CS 370 Operating Systems
Assignment 2: Memory Management

Fall 2025

Lead TAs: Eman Nabeel, Shirin Rehman

Lahore University of Management Sciences
School of Science & Engineering

October 6, 2025

Contents

1 Introduction

Having successfully completed PA1, where you set up the basic kernel initialization and implemented the fundamental system call abstraction (`read`, `write`), you now have the essential groundwork in place to build more complex OS functionality.

PA2 marks the next major milestone in your kernel's development: **memory management**. In this assignment, you'll build the core infrastructure for managing both physical and virtual memory, which is crucial for process isolation, dynamic resource allocation, and multiprogramming.

2 Objectives

The goal of PA2 is to design and implement a modular memory management subsystem within your kernel.

By the end of this assignment, you should be able to:

- Understand how operating systems manage physical and virtual memory at the hardware level
- Implement frame allocation using bitmap-based tracking
- Design and build a two-level paging system for address translation
- Create and manage virtual address spaces using page directories and page tables
- Integrate physical and virtual memory managers with a kernel heap

This includes the following three key components: (Figure 1 shows how the three components interact with each other)

2.1 Physical Memory Manager (KMM)

Responsible for tracking frames of physical memory. You'll implement a frame allocator (e.g., using a bitmap), and provide interfaces to allocate and deallocate memory needed by kernel components.

2.2 Virtual Memory Manager (VMM)

Built atop the KMM, the VMM manages virtual-to-physical address translation. You will configure paging, manage page tables and directories, and lay the foundation for userland memory isolation.

2.3 Kernel Heap Allocator (KHEAP)

To support dynamic memory allocation in kernel space, you'll implement a heap allocator (e.g., using a free list or buddy system). This heap will be used for managing kernel-internal data like process descriptors, I/O buffers, etc.

3 Overview

This assignment is divided into three distinct components, each corresponding to a specific memory management task. Together, they provide a comprehensive foundation for your kernel's memory subsystem.

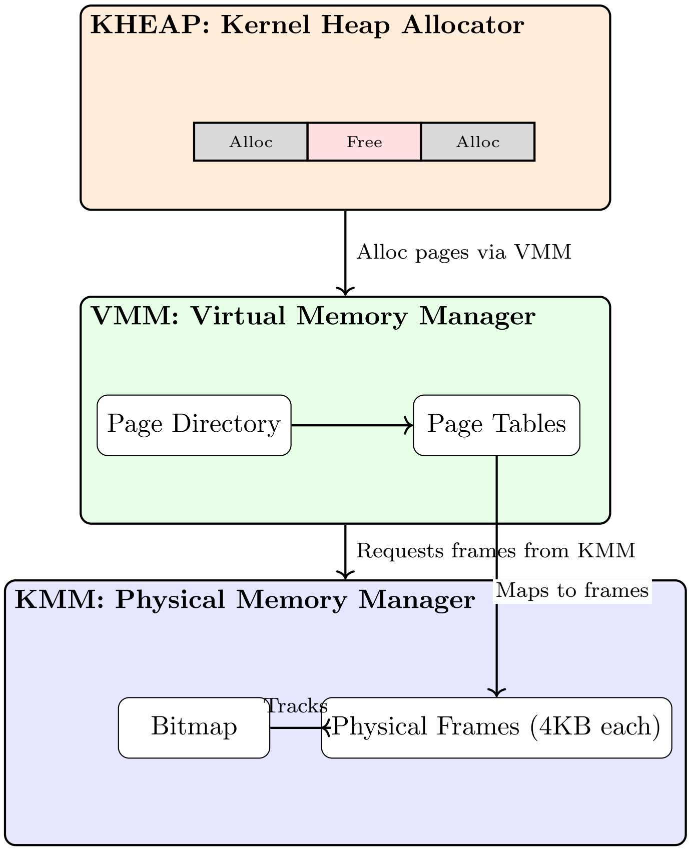

KHEAP: Kernel Heap Allocator
Alloc
Free
Alloc
Alloc pages via VMM
VMM: Virtual Memory Manager
Page Directory
Page Tables
Requests frames from KMM
Maps to frames
KMM: Physical Memory Manager
Bitmap
Tracks
Physical Frames (4KB each)

Figure 1: Interaction between KMM, VMM, and KHEAP in the kernel memory subsystem.

Part	**Component**	**Marks**	**Estimated Effort**
Part 1	Physical Memory Manager (KMM)	30%	1–2 days
Part 2	Virtual Memory Manager (VMM)	40%	3–4 days
Part 3	Kernel Heap Allocator	30%	2–3 days

Table 1: Breakdown of PA2 into components, marks, and estimated time.

Each part builds upon the last. It is recommended that you complete them in the given order, as the Virtual Memory Manager depends on the KMM, and the Kernel Heap depends on both.

3.1 Starting files

You are expected to continue with your PA1 code. Following is the details of the files in `release.zip` downloaded from LMS.

- `boot`: Replace this new `boot` folder with the older one in your PA1 code.
- `include/mm`: Add this new folder in your `include` directory.
- `mm`: Add this folder to the root directory. This directory will contain your implementation of all memory management functions. The makefile required to build all these files is included in this directory.

- `tests`: This directory contains all the tests for PA2, along with PA1. You can choose to replace this with your previous `tests` directory, or you can copy the `mm` directory in to your previous folder.
- `makefile`: This is the new makefile that should work for all the students, includes the compilation of code in `mm`.

Hidden Test cases will carry a weightage of **30%**.

Note

In order to do this assignment, a complete implementation of PA1 is not required. Only the interrupt mechanism is the requirement, however, if you also want terminal output, the terminal and vga part is required. If you have not implemented PA1 at all, **you should contact the course staff immediately. They will provide you with the PA1 implementation binary, and the corresponding instructions.**

4 Part 1: Physical Memory Manager (KMM)

As you may know, managing memory is critically important. All of our data and code share the same physical address space. If we attempt to load and work with more data or programs, we will need to find a way of managing the memory to allow this.

At this stage, our kernel has full control of all of the hardware and memory in your computer. This is great, but bad at the same time. We have no way of knowing what areas of memory are currently in use nor what areas are free. Because of this, there is no real way of working with memory without the possibility of problems: program corruption, data corruption, no way of knowing how memory is mapped, triple faults or other exception errors, etc... The results may be unpredictable.

The Physical Memory Manager part of the kernel (KMM) is responsible for tracking the availability and allocation of physical memory frames in the system. It is initialized using a BIOS-provided memory map (more details in section 4.3) and uses a bitmap-based scheme to track memory usage at the granularity of fixed-size frames (typically 4KB). The bit map approach is very efficient in size. Because each bit represents a block of memory, a single 32-bits word using this bitmap approach represents 32 blocks. Because 32 bits is 4 bytes, this means we can watch 32 blocks of memory using only 4 bytes of memory. This approach is a bit slower as it requires searching the bit map for a free block (the first bit that is 0) every time we want to allocate a block of memory.

