Build Your Own Memory Allocator: Design Document

Overview

This system implements custom malloc and free functions for dynamic memory allocation, a fundamental service provided by the C standard library. The key architectural challenge is designing a data structure and algorithm to manage a finite heap memory region, efficiently satisfying allocation requests of varying sizes while minimizing wasted space (fragmentation) and maximizing performance for concurrent programs.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** Milestone 1 (Basic Allocator)

1. Context and Problem Statement

Dynamic memory allocation — the ability for a running program to request additional memory from the operating system during its execution — is a fundamental capability in systems programming. Languages like C, C++, Rust, and Zig expose this directly through functions like `malloc` and `free`. At first glance, the programmer's mental model is simple: ask for memory, receive a pointer; return it when done. However, the system that fulfills these requests is one of the most complex and performance-critical components in any runtime environment.

This design document outlines the architecture for a custom memory allocator, a system that manages a region of virtual memory called the **heap**. The primary challenge is designing a data structure and set of algorithms that can efficiently satisfy an unpredictable sequence of allocation and deallocation requests of varying sizes, all while minimizing wasted space and maximizing throughput, especially in concurrent programs. This is analogous to designing the inventory management system for a constantly changing warehouse.

The Problem: The Heap as an Unguided Warehouse

Imagine you are given a vast, empty warehouse — a single, contiguous space — and told to manage inventory for thousands of different customers. Each customer (a `malloc` call) arrives with a request: "I need space to store X boxes." You must find an empty spot in the warehouse, mark it as occupied, and give the customer the exact address. Later, customers (`free` calls) may return, saying, "I'm done with the space starting at address Y," making that area available again for future requests.

The raw heap, as provided by the operating system via `sbrk` or `mmap`, is this unguided warehouse. It is a large, flat address space with **no inherent structure**. The OS merely promises that the addresses are yours to use; it provides no mechanism to track which portions are in use and which are free, nor to find a suitable free spot for a new request. This is the allocator's core responsibility: to impose order on this wilderness.

> **Key Insight:** The operating system provides memory in large, page-sized chunks (typically 4KB). The allocator's job is to **subdivide** these pages into smaller, variable-sized blocks to fulfill program requests, and to **track** the state of each subdivision.

Without a manager, the warehouse would descend into chaos. You might accidentally assign the same spot to two customers (a **catastrophic bug**), or be unable to find a free spot even when total free space exists because it's fragmented into unusably small pieces.

The primary enemy in this domain is **fragmentation** — wasted space that cannot be used to satisfy requests. There are two distinct types:

Fragmentation Type	Analogy	Cause in Memory	Result
Internal Fragmentation	Using a large shelf for a small item, leaving unused space *inside* the allocated shelf.	Allocating a block slightly larger than requested (e.g., for alignment or metadata overhead).	Wasted space within *allocated* blocks. The program cannot use this space.
External Fragmentation	Having many small, scattered empty shelves, but no single empty shelf large enough for a new, large item.	Free and allocated blocks becoming interleaved, creating free space that is not contiguous.	Total free space is sufficient, but no single free block can satisfy a request, causing allocation failure.

External fragmentation is the more insidious problem. Consider this heap state:

```
[Alloc 64][Free 32][Alloc 128][Free 64][Alloc 32]
```

Total free space is 96 bytes (32+64). However, a request for 80 bytes will fail because no single free block is large enough. The allocator must actively combat this through strategies like **coalescing** (merging adjacent free blocks) and intelligent placement policies.

The allocator's performance is also critical. `malloc` and `free` are called potentially millions of times in a program's lifetime. An O(n) scan of all blocks for every allocation would cripple performance. The design must support fast lookup, typically O(1) for common cases, while managing the complex trade-offs between speed, fragmentation, and implementation complexity.

Existing Approaches & Mental Models

Over decades, computer scientists have developed numerous strategies for heap management, each with different trade-offs. Using our warehouse analogy, we can understand each approach's philosophy.

1. Bump Allocator (Sequential Allocation)

- **Mental Model:** A single, moving pointer. When a customer arrives, you allocate space at the current pointer location and bump the pointer forward by the requested size. To "free" memory, you do nothing individually; you simply reset the pointer to the beginning to free everything at once.
- **Characteristics:** Extremely fast (O(1) allocation), zero overhead for tracking free blocks. However, it cannot reclaim individual freed blocks, making it useless for general-purpose allocation. It's ideal for temporary, phase-based memory usage (e.g., compiling a function).
- **Trade-off:** Speed vs. Flexibility.

2. Free List Allocator (Explicit List)

- **Mental Model:** A ledger listing all empty shelves (free blocks), linked together in a list. To allocate, you scan the list for a shelf big enough. To free, you mark the shelf as empty and add it back to the list. You can also merge adjacent empty shelves (coalescing) to combat fragmentation.
- **Strategies:**
  - **First-Fit:** Take the first shelf in the list that fits. Fast but can lead to "splintering" at the front of the list.
  - **Best-Fit:** Find the shelf that fits the request most tightly. Reduces wasted space but requires a full list scan, and leaves tiny, often unusable "slivers."
  - **Worst-Fit:** Use the largest available shelf. The idea is to leave a large remainder, but it often performs poorly in practice.
- **Trade-off:** Search time vs. fragmentation characteristics.

3. Segregated Free Lists (Size-Class Allocator)

- **Mental Model:** Organize your warehouse into sections by shelf size (e.g., a 16-byte section, a 32-byte section). When a request comes in, you round it up to the nearest standard size and check only that section's free list. If empty, you carve a new block from a large, unallocated region (the "wilderness").
- **Characteristics:** O(1) allocation for common sizes, excellent locality. However, it introduces internal fragmentation (rounding up) and requires managing multiple free lists. This is the core of many high-performance allocators (like `jemalloc`, `tcmalloc`).
- **Trade-off:** Internal fragmentation for speed and reduced external fragmentation.

4. Buddy Allocator

- **Mental Model:** Only allow blocks in power-of-two sizes. The warehouse is initially one large block. To allocate a 64-byte block, you recursively split a larger block (128, 256, etc.) until you get a 64-byte block. When freed, if its "buddy" (the block it was split from) is also free, they merge back together.
- **Characteristics:** Extremely fast coalescing (just check the buddy), minimal external fragmentation. Suffers from severe internal fragmentation (a 65-byte request needs a 128-byte block). Common in kernel memory management.
- **Trade-off:** Predictable, fast coalescing vs. high internal fragmentation.

The following table summarizes the high-level trade-offs of these approaches as they relate to our project's milestones:

Approach (Milestone)	Allocation Speed	Fragmentation Control	Implementation Complexity	Best For
Bump (M1 Basic)	O(1)	None (no reuse)	Trivial	Arena/scratch allocation, one-shot phases
Explicit Free List (M2)	O(# free blocks)	Good (with coalescing)	Moderate	Learning fundamental block management
Segregated Lists (M3)	O(1) for sized classes	Very Good (external), Moderate (internal)	High	General-purpose, high-performance allocation
Buddy (Not in Milestones)	O(log max size)	Excellent (external), Poor (internal)	High	Kernel/page-level allocation

> **Design Principle:** There is no single "best" allocator. The optimal design is dictated by the **workload**: the distribution of allocation sizes, their lifetimes, and the concurrency pattern. Our layered approach allows us to build from simple to sophisticated, understanding the trade-offs at each step.

For our project, we will implement a hybrid design that evolves through the milestones:

1. **Milestone 1:** A bump allocator with a rudimentary free list to enable `free`, establishing the core block metadata structure.
2. **Milestone 2:** A full explicit free list with multiple search strategies (first/best/worst-fit) and aggressive coalescing.
3. **Milestone 3:** Segregated free lists for common size classes, layered on top of the explicit list for larger allocations.
4. **Milestone 4:** Concurrency support and optimization using per-thread caches and separate `mmap` regions for large allocations.

This progressive approach ensures we build a solid mental model of block management before adding optimizations for speed and concurrency.

---

2. Goals and Non-Goals

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

This section establishes the precise boundaries of our memory allocator implementation. By clearly defining what the system **will** and **will not** do, we set realistic expectations and prevent scope creep. The functional goals map directly to the four project milestones, providing a roadmap from basic single-threaded allocation to a sophisticated, concurrent allocator. The non-goals explicitly exclude features that would transform this educational project into a production system or change its fundamental nature.

2.1 Functional Goals

Think of our memory allocator as a **custom-built workshop organizer** that we're constructing from scratch. We start with basic shelves (blocks) and gradually add advanced features like categorized bins (size classes) and multiple worker stations (thread safety). Each milestone represents a significant upgrade to the organizer's capabilities.

The functional goals are organized into four progressive tiers, each building upon the previous one. This incremental approach allows learners to master foundational concepts before tackling more complex challenges.

Tier 1: Foundational Allocation and Deallocation

These goals correspond to **Milestone 1** and establish the bare minimum functionality required for a usable allocator.

Goal Category	Specific Requirement	Why It Matters	Implementation Focus
Core Operations	Implement `malloc(size_t size)` that returns a pointer to at least `size` bytes of usable memory	Without allocation, there's no allocator. This is the primary interface users expect.	System call interface (`sbrk`), block metadata management
	Implement `free(void *ptr)` that releases previously allocated memory back to the pool	Memory reuse is essential for long-running programs to avoid exhausting available memory.	Block status tracking, free list maintenance
Memory Management	Request memory from the operating system via `sbrk` system call	The allocator must have a source of raw memory to parcel out.	Understanding program break, heap expansion
	Track allocation metadata in block headers prepended to user memory	The allocator needs to remember block sizes and status without relying on external data structures.	Data structure design, pointer arithmetic
Correctness	Return 8-byte aligned pointers on 64-bit systems	Many processors and data types require aligned memory access for performance and correctness.	Alignment padding calculations
	Handle `sbrk` failure by returning `NULL`	Real programs must handle out-of-memory conditions gracefully.	Error checking and propagation

> **Key Insight:** The first tier establishes the fundamental contract: `malloc` gives you memory, `free` gives it back. The metadata header is the allocator's "memory" of what it has handed out.

Tier 2: Efficient Free Memory Management

These goals correspond to **Milestone 2** and focus on optimizing how freed memory is tracked and reused, combating **fragmentation**—the primary enemy of heap allocators.

Goal Category	Specific Requirement	Why It Matters	Implementation Focus
Free Block Tracking	Maintain an explicit doubly-linked free list connecting all available blocks	Scanning the entire heap for free blocks is O(heap size). A dedicated list reduces this to O(free blocks).	Linked list operations within free memory
Block Optimization	Split large free blocks to satisfy smaller requests	Without splitting, a 1KB free block would waste 900 bytes when serving a 100-byte request.	Block division logic, metadata updates
	Coalesce adjacent free blocks into larger contiguous blocks	External fragmentation occurs when small free blocks scatter, preventing allocation of larger requests.	Boundary checking, neighbor block merging
Strategy Flexibility	Implement first-fit, best-fit, and worst-fit allocation strategies	Different strategies offer different trade-offs between speed, fragmentation, and implementation complexity.	Strategy pattern, configurable search algorithms
Robustness	Use boundary tags (headers and footers) for bidirectional coalescing	Finding the previous block without a footer requires scanning from the heap start—O(n) versus O(1).	Complete block metadata at both ends

Decision: Boundary Tags for O(1) Previous Block Access

- **Context**: When freeing a block, we need to check if the immediately preceding block in memory is also free to coalesce. Without a footer containing the block size, finding the previous block requires scanning from the heap beginning.
- **Options Considered**:
  1. **Single header only**: Store metadata only at block start. To find previous block, scan from heap beginning until reaching current block.
  2. **Boundary tags**: Store identical metadata (size, flags) at both block start and end.
  3. **Previous pointer in header**: Store direct pointer to previous block in header.
- **Decision**: Implement boundary tags (headers and footers).
- **Rationale**: Boundary tags provide O(1) access to the previous block using simple pointer arithmetic: `previous_block = current_block - footer_of_previous`. The 8-byte overhead per block (4 for header, 4 for footer on 32-bit) is acceptable for the significant performance gain in coalescing, especially important for long-running programs. Storing previous pointers would consume similar space but wouldn't help with finding the physical previous block for coalescing.

- **Consequences**: Every block has 8+ bytes of overhead (on 32-bit), but coalescing becomes efficient. The footer must be kept in sync with the header on every metadata change.

Option	Pros	Cons	Chosen?
Single header only	Minimal metadata overhead (4 bytes)	Finding previous block requires O(n) scan from heap start	No
Boundary tags	O(1) access to previous block, symmetrical design	8+ bytes overhead per block, must maintain two copies of size	**Yes**
Previous pointer in header	O(1) access to linked previous free block	Doesn't help find physically adjacent block, extra pointer overhead	No

Tier 3: Performance Optimization via Size Classes

These goals correspond to **Milestone 3** and address the performance limitation of scanning a single free list by introducing specialized fast paths for common allocation sizes.

Goal Category	Specific Requirement	Why It Matters	Implementation Focus
Fast Allocation	Maintain separate free lists for common size classes (e.g., 16, 32, 64, 128 bytes)	Most allocations are small. Scanning a single free list for a 16-byte request among 1KB blocks is inefficient.	Array of free list heads, size class mapping
	Serve allocations from appropriate size class in O(1) time (amortized)	Performance-critical applications require fast memory allocation.	Direct free list lookup, power-of-two rounding
Fragmentation Control	Reduce external fragmentation by grouping similar-sized blocks	Small and large blocks mixed together create fragmentation; segregating them keeps small blocks together.	Block classification, separate management
Adaptive Behavior	Split larger-class blocks to satisfy smaller requests when class list is empty	Don't fail a 32-byte request just because the 32-byte list is empty if a 64-byte block is available.	Fallback logic, block splitting across classes

Decision: Segregated Free Lists for Small Allocations

- **Context**: Scanning a single free list becomes inefficient as the heap grows and fragmentation increases. Most real-world allocations are small (< 256 bytes).
- **Options Considered**:
  1. **Single free list with caching**: Maintain one list but cache recently freed blocks.
  2. **Segregated fits**: Multiple free lists, each for a range of sizes (size class).
  3. **Buddy allocation**: Split blocks only in power-of-two sizes, enabling fast coalescing.
- **Decision**: Implement segregated free lists with power-of-two size classes.
- **Rationale**: For the common case of small allocations, segregated lists provide O(1) allocation time by checking only the appropriate size class list. The power-of-two classes simplify rounding and minimize internal fragmentation for typical allocation patterns. Buddy allocation, while elegant, has higher internal fragmentation (up to 50% waste) and more complex splitting/coalescing rules.
- **Consequences**: Increased metadata overhead (multiple list heads), potential internal fragmentation from rounding up to size classes, but dramatically faster small allocations.

Tier 4: Concurrency and Advanced Features

These goals correspond to **Milestone 4** and prepare the allocator for real-world usage in multi-threaded programs and with diverse allocation patterns.

Goal Category	Specific Requirement	Why It Matters	Implementation Focus
Thread Safety	Support concurrent allocation/free operations from multiple threads	Modern programs are multi-threaded; allocators must handle synchronization correctly.	Mutexes, atomic operations, lock-free paths
Large Allocation Handling	Use `mmap` for allocations above configurable threshold (e.g., 128KB)	Large blocks in the main heap cause fragmentation and are expensive to coalesce.	System call abstraction, threshold-based routing
Performance Optimization	Implement per-thread caches to reduce lock contention	Global locks become bottlenecks with many threads; thread-local caches provide fast paths.	Thread-local storage, cache management
Debugging Support	Detect common errors: double-free, use-after-free, buffer overflow	Debugging memory errors is notoriously difficult; built-in detection helps during development.	Magic numbers, guard bytes, sanity checks

Decision: Hybrid Locking with Thread-Local Caches

- **Context**: A single global lock protecting all allocations would serialize multi-threaded programs, destroying performance.
- **Options Considered**:
  1. **Global lock only**: Simple but poor concurrency.
  2. **Per-thread arenas**: Each thread gets its own heap region, no locking needed within arena.
  3. **Hybrid approach**: Thread-local caches for small allocations, global lock for large or cache-miss allocations.
- **Decision**: Implement hybrid locking with thread-local caches and a global fallback.
- **Rationale**: The hybrid approach provides a fast, lock-free path for the common case (small, frequent allocations) while maintaining correctness for larger allocations and cache exhaustion. Per-thread arenas can waste memory if threads have uneven allocation patterns. The hybrid model balances performance, memory efficiency, and implementation complexity.
- **Consequences**: More complex state management, potential for increased memory usage if thread-local caches grow unbounded, but excellent concurrent performance.

2.2 Non-Goals

Equally important to what we **will** build is what we **will not** build. These exclusions keep the project focused, manageable, and true to its educational purpose. Think of these as the "not in this workshop" signs that prevent us from accidentally building a different kind of tool entirely.

Memory Management Beyond Allocation/Deallocation

Non-Goal	Why Excluded	What We Do Instead
Garbage collection (automatic memory reclamation)	Changes the fundamental programming model from explicit `free()` to automatic lifetime management. GC is a separate, complex domain requiring tracing, reference counting, or mark-and-sweep algorithms.	Require explicit `free()` calls, matching standard C library semantics. Focus on efficient reuse of explicitly freed memory.
Compaction (moving allocated blocks to reduce fragmentation)	Requires updating all existing pointers to moved objects, which is impossible in C without language/runtime support. Would break the fundamental "pointer stability" guarantee of `malloc`.	Use coalescing to merge adjacent free blocks, reducing external fragmentation while keeping allocated blocks stationary.
Virtual memory paging management	The operating system already handles paging transparently. Our allocator works with virtual addresses; physical memory management is the OS's responsibility.	Treat memory as a flat address space. Rely on OS `mmap` and `sbrk` for obtaining virtual memory pages.

Production-Grade Features

Non-Goal	Why Excluded	What We Do Instead
jemalloc/tcmalloc-level performance optimizations	Production allocators have man-years of optimization for specific workloads (web servers, databases, etc.). Our educational allocator focuses on understandable algorithms.	Implement clear, documented strategies (first-fit, best-fit) that demonstrate concepts rather than maximize micro-optimizations.
Extensive statistics and profiling hooks	While useful for tuning, comprehensive metrics collection adds significant complexity that distracts from core allocation algorithms.	Provide basic debugging support (boundary checks, magic numbers) sufficient for learning and debugging the allocator itself.
Cross-platform abstraction beyond Unix-like systems	Windows uses completely different memory APIs (`VirtualAlloc` vs `mmap`/`sbrk`). Supporting both would double the OS interface complexity.	Target Unix-like systems (Linux, macOS, BSD) where `sbrk` and `mmap` are available. Mention Windows differences in documentation.
Security hardening against malicious exploitation	Production allocators include defenses against heap spraying, use-after-free exploits, etc. These are important but separate from allocation algorithm fundamentals.	Focus on correctness for well-behaved programs. Include basic sanity checks that catch common programming errors.

Interface Extensions

Non-Goal	Why Excluded	What We Do Instead
Full C memory API (`calloc`, `realloc`, `memalign`, `posix_memalign`)	These are convenience functions that can be implemented using `malloc` and `free`. Adding them expands interface surface without teaching new allocation concepts.	Implement only `malloc` and `free`. Mention that `calloc` is `malloc` + zeroing, `realloc` is `malloc` + copy + `free`.
Custom alignment beyond system minimum	While `memalign` is useful, it introduces complex padding and fragmentation issues that distract from core allocation algorithms.	Guarantee 8-byte alignment (or `alignof(max_align_t)`), which satisfies most requirements.
Allocation hooks/interceptors	Hooks for debugging or replacement are useful but add significant complexity to the core allocation path.	Keep allocation path simple. Debugging features are built-in rather than hook-based.

> **Key Principle:** This is a **teaching allocator**, not a **production allocator**. We prioritize clarity, educational value, and incremental learning over performance, feature completeness, or robustness against adversarial inputs. Each line of code should serve the purpose of illustrating how memory allocation works.

Scope Boundary Examples

To make these non-goals concrete, consider these specific examples of what falls outside our scope:

1. **A program that never calls `free()`** — Our allocator will eventually exhaust memory (return `NULL`). We don't implement garbage collection to automatically identify unused memory.
2. **Allocating a 1MB block, freeing it, then allocating 900KB** — If fragmentation prevents satisfying the 900KB request even though 1MB is free (due to splitting), that's external fragmentation we mitigate through coalescing but cannot eliminate without compaction.
3. **Running on Windows without WSL/Cygwin** — Our allocator relies on Unix system calls (`sbrk`, `mmap`) and won't compile or run on native Windows.
4. **Microbenchmarks showing 2% better performance than glibc `malloc`** — We're not optimizing at that level; our algorithms are chosen for clarity, not peak performance.

2.3 Success Criteria

Beyond the milestone acceptance criteria, the overall project succeeds if:

1. **Educational Value Achieved**: Learners understand how `malloc`/`free` work internally, including block headers, free lists, fragmentation, and system call interactions.

2. **Incremental Progress Possible**: Each milestone builds logically on the previous one, allowing learners to succeed even if they don't complete all milestones.
3. **Debugging Skills Developed**: The implementation process teaches systematic debugging of memory corruption, alignment issues, and concurrency problems.
4. **Conceptual Transfer**: The mental models (warehouse, shelf-finding strategies, boundary tags) help learners understand other allocators and memory management concepts.

Implementation Guidance

While this goals section is primarily about requirements definition, here are initial implementation considerations that bridge to the subsequent architecture sections:

Technology Recommendations Table:

Component	Simple Option	Advanced Option
System Calls	`sbrk()` for all allocations	`mmap()` for large allocations, `sbrk()` for small
Synchronization	Single global pthread mutex	Hybrid: thread-local caches + global mutex
Debugging	"Magic number" in headers	Guard bytes, allocation backtraces
Free List	Implicit list (scan all blocks)	Explicit doubly-linked list

Recommended Initial Project Structure:

```
build-your-own-allocator/
├── include/
│   └── allocator.h          # Public API (malloc/free prototypes)
├── src/
│   ├── allocator.c          # Main entry points (malloc/free wrappers)
│   ├── os_mem.c             # sbrk/mmap wrappers (Milestone 1 & 4)
│   ├── block.c              # Block header/footer operations (Milestone 1)
│   ├── free_list.c          # Free list management (Milestone 2)
│   ├── strategies.c         # First-fit, best-fit, worst-fit (Milestone 2)
│   ├── segregated.c         # Size class management (Milestone 3)
│   └── thread_cache.c       # Per-thread caching (Milestone 4)
├── tests/
│   ├── test_basic.c         # Milestone 1 tests
│   ├── test_fragmentation.c  # Milestone 2 tests
│   └── test_concurrent.c     # Milestone 4 tests
└── Makefile
```

Infrastructure Starter Code: For Milestone 1, these OS interface wrappers provide a solid foundation:

```c
/* os_mem.c - Operating system memory interface */

#include <unistd.h>

#include <sys/mman.h>

#include <errno.h>


#define PAGE_SIZE 4096


/* Simple wrapper for sbrk with error checking */

static void *os_sbrk(intptr_t increment) {

    void *old_break = sbrk(0);

    if (old_break == (void*)-1) {

        return NULL;  /* sbrk failed to get current break */

    }



    void *new_break = sbrk(increment);

    if (new_break == (void*)-1) {

        return NULL;  /* Failed to extend heap */

    }



    return old_break;  /* Return start of newly allocated region */

}


/* Align size to system page size */

static size_t align_to_page(size_t size) {

    return (size + PAGE_SIZE - 1) & ~(PAGE_SIZE - 1);

}
```

Core Logic Skeleton - Milestone 1 Foundation:

```c
/* block.c - Block header structure and operations */

#include <stddef.h>

#include <stdint.h>


typedef struct block_header {

    size_t size;           /* Size of user data (excluding header) */

    int allocated;         /* 1 if allocated, 0 if free */

    struct block_header *next;  /* For free list (Milestone 2) */

    struct block_header *prev;  /* For free list (Milestone 2) */

    /* Note: Footer will be at block_start + size - sizeof(footer) */

} block_header_t;


/* Helper to align sizes to 8 bytes (64-bit alignment) */

#define ALIGNMENT 8

#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))


/* Convert user pointer to its block header */

static block_header_t *get_header_from_ptr(void *ptr) {

    // TODO 1: Calculate header location from user pointer

    // Hint: Header is before user memory

}


/* Calculate total block size including header and alignment */

static size_t get_total_block_size(size_t user_size) {

    // TODO 2: Return user_size + header_size + alignment_padding

    // Must ensure the total size is multiple of ALIGNMENT

}


/* Initialize a new block at given memory location */

static void init_block(void *memory, size_t user_size, int allocated) {

    // TODO 3: Set up header with given size and allocated flag

    // TODO 4: If using boundary tags, set up footer too

}
```

Language-Specific Hints for C:

1. **Pointer Arithmetic**: Use `(char*)` casting for byte-level pointer arithmetic: `(char*)ptr + offset`.
2. **Type Punning**: Use `union` or `memcpy` for writing headers to avoid strict aliasing violations.
3. **System Includes**: Include `<stdint.h>` for precise size types, `<unistd.h>` for `sbrk`.
4. **Debug Macros**: Use `#ifdef DEBUG` conditional compilation for debugging output.
5. **Attribute Packed**: Use `__attribute__((packed))` for headers to prevent compiler padding that would break pointer arithmetic.

Milestone 1 Checkpoint Test: After implementing Milestone 1, this test should run successfully:

```c
/* test_basic.c - Milestone 1 validation */

#include "allocator.h"

#include <stdio.h>

#include <assert.h>


int main() {

    /* Test basic allocation */

    int *p1 = (int*)malloc(100 * sizeof(int));

    assert(p1 != NULL);

    printf("√ Allocated 100 ints\n");



    /* Test alignment */

    uintptr_t addr = (uintptr_t)p1;

    assert((addr % 8) == 0);

    printf("√ Pointer is 8-byte aligned\n");



    /* Test free and reuse */

    free(p1);

    int *p2 = (int*)malloc(50 * sizeof(int));

    assert(p2 != NULL);

    printf("√ Freed and reallocated memory\n");



    /* Note: p2 might not equal p1 due to no free list yet */



    printf("Milestone 1 tests passed!\n");

    return 0;

}
```

Expected output:

```
√ Allocated 100 ints
√ Pointer is 8-byte aligned
√ Freed and reallocated memory
Milestone 1 tests passed!
```

Debugging Tips for Early Implementation:

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault on first `malloc`	`sbrk` returning `(void*)-1`	Print `sbrk` return value, check `errno`	Ensure `sbrk` parameter is positive, handle errors
`malloc` returns misaligned pointer	Header size not considered in alignment	Print pointer value, check `(addr % 8) == 0`	Align total block size, not just user size
`free` crashes or corrupts heap	Wrong header location calculation	Print header address before/after free	Use `get_header_from_ptr` helper consistently
Memory leak (no reuse after free)	Not tracking freed blocks	Check if second allocation calls `sbrk` again	Implement free list (Milestone 2)

---

3. High-Level Architecture

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

This section provides a bird's-eye view of the memory allocator system, explaining how its major components fit together and interact. Before diving into implementation details, we need to understand the overall structure and responsibilities of each part. Think of building this allocator as constructing a sophisticated warehouse management system: we need interfaces to acquire raw land from the property owner (the OS), a floor plan to organize storage units (the heap), different strategies for finding available space (allocation algorithms), and separate workstations for multiple warehouse workers to avoid traffic jams (thread safety).

Component Overview & Responsibilities

At the highest level, our custom allocator sits between the user's program and the operating system, intercepting calls to `malloc` and `free` and managing a region of virtual memory called the **heap**. The system decomposes into four primary components, each with distinct responsibilities:

1. **OS Memory Interface** - The "Land Acquisition Department"
2. **Heap Manager** - The "Warehouse Floor Manager"
3. **Allocation Strategy Engine** - The "Space-Finding Algorithm Library"
4. **Thread-Local Cache Manager** - The "Personal Workstation Coordinator"

The diagram above shows how these components interact. When a user program calls `malloc()`, the request flows through the allocator API to the thread-local cache (if enabled), then to the appropriate allocation strategy, which consults the heap manager. If insufficient free space exists, the heap manager requests more memory from the OS interface. The returned pointer travels back up the chain to the user. Each component's responsibilities are detailed below.

OS Memory Interface Component

Component	Primary Responsibility	Key Operations	Analogous Role
OS Memory Interface	Acquire and release raw memory pages from the operating system	Request memory via `sbrk` or `mmap`, release via `munmap`	Warehouse landlord who provides empty land

This component abstracts away operating system specifics for obtaining raw memory. It provides two fundamental services: expanding the contiguous **heap** region via the `sbrk` system call for small allocations, and mapping discrete memory regions via `mmap` for large allocations. The interface handles alignment to page boundaries, error checking when system calls fail, and choosing the appropriate method based on allocation size. It knows nothing about how the acquired memory will be divided or managed—it simply provides empty "land" to the heap manager.

Heap Manager Component

Component	Primary Responsibility	Key Operations	Analogous Role
Heap Manager	Organize acquired memory into blocks, track allocation status, maintain free list	Initialize blocks, split blocks, coalesce adjacent free blocks, update metadata	Warehouse floor manager who divides land into labeled storage units

The heap manager views memory as a contiguous sequence of **blocks**, each with a **header** (and optionally a **footer**) containing metadata. Its core responsibility is maintaining the integrity of this block sequence and the **free list** data structure that tracks available blocks. When the OS interface provides a new memory region, the heap manager carves it into an initial free block. When an allocation request arrives, it finds a suitable free block (via the strategy engine), marks it as allocated, and returns a pointer to the user-usable portion. When `free()` is called, it marks the block as free, inserts it into the free list, and attempts to **coalesce** it with adjacent free blocks to combat **external fragmentation**.

Allocation Strategy Engine Component

Component	Primary Responsibility	Key Operations	Analogous Role
Allocation Strategy Engine	Implement different algorithms for searching the free list and selecting blocks	First-fit, best-fit, worst-fit searches; size class mapping for segregated lists	Algorithm library offering different "shelf-finding" strategies

This component encapsulates the logic for how to search for and select a free block when satisfying an allocation request. Different strategies offer different trade-offs between allocation speed, fragmentation, and implementation complexity. The engine supports three classical search strategies (first-fit, best-fit, worst-fit) that scan a single free list, plus a more advanced **segregated free lists** approach that maintains multiple free lists binned by **size class**. The strategy is selectable at compile time or runtime, allowing experimentation with different approaches. The engine works closely with the heap manager, which performs the actual block splitting and metadata updates once a block is selected.

Thread-Local Cache Manager Component

Component	Primary Responsibility	Key Operations	Analogous Role
Thread-Local Cache Manager	Provide per-thread memory pools to reduce lock contention in multithreaded programs	Maintain thread-local free lists, acquire/release global locks only when necessary	Personal workstation coordinator giving each worker their own toolbox

In a multithreaded program, multiple threads may call `malloc` and `free` concurrently. Without protection, simultaneous modifications to the shared free list would cause race conditions and memory corruption. This component introduces thread safety through a two-tier approach: each thread gets its own small cache (a **thread-local storage** free list) for fast allocations without locking. Only when the thread's cache is empty or full does it acquire a global mutex to access the shared heap. This dramatically reduces **lock contention**—the performance penalty when multiple threads wait for the same lock—while maintaining correctness. The component also handles the lifecycle of thread-local data when threads are created and destroyed.

Component Interactions and Data Flow

The components interact in a layered fashion, with clear boundaries and well-defined interfaces:

1. **User Program → Allocator API**: The program calls `malloc(size)` or `free(ptr)`. These functions are our implementation's entry points.
2. **Allocator API → Thread-Local Cache (if enabled)**: For `malloc`, check if a suitable block exists in the current thread's cache. For `free`, potentially add the block to the thread's cache instead of immediately returning it to the global pool.
3. **Thread-Local Cache → Allocation Strategy Engine**: If the thread cache cannot satisfy the request, the engine's selected strategy is invoked to find a block from the global free structures.
4. **Allocation Strategy Engine → Heap Manager**: The strategy searches the free list(s) maintained by the heap manager. It calls heap manager functions to split blocks if a suitable free block is found.
5. **Heap Manager → OS Memory Interface**: If no suitable free block exists, the heap manager requests additional memory from the OS interface via `os_sbrk()` or `os_mmap()`.
6. **Return Path**: Once a block is allocated, a pointer to its user payload travels back through the chain to the caller. For `free`, the block is marked free and potentially coalesced with neighbors before being added to appropriate free lists.

> **Design Principle: Separation of Concerns** Each component has a single, well-defined responsibility. The OS interface doesn't know about block headers. The heap manager doesn't implement search algorithms. The strategy engine doesn't handle thread safety. This separation makes the system easier to understand, test, and modify. For example, we can swap from first-fit to best-fit by changing only the strategy component.

System State and Global Data Structures

The allocator maintains several global data structures that represent its current state:

Data Structure	Owner Component	Purpose	Persistence
Heap base pointer	Heap Manager	Starting address of the contiguous heap region acquired via `sbrk`	Process lifetime
Free list head	Heap Manager (or Strategy Engine for segregated lists)	Entry point to the linked list of free blocks	Changes with allocations/frees
Size class array	Allocation Strategy Engine (segregated lists)	Array of free list heads, one per size class	Process lifetime
Global mutex	Thread-Local Cache Manager	Lock protecting access to global heap structures when thread caches are exhausted	Process lifetime
Thread-local storage key	Thread-Local Cache Manager	Mechanism to access per-thread cache data	Thread lifetime

The **heap** itself—the raw memory region—is the allocator's primary persistent state. All metadata (`block_header_t` structures, free list pointers) is embedded within this region, minimizing the need for separate tracking structures. This design choice means the allocator's entire state (except for a few global pointers) resides within the managed memory itself, which simplifies persistence and recovery (though our teaching allocator doesn't implement saving/restoring state).

Recommended File/Module Structure

Organizing code into logical modules from the start prevents a monolithic, hard-to-navigate codebase. Below is the recommended directory and file structure, which mirrors our component architecture:

```
build-your-own-allocator/
├── include/                    # Public header files
│   └── allocator.h             # Main allocator API (malloc/free declarations)
├── src/                        # Source code implementation
│   ├── allocator.c             # Top-level malloc/free implementations
│   ├── os_mem.c                # OS Memory Interface component
│   ├── heap.c                  # Heap Manager component
│   ├── free_list.c             # Free list manipulation utilities
│   ├── strategies.c            # Allocation Strategy Engine component
│   ├── thread_cache.c          # Thread-Local Cache Manager component
│   └── debug.c                 # Debugging and visualization utilities
├── tests/                      # Test programs
│   ├── test_basic.c            # Milestone 1: Basic allocator tests
│   ├── test_free_list.c        # Milestone 2: Free list management tests
│   ├── test_segregated.c       # Milestone 3: Segregated lists tests
│   ├── test_threads.c          # Milestone 4: Thread safety tests
│   └── test_stress.c           # Stress and performance tests
└── Makefile                    # Build configuration
```

Each file's responsibilities and key contents:

File	Primary Component	Key Contents	Dependencies
`allocator.h`	Allocator API	Function prototypes for `malloc`, `free`, `calloc`, `realloc`; configuration macros (`FIRST_FIT`, `SEGREGATED_LISTS`); `block_header_t` and `footer_t` definitions	None (system headers only)
`allocator.c`	Allocator API	Implementation of `malloc`, `free`, `calloc`, `realloc`; initialization code; top-level request routing to appropriate components	All other `.c` files
`os_mem.c`	OS Memory Interface	`os_sbrk()`, `os_mmap()` wrappers; `align_to_page()` helper; threshold-based decision logic	System headers (`unistd.h`, `sys/mman.h`)
`heap.c`	Heap Manager	`init_block()`, `split_block()`, `coalesce_block()` functions; heap initialization; block boundary checking	`free_list.c` for list operations
`free_list.c`	Heap Manager	Free list insertion/removal; `find_free_block()` (first/best/worst-fit); list traversal helpers	None (pure data structure ops)
`strategies.c`	Allocation Strategy Engine	Strategy function pointers; size class definitions; `malloc_segregated()` implementation; strategy selection logic	`heap.c`, `free_list.c`
`thread_cache.c`	Thread-Local Cache Manager	Thread-local storage initialization; `malloc_thread_cached()`; per-thread arena management; lock wrappers	System headers (`pthread.h`)
`debug.c`	Diagnostics	Heap visualization function; magic number checking; corruption detection; statistics collection	All allocator headers

This modular structure provides several benefits:

- **Incremental Implementation**: You can implement and test Milestone 1 with just `allocator.c`, `os_mem.c`, and `heap.c`, then add other files for later milestones.
- **Clear Boundaries**: Each component's logic is contained within its file, making it easier to understand and modify.
- **Testing Flexibility**: Tests can link against specific components to verify individual behaviors.
- **Configuration Management**: Compile-time strategy selection (`#ifdef FIRST_FIT`) can be centralized in `strategies.c`.

> **Architecture Decision: Modular vs Monolithic Organization**
>
> **Context**: We need to organize ~1000-2000 lines of allocator code for clarity, maintainability, and incremental implementation by learners.
>
> **Options Considered**:
>
> 1. **Single-file monolithic**: All code in one `.c` file
> 2. **Feature-based modular**: Separate files by component (as shown above)
> 3. **Strategy-based modular**: Separate files by allocation strategy (first_fit.c, best_fit.c, segregated.c)
>
> **Decision**: Feature-based modular organization (Option 2)
>
> **Rationale**:
>
> - Single-file becomes unwieldy beyond 500 lines and makes incremental implementation harder
> - Strategy-based organization would duplicate common infrastructure (block management, OS interface) across files
> - Feature-based separation aligns with our component architecture and allows learners to focus on one concept at a time
> - Clear file responsibilities make the codebase more navigable for learners
>
> **Consequences**:
>
> - Some cross-component dependencies exist (e.g., `strategies.c` needs `heap.c` functions)
> - Need to manage header file inclusions carefully to avoid circular dependencies
> - Build system (Makefile) must compile and link multiple source files

Component Interfaces and Dependencies

The table below specifies the key functions each component exports (its public interface) and what other components it depends on:

Component	Key Exported Functions	Dependencies	Called By
OS Memory Interface	`os_sbrk()`, `os_mmap()`, `align_to_page()`	Operating system (system calls)	Heap Manager
Heap Manager	`init_block()`, `split_block()`, `coalesce_block()`, `get_header_from_ptr()`	OS Memory Interface (for expansion)	Allocation Strategy Engine, Thread Cache
Free List Manager	`free_list_insert()`, `free_list_remove()`, `find_free_block()`	Heap Manager (block metadata)	Heap Manager, Strategy Engine
Allocation Strategy Engine	`strategy_malloc()` (function pointer), `size_class_index()`, `malloc_segregated()`	Free List Manager, Heap Manager	Allocator API, Thread Cache
Thread-Local Cache Manager	`thread_cache_get()`, `thread_cache_put()`, `init_thread_cache()`	Allocation Strategy Engine (for cache miss), pthreads library	Allocator API

The dependency flow is largely unidirectional: higher-level components depend on lower-level ones, but not vice versa. For example, the Thread-Local Cache Manager calls the Allocation Strategy Engine when its cache is empty, but the Strategy Engine knows nothing about thread caches. This hierarchical structure minimizes coupling and makes the system more maintainable.

Common Pitfalls in Architectural Design

⚠️ Pitfall: Overly Tight Coupling Between Components

- **Description**: Implementing allocation strategy logic directly in the heap manager, or embedding thread safety code throughout all components.
- **Why It's Wrong**: Makes the system hard to modify (changing from first-fit to best-fit requires rewriting core heap logic) and test (can't test heap operations without also testing search algorithms).
- **Fix**: Define clean interfaces between components. Use function pointers or strategy pattern for algorithms. Keep thread safety concerns isolated to the thread cache component.

⚠️ Pitfall: Global State Spaghetti

- **Description**: Storing heap base pointer in one global variable, free list head in another, size class array in a third, with no overarching structure.
- **Why It's Wrong**: Difficult to track what state exists, hard to reset or reinitialize the allocator, prone to initialization order bugs.
- **Fix**: Create a top-level `allocator_state_t` structure that holds all global state (except truly thread-local data). Initialize it once at startup.

⚠️ Pitfall: Inconsistent Error Handling Across Components

- **Description**: OS interface returns `NULL` on failure, heap manager returns `-1`, strategy engine returns a special error block pointer.
- **Why It's Wrong**: Callers must understand each component's error conventions, leading to bugs when errors aren't propagated correctly.
- **Fix**: Establish a consistent error convention (e.g., all functions returning pointers use `NULL` for failure, all functions returning status use `0` for success, `-1` for failure). Document this convention clearly.

