Lab2 物理内存和页表

实验一解决了“启动”的问题，实验二就要开始解决操作系统的物理内存管理的问题。我们需要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。

练习1：理解first-fit连续物理内存分配算法

> first-fit 连续物理内存分配算法是基础的方法，需要理解它的实现过程。仔细阅读 `kern/mm/default_pmm.c` 相关代码，认分析 `default_init` `default_init_memmap` `default_alloc_pages` `default_free_pages` 中相关函数，描述程序在物理内存分配过程及各个函数的作用。简要说明设计实现过程。

- `default_init()`
  - `` `list_init(&free_list);nr_free=0; ``

    初始化free_list，将其置为空链表，将物理页面数置为0

- `default_init_memmap(struct Page *base, size_t n)`：创建以 base为起始地址的连续n个页。将一段连续的物理页面组织成一个链表，并在初始化过程中设置页面的标志位、属性值和引用计数。它还更新了全局变量 `nr_free`，记录了可用的物理页面数量。最后，根据链表 `free_list` 的状态，将 `base` 页面插入到合适的位置，以维护链表的有序性。具体来说：
  - 对于[base,base+n)的页面，如果页面不是为内核保留，则将flag和property、ref置0
  - 设置base的property为n，设置起始块的标志位
  - 增加free_page的数量参数nr_free
  - 将base->page_link插入free_page当中
    - 如果free_list空，插入链表头
    - 否则，遍历free_list,将 base->link插入到base<page的第一个le之前
- `default_alloc_pages(size_t n)`：从可用的物理页面链表中分配 `n` 个连续的页面，更新链表和全局变量以反映已分配的页面数量，然后返回已分配页面的起始地址。如果无法满足分配请求，函数返回 `NULL`。具体来说：
  - 检查n是否合法，assert(n>0&& n<nr_free)即，想要分配的页数不大于当前的free_pages
  - 依次遍历free_list，如果有连续free_pages的数目不小于n，就把这个页单独摘出来
  - 摘出来的页，如果有多的部分，切好再放回去
  - 动态调整参数，如nr_free-=n
- `default_free_pages(struct Page * base,size_t n)`：释放一连串物理页面，更新链表和全局变量以反映释放的页面空闲，同时合并相邻的空闲页面，并将释放的页面插入到合适的位置。具体来说：
  - assert(n>0)
  - 对于连续的[base,base+n)的页，必须是!**PageReserved**(p) && !**PageProperty**(p)，才能对p->flags和p->ref释放为0
  - 将释放的页串入free_list当中
    - 如果free_list空链表，直接将base->page_link放进去
    - 否则，遍历free_list,将base->page_link放到第一个满足base<page的页前面

- 合并可以合并的
  - 观察base->page_link在free_list当中的前向le，若le不是free_list的首元素地址且le所对应的页与base页面相邻，则合并，并且在free_list上删除base->page_link
- ，清除后一个页面的属性标志位，并将后一个页面从空闲页面列表中删除（`list_del`函数）

> 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：你自己的first-fit有没有进一步的改进空间?

改进空间：可以建立AVL搜索树以减少搜索时间；引入页表级别的位图记录页面使用情况以更方便的找到可用的连续页面块。

练习2：实现Best-Fit连续物理内存分配算法

> 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?

> 简述设计实现过程

顾名思义，best_fit算法就是根据请求的数量分配最适合的大小的空闲块。实现这一个功能是较为简单的，于first_fit相比只需要修改分配函数中的判断截至条件即可。在first_fit中，当空闲块大于等于请求大小时，即停止查找；但在best_fit中，我们需要找到大于等于请求大小的空闲块中最小的那一个，我们仅仅是将循环条件修改为如下所示，即实现了这一功能：

```
static struct Page *
best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;
     /*LAB2 EXERCISE 2: YOUR CODE*/
    // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
    // 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，记录该页面以及当前找到的最小连续空闲页框数量
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_size) {
            page = p;
            min_size = p->property;
            //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;
}
```

可见，我们使用min_size记录当前最小空闲块的大小，以达查找到的空闲块比此前的小且满足请求大小，则将结果赋为当前查找到的空闲块。对于其他函数或代码，与default_pmm.c中的定义完全一致，这里不再赘述。

