基于伙伴分配算法的内存管理系统-实验指导书

一、 实验背景

1.1 xv6现有的内存分配策略

在 xv6 操作系统中，内存分配的方式较为简单，采用一个链表结构管理空闲内存页（page）。这是一个经典的基于固定大小块分配的内存分配器模型，专门用于管理页大小的内存块。其内存分配方式的特点如下：

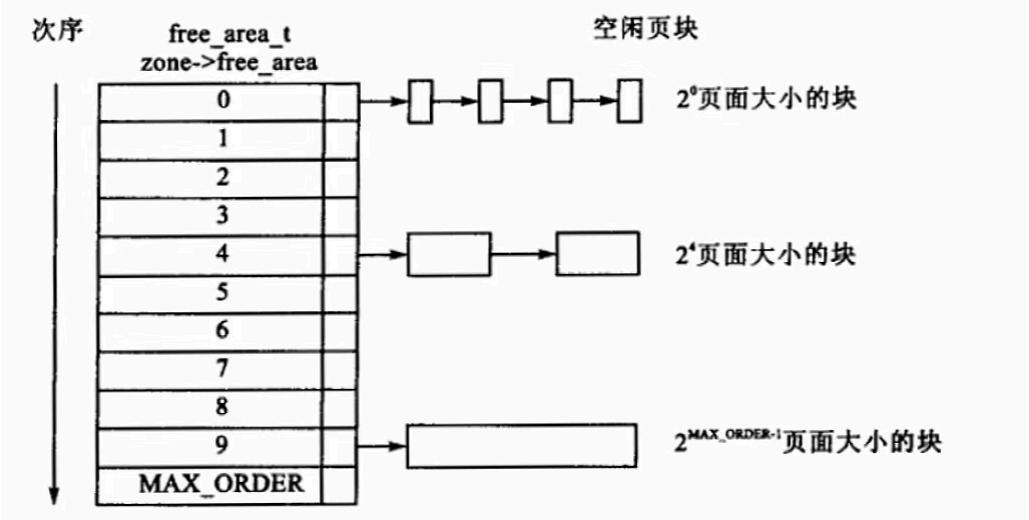
- **优点：**
  1. 简单易实现：基于链表的内存管理逻辑简单，容易在小型系统中实现。
  2. 分配效率高：每次分配或释放内存页的时间复杂度为 O(1)（只需要操作链表头）。
  3. 适合页大小内存分配：分配单一大小（页大小）的内存块，符合操作系统中页表机制的需求。
- **缺点：**
  1. 内存碎片化：无法有效处理小块内存分配或释放，可能会导致严重的内存碎片化。
  2. 固定块大小：无法灵活支持不同大小的内存分配需求（如小块内存）。
  3. 没有合并机制：内存释放后无法将相邻的空闲块合并为更大的块。

1.2 伙伴分配器（Buddy Allocator）的分配策略

伙伴分配器是一种内存分配算法，用于操作系统或其他需要管理内存的系统软件中。

1. 基本原理

- **内存块划分**：它将内存划分为大小为 2 的幂次方的块，如 16 字节、32 字节、64 字节等。这些内存块通过一种称为 “伙伴关系” 的规则相互关联。两个大小相同且相邻的内存块被视为伙伴。例如，在一个以 32 字节为单位划分内存的系统中，地址从 0 - 31 的内存块和 32 - 63 的内存块是伙伴。
- **分配过程**：当有内存分配请求时，伙伴分配器首先找到能满足请求大小的最小 2 的幂次方大小的内存块。例如，如果请求分配 20 字节内存，它可能会找到 32 字节的内存块。如果找到的内存块是空闲的，就将其分配出去。如果没有找到合适大小的空闲块，它会尝试找到更大一级的空闲块，然后将其分割成两个伙伴块，重复这个过程，直到得到合适大小的空闲块。

