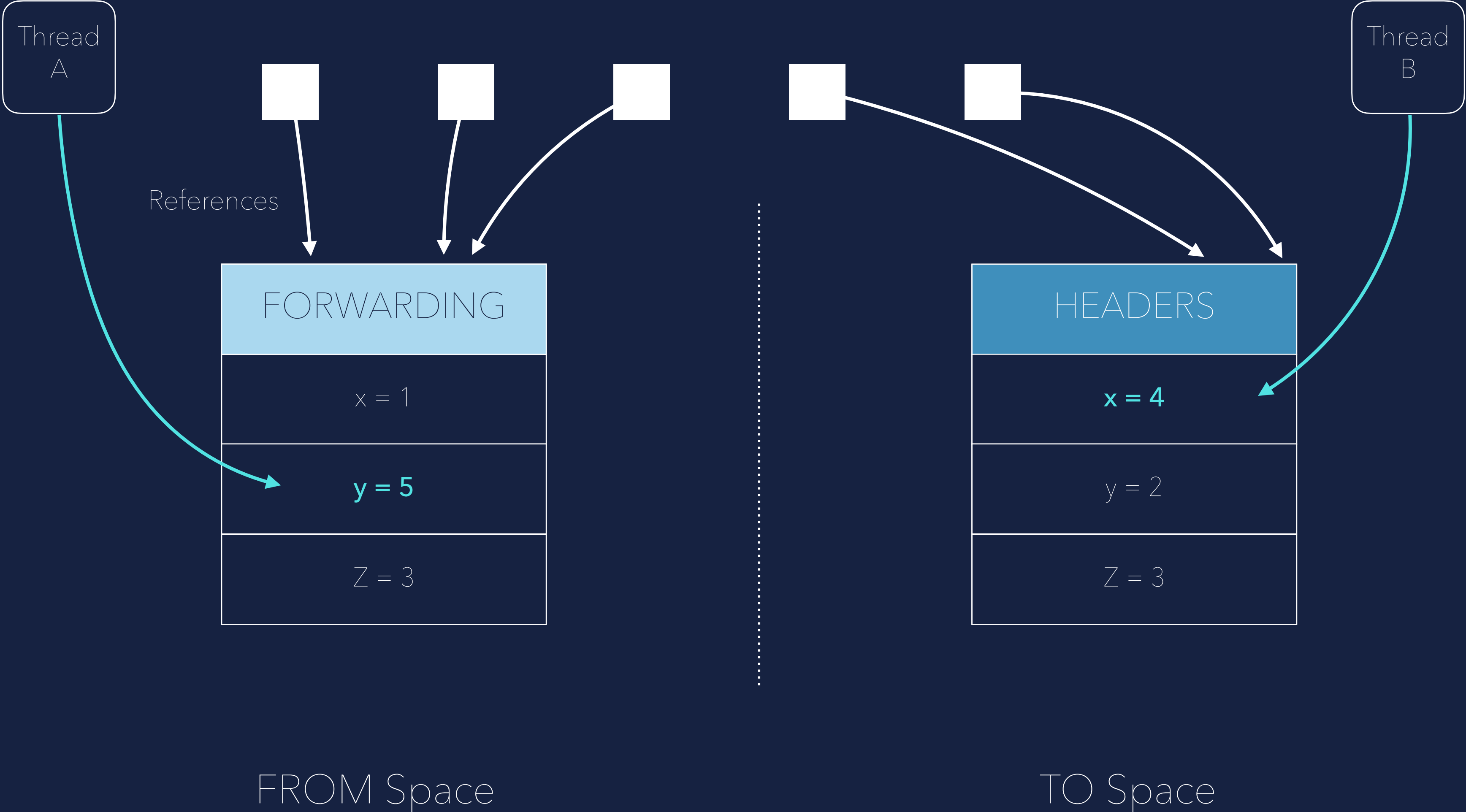


Threads can write to both Objects !



CONCURRENCY IS HARD...

Concurrent copying

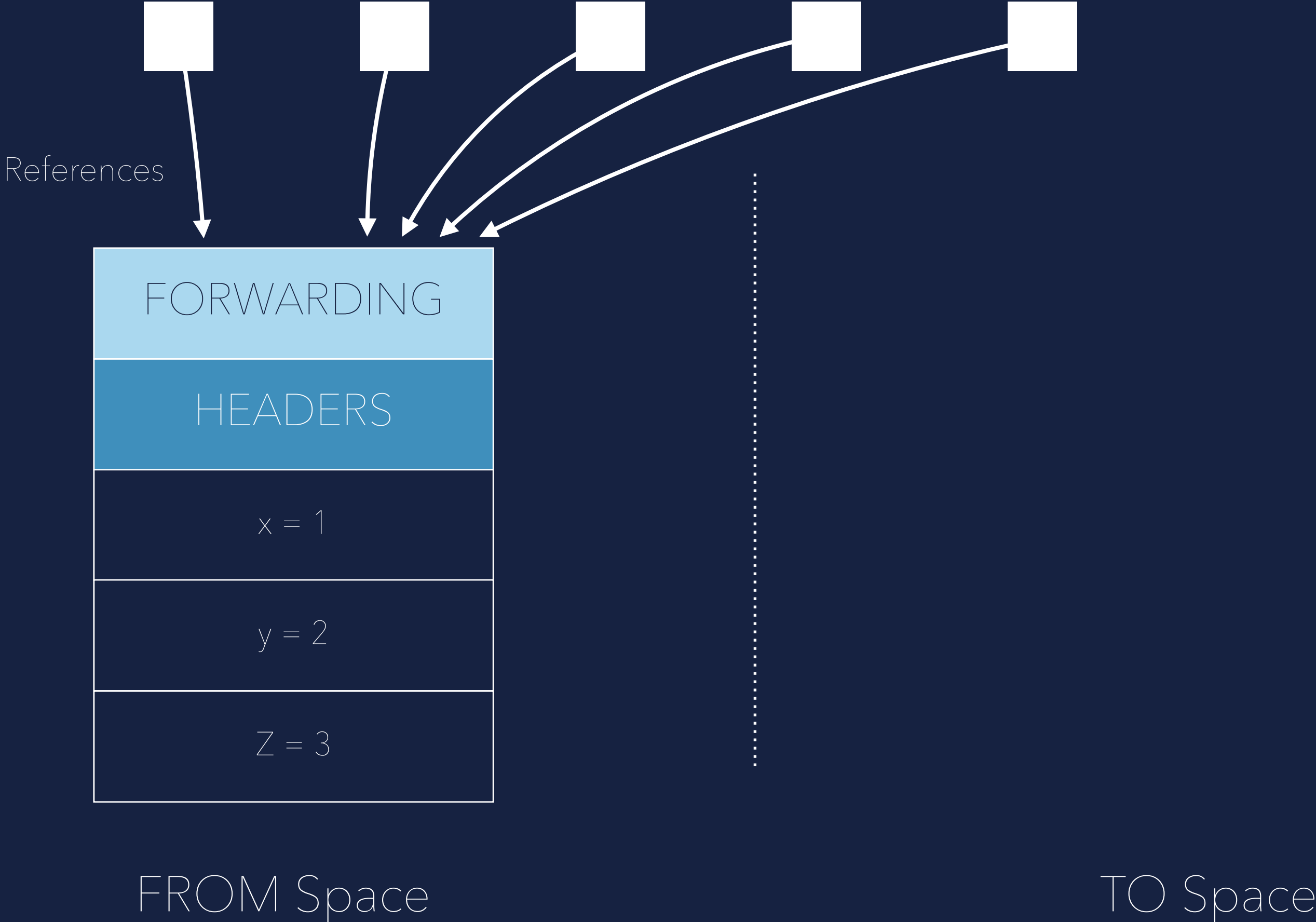


Threads can write to both Objects !
Which copy is correct ?



CONCURRENCY IS HARD...

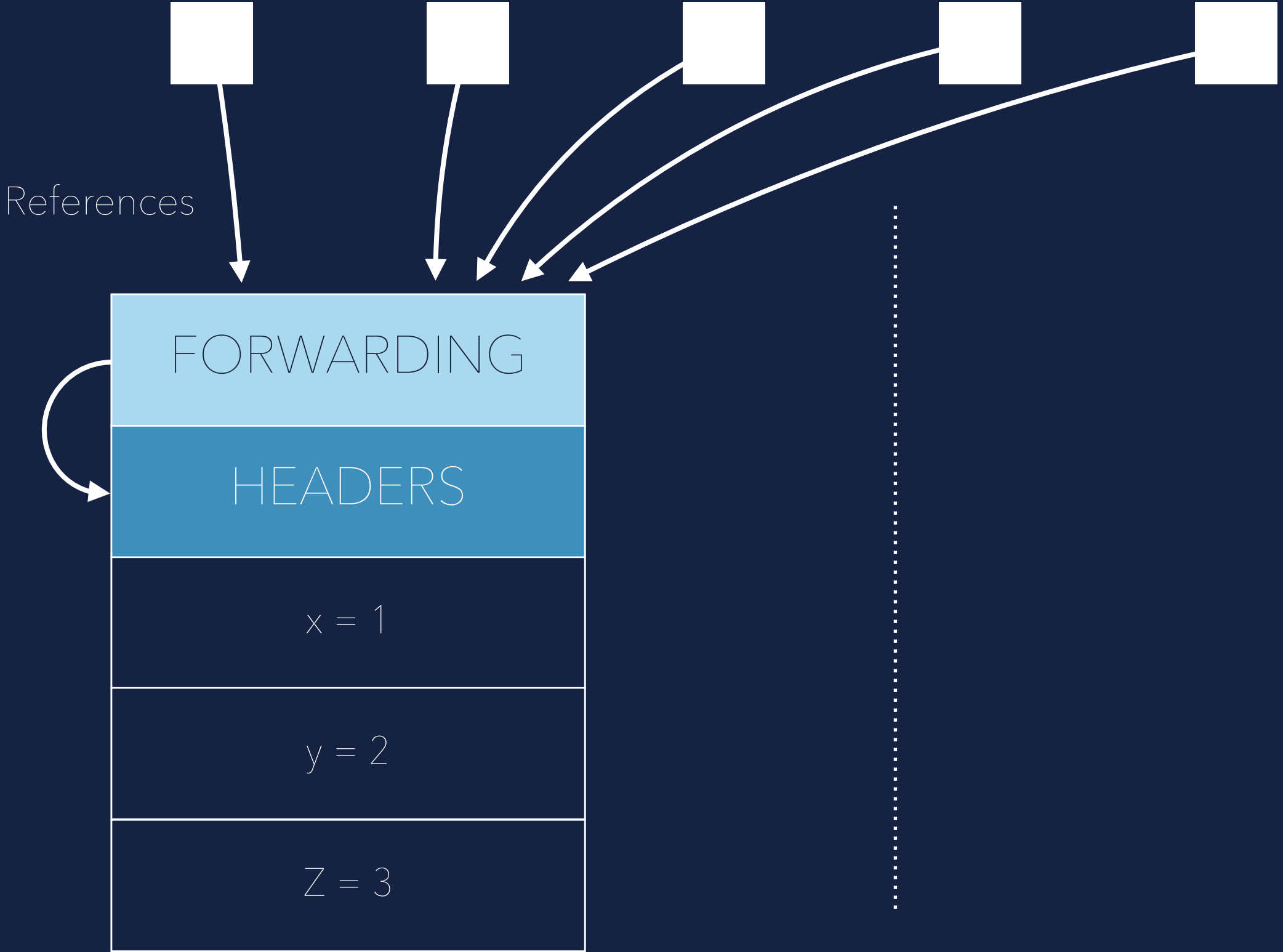
Concurrent copying



Solution could be installing a Brooks Pointer...

CONCURRENCY IS HARD...

Concurrent copying



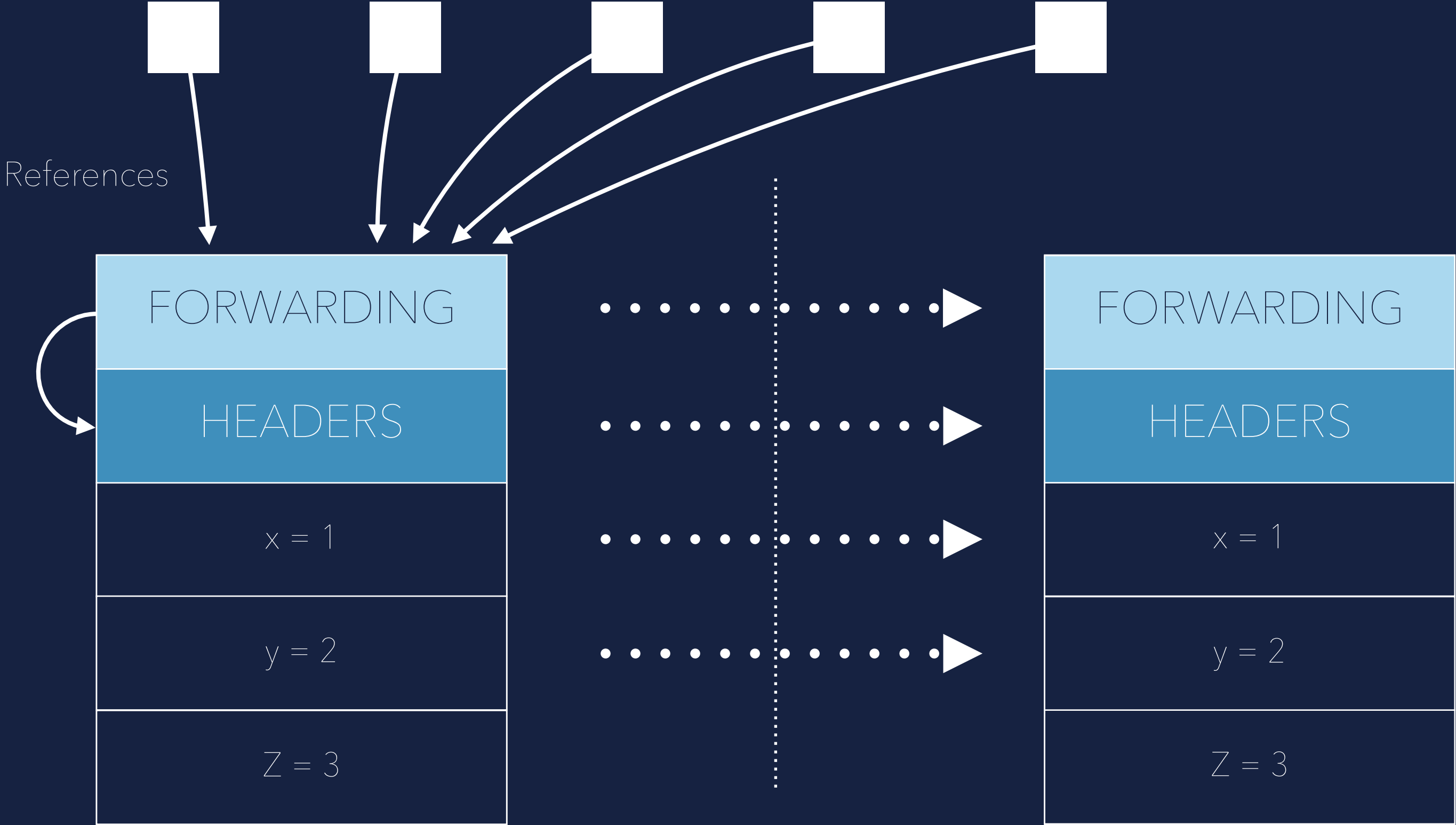
...which points to object header itself

FROM Space

TO Space

CONCURRENCY IS HARD...

Concurrent copying



Atomic

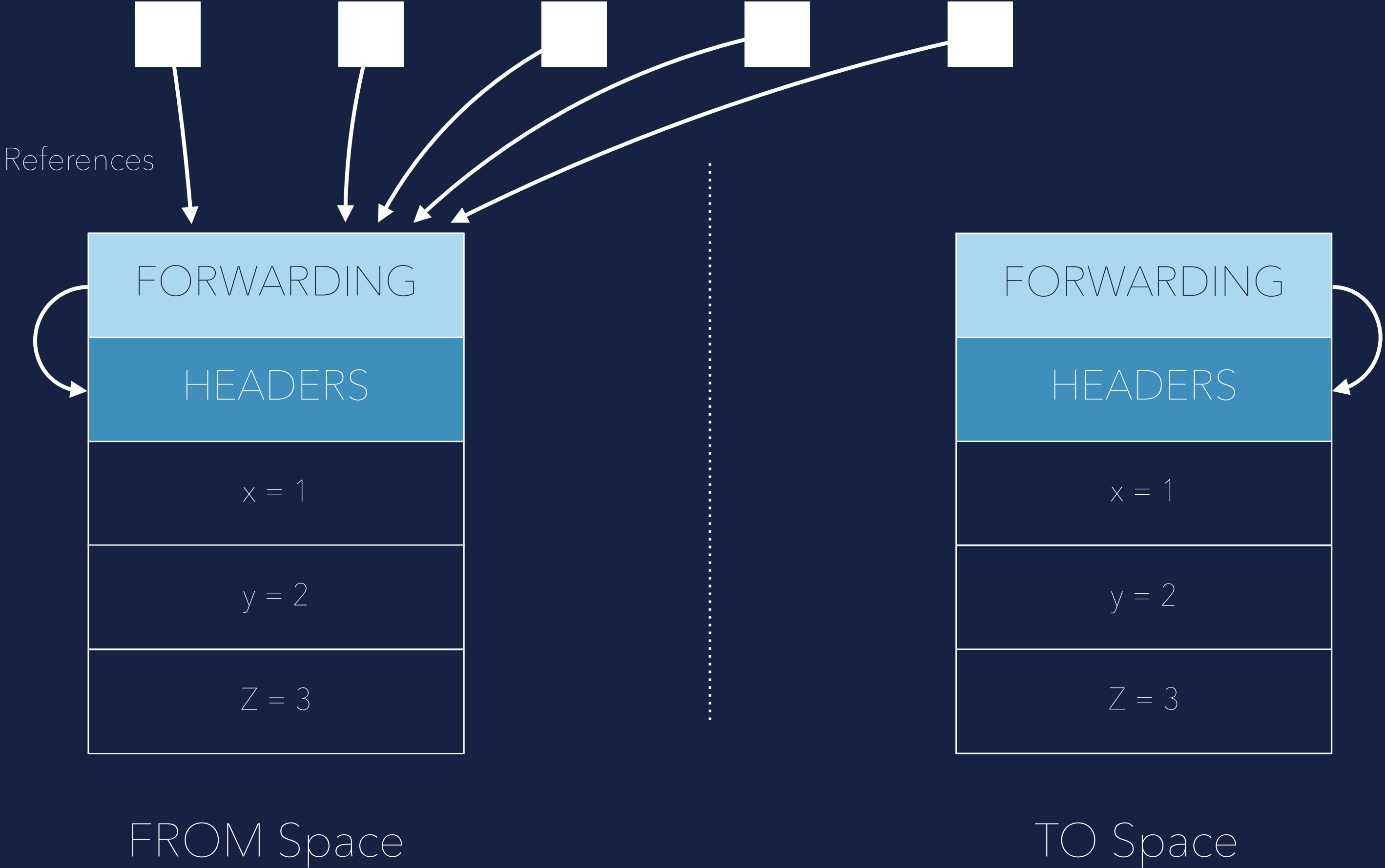
FROM Space

TO Space

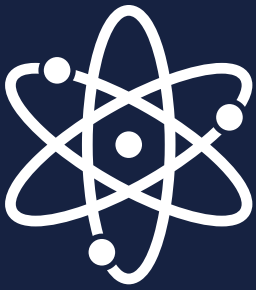
Copy the Object

CONCURRENCY IS HARD...