不同点在于，Best-Fit需要遍历free_list找到最合适的（即：p->property>=n的基础上，还要是最小的）。因此增加一个这个遍历就可以了。

> 是否有进一步的改进空间

1.搜索策略优化。可以先将内存块按照大小排序，之后再进行有效搜索。为了实现较好的搜索策略，我们可以建立AVL搜索树，并及时维护。（不过有个问题：我们的这个best_fit的实现思路很简单，同时它的代价也是很高的——每次分配时都需要遍历所有的空闲块，这个复杂度是O(n)。我们可以对这个空闲链表按照页数量进行排序，从而在查找时将复杂度降为O(logn)。但这里还有一个问题，排序需要花费大量时间（O(nlogn)），这是需要考虑的。

2.碎片整理。当有大块的空闲内存块释放时，可以对已分配的内存进行碎片整理，尽可能地合并相邻的空闲内存块，减少内存碎片。

3.内存回收策略：可以考虑采用更加智能的内存回收策略，例如LRU（最近最少使用）算法，对长时间未被访问的内存块进行回收。

知识点梳理

1、物理内存管理动机与技术

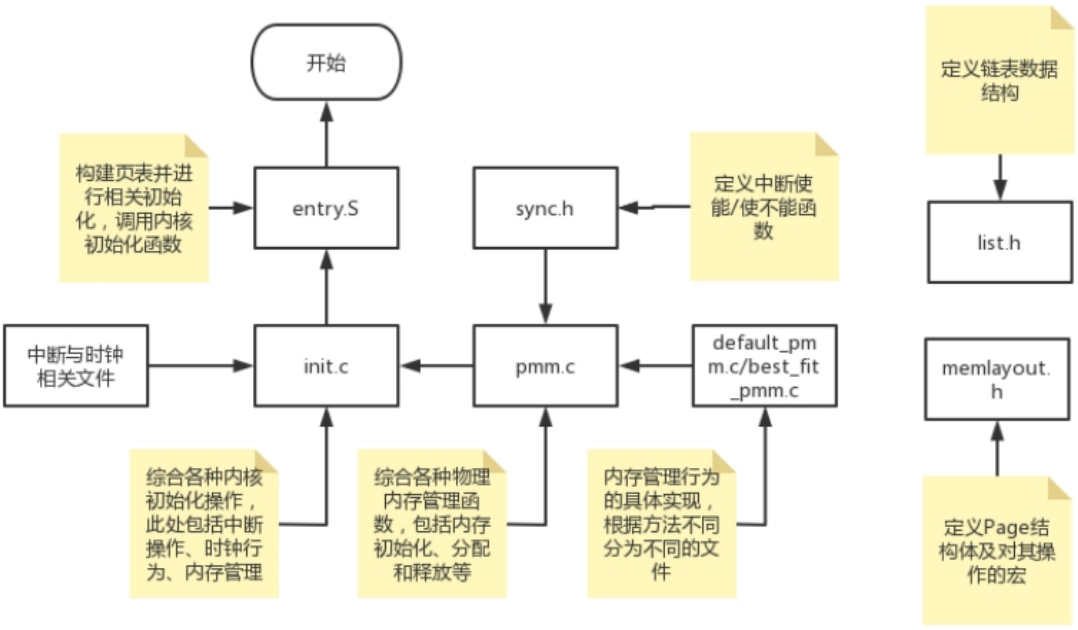
（1）内存是一种稀缺资源，计算机需要管理内存以优化性能；在本次实验中（同时也是当前流行的技术），我们使用的内存管理方法主要包含两个技术——分页存储技术和虚拟内存技术。需要注意的是，前者同时也是后者的条件。

（2）如果将一个进程连续地放入内存，则会产生很多碎片从而降低了内存使用率。分页技术实际上基于了离散分配的思想，将进程分为不同的小块来存储，这样可以减少内存中的碎片。分页后就会产生一个问题，就是进行中的代码在内存中对应哪些块，这就需要页表来解决。

（3）如果一个进程的空间很大以至于超过整个内存（如大型游戏），就需要使用虚拟内存技术实现逻辑上的内存扩大。基于局部性原理，一个进程的所有代码并不需要在短时间内均被运行，一个逻辑上大地址的指令在物理上也可以是一个较小的地址。虚拟技术需要实现信息的调入和换出，同时也需要通过页表实现虚拟地址和物理地址的对应即可。

2、项目主要文件结构及执行流

本次实验主要关注操作系统的物理内存管理，涉及的主要文件包括总体性文件（entry.S, init.c）和内存管理相关文件（pmm.c, default_pmm.c, memlayout.h, sync.h, list.h等），这些文件间的关系及执行流如下图所示：



