操作系统实验报告 Lab2

成员：陈豪斌　朱浩泽　许佳培

在这之前我们必须详细地了解 uCore 启动分页机制的原理是什么。最开始在 GCC 对操作系统编译的时候，会根据 GNU 链接脚本调用 kern entry 这一段汇编代码，并且这段代码将系统载入的地址设置在 0xC01000000（即 KERNBASE）处，因此设置 REALLOC 函数就很有必要了，不然一段地址可能会访问到错误的地方。其中，kern entry 设置入口点的代码在 kern/init/entry.S 中，它主要是将一级页表的基地址，即 boot_pgdir 载入寄存器 cr3 中（Control Register 3），随后将 cr0 的 PG 标志位设置上，标志分页机制已经开启。

在做好这些工作之后，kern_init 函数也就可以正常地执行了。

我们主要关注物理内存管理器的初始化函数 pmm_init 是如何填充页表、对 pmm 进行初始化的。

1. boot_cr3 = PADDR(boot_pgdir)：获取一级页表的基地址，注意 entry.S 和 kernel.ld 完成了将其读入的过程，因此我们可以轻松获取基地址。注意到在 entry.S 中，movl %eax, %cr0 指令结束后，在分段模式下的 uCore 实际上已经允许开启了分页机制。随后，因为此时的线性地址已经不是物理地址了，如果内核直接把运行的线性地址当作物理地址的话，就会发生问题，因为分页机制开启了，但没有正确映射会导致访问出错。为了能够实现内核运作不出错，要重新对自身的虚拟地址进行映射，即将低 0-4M 的虚拟内存区域映射到 KERNBASE + 0-4M 的物理地址上去，这样就支持了内核自身的运作。虽然内核已经把自己映射到了正确的物理地址，但此时的内核（EIP）实际上还占用着 0 4M 的低虚拟地址区域，这部分区域不是内核该存在的地方，因此它就需要进行一次长跳转，让自己处于虚拟地址的高处，留出空间为用户程序使用。

   简而言之，这里的 boot_pgdir[0]，纯粹只是为了能够让内核在虚拟地址下运行而做的映射而已，当内核真正地要开始进行页表映射初始化后，就要把这段原始的映射关系取消。总结来看，这一阶段的目的就是更新映射关系的同时将运行中的内核（EIP）从低虚拟地址“迁移”到高虚拟地址，而不造成伤害。

2. pmm_init()：初始化 pmm_manager，我们实现了 First Fit 算法，因此让其指向 default 版本，如果想要实现别的，可以让它指向一个我们实现的 manager；

3. page_init()：物理内存探测，检测哪些内存区域可用。这个函数比较重要，需要着重进行分析：

   (a) 首先这个函数会获取一个叫做 e820map 的结构体。

   e820 is shorthand for the facility by which the BIOS of x86-based computer systems reports the memory map to the operating system or boot loader.

   这个 E820 的获取，需要联系 BootLoader 的加载过程。关注 bootasm.S 文件中有关 probe memory 小节的描述，最重要的部分是它向 CPU 发送了一个 int 15H 的中断，以及设置 AX 低位为 E820（movl $0xE820, %eax），那么这个中断会让 CPU 返回一个关于内存区域的报告信息，即 E820 结构。它被存到了离内核基址 8000 的内存处，所以根据 KERNBASE + 0x8000 就可以获取到内存分布的 E820 结构了。

一、 实现 First Fit 算法

首先我们需要分析代码中是如何组织空闲内存块的。实际上在代码中并非单独创建某个结构体用以表示 block，而是定义了一个叫做 Page 的结构，然后以每个 block 的基址 Page 来标识各个块，作为它们的 Identifier。这样的话就可以支持地址的比较了，因为，我们只需要定义一个指向 Page 的指针即可。指针的值就是它指向的元素的地址，因此，地址的比较可以在指针的运算的基础上完成。详情可见下图：

