

Lecture 2: March 6

*Lecturer: Alessandro Pellegrini**Scribe: Anzhelo Xhebraj*

2.1 A20 Line

2.1.1 Story

The original 8086 processor featured a 20-bit address bus capable of addressing 1MB of memory. However, the odd segmentation mechanism, theoretically, allowed addressing slightly more than just 1MB. Indeed all the *logical* addresses ranging from F800:8000 (or equivalently FFFF:0010) to FFFF:FFFF and their *linear* equivalent 0x100000 and 0x10FFEF did not have a corresponding *physical* address.

MEMORY WRAP-AROUND

The workaround for this issue was Memory Wrap-Around: "wrap" that portion of memory making those addresses point to the lower portion of memory by dropping bit 20 (indexing bits from 0 from the least significant). This means that the physical addresses shown above would be mapped to 0x00000 and 0x0FFEF.

Such workaround was then used heavily by many programmers to improve performances of their programs. For example it was used by DOS programmers in order to use the same segment for accessing both program data placed in the higher part of the memory and I/O data that was placed in the lower end.

Introducing the Intel 80286 the addressable memory was increased to 16 MB in protected mode, however the new CPU was supposed to be able to run in real mode all programs developed for the 8086 so that it could run OSes and programs that were not written for protected mode. This produced a bug in the IBM PC AT computer since logical address F800:8000 accessed the physical address 0x100000 instead of wrapping around and access 0x00000. The issue was solved by ANDING the CPU A20 (address line 20) with an output of the keyboard controller (KBC).

A20 LINE FIX

Disconnecting just A20 and not also A21, A22, A23 was enough since real mode software cared only about the area slightly above 1MB.

2.1.2 Enabling A20 Line via keyboard controller

A20 Line can be enabled through various methods but the classical one is through the keyboard controller which at the time was just a single chip (8042) today known as PS/2 Controller.

The PS/2 Controller uses 2 IO ports: port 0x60 for reading/writing data from/to the controller and port 0x64 for reading/writing the status/command register of the controller. Bit 1 of the status register tells the Input buffer status of the controller: 0 if empty, 1 if full. It must be clear before attempting to write some data to port 0x60 or 0x64 therefore we must use the busy waiting scheme to communicate with the controller.

By writing in the command register the value 0xD1 we ask the controller to write the next byte (that will be sent to it in the data register) to the Controller Output Port. 0xDF is the code for enabling the A20 line and 0xDD for disabling it.

Follows the code for enabling A20.

```

...
call wait_for_8042    ; busywait to be able to write the command
movb $0xd1, %al
outb %al, $0x64      ; ask to write
call wait_for_8042
movb $0xdf, %al
outb %al, $0x60      ; enable A20
call wait_for_8042
...

wait_for_8042:        ; busy waiting routine
inb %al, $0x64
testb $2, %al        ; check if can write (Bit 1 has value 1 if full)
jnz wait_for_8042
ret

```

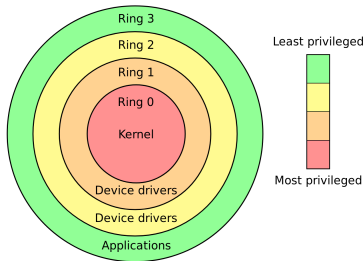


Figure 2.1: Intel CPU privilege levels

2.2 Protected Mode Overview

Protected mode gets its name from the privilege protection mechanisms introduced in the Intel 386 processor to restrict what user-level programs can do employing **Privilege Levels**. There are four privilege levels, numbered 0 (most privileged) to 3 (least privileged), and three main resources being protected: memory, I/O ports, and the ability to execute certain machine instructions. Protected mode was then extended by adding paging. Both paging and protected mode must be enabled on startup. At any given time, an x86 CPU is running in a specific privilege level, which determines what code can and cannot do. These privilege levels are often described as **Protection Rings**, with the innermost ring corresponding to highest privilege. Most modern x86 kernels use only two privilege levels, 0 and 3. Ring 1 has been used in hypervisor systems.

To manage these kind of security mechanisms a new set of registers, namely Control Registers, were introduced. About 15 machine instructions, out of dozens, are restricted by the CPU to ring zero. In protected mode **only privilege level 0** code can read or load the Control Registers. In user space CS is not writable.

