COMP2221 Systems Programming Summative Coursework Report

[001139015]

1 Summary of Approach & Solution Design

1.1 Problem Definition

The task was to design a self-contained memory allocator for the Perseverance rover providing malloc, free, read, and write. Radiation storms may flip memory bits, so the design prioritises integrity, validation, and safe failure rather than speed. Returned payloads are 40-byte aligned relative to the initial heap pointer.

1.2 Metadata Structure

The `BlockHeader` (`allocator.c:41-75`) contains eight fields: a magic number, `size`, its bitwise inverse, request size, `status` flags (allocated `0x1`, quarantined `0x2`), a payload hash, a canary, and a checksum over the other fields. The footer (`allocator.c:81-88`) repeats the magic, mirrors the size and inverse size, and provides its own checksum.

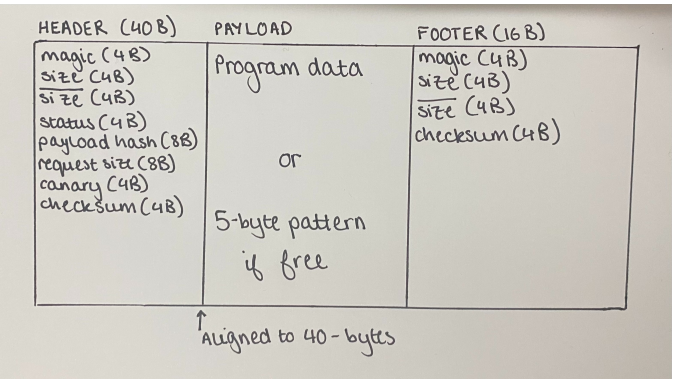

Fig. 1. Block structure showing Header(40B)—Payload—Footer(16B) with all integrity fields labeled.

1.3 Allocation Strategy

I implemented an implicit free list with first-fit placement. Although explicit free lists allow faster allocation, they rely on linked pointers which can become fragile during bit-flip events. The linear scanning approach in `mm_malloc` (`allocator.c:547-714`) skips quarantined or corrupted blocks safely. The `build_block()` function (`allocator.c:274-299`) initialises header, payload, and footer atomically to avoid partial writes.

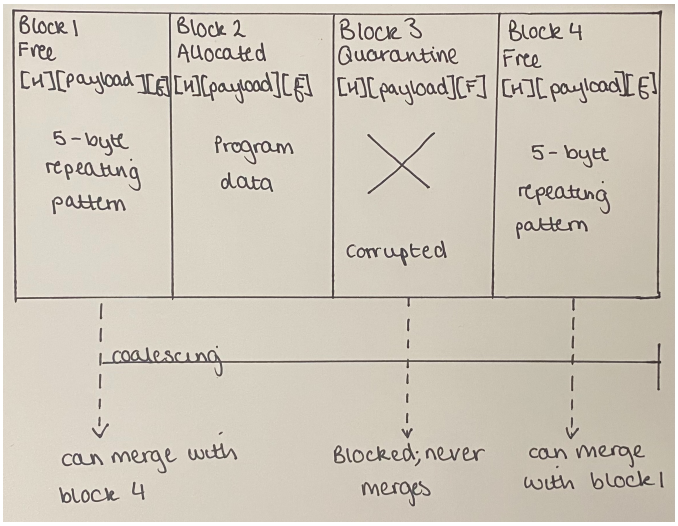

Fig. 2. Heap layout showing FREE, ALLOCATED, and QUARANTINED blocks with coalescing arrows.

1.4 Fault Tolerance

Corrupted blocks are quarantined using `quarantine_block()` (`allocator.c:434-458`) and permanently removed from allocation. Freed payloads are painted with the repeating 5-byte pattern detected during `mm_init()` (`allocator.c:35-37`, `56-57`, `106`) to detect stale writes. Adjacent free blocks coalesce via `coalesce_with_neighbors()` (`allocator.c:460-502`), while corrupted neighbours are quarantined rather than merged.

2 Analysis of Solution

2.1 Time Complexity

The `mm_malloc` function (`allocator.c:547-714`) performs $O(n)$ linear scan where n is the number of blocks that the memory is divided into. While slower than $O(1)$ free list lookup, this design avoids fragile pointer errors occuring under the bit flips. `mm_free` (`allocator.c:788-806`) operates in $O(1)$ for flag updates. `mm_read` and `mm_write` (`allocator.c:717-783`) are $O(p)$ due to hash verification.

2.2 Space Overhead

Each block incurs 56 bytes fixed overhead (40-byte header + 16-byte footer, `allocator.c:43-44, 78, 88`). For a 128-byte payload an overhead of $\sim$ 30% is required. However, for a 1024-byte payload, only $\sim$5% is required. This trade off prioritises corruption detection over space efficiency, appropriate for the Mars mission's robustness requirements.

2.3 Benchmark Results

Table 1 shows performance measurements on a 64KB heap (seed=1, 20,000 iterations).

TABLE 1
Allocator performance under storm conditions.

Condition	Ops/sec	Time/op
Clear skies (0 flips)	14,124	70.8 μs
Storm (8 flips/200 ops)	39,634	25.2 μs

Counterintuitively, under storm conditions the allocator yields a higher throughput; this is because the quarantined blocks reduce the scannable heap area. This demonstrates the quarantine mechanism successfully isolating damaged blocks while maintaining allocator functionality.

