Lab2知识点

实验目的

- 理解页表的建立和使用方法
- 理解物理内存的管理方法
- 理解页面分配算法

实验内容

实验一过后大家做出来了一个可以启动的系统，实验二主要涉及操作系统的物理内存管理。操作系统为了使用内存，还需高效地管理内存资源。本次实验我们会了解如何发现系统中的物理内存，然后学习如何建立对物理内存的初步管理，即了解连续物理内存管理，最后掌握页表相关的操作，即如何建立页表来实现虚拟内存到物理内存之间的映射，帮助我们对段页式内存管理机制有一个比较全面的了解。本次的实验主要是在实验一的基础上完成物理内存管理，并建立一个最简单的页表映射。

练习

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

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。请回答如下问题：

- 你的first fit算法是否有进一步的改进空间?

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，并回答如下问题：

- 你的 Best-Fit 算法是否有进一步的改进空间?

扩展练习Challenge：buddy system（伙伴系统）分配算法（需要编程）

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

- 参考伙伴分配器的一个极简实现， 在ucore中实现buddy system分配算法，要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

扩展练习Challenge：任意大小的内存单元slub分配算法（需要编程）

slub算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。

- 参考linux的slub分配算法/，在ucore中实现slub分配算法。要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

扩展练习Challenge：硬件的可用物理内存范围的获取方法（思考题）

- 如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围?

> Challenges是选做，完成Challenge并回答了助教问题的小组可获得本次实验的加分。

项目组成

表1：实验二文件列表

```
── Makefile
├── kern
|   ├── debug
|   |   ├── assert.h
|   |   ├── kdebug.c
|   |   ├── kdebug.h
|   |   ├── kmonitor.c
|   |   ├── kmonitor.h
|   |   ├── panic.c
|   |   └── stab.h
|   ├── driver
|   |   ├── clock.c
|   |   ├── clock.h
|   |   ├── console.c
|   |   ├── console.h
|   |   ├── intr.c
|   |   ├── intr.h
|   ├── init
|   |   ├── entry.S
|   |   └── init.c
|   ├── libs
|   |   └── stdio.c
|   ├── mm
|   |   ├── best_fit_pmm.c
|   |   ├── best_fit_pmm.h
|   |   ├── default_pmm.c
|   |   ├── default_pmm.h
|   |   ├── memlayout.h
|   |   ├── mmu.h
|   |   ├── pmm.c
|   |   └── pmm.h
|   └── trap
|       ├── trap.c
|       ├── trap.h
|       └── trapentry.S
├── libs
|   ├── atomic.h
```

```
|   ├── defs.h
|   ├── error.h
|   ├── list.h
|   ├── printfmt.c
|   ├── readline.c
|   ├── riscv.h
|   ├── sbi.c
|   ├── sbi.h
|   ├── stdarg.h
|   ├── stdio.h
|   ├── string.c
|   └── string.h
└── tools
    ├── boot.ld
    ├── function.mk
    ├── gdbinit
    ├── grade.sh
    ├── kernel.ld
    ├── kernel_nopage.ld
    ├── kflash.py
    ├── rustsbi-k210.bin
    ├── sign.c
    ├── vector.c
```

编译方法

编译并运行代码的命令如下：

```
make

make qemu
```

则可以得到如下显示界面（仅供参考）

```
chenyu$ make qemu
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200036 (virtual)
  etext  0xffffffffc0201ad2 (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0206470 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
……
```

通过上图，我们可以看到ucore在显示其entry（入口地址）、etext（代码段截止处地址）、edata（数据段截止处地址）、和end（ucore截止处地址）的值后，ucore显示了物理内存的布局信息，其中包含了内存范围。接下来ucore会以页为最小分配单位实现一个简单的内存分配管理，完成页表的建立，进入分页模式，执行各种我们设置的检查，最后显示ucore建立好的页表内容，并在分页模式下响应时钟中断。

物理内存管理

基本原理概述

物理内存管理

什么是物理内存管理？如果我们只有物理内存空间，不进行任何的管理操作，那么我们也可以写程序。但这样显然会导致所有的程序，不管是内核还是用户程序都处于同一个地址空间中，这样显然是不好的。

举个例子：如果系统中只有一个程序在运行，那影响自然是有限的。但如果很多程序使用同一个内存空间，比如此时内核和用户程序都想访问 `0x80200000` 这个地址，那么因为它们处于一个地址空间中就会导致互相干扰，甚至是互相破坏。

那么如何消除这种影响呢？大家显然可以想象得到，我们可以通过让用户程序访问的 `0x80200000` 和内核访问的 `0x80200000` 不是一个地址来解决这个问题。但是如果我们只有一块内存，那么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，我们可以用一个"词典"实现。通过这个"词典"给出翻译之前的地址，可以在词典里查找翻译后的地址。而对每个程序往往都有着唯一的一本"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"。

物理地址和虚拟地址

在本次实验中，我们使用的是RISCV的 `sv39` 页表机制，每个页的大小是4KB，也就是4096个字节。通过之前的介绍相信大家对物理地址和虚拟地址有了一个初步的认识了，页表就是那个"词典"，里面有程序使用的虚拟页号到实际内存的物理页号的对应关系，但并不是所有的虚拟页都有对应的物理页。虚拟页可能的数目远大于物理页的数目，而且一个程序在运行时，一般不会拥有所有物理页的使用权，而只是将部分物理页在它的页表里进行映射。

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，**我们规定 63–39 位的值必须等于第 38 位的值**（大家可以将它类比为有符号整数），否则会认为该虚拟地址不合法，在访问时会产生异常。 不论是物理地址还是虚拟地址，我们都可以认为，**最后12位表示的是页内偏移，也就是这个地址在它所在页帧的什么位置（同一个位置的物理地址和虚拟地址的页内偏移相同）。除了最后12位，前面的部分表示的是物理页号或者虚拟页号。**

> 所以页的大小为2的12次，在代码中定义了！

实验执行流程概述

本次实验主要完成ucore内核对物理内存的管理工作。我们要在lab1实验的工作上对ucore进行相关拓展，修改ucore总控函数kern_init的代码。

kernel在后续执行中能够探测出的物理内存情况进行物理内存管理初始化工作。其次，**我们修改了entry.S中的kern_entry函数。**kern_entry函数的主要任务是设置虚拟内存管理，将三级页表的物理地址和Sv39模式位写入satp寄存器，以建立内核的虚拟内存空间，为之后建立分页机制的过程做一个准备。完成这些工作后，才调用kern_init函数。

```
kern_entry:
    # t0 := 三级页表的虚拟地址，lui加载高20位进入t0，低12位为页内偏移量我们不需要
    # boot_page_table_sv39 是一个全局符号，它指向系统启动时使用的页表的开始位置
    lui     t0, %hi(boot_page_table_sv39)
    # t1 := 0xffffffff40000000 即虚实映射偏移量，这一步是得到虚实映射偏移量
    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（物理地址右移12位抹除低12位后得到物理页号）
    srli    t0, t0, 12

    # t1 := 8 << 60，设置 satp 的 MODE 字段为 Sv39 39位虚拟地址模式
    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
    //一个按位或操作把satp的MODE字段，高1000后面全0，和三级页表的物理页号t1合并到一起
    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
    // satp放的是最高级页表的物理页号（44位），除此以外还有MODE字段（4位）、备用 ASID
（address space identifier）16位
    csrw    satp, t0
    # 使用 sfence.vma 指令刷新 TLB
    sfence.vma
    #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
    # 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) // 指向一个预先定义的虚拟地址 bootstacktop，这是内核栈的顶
部。

    # 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
```

