7.4.4 Reducing Fragmentation

At the beginning of program execution, the heap is one contiguous unit of free space. As the program allocates and deallocates memory, this space is broken up into free and used chunks of memory, and the free chunks need not reside in a contiguous area of the heap. We refer to the free chunks of memory as *holes.* With each allocation request, the memory manager must *place* the requested chunk of memory into a large-enough hole. Unless a hole of exactly the right size is found, we need to *split* some hole, creating a yet smaller hole.

[4] As a machine fetches a word in memory, it is relatively inexpensive to *prefetch* the next several contiguous words of memory as well. Thus, a common memory-hierarchy feature is that a multiword block is fetched from a level of memory each time that level is accessed.

With each deallocation request, the freed chunks of memory are added back to the pool of free space. We *coalesce* contiguous holes into larger holes, as the holes can only get smaller otherwise. If we are not careful, the memory may end up getting *fragmented*, consisting of large numbers of small, noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request, even though there may be sufficient aggregate free space.

Best-Fit and Next-Fit Object Placement

We reduce fragmentation by controlling how the memory manager places new objects in the heap. It has been found empirically that a good strategy for minimizing fragmentation for real-life programs is to allocate the requested memory in the smallest available hole that is large enough. This *best-fit* algorithm tends to spare the large holes to satisfy subsequent, larger requests. An alternative, called *first-fit*, where an object is placed in the first (lowest-address) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.

To implement best-fit placement more efficiently, we can separate free space into *bins*, according to their sizes. One practical idea is to have many more bins for the smaller sizes, because there are usually many more small objects. For example, the Lea memory manager, used in the GNU C compiler `gcc`, aligns all chunks to 8-byte boundaries. There is a bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes. Larger-sized bins are logarithmically spaced (i.e., the minimum size for each bin is twice that of the previous bin), and within each of these bins the chunks are ordered by their size. There is always a chunk of free space that can be extended by requesting more pages from the operating system. Called the *wilderness chunk*, this chunk is treated by Lea as the largest-sized bin because of its extensibility.

Binning makes it easy to find the best-fit chunk.

- If, as for small sizes requested from the Lea memory manager, there is a bin for chunks of that size only, we may take any chunk from that bin.
- For sizes that do not have a private bin, we find the one bin that is allowed to include chunks of the desired size. Within that bin, we can use either a first-fit or a best-fit strategy; i.e., we either look for and select the first chunk that is sufficiently large or, we spend more time and find the smallest chunk that is sufficiently large. Note that when the fit is not exact, the remainder of the chunk will generally need to be placed in a bin with smaller sizes.
- However, it may be that the target bin is empty, or all chunks in that bin are too small to satisfy the request for space. In that case, we simply repeat the search, using the bin for the next larger size(s). Eventually, we either find a chunk we can use, or we reach the "wilderness" chunk, from which we can surely obtain the needed space, possibly by going to the operating system and getting additional pages for the heap.

While best-fit placement tends to improve space utilization, it may not be the best in terms of spatial locality. Chunks allocated at about the same time by a program tend to have similar reference patterns and to have similar lifetimes. Placing them close together thus improves the program's spatial locality. One useful adaptation of the best-fit algorithm is to modify the placement in the case when a chunk of the exact requested size cannot be found. In this case, we use a *next-fit* strategy, trying to allocate the object in the chunk that has last been split, whenever enough space for the new object remains in that chunk. Next-fit also tends to improve the speed of the allocation operation.

Managing and Coalescing Free Space

When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstances, it may also be possible to combine (*coalesce*) that chunk with adjacent chunks of the heap, to form a larger chunk. There is an advantage to doing so, since we can always use a large chunk to do the work of small chunks of equal total size, but many small chunks cannot hold one large object, as the combined chunk could.

If we keep a bin for chunks of one fixed size, as Lea does for small sizes, then we may prefer not to coalesce adjacent blocks of that size into a chunk of double the size. It is simpler to keep all the chunks of one size in as many pages as we need, and never coalesce them. Then, a simple allocation/deallocation scheme is to keep a bitmap, with one bit for each chunk in the bin. A 1 indicates the chunk is occupied; 0 indicates it is free. When a chunk is deallocated, we change its 1 to a 0. When we need to allocate a chunk, we find any chunk with a 0 bit, change that bit to a 1, and use the corresponding chunk. If there are no free chunks, we get a new page, divide it into chunks of the appropriate size, and extend the bit vector.

Matters are more complex when the heap is managed as a whole, without binning, or if we are willing to coalesce adjacent chunks and move the resulting chunk to a different bin if necessary. There are two data structures that are useful to support coalescing of adjacent free blocks:

- *Boundary Tags.* At both the low and high ends of each chunk, whether free or allocated, we keep vital information. At both ends, we keep a free/used bit that tells whether or not the block is currently allocated (used) or available (free). Adjacent to each free/used bit is a count of the total number of bytes in the chunk.

- *A Doubly Linked, Embedded Free List.* The free chunks (but not the allocated chunks) are also linked in a doubly linked list. The pointers for this list are within the blocks themselves, say adjacent to the boundary tags at either end. Thus, no additional space is needed for the free list, although its existence does place a lower bound on how small chunks can get; they must accommodate two boundary tags and two pointers, even if the object is a single byte. The order of chunks on the free list is left

unspecified. For example, the list could be sorted by size, thus facilitating best-fit placement.

Example 7.10: Figure 7.17 shows part of a heap with three adjacent chunks, A, B, and C. Chunk B, of size 100, has just been deallocated and returned to the free list. Since we know the beginning (left end) of B, we also know the end of the chunk that happens to be immediately to B's left, namely A in this example. The free/used bit at the right end of A is currently 0, so A too is free. We may therefore coalesce A and B into one chunk of 300 bytes.

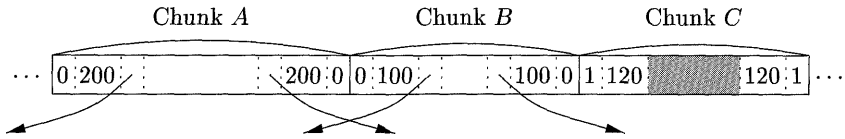

Figure 7.17: Part of a heap and a doubly linked free list

It might be the case that chunk C, the chunk immediately to B's right, is also free, in which case we can combine all of A, B, and C. Note that if we always coalesce chunks when we can, then there can never be two adjacent free chunks, so we never have to look further than the two chunks adjacent to the one being deallocated. In the current case, we find the beginning of C by starting at the left end of B, which we know, and finding the total number of bytes in B, which is found in the left boundary tag of B and is 100 bytes. With this information, we find the right end of B and the beginning of the chunk to its right. At that point, we examine the free/used bit of C and find that it is 1 for used; hence, C is not available for coalescing.

Since we must coalesce A and B, we need to remove one of them from the free list. The doubly linked free-list structure lets us find the chunks before and after each of A and B. Notice that it should not be assumed that physical neighbors A and B are also adjacent on the free list. Knowing the chunks preceding and following A and B on the free list, it is straightforward to manipulate pointers on the list to replace A and B by one coalesced chunk. □

Automatic garbage collection can eliminate fragmentation altogether if it moves all the allocated objects to contiguous storage. The interaction between garbage collection and memory management is discussed in more detail in Section 7.6.4.