lab2

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?

- **程序在进行物理内存分配的过程：**

首先初始化，利用default_init函数初始化一个全局变量`nr_free`来记录当前可用的物理页面数，并初始化一个空闲页面链表`free_list`。

然后进行内存页面的初始化，利用`default_init_memmap`函数对物理内存进行初始化。该函数接收一个页面的起始地址和页面数量，确保这些页面未被保留，并将它们的状态（如标志和引用计数）重置为初始值。将起始页面的属性设置为初始化的页面数量，并更新全局可用页面计数`nr_free`。最后将这些页面添加到空闲页面链表中。

继续则进行内存分配，通过`default_alloc_pages`函数分配指定数量的物理页面。该函数首先检查请求的页面数量是否超过可用页面数量`nr_free`，如果不足则返回 NULL。遍历空闲页面链表，找到第一个具有足够数量页面的空闲页面。如果找到了合适的页面，则更新该页面的属性，调整链表，并更新`nr_free`的值。

最后进行内存释放，通过`default_free_pages`函数释放指定数量的页面。该函数会遍历要释放的页面，重置它们的状态，并将它们的属性设置为可用。将释放的页面添加到空闲链表中。如果释放的页面与相邻的空闲页面相邻，则合并这些页面，更新它们的属性，减少内存碎片。

- **函数作用：**

```
static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;//nr_free可以理解为在这里可以使用的一个全局变量，记录可用的物理页面数
}
```

`default_init`函数的作用是初始化物理内存管理器的内部数据结构。在函数内初始化了空闲链表，且初始化了当前可用页面数nr_free。

```
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));
            }
        }
    }
}
```

`default_init_memmap`函数的作用是初始化一段物理内存页面，并将其添加到内存管理的空闲页面链表中。

首先在函数中需要先进行检查，确保要初始化的页面数量n大于0（如果小于等于0，说明没必须初始化，是无效的）。

然后循环遍历从base开始的n个页面，对于每个页面，利用PageReserved(p)确保页面是预留状态，防止对已分配的页面进行错误的操作。然后将页面的标志flags和属性初始化为0。并且将页面的引用计数设置为0，表示没有任何进程或模块引用此页面。

将基础页base的属性设置为n，表示这一块内存包含n个连续的物理页面。然后增加n个nr_free，以表示空闲页面数的增加。

然后就可以将新页面添加到空闲页面链表，如果空闲链表为空，直接将基础页面添加到链表中。如果不为空，则遍历找到合适位置（确保是按地址顺序排序）。

```
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;
}
```

default_alloc_pages函数作用是分配指定数量的物理页面，遍历空闲页面链表找到合适的页面进行分配，必要时将页面拆分，并更新相关的状态和统计信息。首先需要检查请求的页面数量是否大于0。而且也需要检查可用的空闲页面数是否足够，如果不够就返回NULL。

然后开始遍历空闲页面链表，使用 `le2page(le, page_link)` 将其转换为 `Page` 结构体指针 `p`，p->property表示可用页面的数量，然后看是否大于请求数量n，如果大于，则说明找到可分配的页面，将其指针赋值给 `page`，并使用 `break` 跳出循环。

找到可以分配的页面后，首先用prev存储当前页面在空闲链表中的前一个条目的指针，然后再将当前页面从空闲页面链表中删除，表示该页面已被分配，不可再用。需要查看当前页面的可用数量是不是大于请求的页面数量，如果大于，进行拆分。创建一个指针 `p`，指向当前页面的后 `n` 个位置，表示剩余可用的页面。将 `p` 的 `property` 属性设置为当前页面的 `property` 减去请求的页面数量 `n`，表示剩余的可用页面数量。使用 `SetPageProperty(p)` 函数将 `p` 标记为一个可用页面。然后将剩余页面 `p` 的条目添加回链表。这样，剩余的可用页面就可以在后续的分配请求中再次被使用。

同时需要更新一下nr_free，减去分配的n个页面数量。调用 `ClearPageProperty(page)` 函数，清除已分配页面的属性，表明该页面现在处于已分配状态，而不是空闲状态。

```
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));
        }
    }
}
```

default_free_pages函数作用是释放指定数量的物理页面并将它们重新添加到空闲页面链表中，更新相应的属性和状态。

在该函数中指针p指向要释放的页面的起始位置base。然后循环遍历从base开始的n个页面，在每次循环中，使用 `assert` 确保页面**未被保留（`!PageReserved(p)`）且没有其他属性标志（`!PageProperty(p)`）**。

将起始页面 `base` 的 `property` 设置为释放的页面数量 `n`，表示这 `n` 个页面现在可以重新使用。将可用页面数 `nr_free` 增加 `n`，以反映新释放的页面。

同时尝试合并相邻的空闲页面，以减少空闲页面的碎片化，提高内存利用率。

- **改进空间**：目前使用的是双向链表来存放空闲的页表，考虑使用其他数据结构，如跳表等来提高内存的分配。

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?

best-fit算法核心思想是即要满足要求（即p->property >= n）且还需要空闲部分是最小的。主要修改部分在分配环节：

```
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) {
        if(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;
}
```

在size_t min_size = nr_free + 1;中定义了一个min_size，然后在循环中先找到大于参数n的空闲块，如果其小于min_size，则将其值赋给min_size，遍历完所有大于n的空闲块，最终得到的大小大于n但是又是其中最小的那个空闲块。

测试验证代码正确性：

```
xiekejie@xiekejie-virtual-machine:~/桌面/NKU_OS/riscv64-ucore-labcodes/lab2$ mak
e grade
>>>>>>>>>> here_make>>>>>>>>>>
gmake[1]: 进入目录“/home/xiekejie/桌面/NKU_OS/riscv64-ucore-labcodes/lab2” + cc
kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/k
debug.c + cc kern/debug/kmonitor.c + cc kern/debug/panic.c + cc kern/driver/cloc
k.c + cc kern/driver/console.c + cc kern/driver/intr.c + cc kern/trap/trap.c + c
c kern/trap/trapentry.S + 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-al
l -O binary bin/ucore.img gmake[1]: 离开目录“/home/xiekejie/桌面/NKU_OS/riscv64-
ucore-labcodes/lab2”
>>>>>>>>>> here_make>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=4723
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
  -check ticks:                              OK
Total Score: 30/30
```

- **改进空间：**在代码中，对比first-fit相当于要循环遍去找到那个best的空闲块，如果在排序放置的时候不按照地址而是按照大小，这个时候只要找到的第一个大于参数n的空闲块就是找到的best的了。这样可以更快的搜索到适合大小的空闲块，减少时间开销。

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?

可用物理内存范围：系统中可以被操作系统和应用程序使用的实际物理内存量。

当操作系统无法提前知道当前硬件的可用物理内存范围时，一般采取下列方法：

1.BIOS/UEFI：

大多数现代计算机使用BIOS或UEFI来初始化硬件。在系统启动时，BIOS或UEFI会执行硬件检测并提供内存信息。OS可以通过调用BIOS/UEFI提供的中断服务或接口来获取可用物理内存的范围。具体步骤如下：

- **开机自检（POST）**：BIOS/UEFI在开机时执行自检，记录系统中检测到的内存区域。
- **ACPI（高级配置与电源接口）**：OS可以通过ACPI表（如`Memory Map`）获取内存布局信息。
- **读取内存映射**: 通过ACPI提供的接口，OS可以读取所有物理内存区域的起始地址和大小，并确定可用的内存范围。

2.操作系统内核的内存扫描：

如果OS没有在启动时获取内存信息，或需要动态检测可用内存，可以通过以下方式进行内存扫描：

- **使用特定的硬件指令**: 在某些架构上，OS可以使用特殊的指令（如x86架构上的`CPUID`指令）来查询物理内存的特性。
- **遍历内存地址**: OS可以尝试访问各个物理内存地址并监测访问异常，记录成功访问的内存区域。需要注意，这种方法可能会影响系统性能，并且有风险。

3.使用系统API：
通过调用`GlobalMemoryStatusEx`函数和解析`MEMORYSTATUSEX`结构体，可以获取到所需的内存信息。