Solution by Xiao Jia

Homework 5

Problem 1. (20 points) In lecture *Going Imperative*, the language is extended with exceptions. You can raise an exception by using `raise e` and trap it by using the `try...with...` syntax. In this problem, you are required to define the syntax and the semantics (including evaluation rules and typing rules) of `try` e `catch` e_1 `finally` e_2 in a way that is similar to how `finally` works in Java. You can reuse some extensions such as sequence (e_1`;`e_2).

Solution.

(a) Syntax:

```
e ::= ...  | try e catch e1 finally e2
```

(b) Semantics:

```
                                    e => e'
(E-TrySearch) ------------------------------------------------------
              try e catch e1 finally e2 => try e' catch e1 finally e2

(E-TryValue)  try v catch e1 finally e2 => (v;e2)

(E-TryRaise)  try raise v catch e1 finally e2 => (e1 v;e2)

              G |- e : t      G |- e1 : t_exn->t'      G |- e2 : t''
(T-Try)       ------------------------------------------------------
                        G |- try e catch e1 finally e2 : t''
```

□

Problem 2. (60 points) Implement three GC algorithms in Java. A project skeleton is provided at `https://github.com/stfairy/gccp`. You can download the skeleton from GitHub and import it into Eclipse. You can ONLY modify the following three classes in package `gc`:

(a) `RefCountHeap` (for reference counting)

(b) `MarkSweepHeap` (for mark and sweep)

(c) `CopyCollectHeap` (for copy collection)

Solution.

Listing 1: RefCountHeap

```
1  package gc;
2
3  /**
4   * A reference-counting heap.
5   */
6  public class RefCountHeap extends Heap {
7    private static final int SIZE = -1;
8    private static final int COUNTER = -2;
9
10   public RefCountHeap(int size) {
11     super(size);
12   }
13
14   public void endScope() {
15     for (Var v : currentScope())
16       if (!v.isNull())
17         decreaseCounter(v.addr);
18     super.endScope();
19   }
20
21   /**
22    * Allocate memory with 2 extra slots, one for the object size, the
       other
23    * for the reference counter.
24    */
25   public void allocate(Var v, int size) throws InsufficientMemory {
26     super.allocate(v, size + 2);
27     v.addr += 2;
28     data[v.addr + SIZE] = size;
29     data[v.addr + COUNTER] = 1;
30   }
31
32   public void assign(Var v1, Var v2) {
33     if (!v1.isNull())
34       decreaseCounter(v1.addr);
35
36     super.assign(v1, v2);
37
38     if (v1.addr != -1)
39       increaseCounter(v1.addr);
40   }
41
42   public void readField(Var v1, Var v2, int fieldOffset) {
43     if (!v1.isNull())
44       decreaseCounter(v1.addr);
45
46     super.readField(v1, v2, fieldOffset);
47
48     if (v1.addr != -1)
49       increaseCounter(v1.addr);
```

```
  }

  public void writeField(Var v1, int fieldOffset, Var v2) {
    if (data[v1.addr + fieldOffset] != -1)
      decreaseCounter(data[v1.addr + fieldOffset]);

    super.writeField(v1, fieldOffset, v2);

    if (v2.addr != -1)
      increaseCounter(v2.addr);
  }

  private void increaseCounter(int addr) {
    data[addr + COUNTER]++;
  }

  private void decreaseCounter(int addr) {
    data[addr + COUNTER]--;
    if (data[addr + COUNTER] == 0) {
      int size = data[addr + SIZE];
      for (int i = 0; i < size; i++)
        if (data[addr + i] != -1)
          decreaseCounter(data[addr + i]);
      freelist.release(addr - 2, size + 2);
    }
  }
}
```

Listing 2: MarkSweepHeap

```
package gc;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class MarkSweepHeap extends Heap {
  private static final int SIZE = -1;
  private static final int MARKER = -2;

  private int markTag = 0;
  private Set<Integer> allocatedObjectAddresses = new HashSet<Integer>();

  public MarkSweepHeap(int size) {
    super(size);
  }

  public void allocate(Var v, int size) throws InsufficientMemory {
    allocate_(v, size);
    allocatedObjectAddresses.add(v.addr);
  }

