Lab10 虚拟内存和页表

一、 实验概述

在本次实验中，我们将内核空间放在虚拟地址下，并使用页表将物理地址和虚拟地址对应。

二、实验目的

1. 了解riscv中关于内存管理的知识
2. 掌握内存地址的转换机制
3. 掌握页表的建立和使用方法

三、实验项目整体框架概述

```
// Lab10
├── kern
│   ├── debug
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   ├── console.c
│   │   ├── consh
│   │   ├── ide.c
│   │   ├── ide.h
│   │   ├── intr.c
│   │   └── intr.h
│   ├── fs
│   │   ├── fs.h
│   │   ├── swapfs.c
│   │   └── swapfs.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   ├── mm
│   │   ├── default_pmm.c
│   │   ├── default_pmm.h
│   │   ├── memlayout.h //进行了更新
│   │   ├── mmu.h
│   │   ├── pmm.c
│   │   ├── pmm.h
│   ├── sync
│   │   └── sync.h
│   └── trap
├── libs
├── Makefile
└── tools
```

四、实验内容

1. 理解页表、页表项、多级页表、大页、页表基址的概念和原理
2. 了解块表的概念和原理
3. 掌握内核内存空间虚拟地址与物理地址的映射方式

五、基础知识

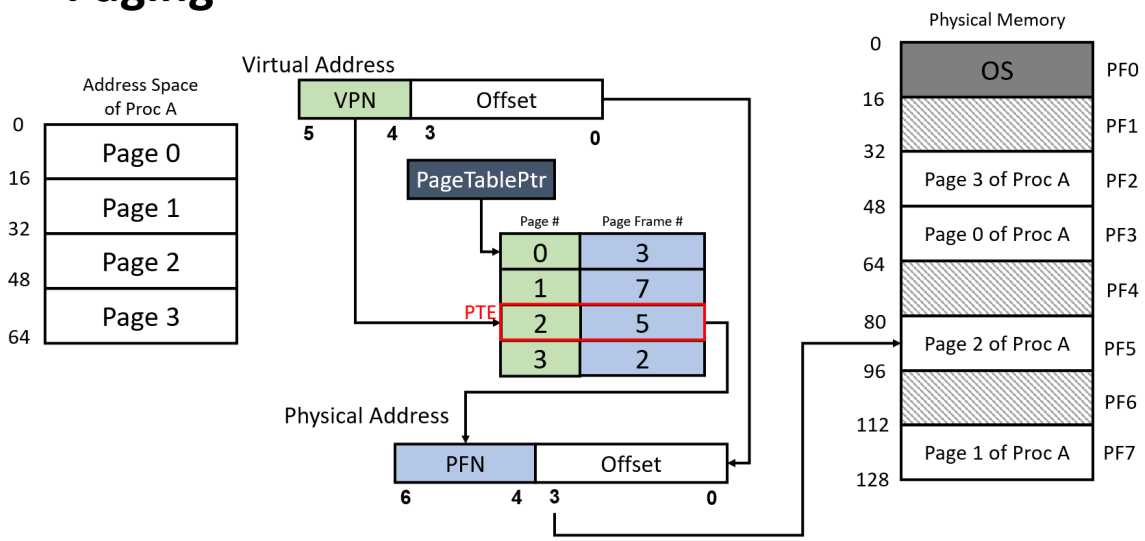
1. Paging

分页机制，是将程序空间（虚拟地址）切割成相同大小的若干个页面，同时将物理内存也切割成同样大小的多个页面，从而可以在物理地址不连续的情况下，给进程分配足够的内存空间，并且从虚拟地址的角度看这个空间是连续的。

> ### 为什么需要物理内存管理
>
> 如果我们只有物理内存空间，那么我们也可以写程序，但是所有的程序，包括内核，包括用户程序，都在同一个地址空间里，用户程序访问的`0x80200000`和内核访问的`0x80200000`是同一个地址。这样好不好？如果只有一个程序在运行，那也无所谓。但很多程序使用同一个内存空间，就会有问题：怎样防止程序之间互相干扰，甚至互相搞破坏？比较粗暴的方式就是，我让用户程序访问的`0x80200000`和内核访问的`0x80200000`不是一个地址。但是我们只有一块内存，为了创造两个不同的地址空间，我们可以引入一个"翻译"机制：程序使用的地址（虚拟地址）需要经过一步"翻译"才能变成真正的内存的物理地址。这个"翻译"过程，我们用一个"词典"（页表）实现---给出翻译之前的地址，可以在词典里查找翻译后的地址。每个程序都有唯一的一本"词典"，而它能使用的内存也就只有他的"词典"所包含的。
>
> "词典"是否对能使用的每个字节都进行翻译？我们可以想象，存储每个字节翻译的结果至少需要一个字节，那么使用1MB的内存将至少需要构造1MB的"词典"，这效率太低了。观察到，一个程序使用内存的数量级通常远大于字节，至少以KB为单位（所以上古时代的人说的是"640K对每个人都够了"而不是"640B对每个人都够了"）。那么我们可以考虑，把连续的很多字节合在一起翻译，让他们翻译前后的数值之差相同，这就是"页"。

