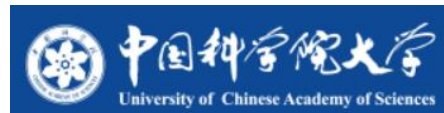




中国科学院软件研究所  
Institute of Software, Chinese Academy  
of Sciences



# 物理内存管理

郑晨

# 改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - <https://ipads.se.sjtu.edu.cn/courses/os/index.shtml>
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。

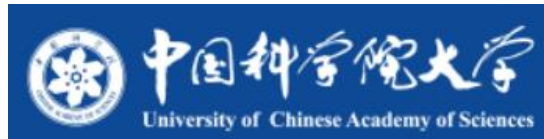


中国科学院软件研究所

Institute of Software, Chinese Academy of Sciences



上海交通大学  
SHANGHAI JIAO TONG UNIVERSITY



# 物理内存管理

# 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. 内核申请用于设备的DMA缓存时
    - DMA缓存通常需要连续的物理页
  4. 发生换页（swapping）时
    - 通过磁盘来扩展物理内存的容量

# 场景-1：应用触发on-demand paging

- **问：当应用调用malloc时，与物理内存是否有关？**
  - 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
  - 回到应用，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

Bitmap:    0        0        0        0        0        0



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

Time 1



Time 2



分配1个页

Time 3



分配2个页

Time 4



释放1个页

Time 5

请求分配2个页，**失败**，实际却有2个页

# 物理内存分配器的指标

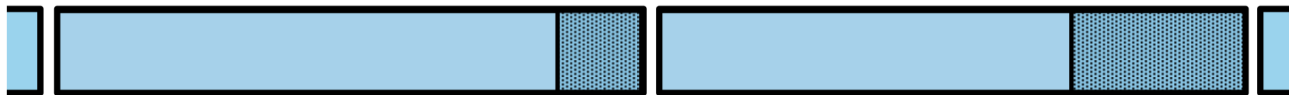
## 1. 资源利用率 2. 分配性能

### – 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



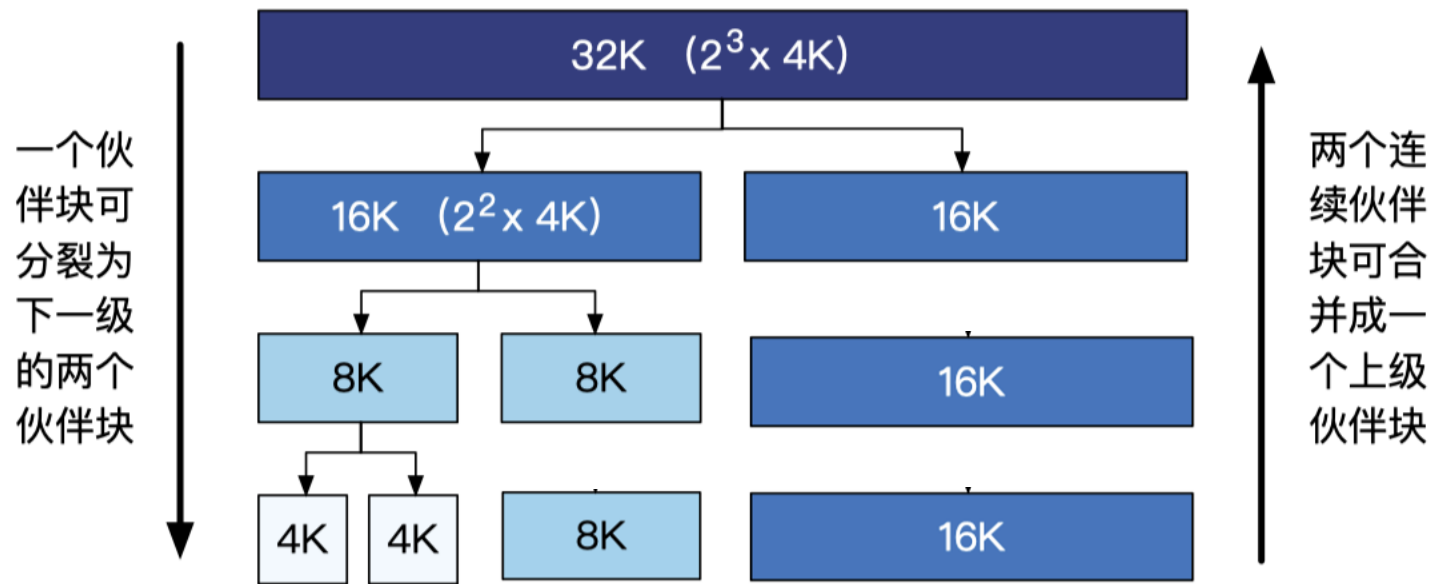
**内部碎片：** 蓝色阴影部分是分配内存大于实际使用内存而导致的内部碎片

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



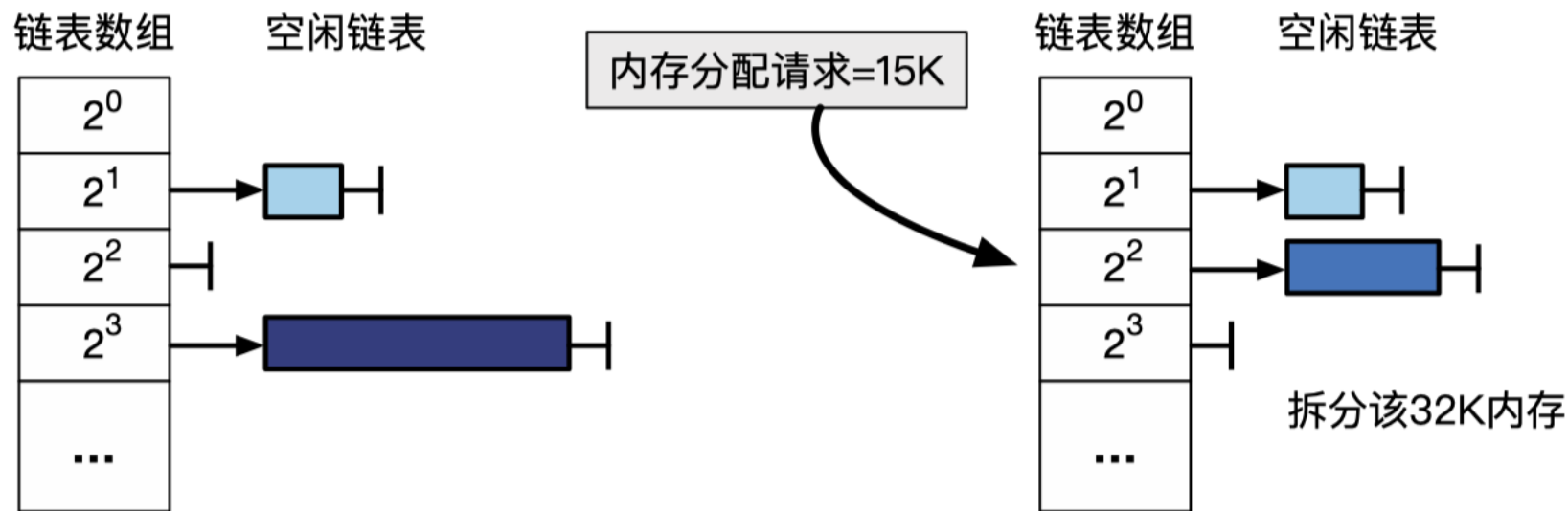
# 伙伴系统 (buddy system)

