# Concurrent copying & compaction

Hanan Rofe Haim

6/1/15

# Outline

## Concurrent Copying

- Reminder
- Baker's algorithm
- Brook's indirection barrier
- Replication copying
- Multi-version copying
- Sapphire algorithm- basics

## Concurrent Compaction

- Reminder
- Compressor
- Pauseless collector

- **Discussion**
- **Summery**

# Copying- recap

Courtesy of Jonathan Kalechstain

Heap divided into 2 equally sized semi spaces:

The From-space and the To-space.

The To-space acts as the heap de facto.

From-space is empty between collection cycles.

During a collection cycle:

1. The To-space becomes the From-space (flip).
2. Live objects are copied to the new empty To-space.
3. References are updated
4. From-space is cleared, collecting all dead objects in the process.

# Copying- recap

## Definitions

**White node-** not yet processed/garbage.

**Grey node-** a node copied to the To-space.

**Black node-** a node whose pointers were updated (meaning all of its children were processed)

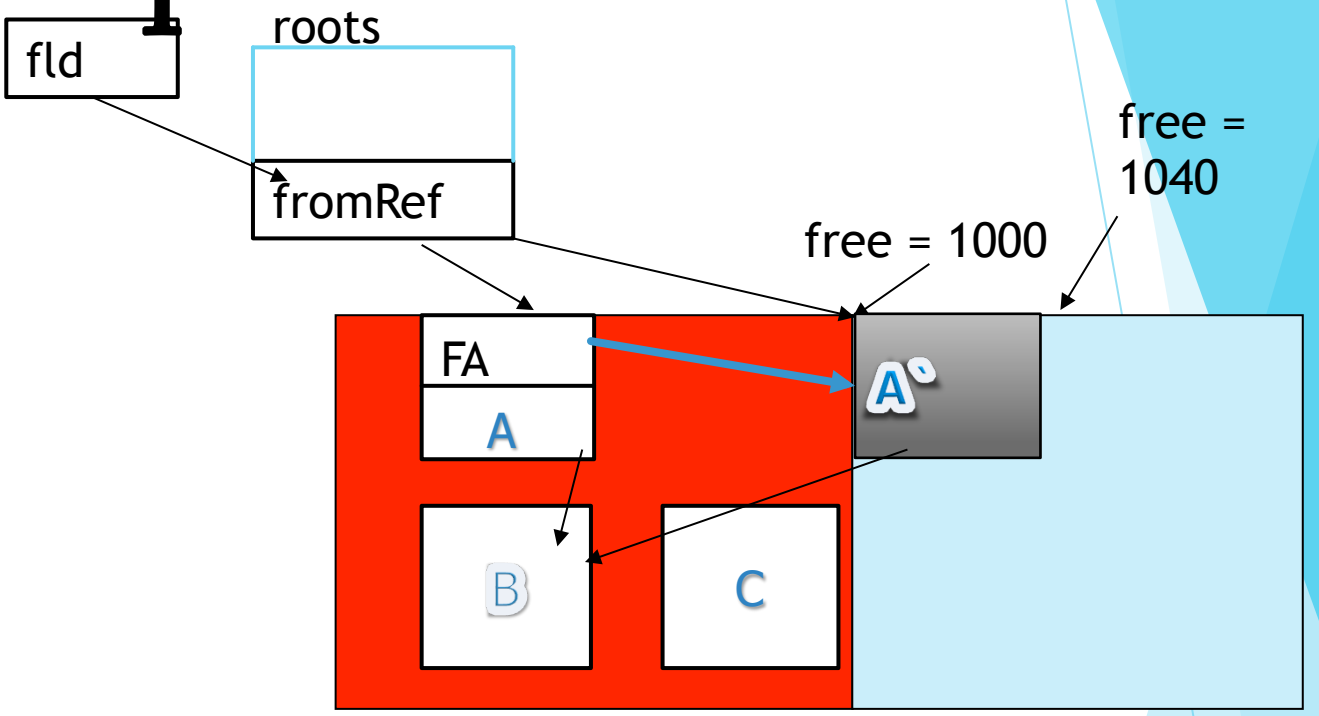
# Copying- recap

## Test run

```
process(fld):
        fromRef ⟵ *fld
        if fromRef != null
                *fld ⟵ forward(fromRef)

forward(fromRef):
        toRef ⟵ forwardingAddress(fromRef)
        if toRef = null
                toRef ⟵ copy(fromRef)
        return toRef

Copy(fromRef):
        toRef ⟵ free
        free ⟵ free+size(fromRef)
        move(fromRef,toRef)
        forwardingAddress(fromRef) ⟵ toRef
```




1. fromRef = &A
2. toRef = null
3. toRef = 1000
4. free = free + 40 = 1040
5. FA(fromRef)=1000
6. Return 1000

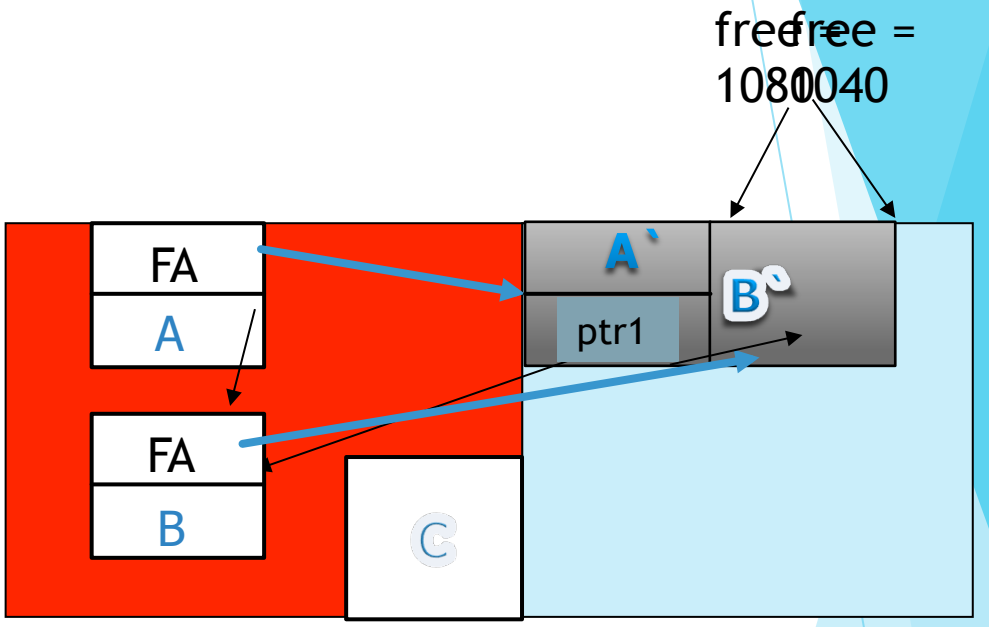
# Copying- recap

## Test run

```
process(fld):
        fromRef ⟵ *fld
        if fromRef != null
                *fld ⟵ forward(fromRef)

forward(fromRef):
        toRef ⟵ forwardingAddress(fromRef)
        if toRef = null
                toRef ⟵ copy(fromRef)
        return toRef

Copy(fromRef):
        toRef ⟵ free
        free ⟵ free+size(fromRef)
        move(fromRef,toRef)
        forwardingAddress(fromRef) ⟵ toRef
```




1. fromRef = A.ptr1
2. toRef = null
3. toRef = 1040
4. free = free + 40 = 1080
5. FA(fromRef)=1040
6. Return 1040

# And now, here's something we hope you'll really like!

# Concurrent copying!

# Concurrent GC

## Recap – definitions

A mutator, just like objects, has a color:

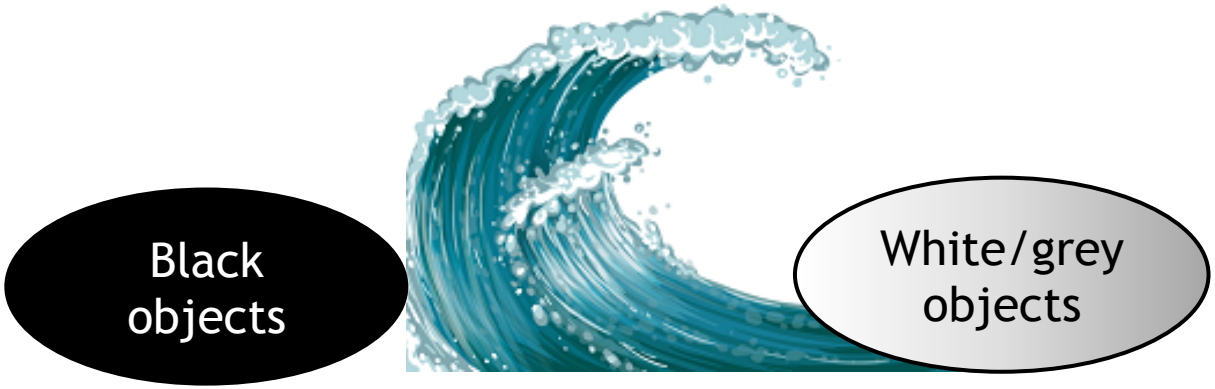
**Grey mutator-** either has not yet been scanned by the collector so its roots are still to be traced or its root has been scanned and need to be rescanned.

**Black mutator-** has been scanned by the collector so its roots have been traced.

We defined the “grey wavefront”- The boundary between black and white objects.

Objects ahead of the wavefront = grey/white objects.

Objects behind of the wavefront = black objects.



# Concurrent copying

ntil now – The collector ran while the mutator was halted.

ow, the collector must, while the mutator is running , be able to:
. Copy objects from From-space to To-space.
. Update pointers to new location of objects.

## nings to notice:

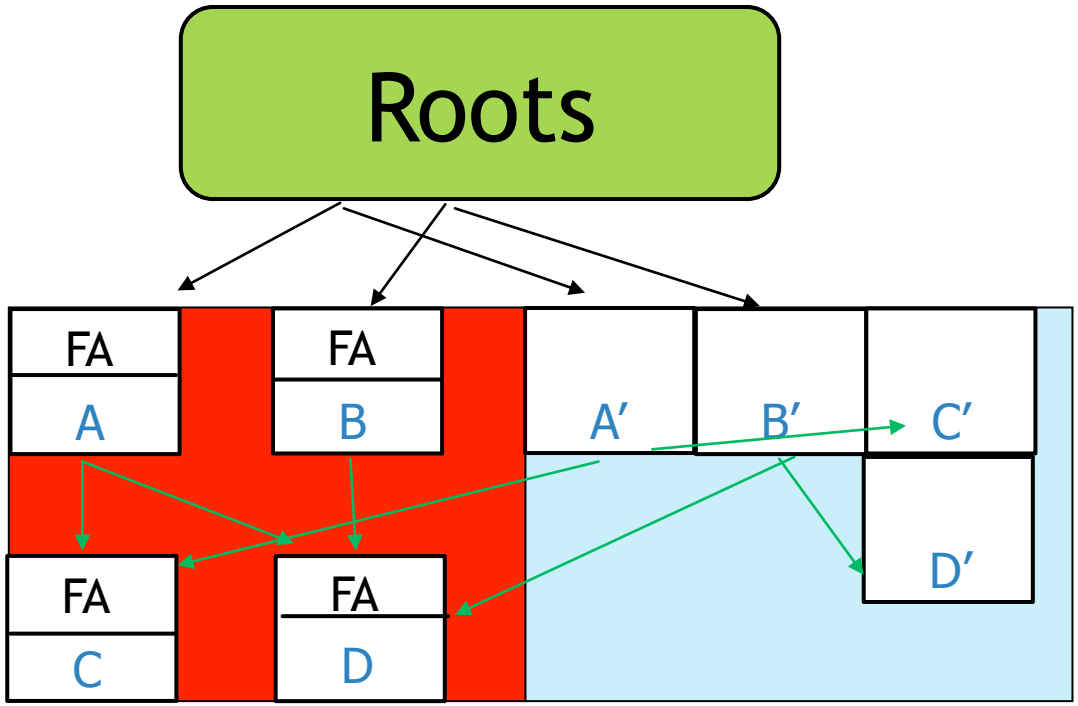
Concurrent copying GC must protect mutator and collector from each other!
Concurrent updates by the mutator must be propagated to the
copies being constructed in To-space by the collector

