TeachOS
Documentation

Design and Implementation of an OS for Use in the Operating Systems Lecture Track

Version: 1.0
Date: 2024-12-17 11:09:15+01:00

Matteo Gmür, Fabian Imhof

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Contents

I Abstract 4

II Goals 5

1 Memory management 5
- 1.1 Memory management unit (MMU) and page tables . . . 6
  - 1.1.1 Single-Level page table . . . 6
  - 1.1.2 Two-Level page table . . . 6
  - 1.1.3 Multi-Level page table . . . 6
- 1.2 Translation lookaside buffer (TLB) . . . 6
- 1.3 Inverted page-table (IPT) . . . 6
- 1.4 Hashed page-table . . . 7
- 1.5 Paging-strategies . . . 7
  - 1.5.1 Fetching policies . . . 7
  - 1.5.2 Cleaning policies . . . 7
  - 1.5.3 Page replacement policies . . . 8
- 1.6 Heap-Implementation . . . 9
  - 1.6.1 Multiple fixed block size . . . 9
  - 1.6.2 Bitlist (linked list) . . . 9
  - 1.6.3 Buddy-System . . . 9
  - 1.6.4 Object-Pools . . . 9
  - 1.6.5 Search-Algorithms for free memory . . . 9

III Research 11

2 Memory management 11
- 2.1 Parsing multiboot2 . . . 11
  - 2.1.1 Executable and linkable format (ELF) . . . 13
- 2.2 Frame allocator . . . 14
- 2.3 Page tables . . . 15
- 2.4 Referencing page tables . . . 16
- 2.5 Mapping page tables . . . 17
  - 2.5.1 Accessing entries . . . 17
  - 2.5.2 Mapping virtual pages into physical frames . . . 18
- 2.6 Owning the page table . . . 19
- 2.7 Remapping the Kernel . . . 20
  - 2.7.1 Unmapping workaround . . . 21
  - 2.7.2 Mapping kernel elf sections . . . 21
- 2.8 Kernel heap . . . 22
  - 2.8.1 Bump heap allocator . . . 22
  - 2.8.2 Linked-List heap allocator . . . 24

IV Conventions 26

3 Assembly syntax **26**

4 C++ syntax **26**
4.1 Function arguments 26
4.2 Bitset flag usage 26
4.3 Indexable enum value 27
4.4 Containers with std::optional values 28
4.5 Inline assembly 28
4.5.1 Output operands 28
4.5.2 Input operands 29
4.5.3 Clobbered registers list 29

V Results **30**

5 Physical memory allocator **30**

6 Virtual memory management **30**

7 Heap memory allocator **30**

VI Future work **31**

Part I

Abstract

This paper examines all aspects of memory management in a custom operating system. The goal is to create a foundation for an interactive learning tool to be used in the "Operating Systems 1" and "Operating Systems 2" lectures at OST. Topics covered include parsing Multiboot2 information, implementing frame allocation and page tables, and using these components to remap the kernel. Based on this, dynamic memory allocation during kernel runtime was implemented using different allocation strategies.

The objective of this work is to develop a modular and extensible memory management system encompassing physical and virtual memory management as well as heap management. This provides a robust foundation for extending other kernel components while serving as a learning platform for hands-on exploration of operating system concepts.

The developed operating system includes:

Physical memory allocator: Manages the allocation and deallocation of physical memory. A simplistic version was implemented without deallocation functionality.

Virtual memory management: Implements multilevel page tables to separate logical and physical address spaces.

Heap memory allocator: Enables dynamic memory management within the kernel heap.

This foundation can be extended in the future to include additional kernel functionalities and to expand the feature set of the existing memory management system.

Part II

Goals

The overarching goal of this research paper is to provide a simple and readable operating system for students at OST, to learn about operating systems in a hands-on and interactive way.

This section defines the functionality we have implemented as well as those that are logical next steps. Generally we used the osdev wiki[1] to research possible and realistic operating system responsibilities.

The cross-compiler and printing text has already been provided before we started this project.

1 Memory management

Implementing memory management is the top priority because it is one of a kernel's most fundamental functions. This feature includes:

- Physical Memory Manager
- Virtual Memory Manager
- Heap Memory Manager

In the interest of using this operating system as an educational tool, below are some more features, that should be implemented on top of the previously mentioned ones in the future.

- Common memory allocation strategies
- Common page management algorithms
- Displaying the current state of memory of a user-space application

The idea is to implement a simple memory management strategy first, and once that is up and running, it can be expanded to allow switching between multiple different types of virtual- and physical memory management strategies.

Changing the memory management strategy should be implemented through compile-time changes. Doing so at run-time would require allocating a certain portion of memory to create a default memory manager which then allocates the actual memory manager depending on the given run-time arguments. This is currently not a priority functionality and due to the complexity of it will be omitted.

The prioritized memory management patterns and algorithms, are based on the "Operating Systems 1" lecture on Dynamic Memory and are listed below.

[1] [wiki, 2024]

1.1 Memory management unit (MMU) and page tables

The MMU handles the mapping of virtual to physical memory addresses. Page Tables are the data structures that enable this mapping process and there are a few different possible implementations, which mainly differ in a trade-off between memory consumption and lookup efficiency.[2]

Certain features are already predefined or are required by the x86 architecture. Namely, the 4-level page tables using a Translation Lookaside Buffer (TLB). Therefore, those features have to be implemented first and can then be followed by the different paging strategies.

1.1.1 Single-Level page table

A single-level page table consists of a single page table containing multiple page table entries. The amount of entries depends on the page size (usually 4 KiB) and the architecture of the CPU (64-Bit or 32-Bit). Each entry corresponds to a virtual page number and contains the frame number in which the page is stored and some additional metadata. The virtual address is divided into the page number (upper bits) and the page offset (lower bits). When a virtual address is accessed, the MMU uses the page number to find the physical frame number and then combines the physical frame number with the page offset to retrieve the physical address.

