The MOS Project

Lab Tutorial

Moody Liu
November 2022

Contents

1 Lab 1 - Setting Up the Development Environment **3**
1.1 Introduction to the Tools . . . 3
1.1.1 CMake . . . 3
1.1.2 NASM . . . 4
1.1.3 cpio . . . 4
1.1.4 qemu-system-i386 . . . 4
1.1.5 i686-elf Toolchain . . . 4
1.2 Installating the Tools . . . 4
1.2.1 CMake . . . 4
1.2.2 NASM . . . 5
1.2.3 cpio . . . 5
1.2.4 qemu-system-i386 . . . 5
1.2.5 i686-elf Toolchain . . . 5
1.3 Building MOS . . . 5
1.3.1 Cloning the Repository . . . 6
1.3.2 Configuring MOS . . . 6
1.3.3 Building MOS . . . 7
1.4 Running MOS . . . 7
1.5 Debugging MOS . . . 8
1.5.1 Required QEMU Arguments . . . 8
1.5.2 Configuring GDB . . . 9
1.5.3 Attaching to QEMU . . . 9
1.5.4 Configure your IDE for Future Debugging . . . 9

Introduction to the MOS Lab

Warning:

- This chapter is a work in progress.

The primary goal of MOS is to provide an platform-independent environment for beginners to learn OS.

Users of MOS will be able to experiment and get them familiar with with these following features of an operating system:

Note: The **bold** items are the main objectives of the project.

- Basic OS Knowledges
- Memory Management
  - **Paging**, and a Brief of Segmentation
  - Common Memory Management Practices in Modern Kernels
- File Systems
  - Underlying File Operations
  - VFS Framework (a file-system abstraction layer)
- Process Management
  - Allocating a New Process
  - The Famous `fork()` Syscall
  - **Process Scheduler**
  - **Threads**
  - **Threads Synchronization** via `mutex`

Chapter 1

Lab 1 - Setting Up the Development Environment

In this lab, we'll set up the basic development environment for the rest of our labs. Firstly I'll introduce you to the tools and we'll install them, then we'll compile MOS, run it in QEMU for the first time, and then attach GDB to it.

Finally, and hopefully, you can get your favorite IDE set up to work with MOS.

1.1 Introduction to the Tools

MOS is an operating system, thus, preparing for a development environment is already not an easy task (bruh). Several efforts have been made to make the process easier.

We are currently targeting the 32-bit `x86` architecture, the tools in table 1.1 are the ones we'll be using.

Tool	**Description**	**Installation**
`CMake`	1.1.1	1.2.1
`i686-elf` toolchain	1.1.5	1.2.5
`NASM`	1.1.2	1.2.2
`cpio`	1.1.3	1.2.3
`qemu-system-i386`	1.1.4	1.2.4

Table 1.1: Tools used in this lab

1.1.1 CMake

> CMake is an open-source, cross-platform family of tools designed to build, test and package software[1]

MOS uses CMake as the build system generator, it supports many build systems like 'Make', 'Ninja', 'Visual Studio' and 'Xcode'.

Note:

- It's the actual build system (e.g. 'Make') that starts the compiler, linker, etc., CMake is

[1]https://cmake.org/

only to **generate** the configuration files for such build system.

We'll use 'Make' as the build system for MOS in this tutorial, but **you can use 'Ninja' if you want**.

1.1.2 NASM

NASM is an assembler for x86 architecture. There are several files under 'arch/x86' that are written in NASM. It has a cleaner syntax than the GNU assembler (i.e. `as`).

1.1.3 cpio

Cpio is the format of an archive, and also the tool to create such archives. MOS uses cpio as the initial root filesystem.

1.1.4 qemu-system-i386

QEMU is an open-source emulator, it also provides a gdb stub for debugging. It can be installed via your Linux's package manager.
(e.g. `apt install qemu-system-i386` or `pacman -S qemu-system-x86`, …).
See its download page for more details.

