

ART Memory Management

Roman Artemev, Syntacore

April 3, 2025

About me

- ▶ JVM for E2K (Elbrus) Arch
- ▶ Kotlin Compiler
- ▶ Android Runtime for RISC-V arch

Plan

1. ART Overview
2. General GC information
3. Mark & Sweep
4. Semi-Space
5. Concurrent Copying
6. Concurrent Mark Compact

Plan

1. **ART Overview**
2. General GC information
3. Mark & Sweep
4. Semi-Space
5. Concurrent Copying
6. Concurrent Mark Compact

ART Overview

- ▶ Android RunTime

ART Overview

- ▶ Android RunTime
- ▶ Executes DEX bytecode (classes.dex)

DEX bytecode

DEX bytecode

```
public int calculate(int a, int b, int c) {  
    return (a + b) * c;  
}
```

DEX bytecode

```
public int calculate(int a, int b, int c) {  
    return (a + b) * c;  
}
```

JVM

```
0: iload_1 // Load a on stack  
1: iload_2 // Load b on stack  
2: iadd    // push a + b  
3: iload_3 // Load c on stack  
4: imul    // push TOP * c  
5: ireturn // return
```

DEX bytecode

```
public int calculate(int a, int b, int c) {  
    return (a + b) * c;  
}
```

JVM

```
0: iload_1 // Load a on stack  
1: iload_2 // Load b on stack  
2: iadd    // push a + b  
3: iload_3 // Load c on stack  
4: imul    // push TOP * c  
5: ireturn // return
```

DEX

```
move v0, p1 // Move a -> v0  
move v1, p2 // Move b -> v1  
move v2, p3 // Move c -> v2  
add-int v0, v0, v1 // v0 + v1 => v0  
mul-int v0, v0, v2 // v0 * v2 => v0  
return v0 // return
```

ART Overview

- ▶ Android RunTime
- ▶ Executes DEX bytecode (classes.dex)
- ▶ Primary VM on Android devices

ART Overview

- ▶ Android RunTime
- ▶ Executes DEX bytecode (classes.dex)
- ▶ Primary VM on Android devices
- ▶ Runs Android applications (.apk/apex)

ART Overview

- ▶ Android RunTime
- ▶ Executes DEX bytecode (classes.dex)
- ▶ Primary VM on Android devices
- ▶ Runs Android applications (.apk/apex)
- ▶ Supports AOT compilation

ART Overview

- ▶ Android RunTime
- ▶ Executes DEX bytecode (classes.dex)
- ▶ Primary VM on Android devices
- ▶ Runs Android applications (.apk/apex)
- ▶ Supports AOT compilation
- ▶ Targets arm/aarch64/x86/x86_64/riscv64

Plan

1. ART Overview
2. **General GC information**
3. Mark & Sweep
4. Semi-Space
5. Concurrent Copying
6. Concurrent Mark Compact

Garbage Collection

- ▶ 4GB Max Heap size

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000-0xFFFFFFFF)

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000-0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000–0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000–0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000–0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:
 1. Foreground

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000-0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:
 1. Foreground
 - ▶ Runs when Application is in active

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000–0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:
 1. Foreground
 - ▶ Runs when Application is in active
 - ▶ Focuses on latency

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000-0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:
 1. Foreground
 - ▶ Runs when Application is in active
 - ▶ Focuses on latency
 2. Background

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000-0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:
 1. Foreground
 - ▶ Runs when Application is in active
 - ▶ Focuses on latency
 2. Background
 - ▶ Run when Application is in background

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000–0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:
 1. Foreground
 - ▶ Runs when Application is in active
 - ▶ Focuses on latency
 2. Background
 - ▶ Run when Application is in background
 - ▶ Focuses on efficiency

Garbage Collection

- ▶ 4GB Max Heap size
 1. Mapped to the low addresses (0x00000000-0xFFFFFFFF)
 2. Uses compressed 32-bit pointer even on 64-bit system
- ▶ Uses so-called *Space* abstraction to represent heap memory
- ▶ 2 GC working states:
 1. Foreground
 - ▶ Runs when Application is in active
 - ▶ Focuses on latency
 2. Background
 - ▶ Run when Application is in background
 - ▶ Focuses on efficiency

Plan

1. ART Overview
2. General GC information
3. **Mark & Sweep**
4. Semi-Space
5. Concurrent Copying
6. Concurrent Mark Compact

Mark & Sweep

Overview

Mark & Sweep

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/mark_sweep.h

Mark & Sweep

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/mark_sweep.h
- ▶ Uses Malloc Space

Mark & Sweep

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/mark_sweep.h
- ▶ Uses Malloc Space
 - ▶ Non-moving collector (preserves object references)

Mark & Sweep

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/mark_sweep.h
- ▶ Uses Malloc Space
 - ▶ Non-moving collector (preserves object references)
- ▶ 2 Versions:

Mark & Sweep

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/mark_sweep.h
- ▶ Uses Malloc Space
 - ▶ Non-moving collector (preserves object references)
- ▶ 2 Versions:
 1. Serial (**-Xgc:MS**), JVM SerialGC, ParallelGC

Mark & Sweep

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/mark_sweep.h
- ▶ Uses Malloc Space
 - ▶ Non-moving collector (preserves object references)
- ▶ 2 Versions:
 1. Serial (**-Xgc:MS**), JVM SerialGC, ParallelGC
 2. Concurrent Marking (**-Xgc:CMS**), JVM Concurrent Mark&Sweep

Mark & Sweep

Algorithm

Mark & Sweep

Algorithm

1. Collect root set

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on
2. Mark reachable objects

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on
2. Mark reachable objects
 - 2.1 DFS on memory graph (opt. Concurrently)

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on
2. Mark reachable objects
 - 2.1 DFS on memory graph (opt. Concurrently)
 - 2.2 Re-mark roots & dirty objects (STW)

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on
2. Mark reachable objects
 - 2.1 DFS on memory graph (opt. Concurrently)
 - 2.2 Re-mark roots & dirty objects (STW)
 - ▶ Scan dirty cards

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on
2. Mark reachable objects
 - 2.1 DFS on memory graph (opt. Concurrently)
 - 2.2 Re-mark roots & dirty objects (STW)
 - ▶ Scan dirty cards
3. Sweep garbage i.e. unreachable objects calling `Free(unreachable_object)`

Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on
2. Mark reachable objects
 - 2.1 DFS on memory graph (opt. Concurrently)
 - 2.2 Re-mark roots & dirty objects (STW)
 - ▶ Scan dirty cards
3. Sweep garbage i.e. unreachable objects calling `Free(unreachable_object)`
 - ▶ `is_gargabe = live[obj] & ~mark[obj]`

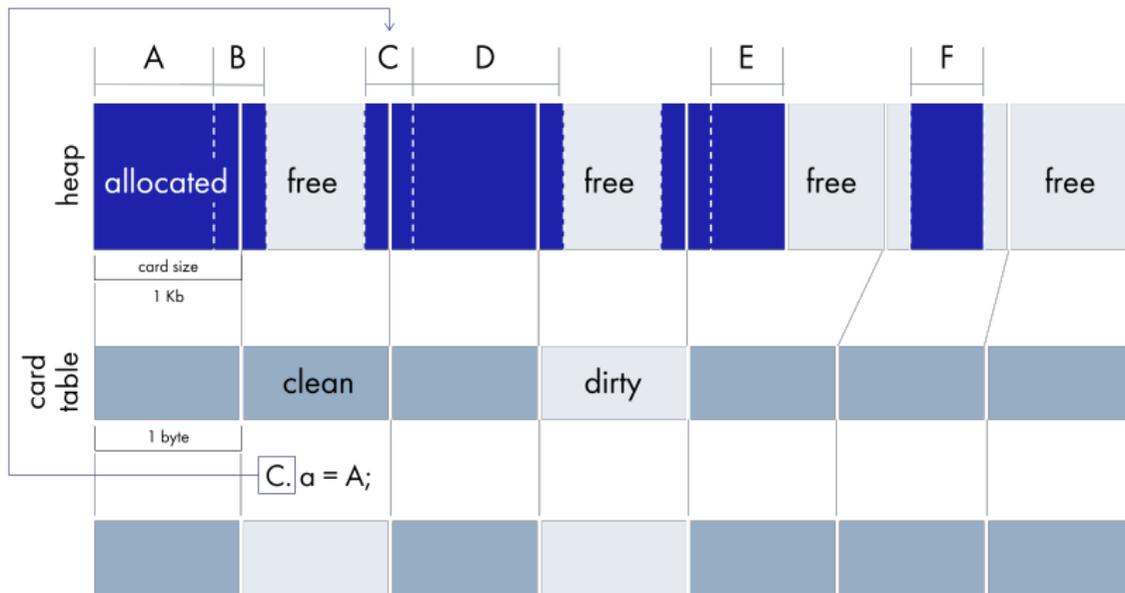
Mark & Sweep

Algorithm

1. Collect root set
 - 1.1 Thread stacks & registers
 - 1.2 Static objects
 - 1.3 many other minor places like native handlers, image/jit references and so on
2. Mark reachable objects
 - 2.1 DFS on memory graph (opt. Concurrently)
 - 2.2 Re-mark roots & dirty objects (STW)
 - ▶ **Scan dirty cards**
3. Sweep garbage i.e. unreachable objects calling `Free(unreachable_object)`
 - ▶ `is_gargabe = live[obj] & ~mark[obj]`

Mark & Sweep

Write Barrier



Mark & Sweep

Problems

Mark & Sweep

Problems

- ▶ Slow allocations

Mark & Sweep

Problems

- ▶ Slow allocations
 - ▶ Non-constant complexity

Mark & Sweep

Problems

- ▶ Slow allocations
 - ▶ Non-constant complexity
 - ▶ Thread unsafe

Mark & Sweep

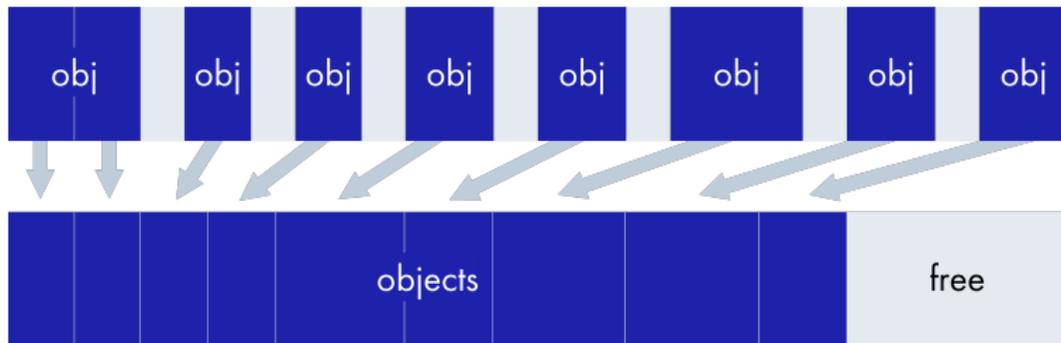
Problems

- ▶ Slow allocations
 - ▶ Non-constant complexity
 - ▶ Thread unsafe
- ▶ Memory fragmentation

Mark & Sweep

Problems

- ▶ Slow allocations
 - ▶ Non-constant complexity
 - ▶ Thread unsafe
- ▶ Memory fragmentation