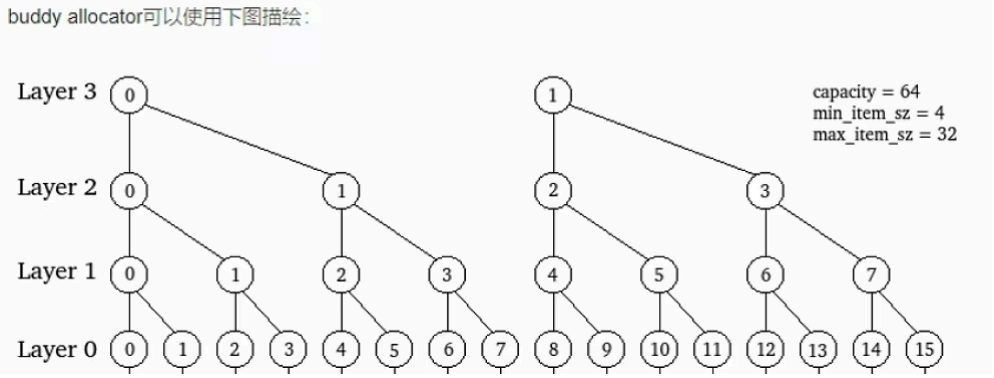


- **回收过程**：当一个内存块被释放时，伙伴分配器会检查它的伙伴块是否空闲。如果伙伴块也是空闲的，就将它们合并成一个更大的内存块。例如，有两个相邻的 32 字节空闲块，它们会被合并成一个 64 字节的空闲块。这个合并过程会一直持续，直到找不到空闲的伙伴块或者内存块达到了系统管理的最大块大小（例如，整个可用内存区域）。这一特性使得伙伴分配器可以动态调整内存块大小，从而最大程度地减少内存碎片。

2. 具体实现分析

伙伴分配器分配和释放物理页的数量单位为阶。其分配 `n` 阶页块的详细流程如下：

- **空闲 `n` 阶页块检查与分配**：
  首先尝试查看是否存在空闲的 `n` 阶页块。若存在，则直接将该空闲 `n` 阶页块进行分配，此过程相对简单直接，可快速满足内存分配需求。若不存在空闲的 `n` 阶页块，则进入下一步骤。
- **空闲 `(n + 1)` 阶页块处理**：
  接着检查是否有空闲的 `(n + 1)` 阶页块。若存在这样的空闲 `(n + 1)` 阶页块，则对其执行分裂操作，将其拆分为两个 `n` 阶页块。其中一个 `n` 阶页块插入到空闲 `n` 阶页块链表中，以便后续其他分配请求使用；另一个 `n` 阶页块则直接分配出去，以满足当前的 `n` 阶页块分配需求。若此步骤中未找到空闲的 `(n + 1)` 阶页块，则继续下一步骤。
- **空闲 `(n + 2)` 阶页块处理**：
  再查看是否有空闲的 `(n + 2)` 阶页块。若存在，先将该 `(n + 2)` 阶页块分裂为两个 `(n + 1)` 阶页块，其中一个 `(n + 1)` 阶页块插入到空闲 `(n + 1)` 阶页块链表中。然后将另一个 `(n + 1)` 阶页块进一步分裂为两个 `n` 阶页块，把其中一个 `n` 阶页块插入到空闲 `n` 阶页块链表，而另一个 `n` 阶页块则分配出去。若此步骤仍未找到合适的空闲页块，则继续向更高阶查找是否存在空闲页块，直至找到合适的页块并完成分配操作或者确定无法满足分配请求为止。



3. 主要特点

- **高效性**：
  由于内存块大小是 `2` 的幂次方，分配和回收操作可以通过简单的位运算和指针操作来实现，具有较高的执行效率。其分配和回收的时间复杂度通常为 ，其中 `n` 是内存块的数量。
- **减少内存碎片**：
  通过合并伙伴块，能够有效地减少内存碎片。内存碎片是指内存中存在许多小的、不连续的空闲内存块，可能导致即使总空闲内存足够，但无法满足较大内存分配请求的情况。伙伴分配器可以将这些小的空闲块合并成大的空闲块，提高内存的可用性。
- **易于管理**：
  内存块的组织方式比较规则，便于对内存的使用情况进行监控和管理。例如，可以很容易地统计出不同大小的空闲内存块的数量，以及整个内存的使用效率等信息。

二、实验目的

在xv6系统中加入伙伴分配器，优化xv6现有的内存分配策略，主要体现在以下方面：

1. 优化内存分配效率

- **减少碎片产生**：xv6 原有的内存分配器可能会随着时间的推移产生内存碎片。碎片的产生会导致即使系统中有足够的空闲内存，但由于这些空闲内存不连续，无法满足较大内存块的分配请求。伙伴分配器通过其合并机制，能有效减少这种碎片。例如，当相邻的两个相同大小的空闲 “伙伴” 内存块被释放时，伙伴分配器会将它们合并成一个更大的空闲内存块，这样就可以更好地满足后续可能出现的较大内存请求。
- **快速分配和回收内存**：伙伴分配器在分配和回收内存时具有较高的效率。它根据内存块大小的二进制幂次方来组织内存，在分配内存时，能够快速定位到合适大小的空闲内存块。其时间复杂度相对较低，通常为，其中是内存块的数量。这比一些简单的内存分配算法在性能上更有优势，能够更快地响应进程对内存的请求和释放操作。

