

实验准备

对于启动过程的理解

第一段

第二段

第三段

第四段

第五段

第六段

第七段

entry.S:设置satp寄存器

entry.S:设置sp后准备跳转

entry.S其他设置

kern_init的代码

pmm流程分析

练习1: 理解first-fit 连续物理内存分配算法 (思考题)

对于流程的理解

对于空闲页块链表freelist的理解

对于default_init的理解

对于页块page的理解

对于default_init_memmap的理解

对于default_alloc_pages的理解

对于default_free_pages的理解:

优化方案

练习2: 实现 Best-Fit 连续物理内存分配算法 (需要编程)

default_init

default_init_memmap

代码填空1

代码填空2

代码填空3

代码填空4

代码填空5

通过验证

优化方案

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

扩展练习Challenge: 任意大小的内存单元slub分配算法 (需要编程)

扩展练习Challenge: 硬件的可用物理内存范围的获取方法 (思考题)

实验准备

在lab1可以运行的前提下, 确保makefile里面的qemu可以工作, 如果makefile被修改错了, 粘贴原始lab2里面的makefile进行覆盖。

中间可能提示 fatal error: -fuse-linker-plugin, but liblto_plugin.so not found, 需要在工具链目录下找到 liblto_plugin.so.0.0.0 复制成一份 liblto_plugin.so 顺利解决

开始实验：修改makefile里面的opensbi的版本为正确版本,修改bios对应文件为GitHub下载下来的文件:

```
hpl@hpl:~/mnt/hgfs/shared_file/OperatingSystem/uboot-github/lab2$ cat Makefile
$(UCOREIMG) $(SWAPIMG) $(SFSIMG)
$(V)$(QEMU) \
-machine virt \
-nographic \
-bios /mnt/hgfs/shared_file/OperatingSystem/opensbi/fw_jump.bin \
-device loader,file=$(UCOREIMG),addr=0x80200000

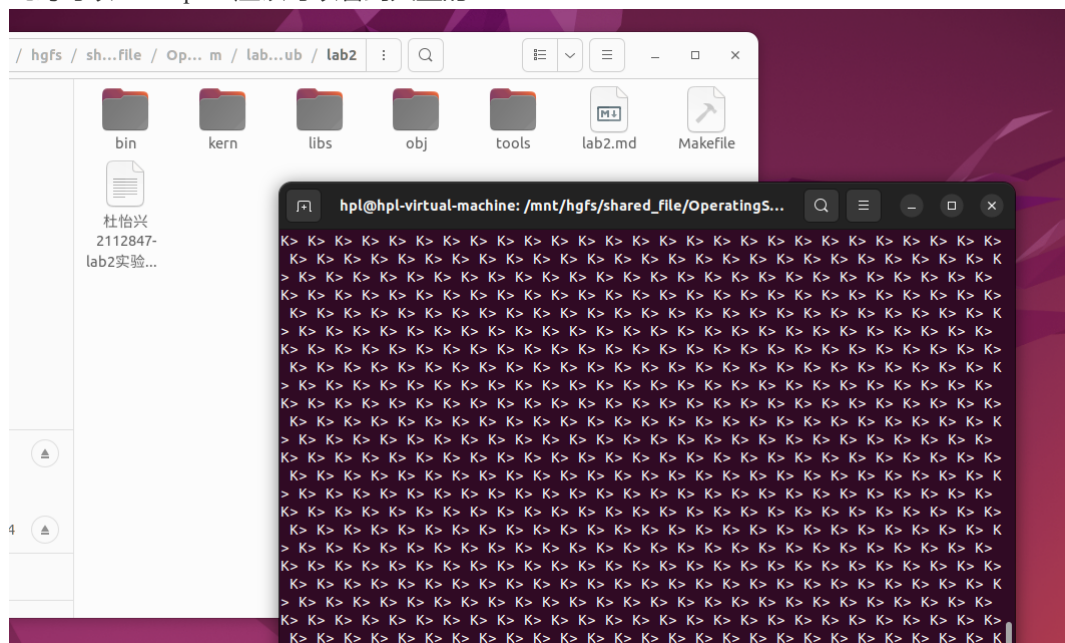
ig: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
$(V)$(QEMU) \
-machine virt \
-nographic \
-bios /mnt/hgfs/shared_file/OperatingSystem/opensbi/fw_jump.bin \
-device loader,file=$(UCOREIMG),addr=0x80200000 \
-s -S

riscv64-unknown-elf-gdb \
-ex 'file bin/kernel' \
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'

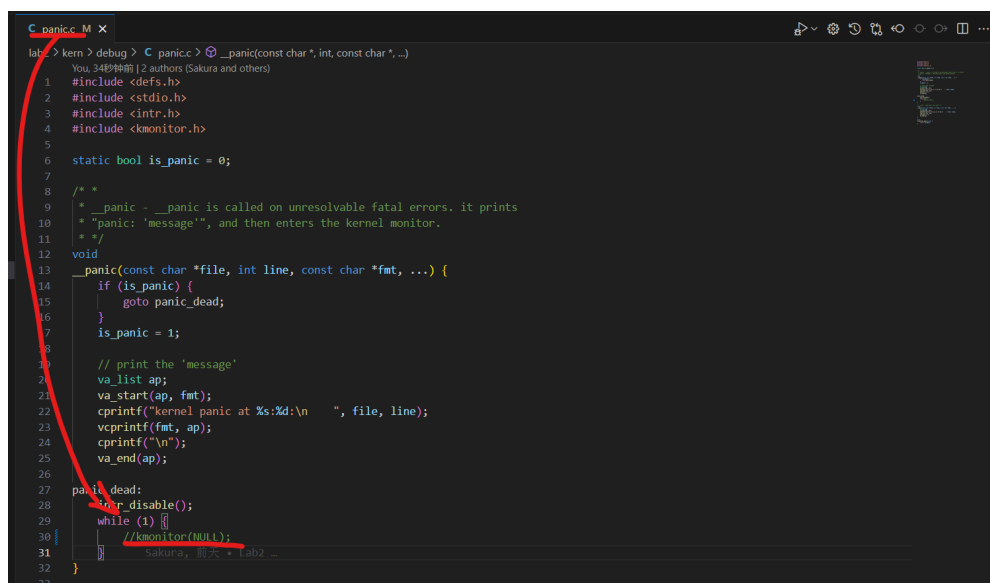
: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
$(V)$(QEMU) \
-machine virt \
-nographic \
-bios /mnt/hgfs/shared_file/OperatingSystem/opensbi/fw_jump.bin \
-device loader,file=$(UCOREIMG),addr=0x80200000

e: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
$(V)$(SPIKE) $(UCOREIMG)
```