CONTROL REGISTER 0 The most important Control Register is CR0 which contains system control flags that control operating mode and states of the processor:

PG Paging (bit 31): Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of register CR0) is not also set; setting the PG flag when the PE flag is clear causes a general-protection exception (#GP)

CD Cache Disable (bit 30): together with NW, when set, fully disable caches

NW Not Write-through (bit 29)

NE Numeric Error (bit 5): enables the reporting of x87 FPU errors

PE Protection Enable (bit 0): Enables protected mode when set; enables real-address mode when clear. This flag does not enable paging directly. It only enables segment-level protection. To enable paging, both the PE and PG flags must be set

The OS may sometimes set PE to 0 to write big chunks of memory to improve performance and then reset it

By switching to protected mode we get that all the instructions that the processor will start executing will be interpreted as 32-bit from now on instead of 16-bit. By this fact two problems might arise: the CS register contains information that had some meaning in real mode but not the same in protected mode and the CPU prefetches the instructions it has to execute in an **Instruction Queue** and interprets it as an *instruction stream* to be more performant but in this case we are switching from 16 bit to 32 bit. In order to make the CPU to work properly we must perform a far **jmp** or **call** instruction to *serialize* the CPU, i.e. flush the Instruction Queue and start reading 32-bit instructions and "re-initialise" the code segment. Afterwards we must also reset Segment Registers since they had the content of 16-bit data/operations.

[Intel()] Ch. 9, Sec. 9.9

2.3 Memory Translation in Protected Mode

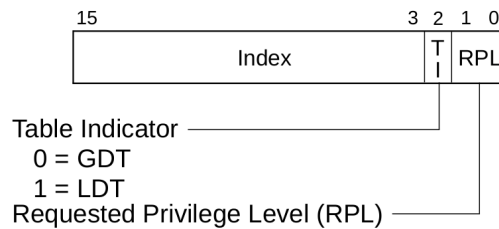
Before entering protected mode, auxiliary data structures must be setup in order for the system to work properly.

One of the main differences between the two modes of operation (real and protected) are the translation from logical addresses to physical ones. In 32-bit protected mode, a segment selector is no longer a raw number, but instead it contains an index pointing to a 64 bit entry in a table of **segment descriptors**.

The segment descriptors are stored in two tables: the **Global Descriptor Table (GDT)** and the **Local Descriptor Table (LDT)**. Each CPU (or core) in a computer contains a register called **GDTR** which stores the linear memory address of the first byte in the GDT (analogously a register for the LDT).

There is one GDT per CPU in multi core systems while there can be many LDTs. LDTs are not really used anymore

To choose a segment, you must load a segment register with a **segment selector** in the following format:



RPL field is called CPL in the case of Code Segment Selector and Stack Segment Selector meaning Current Privilege Level

Figure 2.2: Segment Selector Format [Intel()]

The **Index** part of the selector identifies which Segment Descriptor (and therefore Segment) we are interested in. The **Requested Privilege Level** will be covered later.

The size of the table can be at most 2^{13} descriptor entries

Each Segment Descriptor has the following format

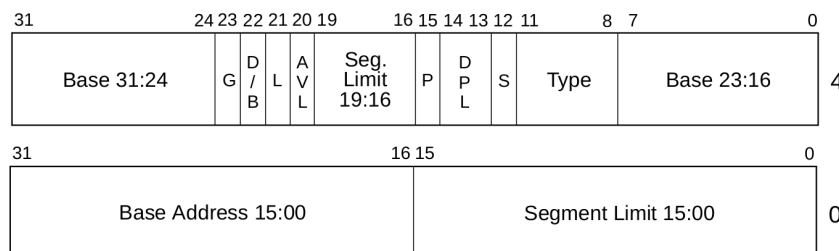


Figure 2.3: Segment Descriptor Entry [Intel()]

The first descriptor entry in the GDT is not used and its value is all 0s ([Intel()] pp. 103)

S (descriptor type flag) Specifies whether the segment descriptor is for a **system** segment (S flag is clear) or a **code** or **data** segment (S flag is set)

Base Defines the location of byte 0 of the segment within the 4-GByte linear address space

Segment Limit Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag

G Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KByte units.

DPL Is the **Descriptor Privilege Level**; it is a number from 0 (most privileged, kernel mode) to 3 (least privileged, user mode) that controls access to the segment as will be shown.