相关文件详细功能说明如下

entry.S

构建页表并进行相关初始化，调用内核初始化函数。

此前的实验我们直接采取物理地址（satp的MODE为0000表示使用物理地址），本次实验我们使用了虚拟地址（在kernel.ld文件中将基地址修改为0xFFFFFFFFC0200000，并将satp的MODE设置为0100表示使用虚拟地址）；但此时的修改并没有使计算机实现逻辑地址和物理地址间的对应关系，因此我们需要在这个文件中建立一个翻译机制（页表）来实现这个功能。

我们在数据段中定义了所使用的三级页表。在代码段中，调用kern_init之前需要获取到其实际物理地址，根据此前分析，依次进行——页表获取（获取三级页表虚拟地址、计算其物理地址、获取其物理页号）、设定内存管理模式（设置satp的MODE为0100）、刷新TLB。这样，在调用kern_init时就可以使用它的虚拟地址了。

init.c

综合各种内核初始化操作，此处包括中断操作、时钟行为、内存管理。

这个文件和此前的相比，只是多了本次实验的物理内存管理初始化的操作，即函数pmm_init()，它在pmm.c中被定义。

pmm.c

综合各种物理内存管理函数，包括内存初始化、分配和释放等。

该文件中的主要函数是pmm_init()（初始化管理者、页初始化page_init、测试分配功能）、alloc_pages()（分配页）、free_pages()（释放页）、nr_free_pages()（查询空闲页数量）。这些函数大都是调用了default_pmm.c/best_fit_pmm.c中封装好的对应功能的函数，这是因为物理内存管理可以基于不同的实现方法，default_pmm.c和best_fit_pmm.c等是根据不同方法定义的功能实现。

default_pmm.c/best_fit_pmm.c

定义内存管理初始化、内存映射初始化、分配页、释放页、查询空闲页数量等功能的函数。需要使用list.h中定义的链表方法和memlayout.h中定义的Page结构体及其宏操作。

sync.h

定义中断使能/使不能函数，在pmm.c的分配&释放函数中被调用，以确保内存管理修改相关数据时不被中断打断。

list.h

定义链表数据结构，这是基于链表方法的物理内存管理的基础。

memlayout.h

定义Page结构体，用以连成链表，并定义了一系列对其操作的宏。

3、以页为单位管理物理内存

页表项

在`sv39`中，一个页表项占据8B，其结构如下：

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							

PPN：Physical Page Number

RSW: Reserved for S Mode，留给Smode的应用程序，用来进行拓展

D：Dirty 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 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 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: U=1表示用户态程序可以通过该页表项映射。S 模式不可以通过U=1的页表项映射，除非S Mode的寄存器sstatus 上的SUM位手动设置为1.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 许可位，可读可写可执行。V=0时，该页表项不合法，页表项其他位的值被忽略！

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

多级页表

一级页表所构造的虚拟空间有限，需要引入多级页表实现更大规模的虚拟地址空间。

在`sv39`中，一个页表项占8B，虚拟地址有39bit，后面12位都是页内偏移，因此还剩下64-39-12=27bit编码不同的虚拟页号。

如果给2^27虚拟页号都分配8B的页表项，其中*Pagetable[vpn]*代表 虚拟页号为*vpn*的虚拟页 的页表项。如此，便会有 1GiB的内存。 但是其中有很多的虚拟地址没有用到，因此会有大片的页表项的标志位为0，如此便浪费了大量的内存空间。

因此，页表分级可以解决这一问题。将很多页表项组合成为大的页。如果某些页表项都没有对应的物理页（非法），那么只需用一个非法的页表项来覆盖这一大的页，而不需用建立一大堆非法页表项。

在Lab2中，权衡`sv39`，使用三级页表分别为： 4KB的页，2MB的页，1GB的页。

> 39=27+12=（9+9+9）+12

之前39位的虚拟地址，被我们拆成27位的页号和12位的页偏移。

在三级页表下，我们把它看成 9位的大大页号，9位的大页号，9位的页号，12位的页内偏移。因此，在整个虚拟空间中，有512=2^9个大大页，每个大大页有512个大页，每个大页有512个页。每个页有4KB的大小。一页是4096Byte，一个页表项是8Byte。一页正好可以4096/8=256个页表项！