kern_init函数在完成一些输出并对lab1实验结果的检查后，将进入物理内存管理初始化的工作，即**调用pmm_init函数完成物理内存的管理。接着是执行中断和异常相关的初始化工作，即调用idt_init函数。**

```
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init();  // init the console
```

```
    const char *message = "(NKU.CST) os is loading ...\0";
    //cprintf("%s\n\n", message);
    cputs(message);

    print_kerninfo();

    // grade_backtrace();
    idt_init();  // init interrupt descriptor table 初始化中断描述符表IDT

    pmm_init();  // init physical memory management 物理内存管理
    /* pmm_init()函数需要注册缺页中断处理程序，用于处理页面访问异常。
        当程序试图访问一个不存在的页面时，CPU会触发缺页异常，此时会调用缺页中断处理程序
        该程序会在物理内存中分配一个新的页面，并将其映射到虚拟地址空间中。
    */

    idt_init();  // init interrupt descriptor table

    clock_init();   // init clock interrupt 时钟中断
    /*
    clock_init()函数需要注册时钟中断处理程序，用于定时触发时钟中断。
    当时钟中断被触发时，CPU会跳转到时钟中断处理程序，该程序会更新系统时间，并执行一些周期性的操作，如调度进程等
    */
    //这两个函数都需要使用中断描述符表，所以要在中断描述符表初始化之后再初始化时钟中断
    intr_enable();  // enable irq interrupt 开启中断

    /* do nothing */
    while (1)
        ;
}
```

> pmm_init函数完成了物理内存的管理；idt_init函数完成了中断和异常的处理

为了完成物理内存管理，这里首先需要探测可用的物理内存资源；了解到物理内存位于什么地方，有多大之后，就以固定页面大小来划分整个物理内存空间，并准备以此为最小内存分配单位来管理整个物理内存，管理在内核运行过程中每页内存，设定其可用状态（free的，used的，还是reserved的），这其实就对应了我们在课本上讲到的连续内存分配概念和原理的具体实现；接着ucore kernel就要建立页表，启动分页机制，让CPU的MMU把预先建立好的页表中的页表项读入到TLB中，根据页表项描述的虚拟页（Page）与物理页帧（Page Frame）的对应关系完成CPU对内存的读、写和执行操作。这一部分其实就对应了我们在课本上讲到内存映射、页表、多级页表等概念和原理的具体实现。

ucore在实现上述技术时，需要解决两个关键问题：

- 如何建立虚拟地址和物理地址之间的联系
- 如何在现有ucore的基础上实现物理内存页分配算法

接下来将进一步分析完成lab2主要注意的关键问题和涉及的关键数据结构。

以页为单位管理物理内存

页表项

一个页表项是用来描述一个虚拟页号如何映射到物理页号的。**如果一个虚拟页号通过某种手段找到了一个页表项，并通过读取上面的物理页号完成映射，那么我们称这个虚拟页号通过该页表项完成映射。**而我们的"词典"（页表）存储在内存里，由若干个格式固定的"词条"也就是页表项（PTE, Page Table Entry）组成。显然我们需要给词典的每个词条约定一个固定的格式（包括每个词条的大小，含义），这样查起来才方便。

那么在sv39的一个页表项占据8字节（64位），那么页表项结构是这样的：

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 位共44位为一个物理页号，表示这个虚拟页号映射到的物理页号。后面的第 9-0 位共10位则描述映射的状态信息。**

介绍一下映射状态信息各位的含义：

- 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，即 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,W,X 为许可位，分别表示是否**可读 (Readable)，可写 (Writable)，可执行 (Executable)。**

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	指向下一级页表的指针
0	0	1	这一页只读
0	1	0	*保留(reserved for future use)*
0	1	1	这一页可读可写（不可执行）
1	0	0	这一页可读可执行（不可写）

X	W	R	Meaning
1	0	1	这一页可读可执行
1	1	0	*保留(reserved for future use)*
1	1	1	这一页可读可写可执行

- **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。

多级页表

在实际使用中显然如果只有一级页表，那么我们构建出来的虚拟地址空间毕竟还是过于有限，因此我们需要引入多级页表以实现更大规模的虚拟地址空间。

但相比于可用的物理内存空间，我们的虚拟地址空间太大，不可能为每个虚拟内存页都分配一个页表项。在Sv39中，因为一个页表项占据8字节（64位），而虚拟地址有39位，后12位是页内偏移，**那么还剩下27位可以编码不同的虚拟页号。**

如果开个大数组Pagetable[], 给个2^27 虚拟页号都分配8字节的页表项，其中Pagetable[vpn]代表虚拟页号为vpn的虚拟页的页表项，那就是整整1 GiB的内存。但这里面其实很多虚拟地址我们没有用到，会有大片大片的页表项的 标志位为0（不合法），显然我们不应该为那么多非法页表项浪费宝贵的内存空间。

因此，我们可以对页表进行“分级”，让它变成一个树状结构。也就是把很多页表项组合成一个”大页“，如果这些页表项都非法（没有对应的物理页），那么只需要用一个非法的页表项来覆盖这个大页，而不需要分别建立一大堆非法页表项。很多个大页(megapage)还可以组合起来变成大大页(gigapage!)，继而可以有更大的页，以此类推，当然肯定不是分层越多越好，因为随着层数增多，开销也会越大。

在本次实验中，我们使用的sv39权衡各方面效率，**使用三级页表。**有4KiB=4096字节的页，大小为2MiB= 2^21 字节的大页，和大小为1 GiB 的大大页。

原先的一个39位虚拟地址，被我们看成27位的页号和12位的页内偏移。那么在三级页表下，**我们可以把它看成9位的“大大页页号”，9位的“大页页号”（也是大大页内的页内偏移），9位的“页号”（大页的页内偏移），还有12位的页内偏移。**这是一个递归的过程，中间的每一级页表映射是类似的。也就是说，**整个Sv39的虚拟内存空间里，有512（2的9次方）个大大页，每个大大页里有512个大页，每个大页里有512个页，每个页里有4096个字节，整个虚拟内存空间里就有512*512*512*4096个字节，是512GiB的地址空间。**

那么为啥是512呢？注意，4096/8 = 512，我们恰好可以在一页里放下512个页表项！

我们可以认为，Sv39的多级页表在逻辑上是一棵树，它的每个叶子节点（直接映射4KB的页的页表项）都对应内存的一页，它的每个内部节点都对应512个更低一层的节点，而每个内部节点向更低一层的节点的链接都使用内存里的一页进行存储。

或者说，Sv39页表的根节点占据一页4KiB的内存，存储512个页表项，分别对应512个1 GiB的大大页，其中有些页表项（大大页）是非法的，另一些合法的页表项（大大页）是根节点的儿子，可以通过合法的页表项跳转到一个物理页号，这个物理页对应树中一个“大大页”的节点，里面有512个页表项，每个页表项对应一个2MiB的大页。同样，这些大页可能合法，也可能非法，非法的页表项不对应内存里的页，合法的页表项会跳转到一个物理页号，这个物理页对应树中一个“大页”的节点，里面有512个页表项，每个页表项对应一个4KiB的页，在这里最终完成虚拟页到物理页的映射。

三级和二级页表项不一定要指向下一级页表。我们知道每个一级页表项控制一个虚拟页号，即控制 4KiB 虚拟内存；每个二级页表项则控制 9 位虚拟页号，总计控制 4KiB×2^9 =2MiB 虚拟内存；每个三级页表项控制 18 位虚拟页号，总计控制 2MiB×2^9 =1GiB 虚拟内存。我们可以将二级页表项的 R,W,X 设置为不是全 0 的许可要求，那么它将与一级页表项类似，只不过可以映射一个 2MiB 的大页 (Mega Page) 。同理，也可以将三级页表项看作一个叶子，来映射一个 1GiB 的大大页(Giga Page)。

