

操作系统实验2

邓人嘉 21301032

一、实验步骤

1.1 编译生成内核镜像

- 执行cargo build --release编译

```
@95ccb44ef577:/mnt/os x + v
PS C:\Users\lenovo> docker run -it --mount type=bind,source=C:\Users\lenovo\Desktop\OperatingSystem\GardenerOS\experimen
t2,destination=/mnt myopenuler

Welcome to 5.15.90.1-microsoft-standard-WSL2

System information as of time: Fri Nov 3 03:09:27 UTC 2023

System load: 0.00
Processes: 5
Memory used: 13.0%
Swap used: 0%
Usage On: 1%
Users online: 0

[root@95ccb44ef577 /]# ls
afs bin dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
[root@95ccb44ef577 /]# cd mnt
[root@95ccb44ef577 mnt]# ls
'21301032-' '$'\351\202\223\344\272\272\345\230\211' '-' '$'\345\256\236\351\252\214' '2.md'
[root@95ccb44ef577 mnt]# cd os
[root@95ccb44ef577 os]# ls
Cargo.lock Cargo.toml src target
[root@95ccb44ef577 os]# cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.25s
[root@95ccb44ef577 os]# cargo build --release
Compiling os v0.1.0 (/mnt/os)
Finished release [optimized] target(s) in 1.45s
[root@95ccb44ef577 os]#
```

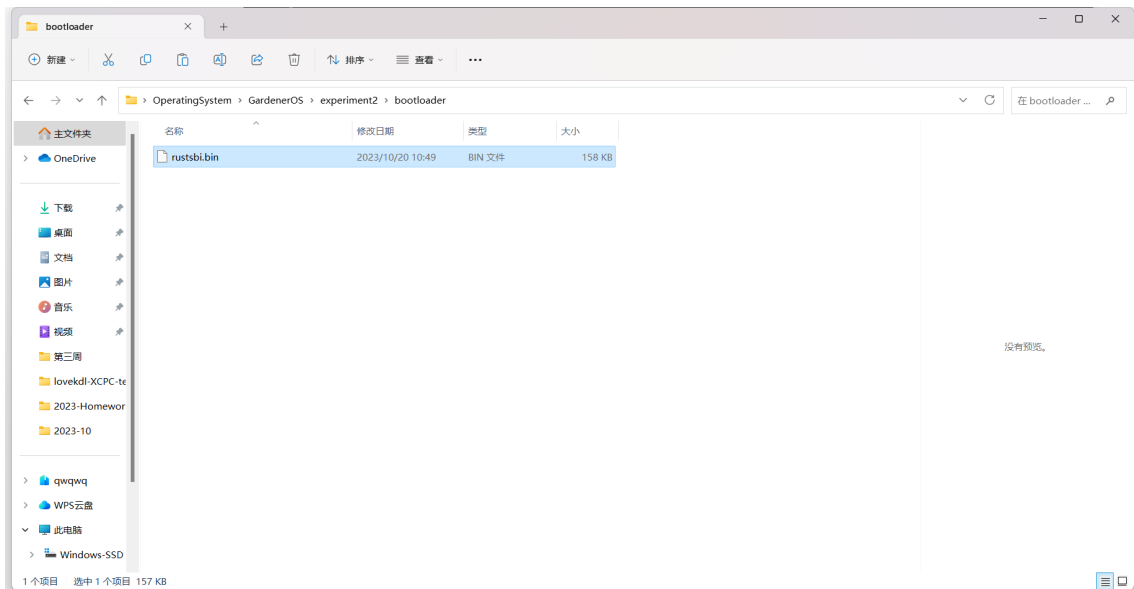
- 把编译生成的ELF执行文件转成binary文件

```
@95ccb44ef577:/mnt/os x + v

System load: 0.01
Processes: 6
Memory used: 12.6%
Swap used: 0%
Usage On: 1%
Users online: 0

[root@95ccb44ef577 /]# ls
afs bin dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
[root@95ccb44ef577 /]# cd mnt
[root@95ccb44ef577 mnt]# ls
'21301032-' '$'\351\202\223\344\272\272\345\230\211' '-' '$'\345\256\236\351\252\214' '2.md'
src
target
[root@95ccb44ef577 mnt]# cd os
[root@95ccb44ef577 os]# ls
Cargo.lock Cargo.toml src target
[root@95ccb44ef577 os]# cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.12s
[root@95ccb44ef577 os]# cargo build --release
Finished release [optimized] target(s) in 0.07s
[root@95ccb44ef577 os]# ls
Cargo.lock Cargo.toml src target
[root@95ccb44ef577 os]# rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/os --strip-
all -O binary target/riscv64gc-unknown-none-elf/release/os.bin
[root@95ccb44ef577 os]#
```

