A Custom Malloc Library

Anran Chen (ac692)

Abstract

The aim of this study is to propose an implementation of C standard Malloc/Free library. The algorithm is built upon a modified doubly linked list data structure. The implementation is a satistactory effort to replicate the C standard libary, but suffereing from relatively slow execution speed and high memory overhead.

Implementation Details

The allocated heap space of the program is devided into segments. Each segment contains two parts, namely the metadata and the storage space. The storage space of a segment is used to store the actual data, and the metadata is used to keep information regarding the segment's relative location, size and availability, which is a struct as follows

```
struct _metadata {
  size_t dataSize;
  bool occupied;
  struct _metadata* prev;
  struct _metadata* next;
};
typedef struct _metadata Metadata;
```

A segment can have two status: occupied or free. Once freed, the segment is added to a doubly linked list call the free list. The free list tracks all freed segments for future use, ranked by each one's physical address.

When an arbitrary size of memory is requested, the free list is searched first to look for any suitable deallocated segments according to the allocation policy. If none are available, a new segment is created using C standard sbrk() library. Otherwise, depending on if the chosen segment is splitable, the entire segment or a portion of the segment will be allocated. If, however, the sbrk() is unsuccessful due to the full occupancy of the heap space, a null pointer will be returned.

A segment is defined "splitable" if it has enough space for the requested data, in addition to an extra metadata block. If the segment is not deemed splitable, the entire segment is allocated for the request. Otherwise, the segment will be splited into two, with the first portion fulfilling the request, and the second standing by as free space.

When a segment is freed and added to the free list, it is coalesced with its neighboring blocks if they are physically adjacent. This would defragment the freed memory space for more optimized memory allocation performed later.

Performances

Experimentation

The library is implemented in C, and executed on a Windows 10 WSL with an Intel i7-7700HQ CPU and 8GB of RAM. Three sets of experiments are performed to measure the outcome: equal size alloc, small random size alloc and large random size alloc. The number of iterations and the block sizes are according to the default values. The execution times and fragmentation proportions are shown in table 1.

	First Fit		Best Fit	
	Speed (s)	Fragmentation	Speed (s)	Fragmentation
Equal Size	17.61	0.45	16.43	0.45
Small Random Size	20.49	0.07	4.12	0.02
Large Random Size	69.71	0.09	98.94	0.04

Table 1: Performance results

Speed

For equal size allocations, both policies display similar performances. First-fit policy relies on a greedy approach to search for the free segment, thus promises a fast execution. While best-fit policy has special conditions to check for equality between the requested size and the freed segment, and allocates immediately if the condition fulfills. This allows the library to break out from exhaustive searching, further optimizing its execution.

For small size random allocations, best-fit policy executes moderately faster than first-fit. In this case, the first-fit policy would split the closest available segment, while the best-fit policy traversing through the free list. If the free list is relatively short, although traversing is of linear complexity, it still has a better performance than the greedy approach, which would spend considerable time splitting and rearranging segments each time upon a request.

Both policies are not suitable for large size random allocations, as the times taken are unrealistic for normal computation demand comparing to its C standard malloc counterpart. First-fit policy performs better, as the greedy allocation does not travese the entire free list, as opposed to the exhaustive searching strategy, when the free list too exceedingly varying to search for the best fit.

Fragmentation

Both policies perform the same under equal size allocation, as the allocation requests are all identical, the segment choices would also be identical disregarding different approaches.

For both random size allocations, besf-fit policy surpasses first-fit policy, as the former has a more optimized segment selection strategy, reducing the need to split a bigger segment and leaving the rest of it too small to be requested. The effect is especially conspicuous for

larget random size allocation, where the latter only results in a third of the fragmentation of its former's.

Discussion and Conclusion

It is discovered that there is a tradeoff between the speed of allocation and the integrity of the memory space. First-fit policy is suitable for a series of more varying and unpredictable memory requests, or when speed is more important than memory integrity. Whereas the best-fit policy should be chosen where the requested sizes are moderately consistent, or when the memory should be preserved as clean as possible. However, both allocation policies are too slow to be considered for practical usage.

Possible Future Work

Although this custom implementation replicates the functionality of its C standard counterpart, it sufferes from two significant drawbacks: the execution speed and the memory overhead. To improve the execution speed, alignment can be used to pre-fragment the segments, which removes a large proportion of clock cycles wasted to promptly split and coalesce blocks during execution as in the current implementation. A second doubly linked list could be introduced to maintain the physical arrangement of the segments, while the free list could be appended to the end, instead of inserted according to the segment's address. This would reduce the free operation to a constant time complexity. Concurrency is also a promising to speed up searching for free segment, as opposed to searching linearly.

A large portion of the memory is consumed by the segments' metadata. The currently implementation has a metadata size of 32 bytes, exceedingly extravagant comparing to its C standard implementation. Smaller size primitive data types and less pointers can be used to reduce this overhead.