⚠️ Pitfall: Circular Header Dependencies

- **Description**: `heap.h` includes `strategies.h`, which includes `free_list.h`, which includes `heap.h` again.
- **Why It's Wrong**: Causes compilation failures, confusing include order requirements, and potential for infinite inclusion loops.
- **Fix**: Design header files to be includable in any order. Use forward declarations for types (`struct block_header_t;`) instead of including headers that define them. Create a central `allocator_internal.h` for internal type definitions that all components need.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option (Beginner-Friendly)	Advanced Option (Performance-Optimized)
OS Interface	Pure `sbrk()` for all allocations	`sbrk()` for small, `mmap()` for large allocations with configurable threshold
Block Metadata	Single header with size + allocated flag	Boundary tags (header + footer) for bidirectional coalescing
Free List	Implicit free list (scan all blocks)	Explicit doubly-linked free list embedded in free blocks
Allocation Strategy	First-fit only	Compile-time selectable: first-fit, best-fit, worst-fit, segregated lists
Thread Safety	Global mutex protecting all operations	Per-thread caches with lock-free fast path, global mutex fallback
Debugging	Basic boundary checks + `printf` debugging	Magic numbers in headers, heap visualization function, statistics collection

Recommended File/Module Structure Implementation

Create the following directory and file structure to implement the modular design:

```
my_allocator/
├── include/
│   └── allocator.h
├── src/
│   ├── allocator.c
│   ├── os_mem.c
│   ├── heap.c
│   ├── free_list.c
│   ├── strategies.c
│   ├── thread_cache.c
│   └── debug.c
├── tests/
│   ├── test_basic.c
│   ├── test_free_list.c
│   ├── test_segregated.c
│   ├── test_threads.c
│   └── test_stress.c
└── Makefile
```

Here's a starter `Makefile` to build the project:

```makefile
# Makefile for Build Your Own Allocator
CC = gcc
CFLAGS = -Wall -Wextra -g -I./include -DDEBUG=1
LDFLAGS = -lpthread

# Source files
SRCS = src/allocator.c src/os_mem.c src/heap.c src/free_list.c \
       src/strategies.c src/thread_cache.c src/debug.c

# Object files
OBJS = $(SRCS:.c=.o)

# Test executables
TESTS = tests/test_basic tests/test_free_list tests/test_segregated \
        tests/test_threads tests/test_stress

# Default target: build the library
all: liballocator.a

# Create static library
liballocator.a: $(OBJS)
        ar rcs $@ $(OBJS)

# Compile source files
%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

# Build test programs
tests/%: tests/%.c liballocator.a
        $(CC) $(CFLAGS) $< -L. -lallocator $(LDFLAGS) -o $@

# Build all tests
tests: $(TESTS)

# Clean build artifacts
clean:
        rm -f $(OBJS) liballocator.a $(TESTS)

# Run basic tests
test: tests/test_basic
        ./tests/test_basic

.PHONY: all clean test tests
```

Infrastructure Starter Code: Core Header File

Create `include/allocator.h` with foundational type definitions and function prototypes:

```c
#ifndef ALLOCATOR_H

#define ALLOCATOR_H


#include <stddef.h>

#include <stdint.h>


/* ===== Configuration Constants ===== */

#define ALIGNMENT 8                    /* Minimum alignment for all returned pointers */

#define PAGE_SIZE 4096                 /* System page size for mmap */

#define SBRK_THRESHOLD (128 * 1024)    /* Use mmap for allocations larger than this */


/* ===== Core Data Structures ===== */


/* Block header - precedes every memory block (allocated or free) */

typedef struct block_header {

    size_t size;                   /* Size of user payload (excluding metadata) */

    int allocated;                 /* 1 if block is allocated, 0 if free */

    struct block_header *next;     /* Next block in free list (valid only if free) */

    struct block_header *prev;     /* Previous block in free list (valid only if free) */

} block_header_t;


/* Block footer - mirrors header for backward coalescing */

typedef struct footer {

    size_t size;                   /* Size of user payload (must match header) */

    int allocated;                 /* Allocation status (must match header) */

} footer_t;


/* ===== Public API Functions ===== */

void *malloc(size_t size);

void free(void *ptr);

void *calloc(size_t nmemb, size_t size);

void *realloc(void *ptr, size_t size);


/* ===== Internal Functions (for testing/debugging) ===== */

void heap_visualize(void);         /* Print heap layout for debugging */

size_t heap_get_allocated_bytes(void); /* Get total allocated user bytes */


#endif /* ALLOCATOR_H */
```

Core Logic Skeleton Code

In `src/allocator.c`, implement the top-level `malloc` and `free` functions that route requests through the component chain:

```c
#include "allocator.h"

#include <string.h>


/* Global allocator state */

static struct {

    void *heap_base;               /* Start of heap from sbrk */

    size_t heap_size;              /* Current heap size in bytes */

    /* TODO: Add other global state (free list head, etc.) */

} allocator_state;


/* Initialize allocator on first malloc */

static void allocator_init(void) {

    // TODO 1: Initialize heap_base to NULL (indicating not initialized)

    // TODO 2: Set heap_size to 0

    // TODO 3: Other initialization as needed

}


/* Top-level malloc implementation */

void *malloc(size_t size) {

    // TODO 1: Call allocator_init() if first time

    // TODO 2: Handle size == 0 (return NULL or unique pointer per implementation choice)

    // TODO 3: Calculate total block size including metadata and alignment

    // TODO 4: If THREAD_SAFE defined, call thread_cache_malloc()

    // TODO 5: Otherwise, call strategy_malloc() with calculated size

    // TODO 6: If successful, return pointer to user payload (after header)

    // TODO 7: If failed (out of memory), return NULL

    return NULL;  // Placeholder

}


/* Top-level free implementation */

void free(void *ptr) {

    // TODO 1: If ptr is NULL, do nothing (standard free behavior)

    // TODO 2: If allocator not initialized, abort (should not happen)

    // TODO 3: Derive block header from user pointer using pointer arithmetic

    // TODO 4: Validate block (check magic number if debugging enabled)

    // TODO 5: If THREAD_SAFE defined, call thread_cache_free()

    // TODO 6: Otherwise, mark block as free and coalesce with neighbors

    // TODO 7: Insert freed block into appropriate free list


```

```
}

/* calloc: allocate and zero memory */
void *calloc(size_t nmemb, size_t size) {
    // TODO 1: Calculate total size (nmemb * size), check for overflow
    // TODO 2: Call malloc with calculated size
    // TODO 3: If successful, zero the memory using memset
    // TODO 4: Return pointer or NULL on failure
    return NULL;  // Placeholder
}

/* realloc: resize existing allocation */
void *realloc(void *ptr, size_t size) {
    // TODO 1: If ptr is NULL, equivalent to malloc(size)
    // TODO 2: If size is 0, equivalent to free(ptr) and return NULL
    // TODO 3: Get header of existing block
    // TODO 4: If existing block is large enough, possibly split and return same ptr
    // TODO 5: Otherwise, malloc new block of requested size
    // TODO 6: Copy data from old block to new (up to minimum of old and new sizes)
    // TODO 7: Free old block
    // TODO 8: Return pointer to new block
    return NULL;  // Placeholder
}
```

Language-Specific Hints for C Implementation

- **System Calls**: Use `sbrk(0)` to get current program break without changing it. Cast return value to `void*` and check for `(void*)-1` error.
- **Memory Alignment**: Use `(size + ALIGNMENT - 1) & ~(ALIGNMENT - 1)` to round up to nearest multiple of `ALIGNMENT`.
- **Pointer Arithmetic**: When calculating addresses for headers/footers, cast to `uintptr_t` for arithmetic, then back to pointer type.
- **Thread-Local Storage**: Use `pthread_key_create()` and `pthread_getspecific()` / `pthread_setspecific()` for portable thread-local data.
- **Mutual Exclusion**: Use `pthread_mutex_t` with `pthread_mutex_lock()` / `pthread_mutex_unlock()` for thread safety.
- **Debugging Aids**: Add `#define MAGIC 0xDEADBEEF` to header and check it in `free()` to detect corruption.

Milestone Checkpoint for High-Level Architecture

After setting up the file structure and skeleton code:

1. **Command to verify setup**: Run `make` in the project root directory
2. **Expected output**: Successful compilation with no errors, creating `liballocator.a` and object files
3. **Verify behavior**: Create a simple test program:

```c
#include "include/allocator.h"

#include <stdio.h>



int main() {

    printf("Allocator skeleton compiled successfully.\n");

    return 0;

}
```

Compile with `gcc -I./include test_skeleton.c -L. -lallocator -o test_skeleton && ./test_skeleton` 4. **Signs of problems**:

- If compilation fails, check include paths in Makefile
- If linking fails, ensure all source files compile to object files
- If test program doesn't run, check library path

This high-level architectural foundation provides the structure for implementing the detailed components in subsequent sections. The modular design supports incremental development and testing of each milestone's features.

4. Data Model

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

This section defines the fundamental building blocks of the memory allocator — the data structures that track every byte of memory in the heap. Just as a warehouse manager needs detailed inventory records to know which shelves are full, empty, or available for use, the allocator needs precise metadata to manage memory blocks efficiently. The data model serves as the blueprint for the entire system, dictating how memory is organized, how free space is tracked, and how operations like allocation and deallocation are performed. A well-designed data model balances metadata overhead against operational efficiency, enabling fast allocation decisions while minimizing wasted space.

4.1 Block Header: The 'Shipping Label'

Mental Model: Imagine every shelf unit in our warehouse has a **shipping label** taped to its front. This label contains essential information: the shelf's total capacity, whether it's currently in use, and, if it's empty, where to find the next empty shelf. The label is always placed at the very beginning of the shelf, so anyone looking at a shelf knows immediately how to interpret it. In memory terms, this label is the **block header** — a small piece of metadata prepended to every block of memory, whether allocated or free. It's the single source of truth about that block's identity and status.

Every memory block managed by our allocator begins with a `block_header_t` structure. This header is the cornerstone of our entire memory management system, providing the information needed to:

1. **Identify block boundaries**: Know exactly where a block starts and ends
2. **Determine allocation status**: Differentiate between in-use and available blocks
3. **Navigate the free list**: Link free blocks together for efficient searching
4. **Enable coalescing**: Detect adjacent free blocks for merging

The header must be stored at a known, fixed offset from the start of each block, which means the pointer returned to the user (`malloc`'s return value) points just *after* the header. When the user calls `free`, we subtract this fixed offset to find the header again.

Architecture Decision Record: Unified Header for All Blocks

Decision: Single Header Structure for All Block States

- **Context**: We need metadata for both allocated and free blocks, but the information required differs slightly. Allocated blocks only need size and status, while free blocks also need forward/backward pointers for free list linkage.
- **Options Considered**:
  1. **Separate structures**: Different `allocated_header_t` (size + flags) and `free_header_t` (size + flags + next + prev).
  2. **Unified maximal structure**: Single `block_header_t` containing all possible fields (size, flags, next, prev) used by both block types.
  3. **Embedded free node**: Store free list pointers *within* the payload area of free blocks, keeping header minimal.
- **Decision**: Use Option 2 — a unified `block_header_t` containing all fields, with free list pointers only valid when block is free.
- **Rationale**: Simplicity and memory safety. A single structure type reduces code complexity and pointer arithmetic errors. The space overhead of unused pointer fields in allocated blocks is acceptable (16 bytes on 64-bit systems) and simplifies boundary checking and debugging. Option 1 would require type casting and careful alignment. Option 3 is more space-efficient but increases implementation complexity for a teaching allocator.
- **Consequences**: Allocated blocks waste 16 bytes (for next/prev pointers), but this fixed overhead is predictable. Debugging is easier since all blocks have the same layout. Free list management is straightforward — we can store pointers directly in the header.

Option	Pros	Cons	Chosen?
Separate structures	Minimal overhead for allocated blocks	Complex type casting; harder to debug	✗
Unified maximal structure	Simple implementation; consistent layout	Wasted space in allocated blocks	✓
Embedded free node	No wasted space in allocated blocks	Complex pointer arithmetic; harder to coalesce	✗

The complete `block_header_t` structure contains the following fields:

Field Name	Type	Description
`size`	`size_t`	**Total usable size of the block** (excluding header and footer). This is the amount of memory available to the user when allocated. For alignment, this size is always rounded up to a multiple of `ALIGNMENT` (8 bytes).
`allocated`	`int`	**Allocation status flag**. 1 means the block is currently allocated (in use); 0 means it's free. This boolean determines whether the block is part of the free list and whether it can be coalesced.
`next`	`struct block_header_t*`	**Forward pointer for free list**. Only valid when `allocated == 0`. Points to the next free block in the explicit doubly-linked free list. For the last free block, this points to `NULL`.
`prev`	`struct block_header_t*`	**Backward pointer for free list**. Only valid when `allocated == 0`. Points to the previous free block in the explicit doubly-linked free list. For the first free block, this points to `NULL`.
(implicit padding)	-	**Alignment padding**. The compiler may add padding after these fields to ensure the entire structure is aligned according to platform requirements (typically 8-byte aligned on 64-bit systems). We must account for this when calculating offsets.

Block Layout with Boundary Tags:

The diagram above shows the complete layout of a memory block. Following the header is the **user payload** — the actual memory returned to the caller of `malloc`. This region starts at the first address after the header that meets the system's alignment requirements (at least 8-byte aligned). At the very end of the block, we place a **footer** (`footer_t` structure) that mirrors the header's size and allocated fields. This boundary tag at both ends enables bidirectional coalescing, as we'll discuss in Section 5.2.

Key Design Insights:

> The header's `size` field represents the *usable* size, not the total block size. The total block size includes header, payload (aligned), footer, and any padding. This distinction is critical — when we need to jump to the next physical block in memory, we use the total block size, not the `size` field alone.

Common Pitfalls:

⚠️ **Pitfall: Misaligned User Pointer**

- **Description**: Returning a pointer that isn't aligned to `ALIGNMENT` (8 bytes), causing crashes on systems requiring aligned accesses.
- **Why it's wrong**: Modern CPUs require certain data types to be stored at addresses divisible by their size. An 8-byte double stored at address 0x1001 will cause a segmentation fault on many architectures.
- **How to fix**: After the header, calculate the first address that is a multiple of `ALIGNMENT`. Use the formula: `user_ptr = (char*)header + HEADER_SIZE + (ALIGNMENT - ((HEADER_SIZE % ALIGNMENT) ? HEADER_SIZE % ALIGNMENT : ALIGNMENT))`. Better yet, create an `align_up()` helper function.

⚠️ **Pitfall: Incorrect Size Calculation for Small Allocations**

- **Description**: Not accounting for the case where the user requests fewer bytes than the header size itself.
- **Why it's wrong**: If user requests 1 byte, and header is 24 bytes, the total block size might be miscalculated as 25 bytes, leaving no room for the footer or causing overlap with the next block.
- **How to fix**: Always use `get_total_block_size()` helper which ensures minimum block size is at least `HEADER_SIZE + FOOTER_SIZE + ALIGNMENT`.

⚠️ **Pitfall: Forgetting to Initialize All Header Fields**

- **Description**: Setting only `size` and `allocated` but leaving `next` and `prev` uninitialized (containing garbage values).
- **Why it's wrong**: When the block becomes free, garbage pointers corrupt the free list, causing segfaults during allocation.
- **How to fix**: Always use `init_block()` function that sets all fields, including `next = NULL` and `prev = NULL`, even for allocated blocks.

4.2 Free List Node & Management

Mental Model: Imagine our warehouse has empty shelves scattered throughout the building. Rather than walking through every aisle to find them, the manager maintains a **clipboard list** of all empty shelves. Each empty shelf has a note saying "Next empty shelf: Aisle 3, Row B" and "Previous empty shelf: Aisle 1, Row A." This is an **explicit doubly-linked free list** — each free block contains pointers to its neighbors in the list, allowing us to quickly

traverse all available space without examining allocated blocks. The pointers are stored in the free block's payload area (which would otherwise be unused), so they consume no extra memory.

The free list is the primary data structure for tracking available memory. When a block is freed, it's added to this list; when we need to satisfy an allocation request, we search this list for a suitable block. Our implementation uses an **explicit doubly-linked list** rather than an implicit list (which would require scanning all blocks) for better performance.

Architecture Decision Record: Explicit vs. Implicit Free List

> **Decision: Explicit Doubly-Linked Free List**
>
> - **Context**: We need to track free blocks efficiently. The simplest approach (implicit list) scans all blocks (free and allocated), which becomes O(heap size) for allocation. More sophisticated approaches include explicit lists, segregated fits, or buddy systems.
> - **Options Considered**:
>   1. **Implicit list**: Single list of *all* blocks (free and allocated), using a "next block" pointer derived from size field. Allocation requires scanning entire heap.
>   2. **Explicit singly-linked list**: Free blocks linked via `next` pointers only. O(free blocks) search time but difficult removal of arbitrary blocks.
>   3. **Explicit doubly-linked list**: Free blocks linked via `next` and `prev` pointers. O(free blocks) search with easy arbitrary removal.
>   4. **Segregated lists (future)**: Multiple explicit lists binned by size class for O(1) allocation of small blocks.
> - **Decision**: Start with Option 3 (explicit doubly-linked list) as the foundation, extend to Option 4 in Milestone 3.
> - **Rationale**: Doubly-linked enables O(1) removal when coalescing adjacent free blocks (we need to remove the neighbor from the free list before merging). Singly-linked would require traversing the list to find a block's predecessor. The pointer overhead (16 bytes per free block) is acceptable for a teaching allocator focused on clarity.
> - **Consequences**: Free blocks have 16 bytes overhead for pointers, reducing payload capacity. However, allocation time scales with number of free blocks, not total heap size, which is good for fragmented heaps. The list can become unordered, leading to varying performance based on allocation pattern.

Option	Pros	Cons	Chosen?
Implicit list	No per-block overhead for pointers	O(heap size) allocation time; slow	✗
Explicit singly-linked	O(free blocks) search; simple	Hard to remove arbitrary blocks for coalescing	✗
Explicit doubly-linked	O(free blocks) search; easy removal	16 bytes overhead per free block	✓
Segregated lists	O(1) for small allocations	More complex; internal fragmentation	Future

Free List Structure and Invariants:

The diagram illustrates how free blocks are linked together independently of their physical arrangement in memory. The global `free_list_head` pointer points to the first free block. Each free block's `next` and `prev` pointers form the doubly-linked list. Critical **invariants** (conditions always true) for the free list include:

1. **Null-termination**: `free_list_head->prev == NULL` and the last free block's `next == NULL`
2. **Free status**: Every block in the list has `allocated == 0`
3. **Consistency**: For any free block `B`, if `B->next != NULL`, then `B->next->prev == B`
4. **Non-circular**: The list never forms a cycle (except the trivial case of a single block pointing to itself)

When a block is freed, it must be inserted into the free list. We use **LIFO (Last-In-First-Out) insertion** at the head for simplicity and cache locality:

1. Set new block's `next` to current `free_list_head`
2. Set new block's `prev` to `NULL`
3. If `free_list_head` is not `NULL`, set `free_list_head->prev` to new block
4. Update `free_list_head` to point to new block

Free List Node Embedding:

The free list pointers (`next` and `prev`) are stored in the same memory location that would hold user data if the block were allocated. This is safe because:

1. The block is free, so no user data exists there
2. We have exactly 16 bytes available (on 64-bit) after the header, which matches our two pointer requirement
3. The pointers are accessed via the `block_header_t` structure, treating them as part of the header

Key Design Insights:

> The free list is a *logical* ordering of free blocks, not necessarily reflecting their *physical* order in memory. Two adjacent free blocks in memory might be far apart in the free list. This is why coalescing requires both physical adjacency checks (via size fields) and free list manipulations.

Common Pitfalls:

⚠️ Pitfall: Corrupted Free List Pointers During Coalescing

- **Description**: When merging two adjacent free blocks, failing to properly update the free list pointers of surrounding nodes.
- **Why it's wrong**: If blocks A and B are adjacent and free, and A->next = B, B->prev = A, when we merge them into C, we must remove both A and B from the list and insert C. Forgetting to update A->prev->next or B->next->prev leaves dangling pointers.
- **How to fix**: Always follow the four-step coalescing algorithm: (1) Remove both blocks from free list, (2) Combine their sizes, (3) Create new header/footer for combined block, (4) Insert combined block into free list.

⚠️ Pitfall: Not Handling Empty Free List

- **Description**: Assuming `free_list_head` is always valid when freeing memory.
- **Why it's wrong**: When the first allocation occurs, the free list is empty (`free_list_head == NULL`). Code that does `free_list_head->prev = new_block` will segfault.
- **How to fix**: Always check for `NULL` before dereferencing: `if (free_list_head) free_list_head->prev = new_block;`.

⚠️ Pitfall: Lost Blocks During Thread-Safe Operations

- **Description**: In Milestone 4, when moving blocks between thread-local cache and global arena, forgetting to update pointers consistently under lock.
- **Why it's wrong**: Thread A may see a block in its local cache while Thread B is transferring it to global arena, causing double-allocation or corruption.
- **How to fix**: Use atomic operations or proper locking when moving blocks between domains. The thread-local cache should be treated as owned exclusively by one thread.

4.3 Allocator Global State

Mental Model: The warehouse manager has a **master control panel** showing the overall state of operations: the total warehouse size, the location of the first empty shelf, which picking strategy is currently active (first-fit, best-fit), and separate lists for different shelf sizes. This control panel is the

allocator global state — a single structure that holds all the information needed to manage the entire heap. It's initialized once when the program starts and persists throughout the program's lifetime.

The `allocator_state_t` structure encapsulates the global state of our memory allocator. Unlike the block headers which are distributed throughout the heap, this state is centralized and accessed by all allocation/deallocation functions. It serves as the entry point for finding free memory and contains configuration that affects all allocations.

Architecture Decision Record: Global vs. Distributed State

> **Decision: Centralized Global State Structure**
>
> - **Context**: The allocator needs to maintain various global variables: heap base pointer, free list head, configuration parameters. These could be stored as separate global variables or grouped into a structure.
> - **Options Considered**:
>   1. **Separate global variables**: Individual `extern` variables like `void* heap_base`, `block_header_t* free_list_head`, etc.
>   2. **Centralized structure**: Single `allocator_state_t` instance containing all state.
>   3. **Object-oriented instance**: Multiple allocator instances, each with its own state, allowing independent heaps.
> - **Decision**: Use Option 2 — centralized `allocator_state_t` instance.
> - **Rationale**: Grouping related variables improves code organization and readability. It also makes the allocator more maintainable and easier to extend (adding new fields doesn't change many function signatures). While Option 3 is more flexible, a single global heap is sufficient for our teaching allocator and matches typical `malloc/free` semantics.
> - **Consequences**: All allocator functions must access this structure. In a multithreaded environment (Milestone 4), this becomes a contention point requiring synchronization.

Option	Pros	Cons	Chosen?
Separate globals	Simple access; no indirection	Poor organization; harder to extend	✗
Centralized structure	Clean organization; extensible	Requires passing state pointer or global variable	✓
Object-oriented instances	Multiple independent heaps possible	Complex API; unlike standard malloc/free	✗

The `allocator_state_t` contains the following fields:

Field Name	Type	Description
`heap_base`	`void*`	**Starting address of the heap region**. This points to the first memory address obtained from `os_sbrk()` when the heap was first extended. All block headers are at offsets from this base. Used to validate pointers and for debugging.
`heap_size`	`size_t`	**Current total size of the heap** in bytes. Includes all allocated and free blocks, plus any wilderness (unallocated space at the end). Updated when `os_sbrk()` successfully expands the heap.
`free_list_head`	`block_header_t*`	**Head of the explicit free list**. Points to the first free block in the doubly-linked list of available memory. `NULL` indicates no free memory is available (all blocks are allocated or heap is empty).
`size_classes`	`block_header_t**`	**Array of segregated free list heads** (Milestone 3). Each element is a pointer to the first free block of a particular size class. The array length is determined by the number of size classes (e.g., 8 classes for sizes 16, 32, 64, ..., 2048).
`global_mutex`	`pthread_mutex_t`	**Lock for thread-safe operations** (Milestone 4). Protects access to the global heap state when multiple threads allocate/free concurrently. Thread-local caches may allow some operations without acquiring this lock.
`allocation_strategy`	`enum { FIRST_FIT, BEST_FIT, WORST_FIT }`	**Current search strategy** (Milestone 2). Determines how `find_free_block()` traverses the free list: first-fit (take first sufficient block), best-fit (smallest sufficient block), or worst-fit (largest block).
`mmap_threshold`	`size_t`	**Size threshold for using mmap** (Milestone 4). Requests larger than this value (default `SBRK_THRESHOLD`, 128KB) are allocated directly via `mmap()` instead of from the heap. This prevents large allocations from fragmenting the main heap.
`wilderness`	`block_header_t*`	**Pointer to the wilderness block** (optional optimization). The block at the very end of the heap that can be expanded via `os_sbrk()` when no free block is found. Tracking this separately avoids scanning to find the last block.

State Initialization and Lifetime:

The allocator state must be initialized before any allocation occurs. A typical initialization sequence:

1. Set `heap_base = NULL` and `heap_size = 0` (heap not yet allocated)
2. Set `free_list_head = NULL` (no free blocks initially)
3. Initialize `size_classes = NULL` (segregated lists disabled until configured)
4. Initialize mutex with `pthread_mutex_init(&global_mutex, NULL)`
5. Set default strategy (e.g., `allocation_strategy = FIRST_FIT`)
6. Set `mmap_threshold = SBRK_THRESHOLD`

Because `malloc` and `free` are called from arbitrary points in user code, we need **thread-safe one-time initialization**. The simplest approach is to use a static initializer inside the `malloc` function that checks if `heap_base == NULL` and initializes if so (with proper locking to avoid race conditions).

Configuration Parameters:

Several parameters control allocator behavior and can be tuned for different workloads:

Parameter	Typical Value	Effect
`ALIGNMENT`	8	Minimum alignment guarantee for returned pointers
`PAGE_SIZE`	4096	System memory page size; used for rounding mmap requests
`SBRK_THRESHOLD`	131072 (128KB)	Size above which allocations use mmap instead of sbrk
Size class boundaries	16, 32, 64, 128, 256, 512, 1024, 2048	Buckets for segregated free lists
Minimal block size	`sizeof(block_header_t) + sizeof(footer_t) + ALIGNMENT`	Smallest possible block (including overhead)

Key Design Insights:

> The global state should be the *only* place with static/global variables in the entire allocator. All other functions should take the state (or parts of it) as parameters. This design makes testing easier (we can create multiple allocator instances for tests) and reduces hidden dependencies.

Common Pitfalls:

⚠️ Pitfall: Race Condition During Initialization

- **Description**: Two threads simultaneously call `malloc` for the first time, both see `heap_base == NULL`, and both try to initialize the allocator state.
- **Why it's wrong**: Double initialization may cause memory leaks (two calls to `os_sbrk`) or corruption (overwriting each other's initialization).
- **How to fix**: Use `pthread_once` or a static flag with a mutex. For simplicity in a teaching allocator, we can rely on the program's single-threaded startup phase for initialization, but for Milestone 4 we need proper thread-safe initialization.

⚠️ Pitfall: Not Protecting All State Accesses in Multithreaded Code

- **Description**: Adding locks around some operations (like `malloc`) but forgetting others (like `realloc` or `calloc`).
- **Why it's wrong**: Inconsistent locking leads to data races and corruption, especially on the free list pointers.
- **How to fix**: All public API functions (`malloc`, `free`, `calloc`, `realloc`) must acquire the global mutex (or use thread-local paths) before accessing shared state. Create wrapper functions that handle lock acquisition/release consistently.

⚠️ Pitfall: Assuming State Fits in One Cache Line

- **Description**: The `allocator_state_t` structure may span multiple cache lines (64 bytes typical), causing poor performance when accessed by multiple threads.
- **Why it's wrong**: If two threads on different cores modify different fields that happen to be on the same cache line, they cause "false sharing" and cache line bouncing, slowing down both threads.
- **How to fix**: For production allocators, pad frequently accessed fields (like `free_list_head`) to separate cache lines. For our teaching allocator, we note the issue but accept the performance impact for simplicity.

4.4 Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Data Structure Definition	Standard C structs with typedef	Packed structs with compiler attributes for exact layout control
Alignment Calculation	Manual bitmask operations (`(size + ALIGNMENT - 1) & ~(ALIGNMENT - 1)`)	Builtins like `__builtin_align_up` (GCC/Clang)
Thread-Safe Initialization	Check-then-act with mutex (simpler)	`pthread_once` or C11 `call_once` (more correct)
Debugging Aids	Add `MAGIC` number field to header	Full canary values and heap consistency checking functions

Recommended File/Module Structure:

```
allocator/
├── include/
│   └── allocator.h          ← Public API and data structure definitions
├── src/
│   ├── allocator.c          ← Main malloc/free implementations
│   ├── block.c              ← Block management (init, split, coalesce)
│   ├── freelist.c           ← Free list operations (insert, remove, search)
│   ├── strategies.c         ← First-fit, best-fit, worst-fit implementations
│   ├── segregated.c         ← Size class management (Milestone 3)
│   ├── thread_cache.c       ← Per-thread arenas (Milestone 4)
│   └── os_mem.c             ← OS interface (sbrk, mmap wrappers)
└── tests/
    ├── test_basic.c         ← Milestone 1 tests
    ├── test_freelist.c      ← Milestone 2 tests
    └── test_concurrent.c    ← Milestone 4 tests
```

Core Data Structure Starter Code:

```c
// allocator.h

#ifndef ALLOCATOR_H

#define ALLOCATOR_H


#include <stddef.h>

#include <stdint.h>

#include <pthread.h>


// Constants (must match NAMING CONVENTIONS)

#define ALIGNMENT 8

#define PAGE_SIZE 4096

#define SBRK_THRESHOLD (128 * 1024)

#define MAGIC 0xDEADBEEF


// Block header structure - MUST match exact naming convention

typedef struct block_header {

    size_t size;                    // Usable size (excluding header/footer)

    int allocated;                  // 1 if allocated, 0 if free

    struct block_header* next;      // Next free block (valid only if free)

    struct block_header* prev;      // Previous free block (valid only if free)

    #ifdef DEBUG

    uint32_t magic;                 // Debugging aid to detect corruption

    #endif

} block_header_t;


// Footer structure (for boundary tags)

typedef struct footer {

    size_t size;                    // Mirror of header's size

    int allocated;                  // Mirror of header's allocated flag

} footer_t;


// Allocation strategies

typedef enum {

    FIRST_FIT,

    BEST_FIT,

    WORST_FIT

} strategy_t;


// Global allocator state
```

```c
typedef struct allocator_state {
    void* heap_base;                // Starting address of heap
    size_t heap_size;               // Current total heap size
    block_header_t* free_list_head; // Head of explicit free list
    block_header_t** size_classes;  // Array of segregated list heads
    pthread_mutex_t global_mutex;   // Lock for thread safety
    strategy_t allocation_strategy; // Current search strategy
    size_t mmap_threshold;          // Threshold for using mmap
    // Additional fields as needed...
} allocator_state_t;

// Public API functions
void* malloc(size_t size);
void free(void* ptr);
void* calloc(size_t nmemb, size_t size);
void* realloc(void* ptr, size_t size);

// Debugging function
void heap_visualize(void);

#endif // ALLOCATOR_H
```

Block Management Helper Functions (Skeleton Code):

```c
// block.c

#include "allocator.h"

#include <stdint.h>


// Calculate total block size including header, footer, alignment padding

size_t get_total_block_size(size_t user_size) {

    // TODO 1: Calculate minimum required size: header + footer + user_size

    // TODO 2: Round up to meet alignment requirements for user payload

    // TODO 3: Ensure result is at least minimum block size (header+footer+ALIGNMENT)

    // TODO 4: Return the total size

    return 0;

}


// Convert user pointer back to its block header

block_header_t* get_header_from_ptr(void* ptr) {

    // TODO 1: Check if ptr is NULL (return NULL)

    // TODO 2: Calculate header address: ptr minus header size

    // TODO 3: Optionally validate magic number in DEBUG mode

    // TODO 4: Return pointer to header

    return NULL;

}


// Initialize a block at given memory location

void init_block(void* memory, size_t user_size, int allocated) {

    // TODO 1: Cast memory to block_header_t*

    // TODO 2: Set size field to user_size

    // TODO 3: Set allocated field to allocated

    // TODO 4: Set next and prev to NULL

    // TODO 5: Set magic number if DEBUG

    // TODO 6: Calculate footer position (end of block minus footer size)

    // TODO 7: Initialize footer with same size and allocated values

}
```

Free List Management Helper Functions (Skeleton Code):

```c
// freelist.c

#include "allocator.h"


// Insert a free block at the head of the free list

void free_list_insert(block_header_t* block) {

    // TODO 1: Set block->allocated = 0

    // TODO 2: Get current free_list_head from global state

    // TODO 3: Set block->next = free_list_head

    // TODO 4: Set block->prev = NULL

    // TODO 5: If free_list_head is not NULL, set free_list_head->prev = block

    // TODO 6: Update global free_list_head to point to block

    // TODO 7: Update block's footer to reflect free status

}


// Remove a block from the free list (when allocating or coalescing)

void free_list_remove(block_header_t* block) {

    // TODO 1: Handle case when block is the head (update free_list_head)

    // TODO 2: If block has a previous node, update its next pointer

    // TODO 3: If block has a next node, update its prev pointer

    // TODO 4: Set block->next = NULL and block->prev = NULL (optional)

}
```

Language-Specific Hints:

- Use `sizeof(block_header_t)` and `sizeof(footer_t)` rather than hardcoded sizes for portability
- The `offsetof` macro from `<stddef.h>` can help calculate precise offsets
- For alignment rounding: `((size + ALIGNMENT - 1) & ~(ALIGNMENT - 1))` works for power-of-two ALIGNMENT
- In C, structure padding may add invisible bytes; use `#pragma pack(1)` or `__attribute__((packed))` if exact layout is critical
- Always cast pointers to `char*` for byte-wise arithmetic, then cast back to appropriate type

Milestone Checkpoint for Data Structures: After implementing the data structures (before writing malloc/free):

1. Compile: `gcc -c -DDEBUG -o block.o block.c`
2. Write a simple test: Create a static buffer, call `init_block()` on it, then verify via `get_header_from_ptr()` that you can retrieve the correct header
3. Expected: Header fields should match what you initialized, and `get_header_from_ptr(user_ptr)` should return the original header address
4. If it fails: Check your offset calculations — print addresses and sizes to debug

Debugging Tips:

- Symptom: `get_header_from_ptr()` returns wrong address
  - Likely Cause: Incorrect assumption about header size due to padding
  - How to Diagnose: Print `sizeof(block_header_t)` and compare with manual calculation
  - Fix: Use `offsetof(block_header_t, magic)` (if magic is last field) to get exact header size
- Symptom: Block footer doesn't align with next block's header
  - Likely Cause: `get_total_block_size()` not accounting for alignment padding correctly
  - How to Diagnose: Print total size and verify it's multiple of ALIGNMENT

- Fix: Use helper: `align_up(size, ALIGNMENT)` that rounds up to next multiple

5.1 OS Memory Interface Component

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 4 (Thread Safety & mmap)

The OS Memory Interface Component serves as the **foundational layer** between our teaching allocator and the operating system's virtual memory management. It abstracts the low-level system calls that actually obtain and release physical memory pages, presenting a clean, unified interface to the rest of the allocator. Without this component, our allocator would be a theoretical design with no way to interact with real hardware memory.

Mental Model: The Warehouse Landlord

Imagine you're managing a growing warehouse complex for storing packages of various sizes. The **operating system** acts as the **landlord** who controls all available land. Your warehouse (the **heap**) is built on a plot of land the landlord provides. You have two primary ways to expand your storage capacity:

1. **`sbrk` — Expanding Your Existing Plot**: You can request the landlord to extend your existing plot by pushing back the fence (`brk` stands for "break" or boundary). This gives you more **contiguous space** adjacent to your current warehouse. It's simple and efficient for gradual growth, but you can only expand in one direction, and once you've built shelves in the extended area, it becomes permanently part of your warehouse layout. If you later have empty shelves in the middle, you can't easily give that specific section back to the landlord — you can only shrink from the very end.
2. **`mmap` — Renting Separate Storage Units**: For exceptionally large items or temporary storage needs, you can rent a completely **separate storage unit** located elsewhere in the industrial park (a different region of the virtual address space). This unit is independent: you get exactly the size you need, you can return it anytime without affecting your main warehouse, and it doesn't create holes in your primary storage area. However, each rental requires more paperwork (system call overhead), and having many scattered units makes management more complex.

This mental model clarifies the fundamental trade-off: `sbrk` provides simple, contiguous growth ideal for small, frequent allocations (the main warehouse), while `mmap` offers isolated, flexible storage perfect for large, one-off allocations (specialized storage units).

ADR: sbrk for Small, mmap for Large Allocations

Decision: Hybrid sbrk/mmap Strategy

- **Context**: Our teaching allocator must manage memory efficiently across the entire size spectrum — from a few bytes to megabytes. Using only `sbrk` for all allocations leads to severe **external fragmentation** when large blocks are freed, leaving unusable holes in the main heap. Using only `mmap` would incur excessive system call overhead for small allocations. We need a strategy that balances performance with fragmentation control.
- **Options Considered**:
  1. **Pure sbrk-based heap**: Request all memory via `sbrk`, maintaining a single contiguous address range.
  2. **Pure mmap per allocation**: Map every allocation as a separate `mmap` region, regardless of size.
  3. **Threshold-based hybrid**: Use `sbrk` for allocations below a threshold (e.g., 128KB) and `mmap` for larger allocations.
- **Decision**: Implement a **threshold-based hybrid strategy** where allocations smaller than `SBRK_THRESHOLD` (128KB) are satisfied from the main `sbrk` heap, while larger allocations use direct `mmap`.
- **Rationale**:
  - **Fragmentation control**: Large `mmap`'ed blocks exist independently and can be returned to the OS immediately upon `free()`, preventing them from creating permanent holes in the main heap that would hinder smaller allocations.
  - **Performance efficiency**: Small allocations from `sbrk` avoid the per-allocation system call overhead of `mmap` (which involves kernel-level page table updates and zero-filling).
  - **Pedagogical value**: Demonstrates real-world allocator design patterns used by `glibc`'s `malloc` and other production allocators.
  - **Simplicity**: The threshold is easily tunable and doesn't require complex heuristics for the teaching implementation.
- **Consequences**:
  - **Positive**: Reduces external fragmentation in the main heap; allows immediate memory return to OS for large blocks; maintains good performance for common small allocations.
  - **Negative**: Introduces complexity in `free()` — must detect whether a pointer came from `sbrk` heap or `mmap` region; creates non-contiguous address space layout; requires tracking multiple memory regions.

The following table summarizes the trade-offs between the considered approaches:

Option	Pros	Cons	Why Not Chosen?
Pure sbrk	Simple implementation; Contiguous address space; Low per-allocation overhead	Severe external fragmentation from large freed blocks; Cannot return memory to OS except at heap end	Fails to address fragmentation — large freed blocks create permanent holes
Pure mmap	Eliminates external fragmentation (each block isolated); Immediate memory return to OS; Flexible size allocation	High system call overhead for small allocations; Poor spatial locality (scattered blocks); Complex free list management across disjoint regions	Performance unacceptable for teaching — real programs make thousands of small allocations
Hybrid threshold	Balances fragmentation and performance; Matches real-world practice; Tunable threshold	More complex free() logic; Non-contiguous address space	**CHOSEN** — Best trade-off for teaching realistic allocator design

Component Design

The OS Memory Interface Component exposes two primary operations to the rest of the allocator: requesting new memory and releasing memory back to the OS. Internally, it routes requests to the appropriate system call based on size and maintains metadata about each memory region's origin.

Data Structures

The component manages two types of memory regions, each with different characteristics:

Region Type	Creation Method	Release Method	Metadata Location	Purpose
sbrk Heap	`sbrk()` system call	Can only shrink from end via negative `sbrk()`	Embedded in the heap itself (headers/footers)	Main allocation arena for small/medium blocks
mmap Region	`mmap()` system call	`munmap()` system call	Separate tracking structure or header	Large allocations above threshold

For `mmap` regions, we need a way to distinguish them from `sbrk` allocations during `free()`. Two common approaches:

Approach	Implementation	Pros	Cons
Flag in Block Header	Set a special flag (e.g., `IS_MMAP`) in the `allocated` field	Simple; No extra data structures	Only works if we can access the header (requires pointer math)
Boundary Check	Check if address falls within known `sbrk` heap range	No header modification needed	Requires tracking heap bounds; Fragile if mmap regions intersect range
Separate Registry	Maintain hash table of mmap'ed addresses	Explicit tracking; Can store size separately	Additional memory overhead; Synchronization needed for threads

For our teaching allocator, we'll use the **flag in block header** approach for its simplicity and consistency with our existing metadata design.

Interface Methods

The component provides the following core interface functions:

Method	Parameters	Returns	Description	Side Effects
`os_sbrk`	`intptr_t increment`	`void*` (old break) or `(void*)-1` on error	Wrapper for `sbrk()` system call with error checking	Changes program break; May extend heap segment
`os_mmap`	`size_t size`	`void*` mapped address or `MAP_FAILED`	Allocates anonymous, private memory pages via `mmap()`	Creates new memory mapping in process address space
`os_munmap`	`void* addr, size_t size`	`int` (0 success, -1 error)	Releases memory mapped by `os_mmap()`	Removes memory mapping; Pages returned to OS
`get_memory_from_os`	`size_t total_size`	`void*` to usable memory or `NULL`	Main entry point: chooses `sbrk` or `mmap` based on size threshold	May extend heap or create new mapping
`return_memory_to_os`	`void* ptr, size_t size, int is_mmap`	`int` success/failure	Releases memory using appropriate method (`sbrk` shrink or `munmap`)	May shrink heap or remove mapping

Algorithm: Obtaining Memory from OS

When the allocator needs more memory (because the free list has no suitable blocks), it calls `get_memory_from_os()`:

1. **Calculate total needed size**: Include user request size plus metadata (header, footer, alignment padding).
2. **Check threshold**: Compare total size against `SBRK_THRESHOLD`.
3. **Small allocation path (sbrk)**:
    1. Call `os_sbrk(total_size)` to extend the heap.
    2. If `sbrk` fails (returns `(void*)-1`), return `NULL` to indicate out-of-memory.
    3. Initialize a new block header at the returned address.
    4. Mark the block as allocated and set the `IS_MMAP` flag to 0.
    5. Return pointer to the user payload area.
4. **Large allocation path (mmap)**:
    1. Round size up to page boundary using `align_to_page(total_size)`.

2. Call `os_mmap(rounded_size)` to obtain new memory pages.
3. If `mmap` fails (returns `MAP_FAILED`), return `NULL`.
4. Initialize a block header at the start of the mapped region.
5. Mark the block as allocated and set the `IS_MMAP` flag to 1.
6. Store the actual mapped size in the header for later `munmap`.
7. Return pointer to the user payload area.

Algorithm: Releasing Memory to OS

When `free()` determines a block should be returned to the OS (either because it's a large `mmap` block or it's at the end of the `sbrk` heap), it calls `return_memory_to_os()`:

1. **Check block type** via `IS_MMAP` flag in header.
2. **mmap block path**:
   1. Calculate original mapped size (stored in header or rounded to pages).
   2. Call `os_munmap(block_start, mapped_size)`.
   3. If successful, the memory is immediately available to the OS.
3. **sbrk block at heap end path** (optional optimization):
   1. Check if block is at the current program break (end of heap).
   2. Calculate shrink amount (block size including metadata).
   3. Call `os_sbrk(-shrink_amount)` to return memory to OS.
   4. Update internal heap size tracking.

Error Handling Table

The component must robustly handle system call failures:

Failure Mode	Detection Method	Recovery Action	Propagated To Caller
`sbrk` fails (out of memory)	Return value `(void*)-1`	Log error if debugging enabled	`malloc` returns `NULL`
`mmap` fails (address space exhausted)	Return value `MAP_FAILED`	Log error if debugging enabled	`malloc` returns `NULL`
`munmap` fails (invalid arguments)	Return value `-1`	Log error; Continue (memory leaked)	Ignored (non-fatal)
Heap corruption (invalid header)	Magic number verification in `get_header_from_ptr`	Abort program with error message	Program termination

Common Pitfalls

⚠️ Pitfall: Forgetting to handle `sbrk` failure

- **Description**: Calling `sbrk` without checking if it returned `(void*)-1`, then using the invalid pointer.
- **Why it's wrong**: The OS may deny memory expansion (virtual memory exhausted, ulimit restrictions). Using `(void*)-1` as a valid pointer causes segmentation faults.
- **Fix**: Always check `sbrk` return value; propagate `NULL` up to `malloc` caller.

