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。

打开default_pmm.c文件，阅读注释。

按照注释要求阅读库文件list.h,其中包含了双向指针数据结构结构 list_entry_t(list_entry),以及list_init(双向指针初始化), list_add(list_add_after)(双向指针尾插), list_add_before(双向指针头插), list_del(删除双向指针), list_next(返回下一个双向指针), list_prev(返回前一个双向指针)函数。

仔细查看和理解default_pmm.c中的注释

物理页数据结构Page (kern/mm/memlayout.h)

```
struct Page {
    int ref;                // page frame's reference counter
    uint32_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:物理页帧引用计数；
- flags:页帧状态描述标志；
- property：从此页之后连续自由块数；
- page_link：链表指针；

空闲内存空间块列表结构free_area_t (kern/mm/mmlayout.h)

```
typedef struct {
    list_entry_t free_list;             // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
```

- free_link:链表头指针;
- nr_free:自由页数；

物理内存管理类pmm_managerd 对象default_pmm_manager(kern/mm/pmm.h)

```
struct pmm_manager {
    const char *name;       // XXX_pmm_manager's name
    void (*init)(void);     // initialize internal description&management data
structure
                            // (free block list, number of free block) of
XXX_pmm_manager
    void (*init_memmap)(struct Page *base, size_t n);
```

```
        // setup description&management data structcure according to the initial
free physical memory space
    struct Page *(*alloc_pages)(size_t n);
        // allocate >=n pages, depend on the allocation algorithm
    void (*free_pages)(struct Page *base, size_t n);
        // free >=n pages with "base" addr of Page descriptor
structures(memlayout.h)
    size_t (*nr_free_pages)(void);      // return the number of free pages
    void (*check)(void);                // check the correctness of
XXX_pmm_manager
};
const struct pmm_manager default_pmm_manager = {
    .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,
};
```

对象名称：
default_pmm_manager;

方法函数：

- default_init(初始化空闲块),
- default_init_memmap(初始化空闲块列表),
- default_alloc_pages(分配空闲页),
- default_free_pages(),
- default_nr_free_pages(释放页),
- default_check(验证函数);

实现First fit算法的相关函数：default_init，default_init_memmap，default_alloc_pages，default_free_pages。

```
default_init代码：

free_area_t free_area;//实体化空闲空间块列表结构 free_area

//宏定义
#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

static void
default_init(void) {
    list_init(&free_list);//调用list_init()函数初始化free_list
    nr_free = 0;//初始化nr_free
}
```

调用堆栈：
kern_init --> pmm_init-->page_init-->init_memmap--> pmm_manager->init_memmap

内核初始化函数 -(调用)- 物理内存初始化函数 -(调用)- 整体物理地址的初始化函数 -(调用)- 初始化空闲块列表函数 -(调用)- 物理地址管理函数 -(调用)- 初始化空闲块列表函数

default_init_memmap()代码：

```
static void
default_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);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    // list_add(&free_list, &(base->page_link));
    // first-fit算法要求将空闲内存块按照地址从小到大的方式连起来。
    list_add_before(&free_list, &(base->page_link));
}
```

default_alloc_pages()代码：

```
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;
        }
    }
    if (page != NULL) {
    //     list_del(&(page->page_link));
    //     if (page->property > n) {
    //         struct Page *p = page + n;
    //         p->property = page->property - n;
    //         list_add(&free_list, &(p->page_link));
    // }
    //     nr_free -= n;
    //     ClearPageProperty(page);

        // alloc n and replace original page
        if (page->property > n){
            struct Page *p = page + n;// only use 0..n-1
            p->property = page->property - n;// page size decrease n
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
```

```
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

其中le2page(le, page_link)可将list_entry_t转换为page类型；

首先判断空闲页的大小是否大于所需的页块大小，如果请求页数大于物理页数，返回NULL；

否则，扫描free_area_t找到适合的页，使该页p->property >= n，则重新设置标志位，调用SetPageReserved(pp)和ClearPageProperty(pp)，设置当前页面预留，以及清空该页面的连续空闲页面数量值。然后从空闲链表，即free_area_t中，记录空闲页的链表，删除此项。

