Project #2 (MiniOS)

CMPSC 473 Fall 2023
Checkpoint 1:
Assigned: Sunday, October 22, 3:30 AM
Due: Saturday, October 28, 11:59 PM
Final:
Assigned: Sunday, October 22, 3:30 AM
Due: Saturday, November 4, 11:59 PM

Abstract

In this project, you will extend a small operating system to set up a simple system call and initialize a simple page table.

Since a lot of this code requires more direct access to the computer, you will be working on your local machine using VMware Workstation Player (Windows, Linux) or VMware Fusion (Mac). Both older Mac (Intel Silicon) and newer Mac (Apple Silicon) are supported, but installation instructions vary (see below). **If you have other hypervisors, e.g., VirtualBox, they will also typically work at least in Windows and Linux, but we will not provide any support for issues that may arise. We do not recommend using WSL in Windows, as it will likely NOT work, though we have not tested it.**

Note that you will not be able to use the lab machines. This project is an exception from our general rule that requires to always test your code on the lab machines. Among other issues, the lab machines have older version of 'gcc' and 'ld' which will not work properly with the project. You should also make sure that you use more recent versions of these tools if you use Linux on your local machine natively (this is why we still recommend using a virtual machine even when you use Linux already).

1 Introduction

1.1 Policy

You will work in groups for this project. The academic integrity rules stay the same as in previous course assignments and projects.

1.2 MiniOS

You are provided with the code of a tiny operating system, which is not based on any other OS (e.g., Linux). This OS does not include any interrupt or trap handlers, except minimal system call support.

The system consists of three pieces: (1). the boot loader (boot.efi), which loads the OS kernel and a user application into memory; you are already given a binary for the boot loader to simplify the build process; (2). the OS kernel (kernel_x86_64), a piece of software that runs in the privileged mode (ring 0); (3). a simple user application (user_x86_64), a piece of software that runs in the unprivileged mode (ring 3).

The OS kernel supports a video frame buffer which uses 800x600 (RGB32) video mode. On top of the video frame buffer console, the system implements text output functions such as `printf`, which is analogous to the user-space equivalents for the most part but used inside the OS kernel.

1.3 Tools

To build this OS, you need a Linux development environment and tools. Although the lab machines will compile the code, the resulting binaries will be broken. More importantly, the lab machines will not be able to **run** this OS.

Thus, we recommend using latest Linux distributions instead. We have tested compilation with Ubuntu 22.04 Desktop x86-64 and ARM64, which is what we strongly recommend to use.

If you use **Windows** or **Linux**, please get and install VMware Workstation Player at `https://www.vmware.com/products/workstation-player.html`. When installing, make sure to select the option that allows to use it for free (non-commercial personal usage license).

If you use **Mac** (either Intel or Apple Silicon), please get and install VMware Fusion at `https://www.vmware.com/products/fusion.html`. Click on "Try For Free." You can get a free license for personal use as well but you need to create an account by following the instructions at `https://www.vmware.com/go/get-fusionplayer-key`.

If you use Windows (x86-64), Linux (x86-64), or **older Macs** (Intel Silicon), you need to download Ubuntu 22.04 for x86-64 at `https://releases.ubuntu.com/jammy/ubuntu-22.04.3-desktop-amd64.iso`.

If you use **newer Macs** (Apple Silicon: M1, M2, etc), you need to download Ubuntu 22.04 for ARM64 at `https://cdimage.ubuntu.com/jammy/daily-live/current/jammy-desktop-arm64.iso`. Important: this is only for newer Macs, DO NOT attempt to use it on older Macs!

Please create a virtual machine with reasonable parameters (e.g., VMware by default recommends 20 GB virtual disk, 4 GB of RAM, 2 CPU cores). Use the corresponding Ubuntu image (x86-64 or ARM64 depending on the system) to create a virtual machine. Make sure to **actually install Ubuntu**. DO NOT start next steps in "Try Ubuntu" screen, rest of the things may not work properly, and your work might be lost. Ensure you have installed Ubuntu properly, i.e., when you boot the virtual machine again, it should take you directly to the login screen without presenting the "try or install" menu.

After installation, please install additional packages.