```

```
23    private void allocate_(Var v, int size) throws InsufficientMemory {
24      try {
25        allocateObject(v, size);
26      } catch (InsufficientMemory e) {
27        markAndSweep();
28        allocateObject(v, size);
29      }
30    }
31
32    /**
33     * Allocate memory with 2 extra slots, one for the object size, the
       other
34     * for the marker.
35     */
36    private void allocateObject(Var v, int size) throws InsufficientMemory {
37      super.allocate(v, size + 2);
38      v.addr += 2;
39      data[v.addr + SIZE] = size;
40      data[v.addr + MARKER] = markTag;
41    }
42
43    private void markAndSweep() {
44      markTag++;
45
46      for (Var v : reachable)
47        if (!v.isNull())
48          mark(v.addr);
49
50      Iterator<Integer> it = allocatedObjectAddresses.iterator();
51      while (it.hasNext())
52        if (sweep(it.next()))
53          it.remove();
54    }
55
56    private void mark(int addr) {
57      if (data[addr + MARKER] == markTag)
58        return;
59      data[addr + MARKER] = markTag;
60      for (int i = 0; i < data[addr + SIZE]; i++)
61        if (data[addr + i] != -1)
62          mark(data[addr + i]);
63    }
64
65    private boolean sweep(int addr) {
66      if (data[addr + MARKER] == markTag)
67        return false;
68      int size = data[addr + SIZE];
69      freelist.release(addr - 2, size + 2);
70      return true;
71    }
72  }
```

Listing 3: CopyCollectHeap

```
1  package gc;
2
3  import java.util.Arrays;
4
5  /**
6   * For simplicity, implement Fenichel's algorithm instead of Cheney's
      algorithm.
7   *
8   * Semi-space garbage collection [Fenichel, 1969] is a copying algorithm,
      which
9   * means that reachable objects are relocated from one address to another
      during
10  * a collection. Available memory is divided into two equal-size regions
      called
11  * "from-space" and "to-space".
12  *
13  * Allocation is simply a matter of keeping a pointer into to-space which
      is
14  * incremented by the amount of memory requested for each allocation (that
       is,
15  * memory is allocated sequentially out of to-space). When there is
      insufficient
16  * space in to-space to fulfill an allocation, a collection is performed.
17  *
18  * A collection consists of swapping the roles of the regions, and copying
       the
19  * live objects from from-space to to-space, leaving a block of free space
20  * (corresponding to the memory used by all unreachable objects) at the
      end of
21  * the to-space.
22  *
23  * Since objects are moved during a collection, the addresses of all
      references
24  * must be updated. This is done by storing a "forwarding address" for an
      object
25  * when it is copied out of from-space. Like the mark-bit, this forwarding
26  * address can be thought of as an additional field of the object, but is
27  * usually implemented by temporarily repurposing some space from the
      object.
28  *
29  * The primary benefits of semi-space collection over mark-sweep are that
      the
30  * allocation costs are extremely low (no need to maintain and search the
      free
31  * list), and fragmentation is avoided.
32  */
33 public class CopyCollectHeap extends Heap {
34   private static final int SIZE = -1;
35   private static final int FORWARD = -2;
36
```

```
37    private int toSpace;
38    private int fromSpace;
39    private int allocPtr;
40
41    /**
42     * Though the super constructor is invoked and the free list is
       initialized,
43     * the free list is not used in the implementation of this copy
       collector.
44     */
45    public CopyCollectHeap(int size) {
46      super(size);
47      toSpace = 0;
48      fromSpace = size / 2;
49      allocPtr = toSpace;
50    }
51
52    public void allocate(Var v, int size) throws InsufficientMemory {
53      if (allocPtr + size + 2 > toSpace + data.length / 2)
54        collect();
55      if (allocPtr + size + 2 > toSpace + data.length / 2)
56        throw new InsufficientMemory();
57      v.addr = allocPtr + 2;
58      Arrays.fill(data, v.addr, v.addr + size, -1);
59      data[v.addr + SIZE] = size;
60      data[v.addr + FORWARD] = -1;
61      allocPtr += size + 2;
62    }
63
64    private void collect() {
65      int swap = fromSpace;
66      fromSpace = toSpace;
67      toSpace = swap;
68
69      allocPtr = toSpace;
70
71      for (Var v : reachable)
72        if (!v.isNull())
73          v.addr = copy(v.addr);
74    }
75
76    private int copy(int addr) {
77      int forwardAddr = data[addr + FORWARD];
78      if (forwardAddr == -1) {
79        int size = data[addr + SIZE];
80        forwardAddr = allocPtr + 2;
81        allocPtr += size + 2;
82        System.arraycopy(data, addr, data, forwardAddr, size + 1);
83        data[forwardAddr + SIZE] = size;
84        data[forwardAddr + FORWARD] = -1;
85        data[addr + FORWARD] = forwardAddr;
```

```
86        for (int i = 0; i < size; i++)
87          if (data[forwardAddr + i] != -1)
88            data[forwardAddr + i] = copy(data[forwardAddr + i]);
89      }
90      return forwardAddr;
91    }
92  }
```

□

Problem 3. (20 points) Implement a generational garbage collection algorithm.

