IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

物理内存管理

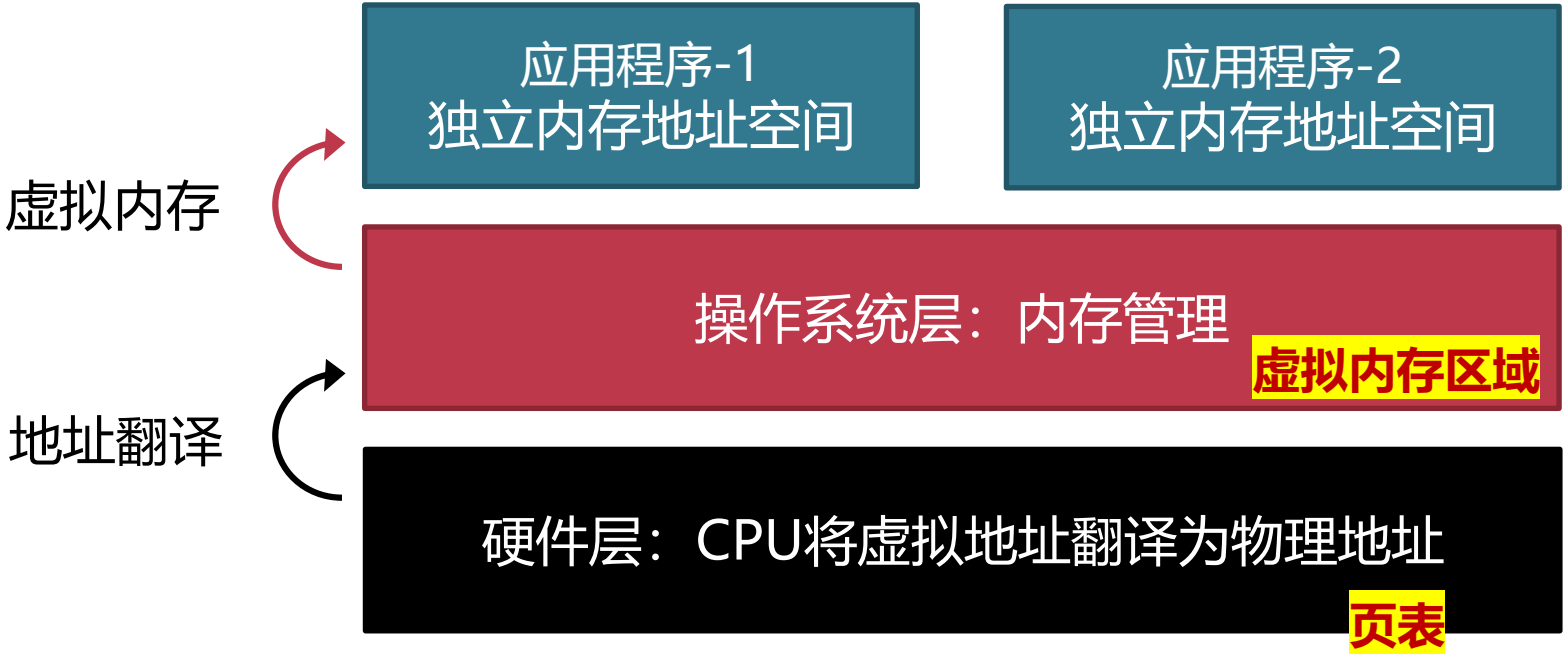
上海交通大学

https://www.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 资料来自上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

内容提纲


应用程序-1
独立内存地址空间
应用程序-2
独立内存地址空间
虚拟内存
操作系统层：内存管理
虚拟内存区域
地址翻译
硬件层：CPU将虚拟地址翻译为物理地址
页表

回顾：AARCH64支持多种最小页面大小

- **TCR_EL1可以选择不同的最小页面大小**
  - 3种配置：4K、16K、64K
  - 4K + 大页：2M/1G
  - 16K + 大页：32M（问：为什么是32M？）
    - 只有L2页表项支持大页
  - 64K + 大页：512M
    - 只有L2页表项支持大页 （ARMv8.2之前）

回顾+思考

- 一条add指令是否可能触发缺页异常?
- 一条ldr指令最多可能几次触发缺页异常?
- 访问0地址：到底发生了什么（全过程）？

OS职责：分配物理内存资源

- **引入虚拟内存后，物理内存分配主要在以下四个场景出现：**
  1. 用户态应用程序触发on-demand paging（延迟映射）时
     - 此时内核需要分配物理内存页，映射到对应的虚拟页
  2. 内核自己申请内存并使用时
     - 如用于内核自身的数据结构，通常通过kmalloc()完成
  3. 发生换页（swapping）时
     - 通过磁盘来扩展物理内存的容量
  4. 内核申请用于设备的DMA内存时
     - DMA内存通常需要连续的物理页

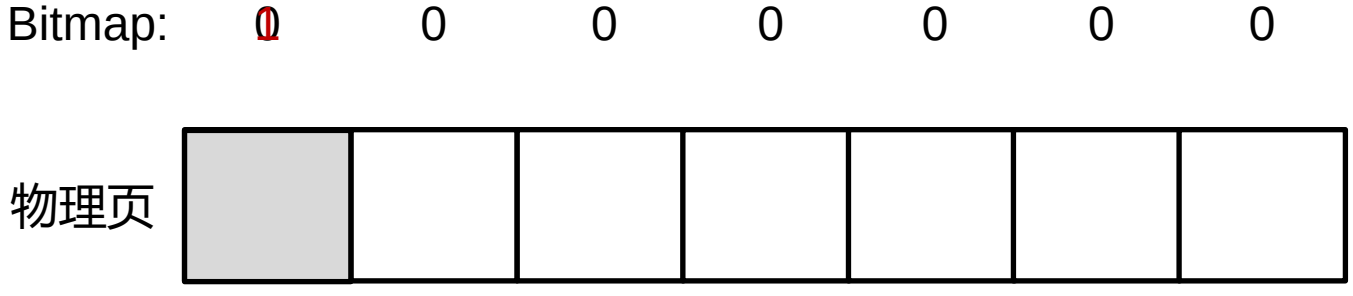
场景-1：应用触发on-demand paging

- **问：当应用调用malloc时，OS是否需要分配物理内存?**
  - 应用调用malloc后，返回的虚拟地址属于某个VMA
  - 但虚拟地址对应的页表项的valid bit可能为0
  - 当第一次访问新分配的虚拟地址时，CPU会触发page fault
- **操作系统需要做（即page-fault handler）：**
  - 找到一块空闲的物理内存页 ← **物理内存管理（页粒度）**
  - 修改页表，将该物理页映射到触发page-fault的虚地址所在虚拟页
  - 恢复应用，重复执行触发page-fault的那行代码

