DiegOS

Diego Torres Gonzalez

February 20, 2024

Introduction

DiegOS is an Open Source project developed with the intention of experimenting with Rust language in a very low level environment and learning about basic operating systems concepts and techniques.

It is a basic kernel for the x86-i386 intel architecture. It implements a bootloader section of 512 bytes written in assembly code. This is the first code executed by the processor after the BIOS finishes the booting routines. The boot loader handles loading the rest of the kernel code into memory, setting up segmentation and taking the processor into 32-bit protected mode.

The other part of the project, which is written in Rust, gets called at the end of the boot loader section and is the main part of the kernel.

The kernel currently has the following features:

- Setting up an error and exception handling mechanism
- Driver for a IBM/AT 8259 Programmable Interrupt Controller (PIC) to handle hardware interrupts
- VGA frame buffer driver
- Standard output mechanism API to use this frame buffer
- Setting up Page Tables in memory to support 32-bit mode, 3-level paging with 4kb pages.

Motivation

For this project, I wanted to go as low level as I could while still being a wortwhile learning experience, considering the time it was going to take me and the expected learning outcome.

For me, the appropiate level was to start at the hand-off between the Basic Input/Output system (BIOS) booting routines and the not so specific boot loader and kernel code.

This decision was motivated on the fact that BIOS code is specific to every motherboard and involves dealing with the different hardware configurations a computer may have. This was not so interesting to me, and I consider that writing a BIOS is a different project to building an OS.

Operation of DiegOS

When the BIOS starts up, after doing the general initialization routines, goes through the available storage devices such as hard drives, usb's, disc drives, etc, one by one looking for the magic value *0xAA55* at the end of the first sector of the media. Whenever that magic value is found, the BIOS then knows that this is a boot sector and can start loading the OS by going to the first instruction on that sector.

This first sector of 512 bytes must prepare the hardware to start executing the main kernel code. In my OS, the boot sector serves to set up the stack, load the kernel code into memory, setting a Global Descriptor Table to enable segmentation, enabling 32-bit protected mode of operation for the CPU and jumping into the entry point of the newly loaded kernel.

The bootloader does all of this with the help of already provided routines by the BIOS to interact with the hardware. More specifically, these BIOS routines handle reading from the boot disk and putting output characters into the screen.

When the BIOS jumps to the first instruction of the boot loader, the stack is not set up yet and the size of the boot loader has to be exactly 512 bytes. This is the reason boot loaders are generally written in assembly code.

There is also standard boot sectors like the GNU GRUB, which is open source and conforms to the MultiBoot standard. This standard is intendend to facilitate having multiple operating systems on a single machine with a boot loader that lets you select the operating system to boot from.

DiegOS does not conform to the MultiBoot standard and has its own boot loader. I did it this way because I also wanted to learn about the boot loader part. Setting up a GRUB header does dome stuff for you and I didn't want that.

Configuration scheme for the CPU and memory layout

In this section I will describe how the CPU is configured in DiegOS, namely, the setup of segmentation and paging. I will also add the memory layout of the system once the initialization of the kernel is finished. For this, I will descibe briefly how these mechanisms work.

Segmentation

When the BIOS hands control to the boot loader, the CPU is in 16-bit legacy or real mode. In order to go from real mode to 32-bit protected mode, segmentation must be first set up.

Enabling 32-bit protected mode means the CPU will work with 32-bit memory addresses, enable extended registers and use segmentation to resolve physical addresses.

Physical addresses are the actual addresses on the ram chip and the processor needs to use these addresses to comunicate with the ram. On the other hand, there are logical addresses, which are of the form *A:B*.

The A part of the logical addresses is typically read from one of the segment registers of the CPU automatically. When segmentation is not enabled, the physical adress is computed by the operation $(A * 0x10) + B$. So A is a kind of base address and B is an offset from that address.

Enabling segmentation changes the mechanism used to resolve physical adresses from logical ones. In this case A is a pointer to an entry to what is called Global Descriptor Table (GDT). The GDT is formed by multiple entries. Each one of these entries describes a memory segment. The memory segment has a base and a limit address, a protection level and some other hardware configurations.

