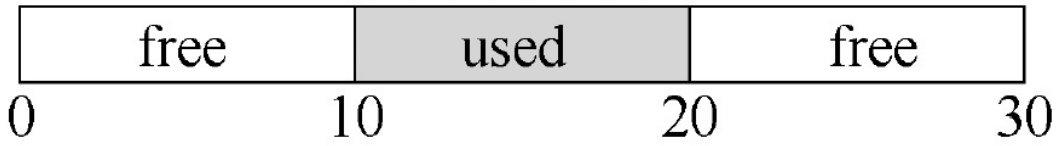
空闲空间管理

邰颖
南京大学
智能科学与技术学院

空闲空间管理（Free-Space Management）

- 必须管理空闲空间以避免碎片化（内部碎片和外部碎片）。
- 如果空闲空间是固定大小（如分页方式），管理较为简单。
- 如果空闲空间大小不固定（如分段方式），管理较为困难：
  - 分段机制会受到**外部碎片**的影响。
  - 请考虑以下内存配置，若请求 15 字节的空间，该请求能否被满足？



假设（Assumptions）

1. 关注**用户级**的内存分配库。

2. 接口遵循 malloc() 和 free()：

```
• void *malloc(size t size)
• void free(void *ptr)
```

3. 关注**解决外部碎片**问题

4. 一旦空闲列表中的内存分配给请求者/客户端，就**不能重新定位到其他位置**：
   - **不进行内存压缩**（Compaction）。

5. **内存管理器管理的内存是固定大小且连续的**。