1.1.5 i686-elf Toolchain

As its name suggests, this is a cross toolchain for 'i686-elf' target. 'i686' means the 32-bit x86 architecture, and 'elf' is the executable format. They together form the 'target-triple' of the toolchain.

Tip:

- Most 64-bit Linux OSes have 'target triple' of `x86_64-pc-linux-gnu`, or `x86_64-linux-gnu`.
- Read more about 'target triple' at `https://wiki.osdev.org/Target_Triplet`.

Unlike other applications (e.g. `bash` or `vim`) that they run on an existing operating system and a standard C library (say, `glibc` or `musl`). MOS itself is the operating system, thus there's not an existing OS for it to run on, neither a standard libc (i.e. no `printf`, no `malloc` etc.) for it to use, considering you're directly interacting with the CPU and the hardware.

This is called 'bare-metal' environment, or 'freestanding' environment, a 'bare-metal' compiler toolchain is exactly for this situation.

Warning:

- One should never use a hosted compiler (e.g. the `gcc` installed on the lab machine) to cross-compile for a bare-metal environment when they are targeting a different architecture, (`x86` and `x86_64` are different), it ***may sometimes*** work, but you'll have to add a lot of unnecessary flags.
- See `https://wiki.osdev.org/Why_do_I_need_a_Cross_Compiler%3F` for more information.

1.2 Installating the Tools

1.2.1 CMake

CMake should come with your Linux distribution's package manager, MOS requires at least version 3.20, but any newer version is recommended.

1.2.2 NASM

NASM can be installed via your Linux's package manager, the minimum version of NASM tested is `2.15.03`. A pre-built binary from NASM's website is also available.

1.2.3 cpio

cpio can also be installed via your Linux's package manager, the minimum version of cpio tested is `2.12`.

1.2.4 qemu-system-i386

TODO

1.2.5 i686-elf Toolchain

Installing the cross-compiler toolchain is probably the most difficult part of this lab, but it's once and for all, you don't have to do it again (unless you want to upgrade the toolchain).

There are majorly two ways to get this toolchain, either by building it from source or by downloading a pre-built binary.

Downloading a pre-built binary

If you don't want to build the toolchain from source, you can download a pre-built binary from moodyhunter/i686-elf-prebuilt (choose the i686 one).

Warning:

- Using pre-built binary saves time, but please consider doing so **only** if you trust the author.
- The above pre-built binary is built with GitHub Actions, and is built on Ubuntu 20.04.5 LTS (Image `ubuntu-20.04` version `20221027.1`).

Building From Source

The source code of binutils and gcc can be found at GNU Binutils's Website and GNU GCC's Website respectively.

Note:

- For Arch Linux users, checkout i686-elf-binutils, i686-elf-gcc and i686-elf-gdb.

The script located at `docs/assets/i686-elf-toolchain.sh` can download, compile and install the toolchain into the directory specified by the `PREFIX` variable.

1.3 Building MOS

Note:

- Before you continue reading, make sure `i686-elf-gcc`, `i686-elf-ld`, `nasm`, `cpio` are in your `PATH`.

1.3.1 Cloning the Repository

The recommended way to get MOS is to clone the repository from GitHub:

```
git clone https://github.com/moodyhunter/MOS
cd MOS
```