# Concurrent copying

# Concurrent copying

## What could go wrong?

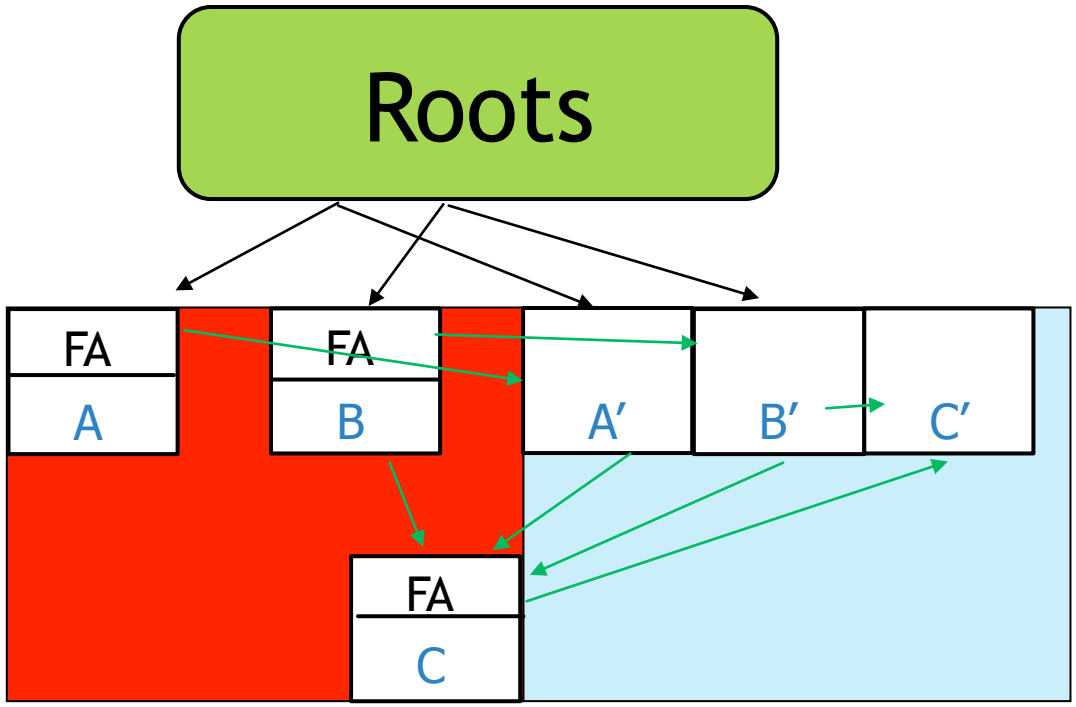
A.ptr1=B.ptr1



# Concurrent copying

## What could go wrong? V2

A.ptr1=B.ptr1



# Concurrent copying

## Invariants

**The black mutator *To-space invariant:***

black mutator must, by definition, hold only To-space pointers.

f it held From-space pointers then the collector would never revisit and orward them, vi-olating correctness.

**The grey mutator *From-space invariant:***

grey mutator must, by definition, hold only From-space pointers at the beginning f the collector cycle.

# Concurrent copying

## Termination

termination of a copying algorithm, all mutator threads must end the collection
e holding only To-space pointers!

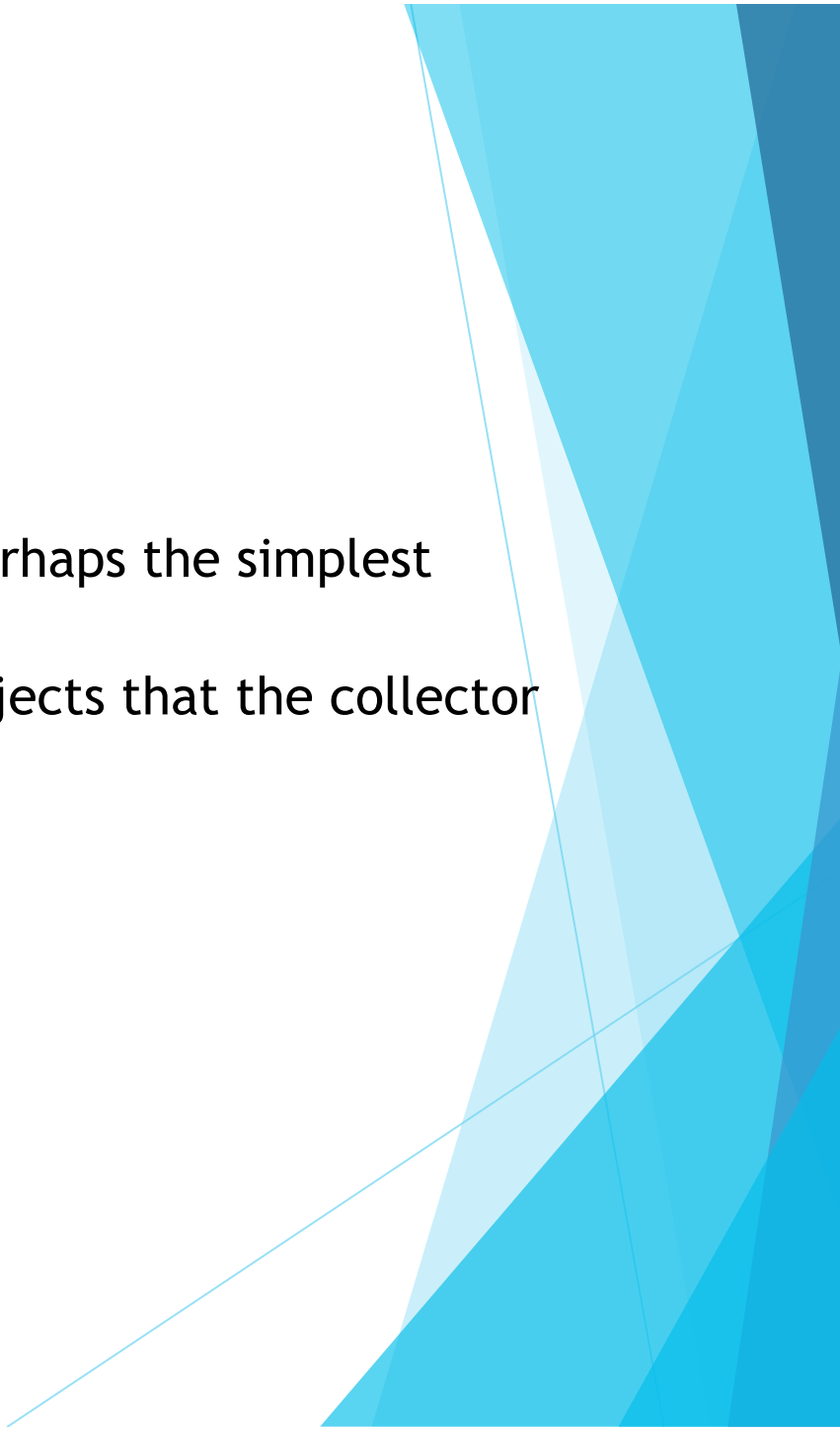
*, any copying collector that allows grey mutator threads to continue operating*
*From-space must eventually switch them all over to To-space by forwarding*
*eir roots.*

# Concurrent copying

## Simple is better

intaining a To-space invariant for all mutator threads is perhaps the simplest
proach to concurrent copying.

cause it guarantees that the mutator threads never see objects that the collector
et to copy, or is in the middle of copying.



# Concurrent copying

## Simple is better– but how?

o, we want to maintain the black mutator invariant.

ut how?

Stop all the mutator threads (atomically).
Forward their roots (copying their targets) at the beginning of the collection cycle.

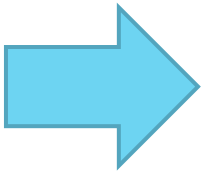
this point, the now-black mutators contain only (grey) To-space pointers.

t wait, there's a catch!
e (unscanned) grey targets will still contain From-space pointers.

# Concurrent copying

## New collect, not completely atomic!

```
c collect():
    flip()
    initialise(worklist)
    for each fld in Roots
        process(fld)
    while not isEmpty(worklist)
        ref ← remove(worklist)
        for each child in Pointers(ref)
            process(child)
```



```
collect():
    atomic
        flip()
        initialise(worklist)
        for each fld in Roots
            process(fld)
    while true
        atomic
            if isEmpty(worklist)
                break
            ref ← remove(worklist)
            for each child in Pointers(ref)
                process(child)
```

# Concurrent copying

## Baker to the rescue!

troducing: Baker's black mutator read barrier (1978)

# Concurrent copying

## Baker's read barrier


Every object read by the mutator must be in the To-space!


When the mutator tries to read through a pointer!

# Concurrent copying

## Baker's read barrier

the read barrier needs to trigger only when loading from a grey To-space object !

```
atomic Read(src, 1):
    ref <— src[i]
    if ref != null && isGrey(src)
        ref <— forward(ref)
    return ref
```

```
forward(fromRef):
        toRef ← forwardingAddress(fromRef)
        if toRef = null
                toRef ← copy(fromRef)
        return toRef

Copy(fromRef):
        toRef ← free
        free ← free+size(fromRef)
        move(fromRef,toRef)
        forwardingAddress(fromRef) ← toRef
        add(worklist,toRef)
        return toRef
```

*Objects allocated in the To-space are considered black!*

# Concurrent copying

## Baker the illusionist

he read barrier has the effect of presenting
he illusion to the mutator threads that the
ollection cycle has completed!

# Concurrent copying

## Baker's read barrier-correctness

Recall we wanted to protect the mutator and the collector from each other.

**Baker's read barrier does just that!**

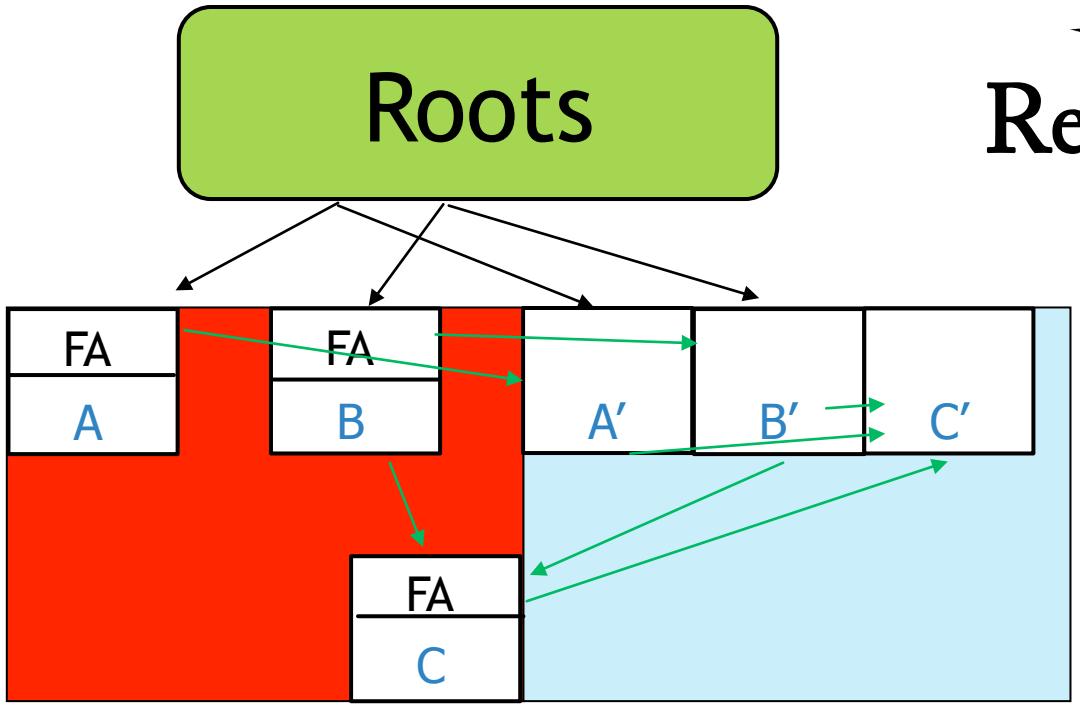
The mutator is protected from the collector!
If the mutator tries to read a white object, it is forwarded and colored grey so no matter what- the mutator always reads the To-space version of the object.

The collector is protected from the mutator!
Since the mutator can't read a pointer to a white object, it can't store such a pointer in a black object and make the collector collect a live object.

# Concurrent copying

## What could go wrong? V2 – fixed!


Read barrier!



# Concurrent copying

## Baker's read barrier- good, could be better

**Baker's barrier is conservative:**

Objects allocated during the collection cycle are considered black
Which means they are not collected, even if they die during the cycle.

Similarly, objects which were already traversed and then died ,while the collection cycle is still running, will not be collected.

# Concurrent copying

## Baker's read barrier- good, could be better

**aker's barrier is expensive in time and space:**

orn (1993) reported that pointer reads form about 15% of the total number of structions.

**lining** the barrier in each such read would create an unacceptably large utator code size.

orn also reported a 20% time overhead when using software read barriers.

# Concurrent copying

## Baker's read barrier- good, could be better

So, according to Zorn, reads are too common in every program, making the read barrier both time consuming and code size enlarging.

