COP 6611: Operating Systems

Project 0 Pre Reading - The Bootloader

---

1. x86 Assembly

Before getting started on this assignment, please read the x86 Assembly Guide to review the basics of x86 assembly. If you are not already familiar with x86 assembly language, you will quickly become familiar with it during this course!

Please also read Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT&T assembly syntax, we'll be using with the GNU assembler in this OS. It also explains how to use x86 assembly inline in C code.

Other Helpful Reading Materials on Assembly:

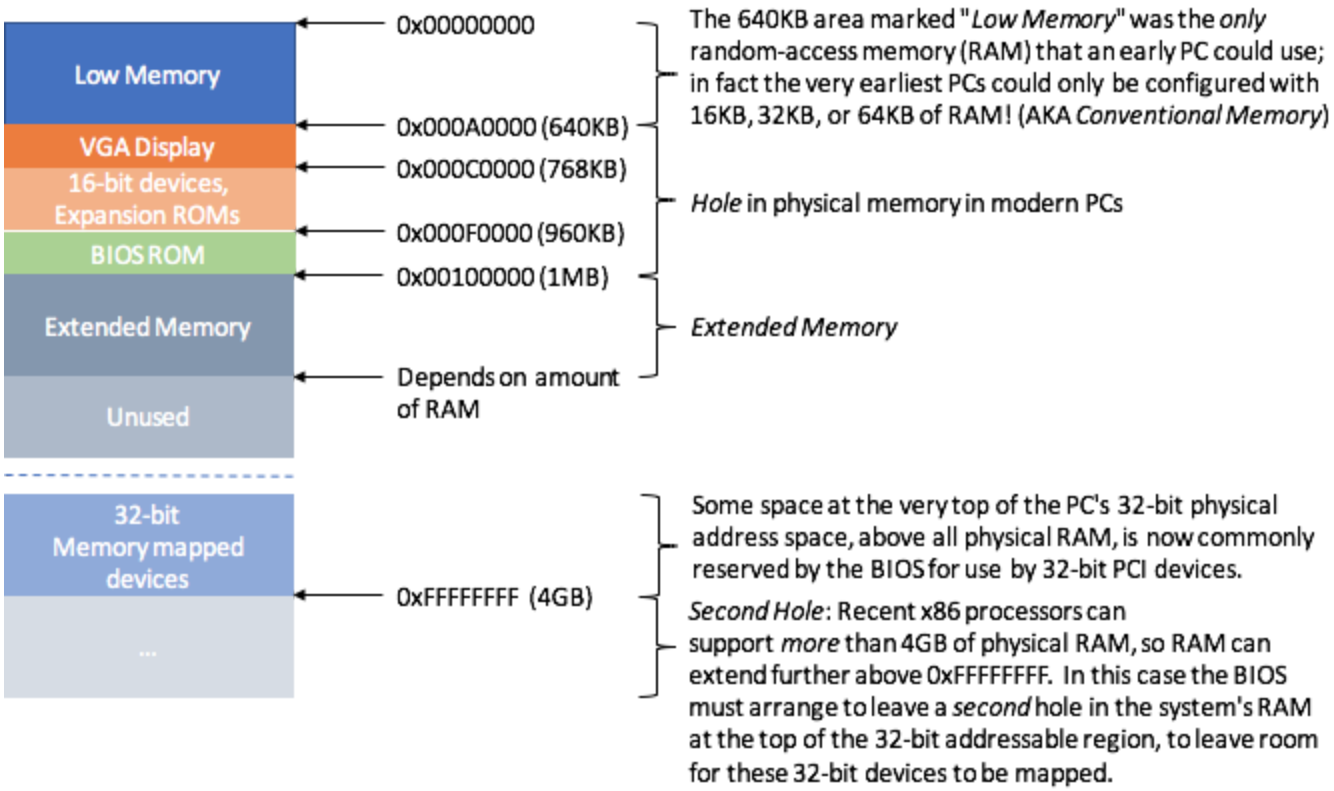
- PC Assembly Language Book
  Warning: Unfortunately the examples in the book are written for the NASM assembler, whereas we will be using the GNU assembler. NASM uses the so-called *Intel* syntax while GNU uses the *AT&T* syntax. While semantically equivalent, an assembly file will differ quite a lot, at least superficially, depending on which syntax is used.
- Assembler Directives
- Certainly the definitive reference for x86 assembly language programming is Intel's instruction set architecture reference, which you can find in two flavors: an HTML edition of the old 80386 Programmer's Reference Manual, which is much shorter and easier to navigate than more recent manuals but describes all of the x86 processor features that we will make use of in COP 6611; and the full, latest and greatest IA-32 Intel Architecture Software Developer's Manuals from Intel, covering all the features of the most recent processors that we won't need in class but you may be interested in learning about. Save the Intel architecture manual for later or use them for reference when you want to look up the definitive explanation of a particular processor feature or instruction.

