Lecture 22: Memory Management

Reading: Chapter 3 in Samet's notes.

Memory Management: One of the major systems issues that arises when dealing with data structures is how storage is allocated and deallocated as objects are created and destroyed. Although *memory management* is really a operating systems issue, we will discuss this topic over the next couple of lectures because there are a number interesting data structures issues that arise. In addition, sometimes for the sake of efficiency, it is desirable to design a special-purpose memory management system for your data structure application, rather than using the system's memory manager.

We will not discuss the issue of how the runtime system maintains memory in great detail. Basically what you need to know is that there are two principal components of dynamic memory allocation. The first is the *stack*. When procedures are called, arguments and local variables are pushed onto the stack, and when a procedure returns these variables are popped. The stacks grows and shrinks in a very predictable way. Variables allocated through `new` are stored in a different section of memory called the *heap*. (In spite of the similarity of names, this heap has nothing to do with the binary heap data structure, which is used for priority queues.) As elements are allocated and deallocated, the heap storage becomes fragmented into pieces. How to maintain the heap efficiently is the main issue that we will consider.

There are two basic methods of doing memory management. This has to do with whether storage deallocation is done *explicitly* or *implicitly*. Explicit deallocation is what C++ uses. Objects are deleted by invoking `delete`, which returns the storage back to the system. Objects that are inaccessible but not deleted become unusable waste. In contrast Java uses implicit deallocation. It is up to the system to determine which objects are no longer accessible and reclaim their storage. This process is called *garbage collection*. In both cases there are a number of choices that can be made, and these choices can have significant impacts on the performance of the memory allocation system. Unfortunately, there is not one system that is best for all circumstances. We begin by discussing explicit deallocation systems.

Explicit Allocation/Deallocation: There is one case in which explicit deallocation is very easy to handle. This is when all the objects being allocated are of the same size. A large contiguous portion of memory is used for the heap, and we partition this into *blocks* of size b, where b is the size of each object. For each unallocated block, we use one word of the block to act as a *next* pointer, and simply link these blocks together in linked list, called the *available space list*. For each `new` request, we extract a block from the available space list and for each `delete` we return the block to the list.

If the records are of different sizes then things become much trickier. We will partition memory into blocks of varying sizes. Each block (allocated or available) contains information indicating how large it is. Available blocks are linked together to form an available space list. The main questions are: (1) what is the best way to allocate blocks for each `new` request, and (2) what is the fastest way to deallocate blocks for each `delete` request.

The biggest problem in such systems is the fact that after a series of allocation and deallocation requests the memory space tends to become *fragmented* into small blocks of memory. This is called *external fragmentation*, and is inherent to all dynamic memory allocators. Fragmentation increases the memory allocation system's running time by increasing the size of the available space list, and when a request comes for a large block, it may not be possible to satisfy this request, even though there is enough total memory available in these small blocks. Observe that it is not usually feasible to compact memory by moving fragments around. This

[1]Copyright, David M. Mount, 2001

is because there may be pointers stored in local variables that point into the heap. Moving blocks around would require finding these pointers and updating them, which is a very expensive proposition. We will consider it later in the context of garbage collection. A good memory manager is one that does a good job of controlling external fragmentation.

Overview: When allocating a block we must search through the list of available blocks of memory for one that is large enough to satisfy the request. The first question is, assuming that there does exist a block that is large enough to satisfy the request, what is the best block to select? There are two common but conflicting strategies:

First fit: Search the blocks sequentially until finding the first block that is big enough to satisfy the request.

Best fit: Search all available blocks and use the smallest one that is large enough to fulfill the request.

Both methods work well in some instances and poorly in others. Some writeups say that first fit is prefered because (1) it is fast (it need only search until it finds a block), (2) if best fit does not exactly fill a request, it tends to produce small "slivers" of available space, which tend to aggravate fragmentation.

