x86 INITIAL BOOT SEQUENCE - (11-03-2020)

The first thing to keep in mind is: Intel tried always to be backward compatible. If they changed smtg, all the software layer would have to be recompiled. The SO is a middle-ware between the hardware and User-Space applications. So while the SO is starting, it has to take control and setup step by step all the hardware and then offer services to users. The SO, in its early stages, can't manage all the actual hardware, but it must rely on services offered by the same hardware in order to drive setup of all hardware devices.

BIOS/UEFI

In the very beginning of the boot sequence, the control is taken by the BIOS. It tries to initialize few necessary services. They will useful in order to load at the end of the sequence the SO image in main memory.
The first thing that an user do is to push the power button. We can't image that the CPU will start to execute some instructions cause CPU can't interact properly with the memory. We have to setup the right voltages in order to let hardware work fine. These voltages have to be provided in a certain order depending on the current machine (x86: 1.15, 3.3, 5, 12 V). So the first thing that the machine does is Power Sequencing: thanks to certain switches (implemented through transistors) that absorb the energy provided by the supply, a certain part of the circuit is enable and then a certain voltage is setup. This mechanism is repeated for all the hardware in order to setup all the required voltages in a clear order). Then we have to setup the clocks. In our motherboards we have several different clock sources. Note that clock sources are less than the different clocks that our hardware uses. We can derive all the clocks thanks to a certain kind of circuit called Phase-Locked Loop Circuit. This process requires some time. CPU is enable with a "Reset Signal" ONLY since the clocks and the voltages of the hardware are correctly setup.

At this point we have the system in a very "basic" state: all the elements are disabled like the Cache system, MMU and others. We have nothing to execute in RAM and most of all we are executing in the REAL MODE (that is the one 8086-compatible).

As we know the MMU is disabled, but we are running using SEGMENTED MEMORY. Basically, this view creates a segmented view overall the available memory. We must access memory via <seg.id + offset> = logical address.
All the assembly instructions refers to LOGICAL ADDRESS not to PHYSICAL. At the time, the 8086 had only 4 16-bit segment registers, each for a different purpose: CS (Code Segment) should keep asm instructions, DS (Data Segment) should contain actual data used by application, SS (Stack Segment) should maintain the stack of application, ES (Extra Segment), provided to programmers to be used. Then in the 80s have been added two more segment registers: FS and GS with no predefined usage, but today they have (FS is used to map TLS, GS is used to map per-CPU variables). How does the translation work? The logical address is taken by the Segmentation Unit and translates it in a physical address (NB this is actually a physical address because we are running in Real Mode, in Protected Mode there will be the Page Table that will translate a linear address into a physical one). Segmentation is still present in the machines in order to be backward compatible but it is used in the flat mode. Each instruction modifies implicitly a segment register: f.e. "jump uses CS, push uses SS. Most Segment Registers can be loaded using MOV instruction. BUT CS can be loaded only with a LONG JMP or LONG CALL.

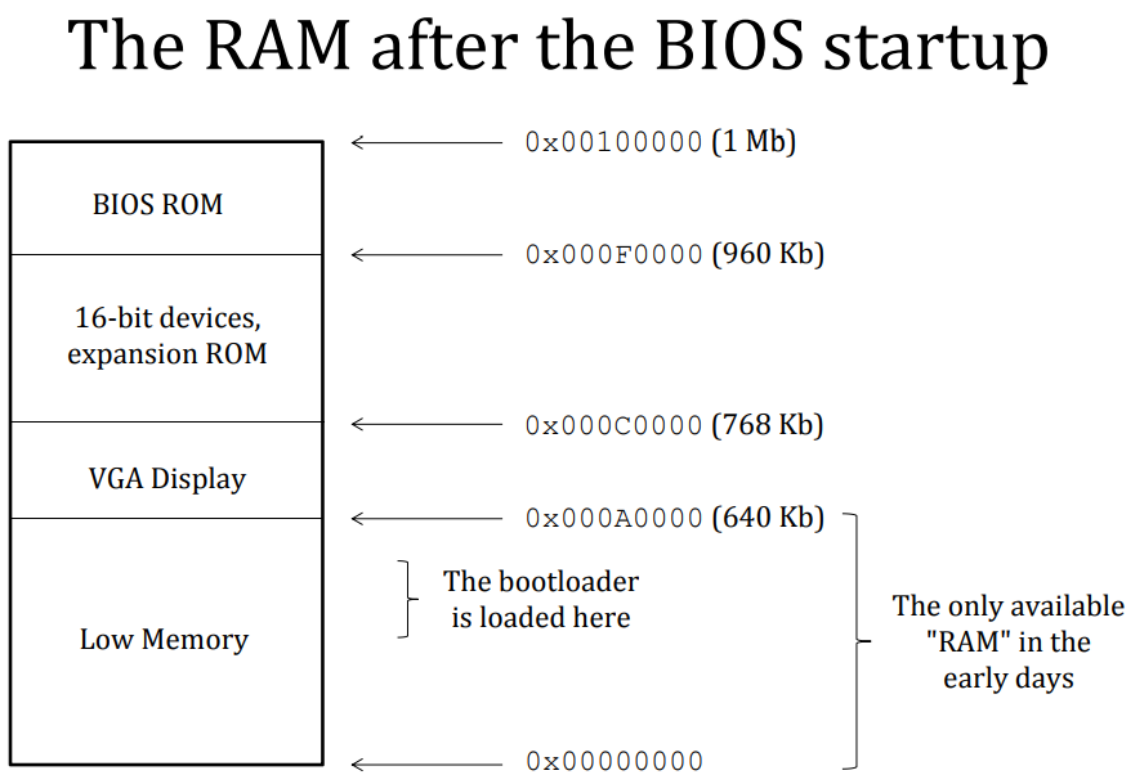
In Real Mode we are running 16-bit instructions and we have a total of 1MB addressable memory cause we are using a 20-bit segmented memory address space. In the register there are the 16-bits higher part of the address.

F.E. JMP 0x6025 -> VALUE_OF(CS) * 16 + OFFSET -> 0x1000 << 4 + 0x6025 = 0x16025 that is the physical address.

With a 16-bits segment register and offset we can reach a max FFFF:FFFF address -> But it is larger 1MB, we have to use segments in a certain way in order to avoid overflow. Notice that Segments can overlap.
The FIRST FETCHED ADDRESS is F000:FFF0. This address is translated into FFFF0, that is only 4 Bytes before the maximum available address in the early memories (1MB = FFFFF). This is known as the reset vector. That portion of memory is mapped to a ROM: the BIOS. The last instruction that the CPU executes is a "ljmp $0xf000, $0xe05b". A LJMP instruction changes the CS register set it to the 0xf000 value plus an offset depending on the position of the first instruction of the BIOS. If the BIOS grows the only value that must be changed is the offset of the long jump! So the control is taken by the BIOS.

The first thing that the BIOS does is to search about available video adapters in order to load in memory their own routines (positioned in C000:0000 and C780:0000) in order to be able to execute few simple graphic operations like the write on screen. Then the BIOS will initialize all the found devices and test their functioning. It also could perform some RAM Consistency check. Now on, it is in charge to load the actual SO image.

