

RustSBI学习

RustSBI学习

什么是SBI: RISC-V Supervisor Binary Interface

什么是RustSBI

什么是qemu

使用RustSBI-qemu - 编译、测试

使用RustSBI-k210 编译、测试

RustSBI-qemu 测试kernel

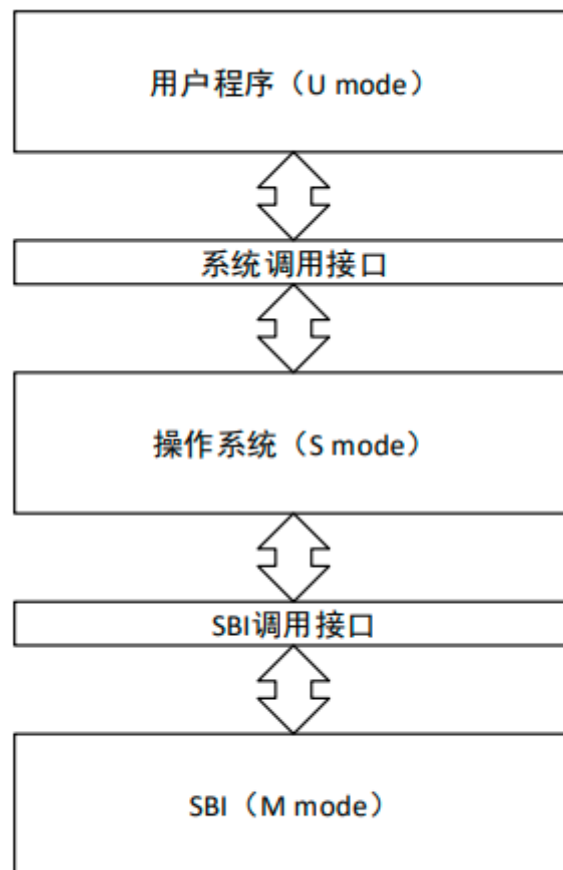
RustSBI作为Bootloader的使用

1. 以xv6-k210为例

2. 以OS-HIT为例

什么是SBI: RISC-V Supervisor Binary Interface

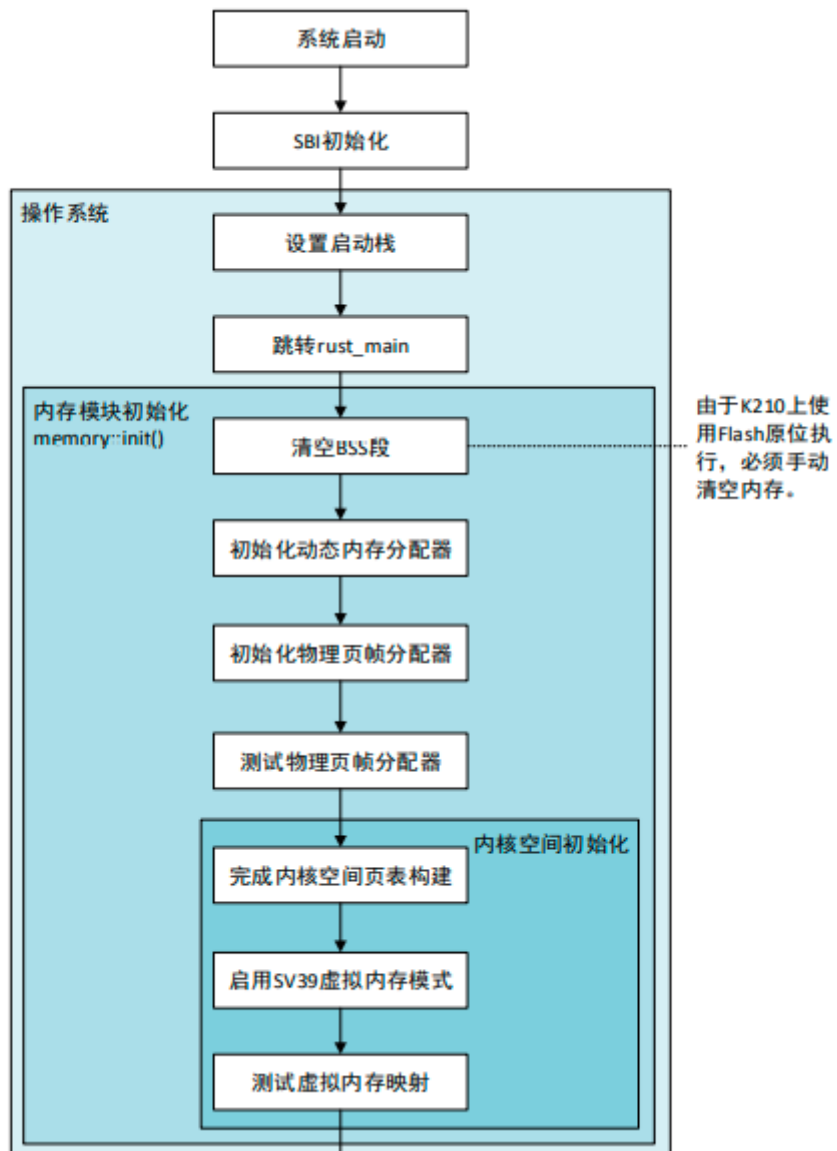
- SBI是操作系统的引导程序和运行时。机器上电时，SBI将配置环境，准备设备树，最终将引导启动操作系统。（copy from: <https://rcore-os.github.io/blog/2020/09/06/os-report-final-luojia65/>）
