物理内存管理

概述

目标

1. 掌握内存管理相关的概念
2. 了解物理内存的管理方法
3. 掌握 First Fit，Best Fit 等分配算法
4. 掌握伙伴内存分配算法
5. 实现用户态 malloc

简述

物理内存管理在操作系统中扮演着重要的角色。它负责管理计算机系统中的物理内存资源，包括分配、回收和映射等操作，以满足进程的内存需求并确保系统的正常运行。

物理内存管理的主要任务包括：

1. 分配内存：操作系统需要根据进程的内存需求，从系统的物理内存池中分配合适的内存空间给进程使用。分配的内存需要满足进程的大小和地址对齐等要求。
2. 回收内存：当进程终止或释放内存时，操作系统需要将已使用的内存空间回收并标记为可重新分配的状态，以便其他进程可以再次使用。
3. 内存映射：物理内存管理需要将进程的虚拟内存地址映射到实际的物理内存地址。这涉及到页表的管理和地址转换，以实现虚拟内存和物理内存之间的映射关系。
4. 内存保护：操作系统需要为每个进程提供独立的内存空间，并通过权限控制机制保护进程之间的内存互不干扰。这可以防止进程越界访问或者非法修改其他进程的内存数据。

物理内存管理需要考虑到内存的碎片化问题，合理管理内存资源，以提高内存利用率和系统性能。

在操作系统实现中，物理内存管理通常涉及到页表的设计和管理、内存分配算法的选择、地址转换的实现等。这些工作对于操作系统的性能和稳定性具有重要影响。

总之，物理内存管理在操作系统中起着至关重要的作用，它负责分配、回收和映射物理内存，以满足进程的内存需求，并保证系统的正常运行。本部分共安排了三个实验，分别是将 first fit 内存分配改为 best fit，实现 buddy 算法，实现用户态字节内存管理 umalloc。

讲解

XV6的物理内存分配

在 xv6 操作系统中，物理内存分布是通过内核进行管理和分配的。其中，由内核管理分配的内存区域被称为"自由内存"，它的范围从 `end` 到 `PHYSTOP`。

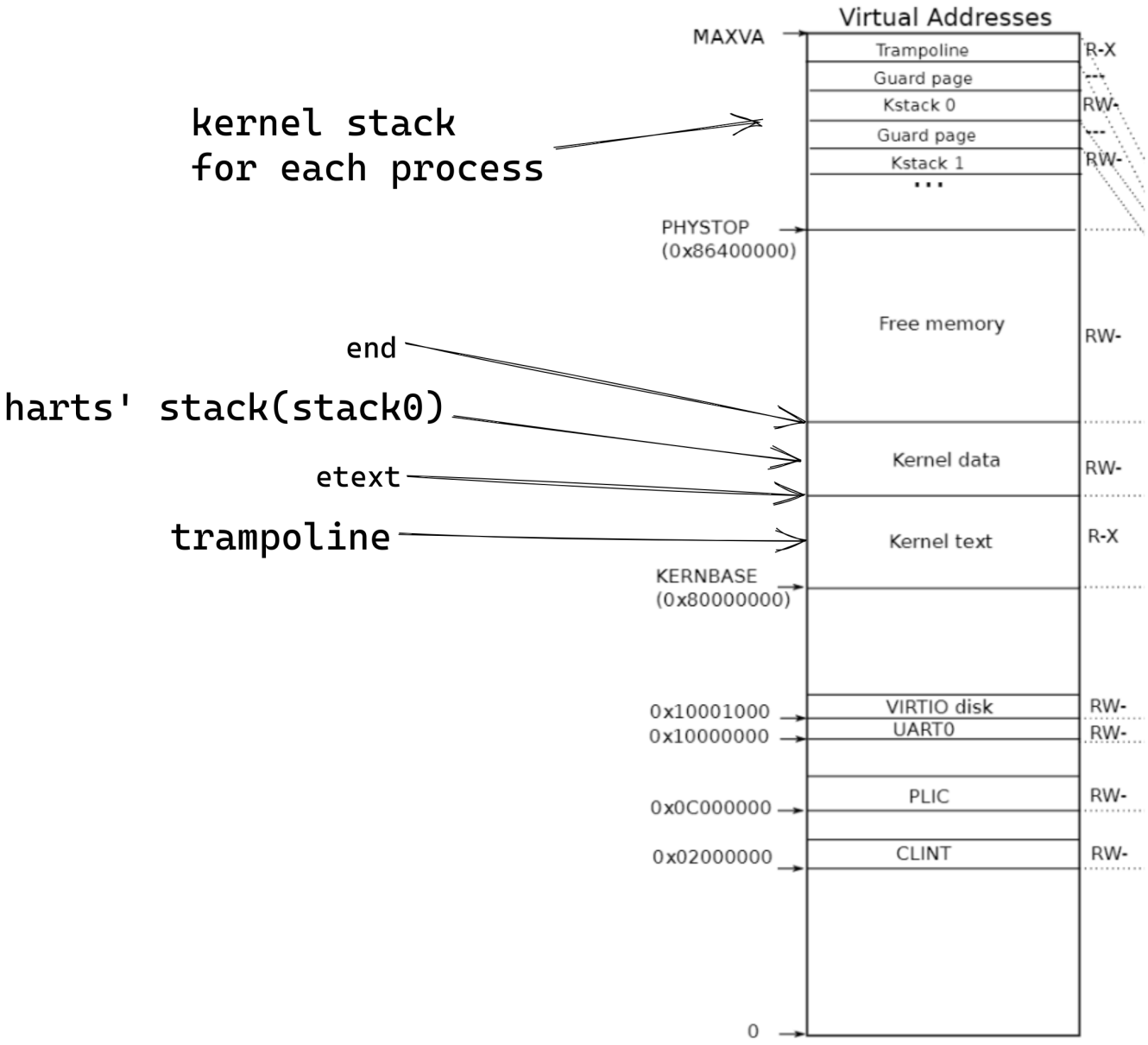
"自由内存"是指操作系统内核可以自由分配和管理的内存空间。在 xv6 中，`end` 是内核代码和数据的结束位置，而 `PHYSTOP` 是系统物理内存的顶部地址。因此，从 `end` 到 `PHYSTOP` 之间的内存区域被视为可供内核分配的自由内存。

