操作系统实验报告 Lab2

成员：陈豪斌　朱浩泽　许佳培

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：

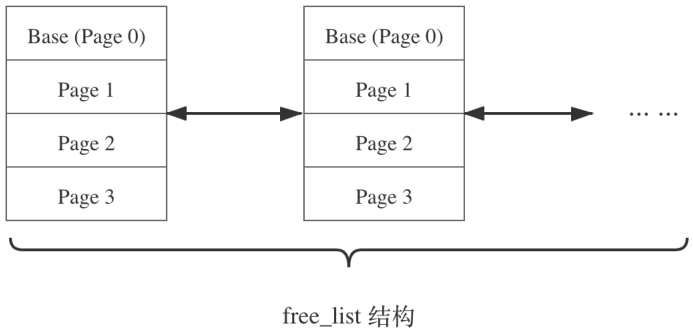

图 1: free_list 和空闲内存块

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：

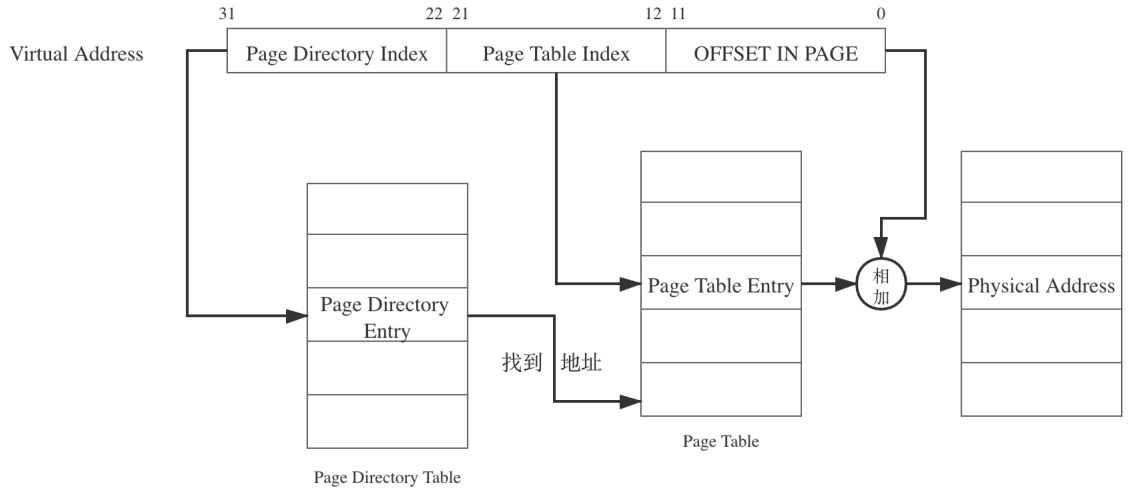

图 2: 页表、页表的表和虚拟地址转换机制

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. 根据第二部找到的页表地址，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。