3. The structure of the Hypervisor

# 3.1. The Hypervisor Kernel

## 3.1.1. Booting into the Kernel

The hypervisor is built into a binary which is loaded into memory and booted from by using the Pre-boot Execution Environment (PXE) method. This method will deploy the binary through network, the client communicating through the Network Interface Controller (NIC) with the PXE server, and the server, being previously configured to deploy the binary, will send it to the virtual machine, where the Basic Input Output System (BIOS) will load this binary into memory. The PXE booting capability uses the Multi Boot standard, which specifies the boot sequence and will give control to the Hypervisor entry point, found in the binary file of the Hypervisor. Multi Boot Specification needs a header structure to be completed in the binary in order to boot correctly. Thus, a binary which needs to boot through Multi Boot will have to complete in the section *“.boot”* of the binary, at the address given by a variable defined in the binary, named MULTIBOOT\_BASE, from the beginning of the section, the following structure, as given in the documentation of Multi Boot Specification in section 3.1.1:

|  |  |  |
| --- | --- | --- |
| Offset | Type | Field Name |
| 0 | DWORD | Magic |
| 4 | DWORD | Flags |
| 8 | DWORD | Checksum |
| 12 | DWORD | header\_addr |
| 16 | DWORD | load\_addr |
| 20 | DWORD | load\_end\_addr |
| 24 | DWORD | bss\_end\_addr |
| 28 | DWORD | entry\_addr |
| 32 | DWORD | mode\_type |
| 36 | DWORD | Width |
| 40 | DWORD | Height |
| 44 | DWORD | Depth |

The following fields in the Multi Boot Header are optional, depending on setting the 16th bit in the flags field: *header\_addr, load\_addr, load\_end\_addr, bss\_end\_addr, entry\_addr*. The following fields in the Header are optional, depending on setting the 2nd bit in the flags field: *mode\_type, width, height, depth*. The magic, flags and checksum fields are not optional and must be completed properly in order to boot through Multi Boot Standard. The *magic* field must be completed to the fixed hexadecimal value of 0x1BADB002. For the purpose of booting the Hypervisor, the 16th flag will be set, that is because the *load\_addr* field must be completed with the base of the Hypervisor binary, such that the binary will be located in memory at a fixed physical base. The *entry\_addr* field must also be set in order to give the Multi Boot Loader the address of the entry point in the Hypervisor code, so that after executing the Multi Boot booting sequence, the Hypervisor code gets executed directly. The *checksum* field is a measure of verification of the Multi Boot Header, and this must be equal to the negation of the sum of the first two fields, namely *magic* and *flags*.

Prior to giving control to the code pointed by the address present in the header as specified, the Multi Boot Loader will complete a structure with different boot information gathered from the BIOS, such as the RAM memory map and some PCI devices needed in the early stage of booting. When the Multi Boot Loader is done completing this structure and is ready to give the control to the booted code, it will put in the EBX register the pointer to this structure and in the EAX register a value which is called MULTIBOOT\_LOADER\_MAGIC, equal to 0x2BADB002 in hexadecimal, which should be checked by the loaded code in order to verify if the boot was done indeed by the Multi Boot Loader and it was successful.

The Multi Boot Loader will give control to the code by first getting the CPU in 32 bits protected mode, and thus, the code present at the address pointed by the *entry\_addr* must be written in 32 bits assembly code. We will talk in the following subchapters about all the transitions that the CPU can make, as we’ll embark into a journey from 32 bits into 16 bits code, and then back into 32 bits, and finally go into 64 bit mode, where, in fact, the Hypervisor code will execute most of the time.

## 3.1.2. The E820 Memory Map

The physical memory is a system of interconnected addressable semiconductors, the most popular being silicon-based transistors. The physical memory is almost always volatile, that means that it will not store the information after the power is off, thus being “released”, and all the information being lost forever if not being prior saved to a non-volatile storage, such as a flash memory disk or a hard disk. The physical memory (or RAM) can be of two types: dynamic random-access memory, which is the primary storage in a computer system, and static random-access memory, which is comprised of fast CPU caches. The RAM memory is often used for storing information which is used in the current process or in the kernel of the operating system, or to store results of operations, for example, in a variable, as the registers of a CPU have a very limited size. The RAM is also the place where code is executed from, as the CPU will always fetch instruction from the RAM, or from an Instruction Cache for some modern CPUs, so one may conclude that without physical memory, there wouldn’t be anything to execute, so the CPU will not do anything. Thus, one may very easily understand that managing the memory is an essential facility of any operating system.

When powering on a computer system, one of the first “programs” that is run on it will be the BIOS. The BIOS will detect the type of RAM, the number of RAM chips and their size. A good question is: if the BIOS is a “program”, and we have concluded before that all programs are executed from the RAM, how BIOS can be executed if it doesn’t know anything about the physical memory? The answer is that BIOS is a special type of “program”, one that is called in the literature “the firmware code”. This code resides in a special area, comprised of Read Only Memory (ROM), that can be executed from, so the BIOS does not really need any knowledge of RAM in order to execute. But, after the BIOS code is executed and gives control to a Boot Loader, the BIOS will know about the physical memory and different PCI devices present in the system. The BIOS can be interrogated for this information through the 0x15 interrupt, having in EAX register the code of the information that the caller requests. For memory management, the 0xE820 code was chosen, thus the name “E820 Memory Map”.