此时可以make qemu应该可以看到大量的kkkk



为什么出现kkk呢，因为debug的panic.c里面故意设置了一个for循环输出k，把它注释掉就行



此时再make qemu,可以看到opensbi被启动了:

```
hpl@hpl-virtual-machine: /mnt/hgfs/shared_file/OperatingSystem/labs-github/lab2
+ cc kern/debug/panic.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

OpenSBI v1.3.1

Platform Name      : riscv-virtio,gemu
Platform Features  : medeleg
Platform HART Count : 1
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 100000000Hz
Platform Console Device : semihosting
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : sifive_test
Platform Shutdown Device : sifive_test
Platform Suspend Device : ---
Platform CPPC Device : ---
Firmware Base      : 0x80000000
Firmware Size       : 194 KB
Firmware RW Offset : 0x20000
Firmware RW Size     : 66 KB
Firmware Heap Offset : 0x28000
Firmware Heap Size   : 34 KB (total), 2 KB (reserved), 9 KB (used), 22 KB (free)
Firmware Scratch Size : 4096 B (total), 760 B (used), 3336 B (free)
Runtime SBI Version : 1.0

Domain0 Name       : root
Domain0 Boot HART  : 0
Domain0 HARTs      : 0*
Domain0 Region00   : 0x0000000002000000-0x000000000200ffff M: (I,R,W) S/U: ()
Domain0 Region01   : 0x0000000008000000-0x0000000008001ffff M: (R,X) S/U: ()
Domain0 Region02   : 0x00000000080020000-0x0000000008003ffff M: (R,W) S/U: ()
Domain0 Region03   : 0x00000000080000000-0xffffffffffffffff M: (R,W,X) S/U: (R,W,X)
Domain0 Next Address : 0x00000000080200000
Domain0 Next Arg1   : 0x00000000082200000
Domain0 Next Mode    : S-mode
Domain0 SysReset    : yes
Domain0 SysSuspend  : yes
```

然后了解一下库函数list后面有用

```
Sakura, 前天 | 1 author (Sakura)
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));

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));
```

list是一个双向链表, 可以被struct list_entry理解为node,又把struct list_entry简写为list_entry_t

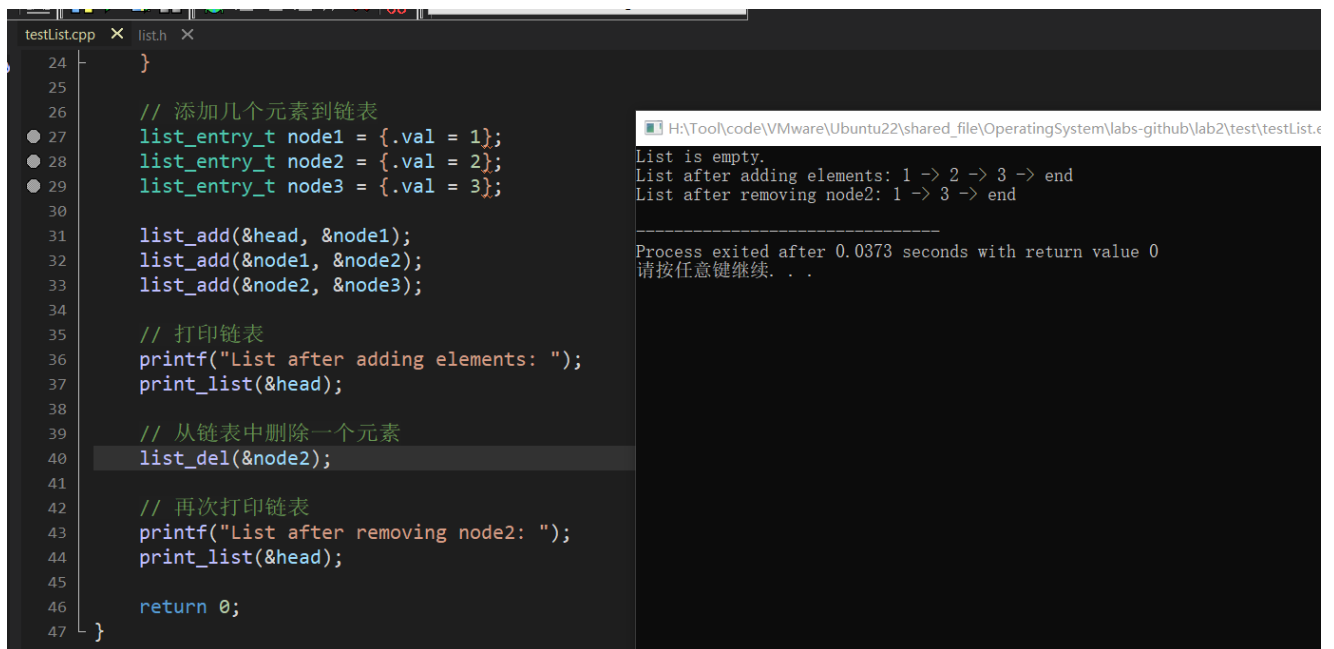
其中这里的add很迷, __list_add是把elm插入节点prev和next之间。list_add_after是把elm插入listelm和listelm->next之间, 和list_add一模一样。而list_add_before是把elm插入listelm和listelm->pre之间。init是让前后指针指向自己。

__list_del是让pre和next指向彼此，list_del是删掉一个元素，list_del_init是删掉元素，让它自己指向自己，list_empty通过是否指向自己判断是不是空的，list_prev和list_next是获取list前后元素的接口。

