

Zircon 笔记

许中兴

May 24, 2018

Contents

1	Bootloader	2
1.1	mkbootfs 一些细节	2
2	ARM 架构	9
2.1	关于 MMU	9
2.2	Exceptions	9
3	Development	10
4	板子	10
5	boot-shim	13
6	start.S	13
6.1	副 cpu	15
7	lk_main	17
7.1	thread_init_early	17
8	System Calls	19
9	userboot	20
10	设备模型和驱动	23
11	线程	24
12	虚拟内存管理	25

1 Bootloader

让 make 输出详细的编译过程。

```
make V=1 arm64 > make.log
```

在 qemu 里运行 Zircon 的命令为：

```
qemu-system-aarch64 -m 2048 -nographic -net none -smp 4 \
    -kernel /Users/xzx/zircon/build-arm64/qemu-zircon.bin \
    -machine virtualization=true -cpu cortex-a53 \
    -machine virt,gic_version=3 \
    -initrd /Users/xzx/zircon/build-arm64/bootdata.bin \
    -append 'TERM=xterm-256color \
    kernel.entropy-mixin=a7324cb430cbac8... \
    kernel.halt-on-panic=true '

./build-arm64/tools/mkbootfs --target=boot -c -o ./build-arm64/bootdata.bin
./build-arm64/bootfs.manifest
```

可以看出内核文件是 qemu-zircon.bin。我们在 make.log 里找一下这个文件是怎么来的。

```
./build-arm64/tools/mkbootfs -o build-arm64/qemu-zircon.bin \
    ./build-arm64/zircon.bin --header ./build-arm64/qemu-boot-shim.bin \
    --header-align 65536
```

mkbootfs 会把 qemu-zircon.bin 所代表的内核映像写在 offset 为 65536 的地方。在之前写入的是 boot-shim 的内容。qemu-zircon.bin 的布局实际上是由 image.S 决定的。

1.1 mkbootfs 一些细节

1. system = true, ramdisk = false, header_align = 65536
2. 调用import_file(path, system, ramdisk)
 - (a) 接着调用import_file_as(fn, ITEM_BOOTDATA, hdr.length, &hdr)
 - (b) 调用import_directory_entry("bootdata", fn, &s),构造一个 fsentry, length 是输入文件的长度。
 - (c) 构造first_item, type=ITEM_BOOTDATA, first, last都指向fsentry
3. 进入write_bootdata(output_file, first_item, header_path, header_align)
 - (a) 首先把 header 原封不动的写入目标文件
 - (b) 然后 seek 到 header align 的位置
 - (c) 然后再留出一个bootdata_t 的位置
 - (d) copybootdatafile(fd, item->first->srcpath, item->first->length)

- i. 首先把BOOTDATA_CONTAINER 读到 hdr 里
 - ii. 然后把剩下的内核映像原封不动的写入目标文件
- (e) 把BOOTDATA_CONTAINER 再写入之前留出来的bootdata_t 的位置

qemu-boot-shim.bin 的来历:

```
./prebuilt/downloads/gcc/bin/aarch64-elf-objcopy -O binary \
    build-arm64/boot-shim/qemu/boot-shim.elf build-arm64/qemu-boot-shim.bin
```

boot-shim.elf 的来历:

```
./prebuilt/downloads/gcc/bin/aarch64-elf-ld -nostdlib --build-id -z noexecstack \
    -z max-page-size=4096 --gc-sections --emit-relocs --build-id=none \
    ./build-arm64/boot-shim/qemu/boot-shim.S.o \
    ./build-arm64/boot-shim/qemu/boot-shim.c.o \
    ./build-arm64/boot-shim/qemu/debug.c.o \
    ./build-arm64/boot-shim/qemu/devicetree.c.o \
    ./build-arm64/boot-shim/qemu/util.c.o \
    -T kernel/target/arm64/boot-shim/boot-shim.ld \
    -o build-arm64/boot-shim/qemu/boot-shim.elf
```

boot-shim.S 中 `_start` 的地址是 0。所以 qemu-zircon.bin 加载之后, 自然就
从 boot-shim.S 的 `_start` 开始执行。那么, 内核的 `_start` 又是如何找到的呢?
注意到有这样一段代码:

```
// x0: pointer to device tree
// x1: pointer to kernel bootdata container
bl      boot_shim

// kernel entry point is returned in x0
mov     tmp, x0

// pass bootdata to kernel in x0
adr     x0, bootdata_return
ldr     x0, [x0]

br      tmp
```

也就是说, `boot_shim` 函数应该会找到 kernel 的 entry point。我们去看一下。

```
return kernel_base + kernel->data_kernel.entry64;
```

这里 `kernel_base` 就是 `KERNEL_ALIGN=65536`。`boot_shim` 函数从 bootdata 中获得入口地址。bootdata 是 `mkbootfs` 这个工具填写到启动映像里的。在 `image.S` 的头上就是 bootdata。

```
// BOOTDATA_KERNEL payload (bootdata_kernel_t)
DATA(_bootdata_kernel_payload)
    // The boot-shim code expects this to be an offset from the beginning
    // of the load image, whatever the kernel's virtual address.
    .quad IMAGE_ELF_ENTRY - _bootdata_file_header
    .quad 0
END_DATA(_bootdata_kernel_payload)
```

从而，上面的 `br tmp` 能够正确跳转到内核入口。

`boot-shim.S` 文件里设置了 `stack pointer`，指向了 `boot-shim` 之后，`kernel` 之前的一块大小为 4k 的内存区域。

`zircon.bin` 的来历：

```
./prebuilt/downloads/gcc/bin/aarch64-elf-objcopy \
    -O binary build-arm64/zircon-image.elf build-arm64/zircon.bin
```

`zircon-image.elf` 的来历：

```
./prebuilt/downloads/gcc/bin/aarch64-elf-ld -nostdlib --build-id -z noexecstack \
    -z max-page-size=4096 --gc-sections --build-id=none \
-o build-arm64/zircon-image.elf -T kernel/image.ld \
    --just-symbols ./build-arm64/zircon.elf \
./build-arm64/kernel-vars.ld ./build-arm64/zircon.image.o
```

链接脚本 `image.ld` 内容：

```
ENTRY(IMAGE_ELF_ENTRY)

SECTIONS {
    . = IMAGE_LOAD_START;

    .load_image : {
        KEEP(*(.text))
    } :load_image

    /*
     * When a boot loader is actually using the ELF headers, it needs to
     * know how much memory to reserve after the load image (p_filesz is
     * the load image, and p_memsz > p_filesz to indicate the extra space
     * to reserve). This ensures that the segment has the right p_memsz.
     */
    .bss : {
        . += ABSOLUTE(IMAGE_MEMORY_END) - ABSOLUTE(.);
        ASSERT(ABSOLUTE(.) == ABSOLUTE(IMAGE_MEMORY_END), "image layout bad");
    }
}

PHDRS {
```

```
    load_image PT_LOAD FLAGS(7); /* PF_R|PF_W|PF_X */
}
```

