About an OS-less Game

This document will explore the different parts of the game's code and the idea behind it. If you are interested in what happens here than hopefully this will be a guide on what not to do. A major reference for us was https://wiki.osdev.org/Main_Page. Be sure to explore.

The Boot Sector

Bochs is an emulator for a standard (if slightly dated) x86 computer. Meaning that if this was programmed correctly it should run on an actual computer. We however will give no guarantee as this was programmed using many bad assumptions due to lack of knowledge and time.

The bootloader makes the following assumptions:

There are 2gb of RAM. The disk is either the primary master or slave. The hard disk is ATA compatible and supports LBA 48.

The Bootloader program is loaded from the Boot Sector of a disk. That is the first sector on the disk, with each sector being 512 bytes. Even if that is not true for a specific disk, most all disks emulate the 512 byte sectors and will expect to be interfaced in that way. This is loaded at the flat address 0x7c00. In our program we use the ORG directive to align with this. This directive offsets all the fixed addresses by 0x7c00, giving them their fixed linear address instead of a relative address. This is important to note as this code will not run at all if relocated. Upon loading we are in 16-bit real mode. That is the computer only has access to roughly 1mb of RAM and there is no process protection. In this mode it is possible to access 32-bit registers.

It should be noted that on a partitioned disk, or a disk with a file system at all, the bootloader often shares its space with header data or a partition record table such as MBR or GPT. The disk generated for this game is a flat binary blob with no structure.

Our program begins by ensuring the data segment is set to 0 so the linear addressing that nasm will generate for the labels work correctly. We then set the stack pointer to the address 0xFFFF and the string direction flag is cleared.

Text Functions

The first function seen is the cls and sprint function. Upon first boot the BIOS has set up the display to be in text mode. This is akin to DOS or the command prompt. Every character is represented with 2 bytes as shown in the following figure. There are 16 colors available.

Byte 1		Byte 0
Background	Foreground	ASCII Character Code

The text memory starts at 0xb8000 and every character from the top left moving right, wrapping back to the left, is stored linearly here. For both cls and sprint, 0xb800 is moved into the es as segment registers are multiplied by 16 before adding the offset. The cls function then clears the screen by zeroing the entire memory. While the sprint function takes c style strings and writes them to memory. It was also made to handle Unix style new lines.

Disk Functions

As only the first 512 bytes are loaded into memory, it is necessary to load the rest of the data. What follows is the complete and absolute wrong way to go about this. As libraries could not be used this section was written on many incorrect assumptions. After it was written, an actual bootloader was referenced and it uses the BIOS while I did not. The BIOS method would be desirable as it is much more compact, saving precious bootloader space.

My approach was to first locate the disk by addressing a common port for the ATA drive controller. I then sent it an IDENTIFY command. The reference for these can be found here: https://wiki.osdev.org/ATA_PIO. This is all under the checkdrive section. By line 95 (after setting the ATA controler to operate in LBA 48 mode), the device has been located and set up. As a note, LBA is Logical Block Addressing and is addresses the drive via sectors rather than bytes. There is a legacy addressing method called CHS that is best left to google to explain. It is not difficult, rather it is just tedious.

Now is for the slightly clever bit (that people have thought of before me, see the bootloader page describing what a hybrid bootloader is). By placing more code directly after the boot sector and having it all compile under the same ORG directive, if we load it into memory right after the bootloader, it will act as a single program. The read_sector function as implemented here only accepts a 32-bit LBA address, but that is well past our needs. It sends the address to the appropriate registers, high byte first, and never the same port twice. The ATA PIO page mentions sending to the same port twice greatly reduces performance. When waiting on the status of the ATA drive, it is important to have a 400ns delay as it will take a moment for the controller to update. After the remaining program is read, the magic id on the other end is checked against the one on the boot sector. If they match, a jump is made into the main bootloader program.

Leaving Real Mode

For our game we wanted to at least have access to 32-bit addressing and extended memory so that we load images. For this reason we had to leave 16-bit real mode. This all takes place in stage_2. First we tell the processor where the gdt descriptor is in memory by using the lgdt commad. The gdt descriptor points to the gdt. The gdt being a description of segments of memory, what permissions they have, and what modes the operate in. Every gdt starts with a null entry. Ours then has a 32-bit code segment that takes up a 4gb space starting from 0. We then have a data segment with the same parameters. And then 2 16-bit segments that only cover the bootloader. This is called a flat segment table since all segments overlap and use all the space. For more information see https://wiki.osdev.org/GDT and https://wiki.osdev.org/GDT_Tutorial. This paired with a local gdt is how processes can have virtualized memory spaces. At least that is my understanding.

