CH1~CH2

CH1

os/src/linker.ld
链接脚本，调整链接器行为，符合预期内存布局
注意，在新的工具链中，这个东西在.cargo/config.toml内配置

裸机启动过程

用 QEMU 软件 `qemu-system-riscv64` 来模拟 RISC-V 64 计算机。加载内核程序的命令如下：

```
qemu-system-riscv64 \
            -machine virt \
            -nographic \
            -bios $(BOOTLOADER) \
            -device loader,file=$(KERNEL_BIN),addr=$(KERNEL_ENTRY_PA)
```

- `-bios $(BOOTLOADER)` 意味着硬件加载了一个 BootLoader 程序，即 RustSBI
- `-device loader,file=$(KERNEL_BIN),addr=$(KERNEL_ENTRY_PA)` 表示硬件内存中的特定位置 `$(KERNEL_ENTRY_PA)` 放置了操作系统的二进制代码 `$(KERNEL_BIN)` 。`$(KERNEL_ENTRY_PA)` 的值是 `0x80200000` 。

当我们执行包含上述启动参数的 qemu-system-riscv64 软件，就意味给这台虚拟的 RISC-V64 计算机加电了。 此时，CPU 的其它通用寄存器清零，而 PC 会指向 `0x1000` 的位置，这里有固化在硬件中的一小段引导代码， 它会很快跳转到 `0x80000000` 的 RustSBI 处。 RustSBI完成硬件初始化后，会跳转到 `$(KERNEL_BIN)` 所在内存位置 `0x80200000` 处， 执行操作系统的第一条指令。

`0x80000000`：在 QEMU 模拟的 RISC-V 中，DRAM 内存的物理地址是从 0x80000000 开始，有 128MB 大小

```
    .section .text.entry
    .globl _start
_start:
    la sp, boot_stack_top
    call rust_main

    .section .bss.stack
    .globl boot_stack_lower_bound
```

```
boot_stack_lower_bound:
    .space 4096 * 16#64K的栈
    .globl boot_stack_top
boot_stack_top:
```

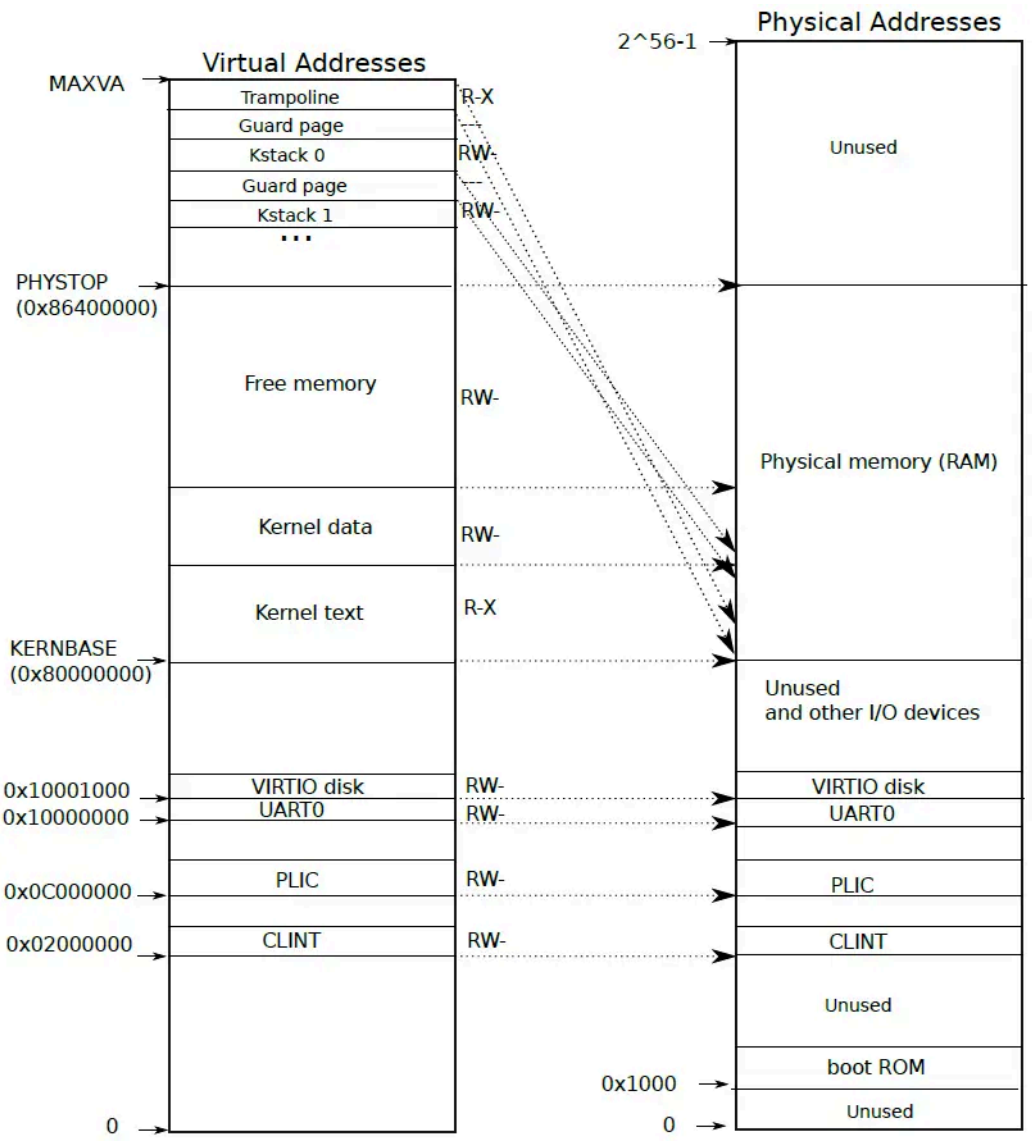
在第 8 行，我们预留了一块大小为 4096 * 16 字节，也就是 64KiB 的空间， 用作操作系统的栈空间。 栈顶地址被全局符号 `boot_stack_top` 标识，栈底则被全局符号 `boot_stack` 标识。 同时，这块栈空间被命名为 `.bss.stack` ，链接脚本里有它的位置。