1.1.2 Two-Level page table

Two-level page tables work inherently the same as a single-level Page Table. The difference lies in the number of nested page tables. Instead of one-page table pointing to a page, the first page table points to another page table, which in turn points to a page.

1.1.3 Multi-Level page table

Extending the hierarchy beyond two levels. Mainly done too further optimize memory usage for large address spaces. This is because each level contains fewer entries, which results in fewer data being loaded into memory.

1.2 Translation lookaside buffer (TLB)

The TLB is a special cache which stores recently used page table entries. This results in much faster load times in case of a TLB hit (accessed page table entry is in cache). However on a TLB miss (accessed page table entry is not in cache) the TLB checks if the page is already in main memory. If not, a page fault is issued and the TLB updates to include the accessed page.

1.3 Inverted page-table (IPT)

In the previously explained page tables, the operating system must translate the virtual address references into a physical memory address. One drawback of this strategy is that each page table may contain a million entries. This can

[2][Abraham Silberschatz and Gagne, 2018]

consume large amount of physical memory. The IPT solves this problem by having one entry for each physical frame. These entries contain the virtual page address and information about the owner process of that page. This results in only one-page table being loaded in the system.

1.4 Hashed page-table

In a hashed page table the hash value is the virtual page number. Each entry contains a linked list of elements with three fields. The virtual page number, the mapped page frame and a pointer to the next element in the linked list. This method is designed for address spaces larger than 32 bit.

1.5 Paging-strategies

Efficient paging is critical to the performance of virtual memory systems. Paging strategies govern when and how pages are loaded into or removed from memory.

1.5.1 Fetching policies

Demand Paging

In demand paging, pages are loaded into memory only when accessed. This lazy-loading strategy minimizes memory usage but may cause higher initial latency during access.

Pre-paging

Pre-paging proactively loads pages that are likely to be accessed soon, reducing future latency. However, this approach increases memory usage and wastes resources if predictions are inaccurate.

Demand Paging with Pre-paging

This hybrid approach combines demand paging and pre-paging, leveraging historical or predictive data to load pages selectively.

1.5.2 Cleaning policies

Demand Cleaning

Demand cleaning is the counterpart to demand paging (Section 1.5.1). Pages will only be written back to disk when they are not being used and there is not enough space for new pages.

Pre-Cleaning

Pre-cleaning is a strategy wherein dirty pages are written to disk proactively. This improves performance when replacing pages at the cost of additional write operations.

1.5.3 Page replacement policies

First-In-First-Out (FIFO)

FIFO replaces the oldest page in memory, regardless of any other criteria. This simple algorithm results most likely in suboptimal performance, because access frequency is not taken into consideration.

Second chance

Second chance extends FIFO by using an access bit. When a page is being accessed, the bit will be set. When a page is replaced, only pages without the access bit set will be replaced according to FIFO. All access bits are then reset to 0.

Clock

The clock algorithm essentially does the same as Second chance. However, it is more efficient and designed to minimize scanning overhead.

Least recently used (LRU)

LRU replaces the least recently accessed page. This strategy resembles the usage of the pages but tracking the access order is resource intensive.

Not frequently used (NFU)

NFU imitates LRU without tracking page access order. This is achieved by adding a counter for page access. Pages with low counts are replaced. This has the downside, that long living pages will not be replaced for a long time regardless of recent access, due to a previously accumulated high counter.

Not frequently used with aging

This strategy fixes the mentioned issue with NFU, by adding aging to the counters. All counters will be decreased after a set amount of time to generally keep the aging counters low.

Working set

This algorithm manages memory by maintaining the set of pages a process has accessed within a defined time window. That time window is called the working set. Pages not accessed within the time frame are considered outdated and are marked to be unloaded if needed.

Working set clock

Working set clock extends the Working set algorithm with the Clock algorithm. Pages in memory are organized in a circular list, with each entry storing the page's last access time. Rotating through all pages, they are checked if they have been accessed within the last working set window. If it has, the time stamp is updated. If not, the page is marked to be unloaded.

1.6 Heap-Implementation

1.6.1 Multiple fixed block size

Memory is allocated in blocks of the same size. This simplifies memory management as allocation and deallocation are more straightforward.

The metadata to manage these blocks can be stored in two different ways:

Central metadata: A single central structure stores all information about free and allocated memory blocks.

Decentral metadata: Each memory block keeps track of its own metadata.

1.6.2 Bitlist (linked list)

A bit-list can be used to manage free and allocated memory blocks. It represents the allocation status of memory blocks using a series of bits (0 for free, 1 for allocated).

A linked list is used for dynamic memory allocation where free blocks are chained together, making it easy to allocate and free memory.

1.6.3 Buddy-System

The Buddy System divides blocks of memory into pairs (buddies) that can be merged or split as needed. The system keeps track of available memory blocks and their buddies, allowing efficient memory allocation and deallocation. When a block is freed, the system checks if its buddy is also free. If that is the case, they are merged back into a larger block.

1.6.4 Object-Pools

An object pool is used to manage and reuse memory blocks for objects of the same type, reducing the overhead of frequent allocation and deallocation. Instead of allocating new memory for every object, the pool maintains a set of pre-allocated objects, which can be reused when needed. This improves performance, especially for systems that frequently allocate and deallocate objects of the same type.

1.6.5 Search-Algorithms for free memory

Search-Algorithms are used to find available (free) memory blocks when memory allocation is requested.

First fit

Searches for the first available block large enough to hold the desired data.

Next fit

Similar to First Fit but continues searching from where the last allocation was made, offering faster allocation if a lot of memory is already occupied, but missing if bigger blocks are freed again.

Best fit

Searches for the smallest available block that fits the requested size, aiming to minimize wasted space. This results in the longest wait times of all search algorithms, because the list of free blocks is always traversed from start to finish.

Worst fit

Searches for the largest available block to allocate, hoping to leave large enough fragments for future use.

Quick fit

Uses several lists of free blocks categorized by size ranges. This allows faster allocation by quickly selecting the appropriate size class.

Following the implementation of a heap, all the aforementioned strategies and algorithms can be implemented.

