lab2

樊子晨

练习1：实现 first-fit 连续物理内存分配算法（需要编程）

在实现first fit 内存分配算法的回收函数时，要考虑地址连续的空闲块之间的合并操作。提示:在建立空闲页块链表时，需要按照空闲页块起始地址来排序，形成一个有序的链表。可能会修改default_pmm.c中的default_init，default_init_memmap，default_alloc_pages， default_free_pages等相关函数。请仔细查看和理解default_pmm.c中的注释。

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

首先我们查看和理解default_pmm.c中的注释

```
/*  In the First Fit algorithm, the allocator keeps a list of free blocks
 * (known as the free list). Once receiving a allocation request for memory,
 * it scans along the list for the first block that is large enough to satis
 * the request. If the chosen block is significantly larger than requested,
 * is usually splitted, and the remainder will be added into the list as
 * another free block.
 *  Please refer to Page 196~198, Section 8.2 of Yan Wei Min's Chinese book
 * "Data Structure -- C programming language".
*/
// LAB2 EXERCISE 1: YOUR CODE
// you should rewrite functions: `default_init`, `default_init_memmap`,
// `default_alloc_pages`, `default_free_pages`.
/*
 * Details of FFMA
 * (1) Preparation:
 *  In order to implement the First-Fit Memory Allocation (FFMA), we should
 * manage the free memory blocks using a list. The struct `free_area_t` is u
 * for the management of free memory blocks.
 *  First, you should get familiar with the struct `list` in list.h. Struct
 * `list` is a simple doubly linked list implementation. You should know how
 * USE `list_init`, `list_add`(`list_add_after`), `list_add_before`, `list_d
 * `list_next`, `list_prev`.
 *  There's a tricky method that is to transform a general `list` struct to
 * special struct (such as struct `page`), using the following MACROs: `le2p
 * (in memlayout.h), (and in future labs: `le2vma` (in vmm.h), `le2proc` (in
 * proc.h), etc).
 * (2) `default_init`:
 *  You can reuse the demo `default_init` function to initialize the `free_l
 * and set `nr_free` to 0. `free_list` is used to record the free memory blo
 * `nr_free` is the total number of the free memory blocks.
```

```
 * (3) `default_init_memmap`:
 *  CALL GRAPH: `kern_init` --> `pmm_init` --> `page_init` --> `init_memmap`
 * `pmm_manager` --> `init_memmap`.
 *  This function is used to initialize a free block (with parameter `addr_b
 * `page_number`). In order to initialize a free block, firstly, you should
 * initialize each page (defined in memlayout.h) in this free block. This
 * procedure includes:
 *  - Setting the bit `PG_property` of `p->flags`, which means this page is
 * valid. P.S. In function `pmm_init` (in pmm.c), the bit `PG_reserved` of
 * `p->flags` is already set.
 *  - If this page is free and is not the first page of a free block,
 * `p->property` should be set to 0.
 *  - If this page is free and is the first page of a free block, `p->proper
 * should be set to be the total number of pages in the block.
 *  - `p->ref` should be 0, because now `p` is free and has no reference.
 *  After that, We can use `p->page_link` to link this page into `free_list`
 * (e.g.: `list_add_before(&free_list, &(p->page_link));` )
 *  Finally, we should update the sum of the free memory blocks: `nr_free +=
 * (4) `default_alloc_pages`:
 *  Search for the first free block (block size >= n) in the free list and r
 * the block found, returning the address of this block as the address requ
 * `malloc`.
 *  (4.1)
 *      So you should search the free list like this:
 *          list_entry_t le = &free_list;
 *          while((le=list_next(le)) != &free_list) {
 *          ...
 *      (4.1.1)
 *          In the while loop, get the struct `page` and check if `p->proper
 *      (recording the num of free pages in this block) >= n.
 *              struct Page *p = le2page(le, page_link);
 *              if(p->property >= n){ ...
 *      (4.1.2)
 *          If we find this `p`, it means we've found a free block with its
 *      >= n, whose first `n` pages can be malloced. Some flag bits of this
 *      should be set as the following: `PG_reserved = 1`, `PG_property = 0`
 *      Then, unlink the pages from `free_list`.
 *          (4.1.2.1)
 *              If `p->property > n`, we should re-calculate number of the r
 *          pages of this free block. (e.g.: `le2page(le,page_link))->proper
 *          = p->property - n;`)
 *          (4.1.3)
 *              Re-caluclate `nr_free` (number of the the rest of all free b
 *          (4.1.4)
 *              return `p`.
 *      (4.2)
 *          If we can not find a free block with its size >=n, then return N
 * (5) `default_free_pages`:
 *  re-link the pages into the free list, and may merge small free blocks in
 * the big ones.
 *  (5.1)
```

```
 *      According to the base address of the withdrawed blocks, search the f
 *  list for its correct position (with address from low to high), and inser
 *  the pages. (May use `list_next`, `le2page`, `list_add_before`)
 *  (5.2)
 *      Reset the fields of the pages, such as `p->ref` and `p->flags` (Page
 *  (5.3)
 *      Try to merge blocks at lower or higher addresses. Notice: This shoul
 *  change some pages' `p->property` correctly.
 */