Concurrent copying



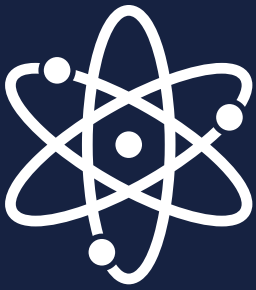
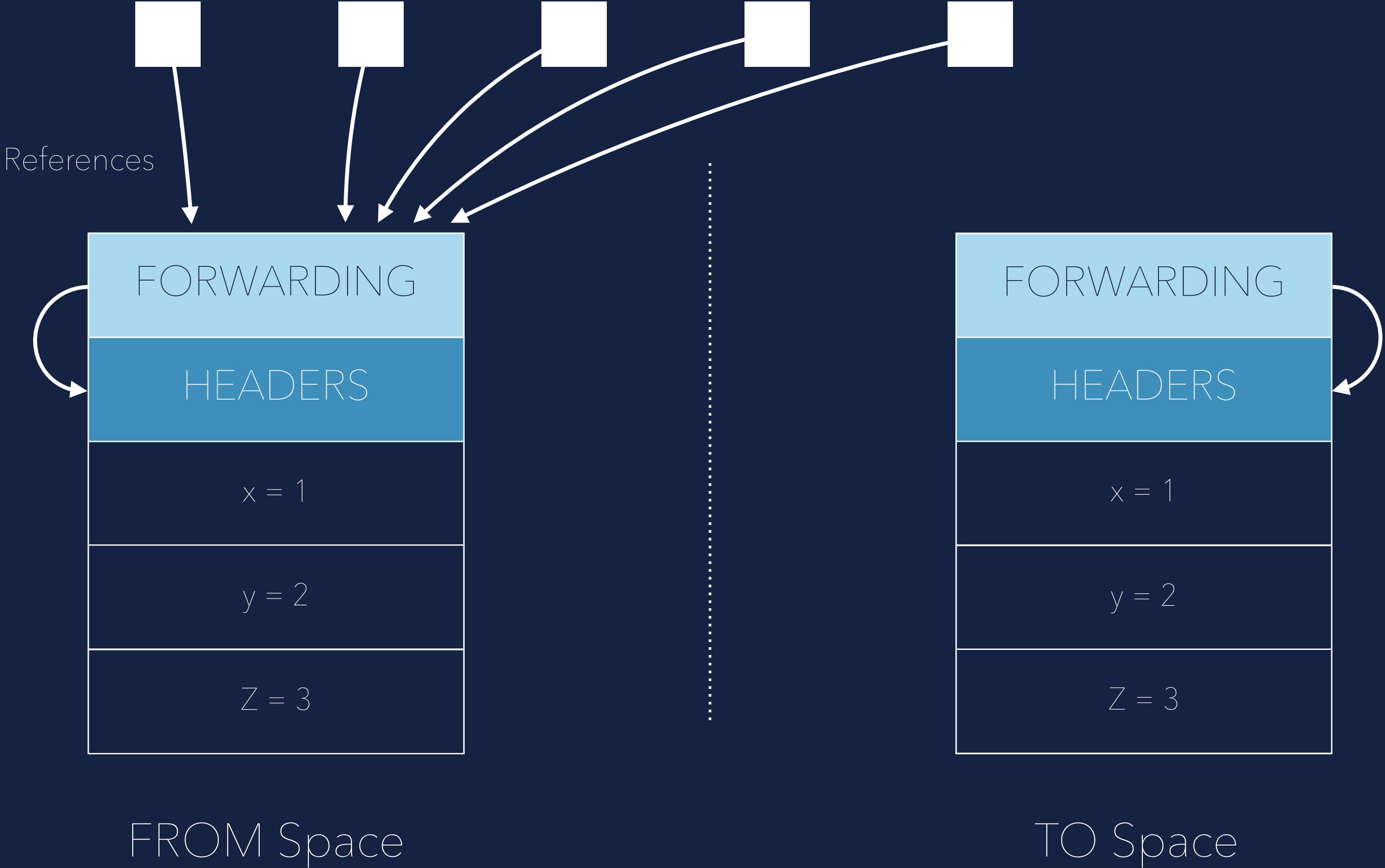
Install forwarding pointer to itself



Atomic

CONCURRENCY IS HARD...

Concurrent copying

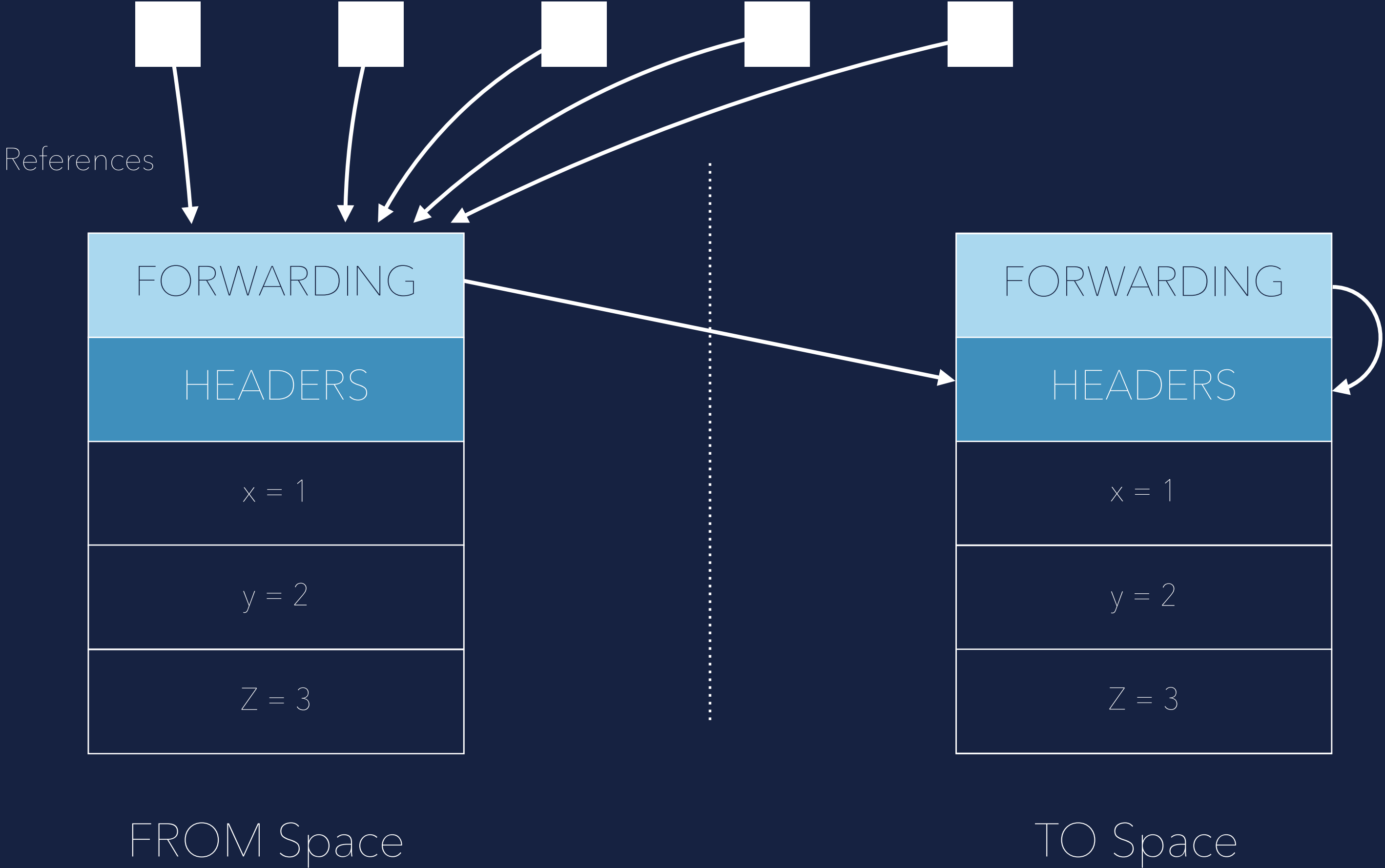


Atomic

Nobody knows about copy

CONCURRENCY IS HARD...

Concurrent copying



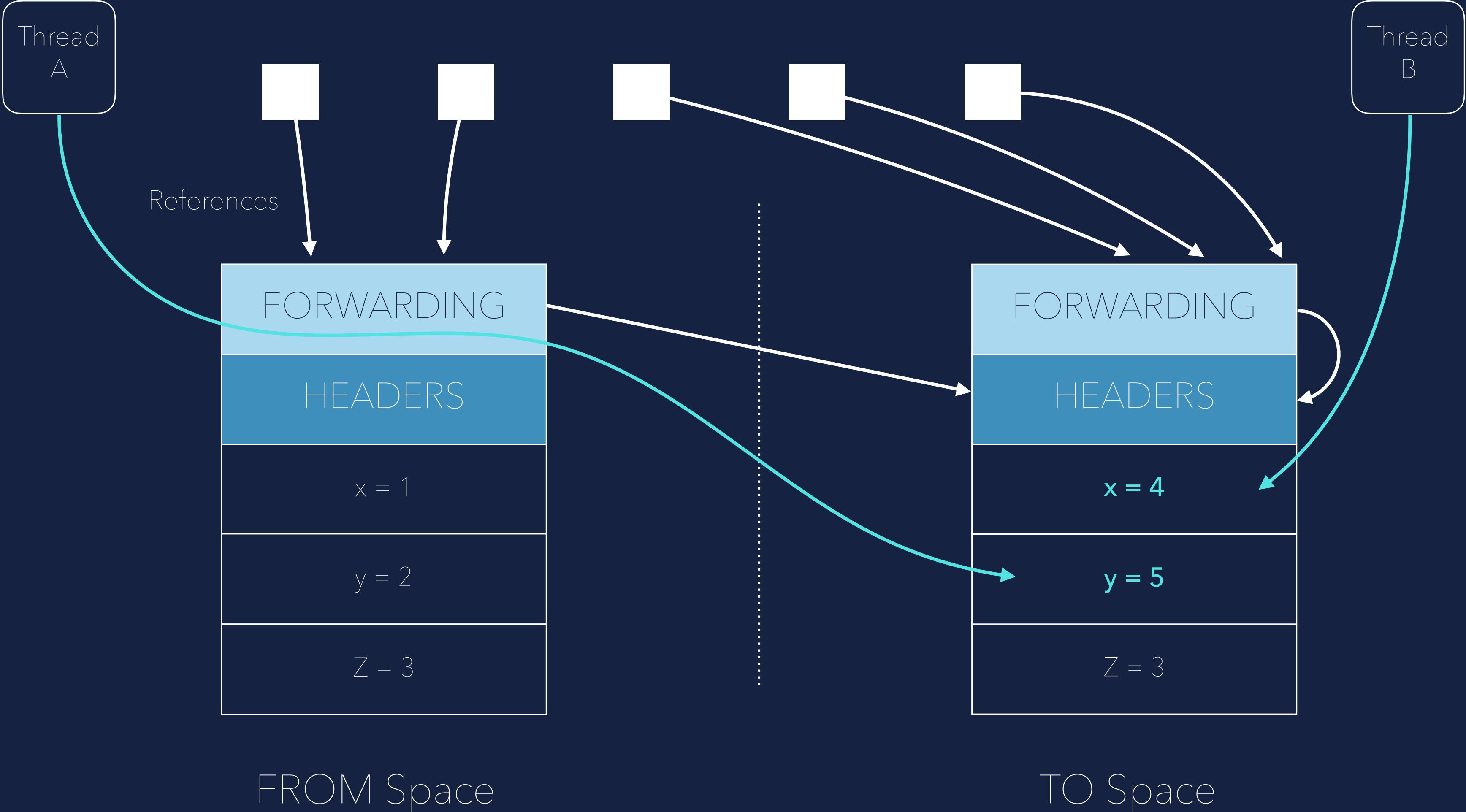
Atomically update forwarding pointer of original object to new copy



Atomic

CONCURRENCY IS HARD...

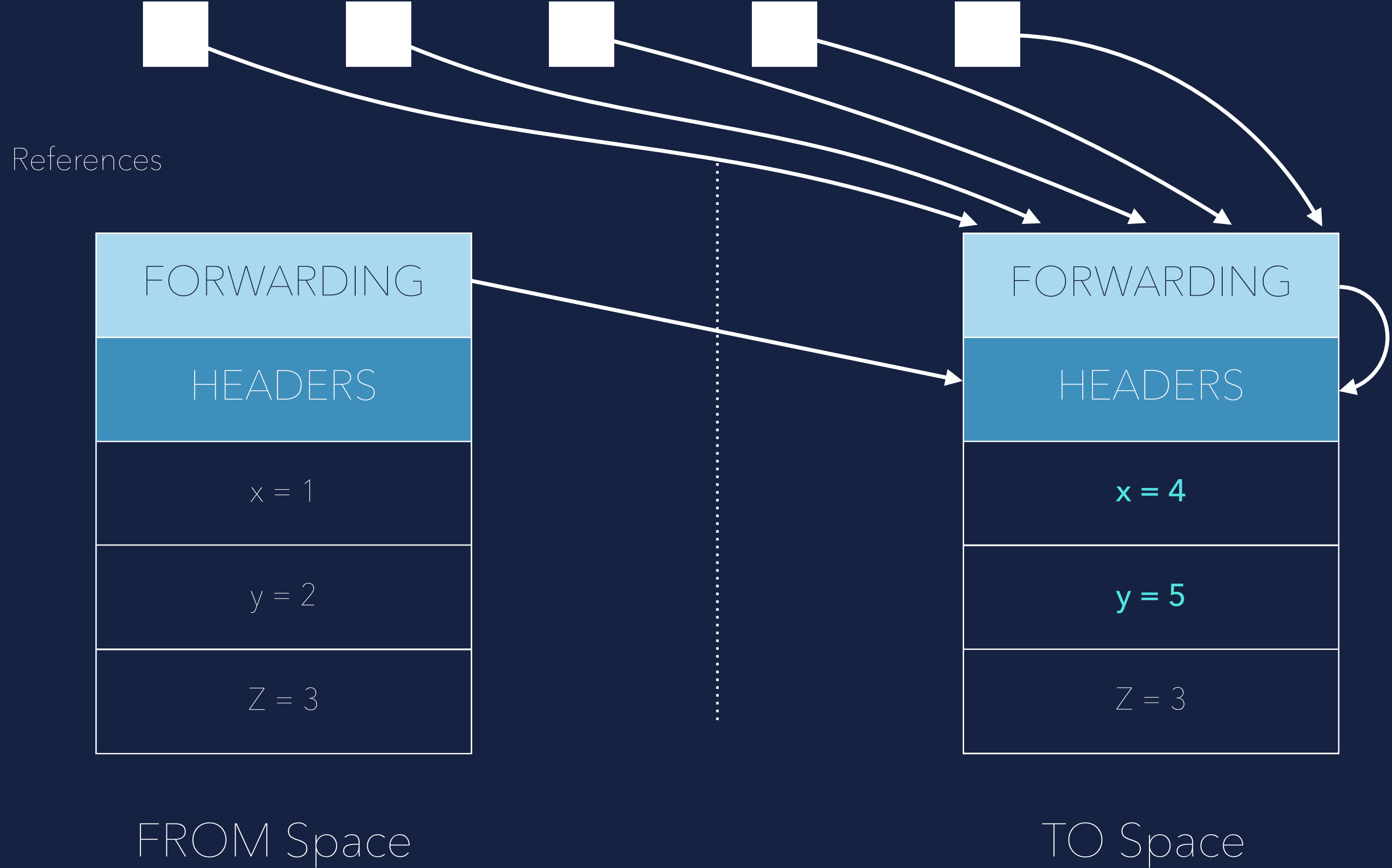
Concurrent copying



Threads now will always find the right object

CONCURRENCY IS HARD...

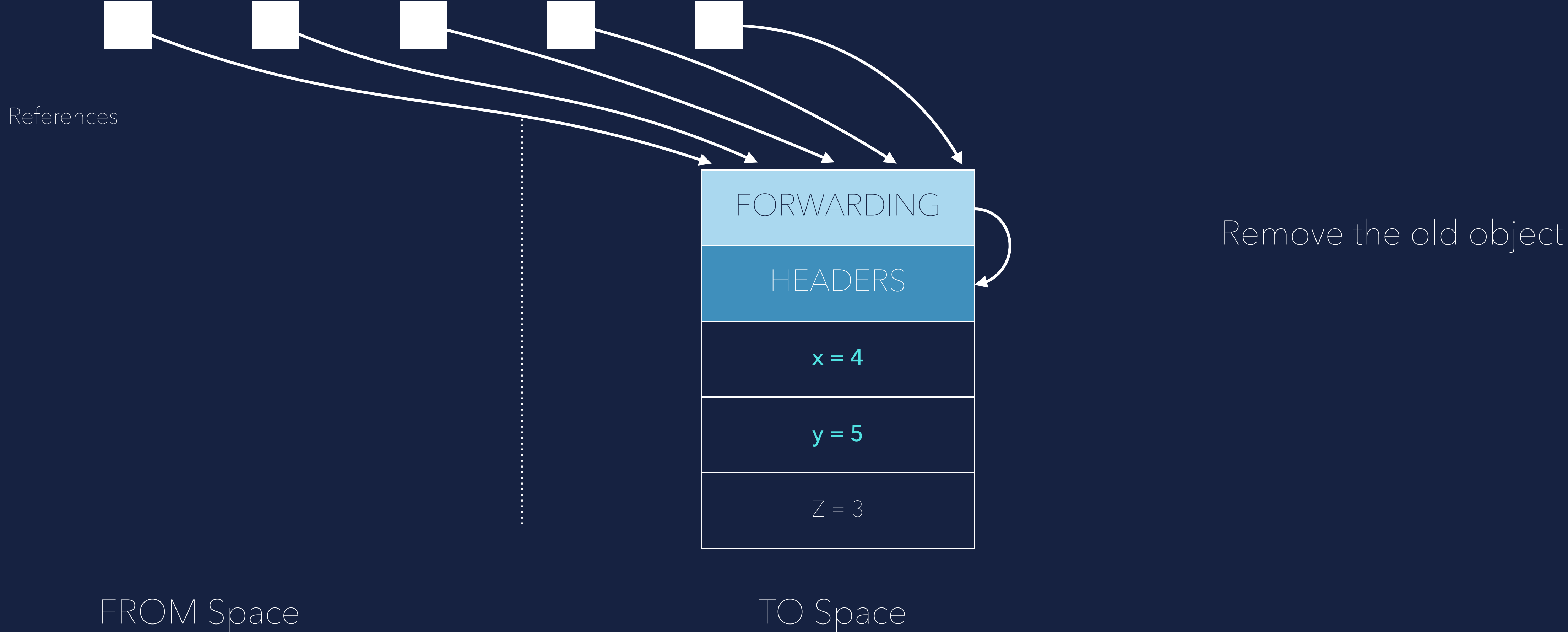
Concurrent copying



When all references are updated...

CONCURRENCY IS HARD...