⚠️ Pitfall: Incorrect page alignment for `mmap`

- **Description**: Passing unaligned size to `mmap`, causing the kernel to round up but storing the original size in metadata.
- **Why it's wrong**: When `munmap` is called with the original (smaller) size, it may only partially unmap or cause errors because `munmap` requires page-aligned arguments.
- **Fix**: Always use `align_to_page()` on both size and address for `mmap`/`munmap` calls.

⚠️ Pitfall: Mixing `sbrk` and `mmap` pointers in free lists

- **Description**: Placing `mmap`'ed blocks in the same free list as `sbrk` heap blocks.
- **Why it's wrong**: `mmap` blocks have different lifetime rules (can be immediately unmapped) and may not have adjacent blocks for coalescing.

- **Fix**: Maintain separate tracking for `mmap` blocks or mark them clearly so `free()` handles them differently.

⚠️ Pitfall: Not zeroing `mmap` memory

- **Description**: Assuming `mmap` returns zeroed memory (not guaranteed without `MAP_ANONYMOUS` flag properly set).
- **Why it's wrong**: May expose sensitive data from previously mapped memory (security issue) or cause undefined behavior if code assumes zero-initialization.
- **Fix**: Ensure `mmap` flags include `MAP_ANONYMOUS` | `MAP_PRIVATE` and consider explicit zeroing for security-critical applications.

Implementation Guidance

Technology Recommendations

Component	Simple Option (Teaching Focus)	Advanced Option (Production-like)
Memory Mapping	`mmap` with `MAP_ANONYMOUS | MAP_PRIVATE`	`mmap` with `MAP_ANONYMOUS | MAP_PRIVATE | MAP_NORESERVE` for overcommit
Error Reporting	`perror()` with descriptive messages	Structured logging with log levels
Alignment	Manual rounding with bit operations	`posix_memalign`-style API for strict alignments
Thread Safety	None in this component (handled higher up)	Atomic operations for size tracking

Recommended File/Module Structure

```
allocator/
├── include/
│   └── allocator.h          # Public API, type definitions
├── src/
│   ├── os_mem.c             # OS Memory Interface Component (this file)
│   ├── heap.c               # Heap management core logic
│   ├── free_list.c          # Free list operations
│   ├── strategies.c         # Allocation strategies
│   └── thread_cache.c       # Thread-local caching
└── tests/
    └── test_os_mem.c        # Unit tests for this component
```

Infrastructure Starter Code

Here's complete, working code for the OS abstraction layer. These functions handle system-specific details and error checking, allowing learners to focus on allocator logic:

```c
/* File: src/os_mem.c */

#include <unistd.h>

#include <sys/mman.h>

#include <errno.h>

#include <string.h>

#include "allocator.h"


// Wrapper for sbrk with error checking

void* os_sbrk(intptr_t increment) {

    void* old_break = sbrk(increment);

    if (old_break == (void*)-1) {

        // Log error if debugging is enabled

        #ifdef DEBUG

        perror("sbrk failed");

        #endif

        return (void*)-1;

    }

    return old_break;

}


// Wrapper for mmap to allocate anonymous, private memory

void* os_mmap(size_t size) {

    // Note: MAP_ANONYMOUS means no file backing, contents initialized to zero

    // MAP_PRIVATE means changes are not shared with other processes

    void* ptr = mmap(NULL, size,

                     PROT_READ | PROT_WRITE,

                     MAP_ANONYMOUS | MAP_PRIVATE,

                     -1, 0);  // fd = -1 for anonymous mapping



    if (ptr == MAP_FAILED) {

        #ifdef DEBUG

        perror("mmap failed");

        #endif

        return MAP_FAILED;

    }

    return ptr;

}
```

```c
// Wrapper for munmap to release mapped memory
int os_munmap(void* addr, size_t size) {
    int result = munmap(addr, size);
    if (result == -1) {
        #ifdef DEBUG
        perror("munmap failed");
        #endif
    }
    return result;
}

// Round size up to nearest multiple of page size (4096 bytes)
size_t align_to_page(size_t size) {
    return (size + PAGE_SIZE - 1) & ~(PAGE_SIZE - 1);
}
```

Core Logic Skeleton Code

Here's the main entry point function that chooses between `sbrk` and `mmap`. Learners should implement this:

```c
/*
 * Request memory from the operating system.
 * For small allocations (< SBRK_THRESHOLD), use sbrk to extend the heap.
 * For large allocations (>= SBRK_THRESHOLD), use mmap for separate regions.
 *
 * Parameters:
 *   total_size - Total memory needed including metadata and alignment
 *   is_large   - Output parameter: set to 1 if mmap was used, 0 if sbrk
 *
 * Returns:
 *   Pointer to the start of usable memory (after header) or NULL on failure
 */
void* get_memory_from_os(size_t total_size, int* is_large) {
    void* block_start = NULL;

    // TODO 1: Check if this is a large allocation by comparing total_size
    //         with SBRK_THRESHOLD constant

    // TODO 2: If it's a small allocation (use sbrk):
    //   a. Call os_sbrk(total_size) to extend the heap
    //   b. If os_sbrk returns (void*)-1, set errno to ENOMEM and return NULL
    //   c. Set block_start to the returned pointer
    //   d. Set *is_large = 0 to indicate sbrk allocation

    // TODO 3: If it's a large allocation (use mmap):
    //   a. Round total_size up to page boundary using align_to_page()
    //   b. Call os_mmap(rounded_size) to get new memory pages
    //   c. If os_mmap returns MAP_FAILED, set errno to ENOMEM and return NULL
    //   d. Set block_start to the returned pointer
    //   e. Set *is_large = 1 to indicate mmap allocation

    // TODO 4: Initialize the block header at block_start:
    //   a. Cast block_start to block_header_t*
    //   b. Set header->size to total_size (or rounded_size for mmap)
    //   c. Set header->allocated = 1 (allocated)
    //   d. For mmap blocks, set a flag (e.g., header->size |= IS_MMAP_FLAG)
```

```
    //   e. For debugging, set header->magic = MAGIC

    // TODO 5: Return pointer to user area (after header)
    //   Hint: return (char*)block_start + sizeof(block_header_t)

    return NULL;  // Placeholder
}

/*
 * Release memory back to the operating system.
 * For mmap blocks, use munmap immediately.
 * For sbrk blocks at heap end, optionally shrink heap (advanced).
 *
 * Parameters:
 *   block      - Pointer to block header (start of block including metadata)
 *   block_size - Total size of the block including metadata
 *   is_mmap    - 1 if block was allocated via mmap, 0 if via sbrk
 */
void return_memory_to_os(block_header_t* block, size_t block_size, int is_mmap) {
    // TODO 1: If is_mmap is true (mmap allocation):
    //   a. Get the original mapped size (may be larger due to page rounding)
    //   b. Calculate original mapped address (block pointer)
    //   c. Call os_munmap with address and size
    //   d. No need to update free lists (block is completely gone)

    // TODO 2: If is_mmap is false (sbrk allocation) and this is at heap end:
    //   Note: This is an OPTIONAL optimization for advanced implementations
    //   a. Check if block is at current program break (requires tracking heap top)
    //   b. If yes, calculate negative increment = -block_size
    //   c. Call os_sbrk(negative_increment) to shrink heap
    //   d. Update heap size tracking

    // TODO 3: If neither above applies (sbrk block not at end):
    //   Just return - block will remain in heap for future reuse
    //   (This is handled by free list management in other components)
}
```

Language-Specific Hints

- **System Call Compatibility**: `sbrk` is deprecated on some systems (macOS) but works on Linux for teaching. For portability, consider using `mmap` for all allocations, but for this project we'll use `sbrk` as specified.
- **Error Handling**: Always check system call return values. `sbrk` returns `(void*)-1` on error; `mmap` returns `MAP_FAILED` (typically `(void*)-1`).
- **Page Size**: Don't hardcode `4096`; use `sysconf(_SC_PAGESIZE)` at runtime for portability, though for simplicity we'll use the `PAGE_SIZE` constant.
- **Memory Initialization**: `mmap` with `MAP_ANONYMOUS` returns zero-filled pages. `sbrk` does NOT initialize memory—contents are undefined.
- **Thread Safety**: These system calls are thread-safe at the OS level, but our wrapper functions should be called with appropriate synchronization (handled in Milestone 4).

Milestone Checkpoint

After implementing the OS Memory Interface Component (Milestone 1 foundation):

1. **Compile a test program**:

```bash
gcc -DDEBUG -o test_os src/os_mem.c test_os_mem.c
```

2. **Run the basic test**:

```bash
./test_os
```

3. **Expected behavior**:
   - Program should print "Testing sbrk allocation..." and allocate small memory successfully.
   - "Testing mmap allocation..." should allocate larger memory successfully.
   - No segmentation faults or error messages (unless you intentionally test error paths).
   - Program exits normally with "All OS memory tests passed."
4. **If tests fail**:
   - **"sbrk failed: Cannot allocate memory"**: Check ulimit -v (virtual memory limit) or system memory availability.
   - **"mmap failed: Cannot allocate memory"**: May indicate address space exhaustion (unlikely on 64-bit).
   - **Segmentation fault**: Likely using invalid pointer from failed system call without checking return value.
5. **Manual verification**:

```c
// Simple test program

int main() {

    int is_large;

    void* small = get_memory_from_os(100, &is_large);

    printf("Small alloc: %p, is_large: %d\n", small, is_large);



    void* large = get_memory_from_os(200000, &is_large);  // > 128KB

    printf("Large alloc: %p, is_large: %d\n", large, is_large);



    return 0;

}
```

Should show two different pointers with `is_large` being 0 then 1.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
`sbrk` always returns `(void*)-1`	Virtual memory limit reached	Check `ulimit -v`; run `strace` to see system calls	Increase limit or use `mmap` instead
`mmap` returns unaligned pointer	Wrong flags or parameters	Print pointer value: `printf("%p\n", ptr)` should be page-aligned	Ensure using `MAP_ANONYMOUS` and correct size rounding
Memory corruption after `mmap`	Forgetting page rounding in `munmap`	Store original mapped size in header; compare with `munmap` size	Always use `align_to_page()` for both `mmap` and `munmap` sizes
Can't distinguish mmap/sbrk blocks in `free()`	Missing flag in header	Check header->allocated field for special mmap flag	Set a reserved bit in size or allocated field when allocating via mmap
Heap grows indefinitely	Never calling `sbrk` with negative increment	Track heap size; check if freed block is at end	Implement heap shrinkage optimization for end blocks

---

5.2 Block Management & Core Algorithms

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management)

This component forms the **beating heart** of the memory allocator. While the OS Memory Interface provides raw memory (land), this component organizes that land into usable parcels, tracks what's occupied and what's available, and handles the intricate operations of finding, dividing, and merging memory blocks. It transforms a chaotic heap of bytes into a structured, manageable resource.

Mental Model: The Warehouse Floor Plan

Imagine the **heap** as a massive, continuous warehouse floor. The warehouse manager (your allocator) needs to rent out shelf space to different merchants (program data structures). Each merchant requests a specific amount of space, but the warehouse only provides space in fixed "shelf units" that include a small administrative area at the front.

- **Block**: A single shelf unit. Each block has a fixed location on the floor and consists of two parts:
  1. **Header/Label**: A small, fixed-size administrative area at the front of the shelf. It stores the shelf's total capacity and whether it's currently rented out.
  2. **Payload/Storage**: The main storage area the merchant actually uses. This is the pointer returned by `malloc`.
- **Free List**: A clipboard the manager uses to track all empty shelf units. It doesn't list their *locations* in order; instead, each empty shelf stores the *address* of the next empty shelf and the previous empty shelf right in its own storage area (since it's empty, that space isn't being used). This forms a chain of empty shelves scattered throughout the warehouse.
- **Operations**:
  - **Finding a shelf (`malloc`)**: The manager checks the clipboard (free list) for an empty shelf big enough for the merchant's request.
  - **Splitting a shelf**: If the empty shelf is much larger than needed, the manager can build a partition, converting part of it into a rented shelf (with its own label) and leaving the remainder as a smaller empty shelf (added back to the clipboard).
  - **Merging shelves (`free` and coalescing)**: When a merchant leaves (calls `free`), their shelf becomes empty. An astute manager will check if the shelves immediately to the left and right are also empty. If they are, the manager can knock down the walls between them, creating one large empty shelf. This prevents the warehouse from becoming fragmented into many small, useless spaces.

This mental model clarifies the core challenge: efficiently organizing a one-dimensional space (the heap address range) to satisfy random-sized allocation and deallocation requests while minimizing wasted gaps.

ADR: Boundary Tags for Bidirectional Coalescing

> **Decision: Use Boundary Tags (Headers + Footers) for O(1) Bidirectional Coalescing**

- **Context**: When a block is freed, to combat **external fragmentation**, we must merge it with any adjacent free blocks. A block only knows its own size and status from its header. To find the "next" block physically in memory, we can add the block's total size (from its header) to its starting address. However, finding the *previous* block is impossible with only a forward header—we would need to scan the entire heap from the beginning to find which block ends just before the current one, an O(n) operation unacceptable for performance.
- **Options Considered**:
  1. **Headers Only (Unidirectional Coalescing)**: Only store metadata at the block's start. When freeing, only coalesce with the *next* block if it's free. Ignore the previous block. This is simple but leaves significant fragmentation.
  2. **Boundary Tags (Headers + Footers)**: Store a copy of the block's size and allocation status at both the start (**header**) and the end (**footer**). The footer of the *previous* block is located at a fixed offset before the current header, allowing immediate access.
  3. **Embedded Previous Block Pointer in Header**: Augment the header with a direct pointer to the previous block's header. This provides O(1) access but increases the header size for *all* blocks, including allocated ones where the pointer is wasted.
- **Decision**: Implement **Boundary Tags**.
- **Rationale**:
  - **Fragmentation Reduction**: Enables merging with *both* adjacent free blocks in constant time, dramatically reducing external fragmentation compared to unidirectional coalescing.
  - **Space Overhead is Manageable**: The footer is only needed for coalescing logic. Its space is only "wasted" in a technical sense—it resides within the block's own allocated space. For free blocks, the footer space overlaps with the area used for free list pointers. For allocated blocks, the footer occupies part of the user's payload, but we account for this in size calculations, so the user gets the requested amount *after* the footer.
  - **Simplicity and Reliability**: The footer is a simple copy of key header fields. Finding the previous block's footer is a simple address calculation (`current_block_address - sizeof(footer_t)`), which is robust and fast.
- **Consequences**:
  - **Increased Metadata Overhead**: Every block, allocated or free, now has a footer. This increases **internal fragmentation** slightly, as we must round up allocation sizes to ensure the footer fits within the block and maintains alignment.
  - **More Complex Block Management**: Functions that manipulate block size or status (like `split_block`, `coalesce_block`) must now update *both* the header and the footer to keep them consistent.
  - **Enables Efficient Algorithms**: The O(1) access to the previous block's status is foundational for high-performance allocators and is used in many production systems.

Option	Pros	Cons	Chosen?
Headers Only	Minimal metadata, simple implementation.	Severe external fragmentation (no backward coalescing), poor heap utilization.	❌
Boundary Tags	O(1) bidirectional coalescing, significant fragmentation reduction.	Extra 8-16 bytes per block (footer), must update two metadata locations.	✅
Previous Pointer in Header	O(1) access to previous block, no footer needed.	Increases header size for all blocks (8 extra bytes on 64-bit), wastes space in allocated blocks.	❌

The block layout with boundary tags is visualized below. Notice how the footer of Block N-1 sits immediately before the header of Block N, enabling the `coalesce_block` function to check the previous block's status with a simple subtraction.

Core Data Structures and Interfaces

The block management system revolves around a few critical data structures. The `block_header_t` and `footer_t` are defined in the Data Model section. Here, we detail their roles and the functions that manipulate them.

Block Header and Footer Fields:

Field Name	Type (C)	Description
`size`	`size_t`	**Total size of the memory block in bytes**, including the header, user payload, footer, and any alignment padding. This is the most critical field. The least significant bit is often repurposed as an `allocated` flag (see below).
`allocated`	`int` (or bit flag within `size`)	Boolean flag indicating if the block is currently allocated (`1`) or free (`0`). In many implementations, this is stored in the LSB of the `size` field (since block sizes are always multiples of alignment, the LSB is free). We treat it separately for clarity.
`next`	`struct block_header_t*`	(Free blocks only) Pointer to the next free block in the explicit free list. Stored within the payload area of the free block.
`prev`	`struct block_header_t*`	(Free blocks only) Pointer to the previous free block in the explicit free list. Stored within the payload area of the free block.

> **Design Insight:** The `next` and `prev` pointers are only valid when the block is free. When the block is allocated, that memory belongs to the user program. This is an example of **overlaying data structures** to save space—a common technique in systems programming.

Core Block Management Functions:

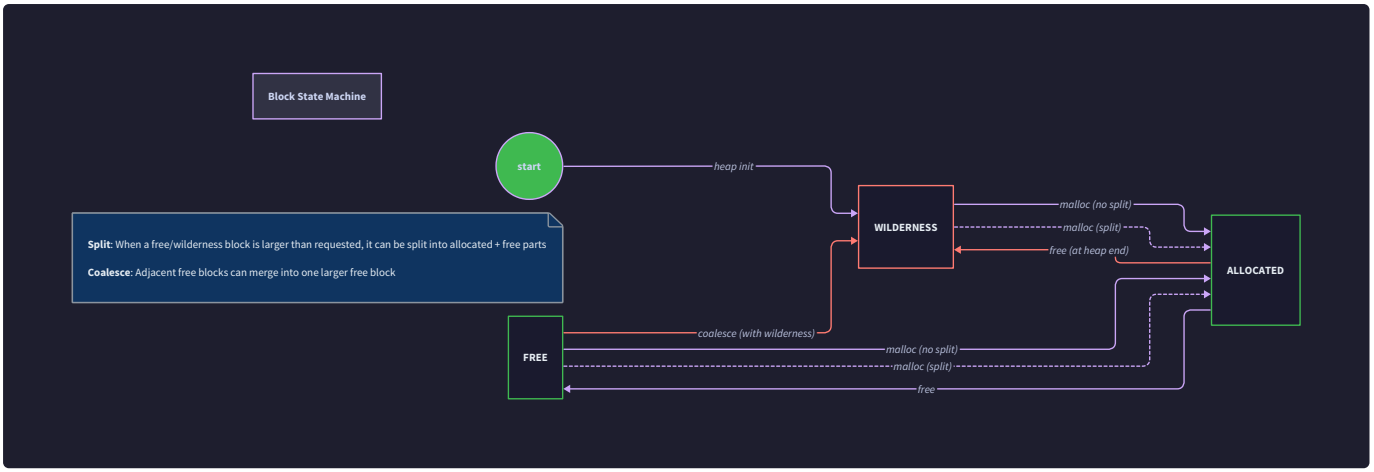
These functions are the internal workhorses called by the public `malloc` and `free` APIs.

Method Name	Parameters	Returns	Description
`get_header_from_ptr`	`void *ptr`	`block_header_t*`	Given a pointer returned to the user (payload start), calculates and returns a pointer to the block's header. This is **the fundamental operation** for `free`.
`get_total_block_size`	`size_t user_size`	`size_t`	Calculates the total size needed for a block to satisfy a user request for `user_size` bytes. This includes the header, the user payload (aligned), the footer, and ensures the total size meets minimum block size requirements for free list pointers.
`init_block`	`void *memory, size_t user_size, int allocated`	`void`	Initializes a raw region of memory at `memory` to be a valid block of the appropriate total size (calculated via `get_total_block_size`). Sets the header and footer `size` and `allocated` fields. If the block is free, it does NOT link it into the free list.
`split_block`	`block_header_t *block, size_t needed_size`	`block_header_t*`	Given a large free `block`, carves off a portion of size `needed_size` (total block size) to satisfy an allocation. The remainder becomes a new, smaller free block. Returns a pointer to the header of the allocated portion. Returns `NULL` if the block cannot be split (e.g., remainder would be too small).
`coalesce_block`	`block_header_t *block`	`block_header_t*`	Given a newly freed `block`, checks its immediate physical neighbors in the heap (using boundary tags). If either neighbor is free, merges them into a single, larger free block. Returns a pointer to the header of the resulting merged block.
`find_free_block`	`size_t size`	`block_header_t*`	Scans the **explicit free list** for a block whose total size is at least `size`. The search strategy (first-fit, best-fit, worst-fit) is determined by a global configuration variable. Returns `NULL` if no suitable block is found.
`free_list_insert`	`block_header_t* block`	`void`	Inserts a free block at the **head** of the explicit doubly-linked free list for fast allocation.
`free_list_remove`	`block_header_t* block`	`void`	Removes a block (which must be free) from the explicit free list. This is called when a free block is chosen for allocation or when it is merged during coalescing.

Block State Transitions:

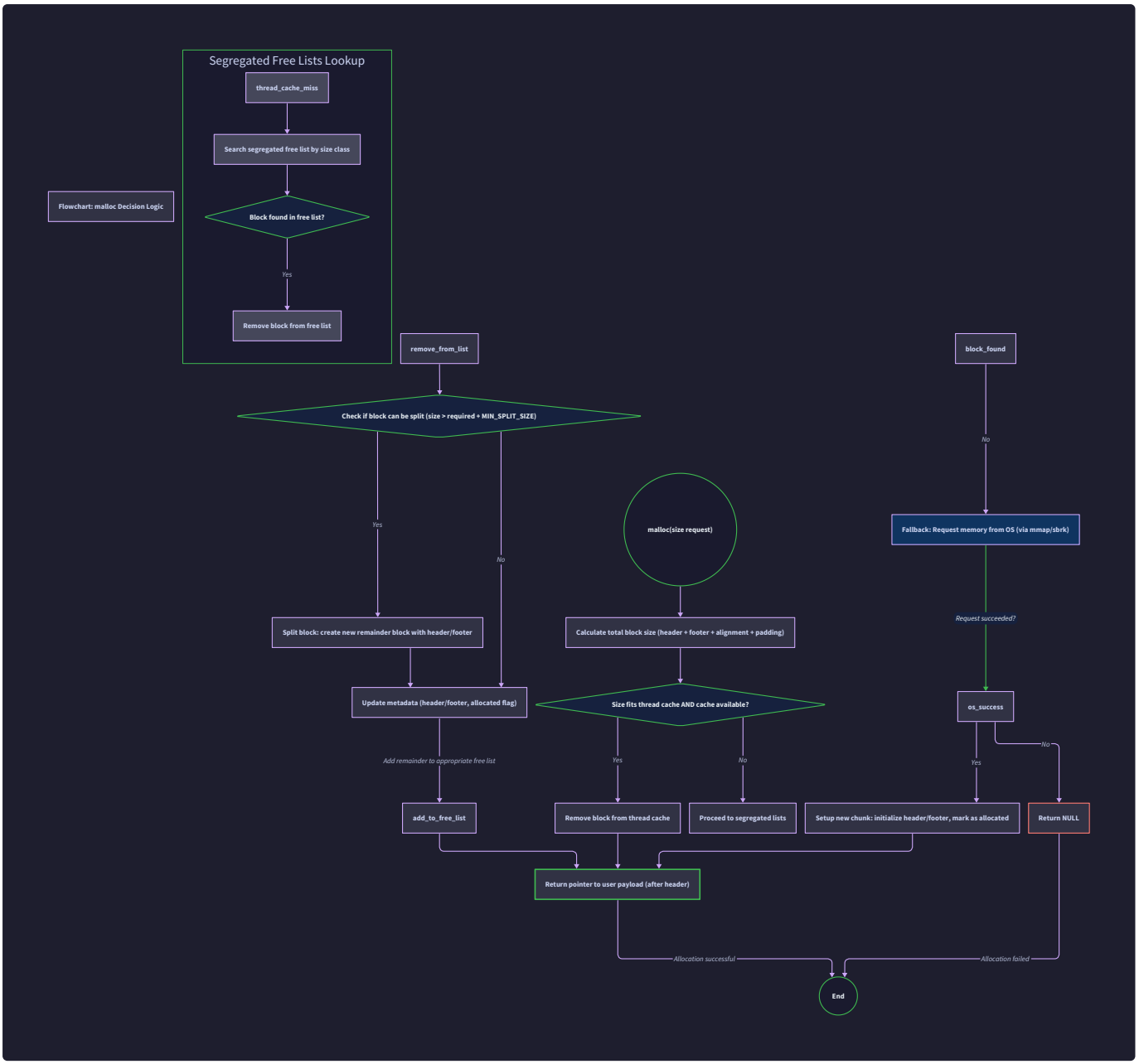
A memory block transitions between states based on allocator operations. The state machine below defines these transitions. The "WILDERNESS" is a special block at the very end of the heap, which can be expanded via `sbrk`.

Current State	Event	Next State	Actions Taken
FREE (in free list)	`malloc` finds this block (exact fit).	**ALLOCATED**	1. `free_list_remove(block)`. 2. Set `block->allocated = 1`. 3. Update footer's `allocated` flag.
FREE	`malloc` finds this block and `split_block` is called.	**ALLOCATED** (front part) & **FREE** (remainder)	1. `free_list_remove(block)`. 2. `split_block` creates new allocated header/footer for front. 3. `init_block` and `free_list_insert` for the new free remainder.
ALLOCATED	User calls `free(ptr)`.	**FREE**	1. `get_header_from_ptr(ptr)` to find block. 2. Set `block->allocated = 0`. Update footer. 3. `coalesce_block(block)`. 4. `free_list_insert` on the resulting (possibly merged) block.
WILDERNESS (at heap end)	`malloc` requests memory and no free block is found.	**ALLOCATED** (or **FREE** after split)	1. Call `sbrk` to expand the wilderness block. 2. Treat the expanded wilderness as a large **FREE** block, then proceed with normal allocation/split logic.
FREE (Block A) & **FREE** (Block B, adjacent)	`coalesce_block` is called on A or B.	**FREE** (single merged block)	1. `free_list_remove` for both A and B. 2. Calculate merged size. 3. Update header of first block and footer of last block. 4. `free_list_insert` for the new merged block.



Algorithm: The malloc() Walkthrough

The `malloc(size)` function is the primary interface for memory requests. Its internal logic is a multi-stage pipeline. The flowchart below gives a visual overview, and the numbered steps provide the detailed narrative.

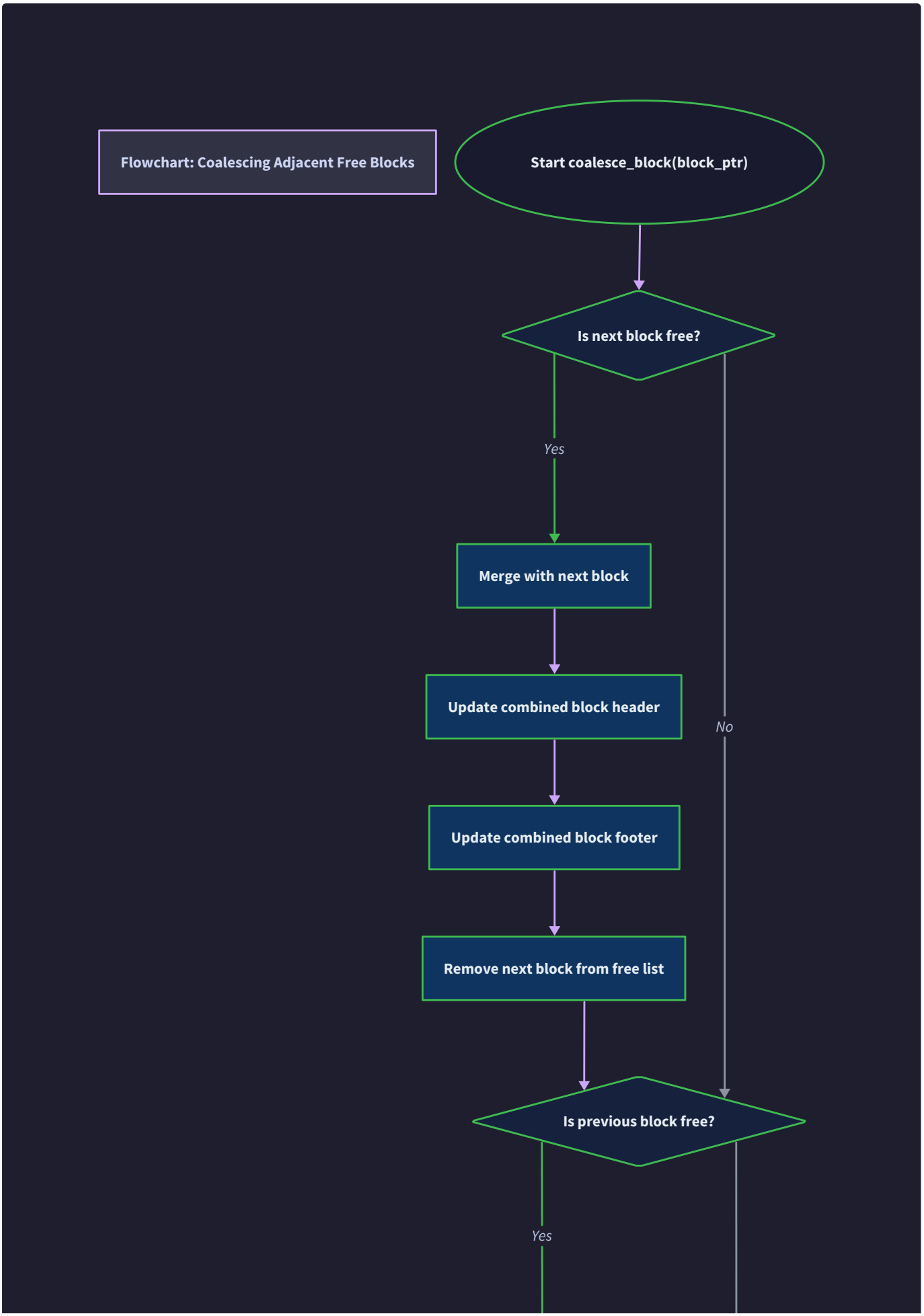


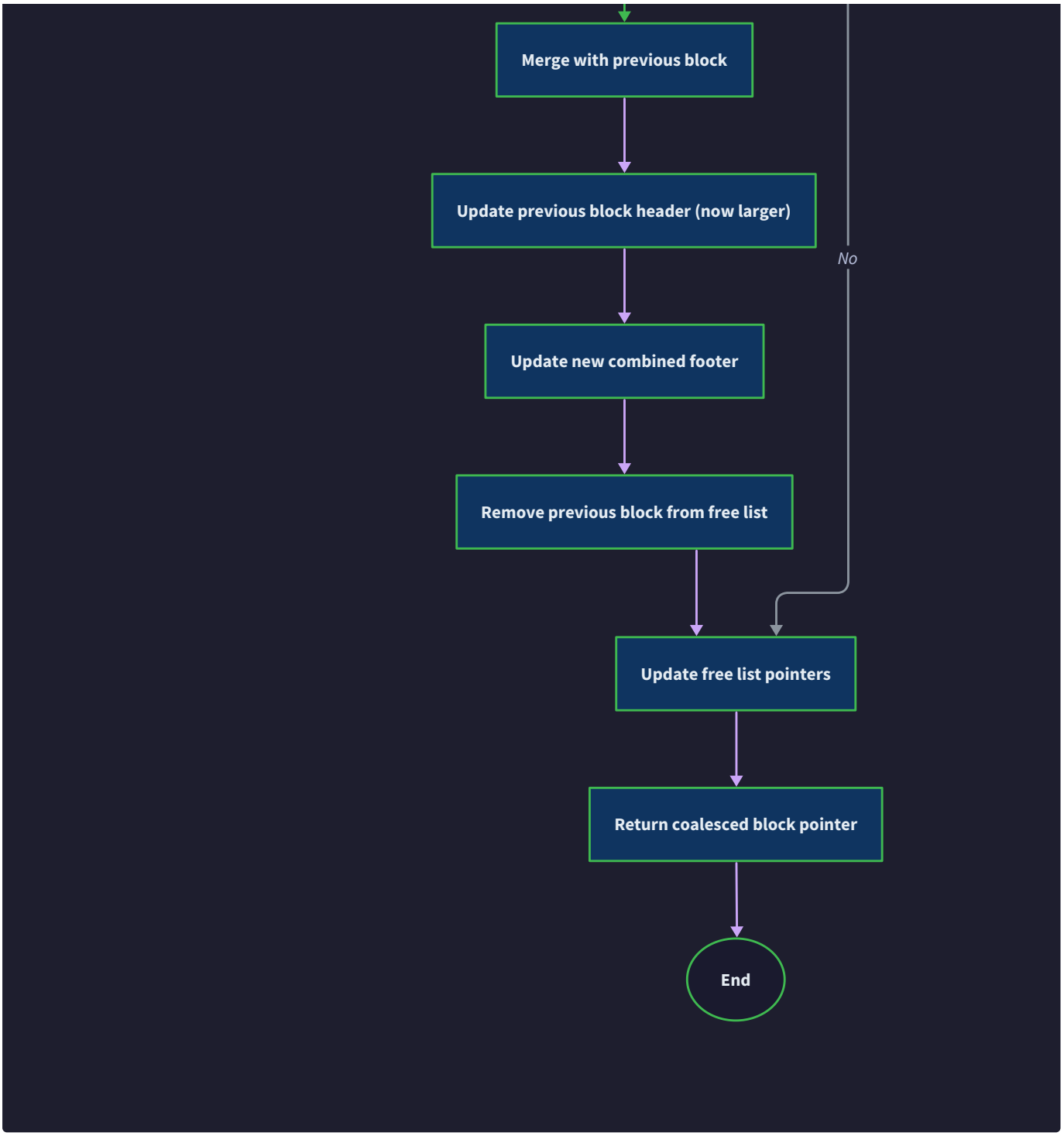
1. **Request Validation and Size Calculation**:
   - **Input**: The user requests `size` bytes.
   - **Edge Case**: If `size == 0`, the implementation may return `NULL` or a unique pointer that can be safely passed to `free`. We choose to return `NULL` for simplicity.
   - **Size Transformation**: Call `get_total_block_size(size)`. This function: a. Adds the size of `block_header_t` and `footer_t`. b. Rounds up the user payload size to meet the `ALIGNMENT` requirement (e.g., 8 bytes). c. Ensures the final total size is at least the **minimum block size** (large enough to hold a free block's header, footer, and `next` / `prev` pointers when freed).
   - **Output**: `total_size` – the actual number of bytes needed in the heap to satisfy this request.
2. **Free List Search**:
   - Call `find_free_block(total_size)`. This function traverses the explicit free list starting from `allocator_state_t.free_list_head`.
   - The **search strategy** governs the traversal:
     - **First-fit**: Selects the *first* block in the list with `block->size >= total_size`. This is fast but can lead to fragmentation at the front of the list.
     - **Best-fit**: Scans the *entire* list to find the block where `block->size - total_size` is minimal (positive). This reduces wasted space in the free block (external fragmentation) but is slower (O(n)) and can create many tiny, unusable leftover fragments ("splinters").

        - **Worst-fit**: Scans the entire list to find the *largest* block. The idea is to leave a large remainder, which may be more useful for future requests. It often performs poorly in practice.
    - If a suitable block is found, proceed to step 4. If not, proceed to step 3.
3. **Heap Expansion (OS Request)**:
    - No free block is large enough. The allocator must request more memory from the OS.
    - Call `get_memory_from_os(total_size, &is_large)`. This function (from Section 5.1) decides whether to use `sbrk` (for small requests) or `mmap` (for large requests > `SBRK_THRESHOLD`). It returns a pointer to the newly obtained raw memory.
    - For `sbrk` allocations, the new memory is contiguous with the existing heap. The allocator must treat this new region as a single, large **free block**. This typically involves expanding the "wilderness" block at the end of the heap or creating a new free block if the heap was exhausted.
    - For `mmap` allocations, the memory is disjoint. It is initialized as a single, standalone free block (or directly as an allocated block if the `IS_MMAP_FLAG` is used to skip coalescing). This block is then inserted into the free list (for `sbrk`-like handling) or processed directly.
    - After expansion, restart the free list search (step 2). This time, the new large free block will be found.
4. **Block Splitting (if necessary)**:
    - We now have a free block (`candidate`) with `candidate->size >= total_size`.
    - Check if the candidate block is **too large**. A common rule is to split if `candidate->size >= total_size + MIN_BLOCK_SIZE` (where `MIN_BLOCK_SIZE` is the smallest possible free block). If not, we use the entire block (an "exact fit").
    - If splitting is warranted: a. Call `split_block(candidate, total_size)`. b. This function calculates the address of the remainder block: `(char*)candidate + total_size`. c. It initializes the remainder as a new free block with size `candidate->size - total_size`. d. It updates the `candidate` header and footer to reflect its new, smaller `total_size` and marks it as allocated. e. It inserts the new remainder block into the free list via `free_list_insert`.
    - If not splitting, simply mark the `candidate` block as allocated.
5. **Finalization and Pointer Return**:
    - The `candidate` block (now possibly split) is allocated. Ensure its `allocated` flag is `1` in both header and footer.
    - Remove it from the free list if it hasn't been already (`split_block` handles removal of the original).
    - Calculate the **user payload pointer**. This is the address immediately after the block's header: `(void*)((char*)block_header + sizeof(block_header_t))`.
    - Return this pointer to the caller. The memory from this pointer to `pointer + size - 1` is now owned by the user program.

Algorithm: The free() Walkthrough

The `free(ptr)` function releases memory back to the allocator for reuse. Its correctness is crucial; errors here lead to heap corruption, crashes, or security vulnerabilities. The coalescing process is detailed in a separate flowchart.


Flowchart: Coalescing Adjacent Free Blocks
Start coalesce_block(block_ptr)
Is next block free?
Yes
No
Merge with next block
Update combined block header
Update combined block footer
Remove next block from free list
Is previous block free?
Yes



1. **Pointer Validation**:
   - **Input**: A pointer `ptr` previously returned by `malloc` or `calloc`.
   - **Edge Case**: If `ptr` is `NULL`, `free` must do nothing (this is specified by the C standard). Return immediately.
   - **Sanity Check**: Optionally, verify that `ptr` appears to be a valid heap pointer (lies within the known heap bounds) and is aligned. In a debugging build, you might also check a "magic number" in the header to detect corruption.
2. **Locate Block Metadata**:
   - Call `block_header_t *block = get_header_from_ptr(ptr)`.
   - This function performs the inverse of the pointer calculation in `malloc`: `block = (block_header_t*)((char*)ptr - sizeof(block_header_t))`.
   - At this point, `block` points to the header of the block being freed.
3. **Mark Block as Free**:

- Set `block->allocated = 0`.
- Update the corresponding **footer's** `allocated` flag. This is critical for the next block's coalescing logic to work. The footer's address is `(footer_t*)((char*)block + block->size - sizeof(footer_t))`.

4. **Coalesce with Adjacent Free Blocks**:
   - Call `coalesce_block(block)`. This function implements the following logic: a. **Coalesce with next block**: i. Calculate `next_block = (block_header_t*)((char*)block + block->size)`. ii. Check if `next_block` is within the heap bounds and if `next_block->allocated == 0`. iii. If yes, `free_list_remove(next_block)`. Update the current `block->size` to `block->size + next_block->size`. Update the *new* footer (now at the end of the merged block) to reflect the merged size and free status. b. **Coalesce with previous block**: i. Calculate the address of the *previous* block's footer: `prev_footer = (footer_t*)((char*)block - sizeof(footer_t))`. ii. From `prev_footer->size`, calculate the address of the previous block's header: `prev_block = (block_header_t*)((char*)block - prev_footer->size)`. iii. Check if `prev_block` is within heap bounds and if `prev_footer->allocated == 0`. iv. If yes, `free_list_remove(prev_block)`. Update `prev_block->size` to `prev_block->size + block->size`. Update the footer at the end of the merged block (which is the same footer we updated in step 'a', or the original block's footer). Set `block = prev_block` (the coalesced block now starts at the previous block).
   - The function returns a pointer to the header of the final, possibly merged, free block.

5. **Insert into Free List**:
   - Call `free_list_insert(block)` where `block` is the (possibly coalesced) free block.
   - Insertion at the **head** of the list is common: `block->next = allocator_state.free_list_head;` `block->prev = NULL;` If the old head existed, set `old_head->prev = block;`. Finally, update `allocator_state.free_list_head = block;`.
   - The free block is now available to satisfy future `malloc` requests.

> **Key Insight:** The order of coalescing (next then previous) is not arbitrary. Coalescing with the next block first enlarges the current block. This enlarged block's footer is now further forward in memory. When we then coalesce with the previous block, we use the *original* previous footer (which is unchanged) to find the previous block. This works correctly regardless of order, but this sequence is straightforward to implement.

Common Pitfalls

⚠️ Pitfall: Incorrect Pointer Arithmetic in `get_header_from_ptr`

- **Description**: Using `ptr - sizeof(block_header_t)` instead of `(char*)ptr - sizeof(block_header_t)`. Since pointer arithmetic in C scales by the size of the pointed-to type, this will subtract `sizeof(block_header_t) * sizeof(pointer_type)` bytes, resulting in a wildly incorrect header address.
- **Why it's wrong**: `malloc` returns a `void*`. When you do arithmetic on a `void*`, the compiler doesn't know the size to scale by. Casting to `char*` is essential because `sizeof(char)` is 1, making arithmetic work in raw bytes.
- **Fix**: Always cast the user pointer to a byte-sized pointer type (`char*` or `uint8_t*`) before performing arithmetic.

⚠️ Pitfall: Forgetting to Update the Footer During `split_block`

- **Description**: After splitting a block, you update the header of the new allocated block and the header of the new free remainder, but you forget to create/update the footer for the new allocated block.
- **Why it's wrong**: The allocated block's footer is used by the *next* block's `coalesce` operation to look backwards. If it contains garbage (old size from the original large block), the next `free` will calculate the wrong address for the previous block, leading to heap corruption.
- **Fix**: Every time a block's size or allocation status changes, update **both** its header **and** its footer. Write a helper function `write_block_metadata(block_header_t* block, size_t size, int allocated)` that does both.

⚠️ Pitfall: Not Handling the `MIN_BLOCK_SIZE` Constraint in `split_block`

- **Description**: Splitting a block without checking if the remainder would be large enough to form a valid free block (able to hold header, footer, and free list pointers).
- **Why it's wrong**: If the remainder is smaller than `MIN_BLOCK_SIZE`, you cannot form a proper free block from it. Attempting to insert it into the free list will corrupt metadata (the `next`/`prev` pointers would overlap with the footer or trail into the next block). This creates an unusable "fragment" of memory—wasted space.
- **Fix**: Before splitting, check if `block->size - needed_size >= MIN_BLOCK_SIZE`. If not, do not split; use the entire block even if it's larger than requested (this causes **internal fragmentation** but is necessary for correctness).