页表基址

在翻译的过程中，我们首先需要知道树状页表的根节点的物理地址。这一般保存在一个特殊寄存器里。**对于RISCV架构，是一个叫做`satp`（Supervisor Address Translation and Protection Register）的CSR**。实际上，`satp`里面存的不是最高级页表的起始物理地址，而是它所在的**物理页号**。除了物理页号，`satp`还包含其他信息。

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

MODE表示当前页表的模式：

- 0000表示不使用页表，直接使用物理地址，在简单的嵌入式系统里用着很方便。
- **1000表示sv39页表，也就是我们使用的，虚拟内存空间高达`512GiB`。**
- 1001表示Sv48页表，它和Sv39兼容。
- 其他编码保留备用 ASID（address space identifier）我们目前用不到 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。

建立快表以加快访问效率

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。

> 也就是说，如果检测到TLB快表中，存在该块，就直接读就行，如果发现修改了部分值，那就用指令刷新整个TLB

分页机制的设计思路

建立段页式管理中需要考虑的关键问题

为了实现分页机制，需要建立好虚拟内存和物理内存的页映射关系，即正确建立三级页表。此过程涉及硬件细节，不同的地址映射关系组合，相对比较复杂。总体而言，我们需要思考如下问题：

- 对于哪些物理内存空间需要建立页映射关系？
- 具体的页映射关系是什么？
- 页目录表的起始地址设置在哪里？
- 页表的起始地址设置在哪里，需要多大空间？
- 如何设置页目录表项的内容？
- 如何设置页表项的内容？

实现分页机制

接下来我们就正式开始实验啦！ 首先我们要做的是内核初始化的修改，我们现在需要做的就是把原本只能直接在物理地址空间上运行的内核引入页表机制。 具体来说，我们现在想将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。那怎么做呢？首先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？大家可以分析一下现在我们相当于是在 bootloader 的 OpenSBI 结束后的现状，这样就可以更好的理解接下来我们需要干什么：

- 物理内存状态：OpenSBI 代码放在 `[0x80000000,0x80200000)` 中，内核代码放在以 `0x80200000` 开头的一块连续物理内存中。这个是实验一我们做完后就实现的效果。
- CPU 状态：处于 S Mode ，寄存器 `satp` 的 `MODE` 被设置为 `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：这部分由于改动了链接脚本的起始地址，**所以它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，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）

接下来，我们需要修改 `entry.S` 文件来实现内核的初始化，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。

在之前的实验中，我们已经在 `entry.S` 自己分配了一块 `8KiB` 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:
```

通过之前的实验大家应该都明白：**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。看上去上面的这个代码也能够实现我们想要的初始化效果，但问题在于，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`， 就意味着会跳转到比 `0xffffffffc0200000` 还大的一个物理地址。但物理地址显然不可能有这么多位！这就会出现问题。

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`，这个地址显然就比较合适了，也就不会出错了。

理论知识告诉我们，所有的虚拟地址有一个固定的偏移量。而要想实现页表结构这个偏移量显然是不可或缺的。而虚拟地址和物理地址之间的差值就可以当成是这个偏移量。

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。**所以当我们需要做到去访问内核里面的一个物理地址 `va` 时，而已知虚拟地址为 `va` 时，则 `va` 处的代码或数据就放在物理地址为 `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` 到 `pa` 的映射。

还记得之前的理论介绍的内容吗？那时我们提到，将一个三级页表项的标志位 `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` 的一个大页。

我们假定内核大小不超过 `1GiB`，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)`，而我们只需要分配一页内存用来存放三级页表，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。对应的代码如下所示：

```
#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
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，没有映射任何物理页面(unmapped)
    .zero 8 * 511
    # 设置最后一个页表项，0x80000 << 10：设置PPN=0x80000，标志位 VRWXAD 均为 1
（DAGU_XWRV 1100_1111）
    # 页表项合法、这一页可读可写可执行
    # 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=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。
    # quad 指令用于定义一个64位宽的数据项，值为 (0x80000 << 10) | 0xcf。
    .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。

到现在为止，看上去复杂无比的虚拟内存空间，我们终于得以窥视一二了。

物理内存管理的设计思路

物理内存管理的实现

在管理虚拟内存之前，我们首先需要能够管理物理内存，毕竟所有虚拟内存页都要对应到物理内存页才能使用。

不妨把我们的内存管理模块划分为物理内存管理和虚拟内存管理两个模块。

物理内存管理应当为虚拟内存管理提供这样的接口：

- 检查当前还有多少空闲的物理页，返回空闲的物理页数目
- 给出n，尝试分配n个物理页，可以返回一个起始地址和连续的物理页数目，也可能分配一些零散的物理页，返回一个连起来的链表。
- 给出起始地址和n，释放n个连续的物理页

在`kern_init()`里，我们调用一个新函数：`pmm_init()`，`kern_init()`函数我们在之前就有学习过，这里我们只是新增一个调用`pmm_init()`的接口。

```
// kern/init/init.c
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init();  // init the console
    const char *message = "(THU.CST) os is loading ...\0";
    cputs(message);
    print_kerninfo();

    idt_init();  // init interrupt descriptor table
    pmm_init();  // 新东西!
    clock_init();   // init clock interrupt
    intr_enable();  // enable irq interrupt
    /* do nothing */
    while (1)
        ;
}
```

那么`pmm_init()`究竟是用来干什么的呢？**其实`pmm_init()`主要就是用来主要负责初始化物理内存管理**，我们可以在pmm.c文件进行初始化操作。

```
// kern/mm/pmm.c
/* pmm_init - initialize the physical memory management */
void pmm_init(void)
{
    // We need to alloc/free the physical memory (granularity is 4KB or other
size).
    // So a framework of physical memory manager (struct pmm_manager)is defined
in pmm.h
    // First we should init a physical memory manager(pmm) based on the
framework.
    // Then pmm can alloc/free the physical memory.
```

```
    // Now the first_fit/best_fit/worst_fit/buddy_system pmm are available.
    init_pmm_manager();

    // detect physical memory space, reserve already used memory,
    // then use pmm->init_memmap to create free page list
    page_init();

    // use pmm->check to verify the correctness of the alloc/free function in a
pmm
    check_alloc_page();

    extern char boot_page_table_sv39[];
    // 启动时树状页表的根节点的虚拟地址和物理地址
    satp_virtual = (pte_t *)boot_page_table_sv39; // pte_t 页表项
    satp_physical = PADDR(satp_virtual);
    cprintf("satp virtual address: 0x%016lx\nsatp physical address: 0x%016lx\n",
satp_virtual, satp_physical);
}
```

- init_pmm_manager()：就是初始化我们的内存管理方式，就是我们可以选择我们自己设计的内存管理的结构体的成员

```
static void init_pmm_manager(void)
{
    // pmm_manager = &default_pmm_manager;
    // pmm_manager = &best_fit_pmm_manager;
    pmm_manager = &buddy_system_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

- **page_init()：将我们的页表内容进行初始化。**首先，设置虚拟到物理地址的偏移，然后规定物理内存的开始地址和结束地址，已经其大小。然后打印物理内存的映射信息。