The BIOS will look to the boot order table stored in the MotherBoard in order to know where load the SO. But different devices use different file systems! BIOS is not so complicated. There is an agreement: the BIOS will read the very first block of the disk and put it at 0000:7c00. It is called Stage 1 Bootloader. In order to be compliant to these agreements, our bootloader has to map the very first instruction to that address! Because BIOS will give it control through a ljmp $0x0000, $0x7c00 instruction.



The only available RAM (at the time) is from 0x00000000 to 0x000A0000. A tiny amount of memory for the actual size of an SO image.

Stage 1 Bootloader

The memory is so small that this phase can't load directly all the SO image in memory. It has to perform some routines in order to do it. We loaded the first block of the device into memory. That first sector is so called Master Boot Record (MBR). It keeps an executable code (Stage 1 Bootloader) and a 4-entry partition table that identify different device partitions (in terms of positioning/offset). In case we need more than 4 partitions we can create an extended partition entry in the MBR: we can reference a partition in which is kept an additional partitions table, just like a linked list with multiple levels. An MBR is a 512 Bytes block (from 436 to 446 Bytes of loading code). At the beginning of the MBR (such as for Microsoft) there are a lot of informations about the actual device and the way that it uses to store data. But since the BIOS will jump into the very first instruction of the MBR, how can we solve this problem? The first bytes of the MBR represent a dummy entry which will jump to the real first instruction of the loading code and it will skip all the DATA Bytes.

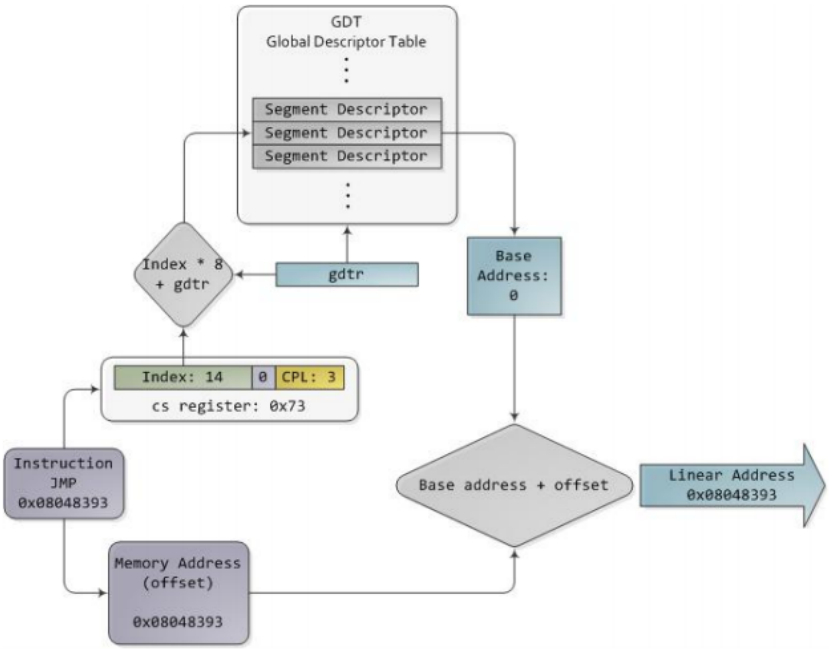
After this jump we can start to execute the very first instructions of the MBR. We have to first of all disable the interrupts. We can't serve them cause we didn't already setup all the interrupt vector table (IVT)/ interrupt descriptor table (IDT). Then we set to zero all the segment registers via MOV instructions cause since the segmentation is still supported in the actual machines we are using a flat mode, in which only offset is used. But how can we set the CS register? Only with a LJMP! But it is already set to zero by the BIOS, so let's move on.

The next pass is to enable A20 in order to be able to map more than 1MB of memory. While the max mapped memory was 1MB, programmers exploit this limitation in order to increase speed up of their programs: since the user programs were mapped to the highest part of physical memory and the OS to the lowest (it contains the syscalls code), every time they wanted to run a syscall they overflowed the address space in order to simulate a sort of rounded memory space. This was cause change the CS register was too costly in their mind. This was a big problem when Intel, generally machines, introduced a larger memory (in the first step up to 16 MB) cause the previous programs would crashed cause the overflow trick would failed. So Intel thought about a simple solution: the A20 line has to be forced to zero! In that case automatically was achieved the backward compatibility. A bootloader programmer has to keep in mind this situation and then he must enable A20 line manually. At the time add a new transistor in the CPU was considered too costly so, since that, it was used the Keyboard Controller in order to manage a control bit for enabling/disabling the A20 Line. Of course a keyboard is a in/out device so the request is made in busy waiting. The Port 0x64 is used to communicate with the controller and 0xd1 means write smtg. Then it is used 0xdd or 0xdf to enable/disable A20 through port 0x60. So now on we can use 16 MB of memory!

But just relying on Real Mode is not a secure procedure! Cause the user programs can easily change CS value simply using a long jump instruction. This was so dangerous at the time: there was no security checks between user space and kernel space. So it was introduced Protected Mode. For backward compatibility CPU starts in Real Mode, so Stage 1 Bootloader has to enable this new mode (again) manually. Moreover it was added the Paging mechanism, disabled by default too. Today in the modern CPUs we have a lot of available registers: starting from that are for general purpose, the float in point ones, the vector ones, but most important the C class and the D class (resp. control and debug ones).
We are interested in CR0: it is a set of several flag such as Protected Mode Enable (PE Bit 0), Write protect (WP Bit 16 global disable write op), Paging (PG 31 Bit it uses CR3 register). The first thing to do in order to enable Protected Mode is to flip Bit 0 of CR0. The flip has to be made via a 16-bits instruction, but the next instruction has to be a 32/64 bits instruction. It is not enough to flip the Bit 0 in CR0 cause the real mode change will be made after the CS register will be updated via a long jump op. This is cause enabling Protected Mode will change totally the meaning of segmentation. So it can be done in this way: LJMP 0x0000, PE_mode where PE_mode is a label to which follows only 32/64 bits code. In protected mode the segment register is no longer a raw

number representing the higher part of the memory address. It is instead an index into a table of segment DESCRIPTORS. There are 3 kinds of descriptor: code, data, system. Each entry in the Description Table is composed by 8 Bytes. In these bytes we have again the base address of the beginning of the segment (split into different part, it is not a continuous value for performance reasons), we have a Limit value that represents the size of the segment, we have a G bit (granularity) that tells the unity measure of the memory mapped with that segment, it means that if it is set the size of the segment is express through multiple of the size of a page, then we have a DPL 2 bit wise field, it is a Descriptor Privilege Level, it tells in which execution mode u can access that segment (we have introduced a security check on memory). Intel introduced the Ring Model to be able to differentiate between kernel and user application. We have 4 rings: ring 3 is the less privileged level, ring 0 is sudo :D, is assigned to the kernel. Notice that ring 1 and ring 2 are not actually used, it is sufficient using 2 levels mode. Basically we have 2 tables: a Global Descriptor Table (GDT) and a Local Descriptor Table (LDT). These 2 tables are kept in memory. Their addresses are kept in 2 physical register (GDTR and LDTR). The GDT is a system-wide table, the LDT was thought to be a per-process scheme but it is not used anymore. We said that in Segment Registers there are no longer addresses but indexes in the tables that we have described before (last 13 bits), there is another bit Table Index that is set to 0 for GDT and 1 for LDT, and 2 bits described the RPL field (Requested Privileged Level it is used for security check that we will see later).