为了测试list，这里写了一个testlist放在test文件夹里面

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <stdlib.h>
4 #include "list.h"
5 // [将上面的list.h代码复制到这里]
6
7 void print_list(list_entry_t *list) {
8     list_entry_t *itr = list_next(list); // 获取首个元素,注意头节点里面没有值
9     while (itr != list) {
10         printf("%d -> ", itr->val);
11         itr = list_next(itr);
12     }
13     printf("end\n");
14 }
15
16 int main() {
17     // 创建一个头结点
18     list_entry_t head;
19     list_init(&head);
20
21     // 确认链表为空
22     if (list_empty(&head)) {
23         printf("List is empty.\n");
24     }
```

可以实现基本的插入删除



```
24 }
25
26 // 添加几个元素到链表
27 list_entry_t node1 = {.val = 1};
28 list_entry_t node2 = {.val = 2};
29 list_entry_t node3 = {.val = 3};
30
31 list_add(&head, &node1);
32 list_add(&node1, &node2);
33 list_add(&node2, &node3);
34
35 // 打印链表
36 printf("List after adding elements: ");
37 print_list(&head);
38
39 // 从链表中删除一个元素
40 list_del(&node2);
41
42 // 再次打印链表
43 printf("List after removing node2: ");
44 print_list(&head);
45
46 return 0;
47 }
```

H:\Tool\code\VMware\Ubuntu22\shared_file\OperatingSystem\labs-github\lab2\test\testList.e

List is empty.
List after adding elements: 1 -> 2 -> 3 -> end
List after removing node2: 1 -> 3 -> end

Process exited after 0.0373 seconds with return value 0
请按任意键继续. . .

对于启动过程的理解

和lab1是一致的，刚开始启动在0x1000,之后跳转到0x80000000,这一段执行力opensbi程序，相当于bios，然后跳转到0x80200000【此时所有的输出opensbi完毕】,去执行内核程序。

```
Type "apropos word" to search for commands related to "word".
Reading symbols from bin/kernel...done.
The target architecture is assumed to be riscv:rv64
Remote debugging using localhost:1234
0x0000000000001000 in ?? (
(gdb) break *0x80200000
Breakpoint 1 at 0x80200000
(gdb) continue
Continuing.