- 伙伴系统：分裂与合并（避免外部碎片）



# 伙伴系统例子：分配15K内存

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  $2^{n-1} < m \leq 2^n$



把空闲块按照大小放在相应的链表中

# 合并过程如何定位伙伴块

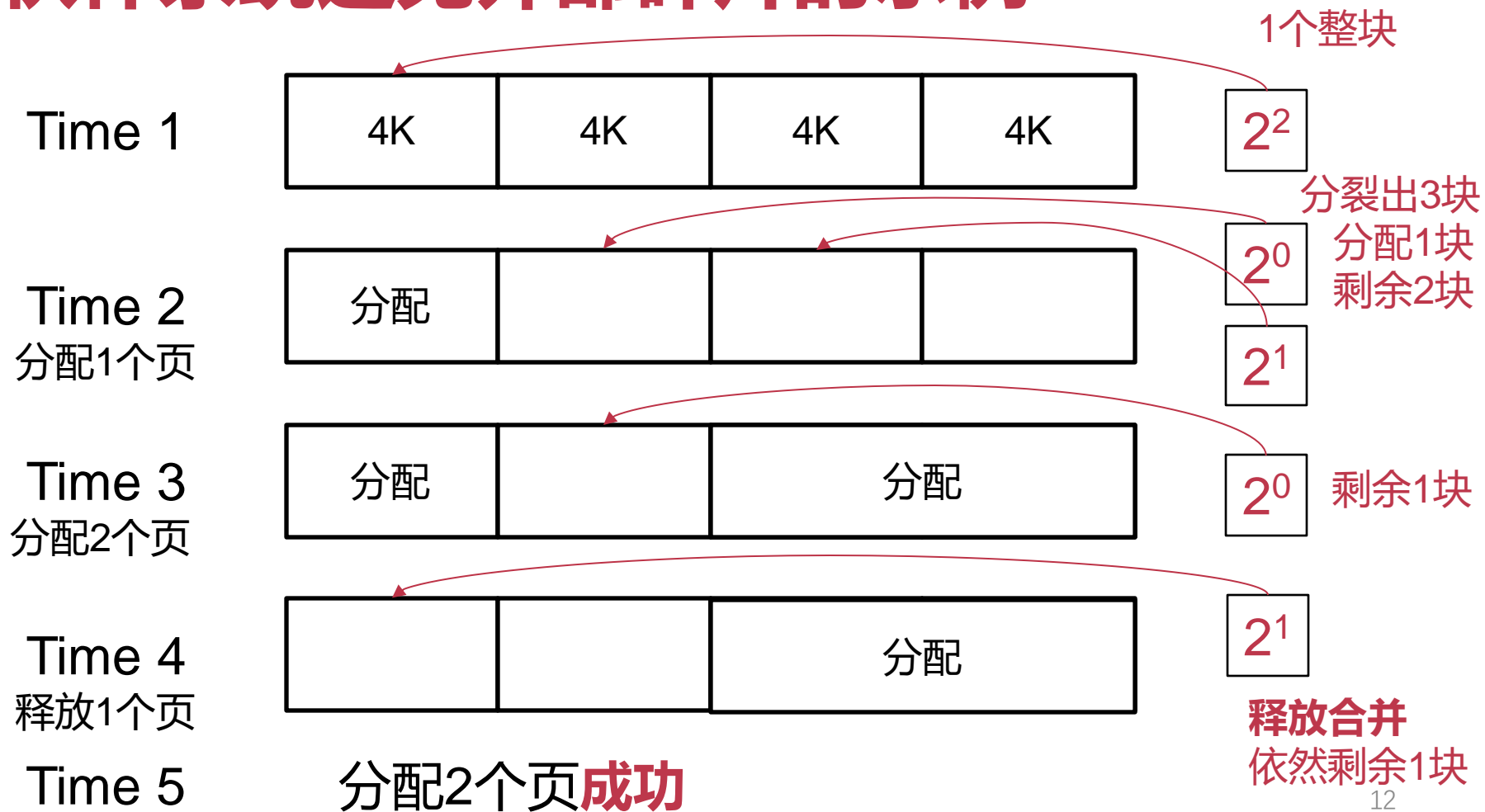
- 高效地找到伙伴块

- 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 ( $2^n$ )
  - 直接映射物理页的物理地址与虚拟地址
- 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？**

# 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
  - **思考：分配的时间复杂度？**  $O(1)$   $O(\text{list-num})$
  - **思考：合并的时间复杂度？**  $O(1)$   $O(\text{list-num})$