## What's the alternative?

# Concurrent copying

Brooks to the rescue!

troducing: Brooks's indirection barrier (1984)

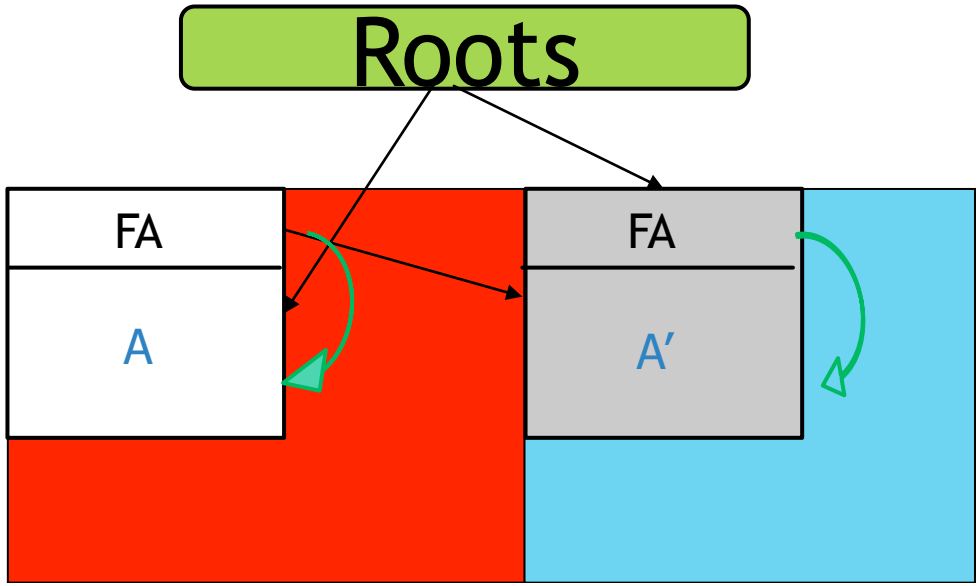
# Concurrent copying

## Brooks's indirection barrier

Brook’s suggested a different approach :
Instead of requiring the To-space invariant allow the mutator to make progress without concern for the wavefront.

Brooks observed that if every object (whether in From-space or To-space) has a non-null forwarding pointer (either to its from-space original or to its copy in To-space) then the test on the src object in the read barrier can be eliminated!

Basically, Brooks suggested:



# Concurrent copying

## Brooks's indirection barrier

```
atomic Read(src, i):
    src <-- forwardingAddress(src)
    return src[i]

atomic Write(src, i, ref):
    src <-- forwardingAddress(src)
    if isBlack(src)
        ref <-- forward(ref)
 src[i] <-- ref
```

Follow forwarding address no matter what!

Dijkstra-style Write barrier to prevent the insertion of From-space pointers behind the wavefront

# Concurrent copying

## Brooks's indirection barrier –is it correct?

```
atomic Read(src, i):
    src <-- forwardingAddress(src)
    return src[i]
```

et's recall that Baker's barrier was introduced in order to prevent reading of rom-space objects.

oes Brooks read barrier takes care of that?
o, it does not!
e read barrier can still read a field ahead of the wave-front that might refer to n uncopied From-space object.

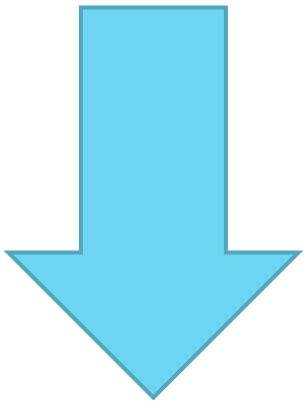
ortunately, Brooks forwarding pointer relaxes the need for the o-space invariant imposed by Baker so the mutator is allowed to operate rey and hold From-space references.

# Concurrent copying

## Brooks's indirection barrier –termination

et’s recall:
rmination requires all mutators to hold only To-space references.

ut Brooks allows mutators to operate grey!



Once copying is finished all mutators need a final scan of their stacks to replace any remaining unforwarded references

# Concurrent copying

## Brooks's indirection barrier –summery

has the advantage of avoiding the need for Baker's To-space invariant which forces e mutator to perform copying work when loading a From-space reference from the ap.

kes sure heap updates are never lost because they occur either to the From-space iginal before it is copied or to the To-space copy afterwards.

## BUT

he Brooks indirection barrier imposes a time and space penalty.

ollowing an indirection pointer adds (bounded) overhead to every mutator heap access- **Vasting time !**

he indirection pointer adds an additional pointer word to the header of every object- **Vasting space !**

# Concurrent copying

## Brooks's indirection barrier – could be better

s we said, Brooks' solution is a bit wasteful both in time and in space.

## Can we do better?

# Concurrent copying

## Nettles to the rescue!

ntroducing: Replication copying collectors

[Nettles *et al*, 1992; Nettles and O'Toole, 1993]

# Concurrent copying

## Replication copying collectors

rook's said: If you have a To-space copy, use it!

ettles says: Let's allow the mutator to use From-space originals, even while the ollector is copying them to the To-space!

## But how?

# Concurrent copying

## Replication copying collectors

General idea:

e mutator threads obey a From-space
variant, updating the From-space objects
rectly, while a write barrier logs all updates
From-space objects to record the
fferences that must still be applied to their
-space copies

# Concurrent copying

## Replication copying collectors

eplication copying collectors allow the state of the To-space copy to lag behind
nat of its From- space original.

ust as long as by the time the collector is fin-ished copying, but before it can
lear the From-space, all mutator updates have been applied from the log to the
o-pace copy and all mutator roots have been forwarded.

**ermination condition:**
he mutation log is empty, the mutator's roots have all been scanned, and all of the
bjects in To-space have been scanned.

# Concurrent copying

## Replication copying collectors- correctness requirements

What are our demands regarding correctness?

ynchronization between the mutator and col-lector via the mutation log, and
when updating the roots from the mutators.

he collector must use the muta-tion log to ensure that all replicas reach a
onsistent state before the collection terminates.

When the collector modifies a black object it must rescan the object to make sure
hat any object referenced as a result of the mutation is also replicated in To-space.

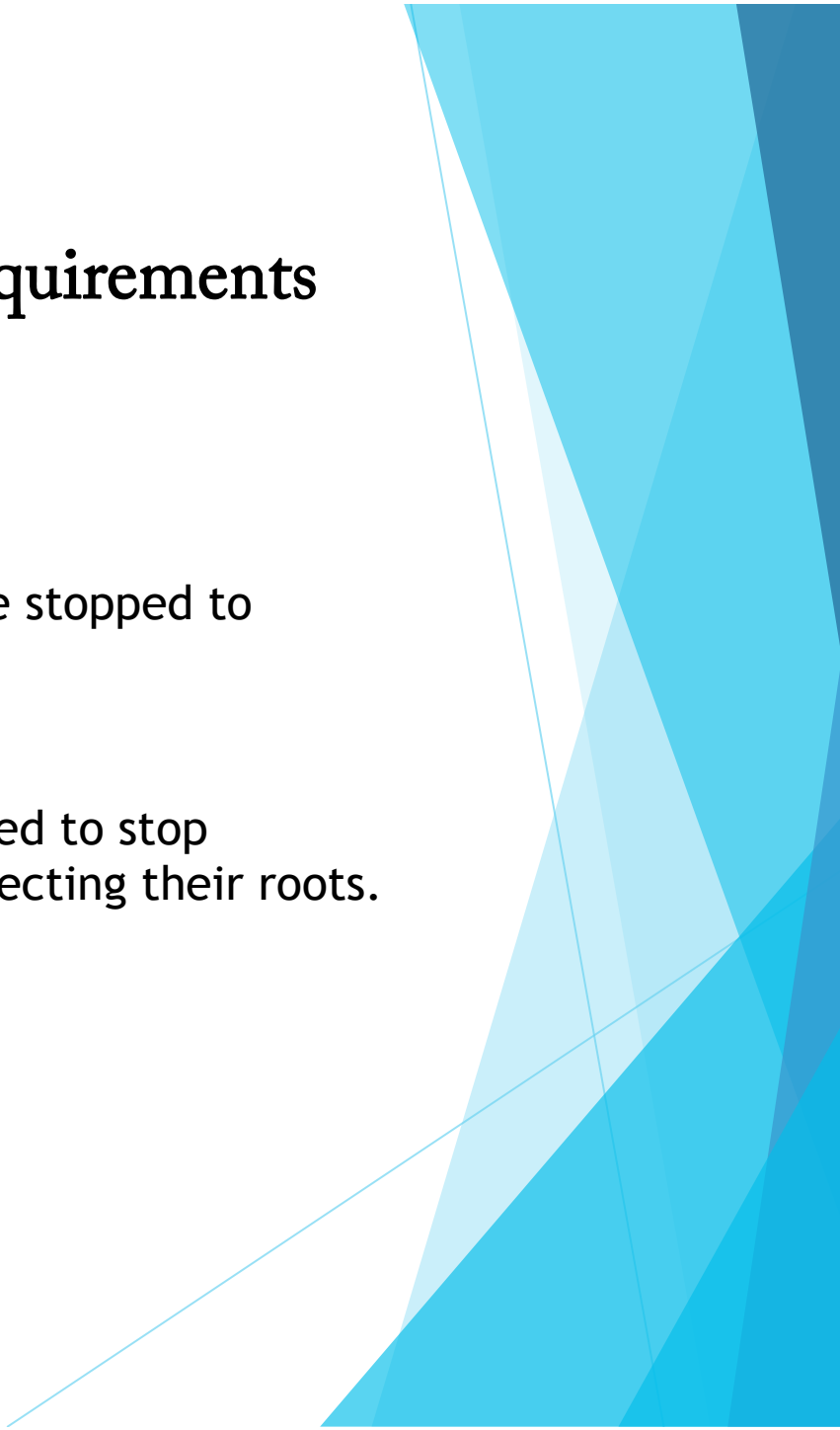
# Concurrent copying

## Replication copying collectors– termination requirements

What are our demands regarding termination?

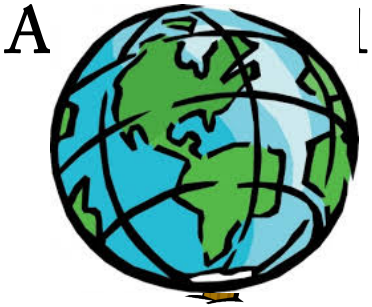
Termination of the collector requires that each mutator thread be stopped to scan its roots.

When the collection cycle is fin-ished, all the mutator threads need to stop together briefly in order to switch them over to To-space by redirecting their roots.



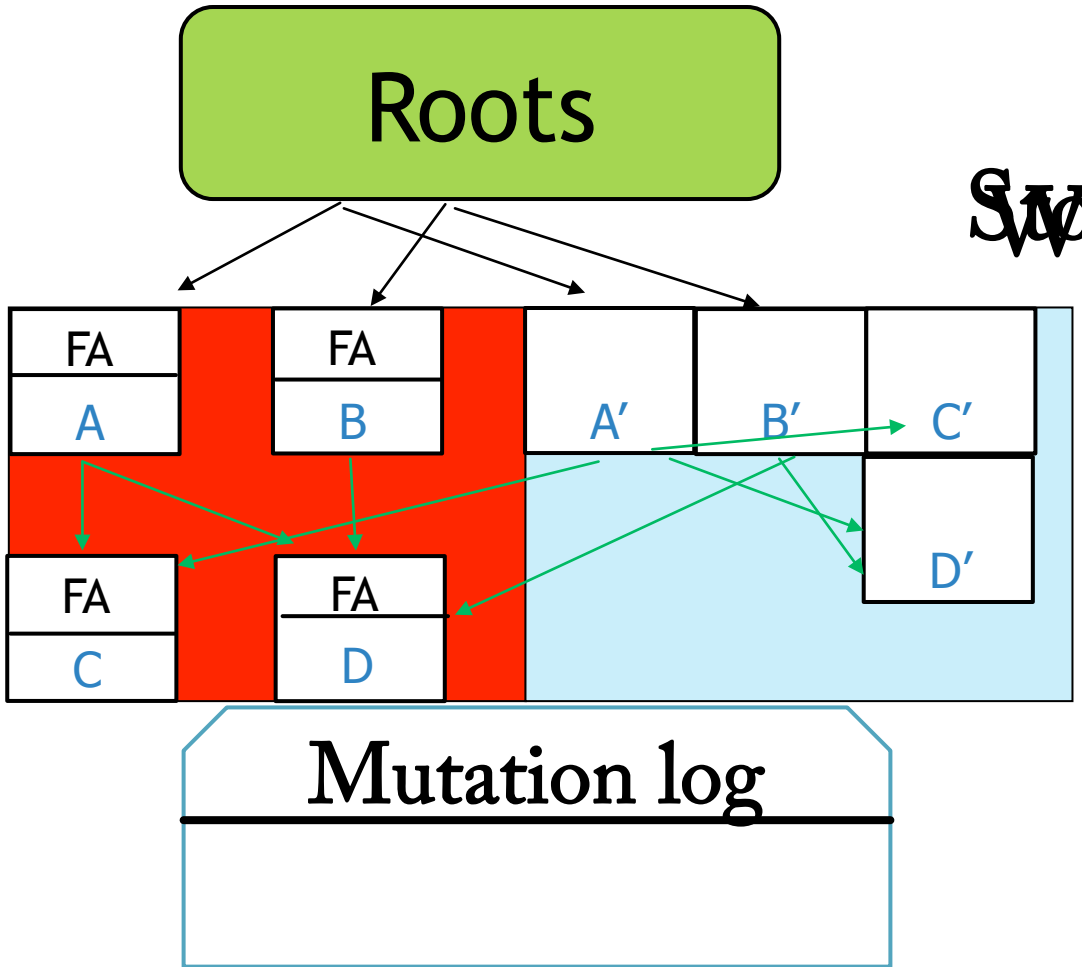
# Concurrent copying

## What could go wrong? – fixed!



Swap the world

Write barrier!