分页机制很重要的一点是如何建立和解析虚拟地址到物理地址的映射，下面我们从"如何从虚拟地址获得相应的物理地址"的角度进行介绍：

如图所示是一个一级页表分页机制（一级对应于后面的多级，一级页表只需要查询一层页表即可得到物理地址）：



以上图为例，我们首先得到一个**虚拟地址（Virtual Address）**，这个地址长度为6位，其中5~4位（高2位）为**页号（VPN，Virtual Page Number）**，3~0位（低4位）为**偏移量（Offset）**。

通过虚拟地址，我们可以查询**页表（Page Table）**，页表存在于**页表基地址（PageTablePtr，Page Table Pointer**，是个物理地址）所指向的内存空间中，由连续存储的若干个**页表项（PTE，Page Table Entry）**构成。在一级页表中，每个页表项内容即为**物理页号（Page Frame #）+部分标志位。**尽管图中每个页表项看似包含**页号（Page #）**但是在实际的设计中，**页号并不写在页表项中**，由于页表项是连续分布的，我们只需要知道页表项的大小（有多少位）以及虚拟地址页号（VPN，代表要查询第几个页表项），就可以通过`页表首地址+页号×页表项大小`得到对应的页表项地址，查询该地址对应内容即可得到物理页号。页表首地址是存储在架构指定的寄存器中的。

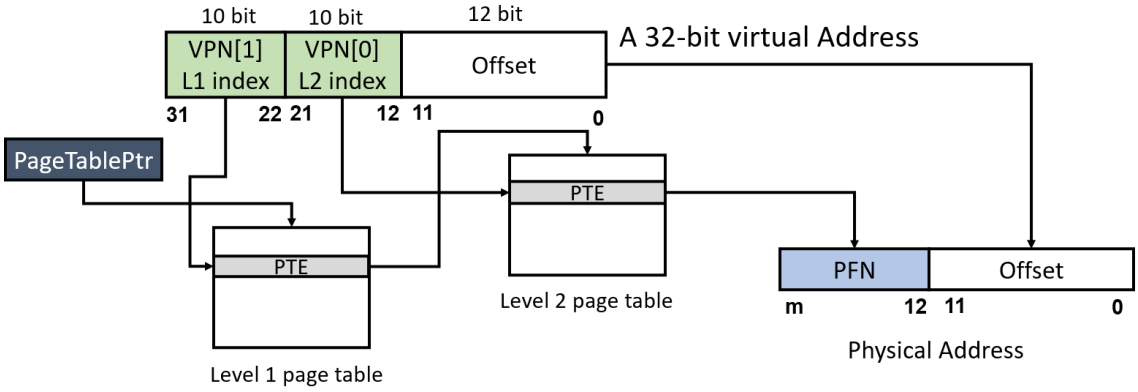
得到物理页号后，通过`物理页号×页面大小`即可得到所在页的物理地址（物理空间首地址为0x0）。一页可能很大，如何得到一页中具体某个字节的地址呢？通过偏移量，`页物理地址+偏移量`即可得到具体虚拟地址对应的具体物理地址。

> 例：
> 假设页面大小为4KB，如果想要定位到页面的每一个Byte，我们的偏移量则需能表示4096个不同的位置，因此对于4KB大小的页面来说，偏移量的位数为12位（2^{12}=4096）。
>
> 在图片的一级页表中，Offset为4位，可以得知页面大小为2^4=16B。如果给定虚拟地址100100_2，可以得出其页号为10_2，即2_{10}，通过页号2_{10}可以查到对应的页表项内容为5_{10}。5是物理页号，可以求得物理页面地址： 物理首地址+物理页号×页面大小=0+5×16=80_{10}。100100中低四位偏移量为0100，因此查询的是该页面的第5个字节对应的地址（地址0也是一个字节），则该物理地址为80+4=84_{10}（5*16+4=0101左移四位+0100=0101 0100_2）。需要注意运算过程中是2进制还是10进制还是16进制数（存储肯定是2进制）。

