Lab-3

先说下自己对entry.asm中这一段的理解

```
boot_page_table:
    .quad 0
    .quad 0
    # 注释 4
    # 第 2 项：0x8000_0000 -> 0x8000_0000，0xcf 表示 VRWXAD 均为 1
    .quad (0x80000 << 10) | 0xcf
    .zero 507 * 8
    # 注释 5
    # 第 510 项：0xffff_ffff_8000_0000 -> 0x8000_0000，0xcf 表示 VRWXAD 均为 1
    .quad (0x80000 << 10) | 0xcf
    .quad 0
```

这里的建立映射关系听起来有点玄乎，其实就是用汇编写入了一个页表项

页表项

Aa 比特位	值	说明
63-54	0	保留字段
53-28	2	PPN[2]
27-19	0	PPN[1]
18-10	0	PPN[0]
9-8	0	RSW
7	1	D
6	1	A
5	0	G
4	0	U
3	1	X
2	1	W
1	1	R
0	1	V

1. 原理：在 `os/src/entry.asm` 中，`boot_page_table` 的意义是什么？当跳转执行 `rust_main` 时，不考虑缓存，硬件通过哪些地址找到了 `rust_main` 的第一条

指令？

boot_page_table就是一个根页表，而且是有内容的三级页表，其第0项为空，第1项为空，第二项是一个1G的大页，之后又是507个空项，第510项又是1个1G的大页，第511项又为空，这就是该页表的全部内容了。为什么两个大页，一个要放在页表的第二项（从0开始），一个要放在第510项呢。

拿第二个页表项举例子，当访问虚拟地址 `0x8000_0000` 至 `0xc000_0000`（不包括0xc000_0000）时，由于这一范围内的地址的VPN[2]都为2，所以就会访问页表项的第二项（从0项开始），而该页表项中存的PPN[2]为2，换算成物理地址就是`0x8000_0000`，所以刚好实现了虚拟地址到物理地址的映射，第511项则是实现了`0xffff_ffff_8000_0000` 至 `0xffff_ffff_c000_0000` 到 `0x8000_0000` 至 `0xc000_0000` 的映射。

为什么要做映射呢？由于linker.ld中指定的起始地址是`0xffff_ffff_8020_0000`，但是opensbi只能把程序放在`0x8020_0000` 上，所以我们只能做这样两个页表项来实现虚拟地址的转换了。

执行 `jal rust_main` 时，硬件首先家在rust_main对应的地址，由于linker.ld中指定的起始地址是`0xffff_ffff_8020_0000` 所以rust_main的地址应该在该地址后面不远。

由于我们启用了Sb39，因此硬件从satp低位读取根页表号，也就是 `boot_page_table` 的物理页号。rust_mian的虚拟地址对应boot_page_table的510项，该项的XWR为111，所以它指向一个物理页，对应的物理地址是 `0x8000_0000` 由于V是1，所以该页有效，直接根据偏移量就可以找到rust_main的入口

2. 分析：为什么 `Mapping` 中的 `page_tables` 和 `mapped_pairs` 都保存了一些 `FrameTracker`？二者有何不同？

page_tables是用来存放页表的物理页的，mapped_pairs是用来存放程序用到的物理页的，两者加起来就是所有的物理页了

3. 分析：假设某进程需要虚拟地址 A 到物理地址 B 的映射，这需要操作系统来完成。那么操作系统在建立映射时有没有访问 B？如果有，它是怎么在还没有映射的情况下访问 B 的呢？

应该不需要访问B，只需要在页表项中插入数据即可，也不需要对物理页初始化之类的，这也能解释为什么C语言未初始化的指针能打印出各种数据。内核的虚拟地址空间虽然只映射了一部分物理内存，但是可以通过偏移量访问映射之外的物理地址

4. 实验：了解并实现时钟页面置换算法（或任何你感兴趣的算法），可以自行设计样例来比较性能

   - 置换算法只需要修改 `os/src/memory/mapping/swapper.rs`
   - 在 `main.rs` 中调用 `create_kernel_thread` 来创建线程，你可以任意修改其中运行的函数，以达到测试效果

   我实现了一个时钟页面置换算法，实现思路非常简单，首先是给页面制定一个优先级

   ```
   a表示页面是否被访问过，m表示页面是否被修改过
    用0表示(a=false, m=false) 没有使用也没有修改，被逐出的优先级最高；
    用1表示(a=true, m=false) 使用过，但是没有修改过，优先级第二；
    用2表示(a=false, m=true) 没有使用过，但是修改过，优先级第三；
    用3表示(a=true, m=true) 使用过也修改过，优先级第四。
   ```

   如果一个页面被修改过，那么优先级+2，如果被访问过那么优先级+1，淘汰页面时优先淘汰优先级低的，页面使用Vec来存储，淘汰时遍历Vec来找到优先级最低的即可。经测试，该算法可以通过swap_test，性能没有验证，但肯定是没有FIFO好了。

   ```
   Terminal:  Local (2)  +
   page_fault 1336
   page_fault 1337
   page_fault 1338
   page_fault 1339
   page_fault 1340
   page_fault 1341
   test passed
   page_fault 1342
   thread 1 exit with code 0
   src/process/processor.rs:87: 'all threads terminated, shutting down'

   # root @ bdb20918c35c in /home/rCore/rCore-Tutorial/os on git:lab-3+ x [16:15:51]
   ```