实验报告

一、基本知识

页式存储管理是一种物理内存管理，它是非连续分配内存的。

sv39是RISCV的一种页表机制，它规定：

- 每个页的大小是4KB，也就是4096个字节
- 定义物理地址(Physical Address)有56位，而虚拟地址(Virtual Address)有39位

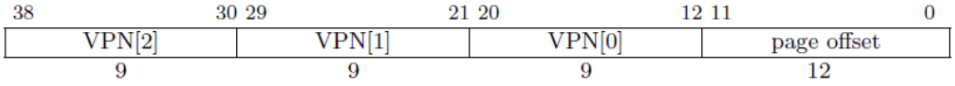


Figure 4.16: Sv39 virtual address.

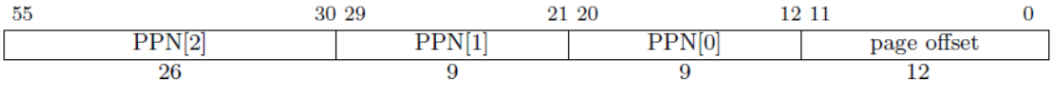


Figure 4.17: Sv39 physical address.

本次实验，我们使用的就是sv39页表机制。需要注意的是，其实一个虚拟地址要占用 64 位。不过，只有低 39 位有效，我们规定 63–39 位的值必须等于第 38 位的值，否则会认为该虚拟地址不合法，在访问时会产生异常。

1. sv39页表机制

sv39规定一个页表项占8字节（64位），页表项结构如下

63-54	53-28	27-19	18-10	9-8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

- PPN：表示这个虚拟页号映射到的物理页号。
- 低10位：描述映射的状态信息。
  - RSW：两位留给 S Mode 的应用程序。
  - D(Dirty)：如果 D=1 表示自从上次 D 被清零后，有虚拟地址通过这个页表项进行写入。
  - A(Accessed)：如果 A=1 表示自从上次 A 被清零后，有虚拟地址通过这个页表项进行读、或者写、或者取指。
  - G(Global)：如果 G=1 表示这个页表项是”全局"的，也就是所有的地址空间（所有的页表）都包含这一项
  - U(user)：U为 1 表示用户态 (U Mode)的程序 可以通过该页表项进映射。在用户态运行时也只能够通过 U=1 的页表项进行虚实地址映射。 注意，S Mode 不一定可以通过 U=1 的页表项进行映射。我们需要将 S Mode 的状态寄存器 sstatus 上的 SUM 位手动设置为 1 才可以做到这一点（通常情况不会把它置1）。否则通过 U=1 的页表项进行映射也会报出异常。另外，不论sstatus的SUM位如何取值，S Mode都不允许执行 U=1 的页面里包含的指令，这是出于安全的考虑。

- R(Readable), W(Writable), X(Executable)；为许可位，分别表示是否可读，可写，可执行。
- V：表示这个页表项是否合法。如果为 0 表示不合法，此时页表项其他位的值都会被忽略。

其中，R,W,X 不同的取值对应不同的含义

X	W	R	Meaning
0	0	0	指向下一级页表的指针
0	0	1	这一页可读（只读）
0	1	0	保留（未来使用）
0	1	1	这一页可读可写（不可执行）
1	0	0	这一页可执行（只执行）
1	0	1	这一页可读可执行
1	1	0	保留（未来使用）
1	1	1	这一页可读可写可执行