# 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

步骤一

```
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    }
```

步骤二

```
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    }
```

步骤三

```
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 }
```



# 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;
10         ↪ ++cur_order) {
11         free_list = &(free_lists[cur_order]);
12         if (!list_empty(free_list)) {
13             // 从空闲链表中取出一个伙伴块
14             page = get_one_entry(free_list);
15             break;
16         }
17     }
18
19     // 若取出伙伴块大于所需大小, 则进行分裂
20     page = split_page(order, page);
21     // 标记已分配。示意代码忽略分配失败的情况
22     page->allocated = 1;
23 }
24
```

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/SLOB：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
- 数据结构大小往往小于页粒度

# 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
  - 发展过程中，提出针对内存稀缺场景的SLOB
- **后续以主流的SLUB为例讲解**

# SLUB分配器的思路

- **观察**

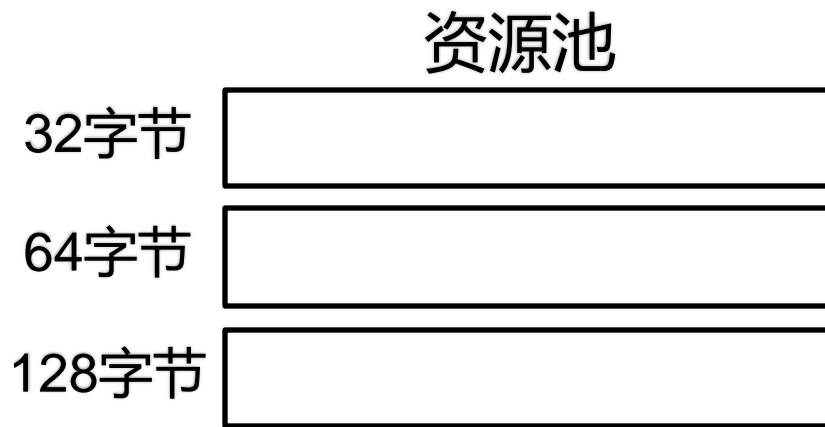
- 操作系统频繁分配的对象大小相对比较固定

- **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,  $3 \leq n < 12$ ）
- 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

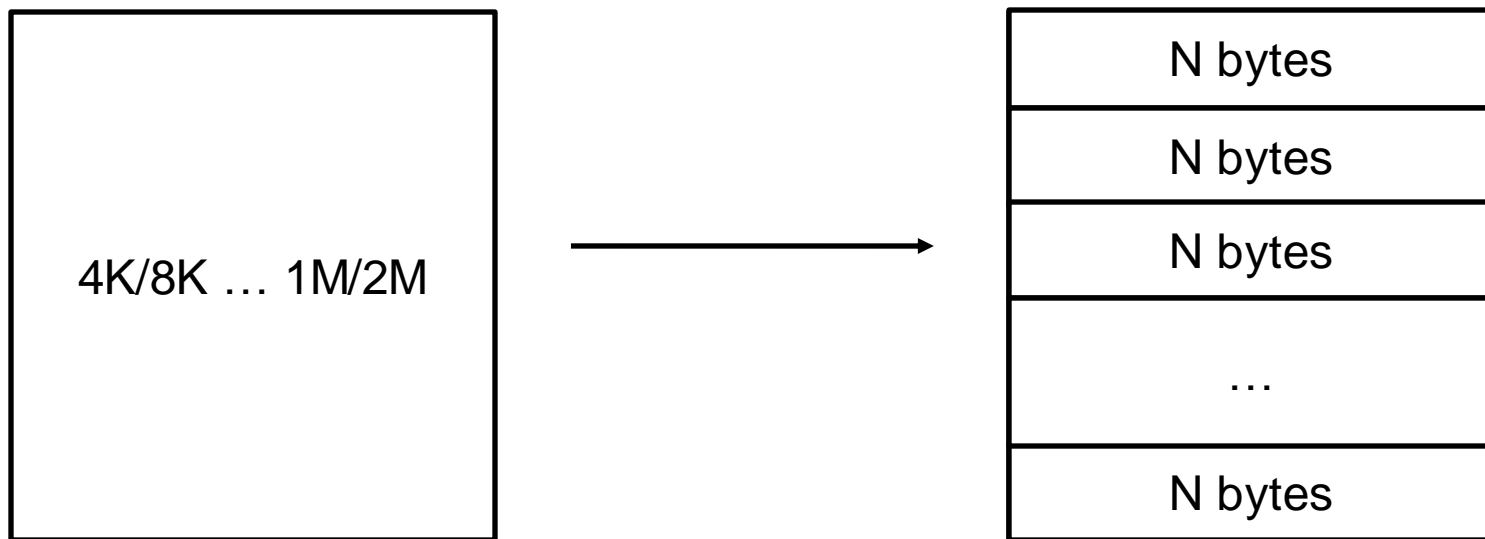
# SLUB设计

- 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

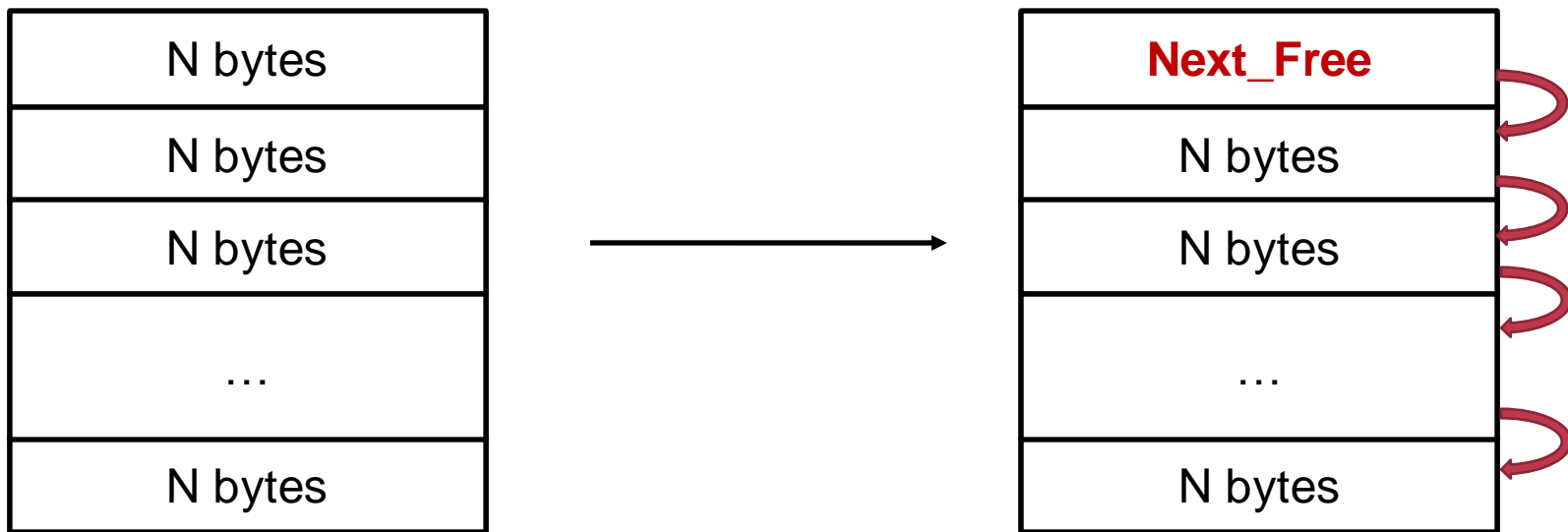
把从伙伴系统得到的连续物理页**划分成若干等份** (slot)





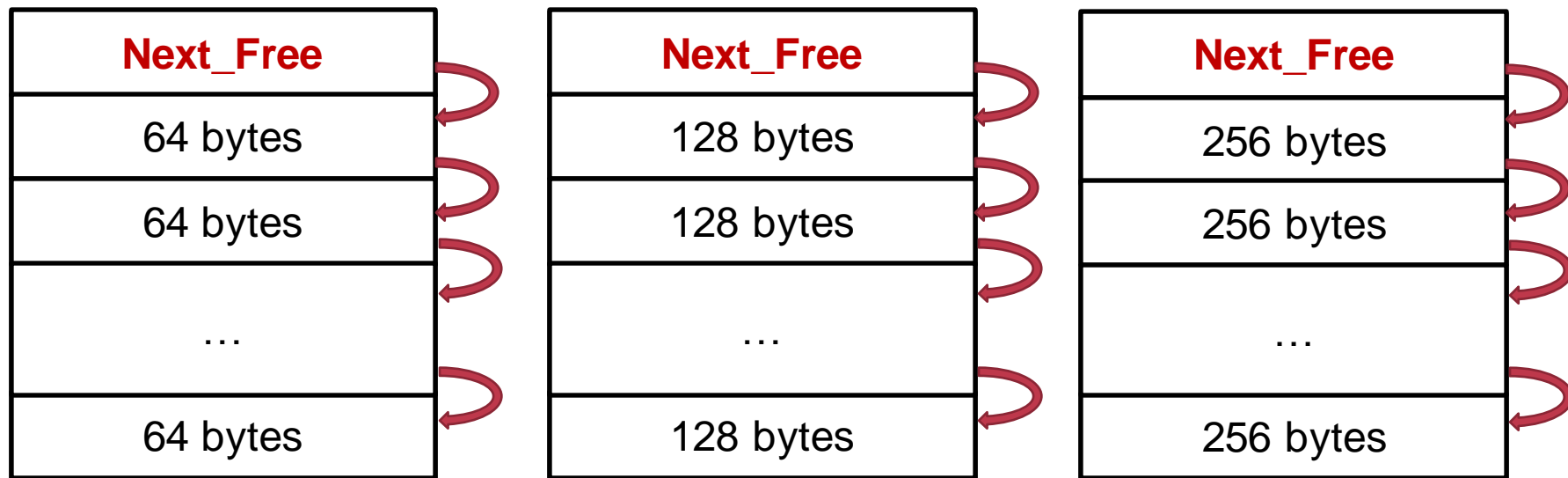
# SLUB设计：空闲链表

当分配时直接分配一个空闲slot：  
**如何区分是否空闲？** 采用空闲链表



# SLUB设计：分配

分配N字节时，首先找到大小最合适的SLAB，  
取走Next\_Free指向的第一个；释放时直接放回Next\_Free后



# SLUB设计：释放

## 释放时如何找到Next\_Free?

提示：SLAB的大小是固定的

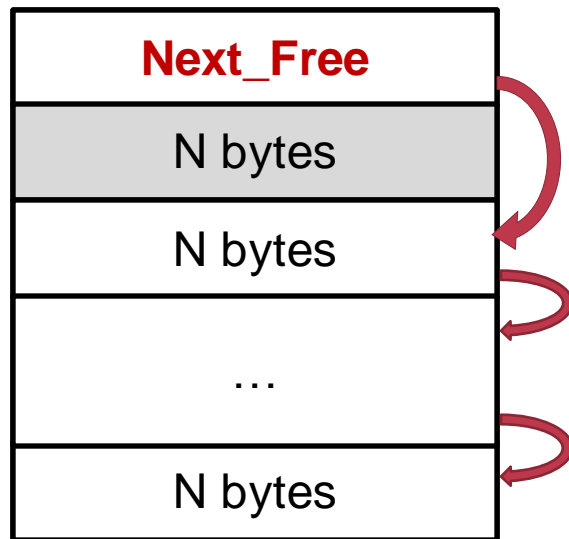
思路：根据object地址计算SLAB起始地址  
 $\text{ADDR} \& \sim(\text{SLAB\_SIZE}-1)$

**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**

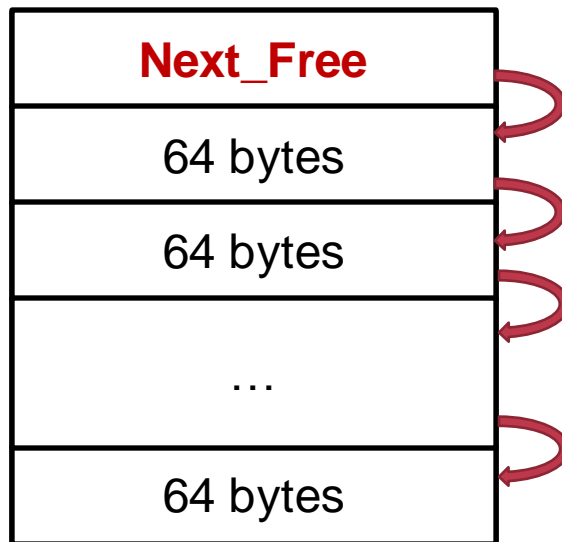
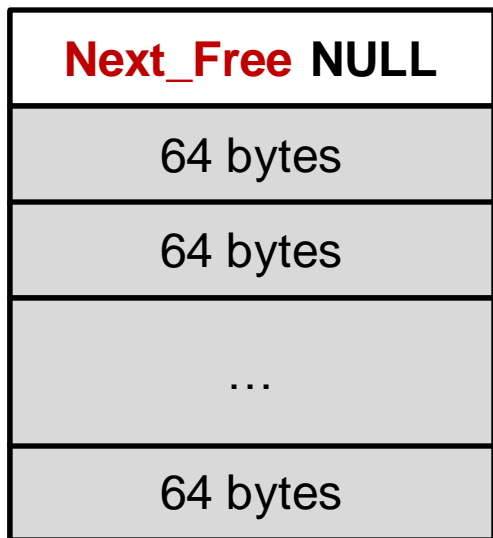
## SLAB