Concurrent copying



COLLECTORS IN THE JVM



SEERIAL



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`

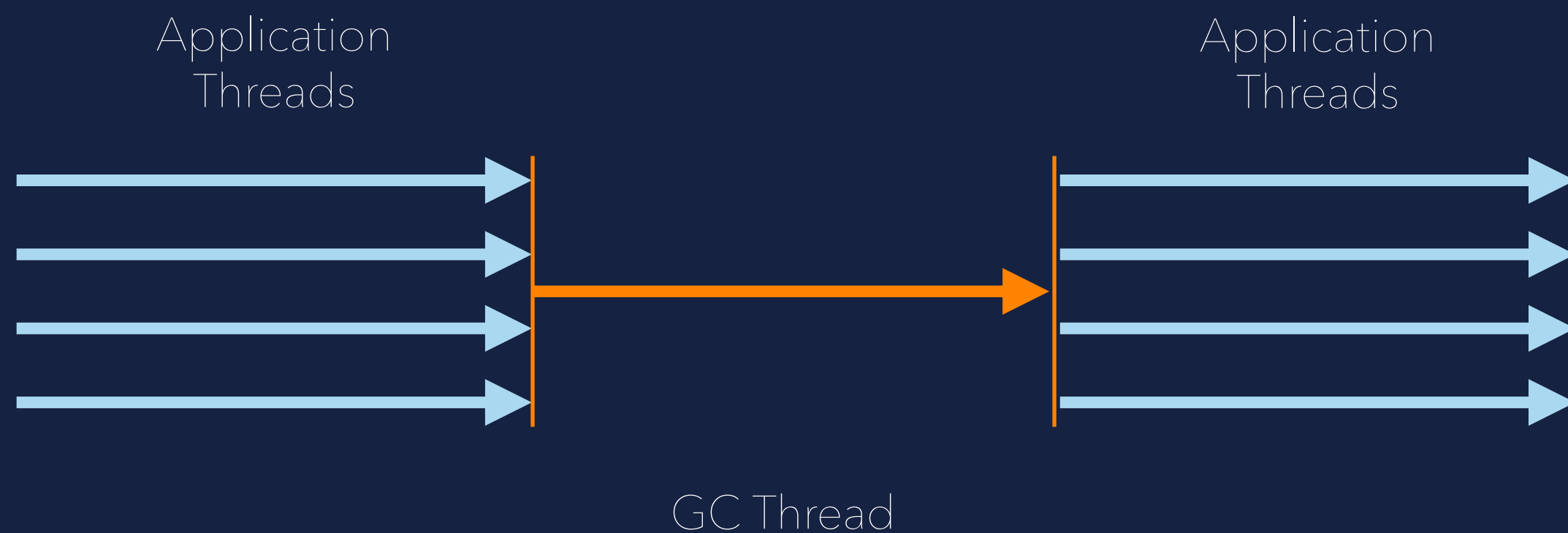


SERIAL

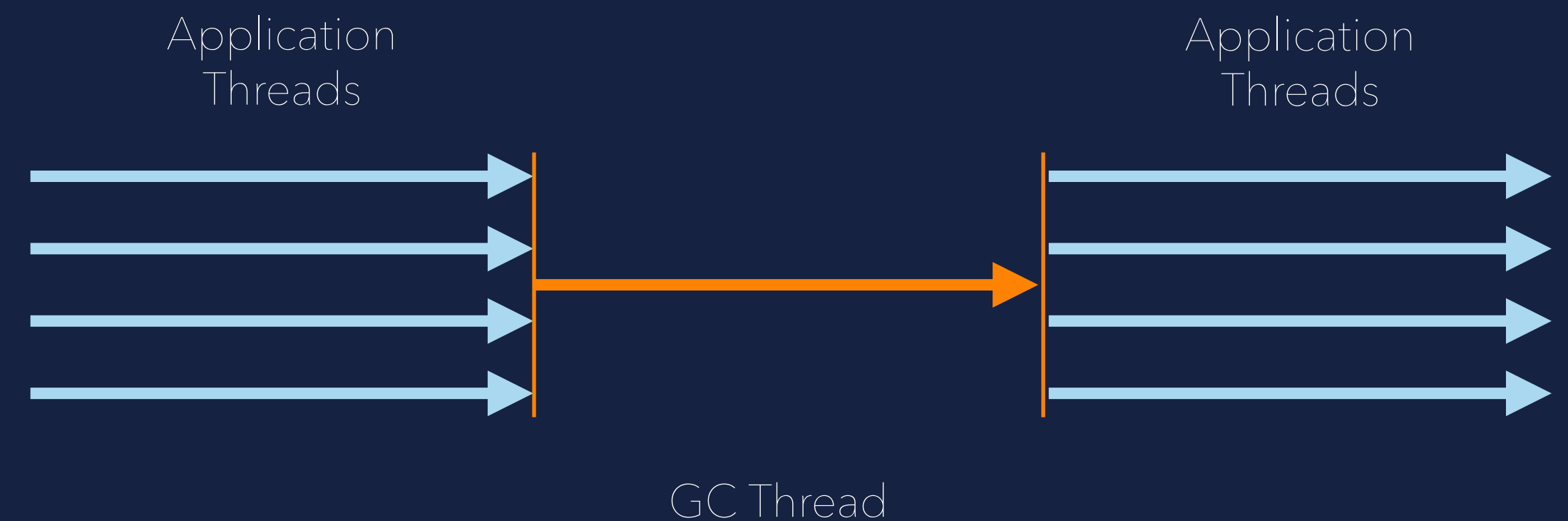
NOTES

- 🗑️ Automatically selected if only a single processor is available
- 🗑️ Automatically selected if the avail. memory less than 1792 MB
- 🗑️ Mark and Compact

Young Generation



Old Generation





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:+UseParallelGC/-XX:+UseParallelOldGC`



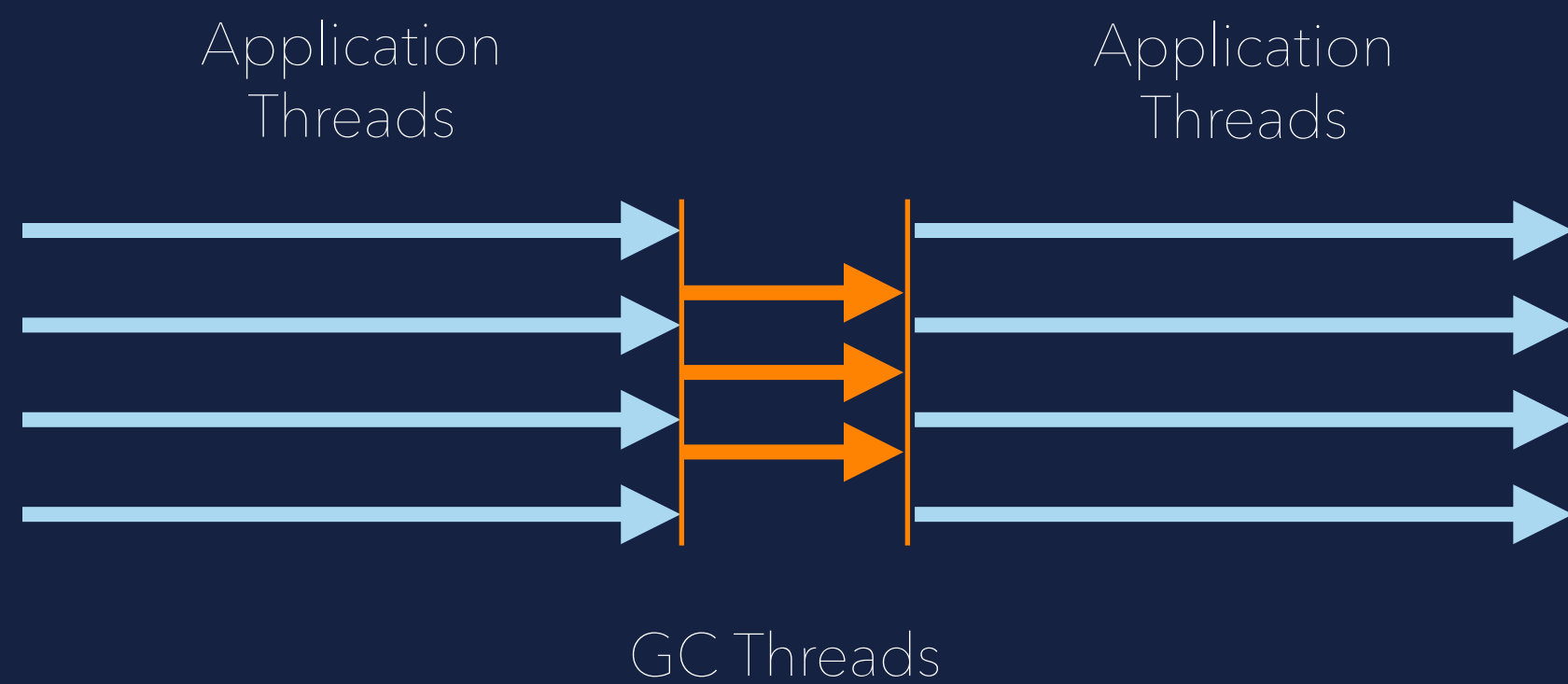
PARALLEL

NOTES

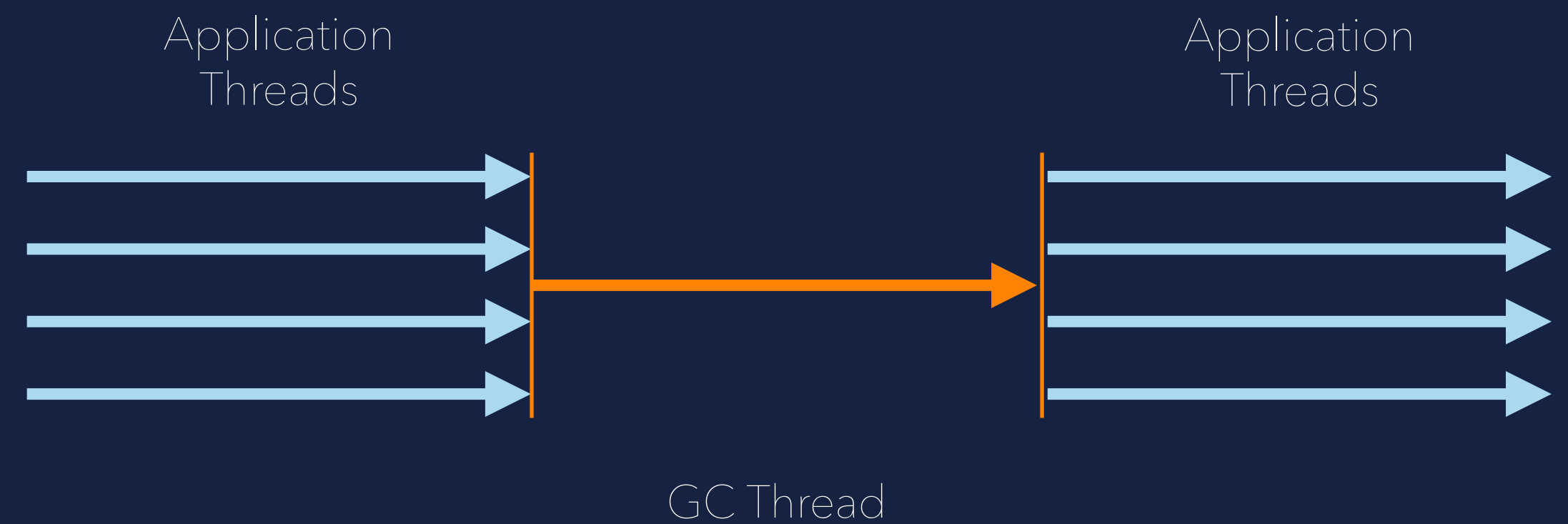
- 🗑️ Default garbage collector from JDK 5 to JDK 8
- 🗑️ Mark and Compact

`-XX:+UseParallelGC`

Young Generation



Old Generation





PARALLEL

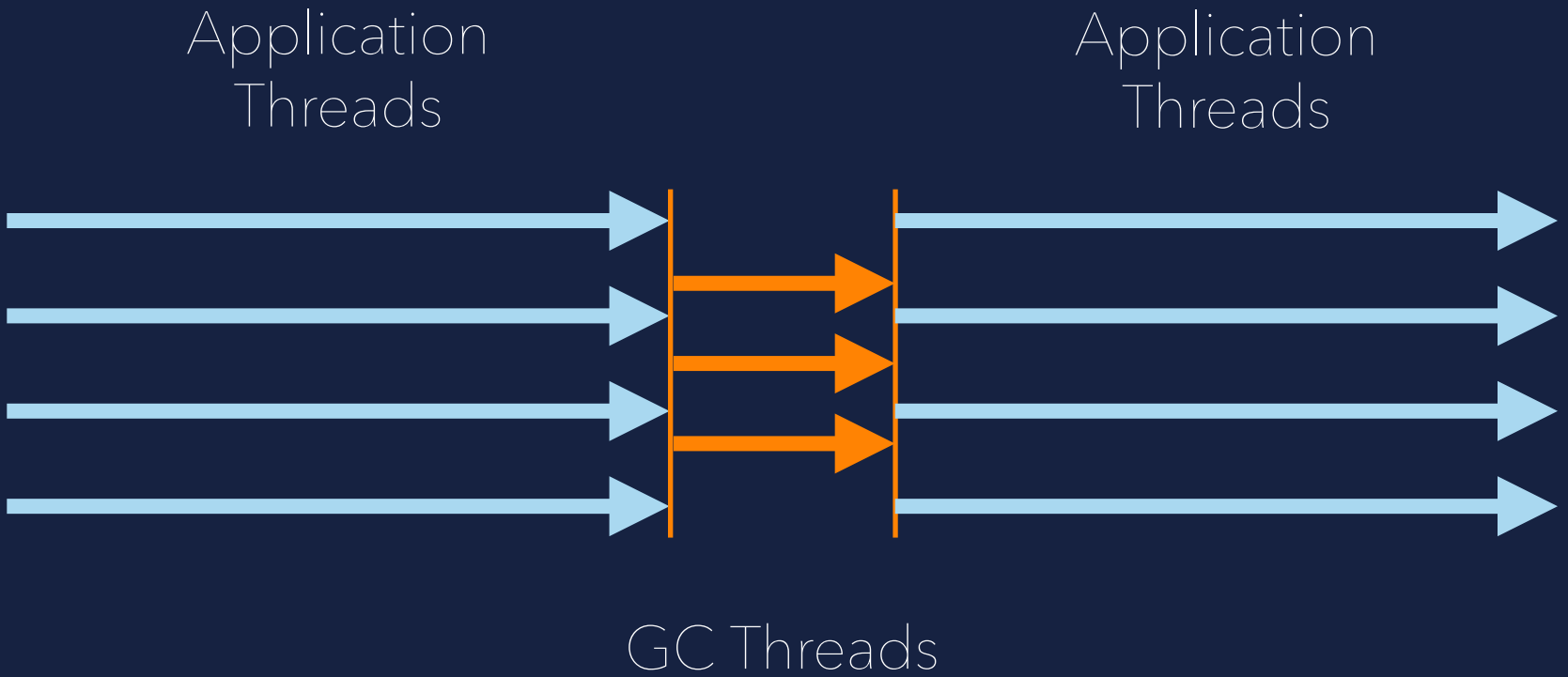
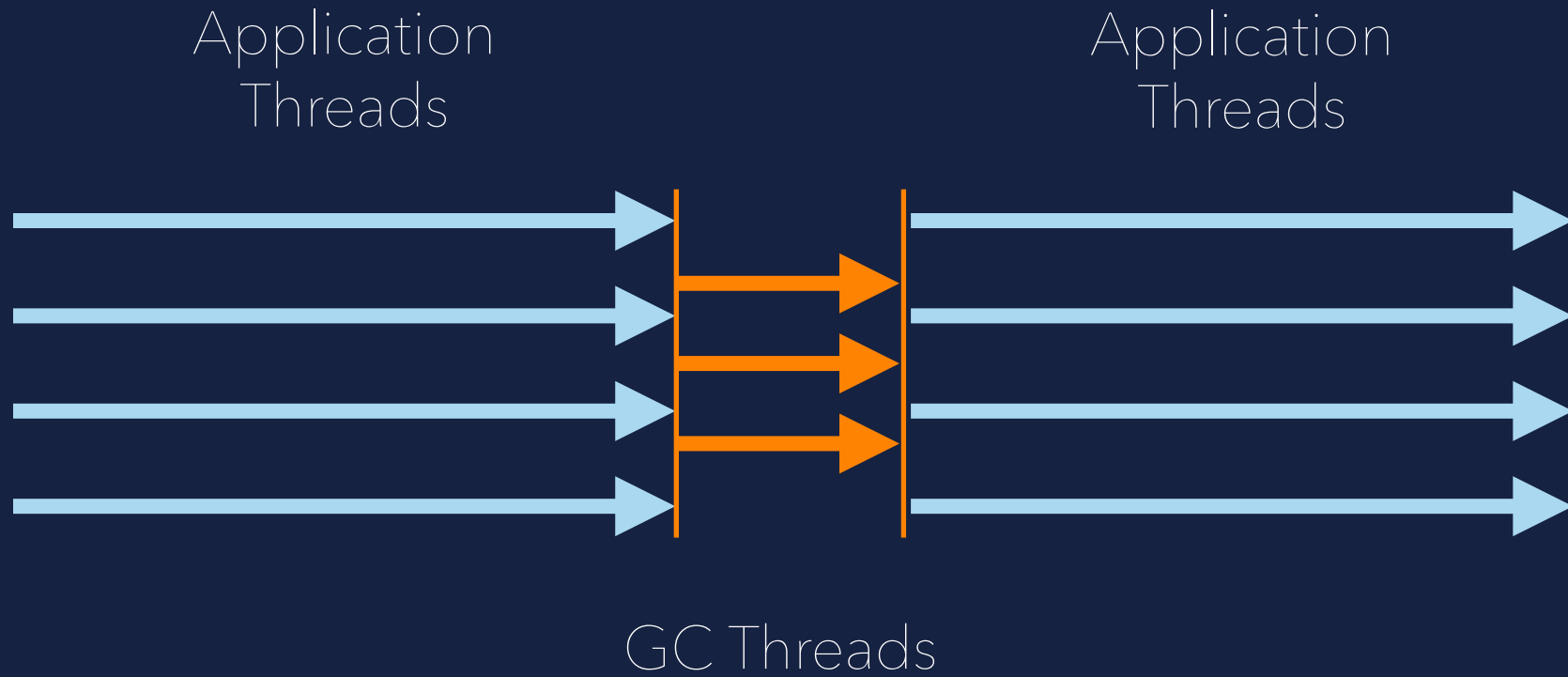
NOTES

- 🗑️ Default garbage collector from JDK 5 to JDK 8
- 🗑️ Mark and Compact

```
-XX:+UseParallelOldGC
```

Young Generation

Old Generation





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%)

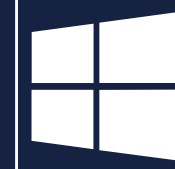
CHOOSE WHEN

-  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

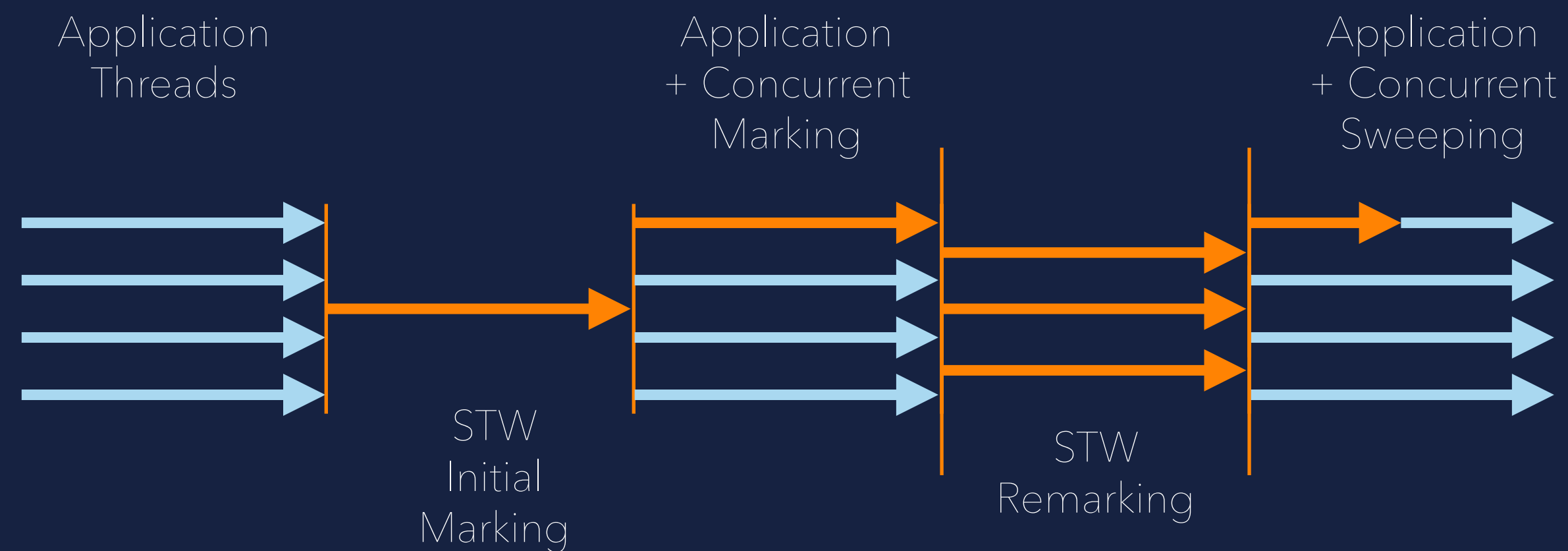


JVM SWITCH

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

NOTES

- 🗑️ Deprecated as of JDK 9
- 🗑️ Removed from JDK 14
- 🗑️ Concurrent marking but no compaction -> Fragmentation