Memory regions are marked as either usable or reserved, and the KMM ensures safe and efficient allocation of physical memory to other kernel subsystems.

4.1 Responsibilities

- Parse and interpret the memory map provided by the bootloader. This memory information is present at a known memory address (details in 4.3)
- Implement and initialize a global bitmap to track the allocation status of memory frames.
- Provide functions to allocate and free memory frames.
- Reserve memory used by the kernel and the bitmap itself.

4.2 Deliverables

You are expected to implement the following core KMM functionalities. Each function below includes a brief description of its purpose and key implementation hints. Use the provided data structures, constants, and BIOS memory map to complete the kernel memory manager.

- `void kmm_init(void)`

  This function initializes the Kernel Memory Manager. Refer to section 4.3.1 for a high level overview of the implementation. But, briefly it should:

  - Retrieve memory size and memory layout information that was provided by the BIOS.
  - Determine total physical memory and compute how many frames exist.
  - Initialize the bitmap that tracks which frames are free or used.
  - Mark all frames as used initially, then mark usable regions (type 1) as free.
  - Reserve the regions occupied by the kernel, the bitmap itself, and low memory (up till low 1MB, this memory is considered to be reserved by the kernel for important information).

  Hint: Use the symbols `kernel_start` and `kernel_end` (provided by the linker script) to find the kernel's memory footprint. You will run into the problem of deciding where to put the memory bitmap. One can decide to put the memory bitmap, right after the kernel code and data (`kernel_end`), on a page-aligned address.

- `void kmm_setup_memory_region(uint32_t base, uint32_t size, bool is_reserved)`

  Marks a specific memory region as either usable or reserved. It should:

  - Convert the base address and size into frame indices (based on the frame size constant).
  - Iterate over the range and either set or clear the bits in the bitmap depending on the `is_reserved` flag.

  Hint: Each bit in the bitmap represents one frame of memory. Setting a bit means the frame is used; clearing it means it's free.

- `void* kmm_frame_alloc(void)`

  Allocates one free physical frame and returns its physical address. It should:

  - Search the bitmap for the first free bit.
  - Mark that frame as used and update the usage counter.
  - Return the physical address corresponding to that frame index.

  Hint: Ensure frame 0 is never returned (it often represents a null or invalid address).

- `void kmm_frame_free(void* phys_addr)`

  Frees a previously allocated physical frame. It should:

  - Convert the given physical address into a frame index.
  - Clear the corresponding bit in the bitmap to mark it as free.
  - Update the used frame counter.

Hint: Always validate that the address is within the valid physical memory range before freeing.

- `uint32_t kmm_get_total_frames(void)`

  Returns the total number of memory frames detected during initialization. Use this for statistics and debugging purposes.

- `uint32_t kmm_get_used_frames(void)`

  Returns the number of currently used (allocated or reserved) frames. This can be used to calculate memory utilization and verify the correctness of allocation/freeing.

4.3 System memory information data structures

`e801_memsize_t`

This structure is returned by BIOS interrupt `0x15, EAX=0xE801`, which reports the amount of extended memory available on the system. It provides a coarse summary of memory both below and above the 16 MB mark. This information is used during early boot to initialize the physical memory manager (KMM).

```
typedef struct {
    uint16_t memLow;   // memory (in KB) between 1MB - 16MB
    uint16_t memHigh;  // memory (in 64KB units) above 16MB
} e801_memsize_t;
```

Field descriptions:

- `memLow`: Reports the amount of memory between `1 MB` and `16 MB`, measured in kilobytes (KB). This value helps determine available conventional extended memory.

- `memHigh`: Reports the amount of memory above `16 MB`, measured in 64 KB blocks. It provides a high-level estimate of total extended memory beyond the 16 MB boundary.

Typically, this information is used to calculate the total available physical memory on the system. The bootloader dumps this data structure at the memory location `MEM_SIZE_LOC`, defined in the file `include/mm/kmm.h`. Note, that this data structure assumes that there's already 1MB of memory on the system, and reports on the memory above that region. Make sure to account for this in your memory calculation.

`e820_entry_t`

An array of this structure is returned by BIOS interrupt `0x15, EAX=0xE820` to describe the system's physical memory layout in detail. Each entry corresponds to a contiguous memory region with a base address, length, and type (usable, reserved, ACPI, etc.). The kernel parses these entries to build a map of usable RAM for frame allocation.

```
typedef struct {
    uint32_t baseLow;
    uint32_t baseHigh;
    uint32_t lengthLow;
    uint32_t lengthHigh;
    uint32_t type;
} e820_entry_t;
```

Field descriptions:

- `baseLow`: The lower 32 bits of the region's starting physical address.
- `baseHigh`: The upper 32 bits of the starting address (used on systems with more than 4 GB of RAM). Together, `baseHigh:baseLow` form the full 64-bit base address.
- `lengthLow`: The lower 32 bits of the region's length in bytes.
- `lengthHigh`: The upper 32 bits of the region's length, allowing regions larger than 4 GB.
- `type`: Indicates the kind of memory:
  - `1`: Usable (available RAM).
  - `2`: Reserved (do not use).
  - `3`: ACPI reclaimable.
  - `4`: ACPI NVS (non-volatile sleep memory).
  - `5`: Unusable or defective memory.

  For our purpose, we are only concerned with *usable* and the rest are considered *unusable*.

Usage: The bootloader puts E820 entries at known memory locations: `MEM_MAP_LOC` stores the array, while the `MEM_MAP_ENTRY_COUNT_LOC` stores the number of elements in the array. These constants are defined in `include/mm/kmm.h`. The kernel collects these multiple `e820_entry_t` entries to build a complete physical memory map. Usable regions (type 1) are then passed to the frame allocator (KMM) to manage as free frames, while reserved and ACPI regions are excluded from allocation.

4.3.1 Memory Initialization: In Plain Words

Before the operating system can use memory, it needs to know what parts are available and what parts are already taken. This is where `kmm_init()` comes in.

Here's what it does, step by step:

- It asks the BIOS: *"How much RAM do I have, and where is it?"*
- It divides all of RAM into small, equal-sized boxes (4KB each), called **frames**.
- It creates a **bitmap**, a big list where each bit represents a memory frame:
  - `1` means the frame is **used**
  - `0` means the frame is **free**
- At first, all bits are set to 1, assuming everything is used.
- Then it checks the memory map provided by the BIOS, and for each usable region, it marks those frames as free in the bitmap.
- It also marks memory used by the kernel itself and the bitmap as used, to avoid overwriting its own data.

