Bootloader Programming Tools

Overview

Emulators, debuggers, and compilers are the tools of the trade for bootloader development. Here we cover the QEMU emulator, GCC toolchain, and GNU Debugger (gdb).

Tool Setup

Setup instructions target the process to set up these tools in Ubuntu 20.04. Students are recommended to use either "native" Ubuntu 20.04 or WSL2 Ubuntu 20.04 (on Windows 10).

To start, update your Ubuntu installation's package index and existing packages:

```
sudo apt update
sudo apt upgrade
```

Next, install the multi-architectural development tools and QEMU emulator for i386 and x64:

```
sudo apt install build-essential d52 gdb-multiarch
sudo apt install qemu-system-x86 gcc-arm-none-eabi qemu-system-arm
```

QEMU (Emulator)

QEMU is a command-line emulator (with optional graphical display) that supports a wide range of processors and platforms. For this lab, we will be emulating the Intel i386 platform to develop a bootloader program in a sandbox. QEMU has many features, of which we will use only a subset. Critical options, which you will need for this activity, along with an example, are listed here.

Command Line Options

`-cpu <CPU>`	Emulate the CPU (e.g., "**`-cpu 486`**")
`-M <machine>`	Emulate the specified platform (e.g., "**`-M akita`**")
`-nographic`	Do not emulate graphical device / video (i.e., serial command line only)
`-s`	Remote debugging on TCP port 1234 (short for "**`-gdb tcp::1234`**")
`-S`	Pause CPU at startup (e.g., so you can connect the debugger)
`-hda <file>`	Creates fixed storge device using specified image (e.g., "**`-hda x86.img`**")
`-device …`	Creates a device using the specified driver options

QEMU drivers and options are specified by a **driver** name (type) and a comma-separated list of properties. Each property is a pair in "**`<key>=<value>`**" format.

The main device driver/type we will use is a **loader**, which allows us to pre-load regions of memory rom binary data files. This is particularly useful on ARM platforms, where devices are frequently mapped to non-standardized memory locations. For example, on the Akita platform, we will use this driver to simulate the mapping of flash ROM into the memory space.

Loader Driver Properties

`file=<path>`	A file which contains binary data to be loaded.
`addr=<address>`	The memory address at which to load the data.
`force-raw=<val>`	Forces file to be loaded as a raw binary image – no preprocessing (**`on`** or **`off`**)

For example, this command emulates the 486 CPU via command line only, enables remote debugging, pauses the CPU at startup, and maps "x86.img" as the first fixed storage device:

```
qemu-system-i386 -cpu 486 -nographic -s -S -hda x86.img
```

This launches the x86_64 emulator (**qemu64** CPU) and maps "image.bin" to address **`0x1000`**:

```
qemu-system-x64 -cpu qemu64 -device loader,file=image.bin,addr=0x1000
```

This command would start a virtual machine emulating the "Akita" platform via command line only, enable remote debugging, QEMU, pause the CPU at startup, and map "image.bin" to the beginning of the memory address space (address **`0x0`**):

```
qemu-system-arm -M akita -nographic -s -S -device loader,file=image.bin,addr=0
```

GDB (Debugger)

The GNU Debugger (GDB) is a common command line debugging tool you have likely encountered in other courses. QEMU supports remote connections to GDB, which makes it the pair a convenient set of tools for low-level development. We will use using the multiarchitecture version of GDB, **`gdb-multiarch`**, for this activity. To start in interactive mode, you can simply run the command:

```
gdb-multiarch
```

Once in GDB, the user can execute commands. For example, we can connect to a running instance of QEMU's remote debugger port using the **`target`** command:

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

Later, you can **`disconnect`**:

```
(gdb) target disconnect
```

While the full functions of GDB are beyond the scope of this activity, there are several that are especially helpful for raw CPU control. For example, at boot, there are no functions to "step into" or "step over", just instructions to execute; likewise, there are no variables – only main memory and register values. Here are some low-level instructions that are useful when using GDB & QEMU:

Control Commands

`break [LOC] [if CND]`:	Sets a breakpoint at **LOC**; addresses signified by [*] (e.g., **`*0xF00D`**)
`stepi [NUMBER]`:	Execute one (or **NUMBER** if specified) instructions.

Informational Commands

`print [OPTS] [/FORMAT] [EXP]`:	Print value of expression in specified format
`/a`	Address notation
`/c`	Character representation
`/d`	Decimal representation
`/f`	Floating-point representation
`/t`	Binary representation
`/x`	Hexadecimal representation

`x [/[COUNT][FORMAT][UNIT]] [ADDR]`	Examine memory in specified format
`/a`	Address notation
`/c`	Character representation
`/d`	Decimal representation
`/f`	Floating-point representation
`/i`	Instruction
`/s`	String representation
`/t`	Binary representation
`/x`	Hexadecimal representation
`info <SET>`:	Get information about a set of system data
`all-registers`	All registers
`float`	All floating-point registers
`registers`	All integer registers

Note that commands can be shortened if they remain unambiguous. Here are some examples:

```
(gdb) si 5 # Execute 5 instructions (stepi; special abbreviation)
(gdb) b *0xbadf00d # Set breakpoint at address 0xbadf00d
(gdb) i r # See info on registers
(gdb) x/5i $pc # See 5 instructions, starting at program counter address
(gdb) x/8xb 0x100 # Print 8 bytes in hex starting at address 0x100
(gdb) p/x $r1 # Print value in register 1 in hexadecimal format
```

16-bit Real Mode (x86)

By default, GDB connects to QEMU i386 instances in 32-bit mode, but bootloaders instead start in 16-bit ***Real Mode*** (for backwards compatibility with 8086 / 80286 boot procedure). To view instructions as 16-bit instructions, after connecting, you can manually set the architecture:

```
(gdb) set architecture i8086
```

Build Tools (Assembler, Linker, Object Copy)

As an assembler operates on a lower level, it does not usually include many of the "freebies" that even a low-level language compiler (like **`gcc`**) includes (e.g., linking). Additionally, building a raw image requires additional steps compared to a typical executable. Generally, the steps are:

1) Generate an ***object file*** from the assembly code via the assembler (**`as`**)
2) Generate an ***executable*** from the object file via the linker (**`ld`**)
3) Dump the text segment to a ***raw binary*** from the ELF executable (**`objcopy`**)

To maximize a bootloader's usability, we should attempt to target the oldest common instruction set. The oldest forward-compatible ARM systems still in use are ARMv3 processors; similarly, i386-compatible processors are still common. As such, we should make sure our assembler (**`as`**) targets the correct platform (see **`-march`** option). While we focus here on the more documented older variants (ARM Classic and BIOS), it is important to note that both ARM and x86 lines have new boot processes for the newest lines (M-Cortex and UEFI respectively). Architecture options for the GNU Assembler are the same as for GCC in general.

Additionally, depending on the platform, the bootloader base address will vary. As noted in previous sections, for ARM, the bootloader begins execution at `0x000`, which for x86, it is loaded into `0x7c00`. Our linker (**`ld`**) will need to specify this (see **`-Ttext`** option).

Finally, we will need to extract the instructions from the executable (**`objcopy`**). We will need this to be in raw binary (**`-O`** option), and we want only the text section (**`--only-section`**). (On x86, we can skip this step, and get the linker to do the work. See if you can figure out how!)

You can also investigate binary files using the **`objdump`** utility (see man page for more information).