内核可以使用自由内存来动态分配给进程、内核数据结构和缓冲区等需要内存的组件。这些分配的内存可以用于存储进程的代码、堆栈、全局变量以及内核的数据结构，如进程控制块、文件描述符表和缓冲区等。

通过合理管理自由内存，内核可以根据系统的需求来动态分配和回收内存空间，以满足不同进程和系统组件的内存需求。这有助于提高系统的性能和资源利用率，并确保操作系统的稳定运行。

需要注意的是，自由内存是由内核管理的，因此用户进程无法直接访问或操作自由内存区域。用户进程只能通过系统调用和内核接口来请求内存分配，并由内核在自由内存区域中进行分配和管理。

xv6 操作系统中的物理内存分布中，由内核管理分配的自由内存范围从 `end` 到 `PHYSTOP`，内核可以根据系统需求动态分配和管理这段内存区域，以满足进程和系统组件的内存需求。



内核必须在运行时为页表、用户内存、内核栈和管道缓冲区分配和释放物理内存。

xv6使用内核末尾到 `PHYSTOP` 之间的物理内存进行运行时分配。它一次分配和释放整个4096字节的页面。

它使用链表的数据结构将空闲页面记录下来。分配时需要从链表中删除页面；释放时需要将释放的页面添加到链表中。

XV6共拥有128MB的内存，内存地址从 `0x80000000` 开始，到 `0x88000000` 结束。

- `0x80000000`-`0x80021d60` ，内核程序本身占用这段内存
- `0x80021d60`-`0x88000000` ，Free memory 由内核管理（kalloc,kfree），以按需分配
  - `0x80021d60` - `0x80022000` ，没有使用，原因 `PGROUNDUP(0x80021d60) == 0x80022000`
  - `0x80022000` - `0x88000000` ，真正被管理的内存空间，32734个4KB的页面。

XV6分配，释放内存都以4KB为单位，`kalloc(void)` 是不要参数的。因此内存都要对4K对齐。代码如下。

```c
#define PGSIZE 4096 // bytes per page

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

使用 `uint16_t` 举例说明