- 将rustsbi.bin放在bootloader目录



- 接着，加载运行生成的二进制文件。

```
@95ccb44ef577:/mnt/os x + v
[root@95ccb44ef577 os]# ls
Cargo.lock Cargo.toml
[root@95ccb44ef577 os]# rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/release/os --strip-all -O binary target/riscv64gc-unknown-none-elf/release/os.bin
[root@95ccb44ef577 os]# cd ..
[root@95ccb44ef577 mnt]# mkdir bootloader
[root@95ccb44ef577 mnt]# cd os
[root@95ccb44ef577 os]#
[root@95ccb44ef577 os]# qemu-system-riscv64 -machine virt -nographic -bios ../bootloader/rustsbi.bin -device loader,file=target/riscv64gc-unknown-none-elf/release/os.bin,addr=0x80200000
[rustsbi] RustSBI version 0.2.0-alpha.6

[rustsbi] Implementation: RustSBI-QEMU Version 0.0.2
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: ssoft, stimer, sext (0x222)
[rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage (0x1blab)
[rustsbi] pmp0: 0x100000000 ..= 0x10001fff (rwx)
[rustsbi] pmp1: 0x800000000 ..= 0x8fffffff (rwx)
[rustsbi] pmp2: 0x0 ..= 0xfffffffffffff (---)
qemu-system-riscv64: clint: invalid write: 00000004
[rustsbi] enter supervisor 0x80200000
```

进入了死循环

- 分析，Entry不是0x80200000

```
@95ccb44ef577:/mnt/os x + v
[root@95ccb44ef577 os]#
[root@95ccb44ef577 os]# rust-readobj -h target/riscv64gc-unknown-none-elf/release/os
File: target/riscv64gc-unknown-none-elf/release/os
Format: elf64-littleriscv
Arch: riscv64
AddressSize: 64bit
LoadName: <Not found>
ElfHeader {
  Ident {
    Magic: (7F 45 4C 46)
    Class: 64-bit (0x2)
    DataEncoding: LittleEndian (0x1)
    FileVersion: 1
    OS/ABI: SystemV (0x0)
    ABIVersion: 0
    Unused: (00 00 00 00 00 00 00)
  }
  Type: Executable (0x2)
  Machine: EM_RISCV (0xF3)
  Version: 1
  Entry: 0x114B6
  ProgramHeaderOffset: 0x40
  SectionHeaderOffset: 0x108C90
  Flags [ (0x5)
    EF_RISCV_FLOAT_ABI_DOUBLE (0x4)
    EF_RISCV_RVC (0x1)
  ]
  HeaderSize: 64
  ProgramHeaderEntrySize: 56
  ProgramHeaderCount: 6
  SectionHeaderEntrySize: 64
  SectionHeaderCount: 18
  StringTableSectionIndex: 16
}
```

