# 前言

本实验指导是为 2023 秋季 OS 训练营第三阶段 - 项目1：Unikernel 方向，而做的前期准备。这是第二周的练习内容，目标是在 ArceOS Unikernel OS 模式下，完成几个基本实验与附加练习，为支持多应用的实习任务做准备。

1. 基本实验：本指导书以增量的方式基本给出了 1 到 5 共 5 个实验的源码及过程，大家照着做一遍，以熟悉基本原理机制。
2. 附加练习：基于基本实验，根据自己的理解，增加一些实现，以达到练习要求目标。总共 6 个练习。

对完成实验和练习的过程要求：

1. 按照下节 ***环境准备*** 第 2 点要求分别建立分支，并 commit。
2. 更新群里的"第二周实习进度汇总"表。
3. 成功的截图发到群里。
4. 仓库链接发邮件到 sun_ye@masscloud.com。**注意**：第一周发过邮件的同学，只要你的仓库未改，**不用**发邮件。我们保留了你们的仓库，可以直接去对应分支上查看各位的代码。所以请务必按照后面的要求建立分支和标记 commit 消息。

## 环境准备

1. Fork ArceOS 的工程，clone 到本地。工程链接如下：

   ```
   git@github.com:arceos-org/arceos.git
   ```

   通过 `git log` 查看 commit id 是否为 *51a42ea4d65a53bf6b43fc35a27a3ff1e9e284c7*。如果不是，回退到这个 commit，确保工作的基线与指导书一致。

   > 注意：进行过项目 1 第一周练习的同学，这步应该已经具备，直接从第 2 步开始。

2. 建立并切换到分支 week2_base：

   ```
   git checkout main
   git checkout -b week2_base
   ```

   这个分支对应**基本实验**。开始实验时，每完成一个，就 commit 一次，commit msg 是 "step N"，N 是实验序号。

3. 建立并切换到分支 week2_exercise：

   ```
   git checkout main
   git checkout -b week2_exercise
   ```

这个分支对应**附加练习**。根据每个附加练习的要求完成，每完成一个，commit 一次，commit msg 是 "exercise N"，N 是练习序号。

4. 执行 `make run ARCH=riscv64` 测试一下环境，我们的实习平台是 **riscv64-qemu-virt**。

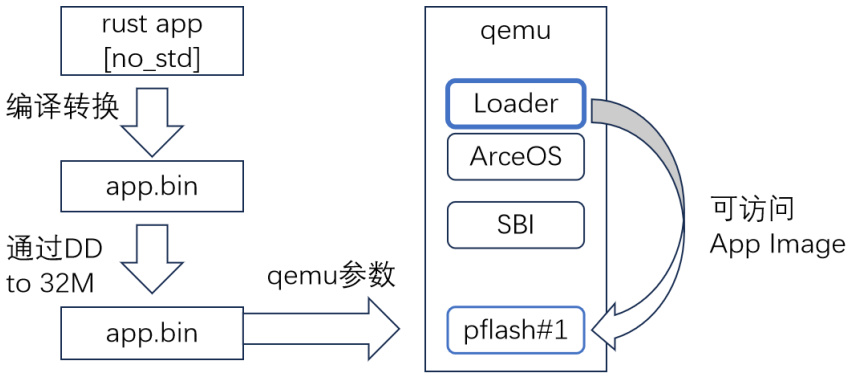
```
       d8888                            .d88888b.   .d8888b.
      d88888                           d88P" "Y88b d88P  Y88b
     d88P888                           888     888 Y88b.
    d88P 888 888d888  .d8888b  .d88b.  888     888  "Y888b.
   d88P  888 888P"   d88P"    d8P  Y8b 888     888     "Y88b.
  d88P   888 888     888      88888888 888     888       "888
 d8888888888 888     Y88b.    Y8b.     Y88b. .d88P Y88b  d88P
d88P     888 888      "Y8888P  "Y8888   "Y88888P"   "Y8888P"

arch = riscv64
platform = riscv64-qemu-virt
target = riscv64gc-unknown-none-elf
smp = 1
build_mode = release
log_level = warn

Hello, world!
```