2. 多级页表

系统中，通常页面大小是4KB，假设我们物理空间是4GB，那么页表中就需要1M个页表项来对应不同的物理页，假设一个页表项为4B，则一个页表就有4MB。由于虚拟地址是连续的（相当于高位VPN是连续的），而PTE的存储方式也是相当于数组的连续存储方式，因此即使进程实际使用的空间非常小，它也需要连续完整的页表来进行地址转换（不能移除中间不使用的PTE）。而操作系统中除了内核页表，为每个进程还会分配自己的页表，进程多的情况下，存储所有页表的开销就变得很大。

因此需要用到多级页表，如图所示，是一个二级页表分页机制：



与一级页表不同的是，二级页表将虚拟地址分割成了三个部分，其中偏移量位数依然与页面大小相关，而VPN被切割成了两个部分VPN[1]和VPN[0]。VPN[1]代表第一级页表的页号，VPN[0]代表第二级页表的页号。

图中32位虚拟地址的翻译过程变成了下面这个流程：

> 1．取得虚拟地址高10位得到一级页号，通过`第一级页表基地址+页号×PTE大小`得到第一级页表项地址（第一级页表基地址存储于指定寄存器中）
> 2．通过第一级页表项地址，取得其内容即第二级页表的物理页号，通过`0+第二级页表物理页号×页面大小`得到第二级页表基地址
> 3．取得虚拟地址中间的10位VPN[0]即二级页号，通过`第二级页表基地址+二级页号×PTE大小`可以得到第二级页表项的地址
> 4．从第二级页表项地址所在空间可以得到整个虚拟地址对应的物理页号，通过`物理页号×页面大小+偏移量`可以得到最终的物理地址

那么二级页表相对于一级页表（这里指分页机制，并非Level 1 page table)为什么可以节省页面开销呢?

思考: 一个4GB的内存空间，页面大小4KB，设每个PTE大小为4Byte，有一个进程只需要使用高虚拟地址空间的1页4KB空间和低虚拟地址空间的1页4KB空间。在一级页表和上图所示的二级页表机制中，这个进程分别需要至少多大的页表空间?

一级页表中，需要`4GB÷4KB=1M`个PTE映射虚拟地址最低位到最高位的空间，因此页表大小为`1M×4Byte=4MB`。

二级页表中，第二级页号VPN[0]为10个bit，因此可以表示2^{10}个不同的页面，每个页面大小为4KB，故一个二级页表可以指向2^{10}×4KB=4MB的物理空间。第一级页号VPN[1]同样为10个bit可以指向2^{10}个不同的二级页表，因此图中的二级页表机制可以指向2^{10}×4MB=4GB的物理空间。一个进程需要使用高虚拟地址空间的1页4KB空间和低虚拟地址空间的1页4KB空间，可以知道这两部分空间属于不同的第二级页表指向的空间。那么，我们首先需要预留两个第二级页表。除此之外我们还需要一个第一级页表存储这两个二级页表的PTE，虽然第一级页表中指向第二级页表的PTE也是连续存储的（与一级页表分页机制相同），但是第一级页表的大小已经变了，第一级页表有2^{10}个PTE，因此大小为2^{10}×4B=4KB远小于一级页表分页机制中的（第一级）页表大小。综上，我们需要2个二级页表+1个一级页表（其余2^{10}-2个二级页表都用不上，直接在一级页表对应的PTE中设置为不可用即可，不需要分配页表空间），由于VPN位数相同，因此一级页表和二级页表大小都是4KB，故总共需要12KB大小的页表空间。

3. sv39分页方案

前面提到有一个指定的寄存器用于存储页表的基地址，从而虚拟地址可以根据这个基地址找到对应的页表，在riscv架构中，这个寄存器叫`satp`（RISCV手册P108）。

在RV64中，satp是如下图所示的64位寄存器：

其中，高四位`MODE`的值对应的含义为：

RV64		
Value	Name	Description
0	Bare	No translation or protection.
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.

上次实验中，我们未设置satp的值，MODE默认为0，因此系统使用物理地址进行操作。本次实验中，我们会把MODE的值设置为8也就是启用`Sv39`分页方案。

satp中低44位的**PPN（Physical Page Number）**存储了根页表的基址，PPN乘以页面大小就是根页表的物理地址`satp.PPN×4096`，也就是前面提到的Page Table Pointer。

在 Sv39 中，定义**物理地址(Physical Address)**有 56位，而**虚拟地址(Virtual Address)** 有 39位。实际使用的时候，一个虚拟地址要占用 64位，只有低 39位有效，规定 63–39 位的值必须等于第 38 位的值（类似有符号整数），否则会认为该虚拟地址不合法，在访问时会产生异常。

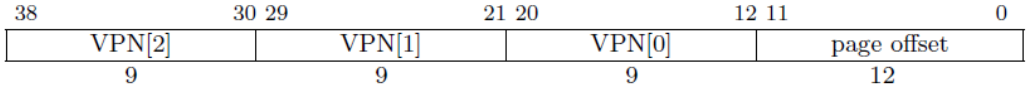


Figure 4.16: Sv39 virtual address.