回顾：分配物理页的简单实现

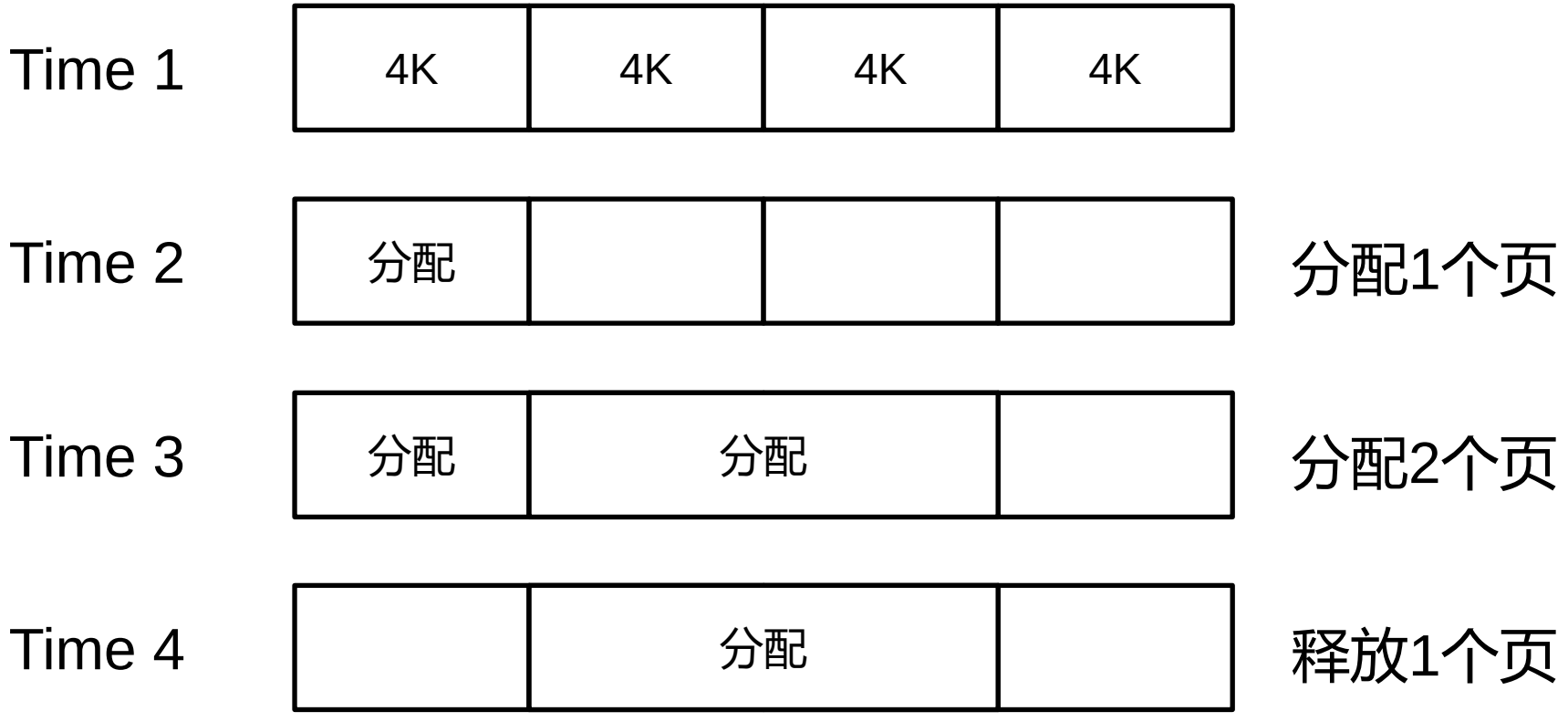
alloc_page()
接口的实现

- **操作系统用位图记录物理页是否空闲**
  - 分配时，通过bitmap查找空闲物理页，并在bitmap中标记非空闲
  - 回收时，在bitmap中，把对应的物理页标记成空闲



物理内存分配需求：需要能够分配连续的4K物理页（如大页、场景-3DMA）

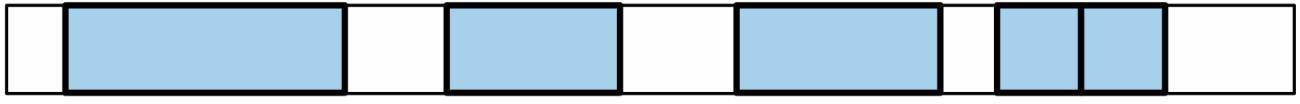
简单管理方法导致外部碎片问题



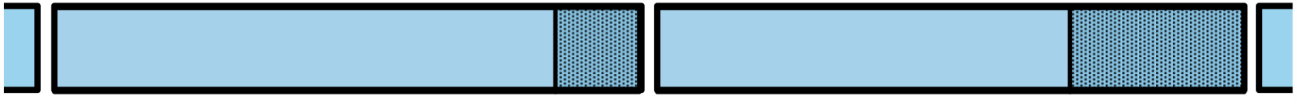
物理内存分配器的指标

1. 资源利用率 2. 分配性能

– 外部碎片与内部碎片



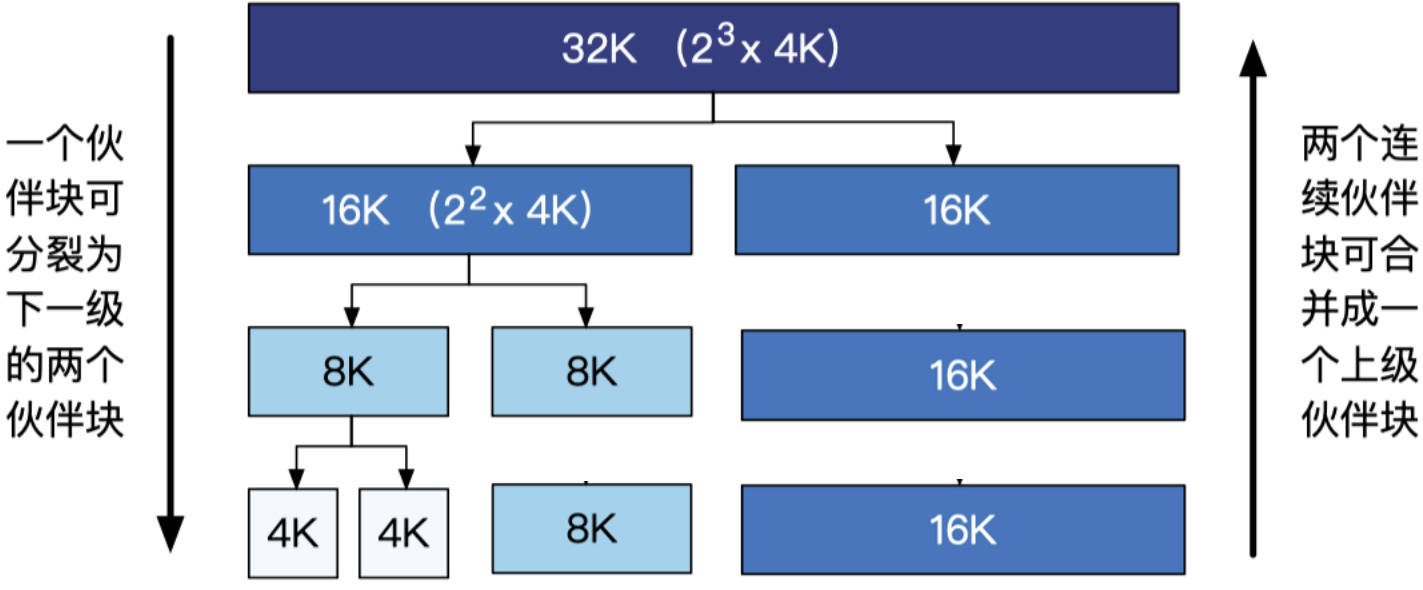
外部碎片：单个空白部分都小于分配请求的内存大小，但加起来足够
注：蓝色部分表示已分配内存，空白部分为未分配内存



