Heap Heap Hooray: Memory Management

Tyler Gutowski, Trevor Schiff,
Dr. Ryan Stansifer (client)

Task	Description	Tyler	Trevor
Fix “copying” GC method	Copying garbage collection previously did not update existing references to objects copied across heaps Unexpected objective not in original plans	0.3	0.7
Run tests across refcount/marksweep/copying garbage collection methods	Create and run thorough test cases across all garbage collection implementations	0.7	0.3
Implement “generational” GC method	“Generational” garbage collection algorithm	0.5	0.5
Write and execute tests for “generational” GC method	Create and run thorough test cases for the generational GC algorithm Blocked by ‘Implement “generational” GC method’	0.5	0.5
Create poster and ebook page for Senior Design Showcase	Create display material (poster and e-book page) for the Senior Design Showcase	0.8	0.2
Conduct evaluation and analyze results	Compare our GC implementations, gather metrics and draw meaningful conclusions about their effectiveness with source code exhibiting certain algorithms/structures.	N/A	N/A

Recap: Copying

When we try to allocate memory, but none is available, perform:

1. Mark Phase
    a. Mark all heap objects reachable from local variables
2. Copy Phase
    a. Copy all live allocations from the “from” heap to the “to” heap
        i. “To” heap contains only live (reachable) allocations
    b. Release allocations which exist in the “from” heap
        i. All memory associated with garbage has been released
    c. Swap handles to the “from” and “to” heaps
        i. “From” heap contains only live (reachable allocations)

Recap: Copying

“From” heap

Name	Address	Size	Marked?
dummy	0x25028	0x000C	Yes
one	0x25068	0x000C	No
two	0x25088	0x000C	No
(free space)	N/A	0xFFDB	N/A

Begin “copy” phase.

“To” heap

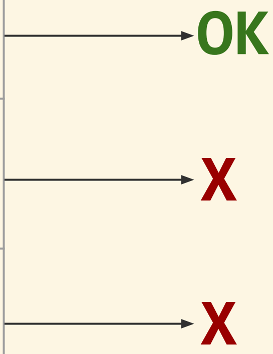
Name	Address	Size	Marked?
(free space)	N/A	0xFFFF	N/A

Recap: Copying

“From” heap

Name	Address	Size	Marked?
dummy	0x25028	0x000C	Yes
one	0x25068	0x000C	No
two	0x25088	0x000C	No
(free space)	N/A	0xFFDB	N/A

Copy all live allocations to the “to” heap



“To” heap

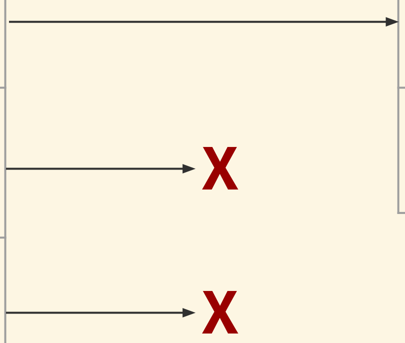
Name	Address	Size	Marked?
(free space)	N/A	0xFFFF	N/A

Recap: Copying

“From” heap

Name	Address	Size	Marked?
dummy	0x25028	0x000C	Yes
one	0x25068	0x000C	No
two	0x25088	0x000C	No
(free space)	N/A	0xFFDB	N/A

Copy all live allocations to the “to” heap



“To” heap

Name	Address	Size	Marked?
dummy	0x36020	0x000C	No
(free space)	N/A	0xFFF3	N/A

‘dummy’ contents have a new memory address!

