

## 12.4 Memory Management

Most of the data structure implementations described in this book store and access objects all of the same size, such as integers stored in a list or a tree. A few simple methods have been described for storing variable size records in an array or a stack. This section discusses memory management techniques for the general problem of handling space requests of variable size.

The basic model for memory management is that we have a (large) block of contiguous memory locations, which we will call the **memory pool**. Periodically, memory requests are issued for some amount of space in the pool. The memory manager must find a contiguous block of locations of at least the requested size from somewhere within the memory pool. Honoring such a request is called a **memory allocation**. The memory manager will typically return some piece of information that permits the user to recover the data that were just stored. This piece of information is called a **handle**. Previously allocated memory may be returned to the memory manager at some future time. This is called a **memory deallocation**. We can define an ADT for the memory manager as follows:

```
// Memory Manager abstract class
class MemManager {
public:
    // Store a record and return the handle
    virtual MemHandle insert(void* space, int length) =0;
    // Release the space associated with a record
    virtual void release(MemHandle h) =0;
    // Get back a copy of a stored record
    virtual int get(void* space, MemHandle h) =0;
};
```

In this ADT, MemManager's client provides space to hold the information to be stored or retrieved (in parameter space). In method `insert`, the client tells the memory manager the length of the information to be stored. This ADT assumes that the memory manager will remember the length of the information associated with a given handle, thus method `get` does not include a length parameter but instead returns the length of the information actually stored.

If all inserts and releases follow a simple pattern, such as last requested, first released (stack order), or first requested, first released (queue order), then memory management is fairly easy. We are concerned in this section with the general case where blocks of any size may be requested and released in any order. This is known as **dynamic storage allocation**. One example of dynamic memory allocation is managing free store for a compiler's runtime environment, such as the system-level `new` and `delete` operations in C++. Another example is managing main memory



**Figure 12.12** Dynamic storage allocation model. Memory is made up of a series of variable-size blocks, some allocated and some free. In this example, shaded areas represent memory currently allocated and unshaded areas represent unused memory available for future allocation.

in a multitasking operating system. Here, a program may require a certain amount of space, and the memory manager must keep track of which programs are using which parts of the main memory. Yet another example is the file manager for a disk drive. When a disk file is created, expanded, or deleted, the file manager must allocate or deallocate disk space.

A block of memory or disk space managed in this way is sometimes referred to as a **heap**. In this context, the term “heap” is being used in a different way than the heap data structure discussed in other chapters. Here “heap” refers to the free memory accessed by a dynamic memory management scheme.

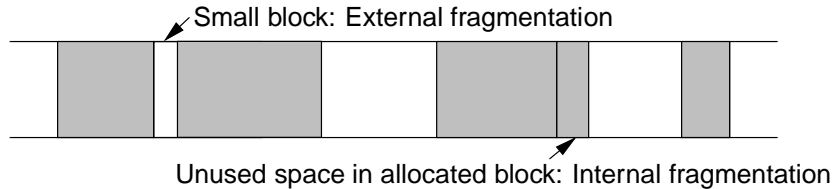
In the rest of this section, we first study techniques for dynamic memory management. We then tackle the issue of what to do when no single block of memory in the memory pool is large enough to honor a given request.

### 12.4.1 Dynamic Storage Allocation

For the purpose of dynamic storage allocation, we view memory as a single array broken into a series of variable-size blocks, where some of the blocks are **free** and some are **reserved** or already allocated. The free blocks are linked together to form a freelist for servicing future memory requests. Figure 12.12 illustrates the situation that can arise after a series of memory allocations and deallocations.

When a memory request is received by the memory manager, some block on the freelist must be found that is large enough to service the request. If no such block is found, then the memory manager must resort to a **failure policy** such as discussed in Section 12.4.2.

If there is a request for  $m$  words, and no block exists of exactly size  $m$ , then a larger block must be used instead. One possibility in this case is that the entire block is given away to the memory allocation request. This may be desirable when the size of the block is only slightly larger than the request. Alternatively, for a free block of size  $k$ , with  $k > m$ , up to  $k - m$  space may be retained by the memory manager to form a new free block, while the rest is used to service the request.



**Figure 12.13** An illustration of internal and external fragmentation.

There are two types of fragmentation encountered in dynamic memory management. **External fragmentation** occurs when the memory requests create lots of small free blocks, no one of which is useful for servicing typical requests. **Internal fragmentation** occurs when more than  $m$  words are allocated to a request for  $m$  words, wasting free storage. This is equivalent to the internal fragmentation that occurs when files are allocated in multiples of the cluster size. The difference between internal and external fragmentation is illustrated by Figure 12.13.

