Durham University

Computer Science

Summative Assignment

Module code and title	COMP2221 Programming Paradigms
Academic year	2025-26
Coursework title	Systems Programming - Summative
Coursework credits	10 credits
% of module's final mark	50%
Lecturer	Stuart James
Submission date*	Thursday, 11 December, 2025 14:00
Estimated hours of work	20
Submission method	Gradescope

Additional coursework files	*Zip with report structure files*
Required submission items and formats	*A zip containing source files (.c / .h) a Makefile as well as report detailing implementation in the layout format provided (Word / Latex) submitted as a PDF.*

* This is the deadline for all submissions except where an approved extension is in place. For assessments carried out in weekly practicals or other scheduled sessions, the date shown will be the Monday of the week in which the assessments take place.

Late submissions received within 5 working days of the deadline will be capped at 40%.
Late submissions received later than 5 days after the deadline will receive a mark of 0.
It is your responsibility to check that your submission has uploaded successfully and obtain a submission receipt.

Your work must be done by yourself (or your group, if there is an assigned groupwork component) and comply with the university rules about plagiarism and collusion. Students suspected of plagiarism, either of published or unpublished sources, including the work of other students, or of collusion will be dealt with according to University guidelines (https://www.dur.ac.uk/learningandteaching.handbook/6/2/4/).

Use of AI
You are allowed to use AI to assist you with the development of the code in this coursework; however, solutions entirely generated with AI will affect your mark.
You are **NOT** allowed to use AI to write your report.

Surviving the Storm:

Robust Dynamic Memory Allocation on Mars

Scenario

The Perseverance mars rover is deep into its extended Mars exploration mission. Its onboard computer, stripped down to the essentials, runs a lightweight operating system that hands each mission program a single contiguous block of memory. This block is all you get, every experiment, every sensor module, every process that needs dynamic storage must operate within it. Normally, this setup works, but Mars has a way of finding weaknesses.

Recently, after a series of intense radiation storms, the rover's operating system began behaving erratically. Logs suggest that the memory allocator, the very component responsible for managing dynamic memory, has been corrupted beyond repair. Single-bit flips caused by high-energy particles have scrambled internal structures, leading to crashes, lost data, and inexplicable behaviour in running software. The onboard system is still responding to commands, but its original memory manager is gone.

From Earth, you have been tasked with writing a replacement memory management system, uploading it to the rover, and activating it remotely. The catch? You can only test it through a restricted, simulated environment that mimics the rover's conditions, and more extensive tests will run after your final solution due to time constraints. Your allocator must work flawlessly in normal operation but also withstand the chaos of future radiation storms, detecting damage early and preventing a total system meltdown. The mission's survival, and months of irreplaceable science, depend on your ability to design a memory manager that is lean, reliable, and battle-hardened against Mars' unforgiving environment.

Problem Definition

You are to design and implement a **robust, fault-tolerant malloc/free system** operating entirely within a **single contiguous block of memory** supplied by the rover's OS. This block is the only dynamic storage available for mission software, and all allocator metadata must live inside it.

The allocator must:

1. **Allocate and free memory** efficiently during normal operation.
2. **Detect signs of corruption** caused by bit-flips in both metadata and payload.
3. **Fail safely** when corruption is detected: no undefined behaviour, no unsafe pointer dereferences, and no corrupted block reuse.
4. **Recover where possible** by isolating or quarantining damaged memory regions.
5. Operate **without** using malloc, free, or any other standard C library allocation functions as this goes outside of the allocated memory.

Operational Scenarios

The autograder will simulate conditions the Mars rover allocator will face, however, some minimal tests will be provided during development:

- **Normal Operations (Clear Skies)**
  Sequential and mixed allocation/free requests with no corruption events. Tests base functionality, alignment, and fragmentation management.
- **Radiation Storms**
  Random bit-flips occur in heap memory, targeting both block metadata and payload. Elevated error rates test the allocator's ability to detect and contain corruption.
- **Brownout Events**
  Simulated power dips interrupt metadata writes mid-operation, leaving some data partially updated. Allocator/access functions must detect these inconsistencies.

Mission Constraints

- **Heap Size**: Provided by the OS; large enough to allow flexibility in design and error-checking strategies.
- **Alignment**: All returned pointers must meet the specified byte-alignment requirement.
- **Metadata Design**: You may choose your own data structure within the block of memory, integrity checks, and allocation policy.
- **Isolation**: Damaged blocks must not re-enter the allocation pool until explicitly repaired or quarantined.
- **No Host Heap**: You may not use dynamic allocation from the C standard library for additional; stack variables should not be used to store data.
- **Unused bytes should be clearly identified:** This should use the recurring 5-byte pattern provided, your allocator should detect this on initialization (mm_init), then when deallocated, reset it to make clear use of memory.

You are expected to implement the functionality within a **dynamic library**, while also providing a **runme executable** for your testing and validation.

