OSlab2

组员：

2111451 朱景博
2110729 张弛
2111566 米奕霖

练习1：理解first-fit 连续物理内存分配算法（思考题）

> 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。请简要说明你的设计实现过程,并回答如下问题：*你的 first fit算法是否有进一步的改进空间?*

First Fit（首次适应） 算法是一种基本的内存分配策略，它会在空闲内存块列表中寻找第一个足够大以容纳请求的内存块，然后将该空闲区划分为两部分：一部分分配给请求的内存块，另一部分仍然是空闲的。

它的基本思想是从空闲内存块链表的头部开始查找，找到第一个满足大小条件的内存块就进行分配。从具体实现的角度出发，在***物理内存分配的过程*** 中，`default_init`初始化了内存管理器，`default_init_memmap`将物理内存映射到`struct Page`结构体中并插入`free_list`链表中。当 ***需要分配内存*** 时，`default_alloc_pages`在`free_list`中寻找符合条件的内存块，并分割之，返回分配的内存块的指针。***释放内存*** 时，`default_free_pages`将被释放的内存块标记为空闲，并尝试合并相邻的空闲内存块。***最后***，`basic_check`和`default_check`函数用于验证内存分配算法的正确性，确保分配和释放的行为符合预期。整个过程中，`free_list`链表记录了空闲内存块的信息，`nr_free`变量记录了剩余的空闲页面数。

上述提到的各相关函数在该分配过程中的具体作用阐述如下：

`default_init()`

该函数用于初始化物理内存管理器。它调用`list_init()`函数初始化了`free_list`，即空闲内存块的链表，同时将`nr_free`（空闲内存块的数量）设置为0。

`default_init_memmap(struct Page *base, size_t n)`

该函数用于初始化一片空闲内存块。它接受一个指向空闲内存块的起始页`base`和内存块的数量`n`作为参数。函数首先遍历这片内存块，将每一页的属性进行初始化，包括标记为`PG_reserved`（表示该页是已保留的）和`PageProperty`（表示该页是空闲的）。然后，将这片内存块添加到`free_list`链表中，同时更新`nr_free`。

`default_alloc_pages(size_t n)`

该函数用于分配连续的内存页。它接受一个参数`n`，表示需要分配的连续内存页数量。函数遍历`free_list`链表，找到第一个大小不小于`n`的空闲内存块，将该内存块分割出需要的页数，将剩余的内存块重新加入到`free_list`链表，并返回分配出来的内存块的指针。

`default_free_pages(struct Page *base, size_t n)`

该函数用于释放连续的内存页。它接受两个参数，`base`是待释放内存块的起始页，`n`是需要释放的内存页数量。函数首先遍历`free_list`链表，找到适当的位置将待释放的内存块插入到链表中，然后尝试合并相邻的空闲内存块。合并的操作会更新相邻内存块的`property`字段，将它们的页数合并为一个大的内存块。最后，函数会将释放的页数加入到`nr_free`中。

`basic_check(void)` **&&** `default_check(void)`:

这两个函数用于检查内存分配和释放的基本正确性，包括分配、释放、空闲内存块管理等方面的测试。这些函数在内存分配的过程中起到了验证内存管理算法正确性的作用。

分配流程概述

1. **初始化阶段**：在系统启动时，通过`default_init`初始化物理内存管理器，将`free_list`和`nr_free`置零。
2. **内存映射阶段**：通过`default_init_memmap`初始化一片空闲内存块，并将其加入`free_list`。在这个阶段，系统获得了一些初始化的内存。
3. **分配阶段**：当系统需要分配内存时，调用`default_alloc_pages`函数，它遍历`free_list`，找到第一个大小不小于需求的内存块，分割出所需内存页，返回分配好的内存块的指针。
4. **释放阶段**：当系统不再需要使用某些内存时，调用`default_free_pages`函数，它将释放的内存块按顺序插入`free_list`，并尝试合并相邻的空闲内存块。
5. **内存管理器状态**：随着分配和释放的进行，`free_list`中的空闲内存块会动态变化，`nr_free`会反映系统当前的可用内存页数。

总体来说，这些函数组成了一个基于 *first-fit 分配算法* 的内存分配器，通过链表管理内存块的分配和释放。在分配时，它会在空闲块中找到第一个足够大的内存块进行分配。在释放时，它会尝试合并相邻的空闲内存块，以提高内存利用效率。

改进方向

不难看出，*first-fit 算法* 是一种简单而非最优的内存分配算法，通常用于一些小型系统或者作为其他高级分配算法的基础。而且就上述给定的算法实现而言，仍存在一些可以改进的地方，具体阐述如下：

- **内存合并策略优化：** 当释放内存时，代码中的合并策略是尝试与前后相邻的内存块合并。可以考虑使用更智能的合并策略，例如合并时选择与大小相近的内存块合并，以减少外部碎片。
- **分割策略改进：** 当分割内存块时，可以考虑将剩余内存块加入到一个“伙伴系统”中。伙伴系统是一种内存分割策略，它将剩余的内存块按照2的幂次划分，然后根据需要分配不同大小的块。这样可以减少内部碎片。
- **空闲内存块的排序：** 当空闲内存块插入和查找时，可以考虑使用更高效的数据结构或者排序算法，以提高查找效率。例如，使用平衡二叉树或者红黑树来维护空闲内存块，使得插入和查找的时间复杂度降低。
- **内存回收策略：** 当释放内存块时，可以考虑实现一些内存回收策略，例如延迟合并（等到内存使用率较低时再进行合并），以及基于内存使用情况的动态调整策略。
- **支持更多的页表尺寸：** 考虑到不同应用场景可能需要不同大小的内存块，可以尝试支持更多的页表尺寸，以满足不同粒度的内存分配需求。
- **错误处理和鲁棒性增强：** 在代码中加入错误处理机制，使得程序在面对异常情况时能够更好地进行处理，提高系统的鲁棒性。

改进以上方面可以使该算法更加高效、健壮和灵活地应对各种内存分配需求。这些改进不仅可以提高内存分配的性能，还可以减少内存碎片，提高系统整体的内存利用率。

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?*

Best-Fit 算法 同样是一种内存分配算法，旨在找到一个尺寸最符合要求（最小且能容纳所需内存大小）的空闲块来分配内存。与 First-Fit 算法不同，Best-Fit 算法会遍历所有的空闲块，选择最小且足够大的块来满足内存分配请求。因此它能够尽量减少内存碎片，但与此同时也导致了较大的性能开销（需要遍历整个空闲块链表来找到最合适的块）。

在设计实现 *best-fit 算法* 时，各个函数模块的包装分工本质上并无太大变化，具体阐述如下：

- **初始化（best_fit_init）：** 初始化 free_list（空闲块链表）和 nr_free（空闲块数量计数器）。
- **初始化内存映射（best_fit_init_memmap）：** 该函数用于初始化一个空闲内存块。它遍历指定数量的页面，初始化它们的标志和属性，然后将它们添加到空闲块链表中。在添加页面到链表时，它会根据地址顺序插入，以确保链表是按照地址从低到高排列的。
- **分配内存块（best_fit_alloc_pages）：** 这个函数用于分配内存。它遍历空闲块链表，查找符合请求大小的最小的空闲块（即 Best-Fit 的逻辑）。如果找到了合适的块，就将它分配出去。如果找到的块比请求的内存大，会将剩余的部分作为新的空闲块加入到链表中。
- **释放内存块（best_fit_free_pages）：** 该函数用于释放内存。它首先根据传入的页面和数量，将对应的页面标记为已分配状态。然后，它会尝试合并相邻的空闲块，以减少碎片。
- **空闲块数量统计（best_fit_nr_free_pages）：** 这个函数用于返回当前系统中的空闲块数量。
- **检查函数（best_fit_check）：** 该函数用于检查 Best-Fit 算法的正确性。它会进行一系列的分配和释放操作，然后检查分配和释放的内存块是否符合预期。

就 ***填充实现的流程*** 而言，涉及的步骤如下：

- 修改`pmm.c`文件，首先本练习的本质要求就是用 *best-fit* 算法替代给定的 *first-fit算法* 并 `make qemu` 成功，故首先将`pmm.c/init_pmm_manager`函数修改如下：

  ```
  static void init_pmm_manager(void) {
      pmm_manager = &best_fit_pmm_manager;
      //pmm_manager = &default_pmm_manager; //First-Fit algorithm
      cprintf("memory management: %s\n", pmm_manager->name);
      pmm_manager->init();
  }
  ```

  即将内存管理器设置为 *best-fit* 算法的管理器，其余任何配置包括`init.c`都无修改，则可将默认的 *first-fit* 算法替换为 *best-fit* 算法来进行内存的分配和释放。

- 填充完善`best_fit_pmm.c`，完成上述整体算法的替换后即可着手 *best-fit* 具体细节的填充实现，具体阐述如下：

  - ***best_fit_init_memmap***，该函数的是为了将一块内存区域分割成页框（Page）并将它们加入到空闲内存块链表中，这个过程对于两种内存分配算法都是必要且一致的，故在填充时只需将 *first-fit* 中的实现在理解的基础上照搬即可，具体实现如下：

    ```
    static void
    best_fit_init_memmap(struct Page *base, size_t n) {
      assert(n > 0);
      struct Page *p = base;
      for (; p != base + n; p ++) {
    ```

```
        assert(PageReserved(p));
        /*LAB2 EXERCISE 2: YOUR CODE*/
        // 清空当前页框的标志和属性信息，并将页框的引用计数设置为0
        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);
             /*LAB2 EXERCISE 2: YOUR CODE*/
            // 编写代码
            // 1、当base < page时，找到第一个大于base的页，将base插入到它前面，并退出循环
            // 2、当list_next(le) == &free_list时，若已经到达链表结尾，将base插入到链表尾部
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(p->page_link));
            }
        }
    }
}
```

对当前页框的标志和属性信息以及引用计数的清零不必多言；将当前内存块按地址升序排列插入到`free_list`中时，也只是简单的双向链表值的判断与插入，仅需要注意实现时需要用对给定的函数接口。

- ***best_fit_alloc_pages***，该函数实现的是内存分配操作，此处的逻辑不同于 *first-fit* 的找到即分配，而是找到最合适的进行分配（满足分配的最小内存块），故在寻找 page时的循环中将 if 分支的判断条件改为`if (p->property >= n && p->property < min_size)`，并且维护了`size_t min_size`，将其初始化为`nr_free + 1`，且每次找到更合适的 page时进行更新，具体代码如下：

```
static struct Page *
best_fit_alloc_pages(size_t n) {
  assert(n > 0);
  if (n > nr_free) {
      return NULL;
  }
  struct Page *page = NULL;
  list_entry_t *le = &free_list;
  size_t min_size = nr_free + 1;
   /*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) {
            min_size = p->property;
            page = p;
        }
    }

    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;
}
```

- ***best_fit_free_pages***，该函数的作用是释放与合并内存块，此处的操作逻辑又与 *first-fit* 算法没有区别，故同样照搬即可，具体代码如下：

```
static void
best_fit_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);
    }
    /*LAB2 EXERCISE 2: YOUR CODE*/
    // 编写代码
    // 具体来说就是设置当前页块的属性为释放的页块数、并将当前页块标记为已分配状态、最后增加nr_free的值
    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));
            }
        }
```

```
    }

    // 尝试合并低地址相邻的空闲页块
    list_entry_t* le = list_prev(&(base->page_link));
    if (le != &free_list) {
        //p = le2page(le, page_link);
        /*LAB2 EXERCISE 2: YOUR CODE*/
         // 编写代码
        // 1、判断前面的空闲页块是否与当前页块是连续的，如果是连续的，则将当前页块合并到前面的空闲页块中
        // 2、首先更新前一个空闲页块的大小，加上当前页块的大小
        // 3、清除当前页块的属性标记，表示不再是空闲页块
        // 4、从链表中删除当前页块
        // 5、将指针指向前一个空闲页块，以便继续检查合并后的连续空闲页块
        p = le2page(le, page_link);
        if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            list_del(&(base->page_link));
            base = p;
        }
    }

    // 尝试合并高地址相邻的空闲页块
    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));
        }
    }
}
```

具体而言，两种操作的实现过程简述如下：

1. 释放内存块

- 遍历内存块中的每一页，确保这些页既不是保留页也不是属性页。
- 清空当前页的属性标志和引用计数，将其标记为已分配状态。
- 设置当前页块的属性为释放的页块数（`base->property = n;`），表示这个内存块的大小。
- 设置页块的属性标志，表示这是一个空闲页块（`SetPageProperty(base);`）。
- 将释放的页块加入空闲页块链表中，确保链表按地址从低到高排列。

2. 尝试合并相邻的空闲页块：

- 首先，尝试合并低地址相邻的空闲页块。
    - 检查前一个空闲页块是否与当前页块是连续的。如果是连续的，将当前页块合并到前一个空闲页块中。更新前一个空闲页块的大小，清除当前页块的属性标记，从链表中删除当前页块，将指针指向前一个空闲页块，以便继续检查合并后的连续空闲页块。
- 接着，尝试合并高地址相邻的空闲页块。

- 检查后一个空闲页块是否与当前页块是连续的。如果是连续的，将当前页块合并到后一个空闲页块中。更新当前页块的大小，清除后一个空闲页块的属性标记，从链表中删除后一个空闲页块。