```
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;  
};
```

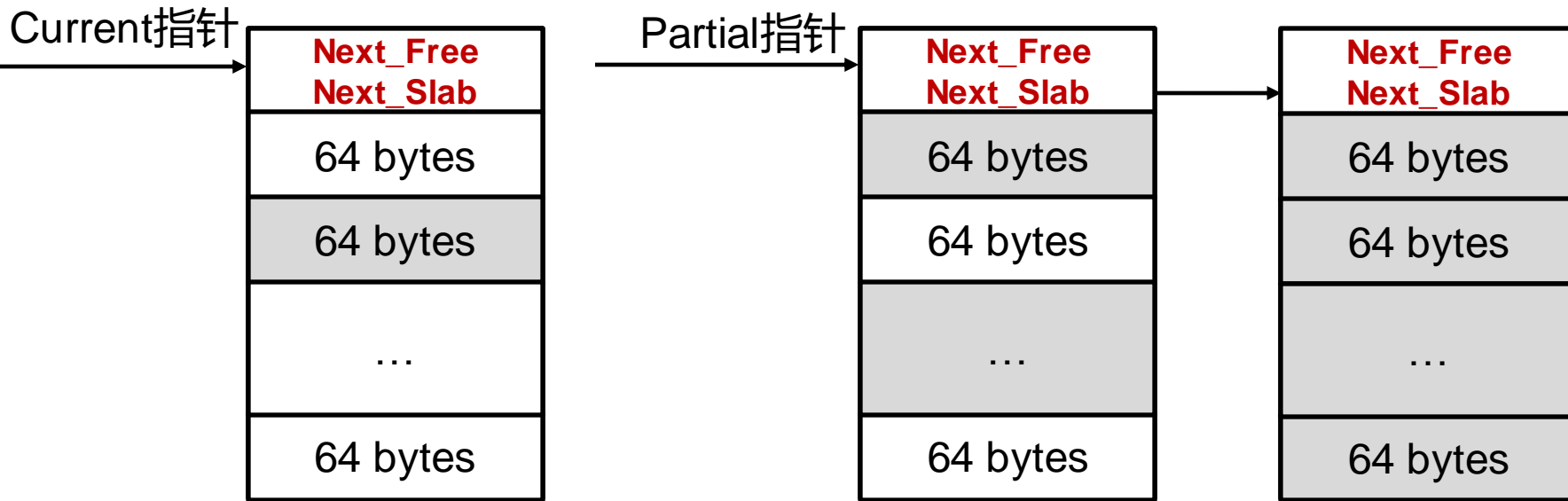
# SLUB设计：新增SLAB

当64字节slot的SLAB已经分配完怎么办？  
再从伙伴系统分配一个SLAB



# SLUB设计：组织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**；  
Current和Partial之间的SLAB移动

**释放**到对应的SLAB；  
若某个SLAB全free则可还给伙伴系统

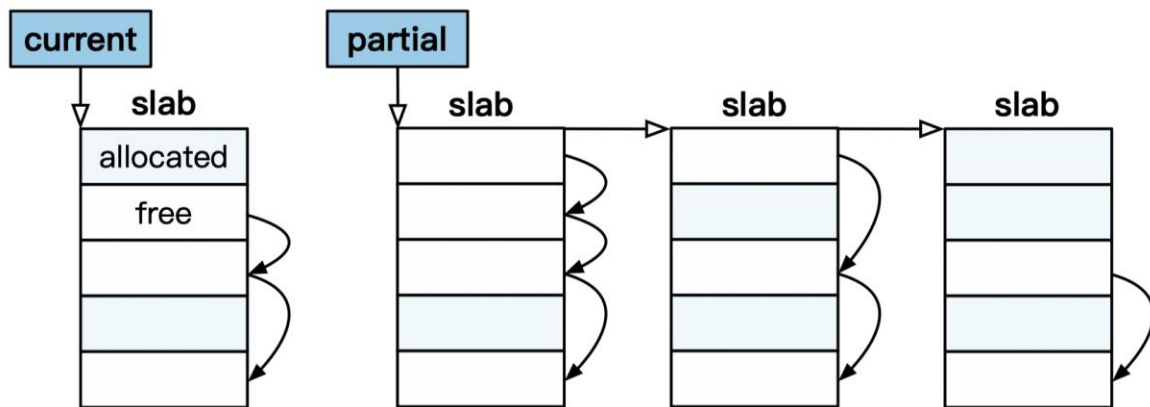
# SLUB小结

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指向未满足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)

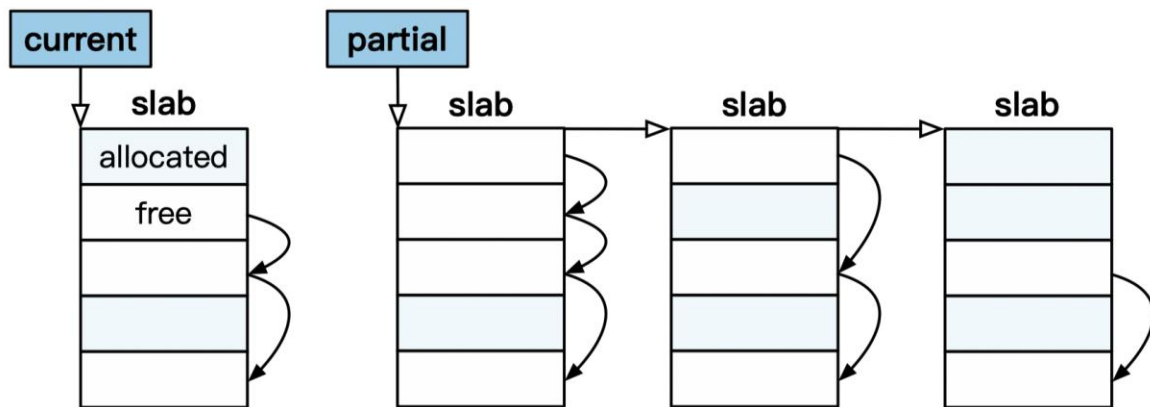
针对每种slot大小维护两个指针:

- current仅指向一个 slab
  - 分配时使用、按需更新
- partial指向未满足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?



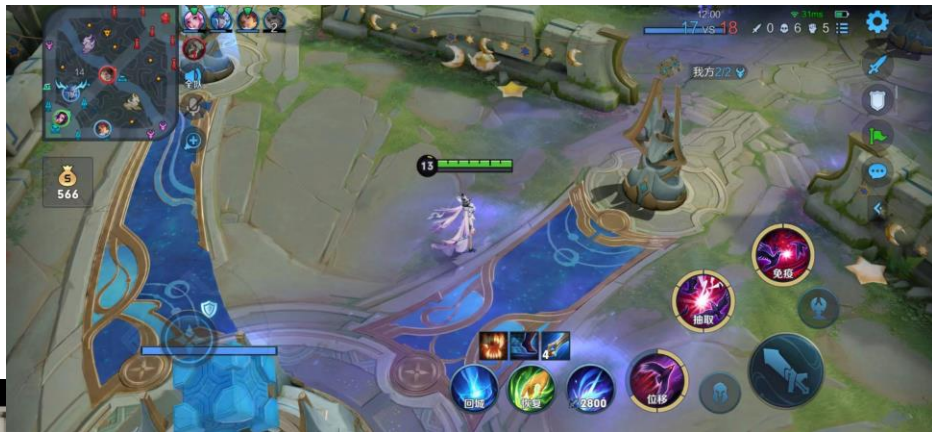
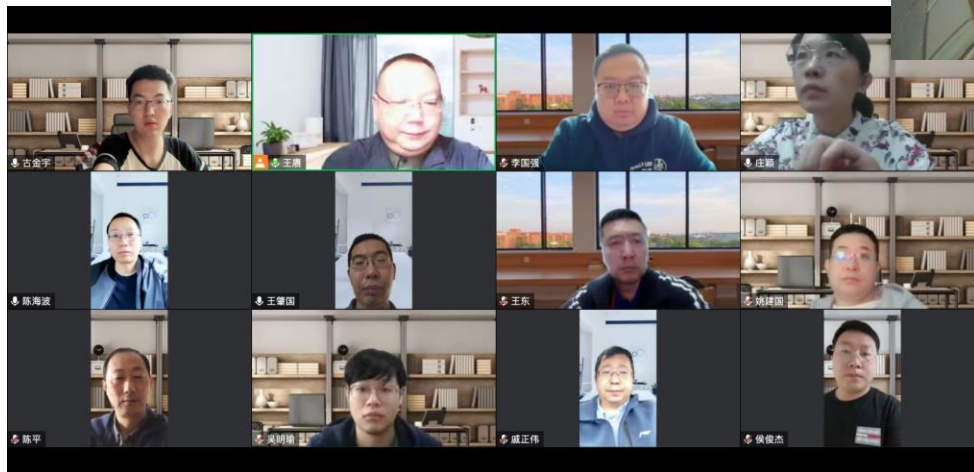
# 突破物理内存容量限制



# 场景-4：物理内存容量<应用进程需求

手机总内存大小6GB

腾讯会议占用2GB



王者荣耀占用6GB

# 换页机制 (Swapping)

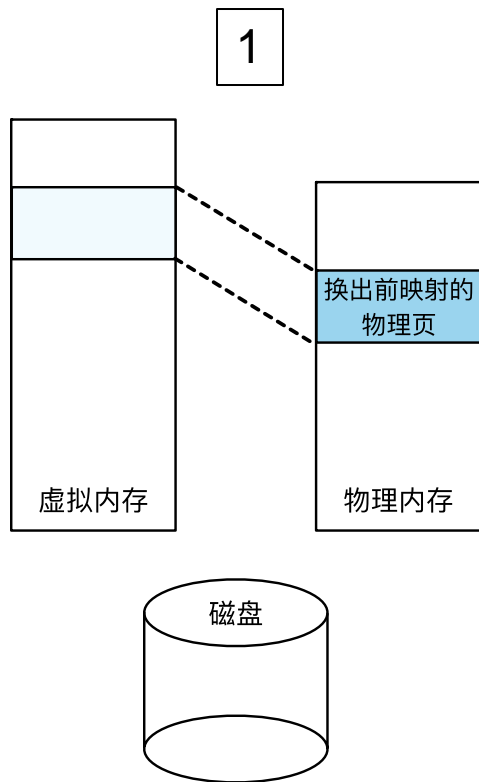
