COMP2221 Systems Programming Summative Coursework Report

001174242

1 Summary of Approach & Solution Design

1.1 Definition of Solution

This allocator is designed for embedded systems – surviving memory faults, maximizing heap memory, and recovering compromised blocks.

1.2 Memory block Structure

Blocks are 80 B minimum and align to 40 B. The header + padding is 40 B, resulting in a 40 B aligned payload pointer.

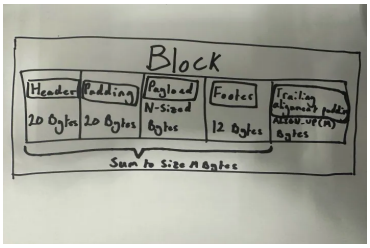

Fig. 1. The 5 components which make up a memory block

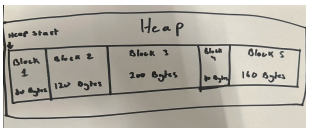

Fig. 2. Example showing block alignment in the heap – All blocks are a multiple of 40

1.2.1 Header and Footer structure

Header and footer mirror attributes, enabling 2-way recovery (see 1.3.4).

```
typedef struct __attribute__((packed)) {
  uint16_t magic;
  uint16_t checksum;
  uint32_t size;
  uint32_t payload_size;
  uint16_t payload_crc;
  uint8_t allocated;
  uint8_t size_class;
  uint32_t next_free_off;
} Header;

typedef struct __attribute__((packed)) {
  uint16_t magic;
  uint16_t checksum;
  uint32_t size;
  uint16_t payload_crc;
  uint8_t allocated;
  uint8_t pad;
} Footer;
```

1.3 Design Goals

1.3.1 Fast allocation

Segregated storage is employed with 9 size classes computed from heap size. Class selection is O(1) (See Fig. 5), and best-fit placement within each class minimizes allocation costs [1] [2].

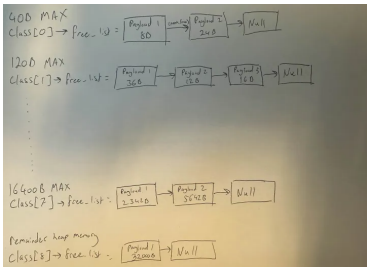

Fig. 3. Implicit visualization of `free_list[class]` for a 64 KB heap – (Note free lists use a LIFO approach during insertion)

1.3.2 Minimizing fragmentation

Coalescing occurs in `mm_free(...)` (`allocator.c 1297` and `703–750`), where adjacent free blocks are merged utilizing metadata. This maximizes allocatable size (see Fig. 4) [3] [2].

1.3.3 Block validation

All public allocator operations use `validate_block(...)` to detect bit flips through magic signature checks, sanity checks, and table-less CRC16 checksums (see fig. 6). We use CRC16 [4] to detect 99.998% of errors and store only 2 bytes in metadata (see 1.2.1).

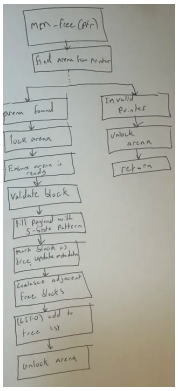

Fig. 4. Data flow diagram of `mm_free(...)`

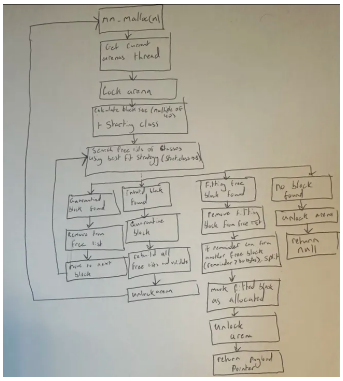

Fig. 5. Data flow diagram of `mm_malloc(...)` – See left 2 branches where validation occurs

1.3.4 Data recovery and containment

When block validation fails, header/footer recovery is attempted (`allocator.c 555–566`; Fig 6). If both header and footer are corrupted, the region is quarantined (see Fig. 7).

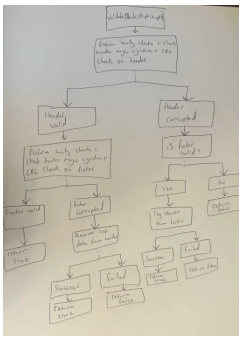

Fig. 6. Data flow diagram of `validate_block(...)`

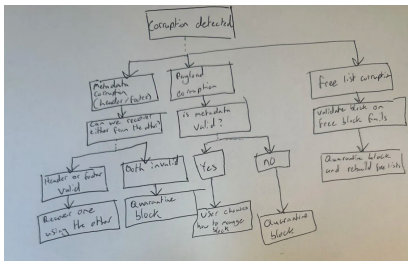

Fig. 7. Data flow diagram showing how bit flips are handled

1.3.5 Brownout recovery

If a payload is partially written during `mm_read(...)`, we free it to remove faults and maximize usable memory. The trade off is that writes can not be smaller than the payload size.