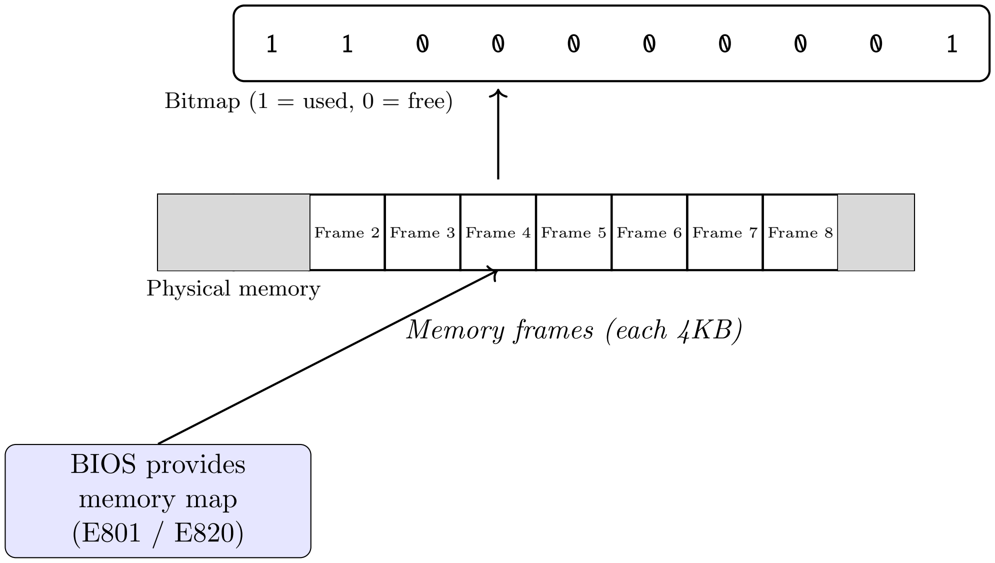


Figure 2: How memory and the bitmap are initialized in `kmm_init()`. Shaded regions are allocated (with a corresponding bit set in the bitmap)

.

- Finally, it prints useful information like total memory, used frames, and free frames to help with debugging.

4.4 Implementation hints

- Start by adding your implementation in `include/mm/kmm.h` and `mm/kmm.c`.
- Read the memory information data structures, their addresses are known.
- Initially, all memory is marked as used. Usable regions are then marked free. This is done because the E820 memory map can contain holes.
- Kernel code and bitmap memory must be explicitly marked as reserved.
- Memory from 0x0 - 1MB must be marked reserved explicitly.
- Skip frame 0 in allocations to avoid accidental NULL pointer usage.
- The bitmap acts as the core data structure of the KMM.
- Every memory operation should consistently update both the bitmap and the used frame counter.
- Logging (via `LOG_DEBUG` or `LOG_ERROR`) is encouraged for tracing and debugging. Look at the file `include/log.h` for usage details, or you can just use your `terminal_write_string` from PA1 for your print debugging.

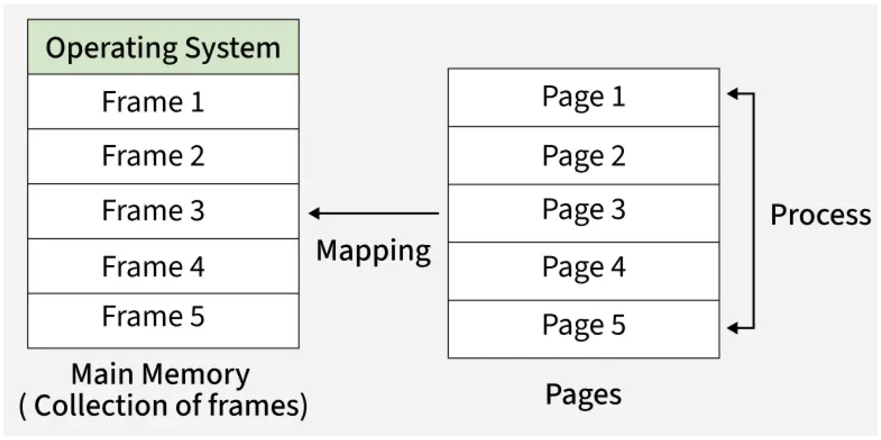


Figure 3: *P*hysical Frames and Pages in Memory Management

5 Part 2: Virtual Memory Manager (VMM)

In this section of the assignment, you will implement the kernel's Virtual Memory Manager (VMM), which provides the crucial abstraction layer between physical memory addresses and the virtual addresses used by the kernel itself and future user processes. By the end of this section, your kernel will have full control over its virtual address space through a **two-level paging system.**

5.1 Virtual Memory

Virtual memory is a fundamental abstraction in modern operating systems that allows each process to operate as if it has its own isolated, contiguous address space, even though the underlying physical memory may be fragmented, shared among processes, or partially swapped to disk. Without virtual memory, systems would suffer from issues like fragmentation, lack of isolation, limited address space, and relocation problems. To overcome these challenges, the Memory Management Unit (MMU), a hardware component built into the CPU, translates *virtual addresses* (used by programs) into *physical addresses* (actual locations in RAM) through kernel-managed page tables, which you will implement and manage in this assignment.

5.1.1 Virtual Memory Layout

Your kernel will use the following virtual address space layout shown in figure 7. It is the division of the virtual address space of the kernel. A description of these regions is given as follows:

Identity Mapping (0x0 - 0x100000)

The identity mapping provides a direct one-to-one correspondence between virtual and physical addresses for the first megabyte of address space. This design decision simplifies several aspects of kernel operation. The VGA text buffer located at physical address `0xB8000` must be accessible for console output, and identity mapping allows the kernel to access it using the same address value.

User Space (0x00100000 - 0xBFFFFFFF)

The region from 1MB to 3GB is reserved for user processes. This massive region (nearly 3GB) will be available for future user programs to use as they see fit. Your VMM implementation will manage

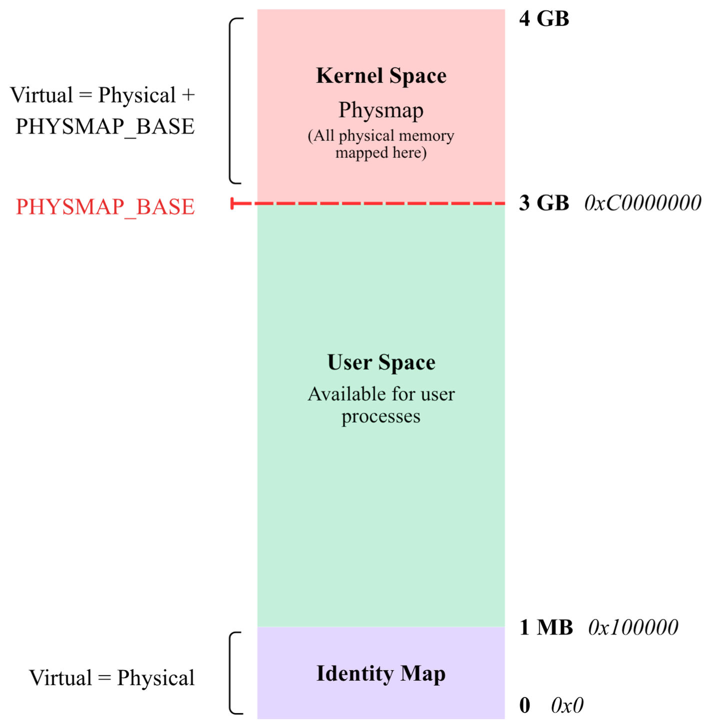


Figure 4: The address space is divided into three regions: a tiny identity-mapped area for hardware access, a large user space for processes, and kernel space containing the **physmap** where all physical memory is accessible

this space by allocating pages on demand and enforcing protection through page table flags.

