《操作系统》

lab2：物理内存和页表

一. 实验要求

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

练习1：理解first-fit 连续物理内存分配算法（思考题）

first-fit 连续物理内存分配算法作为物理内存分配一个很基础的方法，需要同学们理解它的实现过程。请大家仔细阅读实验手册的教程并结合`kern/mm/default_pmm.c`中的相关代码，认真分析default_init，default_init_memmap，default_alloc_pages， default_free_pages等相关函数，并描述程序在进行物理内存分配的过程以及各个函数的作用。 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?

练习2：实现 Best-Fit 连续物理内存分配算法（需要编程）

在完成练习一后，参考kern/mm/default_pmm.c对First Fit算法的实现，编程实现Best Fit页面分配算法，算法的时空复杂度不做要求，能通过测试即可。 请在实验报告中简要说明你的设计实现过程，阐述代码是如何对物理内存进行分配和释放，并回答如下问题：

- 你的 Best-Fit 算法是否有进一步的改进空间?

扩展练习Challenge：buddy system（伙伴系统）分配算法（需要编程）

Buddy System算法把系统中的可用存储空间划分为存储块(Block)来进行管理, 每个存储块的大小必须是2的n次幂(Pow(2, n)), 即1, 2, 4, 8, 16, 32, 64, 128...

- 参考伙伴分配器的一个极简实现， 在ucore中实现buddy system分配算法，要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

扩展练习Challenge：任意大小的内存单元slub分配算法（需要编程）

slub算法，实现两层架构的高效内存单元分配，第一层是基于页大小的内存分配，第二层是在第一层基础上实现基于任意大小的内存分配。可简化实现，能够体现其主体思想即可。

- 参考linux的slub分配算法/，在ucore中实现slub分配算法。要求有比较充分的测试用例说明实现的正确性，需要有设计文档。

扩展练习Challenge：硬件的可用物理内存范围的获取方法（思考题）

- 如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围?

二. 实验过程

练习1

first-fit主要实现过程：

first-fit 连续物理内存分配算法通过维护一个空闲内存块链表来管理可用内存：

- 当有内存分配请求时，算法从链表头部开始查找，选择第一个满足大小要求的空闲块进行分配。
- 在内存回收时，回收的内存块会按地址顺序插入链表，并与相邻的空闲块合并，以减少碎片。

函数default_init：

```
static void
default_init(void) {
  list_init(&free_list);
  nr_free = 0;
}
```

该函数通过调用 `list_init` 初始化双向链表 `free_list`，并将记录空闲块数量的变量 `nr_free` 置零，从而完成对空闲块链表的初始化。

函数default_init_memmap：

```
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;
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
```

```
    }
}
```

该函数用于初始化一段连续的空闲内存页。其执行流程如下：

1. **参数与准备**
   - 输入参数 `base` 为 `Page` 结构体数组的起始地址，代表一段连续的物理内存页。
   - 输入参数 `n` 指定了待初始化的页面数量。
   - 函数首先检查 `n > 0`，以确保有需要处理的页面。
2. **初始化各页面属性**
   - 通过循环遍历每个页面，若页面为非保留状态，则将其 `flags`、`property` 属性及引用计数 `ref` 均清零。
3. **设置首块元信息**
   - 将首个页面的 `property` 属性设置为 `n`，标记此连续空闲块的总页数。
4. **插入空闲链表**
   - 根据空闲链表 `free_list` 是否为空，执行不同操作：
     - **链表为空**：直接将此空闲块加入链表。
     - **链表非空**：遍历链表，按地址顺序找到第一个基地址大于 `base` 的块，并将新块插入其之前；若未找到，则插入链表尾部。
5. **更新全局计数**
   - 将新增的 `n` 个页面累加到全局空闲页计数器 `nr_free` 中。

函数default_alloc_pages：

```
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_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
```

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

函数default_free_pages：

```
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);
    nr_free += n;

    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }

    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    le = list_next(&(base->page_link));
    if (le != &free_list) {
        p = le2page(le, page_link);
```

```
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
    }
}
```

该函数负责分配一个包含 `n` 个页面的连续内存块。其核心流程如下：

1. **查找匹配块**：遍历空闲链表，找到第一个 `property`（所含空闲页数）大于等于 `n` 的块，记为 `page`。
2. **分割内存块**：从 `page` 指向的块中分割出大小为 `n` 的部分用于分配。剩余部分将作为一个新的空闲块保留在系统中。
3. **处理剩余块**：若分割后的剩余块大小（`property` 值）仍大于零，则更新其属性并将其作为新空闲块插回空闲链表。
4. **更新系统状态**：减少全局空闲页计数器 `nr_free`，并清除原内存块的空闲标志，标记其为已分配状态。

函数default_nr_free_pages：

```
static size_t
default_nr_free_pages(void) {
    return nr_free;
}
```

