Master M1 MOSIG – UGA

Principles of Operating Systems and Concurrent Programming

Memory Allocator - Lab2 - Report

Author:
Jit Chatterjee
Arthur Van Schendel

Submitted to:
Thomas Ropars
Renaud Lachaize

Contents

1 - Introduction

The purpose of this lab assignment was to design and implement a dynamic memory allocator, based on an approach known as “segregated memory pools”. Our program will not use the functions alloc(), malloc() or free() etc as built in functions that call those methods were given to us for the purpose of this assignment.

2 - Implemented features

Our program contains the following features:

> A memory allocator for fast pools
> A memory allocator for standard pool
> First-Fit allocation policy for standard pool
> A function that displays the state of the heap (visual representation of the state of the memory) for the Standard Pool.
> Memory Alignment of 8 bytes.

3 - Not Implemented features

> Best-Fit allocation policy for standard pool

4 - Design Description

For all blocks, we have 8 bytes that are used to store the size and the state of the block, 63 bit to store the size, and 1 bit flag to store the state (used or free) for a total of 64 bits -> 8 bytes. Then the free blocks have 2 pointers, that are used to point to the next and previous free block, as we need to keep track of the free block list. The 2 pointers are stored within the payload, as when a free block will be allocated, it will not need to store next and prev to free blocks as it’s now allocated so it will not need extra space for the 2 pointers.

> Fast pools:

For fast pools, as the blocks are always of a fixed size, in the init() function we create every block with a while loop: while the size of all blocks created is less than the size of the pool, create new blocks. This will lead us to a pool filled with free blocks of fixed size (sizes will depend of the pool we are dealing with). After this, we need to implement a function that will allocate all this free space. In this function, we first test if the size of the block we are currently on is big enough so that it can store the request size. Then, if it’s big enough, we allocate the block inside the first available free block (we are using the first fit policy, which is we select the first free block that is the correct size to fulfill the request).

In the free function of the fast pools, we just set the next free block to the current pool first free, and we then set the freed block to the new pool first free. We will be using the LIFO approach when freeing the blocks for the fast pools, meaning that everytime we free a block, we insert it at the start of the free block list (as a first free). This approach is suited for the fast pools as the blocks are all of the same size, and putting back a free block at first in the free block list will not change the address of any pointers to the next and/or prev, and another advantage is that the LIFO is fast an easy to implement.

> **Standard pool**

For the standard pool, it is a bit more complex. The init will just allocate a big space in the pool, one big block of size: pool size. The next and prev pointers are initialized to NULL.

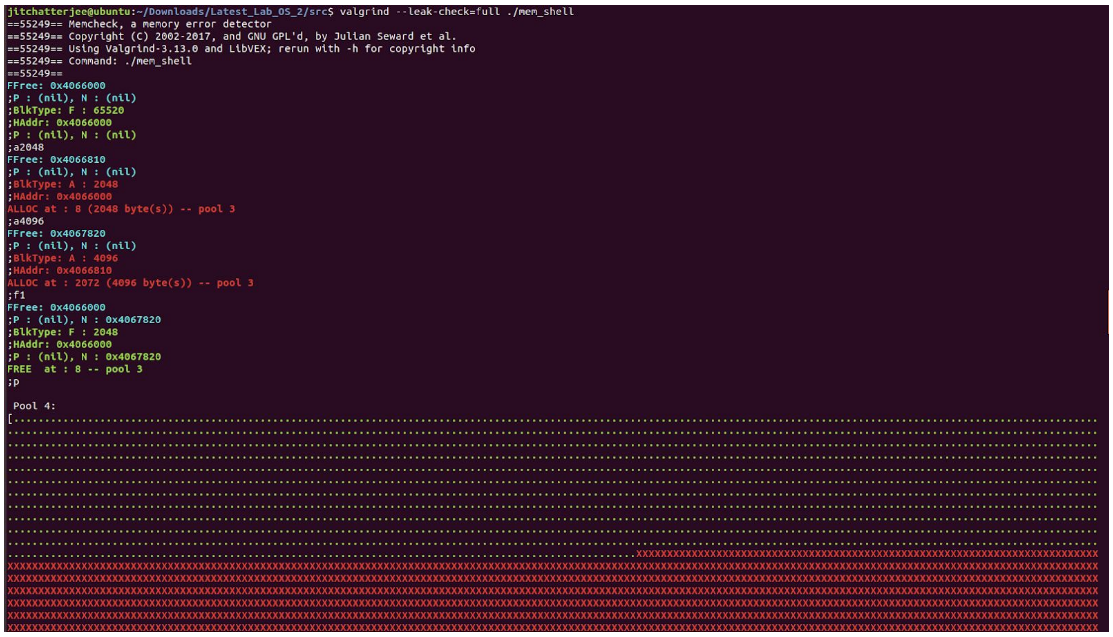
After the init, we had to implement the alloc function, that will, given a pool and size will allocate memory of the request size in a suited block. The suited block will be obtained after splitting the original big one block we had after the init. So in the standard pool, the block sizes are not going to be fixed, they will depend on the user's allocation requests.