Physmap (0xC0000000 - 0xFFFFFFFF)

The physmap represents one of the most important kernel design decisions, providing a window through which the kernel can access all physical memory. Every byte of physical RAM is mapped into kernel virtual address space starting at `PHYSMAP_BASE` (0xC0000000, or 3GB). This design choice means the kernel never needs to create temporary mappings to access physical frames, significantly simplifying memory management code.

The conversion between physical and virtual addresses in the physmap follows these formulas:

```
Virtual = Physical + PHYSMAP_BASE
Physical = Virtual - PHYSMAP_BASE
```

The system provides convenient macros for these conversions:

```c
#define PHYS_TO_VIRT(phys) ((void*)((uintptr_t)(phys) + PHYSMAP_BASE))
#define VIRT_TO_PHYS(virt) ((void*)((uintptr_t)(virt) - PHYSMAP_BASE))
```

Note

Page tables and directories must be accessed through the physmap. You cannot directly dereference physical addresses.

5.2 Paging and Virtual Addresses

Paging is the mechanism that makes virtual memory work. It allows each process to have its own isolated address space while sharing the same physical RAM. When a process accesses memory at a virtual address, the Memory Management Unit (MMU) translates the virtual address to a physical RAM address using page tables. This translation happens for every memory access, allowing processes to run as if they own all of the memory while the OS manages the actual physical memory behind the scenes.

In this assignment, we implement a two-level paging architecture, which uses **Page Directories** and **Page Tables** to organize these translations efficiently.

5.2.1 Paging Structures

Paging divides memory into fixed-size blocks and uses a hierarchical structure to translate addresses:

- **Page:** A fixed-size block of virtual memory (4096 bytes = 4KB).
- **Frame:** A fixed-size block of physical memory (also 4KB).
- **Page Table:** A structure containing 1024 entries, each mapping one virtual page to a physical frame.
- **Page Directory:** The top-level structure containing 1024 entries, each pointing to a Page Table.

5.2.2 Address Translation

The x86 architecture implements virtual memory using a two-level page table structure. Every 32-bit virtual address is divided into three components that guide the translation process. The top 10 bits **(bits 31-22)** form the directory index, the next 10 bits **(bits 21-12)**form the table index, and the bottom 12 bits **(bits 11-0)** form the byte offset.

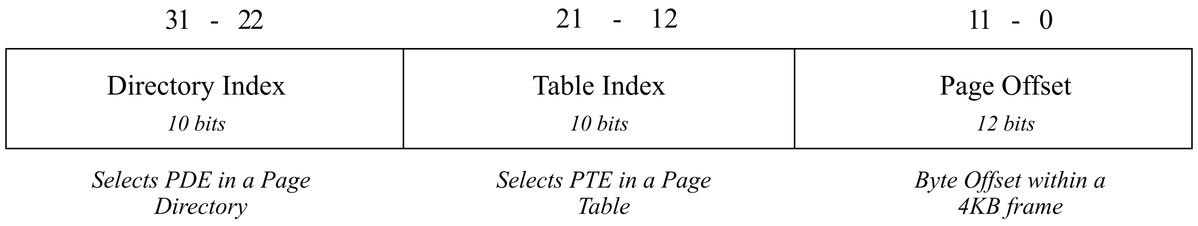

31 - 22
21 - 12
11 - 0
Directory Index
10 bits
Table Index
10 bits
Page Offset
12 bits
Selects PDE in a Page Directory
Selects PTE in a Page Table
Byte Offset within a 4KB frame

Figure 5: 32-bit Virtual Address breakdown

This structure means that a single virtual address like contains all the information needed to locate the corresponding physical byte.

Translation process:

Lets use the following example to see how virtual to physical translation takes place:

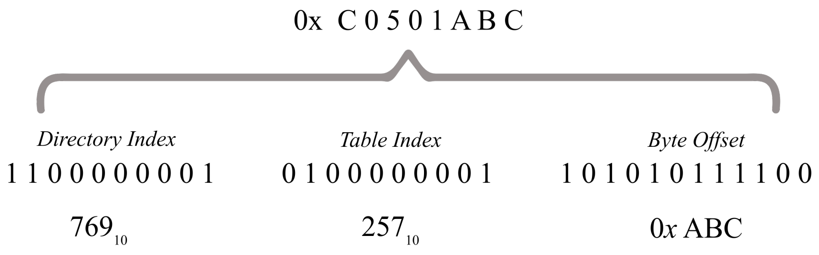


Figure 6: An example 32-bit virtual address broken down

1. **Extract Directory Index:** Extract bits 31 - 22 of the virtual address. This gives the index of one of the 1024 entries in the Page Directory.
2. **Read Page Directory Entry (PDE):** The Page Directory's physical address is stored in the CR3 register. The PDE contains the physical address of a Page Table.
3. **Extract Table Index:** Extract bits 21 - 12 of the virtual address. This gives the index of one of the 1024 entries in the Page Table.
4. **Read Page Table Entry (PTE):** The PTE contains the physical address of the target frame.
5. **Extract Offset:** Extract bits 11 - 0 of the virtual memory to get the byte offset.
6. **Form Physical Address:** The final physical address is the frame base address (from the PTE) plus the offset.