1) Library API (required)

Create allocator.h / allocator.c implementing:

```
// Initialize the allocator over a provided memory block.
// Returns 0 on success, non-zero on failure.
int   mm_init(uint8_t *heap, size_t heap_size);
```

```
// Allocate a block with ALIGN-byte aligned payload. Returns
// NULL on failure.
void *mm_malloc(size_t size);

// Safely read data from an allocated block at offset bytes
into buf.
// Returns the number of bytes read, or -1 if corruption or
invalid pointer detected.
int mm_read(void *ptr, size_t offset, void *buf, size_t len);

// Safely write data into an allocated block at offset bytes
from src.
// Returns the number of bytes written, or -1 if corruption or
invalid pointer detected.
int mm_write(void *ptr, size_t offset, const void *src, size_t
len);

// Free a previously-allocated pointer (ignore NULL).
// Must detect double-free.
void  mm_free(void *ptr);
```

Optional (bonus) functions:

```
// Resize a previously allocated block to new_size bytes,
// preserving data. [See additional credit]
void *mm_realloc(void *ptr, size_t new_size);
// Output current heap usage and integrity statistics
// for debugging (No Credit, helper function).
void  mm_heap_stats(void);
```

2) Memory Requirements

- **Alignment**: All returned payload pointers must be aligned to **40 bytes**.
- **Correctness (clear skies)**: Normal allocate/free sequences work; coalesce adjacent frees.
- **Safety under storms**: The grader will flip **random bits** in your heap (sometimes during metadata writes). Your allocator must:
  - **Detect** inconsistencies (e.g., via canaries, checksums, mirrored header/footer, your choice).
  - **Fail safely**: never follow corrupted pointers; return NULL if the operation can't proceed; **quarantine** suspect blocks rather than reusing/merging them.
  - **Never crash** or access memory outside the provided heap.
- **Error detection:** implement memory error handling, such as double-free detection, safely (don't corrupt structures).

We're intentionally **not** prescribing the internal design; therefore, pick something reasonable and document it briefly in the design section of the report.

3) Test Executable (runme)

To help you develop and debug your allocator, you must include a small testing executable called runme. This program should act as a simple driver for your library, allowing you to exercise your allocator with your own tests before final submission. It will also be used by the autograder to check that your build works correctly.

Requirements for runme:

- Must be compiled by your Makefile (target name: runme).

- May take command-line arguments to control tests (for example arguments: --seed, --storm 1, ---size 8192).
- Should perform a single call to malloc (or calloc) to allocate the simulated heap block passed to mm_init.
- No further dynamic memory allocation is permitted inside your allocator.
- Must link against your library (allocator.c / allocator.h) and demonstrate calls to mm_init, mm_malloc, mm_free, and (if implemented) any additional functions.
- You may design your own test scenarios; examples include allocating and freeing blocks, simulating corruption, or printing heap state.
- Output should be readable and concise, e.g. printing allocation success/failure, corruption detection, or heap summary.

Purpose:
This executable lets you verify early that your allocator compiles, links, and behaves correctly before the autograder runs its full suite of tests on your library.

Hints

Here are a few hints to help get you started:

- Please **read the submission instructions carefully**. Since this is a programming assignment, deviations from the instructions might lead to you losing marks as a result of simple submission mistakes that can lead to your submission not running.

- Your **implementation should be as efficient** in its performance as possible, both in terms of memory and run-time.

- **Creating the right data structures** to manage memory can make your implementation better.

- **Implement error handling** and bounds checking to ensure that your program handles invalid input, doesn't go into an infinite loop or other such issues. Not only will this make your implementation easier to test for yourself, but it is also part of the marking scheme and can affect your marks.

- **Optimise your code for performance**, especially if you are dealing with large memory blocks. This might include algorithmic optimisations and careful memory allocation. Make sure your optimisation methods are explained in your report. Advanced optimisation methods will receive extra credit (if detailed in the appropriate section). Be mindful of memory usage, use the allocated memory efficiently.

- **Organise your code for easy reading**, while minimal functions are defined in the library header you should think about how to make the code easy to understand.

- To help debug, **the autograder provides feedback**. These are provided in the testing setting of images from the onboard camera. The images are indicative to give you an indication of the performance of your method but not quantitatively evaluable as they are compressed imagery.

- A **vanilla implementation should be quite easy to implement** you are therefore encouraged (and marks awarded) to go beyond the specification, thinking about how to implement novel algorithms or optimising your code in a novel way.

- Make sure you **submit your files well before the deadline** to make sure that system or network issues do not affect your submission. Every year, we receive dozens of

emails right after the submission deadline from students with their submissions marked as “late” due to a variety of reasons. Do not leave your submission to the final minutes and leave yourself plenty of time to deal with any issues that might affect your submission. Note that lecturers do not have the power to grant extensions and late submissions are handled by the office centrally.

- You can **submit as many times to the autograder as you like**, only the last submission will count!

- This is a large coursework in contrast to prior ones, equivalent to 10 credits. **The recommended time assumes** you have fully engaged with lectures, labs, recommended reading, and self-development. **If you have not spent the other 80 hours (of 100 hours), well**, then you will need to spend significantly more on the coursework, likely exceeding the sub-modules’ expected time. To aid this, 4 hours of lab time have also been optionally allocated to the coursework to provide space for development.