Recap: Copying

“From” heap

Name	Address	Size	Marked?
~~*dummy*~~	~~0x25028~~	~~0x000C~~	~~Yes~~
~~*one*~~	~~0x25068~~	~~0x000C~~	~~No~~
~~*two*~~	~~0x25088~~	~~0x000C~~	~~No~~
(free space)	N/A	0xFFDB	N/A

Release allocations in the “from” heap

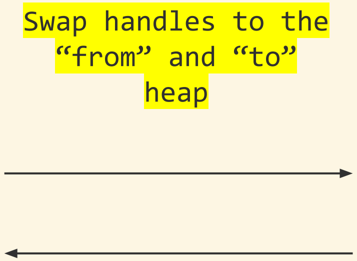
“To” heap

Name	Address	Size	Marked?
dummy	0x36020	0x000C	No
(free space)	N/A	0xFFF3	N/A

Recap: Copying

“From” heap

Name	Address	Size	Marked?
(free space)	N/A	0xFFFF	N/A



Swap handles to the “from” and “to” heap

“To” heap

Name	Address	Size	Marked?
dummy	0x36020	0x000C	No
(free space)	N/A	0xFFF3	N/A

Recap: Copying

“From” heap

Name	Address	Size	Marked?
dummy	0x36020	0x000C	No
(free space)	N/A	0xFFF3	N/A

What happens if we try to access ‘dummy’ from the MiniJava program?

Garbage collection cycle is finished.

“To” heap

Name	Address	Size	Marked?
(free space)	N/A	0xFFFF	N/A

Recap: Copying

```
root@debian:~# ./UseAfterCopyTest
[runtime/config.c:0063] setting heaptype to Chunk
[runtime/config.c:0044] setting gctype to Copying
[runtime/stackframe.c:0054] stackframe_push 0x40800670 (size:96)
[runtime/heap/heap.c:0061] try alloc (size:12)
[runtime/heap/heap.c:0092] alloc success 0x29040 (size:12), userptr: 0x29048
[runtime/stackframe.c:0054] stackframe_push 0x40800610 (size:92)
[runtime/stackframe.c:0071] stackframe_pop
[runtime/stackframe.c:0096] search stack frame: 0x40800670 (size:96)
[runtime/stackframe.c:0114]   local_num=1
[runtime/stackframe.c:0144]   ctx->locals[0 (align:0)] = 00029048
[runtime/stackframe.c:0181] traverse p_obj=0x29040 pp_obj=0x408006cc
[runtime/gc/copying_gc.c:0135] copying mark 0x29040
[runtime/stackframe.c:0185] traverse alloced block 0x29048
[runtime/heap/chunk_heap.c:0308] copying object at 0x29040
[runtime/heap/heap.c:0061] try alloc (size:12)
[runtime/heap/heap.c:0092] alloc success 0x390c8 (size:12), userptr: 0x390d0
[runtime/stackframe.c:0054] stackframe_push 0x40800610 (size:92)
[runtime/stackframe.c:0071] stackframe_pop
[runtime/stackframe.c:0071] stackframe_pop
0
```

The output *should* be “12345”, not “0”.

```java
class Main {
    public static void main(String[] a) {
        System.out.println(new
            UseAfterCopyTest().execute());
    }
}

class MyClass {
    int value;
    public void setValue(int x) { value = x; }
    public int getValue() { return value; }
}


class UseAfterCopyTest {
    public int execute() {
        MyClass dummy;
        dummy = new MyClass();
        dummy.setValue(12345);

        System.gc();
        return dummy.getValue();
    }
}
```

