Manual Memory Management

sk and dbtucker

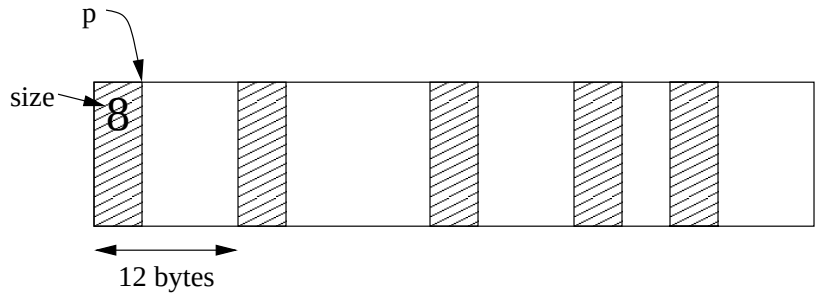
December 1, 2000

Today we'll talk about manual memory management. In C, you can use `malloc()` to allocate a chunk of memory, and `free()` to tell the run-time system to reclaim a chunk of memory. Let's look at these functions in more detail:

1. How does `malloc()` find memory to allocate?
2. What does it mean for the memory to be reclaimed?
3. How does `free()` reclaim memory?

We'll address these questions from last to first.

Say you have a pointer `p` to some memory allocated by `malloc()`. When you call `free(p)`, how does the run-time system know how many cells to reclaim? The system must store the size of the allocated memory somewhere. It uses some space before the pointer—let's just say four bytes, though actual implementations may behave very differently:



So when you ask `malloc()` for eight bytes, the system actually allocates twelve bytes in order to store the size. Now the run-time system knows how much memory to reclaim when you call `free()`—it looks at the four bytes preceding the pointer to determine the size of the object.

What does it mean to restore the memory to the heap? The system maintains a data structure called a *free list*—for now, we can imagine the free list to be a singly-linked list of the chunks of free memory. We'll use the four bytes preceding each chunk to hold the pointer to the next chunk in the free list (for live objects, these four bytes contained the size). For example, say the current

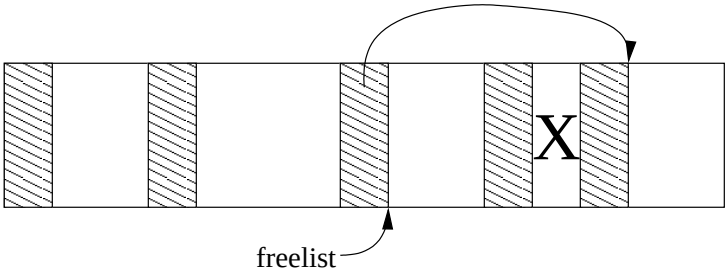


Figure 1: Memory and the free list

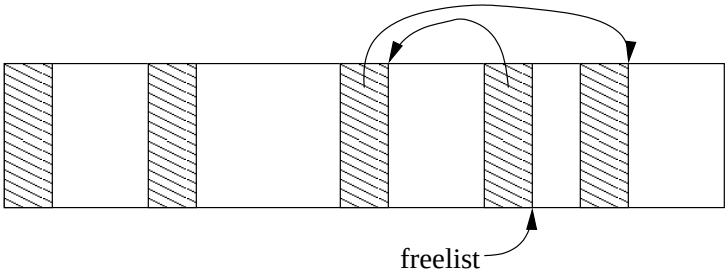


Figure 2: The free list after reclaiming X

free list looks Figure 1. When we free the chunk of memory marked by X, the free list is updated as in Figure 2.

Now let's take a look at allocation. We have been using the same four bytes to represent the size for live objects and the free list pointer for dead objects. In order to do allocation, we'll also need to store the size for dead objects; thus, each chunk of memory has an overhead of eight bytes. When we call `malloc()` to allocate memory for an object, we must chase down the free list until we find a chunk of memory large enough to contain the object plus the eight bytes of information needed by the run-time system. Once we find space for the new object, we update the free list and return a pointer to the allocated memory. For example, say we want to allocate memory for an object four bytes large; we need a chunk of twelve bytes of free memory. Suppose we find a chunk of 100 free bytes (Figure 3). After we have allocated the twelve bytes and updated the free list, the memory looks like Figure 4.

There are several simple algorithms for determining which chunk of memory to allocate; the common ones are First Fit, Best Fit, and Worst Fit. How expensive is, say, Best Fit? Since you have to walk the whole free list, its running time is linear in the size of the free list. How about First Fit? It has a smaller constant on average, but is still linear (you may have to search the entire list to find any fit). Furthermore, all of these algorithms suffer from fragmentation, and may not be able to allocate memory at all, even when there is enough total free memory. In fact, the solution to fragmentation—list coalescing—is

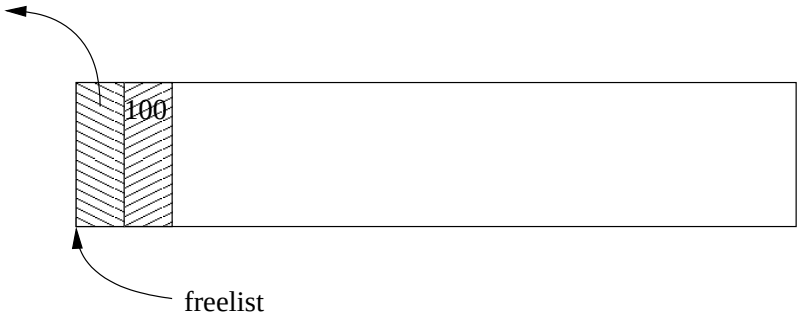


Figure 3: A free chunk of memory

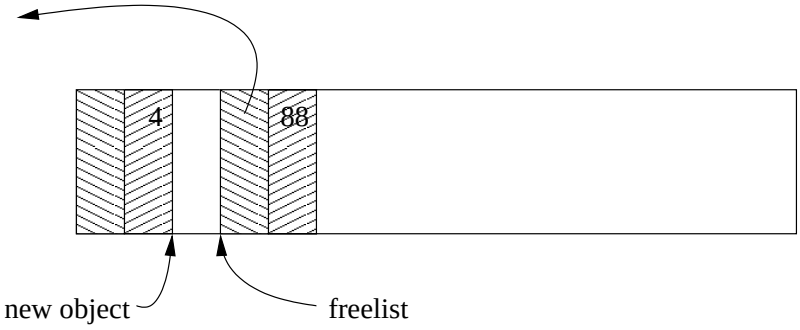


Figure 4: A newly allocated object

difficult because although two chunks of free memory may be next to each other physically, they can be far apart in the free list.

Manual memory management is expensive, even in the trivial case. It gives you no guarantees about safety—you could very well free a reachable object. You can make either `malloc()` cheap or `free()` cheap, but you can't do both.

Moral: Free(dom) has a price.