OpenSBI v1.3.1

OpenSBI
```

而内核程序的执行顺序是：先执行entry.S

但是这一次entry.S和lab1有了很大不一样，在此之前，我们需要阅读实验手册：

5.1.3.1.1 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，不进行任意的管理操作，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，然后在词典里查找翻译后的地址，而对每个程序往往都有着一本“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远远小于于字节，至少几 KB 量级（所以七十年代的人说的是 64KB 对每个人都够了“而不是 64MB 对每个人都够了”），那么我们可以考虑，把连成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”。

第一段

说明了如果没有适当的物理内存管理，所有程序，无论是内核还是用户程序，都将共享同一个地址空间，从而产生冲突。

解决方法是引入一个“翻译”机制，其中程序使用的虚拟地址在访问物理内存之前必须被“翻译”成物理地址。

如果为每个字节都提供一个单独的翻译项，这个字典将变得非常大，这将导致效率低下。

为了提高效率，内存被分组到固定大小的块中，这些块称为“页”。这个映射是以页为单位的。

第二段

说明了使用的是 RISC-V 架构下的 sv39 页表机制。每页大小为 4KB，即 4096 字节。

页表是一个“词典”，它存储了虚拟地址（程序看到和使用的地址）与物理地址（真实内存硬件上的地址）之间的映射。

虚拟页的数量可能远大于物理页的数量，这意味着并非所有的虚拟地址都有与之对应的物理地址。

在 sv39 页表机制中，物理地址有 56 位，而虚拟地址有 39 位。尽管虚拟地址有 64 位，但只有低 39 位是有效的。位于 63-39 位的值必须与第 38 位的值相同，否则这个虚拟地址被认为是非法的，访问它会触发异常。

无论物理还是虚拟地址，最后的 12 位代表页内偏移，即这个地址在其所在页的位置。因为一个页大小是 $4B \cdot 2^{12}$ ，也就是页内字节地址需要 12 位表达。除了最后 12 位，前面的位数代表物理页号或虚拟页号。

5.1.3.1.1.2 物理地址和虚拟地址

在本次实验中，我们使用的是 RISC-V 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于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（大家可以把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。

第三段

lab2和lab1启动有所不同

- 在启动时，bootloader 不再像 lab1 那样直接调用 kern_init 函数。
- 它首先调用 entry.S 中的 kern_entry 函数，其任务是为 kern_init 函数建立良好的 C 语言运行环境，设置堆栈，并临时建立段映射关系，为后续的分页机制建立做准备。
- 完成这些后，才调用 kern_init 函数。
- kern_init 函数首先完成一些输出和检查 lab1 的实验结果，然后开始物理内存管理的初始化，即调用 pmm_init 函数。
- 完成物理内存管理后，它会进行中断和异常的初始化，调用 pic_init 函数和 idt_init 函数。

第四段

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。这个“词典”存储在内存中，由固定格式的“词条”（即页表项）组成。

在 sv39 中，每个页表项大小为 8 字节（64 位）。

- 位 63-54：保留位
- 位 53-10：物理页号（共 44 位）
- 位 9-0：映射的状态信息（共 10 位）

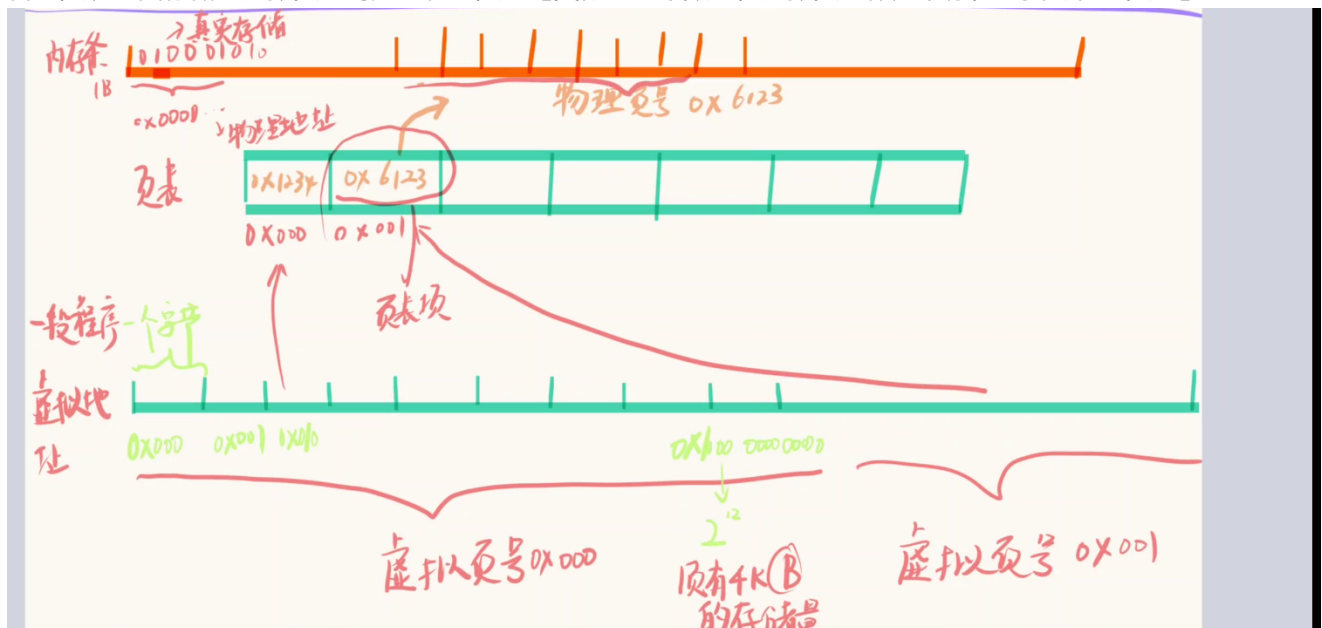
63-54	53-28	27-19	18-10	9-8	7	6	5	4	3	2	1	0
<i>Reserved</i>	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

- RSW：为 S Mode 的应用程序预留的两位，可用于拓展。
- D (Dirty)：标识页表项是否已被写入。
- A (Accessed)：标识页表项是否已被读取或写入。
- G (Global)：标识页表项是否为“全局”的，即所有的地址空间都包含这一项。
- U (User)：标识是否允许用户态程序使用该页表项进行映射。
- R,W,X：分别标识该页是否可读、可写、可执行。
- 根据 X,W,R 的不同组合，映射的类型有：

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 (<i>reserved for future use</i>)
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 (<i>reserved for future use</i>)
1	1	1	这一页可读可写可执行

第五段

普通页表里面存储了虚拟页号对应的物理页号【类似于一个数组，虚拟页号作为下标，查找到物理页号】。

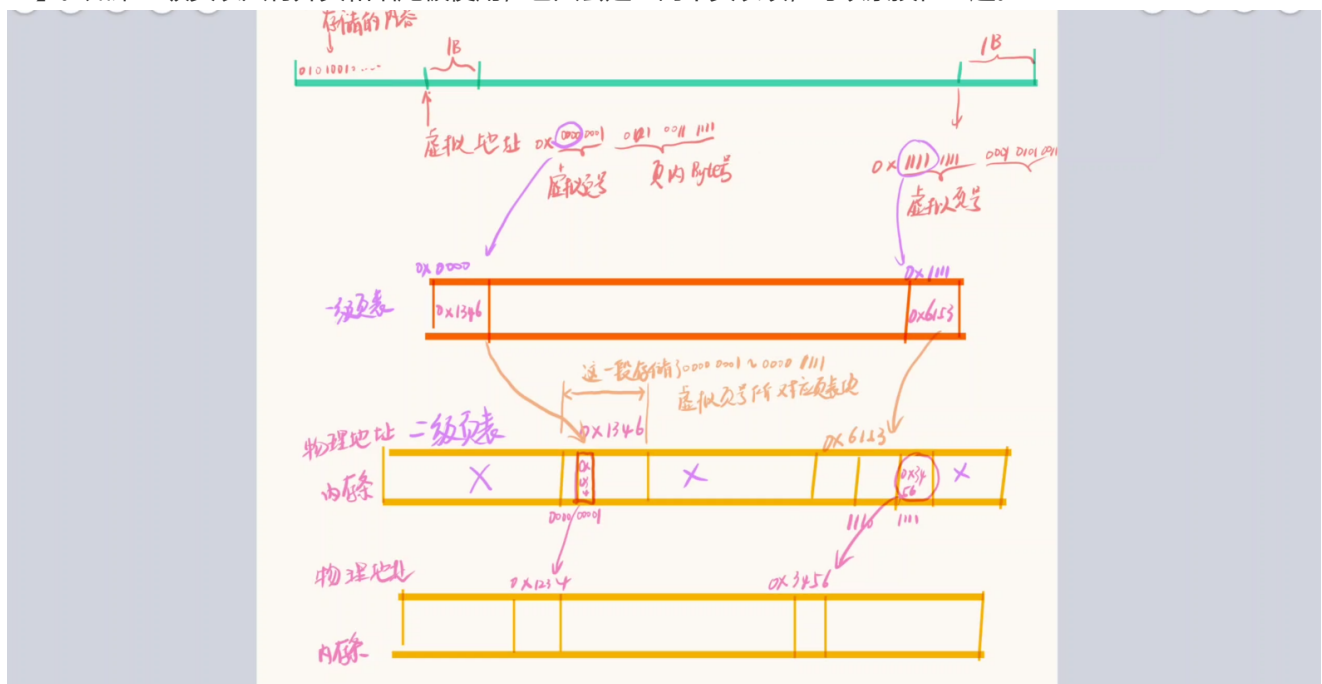


多级页表是为了解决当虚拟地址空间非常大时，单级页表可能会非常浪费空间的问题。【例如对于39位的虚拟地址， $39-12=27$ ，可以分成 2^{27} 个(虚拟)页，每个虚拟页都要映射到一个物理页，而存储这个物理页号需要8bit，意味着仅仅是存储一个页表，居然都要1GB，页表还是存储在内存里的，这显然不现实】

为什么二级页表更高效:

如果是只有一级页表，它必须是连续的，就像一个数组，用虚拟页号作为索引，得到物理页号，就导致了只有开始和结束部分的虚拟地址被使用时，也必须构建一个很大的数组。

但是如果是多级页表，就可以不连续，假设要找到二级页表中虚拟页号0x100对应的物理页号0x1234的映射关系，会使用虚拟地址的某些高位，在一级页表中查找。这些高位会告诉你该虚拟地址可能位于哪个二级页表中【物理地址】。如果二级页表只有开头和结尾被使用，也只会建立两个页表项，可以紧挨在一起。



第六段

satp, 即Supervisor Address Translation and Protection Register, 根页表的起始地址寄存器。

其中高4位还存储了页表的模式

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。
- 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。

第七段

TLB 快表 (Translation Lookaside Buffer): 类似于页表的cache

修改satp (尤其是PNN段) 或者修改页表项, 都会导致TLB不同步, 可以通过sfence.vma指令来刷新TLB, 没有参数, 它会刷新整个TLB。如果提供了一个虚拟地址作为参数, 则只会刷新该地址的映射。

entry.S:设置satp寄存器

在lab1当中, entry.S的工作特别简单, 就是分配栈空间, 设置栈指针, 跳转到kern_init。而内核程序是放在0x8020 0000上面的,

而lab2目标是将内核代码从物理地址空间迁移到虚拟地址空间, 并让它运行在一个特定的高地址空间, 即0xffffffffc0200000。

在0xffffffffc0200000打上断点, 会发现它对应着entry.S的第一句话,也就是说执行完启动程序opensbi【还是在0x8000 0000 ~ 0x8020 0000】之后, 去执行虚拟地址0xffffffffc0200000的内容。