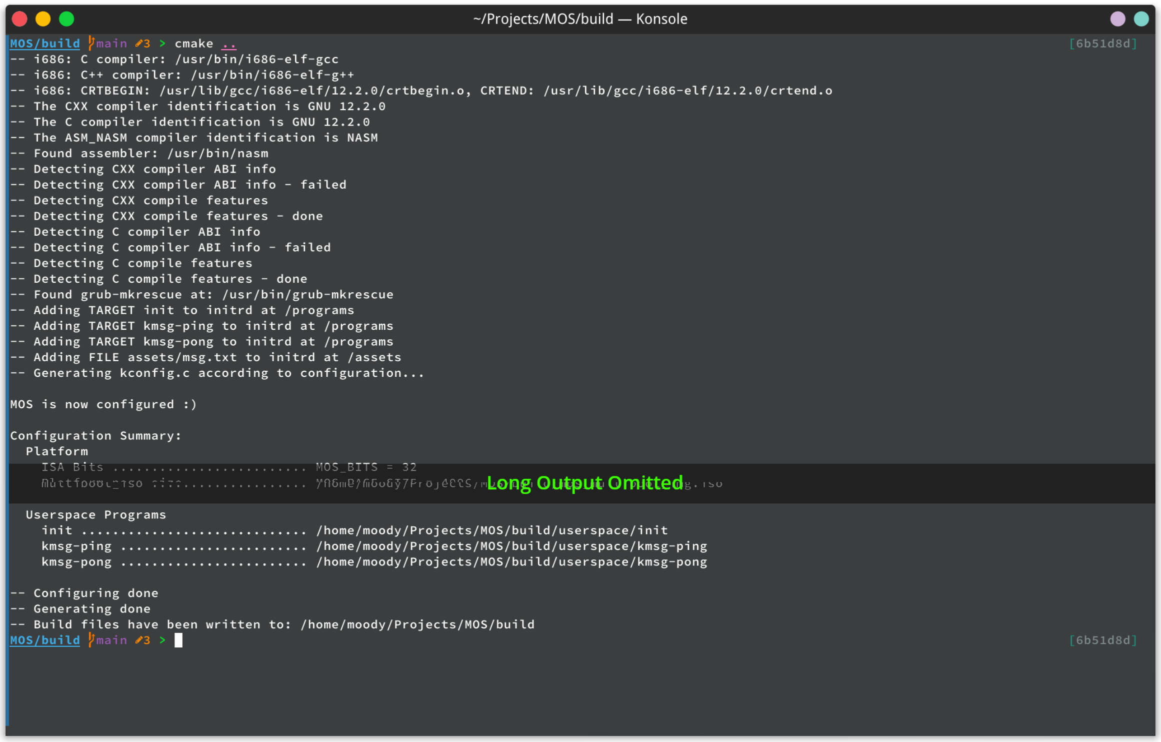
1.3.2 Configuring MOS

If you have a correct setup, then configuring step should be as simple as:

```
$ mkdir build && cd build
$ cmake ..
```

The first line creates a build directory, we are not going to build MOS in the source directory directly, because there will be many generated files, and it's not a good idea to pollute the source directory.

The second line runs CMake to configure MOS, after running this command, you should see a bunch of output, similar to 1.1.



```
~/Projects/MOS/build — Konsole
MOS/build main ✎3 > cmake ..                                                                      [6b51d8d]
-- i686: C compiler: /usr/bin/i686-elf-gcc
-- i686: C++ compiler: /usr/bin/i686-elf-g++
-- i686: CRTBEGIN: /usr/lib/gcc/i686-elf/12.2.0/crtbegin.o, CRTEND: /usr/lib/gcc/i686-elf/12.2.0/crtend.o
-- The CXX compiler identification is GNU 12.2.0
-- The C compiler identification is GNU 12.2.0
-- The ASM_NASM compiler identification is NASM
-- Found assembler: /usr/bin/nasm
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - failed
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - failed
-- Detecting C compile features
-- Detecting C compile features - done
-- Found grub-mkrescue at: /usr/bin/grub-mkrescue
-- Adding TARGET init to initrd at /programs
-- Adding TARGET kmsg-ping to initrd at /programs
-- Adding TARGET kmsg-pong to initrd at /programs
-- Adding FILE assets/msg.txt to initrd at /assets
-- Generating kconfig.c according to configuration...

MOS is now configured :)

Configuration Summary:
  Platform
    ISA Bits ........................ MOS_BITS = 32
    [illegible]              Long Output Omitted

  Userspace Programs
    init ............................ /home/moody/Projects/MOS/build/userspace/init
    kmsg-ping ....................... /home/moody/Projects/MOS/build/userspace/kmsg-ping
    kmsg-pong ....................... /home/moody/Projects/MOS/build/userspace/kmsg-pong

-- Configuring done
-- Generating done
-- Build files have been written to: /home/moody/Projects/MOS/build
MOS/build main ✎3 >                                                                               [6b51d8d]
```

Figure 1.1: CMake Output

Note:

- Don't worry if you see `Detecting C compiler ABI info` and `Detecting CXX compiler ABI info` fails, it's because CMake doesn't know what to do with a freestanding compiler, thus this check is meaningless.
- You may also see `grum-mkrescue` is not found, don't worry as well :), we are not going to

use it.