Part III

Research

2 Memory management

The ”Memory Management” section in the blog ”Writing an OS in Rust (First Edition)”[3] was the primary source for setting up memory management. The most relevant findings are summarized below.

2.1 Parsing multiboot2

Initially, it is essential to load the multiboot2 information structure. It contains vital boot information needed in each of the following sections. Loading this information be achieved by accessing the contents of the `ebx` register

At this stage, the assembly code simply stores the `ebx` register's value into an external `size_t` variable named `multiboot_information_pointer`. This pointer enables access to boot information in C++. The relevant types and fields of the multiboot2 pointer are described in the table below:

Value	Tag Name	Size	Meaning
0	End	8 Bytes	Signals final tag for the multiboot2 information structure
6	Memory Map	variable	Describes the memory layout of the system
9	ELF Symbols	variable	Includes a list of all section headers from the loaded ELF kernel

Figure 1: Used multiboot2 tags [Ford and Boleyn, 1995]

The ”End” tag is needed to iterate through all existing tags within the multiboot2 information structure and serialize their data. Because the data of the underlying tags is different for every of the aforementioned tag types and may differ in size, it is essential to align the `for` loop to 8 Bytes.

This alignment is necessary because the minimum size of a tag is 8 Bytes, as each tag comprises at least two `uint32_t` values. One representing the type and the other indicating the total size of the tag.

[3][Oppermann, 2024]

```
template<typename T> requires std::is_pointer<T>::value
auto align_to_8_Byte_boundary(T ptr, uint32_t size) -> T
{
  return reinterpret_cast<T>(
         reinterpret_cast<uint8_t *>(ptr) + ((size + 7) & ~7)
  );
}

for (auto tag = multiboot_tag; tag->type != tag_type::END;
     tag = align_to_8_Byte_boundary(tag, tag->size))
{ ... }
```

The most recent specification for multiboot2[4] is outdated. The correct data types for the fields `num`, `entsize`, `shndx`, and `reserved` within the ELF-Symbols multiboot2 tag are `uint32_t`, rather than `uint16_t`. Furthermore, the definitions of the Memory Area Entries in the multiboot2 Memory Map Header are also incorrect, as they mistakenly place a reserved variable at the end of the structure instead of the start.

To obtain the correct and up-to-date specification it is recommended to visit the GRUB2 repository[5] instead and look at the `include/multiboot2.h` and `include/multiboot.h` files.

Additionally, the `reserved` variable present in the `struct` referenced in the official documentation is only required if accessed using the C language. When using C++11 or newer, this variable can be omitted. Instead, one can apply the `alignas(8)` specifier to the last `struct` variable. This will fulfill the exact same purpose of aligning to 8 Bytes without the need for an additional structure variable, which should never be accessed.

```
struct multiboot::info_header
{
  uint32_t total_size;
  // Used to ensure we have 2 uint32_t fields to align to 8 Bytes.
  // Not required anymore because alignas(8) does the same
  //uint32_t reserved;
  alignas(8) struct multiboot::tag tags;
};
```

One exception to this statement is in the elf symbols table. There the reserved keyword should not be replaced with `alignas`, because the section headers start directly after the `section_index` with values amounting to 20 Bytes and the section headers being 4 Byte aligned.

To ensure the data is read in 4 Byte aligned mode, the struct must not directly contain the elf symbols table. Instead, a `std::byte` is used which will be cast to a `elf_section_header` pointer using `reinterpret_cast<>`.

[4][Ford and Boleyn, 1995]
[5][rhboot, 2024]

```cpp
struct elf_symbols_section_header
{
  tag info;
  uint32_t number_of_sections;
  uint32_t entry_size;
  uint32_t section_index;
  // Struct can not be used directly because if we continue,
  // to directly read the struct, we will read in 8 Byte alignment
  // and therefore skip data partially or read more than we should
  //uint32_t reserved;
  //struct elf_section_header sections;
  std::byte end;
};
```

2.1.1 Executable and linkable format (ELF)

The elf format consists of multiple parts, the first always being the executable header (Ehdr).

Name	Location[6]	Usage
Executable header (Edhr)	Start of file	Defines magic number, type of ELF, architecture and options linking to other parts of the ELF file
Section Headers (Shdr)	e_shoff	Defines every section and provides a section view of the file (meant for static-linking purposes)
Program Headers (Phdr)	e_phoff	Provides a segment view (execution view) of the binary (used by OS and dynamic-linker to decide what to load into virtual memory)
Sections	sh_addr	Stores the actual elf sections (.text, .data, ...)

Figure 2: Executable and Linkable Format (ELF) file sections [x0nu11byt3, 2024]

The Section Headers (Shdr) represent the most important component, as they provide access to the complete table. The type and flags associated with each section are especially important and can be obtained from the variables `sh_type` and `sh_flags`.

[6]Section header table file offset (e_shoff) and program header table file offset (e_phoff) can be found in the Executable header. The virtual section address (sh_addr) can be found in the Section header table

`sh_type` is an enumeration value describing the section where the header is currently being read from. `sh_flags` is a `std::bitset`, where specific indices are 1 if the condition is true or 0 if it is false.

The most important flags are further described in the table below:

Bit(s)	Name	Meaning
0	SHF_WRITE	Section is writable at runtime. If it isn't then the section is assumed to be READONLY and only that flag is shown in the objdump.
1	SHF_ALLOC	Section occupies memory during execution. ALLOC flag is shown in the objdump.
2	SHF_EXECINSTR	Section is executable. CODE flag is shown in the object dump.

Figure 3: Executable and Linkable Format (ELF) flags [x0nu11byt3, 2024].

2.2 Frame allocator

The next step is to create a frame allocator as well as a representation of a frame. The allocator needs to be able to both allocate and deallocate frames, which simply hold information on their size and an internal counter which shows the order of the allocated frames.

When allocating memory it is important not to overwrite sections that are already in use. This requires to know where the multiboot2 and the kernel code reside. In TeachOS, both are situated in low memory, which allows to utilize all memory from the end of the kernel section to the end of available space.