```
Boot HART Priv Version : v1.10
Boot HART Base ISA : rv64imafdc
Boot HART ISA Extensions : time
Boot HART PMP Count : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count : 0
Boot HART MIDELEG : 0x0000000000000222
Boot HART MEDELEG : 0x0000000000000b109

Breakpoint 1, 0x0000000080200000 in ?? ()
(gdb) break *0xffffffffc0200000
Breakpoint 2 at 0xffffffffc0200000: file kern/init/entry.S, line 8.
(gdb) c
Continuing.

Breakpoint 1, 0x0000000080200000 in ?? ()
```

整个内核加载在虚拟地址0xffffffffc0200000开头的位置上,其中0xffffffffc0200036对应着kern_init

```
(THU.CST) os is loading ...
Special kernel symbols:
  entry 0xffffffffc0200036 (virtual)
  etext 0xffffffffc02017fa (virtual)
  edata 0xffffffffc0206010 (virtual)
  end 0xffffffffc0206070 (virtual)
Kernel executable memory footprint: 25KB
memory management: best_fit_pmm_manager
physical memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
```



```
(gdb) break *0xfffffffffc0200036
Breakpoint 1 at 0xfffffffffc0200036: file kern/init/init.c, line 18.
(gdb)
```

而执行kern_init之前, 还会执行0xfffffffffc0200000~0xfffffffffc0200036这一段, 这一段就是entry.S

```
1 #include <mmu.h>
2 #include <memlayout.h>
3
4 .section .text, "ax", %progbits
5 .globl kern_entry
6 kern_entry:
7     # t0 := 三级页表的虚拟地址
8     lui    t0, %hi(boot_page_table_sv39)
9     # t1 := 0xffffffff40000000 即虚实映射偏移量
10    li      t1, 0xffffffffc0000000 - 0x80000000
11    # t0 减去虚实映射偏移量 0xffffffff40000000, 变为三级页表的物理地址
12    sub     t0, t0, t1
13    # t0 >>= 12, 变为三级页表的物理页号
14    srli    t0, t0, 12
15
```

而要理解entry.S,还需要理解lui, 将立即数加载到寄存器高位, 寄存器低位置为0.

%hi:取一个地址的高20位li将立即数加载到寄存器, srli:将一个寄存器逻辑右移i位。Sv39:risc-V虚拟内存调用模式, 有39位虚拟内存地址, csrw写入csr寄存器。

可以看到, satp最高级页表起始处的物理地址是0x80205000,也就是在内核空间中, 在内存中, 但是刚开始我们不知道它的物理地址, 只知道虚拟地址, 可以用%hi(boot_page_table_sv39)获取。问题是知道物理地址呢, 可以线性映射, 内核起始处的物理地址是0x8020 0000,但是虚拟地址是0xffffffffc0200000, 中间差了0xffffffff40000000, 因此, 根页表的虚拟地址0xffffffffc0205000-0xffffffff40000000就是物理地址0x80205000。

这就是为什么要用li t1, 0xffffffffc0000000 - 0x80000000算偏移量, sub t0, t0, t1算出根页表的物理地址, 而我们知道物理地址到物理页号, 还需要去掉末尾12位, 所以srli t0, t0, 12才是根页表的物理页号。此时t0的高位是0. 之前说过stap最高位记录了模式

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。
- 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。

li t1, 8 << 60表示0100后面有60个0, 恰好对应sv39模式【39位虚拟内存】

or t0, t0, t1, 将t0的高4位置为0100, 此时t0高位记录模式, 低位记录根页表物理地址, 表示的就是satp

csrw satp, t0,已经可以将t0存入satp了

之前说过, 修改satp相当于修改了整个页表, 需要刷新快表: cswr satp, t0

综上所述, 这么一大段的效果就是: 算出satp寄存器的正确值。

entry.S:设置sp后准备跳转

而下一段:

```
# 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
```

就干了两件事：1设置sp，因为被sp标记的是栈空间。

2跳转到kern/init.c/kern_init()函数，只是算出这个地址要分成高低两次加载

entry.S其他设置

.align PGSHIFT页表需要对齐到一个页的起始，一个页有4K，12位

.zero 8 * 511 前511个页表项都是空的，一个页表项有8个bit,存储了对应的物理地址

映射是从0xffffffff_c0000000 map to 0x80000000，这就是为什么内核的物理内存存在0x80020000，而虚拟内存存在0xffffffff_c0200000

kern_init的代码

```
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 ...\n";
    //cprintf("%s\n", message);
    cputs(message);

    print_kerninfo();