# Concurrent copying

## Replication copying collectors– better, still not flawless

The Good- only short pauses to sample (and at the end redirect) he mutator roots.

The Bad - The mutation log can become a bottleneck: he collector reads the mutation log, which is being written by the client.

The Ugly- ***every*** mutation of the heap, not just point-ers, needs to e logged by the mutator threads- Higher overhead than for traditional ointer-only) write barriers.

# Concurrent copying

## Replication copying collectors– bad is relative

### When is the bad not so bad?

rely functional languages (e.g. Haskell) does not allow mutations of
cords/variables once created.

ettles and O'Toole used ML.
., being a functional language (not pure, allows side-effects),
scourages mutations ,so their performance suffered less.

# Concurrent copying

## Replication copying collectors- not good enough

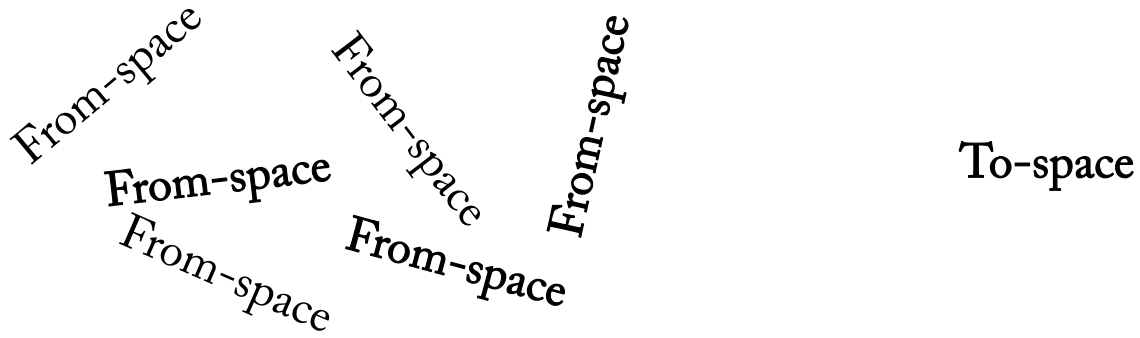
ttles' collector still requires global stop-the-world synchronization of the u-tator threads to transition them to To-space.

e algorithm is not lock-free- When world stop, no mutator can make progress!

Can we do better?

# Concurrent copying

## Herlihy and Moss to the rescue!




troducing: Herlihy and Moss multi-version copying (1992)

# Concurrent copying

Baker 1978- never forget

# But first!

# Concurrent copying

Baker 1978- never forget


Halstead's multiprocessor refinement of Baker's algorithm (1985)

# Concurrent copying

## Halstead's multiprocessor Baker-style algorithm

eap is divided into multiple per-processor regions:
ach processor has its own From-space and To-space

ch processor is responsible for evacuating into its own To-space any From-
ace object it discovers while scanning.

es locking to handle races between processors that compete to copy the
me object, and for updates to avoid writing to an object while it is being
acuated.

es global synchronization to have all the processors perform the flip into
eir To-space before discarding their From-space.

# Concurrent copying

## Herlihy and Moss multi-version copying – the basic idea

ch processor* region composed of **one** To-space and **many (0-n)** From-spaces.

hile copying is performed, multiple From-space ver-sions of an object can cumulate in different spaces.

nly one of these versions is current while the rest are obsolete.

*oday a "processor" is actually a thread, but we'll still refer to it as processor in order to be consistent with Halstead's version.*

# Concurrent copying

## Herlihy and Moss multi-version copying – basic idea

Each processor alternates between 2 tasks:

. A mutator task.

. A scanning task looking for From-space pointers.

When such a pointer is found, the scanner locates the object's current version.

Current version in a From-space -> copied to To-space.
the old version is now obsolete)

# Concurrent copying

## Herlihy and Moss multi-version copying – definitions

# Concurrent copying

## Herlihy and Moss multi-version copying – definitions

***wner-*** A processor which owns From-spaces.

***anner-*** A mutator currently in running a scanning task.

***ean scan-*** for some ***scanner***, a scan which completes without finding any pointers to versions in **any** From-space.

***rty scan-*** opposite of clean.

***ound*** - an interval during which every processor starts and completes a scan.

***ean round-*** one in which every scan is clean and no processor executes a flip.

***andshake bits-*** atomic bits, initially match.

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

The processors need to cooperate!

Each processor is responsible for scanning its own To-space and local variables on its stack for From-space pointers.

Each processor is responsible for copying any From- space object it finds.

- Includ-ing objects in From-spaces of other processors!
- Only if the object does not have a current To-space copy in some processor.

But how can it know?

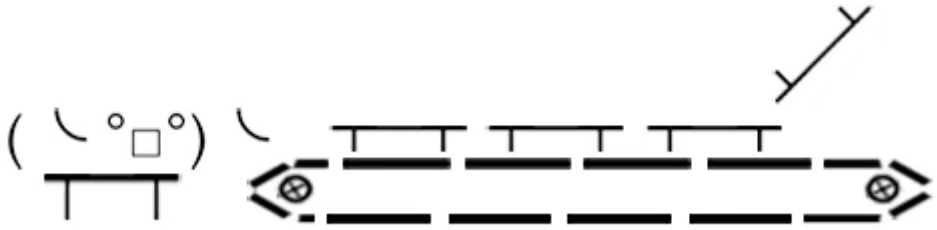
Stay tuned…

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

## One flip is for the weak

- A processor can flip at any time during its mutator task and as many times as it needs **



# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

o, when does a process need to flip?

- When its To-space is full (and not during a scan).
  *Just as long as it has sufficient free space to allocate a new To-space.

After a processor executes a flip, a new From-space is generated.

This From-space can be reclaimed only after completion of a subsequent clean round.

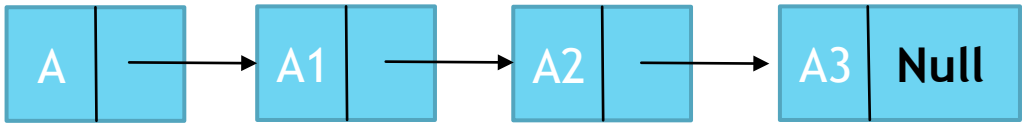
A processor cannot free its From-spaces until it can be sure no other processor holds references to any of its From-space objects.

# Concurrent copying

## Herlihy and Moss multi-version copying – managing versions

eed a way of managing and keeping track of versions

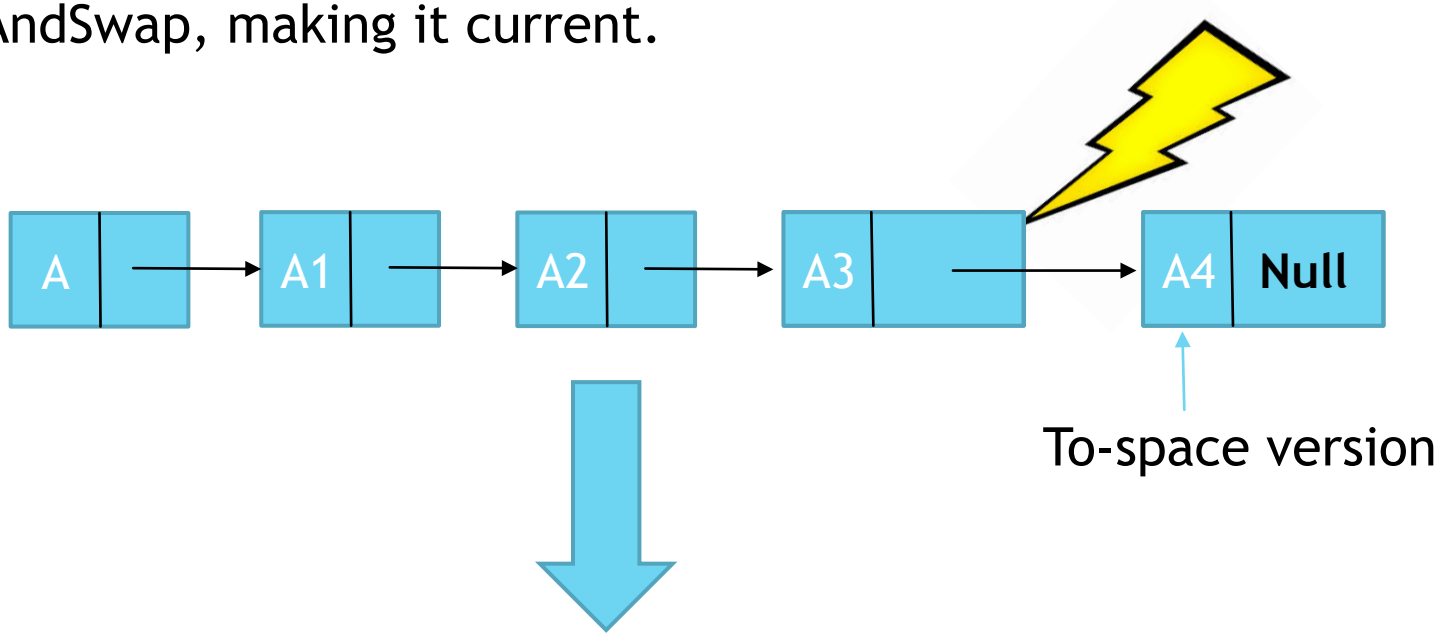
aintain a forwarding pointer field ***next*** at all times in each object, referring
its next (newer version).
r the current version, next=null.



# Concurrent copying

## Herlihy and Moss multi-version copying – managing versions

hen copying a From-space object into its own To-space, a scanning processor
omically installs the To-space copy at the end of the version chain using
ompareAndSwap, making it current.




Every mutator heap access must traverse to the end of the chain of versions before per-forming the access!

# Concurrent copying

## Herlihy and Moss multi-version copying – the algorithm

```
ind-current(x)
        while x.next!=null
                x ← x.next
        return x
```

```
etch (x,i)
        x ← find-current (x)
        return x[i]
```

# Concurrent copying

## Herlihy and Moss multi-version copying –managing versions

eed a way to stay lock-free and make sure heap updates aren't lost.

very store into an object creates a new version of the object in the mutat-ing ocessor's To-space.

Scanning and copying require no global synchronization, while preserving all mutator updates.

# Concurrent copying

## Herlihy and Moss multi-version copying – the algorithm

```
tore(x, i, value)
        /* Allocate space (in my To-space) for a new version of x */
        temp ← allocateSpaceForNewVersion()
        while true
                /* Find current version of x and to new version */
                x ← find-current (x)
                for i in [0,x.size)
                        temp[i] ← x[i]
        /* Set new value to new version */
        temp[i] ← value
        /* Try setting the new version of x */
        if compare&swap (x.next,null,temp)
                return
```

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

We want to make sure an ***owner*** discards its From-spaces only if no other ***scanner*** holds any of its From-space pointers.

An owner detects that another scanner has started and completed a scan using the ***handshake bits***, each bit is written by one processor and read by the other.

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

n order to start a flip the ***owner***:

1. Creates a new To-space.
2. Marks the old To-space as a From-space.
3. Inverts its handshake bit.

t the start of a scan the ***scanner***:

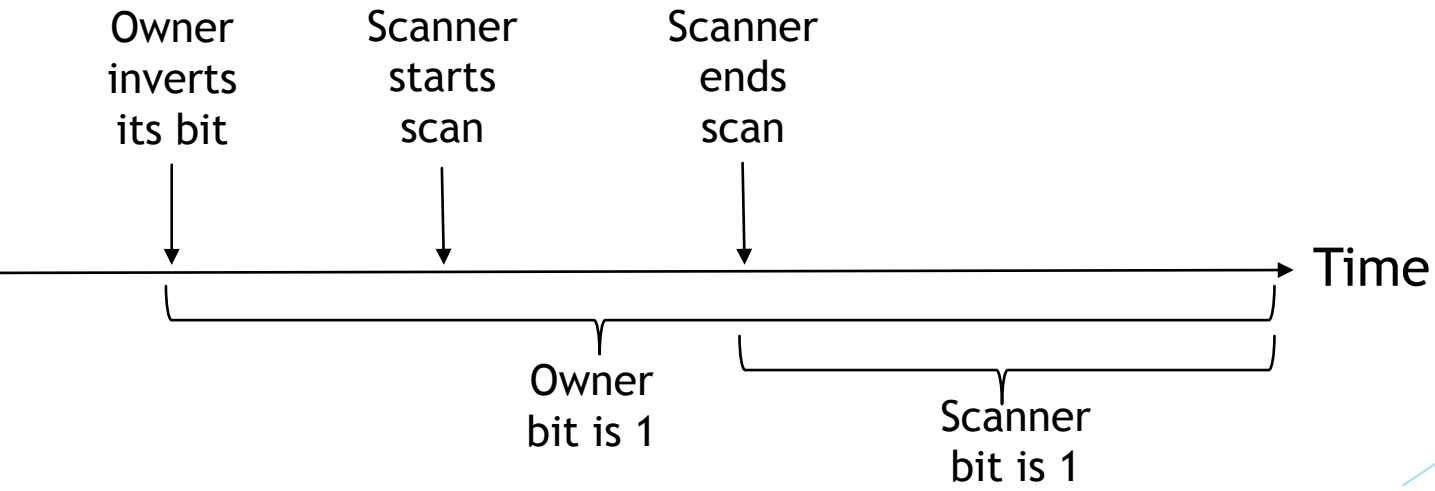
1. Reads the owner's handshake bit.
2. Performs the scan.
3. Sets its handshake bit to the value read from the owner's bit.

