Assignment #1

CSE 511 Fall 2022
Assigned: Wednesday, September 7, 11:59 PM
Due: Wednesday, September 21, 11:59 PM

Abstract

We will be using the Linux operating system to **compile** code in programming assignments. We will use qemu to **run** the compiled code. One possibility is to install Linux natively. Another possibility is to install Linux through WSL (Windows) or VirtualBox/VMware (Windows, x86-64 Mac). No matter how you install and run Linux, you should install and run qemu **inside** your Linux installation (i.e., it will be nested virtualization in case of Windows and Mac.) VirtualBox for Windows and Mac hosts is available for free at `https://www.virtualbox.org/wiki/Downloads`

If you have a more recent Apple M1 Mac (i.e., not x86-64), you need to get access to your own remote VM (which is set up by the IT department). You will have to install XQuartz for X11 forwarding, which is needed by qemu. You will also need to install GlobalProtect VPN used by Penn State when accessing the remote VM outside of the campus. Inside the Linux VM, you can install all necessary packages and qemu to run the compiled code.

Although we do not strictly require it, we recommend that you use Ubuntu 22.04 or 20.04 LTS, available at `https://ubuntu.com/download/desktop`. Alternatively, you can use any other distribution which is more or less recent. However, since many instructions that we provide are for Ubuntu, you may need to resolve arising issues (e.g., package installation) on your own.

Prior to any installation, make sure to back up your system to avoid any potential data loss!

1 Introduction

1.1 Policy

You will work individually to solve the problems for this assignment. Academic integrity is crucial. **Please read the syllabus regarding the academic integrity policy, which contains a list of things that are not permitted!** Note that academic integrity issues may arise when helping other students or looking for an answer online. All online sources have to be cited properly unless the piece of information comes directly from the official UEFI, Intel, or AMD manuals/specifications. OSDev is a good source for OS-related questions, but typically it is allowed only for certain corner cases which will be specifically mentioned by the instructor. For example, you cannot copy and paste code from OSDev or any other online source, or post any homework questions on forums. When in doubt, please ask the instructor or the TA first.

1.2 Tools

You need to run all UEFI experiments using qemu. See our lectures for guidance about qemu. To install qemu, run:

```
sudo apt-get install qemu-system-x86 libsdl2-dev
```

Please compile all code inside your Linux installation. Your system must have standard compilation utilities (including make, gcc, ld, objdump, etc):

```
sudo apt-get install build-essential
```

To compile UEFI code, you additionally need to install Clang and LLD (7.0.0 or a higher version is required):

```
sudo apt-get install clang lld
```

Ubuntu does not create lld-link, which is required to link PE/UEFI binaries (see our lecture), so make sure to create a symbolic link after lld installation (if it does not exist already):

```
cd /usr/bin
sudo ln -s lld lld-link
```

Finally, to create a bootable disk image, you need to install "mkfs.vfat":

```
sudo apt-get install dosfstools
```

1.3 Provided Code

Please get the code and create your own repository similar to the "Setup Assignment" by following this link: `https://classroom.github.com/a/MOsdbcI5`

All boot loader files must reside in "boot", and all kernel files must reside in "kernel". For your convenience, the code already provides the "FwImage" tool from `https://github.com/rusnikola/fwimage`. It will also build the tool automatically. We also provide TianoCore/Edk2 headers (you do not need to download them separately) in "boot/include".

1.4 Compilation and Execution

Please compile the code by running "make" in your repository. This command will also create a bootable disk image ("boot.img"). To run the compiled code in qemu, you can simply run "make test", which will internally execute the following command:

```
qemu-system-x86_64 -bios /usr/share/qemu/OVMF.fd -m 1024
  -drive format=raw,file=<path_to_img_file>
```

Note that the path to the UEFI BIOS is provided for Ubuntu. Please locate OVMF.fd if your system is different. If this file is missing, you can download and unpack "bios.zip" from Canvas.

When building "boot.img", the boot loader is placed into

```
EFI\\BOOT\\BOOTX64.EFI
```

and the kernel is placed into

```
EFI\\BOOT\\KERNEL
```

1.5 Submission

You will receive ***zero*** points if:

- you break any of the assignment rules specified in the document
- your code does not compile/build
- your code is buggy and crashes qemu or system
- the GIT history (in your github repo) is not present or simply shows one or multiple large commits; it must contain the *entire* history which shows the actual *incremental* work.

Please commit your changes relatively frequently so that we can see your work clearly. **Do not forget to push changes to the remote repository in github!** You can get as low as ***zero*** points if we do not see any confirmation of your work. **Do not attempt to falsify time stamps, this may be viewed as an academic integrity violation!**

Your submission can use the late policy, and we will calculate and use the better of the normal score and the late score with the late penalty.

