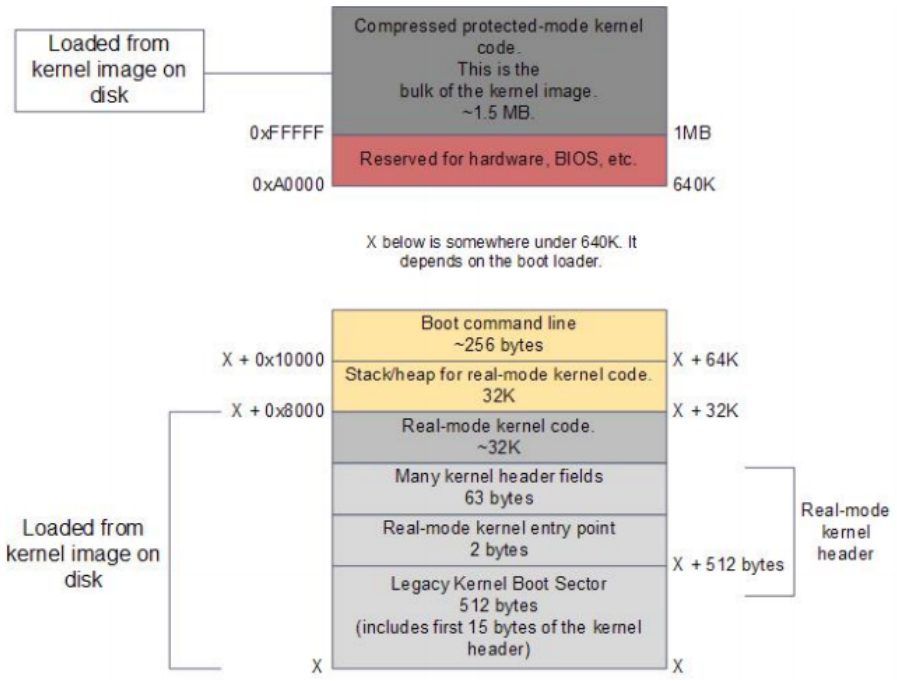
Linux Kernel Boot - (18-03-2020)

How the Linux kernel takes control of the machine? In the early phase of the Linux Kernel there still aren't services offered to the user space. After Stage 2 Bootloader or in UEFI system after Stage 1 Bootloader, a "primitive" image of the OS is loaded in memory, but in order to complete the Startup process, the kernel has to discover a lot about the machine in which is running and it must initialize a lot of its internal structures in order to support all these services offered to the users. So the first step now is to execute some hardware dependent routines in order to let the kernel be aware of the available hardware. Indeed in the /arch folder there are a lot of pieces of code that are hw dependent for each hardware supported by the actual OS. In the end the goal is to jump in a piece of code which is hardware independent.

The first image of the kernel which is loaded is very different from the steady-state one. First of all cause it is compressed! Indeed, it is called "vmlinux" the last char "x" means compressed.

