PROJECT [OS] :: Managed Memory and Multiprocessing in a Faux File Hasher

Table of Contents

Overview

In this multi-stage assignment, you will implement and analyze a simplified file hashing program that explores two core operating system topics: memory management and process creation. You'll begin with a working sequential version that uses standard `malloc` and `free`. You will then replace these calls with your own allocator (`umalloc` and `ufree`) built on a contiguous memory region, implementing a basic **first-fit free list** strategy. Finally, you'll extend the program to fork multiple processes, each handling a separate block of a file, to produce the same deterministic 'signature.'

Your work will demonstrate how memory allocation and process isolation interact in a real system, and how careful resource management leads to consistent and verifiable results.

As a reminder: All grading and testing will be performed in a Linux environment. Your code must compile cleanly with `gcc` and produce identical output under Linux. Behavior differences on macOS or Windows are not grounds for exceptions.

Your Tasks

1. **Step 1: Baseline Version**
   - Use the provided code to compute a file signature serially.
   - Confirm correct output and memory safety (no leaks or invalid frees).
2. **Step 2: Implement a Custom Allocator**

    - Replace `umalloc` and `ufree` with your own versions that manage a fixed-size region allocated via `mmap()`.
    - Implement a *first-fit* free-list strategy using the structures provided (`header_t`, `node_t`).
    - Ensure your program produces identical signatures as the baseline version.
3. **Step 3: Multi-Process Version**
    - Extend your program so that each file block is processed by a separate child process.
    - Use pipes to collect partial hashes and combine them into a final signature in the parent process.
    - Demonstrate that your program's output remains deterministic and identical to the single-process version.

Background: Huffman Encoding For the File Signature

The program you'll be working with uses **Huffman encoding** as its internal workload — but you don't need to implement it yourself. The algorithm is **already provided** in the starter code. This section is meant to help you understand what the code is doing, so that when you debug or inspect memory usage, you can follow the logic confidently.

At a high level, Huffman encoding is a **compression technique** that assigns shorter bit patterns to more frequent symbols and longer patterns to less frequent ones. The result is a **binary tree**, where each leaf node represents a symbol from the input, and the path to that symbol (left or right edges) determines its code.

Here's how it works conceptually:

1. Count how many times each character appears in the input.
2. Treat each unique character as a leaf node, with its frequency as the node's weight.
3. Repeatedly take the two smallest-weight nodes and combine them into a new parent node whose weight is the sum of the two children.
4. When only one node remains, that's the root of the Huffman tree.

The code provided to you implements this process using a **min-heap** to efficiently find the two smallest nodes each time. Once the tree is built, it's traversed to produce a numeric "signature" (a hash in this case), rather than actual compressed data.

In this project, the **Huffman tree is not used for real compression** — it's used to generate a predictable, nontrivial workload that exercises dynamic memory allocation and deallocation. Each block of the input file produces its own tree, with dozens or hundreds of small `malloc()` calls, creating a realistic stress test for your allocator.

When you move on to implement your own allocator in Step 2, you'll be managing all of those allocations yourself. Understanding how the Huffman code allocates and frees memory (nodes, heaps, etc.) will make it much easier to debug your implementation and confirm that your memory manager is working correctly.

Understanding Huffman Coding

```
Huffman Coding is a compression algorithm that assigns shorter bit
patterns to more frequent symbols
and longer bit patterns to less frequent ones. It's widely used in
file compression formats like ZIP
and JPEG. The goal is to minimize the average number of bits used per
symbol.

The algorithm works by building a binary tree where each leaf node
represents a symbol, and the path
from the root to that leaf defines the symbol's code. Moving left
adds a 0, and moving right
adds a 1. More frequent symbols end up closer to the root, giving
them shorter codes.

Let's look at a small example. Suppose we have these characters and
frequencies:

A: 5
B: 9
C: 12
D: 13
E: 16
F: 45

The algorithm repeatedly combines the two lowest-frequency nodes
until only one node remains (the root).

1. Combine A(5) and B(9) → node (14)
2. Combine C(12) and D(13) → node (25)
3. Combine (14) and E(16) → node (30)
```

```
4. Combine (25) and F(45) → node (70)
5. Combine (30) and (70) → root (100)
```

Here’s the resulting tree (simplified view):

```
          [100]
         /     \
     [30]       [70]
    /   \      /   \
 [14]   E     [25]   F
 /  \         /  \
A    B       C    D
```

Now, tracing from the root:

```
A = 000
B = 001
E = 01
C = 100
D = 101
F = 11
```

Notice how F, the most frequent symbol, has the shortest code (11).