 - 在rustsbi的文档中，它被定位为：A SBI implementation will bootstrap your kernel, and provide an environment when your kernel is running.
- 我们可以将理解SBI为**两个作用**：
 - 作为bootloader引导操作系统内核的加载
 - 为操作系统提供操作硬件的接口
- 首先，在RISC-V指令集中，总共划分了三个模式：
 - User-Mode: 运行用户程序，通过系统调用接口与操作系统内核进行交互
 - Supervisor Mode: 内核运行的位置（在港中深的OS课上也叫做kernel mode）。
 - Machine Mode: SBI运行的位置，为操作系统提供一些底层基础支持。

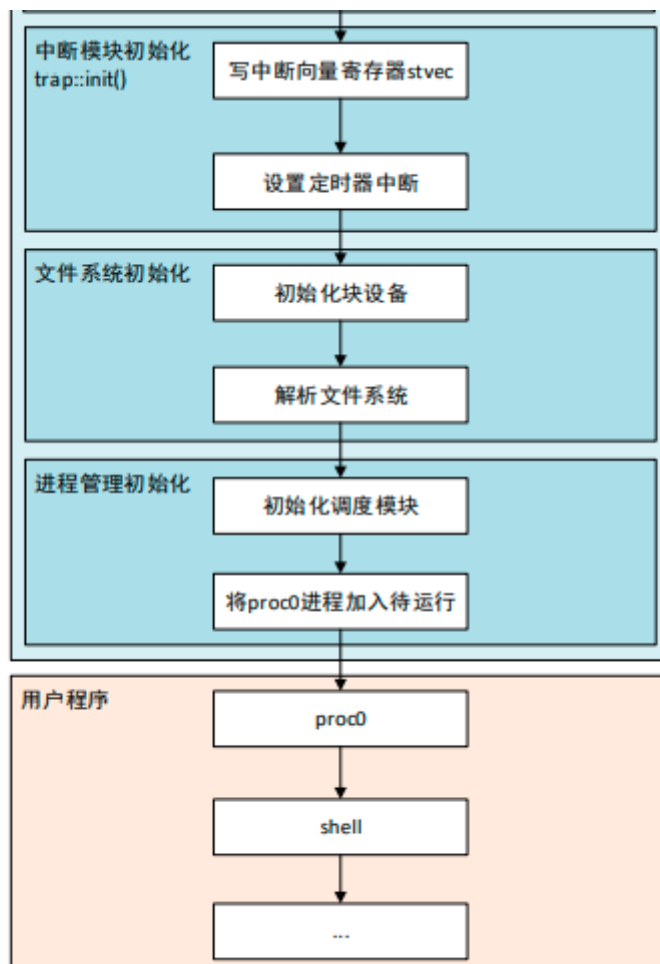


图片 1 本操作系统的基本层次

(图片来源于OSHIT设计报告: https://gitlab.eduxiji.net/willson0v0/oscomp_handin/-/blob/master/doc/OSHIT%E8%AE%BE%E8%AE%A1%E6%8A%A5%E5%91%8A.pdf)