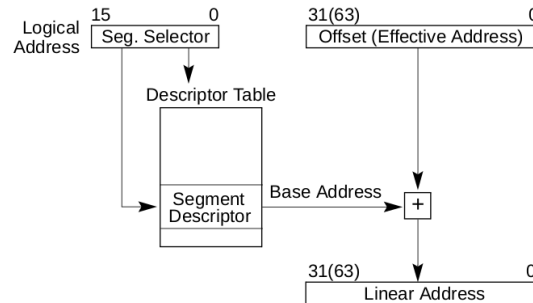


Figure 2.4: Logical to Linear Address [Intel()]

So how is addressing resolved now? Let's look at an example in case of the instruction `jmp 0xDEADBEEF`

1. The processor looks at the value in the Code Segment (CS) Register and checks **TI** to know whether to look in the Global or Local table (Suppose Global in the next steps).
2. Computes the *linear* address $\text{Index} * 8 + \text{GDTR}$ (8 Bytes is the size of a Segment Descriptor Entry) and fetches the **Base** of that Segment Descriptor
3. Computes the linear address that is $\text{Base} + 0xDEADBEEF$

FLAT MODEL Are all those steps really necessary? No. Segmentation has no real advantage in modern x86 kernels due to the fact that all the memory available is addressable through regular registers but since it is not possible to disable the Segmentation Unit, Intel solved the issue by introducing the **Flat Model** which means to create just two descriptors for each of the two privilege levels 0 and 3, one for the code segment and one for the data segment both having **Base** 0 and having as **Limit** the maximum memory available in 32-bit (4GB). This produces a memory access model as if it was linear.

The hidden part is the one used by the processor for fetching the first instruction after a reset

Since a full communication to memory would be too costly in terms of performance a segment register cache has been introduced also known as "descriptor cache" or "shadow

register". The hidden part of the register *is not* invalidated when the entry pointed by that selector changes producing some interesting results. Such — "unattended" feature allowed programmers to address more than 1MB in the so called "Unreal Mode".

2.4 x86 Protection

Due to restricted access to memory and I/O ports, user mode can do almost nothing to the outside world without calling on the kernel. It can't open files, send network packets, print to the screen, or allocate memory. User processes run in a severely limited sandbox set up by the gods of ring zero. That's why it's impossible, by design, for a process to leak memory beyond its existence or leave open files after it exits. All of the data structures that control such things - memory, open files, etc - cannot be touched directly by user code; once a process finishes, the sandbox is torn down by the kernel.

Keep in mind that the CPU privilege level has nothing to do with operating system users. Whether you're root, Administrator, guest, or a regular user, it does not matter. All user code runs in ring 3 and all kernel code runs in ring 0, regardless of the OS user on whose behalf the code operates. Sometimes certain kernel tasks can be pushed to user mode, for example user-mode device drivers.

The whole protection of the system is based on DPL (Descriptor Privilege Level), RPL (Requested Privilege Level), CPL (Current Privilege Level). The first two are stored respectively in the Segment Descriptor and in the Data Segment Selector while the third one is present in Code Segment Selector stored in the Code Segment Register.

The processor uses privilege levels to prevent a program or task operating at a lesser privilege level from accessing a segment with a greater privilege level, except under controlled situations.

Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level.

The CPU protects memory at two crucial points: when a segment selector is loaded and when a page of memory is accessed with a linear address. Protection thus mirrors memory address translation where both segmentation and paging are involved.

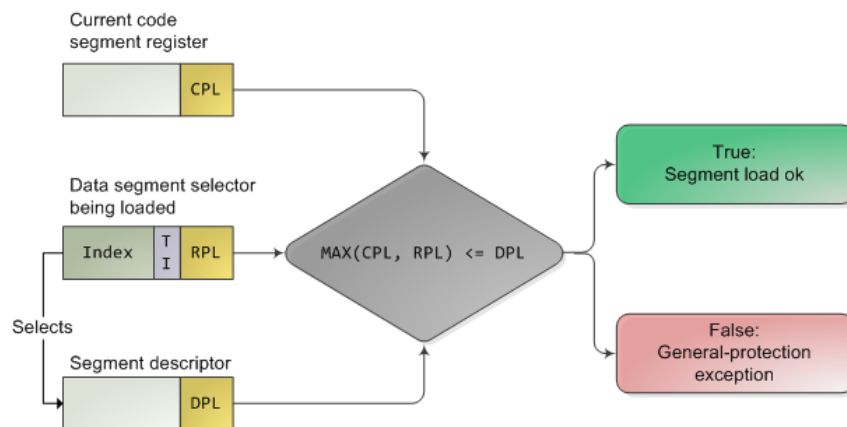


Figure 2.5: Segment Privilege Check [Duarte(2008b)]