内部碎片：蓝色阴影部分是分配内存大于实际使用内存而导致的内部碎片
注：黑色粗线框表示已分配内存，蓝色部分表示实际使用内存，蓝色阴影表示已分配但未被使用部分

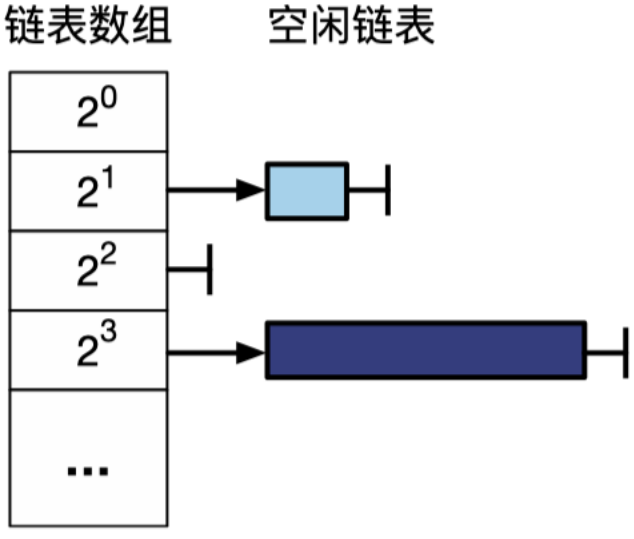
伙伴系统（buddy system）

- 伙伴系统：分裂与合并（避免外部碎片）



分裂：以分配15K内存为例

当一个请求需要分配 m 个物理页时，伙伴系统将寻找一个大小合适的块，该块包含 2^n个物理页，且满足 $2^{n-1} < m \leqslant 2^n$

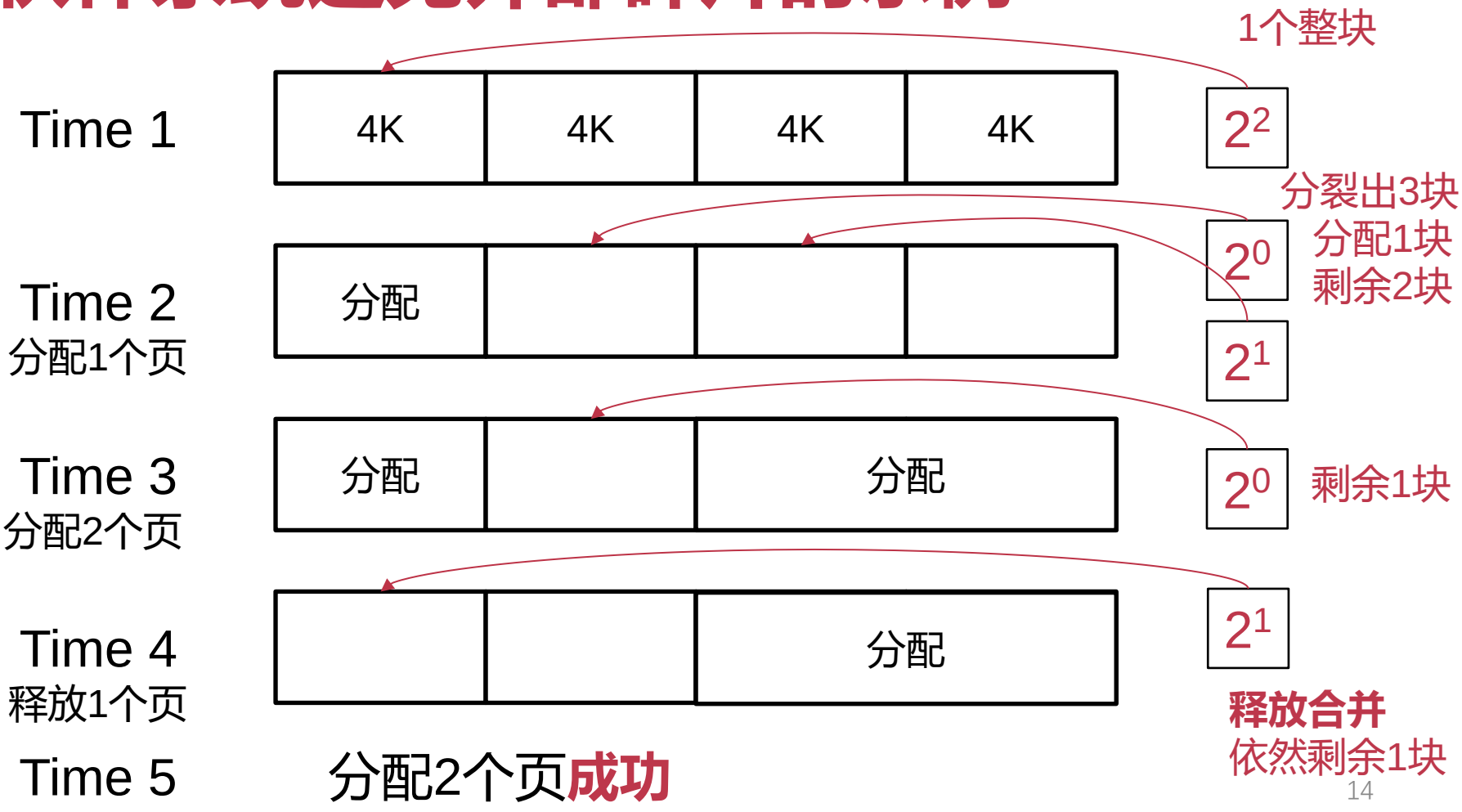


把空闲块按照大小放在相应的链表中

合并：释放块时如何定位伙伴块

- **高效地找到伙伴块**
  - 互为伙伴的两个块的物理地址**仅有一位**不同
  - 而且块的**大小决定**是哪一位
- **例如：**
  - 块A（0-8K）和块B（8-16K）互为伙伴块
  - 块A和B的物理地址分别是 0x0 和 0x2000
    - 仅有第13位不同，块大小是8K（2^{13}）

伙伴系统避免外部碎片的示例



伙伴系统：以页为粒度的物理内存管理

- **分配物理页/连续物理页（2^n）**
  - 直接映射：
    - OS一次性将所有物理内存映射到一段高虚拟地址（OS使用的虚拟地址）
    - 任一物理地址和OS使用的虚拟地址仅相差一个偏移量
    - OS可迅速根据物理地址或虚拟地址互相计算
      - 用途举例1：OS为应用分配物理页，在页表中添加映射需要物理地址；OS对物理页清零需要使用虚拟地址
      - 用途举例2：OS分配页表页（即分配单个物理页），页表项内容需要设置物理地址，而填写页表项需要使用虚拟地址

伙伴系统：以页为粒度的物理内存管理

- **分配物理页/连续物理页（2^n）**
- **资源利用率**
  - 外部碎片程度降低（**思考：是否不再出现外部碎片？**）

如何进一步减少外部碎片？memory compact

伙伴系统：以页为粒度的物理内存管理

- **分配物理页/连续物理页（2^n）**
- **资源利用率**
  - 外部碎片程度降低（**思考：是否不再出现外部碎片？**）
  - 内部碎片依然存在：如请求9KB，分配16KB（4个页）；分配1KB呢?
- **分配性能**

	期望情况	最差情况
思考：分配的时间复杂度?	O(1)	O(list-num) 常数时间
思考：合并的时间复杂度?	O(1)	O(list-num) 常数时间

伙伴系统的代码实现

描述物理页的数据结构

```
1  struct physical_page {
2    // 是否已经分配
3    int allocated;
4    // 所属伙伴块大小的幂次
5    int order;
6    // 用于维护空闲链表，把该页放入/移出空闲链表时使用
7    list_node node;
8  };
9
10 // 伙伴系统的空闲链表数组
11 list free_lists[BUDDY_MAX_ORDER];
```

操作系统维护struct physical_page数组

```
1  // 伙伴系统初始化
2  void init_buddy(struct physical_page *start_page,
3                  u64 page_num)
4  {
5    int order;
6    int index;
7    struct physical_page *page;
8
9    // 初始化物理页结构体数组
10   for (index = 0; index < page_num; ++index) {
11     page = start_page + index;
12     // 标记成已分配
13     page->allocated = 1;
14     page->order = 0;
15   }
16
17   // 初始化伙伴系统的各空闲链表
18   for (order = 0; order < BUDDY_MAX_ORDER; ++order) {
19     init_list(&(free_lists[order]));
20   }
21
22   // 通过释放物理页的接口把物理页插入伙伴系统的空闲链表
23   for (index = 0; index < page_num; ++index) {
24     page = start_page + index;
25     buddy_free_pages(page);
26   }
27 }
```

步骤一

步骤二

步骤三

伙伴系统分配实现

```
1  // 分配伙伴块: 2^order 数量的连续 4K 物理页
2  struct page *buddy_alloc_pages(u64 order)
3  {
4    int cur_order;
5    struct list_head *free_list;
6    struct page *page = NULL;
7
8    // 搜寻伙伴系统中的各空闲链表
9    for (cur_order = order; cur_order < BUDDY_MAX_ORDER;
     ↪  ++cur_order) {
10     free_list = &(free_lists[cur_order]);
11     if (!list_empty(free_list)) {
12       // 从空闲链表中取出一个伙伴块
13       page = get_one_entry(free_list);
14       break;
15     }
16   }
17
18   // 若取出伙伴块大于所需大小，则进行分裂
19   page = split_page(order, page);
20   // 标记已分配。示意代码忽略分配失败的情况
21   page->allocated = 1;
22   return page;
23 }
```

page_to_virt & virt_to_page

问：如何从page结构体，获取物理地址/虚拟地址

伙伴系统释放实现

```
25 // 释放伙伴块
26 void buddy_free_pages(struct page *page)
27 {
28   int order;
29   struct list_head *free_list;
30
31   // 标记成空闲
32   page->allocated = 0;
33   // (尝试) 合并伙伴块
34   page = merge_page(page);
35
36   // 把合并后的伙伴块放入对应大小的空闲链表
37   order = page->order;
38   free_list = &(free_lists[order]);
39   add_one_entry(free_list, page);
40 }
```

SLAB/SLUB：细粒度内存管理

场景-2：内核运行中需要进行动态内存分配

- **内核自身用到的数据结构**
  - 为每个进程创建的process，VMA等数据结构
  - 动态性：用时分配，用完释放，类似用户态的malloc/new
  - 数据结构大小往往远小于页粒度
  - 接口：vaddr_t **kmalloc**(u64 size);

    void **kfree**(vaddr_t kva);

SLAB：建立在伙伴系统之上的分配器

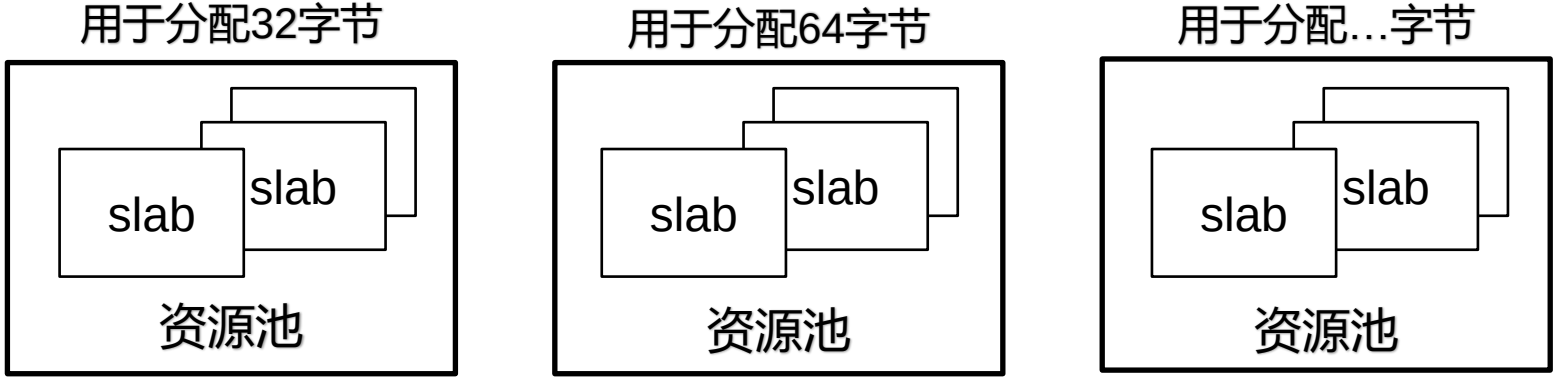
- **目标：快速分配小内存对象**
  - 内核中的数据结构大小远小于4K（例如VMA）
- **SLAB分配器家族：SLAB、SLUB、SLOB**
  - 上世纪 90 年代，Jeff Bonwick在Solaris 2.4中首创SLAB
  - 2007年左右，Christoph Lameter在Linux中提出SLUB
    - Linux-2.6.23之后SLUB成为默认分配器
  - SLOB：针对内存稀缺场景的（从Linux 6.2起被放弃）
- **后续以主流的SLUB为例讲解**

