

# Zircon 笔记

许中兴

May 4, 2018

## Contents

<b>1</b>	<b>Bootloader</b>	<b>2</b>
1.1	mkbootfs 一些细节 . . . . .	2
<b>2</b>	<b>ARM 架构</b>	<b>10</b>
2.1	关于 MMU . . . . .	10
<b>3</b>	<b>start.S</b>	<b>11</b>
3.1	副 cpu . . . . .	13

# 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

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 \
-02 -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

```

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。

### 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](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`.

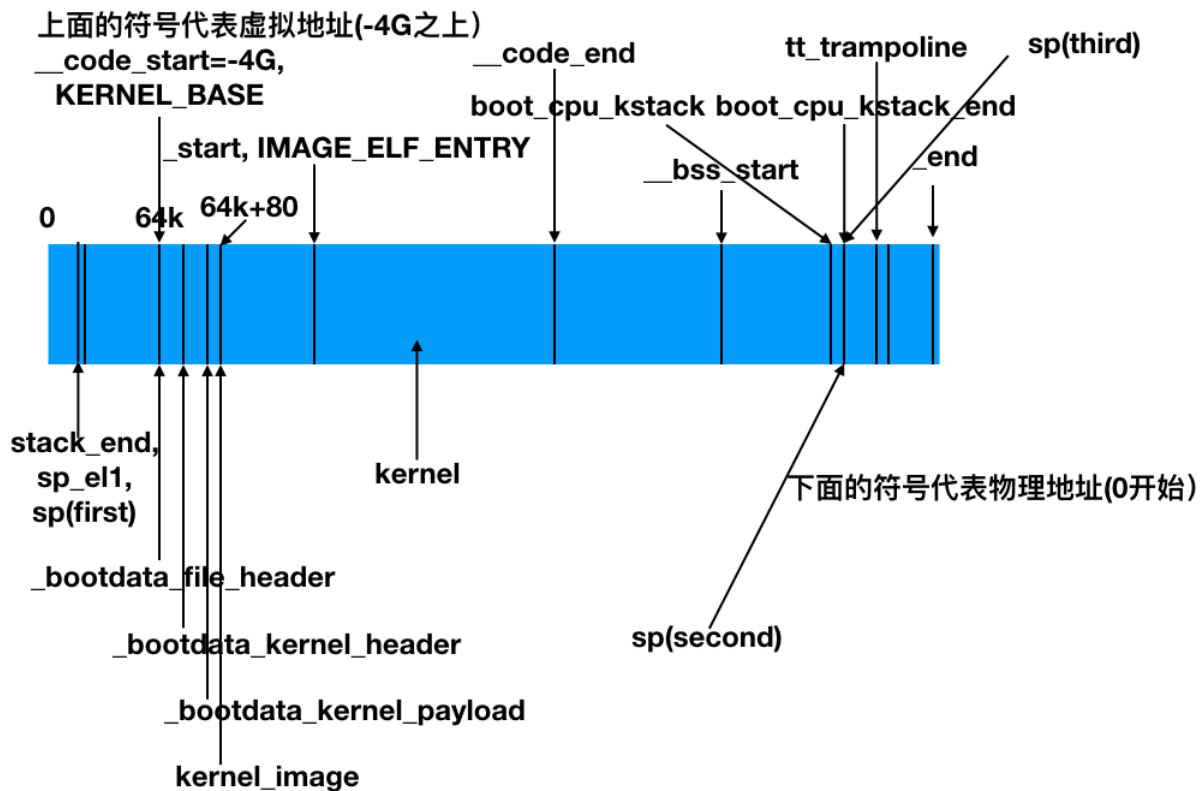
Unless `.cfi_startproc` is used along with parameter `simple` it also emits some architecture dependent initial CFI instructions.

1. 获取MPIDR\_EL1 寄存器的 Aff0 和 Aff1 16bit 信息。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, #:lo12: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`
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。



### 3.1 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 会调用 `psci_smc_call` 重启副 cpu。
3. 最后进入 `arm64_secondary_entry()`