OS-lab2 report

刘泓尊 2018011446 计84

Chapter2 实现的内容

1. 增加了应用程序代码，封装了面向OS的系统调用
2. 实现了批处理系统的初始化和顺序加载逻辑
3. 实现了任务切换，涉及执行流和特权级的切换，保存和恢复现场等
4. 实现了syscall的分发，实现系统调用sys_write与sys_exit
5. 修改了makefile使得测试可以通过os目录下`make run CHAPTER=2/2_bad`完成

sys_write物理地址安全检查

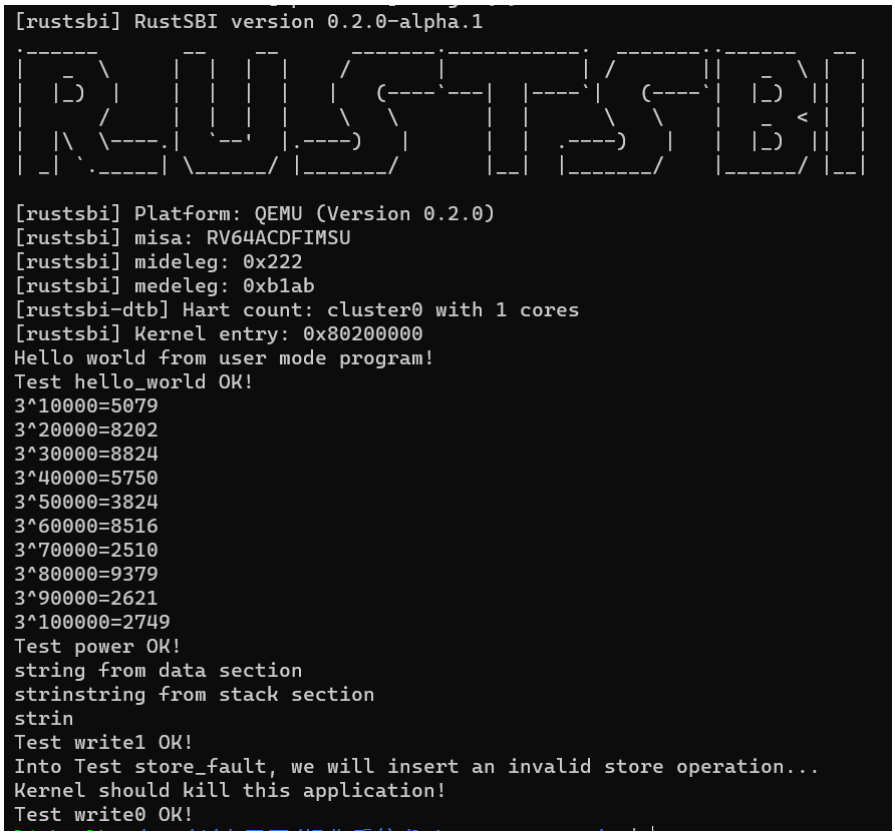
需要检测的是物理地址[buf, buf+len)在用户栈或者程序段内。

用户栈的区间是固定的，为`[USER_STACK.data.as_ptr(), USER_STACK.data.as_ptr()+USER_STACK_SIZE)`,其中`USER_STACK_SIZE = 4096`.

程序段的起始地址固定，为`[APP_BASE_ADDRESS, APP_BASE_ADDRESS + self.app_start[self.current_app] - self.app_start[self.current_app-1])`.其中`APP_BASE_ADDRESS =0x80400000`.

根据两个区间进行判断即可。

测试结果截图



问答作业

1. **正确进入 U 态后，程序的特征还应有：使用 S 态特权指令，访问 S 态寄存器后会报错。目前由于一些其他原因，这些问题不太好测试，请同学们可以自行测试这些内容（参考 前三个测例），描述程序出错行为，同时注意注明你使用的 sbi 及其版本。**

   rustSBI版本: `0.2.0-alpha.1`

   _ch2_bad_instruction:

   执行了S态指令 `sret` ， `IllegalInstruction` ,程序输出 `scause=0x2`

   _ch2_bad_register:

   执行了S态指令 `csrr` ， `IllegalInstruction` ,程序输出 `scause=0x2`

   _ch2t_bad_address:

   访问(写入)了0地址， `StoreFault` (因为页表没有打开，应该不是 `StorePageFault`). 本版本下程序没有输出，直接执行下一个程序了。

```
[rustsbi] RustSBI version 0.2.0-alpha.1
.------.        __    __       _______.-----------.  _______.:------.   __
|   _  \      |  |  |  |     /       |           | /       ||   _  \ |  |
|  |_)  |     |  |  |  |    |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /      |  |  |  |     \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Platform: QEMU (Version 0.2.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
scause=0x2
scause=0x2
```

2. **请结合用例理解 trap.S 中两个函数 `__alltraps` 和 `__restore` 的作用，并回答如下几个问题:**

   1. **L40: 刚进入 `__restore` 时， `a0` 代表了什么值。请指出 `__restore` 的两种使用情景。**

      `a0` 是 内核栈压入 Trap 上下文之后的栈顶地址 。

      1) 加载新的app时，也就是 `run_next_app` 函数会显式调用 `__restore` ，参数是内核栈压入 Trap 上下文之后的栈顶地址。

      2) 异常处理结束之后，从S态返回U态的过程会经过 `__restore` . 也就是 `call trap_handler` 会把 `trap_handler` 的返回地址设置成 `__restore` 的第一条指令。参数同样是内核栈压入 Trap 上下文之后的栈顶地址。

