COMP2221 Systems Programming - Summative Coursework

Student ID: **[YOUR STUDENT ID]**

Section 1: Summary of Approach & Solution Design

Problem Definition

The allocator manages memory for Mars Rover systems facing radiation storms (random bit-flips) and brownout events (power loss during writes). Unlike standard malloc implementations assuming reliable hardware, this solution designs defensively, assuming failure will occur.

Core Design

- **Implicit free list** with boundary tag coalescing for block management (`allocator.c:330-375`)
- **Quadruple-redundant size storage** enabling majority voting recovery from corruption
- **Three-state commit protocol** detecting interrupted write operations (`allocator.c:65-67`)
- **Block quarantine system** permanently isolating corrupted memory (`allocator.c:79-100`)

Data Structures

The Header (40 bytes, `allocator.c:38-50`) contains: magic number (0xDEADBEEF), checksum, size with backup copy, allocation flag, payload checksum, write state, and requested size. The Footer (24 bytes, `allocator.c:52-60`) contains: magic (0xCAFEBABE), checksum, and redundant size copies for boundary tag coalescing.

Fault Resilience Mechanisms

- Rotational XOR checksums with prime-number bit rotations detect single bit-flips with >99% probability (`allocator.c:122-145`)
- Three-state protocol: STATE_UNWRITTEN → STATE_WRITING → STATE_WRITTEN tracks write completion
- Corrupted blocks quarantined rather than recovered, preventing cascade failures

Memory Layout

Each block follows: [Header][Padding][Payload][Footer]. Padding (0-8 bytes) ensures 40-byte payload alignment calculated at `allocator.c:300-310`. Minimum data area of 48 bytes prevents unusably small fragments.

INSERT HAND-DRAWN FIGURE 1 HERE

Block structure showing Header (40B) / Padding / Payload / Footer (24B)

Label all fields and sizes

Figure 1: Memory block structure with quadruple-redundant size storage

INSERT HAND-DRAWN FIGURE 2 HERE

Three-state write protocol state machine

Show: UNWRITTEN → WRITING → WRITTEN with brownout detection branch

Figure 2: Three-state commit protocol for brownout detection

Word count: ~248/250 (excluding figures and captions)

Section 2: Analysis of Solution

Time Complexity Analysis

Function	Best Case	Worst Case	Dominant Operation
mm_malloc	O(1)	$O(n^2)$	Free block search + coalescing retry
mm_free	O(n)	O(n)	Immediate coalescing scan
mm_read/write	O(k)	O(k)	Payload checksum (k=payload size)

Table 1: Time complexity analysis (n=blocks in heap, k=payload size)

Code references: malloc search (`allocator.c:400-430`), coalescing (`allocator.c:330-375`), checksum computation (`allocator.c:147-158`).

Space Overhead Analysis

Allocation Size	Block Overhead	Overhead %
48 bytes	~72 bytes	60%
256 bytes	~72 bytes	22%
1024 bytes	~72 bytes	7%

Table 2: Fixed per-block overhead (40B header + 24B footer + 0-8B padding)

Key Trade-offs

- **Quadruple size redundancy** (+16 bytes/block): Enables majority voting recovery; justified for radiation-prone environment (`allocator.c:175-195`)
- **Payload checksums** (O(n) per access): Detects data corruption, not just metadata; essential for scientific data integrity (`allocator.c:147-158`)
- **Immediate coalescing** (O(n) per free): Prevents fragmentation accumulation; trades speed for memory efficiency (`allocator.c:330-375`)
- **Block quarantine** (memory loss): Guarantees corrupted data never returned; conservative but safe for Mars mission (`allocator.c:92-100`)
- **Implicit free list** (O(n) search): Fewer pointers mean fewer corruption points; simplicity improves reliability

INSERT HAND-DRAWN FIGURE 3 HERE
Graph: X-axis = Allocation size (48B to 4KB), Y-axis = Overhead % (0-70%)
Show decreasing curve from 60% to ~2%