ENTRY 指定了程序入口的虚拟地址，它对应于 ELF Header 中的 entry 项。IMAGE_ELF_ENTRY 的定义在 kernel/arch/arm64/start.S 中。

```
// This symbol is used by image.S.
.global IMAGE_ELF_ENTRY
IMAGE_ELF_ENTRY = _start
```

实际上在最终生成的内核映像中 ELF 结构是不存在的，所以上面的 ENTRY 信息并没有实际的用处。

IMAGE_LOAD_START 是在 kernel.ld 中定义的符号，它的值是-4GB

```
. = KERNEL_BASE;
PROVIDE_HIDDEN(__code_start = .);
...
```

```
IMAGE_LOAD_START = __code_start;
```

KERNEL_BASE 定义在 kernel/arch/arm64/rules.mk

```
KERNEL_BASE := 0xffffffff00000000
```

整个内核映像的起始虚拟地址是-4GB，但是入口并不是这个地址，而是 start.S 文件中的 _start。我们在上面已经看到 boot-shim 是如何找到这个入口地址的。

这里 zircon.elf 只是用来获得 symbol 的地址，并不添加到 zircon-image.elf 里。真正的内核映像文件是 zircon.image.o

```
./prebuilt/downloads/gcc/bin/aarch64-elf-gcc \
-O2 -g -fdebug-prefix-map=/Users/xzx/zircon=. -finline \
  -include ./build-arm64/config-global.h -Wall -Wextra -Wno-multichar \
  -Werror -Wno-error=deprecated-declarations -Wno-unused-parameter \
  -Wno-unused-function -Werror=unused-label -Werror=return-type -fno-common \
  -ffunction-sections -fdata-sections -Wno-nonnull-compare -ffreestanding \
  -include ./build-arm64/config-kernel.h -Wformat=2 -Wvla -Wformat-signedness \
  -fno-exceptions -fno-unwind-tables -fno-omit-frame-pointer -mgeneral-regs-only \
  -fPIE -include kernel/include/hidden.h -mcpu=cortex-a53 -ffixed-x18 \
  -Isystem/public -Isystem/private -I./build-arm64/gen/global/include \
  -I./build-arm64 -Ikernel/include -Ikernel/platform/generic-arm/include \
  -Ikernel/arch/arm64/include -Ikernel/top/include \
  -Ikernel/dev/hdcp/amlogic_s912/include \
  -Ikernel/dev/interrupt/arm_gic/common/include \
  -Ikernel/dev/interrupt/arm_gic/v2/include \
  -Ikernel/dev/interrupt/arm_gic/v2/include \
  -Ikernel/dev/interrupt/arm_gic/v3/include \
```

```

-Ikernel/dev/iommu/dummy/include -Ikernel/dev/pcie/include \
-Ikernel/dev/pdev/include -Ikernel/dev/pdev/power/include \
-Ikernel/dev/power/hisi/include -Ikernel/dev/psci/include \
-Ikernel/dev/timer/arm_generic/include -Ikernel/dev/uart/amlogic_s905/include \
-Ikernel/dev/uart/nxp-imx/include -Ikernel/dev/uart/pl011/include \
-Ikernel/kernel/include -Isystem/ulib/bitmap/include \
-Ikernel/lib/bitmap/include -Ikernel/lib/cbuf/include \
-Ikernel/lib/debugcommands/include -Ikernel/lib/debuglog/include \
-Ikernel/lib/ktrace/include -Ikernel/lib/memory_limit/include \
-Ikernel/lib/mtrace/include -Ikernel/lib/userboot/include \
-Ikernel/lib/version/include -Ikernel/object/include \
-Ikernel/platform/include -Ikernel/syscalls/include \
-Ikernel/target/include -Ikernel/tests/include \
-Ikernel/dev/interrupt/include -Ikernel/dev/pdev/interrupt/include \
-Ikernel/dev/pdev/uart/include -Ikernel/dev/udisplay/include \
-Ikernel/lib/console/include -Ikernel/lib/counters/include \
-Ikernel/lib/crypto/include -Ikernel/lib/debug/include \
-Isystem/ulib/explicit-memory/include -Ikernel/lib/explicit-memory/include \
-Ikernel/lib/fbl/include -Isystem/ulib/fbl/include \
-Ikernel/lib/fbl/include -Ikernel/lib/fixed_point/include \
-Ikernel/lib/header_tests/include -Ikernel/lib/heap/include \
-Ikernel/lib/heap/include -Ikernel/lib/hypervisor/include \
-Ikernel/lib/libc/include -Ikernel/lib/oom/include -Ikernel/lib/pci/include \
-Ikernel/lib/pci/include -Ikernel/lib/pow2_range_allocator/include \
-Isystem/ulib/region-alloc/include -Ikernel/lib/region-alloc/include \
-Ikernel/lib/unittest/include -Ikernel/lib/user_copy/include \
-Ikernel/lib/vdso/include -Isystem/ulib/zxcpp/include \
-Ikernel/lib/zxcpp/include -Ikernel/vm/include \
-Ikernel/arch/arm64/hypervisor/include -Ikernel/dev/hw_rng/include \
-Ikernel/lib/gfx/include -Ikernel/lib/gfxconsole/include \
-Ikernel/lib/heap/cmpctmalloc/include -Ikernel/lib/io/include \
-Isystem/ulib/pretty/include -Ikernel/lib/pretty/include \
-Ithird_party/ulib/cryptolib/include -Ithird_party/lib/cryptolib/include \
-Ithird_party/lib/jitterentropy/include -Ithird_party/lib/jitterentropy/include \
-Ithird_party/ulib/qrcodegen/include -Ithird_party/lib/qrcodegen/include \
-Ithird_party/ulib/uboringssl/include -Ithird_party/lib/uboringssl/include \
-I./build-arm64 \
-c kernel/arch/arm64/image.S -MD -MP -MT build-arm64/zircon.image.o \
-MF build-arm64/zircon.image.d -o build-arm64/zircon.image.o

```

image.S 文件里嵌入了真正的内核 zircon.elf.bin

```

./prebuilt/downloads/gcc/bin/aarch64-elf-objcopy \
-O binary build-arm64/zircon.elf build-arm64/zircon.elf.bin