```

函数修改

default_init_memmap函数：对最初的一整块未被占用的物理内存空间中的每一页所对应的page结构（状态）进行初始化。相邻物理页对应的page结构在内存上也是相邻的，因此可以直接通过第一个空闲物理页对应的page结构加上一个偏移量的方式，来访问所有的空闲的物理页的page结构。具体初始化方式：

遍历所有空闲物理页的page结构，将page结构中描述空闲块数目的变量置零（故该成员变量只有在整个空闲块的第一个page中才有意义），然后清空这些物理页的引用计数，再通过设置flags的位的方式将其标记为空闲

```
struct page *p = base;
   for(;p!=base+n;p++){
       assert(pagereserved(p));
       p->flags=p->property=0;
       set_page_ref(p,0)
       setpageproperty(p);//新添加
   }
```

再对空闲块的第一个页的page结构进行初始化，具体实现为，将其表示空闲块大小的成员变量设置为参数环路的空闲块大小（单位为页），然后更新存储所有空闲页数量的全局变量，再将这个空闲块插入到空闲内存块链表中（只需将第一个page的page_link插入即可）

```
base->property=n;
   nr-free+=n;
   list_add(&free_list,&(base->page_link));
```

default_alloc_pages：分配指定页数的连续空闲物理空间，并将第一页的page结构的指针作为结果返回。具体实现方式：

对参数进行合法性检查，并查询总的空闲物理页数目是否足够进行分配。如果不足够进行分配，直接返回NULL，分配失败。

从头开始遍历保存空闲物理内存块的链表（按照物理地址从小到大的顺序），如果找到一个连续内存块的大小满足当前需要的内存块，则说明可以成功匹配。优先选择第一个遇到的满足条件的空闲内存块来完成内存分配。代码实现如下：

```
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;
        }
    }
}
```

如果内存块的大小大于需要的内存大小，则将空闲内存块分裂为两块，一块大小为所需内存大小，另一块则重新进行初始化。重新初始化包括对第一个page中表示空闲块代销的成员变量进行设置，其应当设置为剩下的空闲块大小，并将这个分裂出来的空闲块插入到空闲块链表中（链表中的空闲块按照物理地址从小到大排序）。如果内存块大小与所需大小相同，则不用分裂，对分配出去的物理内存的每一个描述信息（对应的page结构）进行初始化，修改flags成员变量，设置成非空闲，再将原始空闲块在空闲链表中删除，更新表示总空闲页数量的全局变量；最后返回用于分配到的物理内存的page结构指针。代码如下：

```
if(page!=NULL)
{//如果寻找到了满足条件的空闲内存块
for (struct page *p=page;p!=(page+n);++p)
{
    clearpageproperty(p);//将分配出去的内存页标记为非空闲
}
if (page->property>n)
{//如果原先找到的空闲块大小大于需要的分配内存大小，进行分裂
    struct page *p=page+n;//获得分裂出来的新的小空闲块的第一个页的描述信息
    p->property=page->property-n;//更新新的空闲块的大小信息
    list_add(&(page->page_link),&(p->page_link));//将新的空闲块插入空闲块列表中
}
list_del(&(page->page_link));//删除空闲链表中的原先的空闲块
nr_free -=n;//更新总空闲物理页的数量
}
return page;
```

default_free_pages：释放指定的某一物理页开始的连续物理页，并且完成first-fit算法中需要的一些信息维护。具体实现如下：
考虑遍历需要释放的物理页的page结构，对其进行更新。更新方式：
1.判断原先这些物理页是否被占用，如果释放未被占用的物理页，说明出现异常
2.设置flags将这些物理页标记为空闲
3.清空这些物理页的引用计数

```
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));//进行检查
        //p->flags=0;
        setpageproperty(p);//标记为空闲
        set_page_ref(p,0);//清空引用计数
    }
```

```
}
```

