**x86 Processor Memory Management**

Table of Contents

[Real-Address Mode 3](#_Toc83227227)

[Protected Mode 5](#_Toc83227228)

[Segmentation and Paging 5](#_Toc83227229)

[Segmentation 6](#_Toc83227230)

[Segment Descriptor Tables 6](#_Toc83227231)

[Flat Segmentation Model 6](#_Toc83227232)

[Multi-Segment Model 7](#_Toc83227233)

[Translating Addresses 8](#_Toc83227234)

[Paging 9](#_Toc83227235)

[Page Faults 9](#_Toc83227236)

[Page Translation 9](#_Toc83227237)

In the **8085 microprocessor**, which was the first one we studied, we faced no issues with memory management. The **address bus** was of **16 bits**, as were each of the **memory locations**. To store addresses, we used the **program counter** and the **stack pointer**, both of which were also **16-bit registers**.

From this, we moved to the **8086** and **80186 microprocessors**, where we faced some issues. The **address bus** and **memory locations** were of **20 bits** now, but the **registers** were still **16 bits**. To deal with this, we introduced the concept of **segment numbers** and **offset numbers**, each of **16 bits**, stored in the **segment register** and the **instruction pointer** respectively.

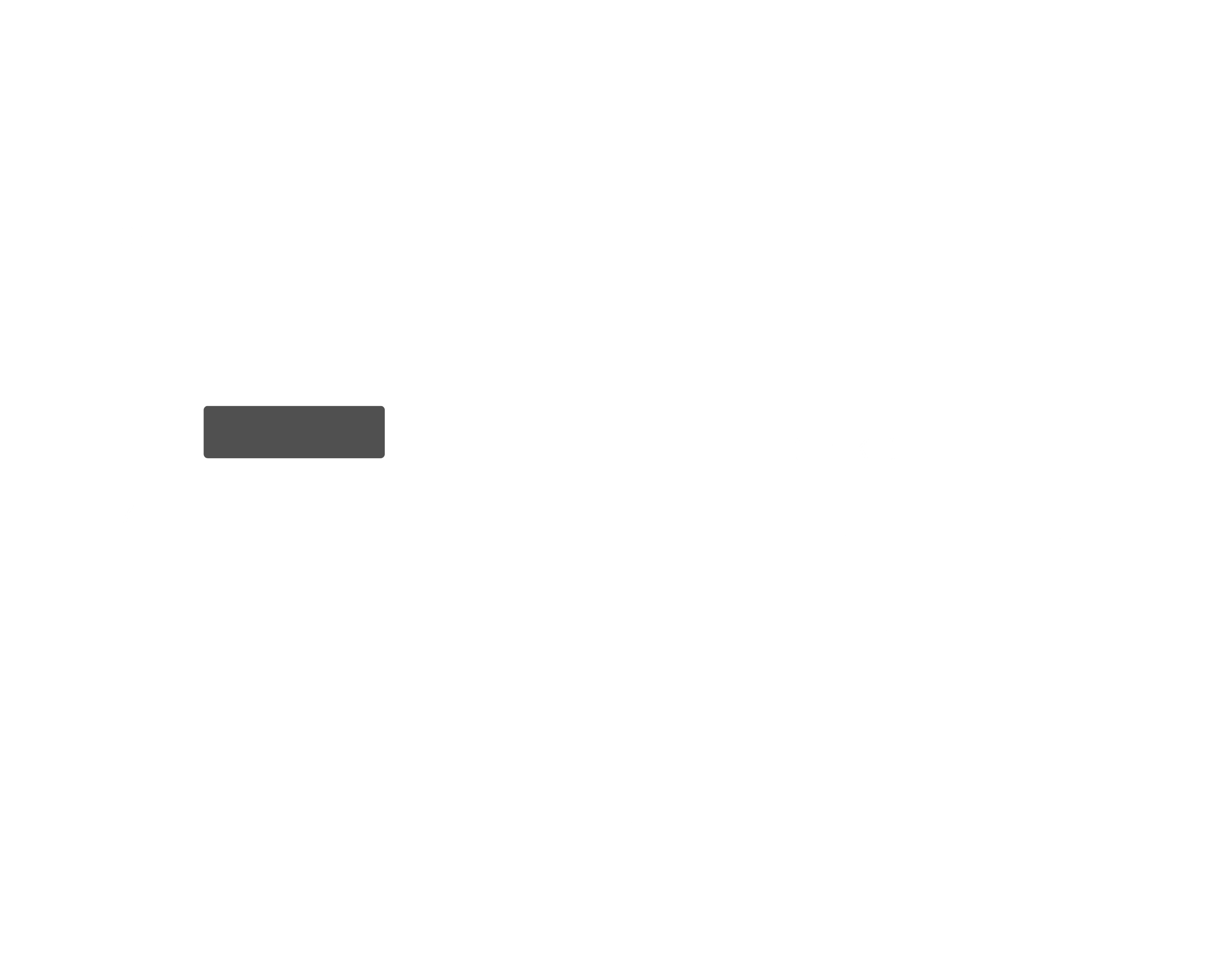
After this, came the **80286 microprocessor**, with a **24-bit address bus**, **80386** and **80486 microprocessors**, with **32-bit address buses**, and finally the **Pentium** microprocessors, with **36-bit** and higher **address buses**. At this point, it became infeasible to have **registers** that were large enough to store these addresses, so we moved to a different memory management process. On top of that, we also needed to deal with the concept of **Virtual Memory**.