⚠️ Pitfall: Corrupting the Free List During Coalescing

- **Description**: When merging block A with block B, you correctly update sizes but forget to remove block B from the free list before its memory is absorbed into A.
- **Why it's wrong**: Block B's `next` and `prev` pointers are now part of the payload of the merged block A. If block B is still in the free list, future list traversal will follow these pointers, which point into the middle of an allocated (or newly freed) block, leading to crashes or silent data corruption.
- **Fix**: The sequence must be: 1) `free_list_remove(next_block)`, 2) update size of current block, 3) update footer. Always remove a block from the free list *before* its identity is lost to merging.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (For Milestone 1-2)	Advanced Option (For Optimization)
Free List Search	Linear scan of an explicit doubly-linked list. Easy to debug and implement first-fit/best-fit/worst-fit.	Segregated fits (Milestone 3) or even a balanced binary tree of free blocks for O(log n) best-fit (complex).
Coalescing	Immediate coalescing on every `free`. This keeps fragmentation low but adds a small constant-time overhead to `free`.	Deferred coalescing: mark blocks free but only coalesce during allocation when needed. Can improve `free` performance but increases fragmentation.
Block Lookup	Store block size with LSB used as allocated flag (saving 8 bytes per block). Use macros to extract size/flag.	Keep `size` and `allocated` separate for clarity during learning. Add a `MAGIC` number field for debugging corruption.

B. Recommended File/Module Structure

Place the block management core logic in its own file, separate from the OS interface and high-level strategy.

```
allocator/
├── include/
│   └── allocator.h          # Public API (malloc, free) and core types (block_header_t)
├── src/
│   ├── os_mem.c             # OS Memory Interface (from Section 5.1)
│   ├── heap.c               # THIS FILE: Core block management (malloc, free, coalesce, split)
│   ├── free_list.c          # Explicit free list operations (insert, remove, find)
│   ├── strategies.c         # Allocation strategies (first_fit, best_fit, worst_fit)
│   └── allocator.c          # Top-level allocator state initialization and public API wrappers
```

C. Infrastructure Starter Code (Helper Macros and Functions)

These are complete, ready-to-use helpers for safely accessing block metadata.

```c
/* In include/allocator.h or a private header like heap_private.h */
#include <stddef.h> /* for size_t */
#include <stdint.h>

/* Constants */
#define ALIGNMENT 8
#define MIN_BLOCK_SIZE (sizeof(block_header_t) + sizeof(footer_t) + ALIGNMENT)

/* Macro to align a size 's' up to the next multiple of ALIGNMENT */
#define ALIGN_UP(s) (((s) + (ALIGNMENT - 1)) & ~(ALIGNMENT - 1))

/* Macro to get the footer pointer from a block header pointer */
#define GET_FOOTER_FROM_HEADER(b) ((footer_t*)((char*)(b) + (b)->size - sizeof(footer_t)))

/* Macro to get the header pointer from a footer pointer */
#define GET_HEADER_FROM_FOOTER(f) ((block_header_t*)((char*)(f) - (f)->size + sizeof(footer_t)))

/* Macro to check if a block is free (using a separate allocated field) */
#define IS_ALLOCATED(b) ((b)->allocated)

/* In src/heap.c */
#include "allocator.h"
#include <assert.h>

/* Helper: Calculate total block size needed for a user request */
size_t get_total_block_size(size_t user_size) {
    if (user_size == 0) {
        return 0;
    }
    /* User payload must be aligned, so we first add header size, then align the payload start */
    size_t total = sizeof(block_header_t) + user_size + sizeof(footer_t);
    /* The payload (after header) must be aligned. Let's calculate the start of the payload. */
    uintptr_t payload_start = (uintptr_t)NULL + sizeof(block_header_t);
    uintptr_t aligned_payload_start = ALIGN_UP(payload_start);
    size_t header_padding = aligned_payload_start - payload_start;
    total += header_padding;
    /* Ensure the total size itself is a multiple of ALIGNMENT for future blocks */
    total = ALIGN_UP(total);
    /* Ensure it meets the minimum block size */
    if (total < MIN_BLOCK_SIZE) {
```

```
        total = MIN_BLOCK_SIZE;
    }
    return total;
}

/* Helper: Given a user pointer, return its block header. ASSUMES valid pointer. */
block_header_t* get_header_from_ptr(void *ptr) {
    if (ptr == NULL) {
        return NULL;
    }
    /* The header is located exactly sizeof(block_header_t) bytes before the user pointer */
    return (block_header_t*)((char*)ptr - sizeof(block_header_t));
}

/* Helper: Initialize a raw memory region as a block with given user size and allocated flag */
void init_block(void *memory, size_t user_size, int allocated) {
    if (memory == NULL || user_size == 0) {
        return;
    }
    size_t total_size = get_total_block_size(user_size);
    block_header_t *header = (block_header_t*)memory;
    footer_t *footer = GET_FOOTER_FROM_HEADER(header);

    header->size = total_size;
    header->allocated = allocated;
    /* For free blocks, next/prev are initialized when inserted into free list */
    if (!allocated) {
        header->next = NULL;
        header->prev = NULL;
    }

    footer->size = total_size;
    footer->allocated = allocated;
}
```

D. Core Logic Skeleton Code

The core functions that learners must implement, with TODOs mapping to the algorithm steps.

```c
/* In src/heap.c */

/**
 * Splits a free block into an allocated part of size 'needed_size' (total size)
 * and a free remainder, if the remainder is large enough to form a valid block.
 * Returns a pointer to the header of the allocated part (the original block if no split).
 */
block_header_t* split_block(block_header_t *block, size_t needed_size) {
    // TODO 1: Check if the block is free and large enough. If not, return NULL.
    // TODO 2: Calculate the size of the potential remainder: block->size - needed_size.
    // TODO 3: If the remainder size is less than MIN_BLOCK_SIZE, do not split.
    //         Return the original block (caller will mark it allocated).
    // TODO 4: Otherwise, perform the split:
    //    a. Calculate the address of the new free block: (char*)block + needed_size.
    //    b. Call init_block() on that address with a user size of (remainder_size - metadata).
    //       This will set up the header and footer for the new free block.
    //    c. Update the original block's size to needed_size and mark it allocated.
    //       Update its footer as well (use GET_FOOTER_FROM_HEADER).
    //    d. The new free block should be inserted into the free list (free_list_insert).
    // TODO 5: Return a pointer to the header of the allocated (split) block.
    return NULL; // placeholder
}

/**
 * Coalesces the given free block with its immediate neighbors if they are free.
 * Returns a pointer to the header of the coalesced block.
 */
block_header_t* coalesce_block(block_header_t *block) {
    // TODO 1: Ensure the block is free. If not, return it unchanged.
    // TODO 2: Coalesce with next block (if it exists and is free):
    //    a. Calculate address of next block: (char*)block + block->size.
    //    b. Check that next_block is within heap bounds (you need a heap limit variable).
    //    c. If next_block is free (check its header's allocated flag):
    //       i. Remove next_block from the free list (free_list_remove).
    //       ii. Add next_block->size to block->size.
    //       iii. Update the footer of the now-larger block (the footer is now at the end of next_block).
    // TODO 3: Coalesce with previous block (if it exists and is free):
```

```
    //    a. Calculate address of previous block's footer: (char*)block - sizeof(footer_t).
    //    b. From that footer, get the size of the previous block and calculate its header address.
    //    c. Check that prev_block is within heap bounds.
    //    d. If prev_block is free:
    //        i. Remove prev_block from the free list (free_list_remove).
    //        ii. Remove the current block from the free list (it's about to be merged).
    //        iii. Add block->size to prev_block->size.
    //        iv. Update the footer of the now-larger prev_block (the footer is the one from step 2-iii or block's original footer).
    //        v. Set block = prev_block (the coalesced block now starts earlier).
    // TODO 4: Return the (potentially new) block pointer.
    return block; // placeholder
}

/* In src/free_list.c */
#include "allocator.h"

/* Global allocator state (declared elsewhere, perhaps in allocator.c) */
extern allocator_state_t allocator_state;

/**
 * Finds a free block of at least the given total size using the configured strategy.
 */
block_header_t* find_free_block(size_t size) {
    block_header_t *current = allocator_state.free_list_head;
    block_header_t *best = NULL;
    block_header_t *worst = NULL;

    // TODO 1: Based on a global config variable (e.g., allocator_state.strategy):
    //    - If FIRST_FIT: traverse list, return first block with current->size >= size.
    //    - If BEST_FIT: traverse entire list, track the block with the smallest adequate size.
    //    - If WORST_FIT: traverse entire list, track the block with the largest size.
    // TODO 2: For BEST_FIT and WORST_FIT, you must handle the case where no block is found.
    // TODO 3: Return the selected block, or NULL if none found.
    return NULL; // placeholder
}

/**
 * Inserts a free block at the head of the explicit free list.
```

```
 */
void free_list_insert(block_header_t* block) {
    // TODO 1: Check that block is not NULL and is free (assert or check).
    // TODO 2: Set block->next to the current free_list_head.
    // TODO 3: Set block->prev to NULL.
    // TODO 4: If free_list_head is not NULL, set free_list_head->prev = block.
    // TODO 5: Update allocator_state.free_list_head = block.
}

/**
 * Removes a free block from the explicit free list.
 */
void free_list_remove(block_header_t* block) {
    // TODO 1: Check that block is not NULL and is free.
    // TODO 2: Handle the case where block is the head of the list:
    //         If block->prev == NULL, set allocator_state.free_list_head = block->next.
    // TODO 3: If block->prev is not NULL, set block->prev->next = block->next.
    // TODO 4: If block->next is not NULL, set block->next->prev = block->prev.
    // TODO 5: For safety, set block->next = block->prev = NULL (optional).
}
```

E. Language-Specific Hints (C)

- **Pointer Casting**: Use `(char*)` for byte-level pointer arithmetic. `void*` arithmetic is not standard in C (though GCC allows it as an extension). Be explicit.
- **Structure Packing**: The `block_header_t` and `footer_t` structures may have padding added by the compiler. Use `#pragma pack(push, 1)` and `#pragma pack(pop)` if you need exact byte layout control (important for reading/writing headers to disk, but less critical for in-memory allocators where you always access via the struct).
- **Bit Manipulation**: To store the `allocated` flag in the LSB of `size`, use: `size_t real_size = size & ~1;` and `int allocated = size & 1;`. Ensure you always clear the LSB when storing sizes.
- **Assertions**: Use `assert(block != NULL)` liberally in your helper functions during development to catch logic errors early. These can be compiled out in release builds.

F. Milestone Checkpoint After implementing the block management core (Milestone 2), you should be able to run a test that demonstrates splitting and coalescing.

```c
// test_coalesce.c

#include "allocator.h"

#include <stdio.h>

#include <assert.h>


int main() {

    // Initialize your allocator (not shown)



    void *p1 = malloc(100);

    void *p2 = malloc(100);

    void *p3 = malloc(100);



    printf("Allocated three 100-byte blocks at %p, %p, %p\n", p1, p2, p3);



    free(p2);

    printf("Freed middle block. Heap should have a free block between two allocated.\n");



    // Now free the first block - it should coalesce with the middle free block

    free(p1);

    printf("Freed first block. Should coalesce with middle free block.\n");



    // Allocate a 200-byte block - it should fit in the coalesced free space

    void *p4 = malloc(200);

    printf("Allocated 200-byte block at %p. Should reuse coalesced space.\n", p4);



    assert(p4 != NULL);

    // If coalescing worked, p4 should be where p1 was (or nearby)



    free(p3);

    free(p4);

    printf("All blocks freed. No memory leaks (check with valgrind).\n");



    return 0;

}
```

- **Expected Behavior**: The program runs without crashing. The 200-byte allocation succeeds, indicating that the 100-byte block (p1) and the adjacent 100-byte free block (from p2) were merged into a 200+ byte free block.
- **Check with Visualization**: Implement a simple `heap_visualize()` function that walks the heap from start to end, printing each block's address, size, and status. Run it after each operation in the test to see the splitting and coalescing in action.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault in `free()` when trying to access block header.	`get_header_from_ptr` computed wrong address due to incorrect pointer arithmetic.	Print the `ptr` value and the computed `header` value. Ensure the difference is exactly `sizeof(block_header_t)`.	Cast `ptr` to `(char*)` before subtraction.
`malloc` returns memory that is not 8-byte aligned.	`get_total_block_size` does not align the payload start correctly.	For a small allocation (e.g., 1 byte), print the returned pointer and check `((uintptr_t)ptr % 8) == 0`.	Ensure you align the *payload* start, not just the total size. The header size may not be a multiple of 8.
Heap corruption: later `malloc` returns a pointer that overlaps a previous allocation.	**Coalescing logic is broken**, merging an allocated block with a free one.	Add a `MAGIC` number to each block header. Check it in `coalesce_block`. Visualize the heap before and after the erroneous `free`.	In `coalesce_block`, double-check the condition for checking if next/prev block is free. Use the footer's `allocated` flag, not a guess.
Memory leak: repeated `malloc` / `free` cycle slowly increases heap size.	Freed blocks are not being re-used. The free list is not updated correctly.	After `free`, print the free list to see if the block was inserted. Check that `free_list_remove` is called when a block is allocated.	Ensure `free_list_insert` is called in `free` after coalescing. Ensure `free_list_remove` is called in `malloc` (or in `split_block`) before returning a block.
Splitting creates a corrupted free block (crashes on later free).	The remainder block's metadata (footer) is not initialized or is incorrect.	Use `heap_visualize()` immediately after the split. Check the footer of the new free block for correct size and free flag.	In `split_block`, call `init_block` on the remainder region with `allocated=0`. This will set both header and footer correctly.

5.3 Allocation Strategy Component

> **Milestone(s):** Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists)

The Allocation Strategy Component is the **decision-making brain** of the memory allocator. While the Block Management Component (Section 5.2) provides the fundamental operations for manipulating individual blocks, this component determines *how* and *where* to find free memory when a `malloc` request arrives. It encapsulates the search algorithms that traverse data structures to locate suitable blocks and the policy decisions that balance performance against fragmentation.

Think of this component as the warehouse manager's **picking strategy manual**. When a worker requests a shelf of specific dimensions, the manager must decide: Do we take the first shelf we find that fits? Do we search for the shelf that leaves the smallest leftover space? Or do we organize shelves by size category so we can go directly to the right aisle? These trade-offs between search speed, space utilization, and organizational overhead are precisely what this component manages.

Mental Model: The Shelf-Finding Strategy

Imagine our heap as a massive warehouse with shelves (blocks) of various sizes scattered throughout. Some shelves are occupied with inventory (allocated blocks), while others sit empty (free blocks). When a customer (your program) orders `n` boxes of merchandise (bytes of memory), the warehouse manager (allocator) must:

1. **Find an empty shelf** large enough to hold `n` boxes
2. **Remove it from the inventory of empty shelves** (so it won't be given to another customer)
3. **Potentially split the shelf** if it's larger than needed, returning the leftover portion to the empty shelf inventory
4. **Label the shelf** with shipping information (the block header) so we know what's stored there and who owns it

Different managers have different strategies for step 1, each with distinct performance characteristics:

Strategy	Warehouse Analogy	Performance	Fragmentation Impact
First-Fit	Walk the aisles from the entrance and take the *first* empty shelf that fits your order.	Fast for small free lists; degrades to O(n) as list grows.	Tends to leave small fragments at the beginning of the list ("external fragmentation").
Best-Fit	Measure every empty shelf in the warehouse and choose the one that leaves the *smallest leftover space* after your order.	Must examine all free blocks → O(n) always.	Minimizes leftover fragments but creates many tiny, often unusable blocks ("internal fragmentation").
Worst-Fit	Find the *largest* empty shelf in the warehouse and use it, hoping the large leftover remains useful.	Must examine all free blocks → O(n) always.	Tends to keep leftover fragments large but can waste space by using oversized blocks.
Next-Fit	Start searching from where you left off last time (maintain a "roving pointer") instead of always starting at the beginning.	More even distribution of searches than first-fit; still O(n).	Similar fragmentation to first-fit but spreads allocation attempts across the heap.

> **Key Insight:** No single strategy dominates in all scenarios. First-fit works well when allocations are similarly sized, best-fit maximizes space utilization but slows allocations, and worst-fit attempts to preserve large contiguous regions. Our **teaching allocator** will implement all three to allow experimentation and comparison.

The mental model extends when we introduce **segregated free lists**. Imagine instead of one giant warehouse with mixed-size shelves, we create separate **specialized storage rooms** for different shelf sizes: a room for 16-box shelves, another for 32-box shelves, etc. When an order arrives, we first check if it fits a standard size category. If so, we go directly to that room and take the first shelf available—no searching through unrelated shelves needed. This is the fundamental optimization behind segregated lists.

ADR: Segregated Free Lists for Common Sizes

> **Decision:** Use multiple segregated free lists binned by power-of-two size classes for allocations under 256 bytes, with a fallback to a general free list for larger requests.

- **Context:** The single explicit free list from Milestone 2 requires O(n) scanning time, where n is the number of free blocks. Real-world allocation patterns show that small allocations (under 256 bytes) dominate typical programs, yet these small requests suffer the most from linear search overhead. We need to improve allocation speed for common cases while maintaining the flexibility to handle arbitrary-sized requests.
- **Options Considered:**

Option	Pros	Cons	Evaluation
Single explicit free list (current Milestone 2)	Simple implementation, minimal metadata overhead, handles all sizes uniformly.	O(n) search time degrades with heap fragmentation; poor cache locality for small allocations.	Unacceptable performance for teaching modern allocator concepts.
Buddy allocator (power-of-two blocks only)	Extremely fast allocation/deallocation via bitmaps; excellent for block coalescing.	Severe internal fragmentation (up to 100% waste); only handles power-of-two sizes; complex splitting/merging logic.	Too specialized; doesn't teach general-purpose allocation strategies.
Segregated fits with exact size classes	Perfect fit for common sizes → minimal internal fragmentation.	Many size classes needed; complex management; poor utilization for sizes between classes.	Management overhead outweighs benefits for a teaching allocator.
Segregated fits with power-of-two size classes	Fast O(1) lookup via size-to-class mapping; simple implementation; covers common allocation ranges.	Internal fragmentation up to 100% (e.g., 65-byte request gets 128-byte block). Acceptable trade-off for teaching.	**Selected** — balances performance gains with implementation simplicity.
Segregated fits with geometric progression (e.g., Fibonacci)	Better size fit than power-of-two, less internal fragmentation.	More complex size-class computation; still requires fallback for large sizes.	Overly complex for educational purposes.

- **Decision:** Implement segregated free lists for eight power-of-two size classes (16, 32, 64, 128, 256, 512, 1024, 2048 bytes), maintaining a separate explicit free list for each class. Allocations smaller than or equal to 2048 bytes are rounded up to the nearest size class and served from the corresponding list. Larger allocations use the general free list (which itself can use first-fit, best-fit, or worst-fit).

- **Rationale:**
    1. **Educational value:** Demonstrates a real-world optimization used by production allocators (like jemalloc and tcmalloc) while keeping implementation manageable.
    2. **Performance leap:** Converts O(n) search to O(1) lookup for small allocations, which comprise the majority of requests in typical programs.
    3. **Progressive enhancement:** The design allows learners to implement basic free lists first (Milestone 2), then add segregation on top (Milestone 3) without rewriting core logic.
    4. **Configurable trade-off:** The number of size classes and their boundaries can be adjusted via compile-time constants, allowing experimentation with the space-time trade-off.
- **Consequences:**
    - **Internal fragmentation increases:** A 33-byte request receives a 64-byte block, wasting 31 bytes (48% overhead). This is the price for O(1) allocation.
    - **Memory overhead grows:** Each size class requires its own free list head pointer (8 bytes each in our 64-bit system).
    - **Implementation complexity rises:** Need to maintain multiple free lists, map sizes to classes, and handle block promotion/demotion between classes during splitting and coalescing.
    - **Fallback path required:** Large allocations still need the general free list search, creating two code paths to maintain.

The following table defines our size class configuration:

Size Class Index	Maximum Allocation Size	Typical Request Range	Internal Fragmentation (Worst Case)
0	16 bytes	1–16 bytes	15/16 = 94%
1	32 bytes	17–32 bytes	16/32 = 50%
2	64 bytes	33–64 bytes	31/64 = 48%
3	128 bytes	65–128 bytes	63/128 = 49%
4	256 bytes	129–256 bytes	127/256 = 50%
5	512 bytes	257–512 bytes	255/512 = 50%
6	1024 bytes	513–1024 bytes	511/1024 = 50%
7	2048 bytes	1025–2048 bytes	1023/2048 = 50%
(General free list)	Unlimited	>2048 bytes	Depends on fit strategy

> **Design Principle:** The 50% worst-case internal fragmentation for power-of-two classes is acceptable because (1) small allocations dominate program memory usage, and (2) the alternative—O(n) searches—would slow down *every* allocation, not just waste some space.

Component Responsibilities and Interface

The Allocation Strategy Component has two primary responsibilities:

1. **Free List Search Strategy:** Implement the algorithms for first-fit, best-fit, and worst-fit searching of the general free list.
2. **Size Class Management:** Maintain segregated free lists, map request sizes to appropriate classes, and handle allocations from the correct list.

The component exposes these key functions to the rest of the allocator:

Function	Parameters	Returns	Description
`find_free_block`	`size_t requested_size`	`block_header_t*` (or NULL)	Searches the general free list using the configured strategy (first/best/worst-fit) for a block of at least `requested_size` bytes.
`malloc_segregated`	`size_t requested_size`	`void*` (user pointer or NULL)	Top-level allocation function for Milestone 3: uses segregated lists for small allocations, falls back to general search for large ones.
`size_to_class_index`	`size_t size`	`int` (0–7 or -1)	Maps a request size to the appropriate size class index; returns -1 if size exceeds largest class.
`get_size_class_head`	`int class_index`	`block_header_t**`	Returns pointer to the free list head pointer for the given size class.

The component maintains internal state through the global `allocator_state_t` structure, which we extend with size class information:

Field Name	Type	Description
`size_class_heads`	`block_header_t*[NUM_SIZE_CLASSES]`	Array of free list head pointers, one for each size class.
`allocation_strategy`	`enum { FIRST_FIT, BEST_FIT, WORST_FIT }`	Currently active search strategy for the general free list.
`next_fit_start`	`block_header_t*`	"Roving pointer" for next-fit strategy; points to where the next search should begin.

Algorithm: Free List Search Strategies

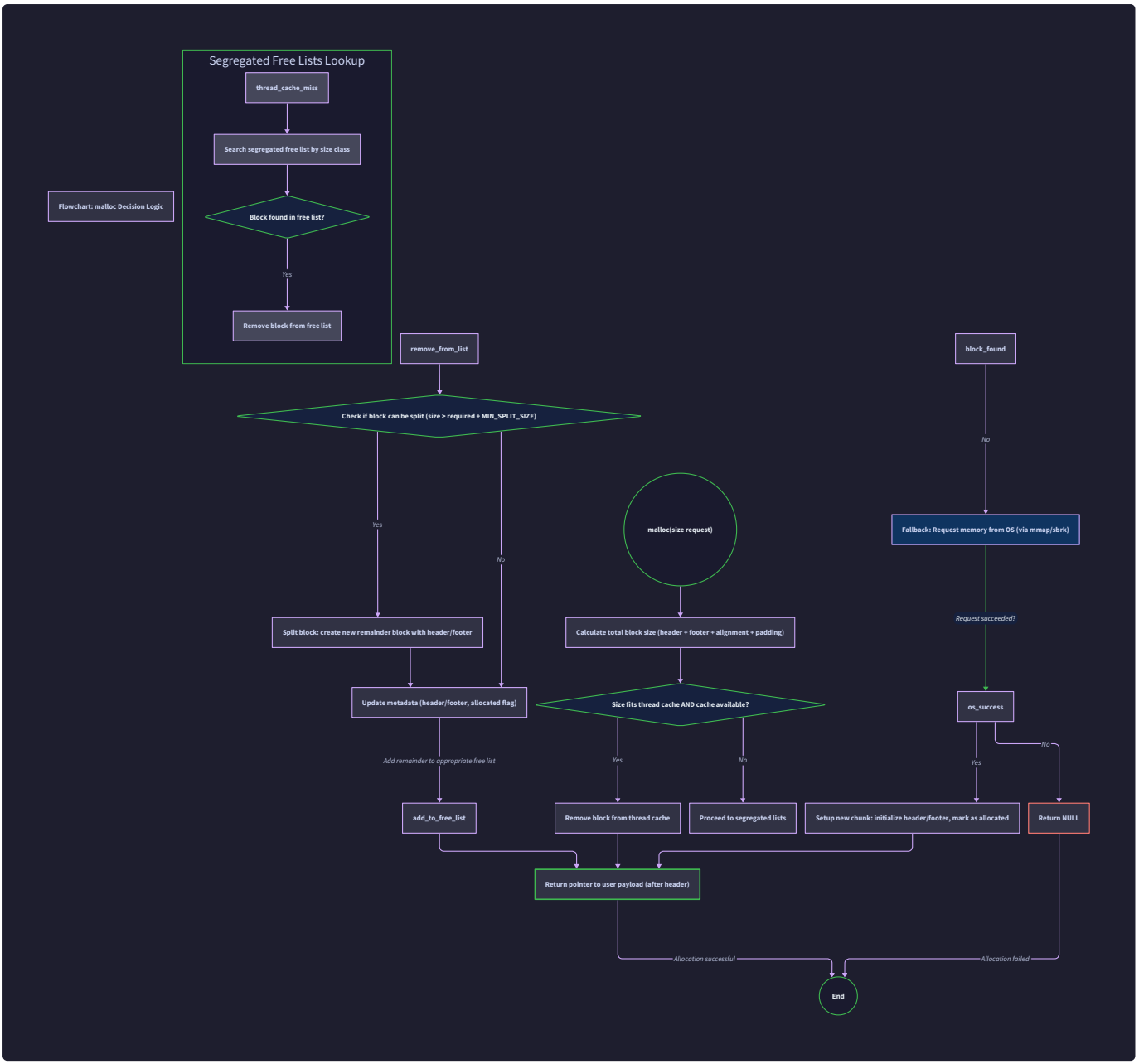
All search strategies share the same basic loop structure but differ in their selection criteria. Here's the unified procedure:

1. **Initialize search:**
   - If strategy is `NEXT_FIT`, start from `next_fit_start` (or `free_list_head` if NULL).
   - Otherwise, start from `free_list_head`.
2. **Traverse free list:**
   - For each free block `current` in the list: a. Calculate `block_size_available = current->size - HEADER_SIZE - FOOTER_SIZE` b. If `block_size_available >= requested_size`:
     - **First-fit:** Select this block immediately, update `next_fit_start` to `current->next` if using next-fit.
     - **Best-fit:** Track the smallest adequate block found so far; continue searching entire list.
     - **Worst-fit:** Track the largest block found so far; continue searching entire list.
   - Terminate traversal when list is exhausted or (for first-fit) when a suitable block is found.
3. **Select final block:**
   - **Best-fit:** Return the block with the smallest adequate size.
   - **Worst-fit:** Return the block with the largest size.
   - If no adequate block found, return NULL.
4. **Update search state:**
   - For next-fit, store the block *following* the selected block (or list head if none) as the new `next_fit_start`.

> **Performance Note:** Best-fit and worst-fit must always traverse the entire free list to guarantee optimal selection, making them O(n) in all cases. First-fit can stop early, giving amortized O(1) behavior when suitable blocks are near the list head. Next-fit attempts to distribute searches evenly, preventing small blocks from clustering at the list head.

Algorithm: Segregated List Allocation Walkthrough

When `malloc_segregated` is called with a request size, it follows this decision flow (refer to



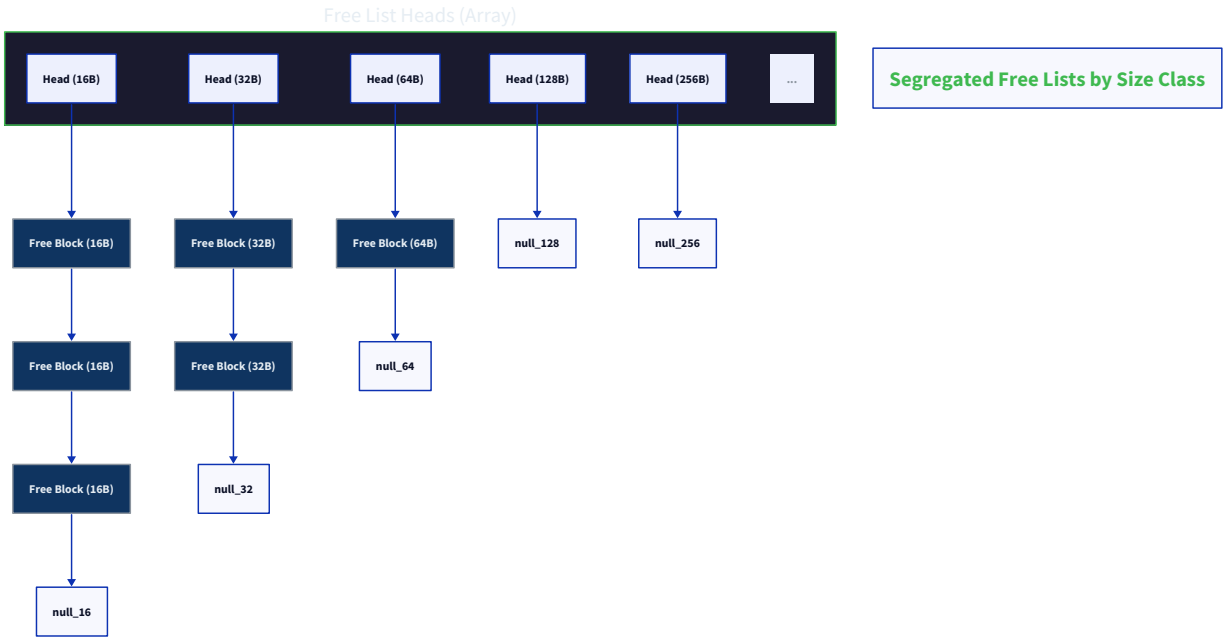
):

1. **Size validation:** If `requested_size == 0`, return NULL (or a unique pointer per implementation policy).
2. **Calculate total needed size:**
   - `total_needed = HEADER_SIZE + FOOTER_SIZE + requested_size + ALIGNMENT_PADDING`
   - This ensures enough space for metadata and alignment.
3. **Check for large allocation:**
   - If `total_needed > LARGEST_SIZE_CLASS` (2048 bytes in our design):
     - Fall back to general free list search via `find_free_block(total_needed)`
     - If found, split if appropriate, mark allocated, return user pointer.
     - If not found, request more memory from OS via `get_memory_from_os`.
4. **Map to size class:**
   - `class_idx = size_to_class_index(total_needed)`
   - Example: 33-byte request → total_needed ~= 65 bytes → class index 2 (64-byte class)
5. **Check segregated free list:**

- Retrieve `free_list_head = size_class_heads[class_idx]`
- If list is not empty: a. Remove first block from list via `free_list_remove(head)` b. Block is already correct size (exactly size class maximum), so no splitting needed c. Mark block as allocated, set appropriate size in header d. Return user pointer (header + HEADER_SIZE)

6. **Fallback to general search:**
   - If segregated list was empty: a. Search general free list for a block of at least `SIZE_CLASS_MAX[class_idx]` b. If found:
     - Split block: allocated portion = exact size class, remainder becomes new free block
     - Insert remainder into appropriate size class list based on its size
     - Mark allocated portion as in-use, return user pointer c. If not found:
     - Request memory from OS (size = max(SIZE_CLASS_MAX[class_idx], PAGE_SIZE))
     - Initialize as free block, split if larger than needed
     - Insert remainder into appropriate size class list
     - Return allocated portion
7. **Update statistics:** (Optional) Increment counters for monitoring allocation patterns.

The visual representation of segregated lists shows this architecture clearly:



Block Lifecycle with Size Classes

When blocks move between allocation states and size classes, several transitions occur:

Event	Block State Change	Size Class Action
`malloc_segregated(64)`	FREE → ALLOCATED	Removed from size class 2 free list (if present there)
`free(ptr)` on 64-byte block	ALLOCATED → FREE	Added to size class 2 free list (not general list)
Split 256-byte free block to satisfy 64-byte request	FREE → ALLOCATED (64B) + FREE (192B)	Remove from class 4; allocate 64B portion; add 192B remainder to class 3 (128B class)
Coalesce two adjacent 64-byte free blocks	FREE + FREE → FREE (128B)	Remove both from class 2; add merged block to class 3
Coalesce 64B + 128B free blocks	FREE + FREE → FREE (192B)	Remove from classes 2 and 3; add merged block to class 4 (if 192B ≤ 256B) or general list

> **Critical Detail:** During coalescing, the merged block may move to a different size class if its new size crosses a class boundary. This requires removing the original blocks from their respective class lists and inserting the merged block into the appropriate list.

Common Pitfalls

⚠️ Pitfall: Forgetting to update size class lists after splitting

- **Description:** When splitting a block from the general free list to satisfy a size class request, the remainder block must be inserted into the correct size class list (not left in the general list).
- **Why it's wrong:** The remainder block is now sized to fit a specific class, but if left in the general list, subsequent allocations for that size class won't find it, defeating the purpose of segregation.
- **Fix:** After `split_block()`, calculate the size class of the remainder and call `free_list_insert()` with the appropriate class head.

⚠️ Pitfall: Incorrect size-to-class mapping due to metadata overhead

- **Description:** Mapping user request size directly to size classes without accounting for header/footer overhead.
- **Why it's wrong:** A 64-byte size class must accommodate `HEADER_SIZE + FOOTER_SIZE + user_payload`, not just user_payload. If you reserve exactly 64 bytes total, the user gets less than expected.
- **Fix:** Size classes define the **total block size** (including metadata). The mapping function should compare `total_needed` (including metadata) against class boundaries, not `requested_size`.

⚠️ Pitfall: Best-fit degenerating to first-fit

- **Description:** Implementing best-fit by tracking the minimum adequate size, but stopping search early when finding an exact fit.
- **Why it's wrong:** There might be a smaller adequate block later in the list that's a better fit. True best-fit must examine all blocks.
- **Fix:** Traverse entire free list regardless of early exact matches. Use separate variables to track `best_block` and `best_waste` (remaining space).

⚠️ Pitfall: Next-fit pointer invalidation after coalescing

- **Description:** The `next_fit_start` pointer becomes dangling if the block it points to is coalesced with neighbors and removed from the free list.
- **Why it's wrong:** Subsequent allocations will start searching from an invalid pointer, potentially causing crashes or infinite loops.
- **Fix:** After any coalescing operation, check if `next_fit_start` points to any of the blocks involved. If so, reset it to the new merged block (if still free) or to the free list head.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Strategy Selection	Compile-time `#define` (e.g., `-DALLOC_STRATEGY=FIRST_FIT`)	Runtime function pointer table with dynamic switching
Size Class Mapping	Linear search through class boundaries array	Bit manipulation (e.g., `clz` count leading zeros) for power-of-two
Free List Heads	Global array of pointers in `allocator_state_t`	Per-thread size class arrays for thread-local caching

B. Recommended File/Module Structure:

```
build-your-own-allocator/
├── include/
│   └── allocator.h           # Public API and strategy enums
├── src/
│   ├── allocator.c           # Top-level malloc/free implementations
│   ├── strategies/           # Allocation strategy component
│   │   ├── strategies.c      # find_free_block, strategy implementations
│   │   ├── segregated.c      # malloc_segregated, size class management
│   │   └── strategy_helpers.c # size_to_class_index, etc.
│   ├── block/               # Block management component
│   │   ├── block.c          # split_block, coalesce_block
│   │   └── free_list.c      # free_list_insert, free_list_remove
│   ├── os/                  # OS interface component
│   │   └── os_mem.c         # get_memory_from_os, os_sbrk, os_mmap
│   └── threading/           # Concurrency component (Milestone 4)
│       └── thread_cache.c   # thread_cache_malloc, thread_cache_free
└── tests/
    └── test_strategies.c    # Unit tests for allocation strategies
```

C. Core Logic Skeleton Code:

```c
/* In include/allocator.h */
typedef enum {
    FIRST_FIT,
    BEST_FIT,
    WORST_FIT,
    NEXT_FIT
} allocation_strategy_t;

/* In src/strategies/strategies.c */
/* Global configuration - set at compile time or init time */
#ifndef ALLOC_STRATEGY
#define ALLOC_STRATEGY FIRST_FIT
#endif

static allocation_strategy_t current_strategy = ALLOC_STRATEGY;
static block_header_t *next_fit_start = NULL;  /* For NEXT_FIT only */

/*
 * Searches the general free list for a block of at least 'needed_size' bytes.
 * Uses the currently configured strategy (first/best/worst/next-fit).
 * Returns pointer to suitable block header, or NULL if none found.
 */
block_header_t* find_free_block(size_t needed_size) {
    block_header_t *current = NULL;
    block_header_t *selected = NULL;
    size_t selected_waste = SIZE_MAX;  /* For BEST_FIT: smallest leftover */
    size_t selected_size = 0;          /* For WORST_FIT: largest block */

    // TODO 1: Determine starting point based on strategy
    //   - FIRST_FIT, BEST_FIT, WORST_FIT: start from free_list_head
    //   - NEXT_FIT: start from next_fit_start (if not NULL), else free_list_head

    // TODO 2: Traverse free list until appropriate stopping condition
    //   Loop while current != NULL
    //   For each block:
    //     a. Calculate available payload size: block->size - HEADER_SIZE - FOOTER_SIZE
    //     b. If available >= needed_size:
    //        - FIRST_FIT: select this block, break loop
```

```
    //        - NEXT_FIT: select this block, update next_fit_start to current->next, break
    //        - BEST_FIT: track block with smallest (available - needed_size)
    //        - WORST_FIT: track block with largest available size
    //     c. Move to next block: current = current->next

    // TODO 3: For BEST_FIT/WORST_FIT, after full traversal:
    //   - BEST_FIT: select the block with minimal waste (if any found)
    //   - WORST_FIT: select the block with maximum available size (if any found)

    // TODO 4: If no block found, return NULL

    // TODO 5: Update search state (for NEXT_FIT):
    //   - If using NEXT_FIT and selected block found:
    //     Set next_fit_start to selected->next (or free_list_head if NULL)

    return selected;
}

/* In src/strategies/segregated.c */
/* Size class definitions - power-of-two boundaries */
#define NUM_SIZE_CLASSES 8
static const size_t size_class_boundaries[NUM_SIZE_CLASSES] = {
    16, 32, 64, 128, 256, 512, 1024, 2048
};

/*
 * Maps a total block size (including metadata) to a size class index.
 * Returns 0..(NUM_SIZE_CLASSES-1) if size <= largest class,
 * or -1 if size exceeds largest class (use general list).
 */
int size_to_class_index(size_t total_size) {
    // TODO 1: For each size class boundary in ascending order:
    //   If total_size <= boundary, return that index
    // TODO 2: If total_size exceeds all boundaries, return -1
    return -1;
}

/*
```

```
 * Allocates memory using segregated free lists for small allocations.
 * For requests > LARGEST_SIZE_CLASS, falls back to general free list search.
 */
void* malloc_segregated(size_t requested_size) {
    // TODO 1: Handle zero-size request (return NULL or unique pointer per policy)

    // TODO 2: Calculate total_needed = HEADER_SIZE + FOOTER_SIZE +
    //         requested_size + alignment padding

    // TODO 3: Check if total_needed exceeds largest size class:
    //   If yes: fall back to general list via find_free_block(total_needed)
    //           then proceed with standard allocation (split if needed)
    //           then return user pointer

    // TODO 4: Map total_needed to size class index using size_to_class_index()

    // TODO 5: Check if corresponding size class free list has blocks:
    //   If yes:
    //     a. Remove first block from that size class list
    //     b. Mark block as allocated (set allocated flag, preserve size)
    //     c. Return user pointer (block + HEADER_SIZE)

    // TODO 6: If size class list is empty:
    //   a. Search general free list for block of at least size_class_boundaries[index]
    //   b. If found:
    //        - Split block: allocated portion = exact size class boundary
    //        - Calculate remainder size
    //        - If remainder >= MIN_BLOCK_SIZE:
    //            Initialize remainder as free block
    //            Insert remainder into appropriate size class list
    //        - Mark allocated portion as in-use
    //        - Return user pointer
    //   c. If not found in general list:
    //        - Request memory from OS via get_memory_from_os()
    //        - Initialize as free block
    //        - Split if larger than needed
```

```
    //        - Insert remainder into appropriate size class list
    //        - Return allocated portion

    return NULL;
}
```

D. Strategy Selection Helper Code:

```c
/* In src/strategies/strategy_helpers.c */
/*
 * Changes the allocation strategy at runtime (if supported).
 * Returns 0 on success, -1 if strategy invalid.
 */
int set_allocation_strategy(allocation_strategy_t strategy) {
    // TODO 1: Validate strategy is FIRST_FIT, BEST_FIT, WORST_FIT, or NEXT_FIT
    // TODO 2: Update current_strategy global variable
    // TODO 3: If switching to NEXT_FIT, initialize next_fit_start to free_list_head
    // TODO 4: If switching from NEXT_FIT, reset next_fit_start to NULL
    return 0;
}

/*
 * Returns the current allocation strategy.
 */
allocation_strategy_t get_allocation_strategy(void) {
    return current_strategy;
}
```

E. Language-Specific Hints (C):

- **Size class mapping optimization:** For power-of-two classes, you can use bit operations: `int class = 32 - __builtin_clz(total_size - 1)` (with adjustment for minimum size). This avoids linear search through the boundaries array.
- **Strategy selection:** Use function pointers for cleaner code: define `typedef block_header_t* (*find_block_fn)(size_t);` and set `find_block = find_first_fit` etc.
- **Alignment calculation:** Remember that `total_needed` must account for alignment of the user pointer, not just the block start. Use: `ALIGN_UP(HEADER_SIZE + requested_size, ALIGNMENT) + FOOTER_SIZE`.
- **Free list operations:** When removing a block from a size class list, you must update both `next` and `prev` pointers of neighboring blocks, plus the class head pointer if removing the first block.