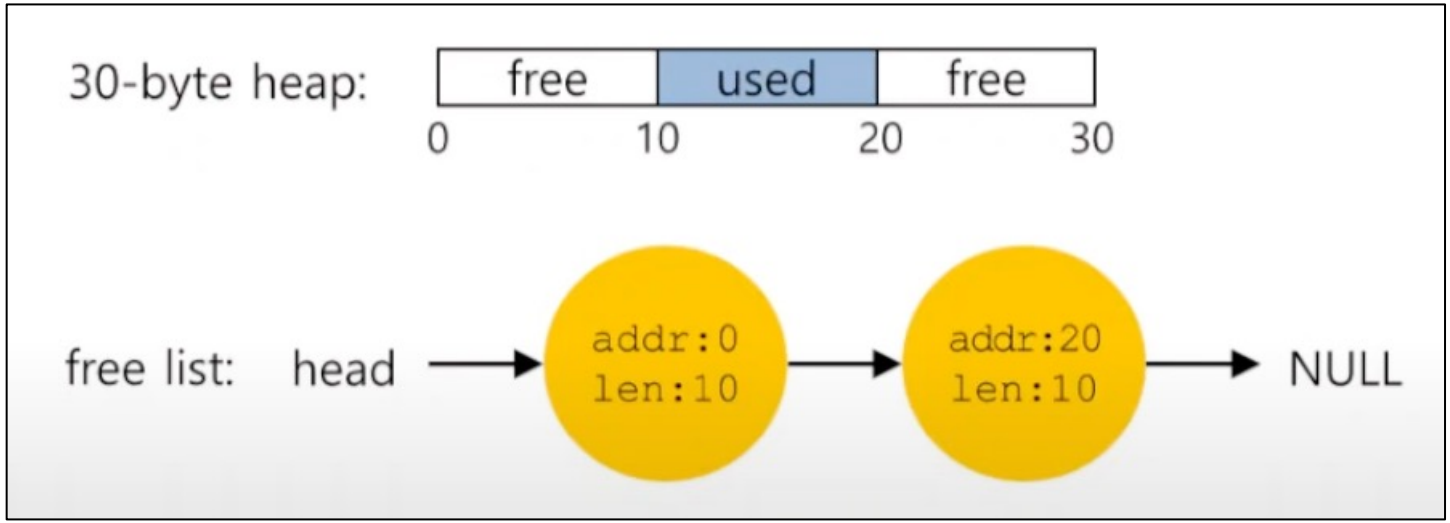
底层机制（Low-level Mechanisms）

1. 分割（Splitting）与合并（Coalescing）
2. 跟踪已分配区域的大小
3. 在空闲空间内构建一个简单的列表，以跟踪空闲空间

分割（Splitting）

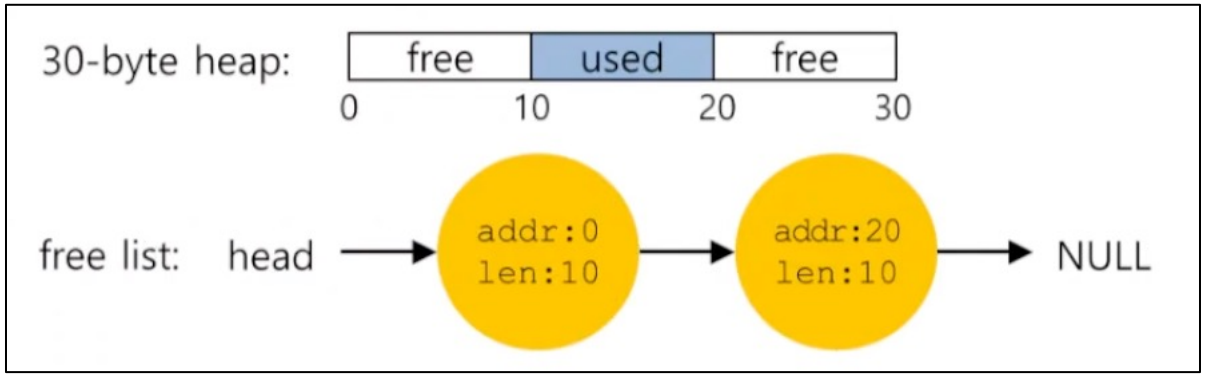
•找到一个可以满足请求的空闲内存块，并将其拆分为两个部分：

- 当分配请求的大小（如1字节）小于现有的单个空闲块时，可拆分。

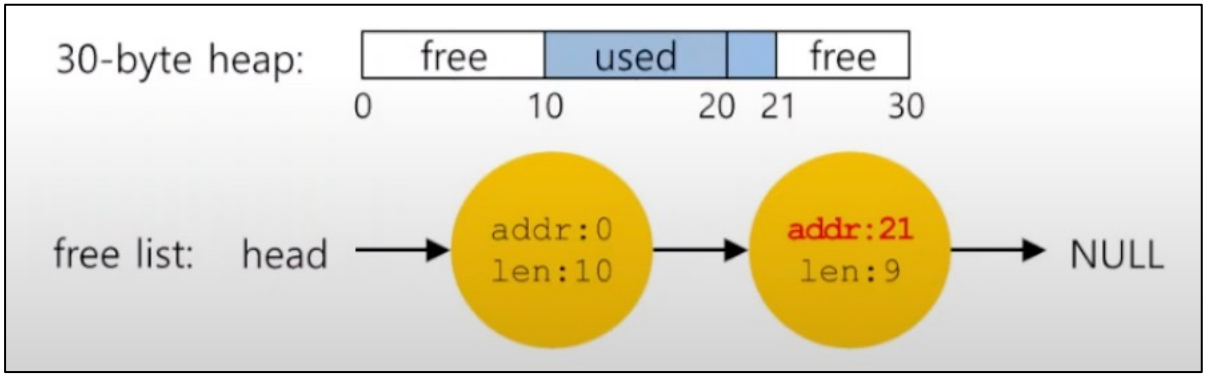


分割（续）

- 两个 10 字节的空闲块，处理 1 字节的分配请求

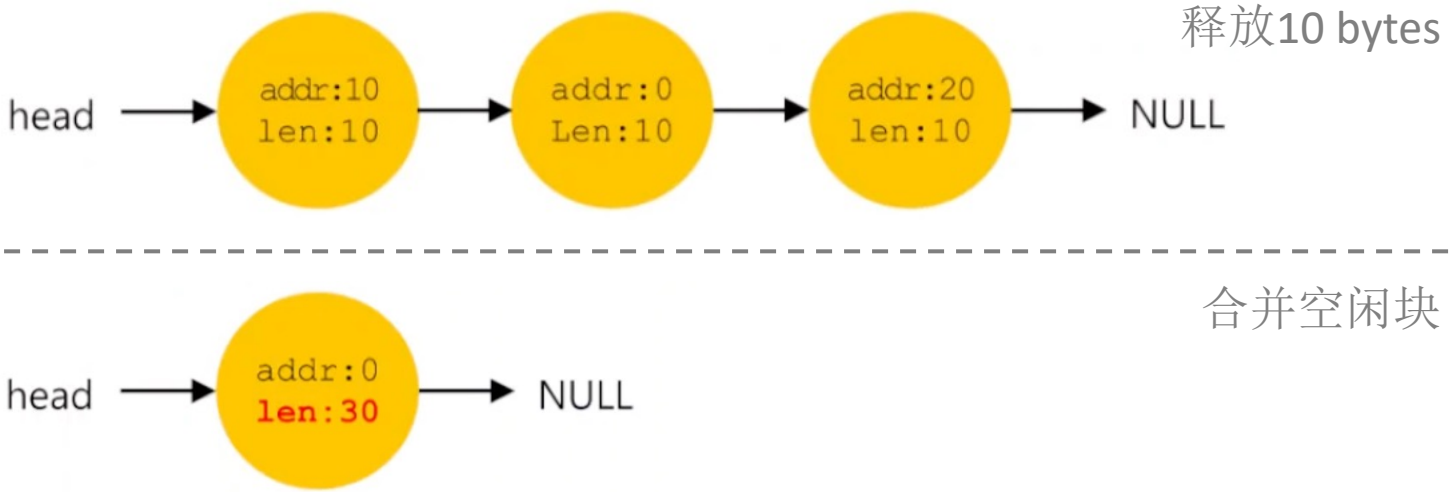


分割 1字节的空闲块



合并（Coalescing）

- 如果用户请求的内存**大于任何单个空闲块的大小**，那么在空闲链表中**找不到合适的块**。
- **合并（Coalescing）**：当多个空闲块的**地址相邻**时，可以合并成**一个更大的空闲块**。