RAM after the bootloader is done

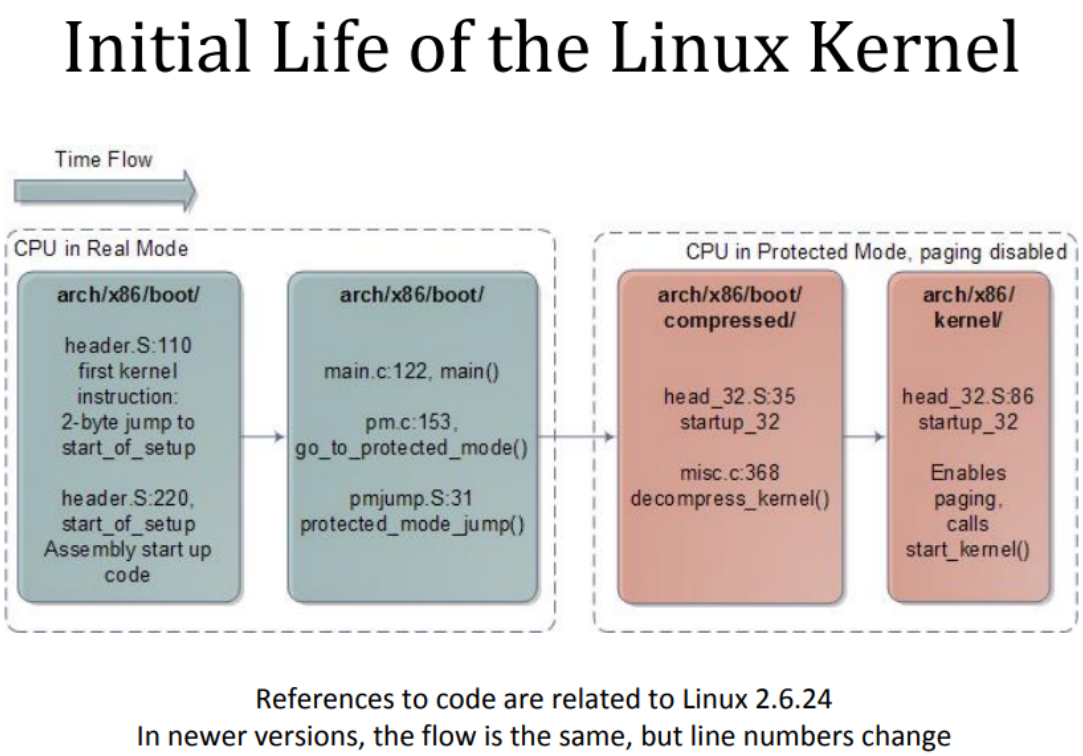


We have a scenario in which 1MB is the limit of memory addressable in Real Mode. We know that the BIOS can load memory in RAM thanks to its I/O routines built in. But the BIOS can't be executed in Protected Mode! We want to use also memory upper than 1 MB and we can do it ONLY in Protected Mode... So there is a sort of deadlock. How has this situation been solved? We have an UNREAL MODE (it is not an execution mode of x86 CPUs, just an hack to solve this situation). We know that the segment register has a selector part and an hidden and non programmable part for caching. That hidden part doesn't change if we don't change the content of CR= and vice versa. So the trick is: pass into Protected Mode setting accordingly CR0. Then adjust the content of the selector of the segment and so cache the address in the hidden part. Then go back to the Real Mode and load through the BIOS the OS image and store it in the address cached in the selector, before solved in Protected Mode (WOW). Another approach is reserved a buffer in Real Mode address space and load a piece of code, then pass to Protected Mode and copy that buffer in the memory over the 1 MB one. Then come back to Real Mode and

do-while it until all the image is loaded. This is the reason behind we have a compressed image of the Kernel! we must be as fast as possible.

In order to decompressed the kernel image the bootloader has to load 2 different kind of images of the kernel: a protected mode one (the one that we discussed before, which is compressed) and a real mode one (the one that must decompressed the protected one). The bootloader will give control to the guy called Real-mode kernel entry point which will jump to the Real-mode kernel code.

This is what we have to do since now:



References to code are related to Linux 2.6.24
In newer versions, the flow is the same, but line numbers change

The first thing which is activated is the entry point in Real Mode. It will perform some startup routines and it will call a main function (written in C but hardware dependent). At a certain point the protected mode is activated and it will jump into it activating a startup_32/64 procedure in order to decompress the kernel. Then it will be executed another startup_32/64 procedure (in this case the one of decompressed image) in order to enable paging and so discover physical memory. At the end it will be called the start_kernel() routine.

The first real instruction executed by the linux kernel is a jump into start_of_setup routine in real mode. The interesting thing that it is explicitly written in the asm code the 2 bytes jump, cause Torvalds thought that the compiler would use a 5 bytes jump.

What does start_of_setup() do? In order to start to execute code generated by the actual compiler we have to setup a minimalistic environment. Until now all the code used jump instructions to move from a point to another. We never used a call instruction. In order to do that we need at least a minimalistic stack and a bss section. This is the role of this function in order to be able to jump into the main() function. Moreover, this function implements part of the Kernel Boot Protocol and it loads in memory boot option passed through the Bootloader.