Segmented Addressing Resolution



Every time we have to translate an address generated by a asm instruction we have to perform all these steps: first of all we check the CS Register in which is contained the index of the GDT/IDT table, then we multiply it with 8 and we add the base address of the GDT/LDT table kept in the corresponding register GDTR/LDTR. Then we check some security constraints and then we add 0 to the offset specified by the asm instruction. The Base Address is always 0 because we are using the flat mode. Segmentation can't be disable! Then we obtain a Linear Address that has to be translated into a physical address via Page Table. This operation is too costly, we have to setup a cache system for segment. In the segment register there is another part near the one that we have already discussed that is not programmable. In that part is copied the value of the selector previously read, in order to cache it to improve performance. But when the GDT is altered we have to invalidate the cached selector in the selector register. How can we do it? We have to

rewrite the same content in the cs register in order to notify the firmware that we have to read a new value (or an updated one) by the GDT/LDT table.

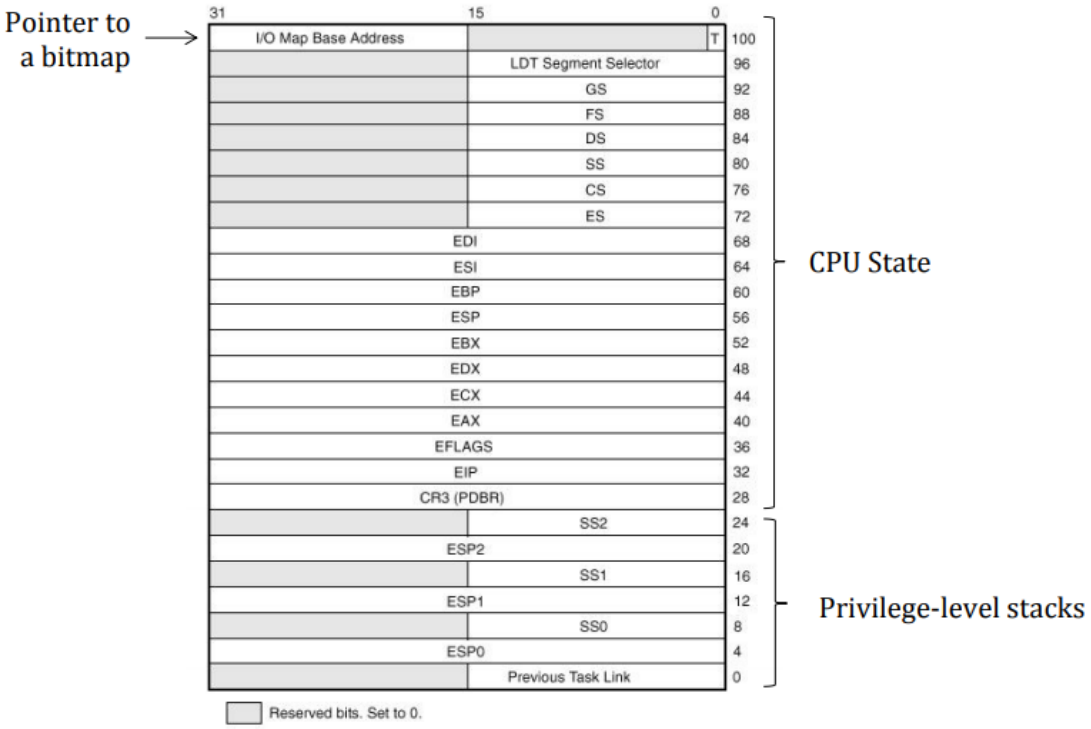
How can we use all these things in order to enforce protection? A descriptor has a DPL field. The firmware must check if an access to a certain segment is allowed. There must be a way to change current privilege: passing to a lower privilege level is always allowed, the contrary (ex. from Ring 3 to Ring 0) must be controlled and in case denied. The Data Segments have the RPL (Request Privilege Level) instead the Code Segments have the Current Privilege Level. Remember that Data Segment can be changed via MOV op, the Code Segment only via LJMP/CALL op. Notice that each Descriptors has the DPL (Descriptor Privilege Level). For loading data the check is made in this way: MAX(CPL, RPL) <= DPL. Ex. If we are an User CPL = 3 and we want to load some Data Segment with a RPL equal to 0, and the Descriptor has a DPL equal to 0, we will generate a General Protection Exception. (Ask for clarifications).
For the execution of code things are quite different. How can we change the CPL value? In order to do it, we have to use GATE (specific entry point). It is associated with a Gate Descriptors and it allows to make some controlled privileges escalation. Ex. i am an user routine that want to call a kernel routine, i must pass through the gate otherwise i will receive a general protection fault. Gate descriptor is a segment descriptor of type SYSTEM. System Descriptors are: Call-gate D, Interrupt-Gate D (HW Interrupts), Trap-gate D (SW Interrupts), Task-gate D. These descriptors are referenced by an Interrupt Descriptor Table (IDT) pointed by the IDTR register. In the Real Mode instead there is the IVT, a table that is only 256 entry wise. Which point directly to the routine address. In protected mode to enforce security it is implemented an IDT, a table of 256 entries again in which each entry contains a selector to a System Descriptor and an offset. The system Descriptor will give us the right Base Address for the CS Register from which the Kernel code starts, the offset will give us the right start of the requested routine. With this mechanism the Interrupts can be handled with a privilege level equal to 0 cause we passed through a gate! There is a verbose copy of the GDT in Linux for each core. There are 4 really important entries: Kernel Code, Kernel Data, User Code, User Data. These Descriptors are used always it is needed to make some context change.

x86 INITIAL BOOT SEQUENCE - (13-03-2020)

