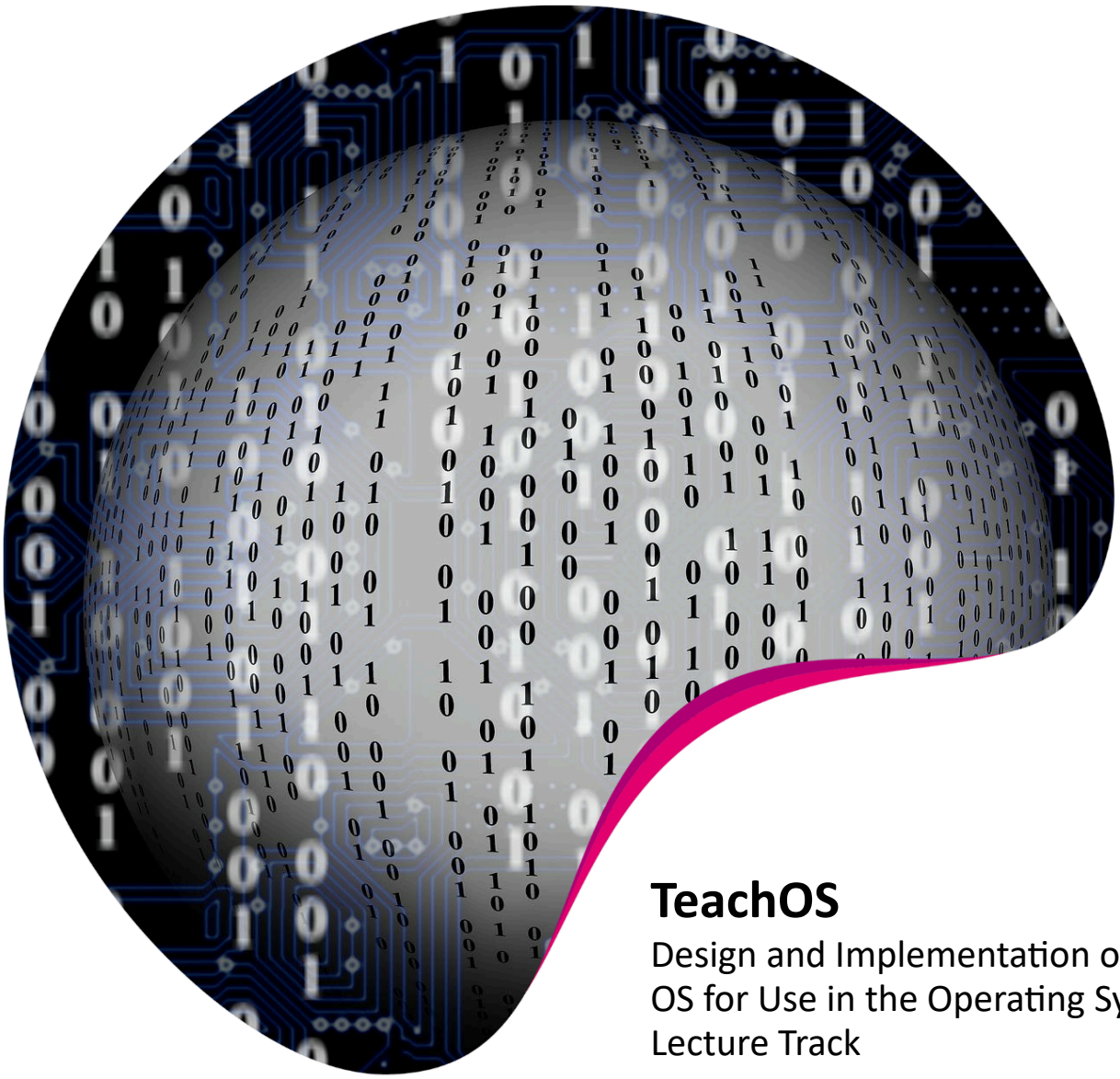
OST
Ostschweizer
Fachhochschule



TeachOS

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

Print time:

11.06.2025

Team / Authors:

Matteo Gmür and Fabian Imhof

Advisors:

Felix Morgner

Copy editor:

Thomas Letsch

Abstract

TeachOS is an operating System for the x86_64 CPU architecture. It is designed to be an interactive learning tool used in the «Operating Systems 1» and «Operating Systems 2» lecture track at OST.

As part of a previous thesis, a foundation in form of memory management, including physical and heap memory was created. Topics covered included 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.

This paper examines the core concept of context-switching in operating systems for the x86 processor architecture. To implement the context-switch mechanisms, multiple operating system concepts had to be researched and implemented. These concepts include Privilege Levels, the Global Descriptor Table, Interrupts and the Interrupt Descriptor Table and Procedure Calls.

To give the newly created access to user mode a purpose, the existing heap implementation has been expanded upon, by utilizing the `new` and `delete` operators in various data-structures. Additionally a separate user mode heap has been created, which is separated from the kernel mode heap via paging, and is dynamically extended when running out of memory.

This foundation can be extended in the future, by separating kernel-code and user-code binaries. Doing this allows for better separation of the different privilege levels inside the paging mechanism, which in turn better resembles real world operating systems, and thus better fulfills the primary goal of providing an operating system designed for education.

OST
Ostschweizer
Fachhochschule

Table of Contents

Part 1
Goals

1.1 Scope

This project focuses on enhancing the TeachOS operating system with improved memory management, task creation, and user- and kernel-space interaction. The main areas of development include:

- Context switching
  - Implement interrupt handling
  - Enable paging in user space
  - Create user-mode heap
  - Ensure correct permissions for the different modes
- Custom Memory Management and Data Structures
  - Override `new` and `delete` to use a heap allocator based on current protection level
  - Build reusable infrastructure to replace parts of the C++ standard library (e.g. `std::vector`, `std::unique_pointer`)

1.2 Memory Management

The memory management should be improved, by making heap allocation as simple as possible. This is required for efficient and viable memory management inside all privilege levels. These improvements have the highest priority and should be implemented first.

The current memory management implementation contains various deficiencies that require some tweaking to be more optimal and developer-friendly:

- Connect memory management infrastructure to `new` and `delete` calls
- Create custom implementations of C++ Standard Library constructs like `std::vector` which use the custom memory management infrastructure
- Improve `linked_list_allocator` to allow for `delete` calls without passing a size attribute (Storing the size in a header of the allocated block instead)

1.3 Context Switching

The primary goal of this bachelor thesis is implementing the ability to switch between privilege levels. Meaning it should be possible to execute code in user mode and switch to kernel mode via `syscall` to handle more privileged instructions.

This goal essentially consists of the following tasks:

- Creating a global descriptor table with kernel and user segment descriptors
- Creating an interrupt descriptor table to avoid system crashes due to exceptions
- Mapping kernel code and user code onto different pages
- Updating the (permission) flags of all page tables to correctly represent their purpose
- Switching into user-mode at the end of kernel initialization so that the “default” operating mode is in privilege level 3
- Setting up the system in a way, that allows `syscall`s inside user mode, to allow switching the current privilege level to let the kernel handle calls requiring elevated privileges

1.4 Out of scope

Currently TeachOS is turning out to be an operating system kernel running on a single processing core. Partly because of the limited resources available during the bachelor thesis. For this reason a multithreaded implementation of any functionality is out of scope. This results in `syscall`s invoked in user mode being blocking until the kernel is done handling it.

The creation of code solely for educational purposes will also be disregarded in this thesis. One reason is that the kernel is not yet in a state, in which it could immediately be used in a classroom. All the in-scope work is necessary for these features to be effective. This will be explained further in Section 5, “Future work”.

Another interesting and worthwhile feature is implementing appropriate keyboard input handling. However, a complete interrupt handling would have to be implemented first, which is simply not possible due to the time constraint of this thesis. For this reason, keyboard interrupt handling is also mentioned as a possible feature to implement next in Section 5, “Future work”.

Part 2
Research

2.1 User Mode Heap

There are three different ways to implement a separate heap for User Mode. This is done to ensure that user applications can not interfere with the kernels' memory management.

All strategies share that they require an additional heap area to be mapped, so instead of only having a 100 KiB Kernel Heap, they also require a separately mapped Heap for User Mode.

Strategy	Advantages / Disadvantages
Changing allocation calls depending on `CS` register	**Advantages:** • Allows to keep the current operator overloads for Kernel and User Mode • Check the value of the `CS` register, when the `new` or `delete` operator is called • Depending on the value, either allocate memory in User Mode Heap or Kernel Mode Heap • Implementation is relatively simple, because it allows to keep the current infrastructure and requires to simply add another member variable to the `global_heap_allocator` **Disadvantages:** • Kernel and User Mode Heap Allocation are not cleanly separated and can not be separated later without a rewrite
Use template argument for allocator with separate Kernel and User Mode Allocator	**Advantages:** • Uses an additional template argument to pass a custom allocator, depending on the mode for Standard Template Library (STL) implementations • Kernel can use `kmalloc` / `kfree` and User Mode can use `malloc` / `free` which forward to `SYSCALL` • User Mode can be the default Allocator used **Disadvantages:** • Requires more work, because separate allocators need to be written that forward calls to `SYSCALL`
Place User Mode operator overload in separate linker section, loaded in Ring 3	**Advantages:** • Cleanest way, allows to simply implement another set of `new` and `delete` operator overloads • Use separate page tables to only load ELF section in User Mode where the User Mode implementation is contained and not the Kernel Mode implementation **Disadvantages:** • Hard to explain, Linker does Magic in the background. Suboptimal for teaching functionality of the operating system • Hard to put the implementation into the correct Linker section • Would require splitting the binary, because the `new` and `delete` can not be redefined

Table. 1: Comparison of User Mode Heap Implementations

2.1.1 Decision

With these three options available, a decision must be made as to which of them TeachOS should utilize.
Option 3 currently can not be implemented, because it would require splitting User and Kernel Code into different binaries.
Option 2 is a viable option, but it only allows using STL data structures in user mode. However, it does not enable using `new` or `delete` calls directly to instantiate an arbitrary data structure. Therefore, the best option for this operating system is Option 1 combined with Option 2, which has been chosen.

This decision is motivated by the following factors:
- **Extensibility**: The decision tree inside the `new` / `delete` operators have been modified to additionally use Option 2, because the actual allocation calls `kmalloc` / `malloc` and deallocation calls `kfree` / `free` have been separated into a user and kernel method already. Which then individually call their respective Allocator implementation.
- **Learning value**: The code is relatively simple and therefore makes understanding, what exactly happens clearer and more useful as a learning tool.

2.2 Privilege levels

Privilege levels are a protection mechanism. There are four protection levels (0 to 3) with level 0 being the most privileged and decreasing with every higher level. Privilege levels are used to prevent a program operating at a lesser privilege level from accessing a segment with a greater privilege level. Instead, this attempted access will generate a General Protection Fault (#GP).

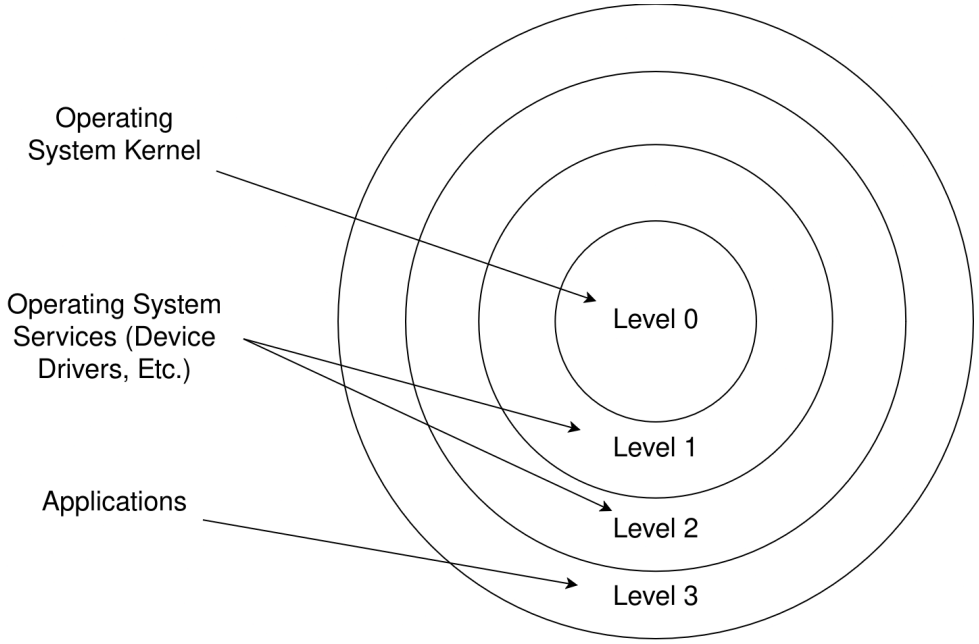


Image. 1: Operating system protection ring (Figure 6-3. Protection Rings [1])

There are multiple recognized types of privilege levels, that check access between Code and Data Segments.

Name	Description
Current Privilege Level (CPL)	Stored in bits 0-1 of the CS and SS registers. Equivalent to the privilege level of the Code Segment the current instructions are being fetched from, for nonconforming Code Segments. Changed when control is being transferred to a Code Segment with a different privilege level, besides for conforming Code Segments, here the CPL is not changed even if control is transferred.
Descriptor Privilege Level (DPL)	Describes privilege level of Segment Descriptor (Section 2.4.1) or Gate Descriptor (Section 2.5.1). Interpretation depends on the type of segment or gate: • Data Segment / TSS / Call Gate: Indicates the numerically highest CPL that is allowed to access the segment. • Nonconforming Code Segment: Exact CPL required to access the segment. • Conforming Code Segment (or Nonconforming through Call Gate): Indicates the numerically lowest CPL that is allowed to access the segment.
Requested Privilege Level (RPL)	Overrides the value of the CPL. Uses either the CPL or RPL value for actual privilege level checks, depending on which value is numerically higher.

Table. 2: Types of Privilege Levels (Vol. 3A, Chapter 6.5 Privilege Levels [1])

2.3 Interrupts

Interrupts are signals generated by devices, such as keyboards or hard drives, or code, that instruct the CPU to halt its current operations and address a different task. For instance, when a key is pressed on a keyboard, the keyboard controller sends an interrupt to the CPU. This allows the operating system to promptly display the corresponding character on the screen, regardless of the CPU's previous activity, after which the CPU can resume its prior task.

When receiving a specific interrupt, the CPU consults a designated table provided by the operating system to locate the relevant entry for that interrupt. In the x86_64 architecture, this table is referred to as the Interrupt Descriptor Table (IDT). Once the CPU identifies the appropriate entry for the interrupt, it executes the code associated with that entry. This code is called "interrupt service routine" (ISR) or "interrupt handler".

There are usually three different types of interrupts: Exceptions, Interrupt requests / Hardware interrupts and Software interrupts. (Article "Interrupts" [2])

2.4 Global Descriptor Table

The Global Descriptor Table (GDT) is a fundamental component of memory management in the x86 architecture. In 32-bit mode it is used as a data structure by the CPU to define and separate memory segments in a system.

In IA-32e 64-bit mode, segmentation using the GDT has been replaced with paging, which is instead used to separate memory into protected areas. This causes the CPU to simply treat all Base field values as if they were 0, regardless of the actual value in the Segment Descriptor. Additionally, limit checks for the registers are disabled and the values of the Limit field are ignored as well. (Vol. 3A, Chapter 3.2.4 Segmentation in IA-32e Mode [1])

Therefore, the setup of the GDT can be minimal in IA-32e 64-bit mode, as it is only used for Context Switching. This minimal setup requires an empty "Null Descriptor" at the start, followed by a Code Segment Descriptor and a Data Segment Descriptor for Kernel Mode, a Code Segment Descriptor and Data Segment Descriptor for User Mode and lastly a Task State Segment Descriptor.

