Heap Allocator

Kartik Sawhney <kartiks2>, Winnie Lin <wl1915>

(Note: throughout this discussion, `requested size` may be greater than the size actually requested by the user to adhere to alignment properties)

DESIGN

The heap allocator makes use of the segregated fits model. Accordingly, an array of free lists is maintained. The bucket in which the free pointer resides (in other words, the index of the array that the free pointer occupies) is determined by its size. A hashing function determines the index corresponding to a range of size. this hashing function seeks to spread out the more commonly malloc'd values to reduce search time. Small values below 128 bytes are put into fixed size buckets, and anything above that is placed into buckets of power of two, with the last bucket containing everything from 8MB up to 1GB.

Each free chunk has a pointer to the previous chunk in the bucket, and another one to the next one. This helps in traversing the list. Further, the header (of type size_t) contains information about the size of the memory chunk, alongwith information about the allocation status of the current and the previous chunk (using the two lower bits in the 4 bytes).

When a malloc request is made, the hashing function leads into the appropriate bucket to search for. If a chunk of adequate size is found, it is removed from the bucket, and the previous and the next pointers of its previous and next chunks are appropriately changed to reflect this removal. If that particular bucket does not contain a chunk of adequate size, then subsequent buckets are searched. If found, a similar removal process takes place. Further, if the size of a chunk is greater than the requested size by a threshold limit, then splitting takes place, wherein the requested size is malloc'd. The remaining chunk continues to be free, and is stored in the bucket corresponding to its size. Finally, if a memory chunk of adequate space is not found anywhere on the heap, then the heap is extended by an appropriate size.

When freeing a pointer, it is first checked if the chunks right before and after it are free. If that is the case, then the memory chunks are coalesced to give chunks of larger sizes, and stored in the appropriate bucket (based on their size). Otherwise, it is simply added to the beginning of the corresponding bucket, and the pointers are updated to reflect this addition.

myrealloc makes use of mymalloc and myfree alongwith memcpy. Further, it allocates double the size requested by the user, anticipating another call to myrealloc soon.

RATIONALE

This approach was adopted due to an optimum balance of throughput and utilization efficiency. Searching time is reduced in this approach, because it is constrained by the hash function. Utilization is improved, because a first fit search of a segregated free list essentially approximates a best fit search of the entire heap.

Other approaches that were discussed during the design process include simple implicit free list, explicit free list and simple segregated system etc. However, for each of these strategies, there appeared to be significant compromise on one parameter at the cost of the other. For instance, block allocation time is linear (O(n)) for an implicit free list, while explicit free lists require the free chunks to be large enough to contain all the necessary pointers, in addition to a header and a footer. Segregated fits was thought to be the ideal combination providing both the benefits of time and utilization efficiency.

However, the linking of lists do eat up a lot of efficiency, and if we have the chance to try this again, it would be interesting to try sorting the lists into trees, and see if the search time would outweigh the time it takes to link the memory chunks.

OPTIMIZATION

The optimization strategies were guided by content in the assignment writeup and discussions with the TAs. Issues such as incorporating builtin functions as opposed to writing our own definitions, declaring inline functions to avoid the overhead due to function calls, etc. were considered. Calgrind was a great help in pinpointing particular sections in the code that were problematic, which helped focus on particular parts of the code that could be improved. One function we used was __builtin_clz to write our hash function. It actually managed to raise throughput quite impressively, due to the sheer amount of times the function was called.

Separating the memory lookup loops into different methods for fixed sizes and variable-sized buckets in malloc proved to also drastically improve throughput, due to eliminating the need for a while loop in the fixed size buckets. Keeping track of the heap size and page size with global variables was also something that reduced the external function calls in the program. These few examples illustrate that some optimization strategies we had were to simplify conditional statements, use inbuilt functions when we could, and restrict calls to external functions as much as possible.

Further, optimization also involved a lot of experimentation. Potential optimization changes were often made, and its impact on efficiency and throughput evaluated. Further, cache hits and misses were analyzed to gauge the cache-friendliness of the program. Compiler optimizations such as -O7 proved to be extremely helpful.

EVALUATION

Our strengths are probably the efficiency of double linked list, and the high utilization achieved with getting the right combination of fixed buckets, splitting thresholds and double coalescing. A major weakness is that we could have written our coalescing function to be a bit more streamlined by reducing the number of calls to the buckets.

REFERENCES

1. Bryant and O'hallaron (section 9.9)
2. New Mexico State University: memory allocators (http://www.cs.nmsu.edu/~ekerriga/presentation/index2.html)