看到这个输出表示环境正常。

# 实验1：从外部加载应用

## 实验目标

实现加载器 loader，从外部加载 bin 应用到 ArceOS 地址空间。




## 实验步骤

1. 编写一个 no_std 应用作为实验对象，命名为 hello_app，目录与 arceos 目录**并列**。它的主文件 main.rs 如下：

```
#![no_std]
#![no_main]

use core::panic::PanicInfo;

#[no_mangle]
unsafe extern "C" fn _start() -> ! {
    core::arch::asm!(
        "wfi",
        options(noreturn)
    )
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

   现在只有一行代码 `wfi`。

2. 在 hello_app 根目录下加一个 rust-toolchain.toml，内容如下：

```
[toolchain]
profile = "minimal"
channel = "nightly"
components = ["rust-src", "llvm-tools-preview", "rustfmt", "clippy"]
targets = ["riscv64gc-unknown-none-elf"]
```

定制默认的 toolchain，关键是指定 target = "riscv64gc-unknown-none-elf"，即 riscv64 体系结构的裸机程序。

3. 执行一系列命名，包括编译，转换和打包，生成可被 ArceOS 加载的 image。

```
cargo build --target riscv64gc-unknown-none-elf --release

rust-objcopy --binary-architecture=riscv64 --strip-all -O binary
target/riscv64gc-unknown-none-elf/release/hello_app ./hello_app.bin

dd if=/dev/zero of=./apps.bin bs=1M count=32
dd if=./hello_app.bin of=./apps.bin conv=notrunc

mkdir -p ../arceos/payload
mv ./apps.bin ../arceos/payload/apps.bin
```

得到image文件apps.bin，上面最后两步把它转移到arceos/payload目录下，以方便启动。

> 这步的一系列动作可以考虑写入一个shell脚本，便于今后执行。

4. 转移到 ArceOS 工程，在 apps 目录下，实现一个新的 app，名为 loader。仿照 helloworld 应用创建，它的 main.rs 如下：

```
#![cfg_attr(feature = "axstd", no_std)]
#![cfg_attr(feature = "axstd", no_main)]

#[cfg(feature = "axstd")]
use axstd::println;

const PLASH_START: usize = 0x22000000;

#[cfg_attr(feature = "axstd", no_mangle)]
fn main() {
    let apps_start = PLASH_START as *const u8;
    let apps_size = 32; // Dangerous!!! We need to get accurate size of
apps.

    println!("Load payload ...");

    let code = unsafe { core::slice::from_raw_parts(apps_start, apps_size)
};
    println!("content: {:#x}", bytes_to_usize(&code[..8]));

    println!("Load payload ok!");
```

```
}

#[inline]
fn bytes_to_usize(bytes: &[u8]) -> usize {
    usize::from_be_bytes(bytes.try_into().unwrap())
}
```

> 注意：
>
> 1、qemu 有两个 pflash，其中第一个被保留做扩展的 bios，我们只能用第二个，它的开始地址 0x22000000。
>
> 2、创建 loader 应用后，注意在 Cargo.toml 中增加依赖：axstd = { path = "../../ulib/axstd", optional = true }

5. ArceOS 目前没有对 pflash 所在的地址空间进行映射，增加映射。

   在文件 modules/axhal/src/platform/riscv64_qemu_virt/boot.rs 中，恒等映射从 0 开始的 1G 空间。

```
unsafe fn init_boot_page_table() {
    // 0x8000_0000..0xc000_0000, VRWX_GAD, 1G block
    BOOT_PT_SV39[2] = (0x80000 << 10) | 0xef;
    // 0xffff_ffc0_8000_0000..0xffff_ffc0_c000_0000, VRWX_GAD, 1G block
    BOOT_PT_SV39[0x102] = (0x80000 << 10) | 0xef;

    // 0x0000_0000..0x4000_0000, VRWX_GAD, 1G block
    BOOT_PT_SV39[0] = (0x00000 << 10) | 0xef;
}
```

> 注意：只有最后两行是我们新增的映射，这样 ArceOS 就可以访问 pflash 所在的地址空间了。

6. 现在可以编译 ArceOS 了，修改一下 Makefile 的默认参数。看一下修改前后 diff 的结果：