If you don't see the `-- Configuring done` or `-- Generating done` line, then something has been wrong, and you should scroll up and see what's wrong.

Tip:

- The `build` directory is where all the generated files are located, you can safely delete it and re-run CMake to re-generate the files, in case you messed up something.

1.3.3 Building MOS

After configuring MOS, you can build MOS by running `make`.

This command will only build the core part of MOS, which is the kernel and the userspace library. See the table below for the list of targets:

Target	**Description**	**Dependencies**
`all` *(default)*		`mos_kernel.elf`, `mos_libuserspace`
`mos_kernel.elf`	Bare kernel without booting	*None*
`mos_libuserspace`	Userspace standard library	`mos_kernel.elf`
`multiboot`	`multiboot` bootable kernel	`mos_kernel.elf`
`mos_initrd`	Initrd image	`mos_userspace_programs`
`mos_userspace_programs`	Userspace programs	`init`, `kmsg-ping`, ...
`init`	The init program	`mos_libuserspace`
`kmsg-ping`	A program to test kernel IPC	`mos_libuserspace`
...	...	...

Table 1.2: Targets

We are going to build the `multiboot` and `mos_initrd` targets, which are the two meta targets that 1) produce a bootable kernel and 2) build and package the initrd image.

So the command will be `make multiboot mos_initrd`.

You'll then see two files being generated in the `build` directory:

- `mos_multiboot.bin` is the kernel image, which is a multiboot-compliant kernel.
- `initrd.cpio` is the initrd image, which contains the userspace programs.

Several other files are also generated, for more information, read the MOS documentation at TODO.

1.4 Running MOS

Once we have compiled MOS, we can run it in QEMU.

The command passed to QEMU will be flexible based on your needs, but the most basic command is:

```
qemu-system-i386 -kernel mos_multiboot.bin -initrd initrd.cpio
```

You can pass more arguments to QEMU:

- `-s` to enable the QEMU GDB stub, which allows you to debug MOS using GDB.
- `-S` to pause the CPU before starting up, which allows GDB to take control of the booting process.
- `-m` to specify the amount of RAM to be allocated to the VM (say `-m 512M`).

- `-serial stdio` to redirect the serial output to your terminal, in the early stage of booting, MOS prints lots of information to the serial port to help you debug.
- `-append` to pass arguments to the kernel, for a list of supported arguments, see the MOS documentation at TODO.