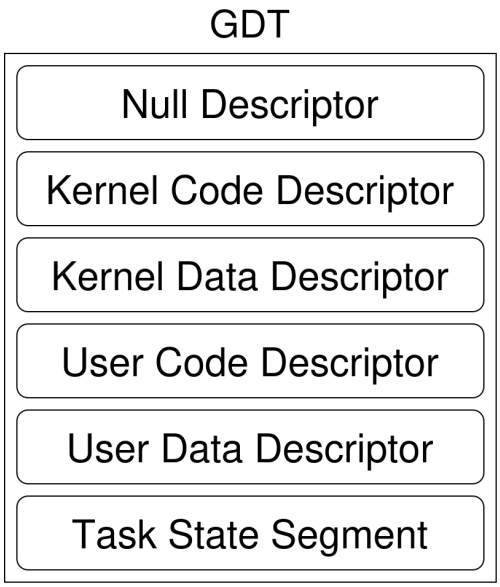


Image. 2: GDT segments in IA-32e 64-bit mode

During the setup of the GDT it is important to ensure that no interrupts are occurring. The code initializing the GDT must not get interrupted, because the program could still attempt to access the segment registers of the old GDT structure. [3]

To prevent this, all interrupts have to be temporarily disabled. This can be achieved by clearing the interrupt flag using the `cli` instruction. Once the IDT has been set up interrupts can again be enabled, simply by setting the interrupt flag again through the `sti` instruction. This will additionally cause any hardware interrupts that were ignored while the flag was cleared, to be received again as soon as the flag has been set.

2.4.1 Segment Descriptor

The memory segments inside the GDT are called Segment Descriptors and contain the following data for the 64-bit version.

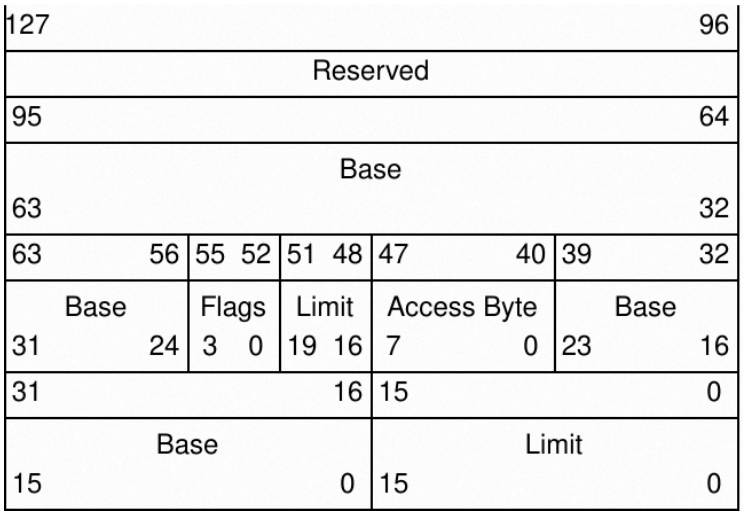


Image. 3: 64-bit Segment Descriptor [1] (Figure 3-8. Segment Descriptor)

The Base and Limit fields are divided across multiple separate fields. While this might initially seem unusual, it originated for historical reasons to maintain backwards compatibility with the 16-bit processor format of the 80286 and the 32-bit processor format of the 80368 processor. The 16-bit processor does not require the additional Flags field, because it is primarily used to signal 32-bit instead of 16-bit registers. Furthermore, it also does not require the 4 additional bits for the Limit field or the additional 8 bit for the Base field. Instead, these 16 bit have to be 0 and reserved to keep compatibility between both processor formats.

63 48	47 40	39 32
Reserved	Access Byte 7 0	Base 23 16
31 16	15 0	
Base 15 0	Limit 15 0	

63 56	55 52	51 48	47 40	39 32
Base 31 24	Flags 3 0	Limit 19 16	Access Byte 7 0	Base 23 16
31 16			15 0	
Base 15 0			Limit 15 0	

Table. 3: 16-bit Segment Descriptor (Figure 6-1. Protection Fields of Segment Descriptors [4]) and 32-bit Segment Descriptor (Figure 6-3. Segment Descriptors [5])

Bits	Name	Description
32	Reserved	Required because the 64-bit version of the Segment Descriptor requires a 64-bit base address instead of only a 32-bit address. This means our new entry would be 96 bit (64 bit from 32-bit Segment Descriptor + 32 bit for remaining half of the base address). However, to be compatible with the design of the 32-bit Segment Descriptor, two entries in little-endian format are now needed in the Global Descriptor Table, because those two entries would add up to 128-bit, the remaining space has to be filled with reserved data.
64	Base	Defines the starting memory address of the segment within the 4-GiB linear address space. Is calculated by combining the three base address fields to form a single 64-bit value.
4	Flags	Defines 4 individual bits that change the behavior of the Segment Descriptor.
20	Limit	Defines the size of the segment. Is calculated by combining the two limit fields to form a single 20-bit value. Behavior depends on the Granularity flag in the Flags field. • 0: Segment size ranges from 1 Byte to 1 MiB, in byte increments • 1: Segment size ranges from 4 KiB to 4 GiB in 4 KiB increments
8	Access Byte	Defines 8 individual bits that change the type of the Segment Descriptor.

Table. 4: Segment Descriptor Components (Vol. 3A, Chapter 3.4.5 Segment Descriptors [1])

2.4.1.1 Flags

IA-32e mode can run in 64-bit mode or 32-bit compatibility mode. However, because Long Mode is enabled, IA-32e in 64-bit mode is used and therefore bit 2 is always enabled for Code Segments.

Flags: Data-Segment Descriptor

Granularity	Big	0	Available

Flags: Code-Segment Descriptor

Granularity	Default	Length	Available

Flags: System-Segment Descriptor

Granularity	0	0	0

Table. 5: Differences in Flags on different Segment Descriptor Types (Vol. 3A, Chapter 3.4.5 Segment Descriptors [1])

Bit	Name	Description
1	Available	Segment is available for use by the system software
2	Length	Defines the size of the Code Segment descriptor. Should be 0 for non Code Segment Descriptors. If this bit is set, the third bit should always be 0. Only defined for IA-32e mode, for other modes this bit should be 0. • 0: Defines a 32-bit Compatibility Code Segment • 1: Defines a 64-bit Code Segment

3	Default / Big	Behavior depends on the actual type of the segment descriptor.
		• Code Segment: Flag indicates the default length for effective addresses and operands
		  ▸ 0: 16-bit addresses and 16-bit or 8-bit operands
		  ▸ 1: 32-bit addresses and 32-bit or 8-bit operands
		• Stack Segment: Specifies the size of the Stack Pointer (SP) used for implicit stack operations
		  ▸ 0: 16-bit SP stored in 16-bit Extended Stack Pointer (ESP) register (value 0)
		  ▸ 1: 32-bit SP stored in 32-bit ESP register
		• Expand-Down Data Segment: Specifies the upper bound of the segment
		  ▸ 0: 64 KiB
		  ▸ 1: 4 GiB

4	Granularity	Signifies the resolution of the Limit value. • 0: Scaled by 1 Byte • 1: Scaled by 4 KiB blocks

Table. 6: Segment Descriptor Flags (Little-Endian) (Vol. 3A, Chapter 3.4.5 Segment Descriptors [1])

2.4.1.2 Access Byte

Bit	Name	Description
1-4	Type Field	Further specifies the underlying type of the segment, depends on the Descriptor Type.
5	Descriptor Type	Defines the underlying type of the segment. • 0: System Segment • 1: Code or Data Segment
6-7	Descriptor Privilege Level	Specifies the privilege level required to access this segment. A higher value means a less privileged access level is required. If only two access levels are required it is recommended to use the highest (0) and the lowest access level (3) respectively. • 0: Level 0 (Kernel) • 1: Level 1 (Used for System Services like Device Drivers) • 2: Level 2 • 3: Level 3 (User Applications)
8	Segment-present	Indicates whether the segment is present in memory or not. Will cause a Segment Not Present (#NP) exception, when this Segment Descriptor is loaded while the value is 0. Allows for more fine-grained control of segments that are actually loaded into physical memory at a give time. (Vol. 3A, Chapter Interrupt 11—Segment Not Present (#NP) [1]) • 0: Not present in memory • 1: Present in memory

Table. 7: Segment Descriptor Access Byte (Little-Endian) (Vol. 3A, Chapter 3.4.5 Segment Descriptors [1])

2.4.1.3 Type Field

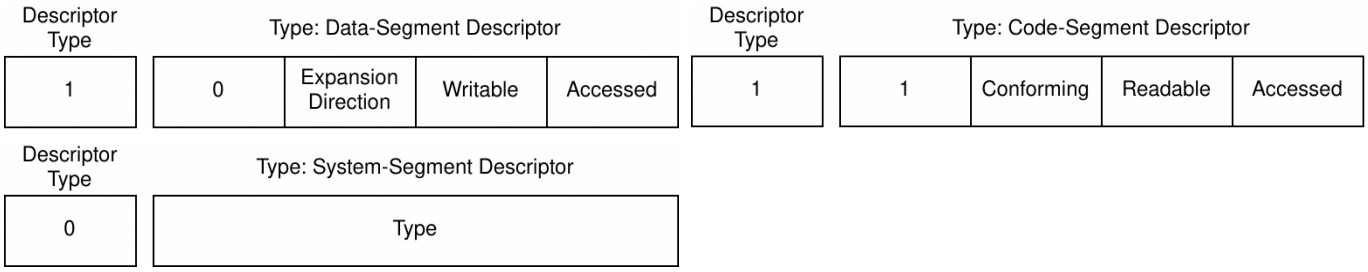


Table. 8: Differences in Type Field on different Segment Descriptor Types (Figure 6-1. Descriptor Fields Used for Protection [1])

2.4.1.3.1 Type Field for Data or Code Segments Descriptors

Bit	Description
1	Behavior is the same for both types Code and Data Segment Descriptors. Whether the segment has been accessed since the last time the operating system has cleared the flag or not. • 0: Not accessed • 1: Accessed
2	Behavior depends on the specific type of the Segment Descriptor as indicated by bit 0. • Data Segment: Determines if the segment is writable or only readable ‣ 0: Read-Only ‣ 1: Read-Write • Code Segment: Indicates if the segment can be read or only executed ‣ 0: Execute-Only ‣ 1: Execute-Read
3	Behavior depends on the actual type of the Segment Descriptor possible in bit 0. • Data Segment: Whether the segment expands downwards or upwards ‣ 0: Expand-Up ‣ 1: Expand-Down • Code Segment: Whether the code is allowed to be executed by different access levels (higher or lower) ‣ 0: Nonconforming (Access by different access level causes a General Protection Fault (#GP) exception, Data Segments are always nonconforming, but allow access from higher levels) ‣ 1: Conforming
4	Defines the actual type of the segment. • 0: Data Segment • 1: Code Segment

Table. 9: Segment Descriptor Type Field (Little-Endian) (Vol. 3A, Chapter 3.4.5 Segment Descriptors [1])

2.4.1.3.2 Type Field for System Segment Descriptors

The Type field for System Segment Descriptors is a special case. This is because the individual bits are not standalone flags. Rather, specific combinations of bits are used to identify the exact type of system segment. Since IA-32e 64-bit mode is in use, only the 64-bit versions of these types are applicable.

All other possible values (0-15) not mentioned below, are reserved for backwards compatibility with 32-bit addresses.

Decimal	Bits	IA-32e Mode
2	0 | 0 | 1 | 0	Local Descriptor Table (LDT)
9	1 | 0 | 0 | 1	64-bit Task State Segment (TSS) Available
11	1 | 0 | 1 | 1	64-bit Task State Segment (TSS) Busy
12	1 | 1 | 0 | 0	64-bit Call Gate
14	1 | 1 | 1 | 0	64-bit Interrupt Gate
15	1 | 1 | 1 | 1	64-bit Trap Gate

Table. 10: System Segment Descriptor Type Field (Table 3-2. System-Segment and Gate-Descriptor Types [1])

2.4.1.4 Segment Types

The Segment Descriptors point to actual Segments, which have different uses depending on their specific type.

- Code Segment: Stores executable instructions
- Data Segment: Stores variables and data
  - Expand-Down Data Segment: Arbitrary Data Segment that expands downwards
  - Stack Segment: Special Data Segment that is pointed to by the Stack-Segment (SS) register. Defines the block of memory that will be used to save stack information. Can also be an Expand-Down Data Segment (Default / Big flag will be used to signify both settings). Additionally has to allow for both Read / Write, or it will cause a GP exception to be thrown.
- System Segments: Includes TSS and LDT

2.4.2 Segment Selector

The Segment Selector is a special pointer that identifies a segment in memory through descriptors in the GDT. It allows selecting any of the preconfigured entries, however selecting the first entry of the GDT (Null Segment Descriptor), will not directly generate an exception. But, if the `CS` or `SS` registers are accessed when they have a Segment Selector pointing to that entry as the value, it will cause a General Protection Fault (#GP) to be thrown.

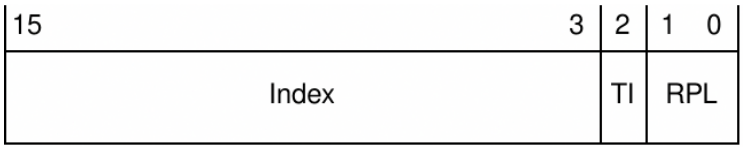


Image. 4: 16-bit Segment Selector (Figure 3-6. Segment Selector [1])

Bit	Name	Description
0-1	Requested Privilege Level	See RPL in Section 2.2
2	Table Indicator	Specifies Descriptor Table to be used: • 0: Selects the GDT • 1: Selects the current LDT
3-15	Index	Select one of the possible $2^{12} = 8192$ descriptors in the GDT or LDT. Processor simply multiplies value by 8 (size of the Segment Selector in bytes in 32-bit mode) and adds the result to the base address of the GDT or LDT.

Table. 11: Segment Selector Components (Vol. 3A, Chapter 3.4.2 Segment Selectors [1])

2.4.3 Descriptor Table Pointer

The descriptor table pointer defines an 80-bit structure in IA-32e 64-bit mode, used by both the `IDTR` and `GDTR` registers to allow the CPU access to the IDT and GDT respectively.

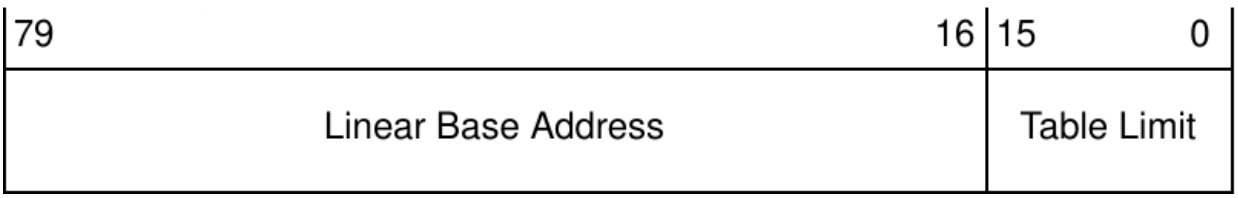


Image. 5: 80-bit Descriptor Table Pointer (Vol. 3A, Chapter 2.4 Memory-Management Registers [1])

Bits	Name	Description
16	Table Limit	Defines the amount of bytes in the Descriptor Table.
64	Linear Base Address	Defines the address of the first byte in the Descriptor Table.

Table. 12: Descriptor Table Pointer Components (Vol. 3A, Chapter 2.4 Memory-Management Registers [1])

2.4.4 Segment Registers

A segment register allows accessing a particular segment in memory. This is achieved using a Segment Selector to the Segment Descriptor in the GDT and then accessing the register, which will access the segment in memory.