If you use Windows, Linux, or **older** Macs (Ubuntu 22.04 for x86-64), please run the following commands:

```
sudo apt update
sudo apt install make gcc binutils
sudo apt install git
sudo apt install ovmf qemu-system-x86
```

If you use **newer** Macs (Ubuntu 22.04 for ARM64), please run different commands instead:

```
sudo apt-add-repository universe
sudo apt update
sudo apt install make gcc gcc-x86-64-linux-gnu binutils-x86-64-linux-gnu
sudo apt install git
sudo apt install ovmf qemu-system-x86
```

You can also install additional packages that can be useful for you (e.g., vim, emacs, or any other editor).

Please make sure you have access to your github repository. For this, you can either copy your existing key or generate a new key and then import this **additional** key to your github account. For the latter case, please follow the steps to generate and import a new key described in "Programming Assignment Setup" that you have already completed previously. (Note that you will be executing these commands on your Linux virtual machine rather than a lab machine.)

Both teammates must do the above steps. It is not acceptable to work on just one machine. Remember that each teammate must have separate commits. That also includes an ability to compile, test, and run the system individually!

Please set up your system as soon as you can, do not put it off until the very last moment. **Do not expect any help from the teaching staff in the very last moment, e.g., prior to the checkpoint or final deadlines.** No excuses such as "I was previously working on my teammate's machine, but now I have to install the system on my own machine and it does not work" will be accepted for the reason mentioned above – you must always contribute to your github repository and test changes individually.

1.4 Provided Code

Please get the code and create your group, similar to Project 1 by following this link: `https://classroom.github.com/a/2ICP5Vd1`

For **newer Macs** (Ubuntu 22.04 for ARM64, see above), you need to edit Makefile. This is the *only* change you are allowed to do in Makefile. Find "CC = gcc" and "LD = ld" strings and replace them with the following:

```
CC = x86_64-linux-gnu-gcc
LD = x86_64-linux-gnu-ld
```

Do not change Makefile if you use Windows, Linux, or older Macs!

If you and your partner use different types of machines (Windows vs. newer Mac), you can just change Makefile locally and do not commit that change to the repository.

Note that groups for Project 2 are different, you must specify **sgroup** rather than **group** when creating or joining github groups to properly distinguish them with Project 1 groups. This is very important to avoid any confusion with Project 1 groups! Any repositories that do not conform to this naming convention will be deleted without any prior warning.

The provided code consists of the "kernel" and "user" parts. The "user" part makes system calls to print some messages. You will only need to modify the "kernel" part. You are only allowed to change `kernel_code.c` and `kernel_extra.c` inside "kernel". **Everything else will be discarded for grading.**

The boot loader allocates some memory (around 32 MB), which will be used inside our kernel. Part of this memory is used for the heap (see the extra credit assignment). But the remaining part can be used to initialize the page table. You can assume that the memory base address is already page-aligned.

In `kernel_code.c`, there are two functions that you need implement:

1. `kernel_init`. It passes addresses to the user space stack, user program (code and data), and memory that can be used to initialize the page table. Initially, user stack and program will reside in kernel space. Our system makes sure that it will still work even if you do not relocate those into user space. However, your eventual goal (Q4) is to move those addresses in a dedicated user space portion of the address space. To load the page table, you need to use `load_page_table`. This is a special function, which checks your page table (and returns a string with an error message if it fails) and then loads it to the CR3 register which is used to specify the top level (root) page table address in x86-64.

2. `syscall_entry`. This is the entry point for all system calls. You need to implement one specific system call (n = 1) and return an error status (-1) in all other cases. Note that the mechanism to route system calls from user space is already implemented, you only need to wire it to the corresponding handler inside this function.

In `kernel_extra.c`, you need to place your memory allocator code (`mm.c`) if you want to complete the extra credit assignment. It is very likely that you will need to do some changes to your code so that it can be compilable in our system.

Remember that none of the standard C library headers are available in the OS kernel. Instead, all headers are provided in `kernel/include`. But you can typically find similar functions there. Please explore that directory and see what additional header files you need to include (if any).

To compile the OS code, you need to run:

```
make
```

Note that sometimes it would be recommended to make a clean build:

```
make clean
make
```

After 'make', a special 'minios.img' disk image will be generated. You can now run the operating system:

```
make test
```

1.5 Submission

You need to submit the checkpoint (Q1) and the final version (Q2-Q4) separately. The goal of the checkpoint is to simply make sure that you have a very basic understanding of the building process and are able to run miniOS. This part is **really** very trivial for which we give ample time to resolve all possible issues. **Thus, please do not wait until the checkpoint is due before you get started on the final version since this is the real part of the project which is likely to take much longer!** In other words, we assume that you are already working towards the final version in your first week, so please do not wait until the very last moment.