如果当前空闲页的大小大于所需大小，则分割页块，具体操作就是，刚刚分配了n个页，如果分配完了，还有连续的空间，则在最后分配的那个页的下一个页（未分配），更新它的连续空闲页值。如果正好合适，则不进行操作。

最后计算剩余空闲页个数并返回分配的页块地址。

default_free_pages()代码：

```
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));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        // space is after base
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        // space is before base
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    // list_add(&free_list, &(base->page_link));

    le = list_next(&free_list);
    // if can not merge find a position and add free link into free list
    while (le != &free_list){
        p = le2page(le, page_link);
```

```
        if (base + base->property <= p){
            assert(base + base->property != p);
            break;
        }
        le = list_next(le);
    }
    list_add_before(le, &(base->page_link));
}
```

default_free_pages主要完成的是对于页的释放操作。
首先使用一个assert语句断言这个基地址所在的页是否为预留，如果不是预留页，那么说明它已经是free状态，无法再次free，也就是之前所述，只有处在占用的页，才能有free操作。

之后，声明一个页p，p遍历一遍整个物理空间，直到遍历到base所在位置停止，开始释放操作。

找到了这个基地址之后呢，就可以将空闲页重新加进来（之前在分配的时候，删除了），之后就是一系列与初始化空闲页一样的设置标记位操作了。

之后，如果插入基地址附近的高地址或低地址可以合并，那么需要更新相应的连续空闲页数量，向高合并和向低合并。

你的first fit算法是否有进一步的改进空间?

对原有的算法进行了优化

我们可以用二叉搜索树来对内存进行管理。用二叉搜索树主要是通过对地址排序，使得在使用free时候可以在O(logn)时间内完成链表项位置的查找，从而实现时间上的优化。

练习2：实现寻找虚拟地址对应的页表项（需要编程）

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。

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）。

pde_t 全称为page directory entry，也就是一级页表的表项，前10位；

pte_t 全称为page table entry，表示二级页表的表项，中10位。

```
//get_pte - get pte and return the kernel virtual address of this pte for la
//        - if the PT contians this pte didn't exist, alloc a page for PT
// parameter:
//  pgdir:  the kernel virtual base address of PDT
//  la:     the linear address need to map
//  create: a logical value to decide if alloc a page for PT
// return vaule: the kernel virtual address of this pte
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
```

```
    /* LAB2 EXERCISE 2: YOUR CODE
     *
     * If you need to visit a physical address, please use KADDR()
     * please read pmm.h for useful macros
     *
     * Maybe you want help comment, BELOW comments can help you finish the code
     *
     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
     *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
     *   KADDR(pa) : takes a physical address and returns the corresponding
kernel virtual address.
     *   set_page_ref(page,1) : means the page be referenced by one time
     *   page2pa(page): get the physical address of memory which this (struct
Page *) page  manages
     *   struct Page * alloc_page() : allocation a page
     *   memset(void *s, char c, size_t n) : sets the first n bytes of the memory
area pointed by s
     *                                       to the specified value c.
     * DEFINEs:
     *   PTE_P           0x001                   // page table/directory entry
flags bit : Present
     *   PTE_W           0x002                   // page table/directory entry
flags bit : Writeable
     *   PTE_U           0x004                   // page table/directory entry
flags bit : User can access
     */
    /*
#if 0
    pde_t *pdep = NULL;   // (1) find page directory entry
    if (0) {              // (2) check if entry is not present
                          // (3) check if creating is needed, then alloc page for
page table
                          // CAUTION: this page is used for page table, not for
common data page
                          // (4) set page reference
        uintptr_t pa = 0; // (5) get linear address of page
                          // (6) clear page content using memset
                          // (7) set page directory entry's permission
    }
    return NULL;          // (8) return page table entry
#endif
    */
    pde_t *pdep = &pgdir[PDX(la)];// get pde physical address from virtual
address
    // pgdir + offset

    if(!(*pdep & PTE_P)){// not present 条目不可用
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL){
            // allocate failed
            return NULL;
        }
        set_page_ref(page, 1);// reference count set to 1
        uintptr_t pa = page2pa(page); // get the physical address of memory which
this (struct Page *) page  manages
        memset(KADDR(pa), 0, PGSIZE);
```

```
        *pdep = pa | PTE_U | PTE_W | PTE_P; // Present, Writeable, User can
access
    }
    // 返回在pgdir中对应于la的二级页表项
    return &((pte_t *)KADDR(PDE_ADDR(*pdep)))[PTX(la)];
}
```