```

真正的 kernel 是 zircon.elf

```

./prebuilt/downloads/gcc/bin/aarch64-elf-ld -nostdlib --build-id -z noexecstack \
-z max-page-size=4096 --gc-sections --emit-relocs -T kernel/kernel.lds \
build-arm64/kernel-vars.lds \
build-arm64/kernel/platform/generic-arm/generic-arm.mod.o \
build-arm64/kernel/arch/arm64/arm64.mod.o build-arm64/kernel/top/top.mod.o \
build-arm64/kernel/dev/hdcp/amlogic_s912/amlogic_s912.mod.o \
build-arm64/kernel/dev/interrupt/arm_gic/common/common.mod.o \
build-arm64/kernel/dev/interrupt/arm_gic/v2/v2.mod.o \
build-arm64/kernel/dev/interrupt/arm_gic/v3/v3.mod.o \
build-arm64/kernel/dev/iommu/dummy/dummy.mod.o \
build-arm64/kernel/dev/pcie/pcie.mod.o \
build-arm64/kernel/dev/pdev/pdev.mod.o \
build-arm64/kernel/dev/pdev/power/power.mod.o \
build-arm64/kernel/dev/power/hisi/hisi.mod.o \
build-arm64/kernel/dev/psci/psci.mod.o \
build-arm64/kernel/dev/timer/arm_generic/arm_generic.mod.o \
build-arm64/kernel/dev/uart/amlogic_s905/amlogic_s905.mod.o \
build-arm64/kernel/dev/uart/nxp-imx/nxp-imx.mod.o \
build-arm64/kernel/dev/uart/pl011/pl011.mod.o \
build-arm64/kernel/kernel/kernel.mod.o \
build-arm64/kernel/lib/bitmap/bitmap.mod.o \
build-arm64/kernel/lib/cbuf/cbuf.mod.o \
build-arm64/kernel/lib/debugcommands/debugcommands.mod.o \
build-arm64/kernel/lib/debuglog/debuglog.mod.o \
build-arm64/kernel/lib/ktrace/ktrace.mod.o \
build-arm64/kernel/lib/memory_limit/memory_limit.mod.o \
build-arm64/kernel/lib/mtrace/mtrace.mod.o \
build-arm64/kernel/lib/userboot/userboot.mod.o \
build-arm64/kernel/lib/version/version.mod.o \
build-arm64/kernel/object/object.mod.o \
build-arm64/kernel/platform/platform.mod.o \
build-arm64/kernel/syscalls/syscalls.mod.o \
build-arm64/kernel/target/target.mod.o \
build-arm64/kernel/tests/tests.mod.o \
build-arm64/kernel/dev/interrupt/interrupt.mod.o \
build-arm64/kernel/dev/pdev/interrupt/interrupt.mod.o \
build-arm64/kernel/dev/pdev/uart/uart.mod.o \
build-arm64/kernel/dev/udisplay/udisplay.mod.o \
build-arm64/kernel/lib/console/console.mod.o \
build-arm64/kernel/lib/counters/counters.mod.o \
build-arm64/kernel/lib/crypto/crypto.mod.o \
build-arm64/kernel/lib/debug/debug.mod.o \
build-arm64/kernel/lib/explicit-memory/explicit-memory.mod.o \
build-arm64/kernel/lib/fbl/fbl.mod.o \
build-arm64/kernel/lib/fixed_point/fixed_point.mod.o \
build-arm64/kernel/lib/header_tests/header_tests.mod.o \

```

```

build-arm64/kernel/lib/heap/heap.mod.o \
build-arm64/kernel/lib/hypervisor/hypervisor.mod.o \
build-arm64/kernel/lib/libc/libc.mod.o \
build-arm64/kernel/lib/oom/oom.mod.o \
build-arm64/kernel/lib/pci/pci.mod.o \
build-arm64/kernel/lib/pow2_range_allocator/pow2_range_allocator.mod.o \
build-arm64/kernel/lib/region-alloc/region-alloc.mod.o \
build-arm64/kernel/lib/unittest/unittest.mod.o \
build-arm64/kernel/lib/user_copy/user_copy.mod.o \
build-arm64/kernel/lib/vdso/vdso.mod.o \
build-arm64/kernel/lib/zxcpp/zxcpp.mod.o \
build-arm64/kernel/vm/vm.mod.o \
build-arm64/kernel/arch/arm64/hypervisor/hypervisor.mod.o \
build-arm64/kernel/dev/hw_rng/hw_rng.mod.o \
build-arm64/kernel/lib/gfx/gfx.mod.o \
build-arm64/kernel/lib/gfxconsole/gfxconsole.mod.o \
build-arm64/kernel/lib/heap/cmpctmalloc/cmpctmalloc.mod.o \
build-arm64/kernel/lib/io/io.mod.o \
build-arm64/kernel/lib/pretty/pretty.mod.o \
build-arm64/third_party/lib/cryptolib/cryptolib.mod.o \
build-arm64/third_party/lib/jitterentropy/jitterentropy.mod.o \
build-arm64/third_party/lib/qrcodegen/qrcodegen.mod.o \
build-arm64/third_party/lib/uboringssl/uboringssl.mod.o \
-o build-arm64/zircon.elf

```

kernel 在链接的时候，以-4G 为基地址。也就是说，内核里所有的东西的虚拟地址都在-4G 地址之上。

2 ARM 架构

sp 指的是 el0 的 stack pointer。其他的 stack pointer 需要指明 el 级别。在 Zircon 里, el1 使用的是 **sp_el1**, 但是在 **arm64_elX_to_el1** 里, **sp_el1** 被设置为与 **sp** 相同的值。

pc 寄存器是专用寄存器, 不能再被显式指代。

frame pointer 不是专用寄存器, 是 X29, 但是 A64 Procedure Call Standard 把 X29 定义为专门的 **frame pointer**. X30 是 **link register**.

异常级别定义程序运行的特权级别。

安全状态下处理器能访问安全内存地址空间。

scr_el3 寄存器定义 EL0 和 EL1 的安全状态。

ELR 存放异常返回地址。PLR 存放函数调用返回地址。

处理异常之前, 处理器状态 (PSTATE) 会保存在 SPSR 中。

2.1 关于 MMU

EL1 有 2 个页表基地址指针寄存器: **ttbr0_el1** 和 **ttbr1_el1**。使用哪一个由虚拟地址的最高几位决定, 如果都是 0 则使用 **ttbr0_el1**, 如果都是 1 则使用 **ttbr1_el1**。具体检查多少位由 **tcr_el1** 的 **t0sz** 和 **t1sz** 决定。在 Zircon 里 **t0sz=22, t1sz=16**。

IPS 设置为 1TB。

ESR_ELn 寄存器存放了当前的异常具体是什么异常的信息。

2.2 Exceptions

sync_exception #(ARM64_EXCEPTION_FLAG_LOWER_EL), 1

```
.macro sync_exception, exception_flags, from_lower_el_64=0
    start_isr_func
    regsave_long
    mrs x9, esr_el1
.if \from_lower_el_64
    // If this is a syscall, x0-x7 contain args and x16 contains syscall num.
    // x10 contains elr_el1.
    lsr x11, x9, #26                // shift esr right 26 bits to get ec
    cmp x11, #0x15                  // check for 64-bit syscall
    beq arm64_syscall_dispatcher    // and jump to syscall handler
.endif
    // Prepare the default sync_exception args
    mov x0, sp
    mov x1, ARM64_EXCEPTION_FLAG_LOWER_EL
    mov w2, w9
    bl arm64_sync_exception
    b arm64_exc_shared_restore_long
.endm
```

3 Development

zedboot means: after booting the target, the target will through network get the real OS files on to the internal storage.

问题 1: 没有特殊的 bootloader 能否启动 vim2

问题 2: zbi 是否会根据 o 里面的 board name 来选择输入文件。不会。vim2 不需要特殊的 bootimage.

问题 3: fastboot flash 后面的参数 boot part 有哪一种。是什么意思。emmc 如何分区的。在安卓里, fastboot flash recovery xxx.img 在 flash-hikey 里, 有很多种。

fastboot 是通过 usb-c 线 flash 过去的。

串口应该只是输出调试信息

zedboot 是通过网络传输文件的。

板子上的 emmc 有一些分区。这些分区也可以用 sd card 启动之后, 用 uboot 重新分。

fastboot 是一种协议, 可以进入 fastboot 模式, 之后可以 flash 各种分区。包括 bootloader 和 boot. 分区用名字来指定。

fastboot 模式是 bootloader 的一种模式, 有多种方式可以进入。要素就是要进入 bootloader (比如 uboot) 的启动过程中, 给它一个信号, 比如按住某个键, 短接某个跳线, 或者是从串口给它一个信号, 它就会进入 fastboot 模式。如果没有信号, 它就正常启动, 加载内核。

先更新 bootloader, 再更新 boot.img. 然后启动到 zedboot 模式下面, 主机要启动好 fx boot 服务器。

启动到 zedboot 模式之后, 会自动开始 pave. 之后完整的系统就安装到 emmc 上面了。

4 板子

first, the SoC loads the BL1 (first lever bootloader) from maskrom, then depending on state described by the flowchart in the S912 datasheet, BL1 loads and authenticates BL2 from external storage.

BL2 is mainly in charge of setting up DDR and clocks, then it loads and authenticates BL30 from the same storage as itself.

BL30 is in charge of various things like setting up (undisclosed) security stuff and loading the firmware of the cortex-M3 core that handles power-management and CPU cores DVFS (also known as SCPI in ARM terminology).

BL2 also loads/authenticates BL31, also called the secure monitor, running in trustzone and in charge of providing secure services to the main OS, and finally BL33, which is u-boot.

通过串口可以看到启动信息:

bl30, bl31 and bl33 are stored in a FIP image on eMMC. you can see that along with the flash offsets in the serial console traces output by BL2 just before u-boot starts.

eg. on the kvim2 pro:

```
Load fip header from eMMC, src: 0x0000c200, des: 0x01400000, size: 0x00004000
New fip structure!
Load bl30 from eMMC, src: 0x00010200, des: 0x01100000, size: 0x0000d600
Load bl31 from eMMC, src: 0x00020200, des: 0x05100000, size: 0x0002c600
Load bl33 from eMMC, src: 0x00050200, des: 0x01000000, size: 0x00055000
```

the 'src' values are offset in eMMC (probably bytes, because all values are aligned to 0x200, which is standard sector size), and 'des' are RAM addresses.

文档地址:<https://github.com/ARM-software/arm-trusted-firmware/tree/master/docs>
通过 adb shell 或者 u-boot 的命令行可以看到 emmc 的分区。

```
dmesg | grep mmcblk0p
```

```
[ 2.849364@2] [mmcblk0p01] bootloader offset 0x000000000000, size 0x000000400000
[ 2.849511@1] [mmcblk0p02] reserved offset 0x000002400000, size 0x000004000000
[ 2.849645@2] [mmcblk0p03] cache offset 0x000006c00000, size 0x000020000000
[ 2.849773@1] [mmcblk0p04] env offset 0x000027400000, size 0x000008000000
[ 2.849897@2] [mmcblk0p05] logo offset 0x000028400000, size 0x000002000000
[ 2.850021@1] [mmcblk0p06] recovery offset 0x00002ac00000, size 0x000002000000
[ 2.850156@2] [mmcblk0p07] rsv offset 0x00002d400000, size 0x000008000000
[ 2.850290@1] [mmcblk0p08] tee offset 0x00002e400000, size 0x000008000000
[ 2.850422@2] [mmcblk0p09] crypt offset 0x00002f400000, size 0x000002000000
[ 2.850559@1] [mmcblk0p10] misc offset 0x000031c00000, size 0x000002000000
[ 2.850691@2] [mmcblk0p11] boot offset 0x000034400000, size 0x000002000000
[ 2.850830@1] [mmcblk0p12] system offset 0x000036c00000, size 0x000080000000
[ 2.850959@2] [mmcblk0p13] data offset 0x0000b7400000, size 0x000dd8400000
```

```
kvim2:/dev/block # hexdump -n 520 /dev/block/mmcblk0boot0
00000000 0000 0000 0000 0000 0000 0000 0000 0000 0000
*
00002000 9fa1 4f92 313d 34c4
0000208
kvim2:/dev/block # hexdump -n 520 /dev/block/mmcblk0boot1
00000000 0000 0000 0000 0000 0000 0000 0000 0000 0000
*
00002000 9fa1 4f92 313d 34c4
0000208
kvim2:/dev/block # hexdump -n 520 /dev/block/mmcblk0
00000000 0000 0000 0000 0000 0000 0000 0000 0000 0000
*
00002000 9fa1 4f92 313d 34c4
0000208
kvim2:/dev/block # hexdump -n 520 /dev/block/boot
boot bootloader
kvim2:/dev/block # hexdump -n 520 /dev/block/bootloader
00000000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
*  
0000200 9fa1 4f92 313d 34c4  
0000208
```

you can also look around in `/sys/class/mmc_host/emmc/emmc:0001/block/mmcblk0/`

this source code might also provide you some hints:

```
https://github.com/khadas/linux/blob/khadas-vim-4.9.y/drivers/amlogic/mmc/emmc\_partitions.c  
https://github.com/khadas/u-boot/blob/ubuntu/common/store\_interface.c#L1174
```

I don't know if the FIP format has a human-readable specification, however you can take a look at fiptool's source to see how it's working:

```
https://github.com/ARM-software/arm-trusted-firmware/tree/master/tools/fiptool  
https://github.com/ARM-software/arm-trusted-firmware/blob/master/include/tools\_share/firmware
```

note also that some of the items in the amlogic boot images may be encrypted, so you may not read them directly (check the calls to `aml_encrypt_$(SOC)` in u-boot's Makefile for that)

5 boot-shim

qemu 的内存布局。物理内存从 0x40000000 开始，最初的 512kb 存放 FDT device tree 数据。kernel 被放置在 0x40080000。init ramdisk 放置在 0x48000000 开始的地方。

启动的一开始，x0 放的是 0x40000000(device tree 的地址)。

从 device tree 里获得的是内存范围，initrd 的地址。kernel 启动命令行。然后会在 initrd 的后面追加一些 bootdata, 包括硬件信息，内存区域信息，命令行等。

6 start.S

内核真正的入口是 start.S 里面的 `_start`。
一些基础知识。

http://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt

The `.eh_frame` section shall contain 1 or more Call Frame Information (CFI) records. The number of records present shall be determined by size of the section as contained in the section header. Each CFI record contains a Common Information Entry (CIE) record followed by 1 or more Frame Description Entry (FDE) records. Both CIEs and FDEs shall be aligned to an addressing unit sized boundary.

`.cfi_startproc` is used at the beginning of each function that should have an entry in `.eh_frame`. It initializes some internal data structures. Don't forget to close the function by `.cfi_endproc`.

Unless `.cfi_startproc` is used along with parameter `simple` it also emits some architecture dependent initial CFI instructions.

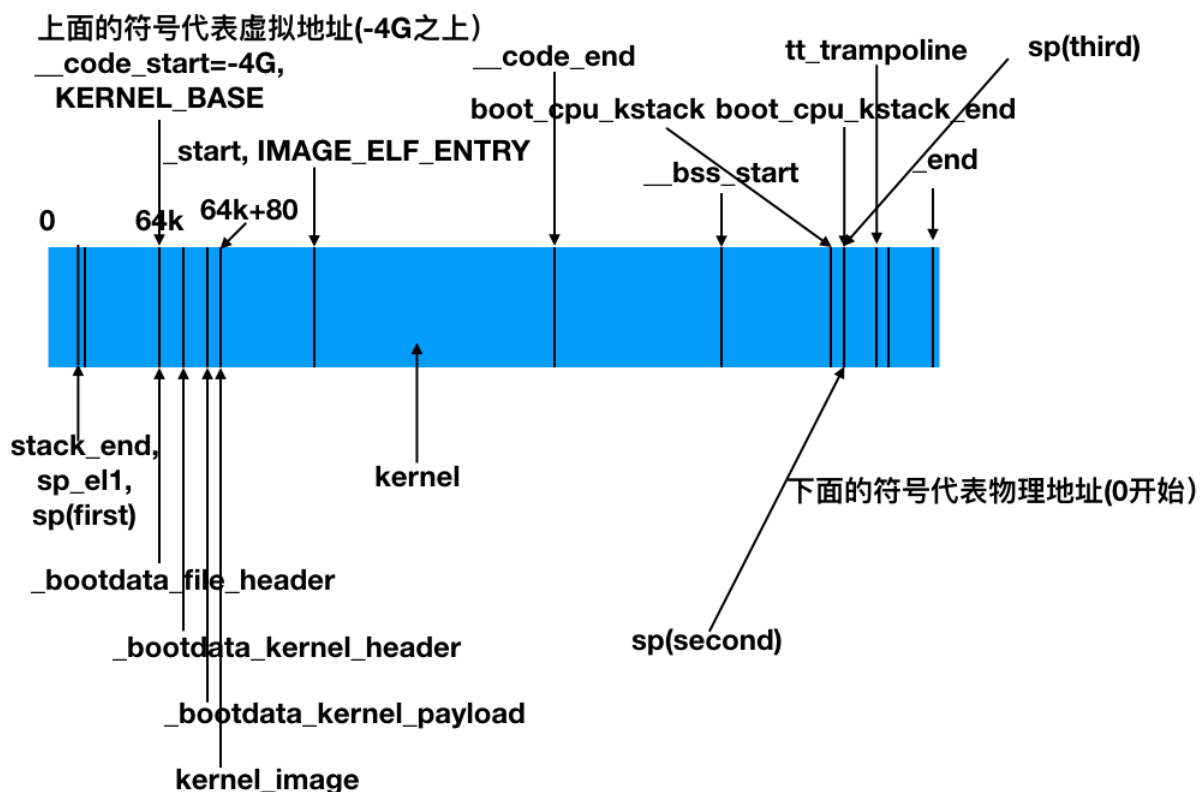
1. 获取MPIDR_EL1 寄存器的 Aff0 和 Aff1 16bit 信息。主 cpu 执行保存启动信息的代码。
2. 把 x0 的内容，也就是内核 header 的地址保存在 `bootdata_paddr` 里。
`str x0, [tmp, #:1012:bootdata_paddr]` 的形式为什么这么奇怪？
因为 `adrp` 指令是拿 4kb 对齐的页地址。低 12 位是丢掉的，所以需要上面这个形式再拿一次低 12 位的相对地址（这个形式应该在静态链接的时候被 `ld` 改写成实际的数值），加到 4kb 对齐的页地址上。为什么不用 `adr` 指令直接取址，因为 `adr` 只能取和当前 PC 相差不超过 1MB 的符号的地址。`bootdata_paddr` 与当前位置的距离应该已经超过了 1MB
具体解释：`adrp tmp, bootdata_paddr` 将 `bootdata_paddr` 相对于 PC 的偏移量（低 12 位清零），再加上 PC（低 12 位清零），存入 `tmp` 中。
`str x0, [tmp, #:1012:bootdata_paddr]` 将 `bootdata_paddr` 相对于 PC 的偏移量的低 12 位（这个值是 linker 算出来写入指令的，as 只是放一个重定位项在目标代码中）加到 `tmp` 上，作为 `str` 的目标地址。
3. 把内核入口地址 `_start` 存入 `kernel_entry_paddr`。这个地址在之后启动副 cpu 的地方 `psci_cpu_on()` 会用到。

4. 把当前异常级别存入`arch_boot_el`
5. 调用`arm64_elX_to_el1`, 将处理器异常级别置为 1。如果有 EL3 的话, 打开 HVC 指令, `mov x9, #((0b1111 << 6) | (0b0101))` 返回到的是 EL1h 模式。使用 `SP_EL1`
6. 调用`arch_invalidate_cache_all`,
7. 打开 icache, dcache
8. `__data_end` 的定义在 `kernel.ld` 中。调用 `image.S` 中的`apply_fixups` `image.S` 里只定义了一个`.text` 段。`fixup` 现在没有实际用处。将来, 因为 `kernel` 在 `ld` 链接时, 用的基地址是固定的-4G, 但是未来内核可能会被加载到一个任意的虚拟地址上。所以需要进行 `fixup`. 也就是说, `kernel_vaddr` 会保存一个任意地址。
9. `tt_trampoline` 存放了 8 个页表项。`arm64_kernel_translation_table` 的定义在 `mmu.cpp` 里。它的大小之后再研究。
10. 如果不是主 cpu, 则跳转到`Lmmu_enable_secondary`, 下面是主 cpu 逻辑。
11. `__bss_start` 的定义在 `kernel.ld` 里。把 `bss` 段清零。
12. 把 `sp` 设置到`boot_cpu_kstack_end`, 在 `bss` 里。
13. 调用`boot_alloc_init`, 把整个内核结束的位置的物理地址记录在 C++ 变量中。这个用作后来分配物理页表使用。把内核开始的物理地址`__code_start` 保存到 C++ 变量里。
14. 把`arm64_kernel_translation_table` 清零
15. 把物理地址 0 映射到内核地址空间 `ffff000000000000`, 范围是 512GB。物理页表的分配从内核结束的位置`_end` 开始。
16. 把物理地址`__code_start` 映射到虚拟地址`kernel_vaddr(-4GB)` 上, 范围就是内核的长度, 到`_end` 为止
17. 用一个 512MB 的 block 做恒等映射。计算中用到的一些常量:


```
MMU_IDENT_SIZE_SHIFT = 42,
MMU_IDENT_PAGE_SIZE_SHIFT = 16,
MMU_IDENT_TOP_SHIFT = 29,
MMU_PAGE_TABLE_ENTRIES_IDENT_SHIFT = 10,
MMU_PAGE_TABLE_ENTRIES_IDENT = 1 << 10,
KERNEL_ASPACE_BITS = 48,
MMU_KERNEL_SIZE_SHIFT = 48
```

18. 打开 MMU，为什么要用 trampoline 进行过渡？因为我们需要做二件事：打开 MMU，设置 PC 到虚拟地址上。如果先打开 MMU，这时 PC 还指向物理地址，而物理地址的直接映射没有设置的话，cpu 就找不到下一条指令了。如果先通过 br 指令弹跳 PC，这时 MMU 还没有打开，PC 指向虚拟地址也会找不到下一条指令。所以必须先设置好直接映射的地址，然后打开 MMU，然后弹跳 PC 到高端虚拟地址上，最后关闭 MMU
19. 再设置一次 stack pointer, 这一次是虚拟地址了。因为adr_global 是 PC 相对计算地址。这时 PC 已经是虚拟地址了。
20. 调用lk_main 进入 C 的世界

整个内核至此的内存映像如下图所示。



6.1 副 cpu

1. 切换到 el1

2. 等待主 cpu 把`arm64_secondary_sp_list` 设置好。在此之前会执行 `wfe`。如果在主 cpu 设置之前执行的话, 副 cpu 会进入死循环。但是主 cpu 会在 `bootstrap2` 里调用`psci_smc_call` 重启副 cpu。实际调用的是 `smc 0`。smc 的异常处理配置是 firmware 设置的。在 Zircon 的代码中没有对 `vbar_el3` 的设置。一个例子: 在 u-boot, `armv8/psci.S` 里有对`vbar_el3` 的设置。但是很有可能应该是 arm trusted firmware 设置的这个。
3. 最后进入`arm64_secondary_entry()`

7 lk_main

7.1 thread_init_early

`thread_construct_first(t, "bootstrap")`, 在主 cpu 上创建一个反映当前运行状态的`thread_t`.

1. 拿到的 cpu num 是 0
2. `thread_t` 清零
3. `THREAD_MAGIC` 暂时不知道什么用
4. 设置 thread 名字
5. `retcode_wait_queue` 的前驱后继都指向自己
6. 从`tpidr_el1` 中拿到`boot_cpu_fake_thread_pointer_location` 的地址
7. `__has_feature(safe_stack)` 是 clang 的特性, 对 gcc 来说, `unsafe_sp` 就是 0 (在 `start.S` 里安排的)
8. 把 x18 保存在`current_percpu_ptr` 里
9. 让`tpidr_el1` 指向这个真正的`thread_t`, 也就是 percpu 里的 `idle_thread`
10. 把这个 thread 加到全局的 thread list 上

`sched_init_early()` 初始化各个 cpu 的 run queue 为空。
调用 C++ 的全局变量的构造函数`__init_array_start`。
`lk_init_level()`
`required_flag = LK_INIT_FLAG_PRIMARY_CPU`
`start_level = LK_INIT_LEVEL_EARLIEST`
`stop_level = LK_INIT_LEVEL_ARCH_EARLY - 1`
这次没有匹配的 init 函数可以被调用。(也许我漏掉了?)
`arch_curr_cpu_num_slow()` 返回 0, 因为这时`arm64_cpu_map` 还没有初始化。

`arch.cpp: arm64_cpu_early_init()`
把 x18 指向当前 cpu 的 percpu.
让`VBAR_EL1` 指向`exceptions.S` 中的`arm64_el1_exception_base`.
把一些 cpu feature 读到`arm64_features` 里
打开 DAIF 遮掩位。
`main.cpp: platform_early_init()`
`bootdata_paddr` 存放的是 `initrd` 的加载地址 `0x48000000`。`boot-shim` 会把从 device tree 中读出来的 `initrd` 的加载地址写入`bootdata_paddr`。此外, `qemu` 会把内核加载到 `0x40080000` 处。内核的实际物理地址就是 `0x40090000`。这些细节需要从 `qemu` 的源码中获得。
`boot_reserve_add_range(get_kernel_base_phys(), get_kernel_size())`
把 kernel 占用的物理地址区域保留起来, 也就是 `0x40090000` 开始的一块区域。

接下来会处理`append_board_bootdata()` 里加入的一些 bootdata 的 section.

然后到`pdev_init()` 里调用各个周边设备驱动的初始化代码。

把 ramdisk 所在的 0x48000000 开始的一块内存保留起来。

`kernel.memory-limit-mb` 这个命令行参数应该还是没有设置的。

在每个物理内存 mem arena 的尾部放置的是一堆描述 page 的`vm_page_t`。

把之前标记为保留的内存区域在 arena 中保留出来, 标记成 WIRED。

`platform_early_init()` 结束。

终于来到了"welcome to Zircon"。

接下来调用`crypto::GlobalPRNG::EarlyBootSeed()`, 初始化一些随机数生成器相关的东西。

进入`vm_init_preheap()`, 初始化一个唯一的内核虚拟地址空间。

把 boot time allocator 用掉的内存 (主要是分配给 translation table) 在物理内存管理 pmm 里标记为已用。

随机保留分配一些页面。可能是为了增加内存随机性。

用 dc zva 把缓存置零。

进入`heap_init()`, 把内核要使用的 heap page 申请出来。

进入`vm_init()`。为内核占用的虚拟地址区域创建 VmObject. VmMapping 好像只是记录一个 vmo 有哪些 flags. 在创建 VmMapping 之后, Activate() 的时候才把它插到 parent, 也就是 RootVmar 的 ChildList 里面。

`platform_init_postvm()` 把外围设备的虚拟地址保留起来。

`kernel_init()`, 初始化多处理器的一些状态。初始化每个处理器的 timer. 之后, 内核就可以创建线程了。缺省的线程 stack 大小是 8k.

线程的切换通过改变 x30 完成。x30 存储的是`arm64_context_switch()` 调用的下一条指令的地址。

`bootstrap2`.

初始化 deferred procedure call (dpc) 线程。

初始化 Handle 和 Arena. 这个过程没有 page committed. 内存页真正分配出去是在做 `Arena::Pool::Pop()` 的时候。在 `VmMappingCoalescer::Flush()` 里会建立物理页表的映射。在做 VmMapping 的时候, 如果不指定起始地址, 则会去找一个 gap 地址。

PortDispatcher 暂时不知道干啥的。

进入`arch_init()`

给每个 cpu 创建一个 idle thread. 释放副 cpu 的 spin lock.

进入`platform_init()`. 给每个副 cpu 创建 kernel stack, 是一个 VmMapping.

`pdev_run_hooks()`

moving to last init leve.

user level init: ktrace, console, userboot.

ktrace: 给它分配一块内存, 缺省为 32MB

kernel console 通过 uart 读命令。

8 System Calls

`syscall-vdso.wrappers.inc` 里定义了 `zx_*` 到 `_zx_*` 的 weak alias.
`syscall-entry.h` 里定义了 `SYSCALL_*` 入口符号。
`syscalls-arm64.S` 里定义了真正的入口。

9 userboot

`attempt_userboot()`

首先拿到 ramdisk 的地址 0x48000000。

创建一个 vmo 作为 userboot 的 user space stack.

调用 `MapRange` 的地方

1. `kstack`
2. `Arena.Pool.Pop`
3. `syscall`
4. `MapObjectInternal()`, 如果要求的话。

`stack_vmo` 还没有 commit

创建一个 vmo 放 ramdisk

在堆上构造 bootstrap message.

`libuserboot.so` 是通过 `userboot-image.S` 嵌入到内核映像里的。

把 `stack_vmo` 映射好, 但是没有 commit page. 这个是 userboot 用户线程的 user stack.

`make_bootstrap_channel()`。一个 channel 由 2 个 peer 组成, 每方都有自己的 message list.

`system/core/userboot/start.c:_start()` 进入用户模式。

附录:

linking userlib build-arm64/system/ulib/zircon/libzircon.so

```
./prebuilt/downloads/gcc/bin/aarch64-elf-ld.gold -nostdlib --build-id
-z noexecstack -z max-page-size=4096 -z combreloc -z relro -z now -z text
--hash-style=gnu --eh-frame-hdr -z defs -T scripts/rodso.ld -shared
-soname libzircon.so
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_cache_flush.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_channel_call.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_deadline_after.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_status_get_string.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_system_get_dcache_line_size.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_system_get_features.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_system_get_num_cpus.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_system_get_physmem.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_system_get_version.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_ticks_get.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_ticks_per_second.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/syscall-wrappers.cpp.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/data.S.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_futex_wake_handle_close_thread_exit.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/zx_vmar_unmap_handle_close_thread_exit.o
./build-arm64/system/ulib/zircon/system/ulib/zircon/syscalls-arm64.S.o
```

```

scripts/dso_handle.ld
/Users/xzx/zircon/prebuilt/downloads/gcc/bin/./lib/gcc/aarch64-elf/6.3.0/libgcc.a
-o build-arm64/system/ulib/zircon/libzircon.so

userboot-code.h:

#define USERBOOT_FILENAME "./build-arm64/system/core/userboot/libuserboot.so"
#define USERBOOT_CODE_START 0x3000
#define USERBOOT_CODE_END (((USERBOOT_CODE_START + 0xa000 + (1 << PAGE_SIZE_SHIFT) - 1) >> 1) & 0xffffffff)
#define USERBOOT_ENTRY 0x00000000000003b50
#define USERBOOT_ENTRY_SIZE 0x00000000000008e4

generating build-arm64/system/core/userboot/vdso-syms.h

scripts/shlib-symbols -a './prebuilt/downloads/gcc/bin/aarch64-elf-nm'
    build-arm64/system/ulib/zircon/libzircon.so >
    build-arm64/system/core/userboot/vdso-syms.h

generating build-arm64/system/core/userboot/vdso-syms.ld

./prebuilt/downloads/gcc/bin/aarch64-elf-gcc -E -P
    -include system/core/userboot/vdso-syms.ld.h
    build-arm64/system/core/userboot/vdso-syms.h >
    build-arm64/system/core/userboot/vdso-syms.ld

linking userlib build-arm64/system/core/userboot/libuserboot.so

./prebuilt/downloads/gcc/bin/aarch64-elf-ld.gold -nostdlib --build-id
    -z noexecstack -z max-page-size=4096 -z combrelloc -z relro -z now -z text
    --hash-style=gnu --eh-frame-hdr -z defs -T scripts/rodso.ld -e _start
    -shared -soname libuserboot.so
    ./build-arm64/system/core/userboot/system/core/userboot/bootdata.c.o
    ./build-arm64/system/core/userboot/system/core/userboot/bootfs.c.o
    ./build-arm64/system/core/userboot/system/core/userboot/userboot-elf.c.o
    ./build-arm64/system/core/userboot/system/core/userboot/option.c.o
    ./build-arm64/system/core/userboot/system/core/userboot/start.c.o
    ./build-arm64/system/core/userboot/system/core/userboot/loader-service.c.o
    ./build-arm64/system/core/userboot/system/core/userboot/util.c.o
    ./build-arm64/system/core/userboot/third_party/ulib/musl/src/string/memcmp.c.o
    ./build-arm64/system/core/userboot/third_party/ulib/musl/src/string/memcpy.c.o
    ./build-arm64/system/core/userboot/third_party/ulib/musl/src/string/memset.c.o
    ./build-arm64/system/core/userboot/third_party/ulib/musl/src/string/strlen.c.o
    ./build-arm64/system/core/userboot/third_party/ulib/musl/src/string/strncpy.c.o
    ./build-arm64/system/core/userboot/third_party/ulib/musl/src/string/memmove.c.o