1.2 指定内存布局

- 修改os/.cargo/config

```
@95ccb44ef577-jmnt/os/.caty x + v
1 # os/.cargo/config
2 [build]
3 target = "riscv64gc-unknown-none-elf"
4
5 [target.riscv64gc-unknown-none-elf]
6 rustflags = [
7     "-C", "link-arg=-Tsrc/linker.ld",
8 ]|
~
~
~
~
~
2 ~
~
~
3 ~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --
```

- 创建链接文件linker.ld

```

1 OUTPUT_ARCH(riscv)
2 ENTRY(_start)
3 BASE_ADDRESS = 0x80200000;
4
5 SECTIONS
6 {
7     . = BASE_ADDRESS;
8     skernel = .;
9
10    stext = .;
11    .text : {
12        *(.text.entry)
13        *(.text .text.*)
14    }
15
16    . = ALIGN(4K);
17    etext = .;
18    srodata = .;
19    .rodata : {
20        *(.rodata .rodata.*)
21        *(.srodata .srodata.*)
22    }
23
24    . = ALIGN(4K);
25    erodata = .;
26    sdata = .;
27    .data : {
28        *(.data .data.*)
29        *(.sdata .sdata.*)
30    }
31
32    . = ALIGN(4K);
33    edata = .;
34    .bss : {

```

"linker.ld" 49L, 695B

1.3 配置栈空间布局

- 创建汇编文件entry.asm

```
@95ccb44ef577:/mnt/os/src x + v
1  .section .text.entry
2  .globl _start
3  _start:
4  la sp, boot_stack_top
5  call rust_main
6
7  .section .bss.stack
8  .globl boot_stack
9  boot_stack:
10 .space 4096 * 16
11 .globl boot_stack_top
12 boot_stack_top:|
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT -- 12,16 All
```

- main.rs 中嵌入汇编代码并声明应用入口 rust_main

```
@95ccb44ef577:/mnt/os/src x + v
1 #![no_std]
2 #![no_main]
3
4
5 use core::arch::global_asm;
6
7 global_asm!(include_str!("entry.asm"));
8
9 #[no_mangle]
10 pub fn rust_main() -> ! {
11     loop{};
12 }
13
14
15 use core::panic::PanicInfo;
16
17 #[panic_handler]
18 fn panic(_info: &PanicInfo) -> ! {
19     loop {}
20 }
21 use core::arch::asm;
22
23 const SYSCALL_EXIT: usize = 93;
24
25 fn syscall(id: usize, args: [usize; 3]) -> isize {
26     let mut ret: isize;
27     unsafe {
28         asm!("ecall",
29             in("x10") args[0],
30             in("x11") args[1],
31             in("x12") args[2],
32             in("x17") id,
33             lateout("x10") ret
34         );
35     }
36 }
```

1.4 清空bss段

- 为了保证内存的正确性，在main.rs中撰写代码清空.bss段

```
@95ccb44ef577:/mnt/os/src x + v
1 #![no_std]
2 #![no_main]
3
4 fn clear_bss() {
5     extern "C" {
6         fn sbss();
7         fn ebss();
8     }
9     (sbss as usize..ebss as usize).for_each(|a| unsafe { (a as *mut u8).write_volatile(0) });
10 }
11
12 use core::arch::global_asm;
13
14 global_asm!(include_str!("entry.asm"));
15
16 #[no_mangle]
17 pub fn rust_main() -> ! {
18     loop{};
19 }
20
21
22 use core::panic::PanicInfo;
23
24 #[panic_handler]
25 fn panic(_info: &PanicInfo) -> ! {
26     loop {}
27 }
28
29 use core::arch::asm;
30
31 const SYSCALL_EXIT: usize = 93;
32
33 fn syscall(id: usize, args: [usize; 3]) -> isize {
34     let mut ret: isize;
35     unsafe {
36         asm!(
37             "syscall",
38             in("x10") id,
39             in("x11") args[0],
40             in("x12") args[1],
41             in("x17") args[2],
42             lateout("x10") ret,
43             :
44         );
45     }
46     ret
47 }
48
49 -- INSERT --
10,2 Top
```

