# Higher Half x86 Bare Bones

From OSDev Wiki

**Difficulty level**

Medium

## Contents



# Theory

A higher half kernel is a kernel that is mapped to the higher half of the virtual address space, despite being loaded to the lower half of the physical address space. In order to understand the concept, it is recommended that you have a firm grasp of theory at bare bones and paging.

When GRUB passes control to the kernel, paging is disabled. This means that physical and virtual addresses correspond to the same addresses.

When writing a higher-half kernel, the steps required are:

- Reserve some pages for the initial mappings, so it is possible to parse GRUB structures before memory management is enabled.
- Create page tables that contain the page frames of the kernel image.
- Create the boot page directory that contains the aforementioned page tables both in lower half and higher half.
- Enable paging.
- Jump to higher half.
- Remove the lower half kernel mapping.

# Code

Here is some sample code for a kernel that is loaded by the GRUB bootloader and then mapped in the upper half of virtual memory. In this case the kernel is loaded at physical address 0x00100000 and mapped to virtual address 0xC0100000.

## Assumptions

This example makes many assumptions:

- It assumes the kernel is smaller than 3 MiB (reserved is only one boot page table).
- It assumes the developer wants to map the kernel at 0xC0000000.
- It assumes every ELF section is both readable and writable (which is not the case for `.text` and `.rodata`)

# boot.s

After GRUB passes control to the kernel, this piece of code is executed, starting from _start. This essentially bootstraps the kernel by performing the aforementioned steps required and calling `main()`.

```
# Declare constants for the multiboot header.
.set ALIGN,    1<<0             # align loaded modules on page boundaries
.set MEMINFO,  1<<1             # provide memory map
.set FLAGS,    ALIGN | MEMINFO  # this is the Multiboot 'flag' field
.set MAGIC,    0x1BADB002       # 'magic number' lets bootloader find the header
.set CHECKSUM, -(MAGIC + FLAGS) # checksum of above, to prove we are multiboot

# Declare a multiboot header that marks the program as a kernel.
.section .multiboot
.align 4
.long MAGIC
.long FLAGS
.long CHECKSUM

# Allocate the initial stack.
.section .bootstrap_stack, "aw", @nobits
stack_bottom:
.skip 16384 # 16 KiB
stack_top:

# Preallocate pages used for paging. Don't hard-code addresses and assume they
# are available, as the bootloader might have loaded its multiboot structures or
# modules there. This lets the bootloader know it must avoid the addresses.
.section .bss, "aw", @nobits
        .align 4096
boot_page_directory:
        .skip 4096
boot_page_table1:
        .skip 4096
# Further page tables may be required if the kernel grows beyond 3 MiB.

# The kernel entry point.
.section .text
.global _start
.type _start, @function
_start:
        # Physical address of boot_page_table1.
        # TODO: I recall seeing some assembly that used a macro to do the
        #       conversions to and from physical. Maybe this should be done in this
        #       code as well?
        movl $(boot_page_table1 - 0xC0000000), %edi
        # First address to map is address 0.
        # TODO: Start at the first kernel page instead. Alternatively map the first
        #       1 MiB as it can be generally useful, and there's no need to
        #       specially map the VGA buffer.
        movl $0, %esi
        # Map 1023 pages. The 1024th will be the VGA text buffer.
        movl $1023, %ecx

1:
        # Only map the kernel.
        cmpl $(_kernel_start - 0xC0000000), %esi
        jl 2f
        cmpl $(_kernel_end - 0xC0000000), %esi
        jge 3f

        # Map physical address as "present, writable". Note that this maps
        # .text and .rodata as writable. Mind security and map them as non-writable.
        movl %esi, %edx
        orl $0x003, %edx
        movl %edx, (%edi)

2:
        # Size of page is 4096 bytes.
        addl $4096, %esi
        # Size of entries in boot_page_table1 is 4 bytes.
```

```
        addl $4, %edi
        # Loop to the next entry if we haven't finished.
        loop 1b

3:
        # Map VGA video memory to 0xC03FF000 as "present, writable".
        movl $(0x000B8000 | 0x003), boot_page_table1 - 0xC0000000 + 1023 * 4

        # The page table is used at both page directory entry 0 (virtually from 0x0
        # to 0x3FFFFF) (thus identity mapping the kernel) and page directory entry
        # 768 (virtually from 0xC0000000 to 0xC03FFFFF) (thus mapping it in the
        # higher half). The kernel is identity mapped because enabling paging does
        # not change the next instruction, which continues to be physical. The CPU
        # would instead page fault if there was no identity mapping.

        # Map the page table to both virtual addresses 0x00000000 and 0xC0000000.
        movl $(boot_page_table1 - 0xC0000000 + 0x003), boot_page_directory - 0xC0000000 + 0
        movl $(boot_page_table1 - 0xC0000000 + 0x003), boot_page_directory - 0xC0000000 + 768 * 4

        # Set cr3 to the address of the boot_page_directory.
        movl $(boot_page_directory - 0xC0000000), %ecx
        movl %ecx, %cr3

        # Enable paging and the write-protect bit.
        movl %cr0, %ecx
        orl $0x80010000, %ecx
        movl %ecx, %cr0

        # Jump to higher half with an absolute jump.
        lea 4f, %ecx
        jmp *%ecx

4:
        # At this point, paging is fully set up and enabled.

        # Unmap the identity mapping as it is now unnecessary.
        movl $0, boot_page_directory + 0

        # Reload crc3 to force a TLB flush so the changes to take effect.
        movl %cr3, %ecx
        movl %ecx, %cr3

        # Set up the stack.
        mov $stack_top, %esp

        # Enter the high-level kernel.
        call kernel_main

        # Infinite loop if the system has nothing more to do.
        cli
1:      hlt
        jmp 1b
```

## linker.ld

This is a little trickier than it was for the bare bones tutorial, since you need to distinguish between virtual addresses (which will be in the higher half) and physical addresses (which GRUB needs to decide where to put your kernel).

```
ENTRY (_start)

SECTIONS
{
        /* The kernel will live at 3GB + 1MB in the virtual address space, */
        /* which will be mapped to 1MB in the physical address space. */
        /* Note that we page-align the sections. */
        . = 0xC0100000;
        /* Add a symbol that indicates the start address of the kernel. */
        _kernel_start = .;
        .text ALIGN (4K) : AT (ADDR (.text) - 0xC0000000)
```

```
	{
		*(.multiboot)
		*(.text)
	}
	.rodata ALIGN (4K) : AT (ADDR (.rodata) - 0xC0000000)
	{
		*(.rodata)
	}
	.data ALIGN (4K) : AT (ADDR (.data) - 0xC0000000)
	{
		*(.data)
	}
	.bss ALIGN (4K) : AT (ADDR (.bss) - 0xC0000000)
	{
		*(COMMON)
		*(.bss)
		*(.bootstrap_stack)
	}
	/* Add a symbol that indicates the end address of the kernel. */
	_kernel_end = .;
}
```

## kernel.c

Use kernel.c from Bare Bones with the following change. In boot.s we reserved the 1024th page for the VGA text buffer. This corresponds to virtual address 0xC03FF000. Change the initialization of `terminal_buffer` to:

```
	terminal_buffer = (uint16_t*) 0xC03FF000;
```


Category: Level 2 Tutorials