```
 # General options
-ARCH ?= x86_64
+ARCH ?= riscv64

 # App options
-A ?= apps/helloworld
+A ?= apps/loader
```

   默认 arch 改为 riscv64，默认应用改为 apps/loader 即我们的加载器。

7. 修改一下 qemu 的启动参数，让 pflash 加载之前的 image 就是那个 apps.bin，然后启动 ArceOS 内核及 loader 应用。

   修改 scripts/make/qemu.mk，在 qemu 启动参数中加上一项：

```
-drive if=pflash,file=$(CURDIR)/payload/apps.bin,format=raw,unit=1
```

> 注意：在 qemu.mk 文件中修改 qemu 启动参数的具体位置不是唯一的。

8. 把 apps/loader 加到根目录 Cargo.toml 中的 [workspace] 下的 members 列表中。执行 `make run` 测试。

```
arch = riscv64
platform = riscv64-qemu-virt
target = riscv64gc-unknown-none-elf
smp = 1
build_mode = release
log_level = warn

Load payload ...
content: 0x7300501000000000
Load payload ok!
```

注意打印的 content 就是测试应用 hello_app.bin 的内容，可以用如下命令确认：

```
xxd -ps ./hello_app.bin
730050100000
```

对比后可发现内容一致，注意后续 0000 是 dd 命令产生的效果。

# 练习

## 练习 1：

main 函数中，固定设置 app_size = 32，这个显然是不合理甚至危险的。

请为 image 设计一个头结构，包含应用的长度信息，loader 在加载应用时获取它的实际大小。

执行通过。

## 练习 2：

在练习 1 的基础上，扩展 image 头结构，让 image 可以包含两个应用。

第二个应用包含唯一的汇编代码是 `ebreak`。

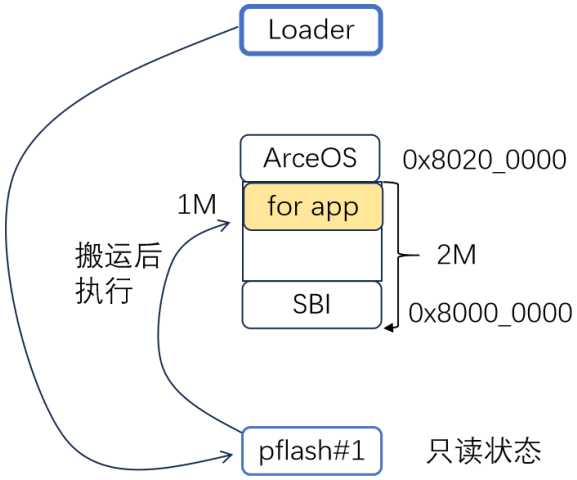
如实验 1 的方式，打印出每一个应用的二进制代码。

# 实验2：把应用拷贝到执行区域并执行

## 实验目标

目前应用已经被加载到 pflash 的地址区间内，但是处于只读状态，一旦执行到写数据的指令时，就会触发异常。

所以本实验的目标就是把应用搬运到可读可写可执行的内存区域。




## 实验步骤

1. 从 pflash 区域拷贝到 0x8010_0000，即 Kernel 前面 1M 处作为应用的执行区，改造一下 loader 的 main.rs （这里只给出增量代码）。

```
#[cfg_attr(feature = "axstd", no_mangle)]
fn main() {
    let load_start = PLASH_START as *const u8;
    let load_size = 32; // Dangerous!!! We need to get accurate size of
apps.

    println!("Load payload ...");

    let load_code = unsafe { core::slice::from_raw_parts(load_start,
load_size) };
    println!("load code {:?}; address [{:?}]", load_code,
load_code.as_ptr());

    // app running aspace
    // SBI(0x80000000) -> App <- Kernel(0x80200000)
    // 0xffff_ffc0_0000_0000
    const RUN_START: usize = 0xffff_ffc0_8010_0000;
```

```
    let run_code = unsafe {
        core::slice::from_raw_parts_mut(RUN_START as *mut u8, load_size)
    };
    run_code.copy_from_slice(load_code);
    println!("run code {:?}; address [{:?}]", run_code,
run_code.as_ptr());

    println!("Load payload ok!");
}
```

`make run` 显示如下，代码被正常拷贝到目标区域。

```
arch = riscv64
platform = riscv64-qemu-virt
target = riscv64gc-unknown-none-elf
smp = 1
build_mode = release
log_level = warn