Stage 1 Bootloader

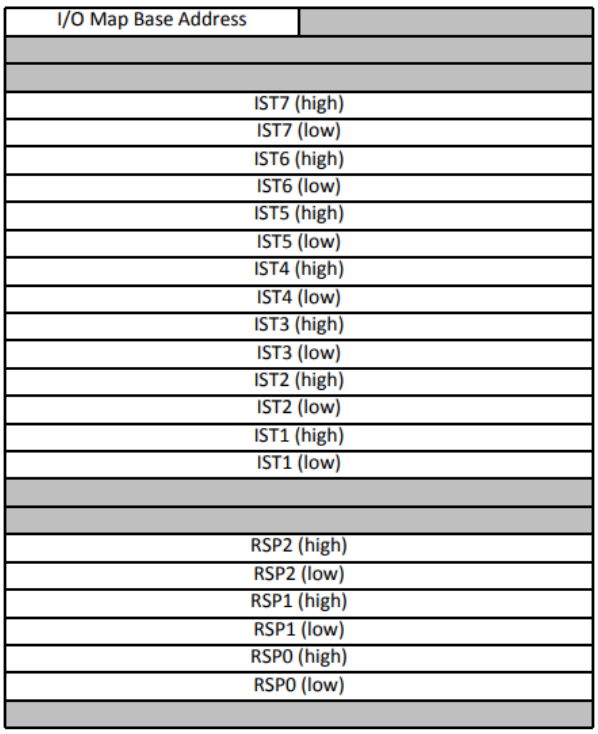
Continuing on GDT. We already said that there is a different GDT for each core. In a GDT there are 3 descriptors called TLS (Thread local storage). We have 2 segments for kernel space: one for kernel code and one for kernel data. The same for user space: one desc for user code, one for user data. There are a lot of BIOS descriptors. The SO can't directly interact with devices, it relies on BIOS to do it. There is an entry that allows to target the LDT. Intel thought that the LDT could implement some kind of multitasking. But it has a lot of limitations and in order to make happy the CPU (otherwise it cannot do nothing) the Linux Kernel setup one portion of memory shared among the cores, a sort of dummy LDT. Then there is a TSS Descriptor. The idea behind the Task State Segment was that every time we are running a Task, it is associated to a Task Segment that describes it. If a new Task is scheduled the TSS has to be changed. But this idea was abandoned. But it is still fundamental to implement context switch between user and kernel space. TSS stores processor registers state, I/O Port Permissions, Inner-level Stack Pointers (different stacks for complete isolation between user app and kernel), Previous TSS Link. It could be everywhere in memory, so we have a GDT entry. At startup Linux Kernel has a int_tss array and each core will use one entry of it to populate its TSS's GDT entry. The Task Register (TR) allows to reach easily the TSS Descriptor and so to perform context switch.

i386 Task State Segment (TSS)



In the TSS (32 bits) we have a lot of interesting fields. First of all the I/O Bitmap Base Address in which are contained all the accessible devices from the Task. Then we have a snapshot of the 8 general purpose register plus the flag register and the instruction pointer. Also we have a snapshot of CR3 register used for paging. We have a snapshot also of all the related segments. And finally we have also Stack Segment and Stack Pointer for each ring. In this way the firmware ensures that every time there is a context switch the stack pointer is correctly setup. The firmware avoids malicious accesses to kernel stack form an user application. A TSS Descriptor has the BASE field set to n-th entry of the int_tss array. Granularity is set to 0 and Limit is set to 0xeb (TSS must be of 236 Bytes). DPL is set to 0 cause TSS cannot be accessed in user mode. How about to 64 bits version?

TSS on x64



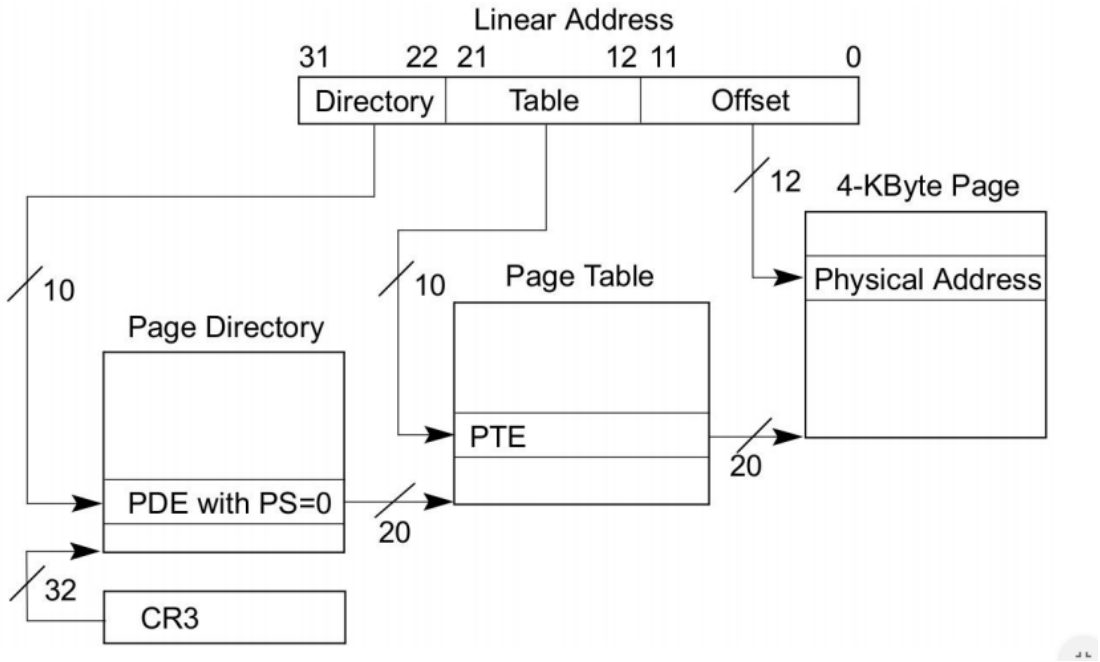
- Registers are gone.
- The Interrupt Stack Table (IST) identifies 7 stack pointers to handle interrupts
- Entries in the IDT are modified to allow picking one of these stacks
- Value 0 tells the firmware not to use the IST mechanism

Now registers are gone. For registers are used different and internal data structures of the OS. That part is replaced with 7 interrupt stack pointers in order to be able to handle interrupts of course. In some case Linux wants to service completely separated different routines. In 64 bits mode the IDT is quite different in order to specify what IST is required. In case the value is set to 0 the routine will use the kernel context. The stack pointers are still available. But where is Ring 3 stack? It will be pushed on the stack! On Kernel stack we have information about source stack. With the special instruction IRET we can set back the User context by POP operation.

How can we entering Ring 0 from Ring 3? We use the GDTR and IDTR scheme. But what about security checks? The idea behind the checks is that we can only change the CS content only if we have to serve an interrupt. For all the hw interrupts the firmware will check that our current CPL is >= to the DPL related to the routine that must be activated. This is because it has to be avoided a passage to a lower privilege level to serve an interrupt. BUT for all sw interrupts (manually generated) there is another check: the firmware will check that our current CPL is less or equal to the GATE DPL. For example, if we want to execute a syscall (sw interrupt within 0x80 entry of IDT) we have to generate a sw interrupt through INT $0x80. In this case our CPL is equal to 3 (user application) and the syscall gate has a DPL set to 3! There are only a subset of entries in the IDT with a DPL equal to 3 associated. This means that just few kernel routine could be accessible to the user. But there are only 256 entries in the IDT so we can't put an entry for each trap. So there are just few gates or maybe one in order to pass from user space to kernel space and thanks to the kernel API we can specify later what kind of service is requesting.

Now move on and talk about Paging though to enhance security. Now a logical address is translated to a Linear one and the thanks to Paging Unit it is translated to a physical one. Paging has to be manually enabled after Protected Mode is activated. Several data structures must be setup in order to use correctly it like the Page Table. Now thanks to Paging, we can implement some security constraints to small chunk of memory as we will see.

i386 Paging Scheme



This is a simplify scheme of Paging. Today the scheme is a little more complicated but it still works in this way. The CR3 is the Control Register that we discussed before. It contains the base address of the most external table of the Paging schema, in this case Page Directory Table. From the Segmentation Unit we receive a Linear Address in which we can identify 3 different offsets. From CR3 content we can easily find the base address of the Page Directory and then thanks to the 10 most significant bits of the Linear Address received we can identify an entry in the table. This entry is called Page Directory Entry (PDE) and after some security check, 20 bits of it identifies the base address of the corresponding Page Table (a middle table in the scheme). Thanks to the 10 bits in the middle of the Linear Address we can find a Page Table Entry (PTE). This page table identifies the base address of a 4 KBytes page of physical memory, called frame of memory. The final portion of memory that we want to access is identified by the remaining 12 bits of the Linear Address, used like offset. This is a solution for 32 bits architecture. Each table fits a 4 KBytes page. So we have 1024 entries in the first and the second table. Overall we have a 1K x 1K x 4KB = 4GB of memory that could be mapped to a virtual address! Moreover, cause CR3 register and all the pointers contained in the tables of the scheme are physical addresses, there must be a entity that BEFORE using the Paging scheme, must setup a lot of physical addresses! The SO does it at startup. There is the possibility to have less amount of physical memory with a 4GB virtual address space. The problem is up to the SO :D.