Before being able to allocate memory for the first time, a free memory area must be located first. After that the allocation is initialized and the `allocate_frame` member function of the `area_frame_allocator` can be used.

The `allocate_frame` member function follows the following logic to find a free frame:

1. Check if there is a memory area inside `current_area`. If not, then there are no free frames left and the function execution stops.
2. Check if the next free frame is inside the current memory area. If not find the next memory area and write its address into `current_area`.
3. Check if the next free frame is occupied by either the kernel or the multiboot2 code. If that is the case, set the next free frame to the first address after the end of the kernel code.
4. If the condition of step 2 and 3 has not been met, return the next free frame and increase the value of `next_free_frame` by 1. Otherwise, repeat from step 1.

Choosing the next memory area works by iterating through all memory areas and comparing the last address of the current memory area with the address of the next free frame. If the next free frame address is within the current memory area, it will be chosen as the new `current_area`.

2.3 Page tables

The x86_64 architecture uses a 4-level page table and its virtual addresses are structured according to the following diagram:

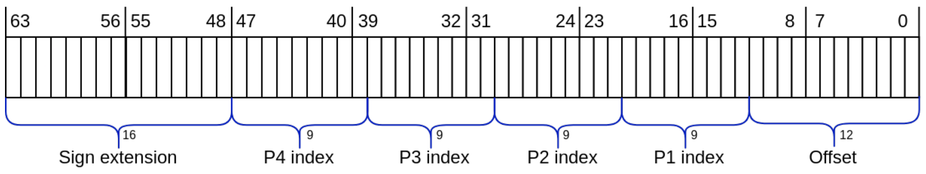

Figure 4: Sketch of virtual address structure [Oppermann, 2024]

Sign Extension: Copies of bit 47 (negative: 1, positive: 0), **16 bits**

Level-4 page index: Address in level-4 page table that points to level-3 page table, **9 bits**

Level-3 page index: Address in level-3 page table that points to level-2 page table, **9 bits**

Level-2 page index: Address in level-2 page table that points to level-1 page table, **9 bits**

Level-1 page index: Address in level-1 page table that points to the physical frame, **9 bits**

Offset Address in physical frame, **12 bits**

All Page tables have $2^9 = 512$ entries, with 8 Bytes per entry. Resulting in $512 * 8\ Byte = 4096\ Byte = 4\ KiB$, which is exactly the size of a page.

The first step is to create an entry into the individual page tables. Those entries could currently be unused, thus it is essential to implement a method that determines whether an entry is unused and another method that marks an entry as unused.

Following table declares all flags a page table entry can have:

Bit(s)	Name	Meaning
0	present	The page is currently in memory
1	writable	Allowed to write to this page
2	user accessible	Only kernel mode can access this page
3	write through caching	Writes go directly to memory
4	disable cache	No cache is used for this page
5	accessed	CPU sets this bit when page is used
6	dirty	CPU sets this bit when page write occurs
7	huge page / null	Must be 0 in level-1 or level-4 page table, 1 GiB page in level-3 page table, 2 MiB page in level-2 page table
8	global	Page isn't flushed from caches on address space switch
9 - 11	available	Freely used by OS
12 - 51	physical address	Page aligned 52bit physical address of frame or next page table
52 - 62	available	Freely used by OS
63	no execute	Forbid executing code on this page

Figure 5: Possible flags of page table entries [Intel Corporation, 2024]

It is necessary to identify an entry's flags programmatically, which can be achieved by creating a method utilizing bit-masking. Additionally, it must be possible to set such flags.

Using these two methods a page can be accessed conditionally if the "present" bit is set, which will return the physical address bits 12 to 51.

Once the entries have been modeled the actual page tables can be created. Each page table contains 512 entries which are all initially set to unused.

2.4 Referencing page tables

Creating and initializing page tables requires defining a virtual address, where the level-4 page table resides.

Level	Pages	Recursive Mapping
0	(page)	-
1	PT	0xFFFF FF80 0000 0000 + 0x4000 0000 * PDPi + 0x20 0000 * PDi + 0x1000 * PTi
2	PD	0xFFFF FFFF C000 0000 + 0x20 0000 * PDPi + 0x1000 * PDi
3	PDP	0xFFFF FFFF FFE0 0000 + 0x1000 * PDPi
4	PML4	0xFFFF FFFF FFFF F000

Figure 6: Long mode (64-bit) page map [wiki, 2024]

This virtual address is then used as the base to calculate the other page tables and their respective addresses. Defining where exactly the page tables reside in memory is not required, instead the subsequent levels are stored wherever the previous table finishes. This behavior makes it possible to precisely control where the tables are using the code structure.

2.5 Mapping page tables

To access page tables and individual entries, recursive mapping can be used. This simply means that the last entry of the level-4 page table contains the address to the level-4 page table itself. This allows to access each page table through a unique virtual address.

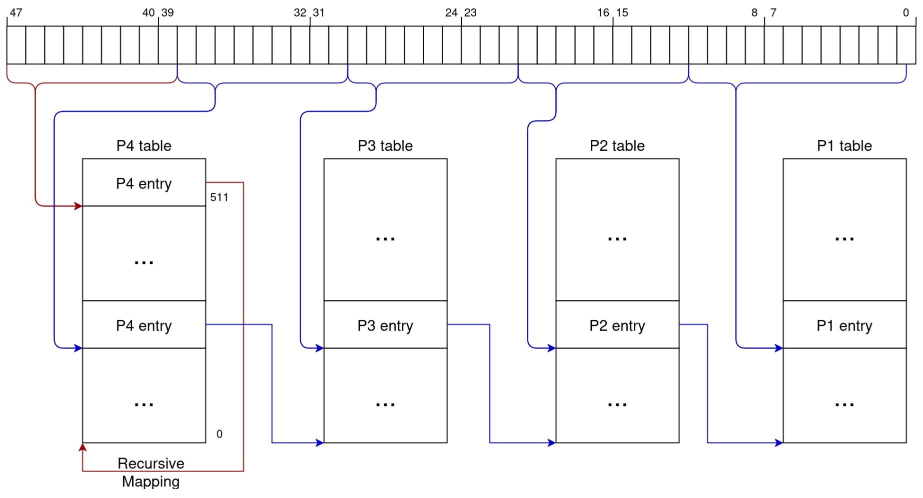

Figure 7: Sketch of the recursive page table structure [Oppermann, 2024]

The $512 - 1 = 511$ mappable entries in the level-4 page table, result in 512 GiB less for the total amount of mappable memory area.