Your project’s implementation uses the same principles:
- Each node stores a symbol (character) and frequency.
- Internal nodes (non-leaves) store the sum of their children’s frequencies.
- The priority queue (often implemented with a min-heap) ensures that the two lowest-frequency nodes are combined first.

When debugging the code, pay attention to:
- How the nodes are compared in the priority queue.
- Whether leaf nodes and internal nodes are handled correctly during encoding and decoding.
- How bit sequences are accumulated during traversal.

The key insight is that Huffman trees are **prefix-free**: no code is a prefix of another.
This property guarantees that decoding is unambiguous, even without explicit separators.

Step 1: Implementing the Basic Version (Using malloc/free)

In **Step 1**, your goal is to make the program fully functional using the standard C memory allocator. You are **not** building a custom allocator yet — you are simply wiring together the provided components so the program runs and produces the correct output.

This step ensures you understand:

- How the Huffman tree is built and used.
- How the process model (single and multi) interacts with the tree builder.
- How modular accumulation of block hashes works.

What to Implement

umalloc() and **ufree()**
For now, these are simple wrappers around malloc() and free(). This lets all the provided code use umalloc/ufree transparently.
`void *umalloc(size_t size) { return malloc(size); }`

1. `void ufree(void *ptr) { free(ptr); }`
2. **process_block()**
   For each 1 KB block:
   - Count the frequency of each symbol (0–255).
   - Build a Huffman tree using `build_tree(freq)`.
   - Compute the hash of the tree using `hash_tree(root, 0)`.
   - Free the tree using `free_tree(root)`.
   - Return the resulting hash.
3. Your implementation does not need to compress or store anything. It only constructs the tree and verifies correctness via hashing.
4. **run_single()**
   Read the input file sequentially in `BLOCK_SIZE` chunks. For each chunk:
   - Call `process_block()` to compute the hash.
   - Print intermediate results using `print_intermediate()`.

    - Accumulate the final hash:
      `final_hash = (final_hash + h) % LARGE_PRIME;`
5. After all blocks are processed, print the final signature using `print_final(final_hash)`.
6. **run_multi()** (optional during Step 1)
   You may leave this unimplemented for now. The single-process version is sufficient for Step 1.

Testing Your Work

Compile and run your code to verify it behaves as expected. ***Note that you may use any binary or text file for testing — the program treats the input purely as raw bytes.***

```
gcc -DDEBUG=1 -o hashproj hashproj.c
./hashproj sample.bin
```

You should see one line per 1 KB block followed by a **Final signature** line. If you compile with `DEBUG=2`, you will also see process IDs for debugging.

Goal of Step 1

- A working single-process version that reads a file, builds Huffman trees, and prints deterministic block hashes.
- Clear understanding of how the provided code uses dynamic memory.
- Confidence that the Huffman and process logic work before implementing your own allocator in Step 2.

Step 2: Implementing a Simple First-Fit Allocator

In **Step 2**, you will replace the simple wrappers around `malloc()` and `free()` with your own **custom allocator** that manages memory inside a single `mmap()`-allocated region. The rest of the Huffman project remains unchanged — it will continue to call `umalloc()` and `ufree()`, but these will now allocate and free memory inside your own managed heap instead of using the system heap.

This step emphasizes practical memory management concepts:

- Managing a fixed-size heap inside your own address space.
- Tracking allocations and frees with headers and a free list.
- Implementing the **First-Fit** allocation algorithm.
- Handling freeing, coalescing, and integrity checking.

Overview

Your allocator will use the provided `init_umem()` function to allocate a contiguous 128 KB region using `mmap()`. Once initialized, all calls to `umalloc()` and `ufree()` must operate entirely within this region. You may not call `malloc()` or `free()` anywhere after initialization.

