OS-lab1 report

刘泓尊 2018011446 计84

Chapter1实现的内容

1. 完成了实验环境的配置和工具链安装
2. 移除了标准库依赖，按照tutorial-book实现了OS的起始逻辑，迁移到了裸机运行环境
3. 实现了console.rs中的print的功能，以及对应的sbi_call系统调用接口。
4. 实现了彩色化LOG的输出功能，支持error, warn, info, debug, trace。代码位于logging.rs
5. 在makefile中加入了方便调试的UNOPTIMIZED+DEBUG模式编译命令`make run-debug`，加入了方便使用二进制调试工具`objdump, readobj, file, addr2line`等工具的脚本

彩色化 LOG

借助Cargo crate`log = "0.4"`这个库实现了彩色化LOG的功能。

为了实现彩色化输出，我实现了print_in_color函数和with_color宏，输出带有颜色的文本。其次实现了SimpleLogger类来实现Log接口，在调用error!等宏时进行等级判断等逻辑。实现了logging::init()函数来在OS初始化的时候获取LOG环境参数，设置运行时的LOG等级，以实现等级过滤的目的。

默认LOG等级是INFO。支持的LOG参数有`[ERROR, WARN, INFO, DEBUG, TRACE, OFF]`。OFF表示关闭所有LOG。(参数均大写)

我在main.rs中添加了如下代码用于测试：

```
// 输出 os 内存空间布局
info!(".text [{:#x}, {:#x})", stext as usize, etext as usize);
info!(".rodata [{:#x}, {:#x})", srodata as usize, erodata as usize);
info!(".data [{:#x}, {:#x})", sdata as usize, edata as usize);
info!(
    "boot_stack [{:#x}, {:#x})",
    boot_stack as usize, boot_stack_top as usize
);
info!(".bss [{:#x}, {:#x})", sbss as usize, ebss as usize);
```

```
// print `Hello World` in different log level
error!("Hello World!");
warn!("Hello World!");
info!("Hello World!");
debug!("Hello World!");
trace!("Hello World!");
```

之后在命令行测试效果:

1. 执行命令`make debug LOG=ERROR`:

```
[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Hello, world!
[ERROR][0] Hello World!
Panicked at src/main.rs:65 Shutdown machine!
```

2. 执行命令 `make debug LOG=WARN`：

```
[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Hello, world!
[ERROR][0] Hello World!
[WARN][0] Hello World!
Panicked at src/main.rs:65 Shutdown machine!
```

3. 执行命令 `make debug LOG=INFO`：

```
[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Hello, world!
[INFO][0] .text [0x80200000, 0x80203000)
[INFO][0] .rodata [0x80203000, 0x80205000)
[INFO][0] .data [0x80205000, 0x80206000)
[INFO][0] boot_stack [0x80206000, 0x80216000)
[INFO][0] .bss [0x80216000, 0x80216000)
[ERROR][0] Hello World!
[WARN][0] Hello World!
[INFO][0] Hello World!
Panicked at src/main.rs:65 Shutdown machine!
```

4. 执行命令 `make debug LOG=DEBUG`：

```
[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Hello, world!
[INFO][0] .text [0x80200000, 0x80203000)
[INFO][0] .rodata [0x80203000, 0x80205000)
[INFO][0] .data [0x80205000, 0x80206000)
[INFO][0] boot_stack [0x80206000, 0x80216000)
[INFO][0] .bss [0x80216000, 0x80216000)
[ERROR][0] Hello World!
[WARN][0] Hello World!
[INFO][0] Hello World!
[DEBUG][0] Hello World!
Panicked at src/main.rs:65 Shutdown machine!
```

5. 执行命令 `make debug LOG=TRACE`：

```
[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Hello, world!
[INFO][0] .text [0x80200000, 0x80203000)
[INFO][0] .rodata [0x80203000, 0x80205000)
[INFO][0] .data [0x80205000, 0x80206000)
[INFO][0] boot_stack [0x80206000, 0x80216000)
[INFO][0] .bss [0x80216000, 0x80216000)
[ERROR][0] Hello World!
[WARN][0] Hello World!
[INFO][0] Hello World!
[DEBUG][0] Hello World!
[TRACE][0] Hello World!
Panicked at src/main.rs:65 Shutdown machine!
```

问答作业

1. **为了方便 os 处理，M态软件会将 S 态异常/中断委托给 S 态软件，请指出有哪些寄存器记录了委托信息，rustsbi 委托了哪些异常/中断？（也可以直接给出寄存器的值）**

   riscv使用`mideleg`，`medeleg`分别将中断和异常从M态委托到S态.

   `mideleg`指示中断委托，每一位是否为1表示对应编号的中断是否委托给S态。`medeleg`则对应异常委托。

   `mideleg`和`medeleg`的值在 RustSBI 启动时以控制台输出信息的方式给出：

```
[rustsbi] RustSBI version 0.1.1
.------       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
```

   中断委托向量 `mideleg` = `0x0222` = `0b 0000_0010_0010_0010`

   异常委托向量 `medeleg` = `0xb1ab` = `0b 1011_0001_1010_1011`

   从riscv 特权态手册中的Exception Code (section 3.1.16 page 37)可以看到:

   中断委托了

   - Supervisor software interrupt (code=1)
   - Supervisor timer interrupt (code=5)
   - Supervisor external interrupt (code=9)

   异常委托了

   - Instruction address misaligned (code = 0)
   - Instruction access fault (code = 1)
   - Breakpoint (code = 3)
   - Load access fault (code = 5)
   - Store/AMO access fault (code = 7)
   - Environment call from U-mode (code = 8)

- Instruction page fault (code = 12)
- Load page fault (code = 13)
- Store/AMO page fault (code = 15)