- 通过上面的三个模式的划分，我们可以理解SBI为一种**运行的环境**。它介于操作系统和硬件之间，为操作系统提供更加底层的支持。因此学习SBI的接口将非常有助于设计一个能充分发挥硬件资源的操作系统。
 - 比如：操作系统需要访问硬件或者特殊的功能，这时候就需要通过ecall指令陷入M层的SBI运行时，由SBI完成这些功能再提供。
- 同时，当系统启动的时候，将优先完成SBI的初始化，再由SBI作为bootloader，将操作系统的内核引导进内存中并完成内核的初始化。我们可以从上一届作评OS-HIT的图片中直观地理解SBI在系统启动后扮演的角色：





图片 8 系统整体启动流程

什么是RustSBI

- RustSBI是SBI的一种实现，除此之外还有其他诸如OpenSBI。RustSBI的优势在于，其支持qemu和k210这两个平台，正好可以作为我们设计适用于k210的操作系统内核的强大武器。
- 源码：<https://github.com/rustsbi/rustsbi.git>

什么是qemu

- Qemu是一个虚拟机，它可以模拟很多的CPU架构。包括我们使用的RISC-V指令集是可以在上面模拟运行的。因此，我们可以以qemu为平台来设计我们的操作系统。
- Ubuntu安装qemu：<https://www.jianshu.com/p/f40eb6f26384>

使用RustSBI-qemu - 编译、测试

- 首先，我们获取在github上RustSBI-qemu的源码：

```
$ git clone https://github.com/rustsbi/rustsbi-qemu.git
```

- 按照 `Readme.md` 中写清的要求去安装一些编译需要的工具链

```
$ cargo install cargo-binutils
$ rustup component add llvm-tools-preview
```

(注意, 还需要提前配好rust环境, 包括cargo, rustc这些)

- 我们可以在 `/rustsbi-qemu/cargo/config.toml` 中找到作者写好的一些编译命令:

```
[alias]
xtask = "run --package xtask --"
make = "xtask make"
qemu = "xtask qemu"
asm = "xtask asm"
size = "xtask size"
debug = "xtask debug"
gdb = "xtask gdb"
```