2. 增强内存管理的可预测性

- **固定大小内存块管理**：伙伴分配器将内存划分为固定大小（2 的幂次方）的内存块，这使得内存管理更加规则和可预测。
- **动态适应内存需求变化**：在系统运行过程中，进程对内存的需求是动态变化的。伙伴分配器可以根据实际的内存需求灵活地分配和回收内存块。当系统需要为新的进程或功能分配大量内存时，它可以拆分较大的空闲内存块来满足需求；当内存需求减少时，又可以将释放的内存块合并，释放出更大的连续空闲内存空间。这种动态适应的特性使得 xv6 能够更好地应对不同的工作负载和应用场景，如在服务器环境中处理多个并发的网络连接请求，或者在嵌入式系统中根据任务的优先级和资源需求分配内存。

三、实验过程

3.1 实验一：实现伙伴分配器

3.1.1 实验说明

本实验基于伙伴分配算法，要求实现一个内存管理系统，包含内存分配（`buddy_malloc`）、内存释放（`buddy_free`）、内存初始化（`buddy_init`）等功能。实验过程中将重点学习如何实现以下功能，其代码框架已经给出：

1. **内存分配与释放：**
   - `buddy_malloc`：分配指定大小的内存块，确保返回的内存块满足对齐要求，并在需要时进行块的拆分。
   - `buddy_free`：释放指定内存块，并在适当时合并相邻的空闲块。
2. **内存初始化与管理：**
   - `buddy_init`：初始化内存管理系统，包括计算内存大小、初始化空闲链表、分配管理结构和标记已使用区域。
3. **辅助函数：**
   - `bit_is_set`、`set_bit`、`clear_bit` 等：用于操作内存中每个块的状态，标记块是否已分配、已拆分等。
   - `find_first_k`：根据请求的内存大小，找出第一个可以容纳该请求的块大小。
4. **测试与验证：**
   - `buddy_malloc` 和 `buddy_free` 的正确性验证，确保内存分配与释放过程中不会出现内存泄漏或非法访问。
   - 合并操作的正确性测试，验证内存块合并时是否能正确释放并合并相邻块。

3.1.2 实验步骤

步骤 1: 理解伙伴分配算法

在实现内存管理系统之前，首先需要理解伙伴分配算法的基本概念。伙伴分配算法的核心思想是：每个内存块的大小是2的幂次方，空闲块组成一个二叉树。相邻的空闲块组成“伙伴”，每次分配或释放内存时，都会检查相邻的伙伴是否为空闲块。如果是空闲块，则进行合并操作。

步骤 2: 查看内存管理结构

本系统主要由以下几个数据结构组成，用来实现内存管理：

- `Sz_info`：用于表示每种大小内存块的信息，包括：
  - `free`：空闲链表，管理空闲的内存块。
  - `alloc`：一个数组，跟踪哪些内存块已分配。
  - `split`：一个数组，跟踪哪些内存块已经被拆分。

```
struct sz_info {
    buddy_list free;  // 空闲链表
    char *alloc;      // 分配标志数组
    char *split;      // 拆分标志数组
};
```

- `buddy_sizes`：一个 `Sz_info` 类型的数组，每个元素表示不同大小的内存块。
- `buddy_base`：表示内存管理区域的起始地址，所有的内存分配与释放都以此为基准进行操作。

步骤 3: 实现内存分配与释放

- **实现 `buddy_malloc` 函数**

  `buddy_malloc` 函数负责从内存池中分配指定大小的内存块。函数逻辑如下：

  1. 根据请求的内存大小计算所需的块的最小大小。
  2. 查找合适的空闲块并从空闲链表中移除该块。
  3. 如果该块的大小大于请求的大小，进行块的拆分，每次分裂都会创建一个新的小块，并将它们加入到更小的空闲链表中。

- **实现 `buddy_free` 函数**

  `buddy_free` 函数负责释放内存块并进行合并操作。合并操作会检查释放块的伙伴是否为空闲块，如果为空闲块，则将它们合并为一个更大的块。函数逻辑如下：

  1. 查找释放内存的块大小。
  2. 标记该块为已释放。
  3. 检查伙伴块的状态，如果伙伴块是空闲的，则进行合并操作：将两个块合并为一个更大的块，并可能递归合并父级块。
  4. 合并后的块再次加入到空闲链表中。

步骤 4: 实现初始化

- `buddy_init`：

  1. 确定内存管理的总大小：计算出需要管理的内存区域的大小，将从`base`到`end`的内存交给伙伴分配器管理。
  2. 初始化每种大小的空闲链表：为每种内存块大小初始化一个空闲链表。
  3. 初始化 `alloc` 和 `split` 数组：这些数组分别用于标记内存块的分配状态和拆分状态。
  4. 标记不可用内存区域：将不可用区域（例如操作系统的内核区域）标记为已分配，避免这些区域被分配。