## Real-Address Mode

We know that the more advanced x86 microprocessors use the concepts of **real** and **protected addresses modes**. In the **real-address mode**, we use **20 bits** to address memory locations, which gives us access to of RAM.

In real-address mode, all **application programs** can access **any area of memory**. However, we are limited to using **single tasking**, meaning only a single program can run at a time. This means that the code segment, stack segment, data segment and extended segment in memory will belong to a single program, and when a new program is loaded, that program will have its own segments.

In real-address mode, we used **segmented memory**, where a **segment number** and an **offset number** are used to calculate a specific memory location.



Each of the segments used by a program has their own segment, with the addresses of the segments being stored in the **code segment register** (CS), the **data segment register** (DS) and the **stack segment register** (SS).

We also previously saw that the segments may **overlap**, to conserve space, since we do not always require all the space provided by a segment. The overlapping may be partial or complete. The **emu-8086** emulator we have been using uses just **64 KB** of memory, which means that the four segments it uses are all **completely overlapped**, i.e. they all have the same segment number.

## Protected Mode

The **protected addressing mode** was created to support **multitasking**. The entire **4GB** of memory space becomes available to application programs in this mode, with each program being assigned a **memory partition** that is **protected** from other programs. This means that each program has its own code segment, data segment, stack segment and extra segment, and **multiple sets of segments**, one for each program, can exist in the memory space at the **same time**. This is what allows multitasking.

Note that we are considering the **32-bit** address bus from the 80486 microprocessor here, which gives us the 4GB of memory space, but the same concept applies for the 24-bit, 36-bit and higher address buses.

### Segmentation and Paging

In protected mode, we make use of two units, the **Segment Unit** and the **Paging Unit**. The segment unit translates a **logical address** into a **linear address**, whereas the paging unit translates a **linear address** into a **physical address**. Paging is only used when dealing with **virtual memory**. Otherwise, the linear address we get from the segment unit is the same as the physical address.

## Segmentation

### Segment Descriptor Tables

Each program has their own code segment, data segment and stack segment. For each segment, there is a corresponding **descriptor**, which is just a **pointer** to the memory location where the segment is stored. These descriptors are stored in a **segment descriptor table**.

