Machine Problem 3 Report

Lab members: Martin Fracker, Chris Findeisen

Date: March. 15, 2015

Introduction

In this week's lab, we focused on dynamic memory allocation, using a buddy algorithm originally documented by Donald Knuth in the Art of Computer Programming. We came up with a memory efficient solution that was also fast and simple to use(from an interface perspective). We then analysed it's running-time and memory consumption using the ackerman function.

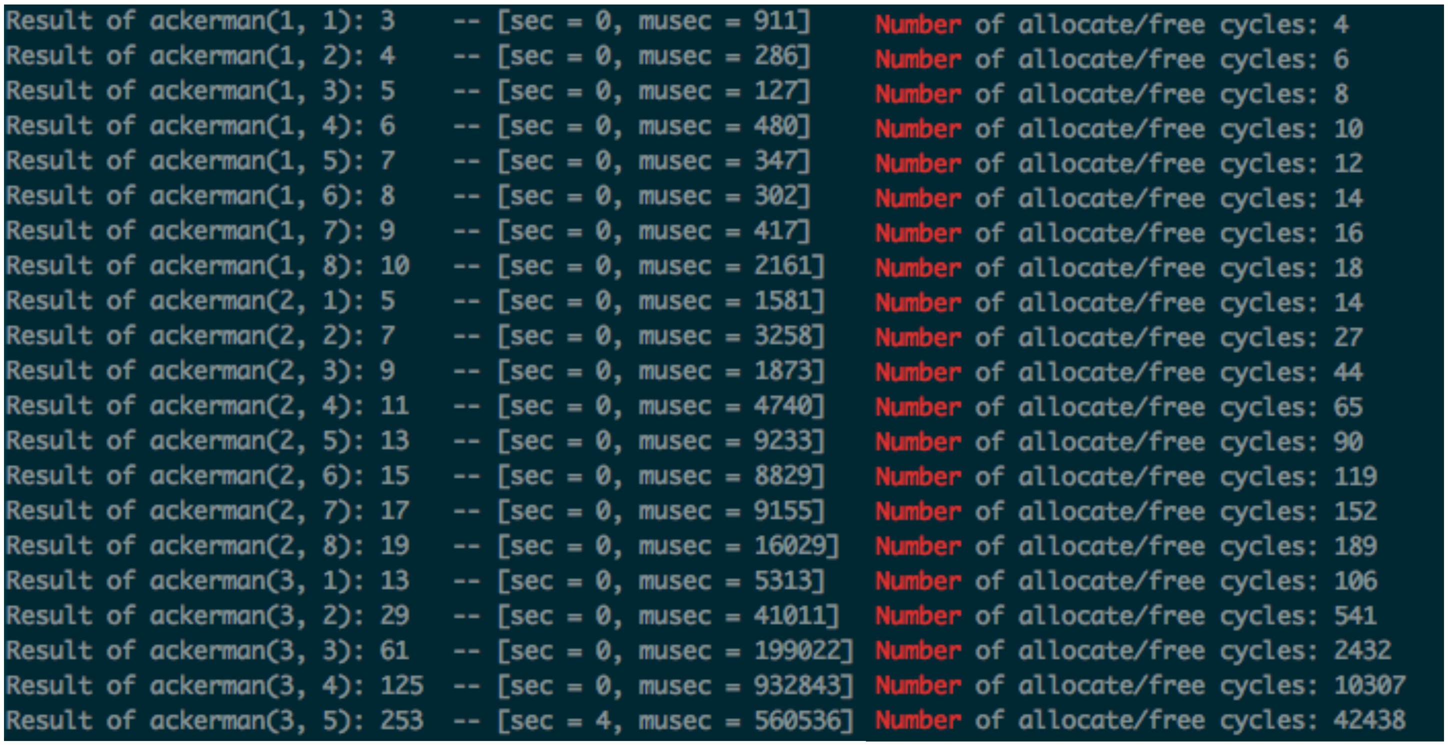
Procedures

We built a char_bitmap that would keep track of the free-lists, as well as the order of the block. This way, we'd have a predictable and constant overhead o = (total_request / basic_block). This less than any proposed(in the spec) implementation.

We built a data structure called memory_map that abstracted these details and provided the user with 4 simple functions— init_memory_map, get_block, release_block, and free_memory_map.

We then integrated with my_allocator and wrote our memtest.c function.

