Master of Science in Informatics at Grenoble

UGA
UFR IM²AG Université Grenoble Alpes

OS Programming - Lab 2 :

Memory allocator design

Participants :
RIDAOUI Hatim & MORONI Louka

Platform description :

Platform 1:

Hardware Information:

- **Hardware Model:** Microsoft Corporation Surface Pro 7
- **Memory:** 8,0 GiB
- **Processor:** Intel® Core™ i5-1035G4 × 8

Software Information:

- **Firmware Version:** 19.101.140
- **OS Name:** Ubuntu 24.04.1 LTS
- **OS Type:** 64-bit
- **GNOME Version:** 46
- **Windowing System:** Wayland
- **Kernel Version:** Linux 6.10.10-surface-1

Platform 2:

Hardware Information:

- **Hardware Model:** Acer Predator Helios 300
- **Memory:** 16,0 GiB
- **Processor:** Intel® Core™ i7-7700HQ × 8

Software Information:

- **Firmware Version:** -
- **OS Name:** Kali Linux 2024.1
- **OS Type:** 64-bit
- **GNOME Version:** 46
- **Windowing System:** Wayland / Xorg
- **Kernel Version:** Linux 6.6

Introduction

Efficient dynamic memory allocation is a critical aspect of modern operating systems, directly impacting performance and resource management. In this assignment, we focus on the implementation of a custom dynamic memory allocator based on segregated memory pools. This lab explores several memory allocation strategies while allowing for the integration of safety mechanisms to detect and mitigate common memory management errors.

The allocator we designed manages memory through four distinct pools, each catering to different request sizes: Pool 0 (for requests ≤ 64 bytes), Pool 1 (≤ 256 bytes), Pool 2 (≤ 1024 bytes), and Pool 3 (for larger requests). The first three pools, referred to as *fast pools*, utilize fixed block sizes and are optimized for small, frequent allocations, while Pool 3, the *standard pool*, employs a more flexible memory allocation approach, using variable-sized blocks managed via a linked list. For Pool 3, we implement three placement policies: *First Fit, Next Fit* and *Best Fit*, with the goal of minimizing wasted memory and improving performance.

To ensure efficient memory management, we also implement coalescing of adjacent free blocks, address alignment constraints, and safety checks against common memory bugs like memory leaks, double frees, and use-after-free errors. This project provides an opportunity to explore how different allocation strategies impact fragmentation and allocation speed, and how memory safety can be enhanced in real-world applications.

In the following sections, we present the design choices made, the implementation details of each memory pool, and the various safety checks integrated into the system. Additionally, we discuss the challenges encountered and the results obtained from testing the allocator across different scenarios.

Overview of Functionality

This memory allocator is designed using a segregated memory pool approach, managing memory requests through four distinct pools.

Fast Pools (Pools 0, 1, 2)

The fast pools handle small, fixed-size memory requests (up to 64 bytes, 256 bytes, and 1024 bytes). These pools use a Last-In-First-Out (LIFO) allocation policy, allowing for fast memory operations with minimal overhead. While internal fragmentation may occur due to fixed block sizes, this design ensures quick allocations. Freed blocks are simply added back to the pool's free list, and no block splitting or coalescing is necessary.

Standard Pool (Pool 3)

The standard pool manages larger, variable-sized requests using a doubly linked list. It implements more complex memory management strategies such as block splitting (for large blocks) and coalescing (for adjacent free blocks) to reduce fragmentation.