Register	Full Name	Description
CS	Code Segment Register	Contains Segment Selector to the Code Segment, where the currently executing instructions are stored. Cannot be loaded explicitly and is instead loaded implicitly through instructions or internal processor operations that pass control to another segment.
SS	Stack Segment Register	Contains Segment Selector to a Stack Segment. In IA-32e 64-bit mode it can simply be the Data Segment as well.
DS	Data Segment Register	Contains Segment Selector to a Data Segment.
ES	Extra Segment Register	Contains Segment Selector to a Data Segment.
FS / GS	General Purpose Segment Registers	Contains Segment Selector to a Data Segment.

Table. 13: Segment Registers (Vol. 1, 3.4.2 Segment Registers [1])

2.4.5 Far Pointer

Allows referencing methods from Code Segments in the GDT, other than the currently loaded segment.

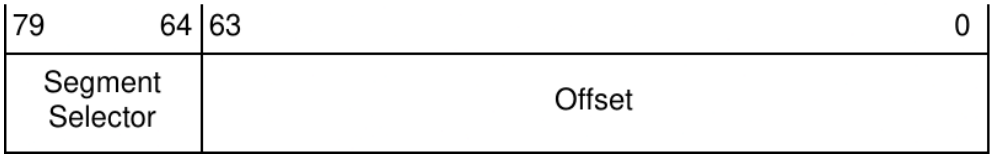


Image. 6: 64-bit Far Pointer (Figure 4-5. Pointers in 64-Bit Mode [1])

Bits	Name	Description
64	Offset	Defines the address to the procedure that should be called or jumped to.
16	Segment Selector	Selects the destination Code Segment, see Section 2.4.2.

Table. 14: Far Pointer Components (Vol. 3A, Chapter 4.3.1 Pointer Data Types in 64-Bit Mode [1])

2.5 Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is a data structure similar to the GDT. Its pointer is loaded into the Interrupt Descriptor Table register (`IDTR`).

The IDT can hold up to 256 entries, of which the first 22 have a predefined purpose defined by the x86 architecture, and an additional 10 are reserved for future use. The 224 remaining entries however, can be used as needed.

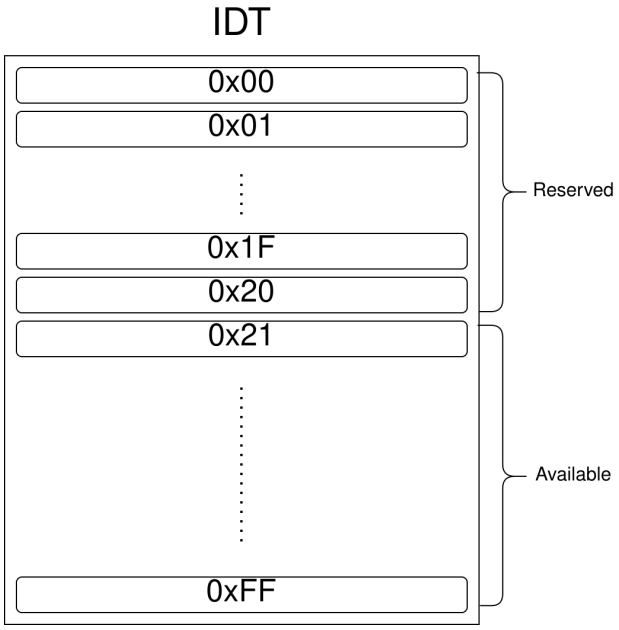


Image. 7: IDT items in x86 (Table 6-1. Exceptions and Interrupts [1])

Each interrupt type is assigned a unique ID, which the processor uses as the index of the IDT. This IDT entry then points to a specific interrupt handler. When an interrupt occurs, the processor stops executing the current task and transitions to that handler to address it. Once the handler has finished processing the interrupt, control is returned to the program or task that was interrupted.

The following table displays some of the more well known predefined interrupts and exceptions which must be present in form of an entry in the IDT.

Vector	Description	Source
0	Divide Error	`DIV` and `IDIV` instructions.
3	Breakpoint	`INT3` instruction.
10	Invalid TSS	Task switch or TSS access.
12	Stack Segment Fault	Stack operations and `SS` register loads.
14	Page Fault	Any memory reference.
16	Floating-Point Error (Math fault)	Floating-Point or `WAIT` / `FWAIT` instruction.

Table. 15: Exceptions and Interrupts (Table 6-1. Exceptions and Interrupts [1])

When an interrupt happens the type of the interrupt defines how the CPU must continue handling its instructions after the interrupt handler is done. This is defined by the gate type field in the IDT entry of the specific interrupt. In case of an exception, it is important to save the address of the currently executing instruction so that it can be retried. These interrupts are called traps. In case of an interrupt request, the address of the next instruction must be saved to prevent the CPU from repeatedly executing the same instruction.

A third kind of interrupt gate type exists, called the “Task Gate”. However, this type is not used in Long Mode and is specifically for hardware task switching, which is why it is not further referenced in this document.

2.5.1 Gate Descriptors

Similar to the GDT, the IDT contains multiple entries, which are called Gate Descriptors.

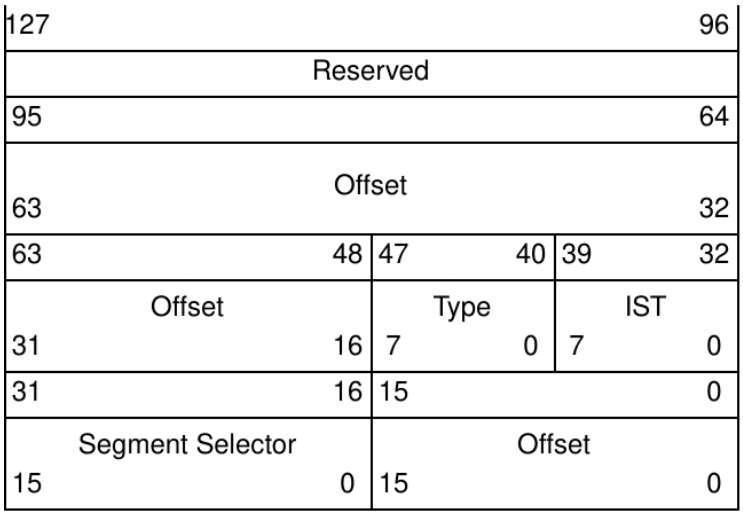


Image. 8: 64-bit Gate Descriptor (Figure 7-8. 64-Bit IDT Gate Descriptors [1])

The structure looks similar to a Segment descriptor (Section 2.4.1), this is the case because it also grew historically from a 16-bit and 32-bit variant of the same structure.

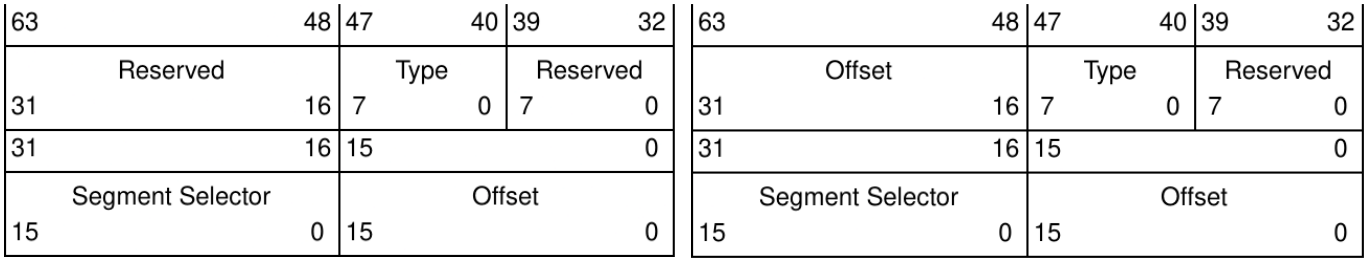


Table. 16: 16-bit Gate Descriptor (Figure 7-10. Gate Descriptor Format [4]) and 32-bit Gate Descriptor (Figure 9-3. 80306 IDT Gate Descriptors [5])

Bits	Name	Description
32	Reserved	Required because the 64-bit version of the Gate Descriptor requires a 64-bit base address instead of only a 32-bit address. This means our new entry would be 96 bit (64 bit from 32-bit Gate Descriptor + 32 bit for remaining half of the base address). However, to be compatible with the design of the 32-bit Gate Descriptor, two entries in little-endian format are now needed in the Interrupt Descriptor Table, because those two entries would add up to 128-bit, the remaining space has to be filled with reserved data.
64	Offset	Defines the address to the procedure handling the interrupt of this specific entry.
8	Type	Define 8 individual bits that change the behavior of the Gate Descriptor.
4	Interrupt Stack Table (IST)	Leftmost 3 bits are the index into the seven IST pointers of the TSS. Value 0 means the IST Mechanism is not used, Value 1 means IST1 is used. Only available in IA-32e 64-bit mode. (Vol. 3A, Chapter 7.14.5 Interrupt Stack Table [1])
16	Segment Selector	Selects the destination Code Segment, see Section 2.4.2

Table. 17: Gate Descriptor Components (Vol. 3A, Chapter 7.14.1 64-Bit Mode IDT [1])

2.5.1.1 Type Field

The Task Gate is a special case, because the 4th bit marks the usage of 32-bit, which is always disabled for a Task Gate. This is because the complete Hardware Task Switching mechanism does not exist on IA-32e 64-bit mode and instead has to be implemented in software. Therefore, the Task Gate can not be used in IA-32e 64-bit mode. (Vol. 3A, Chapter 9.7 Task Management in 64-bit Mode [1])

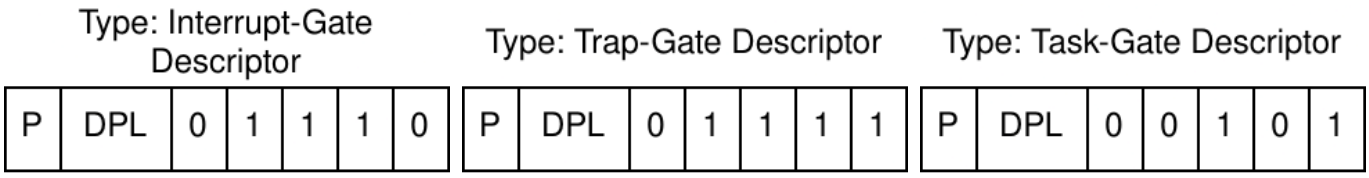


Table. 18: Differences in Type Field on different Gate Descriptor Types (Figure 7-2. IDT Gate Descriptors [1])

Bit	Name	Description
1-5	Type Field	Defines the actual type of the Gate. Can only be either 32-bit Interrupt or 32-bit Trap in IA-32e 64-bit mode.
6-7	Descriptor Privilege Level	Specifies the privilege level required to access this gate. A higher value means a less privileged access level is required. If only two access levels are required it is recommended to use the highest (0) and the lowest access level (3) respectively. • 0: Level 1 (Kernel) • 1: Level 1 (Used for System Services like Device Drivers) • 2: Level 2 • 3: Level 3 (User Applications)
8	Gate-present	Indicates whether the gate is present in memory or not. Will cause a Segment Not Present (#NP) exception, when this Segment Descriptor is loaded while the value is 0. Allows for more fine-grained control of segments that are actually loaded into physical memory at a give time. (Vol. 3A, Chapter Interrupt 11—Segment Not Present (#NP) [1]) • 0: Not present in memory • 1: Present in memory

Table. 19: Gate Descriptor Type Field (Little-Endian) (Vol. 3A, Chapter 7.14.1 64-Bit Mode IDT [1])

2.6 Task State Segment

The Task State Segment (TSS) is used for managing task switching and is primarily used for hardware task switching by storing the Segment Selectors and Stack Pointers of each privilege level. However, this only applies in Protected Mode. In Long Mode the task switching mechanism of the TSS is not supported. Regardless, the TSS in Long Mode must still exist, because it has been repurposed to hold information that is not directly related to the task-switch mechanism instead. (Vol. 3A, Chapter 9.7 Task Management in 64-bit Mode [1])

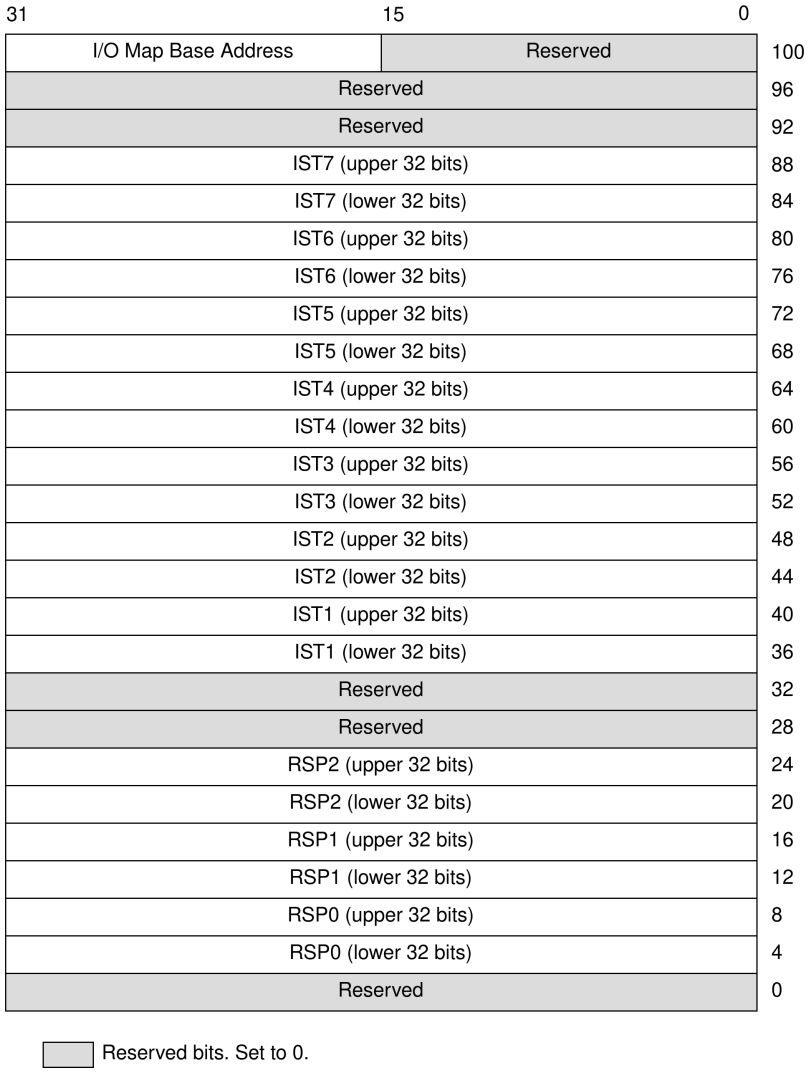


Image. 9: Long-Mode Task State Segment layout (Figure 9-11. 64-Bit TSS Format [1])

Field	Purpose
I/O Map Base Address	The 16-bit offset to the I/O permission bit map from the 64-bit TSS base
ISTn	Interrupt Stack Table (IST) pointers
RSPn	Stack Pointers (`rsp`) for privilege levels 0-2

Table. 20: Task State Segment fields (Vol. 3A, Chapter 9.7 Task Management in 64-bit Mode [1])

2.6.1 Memory Management Register

To actually use any of the configured memory management data structures, the corresponding registers have to be set. That is because the values contained in the registers allow the CPU to know where the data structures lie in memory.

Register	Name	Value Type
GDTR	Global Descriptor Table Register	Descriptor Table Pointer pointing to GDT start address in memory (see Section 2.4.3).
IDTR	Interrupt Descriptor Table Register	Descriptor Table Pointer pointing to IDT start address in memory (see Section 2.4.3).
TR	Task Register	Segment Selector pointing to Task State Segment (see Section 2.4.2).