所以，只要RWX这三位不同时为0，这个页表项就是直接映射物理地址的。

2. 三级页表

首先，计算为什么每一层页表仅有$2^9=512$个页表项。因为一些我似懂非懂的原因，存储一页需要使用4KB的空间。而一个页表项就占8B的空间，于是便有计算公式

$$\frac{4K}{8} = 512$$

其次，约定称最接近物理页面的页表为第一级页表，最远离物理页面的页表为第三级页表。

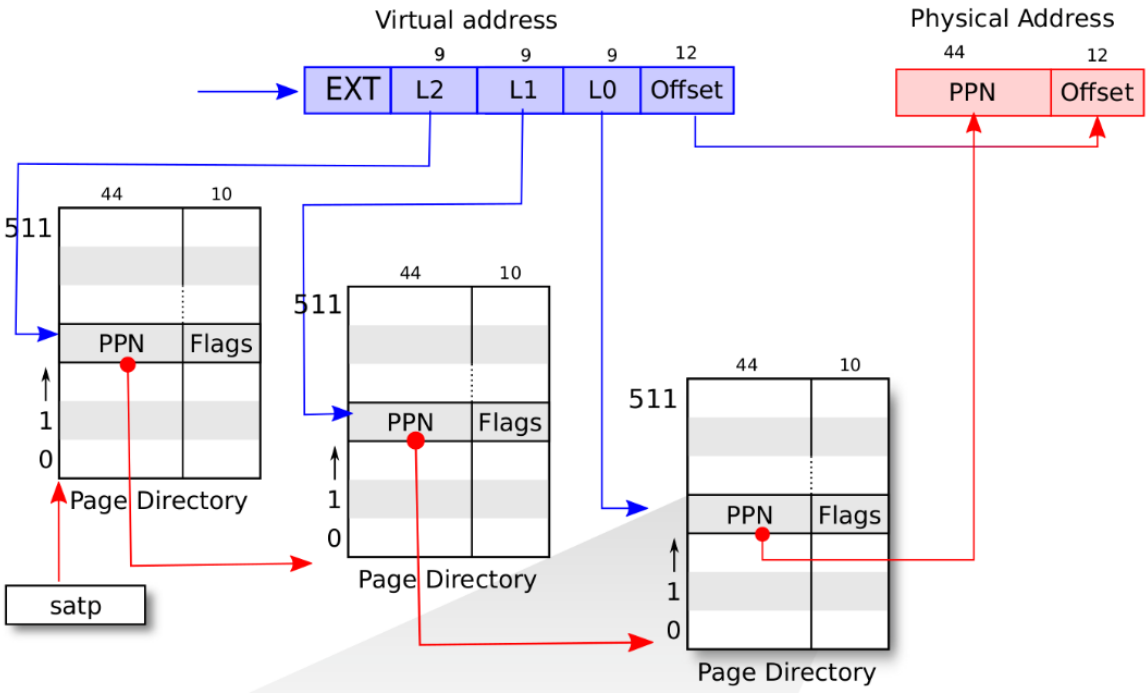
于是，就有：

- 三级页表有512个三级页表项
  - 1个三级页表项可以指向1个二级页表
  - 1个三级页表项可以控制2^{18}个虚拟页号（对应1GB虚拟内存）
- 二级页表有512个二级页表项
  - 1个二级页表项可以指向1个一级页表
  - 1个三级页表项可以控制2^{9}个虚拟页号（对应2MB虚拟内存）
- 一级页表有512个一级页表项
  - 1个一级页表项控制1个虚拟页号（对应4KB虚拟内存）

因此有以下两个说法：

- 二级页表项的 R,W,X 设置为不是全 0 的许可要求时候，映射一个 2MB 的大页 (Mega Page)
- 三级页表项的 R,W,X 设置为不是全 0 的许可要求时候，映射一个 1GB 的大页 (Giga Page)

下图演示使用三级页表时，如何将逻辑地址转成物理地址



3. 页表基址

RISCV架构，一个名为satp（Supervisor Address Translation and Protection Register）的CSR，存放着最高级页表的所在的物理页号。satp的结构如下，

63-60	59-44	43-0
MODE(WARL)	ASID(WARL)	PPN(WARL)
4	16	44

- MODE 控制 CPU 使用哪种页表实现。
  - 将 MODE 设置为 `0100` 即表示 CPU 使用 Sv39 。
  - 将 MODE 设置为 `0000` 即表示 CPU 不使用页表，直接使用物理地址。
- （目前无用）ASID 其他编码保留备用的address space identifier。OS 可以在内存中为不同的应用分别建立不同虚实映射的页表，并通过修改寄存器 satp 的值指向不同的页表，从而可以修改 CPU 虚实地址映射关系及内存保护的行为。
- PPN 存的是三级页表所在的物理页号。

4. 快表

使用三级页表时，将一个虚拟地址转化为物理地址需要访问 3 次物理内存，得到物理地址之后还要再访问一次物理内存。从逻辑上分析，这降低了内存访问的效率（相比原来“不使用页表，直接使用物理地址”而言）。