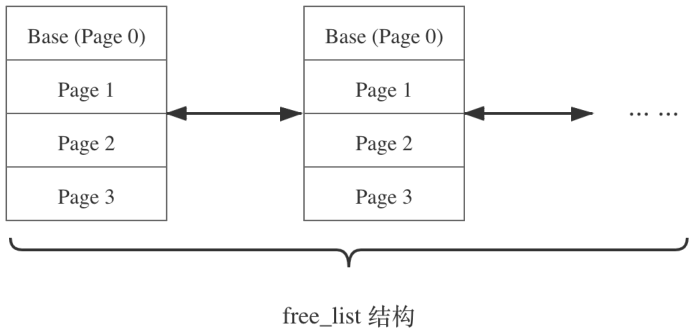

图 1: free_list 和空闲内存块

例如，假设我们想要访问 block 1 中的 page 2，我们先要设置一个指针指向 free_list 的开始位置，随后将每个 list_entry 类型的指针，通过一个叫做 le2page 的函数转换为 page 后判断 property，然后设置基址指针为它，在基址的基础上增加偏移量即可访问任意 page。随后的 free 和 alloc 函数也是基于指针和指针与 page 之间的转换而实现的。如果想要将 page 作为遍历 free_list 的指针使用，就可以直接利用它的成员变量 list_link 即可。

对各个函数的模块功能解释：

1. 有关 defualt_init 函数：它初始化 free_list。这个过程很简单，它把头尾指针都指向自身即可；
2. 有关 defualt_init_memmap 函数：在一个块被分配之后我们需要把它进行初始化。因此它遍历块中的各个 page，将其状态都设置为空闲，把 reference 标志位清空等。特别地，如果 page 恰好是这个块的初始页，那么我们需要将其的 property 设置成块的大小。因为我们想要实现 FF 算法，所以，必须要根据地址进行排序，那么通过指针比较即可实现这个地址排序功能。此处就须将每个初始化好的块放置到 free_list 的起始部分。
3. 有关 default_alloc_pages 函数：显然我们需要遍历 free_list 这个链表来查看是否一个块满足我们所需的要求，即某个 page 的 property（如果不为 0，它代表了块的页数量）大于等于 n（所需大小）。在分配完这一个内存块后，我们需要把剩余的空间（如果有的话）作为一个新的内存块链入当前分配完的内存页的后面。这一步可以很轻松地使用指针完成。假设当前被分配的块的标识符是 base，所需的大小为 n，那么我们只需要将 base + n 之后的指针作为一个新块链入到 base + n 之后即可，与此同时我们需要将 base 从空闲链表上摘下，然后设置 base + n 之后的内存块的大小为原始块大小减去 n，再将其链入 base + n 指针之后（通过 list_add_after 实现）。
4. 有关 default_free_pages 函数：这个函数是将一个用完的内存块链入 free_list 并对碎片进行整理的函数（原函数并没有完全实现）。这个过程实际上也很简单。第一步，我们需要清空要释放的内存块的所有 page；第二步，我们需要遍历 free_list 链表，找到能够合并的块：有两种可能，一是某个块可以和前面的块合并，二是一个块能和它后面的块合并，所以会有一个分支检测。怎么判断两个块恰好能合并呢？显然可以通过 base 的地址加上块大小是否和前后的块的起始地址相等进行判断。第三步，将空闲的块链入 free_list 中，这一步需要保持地址有序，所以依旧需要遍历链表，找到第一个地址恰好比要插入的块小的那个块，然后调用 add_before 函数链入。P.S. 如果先链入再合并会导致这个空闲块没法和前后块合并了。

那么是否还有优化的余地呢？我们分析可见，实际上最花费时间的是遍历链表的过程，但是根据地址排序的链表想要找到第一个大小满足所需的 block 是无法优化的，只有一处可以优化：插入。因为插入是根据地址进行的，并非通过块大小，这意味着如果我们能够优化插入过程，就有希望优

化部分时间复杂度。自然可以联想到使用二叉搜索树之类的树结构实现。不过可能需要保存一份 free_list 的拷贝，并且单纯使用二叉搜索树可能会导致树深度过大，因此使用红黑树或者 AVL 树可能会是更好的解决方案。

二、 **实现寻找虚拟地址对应的页表项**。在实现之前，我们首先需要分析现代操作系统中是如何实现分页机制的。第一，对于每个进程而言，它都有自己的页表，这样能够实现每个应用程序都以为自己拥有全部的内存空间，而应用程序发起的虚拟地址请求将会被内存管理单元 (Memory Management Unit) 截获。第二，通过查询页表，将对应的物理地址传送回来，MMU 在此过程中相当于是一个翻译器。那么，为了查询页表，我们首先需要知道页表的地址在哪里，这就需要一个叫做 Page Directory Table 的表，它能够返回每个进程发起的虚拟地址请求的对应页表地址。原理图如下图所示：