The provided helper structures will be used to manage blocks:

```
typedef struct {
    long size;    // Size of the allocated block (payload only)
    long magic;   // Integrity check pattern (MAGIC)
} header_t;

typedef struct __node_t {
    long size;                 // Size of the free block
    struct __node_t *next;     // Pointer to the next free block
} node_t;
```

Tasks

1. **Initialize the Allocator**
    - Call `init_umem()` once to obtain the 128 KB region.
    - Initialize a global `free_list` pointer to the start of that region.
    - Create one large free block covering the entire region (minus header space).
2. **Implement `umalloc(size_t size)`**
    - Round the requested size up to the nearest 8-byte multiple.

    - Start your search for a free block at the *current position* in the free list (not the beginning).
    - Find the **first block** large enough to satisfy the request.
    - If the block is larger than needed, split it into an allocated portion and a remaining free portion.
    - Mark the allocated block's header with `MAGIC` and return a pointer to the payload (just after the header).
    - If the search reaches the end of the list without finding space, wrap around and continue until you return to the starting point.
    - If no suitable block is found, return `NULL`.
3. **Note:** Unlike Best-Fit, First-Fit does not restart from the head of the list on each allocation. It continues searching from where it left off, making it faster on average but more prone to fragmentation.
4. **Implement `ufree(void *ptr)`**

    - If `ptr` is `NULL`, do nothing.
    - Subtract the size of `header_t` to find the start of the block header.
    - Verify that `magic == MAGIC`. If it doesn't match, print an error and call `exit(1)`.
    - Insert the freed block back into the free list in address order.
    - Attempt to coalesce adjacent free blocks to reduce fragmentation.
    - If the block is already free (double free), print an error and call `exit(1)`.

Algorithm Summary

The **First-Fit** algorithm scans the free list starting from the most recent position, allocating the first block large enough to satisfy the request. It then continues future searches from that point rather than returning to the start of the list.

- Fast in practice — avoids rescanning from the beginning.
- Simple to implement and easy to reason about.
- Can lead to internal fragmentation over time.

Testing Your Allocator

Recompile and test your allocator using the same procedure as in Step 1:

```
gcc -DDEBUG=1 -o hashproj hashproj.c
./hashproj sample.bin
```

The program's output should remain identical to Step 1's results. You are verifying correctness and stability, not changing the hash values.

Goal of Step 2

- Replace `malloc()` and `free()` with a working user-space allocator based on `init_umem()`.
- Implement a true **First-Fit** allocator that resumes its search from the last allocation point.
- Maintain correctness, coalescing, and integrity checks.
- Keep the Huffman project running identically — only the memory management strategy changes.

Step 3: Implementing the Multi-Process Version

In **Step 3**, you will extend your program to run in **parallel** using multiple processes. Instead of processing each file block sequentially, you will create a child process for each block, allowing multiple blocks to be processed concurrently.

This step builds on your understanding of memory management and introduces **process control** using `fork()`, `pipe()`, and `waitpid()`.

Learning Objectives

- Use `fork()` to create child processes.
- Use `pipe()` to communicate between parent and child processes.
- Synchronize and collect results with `waitpid()`.
- Manage resources safely across processes (no shared heap between children).

Overview

The `run_multi()` function will read the input file in 1 KB blocks (just as in `run_single()`), but each block will be processed by a new child process. The parent will coordinate work, collect results, and produce the final hash signature.

Each child runs `process_block()` independently — since it operates only on stack data and the file buffer, there is no shared state between processes.