Load payload ...
load code [115, 0, 80, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]; address [0x22000000]
run code [115, 0, 80, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]; address [0xffffffc080100000]
Load payload ok!
```

2. 然后从新的位置开始执行 App 的逻辑，在上面main函数的末尾追加代码执行。

```
    println!("Execute app ...");

    // execute app
    unsafe { core::arch::asm!("
        li      t2, {run_start}
        jalr    t2
        j       .",
        run_start = const RUN_START,
    )}
```

`make run` 显示 "Execute app ..." 之后卡住了，但这是正常的，注意汇编最后一句是无限循环。

> 如果提示需要 #![feature(asm_const)] 之类的支持，按照提示处理。
>
> 另：qemu 卡住后，退出到命令行的按键是，Ctrl+a 后按 x。

3. 要想知道是否成功，需要通过查看 qemu.log 进行判断。ArceOS 支持输出这种日志，为方便，直接改 Makefile 默认项。

```
-QEMU_LOG ?= n
+QEMU_LOG ?= y
```

再次 `make run`，查看当前产生 qemu.log，

```
IN:
Priv: 1; Virt: 0
0xffffffc080100000:  10500073          wfi
0xffffffc080100004:  0000              illegal
```

可以看到，我们确实执行到了 App 的唯一一行代码 `wfi`。

# 练习

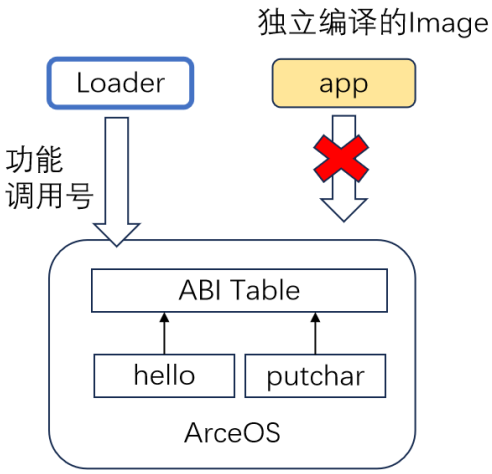
## 练习 3:

批处理方式执行两个单行代码应用，第一个应用的单行代码是 `noop`，第二个的是 `wfi`。

# 实验3：通过 ABI 调用 ArceOS 功能

## 实验目标

到目前为止，我们的外部应用 hello_app 还无法做实际的事情。原因就是，这个应用是独立于 ArceOS 之外编译的单独 Image，现在 ArceOS 还没有为它提供调用接口。本实验中，我们先来做一个准备，为 ArceOS 增加简单的 ABI 接口支持，首先让内嵌应用 Loader 能够通过 ABI 方式调用功能；下个实验我们再进一步改成让外部应用通过ABI 调用功能。




## 实验步骤

1. 在 loader 中引入 abi_table，注册两个调用过程。一个是无参数的 abi_hello，另一个是单参数的 abi_putchar。在 main.rs 中增加：

```
const SYS_HELLO: usize = 1;
const SYS_PUTCHAR: usize = 2;

static mut ABI_TABLE: [usize; 16] = [0; 16];

fn register_abi(num: usize, handle: usize) {
    unsafe { ABI_TABLE[num] = handle; }
}

fn abi_hello() {
    println!("[ABI:Hello] Hello, Apps!");
}

fn abi_putchar(c: char) {
    println!("[ABI:Print] {c}");
```

```
}
```

2. 在 ArceOS 内嵌应用 loader 中，测试按照调用号调用 ABI 功能。我们可以分别测试一下两个功能。

   下面是在 main() 函数中调用的，改造原来的那几行汇编，变成下面这样：

```
    register_abi(SYS_HELLO, abi_hello as usize);
    register_abi(SYS_PUTCHAR, abi_putchar as usize);

    println!("Execute app ...");
    let arg0: u8 = b'A';