- **换页的基本思想**

- 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
    - 即节约内存又能减少缺页异常次数

## 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



## 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

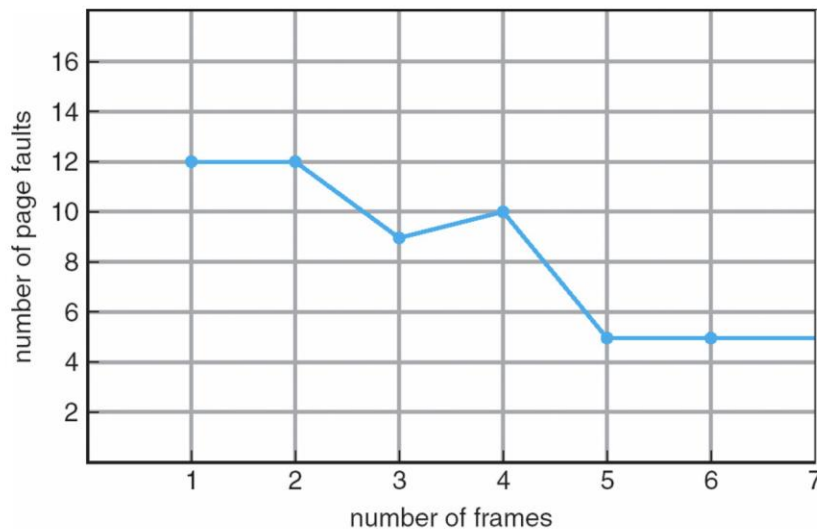
# FIFO策略

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

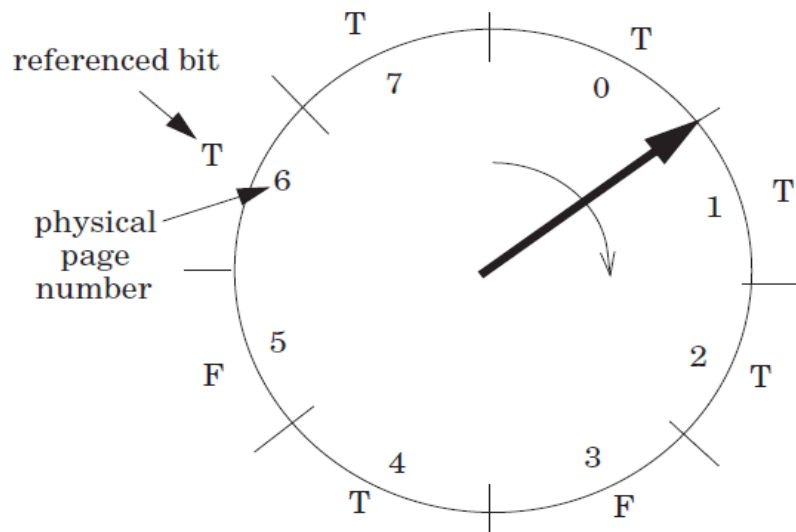
缺点-1：对于特定的序列，效果可能非常差，如循环访问内存

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



# 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**

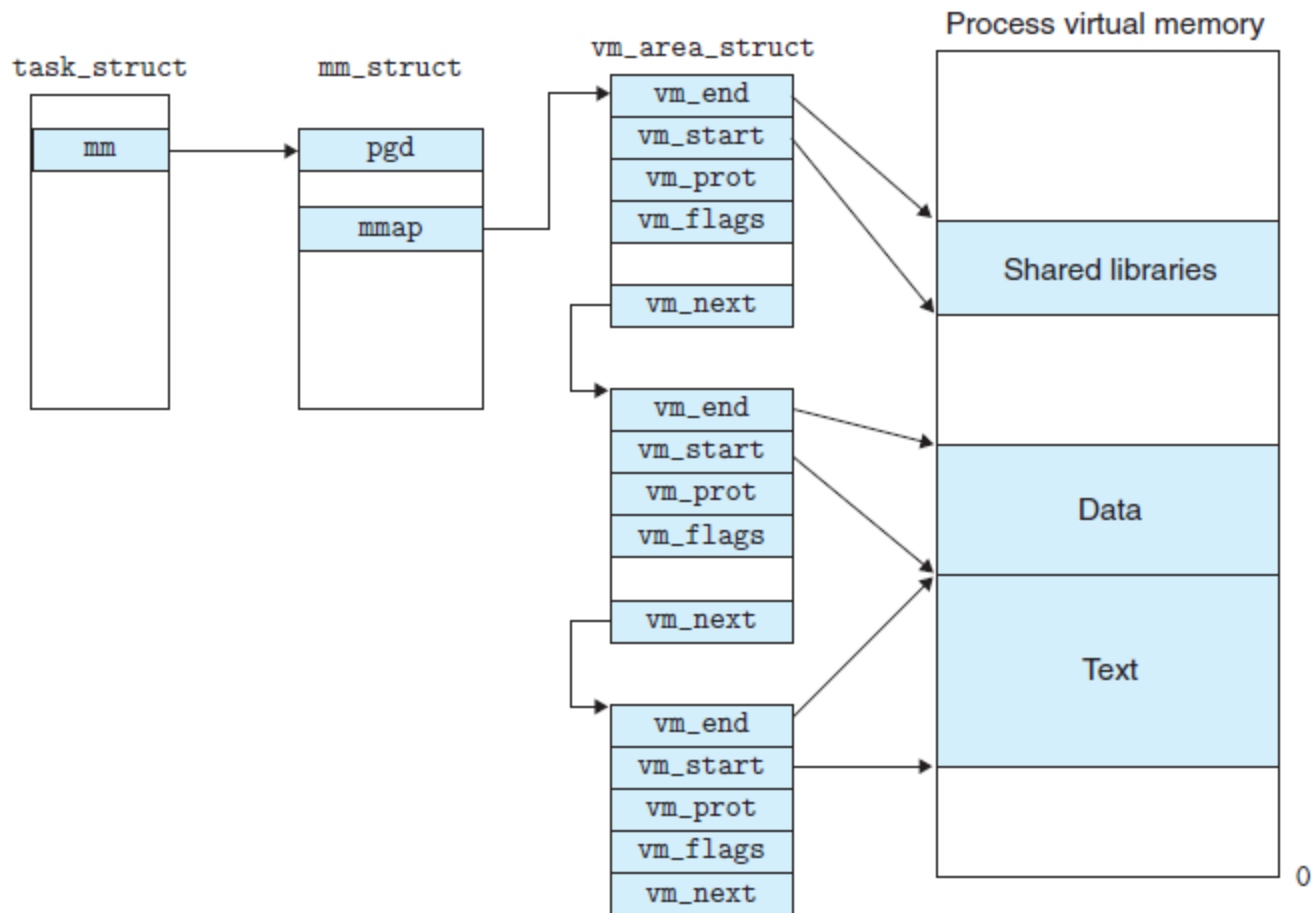
- 等待磁盘 I/O 导致 CPU 利用率下降
- 调度器载入更多的进程以期提高 CPU 利用率
- 触发更多的缺页异常、进一步降低 CPU 利用率、导致连锁反应

# 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

# ▶ LINUX案例分析

# 虚拟内存抽象的核心数据结构



# 进程虚拟地址空间 mm\_struct (图)

- **mmap**

- 虚拟内存区域链表
- 包含多个虚拟内存区域

- **mmap\_base**

- 内存映射区基地址

- **pgd**

- 指向页表的目录