2. The PC's Physical Address Space

The first PCs, which were based on the 16-bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical address space of an early PC would therefore start at `0x00000000` but end at `0x000FFFFF` instead of `0xFFFFFFFF`. The 640KB area marked "Low Memory" was the *only* random-access memory (RAM) that an early PC could use; in fact the very earliest PCs could only be configured with 16KB, 32KB, or 64KB of RAM!

The 384KB area from `0x000A0000` through `0x000FFFFF` was reserved by the hardware for special uses such as video display buffers and firmware held in non-volatile memory. The most important part of this reserved area is the Basic Input/Output System (BIOS), which occupies the 64KB region from `0x000F0000` through `0x000FFFFF`. In early PCs the BIOS was held in true read-only memory (ROM), but current PCs store the BIOS in updateable flash memory. The BIOS is responsible for performing basic system initialization such as activating the video card and checking the amount of memory installed. After performing this initialization, the BIOS loads the operating system from some appropriate location such as floppy disk, hard disk, CD-ROM, or the network, and passes control of the machine to the operating system.

We will now dive into a bit more detail about how a PC starts up. A PC's physical address space is hard-wired to have the following general layout:



When Intel finally "broke the one megabyte barrier" with the 80286 and 80386 processors, which supported 16MB and 4GB physical address spaces respectively, the PC architects nevertheless preserved the original layout for the low 1MB of physical address space in order to ensure backward compatibility with existing software. Modern PCs therefore have a "hole" in physical memory from `0x000A0000` to `0x00100000`, dividing RAM into "low" or "conventional memory" (the first 640KB) and "extended memory" (everything else). In addition, some space at the very top of the PC's 32-bit physical address space, above all physical RAM, is now commonly reserved by the BIOS for use by 32-bit PCI devices.

For our OS, we assume that all PCs have "only" a 32-bit physical address space (thus can support up to 4GB of memory). But dealing with complicated physical address spaces and other aspects of hardware organization that evolved over many years is one of the important practical challenges of OS development.

2.1 CPU Real Mode

The PC starts off in Real Mode, which is characterized by a 20-bit segmented memory address space (i.e., 1MB of addressable memory), like the first PCs mentioned above. Memory access is done using segmentation via a *segment:offset* system. There are six 16-bit segment registers (CS, DS, ES, FS, GS and SS). Address translation is done according to the formula:

```
Physical address = 16 *  segment + offset
Example:
      16 * 0xf000 + 0xfff0      # in hex multiplication by 16 is
         = 0xf0000 + 0xfff0     # easy--just append a 0.
         = 0xffff0
```

All modern operating systems run in *Protected Mode*, due to the limitations of the Real Mode, which include limited (1MB) memory access, no support for security mechanisms (memory protection), multitasking or code privilege levels. But until an OS (or bootloader) has a disk driver loaded, the only way to access disks is through BIOS functions, which are available only in the Real Mode. For our assignment, we will use protected mode, and configure all the default segments to overlap and cover 0-4GB. This will effectively disable segmentation by creating a flat memory space. We will instead implement paging in a later assignment.

3. The BIOS

The IBM PC starts executing at physical address `0x000ffff0`, which is at the very top of the 64KB area reserved for the ROM BIOS. Because the BIOS in a PC is "hard-wired" to the physical address range `0x000f0000-0x000fffff`, this design ensures that the BIOS always gets control of the machine first after power-up or any system restart - which is crucial because on power-up there is no other software anywhere in the machine's RAM that the processor could execute.

When the BIOS runs, it sets up an *Interrupt Vector Table* (IVT), typically from physical address `0x00` to `0x3ff` and detects, enumerates, configures, and initializes the PCI bus and various devices such as the VGA display. After initializing the PCI bus and all the important devices the BIOS knows about, it searches for a list called the *boot-sequence* stored in CMOS for a bootable device such as a floppy, hard drive, or CD-ROM. Eventually, when it finds a bootable disk, the BIOS reads the *boot sector* from the device and transfers control to it. If it does not find a valid boot sector, it will stop and throw an error.

3.1 BIOS Functions

BIOS functions perform hardware control or I/O functions requested by a program, return system information to the program, or do both. They are executed via software interrupts (`INT` instruction), with a command number stored in a CPU register, generally the `AH` register (sometimes the `AX` or `EAX` are used instead, see here for more details).