SLUB分配器的基本设计思路

- **观察**
  - 操作系统频繁分配的对象大小相对比较固定
- **基本思想**
  - 从伙伴系统获得大块内存（名为slab）
  - 对每份大块内存进一步细分成固定大小的小块内存进行管理
  - 块的大小通常是 2^n 个字节（一般来说，$3 \leqslant n < 12$）
  - 也可为特定数据结构增加特殊大小的块，从而减小内部碎片

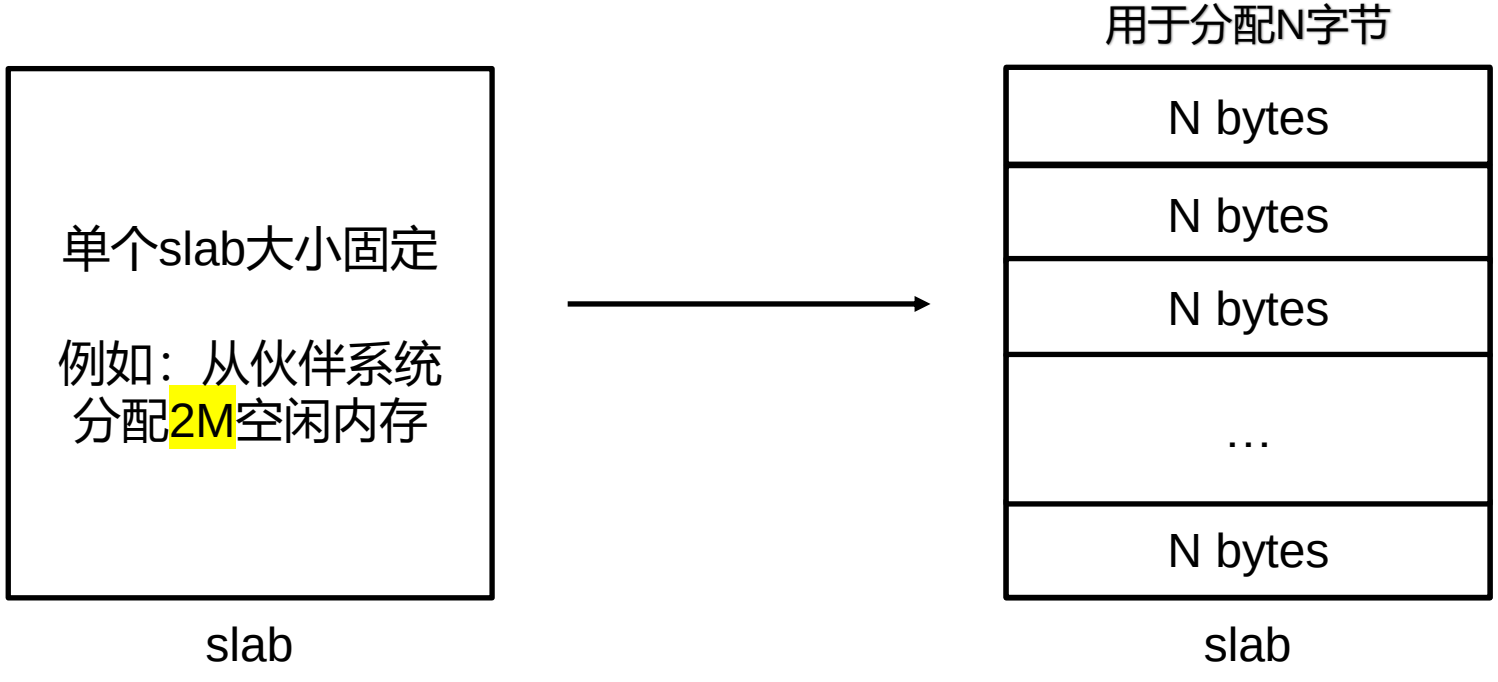
SLUB设计

- **只分配固定大小块**
- **对于每个固定块大小，SLUB 分配器都会使用独立的内存资源池进行分配**
  - 采用best fit定位资源池



SLUB设计：初始化资源池

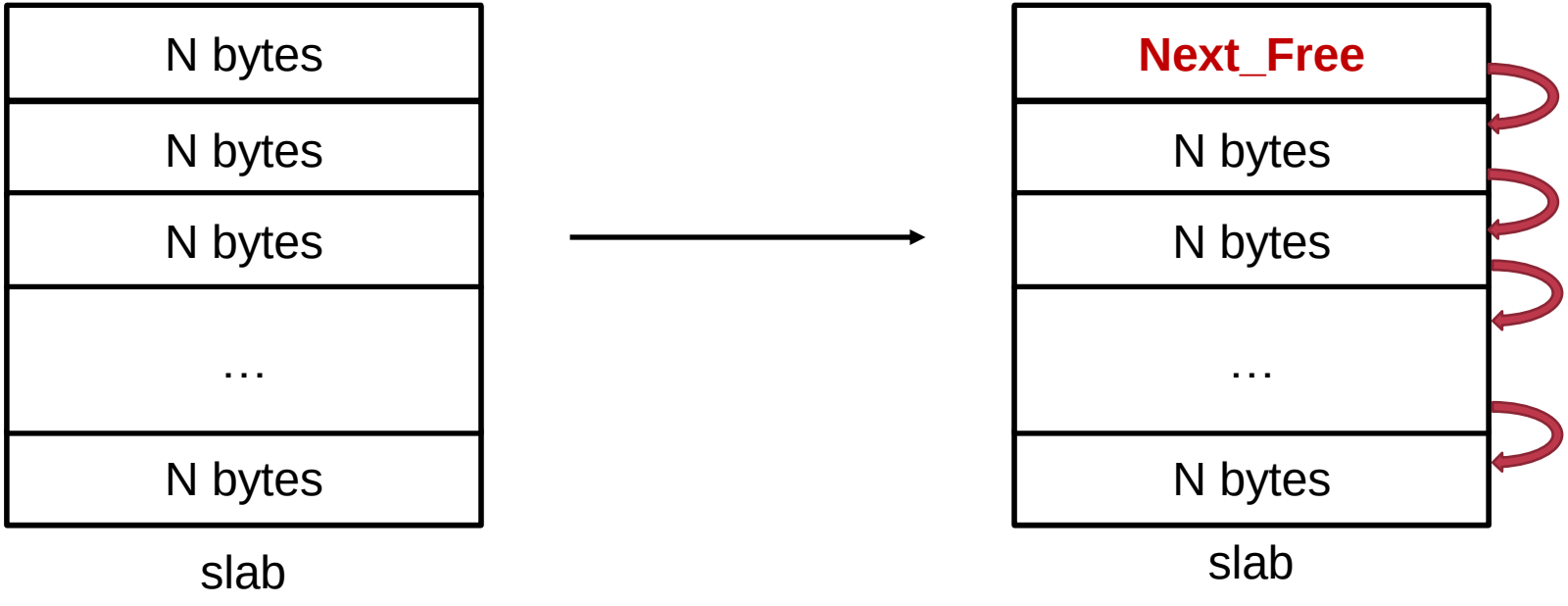
把从伙伴系统得到的连续物理页**划分成若干等份**（slot）



SLUB设计：空闲链表

如何区分是否空闲？采用空闲链表

当分配时直接分配一个空闲slot

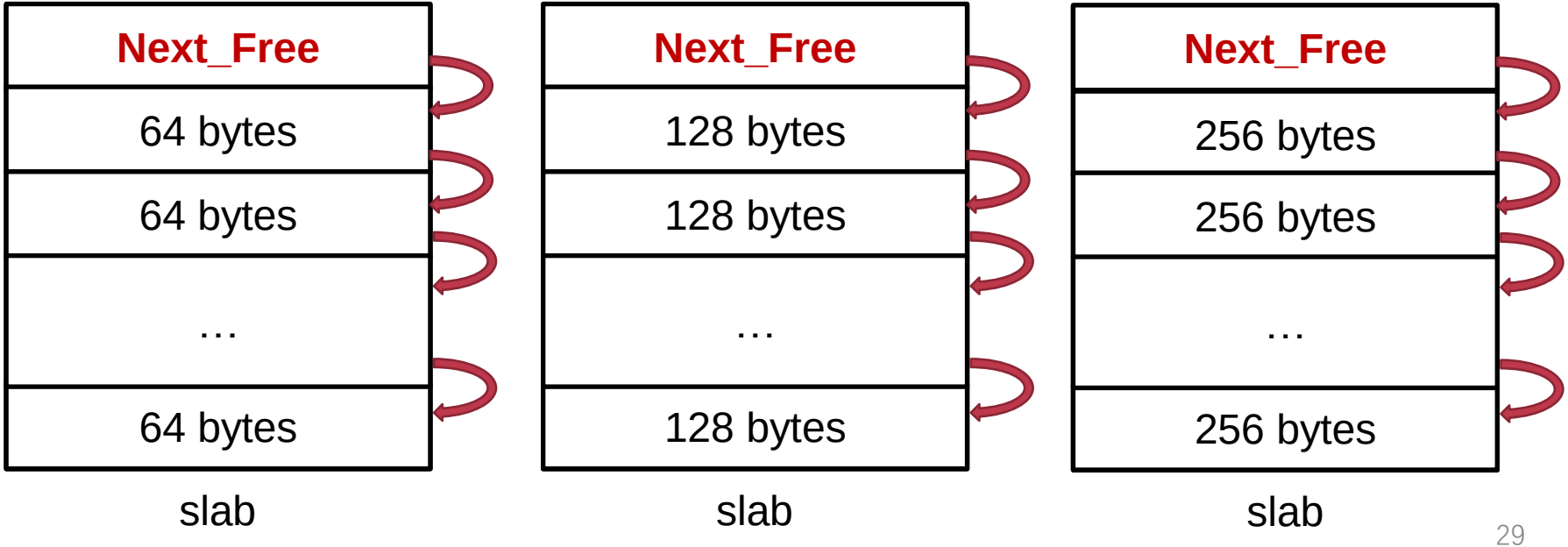


SLUB设计：分配与释放

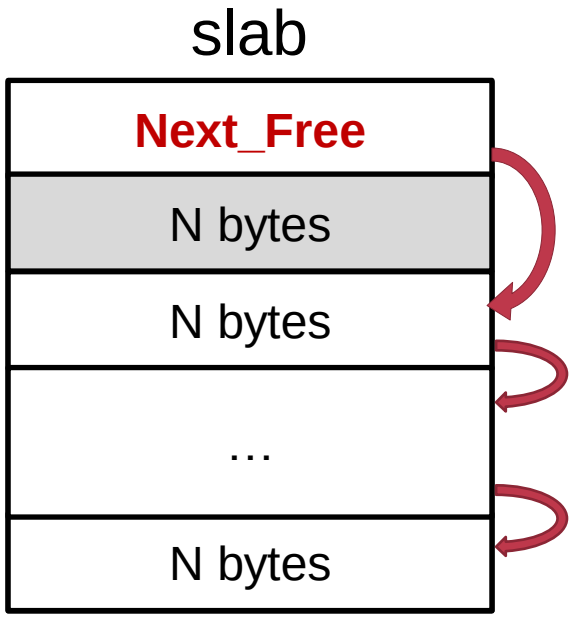
分配N字节：1. 定位到大小最合适的资源池（假设只有一个slab），

2. 从slab中取走Next_Free指向的第一个slot

释放：将Next_Free指针指向待释放内存（slot）