The 0xE820 function is considered the best way to detect the memory in a computer system, being available in all computers built since 2002, and in most computer before this year. This function will detect all physical memory areas, even those above the 4 Gigabytes limit of 32-bits architectures. All the current operating systems, along with different boot loaders, even the Multi Boot Loader, load the E820 memory map in order to know the boundaries of the physical memory in the early stages. A method to get the E820 memory map will be presented below, and in the next subchapter, the method to interrogate the memory map directly from the BIOS, without a boot loader, will be shown.

The initial kernel code is booted, through Multi Boot Loader, directly into 32 bits. As presented before, Multi Boot Loader will give to the loaded code in the EBX register a pointer to a structure of early boot information needed by all kernels of all operating systems, and also needed by our Hypervisor Kernel. This structure has the following form:

|  |  |  |
| --- | --- | --- |
| Offset | Field name | Bits in the Flags field needed to be set |
| 0 | Flags | Always required |
| 4 | Mem\_lower | Bit 0 |
| 8 | Mem\_upper | Bit 0 |
| 12 | Boot\_device | Bit 1 |
| 16 | Cmdline | Bit 2 |
| 20 | Mods\_count | Bit 3 |
| 24 | Mods addr | Bit 3 |
| 28-40 | Syms | Bit 4 or 5 |
| 44 | Mmap\_length | Bit 6 |
| 48 | Mmap\_addr | Bit 6 |
| … | … | … |

The fields of which we are interested are the following: mmap\_length and mmap\_addr. Thus, we need the 6th bit in the flags field of the header to be set, so that the Multi Boot Loader will complete these fields in the structure given to the booted binary. Mmap\_length defines the number of bytes that the E820 memory map, and the Mmap\_addr field defines the address where the memory map can be accessed by the program. The zone pointed by mmap\_addr defines an array of entries, each describing a zone of memory, with the following structure:

|  |  |  |
| --- | --- | --- |
| Offset | Field name | Description |
| 0 | Base\_address | Defines the base address of the zone of memory |
| 8 | Length | Defines the number of bytes the zone of memory has |
| 16 | Type | The type of memory represented by the current entry |
| 20 | Extended | The ACPI 3.0 extended attributes bitfield |

The number of bytes of each entry is variable, but software will always try to store the physical memory descriptor in 24-byte chunks. The type of the base address and length are QWORDs, so that software may detect memory over 4 Gigabyte boundaries. Type and Extended fields have the type DWORD, but the Extended field may not be returned by the BIOS. Multi Boot Loader will always try to complete this field if it is not returned, so, the size of the mmap\_length field presented in the structure returned will always be a multiple of 24.

The type of physical memory can contain 5 possible types, as following:

* Type 1: defines the usable RAM, this is the physical memory that can be used by software, and is the most general type of physical memory
* Type 2: the reserved or unusable memory – this memory is reserved by some PCI devices and cannot be used by software. Software may read or write into these memory areas to communicate with devices, for example, writing to the video memory will print characters on the screen, even if this memory is considered unusable by taking the E820 Memory Map entry for that memory region.
* Type 3: ACPI reclaimable memory – this is the memory that the ACPI uses at boot time and can be used after the boot phase is done and BIOS is giving the control to the boot-loader.
* Type 4: ACPI NVS memory – this type of memory defines the areas that ACPI uses for Inter-Processor IPIs and other processor-dependent structures, such as the LAPIC and the ACPI tables.
* Type 5: Bad memory – usually this memory areas are defined by errors in the RAM chip, and usually these types of memory will make the operating system indicate to the user that a physical memory problem is possible, and attention is required.

The extended field is used for ACPI 3.0 systems and contains bits that, if are set, will give information about this entry to the caller. Only the first two bits are documented are used, the rest of 30 bits are undefined until now. The first bit (bit 0) of the extended field indicates if the current entry is valid. If the first bit is not set, the current entry should be ignored by the caller, as the current entry is not really part of the E820 memory map. This creates a compatibility problem for most of the operating systems, given that, even if this bit is documented, some older versions of operating systems are not supporting this feature, and will not check the bit, thus possible random physical memory accesses in not valid memory areas can occur. The second bit of the extended field indicates if the current entry is volatile or not. If the bit is set, the memory is non-volatile, thus software should verify the volatility of the memory reported by the E820 memory map and decide if the non-volatile memory is suited for being used as conventional RAM.

The E820 memory map can be also queried directly from the software. As the BIOS code and interrupts run in 16 bits, and software loaded by the Multi Boot Loader is running directly into 32-bits protected mode, the software must first make a transition of the CPU mode into 16 bits. After that, a combination of register values must be set, and an interrupt calling loop must be done to interrogate each entry of the memory map. In the next subchapters, we will discuss how to make a transition from 32 bits to 16 bits, such that the BIOS interrupts can be called, and we will discuss in detail how the memory map should be interrogated by the software.

## 3.1.3. CPU Transitions