Zircon 笔记

许中兴

May 3, 2018

Contents

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 '
```

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

一些数据结构的定义。

```
// BootData header, describing the type and size of data
// used to initialize the system. All fields are little-endian.
//
// BootData headers in a stream must be 8-byte-aligned.
//
// The length field specifies the actual payload length
// and does not include the size of padding.
typedef struct {
    // Boot data type
    uint32_t type;

    // Size of the payload following this header
    uint32_t length;

    // type-specific extra data
    // For CONTAINER this is MAGIC.
    // For BOOTFS this is the decompressed size.
    uint32_t extra;

    // Flags for the boot data. See flag descriptions for each type.
    uint32_t flags;

    // For future expansion.  Set to 0.
    uint32_t reserved0;
    uint32_t reserved1;

    // Must be BOOTITEM_MAGIC
    uint32_t magic;

    // Must be the CRC32 of payload if FLAG_CRC32 is set,
    // otherwise must be BOOTITEM_NO_CRC32
    uint32_t crc32;
} bootdata_t;

struct fsentry {
    fsentry_t* next;

    char* name;
    size_t namelen;
    uint32_t offset;
    uint32_t length;

    char* srcpath;
};

struct item {
    item_type_t type;
    item_t* next;

    fsentry_t* first;
    fsentry_t* last;

    // size of header and total output size
    // used by bootfs items
    size_t hdrsize;
    size_t outsize;

    // Used only by ITEM_PLATFORM_ID items.
    bootdata_platform_id_t platform_id;
};
```

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.ld \
    build-arm64/kernel-vars.ld \
    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
```

我们先看kernel.ld。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。

```
/*
 * Symbols used in the kernel proper are defined with PROVIDE_HIDDEN:
 * HIDDEN because everything in the kernel is STV_HIDDEN to make it
 * clear that direct PC-relative references should be generated in PIC;
 * PROVIDE because their only purpose is to satisfy kernel references.
 */
```

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

    /*
     * This just leaves space in the memory image for the boot headers.
     * The actual boot header will be constructed in image.S, which see.
     */
    .text.boot0 : {
        /*
         * Put some data in, or else the linker makes it a SHT_NOBITS
         * section and that makes objcopy -O binary skip it in the image.
         */
        LONG(0xdeadbeef);
        . += BOOT_HEADER_SIZE - 4;
    } :code

    . = ALIGN(8);
    .buildsig : {
        PROVIDE_HIDDEN(buildsig = .);
        BYTE(0x42); BYTE(0x53); BYTE(0x49); BYTE(0x47); /* BSIG */
        BYTE(0x53); BYTE(0x54); BYTE(0x52); BYTE(0x54); /* STRT */
        /*
         * The self-pointer gives a local basis to compute the relative
         * positions of the lk_version_t and .note.gnu.build-id pointers
         * without already knowing the kernel's virtual address base.
         */
        QUAD(buildsig);
        QUAD(version);
        QUAD(__build_id_note_start);
        BYTE(0x42); BYTE(0x53); BYTE(0x49); BYTE(0x47); /* BSIG */
        BYTE(0x45); BYTE(0x4e); BYTE(0x44); BYTE(0x53); /* ENDS */
     }

    /*
     * This is separate from .text just so gen-kaslr-relocs.sh can match
     * it.  The relocation processor skips this section because this code
     * all runs before the boot-time fixups are applied and has its own
     * special relationship with the memory layouts.
     */
    .text.boot : {
        *(.text.boot)
    }

    /*
```

```
 * This is separate from .text just so gen-kaslr-relocs.sh can match
 * it.  This section contains movabs instructions that get 64-bit
 * address fixups in place.  This is safe because this code is never
 * used until long after fixups have been applied.  In general, the
 * script will refuse to handle fixups in text (i.e. code) sections.
 */
.text.bootstrap16 : {
    *(.text.bootstrap16)
}