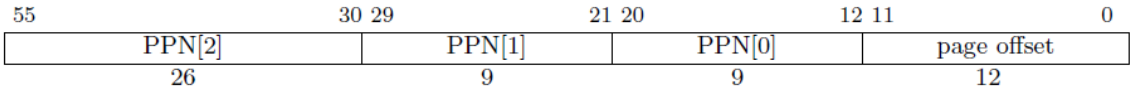


Figure 4.17: Sv39 physical address.

上图所示为Sv39的虚拟地址和物理地址格式，该方案为三级页表。下图为Sv32的32位虚拟地址的转换过程，Sv39的转换过程几乎和Sv32相同，区别在于Sv39的页表项位数更多并且多了一级页表。

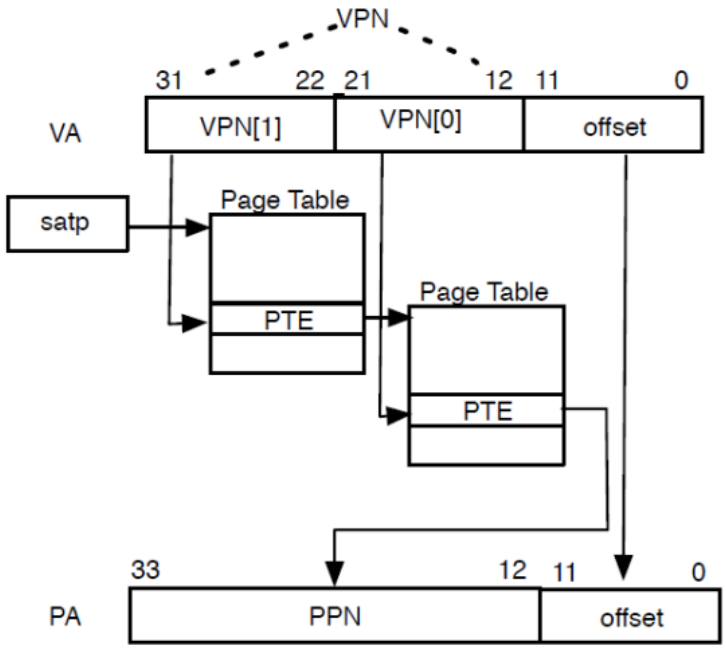


图 10.14：Sv32 中地址转换过程的图示。

当在 satp 寄存器中启用了分页时，S 模式和 U 模式中的虚拟地址会以从根部遍历页表的方式转换为物理地址。图 10.14 描述了这个过程：

1. satp.PPN 给出了一级页表的基址，VA[31:22]给出了一级页号，因此处理器会读取位于地址$(\text{satp.PPN} \times 4096 + \text{VA}[31:22] \times 4)$的页表项。
2. 该 PTE 包含二级页表的基址，VA[21:12]给出了二级页号，因此处理器读取位于地址$(\text{PTE.PPN} \times 4096 + \text{VA}[21:12] \times 4)$的叶节点页表项。
3. 叶节点页表项的 PPN 字段和页内偏移（原始虚址的最低 12 个有效位）组成了最终结果：物理地址就是$(\text{LeafPTE.PPN} \times 4096 + \text{VA}[11:0])$

4. 吉页

下图是RV64中Sv39的页表项（PTE）

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

我们可以看到 Sv39 里面的一个页表项大小为64位 8 字节。其中第 53–10 共 44 位为一个物理页号（分成三个部分是方便中文手册P112的算法描述，表示这个虚拟页号映射到的物理页号。后面的第 9–0 位则描述映射的状态信息。

- RSW两位留给 S Mode 的应用程序，我们可以用来进行拓展。
- D，即 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，即 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，即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)为 1 表示用户态 (U Mode)的程序 可以通过该页表项进行映射。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。

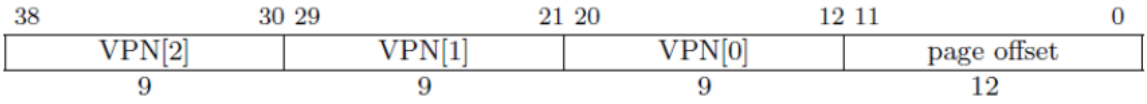
- R,W,X为许可位，分别表示是否可读 (Readable)，可写 (Writable)，可执行 (Executable)。
- V表示这个页表项是否合法。如果为 000 表示不合法，此时页表项其他位的值都会被忽略。
- 以 W这一位为例，如果 W=0 表示不可写，那么如果一条 store 的指令，它通过这个页表项完成了虚拟页号到物理页号的映射，找到了物理地址。但是仍然会报出异常，是因为这个页表项规定如果物理地址是通过它映射得到的，那么不准写入！R,X 也是同样的道理。

