Lab2 物理内存和页表

小组成员：张高2213219 张铭2211289 黄贝杰2210873

实验一解决了“启动”的问题，实验二就要开始解决操作系统的物理内存管理的问题。 我们需要理解页表的建立和使用方法、理解物理内存的管理方法、理解页面分配算法。

练习1：理解first-fit 连续物理内存分配算法

first-fit 连续物理内存分配算法是基础的方法，需要理解它的实现过程。仔细阅读kern/mm/default_pmm.c 相关代码，认分析 default_init 、default_init_memmap 、default_alloc_pages 、default_free_pages 中相关函数，描述程序在物理内存分配过程及各个函数的作用。简要说明设计实现过程。

first-fit 算法概述

- First-Fit算法是一种经典的内存管理算法，主要用于动态分区分配。
- 当一个进程请求内存时，`First-Fit` 算法会从空闲内存块列表中寻找第一个足够大的空闲块，并将其分配给请求。
- 如果找到的空闲块大于请求的大小，则可以将多余的内存分割出来，形成一个新的空闲块。 **优点**：该算法倾向于使用内存中低地址部分的空闲区，在高地址部分的空闲区很少被利用，从而保留了高地址部分的大空闲区。显然为以后到达的大作业分配大的内存空间创造了条件。 **缺点**：低地址部分不断被划分，留下许多难以利用、很小的空闲区，而每次查找又都从低地址部分开始，会增加查找的开销。

first-fit 算法步骤

初始化空闲列表:

- 初始化一个双向链表来记录所有空闲的物理内存块,
- 每个空闲块包含物理地址、大小、引用计数、标志位等信息。 **分配内存**:
- 当一个进程请求内存时，从空闲列表的头部开始遍历，寻找第一个足够大的空闲块,
- 如果找到合适的空闲块，则分配该块，并更新空闲列表,
- 如果没有找到合适的空闲块，则返回 `NULL` 或者抛出异常。 **释放内存**:
- 当一个进程释放内存时，将释放的内存块重新加入到空闲列表中，并尝试与其他相邻的空闲块合并以减少碎片。

first-fit 算法优点

该算法倾向于使用内存中低地址部分的空闲区，在高地址部分的空闲区很少被利用，从而保留了高地址部分的大空闲区。显然为以后到达的大作业分配大的内存空间创造了条件。

first-fit 算法缺点

低地址部分不断被划分，留下许多难以利用、很小的空闲区，而每次查找又都从低地址部分开始，会增加查找的开销。

相关函数的作用

1. `default_init()`

- **作用**：该函数初始化内存分配器，将链表`free_list`初始化为空并将空闲页数量`nr_free`设为0。
- **细节**：
  - `list_init(&free_list)`：初始化空闲页链表，使其成为一个空链表。
  - `nr_free = 0`：初始化空闲物理页数量为0。

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

- **作用**：初始化从`base`开始的`n`个物理页，并将其标记为空闲页。随后，将这片连续的空闲页加入空闲链表中。
- **细节**：
  - 该函数首先检查`n`是否大于0，并遍历`base`到`base + n`的所有页面，确保它们为保留页面（通过`PageReserved(p)`）。
  - 对每个页面，清空标志位和属性，并将引用计数置为0。
  - 为起始页`base`设置属性`property`，表示这片内存块的大小为`n`，并通过`SetPageProperty(base)`将其标记为拥有属性。
  - 将这块内存插入`free_list`中，保持链表的有序性（按地址从小到大排序）。遍历链表，找到插入的位置，插入空闲块。

3. `default_alloc_pages(size_t n)`

- **作用**：分配`n`个连续的物理页，基于First-Fit策略选择合适的空闲块。
- **细节**：
  - 先判断`n`是否大于空闲页总数`nr_free`，如果不足则返回`NULL`。
  - 遍历空闲页链表`free_list`，找到第一个满足`p->property >= n`的物理页块，返回该块。
  - 如果找到的物理页块大于需要的大小`n`，将剩余的部分拆分出来，并继续保留在空闲链表中。
  - 更新空闲页数量`nr_free`，将分配的物理页从链表中删除，并返回起始的`page`指针。

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

- **作用**：释放从`base`开始的`n`个连续物理页，并尝试合并相邻的空闲页块。
- **细节**：
  - 遍历`base`到`base + n`的所有物理页，检查是否为有效页，并重置标志位和引用计数。
  - 将这些物理页重新插入空闲链表中，同时保持链表的有序性。
  - 通过检查前后相邻的空闲页块，进行合并操作。如果前一个页块的结束位置与当前`base`页相连，或者当前页块的结束位置与后一个页块相连，就将这些块合并成一个更大的连续空闲块。

5. `default_nr_free_pages()`

- **作用**：返回当前系统中空闲的物理页总数。
- **细节**：
  - 直接返回全局变量`nr_free`，表示系统中未分配的页数量。

6. `default_check()`

- **作用**：用于检查物理内存分配算法的正确性。
- **细节**：
  - 通过一系列的分配和释放操作，验证分配器是否能正确地执行First-Fit算法。
  - 包含对边界情况的测试，比如分配多于空闲页、内存块合并等。

物理内存分配过程：

1. **初始化阶段**：
    - 使用`default_init()`函数初始化链表，空闲页数为0。使用`default_init_memmap(base, n)`初始化指定范围的物理页，并将这块连续的物理页加入空闲链表`free_list`。
2. **内存分配阶段**：
    - 当需要分配`n`个连续页时，调用`default_alloc_pages(n)`函数。该函数遍历空闲页链表，找到第一个大小大于或等于`n`的物理页块，并返回其起始地址。如果找到的块大于`n`，则进行拆分，保留多余的空闲页。
3. **内存释放阶段**：
    - 当释放`n`个连续页时，调用`default_free_pages(base, n)`函数。该函数将指定的物理页重新插入空闲链表，并检查相邻的页块是否可以合并成更大的块，以提高内存利用率。

总的来说，该代码实现了First-Fit连续物理内存分配算法，分配时会选择第一个足够大的空闲块，并尽可能避免碎片化。当内存释放时，通过合并相邻的空闲块来进一步减少碎片化。

改进和优化：

1. **碎片化问题**
    - **问题**：First-Fit算法会导致外部碎片化。即使系统中有足够的空闲内存，如果这些空闲块分布在不同的位置且没有足够大的连续块，可能无法分配需要的内存。
    - **改进建议**：引入**内存合并与紧凑化**的机制，定期合并和重新组织碎片化的内存块，增加连续内存块的可用性。这可以通过后台运行的“内存整理”线程实现，或者在空闲时主动执行合并操作。
2. **搜索效率**
    - **问题**：每次分配时，First-Fit算法从链表头开始顺序遍历，直到找到第一个合适的空闲块。随着空闲块数量的增加，这种遍历可能会降低性能，尤其是在空闲块较多时。
    - **改进建议**：
        1. **分区空闲链表**：将不同大小的空闲块分为不同的链表进行管理，类似于**分区分配**或**分级链表**。例如，按内存块大小划分多个链表（小于4KB、4KB到16KB、16KB到64KB等），分配时只需查找合适大小范围的链表，可以减少遍历时间。
        2. **二分搜索树**：可以使用**平衡二叉搜索树**（如红黑树）或者其他高效的数据结构来管理空闲块，使得每次查找合适的空闲块时可以通过二分法等手段快速定位。
3. **分配策略优化**
    - **问题**：First-Fit策略可能总是选择第一个符合条件的块，导致链表前端的块被频繁分配和回收，链表后端的块则较少使用，产生潜在的效率问题。
    - **改进建议**：
        1. **Best-Fit算法**：分配时选择刚好满足需求且最小的空闲块，减少大块的浪费。这样可以最大限度地减少剩余空间，降低碎片化的几率。然而，Best-Fit需要遍历所有空闲块，增加查找时间，适合碎片问题较为严重的场景。
        2. **Next-Fit算法**：在First-Fit的基础上，使用“循环查找”机制，记录上次分配的空闲块位置，下一次分配从该位置开始搜索。这可以避免每次都从链表头开始，均匀分布内存块的分配，减少局部性问题。

3. **Worst-Fit算法**：选择最大的空闲块进行分配，避免过度切割小块，减少小块的产生，适合碎片化非常严重的情况。

4. **内存块拆分与合并的改进**

   - **问题**：当前实现中的内存块拆分和合并操作可能较为复杂，尤其是链表的插入操作需要遍历和寻找合适的位置，影响效率。
   - **改进建议**：
     1. **Buddy System（伙伴系统）**：使用**伙伴系统**进行管理，将内存块按照2的幂次方大小进行分割。内存释放时，两个相邻且大小相等的块可以合并成更大的块。伙伴系统的优点是合并和分配操作更加高效，合并时无需遍历链表，只需检查相邻块即可。
     2. **懒惰合并**：在内存释放时，不立即执行合并操作，而是等到分配需要时再进行。这种策略可以减少不必要的合并操作，提高系统性能。

5. **内存回收的改进**

   - **问题**：当前的内存释放操作每次都需要检查前后相邻的块进行合并。这可能导致性能下降，尤其是在频繁的内存释放场景下。
   - **改进建议**：
     1. 可以采用**延迟合并**机制，内存释放时不立即检查和合并，而是将合并操作延迟到空闲块即将被分配时，减少不必要的操作。
     2. **批量释放与回收**：对于频繁分配和释放的小块内存，可以引入批量回收的策略，一次性回收多个小块内存，并合并为大块，提高效率。

练习2：实现 Best-Fit 连续物理内存分配算法

best-fit算法概述

Best-Fit是除First-Fit外另外一种连续物理内存分配算法，大致的思路与First-Fit几乎别无二致，仅仅在alloc_pages函数中对从空闲列表中选出用于分配的空闲页的处理上稍有不同。

- best-Fit算法是一种用于内存管理的动态分区分配算法。
- 当一个进程请求内存时，best-Fit 算法会遍历整个空闲内存块列表，寻找最适合（即最小且能容纳请求大小）的空闲分区进行分配。
- 如果找到多个符合条件的分区，则选择最小的那个，这样可以避免大块内存被分割成更小的块而导致的空间浪费。

best-fit算法步骤

- **初始化空闲列表**：初始化一个双向链表来记录所有空闲的物理内存块，并将整个可用内存空间划分为一系列的空闲块。
- 当需要分配内存给一个进程时，遍历空闲块列表，**找到最小的满足需求的空闲块**。
- 如果找到了满足需求的空闲块，将其分割成两部分：一部分分配给进程，另一部分保留为新的空闲块。
- 更新分配后的空闲块列表。
- 如果没有找到合适大小的空闲块，则需要进行空闲块合并或者申请更多的内存。
- 当进程完成后，将其占用的内存块释放，合并相邻的空闲块。
- 重复上述步骤，以满足后续进程的内存分配需求。

best-fit算法实现过程

我们可以创建变量min_size来记录大于等于请求大小的空闲块中最小的那个内存块，代码如下：

```
// 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
    size_t min_size = nr_free + 1;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            if(p->property < min_size) {
                page = p;
                min_size = p->property;
            }
        }
    }
```

best-fit算法优点：

- **内存高效利用**：最佳适应算法通过分配最小的合适内存块，可以有效地利用内存，减少浪费
- **减少内存碎片**：由于它倾向于分配较小的内存块，这些块不太可能变成碎片，从而有助于减少内存碎片
- **改善内存利用率**：通过精确匹配请求大小，可以提高内存的整体利用率
- **最小化外部碎片**：最佳适应算法通过选择最合适的块来最小化外部碎片的产生

best-fit算法缺点：

- **计算开销大**：搜索最佳适应块的过程可能耗时且需要更复杂的搜索算法，增加了系统的计算负担
- **可能导致内部碎片增加**：最佳适应算法可能会留下许多太小的、难以利用的内存块，从而增加内部碎片
- **内存分配速度慢**：由于需要检查整个内存以找到最合适的块，这可能导致内存分配过程变慢
- **可能导致内存浪费**：在某些情况下，最佳适应算法可能会因为过分追求匹配而留下太小的内存块，这些块可能无法被有效利用

改进思考：

- best fit算法相较于first fit算法而言虽然能够找到最小的满足需求的空闲块，缓解内存空间碎片化，但是却需要以遍历全部空闲块列表为代价，当空闲区较大时，浪费的时间较多。针对这点我们可以考虑使用索引或数据结构加速搜索过程，例如使用二叉搜索树、红黑树或哈希表等数据结构来存储和组织空闲分区信息，以便快速查找满足需求的分区
- Next Fit 算法会从上次分配的位置开始，顺序查找空闲分区列表，直到找到一个合适的分区来满足进程大小需求。这样可以减少遍历空闲分区列表的时间。
- **内存回收策略**：可以考虑采用更加智能的内存回收策略，例如LRU（最近最少使用）算法，对长时间未被访问的内存块进行回收。
- **内存压缩**：在内存分配前进行压缩，将所有已分配的内存块移动，以减少碎片并释放更大的连续空间

扩展练习Challenge：buddy system（伙伴系统）分配算法

Buddy 系统内存管理算法设计文档

算法概述

- **算法名称**：Buddy 系统内存管理算法
- **目标**：
  - 实现一个基于 Buddy 系统的物理内存管理器：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等。
  - 能够进行内存块的分配和释放，并支持合并和拆分操作。
  - 尽量减少内存碎片问题。
- **假设和约束**：
  - 假设内存大小为 2 的幂次。
  - 系统最多管理 32768 个页框，最大内存块阶数为 15（即 `2^15` 页）。
  - 内存的分配和释放粒度为页大小。

需求分析

- **输入描述**：
  - 分配请求：要求分配 `n` 个页框的内存块。
  - 释放请求：要求释放某个指定的内存块，给出其起始地址和大小。
- **输出描述**：
  - 成功分配时返回分配内存块的起始偏移。
  - 分配失败时返回 `-1` 表示失败。
  - 成功释放时更新内存状态，并尝试合并可合并的相邻块。
- **边界情况**：
  - 请求分配的内存块大小超过系统内存总量，不进行分配操作，`buddy_system_alloc_pages`函数返回的地址为空。
  - 释放内存时，输入为空地址则不进行操作，`buddy_system_free_pages`函数直接返回，避免了未分配还要释放的情况。
  - 释放超出系统管理范围的内存块，应触发失败。

设计细节

- **设计思路**： Buddy 系统使用二叉树结构管理内存块，树的根节点代表整个可用内存块，每个节点代表其祖先节点的内存块的某个部分。通过递归划分内存，直到找到合适大小的块进行分配。释放时，会检查相邻的内存块是否可以合并，从而恢复大块的内存。

  - **Buddy 树结构**：通过数组 `buddy_longest` 管理二叉树节点，数组的每个元素表示该节点对应的最大空闲块大小。
  - **内存块划分**：内存从根节点开始，按 2 的幂次划分，每个节点根据左右子节点来判断是否可以分配或合并内存块。
  - **关键操作**：
    1. **分配操作**：从树根开始，根据块大小和空闲情况找到合适的块，将其标记为已分配，更新其祖先节点状态。
    2. **释放操作**：将块标记为空闲，并尝试与相邻的 buddy 块合并，恢复更大的块。

- **核心步骤**：

  1. **初始化**：
  - `buddy2_init` 初始化二叉树，设置每个节点的初始块大小。
  - `buddy_system_init_memmap`调用`buddy2_init`初始化二叉树，然后初始化物理内存的映射，并将物理页（Page 结构）初始化为可管理状态。

2. **分配内存**：
   - `buddy2_alloc` 根据请求的大小分配内存，遍历 Buddy 树，从根节点开始递归划分，找到合适的块并将其标记为已分配。分配完成后，更新 Buddy 树的状态。
   - `buddy_system_alloc_pages`调用`buddy2_alloc`获得分配的内存块的偏移量，并在实际的物理内存中获得相应的物理页地址。
3. **释放内存**：
   - `buddy2_free` 根据偏移量释放内存块，尝试与相邻的块合并，并更新 Buddy 树的状态。
   - `buddy_system_free_pages`首先通过实际的物理页地址计算得到二叉树结构中内存块的偏移量，再调用`buddy2_free`进行释放和合并操作。
4. **最大可分配块查询**：`buddy_system_max_alloc` 查询当前系统可分配的最大内存块大小。

- **数据结构**：
  - `buddy_longest[]`：长度为 `2 * MAX_SIZE - 1` 的数组，用于管理二叉树中各节点对应的最大空闲块大小。
  - `free_area`：包含 `free_list` 和 `nr_free`，管理空闲内存页链表。

复杂度分析

- **时间复杂度**：
  - **分配操作**：`O(log n)`，因为 Buddy 系统使用二叉树管理内存，每次分配时最多需要遍历树的高度，树的高度与总内存块数成对数关系。
  - **释放操作**：`O(log n)`，释放时最多需要递归合并相邻块，合并操作同样是基于二叉树的高度。
- **空间复杂度**：`O(n)`，需要额外存储一个大小为 `2 * MAX_SIZE - 1` 的数组，用于维护二叉树节点状态。

扩展与优化

- 支持非 2 的幂次内存块大小分配，可以通过更精细的内存管理实现。
- 对内存进行更细粒度的管理，减少内存碎片。

测试用例

- **测试用例 1**：分配全内存并释放：分配 32768 个页。
  - **预期结果**：成功分配和释放，系统最大可分配块应恢复为 32768。
- **测试用例 2**：边界条件测试：分配超过最大可分配内存 32769 个页。
- **预期结果**：分配失败。
- **测试用例 3**：连续分配测试1：依次分配 16000、6000、9000 页。
- **预期结果**：前两次分配成功，第三次分配失败，释放后恢复最大可分配块。
- **测试用例 4**：连续分配测试2：依次分配 16000、6000、8000 页。
- **预期结果**：三次都分配成功，释放后恢复最大可分配块。

- **测试用例 5**：释放与合并测试：通过`buddy_system_max_alloc`函数动态获取前几次测试中可分配的最大内存块大小，再次分配 32768 个页。
    - **预期结果**：成功分配和释放，通过获取的结果验证了释放和合并操作的正确进行。

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。

slub分配算法较为繁杂，未能完成，这里介绍一下理论知识和一些个人初步想法。

理论知识：

SLUB（Simplified LInux Memory Allocator）分配算法是Linux内核中用于管理小块内存分配的机制。它是为了解决内核中小块内存分配的需求而设计的，这些小块内存的尺寸通常小于一页（通常是4KB）。SLUB算法在2.6版本的Linux内核中被引入，旨在简化和优化之前的SLAB分配器。

相关的管理数据结构：(具体代码可以参考Linux官方相关代码)

 1729826675189

 1729826758418

 1729826787785

slub接口：

```
struct kmem_cache *kmem_cache_create(const char *name,
    size_t size,
    size_t align,
    unsigned long flags,
    void (*ctor)(void *));
void kmem_cache_destroy(struct kmem_cache *);
void *kmem_cache_alloc(struct kmem_cache *cachep, int flags);
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

1)kmem_cache_create是创建kmem_cache数据结构，参数描述如下：

name：kmem_cache的名称

size： slab管理对象的大小

align：slab分配器分配内存的对齐字节数(以align字节对齐)

flags：分配内存掩码

ctor： 分配对象的构造回调函数

2)kmem_cache_destroy作用和kmem_cache_create相反，就是销毁创建的kmem_cache。

3)kmem_cache_alloc是从cachep参数指定的kmem_cache管理的内存缓存池中分配一个对象，其中flags是分配掩码，GFP_KERNEL是不是很熟悉的掩码？

4)kmem_cache_free是kmem_cache_alloc的反操作

slab分配器提供的接口该如何使用呢？其实很简单，总结分成以下几个步骤：

1)kmem_cache_create创建一个kmem_cache数据结构。

2. 使用kmem_cache_alloc接口分配内存，kmem_cache_free接口释放内存。

3. release第一步创建的kmem_cache数据结构。

数据结构间的关系：

 1729826959770

slub分配过程：

SLUB的分配过程包括创建slab缓存(kmem_cache)、分配对象(object)以及释放对象。SLUB分配器从伙伴系统分配内存，然后将其分割成小块内存进行管理。每个小块内存被称为一个对象，这些对象被组织成单向链表，以便于快速分配和释放。

工作机制：

- 创建SLAB缓存（kmem_cache_create）：当内核需要分配特定大小的对象时，它会创建一个SLAB缓存。这个缓存是针对特定大小的对象的，例如，所有的进程描述符可能会有一个共同的SLAB缓存。
- 分配对象（kmem_cache_alloc）：当需要分配一个对象时，内核会从对应的SLAB缓存中分配。如果CPU的局部缓存中有可用的对象，就直接从这里分配。否则，会从全局缓存中分配一个slab页面到CPU的局部缓存，然后从这个slab页面中分配对象。
- 释放对象（kmem_cache_free）：当对象不再需要时，内核会将其释放回对应的slab页面。如果释放后slab页面中没有其他对象被使用，这个slab页面可能会被释放回伙伴系统。

个人初步想法：

按照要求分为两部分：第一层和第二层

第一层：基于页大小的内存分配

- **页分配**:

  可以用伙伴系统（Buddy System）从物理内存中分配页。伙伴系统能够高效地管理大块内存，并且能够快速地分配和回收页。

- **页管理**：

  维护一个全局的空闲页链表，用于跟踪所有可用的页。

  第二层：基于任意大小的内存分配

  1.**缓存创建**：

  为每种对象大小创建一个slab缓存（kmem_cache）。每个缓存专门用于分配特定大小的对象。

  2.**对象分配**

  3.**对象释放**

扩展练习Challenge：硬件的可用物理内存范围的获取方法

问题描述

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

实验中的获取方法

```
va_pa_offset = PHYSICAL_MEMORY_OFFSET; // 设置虚拟到物理地址的偏移：

// 获取物理内存信息，下面变量表示物理内存的开始、大小和结束地址
uint64_t mem_begin = KERNEL_BEGIN_PADDR;
uint64_t mem_size = PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR;
uint64_t mem_end = PHYSICAL_MEMORY_END; // 硬编码取代 sbi_query_memory()接口
```

这是在代码中硬编码指定可用物理内存的信息，包括其起始地址、大小和结束地址。这种硬编码方式适用于那些无法在运行时获取物理内存信息的情况，或者在引导加载程序（bootloader）等早期阶段使用。一旦这些信息被硬编码到代码中，操作系统内核可以使用它们来管理可用的物理内存。这种方法可能不够灵活，因为它无法应对物理内存配置的变化。

推荐使用SBI（Supervisor Binary Interface）的sbi_query_memory()接口或其他动态获取的方式，相比之下有以下优势：

- **动态适应性**： 物理内存配置可能因不同的硬件环境或不同的引导加载程序而有所不同。使用硬编码方式限制了操作系统的灵活性，不适应多种硬件配置。动态获取方式可以根据实际硬件情况来获取内存信息，从而适应不同的环境。
- **易维护性**： 硬编码方式在内核代码中引入了硬编码的数字，这可能会导致代码的可读性下降，同时也增加了代码的维护难度。如果物理内存配置发生变化，就需要手动修改代码，而动态获取方式则更易于维护和更新。
- **硬件抽象**：通过SBI接口，操作系统可以不必关心具体的硬件细节，而是通过统一的接口与硬件交互。这种抽象层次的设计简化了操作系统的开发，尤其是在不同的硬件平台上
- **可扩展性**：SBI的设计允许扩展，以支持更多的功能和硬件特性。例如，除了查询内存信息，SBI还支持其他如计时器、中断处理等扩展功能

使用这种方法首先需要在操作系统内核中初始化SBI或相关的接口，以便能够与硬件通信；然后使用SBI的sbi_query_memory()接口或其他合适的接口来查询物理内存的信息。最后接收接口返回的物理内存信息，并根据获取的物理内存信息，初始化内存管理子系统。

其他获取方法

1. **使用 BIOS 中断**：可以通过调用 BIOS 提供的中断（如 INT 0x15, AX=0xE820）来获取系统内存的布局。这种方法能够返回系统中物理内存的范围、状态和类型。
2. **内存映射的设备**：现代操作系统可以通过内存映射技术访问硬件信息。通过对特定的设备寄存器进行读取，操作系统可以获取内存的分布情况。 例如，某些平台提供了内存控制器的寄存器，能够查询内存的使用状态。
3. **ACPI（高级配置和电源接口）**：ACPI 提供了一种标准方法，允许操作系统获取硬件资源的信息，包括可用内存的范围。 操作系统可以解析 ACPI 表（如 RSDT 和 MADT）来获取内存区域的详细信息。
4. **使用操作系统的启动参数**：在某些情况下，操作系统可以在启动时接收参数，这些参数包含有关可用内存的信息。 例如，Linux 可以在启动时通过内核命令行参数（如 mem=）来指定可用内存。
5. **扫描内存地址**：操作系统可以通过尝试访问内存的方式来探测可用物理内存。这种方法通常需要小心处理，因为错误的访问可能导致系统崩溃。 一般来说，可以通过访问特定范围内的内存地址并观察系统的响应来判断这些地址是否有效。
6. **读取系统日志或配置文件**：某些系统可能在启动过程中生成日志文件，记录内存的初始化状态。操作系统可以读取这些日志或配置文件，以获取可用物理内存的信息。
7. **使用特定的系统调用或API**：许多操作系统提供特定的系统调用或API，允许应用程序或内核模块查询系统的内存状态。 在 Linux 中，可以通过 /proc/meminfo 文件获取当前系统的内存信息。
8. **使用 UEFI**：在现代计算机上，UEFI（统一可扩展固件接口）取代了传统的 BIOS，并提供了更强大的功能来获取系统信息，包括可用内存的范围。 操作系统可以通过 UEFI 的系统表来查询硬件配置。

知识点梳理

基本分页存储管理

内存空间分为大小相等的分区，每个分区就是一个页框（页帧、内存块、物理块、物理页面） 将进程的逻辑地址空间分为与页框大小相等的一个个部分，成为页（页面） 进程的页和内存的页框一一对应。每一个进程都有一个页表来记录。（页号+块号） 系统中设置一个页表寄存器，存放页表在内存中国的起始地址F和页表长度M。程序未执行时，页表的始址和页表长度放在进程控制块（PCB），当进程被调度时，操作系统内核回把它们放到页表寄存器。

物理地址和虚拟地址

页表就是那个“词典”，里面有程序使用的虚拟页号到实际内存的物理页号的对应关系，但并不是所有的虚拟页都有对应的物理页。虚拟页可能的数目远大于物理页的数目，而且一个程序在运行时，一般不会拥有所有物理页的使用权，而只是将部分物理页在它的页表里进行映射。

在 sv39中，定义**物理地址(Physical Address)有 56位，而虚拟地址(Virtual Address) 有 39位**。实际使用的时候，一个虚拟地址要占用 64位，只有低 39位有效，我们规定 63–39 位的值必须等于第 38 位的值（大家可以将它类比为有符号整数），否则会认为该虚拟地址不合法，在访问时会产生异常。 不论是物理地址还是虚拟地址，我们都可以认为，最后12位表示的是页内偏移，也就是这个地址在它所在页帧的什么位置（同一个位置的物理地址和虚拟地址的页内偏移相同）。除了最后12位，前面的部分表示的是物理页号或者虚拟页号。

物理内存管理动机与技术

（1）内存是一种稀缺资源，计算机需要管理内存以优化性能；在本次实验中（同时也是当前流行的技术），我们使用的内存管理方法主要包含两个技术——分页存储技术和虚拟内存技术。需要注意的是，前者同时也是后者的条件。

（2）如果将一个进程连续地放入内存，则会产生很多碎片从而降低了内存使用率。分页技术实际上基于了离散分配的思想，将进程分为不同的小块来存储，这样可以减少内存中的碎片。分页后就会产生一个问题，就是进行中的代码在内存中对应哪些块，这就需要页表来解决。

（3）如果一个进程的空间很大以至于超过整个内存（如大型游戏），就需要使用虚拟内存技术实现逻辑上的内存扩大。基于局部性原理，一个进程的所有代码并不需要在短时间内均被运行，一个逻辑上大地址的指令在物理上也可以是一个较小的地址。虚拟技术需要实现信息的调入和换出，同时也需要通过页表实现虚拟地址和物理地址的对应即可。