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

**bservation:**

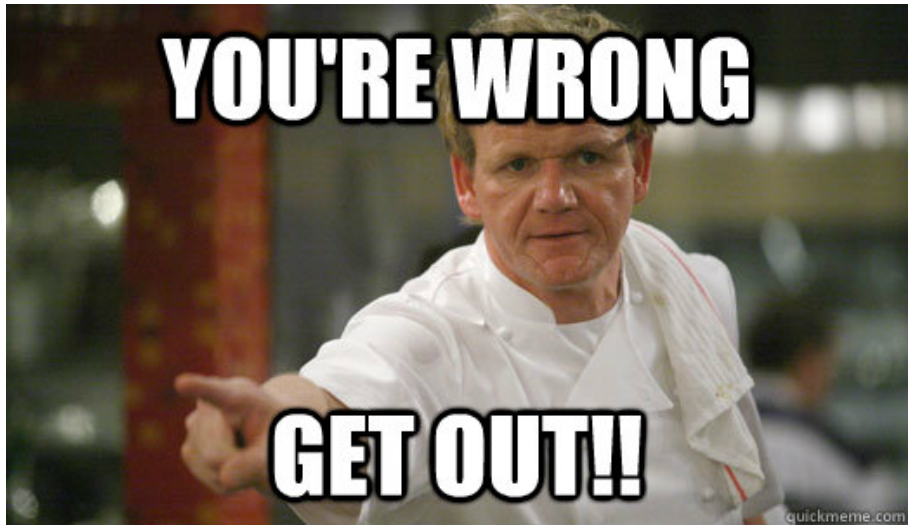
ndshake bits will agree once the scanner has started and
mpleted a scan in the interval since the owner's bit was inverted



# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

o, we covered synchronization between an owners and a scanners




owner must detect that ***all*** processes have started and completed a scan

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

eed to remember: every processor is both an owner and a scanner!

synchronize an ***owner*** with **all** ***scanners*** we do the following:

- The handshake bits are arranged into two arrays:

1. An ***owner*** array containing the ***owners handshake bits***, indexed by ***owner*** processor.
2. A 2-dimensional ***scanner*** array containing the ***scanners*** handshake bits, with an element for each ***owner-scanner*** pair.

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

, how do we use those arrays?

- Because a scan can complete with respect to multiple owners, the scanner must copy the entire ***owner*** array into a local array on each scan.
- At the end of the scan, the scanner must set its corresponding ***scanner*** bits to these previously saved values.

ow an can an owner detect a round is complete?

- A round is complete as soon as its owner bit agrees with the bits from all scanners.

n owner cannot begin a new round until the current round is complete!

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

# Concurrent copying

## Herlihy and Moss multi-version copying – test run

et's do a test run on those arrays:

e'll use 3 processors

**Owner array:**

| 0 | 0 | 0 |
|---|---|---|
| P1 | P2 | p3 |

**Scanner array:**

| | As scanner: P1 | As scanner: P2 | As scanner: p3 |
|---|---|---|---|
| As owner: P1 | 0 | 0 | 0 |
| As owner: P2 | 0 | 0 | 0 |
| As owner: P3 | 0 | 0 | 0 |

# Concurrent copying

## Herlihy and Moss multi-version copying – test run

performs a flip

starts a scan

performs a flip

starts a scan

starts a scan

ends the scan

ends the scan

ends the scan

**Owner array**

| 1 | 0 | 1 |
|---|---|---|
| P1 | P2 | p3 |

**Scanner array**

As scanner

| As owner | P1 | P2 | p3 |
|---|---|---|---|
| P1 | 1 | 1 | 1 |
| P2 | 0 | 0 | 0 |
| P3 | 1 | 0 | 1 |

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

ow can an owner detect a round is clean?

Processes share an array of ***dirty*** bits, indexed by processor

When do processors set values in the dirty array?

1. When an owner executes a flip,
   it sets the ***dirty*** bit for all other processors.


2. When a scanner finds a pointer into another processor's From-space it sets that processor's ***dirty*** bit.

# Concurrent copying

## Herlihy and Moss multi-version copying – the algorithm

```
flip():
        createNewToSpace()
        markOldSpaceAsFromSpace()

        /* Make sure everyone knows this round is dirty */
        for i in [1,n) do
                if i != me
                        dirty[i] ← true

        /* Invert my handshake bit */
        owner[me] ← ! owner[me]
```

```
Scan-start():
        /* Copy owner array to my own local array */
        for i in [1,n) do
                local-owner[i] ← owner[i]
```

# Concurrent copying

## Herlihy and Moss multi-version copying – how does it work

ow to use the dirty array to determine a round is clean?

- If an owner's ***dirty*** bit is clear at the end of a round then the round was clean, and it can clear its From-spaces.
- If not, then it simply clears its ***dirty*** bit.
- Either way, the owner starts a new scan.

# Concurrent copying

## Herlihy and Moss multi-version copying – the algorithm

```
can-end():
        /* Notify other processes I'm done scanning */
        for i in [1,n) do
                scanner[i][me] ← local-owner[i]
        /* Did a round complete? */
        if for every I scanner[me][i] = owner[me]
                /* If round is clean, I can safely clear my From-spaces */
                if !dirty[me]
                        discardFromSpaces()
                /* Clear dirty bit for next scan*/
                dirty[me] ← false
        /* Start another scan*/
        scan-start()
```

# Concurrent copying

## Herlihy and Moss multi-version copying – the algorithm

```
can-value(x):
        /* If found From-space owner, set its owner as dirty*/
        if old(x)
                dirty[owner(x)] ← true
        /* Move object if needed */
        while true
                /* Find current version of x and check if its in a To-space*/
                x ← find-current(x)
                if new(x)
                        return x
                /* Create  a copy of x*/
                temp ← allocateSpaceForNewVersion()
                for I in [1,x.size)
                        temp[j]← x[j]
                /* Try setting the new version of x*/
                if compare&swap (x.next,null,temp)
                        return temp
                else
                        free(temp)
```

# Concurrent copying

## Herlihy and Moss multi-version copying – correctness

Multi-versioning looks good, is it correct?

**Liveness**- Each processor always eventually scans then some processor always eventually re-claims its From-spaces.

**Termination**- At worst, because each processor will eventually run out of free space, further flips will cease, and all processors will eventually focus on scanning until reaching a clean round.

**Performance -** No thank you.

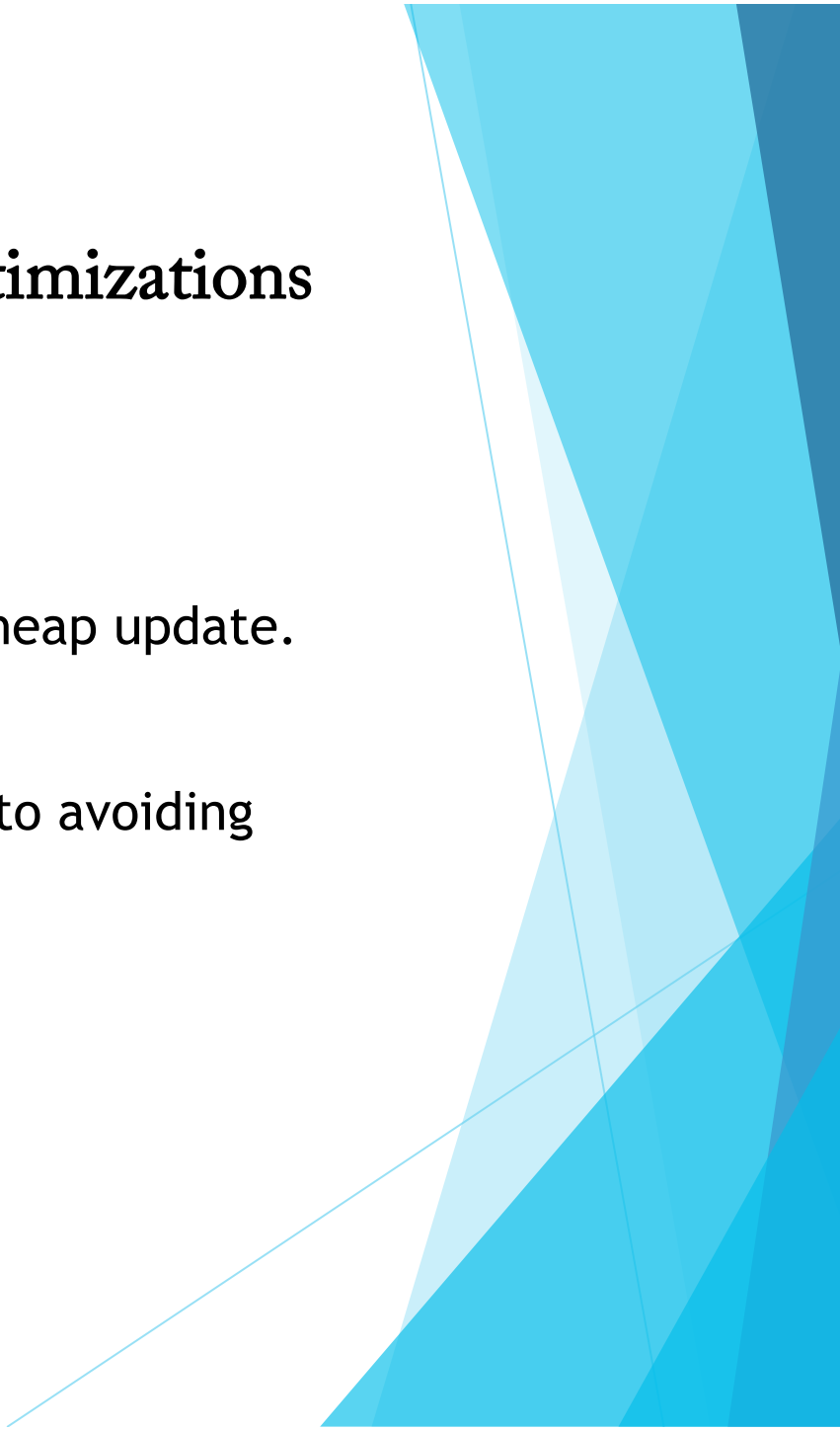
# Concurrent copying

## Herlihy and Moss multi-version copying – optimizations

We just saw an entirely lock-free copying algorithm.

Its downside is the need to create a new version on every heap update.

For you brave souls, Herlihy and Moss offered alternatives to avoiding versioning on every update.



# Concurrent copying

## Herlihy and Moss multi-version copying – still not the best

me to say something bad about the algorithm.

bad scenario (bottleneck) :

One processor happens to hold a large portion of the heap.

Other processors have to wait for it to complete its scan (and end the round) before they can clear their From-spaces

They may end up stalling if they have no free space in which to allocate.

## Can we do better?

# Concurrent copying

## Hudson and Moss to the rescue!

Introducing: Sapphire algorithm

[Hudson and Moss, 2001, 2003]

# Concurrent copying

## Hudson and Moss's sapphire algorithm- basics

Extends previous concurrent copying algorithms.

Has much in common with replication schemes.

t permits only one thread at a time to flip from From-space to To-space
ather than all at once.

Mutators simply update both the From-space and To-space copies of an object
when both exist) to keep them coherent.

Can we do better?

# Concurrent compaction

# Compaction-recap

## Mark and compact (Pavel Brodsky)

wo main phases:

1. Tracing/marking: mark all the live objects.
2. Compacting: relocate live objects and update references to them.

Separation of marking and copying allows us compaction concurrently with mutators!

a way, a compacting algorithm is better than a copying one

ives us control over the order in which objects are relocated.

# Compaction-recap

## Mark and compact

hree ways to rearrange objects in the heap:

. **Arbitrary**: objects are relocated without regard for their original order.
- Fast, but leads to poor spatial locality.

. **Linearising**: objects are relocated close to related objects (siblings, pointer and reference, etc.)

**Sliding**: objects are slid to one end of the heap, “squeezing out” garbage, and maintaining the **original allocation order** in the heap.