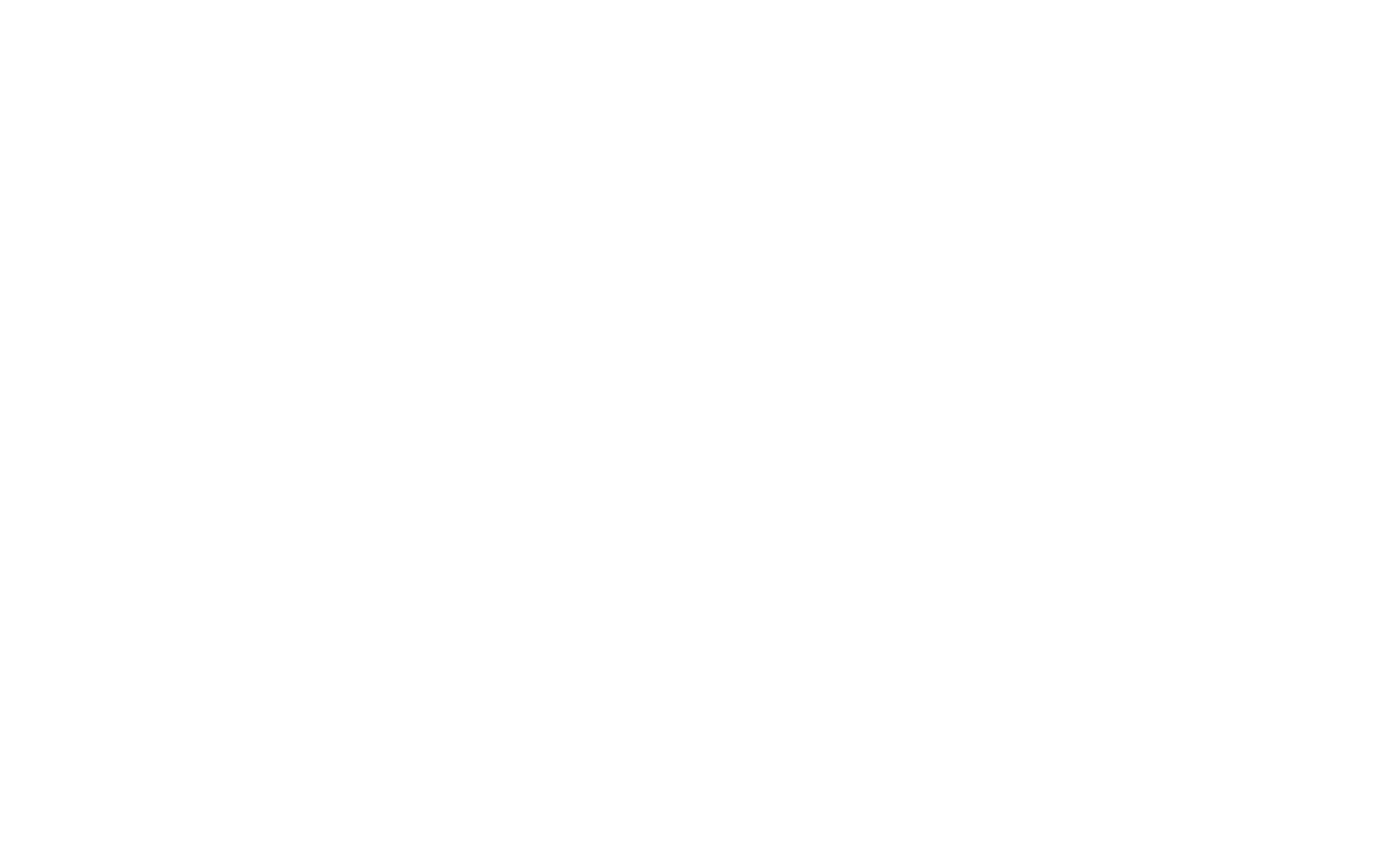
There are two types of **segmentation models**, a flat segmentation model and a multi-segment model.

### Flat Segmentation Model

In the **flat segmentation model**, we use a **Global Descriptor Table** (GDT). For this table, all segments are mapped directly onto the entire **32-bit physical address space**. The segments in the GDT are accessible by **all programs**.

A single **descriptor** has three fields:

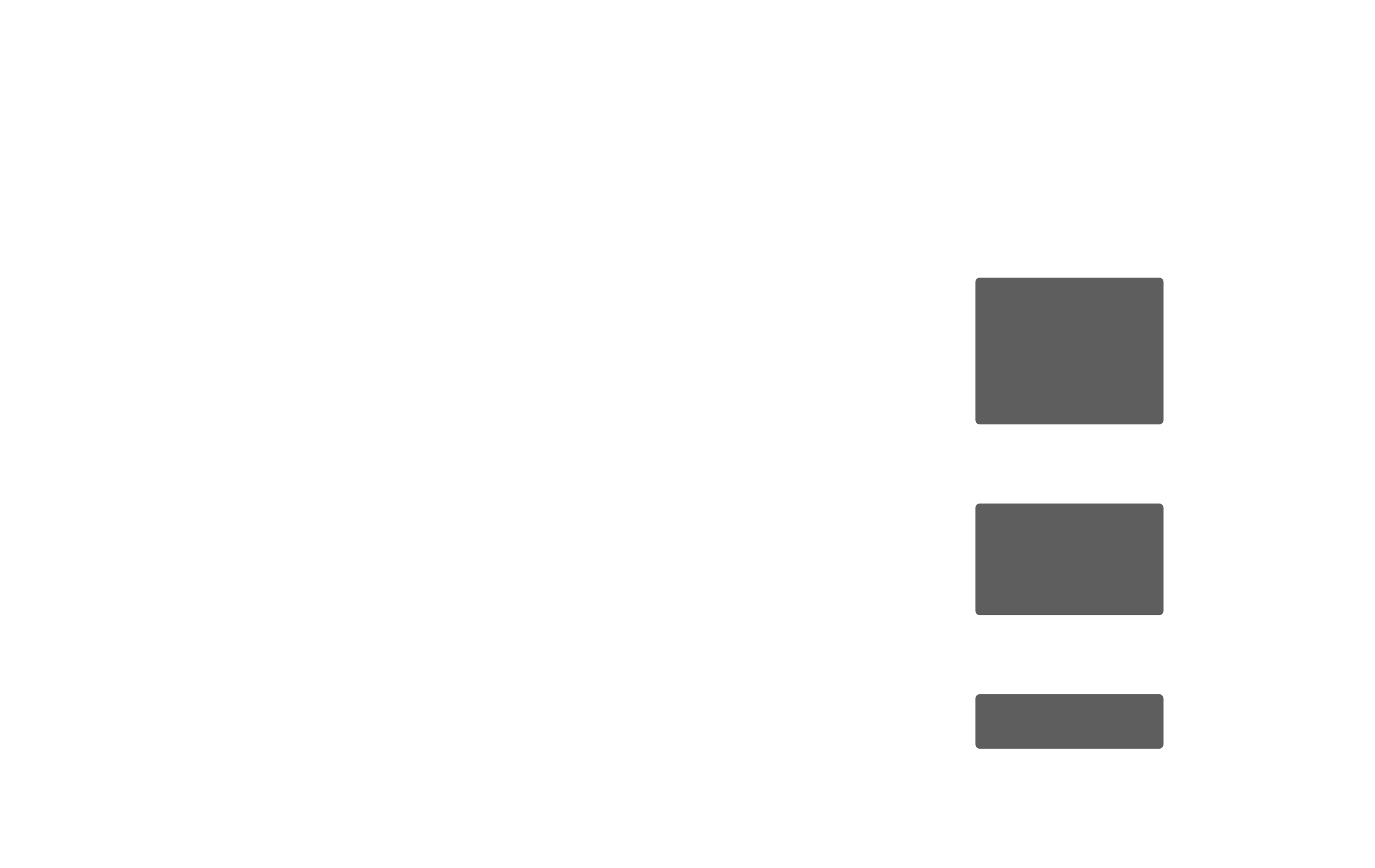
1. **Base Address** - This is a **32-bit** value that denotes the **starting point** of the segment.
2. **Limit** – This is a **16-bit** value that denotes the **end** of the segment. Since it is a 16-bit value, we need to **multiply** the value with 1000h to get the actual address, i.e. we need to add **three 0s** to the right of the value.
3. **Access** – This defines a **privilege level** for memory access.