Tasks

1. **Create Pipes**
   - Before calling `fork()`, create a pipe using `pipe(pipefd)`.
   - This provides two file descriptors:
     - `pipefd[0]` — read end (used by the parent)
     - `pipefd[1]` — write end (used by the child)
2. **Fork a New Process**
   - Call `pid = fork()`.
   - If `pid < 0`: print an error and exit (fork failed).
   - If `pid == 0`: this is the **child.**
   - If `pid > 0`: this is the **parent.**
3. **Child Process Responsibilities**
   - Close the unused read end (`pipefd[0]`).
   - Call `process_block()` on the buffer for this block.
   - Write the resulting hash to the pipe using `write(pipefd[1], &hash, sizeof(hash))`.
   - Close the write end and exit immediately (do not continue the parent loop).
4. **Parent Process Responsibilities**
   - Close the unused write end (`pipefd[1]`).
   - Read the hash value from `pipefd[0]` using `read()`.
   - Close `pipefd[0]` once the value is read.

    - Print the result using `print_intermediate(block_num, hash, pid)`.
    - Add the hash to the running total: `final_hash = (final_hash + hash) % LARGE_PRIME;`
    - Record the child PID so you can wait for it later.
5. **Synchronization**
    - After all blocks are processed, use `waitpid()` in a loop to ensure all child processes have finished.
    - Only after all children have exited, print the final result using `print_final(final_hash)`.

Hints

- Children inherit copies of the parent's address space, so each has its own heap and stack — your allocator is still used independently inside each process.
- Always close unused pipe ends to avoid deadlocks.
- Use `DEBUG=2` during testing to see each process ID in output.
- If a block is smaller than `BLOCK_SIZE` (end of file), it still needs to be processed.

Testing the Multi-Process Version

Compile and test your multi-process version using:

```
gcc -DDEBUG=2 -o hashproj hashproj.c
./hashproj sample.bin -m
```

You should see multiple PIDs in the output — one per block. The final signature should match the single-process version.

Goal of Step 3

- Implement **multi-process parallelism** using `fork()` and `pipe()`.
- Demonstrate safe inter-process communication via simple pipes.

- Preserve deterministic output regardless of process order.
- Understand that each process manages its own heap and memory state independently.

Important Guidelines and Restrictions

This project is designed to assess your understanding of memory management and process control, not your ability to rewrite or reorganize provided code. To ensure fairness, reproducibility, and automated grading, you must follow the rules below exactly.

Editing Policy

- **Do not modify any provided code** outside the clearly marked `STUDENT SECTION`. The only functions you are permitted to edit are:
  - `umalloc()`
  - `ufree()`
  - `process_block()`
  - `run_single()`
  - `run_multi()`
- The provided Huffman routines, data structures, `init_umem()`, `print_*` functions, and the `main()` dispatcher are part of the testing infrastructure. **They must remain unmodified.**

All of the provided code is wrapped inside a preprocessor guard:

```
#ifndef GRADING_MODE
    ...
#endif
```

- **You may not edit anything inside this region.** Doing so will break the autograder's ability to compare your results consistently. Any edits, even minor, to the code inside this block will result in a **significant grade penalty** because it requires manual inspection and regrading.

  Please take this seriously — in past semesters, students have **received failing grades** for modifying code within the grading region.
- Do not alter function prototypes, constants, or structure definitions.

- Do not add extra includes, global variables, or re-declarations above the student section.
- You **must** write **new** functions for STEP 2, include these in the student section.

Printing Policy

- All output in your final submission must be produced using the provided functions:
  - `print_intermediate(block_num, hash, pid)`
  - `print_final(final_hash)`
- You may use `printf()` or other debug prints while testing your code, but **they must be removed** before submitting.
- Unauthorized print statements (e.g., extra debugging text, formatted banners, or error messages not specified in the project) will cause automated tests to fail and result in a grade deduction.
- To confirm compliance, compile without `DEBUG` defined. The program should print **only** the final signature line.

Summary

- Edit **only** the designated student functions.
- **Never** modify or remove anything within the `#ifndef GRADING_MODE` region.
- Use only `print_intermediate()` and `print_final()` for visible output.
- Remove all debug output before submission.
- Modifying provided code or producing incorrect output formatting will result in substantial grade penalties, up to and including a failing grade.

General Guidance for Completing the Project

This project includes a substantial amount of provided scaffold code. That is intentional — the scaffold exists to help you get started quickly, focus your effort on the important parts, and ensure that everyone works from the same foundation. However, do not mistake the amount of provided code for simplicity. This project requires careful testing and incremental progress to succeed.