4.将这一新的空闲块插入到空闲块链表中

```
base->property=n;//设置空闲块大小
list_entry_t *le=list_next(&free_list);
for(; le !=(&free_list)&&le<(&(base->page_link));le=list_next(le));
//寻找新的空闲块在空闲块链表中应当处于的位置
list_add_before(le,&(base->page_link));//将空闲块插入到链表中
nr_free+=n;//更新空闲物理页总量
```

5.对空闲块跟其相邻的空闲块合并。通过merge_backward函数实现，将指定的某一个空闲块与其链表后的空闲块进行合并，如果合并失败返回0，否则返回1.

```
while(merge_backward(base));
//将新插入的空闲块和其物理地址大的一端的所有相邻的物理空间块进行合并
for(list_entry_t*i=list_prev(&(base->page_link));i!=&free_list;i=list_prev(
{//将新插入的空间块和其物理地址小的一端的所有相邻的物理空间块进行合并
if(!merge_backward(le2page(i,page_link)))
break;
}
```

回答问题：你的first-fit算法是否有进一步的改进空间?
答：
有。最主要的不足在时间效率。每次要查询一块适合条件的空闲内存块时，需要对链表进行遍历。最坏的情况需要找遍整个链表，时间复杂度为O(n)，n为当前链表大小。
改进方法：采取树的结构来取代简单的链表结构，按照中序遍历把得到的空闲块序列的物理地址按照从小到大的顺序排列。时间复杂度可以从O(n)提升到O(log n)

练习2：实现寻找虚拟地址对应的页表项（需要编程）

通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。其中的get_pte函数是设置页表项环节中的一个重要步骤。此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表。本练习需要补全get_pte函数 in kern/mm/pmm.c，实现其功能。请仔细查看和理解get_pte函数中的注释。get_pte函数的调用关系图如下所示：
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：

- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处。
- 如果ucore执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

Ucore的页式管理通过一个二级的页表实现。一级页表存放在高10位中，二级页表存放于中间10位中，最后的12位表示偏移量，据此可以证明，页大小为4KB（2的12次方，4096）。

Directory为一级页表
PDX(la)可以获取Directory(获取一级页表)
Table为二级页表
PTX(la)可以获取Table(获取二级页表)
get_pte的注释中给出了一些宏和定义：

PDX(la)： 返回虚拟地址la的页目录索引
KADDR(pa): 返回物理地址pa对应的虚拟地址
set_page_ref(page,1): 设置此页被引用一次
page2pa(page): 得到page管理的那一页的物理地址
struct Page * alloc_page() : 分配一页出来
memset(void * s, char c, size_t n) : 设置s指向地址的前面n个字节为‘c’
PTE_P 0x001 表示物理内存页存在
PTE_W 0x002 表示物理内存页内容可写
PTE_U 0x004 表示可以读取对应地址的物理内存页内容

```
//此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个
// pgdir:PDT的内核虚拟地址 la：需要映射的线性地址 creat：决定是否为PT分配页面的逻辑
//return vaule:这个pte的内核虚拟地址
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
{
 //找一级页表PDE
 pde_t *pdep = pgdir + PDX(la);
 //如果存在的话，返回对应的PTE即可
 if (*pdep & PTE_P) {
 pte_t *ptep = (pte_t *)KADDR(*pdep & ~0x0fff) + PTX(la);
 return ptep;
 }
 //如果不存在的话分配给它page
 struct Page *page;
 if (!create || ((page = alloc_page()) == NULL))
 {
 return NULL;
 }
 //要查找该页表，则引用次数+1
 set_page_ref(page, 1);
 //得到page的线性地址
 uintptr_t pa = page2pa(page) & ~0x0fff;
 //转成虚拟地址并初始化为0，因为还没有开始映射
 memset((void *)KADDR(pa), 0, PGSIZE);
  //设置控制位，同时设置PTE_U,PTE_W和PTE_P，分别代表用户态的软件可以读取对应地址的物
 *pdep = pa | PTE_P | PTE_W | PTE_U;
 //返回PTE
 pte_t *ptep = (pte_t *)KADDR(pa) + PTX(la);
 //返回对应页项地址
 return ptep;
}
```

需要注意的地方：
通过 default_alloc_pages() 分配的页的地址并不是真正的页分配的地址，实际上只是 Page 这个结构体所在的地址，故而需要通过使用 page2pa() 将 Page 这个结构体的地址转换成真正的物理页地址的线性地址，需要注意的是：无论是 * 或是 memset 都是对虚拟地址进行操作的所以需要将真正的物理页地址再转换成内核虚拟地址。
总而言之，页目录项是一级页表，存储了各二级页表的起始地址，页表是二级页表，存储了各个页的起始地址。一个虚拟地址（线性地址）通过页机制翻译得到最终的物理地址。

页表的主要作用是：假如在系统里面，物理内存和虚拟内存是一一对应的，那么在进程空间里面就会存在很多的页表，同时也会占据很多的空间，那么，为了解决这个问题就出现了多级页表。

如果ucore执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?
1. 硬件陷入内核，在堆栈中保存程序计数器。大多数机器将当前指令的各种状态信息保存在特殊的CPU寄存器中。
2. 启动一个汇编代码例程保存通用寄存器和其他易失的信息，以免被操作系统破坏。这个例程将操作系统作为一个函数来调用。
3. 当操作系统发现一个缺页中断时，尝试发现需要哪个虚拟页面。通常一个硬件寄存器包含了这一信息，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析这条指令，看看它在缺页中断时正在做什么。
4.一旦知道了发生缺页中断的虚拟地址，操作系统检查这个地址是否有效，并检查存取与保护是否一致。如果不一致，向进程发出一个信号或杀掉该进程。如果地址有效且没有保护错误发生，系统则检查是否有空闲页框。如果没有空闲页框，执行页面置换算法寻找一个页面来淘汰。
如果选择的页框“脏”了，安排该页写回磁盘，并发生一次上下文切换，挂起产生缺页中断的进程，让其他进程运行直至磁盘传输结束。无论如何，该页框被标记为忙，以免因为其他原因而被其他进程占用。
5. 一旦页框“干净”后（无论是立刻还是在写回磁盘后），操作系统查找所需页面在磁盘上的地址，通过磁盘操作将其装入。该页面被装入后，产生缺页中断的进程仍然被挂起，并且如果有其他可运行的用户进程，则选择另一个用户进程运行。
6.当磁盘中断发生时，表明该页已经被装入，页表已经更新可以反映它的位置，页框也被标记为正常状态。
7.恢复发生缺页中断指令以前的状态，程序计数器重新指向这条指令。
8.调度引发缺页中断的进程，操作系统返回调用它的汇编语言例程。
9.该例程恢复寄存器和其他状态信息

练习3：释放某虚地址所在的页并取消对应二级页表项的映射（需要编程）

当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构Page做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。请仔细查看和理解page_remove_pte函数中的注释。为此，需要补全在 kern/mm/pmm.c中的page_remove_pte函数。page_remove_pte函数的调用关系图如下所示：
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：

- 数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥?
- 如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事?

思路：检查页表项是否存在，如果存在，将其对应的映
射关系取消，并将PTE清除，之后刷新tlb即可。

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep)
{
 //检查page directory是否存在
 //练习二我们已经学到 PTE_P用于表示page dir是否存在
 if (*ptep & PTE_P)
 {
```

```
 struct Page *page = pte2page(*ptep);
 // (page_ref_dec(page)将page的ref减1,并返回减1之后的ref
 if (page_ref_dec(page) == 0) //如果ref为0,则free即可
 {
 free_page(page);
 }
 //清除第二个页表 PTE
 *ptep = 0;
  //刷新 tlb，去除那些清除关系的二级页表，更新页目录项
 tlb_invalidate(pgdir, la);
 }
}

```

完成代码书写后，可以执行qemu，检测练习二和
练习三的代码是否正确，运行结果如下：