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

- you break any of the project rules specified in the document
- your code does not compile/build
- your code is buggy and crashes miniOS
- 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 for **you** as a member of your team

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!** Each member has to commit their changes **separately** to reflect **their** work, i.e., do not commit the code on behalf of your partner since we will not be able to tell who did that part of the work! You can get as low as ***zero*** points if we do not see any confirmation of your work. Please split the work equally! **Do not attempt to falsify time stamps, this may be viewed as an academic integrity violation!**

Each checkpoint and final 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.

1.5.1 Checkpoint (Q1): 30 points

For checkpoint (Q1), you just need to submit a screenshot where you just print a message:

```
Hello from <group>, <your_github_username>
```

Your output will look something like Figure 1.

1.5.2 Final (Q2-Q4): 130 points

Your output will look something like Figure 2, but you do not need to submit the screenshot.

For the final submission, the format is similar to Project 1: You will indicate the GIT commit number corresponding to the final submission. Be sure to update README.md to specify which part was implemented by each partner (similar to Project 1). Please note that each member needs to submit the GIT commit number (must be the same that your partner submits!) and the group name that corresponds to the project (e.g., for the p2-sgroup1 repository, you should specify sgroup1). You should also specify your github username. That should be your github username, not the name of your partner! The submission format in plain text

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

For example, sgroup512:runikola:936c332e7eb7feb5cc751d5966a1f67e8089d331

Note the **sgroup** prefix (rather than **group** that was used for Project 1).

1.5.3 Extra Credit: 40 points

Submit together with the final version (see above).

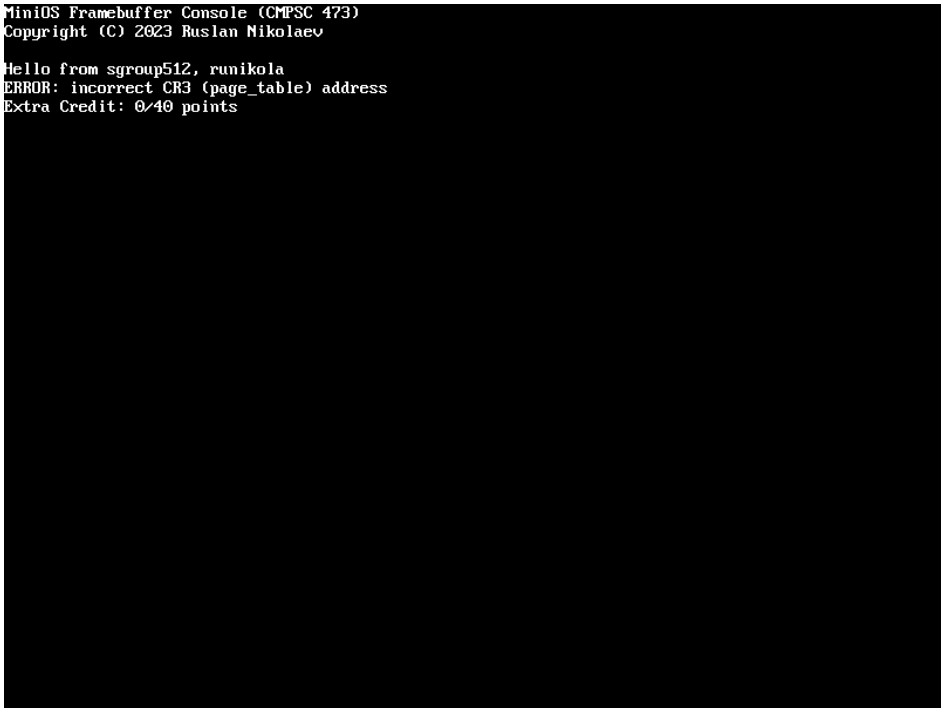

Figure 1: An example output (checkpoint).

