Chapter 17: Free-Space Management

This chapter explores the fundamental challenge of managing free memory, a problem faced by both user-level memory allocation libraries (like `malloc`) and operating systems (when managing physical memory with segmentation). The core difficulty arises when managing **variable-sized** units of memory.

> **The Crux of the Problem: How to Manage Free Space?**
>
> 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?

The primary issue with variable-sized allocation is **external fragmentation**. This occurs when free memory is broken into many small, non-contiguous chunks. Even if the total free space is large, a request for a large *contiguous* block may fail.

1. Low-Level Mechanisms

Before discussing allocation policies, we must understand the basic mechanisms used by nearly all memory allocators.

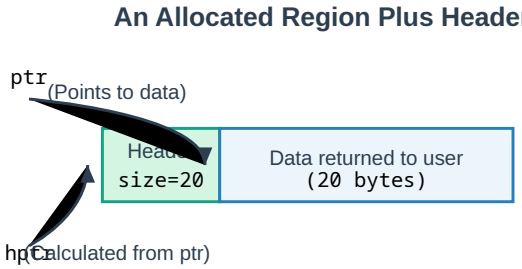
1.1. Splitting and Coalescing

- **Splitting:** When a memory request is smaller than an available free chunk, the allocator **splits** the chunk. It returns a piece of the requested size to the user and keeps the smaller, remaining piece on the free list.
- **Coalescing:** When a user frees a block of memory, the allocator should check if the adjacent memory chunks are also free. If they are, it **coalesces** (merges) them into a single, larger free chunk. This is crucial for combating fragmentation and recreating large free blocks.

1.2. Tracking The Size of Allocated Regions

The `free(ptr)` call doesn't require a size argument. How does the library know how much memory to free?

- **The Header Block:** The allocator stores a small amount of metadata in a **header** just before the chunk of memory it returns to the user.
- **Contents:** This header typically contains the size of the allocated region and often a "magic number" for sanity checks.
- **Operation:** When `free(ptr)` is called, the library calculates the header's location using pointer arithmetic (e.g., `hptr = (header_t *)ptr - 1`), reads the size, and knows exactly how much memory to reclaim.



1.3. Embedding a Free List

The allocator can't call `malloc()` to create nodes for its own free list. Instead, it must embed the list's data structures *within the free space itself.*

- **Mechanism:** A free block of memory is large enough to hold a small `node`

structure (containing its size and a pointer to the next free block) plus the rest of the free space.
- When a block is freed, it is repurposed to hold this `node` information and added to the list.

1.4. Growing The Heap

If the allocator runs out of memory, it can request more from the OS.

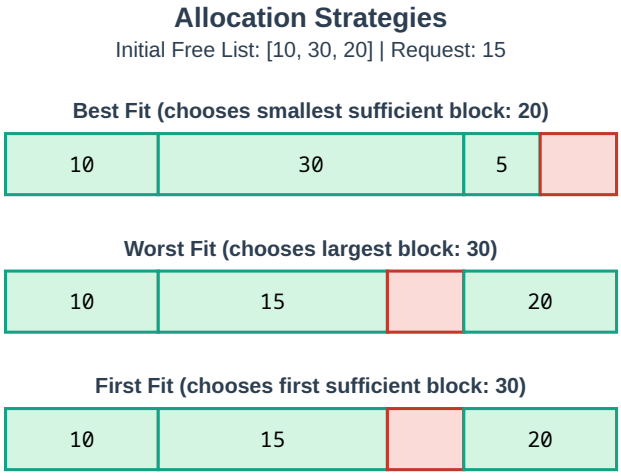
- **Mechanism:** It uses a system call (like `sbrk()` in UNIX) to grow the heap segment. The OS maps new physical pages into the process's address space, and the allocator can then use this new space to satisfy requests.

---

2. Basic Allocation Strategies (Policies)

These policies determine *which* free chunk to use for an allocation request.