```
struct mm_struct {
    struct vm_area_struct * mmap;           /* list of VMAs, 指向线性区对象的链表头部 */
    struct rb_root mm_rb;                  /* 指向线性区对象的红黑树 */
    struct vm_area_struct * mmap_cache;     /* last find_vma result 指向最近找到的虚拟区间 */
#ifdef CONFIG_MMU

    /* 用来在进程地址空间中搜索有效的进程地址空间的函数 */

    unsigned long (*get_unmapped_area) (struct file *filp,
                                         unsigned long addr, unsigned long len,
                                         unsigned long pgoff, unsigned long flags);

    /* 释放线性区的调用方法 */
    void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
#endif

    unsigned long mmap_base;                /* base of mmap area , 内存映射区的基地址 */
    unsigned long task_size;                /* size of task vm space */
    unsigned long cached_hole_size;         /* if non-zero, the largest hole below free_area_cache */
    unsigned long free_area_cache;         /* first hole of size cached_hole_size or larger */
    pgd_t * pgd;                           /* 页表目录指针 */
    atomic_t mm_users;                     /* How many users with user space?, 共享进程的个数 */
    atomic_t mm_count;                     /* How many references to "struct mm_struct" (users co
使用计数器, 采用引用计数, 描述有多少指针指向当前的mm_struct */
    int map_count;                          /* number of VMAs , 线性区个数 */
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;            /* Protects page tables and some counters, 保护页表和引
(使用的自旋锁) */
```

\*部分代码

# 进程虚拟地址区域 vm\_area\_struct

- vm\_next, vm\_prev: 前后指针
- vm\_start: 起始位置
- vm\_end: 结束位置
- vm\_mm: 所属mm\_struct
- pgprot\_t: 访问权限

```
struct vm_area_struct {
    unsigned long vm_start;      /* Our start address within vm_mm. */
    unsigned long vm_end;        /* The first byte after our end address within vm_mm. */
    struct vm_area_struct *vm_next, *vm_prev;
    struct rb_node vm_rb;
    unsigned long rb_subtree_gap;
    struct mm_struct *vm_mm;      /* The address space we belong to. */
    pgprot_t vm_page_prot;        /* Access permissions of this VMA. */
    unsigned long vm_flags;       /* Flags, see mm.h. */
    union {
        struct {
            struct rb_node rb;
            unsigned long rb_subtree_last;
        } shared;
        const char __user *anon_name;
    };
    struct list_head anon_vma_chain;
    struct anon_vma *anon_vma;
    const struct vm_operations_struct *vm_ops;
    unsigned long vm_pgoff;
    struct file * vm_file;
    void * vm_private_data;
    atomic_long_t swap_readahead_info;
#ifdef CONFIG_MMU
    struct vm_region *vm_region;  /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;  /* NUMA policy for the VMA */
#endif
    struct vm_userfaultfd_ctx vm_userfaultfd_ctx;
#ifdef CONFIG_SPECULATIVE_PAGE_FAULT
    seqcount_t vm_sequence;       /* Speculative page fault field */
    atomic_t vm_ref_count;        /* see vma_get(), vma_put() */
#endif
} __randomize_layout;
```

# 进程创建流程

- **第一步：创建虚拟地址空间vm space**
- **第二步：加载代码和数据**
- **第三步：创建堆**
- **第四步：创建主线程**
  - 分配栈空间（栈的mm\_struct和相应的pgd）



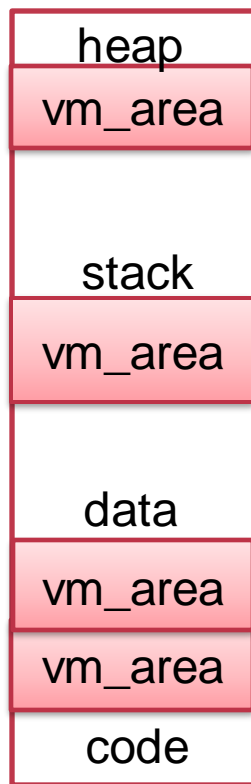
# 进程地址空间创建

第一步: mm\_struct

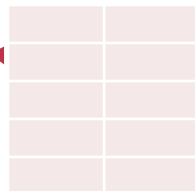
第三步: 创建堆

第四步: 创建栈

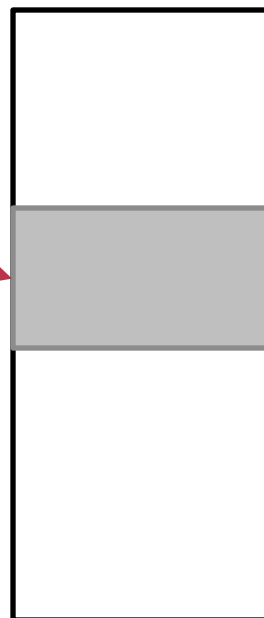
第二步:  
加载代码和数据



页表



物理内存



# 示例：为用户进程创建地址空间

- 创建并且初始化mm