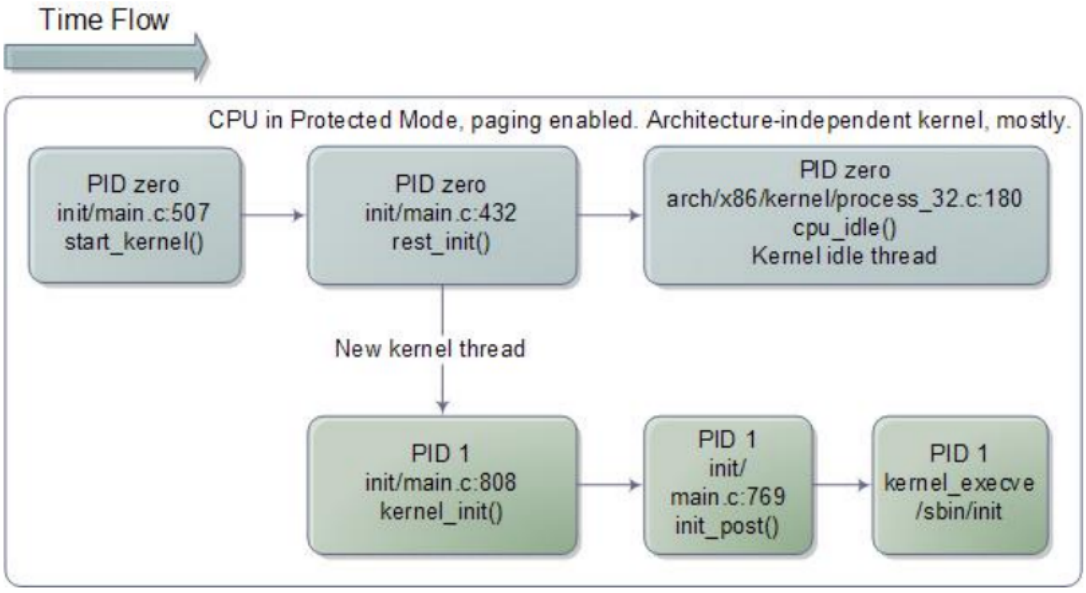
What does main() do? We want to pass into protected mode as fast as possible. In order to do this main() function will enable A20 line, setup an Interrupt Descriptor Table and setup memory. In Real Mode the Interrupt Vector Table is always kept at address 0x00...00. Now on, since we want to transition to protected mode we have to setup the Interrupt DESCRIPTOR Table and set accordingly the IDTR Register, cause protected mode will rely on it. But the kernel is still not able to support interrupts so it writes into IDTR null, for the moment in order to ignore them. Now we

have to setup GDT. Also GDT can't be manage completely at this time. It setups also a minimalistic gdt table. In order to run in protected mode it is needed a TSS entry. In order to do this it is needed to specify the control bit associated, the start address and the size of that segment. Then in order to execute code we need to setup a CS and a DS Segment. They starts at address 0 and fit all the Real Mode memory -> 0xfffff! Also in Real Mode kernel Linux works in a flat mode. In order to make things working it has to setup also GDTR register. But it can't still convert a virtual address to a physical address cause the page table is not already setup. So it has to perform a translation by hand in order to retrieve a valid physical address. So it multiplies with 16 the data segment address and it add the offset of the gdt (like the MMU does). It is the first example of this kind of operation made by the kernel cause some structures needs physical address to work. We will see other examples of that in startup. Now we can move into protected mode (protected_mode_jump()). The first thing is to set the PE bit in CR0. The perform a LJMP in order to load in CS the boot CS selector (segmentation means totally different in 32 bit mode). Set up data segments for flat 32-bit mode and finally sets a temporary stack for the protected mode.

At this point we can decompress the kernel. It is done in the startup_32/64() routine which implement the bzip protocol. Again it sets a new stack, calls decompress_kernel() and clears again BSS section in order to let work properly the decompressed code of the kernel (it is expected that memory to be all zeros). Finally it determines the actual position in memory via call/pop. This is an interesting thing which is associated to a security measure called Kernel Address Space Layout Randomization (KASLR). One thing that an attacker could do, if the system is compromised, is to hijack the content of a syscall, so patching at runtime the kernel code. It is more simple to do if an attacker can loop searching fingerprints of kernel code thanks to the static addresses of that structures in memory. So every time kernel starts, it will do it in at a different offset in order to be in a such way random. It does it at runtime relying on the most accurate source of entropy available. This process anyhow is reduced cause the kernel is mapped using 2MB pages and it has to be align to a page. So the number of valid slots is thus limited. At the end kernel must know where it was decompressed in memory. We can see smtg like: call foo (when foo is a label that point to the next instruction) and POPQ in a register of the return value on the top of the stack put there by the call. So we can easily know at which address the kernel code is stored.

So now we can pass the control to a routine called again startup_32/64(), completely different than the previous cause it is a more similar image of the kernel linux wrt the steady-state one. Now we can setup the Page Table cause also OS uses virtual addresses. This routine again clears the BSS segment. Setups a new GDT (the final one), builds the page table in order to enable paging, finalizes the IDT and finally jump into the first architecture-independent kernel entry point called start_kernel() at init/main.c.

