PLDI Practical 2

150008859

26/11/2018

1 Introduction

In this practical we implemented a generational garbage collector.

2 Usage

The project is a standard eclipse Java project, the main method can be found in "pldi.heap.Main".

The project can be run by performing the following command in the project directory to produce the trace in the "case study" section listed far below.

```
java -cp bin pldi.heap.Main
```

Alternatively, the project can be opened in eclipse.

3 Tasks Accomplished

1. A generational garbage collector with mark and sweep collection between generations and post collection promotion.
2. A thorough case study of an example heap.
3. An investigation in the time complexity of my generational collector.

4 Design

4.1 Memory

The code for this can be found in **"pldi.heap.Memory"**.

A Memory object compartmentalises a list of generation objects. The generations are invisible to system. Each generation represents a portion of contigious memory. The memory object provides an interface to this contigious region of memory via read and write operations.

4.2 Generations

The code for this can be found in **"pldi.heap.Generation"**.

Each generation possesses a region of memory modelled by an array of integers, a base address for that portion of contigious memory and the size of that array.

Each generation functions as a memory allocator and can allocate a "Chunk" of memory. Memory allocation allocation is non-trivial and is managed by FreeLists and supports splitting and coallescing.

Each generation possesses an allocation method which accepts a size and returns an address to the allocated memory. The memory object can be used to read and write to and from this address. In addition, each generation also possesses a free method which accepts an address and releases that memory for reallocation.

An arbitrary number of generations can be set, the user must specify the size of each generation.

4.3 RunTimeObjects

The code for this can be found in the **"pldi.objects"** package.

A RunTimeObject can allocate itself in order to obtain an address, pack (serialize) and unpack (deserialize) itself to and from memory given it's address in memory solely through use of the read and write operations. This is done through knowledge of the tag. A RunTimeObject once deserialized, can provide the list of pointers it contains (e.g. a CON object returns a list containing two pointers, a CTR object returns a list of all the pointers it stores) for use in the mark and sweep collection.

A weak pointer does not emit any pointers for the mark and sweep algorithm as it does not own the pointer so it has no stakes on its reachability.

4.4 Index

The code for this can be found in **"pldi.heap.Index"**.

An interesting design decision I took was storing indexes in RunTimeObjects as opposed to pointers. A table called the "index" maps indexes to pointers.

The advantage of this design decision is claimed in promotion. When a survivor is promoted, its address changes. An index allows us to simply update the pointer for that index in the table without having to scavenge for Objects and updating pointers in memory. The index stored in any RunTimeObject remains unchanged.

However, a drawback to this system would be having to perform an access to the table for each dereference in the actual system that uses the RunTimeObjects.

4.5 The RememberedSet

The RememberedSet stores pointers from RunTimeObjects in older generations to younger generations to prevent RunTimeObjects from being incorrectly collected in younger generations, they are added to the root set during the marking phase.

4.6 Garbage Collection

The code for this can be found in **"pldi.collect.Collector"**.

The algorithm can be summarised as the following:

The algorithm is provided a list of roots and a threshold generation to mark and collect up to. For example, if the first generation is given, a minor collection occurs. On the otherhand, if the oldest generation is supplied, then a major collection happens.

4.6.1 The marking phase

a) Perform a depth-first search with the garbage-collection roots and the remembered set as the entry points, (i.e. a list of pointers).

b) If a pointer has been visited, skip.

c) If a pointer points to a memory address in a higher generation, skip. However, lower generations are not skipped.

d) Otherwise, if the pointer has not been visited mark that memory for collection and add the pointer to the visited set.

e) Deserialize the memory into a RunTimeObject that the pointer points to and visit the pointers stored by that RunTimeObject.

A set is used to store marked addresses. Note that a pointer is retrieved by using the index stored in the RunTimeObject.

4.6.2 The sweeping phase

If a generation younger than the one being contains unmarked pointers, then perform the free operation on those addresses. This leaves behind survivors in each of the younger generations.