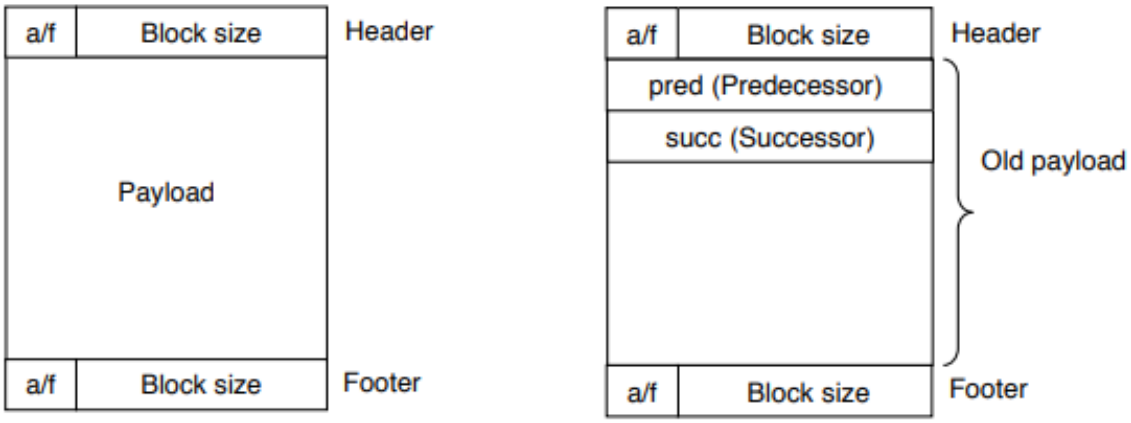


Fig : " Detailed description of a standard pool block "

Memory Display

The `print_mem_state()` function visually represents the memory layout, showing allocated and free blocks across both the fast and standard pools, aiding in debugging and understanding memory fragmentation.

Implemented features

In mem_alloc_fast_pool.c

- **void init_fast_pool(mem_pool_t *p, size_t size, size_t min_request_size, size_t max_request_size)**: Initializes the fast pool, setting up the memory region, free block list, and block size limits.
- **void *mem_alloc_fast_pool(mem_pool_t *pool, size_t size)**: Allocates memory from the fast pool, rounding up the request size to the appropriate block size.
- **void mem_free_fast_pool(mem_pool_t *pool, void *b)**: Frees a block and returns it to the fast pool's free list.
- **size_t mem_get_allocated_block_size_fast_pool(mem_pool_t *pool, void *addr)**: Returns the size of an allocated block in the fast pool.
-

In mem_alloc.c

- **void print_mem_state(void)**: Displays the current memory allocation state for debugging and visualization purposes.
-

In mem_alloc_standard_pool.c

- **void init_standard_pool(mem_pool_t *p, size_t size, size_t min_request_size, size_t max_request_size)**: Initializes the standard pool for variable-sized block management.
- **void *mem_alloc_standard_pool(mem_pool_t *pool, size_t requested_size)**: Allocates memory from the standard pool, using a specified fit policy.
- **void mem_free_standard_pool(mem_pool_t *pool, void *addr)**: Frees a block in the standard pool and coalesces adjacent free blocks.
- **size_t mem_get_allocated_block_size_standard_pool(mem_pool_t *pool, void *addr)**: Returns the size of an allocated block in the standard pool.

Main design

The design focuses on balancing performance, memory management efficiency, and simplicity. The key decisions made throughout the implementation reflect this approach.

1. **Separation of Fast and Standard Pools**
   Memory management is segregated between fast pools for small, fixed-size blocks and a standard pool for larger, variable-sized blocks. This allows fast pools to handle frequent, small allocations with minimal overhead, avoiding complex operations like splitting or coalescing. The standard pool, however, is designed for flexibility, capable of handling more complex memory needs efficiently.

2. **Fixed Block Size in Fast Pools**
   In the fast pools (mem_alloc_fast_pool.c), blocks are allocated using predefined sizes, ensuring that small requests are handled quickly. The **LIFO policy** speeds up allocation by keeping the most recently freed block ready for reuse.

3. **Variable-Sized Block Management in the Standard Pool**
   The standard pool (mem_alloc_standard_pool.c) uses a **doubly linked list** to manage free blocks and supports both **block splitting** and **coalescing**. Splitting optimizes memory usage by dividing larger blocks when a smaller request is made, and coalescing combines adjacent free blocks to reduce fragmentation, maintaining larger contiguous memory regions.

4. **Fit Policies for the Standard Pool**
   We implemented three fit policies to balance speed and fragmentation:
   - **First Fit**: search for the first block large enough to accommodate the request. It's quick, but may cause external fragmentation at the beginning of the list.
   - **Best Fit**: Reduces internal fragmentation by choosing the smallest suitable block, but can create external fragmentation with lts of very small blocks.
   - **Next Fit**: Resumes the search from the last allocation and return the first suitable block, avoiding fragmentation at the start of the list.