```
PGSIZE      0b0001000000000000
PGSIZE-1    0b0000111111111111
~(PGSIZE-1) 0b1111000000000000
-------------------------------
a           0b0010101001010101
a+PGSIZE-1  0b0011101001010100
RU(a)       0b0011000000000000
RD(a)       0b0010000000000000
```

XV6使用free-list链表管理内存，经过初始化，链表里有32734个页面。`kalloc`时取出链表头那个，`kfree`再插入链表头。

因为分配都是固定大小，因此也无所谓什么firsrt fit,best fit等等

所有hart都使用同一个free-list管理内存，因此需要一把大锁来构建临界区。

内核链表

代码位于`kernel/list.h`（请先切换到分支`lab6-1-bestfit`）

根据 linux kernel list ,自己实现的侵入式双向链表,实现了大部分功能，内核/通用链表其实就是一个有头节点的双向循环链表，侵入式的设计使得链表可以把不同数据类型串起来,突破了传统链表仅可以储存相同的数据类型的限制。

代码中定义了一个`list_entry`结构体，它包含了两个指针成员`prev`和`succ`，分别表示前一个节点和后一个节点。

代码中使用了宏定义来实现一些常用的操作和遍历方式：

- `offsetof(type, name)`宏用于计算结构体中成员`name`的地址偏移量。
- `container_of(ptr, type, name)`宏用于根据成员指针`ptr`获取所在结构体的起始地址。
- `listEntry(ptr, type, name)`宏是`container_of`宏的一个别名，用于更方便地获取结构体的起始地址。
- `traverse`宏用于遍历整个链表，`p`是一个循环游标，`head`是链表的头节点。
- `traverseReverse`宏用于反向遍历整个链表。
- `traverseFromEntry`宏用于从某个节点开始遍历剩余的链表，直到链表的头节点。
- `traverseReverseFromEntry`宏用于从某个节点开始反向遍历剩余的链表，直到链表的头节点。

除了宏定义之外，还提供了一些函数和操作：

- `list_entry_init`函数用于初始化链表节点，将节点的`prev`和`succ`指针都指向节点本身，形成一个空的循环链表。
- `isLast`函数用于判断一个节点是否是链表头节点的最后一个节点。
- `pushFront`函数用于将一个节点插入到链表头节点之后，即作为链表的第一个元素。
- `pushBack`函数用于将一个节点插入到链表头节点之前，即作为链表的最后一个元素。
- `delEntry`函数用于从链表中删除一个节点，将节点的`prev`和`succ`指针重新连接，同时将节点的`prev`和`succ`指针置为空。

其中，比较有趣的是

```c
// struct type中成员name的地址偏移量
#define offsetof(type, name) ((size_t) & ((type *)0)->name)

