Loading the kernel, Initializing the page table

Loading the kernel

- So far, we have looked at the first few instructions, written in assembly, to initialize 32-bit mode, segment registers, stack pointer, and a call to the C function `bootmain`.
- `bootmain` is not supposed to return (it is supposed to load the kernel and jump into it). However, it may return if an error was observed during the loading process. In which case, the code following the `call` instruction executes some instructions which allow a user to debug the error.
- The `bootmain` function (on Sheet 85) will read the kernel image (stored in ELF format) from the disk, load its program segments into physical memory, and branch to its first instruction.
- The xv6 disk has the kernel image starting at sector no. 1 (second sector on disk). (Recall that sector no. 0 was the boot sector).
- ELF is a standard format used to represent executable and object files. In this case, we use this format for the kernel. Typically, the OS loader loads an executable by parsing the ELF file. In this case, the "bootloader" will load the kernel by parsing the ELF formatted data stored in disk blocks (starting at disk block 1).
- ELF Format: The first few bytes contain the ELF header, which must have a fixed "magic" number at a fixed offset (to confirm that it is indeed a valid ELF file), and should have a pointer to the "program headers" which store information about "program segments". Each program header represents a program segment (e.g., code, data, etc.). Also, the ELF header will store the number of program segments (or headers).
- Each program header contains the following information:
  - `offset`: the file offset at which the program segment is in the file (or on the disk).
  - `filesz`: the size of the program segment in the file
  - `paddr`: the physical address at which the segment should be loaded.
  - `vaddr`: the virtual address at which the segment should be loaded.
  - `memsz`: the size of the program segment in memory. This must be greater than `filesz`. If `memsz` is not equal to `filesz`, the remaining `(memsz - filesz)` bytes in the segment are initialized to zero by the loader.
- On Linux, type `man elf` to get full details about the ELF format.
- Let's get back to the code in `bootmain.c`. Line 8527 makes a call to `readseg()` to read the first few bytes starting at disk sector 1 (notice the disk sector computed at line 8589 starts at sector 1) into a memory area named by the variable `elf`.
- The `elf` variable is treated as an ELF header and its values are read (e.g., `magic` field, `phoff` field to obtain the location of program headers, `phnum` field to obtain the number of program headers).
- The loop in lines 8536-8541 iterates over the program headers, and loads the program segment corresponding to each program header from disk to memory.
- The call to `readseg()` in line 8538 loads `ph.filesz` bytes from disk at offset `ph.off` and stores them at *physical address* `ph.paddr`.
- Recall that because we are currently working with physical memory (paging has not yet been enabled), the bootloader uses the physical address in the program header (`paddr`) to load the corresponding segment.
- The lines 8539-8540 check if `ph.memsz` is greater than `ph.filesz`, and if so, zeroes out the remaining bytes (as required).
- Finally, the ELF header also contains the starting instruction address in the `entry` field, to which the bootloader makes a function call at line 8546. At this point the control has shifted from the bootloader to the first instruction in the kernel.
- To understand exactly what is happening, type the following command to look inside the kernel ELF file:

  ```
  xv6$ objdump -p kernel
  Program Header:
      LOAD off    0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12
           filesz 0x0000b596 memsz 0x000126fc flags rwx
     STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
           filesz 0x00000000 memsz 0x00000000 flags rwx
  ```

  The first segment is supposed to be loaded from offset `0x1000`, at virtual address `0x80100000`, physical address `0x100000`. Its size in the file is `0xb596` bytes, and its size in memory is `126fc` bytes.

  The second segment indicates a stack segment, and we can ignore it for now.

- Thus the kernel has only one relevant segment that is loaded at physical address of 1MB (0x100000) and virtual address of KERNBASE + 1MB (0x80100000). KERNBASE is the starting address of the kernel's address space (0x80000000 or 2GB in xv6).
- Why load at paddr 0x100000? that's 1MB. Why not 0x0? there are memory-mapped devices at physical address less than 1MB. Why not 2MB? that would work too but that may waste more physical memory space.
- To see the starting instruction address of the kernel, type the following command:

  ```
  xv6$ objdump -f kernel

  kernel:     file format elf32-i386
  architecture: i386, flags 0x00000112:
  EXEC_P, HAS_SYMS, D_PAGED

  start address 0x0010000c
  ```

Notice that the start address is a physical address (and deliberately so). This is correct because at the time of branching to the start address, we are still operating in the physical address space.
- All other symbols in the kernel are assigned virtual addresses (above KERNBASE).

Initializing the page table

- The first few instructions in the kernel (`entry.S`) execute from physical address space and initialize the page table and enable paging. After paging is enabled, the kernel will start executing from virtual address space.
- Ignore lines 1042-1044. They enable large pages in the hardware
- Lines 1046-1047 load a page directory into cr3. The contents of the page directory are provided in a global array called `entrypgdir`. `entrypgdir` is assigned a virtual address (like all other kernel variables). Thus the code converts it into a physical address (by subtracting KERNBASE from it using V2P_WO macro) before loading it into `cr3` register (as `cr3` needs a physical address).
- Let's look at `entrypgdir` defined at Sheet 13. `entrypgdir` is an array of 1024 `pde_t` entries. In this case, all other entries are set to zero (which means the corresponding pages are not present), except the first entry (entry #0) and the `(KERNBASE >> PDXSHIFT)`'th entry (entry #512) which both point to the same first 4MB page in the physical address space. Notice that these are the only two entries where the present bit (indicated by PTE_P) is set. Also, notice that both entries use large pages and allow writes.
- As soon as paging is enabled (line 1051), the address space changes, and it is possible that the program counter `EIP` may now point to a different byte. To avoid this, the kernel developer has carefully used an identity mapping for the first 4MB, i.e., the first 4MB of VA space (virtual address space) map to the first 4MB of PA space (physical address space). Through this, the developer ensures that the current execution addresses will remain valid even after paging is enabled (all these addresses are less than 4MB by design).
- Also, the size of the kernel (code and data) is less than 4MB, so this ensures that the kernel is fully mapped in the virtual address space at their corresponding physical addresses, even after paging is enabled.
- Moreover, the first 4MB of the physical address space is also mapped at virtual address KERNBASE (from KERNBASE to KERNBASE+4MB) in this address space. The next few instructions will re-initialize the stack and the program counter to virtual addresses, and the kernel will operate in the virtual address space from there on (leaving the bottom 2GB of virtual address space for user processes).
- Line 1054 pushes the value of the `stack` variable into `esp`. Note that this will be a virtual address value (greater than KERNBASE), just like all other variables in the kernel. Because the address space in that region is mapped, this value will point to a valid address.
- Finally, lines 1060-1061 branch to `main`, which is also a virtual address (greater than KERNBASE). After this instruction, the execution remains in virtual address mode.
- Notice that the kernel used a temporary page table (`entrypgdir`) to switch from the physical address space to the virtual address space. This temporary page table had both physical-side mappings (0-4MB) and virtual-side mappings (KERNBASE to KERNBASE+4MB) for the same physical addresses (0-4MB). This allowed transitioning from one address space to another.
- After the transition has taken place, the kernel will install a new page table which will only have virtual-side mappings. All virtual addresses below KERNBASE will be used for user processes from there on.
- Let's look at the `main()` function at line 1217. This function initializes the physical memory, page tables, I/O devices, and other things, before initializing the first user process (line 1239) and going into an infinite scheduling loop thereafter (which schedules the available processes). Because xv6 is a multiprocessor OS, the first CPU that has booted also initializes the other CPUs in the system (line 1237).