Lousine Kernel Documentations

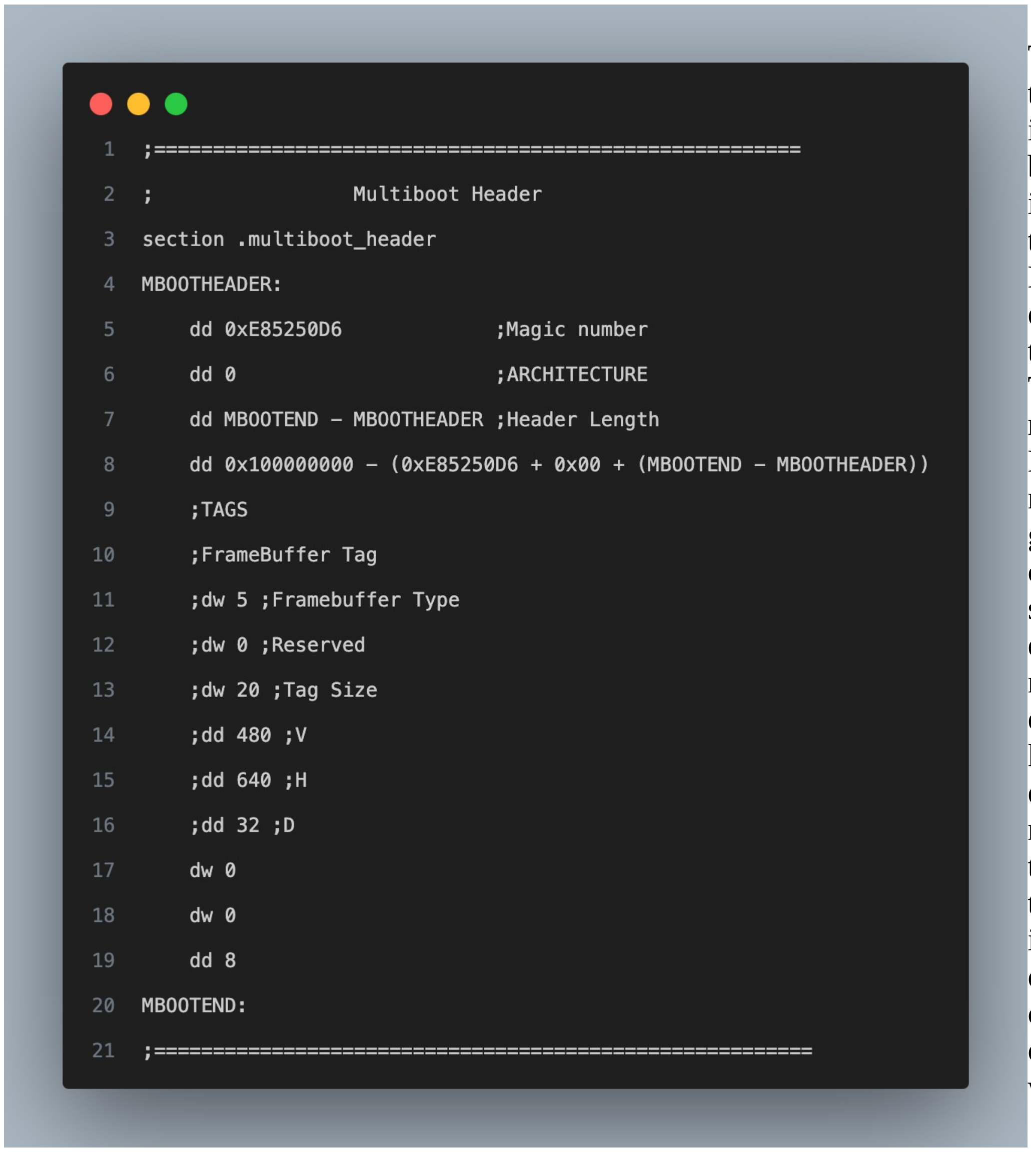
Name Of File: Boot.asm
Subject Of File: Boot.asm
Documentation Revisions: Version 1.01

