Memory Management Program

Written Report

Hope Brooke

sc21hb / 201438799

University of Leeds

COMP2211 - Operating Systems

29/11/22

Design Decisions

When planning my design, I combined research on the theory behind memory allocation and the implementation of linked lists. I decided that a good way of implementing my solution was to take inspiration from the free list method (A & R Arpaci-Dusseau, 2015), but make it more simple to free memory, by having a linked list that contains the non free segments, as well as the free segments, so traversal and coalescing adjacent free segments are much easier. When planning the malloc function, I chose to follow a 'first fit' method, meaning that when memory is allocated the program finds the first instance of memory that can be allocated and uses that. Although there are disadvantages to this method in comparison to 'Best Fit' or other methods, it does not require the implementation of a complicated search , and is therefore faster. As stated before, the free function requires coalescing to prevent the fragmentation of memory, and the implementation of a linked list containing both free and not free segments of data means a simple while loop traversing the linked list, with comparisons of neighboring nodes is sufficient to coalesce freed chunks of memory.

Explanation and Reasoning for Code and Features

To start with, I defined a struct called 'segment', which would essentially be the header for each segment of data in the heap. This struct holds three values, the length or the data allocated, whether it is free or not, and a pointer to the next segment. This is implementing the heap memory available for allocation as a linked list.

The first step of the malloc function was to validate the size parameter, ensuring it is greater than 0. Then, if the heap hadn't already been created, I requested memory using sbrk system call, and set the first and only node in the linked list to this. When setting the length, it was important to remember to subtract the size of the struct (header) from the memory allocated. I set the next segment to 0 to represent the end. Then I worked out the required size for allocation, which is the size parameter sent to the function, plus the size of the struct, as every new allocation of memory causes splitting, and therefore the storing of new struct header data. After this, I created a new segment struct to traverse the linked list and find free segments greater or equal to required size, this would break when found. At the end of this loop, the new segment struct will either be an available spot, or if not found, just the last struct in the list. If a free segment of required size is found, I split it. If it was not free, I expanded the heap which added a new segment on the end, and then set this new segment struct to be the added segment. Then I checked if it is of required size, and if not I expanded the heap. There is a loop that expands the segment until the size is big enough. Once the segment size is big enough, the segment is split.

To split a segment, I created a new segment, assigned a pointer and cast this pointer to the segment. This new segment is the second, free half of the original one. The values of both segments are changed to accommodate this.

To expand the heap, I created a separate function that creates a new segment. This function requests memory using sbrk system call, and then makes this the last segment in the linked list. If the previous last segment was free, the two segments are coalesced to prevent fragmentation.

The malloc function returns the pointer to the allocated memory, which is a pointer to the struct representing allocated memory + 1 (so it points to the allocated memory, and not the header storing the length, whether it's free, next segment etc.)

The free function takes a pointer to memory originally allocated, first I subtract 1 from this, and cast it to a pointer to a segment struct, which gets a pointer to the struct header. Then I check that it is a valid pointer, returning nothing if so. If the pointer was to a valid segment, I change the struct value of free to 1. To prevent external fragmentation, I then traverse the linked list of segment structs, checking if adjacent ones are free, and coalescing them if so.

Reflection

Over the course of this project, I developed a much greater understanding of many aspects of the operating system. Primarily, the use of system calls to communicate with the kernel from the user side, the different responsibilities of programs and the management of memory within an operating system. On top of an enhanced comprehension of theory, this project greatly influenced my c-programming capabilities, specifically the use of pointers and structs.

There were some things that I would do differently next time. When expanding memory I chose to expand by 512 Bytes each time, though it would be much more efficient to expand by the size requested to be allocated. When coalescing memory segments in my free function, I ran into problems within my while loop, and pointers to next segments, meaning I had to do a check for the first two nodes in the linked list before looping to check the rest. Despite this, the project went well overall.

References

Arpaci-Dusseau, R.H. and Arpaci-Dusseau, A.C. 2018. *Operating systems: Three easy pieces*. North Charleston, SC, USA: Createspace Independent Publishing Platform.