After loading the gdt we need to enable the a20 gate. The a20 gate is a hold over from an early iteration of the x86 line. Some early programmers relied on the fact that the memory wrapped from the end to the beginning because the address space was bigger than available memory (around 1mb). To make sure programs with this very bad assumption still worked, the a20 gate starts disabled. It now stands as the gateway between the system programmer and extended memory (anything above 1mb). Note that you still do not have full access to memory while in 16-bit real mode. There are multiple ways for it to be activated and not all of them work on all systems. So multiple methods are used in out implementation. See https://wiki.osdev.org/A20.

Just before we move to protected mode, we use a BIOS interrupt to set the video mode. We have it set to VGA 320x200 256 color mode. This mode allows for linear addressing to the video memory. Most other modes do not have linear addressing and instead have a multi page structure. See https://wiki.osdev.org/Drawing_In_Protected_Mode.

Finally we switch to 32-bit real mode by setting the control register bit. 64-bit mode can be set in a similar way, but that is excessive for our purposes and requires a different format for the gdt and such. We then do a far jump to the start of protected mode code. This sets up the code segment register with the correct values. The BITS 32 directive is used here to make sure the opcodes are compiled to their 32 bit variants instead of the 16-bit versions. Since the processor is now looking for 32 bit opcodes, the functions defined earlier in 16-bit land no longer function. They also were using different addressing methods that do not work in 32-bit protected mode. Now, instead of the segment registers holding an address, they now hold an offset into the gdt. This tells them how to operate and what they can and cannot do.

Afterwards we set up the segment registers, move stack to the 1gb point. It is assumed 1gb is reserved. Under normal circumstances a memory map is done, usually via a BIOS call, and that is used. However for simplicity and thanks to the controlled environment it is known that 1gb is the end of memory. We set the destination of sector_read32 to the 16mb part of memory as to avoid any potential reserved addresses that would have shown up in a memory map. The 8th sector, the one just past the bootloader, has a number at the beginning that tells us how many sectors the main program is. It then reads in that many sectors. Afterwards we far jump into the main program as it has been loaded to a known address.

Main Program

This document will only go over the technical aspects of the main program as after those are covered, it is just normal programming logic. Everything starts in main.asm. Here we see the ORG directive again. Since we are loading it to a fixed memory location, it is needed as the gdt was not setup for a segment to call this part address 0.

Keyboard

The keyboard in this case is a PS2 keyboard and is communicated with the PS2 controller. The controller has a command/status port specific to it on 0x64 and a data IO port at 0x60. Now let me briefly mention that we have again made very bad assumptions here. Bochs inits the controller and keyboard under scancode page 2. That is the encoding the keyboard will use to indicate what key has been pressed. It is not in ASCII and needs to be mapped per keyboard layout. However bochs seems to be be sending scancode page 1 encodings. This encoding is legacy and no longer used. This could be due to a few reasons. Either there is a glitch with how bochs is handling the keyboard, or it is actually an emulated PS2 controller and this is an emulated glitch you may find with them. Under normal circumstances you will start a device discovery search and disable any emulated PS2 devices and manually initialize the controller. For our project we assumed it will report scancode page 1 and went with that. Also, the keyboard should be handled with interrupts, but they were disabled during the boot loader and enabling them causes issues. This is most likely due the bad assumptions and due to how poorly written the boot loader is compared to a proper one. Instead we poll it by reading the data input buffer and handling the scancodes as they read in. Our implementation is only configured to detect if a key is pressed or not, which is all that is needed for a game. If a status or data is expected, it is necessary to wait for the ready bits to be set (or cleared). See https://wiki.osdev.org/PS2_Keyboard.

Graphics

Since we switched graphics modes it is now possible to draw images to the screen. All images used in the game have been preprocessed and converted into a raw bitmap that is defined as a variable in an asm file. Most images are in char.asm. This is primarily due to the fact that this drive lacks a file system. So all assets are loaded into memory at the same time and nasm assigns everything the correct address so they can be referenced when included. The graphics mode memory is located at 0xA0000. Each pixel is 1 byte. So all images have been using a constrained color palette. Animations and game play are dependent on how well bochs runs on your system. This is due to the frames not being fixed time dependant.