Plan

1. ART Overview
2. General GC information
3. Mark & Sweep
4. **Semi-Space**
5. Concurrent Copying
6. Concurrent Mark Compact

Semi-Space

Overview

Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/semi_space.h

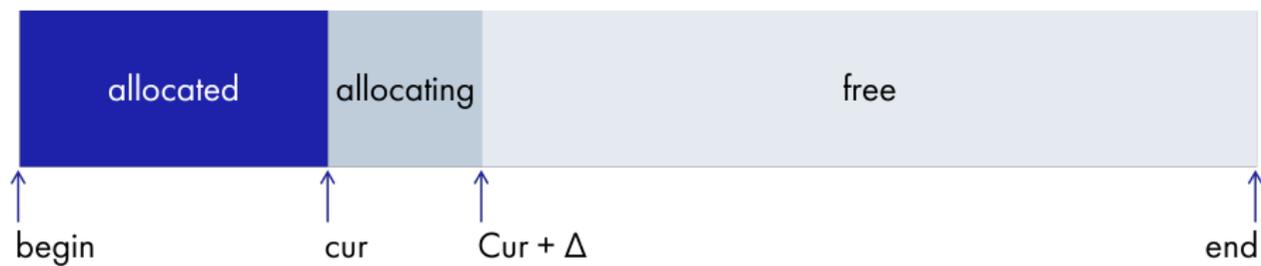
Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*

Semi-Space

Bump Pointer Space



Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*

Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*
 - ▶ from and to spaces

Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*
 - ▶ from and to spaces
 - ▶ Moving collector

Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*
 - ▶ from and to spaces
 - ▶ Moving collector
- ▶ **-Xgc:SS**

Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*
 - ▶ from and to spaces
 - ▶ Moving collector
- ▶ **-Xgc:SS**
- ▶ Uses *mark-is-relocate* concept

Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*
 - ▶ from and to spaces
 - ▶ Moving collector
- ▶ **-Xgc:SS**
- ▶ Uses *mark-is-relocate* concept
- ▶ Always STW

Semi-Space

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/semi_space.h
- ▶ Uses 2 *Bump Pointer Spaces*
 - ▶ from and to spaces
 - ▶ Moving collector
- ▶ **-Xgc:SS**
- ▶ **Uses *mark-is-relocate* concept**
- ▶ Always STW

Semi-Space

Algorithm

Semi-Space

Algorithm

1. Collect root set

Semi-Space

Algorithm

1. Collect root set
 - 1.1 Stop threads

Semi-Space

Algorithm

1. Collect root set
 - 1.1 Stop threads
2. Relocate any reached object in *from-space* into *to-space*

Semi-Space

Algorithm

1. Collect root set
 - 1.1 Stop threads
2. Relocate any reached object in *from-space* into *to-space*
 - 2.1 Update reference

Semi-Space

Algorithm

1. Collect root set
 - 1.1 Stop threads
2. Relocate any reached object in *from-space* into *to-space*
 - 2.1 Update reference
3. Exchange *from* and *to* spaces

Semi-Space

Algorithm

1. Collect root set
 - 1.1 Stop threads
2. Relocate any reached object in *from-space* into *to-space*
 - 2.1 Update reference
3. Exchange *from* and *to* spaces
 - 3.1 Any object in *from-space* is garbage

Semi-Space

Algorithm

1. Collect root set
 - 1.1 Stop threads
2. Relocate any reached object in *from-space* into *to-space*
 - 2.1 Update reference
3. Exchange *from* and *to* spaces
 - 3.1 Any object in *from-space* is garbage
 - 3.2 All object in *to-space* is alive

Semi-Space

Algorithm

1. Collect root set
 - 1.1 Stop threads
2. Relocate any reached object in *from-space* into *to-space*
 - 2.1 Update reference
3. Exchange *from* and *to* spaces
 - 3.1 Any object in *from-space* is garbage
 - 3.2 All object in *to-space* is alive
 - 3.3 *to-space* becomes Allocation space

Semi-Space

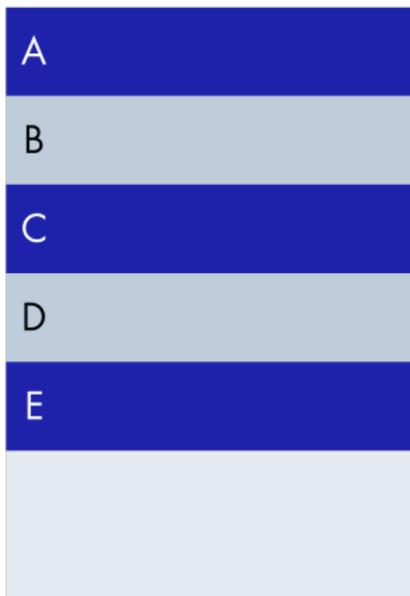
Algorithm

1. Collect root set
 - 1.1 Stop threads
2. Relocate any reached object in *from-space* into *to-space*
 - 2.1 Update reference
3. Exchange *from* and *to* spaces
 - 3.1 Any object in *from-space* is garbage
 - 3.2 All object in *to-space* is alive
 - 3.3 *to-space* becomes Allocation space
4. Resume threads

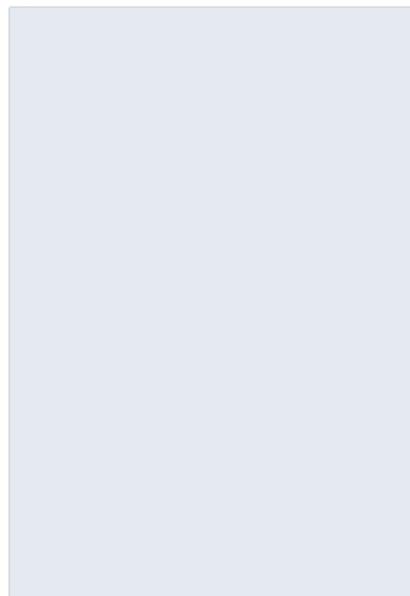
Semi-Space

Scheme

Space 1 (F)

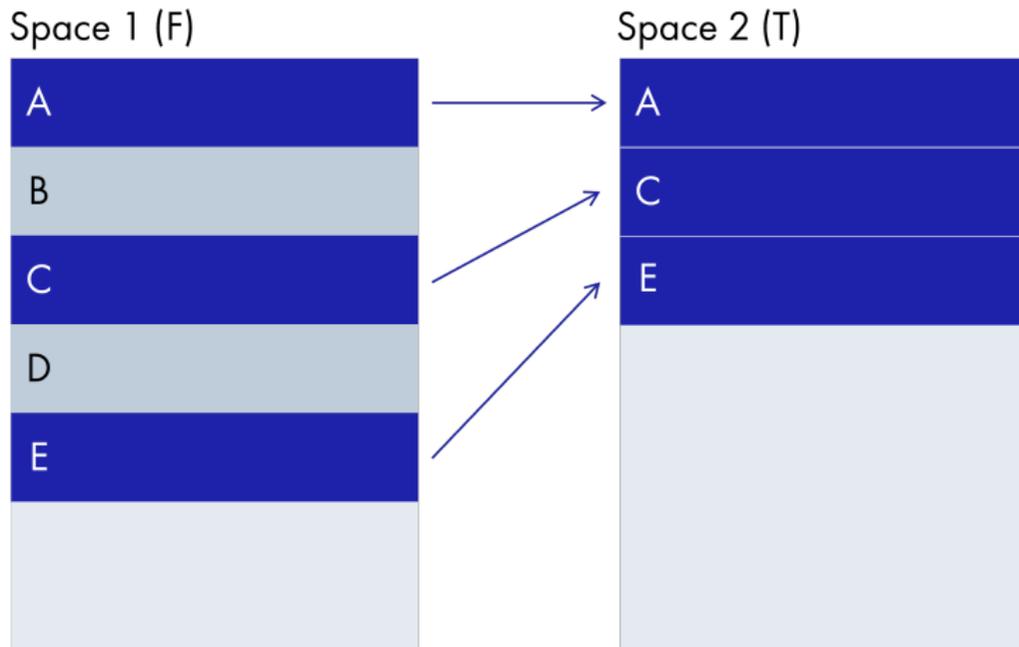


Space 2 (T)



Semi-Space

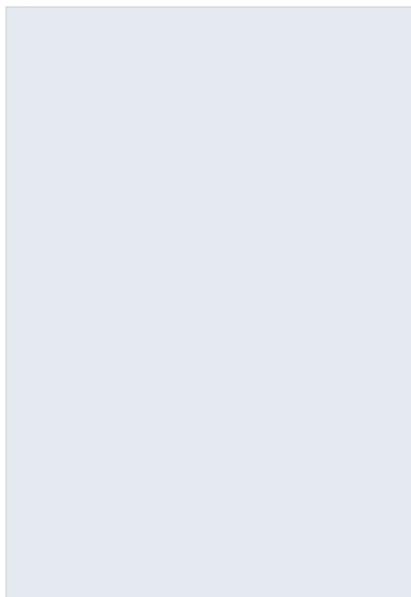
Scheme



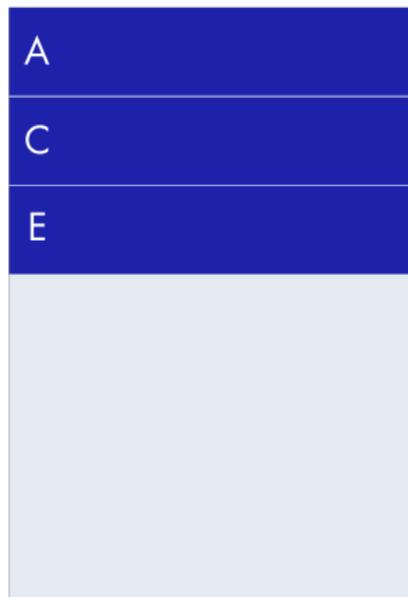
Semi-Space

Scheme

Space 1 (F)



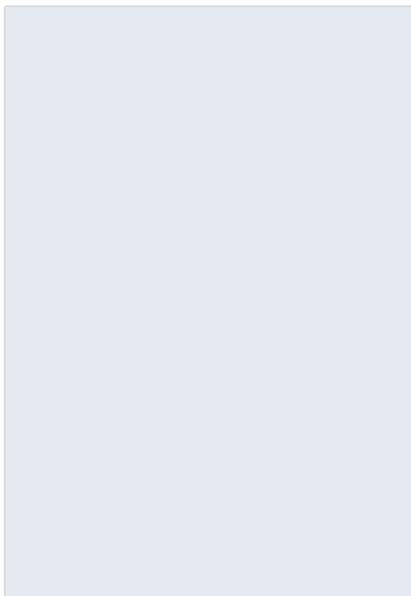
Space 2 (T)