Recap: Copying

```
root@debian:~# ./UseAfterCopyTest
[runtime/config.c:0063] setting heaptype to Chunk
[runtime/config.c:0044] setting gctype to Copying
[runtime/stackframe.c:0054] stackframe_push 0x40800670 (size:96)
[runtime/heap/heap.c:0061] try alloc (size:12)
[runtime/heap/heap.c:0092] alloc success 0x29040 (size:12), userptr: 0x29048
[runtime/stackframe.c:0054] stackframe_push 0x40800610 (size:92)
[runtime/stackframe.c:0071] stackframe_pop
[runtime/stackframe.c:0096] search stack frame: 0x40800670 (size:96)
[runtime/stackframe.c:0114]   local_num=1
[runtime/stackframe.c:0144]   ctx->locals[0 (align:0)] = 00029048
[runtime/stackframe.c:0181] traverse p_obj=0x29040 pp_obj=0x408006cc
[runtime/gc/copying_gc.c:0135] copying mark 0x29040
[runtime/stackframe.c:0185] traverse alloced block 0x29048
[runtime/heap/chunk_heap.c:0308] copying object at 0x29040
[runtime/heap/heap.c:0061] try alloc (size:12)
[runtime/heap/heap.c:0092] alloc success 0x390c8 (size:12), userptr: 0x390d0
[runtime/stackframe.c:0054] stackframe_push 0x40800610 (size:92)
[runtime/stackframe.c:0071] stackframe_pop
[runtime/stackframe.c:0071] stackframe_pop
0
```

During GC, the address of ‘dummy’ changes from 0x29040 -> 0x390c8.

So, **dummy.getValue()** will erroneously read from 0x29040!

```java
class Main {
    public static void main(String[] a) {
        System.out.println(new
            UseAfterCopyTest().execute());
    }
}

class MyClass {
    int value;
    public void setValue(int x) { value = x; }
    public int getValue() { return value; }
}


class UseAfterCopyTest {
    public int execute() {
        MyClass dummy;
        dummy = new MyClass();
        dummy.setValue(12345);

        System.gc();
        return dummy.getValue();
    }
}
```

Recap: Copying

When copying objects across heaps, any reference to them must be updated.

- The “object traversal” process seen in the mark phase solves this problem
    - All *live* references to objects will be found during traversal

During the copy phase, maintain a map of addresses in the “from” heap to addresses in the “to” heap

- After the copy phase, traverse the object graph again and overwrite references we find along the way

Recap: Copying

Relevant implementation details:

```
/**
 * @brief Copy live allocations of one chunk heap to another
 *
 * @param src Source heap (copy from)
 * @param dst Destination heap (copy to)
 */
BOOL chunkheap_purify(Heap* src, Heap* dst) {
    // . . .
    // . . .

    // Fix object pointers that were changed
    stackframe_traverse(__chunkheap_fix_obj, src);
}
```

```
/**
 * @brief Repair object pointers after copying (stack traversal function)
 *
 * @param arg User argument (optional)
 * @param obj Heap object that was found
 * @param pp_obj Address of the pointer to the object
 */
static void __chunkheap_fix_obj(void* arg, Object* obj, void** pp_obj) {
    // . . .
    // . . .

    // Convert address in source heap -> address in destination heap
    const ChunkMapping* map = NULL;
    LINKLIST_FOREACH(&src->mappings, const ChunkMapping*,
        if (ELEM->from == obj) {
            map = ELEM;
            break;
        }
    );

    // Overwrite reference
    if (map != NULL) {
        *pp_obj = map->to;
        MJC_LOG("chunkheap fix %p -> %p\n", map->from, map->to);
    } else {
        MJC_LOG("chunkheap fix %p -> NONE\n", obj);
    }
}
```