### Multi-Segment Model

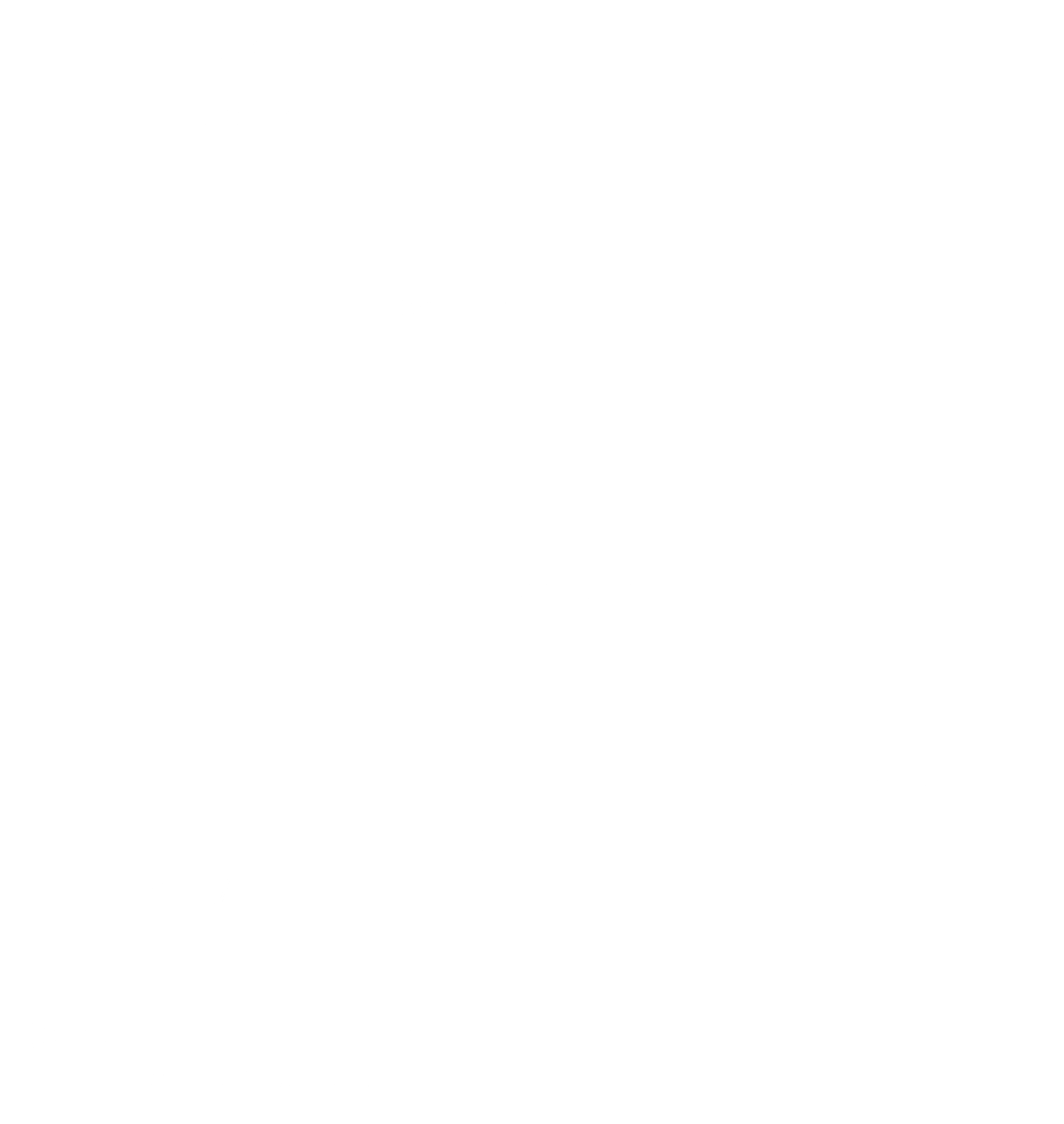
Under the **multi-segment model**, we use the **Local Descriptor Table** (LDT). Each program has its **own LDT**, which holds the descriptors for **each segment** used by the program. Of course, we need to keep track of where the LDT of a particular program is, and this is done using the **LDT register**.

The way the LDT works is the same as the GDT.



### Translating Addresses

Each program stores its data as **logical addresses**. These addresses are stored in **two registers**, as two parts. One register holds the **selector**, which is a pointer to an entry in the descriptor table. The other register holds an **offset**, which is added to the **base address** in the descriptor table entry to create the **linear address**.



Consider that we are dealing with the LDT of a particular program. Thus, each **segment** the program uses will have its own entry in the LDT. Using those entries, we can generate a linear address.

## Paging

By using **virtual memory**, the total space occupied by all programs currently in memory can be **larger** than the available **physical memory**. This is because we will be using the **disk storage** as part of our memory.

When running a program, only a **part of the program** will actively be used at any given time and must be kept in memory. The other parts can be kept on **disk**. The total memory used by a particular program is divided into units of **4096 bytes**, called **pages**. As the program runs, the processor **selectively unloads** inactive pages from memory and loads other pages that are **immediately required**.

### Page Faults

When a required page is not in memory, a **page fault** occurs. The microprocessor interrupts the program while the required page is loaded into memory. The loading and unloading of pages is managed by an operating system utility called the **Virtual Memory Manager** (VMM). The operating system copies the page into memory and the program resumes execution.

### Page Translation

The operating system maintains a **page directory** and a **page table** to assist in paging. The microprocessor uses these to convert **linear addresses** into **physical addresses** in a process called **page translation**.

A **linear address** has three fields, each stored in a **register**.

1. **Directory** – The **Page Directory** is a set of entries which point to different page tables. The **Directory** field tells us which entry from the page directory to look at to locate the required page table. This is a **10-bit** field.
2. **Table** – Each page table is also a set of entries, which point to **page frames**. The **Table** entry tells us which entry from the page table to look at to find to required page frame. This is a **10-bit** field.
3. **Offset** – The page frame contains a set of **physical addresses**. The **Offset** field tells us which entry in the page frame to use. This is a **12-bit** field.