Results:



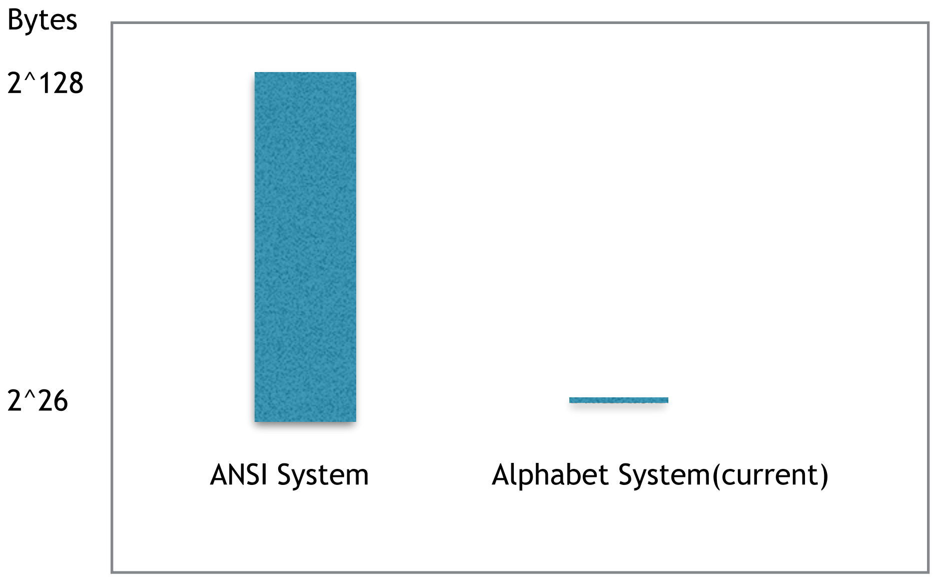
```
Result of ackerman(1, 1): 3    -- [sec = 0, musec = 911]    Number of allocate/free cycles: 4
Result of ackerman(1, 2): 4    -- [sec = 0, musec = 286]    Number of allocate/free cycles: 6
Result of ackerman(1, 3): 5    -- [sec = 0, musec = 127]    Number of allocate/free cycles: 8
Result of ackerman(1, 4): 6    -- [sec = 0, musec = 480]    Number of allocate/free cycles: 10
Result of ackerman(1, 5): 7    -- [sec = 0, musec = 347]    Number of allocate/free cycles: 12
Result of ackerman(1, 6): 8    -- [sec = 0, musec = 302]    Number of allocate/free cycles: 14
Result of ackerman(1, 7): 9    -- [sec = 0, musec = 417]    Number of allocate/free cycles: 16
Result of ackerman(1, 8): 10   -- [sec = 0, musec = 2161]   Number of allocate/free cycles: 18
Result of ackerman(2, 1): 5    -- [sec = 0, musec = 1581]   Number of allocate/free cycles: 14
Result of ackerman(2, 2): 7    -- [sec = 0, musec = 3258]   Number of allocate/free cycles: 27
Result of ackerman(2, 3): 9    -- [sec = 0, musec = 1873]   Number of allocate/free cycles: 44
Result of ackerman(2, 4): 11   -- [sec = 0, musec = 4740]   Number of allocate/free cycles: 65
Result of ackerman(2, 5): 13   -- [sec = 0, musec = 9233]   Number of allocate/free cycles: 90
Result of ackerman(2, 6): 15   -- [sec = 0, musec = 8829]   Number of allocate/free cycles: 119
Result of ackerman(2, 7): 17   -- [sec = 0, musec = 9155]   Number of allocate/free cycles: 152
Result of ackerman(2, 8): 19   -- [sec = 0, musec = 16029]  Number of allocate/free cycles: 189
Result of ackerman(3, 1): 13   -- [sec = 0, musec = 5313]   Number of allocate/free cycles: 106
Result of ackerman(3, 2): 29   -- [sec = 0, musec = 41011]  Number of allocate/free cycles: 541
Result of ackerman(3, 3): 61   -- [sec = 0, musec = 199022] Number of allocate/free cycles: 2432
Result of ackerman(3, 4): 125  -- [sec = 0, musec = 932843] Number of allocate/free cycles: 10307
Result of ackerman(3, 5): 253  -- [sec = 4, musec = 560536] Number of allocate/free cycles: 42438
```

Analysis:

Larger n and m values yielded big jumps in allocate/free cycles. However, the biggest bottleneck in our program seems to be caused by freeing bigger small blocks, since this would have to call the recursive “coalesce” function.

We’d like to allocate our memory_map struct on the stack. This would result in a big jump in performance because of spatial locality. This would cause fewer cache misses, and thus a speed increase.

Currently, we’re using a bitmap system that can support up to a memory allocation of 2^26 bytes. This is a pretty significant memory ceiling. The cause is our char bitmap, which is currently alphabet-based. We would like to modify it to use any ANSI value, which would increase it’s support to 2^128 bytes! This would be sufficient for a scalable memory bitmap.



Conclusion

Since the Ackerman function is highly recursive, the number of memory allocations (and thus the required time) dramatically increased especially with the increase of n.

We believe that we have a strong solution, since our char bitmap consumes less memory than a header/linked-list solution would. It's also reasonably efficient at finding free positions, O(n), releasing O(n), and getting blocks O(n). We have implemented efficient algorithms that make the average-case theta(log_2 (n)).