```
/* All physical memory mapped at this address */
#define KERNBASE 0xFFFFFFFFC0200000 // = 0x80200000(物理内存里内核的起始位
置, KERN_BEGIN_PADDR) + 0xFFFFFFFF40000000(偏移量,
PHYSICAL_MEMORY_OFFSET)
// 把原有内存映射到虚拟内存空间的最后一页
#define KMEMSIZE 0x7E00000 // the maximum amount of physical memory
// 0x7E00000 = 0x8000000 - 0x200000
// QEMU 缺省的RAM为 0x80000000到0x88000000, 128MiB, 0x80000000到0x80200000
被OpenSBI占用
#define KERNTOP (KERNBASE + KMEMSIZE) // 0x88000000对应的虚拟地址
```

这一块实际上，KERNTOP为KERNBASE + KMEMSIZE，也就是虚拟地址。KERNBASE是物理内存里内核的起始位置+偏移量，也就是虚拟地址。KMEMSIZE相当于是全部内存的大小。

然后初始化了物理页面的数组，求得总的物理页面数

ROUNDUP是一个宏或函数，将给定的地址向上舍入到最接近的 PGSIZE 边界。保证最后的指针指向4kB对齐的地址

```
static void page_init(void)
{
```

```
    va_pa_offset = PHYSICAL_MEMORY_OFFSET; // 设置虚拟到物理地址的偏移: 硬编码0xFFFFFFFF40000000

    // 获取物理内存信息，下面变量表示物理内存的开始、大小和结束地址
    uint64_t mem_begin = KERNEL_BEGIN_PADDR; // 0x8020 0000
    uint64_t mem_size = PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR;
    uint64_t mem_end = PHYSICAL_MEMORY_END; // 硬编码取代 sbi_query_memory()接口 0x8800 0000

    // 打印物理内存映射信息
    cprintf("physcial memory map:\n");
    cprintf("  memory: 0x%016lx, [0x%016lx, 0x%016lx].\n", mem_size, mem_begin,
            mem_end - 1);

    // 限制物理内存上限:
    uint64_t maxpa = mem_end;
    cprintf("maxpa: 0x%016lx.\n", maxpa); // test point

    // ctrl+左键点进去看一下KERNTOP具体实现（在memlayout.h中，KERNTOP是KERNBASE + KMEMSIZE）:
    if (maxpa > KERNTOP)
    {
        maxpa = KERNTOP;
    }

    // 初始化物理页面数组
    // end是链接脚本中定义的内核结束位置，其实是个常量指针
    extern char end[];

    // 求得总的物理页面数
    npage = maxpa / PGSIZE;
    cprintf("npage: 0x%016lx.\n", npage); // test point,为0x8800_0
    cprintf("nbase: 0x%016lx.\n", nbase); // test point，为0x8000_0

    // kernel在0x8020 0000开始加载，在end[]结束，pages是剩下的页的开始，是一个指向物理页面数组的指针
    // ROUNDUP是一个宏或函数，将给定的地址向上舍入到最接近的 PGSIZE 边界。保证最后的指针指向4kB对齐的地址
    // 把page指针都指向内核所占内存空间结束后的第一页
    cprintf("end pythical address: 0x%016lx.\n", PADDR((uintptr_t)end)); // test point
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
    cprintf("pages pythical address: 0x%016lx.\n", PADDR((uintptr_t)pages)); // test point
    // pages pythical address是0x8020 7000，有0x7000的位置被kernel映像占用

    // 一开始把所有页面都设置为保留给内存使用的，然后再设置那些页面可以分配给其他程序
    for (size_t i = 0; i < npage - nbase; i++)
    {
        SetPageReserved(pages + i); // 在memlayout.h中，SetPageReserved是一个宏，将给定的页面标记为保留给内存使用的
    }

    // test ponit begin
    for (size_t i = 0; i < 5; i++)
```

```
    {
        cprintf("pages[%d] pythical address: 0x%016lx.\n", i,
PADDR((uintptr_t)(pages + i))); // test point
    }
    // test point end

    // 初始化空闲页面列表
    // PADDR 宏将这个虚拟地址转换为物理地址
    // 从这个地方开始才是我们可以自由使用的物理内存
    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) *
(npage - nbase)); // 0x8034 7000 = 0x8020 7000 + 0x28 * 0x8000
    cprintf("page结构体大小: 0x%016lx.\n", sizeof(struct Page));
              // test point

    // 按照页面大小PGSIZE进行对齐，ROUNDUP，ROUNDDOWN是在libs/defs.h定义的
    mem_begin = ROUNDUP(freemem, PGSIZE);
    mem_end = ROUNDDOWN(mem_end, PGSIZE);
    cprintf("freemem: 0x%016lx.\n", freemem);      // test point
    cprintf("mem_begin: 0x%016lx.\n", mem_begin); // test point
    cprintf("mem_end: 0x%016lx.\n", mem_end);      // test point

    if (freemem < mem_end)
    {
        // 初始化可以自由使用的物理内存
        init_memmap(pa2page(mem_begin), (mem_end - mem_begin) / PGSIZE);
    }
    cprintf("mem_begin对应的页结构记录(结构体page)虚拟地址: 0x%016lx.\n",
pa2page(mem_begin));         // test point
    cprintf("mem_begin对应的页结构记录(结构体page)物理地址: 0x%016lx.\n",
PADDR(pa2page(mem_begin))); // test point

    cprintf("可用空闲页的数目: 0x%016lx.\n", (mem_end - mem_begin) /
PGSIZE); // test point
    // 可用空闲页数目0x7cb9 ，0x7cb9>>12 + 0x80347000（membegin）
=0x88000000（memend）
    // 从0x8800 0000到0x8000 0000总共0x8000个页，其中0x7cb9个页可用，也就是总
共空闲页内存是0x7cb9000，也就是124MB
    // 0x8000-0x7cb9=0x0347个不可用，这些页存的是结构体page的数据
}
```

- check_alloc_page()：检查我们的分配的物理内存情况，调用我们写的check函数进行检查，输出成功信息

```
static void check_alloc_page(void)
{
    pmm_manager->check();
    cprintf("check_alloc_page() succeeded!\n");
}
```

我们在lab2增加了一些功能，方便我们编程：

- 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。

```
// kern/sync/sync.h
#ifndef __KERN_SYNC_SYNC_H__
#define __KERN_SYNC_SYNC_H__

#include <defs.h>
#include <intr.h>
#include <riscv.h>

static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void __intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}
//思考：这里宏定义的 do{}while(0)起什么作用?
#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)
#define local_intr_restore(x) __intr_restore(x);

#endif /* !__KERN_SYNC_SYNC_H__ */
```

- `static inline bool __intr_save(void)`：它检查当前处理器状态寄存器（`sstatus`）中的 `SSTATUS_SIE` 位。如果该位为 1，表示中断是启用的。在这种情况下，函数会禁用中断，并返回 `true`。如果中断本来就是禁用的，则直接返回 `false`。
- `static inline void __intr_restore(bool flag)`：如果传入的标志 `flag` 为 `true`，则启用中断；如果 `flag` 为 `false`，则不进行任何操作。
- local_intr_save：它使用了一个 `do-while(0)` 语句块来执行 `__intr_save()` 函数，并将返回值保存到变量 `x` 中。
- local_intr_restore：它调用 `__intr_restore(x)` 函数来恢复中断状态。

`list.h` 里面实现了一个简单的双向链表。虽然接口很多，但是只要对链表熟悉，不难理解。如果理解不了，可以先去学学数据结构这门课。

```
// libs/list.h
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct list_entry list_entry_t;

static inline void list_init(list_entry_t *elm) __attribute__((always_inline));
static inline void list_add(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
static inline void list_add_before(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
static inline void list_add_after(list_entry_t *listelm, list_entry_t *elm)
__attribute__((always_inline));
static inline void list_del(list_entry_t *listelm)
__attribute__((always_inline));
static inline void list_del_init(list_entry_t *listelm)
__attribute__((always_inline));
static inline bool list_empty(list_entry_t *list) __attribute__((always_inline));
static inline list_entry_t *list_next(list_entry_t *listelm)
__attribute__((always_inline));
static inline list_entry_t *list_prev(list_entry_t *listelm)
__attribute__((always_inline));
//下面两个函数仅在内部使用，不对外开放作为接口。
static inline void __list_add(list_entry_t *elm, list_entry_t *prev, list_entry_t
*next) __attribute__((always_inline));
static inline void __list_del(list_entry_t *prev, list_entry_t *next)
__attribute__((always_inline));
```

看起来`list.h`里面定义的`list_entry`并没有数据域，但是，如果我们把`list_entry`作为其他结构体的成员，就可以利用C语言结构体内存连续布局的特点，从`list_entry`的地址获得它所在的上一级结构体。

于是我们定义了可以连成链表的`Page`结构体和一系列对它做操作的宏。这个结构体用来管理物理内存。

```
// libs/defs.h

/* Return the offset of 'member' relative to the beginning of a struct type */
// 空指针强转为结构体指针，然后取成员的地址，再转换为size_t，就是把结构体放在地址0的地方
// 然后取成员的地址，就是成员在结构体中的偏移量
#define offsetof(type, member)                                      \
    ((size_t)(&((type *)0)->member))

/* *
 * to_struct - get the struct from a ptr
 * @ptr:    a struct pointer of member
 * @type:   the type of the struct this is embedded in
 * @member: the name of the member within the struct
 * */
// 将一个结构体中的成员的指针转换为结构体的指针
#define to_struct(ptr, type, member)                               \
    ((type *)((char *)(ptr) - offsetof(type, member)))

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

```
 * struct Page - Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as physical address.
 * */
struct Page
{
    int ref;                // page frame's reference counter表示有多少个实体（例如进程、文件等）正在使用这个页面
    uint64_t flags;         // array of flags that describe the status of the page frame
    unsigned int property;  // the num of free block, used in first fit pm manager空闲块数目，BS中是这个块的大小（存的是幂次，2的这个property是块大小）
    list_entry_t page_link; // free list link
};

/* Flags describing the status of a page frame */
#define PG_reserved                 0       // if this bit=1: the Page is reserved for kernel, cannot be used in alloc/free_pages; otherwise, this bit=0
#define PG_property                 1       // if this bit=1: the Page is the head page of a free memory block(contains some continuous_addrress pages), and can be used in alloc_pages; if this bit=0: if the Page is the the head page of a free memory block, then this Page and the memory block is alloced. Or this Page isn't the head page.

// ***综上----flags是00表示这个页不被占用且（是空闲块头页 或 ），
// 01表示这个页不被内核占用的，且空闲
// 10表示这个页是是被占用块的头页

// 第二位为0表示个页是空闲的，且不是空闲块的头页 或 是被占用块的头页
// 第二位为1表示这个页是空闲块的头页

//这几个对page操作的宏用到了atomic.h的原子操作
#define SetPageReserved(page)       set_bit(PG_reserved, &((page)->flags))
#define ClearPageReserved(page)     clear_bit(PG_reserved, &((page)->flags))
#define PageReserved(page)          test_bit(PG_reserved, &((page)->flags))
#define SetPageProperty(page)       set_bit(PG_property, &((page)->flags))
#define ClearPageProperty(page)     clear_bit(PG_property, &((page)->flags))
#define PageProperty(page)          test_bit(PG_property, &((page)->flags))


// 由Page中链表的结构找到Page结构体的地址
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)

/* free_area_t是一个双向链表，用来存储空闲的页表
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
```

我们知道，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！

我们先不管那些外设，目前我们只关注物理内存。

物理内存探测的设计思路

操作系统怎样知道物理内存所在的那段物理地址呢？在 RISC-V 中，这个一般是由 bootloader，即 OpenSBI 来完成的。它来完成对于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。随后 OpenSBI 会将其地址保存在 `a1` 寄存器中，给我们使用。

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。

> 扩展 **Qemu 模拟的 RISC-V virt 计算机中的物理内存**
>
> 通过查看virt.c的**virt_memmap[]**的定义，可以了解到 Qemu 模拟的 RISC-V virt 计算机的详细物理内存布局。可以看到，整个物理内存中有不少内存空洞（即含义为**unmapped**的地址空间），也有很多外设特定的地址空间，现在我们看不懂没有关系，后面会慢慢涉及到。目前只需关心最后一块含义为**DRAM**的地址空间，这就是 OS 将要管理的 128MB 的内存空间。

起始地址	终止地址	含义
0x0	0x100	QEMU VIRT_DEBUG
0x100	0x1000	unmapped
0x1000	0x12000	QEMU MROM (包括 hard-coded reset vector; device tree)
0x12000	0x100000	unmapped
0x100000	0x101000	QEMU VIRT_TEST
0x101000	0x2000000	unmapped
0x2000000	0x2010000	QEMU VIRT_CLINT
0x2010000	0x3000000	unmapped
0x3000000	0x3010000	QEMU VIRT_PCIE_PIO
0x3010000	0xc000000	unmapped
0xc000000	0x10000000	QEMU VIRT_PLIC
0x10000000	0x10000100	QEMU VIRT_UART0
0x10000100	0x10001000	unmapped
0x10001000	0x10002000	QEMU VIRT_VIRTIO
0x10002000	0x20000000	unmapped
0x20000000	0x24000000	QEMU VIRT_FLASH
0x24000000	0x30000000	unmapped

起始地址	终止地址	含义
0x30000000	0x40000000	QEMU VIRT_PCIE_ECAM
0x40000000	0x80000000	QEMU VIRT_PCIE_MMIO
0x80000000	0x88000000	DRAM 缺省 128MB，大小可配置

不过为了简单起见，我们并不打算自己去解析这个结果。因为我们知道，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)`。我们直接将 DRAM 物理内存结束地址硬编码到内核中：

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

/* All physical memory mapped at this address */
#define KERNBASE 0xFFFFFFFFC0200000 // = 0x80200000(物理内存里内核的起始位置,
KERN_BEGIN_PADDR) + 0xFFFFFFFF40000000(偏移量, PHYSICAL_MEMORY_OFFSET)
// 把原有内存映射到虚拟内存空间的最后一页
#define KMEMSIZE 0x7E00000 // the maximum amount of physical memory
// 0x7E00000 = 0x8000000 - 0x200000
// QEMU 缺省的RAM为 0x80000000到0x88000000, 128MiB, 0x80000000到0x80200000被OpenSBI
占用
#define KERNTOP (KERNBASE + KMEMSIZE) // 0x88000000对应的虚拟地址

#define PHYSICAL_MEMORY_END 0x88000000
#define PHYSICAL_MEMORY_OFFSET 0xFFFFFFFF40000000
#define KERNEL_BEGIN_PADDR 0x80200000
#define KERNEL_BEGIN_VADDR 0xFFFFFFFFC0200000
```

但是，有一部分 DRAM 空间已经被占用，不能用来存别的东西了！

- 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；
- 其实设备树扫描结果 DTB 还占用了一部分物理内存，不过由于我们不打算使用它，所以可以将它所占用的空间用来存别的东西。

于是，我们可以用来存别的东西的物理内存的物理地址范围是：`[KernelEnd, 0x88000000)`。这里的 `KernelEnd` 为内核代码结尾的物理地址。在 `kernel.ld` 中定义的 `end` 符号为内核代码结尾的虚拟地址。

为了管理物理内存，我们需要在内核里定义一些数据结构，来存储"当前使用了哪些物理页面，哪些物理页面没被使用"这样的信息，使用的是Page结构体。我们将一些Page结构体在内存里排列在内核后面，这要占用一些内存。而摆放这些Page结构体的物理页面，以及内核占用的物理页面，之后都无法再使用了。我们用 `page_init()` 函数给这些管理物理内存的结构体做初始化。下面是代码：

