输出页表信息

一、实验目的

了解 Linux 内核的用户空间和内核空间的概念，熟悉页表的格式，通过遍历内核空间的地址划分，打印内核虚拟地址空间的内存布局信息，加深对内存管理中页表的掌握。

二、实验内容

参考 arm64 的代码，修改内核源码，使得能够在 riscv 平台上输出内核虚拟内存的布局信息，包括各个划分的名称和起始地址。

- 下载 5.3.4 内核源码；
- 了解 riscv 的页表格式，修改相应的代码；
- 编译内核，并运行，得到输出结果。

三、实验指导

3.1 预备知识

虚拟地址空间：

RISC-V 提供了三种虚拟地址实现，Sv32，Sv39 以及 Sv48。

对于 32 位 RISC-V 而言，它仅能使用 Sv32，即 32 位虚拟地址空间，RISC-V Linux 把虚拟地址空间的高 1GB(0xc0000000-0xffffffff)被划分为内核地址空间，而把低 3GB (0x00000000-0xbfffffff)则划分为用户地址空间。对于 64 位 RISC-V 而言，它可以使用 Sv39 或 Sv48，对应的有效虚拟地址位数分别为 39 位和 48 位，低位有效。RISC-V Linux 默认使用的是 Sv39，有效虚拟地址空间同样被划分为两部分，0xffffff8000000000-0xffffffffffffffff 为内核地址空间，

0x0000000000000000-0x0000007fffffffff 为用户地址空间，其余地址均为无效虚拟地址。Linux 内核在启动时,使用的是物理地址，Linux 内核开启虚拟地址后，最终会将内核从加载物理地址映射到 PAGE_OFFSET，即虚拟地址与真实物理地址存在一个固定的 offset，其值为 PAGE_OFFSET(宏定义) - 加载物理地址，对于 RISC-V Linux 而言，这个值是 0xffffffe000000000 - 0x80000000。

Sv39 页表：

Sv39 的页表为 3 级页表，支持 page(4 KB), megapage (2 MB), gigapage (1 GB)，Sv39 页表项格式如下：

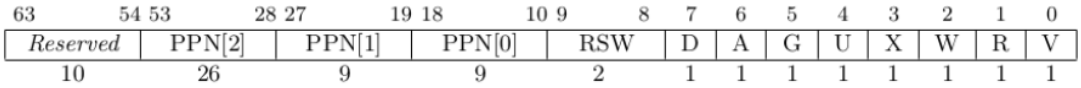

Figure 4.18: Sv39 page table entry.

最高 10 位作为保留位，恒为 0，PPN[2]，PPN[1]，PPN[0]分别为物理页索引，第 9-8 位作为保留位，提供给操作系统使用，低 8 位的意义如下：

V：有效位，1 为有效表项，0 为无效表项；

R：可读位，1 为可读，0 为不可读；

W：可写位，1 为可写，0 为不可写；

X：可执行位，1 为可执行，0 为不可执行；

U：用户位，保护位；

G：Global 位，表示该映射是否对所有虚拟地址空间有效；

A：Access 位，表示该表项是否被访问过；

D：Dirty 位，表示该页数据是否被修改过；

RWX 位的组合的特殊意义如下：

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

图 3.1 encoding of PTE R/W/X fields.

3.2 实验环境和环境设置

实验环境：

Ubuntu18.04、Linux 5.3.4 内核、qemu 模拟器。

```
# 下载 Linux 源码
wget https://mirror.bjtu.edu.cn/kernel/linux/kernel/v5.x/linux-5.3.4.tar.gz
# 安装 gcc-riscv64-linux-gnu
sudo apt install gcc-riscv64-linux-gnu
```

其余文件可从浙大云盘下载：

https://pan.zju.edu.cn/share/3c79ed6bfd4f366b38917dbb62

建议：参考 arch/arm64 下的代码架构，arm64 和 riscv 只有页表格式、内存布局不同，其他代码都可以参考。

3.3 添加头文件和修改 Makefile

（1）添加 arch/riscv/include/asm/ptdump.h。

复制 arch/arm64/include/asm/ptdump.h 的内容，并做如下修改：

- 修改 CONFIG_ARM64_PTDUMP_CORE 为 CONFIG_RISCV_PGDUMP。

- 修改 CONFIG_ARM64_PTDUMP_DEBUGFS 为 CONFIG_RISCV_PGDUMP。
- 删除 debug_wx 相关内容(line 31,line 34-38)。

（2）修改 arch/riscv/mm/Makefile 文件。

在结尾添加两行代码：

```
1.    obj-$(CONFIG_RISCV_PGDUMP) += ptdump_debugfs.o
2.    obj-$(CONFIG_RISCV_PGDUMP) += dump.o
```