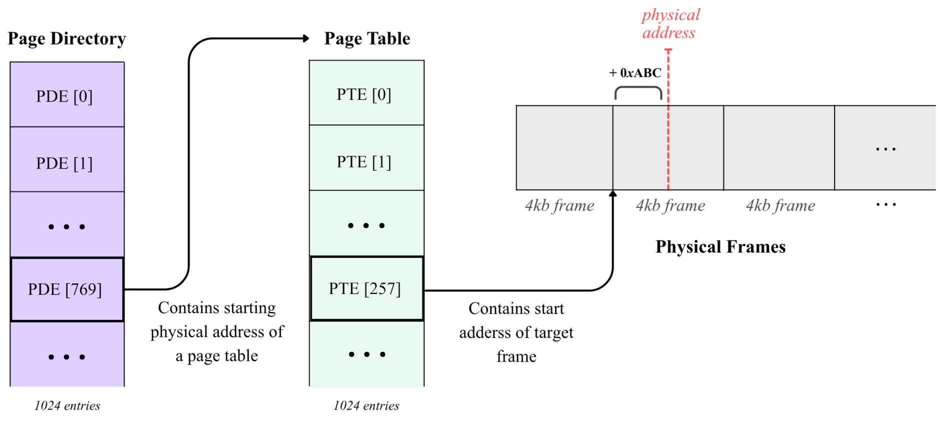


Figure 7: An overview of how virtual addresses are converted to physical addresses

5.2.3 Page Directory Entry (PDE) Format

Each PDE is a 32-bit value that packs both an address and control flags:

```
Bits 31-12: Physical address of Page Table (20 bits, 4KB-aligned)
Bits 11-9:  Available for OS use
Bit  8:     Global (ignored for PDEs)
Bit  7:     Page Size (0 = 4KB pages)
Bit  6:     Available
Bit  5:     Accessed
Bit  4:     Cache Disabled
Bit  3:     Write-Through
Bit  2:     User/Supervisor (0=kernel, 1=user accessible)
Bit  1:     Read/Write (0=read-only, 1=writable)
Bit  0:     Present (0=not present, 1=present)
```

Important flags:

- `PRESENT (0x001)`: Must be 1 for the PDE to be valid. If clear, accessing any address under this PDE's range triggers a page fault.
- `WRITABLE (0x002)`: Controls write permissions for all pages covered by this page table.
- `USER (0x004)`: Determines whether user-mode code can access pages in this page table.

5.2.4 Page Table Entry (PTE) Format

Each PTE has a similar structure to PDEs:

```
Bits 31-12: Physical frame address (20 bits, 4KB-aligned)
Bits 11-9:  Available for OS use
Bit  8:     Global
Bit  7:     Page Attribute Table
Bit  6:     Dirty (written to)
Bit  5:     Accessed
Bit  4:     Cache Disabled
Bit  3:     Write-Through
Bit  2:     User/Supervisor
Bit  1:     Read/Write
Bit  0:     Present
```

Important flags:

- `PRESENT (0x001)`: Must be 1 for the page to be accessible.
- `WRITABLE (0x002)`: Controls whether the page is read-only (0) or writable (1).
- `USER (0x004)`: Determines kernel-only (0) vs user-accessible (1) pages.
- `ACCESSED (0x020)`: Automatically set by CPU when page is accessed.
- `DIRTY (0x040)`: Automatically set by CPU when page is written to.

5.2.5 The CR3 Register and Context Switching

The `CR3` control register holds the physical address of the ***current*** Page Directory. When you switch processes, you load a different page directory address into CR3, instantly changing the entire virtual address space visible to the CPU.

To switch page directories:

```
uint32_t phys_addr = (uint32_t)VIRT_TO_PHYS(page_directory);
asm volatile("mov %0, %%cr3" :: "r"(phys_addr));
```

To read CR3:

```
uint32_t cr3_value;
asm volatile("mov %%cr3, %0" : "=r"(cr3_value));
```

5.2.6 Translation Lookaside Buffer (TLB)

The TLB is a CPU cache that stores recent virtual-to-physical address translations. Without it, every memory access would require multiple memory reads to walk the page tables.

When you modify page tables, you must flush the TLB to ensure the CPU doesn't use stale translations.

5.2.7 Clearing Frames

When referencing the Kernel Memory Manager (KMM) to allocate new frames, you must remember to always clear the newly allocated frames.

Note

Use `memset(frame_virt, 0, sizeof(your_structure))`

5.3 Useful data types

Here's some useful data types you should include in your implementation to make it more readable. The functions expected from you use these data types too. These data types along with some useful constants are found in the `include/mm/{pde.h,pte.h}`.

```
typedef uint32_t pte_t;  // Page Table Entry
typedef uint32_t pde_t;  // Page Directory Entry

typedef struct {
    pte_t table[VMM_PAGES_PER_TABLE];  // 1024 entries
} pagetable_t;

typedef struct {
    pde_t table[VMM_PAGES_PER_DIR];    // 1024 entries
} pagedir_t;
```

5.4 Requirements

You are required to implement the following functions to support virtual memory management in your kernel.

- `pagedir_t* vmm_get_kerneldir(void)`

  Returns a pointer to the kernel's main page directory. Note that this will be a virtual address.

- `pagedir_t* vmm_get_current_pagedir(void)`

  Returns a pointer to the page directory currently loaded in the CPU's CR3 register. Note that the function returns a virtual address.

- `pagedir_t* vmm_create_address_space(void)`

  Creates a new, empty virtual address space by allocating a page directory. It should:

  – Allocate a physical frame for the page directory structure.

  – Initialize the directory appropriately to ensure all entries are invalid.

  – Return the virtual address of the page directory, or NULL on failure.

  Hint

  – Frames are allocated as physical addresses but must be accessed through virtual addresses.

  – Always clear newly allocated paging structures.

- `bool vmm_switch_pagedir(pagedir_t* new_pagedir)`

  Switches the CPU to use a different page directory, changing the active virtual address space. It should:

  – Validate the input parameter.

  – Update the `CR3` register to track the current page directory.

  – Ensure that the new page directory will now be used for address translation.

  – Return true on success, false on failure.

- `void vmm_create_pt(pagedir_t* pdir, void* virtual, uint32_t flags)`

  Creates a new page table for the given virtual address range within a page directory. It should:

  – Determine which directory entry corresponds to the virtual address.

  – Check if a page table already exists — if so, no action is needed.

  – Allocate and initialize a new page table structure.

  – Update the page directory to reference the new page table with appropriate flags.

- `void vmm_map_page(pagedir_t* pdir, void* virtual, void* physical, uint32_t flags )`

  Maps a single virtual page to a physical frame with specified permissions. It should:

  - Ensure a page table exists for the virtual address (create if necessary).
  - Locate the appropriate entry within the page table.
  - Configure the entry to point to the physical frame with the given flags.
  - Note that this function only does the mapping by putting appropriate entries in the page tables. It doesn't allocate any physical frame itself.

- `void* vmm_get_phys_frame(pagedir_t* pdir, void* virtual)`

  Translates a virtual address to its corresponding physical frame address. It should:

  - Validate inputs and check if a page table exists for this address.
  - Navigate through the paging structures to find the relevant entry.
  - Extract and return the physical frame address, or NULL if not mapped.

- `void _vmm_page_fault_handler(interrupt_context_t* ctx)`

  This is the default handler to handle page faults (exception nr. 14). Note that the page fault is the type of exception that pushes an error code on the stack. Its bit structure is as follows:

  - **`Bit 0 (P)`**: Set if the fault was caused by a protection violation (page present but access not allowed); clear if caused by a non-present page.
  - **`Bit 1 (W/R)`**: Set if the fault occurred during a write; clear if during a read.
  - **`Bit 2 (U/S)`**: Set if the fault originated in user mode; clear if in supervisor (kernel) mode.
  - **`Bit 3 (RSVD)`**: Set if the fault was caused by reserved bits being set in a page table entry.
  - **`Bit 4 (I/D)`** Set if the fault occurred during an instruction fetch (on some CPUs).