Table. 21: Memory Management Registers (Vol. 3A, Chapter 2.4 Memory-Management Registers [1])

2.7 Procedure Calls

Procedure calls are a mechanism to access kernel functionality from User Mode. Generally there are three different options how such a call can be made. Each comes with its own advantages and disadvantages.

2.7.1 SYSCALL / SYSRET

SYSCALL / SYSRET are designed for 64-Bit Mode. SYSCALL is used in privilege level 3 to access operating system procedures, while SYSRET is used in privilege level 0 to return to level 3.

2.7.1.1 Overview

The processor uses certain preconfigured Model Specific Registers (MSR) as configuration on what the SYSCALL / SYSRET instructions actually do. The structure of these registers and how they should be configured can be seen below:

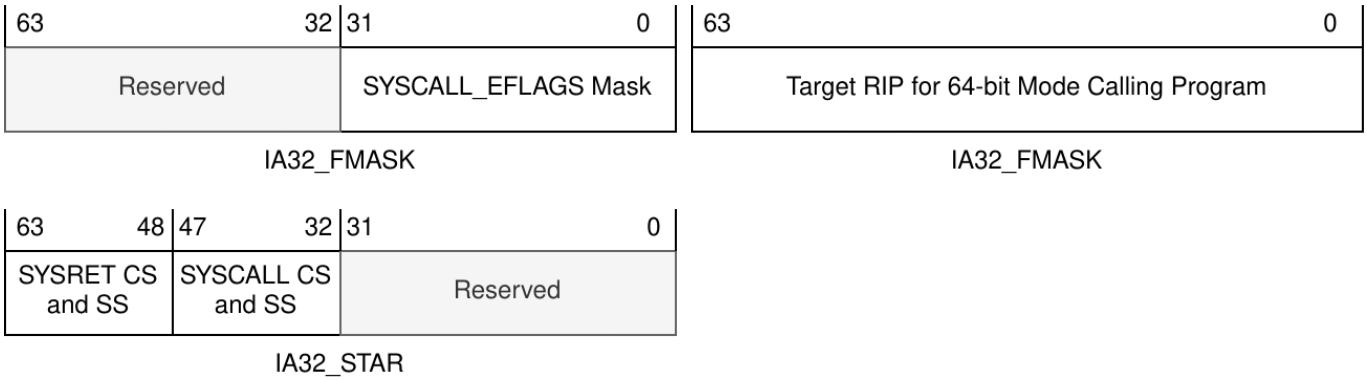


Table. 22: Fast System Calls Registers (Figure 6-14. MSRs Used by SYSCALL and SYSRET [1])

The values in this register is then used differently depending on if the SYSCALL or SYSRET instruction is executed.

MSR Name	Usage by SYSCALL	Usage by SYSRET
IA32_FMASK	RFLAGS mask to ignore certain bits. All bit set in the Mask will be disabled if they are currently enabled. RFLAGS & (~IA32_FMASK).	Not used. Instead RFLAGS are saved into R11 register by SYSCALL instruction. Restored when SYSRET instruction is executed into RFLAGS.
IA32_LSTAR	Target instruction pointer that will be called on SYSCALL instruction.	Not used. Instead return function is copied into by SYSCALL instruction. Which will be called when the SYSRET instruction is executed loading the value into RIP.
IA32_STAR	Uses the Segment Selector that is the SYSCALL CS and SS value, being bits 32:47 of the IA32_STAR as the base for calculations. • Target Code Segment the SYSCALL will switch into is simply this value. • Target Stack Segment the SYSCALL will switch into is simply this value + 8, results in Segment Selector Index + 1.	Uses the Segment Selector that is the SYSRET CS and SS value, being bits 48:63 of the IA32_STAR as the base for calculations. • Target Code Segment the SYSRET will switch into is simply this value + 16, results in Segment Selector Index + 2. • Target Stack Segment the SYSRET will switch into is simply this value + 8, results in Segment Selector Index + 1.

Table. 23: Fast System Calls in 64-Bit mode (Vol. 3A, Chapter 6.8.8 Fast System Calls in 64-Bit mode [1])

However, the SYSCALL / SYSRET instructions does not save the Stack Pointer. If the System Call intends to change the stack, this must be done by the operating system. (Vol. 3A, Chapter 6.8.8 Fast System Calls in 64-Bit mode [1])

2.7.1.2 Evaluation of SYSCALL and SYSRET

Advantages	Disadvantages
Specifically designed for fast System Call Entry / Exit	Does not automatically switch stacks or reload CS and SS registers from the GDT
Currently the go-to instruction for modern operating systems	Requires IA32_LSTAR, IA32_STAR, and IA32_FMASK Model Specific Registers (MSRs) to be properly set up
Automatically stores RIP and RFLAGS	Does not save the current Stack Pointer
Lower latency compared to INT n	Harder to set up compared to INT n

Table. 24: Advantages and disadvantages of SYSCALL / SYSRET instruction

2.7.2 SYSENTER / SYSEXIT

SYSENTER / SYSEXIT are designed for performance. As with the previous companion instructions, SYSENTER is used in privilege level 3 to access operating system procedures and SYSEXIT is used in privilege level 0 to return to level 3. However, these two instructions do not work on x86 on processors by all manufacturers, as they only work on x86_64 Intel processors.

2.7.2.1 Overview

MSR Name	Usage by SYSENTER	Usage by SYSEXIT
IA32_SYSENTER_CS	Uses the Segment Selector that is the IA32_SYSENTER_CS value as the base for calculations. • Target Code Segment the SYSENTER will switch into is simply this value. • Target Stack Segment the SYSENTER will switch into is simply this value + 8, results in Segment Selector Index + 1.	Uses the Segment Selector that is the IA32_SYSENTER_CS value as the base for calculations. • Target Code Segment the SYSEXIT will switch into is simply this value + 32, results in Segment Selector Index + 4. • Target Stack Segment the SYSEXIT will switch into is simply this value + 40, results in Segment Selector Index + 5.
IA32_SYSENTER_EIP	Target instruction pointer that will be called on SYSENTER instruction.	Not used. Instead return function is copied into by SYSENTER instruction. Which will be called when the SYSEXIT instruction is executed loading the value into RIP.
IA32_SYSENTER_ESP	Target Stack Pointer that will be loaded on SYSENTER instruction.	Not used. Instead the Stack Pointer is copied into by SYSENTER instruction. Which will be loaded into RSP when the SYSEXIT instruction is executed.

Table. 25: Fast System Calls in 64-Bit mode (Vol. 3A, Chapter 6.8.8 Fast System Calls in 64-Bit mode [1])

During SYSEXIT, the processor loads the new value for the CS register from the same bits as SYSENTER does, but to get the offset to the user Code Segment, it adds 32 to that number. This behavior forces a specific GDT layout, which is showcased in the table below.

Offset (Hex)	Offset (Dec)	Segment Descriptor
0x0	0	Null Segment
0x10	16	Kernel Code Segment
0x20	32	Kernel Data Segment
0x30	48	User Code Segment
0x40	64	User Data Segment

Table. 26: Expected GDT layout by SYSEXIT instruction

2.7.2.2 Evaluation of SYSENTER and SYSEXIT

Advantages	Disadvantages
Easy to setup	Only works on Intel processors in 64-bit mode
Specifically designed for fast System Call Entry / Exit	Not very educational (hides a lot of work from the developer)
Lower latency compared to `INT 0x80`	

Table. 27: Advantages and disadvantages of SYSENTER / SYSEXIT instructions

2.7.3 INT/IRETQ

In contrast to SYSCALL and SYSENTER, the instruction `INT n` is not specifically designed to execute operating system procedure calls. `INT n` exists to send an interrupt signal, which will be handled inside privilege level 3. But exactly due to this behavior, it can be used as a stand-in for SYSCALL / SYSENTER.

2.7.3.1 Overview

The operand `n` specifies a vector, which essentially is an index to a Gate Descriptor in the IDT. If the `INT n` instruction is used to handle system calls, the vector `0x80` should be used, as this is the vector used by the Linux kernel. The interrupt service routine (ISR) inside the gate descriptor at index 0x80 is then called and should contain a branching mechanism to determine which type of "SYSCALL" operation is being executed. The System Call type must be specified in the RAX register by the caller.

The registers RDI, RSI, RDX, RCX, R8 and R9 can be used to pass arguments to the ISR. However these registers are not persisted and available with the defined values in the ISR. These registers must be pushed to the stack before actually calling `INT 80` to be available. This can be done best by utilizing a wrapper function like this:

```
1  [[gnu::naked]]
2  auto system_call() -> void {
3    asm volatile(
4      "push %%rdi\n"
5      "push %%rsi\n"
6      "push %%rdx\n"
7      "push %%rcx\n"
8      "push %%r8\n"
9      "push %%r9\n"
10     "INT 80\n"
11     "ret"
12   );
13 }
```

Listing. 1: `INT 80` wrapper function

2.7.3.2 Evaluation of INT 80

Advantages	Disadvantages
System Call interface is just an interrupt and uses standard x86 interrupt mechanisms	Slower because of full interrupt handling overhead (saving / restoring many registers, checking privilege transitions, etc.)
Only needs the ISR to be set up once. No model specific registers are required	Not specifically optimized for frequent user / kernel transitions unlike actual system calls
Automatically handles privilege level transitions, stack switching (via TSS), and loading of new CS and SS registers	

Table. 28: Advantages and disadvantages of INT n instruction in context switching

2.7.4 Decision

With these three options available, a decision must be made as to which of them the TeachOS should utilize. Because this operating system is for educational purposes, the most common implementation used in the most operating systems should be preferred, even if it is more complex to set up. Therefore, TeachOS will use `SYSCALL` / `SYSRET` as its procedure call mechanism.

This decision is motivated by the following factors:
- **Industry relevance**: Modern 64-bit operating systems like Linux and Windows use `SYSCALL` / `SYSRET`, making it the most practical and educational choice.
- **Performance**: Despite its complexity, `SYSCALL` provides the best performance due to its lower overhead compared to `INT n`.
- **Learning value**: Setting up the MSRs and handling privilege transitions manually, gives deeper insight into system internals, which aligns with the goals of TeachOS.

Future extensions could add support for `INT n` as a fallback for compatibility or educational comparison, but the primary implementation will use `SYSCALL`.

2.7.5 Passing Arguments to System Calls

There are multiple ways to pass arguments to a System Call.

Option	Description	Advantages	Disadvantages
Register	Easiest way to pass arguments. Simply uses certain predefined CPU registers and reads the values from them.	• Very fast • Easy to implement	• Limited number of available registers • Callee has to save and restore the used registers if the old values are needed • Passing less arguments is not differentiable from passing all arguments
Stack	Requires configuring System Calls with multiple stacks. Then uses that configured stack to pass arguments.	• Makes Nested System Calls possible (Calling System Call in a System Call) • Mirrors the C way to pass arguments to function • No limitation on amount of parameters	• Requires allocating different stack section and switching before a System Call • Passing less arguments is not differentiable from passing all arguments
Memory	Caller needs to store a pointer to arguments location and saves that address in a register. Alternatively the memory location of the arguments can also be fixed.	• No limitation on amount of parameters • Secure because memory can be accessible to Kernel Mode only	• Nested System Calls possible only if the arguments are copied • Passing less arguments is not differentiable from passing all arguments

Table. 29: Passing Arguments to System Call (Chapter 11. x86 Assembly Language Programming, A.3.2. Alternate Calling Convention [6])

2.7.6 Decision

With these three options available, a decision must be made as to which of them the TeachOS should utilize. Because this operating system is for educational purposes, the implementation used by Linux should be preferred. Additionally, it removed the need to implement the overhead of a Stack infrastructure, which would probably not have been possible in the remaining time of this thesis.

This decision is motivated by the following factors:
- **Mirroring Linux**: The Linux Operating System uses the same way to pass parameters to System Calls, therefore seeing this way of passing arguments implemented helps to understand how it works on Linux as well. (syscall(2) - Linux manual page [7])
- **Learning value**: Loading and storing the registers for each passed argument provides the deepest insight into system internals, which aligns with the goals of TeachOS.

Future extensions could add support for the option utilizing memory as well, because it still partially uses registers and negates the drawback of limited parameters. Both calls using either method of passing parameters could then be supported to additionally expand the learning experience. Additionally, actually implementing the Stack infrastructure could be useful as well, because it shares the underlying need for the infrastructure to allow mapping and allocating physical frames for Stack space with the Interrupt Stack.

2.7.7 Using System Calls with multiple stacks

System Calls generally can work with a single stack. However, if the Stack method to pass parameters to System Calls is used this is not possible anymore.

Instead, to use the stack for procedure calls, it must be setup first, because the stack is contained in a segment in the GDT and identified by the Segment Selector. This can be done through the following steps, where steps 2 and 3 must be repeated before each System Call.

1. Create a Stack Segment and add it to the GDT
2. Load the Segment Selector for the Stack Segment into the SS register
3. Load the Stack Pointer for the stack into the ESP register. The LSS instruction can be used to load the SS and ESP registers in one operation

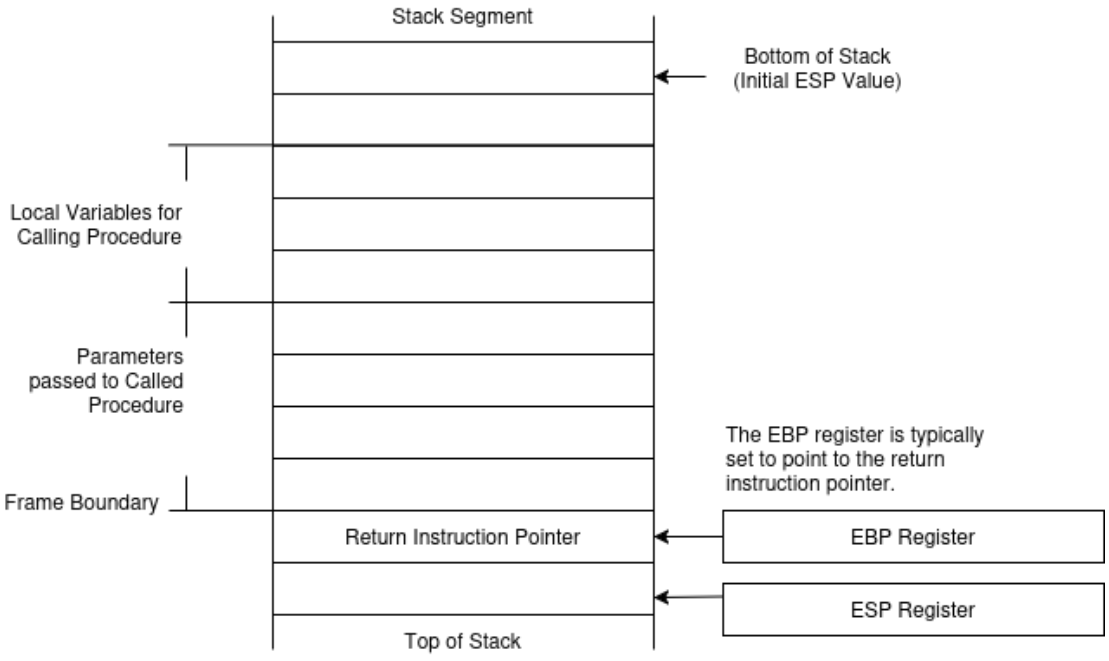


Image. 10: Stack Segment Layout (Figure 6-1. Stack Structure [1])

Part 3
Results

3.1 Memory Management

3.1.1 Current problem

Currently, the heap memory can be managed by utilizing “placement new”, which instantiates an object at the provided memory address.

```
memory::heap::linked_list_allocator heap_allocator{
  memory::heap::HEAP_START,
  memory::heap::HEAP_START + memory::heap::HEAP_SIZE
};
auto heap_address = heap_allocator.allocate(1024);
auto object = new (heap_address) memory::multiboot::memory_information{};
```

Listing. 2: Old way of allocating heap memory

This can now be utilized to instantiate our own local instance of a heap allocator that can then be used to allocate blocks of memory by hand.

