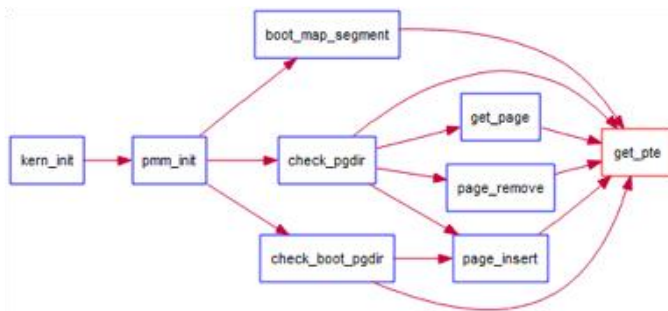


《操作系统》实验报告

实验题目	内存管理
<div>一、实验目的</div> <div>1. 理解基于页表的转换机制;</div> <div>2. 理解页表的建立和使用方法;</div> <div>3. 理解物理内存的管理方法;</div>	
<div>二、实验项目内容</div> <div>1. 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。请在仔细阅读实验指导书和源码的基础上理解 ucore 操作系统中 first fit 内存分配算法的实现过程，并修改 default_pmm.c 中的 default_init, default_init_memmap, default_alloc_pages, default_free_pages 等相关函数来重新实现算法。请在实验报告中简要说明你的设计实现过程，并思考你的 first fit 算法是否有进一步的改进空间?</div> <div>2. 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。请在仔细阅读和理解 get_pte 函数中的注释，编程补全 get_pte 函数 (kern/mm/pmm.c 中) 代码，实现其功能。get_pte 函数的调用关系图如下所示:</div> <div data-bbox="426 1151 1096 1460"><pre>graph LR; kern_init --> pmm_init; pmm_init --> boot_map_segment; pmm_init --> check_pgdir; pmm_init --> check_boot_pgdir; boot_map_segment --> get_page; check_pgdir --> get_page; check_pgdir --> page_remove; check_boot_pgdir --> page_insert; get_page --> get_pte; page_remove --> get_pte; page_insert --> get_pte; get_pte --> page_remove;</pre></div> <div>图 1. get_pte 函数的调用关系图</div> <div>请在实验报告中简要说明你的设计实现过程，并描述页目录项 (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。</div> <div>3. 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，并编程补全在 kern/mm/pmm.c 中的 page_remove_pte 函数代码。</div> <div>page_remove_pte 函数的调用关系图如下所示:</div>	

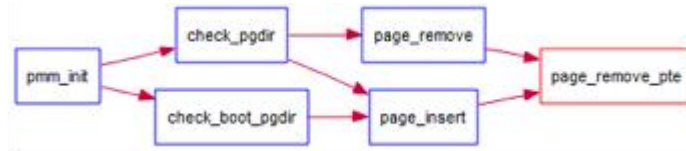


图 2. page_remove_pte 函数的调用关系图

请在实验报告中简要说明你的设计实现过程，并回答数据结构 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，其对应关系是什么？

三、实验过程或算法（源程序）

实验过程：

1. 前期准备：连接到 WSL，进入 Ubuntu，启动 Docker 并进入容器，从仓库中克隆不含答案的项目；

2. 修改 Makefile:

```

LAB1    := -DLAB1_EX2 -DLAB1_EX3 #-D_SHOW_100_TICKS -D_SHOW_SERIAL_INPUT
LAB2    := -DLAB2_EX1 -DLAB2_EX2 -DLAB2_EX3
# LAB3  := -DLAB3_EX1 -DLAB3_EX2
# LAB4  := -DLAB4_EX1 -DLAB4_EX2
  