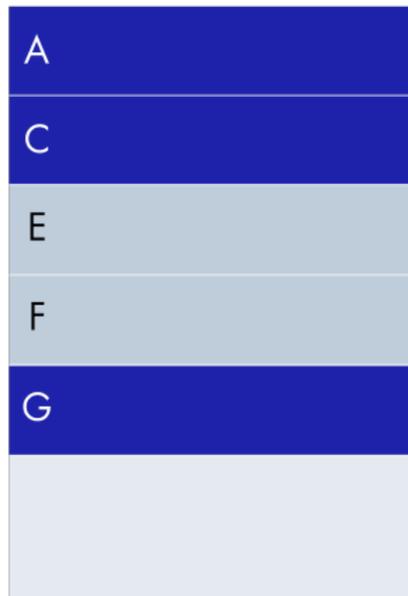
Semi-Space

Scheme

Space 1 (T)

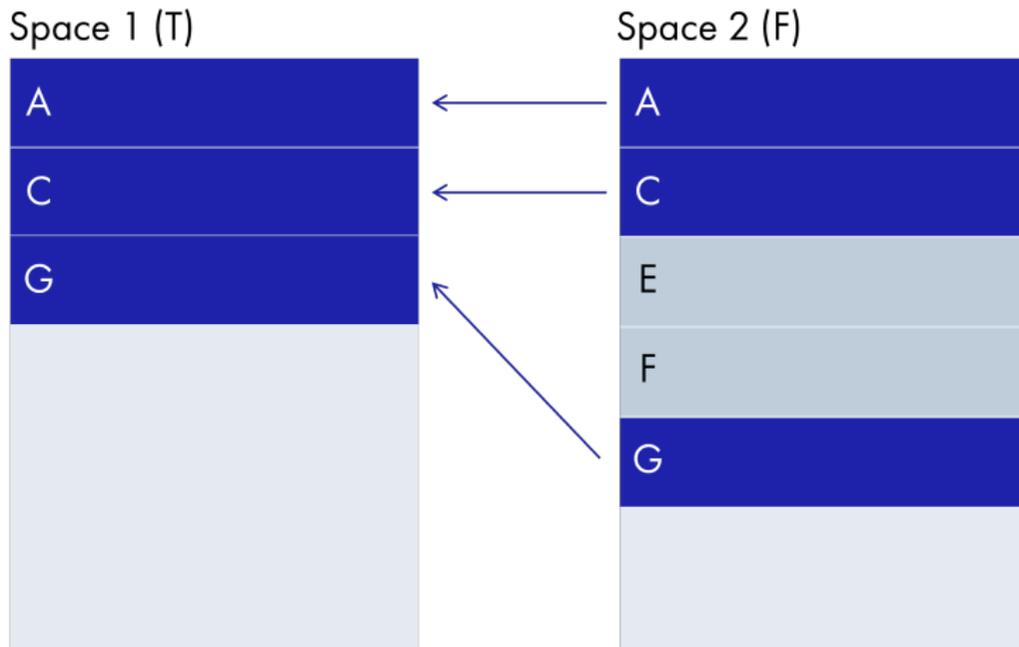


Space 2 (F)



Semi-Space

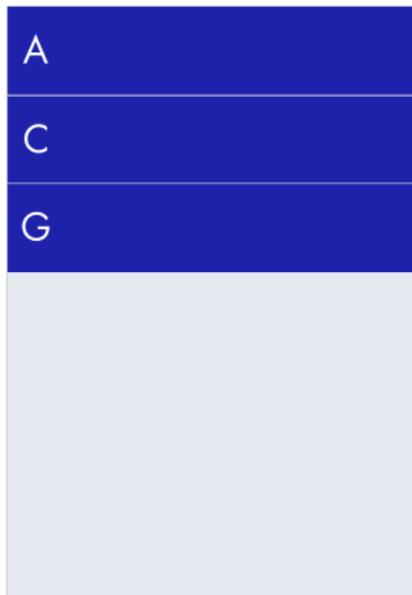
Scheme



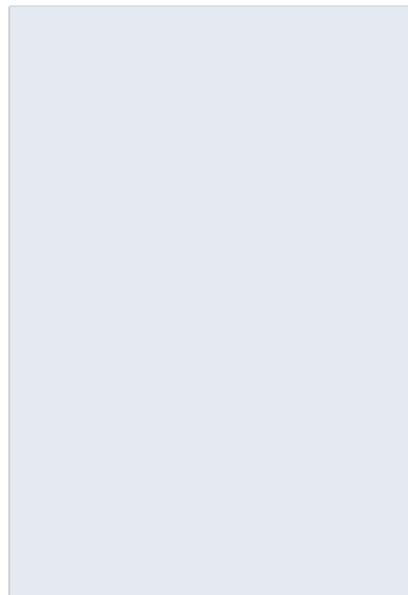
Semi-Space

Scheme

Space 1 (T)



Space 2 (F)



Semi-Space

Problems

Semi-Space

Problems

- ▶ STW Compation

Semi-Space

Problems

- ▶ STW Compation
 - ▶ UI Stalls

Semi-Space

Problems

- ▶ STW Compation
 - ▶ UI Stalls
 - ▶ Network package loses

Semi-Space

Problems

- ▶ STW Compation
 - ▶ UI Stalls
 - ▶ Network package loses
 - ▶ ...

Semi-Space

Problems

- ▶ STW Compation
 - ▶ UI Stalls
 - ▶ Network package loses
 - ▶ ...
- ▶ 2x times smaller heap size

Plan

1. ART Overview
2. General GC information
3. Mark & Sweep
4. Semi-Space
5. **Concurrent Copying**
6. Concurrent Mark Compact

Concurrent Copying

Overview

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h

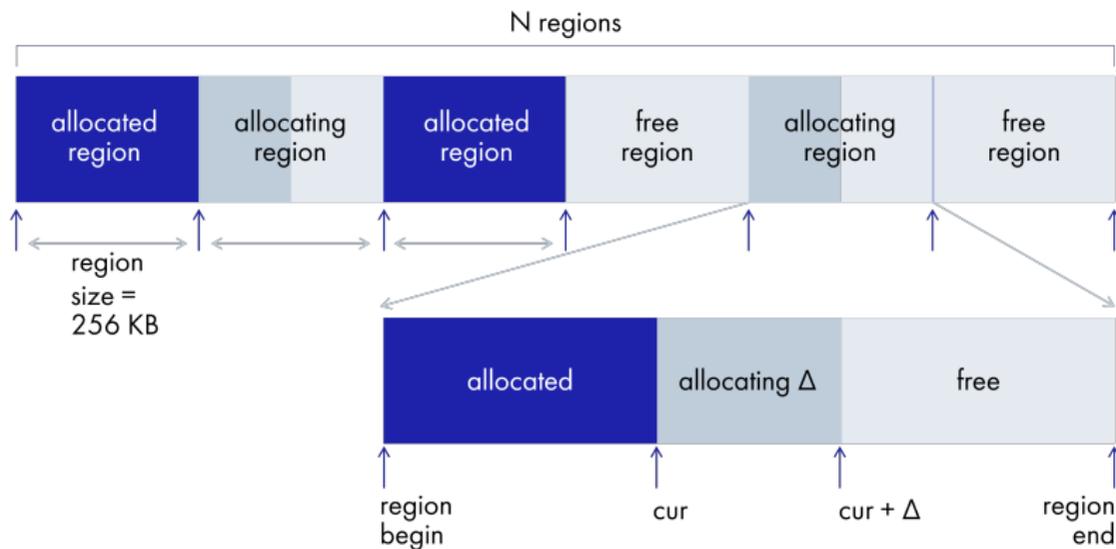
Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*

Concurrent Copying

Region Space



Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**
- ▶ Uses *mark-is-relocate* concept

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**
- ▶ Uses *mark-is-relocate* concept
- ▶ Fully Concurrent

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**
- ▶ Uses *mark-is-relocate* concept
- ▶ Fully Concurrent
- ▶ Requires Read Barrier

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**
- ▶ Uses *mark-is-relocate* concept
- ▶ Fully Concurrent
- ▶ Requires Read Barrier
- ▶ Supports *generational* & *young* collections

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**
- ▶ **Uses *mark-is-relocate* concept**
- ▶ Fully Concurrent
- ▶ Requires Read Barrier
- ▶ Supports *generational* & *young* collections

Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**
- ▶ **Uses *mark-is-relocate* concept**
- ▶ **Fully Concurrent**
- ▶ Requires Read Barrier
- ▶ Supports *generational* & *young* collections

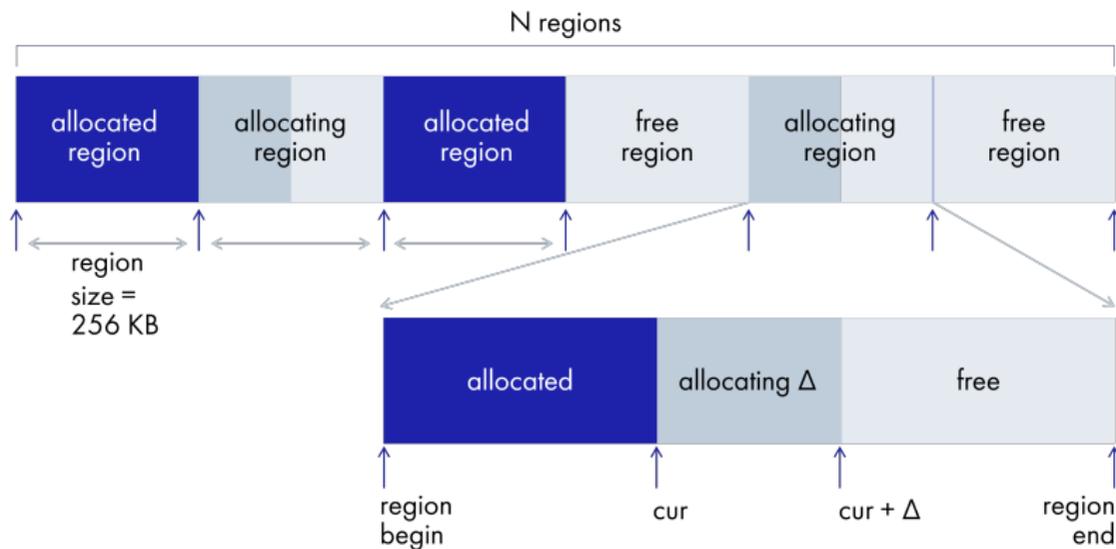
Concurrent Copying

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/gc/collector/concurrent_copying.h
- ▶ Uses *Region Space*
 - ▶ Moving collector
- ▶ **-Xgc:CC**
- ▶ **Uses *mark-is-relocate* concept**
- ▶ **Fully Concurrent**
- ▶ **Requires Read Barrier**
- ▶ Supports *generational* & *young* collections

Concurrent Copying

Region Space



Concurrent Copying



Concurrent Copying



Concurrent Copying

INVARIANT
Mutator always holds correct references

Concurrent Copying

Read Barrier

Executed as 2nd stage of reference field reading

Concurrent Copying

Read Barrier

Executed as 2nd stage of reference field reading

Without RB

A f = o.f;

Concurrent Copying

Read Barrier

Executed as 2nd stage of reference field reading

Without RB

```
A f = o.f;
```

With RB

```
A f = o.f;  
if (thread->rb_enabled) {  
    f = __rb_resolve(f);  
}
```

Concurrent Copying

Read Barrier

Executed as 2nd stage of reference field reading

Without RB

```
A f = o.f;
```

With RB

```
A f = o.f;  
if (thread->rb_enabled) {  
    f = __rb_resolve(f);  
}
```