There are multiple issues with this implementation:

- Heap allocator is local to a method and not shared globally (Multiple instances of heap allocators might attempt to write into the same heap area in the kernel)
- Requires individual calls to allocate and deallocate to actually create and free the memory
- Requires using placement new every time a new object on the heap should be created
- Does not allow for the usage of C++ STL types like `std::vector`

3.1.2 Overriding new and delete

To fix the issue of only being able to use the placement new, the instantiation of heap memory by using the simpler and more widespread `new` operator has to be allowed. Calls to `new` and `delete` are also required for implementing components of the C++ STL, allowing to create custom implementation.

This is not directly possible, because the `new` and `delete` keywords do not magically use the correct heap allocator. Instead, to be able to use these two operators, the following operators must be overwritten with a custom implementation.

```
auto operator new(std::size_t size) -> void *;
auto operator delete(void * pointer) noexcept -> void;
auto operator delete(void * pointer, std::size_t size) noexcept -> void;
auto operator new[](std::size_t size) -> void *;
auto operator delete[](void * pointer) noexcept -> void;
auto operator delete[](void * pointer, std::size_t size) noexcept -> void;
```

Listing. 3: Required global operator overrides

3.1.3 Global Heap Allocator

In our case the methods have to be overwritten with calls to a global instance of an arbitrary heap allocator. This allows fixing the problem, where multiple instances of the heap allocator might write to the same area in memory. Now there only exists one global shared instance that is used instead.

To achieve that use case, a global static class called global heap allocator is created. This static class needs to contain:

- A Factory Method that instantiates the actual heap implementation. This can only be done once the kernel and, especially, the heap area has been fully remapped
- A Getter for the heap implementation that should be used

Using this global heap allocator allows to simply bind `new` and `delete` calls to the corresponding static calls of the global memory manager. This also allows removing the separate allocation calls, which were previously necessary, because this is now handled by the calls to `new` and `delete`.

3.1.4 Factory Method Pattern

The Factory Method Pattern was used to make the underlying heap allocation implementation easily configurable. This is possible, because a specific method creates the used heap allocation implementation, instead of directly instantiating it in the constructor, which would cause the global heap allocator to have a fixed underlying heap allocation strategy.

Therefore, that method requires a parameter that changes which instance should be created in the factory method and all future calls to the global heap allocator will use that instance. (Factory Method [8])

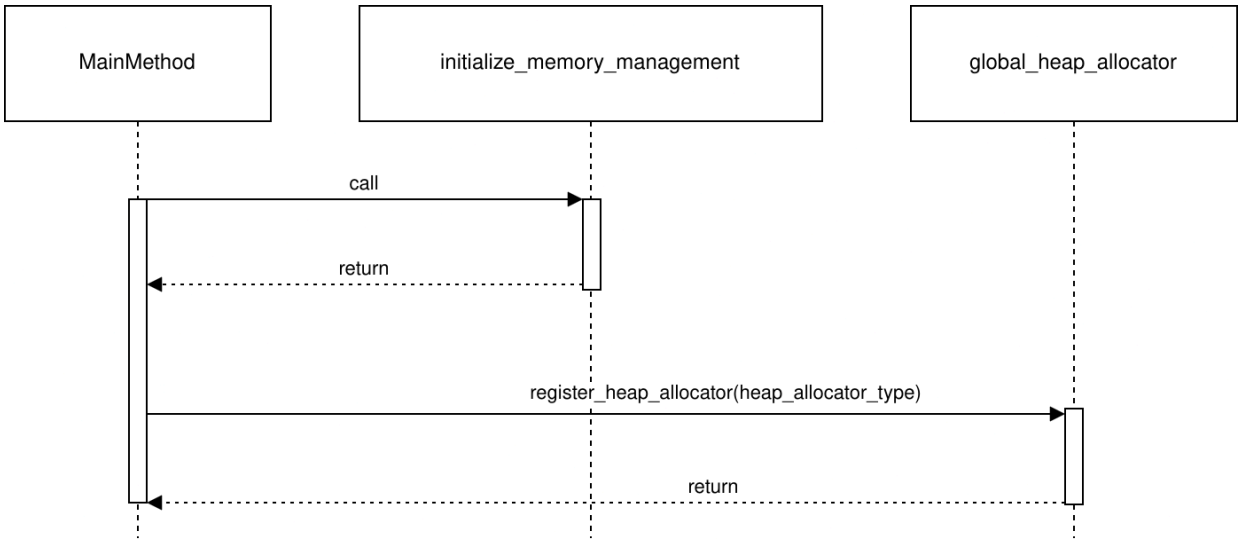


Image. 11: Sequence Diagram for function call with Factory Method Pattern

This additionally allows the configuration of the global instance to be done after initializing the memory management of the kernel. This has to be done, because the global heap allocation accesses the kernel heap area and if that section has not been remapped already the operating system would crash.

3.1.5 Improving the Linked List Allocator

Currently, the linked list allocator has multiple issues in its implementation, which makes using it incompatible with the newly created global heap allocator:

- Deallocating a memory area starting from a certain address is only possible if the size of that memory area is known as well. This is the case, because the linked list only keeps track of free memory blocks and as soon as a block is allocated, the total allocated size of that block is not recoverable anymore.
- Allocating an `int` or other data smaller than 16 bytes on the heap is not possible. This is the case because when freeing up the memory area again, the `linked_list_allocator` needs enough space to safe both the total size of the block and a pointer to the next free block in the linked list. Otherwise, the free memory area can not be attached to the linked list and therefore the memory is leaked, because it can not be recovered.

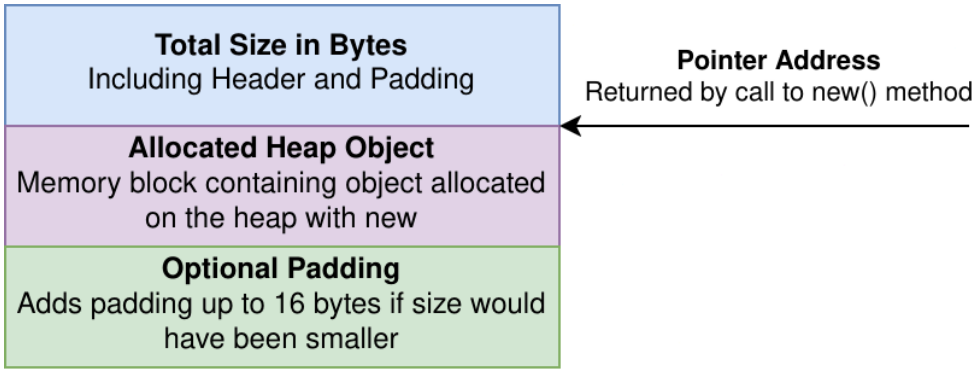


Image. 12: Structure of allocated memory area of the Linked List allocator

To resolve the first issue, the allocation algorithm has to be adjusted. Instead of simply allocating the necessary amount of memory for the heap object, it needs to allocate enough space for an additional header as well. This header then simply contains the total size of allocated memory, allowing that value to be read when deallocating. This removes the need for knowing the size that was allocated in the first place.

Furthermore, this also allows resolving the second issue. Now that the size of the allocated memory block can be expanded without leaking memory, because the size of the allocation is managed internally, the total size in the header can simply be increased to the necessary 16 bytes if the allocated object would be smaller.

3.2 Context Switching

3.2.1 Segment Descriptor

The Segment Descriptors need to be tightly packed, because certain fields like the Segment Limit or Base Address fields are split over multiple partial bytes. For example, the first portion of the Base Address is only 24 bits, but C++ only allows allocating fields with sizes that are powers of two. In this case the value would need to be in an 32-bit integer. Luckily C++ has a feature, that allows for a field size to not be a power of 2 in bits, called C++ Bit-Fields.

```
struct S {
    uint8_t b1 : 3; // 1st 3 bits (in 1st byte) are b1
    uint8_t    : 2; // next 2 bits (in 1st byte) are blocked out as unused
    uint8_t b2 : 6; // 6 bits for b2 - doesn't fit into the 1st byte => starts a 2nd
    uint8_t b3 : 2; // 2 bits for b3 - next (and final) bits in the 2nd byte
};
```

Listing. 4: C++ Bit-Field (Article “Bit-Field” [9])

This feature allows including multiple values in one byte, which normally would not be possible. Additionally, it also allows using an unnamed Bit-Field for the 32-bit reserved field, because the value of that field should never be set or read anyway, it is simply used to provide backwards compatibility with the 32-bit version of the Segment Descriptor.

There is an additional problem with the example above. The member variable `b2`, will not save the left-most 3 bits of its value into the remaining space of b1, but instead start a new byte. This is the case because the compiler automatically adds padding to byte-align a field.

This is a problem for the Segment Descriptor which, as previously established, needs to be tightly packed. To make that possible, the `[[gnu::packed]]` attribute can be used.

```
struct [[gnu::packed]] S { ... };
```

Listing. 5: Packed Structures (15.6 Packed Structures [10])

This will tightly pack all members of the struct, causing the aforementioned left-most 3 bits of `b2` to still be in the first Byte where `b1` resides.

These features combined allow creating a Segment Descriptor that uses 128 bit, where every member is tightly packed, with overlapping byte borders and no padding.

```
struct [[gnu::packed]] segment_descriptor_base {
  uint16_t _limit_1 = {};      ///< First part of the limit field (0 - 15)
  uint32_t _base_1 : 24 = {};  ///< First part of the base field (16 - 39)
  access_byte _access = {};    ///< Access byte field (40 - 47)
  gdt_flags _flag = {};        ///< Second part of the limit field + Flags field (48 - 55)
  uint8_t _base_2 = {};        ///< Second part of the base field (56 - 63)
};

struct [[gnu::packed]] segment_descriptor_extension {
  segment_descriptor_base _base = {};  ///< Base Segment Descriptor = Single entry in GDT (0 - 63)
  uint32_t _base_3 = {};               ///< Third part of the base field (63 - 95)
  uint32_t : 32;                       ///< Reserved field used to ensure 128 bits lnegth (96 - 127)
};
```

Listing. 6: Segment Descriptor Structures

However, the Global Descriptor Table requires certain entries to use the 32-bit Segment Descriptor (`segment_descriptor_base`) instead of being able to use the 64-bit Segment Descriptor (`segment_descriptor_extension`) for all entries. Therefore, the implementation needs to make the 64-bit Segment Descriptor an extension of the 32-bit Segment Descriptor. The exact reason is explained in more detail in Section 3.2.2.

To allow the configuration of both Segment Descriptors to be as readable as possible. To achieve this a `std::bitset` in combination with an unscoped `enum` is used, because it allows for direct conversion into an integral type, which also means multiple enum values can be combined with the binary OR-operator (`|`) in a call to the constructor.

```
std::bitset<20U> constexpr LIMIT{0xFFFFF};
uint64_t constexpr BASE = 0x0;
access_byte user_data_access_byte{access_byte::PRESENT | access_byte::ACCESS_LEVEL_USER |
                                  access_byte::CODE_OR_DATA_SEGMENT | access_byte::WRITABLE};
gdt_flags user_data_gdt_flags{gdt_flags::GRANULARITY | gdt_flags::UPPER_BOUND, LIMIT};
segment_descriptor_base user_data_segment{user_data_access_byte, user_data_gdt_flags, BASE, LIMIT};

uint64_t constexpr TSS_LIMIT = sizeof(task_state_segment) - 1;
access_byte const tss_access_byte{access_byte::PRESENT | access_byte::DESCRIPTOR_LEVEL_KERNEL |
                                  access_byte::TASK_STATE_SEGMENT_AVAILABLE};
gdt_flags const tss_gdt_flags{0U, TSS_LIMIT};
segment_descriptor_extension const tss_descriptor{tss_access_byte, tss_gdt_flags,
                                                  reinterpret_cast<uint64_t>(tss), TSS_LIMIT};
```

Listing. 7: Segment Descriptors Configuration example

The same is the case for the `contains_flags` method, which, thanks to implicit integral conversion, allows to be called with multiple enum values with the bitwise-OR operator (`|`) directly as well.

```
flags.contanins_flags(gdt_flags::GRANULARITY | gdt_flags::UPPER_BOUND);

auto gdt_flags::contains_flags(std::bitset<4U> other) const -> bool
{
  return (std::bitset<4U>{_flags} & other) == other;
}
```

Listing. 8: Contains flag call and implementation

The `contains_flags` utilizes `std::bitset`, because it easily allows to only compare certain bits without the need for bit-shifting, which makes the code much easier to read. Additionally, the overridden binary AND-operator (`&`) can be used to create a new `std::bitset` that only sets the bits if it is set in both operands.

If the result of this operation is then compared with the `std::bitset` passed as an argument, and it returns true, then the argument is either the same as the internal value or a subset of it.

Furthermore, because the unscoped `enum` itself is in a struct, the members of the `enum` will not leak into the namespace, but instead into the struct. This means the structs name still needs to be specified before an `enum` value can be accessed.

An enum also makes it possible to define multiple names for the same value, this is useful for the Segment Descriptor, because as seen in Section 2.4.1, the actual meaning of a bit varies depending on the type of Segment Descriptor.

```
struct [[gnu::packed]] gdt_flags
{
  enum bitset : uint8_t {
    LENGTH = 1U << 0U,
    UPPER_BOUND = 1U << 1U,
    STACK_POINTER_SIZE = 1U << 1U,
    DEFAULT_LENGTH = 1U << 1U,
    GRANULARITY = 1U << 2U,
  }

  gdt_flags(uint8_t flags, std::bitset<20U> limit);

  auto contains_flags(std::bitset<4U> other) const -> bool;

private:
  uint8_t _limit_2 : 4 = {};
  uint8_t _flags : 4 = {};
};
```

Listing. 9: GDT Flags Field Structure

The field `gdt_flags` however is a special case, because every struct created has to be at least one byte in size, even if packed structs and Bit-fields are used. Therefore, the `gdt_flags` additionally has to include the right-most 4 bits of the limit field, so that the total of the members result in one byte, because otherwise the Segment Descriptor would not be tightly packed anymore, because `gdt_flags` would be padded to one byte instead.

3.2.2 Global Descriptor Table

The GDT is a simple custom vector implementation that contains multiple base Segment Descriptors.

The exact Segment Descriptors and their order is strictly enforced in long mode, because the Base and Limit values are ignored. This is the case, because in Long Mode the usage of segmentation is disallowed, and it strictly enforces the usage of paging to protect memory regions. Therefore, the Base field starts at address 0 and with the Limit field set to the maximum value, it allows access to the complete memory region.

The GDT only requires a Null Descriptor, one descriptor for each combination of privilege level, and segment type, and a Task State Segment.

Offset	Name	Content
0x0000	Null Descriptor	Base = 0, Limit = 0x00000, Access Byte = 0x00, Flags = 0x0
0x0008	Kernel Mode Code Segment	Base = 0, Limit = 0xFFFFF, Access Byte = 0x9A, Flags = 0xC
0x0010	Kernel Mode Data Segment	Base = 0, Limit = 0xFFFFF, Access Byte = 0x92, Flags = 0xC
0x0018	User Mode Code Segment	Base = 0, Limit = 0xFFFFF, Access Byte = 0xFA, Flags = 0xA
0x0020	User Mode Data Segment	Base = 0, Limit = 0xFFFFF, Access Byte = 0xF2, Flags = 0xC
0x0028	Task State Segment	Base = &TSS, Limit = sizeof(TSS) - 1, Access Byte = 0x89, Flags = 0x0

Table. 30: 64-bit minimal GDT for Long Mode Setup (Article “GDT Tutorial” [2])

As shown in the table above all Segment Descriptors, besides the TSS Descriptor, only use the 32-bit Segment Descriptor. That is the case, because only the TSS Descriptor, actually contains a 64-bit address to the TSS. Whereas the other Segment Descriptors ignore the Base and Limit fields, and assuming these fields are always 32-bit makes the structure compatible with older 32-bit implementations (e.g. Fast System Call in Section 2.7.1).