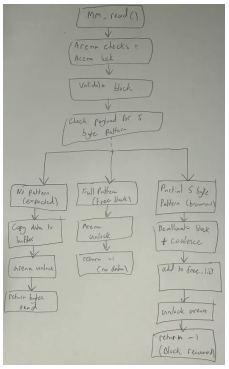

Fig. 8. Data flow diagram of `mm_read(...)` – See right branch where brownouts are dealt with.

2 Analysis of Solution

2.1 Overview

This section utilizes benchmark.c's results to evaluate the quantitative and qualitative impacts of my design. Results vary by system, but trends remain consistent.

2.2 Performance Speed

2.2.1 Allocation and Deallocation speed

Small allocations benefit from the class selection system in `mm_malloc(...)` (lines `1003-1080`), optimized for small workloads (see 1.3.1 and Fig. 5).

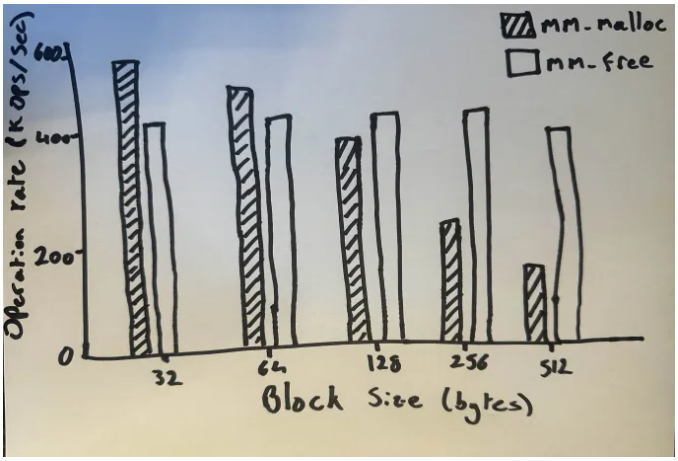

Fig. 9. Allocation and deallocation against the rate of operations. Performance of `mm_malloc(...)` decreases with larger blocks due to best-fit searches across larger classes. `mm_free(...)` remains relatively constant due to coalescing occurring once per free. **BAR HEIGHTS ARE APPROXIMATE AND NOT ALIGNED TO SCALE, TREND REMAINS**

The 512 B blocks show a $\sim$ 68% operation rate drop-off from 32 B in performance due to deeper best-fit traversal; however, embedded systems rarely perform high-rate large allocations. We trade large allocation speed for small allocation speed while keeping `mm_free(...)` stable, see section 1.1.

2.2.2 Read and Write speeds

Read/write operations require additional computation from CRC16 checks and branching sanity checks (`allocator.c 356-381`) see 1.3.3 and 1.3.4. This is the validation cost.

TABLE 1
Read/write operation rate in MB/s. The gap between read and write shows the cost of CRC updates. The gap vs memcpy is the total cost of validation.

Size (B)	mm_read	mm_write	memcpy
32	38.3	16.8	13,491
64	56.1	24.0	27,138
128	73.0	30.1	43,344
256	86.3	34.5	78,895
512	94.9	37.2	102,116
1024	99.9	38.9	119,933

Reads outperform writes as writes recompute CRC metadata, see `allocator.c 1492-1494`. We trade memcpy-tier speed for guaranteed fault containment.

2.3 Memory Efficiency and Longevity

2.3.1 Block space efficiency

Header and footer blocks allow for CRC validation, bidirectional recovery, and brownout detection for fault tolerance (`allocator.c` lines `409-466`); see section 1.2.

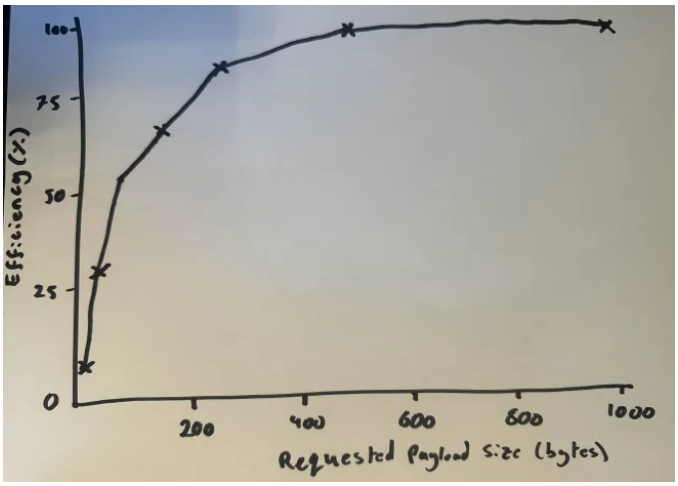

Fig. 10. Efficiency (computed as payload size over by block size) against payload size. 32-byte metadata (20B header + 12B footer) for cross recovery. Requests above 64B exceed 50% efficiency; 1KB+ requests achieve above 92.6%. **PLOTTED POINTS ARE APPROXIMATE AND NOT ALIGNED TO SCALE, TREND REMAINS**