  You can use this error code to determine the type of error that occurred. Similarly, the faulting address (the virtual memory access that caused the fault) can be retrieved from the register `cr2`. A good interrupt handler will help you catch your errors quickly. Normally, the kernel executes normal execution after a page fault, but in this case, you can feel free to halt the machine.

- `void vmm_init(void)` Initializes the kernel's Virtual Memory Manager and enables paging. It should:

  - Register the page fault interrupt handler. Use the function `register_interrupt_handler` implemented in the previous assignment to register the handler for the interrupt/exception number 14.
  - Create a new address space for the kernel.
  - Set up identity mapping for the first 1MB of physical memory (0x0 to 0x100000), mapping each 4KB page with virtual address equal to physical address.
  - Set up the physmap by mapping all available physical memory to high memory starting at `PHYSMAP_BASE` (3GB).
  - Switch to the newly created kernel page directory.
  - Store the kernel's page directory, as well as the currently active page directory in a private state variable.

Hint

- Use functions from your physical memory manager to retrieve information for all the memory.

- `int32_t vmm_page_alloc(pte_t* pte, uint32_t flags)`

  Allocates a physical frame and associates it with the given Page Table Entry. It should:

  - Validate the PTE pointer.
  - Check if the entry already has a frame allocated (idempotent behavior).
  - Allocate a new physical frame and create a PTE to reference it.
  - Return 0 on success, -1 on failure.

  If the PTE is already present, this should succeed without allocating a new frame. Handle out-of-memory conditions gracefully.

- `void vmm_page_free(pte_t* pte)` Frees the physical frame associated with a Page Table Entry and marks it as not present. It should:

  - Validate the PTE and check if it references a frame.
  - Return the physical frame to the free pool.
  - Mark the entry as not present to prevent future access.

  Always free the frame before clearing the present bit, otherwise you'll lose the physical address.

- `bool vmm_alloc_region(pagedir_t* pdir, void* virtual, size_t size, uint32_t flags )`

  Allocates a contiguous region of virtual memory by allocating frames for multiple pages. It should:

  - Validate parameters.
  - Determine the range of pages to allocate based on the starting address and size.
  - For each page, ensure the necessary paging structures exist and allocate a frame.
  - Return true if all allocations succeed, false otherwise.

  Hint: Iterate through the region one page at a time. Consider how to handle unaligned addresses and sizes.

- `bool vmm_free_region(pagedir_t* pdir, void* virtual, size_t size)`

  Frees a contiguous region of virtual memory, including the physical frames and potentially empty page tables. It should:

  - Validate parameters (return false if invalid).
  - Calculate the range of pages to free based on the virtual address and size.
  - For each page in the region:
    - Check if the page table exists (skip if PDE is not present).
    - Free the physical frame associated with the page using `vmm_page_free()`.

    - Clear the page table entry.
    - Invalidate the TLB entry for this virtual address.
  - After freeing all pages, check if any page tables are now completely empty.
  - Free empty page table structures to prevent memory leaks.
  - Return true on success.

Hint

The TLB must be flushed for each unmapped page using `invlpg`.

- `pagetable_t* vmm_clone_pagetable(pagetable_t* src)`

  Creates a deep copy of a page table, duplicating both the structure and the physical frames it maps. It should:

  - Validate that `src` parameter (return NULL if invalid).
  - Allocate a new physical frame for the cloned page table and clear it.
  - Iterate through all entries in the source page table.
  - For each present entry:
    - Allocate a new physical frame for the data.
    - Copy the entire 4KB page contents from source to destination using `memcpy()`.
    - Set up the new PTE with the same flags as the original.
  - Return the virtual address of the new page table.

- `pagedir_t* vmm_clone_pagedir(void)`

  Clones the current address space, creating a complete copy suitable for process creation. It should:

  - Create a new empty page directory using `vmm_create_address_space()`.
  - Iterate through all entries in the current page directory.
  - For each present PDE:
    - If it's a kernel mapping (matches the kernel directory), perform a shallow copy by copying just the PDE.
    - If it's a user mapping, perform a deep copy using `vmm_clone_pagetable()`.
    - Set up the new PDE to point to the appropriate page table.
  - Return the new page directory.

  Kernel mappings are shared across all processes (shallow copy), while user mappings must be isolated (deep copy). Compare PDEs with the kernel directory to identify kernel vs. user mappings.

Note

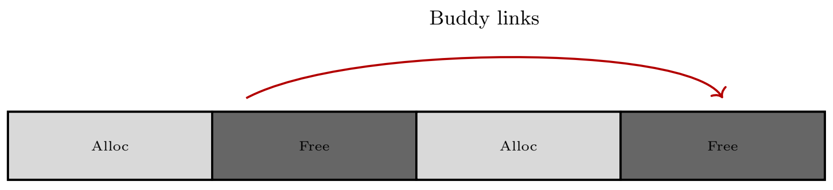
- Physical addresses obtained from the KMM must be converted before dereferencing.
- Addresses stored in paging structures (PDEs, PTEs, CR3) must be physical.
- Newly allocated paging structures must be cleared to prevent garbage data from appearing valid.
- Modification of page mappings requires invalidating CPU caches (TLB).
- Comprehensive logging helps debug complex paging issues and page faults.

5.5 Implementation files

Start by adding your implementation in `include/mm/vmm.h` and `mm/vmm.c`.

6 Part 3: Kernel Heap Allocator (KHEAP)

The Kernel Heap Allocator (KHEAP) provides dynamic memory management within the kernel's virtual address space using a **buddy allocation system**. It allows the kernel to request and release variable-sized memory blocks at runtime for components such as process tables, buffers, and driver data.



Heap layout under the buddy system

Figure 8: Buddy-based kernel heap: allocated (gray) and free (pink) blocks.

6.1 Overview

The buddy allocator divides the heap into power-of-two-sized blocks. Each allocation request is rounded up to the nearest power of two. When memory is freed, the allocator checks whether its "buddy" block (the adjacent region of the same size) is also free, if so, the two are merged to form a larger block. This recursive merging keeps fragmentation low and allocation efficient. Figure 9 shows the heap state over multiple allocations and deallocations.

6.2 Responsibilities

Your implementation should:

- Initialize the heap over a given region of memory.
- Provide dynamic allocation via `kmalloc()` and deallocation via `kfree()`.
- Support resizing allocations through `krealloc()`.