The total memory is still 225.5 TiB, therefore the "lost" level-4 page table entry is negligible. This is the case because one level-4 page table has 511 entries pointing to level-3 page tables, which results in $511 * 512$ total level-3 page table entries. These entries then all point to a level-2 page table, resulting in $511*512*512$ level-2 page table entries, which themselves all point to a level-1 page table. Finally, these level-1 page tables contain 512 entries again, resulting in $511 * 512 * 512 * 512$ level-1 page table entries, which are 4 KiB pages.

2.5.1 Accessing entries

To access level-1 page table entries, the address is left-shifted by 9, and the entry index, ranging from 0 to 511, is left-shifted by 12 and combined with the address. This ensures that writes do not occur in the offset portion of the virtual address, as shown in Figure 4.

This procedure, can then get repeatedly executed on the newly calculated address, up to the level-1 page table, where the index can be used to read from physical memory.

Example calculation for accessing the 511 entry of the level-4 page table:

$0xFFFFFFF'FFFFF000 << 9 = 0xFFFFFFFF'FFE00000$

$511 << 12 = 0x1FF000$

$0xFFFFFFFF'FFE00000 + 0x1FF000 = 0xFFFFFFFF'FFFFF000$

```
auto const table_address = reinterpret_cast<std::size_t>(this);
return ((table_address << 9) | (table_index << 12));
```

Huge pages require additional work, because only level-2 and level-3 page tables can be huge. These are currently not supported.

2.5.2 Mapping virtual pages into physical frames

After that, a way to map virtual pages into a specific physical frame must be created. This method will then use the previously created allocator instance to create an entry in every page table level if it does not already exist and use the aforementioned way to access the entries at the page table indices, to move down to the level-1 page table. Once that is done an entry marked as present can be created and inserted into the level-1 page table, so it can later be read.

A method to unmap previously mapped pages is also required. This can be done by traversing the page table to level-1 and mark the entry as unused. The frame occupied by this unmapped page also must be deallocated to correctly reflect the logical memory state. If the just unmapped page was the only mapped page of a page table, the page table entry must be unmapped as well, continuing with this pattern until page table level-4 is reached.

To prevent the unmapped addresses from being in cache and thus accessed, the Translation Lookaside Buffer must be flushed. On an x86 architecture this can be done using the `invlpg` call.

```
asm volatile("invlpg (%[input])" :
             /* no output from call */ :
             [input] "r"(address) :
             "memory");
```

However, this setup currently does not work as intended because in the assembly code, huge pages are being mapped. This mismatch results in only a partial clearing of the page table and thus the old mapping still being in cache.

To circumvent this issue the entire TLB has to be flushed, which can be achieved by reading from the `CR3` register and writing that value back into the `CR3` register. This makes the system assume the value of the `CR3` register changed, and invalidates the entire cache.

```
uint64_t current_value;
asm volatile("mov %%cr3, %[output]" :
             [output] "=r"(current_value));
asm volatile("mov %[input], %%cr3" :
             /* no output from call */ :
             [input] "r"(current_value) :
             "memory");
```

2.6 Owning the page table

To ensure thread safety, following rules must be followed.

1. There is only ever one active page table at once
2. A page table owns all of its lower level sub-tables

This can be done by creating a method which only creates a page table handle on the first call. All subsequent calls will return the previously created handle.

```
namespace
{
  std::size_t constexpr PAGE_TABLE_LEVEL_4_ADDRESS =
    0xffffffff'fffff000;
}  // namespace

auto active_page_table::create_or_get() -> active_page_table &
{
  static page_table_handle active_handle{
    reinterpret_cast<page_table *>(PAGE_TABLE_LEVEL_4_ADDRESS),
    page_table_handle::LEVEL4
  };
  static active_page_table active_page{active_handle};
  return active_page;
}
```

To strictly enforce nothing can interact with the page table directly or create a new instance of it, the implementation and definition is hidden in the .cpp file. This results in only the page table handle being able to access the page table. The page table handle exposes some of the page table's member functions through own implementations, but is able to restrict, monitor and modify all access.

```
struct page_table;

struct page_table_handle {
    ...

private:
  page_table* table;
  level table_level;
};
```

One example of restricting access to page table member functions through the page table handle is the `next_table` function. The page table handle ensures the current page table is not of level-1, before invoking `next_table` on it.

That the page table owns all of its lower levels is also ensured by the next table method. Which is only accessible over a page table handle and returns a new page table handle, which now points to the page table one level lower.

2.7 Remapping the Kernel

The kernel memory is currently identity mapped in the assembly code. All of it is mapped with full access rights (read, write and execute). This can be observed by comparing the bits set below to their respective flag (Figure 5).

```
or $((1 << 0) | (1 << 1) | (1 << 7)), %eax
```

To instead map the individual kernel sections correctly according to their flags, the mapping must be done through C++ code. However, the previous mapping mechanism can not directly be used for this.

This is due to the memory regions that must be remapped being actively used by the CPU. Therefore, a workaround has to be added that allows to map a different virtual address in memory, which is unused by the CPU, as the level-4 page table. This region then holds the correct mapping and can be switched out with the current mapping once everything is done.

To accomplish this, it is necessary to establish a temporary page that includes an allocator. For this use-case it is enough to only hold one frame on the level-1, level-2 and level-3 page table respectively, because the same page is being mapped temporarily.

Once the temporary frame has been allocated, it should be mapped into the active page table, its internal memory zeroed, and recursive mapping established for itself. Upon completion, the temporary page and its allocated frames must be unmapped.