Larger payloads scale better, exceeding 90% efficiency. We trade small block space inefficiency for time efficiency and strong corruption resilience.

2.3.2 Rigorous usage

The allocator has been designed for long-term stability – Both in normal use and extreme environments.

TABLE 2
Perpetual workload: normal vs radiation storm. Both tests run 5M operations (40% alloc, 30% free, 15% read, 15% write). Storm test injects 100 bit flips every 10K ops and 1 brownout every 50K ops. *Note: Only 16 quarantined regions appear, as that's the maximum we store to traverse faster by skipping quarantined regions (`allocator.c 106-118`)*

Metric	Normal	Under Storm
Starting memory	1,048,476 B	1,048,476 B
Total operations	5,000,000	5,000,000
Bit flips injected	0	49,900
Brownouts injected	0	99
Allocations	1,750,929	1,751,774
Frees	1,749,934	1,748,276
Reads	750,399	747,809
Writes	748,734	747,093
Corruption detected	0	5,044
Quarantined regions	0	16
Corrupt blocks (healed)	0	1,270
Memory recovered	1,048,476 B	683,932 B
Recovery rate	100%	65.22%
Final heap status	**OK**	**Corrupt block found**

Under normal conditions, all 5 million operations complete gracefully. Under storm conditions, over 5000 corruptions are detected; however, header-footer mirroring, CRC reconstruction, and brownout overrides keep over 60% of the heap usable (See section 1.3.4 and Fig. 7). We quarantine unrecoverable blocks – a necessary trade-off to maintain system safety and long-term stability.

3 Use of Generative AI, Tools, or other Resources

3.1 Overview

Generative AI was used to support development and debugging as per coursework guidelines. All allocator code was produced by me.

3.2 Use of Generative AI (ChatGPT)

3.2.1 Code Refactoring and Reconstruction

AI helped transition from a single-heap allocator to a multi-arena allocator, rewriting function signatures and function calls scattered throughout the code.

3.2.2 Arena logic assistance

Basic arena locking logic was learned through AI. It also helped with reasoning cross-arena pointer hazards, telling me where to implement debug warnings.

3.2.3 Test and Benchmark Generation

AI assisted with generating the arena test-suite and the benchmark tests for Table 2 and Table 3. Other tests written by me, originally only outputting numeric data, had informative labels and console sectioning generated.

3.2.4 Commenting support

I couldn't meet the comment-to-code ratio alone due to the code base size. All comments in runme.c and benchmark.c are AI-generated with minor tweaks by me. Just over 50% of the comments in allocator.c and allocator.h are AI-generated. AI comments have been checked by me to ensure maintainability.

3.2.5 Research Assistance

AI helped with finding references such as ptmalloc internals [2] and allocator-design papers [1] [3] [4], pointing out key design structures in the code-base/papers.

3.2.6 Additional suggestions

`mm_heal(...)` was inspired during conversations about benchmarks.

3.3 Other Tools Used

Cpplint was used to adhere to Google C standard guidelines. Existing allocators [2] informed naming patterns, design features, and additional API (see section 4.3).

3.4 Limitations

AI was not useful for code generation within the allocator. It produced broken functions and didn't consider the underlying design. It explained design approaches well, but couldn't implement them.

4 ADDITIONAL FUNCTIONALITY

4.1 Overview

A thread-safe multi-arena system and an extended API for enhanced usability, diagnostics, and heap repair. These additions add robustness and parallel performance.

4.2 Thread safety and Speed

4.2.1 Arena Locking

Each arena is assigned using thread-local storage. Locking remains necessary as multiple threads can share the same arena, which contains free_lists and header/footer metadata. Concurrency risks corrupting metadata. (see Sections 1.3.3-1.3.5 & `allocator.c` API to see locks).

