The MemOS Simple Memory System

By Rich West, Boston University.

**** THIS PROJECT CAN BE PERFORMED IN GROUPS OF TWO ****
If working in a group of two, only one person is required to submit a solution. In a README file, please include the name of the other person.

Background

In this assignment, you are going to write a very simple OS. Okay, it's not a particularly useful OS but it will provide a way to understand how systems are booted, and how system information is displayed on the screen. The idea is to bootstrap a program that probes the system BIOS and reports the amount of physical memory available in your machine. This is then displayed as a message, in the form:

"MemOS: Welcome *** System Memory is: XXXMB"

The value XXX is replaced by the actual memory your system has available.

You should also give a breakdown of the ranges of physical memory and their type, where the type is USABLE RAM, RESERVED, ACPI RECLAIMABLE MEMORY, ACPI NVS MEMORY, or BAD MEMORY according to the flags reported by the BIOS. You can find more on the osdev page for the five types of memory reported using BIOS INT 0x15, EAX=0xE820. For each memory range you should output a line to the screen of the form:

"Address range [xxxx : yyyy] status: zzzz" where xxxx is the start address, yyyy is the end address and zzzz is the type of the memory in that range.

Step 1: Building a Virtual Disk

The first step is to build a virtual disk for MemOS. To do this you should follow the general guidelines I provided for creating a simple disk image for use in BOCHS. If you do not have BOCHS, you can replace `bximage` with either `dd` or `qemu-img`. The latter requires you to have QEMU available.

An example using `dd` instead of `bximage` to create a virtual disk file is as follows:

```
$dd if=/dev/zero of=disk.img bs=1k count=32760
```

Here, we simply use the pseudo-device `/dev/zero` to fill an output file, `disk.img`, with zero'd bytes, with a block size of 1024 and a count of blocks equal to 32760. This will create a file whose size is 32760*1024 bytes. For larger or smaller files, you can choose different count values. Similarly, you can change the block size as it's not particularly important unless we're dealing with a real disk device.

If you have `qemu-img`, you can create a raw disk image using the following command as an example:

```
$qemu-img create -f raw disk.img 32760K
```

In choosing the size of your virtual disk, you should be aware of disk geometries. In a real disk, at least older ones based on CHS geometries rather than logical block addressing (LBA), the size is calculated as:

cylinders * heads * sectors * sector-size

This is equivalent to:

cylinders * (tracks / cylinder) * (sectors / track) * sector-size

Let's assume that we're going to adopt the default sectors-per-track value for DOS compatibility. This is 63.
Also, let's assume we have a complete geometry as follows:

Cylinders = 65
Heads = 16 (same as tracks / cylinder)
Sectors = 63 (actually, sectors / track)
Sector-size = 512 bytes

This gives us a disk size of:

65 * 16 * 63 * 512 = 32760KB (where 1KB is 1024 bytes)

Once we have a raw virtual disk partition file, we can start to properly configure its geometry and its filesystem. Then, we can install a bootloader.

You should follow steps 2 onwards in the BOCHS HOWTO to create your formatted disk image. Step 5 is only required to install GRUB, which is not necessary unless we use this later to boot our OS code.

NOTE: if you use the geometry settings above (Cylinders=65, Heads=16, Sectors=63), make sure you use those in Step 2 of the BOCHS virtual disk HOWTO, and also later when using the GRUB shell.

If installing GRUB, we will assume the bootloader is based on version 1 (GRUB legacy) rather than GRUB2. You will need to copy `stage1`, `stage2`, and `e2fs_stage1_5` to a `/boot/grub` directory on your virtual disk, as described in the HOWTO step 5. Then you will need to install stage1 in the master boot record (MBR) region of your disk image, using the interactive grub shell. Once successfully installed, you are ready for Step 2...

Step 2: Writing the MemOS Code

Here, you will have to write an x86 assembly program, called `memos-X.s`, where X is replaced with "1" or "2" depending on the version (described later). To help, I have provided a test program called `vga16.s`, written for use with the GNU assembler, `gas`. You should study `vga16.s` to see how it works. Intel's Software Developers Manual Volume 2 (Instruction Set) is helpful here.

Notice how the size of vga16.s is limited to 512 bytes. It's actually possible to load this

code, after assembly and linkage into the MBR of your disk partition and treat it as a bootable program. This is because it has a valid boot signature 0xAA55 in the last two bytes of the 512 byte sector.

To assemble and link your `memos-X.s` program, you will need to follow the instructions at the bottom of `vga16.s` as a guideline. What is missing is the linker script to complete the linkage of your program. Here, I provide the linker script for `vga16.s`. You should read the info or man pages on GNU `ld`, which is part of the binutils package, to understand the format of linker scripts. Notice how, at the bottom of `vga16.s` we use `dd` again, to create a sector image of 512 bytes that will fit in an MBR if desired. Here, we skip the first 4096 bytes to bypass the object file program header, generated by `ld`. This is because the assembler (`as`) and linker (`ld`) produce an output file in ELF binary format and what we really want are just the program sections if we're to map this code into an MBR.

Now, let's backtrack a moment. It is possible using a PC emulator such as QEMU to boot the resultant binary, `memos-X`, without mapping it to a virtual disk. To do this, we can assume it is written jut like `vga16_test` (created from `vga16.s` using `dd`). We can execute `memos-X` as follows:

`$qemu -hda memos`-X (depending on your version of qemu, you might need to use qemu-system-i386 or something similar for the binary name)

The -hda flag, above, is actually optional here and won't really affect anything as it stands.