To enable recursive mapping for inactive page tables, the 511th entry, which previously referenced the active level-4 page table, should be updated to reference the inactive level-4 page table instead. This adjustment allows the previously implemented methods for the active page table to be applied to the inactive page table, as all other functionality remains consistent except for the recursive mapping.

The recursive mapping can then be switched as described, followed by clearing the entire Translation Lookaside Buffer (TLB) to remove any old translations. Once the mapping has been updated, the kernel ELF sections can be correctly remapped to the configured unused virtual address.

After completing these steps, the previous mapping must be restored by temporarily pointing the active level-4 page table to an alternate address and then restoring the original recursive mapping. Finally, the TLB must be cleared again to eliminate any stale translations where the recursive mapping was directed to the inactive page table.

2.7.1 Unmapping workaround

Unmapping page tables would normally be done like shown below:

1. Unmap the level 1-page table entry
2. If the just removed entry was the last one of its table, unmap the level 2-page table entry it was contained in.
3. Repeat this procedure for the level 3 and level 4-page table entry as well

This process, however, causes issues with the recursively mapped structure of the inactive page table. This is the case because, even tough the level 1-page entry can be unmapped the remaining mappings are required for the previously set up recursive mapping process to work correctly. These entries are used when accessing the underlying active page table, because the recursive mapping has been changed so that now resolving the complete recursive mapping results in the address of the active level 4-page table. However, resolving this mapping is not possible anymore, if we unmap the upper level entries as well. Therefore, it would make remapping the active kernel impossible.

That is the reason why unmapping upper layers, is simply skipped and only the level 1-page table entry is unmapped instead.

2.7.2 Mapping kernel elf sections

Before restoring the recursive mapping, it is necessary to perform mapping as described above. Subsequently, all ELF sections originally retrieved from the Multiboot2 header must be iterated through.

To ensure successful mapping of these sections, it is important to align all ELF sections to 4 KiB boundaries, enabling them to be mapped to memory pages. Using the most important flags specified in Figure 3, it must be verified whether a section occupies memory. Sections that do not occupy memory should not be mapped.

The mapping process involves covering the entire range of each ELF section, starting from the physical address of the section and mapping every frame up to the last Byte of the section. During this process, the entry is marked as writable if the section is writable and executable if the section is executable.

Special attention must be given to calculating the range of frames to map. The starting frame is determined by the physical address of the section, while the end frame corresponds to the physical address plus the section size, minus one. This subtraction is important to avoid mapping the first Byte of the next section instead of the last Byte of the current one.

Additionally, the end frame must always be incremented by one. Failing to do so would omit the configuration of the final frame in each section, as the end frame should always point to one past the last frame in the range.

```
auto const start_frame =
  allocator::physical_frame::containing_address(
    section.physical_address);
auto end_frame =
  ++(allocator::physical_frame::containing_address(
    section.physical_address + section.section_size - 1));
```

2.8 Kernel heap

The previously created paging and kernel mapping serves as the foundation for heap management. This setup enables the use of data structures like `std::vector`, which require an allocator. Ideally, the allocator should prioritize speed, reliability, and especially cache locality to minimize performance losses caused by inefficient memory ordering. Misaligned memory can lead to frequent cache misses, forcing additional memory reads even when data could have been retrieved from the previous cache line.

To use the allocator effectively, the heap pages must first be correctly mapped. This involves using the active kernel page table, after it has been remapped, to map all pages from the virtual start address to the virtual end address minus one of the heap.

2.8.1 Bump heap allocator

Once that has been done, it is possible to create heap allocators. The most simple one is a bump allocator, which simply allocates linearly and leaks all memory that is deallocated. This can be done relatively easily by getting the heap start and end address and then incrementing a number starting from the heap start address that is then returned as a `void *`.

```
auto const alloc_start = next;
auto const alloc_end = next + size;
arch::exception_handling::assert(alloc_end <= heap_end,
                                 "[Heap Allocator] Out of memory");
next = alloc_end;
return reinterpret_cast<void *>(alloc_start);
```

There are a few problems with this simple example:

1. `next + size` can potentially overflow the underlying type causing the code to return a memory address that has already been used.

2. The allocate method can be called by multiple threads and cause missed updates when returning the `alloc_start` address, causing the same issue.

To fix the first issue, the addition has to be clamped to the max value if it would overflow.

```
template<typename T>
auto saturating_add(T x, T y) -> T
  requires std::is_unsigned_v<T>
{
  if (x > std::numeric_limits<T>::max() - y)
  {
    return std::numeric_limits<T>::max();
  }
  T result = x + y;
  return result;
}
```

The second one can be fixed by making the variable `next` atomic. To achieve this the `atomic` header can be included to use the `std::atomic_uint64_t` type for it.

After this change the value from the variable `next` has to be accessed with the `load` method. After calculating the new address, the underlying value should not have been changed by another thread already, this can be checked with `compare_exchange_weak`.

If it did not, the value to the newly calculated address is the new start address and the previous start address can be returned as a pointer. If it did, we have to repeat the aforementioned steps until no other threads have overtaken this thread, while trying to allocate memory.

```
for (;;)
{
  auto alloc_start = next.load(std::memory_order::relaxed);
  auto const alloc_end = saturating_add(alloc_start, size);
  arch::exception_handling::assert(alloc_end <= heap_end,
    "[Heap Allocator] Out of memory");
  auto const updated = next.compare_exchange_weak(
                          alloc_start, alloc_end,
                          std::memory_order::relaxed);
  if (updated)
  {
    return reinterpret_cast<void *>(alloc_start);
  }
}
```

The `std::memory_order::relaxed` ordering is used, because we simply require the variable to be atomic, meaning we do not miss updates of other threads. But the variable is not used for synchronization between threads. Therefore, no restriction on what happens before we read the value is necessary.

Additionally, `compare_exchange_weak` is used, because the only drawback compared to the strong version, is that spurious failure is possible. Meaning the value was actually still the same but the checked still failed, the strong version then internally retries the check. But because the whole code is in an infinite for loop this case is handled already.

