Kernel Virtual Memory Mapping

When the kernel needs to create a new mapping between a physical address and a virtual address in the higher-half memory (starting at 0xffff800000000000), it calls the kernel_map function. This function dynamically expands the kernel's page table if necessary and ensures that the physical address is correctly mapped.

Mapping Logic

The kernel always uses entry 511 in the PML4 table to store its mappings in the higher-half address space.
This corresponds to the top 512GB of virtual memory.

The function traverses the kernel page table hierarchy:

- Level 4 (PML4): Always starts at entry 511.
- Level 3 (PDPT): Iterates through all 512 entries.
  If an entry is empty, a reserved block is claimed and linked into the page table.
- Level 2 (PD): Same treatment as above.
- Level 1 (PT): If an entry is available (not marked as present), the desired physical address is mapped with read/write and present flags.

Each time a new level of the page table hierarchy is created (i.e., the next level does not exist), the kernel uses one of its reserved blocks and zeroes it to serve as the new page table.

Virtual Address Computation

Once a free slot is found at the lowest level (PT), the virtual address is assembled using the current indices:

```
virt_addr =
    0xffffUL << 48 |
    (level_4_index & 0b111111111) << 39 |
    (level_3_index & 0b111111111) << 30 |
    (level_2_index & 0b111111111) << 21 |
    (level_1_index & 0b111111111) << 12;
```

This ensures that the returned virtual address always falls within the higher-half space and that no lower-half mappings are ever created.

Reserved Block Recycling

Every time a reserved block is used to expand a page table level, it is also linked into the allocated_page_frames list. This ensures that it is tracked properly and not inadvertently lost from internal management.

Failure and Limits

If no virtual address slot is available across the entire 512GB kernel virtual space, the function prints a warning and returns -1.
This case should be unreachable under realistic conditions unless: - The entire higher-half space is exhausted. - Unmapping is never performed.

Implementation Notes:

TODOs in the code correctly identify future improvements:

- Recursion in allocate_memory_blocs should be removed or replaced with a more predictable allocation scheme.
- Reserved blocks are currently identity-mapped, which may not align with future kernel address space restrictions.
- kernel_map must be audited for use outside initialisation, as it assumes all parent table levels remain present.

This function is critical for enabling dynamic mapping during runtime and supports the kernel’s ability to manipulate memory structures post-boot safely.

Page Frame Claiming and Allocation

The COS kernel handles physical memory in fixed-size 4KB blocks, known as page frames.
All memory allocations, reservations, and mappings are done on these page-sized units.

Two key functions manage this page-based allocation logic: - claim_free_page_frames(…) – responsible for finding and preparing free memory blocks. - allocate_memory_blocs(…) – consumes those free blocks and moves them into the allocated list.

claim_free_page_frames

This function is the backbone of dynamic memory discovery in COS. It searches through memory regions defined at initialization (entries_addr[] and entries_len[]) and builds a linked list of available page frame structures.

The function allocates page frame structures (page_frame_entry_t) inside the very blocks they describe.
To avoid circular dependencies (e.g., trying to allocate a structure on itself), the function performs mapping and tracking in a very specific order.

Before any pointer is dereferenced or memory is written, the physical address is passed to kernel_map to ensure it is virtually mapped in the kernel address space. Memory regions are aligned and sized in multiples of 0x1000 (4KB), ensuring compatibility with hardware page tables.

Control Flow Highlights

The function loops through available memory regions, tracking:

- Which region (entry_addr_index) is currently being carved.
- The current offset (offset) into the region.

Whenever it needs to insert a new metadata structure (page_frame_entry_t) in a new page frame used by the page frames list, it: - Calls kernel_map to map the physical address into the kernel’s virtual space. - Initializes and links the new structure to the allocated_page_frames list. - Adds subsequent entries to the available_page_frames list. - Calls check_reserved_blocks_status(…) every time a new structure page is created to ensure reserved blocks are not exhausted, as these are needed for further virtual memory expansion.

If at any point mapping fails or no more free pages are found, the function prints a warning and aborts.
It also updates global counters like number_of_allocated_page_frames and number_of_available_page_frames for accurate memory accounting.

allocate_memory_blocks

This function acts as a consumer of the available_page_frames list and moves entries into the allocated_page_frames list based on demand.

Function Purpose Allocate number_of_blocks_requested page frames from the available list. Return a pointer to the head of a contiguous sublist representing those blocks.

Control Flow Highlights

- If insufficient free blocks exist, calls claim_free_page_frames(…) to replenish.
- Walks forward in the available_page_frames list to isolate a sublist of the requested size.
- Moves this sublist from available_page_frames into allocated_page_frames.
- Ensures all list pointers (next_frame, prev_frame) are updated correctly.
- Returns a pointer to the start of the newly allocated block chain.