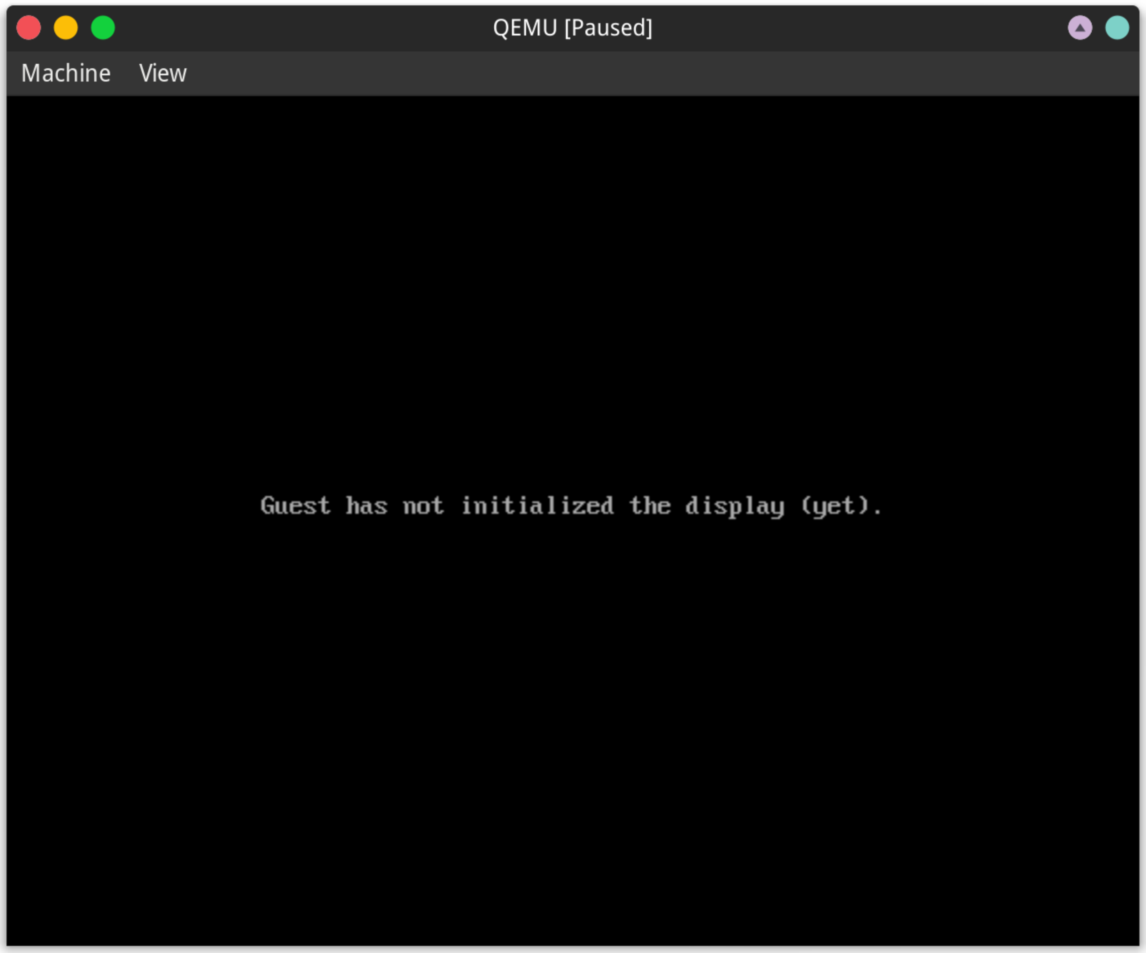
1.5 Debugging MOS

After successfully building and running MOS, you may want to debug it (just in case your code crashes the kernel).

1.5.1 Required QEMU Arguments

As mentioned above, we need to pass 2 more arguments `-s` and `-S` to QEMU so that it 1) enables the GDB stub, and 2) pauses the CPU before starting up (so that we have time to attach and place breakpoints).

After adding these two arguments, when starting up QEMU, you will see a message like Figure 1.2.



```
Guest has not initialized the display (yet).
```

Figure 1.2: QEMU GDB in Paused State

This means that QEMU is waiting for a GDB connection, and we can now continue to the next step.

1.5.2 Configuring GDB

Note that we are targeting the `i686-elf` architecture, so you should use `i686-elf-gdb` instead of `gdb`.

To begin with, we need to tell GDB where the kernel image file is, so that it can load the symbols and debug information from it.

You've probably already seen a `gdbinit` file in the `build` directory. This file contains commands for GDB to recognize our userspace programs, we'll pass this file to GDB using the `-x` argument.