SLUB设计：释放



释放时如何找到Next_Free?

提示：slab的大小是固定的

思路：根据待释放内存地址（X）计算slab起始地址

X & ~(SLAB_SIZE-1)

即为：向下取整到对齐SLAB_SIZE的地址

SLUB设计：释放

kmalloc(u64 size)

- size >= 4K，从伙伴系统获取
- size < 4K, 从SLUB分配器获取
- 释放统一用kfree(void *ptr)

释放时如何找到Next_Free?

提示：slab的大小是固定的

思路：根据待释放内存地址可计算slab起始地址

不可行 X

```
struct physical_page {
  // 是否已经分配
  int allocated;
  // 所属伙伴块大小的幂次
  int order;
  // 用于维护空闲链表，把该
  list_node node;
+ struct slab *slab;
};
```

思考：上述方法在kfree(addr)接口下可行吗?

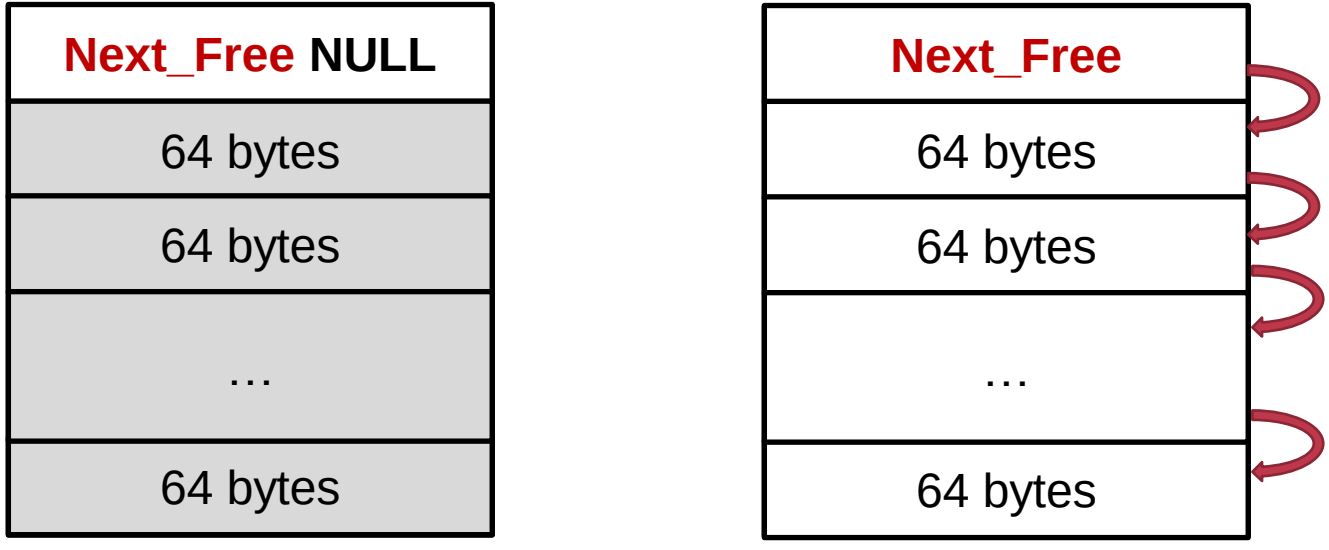
问题：没有size信息，无法判断addr是被slab分配的，还是伙伴系统分配的

解决方法：在物理页结构体中记录所属slab信息

SLUB设计：新增slab

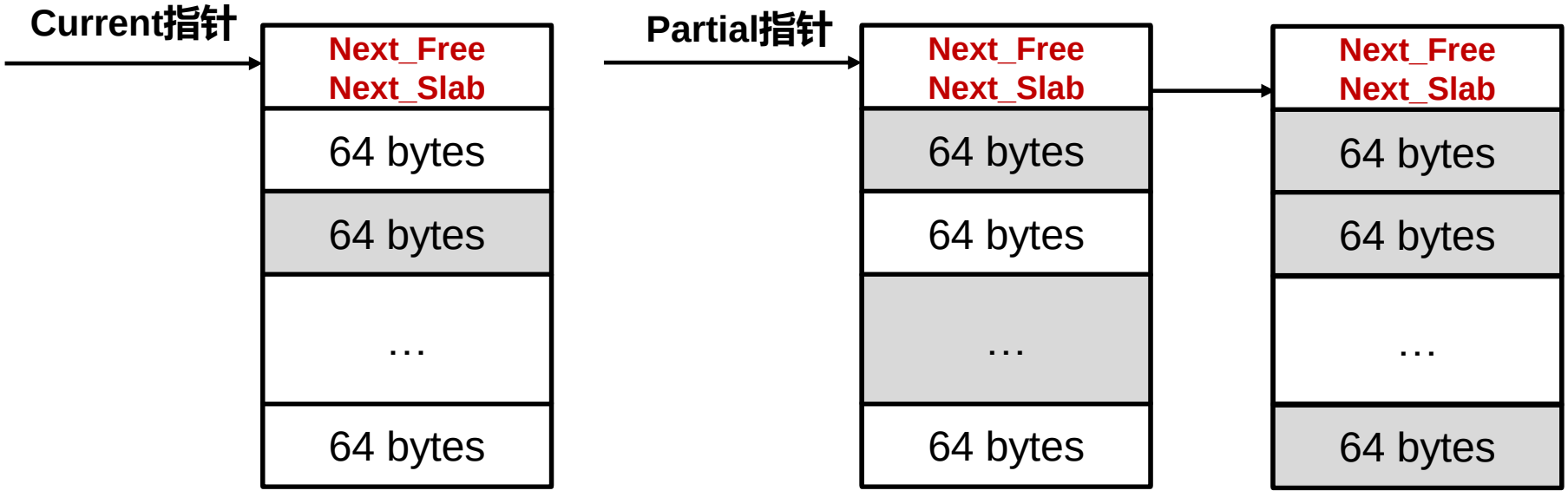
当某个资源池（例如64字节对应的资源池）中的slab已经分配完怎么办?