Solution. Below is an article on Generational GC in .NET. It is chosen because it is the most well explained one that I can find on the Internet. If you are interested in Java or other systems, you can get a better understanding after reading this article.

Back To Basics: Generational Garbage Collection

Mark-sweep garbage collection introduces very large system pauses when the entire heap is marked and swept. One of the primary optimization employed to solve this issue is employing generational garbage collection. This optimization is based on the following observations

1. Most objects die young
2. Over 90% garbage collected in a GC is newly created post the previous GC cycle
3. If an object survives a GC cycle the chances of it becoming garbage in the short term is low and hence the GC wastes time marking it again and again in each cycle

The optimization based on the above observations is to segregate objects by age into multiple generations and collect each with different frequencies.

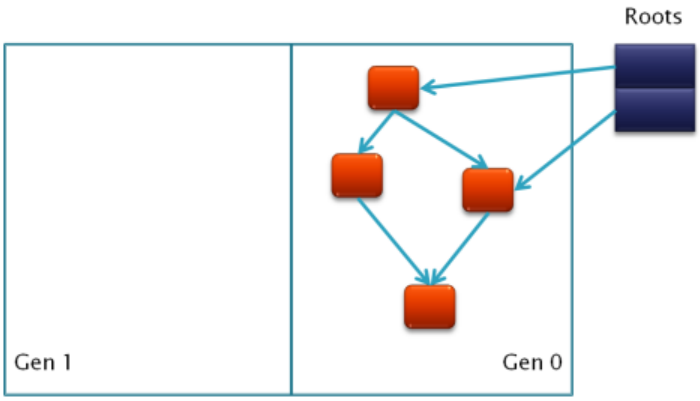
Detailed algorithm

The objects can be segregated into age based generations in different ways, e.g. by time of creation. However one common way is to consider a newly created object to be in Generation 0 (Gen0) and then if it is not collected by a cycle of garbage collection then it is promoted to the next higher generation, Gen1. Similarly if an object in Gen1 survives a GC then that gets promoted to Gen2.

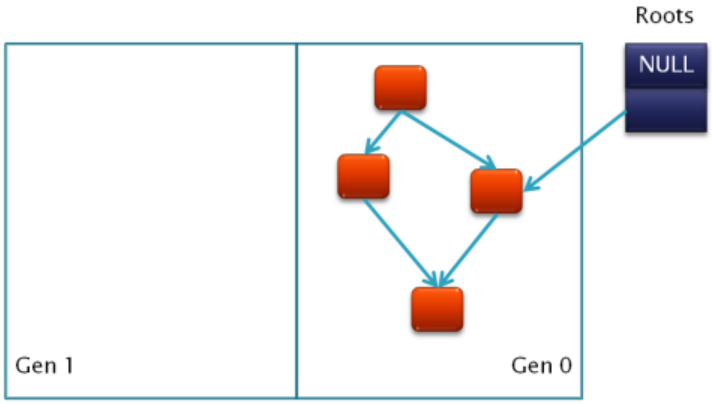
Lower generations are collected more often. This ensures lower system pauses. The higher generation collection is triggered fewer times.

How many generations are employed, varies from system to system. In .NET 3 generations are used. Here for simplicity we will consider a 2 generation system but the concepts are easily extended to more than 2.