- `INT 0x10` = Video display functions (including VESA/VBE)
- `INT 0x13` = mass storage (disk, floppy) access
- `INT 0x15` = memory size functions1
- `INT 0x16` = keyboard functions

3.2 Reading Disk

`INT 0x13` provides a sector based hard disk (and floppy disk) read and write services. Setting command number in AH register to `0x42` allows us to read sectors from drive. This command takes as argument a pointer to a *Disk Address Packet* (DAP), in the form of `DS:SI` register pair. The DAP structure can itself be set in memory using the stack. The DAP format is as follows:

Offset	Size (bytes)	Description
0	1	Size of DAP (16 bytes)
1	1	Unused. Set to 0
2	2	Number of sectors to read
4	4	Destination Address (segment:offset pointer to memory buffer to which sectors will be transferred). NOTE: x86 is little endian.
8	4	Lower 32-bits of 48-bit starting LBA
12	4	Upper 16-bits of 48-bit starting LBA

Logical Block Address (LBA): Common scheme for specifying the location of blocks of data on a disk.

3.3 Memory Map

The kernel doesn't know which regions of physical memory are available for use at the start, so the bootloader also needs to detect the complete physical memory map and pass it to the kernel. This is done using the `INT 0x15` BIOS function with command number `0x0e820`. This process is [described here].

4. PC Bootstrap Process

Floppy and hard disks for PCs are divided into 512 byte regions called *sectors*. A sector is the disk's minimum transfer granularity: each read or write operation must be one or more sectors in size and aligned on a sector boundary. If the disk is bootable, the first sector is called the *boot sector*, since this is where the boot loader code resides.

4.1 Hard Disk and the Master Boot Record (MBR)

The BIOS identifies a hard disk as bootable if the first sector ends with the MBR boot signature (`0x55AA`). This first sector (LBA 0) of the hard disk is it's bootsector and is called the Master Boot Record (MBR). It is the traditional way of storing partition information about a hard disk (*partition table*) and the boot code (*boot loader*). MBR is written to the first sector of the hard disk when a partitioning software (such as FDISK) partitions an empty drive for the first time and it is not a part of any partition but is written on the "raw device". When new partitions are created, the partition table is updated with the relevant information. The partition table is independent of the operating system and has a standard layout.

When multiple partitions exist within a hard disk, the conventional MBR code checks the partition table to find the partition that is marked as *bootable*. If a *bootable* partition is found, the MBR will load and execute the boot sector from that partition, called the *Volume Boot Record* (VBR).

The conventional MBR only allows for a single bootable partition. It can be replaced by a custom dual booting MBR, which allows the user to select any partition on the current drive to boot from.

The ability to boot from a CD-ROM came much later during the evolution of the PC, and as a result the PC architects took the opportunity to rethink the boot process slightly. As a result, the way a modern BIOS boots from a CD-ROM is a bit more complicated (and more powerful). CD-ROMs use a sector size of 2048 bytes instead of 512, and the BIOS can load a much larger boot image from the disk into memory (not just one sector) before transferring control to it. For more information, see the ["El Torito" Bootable CD-ROM Format Specification].

However, we will use the conventional hard drive boot mechanism, which means that our boot loader must either fit into a measly 512 bytes, or split into two parts, and let the first part load the second one.

When the BIOS finds a bootable floppy or hard disk, it loads the 512-byte boot sector into memory at physical addresses `0x7c00` through `0x7dff`, and then uses a `jmp` instruction to set the CS:IP to `0000:7c00`, passing control to the boot loader. It also loads the CPU register DL with the *drive number*. Like the BIOS load address, these addresses are fairly arbitrary - but they are fixed and standardized for PCs.

The memory map after the BIOS loads the MBR can be seen here:
http://flint.cs.yale.edu/feng/cos/resources/BIOS/mem.htm.

4.2 CPU Protected Mode

Next, the bootloader needs to switch to protected mode. Protected mode is the main operating mode of PCs and allows working with several virtual address spaces, each of which has a maximum of 4GB of addressable memory (compared to 1MB in real mode) and also supports strict memory and hardware I/O protection. It also has support for restricting the available instruction set to user programs. For more information about protected mode read the Protected Mode Tutorial.

Segmentation address translation in protected mode is different than in real mode. The address is still represented by the *selector:offset* pair, but the selector is different and has the following format:

15 ... 3	2	1	0
Index	TI	RPL	

`RPL` = Requestor Privilege Level
`TI` = Table Indicator: Index into GDT if `TI` = 0, otherwise to LDT
`Index` = Index into table

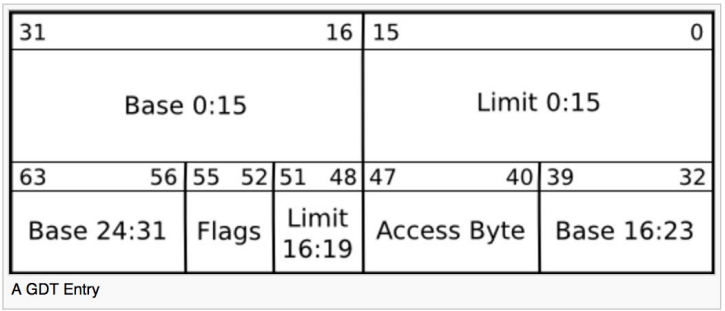
The privilege protection is provided for segment and is how protected mode gets its name. The *index* portion of the segment register (or *selector*) is an index into either the *Global Descriptor Table* (GDT) or a *Local Descriptor Table* (LDT), depending on the TI bit, to obtain a *segment descriptor entry*, which contains the base address (32-bit) of the segment, the length (20-bit) of the segment (AKA limit), privilege level needed to access the segment, and other flags. The lowest two bits of the selector (RPL) represent the privilege level of the process making the memory access request.

