1044414: Advanced Operating Systems and Virtualization **Spring 2018**

Lecture 4: March 13

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj*

In the previous lecture (Lec 3, Sec 3.4) we described the boot process for the early Linux kernel implementation with the built in bootloader. With the spread adoption of Linux, various bootloaders were implemented and to regulate how the kernel has to be setup into memory by bootloaders the linux kernel defined the Linux Boot Protocol [lbp()]. The main component of the protocol is the Real-mode kernel header which is a set of variables, some statically set at compile time and some that must be written by the bootloader that allow the kernel to know the initial memory map of the system.

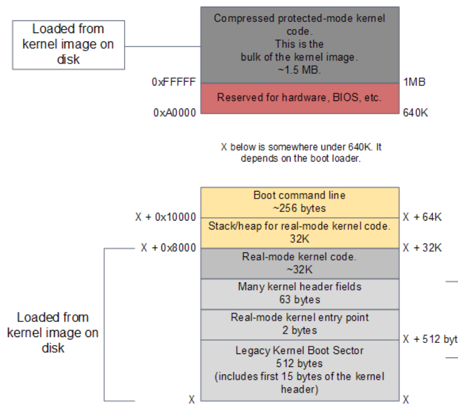
The kernel initialization can be divided into two parts. The first part is the Real Mode kernel and the second part is vmlinux which is the full binary object of the linux kernel and starts executing with `startup_32` at `arch/x86/boot/compressed/head_{32,64}.S`. With the introduction of (U)EFI and the advanced interface provided with it the kernel setup code became useless during time therefore the Efi Handover Protocol was introduced along the Linux Boot Protocol that allows to load the kernel directly in Protected Mode instead of Real Mode skipping the kernel setup code.

4.1 Kernel Initialization

The bootloader ensures that the kernel is loaded as shown in Figure 4.1 with the Real-mode kernel loaded in the first megabyte of memory and the compressed Protected-mode image right after the first megabyte of memory [1]. After loading the kernel, it jumps to the `_start` label placed at 512 bytes of offset from the address where the first Real-mode sector was loaded. From now on is the kernel that will execute in Real-mode.

The first instruction performed is an hardcoded instruction opcode 16-bit short jump [2] to skip the second part of the header and execute code at `start_of_setup`.

```
> arch/x86/boot/header.S
    _start:
        .byte 0xeb
        .byte start_of_setup-1f
    1:
```



Loaded from kernel image on disk
Compressed protected-mode kernel code. This is the bulk of the kernel image. ~1.5 MB.
0xFFFFF
1MB
Reserved for hardware, BIOS, etc.
0xA0000
640K
X below is somewhere under 640K. It depends on the boot loader.
Boot command line ~256 bytes
X + 0x10000
X + 64K
Stack/heap for real-mode kernel code. 32K
X + 0x8000
X + 32K
Real-mode kernel code. ~32K
Many kernel header fields 63 bytes
Real-mode kernel entry point 2 bytes
X + 512 byt
Loaded from kernel image on disk
Legacy Kernel Boot Sector 512 bytes (includes first 15 bytes of the kernel header)
X
X

Figure 4.1: Memory after bootloader [Duarte(2008)]

The `start_of_setup` routine first ensures `%es = %ds` so it is possible to use the two segment registers interchangeably and sets the `%cs` register to the same value of the other segment

START OF SETUP
```
        pushw %ds
        pushw 6f,
        lret
6:
```

[1] The loading above the first megabyte is performed either through BIOS facilities or Unreal Mode. Remember that the bootloader as well runs in Real-mode first therefore it cannot access that region of memory directly

[2] The jump is hardcoded to prevent the compiler from producing 32-bit code (instead of 16-bit) where such instruction has a 3 byte opcode. The instruction takes as input an immediate and will set the instruction pointer to the address of the instruction following the jump plus the immediate passed as input. The immediate is computed at compile time through the gnu assembler functionalities where `start_of_setup` is the label corresponding to the address where the kernel has to jump to and `1f` is the label `1` forward (`f`) which corresponds to the value of the instruction pointer when the jump instruction will be evaluated.

registers through the code shown on the left which sets `%cs` to the value written on the stack above the return address and jumps to the label following.

