ECE254: Operating Systems and Systems Programming Spring 2015

Lecture 20 — Dynamic Memory Allocation

Jeff Zarnett

Dynamic Memory Allocation

By now you must surely be familiar with dynamic memory allocation from the perspective of the application developer. To create a new instance of an object in Java, for example, you use the `new` keyword and the runtime will come and garbage collect it when it is no longer needed. In C++ we have the `new` and `delete` operators to allocate and deallocate memory. The `new` and `delete` operators also allocate memory, but they also invoke the constructor and destructor, respectively. C works on memory at a lower level: to allocate a block of memory in C, there is `malloc()` and when finished, you return it with `free()`. This level is a lot closer to the way the operating system thinks about memory: just tell me how much you need and tell me when you are finished with it.

If we generalize this interface, we get two signatures [HZM14]:

`void *allocate_memory( int size )`
Allocate a block of `size` bytes of memory; return a pointer to the address of the first byte.

`void deallocate_memory( void *mem_block )`
Return the allocated block of memory at the address `mem_block` to the pool of free memory.

This should square nicely with your experience of using `malloc()` and `free()` in C. To allocate an integer, you call `malloc( sizeof( int ) )`. This creates, somewhere in memory, a new integer and returns the address of it that can be stored in a pointer (presumably an integer pointer, but you can store it in a void pointer too). To be sure to ask for the correct amount of memory, we have `sizeof` which works out the size of its argument (integer) and then the size of an integer, say, 4 bytes, is supplied to `malloc()`, so 4 bytes are allocated.

When you `free()` that pointer, all that happens is that the memory is marked as available, which is why you can sometimes get away with dereferencing a pointer after it has been freed. Sometimes it takes a while for that memory to be reclaimed or reused so the old value just happens to still be there in memory. Note that `free()` does not specify how much memory is being returned. This means two things: (1) that the operating system is keeping track of each allocated block's size, and (2) that it is not possible to return part of a block.

With the preliminaries about memory allocation out of the way, now it is time to turn our attention to fulfilling the memory allocation requests that we receive. As we will see, this is not a trivial problem. The operating system will try to find some free memory to meet the request. Although running out of memory is a rare thing given the size of main memory in a modern computer, there is still the possibility that some request may not be fulfilled because no block meeting that need is available.

Fixed Block Sizes

One possibility for how to allocate memory is in fixed block sizes. All blocks of memory allocated are the same size. This does not mean that requests are not of varying size, it just means that all blocks allocated are the same size. If a request comes in for 1 byte, 1 block is allocated. If a request comes in that is, say, 1.5 blocks, 2 blocks are allocated.

It is immediately obvious when we look at this that some memory is "wasted". If 1.5 blocks are requested and 2 blocks are allocated and returned, we are using up an extra 0.5 blocks. This space cannot be used for anything useful (as it shows as allocated). This is a problem called `internal fragmentation` – unused memory that is internal to a partition. This is obviously going to occur often when fixed block sizes are used, and the bigger each block is, the more memory will be wasted in internal fragmentation.

One Size of Blocks. Suppose the system has only one size of blocks, perhaps, 1 KB. To implement this strategy, divide up memory into blocks of this fixed size and maintain a linked list of addresses of all currently available blocks. When a block is allocated, remove its corresponding node from the linked list; when a block is freed, put a node with that address into the linked list. If the list is empty, a memory request cannot be satisfied, and null will be returned. This is definitely fast as we can allocate memory in $\Theta(1)$ time [HZM14].

Fixed Block Sizes, Multiple Size Options. Recognizing that some memory allocation requests are bigger than others, it might make sense to have several different block sizes; perhaps 1 KB, 2 KB, and 4 KB. These can generally be allocated and deallocated in $\Theta(1)$ time if we have one linked list for each different size of block [HZM14].

Unfortunately, fixed block sizes suffer from a lot of internal fragmentation. While this may be suitable for embedded systems where simplicity and speed of operations are more important than worrying about wasting memory. It is obvious from working with languages like C that this is not how `malloc()` works: 1 KB of memory is not allocated to store a 4-byte integer. What we need instead is a variable block size.

Variable Block Sizes

To a certain extent, variable block sizes are not that different from fixed block sizes; we just take the size of blocks down to the smallest they can be. In a typical system with byte-addressable memory, in a way, the smallest block is one byte. Now we have a different problem: keeping track of what is allocated and what is free.

Bitmaps. It is possible to divide memory into M units of n bits, and then to create a bit array of size M storing the status of each of those units. If a bit m in M is 0, it means that unit is unallocated; if it is 1 then that unit is allocated. How much memory is lost to this overhead? $100/(n+1)$% of the memory is used. If a unit is 4 bytes, the bitmap is about 3% of memory; if it is 16 bytes the bitmap takes about 0.8% of memory. Finding a block of k bytes requires searching the bitmap for a run of $\frac{8m}{n}$ zeros [HZM14].

Linked Lists. The other approach, as in the case of fixed size blocks, is to use linked lists. The information of the linked list can be stored separately from all memory allocation or as part of the block of memory. Either approach is workable.

References

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghami. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.