2. **请学习 gdb 调试工具的使用，并通过 gdb 简单跟踪从机器加电到跳转到 `0x80200000` 的简单过程。只需要描述重要的跳转即可，只需要描述在 qemu 上的情况。**

   a. **加电后跳转到 `0x8000_0000`**:

   运行 `make debug` 开启GDB调试之后，我们首先来看最开始的10条指令。 `0x1000` 处将 `t0` 设置为 `0x1000` , 之后在 `0x100c` 处将 `t0` 加载为 `0x101a` 处的地址 `0x8000_0000` , 之后在 `0x1010` 处跳转到了地址t0 = `0x8000_0000` . 这个地址就是RustSBI的起始地址。

```
0x0000000000001000 in ?? ()
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a1,t0,32
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      t0,24(t0)
   0x1010:      jr      t0
   0x1014:      unimp
   0x1016:      unimp
   0x1018:      unimp
   0x101a:      0x8000
   0x101c:      unimp
```

   b. `0x8000_0000` **到RustSBI::main()**:

   对比 `0x8000_0000` 处的汇编代码可以发现，它正是RustSBI 的 `start()` 函数。

```
(gdb) x/10i $pc
=> 0x80000000:  csrr    a2,mhartid
   0x80000004:  lui     t0,0x0
   0x80000008:  addi    t0,t0,7
   0x8000000c:  bltu    t0,a2,0x8000003a
   0x80000010:  auipc   sp,0x200
   0x80000014:  addi    sp,sp,-16
   0x80000018:  lui     t0,0x10
   0x8000001c:  mv      t0,t0
   0x80000020:  beqz    a2,0x8000002e
   0x80000022:  mv      t1,a2
```

```
// extern "C" for Rust ABI is by now unsupported for naked functions
unsafe extern "C" fn start() -> ! {
    asm!(
            "
        csrr    a2, mhartid
        lui     t0, %hi(_max_hart_id)
        add     t0, t0, %lo(_max_hart_id)
        bgtu    a2, t0, _start_abort
        la      sp, _stack_start
        lui     t0, %hi(_hart_stack_size)
        add     t0, t0, %lo(_hart_stack_size)
```

在 `start()` 函数最后，跳转到了 **RustSBI::main()** 函数(in rustsbi/platform/qemu/src/main.rs):

```
2:
.endif
    sub     sp, sp, t0
    csrw    mscratch, zero
    j       main

_start_abort:
    wfi
    j _start_abort
", options(noreturn))
```

使用GDB反汇编可以得到 **main** 的入口地址是 `0x8000_2572` .

```
(gdb) x/10i $pc
=> 0x8000002e:  sub     sp,sp,t0
   0x80000032:  csrw    mscratch,zero
   0x80000036:  j       0x80002572
   0x8000003a:  wfi
   0x8000003e:  j       0x8000003a
   0x80000040:  unimp
```

c. **`RustSBI::main()` 到 `s_mode_start()`** :

进入main()函数之后，在main()函数最后，设置了 `mepc` = `s_mode_start` ,这也是mret命令将会跳转到的地址。最后调用了 `enter_privileged()` (in rustsbi/rustsbi/src/privileged.rs) 函数。

```
unsafe {
    mepc::write(s_mode_start as usize);
    mstatus::set_mpp(MPP::Supervisor);
    rustsbi::enter_privileged(mhartid::read(), dtb_pa)
}
```

我们再来看 `enter_privileged` 函数, 该函数最后调用了 `mret` ，也就是跳转到了之前设置的 `mepc` 向量，即 `s_mode_start` (in rustsbi/platform/qemu/src/main.rs).

```
pub unsafe fn enter_privileged(mhartid: usize, dtb_pa: usize) -> ! {
    match () {
        #[cfg(any(target_arch = "riscv32", target_arch = "riscv64"))]
        () => asm!("
            csrrw   sp, mscratch, sp
            mret
        ", in("a0") mhartid, in("a1") dtb_pa, options(nomem, noreturn)),
```

使用GDB获得 **`s_mode_start`** (也就是此时 `mepc`)的地址是 `0x800023da` :

```
(gdb) info r mepc
mepc           0x800023da       2147492826
```

d. **`s_mode_start` 到 `0x8020_0000`** :

`s_mode_start` 通过 `jr ra` 进行了一次跳转，对应的源代码和反汇编如下：

```
unsafe extern "C" fn s_mode_start() -> ! {
    asm!("
1:  auipc ra, %pcrel_hi(1f)
    ld ra, %pcrel_lo(1b)(ra)
    jr ra
.align  3
1:  .dword 0x80200000
    ", options(noreturn))
}
```

```
(gdb) x/10i $pc
=> 0x800023da:  auipc   ra,0x0
   0x800023de:  ld      ra,14(ra)
   0x800023e2:  ret
   0x800023e4:  nop
   0x800023e8:  unimp
   0x800023ea:  0x8020
   0x800023ec:  unimp
```

在 `jr ra` 之前设置了 `ra` 的值，可以看到最终 `ra` 等于 `0x8000_23ea` 处的数值 `0x8020_0000`，最终 `jr ra` 就跳到了 `0x8020_0000`，也就是我们自己实现的**OS起始代码**。

e. `0x8020_0000` **到OS 的** `rust_main()`

之后就进入了.stext段，而里面第一个被放置的又是来自 `entry.asm` 中的段 .text.entry，也就是OS入口点 `_start`，之后就跳转到了 `rust_main()`，开始执行OS的rust部分的代码。

```
    .globl _start
_start: // _start 是整个程序的入口点
    la sp, boot_stack_top // 将 sp 设置为我们预留的栈空间的栈顶位置
    call rust_main // 调用 rust_main
    // 以上这两条指令单独作为一个名为 .text.entry 的段
```