Dynamic Memory Allocation

Tuesday, November 12, 2019 2:56 PM

It's pretty common to encounter memory items that we don't know the size of until runtime
-- These are stored dynamically on the heap

heap - an area of demand-zero memory that begins after the uninitialized data area, grows upwards
kernel variable "brk" (break) points to the TOP of the heap

The heap is realized as a collection of *blocks*, that can be either *free* or *allocated*.

Allocated blocks are reserved for use by the program, and are made this way by a program that manages the heap (malloc)
Free blocks are available to be allocated. Blocks start as this, and allocated blocks can be freed by a program that manages the heap (free).

Implicit Allocators: Languages like Java/Python are *managed languages* - no explicit pointers, don't have to worry about freeing/blocking memory like in C. Implicit allocators are also known as *garbage collectors* - common in higher-level languages.

Languages like C use Explicit Allocators - the program/code has to explicitly free any allocated blocks

```
void foo() {
        int* p = malloc(128);
        return;
}
```

- C: This will crash your program, and maybe your entire system, if you loop it enough times! We're leaking memory folks!!!
- Python/Java will automatically free blocks that have absolutely no pointers pointing to them. This garbage collection will take care of memory leaks

We use Virtual Memory - A "precious concept"

C uses malloc to allocate memory
(C++ equivalent: new)

C uses free to deallocate memory
(C++ equivalent: delete)

void *malloc(size_t size)

- successful: returns a generic pointer (void*) to a memory block of at least *size* bytes, aligned to a 16-byte boundary in x86-64. (blocks left uninitialized)
- unsuccessful: returns NULL(0) & sets *errno*
- we like to align things to words, so generally calls look like malloc(4*sizeof(int)) <- allocated 4

words, aka space for 4 ints

void free(void *p)

- frees the block pointed at by *p*, by putting it back in the pool of available memory
- p has to come from a previous call to *malloc* or *realloc*
- doesn't return anything, easy to get enigmatic errors here for that reason

calloc : version of malloc, initializes allocated blocks to 0
realloc: changes size of a previously allocated block
sbrk : used internally to grow/shrink heap

void *sbrk(intptr_t incr);

- adds *incr* to the heap's brk pointer
- returns the OLD value of brk on success
- failure: returns -1, sets *errno* to ENOMEM
- possible to call with negative numbers, but kinda stupid

Memory is word-addressed. In this context, 1 word = 4 bytes
a double word = 8 bytes
this contrasts with Intel's Words: 2 byte word // 4 byte double words

Allocators (the program that allocates memory) have many constraints:

- They can't control number or size of blocks (have to handle an arbitrary number of arbitrarily large requests)
- They have to respond immediately, can't buffer requests
- They can only place blocks in free memory, and can only use the heap
- They have to satisfy alignment requirements when allocating blocks (16-bit on x86-64)
- They can only manipulate/modify free memory. This means they can't move *malloc*-ed blocks at all (no compaction!)

Two Main Goals:

- Maximizing Throughput
- Peak Memory Utilization

Given some sequence of *malloc* / *free* requests,

malloc(p) results in a blocks with a payload of p bytes. After Rk has completed, the aggregate payload Pk is the sum of currently allocated payloads

Hk is the current heap size

Internal Fragmentation occurs if payload is smaller than the block size

Block

unused	Payload	aughhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh	unused

these unused blocks are the internal fragmentation
Easy to measure, based off the payload of previous requests

External fragmentation occurs when there's enough aggregate heap memory, but no single free block is large enough
Difficult to measure, because it's based on the pattern of future requests

Issues with minimizing fragmentation:
…

there's no one best way to do this. Variable sized memory allocation is a deep field that's pretty fundamentally tough

Keeping track of free blocks:
make an implicit list using length, which links all the blocks
can also make an explicit list, using pointers among the free blocks. There's more overhead, but this method is actually better

Implicit list: (default code)

For each block, we need both size and allocation status. Store this in a single word

The addresses are always aligned to double words, which means the last three bytes will always be 0. That means we can use one of these always-zero bits to store whether the address is free or taken!

Finding free blocks:

First Fit: search list from beginning, and choose the first free block that fits
- Linear time in total number of blocks, but causes "splinters" at beginning of list

Next fit: like first fit, but start where the previous search finished

Best fit: search, and choose the best free block (appropriate block with the smallest size)
- keeps fragmentation to a minimum, improves memory utilization -- slower

When placing an item into the list, it's common practice to SPLIT FREE BLOCKS into multiple new blocks IF THE FIT IS NOT GOOD. What this looks like:
3 word block requested (4 words including header), 8 word free block found.
Use the first 4 words to allocate, and split the last 4 words into its own new free block.

Bidirectional Coalescing
Replicate size/allocated word at the end of free blocks
allows us to traverse the memory "list" backwards. Takes more space, but adds tons of functionality!

Constant time Coalescing

Explicit List:
linked list of free elements. When you allocate something, iterate through and find a good block, then remove it from the list.
Still need tags to indicate if blocks are free/allocated, as coalescing requires this to be efficient

Segregated List Allocators
Each size class of blocks has its own free list - the list contains only free blocks
First-fit search on appropriate free list for a blocks that fits
if found, split it and insert the fragment in appropriate free list
if not found, try next larger size class
if no blocks found, request additional heap memory from kernel/OS - place remainder in a free list

1-2
3
4
5-8
9-inf

one class for each two-power size. Separate classes for each small size tho
efficient to index into as well - USE THIS IN MALLOC LAB

Implicit List
Explicit List
Segregated Free List
Blocks sorted by size - use a balanced tree with pointers in each free block. Use the length of the block as the search key

These are all used for explicit allocators, e.g. malloc and free in C

2 common policies for inserting new free blocks in the list:

LIFO policy: insert at the beginning of the list
- constant-time freeing
- first fit placement policy inspects recently used blocks first

Address-ordered policy: address of each block in the list is less than its sucessor
- freeing is a linear-time search :(
- first fit utilizes memory better
- easier to keep track of probably