One method which is commonly used to reduce fragmentation is when a request is just barely filled by an available block that is slightly larger than the request, we allocate the entire block (more than the request) to avoid the creation of the sliver. This keeps the list of available blocks free of large numbers of tiny fragments which increase the search time. The additional waste of space that results because we allocate a larger block of memory than the user requested is called *internal fragmentation* (since the waste is inside the allocated block).

When deallocating a block, it is important that if there is available storage we should merge the newly deallocated block with any neighboring available blocks to create large blocks of free space. This process is called *merging*. Merging is trickier than one might first imagine. For example, we want to know whether the preceding or following block is available. How would we do this? We could walk along the available space list and see whether we find it, but this would be very slow. We might store a special bit pattern at the end and start of each block to indicate whether it is available or not, but what if the block is allocated that the data contents happen to match this bit pattern by accident? Let us consider the implementation of this scheme in greater detail.

Implementation: The main issues are how to we store the available blocks so we can search them quickly (skipping over used blocks), and how do we determine whether we can merge with neighboring blocks or not. We want the operations to be efficient, but we do not want to use up excessive pointer space (especially in used blocks). Here is a sketch of a solution (one of many possible).

Allocated blocks: For each block of used memory we record the following information. It can all fit in the first word of the block.

`size`: An integer that indicates the size of the block of storage. This includes both the size that the user has requested and the additional space used for storing these extra fields.

`inUse`: A single bit that is set to 1 (true) to indicate that this block is in use.

`prevInUse`: A single bit that is set to 1 (true) if the previous block is in use and 0 otherwise. (Later we will see why this is useful.)

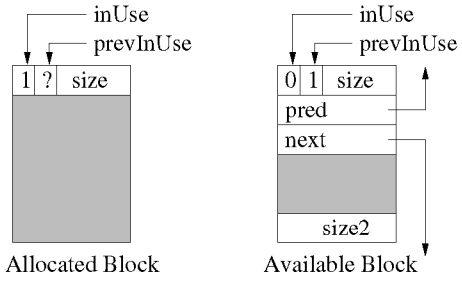

Figure 66: Block structure for dynamic storage allocation.

Available blocks: For an available block we store more information, which is okay because the user is not using this space. These blocks are stored in a doubly-linked circular list, called `avail`.

`size`: An integer that indicates the size of the block of storage (including these extra fields).

`inUse`: A bit that is set to 0 (false) to indicate that this block is not in use.

`prevInUse`: A bit that is set to (true) if the previous block is in use (which should always be true, since we should never have two consecutive unused blocks).

`pred`: A pointer to the predecessor block on the available space list. Note that the available space list is not sorted. Thus the predecessor may be anywhere in the heap.

`next`: A pointer to the next block on the available space list.

`size2`: Contains the same value as `size`. Unlike the previous fields, this size indicator is stored in the *last* word of the block. Since is not within a fixed offset of the head of the block, for block p we access this field as `*(p+p.size-1)`.

Note that available blocks require more space for all this extra information. The system will never allow the creation of a fragment that is so small that it cannot contain all this information.

Block Allocation: To allocate a block we search through the linked list of available blocks until finding one of sufficient size. If the request is about the same size (or perhaps slightly smaller) as the block, we remove the block from the list of available blocks (performing the necessary relinkings) and return a pointer to it. We also may need to update the `prevInUse` bit of the next block since this block is no longer available. Otherwise we split the block into two smaller blocks, return one to the user, and leave the other on the available space list.

Here is pseudocode for the allocation routine. Note that we make use of pointer arithmetic here. The argument `b` is the desired size of the allocation. Because we reserve one word of storage for our own use we increment this value on entry to the procedure. We keep a constant `TOO_SMALL`, which indicates the smallest allowable fragment size. If the allocation would result in a fragment of size less than this value, we return the entire block. The procedure returns a generic pointer to the newly allocated block. The utility function `avail.unlink(p)` simply unlinks block p from the doubly-linked available space list. An example is provided in the figure below. Shaded blocks are available.