首先判断pte是否可用

如果不可用就分配一个新的页

将该页的引用计数设置为1

获取页面的物理地址 并清空页内数据

设置该页的权限为 可用 可写 用户可获取

返回值解释：

[PTX](la):

取la的中间10bit 也就是二级页表的偏移量

((pte_t *)KADDR(PDE_ADDR(*pdep))) :

取出一级页表项的高20位作为二级页表的基地址

最后取地址返回pte_t*类型的指针

请描述页目录项（Pag Director Entry）和页表（Page Table Entry）中每个组成部分的含义和以及对ucore而言的潜在用处。

每个页表项（PTE）都由一个32位整数来存储数据，其结构如下

```
COPY      31-12      9-11     8    7    6   5   4      3    2   1   0
+--------------+-------+-----+----+---+---+-----+-----+---+---+---+
|     Offset   | Avail | MBZ | PS | D | A | PCD | PWT | U | W | P |
+--------------+-------+-----+----+---+---+-----+-----+---+---+---+
```

- 0 - Present: 表示当前PTE所指向的物理页面是否驻留在内存中
- 1 - Writeable: 表示是否允许读写
- 2 - User: 表示该页的访问所需要的特权级。即User(ring 3)是否允许访问
- 3 - PageWriteThough: 表示是否使用write through缓存写策略
- 4 - PageCacheDisable: 表示是否**不对**该页进行缓存
- 5 - Access: 表示该页是否已被访问过
- 6 - Dirty: 表示该页是否已被修改
- 7 - PageSize: 表示该页的大小
- 8 - MustBeZero: 该位必须保留为0
- 9-11 - Available: 第9-11这三位并没有被内核或中断所使用，可保留给OS使用。
- 12-31 - Offset: 目标地址的后20位。

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？

- 将引发页访问异常的地址将被保存在cr2寄存器中
- 设置错误代码
- 引发Page Fault

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 - free an Page sturct which is related linear address la
//                - and clean(invalidate) pte which is related linear address la
//note: PT is changed, so the TLB need to be invalidate
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    /* LAB2 EXERCISE 3: YOUR CODE
     *
     * Please check if ptep is valid, and tlb must be manually updated if mapping
is updated
     *
     * Maybe you want help comment, BELOW comments can help you finish the code
     *
     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
     *   struct Page *page pte2page(*ptep): get the according page from the value
of a ptep
     *   free_page : free a page
     *   page_ref_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 ,
then this page should be free.
     *   tlb_invalidate(pde_t *pgdir, uintptr_t la) : Invalidate a TLB entry, but
only if the page tables being
     *                        edited are the ones currently in use by the
processor.
     * DEFINEs:
     *   PTE_P           0x001                   // page table/directory entry
flags bit : Present
     */
/*
#if 0
    if (0) {                      //(1) check if this page table entry is present
        struct Page *page = NULL; //(2) find corresponding page to pte
                                  //(3) decrease page reference
                                  //(4) and free this page when page reference
reachs 0
                                  //(5) clear second page table entry
                                  //(6) flush tlb
    }
#endif
*/
    if (*ptep & PTE_P) { // if present
```