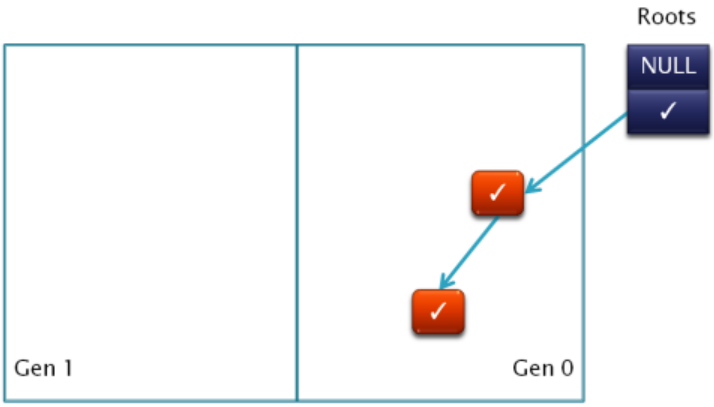
Let us consider that the memory is divided into two contiguous blocks, one for Gen1 and the other for Gen0. At start memory is allocated only from Gen0 area as follows



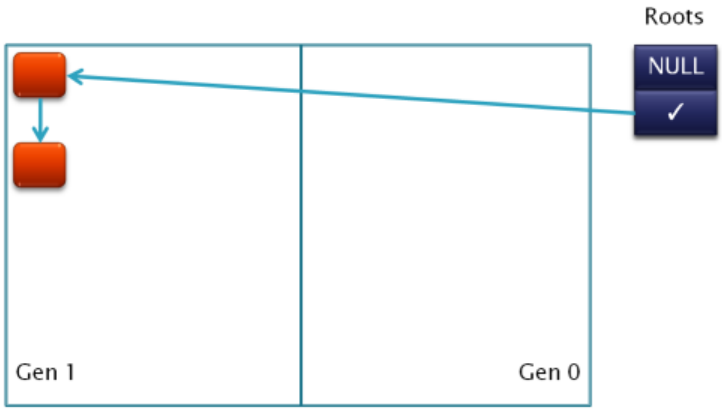
So we have 4 objects in Gen0. Now one of the references is released



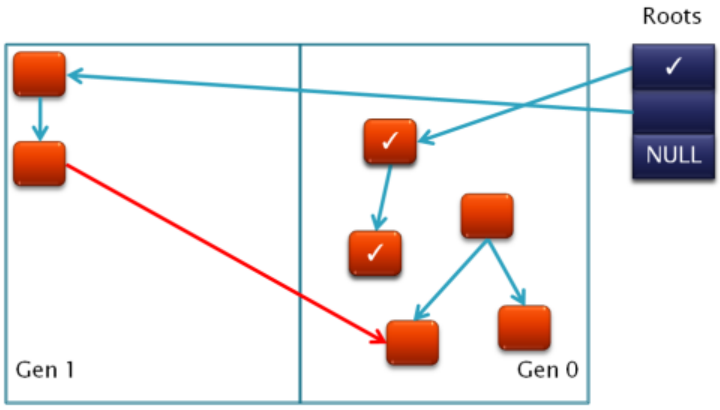
Now if GC is fired it will use mark and sweep on Gen0 objects and cleanup the two objects that are not reachable. So the final state after cleaning up is



The two surviving objects are then promoted to Gen1. Promotion includes copying the two objects to Gen1 area and then updating the references to them



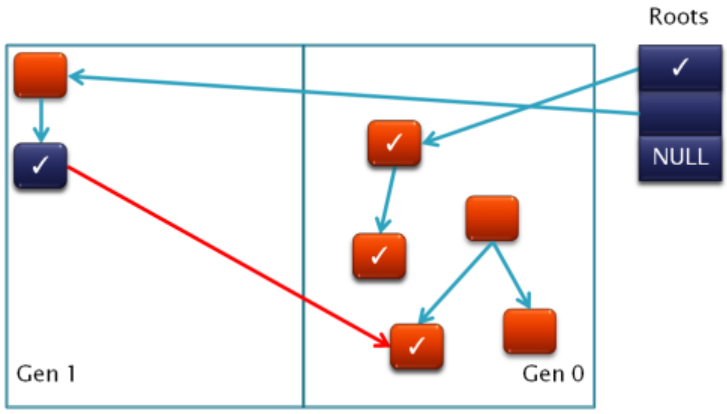
Now assume a whole bunch of allocation/de-allocation has happened. Since new allocations are in Gen0 the memory layout looks like