/**
 * container_of - cast a member of a structure out to the containing structure
 * @ptr:        the pointer to the member.
 * @type:       the type of the container struct this is embedded in.
 * @member:     the name of the member within the struct.
 * 得到ptr所指向的结构体的起始地址
 * typeof为gnu拓展,省去仅少了类型检查,不影响正常实现
 */
#define container_of(ptr, type, name)                                   \
  (type *)((char *)ptr - offsetof(type, name))
```

以上代码定义了两个宏：`offsetof` 和 `container_of`。

`offsetof(type, name)` 宏用于计算结构体 `type` 中成员 `name` 的地址偏移量。它通过将空指针强制转换为 `type` 类型的指针，然后获取成员 `name` 的地址，并将其转换为 `size_t` 类型，从而得到地址偏移量。`container_of(ptr, type, name)` 宏用于将结构体 `type` 中的某个成员 `name` 的指针 `ptr` 转换为包含该成员的结构体的指针。它通过使用 `offsetof` 宏获取成员 `name` 在结构体中的地址偏移量，然后将指针 `ptr` 减去该偏移量，得到包含该成员的结构体的起始地址。这样，通过 `container_of` 宏，我们可以从一个结构体的成员指针获取整个结构体的指针，方便进行操作和访问结构体的其他成员。

First-fit

First-fit（首次适应）分配算法是一种常用的内存分配算法，用于管理操作系统中的内存分配。该算法的核心思想是在空闲内存块列表中查找第一个满足需求的空闲块进行分配。

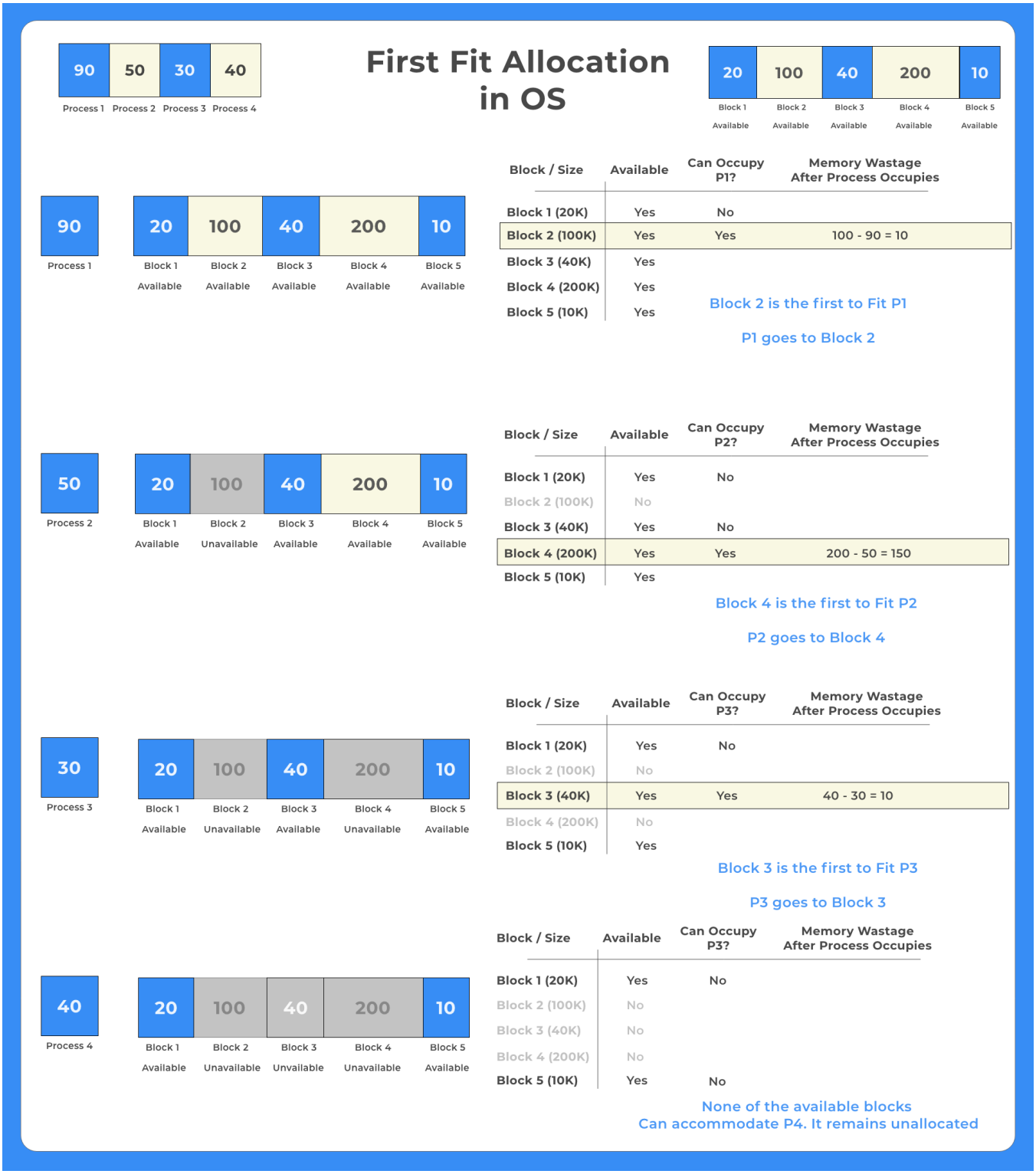
具体工作流程如下：

1. 当一个进程请求内存分配时，系统会遍历空闲内存块列表，从列表的开头开始查找。
2. 对于每个空闲块，系统检查其大小是否足够满足进程的需求。
3. 如果找到了一个足够大的空闲块，系统将该块划分为两部分：一部分用于满足进程的内存需求，另一部分则成为新的空闲块。
4. 分配给进程的内存块被标记为已用状态，并返回给进程使用。
5. 如果没有找到足够大的空闲块，则继续遍历下一个空闲块，直到找到满足需求的内存块或遍历完整个空闲块列表。
6. 如果整个空闲内存块列表都被遍历完，仍然没有找到足够大的空闲块，则系统可能需要进行内存碎片整理或者向操作系统请求更多的内存空间。

First-fit 分配算法的优点是简单且高效。它能够快速找到满足进程需求的第一个空闲块，减少了搜索的时间开销。此外，由于分配的是第一个满足需求的空闲块，所以分配的内存地址相对较低，有利于减少外部碎片的产生。

然而，First-fit 分配算法也存在一些缺点。由于该算法倾向于使用较早出现的空闲块，可能会导致大的空闲块被分割为较小的块，从而增加了外部碎片的数量。此外，如果某个空闲块被频繁地分配和释放，可能会导致空闲块的碎片化，降低了内存利用率。

First-fit 分配算法是一种简单而常用的内存分配算法，通过查找第一个满足需求的空闲块进行分配。它具有高效的搜索速度，但可能导致外部碎片的增加。在实际应用中，需要权衡算法的优缺点，并根据具体系统的特点选择合适的内存分配策略。



在本实验的实现中，已经实现了 First-fit 算法，其具体实现如下。

空闲链表的定义如下，`start` 是空闲块的物理起始地址，`size` 是该空闲块的大小，以 page 为单位，因为 XV6 的内核物理内存管理，分配是以 page 为最小单位的，如此简化了实现。