G1

Garbage First



Heap-Layout

Region size 1 - 32 MB

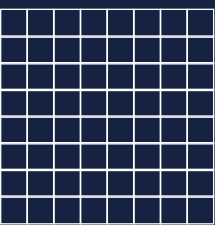
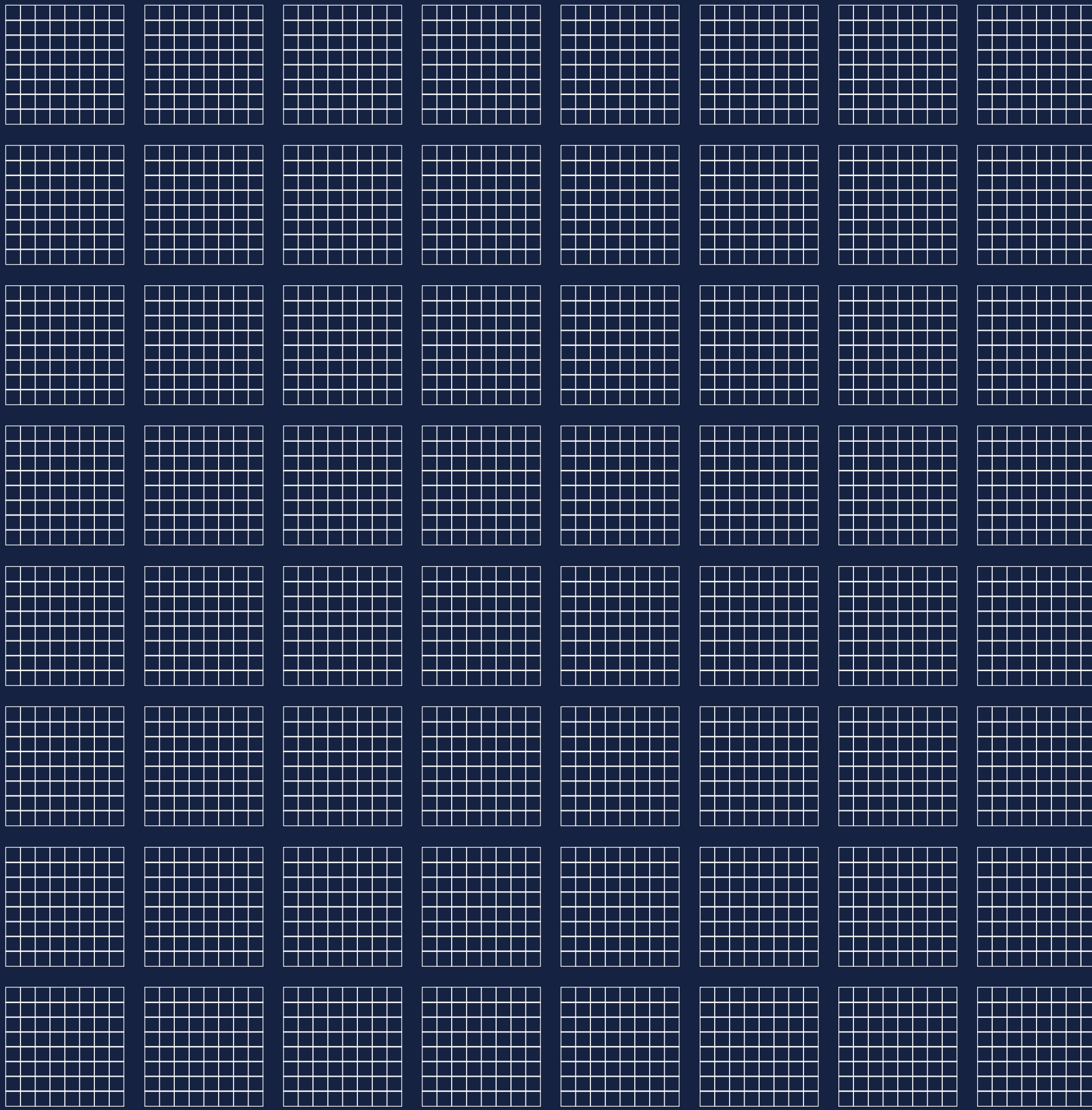
Max no. of region <= 2048

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

Heap-Layout

Region size 1 - 32 MB

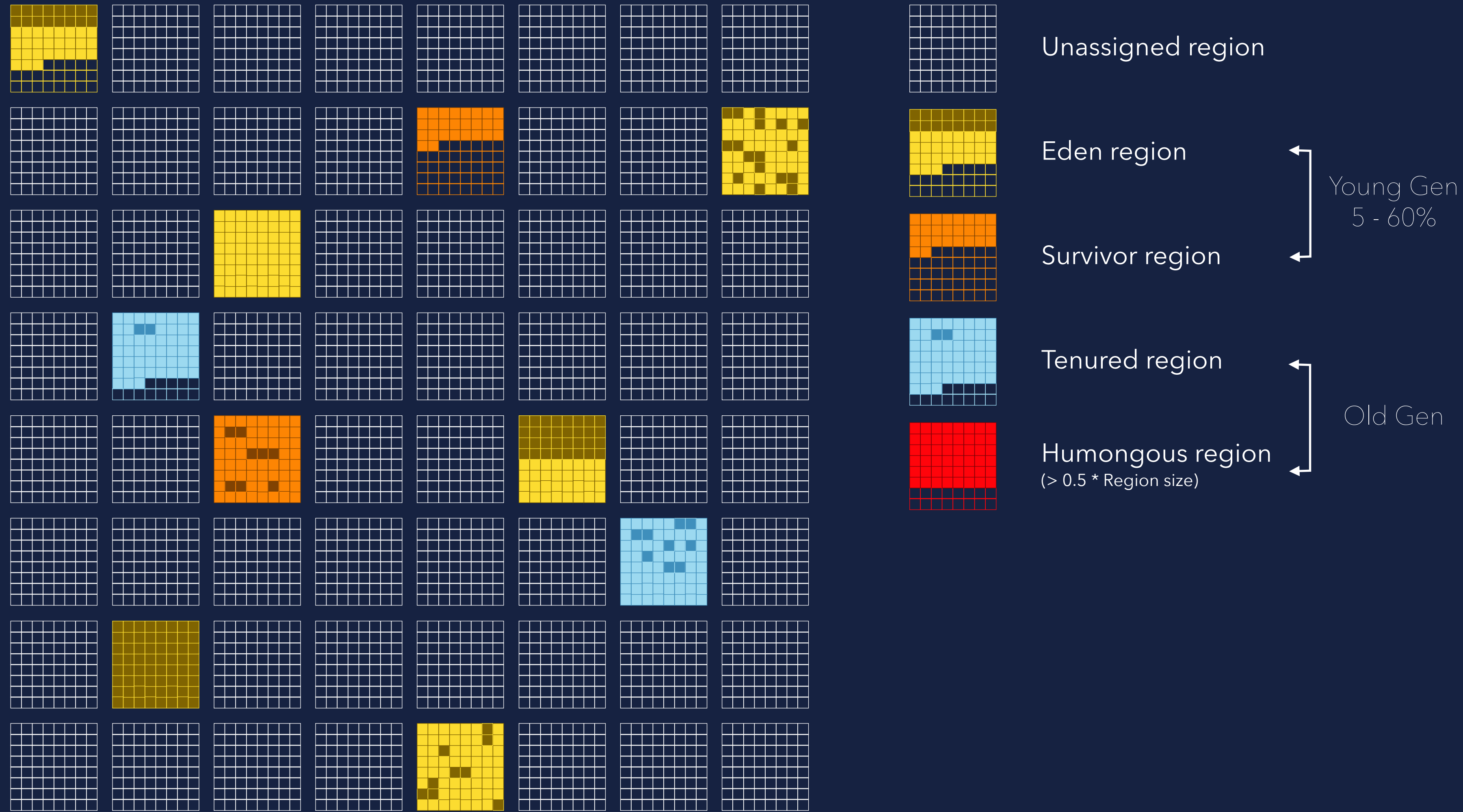
Max no. of region ≤ 2048

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

$8192 \text{ MB} / 2048 = 4 \text{ MB region size}$



Heap-Layout

Region size 1 - 32 MB

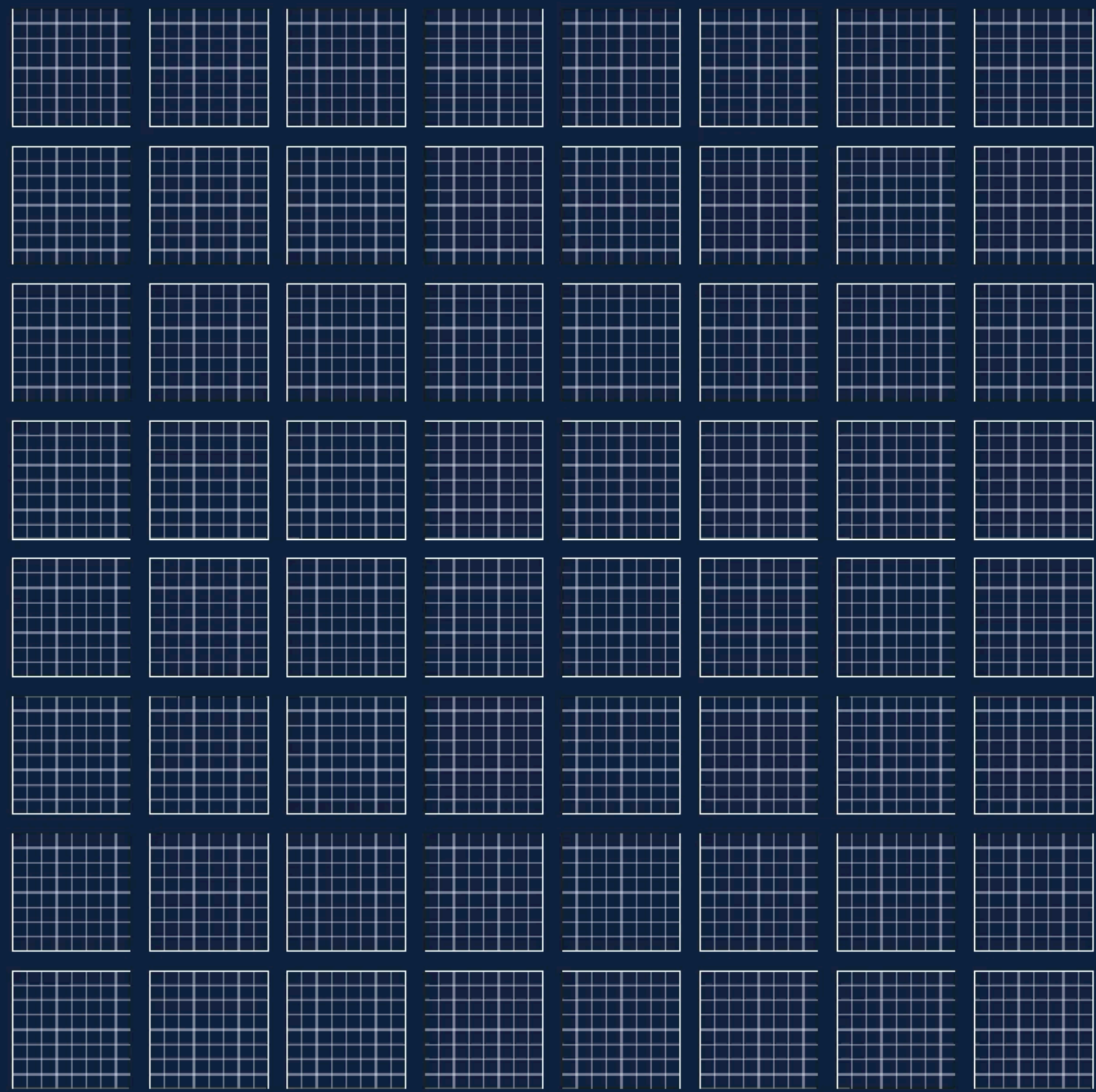
Max no. of region <= 2048

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

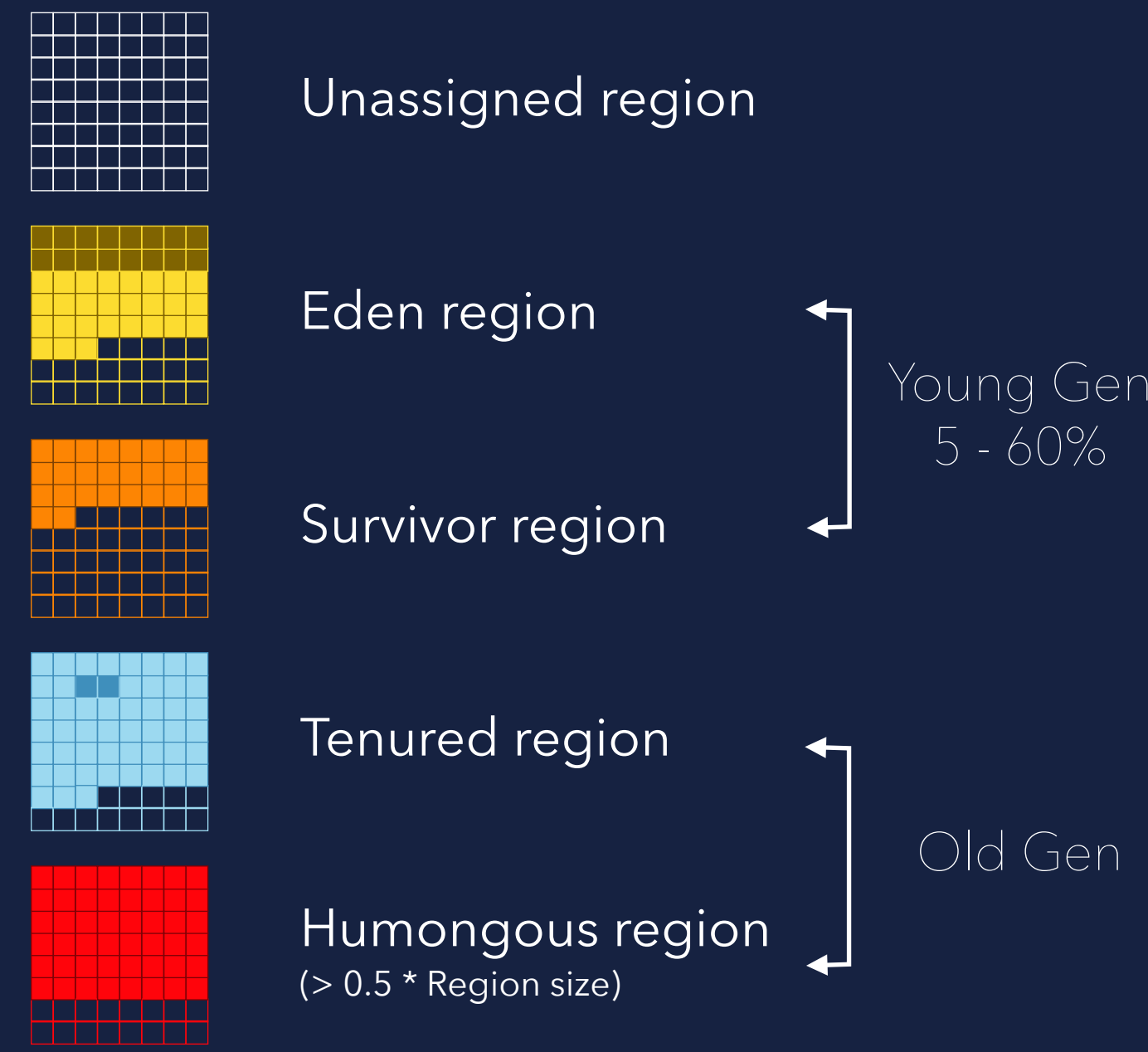
Example 8GB Heap:

8 GB Heap = 8192 MB

8192 MB / 2048 = 4 MB region size



Heap




Example:
6 Eden Regions
3 Survivor Regions

2 Regions with most garbage will be collected/promoted



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 should be around 200 ms
-  Heap size is not larger than 16-32 GB

BEST SUITED FOR

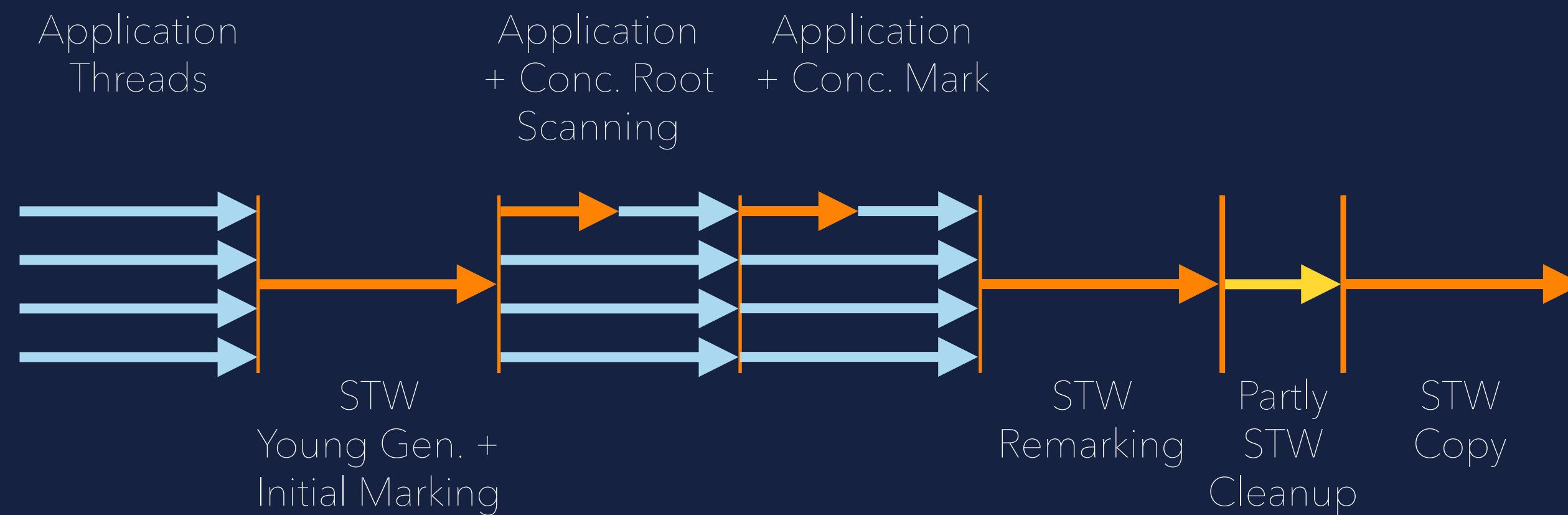
-  Mixed workloads
-  Large sized enterprise systems
-  Responsive in medium to large heaps

OS SUPPORT   

JVM SWITCH `> java -XX:+UseG1GC`

NOTES

-  Default collector from JDK 9 onwards
-  Concurrent marking





EPSILON



EPSILON

AVAILABILITY	JDK 11+
PARALLEL	-
CONCURRENT	-
GENERATIONAL	-
HEAP SIZE	-
PAUSE TIMES	-
THROUGHPUT	VERY HIGH
LATENCY	VERY LOW
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	<pre>> java -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC</pre>
------------	---



SHENANDOAH

SHENANDOAH



AVAILABILITY

JDK 11.0.9+ / JDK 24

PARALLEL

YES

CONCURRENT

FULLY

GENERATIONAL

NO / YES

HEAP SIZE

MEDIUM - LARGE

PAUSE TIMES

SHORT

THROUGHPUT

VERY HIGH




LATENCY

VERY LOW




CPU OVERHEAD

MODERATE (10-20%)

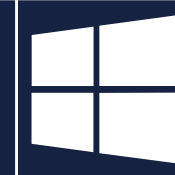
CHOOSE WHEN

-  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
- 🗑 First generational version in JDK 24