再从伙伴系统分配一个slab



SLUB设计：资源池内组织多个slab

如何组织多个64字节slot的SLAB? 引入两个指针



- Current指向一个slab，**并从其中分配；**
- 当Current slab全满，则从Partial链表中取出一个放入Current
- 释放后，若某个slab不再全满，则加入partial
- 释放后，若某个slab全空则可还给伙伴系统

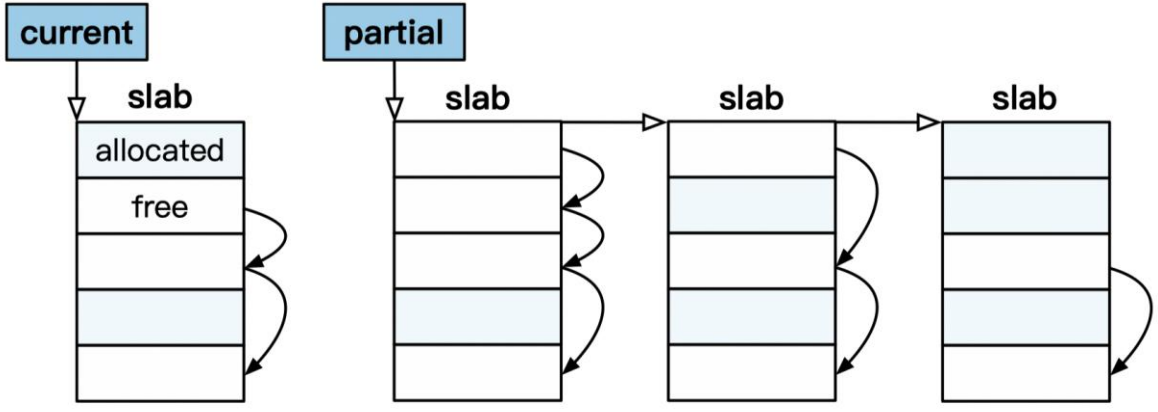
SLUB小结

针对每种slot大小维护两个指针：

- current仅指向一个 slab
  - 分配时使用、按需更新
- partial指向未满slab链表
  - 释放时若全free，则还给伙伴系统

从伙伴系统获得的物理内存块称为 slab

slab内部组织为空闲链表



SLUB小结

优势：

1. 减少内部碎片（可根据开发需求）
2. 分配效率高（常数时间）

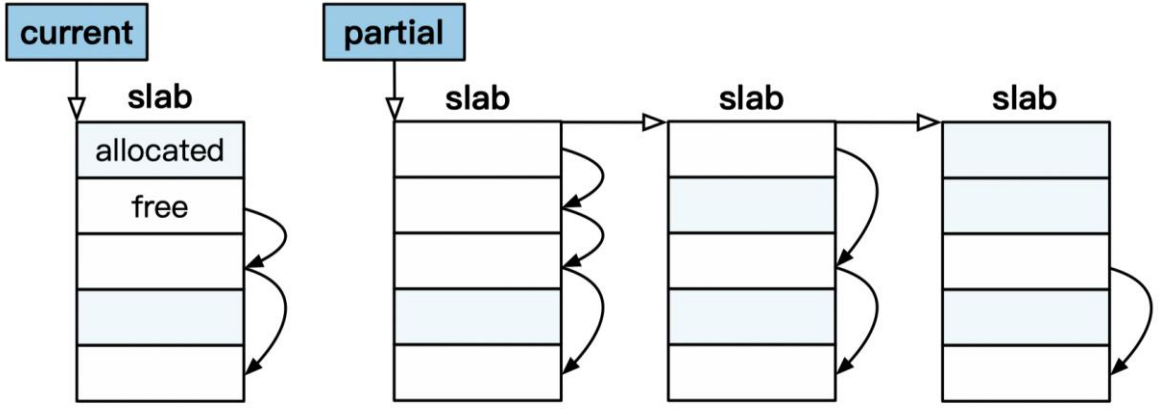
针对每种slot大小维护两个指针：

- current仅指向一个 slab
  - 分配时使用、按需更新
- partial指向未满slab链表
  - 释放时若全free，则还给伙伴系统

从伙伴系统获得的物理内存块称为 slab

slab内部组织为空闲链表

1. 思考：选择哪些slot大小？
2. 思考：分配与释放的时间复杂度？



突破物理内存容量限制

场景-3：物理内存容量<应用进程需求（swap)

手机总内存大小12GB

列出一些具体的APP名称和通常占用的大小

1. 大型游戏应用：这些是图形密集型游戏，通常包含高质量的图形、音频和视频资源。它们的大小通常在几百兆字节（MB）到数个几十GB之间。

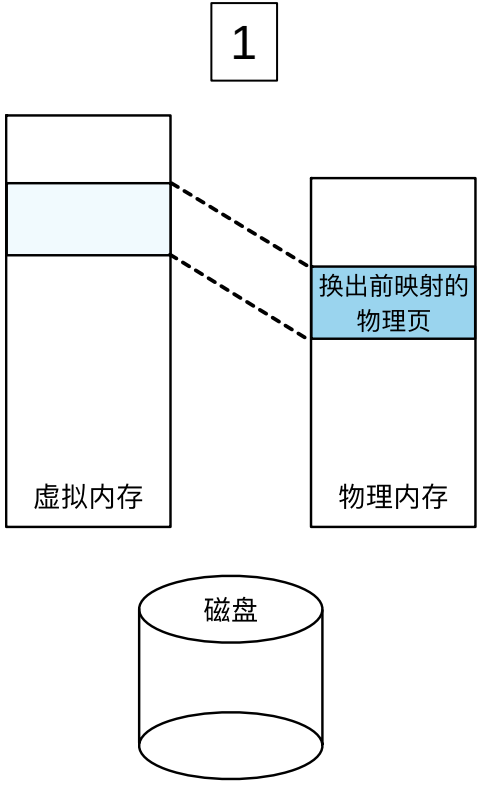
2. 多媒体应用：多媒体应用包括音乐播放器、视频播放器和图库应用等。它们的大小通常在几十MB到几个GB之间，取决于媒体库的大小和应用的功能。

3. 社交媒体应用：社交媒体应用如Facebook、Instagram和微信等，通常包含大量的用户数据、图片和视频等资源。它们的大小通常在几十MB到几个GB之间。

换页机制（Page Swap）

- **换页的基本思想**
  - 用磁盘作为物理内存的补充，且对上层应用透明
  - 应用对虚拟内存的使用，不受物理内存大小限制
- **如何实现**
  - 磁盘上划分专门的Swap分区，或专门的Swap文件
  - 在处理缺页异常时，触发物理内存页的换入换出

换页示例：换出与换入



问题1. 如何判断缺页异常是由于换页引起的

- **导致缺页异常的三种可能**
  - 访问非法虚拟地址
  - 按需分配（尚未分配真正的物理页）
  - 内存页数据被换出到磁盘上
- **OS如何区分?**
  - 利用VMA区分是否为合法虚拟地址（合法缺页异常）
  - 利用页表项内容区分是按需分配还是需要换入

练习

应用进程地址空间中的虚拟页可能存在四种状态，分别是：

1. 未分配；
2. 已分配但尚未为其分配物理页；
3. 已分配且映射到物理页；
4. 已分配但对应物理页被换出。

请问当应用进程访问某虚拟页时，在上述四种状态下，操作系统会分别做什么？

问题2：何时进行换出操作

- **策略A**
  - 当用完所有物理页后，再按需换出
  - 回顾：alloc_page，通过伙伴系统进行内存分配
  - 问题：当内存资源紧张时，大部分物理页分配操作都需要触发换出，造成分配时延高
- **策略B**
  - 设立阈值，在空闲的物理页数量低于阈值时，操作系统择机（如系统空闲时） 换出部分页，直到空闲页数量超过阈值
  - Linux Watermark：高水位线、低水位线、 最小水位线