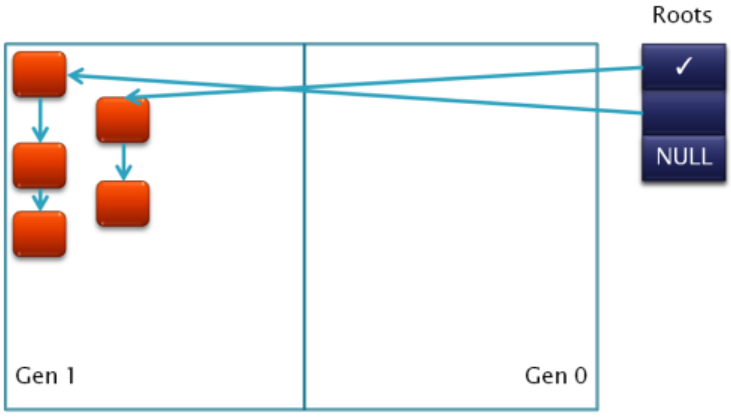
The whole purpose of segregating into generations is to reduce the number of objects to inspect for marking. So the first root is used for marking as it points to a Gen0 object. While using the second root the moment the marker sees that the reference is into a Gen1 object it does not follow the reference, speeding up marking process.

Now if we only consider the Gen0 objects for marking then we only mark the objects indicated by ✓. The marking algorithm will fail to locate the Gen1 to Gen0 references (shown in red) and some object marking will be left out leading to dangling pointers.

One of the way to handle this is to somehow record all references from Gen1 to Gen0 (way to do that is in the next section) and then use these objects as new roots for the marking phase. If we use this method then we get a new set of marked objects as follows



This now gives the full set of marked objects. Post another GC and promotion of surviving objects to higher generation we get



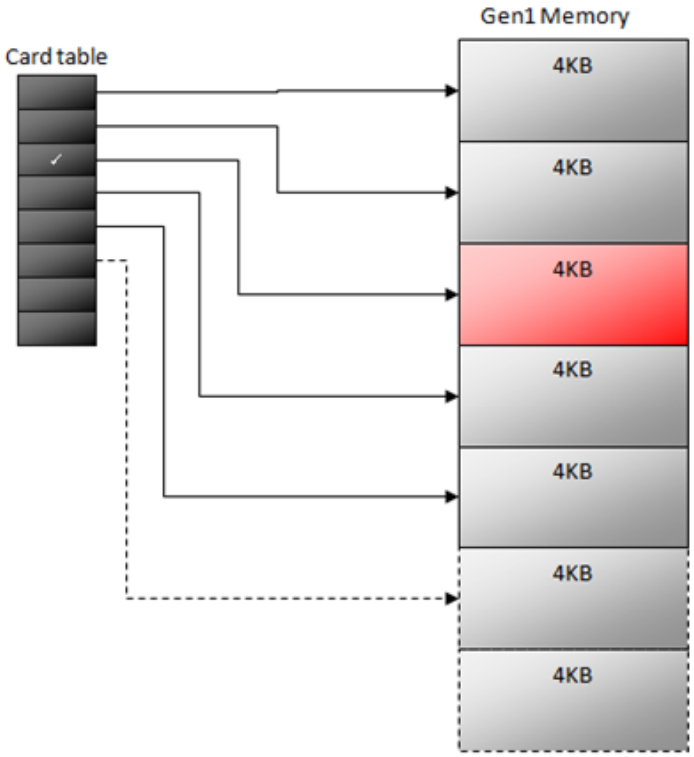
At this point the next cycle as above resumes

Tracking higher to lower generation references

In general applications there are very few (some studies show $< 1\%$ of all references) of these type of references. However, they all need to be recorded. There are two general approached of doing this

Write barrier + card-table

First a table called a card table is created. This is essentially an array of bits. Each bit indicates if a given range of memory is dirty (contains a write to a lower generation object). E.g. we can use a single bit to mark a 4KB block.



Whenever an reference assignment is made in user code, instead of directly doing the assignment it is redirected to a small thunk (incase .NET the JITter does this). The thunk compares the assignees address to that of the Gen1 memory range. If the range falls within, then the thunk updates the corresponding bit in the card table to indicate that the range which the bit covers is now dirty (shown as red).

First marking uses only Gen0 objects. Once this is over it inspects the card table to locate dirty blocks. Then it considers every object in that dirty block to be new roots and marks objects using it.

As you can see that the 4KB block is just an optimization to reduce the size of the card table. If we increase the granularity to be per object then we can save marking time by having to consider only one object (in contrast to all in 4KB range) but our card table size will also significantly increase.

One of the flip sides is that the thunk makes reference assignment slower.

HW support

Hardware support also uses card table but instead of using thunk it simply uses special features exposed by the HW+OS for notification of dirty writes. E.g. it can use the Win32 API `GetWriteWatch` to get the list of pages where write happened and use that information to get the card table entries. □