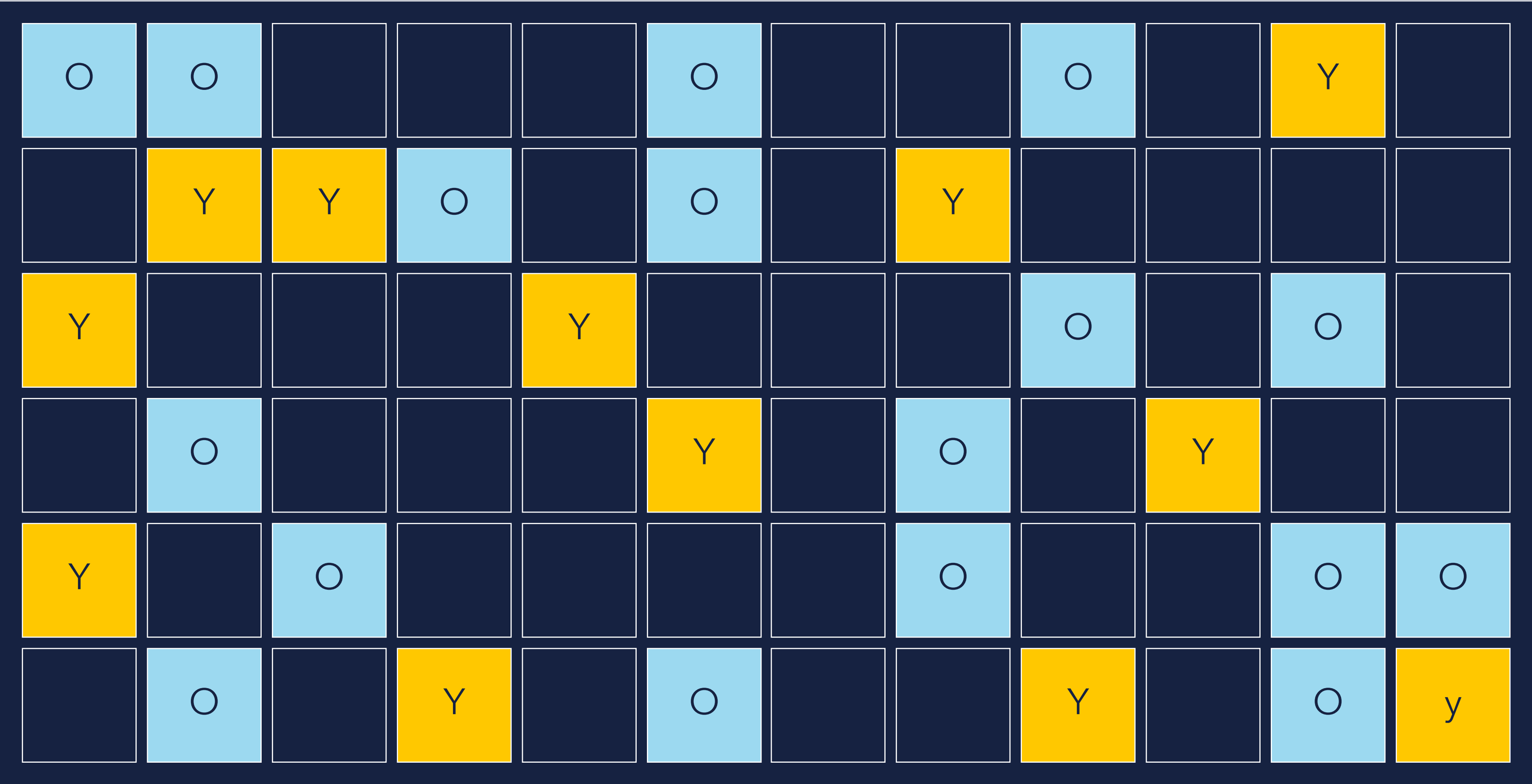


ZGC

Z Garbage Collector

Heap-Layout

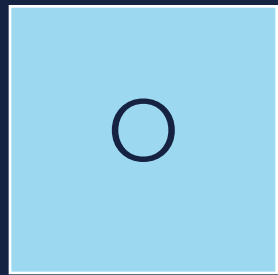
HEAP



EMPTY REGION



YOUNG GEN REGION



OLD GEN REGION

AVAILABILITY

JDK 15-23 / JDK 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%)

