Lab 6: Garbage Collector

May 4, 2016

The goal of this lab is to develop a garbage collector. In the first part of the lab, you will be developing a memory allocator, using the best fit allocation algorithm. In the second part of the lab, you will be implementing mark-sweep algorithm and tri-color marking garbage collector.

1 Get the code

The base code used in the labs is in the directory */home/TDDA69/Labs/Lab6/src*. To get the code and start working on it, in your home directory:

```
cp -r /home/TDDA69/Labs/Lab6/src $HOME/Lab6
```

This will copy the skeleton for the *Lab6* assignments, you can now find them in the directory *$HOME/Lab6*. In the rest of the document and in other lab, we will refer to this directory as *dir_to_lab6*.

2 Memory structure

First you need to be able to store and represent values in memory.

2.1 Python bytearray

During this lab, you will be using a byte array as a block of memory. To create a byte array in python you can use the *bytearray* function. It takes an integer as an argument to select the size, for instance, the following create a 200-bytes array:

```
arr = bytearray(200)
```

You can access the values with the normal *[]* operator:

```
arr[5] = 124
print(arr[5])
```

2.2 Minimal type system

The strict minimum number of type that you need to support are:

- array of bytes (8bits integers)
- array of pointers, in this lab, we will use 32bits integers to represent pointers, the pointers correspond to an index in the memory block

Using a combination of those two types, you can represent any value, for instance:

- 32bits integers can be represented by a 4-elements array of bytes
- strings are stored in array of bytes
- dictionaries can be represented with an array of pointers (as shown in lecture 2)
- an object is an array of pointers, where the first field is a pointer to the class type, and the second one is a pointer to a dictionnary where the value members will be stored
- ...

Such a type system might not give the best performance but it is the simplest one.

2.3 Header

An object in memory is represented by a byte array, where the first 32bits correspond to the header, and the rest to the value.

The header should follow the format:

- bit 0: reserved for the garbage collection algorithm
- bit 1: indicate if a block of memory is used or not
- bit 2: indicate the type of the object, either array of bytes or array of pointers (0 means array of bytes, 1 means are of pointers)
- bit 3-31: the size of the memory block (in bytes)

The first task in the lab is to create a couple of helper function to manipulate the header of an object:

- *header_get_garbage_flag(heap, pointer)* return true or false, depending on if the garbage flag is set or not
- *header_set_garbage_flag(heap, pointer, value)* value is true or false, after a call to true, the flag should be set to 1
- *header_get_used_flag(heap, pointer)* return true or false, depending on if the object is used or not
- *header_set_used_flag(heap, pointer, value)* value is true or false, after a call to true, the flag should be set to 1
- *header_is_pointers_array(heap, pointer)* return true if the object is an array of pointers, false otherwise
- *header_mark_as_pointers_array(heap, pointer)* mark the object as an array of pointers
- *header_mark_as_bytes_array(heap, pointer)* mark the object as an array of bytes
- *header_get_size(heap, pointer* get the size (in bytes) of the object (without including the size of the header)

- *header_set_size(heap, pointer, size* set the size (in bytes) of the object (without including the size of the header)

Added A note on representation: `int.from_bytes` and `(int).to_bytes` might prove helpful. Note that the byte array

```
b'\x05\x00\x00\x00'
```

represents the integer 5, and has bit 31 set to 1.

As per usual, you can test your code using:

```
tdda69_lab6_tests dir_to_lab6 header_helpers
```

2.4 Data

In this lab, we can assume that the content of an array of bytes is random and we do not need to be concern by the actual values. We do need to be able to manipulate pointer-arrays, and for this purpose we need to be able to access some values:

- *pointer_array_count(heap, pointer)* return the number of elements in the array of pointers
- *pointer_array_get(heap, pointer, index)* return the pointer in the array at the given *index*
- *pointer_array_set(heap, pointer, index, value)* set the value (which is a pointer) in the array at the given *index*

```
tdda69_lab6_tests dir_to_lab6 pointer_array_helpers
```

3 Best-fit allocator

The best-fit allocator is a memory allocator that attempt to ballance between performance and reduced fragmentation.