```
jzb@ubuntu:~/os_kernel_lab/labcodes/lab2$ make qemu
+ cc kern/mm/pmm.c
kern/mm/pmm.c:266:1: warning: ‘boot_alloc_page’ defined but not
used [-Wunused-function]
 boot_alloc_page(void) {
 ^~~~~~~~~~~~~~~
+ ld bin/kernel
+ ld bin/kernel_nopage
记录了10000+0 的读入
记录了10000+0 的写出
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0208672 s, 245 MB/s
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000204289 s, 2.5 MB/s
记录了265+1 的读入
记录了265+1 的写出
135832 bytes (136 kB, 133 KiB) copied, 0.000493457 s, 275 MB/s
WARNING: Image format was not specified for 'bin/ucore.img' and
probing guessed raw.
 Automatically detecting the format is dangerous for raw
images, write operations on block 0 will be restricted.
 Specify the 'raw' format explicitly to remove the
restrictions.
(THU.CST) os is loading ...
Special kernel symbols:
 entry 0xc0100036 (phys)
 etext 0xc0106f2e (phys)
 edata 0xc011d000 (phys)
 end 0xc02a4960 (phys)
Kernel executable memory footprint: 1683KB
ebp:0xc0119f38 eip:0xc0100aa9 arg:0x00010094 0x00010094
0xc0119f68 0xc01000cd
 kern/debug/kdebug.c:305: print_stackframe+21
ebp:0xc0119f48 eip:0xc0100d9f arg:0x00000000 0x00000000
0x00000000 0xc0119fb8
```

```
 kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0119f68 eip:0xc01000cd arg:0x00000000 0xc0119f90
0xffff0000 0xc0119f94
 kern/init/init.c:48: grade_backtrace2+33
ebp:0xc0119f88 eip:0xc01000f7 arg:0x00000000 0xffff0000
0xc0119fb4 0x0000002b
 kern/init/init.c:53: grade_backtrace1+38
ebp:0xc0119fa8 eip:0xc0100116 arg:0x00000000 0xc0100036
0xffff0000 0x0000001d
 kern/init/init.c:58: grade_backtrace0+23
ebp:0xc0119fc8 eip:0xc010013c arg:0xc0106f5c 0xc0106f40
0x00187960 0x00000000
 kern/init/init.c:63: grade_backtrace+34
ebp:0xc0119ff8 eip:0xc010008b arg:0xc010713c 0xc0107144
0xc0100d27 0xc0107163
 kern/init/init.c:28: kern_init+84
memory management: default_pmm_manager
e820map:
 memory: 0009fc00, [00000000, 0009fbff], type = 1.
 memory: 00000400, [0009fc00, 0009ffff], type = 2.
 memory: 00010000, [000f0000, 000fffff], type = 2.
 memory: 07ee0000, [00100000, 07fdffff], type = 1.
 memory: 00020000, [07fe0000, 07ffffff], type = 2.
 memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
 |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
 |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
 |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ----------------------
++ setup timer interrupts
0: @ring 0
0: cs = 8
0: ds = 10
0: es = 10
0: ss = 10
+++ switch to user mode +++
100 ticks
100 ticks
100 ticks

```

通过运行结果，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，探测出计算机系统中的物理内存的布局（e820map下的显示内容）。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，完成二级页表的建立，进入分页模式，执行各种我们设置的检查，最后显示ucore建立好的二级页表内容，并在分页模式下响应时钟中断。

数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥？

pages的每一项与页表中的页目录项和页表项有对应，pages每一项对应一个物理页的信息。一个页目录项对应一个页表，一个页表项对应一个物理页。假设有N个物理页，pages的长度为N，而页目录项、页表项的前20位对应物理页编号。
一个PDE对应1024个PTE，一个PTE对应1024个page。

如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事？ 鼓励通过编程来具体完成这个问题？

物理地址和虚拟地址之间存在offset：

```
phy addr + KERNBASE = virtual addr
```

所以，KERNBASE = 0时，phy addr = virtual addr
所以把memlayout.h中的

```
/* 映射到此地址的所有物理内存 */
#define KERNBASE 0x00000000
```

将KERNBASE改为0即可。