If allocation fails at any point (e.g., out of memory), returns NULL.

Notes:
This mechanism is flawed.
Once a proper virtual address manager will be made (alongside a scheduler to introduce the concept of processes), allocated blocks will be remembered by any process, including the kernel which will be considred a process itself.
Thus, reserved blocks will no longer need to be extracted from this list.

Kernel Virtual Memory Allocation and Page Table Construction

To provide isolated user-space virtual memory in COS, a memory mapping routine is required to connect physical pages of memory to addresses in the virtual address space of each process.
In x86_64 architecture, user-space memory is conventionally located in the lower half of the 64-bit address range, while kernel memory is placed in the higher half.
This is done via the function **cos_mmap**, which attempts to find the next available lower-half virtual address in a page table and link it with a given physical memory address.

Since paging structures are sparse and need to be dynamically allocated at runtime, this function ensures that all intermediate levels of the page table (PML4 → PDPT → PD → PT) are created as needed.

If a valid mapping is made, the function returns the corresponding virtual address.
If all potential virtual addresses in the lower-half are exhausted or a mapping cannot be established, the function returns (uint64_t)-1 as a sentinel failure value.

This function is a key component of per-process memory management and serves as the foundation for later routines which allocate and map executable binaries or heap memory.

It is important to note that:

- Only 4KB pages are supported (no huge pages).
- The function makes use of recursive mapping and identity-mapped kernel memory.
- Kernel space addresses are excluded from the mapping range.

What's more is that, compared to a proper **mmap** reimplementation, cos_mmap currently possesses the following parameters.

- properties: A pointer to a cos_memory_properties_t structure, representing the kernel's memory allocator and management context.
- table: The root PML4 table pointer for the process to which the physical page is to be mapped.
- phys_addr: The physical memory address of the 4KB block to be mapped.

User Process Page Table Constructor

To support process isolation and virtual memory separation in COS, each process is assigned a unique page table constructed dynamically by the kernel. The **kernel_make_process_page_table** function is responsible for allocating and initializing this page table and mapping a given number of physical memory blocks into the virtual address space of the process.

It performs the following steps: - Allocates a new PML4 (Page Map Level 4) table. - Copies the kernel-space mapping (index 511) from the master kernel PML4 to the new table, allowing kernel code to remain accessible from all processes.

WARNING: Currently, every process is ran in Ring 0. - Adds a recursive page table mapping at index 510 to enable runtime access to the process's own page tables via a known address. - For the specified number of blocks (bin_size), calls cos_mmap to allocate and map the physical pages consecutively into virtual space. - Stores the address of the final mapped page in final_bin_virt_addr to be used as the process entry point or heap start. - Returns a page_table_t* representing the root of the new PML4 page table, or NULL on failure.

Recursive page mapping allows each process to introspect or modify its own page table through a single virtual address range. This design is widely used in systems like Linux, and described in **Section 4.5 of the Intel Manual**, which explains how paging structures can be mapped recursively for debugging or context switching purposes.

CPU Paging and TLB Control Functions

Modern x86_64 CPUs use a Translation Lookaside Buffer (TLB) to cache recent virtual-to-physical address translations and accelerate memory access.
When page tables are modified (e.g., entries added or removed), cached TLB entries must be invalidated to ensure consistency between page tables and actual address translation.

The **flush_tlb** function performs a full TLB flush by reloading the CR3 control register with its current value.
This operation causes the CPU to discard all cached page translations and re-read page tables from memory on subsequent accesses.

This technique is documented in **Intel Volume 3, Section 4.5.3** where reloading CR3 is described as the standard method for invalidating the entire TLB.

Sometimes, it is desirable to flush only a single page's translation from the TLB to reduce overhead.
The **flush_address_from_tlb** function uses the invlpg instruction to invalidate the TLB entry for a specific virtual address.

This is particularly useful after modifying or removing a single page table entry without flushing the entire TLB.
This instruction is recommended by Intel for fine-grained TLB management and is detailed in Intel SDM Volume 3, Section 4.5.3.

Switch Active Page Table

To switch the current address space, the CPU's CR3 register must be updated with the physical address of a new PML4 page table.
The **load_page_table** function safely performs this switch by combining the given page table pointer with existing CR3 flags, preserving the non-address bits such as cache disabling or protection bits.

This approach avoids inadvertently clearing or corrupting CR3 flags that control caching or paging behavior, as outlined in **Intel SDM, Volume 3, Section 4.5.2**.

Virtual to Physical Address Translation

The **virtual_to_physical_address** function manually walks the four-level x86_64 page table hierarchy to resolve a virtual address to its corresponding physical address.

