Free RAM Retrieval

After the hardware has been initialised by the bootloader (GRUB), control is returned to the kernel at the address pointed to via the **start** label.
On return to the **start** label, the ebx register is loaded with an address located in the first MB of memory.
This address points to the multiboot structure which contains information about the hardware.

Most notably, the Multiboot 1 structure returns a field with an address to a table containing **multiboot_mmap_entry** structures.
These structures represent the current memory blocks after initialisation.
The address may be accessed via the **mmap_addr** field and the table is of size **mmap_length**.

Using the **cos_init_memory** function located in **src/kernel/memory/mmap_init.c** in the main COS repository, structures are enumerated from the table. The type field in the **multiboot_mmap_entry** structure allows the kernel to retrieve the memory blocks marked as free.

Basic Memory Manager Initialisation

The **initialise_memory_manager_addressing** function of the MemoryManagement module is called by the COS kernel during the final stages of the **cos_init_memory** function.
Calling this function will initialise two linked lists with the address passed as a parameter.

The kernel passes the **kernel_end_address** variable as its parameter and it represents the first address in memory after kernel data and code.
This address is retrieved from the **linker.ld** file located at the root of the main COS repository and it points to a memory area inside a free memory block.

These 64kb memory blocks are defined via the **page_frame_entry_t** structure and the linked lists that were initialised points to **page_frame_entry_t** entries.
The **available_process_frames** list is the one loaded with page frames. Although it is known that huge pages can be used (2MB and 1GB pages), the implementation currently chosen as the MemoryManager exclusively uses 4kb frames.

As paging is enabled during hardware initialisation, the pointer to this table must be passed to the cr3 register.

Free Block Claiming From Kernel Page Table

Once the kernel has initialised its internal memory tracking structures, the **claim_free_kernel_blocks** function is called to extract any remaining usable memory blocks from the kernel's page tables.
This function is located inside the **memory/mmap_init.c** file and is invoked near the end of **cos_init_memory**.

At this point, paging is already active and the kernel page tables are set up.
The kernel's level-4 page table (PML4) is stored in the **kernel_page_table** field of the **cos_memory_properties_t** structure passed to this function.
To access all mapped regions of physical memory, the function begins its traversal at entry 511 of the PML4.
This corresponds to the canonical high half of the virtual address space (starting at 0xffff800000000000), where the kernel resides.

Page Table Traversal

The function walks through all levels of the page table hierarchy:

```
PML4 Entry (Level 4) → kernel_page_table->entries[511]
PDPT (Level 3) → kernel_page_table_level_3
PD (Level 2) → kernel_page_table_level_2
PT (Level 1) → kernel_page_table_level_1
```

At each level, it verifies that the current entry is valid (present and either read_write or pmd_address as required).
If any check fails, that branch of the address space is skipped.

Filtering Memory Blocks

For every valid Level 1 Page Table Entry (PTE), the function checks whether the corresponding physical block should be reclaimed:

- If the block is part of the kernel image (is_kernel_block)
- If the block is part of the reserved low memory range (is_reserved_low_memory)
- If the block lies in the ISA memory hole (is_isa_memory_hole)
- If a block satisfies none of these conditions, it is considered usable and is processed accordingly.

Allocating Page Frame Structures

The function attempts to populate **page_frame_entry_t** structures to represent each reclaimed 4KB memory block.
These are stored in memory allocated at runtime from free memory blocks — tracked by the **next_kern_addr_page_frames_list** field of the memory properties.

There are two cases:

- Start of a New Block:
  - If no structures exist yet (last_page_frame_list_structure_ptr == NULL) or the end of the block is reached (checked via is_end_of_block), a new 4KB memory block is reserved and zeroed.
  - The first page_frame_entry_t structure inside this block is added to the allocated frames list, and the bookkeeping pointer (last_page_frame_list_structure_ptr) is updated.
- Within Current Block:
  - If space remains in the current 4KB page-frame-list block, a new page_frame_entry_t is added immediately after the previous one.
    This entry is added to the available frames list, and the list tail pointer is updated.

The physical address of the memory block is stored in the **.address** field of the **page_frame_entry_t**.
Once recorded, the original mapping in the page table is invalidated by setting its PTE to 0.

Out-of-Memory Handling

If, for any reason, the kernel runs out of space for allocating page frame structures, it prints a red warning to the terminal using **cos_term_set_color**, and **out_of_memory_warning** is triggered (though this branch is not used in the current code).

This function finalises the memory manager's initialisation by scavenging for additional available RAM directly from the kernel's paging structures.
By iterating through the kernel's address space and selectively reclaiming unused physical blocks, it ensures all usable memory is accounted for.
Each block is tracked via page_frame_entry_t entries, dynamically linked and used by the rest of the memory management subsystem.

This completes the initialisation phase of the kernel memory manager. From this point forward, page frame allocation and deallocation can proceed via these internal structures.

Page Table Expansion and Reserved Memory Blocks

As the COS kernel dynamically allocates physical memory for its internal page table structures, it must ensure that it always has a buffer of reserved blocks available for emergency or recursive use. This need is handled through the use of reserved memory blocks, tracked by the **kernel_reserved_blocks** and **kernel_reserved_blocks_end** pointers.

These reserved blocks are distinct from the general-purpose available_page_frames list.
While both lists are built from the same backing memory block chain, the reserved list is excluded from the general allocation pool to prevent unintended reuse during critical kernel operations (such as mapping new page tables).

These reserved memory addresses can be instantaneously reused by the kernel as they are always kernel mapped.

Reserved Block Replenishment

The function **check_reserved_blocks_status** is responsible for ensuring that a minimum number of reserved blocks are always available to the kernel.
In the current implementation, this minimum is set to 5 blocks, and the target replenishment size is 10 blocks.
This is a conservative default that allows for:

- Expansion of page tables up to two levels deep
- Handling of recursive mappings
- Safety during kernel memory pressure conditions

The function calls **allocate_memory_blocs** to retrieve the needed blocks and re-links them exclusively into the reserved block list.
If allocation fails, a warning is printed, and the function returns false.

Note: While functionally sufficient, this implementation introduces recursion through allocate_memory_blocs, which may not be ideal. The code includes TODO comments acknowledging this limitation and the need for eventual refactoring.

External Documentation

- Low Memory Extraction
- Understanding x86_64 paging