This however causes another issue, because a vector can only contain data of the same type. Therefore, to circumvent ugly workarounds with the vector containing only raw integer data and then converting to and from Segment Descriptors, the `segment_descriptor_extension` can simply be split into two `segment_descriptor_base`.

These two entries can then simply be inserted into the Global Descriptor Table like expected, which allow keeping the `global_descriptor_table` as a vector of `segment_descriptor_base`.

```
1 global_descriptor_table global_descriptor_table{...,
2                                                 tss_descriptor.get_first_gdt_entry(),
3                                                 tss_descriptor.get_second_gdt_entry()};
```

Listing. 10: Global Descriptor Table Configuration

Once the complete GDT has been configured, it needs to be saved into memory and kept alive for the remainder of the system lifetime. That is because a Descriptor Table Pointer pointing to the GDT (see Section 2.4.3) needs to be loaded into the Global Descriptor Table Register (GDTR).

```
1  struct [[gnu::packed]] global_descriptor_table_pointer
2  {
3    uint16_t table_length = {};
4    segment_descriptor * address = {};
5  };
6
7  global_descriptor_table_pointer gdt_pointer{static_cast<uint16_t>((gdt.size() *
   sizeof(segment_descriptor)) - 1), gdt.data()};
8
9  auto load_global_descriptor_table(global_descriptor_table_pointer const & gdt_pointer) -> void
10 {
11   asm volatile("lgdt %[input]" : /* no output from call */ : [input] "m"(gdt_pointer));
12 }
```

Listing. 11: Loading GDT Pointer Structure (Vol. 2A, LGDT/LIDT—Load Global/Interrupt Descriptor Table Register [1])

To initialize the `global_descriptor_table_pointer` correctly, the value of the Size Limit field has to be passed first, which is the size of the GDT in bytes. This can be calculated by multiplying the amount of entries in the GDT with the size of each entry and subtracting one for byte 0.

Additionally, the member variable `address` is of type `segment_descriptor` instead of `global_descriptor_table`, because the pointer must refer to the first entry of the GDT, and not to the address of the vector on the heap. To then actually pass the memory address pointing to the first entry two methods can be used:

- `gdt.data()`: Returns the underlying c-style array, which is simply the memory address pointing to the start of the actual elements on the heap.
- `&gdt.at(0)`: Returns the reference to the first element, which can then be used to access its address.

Passing this structure to the `load_global_descriptor_table` method overrides the GDTR entry that has been preconfigured in the assembler file. The asm constraint `m` has to be used to signify that it should be interpreted as a memory address.

At this point, the `CS` register still contains a value referencing the previous GDT. To actually register the changes to the GDTR, the Segment Registers need to be reset. This can be done through a Far Call using a Far Pointer (see Section 2.4.5), where the Segment Selector has to point to the first non-null entry of the GDT (Vol. 2A, CALL — Call Procedure [1]).

```
auto call(far_pointer pointer) -> void
{
  asm volatile("rex64 lcall *%[input]" : /* no output from call */ : [input] "m"(pointer));
}

interrupt_descriptor_table::segment_selector segment_selector{
    1U, interrupt_descriptor_table::segment_selector::REQUEST_LEVEL_KERNEL};
kernel::cpu::far_pointer pointer{&kernel::cpu::reload_segment_registers, segment_selector};
kernel::cpu::call(pointer);
```

Listing. 12: Far jump to reload `CS` registers

3.2.3 Interrupt Descriptor Table (IDT)

Once the GDT has been set up the IDT can be created and initialized. The IDT is required to handle interrupts and exceptions produced by the operating system or an application.

Its setup is very similar to the GDT, requiring a globally available vector of Gate Descriptors. The initial setup requires 32 entries where each index corresponds to a different exception or interrupt type that has to be handled. 32 entries, because that is the amount that is reserved by the architecture see Section 2.5.

To get the IDT up and running, these entries are simply connected to a generic interrupt handler that essentially does nothing. This is enough to initialize a working IDT, but will not handle the interrupts or exceptions correctly.

```
auto create_interrupt_descriptor_table() -> interrupt_descriptor_table
{
  // Only account for the reserved Vectors for now (0 - 31)
  interrupt_descriptor_table interrupt_descriptor_table{32};

  uint64_t offset = reinterpret_cast<uint64_t>(interrupt_handling::generic_interrupt_handler);
  segment_selector selector{1U, segment_selector::REQUEST_LEVEL_KERNEL};
  ist_offset ist{0U};
  idt_flags flags{idt_flags::DESCRIPTOR_LEVEL_KERNEL | idt_flags::INTERRUPT_GATE |
idt_flags::PRESENT};

  for (std::size_t i = 0; i < 32; i++)
  {
    interrupt_descriptor_table.at(i) = {selector, ist, flags, offset};
  }

  return interrupt_descriptor_table;
}
```

Listing. 13: Creating the interrupt descriptor table

To implement interrupt handlers, which correctly handle an interrupt signal, a more advanced setup would be required, were different interrupt handlers are called, depending on the actual interrupt received. Additionally, the `ist_offset` can be used to allow certain interrupts to always execute in a known good stack. This is especially useful for a double-fault. A double- fault

results in a triple-fault, if another interrupt occurred within its handler and a triple-fault always results in a system crash (Vol. 3A, Chapter 7.14.5 Interrupt Stack Table [1]).

Since these interrupts will not be handled properly in this operating systems current state, no Interrupt Stack Table (IST) needs to be created and registered. This allows setting the value of `ist_offset` to 0, which signals the system that the IST is not used. If this mechanism were to be used the TSS would need to point to 7 allocated stack areas in its IST1 through IST7 pointers.

The IDT has to be loaded into the `IDTR` register using the `lidt` instruction. To achieve that a `idt_pointer` is created, whose structure exactly matches the `gdt_pointer`, but instead of pointing to the first Segment Descriptor it points to the first Gate Descriptor.

```
struct [[gnu::packed]] interrupt_descriptor_table_pointer
{
  uint16_t table_length = {};
  gate_descriptor * address = {};
};

interrupt_descriptor_table_pointer idt_pointer{static_cast<uint16_t>((idt.size() * sizeof(gate_descriptor)) - 1), idt.data()};

auto load_interrupt_descriptor_table(interrupt_descriptor_table_pointer const & idt_pointer) -> void
{
  asm volatile("lidt %[input]" : /* no output from call */ : [input] "m"(idt_pointer));
}
```

Listing. 14: Loading IDT Pointer Structure (Vol. 2A, LGDT/LIDT—Load Global/Interrupt Descriptor Table Register [1])

In contrast to the GDT and its loading mechanisms, no Far Call is required for the processor to recognize the newly created IDT. Instead, the interrupts that were disabled before the GDT was set up simply have to re-enabled, and it will start using the newly configured interrupt handler.
Interrupts should only be re-enabled after the entire context-switching process is fully completed. Re-enabling them too early could allow the program to access Segment Registers that still reference the old GDT structure, leading to undefined behavior or system errors. [3]

3.2.4 Interrupt Service Routine (ISR)

As mentioned before, only a generic interrupt service routine has been implemented. The interrupts themselves are not actually addressed in this handler.

```
[[gnu::interrupt]]
auto generic_interrupt_handler(interrupt_frame * frame) -> void
{
  (void)frame;
  video::vga::text::write("An Interrupt occurred.", video::vga::text::common_attributes::green_on_black);
}
```

Listing. 15: Generic interrupt handler

The `[[gnu::interrupt]]` annotation marks this function as an interrupt handler for the compiler. With it, the compiler will generate the function entry and exit sequences necessary to be able to use it as an interrupt handler (5.24 Declaring Attributes of Functions [11]).

```
...
push   %rax             # Preserve original value of all general-purpose register
...
lea    0x8(%rbp),%rax   # Save address of the interrupt frame on the stack frame
mov    %rax,-0x10(%rbp)
...
mov    -0x8(%rbp),%rax
iretq                   # Return from the interrupt
```

Listing. 16: C++ Assembly with `[[gnu::interrupt]]`

This leads to general-purpose registers being saved when entering and restored before leaving the interrupt method. Additionally, arguments are not passed using a common calling convention. Meaning, the `RDI` register can not simply be read for

the passed interrupt frame. Instead they are read directly by interacting with the stack frame without manipulating it. This is done through negative indexes like this: `mov %rdi,-0x8`.

3.2.5 Task State Segment (TSS)

The Task State Segment Descriptor requires additional work, because the Base field is not ignored and instead has to point to the actual Task Segment in memory, see Section 2.6.

```
struct [[gnu::packed]] task_state_segment
{
private:
  uint32_t : 32;
  uint64_t rsp0 = {};
  uint64_t rsp1 = {};
  uint64_t rsp2 = {};
  uint64_t : 64;
  uint64_t ist1 = {};
  uint64_t ist2 = {};
  uint64_t ist3 = {};
  uint64_t ist4 = {};
  uint64_t ist5 = {};
  uint64_t ist6 = {};
  uint64_t ist7 = {};
  uint64_t : 64;
  uint16_t : 16;
  uint16_t io_map_base_address = {};
};
```

Listing. 17: Task State Segment Structure

For now the Interrupt Stack Table is not used and the values are default initialized.

Similar to the GDT and IDT, the TSS has to be loaded into the `TR` register, but only after the GDT has been fully loaded into the `CS` register. This is essential, because when executing the “Load Task Register” (`ltr`) instruction, the CPU accesses the TSS Descriptor to set the “Accessed” Flag. If the correct GDT is not loaded at this point, it will cause an exception.

3.2.6 Changing Segment Registers

To actually switch control into another segment, the Segment Registers have to be set up correctly. This can be done once the GDT and IDT have been configured, because the Segment Registers need to point to valid Segment Descriptors in the GDT using a Segment Selector.

The actual Segment Registers can be seen in Section 2.4.4. To switch the context, the values of these registers needs to be changed. This can simply be achieved by using `mov` statements in inline assembly that copy the value of the Segment Selector pointing to the User Data Segment Descriptor into the registers.

```
asm volatile("xor %%rax, %%rax\n"
             "mov %[input], %%ax\n"
             "mov %%rax, %%ds\n"
             "mov %%rax, %%es\n"
             "mov %%rax, %%fs\n"
             "mov %%rax, %%gs\n"
             : /* no output from call */
             : [input] "m"(data_segment)
             : "rax");
```

Listing. 18: Changing Data Segment Registers

Changing the value of the `SS` register can also be done using the `mov` instruction, but it cannot simply be used with any arbitrary value. The `SS` register must be loaded with a Segment Selector, that references a writable data segment.

Changing the value of `CS` is even more difficult. This can not be done directly but instead requires an instruction or internal processor operation that switches the control to another segment. That is the case, because the `CS` register displays the Code Segment we are currently executing code on. Therefore, changing the `CS` register always entails Context Switching.

There are multiple options that can achieve that; The Far Jump, Far Call, Interrupt, Far Return or Interrupt Return. In our case the Interrupt Return was chosen, because the other calls do not work in IA-32e mode. This is the case because, in IA-32e mode, a privilege level change from kernel mode to user mode requires a controlled context switch that updates the Code Segment, Stack Segment, and privilege level atomically, which is something that only the interrupt return mechanism is designed to handle. After setting up the kernel, the default mode the operating system runs in should be the User Mode. Felix Cloutier describes the purpose of the `IRET` statement as follows:

> "Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task."
>
> — Felix Cloutier [12]

Derived from this, using the `IRET` instruction instead of `JMP` or `CALL`, tricks the system into assuming that it is *returning* into the User Mode, and not just jumping there. This results in future `SYSCALL` and other ring level changing calls, to automatically return into User Mode when the `IRET` instruction is executed.

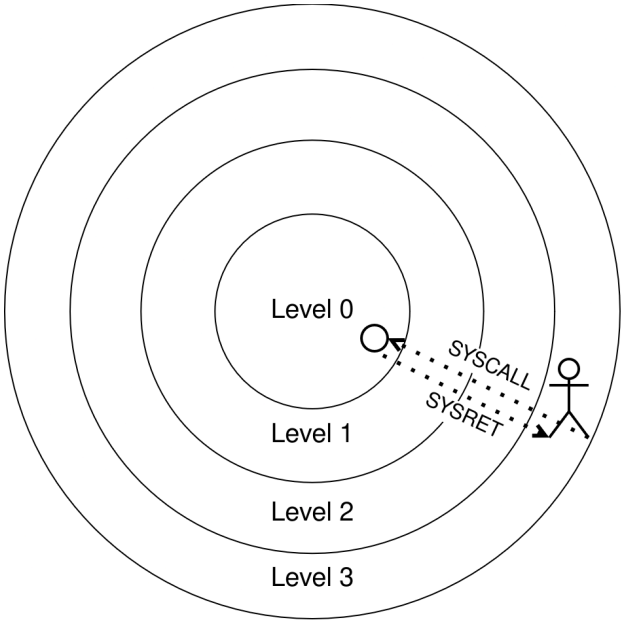


Image. 13: Executing SYSCALL in User Mode

This also increases safety because all future access requiring kernel mode, must be handled by `SYSCALL` and will only be in that mode for the limited execution time.

This is necessary because kernel mode provides access to critical system functionalities that should not be used after the initial system setup, except through predefined `SYSCALL` instructions. Unrestricted access would allow direct manipulation of CPU registers and memory, which could be catastrophic. For example, a user application could overwrite the IDT or GDT structures or disable interrupts entirely. To prevent this, an interrupt return must be used to safely switch to the user mode while making it the default operating mode.

Additionally, the `iretq` instruction, has to be called because the default instruction `iret` is 32 bits in size. Therefore, because IA-32e in 64-bit mode is used the 64-bit version of the instruction has to be utilized instead.

Furthermore, the stack needs to be configured accordingly and push multiple different values. This is the case because the instruction pops values from the stack in the following order:

- Return Instruction Pointer
- Return Code Segment Selector
- Extended Flags Register (`EFLAGS`)
- 64-bit Stack Pointer Register (`RSP`)
- Stack Segment Register (`SS`)

The actual call results in a Far Return because the Nested Task (NT) flag in the `EFLAGS` register must always be cleared in IA-32e mode. This is required since hardware task switching is not supported in IA-32e mode. If the NT flag was set, it would

trigger a General Protection Fault (#GP) when executing an interrupt return (Vol. 2A, IRET/IRETD/IRETQ— Interrupt Return [1]).

```
asm volatile("mov %%rsp, %%rax\n"
             "push %[data_segment]\n"
             "push %%rax\n"
             "pushfq\n"
             "push %[code_segment]\n"
             "mov %[return_function], %%rax\n"
             "push %%rax\n"
             "iretq\n"
             : /* no output from call */
             : [data_segment] "m"(data_segment)
             , [code_segment] "m"(code_segment)
             , [return_function] "r"(address)
             : "rax");
```

Listing. 19: Changing Code Segment Register

This additionally causes a problem, because if the stack is updated in C++, the method call should not generate additional stack pushes or pops. These pre- and post-ambles are normally generated by the compiler in the assembler code and includes returning from the method and saving the previous stack of the method that called.

```
push   %rbp            ; Save address of previous stack frame
mov    %rsp,%rbp       ; Copy address of current stack frame
...
sub    $0x8,%rsp       ; Save space for 8 bytes on the stack frame
mov    %rdi,-0x8(%rbp) ; Copy the passed value expected to be in RDI register into stack
...
leave                  ; Pops the saved previous stack frame (pop %rbp)
ret                    ; Returns control from this method to the one that called it
```

Listing. 20: C++ Assembly without `[[gnu::naked]]`

The preamble must be removed because it pushes the frame pointer (RBP) register onto the stack and loads the current stack pointer (RSP) into the frame pointer register. This is of course a problem for this setup, because the stack is modified and does not match the expected stack of IRETQ anymore.

Additionally, the preamble will allocate space for the arguments on the stack frame and copy the values from the registers (where the arguments are initially expected to be) into the corresponding slots on the stack. This preamble is only generated in non-`[[gnu::naked]]` mode and only if the function actually has arguments that are passed to it.

The post-amble however, executes the LEAVE and RET instructions. LEAVE essentially restores the frame- and stack pointer by executing the reverse instructions of the preamble. In itself this would not be an issue for this function, because the IRETQ instruction is called before the post-amble is ever executed, but skipping the generation of only the preamble is not possible.

Skipping the generation of both, can be achieved with the `[[gnu::naked]]` specifier above the function, which prevents the creation of any additional assembly statements. This will however also remove the generated RET statement in assembly, meaning if the method does not call RET it will not return control out of the function anymore. Furthermore, using this attribute on any statement besides inline assembly is not recommended, because the function is mangled if the correct assembly instructions are not added (5.24 Declaring Attributes of Functions [11]).

```
[[gnu::naked]]
auto example() -> void { ... }
```

Listing. 21: Preventing C++ Assembly pre- and post-amble

This attribute will additionally generate an Undefined Instruction (UD) after the C++ Assembly itself. This is done to ensure that the C++ Assembly actually returns, because if it does not, the UD instruction will be called which generates an Invalid Op-Code Exception (#UD).

3.3 Separating User from Kernel Mode

Once the return function has been entered, all code will now be executed in the user mode. Accessing Kernel functionality however, is at this point only partially blocked. To completely separate them and ensure safety, more measures must be taken.

3.3.1 Configuring and Enabling System Calls

The first step is to separate system functionality into system calls. This serves as the foundation to later disallow access to this functionality from user mode if it is not called through SYSCALL.

To achieve this, the SYSCALL instruction needs to be configured and enabled. This is done by setting the required MSRs with the expected values and by setting the System Call Extensions bit in the EFER flags (see Section 2.7.1).

```
auto enable_syscall() -> void
{
  uint64_t const syscall_function = reinterpret_cast<uint64_t>(syscall_handler);
  kernel::cpu::write_msr(IA32_LSTAR_ADDRESS, syscall_function);
  kernel::cpu::write_msr(IA32_FMASK_ADDRESS, 0U);

  uint64_t const kernel_cs = KERNEL_CODE_SEGMENT_SELECTOR;
  uint64_t const star_value = (kernel_cs << 32) | (kernel_cs << 48);
  kernel::cpu::write_msr(IA32_STAR_ADDRESS, star_value);

  kernel::cpu::set_efer_bit(kernel::cpu::efer_flags::SCE);
}
```

Listing. 22: Configuring and Enabling the SYSCALL instruction

This has to be called before the actual switch to user mode occurs, because calling certain privileged instruction once the user mode switch has occurred will cause a General Protection Fault (#GP) to be thrown. Write Model Specific Register WRMSR, which is used to configure the SYSCALL instruction is one of those privileged instructions. (Vol. 3A, Chapter 6.9 Privileged Instructions [1])

3.3.2 Implementing and Wrapping System Calls

Once the SYSCALL instruction has been configured the next step is actually implementing the `syscall_handler` function for the required functionality and wrapping it to provide an normal method interface and remove the need to directly set CPU registers.

```
auto syscall(type syscall_number, arguments args) -> response
{
  asm volatile("mov %[input], %%rax" :: [input] "m"(syscall_number) : "memory");

  asm volatile("mov %[input], %%rdi " :: [input] "m"(args.arg_0) : "memory");
  asm volatile("mov %[input], %%rsi" :: [input] "m"(args.arg_1) : "memory");
  asm volatile("mov %[input], %%rdx" :: [input] "m"(args.arg_2) : "memory");
  asm volatile("mov %[input], %%r10" :: [input] "m"(args.arg_3) : "memory");
  asm volatile("mov %[input], %%r8" :: [input] "m"(args.arg_4) : "memory");
  asm volatile("mov %[input], %%r9" :: [input] "m"(args.arg_5) : "memory");

  asm volatile("syscall");

  arguments values{};
  asm volatile("mov %%rdi, %[output]" : [output] "=m"(values.arg_0));
  asm volatile("mov %%rsi, %[output]" : [output] "=m"(values.arg_1));
  asm volatile("mov %%rdx, %[output]" : [output] "=m"(values.arg_2));
  asm volatile("mov %%r10, %[output]" : [output] "=m"(values.arg_3));
  asm volatile("mov %%r8, %[output]" : [output] "=m"(values.arg_4));
  asm volatile("mov %%r9, %[output]" : [output] "=m"(values.arg_5));
  error error_code{};
  asm volatile("mov %%rax, %[output]" : [output] "=m"(error_code));

  return {error_code, values};
}
```

Listing. 23: Preparing arguments and loading return values for SYSCALL instruction

This is done by passing an `enum` indicating the specific `type` of `SYSCALL` that should be executed, which is then moved into the `RAX` register. And by passing the optional `arguments struct`, up to 6 arguments that will be passed to the `SYSCALL` instruction can be used.

This passing of the argument is done through CPU Registers (`RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9`). For the reasoning and other possible implementations see Section 2.7.5.

These same CPU registers are then also reused to return up to 6 possible arguments as well as a possible error code signaling if and why the `SYSCALL` failed.

3.3.3 Implementing the System Call Handler

These arguments can then be read from the according CPU registers when the System Call Handler is executed by the instruction.

```
auto syscall_handler() -> void
{
  uint64_t return_instruction_pointer, rflags = {};
  asm volatile("mov %%rcx, %[output]" : [output] "=m"(return_instruction_pointer));
  asm volatile("mov %%r11, %[output]" : [output] "=m"(rflags));

  uint64_t syscall_number, arg_0, arg_1, arg_2, arg_3, arg_4, arg_5 = {};
  asm volatile("mov %%rdi, %[output]" : [output] "=m"(arg_0));
  asm volatile("mov %%rsi, %[output]" : [output] "=m"(arg_1));
  asm volatile("mov %%rdx, %[output]" : [output] "=m"(arg_2));
  asm volatile("mov %%r10, %[output]" : [output] "=m"(arg_3));
  asm volatile("mov %%r8, %[output]" : [output] "=m"(arg_4));
  asm volatile("mov %%r9, %[output]" : [output] "=m"(arg_5));

  asm volatile("mov %%rax, %[output]" : [output] "=m"(syscall_number));

  response result;
  switch (static_cast<type>(syscall_number)) { ... }

  asm volatile("mov %[input], %%rax" :: [input] "m"(result.error_code) : "memory");

  asm volatile("mov %[input], %%rdi" :: [input] "m"(result.values.arg_0) : "memory");
  asm volatile("mov %[input], %%rsi" :: [input] "m"(result.values.arg_1) : "memory");
  asm volatile("mov %[input], %%rdx" :: [input] "m"(result.values.arg_2) : "memory");
  asm volatile("mov %[input], %%r10" :: [input] "m"(result.values.arg_3) : "memory");
  asm volatile("mov %[input], %%r8" :: [input] "m"(result.values.arg_4) : "memory");
  asm volatile("mov %[input], %%r9" :: [input] "m"(result.values.arg_5) : "memory");

  asm volatile("mov %[input], %%rcx" :: [input] "m"(return_instruction_pointer) : "memory");
  asm volatile("mov %[input], %%r11" :: [input] "m"(rflags) : "memory");

  asm volatile("leave\n"
               "sysretq");
}
```

Listing. 24: `SYSCALL` handler implementation

That instruction additionally also saves the Return Instruction Pointer into the `RCX` Register, as well as the `RFLAGS` which are saved in the `R11` Register. That has to be done, because calling another method in the `switch case` statement, will override the Return Instruction Pointer. Therefore to ensure that returning from the method is still possible both values have to be cached first so they can be restored later, before `SYSRET` is called.

The optional arguments returned from the method as well as the error code are then saved into the according CPU registers, so they can be read after the `SYSRET` has occurred.

Returning from the System Call Handler would already work like expected, but then if the next return from a method is hit the program will crash. That is the case, because if the `leave` instruction is not called before calling the `SYSRET` instruction, then the stack space allocated to the stack frame by the System Call Handler itself will not be cleaned up. That is the case, because the `SYSCALL` does not return normally from the method instead using an early return with `sysretq` and therefore the `leave` instruction is not called implicitly by the return generated by C++ but has to be called explicitly instead. (Vol. 2A, LEAVE—High Level Procedure Exit [1])

3.3.4 Creating and Moving Code into User Linker Section

With the SYSCALL infrastructure set up, `.user_code` and `.user_data` linker sections should now be separated from the kernel's code and data section to disallow direct access to Kernel resources from user mode, which would otherwise directly circumvent the protection mechanism of the privilege levels.

Firstly all functions which need to be executable in user mode are annotated with the `[[gnu::section(".user_text")]]` annotation. But doing this does not yet fully define into which section it is linked. This is because in the linker script, all available sections are either kept or discarded by the linker. For the `.user_text` section to be kept, it must be added in the following way:

```
.user_text ALIGN(4K) : AT(ADDR (.user_text))
{
  *(.user_text .user_text.*)
}
```

Listing. 25: Adding `.user_text` section to the Linker script

The same needs to be done for all static variables which are accessed in user mode, because they need to be annotated with `[[gnu::section(".user_data")]]`. Otherwise the static variables will be put into the Kernel Data Section `.data` instead. Similar to above a separate `.user_data` section needs to be kept as well. For normal member variables this annotation does not have to be added, because they are simply placed into the User Data Section where they are instantiated instead.

Thanks to these annotation all code and data, which should be accessible in User Mode is in its own section. However, the C++ STL should also be accessible in user mode. But putting that code into a specific section is not possible in the same manner, because GNU annotations can not be added to already compiled code. Therefore, the C++ STL must be added to the section by referencing its archive file (group of objects or static libraries) instead.

```
.stl_text ALIGN(4K) : AT(ADDR (.stl_text))
{
  *(.stl_text .stl_text*)
  KEEP(*libstdc++.a:*(.text .text.*))
}
```

Listing. 26: Adding the standard library to a specific section

3.3.4.1 Template Code Workaround

The section `.stl_text` contains only some parts of the C++ STL, as well as the custom implementations that were created. However template definition are not fully linked into the `.stl_text` section.

> "The compiler must have full visibility to the definition at the time of the instantiation. In the above mentioned .cpp, you instantiate this class Template with specific types. The compiler will generate full class type definition at the point of this explicit instantiation."
>
> — Martin Vorbrodt [13]

Because of both of those rules, the templates are always mapped into the Linker Section which first uses them, which is the Kernel Linker Section. Because of the One-Definition Rule (ODR) the Linker then does not create another copy of those templates in the User Linker Section.

Due to this behavior, when accessing the templated definition of the C++ STL the User Mode will attempt to access code that is in the Kernel Text Section `.text`. Which will then cause the Page Fault Exception (#PF), unless the User Mode is explicitly allowed access to Kernel Mode code.

3.3.5 Mapping Pages User Accessible

To allow access to these section in User Mode, they need to be mapped with the user accessible flag turned on. For this to work the alignment of the sections is important, because the pages themselves also have a size of 4 KiB. If this alignment is neglected, the mapping process would be increasingly difficult.

Usually the Linker Sections are mapped to pages using their names.

```
if (section.name == ".user_text") {
  entry.set_user_accessible();
}
```

Listing. 27: Mapping Linker Sections to pages by name example

However, this requires setting up access to the Symbol Table, which contains these names. The remaining time of this thesis does not allow this mechanism to be implemented. For that reason, the Linker Sections are mapped to pages using their starting address instead.

These addresses usually change when the Linker script is updated. So each time a new Linker Section is added or removed, the `map_elf_kernel_sections` function inside the `kernel_mapper.hpp` file must be checked for correctness and updated accordingly.

This can be achieved by querying the addresses of each section using the `readelf -WSl build/bin/Debug/_kernel` command in the console, which prints the following output.

```
Section Headers:
[Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
[ 1] .boot_rodata      PROGBITS        0000000000100000 001000 0000f7 00   A  0   0  8
[ 2] .boot_text        PROGBITS        0000000000101000 002000 0001ae 00  AX  0   0 16
[ 3] .boot_bss         NOBITS          0000000000102000 0021ae 104000 00  WA  0   0 4096
[ 4] .boot_data        PROGBITS        0000000000206000 003000 000004 00  WA  0   0  1
[ 5] .init             PROGBITS        0000000000207000 004000 000010 00  AX  0   0  1
[ 6] .fini             PROGBITS        0000000000208000 005000 00000b 00  AX  0   0  1
[ 7] .stl_text         PROGBITS        0000000000209000 006000 0000d9 00  AX  0   0 16
[ 8] .text             PROGBITS        000000000020a000 007000 00ded0 00  AX  0   0 32
[ 9] .user_text        PROGBITS        0000000000218000 015000 00105c 00  AX  0   0  2
[10] .rodata           PROGBITS        000000000021a000 017000 000bd3 00   A  0   0 32
[11] .ctors            PROGBITS        000000000021b000 018000 000010 00  WA  0   0  8
[12] .dtors            PROGBITS        000000000021c000 019000 000010 00  WA  0   0  8
[13] .bss              NOBITS          000000000021d000 019010 000500 00  WA  0   0 32
[14] .data             PROGBITS        000000000021e000 01b000 000018 00  WA  0   0  8
[15] .user_data        PROGBITS        000000000021f000 01c000 000020 00  WA  0   0 16
[16] .debug_line       PROGBITS        0000000000000000 01c020 00fb28 00      0   0  1
[17] .debug_line_str   PROGBITS        0000000000000000 02bb48 002631 01  MS  0   0  1
[18] .debug_info       PROGBITS        0000000000000000 02e179 06d714 00      0   0  1
[19] .debug_abbrev     PROGBITS        0000000000000000 09b88d 01045e 00      0   0  1
[20] .debug_aranges    PROGBITS        0000000000000000 0abcf0 003690 00      0   0 16
[21] .debug_str        PROGBITS        0000000000000000 0af380 049107 01  MS  0   0  1
[22] .debug_rnglists   PROGBITS        0000000000000000 0f8487 00221f 00      0   0  1
[23] .debug_macro      PROGBITS        0000000000000000 0fa6a6 03a6d4 00      0   0  1
[24] .debug_loclists   PROGBITS        0000000000000000 134d7a 00030f 00      0   0  1
[25] .symtab           SYMTAB          0000000000000000 135090 005d90 18     26 910  8
[26] .strtab           STRTAB          0000000000000000 13ae20 016840 00      0   0  1
[27] .shstrtab         STRTAB          0000000000000000 151660 000118 00      0   0  1
```

Listing. 28: Output of `readelf` command

This output can then be used to get the hard-coded address to all sections that need to be mapped as user accessible.

```
std::array<uint64_t, 6U> constexpr USER_SECTION_BASES = {
    0x102000, 0x209000, 0x218000, 0x21F000, 0x20A000
};
```

Listing. 29: Section addresses mapped as user accessible

Which means, this variable contains the addresses for `.boot_bss`, `.stl_text`, `.user_text`, `.user_data` and temporarily, until the previously mentioned issue is resolved, also `.text`.

