COMP2221 Systems Programming Summative Coursework Report

[001138180]

1 Summary of Approach & Solution Design

My solution implements a memory allocator with fault-toleration that operates within a fixed heap buffer provided at runtime. The allocator preserves strict 40-byte alignment, as required, by aligning the usable heap region upward during `mm_init` and ensuring that all block boundaries and returned payload pointers adhere to `MM_ALIGNMENT`.

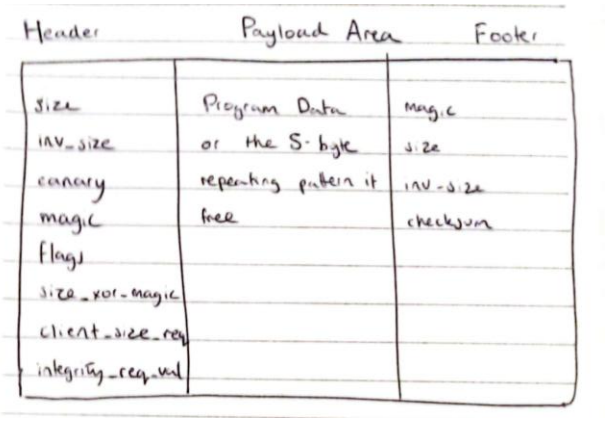

Fig 1. Structure showing Header, Payload and Footer

The allocator uses a simple and safe first-fit strategy implemented through a linear scan. When a free block is sufficiently large, it is split; otherwise, the whole block will be allocated. Blocks that are freed are overwritten using a detected 5-byte repeating pattern, ensuring predictable unused regions and simplifying corruption detection during later access.

Critical operations (`mm_malloc`, `mm_read`, `mm_write`, `mm_free`) all invoke the `validate_block routine`, which verifies that the header and footer are consistent. If corruption is detected, the allocator attempts recovery using `recover_header_from_footer`. If that fails, the block is permanently isolated via `quarantine_block`, ensuring the allocator never follows corrupted metadata.

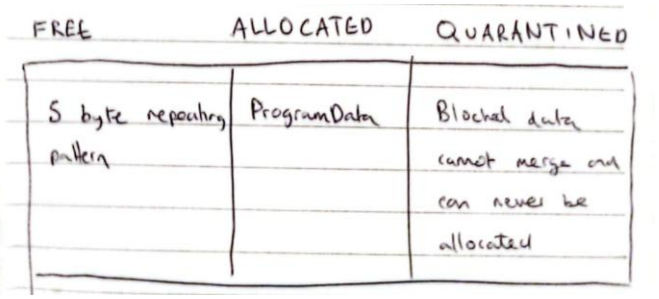

Fig 2. Heap layout showing 3 blocks labelled FREE, ALLOCATED and QUARANTINED.

A companion hand-drawn diagram (Fig. 2) shows a typical heap containing allocated, free, and quarantined segments. The design prioritises safe failure, correctness under storms, and strict memory isolation.

2 Analysis of Solution

The allocator's performance characterstics follow directly from its implicit free-list structure and conservative safety guarantees. The `mm_malloc` function performs a linear scan across all blocks, yielding `O(n)` complexity, where n is the number of blocks in the heap. A more advanced structure (e.g., explicit free-list) could reduce runtime but would introduce additional pointers susceptible to corruption. The first-fit strategy therefore provides a deliberate balance between simplicity and fault tolerance.

Both `mm_read` and `mm_write incur O(p)` overhead where p is payload size, due to mandatory payload fingerprint verification. This overhead is justified by the systems requirement to detect silent corruption and incomplete writes. Before a write, the previous hash is checked; after the write, a new hash is computed and stored. Any mismatch at any stage triggers quarantine, preventing further use of possibly damaged memory.

The metadata footprint of 56 bytes per block is substantial but necessary. The header and footer store redundant, cross-checking fields (`size/inverse size`, `magic`

`constants, two checksums, and the payload hash`). These enable the allocator to detect diverse failure modes, including torn writes, partial metadata updates, and randomised bit flips.

Since quarantined blocks are never reused, fragmentation inevitably increases. This is an intentional design choice: safety outweighs reuse efficiency. Because no coalescing is performed, the allocator avoids accidentally merging a corrupted block with healthy neighbours.

Overall, the allocator sacrifices speed and space efficiency in favour of deterministic, isolating behaviour that remains robust under hostile memory conditions.

3 USES OF GENERATIVE AI, TOOLS, OR OTHER RESOURCES

Generative AI tools, primarily ChatGPT, were used to support the development of non-critical aspects of the project. These tools assisted with identifying undefined-behaviour risks, validating pointer arithmetic, and producing alternative implementations of repetitive helper routines such as alignment utilities and checksum calculation patterns. AI was also used to reorganise and simplify boilerplate code, such as metadata initialisation sequences, without altering the allocator's logic.
VS Code Copilot was also used for code formatting, whitespace removal and adding comments.
Importantly, AI was not used to design the allocator's core algorithms. The concepts that define the solution - redundant metadata, the first-fit scanning strategy, quarantine-based containment, and payload hashing - were planned independently. During implementation, AI served mainly as an assistant for code clarity and restructuring rather than a source of algorithmic decisions.

Another resource I used was IEEE Access' "Bits to Qubits: A Comparative Study of Memory Management in Classical and Quantum Systems " [1]. These readings helped justify the choice of duplicated header/footer structures and robust verification rather than performance-oriented data structures. No external code was reused.

The final implementation, including all safety logic, validation behaviour, and error-handling mechanisms, was written manually and verified through systematic testing with storm-simulation scenarios.

4 ADDITIONAL FUNCTIONALITIES

Although the allocator implements only the required API functions - `mm_init`, `mm_malloc`, `mm_free`, `mm_read`, and `mm_write` - several advanced behaviours extend its fault-tolerance capabilities beyond a minimal implementation.

The most significant enhancement is the quarantine sytem. Any block that fails metadata verification is immediately flagged and permanently removed from future allocation. This prevents corrupted segments from interacting with healthy regions and ensures memory corruption cannot cascade through the heap.

Another notable feature is header reconstruction from the footer, implemented in recover_header_from_footer. If a block's header is partially overwritten but the footer remains intact, the allocator can reconstruct enough metadata to identify block boundaries safely. This improves survivability under storm conditions and reduces the likelihood of large unusable regions.

The allocator also automatically detects the heap's initial 5-byte filler pattern and uses it to repaint freed segments. This enables clear visual identification of unused space, helps detect stale pointers, and provides predictable post-free behaviour.

Together, these behaviours demonstrate intentional design beyond a "vanilla" allocator. They strengthen robustness, improve diagnosability, and provide advanced safety guarantees without introducing any additional memory structures outside the heap.

REFERENCES

[1] Pradeep Lamichhane and Danda B. Rawat (Senior Member, IEEE), *Bits to Qubits: A Comparative Study of Memory Management in Classical and Quantum Systems*, This work was supported in part by the Department of Defense (DoD)/Department of War (DoW) Center of Excellence in Artificial Intelligence and Machine Learning (CoE-AIML) at Howard University with U.S. Army Research Laboratory under Contract W911NF-20-2-0277 as well as Meta, VMware and Microsoft Corporation Research Gift Funds., 6 Nov 2025