Allocate a block of storage

```
(void*) alloc(int b) {                      // allocate block of size b
    b += 1                                  // extra space for system overhead
    p = search available space list for block of size at least b
```

```
    if (p == null) { ...error: insufficient memory...}
    if (p.size - b < TOO_SMALL) {          // allocate whole block
        avail.unlink(p)                    // unlink p from avail space list
        q = p
    }
    else {                                 // split the block
        p.size -= b                        // decrease size by b
        *(p+p.size-1) = p.size             // store size in p's size2 field
        q = p + p.size                     // offset to start of new block
        q.size = b                         // size of new block
        q.prevInUse = 0                    // previous block is unused
    }
    q.inUse = true                         // new block is used
    (q+q.size).prevInUse = true            // adjust prevInUse for following block
    return q
}
```

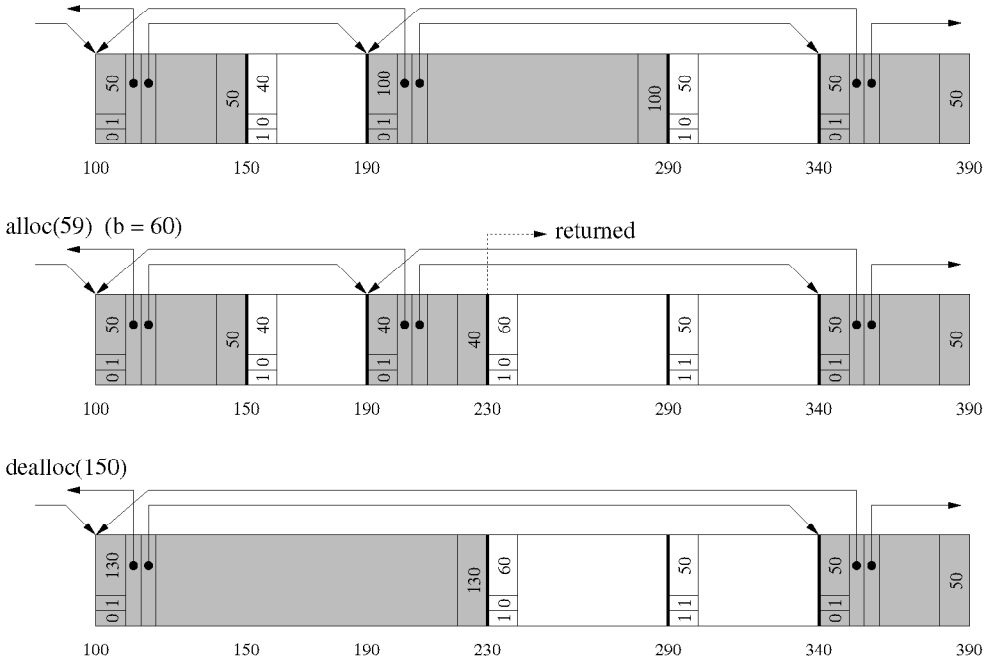

Figure 67: An example of block allocation and deallocation.

Block Deallocation: To deallocate a block, we check whether the next block or the preceding blocks are available. For the next block we can find its first word and check its `inUse` field. For the preceding block we use our own `prevInUse` field. (This is why this field is present). If the previous block is not in use, then we use the size value stored in the last word to find the block's header. If either of these blocks is available, we merge the two blocks and update the header values appropriately. If both the preceding and next blocks are available, then this result in one of these blocks being deleting from the available space list (since we do not want to have two consecutive available blocks). If both the preceding and next blocks are in-use, we simply link this block into the list of available blocks (e.g. at the head of the list). We will leave the details of deallocation as an exercise.

Analysis: There is not much theoretical analysis of this method of dynamic storage allocation. Because the system has no knowledge of the future sequence of allocation and deallocation requests, it is possible to contrive situations in which either first fit or best fit (or virtually any other method you can imagine) will perform poorly. Empirical studies based on simulations have shown that this method achieves utilizations of around 2/3 of the total available storage before failing to satisfy a request. Even higher utilizations can be achieved if the blocks are small on average and block sizes are similar (since this limits fragmentation). A rule of thumb is to allocate a heap that is at least 10 times larger than the largest block to be allocated.