    // execute app
    unsafe { core::arch::asm!("
        li      t0, {abi_num}
        slli    t0, t0, 3
        la      t1, {abi_table}
        add     t1, t1, t0
        ld      t1, (t1)
        jalr    t1
        li      t2, {run_start}
        jalr    t2
        j       .",
        run_start = const RUN_START,
        abi_table = sym ABI_TABLE,
        //abi_num = const SYS_HELLO,
        abi_num = const SYS_PUTCHAR,
        in("a0") arg0,
    )}
```

可以看到，在启动应用之前，我们在 loader 本地先测试了 **SYS_PUTCHAR** 的功能调用。如下是执行结果：

```
arch = riscv64
platform = riscv64-qemu-virt
target = riscv64gc-unknown-none-elf
smp = 1
build_mode = release
log_level = warn

Load payload ...
load code [[115, 0, 80, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]; address [0x22000000]
run code [[115, 0, 80, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]; address [0xffffffc080100000]
Load payload ok!

Execute app ...
[ABI:Print] A
```

```
QEMU: Terminated
```

看到打印出字符 'A'，测试成功！

打印后卡住了，还是用 Ctrl+a 后 x 退出。下面练习 4 就实验一下退出功能。

# 练习

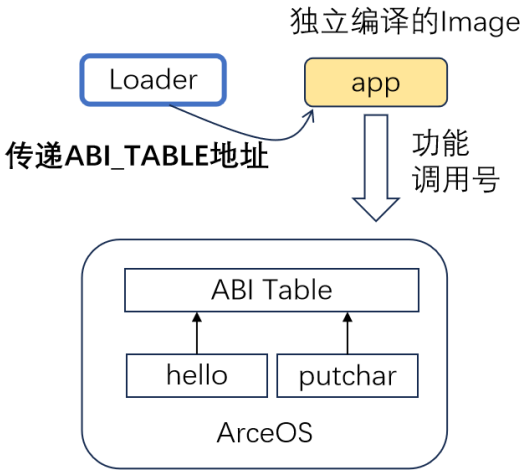
## 练习 4:

本实验已经实现了1 号调用 - SYS_HELLO，2 号调用 - SYS_PUTCHAR，请实现 3 号调用 - SYS_TERMINATE 功能调用，作用是让 ArceOS 退出，相当于 OS 关机。

# 实验4：正式在 App 中调用 ABI

## 实验目标

上个实验已经实现了 ABI 机制，本实验我们让外部应用正式使用 ABI。这里需要解决一个问题，外部应用必须获得 **ABI 入口表的基地址**，才能以调用号为偏移，找到对应的功能。因为 loader 是 ArceOS 的内嵌应用，知道这个地址，我们让它把地址传过来。




## 实验步骤

1. 在 loader 的 main 函数中，把直接调用 abi 的代码删除，改为如下代码：

```
    println!("Execute app ...");

    // execute app
    unsafe { core::arch::asm!("
        la      a7, {abi_table}
        li      t2, {run_start}
        jalr    t2
        j       .",
        run_start = const RUN_START,
        abi_table = sym ABI_TABLE,
    )}
```

   loader 不再调用 abi，只是把 ABI_TABLE 的地址传给外部应用 hello_app。注意：我们传递地址用的是 a7 寄存器。

2. 应用 hello_app 通过 ABI 获取 ArceOS 服务，修改它的 main.rs：

```
#![feature(asm_const)]
#![no_std]
```

```
#![no_main]

//const SYS_HELLO: usize = 1;
const SYS_PUTCHAR: usize = 2;

#[no_mangle]
unsafe extern "C" fn _start() -> ! {
    let arg0: u8 = b'C';
    core::arch::asm!("
        li      t0, {abi_num}
        slli    t0, t0, 3
        add     t1, a7, t0
        ld      t1, (t1)
        jalr    t1
        wfi",
        abi_num = const SYS_PUTCHAR,
        in("a0") arg0,
        options(noreturn),
    )
}