Documentation Author: Tyler Grenier
Project Affiliation: Kernel Owner/Creator

Boot.asm

Section 1: Multiboot Header

Boot.asm may look small because it is. Typically the kernel has no use in using assembly for major tasks other than what is required. Now although this file is small this is one of the most important files in the entire source tree because the system can not boot the kernel without it. Currently the only kernel version is the AMD x64 (located in LOUOSKRNL.EXE/boot/x86_64/boot.asm) version due to restraints in the complexity of the kernel falling into the hands of one person, the kernel will eventually expand into more than just a simplistic single architecture system, but a journey of a thousand miles has to start at one step and I only have x64 computers laying around.



```
;========================================================
;                  Multiboot Header
section .multiboot_header
MBOOTHEADER:
    dd 0xE85250D6             ;Magic number
    dd 0                      ;ARCHITECTURE
    dd MBOOTEND - MBOOTHEADER ;Header Length
    dd 0x100000000 - (0xE85250D6 + 0x00 + (MBOOTEND - MBOOTHEADER))
    ;TAGS
    ;FrameBuffer Tag
    ;dw 5 ;Framebuffer Type
    ;dw 0 ;Reserved
    ;dw 20 ;Tag Size
    ;dd 480 ;V
    ;dd 640 ;H
    ;dd 32 ;D
    dw 0
    dw 0
    dd 8
MBOOTEND:
;=========================================================
```

Figure 1.1. Multiboot header

THE MULTIBOOT STRUCTURE:

The multiboot structure is the most important struct in the kernel It allows our boot loader (GRUB), To identify several important things about the LOUOSKRNL.EXE executable that are needed to run the kernel.
These structures that are required consist of The..:
Magic Number: the magic number is required for grub or any multiboot2 compliant boot loader to see the kernel as a valid executable. Magic numbers are not actually exclusive to just the kernel, in fact all executable files have a magic header to confirm that the segments of data that has been loaded is indeed a valid executable otherwise the system could cause a GPF Fault or and invalid opcode which is usually fatal for

Lousine Kernel Documentations

Name Of File: Boot.asm
Subject Of File: Boot.asm
Documentation Revisions: Version 1.01

Documentation Author: Tyler Grenier
Project Affiliation: Kernel Owner/Creator

any running code and would cause a DestroyThread call for that function an if the kernel itself does it, it means that the error is 99% time unrecoverable and you get a crash screen.

Architecture:

the architecture in the multiboot2 header is just as important if not more as the checksum for the architecture say in x86_64/AMD64 is not compatible with an ARM processor or the x86 processor. The reason for this is that companies like AMD, Intel, Nvidia, VIA, and Apple all have their own products some of them have features x, y and z some have features w, x and y. the architectures of these products vary. For instance in the early 2000s and 90s the popular standard for computer processors were the x86 processors such as the 8086, the 386 the 486 and the pentiums. Then in the mid 2000s the x86_64 or AMD64 processors came out being able to run more than the 32 bit limit hex value 0xFFFFFFFF or 4,294,967,295 in decimal value of bytes in ram space. Allowing for larger programs which we have all been using up until recently when ARM 64 has taken Over. The difference between these to systems is their instructions AMD is a CISC Processor, and the ARM is a RISC Processor similar to the mips systems that were in the Sony Playstation 1 & 2 & 3. the difference is the RISC or Reduced Instruction Set is able to do more computation in less amount of instructions being “REDUCED INSTRUCTION SET”. Currently this is set to 0 for x86 System.

Header Length:

the header length is to tell the loader of the multiboot2 program that the size of the data in the header is X amount of bytes this is required because even thought like most executables (.exe, .o, .elf, .s) that are standardized through the community they might either have optional flags OR the entries in the data might have vendor specific information (vendor specific information is the bane of my damn existance).

Frame Buffer Tag:

for now I’m going to briefly explain this tags and other tag if you want to know more about the multiboot2 systems it on the GNU website for multiboot2 specification. As for the actual information on the tags the tags are entries in the header that tell the program loader what the program loader should do before loading the program such as loading a frame buffer, where exactly to put the executable, the alignment and finally after all required and extra flags the null tag that tells the program loader that we reached the end of the header.