```

```

./build-arm64/system/core/userboot/system/ulib/elfload/elf-load.c.o
./build-arm64/system/core/userboot/system/ulib/bootdata/decompress.c.o
./build-arm64/system/core/userboot/third_party/ulib/lz4/lz4.c.o
./build-arm64/system/core/userboot/system/ulib/ldmsg/ldmsg.c.o
./build-arm64/system/core/userboot/vdso-syms.ld scripts/dso_handle.ld
./build-arm64/system/ulib/runtime/libruntime.a
/Users/xzx/zircon/prebuilt/downloads/gcc/bin/./lib/gcc/aarch64-elf/6.3.0/libgcc.a
-o build-arm64/system/core/userboot/libuserboot.so

generating build-arm64/system/core/userboot/libuserboot.so.strip

./prebuilt/downloads/gcc/bin/aarch64-elf-objcopy --strip-all
  build-arm64/system/core/userboot/libuserboot.so
  build-arm64/system/core/userboot/libuserboot.so.strip

generating build-arm64/kernel/lib/userboot/userboot-code.h

scripts/gen-rodso-code.sh './prebuilt/downloads/gcc/bin/aarch64-elf-nm'
  './prebuilt/downloads/gcc/bin/aarch64-elf-readelf'
  build-arm64/kernel/lib/userboot/userboot-code.h.new USERBOOT
  ./build-arm64/system/core/userboot/libuserboot.so

```

10 设备模型和驱动

有一个中心服务 devmgr coordinator. 它会扫描驱动, 为每个驱动 launch 一个 devhost 进程。launch 完了之后会发送 create device, bind driver 这 2 个 rpc 给 devhost. devhost 会处理这 2 个 rpc. 对于 root device, 不会处理, 因为 libname=""

文档推荐在 bind driver 里调用 add device. 驱动本身是在 devhost 里运行的。devhost 会 rpc 调用 dev coordinator 的 add device 方法。这个方法主要是创建 dc 里对设备的代表, 发布到 devfs 里。

在启动进程的时候, 传递的第一个参数是一个 channel handle. libc start main 在拿到这个 channel handle 之后, 读取它。channel 里的 message 是 MessagePack 里面的 data 是 `zx_proc_args_t`。这个东西是由另一头用 `channel_write()` 写进来的。userboot 里面是手工组装这个消息。如果是普通应用层应该会把这个封装掉。

`zx_proc_args_t` 是传递进程参数的固定的消息格式。作为 data 写入 MessagePack.

到达 `devmgr:main()`

`devfs_init():`

`iostate_create(&root_devnode)`

`devmgr_vfs_init():`

`fshost_start():`

`loader_service:`

`svc async`的port等待的是另一头的handle

`async`里的线程等待的是port

`dl_set_loader_service()` 把新创建的loader service设置为缺省的, 关闭老的

`devfs_root_clone():` 创建一个channel, 返回一头, 另一头封装在一个消息里发送给devfs_root

`devfs_root`会在`coordinator()`中被轮询

问题在于devfs_root的另一头只是和dc_port wait了一次。在处理完成之后会立刻又wait一次!

另一头会传递给devfs_open()

`fs_clone("svc"):`

`fdio_open_at(fs=appmgr_req_cli, path, flag, h1)`

`this function creates a channel and send one handle to appmgr, and returns the other`

设置了一堆 handle 之后, 启动 fshost use launchpad, to be looked ...

发送了几个设备的 channel handle 给fs_root

11 线程

创建 bootstrap2 线程时, `thread_t` 是在 `malloc` 里创建的。stack 也是在 `malloc`。如果是用户进程通过系统调用创建线程, 则 stack 是通过 kernel aspece 分配出来的 vm object。user sp 是通过 vm object mapping 得到的。

放到 stack frame 里的入口是 `initial_thread_func`。这个函数会调用 `thread_t` 的 entry。 `thread_t` 的 entry 是 `ThreadDispatcher::StartRoutine`。 `StartRoutine` 里会调用 `user_entry_`, 这个是在 `ThreadDispatcher::Start()` 里设置的。

thread 的启动: `ThreadDispatcher::Start()` 里设置好 `user_entry_`, 然后触发线程调度, 切换到线程的 stack 上, 然后进入 stack frame 里指定的入口 `initial_thread_func`。

thread 的退出: 在 `thread_create_etc()` 里, 线程入口被设置为 `initial_thread_func`。在这个函数里, 如果是用户线程, 则不会回来了。用户线程自己调用 `process_exit` 退出。如果是 kernel 线程, 会回到这里调用 `thread_exit()`。在这个函数里, 线程把自己从调度队列里拿下来, 然后触发调度函数, 进入别的线程。

中断: 中断会保存 `spsr_el1`, 然后切换到 `sp_el1`, 然后进入中断向量

12 虚拟内存管理

zircon 的每个进程的页表对应的 page 是初始化 process handle 时创建的。页表地址会记录在 aspace 里面。

当切换进程后，如果通过 channel 传递过来的 vmr 里面已经映射的 vmo 的地址没有对应的物理页面，会触发 Page fault。pagefault 会优先从 vmo 里拿它已有的页面，然后到页表里建立映射关系。

当 load devmgr 的时候，如果有动态链接器，则 entry 会设置为 interp 的入口。

gold 对于 aarch64 默认 entry symbol 是 **_start**

vmobject 可以从一个地址和 size 创建出来。也可以 clone 出来。

ld.so 会把 application 的 phdr 全部映射一遍，其中有一个 segment 包含 ehdr, 从中得到 entry addr (需要用映射到的地址进行修正)