OS Final Project Report

Roberto Noel

December 15, 2019

1 Problem Description

1.1 Summary

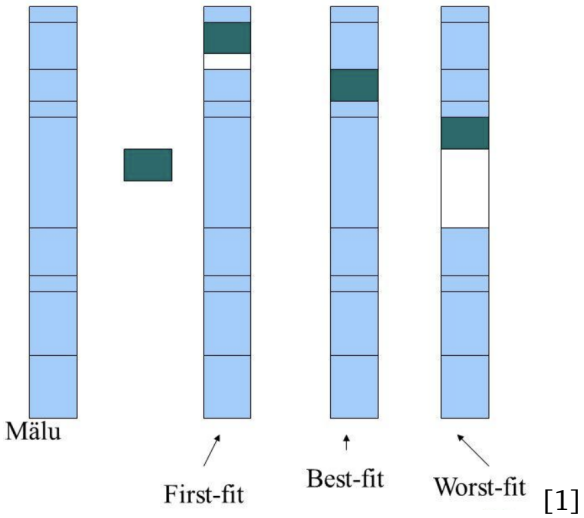
In order to store memory in the heap, computers use strategies to store as much data in memory as possible while minimizing computational overhead. This paper lays out our approach to tackling this problem using a memory simulator. For this we use two main functions: *heap malloc*, and *heap free*. We also create three supporting functions, each a different strategy for finding available memory space to allocate to: *first fit*, *best fit*, and *worst fit*.

1.2 Malloc & Free

These two main functions are the calls that the end user will make in order to allocate and deallocate space in the heap. The *heap malloc* function depends on one of the three supporting functions as the strategy to find a free memory zone to allocate to.

1.3 First Fit, Best Fit & Worst Fit

These three supporting functions are simple algorithms used to find available memory while reducing overhead. The *first fit* function simply returns the first block in the list of free memory which is able to fit the amount of memory being allocated. The *best fit* function returns the smallest free memory block that can fit this amount of memory (this introduces some computational overhead as you must iterate through the entire list every time). Lastly, the *worst fit* function finds the largest free memory block that can fit the data (same computational overhead issue).



[1]

2 User Manual

This project contains five important files, the header (os-memory.h), the memory manager (memory-management.c), the simulator (memory-simulator.c), the simulation test (memory-simulation-run.c), and the make file (Makefile).

2.1 The Header

The header defines global variables as well as functions that must be defined in order for the simulator to work. The most important global variables include: HEAP_SIZE (size of total memory), ALLOCATION_STRATEGY (default strategy), heap[HEAP_SIZE] (the simulated heap being written to), and freelist (the first element in the list of available memory zones). Lastly, a structure is declared called freezone which contains the index of the memory zone found by an allocation strategy as well as the memory zone preceding it.

2.1.1 The Memory Manager

The memory manager defines the essential functions described in the problem description. We will get into the details of how each function works in the Architecture overview.

2.2 The Simulator

The simulator defines functions that are crucial to simulating and visualizing our strategies. These include *init heap* which initializes the heap and the freelist, *print heap* which visualizes the current state of the heap, *find free zone* which switches the simulator to the allocation strategy defined in the make file, and *main* which executes the previous two functions and starts the simulation.

2.3 The Simulation Test

The simulation test is the only part of this code (aside from the make file) that we encourage you to edit. It allows the user to test our *heap malloc* and *heap free* functions. Please keep in mind that our code does not protect against buffer overflow, so only write data that is less than or equal to your allocated size.

2.4 The Make File

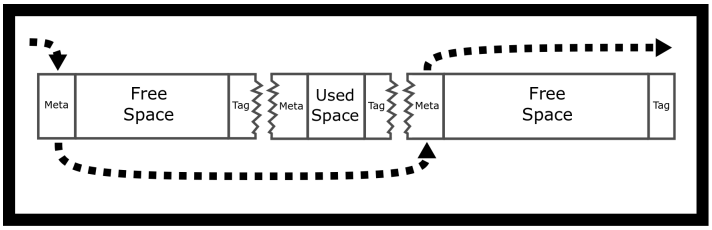
The make file compiles and runs the code. Here you can edit the STRATEGY variable to which ever allocation strategy you wish to test.

3 Architecture Overview

In this section we go over the implementation of our methods and structures as well as how we addressed edge cases.

3.1 Heap Structure

The heap is an array of HEAP_SIZE characters which simulates computer memory. Each block of free memory comprises the freelist, which is a linked list indicating each available block. The first cell in a block of free memory contains the size of memory available, the second cell contains the start index of the next available free memory block or -1 if it is the last block in the freelist. Similarly, the first cell in a block of reserved memory contains the size of memory allocated, the proceeding blocks, however, contain the data which is stored at that location.