Used by most modern **mark-compact** collectors

# Compaction-recap

## Compressor

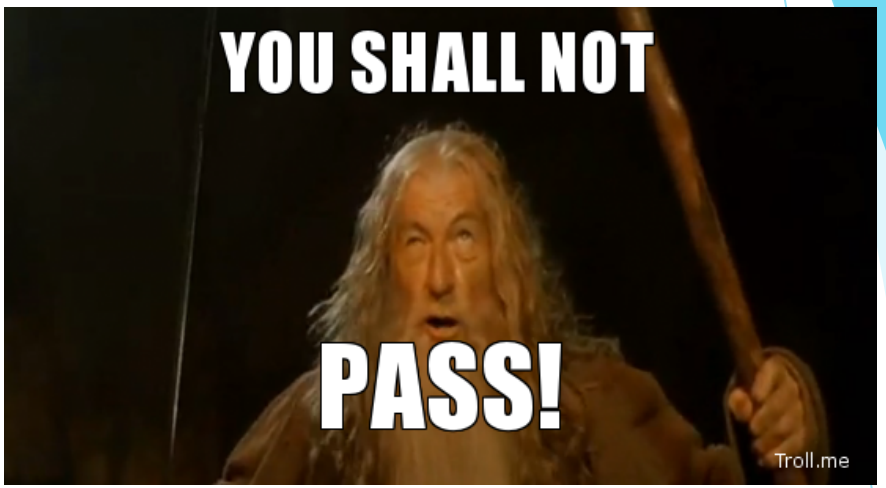
At first, Pavel showed us compacting algorithms which need more than one pass on the full heap in order to relocate objects and update pointers (Lisp 2).


We also saw a **one-pass** sliding algorithm

# Compaction-recap

## One-pass compressor

[Kermany and Petrank, 2006]




Basic idea:

Heap is divided into equally sized blocks.

We maintain a **mark-bitmap** with one bit for each word (could be other size).

Marking phase sets the bits corresponding to the first and last words of each live object.

Additionally, we maintain an **offset-vector** - storing the forwarding address of the first live object in each block.

# Compaction-recap

## One-pass compressor

How does it work:

After marking is done, a routine passes over the **mark-bitmap** and constructs the **offset-vector**.

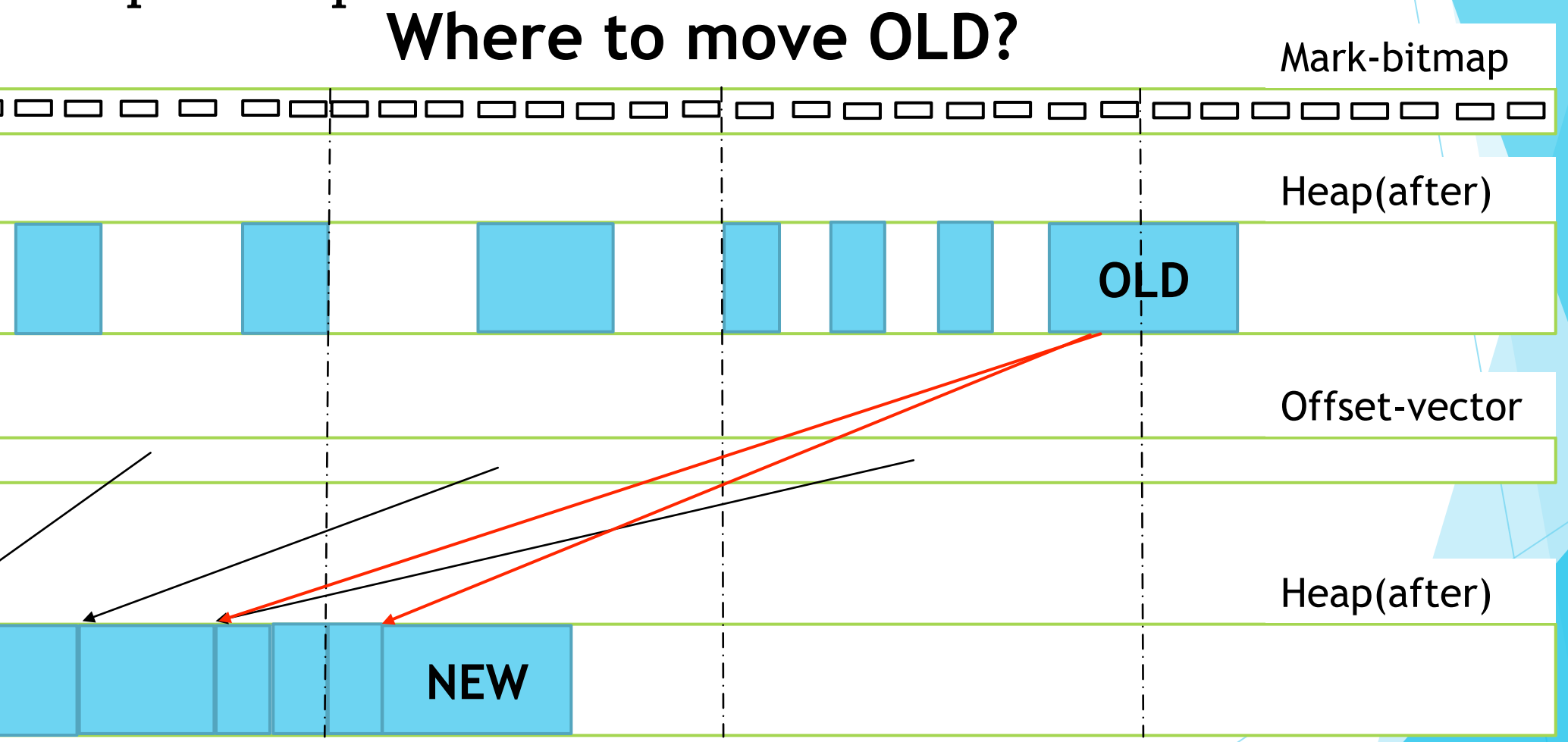
Once the **offset-vector** is initialized, the roots and live references are updated to point to the objects new locations.

Combined use of the **offset-vector** and the **mark-bitmap** gives us the ability to calculate an object's new address easily and rapidly.

Too theoretical, let's see some shiny pictures!

# Compaction-recap

One-pass compressor



# Compaction-recap

## One-pass compressor- algorithm (Originally stolen from Pavel)

```
pact():
 computeLocations(HeapStart, HeapEnd, HeapStart)
 updateReferencesRelocate(HeapStart, HeapEnd)

puteLocations(start, end, toRegion):      // Produce the offset vector from the bitmap
 loc <- toRegion                          // Start at the beginning of the heap
 block <- getBlockNum(start)              // Start from the first block
 for b <- 0 to numBits(start, end) - 1    // Traverse the bitmap
     if b % BITS_IN_BLOCK = 0             // If crossed to new block
         offset[block] <- loc             // Update the offset of this block to be loc
         block <- block + 1               // Advance the block
     if bitmap[b] = MARKED                // Advance loc for every marked bit
         loc <- loc + BYTES_PER_BIT
```

# Compaction-recap

## One-pass compressor- algorithm (Thanks again Pavel)

```
teReferencesRelocate(start, end):
for each fld in Roots                          // Assign new addresses for the Roots
    ref <- *fld
    if ref != null
        *fld <- newAddress(ref)

scan <- start
while scan < end
    scan <- nextMarkedObject(scan)             // Move until the next object with a set bit
    for each fld in Pointers(scan)             // Find new addresses for its every child
        ref <- *fld
        if ref != null
            *fld <- newAddress(ref)

    dest <- newAddress(scan)                   // Find a new address for the object itself
    move(scan, dest)                           // Make the actual move
```

# Concurrent compaction

## One-pass compressor- concurrently

ut how does the Compressor handles concurrent compaction?

Virtual memory page protec-tion primitives

# Concurrent compaction

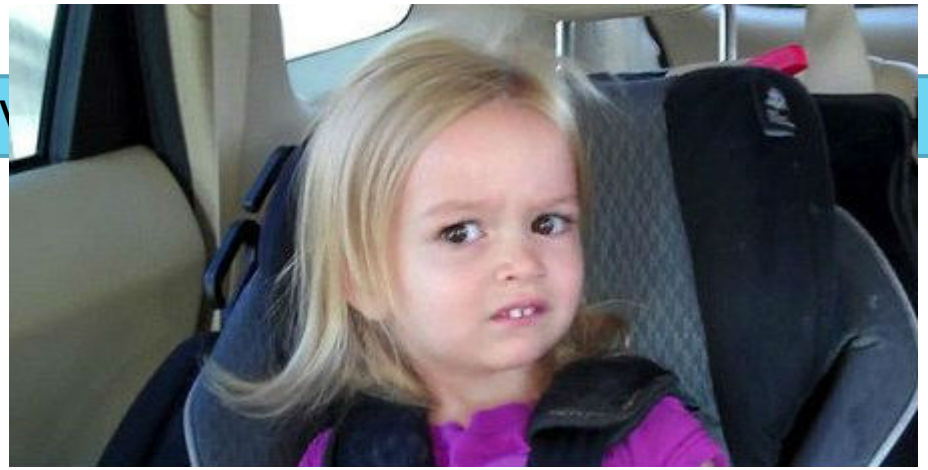
## One-pass compressor- concurrently

ompressor uses page protection primitives:

**PROT1:** decrease the accessibility of a page.
**UNPROT**: increase the accessibility of a page.

ut most importantly:

**MAP2**:
map the same physical page at two different virtual addresses, at different levels of protection.



# Concurrent compaction

## One-pass compressor- concurrently

o, how does it work?

ompressor protects the To-space pages from read/write access
without yet mapping them to physical pages).

ffset-vector construction and To-space protection happens while
nutators do their thing in the From-space.

ventually, need to stop the world to switch mutators roots to To-space.

# Compaction-recap

## One-pass compressor- problem

et's recall the Compressor changes pointers before the actual copying.

Mutators can access uncopied To-space addresses!

# Concurrent compaction

## One-pass compressor- problem and solution

olution: Trap any mutator trying to
ccess a protected To-space copy!

# Concurrent compaction

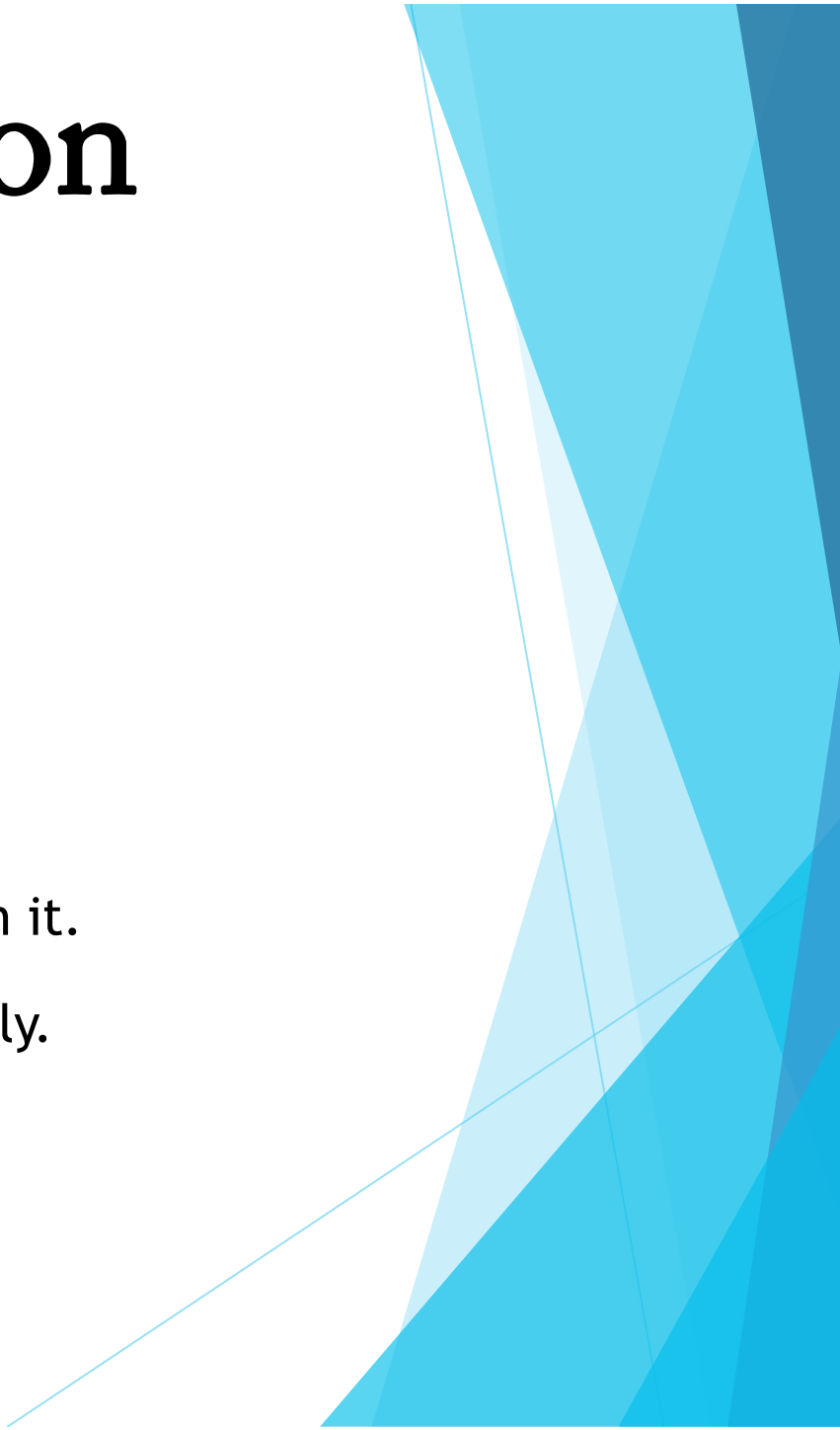
## One-pass compressor- problem and solution

What do we want from our trap?

