Lecture 23: More on Memory Management

Reading: Chapter 3 in Samet's notes.

Buddy System: The dynamic storage allocation method described last time suffers from the problem that long sequences of allocations and deallocations of objects of various sizes tends to result in a highly fragmented space. The *buddy system* is an alternative allocation system which limits the possible sizes of blocks and their positions, and so tends to produce a more well-structured allocation. Because it limits block sizes, *internal fragmentation* (the waste caused when an allocation request is mapped to a larger block size) becomes an issue.

The buddy system works by starting with a block of memory whose size is a power of 2 and then hierarchically subdivides each block into blocks of equal sizes (like a 1-dimensional version of a quadtree.) To make this intuition more formal, we introduce the two elements of the buddy system. The first element that the sizes of all blocks (allocated and unallocated) are powers of 2. When a request comes for an allocation, the request (including the overhead space needed for storing block size information) is artificially increased to the next larger power of 2. Note that the allocated size is never more than twice the size of the request. The second element is that blocks of size 2^k must start at addresses that are multiples of 2^k. (We assume that addressing starts at 0, but it is easy to update this scheme to start at any arbitrary address by simply shifting addresses by some offset.)

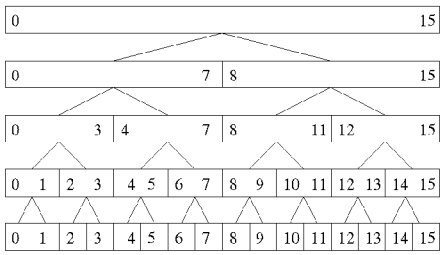

Figure 68: Buddy system block structure.

Note that the above requirements limits the ways in which blocks may be merged. For example the figure below illustrates a buddy system allocation of blocks, where the blocks of size 2^k are shown at the same level. Available blocks are shown in white and allocated blocks are shaded. The two available blocks at addresses 5 and 6 (the two white blocks between 4 and 8) cannot be merged because the result would be a block of length 2, starting at address 5, which is not a multiple of 2. For each size group, there is a separate available space list.

For every block there is exactly one other block with which this block can be merged with. This is called its *buddy*. In general, if a block b is of size 2^k, and is located at address x, then its buddy is the block of size 2^k located at address

$$buddy_k(x) = \begin{cases} x + 2^k & \text{if } 2^{k+1} \text{ divides } x \\ x - 2^k & \text{otherwise.} \end{cases}$$

Although this function looks fancy, it is very easy to compute in a language which is capable of bit manipulation. Basically the buddy's address is formed by complementing bit k in the binary representation of x, where the lowest order bit is bit 0. In languages like C++ and Java this can be implemented efficiently by shifting a single bit to the left by k positions and

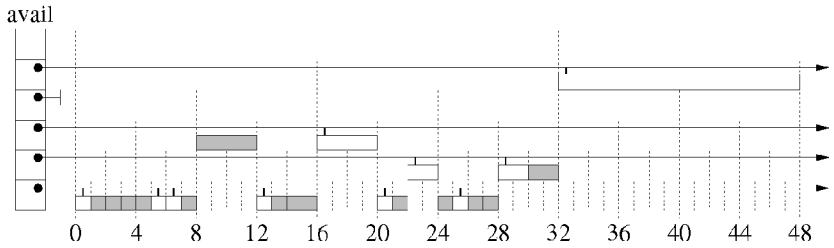

Figure 69: Buddy system example.

exclusive or-ing with x, that is, `(1<<k)^x`. For example, for $k = 3$ the blocks of length 8 at addresses 208 and 216 are buddies. If we look at their binary representations we see that they differ in bit position 3, and because they must be multiples of 8, bits 0–2 are zero.

$$\begin{aligned} \text{bit position:} &\quad 876543210 \\ 208 &= 011010000_2 \\ 216 &= 011011000_2. \end{aligned}$$

As we mentioned earlier, one principle advantage of the buddy system is that we can exploit the regular sizes of blocks to search efficiently for available blocks. We maintain an array of linked lists, one for the available block list for each size, thus `avail[k]` is the header pointer to the available block list for blocks of size k.

Here is how the basic operations work. We assume that each block has the same structure as described in the dynamic storage allocation example from last time. The `prevInUse` bit and the size field at the end of each available block are not needed given the extra structure provided in the buddy system. Each block stores its size (actually the log k of its size is sufficient) and a bit indicating whether it is allocated or not. Also each available block has links to the previous and next entries on the available space list. There is not just one available space, but rather there are multiple lists, one for each level of the hierarchy (something like a skip list). This makes it possible to search quickly for a block of a given size.

Buddy System Allocation: We will give a high level description of allocation and dealloction. To allocate a block of size b, let $k = \lceil \lg(b+1) \rceil$. (Recall that we need to include one extra word for the block size. However, to simplify our figures we will ignore this extra word.) We will allocate a block of size 2^k. In general there may not be a block of exactly this size available, so find the smallest $j \geq k$ such that there is an available block of size 2^j. If $j > k$, repeatedly split this block until we create a block of size 2^k. In the process we will create one or more new blocks, which are added to the appropriate available space lists.

For example, in the figure we request a block of length 2. There are no such blocks, so we remove the the block of length 16 at address 32 from its available space list, split it into subblocks of sizes 2, 2, 4 and 8, add the last three to the appropriate available space lists, and return the first block.

Deallocation: To deallocate a block, we first mark this block as being available. We then check to see whether its buddy is available. This can be done in constant time. If so, we remove the buddy from its available space list, and merge them together into a single free block of twice the size. This process is repeated until we find that the buddy is allocated.

The figure shows the deletion of the block of size 1 at address 21. It is merged with its buddy at address 20, forming a block of size 2 at 20. This is then merged with its buddy at 22,

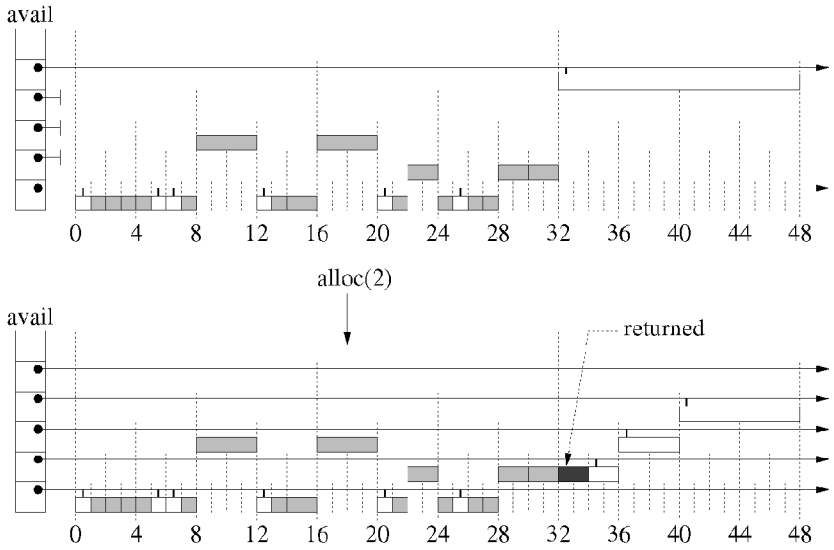

Figure 70: Buddy allocation.

forming a block of size 4 at 20. Finally it is merged with its buddy at 16, forming a block of size 8 at 16.

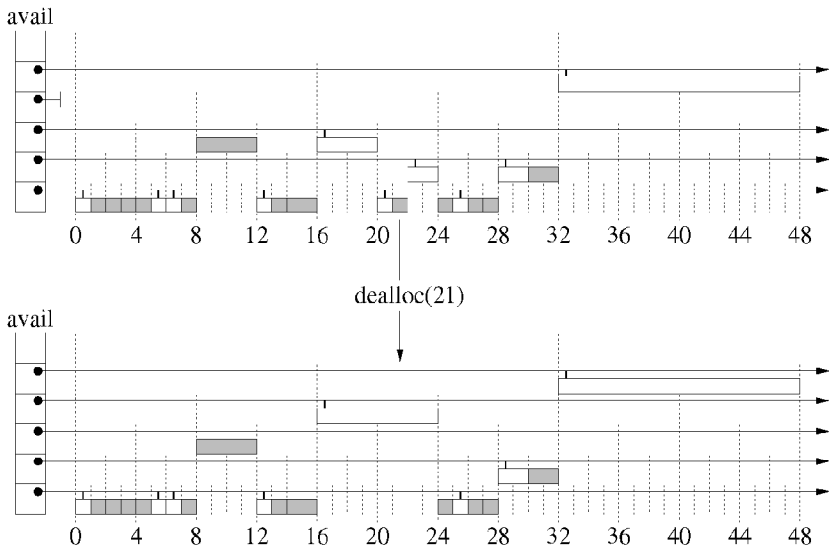

Figure 71: Buddy deallocation.

Lecture 24: Garbage Collection

Reading: Chapter 3 in Samet's notes.

Garbage Collection: In contrast to the explicit deallocation methods discussed in the previous lectures, in some memory management systems such as Java, there is no explicit deallocation of memory. In such systems, when memory is exhausted it must perform *garbage collection* to reclaim storage and sometimes to reorganize memory for better future performance. We will consider some of the issues involved in the implementation of such systems.

Any garbage collection system must do two basic things. First, it must detect which blocks of memory are unreachable, and hence are "garbage". Second, it must reclaim the space used by these objects and make it available for future allocation requests. Garbage detection is typically performed by defining a set of *roots*, e.g., local variables that point to objects in the heap, and then finding everything that is reachable from these roots. An object is *reachable* (or *live*) if there is some path of pointers or references from the roots by which the executing program can access the object. The roots are always accessible to the program. Objects that are not reachable are considered garbage, because they can no longer affect the future course of program execution.

Reference counts: How do we know when a block of storage is able to be deleted? One simple way to do this is to maintain a *reference count* for each block. This is a counter associated with the block. It is set to one when the block is first allocated. Whenever the pointer to this block is assigned to another variable, we increase the reference count. (For example, the compiler can overload the assignment operation to achieve this.) When a variable containing a pointer to the block is modified, deallocated or goes out of scope, we decrease the reference count. If the reference count ever equals 0, then we know that no references to this object remain, and the object can be deallocated.