You should implement a *heap* class with the following prototype:

```
class heap(object):
  # size: the size (in bytes) of the heap
  def __init__(self, size):
    pass
  # return the index to the begining of a block with size (in bytes)
  def allocate(self, size):
    pass
  # unallocate the memory at the given index
  def disallocate(self, size):
    pass
  # Return the current total free space
  def total_free_space(self):
    pass
  # Return the current total allocated memory
  def total_allocated_space(self):
    pass
```

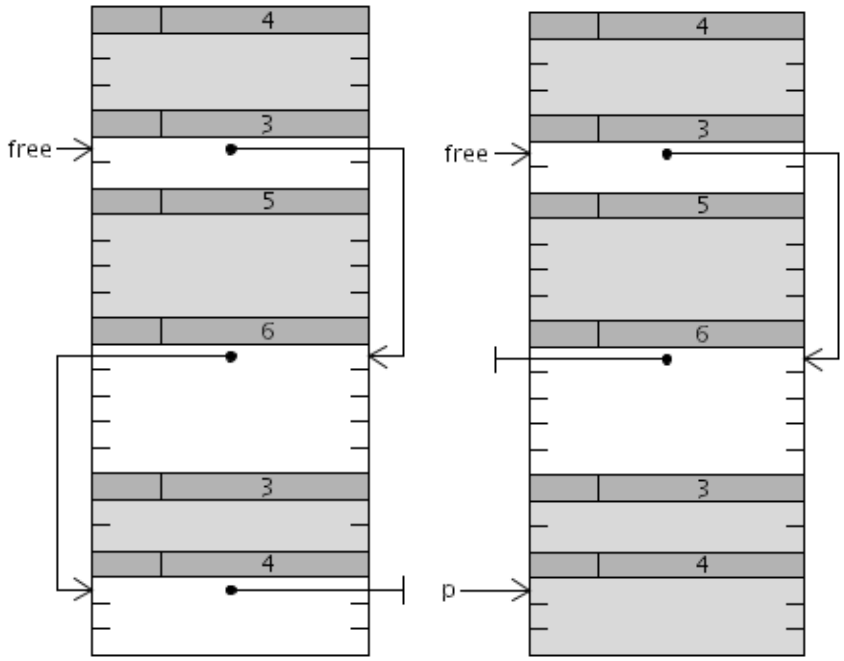

Figure 1: Allocation of a 3 bytes object in the heap with a best-fit allocator. On the left, the state of the heap before allocation, on the right, the state of the heap after allocation.

Remember that the header has a size of four bytes, so you need to allocate a byte array of slightly longer dimension.

```
tdda69_lab6_tests dir_to_lab6 best_fit_allocator
```

4 Mark-sweep garbage collector

The object at index 0 is assumed to be the root object. At the begining of the collection, all objects are *unmarked*, except for the root object.

The algorithm follow the steps:

1. Unmark all objects
2. Mark the root object
3. Mark all the objects directly reachable by root
4. Repeat *step 3* for all marked objects

Once all objects directly reachable from root are marked, the garbage collector can disallocate all the *unmarked* objects.

Implement a *mark_sweep_gc* class with the following prototype:

```
class mark_sweep(object):
  def __init__(self, heap):
    pass
  # This function should collect the memory in the heap
  def collect(self):
    pass
```

Test it with:

```
tdda69_lab6_tests dir_to_lab6 mark_sweep
```

5 tri-color marking garbage collector

The mark-sweep algorithm has several issues, the most important one is that it requires that the entire system is suspended for the duration of the garbage collection, which is problematic for real-time application or user interfaces. The other issues is that the entire heap memory is examined twice, one time for marking a second time for disallocation.

The tri-color garbage collector use three sets:

- The *white* set contains the list of objects that are candidates for disallocation
- The *black* set contains all the objects that have no connection to an object in the white set
- The *gray* set contains all the objects that are reachable from root and that might have a connection to objects in the white set

The algorithm follow the steps:

1. Initialise the *white* set with all objects except the root set
2. The *black* set starts empty
3. The *gray* set is initialised with the root object
4. Repeat the following until the *gray* set is empty:
    - Take an *object* from the *gray* set and move it to the *black* set
    - Move all the white objects that are directly reachable from *object* in the *gray* set

Implement a *tri_color_gc* class with the following prototype:

```
class tri_color(object):
  def __init__(self, heap):
    pass
  # This function should collect the memory in the heap
  def collect(self):
    pass
```

Test it with:

```
tdda69_lab6_tests dir_to_lab6 tri_color
```