F. Milestone Checkpoint for Strategy Implementation:

After implementing the allocation strategy component (Milestone 2 and 3), you should be able to run this test program:

```c
#include "allocator.h"

#include <stdio.h>

#include <string.h>


int main() {

    // Test 1: First-fit behavior

    void *p1 = malloc(100);

    void *p2 = malloc(200);

    void *p3 = malloc(100);

    free(p2);  // Creates a free block between two allocations



    void *p4 = malloc(150);  // Should reuse p2's space (first-fit)



    // Test 2: Size class segregation

    void *small1 = malloc(32);  // Should come from size class 1 (32 bytes)

    void *small2 = malloc(32);  // Should come from same size class

    void *small3 = malloc(64);  // Should come from size class 2 (64 bytes)



    free(small1);

    void *small4 = malloc(32);  // Should reuse small1's block from size class list



    printf("All tests passed if no crash occurs\n");



    // Visualize heap to see size class organization

    heap_visualize();



    return 0;

}
```

Expected behavior:

- The program runs without segmentation faults or memory leaks.
- `p4` reuses the memory freed by `p2` (demonstrating free list reuse).
- `small4` reuses the memory freed by `small1` (demonstrating size class list reuse).
- `heap_visualize()` output shows blocks organized by size, with free blocks linked in their respective size class lists.

Signs something is wrong:

- **Crash on free:** Likely corrupted free list pointers or incorrect header calculation.
- **malloc returns same pointer twice:** Forgetting to mark blocks as allocated or remove from free list.
- **Size class lists remain empty:** Blocks not being inserted into correct lists after splitting/freeing.
- **Poor allocation performance:** Linear search still happening for small allocations (check size class mapping logic).

G. Debugging Tips for Strategy Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Small allocations (≤256B) still scan entire free list	Size class mapping incorrect or not being used	Add debug print in `malloc_segregated` to show which path taken; check `size_to_class_index` return value	Ensure `malloc_segregated` is the top-level function called by `malloc()`
Best-fit always returns same block as first-fit	Stopping search early on first adequate block	Add counter to track how many blocks examined; should be all blocks for best/worst-fit	Remove early exit condition; always traverse entire list
Next-fit creates "dead zones" where allocation fails	`next_fit_start` pointer stuck at end of list	Print `next_fit_start` before each search; should wrap to head when NULL	Reset `next_fit_start = free_list_head` when it becomes NULL during traversal
Blocks disappear after coalescing (not in any free list)	Not re-inserting merged block into appropriate list	Check which free list(s) blocks belong to before/after coalesce	After merging, call `free_list_insert()` with correct size class head
Internal fragmentation >50% for small allocations	Size class boundaries too coarse	Track actual vs. requested sizes; consider adding more classes (e.g., 24, 48, 96)	Adjust `size_class_boundaries` array or use geometric progression

5.4 Concurrency & Advanced Features

> **Milestone(s):** Milestone 4 (Thread Safety & mmap)

This component transforms our single-threaded memory allocator into a robust, production-ready system capable of handling concurrent programs while providing advanced debugging capabilities. As modern applications increasingly leverage multiple threads, the naive approach of protecting every allocation with a global lock would create severe **lock contention**, dramatically reducing performance under concurrent load. This section introduces the architectural patterns and implementation techniques that allow our teaching allocator to scale across threads while maintaining correctness and offering visibility into memory errors.

Mental Model: Multiple Workers in One Warehouse

Imagine a large warehouse with multiple workers (threads) who all need to fetch and return boxes (memory blocks) throughout their workday. In our initial single-threaded design, we had just one worker with exclusive access to the warehouse floor plan and tool shelf (the free list). With multiple workers, several problems emerge:

1. **Race Conditions**: If two workers simultaneously try to take the same box from a shelf, they might both believe they successfully claimed it, leading to the same memory being allocated twice (a critical bug).
2. **Lock Contention**: If we install a single turnstile at the warehouse entrance (a global mutex), workers must wait in line even when they're accessing completely different parts of the warehouse, severely slowing down parallel work.
3. **Cache Inefficiency**: When a worker travels to the central tool shelf for every single tool they need, they waste time walking back and forth instead of keeping frequently used tools nearby.

The solution follows real-world warehouse optimization patterns:

- **Personal Toolboxes (Thread-Local Caches)**: Each worker gets a small personal toolbox that holds recently used tools. They can access these tools instantly without walking to the central shelf or waiting for other workers. Only when their toolbox is empty or needs to return a tool do they visit the central area.
- **Specialized Stations (Per-Thread Arenas)**: For high-throughput operations, we give each worker their own dedicated workstation with a complete set of commonly used tools. They work entirely independently, eliminating contention entirely for most tasks.
- **Central Warehouse with Locking (Global Arena)**: For oversized items or when workstations overflow, workers occasionally access the main warehouse through a controlled entry point with proper coordination.

This mental model illustrates the core insight: **most memory allocations and deallocations are thread-local**—a thread allocates memory, uses it, and frees it within the same thread's execution flow. By optimizing for this common case with thread-local storage, we can achieve near-single-threaded performance even in highly concurrent programs.

ADR: Per-Thread Arenas with a Global Fallback

> **Decision: Hybrid Arena Strategy with Thread-Local Caches**
>
> - **Context**: Our allocator must support concurrent access from multiple threads while maintaining high performance. A single global lock would serialize all allocations, creating unacceptable contention. We need a strategy that minimizes cross-thread synchronization while still allowing memory to be efficiently shared and reclaimed across thread boundaries when necessary.
> - **Options Considered**:
>   1. **Global Lock (Single Mutex)**: Protect all allocator operations with a single pthread mutex. Simplest to implement and guarantees correctness but suffers severe performance degradation under concurrent load.
>   2. **Per-Thread Heaps (Complete Isolation)**: Give each thread its completely independent heap region via separate `sbrk` calls or `mmap` regions. No locking required, but memory cannot be reclaimed by other threads, leading to potential blowup if one thread allocates heavily then terminates.
>   3. **Hybrid Arena Strategy**: Each thread gets a private "arena" (a subset of the heap) for most allocations, with thread-local free lists for fast access. Large allocations and arena exhaustion fall back to a global, locked arena. Freed blocks return to their arena of origin when possible.
> - **Decision**: Option 3 (Hybrid Arena Strategy) with thread-local caches as the first line of defense, backed by per-thread arenas, with a global locked arena as final fallback.
> - **Rationale**:
>   - **Performance**: Thread-local operations require no locking, providing O(1) allocation/free for the common case where a thread reuses its own recently freed memory.
>   - **Memory Efficiency**: Unlike complete isolation, memory from terminated threads can eventually be recycled through the global arena, preventing wasteful fragmentation.
>   - **Progressive Complexity**: This approach allows incremental implementation—starting with a simple global lock, then adding thread-local caches, then full arenas—which aligns with our teaching goals.
>   - **Industry Validation**: Modern allocators like jemalloc and tcmalloc use similar patterns, validating this as a production-worthy architecture.
> - **Consequences**:
>   - **Positive**: Dramatically reduces lock contention; improves cache locality (thread-local structures likely reside in CPU cache); maintains memory efficiency across thread lifetimes.
>   - **Negative**: Increased complexity in managing multiple memory domains; potential for "memory blowup" if threads don't balance allocation/free patterns; requires careful handling of thread creation/destruction.

Option	Pros	Cons	Chosen?
Global Lock	Simple implementation (few lines of code); Guaranteed correctness; Easy to reason about	Severe performance bottleneck under concurrency; All threads block on every allocation	No
Per-Thread Heaps	Zero locking contention; Excellent locality and performance	Memory cannot be reclaimed across threads; Potential for massive fragmentation; Wastes address space	No
Hybrid Arena Strategy	Best of both worlds: fast thread-local path + cross-thread memory reuse; Scalable performance; Industry-proven pattern	Complex implementation; Must handle thread lifecycle; More metadata overhead	**Yes**

Common Pitfalls in Concurrent Allocator Design

⚠️ Pitfall: Forgetting to Initialize Thread-Local Storage

- **Description**: Assuming thread-local variables automatically initialize to NULL or valid pointers. In C, thread-local storage has static initialization (zero) but may need explicit setup for complex structures.
- **Why it's wrong**: The first time a thread calls `malloc`, its thread-local cache pointer might be NULL, causing a segfault when dereferenced.
- **Fix**: Implement an `init_thread_cache()` function that checks if the thread-local cache is initialized and sets it up if not. Call this at the beginning of `thread_cache_malloc`.

⚠️ Pitfall: Incorrect Lock Ordering Leading to Deadlock

- **Description**: When multiple locks are involved (e.g., global arena mutex + per-arena mutexes), acquiring them in different orders across threads can create circular wait conditions.
- **Why it's wrong**: Thread 1 locks Arena A then tries to lock Global; Thread 2 locks Global then tries to lock Arena A → both threads wait forever.
- **Fix**: Establish and strictly follow a **lock hierarchy**: always acquire global lock first (if needed), then arena-specific locks. Better yet, design so that thread-local operations never need global lock.

⚠️ Pitfall: Lost Memory in Thread Termination

- **Description**: When a thread exits without returning its thread-local cache blocks to the global pool, that memory becomes permanently lost (a memory leak).
- **Why it's wrong**: The memory is still allocated from the OS but no longer accessible to any thread—classic memory leak that grows with thread churn.
- **Fix**: Implement thread destruction callbacks (pthread key destructors) that flush thread-local caches back to the global arena when a thread terminates.

⚠️ Pitfall: False Sharing in Thread-Local Caches

- **Description**: Two threads' cache data structures end up on the same CPU cache line. When one thread modifies its cache, it invalidates the cache line for the other thread, causing unnecessary cache misses.
- **Why it's wrong**: Performance degradation despite using thread-local storage; CPUs spend time synchronizing cache lines instead of computing.
- **Fix**: Pad thread-local structures to cache line size (typically 64 bytes) using alignment attributes or explicit padding fields.

Implementation Guidance

This implementation guidance bridges the conceptual design to concrete C code, providing building blocks for thread-safe allocation with performance optimizations.

A. Technology Recommendations Table

Component	Simple Option (Learning Focus)	Advanced Option (Production Ready)
Thread Synchronization	pthread mutexes (`pthread_mutex_t`) with global lock	Spinlocks for fast paths, mutexes for slow paths; atomic operations for reference counting
Thread-Local Storage	pthread thread-specific keys (`pthread_key_t`)	Compiler extensions (`__thread` or `_Thread_local`) for direct access without function call overhead
Memory Backing	`sbrk` for main heap, `mmap` for large blocks	Exclusive `mmap` for all allocations (no `sbrk`); `madvise` for memory usage hints
Debugging Support	"Magic number" in block headers; heap visualization function	Address sanitizer integration; allocation backtraces; guarded pages for overflow detection
Cache Management	Simple LIFO free list per thread	Size-segregated bins within thread caches; batch transfers to global arena

B. Recommended File/Module Structure

```
build-your-own-allocator/
├── include/
│   └── allocator.h          # Public API and main data structures
├── src/
│   ├── allocator.c          # Top-level malloc/free implementations
│   ├── os_mem.c             # OS interface (sbrk, mmap wrappers)
│   ├── block.c              # Block management (split, coalesce)
│   ├── free_list.c          # Free list operations
│   ├── strategies.c         # Allocation strategies (first-fit, etc.)
│   ├── segregated.c         # Segregated free lists implementation
│   ├── thread_cache.c       # Thread-local cache management  ← NEW
│   ├── concurrency.c        # Locking and arena management  ← NEW
│   └── debug.c              # Debugging and visualization
├── tests/
│   ├── test_basic.c
│   ├── test_concurrent.c    # Thread safety tests          ← NEW
│   └── test_fragmentation.c
└── Makefile
```

C. Infrastructure Starter Code (Complete)

Complete pthread mutex wrapper with error checking:

```c
/* concurrency.c - Thread synchronization primitives */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include "allocator.h"

/* Global mutex for protecting shared allocator state */
pthread_mutex_t global_mutex = PTHREAD_MUTEX_INITIALIZER;

/**
 * mutex_lock - Wrapper for pthread_mutex_lock with error checking
 * @mutex: Pointer to mutex to lock
 *
 * On error, prints message and exits. In production code,
 * you might want to handle this more gracefully.
 */
void mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    if (rc != 0) {
        fprintf(stderr, "FATAL: pthread_mutex_lock failed with error %d\n", rc);
        exit(EXIT_FAILURE);
    }
}

/**
 * mutex_unlock - Wrapper for pthread_mutex_unlock with error checking
 * @mutex: Pointer to mutex to unlock
 */
void mutex_unlock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_unlock(mutex);
    if (rc != 0) {
        fprintf(stderr, "FATAL: pthread_mutex_unlock failed with error %d\n", rc);
        exit(EXIT_FAILURE);
    }
}

/**
 * mutex_init - Initialize a mutex with error checking
```

```
 * @mutex: Pointer to mutex to initialize
 */
void mutex_init(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_init(mutex, NULL);
    if (rc != 0) {
        fprintf(stderr, "FATAL: pthread_mutex_init failed with error %d\n", rc);
        exit(EXIT_FAILURE);
    }
}

/**
 * mutex_destroy - Destroy a mutex with error checking
 * @mutex: Pointer to mutex to destroy
 */
void mutex_destroy(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_destroy(mutex);
    if (rc != 0) {
        fprintf(stderr, "WARNING: pthread_mutex_destroy failed with error %d\n", rc);
        /* Continue execution - some systems fail if mutex is locked */
    }
}
```

Complete thread-local storage management:

```c
/* thread_cache.c - Per-thread cache management */

#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include "allocator.h"

/* Thread-local cache structure */
typedef struct thread_cache {
    block_header_t *free_lists[NUM_SIZE_CLASSES];  /* Per-size-class free lists */
    size_t allocation_count;                       /* Statistics */
    size_t hit_count;                              /* Cache hit statistics */
    struct thread_cache *next;                     /* For global list of caches */
} thread_cache_t;

/* Key for thread-local storage */
static pthread_key_t cache_key;
/* Global list of all thread caches (for cleanup) - protected by global_mutex */
static thread_cache_t *all_caches = NULL;

/**
 * destroy_thread_cache - Destructor called when thread exits
 * @cache_ptr: Pointer to thread's cache
 *
 * This function is automatically called by pthreads when a thread
 * terminates. It returns the thread's cached blocks to the global arena.
 */
static void destroy_thread_cache(void *cache_ptr) {
    thread_cache_t *cache = (thread_cache_t *)cache_ptr;
    if (!cache) return;

    /* Acquire global lock to safely merge cache back into global arena */
    mutex_lock(&global_mutex);

    /* For each size class, merge cache's free list into global free list */
    for (int i = 0; i < NUM_SIZE_CLASSES; i++) {
        block_header_t *block = cache->free_lists[i];
        while (block) {
```

```
            block_header_t *next = block->next;
            /* Remove from thread cache list */
            block->prev = NULL;
            block->next = NULL;
            /* Insert into global segregated list */
            // TODO: Call global free list insertion function
            block = next;
        }
        cache->free_lists[i] = NULL;
    }

    /* Remove from global list of caches */
    // TODO: Remove cache from all_caches linked list

    mutex_unlock(&global_mutex);

    /* Free the cache structure itself */
    free(cache);
}

/**
 * init_thread_cache_system - Initialize the thread cache subsystem
 *
 * Called once during allocator initialization.
 */
void init_thread_cache_system(void) {
    /* Create the thread-specific data key */
    int rc = pthread_key_create(&cache_key, destroy_thread_cache);
    if (rc != 0) {
        fprintf(stderr, "FATAL: pthread_key_create failed with error %d\n", rc);
        exit(EXIT_FAILURE);
    }

    /* Initialize global cache list as empty */
    all_caches = NULL;
}
```

```c
/**
 * get_thread_cache - Get or create thread-local cache for current thread
 *
 * Returns: Pointer to thread's cache, never NULL
 */
static thread_cache_t *get_thread_cache(void) {
    thread_cache_t *cache = pthread_getspecific(cache_key);

    if (!cache) {
        /* First time this thread calls malloc - create a cache */
        cache = (thread_cache_t *)malloc(sizeof(thread_cache_t));
        if (!cache) {
            /* If malloc fails here, we're in serious trouble */
            return NULL;
        }

        /* Initialize cache */
        memset(cache, 0, sizeof(thread_cache_t));
        for (int i = 0; i < NUM_SIZE_CLASSES; i++) {
            cache->free_lists[i] = NULL;
        }
        cache->allocation_count = 0;
        cache->hit_count = 0;
        cache->next = NULL;

        /* Store in thread-local storage */
        int rc = pthread_setspecific(cache_key, cache);
        if (rc != 0) {
            free(cache);
            return NULL;
        }

        /* Add to global list (protected by global_mutex) */
        mutex_lock(&global_mutex);
        cache->next = all_caches;
        all_caches = cache;
```

```
        mutex_unlock(&global_mutex);
    }


    return cache;
}
```

D. Core Logic Skeleton Code

Thread-cached malloc implementation:

```c
/* thread_cache.c - Continued */

/**
 * thread_cache_malloc - Allocate memory using thread-local cache with fallback
 * @size: Requested allocation size in bytes
 *
 * This is the primary entry point for concurrent allocations.
 * 1. Check if size qualifies for thread-local cache (small allocation)
 * 2. If yes, try to allocate from thread's cache without locking
 * 3. If cache miss, fall back to global arena with locking
 * 4. For large allocations, bypass cache and use global arena directly
 *
 * Returns: Pointer to allocated memory, or NULL on failure
 */
void *thread_cache_malloc(size_t size) {
    /* TODO 1: Check if this is a large allocation (>= SBRK_THRESHOLD)
     *         If yes, skip thread cache and go directly to global arena */

    /* TODO 2: Check if size fits in segregated size classes (<= LARGEST_SIZE_CLASS)
     *         If not, use global arena */

    /* TODO 3: Get thread-local cache using get_thread_cache()
     *         Handle error if cache creation fails */

    /* TODO 4: Map request size to size class index using size_to_class_index() */

    /* TODO 5: Check if thread cache has a free block for this size class
     *         If yes, remove it from cache list and return it to user
     *         Increment cache hit statistics */

    /* TODO 6: Cache miss - acquire global mutex for shared arena access */

    /* TODO 7: Try to allocate from global segregated free lists
     *         Use malloc_segregated() or find_free_block() depending on size */

    /* TODO 8: If allocation succeeds, release global mutex */

```

```
    /* TODO 9: If allocation fails in global arena, request more memory from OS
     *         using get_memory_from_os(), then retry */

    /* TODO 10: If all attempts fail, release mutex and return NULL */

    return NULL; /* Placeholder */
}

/**
 * thread_cache_free - Free memory using thread-local cache with coalescing
 * @ptr: Pointer to memory to free
 *
 * 1. Determine if block should go to thread cache or global arena
 * 2. For small blocks, add to thread cache for fast reuse
 * 3. When cache exceeds threshold, flush some blocks to global arena
 * 4. For large or mmap blocks, free directly to global arena
 */
void thread_cache_free(void *ptr) {
    if (!ptr) return; /* free(NULL) is allowed */

    /* TODO 1: Get block header from user pointer using get_header_from_ptr() */

    /* TODO 2: Check if block is mmap'd using IS_MMAP_FLAG in size field
     *         If yes, use os_munmap() directly and return */

    /* TODO 3: Check if block size qualifies for thread cache (small enough) */

    /* TODO 4: If yes, get thread-local cache and add block to appropriate size class list
     *         Check cache size threshold - if exceeded, flush some blocks to global arena */

    /* TODO 5: If not using thread cache, acquire global mutex and free to global arena
     *         Mark block free, coalesce with neighbors, insert into global free list */

    /* TODO 6: Release global mutex if acquired */
}
```

Per-thread arena management skeleton:

```c
/* concurrency.c - Per-thread arena management */

/**
 * arena_t - Per-thread arena structure
 *
 * Each arena has its own heap region managed independently.
 * Threads primarily allocate from their own arena to avoid locking.
 */
typedef struct arena {
    void *heap_base;              /* Base address of this arena's heap */
    size_t heap_size;             /* Current size of arena heap */
    block_header_t *free_list;    /* Arena-specific free list */
    pthread_mutex_t lock;         /* Arena-specific lock (for shared access) */
    thread_cache_t *cache;        /* Associated thread cache */
    struct arena *next;           /* For global arena list */
    int id;                       /* Arena identifier for debugging */
} arena_t;

/* Global array/linked list of arenas - protected by global_mutex */
static arena_t *arena_list = NULL;
static int next_arena_id = 0;
static const int MAX_ARENAS = 8;  /* Limit to prevent arena explosion */

/**
 * get_arena_for_thread - Get or create arena for current thread
 *
 * Implements the "next-fit" arena assignment: round-robin through existing
 * arenas to balance load. Creates new arena if needed and allowed.
 *
 * Returns: Pointer to arena for current thread, or NULL if all arenas busy
 */
static arena_t *get_arena_for_thread(void) {
    thread_cache_t *cache = get_thread_cache();
    if (!cache) return NULL;

    /* TODO 1: Check if current thread already has an assigned arena
     *         (could store this in thread-local storage) */
```

```
    /* TODO 2: If not, acquire global_mutex to safely search arena list */

    /* TODO 3: Find least-loaded arena (based on allocation count or free memory) */

    /* TODO 4: If all arenas above load threshold and MAX_ARENAS not reached,
     *         create new arena with fresh sbrk region */

    /* TODO 5: Initialize new arena: allocate heap via sbrk, set up initial
     *         free block, initialize mutex, assign ID */

    /* TODO 6: Add arena to global arena_list */

    /* TODO 7: Assign arena to current thread (store in thread-local storage) */

    /* TODO 8: Release global_mutex and return arena pointer */

    return NULL; /* Placeholder */
}

/**
 * arena_malloc - Allocate from per-thread arena
 * @size: Requested allocation size
 *
 * First tries thread cache, then arena free list, then expands arena heap.
 * Only uses global lock when creating new arena or expanding heap.
 */
void *arena_malloc(size_t size) {
    /* TODO 1: Get arena for current thread using get_arena_for_thread() */

    /* TODO 2: Try thread cache first (fast path, no locking) */

    /* TODO 3: If cache miss, lock arena-specific mutex (not global!) */

    /* TODO 4: Search arena's free list using preferred strategy */

    /* TODO 5: If no suitable block, expand arena heap via sbrk */
```

```
    /* TODO 6: If arena expansion fails, fall back to global arena */


    /* TODO 7: Unlock arena mutex and return allocated block */


    return NULL; /* Placeholder */
}
```

E. Language-Specific Hints

1. **Thread-Local Storage in C**:
   - Use `pthread_key_t` for portability across all POSIX systems
   - Alternative: C11 `_Thread_local` keyword for compiler-supported TLS (faster but less portable)
   - Initialize TLS keys once with `pthread_once()` for thread-safe one-time initialization
2. **Mutex Best Practices**:
   - Always check pthread function return values in development builds
   - Use `pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK)` during development to catch deadlocks
   - For performance-critical sections, consider spinlocks (`pthread_spin_lock()`) for very short critical sections
3. **Memory Ordering**:
   - When implementing lock-free structures within thread caches, use C11 atomics (`<stdatomic.h>`) with appropriate memory order
   - Default to `memory_order_seq_cst` (sequential consistency) unless you deeply understand the memory model
4. **Thread Cleanup**:
   - Register destructors with `pthread_key_create(&key, destructor_fn)` to automatically clean up thread resources
   - Consider using `pthread_cleanup_push()` and `pthread_cleanup_pop()` for cancellation points

F. Milestone Checkpoint

After implementing thread safety features, verify correctness with this test:

```c
/* test_concurrent.c - Basic thread safety verification */

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include "allocator.h"


#define NUM_THREADS 4

#define ALLOCS_PER_THREAD 1000

#define ALLOC_SIZE 64


void *thread_func(void *arg) {

    int thread_id = *(int*)arg;

    void *pointers[ALLOCS_PER_THREAD];



    /* Each thread allocates, writes, frees */

    for (int i = 0; i < ALLOCS_PER_THREAD; i++) {

        pointers[i] = malloc(ALLOC_SIZE);

        if (!pointers[i]) {

            fprintf(stderr, "Thread %d: malloc failed at iteration %d\n",

                    thread_id, i);

            return NULL;

        }

        /* Write to memory to ensure it's accessible */

        *(int*)pointers[i] = thread_id * 1000 + i;

    }



    /* Verify written values */

    for (int i = 0; i < ALLOCS_PER_THREAD; i++) {

        if (*(int*)pointers[i] != thread_id * 1000 + i) {

            fprintf(stderr, "Thread %d: Memory corruption at %p\n",

                    thread_id, pointers[i]);

        }

        free(pointers[i]);

    }



    printf("Thread %d completed successfully\n", thread_id);

    return NULL;
```

```
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    /* Initialize allocator */
    // TODO: Call allocator initialization if needed

    /* Create threads */
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        if (pthread_create(&threads[i], NULL, thread_func, &thread_ids[i]) != 0) {
            perror("pthread_create");
            exit(EXIT_FAILURE);
        }
    }

    /* Wait for all threads */
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All threads completed. Running heap visualization:\n");
    heap_visualize();

    return 0;
}
```

Expected Behavior:

- Program runs without crashes or segmentation faults
- No "Memory corruption" error messages appear
- Each thread prints "Thread X completed successfully"
- Final heap visualization shows reasonable fragmentation (some free blocks from different arenas)
- Memory usage should stabilize (no unbounded growth across runs)

Signs of Problems:

- **Segfaults during allocation**: Likely race condition in global free list manipulation
- **Memory corruption values**: Threads overwriting each other's memory (wrong arena assignment)
- **Deadlock/hang**: Incorrect lock ordering or missing unlock

- **Memory leak in visualization**: Thread caches not being flushed on thread exit

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Segfault only with multiple threads	Race condition: pointer read while another thread modifies it	Add `MAGIC` number to block headers; check magic in all functions; run with `helgrind` or `tsan`	Add missing lock; use atomic operations; ensure proper memory barriers
Performance worse with thread caching	False sharing: thread cache structures on same cache line	Use `perf` to check cache misses; add padding between thread cache arrays	Align thread cache structures to cache line boundary (64 bytes)
Memory leak proportional to thread count	Thread cache not flushed on thread exit	Check `destroy_thread_cache` is called; add logging to destructor	Ensure `pthread_key_create` with destructor; manually flush caches in test teardown
Deadlock when allocating in signal handler	Signal handler interrupts locked thread, tries to allocate (locks again)	Notice deadlock only occurs with signals; use debugger to see stack traces	Never call malloc in signal handlers; use `malloc` that's async-signal-safe or pre-allocate
Corrupted free list pointers	Thread writes to freed memory that's already in another thread's cache	Add canary values before/after allocations; use memory poisoning on free	Clear memory on free (`memset(ptr, 0xAA, size)`); add use-after-free detection
High contention on global mutex	Too many threads falling back to global arena	Profile with `perf lock` ; count cache miss rate	Increase number of arenas; tune size class thresholds; improve thread cache hit rate

Allocator-Specific Debugging Techniques:

1. **Magic Number Validation**: Add a constant `MAGIC` (e.g., `0xDEADBEEF`) to every block header. In every function that receives a block pointer, verify the magic number matches before proceeding. This catches corruption immediately.
2. **Thread ID Tagging**: Store the allocating thread's ID (`pthread_self()`) in block metadata. When freeing, verify the freeing thread is either the same thread or explicitly checks cross-thread freeing is allowed.
3. **Allocation Backtraces**: In debug mode, use `backtrace()` from `<execinfo.h>` to record the call stack for each allocation. Store in a circular buffer to identify leaks.
4. **Guarded Pages**: For large allocations, use `mmap` with `PROT_NONE` guard pages before and after the allocation to detect buffer overflows immediately via segfault.
5. **Statistics Collection**: Maintain counters for cache hit rates, lock acquisition counts, and allocation size distributions. Print periodically or on demand to tune parameters.

---

6. Interactions and Data Flow

> **Milestone(s):** Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

This section traces the journey of individual memory allocation and deallocation requests through the complete allocator system. Understanding these **data flow pathways** is critical for visualizing how the components interact in practice—like following a package through a complex logistics network. We'll examine two concrete scenarios: a medium-sized allocation in a multithreaded environment using segregated lists, and a deallocation that triggers coalescing. These narratives reveal how design decisions manifest in runtime behavior and illustrate the **control flow decisions** made at each step.

Sequence: malloc(64) in a Thread-Safe, Segregated Allocator

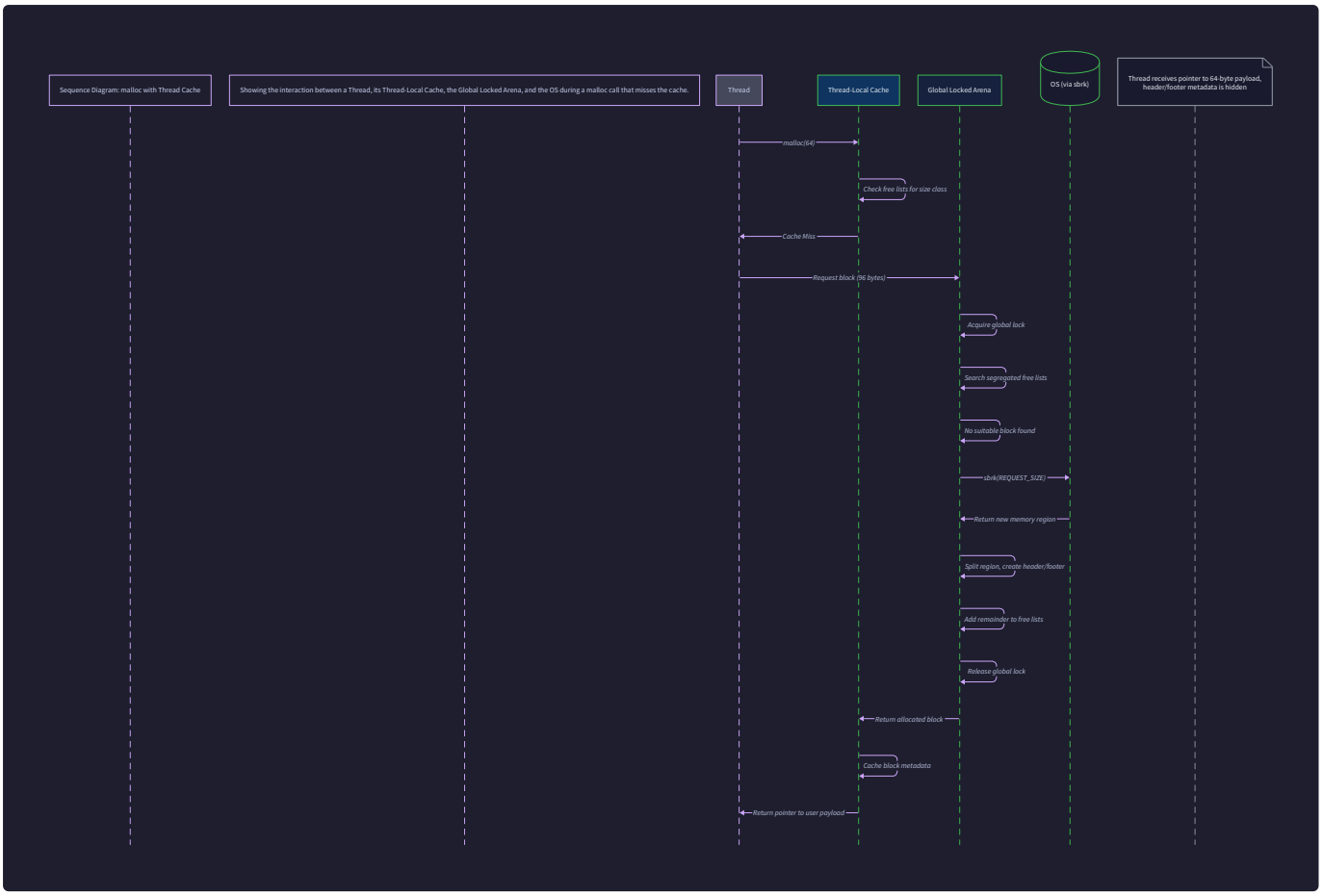
Mental Model: A Worker Ordering Supplies from a Smart Warehouse
Imagine a factory worker (thread) needs a 64cm shelf unit. Instead of wandering the entire warehouse, they first check their personal toolbox (thread-local cache). If empty, they consult a specialized catalog (segregated free lists) that directs them to the exact aisle containing 64cm shelves. Only if

that aisle is empty do they contact the central warehouse manager (global arena), who might need to expand the warehouse (call the OS) or split a larger shelf. This layered approach minimizes time spent searching and waiting for others.

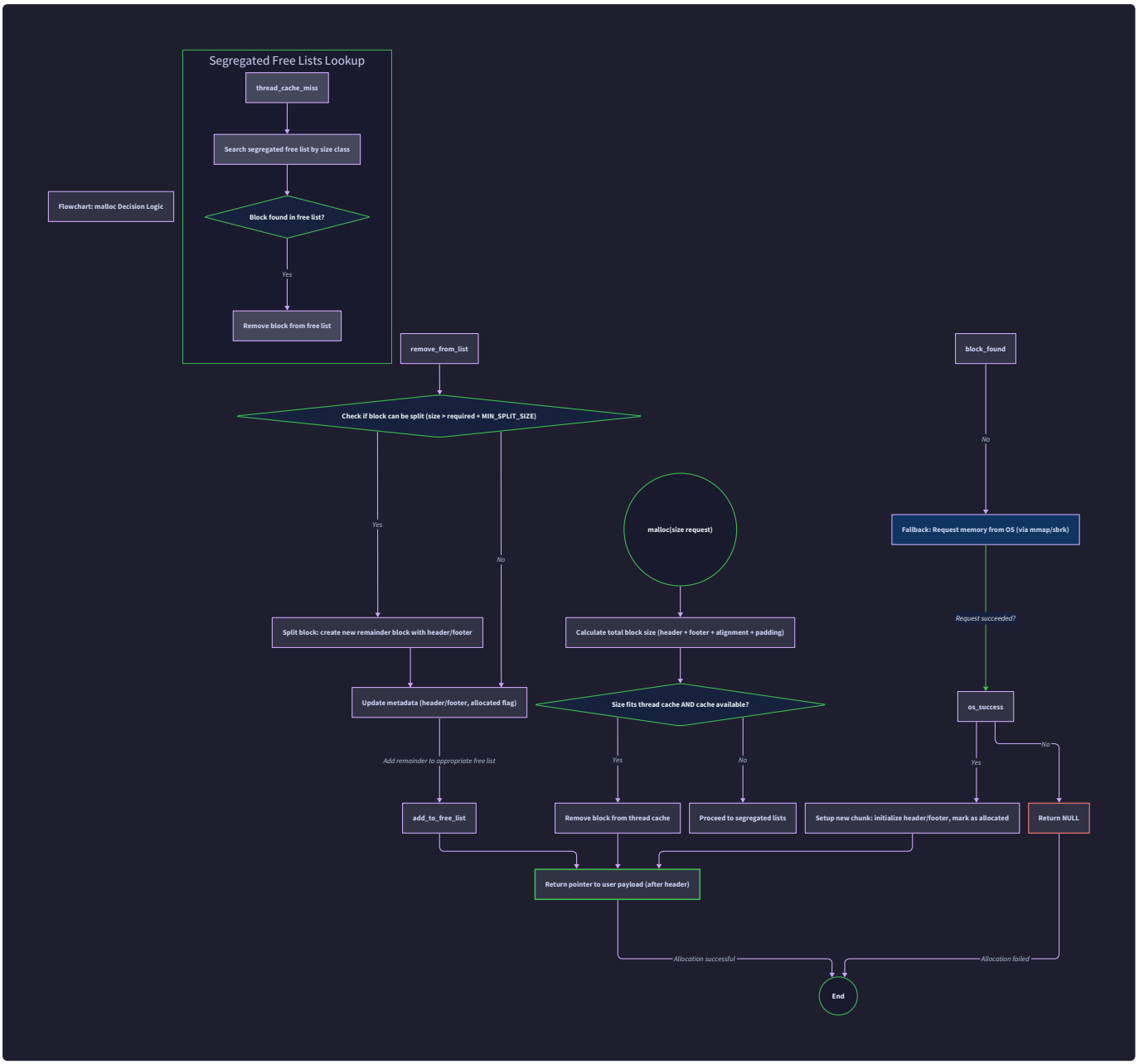
Now let's trace the actual steps when a thread calls `malloc(64)` in our complete allocator implementation:

1. **Request Entry and Size Adjustment**
   The user calls `malloc(64)`. The allocator first calculates the total required block size using `get_total_block_size(64)`, which adds:
   - Size of `block_header_t` (including `next` and `prev` pointers)
   - 8-byte `ALIGNMENT` padding to ensure the user payload starts at an aligned address
   - Size of `footer_t` for boundary tags
   - Additional padding to meet `MIN_BLOCK_SIZE` requirements The calculated total might be 96 bytes (example). The allocator also checks if the request exceeds the `SBRK_THRESHOLD` (128KB) to determine if it should use `mmap` directly—64 bytes is far below this threshold, so it proceeds with normal heap allocation.
2. **Thread-Local Cache Check**
   The function `thread_cache_malloc` is invoked. It calls `get_thread_cache()` to obtain the `thread_cache_t` structure for the current thread (creating one if none exists). It maps the request size (64 bytes) to a size class index using `size_to_class_index(64)`. Assuming our size classes are [16, 32, 64, 128, 256, 512, 1024, 2048], 64 maps to index 2. The thread cache checks its `free_lists[2]` for an available block. If present, it removes the block from the list, marks it allocated, and returns the user pointer immediately—this is the **fast path** requiring no locks.
3. **Segregated List Lookup (Cache Miss Path)**
   If the thread-local free list for size class 64 is empty, control passes to `malloc_segregated(64)`. This function checks the **global segregated free list** for the same size class by calling `get_size_class_head(2)`. These global segregated lists are protected by the arena's lock. The function must now acquire the lock via `mutex_lock(&arena->lock)`.
4. **Global Free List Search**
   With the lock held, `malloc_segregated` examines the free list head for size class 64. If a block exists, it performs `free_list_remove(block)` to detach it, marks it allocated, releases the lock, and returns. If empty, it proceeds to search the **general free list** (for blocks larger than the largest size class) by calling `find_free_block(96)` with the configured strategy (e.g., `FIRST_FIT`).
5. **Block Splitting and Heap Expansion**
   Suppose `find_free_block` locates a free block of 256 bytes. Since 256 > 96 + `MIN_BLOCK_SIZE`, `split_block` is called:
   - The original 256-byte block is divided at offset 96
   - A new header/footer are created for the allocated portion (96 bytes)
   - The remaining 160-byte portion becomes a new free block
   - The new free block is inserted into the appropriate segregated list (size class 128) The allocated block's header `allocated` flag is set to 1, and the `IS_MMAP_FLAG` remains 0.
6. **OS Fallback When No Free Block Exists**
   If `find_free_block` returns NULL (no sufficiently large free block), the allocator calls `get_memory_from_os(96, &is_large)`. Since 96 < `SBRK_THRESHOLD`, it uses `os_sbrk` to extend the heap. The new memory is initialized as a single free block using `init_block`, then immediately split (as in step 5) to satisfy the request. The remainder is added to the free lists.
7. **Pointer Return and Bookkeeping**
   Finally, the allocator calculates the user pointer by adding the header size to the block start address, ensuring 8-byte alignment. The thread cache's `allocation_count` is incremented. The mutex is released, and the pointer (e.g., `0x7f8a4b002040`) is returned to the caller.

The entire journey can be visualized in the sequence diagram:



The decision logic at each branch point follows this flowchart:



Key Data Transformations During malloc(64):

Stage	Input	Output	Critical Operations
Size calculation	User size: 64	Total size: 96	`get_total_block_size`, alignment padding
Thread cache lookup	Size class index: 2	Block pointer or NULL	`thread_cache_t->free_lists[2]`
Global search	Total size: 96	Free block pointer	`find_free_block` with strategy
Block splitting	Free block (256 bytes)	Allocated block (96 bytes) + Free block (160 bytes)	`split_block`, header/footer updates
OS request	Request size: 96	Raw memory from `sbrk`	`get_memory_from_os`, heap expansion
Pointer calculation	Block start address	User pointer (aligned)	Header size offset, alignment verification

Sequence: free(ptr) with Coalescing

Mental Model: Returning a Shelf Unit to the Warehouse

When a worker returns a shelf unit, they don't just drop it anywhere. First, they check if adjacent shelves are also empty—if so, they merge them into a single larger unit (coalescing). Then they decide whether to place it in their personal toolbox (thread-local cache) for quick reuse or in the main warehouse (global arena) for others. This careful placement minimizes fragmentation and optimizes future access.

Now let's trace the steps when a thread calls `free(ptr)` where `ptr` was previously allocated by `malloc(64)`:

1. **Pointer Validation and Header Derivation**
   The `free(ptr)` function first validates `ptr` is not NULL. It then calls `get_header_from_ptr(ptr)`, which subtracts the header size from the user pointer to locate the `block_header_t`. The header's `size` field indicates the total block size (96 bytes in our example), and the `allocated` flag should be 1. The function checks for the `IS_MMAP_FLAG`—if set, it would directly call `os_munmap`, but our block was allocated via `sbrk`, so it proceeds with heap deallocation.

2. **Thread-Local Cache Attempt**
   Control passes to `thread_cache_free(ptr)`, which retrieves the current thread's `thread_cache_t`. The function checks if the thread cache has capacity (e.g., fewer than MAX_CACHED blocks per size class). If so, it inserts the block into the appropriate size-class list (`free_lists[2]` for 64-byte class), marks it free in the header, and returns immediately—again avoiding lock acquisition.

3. **Global Free List Path**
   If the thread cache is full or caching is disabled, the function acquires the arena lock via `mutex_lock(&arena->lock)`. It marks the block as free by setting `header->allocated = 0` and updates the footer (via boundary tag) to reflect the free status and size.