```
        struct Page *page = pte2page(*ptep);// get the according page from the
value of a ptep

        if (page_ref_dec(page) == 0) {
            // if reference count is zero free page
            free_page(page);
        }
        // PTE clear
        *ptep = 0;
        // refresh tlb data
        tlb_invalidate(pgdir, la);
    }
}
```

pde_t 全称为page directory entry，也就是一级页表的表项，前10位；

pte_t 全称为page table entry，表示二级页表的表项，中10位。

首先判断二级页表项是否有效 无效直接不用进行任何操作

通过二级页表项获取页面的结构体

查看该页面的引用计数 如果引用计数为零 就直接释放该页面的空间

清空二级页表表项

最后刷新tlb缓存

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？

页目录项或页表项有效时，Page数组中的项与页目录项或页表项存在对应关系。

页目录表中存放着数个页表条目PTE，这些页表条目中存放了某个二级页表所在物理页的信息，包括该二级页表的**物理地址**，但使用**线性地址**的头部PDX(Page Directory Index)来索引页目录表。

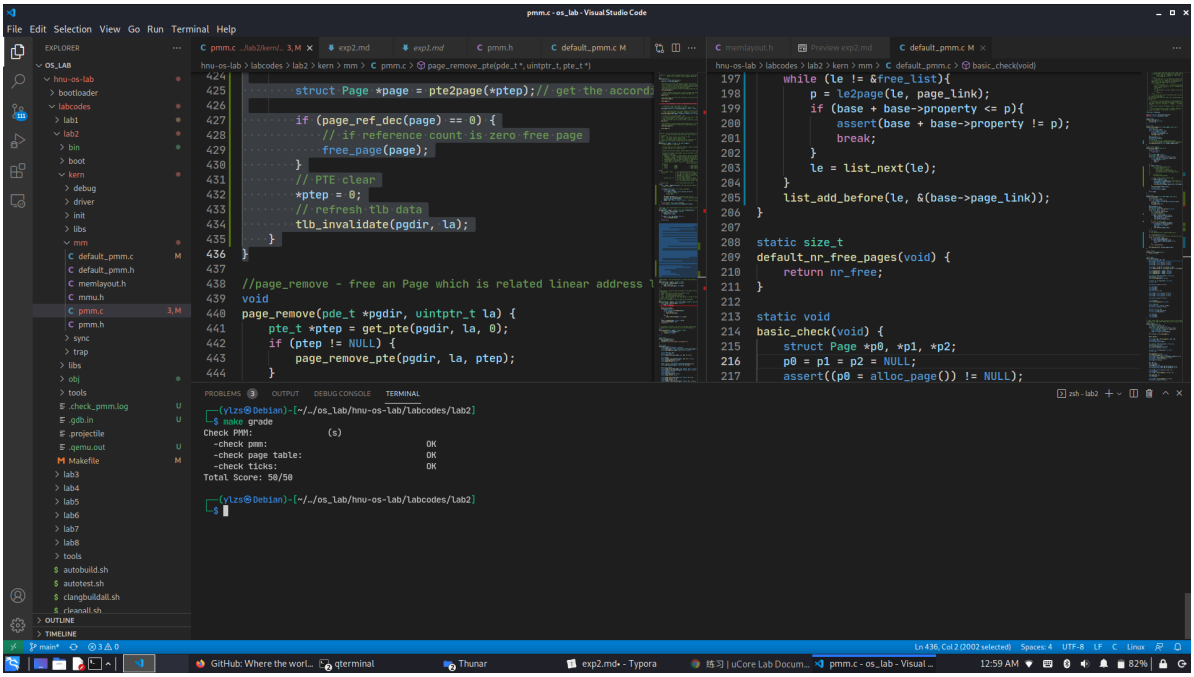
而页表（二级页表）与页目录（一级页表）具有类似的特性，页表中的页表项指向所管理的物理页的**物理地址**（不是数据结构Page的地址），使用线性地址的中部PTX(Page Table Index)来索引页表。

当二级页表获取物理页时，需要对该物理页所对应的数据结构page来做一些操作。其操作包括但不限于设置引用次数，这样方便共享内存。

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？

- 修改kernel的加载地址为0x0
- 将mm/中的内核偏移地址修改为0x0
- 关闭页机制

结果测试：



测试结果正确 得分为满分

扩展练习 伙伴分配算法

```
/*
free_area.free_list中的内存块顺序:

1. 一大块连续物理内存被切割后，free_area.free_list中的内存块顺序
    addr: 0x34        0x38            0x40
        +----+      +--------+      +---------------+
    <-> | 0x4| <-> | 0x8    | <-> |      0x10       | <->
        +----+      +--------+      +---------------+