- 这里, 我们要用的第一步主要是 `cargo qemu`
- 运行 `cargo qemu`
 - 中间遇到了一系列的报错, 其主要都是该安装的包没有安装好, 如果成功运行了, 会看到下面的信息:

```
ubuntu@VM-20-8-ubuntu:~/Desktop/rustsbi-qemu/rustsbi-qemu$ cargo qemu
Finished dev [unoptimized + debuginfo] target(s) in 0.13s
    Running `target/debug/xtask qemu`
xtask: mode: Debug
    Compiling bit_field v0.10.1
    Compiling spin v0.5.2
    Compiling spin v0.7.1
    Compiling lazy_static v1.4.0
    Compiling riscv v0.7.0
    Compiling riscv v0.7.0 (https://github.com/rust-embedded/riscv?
rev=dc0bc37e#dc0bc37e)
    Compiling buddy_system_allocator v0.8.0
    Compiling rustsbi v0.2.1
    Compiling rustsbi-qemu v0.1.0 (/home/ubuntu/Desktop/rustsbi-qemu/rustsbi-
qemu/rustsbi-qemu)
    Finished dev [unoptimized + debuginfo] target(s) in 3.61s
    Compiling bit_field v0.10.1
    Compiling spin v0.5.2
    Compiling spin v0.7.1
    Compiling lazy_static v1.4.0
    Compiling riscv v0.6.0
    Compiling buddy_system_allocator v0.8.0
    Compiling test-kernel v0.1.0 (/home/ubuntu/Desktop/rustsbi-qemu/rustsbi-
qemu/test-kernel)
    Finished dev [unoptimized + debuginfo] target(s) in 2.13s
[rustsbi] RustSBI version 0.2.1, adapting to RISC-V SBI v0.3

.----- .----- .----- .----- .-----
| _ \ | | | | / | | / | | _ \ | | |
| |_) | | | | | (----`---| |---` (----` |_) | |
| / | | | | \ \ | | \ \ | _ < | |
| |\ \---. | `---' |.----) | | |.----) | | |_) | |
|_| \_.---| \---/ |---/ | | |---/ |---/ | |

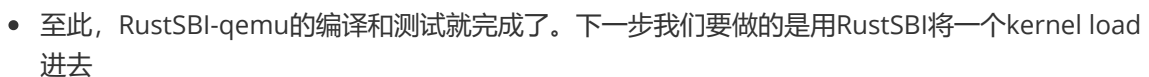
[rustsbi] Implementation: RustSBI-QEMU Version 0.1.0
[rustsbi-dtb] Hart count: cluster0 with 8 cores
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: ssoft, stimer, sext (0x222)
```

```

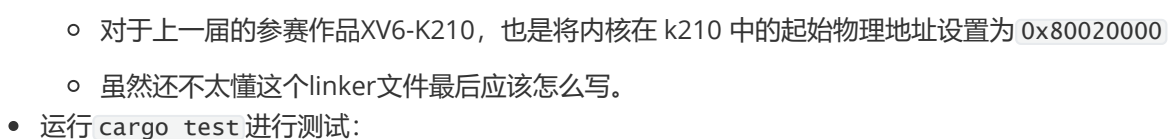
[rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage
(0xb1ab)
[rustsbi] pmp0: 0x10000000 ..= 0x10001fff (rw-)
[rustsbi] pmp1: 0x20000000 ..= 0x200fffff (rw-)
[rustsbi] pmp2: 0xc0000000 ..= 0xc3ffffff (rw-)
[rustsbi] pmp3: 0x80000000 ..= 0x8fffffff (rwx)
[rustsbi] enter supervisor 0x80200000
<< Test-kernel: Hart id = 0, DTB physical address = 0x1020
>> Test-kernel: Testing base extension
<< Test-kernel: Base extension version: 1
<< Test-kernel: SBI specification version: 3
<< Test-kernel: SBI implementation Id: 4
<< Test-kernel: SBI implementation version: 201
<< Test-kernel: Device mvendorid: 0
<< Test-kernel: Device marchid: 0
<< Test-kernel: Device mimpid: 0
>> Test-kernel: Testing SBI instruction emulation
<< Test-kernel: Current time: 12f204
<< Test-kernel: Time after operation: 13239b
>> Test-kernel: Trigger illegal exception
<< Test-kernel: Value of scause: Exception(IllegalInstruction)
<< Test-kernel: Illegal exception delegate success
>> Stop hart 3, return value 0
>> Hart 0 state return value: 0
>> Hart 1 state return value: 4
>> Hart 2 state return value: 4
>> Hart 3 state return value: 3
>> Hart 4 state return value: 0
<< Test-kernel: test for hart 0 success, wake another hart
>> Wake hart 1, sbi return value 0
>> Start test for hart 1, retentive suspend return value 0
<< The parameter passed to hart 2 resume is: 0x4567890a
>> Start hart 3 with parameter 0x12345678
>> Wake hart 2, sbi return value 0
>> SBI return value: 0
<< The parameter passed to hart 3 start is: 0x12345678
<< Test-kernel: All hart SBI test SUCCESS, shutdown

```

- 同时，我们看到 rustsbi-qemu.bin 文件被编译了出来。



- 下载源码，然后按照readme.md中的教程来：<https://github.com/rustsbi/rustsbi-k210.git>
- 它是启动位于 0x80020000 的操作系统内核，并在a1寄存器提供一个简单的设备树。-> 这大概意味着，我们写链接器的时候，需要将kernel的起始physical address放在这个位置，然后等着SBI去启动到这个位置。
 - 好像是，我看其提供的测试kernel的linker文件的base_address也是写在这个位置：



```
running 1 test
test test::run_test_kernel ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

- 测试成功
- 把板子连上进行测试:

RustSBI-qemu 测试kernel

- RustSBI-qemu的作者在源码中也准备一个测试用的kernel（在其提供的Readme.md文件中也有说明），现在试试按Readme.md的指引去load一次kernel