A PDE entry is composed in this way --> 20 bits for a PTE pointer. There is a bit called User/Supervisor that means if the memory pointed is associated to the User or to Kernel (Intel understood that only 2 rings are necessary). There is a Present bit that identifies the validity of the pointed portion of memory (in fact we can have only a partial mapped memory in the virtual space of a process, ex. when during the translation of a linear address we find a Present bit equal to 0, it is activated a trap routine of the Kernel in which if the application is doing smtg wrong, SO will crash it ---> Meltdown lol), in case of less physical memory than the available virtual one, the SO has to check, before to send the crash signal, if that requested address was swapped to disk, cause swapping a page means write to disk some not actually used memory in order to free space for the running user application, the Page fault mechanism is really complex. There is a bit called Read/Write that specifies for range memory the permissions. There is an Accessed bit which is a sticky bit that tells if the system has navigated this memory reference or not, it is so

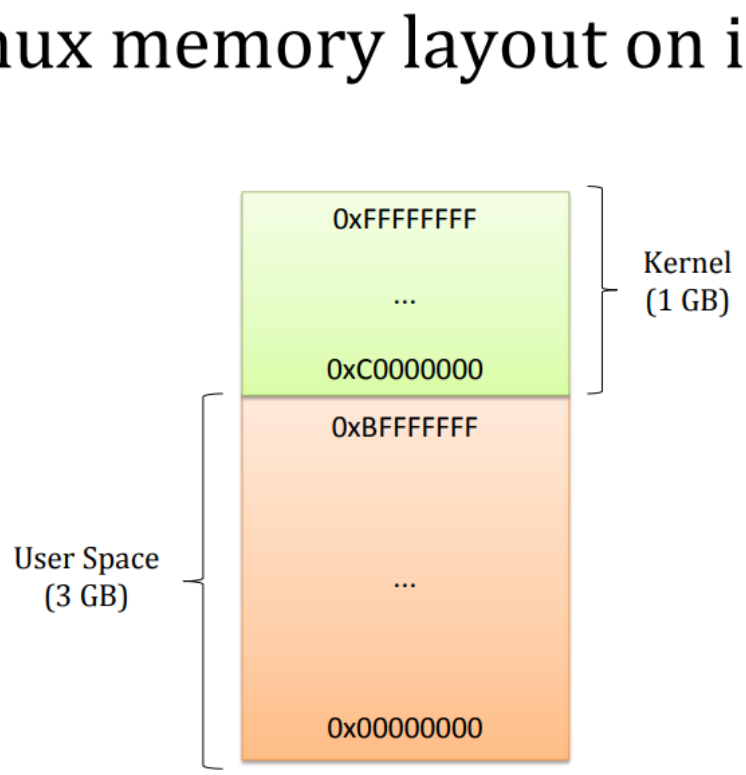
useful to reduce the overhead every time we have to change the virtual to physical mapping. There is another important bi called Page Size which means if it is set to 0 that it is used a 4KB page, otherwise it is used a so called Huge Page. In fact if the PS bit is set to 0 we have to pass through the second level of the scheme, the Page Table, otherwise if PS bit is set to 1 it means that we have to skip it and reference directly an huge page, using as offset inside it the remaining 22 bits (the size of the huge page is 2MB)!

A PTE entry is composed in this way --> 20 bits for a 4KB Page pointer. There is always a Present bit, the User/Supervisor bit, the Accessed bit. Moreover there is the Dirty bit that means if a write operation on the destination pointed has been performed. There is still the Read/Write bit which as we will see it is used for the Copy-On-Write method.

Now, the scenario is quite complex: every time we generate a logical address, it mus pass through the Segmentation Unit AND the Paging Unit! I don't want to make that translation every time. I want to find out a cache system of addresses. If segments don't change and the page entries don't change, the virtual address would be translated in the same physical address. So there is the Translation Lookaside Buffer (TLB) implemented in the CPU (modern CPU has a 2-level implementation). Every time the CPU receives a virtual address will search in the TLB if the translation was already been done before. The virtual address is split in 2 part: the page number and the offset. The page number is used to access the TLB table, if there is an hit we can retrieve immediately the physical page number, multiply it for 4KB (in order to retrieve the physical base address) and then we can add the offset. Otherwise we have to translate that address through Paging Unit. Notice that we can talk about page number because the giant memory is divided in blocks, each one of the same size (theoretically).

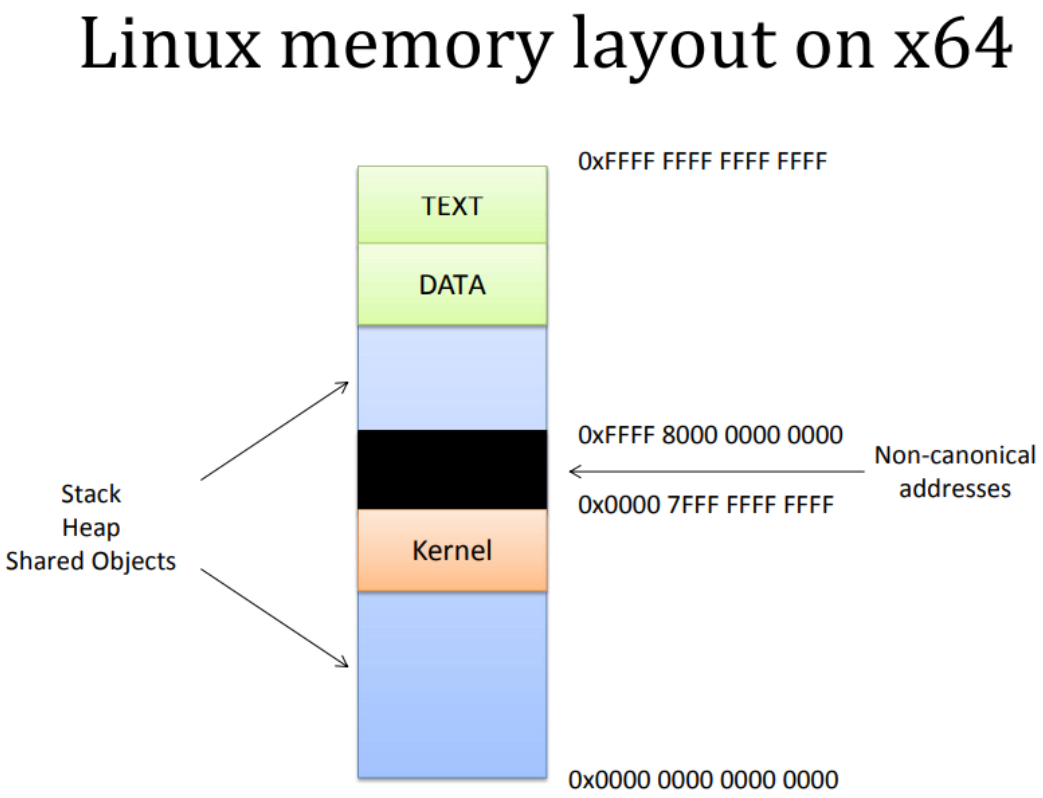
If smtg is wrong OS will send a Segmentation Fault to the app.