Recap: Copying

```
root@debian:~# ./UseAfterCopyTest
[runtime/config.c:0063] setting heaptype to Chunk
[runtime/config.c:0044] setting gctype to Copying
[runtime/stackframe.c:0054] stackframe_push 0x40800670 (size:96)
[runtime/heap/heap.c:0061] try alloc (size:12)
[runtime/heap/heap.c:0092] alloc success 0x29040 (size:12), userptr: 0x29048
[runtime/stackframe.c:0054] stackframe_push 0x40800610 (size:92)
[runtime/stackframe.c:0071] stackframe_pop
[runtime/stackframe.c:0096] search stack frame: 0x40800670 (size:96)
[runtime/stackframe.c:0114]   local_num=1
[runtime/stackframe.c:0144]   ctx->locals[0 (align:0)] = 00029048
[runtime/stackframe.c:0181] traverse p_obj=0x29040 pp_obj=0x408006cc
[runtime/gc/copying_gc.c:0135] copying mark 0x29040
[runtime/stackframe.c:0185] traverse alloced block 0x29048
[runtime/heap/chunk_heap.c:0308] copying object at 0x29040
[runtime/heap/heap.c:0061] try alloc (size:12)
[runtime/heap/heap.c:0092] alloc success 0x390c8 (size:12), userptr: 0x390d0
[runtime/stackframe.c:0096] search stack frame: 0x40800670 (size:96)
[runtime/stackframe.c:0114]   local_num=1
[runtime/stackframe.c:0144]   ctx->locals[0 (align:0)] = 00029048
[runtime/stackframe.c:0181] traverse p_obj=0x29040 pp_obj=0x408006cc
[runtime/heap/chunk_heap.c:0367] chunkheap fix 0x29040 -> 0x390d0
[runtime/stackframe.c:0185] traverse alloced block 0x29048
[runtime/stackframe.c:0054] stackframe_push 0x40800610 (size:92)
[runtime/stackframe.c:0071] stackframe_pop
[runtime/stackframe.c:0071] stackframe_pop
12345
```

The value of the reference ‘dummy’ gets overwritten, allowing the **getValue()** call to return the correct result!

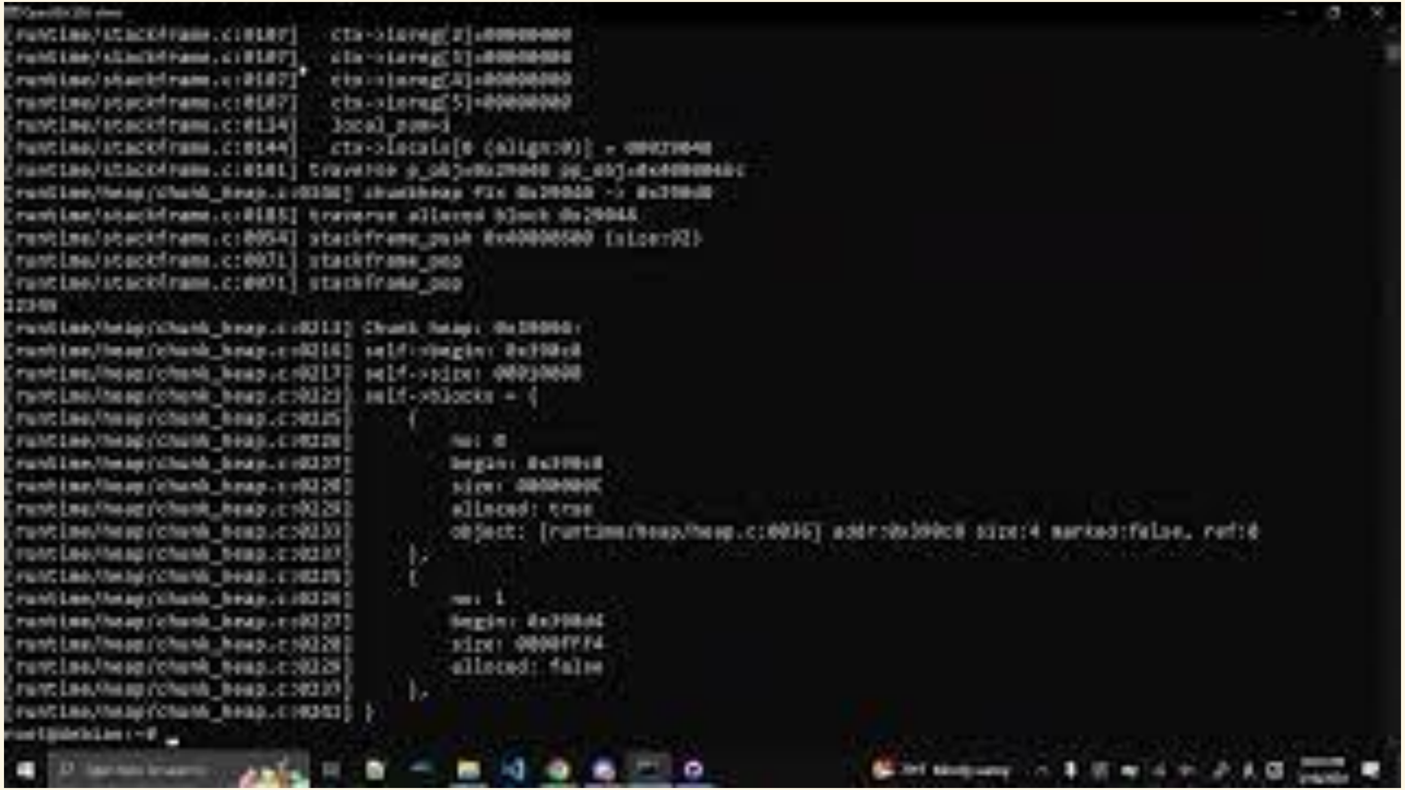
```java
class Main {
    public static void main(String[] a) {
        System.out.println(new
            UseAfterCopyTest().execute());
    }
}

class MyClass {
    int value;
    public void setValue(int x) { value = x; }
    public int getValue() { return value; }
}


class UseAfterCopyTest {
    public int execute() {
        MyClass dummy;
        dummy = new MyClass();
        dummy.setValue(12345);

        System.gc();
        return dummy.getValue();
    }
}
```