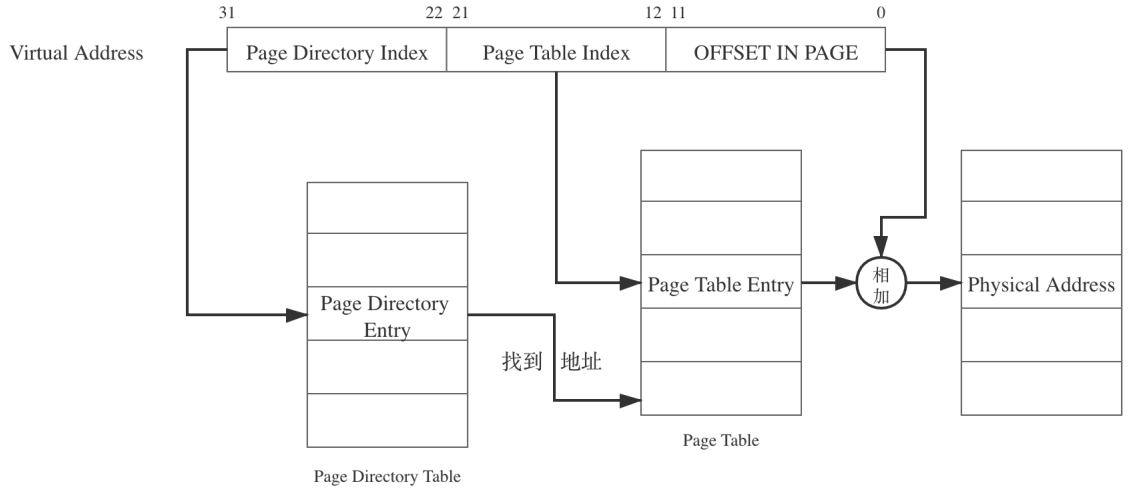

图 2: 页表、页表的表和虚拟地址转换机制

另，若页表的表访问发生缺失，说明这个进程刚刚创建，所以我们需要为其开辟一片新的页（根据 create 变量来判断）。随后，我们需要将访问的页设置为 referenced。故函数实现方式如下：

1. 根据虚拟地址的高位和页表的表的基址，找到对应于这个虚拟地址的页表的地址。如果不存在，就需要分配一片新的内存页给页表的表，并对内容初始化为 0，状态位重设；
2. 接下来通过 PTE_ADDR 函数，获取这个 Page Directory Entry（即页表的表中的某一项）指向的页表地址；
3. 根据第二步找到的页表地址，加上虚拟地址中间一部分对应的页表内的偏移，形成最终的页表项返回。

下图是 PDE 和 PTE 的结构：

Table 5.12. Directory Descriptor Entry

Descriptor Bit	Bit Name	Description
bit 0	Present	1 if the page descriptor is present and valid
bit 2	R/W	Read/write; if 0, writes may not be allowed to the 4-MB region controlled by this entry
bit 2	U/S	User/supervisor; if 0, accesses with CPL = 3 are not allowed to the 4-MB region controlled by this entry
bit 3	PWT	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry
bit 4	PCD	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry
bit 5	A	Accessed; indicates whether this entry has been used for linear-address translation
bit 6	D	Ignored
bit 7	PS	If CR4.PSE = 1, must be 0
bits 8–11	Ignored	Should be zero
bits 12–31	Addr	Physical address of 4-kB aligned page table referenced by this entry

图 3: Page Directory Entry

Table 5.13. Page Table Entry (That Maps a 4-kB Range)

Page Descriptor Bit	Bit Name	Description
bit 0	Present	1 if the page descriptor is present and valid
bit 2	R/W	Read/write; if 0, writes may not be allowed to the 4-kB region controlled by this entry
bit 2	U/S	User/supervisor; if 0, accesses with CPL = 3 are not allowed to the 4-kB region controlled by this entry
bit 3	PWT	Page-level write-through; indirectly determines the memory type used to access the 4-kB page referenced by this entry
bit 4	PCD	Page-level cache disable; indirectly determines the memory type used to access the 4-kB page referenced by this entry
bit 5	A	Accessed; indicates whether software has accessed the 4-kB page referenced by this entry
bit 6	D	Dirty; indicates whether software has written to the 4-kB page referenced by this entry
bit 7	PAT	If the PAT is supported, indirectly determines the memory type used to access the 4-kB page referenced by this entry
bit 8	Global	Global; if CR4.PGE = 1, determines whether the translation is global
bit 9-11	Ignored	Should be zero
bit 12-31	Address	Physical address of 4-kB aligned page referenced by this entry

图 4: Page Table Entry