Kernel Development with Qemu (Printing "Hello World" On Bare Metal)

How to get started with kernel development on Qemu platform

October 28, 2020 · 11 min · Anton Burtsev | Suggest Changes

If you plan to start working on a new operating system kernel things get hard fast—there is a ton of low-level hardware details you have to understand and a number of design decisions you have to make (after all, why would you build yet another kernel—you have to invent something new, right?). Complexity is intimidating. Nevertheless, if you take it step by step, the basics are simple. And it's worth trying. We're sure you'll venture into your own kernel story—multicore-scalability, security, safety, fast device access,

This post provides a minimal set of steps needed to print "Hello world!" on the screen.

Qemu: the Fastest Way to Start

The fastest way to get started on a new kernel is to use a virtual machine like Qemu. Note, we like Qemu the most because it is open source (if things go wrong, you can actually debug what is going on), and good support on Linux servers (we carry most of our development remotely in a baremetal datacenter, and Qemu just works out of the box). Compared to running on real hardware (which has a long reboot cycle, and

limited debugging mechanisms), development under a virtual machine, e.g., Qemu has a much quicker cycle and gives a ton of debugging opportunities (e.g., attaching GDB, dumping page tables, understanding triple faults, etc.). Moreover, for serious kernel development, the Qemu+KVM bundle provides performance that is nearly identical to bare metal for most workloads, i.e., a nested page table slows you down by a couple of percents on workloads that touch a lot of memory, e.g., a hash table, but overall by running on Qemu+KVM you get a fair estimate of your system's performance.

Qemu can boot a multiboot-compatible kernel with the (`-kernel`) option. Multiboot is a general specification. This is convenient, it can be booted by any multiboot-compatible boot loader, e.g., GRUB—this makes it convenient the moment you are ready to test your kernel on real hardware you can boot it with GRUB (see below).

The Multiboot specification requires that the kernel binary starts with a special header. We will make this header in two steps: 1) we will use assembly to create specific header constants (it's possible to make the header in C, but what if you program your kernel in Rust?) and 2) we will use a linker script to make sure that the header is placed exactly at the beginning of the kernel binary.

Multiboot Header

A multiboot header is easy (the code is adapted from "Writing kernels that boot with Qemu and Grub" tutorial by Herbert Boss and it uses the Intel assembly):

```
; Multiboot v1 - Compliant Header for QEMU
; We use multiboot v1 since Qemu "-kernel" doesn't support
; multiboot v2

; This part MUST be 4-byte aligned, so we solve that issue using 'ALIGN 4'
ALIGN 4
section .multiboot_header
    ; Multiboot macros to make a few lines later more readable
    MULTIBOOT_PAGE_ALIGN    equ 1<<0
    MULTIBOOT_MEMORY_INFO   equ 1<<1
    MULTIBOOT_HEADER_MAGIC  equ 0x1BADB002                                  ;
```

```
MULTIBOOT_HEADER_FLAGS  equ MULTIBOOT_PAGE_ALIGN | MULTIBOOT_MEMORY_INFO ;
MULTIBOOT_CHECKSUM  equ - (MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
                          ; (magic number + checksum + flags should equal

; This is the GRUB Multiboot header. A boot signature
dd MULTIBOOT_HEADER_MAGIC
dd MULTIBOOT_HEADER_FLAGS
dd MULTIBOOT_CHECKSUM
```

Here we first define several constants, e.g., `MULTIBOOT_PAGE_ALIGN`, `MULTIBOOT_MEMORY_INFO`, `MULTIBOOT_HEADER_MAGIC`, combine them into `MULTIBOOT_HEADER_FLAGS` and `MULTIBOOT_CHECKSUM`, and then define the header as three 4 byte values in the last three lines.

While this looks criptic in practice it's rather simple. Go ahead and read the Multiboot Specification, Section 3.1.1 to see which fields are put inside the header.

We can compile this with `nasm` like

```
nasm -felf32 multiboot_header.asm -o multiboot_header.o
```

Linker Script

Now we need to compile a kernel binary in such a way that the multiboot header appears as the first section in the ELF file. For this we will use a linker scripts that instructs the linker to copy sections from multiple files in a specific order:

```
ENTRY(start)

SECTIONS {
    . = 0x100000; /* Tells GRUB to load the kernel starting at the 1MB */

    .boot :
    {
        /* Ensure that the multiboot header is at the beginning */
        *(.multiboot_header)
    }

    .text :
    {
```

```
        *(.text)
    }

}
```

Here we specify that the kernel entry point will be the `start` label. The bootloader that understand ELF format will load the kernel in memory (by the way, we ask to load the kernel at address `0x100000`) and will jump to the entry point specified in the ELF header. The linker will arrange that the ELF entry point points to the `start` label in the code. The `multiboot_header` will be copied by the linked to be above the rest of the ELF file, e.g., `text` section.