- `buddy_initfree_pair`：检查某个块是否已分配，如果还未分配且它的伙伴块已被分配的，则将其放入相应的空闲链表中，并返回该块的大小。

  1. 寻找伙伴块: `buddy` 计算的是当前块 `bi` 的伙伴块索引。由于在伙伴分配中，两个相邻的内存块组成一个伙伴，`bi` 和 `buddy` 是一对伙伴，索引差为 1。
  2. 检查分配位: `get_mutual_bit` 用来检查当前块是否已经被分配。通过 `buddy_sizes[k].alloc` 中对应的位置判断，如果该位置的位已经被设置为 1，则说明当前块已被分配。
  3. 判断并添加到空闲链表:
     - 如果块 `bi` 被标记为已分配（即已经有相应的资源被分配给它），就将其放入 `buddy_sizes[k].free` 中。
     - 这个判断还包括是否当前块是 `left` 还是 `right`，如果 `bi` 与 `left` 相同，则将当前块（地址为 `address(k, bi)`）放入链表；否则，将其伙伴块（地址为 `address(k, buddy)`）放入链表。

    - 图中用 `1,1` 表示块已被分配且已被分割。
    - 最底层 `size 0` 级别的每个最小块用 `0` 或 `1` 表示是否被分配。
    - 图中着色部分内存块都应加入空闲链表中，这些空闲块都出现在每个分级状态的两端，因此我们只需要考察是否加入左右两端的内存块即可。
- `buddy_initfree`：函数通过对每个块级别的内存块进行遍历，初始化每个块并把空闲的块放入对应的空闲链表中，最后返回所有空闲内存的总大小。
    - 处理左块和右块，调用 `buddy_initfree_pair(k, left)` 将其初始化并放入空闲链表。如果 `right` 索引大于 `left`，则继续处理右块。

步骤 5：辅助函数的实现

为了实现内存分配和释放，文件中还包含了一些辅助函数。以下是几个关键的辅助函数说明：

1. `bit_is_set`：检查某个位置的位是否为 1，用于判断某个内存块是否已经被分配。
2. `set_bit`：将指定位置的位设置为 1，用于标记某个内存块已经分配。
3. `clear_bit`：将指定位置的位清除（设置为 0），用于标记某个内存块为空闲。
4. `find_first_k`：根据给定请求的内存大小，返回最小的满足该大小的块的指数（即2的幂）。
5. `block_index` **和** `address`：将内存地址转换为块索引，或将块索引转换为内存地址，确保分配和释放时内存地址的正确性。
6. `log2`：计算一个数的二进制对数，用于计算需要多少次拆分才能获得足够大的内存块。

辅助函数可以通过位操作实现，用于设置、清除、反转指定的位。这些操作直接影响内存块的分配和释放。通过这些操作，我们能够有效地追踪每个内存块的分配状态、是否已拆分、以及是否需要合并。

步骤 6：额外优化

1. **高效的内存分配与释放**
    - 延迟合并：当发现释放的块和其伙伴块可以合并时，代码采用延迟合并策略。这样，内存块在释放时不会立即合并，而是等待后续的内存操作。这减少了频繁合并可能带来的性能损失。
    - 内存块状态标记：通过 `alloc` 和 `split` 位图来跟踪每个内存块的状态（已分配或已拆分），使得内存分配和释放的操作更为高效，避免了复杂的数据结构操作。
2. **内存对齐**
    - 在 `buddy_init` 中，`buddy_base` 的地址通过 `ROUNDUP` 宏对齐到 `LEAF_SIZE` 的倍数。`ROUNDUP` 宏确保了起始地址是 `LEAF_SIZE` 的整数倍，即对齐到最小块大小的边界。
3. **并发安全性**
    - 使用了自旋锁（`spinlock`）来保护内存分配和释放操作，确保在多核处理器系统中并发访问内存时的安全性。每次分配和释放操作都需要获取锁，防止了数据竞争问题。
4. **空闲链表的管理**
    - 内核态中已添加了`list.c`文件，实现了一个双向链表的基本操作，通过链表来动态地管理内存块的分配和释放。

- 代码通过 `buddy_sizes[k].free` 来维护每个大小级别的空闲链表，并利用 `lst_push` 和 `lst_pop` 等操作进行动态管理。这使得在内存分配和释放时，能够快速找到空闲块并进行操作。