`_start` 作为操作系统的入口地址，将依据链接脚本被放在 `BASE_ADDRESS` 处。 `la sp, boot_stack_top` 作为 OS 的第一条指令， 将 sp 设置为栈空间的栈顶。 简单起见，我们目前不考虑 sp 越过栈底 `boot_stack` ，也就是栈溢出的情形。 第二条指令则是函数调用 `rust_main` ，这里的 `rust_main` 是我们稍后自己编写的应用入口。

随后，在main.rs嵌入这些汇编代码

```
OUTPUT_ARCH(riscv)
ENTRY(_start)
BASE_ADDRESS = 0x80200000;

SECTIONS
{
    . = BASE_ADDRESS;
    skernel = .;

    stext = .;
    .text : {
        *(.text.entry)
        *(.text .text.*)
    }

    . = ALIGN(4K);
    etext = .;
    srodata = .;
    .rodata : {
        *(.rodata .rodata.*)
        *(.srodata .srodata.*)
    }

    . = ALIGN(4K);
    erodata = .;
    sdata = .;
    .data : {
```

```
        *(.data .data.*)
        *(.sdata .sdata.*)
    }

    . = ALIGN(4K);
    edata = .;
    .bss : {
        *(.bss.stack)
        sbss = .;
        *(.bss .bss.*)
        *(.sbss .sbss.*)
    }

    . = ALIGN(4K);
    ebss = .;
    ekernel = .;

    /DISCARD/ : {
        *(.eh_frame)
    }
}
```



奇怪的内存地址？看上面这个图

CH2

main.rs 中 `global_asm!(include_str!("link_app.S"));`，引入了一段汇编代码，在编译时生成

特权模式

CSR 名	该 CSR 与 Trap 相关的功能
sstatus	`SPP` 等字段给出 Trap 发生之前 CPU 处在哪个特权级（S/U）等信息

CSR 名	该 CSR 与 Trap 相关的功能
sepc	当 Trap 是一个异常的时候，记录 Trap 发生之前执行的最后一条指令的地址
scause	描述 Trap 的原因
stval	给出 Trap 附加信息
stvec	控制 Trap 处理代码的入口地址

CH2中设计特权级别切换的操作

- 启动应用程序时，需要初始化应用程序的用户态上下文，并能切换到用户态执行应用程序；
- 应用程序发起系统调用后，需要切换到批处理操作系统中进行处理；
- 应用程序执行出错时，批处理操作系统要杀死该应用并加载运行下一个应用；
- 应用程序执行结束时，批处理操作系统要加载运行下一个应用。

当 CPU 执行完一条指令并准备从用户特权级 陷入（ `Trap` ）到 S 特权级的时候，硬件会自动完成如下这些事情：

- `sstatus` 的 `SPP` 字段会被修改为 CPU 当前的特权级（U/S）。
- `sepc` 会被修改为 Trap 处理完成后默认会执行的下一条指令的地址。
- `scause/stval` 分别会被修改成这次 Trap 的原因以及相关的附加信息。
- CPU 会跳转到 `stvec` 所设置的 Trap 处理入口地址，并将当前特权级设置为 S ，然后从 Trap 处理入口地址处开始执行。

… …