跟踪已分配区域的大小

- free(void *ptr) 接口不包含大小参数
  - 那么库是如何知道需要释放的内存块大小的？

- 大多数分配器（allocator）会在**头部块（header block）**中存储额外信息。

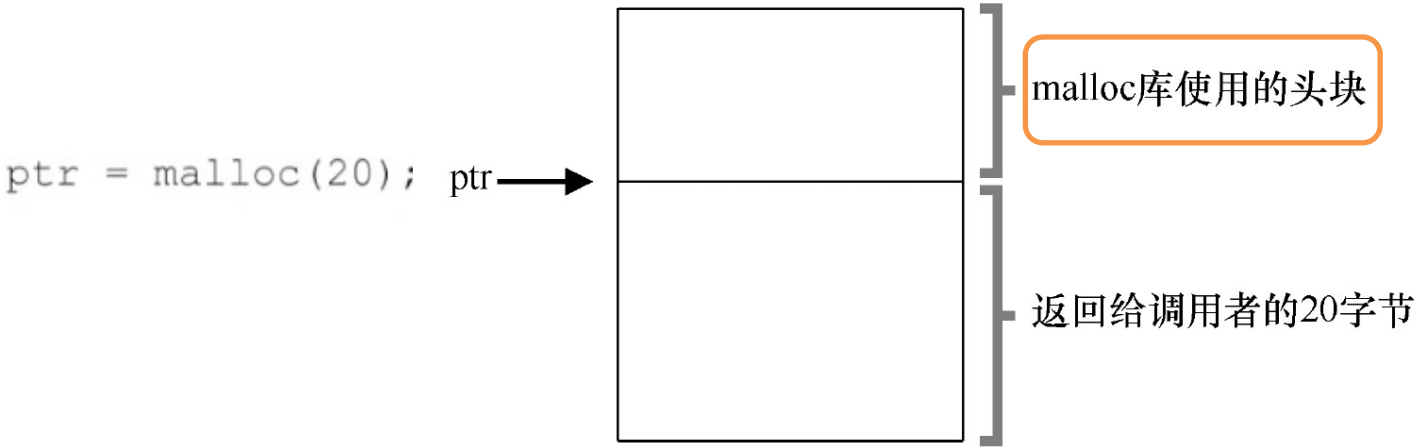
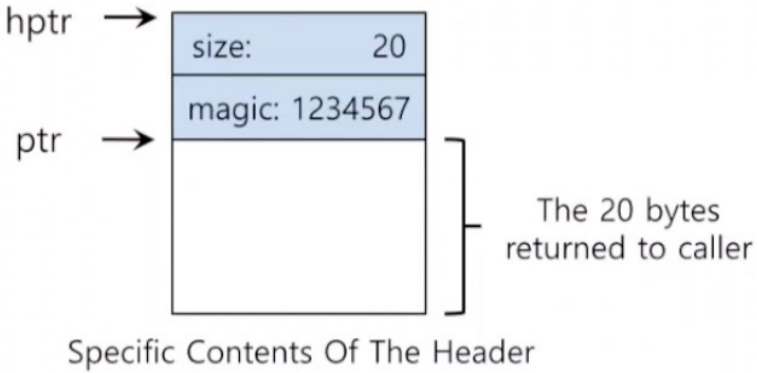


图 17.1　一个已分配的区域加上头块

已分配内存块的头部

- **头部至少包含**已分配内存区域的**大小（size）**。

- 头部还可能包含：
  - **魔数/幻数（magic number）**，用于完整性检查（integrity checking）。
  - 额外的指针，用于**加速释放（de-allocation）**。



Specific Contents Of The Header

```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

A Simple Header

已分配内存块的头部（续）

- 使用简单的指针运算来找到头部指针：

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

- 释放的空闲区域大小 = 头部大小 + 分配给用户的空间大小

  - 如果用户请求 N 字节，内存管理器会搜索大小至少为 *N + 头部大小* 的空闲块。

嵌入空闲列表（Embedding A Free List）

•如何在空闲内存内部建立空闲列表？

•内存分配库在初始化堆时，会将空闲列表的第一个元素存放在空闲空间中。

- 内存分配库不能使用 **malloc()** 在自身内部构建列表。

嵌入空闲列表（续）

- 列表节点的描述（Description of a node of the list）