2.8.2 Linked-List heap allocator

To improve upon the simple bump allocator and to be able to deallocate memory, an implementation utilizing a linked list has been added.

The most difficult part of memory management is tracking and reusing freed memory blocks to minimize fragmentation. This advanced method uses a singly linked list of free memory blocks, each pointing to the next block. The linked list header also contains the size of the block to be able to efficiently allocate memory into the free memory blocks.

Initially, the singly linked list contains a single entry representing the entire heap with no additional blocks. The memory is being allocated using the First fit search algorithm (Section 1.6.5). This algorithm allocates the desired data onto the first free block that it fits in. To avoid a big mismatch of the allocated data size and the heap space being used, the free memory block is split into two, one with a fitting size and one with the remaining space. The memory block that is no longer free is then removed, and the previous block updated to point to the second created memory block. The memory address of the removed memory block is then returned for the caller to be able to construct an object of fitting size at that location

```
auto const start_address =
  reinterpret_cast<std::size_t>(current_block);
auto const end_address =
  reinterpret_cast<std::size_t>(current_block) + size;
auto const new_block =
  new (reinterpret_cast<void *>(end_address))
  memory_block(current_block->size - size, current_block->next);

if (previous_block == nullptr)
{
  first = new_block;
}
else
{
  previous_block->next = new_block;
}
return reinterpret_cast<void *>(start_address);
```

To deallocate a block, a free memory block must be inserted into the singly linked list at the correct position. If the deallocated block is adjacent to one or two existing free blocks, the blocks must be merged into a single larger free block to reduce fragmentation.

Both the bump allocator and the linked list allocator are thread safe. Although where the bump allocator simply uses an `std::atomic_uint64_t` to safely increase the allocated block count, the linked list allocator must utilize a mutex on both allocation and deallocation.

Since it is not possible to use the `mutex` STL library for development, a simple mutex class had to be created. The custom mutex implementation follows the interface of the STL library, with the exception that `lock` does not yield and instead locks the process in an infinite loop until the resource is available.

```
auto mutex::lock() -> void
{
  while (!try_lock())
  {
    // Nothing to do
  }
}

auto mutex::try_lock() -> bool {
  return !locked.exchange(true, std::memory_order_acquire);
}

auto mutex::unlock() -> void {
  locked.store(false, std::memory_order_release);
}
```

Part IV

Conventions

3 Assembly syntax

The assembly files are not written in plain assembly, but instead in GNU assembly (GAS). GAS has some subtle differences to regular assembly, of which the most notable are documented here.

- Instructions like `mov` parse its two arguments the other way around. The value of the first argument will be moved into the second.
- Writing plain values into registers can not be done by simply writing `mov 1000, ebx`, instead it must be done like this: `mov $(1000), ebx`
- Variables when copied from or when copied into, have to be accessed with `$variable`, which accesses their value
- Writing values into the address of variables with a certain offset, can not be done by writing `mov %eax, [$variable + 1]`, but instead requires to access the address of the `variable` like this `mov %eax, (variable + 1)`

4 C++ syntax

4.1 Function arguments

All variables containing data smaller than 64 bits (8 Bytes) will be passed by value, others will be passed by reference. Vectors, arrays and other containers including more complex objects will also be passed by reference. This is done because on a 64-bit architecture every reference or pointer is also 64-bit, so it doesn't matter if we copy the value or return a reference to it, we might even require more memory if the type is actually smaller than 64 bits.

The exception to this rule is if the object should be changed by the function. In that case it will always be passed by reference instead, because passing by value would result in the updates only applying inside the function scope.

4.2 Bitset flag usage

For all components that have to read specific bits from a `uint64_t`, we use a `std::bitset` to handle that data. The flags themselves are inside an enum and are a `uint64_t` value with only the specified bit set to 1.

That `enum` is put into the `struct` and is itself unscoped, because this allows direct conversion of the `enum` value to the actual underlying value, without leaking the constants to the namespace.

```
struct entry {
  enum bitset : uint64_t {
    PRESENT = 1UL << 0UL,
    WRITABLE = 1UL << 1UL,
    USER_ACCESIBLE = 1UL << 2UL,
    WRITE_THROUGH_CACHING = 1UL << 3UL,
    DISABLED_CACHING = 1UL << 4UL,
    ACCESSED = 1UL << 5UL,
    DIRTY = 1UL << 6UL,
    HUGE_PAGE = 1UL << 7UL,
    GLOBAL = 1UL << 8UL,
    EXECUTING_CODE_FORBIDDEN = 1UL << 63UL,
  };
}
```

The individual bit shifts without additional symbols like `U (unsigned value)` or `L (long value)`, will result in always using `int32_t`, which in turn results in unintended behavior if the resulting value would overflow.

These defined `enum` values should then be used to read the state from the underlying `std::bitset`. This is done using a helper method that takes a `std::bitset` "a" by copy and then applies the `& (bitwise AND)` operator on it and the underlying `std::bitset` "b". If the result is the same as "a", then "b" has the flag "a" set to 1.

```
auto contains_flags(std::bitset<64U> other) const -> bool
{
  return (flags & other) == other;
}
```

Furthermore, because we use a `std::bitset` as the argument, which has an implicit constructor from `uint64_t`, we can directly pass the previously created flags. Additionally, this even makes it possible to check multiple flags at once by using the `| (bitwise OR)` operator on 2 flags.

```
contains_flags(entry::PRESENT | entry::WRITABLE);
```

4.3 Indexable enum value

For all components that require an `enum` that can be utilized in a range-based `for` loop, we use an unscoped `enum` in the struct, that implements the pre-increment `friend operator` for that `enum`.

```
for (auto level = page_table_handle::LEVEL4;
     level != page_table_handle::LEVEL1;
     --level)
{ ... }
```

We use an unscoped `enum`, because it allows implicit conversion to the underlying type, while not leaking the definitions into namespace. The pre-increment operator override could both be done as an inline function in the header, or as a friend function. We use the latter, because it also allows using the operator in the struct that defined it and allows to write the definition in the implementation instead of the header file, which can decrease the amount of required includes in the header.

```
struct page_table_handle
{
  enum level : uint8_t { ... };

  friend auto operator--(level & value) -> level &;
}
```

4.4 Containers with std::optional values

If a container has an `std::optional` as it's value we need to make sure to correctly initialize it. Directly initializing the container with `std::optional`, will not work because it does copy the underlying value of the `std::optional` if it exists, but it does not set the engaged flag. Meaning even if it did receive an `std::optional` that was valid the constructed element itself will not be valid and return `false` from the `std::optional::has_value` method.