We have to set up a few things before we can switch to protected mode:

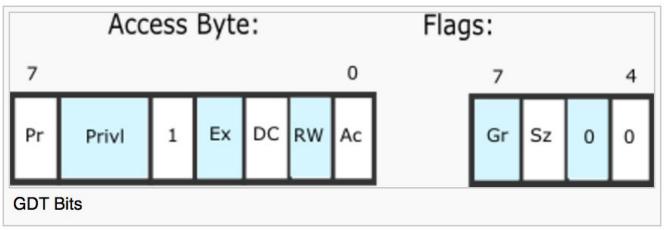
- Disable interrupts
- Setup Global Descriptor Table (GDT)
- Enable A20 line

Global Descriptor Table (GDT)

GDT is specific to IA32 architecture. Each GDT entry is 64-bit long (8-bytes) and contains the base address, size, access privilege, and other necessary flags. The structure of each GDT entry is shown below [1]:



A GDT Entry



GDT Bits

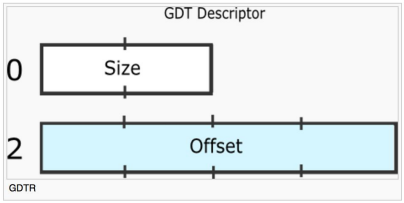
The Base address, spread into three parts in the GDT entry, is a 32 bit value containing the linear address of the beginning of the segment. Limit is a 20 bit value (spread into two parts in the GDT entry) indicating the maximum addressable unit. The 20-bit limit can only represent 2^{20} = 1MB of memory. But the descriptor entry has a *Granularity (Gr) bit* to indicate whether the limit uses 1 byte (0) or 4KB (1) for one unit. Hence with the bit set, each segment can address a maximum of 2^{20} * 4KB = 4GB memory.

Segment based protection cannot be disabled on x86. But for our assignments we will bypass it by setting all the descriptor entries in the GDT to have the same base address.

The entries in the GDT are:

- **Null Descriptor:** The first GDT entry is reserved by the processor, is set to zeros and cannot be used.
- **Code Segment Descriptor:** We will set the second GDT entry to have a base address of `0x00000000` and the limit to `0xfffff`. The *executable* (`Ex`) bit also has to be set for the code segment. The selector will have to be set up to use GDT (TI=0) with kernel level permissions (RPL=00) and with index set to 1.
- **Data Segment Descriptor:** We will use the same values for this descriptor, with the only difference being the *Ex* bit equal to 0. The selector for this descriptor will have index 2.

The hardware locates the GDT by using the special GDTR register. GDTR holds 6-bytes of data which includes the size and the base address of the GDT. A special instruction (`lgdt`) is used to load this register [1].



A20 Line

A20 line is the physical connection of the 21st bit (counting from 0) of a memory address in the system bus on an x86. Originally, the A20 gate was a part of the keyboard controller which was used to open or close it as necessary. To maintain backward compatibility, this line is kept closed by default. But modern OS needs to enable this line to be able to use the full 32 bits of the bus for memory addressing. Note: Intel no longer supports the A20 gate on their newer 64 bit processors (since Haswell). There are several different ways to enable it. We have already implemented this in the skeleton code for the first assignment for you.

We disable maskable hardware interrupts using the `CLI` instruction.

After all the setup is complete, to enter protected mode, we simply set the PE flag in the control register `CR0`. There are two ways in which `CR0` register can be read and written to:

Read:

```
mov %cr0, %reg
smsw operand# Stores the machine status word (bits 0-15 of CR0) into the destination operand.
              # The destination operand can be a 16-bit general-purpose register or a memory location.
```

Write:

```
mov %reg, %cr0
lmsw %reg     # Loads the source operand into the machine status word (bits 0-15 of register CR0).
              # The source operand can be a 16-bit general-purpose register or a memory location.
```

Then we long jump (`ljmp`) to the code segment with the offset of the desired 32-bit code segment.

References

1. Global Descriptor Table. (n.d.). Retrieved August 28, 2017, from http://wiki.osdev.org/GDT
2. Shao, Z. (2017). CS422/522 Lab 1: Bootloader & Physical Memory Management. Retrieved January 2, 2019, from http://flint.cs.yale.edu/cs422/assignments/as1.html