So the overall command will be:

```
cd /PATH/TO/MOS/build
i686-elf-gdb ./mos_multiboot.bin -x ./gdbinit
```

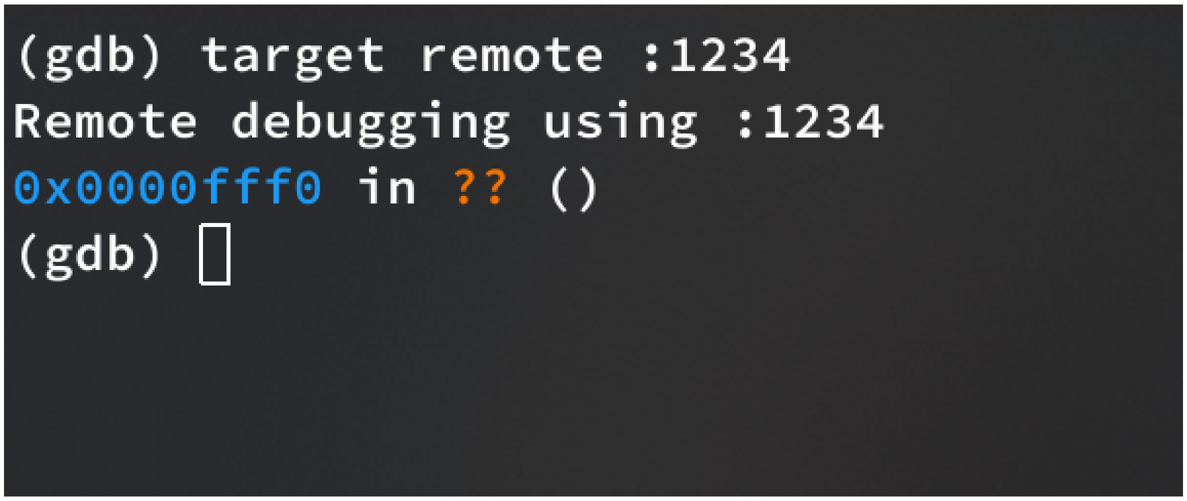
1.5.3 Attaching to QEMU

After GDB starts, you'll see it's 'adding symbol table from file' thanks to our `gdbinit` file. Now we need to attach GDB to QEMU, so that we can place breakpoints and debug the kernel.

QEMU (by default) listens on port 1234 for GDB connections, so we need to tell GDB to connect to it:

```
(gdb) target remote localhost:1234
```

GDB will then connect to QEMU, and you'll see a message like Figure 1.3.



```
(gdb) target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) 
```

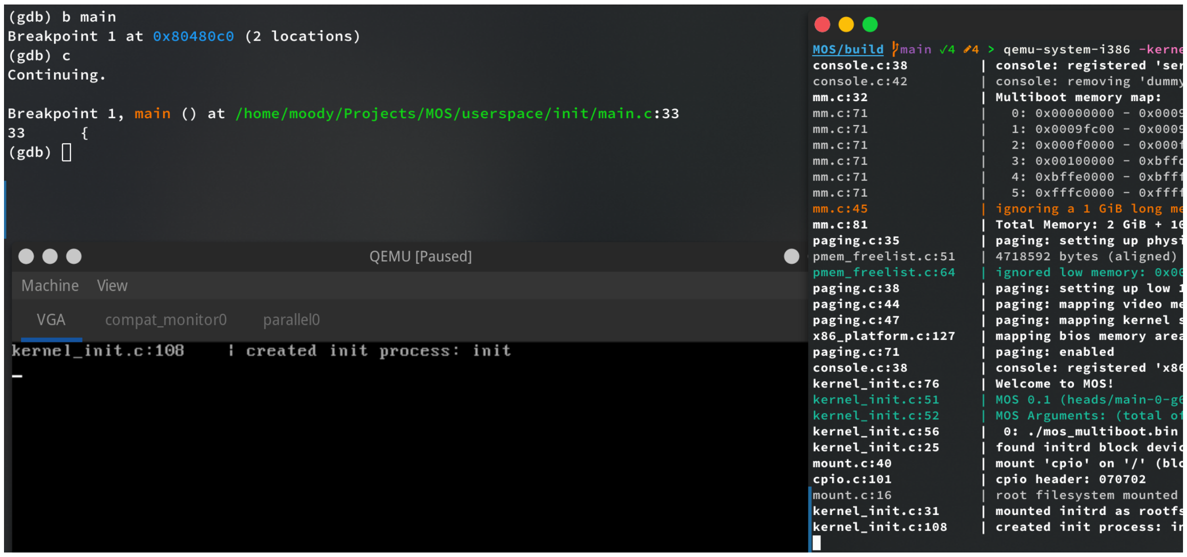
Figure 1.3: GDB Attached to QEMU

Try typing `break main` and then `continue` to see if it pauses at the `main` function, you should see a message like Figure 1.4.