In this configuration, the CPU uses A to read an entry in the GDT. Before going any further, the CPU, through its Memory Management Unit (MMU), does hardware level permission checks and fires an exception if the executing code has no permission to access this particular address. When the checks are sucessful, it procedes to use the base address in the entry along with the offset B to compute the physical address in a similar fashion to what was earlier described.

The segments setup that DiegOS uses is the simplest and is described in the *Intel® 64 and IA-32 Architectures Software Developer's Manual* as the flat model. This model puts the code and data segments in the same physical addresses. This means that there is no real hardware memory protection at this point but it lets us address up to 4 GiB of physical memory and is needed to enable 32-bit protected mode.

Segmentation is a memory protection scheme that is now considered legacy and most modern OS's do not use it. What they use instead is Paging, another memory management mechanism that we will describe below.

Paging

The main advantage of Paging is that it creates the notion of linear or virtual memory. Virtual memory addresses are what is used by the code. The CPU does hardware-level translation of these addresses to their physical counterpart. It does this by means of a Page Directory Table (PDT), which is similar to a GDT, but has more levels of indirection, allows you to set up permissions for smaller regions of memory and to have data mapped to a virtual address stored in a secondary storage device (hdd, floppy, disc, etc) and only move it to ram when it is needed.

A PDT contains *1024* Page Directory Entries (PDE), each of these entries controls an area of memory of *4 MiB* in size. These entries describe if the region is present in physical memory, if it has been accesed before, if it can be cached, write and read permissions and some other stuff. They also contain a pointer to a Page Table.

A Page Table (PT) contains *1024* Page Table Entries (PTE). These entries are similar to the PDE's except they each control an area of *4 KiB*. Each of one of these *4 KiB* is called page.

Virtual addresses of 32 bits are divided into three parts. The most significant 10 bits are an index in the PDT, the next 10 bits are and index in the PT, and least significant 12 bits

are an offset in the page that is pointed to by the PDT and PT's.

When the processor needs to resolve a virtual address, it first goes to an index in the PDT, checks if that region is present and if the executing code is allowed to do that operation, and looks for the PT pointed by that PDE. It next goes to the specified index in that PT, does the same checks, and finally uses the PTE to read the base address of the page. Whenever any of the checks fail, the processor fires a Page Fault Exception. With the base address it then adds the offset to obtain the physical memory address. This process is called *page walk*.

The memory management mechanism I just described is called *32-bit 4KiB Paging* and is what I use in DiegOS. There is also other configurations for paging like using *4 MiB* pages insted of *4 KiB*. In the x86-64 platform, another level of indirection is added by default and can be extended to 5-level paging.

In the current configuration of DiegOS, paging is setup to do an identity map of the first megabyte of memory and to map physical addresses *0x100000 - 0xEFFFFF* to virtual *0xC0100000 - 0xC0CFFFFF*.

Identity mapping the first megabyte is the easiest thing to do because some of the addresses on this region are hardware-mapped and also the BIOS and Video BIOS code are stored there and are needed even after the kernel has taken over. It is inside the first megabyte that I store the kernel code running at the time of enabling paging, so it would cause trouble if I didn't have that identity mapped.

Interrupts

Interrupts are a way of asking the CPU to inmediately handle an event. There are 3 types of interrupts; exceptions, hardware interrupts and software interrupts.

An exception happens when the processor encounters a situation that should be handled by the kernel. Examples of this are division by zero, breakpoints, page faults, double faults and general protection faults.

The hardware interrupts occur when the Programmable Interrupt Controller signals to the processor that one of the connected hardware devices has registered an event. This could be a keyboard key press, a mouse click, etc.

Finally, software interrupts are trigered when the code running executes the special *int* instructions. These interrupts can be used freely by the kernel.

In DiegOS, the kernel loads an Interrupt Descriptor Table (IDT). The IDT contains entries that point to the appropiate interrupt handler routine for each interrupt code and also if the interrupt is masked. If an interrupt code is masked, it means the processor will just ignore any instance of that interrupt.

There is also a driver for configuring the standard IBM/AT 8259 PIC, which functions as a buffer between hardware devices and the processor.

Currently, I'm using the PIC to read input from the keyboard with a simple keyboard driver that I also wrote.

Memory layout

The first megabyte of memory is used by the BIOS, Video BIOS, Video Display Memory and some other stuff. A more detailed description can be found at the OS Dev Wiki.