```
typedef struct __node_t {
    int size;
    struct __node_t *next;
} node_t;
```

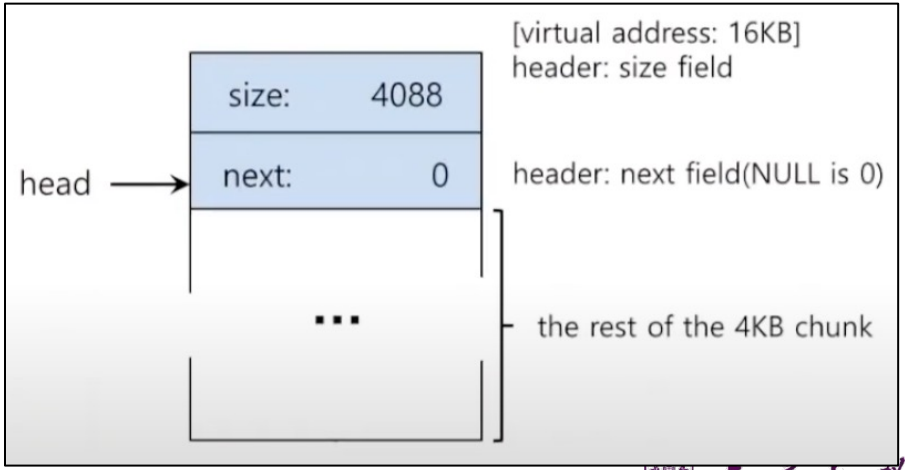
- **构建堆并初始化空闲列表（Building heap and putting a free list）**
  - 假设堆是通过 mmap() 系统调用创建

```
// mmap() 返回指向一块空闲空间的指针
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);

head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

包含一个空闲块的堆（A Heap With One Free Chunk）

- head 指向空闲块的头部
- **头部字段：** size: 4088（整个 4KB 块减去头部大小）
- next: NULL，表示这是唯一的空闲块
- **虚拟地址：** 16KB（示例地址）
- **其余部分：** 可供分配的 4088 字节

```
// mmap() 返回指向空闲内存块的指针
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);

head->size = 4096 - sizeof(node_t);
head->next = NULL;
```



嵌入空闲列表：内存分配

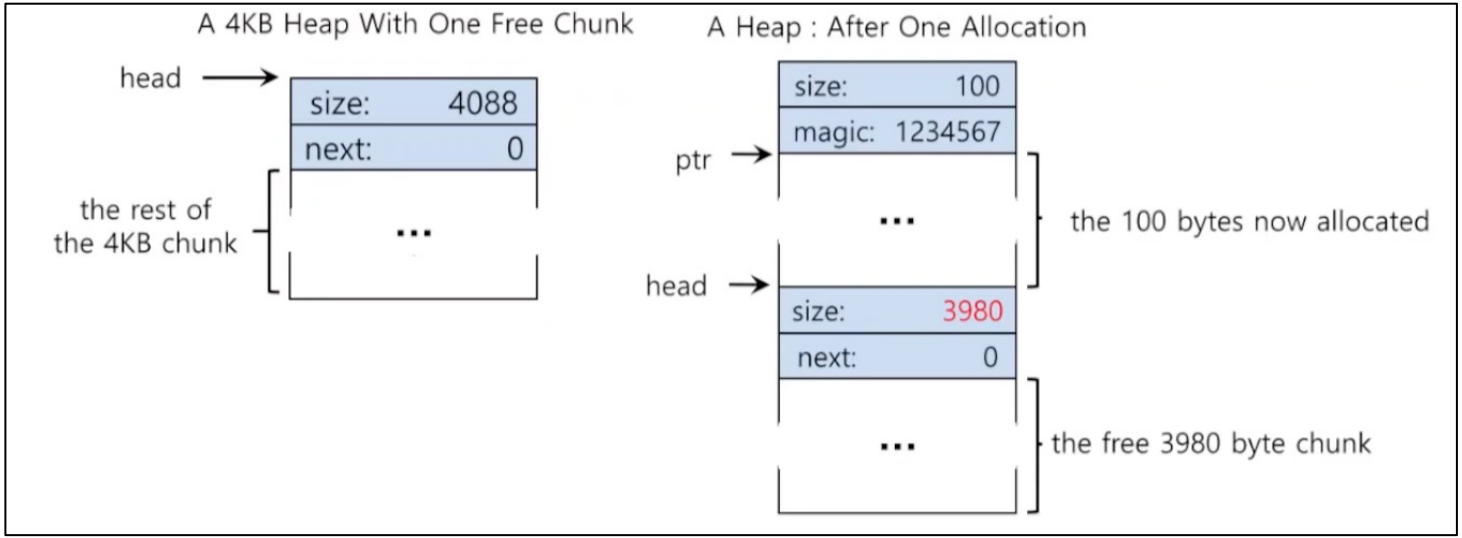
- 当请求一块内存时，库会**首先找到**一个**足够大的**空闲块来满足请求。
- 内存分配库的操作：
  - **分割（Split）** 大的空闲块为两个部分：
    - **一部分用于满足请求**
    - **剩余部分仍作为空闲块**
- **缩小（Shrink）** 空闲链表中该块的大小。