The kernel memory is always mapped and stored into physical memory. It can't be swapped, cause otherwise the OS implementation would be very more complex. What about virtual memory? In a 32 bits architecture the 4GB are divided in the following way:



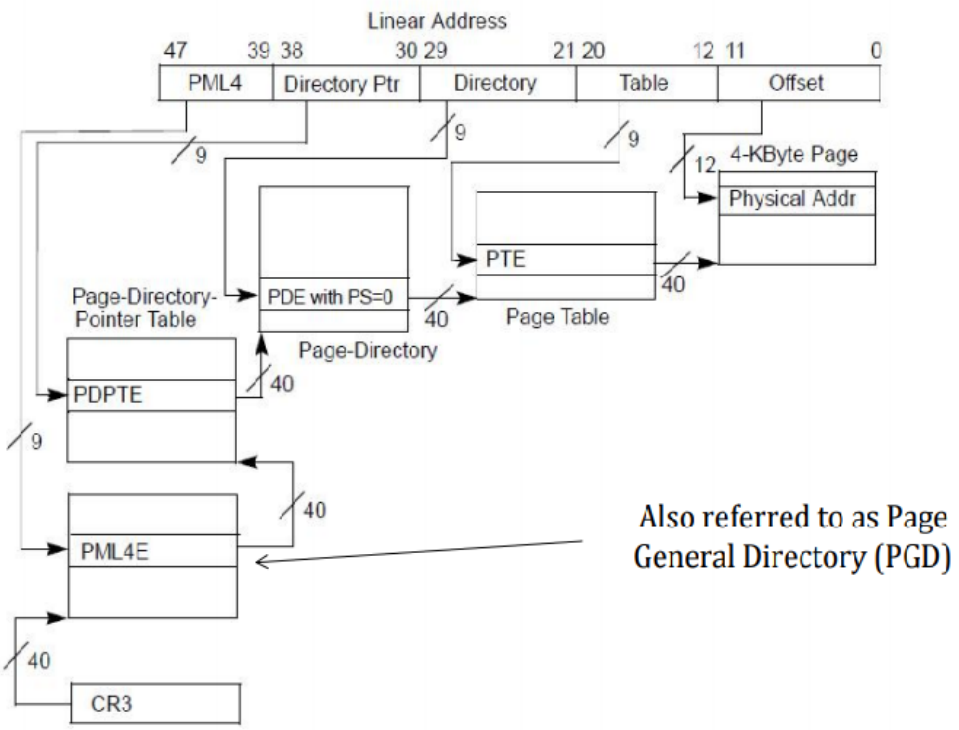
The highest 1 GB is reserved to always map the kernel memory. But how can we enforce security in this scheme? About reading kernel structures we have the User/Supervisor bit that would generate a General-Protection Fault in case of a wrong read and we have the Read Only bit that prevents user to write into that structures. About calling some routine we have the gate mechanism that protects the kernel routine call from user application. Meltdown born here: Kernel is mapped in process virtual space and the fault signal arrives late.

The 4GB memory is a limitation so was introduced Physical Address Extension that let us to map up to 64 GB of RAM using 36 bits. This implies that we have to use 3 levels of Paging. PAE is enabled thanks to the PAE-bit (bit 5) in the CR4 register. In this way the firmware will know that we are using 3 levels of paging.
This scheme is again extended during time with the 64 bit architectures. It is called LONG ADDRESSING. In theory we have 2^64 bytes of logical memory but it is enough to use 48 bits, so from 49 to 64 are short-circuited. We now have a total of 256 TB addressable memory. Having only 48 bits means that we have the canonical form addresses (the lower and the upper half, with a huge hole in the middle). The remaining bits are sign extended.



After 0x0000 7FFF FFFF FFFF there is 0xFFFF 8000 0000 0000 WTF! So in order to set a 48-bit Page Table we need to use a 48 bits linear address. To do that we reuse the same approach of PAE and we add a nee level in the scheme. We add the fourth level called Page Management Level 4 (PML4). So CR3 points the PML4 table.

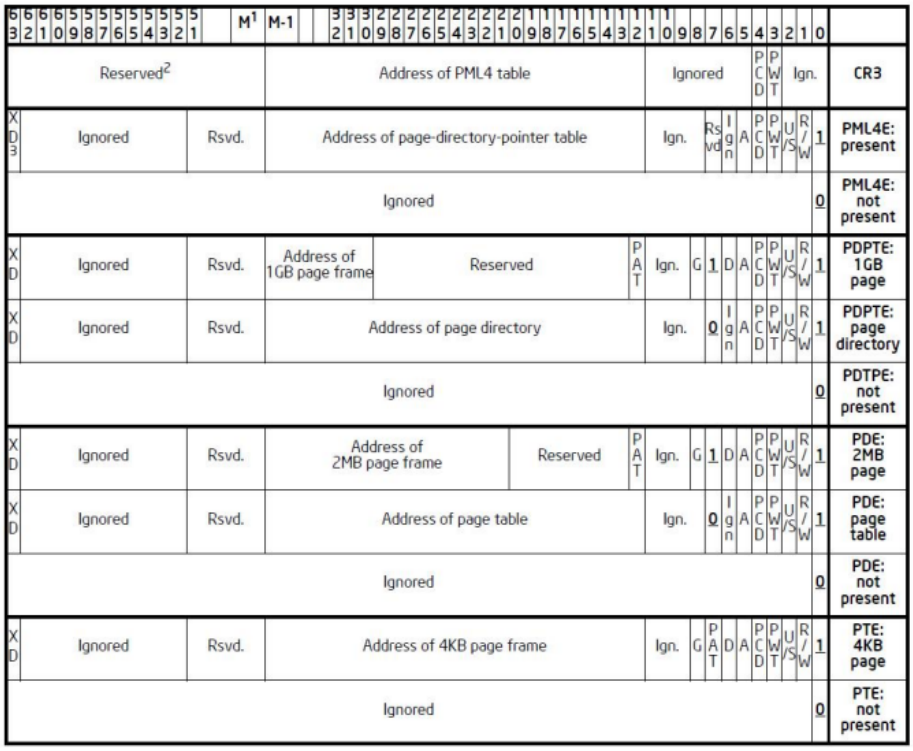
48-bit Page Table (4KB pages)



Now, since an entry of a table in the scheme is long 64 bits (see the offset set up to 40 bits), we have a less amount of entries in a table -> only 512! Cause a table fits a frame (4KB) and each entry is long 64 bits -> 4KB/8B = #512 entries. Thx to the 4 levels scheme, here we can map up to a 512 * 512 * 512 * 512 * 4KB of virtual memory.