Code and Submission Specifications

- Your program **MUST** run on the autograder. This will be made available early for your testing. For a similar development environment run on Linux. A Docker image of Ubuntu 22.04 with

  `apt-get install -y gcc`

  Alternatively, you can use the University Linux system Mira. Note the Mira might occasionally experience downtime due to maintenance or other potential issues. **The deadline will not be extended due to issues with Autograder, Mira** or other university provided systems. Ample access is provided to submit. Failure for your code to work on the autograder may result in all marks being lost regardless of whether it compiles and runs on your own windows machine.

- You are allowed to use **ONLY** C for your implementation.

- You are **not allowed** to use any external libraries, but you can use all standard headers.

- **Only** use the headers that are needed. Do include what is not necessary, as that will affect your marks.

- You may have as many source files and headers files as you need - considering that your code should be organised well and easy to follow - but your program should be run via an executable called `runme`, which should be produced by your program. This executable should only be called `runme` and should not have any extensions.

- You will need to include a Makefile and your Makefile should include an `all` option, which compiles all the necessary source files and headers that you submit and creates an a library called ` liballocator.so` and a testing executable called `runme`. Our automated marking script will try to compile your program to get the executable by running the command `make all`.

- Your Makefile will be marked for functionality, clarity and correctness. Your Makefile should also contain `test` and `clean` options that respectively test the program with expected data and clean the compile environment.

- Your testing executable is primarily for your testing. Your program will be tested by running the general formulation `` `./runme –seed [int] --storm [int] --size [int]` ``

Submission

- Full program source code files for your final solution to the above task as a working C program, meeting the above *"**code and submission specifications**"* for testing.

- You must submit the following files:
  - All source and header files that are needed to compile your program.
  - A Makefile that has the `` `all` ``, `` `test` ``,`` `runme` `` and `` `clean` `` options and will compile your program and generate the `` `runme` `` executable.
  - All files **MUST** be in the root folder of a zip do **NOT** use folders, as this may result in your code not automatically running on the autograder.
  - Report (max. 4 pages in the defined sections with the limit of 1 page each and 250 words) detailing your approach to the problem and the success of your solution in the task specified. Provide **hand-drawn illustrative figures** and tables of the performance of your solution. Summarise the success of your system in performing the required task, speed during run-time and reducing memory requirements. In addition to describing any advanced features you may have added.
    Your report **MUST follow the provided structure**, with a minimum font size of 10 and without changing layout aspects (e.g., page borders).
    Submit a PDF (not in any other format). Your submission will be marked down if any file format other than a pdf is submitted as the report.
    Summary of section structure:
    1) Summary of Approach & Solution Design;
       - Definition of solution
       - Hand drawn solution design
       - Make specific reference to your code to justify solution
    2) Analysis of Solution;
       - Qualitative and quantitative evaluation of the trade-off for memory and performance.
       - Make specific references to your code to connect to your analysis.
    3) Use of Generative AI, Tools, or other Resources;
       - How you used tools in the development of your solution, how they helped or hindered you, and whether you reused elements in your solution.
    4) Additional functionality (i.e., extra credit evidence).
       - Definition of additional functionality solution(s)
       - Hand drawn additional functionality solution(s) design
       - Make specific reference to your code to justify additional functionality solution(s)

    Figures, tables and references do not count towards the 250 words section count. However, they should be used appropriately.
    You should follow approaches to reporting from your lab sessions. You are expected to provide hand-drawn figures to explain your solution; simply draw on paper and take a photo to include. Computer-generated diagrams and figures will be ignored.
    Note: Additional functionality will be evaluated in combination with the report and code; lack of detail or quality on either side will affect your mark.

Marks

The marks will be awarded as follows:

- Programmatic (Autograder) Evaluation:
  - Correctness of solution 10%
  - Storm Fault resilience 20%
    - Performance of your approach in terms of balance of run-time (speed) memory utilisation (bytes) measured based on mm_malloc / mm_free / mm_read / mm_write in different storm configurations.
  - Quality of the implementation
    - Clear, readable, well-documented (with sufficient comments) and well-presented program source code 5%
    - Error handling 10%
- Report:
  - Discussion/detail of solution design and choices made 10%
  - Quantitative evidence and analysis of performance 10%
  - Quality and clarity of hand-drawn diagrams 10%
  - Use of Generative AI, Tools, or other Resources 5%
  - Additional Credit Description
    (for any of the below, up to a maximum, dependent on quality) 20%

  Additional credit for one or more of, but not limited to, the following:
  - thread-safe code with consideration of performance impact;
  - implementation of realloc or other memory related functions;
  - use of advanced optimisation methods to improve performance;
  - superior performance in terms of speed and memory use.

  (Implementing one does not equal full marks, marks dependent on the quality of solution)

Total 100%

Plagiarism: You must not plagiarise your work. Attempts to hide plagiarism by simply changing comments/variable names will be detected. You should have been made aware of the Durham University policy on plagiarism.

You should not make your submission publicly available or share it with others as that will be considered plagiarism.

Submissions should be made through the Gradescope system. Delayed submissions are handled centrally and are not under the control of the lecturers.