嵌入空闲列表：内存分配（续）

- 示例：请求 100 字节的内存

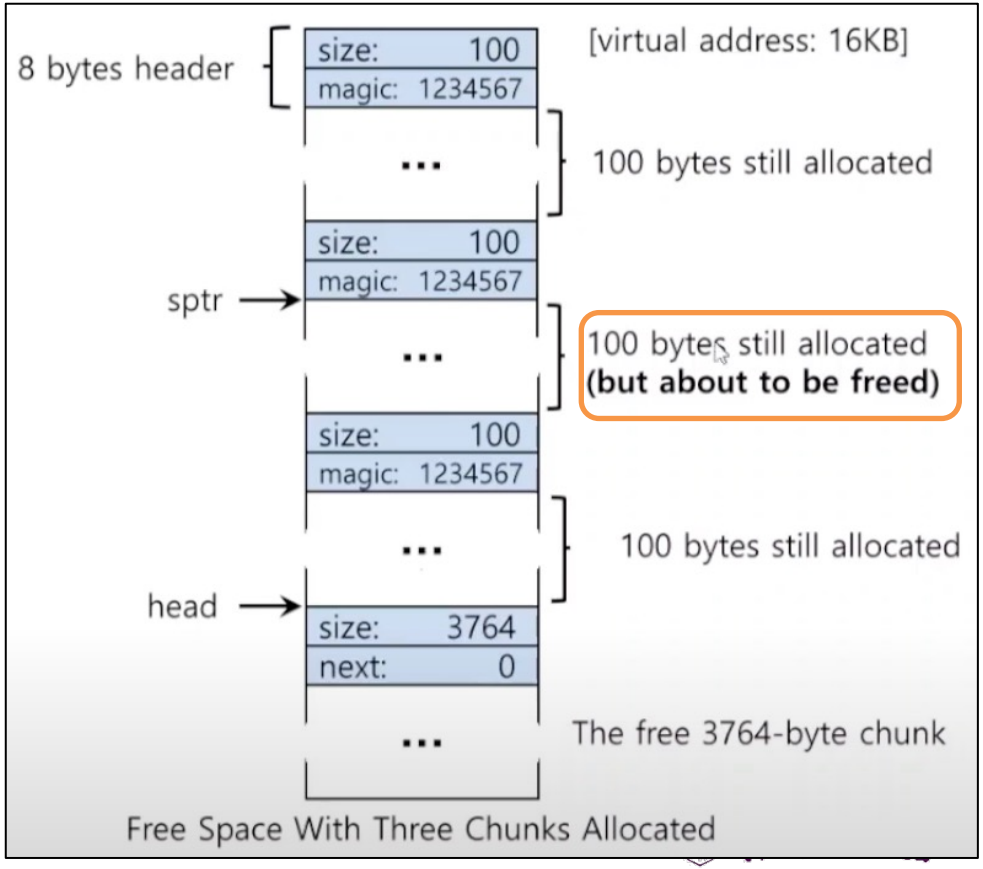
```
ptr = malloc(100);
```

  - 从现有的空闲块中分配 108 字节（其中 8 字节用于头部）
  - 将剩余的空闲块缩小至 3980 字节（4088 - 108）



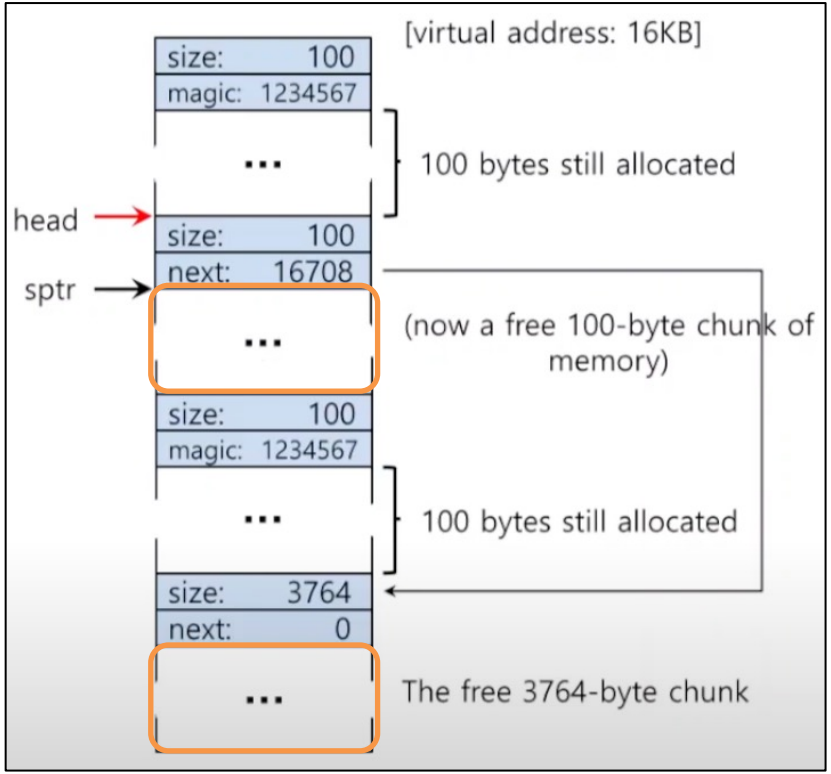
已分配块的空闲空间

- 每个已分配块包含：
  - **8** 字节头部（存储大小、魔数）。
  - **100** 字节数据区。
- 虚拟地址：16KB。
- 分配情况：
  - 三个 **100** 字节块已分配（带有 magic: 1234567）。
  - 第二个块即将被释放（标记 about to be freed）。
- 剩余空闲空间：
  - **3764** 字节的空闲块（由 head 指向，next = 0）。