> 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。

每一级页表项控制一个虚拟页号，4KB的虚拟内存。每二级页表项控制4KB*2^9=2MB的虚拟内存；每三级页表项控制2MB *2^9=1GB的虚拟内存。 如果二级页表项设置为RWX不全为0，那么它将会与一级页表类似，只不过可以应设一个2MB的大页；同理也可以将三级页表项看作是一个叶子，映射1GiB的大大页

页表基址

在翻译的过程中，我们需要知道根节点的物理地址。

- `stap`寄存器 Supervisor Address Translation and Protection Register 的CSR
  - 最高级页表的所在的物理页号，并不是其对应的物理起始地址
  - MODE(WARL)63~60=4
  - ASID(WARL)59~44=16
  - PPN(WARL)43~0=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。
- 0100表示sv39页表，也就是我们使用的，虚拟内存空间高达`512GiB`。
- 0101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。

建立快表访问

根据上述三级页表体制，访问一次物理内存就要访问3次物理内存来求得物理地址，之后才能访问到物理内存。

利用时空局部性，我们可以使用TLB (Translation Lookaside Buffer)块表来记录刚刚找到的物理页号与虚拟页号的映射。先去TLB查找，有的话就可以直接找到物理地址，很快。但是如果修改了`satp`寄存器（eg：将PPN字段进行修改）此时块表当中存储的映射结果与实际映射结果就会不同。这种情况下需要`sfence.vma`指令刷新整个TLB。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。

4、分页设计思路

实现分页

内核初始化的修改：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。

> 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。
>
> ```
> // tools/kernel.ld
> BASE_ADDRESS = 0xFFFFFFFFC0200000;
> //之前这里是 0x80200000
> ```
>
> 修改了链接脚本中的起始地址。

- 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当中。
- CPU状态： 处于SMode。寄存器satp 的MODE 被设置为Bare。无论取指还是访存，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`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，所以他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当中，并以此为依据确定代码里的每个部分的地址。（每一段都是`BASE_ADDRESS`往后依次摆开，代码里的各个段都会认为自己在0xff地址后的某个地址上，或者说，编译器和链接器会把符号/变量地址都对应到0xf之后的某个地址上）

> 接下来修改了`entry.S`文件来实现内核的初始化。
>
> 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；同时需要跳到 kern_init之中
>
> ```
> #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 ; 自己分配了一块16KB的内存作为启动栈
>  .align PGSHIFT
> ```

```
  .global bootstack
bootstack:
  .space KSTACKSIZE
  .global bootstacktop
bootstacktop:
```

> `bootstacktop`就是我们需要的栈顶地址，`kern_init`代表了要跳转到的地址

之间，直接将`bootstacktop`的值传给sp，再跳转到`kern_init`就可以了。但是如今已经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的地址为0xffffffffc0200000，`bootstacktop`和`kern_init`会被翻译成比这个开头地址还要高的虚拟地址。但是！由于CPU处于Bare模式，会将地址当成物理地址进行处理，此时就会出错！

因此，我们需要利用页表的知识，帮助内核将虚拟地址空间构造出来。即：构建一个合适的页表，`satp`指向这个页表，使用地址的时候要经过页表的翻译，使得 结果为`0x80200000`。

所有的虚拟地址都有一个固定的偏移量，相对虚拟地址就是这个偏移量。

`p_addr+虚实映射偏移量=v_addr`

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[0x80000000,0xc0000000)。此时需要分配一页内存来存放三级页表，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

5、物理内存探测

在risc-v中，bootloader （OpenSBI）可以通过对包括物理内存在内的各种外设的扫描，探测物理内存所在的物理地址。扫描结果以 DTB的格式保存在物理内存中，其地址保存在`a1`寄存器。

QEMU定义的DRAM物理内存的物理地址为`0x80000000`。使用`-m`指定RAM的大小，默认是128MiB。因此，DRAM物理内存地址的范围就是[0x80000000,0x88000000)。将DRAM物理内存结束地址硬编码到内核中。

```
// 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
```

在DRAM中，`[0x80000000,0x80200000)`被 OpenSBI 占用；`[0x80200000,KernelEnd)`被内核各代码与数据段占用；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。因此，`[KernelEnd, 0x88000000)`可以用来存别的东西。

> 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。

使用`Page`结构体描述物理页面的使用情况。将结构体在内存中排列在内核后面。注意，摆放Page结构体的物理页面和内核占用的物理页面之后都无法使用！