Concurrent Copying

Read Barrier

```
Object _rb_resolve(Object o) {  
    if (isToRegion(o))  
        return o;  
    if (isMarked(o))  
        return toRegionPtr(o);  
    return mark_and_relocate(o);  
}
```

Concurrent Copying

Read Barrier

```
Object _rb_resolve(Object o) {  
    if (isToRegion(o))  
        return o;  
    if (isMarked(o))  
        return toRegionPtr(o);  
    return mark_and_relocate(o);  
}
```

Concurrent Copying

Read Barrier

```
Object _rb_resolve(Object o) {  
    if (isToRegion(o))  
        return o;  
    if (isMarked(o))  
        return toRegionPtr(o);  
    return mark_and_relocate(o);  
}
```

Concurrent Copying

Read Barrier

```
Object __rb_resolve(Object o) {  
    if (isToRegion(o))  
        return o;  
    if (isMarked(o))  
        return toRegionPtr(o);  
    return mark_and_relocate(o);  
}
```

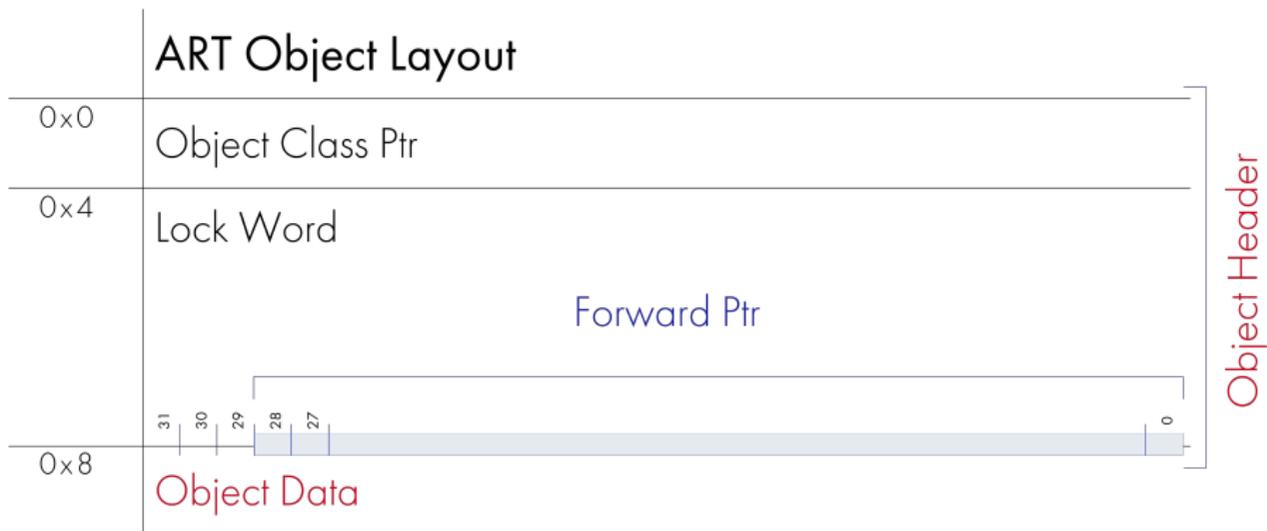
Concurrent Copying

Read Barrier

```
Object _rb_resolve(Object o) {  
    if (isToRegion(o))  
        return o;  
    if (isMarked(o))  
        return toRegionPtr(o);  
    return mark_and_relocate(o);  
}
```

Concurrent Copying

Pointer Forwarding



Concurrent Copying

Algorithm

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated
 - 2.2 Enable Read Barrier

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated
 - 2.2 Enable Read Barrier
3. Walk mark stack and collect alive subgraph

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated
 - 2.2 Enable Read Barrier
3. Walk mark stack and collect alive subgraph
 - 3.1 Relocate objects in *from-regions* to *to-regions*

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated
 - 2.2 Enable Read Barrier
3. Walk mark stack and collect alive subgraph
 - 3.1 Relocate objects in *from-regions* to *to-regions*
 - 3.2 Once stack walking is done there is no alive object left in *from-region*

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated
 - 2.2 Enable Read Barrier
3. Walk mark stack and collect alive subgraph
 - 3.1 Relocate objects in *from-regions* to *to-regions*
 - 3.2 Once stack walking is done there is no alive object left in *from-region*
4. Sync mutators and GC threads

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated
 - 2.2 Enable Read Barrier
3. Walk mark stack and collect alive subgraph
 - 3.1 Relocate objects in *from-regions* to *to-regions*
 - 3.2 Once stack walking is done there is no alive object left in *from-region*
4. Sync mutators and GC threads
 - 4.1 Disable Read Barrier

Concurrent Copying

Algorithm

1. Determine which regions are to be evacuated (*from-region*)
2. Collect root set (STW)
 - 2.1 During this phase root set objects are relocated
 - 2.2 Enable Read Barrier
3. Walk mark stack and collect alive subgraph
 - 3.1 **Relocate objects in *from-regions* to *to-regions***
 - 3.2 Once stack walking is done there is no alive object left in *from-region*
4. Sync mutators and GC threads
 - 4.1 Disable Read Barrier

Concurrent Copying

Generations

Concurrent Copying

Generations

1. Minor (young gen) collection
 - ▶ Evacuate only newly allocated regions

Concurrent Copying

Generations

1. Minor (young gen) collection
 - ▶ Evacuate only newly allocated regions
2. Major collection
 - ▶ Evacuate newly allocated and regions filled $> 75\%$

Concurrent Copying

Generations

1. Minor (young gen) collection
 - ▶ Evacuate only newly allocated regions
2. Major collection
 - ▶ Evacuate newly allocated and regions filled $> 75\%$
3. Full collection
 - ▶ Evacuate all regions

Concurrent Copying

Problems

Concurrent Copying

Problems

- ▶ Read barrier overhead
 - ▶ Lower app throughput compared to other algorithms

Plan

1. ART Overview
2. General GC information
3. Mark & Sweep
4. Semi-Space
5. Concurrent Copying
6. **Concurrent Mark Compact**

Concurrent Mark Compact

Overview

Concurrent Mark Compact

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/mark_compact.h

Concurrent Mark Compact

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/mark_compact.h
- ▶ Uses Single *Bump Pointer Space*

Concurrent Mark Compact

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/mark_compact.h
- ▶ Uses Single *Bump Pointer Space*
 - ▶ Moving collector

Concurrent Mark Compact

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/mark_compact.h
- ▶ Uses Single *Bump Pointer Space*
 - ▶ Moving collector
- ▶ **-Xgc:CMC**

Concurrent Mark Compact

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/mark_compact.h
- ▶ Uses Single *Bump Pointer Space*
 - ▶ Moving collector
- ▶ **-Xgc:CMC**
- ▶ Fully Concurrent

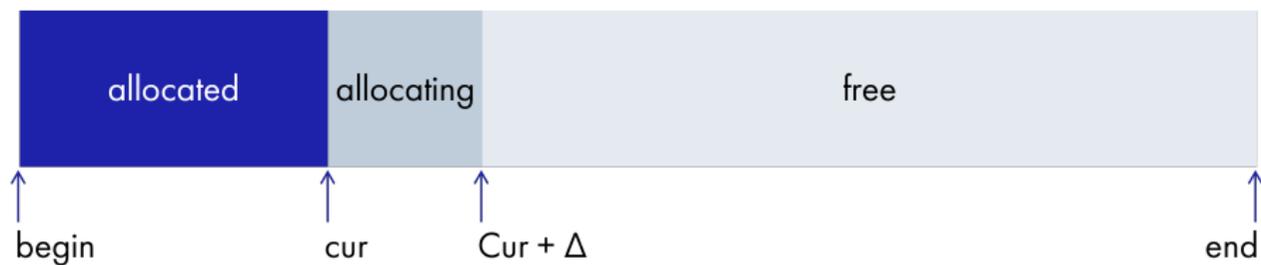
Concurrent Mark Compact

Overview

- ▶ https://cs.android.com/android/platform/superproject/main/+main:art/runtime/gc/collector/mark_compact.h
- ▶ Uses Single *Bump Pointer Space*
 - ▶ Moving collector
- ▶ **-Xgc:CMC**
- ▶ Fully Concurrent
- ▶ *The Pauseless GC Algorithm by Click, Tene & Wolf*

Concurrent Mark Compact

Bump Pointer Space



Concurrent Mark Compact

INVARIANT
Mutator always holds correct references

Concurrent Mark Compact

Read Barrier

Executed as 2nd stage of reference field reading

Without RB

```
A f = o.f;
```

With RB

```
A f = o.f;  
if (thread->rb_enabled) {  
    f = __rb_resolve(f);  
}
```

Concurrent Mark Compact

MMU Trigger

Concurrent Mark Compact

MMU Trigger

Thread 1

```
mprotect(mem, NONE) // t=0
// Do something ... // 1<t<4
mprotect(mem, READ | WRITE) // t=5
```

Concurrent Mark Compact

MMU Trigger

Thread 1

```
mprotect(mem, NONE) // t=0
// Do something ... // 1<t<4
mprotect(mem, READ | WRITE) // t=5
```

Thread 2

```
Value = *mem // t=1, SIGSEGV
// Do something else
mprotect(mem, READ | WRITE)
```

Concurrent Mark Compact

MMU Trigger

Thread 1

```
mprotect(mem, NONE) // t=0
// Do something ... // 1<t<4
mprotect(mem, READ | WRITE) // t=5
```

Thread 2

```
Value = *mem // t=1, SIGSEGV
// Do something else
mprotect(mem, READ | WRITE)
```

1. Granularity – memory page

Concurrent Mark Compact

MMU Trigger

Thread 1

```
mprotect(mem, NONE) // t=0
// Do something ... // 1<t<4
mprotect(mem, READ | WRITE) // t=5
```

1. Granularity – memory page
2. Uses heavy VMA structures

Thread 2

```
Value = *mem // t=1, SIGSEGV
// Do something else
mprotect(mem, READ | WRITE)
```

Concurrent Mark Compact

MMU Trigger

Thread 1

```
mprotect(mem, NONE) // t=0
// Do something ... // 1<t<4
mprotect(mem, READ | WRITE) // t=5
```

Thread 2

```
Value = *mem // t=1, SIGSEGV
// Do something else
mprotect(mem, READ | WRITE)
```

1. Granularity – memory page
2. Uses heavy VMA structures
 - 2.1 *userfaultfd* kernel feature (5.15+) does not use VMA

Concurrent Mark Compact

MMU Trigger

Thread 1

```
mprotect(mem, NONE) // t=0
// Do something ... // 1<t<4
mprotect(mem, READ | WRITE) // t=5
```

Thread 2

```
Value = *mem // t=1, SIGSEGV
// Do something else
mprotect(mem, READ | WRITE)
```

1. Granularity – memory page
2. Uses heavy VMA structures
 - 2.1 *userfaultfd* kernel feature (5.15+) does not use VMA
 - 2.2 <https://man7.org/linux/man-pages/man2/userfaultfd.2.html>

Concurrent Mark Compact

userfaultfd

Usage Overview:

Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)

Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)
- ▶ Register memory, associate file descriptor

Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)
- ▶ Register memory, associate file descriptor
- ▶ Kernel delivers a `SIGBUS` signal on access to an unpopulated page

Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)
- ▶ Register memory, associate file descriptor
- ▶ Kernel delivers a `SIGBUS` signal on access to an unpopulated page
- ▶ App's signal handler fills the page (e.g. from file, network, etc.)

Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)
- ▶ Register memory, associate file descriptor
- ▶ Kernel delivers a `SIGBUS` signal on access to an unpopulated page
- ▶ App's signal handler fills the page (e.g. from file, network, etc.)

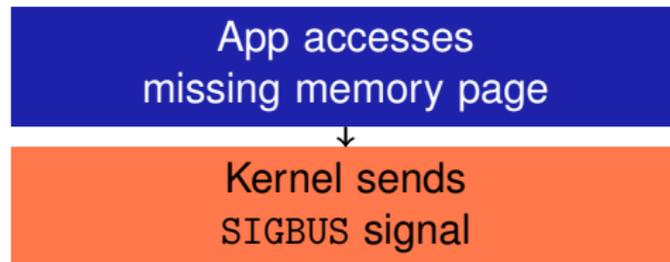
App accesses
missing memory page

Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)
- ▶ Register memory, associate file descriptor
- ▶ Kernel delivers a `SIGBUS` signal on access to an unpopulated page
- ▶ App's signal handler fills the page (e.g. from file, network, etc.)

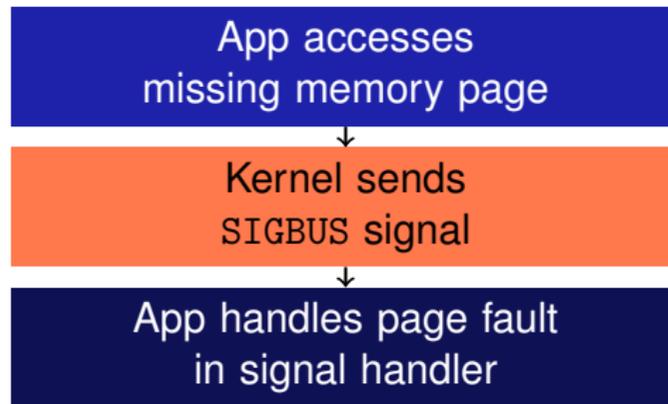


Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)
- ▶ Register memory, associate file descriptor
- ▶ Kernel delivers a `SIGBUS` signal on access to an unpopulated page
- ▶ App's signal handler fills the page (e.g. from file, network, etc.)

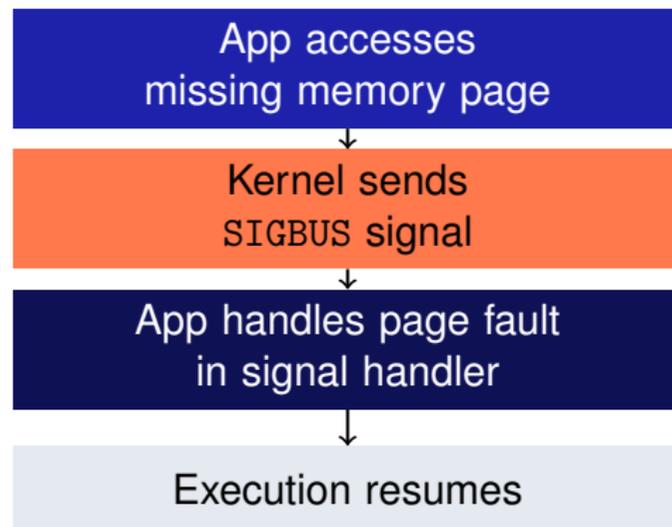


Concurrent Mark Compact

userfaultfd

Usage Overview:

- ▶ Allocate memory (e.g. `mmap`)
- ▶ Register memory, associate file descriptor
- ▶ Kernel delivers a `SIGBUS` signal on access to an unpopulated page
- ▶ App's signal handler fills the page (e.g. from file, network, etc.)



Concurrent Mark Compact

Algorithm

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)
 - 4.1 If mutator triggers SIGBUS it processes requested page itself

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)
 - 4.1 If mutator triggers SIGBUS it processes requested page itself
5. Release userfaultfd and GC structs

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)
 - 4.1 If mutator triggers SIGBUS it processes requested page itself
5. Release userfaultfd and GC structs

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)
 - 4.1 If mutator triggers SIGBUS it processes requested page itself
5. Release userfaultfd and GC structs

Concurrent Mark Compact

Address Translation

Concurrent Mark Compact

Address Translation



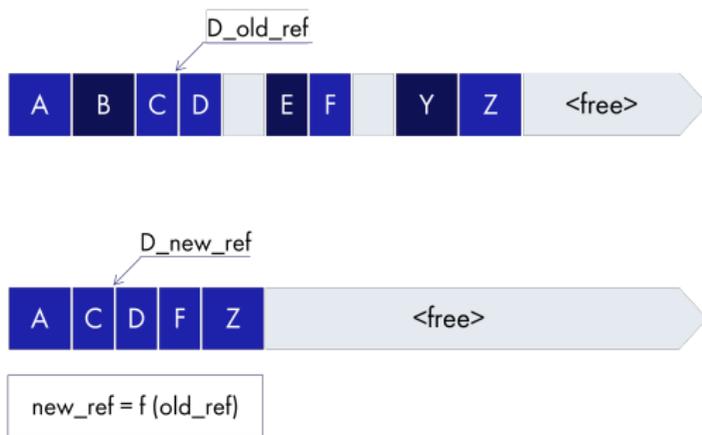
Concurrent Mark Compact

Address Translation



Concurrent Mark Compact

Address Translation



Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)
 - 4.1 If mutator triggers SIGBUS it processes requested page itself
5. Release userfaultfd and GC structs

Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)
 - 4.1 If mutator triggers SIGBUS it processes requested page itself
5. Release userfaultfd and GC structs