2. 几大块物理内存（这几块之间可能不连续）被切割后，free_area.free_list中的内存块顺序
    addr: 0x34        0x104        0x38            0x108           0x40
  0x110
        +----+      +----+      +--------+      +--------+      +---------------+
+---------------+
    <-> | 0x4| <-> | 0x4| <-> | 0x8    | <-> | 0x8    | <-> |      0x10       | <->
|      0x10       | <->
        +----+      +----+      +--------+      +--------+      +---------------+
+---------------+
*/
```

编写初始化函数如下

```
static void
buddy_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);
    }
```

```
    nr_free += n;
    base += n;
    while(n != 0)
    {
        size_t curr_n = getLessNearOfPower2(n);
        base -= curr_n;
        base->property = curr_n;
        SetPageProperty(base);
        list_entry_t* le;
        for(le = list_next(&free_list); le != &free_list; le = list_next(le))
        {
            struct Page *p = le2page(le, page_link);

            if((p->property > base->property)
                 || (p->property ==  base->property && p > base))
                break;
        }
        list_add_before(le, &(base->page_link));
        n -= curr_n;
    }
}
```

编写分配空间的函数如下

```
static struct Page *
buddy_alloc_pages(size_t n) {
    assert(n > 0);

    size_t lessOfPower2 = getLessNearOfPower2(n);
    if (lessOfPower2 < n)
        n = 2 * lessOfPower2;

    if (n > nr_free) {
        return NULL;
    }
    // 寻找一块合适的内存块
    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;
        }
    }

    if (page != NULL) {
        // 切割内存块
        while(page->property > n)
        {
            page->property /= 2;
            struct Page *p = page + page->property;
            p->property = page->property;
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
        }
        nr_free -= n;
```

```
        ClearPageProperty(page);
        assert(page->property == n);
        list_del(&(page->page_link));
    }
    return page;
}
```

编写内存释放函数如下

```
static void
buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);

    size_t lessOfPower2 = getLessNearOfPower2(n);
    if (lessOfPower2 < n)
        n = 2 * lessOfPower2;
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    list_entry_t *le;
    // 插入双向链表
    for(le = list_next(&free_list); le != &free_list; le = list_next(le))
    {
        p = le2page(le, page_link);
        if ((base->property < p->property)
                || (p->property ==  base->property && p > base)) {
            break;
        }
    }
    list_add_before(le, &(base->page_link));
    // 向左合并
    if(base->property == p->property && p + p->property == base) {
        p->property += base->property;
        ClearPageProperty(base);
        list_del(&(base->page_link));
        base = p;
        le = &(base->page_link);
    }
    // 向右合并
    while (le != &free_list) {
        p = le2page(le, page_link);
        if (base->property == p->property && base + base->property == p)
        {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
            le = &(base->page_link);
        }
        else if(base->property < p->property)
        {
            list_entry_t* targetLe = list_next(&base->page_link);
```