```
// kern/mm/pmm.h

/* *
 * 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.
```

```
 * 用于将一个内核虚拟地址转换为对应的物理地址，它会检查传入的地址是否为内核虚拟地址，如果不是则会触发 panic 异常。
 * */
#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);                            \
    })
*/

extern struct Page *pages;
extern size_t npage;

// kern/mm/pmm.c

// pages指针保存的是第一个Page结构体所在的位置，也可以认为是Page结构体组成的数组的开头
// 由于C语言的特性，可以把pages作为数组名使用，pages[i]表示顺序排列的第i个结构体
struct Page *pages;
size_t npage = 0;
uint64_t va_pa_offset;
// memory starts at 0x80000000 in RISC-V
const size_t nbase = DRAM_BASE / PGSIZE;//(npage - nbase)表示物理内存的页数

static void page_init(void) {
    va_pa_offset = PHYSICAL_MEMORY_OFFSET; //硬编码 0xFFFFFFFF40000000

    uint64_t mem_begin = KERNEL_BEGIN_PADDR;//硬编码 0x80200000
    uint64_t mem_size = PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR;
    uint64_t mem_end = PHYSICAL_MEMORY_END; //硬编码 0x88000000

    cprintf("physcial memory map:\n");
    cprintf("  memory: 0x%016lx, [0x%016lx, 0x%016lx].\n", mem_size, mem_begin,
            mem_end - 1);

    uint64_t maxpa = mem_end;

    if (maxpa > KERNTOP) {
```

```
        maxpa = KERNTOP;
    }

    npage = maxpa / PGSIZE;

    extern char end[];
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
    //把pages指针指向内核所占内存空间结束后的第一页

    //一开始把所有页面都设置为保留给内核使用的，之后再设置哪些页面可以分配给其他程序
    for (size_t i = 0; i < npage - nbase; i++) {
        SetPageReserved(pages + i);//记得吗？在kern/mm/memlayout.h定义的
    }
    //从这个地方开始才是我们可以自由使用的物理内存
    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * (npage -
nbase));
    //按照页面大小PGSIZE进行对齐，ROUNDUP，ROUNDDOWN是在libs/defs.h定义的
    mem_begin = ROUNDUP(freemem, PGSIZE);
    mem_end = ROUNDDOWN(mem_end, PGSIZE);
    if (freemem < mem_end) {
        //初始化我们可以自由使用的物理内存
        init_memmap(pa2page(mem_begin), (mem_end - mem_begin) / PGSIZE);
    }
}
```

在`page_init()`的代码里，我们调用了一个函数`init_memmap()`，这和我们的另一个结构体`pmm_manager`有关。虽然C语言基本上不支持面向对象，但我们可以用类似面向对象的思路，把"物理内存管理"的功能集中给一个结构体。我们甚至可以让函数指针作为结构体的成员，强行在C语言里支持了"成员函数"。可以看到，我们调用的`init_memmap()`实际上又调用了`pmm_manager`的一个"成员函数"。

```
// kern/mm/pmm.c

// physical memory management
const struct pmm_manager *pmm_manager;


// init_memmap - call pmm->init_memmap to build Page struct for free memory
static void init_memmap(struct Page *base, size_t n) {
    pmm_manager->init_memmap(base, n);
}
// kern/mm/pmm.h
#ifndef __KERN_MM_PMM_H__
#define __KERN_MM_PMM_H__

#include <assert.h>
#include <atomic.h>
#include <defs.h>
#include <memlayout.h>
#include <mmu.h>
#include <riscv.h>

// pmm_manager is a physical memory management class. A special pmm manager -
// XXX_pmm_manager
// only needs to implement the methods in pmm_manager class, then
// XXX_pmm_manager can be used
```

```
// by ucore to manage the total physical memory space.
struct pmm_manager {
    const char *name;  // XXX_pmm_manager's name
    void (*init)(
        void);  // 初始化XXX_pmm_manager内部的数据结构（如空闲页面的链表）
    void (*init_memmap)(
        struct Page *base,
        size_t n);  //知道了可用的物理页面数目之后，进行更详细的初始化
    struct Page *(*alloc_pages)(
        size_t n);  // 分配至少n个物理页面，根据分配算法可能返回不同的结果
    void (*free_pages)(struct Page *base, size_t n);  // free >=n pages with
                                                      // "base" addr of Page
                                                      // descriptor
                                                      // structures(memlayout.h)
    size_t (*nr_free_pages)(void);  // 返回空闲物理页面的数目
    void (*check)(void);            // 测试正确性
};

extern const struct pmm_manager *pmm_manager;

void pmm_init(void);

struct Page *alloc_pages(size_t n);
void free_pages(struct Page *base, size_t n);
size_t nr_free_pages(void); // number of free pages

#define alloc_page() alloc_pages(1)
#define free_page(page) free_pages(page, 1)
```

pmm_manager提供了各种接口：分配页面，释放页面，查看当前空闲页面数。但是我们好像始终没看见pmm_manager内部对这些接口的实现，其实是因为那些接口只是作为函数指针，作为pmm_manager的一部分，我们需要把那些函数指针变量赋值为真正的函数名称。

还记得最早我们在`pmm_init()`里首先调用了`init_pmm_manager()`，在这里面我们把pmm_manager的指针赋值成`&default_pmm_manager`， 看起来我们在这里实现了那些接口。

```
// init_pmm_manager - initialize a pmm_manager instance
static void init_pmm_manager(void) {
    pmm_manager = &default_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
// alloc_pages - call pmm->alloc_pages to allocate a continuous n*PAGESIZE
// memory
struct Page *alloc_pages(size_t n) {
    // 在这里编写你的物理内存分配算法。
    // 你可以参考nr_free_pages() 函数进行设计，
    // 了解物理内存管理器的工作原理，然后在这里实现自己的分配算法。
    // 实现算法后，调用 pmm_manager->alloc_pages(n) 来分配物理内存，
    // 然后返回分配的 Page 结构指针。
}

// free_pages - call pmm->free_pages to free a continuous n*PAGESIZE memory
void free_pages(struct Page *base, size_t n) {
    // 在这里编写你的物理内存释放算法。
```

```
    // 你可以参考nr_free_pages() 函数进行设计，
    // 了解物理内存管理器的工作原理，然后在这里实现自己的释放算法。
    // 实现算法后，调用 pmm_manager->free_pages(base, n) 来释放物理内存。
}

// nr_free_pages - call pmm->nr_free_pages to get the size (nr*PAGESIZE)
// of current free memory
size_t nr_free_pages(void) {
    size_t ret;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        ret = pmm_manager->nr_free_pages();
    }
    local_intr_restore(intr_flag);
    return ret;
}
```

到现在，我们距离完整的内存管理， 就只差`default_pmm_manager`结构体的实现了，也就是我们要在里面实现页面分配算法。

页面分配算法

我们在`default_pmm.c`定义了一个pmm_manager类型的结构体，并实现它的接口

```
// kern/mm/default_pmm.h
#ifndef __KERN_MM_DEFAULT_PMM_H__
#define  __KERN_MM_DEFAULT_PMM_H__

#include <pmm.h>

extern const struct pmm_manager default_pmm_manager;