3.1.3 测试方法

运行 `buddytests` 测试，可以看到第一个测试 `buddytest` 已通过，如图所示。

```
$ buddytests
Allocated memory at 0x000000008032D000
Allocated memory at 0x000000008032D000
Allocated memory at 0x000000008032D000
Freed memory at 0x000000008032D000
Freed memory at 0x000000008032D000
Freed memory at 0x000000008032D000
buddytest: OK
optimaltest: start
expected to allocate at least 31950, only got 31717
optimaltest: FAILED
```

3.2 实验二：优化伙伴分配器部分

3.2.1 实验说明

由于如果每种规格大小的内存块都保留一个比特位，用于标识该块是被占有还是空闲，伙伴分配器会使用大量的内存用于保存这些比特位，这导致其空间利用效率变得低下。考虑进一步优化伙伴分配器，可以利用一个比特位表示一对 `buddy` 内存块的状态，以节省内存。对于每对 `buddy` 块，利用 `XOR` 操作来判断其空闲或被占用的状态。这样可以在释放或分配内存时，依赖于一个比特位来决定是否合并这两个内存块。

- 可行性
  - 对于空闲链表的操作，当且仅当两个`buddy`块一个空闲，而另一个被占用才能将其中一个加入空闲链表，因此`XOR`操作可行。
  - 在释放过程中，传入的内存块一定是被占用的，释放掉之后，如果其`buddy`块被占用，那么XOR为1，否则XOR为0，同样可以用XOR进行判断。
- 测试与验证
  - 内存利用率是否提升：比较优化前后相同情况下的内存分配情况，确保优化后的方案能够有效节省内存。
  - 系统稳定性：测试系统在进行大量内存分配和释放时，优化方案是否能够保持稳定运行，并且正确地合并 `buddy` 块。

3.2.2 实验步骤

步骤 1： 修改伙伴分配器中的内存块表示

使用`flip_mutual_bit`和`get_mutual_bit`函数来管理这个共同的比特位

- `flip_mutual_bit`：翻转 buddy 块对的状态，便于在分配或释放内存时调整状态。
- `get_mutual_bit`：用于查询给定 buddy 块对的当前状态。返回 1 或 0，表示 buddy 块对的状态（例如，空闲或已分配）。

步骤 2：优化 `buddy_initfree_pair` 函数

优化后的 `buddy_initfree_pair` 函数不再为每个内存块分配单独的比特位，而是通过 `flip_mutual_bit()` 来判断某个内存块是否应该加入空闲列表。通过判断 `left == bi` 来决定是将当前块加入空闲列表，还是将它的 buddy 块加入空闲列表（`left`代表的是在相应`size k`对应的内存块`p`的后一块）。

我们发现，由于`left`对应的块永远为空闲，而right对应的块永远都是已被分配的。于是我们应该将`left`块加入到空闲链表中，并将`right`块的`buddy`加入空闲链表。

步骤 3：优化 `buddy_free` 函数

在释放内存块时，通过 `XOR` 操作判断是否需要合并 `buddy` 块。如果释放的内存块的 `buddy` 为空闲，则应当进行合并。

步骤 4：内存块数组大小调整

由于每对 buddy 块只需要一个比特位来表示状态，所以可以调整 `alloc` 数组的大小，减少内存消耗。根据每对 buddy 块使用一个比特位进行内存管理，原本需要的数组大小可以减少一半。

3.2.3 测试方法

- 再次运行`buddytests`测试，可以看到`optimaltest`此时也已通过，如图所示。

```
$ buddytests
Allocated memory at 0x000000008022C310
Allocated memory at 0x000000008022C320
Allocated memory at 0x000000008022C330
Freed memory at 0x000000008022C310
Freed memory at 0x000000008022C320
Freed memory at 0x000000008022C330
buddytest: OK
optimaltest: start
optimaltest: OK
```

- 运行测试文件`grade-lab-buddy`，可以看到测试通过，如图所示。

```
make: 'kernel/kernel' is up to date.
== Test buddytests == buddytests: OK (3.4s)
== Test usertests == usertests: OK (44.8s)
Score: 100/100
```

四、优化与对比

4.1 原始内存分配器

在 xv6 环境中存在一个简化的内存分配器，用于内核内存的管理，提供内存页的分配（`kalloc`）和释放（`kfree`）操作，同时也包含了内存碎片统计功能。`kmem` 分配器只是基于一个简单的 链表结构 来管理空闲内存页。它将所有空闲页面通过链表链接在一起，当需要分配内存时，从链表头部取出一个页面。当页面被释放时，它也会被添加回链表中。

在系统启动时，`kinit` 会初始化内存池，将内核的空闲内存页放入 `freelist` 链表中；通过 `kalloc` 从空闲链表中分配内存页，每次分配都会减少空闲页计数并增加已分配页计数；通过 `kfree` 将已分配的内存页归还给 `freelist`，增加空闲页计数并减少已分配页计数。我们定义了函数 `kmem_fragmentation_statistics` 来打印当前内存池的碎片化情况，包括空闲页、已分配页和它们所占用的内存量。

该内存管理存在一些潜在问题，如 `kfree` 函数通过遍历 `freelist` 来检查重复释放内存，但这种方法的效率较低，为了优化，应该维护一个额外的标志或者使用更高效的数据结构来记录哪些内存页已经被释放；`freerange` 函数逐页将空闲内存页插入 `freelist`，如果内存范围较大，这可能会导致一定的性能损耗。分配内存时，系统会查找一个足够大的空闲块，若找不到合适的块，系统会从较大的块中拆分出适当大小的块，这会导致较大的空闲块数目减少，同时增加更小块的数量；释放内存时，如果释放的块和其相邻的 "`buddy`" 块都为空闲的，它们会被合并成一个更大的块，合并后的块会影响空闲块的数量和空闲内存的总量。

`kmem_fragmentation_statistics` 函数输出的统计结果如下图：（中间的数据已省略）

```
Total Free Pages: 32728, Total Allocated Pages: 0,Total Free Memory: 134053888 bytes, Total Allocated Memory: 0 bytes
Total Free Pages: 32727, Total Allocated Pages: 1,Total Free Memory: 134049792 bytes, Total Allocated Memory: 4096 bytes
Total Free Pages: 32726, Total Allocated Pages: 2,Total Free Memory: 134045696 bytes, Total Allocated Memory: 8192 bytes
Total Free Pages: 32725, Total Allocated Pages: 3,Total Free Memory: 134041600 bytes, Total Allocated Memory: 12288 bytes
Total Free Pages: 32724, Total Allocated Pages: 4,Total Free Memory: 134037504 bytes, Total Allocated Memory: 16384 bytes
Total Free Pages: 32723, Total Allocated Pages: 5,Total Free Memory: 134033408 bytes, Total Allocated Memory: 20480 bytes
Total Free Pages: 32722, Total Allocated Pages: 6,Total Free Memory: 134029312 bytes, Total Allocated Memory: 24576 bytes
Total Free Pages: 32721, Total Allocated Pages: 7,Total Free Memory: 134025216 bytes, Total Allocated Memory: 28672 bytes
Total Free Pages: 32720, Total Allocated Pages: 8,Total Free Memory: 134021120 bytes, Total Allocated Memory: 32768 bytes
Total Free Pages: 32719, Total Allocated Pages: 9,Total Free Memory: 134017024 bytes, Total Allocated Memory: 36864 bytes
Total Free Pages: 32718, Total Allocated Pages: 10,Total Free Memory: 134012928 bytes, Total Allocated Memory: 40960 bytes
.....
Total Free Pages: 32418, Total Allocated Pages: 310,Total Free Memory: 132784128 bytes, Total Allocated Memory: 1269760 bytes
Total Free Pages: 32417, Total Allocated Pages: 311,Total Free Memory: 132780032 bytes, Total Allocated Memory: 1273856 bytes
Total Free Pages: 32418, Total Allocated Pages: 310,Total Free Memory: 132784128 bytes, Total Allocated Memory: 1269760 bytes
Total Free Pages: 32419, Total Allocated Pages: 309,Total Free Memory: 132788224 bytes, Total Allocated Memory: 1265664 bytes
Total Free Pages: 32420, Total Allocated Pages: 308,Total Free Memory: 132792320 bytes, Total Allocated Memory: 1261568 bytes
Total Free Pages: 32421, Total Allocated Pages: 307,Total Free Memory: 132796416 bytes, Total Allocated Memory: 1257472 bytes
Total Free Pages: 32422, Total Allocated Pages: 306,Total Free Memory: 132800512 bytes, Total Allocated Memory: 1253376 bytes
Total Free Pages: 32423, Total Allocated Pages: 305,Total Free Memory: 132804608 bytes, Total Allocated Memory: 1249280 bytes
Total Free Pages: 32424, Total Allocated Pages: 304,Total Free Memory: 132808704 bytes, Total Allocated Memory: 1245184 bytes
Total Free Pages: 32425, Total Allocated Pages: 303,Total Free Memory: 132812800 bytes, Total Allocated Memory: 1241088 bytes
Total Free Pages: 32426, Total Allocated Pages: 302,Total Free Memory: 132816896 bytes, Total Allocated Memory: 1236992 bytes
```