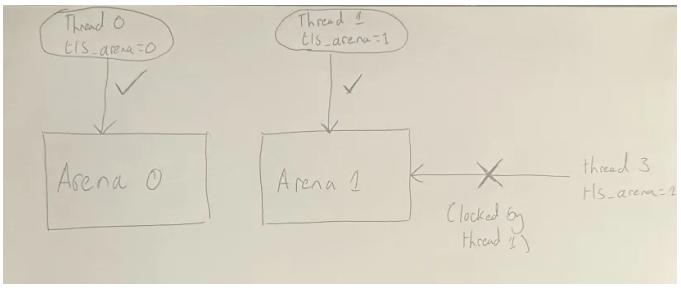

Fig. 11. Example of threads trying to operate on the same arena

4.2.2 Cross-Arena checking

Cross-Arena pointers are resolved using the `find_arena_for_ptr(...)` method (`allocator.c 182-196`). Frees, reads, and writes are applied safely under the arena's lock, keeping operations from threads with a different TLS arena safe.

4.2.3 Performance

TABLE 3
Operation Rate and measured speed increase under per-arena locking. $OperationRate = OperationRateOfAllArenas \div RuntimeOfSlowestThread$

Arenas / Threads	Operation Rate (K ops/sec)	Speed increase (×)
1	510	1.00
2	869	1.70
4	1120	2.19
8	2551	4.93

Despite locking arenas, a considerable speed increase is observed with more threads (up to 4.93x). TLS selection with cross-arena functionality and arena-local locking gives high concurrency while preventing races. Again, we trade optimal parallelism for correct, safe, predictable behavior.

4.3 Additional API

4.3.1 Usability

`mm_calloc(...)` and `mm_realloc(...)` are implemented, offering safe, system-universal, zero-filled initialization and fully validated resizing.

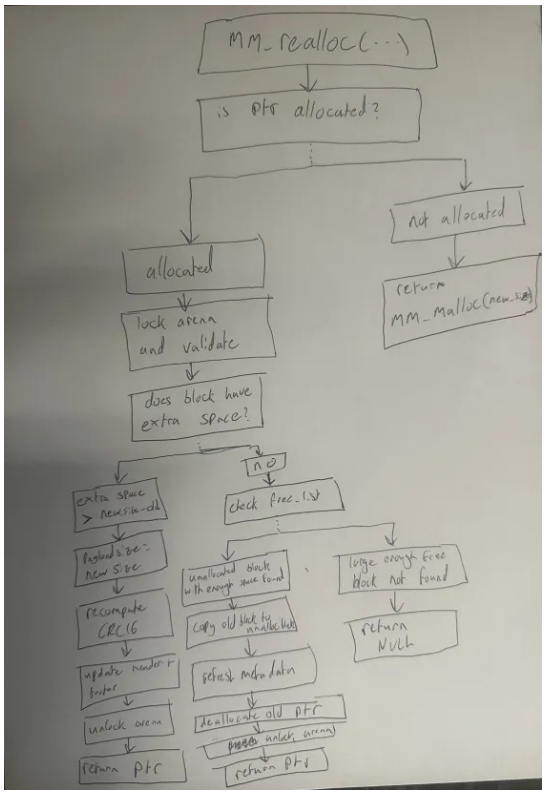

Fig. 12. `mm_realloc(...)` (`allocator.c 1506-1569`) Data flow Diagram

4.3.2 Diagnostics

The `Mmstatus` enum error codes are returned during `mm_arena_check(...)` and `mm_heap_check(...)`, validating the integrity of every block. Codes returns also happen for diagnostic deallocation (`mm_free_checked(...)`) and single block validation (`mm_validate(...)`). Example error codes include indicating invalid pointers or quarantined blocks.

Arena diagnostics include `mm_ptr_arena(...)` to find an arena from a pointer and `mm_is_cross_arena(...)` to identify potential cross-arena deallocations.

4.3.3 Repair

`mm_heal()` and `mm_arena_heal(...)` scan every block in memory, skipping, quarantining or repairing them. Partial repairs also occur during `mm_validate(...)`, isolating unsafe regions while maintaining heap integrity.

4.4 Optimized compilation

Compilation occurs with the `-O2` optimization flag set, trading slightly longer build times for enhanced runtime performance. (`Makefile 4`).

REFERENCES

[1] E. Auvinen, "Embedded dynamic memory allocator optimisation," Master's Thesis, Tampere University, Tampere, Finland, 2022. [Online]. Available: https://trepo.tuni.fi/bitstream/10024/140229/2/AuvinenEetu.pdf

[2] glibc Developers, "ptmalloc: malloc implementation in the gnu c library," https://sourceware.org/glibc/wiki/MallocInternals, accessed: 2025-02-10.

[3] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," *Memory Management*, pp. 1–116, 1995.

[4] R. N. Williams, "A painless guide to crc error detection algorithms," 1993, online document; describes both bitwise and table-driven CRC implementations.