How to Begin

- **Step 0 — Verify Compilation:** Before changing anything, make sure the provided scaffold compiles cleanly on your system. Run:
  `gcc -o hashproj hashproj.c`
  If it does not compile, check that you haven't accidentally altered indentation, brackets, or preprocessor regions.
- Once you have a working build, read through the code carefully and trace the flow from `main()` through `run_single()` and `process_block()` to understand how data moves through the system.

Debug Flags
Note that there are specific DEBUG flags to help with each section. This is how you use them:

```
Step 1: gcc -DDEBUG=1 -o hashproj hashproj.c
```

```
Step 2: gcc -DDEBUG=1 -o hashproj hashproj.c
```

```
Step 3: gcc -DDEBUG=2 -o hashproj hashproj.c
```

Additional Headers
You may include small additional headers inside the student section (e.g., `<errno.h>`) if your allocator requires them, but do not add includes above the grading region.

Step 1: Basic Functionality

Start by implementing `process_block()` using the provided Huffman and hashing routines. This step uses `malloc()` and `free()` directly — your goal is to make sure that block processing and hashing work correctly before moving on.

- Test with small files first — the output should show block hashes and a final signature.
- If your output doesn't match expectations, add temporary `printf()` statements to verify that your frequency counting and hash calculations are correct.
- Once working, remove debug prints before moving forward.

Step 2: Custom Allocator

In this step, you'll replace your `malloc()` and `free()` wrappers with a simple **First-Fit allocator** using `init_umem()` and a free list. Expect this part to take time — debugging allocators can be tricky.

- Start by writing initialization code to set up your free list.
- Implement allocation and freeing one feature at a time.
- Use print statements (temporarily) to trace your free list and verify coalescing behavior.
- Watch for off-by-one errors, pointer arithmetic mistakes, or missing header fields — these are the most common bugs.
- If you encounter strange segmentation faults, suspect corruption of the `MAGIC` field or a misaligned pointer.

Step 3: Multi-Process Implementation

Finally, implement the `run_multi()` version, which forks one process per data block and uses pipes for communication. This step introduces concurrency, so you'll need to be careful about resource management.

- Always close unused pipe ends to prevent deadlocks.
- Ensure each child calls `exit(0)` after finishing — otherwise, it will continue running the parent loop.
- Check that all child PIDs are reaped using `waitpid()` before printing the final signature.
- Use `DEBUG=2` while testing to see which process handled each block.

Testing and Debugging with Valgrind

Once your allocator is functional, you should test for memory errors using **Valgrind**. It helps detect invalid reads, writes, and leaks. You can run your program through Valgrind like this:

```
valgrind --leak-check=full ./hashproj sample.bin
```

Valgrind is a great tool, but it’s not perfect — it may report false positives or behave oddly with custom allocators. Treat its output as a guide rather than absolute truth. If it flags a consistent error, it’s usually worth investigating.

Note: Valgrind does not work on macOS. If you are using a Mac, you will need to test your allocator in the Linux Lab or on a Linux virtual machine. This is non-negotiable — testing memory behavior without Valgrind is much harder. All grading will occur in a Linux environment. If your code compiles or behaves differently on macOS, that will not excuse incorrect results under Linux.

Final Advice

- Work incrementally. Do not jump ahead to Step 2 or Step 3 before Step 1 is correct.
- Test often and with small, controlled inputs.
- Keep a clean working version before major changes.
- Read compiler warnings carefully — they often point directly to logic bugs.
- Ask questions early if you get stuck; debugging memory issues is harder under deadline pressure.
- Your output must come from the program’s logic, not hard-coded constants or precomputed hashes.
- Readable, well-commented code helps ensure you receive full credit during manual review.
- You should not define `GRADING_MODE` yourself at any time. It is used internally by the autograder to isolate student code.
- You must not include any definitions in your submitted code of `GRADING_MODE` or `DEBUG.`
- As with all multi-step projects in this course, you are encouraged to compile your code with the flags `-Wall -Wextra` to catch potential issues early. Clean, warning-free builds make your debugging easier and help ensure your code behaves consistently during grading.