```
struct mm_struct *mm_alloc(void)
{
    struct mm_struct *mm;

    mm = allocate_mm();
    if (!mm)
        return NULL;

    memset(mm, 0, sizeof(*mm));
    return mm_init(mm, current, current_user_ns());
}
```

# 初始化mm\_struct

- 将vm\_area\_struct联系到对应的mm\_struct



```
static inline void vma_init(struct vm_area_struct *vma, struct mm_struct *mm)
{
    static const struct vm_operations_struct dummy_vm_ops = {};

    memset(vma, 0, sizeof(*vma));
    vma->vm_mm = mm;
    vma->vm_ops = &dummy_vm_ops;
    INIT_LIST_HEAD(&vma->anon_vma_chain);
}
```

- 一些域的初始化



```
static struct mm_struct *mm_init(struct mm_struct *mm, struct task_struct *p,
    struct user_namespace *user_ns)
{
    mt_init_flags(&mm->mm_mt, MM_MT_FLAGS);
    mt_set_external_lock(&mm->mm_mt, &mm->mmap_lock);
    atomic_set(&mm->mm_users, 1);
    atomic_set(&mm->mm_count, 1);
    seqcount_init(&mm->write_protect_seq);
    mmap_init_lock(mm);
    INIT_LIST_HEAD(&mm->mmlist);
    mm_pgtables_bytes_init(mm);
    mm->map_count = 0;
    mm->locked_vm = 0;
    atomic64_set(&mm->pinned_vm, 0);
    memset(&mm->rss_stat, 0, sizeof(mm->rss_stat));
    spin_lock_init(&mm->page_table_lock);
    spin_lock_init(&mm->arg_lock);
    mm_init_cpumask(mm);
    mm_init_aio(mm);
    mm_init_owner(mm, p);
    mm_pasid_init(mm);
    RCU_INIT_POINTER(mm->exe_file, NULL);
    mmu_notifier_subscriptions_init(mm);
    init_tlb_flush_pending(mm);
}
```

\*部分代码

# 示例：为第一个用户线程创建栈

- 为虚拟地址空间分配栈



```
static int alloc_thread_stack_node(struct task_struct *tsk, int node)
{
    struct vm_struct *vm;
    void *stack;
    int i;
    for (i = 0; i < NR_CACHED_STACKS; i++) {
        struct vm_struct *s;
        s = this_cpu_xchg(cached_stacks[i], NULL);
        if (!s)
            continue;
        kasan_unpoison_range(s->addr, THREAD_SIZE);
        stack = kasan_reset_tag(s->addr);
        if (memcg_charge_kernel_stack(s)) {
            vfree(s->addr);
            return -ENOMEM;
        }
        tsk->stack_vm_area = s;
        tsk->stack = stack;
        return 0;
    }
    stack = __vmalloc_node_range(THREAD_SIZE, THREAD_ALIGN,
                                VMALLOC_START, VMALLOC_END,
                                THREADINFO_GFP & ~__GFP_ACCOUNT,
                                PAGE_KERNEL,
                                0, node, __builtin_return_address(0));

    if (!stack)
        return -ENOMEM;
    vm = find_vm_area(stack);
    if (memcg_charge_kernel_stack(vm)) {
        vfree(stack);
        return -ENOMEM;
    }
    tsk->stack_vm_area = vm;
    stack = kasan_reset_tag(stack);
    tsk->stack = stack;
    return 0;
}
```

# 分配页全局目录

```
static inline int mm_alloc_pgd(struct mm_struct *mm)
{
    mm->pgd = pgd_alloc(mm);
    if (unlikely(!mm->pgd))
        return -ENOMEM;
    return 0;
}
```

- 虚拟地址空间指向对应页全局目录

# 将虚拟地址映射到地址空间中

- `kvm_riscv_gstage_map`

1. 定义vma
2. 初始化vma

```
615 int kvm_riscv_gstage_map(struct kvm_vcpu *vcpu,
616                          struct kvm_memory_slot *memslot,
617                          gpa_t gpa, unsigned long hva, bool is_write)
618 {
619     int ret;
620     kvm_pfn_t hfn;
621     bool writable;
622     short vma_pageshift;
623     gfn_t gfn = gpa >> PAGE_SHIFT;
624     struct vm_area_struct *vma;
625     struct kvm *kvm = vcpu->kvm;
626     struct kvm_mmu_memory_cache *pcache = &vcpu->arch.mmu_page_cache;
627     bool logging = (memslot->dirty_bitmap &&
628                    !(memslot->flags & KVM_MEM_READONLY)) ? true : false;
629     unsigned long vma_pagesize, mmu_seq;
630
631     /* We need minimum second+third level pages */
632     ret = kvm_mmu_topup_memory_cache(pcache, gstage_pgd_levels);
633     if (ret) {
634         kvm_err("Failed to topup G-stage cache\n");
635         return ret;
636     }
637
638     mmap_read_lock(current->mm);
639
640     vma = vma_lookup(current->mm, hva);
641     if (unlikely(!vma)) {
642         kvm_err("Failed to find VMA for hva 0x%lx\n", hva);
643         mmap_read_unlock(current->mm);
644         return -EFAULT;
645     }
646
647     if (is_vm_hugetlb_page(vma))
648         vma_pageshift = huge_page_shift(hstate_vma(vma));
649     else
650         vma_pageshift = PAGE_SHIFT;
651     vma_pagesize = 1ULL << vma_pageshift;
```

# 将虚拟地址映射到地址空间中

- `kvm_riscv_gstage_map`

1. 定义vma
2. 初始化vma
3. 计算全局页帧号
4. 转换物理页框号,并确定页的保护位

```
651 vma_pagesize = 1ULL << vma_pageshift;
652 if (logging || (vma->vm_flags & VM_PFNMAP))
653     vma_pagesize = PAGE_SIZE;
654
655 if (vma_pagesize == PMD_SIZE || vma_pagesize == PUD_SIZE)
656     gfn = (gpa & huge_page_mask(hstate_vma(vma))) >> PAGE_SHIFT;
657
658 /*
659  * Read mmu_invalidate_seq so that KVM can detect if the results of
660  * vma_lookup() or gfn_to_pfn_prot() become stale prior to acquiring
661  * kvm->mmu_lock.
662  *
663  * Rely on mmap_read_unlock() for an implicit smp_rmb(), which pairs
664  * with the smp_wmb() in kvm_mmu_invalidate_end().
665  */
666 mmu_seq = kvm->mmu_invalidate_seq;
667 mmap_read_unlock(current->mm);
668
669 if (vma_pagesize != PUD_SIZE &&
670     vma_pagesize != PMD_SIZE &&
671     vma_pagesize != PAGE_SIZE) {
672     kvm_err("Invalid VMA page size 0x%lx\n", vma_pagesize);
673     return -EFAULT;
674 }
675
676 hfn = gfn_to_pfn_prot(kvm, gfn, is_write, &writable);
677 if (hfn == KVM_PFN_ERR_HWPOISON) {
678     send_sig_mceerr(BUS_MCEERR_AR, (void __user *)hva,
679                    vma_pageshift, current);
680     return 0;
681 }
```

# 将虚拟地址映射到地址空间中

- `kvm_riscv_gstage_map`

1. 定义vma
2. 初始化vma
3. 计算全局页帧号
4. 转换物理页框号,并确定页的保护位
5. 映射到内存页表



```
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715

    if (logging && !is_write)
        writable = false;

    spin_lock(&kvm->mmu_lock);

    if (mmu_invalidate_retry(kvm, mmu_seq))
        goto out_unlock;

    if (writable) {
        kvm_set_pfn_dirty(hfn);
        mark_page_dirty(kvm, gfn);
        ret = gstage_map_page(kvm, pcache, gpa, hfn << PAGE_SHIFT,
                               vma_pagesize, false, true);
    } else {
        ret = gstage_map_page(kvm, pcache, gpa, hfn << PAGE_SHIFT,
                               vma_pagesize, true, true);
    }

    if (ret)
        kvm_err("Failed to map in G-stage\n");

out_unlock:
    spin_unlock(&kvm->mmu_lock);
    kvm_set_pfn_accessed(hfn);
    kvm_release_pfn_clean(hfn);
    return ret;
}
```



# 示例：如何加入新页表