[Alex()] After that it setups a preliminary stack and zeroes the `.bss` section that holds the unitialized variables since a malevolent or buggy bootloader might have changed some of them.

Finally it jumps to the `main()` function found in `arch/x86/boot/main.c`

Boot Main In the `main` function the following initializations are performed.

```
> arch/x86/boot/main.c

main(void)
    copy_boot_params(void)
        copies the parameters of the Kernel header into a struct

    console_init(void)
        initializes console for printing on screen or serial
        communication

    init_heap(void)
        initializes a heap

    check_cpu(void)
        discovers cpu features

    detect_memory(void)
        invokes the E820 BIOS facility for generating the physical
        memory map

    // other initialization ...

    go_to_protected_mode(void)
        switches to Protected Mode
```

Go To Protected Mode The `go_to_protected_mode()` function performs the last preparations before entering Protected Mode and finally executes Protected-mode code.

```
> arch/x86/boot/pm.c

go_to_protected_mode(void)

    realmode_switch_hook(void)
        disable Non Maskable Interrupts (NMI)

    enable_a20(void)
        enable A20 Line as shown in previous lectures

    mask_all_interrupts(void)

    setup_idt(void) {
            static const struct gdt_ptr null_idt = {0, 0};
        asm volatile("lidtl %0" : : "m" (null_idt));
    }

    setups a dummy IDT
```

The IDT and GDT are initialized at this stage because otherwise the processor won't enter in Protected mode as shown in the commit message

```
setup_gdt(void) {
        static const u64 boot_gdt[] __attribute__((aligned(16))) = {
            [GDT_ENTRY_BOOT_CS] = GDT_ENTRY(0xc09b, 0, 0xfffff),
            [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0, 0xfffff),
            [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096, 103),
        };

        static struct gdt_ptr gdt;
        gdt.len = sizeof(boot_gdt)-1;
        gdt.ptr = (u32)&boot_gdt + (ds() << 4);

        asm volatile("lgdtl %0" : : "m" (gdt));
    }

protected_mode_jump(boot_params.hdr.code32_start,
                    (u32)&boot_params + (ds() << 4));
```

`GDT_ENTRY` takes in input flags, base and limit of a descriptor and generates the entry. Limit is 4KB $\times(2^{20}-1) \approx$ 4GB (G bit set to 1 through flags)

The GDT entries indices `GDT_ENTRY_BOOT_CS`, `GDT_ENTRY_BOOT_DS`, `GDT_ENTRY_BOOT_TSS` are respectively 2, 3, 4. Entry 0 is reserved as shown in the Intel documentation [Intel()] while entry 1 is not used by Linux.

The GDT `len` must be an unsigned short integer that summed with `ptr` tells the last byte addressable of the table (that's why 1 is subtracted to it) while the `ptr` must be the 32 bit physical address (that's why `gdt.ptr = &boot_gdt + (%ds` $\times$ 16) of the 16 byte-aligned memory containing the GDT.

Both the GDT and IDT setup functions use inline assembly to perform hardware specific operations which are respectively to load the GDT register and the IDT register with the content of the memory referenced by `gdt/null_idt`. In the next paragraph we'll describe more deeply the syntax.

Protected Mode Jump

Finally the routine `protected_mode_jump(u32 entrypoint, u32 bootparams)` defined in `arch/x86/boot/pmjump.S` performs the jump to Protected Mode code in `startup_32`.

```
> arch/x86/boot/pmjump.S

protected_mode_jump(u32 entrypoint, u32 bootparams)
    .code16                 ; tell compiler to produce 16bit opcode
    ; store bootparams in %esi
    ; calculate pm32 physical address and keep in in 2f

    movw $__BOOT_DS, %cx
    movw $__BOOT_TSS, %di

    movl %cr0, %edx         ; enable protected mode
    orb $X86_CR0_PE, %dl    ; through CR0
    movl %edx, %cr0

    .byte   0x66, 0xea      ; 32-bit ljmp opcode
2:
    .long   in_pm32
    .word   __BOOT_CS

    .code32                 ; tell compiler to produce 32bit opcode

in_pm32:

    ; initialize segment registers
    ; setup stack for debugging
    ; setup Task Register otherwise cannot enable Protected-mode
    ; clear general purpose registers
    ; load Local Descriptor Table otherwise cannot enable Protected-mode
    ; jump to 32-bit entry point startup_32 in
    ;       arch/x86/boot/compressed/head_64.S
```

As the reader may have noticed the code to which the previous snippet jumps to belongs to the `compressed/` folder under the `arch/x86/boot/` directory. This is the entry point of the kernel in case of booting through EFI.

