Supplemental assignment: Memory management

1DT056: Programming Embedded Systems
Uppsala University

March 20th, 2015

You can achieve a maximum number of 20 points in this assignment. 12 out of the 20 points are required to pass the assignment.

Assignments are to be solved by students individually. Several students handing in (literally) the same assignment solution will be considered as cheating, and reported accordingly.

Introduction and setup

In this exercise you will be implementing your own memory manager with malloc and free functions. In the `http://www.it.uu.se/edu/course/homepage/pins/vt15/labs/add_env.zip` you will find the following:

- *mem.c* - malloc/free implementation
  The the only file that you need to modify. Here you will put your own code. To assist you there are some definitions and the function *internalAlloc*() to request memory from the operating system (using *pvPortMalloc*()). Initialization code should be put in the function *initialize_mem*(), so it only runs once when the first call to *malloc*() occurs.
- *main.c* - test program
  If you want to try your own testcases, you may modify this file. The original test case in the file will be used to verify that your implementation works.

Some remarks:

- Your code should be clear and easy to follow.
- Names of variables, functions and structures should make sense.
- Document the code appropriately (overzealous documentation is also bad).
- Unreadable code will be treated as incorrect.

Exercises

Exercise 1 Datastructure (3p)

First, we need a datastructure to keep track of allocations. For this purpose you need to implement a doubly linked list. Every node in the list will need to keep a pointer to the previous node, a pointer to the next node, how much free/allocated memory it has and whether it's allocated or not. Allocations will be done in chunks of 128B so internally "chunks" is the unit we use for memory size.

Exercise 2 Initialization (3p)

We need to perform an initialization of our datastructure the first time *malloc*() is called. Start by asking for some memory from the OS and assigning it to a global pointer to the first node. Then set the values for your datastructure. NULL values should be used for pointers that aren't set.

Exercise 3 Operations on the datastructure (5p)

Operations performed on our datastructure:

- Find free space
  Implement a function that takes a number of chunks as argument, traverses the list of nodes and returns a pointer to the first node in the list that has at least the given number of free chunks. NULL should be returned upon failure to find free space.
- Growing
  Implement a function that given a number of chunks, will grow the datastructure with at least that number of chunks. When requesting memory from the OS, use multiples of MIN_ALLOC_SIZE. If the last node is not allocated already, the number of chunks in the last node can then be updated with the newly gotten amount of memory. Otherwise, create a new node and attach it to the last node. Please note that it is assumed that the OS will give us contiguous memory when we ask for multiples of the page size of the system.
- Splitting one node
  Implement a function that takes a node and a size (in chunks) as argument. The node should be split into two nodes, with the first

one having the argument number of chunks and the next node having whatever is left from the original node.

- Merging two nodes
  Implement a function that takes a node as argument and joins it with the next node in the list if they both are unallocated.

Exercise 4 Tying it together (6p)

You now have all the necessary components to implement *malloc*() and *free*() so you can proceed to add your own code to perform allocation and freeing. When freeing up a node the memory allocated to that node should be merged with the two surrounding nodes if they are free to combat fragmentation.

Something to keep in mind is that your datastructure consumes some of the memory, so when returning a pointer from *malloc*(), you should of course offset it from the node datastructure with the size of your datastructure. You also need to factor this in when calling your "find space" function.

Exercise 5 Questions (3p)

1. Using a linked list to keep track of allocations gives us a performance of $O(n)$ when allocating memory, where n is the number nodes in the list. How could you improve the overall complexity of your code?
2. Suppose a task performs memory accesses outside of its allocated memory. This could possibly destroy our internal datastructure and we would get some undefined behaviour of our code. How could you implement a fault detection mechanism in your code? (Just the idea, not the actual code)
3. Is 128B a good chunk size? Why or why not?

Submission

Solutions to this assignment are to be submitted
via the Student Portal by **Sunday, April 12th, 2015**.