5. **Metadata Management**
   For the fast pools, the metadata is minimal, as we manage free blocks with a one way linked list. In the standard pool, **headers** and **footers** store metadata like block size and status, which allows for efficient management of splitting and coalescing operations.

6. **Memory Display for Debugging**
   The print_mem_state() function was crucial for debugging, as it visualizes the current memory allocation state. This helped us identify fragmentation issues and track allocation patterns during development.

Code Explanation

In `mem_alloc_fast_pool.c`

1. **`void init_fast_pool(mem_pool_t *p, size_t size, size_t min_request_size, size_t max_request_size)`**
    - This function initializes a fast pool by allocating a contiguous memory region using the `my_mmap` function. The pool is responsible for handling requests within a specific size range (`min_request_size` to `max_request_size`).
    - The free block list is set up, where each free block points to the next free block, forming a singly linked list. This allows the pool to quickly serve requests without needing to split or coalesce blocks.

2. **`void *mem_alloc_fast_pool(mem_pool_t *pool, size_t size)`**
    - This function handles memory allocation in the fast pool. First, it checks if the requested size fits within the pool's range (from `min_request_size` to `max_request_size`).
    - If a suitable block is available in the free list, the function removes the block from the list and returns its address to the user. The LIFO policy is used to prioritize the most recently freed block, reducing fragmentation and keeping the allocation process fast.
    - The block size is rounded up to the next multiple of the pool's predefined block sizes. If no block is available or the request exceeds the maximum size for the pool, the function returns `NULL`.

3. **`void mem_free_fast_pool(mem_pool_t *pool, void *b)`**
    - This function returns a previously allocated block to the fast pool's free list. The freed block is inserted at the head of the free list, following the LIFO policy to ensure fast reallocation of recently freed blocks.
    - Since all blocks in a fast pool are of fixed size, the function doesn't need to check or merge adjacent blocks. Instead, it simply updates the free list pointers to add the block back to the pool.

4. **`size_t mem_get_allocated_block_size_fast_pool(mem_pool_t *pool, void *addr)`**
    - This function retrieves the size of a block that was allocated from the fast pool. Since fast pool blocks are all of fixed sizes, this function simply returns the pool's block size associated with the allocation.
    - **N-B:** We hesitated between giving the actual size of the block, or the live data size (Allocation request) without including the internal fragmentation.

In `mem_alloc.c`

1. **`void print_mem_state(void)`**
    - This function provides a visual representation of the current state of the memory allocator, showing whether each block in the memory region is free or allocated. The function scans through all memory pools and prints out the status of each block in a human-readable format, typically using symbols like `.` for free bytes and # for allocated blocks. For the fast pool each memory block is represented with either # or . , and for the standard pool, #size# and .size.
    - The function loops through both the fast pools and the standard pool, displaying the state of each block in the memory region associated with each pool. It helps users visualize fragmentation, allocation patterns, and usage efficiency.

In `mem_alloc_standard_pool.c`

1. **`void init_standard_pool(mem_pool_t *p, size_t size, size_t min_request_size, size_t max_request_size)`**
    - This function initializes the standard pool, which handles large, variable-sized blocks. Unlike fast pools, the standard pool uses a more complex structure with a doubly linked list to track free blocks.
    - Each block in the pool has a header and footer that stores metadata (such as block size and free/allocated status). The function sets up the pool's free list and ensures that the memory region is properly initialized.