First Deliverable

For the first solution to the problem of detecting memory, you should produce a `memos-1.s` version that can run as standalone boot code. Your solution will use BIOS interrupts to detect memory and write it to the screen. For this you will need to understand:

- the BIOS
- ways to use the BIOS to detect memory
- how to use the BIOS to write to the VGA display. Here, you should use INT 0x10 interrupts, similar to how printing is done in `vga16.s`.

The next test is to show that you can boot your memos-1 code assembled and linked into the MBR of a virtual disk. We will not expect you to submit a virtual disk image as they are relatively large. Instead, you should submit a README-1 file that documents exactly the steps taken to get your memos-1 binary into the virtual disk's MBR, so that it can execute to produce the output strings at the top of this document. Specifically, the messages:

"MemOS: Welcome *** System Memory is: XXXMB"
"Address range [xxxx : yyyy] status: zzzz"
"Address range..." etc

You should use gsubmit to submit all your files (discounting the virtual disk image) to a subdirectory called memos. Include the linker script, memos-1.s files, a Makefile and README-1.

Second Deliverable

Here, you will create a file called `memos-2.s` and appropriate linker scripts to generate a binary that is booted by GRUB. Here, we will assume *legacy* GRUB rather than GRUB version 2. You will need to go back to the virtual disk creation described above and install GRUB in your MBR (or, alternatively, create a second virtual disk and install GRUB on that). This is documented in step 5 of the BOCHS virtual disk HOWTO. Then you will need to add a file called `menu.lst` to `/boot/grub` on your virtual disk. To do this, make sure you have mounted your virtual disk as a loopback device. For example, you can do this as follows:

```
$mount -t ext2 disk.img /mnt -o loop,offset=32256
```

The offset, above, is because the ext2 filesystem begins after the first track on your virtual disk. If you have formatted your disk to have a sectors-per-track value of 63 (the DOS compatibility value) then the first track has a size of 63*512 bytes, which is where the 32256 value comes from. To make sure you use 63 sectors/track when creating `disk.img`, be sure that your version of `fdisk` is running in DOS compatibility mode. If it's an older version, it will automatically assume 63 sectors/track. If you get a value such as 2048 as your default, you'll need to force DOS compatibility by using `fdisk -c=dos` ... when creating your virtual disk.

The `menu.lst` file will look something like the following:

```
title MemOS
root (hd0,0)
kernel /boot/memos-2
```

This file is interpreted by GRUB to determine which OS should actually be loaded into memory. GRUB is intelligent enough to be able to read the filesystem of your bootable disk partition. This is where `e2fs_stage1_5` is used by GRUB. Finally, the second stage of bootloading copies your bootable system into memory. However, your OS image, `memos-2`, will not load because it does not have a valid GRUB-compliant multiboot header. Since GRUB is based on the multiboot spec, it requires the OS to have a special magic number in the first 4096 bytes of memory. So, to get your `memos-2` binary to be booted by GRUB you'll need to add something like the following at the beginning of `memos-2.s`:

```
_start:
        jmp real_start

        /* Multiboot header -- Safe to place this header in 1st page of
memory for GRUB */
        .align 4
        .long 0x1BADB002 /* Multiboot magic number */
        .long 0x00000003 /* Align modules to 4KB, req. mem size */
                         /* See 'info multiboot' for further info */
        .long 0xE4524FFB /* Checksum */

real_start:

        # This is where the rest of your program goes
```

Once you have successfully generated your `memos-2` binary image, you should copy it to

`/boot` within your virtual disk. To do this, you'll need your disk.img mounted as a loopback device again, as described earlier.

But it Still Doesn't Work!

If you've got this far with `memos-2`, congratulations! However, in all likelihood nothing is working. This is because GRUB boots your OS code into a protected mode environment. In protected mode, you no longer have access to the BIOS, so all you real-mode software interrupts will not work. At this stage you can either write some code to force your `memos-2` code back into real-mode, or you can "poke" directly into VGA video memory, the bytes that you want displayed. To do this, you need to write to video memory, which is mapped at address 0xB8000 for text mode graphics. Further details can be found on the OSDEV wiki:

- How to print a character to the screen
- How to work with VGA in text mode

Note that GRUB boots your system in VGA Text Mode so you will not have to set this yourself. You can also use GRUB itself to report how much memory your system has.

If you get this far you should be able to use a PC emulator to boot your disk image and display text on the screen. For QEMU, type something like:

```
$qemu -hda disk.img
```

Submission

You should use `gsubmit` to submit files to a `primer2` directory, as stated earlier. Minimally, you should submit a solution to the first deliverable, including a `README-1` file that documents *exactly* all the steps you performed to create a virtual disk, even if you did not use it. If you are able to work on the second deliverable and use GRUB to boot `memos-2` then additionally submit `memos-2.s`, relevant linker scripts and a `README-2` file documenting how the code can be configured to execute and how it basically works.

Support Files

- Puppy Linux virtual disk image
  - To run this image with full guest root privileges, you can use QEMU, VirtualBox, BOCHS, VMPlayer or a similar PC emulator. For example:
    - $qemu -hda cs552.vdi -net nic -net user -vga std
  - You can also convert the disk image using a tool such as qemu-img, for use with other PC emulators:
    - $qemu-img convert -O <format> cs552.vdi cs552.<format>
- If you need access to the GRUB shell, `stage1`, `stage2` and `e2fs_stage1_5` files, they can be found in `/boot/grub` within the Puppy Linux virtual disk image.

Happy Programming!