.text : {
    *(.text* .sram.text)
    *(.gnu.linkonce.t.*)
}

PROVIDE_HIDDEN(__code_end = .);

. = ALIGN(4096);
PROVIDE_HIDDEN(__rodata_start = .);

/*
 * These are page-aligned, so place them first.
 */
.rodata.rodso_image : {
    *(.rodata.rodso_image.*)
}

.note.gnu.build-id : {
    PROVIDE_HIDDEN(__build_id_note_start = .);
    *(.note.gnu.build-id)
    PROVIDE_HIDDEN(__build_id_note_end = .);
} :rodata :note

/*
 * The named sections starting with kcountdesc are sorted
 * by name so that tools can provide binary search lookup
 * for k_counter_desc::name[] variables.
 */
.kcounter.desc : ALIGN(8) {
    PROVIDE_HIDDEN(kcountdesc_begin = .);
    KEEP(*(SORT_BY_NAME(kcountdesc.*)))
    PROVIDE_HIDDEN(kcountdesc_end = .);
} :rodata

.rodata : {
    *(.rodata* .gnu.linkonce.r.*)
```

```
} :rodata

/*
 * When compiling PIC, the compiler puts things into sections it
 * thinks need to be writable until after dynamic relocation.  In
 * the kernel, these things all go into the read-only segment.  But
 * to the linker, they are writable and so the default "orphans"
 * placement would put them after .data instead of here.  That's bad
 * both because we want these things in the read-only segment (the
 * kernel's self-relocation applies before the read-only-ness starts
 * being enforced anyway), and because the orphans would wind up
 * being after the __data_end symbol (see below).
 *
 * Therefore, we have to list all the special-case sections created
 * by __SECTION("foo") uses in the kernel that are RELRO candidates,
 * i.e. things that have address constants in their initializers.
 * All such uses in the source use sections named ".data.rel.ro.foo"
 * instead of just "foo" specifically to ensure we write them here.
 * This avoids the magic linker behavior for an "orphan" section
 * called "foo" of synthesizing "__start_foo" and "__stop_foo"
 * symbols when the section name has no . characters in it, and so
 * makes sure we'll get undefined symbol references if we omit such
 * a section here.  The magic linker behavior is nice, but it only
 * goes for orphans, and we can't abide the default placement of
 * orphans that should be RELRO.
 */
.data.rel.ro : ALIGN(8) {
    PROVIDE_HIDDEN(__start_commands = .);
    KEEP(*(.data.rel.ro.commands))
    PROVIDE_HIDDEN(__stop_commands = .);

    PROVIDE_HIDDEN(__start_ktrace_probe = .);
    KEEP(*(.data.rel.ro.ktrace_probe))
    PROVIDE_HIDDEN(__stop_ktrace_probe = .);

    PROVIDE_HIDDEN(__start_lk_init = .);
    KEEP(*(.data.rel.ro.lk_init))
    PROVIDE_HIDDEN(__stop_lk_init = .);

    PROVIDE_HIDDEN(__start_lk_pdev_init = .);
    KEEP(*(.data.rel.ro.lk_pdev_init))
    PROVIDE_HIDDEN(__stop_lk_pdev_init = .);

    PROVIDE_HIDDEN(__start_unittest_testcases = .);
    KEEP(*(.data.rel.ro.unittest_testcases))
    PROVIDE_HIDDEN(__stop_unittest_testcases = .);
    *(.data.rel.ro* .gnu.linkonce.d.rel.ro.*)
}

.init_array : ALIGN(8) {
    PROVIDE_HIDDEN(__init_array_start = .);
    KEEP(*(.init_array .ctors))
    PROVIDE_HIDDEN(__init_array_end = .);
}