1.5 实现裸机打印输出信息

- 创建文件sbi.rs

```
@95ccb44ef577:/mnt/os/src x + v
1 #![allow(unused)]
2
3 use core::arch::asm;
4
5 const SBI_SET_TIMER: usize = 0;
6 const SBI_CONSOLE_PUTCHAR: usize = 1;
7 const SBI_CONSOLE_GETCHAR: usize = 2;
8 const SBI_CLEAR_IPI: usize = 3;
9 const SBI_SEND_IPI: usize = 4;
10 const SBI_REMOTE_FENCE_I: usize = 5;
11 const SBI_REMOTE_SFENCE_VMA: usize = 6;
12 const SBI_REMOTE_SFENCE_VMA_ASID: usize = 7;
13 const SBI_SHUTDOWN: usize = 8;
14
15 #[inline(always)]
16 fn sbi_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> isize {
17     let mut ret;
18     unsafe {
19         asm!(
20             "ecall",
21             in("x10") arg0,
22             in("x11") arg1,
23             in("x12") arg2,
24             in("x17") which,
25             lateout("x10") ret,
26             :
27         );
28     }
29     ret
30 }
31
32 pub fn console_putchar(c: usize) {
33     sbi_call(SBI_CONSOLE_PUTCHAR, c, 0, 0);
34 }
35
36 pub fn console_getchar() -> isize {
37     "sbi.rs" 41L, 952B
38 }
1,1 Top
```

- [illegible]

- 实现os/src/lang_items.rs

- ```
@95ccb44ef577:/mnt/os/src x + v
```
- ```
1 use crate::sbi::shutdown;
2 use core::panic::PanicInfo;
3
4 #[panic_handler]
5 fn panic(info: &PanicInfo) -> ! {
6     if let Some(location) = info.location() {
7         println!(
8             "Panicked at {}:{} {}",
9             location.file(),
10            location.line(),
11            info.message().unwrap()
12        );
13     } else {
14         println!("Panicked: {}", info.message().unwrap());
15     }
16     shutdown()
17 }
```
- ```
"lang_items.rs" 17L, 406B
```
- 1,1 All

## 1.7 修改main.rs输出测试信息

- 修改main.rs的内容

```
@95ccb44ef577:/mnt/os/src x + v
1 #![no_std]
2 #![no_main]
3 #![feature(panic_info_message)]
4 #[macro_use]
5
6 mod console;
7 mod lang_items;
8 mod sbi;
9
10 use core::arch::global_asm;
11
12 global_asm!(include_str!("entry.asm"));
13 fn clear_bss() {
14 extern "C" {
15 fn sbss();
16 fn ebss();
17 }
18 (sbss as usize..ebss as usize).for_each(|a| unsafe { (a as *mut u8).write_volatile(0) });
19 }
20
21 #[no_mangle]
22 pub fn rust_main() -> ! {
23 extern "C" {
24 fn stext();
25 fn etext();
26 fn srodata();
27 fn erodata();
28 fn sdata();
29 fn edata();
30 fn sbss();
31 fn ebss();
32 fn boot_stack();
33 fn boot_stack_top();
34 }
35 clear_bss();
36 println!("Hello, world!");
37 println!("{}", text [{}:{}], {}:{}", stext as usize, etext as usize);
38 println!("{}", srodata [{}:{}], {}:{}", srodata as usize, erodata as usize);
39 println!("{}", sdata [{}:{}], {}:{}", sdata as usize, edata as usize);
40 println!(
41 "boot_stack [{}:{}]",
42 boot_stack as usize, boot_stack_top as usize
43);
44 println!("{}", sbss [{}:{}], {}:{}", sbss as usize, ebss as usize);
45 println!("Hello, world!");
46 panic!("Shutdown machine!");
47 }
48
"main.rs" 48L, 1139B 1,1 All
```