#endif /* ! __KERN_MM_DEFAULT_PMM_H__ */
```

较为关键的，是一开始如何初始化所有可用页面，以及如何分配和释放页面。大家可以学习下面的代码，其实现了First Fit算法。

```
// kern/mm/default_pmm.c
free_area_t free_area;

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;//nr_free可以理解为在这里可以使用的一个全局变量，记录可用的物理页面数
}
static void
default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
```

```
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
}

static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
```

```
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }

    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}

const struct pmm_manager default_pmm_manager = {
    .name = "default_pmm_manager",
    .init = default_init,
    .init_memmap = default_init_memmap,
    .alloc_pages = default_alloc_pages,
    .free_pages = default_free_pages,
    .nr_free_pages = default_nr_free_pages,
    .check = default_check,
};
```

所谓First Fit算法就是当需要分配页面时，它会从空闲页块链表中找到第一个适合大小的空闲页块，然后进行分配。当释放页面时，它会将释放的页面添加回链表，并在必要时合并相邻的空闲页块，以最大限度地减少内存碎片。

完成页面分配算法后我们的物理内存管理算是基本实现了，接下来请同学们完成本次实验练习。

答辩提问

双向链表（list.h）

```
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct list_entry list_entry_t;
```

在list.h中定义了双向链表，分别有两个指针，一个往前指，一个往后指

虚拟地址与物理地址的映射

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，**我们规定 63–39 位的值必须等于第 38 位的值**（大家可以将它类比为有符号整数），否则会认为该虚拟地址不合法，在访问时会产生异常。 不论是物理地址还是虚拟地址，我们都可以认为，**最后12位表示的是页内偏移，也就是这个地址在它所在页帧的什么位置（同一个位置的物理地址和虚拟地址的页内偏移相同）。除了最后12位，前面的部分表示的是物理页号或者虚拟页号。**

> 所以页的大小为2的12次，在代码中定义了！

default，best_fit，buddy_system的分配思想

- default：就是匹配第一个找到的比所需页数大的，简单直接，但是缺点是容易浪费小块的内存
- best_fit:可以找到最接近所需内存块大小的，但是时间效率不高
- buddy_system:相邻的作为伙伴，只对他们拆分与合并。这个是节省了时间效率，但是空间消耗很大
- slub：固定大小，使用缓存, 时间复杂度低（缺点：牺牲空间）

sv39页表项的组成和作用

一个页表项是用来描述一个虚拟页号如何映射到物理页号的。**如果一个虚拟页号通过某种手段找到了一个页表项，并通过读取上面的物理页号完成映射，那么我们称这个虚拟页号通过该页表项完成映射。**而我们的"词典"（页表）存储在内存里，由若干个格式固定的"词条"也就是页表项（PTE, Page Table Entry）组成。显然我们需要给词典的每个词条约定一个固定的格式（包括每个词条的大小，含义），这样查起来才方便。

那么在sv39的一个页表项占据8字节（64位），那么页表项结构是这样的：

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 位共44位为一个物理页号，表示这个虚拟页号映射到的物理页号。后面的第 9-0 位共10位则描述映射的状态信息。**

介绍一下映射状态信息各位的含义：

- 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，即 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,W,X 为许可位，分别表示是否**可读 (Readable)，可写 (Writable)，可执行 (Executable)。**

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	指向下一级页表的指针
0	0	1	这一页只读
0	1	0	*保留(reserved for future use)*
0	1	1	这一页可读可写（不可执行）
1	0	0	这一页可读可执行（不可写）
1	0	1	这一页可读可执行
1	1	0	*保留(reserved for future use)*
1	1	1	这一页可读可写可执行

- **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。

三级页表的实现

在本次实验中，我们使用的sv39权衡各方面效率，**使用三级页表。**有4KiB=4096字节的页，大小为2MiB= 2^21 字节的大页，和大小为1 GiB 的大大页。

原先的一个39位虚拟地址，被我们看成27位的页号和12位的页内偏移。那么在三级页表下，**我们可以把它看成9位的“大大页页号”，9位的“大页页号”（也是大大页内的页内偏移），9位的“页号”（大页的页内偏移），还有12位的页内偏移。**这是一个递归的过程，中间的每一级页表映射是类似的。也就是说，**整个Sv39的虚拟内存空间里，有512（2的9次方）个大大页，每个大大页里有512个大页，每个大页里有512个页，每个页里有4096个字节，整个虚拟内存空间里就有512*512*512*4096个字节，是512GiB的地址空间。**

页表多大

最大是1GiB，大大表

2MiB，是大表

4KiB，是正常的表

defs.h的内容

```
/* *
 取整操作（当 n 是 2 的幂次方时很高效）。向下取整到最接近的 n 的倍数
 * */
#define ROUNDDOWN(a, n) ({                                          \
            size_t __a = (size_t)(a);                               \
            (typeof(a))(__a - __a % (n));                           \
        })

/* 向上取整到最接近的 n 的倍数。 */
#define ROUNDUP(a, n) ({                                            \
            size_t __n = (size_t)(n);                               \
            (typeof(a))(ROUNDDOWN((size_t)(a) + __n - 1, __n));     \
        })

/*  返回成员'member'相对于结构体类型起始位置的偏移量。 */
#define offsetof(type, member)                                      \
    ((size_t)(&((type *)0)->member))

/* *
 * to_struct - get the struct from a ptr
 * @ptr:    a struct pointer of member
 * @type:   the type of the struct this is embedded in
 * @member: the name of the member within the struct
 * to_struct - 从一个指针获取结构体。@ptr：成员的结构体指针。@type：这个成员所在的结构体的类型。@member：结构体内成员的名称。
 * */
#define to_struct(ptr, type, member)                                \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

主要是to_struct

```
// 将列表项转换为页
#define le2page(le, member)                 \
    to_struct((le), struct Page, member)
```

将le指针转为Page结构体的member成员

物理地址是多少位

56位

虚拟地址是多少位

39位

一个页有多大

2的12次bytes，也就是4KB

一个页表有多少页表项

一个页表项 8字节，一共4096字节，所以有512个页表项

512个页表项

页表项的组成

那么在sv39的一个页表项占据8字节（64位），那么页表项结构是这样的：

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 位共44位为一个物理页号，表示这个虚拟页号映射到的物理页号。后面的第 9-0 位共10位则描述映射的状态信息。**

pmm.c文件

调用接口的函数：可以选择自己使用的接口

```
// init_pmm_manager - 初始化一个内存管理器的方式
static void init_pmm_manager(void) {
    //pmm_manager = &best_fit_pmm_manager;
    pmm_manager = &buddy_system_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

page_init函数:

```
static void page_init(void)
{
    va_pa_offset = PHYSICAL_MEMORY_OFFSET; // 设置虚拟到物理地址的偏移: 硬编码 0xFFFFFFFF40000000。va_pa_offset就是虚拟地址到物理地址的偏移量

    // 获取物理内存信息，下面变量表示物理内存的开始、大小和结束地址
    uint64_t mem_begin = KERNEL_BEGIN_PADDR; // 0x8020 0000 可存储内存的开始位置
    uint64_t mem_size = PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR;  //可以使用的内存区域的大小
    uint64_t mem_end = PHYSICAL_MEMORY_END; // 硬编码取代 sbi_query_memory()接口 0x8800 0000 就是内存区域的结束位置

    // 打印物理内存映射信息
    cprintf("physcial memory map:\n");
    cprintf("  memory: 0x%016lx, [0x%016lx, 0x%016lx].\n", mem_size, mem_begin,
            mem_end - 1);

    // 限制物理内存上限:
    uint64_t maxpa = mem_end;
    cprintf("maxpa: 0x%016lx.\n", maxpa); // test point

