ITP 30002-02 Operating System, 2021 Spring

Homework 4

Jung Bo Moon, 21600635, 21600635@handong.edu

1. Introduction

Because the C language explicitly allocates memory, if the memory is not managed in an appropriate way, external fragmentation occurs and memory is wasted. To prevent this, we implemented appropriate memory coalescing and memory shrinking. In particular, in the case of memory coalescing, it is important to realize and implement that there are so many cases. So, since I thought a lot about solving memory coalescing and implemented the code, I implemented rmshrink very easily. In this task, the rmalloc library, in order to efficiently implement the above-mentioned things, a structure containing the spatial information obtained through mmap, header, free list, and used list was implemented. In addition, external fragmentation can be reduced by finding and allocating an appropriate memory space through three policies.

2. Approach

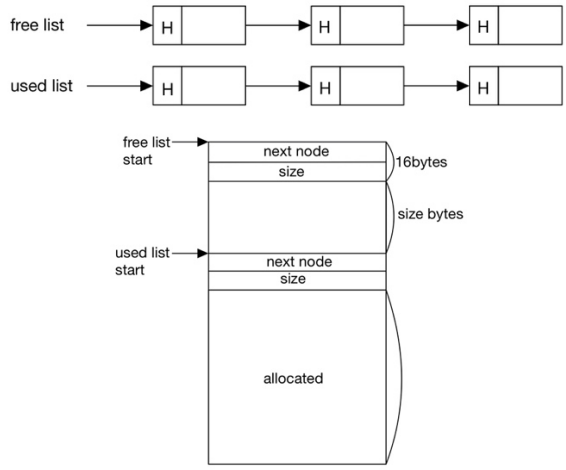

figure 1. overall design of data structure

2.1. data structure

Several data structures were used to implement rmalloc. First, I created an array to store the starting address and size of the memory reserved by mmap. Among the reserved spaces, chunks of memory that have not yet been allocated are connected to the free list, and chunks with allocated memory are connected to the used list. Thus, memory chunks move through two lists. At this time, in order to efficiently manage chunks, the first 16 bytes of all memory chunks are headers. Header stores a pointer to the next chunk and a size that can actually store data excluding its own size.

2.2. memory allocation (rmalloc)

When a user requests memory allocation in a desired size, it searches for a suitable memory chunk from the free list according to the policy type. If the chunk is the same as the requested size, you can immediately add it to the used link from the free list, but if the chunk is larger, split the chunk as needed and manage it in each list. However, when there is no adequate space, allocate a page size * n memory chunk that can accommodate the requested size through mmap, and then split it by the requested size. If the requested size is the wrong size, null is returned.

2.3. deallocation (rfree)

If the memory receives a memory release request from the user, it is easiest to find the corresponding chunk in the used list, remove it from the used list, and put it in the free list. However, this method does not efficiently manage memory because external fragmentation occurs and memory waste occurs. Therefore, it is determined whether the found chunk and adjacent chunks can be coalesced, and if they are adjacent, coalescing should be performed. However, there are many cases because there is no guarantee of contiguous memory space every mmap in rmalloc. The number of cases is when both the front and back adjacent chunks are freed, only the previous chunk is freed, only the back chunk is freed, and the front and back chunks are in use chunks(subdivided as in figure 2). If not, it is handled separately because the requested chunk is at the very beginning and at the end. If the chunk that was requested to be released is not in the used list, an error message (rsegmentation fault) is displayed and the program is terminated.

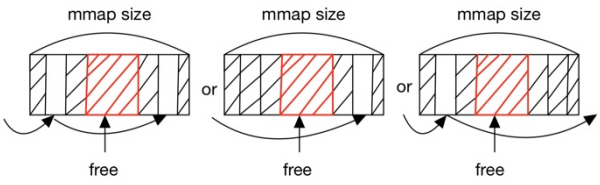

figure 2. subdivided case

2.4. reallocate (rrealloc)

Resizes the size of the previously allocated chunk to the size requested by the user. At this time, if the requested size is the wrong size, null is returned. If the requested size is 0, rfree is performed. If not in the above two cases, rmalloc

is performed as much as the requested size, then the existing information is transferred through memcpy, and the previously used chunk is released through rfree.

2.5. rmshrink

When rmshrink is called, an array containing information about the address and size of the space reserved through mmap is used. If the size of the chunks of the free list within the address range allocated by mmap is the same as the size allocated by mmap, the corresponding space is released by munmap and waste is reduced.

3. Evaluation

The data structure containing information about the memory allocated through mmap is a global array. Therefore, the number of times mmap can be executed is predetermined. However, the actual heap is also a limited space. In other words. After the specified number of times, it is assumed that there is no more memory space available for allocation. Therefore, the error message 'rheap overflow' is printed and the program is terminated, preventing error in advance. When I ran many test cases, I couldn't find a case where coalescing went wrong.

3.1. the result of declaring rmalloc 4 times in order

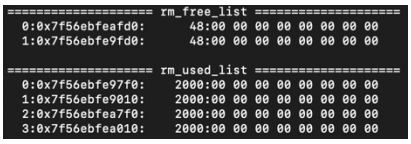

figure 3. result of 4 times rmalloc

3.2. rrealloc the last allocation to 3000

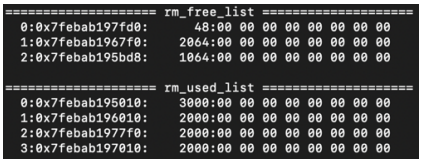

figure 4. result of rrealloc

3.3. rfree all allocated and rmshrink

You can see that coalescing and rmshrink went well.

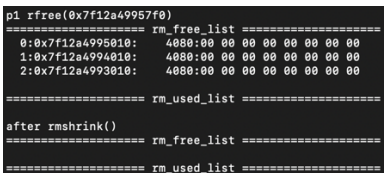

figure 5. result of rfree and rmshrink

4. Discussion

The method of using the free list and the used list is also efficient, but considering other methods, there is a way to manage memory with a single heap list. Instead, the header must additionally contain information that can be used to determine whether or not to be allocated. If the code is implemented in this way, I think it will be very easy to do when coalescing. The reason is that you can easily implement coalescing with time complexity O(n) by reading only the front and rear chunk header information in the rfree chunk.

5. Conclusion

C language explicitly allocates memory, so we need to have an accurate understanding of how memory is managed. The overall flow of memory management is that when a user requests memory allocation to a desired size, the appropriate chunk is found in the free list according to the policy, and the chunk of the required size is moved to the used list. Also, when rfree is requested, the chunk is removed from the used list and placed in an appropriate position in the free list. At this time, if the free list is managed without coalescing, external fragmentation occurs, which wastes a lot of space. Therefore, I implemented the code to be able to coalescing according to the number of cases. As a result of executing various test codes, you can see that the rmalloc library functions work well for Best Fit, Worst Fit, and First Fit.

In conclusion, external fragmentation may occur in the process of memory allocation and deallocation, and I think that preventing this phenomenon from occurring is the core of this task. Therefore, the problem of wasted memory can be solved by implementing well how the chunk freed when rfree is requested and how to reduce the unused space in the allocated space when rshrink is performed.