```
/* Page Table entry */  
typedef struct {  
    unsigned long pte;  
} pte_t;
```

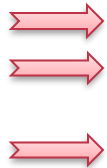
- 调用页表映射函数；根据页面大小计算对应的页表级别；pte\_t 是页表项类型

```
177 static int gstage_map_page(struct kvm *kvm,  
178                             struct kvm_mmu_memory_cache *pcache,  
179                             gpa_t gpa, phys_addr_t hpa,  
180                             unsigned long page_size,  
181                             bool page_rdonly, bool page_exec)  
182 {  
183     int ret;  
184     u32 level = 0;  
185     pte_t new_pte;  
186     pgprot_t prot;  
187  
188     ret = gstage_page_size_to_level(page_size, &level);  
189     if (ret)  
190         return ret;
```

# 加入新页表

- 将给定的物理页帧号和访问权限转换为一个页表项(pte)，并将其设置为脏页。设置页表中指定虚拟地址对应的页表项为新的页表项。

```
205         if (page_exec) {
206             if (page_rdonly)
207                 prot = PAGE_READ_EXEC;
208             else
209                 prot = PAGE_WRITE_EXEC;
210         } else {
211             if (page_rdonly)
212                 prot = PAGE_READ;
213             else
214                 prot = PAGE_WRITE;
215         }
216         new_pte = pfn_pte(PFN_DOWN(hpa), prot);
217         new_pte = pte_mkdirty(new_pte);
218
219         return gstage_set_pte(kvm, level, pcache, gpa, &new_pte);
220     }
```



# Cont. 加入新页表


- 使用循环结构遍历从根页表到叶节点页表，并逐一查找每个页表级别对应的页表项。

- 如果发现某个页表项为空（即未映射到物理页），则分配一个页表项，并将其物理地址设置为下一级页表的地址。
- 如果页表项不为空，即已经映射到物理页，则将下一级页表项的虚拟地址赋值给 next\_ptep。

```
137 static int gstage_set_pte(struct kvm *kvm, u32 level,
138                          struct kvm_mmu_memory_cache *pcache,
139                          gpa_t addr, const pte_t *new_pte)
140 {
141     u32 current_level = gstage_pgd_levels - 1;
142     pte_t *next_ptep = (pte_t *)kvm->arch.pgd;
143     pte_t *ptep = &next_ptep[gstage_pte_index(addr, current_level)];
144
145     if (current_level < level)
146         return -EINVAL;
147
148     while (current_level != level) {
149         if (gstage_pte_leaf(ptep))
150             return -EEXIST;
151
152         if (!pte_val(*ptep)) {
153             if (!pcache)
154                 return -ENOMEM;
155             next_ptep = kvm_mmu_memory_cache_alloc(pcache);
156             if (!next_ptep)
157                 return -ENOMEM;
158             *ptep = pfn_pte(PFN_DOWN(__pa(next_ptep)),
159                             __pgprot(_PAGE_TABLE));
160         } else {
161             if (gstage_pte_leaf(ptep))
162                 return -EEXIST;
163             next_ptep = (pte_t *)gstage_pte_page_vaddr(*ptep);
164         }
165
166         current_level--;
167         ptep = &next_ptep[gstage_pte_index(addr, current_level)];
168     }
```

# 加入新页表

- 将 ptep 指向的页表项更新为 new\_pte 所指向的新的页表项，并检查更新后的页表项是否为叶节点页表项。如果是，刷新远程TLB缓存。



```
170     *ptep = *new_pte;
171     if (gstage_pte_leaf(ptep))
172         gstage_remote_tlb_flush(kvm, current_level, addr);
173
174     return 0;
175 }
```

# 缺页异常

- **基于缺页异常，实现按需分配**
  - 例如，访问按需分配映射的区域会触发缺页
- **(64位) 用户态程序发生缺页时下陷到内核**
  - 在RV64上，通过scause寄存器的值来识别page fault
  - 13: page fault caused by a read
  - 15: page fault caused by a write

```
101 #define EXC_LOAD_PAGE_FAULT 13
102 #define EXC_STORE_PAGE_FAULT 15
```

# 缺页异常处理函数

- 获取触发缺页的地址
  - RV64: htval或stval
- 根据scause中错误状态码具体处理
  - 读取错误
  - 写入错误

```
13 static int gstage_page_fault(struct kvm_vcpu *vcpu, struct kvm_run *run,
14                             struct kvm_cpu_trap *trap)
15 {
16     struct kvm_memory_slot *memslot;
17     unsigned long hva, fault_addr;
18     bool writable;
19     gfn_t gfn;
20     int ret;
21
22     fault_addr = (trap->htval << 2) | (trap->stval & 0x3);
23     gfn = fault_addr >> PAGE_SHIFT;
24     memslot = gfn_to_memslot(vcpu->kvm, gfn);
25     hva = gfn_to_hva_memslot_prot(memslot, gfn, &writable);
26
27     if (kvm_is_error_hva(hva) ||
28         (trap->scause == EXC_STORE_GUEST_PAGE_FAULT && !writable)) {
29         switch (trap->scause) {
30             case EXC_LOAD_GUEST_PAGE_FAULT:
31                 return kvm_riscv_vcpu_mmio_load(vcpu, run,
32                                                  fault_addr,
33                                                  trap->htinst);
34             case EXC_STORE_GUEST_PAGE_FAULT:
35                 return kvm_riscv_vcpu_mmio_store(vcpu, run,
36                                                  fault_addr,
37                                                  trap->htinst);
```

# 处理按需分配导致的读取错误

- Bit[0]=1时, 表明该指令是一个转换指令或自定义指令。
- Bit[0]=0时, 表明该指令的值是零或者是一个特殊值。

```
453 int kvm_riscv_vcpu_mmio_load(struct kvm_vcpu *vcpu, struct kvm_run *run,
454                             unsigned long fault_addr,
455                             unsigned long htinst)
456 {
457     u8 data_buf[8];
458     unsigned long insn;
459     int shift = 0, len = 0, insn_len = 0;
460     struct kvm_cpu_trap utrap = { 0 };
461     struct kvm_cpu_context *ct = &vcpu->arch.guest_context;
462
463     /* Determine trapped instruction */
464     if (htinst & 0x1) {
465         /*
466          * Bit[0] == 1 implies trapped instruction value is
467          * transformed instruction or custom instruction.
468          */
469         insn = htinst | INSN_16BIT_MASK;
470         insn_len = (htinst & BIT(1)) ? INSN_LEN(insn) : 2;
471     } else {
472         /*
473          * Bit[0] == 0 implies trapped instruction value is
474          * zero or special value.
475          */
476         insn = kvm_riscv_vcpu_unpriv_read(vcpu, true, ct->sepc,
477                                           &utrap);
478         if (utrap.scause) {
479             /* Redirect trap if we failed to read instruction */
480             utrap.sepc = ct->sepc;
481             kvm_riscv_vcpu_trap_redirect(vcpu, &utrap);
482             return 1;
483         }
484         insn_len = INSN_LEN(insn);
485     }
```

# 处理按需分配导致读取错误

- 保存指令解码信息
- 更新MMIO的详细信息
- 进行MMIO读操作，等待I/O模拟器完成操作并返回结果

531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558

```
/* Fault address should be aligned to length of MMIO */  
if (fault_addr & (len - 1))  
    return -EIO;
```

```
/* Save instruction decode info */  
vcpu->arch.mmio_decode.insn = insn;  
vcpu->arch.mmio_decode.insn_len = insn_len;  
vcpu->arch.mmio_decode.shift = shift;  
vcpu->arch.mmio_decode.len = len;  
vcpu->arch.mmio_decode.return_handled = 0;
```

```
/* