entry.S中的kern\_entry函数。kern\_entry函数的主要任务是设置虚拟内存管理，将三级页表的物理地址和Sv39模式位写入satp寄存器，以建立内核的虚拟内存空间，为之后建立分页机制的过程做一个准备。完成这些工作后，才调用kern\_init函数。

要进入虚拟内存访问方式，需要如下步骤：

1.分配页表所在内存空间并初始化页表；

2.设置好页基址寄存器（指向页表起始地址）；

3.刷新 TLB。

一个页表项占据8字节（64位），而虚拟地址有39位，后12位是页内偏移。

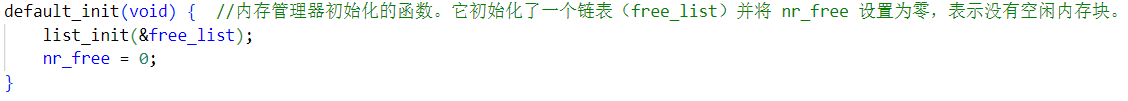
使用三级页表。有4KiB=4096字节的页，大小为2MiB= 2^21 字节的大页，和大小为1 GiB 的大大页。

原先的一个39位虚拟地址，被我们看成27位的页号和12位的页内偏移。那么在三级页表下，我们可以把它看成9位的“大大页页号”，9位的“大页页号”（也是大大页内的页内偏移），9位的“页号”（大页的页内偏移），还有12位的页内偏移。这是一个递归的过程，中间的每一级页表映射是类似的。也就是说，整个Sv39的虚拟内存空间里，有512（2的9次方）个大大页，每个大大页里有512个大页，每个大页里有512个页，每个页里有4096个字节，整个虚拟内存空间里就有512∗512∗512∗4096个字节，是512GiB的地址空间。

4096/8 = 512，我们恰好可以在一页里放下512个页表项.

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？

First Fit 在处理碎片问题时，可能存在一些改进的空间。以下是一些可能的改进方向：

1. Best Fit 和 Worst Fit：First Fit 算法通常会选择第一个足够大的内存块来满足请求，这可能导致产生许多小碎片。Best Fit 算法会选择最适合请求大小的内存块，而 Worst Fit 算法会选择最差的内存块。这些算法可能更好地管理碎片，但它们通常需要更多的搜索操作，因此可能会导致性能损失。

2. Compaction：碎片问题是连续内存分配的一个常见挑战。一种改进方法是定期进行内存整理（或称为内存压缩或碎片整理），以将碎片汇总到一起，从而形成更大的可用内存块。这需要在系统不活跃时执行，但可以减少碎片问题。

3. 分区内存管理：将内存分为多个固定大小的分区，每个分区用于存储特定大小的内存块，可以更好地管理内存。这是内存池技术的一种形式，可以避免外部碎片问题。

4. 非连续内存分配：一些操作系统采用非连续内存分配，例如页表分页或段式内存管理，这种方式可以更好地管理内存碎片，但也引入了一些复杂性。

5. 动态调整算法：使用动态算法来选择最佳内存分配策略，可以根据实际内存使用情况和请求来选择不同的策略。这需要更复杂的算法，但可以更好地适应不同的工作负载。

6. First Fit 改进：如果仍然希望使用 First Fit，可以考虑改进版本，例如 Next Fit（从上次分配的位置继续搜索）或 Quick Fit（维护多个空闲链表，以便更快地找到合适的内存块）。

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？

实验代码

static void

best\_fit\_init\_memmap(struct Page \*base, size\_t n) {  //清空内存页的标志和属性信息，并将引用计数设置为 0：

    assert(n > 0);

    struct Page \*p = base;

    for (; p != base + n; p ++) {

        assert(PageReserved(p));

        /\*LAB2 EXERCISE 2: YOUR CODE 2111642\*/

        // 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 2111642\*/

            // 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, &(base->page\_link));

            }

        }

    }

}

static struct Page \*

best\_fit\_alloc\_pages(size\_t n) {  //实现 Best Fit 算法，即查找满足需求的最适合的内存块：

    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;    //满足需求的最小内存块大小

     /\*LAB2 EXERCISE 2: YOUR CODE 2111642\*/

    // 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

    //first-fit:

    /\* while ((le = list\_next(le)) != &free\_list) {

        struct Page \*p = le2page(le, page\_link);

        if (p->property >= n) {

            page = p;

            break;

        }

    } \*/

    //best-fit:

    // 遍历空闲链表，查找满足需求的空闲页框

    while ((le = list\_next(le)) != &free\_list) {

        struct Page \*p = le2page(le, page\_link);

        // 如果 p 的空闲块大小能满足需求且小于 min\_size，则更新 page

        if (p->property >= n && p->property < min\_size) {

            page = p;

            min\_size = p->property;   //更新min\_size

        }

    }

    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;

}

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 2111642\*/

    // 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

    // 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;  //设置释放的第一个内存页的 property 属性为释放的页数 n，以表示该内存块包含了多少个空闲页。

    SetPageProperty(base); //将第一个内存页标记为已分配状态，即设置 PG\_property 位为 1，表示它属于一个连续的内存块中的一部分

    nr\_free += n; //增加系统中可用内存页的数量

    // 插入空闲链表

    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);

            // 正确插入 base 到空闲链表合适位置中

            if (base < page) {

                list\_add\_before(le, &(base->page\_link));

                break;

            } else if (list\_next(le) == &free\_list) {

                list\_add(le, &(base->page\_link));

            }

        }

    }

    // 检查前后的内存块是否可以合并

    list\_entry\_t\* le = list\_prev(&(base->page\_link));

    if (le != &free\_list) {

        p = le2page(le, page\_link);

        /\*LAB2 EXERCISE 2: YOUR CODE 2111642\*/

         // 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

        // 编写代码来合并前面的连续空闲页块

        if (p + p->property == base) {

            p->property += base->property;

            ClearPageProperty(base);  //将当前内存页的 PG\_property 标志位清零，表示当前内存页不再属于一个连续内存块中的一部分。不可分配

            list\_del(&(base->page\_link));   //从链表中删除当前内存页，因为它已经不再是一个独立的内存块。

            base = p;

        }

    }

    // 编写代码来合并后面的连续空闲页块

    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));

        }

    }

}

结果验证：



