Operating Systems

Being assigned to handle PGT200 - an undergraduate course on Operating Systems have grown some interest in me to explore more details (I only had one course on this topic while doing my B.Eng. - and that was it!). I just thought I should keep a note of this just in case I want to do more of this in the future.

Most of the information here will be about implementations on standard PC operating systems, but my interest would be to implement it on machines related to embedded systems.

Background Info

Booting a system (a more technical term would be bootstrapping):

- usually BIOS on non-volatile memory (ROM or NVRAM)
  - Memory location 0xFFFFFFF0 on x86 systems? (0xFFFF0 on 8086/8088)
  - on some sytems, power-related stuffs may run before handing over to BIOS?
- handles/prepares low-level hardware/software interface
  - bios interrupt service (interrupt vector table): 10h (display), 13h (disks), 16h (keyboard)
  - stack pointer is 512 bytes after bootsector (stack size!)?
  - 512-bytes sector was standard disk sector size
- mostly bypassed by modern 'plug-n-play' operating systems
  - PnP OS writes their own service routine (overwrites IVT?)
- loads sector 0, cylinder 0 of selected boot drive
  - first sector (512 bytes) will be loaded to 0x7C00 (segment 0)
  - some bios used 0x07C0:0x0000 (same physical, different segment!)
  - execution transferred to that location
  - DL register holds boot drive number (e.g. 0x80 for first hard disk)
  - some BIOS will NOT transfer if INVALID MBR (e.g. no boot signature 0xAA55)
- the code in sector 0 (first sector) is usually the bootloader
  - for a partitioned disk, there should be an MBR (classic MBR? 'modern' MBR?)
  - for non-partitioned disk (e.g. floppy) simply the bootloader?
  - enables multiple kernel to be selected at run-time
- can be skipped for static single kernel usage? coreboot?
- loads the kernel specified in its configuration

Master Boot Record

- first sector of a partitioned storage
- classic mbr:
  - 446 bytes executable code
  - 64 bytes partition entries (4 primary partitions)
  - 2 bytes boot signature (0x55 0xAA)
- modern mbr
  - 218 bytes executable code

    - 2 bytes (always 0x00?)
    - 4 bytes disk timestamp
    - 216 bytes executable code
    - 4 bytes disk signature
    - 2 bytes (always 0x00?)
    - 64 bytes partition entries (4 primary partitionss)
    - 2 bytes boot signature (0x55 0xAA)
- superseeded by GUID partition table (GPT) - Read more @ wikipedia

Read more @ wikipedia

Bootloader Program

This is a simple tutorial on how to develop a simple bootloader for x86 compatible machines, meant to be used with a USB drive.

Using QEMU for Testing

It would be a nightmare to test a bootloader code using an actual USB drive booting an actual machine unless you have other machine than the one you are using for writing/building the code. The best way to do this is using a virtual machine and I recommend QEMU for this.

First we create a 256MB USB disk image using `qemu-img`

```
qemu-img create -f raw usb_drive.img 256M
```

Although not necessary, most USB drives nowadays are detected HDD and thus BIOS expects an MBR (I'm not going into GPT for now). To create proper partition table in MBR with a single bootable W95 FAT32 (LBA) partition I use `fdisk`

```
fdisk -u usb_drive.img
```

A 'faster' way would be to simply type

```
fdisk -u usb_drive.img <<EOF
n
p
1


t
c
a
1
w
EOF
```

Notice there are two newline characters (hit ENTER twice) between '1' and 't'. The `fdisk` version I'm using writes a random (at least I think it's random) 32-bit disk signature at location 0x1b8 that is optional and can be overwritten. Use

```
hexdump -C usb_drive.img
```

to verify 0x55, 0xAA sequence at location 0x1fe and 0x1ff respectively.

This is the part which is previously NOT needed due to the fact that floppy disks do not need partition tables (non-partitioned storage). Since the bootloader code we developed is exactly 512 bytes, it is obvious that the code will overwrite the partition information in the MBR. To avoid this we only write 446 bytes instead on the full 512 bytes, which is fine since the 0x55 0xAA sequence is already there. Due to the fact that we are writing to a disk image, we need to add `conv=notrunc` option so that the image file will NOT be truncated. (This is not a problem when the target is a device file!)

```
dd if=loader.bin of=usb_drive.img bs=1 count=446 conv=notrunc
```

Finally... to test the USB disk image

```
qemu-system-i386 -hda usb_drive.img -boot c
```