4.6.3 The promotion phase

Survivors are promoted to a higher generation if it exists.

This is performed by allocating memory in the higher generation for each survivor and copying it across then freeing the memory in the previous generation.

For each promoted survivor, the index is updated to map the index to the new memory address.

4.6.4 The unmarking phase

Empty the set used for storing visited addresses.

5 A Testing Case Study

In this section, we test the garbage collector by performing a case study on an example heap. Results have been collected via a trace produced by the program.

The initial heap setup is as follows:

5.1 The initial state

This is an isolated object which is unreachable. Perhaps it was an item removed from the list “a”.

This item is not present in the stack.

```
b = (ctr int bul)
```

The following items are members of the stack, i.e. the root set.

This is a list of constructors storing an integer and a boolean.

```
a = (Cons (ctr int bul) (Cons (ctr int bul) 'nul))
```

This is a weak pointer to the removed object.

```
b = (WPT b)
```

This is a constructor with an indirection to a closure named “f” which has an integer.

```
c = (Ctr (Ind (Fcn 'f' Int))
```

Finally, one indirection to an Integer.

d = (Ind (Int))

5.2 A minor collection a)

Memory in the following traces are expressed as indexes against RTOs.

Before the promotion the memory looks like this.

```
Generations:
Gen 0 with Base Addr 0 of Size 512 contains:
    0:(INT: 42)
    13:(INT: 0)
    1:(BUL: false)
    14:(FUN f n=1 :p=13 )
    2:(CTR n=2 :p=0 1 )
    15:(IND: 14)
    3:(WPT: 2)
    16:(CTR n=1 :p=15 )
    4:(INT: 0)
    5:(BUL: false)
    17:(INT: 0)
    6:(CTR n=2 :p=4 5 )
    18:(IND: 17)
    7:(NUL)
    8:(CON  6, 7)
    9:(INT: 0)
    10:(BUL: false)
    11:(CTR n=2 :p=9 10 )
    12:(CON  11, 8)
Gen 1 with Base Addr 512 of Size 1024 contains:
    Nothing
Gen 2 with Base Addr 1536 of Size 2048 contains:
    Nothing
```

The collector then runs and reports it has collected the following.

```
Collected 0:(INT: 42)
Collected 1:(BUL: false)
Collected 2:(CTR n=2 :p=0 1 )
```

After collection the memory looks like this.

```
Generations:
Gen 0 with Base Addr 0 of Size 512 contains:
    Nothing
Gen 1 with Base Addr 512 of Size 1024 contains:
    13:(INT: 0)
    11:(CTR n=2 :p=9 10 )
    14:(FUN f n=1 :p=13 )
    12:(CON  11, 8)
    15:(IND: 14)
    3:(WPT: 2)
    16:(CTR n=1 :p=15 )
    4:(INT: 0)
    5:(BUL: false)
    17:(INT: 0)
    6:(CTR n=2 :p=4 5 )
    18:(IND: 17)
    7:(NUL)
    8:(CON  6, 7)
    9:(INT: 0)
    10:(BUL: false)
Gen 2 with Base Addr 1536 of Size 2048 contains:
    Nothing
```

The first collection releases b including both of its children. In addition, the survivors are promoted from gen 0 to gen 1.

5.3 A substantial collection

We can allocate some more RunTimeObjects and perform a substantial collection, in this case this would be a collection up to generation 1. I modify the constructor with an indirection to a closure to point to a closer named “g” instead of f.

```
c = (Ctr (Ind (Fcn 'g' Int))
```

The previous closure “f” is collected.

In addition to demonstrate the remembered set, I modify the indirection “d” to point from the 1st generation to an integer in the 0th generation with the value of 99. The aim is to have the garbage collector not accidently collect the integer with the value 99.

```
Generations:
Gen 0 with Base Addr 0 of Size 512 contains:
    0:(INT: 0)
    1:(FUN g n=1 :p=0 )
    19:(INT: 99)
Gen 1 with Base Addr 512 of Size 1024 contains:
    13:(INT: 0)
    11:(CTR n=2 :p=9 10 )
    14:(FUN f n=1 :p=13 )
    12:(CON  11, 8)
    15:(IND: 1)
    3:(WPT: 2)
    16:(CTR n=1 :p=15 )
    4:(INT: 0)
    5:(BUL: false)
    17:(INT: 0)
    6:(CTR n=2 :p=4 5 )
    18:(IND: 17)
    7:(NUL)
    8:(CON  6, 7)
    9:(INT: 0)
    10:(BUL: false)
Gen 2 with Base Addr 1536 of Size 2048 contains:
    Nothing


Collected 13:(INT: 0)
Collected 14:(FUN f n=1 :p=13 )

Generations:
Gen 0 with Base Addr 0 of Size 512 contains:
    Nothing
Gen 1 with Base Addr 512 of Size 1024 contains:
    0:(INT: 0)
    1:(FUN g n=1 :p=0 )
    19:(INT: 99)
Gen 2 with Base Addr 1536 of Size 2048 contains:
    11:(CTR n=2 :p=9 10 )
    10:(BUL: false)
    12:(CON  11, 8)
    15:(IND: 1)
    3:(WPT: 2)
    16:(CTR n=1 :p=15 )
    4:(INT: 0)
    5:(BUL: false)
    17:(INT: 0)
    6:(CTR n=2 :p=4 5 )
    18:(IND: 17)
    7:(NUL)
    8:(CON  6, 7)
    9:(INT: 0)
```

Naturally, the old Integer is collected. the RTO's are all promoted by 1 generation and our remembered set allows us to not accidently collect the new Integer that is pointed to by an Indirection from a higher generation. Ordinarily, since we do not descend into objects in higher generations during the marking phase, it is not possible to mark this RTO. However, the remembered set stores this pointer and allows us to mark it as reachable.

5.4 A major collection

Finally, we can cause a major collection, a collection up to the final generation.

The only change I make here is removing all the pointers from the root set with the exception of the list.

```
Collected 0:(INT: 0)
Collected 1:(FUN g n=1 :p=0 )
Collected 19:(INT: 99)
Collected 15:(IND: 1)
Collected 3:(WPT: 2)
Collected 16:(CTR n=1 :p=15 )
Collected 17:(INT: 0)
Collected 18:(IND: 17)

Generations:
Gen 0 with Base Addr 0 of Size 512 contains:
    Nothing
Gen 1 with Base Addr 512 of Size 1024 contains:
    Nothing
Gen 2 with Base Addr 1536 of Size 2048 contains:
    11:(CTR n=2 :p=9 10 )
    10:(BUL: false)
    12:(CON  11, 8)
    4:(INT: 0)
    5:(BUL: false)
    6:(CTR n=2 :p=4 5 )
    7:(NUL)
    8:(CON  6, 7)
    9:(INT: 0)
```

The trace shows that only elements of the list "a" remain, since there is no generation for the survivors in the highest generation to be promoted into, they remain there.

6 Complexity Analysis

I found it very difficult to devise a fair experiment to measure the time complexity. Hence, I shall hypothesize what I believe to be the time complexity. I believe the time complexity in my collector is proportional to the number of allocated objects. Regardless of the "depth" of the collection, in terms of a minor, substantial or major collection, the underlying algorithm used is still mark and sweep. In my case, the sweep portion of the algorithm visits only the objects which have been allocated. The mark and sweep algorithm is linear can be broken into two phases, the mark portion which is a linear time complexity depth-first search (restricted by generations) and the sweep portion which is also a linear time complexity operation. The time saving is done by limiting the generations which are searched, hence the time complexity is proportional to the number of allocated objects.