Finally, include "report.pdf" in your github repository with a qemu screenshot (that shows a fully booted and running kernel) and a brief description of what was implemented for each of the questions below. Please be specific when describing what was (and was not) implemented but do not be too verbose.

In your Canvas submission, please indicate the GIT commit number corresponding to the final submission. You should also specify your github username. The submission format in plain text

```
<your_github_username>:GIT commit
```

For example, runikola:936c332e7eb7feb5cc751d5966a1f67e8089d331

1.6 Disassembler

If you need to disassemble the kernel binary, you need to specify corresponding options to objdump (since it is a "raw binary"):

```
objdump -D -Mx86-64 -b binary -m i386 kernel.x86_64
```

or this if you prefer Intel/Microsoft syntax:

```
objdump -D -Mintel,x86-64 -b binary -m i386 kernel.x86_64
```

2 Assignment Questions

2.1 Basic Boot Loader [50 pts]

First, implement a very simple boot loader (BOOTX64.EFI) which loads an OS kernel (KERNEL). The assumption is that the kernel is in a "raw binary" format (position-independent code) to simplify this the task. Please see the corresponding templates in "boot/boot.c" and "kernel/kernel.c."

If you need to modify “Makefile” for your system, you can do so, but you should not modify the overall structure (i.e., we should be able to compile and run your code using the original Makefile on our own Ubuntu system).

You should avoid using any UEFI calls from the OS kernel. The OS kernel is somewhat trivial and its only purpose is to indicate that it was loaded. To that end, you have to use a 32-bit (BGRA) 800x600 video framebuffer. The framebuffer is a linear array of pixels, where each display pixel (x,y) is defined as shown in Figure 1. Using the framebuffer does not trigger the use of UEFI since all video memory is directly mapped to the memory address space. You should draw some figure (a square, triangle, rectangle, or any other trivial picture) and use some colors.

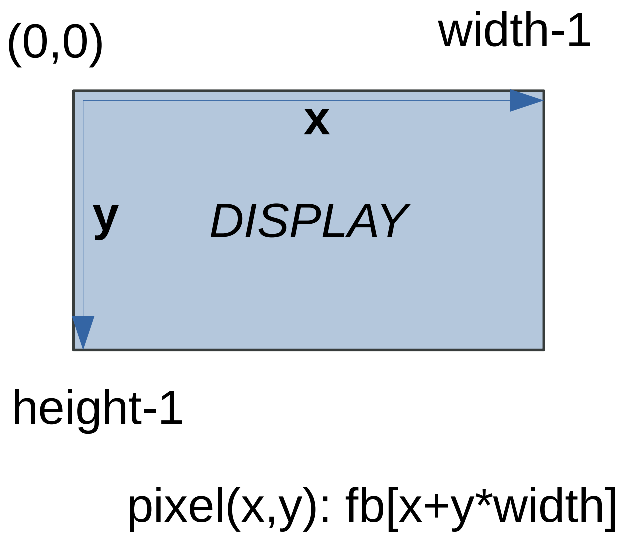

Figure 1: A video framebuffer (“fb” is the base address).

This approach shows how a simple (unaccelerated) video driver in an OS can work while avoiding any calls to the firmware. A real OS kernel can also load some fonts and render text directly into the framebuffer (but this is not required for this assignment).

Once the kernel takes over, you should never return to the boot loader. You can use a simple busy-wait loop (already provided in “kernel/kernel.c”).

2.2 Reclaiming Memory [25 pts]

Extend the boot loader such that it calls ExitBootServices() and reclaims boot-time memory. Make sure you do not accidentally reclaim the memory for the kernel code and/or the system page table.

Note that you need to implement this functionality in the boot loader. For simplicity, you can avoid passing the memory map to the kernel.

Please read the description of ExitBootServices() carefully. More specifically, ExitBootServices() is not guaranteed to succeed right away. The rationale for this is that the firmware may not be able to recycle all memory immediately, so you need to repeat this call multiple times. Each time, newly recycled memory will be placed into the updated memory map.

Take a closer look at the EFI_INVALID_PARAMETER error status, which should indicate that you need to obtain a new memory map and call ExitBootServices() again. Thus, you need to create a loop, in which you first get a new memory map (make sure that the return status is success), then attempt to call ExitBootServices(). If the latter succeeds, you can exit from the loop and proceed to the kernel. If not, you need to analyze the error status. EFI_INVALID_PARAMETER is not a fatal error, it simply means you need to go to another iteration and get a new memory map. Do not forget to deallocate any temporary buffers that hold the previous memory map before getting a new memory map! Please do not call ConOut for EFI_INVALID_PARAMETER since any screen output can unnecessarily change the memory map, which will prevent the loop from ever converging.