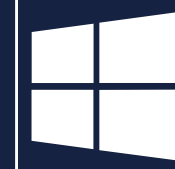
CHOOSE WHEN

-  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
```



C4

Concurrent Continues Compacting Collector

NOTES

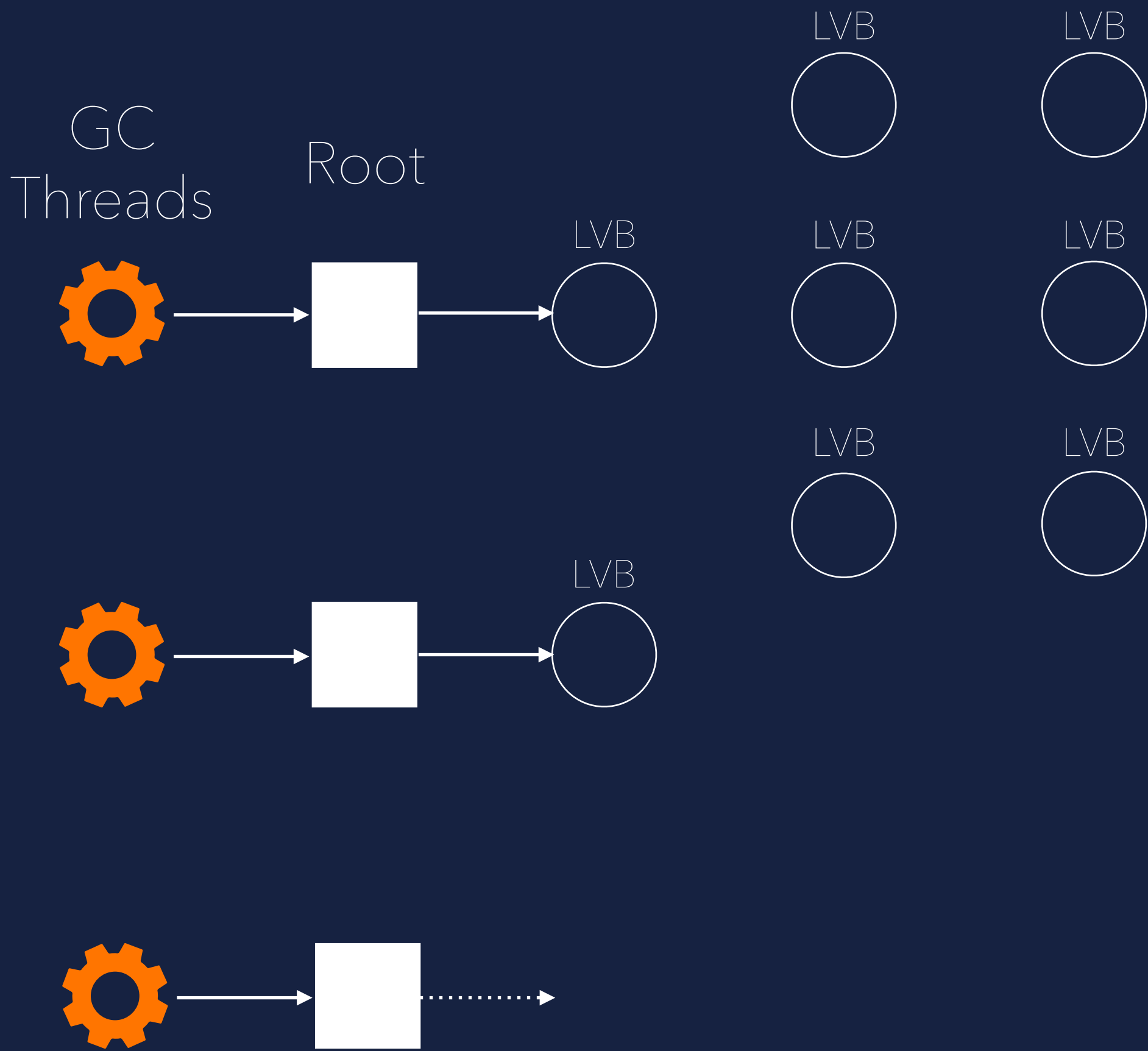
-  Part of Azul Zing JVM
-  Makes use of Loaded Value Barrier (LVB) everywhere
(Test + Jump which only takes 1 cpu cycle -> very fast)
-  LVB is read and write barrier
(guaranteed to be hit on every access)
-  Best performance by using Transparent Huge Pages
(Normal page size 4kB, THP size 2MB)

C4

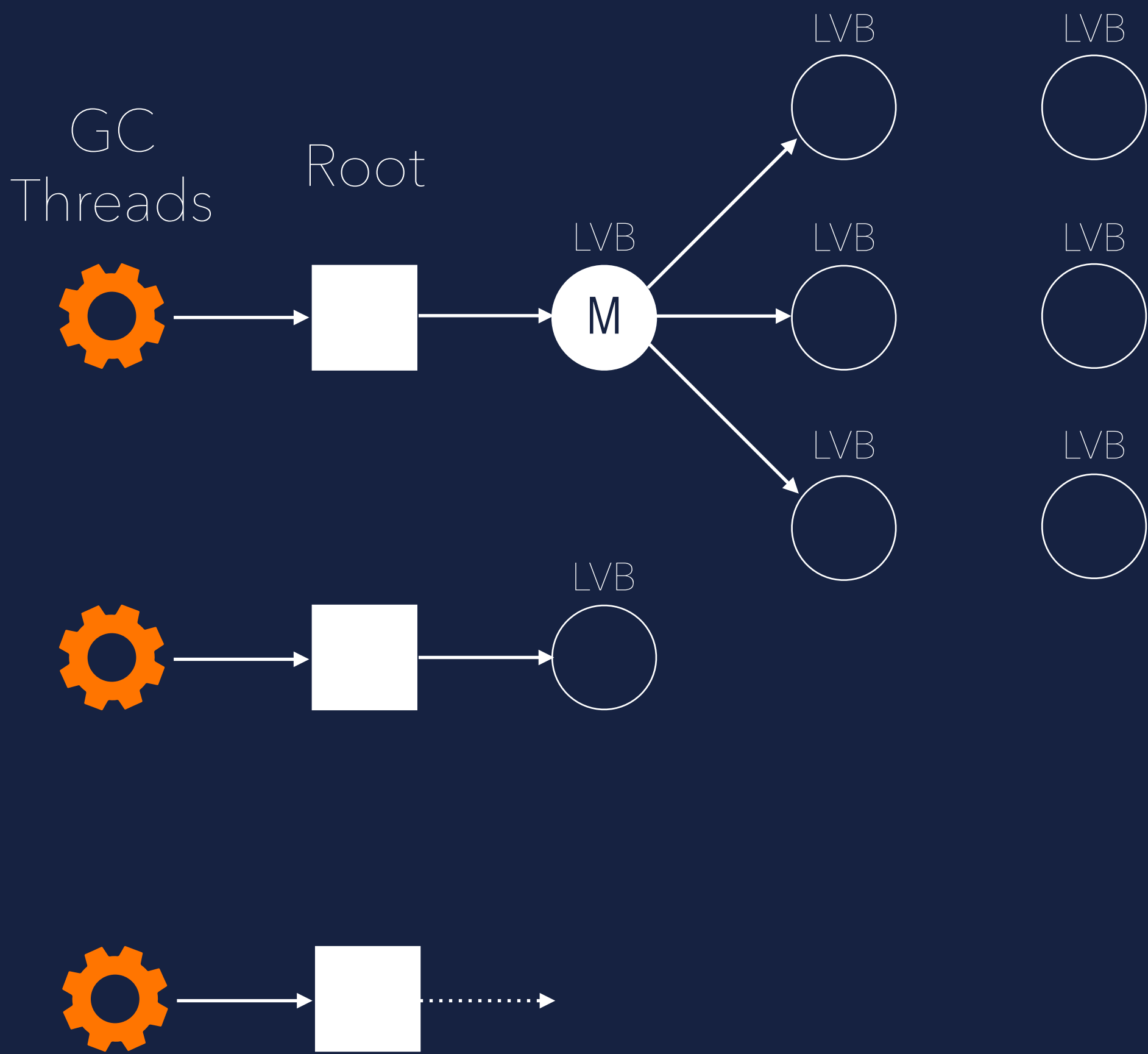


MARKING PHASE

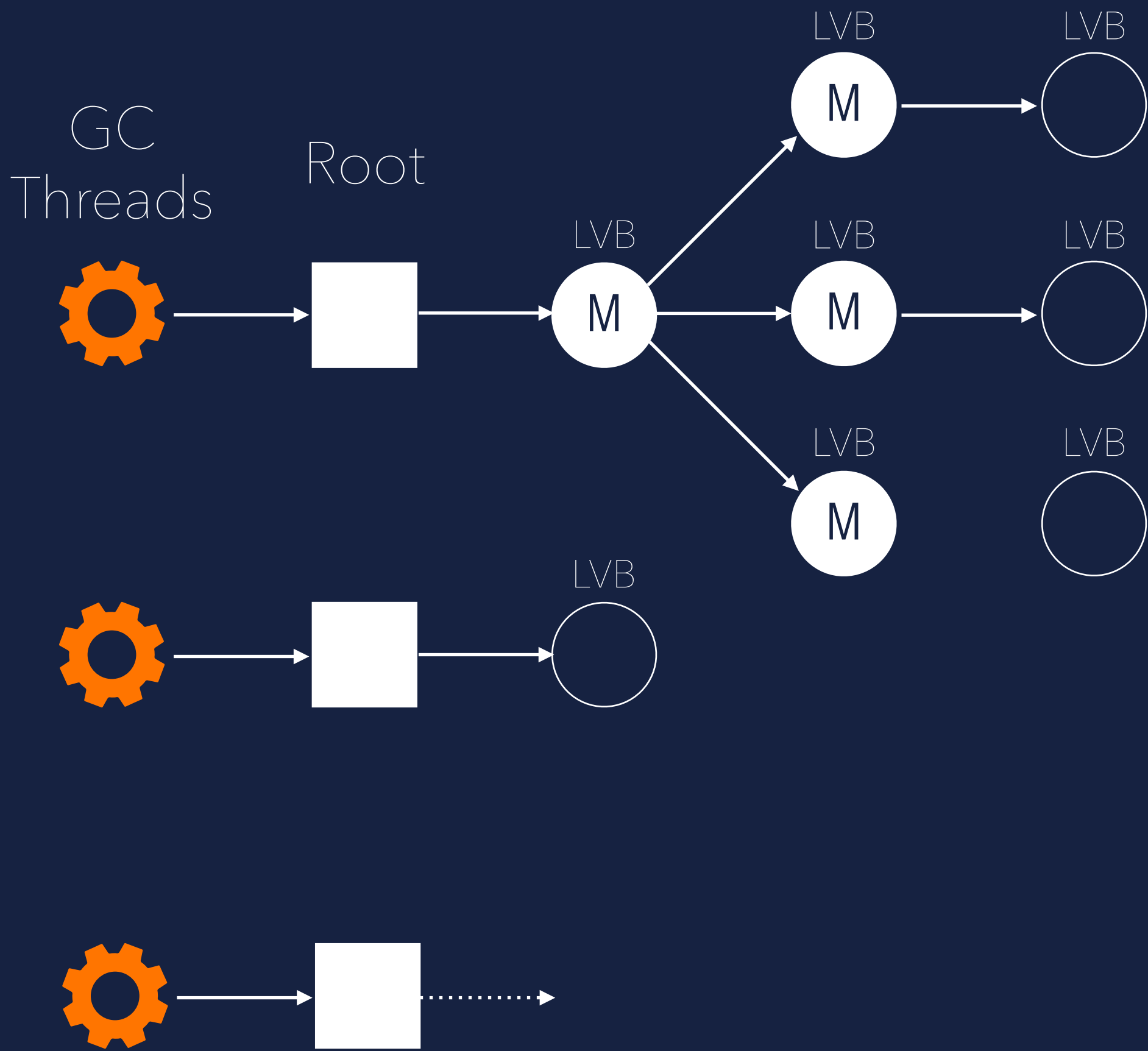
Marking Phase



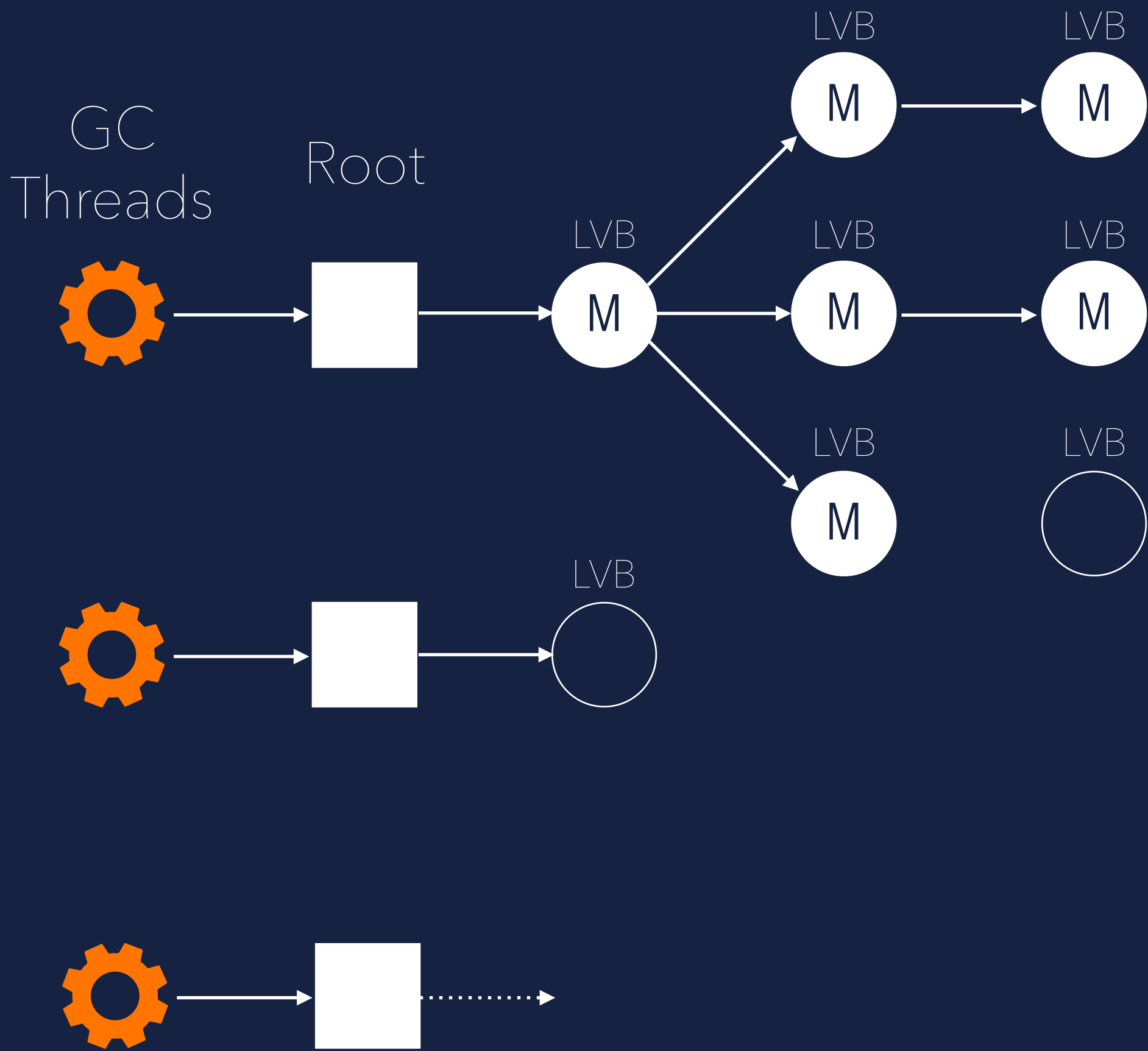
Marking Phase



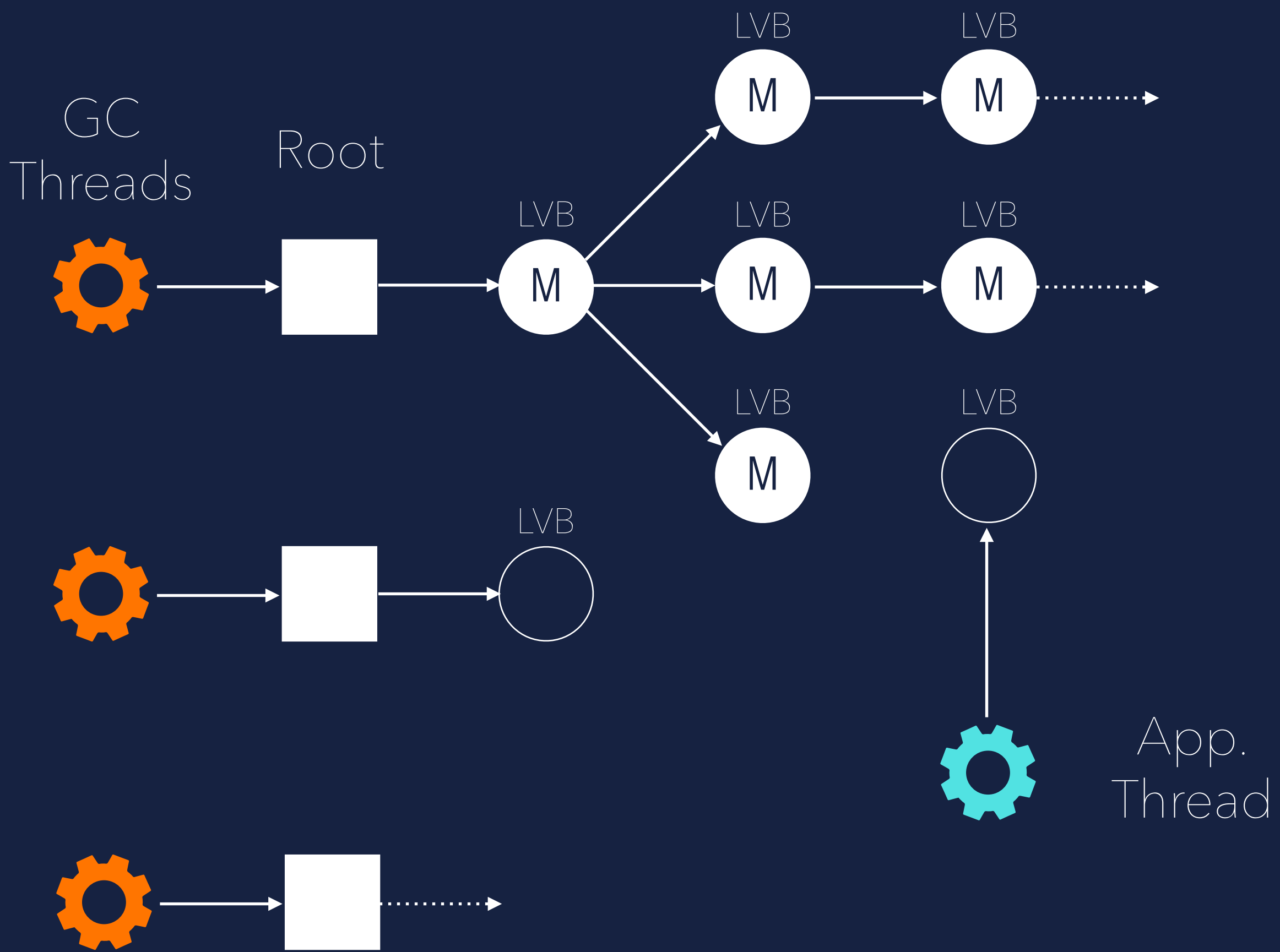
Marking Phase



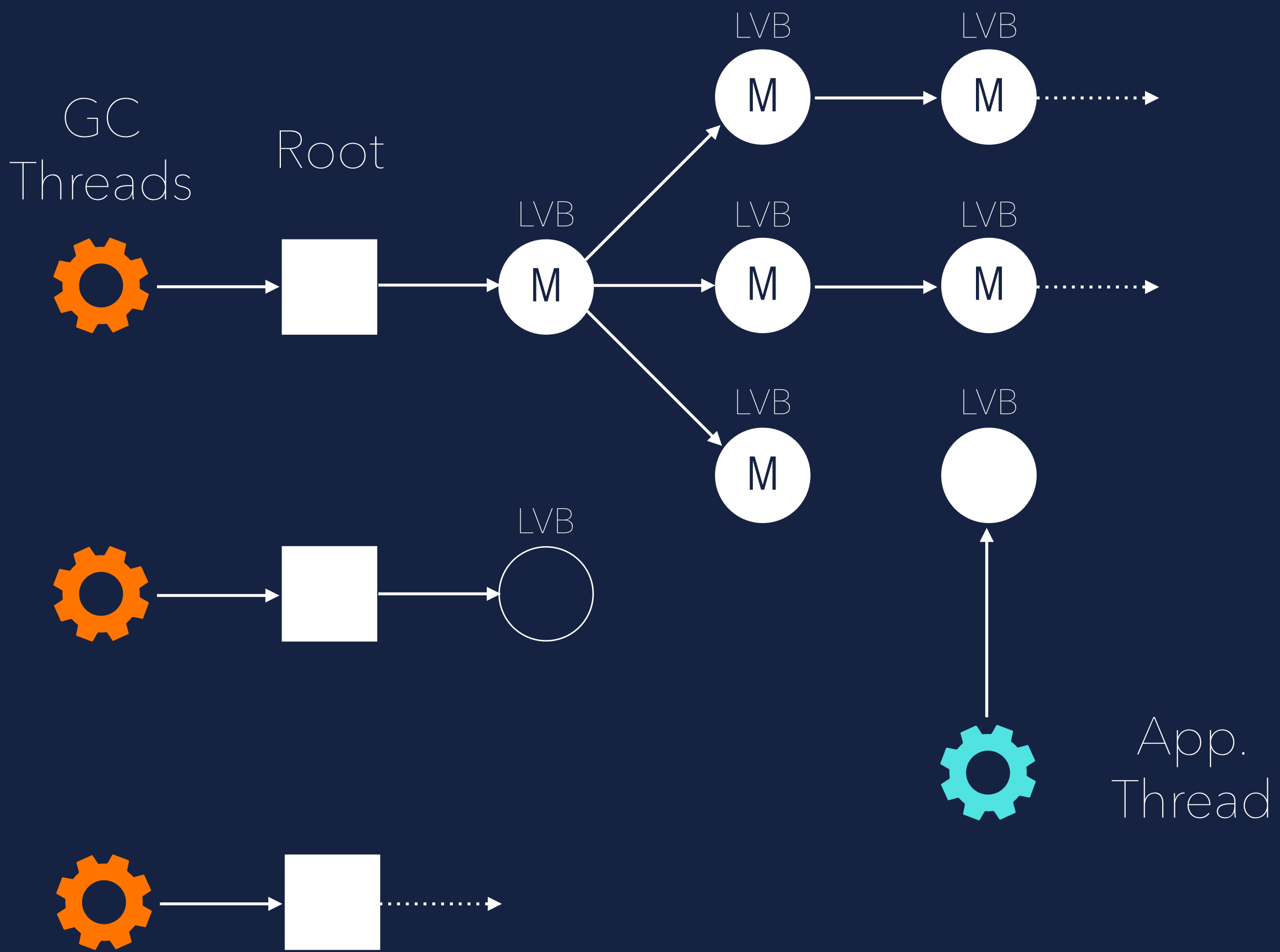
Marking Phase



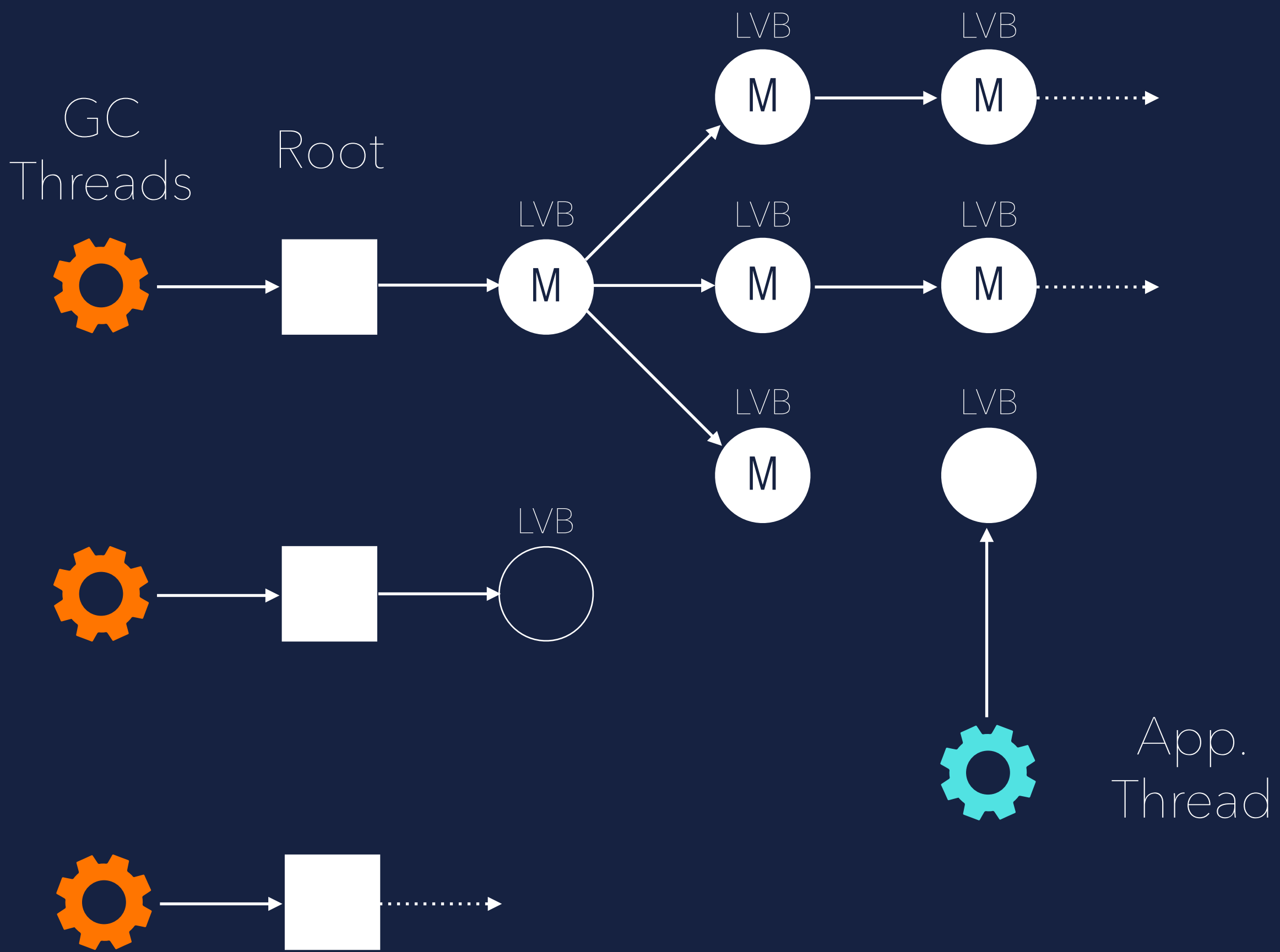
Marking Phase



Marking Phase

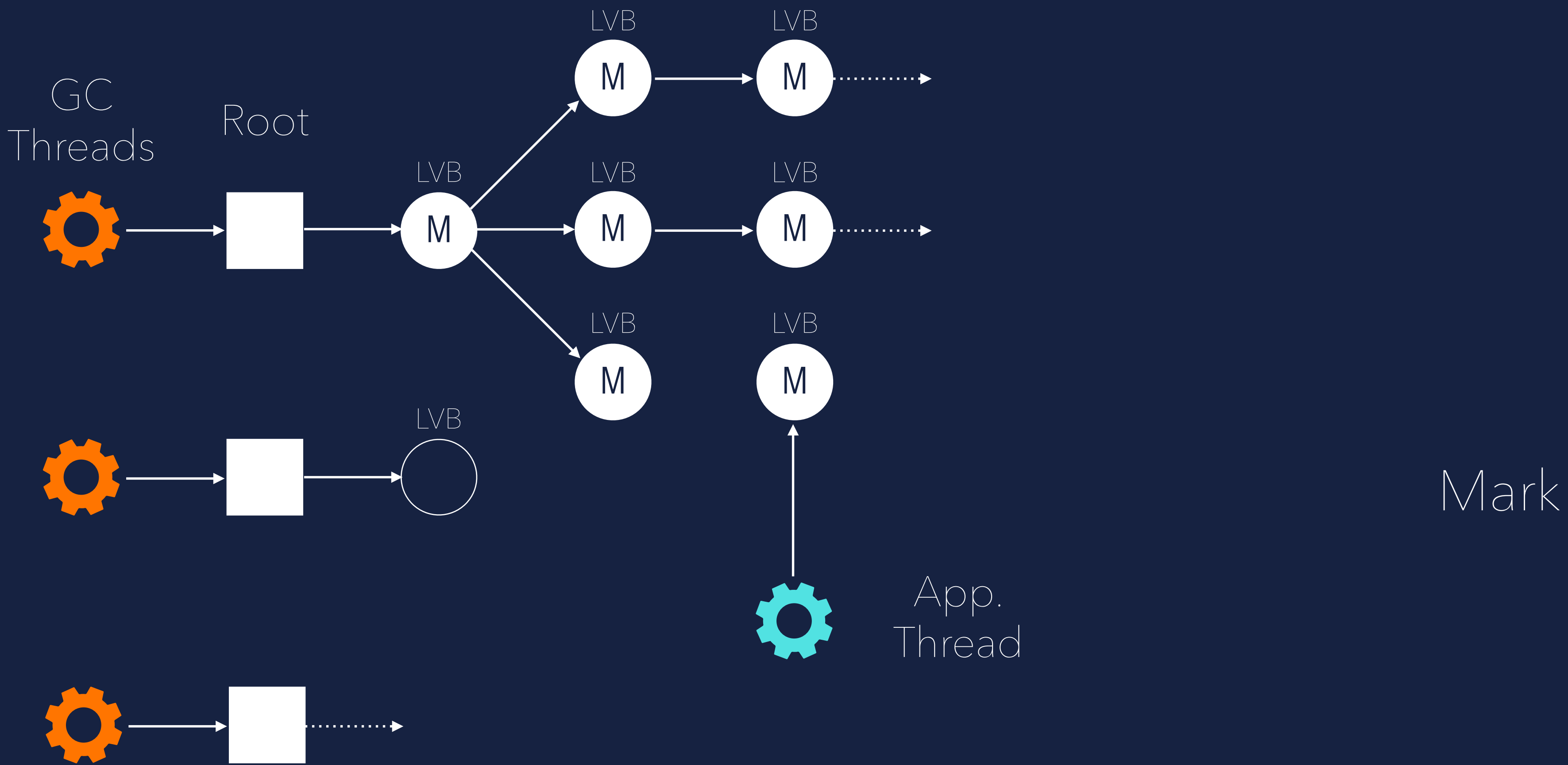


Marking Phase

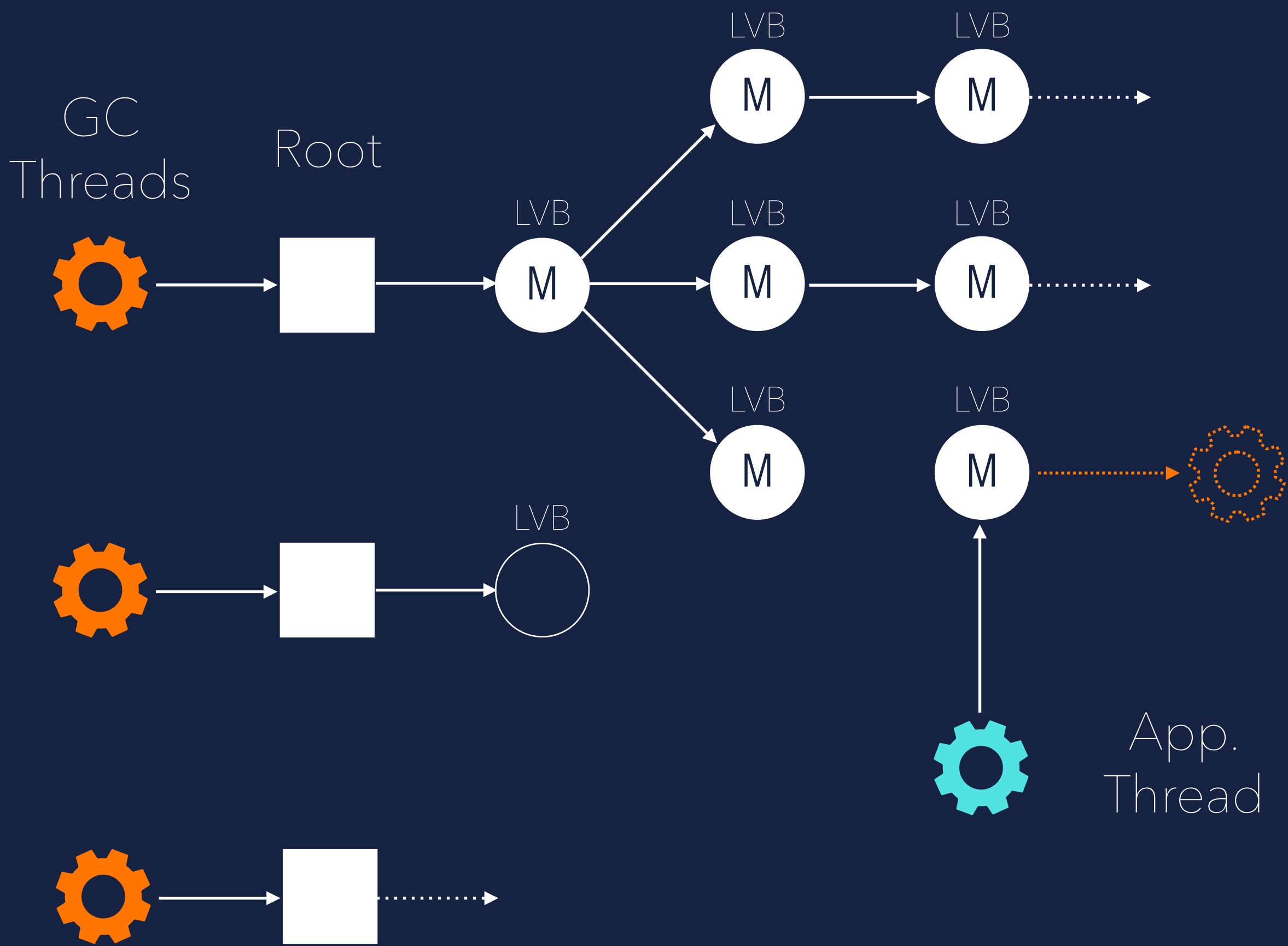


Test+Jump

Marking Phase

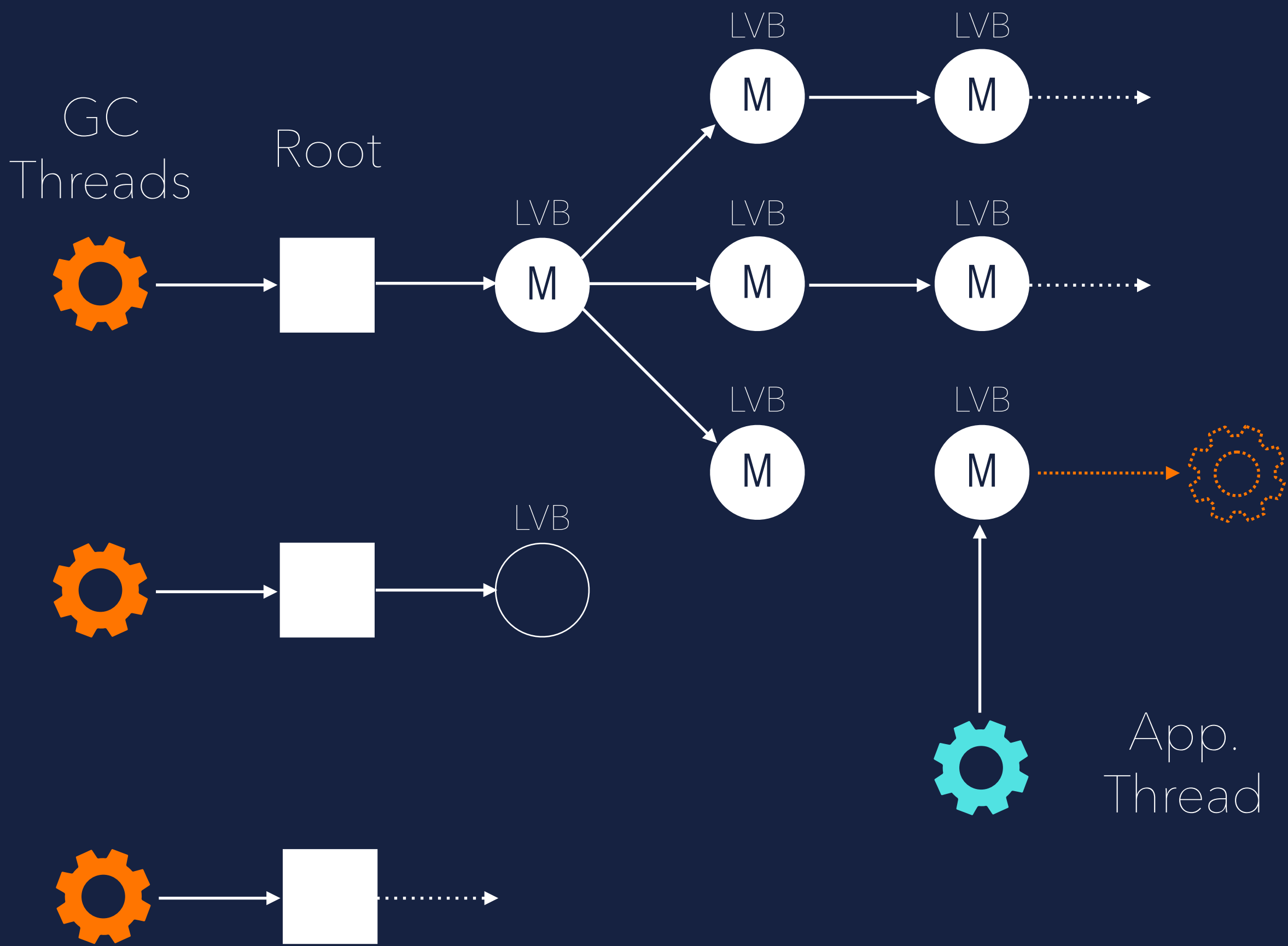


Marking Phase



Hand over
to GC

Marking Phase



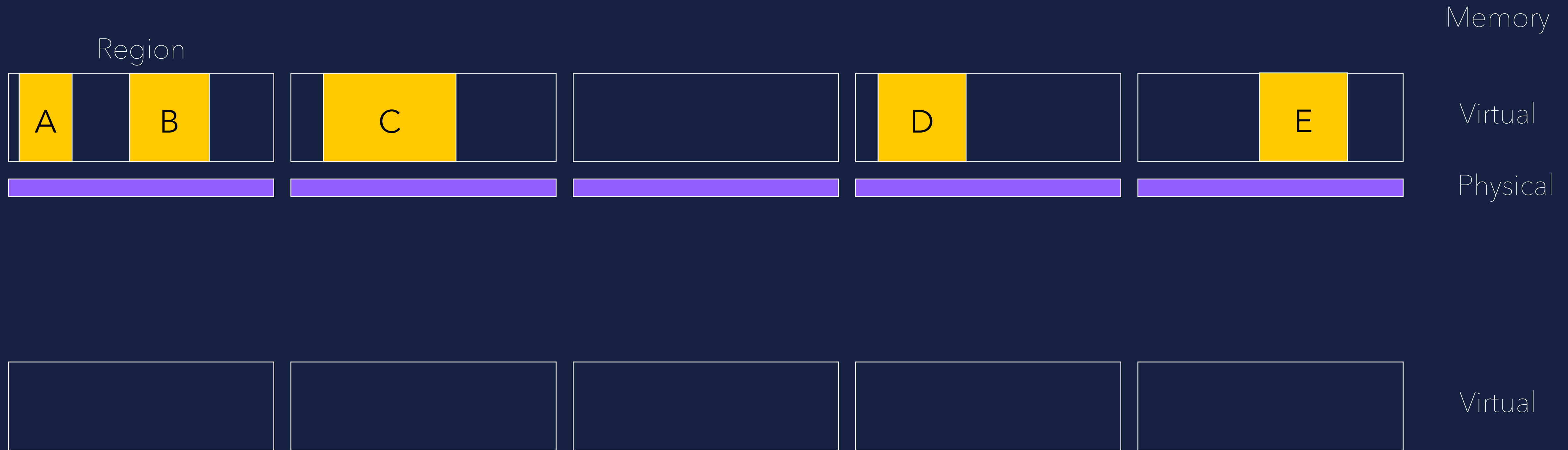
No need to mark again by GC !

C4

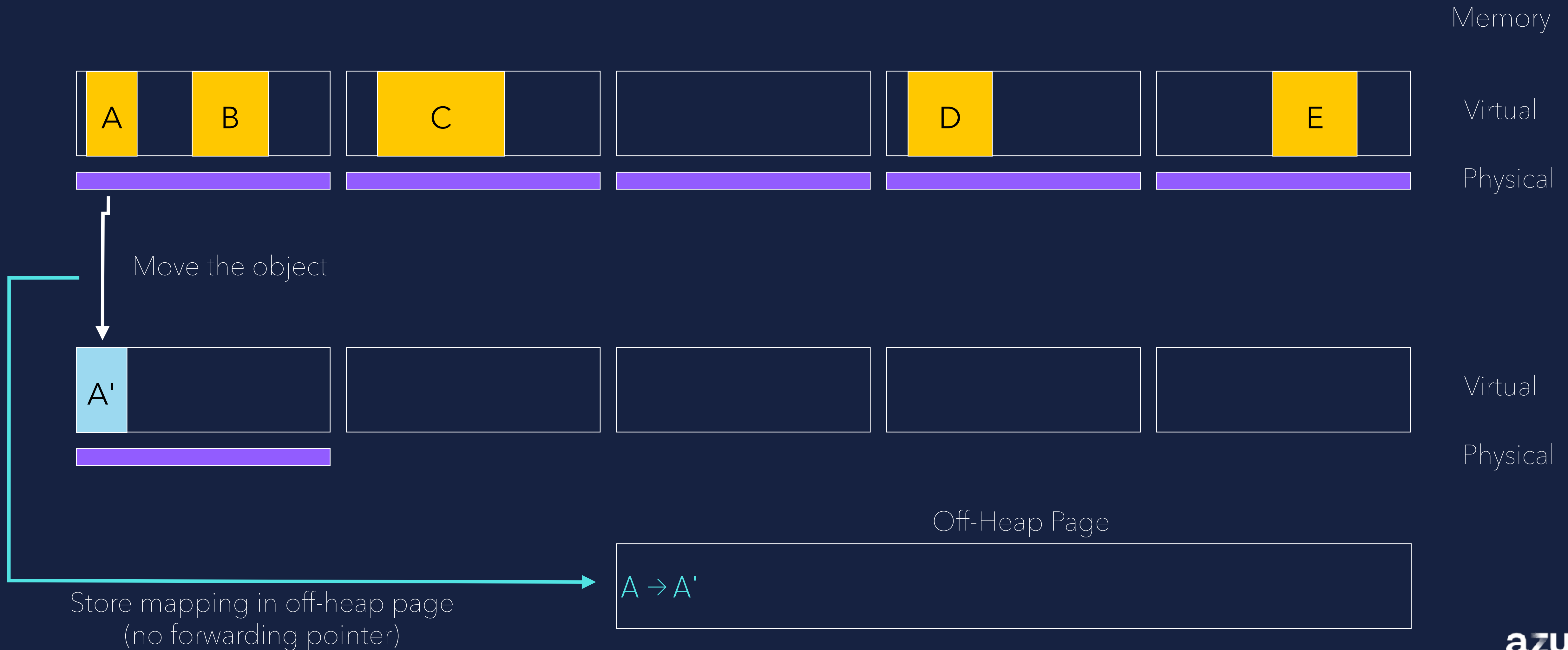


QUICK RELEASE

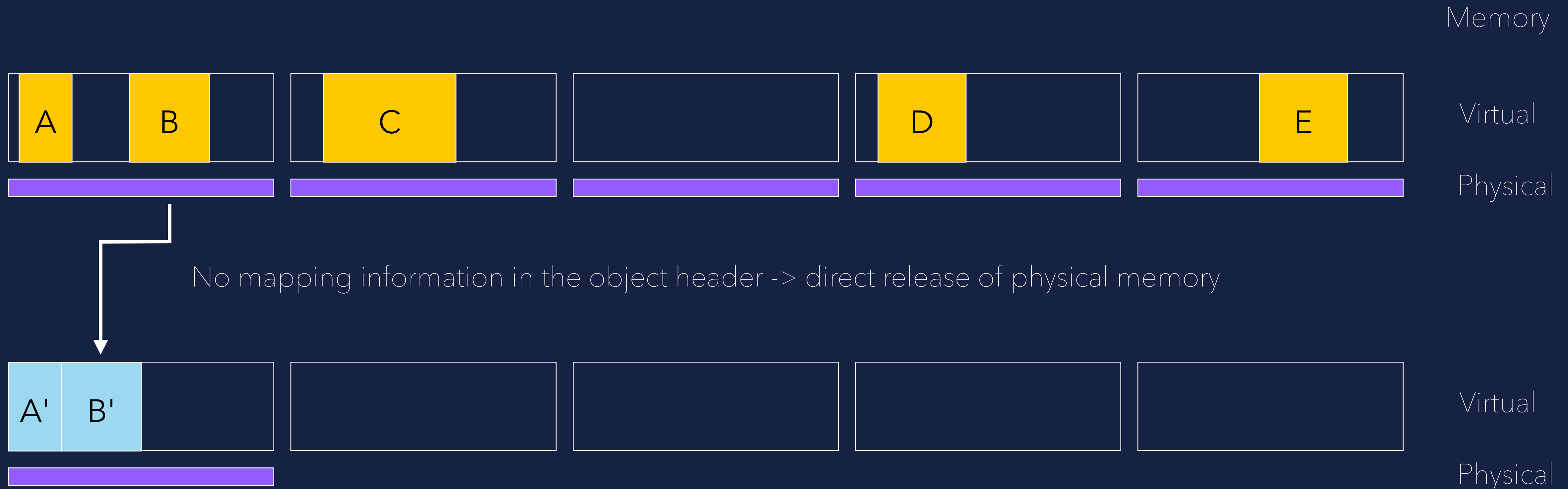
Relocation Phase



Relocation Phase (Compaction)



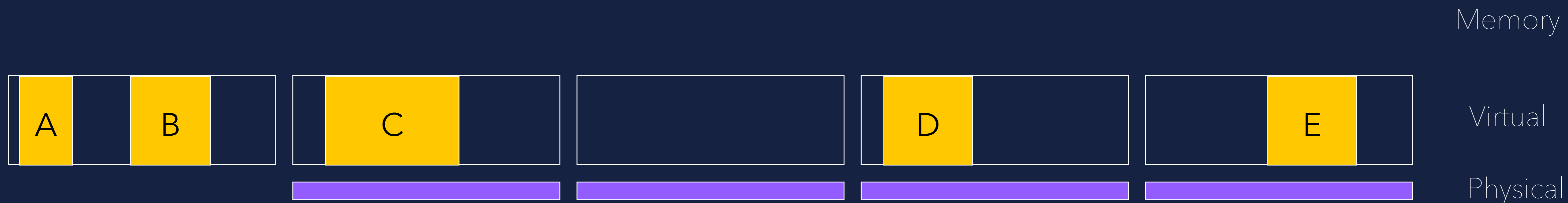
Relocation Phase (Compaction)



Off-Heap Page

A → A' B → B'

Relocation Phase (Compaction)



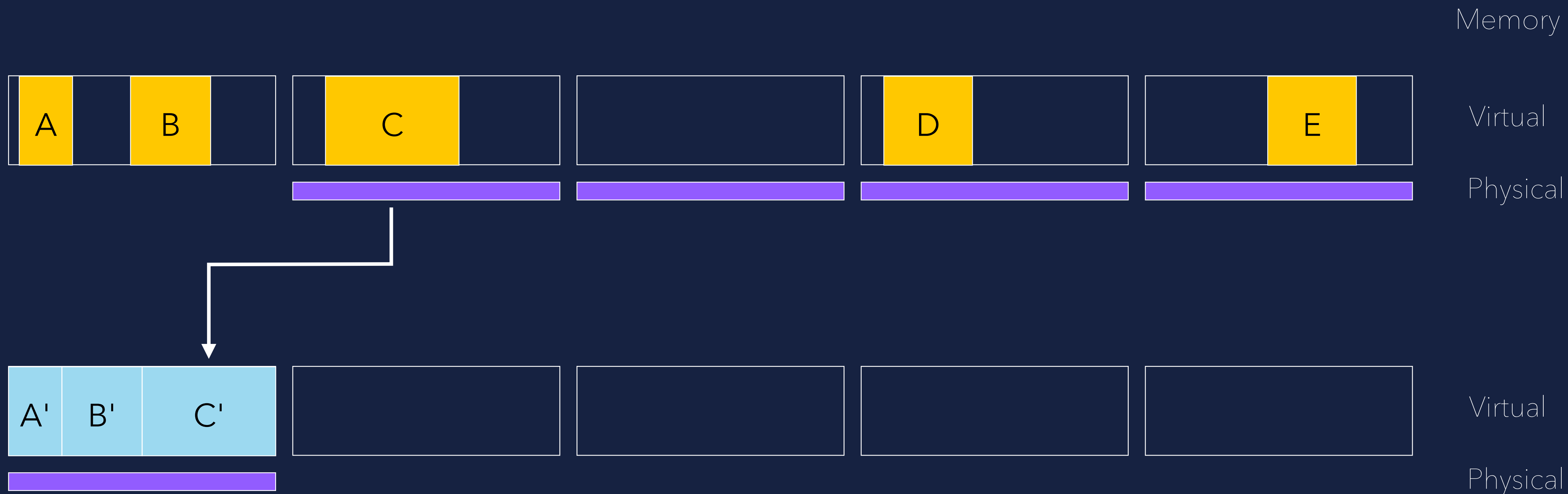
No mapping information in the object header -> direct release of physical memory



Off-Heap Page

A → A' B → B'

Relocation Phase (Compaction)