根据 R,W,X取值的不同，我们可以分成下面几种类型：

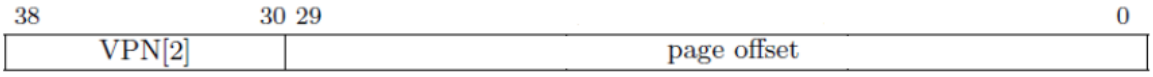
X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

可以注意到当XWR取值均为0时，该PTE是指向下一级页表的指针，而当这XWR中存在非0位时，则该PTE是页表树的一个叶节点，也就是此时该PTE的PPN值代表了物理地址的基址而非页表的物理基址。

比如，Sv39中给定一个虚拟地址VA[38:0]，然后我们根据satp.PPN×4096+VA[38:30]×8求得一级页表的页表项PTE的地址，假设该PTE的XWR为001，则该PTE为叶节点页表项。此时由于一级页表的页表项就是叶节点页表项，虚拟地址的含义就从原本的



变成了



而物理地址的计算则变成了PTE.PPN×4096+VA[29:0]。此时可以发现30位偏移量可以表示从PTE.PPN×4096开始的一个很大的空间（空间大小为2^{30}Byte即1GiB），我们称之为**吉页**，本次实验的内核虚拟地址映射中会应用到它。（注意常见支持大页机制的架构中大页基地址必须对齐自己的大页空间，即吉页基地址必须对齐1GB，PTE.PPN*4096的低30位需要为0。）

相应的，如果第二级页表的页表项为叶节点页表项，则该页表项的[20:0]位将表示一个整体的偏移量，代表一个小于吉页的页，我们称之为**巨页**。一般情况下三级页表项是叶节点页表项，指向一个**基页**（4KB）。

5. 快表

物理内存的访问速度要比 CPU 的运行速度慢很多，去访问一次物理内存可能需要几百个时钟周期（带来所谓的“冯诺依曼瓶颈”）。如果我们按照页表机制一步步走，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，才能读到我们想要的数据。这很大程度上降低了效率。

好在，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。

因此，在 CPU 内部，我们使用**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。由于局部性，当我们要做一个映射时，会有很大可能这个映射在近期被完成过，所以我们可以先到 TLB 里面去查一下，如果有的话我们就可以直接完成映射，而不用访问那么多次内存了。

但是，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。

六、实验流程

Step1. 建立内核空间虚拟地址映射

之前实验中的内核实际上是直接在物理地址空间上运行的。这样虽然比较简单，但是为了后续能够支持分页机制，我们要先把内核的运行环境从物理地址空间转移到虚拟地址空间，为之后的功能打好铺垫。

因此本次实验中，我们将会把内核代码放在虚拟地址空间中以 `0xffffffffc0200000` 开头的一段高地址空间中，该虚拟地址对应物理空间中的 `0x80200000`。为达到这个目的，我们需要将下面的参数修改一下：

```
// tools/kernel.ld
BASE_ADDRESS = 0xFFFFFFFFC0200000;
//之前这里是 0x80200000
```

我们修改了链接脚本中的起始地址。但是这样做的话，就能从物理地址空间转移到虚拟地址空间了吗？是的。让我们回顾一下在相当于 bootloader 的 OpenSBI 结束后，我们要面对的是怎样一种局面：

- 物理内存状态：OpenSBI 代码放在 `[0x80000000,0x80200000)` 中，内核代码则通过 `-device loader,file=bin/ucore.bin,addr=0x80200000` 被指定放在以 `0x80200000` 开头的一块连续物理内存中。
- CPU 状态：处于 S Mode ， 寄存器 `satp` 的 MODE被初始化为 `Bare` ，即无论取指还是访存我们都通过物理地址直接访问物理内存。 PC=0x80200000 指向内核的第一条指令。栈顶地址 SP 处在 OpenSBI 代码内。
- 内核代码：由于改动了链接脚本的起始地址，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，以此为依据确定代码里每个部分的地址（每一段都是从 `BASE_ADDRESS` 往后依次摆开的，所以代码里各段都会认为自己在 `0xffffffffc0200000` 之后的某个地址上，或者说编译器和链接器会把里面的符号/变量地址都对应到 `0xffffffffc0200000` 之后的某个地址上）。而实际上这个Base地址对应的物理地址是 `0x80200000` 。