```
            while(le2page(targetLe, page_link)->property < base->property)
                targetLe = list_next(targetLe);
            if(targetLe != list_next(&base->page_link))
            {
                list_del(&(base->page_link));
                list_add_before(targetLe, &(base->page_link));
            }
            break;
        }
        le = list_next(le);
    }
}
```

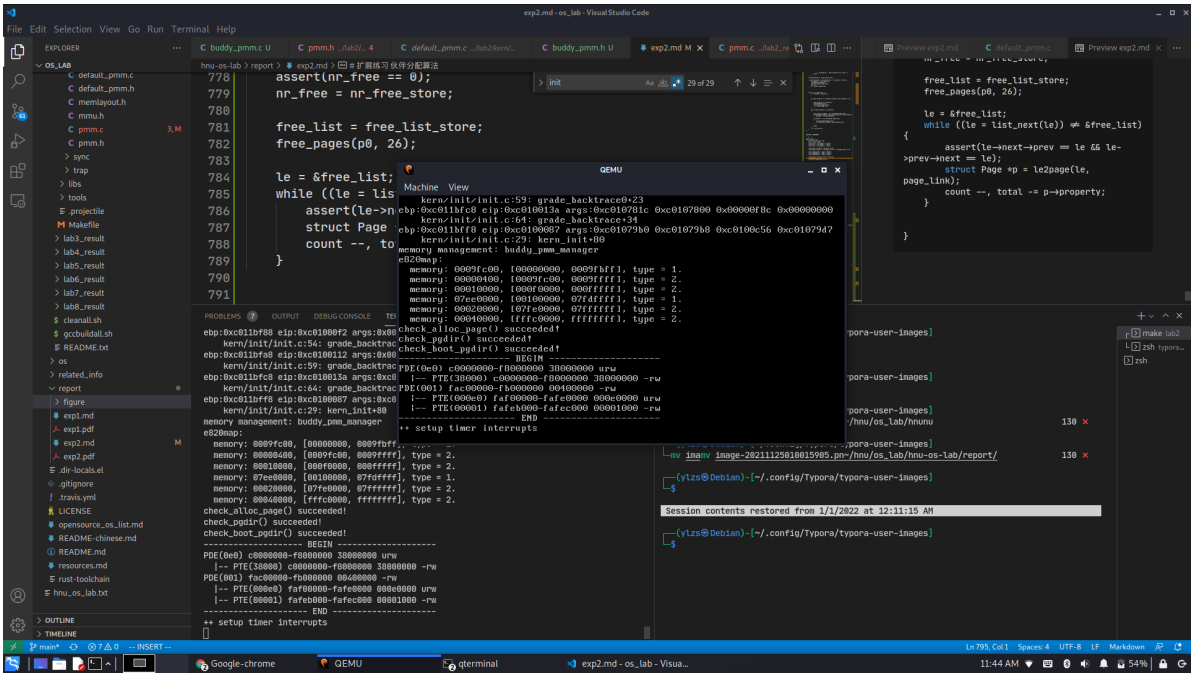
最后编写检查代码如下

```
//.....................................................
// 先释放
free_pages(p0, 26);     // 32+  (-:已分配 +: 已释放)
// 首先检查是否对齐2
p0 = alloc_pages(6);    // 8- 8+ 16+
p1 = alloc_pages(10);   // 8- 8+ 16-
assert((p0 + 8)->property == 8);
free_pages(p1, 10);     // 8- 8+ 16+
assert((p0 + 8)->property == 8);
assert(p1->property == 16);
p1 = alloc_pages(16);   // 8- 8+ 16-
// 之后检查合并
free_pages(p0, 6);      // 16+ 16-
assert(p0->property == 16);
free_pages(p1, 16);     // 32+
assert(p0->property == 32);

p0 = alloc_pages(8);    // 8- 8+ 16+
p1 = alloc_pages(9);    // 8- 8+ 16-
free_pages(p1, 9);     // 8- 8+ 16+
assert(p1->property == 16);
assert((p0 + 8)->property == 8);
free_pages(p0, 8);      // 32+
assert(p0->property == 32);
// 检测链表顺序是否按照块的大小排序的
p0 = alloc_pages(5);
p1 = alloc_pages(16);
free_pages(p1, 16);
assert(list_next(&(free_list)) == &((p1 - 8)->page_link));
free_pages(p0, 5);
assert(list_next(&(free_list)) == &(p0->page_link));

p0 = alloc_pages(5);
p1 = alloc_pages(16);
free_pages(p0, 5);
assert(list_next(&(free_list)) == &(p0->page_link));
free_pages(p1, 16);
assert(list_next(&(free_list)) == &(p0->page_link));

// 还原
p0 = alloc_pages(26);
//.....................................................
```


exp2.md - os_lab - Visual Studio Code
QEMU
Machine View
kern/init/init.c:59: grade_backtrace0+23
kern/init/init.c:64: grade_backtrace+34
kern/init/init.c:29: kern_init+80
memory management: buddy_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
Session contents restored from 1/1/2022 at 12:11:15 AM
Google-chrome
QEMU
qterminal
exp2.md - os_lab - Visua...

实验成功