该函数用于获取系统中当前空闲页面的总数。

函数basic_check：

```
static void
basic_check(void) {
    struct Page *p0, *p1, *p2;
    p0 = p1 = p2 = NULL;
    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(p0 != p1 && p0 != p2 && p1 != p2);
    assert(page_ref(p0) == 0 && page_ref(p1) == 0 && page_ref(p2) == 0);

    assert(page2pa(p0) < npage * PGSIZE);
    assert(page2pa(p1) < npage * PGSIZE);
    assert(page2pa(p2) < npage * PGSIZE);

    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));

    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    assert(alloc_page() == NULL);
```

```
    free_page(p0);
    free_page(p1);
    free_page(p2);
    assert(nr_free == 3);

    assert((p0 = alloc_page()) != NULL);
    assert((p1 = alloc_page()) != NULL);
    assert((p2 = alloc_page()) != NULL);

    assert(alloc_page() == NULL);

    free_page(p0);
    assert(!list_empty(&free_list));

    struct Page *p;
    assert((p = alloc_page()) == p0);
    assert(alloc_page() == NULL);

    assert(nr_free == 0);
    free_list = free_list_store;
    nr_free = nr_free_store;

    free_page(p);
    free_page(p1);
    free_page(p2);
}
```

该测试模块用于验证以下核心功能的正确性：

- **页面分配**：确保能正确分配指定大小的连续物理内存。
- **引用计数**：验证页面在被分配、共享和释放时，其引用计数能准确更新。
- **空闲页面管理**：检查空闲页面的链接操作，包括插入、合并与删除，是否维护了链表的正确性与完整性。

函数default_check：

```
static void
default_check(void) {
    int count = 0, total = 0;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        assert(PageProperty(p));
        count ++, total += p->property;
    }
    assert(total == nr_free_pages());

    basic_check();

    struct Page *p0 = alloc_pages(5), *p1, *p2;
    assert(p0 != NULL);
    assert(!PageProperty(p0));
```

```
    list_entry_t free_list_store = free_list;
    list_init(&free_list);
    assert(list_empty(&free_list));
    assert(alloc_page() == NULL);

    unsigned int nr_free_store = nr_free;
    nr_free = 0;

    free_pages(p0 + 2, 3);
    assert(alloc_pages(4) == NULL);
    assert(PageProperty(p0 + 2) && p0[2].property == 3);
    assert((p1 = alloc_pages(3)) != NULL);
    assert(alloc_page() == NULL);
    assert(p0 + 2 == p1);

    p2 = p0 + 1;
    free_page(p0);
    free_pages(p1, 3);
    assert(PageProperty(p0) && p0->property == 1);
    assert(PageProperty(p1) && p1->property == 3);

    assert((p0 = alloc_page()) == p2 - 1);
    free_page(p0);
    assert((p0 = alloc_pages(2)) == p2 + 1);

    free_pages(p0, 2);
    free_page(p2);

    assert((p0 = alloc_pages(5)) != NULL);
    assert(alloc_page() == NULL);

    assert(nr_free == 0);
    nr_free = nr_free_store;

    free_list = free_list_store;
    free_pages(p0, 5);

    le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        count --, total -= p->property;
    }
    assert(count == 0);
    assert(total == 0);
}
```

该测试旨在对内存管理系统进行全面的验证，核心内容包括：

- **空闲链表完整性检验**：遍历空闲页面链表，验证其链接顺序的正确性及各节点`property`等属性的准确性。
- **分配功能测试**：模拟多种内存申请场景，测试分配算法的正确性及边界处理能力。
- **释放与合并测试**：检验页面释放后，是否能正确回收并与相邻空闲块进行合并，以消除碎片。

结构体default_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,
};
```

该结构体定义了内存管理器的统一接口，其各成员功能如下：

成员	类型	说明
`.name`	`char[]`	管理器名称标识，例如 "default_pmm_manager"。
`.init`	`function`	初始化内存管理器自身数据结构。
`.init_memmap`	`function`	根据物理内存布局，初始化可用页面的映射信息。
`.alloc_pages`	`function`	分配指定数量的连续物理页面。
`.free_pages`	`function`	释放已分配的物理页面，并合并空闲块。
`.nr_free_pages`	`function`	返回当前系统中可用的空闲页面总数。
`.check`	`function`	用于内部一致性检查与调试的接口。

各函数在物理内存分配过程中的作用小结：

`first-fit` 连续内存分配算法通过以下核心函数实现：

- `default_init`：初始化空闲块链表，并将空闲块计数置零。
- `default_init_memmap`：初始化一个连续空闲内存块，并按地址顺序将其插入链表，同时更新空闲页总数。
- `default_alloc_pages`：分配指定大小的内存块。它从链表中查找首个足够大的块进行分割，分配所需部分，并更新剩余块信息。
- `default_free_pages`：释放内存块。将其按地址顺序插回链表，并与相邻空闲块合并，以减少外部碎片。
- `default_nr_free_pages`：返回当前系统的空闲页面总数。
- `basic_check`：对上述基本功能进行正确性验证。
- `default_check`：进行更全面的进阶功能检测。

上述功能被封装于 `default_pmm_manager` 结构体中，为内存管理提供了一个统一、可调用的接口。

改进空间

以下是几个可能的改进方向，旨在提升其性能、效率与灵活性：

1. 高效内存块合并策略

当前合并操作仅在释放内存时触发。可引入**预合并**与**延迟合并**机制，在分配前后主动扫描相邻块，或积累一定数量的释放操作后批量合并，以减少碎片化并降低频繁合并的开销。

2. 快速空闲块搜索算法

当前首次适应算法采用线性扫描，时间复杂度为 O(n)。可改用**平衡树结构，如红黑树、AVL树**或**分离空闲链表**，将不同大小的块分类管理，从而实现接近 O(log n) 的搜索效率，显著提升大容量内存下的分配速度。

3. 智能内存回收策略

引入**页面换出机制**，当物理内存紧张时，将不常用的页面换出到磁盘，并在需要时换入，从而提高内存利用率。可结合时钟算法或LRU近似算法，识别并换出"冷"页面，支持超量内存分配。

练习2

为了实现 Best-Fit 连续物理内存分配算法，参考kern/mm/default_pmm.c对First Fit算法的实现，我们在kern/mm/best_fit_pmm.c进行编程的补充，与First Fit算法相比，只有best_fit_alloc_pages函数补充部分需要修改：