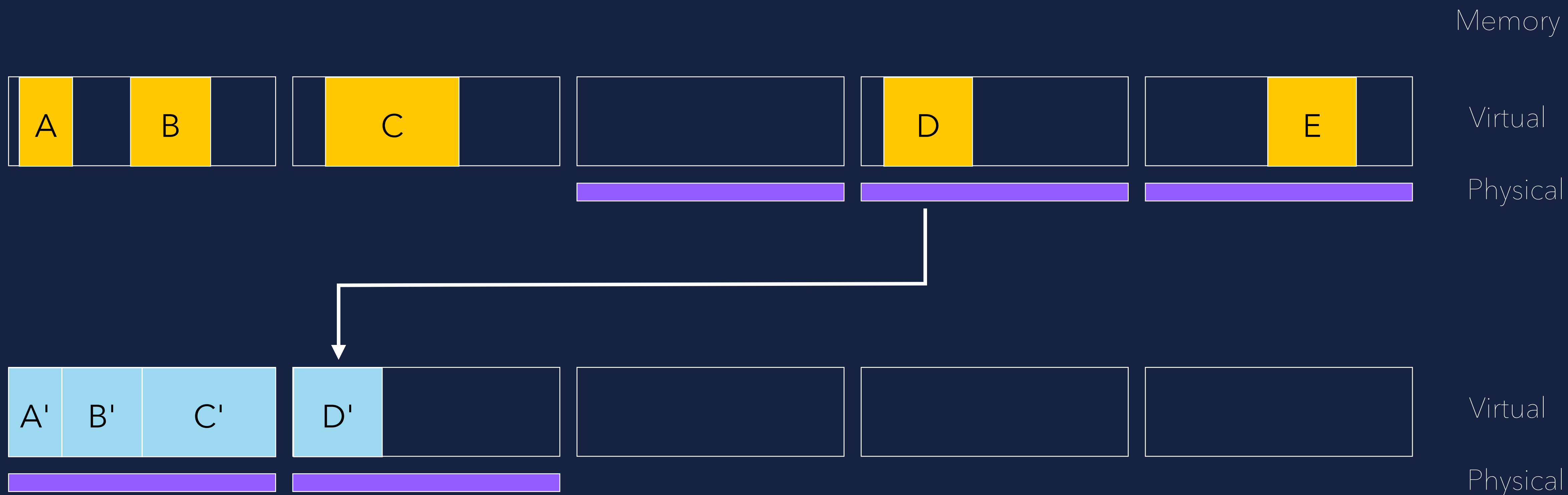


Off-Heap Page

$A \rightarrow A' \quad B \rightarrow B' \quad C \rightarrow C'$



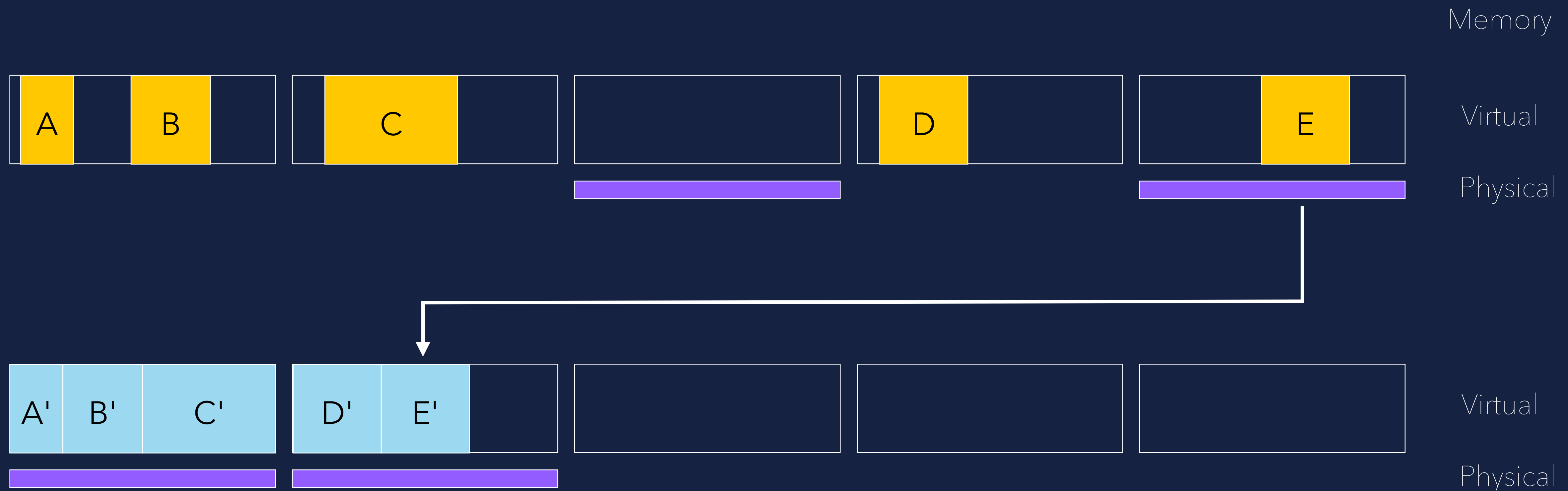
Relocation Phase (Compaction)



Off-Heap Page

$A \rightarrow A'$ $B \rightarrow B'$ $C \rightarrow C'$ $D \rightarrow D'$

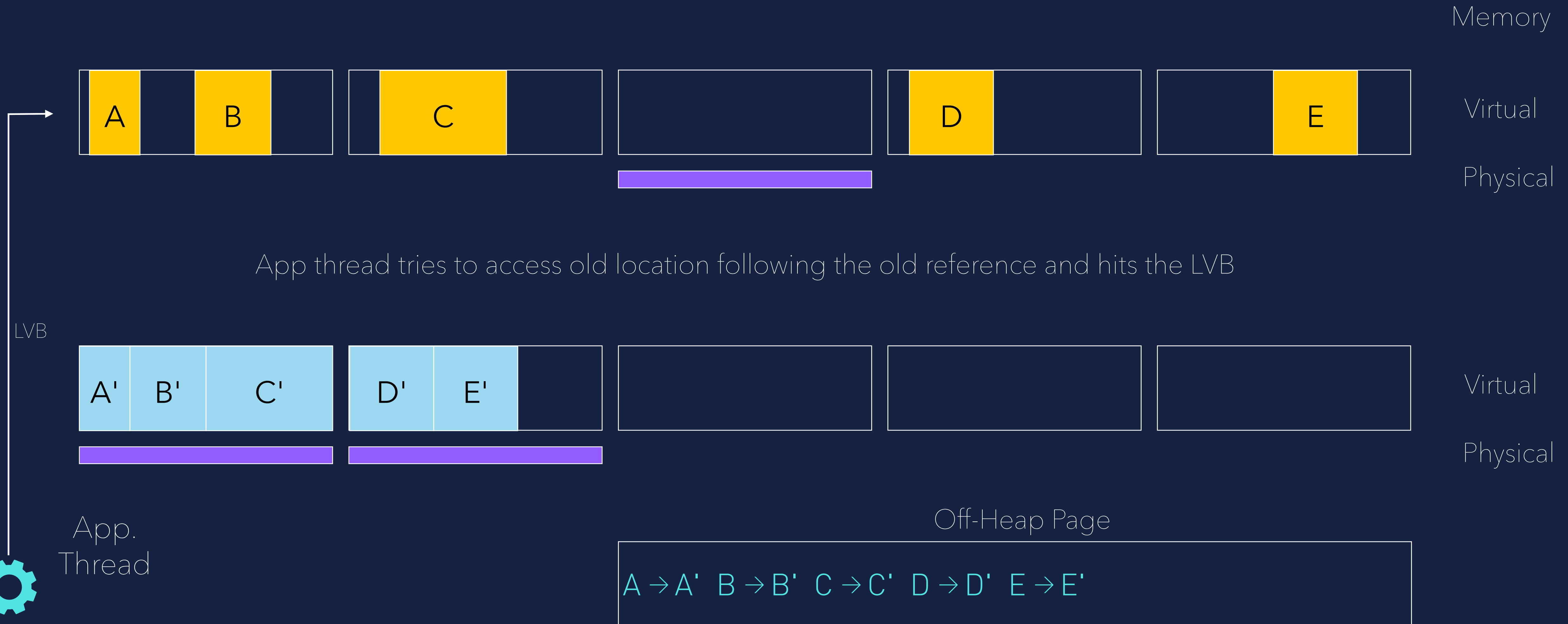
Relocation Phase (Compaction)



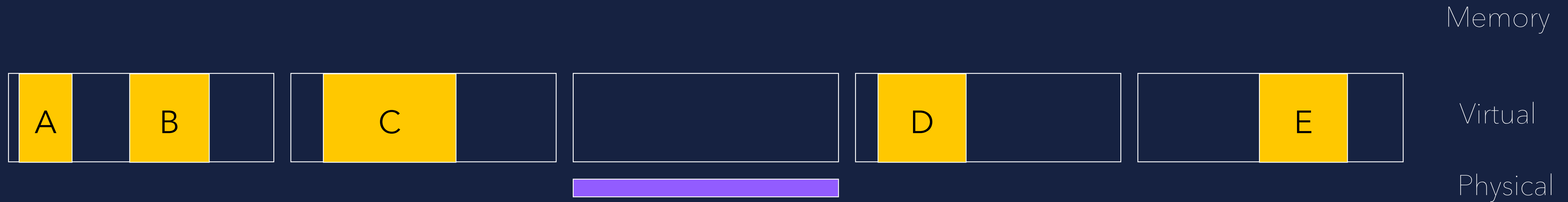
Off-Heap Page

$A \rightarrow A'$ $B \rightarrow B'$ $C \rightarrow C'$ $D \rightarrow D'$ $E \rightarrow E'$

Relocation Phase (Quick Release)



Relocation Phase (Quick Release)

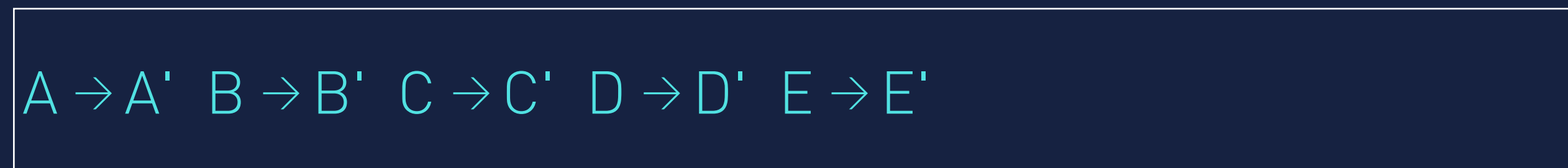


Gets new location from Off-Heap forwarding page

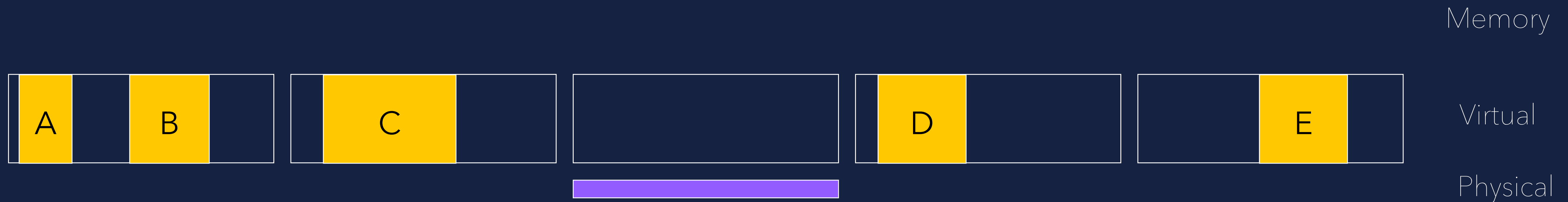


App.
Thread

Off-Heap Page



Relocation Phase (Quick Release)



Updates the reference and can access object at new location



App.
Thread

Off-Heap Page

$A \rightarrow A'$ $B \rightarrow B'$ $C \rightarrow C'$ $D \rightarrow D'$ $E \rightarrow E'$

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

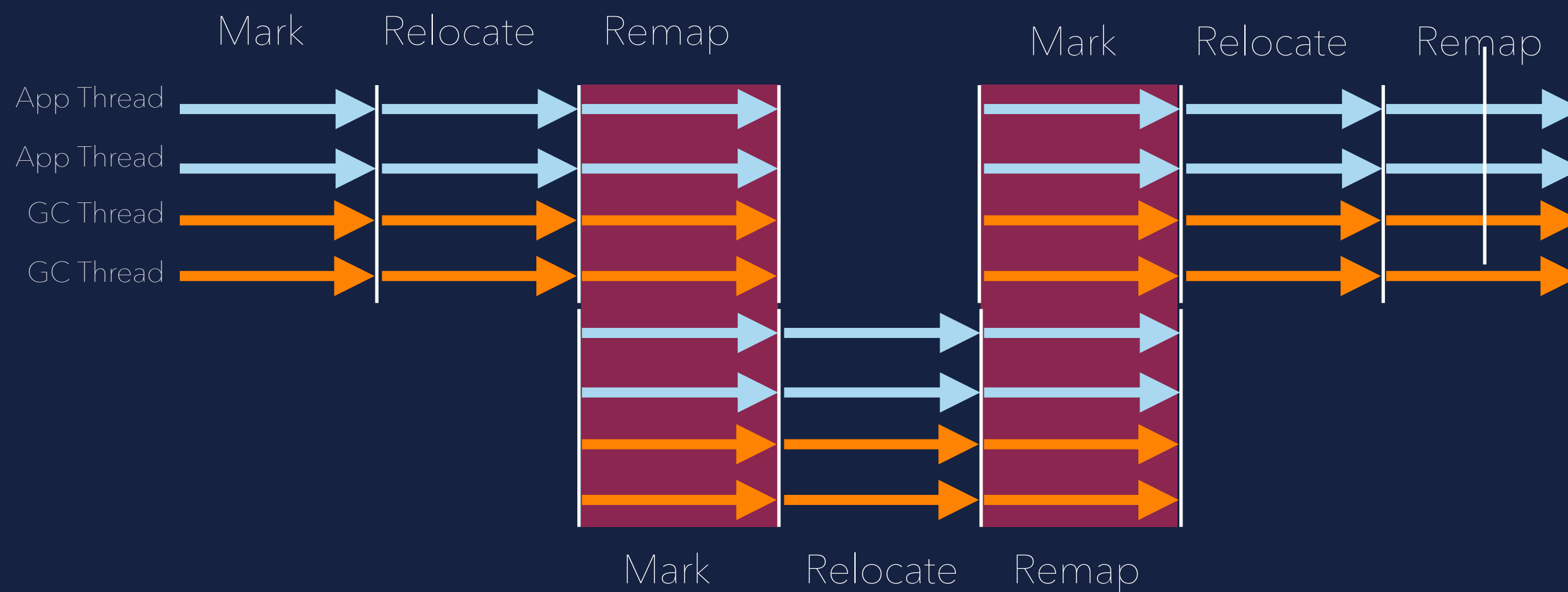
- 🗑️ Low latency sensitive applications
- 🗑️ Large scale systems
- 🗑️ Highly concurrent applications

OS SUPPORT			
------------	--	--	--

JVM SWITCH	> -
------------	-----

NOTES

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



WHICH ONE...?

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

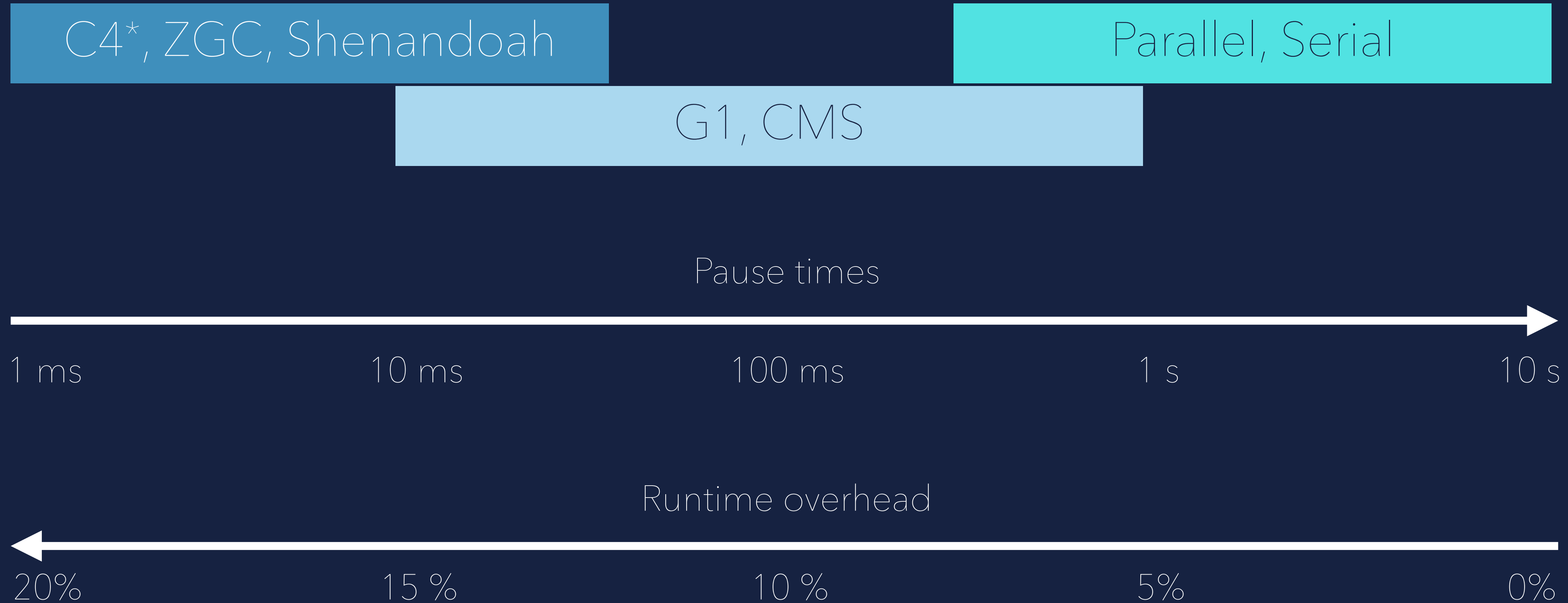
WHICH ONE...?

Essential Criteria



WHICH ONE...?









Choose dependent on your workload



* C4 has less overhead due to faster Falcon compiler

OVERVIEW

OVERVIEW

	Serial GC	Parallel GC	CMS GC	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 Zing 8+