[2]

3.2 Heap Malloc

Heap malloc is passed a size variable. The function begins by fetching the found (result.found) block (which will be written to) from whichever allocation strategy is chosen. If the strategy returns -1, it means that there is no allocatable memory left in the heap, and the malloc function returns NULL.

Otherwise, the function changes the metadata cell to the size being allocated and saves a pointer to the first writable cell in the block (this is returned later). After this it checks for two main cases.

In the first case, the size of the memory being allocated fits within the free block with at least two cells to spare. In the second case, the size of the memory being allocated is greater than the size of the free block minus two.

Each case changes the way the freelist is manipulated as a result of the allocation. In the first case, a new free block must be defined containing the metadata of the remaining space that is left over after the found free block is allocated to. After this, the new block is inserted into the linked list. To do this, the new block inherits the pointer that the found block had to the next block. Then, if the found block was the first block in the freelist, the first block is updated to be the new one. Otherwise the block preceding the found free block is changed to point to the new block.

In the second case, no new block needs to be created. If the found block was the first block in the freelist, the freelist variable is updated to be the next block. Otherwise, the previous block inherits the cell pointer of the found block.

3.3 Heap Free

The heap free function is passed the memory address of the first writable cell in the block of memory that the user seeks to deallocate. This is converted to an array index by subtracting the address of the first element in the array and dividing the result by the size of each cell, in this case, the size of char. Then, 1 is subtracted from the index so the new index describes the location of the metadata. This metadata is then saved to a size variable.

After this, the method checks for two main cases which determine the manipulation of the freelist. In the first case, the index of the block being freed is smaller than the first index in the freelist. In the second case, this index is larger than the first index in the freelist.

To control for the first case, the function checks if the first block in the freelist is contiguous to the block being freed. If it is, then the two free blocks are merged by setting the metadata of the new freed block to the sum of each corresponding metadata cell. Then, the second cell of the new block is updated to the index of the cell that the first element in the freelist was pointing to. Lastly, regardless of contiguity, the first element in the freelist is updated to be the newly freed block.

To control for the second case, the function must perform more contiguity checks. It first finds the preceding neighbor in the freelist to the current block using a while loop, which breaks once the cell being pointed to has an index greater than the current block. It then must check if the current block being freed is contiguous to either of its directly preceding or following neighbors in the freelist. If so, the same merging procedure is done as in the first case. If the current block is not merged with its following neighbor in the freelist, its linked list pointer is updated to point to this neighbor. Lastly, if the block is not merged with its preceding neighbor, the preceding neighbor's pointer is updated to

point to the freed cell.

3.4 Fit Algorithms

1. First Fit: This algorithm simply loops through the freelist until it finds a free block which can fit the size being allocated. If there is space for allocation, the index of the metadata for the found block is passed to the freezone structure along with the preceding free block in the freelist. Otherwise, a -1 value is passed which is later caught by the heap malloc function.

2. Best Fit: This algorithm also loops through the freelist, but it stores three important variables: the index of the tightest fit block, the index of the preceding block, and the tightest current size. The tightest current size is initialized to the size of the heap, and the other two variables are initialized to -1. These variables are updated when a better fit is found while looping through the freelist. If no better fit is found, the initialized -1 value is passed which is later caught by the heap malloc function.

3. Worst Fit: This algorithm works in exactly the same way as the Best Fit, except the loosest current size variable is initialized to the size being allocated. Additionally, the three tracking variables are only updated when a looser fit is found (the sign is flipped).

4 Problem Analysis

While implementing the various functions in this simulator, we faced a few design challenges which included trade-offs between different implementations.

4.1 Matadata Storage

One decision we faced was whether or not to structure the linked list as a doubly linked list for easier iteration through the free blocks. We ultimately decided against doing this, as we realized that it was more effective to simply pass the previous free block in the freezone structure (defined in the header). Implementing the freelist as a doubly linked list would drastically reduce our memory capacity in the heap as we would need more metadata.

4.2 Another Way to Structure the Freelist

In our *heap free* function, we use a while loop to iterate through each element in the freelist and find the free block preceding the block that we are currently freeing. We could increase the speed of this process by storing the indexes of our freelist in an additional binary search tree. Ultimately,

however, the increased memory overhead as well as edits that would have to be made to the tree would outweigh the benefit of a slightly faster search for this index.

We could also increase the efficiency of the best and worst fit functions by taking a similar approach and storing a binary search tree based on free block sizes. Despite the slight increase in efficiency for these functions however, the tradeoffs with this approach are similar.

4.3 Allowing Free Blocks of Size Zero

The last decision we faced was whether or not to allow free blocks of size zero in our freelist. This would happen when exactly two spaces are available for a free block and they are reserved for metadata. Ultimately, we decided that even though it might mean iteration through the freelist would take slightly longer, the effect of this would hardly be noticeable.

References

[1] http://www.myshared.ru/slide/485756/

[2] https://github.com/illinois-cs241/coursebook/wiki/Malloc