统计输出显示，在初始化时系统的空闲内存页数为 **32728** 页，每页大小为 **4096 bytes**（即 4KB），所以空闲内存的总大小为 **134053888 bytes**，这表明系统在启动时有 134MB 的空闲内存。每次调用 `kalloc()` 分配一个页面后，空闲页面数 `free_count` 减少了 **1**，而已分配的页面数 `allocated_count` 增加了 **1**。每分配一个页面，空闲内存减少 **4096 bytes**，而已分配内存增加 **4096 bytes**。这反映了内存分配的正常行为。随着时间推移，空闲页面数和已分配页面数的变化趋势继续保持一致。每次内存分配后，空闲页面数减小，已分配的页面数增大。`Total Free Memory` 和 `Total Allocated Memory` 之间的差异，正好等于已分配页面数乘以每页的大小（4096 bytes）。通过观察 `Total Free Pages` 和 `Total Allocated Pages` 的变化，有时空闲页面数会恢复增长，而已分配页面数保持稳定。这种变化说明在某些情况下，系统进行了内存释放操作（`kfree`）。释放内存时，空闲页面数会增加，并且已分配页面数会减少。

4.2 伙伴分配算法

伙伴分配算法通过将内存划分为大小为 2 的幂次的块（例如，16 字节，32 字节，64 字节等），并根据块的大小进行分配和释放，来优化内存使用。编写`buddy_fragmentation_statistics`函数，统计并输出内存碎片信息，包括每种大小的空闲块数量及其占用的总内存、`Total Free Memory`和`Total Free Blocks`随着时间变化的情况，反映出内存的碎片化情况和分配/释放行为。

`buddy_fragmentation_statistics`函数输出的汇总统计结果如下图：（中间的数据已省略）

```
Total Free Memory: 131951664 bytes in 29 blocks
Total Free Memory: 131947568 bytes in 28 blocks
Total Free Memory: 131943472 bytes in 27 blocks
Total Free Memory: 131939376 bytes in 27 blocks
Total Free Memory: 131935280 bytes in 26 blocks
Total Free Memory: 131931184 bytes in 26 blocks
Total Free Memory: 131927088 bytes in 25 blocks
Total Free Memory: 131922992 bytes in 26 blocks
Total Free Memory: 131918896 bytes in 25 blocks
Total Free Memory: 131914800 bytes in 25 blocks
Total Free Memory: 131910704 bytes in 24 blocks
Total Free Memory: 131906608 bytes in 25 blocks
Total Free Memory: 131902512 bytes in 24 blocks
......
Total Free Memory: 131279920 bytes in 21 blocks
Total Free Memory: 131284016 bytes in 22 blocks
Total Free Memory: 131288112 bytes in 23 blocks
Total Free Memory: 131292208 bytes in 23 blocks
Total Free Memory: 131296304 bytes in 22 blocks
Total Free Memory: 131300400 bytes in 23 blocks
Total Free Memory: 131304496 bytes in 23 blocks
Total Free Memory: 131308592 bytes in 24 blocks
Total Free Memory: 131312688 bytes in 24 blocks
```

`buddy_fragmentation_statistics`函数输出的完整统计结果如下图：（以最后一条输出为例）

```
Buddy Memory Fragmentation Statistics:
Size Index | Block Size | Free Blocks | Total Free Memory
-----------------------------------------------------------
         0 |         16 |           1 |                16
         1 |         32 |           1 |                32
         2 |         64 |           0 |                 0
         3 |        128 |           0 |                 0
         4 |        256 |           0 |                 0
         5 |        512 |           0 |                 0
         6 |       1024 |           1 |              1024
         7 |       2048 |           1 |              2048
         8 |       4096 |           2 |              8192
         9 |       8192 |           2 |             16384
        10 |      16384 |           1 |             16384
        11 |      32768 |           0 |                 0
        12 |      65536 |           1 |             65536
        13 |     131072 |           1 |            131072
        14 |     262144 |           0 |                 0
        15 |     524288 |           2 |           1048576
        16 |    1048576 |           2 |           2097152
        17 |    2097152 |           1 |           2097152
        18 |    4194304 |           2 |           8388608
        19 |    8388608 |           2 |          16777216
        20 |   16777216 |           2 |          33554432
        21 |   33554432 |           2 |          67108864
        22 |   67108864 |           0 |                 0
        23 |  134217728 |           0 |                 0
-----------------------------------------------------------
Total Free Memory: 131312688 bytes in 24 blocks
```

输出结果显示，空闲内存和空闲块数逐渐减少。最开始有 29 个空闲块和约 132MB 的空闲内存，但随着时间推移，空闲块数逐渐减少，最终稳定在 24 blocks和 131312688 bytes，这表示内存池中的块被分配和合并的过程。空闲块数量的波动（从 29 到 19，再到 24 块）表明，内存池中不同大小的内存块正在被频繁分配和释放，导致空闲块的数量发生变化。