In the 32 bit scheme of paging, there was no way to differentiate between a page that contains code or data! It was needed a bit specifying the executable property of a chunk of memory. It is in the actual implementation made by the Execution Disabled bit (XD).

CR3 and Paging Structure Entries



63	62	61	60	59	58	57	56	55	54	53	52	51	M[1]	M-1	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved[2]														Address of PML4 table												Ignored															PCD	PWT	Ign.			CR3
XD	Ignored												Rsvd.	Address of page-directory-pointer table												Ign.											Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored																																															0	PML4E: not present
XD	Ignored												Rsvd.	Address of 1GB page frame												Reserved											PAT	Ign.										G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored												Rsvd.	Address of page directory																							Ign.										0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored																																															0	PDTPE: not present
XD	Ignored												Rsvd.	Address of 2MB page frame																							Reserved									PAT	Ign.										G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored												Rsvd.	Address of page table																							Ign.										0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																																															0	PDE: not present
XD	Ignored												Rsvd.	Address of 4KB page frame																							Ign.										G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																																															0	PTE: not present

So let's recap the scenarios that could happen through an address translation. The first thing to do is to read the PML4 address stored in the CR3 register. Then if the Presence bit is set to 0, the translation will stop there and the remaining bits will be ignored, it means that they will be never read or modified (for this reason in those bits the kernel linux will write the swap file location if it is needed! to save space in memory obv). Otherwise we can take the Page Directory Pointer address and go down by a level in the translation. As for the 2MB huge page, here we can have a page up to 1 GB! if the Present bit is 1, we have to check the Page size bit: if it is set to 0 it means that we are referencing a Page Directory entry and so the translation will go on, otherwise the translation stops cause we are referencing directly a 1 GB frame (notice that cause we are in the third level of the scheme we can just map a 512 * 512 * 4KB page). As we said we can reference a Page Directory entry and then we have to follow the same rules described so far. The only system call that can make request to the OS for pages is the MMAP. This system call requires a page to the OS and, thanks to an integer flag that it takes as a parameter, it can distinguish between an ordinary or a huge page.

How to enable x64 longmode? Since the compatibility to the 32 bits system of a 64 bits one was developed by AMD, in order to enable the x64 longmode they couldn't use a control bit as for enabling Protected Mode, cause if Intel used that bit for another task, the compatibility would be broken. But the control registers are not the only thing that we have in our CPUs. In order to be compatible, it was introduced a register called Model Specific Register (MSR). It was introduced also to make some experiments on new features that will or will not remain in next CPU generation! Some version of CPUs has not this register, it can come and go, and the MST has no name! There are some specific asm instructions in order to read or write it. So for our purpose we have to deal with the MSR_EFER register (it stands for Extended Feature Enabled Register) in which there is a bit called _EFER_LME (it stand for Long Mode Enable). The semantic is put the MSR_EFER code in the C register, then we can use RDMSR instruction to load its content into the A register, then set the LME bit via BTS instruction and then write back the modified register with WRMSR. Now we want to start to run 64 bits code. **Ask professor!**

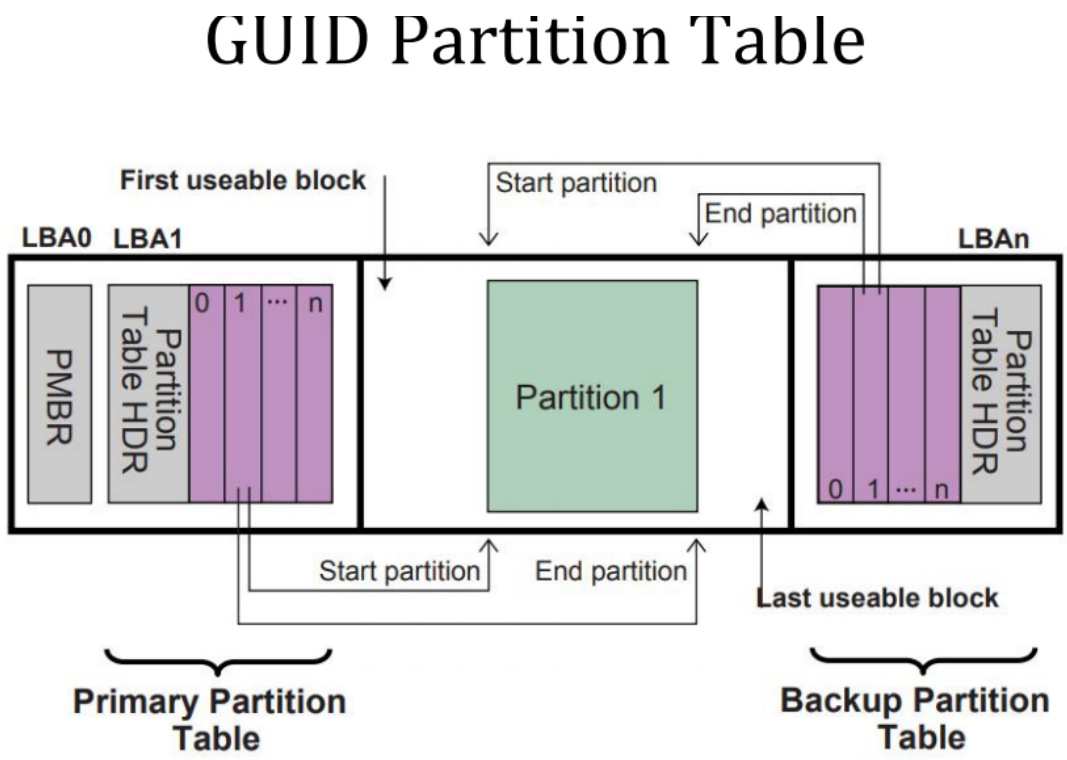
x86 INITIAL BOOT SEQUENCE - (18-03-2020)

Stage 2 Bootloader

Stage 1 knows where Stage 2 is located. Stage 2 allows us to select what kind and how (with which parameters) OS we want to start, also thanks to a gui. It is a simple interactive application. There are many variants of this software. In linux usually we use the GRUB software (GRand Unified Bootloader). Typically we can configure the GRUB through a config file called grub.conf. Since the GRUB can't load directly windows, to load Windows NT Kernel it first of all load the NTLDR that is the stage 2 bootloader of windows. It will load correctly the Windows OS. Stage 2 Bootloader, in order to load the OS image, relies on the BIOS. It use some disk I/O services offered by the BIOS in order to load the correct kernel image to load (for linux the path is /boot/vmlinuz-*).

In the early days, Intel is using what is so called UEFI (Unified Extensible Firmware Interface) that is so much modular (we can extend it with our driver f.e.) runs on various platforms and it is written in C! It simplifies a lot the boot sequence that we have already seen so far. UEFI boot manager takes control right after the system is powered on and it looks at the boot configuration in order to search bootable devices. At the end it loads firmware into RAM. The Startup files are stored on a dedicated EFI System Partition (ESP) which is a FAT32 one and there is one folder for each OS present on the system. Moreover UEFI overcomes the MBR limitation to handle disks at most 2TB large (due to the size of MBR block of course). It can automatically detect new uefi-boot targets thanks to the standardization of names and paths! The main function in UEFI programs is called efi_main and in order to write one we have to include just 2 libraries: efi.h and efilib.h, so simple.