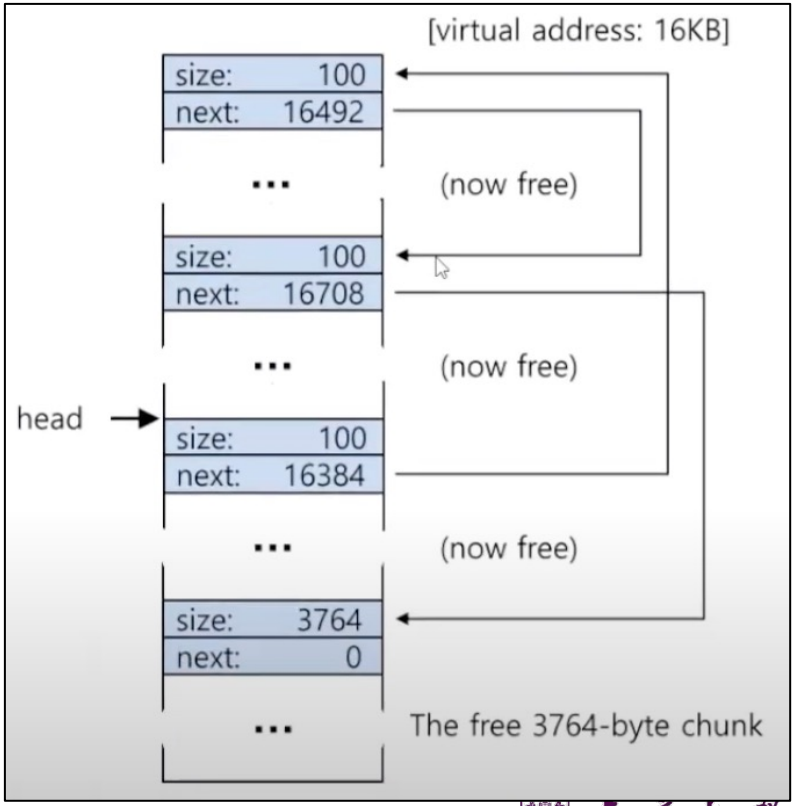
调用 free() 释放空间

- 例子：free(sptr);
  - 等价于 free(16500)
  - 计算方式: 16384 + 108 + 8
- 100 字节的块**回到空闲列表**
- **空闲列表被分割成两块：**
  - 头指针 head 现在指向这个 刚被释放的100 字节的小空闲块。
  - Head->next指向一个大空闲块（3764字节），地址为16708 = 16500 + 100 + 108