```
$ cargo test
```

- 运行成功的话会看到下面的输出:

```
ubuntu@VM-20-8-ubuntu:~/Desktop/rustsbi-qemu/rustsbi-qemu$ cargo install
cargo-binutils
    Updating crates.io index
    Ignored package `cargo-binutils v0.3.5` is already installed, use --
force to override
ubuntu@VM-20-8-ubuntu:~/Desktop/rustsbi-qemu/rustsbi-qemu$ rustup component
add llvm-tools-preview
info: component 'llvm-tools-preview' for target 'x86_64-unknown-linux-gnu'
is up to date
ubuntu@VM-20-8-ubuntu:~/Desktop/rustsbi-qemu/rustsbi-qemu$ cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.38s
    Running unittests src/main.rs (target/debug/deps/xtask-
5828b4179677ad42)

running 1 test
    Compiling bit_field v0.10.1
    Compiling spin v0.5.2
    Compiling spin v0.7.1
    Compiling lazy_static v1.4.0
    Compiling buddy_system_allocator v0.8.0
    Compiling riscv v0.7.0
    Compiling riscv v0.7.0 (https://github.com/rust-embedded/riscv?
rev=dc0bc37e#dc0bc37e)
    Compiling rustsbi v0.2.1
    Compiling rustsbi-qemu v0.1.0 (/home/ubuntu/Desktop/rustsbi-qemu/rustsbi-
qemu/rustsbi-qemu)
    Finished dev [unoptimized + debuginfo] target(s) in 3.59s
    Compiling bit_field v0.10.1
    Compiling spin v0.7.1
    Compiling spin v0.5.2
    Compiling buddy_system_allocator v0.8.0
    Compiling lazy_static v1.4.0
    Compiling riscv v0.6.0
    Compiling test_kernel v0.1.0 (/home/ubuntu/Desktop/rustsbi-qemu/rustsbi-
qemu/test_kernel)
```



```
Finished dev [unoptimized + debuginfo] target(s) in 2.10s
test run_test_kernel ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 6.75s
```

- 现在为止，编译，load kernel的测试都能正常进行。但对于RustSBI工作的原理还完全是一片模糊，接下来需要对阅读RustSBI-qemu的源码，去了解其工作的原理。

RustSBI作为Bootloader的使用

1. 以xv6-k210为例

- 参考文章：上一届作品xv6-k210 开机启动.md: <https://gitlab.eduxiji.net/AaronWu/oskernel2021-x/-/blob/ver1.4/doc/%E6%9E%84%E5%BB%BA%E8%B0%83%E8%AF%95-%E5%BC%80%E6%9C%BA%E5%90%AF%E5%8A%A8.md>. 下面的文字也主要是对 开机启动.md 这篇文章的总结，加上我自己的一些理解：
- 我们要实现的是在k210上的双核启动：
- 首先在linker.ld文件中，其规定了内核的起始点在 0x80020000 (和上文RustSBI要启动内核的位置对应)

```
/* xv6-k210的linker.ld文件 */
OUTPUT_ARCH(riscv)
ENTRY(_start)

BASE_ADDRESS = 0xffffffff80020000;

SECTIONS
{
    /* Load the kernel at this address: "." means the current address */
    . = BASE_ADDRESS;

    kernel_start = .;

    . = ALIGN(4K);