```c
struct free_chunk {
  void *start;         // start address of free chunk
  size_t size_in_page; // size of free chunk in page
  list_entry entry;    // list entry
};
```

以上代码是一个用于管理物理内存分配的物理内存分配器。它使用了基于页的分配策略，每个分配单位是一个大小为4096字节的页面。

代码中使用了双向链表来维护空闲内存块的列表。每个空闲内存块由结构体`free_chunk`表示，包含了起始地址、页面数量和链表节点。`kmem`结构体用于存储内存分配器的状态，其中包含了一个自旋锁和空闲内存块列表。

在`kinit`函数中，首先初始化了自旋锁，并计算了可用的物理内存范围。然后，将整个可用的物理内存范围作为一个空闲内存块加入到空闲内存块列表中。`kalloc`函数用于分配一个页面大小的物理内存块。它调用了`free_chunk_alloc`函数从空闲内存块列表中寻找合适的内存块。如果成功找到一个合适的内存块，就将其标记为已使用，并返回分配的内存块的起始地址。为了方便调试，分配的内存块还被填充为固定的值。
`kfree`函数用于释放一个页面大小的物理内存块。它首先检查传入的内存块是否有效，并且位于合法的物理内存范围内。然后，将内存块填充为垃圾值以防止悬空引用，并调用`kfreen`函数将内存块标记为未使用状态。`free_chunk_alloc`函数是物理内存分配的核心部分，它遍历空闲内存块列表，查找第一个能够满足需求的内存块。如果找到了合适的内存块，就将其从空闲内存块列表中删除，并返回分配的内存块的起始地址。如果没有找到合适的内存块，就返回NULL。`free_chunk_free`函数用于释放一定数量的物理内存块，并将其加入到空闲内存块列表中。它会尝试合并相邻的空闲内存块，以减少内存碎片的产生。

在初始化阶段（`kinit`函数），物理内存的起始和结束地址被计算，并且将整个可用的物理内存范围作为一个空闲内存块加入到空闲内存块列表中。这个列表被存储在`kmem`结构体的`free_chunk_list`成员中。

分配内存的操作（`kallocn`函数）首先会获取`kmem.lock`自旋锁，以确保同一时间只有一个线程访问内存分配器。然后，调用`free_chunk_alloc`函数从空闲内存块列表中查找一个能够满足需求的内存块。如果找到了合适的内存块，它会将该内存块从空闲内存块列表中移除，并返回分配的内存块的起始地址。如果没有找到合适的内存块，则返回NULL。

释放内存的操作（`kfreen`函数）首先会获取`kmem.lock`自旋锁。然后，调用`free_chunk_free`函数将释放的内存块加入到空闲内存块列表中。在此过程中，它会尝试合并相邻的空闲内存块，以减少内存碎片的产生。

代码中使用了双向链表的数据结构来管理空闲内存块列表。每个空闲内存块被表示为一个`free_chunk`结构体，包含了起始地址、页面数量和链表节点。通过使用链表，可以方便地插入、删除和遍历空闲内存块。

为了确保线程安全性，使用了自旋锁`kmem.lock`来保护对内存分配器的并发访问。在分配和释放内存时，需要先获取锁，以防止多个线程同时修改内存分配状态。

Best-fit

Best-fit（最佳适应）分配算法是一种常用的内存分配算法，用于管理操作系统中的内存分配。该算法的核心思想是在空闲内存块列表中查找最小的能够满足需求的空闲块进行分配。

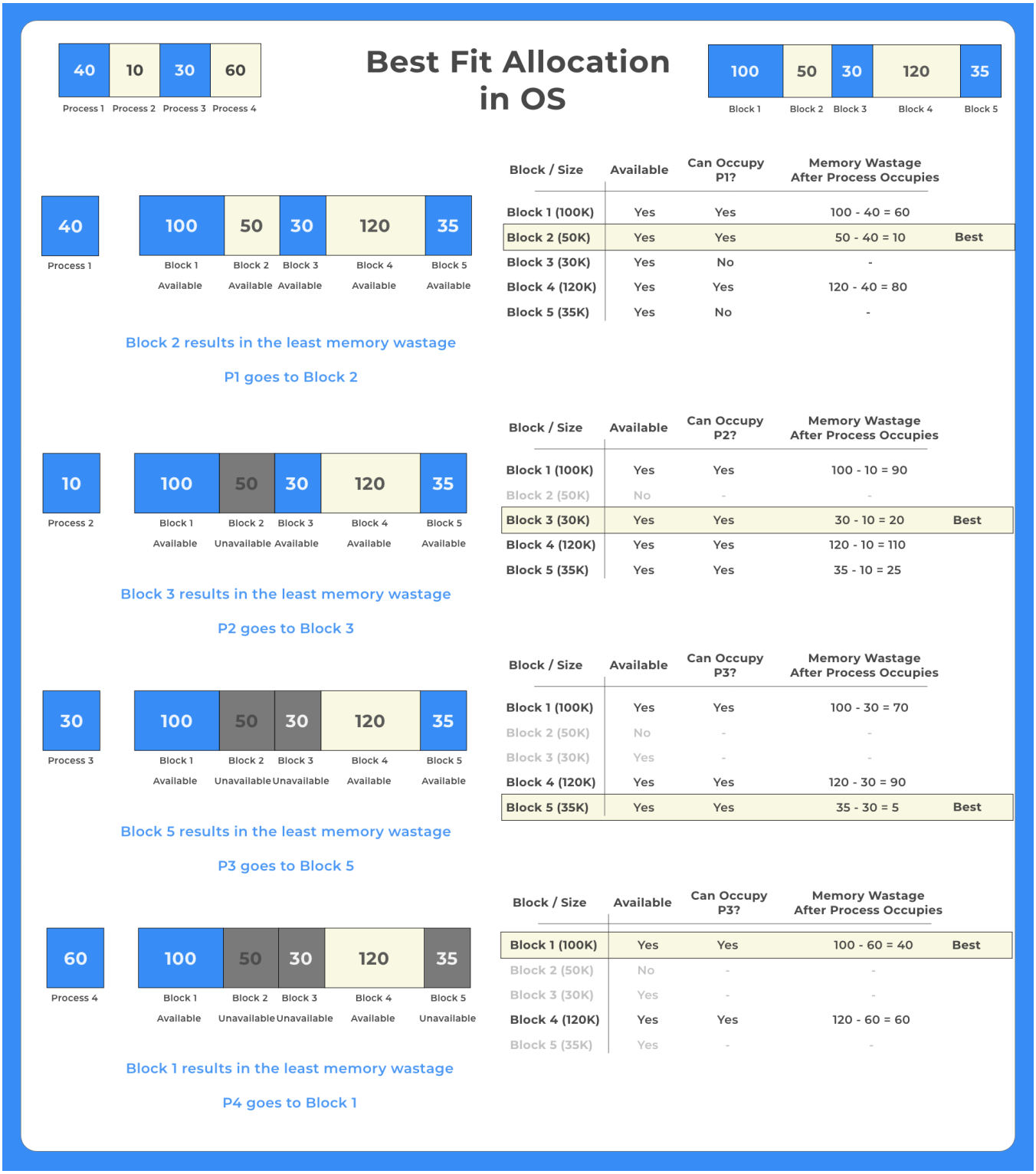
具体工作流程如下：

1. 当一个进程请求内存分配时，系统会遍历空闲内存块列表，寻找大小最接近进程需求的空闲块。
2. 对于每个空闲块，系统检查其大小是否足够满足进程的需求。
3. 如果找到了一个足够大的空闲块，并且它的大小比已找到的空闲块更接近进程需求的大小，则将该块作为最佳适应块。
4. 如果遍历完整个空闲块列表后找到了最佳适应块，则将该块划分为两部分：一部分用于满足进程的内存需求，另一部分则成为新的空闲块。
5. 分配给进程的内存块被标记为已用状态，并返回给进程使用。
6. 如果整个空闲内存块列表都被遍历完，仍然没有找到足够大的空闲块，则系统可能需要进行内存碎片整理或者向操作系统请求更多的内存空间。

Best-fit 分配算法的优点是可以更好地利用内存空间，减少了外部碎片的产生。它选择最接近进程需求大小的空闲块，使得分配的内存块相对较小，有利于减少碎片化现象。

然而，Best-fit 分配算法也存在一些缺点。由于该算法需要遍历整个空闲块列表来找到最佳适应块，因此搜索的时间开销较大。此外，频繁的分配和释放可能会导致空闲块的碎片化，降低了内存利用率。

Best-fit 分配算法是一种常用的内存分配算法，通过寻找最接近进程需求大小的空闲块进行分配。它具有较好的内存利用率，但搜索时间开销较大。在实际应用中，需要综合考虑算法的优缺点，并根据具体系统的需求选择适合的内存分配策略。



实验

实验介绍

实验内容 1 - bestfit