The section `.boot_bss` usually contains only uninitialized static variables. However, it still needs to be mapped user accessible as well. This is because the `.boot_bss` also includes the stack used by both User and Kernel Mode.

If `.boot_bss` would not be mapped user accessible, it would therefore cause a Page Fault Exception (#PF) to be thrown when entering the main method because when entering a method, local variables are pushed to the stack. To actual fix this separate stack infrastructure for Kernel and User Mode would need to be implemented, which is out of scope for this thesis.

To actually remap these pages as user accessible a simple for-loop is used that accesses all configured ELF Sections and maps them to actual physical frames. It is then possible to simply additionally add the user accessible flag for all ELF Sections that are in the predefined array.

```
if (std::ranges::find(USER_SECTION_BASES, section.physical_address) != USER_SECTION_BASES.end())
{
  entry.set_user_accessible();
}
```

Listing. 30: Looping through selected sections to mark as user accessible

This has the additional problem of only setting the Level 1 Page Table Entry to be user accessible. All other Page Table Entries (Level 4, Level 3 and Level 2) are currently not mapped user accessible.

This of course causes an issue, because even if the Level 1 Page Table Entry is mapped user accessible if the other Page Table Entries leading to it are not mapped user accessible as well a Page Fault Exception (#PF) will still be thrown.

Therefore to fix this the underlying mapping mechanism has to be adjusted so any Page Table Entries that contain Level 1 Page Table Entries that are user accessible, need to be user accessible themselves.

```
auto next_table_or_create(T & allocator, std::size_t table_index, std::bitset<64U> flags) ->
page_table_handle
{
  ...

  // Check if the now created or previously created level 4, level 3 or level 2 page entry is used
  // by user accessible code. If it is that page entry needs to be user accessible as well.
  entry entry{flags.to_ulong()};
  if (entry.contains_flags(entry::USER_ACCESSIBLE))
  {
    this->operator[](table_index).set_user_accessible();
  }
  return next_handle.value();
}
```

Listing. 31: Fixing Issue with Shared Level 4 Page Table Entries between Kernel and User Mode

This is achieved by additionally checking in the mapping code if the flags used by the Level 1 Page Table Entry is user accessible. If it is then the previously created or newly created Page Table Entry needs to also be user accessible.

3.3.6 Creating a User Heap Allocator

The ELF Sections however are not the only thing that needs to be mapped user accessible. To separate Kernel from User Heap there also needs to be a separate allocator for User Mode, which uses a different Heap Section for the actual allocations.

This has been achieved through a custom User Heap Allocator, which is based on the Linked List Allocator used by the Kernel with a few major differences.

The constructor is defaulted and the Allocator does not allocate any memory when it is first instantiated. This allows to only map the User Heap into physical memory once the first Heap Allocation occurs and therefore allows to separate the Kernel Heap mapping from the User Heap mapping code.

Additionally, the `allocate()` method does not directly fail if there is no Heap Memory left, instead it attempts to expand the User Heap by a fixed amount (100 KiB) instead. If that is successful it uses the exact same behaviour as the Linked List Allocator and simply allocates a block with the given size into that Memory Block.

```
auto user_heap_allocator::allocate(std::size_t size) -> void *
{
  ...

  current = expand_heap_if_full();

  if (current != nullptr)
  {
    auto memory = allocate_into_memory_block_if_big_enough(current, previous, total_size);
    if (memory.has_value())
    {
      return memory.value();
    }
  }

  char constexpr OUT_OF_MEMORY_ERROR_MESSAGE[] = "[Linked List Allocator] Out of memory";
  context_switching::syscall::syscall(context_switching::syscall::type::ASSERT,
                                      {false,
reinterpret_cast<uint64_t>(&OUT_OF_MEMORY_ERROR_MESSAGE)});
  return nullptr;
}
```

Listing. 32: User Heap Allocate Implementation

Additionally it is not possible to directly assert and the SYSCALL Implementation of it has to be called instead, if expanding the User Heap failed.

The same is the case for the `expand_heap_if_full()` method, because being able to directly map Heap Memory in the User Mode could be potentially dangerous. Therefore, the implementation is done by a SYSCALL instead, which will expand the heap by a fixed amount (100 KiB) and return the starting address of that memory.

```
auto user_heap_allocator::expand_heap_if_full() -> memory_block *
{
  auto const result =
context_switching::syscall::syscall(context_switching::syscall::type::EXPAND_HEAP);

  uint64_t const heap_start = result.values.arg_0;
  uint64_t const heap_size = result.values.arg_1;
  return !result.error_code ? new (reinterpret_cast<void *>(heap_start)) memory_block(heap_size,
nullptr) : nullptr;
}
```

Listing. 33: Expanding User Heap Implementation

The implementation of the System Call itself is relatively easy. It simply starts by expanding the User Heap from a fixed address and increases the area that is assigned to the User Heap by the hard-coded amount (100 KiB). This is done using the same method the Kernel Heap uses to map its physical frames.

```
auto expand_user_heap() -> response
{
  static auto current_heap_end = memory::heap::USER_HEAP_START;
  uint64_t const heap_start = current_heap_end;
  memory::remap_heap(heap_start, memory::heap::USER_HEAP_SIZE,
                     memory::paging::entry::USER_ACCESSIBLE);
  current_heap_end += memory::heap::USER_HEAP_SIZE;
  return {error::OK, {heap_start, memory::heap::USER_HEAP_SIZE}};
}
```

Listing. 34: Expanding User Heap System Call Implementation

The method additionally saves the previous User Heap End Address so that address can simply be used as the next starting point if the User Heap is expanded multiple times. This has an additional advantage of the virtual addresses in the User Heap staying continuous even with expansion at runtime.

Part 4
Assembly caveats

4.1 GNU Assembly (GAS)

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 from vanilla assembly. In the case of `mov`, the value of the first argument will be moved into the second.
- Writing plain values into registers can not be done by using the values as is. Instead plain values must be surrounded by `$()`. For example, the vanilla assembly move statement `rax, 1000` must be written like this: `mov $(1000), %rax`.
- Variables when copied from or when copied into, have to be accessed with a dollar sign prefix: `$variable`.
- When writing values into an offset whose base address is contained in a variable, the address must be wrapped with braces `()` instead of brackets `[]`. For example `mov %rax, [$variable + 1]` must be written like this: `mov %rax, (variable + 1)`.

4.2 Inline assembly

Using extended asm, operands can be specified, which allow to specify input registers, output registers and a list of clobbered registers. The format for extended inline assembly is the following:

```
asm (assembler template
   : output operands                /* optional */
   : input operands                 /* optional */
   : list of clobbered registers    /* optional */
);
```

Listing. 35: Example inline assembly with output variable

4.2.1 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.

```
asm volatile("sgdt %[output_label]" : [output_label] "=m"(output_value) : /* No input to call */);
```

Listing. 36: Example inline assembly with output variable

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.

4.2.2 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.

```
asm volatile("lgdt %[input_label]" : /* No output from call */ : [input_label] "m"(input_value));
```

Listing. 37: Example inline assembly with input variable

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.

4.2.3 Clobber list

Some instructions clobber some hardware registers. These registers need to be listed in the list of clobbered registers mentioned earlier in the extended asm template. This will inform GCC that these registers are being used and modified. This includes registers implicitly clobbered by the instructions in the assembler templates.

Input and output registers should not be listed in the clobber list, because GCC is aware of them being used.

4.2.4 Operand constraints

Constraints define whether an input or output operand is in a register, a memory address or other details. The most used are the operands “r” and “m”. Below is a list of commonly used constraints.

Constraint	Purpose
r	The operand should be placed in a general-purpose register. The compiler is free to choose any available register.
m	The operand should be placed in memory and should be accessed from a memory location.
g	A “generic” constraint, meaning the operand can either go in a register or memory, depending on what the compiler thinks is best.
i	Defines an immediate value. For example a constant that is directly embedded in the assembly code.
n	A constant that is a valid operand in an instruction that takes a number, like a shift amount or an immediate value.

Table. 31: Common operand constraints (5.35.1 Simple Constraints [10])

4.2.5 Constraint modifiers

Operand constraints can be prefixed with a modifier for precise control over their effects. The following two modifiers are the most used ones:

Modifier	Purpose
=	The operand is written by the instruction and the previous value is replaced by the output.
&	The operand is an “earlyclobber” operand, which is modified before the instruction is finished using the input operands. This prevents the compiler from allocating the same register or memory location for an input and an output operand.

Table. 32: Common constraint modifiers (5.35.3 Constraint Modifier Characters [10])

4.2.6 Compile time optimization

The C++ compiler does not understand the content of the inline assembly statement. Because of that, the compiler might move or delete it during compile time optimizations. GCC will never remove an asm instruction, if it has an output, however it might still be removed. Using the `volatile` keyword after `asm` instructs GCC to not do any optimization around it.

4.2.7 Loading effective address

To load an address of a variable instead of the content of the variable, which is required for function references `mov` can not be directly used when using raw assembly.

```
1 lea [label], %rax
2 mov %rax, %rdx
3
4 asm volatile("mov %[input]", %%rdx : /* No output from call */ : [input] "m"(label));
```

Listing. 38: Loading a function: Assembly vs. Inline Assembly

However, in inline assembly it is possible, because the variable passed to the `mov` instruction is already an address in C++.

Part 5
Future Work

The operating system is now able to manage memory through paging as well as heap allocation and deallocation in User Mode. Furthermore, it is possible to switch between different ring levels thanks to the newly implemented GDT and IDT.

From here there are a few possible ways to further improve TeachOS and getting it closer to a viable teaching tool. Below are the topics which can be implemented with the current state of the operating system.

5.1 Interrupt Handler

Currently, only one interrupt handler has been created, which does not actually handle the interrupts. It simply prints the information that an interrupt has occurred and then returns.

Implementing different interrupt handlers for the various different x86_64 reserved interrupts is a logical and necessary next step. A good understanding of why a certain interrupt happened, what the possible state of the affected registers is and what the state of the affected registers must be acquired to correctly implement these handlers and serves as a good foundation for an operating system focused on teaching.

5.2 System Call over Interrupts

To show more possible implementations of System Calls, support for System Calls over `INT n` could be added. With the possibility of switching between them and displaying the difference to the student using the operating system.

5.3 Interrupt Stack

Modern x86_64 CPUs support the Interrupt Stack Table (IST) as part of the TSS. Setting up dedicated stacks for critical exceptions like double faults (which may occur due to stack corruption) enhances system reliability.

5.4 System Call Parameter Passing

System Calls could additionally be expanded with the possibility of supporting memory to pass parameters as well. Additionally, using the Stack to pass parameters could be implemented as well. This can be achieved much easier as well if the infrastructure to create Interrupt Stacks has been created already or vice versa.

5.5 Keyboard Interrupts

Now that interrupts can be caught and properly handled, an interesting feature to implement is properly handling keyboard interrupts. This would allow a user to interact with the system and provides a base for a proper shell.

5.6 Creating a User Mode Stack

Currently, TeachOS only utilizes one stack for both Kernel and User Mode. This is suboptimal and results in an issue with the mapping of Linker Sections to pages. This issue has been introduced in Section 3.3.5, and boils down to the operating system throwing a page fault, if the section `.boot_bss` is not mapped as user accessible, since it contains the stack data.

This behavior can be improved by creating a user stack and mapping it to a different Linker Section, which is in turn mapped as `user_accessible`. When this is implemented, the section `.boot_bss` can be removed from the list of sections to map as user accessible.

5.7 Educational Structure

The primary purpose of TeachOS is its use in various lectures as an educational tool. To fulfill this purpose, a possible next step is implementing different memory management strategies and an architecture which allows students to switch between

them. This can be expanded even more by implementing a graphical representation of the loaded pages, memory hits and misses and other data surrounding memory management.

Part 6
Glossary

Term	Meaning
C++ STL	C++ Standard Template Library; A collection of generic classes and functions that provide common data structures and algorithms.
CPL	Current Privilege Level
DPL	Descriptor Privilege Level; The privilege level of the segment. Also defines the minimum privilege level to access the segment.
EFER	Extended Feature Enable Register; A model-specific register to allow enabling `SYSCALL`/`SYSRET` instructions.
GAS	GNU Assembler; The assembler developed by the GNU Project.
GCC	GNU Compiler Collection; A collection of compilers from the GNU Project.
GDT	Global Descriptor Table; Defines segments through segment descriptors.
`GDTR`	Global Descriptor Table Register; Stores the GDT Pointer and is used by the CPU to access the GDT.
#GP	General Protection Fault; Occurs when executing privileged instructions while not in ring 0.
IDT	Interrupt Descriptor Table; Defines interrupt types and is the logical link between interrupt number and interrupt service routine.
`IDTR`	Interrupt Descriptor Table Register; Stores the IDT Pointer and is used by the CPU to access the IDT.
ISR	Interrupt Service Routine; A code section executed during interrupt handling to restore the system to a valid state.
IST	Interrupt Stack Table; The stack pointers used to load the stack when a privilege level change occurs.
LDT	Local Descriptor Table; Holds segment descriptors just like the GDT. The difference is that every task can have its own LDT and the `LDTR` can bee changed on every task switch.
MSR	Model Specific Register; Control registers used for debugging, program execution tracing, computer performance monitoring and toggling CPU features.
#NP	Segment Not Present; Exception that occurs when trying to load a segment or gate which has its “present” bit set to 0.
#PF	Page fault; Exception that occurs when a page table is not present in physical memory or when a protection check failed.
RPL	Requested Privilege Level; Privilege level of a segment selector.
Segment Registers	Registers that help manage memory by dividing it into segments. • Code Segment (`CS`) -> Defines the memory location where the program code is stored. • Data Segment (`DS`) -> Defines where data of the program will be stored. • Extra Segment (`ES`) -> Defines where additional data will be stored. • Stack Segment (`SS`) -> Defines where the stack is stored. • General Purpose Segment (`FS`/`GS`) -> No processor defined purpose.
SP	Stack Pointer; A register (`rsp`) containing the memory address of the last data element added to the stack.
`TR`	Task Register; Segment selector pointing to the TSS.
TSS	Task State Segment; Data structure used to change the stack pointer after an interrupt or CPL change occurred.
UD	Undefined Instruction; Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. **Not to confuse with #UD.**

#UD	Invalid Opcode Exception; Occurs when the processor tries to execute an invalid or undefined opcode.

Part 7
Bibliography

[1] Intel-Corporation, “Intel 64 and IA-32 Architectures Software Developer's Manual.” 2024.
[2] O. wiki, “Operating System Development wiki,” 2025.
[3] G. Brunmar, “The world of protected mode,” 2025.
[4] Intel-Corporation, “Intel 286 Programmer's Reference Manual.” 1987.
[5] Intel-Corporation, “Intel 386 Programmer's Reference Manual.” 1986.
[6] T. F. D. Project, “FreeBSD Developers' Handbook.” 2025.
[7] M. Kerrisk, “Linux manual page.” 2024.
[8] A. Shvets, *Dive into Design Patterns*. Refactoring.Guru, 2019.
[9] CPPReference, “C++ reference,” 2024.
[10] I. Free Software Foundation, “GNU C Language Manual,” 2025.
[11] I. Free Software Foundation, “GNU GCC Compiler Manual.” 2005.
[12] F. Cloutier, “x86 and amd64 instruction reference,” 2024.
[13] M. Vorbrodt, “Vorbrodt's C++ Blog,” 2021.

Part 8
Table of Figures

8.1 Listings

8.2 Images

8.3 Tables

8.4 Tools

Purpose	**Tools**
Literature-Research	Internet Search Engines
Inspiration for paragraph structuring	ChatGPT
Coding	Visual Studio Code, CMake, QEMU, Neovim
Documentation	Typst, LTeX+
Project management	GitLab, Jitsi Meet, Teams, Matrix
Dev Ops	Docker, GitLab