Demo: Copying Fix



```
[runtime/stackframe.c:0107]   ctx->ireg[2]=00000000
[runtime/stackframe.c:0107]   ctx->ireg[3]=00000000
[runtime/stackframe.c:0107]   ctx->ireg[4]=00000000
[runtime/stackframe.c:0107]   ctx->ireg[5]=00000000
[runtime/stackframe.c:0134]   local_num=1
[runtime/stackframe.c:0144]   ctx->locals[0 (align:0)] = 00020040
[runtime/stackframe.c:0181] traverse p_obj=0x20000 pp_obj=0x40000040c
[runtime/heap/chunk_heap.c:0338] chunkheap fix 0x20040 -> 0x3f00c8
[runtime/stackframe.c:0185] traverse allocted block 0x20040
[runtime/stackframe.c:0054] stackframe_push 0x40000500 (size:32)
[runtime/stackframe.c:0071] stackframe_pop
[runtime/stackframe.c:0071] stackframe_pop
12345
[runtime/heap/chunk_heap.c:0213] Chunk heap: 0x3f0000:
[runtime/heap/chunk_heap.c:0216] self->begin: 0x3f00c8
[runtime/heap/chunk_heap.c:0217] self->size: 00010000
[runtime/heap/chunk_heap.c:0223] self->blocks = {
[runtime/heap/chunk_heap.c:0225]     {
[runtime/heap/chunk_heap.c:0226]         no: 0
[runtime/heap/chunk_heap.c:0227]         begin: 0x3f00c8
[runtime/heap/chunk_heap.c:0228]         size: 0000000C
[runtime/heap/chunk_heap.c:0229]         allocd: true
[runtime/heap/chunk_heap.c:0230]         object: [runtime/heap/heap.c:0036] addr:0x3f00c8 size:4 marked:false, ref:0
[runtime/heap/chunk_heap.c:0237]     },
[runtime/heap/chunk_heap.c:0225]     {
[runtime/heap/chunk_heap.c:0226]         no: 1
[runtime/heap/chunk_heap.c:0227]         begin: 0x3f00d4
[runtime/heap/chunk_heap.c:0228]         size: 0000FFF4
[runtime/heap/chunk_heap.c:0229]         allocd: false
[runtime/heap/chunk_heap.c:0237]     },
[runtime/heap/chunk_heap.c:0243] }
root@debian:~# _
```

Overview: Generational

Objects on the heap are categorized into groups (known as “generations”).