Kernel Initialization

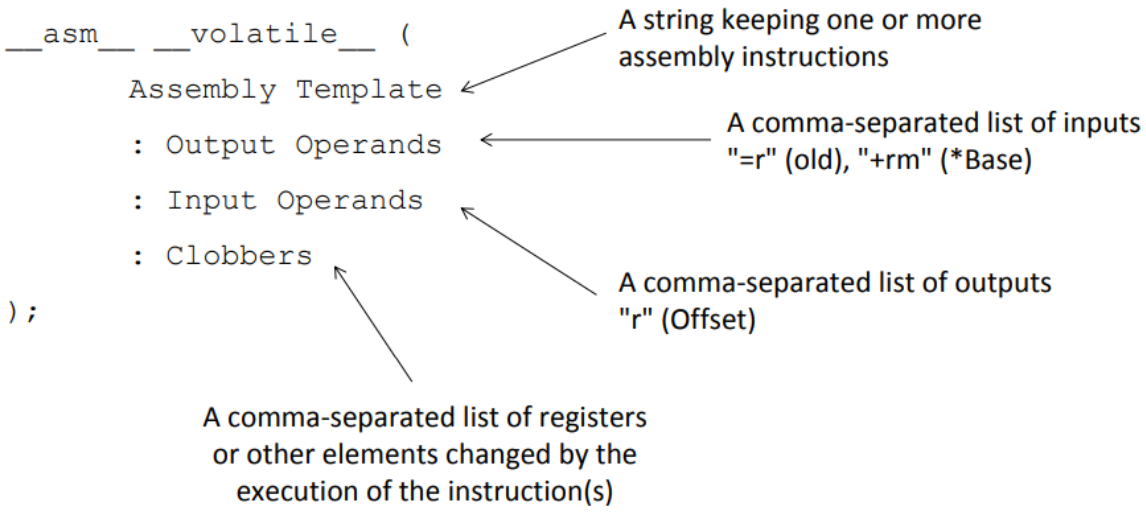


start_kernel() at a certain point will finish to initialize all the internal structures of the kernel, it will then call rest_init() routine. This piece of code is associated implicitly to the PID zero. That PID zero will eventually jump into cpu_idle() routine. Before that rest_init() will create a new Kernel thread which will be PID 1. PID 1 is in charge to call /sbin/init which in the very end will configure the user space environment.

All the initialization stuff that start_kernel() does is made sequentially cause all the other cores are spinning on a barrier in the meanwhile. When and only if all the kernel is setup and its structures are valid then all the cores can be unblocked and used by the OS steady-state image.

Now let's have a look to the INLINE ASSEMBLY used in the kernel code to embed asm instruction in a C workflow. It is needed because there are some operation that cannot be called in C. The syntax is:

Inline Assembly



The keyword is "__ asm __ __ volatile ()". Volatile means that I don't want the compiler reorder the code that I'm writing. The first parameter is an Assembly Template which is a string composed by asm instructions and placeholders. Then we have the Output Operands and Input Operands which are lists of inputs/outputs that specify the way the asm code uses them. Finally we have the Clobbers: it is so important specify also what are the changes that our inline code makes in order to let compiler be aware of the fact that our code is related to the one before and the one after generated by it. We don't have to smash nothing, but we have to tell what our assembly template modifies.

Linux Kernel Boot - (20-03-2020)

We now analyze how start_kernel() is declared: asmlinkage __ visible void __ init start_kernel(void);

"asmlinkage" is very common in the source tree code of the kernel. It tells the compiler that the called function should not expect to find arguments in registers. Now in order to explain this attribute we have to overview the calling conventions. We refer to the SysVABI that declare the calling conventions for all the Application Binary Interface for those OS of System V (Unix based). There are two way to pass arguments to a called function: stack one and registers one. Moreover, we can use the stack in two ways: the first is the callee cleanup approach (pascal compilers), it means that the caller put on the stack the parameters before doing the call and then the called function is in charge to clean those parameters (in order to do that the called function relied on an asm instruction called RET N when N are the bytes that have to be removed from the stack), the second approach is the caller cleanup (C programs), it means that the parameters are still passed on the stack but it is the caller function that must clean the stack, this is cause there are some function declared as variadic ones like the printf, it means that a function can have an infinite number of parameters so only the caller can easily clean the stack. The Caller Cleanup approach is used in 32 bits programs. The registers approach is used in 64 bits programs. Why? Cause we have more register space obv. The order of parameters is: RDI, RSI, RDX, RCX, R8, R9 (for integers/pointers) and XMM0, ..., XMM6 (for floats). The ret value is always in the A register. If there is a function that uses more than 6 parameters then we can pay the cost to put them onto the stack. Moreover, the calling conventions divides the registers in: caller save and callee save. The first one must be restored by the caller function cause they could be clobbered by the called function, instead the second one must be restored by the called function, it means that before

using that registers it must save onto the stack its actual value and then restore it before return to the caller function. Callee save: RBX, RBP, R12-R15. Caller save: others.
The kernel want to be fast: in the 32 bit architecture, kernel linux was compiled with a flag available in gcc compiler that can change manually the calling convention. The kernel in many functions expected the parameters to be in registers (for speed). So far so good since we are leaving in the kernel space. But if it was performed a syscall that pass from user space to kernel space it would be a problem: user space relied on stack caller cleanup convention, kernel not. In that specific entry point (from user space to kernel space on more general from a different calling convention wrt to the kernel one) it is used the ASMLINKAGE attribute that prevents kernel function the default behavior and aligns the calling convention in order to not break all. If the calling convention are already aligned the ASMLINKAGE attribute is projected into an NO-OP. So kernel portability is enforced.

Another notation to explain is __VISIBLE that prevents Link-Time Optimization. The compiler generates an object file from each code file. Then all the object files will be linked into an executable. If there is a call to f() in an object but the declaration of f() is in another object file there will generate a non available mapping for that function. In the end the linker will resolve that missing linking that symbol to another object file symbol. It was introduced the Link-Time Optimization (LTO) that removes all the so called dead code that survived into an executable but it was never effectively used (one of the optizimation). It does a lot of think like inline asm functions. The start_kernel() is the entry point in the linux kernel initialization and it is called by some asm code which might never reference directly the symbol. So it can happen that the Linker could think to delete this portion of code marking it as not used. The VISIBLE attribute prevents this.

The third attribute is __ INIT that frees this memory after initialization. All the memory used in a INIT marked function is placed in a specific place in order to free that at the end of the initialization process by the kernel for future usage.

Memory initialization

We are running in Protected Mode, the Kernel image is in RAM. It must initialize the Page Table cause paging is still disabled but it can't do it dynamically cause the Kernel doesn't have already setup the Memory Management System and it doesn't know anything about physical available memory. We are in a very bad deadlock situation. So the kernel exploit a nice trick in order to resolve this situation. It will use a hard coded minimal Page Table in order to come out with the setup of the Kernel Page Table to let it finish its startup configuration (notice that we are dealing ONLY with kernel memory and still not with user space memory). This is fundamental cause the compiler generated a kernel image which uses virtual addresses! We really need a Page Table to translate that addresses.

Enabling Paging

```
movl $swapper_pg_dir-__PAGE_OFFSET,%eax
movl %eax,%cr3 /* set the page table pointer */
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0 /* set paging (PG) bit */
```

The last three lines are quite simple to understand cause its goal is just to flip the PG bit in the CR0 register. Then the second line write a physical address into CR3 (remember it is the first physical base address for the Page Table). But what is this magic physical address put in that place? The swapper_pg_dir symbol is actually a virtual address that point to an minimal hard coded page table in the kernel image. How can we transform at code time without the Page Table a virtual address into a physical one? So, the Segment Unit is not actually a problem cause all the segments are set to 0 for the flat mode that is used. In order to make the physical trnaslation by hand we need to reason about the memory situation at startup_kernel(): the Kernel in Real Mode unzip itself after the first 1MB of physical memory. The peculiarity of the unzip function is that the kernel image is stored in a continuous piece of memory! In a 32 bit virtual address space (for simplicity but it is the same mechanism used in 64 bit architecture plus a runtime fix due to the canonical addresses and the KASLR approach) the kernel reserves for its image the last GB of virtual memory... again contiguous! So the real distance or better the real translation can be done just applying a constant offset! It is called __ PAGE _ OFFSET. In the end we can reach the physical address of the hard coded page table called swapper pg dir. The idea is to occupy the minimum amount of space in order to let things work correctly, so it was decided to implement a 1 Level Paging Scheme which let to use 4MB frame address in 32 bit mode. The Page Table at compile time generated has only 2 valid entries, each one for a 4MB page. This allows us to map in Virtual Memory ONLY 8 MB. The remaining part of the kernel memory is still not mapped in virtual memory. In these 8MB of mapped memory we have kernel code, some kernel data and also free memory. The free memory is essential in order to enlarge the hard coded minimal page table for mapping all the remaining kernel virtual addresses. What we have until now? What we want to do?

1. We need to reach the correct granularity for paging (4KB)
2. We need to span logical to physical address across the whole 1GB of manageable physical memory
3. We need to re-organize the page table in two separate levels
4. So we need to determine 'free buffers' within the already reachable memory segment to initially expand the page table
5. We cannot use memory management facilities other than paging (Kernel-level memory manager is not ready yet!)
6. We need to find a way to describe the physical memory
7. We're not dealing with userspace memory yet!

The structures that we want to compute at steady-state are essentially two: the Kernel Page Table and the Core Map. The first one keeps the memory mapping for kernel level code and data (also thread stack). The second, instead, keeps status information for any frame of physical memory and for any NUMA node! Cause we have already said that kernel linux is NUMA aware. In fact there is a free list of physical memory frames for any NUMA node. These two structures are the point 5 and 6 in the above image. We need a way to describe physical memory in order to allow kernel linux to allocate and deallocate memory in its life. In order to do this the kernel linux relies on the Bootmem: a very minimalistic and not efficient allocator which is single threaded. Remember we are running sequentially. It provides useful allocation capability only in startup phase. At the end it will be collected by the garbage collector cause it is declared as __init. Since there is only an offset between virtual and physical addresses and since the space already occupied is well known by the kernel at compile time, the Bootmem allocator relies on a Bitmap that describes if a 4KB of memory of the 8MB initially map is busy or not. It allows the kernel to solve point 1 (granularity) and point 4 cause in this way it can thanks to the Bootmem API allocates simply 4KB of free buffer in order to expand the Page Table and map all its 1GB of virtual Memory.

What the kernel has to do is to complete the 2 levels scheme of paging. How can it do that? Easily enough. At startup only two entries of one level are valid and point to 4MB pages. But CR3 points to a 4KB buffer! So we still have in the external level of paging space to store information about the second level of paging. We have to find 1K 4KB free buffers in order to set second level paging, populate them as PT Entries which point to 4KB Frame and write their addresses into the free entries of the hard coded page table! In this way we have setup a 2 levels page table with a 4GB of virtual address space reachable.

This stuff is done by the bootmem but in the early days it was superseded by the Logical Memory Block (LMB). It is less efficient and elegant but it manage in a better way the more and more scattered memory situation and also it can manage an amount of memory becoming bigger over time. Memory is represented as two arrays of regions: Physically-contiguous memory and the Allocated regions. It allows to register a physical memory range associated to a node, it can reserve some memory for future purposes and it can find an aligned free area in a given range.

To finish the setup of the Page Table we have to rely on the architecture which is running, because it could happen that different architecture requires different paging scheme. Linux starts with a 2 levels paging but already at that time with the possibility to increase up to 3 levels the scheme. During the time the amount of available memory grew a lot and all the operating systems had to aligned themselves. Linux on the 32 bit mode runs with 3 levels of paging: Page Table, Page Middle Directory and Page General Directory. It has also the possibility to manage 4 levels of paging introducing a Page Upper Directory between the Middle Directory and the

General Directory. Why this position? Cause the Page General Directory is always referenced in the code as the last part of the scheme, so in order to touch the minimum amount of code, the new level is added in the middle (only the paging unit has to be touch). Another problem is the number of bits to assign to each field to let the Page Unit find correctly the entry in each table, cause there are different number of bits for different architectures! We want to write code which is most independent from the architecture as possible. As always we rely on MACROS. Macros let us to write uniform code, but they are dependent from the architecture! For each architecture we have different definitions for the number of bits of the fields, we have the possibility to customize them and set also the length of a linear address, we can set also the SHIFT SIZE of each field (SHIFT cause in order to retrieve a specific field we have to shift the entire address for a fixed amount of positions), we have also masks in order to clear out the lower part of the address. WE already said that a 32 bit version uses a 2 levels paging scheme at firmware but 3 in software. How can we solve this difference? Relying again on macros called PTRS_PER_x where x is the table in the scheme that we want to set. For example in that case the Middle Directory is set to just one entry skipping so that table!

The entries of the tables of the scheme are obv of different types (cause they refer to different table) but in the very end they are all long value! In order to distinguish between them linux keep some typedeffed structs that contains just an unsigned long. Why is this complicated representation? Cause C is a WEAK TYPED language: it means that if I make a typedef of the unsigned long into 2 different types, the compiler won't generate any warning or error if I make some wrong assignment (one type to another). This is so dangerous while debugging kernel code. So that representation using the typedef struct let the compiler warning the programmer that a wrong assignment between DIFFERENT types has been done in the code. Each long address contains the physical address of the frame or the next entry of the scheme and some control bits. For each control bit there is a macro that define its position in the entry. So each time we want to check or set a control bit we can rely on the struct which we are defined so fare and mask them logically with the macros defined in kernel code. It is so interesting because for example the dirty or accessed bit is set by the firmware, and through code we can access easily that information due to the fact that software and firmware share all the page table scheme. Moreover the operating system has also a lot of fancy macros in order to make the code more readable. For example it has two macros to distinguish between PT entries associated to user space and kernel space (the only bit that changes is the USER BIT).

pagetable_init() (2.4.22)

```
for (; i < PTRS_PER_PGD; pgd++, i++) {

    vaddr = i*PGDIR_SIZE;  /* i is set to map from 3 GB */
    if (end && (vaddr >= end)) break;
    pmd = (pmd_t *)pgd;/* pgd initialized to (swapper_pg_dir+i) */
    .........
    for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
        .........
        pte_base = pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);

        for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
            vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
            if (end && (vaddr >= end)) break;
                .........
            *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
        }
        set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte_base)));
        .........
    }
}
```

So in the end this is a piece of code which describes what is the way in which kernel setup the page table at startup. In the more recent version of the kernel the idea is the same, but due to the increase amount of memory and levels, the code was split in more functions and some other macros were added. This is some architecture independent code which initialize the page table for KERNEL translate of addresses from virtual to physical mapping. At the time we had only 3 levels so we have 3 nested loops, one for each intermediate table. It is interesting to notice that if we have to set just 2 levels the middle loop will iterate just one! Thanks to the definitions set in the kernel code as we said before. So, we start to iterate from an i value in the first loop in order to map only the last virtual gigabyte. We don't need the entire last gigabyte but we check if the size of the kernel image is completed mapped, then we finish. We are going to overwrite also the first 2 entries of the swapping page dir in order to reach the 4KB granularity (notice that the first 2 entries are rewritten with a pointer to a PT entry, which remapped all the 4MB Page, so it is not changed but only adjusted for the required granularity). The population of the tables is done bottom up. Since CR3 points to the first level and since to translate correctly an address we have to traverse all the levels, we want to ensure that only if the below levels are correctly set an upper entry is available. So first of all we set the bottom up the entries. Pay attention to the fact that each physical address is computed relying on the constant offset between the virtual space and physical one of the kernel (kernel is contiguous decompressed!). In order to make this translation again the kernel relies on macros called __pa(virtual address) and __va(physical address).

Is it enough? No we have to consider the so called Translation Lookaside Buffer (TLB)! We have previously overwritten the Page Table (changing 4MB Pages mapping into 4KB Pages) but in the TLB we have the previous translation cached yet. In order to make things leveled we have to execute two lines of code: first of all we load_cr3(swapper_pg_dir), it means we set the base address of the most external layer of the paging scheme into the CR3 register (as the protocol wants, notice that we are writing the same address and the function will perform internally the virtual to physical translation of that address), why we are doing this? cause the Intel architecture, every time we overwrite the CR3 content it will automatically flushes the content of TLB, but this is not true for all the architectures so we have by hand to do this relying on a __flush_tlb_all() function that flushes the tlb explicitly and synchronously on ALL CPUs. This is done obv for being more portable as we can.

