Lecture 3

Juni Kim

6.S913

January 23, 2026

Recap

- Build scripts that are defensive to undeclared external state (working directory, environment vars, filesystem)
- source-artifact distinction
- host-target distinction
- Careful manipulation of env vars and options to force the compiler to produce target-correct binaries.
- **Reminder: Do Attendance!**

Virtual Hardware

What is QEMU?

- An emulator that virtualizes the hardware (different from containers!)
- Sometimes used to run binaries for other architectures.
- We can test run kernels, firmware, etc.
- Drivers for virtual hardware are readily available and much simpler than real hardware.
- The kernel *thinks* it's booting on real hardware.

You should also check out the lecture 3 handout for more information on how qemu actually works.

QEMU Simulates An Entire Board

The CPU arch is not the only thing to consider!

- The CPU
- UART device (console output)
- Firmware
- Network Card
- Disks
- Memory

The kernel needs to be able to interface with all of these!

Kernel related things

What is an OS kernel really?

- Entire courses dedicated to answering this question!
- The kernel is the first program that runs after firmware and never exits.
- Privileged code with complete access to all hardware
- Userspace code is at the mercy of the kernel (plus or minus security bugs). Gets information through syscalls, talks with other processes via IPC orchestrated by the kernel, ...
- There are (non-commercial) operating systems that run everything in kernel mode (e.g. TempleOS)

In Linux, the entire "operating system" resides in the kernel.

- IPC and Virtual Memory (On All Kernels)
- Filesystem
- Device Drivers
- Networking
- Other cool stuff like virtualization, containers, ...

Just important to note for configuration

Compiling the Linux Kernel

- Configure the kernel via menuconfig (See handout 2 for information on kernel options)
- See the menu for the menuconfig on top, pressing `?` lets you see what option things correspond to
- There is no `make install`; you just extract out the kernel image from the build tree.

Device Drivers:

- You are welcome to eliminate as many of them as you can find. Almost all of them will be useless for this class.
- You should however not remove drivers for the UART devices (see handouts) and the network card.

Live Demo of Kernel Config!

Kernel Module System

- The kernel can build features either built-in or as loadable modules.
- Kernel modules must be loaded dynamically via userspace programs (eudev)
- Modules are typically placed under `/lib/modules`.
- We will ignore these and only use the core kernel binary.

The Kernel Command Line

- The kernel has no configuration files like `/etc`.
- All runtime configuration is passed as a single string at boot.
- This string is known as the **kernel command line**.
- Provided by firmware, bootloader, or the default that you compiled into the kernel.

Key Properties

- Parsed very early during kernel initialization.
- Exposed to userspace at `/proc/cmdline` (assuming you mounted your procfs there).
- Typos usually fail silently.

Serial Console

- Determines whether you will see output on your screen :)
- Correspond to physical devices on the motherboard and require drivers (usually baked into the kernel).
- The kernel will probe the system for all device types and register device files corresponding to uart devices.
- The kernel does *not* print anywhere unless a console is registered
- Consoles are selected by name via `console=<tty>` on the kernel command line
- Passing a non-existent console (e.g. `ttyAMA0` on an x86 board) results in silence
- QEMU emulates serial hardware; the kernel must include the driver
- `/dev/console` points to whatever console the kernel chose

If you are really curious about what is happening, feel free to go spelunking in `/sys/class/tty/<ttyName>`!

The Kernel as an EFI Stub

So how could we start the kernel?

- Use QEMU's "firmware" to manually launch a kernel with some command line
- Use a bootloader (GRUB, systemd-boot, etc.) to launch the kernel
- Launch the kernel via UEFI Firmware
- Getting ahead, but important to prevent you from recompiling kernel later on.
- We will be bypassing bootloaders like GRUB.
- We will place the kernel binary in a special place on the bootable disk where it can get detected by firmware
- And that firmware will launch the kernel, so we need the kernel to be able to jump start itself.

Preparing the Initramfs

FHS throwback

- You are constructing a filesystem tree that will mirror the filesystem tree on your initramfs.
- You need to create the directory where your mountpoints will exist.
- You definitely need
  - `/usr/include`, `/usr/lib`, `/usr/bin` - everything userspace programs
  - `/bin` and `/sbin` - critical system binaries
  - Pseudofilesystem mount points (see next few slides)

Filesystem Tree and Mount Points

- Before userspace runs, the kernel provides no filesystems other than the initramfs by default.
- Your init script must mount essential filesystems manually.
- Mount points must exist before mounting.
- You can inspect mount points by `lsblk` (only block devices) or `df -Th` (see all filesystems)

```
mount -t devtmpfs devtmpfs /dev
mount -t sysfs sys /sys
mount -t tmpfs run /run
mount -t proc proc /proc
# mounting a disk partition is comparatively more straightforward.
mount /dev/vda2 /mnt
```

Pseudo Filesystems

- Exposed by the kernel to interact with userspace
- Main ones are `/proc`, `/sys`, `/dev`, `/tmp`, and `/run`.
- Tip: you should download useless files to `/tmp` instead of Downloads folder, that way they get erased on reboot!

Useful device files

- `/dev/random` - constant stream of random bits
- `/dev/zero` - stream of zero bits
- `/dev/null` - black hole (used in scripts to throw away output)
- `/dev/vda`, `/dev/vda1`, … - what your on-disk block devices will appear.

Live demo of device files!

What is an initramfs?

- An archive that the kernel unpacks when first booting (usually in the form of a cpio file).
- MUST contain an executable script at `/init` (you cannot configure this path)
- The kernel executes `/init` as PID 1.
- This gives you the opportunity to mount filesystems and switch root to a filesystem on your disk (later lecture!).

More about PID's

- Each process gets a unique process ID.
- When a parent process dies, the child processes all die. **note: briefly cover fork**
- The kernel will just load up PID 1 and let it do everything else.
- PID 1 must never exit.

Embedding an initramfs

- There are cases where the kernel has to start by itself
- E.g. when used as an EFI stub
- Can't rely on an external initramfs archive, must have all contents in the kernel binary.
- There is no filesystem if the kernel bootstraps itself!
- Options in the kernel let you do this (see handout).

`switch_root` to actual init

- We run a `/init` script inside the initramfs and can get a shell.
- But nothing is persistent, there is no real disk here.
- `switch_root` helps us switch to an actual filesystem.
- It only works when you are PID 1!
- Different from chroot in that it changes the global filesystem namespace

```
# /dev/vda2 is the device representing our root filesystem.
/bin/mount /dev/vda2 /mnt
# /mnt/sbin/init is a proper PID 1.
# you now run the switch_root here.
```

Next Lecture

- What does real init do?
- What are some essential services busybox must help set up?
- How do bootable images get read? How do partitions work?