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

可以看到，我们从 a7 寄存器获得了 ABI_TABLE 的基地址，再结合调用号就可以获得对应功能的入口。

> 注意：调用号乘以 8 才是偏移（64 位系统的函数指针 8 个字节）。

3. **重新执行** hello_app 的编译转换步骤，见实验 1 的第 3 步。

   之前如果已经把步骤写入 shell 脚本，这步就比较简单。

4. 执行 `make run`，测试结果：

```
Execute app ...
[ABI:Print] C
QEMU: Terminated
```

打印字符 'C'，成功！

# 练习

## 练习 5：

按照如下要求改造应用 hello_app：

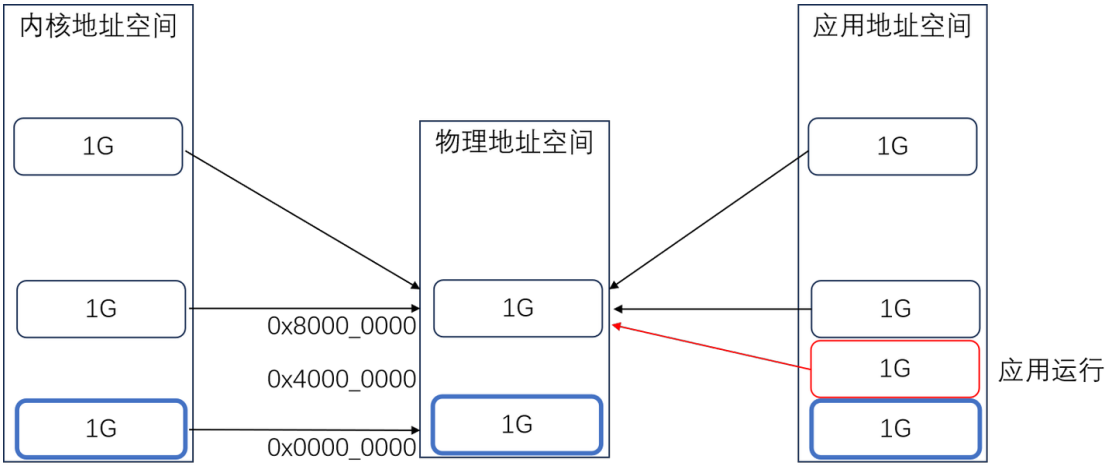
1. 把三个功能调用的汇编实现封装为函数，以普通函数方式调用。例如，SYS_PUTCHAR 封装为 `fn putchar(c: char)`。
2. 基于打印字符函数 putchar 实现一个高级函数 `fn puts(s: &str)`，可以支持输出字符串。
3. 应用 hello_app 的执行顺序是：Hello 功能、打印字符串功能、退出功能。

> 别忘了应用修改后，还要执行实验 1 的第 3 步完成编译转换和覆盖旧应用。如果当时封装了 shell 脚本，这步比较方便。

# 实验5：支持内核和应用分离的地址空间及切换

## 实验目标

目前，ArceOS Unikernel 是单地址空间。我们希望为每个外部应用建立独立的地址空间，当应用被调度时，切换到此应用的地址空间上。这样对每个应用，就可以采用固定的地址空间布局。现在从 0x4000_0000 地址开始的 1G 区域空闲，那我们就以它作为应用的地址空间。




## 实验步骤

1. 在应用 loader 中，为应用 hello_app 建立独立的页表（仅有一级），并实现初始化和切换函数。main.rs 最后追加如下：

```
//
// App aspace
//

#[link_section = ".data.app_page_table"]
static mut APP_PT_SV39: [u64; 512] = [0; 512];

unsafe fn init_app_page_table() {
    // 0x8000_0000..0xc000_0000, VRWX_GAD, 1G block
    APP_PT_SV39[2] = (0x80000 << 10) | 0xef;
    // 0xffff_ffc0_8000_0000..0xffff_ffc0_c000_0000, VRWX_GAD, 1G block
    APP_PT_SV39[0x102] = (0x80000 << 10) | 0xef;

    // 0x0000_0000..0x4000_0000, VRWX_GAD, 1G block
    APP_PT_SV39[0] = (0x00000 << 10) | 0xef;

    // For App aspace!
    // 0x4000_0000..0x8000_0000, VRWX_GAD, 1G block
```

```
    APP_PT_SV39[1] = (0x80000 << 10) | 0xef;
}

unsafe fn switch_app_aspace() {
    use riscv::register::satp;
    let page_table_root = APP_PT_SV39.as_ptr() as usize -
axconfig::PHYS_VIRT_OFFSET;
    satp::set(satp::Mode::Sv39, 0, page_table_root >> 12);
    riscv::asm::sfence_vma_all();
}
```