- **Best Fit:**
  - **Policy:** Search the entire free list and choose the smallest chunk that is large enough to satisfy the request.
  - **Goal:** Tries to minimize wasted space by leaving the smallest possible leftover chunk (or no leftover chunk).
  - **Drawback:** Can be slow due to the exhaustive search and can lead to many tiny, unusable free fragments.
- **Worst Fit:**
  - **Policy:** Search the entire free list and choose the *largest* available chunk.
  - **Goal:** Tries to leave large, useful chunks of free space remaining after a split.
  - **Drawback:** Also slow due to the exhaustive search and studies show it often performs poorly, leading to faster fragmentation.
- **First Fit:**
  - **Policy:** Search the free list from the beginning and choose the *first* block that is large enough.
  - **Goal:** Speed. It avoids an exhaustive search.
  - **Drawback:** Can lead to a collection of small fragments at the beginning of the list. Keeping the list sorted by address can help with performance and coalescing.
- **Next Fit:**
  - **Policy:** A variant of First Fit. It starts each search from where the last search left off, rather than from the beginning of the list.
  - **Goal:** Spreads the search for free space more evenly throughout the list.
  - **Performance:** Similar to First Fit in speed.

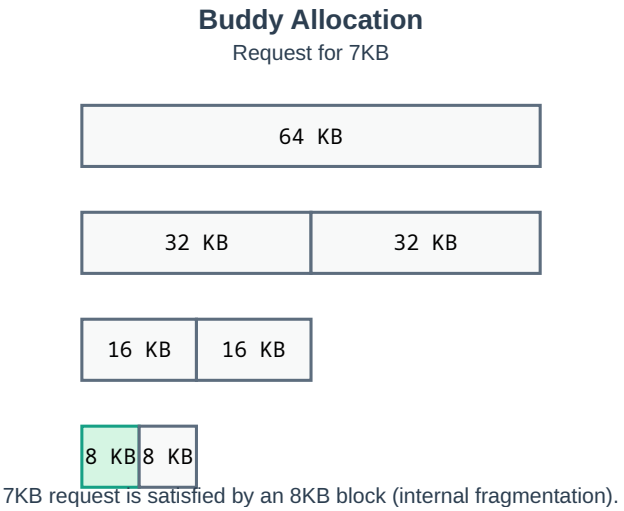


3. Other Approaches

More advanced allocators use sophisticated techniques to improve performance and reduce fragmentation.

- **Segregated Lists:**
  - **Idea:** Maintain separate free lists for different, popular object sizes.
  - **Mechanism:** If a request is for a popular size (e.g., 32 bytes), it is serviced from the dedicated 32-byte list. All other sizes go to a general-purpose allocator.
  - **Benefits:** Extremely fast for common sizes and eliminates internal fragmentation for those sizes.
  - **Slab Allocator:** A famous implementation used in OS kernels (e.g., Solaris, Linux). It manages "slabs" of memory for specific kernel objects, requests memory from a general allocator when its caches are low, and keeps freed objects pre-initialized to speed up future allocations.
- **Buddy Allocation:**
  - **Idea:** A memory region is recursively split into power-of-two "buddy" pairs until a block of the right size is found.
  - **Mechanism:** The entire heap is a power of two (e.g., 64KB). A request for 7KB would cause the 64KB block to be split into two 32KB buddies, one of those into two 16KB buddies, and one of those into two 8KB buddies. One of the 8KB blocks is then allocated.
  - **Benefit:** Coalescing is extremely fast. When a block is freed, the allocator can instantly calculate its buddy's address and check if it's

also free. If so, they are merged, and the process repeats up the tree.
  - **Drawback:** Suffers from **internal fragmentation**, as requests are rounded up to the next power of two.



4. Summary

Free-space management is a classic systems problem with inherent trade-offs.

- Simple strategies like **First Fit** are fast but can be inefficient.
- **Best Fit** aims to reduce waste but is slow.
- Advanced techniques like **Segregated Lists** and **Buddy Allocation** offer better performance and clever solutions to coalescing but introduce their own complexities.
- Modern allocators often use a hybrid of these approaches, along with complex data structures (like balanced trees) and per-CPU caches to achieve scalability on multi-threaded, multi-processor systems.