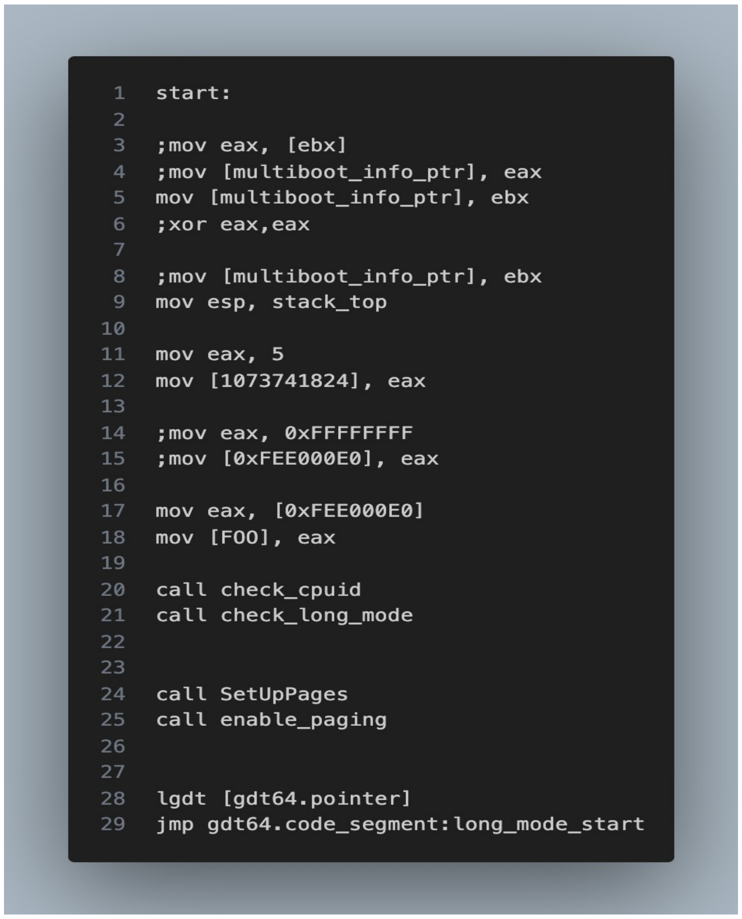
section 1 reflection:

So in conclusion the multiboot2 header system might be one of the most important structure in the system because it handles the relation of the kernels setup from the boot loaders (GRUBS) perspective.

Lousine Kernel Documentations

Name Of File: Boot.asm
Subject Of File: Boot.asm
Documentation Revisions: Version 1.01

Documentation Author: Tyler Grenier
Project Affiliation: Kernel Owner/Creator



```
start:

;mov eax, [ebx]
;mov [multiboot_info_ptr], eax
mov [multiboot_info_ptr], ebx
;xor eax,eax

;mov [multiboot_info_ptr], ebx
mov esp, stack_top

mov eax, 5
mov [1073741824], eax

;mov eax, 0xFFFFFFFF
;mov [0xFEE000E0], eax

mov eax, [0xFEE000E0]
mov [FOO], eax

call check_cpuid
call check_long_mode


call SetUpPages
call enable_paging


lgdt [gdt64.pointer]
jmp gdt64.code_segment:long_mode_start
```

Figure 2.1:Kernel x64 Entry

KERNEL X64 Entry:

The kernel entry is the next important step on our list this sets up important processor features that are required for x86_64 processor operation in 32 bit x86 mode. But Wait? You thought this was a 64 bit OS thats odd that we are in 32 bit operation… well you are correct this is a 64 bit system however the actual computer for bios systems starts in x16 bit real mode and switched to x86 mode during the boot loaders process and uefi starts in 32 bit mode. This is actually due to two reasons that really just coincide with each other.and for this the two reasons for the invisible hardware tornado that has made all hardware plateau leading to the computer industry switching to newer systems. These two reasons are the designs of the systems being poor due to reality of how things actually work in the real world and the legacy/backwards compatibility requirements of large corporations slowing the evolution of the rest of our species, once again the large corporations are whats going to get us all killed. For example when a plumber or electrician does work on the house the worker usually accesses things in the basement to fix something but when something is in the living quarters such as a wall they have to tear things apart and rebuild them from scratch rather than having a leaky pipe soldered up or a new ire strung to add or fix to the old one. This is the same thing with computers except it is more of burning the whole house down and starting from scratch whenever something as minor as deleting a function of code. This is actually why as you browse through t some code you will see some code is not being used right now because its now obsolete and is only there for reference until its replacement is created. Without further ado lets get into this function in depth.

Section 2.1: Setting Up For The Kernel:

the first thing that we do when we enter the start function is we save important data to our heap that we get from the boot loader so we don’t over over write the data because the data pointes to a table that will give us information like the RSDP or the Root System Directory Pointer, which points to information like ACPI Tables that point to the APIC and memory systems and most importantly the ACPI (Advanced Configuration Power Interface) system which is responsible for managing the power in all of the devices (This is what is programmed when the shutdown button is hit in the os to do a software shutdown).