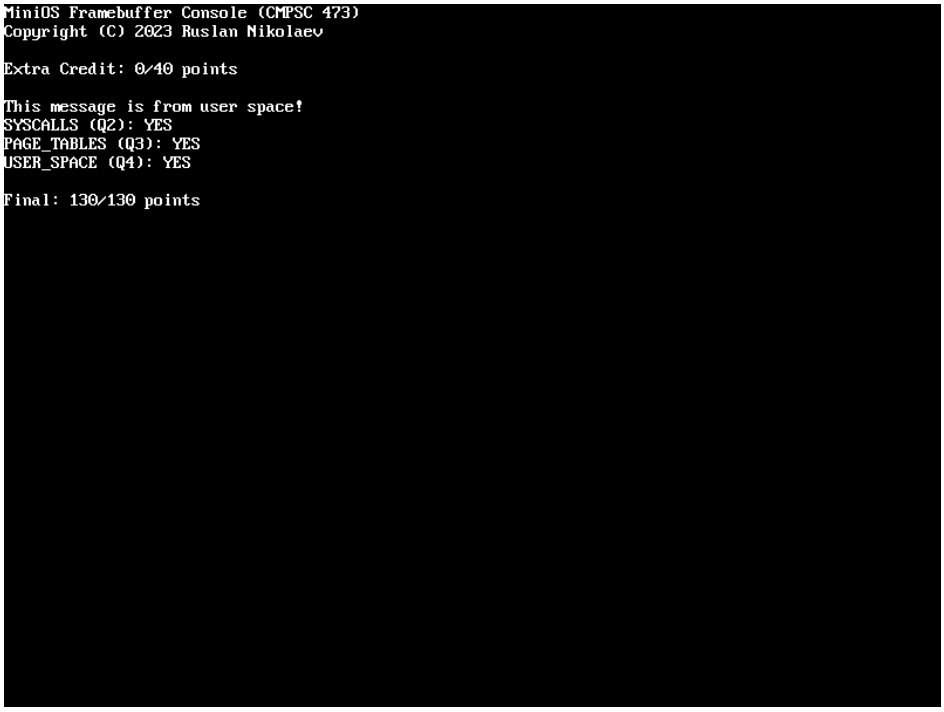

Figure 2: An example output (final).

1.6 Executable Formats

For simplicity, we use the "pure binary" executable format, i.e., no executable format at all. Our executable files are very special. They are compiled to use position-independent code and thus can be placed anywhere in the address space. Specifically, this is very important for the final part with the user space program (Q4), since the compiled code will make sure that the user program will run no matter what virtual address location it is placed in.

1.7 Disassembler

If you every need to disassemble "kernel" (or "user") binary, you need to specify corresponding options to objdump (since they are "pure binaries"):

```
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
```

Note that for newer Macs (Ubuntu 22.04 ARM64), you need to use x86_64-linux-gnu-objdump instead of objdump.

2 Project Questions

2.1 Installation [30 pts]

For checkpoint (Q1), you just need to submit a screenshot where you just print a message:

```
Hello from <group>, <your_github_username>
```

For example,

```
Hello from sgroup512, runikola
```

Please print this message in the beginning of `kernel_init`. **Each partner has to submit their own screenshot!**

2.2 System calls [40 pts]

You need to modify `syscall_entry` and implement one specific system call (n = 1) to print a message on the screen using 'printf'. The return status of this call should be 0. You will return an error status (-1) in all other cases. Note that the mechanism to route system calls from user space is already implemented in `kernel_asm.S`, you only need to wire it to the corresponding handler inside `syscall_entry` in `kernel_code.c`.

Remember that you are allowed to change `kernel_code.c` only.

2.3 Page table [60 pts]

The boot loader already sets up the default page table, but you need to create your own x86-64 page table (this can be done fully in C). 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 (used by 'printf') 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 code provided in `kernel_init` will already call a special `load_page_table` function, which will verify your page table and load it into the CR3 hardware register. You need to initialize the page table before this function is called.

Please follow our lecture slides for Oct 10 (`lecture15.pdf`) and video recordings for Oct 10 and Oct 19. 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 reserved bits.)** It may be more convenient (and less error prone), if you simply use 64-bit integers (uint64_t) and set corresponding bits yourself (i.e., avoiding bitfields completely), just like you did that for the previous project.

For simplicity, we will only allow two 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 (except the 'user' bit in the next question)! Note that `load_page_table` will fail if you set any other bits. Please refer to our slides and CPU manuals. See the AMD64 reference, Figures 5-17, 5-20, 5-21, 5-22, 5-23; pp.142-145, p.154-155; you only need to consider the "long" (64-bit) mode and 4 KB pages. **Please ignore 5-18 and 5-19, we will not use this newer feature - PML5 - our page table will only use traditional 4 levels!**

Note that even though user applications are only allowed to be executed in user space, the provided `load_page_table` function will take care of that problem as long as you do not modify `user_stack` and `user_program`. For now, you are only asked to create kernel-space mappings. You will modify `user_stack` and `user_program` in the next question.