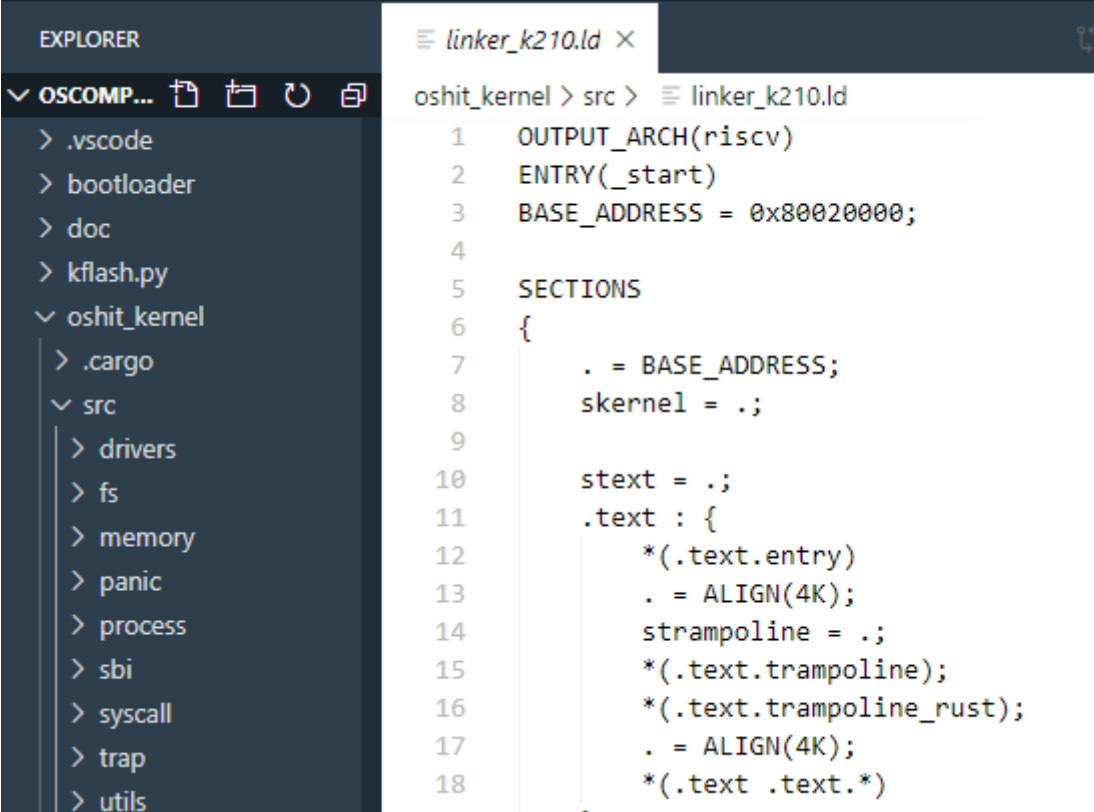
    text_start = .;
    .text : {
        *(.text .text.*)
        . = ALIGN(0x1000);
        _trampoline = .;
        *(trampsec)
        . = ALIGN(0x1000);
        ASSERT(. - _trampoline == 0x1000, "error: trampoline larger than one page");
        PROVIDE(etext = .);
    }

    . = ALIGN(4K);
    rodata_start = .;
    .rodata : {
        srodata = .;
        *(.rodata .rodata.*)
        erodata = .;
    }
}
```


- 跳转到main.c中的main()函数。最终在main()函数下，将会完成模块的初始化工作，包括内存管理，中断，进程管理等等。
- 开机启动.md中提到，我们这么做是得益于RustSBI已经将M Mode下要做的准备工作都做完了，我们只需要在S Mode下做内核的初始化工作就好了

2. 以OS-HIT为例

- 以下总结是基于上一届的作品OS-HIT: https://gitlab.eduxiji.net/willson0v0/oscomp_handin
- 首先，链接器也是将操作系统内核的起始位置放在 0x80020000 上的，linker_k210.ld的源码：



```

linker_k210.ld
1  OUTPUT_ARCH(riscv)
2  ENTRY(_start)
3  BASE_ADDRESS = 0x80020000;
4
5  SECTIONS
6  {
7      . = BASE_ADDRESS;
8      skernel = .;
9
10     stext = .;
11     .text : {
12         *(.text.entry)
13         . = ALIGN(4K);
14         strampoline = .;
15         *(.text.trampoline);
16         *(.text.trampoline_rust);
17         . = ALIGN(4K);
18         *(.text .text.*)

```

- 在0x80020000位置，其内核的入口为 entry.asm 文件：

```

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

.section .bss.stack
.globl boot_stack
boot_stack:          # CRT setup: stack
.space 4096 * 16
.globl boot_stack_top
boot_stack_top:

```

- 可以看到，它做的事也是分配内核栈空间，然后跳转到rust_main这个位置，进行各个模块的初始化！