APP_PT_SV39 的链接位置 ".data.app_page_table"，定义在 modules/axhal/linker.lds.S 中：

```
        _sdata = .;
        *(.data.boot_page_table)
        . = ALIGN(4K);
        *(.data.app_page_table)
        . = ALIGN(4K);
        *(.data .data.*)
```

就紧跟在系统页表位置 *(.data.boot_page_table) 的下面。**注意**，咱们增加的只有中间两行。

此外，代码中引用了两个外部的 crate，分别是 axconfig 和 riscv，修改 loader 的 Cargo.toml：

```
[dependencies]
axstd = { path = "../../ulib/axstd", optional = true }
axconfig = { path = "../../modules/axconfig" }

[target.'cfg(any(target_arch = "riscv32", target_arch =
"riscv64"))'.dependencies]
riscv = "0.10"
```

2. 虽然已经建立了应用的页表，但我们先不切换，直接去访问应用的地址空间 0x4010_0000，去看看这将会导致什么样的状况。

   在 loader 的 main 函数中修改如下：

```
    - const RUN_START: usize = 0xffff_ffc0_8010_0000;
    + const RUN_START: usize = 0x4010_0000;
```

> 注意：这个 0x4010_0000 所在的 1G 空间在原始的内核地址空间中是不存在的。

执行 `make run`，系统异常 **STORE_FAULT**，因为没有启用应用的地址空间映射。

```
Unhandled trap Exception(StorePageFault) @ 0xffffffc080202aa8:
TrapFrame {
    regs: GeneralRegisters {
        ra: 0xffffffc0802005f4,
        sp: 0xffffffc080247d30,
```

```
        gp: 0x0,
        tp: 0x0,
        t0: 0x20,
        t1: 0xffffffc080202b38,
        t2: 0x40100000,
        s0: 0xffffffc0802001aa,
        s1: 0xffffffc080200488,
        a0: 0x40100000,
        a1: 0x22000000,
        a2: 0x28e428904300513,
        a3: 0x40100020,
        a4: 0x2,
        a5: 0xffffffc0802018be,
        a6: 0x20,
        a7: 0x22000000,
        s2: 0x1,
        s3: 0xffffffc080247db0,
        s4: 0xffffffc080247d48,
        s5: 0x3,
        s6: 0xffffffc080247d58,
        s7: 0x2,
        s8: 0x40100000,
        s9: 0x20,
        s10: 0x0,
        s11: 0x0,
        t3: 0x10,
        t4: 0xffffffc080203fe0,
        t5: 0x27,
        t6: 0x1,
    },
    sepc: 0xffffffc080202aa8,
    sstatus: 0x8000000200006100,
}
```

3. 现在正式切换地址空间。在拷贝 Image 到 0x4010_0000 的地址之前，切换到应用的地址空间。

   即在`const RUN_START: usize = 0x4010_0000;`代码行之前，先调用下面的两行：

```
    // switch aspace from kernel to app
    unsafe { init_app_page_table(); }
    unsafe { switch_app_aspace(); }
```

4. 执行`make run`

```
Load payload ...
load code [19, 5, 48, 4, 137, 66, 142, 2, 51, 131, 88, 0, 3, 51, 3, 0, 2,
147, 115, 0, 80, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]; address [0x22000000]
run code [19, 5, 48, 4, 137, 66, 142, 2, 51, 131, 88, 0, 3, 51, 3, 0, 2,
147, 115, 0, 80, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]; address [0x40100000]
Load payload ok!
Execute app ...
[ABI:Print] C
```

又能看到打印字符了，切换地址空间成功！

# 练习

## 练习 6:

1. 仿照 hello_app 再实现一个应用，唯一功能是打印字符 'D'。
2. 现在有两个应用，让它们分别有自己的地址空间。
3. 让 loader 顺序加载、执行这两个应用。这里有个问题，第一个应用打印后，不能进行无限循环之类的阻塞，想办法让控制权回到 loader，再由 loader 执行下一个应用。