/*
 * Any read-only data "orphan" sections will be inserted here.
 * Ideally we'd put those into the .rodata output section, but
 * there isn't a way to do that that guarantees all same-named
 * input sections collect together as a contiguous unit, which
 * is what we need them for.  Linkers differ in how they'll
 * place another dummy section here relative to the orphans, so
 * there's no good way to define __rodata_end to be exactly the
 * end of all the orphans sections.  But the only use we have
 * for __rodata_end is to round it up to page size anyway, so
 * just define it inside the .data section below, which is
 * exactly the end of the orphans rounded up to the next page.
 */

.data : ALIGN(4096) {
    PROVIDE_HIDDEN(__rodata_end = .);
    PROVIDE_HIDDEN(__data_start = .);
    *(.data .data.* .gnu.linkonce.d.*)
    /*
     * Make sure the total file size is aligned to 8 bytes so the image
     * can go into a BOOTDATA container, which requires 8-byte alignment.
     */
    . = ALIGN(8);
    PROVIDE_HIDDEN(__data_end = .);
} :data

/*
 * Any writable orphan sections would be inserted here.
 * But there's no way to put the __data_end symbol after
 * them, so we cannot allow any such cases.  There is no
 * good way to assert that, though.
 */

.bss : ALIGN(4096) {
    PROVIDE_HIDDEN(__bss_start = .);
```

```
        /*
         * See kernel/include/lib/counters.h; the KCOUNTER macro defines a
         * kcounter.NAME array in the .bss.kcounter.NAME section that
         * allocates SMP_MAX_CPUS counter slots.  Here we collect all those
         * together to make up the kcounters_arena contiguous array.  There
         * is no particular reason to sort these, but doing so makes them
         * line up in parallel with the sorted .kcounter.desc section.
         */
        . = ALIGN(8);
        PROVIDE_HIDDEN(kcounters_arena = .);
        KEEP(*(SORT_BY_NAME(.bss.kcounter.*)))

        /*
         * Sanity check that the aggregate size of kcounters_arena
         * SMP_MAX_CPUS slots for each counter.  The k_counter_desc structs
         * in .kcounter.desc are 8 bytes each, which matches the size of a
         * single counter.  (It's only for this sanity check that we need
         * to care how big k_counter_desc is.)
         */
        ASSERT(. - kcounters_arena == SIZEOF(.kcounter.desc) * SMP_MAX_CPUS,
               "kcounters_arena size mismatch");

        *(.bss*)
        *(.gnu.linkonce.b.*)
        *(COMMON)
    }

    /*
     * Any SHT_NOBITS (.bss-like) sections would be inserted here.
     */

     . = ALIGN(4096);
    PROVIDE_HIDDEN(_end = .);
}

PHDRS
{
    code PT_LOAD FLAGS(5); /* PF_R|PF_X */
    rodata PT_LOAD FLAGS(4); /* PF_R */
    data PT_LOAD FLAGS(6); /* PF_R|PF_W */
    note PT_NOTE FLAGS(4); /* PF_R */
}

/*
 * This is not actually used since the entry point is set in image.ld,
 * but it prevents the linker from warning about using a default address
 * and it keeps --gc-sections from removing .text.boot.
 */
ENTRY(IMAGE_ELF_ENTRY)

/*
 * These special symbols below are made public so they are visible via
 * --just-symbols to the link of image.S.
 */