- Newly created objects are inserted into generation zero
- As objects survive GC cycles, they move up

When garbage must be collected, the copying algorithm is performed to free memory in the youngest generation (generation zero).

- Objects that survive move up to generation one
- If the next generation is full and cannot hold more survivors, the copying GC propagates up to that generation, and so on

Popular generational GCs tend to use two to three generations, and MJC uses a two-generation system.

- .NET and Java garbage collectors use three generations

Overview: Generational

When we try to allocate memory, but none is available, perform:

1. Select generation (“target” generation) to act upon
    a. Unless otherwise specified, begin garbage collection with gen zero as the target
2. Mark Phase
    a. Mark all reachable heap objects in the target generation
3. Collect Phase
    a. *If the target generation is the last (oldest) generation:*
        i. Release unmarked objects in the target generation
    b. *Otherwise:*
        i. Copy all live allocations from the target gen heap to the next gen heap
            1. If the next gen heap is full, restart garbage collection with the next gen as the target
        ii. Release all allocations which exist in the target gen heap

Implementation: Generational

```
void generational_collect(GC* gc) {
    GenerationalGC* self = GC_DYNAMIC_CAST(gc, GenerationalGC);

    // Mark garbage in generation zero
    stackframe_traverse(__generational_mark_obj, NULL);

    // Copy all live allocations from the target gen heap to the next gen heap
    BOOL success = chunkheap_purify(curr_heap, self->gen_one);
    if (success) {
        // Release all allocations which exist in the target gen heap
        __generational_sweep(gc);
        return;
    }

    // Make new room in generation one
    Heap* old_curr_heap = curr_heap;
    curr_heap = self->gen_one;
    {
        stackframe_traverse(__generational_mark_obj, NULL);
        __generational_sweep(gc);
    }
    curr_heap = old_curr_heap;
}
```

NORTHROP GRUMMAN
ENGINEERING & SCIENCE
STUDENT DESIGN SHOWCASE
FLORIDA TECH

Heap Heap Hooray: Reinventing the Garbage Collector

Trevor Schiff, Tyler Gutowski

Faculty Advisor(s): Ryan Stansifer, Dept. of Electrical Engineering and Computer Science,
Florida Institute of Technology

Motivation

Research different garbage collection algorithms.

Algorithms

Reference Counting

- Involves counting number of references for each object.
- Suitable for scenarios with simple object lifetimes but suffers from cycles and overhead.

Mark-Sweep

- Involves marking reachable objects and sweeping away unreachable ones.
- More efficient than reference counting but can lead to fragmentation.

Copying

- Involves managing two independent heaps.
- During garbage collection, live allocations are copied to a new heap, defragmenting it in the process.
- Requires a minimal heap system for better control over memory allocation.

Generational

- Involves managing multiple independent heaps.
- Based on the idea that most objects die young.
- Separating objects based on age.

Design

Integration with MiniJava Compiler

- Uses Andrew Appel's compiler support library
- Integration with C runtime for memory management

Heap Management

- Minimal wrapper over standard memory allocations for more control.
- Required for copying, generational, and other types of algorithms for defragmentation.

Compiler Flags

- Designed user-friendly methods for setting configurations.

Testing

Test Suites

- Writing test suites to evaluate different garbage collection methods.
- Testing across various programs to determine what types of algorithms work for the varius garbage collection methods.

Gathering Metrics

- Difficult to quantify some data, as wall-clock speed isn't valuable.
- Key metrics, including the number of machine instructions, memory usage, stop-the-world time, and execution time.

Test Results

Conclusions

Research different garbage collection algorithms.

Milestone 6 Goals

- Finish generational GC implementation
  - Write and execute test cases with thorough coverage
- Conduct evaluation and analyze results
  - Compare implementations
  - Gather metrics and meaningful conclusions
- Create user/developer manual
  - Compiler usage for users
  - Implementation details for developers
- Create demo video