    // ctrl+左键点进去看一下KERNTOP具体实现（在memlayout.h中，KERNTOP是KERNBASE + KMEMSIZE）也就是0x88000000对应的虚拟地址
    if (maxpa > KERNTOP)
    {
        maxpa = KERNTOP;
    }

    // 初始化物理页面数组
    // end是链接脚本中定义的内核结束位置，其实是个常量指针
    extern char end[];

    // 求得总的物理页面数，用页面所占的内存大小除以页面大小
    npage = maxpa / PGSIZE;
    cprintf("npage: 0x%016lx.\n", npage); // test point,为0x8800_0
    cprintf("nbase: 0x%016lx.\n", nbase); // test point，为0x8000_0

    // kernel在0x8020 0000开始加载，在end[]结束，pages是剩下的页的开始，是一个指向物理页面数组的指针
    // ROUNDUP是一个宏或函数，将给定的地址向上舍入到最接近的 PGSIZE 边界。保证最后的指针指向4kB对齐的地址
    // 把page指针都指向内核所占内存空间结束后的第一页
    cprintf("end pythical address: 0x%016lx.\n", PADDR((uintptr_t)end)); // test point
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
```

```
    cprintf("pages pythical address: 0x%016lx.\n", PADDR((uintptr_t)pages)); //
test point
    // pages pythical address是0x8020 7000，有0x7000的位置被kernel映像占用

    // 一开始把所有页面都设置为保留给内存使用的，然后再设置那些页面可以分配给其他程序
    for (size_t i = 0; i < npage - nbase; i++)
    {
        SetPageReserved(pages + i); // 在memlayout.h中，SetPageReserved是一个宏，将
给定的页面标记为保留给内存使用的
    }

    // test ponit begin
    for (size_t i = 0; i < 5; i++)
    {
        cprintf("pages[%d] pythical address: 0x%016lx.\n", i, PADDR((uintptr_t)
(pages + i))); // test point
    }
    // test point end

    // 初始化空闲页面列表
    // PADDR 宏将这个虚拟地址转换为物理地址
    // 从这个地方开始才是我们可以自由使用的物理内存
    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * (npage -
nbase)); // 0x8034 7000 = 0x8020 7000 + 0x28 * 0x8000
    cprintf("page结构体大小: 0x%016lx.\n", sizeof(struct Page));
      // test point

    // 按照页面大小PGSIZE进行对齐，ROUNDUP，ROUNDDOWN是在libs/defs.h定义的
    mem_begin = ROUNDUP(freemem, PGSIZE);
    mem_end = ROUNDDOWN(mem_end, PGSIZE);
    cprintf("freemem: 0x%016lx.\n", freemem);     // test point
    cprintf("mem_begin: 0x%016lx.\n", mem_begin); // test point
    cprintf("mem_end: 0x%016lx.\n", mem_end);     // test point

    if (freemem < mem_end)
    {
        // 初始化可以自由使用的物理内存
        init_memmap(pa2page(mem_begin), (mem_end - mem_begin) / PGSIZE);
    }
    cprintf("mem_begin对应的页结构记录(结构体page)虚拟地址: 0x%016lx.\n",
pa2page(mem_begin));        // test point
    cprintf("mem_begin对应的页结构记录(结构体page)物理地址: 0x%016lx.\n",
PADDR(pa2page(mem_begin))); // test point

    cprintf("可用空闲页的数目: 0x%016lx.\n", (mem_end - mem_begin) / PGSIZE); //
test point
    // 可用空闲页数目0x7cb9 ，0x7cb9>>12 + 0x80347000（membegin）=0x88000000
（memend）
    // 从0x8800 0000到0x8000 0000总共0x8000个页，其中0x7cb9个页可用，也就是总共空闲页内存
是0x7cb9000，也就是124MB
    // 0x8000-0x7cb9=0x0347个不可用，这些页存的是结构体page的数据
}
```

内存块表管理（可能是快表？）

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。

> 也就是说，如果检测到TLB快表中，存在该块，就直接读就行，如果发现修改了部分值，那就用指令刷新整个TLB

虚拟地址各个二进制位的作用

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，**我们规定 63–39 位的值必须等于第 38 位的值**（大家可以将它类比为有符号整数），否则会认为该虚拟地址不合法，在访问时会产生异常。 不论是物理地址还是虚拟地址，我们都可以认为，**最后12位表示的是页内偏移，也就是这个地址在它所在页帧的什么位置（同一个位置的物理地址和虚拟地址的页内偏移相同）。除了最后12位，前面的部分表示的是物理页号或者虚拟页号。**

- 0-11位表示页内偏移
- 12-38位表示物理页号
- 39-63位必须等于第 38 位（规定）

Page结构体属性的作用

```
/* *
 * struct Page - 页描述符结构。每个 Page 描述一个物理页。
 在 kern/mm/pmm.h 中，你可以找到许多有用的函数，这些函数可以将 Page 转换为其他数据类型，例如物理地址。
 * */
struct Page {
    int ref;                        // 页帧的引用计数器
    uint64_t flags;                 // 描述页帧状态的标志数组
    unsigned int property;          // 空闲块的数量，在首次适应内存管理器中使用
    list_entry_t page_link;         // 空闲列表链接
};
```

Page和page_link的关系

```
struct Page {
    int ref;                          // 页帧的引用计数器
    uint64_t flags;                   // 描述页帧状态的标志数组
    unsigned int property;            // 空闲块的数量，在首次适应内存管理器中使用
    list_entry_t page_link;           // 空闲列表链接
};
```

Page是一个结构体

page_link是该结构体的一个成员，相当于空闲列表的链接

如何通过page_link获取Page的方法（具体到函数）

如何判断一个Page后面有几个空闲页（具体到变量），如果不是第一个那个变量应该是多少

怎么切换不同管理器（具体到代码）

pmm.c文件

调用接口的函数：可以选择自己使用的接口

```
// init_pmm_manager - 初始化一个内存管理器的方式
static void init_pmm_manager(void) {
    //pmm_manager = &best_fit_pmm_manager;
    pmm_manager = &buddy_system_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

DRAM 空间从哪里到哪里

默认的 DRAM 物理内存地址范围就是 `[0x80000000,0x88000000)`

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

#define KERNBASE            0xFFFFFFFFC0200000
#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
#define KERNEL_BEGIN_VADDR          0xFFFFFFFFC0200000
```

DRAM 空间都存了什么

但是，有一部分 DRAM 空间已经被占用，不能用来存别的东西了！

- 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；
- 其实设备树扫描结果 DTB 还占用了一部分物理内存，不过由于我们不打算使用它，所以可以将它所占用的空间用来存别的东西。

于是，我们可以用来存别的东西的物理内存的物理地址范围是：`[KernelEnd, 0x88000000)` 。这里的 `KernelEnd` 为内核代码结尾的物理地址。在 `kernel.ld` 中定义的 `end` 符号为内核代码结尾的虚拟地址。

> 最前面存了opensbi，中间存了内核的代码段和数据段

```
pages pythical address是0x8020 7000，有0x7000的位置被kernel映像占用
可用空闲页数目0x7cb9 ，0x7cb9>>12 + 0x80347000（membegin）=0x88000000（memend）
从0x8800 0000到0x8000 0000总共0x8000个页，其中0x7cb9个页可用，也就是总共空闲页内存是
0x7cb9000，也就是124MB
0x8000-0x7cb9=0x0347个不可用，这些页存的是结构体page的数据
```

所以：范围为**0x80347000（membegin）到0x88000000（memend）**

怎么获取可用物理内存大小

见上一个问题