1.5.4 Configure your IDE for Future Debugging

Note:

- Since I'm personally using VSCode, only the VSCode part will be covered here. If you are using any other IDE (e.g. CLion), things **may** be different, but the underlying idea of



```
(gdb) b main
Breakpoint 1 at 0x80480c0 (2 locations)
(gdb) c
Continuing.

Breakpoint 1, main () at /home/moody/Projects/MOS/userspace/init/main.c:33
33      {
(gdb) 
```

```
kernel_init.c:108    | created init process: init
_
```

```
MOS/build main ✓4 ✎4 > qemu-system-i386 -kerne
console.c:38            | console: registered 'ser
console.c:42            | console: removing 'dummy
mm.c:32                 | Multiboot memory map:
mm.c:71                 |   0: 0x00000000 - 0x0009
mm.c:71                 |   1: 0x0009fc00 - 0x0009
mm.c:71                 |   2: 0x000f0000 - 0x000f
mm.c:71                 |   3: 0x00100000 - 0xbffc
mm.c:71                 |   4: 0xbffe0000 - 0xbfff
mm.c:71                 |   5: 0xfffc0000 - 0xffff
mm.c:45                 | ignoring a 1 GiB long me
mm.c:81                 | Total Memory: 2 GiB + 10
paging.c:35             | paging: setting up physi
pmem_freelist.c:51      | 4718592 bytes (aligned)
pmem_freelist.c:64      | ignored low memory: 0x00
paging.c:38             | paging: setting up low 1
paging.c:44             | paging: mapping video me
paging.c:47             | paging: mapping kernel s
x86_platform.c:127      | mapping bios memory area
paging.c:71             | paging: enabled
console.c:38            | console: registered 'x86
kernel_init.c:76        | Welcome to MOS!
kernel_init.c:51        | MOS 0.1 (heads/main-0-g6
kernel_init.c:52        | MOS Arguments: (total of
kernel_init.c:56        |  0: ./mos_multiboot.bin
kernel_init.c:25        | found initrd block devic
mount.c:40              | mount 'cpio' on '/' (blo
cpio.c:101              | cpio header: 070702
mount.c:16              | root filesystem mounted
kernel_init.c:31        | mounted initrd as rootfs
kernel_init.c:108       | created init process: in
```

Figure 1.4: GDB Paused at `main`

remote debugging should be the same.

- IDEs are updating very fast, so the screenshots **may** be outdated. Ask me if you have any questions.