（3）修改 arch/riscv/include/asm/pgtable-64.h。

修改 line21-27 如下：

```
1.    #if CONFIG_PGTABLE_LEVELS > 2
2.    typedef struct { pmdval_t pmd; } pmd_t;
3.    #define pmd_val(x)  ((x).pmd)
4.    #define __pmd(x)    ((pmd_t) { (x) } )
5.    #endif
6.
7.    #if CONFIG_PGTABLE_LEVELS > 3
8.    typedef struct { pudval_t pud; } pud_t;
9.    #define pud_val(x)  ((x).pud)
10.   #define __pud(x)    ((pud_t) { (x) } )
11.   #endif
12.
13.   #if CONFIG_PGTABLE_LEVELS == 2
14.   #define __ARCH_USE_5LEVEL_HACK
15.   #include <asm-generic/pgtable-nopmd.h>
16.   #elif CONFIG_PGTABLE_LEVELS == 3
17.   #define __ARCH_USE_5LEVEL_HACK
```

```
#include <asm-generic/pgtable-nopud.h>
#elif CONFIG_PGTABLE_LEVELS == 4
#include <asm-generic/5level-fixup.h>
#endif
```

（4）修改 arch/riscv/Kconfig.debug 文件。

插入如下代码：

```
config RISCV_PGDUMP
    def_bool y
    depends on DEBUG_KERNEL
    select DEBUG_FS
```

3.4 添加 C 文件

（1）添加 arch/riscv/mm/ptdump_debugfs.c。

复制 arch/arm64/mm/ptdump_debugfs.c 内容即可。

（2）添加 arch/riscv/mm/dump.c。

复制 arch/arm64/mm/dump.c,并对其做简单修改。

- 修改包含的头文件。

```
#include <asm/page.h>
#include <asm/pgtable-bits.h>
#include <asm/pgtable-64.h>
#include <asm/pgtable.h>
#include <asm/ptdump.h>

#include <linux/types.h>
#include <linux/kernel.h>
```

- 修改 struct pg_state。

```
1.    struct pg_state {
2.        struct seq_file *seq;
3.        const struct addr_marker *marker;
4.        unsigned long start_address;
5.        unsigned level;
6.        u64 current_prot;
7.    };
```

- 删除函数 note_prot_uxn, note_prot_wx，ptdump_check_wx 的定义以及调用。
- 修改 walk_pgd 函数中第一行为 unsigned long end = 0;。
- 判断 pmd_sect 和 pud_sect 的功能，添加对 pmd_sect 和 pud_sect 的宏定义。
- 结构数组 pte_bits 的功能是定义 page table entry 的低位，请根据 riscv PTE 的格式,修改 pte_bits 的值。
- 结构数组 address_markers 的功能是定义 riscv 内核地址布局，请根据 riscv 内核地址布局修改其内容，地址布局可以从 arch/riscv/include/asm/pgtable.h 中找到。
- 完成对 VA_START 的定义(内核虚拟地址空间起始地址)。

3.5 编译运行

（1）编译。

```
1.    $ cd linux-5.3.4/
2.    $ make ARCH=riscv defconfig
3.    $ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -
j$(nproc)
```

```
$ export PATH=path/to/riscv-unknown-linux-gnu-gcc9.2/bin:$PATH
$ cd ../riscv-pk/build && rm -rf *
$ ../configure --enable-logo --host=riscv64-unknown-elf
    --with-payload=../../linux-5.3.4/vmlinux
$ make -j$(nproc)
```

（2）运行。

```
$ cd ../../
$ ./qemu-system-riscv64 \
    -nographic -machine virt \
    -kernel riscv-pk/build/bbl \
    -append "root=/dev/vda ro console=ttyS0" \
    -drive file=busybear.bin,format=raw,id=hd0 \
    -device virtio-blk-device,drive=hd0
```

用户名：root，密码：busybear。可能会出现 ntpd: bad address，这是正常情况，忽略即可，结果如下：

（3）打印页表。

```
$ mount -t debugfs nodev /sys/kernel/debug
$ cat /sys/kernel/debug/kernel_page_tables
```

图 3.2 运行结果

四、实验提交

（1）需上传测试结果的截图，并上传编写过或修改过的全部代码。

（2）x86 和 arm64 代码中已经提供了相应函数，尝试直接利用函数的功能，输出页表的相关信息，并分析页表信息。

（3）分析 walk_pte、walk_pmd、walk_pud、walk_pgd 之间的关系。

（4）简单谈谈你在实验中遇到的困难。