实践表明虚拟地址的访问具有时间局部性和空间局部性。

- 时间局部性是指，被访问过一次的地址很有可能 **不远的将来** 再次被访问；
- 空间局部性是指，如果一个地址被访问，则这个地址 **附近的地址** 很有可能在不远的将来被访问。

基于此，在 CPU 内部，使用快表 (TLB, Translation Lookaside Buffer) 来记录近期已完成的虚拟页号到物理页号的映射。

刷新快表

我们如果修改了 satp 寄存器，比如将上面的 PPN 字段进行了修改，说明我们切换到了一个与先前映射方式完全不同的页表。此时快表里面存储的映射结果就跟不上时代了，很可能是错误的。这种情况下我们要使用 `sfence.vma` 指令刷新整个 TLB 。

同样，我们手动修改一个页表项之后，也修改了映射，但 TLB 并不会自动刷新，我们也需要使用 `sfence.vma` 指令刷新 TLB 。

如果不加参数的，`sfence.vma` 会刷新整个 TLB 。你可以在后面加上一个虚拟地址，这样 `sfence.vma` 只会刷新这个虚拟地址的映射。

5. （补充）外设与物理内存

`物理内存` 通常是一片 RAM ，我们可以把它看成一个以字节为单位的大数组，通过物理地址找到对应的位置进行读写。

但是，`物理地址` 并不仅仅只能访问物理内存，也可以用来访问其他的外设，因此你也可以认为物理内存也算是一种外设。

这样设计是因为：如果访问其他外设要使用不同的指令（如 x86 单独提供了in, out 指令来访问不同于内存的IO地址空间），会比较麻烦，于是很多 CPU（如 RISC-V，ARM，MIPS 等）通过 `MMIO(Memory Mapped I/O)` 技术将外设 `映射` 到一段物理地址，这样我们访问其他外设就和访问物理内存一样啦！

本次实验先不管那些外设，目前只关注物理内存。

6. （补充）物理内存探测

在 RISC-V 中，`OpenSBI` 会包括物理内存在内的各外设的扫描，将扫描结果以 `DTB` (Device Tree Blob)的格式保存在物理内存中的某个地方（这个扫描结果描述了所有外设的信息，当中也包括 Qemu 模拟的 RISC-V 计算机中的物理内存）。随后 OpenSBI 会将其地址保存在 a1 寄存器中，供操作系统使用。

不过本次实验不打算解析这个结果。因为我们已经知道，Qemu 规定的 DRAM 物理内存的起始物理地址为 `0x80000000` 。而在 Qemu 中，可以使用 -m 指定 RAM 的大小，默认是 128MiB 。因此，默认的 DRAM 物理内存地址范围就是 `[0x80000000,0x88000000)` 。因此，本次实验直接将 DRAM 物理内存结束地址硬编码到内核中。

<!-- 备注：

$$ -->

二、实验内容

1. 修改链接脚本的起始地址

(1) 具体操作

在文件 `tools/kernel.ld` 中，将变量 `BASE_ADDRESS` 的值从原来的 `0x80200000` 修改为 `0xFFFFFFFFC0200000`

(2) 解释

改动了链接脚本的起始地址后，编译器和链接器会把内核代码里面的符号/变量地址都对应到`0xffffffffc0200000`之后的某个地址上。从此，内核代码会认为自己处在以虚拟地址`0xffffffffc0200000`开头的一段连续虚拟地址空间中。

（But，启动qemu后，内核代码仍然放在以`0x80200000`开头的一块连续物理内存中）

2. 内核初始映射

这一步的主要操作在文件entry.S中展开，故见md文档阅读entry.S

3. 物理内存管理初始化

在文件`lab2/kern/init/init.c`中，在函数`kern_init`的body中添加函数调用`pmm_init()`（当然，需要添加相应的头文件）

然后，主要操作在文件pmm.c中展开，故见md文档阅读pmm.c。

4. 一些准备工作

在lab2中，增加了一些功能，接下来的实验操作

- kern/sync/sync.h：为确保内存管理修改相关数据时不被中断打断，提供以下两个功能
  - 一个是保存 sstatus寄存器中的中断使能位(SIE)信息并屏蔽中断的功能
  - 另一个是根据保存的中断使能位信息来使能中断的功能
- libs/list.h：定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理（以及其他内核功能）的基础。其他有类似双向链表需求的内核功能模块可直接使用 list.h 中定义的函数。
- libs/atomic.h：定义了对一个二进制位进行读写的原子操作，确保相关操作不被中断打断。包括
  - set_bit()设置某个二进制位的值为1，
  - change_bit()给某个二进制位取反
  - test_bit()返回某个二进制位的值。