To prevent this we have to check individually if the value we attempted to insert is actually valid and if it is, meaning `std::optional::has_value` is `true` we can directly `std::optional::emplace` the underlying `std::optional::value` into the `std::optional` of our container.

4.5 Inline assembly

For certain instructions it is not possible to write C++ code. It is therefore sometimes necessary to directly use assembly language. This can be done through the use of inline assembly. [Assembly, 2024]

The syntax of inline assembly consists of the following components:

```
asm (
assembly template
: output operands
: input operands
: clobbered registers list
);
```

4.5.1 Output operands

The output operand is an optional argument that tells the compiler how it should handle variables that are used to store output from the assembly template.

```
int a, b;
asm("movq $10, %0; movq $20, %[output]" :
    "=r" (a),
    [output] "=r" (b)
);
```

The important part is that we can either use labels that we write values into, or we can use the enumeration, where the first declared variable is 0 and then every following variable is simply incremented by one. The "r" means we are writing a register and the "=", indicates we do not care about the initial value of the variable, which allows some optimizations.

4.5.2 Input operands

The input operand is an optional argument that tells the compiler how it should handle variables that are used to store input into the assembly template.

```
int a, b;
asm("movq %0, %eax; movq %[input], %ebx" :
    /* No output */ :
    "r" (a), [input] "r" (b) :
    "eax, ebx"
);
```

The syntax is similar to the output operand, but the order of the `movq` statement is inverted and we of course now do care about the initial value of the variables, because we copy them, therefore we do not use the "=" sign.

4.5.3 Clobbered registers list

The clobbered registers list is an optional comma separated list of register names as string literals, that have been changed by the assembly template.

Especially important is to understand that the C++ compiler knows nothing about assembly. The inline assembly we write is completely opaque, and if it does not have any output it might even optimize the statement away, because it thinks it is a no-op.

To prevent this the `volatile` keyword after the `asm` statement can be used, because it ensures the compiler knows this statement has important side effects. Additionally, it is also useful to append `: memory` in the clobbered registers list to prevent the statement from being moved by the compiler.

Part V

Results

As part of this research paper, we were able to implement all the necessary groundwork of the kernel. These results are described in the following sections.

5 Physical memory allocator

To be able to use or store any data, a physical memory manager must be in place. It manages the allocation and deallocation of physical memory. We implemented a very simplistic version of this allocator, which is able to allocate memory in chunks of 4 KiB. It is not able to deallocate any memory, because that feature is not needed for the purposes of the kernel in its current state.

However, it is a good starting point, when continuing development on this kernel, to create additional implementations utilizing different strategies and data structures.

6 Virtual memory management

Building upon the physical memory allocator is the virtual memory management. It is an abstraction of physical memory to enable processes to use more memory than physically available. It also separates a programs logical address space from the physical address space. We implemented virtual memory management using paging with a multilevel page table. This structure is a solid base for the operating system. However, similar to the physical memory allocator, we have not implemented any paging strategies.

Creating different fetching, cleaning and page replacement policy implementations as described in Section 1.5 is a solid and reasonable way of continuing the kernel development.

7 Heap memory allocator

A heap memory allocator manages dynamic memory allocation in a program's heap, where memory is allocated and freed during runtime. We have created an interface for a heap allocator and two implementations. One of them is very simple and is unable to deallocated heap memory, which results in a permanent memory leak, while the other utilizes a linked-list data structure and the First fit algorithm (Section 1.6.5, First fit) to manage free memory blocks.

Again, an excellent place to begin extending this functionality is implementing different algorithms as described in Section 1.6.5.

Part VI

Future work

The next tasks of this project are fixing the unmapping workaround (Section 2.7.1), as well as implementing different heap allocation strategies. Integration of those strategies into standard containers and enabling the use of the C++ STL.

Planning further into the future, developing tools to visualize and monitor the current state of memory, including allocations, deallocations, and fragmentation to eventually create a framework for simulating user-space programs, allowing the monitoring of memory management strategies and container implementations under realistic workloads.

References

[Abraham Silberschatz and Gagne, 2018] Abraham Silberschatz, P. B. G. and Gagne, G. (2018). *Operating System Concepts*. Wiley. https://archive.org/details/operatingsystemconcepts10th.

[Assembly, 2024] Assembly, O. I. (Accessed: 05–10–2024). Operating system development inline assembly. https://wiki.osdev.org/Inline_Assembly.

[Ford and Boleyn, 1995] Ford, B. and Boleyn, E. S. (1995). *The Multiboot2 Specification*. Free Software Foundation. https://www.gnu.org/software/grub/manual/multiboot2/multiboot.pdf.

[Intel Corporation, 2024] Intel Corporation (31–10–2024). *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[Oppermann, 2024] Oppermann, P. (Accessed: 24–9–2024). Writing an os in rust. https://os.phil-opp.com/edition-1/.

[rhboot, 2024] rhboot (Accessed: 19–10–2024). Grub2 repository. https://github.com/rhboot/grub2.

[wiki, 2024] wiki, O. (Accessed: 18–9–2024). Operating system development wiki. https://wiki.osdev.org/Expanded_Main_Page/.

[x0nu11byt3, 2024] x0nu11byt3 (Accessed: 7–10–2024). Elf format cheatsheet. https://gist.github.com/x0nu11byt3/bcb35c3de461e5fb66173071a2379779.