```

需要将 LAB3 和 LAB4 注释掉，因为本次实验只实现 LAB2；

3. 复写 lab2/ucore-loongarch32/kern/mm/default_pmm.c 文件中的 default_init, default_init_memmap, default_alloc_pages 和 default_free_pages 函数：

```

//复写函数
static void
default_init(void) {
    list_init(&free_list);    //定义表头
    nr_free = 0;              //定义空闲页个数
}

static void
default_init_memmap(struct Page *base, size_t n) {
#ifdef LAB2_EX1
    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_before(&free_list, &(base->page_link));
#endif
}
  
```

```

static struct Page *
default_alloc_pages(size_t n) {
#ifdef LAB2_EX1
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    // TODO: optimize (next-fit)
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
#endif
}

```

```

static void
default_free_pages(struct Page *base, size_t n) {
#ifdef LAB2_EX1
    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);
        // TODO: optimize
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    le = list_next(&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));
#endif
}

```

其中在实现 first-fit 内存分配算法的回收函数时，考虑了地址连续的空闲块之间的合并操作。在建立空闲页块链表时，按照空闲页块起始地址来排序，形成一个有序的链

表。

4. 设置页表和对应的页表项，建立虚拟内存地址和物理内存地址的对应：

```
pte_t *
get_pte(pde_t *pgdir, uintptr_t la, bool create) {
#ifdef LAB2_EX2
    pde_t *pdep = NULL;           // 查找页面页表项
    pdep = pgdir + PDX(la);

    if ((*pdep) & PTE_P) == 0){    // 检查页表项是否存在
        // 检查是否需要创建，然后为页面表分配页面
        if (!create)
            return NULL;
        // 建立页表引用
        struct Page *new_pte = alloc_page();
        if (!new_pte)
            return NULL;
        page_ref_inc(new_pte);
        uintptr_t pa = (uintptr_t)page2kva(new_pte); // 获得页面的虚拟地址
        // 使用memset清除页面内容
        memset((void *)pa, 0, PGSIZE);
        // 设置页表项的权限
        *pdep = PADDR(pa);
        (*pdep) |= (PTE_U | PTE_P | PTE_W);
    }
    pte_t *ret = (pte_t *)KADDR((uintptr_t)((pte_t *) (PDE_ADDR(*pdep)) + PTX(la)));
    return ret;                    // 返回页表项
#else
    /* LAB2_EXERCISE2: YOUR CODE */
#endif
}
```

具体流程是：先定义一个指针，将该指针指向虚地址 `la` 对应的页表项的地址。如果该页表项不存在，就分配一个页面用来装载该页表项。如果系统设置不可创建或无法再分配页面，则函数返回空地址。成功分配页面后，通过 `page2kva` 函数获得页面的虚拟地址，并通过将整页内容全部清零完成初始化。最后再使用函数将虚拟内存地址转换为物理内存地址。

5. 释放虚地址所在的页并取消对应的二级页表项的映射：

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
#ifdef LAB2_EX3
    if (ptep && (*ptep & PTE_P)){    // 检查页面目录是否存在
        struct Page *page = pte2page(*ptep); // 找到与页表项对应的页面
        // 减少页面引用
        page_ref_dec(page);
        // 如果页表引用数减至0就释放它
        if (page_ref(page) == 0){
            free_page(page);
        }
        // 清除页面目录条目
        *ptep = 0;
    }
    // 清空快表
    tlb_invalidate_all();
#else
    /* LAB2_EXERCISE3: YOUR CODE */
#endif
}
```

具体流程是：先通过按位与判断页面是否存在，若页面存在，就不断减少其页面引用

直至减少到 0。然后释放该空间，并将页面条目数置为 0。完成上述工作后，还需要清空快表。

6. 完成以上代码编写后，编译运行该项目，运行结果如下：