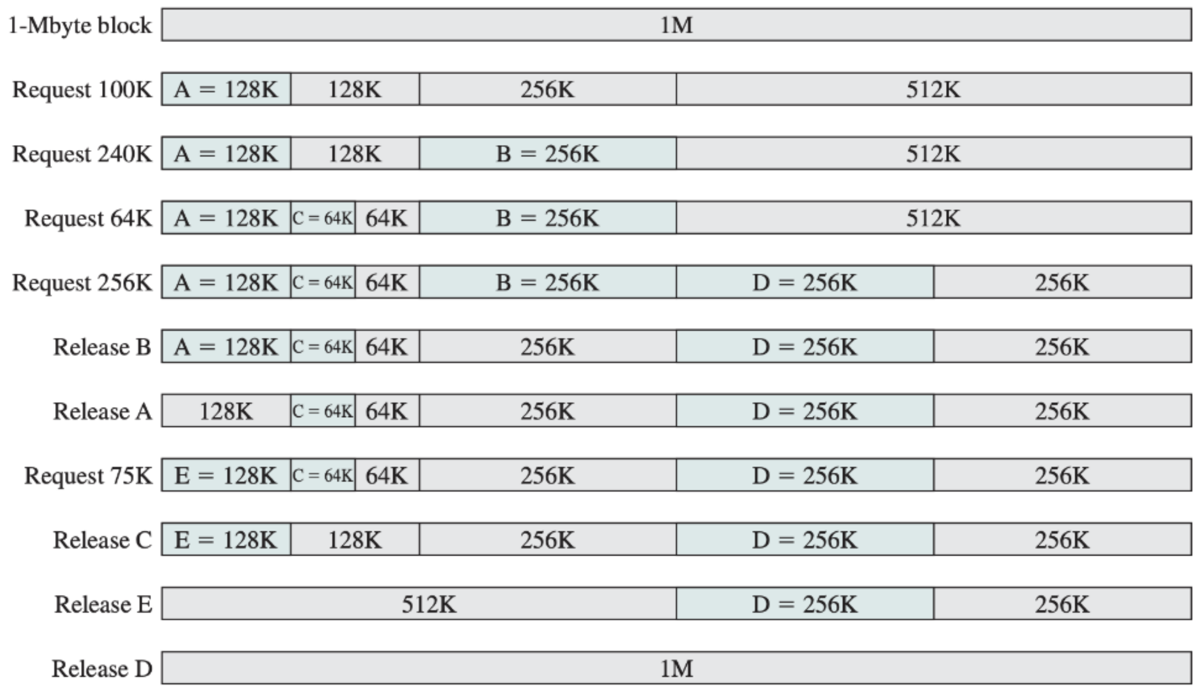


Figure 9: An overview of how buddy allocator works

- Split larger blocks when smaller requests are made.
- Merge ("coalesce") freed blocks with their buddies when possible.
- Detect and safely handle invalid or double frees.

6.3 Functions to Implement

You must implement the following core functions. Each is described conceptually below. The data type `heap_t` is defined in `kheap.h`.

- `void kheap_init(heap_t *heap, void *start, size_t size, ...)`

  Initializes the kernel heap within the provided memory range.

  - Aligns the heap start address and total size to the minimum block size (a power of two).
  - Determines how many block "orders" can fit within the heap (e.g., 32B, 64B, 128B, ... up to heap size).
  - Creates a series of free lists, one per order, and populates them so that initially the heap contains one large free block.
  - Stores internal bookkeeping information (such as base address, size, and free-list heads).
  - Finally, the kernel must map this provided virtual memory region, using the earlier implemented `vmm_alloc_region()`.

  After initialization, the heap is ready to serve dynamic allocations using the buddy algorithm.

- `void* kmalloc(heap_t *heap, size_t size)`

  Allocates a memory block from the heap.

  - Adds space for a small header (to store block size and validation tag).
  - Rounds the total size up to the nearest power of two (the smallest block order that fits).
  - Searches the free list for a block of the required order or larger.
  - If a larger block is found, it is repeatedly split into two halves until the target order is reached.
  - Marks one half as allocated and returns its address to the caller.

  This function should gracefully handle out-of-memory situations by returning `NULL` if no suitable block exists.

- `void kfree(heap_t *heap, void *ptr)`

  Frees a previously allocated block.

  - Retrieves the block's header to determine its size and validate its integrity using a magic number.
  - Locates the block's buddy, the adjacent block of the same size.
  - If the buddy is also free, merges the two blocks into a larger one (increasing their order).
  - Continues merging while possible, ensuring the free lists remain coalesced and minimal.
  - Reinserts the resulting free block into the appropriate list.

  The allocator must safely handle invalid or double frees without corrupting the heap.

- `void* krealloc(heap_t *heap, void *ptr, size_t new_size)`

  Resizes an existing allocation.

  - If `ptr` is `NULL`, behaves like `kmalloc()`.
  - If `new_size` is zero, frees the block.
  - If the existing block can accommodate the new size, returns it unchanged.
  - Otherwise, allocates a new block, copies the old contents, and frees the old one.

  This function enables flexible memory management for dynamic kernel structures.

- `void kheap_stats(heap_t *heap)`

  (Optional) Prints diagnostic information about heap state.

  - Displays heap boundaries, alignment, and configuration flags.
  - Lists how many free blocks exist at each order (e.g., how fragmented the heap is).

  This is primarily for debugging and performance analysis.

Analogy: Buddy System as Lego Blocks

Think of the heap as a box of Lego bricks. Each brick can be split into two smaller pieces or joined back together. When you allocate memory, you pick the smallest brick that fits your structure. When you free memory, the allocator checks if the neighboring brick of the same size is also free, if it is, they snap together to form a larger piece again. This keeps the heap neat, balanced, and easy to reuse.

7 Getting Started

This section describes the setup process for building, running, and testing your kernel on various platforms. It covers installation of required dependencies, compilation, debugging, and the test framework used to validate your memory management subsystems.

7.1 Installation and Environment Setup

Before you begin, ensure your development environment includes the necessary build tools, cross-compilers, and emulators. The instructions differ slightly for **Linux** and **macOS** systems.

Linux Installation

To install all required dependencies on a Debian-based system (e.g., Ubuntu), run the following commands:

```
sudo apt update
sudo apt install build-essential gcc-multilib qemu-system-x86 gdb make
   python3 python3-pip mtools dosfstools
# Install Python testing dependencies (preferrably in a venv)
pip3 install pytest
```

macOS Installation

For macOS users, it is recommended to use **Homebrew** to install the cross-compilation toolchain and other utilities.

```
# Install Homebrew if not already installed
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
   install/HEAD/install.sh)"

# Install the i686 cross-compiler toolchain (for 32-bit kernel builds)
brew install i686-elf-gcc i686-elf-binutils i386-elf-gdb

# Install supporting tools
brew install make git qemu mtools dosfstools python@3.11

# Install Python testing dependencies
pip3 install pytest
```

After installation, verify that the following commands are available in your terminal: `gcc`, `make`, `qemu-system-i386`, `gdb`, and `pytest`.

7.2 Building and Running

Clean Build

To ensure a fresh build, clean any previous compilation artifacts before compiling:

```
make clean
make
```

This produces a bootable disk image named `disk.img`, which contains your compiled kernel and supporting files.

Running in QEMU

To run the kernel in QEMU (a hardware emulator), execute:

```
make qemu
```

QEMU will boot your kernel and display its console output. This is the fastest way to verify that your system boots and initializes correctly, while outputting values as expected.

Running with GDB Debugger

For in-depth debugging, QEMU can run in a mode that allows you to attach a GDB session. This lets you set breakpoints, inspect registers, and step through kernel code:

```
make qemu-dbg
```

This workflow is invaluable when diagnosing issues in paging, heap management, or interrupt handling.

7.3 Running the Test Suite

Testing ensures correctness of your memory management implementations (KMM, VMM, and KHEAP).

Running All Tests

Open two terminal windows:

1. In the first terminal, start QEMU in test mode:

   ```
   make test
   ```

   This automatically rebuilds the kernel with testing macros enabled and launches QEMU.

2. In the second terminal, navigate to the `tests/` directory and execute:

   ```
   cd ./tests
   pytest
   ```

   This runs the complete test suite against your kernel implementation.

Running Specific Tests

You can target specific test files to validate individual subsystems:

```
# Run tests matching a keyword
pytest -k "kmm"
pytest -k "vmm"
pytest -k "kheap"

# Run individual component tests
pytest -m mm/test_kmm.py      # Tests Physical Memory Manager (KMM)
pytest -m mm/test_vmm.py      # Tests Virtual Memory Manager (VMM)
pytest -m mm/test_kheap.py    # Tests Kernel Heap Allocator (KHEAP)

# Run tests with detailed output
pytest -v
```

A quick little shortcut to run compilation and tests quickly with a single command:

```
# adjust sleep time according to your compilation time
make test & sleep 3 && cd tests && pytest ; cd ..
```

7.4 Debugging Test Failures

If a test fails, rerun it with verbose output (`-v`) to see detailed logs. For deeper debugging:

- Add `printf()/terminal_write()` statements in your kernel code.
- Use `make qemu-dbg` to use GDB to inspect memory or stack frames.
- Review kernel log messages printed to the VGA terminal.

7.5 Tips for Development

- Always start with a clean build after modifying low-level memory structures.
- Use small test heaps (e.g., 4 KB) during debugging to simplify visualization of allocations.
- Step through your allocator (KHEAP) or pager (VMM) logic using GDB breakpoints.
- Consult `kheap_stats()` to print internal buddy allocator state.

These steps will help you efficiently develop, debug, and validate each part of your kernel memory management subsystem.

8 Grading Criteria

Your assignment will be evaluated using an automated testing framework designed to ensure fair and consistent grading across all submissions. Understanding the grading structure will help you focus your implementation efforts effectively.

8.1 Grade Distribution

The total grade for this programming assignment is divided into two major components:

- **Visible Tests (70%):** These are the test cases provided in your development environment. You can execute them using `pytest` to verify correctness during implementation. The visible test suite consists of **23 individual test cases**, and each contributes equally to this portion of your grade.
- **Hidden Tests (30%):** These are additional tests that assess your kernel's handling of edge cases, rare conditions, and compliance with function specifications. Hidden tests are not available during development but follow the same logical structure as the visible ones. They focus on robustness, correctness under stress, and adherence to expected memory safety behavior.

8.2 Maximizing Your Grade

To achieve the highest possible score, follow these best practices throughout your implementation:

1. **Ensure all visible tests pass.** This guarantees at least 70% of your grade and confirms that the basic functionality is correct.
2. **Implement robust error handling.** Many hidden tests explicitly target error scenarios such as invalid frees, double allocation, or out-of-memory conditions.
3. **Follow all function specifications precisely.** Hidden tests verify strict API conformance, parameter validation, and return values. Even minor deviations (e.g., incorrect return on error) may cause test failures.
4. **Test beyond the provided suite.** Manually create additional test cases that explore corner cases, alignment issues, and coalescing behavior in your allocator.
5. **Review and refactor your code.** Hidden tests are designed to catch subtle bugs such as incorrect pointer arithmetic, uninitialized state, or missed coalescing opportunities.

A methodical approach to testing and attention to implementation details will not only maximize your score but also ensure your kernel memory management subsystems are correct, efficient, and stable.

9 Submission Requirements

Before submitting your work for **Programming Assignment 2 (PA2)**, ensure your project directory is properly cleaned and packaged according to the following steps:

1. Run `make clean` to remove all compiled binaries, object files, and build artifacts.
2. Compress your entire PA2 project directory into a single ZIP archive.
3. Use the following naming convention for your submission file:

   `s_<rollnumber>.zip`

   where `<rollnumber>` should be replaced with your actual student roll number. For example:

   `s_24100173.zip`

4. Upload the ZIP archive to the Learning Management System (LMS) before the submission deadline.

Failure to follow this submission format (especially filename conventions or cleanup steps) may result in automated grading errors or penalties.

10 General Tips and Best Practices

Programming Assignment 2 builds directly upon the foundations established in PA1. The tasks in this phase demand deeper reasoning about kernel subsystems, especially memory management and paging. Follow these recommendations to streamline your development and debugging process:

1. **Start early.** Kernel-level development becomes significantly more intricate in PA2. Subtle logic errors in the memory manager or paging system can lead to complex bugs. Begin as soon as possible to allow time for investigation and debugging.

2. **Engage actively on the PA2 Slack channel.** Discussion among peers can help uncover conceptual misunderstandings quickly. Check the provided FAQ and previous posts before asking questions to avoid duplicates.

3. **Adopt an incremental development approach.** Implement one subsystem at a time, for instance, complete and test the kernel heap before proceeding to virtual memory (VMM). Each layer in PA2 depends on the correctness of earlier ones, so frequent testing and commits are crucial.

4. **Review all documentation and specifications thoroughly.** Every function prototype and expected behavior described in this manual is a requirement, not a suggestion. Deviating from these specifications will cause hidden tests to fail.

5. **Test frequently and systematically.** Use `pytest` after each feature milestone. The earlier you isolate a failure, the easier it is to locate and fix the root cause.

6. **Use debugging tools effectively.** Utilize `GDB` with QEMU to set breakpoints, inspect page tables, and verify frame allocations. Read all error messages carefully, they often reveal subtle inconsistencies in pointer arithmetic or mapping logic.

7. **Write your own code.** While conceptual help from online resources or LLMs may clarify ideas, directly generated code tends to introduce deep, hard-to-detect kernel bugs. Understanding your implementation fully is essential, both for debugging and for success in future PAs.

Remember, PA2 introduces real kernel-level memory management. With structured planning, incremental testing, and persistence, you'll master this critical part of operating system development.

Good Luck, and Happy Coding!