Building Hello world

Now we're ready to actually implement the code that prints "Hello world!" on the screen (this is adapted from intermezzOS).

```
global start

section .text
bits 32    ; By default, GRUB loads the kernel in 32-bit mode
start:

    ; Print `Hello world!` on the screen by placing ASCII
    ; characters in the VGA screen buffer that starts at 0xb8000
    mov word [0xb8000], 0x0248 ; H
    mov word [0xb8002], 0x0265 ; e
    mov word [0xb8004], 0x026c ; l
    mov word [0xb8006], 0x026c ; l
    mov word [0xb8008], 0x026f ; o
    mov word [0xb800a], 0x0220 ;
    mov word [0xb800c], 0x0277 ; w
    mov word [0xb800e], 0x026f ; o
    mov word [0xb8010], 0x0272 ; r
    mov word [0xb8012], 0x026c ; l
    mov word [0xb8014], 0x0264 ; d
    mov word [0xb8016], 0x0221 ; !

    hlt ; Halt CPU
```

The code moves ASCII characters `H`, `e`, `l`, etc., into the VGA frame buffer and

then halts the CPU with the `hlt` instruction.

Now we're ready to build and run this simple kernel. First, we compile the multiboot header:

```
nasm -felf32 multiboot_header.asm -o multiboot_header.o
```

Then compile the hello code (let's put it into the `boot.asm` file):

```
nasm -felf32 boot.asm -o boot.o
```

Then invoke a linker to build a final kernel ELF binary:

```
ld -m elf_i386 -n -T linker.ld -o kernel.bin boot.o multiboot_header.o
```

And now you're ready to boot with Qemu:

```
qemu-system-x86_64 -kernel kernel.bin
```

Debugging with GDB

Now lets see how we can debug our kernel with GDB. In the same folder where you work on your kernel create a simple `.gdbinit` file

```
target remote localhost:1234
symbol-file kernel.bin
```

This file instructs GDB to connect to the `localhost:1234` and load the symbol file for `kernel.bin`. Now we're ready to start our debugging session. In one terminal start Qemu

```
qemu-system-x86_64 -kernel kernel.bin -S -s
```

In another terminal simply start gdb and the `.gdbinit` will connect it to the Qemu instance running your OS

```
gdb
```

Inside GDB you can set a breakpoint on the `start` label, e.g.

```
(gdb) b start
Breakpoint 1 at 0x100010
```

Now switch layout to `regs` or `asm` and hit `c` for continue. Note, you don't get to immediately see your "Hello world" code as Qemu will executes BIOS.

```
(gdb) layout regs
(gdb) c
```

You can see your assembly sequence and can single step through it with `si`

```
(gdb) si
```

Debugging tips

You can always check if your multiboot header is correct by running

```
grub-file --is-x86-multiboot kernel.bin
```

The `grub-file` is quiet but return 0 if it finds a header, and 1 otherwise. You can check for the return code with

```
echo $?
```

You can change `--is-x86-multiboot` to `--is-x86-multiboot2` for checking the multiboot2 specification.

Automation with Make

It's much more convenient to assemble all build commands in a simple Makefile

```
kernel := kernel.bin
```

```
linker_script := linker.ld
assembly_source_files := $(wildcard *.asm)
assembly_object_files := $(patsubst %.asm, build/%.o, $(assembly_source_files))

.PHONY: all clean kernel qemu qemu-gdb

all: $(kernel)

clean:
    - @rm -fr build *.o $(kernel)
    - @rm -f serial.log

qemu: $(kernel)
    qemu-system-x86_64 -vga std -s -serial file:serial.log -kernel $(kernel)

qemu-gdb: $(kernel)
    qemu-system-x86_64 -vga std -s -serial file:serial.log -S -kernel $(kernel)

$(kernel): $(assembly_object_files) $(linker_script)
    ld -m elf_i386 -n -T $(linker_script) -o $(kernel) $(assembly_object_files)

# compile assembly files
build/%.o: %.asm
    @mkdir -p $(shell dirname $@)
    nasm -felf32 $< -o $@
```

Qemu: Booting a 64bit Kernel

Remember, Qemu supports only the mutiboot v1 specification, and will only boot a 32bit ELF binary. Since most likely you will be building a 64bit kernel the `-kernel` flag to Qemu is a somewhat limited option. An alternative way to boot is to really go through the full boot protocol with a real boot loader that can boot our kernel from some kind of a storage device. The simplest way is to create a CD-ROM image that contains the GRUB loader and your kernel. Qemu will run the BIOS and the BIOS will follow the boot protocol loading GRUB the CD-ROM device. Then GRUB that supports both Multiboot v1 and v2 will load our 64bit kernel.

To create a bootable ISO, we're going to use the `grub2-mkrescue` program that

generates a GRUB rescue image. We first create a folder layout that will contain our kernel on disk, and then use `grub2-mkrescue` to create a rescue image.

```
mkdir -p build/isofiles/boot/grub
```

The -p flag to mkdir will make the directory we specify, as well as any directories missing in the path (`-p` stands for "parent" as in parent directories). In other words, this will make the `build` directory with the `isofiles` directory inside that has `boot` inside, and finally the `grub` directory inside of that.