通过这个修改，我们实现了虚拟地址和物理地址的映射，尽管这个映射只是简单的进行了一个偏移。

现在，我们可以直接将 DRAM 物理内存起止地址以及物理地址和虚拟地址的映射偏移量硬编码到内核中：

```
// kern/mm/memlayout.h

#define KERNBASE            0xFFFFFFFFC0200000 //我们设定的虚拟地址base
#define KMEMSIZE            0x7E00000
#define KERNTOP             (KERNBASE + KMEMSIZE)

#define PHYSICAL_MEMORY_END         0x88000000
#define PHYSICAL_MEMORY_OFFSET      0xFFFFFFFF40000000 //物理地址和虚拟地址的偏移量
#define KERNEL_BEGIN_PADDR          0x80200000 //起始物理地址base
#define KERNEL_BEGIN_VADDR          0xFFFFFFFFC0200000
```

接下来，我们在入口点 `entry.S` 中所要做的事情是：将 SP 寄存器从原先指向OpenSBI 某处的栈空间，改为指向我们自己在内核的内存空间里分配的栈；同时需要跳转到函数 `kern_init` 中。

在之前的实现中，我们已经在 `entry.S` 自己分配了一块 16KiB 的内存用来做启动栈：

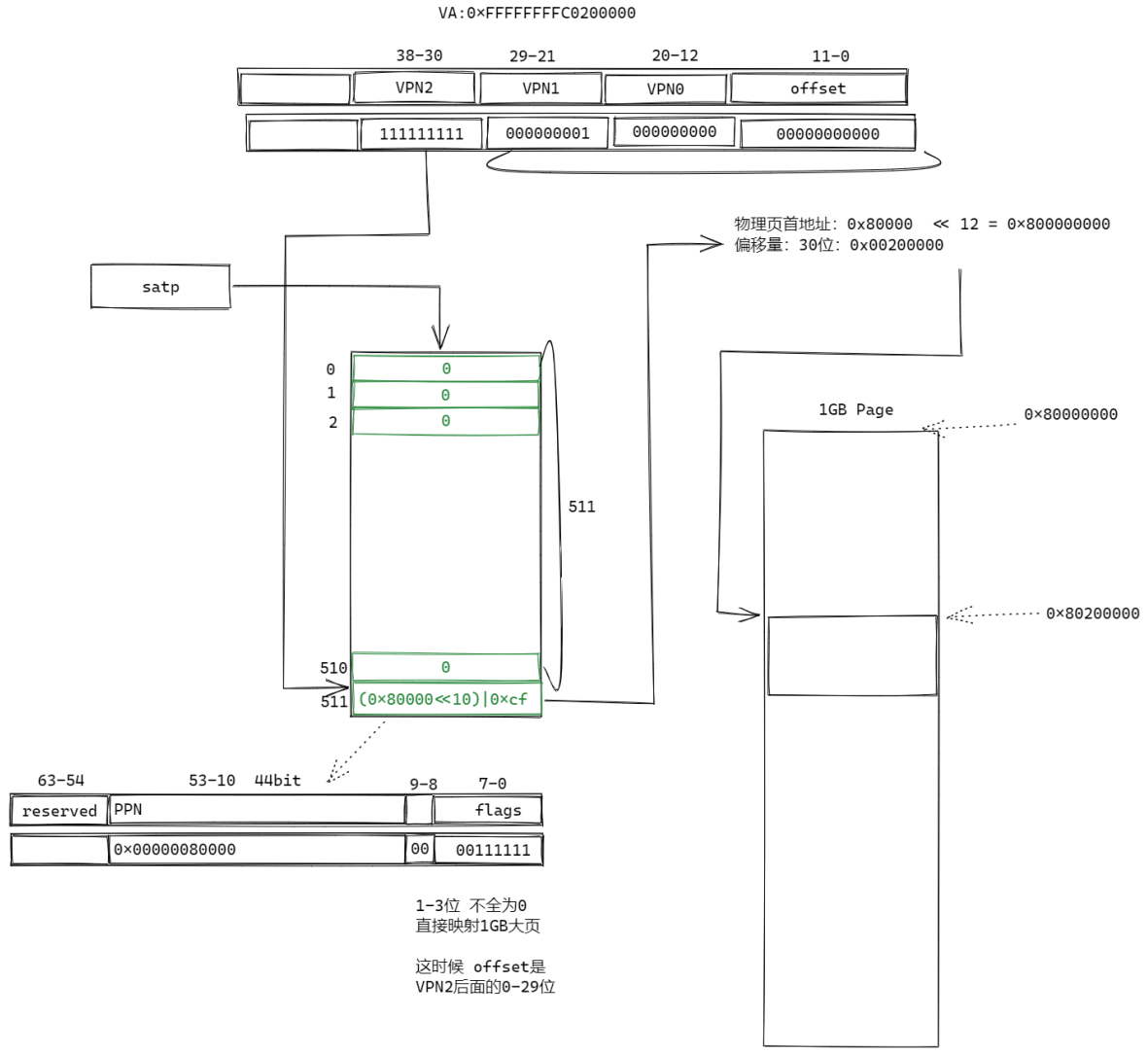
```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

符号 `bootstacktop` 就是我们需要的栈顶地址，符号 `kern_init` 代表了我们要跳转到的地址。之前我们直接将 `bootstacktop` 的值给到 SP， 再跳转到 `kern_init` 就行了。看起来原来的代码仍然能用啊？

问题在于，由于我们修改了链接脚本的起始地址，编译器和链接器认为内核开头地址为 `0xffffffffc0200000`，因此这两个符号会被翻译成比这个开头地址还要高的某个虚拟地址。而我们的 CPU 目前还处于 `Bare` 模式，会将地址都当成物理地址处理。这样，我们跳转到 `kern_init` 就会跳转到比 `0xffffffffc0200000` 还大 的一个物理地址，物理地址都没有这么多位！这显然是会出问题的。

于是，我们需要想办法利用刚学的页表知识，帮内核将需要的虚拟地址空间构造出来。也就是：构建一个合适的页表，让 `satp` 指向这个页表，然后使用地址的时候都要经过这个页表的翻译，使得虚拟地址 `0xFFFFFFFFC0200000` 经过页表的翻译恰好变成 `0x80200000`，就不会出错了。

下面这张图解释了我们实现的过程：



我们实现一个最简单的页表，所有的虚拟地址有一个固定的偏移量。比如内核的第一条指令，虚拟地址为 `0xffffffffc0200000`，物理地址为 `0x80200000`，因此，我们只要将虚拟地址减去 `0xffffffff40000000`，就得到了物理地址。

使用上一节页表的知识，我们只需要做到当访问内核里面的一个虚拟地址 va 时，我们知道 va处的代码或数据放在物理地址为 `pa = va - 0xffffffff40000000` 处的物理内存中，我们真正所要做的是要让 CPU 去访问 }pa。因此，我们要通过恰当构造页表，来对于内核所属的虚拟地址，实现这种 va→pa 的映射。

还记得上一节中所讲的吉页吗？那时我们提到，将一个根页表项的标志位 R,W,X不设为全 0，可以将它变为一个叶子，从而获得大小为 1GiB的一个吉页。

我们假定内核大小不超过 1GiB，则可以通过一个吉页将虚拟地址区间 `[0xffffffffc0000000, 0xffffffffffffffff]` 映射到物理地址区间 `[0x80000000, 0xc0000000)`，那么我们只需要分配一页内存用来存放根页表，并将其最后一个页表项（也就是对应我们使用的虚拟地址区间的页表项）进行适当设置即可。