UEFI uses a complete different partition scheme called GUID Partition Table.

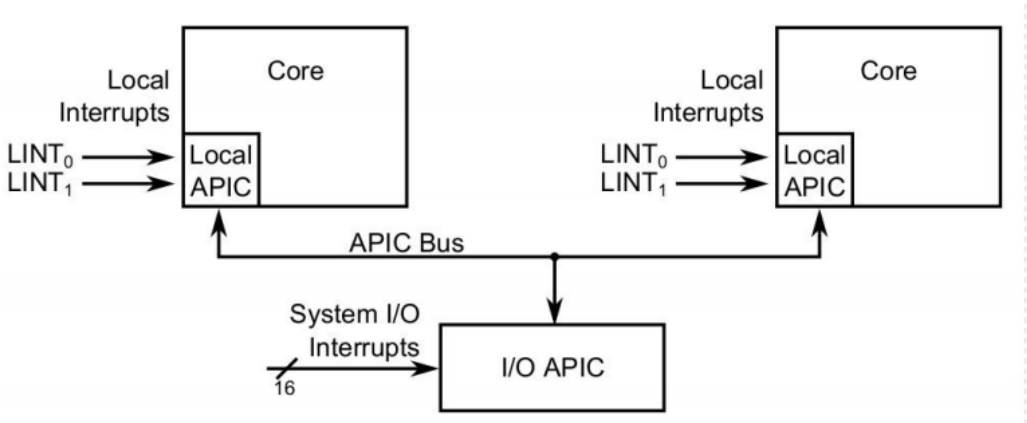


The interesting thing is that in the very beginning one block which is the MBR. UEFI is still backward compatible with the previous scheme and we can disable new UEFI approach and rely on the MBR Scheme only. Then we have a Partition Table HDR that can contain a custom number of partitions. WE have no longer the previous limitations about 4 entries in the partition table. In a entry we have a pointer to the start partition and one to the end. We have also a backup partition table for dependability purpose, it is stored in the last usable block of the device.

It was introduced also the Secure Boot. There was a security problem in the previous scheme: the content of MBR could be overwritten in order to perform an MBR Rootkits. The goal was to hijack the IDT for I/O operations. When any program performed a load of a block from disk relying on the BIOS routines, it would trigger the MBR Rootkit routines that will patch the binary code while loading it into memory. The OS will never notice that cause the patch was applied before it took control. UEFI allows to load only signed executables. The keys are installed in the system. We have a Platform Keys which tells who owns and controls the hardware platform, we have a Key-Exchange Keys that tells who is allowed to update keys in the hardware platform and the Signature Database Keys which show who is allowed to boot the platform in secure mode. We can boot only signed OS images! In the open source world this is a problem. I can't easily sign my own OS image cause the system doesn't have the asymmetric key to verify the correctness of the sign. The only way to boot in these OS is to disable the Secure Boot.

Now focus on dealing with multicores. Until now all the discussion was about sequential code. Who shall execute the startup code? For legacy reason, one core, the master (typically core 0), will execute it and the remaining will be in an idle state. But we have to wake up the other cores at a certain point. In order to do this we need a sort of synchronization within the cores. In fact each core has a Local APIC controller which can send Inter-Processor Interrupts (IPIs). All the LAPICs are connected through the (logical) APIC Bus (virtual network). Line 0 is associated to normal interrupts, Line 1 is associated to non-maskable interrupts (also if Interrupt Flag in the CPU is set to 0, I can't ignore a INT1). There is also a redirection table in I/O APICs used to route (how to dispatch) interrupts from peripheral buses to one or more local APICs (programmable).

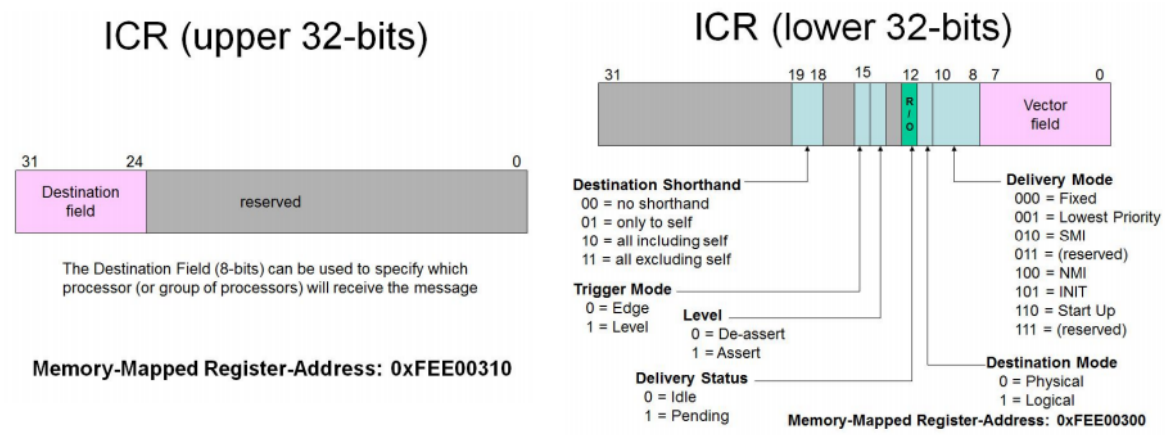
LAPIC



How does LAPIC work? To send messages among different cores, we rely on Interrupt Command Register (ICR) which is memory-mapped (we have to write in some memory location in order to write into it). It is divided in two part: the upper 32-bits and the lower 32-bits. We can relocate these data structures in memory.

Interrupt Command Register

- The ICR register is used to initiate an IPI
- Values written into it specify the type of interrupt to be sent, and the target core



In the upper part we have a Destination field which is a bit-mask. Each bit represents a core, if 1 the core will receive the message, otherwise not. Then we have an amount of reserved bits (not actually used). In the lower part we have a Destination Shorthand field which shortcut the destination field (ex. 11 is all excluding self). Then we have a Delivery Mode which tells how we want to schedule the IPI (ex. 100 is a Non-Maskable Interrupt). Two interesting values of the latter field are 101 -> INIT and 110 -> Startup: they are actually used to wake up all the remaining core in startup process. There is a sequence which is called INIT-SIPI-SIPI telling that the master core will send a INIT and a SIPI message to the other cores in a fixed sequence depending on the hardware. Different CPUs require different initialization sequence. It will be performed by the BIOS. In the Vector Field it is written the index of the IDT which tells what routine that specific interrupt will activate (cause we have just a line for interrupt so we can use an hardware coding). In case of INIT or SIPI message the Vector field doesn't contain any IDT index but a Page Frame Number PFN. So we can send also the address of the first instruction that each woken up core will fetch at startup. Moreover, the Vector Field allow us to manage exceptions: in fact in the Destination Shorthand with 01 we can send an IPI to ourself and then activate a routine (classic way to manage Exceptions). The Delivery status tells us if the IPI was correctly delivered (nothing on the execution because it is asynchronous). When we have correctly setup all the field in the ICR we can rise that flag and the LAPIC will send for us the IPI and turn down that flag.