For the free part, we will just be updating the content in the headers and footers of the block we are trying to free. We will be updating the state of the block, from used to free, and we will update the free block list, which is a doubled linked list. We will make sure that every case is taken into account, for example when the block we are freeing is before the first free, we need to update first free, and the free block list will change. There is an additional feature we need to implement in the free part, which is **coalescing blocks**. This is if we free a block, and that block is adjacent to another free block, we are going to coalesce them in order to avoid internal fragmentation. For coalescing we have 3 cases, left coalesce, right coalesce and both.

For this, we will check if blocks are adjacent, and are both free, if yes then we coalesce them and update the size and footer address of the new coalesced block.

Best Fit Policy : This was not implemented due to a lack of time, but if we had time we would've implemented it like this: when a alloc request is made, check the size of the request. Then, parse through all freed blocks, and compute the difference between the size of the free block and the size of the request. Then when you parsed through all free blocks, choose the free block where the computed difference is the smallest, this will be the best fit.

Print Heap State :



```
jitchatterjee@ubuntu:~/Downloads/Latest_Lab_OS_2/src$ valgrind --leak-check=full ./mem_shell
==55249== Memcheck, a memory error detector
==55249== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==55249== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==55249== Command: ./mem_shell
==55249==
FFree: 0x4066000
;P : (nil), N : (nil)
;BlkType: F : 65520
;HAddr: 0x4066000
;P : (nil), N : (nil)
;a2048
FFree: 0x4066810
;P : (nil), N : (nil)
;BlkType: A : 2048
;HAddr: 0x4066000
ALLOC at : 8 (2048 byte(s)) -- pool 3
;a4096
FFree: 0x4067820
;P : (nil), N : (nil)
;BlkType: A : 4096
;HAddr: 0x4066810
ALLOC at : 2072 (4096 byte(s)) -- pool 3
;f1
FFree: 0x4066000
;P : (nil), N : 0x4067820
;BlkType: F : 2048
;HAddr: 0x4066000
;P : (nil), N : 0x4067820
FREE  at : 8 -- pool 3
;p

 Pool 4:
[..............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
...............................................................................................................................................................................
.....................................................................................................xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

5 - Questions and Answers

“*Explain why a LIFO policy for free block allocation is interesting in the context of fast pools*”

> The LIFO (Last In First Out) policy is based on a data structure called stack, where you can either push (insert element in stack) or pop. The LIFO policy is when the last element inserted is the first one being popped. In our case, the free blocks when inserted into the free list, will be the first one to be “popped” (when alloc calls a free block to be allocated) or in other words it will be the first one to be allocated. The reason this policy is suited for fast pools is that all blocks in the fast pools have the same size, and hence we don’t need a specific order of blocks, hence we can just return the last free block inserted in the list to the alloc request. This will simplify the code for the free method in fast pools. This makes the pool work fast.

“*Describe the main principles of the algorithm you use to insert a freed block back into the free list, and to apply coalescing if need be. You should try to make best use of the additional metadata included in blocks (footer, doubly linked list)*”

> When an allocated block is freed, we need to re insert it into the free list. For this, we will take the address of the new freed block, and test if it is less than the pool first free. If yes, the freed block becomes the new first free and the pointers to the next and prev are updated (here the prev will be NULL). For other cases, the new freed block will just be inserted by increasing address, meaning that we will not put it at the top of the free list. We will insert it where it used to be when it was a free block, and update the next and prev accordingly.

6 - Additional Features and Test Scenarios

> Double Free check.

```
assert(!(cur_free == first_free || cur_free == last_free)); // double free check
```

> Passed all the test scenarios from Alloc1 to Alloc7.
> Fails Alloc8 test case.

7 - Feedback

We have learned a lot regarding different memory allocation techniques and use of pointers in C.