IMAGE_LOAD_START = __code_start;
IMAGE_LOAD_END = __data_end;
IMAGE_MEMORY_END = _end;
```

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.

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。

3 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.html

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。如果不是 0，则不保存 cpu 信息。

2. 把 x0 的内容，也就是内核 header 的地址保存在bootdata_paddr 里。`str     x0, [tmp, #:lo12: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，所以需要用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 https://stackoverflow.com/questions/38570495/aarch64-relocation-prefixes

3. 把内核入口地址_start 存入kernel_entry_paddr

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。

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. 把tt_trampoline 页表清零

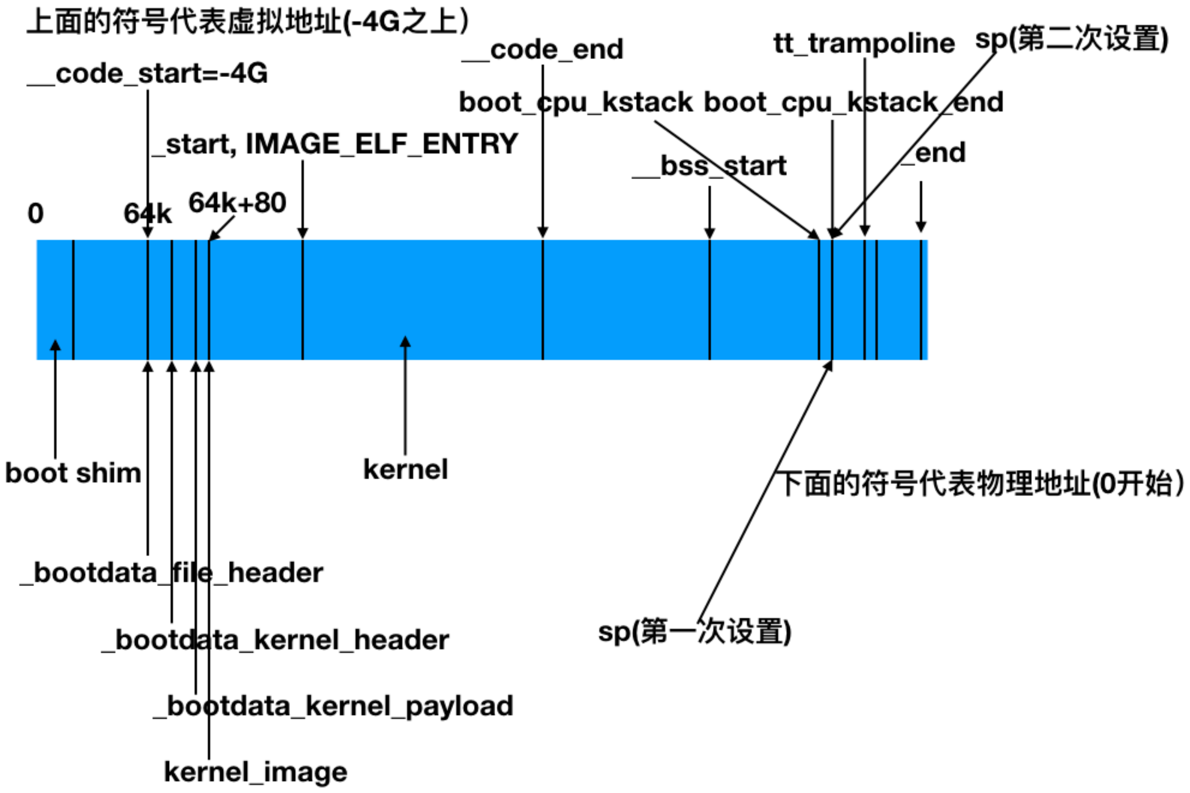
18. MMU_IDENT_SIZE_SHIFT = 42, MMU_IDENT_PAGE_SIZE_SHIFT = 16 MMU_IDENT_TOP_SHIFT = 32, MMU_PAGE_TABLE_ENTRIES_IDENT_SHIFT = 10 MMU_PAGE_TABLE_ENTRIES_IDENT = 1 << 10

19. 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

20. 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。

21. 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。


上面的符号代表虚拟地址(-4G之上）
__code_start=-4G
_start, IMAGE_ELF_ENTRY
__code_end
boot_cpu_kstack
boot_cpu_kstack_end
tt_trampoline
sp(第二次设置)
__bss_start
_end
0
64k
64k+80
boot shim
kernel
下面的符号代表物理地址(0开始）
_bootdata_file_header
_bootdata_kernel_header
_bootdata_kernel_payload
kernel_image
sp(第一次设置)