OS homework 4

于新雨 计25 2022010841

第一题

选择了 rcore

批处理操作系统中应用程序管理数据结构的组成

应用程序的内存布局在 `user/src/linker.ld` 中，然后管理数据结构的结构为 AppManager，组成如下

```
struct AppManager {
    num_app: usize,
    current_app: usize,
    app_start: [usize; MAX_APP_NUM + 1],
}
```

记录了当前跑的 app，总的 app 数，app 的起始地址

应用程序管理数据结构的初始化过程

在 os/build.rs/insert_app_data 函数里面读取了 `../user/build/bin/` 中的程序数量，然后写到 _num_app 里面，然后把程序写到了`app_{num}_start app_{num}_end` 之间，然后在 `APP_MANAGER` 初始化时用来初始化 app_start num_app，current_app 被赋值为0

trapframe 组成

Trap 上下文组成如下

```
pub struct TrapContext {
    /// general regs[0..31]
    pub x: [usize; 32],
    /// CSR sstatus
    pub sstatus: Sstatus,
    /// CSR sepc
    pub sepc: usize,
}
```

分别保存了通用寄存器，sstatus 和 sepc

syscall 中 trapframe 保存

在 `__all_traps` 函数里面，分别将几乎所有用到了的通用寄存器保存到栈上，把 sstatus，sepc 保存在栈上，再把用户态栈地址（保存在 sscratch 中）存到栈上

syscall 后恢复应用程序上下文

在 `__restore` 函数中依次从栈上 load 出通用寄存器和 sstatus sepc 的值，然后交换 sp 和 sscratch 的值恢复用户栈

参数和返回值传递

看到 user/src/syscall.rs 里面，syscall number 用 x17 传，参数可以用 x10 - x12 传，或者 x10 - x15 传，返回值在 x10 中被返回

第二题

其他的启动引导方式：

1. UEFI
    - 见 wikipedia 上的介绍，UEFI 是 BIOS 的替代物，它使用UEFI（统一可扩展固件接口）标准，支持安全启动功能，验证引导加载程序和内核签名
2. CoreBoot
    - 见这里, 是一个旨在替代大多数计算机中专有固件（BIOS或UEFI）的软件项目，提供一种轻量级固件，设计为仅执行加载和运行现代32位或64位操作系统所需的最少任务

启动引导约定：

- $a0 应包含当前核心的hartid，$a1 应包含内存中设备树的地址。
- $satp = 0： 如果存在MMU，必须将其禁用。
- RISC-V内核期望被放置在PMD边界（对于rv64为2MB对齐，对于rv32为4MB对齐）。 请注意，如果不是这样，EFI stub 将重定位内核

如何运行 qemu-gdb：
先在 ./os 的 Makefile 把 make gdbclient 中的命令改成 `@gdb-multiarch -ex 'file $(KERNEL_ELF)' -ex 'set arch riscv:rv64' -ex 'target remote localhost:1234'`

在 ./os 里面运行 make gdbserver 和 make gdbclient，gdb 会 attach 到 qemu 进行调试

我们在 rust_main 的位置下断点，然后 vmmap 命令运行

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
             Start                End Perm     Size Offset File
            0x1000            0x10000 rw-p     f000      0 <explored>
         0x7fe00000         0x80600000 rw-p   800000      0 <explored>
         0x7fe01000         0x80601000 rwxp   800000      0 <explored>
         0x7fe01000         0x80601000 rwxp   800000      0 <explored>
         0x7fe01000         0x80601000 rwxp   800000      0 <explored>
         0x7fe03000         0x80603000 rw-p   800000      0 <explored>
         0x7fe03000         0x80603000 rw-p   800000      0 <explored>
         0x7fe09000         0x80609000 rw-p   800000      0 <explored>
         0x7fe09000         0x80609000 rw-p   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe36000         0x80636000 rwxp   800000      0 <explored>
         0x7fe37000         0x80637000 rw-p   800000      0 <explored>
         0x7fe38000         0x80638000 rw-p   800000      0 <explored>
         0x7fe38000         0x80638000 rw-p   800000      0 <explored>
         0x86c00000         0x87400000 rw-p   800000      0 <explored>
```

得到如图结果，所以知道 RISCV 内核是在 2MB 对齐的位置

$a0, $a1 值分别如下

```
*A0   0x80238292 ◂— 0
*A1   0x87000000 ◂— 0xf20e0000edfe0dd0
```

运行 `p $satp` 可以得到 satp 寄存器值为0，所以整体符合条件