回顾：延迟映射 vs. 立即映射

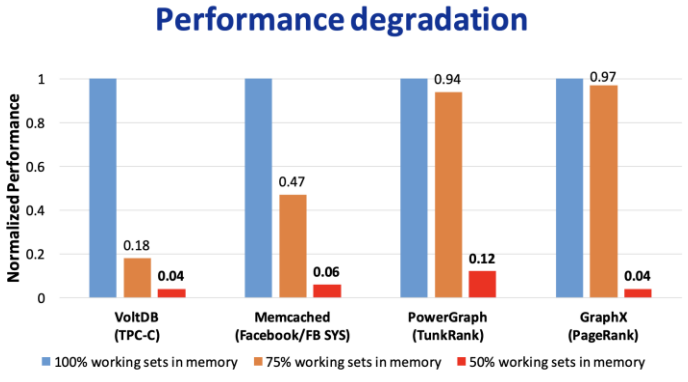
- **优势：节约内存资源**
- **劣势：缺页异常导致访问延迟增加（换页面临相似问题）**
- **如何取得平衡?**
  - 应用程序访存具有时空局部性（Locality）
  - 在缺页异常处理函数中采用预先映射的策略（预测相邻的虚拟页也会被访问，提前映射）
    - 在节约内存和减少缺页异常次数之间取得平衡

问题3：换页机制的代价

- **优势：突破物理内存容量限制**
- **劣势：缺页异常+磁盘操作导致访问延迟增加**
- **如何取得平衡?**
  - 预取机制 （Prefetching）
    - 预测接下来进程要使用的页，提前换入
    - 在缺页异常处理函数中，根据应用程序访存具有的空间本地性进行预取

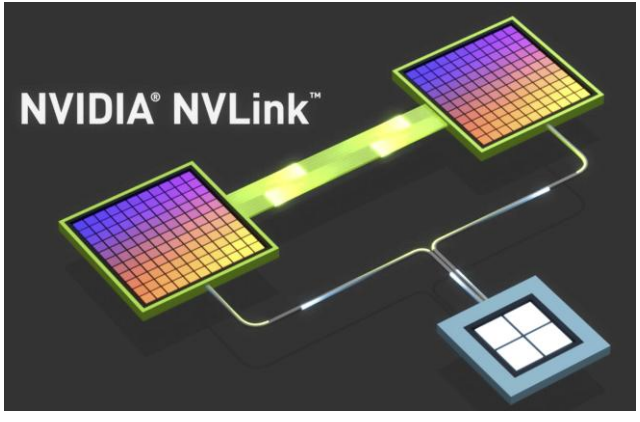
小知识

- **换出到磁盘/SSD的性能开销较高**
  - 研究方向：内存池化/远端内存
  - 硬件互联：RDMA/CXL



NSDI 2017 Infiniswap

- **GPU内存不够怎么办**
  - GPU内存换出到CPU侧的内存
  - NVLINK



问题4：如何选择换出的页

- **页替换策略**
  - 选择一些物理页换出到磁盘
  - 猜测哪些页面**应该**被换出（短期内大概率不会被访问）
  - 策略实现的开销

理想的换页策略（OPT策略）

假设物理内存中可以存放三个物理页，初始为空，

某应用程序一共需要访问物理页面 1～5

OPT：优先换出未来最长时间内不会再访问的页面

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
物理内存中 存放的物理页面												
缺页异常（共 6 次）	是	是	否	是	是	否	是	否	否	是	否	否

FIFO策略

操作系统维护一个队列用于记录换入内存的物理页号，

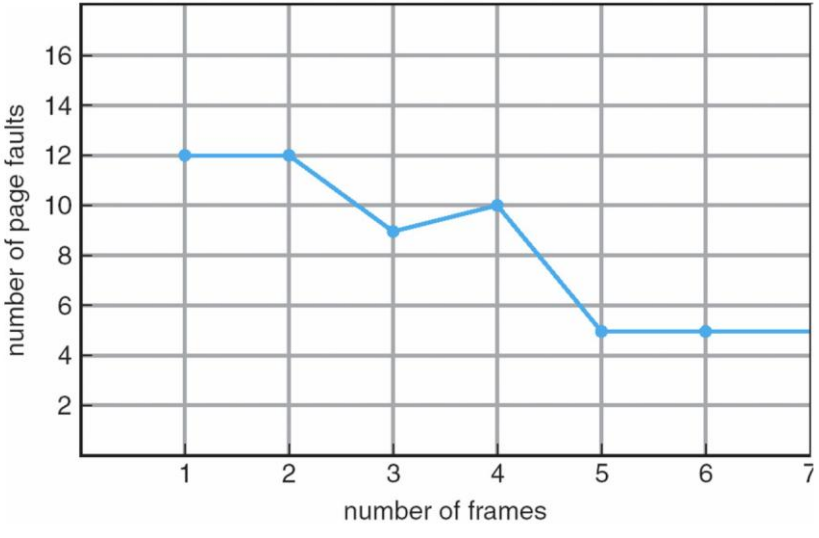
每换入一个物理页就把其页号加到队尾，

因此最先换进的物理页号总是处于队头位置

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
（该行为队列头）存储物理页号的 FIFO 队列												
缺页异常（共 9 次）	是	是	否	是	是	是	是	否	是	否	是	是

Belady's Anomaly

- **访问顺序：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
  - 3个物理页：几次缺页异常？ 9次
  - 4个物理页：几次缺页异常？ 10次



Second Chance策略

FIFO 策略的一种改进版本：为每一个物理页号维护一个访问标志位。

如果访问的页面号已经处在队列中，则置上其访问标志位。

换页时查看队头：1）无标志则换出；2）有标志则去除并放入队尾，继续寻找

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
该行是队列头部 FIFO 队列 存储物理页号												
缺页异常（共 6 次）	是	是	否	是	是	否	是	否	是	否	否	否

LRU策略

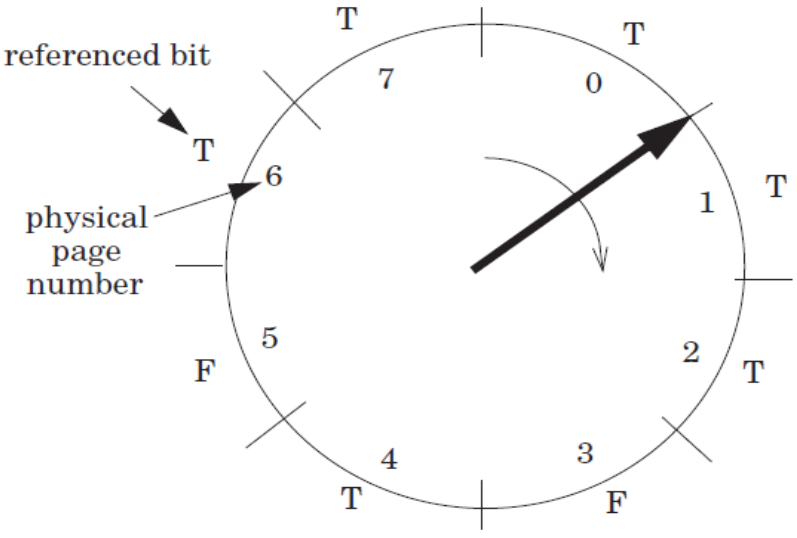
OS维护一个链表，在每次内存访问后，OS把刚刚访问的内存页调整到链表尾端；每次都选择换出位于链表头部的页面

缺点-1：对于特定的序列，效果可能非常差，如循环访问内存

缺点-2：需要排序的内存页可能非常多，导致很高的额外负载

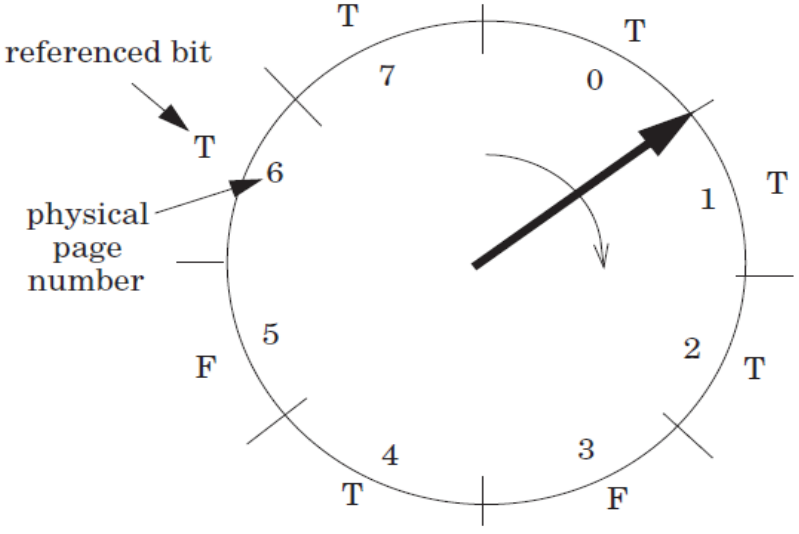
物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
该行为链表头部越不常访问的页号离头部更近												
缺页异常（共7次）	是	是	否	是	是	否	是	否	是	是	否	否