1. Map and populate the page with its copies.
2. Forward the references in those copies.

Two things to notice:

1. An object must start inside the page to be considered in it.
2. The trap de facto makes the mutator a compactor briefly.



# Concurrent compaction

## One-pass compressor- problem and solution

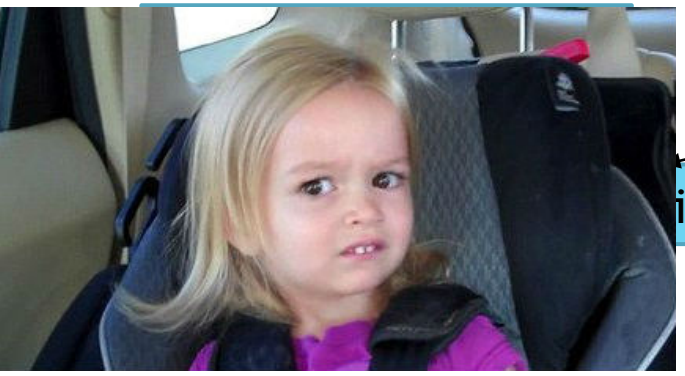
o we prevented mutator access to uncopied objects by trapping them.

ut wait, won't the compactor thread also be trapped?

emember, we have the primitive "Map2"

ysical page is mapped twice:

- A protected To-space virtual page (for mutators access).
- An protected virtual page (private for compactor access). Discarded when work on page done.



Virtual 2 (unpr

ical

# Concurrent compaction

## One-pass compressor- colors

olors are assigned to pages now.
Need to define what each page color means:

. White page- From-space page.

. Grey page- protected To-space page.

. Black page- unprotected To-space page.

When does a grey page turns black?

Once the compaction work for that page has been done.

# Concurrent compaction

## One-pass compressor- more problems

roblem 1:
Vhile the offset-vector is being computed, mutators can allocate objects
n “holes” in the To-space.

olution:
efore the offset-vector is being computed, mutators allocates object only in
ne To-space.

# Concurrent compaction

## One-pass compressor- more problems

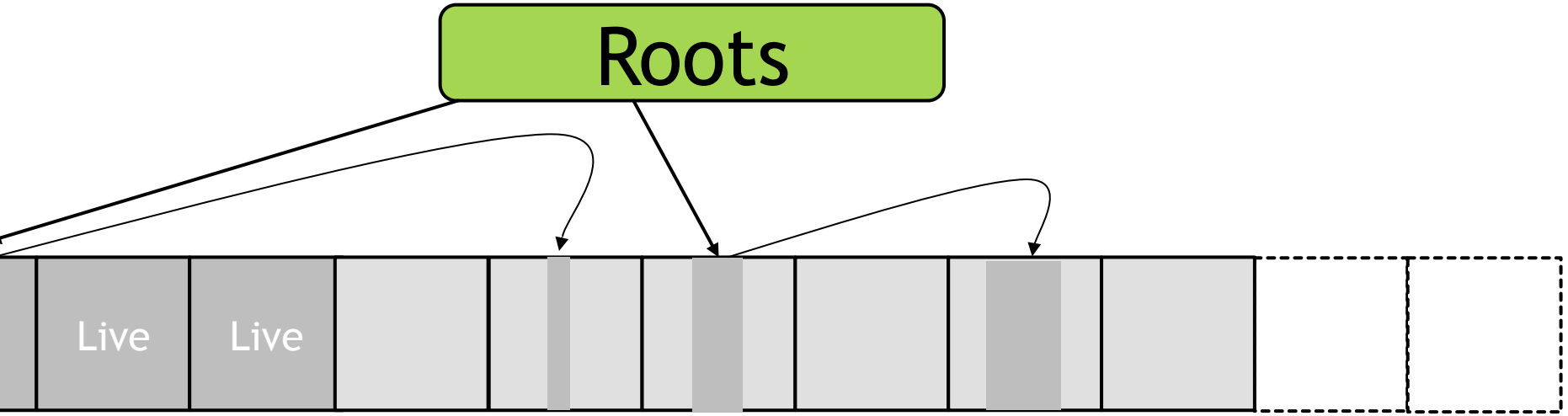
roblem 2:
Newly allocated To-space objects can point to stale From-space objects.
Same problem for roots!

olution:
can pointer fields after mutators flip to To-space and redirect pointers if
eeded.

*equires adding another trap to new pages and to the roots which will*
*erform the pointers scanning.*