The virtual address is decomposed into its paging indices according to the 4-level paging scheme:

- PML4 index: bits 47–39
- PDPT (Page Directory Pointer Table) index: bits 38–30
- PD (Page Directory) index: bits 29–21
- PT (Page Table) index: bits 20–12

At each level, the relevant page table entry is checked for validity by masking out flags and checking if the base address is zero.

If any entry in the hierarchy is invalid or absent (zero base address), the function returns (uint64_t)-1 indicating that the virtual address is unmapped.
Otherwise, the physical page base address from the lowest-level page table entry is combined with the page offset (bits 11–0) to produce the final physical address.

WARNING: It does not handle large pages (2MB or 1GB) or page faults; only 4KB page mappings.

Recursive Page Table Trick Explained

Recursive page table mapping is a clever technique that maps the page table hierarchy into itself:

By setting one PML4 entry (here at index 510) to point back to the PML4 table itself, the page tables become self-referential.
This allows the OS kernel to access and modify page tables as if they were normal memory pages.
As such, there is mo need to store or switch to physical addresses when updating page tables — the kernel can access all levels of paging structures through a well-known virtual address range.

This assembly snippet and accompanying macros implement a recursive page table mapping for the x86_64 architecture.

Default Kernel Page Table Setup

```
page_table_level_4:
    dq page_table_level_3 - VIRT_ADDR + 0b11  ; Recursive entry
pointing to level 3 table, with Present and Writable bits set.
    %rep 509
        dq 0x0                                ; Zero out remaining
entries in the PML4 table.
    %endrep
    dq page_table_level_4 - VIRT_ADDR + 0b11  ; Entry for
recursive mapping of PML4 itself.
    dq page_table_level_3 - VIRT_ADDR + 0b11  ; Entry for higher
half addresses.
```

The level 4 page table (PML4) has 512 entries, each corresponding to a 512 GiB chunk of the virtual address space.

- Entry Values: Each entry is a 64-bit descriptor (defined with dq). The value assigned is a pointer to a page table at the next lower level, with flags appended.
- Flags (0b11): These are bitwise flags to mark the entry as present (bit 0) and writable (bit 1).

Entries 510 and 511 are specially set:

- Entry 510: points back to PML4 itself for recursive mapping.
- Entry 511: points to the higher half kernel addresses.
- Zeroed Entries: The remaining 509 entries are set to 0, indicating unmapped regions.

Macros for Calculating Page Table Entries and Virtual Addresses

```
#define RECURSIVE_BIT_EXPANSION (0xFFFFUL << 48)
#define RECURSIVE_PAGE_TABLE_ENTRY 510UL
```

```
#define PML4_ENTRY(pml_index)
         (RECURSIVE_BIT_EXPANSION | (RECURSIVE_PAGE_TABLE_ENTRY <<
         39) | \
(RECURSIVE_PAGE_TABLE_ENTRY << 30) | (RECURSIVE_PAGE_TABLE_ENTRY
         << 21) | ((uint64_t)(pml_index) << 12))

#define PUD_ENTRY(pml_index, pud_index)
         (RECURSIVE_BIT_EXPANSION | (RECURSIVE_PAGE_TABLE_ENTRY <<
         39) | \

         (RECURSIVE_PAGE_TABLE_ENTRY << 30) | ((uint64_t)
         (pml_index) << 21) | ((uint64_t)(pud_index) << 12))

#define PMD_ENTRY(pml_index, pud_index, pmd_index)
         (RECURSIVE_BIT_EXPANSION | (RECURSIVE_PAGE_TABLE_ENTRY <<
         39) | \

         ((uint64_t)(pml_index) << 30) | ((uint64_t)(pud_index) <<
         21) | (uint64_t)(pmd_index) << 12)

#define PT_ENTRY(pml_index, pud_index, pmd_index, pt_index)
         ((RECURSIVE_BIT_EXPANSION | ((uint64_t)(pml_index) << 39)
         | \

         ((uint64_t)(pud_index) << 30) | ((uint64_t)(pmd_index) <<
         21) | ((uint64_t)(pt_index) << 12)))
```

The RECURSIVE_BIT_EXPANSION expands the upper 16 bits to 0xFFFF (sign extension for canonical 64-bit addresses on x86_64).
This ensures addresses are valid canonical user or kernel addresses.
These macros construct virtual addresses that directly index into each level's table entry via recursive mapping.
The RECURSIVE_PAGE_TABLE_ENTRY (510) is inserted into higher level indices where appropriate to maintain the self-reference chain. The lowest 12 bits (<< 12) correspond to the page offset.