The important parts for DiegOS are described in the table below using virtual memory addresses and approximate sizes.

Section	Start	End	Size
Boot Sector	0x7c00	0x7dff	512 bytes
Stack	0x7e00	0x8ffc	4.5 KiB
Main kernel code	0x9000	0x75fff	436 KiB
Page Tables	0x76000	0x80000	40 KiB
Currently usable memory	0xc0100000	0xc0effffff	14 MiB

The addressable memory can be easily extedend to 4 GiB. There is just currently no need for that much memory.

Project description

The main parts of the project are the following:

File / Directory	Description
bochsrc	Configuration file for the Bochs simulator instance.
bochsout	Bochs log file that is rewritten on every run.
Makefile	Build file containing all the automated recipes for building and running the project.
disk.img	Hard drive image that is created by $ make.
memory-map.txt	A description of the memory usage by the system.
dir:report	Directory containing this report's LaTeX source.
dir:build	Where temporary build files are written.
dir:src	All source files for the project are located here.

The src directory has 3 elements. The boot_loader directory, which contains the assembly code necessary for the boot sector. The Rust project for the main part of the kernel in the os_code directory. The linker script that tells the rust-lld and the GNU ld linkers the form of the desired output.

The boot_loader source files are arranged like so:

boot_sect.asm	Main source file of the boot sector. Sets up the stack loads kernel code into memory, switches CPU to 32-bit PM and jumps to entry point of the main kernel code.
gdt.asm	Contains the GDT that will be loaded into the GDTR register.
kernel_entry.asm	Simple assembly file that jumps to the entry point of the Rust kernel code. It is the first thing after the boot sector in the final hdd image.
switch_to_pm.asm	Auxiliary routine that loads GDT and goes to 32-bit pm.
dir:utility	Auxiliary files that provide routines for printing and loading code from hdd to memory.

The Rust project contained in the directory os_code has several configuration files and the source code.

The most important configuration file is *i386-pc-none-gnu.json*. This file specifies a custom build target and it tells the *rustup* toolchain about the architecture of the target. It specifies things like the endiannes of the target, the linker to use, the different data operations that are supported, etc.

The src directory is divided into modules. The main module just uses the rest of the modules to do the necessary initializations and to print some stuff to the screen. After the initialization is completed, the main thread just fires a panic to demonstrate the error handling mechanism and continues to receive input from the keyboard.

The drivers module is subdivided in two modules, the VGA Frame Buffer driver and the keyboard driver. The screen driver provides a standard output mechanism for the rest of the code.

The interrupts module is divided into three modules. The handler routines for all interrupts not currently masked, the code that sets up the Interrupt Descriptor Table (IDT) to handle interrupts correctly and the PIC driver, which allows us to interact with the hardware of the computer.

The mem module only has one submodule, paging. It handles seting up the page tables in memory and enabling paging.

Conclussion

Developing this basic kernel took me many days and nights and was a very energy draining project.

On the other side, it was very gratifying when things finally worked and I got a sense for how modern computers actually work from a very low level perspective.

This kernel is not finished. The initial goal was to have a file system, a user system and command line interface that would allow the user to execute programs and manipulate files.

I was too optimistic about this and had really no idea how much work that is.

If I ever wanted to continue this project to achieve the initial goal, it would require the following:

- Creating a system library available to user code that sets up a heap and is able to borrow memory to programs on that space. That is, a global heap allocator and an analogue to C's malloc.

  This would facilitate the use of Rust's boxed values and general dynamic memory utilization.

- Creating a file system along with permissions mechanisms.
- Separate memory regions into kernel regions and user regions.
- Improving keyboard driver.
- User permissions system.
- My own hdd driver to interact with hdd without needing to depend on the BIOS buggy hdd load routine.
- Enabling multithreading
- Writing a command line interface.

There is a lot of work to be done but I'm happy with the project so far.

References

- Allmost every article on OS Dev Wiki.
  https://wiki.osdev.org/Main_Page
- OS Phil blog showed me that it was actually possible to write a kernel in Rust.
  https://os.phil-opp.com
- https://littleosbook.github.io
- Intel® 64 and IA-32 Architectures Software Developer's Manual. Vol 3A, pt. 1.
  https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- And many Stack Overflow questions and obscure forum topics that I couldn't list.