# Concurrent compaction

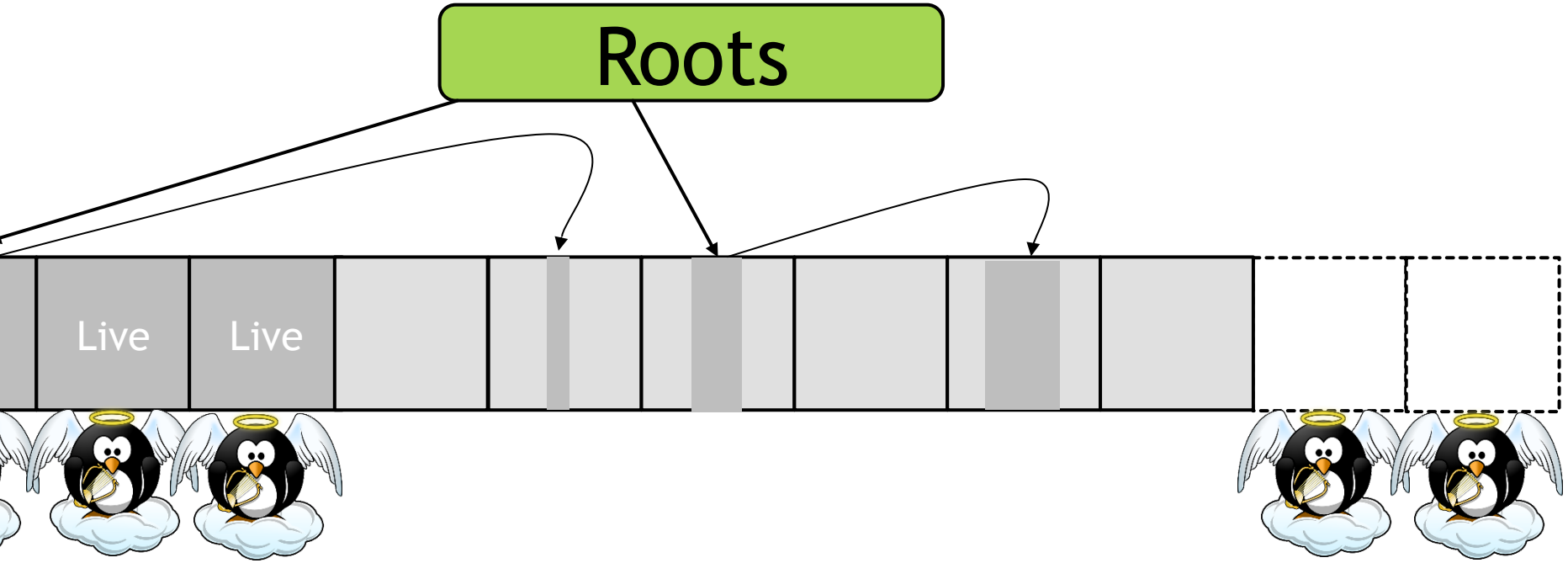
## One-pass compressor- illustrated




Compute forwarding information

# Concurrent compaction

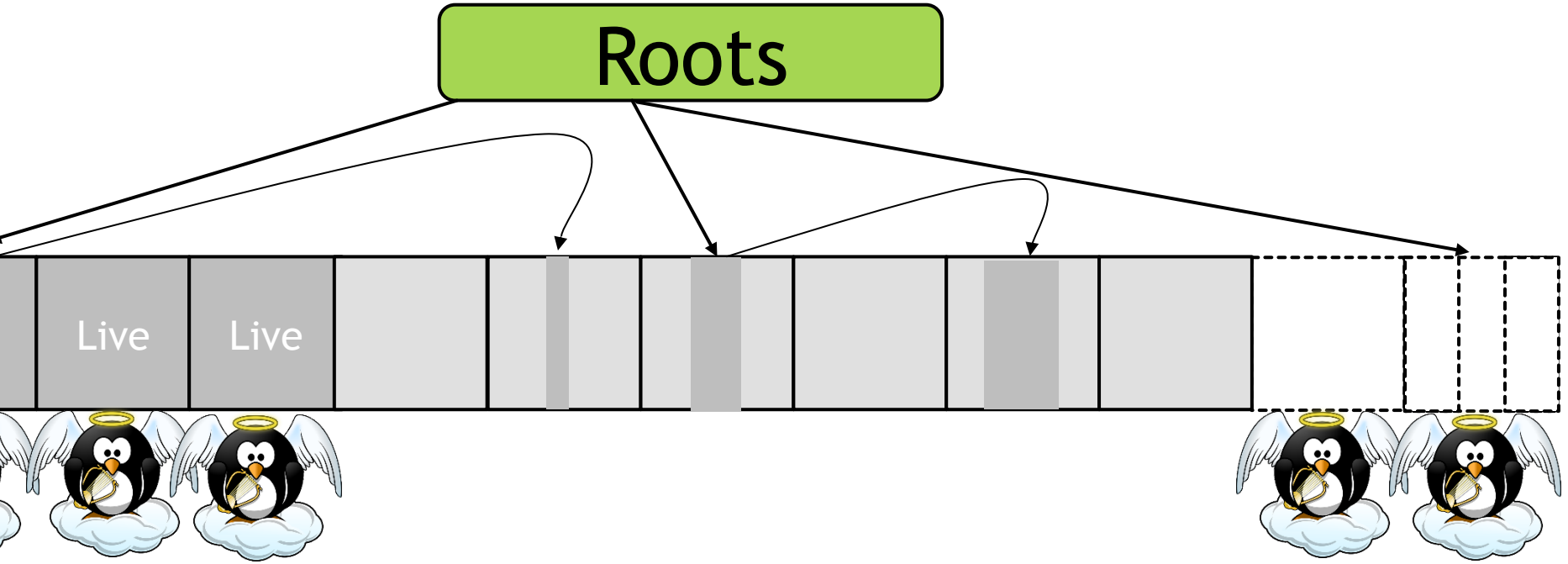
## One-pass compressor- illustrated




Protect To-space pages

# Concurrent compaction

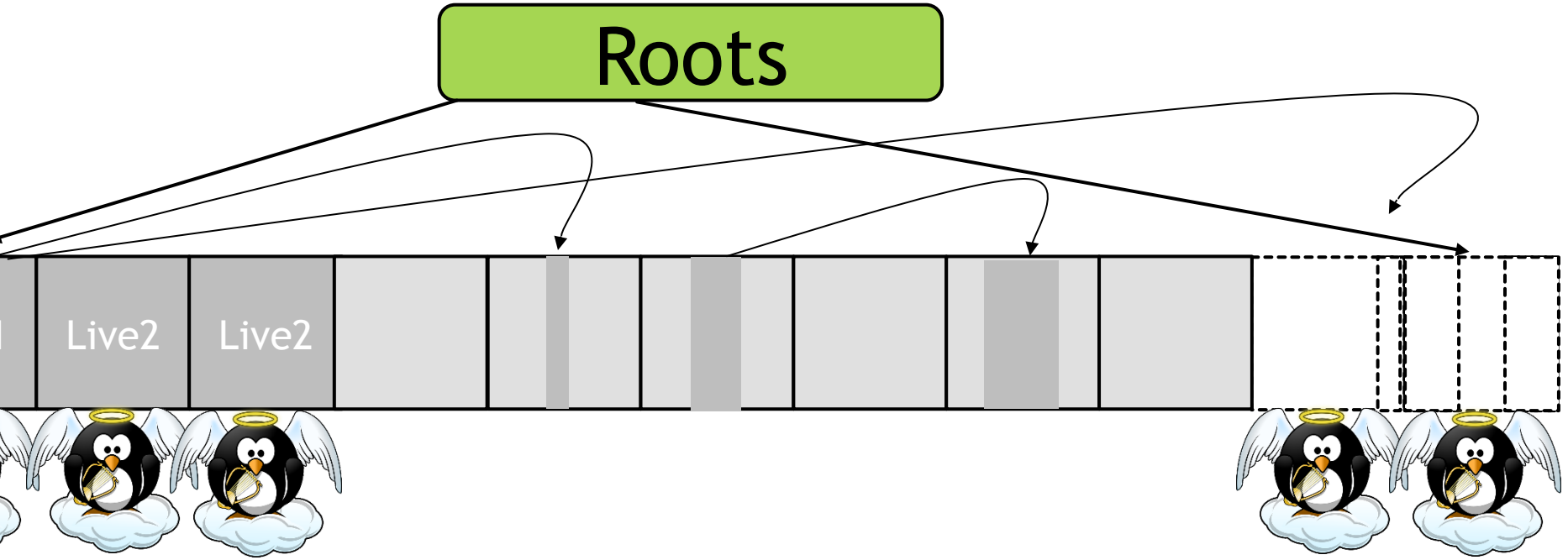
## One-pass compressor- illustrated




Flip mutator roots

# Concurrent compaction

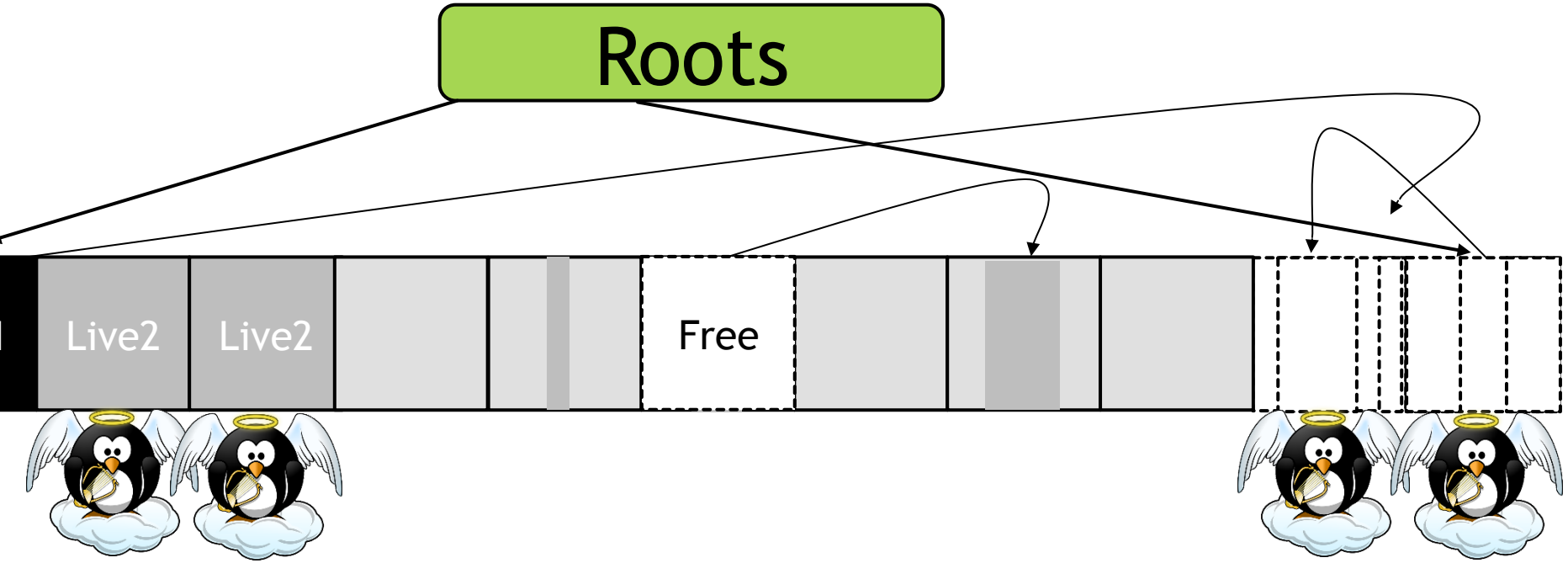
## One-pass compressor- illustrated




A mutator is trying to access Live1
Trap it!

# Concurrent compaction

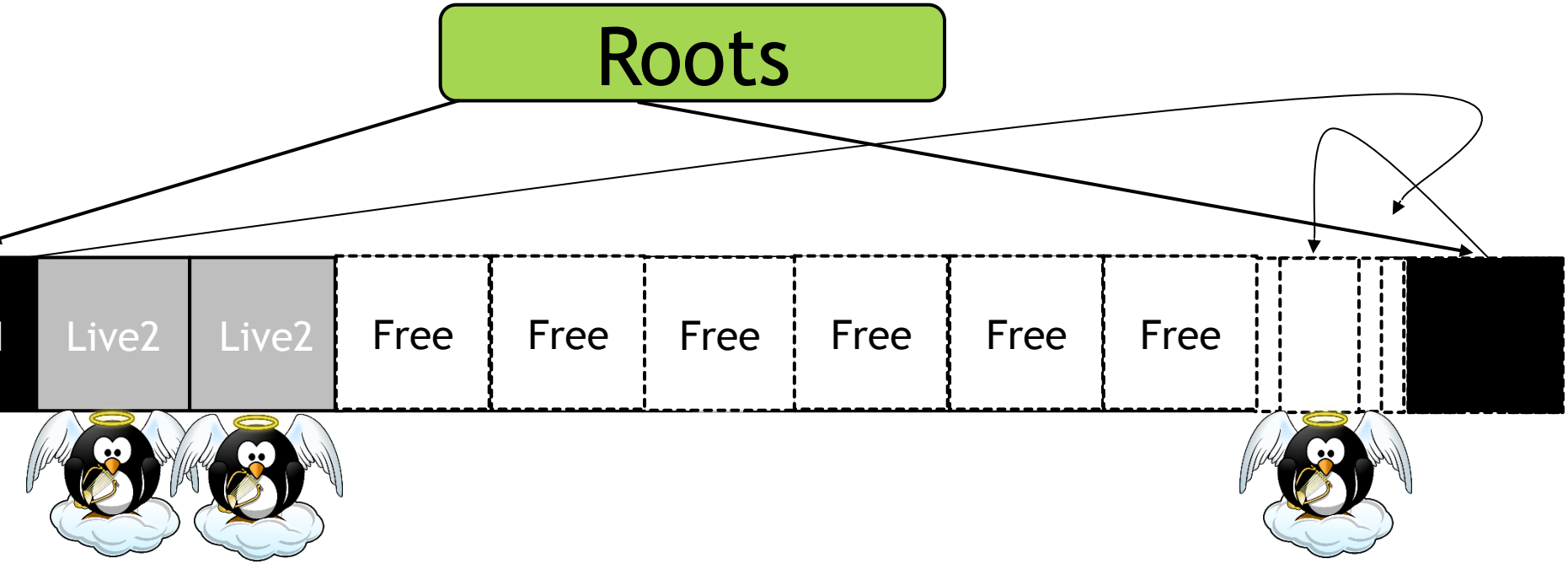
## One-pass compressor- illustrated




A mutator is trying to access rightmost To-space page
Trap it!

# Concurrent compaction

## One-pass compressor- illustrated



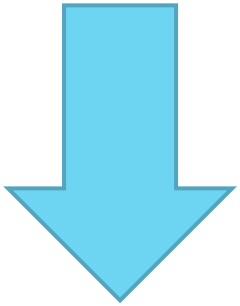

Mutator is done interfering, finish compaction

# Concurrent compaction

## One-pass compressor- not perfect

The Compressor we just saw has a downside.

Trapping a mutator means copying all of the accessed page objects and forward all the pointers in those objects to refer to To-space locations.



Significant pauses on the mutator!

# Concurrent compaction

## Click and Azul to the rescue!

Introducing:  The Pauseless collector

[Click *et al*, 2005; Azul, 2008]

# Concurrent compaction

## Pauseless collector- basic idea

The Compressor we just saw protected To-space pages.

Pauseless algorithm basic idea:

- Protect From-space pages that contain objects being moved.

  * *A much smaller set of pages to protect.*

- Pages can be protected incrementally.

- Use a read barrier whenever a mutator tries to read a stale From-space reference.

# Concurrent compaction

## Pauseless collector- basic idea

Pauseless can be implemented in 2 ways:

. Use special hardware to implement the read barrier as an instruction.
**Original implementation!*

. On stock hardware, lnline barrier logic wherever a reference is being loaded.

# Concurrent compaction

## Pauseless collector- implementation

Pauseless can be implemented in 2 ways:

. Use special hardware to implement the read barrier as an instruction.
**Original implementation!*

. On stock hardware, lnline barrier logic wherever a reference is being loaded.

We'll focus on the hardware implementation

# Concurrent compaction

## Pauseless collector- hardware support

Azul's hardware differs from a stock hardware:

. Supports several fast user-mode trap handlers (for barriers).

. Adds another privilege level: GC-mode.
**Several of the user-mode traps switch to GC-mode.*

. Supports large (up to 2MB) pages.
These pages are the standard pages for the Pauseless collector.

# Concurrent compaction

## Pauseless collector– don't stop me now

Until now we used a Stop-the-world technique, Pauseless uses a **checkpoint:**

When a mutator reaches the checkpoint, it can be asked to do some GC work before it could proceed.

The collector helps blocked threads by performing the GC work for them.

s opposed to Stop-the-world,
**heckpoint** never stops a
unning thread!

# Concurrent compaction

## Pauseless collector- NMT

Need a way to track scanned references.

Pauseless steals one bit from every address.

This bit is called **Not-Marked-Through** (NMT) .

The hardware maintains a desired-value for the bit.

Any use of a reference which has an un-desired NMT value will set off a trap.

# Concurrent compaction

## Pauseless collector- phases

So, how does it work?

The Pauseless collector is divided into 3 phases:

All phases are fully parallel and concurrent!

. **Mark phase-** responsible for periodically refreshing the mark bits.

. **Relocate phase-** uses the most recently available mark bits to
ind pages with little amount of live data, relocate their objects
nd free their physical memory.

. **Remap phase-** updates every pointer in the heap whose target
as been relocated.

# Concurrent compaction

## Pauseless collector- got to love it

Time to see how each phase works

**BUT FIRST**

et's see what's so great about the Pauseless collector!

# Concurrent compaction

## Pauseless collector- got to love it

- Pauseless collector is more “relaxed”:

We’re never in a hurry to finish any given phase due to the fact that no phase bothers the mutators too much.

There’s no race to finish some phase before starting another collection:
The relocate phase runs non-stop and can immediately free memory at any point.

**What if we have to keep up with a substantially large amount of mutators?**

Since all phases are parallel, the collector can keep up simply by adding more collector threads.

# Concurrent compaction

## Pauseless collector- got to love it

he phases incorporate a “self-healing” effect:


If a mutator is trapped in a read barrier,
replaces the old reference with updated one, thus preventing triggering the trap again.

# Concurrent compaction

## Pauseless collector- mark phase

- Pauseless first phase is **mark**:

ach object has two mark bits, one for the current cycle and one for the previous cycle.

each scan, mark is responsible for:

. At the start, clearing the current cycle's mark bits.

. Periodically refreshing the mark bits.

. Set the NMT bit for all references it **encounters** to the desired value.

Gather liveness information for each page.

After scan is done, flip the per-thread desired NMT value.

# Concurrent compaction

## Pauseless collector- mark phase, NMT

's recall we have a NMT bit:

a mutator tries to load a reference which has the wrong NMT value:

A trap will set off:

- Communicate the reference to the marker threads.
- Store the corrected (marked) reference back to memory, so the trap won't set off again.

utators does not have to wait for the marker threads to flip the NMT bit for objects.
Instead, each mutator flips each reference it encounters as it runs.

# Concurrent compaction

## Pauseless collector- mark phase

's recall the Pauseless collector uses a checkpoint!

can give the mutator some of the collector's dirty work:

When a running thread reaches the checkpoint, it marks it's own root set.

What about blocked threads?

Collectors in the mark phase take care of it!

Each mutator can proceed safely once its roots has been marked.

**BUT**

The mark phase cannot proceed until all threads have passed the checkpoint.

# Concurrent compaction

## Pauseless collector- relocate phase

relocate phase is responsible for (in that order):

Finding sparsely occupied pages (good candidates for compaction).

Building  side arrays to hold forwarding addresses for the objects to be relocated.

*Why not just keep the FA in the From-space originals like we're used to?*

Because physical memory is reclaimed right after copying is done and before the pointers are forwarded.

Protecting the candidates pages from access by the mutators.

If a mutator tries accessing a protected page, it will set off a trap!

# Concurrent compaction

## Pauseless collector- remap phase

- remap phase is responsible for:

Forwarding the re-maining stale references in the live pages.

Reclaiming virtual memory.

Since there are no more stale references to the From-space pages, their virtual memory can now be recycled.

# Summery

ll algorithms had a common goal:
ollect garbage while protecting the mutator and the collector from each other.

ome algorithms accomplish this with the To-space invariant → Baker's algorithm

thers let the mutator use the To-space copy, if one exists, or otherwise use From-pace objects → Brook's indirection barrier

thers let the mutator operate in the From-space, as long as all the changes it makes ventually propagate to the To-space → Nettles and O'Toole mutator log

# Summery

lgorithms which wanted to prevent stopping the world to redirect root pointers
nded up creating multiple versions for each object → Herlihy and Moss multi-version collector

/e also saw compaction can be done in similar ways, but without the need to copy all
bjects at every collection → Compressor and Pauseless

# Discussion

There one common bad thing all copying algorithms we just saw share in common

**They copy!**

Any ideas?

# Discussion

Compressor heavily depends on virtual memory and primitives (map2) which can be costly

Any ideas?

# FIN

# Thank you!