    // grade_backtrace();
    idt_init(); // init interrupt descriptor table

    pmm_init(); // init physical memory management

    idt_init(); // init interrupt descriptor table

    clock_init(); // init clock interrupt
    intr_enable(); // enable irq interrupt
}
```

从外部获取到数据段结束位置和内核结束位置，将数据段结束位置到内核结束位置之间部分0xffffffffc0206010到0xffffffffc0206070，通常是bss段设置为0。

初始化啊输出控制台cons_init

输出一句话(THU.CST) os is loading ...

print_kerninfo()输出内核信息：

```
Special kernel symbols:
    entry  0xffffffffc0200036 (virtual)
    etext  0xffffffffc02017fa (virtual)
    edata  0xffffffffc0206010 (virtual)
    end    0xffffffffc0206070 (virtual)
Kernel executable memory footprint: 25KB
```

```
void print_kerninfo(void) {
    extern char etext[], edata[], end[], kern_init[];
    cprintf("Special kernel symbols:\n");
    cprintf("  entry  0x%016lx (virtual)\n", kern_init);
    cprintf("  etext  0x%016lx (virtual)\n", etext);
    cprintf("  edata  0x%016lx (virtual)\n", edata);
    cprintf("  end    0x%016lx (virtual)\n", end);
    cprintf("Kernel executable memory footprint: %dKB\n",
        (end - kern_init + 1023) / 1024);
}
```

初始化中断向量表,执行了trap.c里面的idt_init()【中断描述表初始化】，定时发生中断，而中断时pc指向stvc里面的处理程序，处理程序对应trapentry.S里面的all_trap,先保存寄存器，跳转处理程序，恢复寄存器。处理程序又写在trap.c里面，这里要复制lab1的处理方式，输出10个kick之后停止。

clock_init(); // 之后才会定时有时钟中断，注意把关闭程序也引入。

intr_enable(); // enable irq interrupt允许外部中断

pmm_init();这一句话才是本实验的重点，可以看到，他会执行pmm.c的以下操作：

```
memory management: best_fit_pmm_manager
physical memory map:
  memory: 0x0000000007e00000, [0x0000000008020000, 0x00000000087fffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x00000000080205000
```

首先来分析kern/mm/default_pmm.c,首先要理解单词的意思, kern:kernel

mm: **memory management**内存管理

pmm:**physical memory management**物理内存管理, 区别于虚拟内存管理vmm

pmm流程分析

```
/* 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[];
    satp_virtual = (pte_t*)boot_page_table_sv39;
    satp_physical = PADDR(satp_virtual);
    printf("satp virtual address: 0x%016lx\nsatp physical address: 0x%016lx\n", satp_virtual, satp_physical);
}
```

pmm.c/pmm_init()调用init_pmm_manager()

```
// init_pmm_manager - initialize a pmm_manager instance
static void init_pmm_manager(void) {
    pmm_manager = &best_fit_pmm_manager;
    printf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

选中要调用的管理方式 `struct pmm_manager *pmm_manager;`, 可以是default或者是bestfit,输出它的名字 `memory management: best_fit_pmm_manager`, 然后让它初始化调用page_init()

```
static void page_init(void) {
    va_pa_offset = PHYSICAL_MEMORY_OFFSET;

    uint64_t mem_begin = KERNEL_BEGIN_PADDR;
    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

    printf("physical memory map:\n");
    printf("  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;
    }
}
```

这一段确定了内核的物理内存的开始和结束位置, 内存的大小, 并且控制台里还输出了它

```
physical memory map:
memory: 0x0000000007e00000, [0x0000000008020000, 0x00000000087ffffff].
```

可以看到内核的物理内存是从0x8020000开始的【和过去一样】，从0x87ffff结束的

调用check_alloc_page();

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

调用内存管理器（default的或者是best的进行测试），如果中间没有报错，视为成功，输出成功。

```
check_alloc_page() succeeded!
```

最后还输出了根页表的物理地址和虚拟地址。

```
cprintf("satp virtual address: 0x%016lx\nsatp physical address: 0x%016lx\n", satp_virtual, satp_physical);
```

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？

对于流程的理解

在 default_pmm.h 里 面 可 以 看 出 ，

```
extern const struct pmm_manager default_pmm_manager;
```

默认内存管理器default_pmm_manager就是内存管理器pmm_manager,但是这里有具体的名字，和方法的名字。类似于c++的子类,属于用c语言实现类似c++子类的技巧。

```
//这个结构体在
const struct pmm_manager default_pmm_manager = {      Sakura, 4天前 • Lab2
    .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,
};
```

在

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

pmm_manager-

>check的时候，执行的实际上是

default_check,而default_check里面又表用来basic_check，总体上进行了一些测试。

对于空闲页块链表freelist的理解

然后要理解什么是free_area_t

```
free_area_t free_area;

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

static void      Sakura, 3天前 • Lab2
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

free_area_t是freelist【最好的名字叫“空闲页块链表”】nr_free和空闲页数的结合

页块是连续的若干页，一页一般有4k

其中freelist是一个链表(开头)，将页块链接在一起

nr_free记录的是整个freelist这么多个页块里面，空闲页的总数

```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
Sakura, 3天前 | 1 author (Sakura)
typedef struct {
    list_entry_t free_list;           // the list header
    unsigned int nr_free;             // number of free pages in this free list
} free_area_t;
```

对于default_init的理解

初始化操作，free_list链表头尾指向自己(调用的是list.h里面的初始化函数)，然后把nr_free计数为0

```
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}
```

此时表示的状态是：空闲页块一个没有，空闲页也一个没有。



对于页块page的理解

还要理解page是什么，page来自于memlayout.h【内存结构】

在一个页块头部的页，property=0~n，表示这个页块内的空闲页数,此时page可以代表一个页块
其他的页property=0,page理解为一个普通的页

```

typedef uintptr_t pte_t;
typedef uintptr_t pde_t;
Sakura, 3天前 • Lab2
/*
 * 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.
 */
Sakura, 3天前 | 1 author (Sakura)
struct Page {
    int ref;                // page frame's reference counter
    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
    list_entry_t page_link; // free list link
};

```

ref: 通常表示页被进程等引用的次数

flag表示页的状态，其中有两个重要状态

PG_reserved:

- 当这个标志被设置（即为1）时，意味着这个页是为内核保留的，不能用于常规的页面分配或释放。

PG_property:

相当于在问property是否不为0？

- 当这个标志被设置（即为1）时，property不为0，这个页在页块头部，被理解为页块。
- 否则有两种可能：1这个页在页块头部，但是该页块内部的页被分配完了。2该页不在页块头部。

page_link对应着freelist

对于default_init_memmap的理解