4.3 伙伴分配分配算法的优势

1. **内存管理方面**

- 减少碎片化：通过对内存进行拆分和合并，Buddy 系统可以有效减少内存的外部碎片化。这意味着，即使有很多小块空闲内存，也能将它们合并成较大的块，用于后续的内存分配。
- 内存管理的高效性：Buddy 算法能够快速找到适合的内存块，并且通过合并相邻的空闲块来优化内存利用率，减少内存的浪费。
- 平衡的分配策略：内存的拆分和合并操作使得内存分配更加灵活，可以适应不同的内存需求，从而避免了简单链表管理的局限性。

2. **内存使用方面**

- 内存分配的更为高效：Buddy 分配器在内存分配时能够根据需求拆分合适的内存块，而不仅仅是简单地从链表中拿一个空闲块，这样大大减少了内存碎片化的情况。
- 内存释放后的合并：Buddy 系统通过合并相邻的空闲内存块来减少内存碎片，使得释放操作不会造成大量的小空闲块堆积。这减少了内存池的碎片化，并确保了系统能够更加有效地利用内存。
- 更少的空闲块：从测试结果来看，在加了 Buddy 分配器后，内存池中的空闲块数量更少，内存的利用更加密集。这意味着系统分配了更多内存，而没有浪费很多小块的空闲内存。
- 内存碎片统计结果更好：通过 `kmem_fragmentation_statistics()` 输出的统计信息，我们可以看到加了 Buddy 分配算法后，空闲内存的分布更加均匀，减少了碎片化现象，提升了内存使用效率。

3. **相比原先的 `kmem` 分配器**

    - 效率：添加伙伴分配器后，内存分配和释放的效率更高，尤其是在多次分配和释放内存的场景下，减少了由于内存碎片化带来的性能损失。
    - 内存的更好利用：通过合并和拆分内存块，Buddy 分配器有效减少了无法利用的小内存块，系统的内存池得到了更高效的利用。
    - 碎片化情况减少：在加了 Buddy 分配器后，内存碎片减少，尤其是在长期运行过程中，内存池的碎片化情况更加可控。

4. **总结**

    - 伙伴分配算法通过拆分和合并内存块，减少了内存碎片化，提高了内存分配和释放的效率，尤其在长时间运行的系统中，能够显著改善内存使用情况。
    - `kmem` 分配器使用简单的链表结构来管理内存，虽然实现上较为简单，但它没有内存合并机制，容易产生碎片化，尤其是在大量内存分配和释放时，可能导致内存池中的空闲块无法被有效利用。
    - 使用 伙伴分配器后，内存碎片化显著减少，内存利用效率更高，系统的性能得到了提升。

因此，我们得以验证伙伴分配器优化了内存的碎片化，提升了内存的使用效率，并减少了内存管理的开销。

五、其他工作内容

1. **系统调用的建立**

   由于测试文件处于用户态下，不能直接访问内核态的代码和数据，因此需要建立系统调用，修改了`sysproc.c`和`syscall.c`等文件，使得用户能通过系统调用向内核申请分配内存（`buddy_alloc`）并释放分配的内存块（`buddy_free`）。

2. **测试文件撰写**

   - `buddytest` **测试：**针对伙伴分配器基础功能的验证测试，包括：
     - 分配内存： 调用 `buddy_malloc` 分配多个内存块，检查分配是否成功，并打印内存地址。
     - 释放内存： 释放之前分配的内存块，并打印释放的信息。
     - 通过分配和释放内存块，确保 Buddy 分配器的基本功能（分配、释放、合并）正常，确保分配和释放的内存块按逻辑工作，不会出现资源泄漏或分配失败。
   - `optimaltest` **测试：**针对内存分配优化的测试，用来检测系统伙伴分配器的性能是否有所提升。
     - 通过 `fork` 创建子进程，调用 `sbrk` 不断分配内存。
     - 子进程分配内存后，向父进程通过管道发送一个字节，父进程通过接收管道中的数据来计数，统计内存分配的页数。
     - 检查分配的页数是否达到设定的下限（31950 页，约 125MB 内存）。

六、材料说明

我们在仓库中建立了四个分支，包含了实验的完整工程文件以及过程性文件：

- `buddy-text`分支：实验题目文件。
- `buddy-answer`分支：实验完整答案文件。
- `print-without-buddy`分支：对于`xv6`内存分配的输出验证性文件。
- `print-with-buddy`分支：对于添加伙伴分配器以后的输出验证性文件。