Concurrent Mark Compact

Page Processing

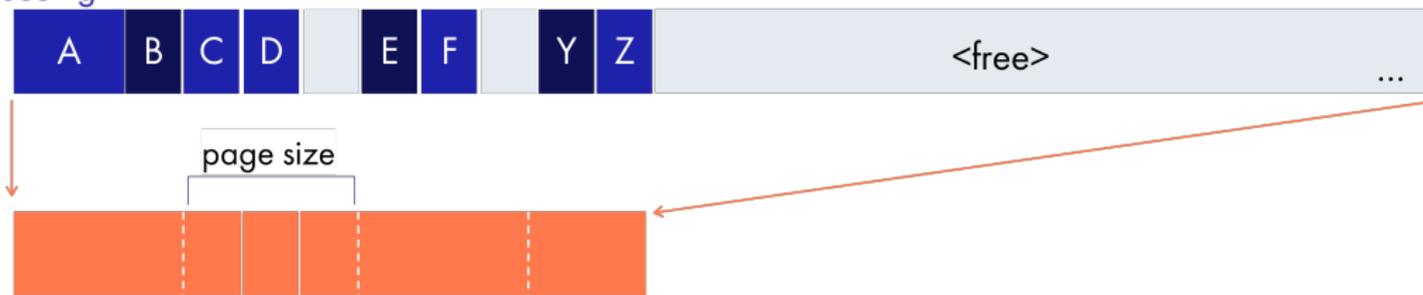
Concurrent Mark Compact

Page Processing



Concurrent Mark Compact

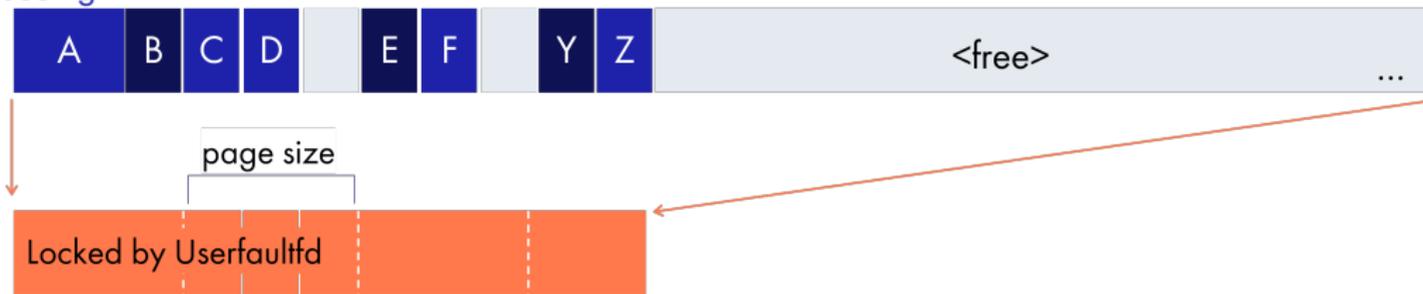
Page Processing



Heap Space = Virtual Addresses

Concurrent Mark Compact

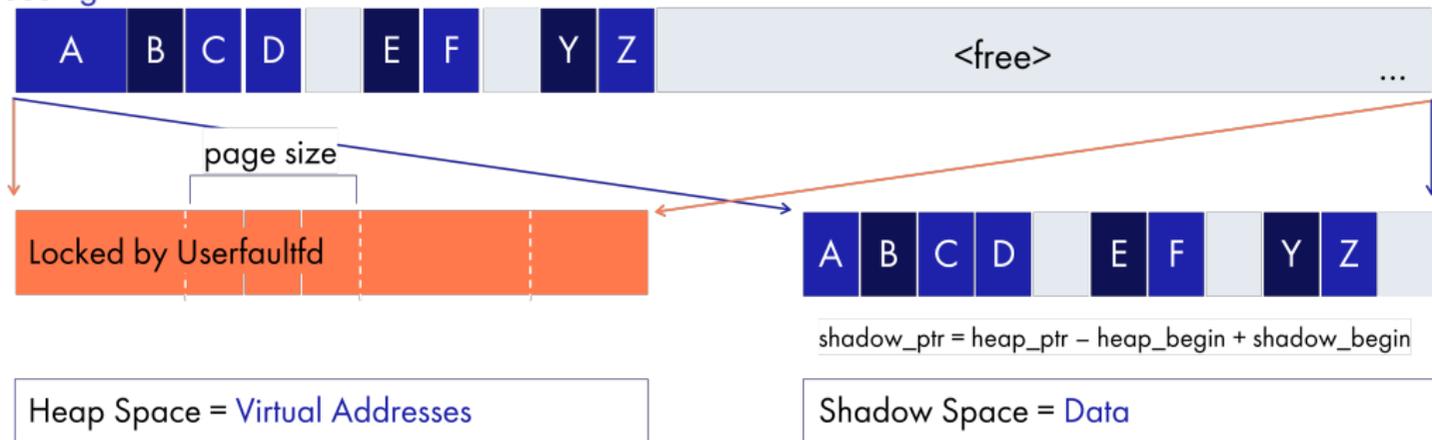
Page Processing



Heap Space = Virtual Addresses

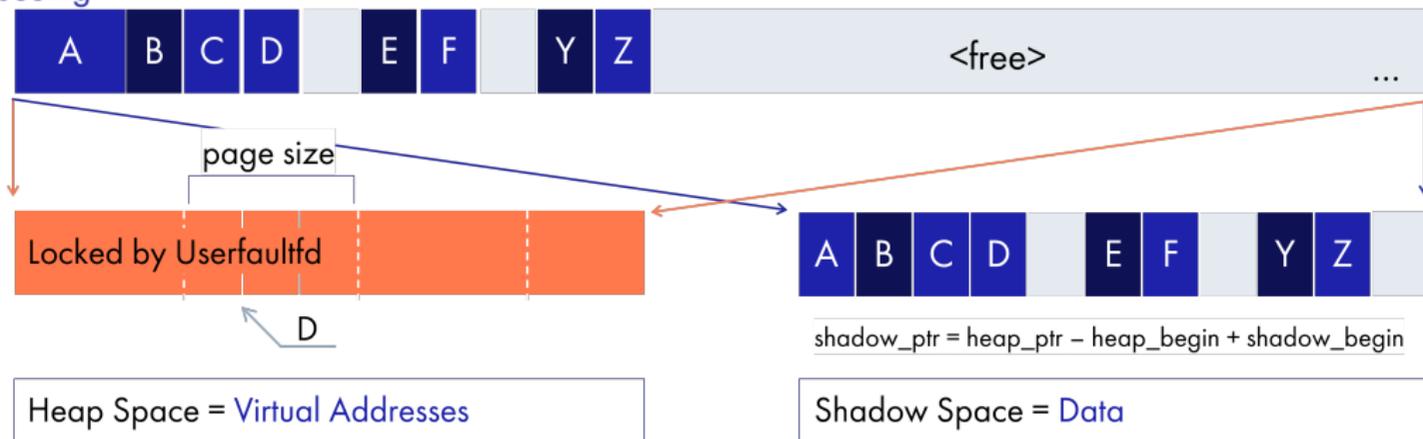
Concurrent Mark Compact

Page Processing



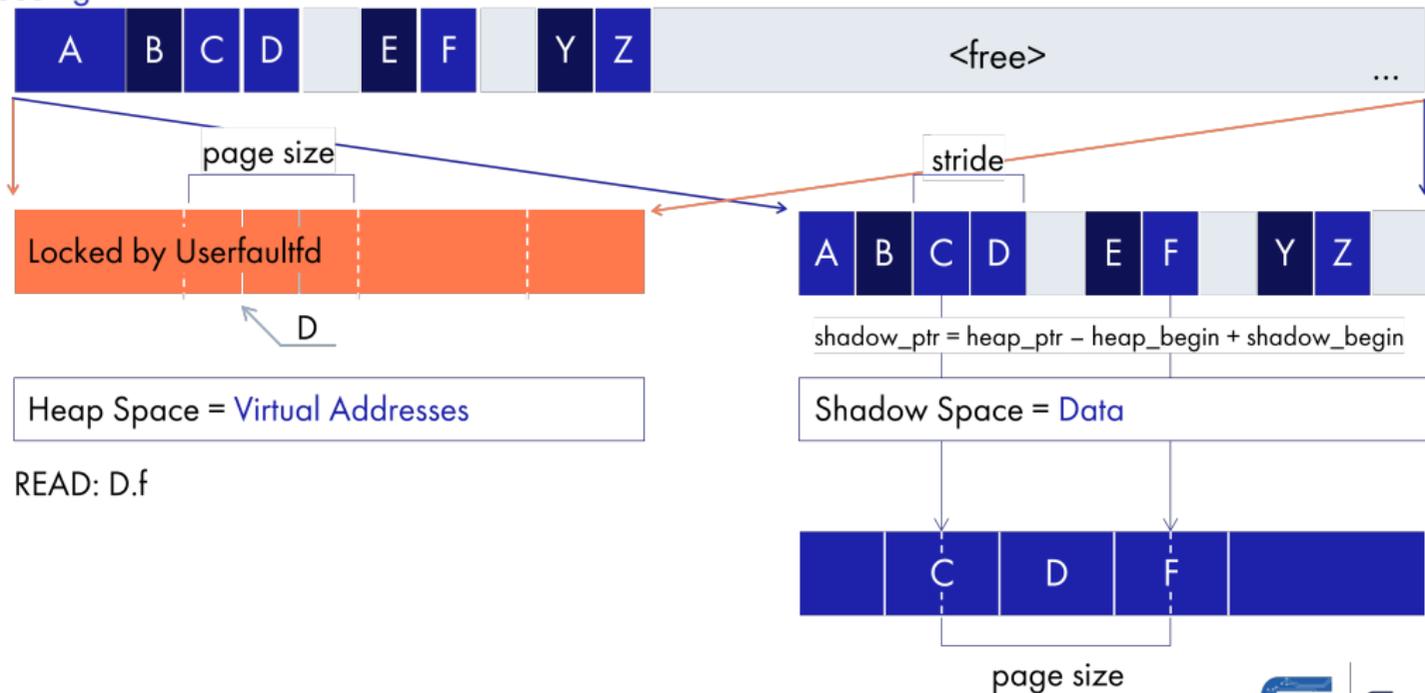
Concurrent Mark Compact

Page Processing



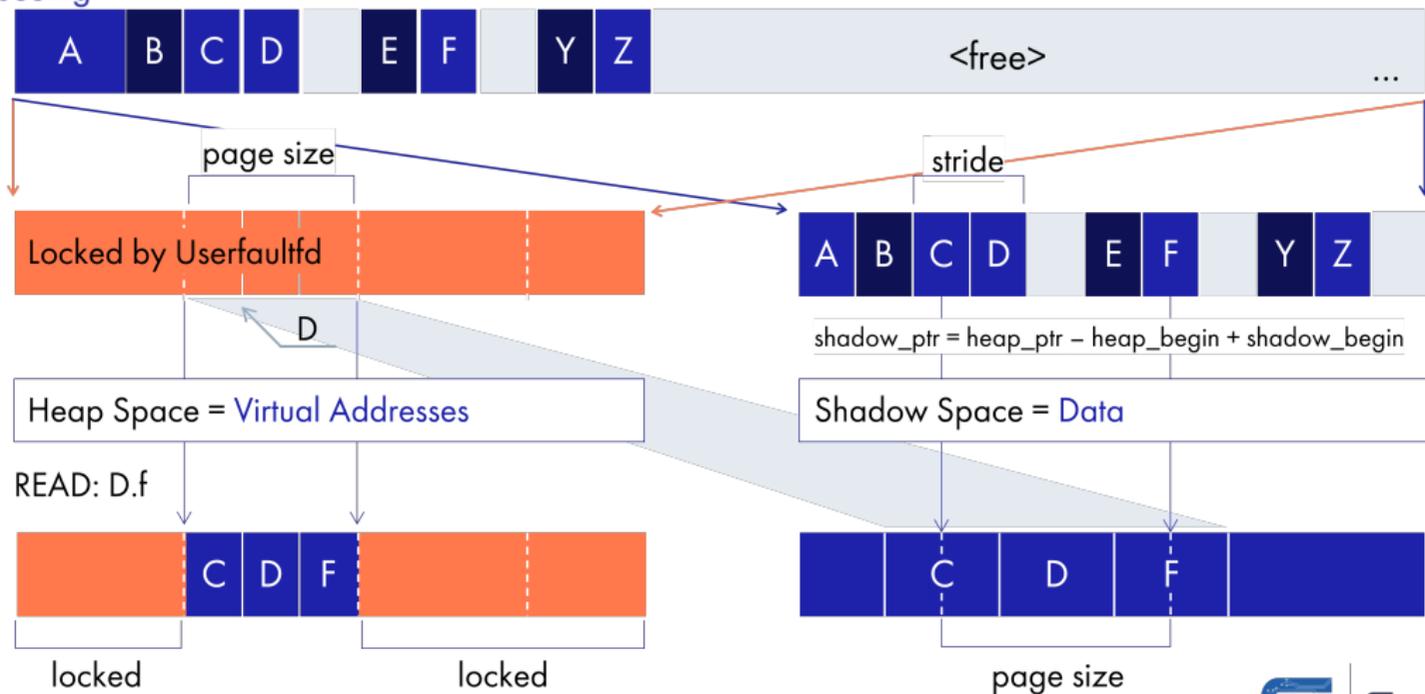
Concurrent Mark Compact

Page Processing



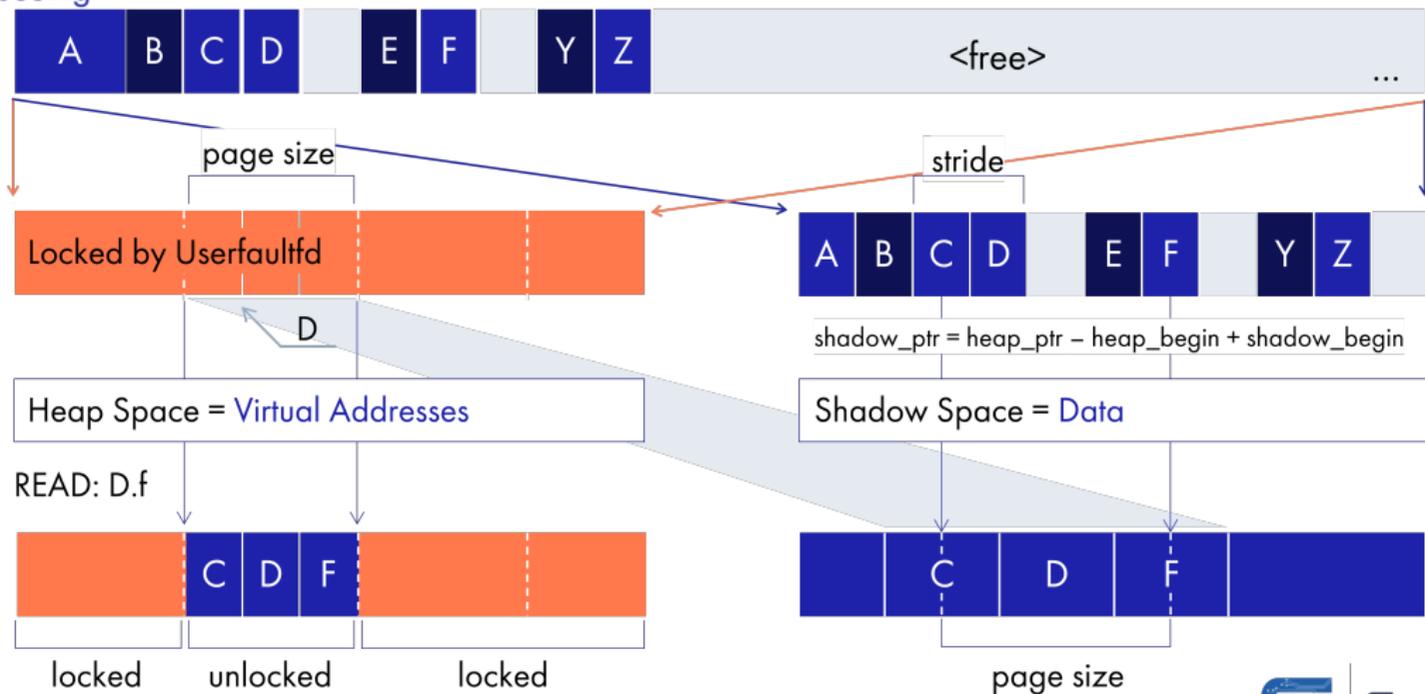
Concurrent Mark Compact

Page Processing



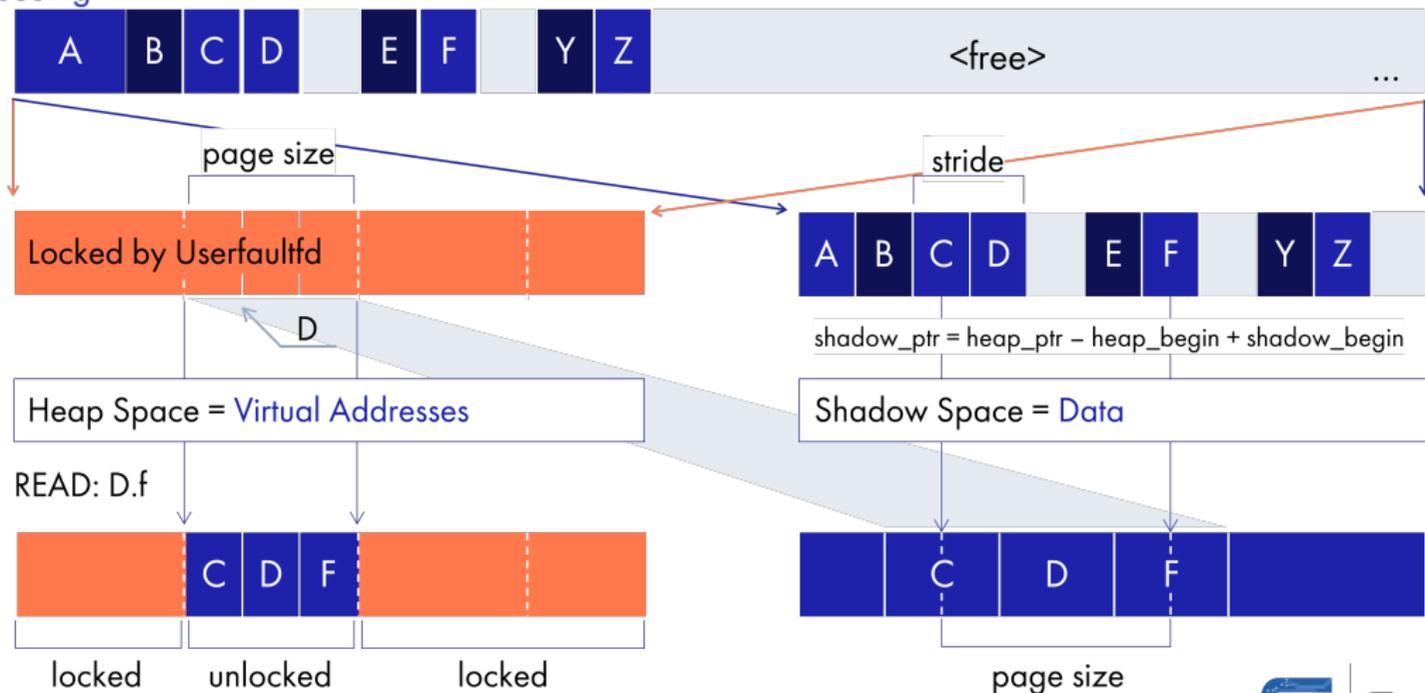
Concurrent Mark Compact

Page Processing



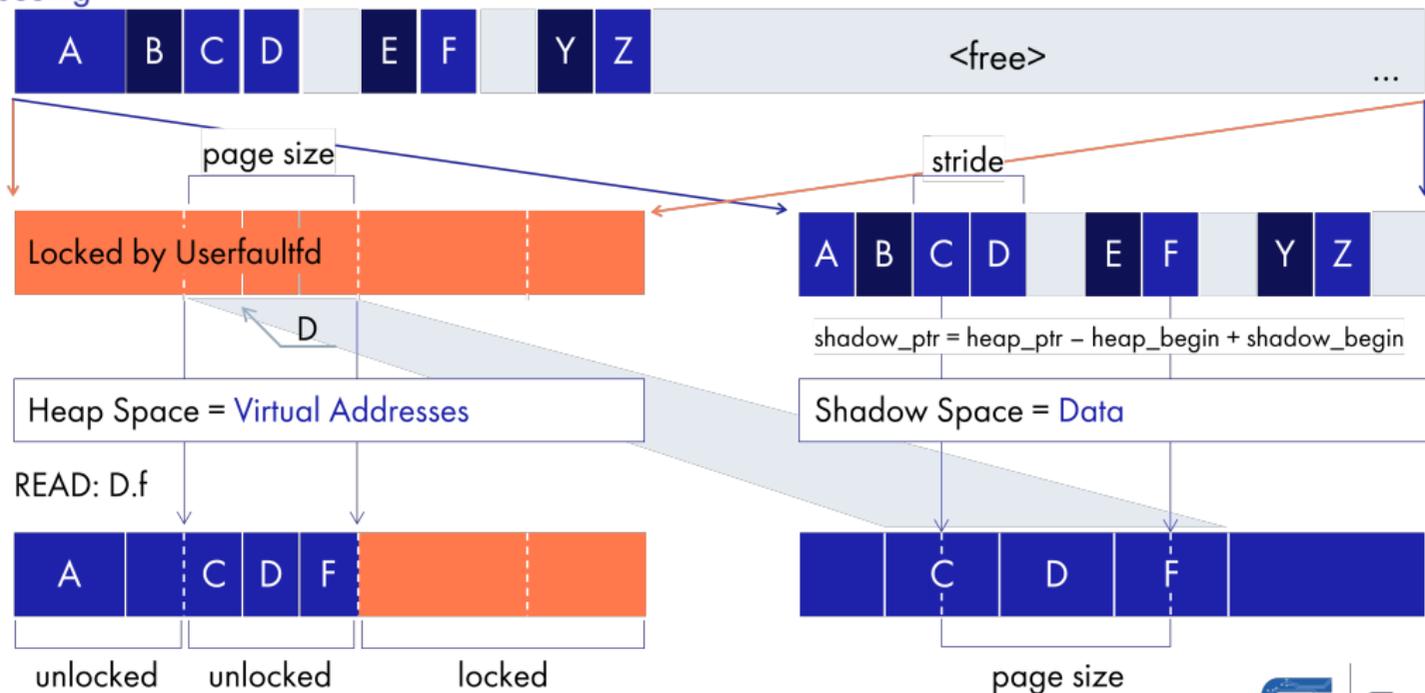
Concurrent Mark Compact

Page Processing



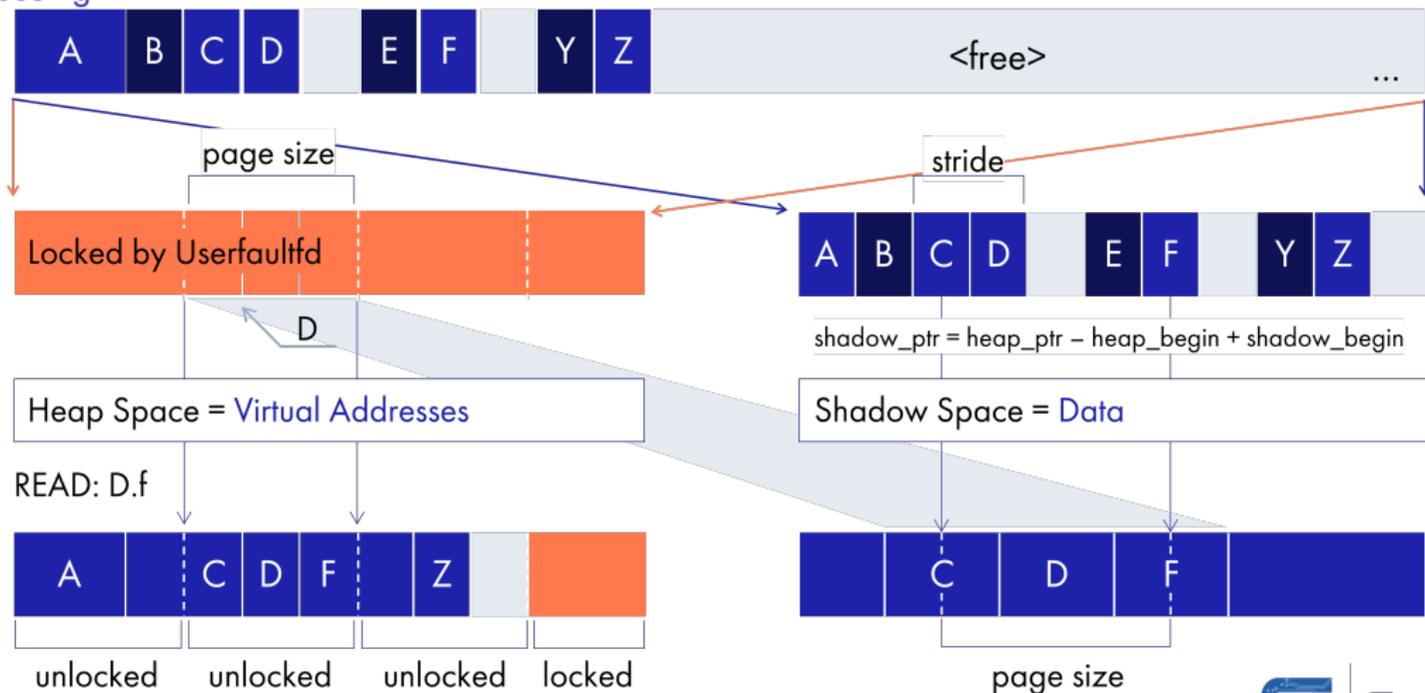
Concurrent Mark Compact

Page Processing



Concurrent Mark Compact

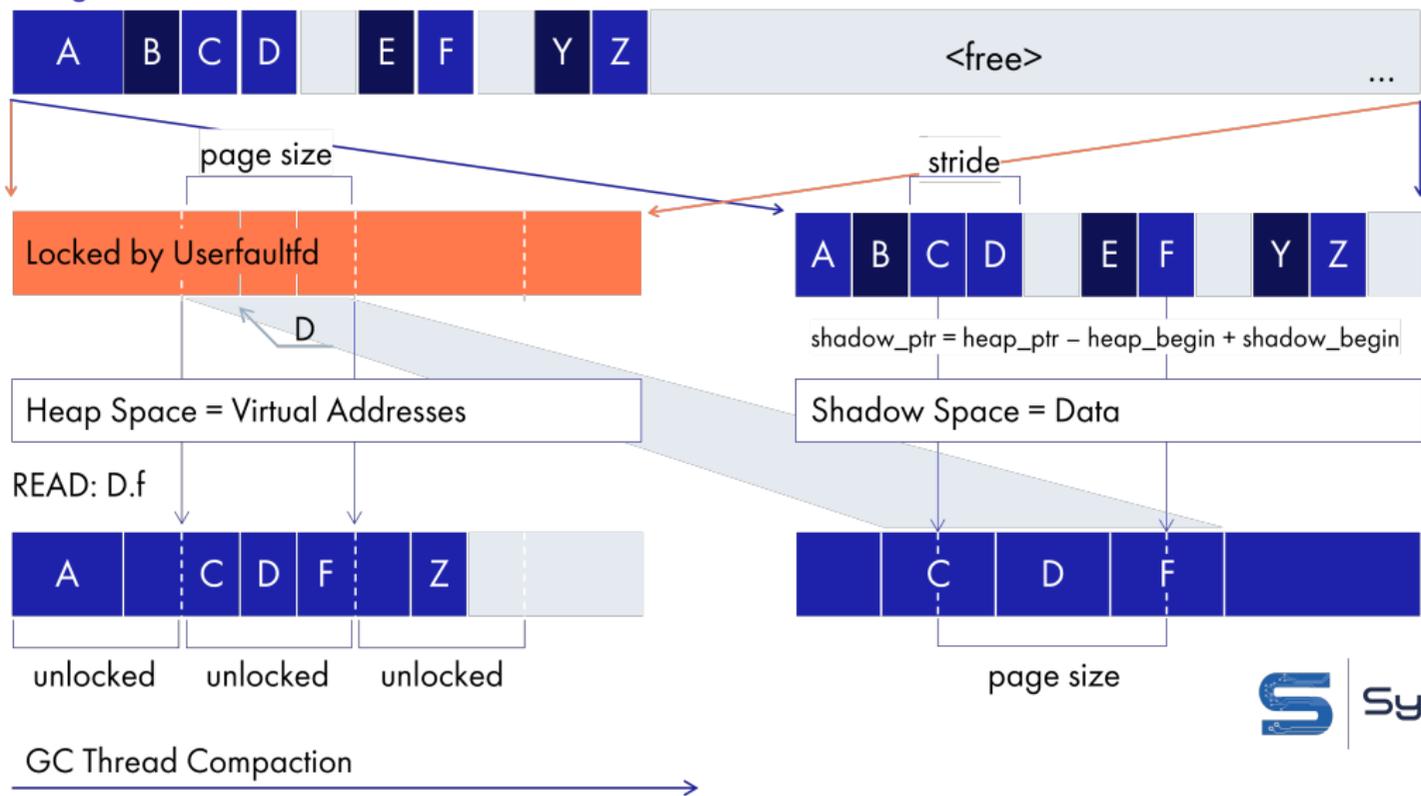
Page Processing



GC Thread Compaction →

Concurrent Mark Compact

Page Processing



Concurrent Mark Compact

Algorithm

1. Collect roots (STW)
2. Walk mark stack
3. Prepare for compaction (STW)
 - 3.1 Pre-compute evacuated objects addresses
 - 3.2 Remap relocating heap to shadow space
 - 3.3 Lock relocating virtual space using userfaultfd
 - 3.4 Update root references (no relocation)
4. Process each heap page separately (GC thread)
 - 4.1 If mutator triggers SIGBUS it processes requested page itself
5. Release userfaultfd and GC structs

Concurrent Mark Compact

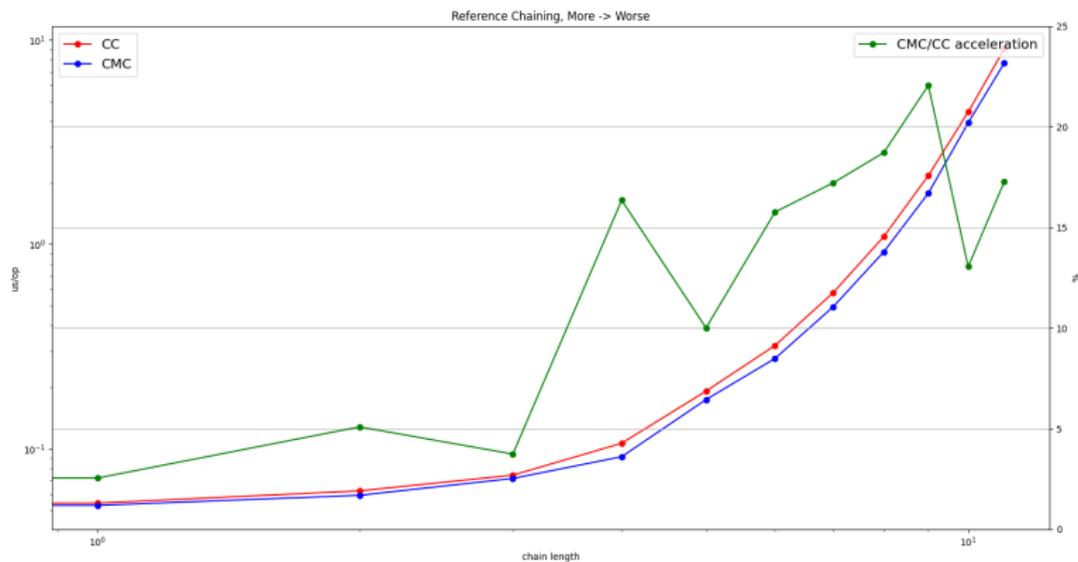
Compare

```
@Benchmark
int ReferenceChaining(Chain c) {
    int r = 0;
    while (c != null) {
        r += c.value;
        c = c.next;
    }
    return r;
}
```

Concurrent Mark Compact

Compare

```
@Benchmark
int ReferenceChaining(Chain c) {
    int r = 0;
    while (c != null) {
        r += c.value;
        c = c.next;
    }
    return r;
}
```



Q/A

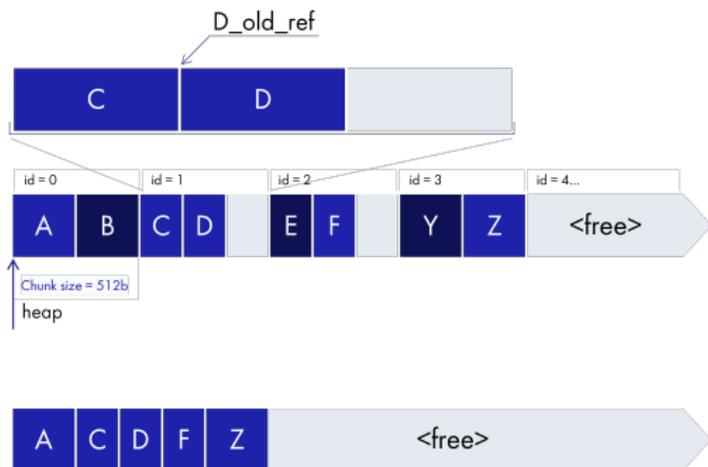
- ▶ AOSP repo
<https://cs.android.com/android/platform/superproject/main>
- ▶ The Pauseless GC Algorithm by Click, Tene & Wolf
https://www.researchgate.net/publication/221137840_The_pauseless_GC_algorithm
- ▶ Userfaultfd
<https://man7.org/linux/man-pages/man2/userfaultfd.2.html>

Concurrent Mark Compact

Address Translation

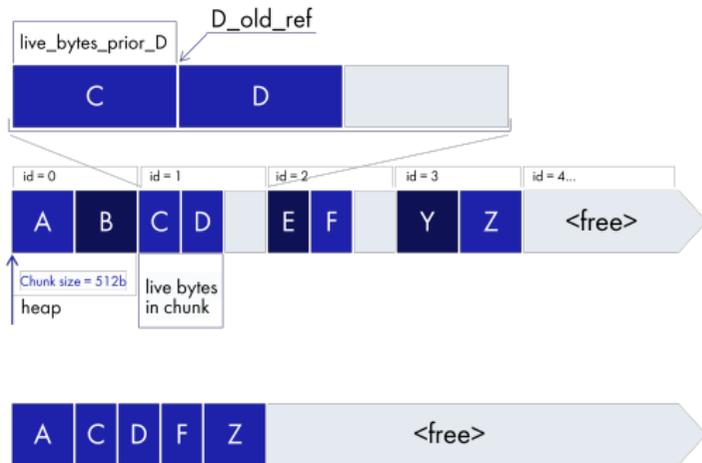
Concurrent Mark Compact

Address Translation



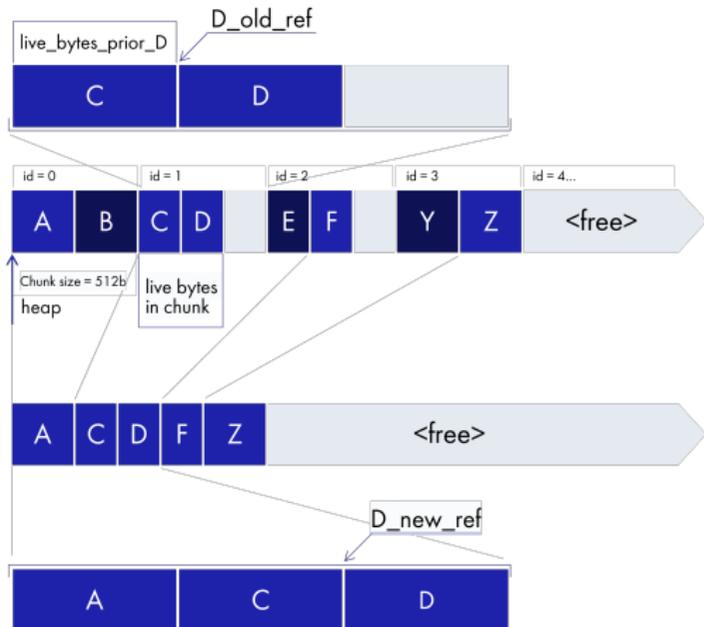
Concurrent Mark Compact

Address Translation



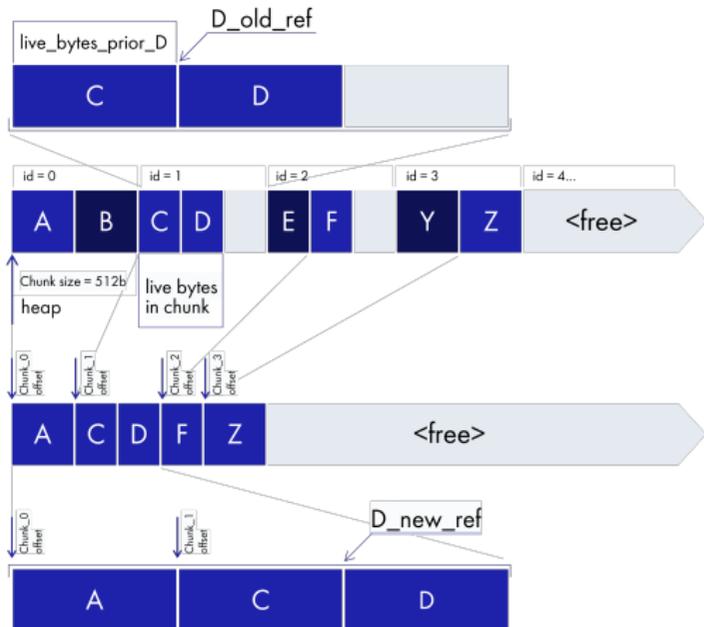
Concurrent Mark Compact

Address Translation



Concurrent Mark Compact

Address Translation



$$\text{ChunkOffset}_k = \sum_{i=0}^{k-1} \text{LiveBytes}_i$$

Concurrent Mark Compact

BitMaps

To track alive bytes, object bytes and so on

Total size = $4Gb / 8 \text{ bytes} / 8bit = 64Mb$