```

static void
default_init_memmap(struct Page *base, size_t n) {
    //从base这个页开始，将后面的n页(包含base)作为一个页块，设置为free状态，加入空闲页块链表
    assert(n > 0); //assert是一个宏，如果断言失败，终止执行。将0个以及以下的页作为一个页块显然不合理
    struct Page *p = base; //p是一个页指针，刚开始指向base
    for (; p != base + n; p++) { //遍历了页面[base base+1 ... base+n-1] 总共有n个页面
        assert(PageReserved(p)); //中间判断页面是否是保留的，初始化工作应该是对保留的进行，如果不是保留的，退出
        p->flags = p->property = 0; //所有页面开始的时候flags和property都是0
        set_page_ref(p, 0); //ref也是0
    }
    base->property = n; //只有第一个页面property是n，表示它是页块的第一个页
    SetPageProperty(base); //通过flags里面的property表示一个页是不是页块开头
    nr_free += n; //从base这个页开始，将后面的n页(包含base)作为一个页块，设置为free状态，因此free_list里面的空闲页增加了n页
}

```

```

if (list_empty(&free_list)) { //如果freelist是空的
    list_add(&free_list, &(base->page_link)); //将base代表的页块直接加入property
} else {
    list_entry_t* le = &free_list; //le list_entry链表节点指针，指向freelist链表的第一个节点
    while ((le = list_next(le)) != &free_list) { //当le的next没有指向结尾【由于freelist是一个循环链表，结尾是它自己】
        struct Page* page = le2page(le, page_link); //le2page是一个宏，从freelist链表节点，映射到对应页，这里相当于在遍历页
        if (base < page) { //如果base这个页地址小于page这个页的地址
            list_add_before(le, &(base->page_link)); //此时把base页里面的链表节点放在page对应的链表节点的前面
            break; //结束
        } else if (list_next(le) == &free_list) { //如果到了结尾，相当于下一个链表节点是开头
            list_add(le, &(base->page_link)); //此时把base插在最后面
        }
    }
}
}

```

注意，freelist链接的只是页块的第一个页的page_link

页表不为空，按照base页的地址插入

对于default_alloc_pages的理解

```
static struct Page *
default_alloc_pages(size_t n) { //需要n页的空闲空间，在空闲页块链表的所有页块里面，找一块至少有n页的，对它进行分配
    assert(n > 0); //n小于等于0没有意义
    if (n > nr_free) { //总的只nr_free页空闲页，需要n页的空闲页一定不行
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list; //le指针指向链表的开头
    while ((le = list_next(le)) != &free_list) { //当没有结束时
        struct Page *p = le2page(le, page_link); //从链表指针变成页指针，相当于在遍历页块
        if (p->property >= n) { //如果当前页块的空闲页数大于等于n
            page = p; //此时确定用它。找到第一个就用，因此叫做first-fit
            break;
        }
    }
}
```

```
}
if (page != NULL) { //如果找到了至少有n页空闲页的一个页块page
    list_entry_t* prev = list_prev(&(page->page_link)); //暂时存储前面的页块，后面有用
    list_del(&(page->page_link)); //暂时先把这个页块从空闲页块链表当中拿走
    if (page->property > n) { //如果这个页块不是刚好够【有n页空闲页，那样就不用放回来了】，那么还需要把剩下的放回来
        struct Page *p = page + n; //从page-page+n-1这些页都要拿走
        p->property = page->property - n; //第page+n页重新称为页块头部，后面跟着原来页块数-n个页块
        SetPageProperty(p); //设置这个页称为页块头部
        list_add(prev, &(p->page_link)); //将这个新的页块插入链表
    }
    nr_free -= n; //由于分走了n个空闲页，所以空闲页总数-n
    ClearPageProperty(page); //page不再是页块的头部
}
return page; //返回找到的页块头部
```

对于default_free_pages的理解：

```
static void
default_free_pages(struct Page *base, size_t n) { //从base这一页开始，将[base,base+1,...base+n-1]
    //这一段和init绝大部分是一样的，就不分别写了，区别在于下面
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(!PageReserved(p) && !PageProperty(p)); //init针对的所有页面都是被保留的，但是free针对的页面必须不是保留的，并且不是页块的第一块
        p->flags = 0; //这里没有property=0操作 You, 现在 * Uncommitted changes
        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));
            }
        }
    }
}
```

向前合并

```
list_entry_t* le = list_prev(&(base->page_link)); //链表前一个
if (le != &free_list) { //如果添加链表元素之前，链表是空的，添加之后，新元素的前一个元素就是free_list，如果前一个元素le不是free_list，说明之前链表不为空
    p = le2page(le, page_link); //找到le对应的页p
    if (p + p->property == base) { //如果页p开头的页块紧邻着base开头的页块，就可以合并，把base丢掉
        p->property += base->property; //p代表的页块里面的空闲页数增加了base页块的空闲页数 那么多
        ClearPageProperty(base); //base不再是空闲页块的开头
        list_del(&(base->page_link)); //base对应的页块从链表中移除
        base = p; //base暂时移动到p，因为可能还要向后合并
    }
}
```

向后合并

```

le = list_next(&(base->page_link)); //base下一个节点
if (le != &free_list) { //不是哨兵节点
    p = le2page(le, page_link); //从链表节点找页块
    if (base + base->property == p) { //base页块恰好和这个页块相邻
        base->property += p->property; //base对应页块的容量+=后面页块的容量
        ClearPageProperty(p); //后面不再是页块
        list_del(&(p->page_link)); //移除后面的页块
    }
}
}

```

优化方案

伙伴系统（Buddy System）：它将内存分成大小为2的幂的块，并尝试满足内存请求的最接近的块大小。好处是减少内存碎片。

延迟合并（Deferred Coalescing）：而不是立即合并空闲块，可以延迟合并直到有需求或系统处于低负载状态。

内存碎片：在default_alloc_pages里面，当我们需要n页的时候，必须要找到一个至少有n页的页块，不一定能满足，但是可能有很多碎片加起来是够的。因此我们可以考虑利用这些内存碎片。

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？

default_init

这一段best fit和default fit没有差别

free_area表示一大块内存空间，初始化时没有空的内存，freelist里面是空的,可用的内存大小为0