2. **`void *mem_alloc_standard_pool(mem_pool_t *pool, size_t requested_size)`**

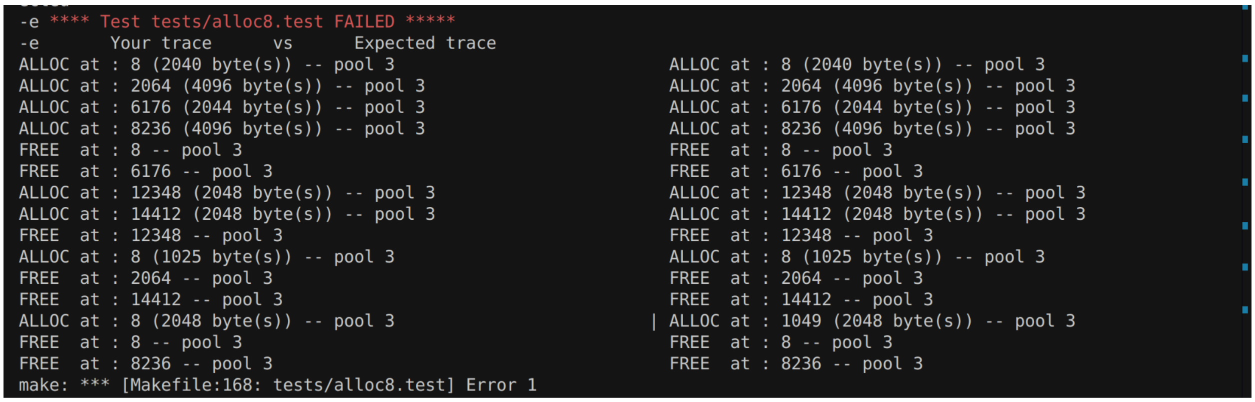
- This function allocates a block from the standard pool. The allocator first searches the free list for a block that can satisfy the requested size. It uses one of the fit policies (First Fit, Best Fit, or Next Fit) to find an appropriate block:
  - **First Fit**: Scans the list and selects the first block large enough to handle the request.
  - **Best Fit**: Finds the block that minimizes leftover space after allocation.
  - **Next Fit**: Continues scanning from the point of the last allocation.
- If the selected block is larger than the requested size, the function splits the block and returns the remaining portion to the free list. If no block can fulfill the request, `NULL` is returned, indicating that memory could not be allocated.

3. **`void mem_free_standard_pool(mem_pool_t *pool, void *addr)`**
   - This function handles the deallocation of a block in the standard pool. It marks the block as free and inserts it back into the pool's free list in address order. The function also checks adjacent blocks to determine if they can be coalesced into a larger block to reduce fragmentation.
   - The coalescing process looks at both the previous and next blocks, merging them if they are free, and updating the free list pointers to reflect the new larger block.

4. **`size_t mem_get_allocated_block_size_standard_pool(mem_pool_t *pool, void *addr)`**
   - This function retrieves the size of a block that was allocated from the standard pool. Since the standard pool handles variable-sized blocks, the function reads the block's header to determine its size.
   - The size is stored as part of the block's metadata (in both the header and footer), allowing the allocator to quickly retrieve it. This information is essential for tracking and managing memory usage in the standard pool.

Execution and testing

Our implementation of the memory allocator passes simple allocation/freeing tests on each pool, and every set of tests provided in the original code, including test 9, and that on every placement policy. There is an exception related to test 8, where the last allocation is done at the 1049th byte instead of the 8th, we did not quite figure out why this is happening.



```
-e **** Test tests/alloc8.test FAILED *****
-e       Your trace      vs      Expected trace
ALLOC at : 8 (2040 byte(s)) -- pool 3                           ALLOC at : 8 (2040 byte(s)) -- pool 3
ALLOC at : 2064 (4096 byte(s)) -- pool 3                        ALLOC at : 2064 (4096 byte(s)) -- pool 3
ALLOC at : 6176 (2044 byte(s)) -- pool 3                        ALLOC at : 6176 (2044 byte(s)) -- pool 3
ALLOC at : 8236 (4096 byte(s)) -- pool 3                        ALLOC at : 8236 (4096 byte(s)) -- pool 3
FREE  at : 8 -- pool 3                                          FREE  at : 8 -- pool 3
FREE  at : 6176 -- pool 3                                       FREE  at : 6176 -- pool 3
ALLOC at : 12348 (2048 byte(s)) -- pool 3                       ALLOC at : 12348 (2048 byte(s)) -- pool 3
ALLOC at : 14412 (2048 byte(s)) -- pool 3                       ALLOC at : 14412 (2048 byte(s)) -- pool 3
FREE  at : 12348 -- pool 3                                      FREE  at : 12348 -- pool 3
ALLOC at : 8 (1025 byte(s)) -- pool 3                           ALLOC at : 8 (1025 byte(s)) -- pool 3
FREE  at : 2064 -- pool 3                                       FREE  at : 2064 -- pool 3
FREE  at : 14412 -- pool 3                                      FREE  at : 14412 -- pool 3
ALLOC at : 8 (2048 byte(s)) -- pool 3                         | ALLOC at : 1049 (2048 byte(s)) -- pool 3
FREE  at : 8 -- pool 3                                          FREE  at : 8 -- pool 3
FREE  at : 8236 -- pool 3                                       FREE  at : 8236 -- pool 3
make: *** [Makefile:168: tests/alloc8.test] Error 1
```