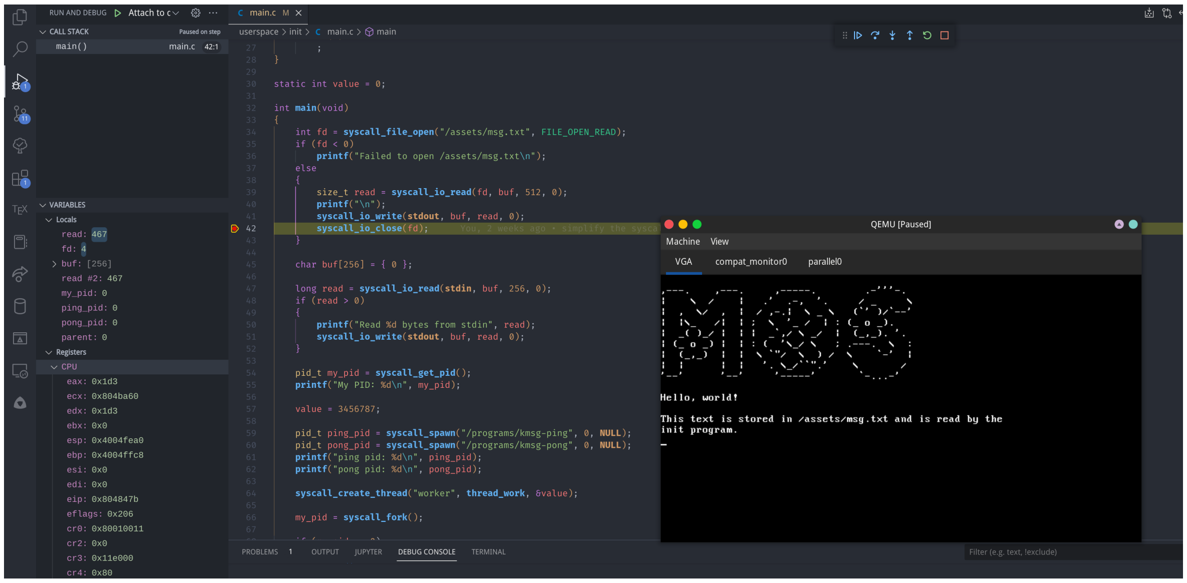
As you may (or may not) realize, attaching GDB to QEMU is basically the process of 'remote debugging', So we can configure our IDE to do this automatically, with the only two exceptions being that 1) the GDB in use is our `i686-elf-gdb` instead of the default `gdb` and 2) we need to specify the `gdbinit` file.

Prepared VSCode Setup

To use a prepared setup for VSCode, see the `launch.json` file in the `.vscode` directory, which is also my personal configuration for debugging MOS.

Install `tmux` on your machine and press `F5`, a QEMU window will pop up and you'll see the kernel booting up, with the breakpoints you set in your IDE as shown in Figure 1.5.

In this setup, you can also use `tmux attach-session -t mos_kernel_debug` to attach to the serial port.



```
VARIABLES
  Locals
    read: 467
    fd: 4
  > buf: [256]
    read #2: 467
    my_pid: 0
    ping_pid: 0
    pong_pid: 0
    parent: 0
  Registers
    CPU
      eax: 0x1d3
      ecx: 0x804ba60
      edx: 0x1d3
      ebx: 0x0
      esp: 0x4004fea0
      ebp: 0x4004ffc8
      esi: 0x0
      edi: 0x0
      eip: 0x804847b
      eflags: 0x206
      cr0: 0x80010011
      cr2: 0x0
      cr3: 0x11e000
      cr4: 0x80
```

```c
        ;
}

static int value = 0;

int main(void)
{
    int fd = syscall_file_open("/assets/msg.txt", FILE_OPEN_READ);
    if (fd < 0)
        printf("Failed to open /assets/msg.txt\n");
    else
    {
        size_t read = syscall_io_read(fd, buf, 512, 0);
        printf("\n");
        syscall_io_write(stdout, buf, read, 0);
        syscall_io_close(fd);      You, 2 weeks ago • simplify the sysca
    }

    char buf[256] = { 0 };

    long read = syscall_io_read(stdin, buf, 256, 0);
    if (read > 0)
    {
        printf("Read %d bytes from stdin", read);
        syscall_io_write(stdout, buf, read, 0);
    }

    pid_t my_pid = syscall_get_pid();
    printf("My PID: %d\n", my_pid);

    value = 3456787;

    pid_t ping_pid = syscall_spawn("/programs/kmsg-ping", 0, NULL);
    pid_t pong_pid = syscall_spawn("/programs/kmsg-pong", 0, NULL);
    printf("ping pid: %d\n", ping_pid);
    printf("pong pid: %d\n", pong_pid);

    syscall_create_thread("worker", thread_work, &value);

    my_pid = syscall_fork();
```

```
Hello, world!

This text is stored in /assets/msg.txt and is read by the
init program.
_
```

Figure 1.5: VSCode Debugging MOS