In truth, segment protection scarcely matters because modern kernels use a flat address space where the user-mode segments can reach the entire linear address space. Useful

memory protection is done in the paging unit when a linear address is converted into a physical address.

2.5 Switching between privilege levels

Another data structure must be setup before entering Protected Mode: the **Interrupt Descriptor Table**. Such data structure is then used by the OS to handle interrupts and switch between privilege levels.

2.5.1 Interrupt Descriptor Table

In real mode interrupts are handled through the **Interrupt Vector Table (IVT)** which is a table, typically located at 0000:000H, that specifies the addresses of all the 256 *Interrupt Service Routine*. In the case of protected mode the table can be placed anywhere as long the IDTR is set to the right address pointing to its start. The **IDT** stores a collection of system type segment descriptors (S field cleared in the segment descriptor entry) called **Gate Descriptors**.

Interrupts are asynchronous events not related to the CPU execution flow while Traps (or exceptions) are synchronous. Traps are the historical way to demand access to kernel mode

System type descriptors fall in two categories: system-segment descriptors and gate descriptors. System-segment descriptors point to system segments (LDT and **Task-state segment** segments). Gate descriptors are in themselves "gates", which hold pointers to procedure entry points in code segments (**call**, **interrupt**, and **trap** gates) or which hold segment selectors for TSS's (**task** gates). Call gates provide a kernel entry point that can be used with ordinary call and jmp instructions, but they aren't used much so we'll ignore them. Task gates aren't so hot either (in Linux, they are only used in double faults, which are caused by either kernel or hardware problems).

Gate descriptors in the IDT can be **interrupt**, **trap**, or **task gate** descriptors.

Each interrupt is assigned a number **between 0 and 255** called a vector, which the processor uses as an index into the IDT when figuring out which gate descriptor to use when handling the interrupt.

2.5.2 Privilege level switch

To access an interrupt or exception handler, the processor first receives an interrupt vector from internal hardware, an external interrupt controller, or from software. The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

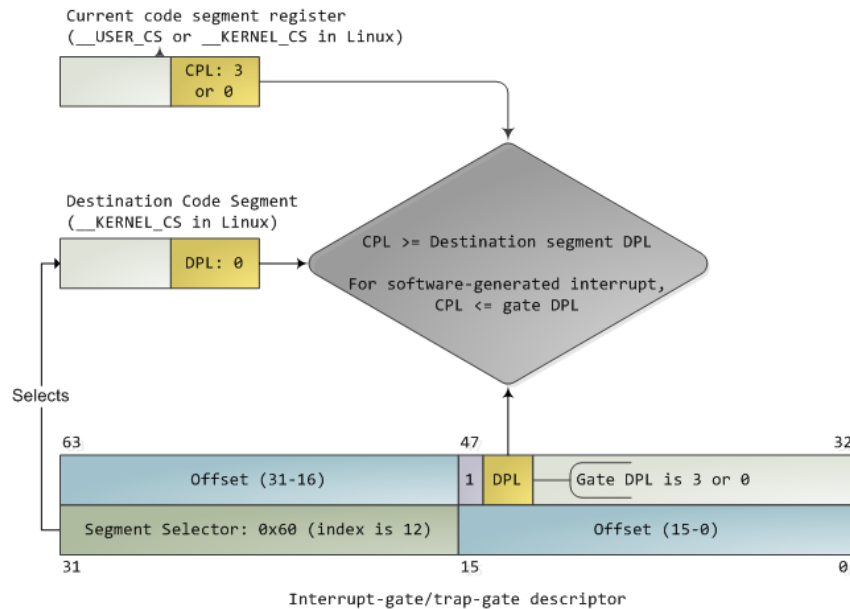


Figure 2.6: Privilege Management on Interrupt/Exception [Duarte(2008b)]