Figure : " Test 8 fail "

Figure : " State of the memory before and after "f2" in test 8 (generated with print_mem_state) "

The error could have originated from the **`mem_free_standard_pool`** function and the placement of free blocks in the free list, but we could not locate it.

Questions raised in the assignment

Step 1 :

Why is a LIFO policy interesting for fast pools?

The LIFO policy is ideal for fast pools because it simplifies both allocation and deallocation. With LIFO, the most recently freed block is always the next to be allocated, reducing fragmentation and ensuring that memory is reused efficiently. This policy is well-suited to fast pools, where the focus is on minimizing the overhead associated with managing free memory. Additionally, this approach requires minimal metadata and ensures that the allocator operates quickly, making it optimal for small block sizes.

Step 2 :

The algorithm used for inserting a freed block into the free list and applying coalescing follows these principles:

1. **Inserting a Freed Block into the Free List**:
    - When a block is freed, it is inserted back into the free list in address order. This ensures that blocks in the list are ordered by their memory addresses, which simplifies the process of detecting adjacent free blocks for coalescing.
    - The free list is a **doubly linked list**, where each free block has a pointer to the previous and next free blocks. When inserting a new block, the allocator traverses the list to find the correct position, ensuring that the memory addresses remain in ascending order.
    - After finding the appropriate position, the freed block's next and prev pointers are updated to point to its neighbors in the list, and the neighbors' pointers are updated accordingly to include the newly freed block.

2. **Coalescing Adjacent Free Blocks**:
    - Coalescing is the process of merging adjacent free blocks into a single larger block to reduce fragmentation and improve memory reuse. After inserting a freed block, the algorithm checks whether the block can be merged with the blocks immediately before or after it.

- **Header and Footer Metadata**: Each block in the standard pool has a header and footer that contain metadata, including the block's size and whether it is free or allocated. The algorithm uses this metadata to determine whether adjacent blocks are free and can be coalesced.
- If the block immediately before or after the freed block is also free, the algorithm merges the two blocks by:
  - Combining their sizes into a single, larger block.
  - Updating the header of the new larger block and the footer to reflect the new size.
  - Removing the adjacent blocks from the free list and updating the prev and next pointers as necessary.
- The algorithm continues this process for both the previous and next blocks, ensuring that all adjacent free blocks are coalesced into the largest possible contiguous memory block.

3. **Handling Edge Cases**:
   - If the freed block is at the beginning or the end of the memory region, the algorithm ensures that no invalid memory accesses occur by carefully checking the boundaries.
   - Blocks that cannot be coalesced are left as-is in the free list, ensuring that future allocations can make use of them.

Conclusion and feedback

This lab was both challenging and rewarding, as it pushed us to develop a deeper understanding of pointers and introduced us to new data structures, such as simple and doubly linked lists. Integrating these structures with memory management techniques, including FIFO and LIFO strategies, required careful thought and design.

We successfully implemented all the main features described in the assignment, including the fast memory pools and the standard pool with support for different allocation strategies (First Fit, Best Fit, and Next Fit). Our allocator passes most of the provided tests, though a few minor bugs remain, particularly in edge cases related to block coalescing and memory alignment.

However, one of the most significant challenges we faced was debugging. As neither of us were familiar with GDB initially, troubleshooting issues in the code was difficult.

Overall, this lab provided a valuable learning experience, offering insight into the complexities of memory management and dynamic allocation, as well as the importance of efficient debugging techniques.