测试验证

完成上述代码填充后，经过反复的测试验证，最终可以在 lab2 根目录下 `make qemu` 成功，且输出中的 *entry*、*etext*、*edata*、*end*、*physcial memory map* 等信息也与设想中的一致，截图如下：

```
maledingda53@maledingda53-virtual-machine:~/OSlab/riscv64-ucore-labcodes/lab2$ make qemu

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x0000000080001fff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200036 (virtual)
  etext  0xffffffffc0201b60 (virtual)
  edata  0xffffffffc0206010 (virtual)
  end    0xffffffffc0206470 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
```

退出后再进行 `make grade` 的结果如下：

```
maledingda53@maledingda53-virtual-machine:~/OSlab/riscv64-ucore-labcodes/lab2$ make grade
>>>>>>>>>> here_make>>>>>>>>>>>
gmake[1]: Entering directory '/home/maledingda53/OSlab/riscv64-ucore-labcodes/lab2' + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/kdebug.c + cc
kern/debug/kmonitor.c + cc kern/debug/panic.c + cc kern/driver/clock.c + cc kern/driver/console.c + cc kern/driver/intr.c + cc kern/trap/trap.c + cc kern/trap/trapentry.S + cc kern/m
m/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-objcop
y bin/kernel --strip-all -O binary bin/ucore.img gmake[1]: Leaving directory '/home/maledingda53/OSlab/riscv64-ucore-labcodes/lab2'
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=55611
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
  -check_ticks:                              OK
Total Score: 30/30
```

经验证，上述实现的 *best-fit* 算法通过了测试。

改进方向

上述基于 *first-fit* 架构实现的 *best-fit* 算法由于没有考虑时间空间的开销，仅以通过测试为目标，故仍存在许多可供改进之处：

1. **循环优化：** 在 `best_fit_alloc_pages` 函数中，遍历空闲链表查找合适的空闲页框时，可以考虑使用更高效的查找算法，例如二分查找。因为Best-Fit要求找到最小但足够的空闲页框，如果空闲链表按照大小有序排列，可以通过二分查找快速定位合适的位置。
2. **合并策略优化：** 在释放内存块的时候，可以考虑使用更智能的合并策略。例如，可以实现一种策略，在分配的时候就合并相邻的空闲块，避免在释放的时候进行多次合并，从而提高分配和释放的效率。
3. **分配算法改进：** 在Best-Fit算法中，当找到满足需求的页面时，会将该页面分割为两部分，其中一部分可能很小，无法再继续被分配。可以考虑在分割时，如果剩余部分过小，直接放弃分割，将整个页面分配给请求。
4. **空闲页块的组织结构：** 考虑使用更高效的数据结构来组织空闲页块链表，以加速插入和删除操作。例如，可以使用平衡二叉树、跳表等数据结构，以提高查找和插入的效率。

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

> 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）

> 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?

在大多数操作系统中，获取可用物理内存范围通常需要直接与硬件进行交互，因此这些操作通常需要在操作系统内核级别或者特权级别进行。以下是一种在x86架构下的通用方法，用于获取可用物理内存范围：

在x86架构下，BIOS（基本输入/输出系统）提供了ACPI（高级配置和电源管理接口）表，其中包含了系统的硬件配置信息，包括内存信息。操作系统可以通过访问这些表来获取可用物理内存范围。

以下是一个获取可用物理内存范围的基本步骤：

1. **访问ACPI表：** 操作系统在启动时通常会从BIOS中读取ACPI表。ACPI表包含了系统硬件的详细信息，包括内存配置。
2. **解析ACPI表：** 操作系统需要解析ACPI表的内容，特别是RSDP（Root System Description Pointer）和RSDT（Root System Description Table）这两个表。RSDT包含了指向其他ACPI表的指针，其中可能包含了内存配置信息。
3. **查找内存描述符：** 在ACPI表中，有可能存在描述系统内存配置的数据结构，通常被称为内存描述符。这个描述符包含了可用物理内存的起始地址和结束地址。
4. **处理内存范围：** 一旦找到了内存描述符，操作系统可以将这些信息保存起来，以便后续的内存管理。通常，这些信息会被用于建立页表，设置内存分配策略等。

当然，上述方法是在特权级别较高的情况下运行的，需要操作系统内核的支持和相应的硬件访问权限。如果在用户空间下运行程序，则无法依靠上述方法直接获取到这些信息，故该方法也仅作理论参考，具体实施还要因操作系统而异。