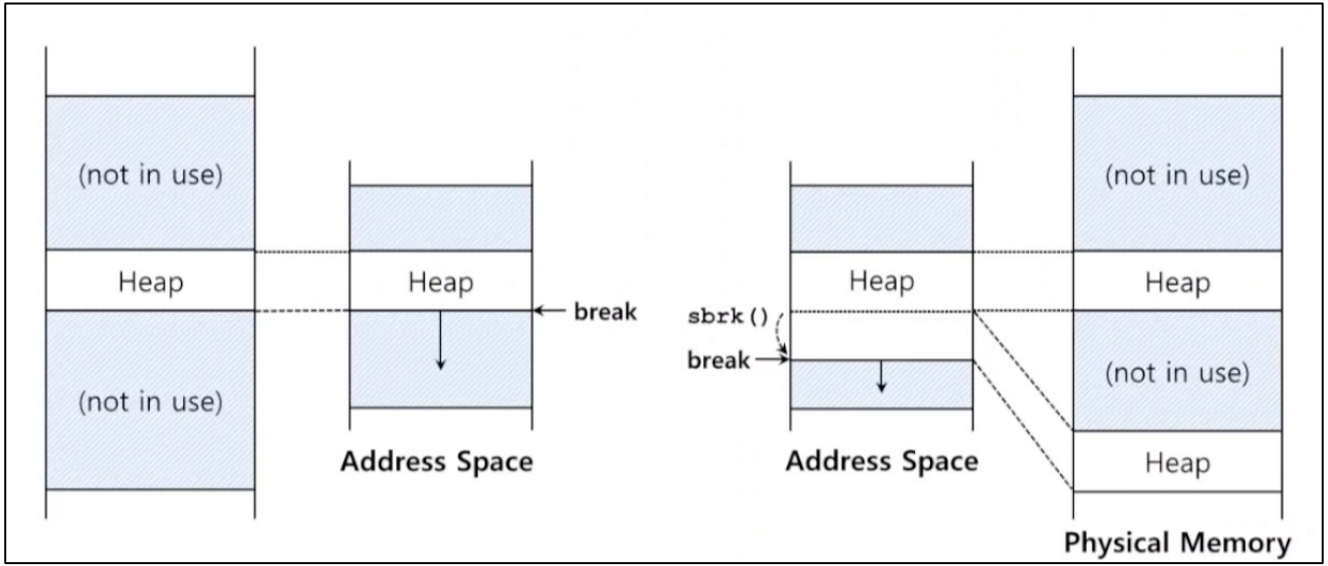
包含已释放块的空闲空间

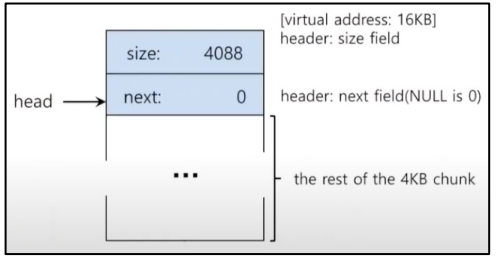
- 假设最后两个正在使用的块被释放。
- 发生外部碎片（**External Fragmentation**）：
  - 需要合并（**Coalescing**） 以减少碎片



堆的增长（Growing The Heap）

- **大多数内存分配器**从小型堆（small-sized heap）开始，当内存不足时，会向操作系统请求更多内存。
  - 例如，在 UNIX 系统中使用 sbrk() 和 brk() 进行堆扩展。





管理空闲空间的基本策略

- 理想的内存分配器应当既快速，又能最小化碎片。

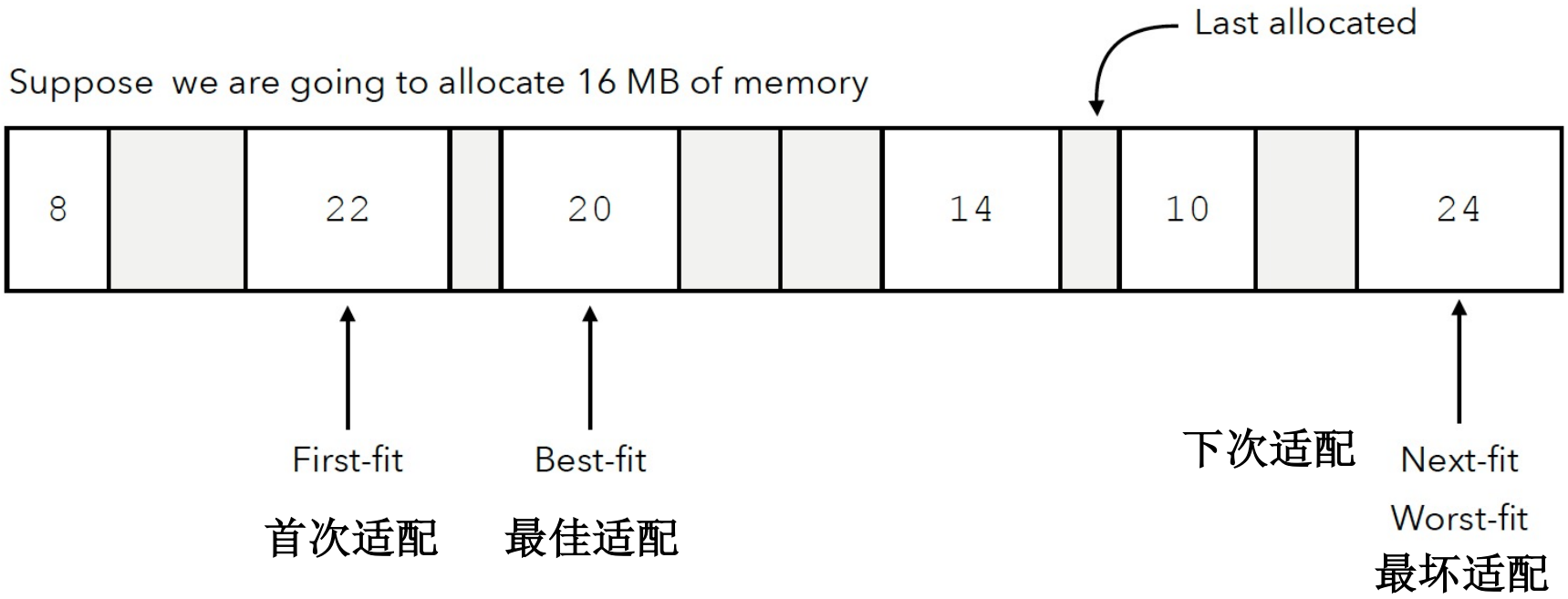
- **最佳适配（Best Fit）**
  - 寻找**大于或等于请求大小**的空闲块。
  - 选择其中**最小**的块进行分配。

- **最坏适配（Worst Fit）**
  - 寻找**最大的空闲块**，并分配所需大小。
  - **剩余部分仍留**在空闲列表中。

管理空闲空间的基本策略（续）

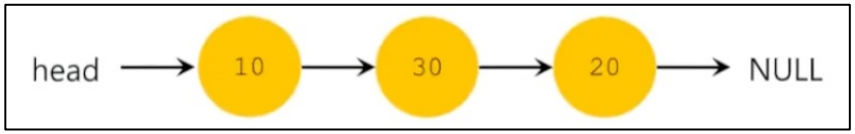
- 首次适配（**First Fit**）
  - **查找第一个足够大的空闲块**，用于满足请求。
  - **分配所需空间**，剩余部分保留在空闲列表中，以供后续请求使用。
- 下次适配（**Next Fit**）
  - **查找第一个足够大的空闲块**，用于满足请求。
  - **搜索从上次查找结束的位置继续进行**，而不是从列表头开始。