Also notice that the code that we are going to explore is for the `x86_64` architecture. The flow is similar for `x86_32` and the code can be found in the same folder in `head_32.S`

Startup 32 — Compressed
Note that the initialization shown in class is about the `x86_32` kernel following the code in `head_32.S` and described in the previous lecture

The first instruction performed is `cld` which clears DF used by `stos` and `scas` instructions. Then a test on the `KEEP_SEGMENTS` parameter in the headers set by the bootloader is performed to check whether to reinitialize the segment registers.

In order to know the current physical address where the kernel was loaded the following instructions are performed

```
> arch/x86/boot/compressed/head_64.S

    leal    (BP_scratch+4)(%esi), %esp
    call    1f
1:  popl    %ebp
    subl    $1b, %ebp
```

The stack pointer is set to a physical address known to be "scratchable" to perform the instructions. A `call` to the following instruction is issued which pushes in the stack the value of the current address of execution which is then `popped` in the `%ebp` register. To this value is subtracted the relative address (with respect to the start of the binary object) of label `1` getting the physical address of `startup_32`.

Next, knowing the physical address, a stack is setup, a cpu verification is performed to know whether the architecture provides long mode and calculates the relocation address (physical address where the kernel will be loaded).

The last operations are preparations for entering long mode. A new GDT is setup since the previous ones were set for only 32-bit mode (different L and D flags from these ones). The

PAE bit is set in %cr4 and an early page table initialization is performed and its address set into %cr3.

Finally long mode is enabled through the EFER model specific register as shown in the previous lectures, the address of the next routine to be executed is pushed in the stack, paging is enabled by setting the PG bit and the PE bit in %cr0 and lret instruction is issued to start execute startup_64 still in head_64.S. In startup_64 the bss section is cleared to prepare for the jump to the C code of extract_kernel() (arch/x86/boot/compressed/misc.c) which will decompress the kernel and set %rax to the address at which the kernel was decompressed. head_64.S then jumps to it with jmp *%rax.

Startup 64 — Kernel

The kernel entry point is startup_64 which code is in arch/x86/kernel/head_64.S (similarly there are two files for the x86_32 kernel) that modifies slightly the page table and jumps to x86_64_start_kernel() defined in arch/x86/kernel/head64.c which finally calls start_kernel() defined in init/main.c. start_kernel() will perform many kinds of system initializations and wake up the other cores through the INIT-SIPI signals.

4.2 Inline Assembly

Inline Assembly is a feature provided by the GNU compiler and adopted by other compilers to embed assembly instructions inside C/C++ code. Such construct is useful when some code needs to be highly efficient and even writing it as an asm function (fun.S) the call overhead would be too much or implementing highly hardware-specific code. Both cases suggest that a lot of kernel code contains inline assembly.

The syntax for using inline assembly is:

```
asm [volatile] ( AssemblerTemplate
                 [ : OutputOperands ]
                 [ : InputOperands  ]
                 [ : Clobbers       ] )
```

Both the asm and volatile keywords can be used with or without underscores (e.g. __asm__). The volatile keyword prevents the assembler from optimizing this portion of code by either removing it or placing it into other positions. It does *not* ensure the ordering of the instructions between assembly and C code.

An example of inline assembly is the following: asm ("movl %%eax, %%ebx");.

When referring to registers two % symbols are needed since the usage of just one % symbol is reserved for referring to C variables inside the Assembler Template. To refer to C variables in the inline assembly code we must "define" them inside the Input or Output operands. The Clobber part informs the compiler what registers or memory areas are going to be modified since it does not parse the assembler template.

Let's look at some examples to get a better idea.

```
void cpuid(int code, uint32_t *a, uint32_t *d) {
    asm volatile("cpuid"
        :"=a"(*a),"=d"(*d)
        :"a"(code)
        :"ecx","ebx");
}
```

The code above performs the `cpuid` instruction. The `code` variable is stored int the `%eax` register before issuing the `cpuid` instruction through the input operands and the contents in the `%eax` (`a`) register and `%edx` (`d`) register are respectively written (`=`) in the memory area pointed by the variable `a` (`(*a)`) and `d` (`(*d)`) after the instruction. Indeed if we compile the code above we get:

```
cpuid(int, int*, int*):
    pushl %ebx
    movl 8(%esp), %eax      ; a (code)
    cpuid
    movl 12(%esp), %ecx     ;
    movl %eax, (%ecx)       ; =a (*a)
    movl 16(%esp), %eax     ;
    movl %edx, (%eax)       ; =d (*d)
    popl %ebx
    ret
```

Another example is the `swap` function below.

```
void swap(long *a, long *b){
    __asm__ __volatile__ (
        "push (%%rax) \n"
        "push (%%rbx) \n"
        "pop (%%rax)  \n"
        "pop (%%rbx)"
        :  "=a" (a) , "=b" (b)
        : "0" (a) , "1" (b)
        : "memory"
    );
}
```

The variables passed as input are moved to the `rax` and `rbx` registers and the values stored in the memory pointed by the two pointers are pushed to then be popped in the reverse order into the memory pointed by the two pointers. In the input operands the registers are referred through the sequential enumeration offered by GCC. With string `"0"` it refers to the 0th register declared in the output/input operands, i.e. `%rax` in this case, while string `"1"` refers to the 1st register declared which is `%rbx`.

Setting the string `"memory"` into the clobbers informs the compiler that the portion of inline assembly performs side effect in memory and ensures that the compiler won't perform any kind of reordering in the scope in which the inline assembly is written.

Last example

```
bool CAS(volatile unsigned long long *ptr,
        unsigned long long oldVal,
        unsigned long long newVal) {
    unsigned long res = 0;

    __asm__ __volatile__(
        "lock cmpxchgq %1, %2;"//ZF = 1 if succeeded
        "lahf;"         // to get the correct result even if oldVal == 0
        "bt $14, %%ax;" // is ZF set? (ZF is the 6'th bit in %ah,
                        // so it's the 14'th in ax)
        "adc %0, %0"    // get the result
        : "=r"(res)
        : "r"(newVal), "m"(*ptr), "a"(oldVal), "0"(res)
        : "memory"
);

    return (bool) res;
}
```

The code produced by the assembler is the following

```
CAS(unsigned long long volatile*, unsigned long long, unsigned long long):
    xorl %ecx, %ecx
    movq %rsi, %rax
    lock cmpxchgq %rdx, (%rdi);
    lahf;
    bt $14, %ax
    adc %rcx, %rcx
    testq %rcx, %rcx
    setne %al
    ret
```

From the `x64` calling convention the first three parameters are stored into registers `%rdi, %rsi, %rdx`. `oldVal` which is in `%rsi` is moved into `%rax` as required by the input operand. In the inline assembly `cmpxchgq` takes in input the 1st and 2nd input operands which are `newVal` and `ptr` that are stored `%rdx` and `%rdi`.

4.2.1 Kernel Initialization Signature

Another important fact to be analyzed is the `start_kernel()` signature which has the following macros prepended to it:

`asmlinkage`: in order to be more performant in i386 architecture the kernel is compiled with the option `-mregparm=3` which forces GCC to optimize function calls by putting the parameters into the registers instead of putting them on stack as defined by the ABI. This macro ensures that the parameters passed to the function will be on stack.

`__visible`: informs the compiler to not remove the function during the link-time optimization even if it is not called by any code

`__init`: sets the function binary into a specific region of memory called `.init.text` allowing the kernel to safely reclaim that space after initialization is concluded

References

[lbp()] Linux boot protocol. URL https://www.kernel.org/doc/Documentation/x86/boot.txt.

[Alex()] Alex. Linux inside. URL https://0xax.gitbooks.io/linux-insides/content/index.html.

[Bovet and Cesati(2006)] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. OReilly, 2006.

[Duarte(2008)] Gustavo Duarte. The kernel boot process, Jun 2008. URL https://manybutfinite.com/post/kernel-boot-process/.

[Intel()] Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual*, volume 3A.