Some memory management schemes sacrifice space to internal fragmentation to make memory management easier (and perhaps reduce external fragmentation). For example, external fragmentation does not happen in file management systems that allocate file space in clusters. Another example is the **buddy method** described later in this section.

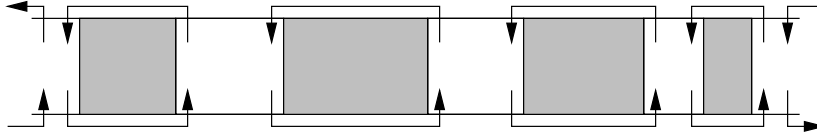
The process of searching the memory pool for a block large enough to service the request, possibly reserving the remaining space as a free block, is referred to as a **sequential fit** method.

### Sequential-Fit Methods

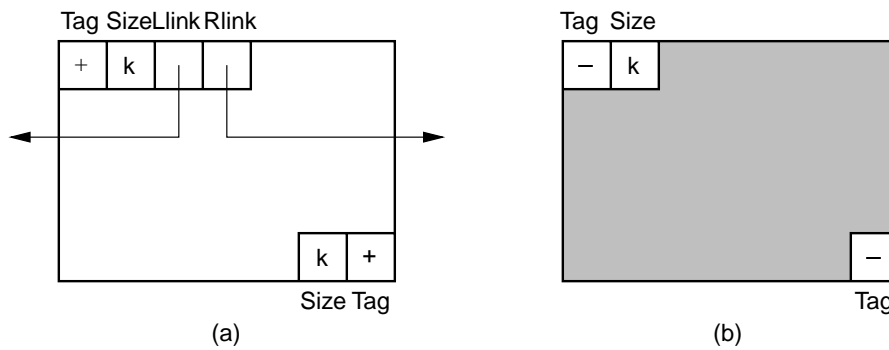
Sequential-fit methods attempt to find a “good” block to service a storage request. The three sequential-fit methods described here assume that the free blocks are organized into a doubly linked list, as illustrated by Figure 12.14.

There are two basic approaches to implementing the freelist. The simpler approach is to store the freelist separately from the memory pool. In other words, a simple linked-list implementation such as described in Chapter 4 can be used, where each node of the linked list corresponds to a single free block in the memory pool. This is fine if there is space available for the linked list nodes separate from the memory pool.

The second approach to storing the freelist is more complicated but saves space. Since the free space is free, it can be used by the memory manager to help it do its job; that is, the memory manager temporarily “borrows” space within the free blocks to maintain its doubly linked list. To do so, each unallocated block must be large enough to hold these pointers. In addition, it is usually worthwhile to



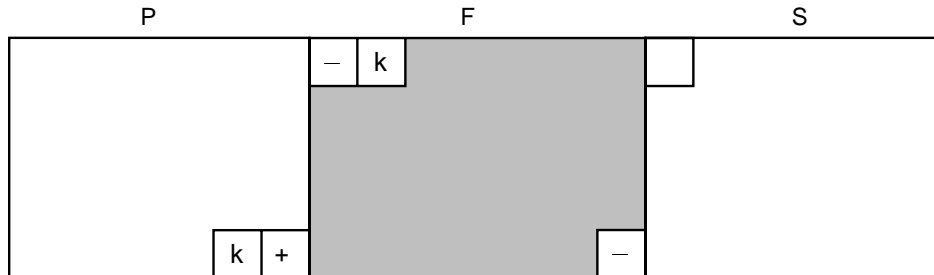
**Figure 12.14** A doubly linked list of free blocks as seen by the memory manager. Shaded areas represent allocated memory. Unshaded areas are part of the freelist.



**Figure 12.15** Blocks as seen by the memory manager. Each block includes additional information such as freelist link pointers, start and end tags, and a size field. (a) The layout for a free block. The beginning of the block contains the tag bit field, the block size field, and two pointers for the freelist. The end of the block contains a second tag field and a second block size field. (b) A reserved block of  $k$  bytes. The memory manager adds to these  $k$  bytes an additional tag bit field and block size field at the beginning of the block, and a second tag field at the end of the block.

let the memory manager add a few bytes of space to a reserved block for its own purposes. In other words, a request for  $m$  bytes of space might result in slightly more than  $m$  bytes being allocated by the memory manager, with the extra bytes used by the memory manager itself rather than the requester. We will assume that all memory blocks are organized as shown in Figure 12.15, with space for tags and linked list pointers. Here, free and reserved blocks are distinguished by a tag bit at both the beginning and the end of the block. In addition, both free and reserved blocks have a size indicator immediately after the tag bit at the beginning of the block to indicate how large the block is. Free blocks have a second size indicator immediately preceding the tag bit at the end of the block. Finally, free blocks have left and right pointers to their neighbors in the free block list.