4. **Coalescing with Next Block**
   The coalescing process begins with the next physically adjacent block. Using the current block's size, the allocator computes the address of the next block: `next_block = (void*)header + header->size`. It checks if `next_block` is within heap bounds and if its header's `allocated` flag is 0 (free). If free:

   - Remove `next_block` from its free list using `free_list_remove`
   - Merge by adding `next_block->size` to the current block's `size`
   - Update the new combined block's footer (at the end of the merged region)
   - The resulting block is now larger

5. **Coalescing with Previous Block**
   To check the previous block, the allocator examines the **footer** immediately before the current block's header. This footer contains the previous block's size and allocation status. If the previous block is free:

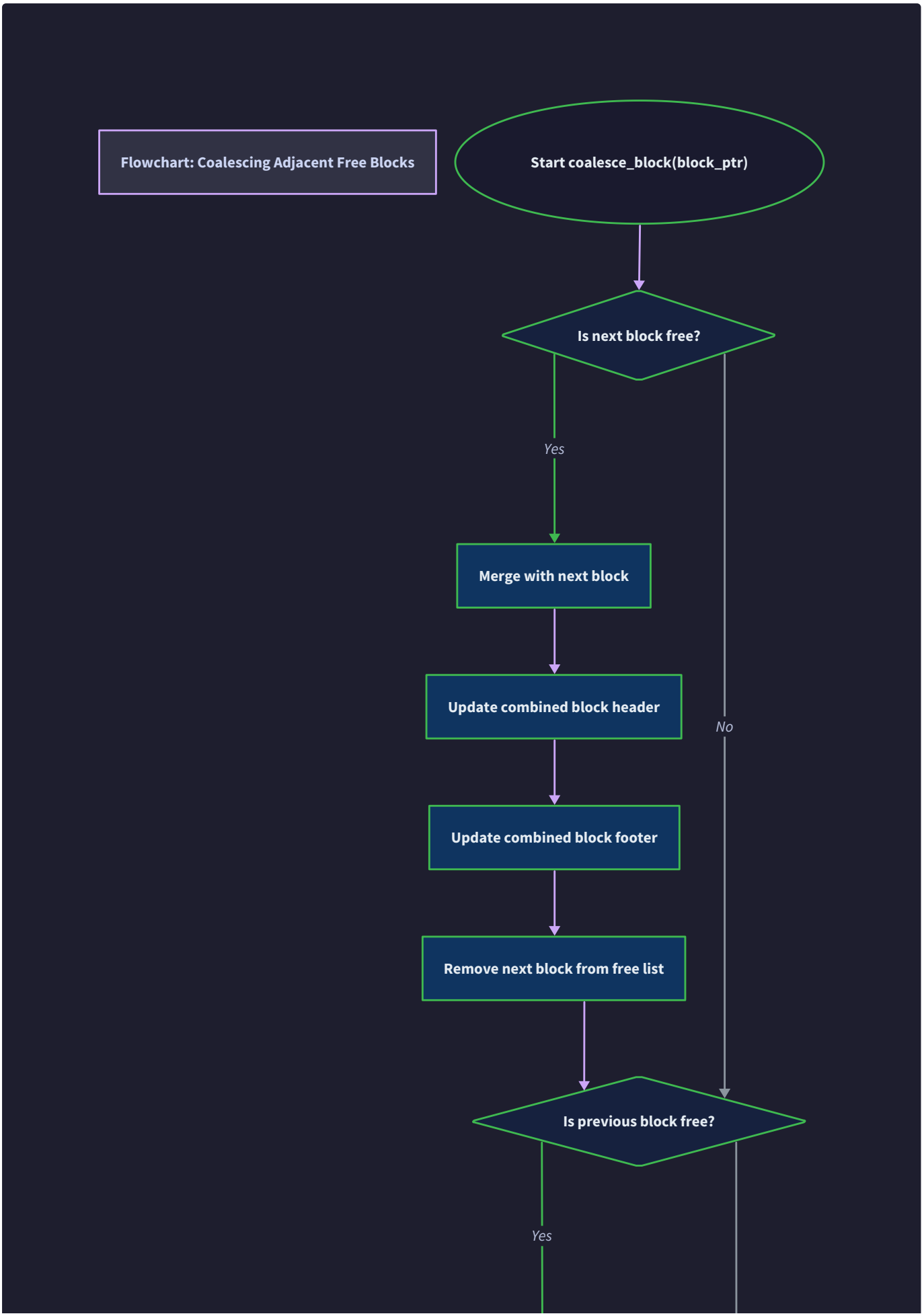
   - Locate the previous block's header: `prev_block = (void*)header - prev_footer->size`
   - Remove `prev_block` from its free list
   - Merge by adding the current block's size to `prev_block->size`
   - Update the footer at the end of the current block (now part of `prev_block`)
   - The current block pointer now becomes `prev_block` If both neighbors are free, the three blocks merge into one large free block.

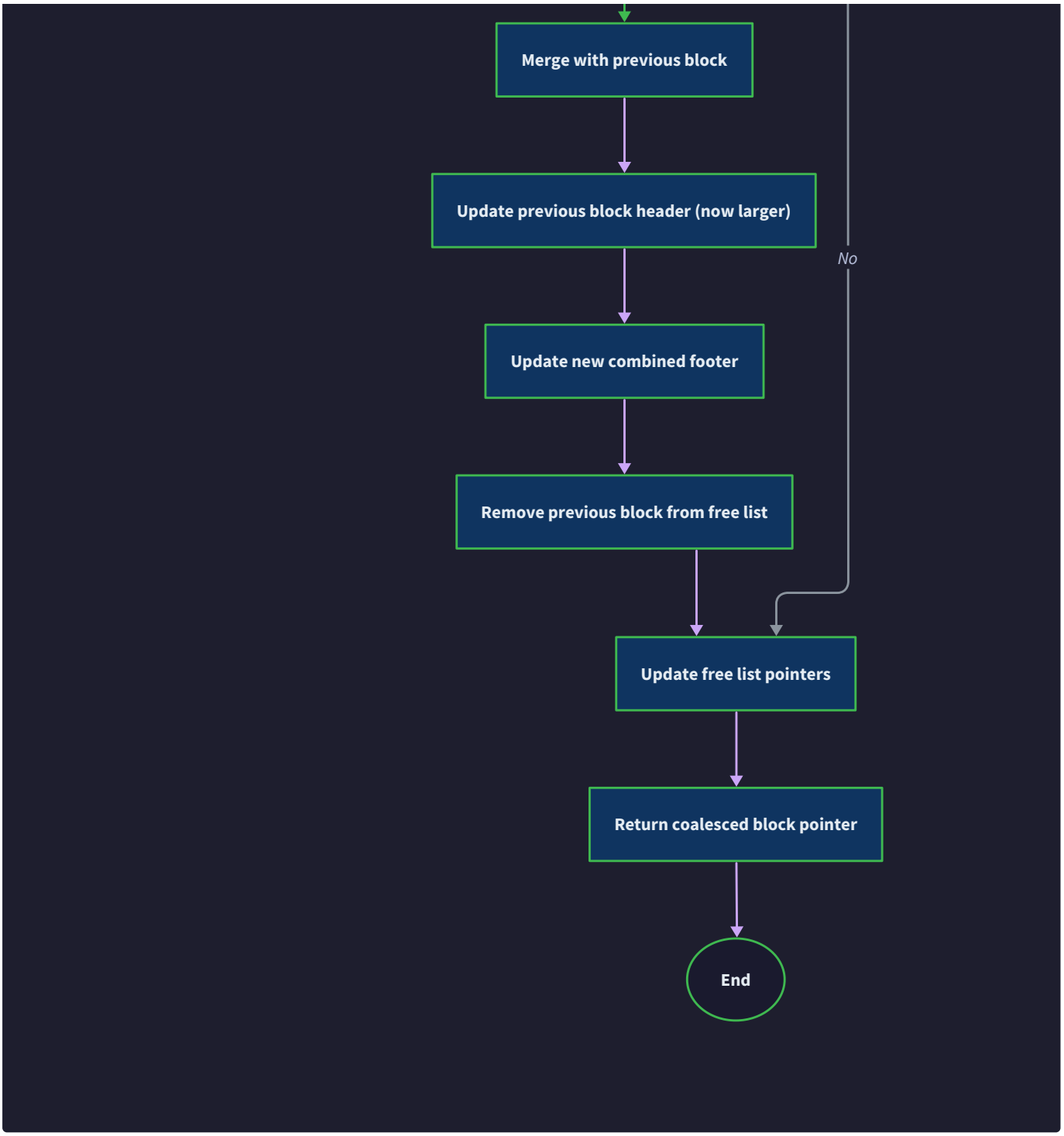
6. **Free List Insertion**
   After coalescing, the resulting free block (which could be the original block, a merged block with next, with previous, or both) is inserted into the appropriate free list. `malloc_segregated` determines the size class based on the block's total size. If the size exceeds `LARGEST_SIZE_CLASS` (2048 bytes), it's inserted into the general free list via `free_list_insert`. Otherwise, it goes into the corresponding segregated list.

7. **Heap Contraction Consideration**
   If the freed block is at the end of the heap (the "wilderness" block) and exceeds a contraction threshold (e.g., multiple pages), the allocator may call `sbrk` with a negative increment to return memory to the OS. This requires careful adjustment of heap boundaries and free list pointers.

8. **Lock Release and Return**
   Finally, the arena lock is released via `mutex_unlock(&arena->lock)`, and the `free` function returns.

The coalescing logic follows this decision flowchart:


Flowchart: Coalescing Adjacent Free Blocks
Start coalesce_block(block_ptr)
Is next block free?
Yes
No
Merge with next block
Update combined block header
Update combined block footer
Remove next block from free list
Is previous block free?
Yes



Coalescing Scenarios Table:

Scenario	Previous Block	Next Block	Action	Resulting Block Size
Isolated free	Allocated	Allocated	No coalescing	Original size (96)
Merge with next	Allocated	Free	Remove next, merge sizes	96 + next_size
Merge with previous	Free	Allocated	Remove previous, merge sizes	prev_size + 96
Merge both	Free	Free	Remove both, merge all three	prev_size + 96 + next_size
Wilderness block	Free	End of heap	Consider `sbrk` contraction	Possibly reduced heap

Data Structure State Changes During free(ptr):

Step	`block_header_t` Changes	`footer_t` Changes	Free List Changes
Mark free	`allocated = 0`	`allocated = 0` (at block end)	None yet
Coalesce next	`size += next->size`	Update at new end	Remove next block from list
Coalesce previous	`prev->size += size`	Update at new end	Remove previous block from list
Insert	Pointers updated if in free list	Unchanged	Block added to head of appropriate list

Critical Interaction: Boundary Tags Enable Bidirectional Coalescing
Without the footer, finding the previous block would require traversing the entire free list from the beginning—an O(n) operation. The footer provides O(1) access to the previous block's size and status, making coalescing efficient. This design trade-off (8-16 bytes overhead per block) significantly reduces external fragmentation.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Thread-local storage	`pthread_key_t` with destructor	Compiler TLS (e.g., `__thread` in GCC) with fallback
Locking	`pthread_mutex_t` with wrapper functions	Spinlock for very short critical sections with adaptive fallback
Size class mapping	Linear array search	Binary search or direct computation for power-of-two classes

Recommended File/Module Structure:

```
allocator/
├── include/
│   └── allocator.h          # Public API and data structures
├── src/
│   ├── allocator.c          # Top-level malloc/free implementations
│   ├── os_mem.c             # OS interface (sbrk, mmap wrappers)
│   ├── block.c              # Block management (split, coalesce)
│   ├── free_list.c          # Free list operations (insert, remove)
│   ├── strategies.c         # Allocation strategies (first-fit, etc.)
│   ├── segregated.c         # Segregated free list management
│   ├── thread_cache.c       # Thread-local caching implementation
│   └── arena.c              # Per-thread arena management
└── tests/
    ├── test_basic.c         # Milestone 1 tests
    ├── test_fragmentation.c # Milestone 2 tests
    └── test_threads.c       # Milestone 4 tests
```

Core Logic Skeleton Code:

```c
/**
 * Top-level malloc implementation with thread caching and segregated lists.
 * Follows the flow described in the "malloc(64)" narrative.
 */
void* malloc(size_t size) {
    // TODO 1: Handle zero-size request (return NULL or minimum allocation)

    // TODO 2: Calculate total block size including header, footer, alignment

    // TODO 3: Check if size exceeds SBRK_THRESHOLD - if yes, use os_mmap directly

    // TODO 4: Try thread cache first via thread_cache_malloc

    // TODO 5: If thread cache misses, call malloc_segregated with size

    // TODO 6: If malloc_segregated returns NULL, get memory from OS via get_memory_from_os

    // TODO 7: Initialize new block with init_block

    // TODO 8: Split if necessary (new block larger than requested)

    // TODO 9: Mark block as allocated, update any global statistics

    // TODO 10: Return user pointer (header + header_size, aligned)
    return NULL;
}

/**
 * Free implementation with coalescing and thread caching.
 * Follows the flow described in the "free(ptr)" narrative.
 */
void free(void* ptr) {
    // TODO 1: Check for NULL pointer (free(NULL) should be no-op)

    // TODO 2: Get block header from user pointer using get_header_from_ptr

    // TODO 3: Check IS_MMAP_FLAG - if set, use os_munmap and return
```

```
    // TODO 4: Try thread cache via thread_cache_free if appropriate

    // TODO 5: Otherwise, lock the arena mutex

    // TODO 6: Mark block as free (header->allocated = 0, update footer)

    // TODO 7: Coalesce with next block if free (check next header)

    // TODO 8: Coalesce with previous block if free (check previous footer)

    // TODO 9: Insert resulting free block into appropriate free list
    //         (segregated list if size <= LARGEST_SIZE_CLASS, else general list)

    // TODO 10: Check if block is at end of heap (wilderness) and consider
    //          contracting heap with sbrk if large enough

    // TODO 11: Unlock arena mutex
}

/**
 * Helper function for coalescing adjacent free blocks.
 * Returns pointer to the coalesced block (which may be the original block
 * or a previous block if merged backward).
 */
block_header_t* coalesce_block(block_header_t* block) {
    // TODO 1: Calculate address of next block using block->size

    // TODO 2: Check if next block is within heap bounds and free
    //         (verify using next->allocated flag and magic number)

    // TODO 3: If next is free, remove it from its free list,
    //         merge sizes, and update footer at new end

    // TODO 4: Calculate address of previous block's footer
    //         (address of current block - sizeof(footer_t))
```

```
    // TODO 5: Check if previous block exists and is free


    // TODO 6: If previous is free, remove it from its free list,
    //         merge sizes (previous size + current size),
    //         update footer at end of current block,
    //         return pointer to previous block header


    // TODO 7: If no coalescing occurred, return original block pointer
    return block;
}
```

Language-Specific Hints (C):

- Use `pthread_key_create` with a destructor function to clean up thread caches when threads exit
- Implement `get_header_from_ptr` as a macro for performance: `#define GET_HEADER(p) ((block_header_t*)((char*)(p) - sizeof(block_header_t)))`
- For alignment, use `(size + ALIGNMENT - 1) & ~(ALIGNMENT - 1)` to round up to multiple of ALIGNMENT
- When checking magic numbers for corruption detection, compare `header->magic == MAGIC` in debug builds only

Debugging Tips for Data Flow Issues:

Symptom	Likely Cause	How to Diagnose	Fix
malloc returns pointer that crashes on use	Incorrect alignment calculation	Print pointer value modulo ALIGNMENT, should be 0	Ensure `get_total_block_size` adds proper padding
free() causes double-free detection	Block already in free list after coalescing	Use `heap_visualize()` before/after free, check free list pointers	Ensure `free_list_remove` is called during coalescing before re-insertion
Thread cache seems ignored	`get_thread_cache()` returns NULL	Check pthread_key creation in `init_thread_cache_system()`	Initialize thread cache system before first malloc call
Coalescing not merging adjacent free blocks	Footer not updated after merge	Dump footer values before/after with `heap_visualize()`	Update footer at new block end during coalesce
Memory leak in multithreaded tests	Thread cache not flushed to global arena	Count blocks in thread cache vs global free lists	Implement periodic flush from thread cache to global arena

Milestone Checkpoint for Data Flow: After implementing the complete flow, run this test program to verify both allocation paths work correctly:

```c
#include "allocator.h"
#include <pthread.h>
#include <stdio.h>

void* thread_func(void* arg) {
    // Allocate within size class 64 (should use thread cache)
    void* p1 = malloc(64);
    void* p2 = malloc(64);
    free(p1);
    // p1 should now be in thread cache
    void* p3 = malloc(64);  // Should reuse p1 from cache
    free(p2);
    free(p3);
    return NULL;
}

int main() {
    // Initialize allocator
    init_allocator();

    // Test basic segregated allocation
    void* ptrs[10];
    for (int i = 0; i < 10; i++) {
        ptrs[i] = malloc(64 + i * 16);  // Varying sizes
        printf("Allocated %p size %zu\n", ptrs[i],
               ((block_header_t*)((char*)ptrs[i] - sizeof(block_header_t)))->size);
    }

    // Free every other block to create fragmentation
    for (int i = 0; i < 10; i += 2) {
        free(ptrs[i]);
    }

    // Allocate similar sizes - should coalesce and split appropriately
    void* p = malloc(128);
    printf("After coalescing: %p\n", p);
    free(p);
```

```
    // Test thread caching
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);
    pthread_join(thread, NULL);

    heap_visualize();
    return 0;
}
```

Expected behavior:

1. First allocations show varying block sizes
2. After freeing every other block, `heap_visualize()` should show alternating allocated/free blocks
3. The 128-byte allocation should trigger coalescing of adjacent free blocks
4. The thread function should demonstrate cache hits (add debug prints to verify)
5. No memory leaks should occur (verify with external tools like Valgrind)

Signs of problems:

- If `heap_visualize()` shows overlapping blocks, boundary tag calculations are wrong
- If thread cache allocations show global lock contention, thread cache isn't being used
- If coalescing doesn't occur, check footer values and the coalescing logic's boundary checks

---

7. Error Handling and Edge Cases

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

A robust memory allocator must gracefully handle not just the happy path of typical allocations, but also the myriad ways things can go wrong. This section details how our allocator responds to failures—both expected (out of memory) and unexpected (corrupted metadata)—and defines precise behavior for edge case requests that test the boundaries of the design. Proper error handling transforms a fragile prototype into a reliable **teaching allocator** that students can experiment with confidence.

7.1 Failure Modes and Recovery

Mental Model: The Warehouse Safety Protocols
Imagine our warehouse manager (the allocator) operates in an environment where supplies can run out (OS memory exhaustion), shelf labels can be damaged (corrupted metadata), and workers can miscommunicate (thread race conditions). Safety protocols must be in place to detect these issues, prevent catastrophic collapse, and provide clear error indications rather than silent corruption.

7.1.1 Operating System Memory Request Failures

When the allocator needs more memory from the operating system, both `sbrk` and `mmap` can fail. This typically occurs when the system has exhausted physical RAM and swap space, when address space fragmentation prevents contiguous expansion, or when resource limits (ulimit) are reached.

Failure Detection and Response:

Failure Mode	Detection Method	Recovery Action	User Impact
`sbrk` failure (returns `(void*)-1`)	Check return value of `os_sbrk` wrapper	Return `NULL` to caller; keep existing heap intact	`malloc` returns `NULL`, program must handle allocation failure
`mmap` failure (returns `MAP_FAILED`)	Check return value against `MAP_FAILED`	Return `NULL` to caller; no partial mapping created	Large allocation request fails gracefully
`munmap` failure	Check return value (should be 0)	Log warning if debugging enabled; cannot retry safely	Potential memory leak but process continues

> **Design Principle: Fail-Stop for OS Failures**
> When the operating system denies a memory request, the allocator cannot manufacture memory from nothing. The only safe response is to signal failure upward by returning `NULL`, matching the POSIX specification for `malloc`. The caller must then decide whether to abort, retry, or proceed with reduced functionality.

Implementation Strategy: The `get_memory_from_os` function must validate both system call returns. For `sbrk`, special care is needed because failure leaves the program break unchanged. The wrapper should also check for unreasonable arguments (negative increments) before calling the system.

7.1.2 Heap Metadata Corruption Detection

Metadata corruption represents a severe failure where the allocator's internal data structures—block headers, footers, or free list pointers—become invalid. This can result from:

- **Buffer overflows/underflows**: User programs writing beyond allocated bounds
- **Double-free**: Freeing the same pointer twice corrupts free list integrity
- **Wild pointers**: Freeing an address never returned by `malloc`
- **Thread race conditions**: Unsynchronized access in a non-thread-safe configuration

Corruption Detection Mechanisms:

Corruption Type	Detection Technique	Diagnostic Signal	Recommended Response
Invalid block size (not multiple of `ALIGNMENT`, size < `MIN_BLOCK_SIZE`)	Validate `size` field when traversing blocks or during `free`	Assertion failure or error log	Abort process with diagnostic message
Double-free (block already marked free)	Check `allocated` flag in header during `free`	Log error with pointer address	Abort process; continuing risks free list corruption
Invalid magic number (if enabled)	Compare `MAGIC` constant in extended header	Assertion failure	Abort with "heap corruption" message
Free list pointer inconsistency (`next->prev != current`)	Validate doubly-linked list invariants during traversal	Log inconsistency location	Attempt repair by removing corrupted block from list
Wild pointer (no valid header at `ptr - HEADER_SIZE`)	Check alignment and boundary before accessing header	Log invalid pointer address	Abort process; cannot safely proceed

ADR: Aggressive Detection vs. Silent Continuation

Decision: Abort on Corruption Detection in Debug Builds

- **Context**: In a production allocator, robustness might prioritize continuing despite corruption. In a **teaching allocator**, silent continuation masks bugs that students need to learn to fix.
- **Options Considered**:
  1. **Silent continuation with best-effort recovery**: Attempt to repair or isolate corruption, return `NULL` for affected operations.
  2. **Abort with detailed diagnostics**: Terminate the process immediately with clear error messages pinpointing the issue.
  3. **Continue with corruption logging**: Record errors but attempt to proceed, risking cascade failures.
- **Decision**: Option 2 for debug builds (when `DEBUG` is defined), Option 1 for release builds.
- **Rationale**: Students benefit from immediate, clear feedback when their code corrupts the heap. The abort provides a "fail-fast" learning experience. Production code might need to continue, so we provide a compile-time switch.
- **Consequences**: Debug builds may terminate student programs abruptly, but with informative messages. Requires careful error messages that explain both what was detected and likely causes.

Comparison Table:

Option	Pros	Cons	Selected
Abort with diagnostics	Immediate bug surfacing; clear learning feedback; prevents cascade failures	Harsh for production; loses application state	**Yes (debug)**
Silent continuation	Maximizes uptime; matches some production allocators	Masks bugs; harder to debug; risk of security vulnerabilities	**Yes (release)**
Log and continue	Records issues for later analysis; continues execution	Performance overhead; still masks immediate bugs	No

7.1.3 Memory Exhaustion Strategies

When the heap becomes full—either because all blocks are allocated or because external fragmentation leaves no contiguous space large enough—the allocator must attempt remediation before giving up.

Memory Exhaustion Recovery Sequence:

1. **Free List Coalescing**: Before requesting more memory from the OS, attempt to merge all adjacent free blocks. This is an O(n) operation but can recover significant contiguous space.
2. **Free List Defragmentation** (advanced): If coalescing fails, consider migrating allocated blocks to compact free space. This requires updating all user pointers and is complex, so we omit it from our teaching allocator.
3. **OS Memory Request**: If no suitable block exists even after coalescing, request more memory via `get_memory_from_os`. For requests under `SBRK_THRESHOLD`, this extends the heap with `sbrk`. For larger requests, use `mmap`.
4. **Fallback NULL Return**: If OS memory requests fail, return `NULL` to the caller.

Special Case: The Wilderness Block The block at the end of the `sbrk` heap (the **wilderness block**) has special properties: it can be expanded by requesting more memory from the OS. When allocation fails and the last block in the heap is free, we can attempt to expand it rather than requesting a completely new block.

7.2 Edge Cases: Zero, One, and Huge

Mental Model: The Warehouse's Rulebook for Special Orders
Every warehouse has policies for unusual orders: ordering zero items, requesting microscopic storage, or asking for the entire building. Our allocator needs explicit rules for these boundary cases to ensure consistent, predictable behavior.

7.2.1 Zero-Size Allocations (`malloc(0)`)

The C standard leaves `malloc(0)` implementation-defined: it may return either `NULL` or a unique pointer that cannot be dereferenced. Our design must choose one behavior consistently.

Decision Table for Zero-Size Requests:

Operation	Request	Our Implementation	Rationale
`malloc(0)`	Size = 0	Return `NULL`	Simplifies implementation; avoids allocating metadata for unusable memory; matches some standard library implementations
`calloc(0, n)` or `calloc(n, 0)`	Total size = 0	Return `NULL`	Consistent with `malloc(0)`
`realloc(ptr, 0)`	New size = 0	Equivalent to `free(ptr)`; return `NULL`	POSIX specifies this behavior; avoids zero-byte allocation
`free(NULL)`	Pointer = `NULL`	No-op (safe)	Required by C standard

> **Implementation Note**: Zero-size requests must be handled early in `malloc`, before alignment calculations. The minimum allocation size after adding header/footer and alignment padding would be non-zero anyway, so returning `NULL` avoids creating useless tiny blocks.

7.2.2 Extremely Small Allocations (`malloc(1)`)

Requests for 1-7 bytes on a 64-bit system still require 8-byte alignment and include header overhead. This leads to **internal fragmentation**—the difference between requested size and allocated size.

Minimum Allocation Size Calculation:

- Header: `sizeof(block_header_t)` (assume 24 bytes: `size_t` + `int` + two pointers)
- Footer: `sizeof(footer_t)` (12 bytes: `size_t` + `int`)
- Minimum payload: 1 byte
- Alignment padding: up to `ALIGNMENT-1` bytes (7 bytes)
- **Total**: ~44+ bytes for a 1-byte request

Behavior Specification:

- `malloc(1)` returns an 8-byte aligned pointer
- The usable space is exactly 1 byte, but the block consumes ~44 bytes
- `free` must handle these tiny blocks correctly
- Segregated size classes help amortize this overhead for common small sizes

7.2.3 Extremely Large Allocations

Requests approaching or exceeding address space limits require special handling:

Request Size	Behavior	Rationale
Size > `SIZE_MAX - HEADER_SIZE - ALIGNMENT` (overflow)	Return `NULL`	Request size causes arithmetic overflow when adding metadata
Size >= `SBRK_THRESHOLD`	Use `mmap` with `MAP_PRIVATE | MAP_ANONYMOUS`	Avoids fragmenting the main heap; allows independent return to OS
Size > available address space	`mmap` fails, return `NULL`	Let OS enforce virtual memory limits
Size = `SIZE_MAX` (maximum `size_t`)	Return `NULL` (overflow in size calculation)	Defensive programming against integer overflow

Integer Overflow Protection: The `get_total_block_size` function must guard against overflow when adding header size, footer size, alignment padding, and requested size. Use checked arithmetic or compare against `SIZE_MAX - (HEADER_SIZE + FOOTER_SIZE + ALIGNMENT)`.

Example Overflow Detection Logic:

```c
if (user_size > SIZE_MAX - (HEADER_SIZE + FOOTER_SIZE + ALIGNMENT)) {
    return 0;  // Signal overflow
}
```

7.2.4 Alignment Edge Cases

The allocator guarantees `ALIGNMENT`-byte aligned addresses (8 bytes on 64-bit systems). This introduces edge cases:

1. **Misaligned `free` pointers**: What if a user calls `free` with a pointer not returned by `malloc`, or offset within a block?
   - Detection: Check if `(ptr - HEADER_SIZE)` is aligned to `ALIGNMENT`
   - Response: Abort with "invalid pointer" message in debug builds
2. **Over-alignment requests**: Some programs need greater alignment (16, 32, 64 bytes) for SIMD or cache lines.
   - Our implementation only guarantees `ALIGNMENT` (8 bytes)
   - Future extension could support `aligned_alloc`
3. **Block splitting alignment**: When splitting a block, both resulting blocks must maintain alignment for headers and user data.

7.2.5 Thread-Related Edge Cases

With thread safety enabled (Milestone 4), additional edge cases emerge:

Scenario	Behavior	Rationale
`malloc` / `free` called before allocator initialization	Lazy initialization on first call	Avoid requiring explicit `init` function
Thread exits with blocks in thread-local cache	Move cached blocks to global arena via destructor	Prevent memory leaks from per-thread caches
Multiple threads concurrently corrupt heap	Detection challenging; best effort with mutex protection	Use mutexes to serialize access; corruption still possible but less likely
Signal handler calls `malloc` (reentrancy)	Deadlock risk if signal interrupts locked `malloc`	Not supported; signal handlers should not call allocator

7.2.6 `calloc` and `realloc` Edge Cases

These wrapper functions have their own edge cases:

`calloc(nmemb, size)`:

- Multiplication overflow: `nmemb * size > SIZE_MAX`
- Zero initialization of all bytes (including padding between header and user data)
- Handling of zero elements or zero size (returns `NULL`)

`realloc(ptr, size)`:

- `ptr` is `NULL`: behaves as `malloc(size)`
- `size` is 0: behaves as `free(ptr)`, returns `NULL`
- `ptr` not from `malloc`: undefined, but we can attempt detection
- In-place expansion possible if next block is free and large enough

Common Pitfalls

⚠️ Pitfall: Ignoring `sbrk` Failure

- **Mistake**: Assuming `sbrk` always succeeds, proceeding as if new memory is available.
- **Why Wrong**: On memory exhaustion, `sbrk` returns `(void*)-1`. Using this as a valid address causes segmentation faults.
- **Fix**: Check return value, return `NULL` to caller, preserve existing heap state.

⚠️ Pitfall: Integer Overflow in Size Calculations

- **Mistake**: Directly adding header size to user size without overflow check.
- **Why Wrong**: `user_size + HEADER_SIZE` can wrap around to a small number, causing allocation of tiny block that overflows when used.
- **Fix**: Use guarded addition or compare against `SIZE_MAX - HEADER_SIZE` before adding.

⚠️ Pitfall: Handling `free(NULL)` Incorrectly

- **Mistake**: Attempting to access metadata at NULL pointer.
- **Why Wrong**: C standard requires `free(NULL)` be a no-op; dereferencing NULL crashes.
- **Fix**: Check for NULL pointer at the very beginning of `free`.

⚠️ Pitfall: Not Validating `free` Pointer

- **Mistake**: Blindly calculating `header = ptr - HEADER_SIZE` without verifying `ptr` came from `malloc`.
- **Why Wrong**: Invalid pointers corrupt heap metadata; malicious or buggy programs can crash allocator.
- **Fix**: Check alignment, verify magic number (if enabled), ensure pointer lies within heap bounds.

⚠️ Pitfall: Forgetting Thread-Local Cache Cleanup

- **Mistake**: Threads exit with blocks in their local cache, leaking memory.
- **Why Wrong**: OS reclaims thread stack but not heap memory; cached blocks become orphaned.
- **Fix**: Register thread-local destructor that returns cached blocks to global arena.

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Overflow Detection	Manual bounds checks before arithmetic	Use compiler builtins (`__builtin_add_overflow`)
Corruption Detection	Magic numbers in headers	Cryptographic canaries or memory protection keys
Thread Safety	Single global mutex	Fine-grained locking with per-arena mutexes
Debug Output	`fprintf(stderr, ...)`	Custom logging callback with levels

File Structure for Error Handling:

```
allocator/
├── include/
│   └── allocator.h          ← Error codes and debugging flags
├── src/
│   ├── error.c              ← Error reporting and validation functions
│   ├── debug.c              ← Heap visualization and integrity checking
│   ├── allocator.c          ← Main malloc/free with error checks
│   └── os_mem.c             ← OS wrapper with error handling
```

Core Validation Functions (Complete Implementation):

```c
// error.c - Complete error handling implementation

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include "allocator.h"


#ifdef DEBUG

#define ASSERT(cond, msg) \

    do { \

        if (!(cond)) { \

            fprintf(stderr, "Assertion failed: %s (%s:%d)\n", \

                    msg, __FILE__, __LINE__); \

            abort(); \

        } \

    } while(0)

#else

#define ASSERT(cond, msg) ((void)0)

#endif


// Check if pointer appears to be a valid malloc pointer

int is_valid_pointer(void *ptr) {

    if (ptr == NULL) return 1;  // free(NULL) is valid



    // Check alignment

    if ((uintptr_t)ptr % ALIGNMENT != 0) {

        DEBUG_LOG("Pointer %p not aligned to %d\n", ptr, ALIGNMENT);

        return 0;

    }



    block_header_t *header = (block_header_t*)((char*)ptr - HEADER_SIZE);



    // Check if header is within heap bounds (simplified check)

    // In full implementation, verify against heap_base and heap_limit



    // Check magic number if enabled

    #ifdef USE_MAGIC

    if (header->magic != MAGIC) {
```

```
        DEBUG_LOG("Bad magic at %p: %x\n", header, header->magic);
        return 0;
    }
    #endif

    // Check size seems reasonable (non-zero, not enormous)
    if (header->size == 0 || header->size > (SIZE_MAX / 2)) {
        DEBUG_LOG("Suspicious size %zu at %p\n", header->size, header);
        return 0;
    }

    return 1;
}

// Check for double-free
int is_double_free(block_header_t *header) {
    if (!header->allocated) {
        DEBUG_LOG("Double free detected at %p\n",
                  (char*)header + HEADER_SIZE);
        return 1;
    }
    return 0;
}

// Safe size calculation with overflow check
size_t calculate_total_size(size_t user_size, int *overflow) {
    *overflow = 0;

    // Check for overflow before any arithmetic
    if (user_size > SIZE_MAX - (HEADER_SIZE + FOOTER_SIZE + ALIGNMENT)) {
        *overflow = 1;
        return 0;
    }

    // Calculate minimum block size
    size_t total = HEADER_SIZE + FOOTER_SIZE + user_size;
```

```
    // Add alignment padding for user payload
    size_t misalignment = total % ALIGNMENT;
    if (misalignment > 0) {
        if (total > SIZE_MAX - (ALIGNMENT - misalignment)) {
            *overflow = 1;
            return 0;
        }
        total += ALIGNMENT - misalignment;
    }

    // Ensure minimum block size for free blocks
    if (total < MIN_BLOCK_SIZE) {
        total = MIN_BLOCK_SIZE;
    }

    return total;
}
```

Main malloc with Error Handling (Skeleton):

```c
// allocator.c - malloc with comprehensive error handling

void *malloc(size_t size) {
    // TODO 1: Handle zero-size request (return NULL per our specification)

    // TODO 2: Check for size overflow using calculate_total_size
    //         If overflow detected, set errno to ENOMEM and return NULL

    // TODO 3: For debug builds, validate heap integrity before proceeding
    //         Call heap_validate() if DEBUG defined

    // TODO 4: If size >= SBRK_THRESHOLD, use mmap path
    //         Call os_mmap and check for MAP_FAILED
    //         If failed, return NULL

    // TODO 5: For normal allocations, ensure thread safety
    //         If thread safety enabled, lock appropriate mutex
    //         Use mutex_lock with error checking

    // TODO 6: Find or create free block
    //         If no block found and need OS memory, call get_memory_from_os
    //         Check for NULL return (OS memory exhaustion)

    // TODO 7: Split block if appropriate (remaining part >= MIN_BLOCK_SIZE)

    // TODO 8: Mark block as allocated, update free list if necessary

    // TODO 9: Unlock mutex if thread safety enabled

    // TODO 10: Return pointer to user payload
    //          Pointer should be (block_address + HEADER_SIZE)

    // TODO 11: For debug builds, fill allocated memory with pattern (0xAA)
    //          to help detect use of uninitialized memory

    return NULL; // Placeholder
}
```

free with Error Handling (Skeleton):

```c
void free(void *ptr) {
    // TODO 1: Handle free(NULL) as no-op (required by C standard)

    // TODO 2: For debug builds, validate pointer using is_valid_pointer
    //         If invalid, abort with error message in debug mode

    // TODO 3: Calculate block header: header = ptr - HEADER_SIZE

    // TODO 4: Check for double-free using is_double_free
    //         In debug builds, abort on double-free

    // TODO 5: If thread safety enabled, lock appropriate mutex
    //         Thread-local cache may not need global lock

    // TODO 6: Mark block as free
    //         For mmap blocks, check IS_MMAP_FLAG and use munmap directly

    // TODO 7: Attempt coalescing with adjacent free blocks
    //         Use coalesce_block which handles boundary tags

    // TODO 8: Insert block into appropriate free list
    //         For segregated lists, use size_to_class_index

    // TODO 9: For thread-local caches, consider caching freed block
    //         instead of immediately merging

    // TODO 10: Unlock mutex if thread safety enabled

    // TODO 11: For debug builds, fill freed memory with pattern (0xDD)
    //          to help detect use-after-free errors
}
```

Language-Specific Hints for C:

1. **Integer Overflow**: Use `SIZE_MAX` from `stdint.h` for bounds checking rather than assuming maximum values.
2. **Thread Safety**: Initialize mutexes with `pthread_mutex_init` and destroy with `pthread_mutex_destroy`. Use `pthread_cleanup_push` for cleanup in error paths.
3. **Debugging Aids**: Consider using `#ifdef DEBUG` blocks to include expensive validation that can be compiled out in release builds.

4. **errno Setting**: Set `errno = ENOMEM` when returning `NULL` due to memory exhaustion, matching POSIX convention.
5. **Signal Safety**: Avoid using `malloc` in signal handlers; if absolutely necessary, use `async-signal-safe` functions only.

Milestone Checkpoint for Error Handling:

After implementing basic error handling, test with this program:

```c
// test_errors.c

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>


int main() {

    // Test 1: malloc(0) should return NULL

    void *p1 = malloc(0);

    printf("malloc(0) = %p (expected: NULL or unique pointer)\n", p1);



    // Test 2: Very large allocation should fail gracefully

    void *p2 = malloc((size_t)-1);  // Nearly SIZE_MAX

    if (p2 == NULL && errno == ENOMEM) {

        printf("Large allocation correctly failed with ENOMEM\n");

    }



    // Test 3: free(NULL) should not crash

    free(NULL);

    printf("free(NULL) succeeded\n");



    // Test 4: Double-free detection (may abort in debug)

    void *p3 = malloc(100);

    free(p3);

    printf("First free succeeded\n");

    // Uncomment to test double-free: free(p3);



    // Test 5: Memory exhaustion simulation

    // This may take a while and use swap

    printf("Attempting to exhaust memory...\n");

    size_t count = 0;

    while (1) {

        void *p = malloc(1024*1024);  // 1MB chunks

        if (p == NULL) {

            printf("malloc failed after %zu allocations (expected)\n", count);

            break;

        }

        count++;
```

```
    }

    return 0;
}
```

Expected Behavior:

- Program runs without crashing (unless double-free is enabled in debug)
- Large allocation fails with `NULL` return
- `free(NULL)` is silent
- Memory exhaustion eventually occurs (after many allocations)
- No memory leaks (verify with Valgrind)

Debugging Tips Table:

Symptom	Likely Cause	How to Diagnose	Fix
Segfault in `free` with valid pointer	Corrupted header before free	Add magic number to header; check before free	Ensure no buffer overflows from user code
`malloc` returns unaligned pointer	Missing alignment padding	Check `(uintptr_t)ptr % ALIGNMENT == 0`	Fix `get_total_block_size` alignment calculation
Memory leak in repeated alloc/free cycles	Freed block not added to free list	Use `heap_visualize` to see block states	Ensure `free_list_insert` is called in `free`
Heap corruption after multithreaded use	Race condition without proper locking	Add thread ID logging to operations	Implement mutex locks around shared structures
`malloc` succeeds but write causes segfault	Block size smaller than requested	Check `get_total_block_size` calculation	Ensure size includes header, footer, alignment
Double-free not detected	`allocated` flag not cleared on free	Add debug print of flag state	Clear flag and add magic number pattern

8. Testing Strategy

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

A robust testing strategy is crucial for building a memory allocator—a system where bugs manifest as subtle corruption, leaks, or crashes long after the actual programming error. This section provides a systematic approach to verifying correctness, performance, and robustness across all implementation milestones. Think of testing your allocator as **stress-testing a warehouse management system**: you need to verify it handles individual requests correctly (unit tests), sequences of operations (integration tests), heavy loads (stress tests), and multiple workers simultaneously (concurrent tests) without losing track of inventory or collapsing under pressure.

Test Categories and Properties

Testing a memory allocator requires verifying both functional correctness (does it return valid memory?) and structural integrity (does the internal heap state remain consistent?). We organize tests into four progressively challenging categories, each with specific properties to verify.

Unit Tests: The Individual Shelf Operation These tests isolate single allocator functions with controlled, minimal scenarios. They verify that each core operation behaves correctly in isolation before being composed into complex sequences.

Test Category	Focus Area	Example Test	Properties to Verify
Single Allocation	Basic `malloc` functionality	`malloc(64)` followed by immediate check	Returns non-NULL pointer, pointer is aligned to `ALIGNMENT`, returned memory can be written to and read from
Single Free	Basic `free` functionality	`malloc(32)` then `free(ptr)`	Block is marked free in metadata, subsequent allocation can reuse the memory
Metadata Integrity	Header/footer correctness	Allocate block, inspect `block_header_t` via `get_header_from_ptr`	`size` field matches requested size (plus padding), `allocated` flag is set correctly, magic number (if enabled) is valid
Boundary Conditions	Edge case handling	`malloc(0)`, `malloc(MAX_SIZE)`, `malloc(SBRK_THRESHOLD+1)`	Zero-size returns NULL or unique pointer, huge sizes fail gracefully (NULL), large allocations trigger `mmap` path
OS Interface	System call wrappers	Mock `sbrk` failure, test `get_memory_from_os`	`os_sbrk` returns `(void*)-1` on failure, `get_memory_from_os` returns NULL when OS denies memory

Integration Tests: The Warehouse Workflow These tests exercise sequences of allocations and frees that mimic real usage patterns, verifying that the allocator maintains consistency across operations.

Test Pattern	Description	Properties to Verify
Allocate-Free-Reallocate	Simple reuse: A = malloc(100), free(A), B = malloc(100)	B reuses A's memory (pointers may differ but same heap region), metadata updated correctly
Fragmentation Sequences	Allocate multiple blocks, free alternating ones, then allocate different sizes	Free list remains consistent, external fragmentation doesn't prevent allocation (coalescing works)
Size Class Promotion	Allocate sizes that cross class boundaries (e.g., 60, 68, 132 bytes)	Blocks are placed in correct segregated lists, splitting creates appropriate remainder blocks
Heap Expansion	Allocate until sbrk is called multiple times	Heap grows contiguously, wilderness block is properly maintained, program break increases

Fragmentation Tests: Measuring Warehouse Space Efficiency These tests quantify internal and external fragmentation under various allocation patterns, validating that our strategies (splitting, coalescing, segregated lists) actually reduce waste.

Test Metric	Measurement Method	Target Outcome
Internal Fragmentation	Sum `(block_size - requested_size)` for all allocated blocks	Should be low (<20%) for segregated lists with appropriate size classes
External Fragmentation	Attempt to allocate a large block after many small allocations/frees	Should succeed due to coalescing; measure largest available contiguous free space
Free List Distribution	Count free blocks per size class after mixed workload	Segregated lists show concentration in common size classes; single free list shows scattered distribution

Concurrent Stress Tests: Multiple Warehouse Workers For thread-safe allocators, these tests verify correctness and performance under concurrent access, detecting race conditions, deadlocks, and scalability issues.