2.4 Fragmentation

Internal fragmentation is crippled by the 40-byte alignment (`allocator.c:28-44`), wasting up to 39 bytes per allocation, albeit a requirement for safety. However, external fragmentation can and is mitigated, by `coalescing_with_neighbors()` (`allocator.c:460-502`), though storms create permanent quarantine holes. Although this is accepted in the design to ensure the program doesn't use unsafe blocks. Although, quarantining damaged blocks also reduces external fragmentation by preventing repeatedly corrupted regions from re-entering the free list, as implemented in `quarantine_block()` (allocator.c:434–458).

3 Use of Generative AI, Tools, or other Resources

3.1 Generative AI Usage

I used Google's Gemini 3.0 Pro to help design the block layout structure and to develop a strategy for brownout recovery. Gemini suggested the redundant inverse size approach and the concept of scanning for valid footers to reconstruct corrupted headers and vice versa.

For code completion, I used Claude Sonnet 4.5 through VS Code Copilot. It assisted with basic syntax. However, suggestions and approaches often assumed standard `malloc()` was available, requiring modification for the heap passed-only constraint.

3.2 Non-AI Resources

Key non-AI resources included:

- Dev boot and TJ DeVries's YouTube tutorial on C programing and memory management [1] for foundational understanding of how the C language works and memory management in it
- Mishra's "Everything About Memory Allocators" tutorial [2] for malloc/free implementation patterns, header structures, and pthread mutex thread safety
- Course content and practicals on unix, c and makefiles

The video tutorial was greatly helpful to the development of this program as it enhanced my understand of C and equipped me with the tools to code the project. Mishra's written tutorial provided the malloc/free implementation details I adapted for the Mars scenario.

3.3 Independent Development

The most challenging aspect was brownout recovery logic in `recover_header_from_footer()` (`allocator.c:315-333`) and the quarantine mechanism (`allocator.c:434-458`) which were implemented manually after carefully designing an approach based off the specification's corruption scenarios. Approximately 80% of the final code was written without the assistance of AI.

4 Additional Functionality

4.1 Thread Safety

All public API functions are protected by a global mutex `g_lock` (`allocator.c:101-104`). The `LOCK()/UNLOCK()` macros wrap `pthread_mutex_lock` and `pthread_mutex_unlock`, adding approximately 50-110ns overhead per operation. This ensures expected behaviour when multiple processes are running on the rover concurrently.

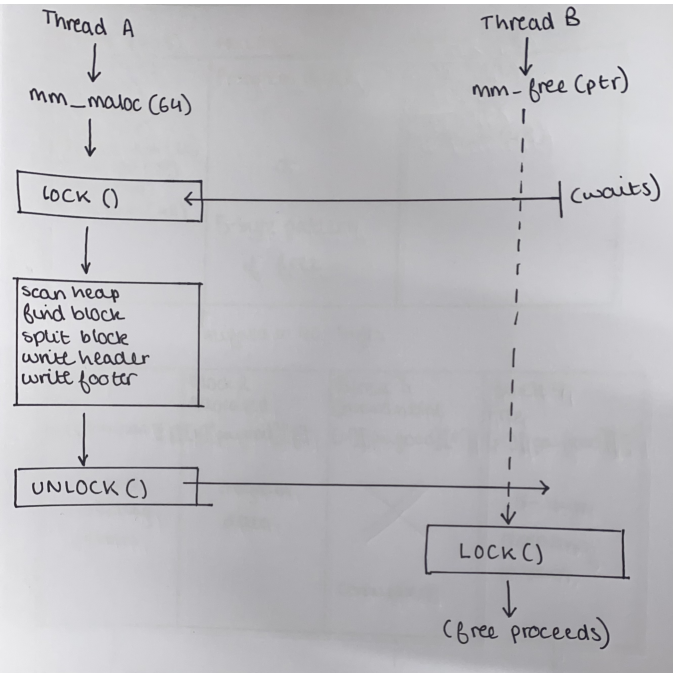

Fig. 3. Mutex protection around critical sections in API functions.

4.2 Realloc Implementation

`mm_realloc()` (`allocator.c:811-851`) handles growth, shrinkage, and NULL pointer edge cases. When growing, it allocates a new larger block, copies existing data via `memcpy()`, updates the payload hash for integrity, and frees the original block. When shrinking, it returns the same pointer if `new_size` fits within the existing payload, avoiding unnecessary memory operations. NULL pointer input is handled by delegating to `mm_malloc()`.

4.3 Test Harness

`runme_full.c` implements over 15 test cases covering normal operations, brownout simulation, payload corruption detection, double free checks, and random allocation and free fuzzing. The test suite also includes a benchmarking mode activated using the `--bench` flag, measuring operations per second under both clear-sky and storm conditions with configurable bit-flip intensity (`--bench-flips`). Which is equivalent to the intensity of the storm, this enabled the performance analysis presented in Section 2. Please note: only a minimal runme.c was uploaded to the autograder.

References

[1] Boot dev and TJ DeVries. (2025) C programming and memory management - full course. Accessed: 30-11-2025. [Online]. Available: https://www.youtube.com/watch?v=rJrd2QMVbGM

[2] M. Mishra. (2024) Everything about memory allocators: Write a simple memory allocator. Accessed: 29-11-2025. [Online]. Available: https://mohitmishra786.github.io/chessman/2024/11/24/Everything-About-Memory-Allocators-Write-a-simple-memory-allocator.html