Figure 3: Memory overhead percentage decreases as allocation size increases

Word count: ~247/250 (excluding tables and figures)

Section 3: Use of Generative AI, Tools, or other Resources

Development Tools

GCC Compiler (`-Wall -Wextra -Werror`)

- Caught implicit integer conversions between `size_t` and `uint32_t`
- Identified potential buffer overflows in checksum loops
- Added explicit casts at `allocator.c:130, 142, 156`

GDB Debugger

- Traced `find_block_header()` pointer resolution issues
- Discovered padding calculation error; fixed at `allocator.c:305-310`
- Watched write_state transitions during brownout simulation

Valgrind

- Verified no memory leaks in test harness
- Detected uninitialised reads during development
- Limited use since we're implementing malloc itself

Reference Materials

- **"Computer Systems: A Programmer's Perspective"**: Boundary tag coalescing pattern, implicit free list concepts
- **Database transaction theory**: Three-state commit protocol inspired by write-ahead logging
- **OWASP guidelines**: Integer overflow prevention, bounds checking patterns

Generative AI Usage

[WRITE YOUR HONEST AI USAGE HERE - Examples below:]

- **[If used: "AI assisted with explaining checksum algorithm options and debugging specific validation failures. Suggestions assumed reliable hardware, requiring significant modification to add fault tolerance."]**
- **[If not used: "No generative AI tools were used. Solution developed using lecture materials, textbook references, and manual debugging."]**

Reused Elements

Element	Source	Modification Made
Boundary tags	CS:APP textbook	Added redundant size fields
XOR checksums	Algorithm theory	Added prime rotations
Commit protocol	Database concepts	Adapted for memory writes

Table 3: Elements adapted from external sources

Word count: ~244/250 (excluding table)

Section 4: Additional Functionality

Feature 1: mm_realloc (`allocator.c:800-870`)

Resizes allocations while maintaining fault tolerance guarantees:

- NULL pointer returns `mm_malloc(new_size)`
- Zero size calls `mm_free(ptr)`, returns NULL
- **In-place optimization**: If new size fits current capacity, returns same pointer without copying
- **Full write protocol**: New blocks use three-state commit during data migration
- If brownout occurs during copy, corruption is detectable; old block remains valid until free completes

Feature 2: Block Quarantine System (`allocator.c:79-100`)

Corrupted blocks permanently isolated rather than recovered:

- Array stores up to 64 quarantined block pointers (`MAX_QUARANTINE`)
- `is_quarantined()` checks before every operation (`allocator.c:92-100`)
- Triggers: invalid magic, checksum mismatch, brownout state, size disagreement
- **Rationale**: Recovery attempts could return partially corrupted data; for Mars scientific data, isolation is safer than uncertain recovery

Feature 3: Advanced Checksum Algorithm (`allocator.c:122-158`)

Rotational XOR with prime-number bit rotations:

- Different seeds prevent collision: header (0x5A5A5A5A), footer (0xA5A5A5A5), payload (0x12345678)
- Prime rotations (3,5,7,11,13,17,19,23,29 bits) maximize bit dispersion
- **Critical design**: `write_state` excluded from header checksum (`allocator.c:133`), enabling brownout detection even when checksum appears invalid

Feature 4: Statistics Reporting (`allocator.c:880-927`)

Runtime introspection reporting: heap range, allocated bytes, corruption count, quarantine count, free block statistics. Enables ground control debugging and performance monitoring.

INSERT HAND-DRAWN FIGURE 4 HERE

Realloc decision flowchart: ptr==NULL? → malloc. size==0? → free.
size≤capacity? → return same. Else → allocate, copy, free old

Figure 4: mm_realloc decision flowchart

INSERT HAND-DRAWN FIGURE 5 HERE

Heap with blocks, some marked corrupted (X), arrows to quarantine list

Figure 5: Block quarantine system architecture

Word count: ~249/250 (excluding figures)