The load_cr3 function is architecture independent cause it performs only a virtual to physical translation and then relies on the native_write_cr3 function, which is architecture dependent. Both the functions are declared as INLINE function, it means that the only goal of this code chain is to check at compile time if some type error occurred. At the very end the chain will be translate in a simple asm instruction that MOV some long unsigned physical address into the CR3 register. Remember that on Intel architecture this means also flushing TLB on the cpu that executes this instruction. Another security measure that is taken is to enforce the fact that we don't want that the MOV instruction would be reordered. We want that the overwrite of the CR3 content is a sort of fence for our code, but fence is too slow and so we don't explicitly set the memory clobber definition in the asm code. The volatile attribute grantees that the code is not reordered in any case but we want also a sort of serialization of the code during memory accesses. So the trick is abusing of compiler's functions: we add a memory operand that load some data stored in the __force_order variable which is defined extern, moreover this variable is never really declared and allocated in the source code of the kernel so the compiler will wait the linker in order to allocate memory, at the end this __force_order symbol will disappear but since this is a memory operand it is enough to ask the compiler to force serialization also in memory accesses. This is much more light then fence instructions! A really good optimization :D.

In order to be portable, we have to keep in mind the worst scenario that we want to support. This is the reason behind the flush_tlb_all() function. We have to write code that must cover all the possible situation. Then if a particular architecture does smtg automatically then we can map a specific function to a non op one, but we can't to the reverse reason! So think before write code. TLB changes are really important and dangerous at the same time cause we could violate the processes isolation every time we perform a context switch! So the types of TLB relevant events are divided for a Scale classification or for a Typology classification. Scale: global, if they deal with virtual addresses accessible by every CPU in real-time-concurrency, or local meaning that just a small part of virtual address space should be affected. Topology: first type is a virtual to physical address remapping, second one is virtual address access rule modification. There are some direct costs for flushing TLB like the latency of the firmware for the "physical" entries invalidation or the latency due to the cross-CPU coordination in case of global TLB flushes. There are also some indirect costs like the misses that will be generated after an invalidation of the TLB (it means that firmware has to navigate again the MMU in order to translate a specific v.a.).

What are the API? The first one which we have already seen is the void flush_tlb_all(void) which flushes the entire TLB on ALL processors running in the system, obv it is the most expensive one. After it all modifications to the Page Table are globally visible, a specific case in which it is absolutely necessary is after updating the kernel page table (globally for definition). Then we have a flush_tlb_mm(struct mm_struct *mm) which is a partial flushing of the TLB related only to a portion of virtual memory associated to an user context. In fact it relies on a mm_struct parameter (we will see it later on). This function will be correctly mapped to a set of architecture dependent operations cause for example in MIPS architecture is required to do this on all cores (via IPI messages). It is used usually after a fork() or in the COW protection. Then there is the flush_tlb_page that takes in input another memory struct which is vm_area_struct. It flushes one specific page, it is useful when a page fault occurs. If we have this kind of feature we can do this thing very efficient, instead when it missing it could mean that we would invalidate all the TLB. Then we have a flush_tlb_range which from a mm_struct flushes all the TLB entries related to a start and a end pointer memory region. It is used when function like mremap() and mprotect() are called. When it is possible to use this function, it is faster to do this for a range than iterate through pages and use flush_tlb_page(). Then we have the flush_tlb_pgtables that let us to flush entries associated to the page table, from a start pointer to an end one related to an mm_struct. It is used when an unmap operation is performed. Then we have update_mmu_cache which prefetch information about the translation of a specific PTE. It is used after a page fault

completes (in some architecture)in order to load manually that information in the TLB and let the next translation to avoid to go through the page table in order to complete it.

The last two things that the kernel must do in startup mode are to set the final version of the IDT and the GDT, now the kernel space is correctly mapped so we can do this. At the beginning kernel mapped the IDT to a null one in order to avoid some crash due to the interrupts, now it calls trap_init() that in the very end will populate the IDT with all the required routines. Then kernel populates the final GDT. It must be able to defined Kernel Code Segment, Kernel Data Segment, User Code Segment and User Data Segment in order to serve interrupt requests. Also it has to initialize the TSS for each different core otherwise we are not able to transition from user space to kernel space, also the dummy shared LDT in order to make happy the firmware. Since there is a different instance of TSS per cpu we have the GDT as a per cpu variable! TSS is used to have different execution context (ring stacks) on different CPUs.

Now we can jump into cpu_idle()! It calls a cpu_idle_loop which is a while(1) loop. In this infinite loop we are asking ourself if there is the need to do some rescheduling operations (relying on different flags kernel depending). If it is not the case we rely on a cpuidle_idle_call() which is an inline asm instruction mapped to the HLT one, otherwise (thanks also to a timer interrupt that let the process exit by the idle state), we ask explicitly to the scheduler to reschedule us cause there is something more important to do.

THE END OF THE BOOTING PROCESS.