An interrupt can never transfer control from a more-privileged to a less-privileged ring. Privilege must either stay the same (when the kernel itself is interrupted) or be elevated (when user-mode code is interrupted) to make sure that the interrupts are handled only by Privilege Level 0 code. In either case, the resulting CPL will be equal to the DPL of the destination code segment; if the CPL changes, a stack switch also occurs. If an interrupt is triggered by code via an instruction like `int n`, one more check takes place: the gate DPL must be at the same or lower privilege as the CPL. This prevents user code from triggering random interrupts placed in more privileged code segments. If these checks fail a general-protection exception happens. All Linux interrupt handlers end up running in ring zero.

comm

2.5.3 TSS Task State Segment

The Task State Segment (TSS) is a special data structure for x86 processors which holds information about a task. The TSS is primarily suited for hardware multitasking, where each individual process has its own TSS.

The task register holds the 16-bit segment selector, base address, segment limit, and descriptor attributes for the TSS of the current task which can be placed anywhere in memory.

The task's current execution space is defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).

Although a TSS could be created for each task running on the computer, Linux kernel only creates one TSS for each CPU as required by the processor architecture and uses them for all tasks. This approach was selected as it provides easier portability to other architectures (for example, the AMD64 architecture does not support hardware task switches), and improved performance and flexibility. Linux only uses the I/O port permission bitmap and inner stack features of the TSS; the other features are only needed for hardware task switches, which the Linux kernel does not use.

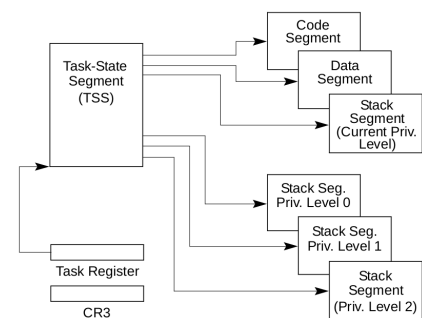


Figure 2.7: Task Management [Intel()]

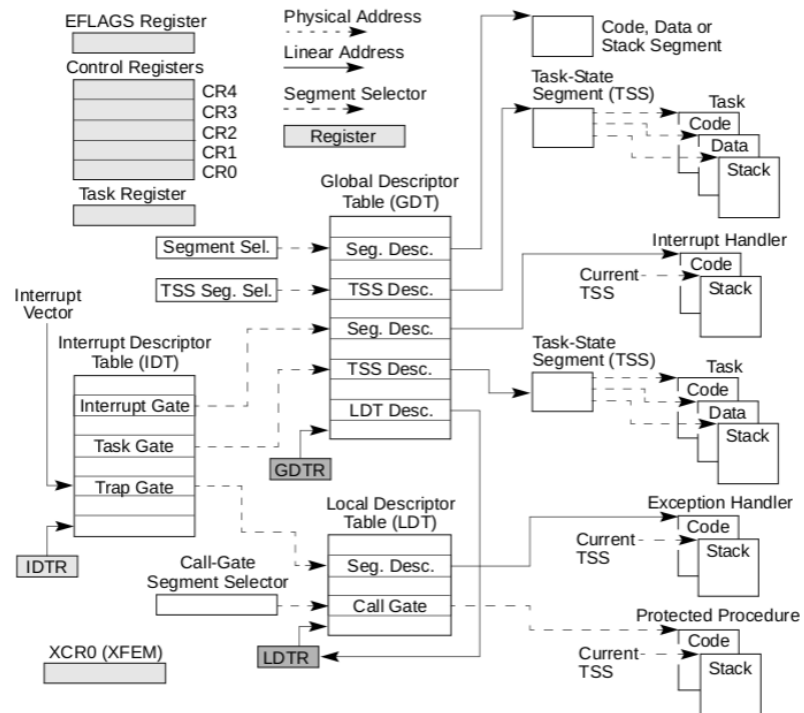
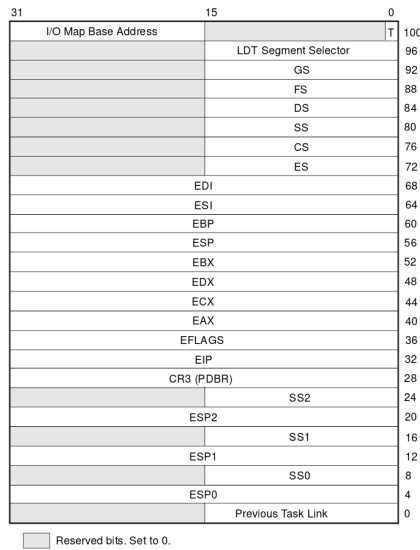