Remember that you are allowed to change `kernel_code.c` only. Do not attempt to "trick" the operating system to get points without actually implementing the page table.

2.4 User space support for the page table [30 pts]

You should extend the page table for the application so that you can also properly execute application code from user space. The page table should still point to the kernel portion (i.e., the first entry of PML4). However, this page table will also refer to the pages that are assigned to the user application itself (the user program and the user stack). Note that the kernel portion should *only* be accessible in kernel space, whereas the application portion is accessible in both user and kernel space. Also note that both `user_program` and `user_stack` must point to user-space (rather than kernel-space) virtual addresses. Initially, they point to the kernel-space virtual addresses (somewhere in the first 4 GB).

You are allowed to assign any virtual addresses from the bottom 2 MB for the application (i.e., 0xFFFFFFFFFFE00000 to 0xFFFFFFFFFFFFFFFF). As mentioned during our lecture, x86-64 sign extends 48-bit virtual addresses (from the page table) to 64-bit virtual addresses. You can

assume that the program occupies just 1 page and the user stack is also just 1 page. **Recall that the stack grows downwards**, thus the initial `user_stack` address points to the next page (i.e., stack_memory_base + 4096). Consequently, you need to get the physical page address correctly and map that physical page to the new virtual address. Finally, the new virtual address that you will write into `user_stack` should also point to the next *virtual* page (i.e., stack_memory_base + 4096) since the stack grows downwards. Note that `user_program` will still point to a "normal" base address, but you also need to map the corresponding page and write the user-space virtual address.

As shown in Figure 3, the user-space mappings correspond to the last entry of PML4 (Level 4), last entry of PDP (Level 3), and the last entry of PD (Level 2). You can use any entries in PT (Level 1). Please make sure to set up correct permissions for these new mappings. Recall that the user-space portion can be accessed in both modes, and the kernel-space portion can only be accessed in kernel/privileged mode.

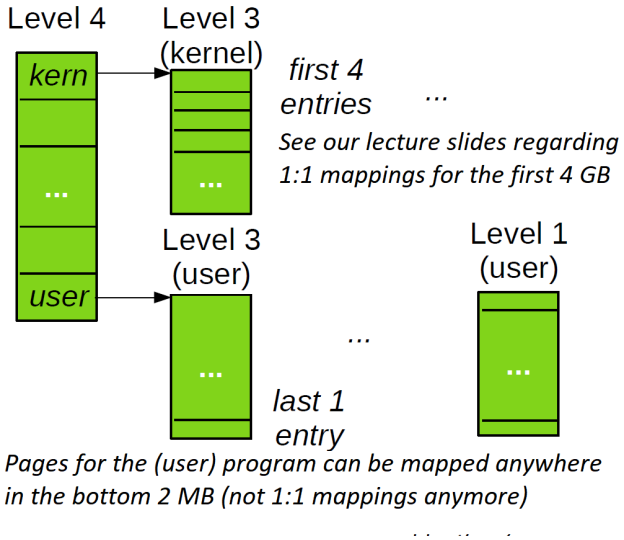

Figure 3: A page table with both kernel- and user-space mappings.

Remember that you are allowed to change `kernel_code.c` only. Do not attempt to "trick" the operating system to get points without actually implementing the user-space portion of the page table.

2.5 Extra Credit [40 pts]

This is an optional extra credit part. You need to integrate the memory allocator implementation from Project 1 (`mm.c`) into `kernel_extra.c`. Remember that you are not allowed to use any standard header files. The template in `kernel_extra.c` already includes some headers that you will likely need. You need to provide `mm_init`, `malloc`, and `free`.

The miniOS kernel will do some tests on your memory allocator. You can use *most* support functions that you previously used in Project 1. They are now implemented in `kernel_malloc.c` but you should not modify that file anyhow.

Note that since this is an extra credit part (which is completely optional), only very limited support from the teaching staff will be provided.

Remember that you are allowed to change `kernel_extra.c` only. Do not attempt to "trick" the operating system to get points without actually implementing the extra-credit part.

3 References

AMD64 Volume 2, System Programming `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf`

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`

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