## 1.8 重新编译以及生成二进制文件

- 编译

```
@95ccb44ef577:/mnt/os x + v
[root@95ccb44ef577 src]# vim linker.ld
[root@95ccb44ef577 src]# vim linker.ld
[root@95ccb44ef577 src]# touch entry.asm
[root@95ccb44ef577 src]# vim entry.asm
[root@95ccb44ef577 src]# vim main.rs
[root@95ccb44ef577 src]# ls
entry.asm linker.ld main.rs
[root@95ccb44ef577 src]# touch sbi.rs
[root@95ccb44ef577 src]# vim sbi.rs
[root@95ccb44ef577 src]# vim sbi.rs
[root@95ccb44ef577 src]# ls
entry.asm linker.ld main.rs sbi.rs
[root@95ccb44ef577 src]# vim console.rs
[root@95ccb44ef577 src]# vim console.rs
[root@95ccb44ef577 src]# vim lang_items.rs
[root@95ccb44ef577 src]# vim lang_items.rs
[root@95ccb44ef577 src]# vim main.rs
[root@95ccb44ef577 src]# vim main.rs
[root@95ccb44ef577 src]# cd ..
[root@95ccb44ef577 os]# ls
Cargo.lock Cargo.toml src target
[root@95ccb44ef577 os]#
[root@95ccb44ef577 os]#
[root@95ccb44ef577 os]#
[root@95ccb44ef577 os]#
[root@95ccb44ef577 os]#
[root@95ccb44ef577 os]# cargo build --release
Compiling os v0.1.0 (/mnt/os)
Finished release [optimized] target(s) in 0.85s
[root@95ccb44ef577 os]#
```

- 生成二进制文件





- ```
Panicked at src/main.rs:46 Shutdown machine!
[root@95ccb44ef577 os]# vim Makefile
[root@95ccb44ef577 os]# |

@95ccb44ef577:/mnt/os x + v
1 # Building
2 TARGET := riscv64gc-unknown-none-elf
3 MODE := release
4 KERNEL_ELF := target/${TARGET}/${MODE}/os
5 KERNEL_BIN := ${KERNEL_ELF}.bin
6 DISASM_TMP := target/${TARGET}/${MODE}/asm
7
8 # BOARD
9 SBI ?= rustsbi
10 BOOTLOADER := ../bootloader/${SBI}.bin
11
12 # KERNEL ENTRY
13 KERNEL_ENTRY_PA := 0x80200000
14
15 # Binutils
16 OBJDUMP := rust-objdump --arch-name=riscv64
17 OBJCOPY := rust-objcopy --binary-architecture=riscv64
18
19 # Disassembly
20 DISASM ?= -x
21
22 build: $(KERNEL_BIN)
23
24 env:
25     (rustup target list | grep "riscv64gc-unknown-none-elf (installed)") || rustup target add $(TARGET)
26     cargo install cargo-binutils
27     rustup component add rust-src
28     rustup component add llvm-tools-preview
29
"Makefile" 60L, 1512B 1,1 Top

@95ccb44ef577:/mnt/os x + v
[root@95ccb44ef577 os]# vim Makefile
[root@95ccb44ef577 os]# make run
    Finished release [optimized] target(s) in 0.13s
[rustsbi] RustSBI version 0.2.0-alpha.6

┌───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐
│   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │
│   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │
│   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │
│   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │ │   │
└───┘ └───┘ └───┘ └───┘ └───┘ └───┘ └───┘ └───┘ └───┘ └───┘

[rustsbi] Implementation: RustSBI-QEMU Version 0.0.2
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: ssoft, stimer, sext (0x222)
[rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage (0xb1ab)
[rustsbi] pmp0: 0x100000000 ..= 0x10001fff (rwx)
[rustsbi] pmp1: 0x800000000 ..= 0x8fffffff (rwx)
[rustsbi] pmp2: 0x0 ..= 0xffffffffffffff (---)
qemu-system-riscv64: clint: invalid write: 00000004
[rustsbi] enter supervisor 0x80200000
Hello, world!
.text [0x80200000, 0x80202000)
.rodata [0x80202000, 0x80203000)
.data [0x80203000, 0x80204000)
boot_stack [0x80204000, 0x80214000)
.bss [0x80214000, 0x80214000)
Hello, world!
Panicked at src/main.rs:46 Shutdown machine!
[root@95ccb44ef577 os]#
```