```
    /*LAB2 EXERCISE 2: YOUR CODE*/
    // 下面的代码是first-fit的部分代码，请修改下面的代码改为best-fit
    // 遍历空闲链表，查找满足需求的空闲页框
    // 如果找到满足需求的页面，记录该页面以及当前找到的最小连续空闲页框数量
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_size) {
            page = p;
            min_size = p->property;
        }
    }
```

我们的设计思路如下就是在分配内存块时，按顺序查找空闲链表。当遇到第一个大于所需内存的空闲块时，先将其分配给目标页，然后继续向后查找。如果找到比已分配块更小的空闲块，则将当前页更新为该内存块。在释放内存块时，按顺序将其插入链表，并合并相邻的空闲块以形成更大的连续空间。

在具体实现中，我们使用框架提供的 `min_size` 来记录当前最小块的大小。在遍历过程中，每当发现比当前记录更小的空闲块，就更新 `min_size` 的值。这样，当循环结束时，我们就能确定最小块的位置，并将其对应的页指针存储在变量 `page` 中。

接着，我们运行wsl，运行`make qumu`对文件进行编译，控制台输出如下：

```
PS D:\os-riscv\labcode\lab2> wsl
wey@wey:/mnt/d/os-riscv/labcode/lab2$ make clean
rm -f -r obj bin
wey@wey:/mnt/d/os-riscv/labcode/lab2$ make qemu

+ cc kern/init/entry.S
+ cc kern/init/init.c
+ cc kern/libs/stdio.c
+ cc kern/debug/panic.c
+ cc kern/driver/console.c
+ cc kern/driver/dtb.c
+ cc kern/mm/best_fit_pmm.c
+ cc kern/mm/default_pmm.c
```

```
+ cc kern/mm/pmm.c
+ cc libs/printfmt.c
+ cc libs/readline.c
+ cc libs/sbi.c
+ cc libs/string.c
+ ld bin/kernel
  riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

OpenSBI v1.0


____                   _____ ____ _____

  / __ \                 / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : medeleg
Platform HART Count       : 1
Platform IPI Device       : aclint-mswi
Platform Timer Device     : aclint-mtimer @ 10000000Hz
Platform Console Device   : uart8250
Platform HSM Device       : ---
Platform Reboot Device    : sifive_test
Platform Shutdown Device  : sifive_test
Firmware Base             : 0x80000000
Firmware Size             : 252 KB
Runtime SBI Version       : 0.3

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000002000000-0x000000000200ffff (I)
Domain0 Region01          : 0x0000000080000000-0x0000000008003ffff ()
Domain0 Region02          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000000000000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsuh
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000001666
```

```
Boot HART MEDELEG          : 0x0000000000f0b509
```

然后，我们运行`make grade`进行测试：

控制台具体输出如下：

```
wey@wey:/mnt/d/os-riscv/labcode/lab2$ make grade
>>>>>>>>>> here_make>>>>>>>>>>>
gmake[1]: Entering directory '/mnt/d/os-riscv/labcode/lab2' + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/panic.c + cc kern/driver/console.c + cc kern/driver/dtb.c + cc kern/mm/best_fit_pmm.c + cc kern/mm/default_pmm.c + cc kern/mm/pmm.c + cc libs/printfmt.c + cc libs/readline.c + cc libs/sbi.c + cc libs/string.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img
gmake[1]: Leaving directory '/mnt/d/os-riscv/labcode/lab2'
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=11080
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
Total Score: 25/25
```

根据测试结果，本次实验成功通过了所有检查项，总得分为25/25。具体来看，物理内存映射信`physical_memory_map_information`和最佳适应分配算法`best_fit`两项测试均显示“OK”，表明系统能够正确管理物理内存空间，并且所实现的最佳适应算法在内存分配与释放过程中功能正常，能够有效查找满足需求的最小空闲块并进行合并操作，保证了内存管理的效率和可靠性。