见md文档阅读sync.h。

见md文档阅读list.h。

5. 定义Page结构体

Page结构体是页块（多个页框）的代码实现，详情请见md文档阅读memlayout.h。

6. 物理内存管理结构体的初始化

见md文档阅读pmm.h。

见md文档阅读pmm.c。

7. default_pmm_manager结构体的实现

见md文档阅读default-pmm.c。

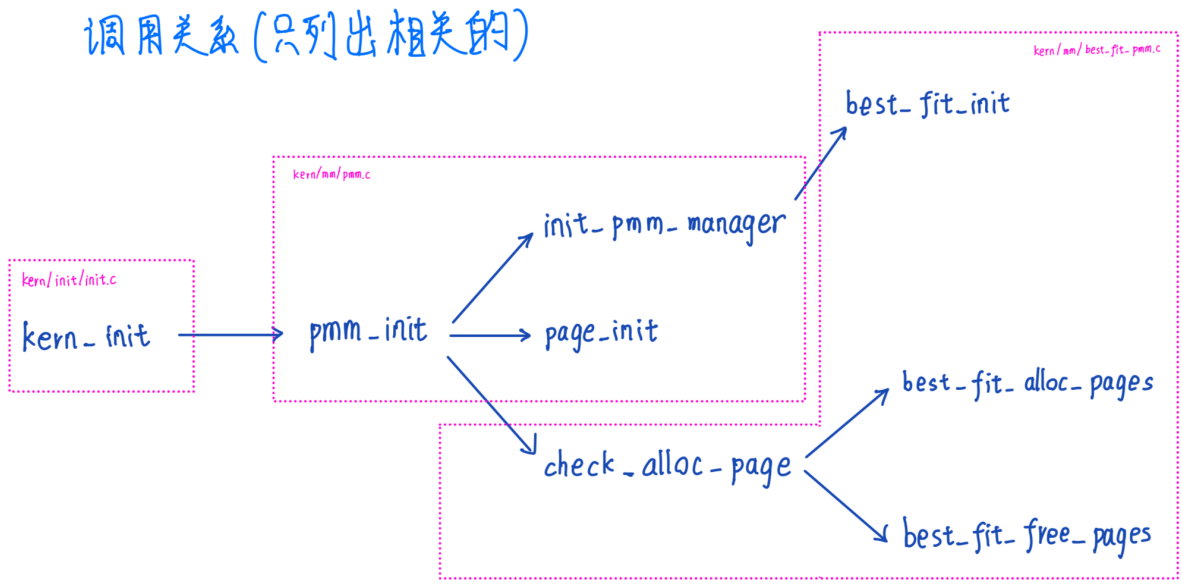
8. default_pmm_manager结构体的实现

见md文档阅读best_fit_pmm.c。三、练习与问答

练习1：理解first-fit 连续物理内存分配算法（思考题）

first-fit 连续物理内存分配算法作为物理内存分配一个很基础的方法，需要同学们理解它的实现过程。请大家仔细阅读实验手册的教程并结合kern/mm/default_pmm.c中的相关代码，认真分析default_init，default_init_memmap，default_alloc_pages， default_free_pages等相关函数，并描述程序在进行物理内存分配的过程以及各个函数的作用。 请在实验报告中简要说明你的设计实现过程。并回答问题：你的first fit算法是否有进一步的改进空间?