Parallel	NO	YES	YES	YES	NO	YES	YES	YES
Concurrent	NO	NO	PARTIALLY	PARTIALLY	NO	FULLY	FULLY	FULLY
Generational	YES	YES	YES	YES	NO	NO	NO / YES	YES
Heap Size	SMALL - MEDIUM	MEDIUM - LARGE	MEDIUM - LARGE	MEDIUM - LARGE	NO	LARGE	VERY LARGE	VERY LARGE
Pause Times	LONGER	MODERATE	MODERATE	SHORT - MEDIUM	NO	VERY SHORT (<10ms)	VERY SHORT (<1ms)	VERY SHORT (<1ms)
Throughput	LOW	VERY HIGH	MODERATE	HIGH	VERY HIGH	VERY HIGH	VERY HIGH	VERY HIGH
Latency	HIGHER	LOWER	MODERATE	LOWER	NO	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	MODERATE - HIGH	MODERATE - HIGH
Tail latency	HIGH	HIGH	HIGH	HIGH	NO	MODERATE	LOW	LOW

TOOLING...

TOOLING



`-Xlog:gc*:/gc-%p-`

`%t.log:tags,uptime,time,level:filecount=10,filesize=128m`

Output of garbage collection details to log file



`jstat`

Tool that provides info on performance and resource consumption of running applications

TOOLING



JITWatch

A tool for understanding the JVM JIT (<https://github.com/AdoptOpenJDK/jitwatch/wiki>)



jHiccup

A non intrusive tool to monitor platform "hiccups" incl. JVM stalls (<https://github.com/giltene/jHiccup>)



VisualVM

All in one Java troubleshooting tool (<https://visualvm.github.io/>)



GCeasy

Universal GC Log Analyzer (<https://gceasy.io>)



JProfiler

All in one Java profiler (<https://www.ej-technologies.com/jprofiler>)



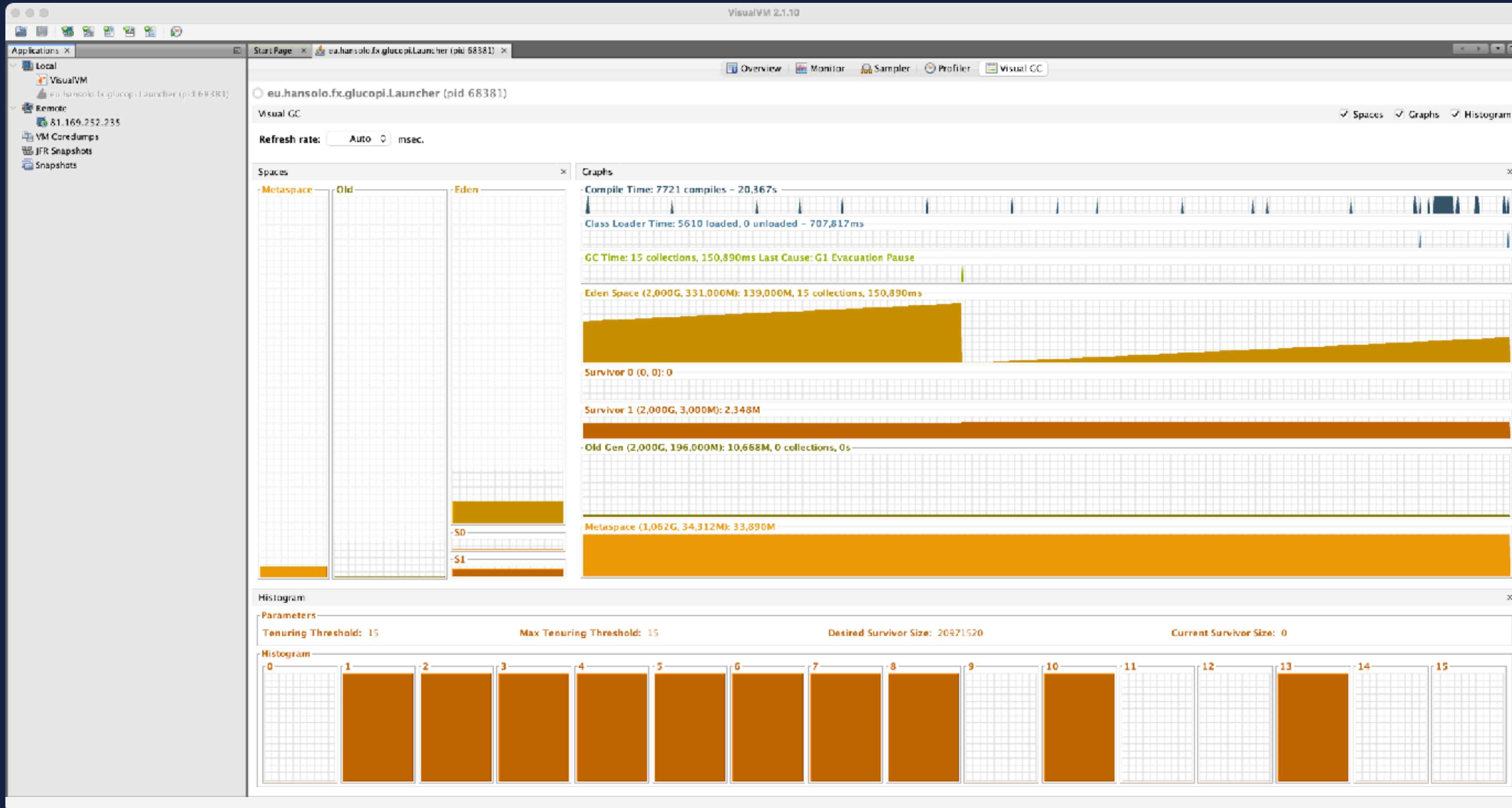
YourKit Java Profiler

CPU and Java profiler (<https://www.yourkit.com/features/>)



TOOLING

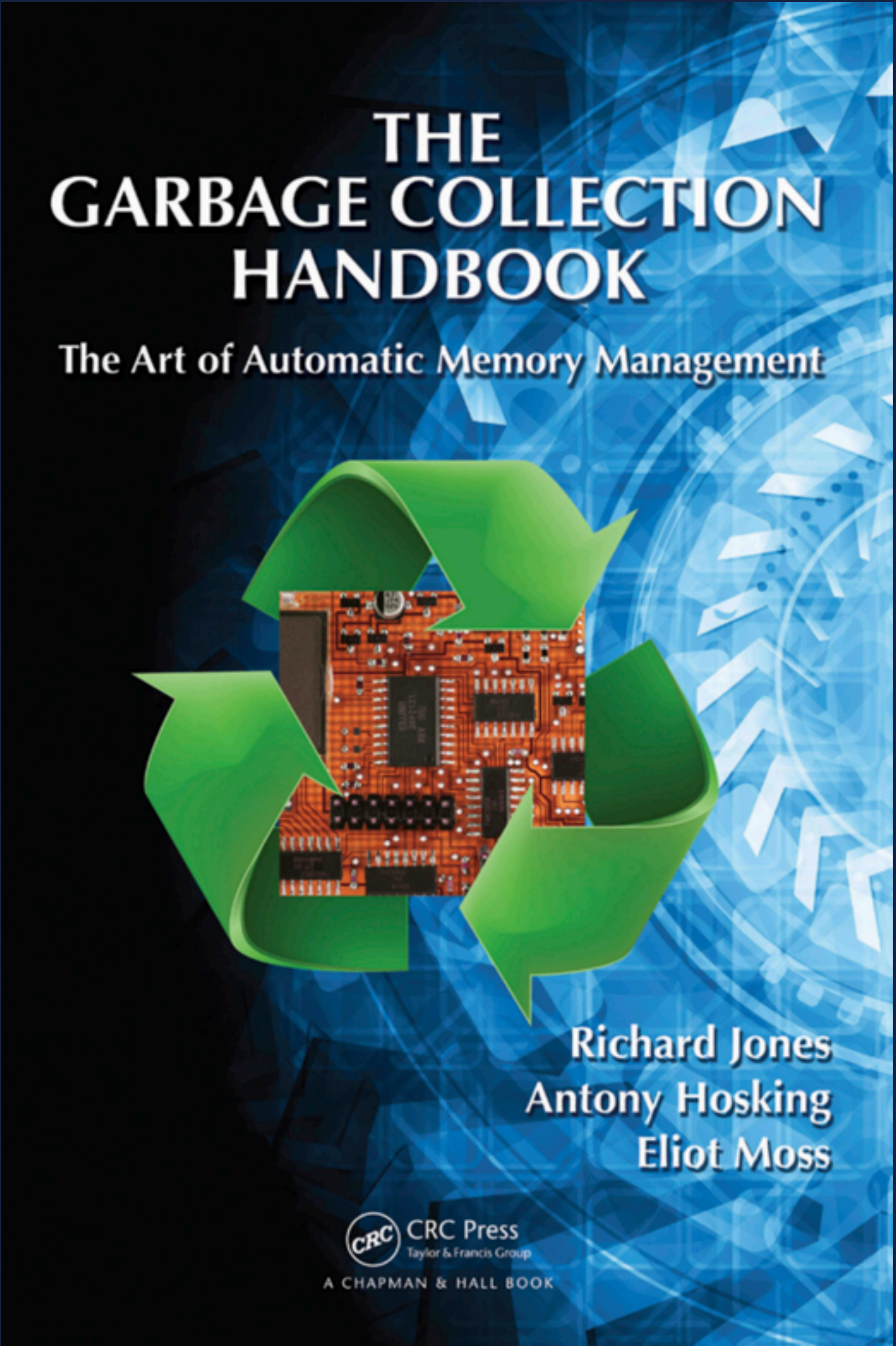
VisualVM + VisualGC plugin



**WANNA KNOW
MORE ?**

WANNA KNOW MORE ?

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



THANK YOU