时钟算法策略



- **精准的LRU策略难以实现**
- **物理页环形排列（类似时钟）**
  - 为每个物理页维护一个访问位
  - 当物理页被访问时， 把访问位设成T
  - OS依次（如顺时针）查看每个页的“访问位”
    - 如果是T，则置成F
    - 如果是F，则驱逐该页

时钟算法策略



针臂：

- 指向换入位置
- 需要换出时：从下一个位置开始转

物理页面访问顺序	3	2	3	1	4	3	5	4	2	3	4	3
	3*	3* 2*	3* 2*	3* 2* 1*	4* 2 1	4* 3* 1	4* 3* 5*	4* 3* 5*	2* 3 5	2* 3* 5	2* 3 4*	2* 3* 4*
缺页异常	是	是	否	是	是	是	是	否	是	否	是	否

实现时钟算法

```
1 // 在 physical_page 结构体中新增成员变量
2 struct physical_page {
3   ...
4
5   // 记录该物理页被映射到哪些页表项（称为反向映射）
6   list pgtbl_entries;
7 };
```

- **每个物理页需要有一个“访问位”**
  - MMU在页表项里面为**虚拟页**打上“访问位”
  - 回顾：页表项中的Access Flag
- **如何实现**
  - OS在描述物理页的结构体里面记录页表项位置
    - 当物理页被填写到某张页表中时，把页表项的位置记录在元数据中（在Linux中称为“反向映射”：**reverse mapping**）
    - 根据物理页对应的页表项中的“访问位”判断是否驱逐
    - 驱逐某页时应该清空其对应的所有页表项（例如共享内存）

页替换策略小结

- **常见的替换策略**
  - FIFO、LRU/MRU、时钟算法、随机替换 …
- **替换策略评价标准**
  - 缺页发生的概率 （参照理想但不能实现的**OPT策略**）
  - 策略本身的性能开销
    - 如何高效地记录物理页的使用情况?
      - 页表项中Access/Dirty Bits
- **小知识：颠簸现象 Thrashing Problem**

Thrashing Problem

- **直接原因**
  - 过于频繁的缺页异常（物理内存总需求过大）
- **大部分 CPU 时间都被用来处理缺页异常**
  - 等待缓慢的磁盘 I/O 操作
  - 仅剩小部分的时间用于执行真正有意义的工作
- **调度器造成问题加剧**
  - 等待磁盘 I/O导致CPU利用率下降
  - 调度器载入更多的进程以期提高CPU利用率
  - 触发更多的缺页异常、进一步降低CPU利用率、导致连锁反应

工作集模型（有效避免Thrashing）

- **一个进程在时间t的工作集W(t, x)：**
  - 其在时间段（t - x, t)内使用的内存页集合
  - 也被视为其在未来（下一个x时间内）会访问的页集合
  - 如果希望进程能够顺利进展，则需要将该集合保持在内存中

- **工作集模型：All-or-nothing**
  - 进程工作集要么都在内存中，要么全都换出
  - 避免thrashing，提高系统整体性能表现

小结

场景-1：应用触发on-demand paging

- **问：当应用调用malloc时，OS是否需要分配物理内存?**
  - 应用调用malloc后，返回的虚拟地址属于某个VMA
  - 但虚拟地址对应的页表项的valid bit可能为0
  - 当第一次访问新分配的虚拟地址时，CPU会触发page fault
- **操作系统需要做（即page-fault handler）：**
  - 找到一块空闲的物理内存页 ← **物理内存管理（页粒度）**
  - 修改页表，将该物理页映射到触发page-fault的虚地址所在虚拟页
  - 恢复应用，重复执行触发page-fault的那行代码

场景-2：内核运行中需要进行动态内存分配

- **内核自身用到的数据结构**
  - 为每个进程创建的process，VMA等数据结构
  - 动态性：用时分配，用完释放，类似用户态的malloc/new
  - 数据结构大小往往远小于页粒度
  - 接口：vaddr_t **kmalloc**(u64 size);

    void **kfree**(vaddr_t kva);

场景-3：换页（swap）

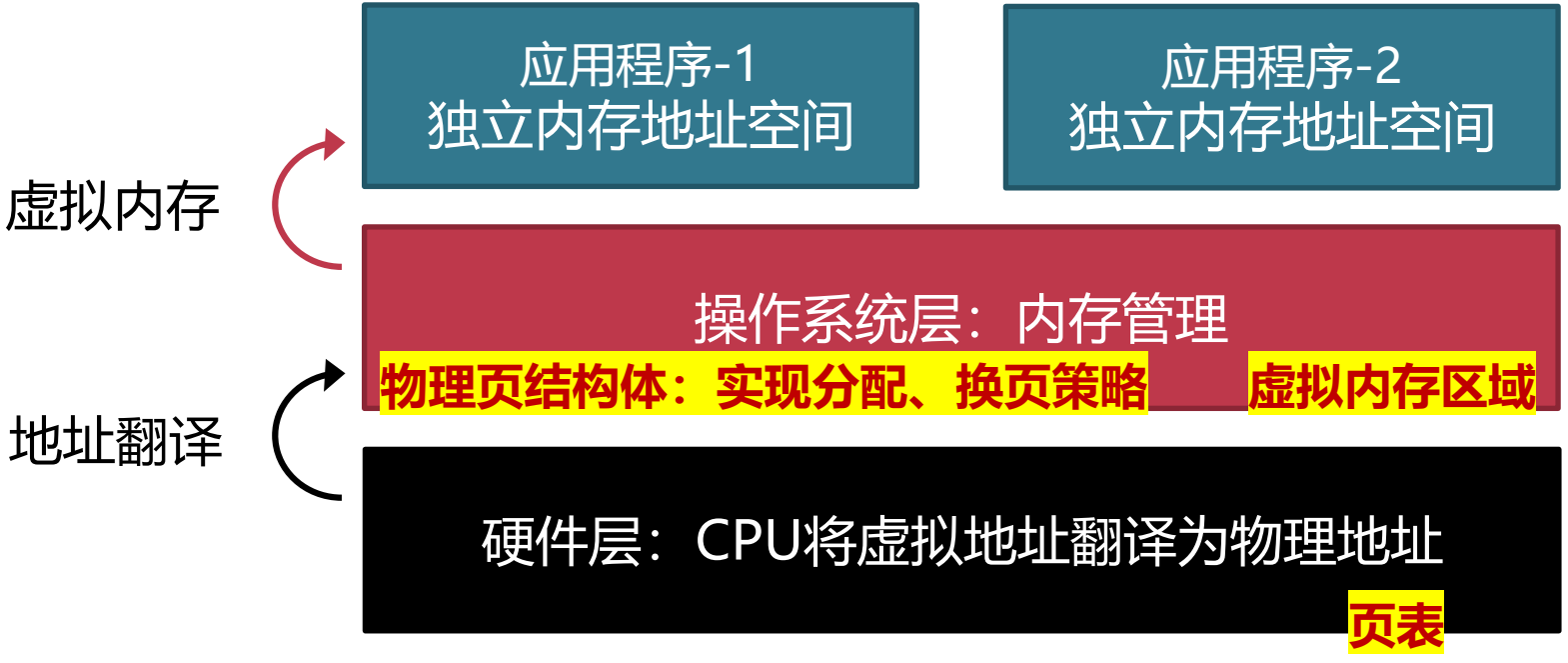
- **换出操作：物理内存不够时**
  - OS选择不常用的物理内存（不同的选择策略）
  - OS将内存中的数据写入磁盘块，并记录磁盘块与内存的关联
  - OS更新页表，将对应页表项的valid bit设置为0
- **换入操作：当换出的页被访问时，触发page fault**
  - OS判断该地址所在页被换出，找到对应的磁盘块
  - OS分配空闲的物理内存页；若没有空闲页，则再次进行换出操作
  - OS将磁盘块中的数据读入前一步分配的内存页
  - OS更新页表，将对应页表项的valid bit设置为1

场景-4：设备需要分配DMA内存

- **DMA：设备绕过CPU直接访存**
  - 由于绕过CPU的MMU，因此直接访问物理地址
  - 通常需要大段连续的物理内存
- **操作系统必须有能力分配连续的物理页**
  - 需要用一种高效的方式来组织和管理物理页

- 之后的课程会进一步介绍

总结：关键数据结构


应用程序-1
独立内存地址空间
应用程序-2
独立内存地址空间
虚拟内存
操作系统层：内存管理
物理页结构体：实现分配、换页策略
虚拟内存区域
地址翻译
硬件层：CPU将虚拟地址翻译为物理地址
页表