问题1：描述物理内存分配的过程



- kern/init/entry.S
  - 符号boot_page_table_sv39建立了一张三级页表
  - 符号kern_init设置satp的内容，刷新快表，设置sp，跳转函数kern_init
- kern/init/init.c
  - 函数kern_init调用函数pmm_init
- kern/mm/pmm.c
  - 函数pmm_init依次调用函数init_pmm_manager、page_init、check_alloc_page
  - 函数init_pmm_manager选择best_fit_pmm_manager作为物理内存管理器，调用函数best_fit_init
- kern/mm/best_fit_pmm.c
  - 函数best_fit_init完成物理内存管理器的初始化
- kern/mm/pmm.c
  - 函数page_init完成检测物理内存空间，保留已经使用的内存并创建未分配内存链表，调用函数best_fit_init_memmap
- kern/mm/best_fit_pmm.c

    - 函数best_fit_init利用输入的未分配内存链表初始化一个空闲块节点
- kern/mm/pmm.c
    - 函数check_alloc_page测试物理内存分配是否正确，期间会反复调用函数best_fit_alloc_pages和函数best_fit_free_pages
- kern/mm/best_fit_pmm.c
    - 函数best_fit_alloc_pages分配物理内存
    - 函数best_fit_free_pages回收物理内存

问题2：default_init，default_init_memmap，default_alloc_pages，default_free_pages等函数的作用

- default_init
    - 初始化物理内存管理器（包括初始化链表指针、空闲页数设为0）
- default_init_memmap
    - 初始化一个空闲块节点
- default_alloc_pages
    - 以first-fit的方式分配物理内存
- default_free_pages
    - 回收物理内存

问题3：你的first fit算法是否有进一步的改进空间?

~~放屁！这个first fit算法压根就不是我的~~没有想法

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

在完成练习一后，参考kern/mm/default_pmm.c对First Fit算法的实现，编程实现Best Fit页面分配算法，算法的时空复杂度不做要求，能通过测试即可。 请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放，并回答问题：你的 Best-Fit 算法是否有进一步的改进空间?

编程1：实现Best Fit页面分配算法

见md文档阅读default-pmm.c。

问题1：阐述代码是如何对物理内存进行分配和释放

- 分配物理内存
    - 检查（空闲页框是否足够）
    - 遍历空闲链表，查找满足需求的空闲页框
    - 如果找到满足需求的页面，记录该页面（但不会停止遍历）
    - 如果再次找到满足需求的页面，比该页面与记录页面的连续空闲页框数量，记录较小者的页面（但不会停止遍历）
    - 遍历结束后，根据记录的页分配物理内存，将多余的空闲页框放回空闲链表
- 释放物理内存
    - 检查（这些页框是否都是被分配出去的、非os保留的）

- 设置当前页块的属性为释放的页块数、并将当前页块标记为已分配状态、最后增加nr_free的值
- 将该页块放入空闲链表
- 承上启下（也就是合并）

问题2：你的 Best-Fit 算法是否有进一步的改进空间?

- 分配物理内存
  - 可以提前跳出遍历，如果找到大小刚刚好的空闲页块
- 释放物理内存
  - 可以在先考虑能否合并，在考虑将回收的页块放入空闲链表

建立段页式管理中需要考虑的关键问题

问题1：对于哪些物理内存空间需要建立页映射关系?

先回忆物理地址空间的使用情况：

- 默认的 DRAM 物理内存地址范围是 `[0x80000000,0x88000000)`。
- 物理地址空间 `[0x80000000,0x80200000)` 被 OpenSBI 占用。
- 物理地址空间 `[0x80200000,KernelEnd)` 被内核各代码与数据段占用。

因此，物理地址空间`[KernelEnd,0x88000000)`是空闲可分配的；物理地址空间`[0x80000000,KernelEnd)`是os保留的。

本次实验中，我们对物理地址区间 `[0x80000000,0x88000000)` 都建立页映射关系。

问题2：具体的页映射关系是什么?

我们通过一个大页将虚拟地址区间[0xffffffffc0000000,0xffffffffffffffff] 映射到物理地址区间[0x80000000,0xc0000000)

问题3：页目录表的起始地址设置在哪里?

在os代码中，它被符号boot_page_table_sv39记录着。这个地址是虚拟地址（也就是在地址0xffffffffc0000000之后），在编写os代码时未知。

问题4：页表的起始地址设置在哪里，需要多大空间?

页表的起始地址在物理地址0x80000000处，需要4KiB。

问题5：如何设置页目录表项的内容?

将立即数按序存放进os代码中，如下代码所示：

```
    .zero 8 * 511
    .quad (0x80000 << 10) | 0xcf
```

（注意12位对齐）

问题6：如何设置页表项的内容?

也许和设置页目录表项差不多，不会（这次实验也没做）

（不过现在还没搞虚拟内存，所以不知道如何在分配物理内存后如何更新页表内容）