```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    # t0 := 三级页表的虚拟地址
    lui     t0, %hi(boot_page_table_sv39)
    # t1 := 0xffffffff40000000 即虚实映射偏移量
```

```
    li      t1, 0xffffffffc0000000 - 0x80000000
    # t0 减去虚实映射偏移量 0xffffffff40000000，变为三级页表的物理地址
    sub     t0, t0, t1
    # t0 >>= 12，变为三级页表的物理页号
    srli    t0, t0, 12

    # t1 := 8 << 60，设置 satp 的 MODE 字段为 Sv39
    li      t1, 8 << 60
    # 将刚才计算出的预设三级页表物理页号附加到 satp 中
    or      t0, t0, t1
    # 将算出的 t0(即新的MODE|页表基址物理页号) 覆盖到 satp 中
    csrw    satp, t0
    # 使用 sfence.vma 指令刷新 TLB
    sfence.vma
    # 从此，我们给内核搭建出了一个完美的虚拟内存空间！
    #nop # 可能映射的位置有些bug。。插入一个nop

    # 我们在虚拟内存空间中：随意将 sp 设置为虚拟地址！
    lui sp, %hi(bootstacktop)

    # 我们在虚拟内存空间中：随意跳转到虚拟地址！
    # 跳转到 kern_init
    lui t0, %hi(kern_init)
    addi t0, t0, %lo(kern_init)
    jr t0

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:

.section .data
    # 由于我们要把这个页表放到一个页里面，因此必须 12 位对齐
    .align PGSHIFT
    .global boot_page_table_sv39
# 分配 4KiB 内存给预设的三级页表的根页表
boot_page_table_sv39:
    # 0xffffffff_c0000000 map to 0x80000000 (1G)
    # 前 511 个页表项均设置为 0 ，因此 V=0 ，意味着是空的(unmapped)
    .zero 8 * 511
    # 设置最后一个页表项，PPN=0x80000，标志位 VRWXAD 均为 1
    .quad (0x80000 << 10) | 0xcf # VRWXAD
```

总结一下，要进入虚拟内存访问方式，需要如下步骤：

1. 分配页表所在内存空间并初始化页表；
2. 设置好页基址寄存器（指向页表起始地址）；
3. 刷新 TLB

Step2. 了解地址转换函数

这里是`pmm.h`里对物理页面和虚拟地址，物理地址进行转换的一些函数或者是宏定义。

1.KADDR, PADDR

这两个宏进行的是物理地址和虚拟地址的互换，由于我们在ucore里实现的页表映射很简单，所有物理地址和虚拟地址的偏移值相同，所以这两个宏本质上只是做了一步加法/减法，额外还做了一些合法性检查。

```
/* *
 * PADDR - takes a kernel virtual address (an address that points above
 * KERNBASE),
 * where the machine's maximum 256MB of physical memory is mapped and returns
 * the
 * corresponding physical address.  It panics if you pass it a non-kernel
 * virtual address.
 * */
#define PADDR(kva)                                                  \
    ({                                                              \
        uintptr_t __m_kva = (uintptr_t)(kva);                       \
        if (__m_kva < KERNBASE) {                                   \
            panic("PADDR called with invalid kva %08lx", __m_kva); \
        }                                                           \
        __m_kva - va_pa_offset;                                     \
    })

/* *
 * KADDR - takes a physical address and returns the corresponding kernel virtual
 * address. It panics if you pass an invalid physical address.
 * */
#define KADDR(pa)                                                   \
    ({                                                              \
        uintptr_t __m_pa = (pa);                                    \
        size_t __m_ppn = PPN(__m_pa);                               \
        if (__m_ppn >= npage) {                                     \
            panic("KADDR called with invalid pa %08lx", __m_pa); \
        }                                                           \
        (void *)(__m_pa + va_pa_offset);                            \
    })
```

2. 关于page结构体转换的相关函数

page2ppn 结构体 -> 物理页号

我们曾经在内存里分配了一堆连续的Page结构体，来管理物理页面。可以把它们看作一个结构体数组。pages指针是这个数组的起始地址，减一下，加上一个基准值nbase, 就可以得到正确的物理页号。pages指针和nbase基准值我们都在其他地方做了正确的初始化

```
static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }
```

Page结构体和物理地址的相互转化

指向某个Page结构体的指针，对应一个物理页面，也对应一个起始的物理地址。左移若干位就可以从物理页号得到页面的起始物理地址。

```
static inline uintptr_t page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}
```

倒过来，从物理页面的地址得到所在的物理页面。实际上是得到管理这个物理页面的Page结构体。

```
static inline struct Page *pa2page(uintptr_t pa) {
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa");
    }
    return &pages[PPN(pa) - nbase];//把pages指针当作数组使用
}
```

Page结构体和虚拟地址的相互转化

这里再物理地址转化的基础上使用了KADDR 和 PADDR完成了虚拟地址的转化

```
static inline void *page2kva(struct Page *page) { return KADDR(page2pa(page)); }

static inline struct Page *kva2page(void *kva) { return pa2page(PADDR(kva)); }
```

Page结构体和对应页表项的相互转化

```
//从页表项得到对应的页，这里用到了 PTE_ADDR(pte)宏，对页表项做操作，在mmu.h里定义
static inline struct Page *pte2page(pte_t pte) {
    if (!(pte & PTE_V)) {
        panic("pte2page called with invalid pte");
    }
    return pa2page(PTE_ADDR(pte));
}
//PDE(Page Directory Entry)指的是不在叶节点的页表项（指向低一级页表的页表项）
static inline struct Page *pde2page(pde_t pde) { //PDE_ADDR这个宏和PTE_ADDR是一样的
    return pa2page(PDE_ADDR(pde));
}

// address in page table or page directory entry
// 把页表项后10位标志位设置为0  然后左移2位  offset是12位
#define PTE_ADDR(pte)   (((uintptr_t)(pte) & ~0x3FF) << (PTXSHIFT -
PTE_PPN_SHIFT))
#define PDE_ADDR(pde)   PTE_ADDR(pde)
```

七、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. riscv虚拟和物理内存管理的机制
2. 初始化内核映射的方式
3. 如何在代码实现虚拟和物理地址的转换

八、下一实验简单介绍

在下一个实验中，我们将建立页表映射，处理缺页异常（Page Fault）。