```
● root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125# cd lab2/ucore-loongarch32
● root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab2/ucore-loongarch32# make clean
rm -rf dep
rm -rf boot/loader.o boot/loader boot/loader.bin
rm -rf obj
● root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab2/ucore-loongarch32# make
DEP kern/fs/devs/dev_stdout.c
DEP kern/fs/devs/dev_stdin.c
DEP kern/fs/devs/dev_disk0.c
DEP kern/fs/devs/dev.c
DEP kern/fs/sfs/sfs_lock.c
DEP kern/fs/sfs/sfs_io.c
DEP kern/fs/sfs/sfs_inode.c
DEP kern/fs/sfs/sfs_fs.c
DEP kern/fs/sfs/sfs.c
DEP kern/fs/sfs/bitmap.c
DEP kern/fs/vfs/vfspath.c
DEP kern/fs/vfs/vfslookup.c
DEP kern/fs/vfs/vfsfile.c
DEP kern/fs/vfs/vfsdev.c
DEP kern/fs/vfs/vfs.c
DEP kern/fs/vfs/inode.c

○ root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab2/ucore-loongarch32# make qemu -j 16
loongson32_init: num_nodes 1
loongson32_init: node 0 mem 0x2000000
++setup timer interrupts
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xA0000120 (phys)
  etext 0xA0021000 (phys)
  edata 0xA017B440 (phys)
  end 0xA017E720 (phys)
Kernel executable memory footprint: 1398KB
memory management: default_pmm_manager
memory map:
  [A0000000, A2000000]

freemem start at: A01BF000
free pages: 00001E41
## 00000020
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_slab() succeeded!
kmalloc_init() succeeded!
LAB2 Check Pass!
□
```

可以看到，ucore 显示了内核的一些分配情况（入口地址 entry、代码段截止地址 etext、数据段截止处地址 edata、ucore 截止地址 end、内存范围 memory map、空闲内存的起始地址以及可以分为多少个 page 等），探测出计算机系统物理内存的布局。最后执行了各种代码中设置的检查，然后响应了时钟中断。

实验算法函数源代码：

练习 1. 实现 first-fit 连续物理内存分配算法：

```
//复写函数
```



```
static void
default_init(void) {
    list_init(&free_list);      //定义表头
    nr_free = 0;                //定义空闲页个数
}
```

```
static void
default_init_memmap(struct Page *base, size_t n) {
#ifdef LAB2_EX1
    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_before(&free_list, &(base->page_link));
#endif
}
```

```
static struct Page *
default_alloc_pages(size_t n) {
#ifdef LAB2_EX1
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    // TODO: optimize (next-fit)
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
        }
    }
}
```

```

        SetPageProperty(p);
        list_add_after(&(page->page_link), &(p->page_link));
    }
    list_del(&(page->page_link));
    nr_free -= n;
    ClearPageProperty(page);
}
return page;
#endif
}

```

```

static void
default_free_pages(struct Page *base, size_t n) {
#ifdef LAB2_EX1
    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);
        // TODO: optimize
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    le = list_next(&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));
#endif
}

```

设计思路见下一部分练习答案

练习 2. 实现寻找虚拟地址对应的页表项:

```

get_pte(pde_t *pgdir, uintptr_t la, bool create) {
#ifdef LAB2_EX2

    pde_t *pdep = NULL;          // 查找页面页表项
    pdep = pgdir + PDX(la);

    if (((*pdep) & PTE_P) == 0){ // 检查页表项是否存在
        // 检查是否需要创建，然后为页面表分配页面
        if (!create)
            return NULL;
        // 建立页表引用
        struct Page *new_pte = alloc_page();
        if (!new_pte)
            return NULL;
        page_ref_inc(new_pte);
        uintptr_t pa = (uintptr_t)page2kva(new_pte); // 获得页面的虚拟地址
        // 使用 memset 清除页面内容
        memset((void *)pa, 0, PGSIZE);
        // 设置页表项的权限
        *pdep = PADDR(pa);
        (*pdep) |= (PTE_U | PTE_P | PTE_W);
    }
    pte_t *ret = (pte_t *)KADDR((uintptr_t)((pte_t *) (PDE_ADDR(*pdep)) +
PTX(la)));
    return ret;                  // 返回页表项
#else
    pde_t *pdep = NULL;
    if (0) {
        uintptr_t pa = 0;
    }
    return NULL;
#endif
}

```


在 kern/mm/pmm.c 文件下实现寻找虚拟地址对应的页表项。具体思路是：先定义一个指针，将该指针指向虚地址 la 对应的页表项的地址。如果该页表项不存在，就分配一个页面用来装载该页表项。如果系统设置不可创建或无法再分配页面，则函数返回空地址。成功分配页面后，通过 page2kva 函数获得页面的虚拟地址，并通过将整页内容全部清零完成初始化。最后再使用函数将虚拟内存地址转换为物理内存地址。

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：

```
static inline void
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
#ifdef LAB2_EX3
    if (ptep && (*ptep & PTE_P)){                // 检查页面目录是否存在
        struct Page *page = pte2page(*ptep);    // 找到与页表项对应的页面
        // 减少页面引用
        page_ref_dec(page);
        // 如果页表引用数减至 0 就释放它
        if (page_ref(page) == 0){
            free_page(page);
        }
        // 清除页面目录条目
        *ptep = 0;
    }
    // 清空快表
    tlb_invalidate_all();
#else
    if (0) {
        struct Page *page = NULL;
    }
#endif
}
```

在 kern/mm/pmm.c 文件下释放某虚地址所在的页并取消对应二级页表项的映射。具体思路是：先通过按位与判断页面是否存在，若页面存在，就不断减少其页面引用直至减少到 0。然后释放该空间，并将页面条目数置为 0。完成上述工作后，还需要清空快表。

四、实验结果及分析

1. 实验结果截图：

```

○ root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab2/ucore-loongarch32# make qemu -j 16
loongson32_init: num_nodes 1
loongson32_init: node 0 mem 0x2000000
++setup timer interrupts
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0xA0000120 (phys)
  etext 0xA0021000 (phys)
  edata 0xA017B440 (phys)
  end   0xA017E720 (phys)
Kernel executable memory footprint: 1398KB
memory management: default_pmm_manager
memory map:
  [A0000000, A2000000]

freemem start at: A01BF000
free pages: 00001E41
## 00000020
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_slab() succeeded!
kmallocc_init() succeeded!
LAB2 Check Pass!

```

可以看到，ucore 显示了内核的一些分配情况，探测出计算机系统物理内存的布局：

- 入口地址：0xA0000120；
- 代码段截止处地址：0xA0021000；
- 数据段截止处地址：0xA017B440；
- ucore 截止处地址：0xA017E720；
- 内存范围：A0000000~A2000000；
- 空闲内存起始地址：00001E41；
- 可分配页数：20；

最后执行了各种代码中设置的检查，然后响应了时钟中断。

2. 练习：

练习 1：

- 页空闲块维护

`first_fit` 分配算法需要维护一个查找有序（地址按从小到大排列）空闲块（以页为最小单位的连续地址空间）的数据结构，双向链表是一个很好的选择。本实验中使用了 `libs/list.h` 定义的可挂接任意元素的通用双向链表，可以完成对双向链表的初始化/插入/删除等。具体来说，是定义了一个名为 `free_area_t` 的数据结构：