In other words we are creating the following layout:

```
├── build
│   ├── boot.o
│   ├── hello.iso
│   ├── isofiles
│   │   └── boot
│   │       ├── grub
│   │       │   └── grub.cfg
│   │       └── kernel.bin
```

Next, we create `grub.cfg` , a GRUB configuration file inside of that build/isofiles /boot/grub directory. The GRUB config file will instruct GRUB how to boot our kernel:

```
set timeout=0
set default=0

menuentry "hello" {
    multiboot2 /boot/kernel.bin
    boot
}
```

Note that we set the `default` GRUB entry to 0 (that's the only entry we have), and configure the `timeout` to be 0 seconds (after all we just want to boot immediately).

One additional detial here is that we switch from Multiboot v1 to v2, so we use a slightly different `multiboot_header.asm` file:

```
; Multiboot 2 - Compliant Header
; https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html (Section 3
section .multiboot_header
header_start:
    ; Multiboot macros to make a few lines later more readable
    MULTIBOOT_PAGE_ALIGN    equ 1<<0
    MULTIBOOT_HEADER_ARCH       equ 0         ; 32-bit (protected) mode of i386
    MULTIBOOT_HEADER_MAGIC  equ 0xe85250d6          ; magic number
    MULTIBOOT_CHECKSUM  equ - (MULTIBOOT_HEADER_MAGIC + (header_end - header_st
                                                    ; (magic number + check


    MULTIBOOT_TYPE      equ 0
    MULTIBOOT_FLAGS     equ 0
    MULTIBOOT_SIZE      equ 8

    ; This is the GRUB Multiboot header. A boot signature
    dd MULTIBOOT_HEADER_MAGIC
    dd MULTIBOOT_HEADER_ARCH
    dd header_end - header_start ; Size of the Header
    dd MULTIBOOT_CHECKSUM

    ; Required end tag
    dw MULTIBOOT_TYPE
    dw MULTIBOOT_FLAGS
    dd MULTIBOOT_SIZE
header_end:
```

Again, don't be shy to check the Multiboot Specification v2, Section 3.1.1 to see the exact meaning of all fields that we use above.

Now don't have to bother about compiling the 32bit kernel, and instead use normal 64bit ELF file.

```
nasm -felf64 boot.asm -o build/boot.o
nasm -felf64 multiboot_header.asm -o build/multiboot_header.o
ld -n -T linker.ld -o build/kernel.bin  build/boot.o  build/multiboot_header.o
```

Here we ask the linker to put the `kernel.bin` file inside `boot`.

We can use `grub2-mkrescue` to generate a bootable ISO image:

```
$ grub2-mkrescue -o build/hello.iso build/isofiles
```

The -o flag controls the output filename, which we choose to be `build/hello.iso`. And then we pass it the directory to make the ISO out of, which is the `build/isofiles` directory we just set up.

This will produce an `build/hello.iso` file with our kernel inside. Now we can pass this ISO file to QEMU

```
$ qemu-system-x86_64 -cdrom build/hello.iso
```

Booting off a USB stick

Copy the ISO disk image to the USB stick (make sure to use correct device for the USB drive, otherwise you can overwrite your hard disk). You can use lsblk on Ubuntu to list block devices

```
lsblk
```

For me it's `/dev/sda` or `/dev/sdb` but my laptop runs off an NVMe device, so for you `/dev/sda` may very well be your root device, not a USB!

```
sudo dd if=build/hello.iso of=/dev/<your_usb_drive> bs=1MB
sync
```

Boot on baremetal from a Linux partition

```
sudo cp build/kernel.bin /boot/
```

Add the following entry to the grub menu list. On a Linux machine this can be done by adding this to the /etc/grub.d/40_custom. You might adjust the root='hd0,2' to reflect where your Linux root is on disk, e.g., maybe it's on root='hd0,1'

```
set timeout=30
```

```
menuentry "Hello World" {
    insmod ext2
    set root='hd0,1'
    set kernel='/boot/kernel.bin'
    echo "Loading ${kernel}..."
    multiboot2 ${kernel} ${kernel}
    boot
}
```

Update grub

```
sudo sudo update-grub2
```

Reboot and choose the “Hello World” entry. Make sure that you can see the grub menu list by editing /etc/default/grub making sure that GRUB_HIDDEN_TIMEOUT_QUIET is set to “false”.

```
GRUB_HIDDEN_TIMEOUT_QUIET=false
```

Source code

The source code for this post can be found at hello-os `master` and `qemu-kernel` branches.

Resources

- intermezzOS, an operating system for learning provides an excellent and detailed overview of how to boot into Rust (including printing the “Hello World!” discussed here.
- A minimal Multiboot Kernel is a blog post that describes a minimal multiboot kernel.
- How Does an Intel Processor Boot? is a good overview of the boot process on Intel CPUs.
- Intel SGX Explained provides yet another, more in-depth overview of the boot

process on Intel platforms.

- linux-insides provides an overview of the Linux boot process.