Figure 2.8: TSS [Intel()]

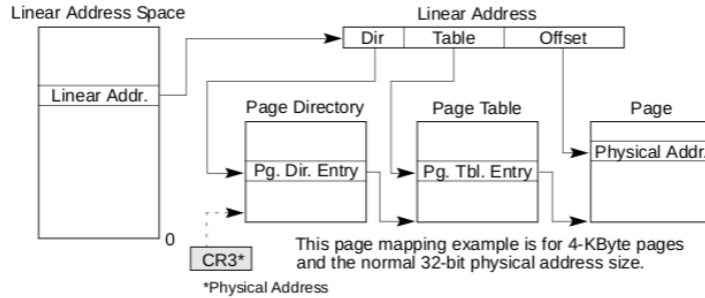


Figure 2.9: Data Structures Overview [Intel()]

References

- [a20()] A20 line. URL https://wiki.osdev.org/A20_Line.
- [car()] Writing a bootloader from scratch. URL <https://www.cs.cmu.edu/~410-s07/p4/p4-boot.pdf>.
- [con()] Context switching. URL https://wiki.osdev.org/Context_Switching.
- [des(a)] Descriptor cache, a. URL https://wiki.osdev.org/Descriptor_Cache.
- [des(b)] Interrupt descriptor table, b. URL https://wiki.osdev.org/Interrupt_Descriptor_Table.
- [osd()] "8042" ps/2 controller. URL https://wiki.osdev.org/8042_PS2_Controller.
- [rma()] The workings of: x86-16/32 realmode addressing. URL https://web.archive.org/web/20130609073242/http://www.osdever.net/tutorials/rm_addressing.php?the_id=50.
- [ser()] Interrupt service routines. URL https://wiki.osdev.org/Interrupt_Service_Routines.
- [tas()] Task state segment. URL https://wiki.osdev.org/Task_State_Segment.
- [vec()] Interrupt vector table. URL https://wiki.osdev.org/Interrupt_Vector_Table.
- [wra()] Who needs the address wraparound, anyway? URL <http://www.os2museum.com/wp/who-needs-the-address-wraparound-anyway/>.
- [x86()] Why doesn't linux use the hardware context switch via the tss? URL <https://stackoverflow.com/questions/2711044/why-doesnt-linux-use-the-hardware-context-switch-via-the-tss>.
- [wik(2017)] Task state segment, Jun 2017. URL https://en.wikipedia.org/wiki/Task_state_segment.
- [wik(2018a)] x86 memory segmentation, Mar 2018a. URL https://en.wikipedia.org/wiki/X86_memory_segmentation#cite_note-Arch-1.
- [wik(2018b)] A20 line, Feb 2018b. URL https://en.wikipedia.org/wiki/A20_line.
- [Bovet and Cesati(2006)] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, 2006.
- [Brouwer()] Andries E. Brouwer. A20 - a pain from the past. URL <https://www.win.tue.nl/~aeb/linux/kbd/A20.html>.
- [Collins(a)] Robert Collins. A20 - reset anomalies, a. URL <http://www.rcollins.org/Productivity/A20Reset.html>.
- [Collins(b)] Robert Collins. The segment descriptor cache, b. URL <http://www.rcollins.org/ddj/Aug98/Aug98.html>.
- [Duarte(2008a)] Gustavo Duarte. Memory translation and segmentation, Aug 2008a. URL <https://manybutfinite.com/post/memory-translation-and-segmentation/>.
- [Duarte(2008b)] Gustavo Duarte. Cpu rings, privilege, and protection, Nov 2008b. URL <https://manybutfinite.com/post/cpu-rings-privilege-and-protection/>.

- [Intel()] Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual*, volume 3A.
- [Oostenrijk(2016)] Alexander van Oostenrijk. Writing your own toy operating system: Jumping to protected mode, Sep 2016. URL <http://www.independent-software.com/writing-your-own-toy-operating-system-jumping-to-protected-mode/>.