- It is easy to manage free space when the space you are managing is divided into fixed-sized units; in such a case, you just keep a list of these fixed-sized units; when a client requests one of them, return the first entry.
- free-space management becomes more difficult when the free space you are managing consists of variable-sized units; this arises in a user-level memory-allocation library (as in malloc() and free()) and in an OS managing physical memory when using segmentation to implement virtual memory.
    - In either case, the problem that exists is known as external fragmentation:
        - the free space gets chopped into little pieces of different sizes and is thus fragmented;
        - subsequent requests may fail because there is no single contiguous space that can satisfy the request, even though the total amount of free space exceeds the size of the request.

How should free space be managed, when satisfying variable-sized requests? What strategies can be used to minimize fragmentation? What are the time and space overheads of alternate approaches?

Low-level Mechanisms

- Splitting and Coalescing
    - A free list contains a set of elements that describe the free space still remaining in the heap.
    - Assume we have a request for just a single byte of memory. In this case, the allocator will perform an action known as splitting: it will find a free chunk of memory that can satisfy the request and split it into two. The first chunk will return to the caller; the second chunk will remain on the list.
        - Splitting is done so that more memory remains available for future allocations.
    - Coalescing is the act of joining adjacent free blocks into a single larger free block of memory.
        - when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly freed space sits right next to one (or two, as in this example) existing free chunks, merge them into a single larger free chunk.
- Most allocators store a little bit of extra information in a header block which is kept in memory, usually just before the handed-out chunk of memory.

Free List:

- Concept of a Free List: The free list is a conceptual tool used for managing free memory chunks within a heap. It differs from typical lists because it is built inside the free space of the memory it manages.
- Construction within Free Space: The key challenge in building a free list is that it must be embedded within the very free space it manages. This is necessary because memory allocation functions like malloc() are not available within the context of a memory management library

- Node Structure: Each node in the free list typically contains information about the size of the free chunk and a pointer to the next node. This structure allows the list to effectively track various chunks of free memory.
- Heap Initialization for Free List: The free list is initialized by setting up a starting node that represents the entire heap. The size of this initial node is adjusted to account for the metadata storage required by the node itself.
- Memory Allocation and Free List Update: When memory is allocated, the free list is updated. A portion of a free chunk is allocated for use, and the remaining part continues to exist as a free chunk in the list, with adjusted size information.
- Memory Deallocation and Free List Management: Upon deallocation (freeing of memory), the freed chunk is re-inserted into the free list. This process may involve updating the list structure to reflect the newly freed space.
- Fragmentation and Coalescing: One of the challenges in managing a free list is fragmentation. As memory is allocated and freed, the heap can become fragmented. To address this, coalescing is used, where adjacent free chunks are merged to form larger, contiguous free spaces, thus optimizing memory usage and reducing fragmentation.

Basic Strategies

- Let us go over some basic strategies for managing free space. The ideal allocator is both fast and minimizes fragmentation. Unfortunately, because the stream of allocation and free requests can be arbitrary (after all, they are determined by the programmer), any particular strategy can do quite badly given the wrong set of inputs
- The best fit strategy is quite simple: first, search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates; this is the so called best-fit chunk (it could be called smallest fit too). One pass through the free list is enough to find the correct block to return.
    - However, there is a cost; naive implementations pay a heavy performance penalty when performing an exhaustive search for the correct free block.
- The worst fit approach is the opposite of best fit; find the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list.
    - Worst fit tries to thus leave big chunks free instead of lots of small chunks that can arise from a best-fit approach. Once again, however, a full search of free space is required, and thus this approach can be costly
- The first fit method simply finds the first block that is big enough and returns the requested amount to the user. As before, the remaining free space is kept free for subsequent requests
    - First fit has the advantage of speed — no exhaustive search of all the free spaces are necessary — but sometimes pollutes the beginning of the free list with small objects. Thus, how the allocator manages the free list's order becomes an issue. One approach is to use address-based ordering; by keeping the list ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.
- Instead of always beginning the first-fit search at the beginning of the list, the next fit algorithm keeps an extra pointer to the location within the list where one was looking

last. The idea is to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list.

Other Approaches

- Segregated Lists:
    - Concept: This approach involves maintaining separate lists for managing objects of specific, frequently requested sizes, with other requests being handled by a general memory allocator.
    - Benefits: It reduces fragmentation and speeds up allocation and deallocation for common sizes, as these operations don't require a complex search.
    - Challenges: Determining the optimal amount of memory to allocate to these specialized pools versus a general pool.
    - Example: The slab allocator, used in the Solaris kernel, which pre-allocates object caches for frequently requested kernel objects.
- Slab Allocator:
    - Functionality: It keeps free objects in a pre-initialized state, reducing the overhead of frequent initialization and destruction.
    - Efficiency: This approach is notably efficient in managing frequent allocations and deallocations of objects of a fixed size.
- Buddy Allocation:
    - Concept: This system divides the memory into blocks, recursively halving the free space to find a block large enough for the request.
    - Coalescing Process: When memory is freed, the allocator checks if the adjacent "buddy" block is free. If so, it merges them, continuing recursively to optimize space.
    - Fragmentation: While effective in coalescing, it can suffer from internal fragmentation due to the restriction of allocating blocks of power-of-two sizes.
- Scalability Issues and Advanced Data Structures:
    - Problem: Many traditional approaches don't scale well, particularly in terms of list searching.
    - Solution: Advanced allocators use more complex data structures like balanced binary trees, splay trees, or partially-ordered trees for better performance.
- Multiprocessor Systems and Multithreaded Workloads:
    - Modern Systems Consideration: Allocators are being designed to perform efficiently on multiprocessor systems and in multithreaded environments.
    - Examples: Berger et al. and Evans have conducted notable work in this area, developing allocators optimized for multiprocessor-based systems.

Summary:

This chapter explores the complexities of managing free space in memory allocation, especially when dealing with variable-sized units, highlighting the challenge of external fragmentation where free space is unevenly divided. It introduces low-level mechanisms like splitting, where larger memory chunks are divided to satisfy smaller requests, and coalescing, where adjacent free blocks are merged to optimize space.

The chapter also discusses the implementation and management of a free list, a structure embedded within the heap to track and manage free memory chunks. Various allocation

strategies are examined, including best fit, worst fit, first fit, and next fit, each with its own approach to minimizing waste and fragmentation. Additionally, the chapter delves into more advanced and specialized methods like segregated lists, slab allocators, and buddy allocation, addressing their unique efficiencies and challenges. The considerations for scalability and multiprocessor systems are also highlighted, indicating the evolving nature of memory allocation strategies in response to modern computing demands.