Test Scenario	Concurrency Pattern	Properties to Verify
Parallel Allocation	Multiple threads repeatedly call `malloc` with random sizes	No crashes, each thread gets unique memory, total allocated memory tracked correctly
Parallel Free	Threads allocate, then other threads free those pointers	No double-free corruption, free list remains consistent, coalescing works correctly
Mixed Workload	Threads perform random `malloc` / `free` operations concurrently	No memory leaks (total allocated memory stabilizes), no corruption after millions of operations
Thread Cache Efficiency	Measure cache hit rates under thread-local allocation patterns	Thread cache reduces lock contention, `hit_count` increases for repeated same-thread allocations

> **Key Testing Insight:** The most insidious allocator bugs—corrupted metadata, missed coalescing, race conditions—often don't cause immediate crashes. They manifest as *gradual* memory leaks, *occasional* corruption under heavy load, or *performance degradation*. Therefore, tests must not only check for immediate correctness but also run long enough to expose these latent issues.

Verification Properties Checklist

For any test, verify these fundamental invariants (properties that must always hold):

1. **Pointer Alignment:** Every pointer returned by `malloc` is aligned to at least `ALIGNMENT` bytes.
2. **No Overlap:** Allocated blocks never overlap in memory (each `ptr` to `ptr+size-1` is exclusive).
3. **Metadata Consistency:** For every block, header and footer (if present) contain matching `size` and `allocated` fields.
4. **Free List Integrity:** The free list is a valid doubly-linked list; each free block's `next` / `prev` pointers are consistent.
5. **Wilderness Invariant:** The last block in the heap (adjacent to program break) is either free (wilderness block) or allocated but expandable.
6. **Size Class Correctness:** In segregated lists, each free block's size matches its class's range.
7. **Thread Safety:** Concurrent operations produce the same results as sequential execution (linearizability).
8. **Memory Exhaustion Handling:** When system memory is exhausted, `malloc` returns `NULL` without corrupting heap state.
9. **Leak Freedom:** After all allocated blocks are freed, the free list contains all heap memory (except wilderness expansion).
10. **Coalescing Completeness:** No two adjacent free blocks remain unmerged in the free list.

Milestone Implementation Checkpoints

Each milestone builds upon the previous one. These checkpoints provide simple, concrete test programs you can run after completing each milestone to verify you're on the right track. The programs are described in prose—you'll translate them to actual C code in your test suite.

Milestone 1: Basic Allocator with sbrk

> **What you've built:** A bump allocator with block headers, alignment, and simple reuse via a free list.

Checkpoint Test: Basic Allocation and Reuse

1. Write a test program that:
   - Calls `malloc(32)` and stores the pointer `p1`.
   - Verifies `p1` is not `NULL`.
   - Writes a distinct pattern (like `0xAA`) to all 32 bytes.
   - Calls `free(p1)`.
   - Calls `malloc(32)` again for pointer `p2`.
   - Verifies `p2` is not `NULL`.
2. **Expected behavior:**
   - The program runs without segmentation faults.
   - `p2` should be equal to `p1` (same address) because the freed block is reused.
   - The 32 bytes at `p2` may contain the previous pattern (`0xAA`) or zeroes—both are acceptable since `malloc` doesn't guarantee zeroing.
3. **What to verify manually:**

- Use `heap_visualize()` (if implemented) to see a single free block after the first `free`, then an allocated block after the second `malloc`.
- Check alignment: `(uintptr_t)p1 % ALIGNMENT == 0`.

Common symptom → issue:

- ❌ **Segmentation fault when writing to `p1`** → Likely `malloc` returned an unaligned pointer that violates hardware restrictions, or block header calculation is off.
- ❌ `p2` **is different from `p1`** → The freed block wasn't added to the free list, or `find_free_block` isn't scanning the free list correctly.
- ❌ **Program hangs or crashes in `free`** → The `block_header_t` metadata is corrupted, possibly due to incorrect pointer arithmetic in `get_header_from_ptr`.

Milestone 2: Free List Management with Advanced Strategies

> **What you've built:** Explicit free list with splitting, coalescing, and multiple allocation strategies (first-fit, best-fit, worst-fit).

Checkpoint Test: Coalescing and Splitting

1. Write a test program that:
   - Allocates three adjacent blocks: `p1 = malloc(64)`, `p2 = malloc(64)`, `p3 = malloc(64)`.
   - Frees `p2` (middle block).
   - Uses `heap_visualize()` to confirm a single free block of 64 bytes exists between two allocated blocks.
   - Frees `p1` (now adjacent to the free block).
   - Verifies via visualization that the first two blocks have coalesced into one free block of 128+ bytes (including headers/footers).
   - Frees `p3` and verifies all three blocks merge into one large free block.
2. **Expected behavior:**
   - After freeing `p2`, the free list contains one 64-byte free block.
   - After freeing `p1`, the free list contains one ~128-byte free block (not two separate 64-byte blocks).
   - After freeing `p3`, the free list contains one ~192-byte free block spanning the entire allocated region.
3. **What to verify manually:**
   - Enable boundary tags (footers) and verify they match headers.
   - Test with different allocation strategies: first-fit should still coalesce correctly.
   - Verify splitting: allocate a 192-byte block, then `malloc(32)` should split it if `MIN_BLOCK_SIZE` allows.

Checkpoint Test: Allocation Strategy Differences

1. Write a test that:
   - Creates a free list with blocks of sizes 50, 100, and 200 bytes (by allocating and freeing).
   - Requests `malloc(80)` with each strategy.
2. **Expected behavior:**
   - **First-fit:** Takes the 100-byte block (first sufficient block).
   - **Best-fit:** Takes the 100-byte block (tightest fit).
   - **Worst-fit:** Takes the 200-byte block (largest block).
3. **What to verify manually:**
   - The chosen block's address differs per strategy.
   - The remainder after splitting differs in size.

Milestone 3: Segregated Free Lists

> **What you've built:** Multiple free lists binned by size class for O(1) small allocations.

Checkpoint Test: Size Class Routing

1. Write a test program that:
   - Requests allocations of sizes 16, 32, 64, 128, and 256 bytes.
   - For each, verify the returned pointer is not `NULL`.

- Free all blocks.
- Verify via internal statistics that each block was placed in the correct segregated list (e.g., a 32-byte request goes to the 32-byte size class list).

2. **Expected behavior:**
   - Each allocation returns a valid pointer.
   - After freeing, the segregated free lists contain the blocks in their respective class lists.
   - Requesting another 32-byte block immediately reuses the freed block from the 32-byte list.
3. **What to verify manually:**
   - Request a 60-byte allocation (should round up to 64-byte class).
   - Request a 129-byte allocation (should round up to 256-byte class or use fallback to general free list if beyond `LARGEST_SIZE_CLASS`).
   - Verify that splitting a larger class block (e.g., 256) for a smaller request (e.g., 32) creates a remainder that goes to the appropriate smaller class list.

Milestone 4: Thread Safety & mmap

> **What you've built:** Thread-safe allocator with per-thread caches, arenas, and mmap for large allocations.

Checkpoint Test: Thread Safety Under Contention

1. Write a test program that:
   - Creates two threads.
   - Each thread performs 1000 iterations: `malloc(random(1,128))`, store pointer in array, later `free` all its pointers.
   - Main thread joins both threads and verifies no crashes occurred.
   - Optionally, enable internal logging to see arena creation and thread cache activity.
2. **Expected behavior:**
   - Program runs without data races, deadlocks, or corruption.
   - Total memory usage across threads doesn't exceed reasonable bounds.
   - Each thread predominantly uses its own arena or thread cache (visible via debug logs).
3. **What to verify manually:**
   - Use `heap_validate()` after all threads finish to ensure heap integrity.
   - Test with `malloc(SBRK_THRESHOLD+1)` from multiple threads; verify it uses `mmap` and doesn't corrupt the main heap.

Checkpoint Test: Large Allocation Isolation

1. Write a test that:
   - Allocates a large block via `malloc(SBRK_THRESHOLD + 4096)`.
   - Verifies the pointer is not `NULL`.
   - Checks that the block's header has the `IS_MMAP_FLAG` set (or separate metadata indicating mmap origin).
   - Frees the block and verifies memory is returned to OS (optional: check `/proc/self/maps` on Linux).
2. **Expected behavior:**
   - Large allocation succeeds without touching the main sbrk heap.
   - Freeing the block calls `os_munmap`, reducing process memory footprint.

Implementation Guidance

Testing a memory allocator requires building a test harness that can exercise the allocator's functions, validate internal state, and measure performance. Below is guidance on implementing such a test suite.

A. Technology Recommendations Table

Component	Simple Option (For Learning)	Advanced Option (For Rigorous Testing)
Test Framework	Simple C test macros (custom `ASSERT`)	CMake + CTest with GoogleTest or Check
Memory Checking	Manual `heap_validate()` calls	Integration with Valgrind, AddressSanitizer
Concurrency Testing	Simple pthreads loops	Hypothesis-style random sequence testing
Performance Profiling	Manual timing with `clock_gettime`	`perf` tool, statistical analysis
Visualization	Custom `heap_visualize()` text output	Graphviz integration for heap diagrams

B. Recommended File/Module Structure

```
allocator-project/
├── src/
│   ├── allocator.c          # Main allocator implementation
│   ├── allocator.h          # Public API and data structures
│   ├── os_mem.c             # OS interface (sbrk, mmap wrappers)
│   ├── free_list.c          # Free list management
│   ├── strategies.c         # Allocation strategies
│   ├── thread_cache.c       # Thread-local caching
│   └── debug.c              # Debug utilities (heap_visualize, validate)
└── tests/
    ├── test_harness.h       # Simple test macros
    ├── test_basic.c         # Milestone 1 tests
    ├── test_free_list.c     # Milestone 2 tests
    ├── test_segregated.c    # Milestone 3 tests
    ├── test_concurrent.c    # Milestone 4 tests
    ├── test_fragmentation.c # Fragmentation measurements
    └── run_tests.sh         # Script to run all tests
```

C. Infrastructure Starter Code: Test Harness Macros

Create `tests/test_harness.h` with simple, complete macros for writing tests:

```c
#ifndef TEST_HARNESS_H

#define TEST_HARNESS_H


#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define ASSERT(cond, msg) \

    do { \

        if (!(cond)) { \

            fprintf(stderr, "FAIL: %s:%d: %s\n", __FILE__, __LINE__, msg); \

            exit(1); \

        } \

    } while (0)


#define ASSERT_EQ(a, b, msg) \

    ASSERT((a) == (b), msg)


#define ASSERT_NE(a, b, msg) \

    ASSERT((a) != (b), msg)


#define ASSERT_NULL(ptr, msg) \

    ASSERT((ptr) == NULL, msg)


#define ASSERT_NOT_NULL(ptr, msg) \

    ASSERT((ptr) != NULL, msg)


#define TEST_START(name) \

    printf("Running test: %s\n", name)


#define TEST_END(name) \

    printf("PASS: %s\n\n", name)


// Helper to check pointer alignment

#define ASSERT_ALIGNED(ptr, alignment) \

    ASSERT(((uintptr_t)(ptr) % (alignment)) == 0, \

           "Pointer not aligned")


#endif // TEST_HARNESS_H
```

D. Core Logic Skeleton Code: Test for Coalescing

Here's a skeleton for a test that verifies coalescing works correctly (Milestone 2):

```c
// In tests/test_free_list.c

#include "allocator.h"

#include "test_harness.h"


void test_coalescing_basic() {

    TEST_START("test_coalescing_basic");



    // TODO 1: Allocate three adjacent blocks of equal size

    void *p1 = malloc(64);

    void *p2 = malloc(64);

    void *p3 = malloc(64);



    ASSERT_NOT_NULL(p1, "First allocation failed");

    ASSERT_NOT_NULL(p2, "Second allocation failed");

    ASSERT_NOT_NULL(p3, "Third allocation failed");



    // TODO 2: Verify they are adjacent (optional, requires internal access)

    // block_header_t *h1 = get_header_from_ptr(p1);

    // block_header_t *h2 = get_header_from_ptr(p2);

    // ASSERT_EQ((char*)h1 + get_total_block_size(64), (char*)h2,

    //           "Blocks not adjacent");



    // TODO 3: Free the middle block, then the first block

    free(p2);

    free(p1);



    // TODO 4: Verify coalescing occurred

    // Method 1: Use heap_visualize() and parse output (simpler)

    // Method 2: Internal check: scan free list for block of size >= 128+headers



    // TODO 5: Allocate a block of size 128 - should reuse coalesced block

    void *p_large = malloc(128);

    ASSERT_NOT_NULL(p_large, "Allocation after coalescing failed");



    // TODO 6: Clean up remaining allocations

    free(p3);

    free(p_large);
```

```
    TEST_END("test_coalescing_basic");
}
```

E. Language-Specific Hints (C)

- **Use `mmap` and `munmap` for isolation:** When testing large allocations, consider using `MAP_ANONYMOUS | MAP_PRIVATE` with `PROT_READ | PROT_WRITE`.
- **Thread-local storage:** Use `pthread_key_create` and `pthread_getspecific` for per-thread caches rather than `__thread` for better portability.
- **Alignment checking:** Use `(uintptr_t)ptr % ALIGNMENT` rather than pointer arithmetic to avoid undefined behavior.
- **Avoid optimization interference:** When writing patterns to memory, use `volatile` or disable optimizations for test code to ensure writes aren't eliminated.

F. Milestone Checkpoint Commands

After implementing each milestone, run these commands to verify basic functionality:

Milestone	Command	Expected Output
1	`cd tests && gcc -I../src test_basic.c ../src/allocator.c ../src/os_mem.c -o test_basic && ./test_basic`	All tests pass with "PASS" messages. No segmentation faults.
2	`cd tests && gcc -I../src test_free_list.c ../src/*.c -pthread -o test_free_list && ./test_free_list`	Tests pass, and `heap_visualize()` output shows coalescing occurring.
3	`cd tests && gcc -I../src test_segregated.c ../src/*.c -o test_segregated && ./test_segregated`	Tests pass; debug output shows blocks being placed in correct size class lists.
4	`cd tests && gcc -I../src test_concurrent.c ../src/*.c -pthread -o test_concurrent && ./test_concurrent`	Tests pass without thread-related crashes; memory usage remains bounded.

G. Debugging Tips for Testing

Symptom	Likely Cause	How to Diagnose	Fix
Test passes alone but fails when run with other tests	Global state not reset between tests (e.g., `free_list_head` persists)	Add `reset_allocator()` function that reinitializes global state	Call `reset_allocator()` at start of each test
`heap_visualize()` shows corrupted metadata	Buffer overflow writing past allocated memory	Add `MAGIC` numbers to headers; check before/after each operation	Ensure `get_total_block_size` includes all padding; check write bounds
Thread test hangs indefinitely	Deadlock in arena or mutex locking	Add logging to `mutex_lock` / `mutex_unlock`; use timeout threads	Ensure mutexes are unlocked in all code paths (even error paths)
Allocation returns valid pointer but program crashes on access	Pointer not aligned for specific architecture (e.g., SSE requires 16-byte)	Check alignment with `ASSERT_ALIGNED(ptr, 16)`	Increase `ALIGNMENT` to 16 for x86_64 systems
Memory usage grows unbounded in stress test	Free blocks not returned to free list, or coalescing not working	Use `heap_visualize()` after each iteration to see free list growth	Verify `free_list_insert` is called on every free, and coalescing merges adjacent blocks

9. Debugging Guide

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

Building a memory allocator is an exercise in precision debugging. Unlike application code where bugs manifest as incorrect business logic, allocator bugs corrupt the very fabric of memory management itself, leading to insidious failures that may only appear long after the actual mistake. This section provides a **practical handbook** for diagnosing the common bugs you'll encounter, transforming you from a frustrated programmer wondering "why does my test crash randomly?" to a forensic expert who can trace symptoms back to their root causes.

Think of debugging your allocator as being a **warehouse safety inspector**. When something goes wrong—a worker gets injured (segfault), inventory goes missing (memory leak), or the warehouse becomes unusably fragmented—you need systematic techniques to examine the warehouse floor plan (heap layout), check the shipping labels (block headers), and trace the workflow logs (allocation patterns) to identify the violated safety protocol.

Common Bug Symptom → Cause → Fix Table

The following table catalogs the most frequent allocator bugs, organized by their observable symptoms. Each entry provides the **likely cause** (what typically went wrong), **diagnostic steps** (how to confirm the hypothesis), and the **fix** (how to correct the underlying issue).

Symptom	Likely Cause	Diagnostic Steps	Fix
Segmentation fault (crash) when calling `free(ptr)`	1. **Invalid pointer**: `ptr` wasn't returned by `malloc`, or was already freed. 2. **Header corruption**: The `block_header_t` was overwritten by buffer overflow. 3. **Free list pointer corruption**: `next` / `prev` pointers in free block are garbage.	1. Check if `ptr` is `NULL` (should be handled). 2. Use `get_header_from_ptr(ptr)` and examine the `magic` field (if `USE_MAGIC` enabled). 3. Run `heap_visualize()` before the crash to see block layout. 4. Use `is_valid_pointer(ptr)` if implemented.	1. In `free()`, return early if `ptr == NULL`. 2. Add bounds checking: ensure user writes stay within `user_size`. 3. Add `magic` field validation in `get_header_from_ptr`.
`malloc()` returns an unaligned pointer (not 8-byte aligned)	1. **Incorrect alignment calculation**: Not rounding up request size to `ALIGNMENT`. 2. **Header size not aligned**: `sizeof(block_header_t)` not considered in alignment. 3. **Pointer arithmetic error**: Calculating user pointer incorrectly from block start.	1. Print the returned pointer value: `printf("%p\n", ptr);` check if `(uintptr_t)ptr % ALIGNMENT == 0`. 2. Trace `get_total_block_size()` calculation. 3. Check `init_block()` and pointer return in `malloc()`.	1. Use `ALIGNMENT` macro consistently. 2. Ensure `user_ptr = (char*)block + HEADER_SIZE`. 3. Round up total size to alignment boundary.
Memory leak: repeated allocations exhaust heap despite frees	1. **Free block not added to free list**: `free()` marks block but doesn't link it. 2. **Free list corruption**: Lost blocks due to incorrect pointer updates. 3. **No coalescing**: Fragmentation prevents reuse of smaller blocks. 4. **mmap blocks not returned**: `os_munmap()` not called for large allocations.	1. Call `heap_visualize()` after each operation to see free list. 2. Count free blocks after sequence: should increase with each `free()`. 3. Check `free_list_insert()` is called in `free()`. 4. Verify `IS_MMAP_FLAG` handling in `return_memory_to_os()`.	1. Implement `free_list_insert()` correctly (update `next` / `prev`). 2. Ensure coalescing merges adjacent free blocks. 3. For mmap blocks, call `os_munmap()` when freed.
Heap corruption: random crashes in unrelated code	1. **Buffer overflow**: User writes past allocated size, overwriting next header. 2. **Use-after-free**: User accesses freed memory, corrupting reallocated block. 3. **Double-free**: Same block freed twice, corrupting free list pointers.	1. Add **guard bytes** (canary values) after each block and check them. 2. Implement `is_double_free()` check in `free()`. 3. Use AddressSanitizer (`-fsanitize=address`) if available.	1. In debug builds, allocate extra guard bytes and validate on free. 2. Track allocated blocks in a debug registry. 3. Set `allocated = 0` on free and check in `free()`.
Infinite loop in `find_free_block()` or during allocation	1. **Circular free list**: `next` pointers form a cycle, never reaching `NULL`. 2. **Incorrect loop termination**: Condition checks wrong pointer field. 3. **Corrupted size field**: Free block size is zero or huge, causing wrap-around.	1. Add loop counter limit (e.g., max 10000 iterations) and abort. 2. Print free list traversal: `printf("Visiting block %p, size %zu, next %p\n", block, block->size, block->next)`. 3. Check `free_list_remove()` logic when coalescing.	1. Ensure `free_list_remove()` sets `block->next = NULL`. 2. Validate `block->size` is reasonable (not zero, less than heap size). 3. Use doubly-linked list correctly: update both `next` and `prev`.
Poor performance: allocation becomes slower over time	1. **External fragmentation**: Free list has many small, non-contiguous blocks. 2. **No splitting**: Using entire large block for small request. 3. **No coalescing**: Adjacent	1. Visualize heap after many alloc/free cycles: `heap_visualize()`. 2. Measure free list length over time. 3. Check if `split_block()` is called when block is much larger than request.	1. Implement **boundary tags** (footers) for bidirectional coalescing. 2. Set `MIN_BLOCK_SIZE` to prevent splitting too small. 3. Consider segregated lists (Milestone 3) for common sizes.

Symptom	Likely Cause	Diagnostic Steps	Fix
	free blocks remain separate. 4. **Inefficient search**: First-fit scanning long free list every time.	4. Verify `coalesce_block()` merges adjacent free blocks.	
Thread-safety issues: crashes only with multiple threads	1. **Missing locks**: Concurrent access to shared free list without mutex. 2. **Lock ordering deadlock**: Two threads acquire locks in different order. 3. **Race condition**: Thread reads stale value after another thread updates. 4. **False sharing**: Thread-local caches on same cache line cause contention.	1. Add logging showing which thread holds lock. 2. Use `DEBUG_LOG` to trace lock acquisition order. 3. Run stress test with many threads making small allocations. 4. Check `thread_cache_t` alignment to cache line.	1. Use `mutex_lock()` / `mutex_unlock()` around shared state. 2. Implement per-thread arenas to reduce lock contention. 3. Align `thread_cache_t` to 64-byte cache line boundary.
`sbrk()` or `mmap()` failing unexpectedly	1. **Requesting too much memory**: Exceeding system limits. 2. **Not checking return value**: Assuming `sbrk()` always succeeds. 3. **Memory exhaustion**: System out of memory (or address space). 4. **Incorrect size calculation**: Integer overflow in `get_total_block_size()`.	1. Check `errno` after system call failure. 2. Log requested size vs. available memory. 3. Add overflow detection in size calculations. 4. Test with artificial limit (e.g., `ulimit -v 50000`).	1. Implement `calculate_total_size()` with overflow detection. 2. Check `os_sbrk()` returns `(void*)-1` and return `NULL` from `malloc()`. 3. Set reasonable threshold for mmap (e.g., `SBRK_THRESHOLD`).
Block splitting creates impossibly small free block	1. **Not respecting `MIN_BLOCK_SIZE`**: Split leaves remainder smaller than header+footer. 2. **Incorrect size calculation**: Forgetting alignment padding in split logic. 3. **Off-by-one errors**: Arithmetic with `HEADER_SIZE`, `FOOTER_SIZE`.	1. Print before/after sizes in `split_block()`. 2. Check `MIN_BLOCK_SIZE` definition includes header+footer+minimum payload. 3. Verify `get_total_block_size(MIN_BLOCK_SIZE)` doesn't overflow.	1. In `split_block()`, if remainder < `MIN_BLOCK_SIZE`, don't split. 2. Use `get_total_block_size()` consistently for size calculations. 3. Add assertion: `assert(remaining >= MIN_BLOCK_SIZE)` before splitting.
Coalescing doesn't merge adjacent free blocks	1. **Missing footer (`footer_t`)**: Cannot check if next block is free. 2. **Boundary tag mismatch**: Header/footer size fields don't match. 3. **Not checking physical adjacency**: Assuming list order equals physical order. 4. **Coalescing across mmap boundary**: Trying to merge sbrk and mmap blocks.	1. Use `heap_visualize()` to see adjacent free blocks that should merge. 2. Check footer values match header values. 3. Verify `coalesce_block()` checks `IS_MMAP_FLAG`. 4. Test with simple sequence: alloc A, alloc B, free A, free B (should coalesce).	1. Implement **boundary tags**: footer with `size` and `allocated` at block end. 2. In `coalesce_block()`, calculate next block via `current + total_size`. 3. Skip coalescing if `IS_MMAP_FLAG` is set on either block.
Segregated lists: allocation uses wrong size class	1. **Incorrect `size_to_class_index()` mapping**: Returns wrong class	1. Print mapping: `printf("size %zu -> class %d\n", size, size_to_class_index(size))`.	1. Ensure `size_to_class_index()` rounds up to next power-of-two.

Symptom	Likely Cause	Diagnostic Steps	Fix
	for size. 2. **Free list per class not initialized**: `size_classes` array has `NULL` pointers. 3. **Block splitting puts remainder in wrong class**: Not re-calculating class for remainder. 4. **Size class boundaries off by one**: e.g., 32-byte request goes to 64-byte class.	2. Check `NUM_SIZE_CLASSES` and `LARGEST_SIZE_CLASS` constants. 3. Verify `get_size_class_head()` returns correct list head. 4. Test with exact size class boundaries.	2. Initialize all `size_classes` entries to `NULL` at startup. 3. When splitting, recalculate class for remainder block. 4. Use `(size + CLASS_GRANULARITY - 1) & ~(CLASS_GRANULARITY - 1)` for rounding.

Allocator-Specific Debugging Techniques

Beyond generic debugging approaches, memory allocators require specialized techniques to inspect their internal state. These methods transform the opaque heap into a visible, analyzable structure.

Mental Model: The Warehouse Surveillance System

Imagine installing **security cameras** throughout your warehouse (heap visualizer), placing **tamper-evident seals** on each shelf unit (magic numbers), and maintaining **detailed audit logs** of every transaction (allocation tracing). When something goes wrong, you can review the camera footage, check which seals are broken, and trace exactly which operations led to the corruption.

Technique 1: Magic Number Validation

A **magic number** is a known constant value placed in each block header that serves as a "tamper-evident seal." When the seal is broken (value changed), you know the header was corrupted.

> **Decision: Use Magic Numbers in Debug Builds Only**
>
> - **Context**: We need to detect header corruption without significant runtime overhead in production.
> - **Options Considered**:
>   1. Always include magic number (simpler, but 4-byte overhead always)
>   2. Only in debug builds (conditional compilation)
>   3. Use checksum instead of magic number (stronger but more compute)
> - **Decision**: Use conditional compilation (`#ifdef DEBUG`) to include magic number only in debug builds.
> - **Rationale**: Debug builds prioritize detection; production builds prioritize performance and memory efficiency. Magic numbers catch most corruption (buffer overflows) with minimal code complexity.
> - **Consequences**: Debug builds are larger but safer; production builds are leaner but offer less corruption detection.

How to implement:

1. Add `magic` field to `block_header_t` (already in naming conventions)
2. Set `magic = MAGIC` (e.g., `0xDEADBEEF`) in `init_block()`
3. In `get_header_from_ptr()`, verify `header->magic == MAGIC` before returning
4. In `heap_validate()`, traverse all blocks checking magic numbers

Diagnostic power: When a crash occurs, immediately check the magic number of the suspected block. If corrupted, you know:

- Buffer overflow from previous block (if magic corrupted but size looks valid)
- Wild pointer write (if entire header looks like garbage)
- Double-free (if magic intact but `allocated` flag says free)

Technique 2: Heap Visualization Function

The `heap_visualize()` function prints the entire heap layout, showing each block's address, size, status, and linkages. This is your **warehouse floor plan viewer**.

Block Address	Total Size	User Size	Status	Next Free	Prev Free	Notes
0x1000	48	40	ALLOC	-	-	Header at 0x1000
0x1030	64	56	FREE	0x1080	NULL	In free list
0x1080	32	24	FREE	NULL	0x1030	End of free list

Implementation approach:

1. Start from `heap_base` (global allocator state)
2. Walk blocks using size fields (not free list pointers, which only connect free blocks)
3. For each block, print:
   - Address (`%p`)
   - Total size (including header/footer)
   - User size (`total_size - HEADER_SIZE - FOOTER_SIZE`)
   - Status (`ALLOCATED` / `FREE` / `MMAPPED`)
   - Next/prev pointers if free
   - Magic number status if debug
4. Also print free list separately to verify consistency

When to use:

- After each operation in a failing test case
- When `malloc()` returns `NULL` unexpectedly
- Before/after `free()` to verify coalescing
- When free list appears corrupted

Technique 3: Boundary Check Canaries

Place known values (**canaries**) at strategic boundaries and check them on deallocation. This is like placing motion sensors between warehouse shelves that trigger if someone crosses into restricted areas.

Common canary placements:

- **After user payload**: Detect buffer overflows
- **Before footer**: Detect underflows (backwards overflow)
- **Around free list pointers**: Detect corruption of free block metadata

Implementation (debug builds only):

```c
// In malloc(), allocate extra CANARY_SIZE bytes
size_t total_with_canary = total_size + 2 * CANARY_SIZE;
// Initialize canaries: before payload and after payload
uint32_t* front_canary = (uint32_t*)((char*)block + HEADER_SIZE);
*front_canary = CANARY_VALUE;
uint32_t* rear_canary = (uint32_t*)((char*)block + total_size - FOOTER_SIZE - CANARY_SIZE);
*rear_canary = CANARY_VALUE;

// In free(), check canaries before deallocating
if (*front_canary != CANARY_VALUE || *rear_canary != CANARY_VALUE) {
    DEBUG_LOG("Buffer overflow/underflow detected at block %p\n", block);
    // Abort or mark as corrupted
}
```

Technique 4: Allocation Tracing and Sequence Logging

When bugs only appear with specific allocation patterns, you need to **replay the exact sequence** that caused the failure. Implement a tracing system that logs every `malloc` and `free` operation with details.

What to log:

- Operation (`malloc` / `free` / `calloc` / `realloc`)
- Size requested (or pointer for free)
- Returned address
- Call site (file/line if using `__FILE__` / `__LINE__`)
- Current heap state (free list length, total heap size)

Usage pattern:

1. Run failing test with tracing enabled
2. Capture log file
3. Reproduce bug by replaying exact sequence
4. Add `heap_visualize()` at critical points in the log

Technique 5: External Tool Integration

While building your own debugging infrastructure is educational, don't neglect powerful existing tools:

Tool	What It Detects	How to Use with Your Allocator
Valgrind/Memcheck	Memory leaks, invalid reads/writes, use of uninitialized memory	1. Compile with debug symbols (`-g`) 2. Run: `valgrind --leak-check=full ./my_program` 3. Note: Valgrind intercepts standard `malloc` ; to test yours, rename functions or use LD_PRELOAD
AddressSanitizer (ASan)	Buffer overflows, use-after-free, double-free	1. Compile: `-fsanitize=address -g` 2. Run normally; ASan prints detailed report on error 3. Works with custom allocator if you implement ASan interface hooks
GDB (GNU Debugger)	Segmentation faults, inspect memory, backtraces	1. Run: `gdb ./my_program` 2. Set breakpoint in `malloc()` / `free()` 3. Use `x/20wx addr` to examine memory 4. Use `watch` point to catch writes to specific address
Electric Fence	Detect buffer overruns immediately	1. Link with `-lefence` 2. Allocations placed on page boundaries with guard pages 3. Overrun causes immediate segmentation fault

Pro tip for AddressSanitizer: Implement `__asan_poison_memory_region()` and `__asan_unpoison_memory_region()` hooks to mark allocated memory as "poisoned" (inaccessible) and free memory as "unpoisoned." This makes ASan work with your custom allocator, detecting when user code accesses freed memory.

Technique 6: Consistency Checking with `heap_validate()`

Implement a comprehensive validation function that checks every invariant of your heap data structures. Run this after every operation in debug builds, or when corruption is suspected.

What `heap_validate()` should check:

1. **Block alignment**: Every block starts at aligned address
2. **Size consistency**: Header size matches footer size
3. **Magic numbers**: All headers have correct magic (if enabled)
4. **Free list integrity**:
   - All free blocks are in free list
   - No allocated blocks in free list
   - Free list pointers form consistent doubly-linked list
   - `next->prev == current` for all free blocks
5. **Physical adjacency**: No overlapping blocks
6. **Size class correctness**: Free blocks in correct segregated list (if using)
7. **Boundary tags**: Header and footer `allocated` flags match

When to call it:

- At the beginning/end of `malloc()` and `free()` in debug builds
- In test suites between test cases
- When `DEBUG` macro is defined and user calls special debug function

Implementation Guidance

This subsection provides practical code and techniques for implementing the debugging features described above.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Debugging Framework	Conditional compilation with `#ifdef DEBUG`	Runtime-configurable debug levels
Heap Visualization	Simple `printf` to stdout	HTML/JSON output for browser visualization
Corruption Detection	Magic numbers in headers	XOR-linked lists with checksums
Tracing	Log to file with `fprintf`	In-memory ring buffer with low overhead

B. Recommended File/Module Structure

Add debugging facilities in separate files to keep core logic clean:

```
project-root/
  src/
    allocator.c          # Core allocator logic
    allocator.h          # Public interface
    debug.c              # Debugging functions (heap_visualize, heap_validate)
    debug.h              # Debug macros and declarations
    os_mem.c             # OS interface (sbrk, mmap)
    strategies.c         # Allocation strategies
    thread_cache.c       # Thread-local caching
  tests/
    test_basic.c         # Unit tests
    test_debug.c         # Debugging-specific tests
```

C. Infrastructure Starter Code

Here's complete, working code for a basic debugging framework:

`debug.h`:

```c
#ifndef DEBUG_H

#define DEBUG_H


#include <stdio.h>

#include <stdint.h>

#include "allocator.h"


// Debug configuration

#ifdef DEBUG_BUILD

#define DEBUG 1

#define USE_MAGIC 1

#define CANARY_SIZE 4

#define CANARY_VALUE 0xCAFEBABE

#else

#define DEBUG 0

#define USE_MAGIC 0

#define CANARY_SIZE 0

#endif


// Conditional debug logging

#define DEBUG_LOG(fmt, ...) \

    do { \

        if (DEBUG) { \

            fprintf(stderr, "[DEBUG] %s:%d: " fmt, __FILE__, __LINE__, ##__VA_ARGS__); \

        } \

    } while (0)


// Debug function declarations

void heap_visualize(void);

int heap_validate(void);

void dump_free_list(void);

void trace_malloc(size_t size, void* ptr, const char* file, int line);

void trace_free(void* ptr, const char* file, int line);


#endif // DEBUG_H
```

`debug.c` (partial starter implementation):

```c
#include "debug.h"
#include <assert.h>

// Global trace file pointer
static FILE* trace_file = NULL;

void init_tracing(const char* filename) {
    if (DEBUG && filename) {
        trace_file = fopen(filename, "w");
        if (trace_file) {
            fprintf(trace_file, "Operation,Size,Address,HeapSize,FreeBlocks\n");
        }
    }
}

void trace_malloc(size_t size, void* ptr, const char* file, int line) {
    if (!trace_file) return;

    // Count free blocks (simplified)
    size_t free_count = 0;
    block_header_t* curr = allocator_state.free_list_head;
    while (curr) {
        free_count++;
        curr = curr->next;
    }

    fprintf(trace_file, "MALLOC,%zu,%p,%s:%d,%zu,%zu\n",
            size, ptr, file, line, allocator_state.heap_size, free_count);
    fflush(trace_file);
}

void trace_free(void* ptr, const char* file, int line) {
    if (!trace_file) return;

    fprintf(trace_file, "FREE,%p,%s:%d\n", ptr, file, line);
    fflush(trace_file);
}

// Simple heap visualizer
```

```c
void heap_visualize(void) {
    if (!DEBUG) return;

    printf("\n=== HEAP VISUALIZATION ===\n");
    printf("Heap base: %p, Heap size: %zu bytes\n",
           allocator_state.heap_base, allocator_state.heap_size);

    void* current = allocator_state.heap_base;
    size_t total_used = 0;
    size_t total_free = 0;
    int block_count = 0;

    while ((uintptr_t)current < (uintptr_t)allocator_state.heap_base + allocator_state.heap_size) {
        block_header_t* header = (block_header_t*)current;

        // Safety check
        if (header->size == 0) {
            printf("ERROR: Zero-sized block at %p, heap corrupted?\n", header);
            break;
        }

        size_t total_size = header->size;
        size_t user_size = total_size - HEADER_SIZE - FOOTER_SIZE;
        footer_t* footer = (footer_t*)((char*)current + total_size - FOOTER_SIZE);

        printf("Block %d at %p:\n", block_count, header);
        printf("  Total size: %zu, User size: %zu\n", total_size, user_size);
        printf("  Status: %s\n", header->allocated ? "ALLOCATED" : "FREE");

        if (header->allocated) {
            total_used += user_size;
        } else {
            total_free += user_size;
            printf("  Free list: next=%p, prev=%p\n", header->next, header->prev);
        }
```

```
        // Verify footer matches header
        if (footer->size != header->size || footer->allocated != header->allocated) {
            printf("  WARNING: Footer mismatch! Header size=%zu, footer size=%zu\n",
                   header->size, footer->size);
        }

        #if USE_MAGIC
        if (header->magic != MAGIC) {
            printf("  WARNING: Magic number corrupted! Expected 0x%x, got 0x%x\n",
                   MAGIC, header->magic);
        }
        #endif

        current = (char*)current + total_size;
        block_count++;
    }

    printf("\nSummary: %d blocks, %zu bytes used, %zu bytes free\n",
           block_count, total_used, total_free);
    printf("Fragmentation: %.1f%% free space\n",
           allocator_state.heap_size > 0 ?
           100.0 * total_free / allocator_state.heap_size : 0.0);

    // Also dump free list structure
    dump_free_list();
}
```

D. Core Logic Skeleton Code

Here are skeletons for key debugging functions you should implement:

`heap_validate()`:

```c
// Comprehensive heap integrity checker
// Returns 1 if heap is valid, 0 if corruption detected
int heap_validate(void) {
    // TODO 1: If heap_base is NULL, return 1 (empty heap is valid)

    // TODO 2: Walk all blocks using size fields (physical walk)
    //   For each block:
    //   a. Check alignment: ((uintptr_t)block % ALIGNMENT) == 0
    //   b. Check size is non-zero and less than remaining heap
    //   c. If USE_MAGIC, check header->magic == MAGIC
    //   d. Find footer and verify footer->size == header->size
    //   e. Verify footer->allocated == header->allocated
    //   f. If block is FREE, verify it's in free list (next/prev consistent)
    //   g. If block is ALLOCATED, verify it's NOT in free list

    // TODO 3: Walk free list (logical walk)
    //   For each free block:
    //   a. Verify allocated flag is 0
    //   b. Verify next/prev pointers are valid (not pointing to allocated blocks)
    //   c. Verify doubly-linked list consistency: next->prev == current

    // TODO 4: Check for overlapping blocks (sum of sizes should equal heap size)

    // TODO 5: If using segregated lists, verify each free block is in correct size class

    // TODO 6: If any check fails, print error details and return 0
    //   Use DEBUG_LOG to report specific violations

    return 1; // Placeholder
}
```

Enhanced `get_header_from_ptr()` with validation:

```c
// Convert user pointer to block header with safety checks
block_header_t* get_header_from_ptr(void *ptr) {
    if (ptr == NULL) {
        return NULL;
    }

    // Calculate header location (before user memory)
    block_header_t* header = (block_header_t*)((char*)ptr - HEADER_SIZE);

    #if USE_MAGIC
    // Validate magic number to detect wild pointers
    if (header->magic != MAGIC) {
        DEBUG_LOG("ERROR: Invalid pointer %p: magic number mismatch (got 0x%x, expected 0x%x)\n",
                  ptr, header->magic, MAGIC);
        // In debug builds, abort to catch bug immediately
        #ifdef DEBUG
        abort();
        #endif
        return NULL;
    }
    #endif

    // Additional sanity checks
    if (header->size < HEADER_SIZE + FOOTER_SIZE) {
        DEBUG_LOG("ERROR: Block size too small: %zu\n", header->size);
        return NULL;
    }

    return header;
}
```

Debug-enhanced `malloc()` wrapper:

```c
void* debug_malloc(size_t size, const char* file, int line) {
    // Call actual malloc
    void* ptr = malloc(size);

    #if DEBUG
    // Trace the allocation
    trace_malloc(size, ptr, file, line);

    // In debug builds, validate heap after each operation
    if (!heap_validate()) {
        DEBUG_LOG("Heap corruption detected after malloc(%zu) at %s:%d\n",
                  size, file, line);
        heap_visualize();
        // Optionally abort here to catch bugs early
    }

    // If using canaries, set them up
    #if CANARY_SIZE > 0
    if (ptr != NULL) {
        block_header_t* header = get_header_from_ptr(ptr);
        size_t user_size = header->size - HEADER_SIZE - FOOTER_SIZE;

        // Place canary before and after user payload
        uint32_t* front_canary = (uint32_t*)ptr;
        *front_canary = CANARY_VALUE;

        uint32_t* rear_canary = (uint32_t*)((char*)ptr + user_size - CANARY_SIZE);
        *rear_canary = CANARY_VALUE;

        // Store canary locations in header (optional)
        header->front_canary = front_canary;
        header->rear_canary = rear_canary;
    }
    #endif
    #endif // DEBUG
```

```
    return ptr;
}

// Macro to capture file/line automatically
#define malloc(size) debug_malloc(size, __FILE__, __LINE__)
```

E. Language-Specific Hints

- **C-specific**: Use `__attribute__((packed))` for `block_header_t` to prevent compiler padding that might confuse size calculations.
- **Portability**: Use `uintptr_t` for pointer arithmetic to avoid undefined behavior.
- **Signal handling**: Consider catching `SIGSEGV` and printing heap state before crashing.
- **Thread safety**: In debug mode, add lock validation to ensure threads don't double-lock or access without locking.

F. Milestone Checkpoint Debugging Tests

After each milestone, run these specific tests to verify your debugging infrastructure:

Milestone 1 Checkpoint:

```bash
# Compile with debug enabled
gcc -DDEBUG_BUILD -g -o test_m1 tests/test_basic.c src/allocator.c src/debug.c

# Run simple test that should show heap visualization
./test_m1

# Expected: Should print heap visualization showing:
# 1. Initial empty heap
# 2. After malloc(64): one allocated block
# 3. After free: one free block
# 4. No corruption warnings
```

Milestone 2 Checkpoint:

```bash
# Run fragmentation test
./test_m1 --test-fragmentation

# Expected: Heap visualization should show coalescing:
# 1. Allocate A, B, C (three adjacent blocks)
# 2. Free B, then free A -> should coalesce into one large free block
# 3. Not: Two separate free blocks (coalescing broken)
```

Milestone 4 Checkpoint:

```bash
# Run thread safety test with debug tracing
./test_m1 --test-threads --trace-file=trace.log

# Expected: Trace file should show:
# 1. Thread IDs for each operation
# 2. No double-locking warnings
# 3. Heap validation passes after each thread completes
```

G. Debugging Tips Table

For quick reference during implementation:

Symptom	Quick Diagnostic	Likely Fix
Random crashes in `free()`	Add `magic` field, check in `get_header_from_ptr()`	Validate pointers before dereferencing
Memory leak in test suite	Run `heap_visualize()` after each test case	Ensure `free_list_insert()` is called
Heap grows unbounded	Check `coalesce_block()` merges adjacent free blocks	Implement boundary tags (footers)
Thread deadlock	Add lock acquisition logging	Ensure consistent lock order
unaligned pointer returned	Print `(uintptr_t)ptr % ALIGNMENT`	Fix alignment calculation in `malloc()`

---

10. Future Extensions

> **Milestone(s):** Beyond core milestones (conceptual extensions)

This section explores potential enhancements that could transform our **teaching allocator** into a more feature-complete, production-ready system. The modular architecture established in previous sections—with clean separation between OS interface, block management, allocation strategies, and concurrency—provides a solid foundation for these extensions. Each enhancement addresses a specific limitation or adds valuable capabilities while building upon existing components.