   2. **L46-L51: 这几行汇编代码特殊处理了哪些寄存器？这些寄存器的的值对于进入用户态有何意义？请分别解释。**

      特殊处理了 `sstatus` , `spec` , `sscratch` . 同时把 `t0, t1, t2` ,作为临时的中转寄存器， 因为 `csrw` 并不能直接从内存中读值。

      `sstatus` : 它的SPP字段给出了Trap之前CPU处于哪个特权级，进入用户态的时候需要恢复原样。我们需要保存它到上下文，因为 `sstatus` 在处理Trap的全程有意义，在 `sret` 的时候还用到了它，并且Trap嵌套的时候它的值也会被修改。

      `spec` : 记录Trap发生时执行的最后一条指令，是 `sret` 的目标地址。我们需要保存它到上下文，因为 `spec` 在处理Trap的全程有意义，在 `sret` 的时候还用到了它，并且Trap嵌套的时候它的值也会被修改。

`sscratch`：作为重要的**换栈**中转寄存器。Trap发生的时候，进入__alltraps，将sp设置到内核栈栈顶，而sscratch保存之前的sp，即用户栈栈顶。这使得Trap操作可以在内核栈上正常进行，并且之前的sp可以正确被保存到上下文中。在restore之后，先从上下文中取出用户栈地址，放到sscratch中暂存，之后交换sp和sscratch，将sp指向用户栈，sscratch保存内核栈栈顶。完成换栈。

3. **L53-L59: 为何跳过了 `x2` 和 `x4`？**

   `x2`是`sp`. 此时Trap处理还没有完成有关内核栈的工作，需要sp在内核栈指向正确的位置，因此不能提前加载到sp. 我们需要先将用户栈栈顶加载到sscratch, 在最后再设置到sp.

   `x4`是`tp`. 一般编译器会避免用它，我们也没有用到这个寄存器，所以不用保存和恢复。

4. **L63: 该指令之后，`sp` 和 `sscratch` 中的值分别有什么意义？**

   `sp`将指向用户栈栈顶

   `sscratch`将指向内核栈栈顶（此时已经弹出了Trap上下文）

5. **`__restore`：中发生状态切换在哪一条指令？为何该指令执行之后会进入用户态？**

   `sret`将从S态切换到U态。此时CPU会完成2件事：

   - CPU 会将当前的特权级按照 `sstatus` 的 `SPP` 字段设置为 U 或者 S；
   - CPU 会跳转到 `sepc` 寄存器指向的那条指令，然后开始向下执行。

   而`sstatus`的SPP位在`app_init_context`中就设置为了U态，所以sret之后会回到U态。

   ```
   1   sstatus.set_spp(SPP::User);
   ```

6. **L13： 该指令之后，`sp` 和 `sscratch` 中的值分别有什么意义？**

   `sp`是内核栈栈顶地址

   `sscratch`是用户栈栈顶地址

7. **从 U 态进入 S 态是哪一条指令发生的？**

   用户程序发生异常（Exception）或调用`ecall`等中断(Interrupt)的时候，CPU会将pc设置为Trap入口地址`stvec`(在我们的OS中也就是`__alltraps`), 同时完成U到S态的特权级切换。

3. **程序陷入内核的原因有中断和异常（系统调用），请问 riscv64 支持哪些中断 / 异常？如何判断进入内核是由于中断还是异常？描述陷入内核时的几个重要寄存器及其值。**

   rv64支持的中断和异常（仅仅描述了S态和U态异常，因为实验没涉及到M态相关知识）如下表，1开头的是中断，0开头的是异常：

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved for future standard use*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved for future standard use*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved for future standard use*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

`scause`的最高位是0时表示异常，是1时表示中断。

重要寄存器:

`sepc`: 保存发生Trap时的pc(当然S态处理Trap时也可以修改它来实现APP切换). 用于Trap处理结束之后返回到spec.

`stvec`: Trap入口向量。发生异常的时候CPU将pc设置成`stvec`的BASE << 2, 在这里BASE就是`__alltraps`

`scause`: 记录Trap的原因（Trap代码）。

`stval`: 给出 Trap 附加信息。

`sstatus`： S 特权级最重要的 CSR, 可以从很多方面控制 S 特权级的CPU行为和执行状态. `SPP` 等字段给出 Trap 发生之前 CPU 处在哪个特权级（S/U）等信息.

4. **对于任何中断，`__alltraps`中都需要保存所有寄存器吗？你有没有想到一些加速`__alltraps`的方法？简单描述你的想法。**

   并不需要。比如`ecall`系统调用的时候，可能涉及不到APP的切换，而编译器会在U态执行包裹`ecall`函数的前后保存调用者保存寄存器，这时我们就不需要全部保存。何况`x0`始终不变，不必保存，`tp`几乎用不到，也不用保存。

   可以根据不同种类的Trap设计不同的上下文保存与恢复流程，跳过用不到的寄存器，细粒度地进行Trap分发。当然这会增加代码的复杂度，也不方便对寄存器的索引。