```

free_area_t free_area;

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

static void
best_fit_init(void) {
    list_init(&free_list);
    nr_free = 0;
}

```

default_init_memmap

代码填空1

这里和default完全一样

内存页框的标志位置为0，表示free

```
best_fit_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);
        /*LAB2 EXERCISE 2: YOUR CODE*/
        // 清空当前页框的标志和属性信息，并将页框的引用计数设置为0
    }
    base->property = n;
}
```

代码填空2

default也是一样的，

```
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);
        /*LAB2 EXERCISE 2: YOUR CODE*/
        // 编写代码
        // 1、当base < page时，找到第一个大于base的页，将base插入到它前面，并退出循环
        if (base < page) {
            list_add_before(le, &(base->page_link));
            break;
        }
        // 2、当list_next(le) == &free_list时，若已经到达链表结尾，将base插入到链表尾部
        else if (list_next(le) == &free_list) {
            list_add(le, &(base->page_link));
        }
    }
}
```

效果也是相同的：

假设我们有以下 free_list（假设使用基地址表示）：

```
free_list: HEAD -> 100 -> 300 -> 500 -> TAIL
```

如果我们要插入一个基地址为 250 的新页面块，则新页面块应插入 100 和 300 之间：

```
free_list: HEAD -> 100 -> 250 -> 300 -> 500 -> TAIL
```

此时，当我们遍历到基址为 100 的页面块时，下一个页面块是 300， $250(\text{base}) < 300(\text{page})$ ，所以我们使用 `list_add_before` 将新页面块插入 300(page) 之前。

现在，考虑我们要插入一个基址为 600 的新页面块，这个页面块应该插入 500 后面，即链表的末尾：

```
free_list: HEAD -> 100 -> 250 -> 300 -> 500 -> 600 -> TAIL
```

当我们遍历到 500 时，它的下一个链表入口是 `free_list` 的头部，所以我们知道已经到达链表的末尾。这时，我们使用 `list_add` 将新页面块添加到 500 之后。

也就是说best-fit的效果需要后面的函数才能体现

代码填空3

default 的 逻辑 是 :

```
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;
    }
}
```

在free list里面逐个遍历页块，找到第一个大小超过或等于了n（我们需要的页块大小）的页块返回

```
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;
    }
}
```

修改之后的逻辑是，找到第一个大小足够的页块时，先不要返回，因为后面可能会有更合适的，把页块大小存储下来，如果后面页块大小比这个还小，就选取他，最后就可以挑选到大小 $\geq n$ 的页块中最小的页块，也就是最合适的。

如果所有页块都小于n，返回空，并且min_size初始值是页块的大小极限，足够大，不会有逻辑问题。

代码填空4

这一段和default一样

```
/*LAB2 EXERCISE 2: YOUR CODE*/
// 编写代码
// 具体来说就是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
base->property = n;
SetPageProperty(base);
nr_free += n;
```

代码填空5

这一段也和default一样

```
list_entry_t* le = list_prev(&(base->page_link));
if (le != &free_list) {
    p = le2page(le, page_link);
    /*LAB2 EXERCISE 2: YOUR CODE*/
    // 编写代码
    if (p + p->property == base) { // 1、判断前面的空闲页块是否与当前页块
        p->property += base->property; // 2、首先更新前一个空闲页块的大小，
        ClearPageProperty(base); // 3、清除当前页块的属性标记，表示不再是空
        list_del(&(base->page_link)); // 4、从链表中删除当前页块
        base = p; // 5、将指针指向前一个空闲页块，以便继续检查合并后的连续空
    }
}
```

通过验证

```
memory management: best_fit_pmm_manager
physical memory map:
  memory: 0x0000000007e00000, [0x0000000008020000, 0x00000000087fffffff].
check_alloc_page() succeeded!
```

优化方案

使用更好的数据结构: 使用如平衡树、跳表等可以在 $O(\log n)$ 时间内查找、插入和删除的数据结构，可以提高某些操作的效率。

伙伴系统 (Buddy System) : 这种方法在请求分配和释放内存时使用特定的算法来避免内存碎片。它将内存分成大小为2的幂的块，并尝试满足内存请求的最接近的块大小。

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的同学可单独提交Challenge。完成得好的同学可获得最终考试成绩的加分。

知识点总结和对比：

1. 物理内存管理：

- 如果没有适当的物理内存管理，所有程序的地址空间都会冲突。
- 为了避免每个字节都需要翻译，引入“页”的概念，进行地址翻译。

2. RISC-V架构下的sv39页表机制：

- 每页大小为4KB，页表存储虚拟地址和物理地址之间的映射。
- 虚拟页数量可能远大于物理页。
- 虚拟地址有64位，但只有低39位是有效的。
- 地址的最后12位代表页内偏移。

3. 操作系统启动：

- bootloader在启动时不直接调用kern_init。
- 首先调用kern_entry函数，为kern_init建立环境，设置堆栈并建立段映射关系。
- kern_init函数进行物理内存管理初始化，然后进行中断和异常的初始化。

4. 页表项的构成：

- 描述虚拟页如何映射到物理页。
- sv39中，每个页表项大小为8字节，其中包含保留位、物理页号和映射状态信息等。
- 页表项还有一些特定的标志位，如Dirty、Accessed、Global等。

5. 多级页表：

- 普通页表是一个数组，使用虚拟页号作为索引来查找物理页号。
- 多级页表可以节省空间，特别是当虚拟地址空间非常大时。
- 二级页表可以非连续，只需要建立必要的页表项。

6. satp寄存器：

- 存储根页表的起始地址。
- 高4位存储页表模式。

7. TLB快表：

- TLB是页表的缓存。
- 修改satp或页表项会导致TLB不同步，需要刷新TLB。