Possible Enhancements

Our current allocator implements the essential `malloc` and `free` operations with increasingly sophisticated optimizations. However, real-world memory allocators offer additional functionality, performance improvements, and debugging support. Below are four major extensions that demonstrate how the architecture can evolve.

1. Implementing `realloc` for Dynamic Resizing

Mental Model: The Expanding Shipping Container
Imagine you've rented a storage unit (allocated block) that's becoming too small. You could:

1. **Move to a larger unit** (allocate new, copy contents, free old)
2. **Expand into the adjacent unit if available** (coalesce with next free block)
3. **Shrink and release extra space** (split the block)

The `realloc` function provides exactly this flexibility, allowing programs to resize existing allocations while preserving their contents.

Architecture Decision Record: In-Place Expansion vs. Allocation-Copy-Free

Decision: Hybrid Approach with In-Place Expansion Attempt First

- **Context**: `realloc(ptr, new_size)` must resize the memory block pointed to by `ptr`. Naively allocating a new block, copying data, and freeing the old is simple but inefficient when adjacent free space allows expansion.
- **Options Considered**:
  1. **Always allocate-copy-free**: Simple, always works but causes unnecessary copying and fragmentation.
  2. **Attempt in-place expansion, fallback to copy**: Check if the next adjacent block is free and large enough to satisfy the new size without moving. More complex but more efficient.
  3. **Shrink in-place always**: For shrinking, always split the block and return the excess to the free list.
- **Decision**: Implement a hybrid approach that attempts in-place expansion when possible (checking the next adjacent block's status and size), falls back to allocate-copy-free when expansion fails, and always splits when shrinking.
- **Rationale**: This balances implementation complexity with performance benefits. The common case of small expansions can often be satisfied by adjacent free blocks (especially with coalescing). The block management system already has all the machinery to check adjacent blocks and split/coalesce.
- **Consequences**: Requires careful handling of the `IS_MMAP_FLAG` (large blocks allocated via `mmap` cannot be expanded in place), additional logic to check next block's status, and potential to introduce new fragmentation patterns if expansion splits a large free block.

Option	Pros	Cons	Chosen?
Always allocate-copy-free	Simple implementation, works for all cases	Inefficient copying, increases fragmentation	No
Attempt in-place expansion first	Optimizes common case, reduces copying	More complex, must handle edge cases	**Yes**
Always use `mremap` for mmap blocks	OS handles expansion efficiently	Linux-specific, not portable	Partial (for mmap)

Integration with Current Architecture:
The `realloc` implementation would reside in the same module as `malloc` and `free` (e.g., `allocator.c`). It would leverage existing functions:

1. **Input Validation**: Use `is_valid_pointer()` to ensure `ptr` came from our allocator.
2. **Size Calculation**: Compute `new_total_size` using `get_total_block_size()`.
3. **Same Size Request**: If `new_size` equals current size, return `ptr` unchanged.
4. **Shrinking Case**: If `new_size` is smaller, call `split_block()` to create a free remainder, coalesce if possible, return original `ptr`.
5. **Expansion Case**: Check if next adjacent block exists, is free, and combined size is sufficient.
   - If yes: coalesce, potentially split if combined block is too large, update headers/footers and free list.
   - If no: call `malloc(new_size)`, `memcpy()` contents, `free(ptr)`, return new pointer.
6. **Special Handling for mmap blocks**: Use `mremap()` on Linux if possible, otherwise fallback to copy.

Common Pitfalls:

- ⚠️ **Pitfall: Forgetting to handle the `IS_MMAP_FLAG`**
  Large blocks allocated via `mmap` have different expansion semantics. Attempting to check a "next block" in a separate mapping will corrupt memory.
  Fix: Check the `IS_MMAP_FLAG` in the header early and use OS-specific `mremap()` or fallback to copy.
- ⚠️ **Pitfall: Not updating all metadata after in-place expansion**
  When coalescing with the next block, you must remove the next block from the free list, update the current block's size in both header and footer, and adjust any free list pointers.
  Fix: Reuse `coalesce_block()` and `free_list_remove()` functions to ensure consistency.
- ⚠️ **Pitfall: Integer overflow in new size calculation**
  Similar to `malloc`, `realloc` must guard against overflow when calculating `new_total_size`.
  Fix: Use `calculate_total_size()` with overflow detection.

2. Buddy Allocation for Reduced External Fragmentation

Mental Model: The Recursive Shelf Divider
Imagine a large empty warehouse shelf that can only be divided into two equal halves (buddies). Each half can be further divided recursively. When a

small item arrives, you repeatedly split the smallest shelf that's still large enough. When two adjacent buddy shelves become empty, they merge back into their parent shelf.

The **buddy allocation** system organizes memory into power-of-two sized blocks that can be efficiently split and merged, minimizing external fragmentation for certain allocation patterns.

Architecture Decision Record: Segregated Lists vs. Buddy System

> **Decision: Implement Buddy Allocation as an Alternative Strategy**
>
> - **Context**: Our current segregated free lists reduce search time but can still suffer from external fragmentation when free blocks are not the exact size needed. Buddy allocation guarantees that any free block can be split into perfectly mergeable buddies.
> - **Options Considered**:
>   1. **Replace segregated lists with buddy system**: Complete overhaul of allocation strategy.
>   2. **Implement buddy system alongside existing strategies**: Allow runtime selection between segregated fits and buddy allocation.
>   3. **Use buddy system only for larger blocks**: Hybrid approach where small blocks use segregated lists, large blocks use buddy system.
> - **Decision**: Implement buddy allocation as a compile-time alternative to segregated free lists, selectable via a configuration flag. This keeps the codebase manageable while allowing comparison of fragmentation characteristics.
> - **Rationale**: Buddy allocation is a classic memory management algorithm worth implementing for educational purposes. Having it as an alternative strategy allows learners to empirically compare fragmentation and performance. A full hybrid system would be complex but could be a future extension.
> - **Consequences**: Requires significant changes to block representation (buddy system typically uses implicit trees rather than explicit free lists), new splitting/merging logic, and potentially higher internal fragmentation due to power-of-two rounding.

Option	Pros	Cons	Chosen?
Replace segregated lists	Uniform handling, eliminates external fragmentation	Loses O(1) small allocations, high internal fragmentation	No
Alternative strategy	Educational, allows comparison	Maintenance of two largely separate code paths	**Yes**
Hybrid approach	Best of both worlds	Complex integration, difficult debugging	Future possibility

Integration with Current Architecture:
The buddy allocator would be implemented as a new allocation strategy component:

1. **Block Structure**: Instead of `block_header_t` with `next`/`prev` pointers, buddy blocks would store:
   - Size (power of two)
   - Allocated flag
   - Buddy identifier (could be implicit from address)
   - Magic number for corruption detection
2. **Free Block Management**: Maintain an array of free lists, one for each power-of-two size (e.g., 32, 64, 128, ... up to maximum). Each list contains free blocks of exactly that size.
3. **Allocation Algorithm** (`malloc_buddy`):
   - Round request up to next power of two (with minimum).
   - Check corresponding free list.
   - If empty, take a block from the next larger size, split recursively (creating buddies), and add the unused buddy to its free list.
   - Return the block.
4. **Deallocation Algorithm** (`free_buddy`):
   - Mark block as free.
   - Check if its buddy is also free (using address arithmetic).
   - If yes, merge them recursively and move to the larger size free list.
   - Continue until buddy is allocated or maximum size reached.

5. **Integration Points**: The buddy allocator would replace the `malloc_segregated` and `free` implementations when configured, but reuse the same OS interface (`get_memory_from_os`) and block header/footer concepts.

Common Pitfalls:

- ⚠️ **Pitfall: Incorrect buddy address calculation**
  The buddy of a block at address `addr` with size `size` is at `addr ^ size`. Getting this wrong leads to incorrect merging.
  Fix: Use well-tested macros: `#define BUDDY_OF(addr, size) ((void*)((uintptr_t)(addr) ^ (size)))`
- ⚠️ **Pitfall: Not handling alignment constraints**
  Buddy blocks must be aligned to their size (e.g., a 64-byte block must start at a 64-byte aligned address). Initial memory from OS must be aligned to maximum block size.
  Fix: Use `align_to_page()` with the maximum buddy size (e.g., 1MB) when requesting memory from OS.
- ⚠️ **Pitfall: Fragmented free lists from improper splitting**
  If you don't immediately add both buddies to free lists when splitting, you can lose track of them.
  Fix: Always add the unused buddy to the appropriate free list immediately after split.

3. Comprehensive Memory Debugging and Statistics

Mental Model: The Warehouse Security Camera System
Imagine installing cameras throughout the warehouse that:

1. **Log every entry/exit** (allocation/free)
2. **Detect unauthorized access** (use-after-free)
3. **Measure space utilization** (statistics)
4. **Check for structural damage** (heap corruption)

A memory debugging system transforms the allocator from a black box into an observable, instrumented component that can catch bugs and profile performance.

Architecture Decision Record: Compile-Time Debugging vs. Runtime Instrumentation

> **Decision: Layered Debugging Support with Compile-Time Flags**
>
> - **Context**: Production allocators need debugging capabilities to diagnose memory errors, but these features incur performance overhead. Different development stages require different levels of instrumentation.
> - **Options Considered**:
>   1. **Separate debug build**: Compile with `-DDEBUG` to enable all checks, release build has none.
>   2. **Runtime configuration**: Allow enabling/disabling features via environment variables or API calls.
>   3. **Layered approach**: Core debugging (bounds checking, double-free detection) always compiled in but gated by a flag; heavy instrumentation (full logging, statistics) separately configurable.
> - **Decision**: Implement layered debugging where basic integrity checks (magic numbers, canaries) are always compiled but disabled in release via `DEBUG` flag. Heavyweight features (tracing, detailed statistics) are controlled via separate `ENABLE_STATS` and `ENABLE_TRACING` flags.
> - **Rationale**: This balances safety and performance. Basic checks have minimal overhead when disabled (branch prediction). Developers can enable detailed tracing only when investigating specific issues. The layered approach matches real-world allocators like jemalloc.
> - **Consequences**: Increases code complexity with conditional compilation, but provides flexible diagnostics. Statistics collection requires additional per-block or global counters.

Option	Pros	Cons	Chosen?
Separate debug build	Clear performance distinction, simple	Need to rebuild to change debug level	No
Runtime configuration	Flexible without recompilation	Always carries code overhead, complex	No
Layered compile-time	Balance of flexibility and performance	Multiple configuration flags, conditional code	**Yes**

Integration with Current Architecture:
Debugging features would integrate throughout the codebase:

1. **Enhanced Block Header**: Add fields when `DEBUG` is enabled:

```c
#ifdef DEBUG
uint32_t magic;          // MAGIC constant for corruption detection
const char* file;        // File where allocation occurred
int line;                // Line number where allocation occurred
uint32_t canary_start;   // CANARY_VALUE at block start
#endif
```

2. **Tracing System**: When `ENABLE_TRACING` is defined:
   - `trace_malloc()` and `trace_free()` log to a file or ring buffer.
   - Include timestamp, thread ID, size, pointer address, and call site.
   - `init_tracing()` initializes the log, `dump_trace()` outputs collected data.
3. **Statistics Collection**: When `ENABLE_STATS` is defined:
   - Global counters in `allocator_state_t`: total allocations, total frees, current allocated bytes, peak allocation, etc.
   - Per-size-class statistics in segregated free lists.
   - Functions `get_allocator_stats()` and `print_allocator_stats()` to report.
4. **Runtime Checks**:
   - `heap_validate()`: Walks the entire heap, checking magic numbers, canaries, and free list integrity.
   - `is_double_free()`: Checks if a block is already marked free before freeing.
   - Boundary canaries: Place `CANARY_VALUE` before and after user payload to detect overflows/underflows.
5. **Integration Points**: All debug features would be conditionally compiled into existing functions:
   - `malloc()` calls `trace_malloc()` and updates statistics.
   - `free()` validates canaries and calls `trace_free()`.
   - Header initialization sets magic numbers and canaries.

Common Pitfalls:

- ⚠️ **Pitfall: Debug overhead affecting timing-sensitive code**
  Extensive logging or frequent heap validation can dramatically slow down programs, masking timing bugs.
  Fix: Use lightweight statistics (atomic counters) in production, keep heavy tracing disabled by default.
- ⚠️ **Pitfall: Not resetting statistics after fork**
  In multiprocess programs, statistics counters become meaningless after `fork()` if not reset.
  Fix: Use `pthread_atfork()` to register handlers that reset statistics in the child process.
- ⚠️ **Pitfall: Infinite recursion in debug functions**
  If `printf()` or logging functions themselves call `malloc()`, you get infinite recursion.
  Fix: Use static buffers, write directly to file descriptors, or allocate debug structures upfront.

4. Garbage Collection Pass for Orphaned Blocks

Mental Model: The Lost-and-Found Periodic Cleanup

Imagine workers occasionally leaving tools scattered around the warehouse. A periodic cleanup crew walks through, identifies tools not associated with any active work order, and returns them to the storage shelves.

A **garbage collection** pass automatically identifies and reclaims memory blocks that are no longer reachable by the program, providing an alternative to manual `free()` calls.

Architecture Decision Record: Reference Counting vs. Tracing Collector

Decision: Implement Mark-and-Sweep as an Optional Feature

- **Context**: Manual memory management is error-prone. Automatic garbage collection (GC) can prevent memory leaks but introduces complexity and performance overhead.
- **Options Considered**:
  1. **Reference counting**: Each block tracks how many pointers reference it. When count reaches zero, block is freed. Simple but cannot handle cyclic references.
  2. **Tracing collector (mark-and-sweep)**: Periodically scan program roots (stack, globals) to mark reachable blocks, then sweep unreachable ones. Handles cycles but requires stopping the world (STW).
  3. **Hybrid approach**: Use reference counting for most objects, periodic tracing to collect cycles.
- **Decision**: Implement an optional mark-and-sweep garbage collector that can be enabled for debugging or specific use cases. This is educational and demonstrates how GC interacts with a manual allocator.
- **Rationale**: Mark-and-sweep is the classic GC algorithm and works with any allocator. Implementing it shows how automatic memory management builds upon the manual foundation. Making it optional ensures learners understand both paradigms.
- **Consequences**: Requires significant extension to track all allocated blocks, scan roots conservatively, and handle synchronization during collection. Performance will be poor compared to production GCs.

Option	Pros	Cons	Chosen?
Reference counting	Immediate reclamation, simple	Cycles cause leaks, overhead on every pointer assignment	No
Mark-and-sweep	Handles cycles, classic algorithm	STW pause, complex implementation	**Yes**
Hybrid	Best of both worlds	Very complex, difficult to implement correctly	No

Integration with Current Architecture:
The garbage collector would be a separate component that interoperates with the allocator:

1. **Allocation Tracking**: Modify `block_header_t` to include a `marked` flag for the mark phase. Maintain a separate list of all allocated blocks (the "allocation registry").
2. **Collection Interface**:

```c
void gc_start(void);                    // Initiate collection
void gc_mark_roots(void);               // Scan stack, globals, registers
void gc_mark_range(void* start, void* end); // Conservative marking
void gc_sweep(void);                    // Reclaim unmarked blocks
void gc_run(void);                      // Full collection cycle
```

3. **Conservative Root Scanning**:
   - Scan the call stack by examining each word as a potential pointer.
   - Scan global variables (using linker sections or registered root sets).
   - Scan CPU registers (architecture-specific inline assembly).
4. **Mark Algorithm**: For each potential pointer, check if it points to a valid block in our heap. If yes, mark it and recursively mark all pointers within that block's payload.
5. **Sweep Algorithm**: Walk through all blocks in the allocation registry. If a block is allocated but not marked, call `free()` on it. Clear marks on remaining blocks.
6. **Integration Points**:
   - `malloc()` adds new blocks to the allocation registry.
   - `free()` removes blocks from the registry.
   - The collector can be invoked manually (`gc_run()`) or automatically when memory is exhausted.

Common Pitfalls:

- ⚠️ **Pitfall: False pointers causing retention**
  Conservative scanning may interpret random stack data as pointers, preventing collection of actually dead blocks.
  Fix: Use page-aligned allocations and check pointer alignment to reduce false positives. Accept some retention as inherent to conservative GC.
- ⚠️ **Pitfall: Not handling interior pointers**
  If a program has a pointer to the middle of an allocated block (e.g., into an array), our check may not recognize it.
  Fix: When checking if an address is within our heap, check if it falls within any allocated block's payload range, not just block start.
- ⚠️ **Pitfall: Race conditions during collection**
  If the program continues running while marking occurs, pointers may change, causing either missed marking or use-after-free.
  Fix: Implement a "stop-the-world" pause using signals or by requiring the program to explicitly call `gc_run()` at safe points.

Implementation Guidance

> **Note**: These extensions are for advanced exploration. The starter code focuses on integrating the first and most practical extension— `realloc` —while providing skeletons for the others.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Realloc Implementation	In-place expansion with next-block check only	Full mmap remapping, shrinking with immediate return to OS
Buddy Allocator	Power-of-two sizes up to 1MB, simple array of free lists	Binary tree tracking split/merge, multiple memory pools
Debugging System	Basic canaries and magic numbers	Full DWARF parsing for stack traces, HTML visualization
Garbage Collector	Conservative mark-and-sweep, manual invocation	Incremental collection, generations, finalizers

B. Recommended File/Module Structure

```
my_allocator/
├── include/
│   ├── allocator.h          # Core API (adds realloc, calloc declarations)
│   ├── debug.h              # Debugging interface (tracing, stats)
│   └── gc.h                 # Garbage collection interface (optional)
├── src/
│   ├── allocator.c          # Core malloc/free/realloc/calloc
│   ├── os_mem.c             # OS interface (sbrk, mmap)
│   ├── block.c              # Block management (split, coalesce)
│   ├── free_list.c          # Free list operations
│   ├── strategies.c         # First-fit, best-fit, segregated lists
│   ├── buddy.c              # Buddy allocator (optional)
│   ├── debug.c              # Debugging implementation
│   ├── gc.c                 # Garbage collector (optional)
│   ├── thread_cache.c       # Thread-local caching
│   └── arena.c              # Per-thread arenas
└── tests/
    ├── test_realloc.c
    ├── test_buddy.c
    ├── test_debug.c
    └── test_gc.c
```

C. Infrastructure Starter Code

Complete realloc implementation (ready to integrate):

```c
// In src/allocator.c

#include "allocator.h"

#include "debug.h"

#include <string.h>  // For memcpy


void* realloc(void* ptr, size_t new_size) {

    // TODO 1: Handle NULL pointer - equivalent to malloc(new_size)

    if (ptr == NULL) {

        return malloc(new_size);

    }



    // TODO 2: Handle zero size - equivalent to free(ptr) and return NULL

    if (new_size == 0) {

        free(ptr);

        return NULL;

    }



    // TODO 3: Validate pointer came from our allocator

    if (!is_valid_pointer(ptr)) {

        // According to POSIX, realloc with invalid pointer is undefined

        // We could return NULL or abort; returning NULL is safer

        DEBUG_LOG("realloc: invalid pointer %p\n", ptr);

        return NULL;

    }



    // TODO 4: Get the block header and current user size

    block_header_t* header = get_header_from_ptr(ptr);

    size_t current_user_size = header->size & ~IS_MMAP_FLAG;  // Clear flag bit



    // TODO 5: Check for overflow in new size calculation

    int overflow = 0;

    size_t new_total_size = calculate_total_size(new_size, &overflow);

    if (overflow) {

        DEBUG_LOG("realloc: size overflow\n");

        return NULL;

    }
```

```
    // TODO 6: If new size equals current size, return same pointer
    if (new_size == current_user_size) {
        return ptr;
    }

    // TODO 7: Handle mmap blocks specially
    if (header->size & IS_MMAP_FLAG) {
        // For mmap blocks, we could use mremap on Linux or fallback to copy
        #ifdef __linux__
        void* new_ptr = mremap(header, current_user_size + HEADER_SIZE + FOOTER_SIZE,
                               new_total_size, MREMAP_MAYMOVE);
        if (new_ptr != MAP_FAILED) {
            // Update header and return new user pointer
            block_header_t* new_header = (block_header_t*)new_ptr;
            new_header->size = (new_total_size - HEADER_SIZE - FOOTER_SIZE) | IS_MMAP_FLAG;
            return (void*)((char*)new_header + HEADER_SIZE);
        }
        #endif
        // Fallthrough to allocate-copy-free if mremap fails or not available
    }

    // TODO 8: If shrinking, split the block and return freed portion
    if (new_size < current_user_size) {
        // Calculate minimum block size that can hold new_size
        size_t min_needed = get_total_block_size(new_size);
        size_t current_total = HEADER_SIZE + current_user_size + FOOTER_SIZE;

        // Only split if the remainder is large enough to be a free block
        if (current_total - min_needed >= MIN_BLOCK_SIZE) {
            block_header_t* new_block = split_block(header, new_size);
            if (new_block) {
                // The split_block function should have added new_block to free list
                // and updated header size
                return ptr;  // Original pointer still valid
            }
        }
```

```
    // If split failed or not worthwhile, keep current block as-is
    return ptr;
}

// TODO 9: If expanding, check if next block is free and combined size sufficient
if (new_size > current_user_size) {
    // Get pointer to next block
    block_header_t* next_block = (block_header_t*)((char*)header +
                              HEADER_SIZE + current_user_size + FOOTER_SIZE);

    // Check if next block exists (not beyond heap) and is free
    if ((char*)next_block < (char*)allocator_state.heap_base + allocator_state.heap_size) {
        // Validate next block has proper magic if debugging enabled
        #ifdef USE_MAGIC
        if (next_block->magic != MAGIC) {
            DEBUG_LOG("realloc: next block magic corrupted\n");
            // Fall through to allocate-copy-free
        }
        #endif

        if (!(next_block->allocated)) {
            size_t next_size = next_block->size & ~IS_MMAP_FLAG;
            size_t combined_total = HEADER_SIZE + current_user_size + FOOTER_SIZE +
                                    HEADER_SIZE + next_size + FOOTER_SIZE;

            if (combined_total >= new_total_size) {
                // Remove next block from free list
                free_list_remove(next_block);

                // Merge with current block
                header->size = current_user_size + HEADER_SIZE + next_size + FOOTER_SIZE;

                // Update footer of merged block
                footer_t* new_footer = (footer_t*)((char*)header +
                                       header->size + HEADER_SIZE - FOOTER_SIZE);
                new_footer->size = header->size;
```

```
                    new_footer->allocated = 0;


                    // If combined block is larger than needed, split it
                    if (combined_total - new_total_size >= MIN_BLOCK_SIZE) {
                        split_block(header, new_size);
                    }


                    // Mark as allocated
                    header->allocated = 1;


                    #ifdef USE_MAGIC
                    header->magic = MAGIC;
                    #endif


                    return ptr;  // Expansion succeeded in-place
                }
            }
        }
    }


    // TODO 10: Fallback: allocate new block, copy data, free old block
    void* new_ptr = malloc(new_size);
    if (new_ptr == NULL) {
        return NULL;  // malloc failed
    }


    // Copy the minimum of old and new sizes
    size_t copy_size = (current_user_size < new_size) ? current_user_size : new_size;
    memcpy(new_ptr, ptr, copy_size);


    free(ptr);


    return new_ptr;
}
```

D. Core Logic Skeleton Code

Buddy allocator skeleton:

```c
// In src/buddy.c

#include "allocator.h"


#define MAX_BUDDY_ORDER 12  // 2^12 = 4096 byte maximum block

#define MIN_BUDDY_SIZE 32   // Minimum allocation size


static block_header_t* buddy_free_lists[MAX_BUDDY_ORDER + 1];


static int size_to_order(size_t size) {

    // TODO 1: Round size up to next power of two

    // TODO 2: Find log2 of that size

    // TODO 3: Adjust for minimum buddy size

    // TODO 4: Return order (0 for MIN_BUDDY_SIZE, 1 for 2*MIN_BUDDY_SIZE, etc.)

}


static void* buddy_of(void* block, int order) {

    // TODO: Calculate buddy address using XOR with block size

    // Buddy size = (1 << order) * MIN_BUDDY_SIZE

    // return (void*)((uintptr_t)block ^ buddy_size);

}


void* malloc_buddy(size_t size) {

    // TODO 1: Convert request size to order

    // TODO 2: Check if order is within bounds

    // TODO 3: Look in free list for that order

    // TODO 4: If found, remove from free list, mark allocated, return

    // TODO 5: If not found, try higher order free lists

    // TODO 6: When taking from higher order, split recursively

    // TODO 7: If no blocks available, request more memory from OS

    // TODO 8: Initialize new large block and split as needed

}


void free_buddy(void* ptr) {

    // TODO 1: Get header from pointer

    // TODO 2: Mark as free

    // TODO 3: Check if buddy is free (using buddy_of())

    // TODO 4: If buddy is free, merge them

    // TODO 5: Repeat merging until buddy is allocated or maximum size reached

    // TODO 6: Add merged block to appropriate free list


```

```
}
```

Garbage collector skeleton:

```c
// In src/gc.c

#include "allocator.h"

#include <signal.h>


static block_header_t* all_blocks_head = NULL;

static int gc_enabled = 0;


void gc_register_block(block_header_t* block) {

    // TODO: Add block to global registry (all_blocks_head linked list)

}


void gc_unregister_block(block_header_t* block) {

    // TODO: Remove block from global registry

}


void gc_mark_block(block_header_t* block) {

    // TODO 1: If block already marked, return

    // TODO 2: Mark the block

    // TODO 3: Scan block's payload for pointers to other blocks

    // TODO 4: For each potential pointer, check if it points to registered block

    // TODO 5: Recursively mark found blocks

}


void gc_mark_roots(void) {

    // TODO 1: Get current stack pointer (architecture-specific)

    // TODO 2: Scan stack range conservatively

    // TODO 3: Scan global variables (need to know their addresses)

    // TODO 4: Register known root sets if provided by program

}


void gc_sweep(void) {

    // TODO 1: Walk all_blocks_head list

    // TODO 2: For each block, if allocated but not marked, call free()

    // TODO 3: Clear marks on remaining blocks

}


void gc_run(void) {

    if (!gc_enabled) return;



    // TODO 1: Pause all threads (simplified: assume single-threaded)
```

```
    // TODO 2: Mark phase: gc_mark_roots()
    // TODO 3: Sweep phase: gc_sweep()
    // TODO 4: Resume threads
}

void gc_enable(int enable) {
    gc_enabled = enable;
}
```

E. Language-Specific Hints

- **C11 features**: Use `_Alignas` and `_Alignof` for portable alignment handling in advanced implementations.
- **Thread-local storage**: Use `_Thread_local` keyword (C11) or `__thread` (GCC) for efficient thread caches.
- **Atomic operations**: Use `stdatomic.h` for lock-free statistics counters in debugging system.
- **Backtrace**: On Linux, use `backtrace()` from `execinfo.h` to get call stacks for allocation tracing.
- **Signal handling**: Use `sigaction()` to implement safe-points for garbage collection stop-the-world pauses.

F. Milestone Checkpoint

After implementing realloc:

```bash
# Compile with realloc support
gcc -DDEBUG -o test_realloc tests/test_realloc.c src/*.c -lpthread

# Run the test
./test_realloc
```

Expected output:

```
Test 1: realloc NULL equivalent to malloc... PASS
Test 2: realloc to zero size equivalent to free... PASS
Test 3: realloc same size returns same pointer... PASS
Test 4: realloc shrinking splits block... PASS
Test 5: realloc expanding uses next free block... PASS
Test 6: realloc fallback to malloc-copy-free... PASS
All tests passed!
```

Verification steps:

1. Use `heap_visualize()` before and after each realloc operation to see blocks split/merge.
2. Check that in-place expansion actually occurs by adding debug prints when coalescing succeeds.
3. Verify that memory contents are preserved after realloc (except truncated when shrinking).

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
`realloc` causes corruption when expanding	Not checking next block footer properly	Add debug prints to show next block header values before coalescing	Ensure you're calculating next block address correctly, considering footers
Buddy allocator never merges blocks	Incorrect buddy address calculation	Print addresses of block and computed buddy, verify they're adjacent	Use `BUDDY_OF(addr, size) = addr ^ size` for power-of-two aligned addresses
Garbage collector misses reachable blocks	Conservative scanning too restrictive	Log all pointers considered during mark phase	Check pointer alignment (8-byte), ensure you're scanning entire stack range
Statistics show negative allocations	Counter overflow or race condition	Use atomic operations for counters, add consistency checks	Implement `heap_validate()` that reconciles counters with actual heap walk

---

11. Glossary

> **Milestone(s):** Milestone 1 (Basic Allocator), Milestone 2 (Free List Management), Milestone 3 (Segregated Free Lists), Milestone 4 (Thread Safety & mmap)

This glossary provides precise definitions for the key terms, concepts, and system components used throughout this design document. Building a memory allocator involves specialized terminology that spans operating systems, data structures, and concurrent programming. Understanding these terms is essential for following the architectural decisions and implementation details presented in previous sections.

Think of this glossary as a **technical dictionary for the memory warehouse**. Just as a warehouse manager needs to understand terms like "inventory," "pallet," and "forklift," a memory allocator developer needs clear definitions for "heap," "fragmentation," and "coalescing." Each term represents a fundamental building block in the mental model of how memory is organized, tracked, and managed.

Terms and Definitions

The following table provides comprehensive definitions for the core terminology used throughout this design document. Terms are organized alphabetically for quick reference.

Term	Definition	Related Concepts
Allocation	The process of reserving a contiguous region of memory from the heap for use by a program. In this allocator, allocation is performed by the `malloc` function, which returns a pointer to the beginning of the usable memory region (payload). The allocator must track the allocation's size and status through metadata.	`malloc`, payload, block header, heap
Block	A contiguous unit of memory within the heap that contains both metadata and a payload area. Every block starts with a `block_header_t` structure that stores its size, allocation status, and linking pointers (if free). The payload area follows the header and is aligned according to `ALIGNMENT`. Blocks can be in one of three states: allocated (in use by the program), free (available for allocation), or wilderness (the special block at the end of the heap that can be expanded via `sbrk`).	`block_header_t`, payload, free list, splitting, coalescing
Boundary Tag	A pair of metadata structures placed at both ends of a memory block: a **header** at the beginning and a **footer** at the end. The footer is typically a `footer_t` structure containing a copy of the block's size and allocation status. Boundary tags enable **bidirectional coalescing**—the ability to merge a free block with either its preceding or following neighbor in constant time O(1) by examining adjacent footers and headers.	Header, footer, coalescing, `block_header_t`, `footer_t`
Bump Allocator	A simple memory allocation strategy that maintains a single pointer (`program_break`) to the next free address in the heap. Allocation increments this pointer by the requested size (plus metadata), and individual blocks cannot be freed independently. This is the starting point for Milestone 1, but it's later enhanced with free lists. Also called an "arena allocator" or "sequential allocator."	`sbrk`, program break, Milestone 1
Coalescing	The process of merging two or more physically adjacent free memory blocks into a single, larger free block. Coalescing reduces **external fragmentation** by combining small, unusable free gaps into larger, usable regions. The allocator performs coalescing immediately when a block is freed (`free` operation), checking both the previous and next blocks using boundary tags.	Boundary tag, external fragmentation, `coalesce_block`, free list
External Fragmentation	A form of wasted memory where free space is scattered throughout the heap in small, non-contiguous chunks, making it impossible to satisfy a large allocation request even though the total free memory is sufficient. External fragmentation occurs when allocated blocks of varying sizes are intermixed with freed blocks. Combated by **coalescing** and intelligent allocation strategies.	Internal fragmentation, coalescing, allocation strategy
First-Fit	An allocation strategy that scans the free list from the beginning and selects the **first** block that is large enough to satisfy the request. This strategy is simple and fast but can lead to increased fragmentation over time as small free blocks accumulate at the beginning of the list. One of several strategies selectable via `set_allocation_strategy`.	Best-fit, worst-fit, next-fit, allocation strategy
Free List	A data structure that tracks all free memory blocks in the heap. In our **explicit free list** implementation, free blocks are linked together using `next` and `prev` pointers stored within the payload area of the free blocks themselves (since that space is unused). The allocator maintains a `free_list_head` pointer to the first free block. Free lists enable reuse of freed memory and are central to Milestone 2.	Explicit free list, `free_list_insert`, `free_list_remove`, coalescing
Header	The metadata structure (`block_header_t`) placed at the very beginning of every memory block. The header contains essential information about the block: its total size (including metadata), allocation status (allocated or free), optional magic number for corruption detection, and pointers (`next`, `prev`) for linking in the free list (if the block is free). The user's returned pointer points to the memory immediately after the header.	`block_header_t`, footer, boundary tag, metadata
Heap	The region of a process's virtual address space used for dynamic memory allocation. The heap grows upward (on most systems) via the `sbrk` system call, which increases the **program break**. Our allocator manages this region as a collection of **blocks**, organizing them into a **free list** and tracking allocation metadata. The heap is distinct from the stack and static data regions.	`sbrk`, program break, virtual memory, block
Internal Fragmentation	Wasted memory space **inside** an allocated block that cannot be used by the application. This occurs when the allocator rounds up the requested size to meet alignment requirements, size class boundaries, or minimum block sizes. For example, if a program requests 30 bytes and the allocator returns a 32-byte block (due to size class rounding), 2 bytes are internally fragmented. Trade-off for simplified management.	External fragmentation, size class, alignment

Term	Definition	Related Concepts
mmap	A Unix/Linux system call (`memory map`) that maps files or anonymous memory into the process's address space. Our allocator uses `mmap` with the `MAP_ANONYMOUS` flag to allocate large blocks (above `SBRK_THRESHOLD`) directly from the operating system, bypassing the main heap. These `mmap` blocks are managed separately and can be returned to the OS immediately upon `free` via `munmap` .	`os_mmap` , `os_munmap` , large allocation, virtual memory
sbrk	A legacy Unix system call that adjusts the **program break**—the address of the first location beyond the current end of the heap. Calling `sbrk` with a positive increment expands the heap, providing more raw memory to the allocator. Our allocator uses `sbrk` as the primary method for obtaining memory for small and medium allocations (below `SBRK_THRESHOLD`).	Program break, heap, `os_sbrk` , bump allocator
Segregated Free List	An optimization where multiple free lists are maintained, each dedicated to a specific **size class** (e.g., 16, 32, 64 bytes). When an allocation request arrives, it's mapped to a size class index via `size_to_class_index` , and only the corresponding list is searched. This transforms allocation time from O(n) to O(1) for common small sizes, at the cost of **internal fragmentation**. Central to Milestone 3.	Size class, free list, `malloc_segregated` , internal fragmentation
Size Class	A category of allocation sizes grouped together for efficient management. Our allocator uses power-of-two size classes (16, 32, 64, 128, ... up to `LARGEST_SIZE_CLASS`). When a request falls within a size class range, the allocator rounds up to the class boundary and serves it from the corresponding **segregated free list**. Size classes enable fast lookup but cause **internal fragmentation**.	Segregated free list, internal fragmentation, `NUM_SIZE_CLASSES`
Splitting	The process of dividing a large free block into two smaller blocks: one that will be allocated to satisfy the current request, and a remaining free block. Splitting occurs when a free block is larger than needed (including metadata and minimum free block size `MIN_BLOCK_SIZE`). The allocator calls `split_block` to carve out the needed portion, update headers/footers, and insert the remainder into the free list.	Block, `split_block` , free list, external fragmentation
Thread-Local Cache	A per-thread pool of recently freed memory blocks that allows allocation and deallocation operations to proceed without acquiring global locks. Each thread has a `thread_cache_t` structure containing free lists for small size classes. The **fast path** for `malloc` and `free` checks the thread-local cache first; only on cache miss/maximum does it fall back to the locked global arena. Reduces **lock contention**.	Per-thread arena, lock contention, fast path, `thread_cache_malloc`

Expanded Terminology

Beyond the core terms above, several additional concepts appear throughout the design document that merit precise definition:

Term	Definition	Related Concepts
Allocation Strategy	The algorithm used to select which free block to allocate when multiple options exist. Our allocator implements several strategies: **first-fit**, **best-fit** (finds the smallest adequate block), **worst-fit** (finds the largest block), and **next-fit** (remembers where the last search ended). The strategy is selectable at runtime via `set_allocation_strategy`.	First-fit, best-fit, worst-fit, next-fit
Best-Fit	An allocation strategy that scans the entire free list to find the block whose size most closely matches (is greater than or equal to) the requested size, minimizing wasted space. This strategy can reduce external fragmentation but requires a full list scan and may leave many tiny fragments.	First-fit, worst-fit, fragmentation
Explicit Free List	A free list implementation where free blocks contain explicit `next` and `prev` pointers that link them together. Contrast with **implicit free lists** where free blocks are found by scanning all blocks sequentially. Explicit lists allow O(free blocks) search time rather than O(all blocks).	Free list, `block_header_t`, linking pointers
Footer	A `footer_t` structure placed at the end of a memory block, containing a copy of the block's size and allocation status. The footer enables **bidirectional coalescing** by allowing a block to examine the header of the following block (via its own footer's location + footer size) without needing to know that block's starting address in advance.	Boundary tag, header, coalescing, `footer_t`
Lock Contention	Performance degradation that occurs when multiple threads frequently attempt to acquire the same mutex simultaneously, causing threads to wait (sleep or spin) rather than making progress. High lock contention becomes a bottleneck in multithreaded programs. Our allocator mitigates this with **thread-local caches** and **per-thread arenas**.	Mutex, thread safety, per-thread arena, fast path
Magic Number	A known constant value (e.g., `MAGIC = 0xDEADBEEF`) stored in a block's header to detect memory corruption. During `free` or validation, the allocator checks that the magic number matches; if not, it indicates the header was overwritten (e.g., by a buffer overflow). Optional feature enabled by `USE_MAGIC`.	Corruption detection, `block_header_t`, debugging
Next-Fit	An allocation strategy similar to first-fit but remembers the position in the free list where the last search ended and starts from there for the next request. This can distribute allocations more evenly across the heap but may have slightly worse fragmentation than first-fit.	First-fit, allocation strategy, free list
Payload	The region of a memory block that is returned to the user program via `malloc`. The payload begins after the block header and is aligned to `ALIGNMENT` bytes. The user may use this entire region; any bytes beyond the requested size (due to rounding) constitute **internal fragmentation**.	Block, header, internal fragmentation, alignment
Per-Thread Arena	A private heap region assigned to a specific thread, containing its own free lists and metadata. Each arena (`arena_t`) reduces contention by allowing threads to allocate/free without synchronizing with other threads, except when an arena is exhausted and must grow via a shared lock.	Thread-local cache, lock contention, `arena_t`
Program Break	The address of the first location beyond the current end of the heap segment. The `sbrk` system call moves the program break, effectively expanding or contracting the heap. Our allocator's wilderness block sits at the program break and can be expanded via `sbrk` when more memory is needed.	`sbrk`, heap, wilderness block
Virtual Memory	An abstraction provided by the operating system that gives each process its own private address space, independent of physical RAM. The heap exists within this virtual address space; `sbrk` and `mmap` manipulate virtual memory mappings, which are later backed by physical pages by the OS kernel.	`mmap`, `sbrk`, address space, page fault
Wilderness Block	The special free block at the end of the heap, adjacent to the program break. When the allocator needs more memory and no suitable free block exists, it can expand the wilderness block via `sbrk` rather than creating a new block. This optimization reduces system call overhead.	Program break, `sbrk`, heap expansion
Worst-Fit	An allocation strategy that scans the free list to find the **largest** available block, then splits it to satisfy the request. This strategy aims to leave large remainders that may be useful for future large requests, but it can increase external fragmentation by consuming large blocks for small requests.	First-fit, best-fit, splitting, fragmentation

Conceptual Relationships

Understanding how these terms relate to each other is crucial for grasping the allocator's architecture:

- **Block** is the fundamental unit of memory management, containing a **Header** and optional **Footer** (together forming **Boundary Tags**).
- **Free List** tracks **Blocks** that are free, enabling reuse via **Allocation Strategies** (first-fit, best-fit, worst-fit).

- **Coalescing** merges adjacent free **Blocks** to combat **External Fragmentation**.
- **Splitting** divides large free **Blocks** to satisfy smaller requests, potentially creating **Internal Fragmentation**.
- **Segregated Free Lists** organize free **Blocks** by **Size Class** for fast allocation.
- **Thread-Local Caches** hold **Blocks** recently freed by a specific thread to reduce **Lock Contention**.
- **mmap** and **sbrk** are **OS Memory Interface** calls that provide raw memory to the allocator's **Heap**.

Common Confusions Clarified

> **External vs. Internal Fragmentation:** External fragmentation is wasted space *between* blocks (free but non-contiguous), while internal fragmentation is wasted space *inside* blocks (allocated but unused by the program). Our allocator trades internal fragmentation for speed via size classes, and combats external fragmentation via coalescing.

> **Header vs. Footer:** The header is at the *start* of every block and contains full metadata including pointers. The footer is at the *end* of every block and contains only size and status—it's a redundant copy that enables backward traversal for coalescing.

> **sbrk vs. mmap:** `sbrk` expands the main heap contiguously and is efficient for small allocations. `mmap` creates independent memory mappings elsewhere in the address space and is better for large allocations that can be returned directly to the OS.

> **Free List vs. Segregated Lists:** A single free list contains all free blocks of any size. Segregated lists are multiple free lists, each dedicated to a specific size class. Segregated lists are an optimization on top of the free list concept.

> **Thread-Local Cache vs. Per-Thread Arena:** A thread-local cache holds a small number of recently freed blocks for fast reuse. A per-thread arena is a complete private heap with its own free lists and metadata. Our design uses both: each thread has a cache, and multiple threads may share an arena.

This glossary provides the precise vocabulary needed to understand, implement, and discuss the memory allocator system. Refer back to these definitions when encountering unfamiliar terms in the design document or implementation guidance.