管理空闲空间的基本策略（续）



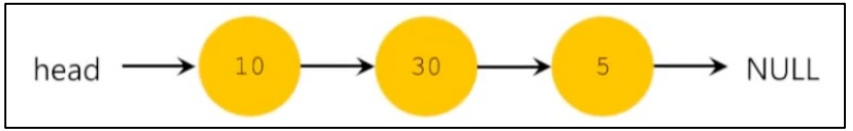
基本策略示例（Examples of Basic Strategies）

•内存分配请求大小：**15** 字节



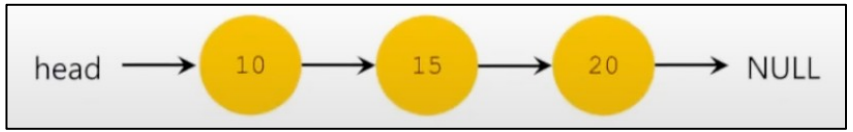
•最佳适配（**Best Fit**）结果：

- 选择**最接近 15 字节的块**（即 20 字节）。
- 15 字节被分配后，**剩余 5 字节**继续作为空闲块存储。



•最坏适配（**Worst Fit**）结果：

- 选择**最大的空闲块**（30 字节）。
- 15 字节分配后，**剩余 15 字节仍在空闲链表**中。



•关于首次适配（**First Fit**）和下次适配（**Next Fit**）？

其他方法：分离列表（Segregated Lists）

- 分离列表（**Segregated Lists**）：
  - **为常见的请求大小维护专用列表**，其他大小请求由通用分配器处理。
    - **示例**：内核数据结构，如锁（locks）、索引节点（inodes）等。
  - **引入的新问题**：
    - **需要确定**：分配给特定大小请求的**内存池**应有多大？
  - **Slab 分配器（Slab Allocator）** 解决了这个问题。

其他方法：分离列表（Segregated Lists）

- **Slab 分配器**

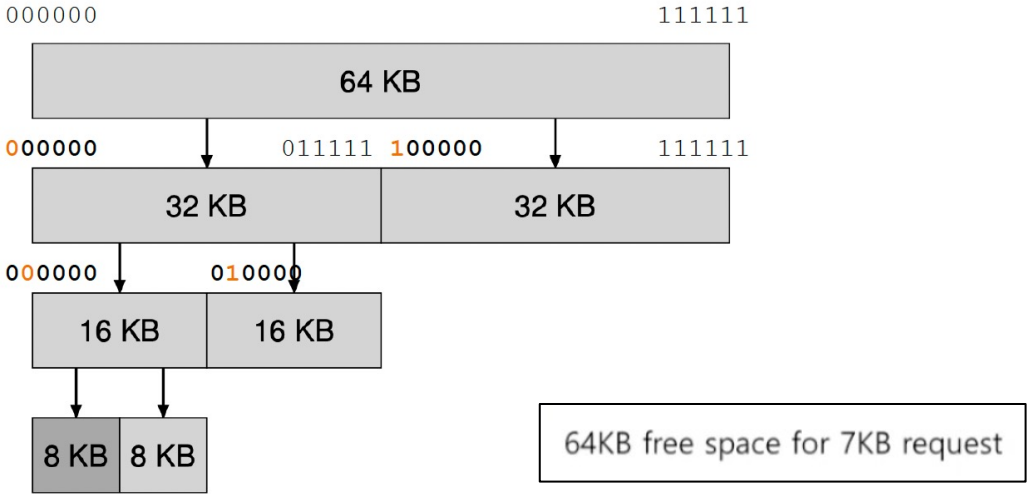
  •分配多个对象缓存：

    - 这些对象可能会被频繁请求
    - 例如：锁、文件系统 inodes 等

  •当某个缓存空间不足时，从更通用的内存分配器请求额外内存

其他方法：伙伴分配（Buddy Allocation）

- 二分伙伴分配系统（**Binary Buddy Allocation**） — 优化合并（**Coalescing**）
  - 当发出内存请求时，系统会**递归地按 2 的幂分割空闲空间**，直到找到**足够大的块**来满足请求。
  - 分配器**不断将空闲空间对半分割**，直到找到适合的块。



其他方法：伙伴分配（Buddy Allocation）

- 伙伴分配可能会遭受**内部碎片**的影响。

- 伙伴系统使**合并（coalescing）**变得简单。

  •**合并**两个相邻的块形成更大的块，提升内存利用率。

- 其他方法使用更高级的数据结构来解决扩展性和多处理器系统的问题。

小结

介绍了空闲空间管理，及一些底层机制

介绍了如何建立空闲列表

介绍了管理空闲空间的几个基本策略