The second thing we do is we Set up a new stack for our kernel which is in our case 16 Kilobyte and we move the top of the stack to RSP, which is the x86_64/AMD x64 register for managing stack space.

Then after doing some check operations we call CPUID. CPUID will be covered more in depth later in the x86_64 drivers that are within the kernels main tree, for now all you need to know is that certain computers don’t have CPUID if the computer doesn’t have CPUID then the computer will NOT have x64 mode and then the kernel will actually halt and crash.

After CPUID is proven to work in the kernel the system will then check for long mode. Long mode is the technical term for the x86_64/AMD x64 mode that allows the computer to run 64 bit data types. Checking for this mode typically relies on directly telling the ALU register to set the 33 bit explicitly then testing it to see if the value in “doubled memory” is equal to the data relayed from the test.

Once long mode is initialized we have three main tasks we are required to perform before even considering entering long mode and they cannot be undone or tampered with in runtime or they will cause Fatal Faults due to the nature of the long mode states run time.

- Setting Up Page Tables And Enable Paging
- Create a Global Descriptor Table (that will under no circumstances be tampered with)
- Perform a long jump to 64 bit code and flush the system

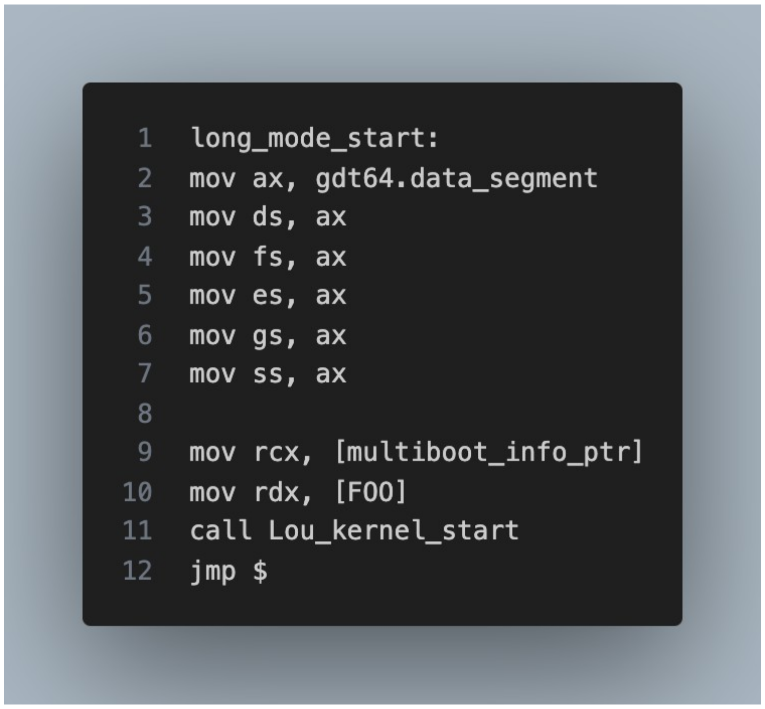
I will be covering paging later in the paging file however the important thing is you need 4,512, 64 bit variables aligned by 4 kilobytes for the tables, move the (addresses | 0b11) of the Lower table into the higher table down to L2 and then fill L2 with ((addresses 1 through 512 * 4mb) | 0b100011) with a gap of 4mb in the address per L2 entry. This essentially identity maps the first gigabyte to memory, Then marl the l4 table as a global variable this will be important for later. Then we load the L4 with some flags into cr3 effectively enabling paging.
Then we set the GDT but I am not going into how because the GDT is pretty much irrelevant in x86_64 and all I use it for it to separate the stacks of certain processeses

Lousine Kernel Documentations

Name Of File: Boot.asm
Subject Of File: Boot.asm
Documentation Revisions: Version 1.01

Documentation Author: Tyler Grenier
Project Affiliation: Kernel Owner/Creator

finally we jump into the GDT code segment and flush the system below push the data we stored into the passing parameters for the MSVC ABI then call our LouKernelStart Function and loop.



```
long_mode_start:
mov ax, gdt64.data_segment
mov ds, ax
mov fs, ax
mov es, ax
mov gs, ax
mov ss, ax

mov rcx, [multiboot_info_ptr]
mov rdx, [FOO]
call Lou_kernel_start
jmp $
```