2.1 分析linker.ld和entry.asm所完成的功能

- No. 9 / 14

2.2 分析sbi模块和lang_items模块所完成的功能

- **sbi**模块可以更改系统调用为sbi调用。**sbi.rs**文件实现了**console_puchar**、**console_getchar**、**shutdown**三个函数，作用分别是输出一个字符、输入一个字符、关闭系统。**console_puchar**接口在**console.rs**文件中被调用，封装为输出一个字符串。**shutdown**接口在**lang_items.rs**文件中的**panic**函数被调用，用于关闭系统。
- **lang_items**实现了**panic**处理器。调用了**sbi**模块的**shutdown**接口，输出**panic**发生的错误位置以及错误信息。

2.3 如果将rustsbi.bin换成最新版本的会造成代码无法运行，分析原因并给出解决方法。

- 使用最新版的rustsbi.bin, 先输出了正常信息, 然后进入了死循环, 并且输出乱码, 如下图:

[illegible]

然后运行发现程序没有终止，进入了死循环，也没有输出，说明原因确实是：调用 `sbi_call(SBI_SHUTDOWN, 0, 0, 0)` 时没有成功关闭系统。

查看 `sbi_call` 函数：

```
16 fn sbi_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {
17     let mut ret;
18     unsafe {
19         asm!("ecall",
20             in("x10") arg0,
21             in("x11") arg1,
22             in("x12") arg2,
23             in("x17") which,
24             lateout("x10") ret
25         );
26     }
27     ret
28 }
```

考虑到新版 `rustsbi` 可能不是这样调用接口的。发现文档中说一些 SBI 函数被遗弃了，其中包含了 EID 为 `0x08` 的 `shutdown`。

The legacy SBI extensions is deprecated in favor of the other extensions listed below. The legacy console SBI functions (`sbi_console_getchar()` and `sbi_console_putchar()`) are expected to be deprecated; they have no replacement.

于是参照下图的 replacement EID 把 `shutdown` 的 EID 修改为 `0x53525354` (实际上是 `sbi_system_reset` 的 EID)。但其它被遗弃的函数的 EID 使用原来的并没有影响，所以不用改变，只需要改变 `shutdown` 的 EID 为 `sbi_system_reset` 的 EID `0x53525354` 就好。（并且需要添加 FID）

5.10. Function Listing

Table 5. Legacy Function List

Function Name	SBI Version	FID	EID	Replacement EID
<code>sbi_set_timer</code>	0.1	0	0x00	0x54494D45
<code>sbi_console_putchar</code>	0.1	0	0x01	N/A
<code>sbi_console_getchar</code>	0.1	0	0x02	N/A
<code>sbi_clear_ipi</code>	0.1	0	0x03	N/A
<code>sbi_send_ipi</code>	0.1	0	0x04	0x735049
<code>sbi_remote_fence_i</code>	0.1	0	0x05	0x52464E43
<code>sbi_remote_sfence_vma</code>	0.1	0	0x06	0x52464E43

16

Function Name	SBI Version	FID	EID	Replacement EID
<code>sbi_remote_sfence_vma_asid</code>	0.1	0	0x07	0x52464E43
<code>sbi_shutdown</code>	0.1	0	0x08	0x53525354
RESERVED			0x09-0x0F	

修改后的 `sbi.rs` 如下：

三、git截图

- git截图