Likewise, avoid any output when ExitBootServices() succeeds since you are not allowed to use any boot services (including ConOut) when ExitBootServices() cleans up the boot-time memory. Given that you cannot print any output and ExitBootServices() is the very last step in the boot loader before jumping to the kernel, we highly recommended to debug your code first and make sure you can successfully jump to the kernel without calling ExitBootServices(), which should still work. Only after that, include ExitBootServices() prior to jumping to the kernel.

2.3 Setting Up a Page Table [25 pts]

Finally, extend the kernel such that the kernel will redefine and use its own page table. For this task, you may need to use some inline assembly instructions to load a new page table into CR3 (see our lecture slides). You should generate your own x86-64 page table, which can be done fully in C.

UEFI already loads the default page table that provides identical virtual-to-physical mappings, but the kernel, technically, cannot rely on it since the kernel does not know where this page table is located and how much memory it maps. Our goal is to avoid this unwanted dependency when we launch the kernel. (Note that you can only modify the default page table after ExitBootServices() completes since boot services may have implicit assumptions about the default page table.) We recommend that you initialize and load the page table as the very first step in the kernel (i.e., outside of the boot loader).

This page table will provide identical virtual-to-physical mappings for the first **4 GB** of physical memory, and you have to use **4 KB** pages. All pages are assumed to be used for the privileged (ring 0) kernel mode. Although your memory size does not exceed 1 GB, we are asking you to map 4 GB because the frame buffer's video memory is typically located somewhere in the region of 3-4 GB. You can ignore the fact that some mappings will point to non-existent physical pages in this 4 GB region.

The page table address has to be loaded into the CR3 hardware register. You need to initialize the page table before writing to CR3. Please follow our lecture slides. The slides show an example with bitfields for better illustration. **Remember that all entries of bitfields for page tables must be initialized, including those that are not currently used. (You should specify 0 for any of the reserved or unused bits.)** It may be more convenient (and less error prone), if you simply use 64-bit integers (u64) and set corresponding bits yourself using bitwise OR/AND operations (i.e., avoiding bitfields completely).

For simplicity, we will only allow two permission bits: the 'present' bit and the 'writable' bit, both of which must be set to 1 in the page table. Please do not attempt to use any other bits for now. See the AMD64 reference, Figures 5-18, 5-19, 5-20, 5-21; pp.144-145, p.151; you only need to consider the "long" (64-bit) mode and 4 KB pages.

When you initialize page table descriptors, note that PDP contains 4 entries and PML4 refers to just 1 entry. However, you still have to initialize the entire page (i.e., all 512 entries)! Initialize them appropriately with a value that does not refer to a real page, i.e., 0. Also make sure you allocate/reserve one full page for PDP and one full page for PML4.

For page table descriptors, you need to allocate some space (slightly more than 8 MB for our 4 GB mappings, see our lecture). We recommend to allocate this buffer directly in the boot loader (using an appropriate data type to avoid deletion by ExitBootServices()) and pass this buffer address as the 4th parameter to the kernel entry function. Note that AllocatePool() typically works for allocating smaller buffers, i.e., it is not guaranteed to work with > 8 MB. We recommend using AllocatePages() which works for larger buffers. Another advantage of AllocatePages() is that you can directly specify the number of pages and the returned address is already page-aligned, which is required for page table descriptors. This way, you can avoid aligning the address yourself.

When you complete this question, please move your framebuffer output (i.e., any figure that you draw) **after** you initialize the new page table. This way you will demonstrate that your kernel works **after** you update the page table register.

3 References

The following documentation can be useful for the assignment:

UEFI Specification Version 2.9 (released March 2021) `https://www.uefi.org/specifications`

Phoenix UEFI documentation `https://web.archive.org/web/20181012151104/http://wiki.phoenix.com/wiki/index.php/Category:UEFI_2.0`

RGBA color model `https://en.wikipedia.org/wiki/RGBA_color_model`

Framebuffer (32-bit pixels in our case) `https://en.wikipedia.org/wiki/Framebuffer`

Intel 64 documentation `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html`

AMD64 Volume 2, System Programming `https://www.amd.com/system/files/TechDocs/24593.pdf`

OSDev Paging Examples `https://wiki.osdev.org/Paging`, `https://wiki.osdev.org/Identity_Paging`, `https://wiki.osdev.org/Page_Tables`, and others. Note that we use 64-bit (long) mode.