```

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

```

其中 `free_area_t` 包含了一个 `list_entry` 结构的双向链表指针和记录当前空闲页的个数的无符号整型变量 `nr_free`。其中的链表指针指向了空闲的物理页。

- 空闲页链表初始化

`default_init_memmap` 函数将根据每个物理页帧的情况来建立空闲页链表，也就是初始化了一个空闲块，空闲块应该是根据地址高低形成一个有序链表。地址从高到低的设计是为了在 `default_free_pages` 中实现空闲块的合并。初始化过程中 `default_init_memmap(base, n)` 会在物理内存中从 `base` 开始连续申请 `n` 个 Page，并对这些 Page 做初始化，比如将 `ref` 设置为 0（表示这一表项没有被其他虚拟页表引用），

将除第一页的其他 Page 的 property, flag 设置为 0, 表示 property 属性无效并且非预留, 将第一页 property 设置为 n, 表示这一个空闲块里面有 n 个 Page。

- 块分配

default_alloc_pages 函数从空闲页链表中寻找第一个大小足够的空闲块并分配 (first fit)。当想要申请 n 个 Pages, 先会判断 nr_free 是否大于等于 n, 如果小于则返回 NULL。firstfit 需要从空闲链表头开始查找最小的地址, 通过 list_next 找到下一个空闲块元素, 通过 le2page 宏可以由链表元素获得对应的 Page 指针 p。通过 p->property 可以了解此空闲块的大小。如果 $\geq n$, 就找到并返回, 如果 $< n$, 则 list_next, 继续查找。直到 list_next == &free_list, 这表示找完了一遍了。找到后, 就要从新组织空闲块, 然后把找到的 page 返回。

- 块释放

default_free_pages 函数的实现其实是 default_alloc_pages 的逆过程。先遍历这个块的所有页, 确认所有页都合法 (reserved==0&&property==0), 然后将需要释放的空间标记为空之后, 找到空闲表中合适的位置。由于空闲表中的记录都是按照物理页地址排序的, 所以如果插入位置的前驱或者后继刚好和释放后的空间邻接, 那么需要将新的空间与前后邻接的空间合并形成更大的空间。

- 改进

在不改变 first fit 算法的情况下, default_free_pages 函数中存在可优化空间。在 default_free_pages 函数中, 是通过遍历整个链表寻找可以合并的前驱后继节点的, 时间复杂度是 $O(N)$ 。如果把链表数据结构换成树结构等可以通过内存地址进行快速检索的结构, 就可以降低 default_free_pages 函数的时间复杂度。

练习 2: 页目录项是一个页存储系统的基本构成单元, 一个页就是一段连续的内存, 而页目录项就是构成页的每一项元素。一个物理地址可以映射到一个**页目录项**, 页目录项存储了真实的数据。而**页表项**是页表的基本构成单元, 一个页号可以映射到一个页表项, 然后通过页表项中的页帧号得到该页面的物理地址。页表项由页帧号组成, 提供虚页号到页帧号的映射, 通过页帧号与页内偏移量的拼接可以得到真实内存地址, 从而访问内存中的数据。

练习 3: 数据结构 Page 的全局变量的每一项与页表中的页目录项和页表项**有关**。虚地址由页目录项、页表项和页内偏移量组成, 对于任意有效的页目录项和页表项, 都可以用于索引一个 page。页目录项存放页表项的物理地址, 但用虚地址的 DIR 项来作为索引; 页表项存放物理页的物理地址, 由虚地址的 PAGE 项来索引。通过页目录项、页表项和页内偏移量就可以得到 page 的索引。

3. 实验感悟:

本次实验主要完成了 ucore 内核对物理内存的管理工作: 在 kern_init 函数完成一些指定输出后, 就进入了**物理内存管理初始化**的工作, 即调用 pmm_init 函数完成物理内存的管理。

为了完成物理内存管理, 这里首先需要探测可用的物理内存资源:**了解物理内存的位置与大小**。在确定了物理内存大小后, 就以固定页面大小来**划分整个物理内存空间**, 并准备以此为最小内存分配单位来管理整个物理内存。管理在内核运行过程中每页内存, 设定其可用状态 (free 的, used 的, 还是 reserved 的)。接着 ucore 内核就要**建立页表**, 启动**分页机制**, 当缺页异常发生时, 就会跳转到内核的异常处理地址上, 由内核完成 TLB 填充, 根据页表项描述的**虚拟页与物理页帧**的对应关系完成 CPU 对内存的读、写和执行操作。

当上述操作都完成后，执行 `intr_enable` 函数**开启中断**。