The information fields associated with each block permit the memory manager to allocate and deallocate blocks as needed. When a request comes in for  $m$  words of storage, the memory manager searches the linked list of free blocks until it finds



**Figure 12.16** Adding block  $F$  to the freelist. The word immediately preceding the start of  $F$  in the memory pool stores the tag bit of the preceding block  $P$ . If  $P$  is free, merge  $F$  into  $P$ . We find the beginning of  $F$  by using  $F$ 's size field. Likewise, the word following the end of  $F$  is the tag field for block  $S$ . If  $S$  is free, merge it into  $F$ .

a “suitable” block for allocation. How it determines which block is suitable will be discussed below. If the block contains exactly  $m$  words (plus space for the tag and size fields), then it is removed from the freelist. If the block (of size  $k$ ) is large enough, then the remaining  $k - m$  words are reserved as a block on the freelist, in the current location.

When a block  $F$  is freed, it must be merged into the freelist. If we do not care about merging adjacent free blocks, then this is a simple insertion into the doubly linked list of free blocks. However, we would like to merge adjacent blocks, since this allows the memory manager to serve requests of the largest possible size. Merging is easily done due to the tag and size fields stored at the ends of each block, as illustrated by Figure 12.16. The memory manager first checks the unit of memory immediately preceding block  $F$  to see if the preceding block (call it  $P$ ) is also free. If it is, then the memory unit before  $P$ 's tag bit stores the size of  $P$ , thus indicating the position for the beginning of the block in memory.  $P$  can then simply have its size extended to include block  $F$ . If block  $P$  is not free, then we just add block  $F$  to the freelist. Finally, we also check the bit following the end of block  $F$ . If this bit indicates that the following block (call it  $S$ ) is free, then  $S$  is removed from the freelist and the size of  $F$  is extended appropriately.

We now consider how a “suitable” free block is selected to service a memory request. To illustrate the process, assume there are four blocks on the freelist of sizes 500, 700, 650, and 900 (in that order). Assume that a request is made for 600 units of storage. For our examples, we ignore the overhead imposed for the tag, link, and size fields discussed above.

The simplest method for selecting a block would be to move down the free block list until a block of size at least 600 is found. Any remaining space in this

block is left on the freelist. If we begin at the beginning of the list and work down to the first free block at least as large as 600, we select the block of size 700. Since this approach selects the first block with enough space, it is called **first fit**. A simple variation that will improve performance is, instead of always beginning at the head of the freelist, remember the last position reached in the previous search and start from there. When the end of the freelist is reached, search begins again at the head of the freelist. This modification reduces the number of unnecessary searches through small blocks that were passed over by the last request.

There is a potential disadvantage to first fit: It may “waste” larger blocks by breaking them up, and so they will not be available for large requests later. A strategy that avoids using large blocks unnecessarily is called **best fit**. Best fit looks at the entire list and picks the smallest block that is at least as large as the request (i.e., the “best” or closest fit to the request). Continuing with the preceding example, the best fit for a request of 600 units is the block of size 650, leaving a remainder of size 50. Best fit has the disadvantage that it requires that the entire list be searched. Another problem is that the remaining portion of the best-fit block is likely to be small, and thus useless for future requests. In other words, best fit tends to maximize problems of external fragmentation while it minimizes the chance of not being able to service an occasional large request.

A strategy contrary to best fit might make sense because it tends to minimize the effects of external fragmentation. This is called **worst fit**, which always allocates the largest block on the list hoping that the remainder of the block will be useful for servicing a future request. In our example, the worst fit is the block of size 900, leaving a remainder of size 300. If there are a few unusually large requests, this approach will have less chance of servicing them. If requests generally tend to be of the same size, then this may be an effective strategy. Like best fit, worst fit requires searching the entire freelist at each memory request to find the largest block. Alternatively, the freelist can be ordered from largest to smallest free block, possibly by using a priority queue implementation.

Which strategy is best? It depends on the expected types of memory requests. If the requests are of widely ranging size, best fit may work well. If the requests tend to be of similar size, with rare large and small requests, first or worst fit may work well. Unfortunately, there are always request patterns that one of the three sequential fit methods will service, but which the other two will not be able to service. For example, if the series of requests 600, 650, 900, 500, 100 is made to a freelist containing blocks 500, 700, 650, 900 (in that order), the requests can all be serviced by first fit, but not by best fit. Alternatively, the series of requests 600, 500, 700, 900 can be serviced by best fit but not by first fit on this same freelist.