```shell
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab6-1-bestfit    #切换到本实验的分支
```

任务：

- 本部分已经默认实现了 First-fit 内存分配算法，请改写为 Best-fit 算法。

Tips：

- 可以维护空闲链表有序，按照剩余内存升序排序，如此分配时从前向后遍历，第一个符合条件的块即为 best-fit。

实验内容 2 - buddy

```shell
> git add .
> git commit -m 'message'   #暂存修改
> git switch lab6-2-buddy   #切换到本实验的分支
```

任务：

- 请实现 buddy 算法，以进行物理内存管理。

Tips：

- Buddy 算法将可用内存空间划分为大小为2的幂次方的块，这些块被组织成一棵二叉树，称为 Buddy 系统或 Buddy 树。树的根节点表示整个可用内存空间，而每个内部节点都表示一个内存块，其子节点表示该块被分割后的两个更小的子块，叶子节点表示最小的可分配内存块（通常是页面大小）。
- 当需要分配内存时，Buddy 算法从根节点开始，在 Buddy 树中找到满足要求的大小的最小空闲块。如果找到了合适大小的空闲块，则将其标记为已分配，并根据需要将该块进一步分割为更小的块，直到满足所需大小的要求。
- Buddy 树的每一层可以使用一个链表，每个链表就代表着固定大小的块的集合。
- 当分配时，从当前层的链表中取出一块，分成两块加入到下一层的链表中；当合并时，反之。

实验内容 3 - umalloc

```shell
> git add .
> git commit -m 'message'   #暂存修改
> git switch lab6-3-umalloc   #切换到本实验的分支
```

任务：

- 请在用户态实现 `umalloc` 和 `ufree` 函数，其作用与我们熟知的 `malloc` 和 `free` 一致。
- XV6 内核都是以页为单位分配，`umalloc` 要求以字节为单位。
- 用户进程使用 `sbrk` 系统调用增大堆区，你需要对堆区进行管理。
- 不必考虑字节对齐问题，不必考虑多线程，不必考虑将内存还回操作系统。

Tips：

- `ufree` 时参数只有一个指针，是不知道分裂了多少内存的。因此，可以在分配内存的前边加一个头块，里面包含一些必要的信息。
- https://moss.cs.iit.edu/cs351/slides/slides-malloc.pdf

拓展实验内容

> 上文实验 3 实现了基本的 `umalloc`，请你完善实现，以实现分配给用户的内存对齐到 64 字节，支持多线程分配。

> 更进一步，你可以考虑实现一个简单的内存池。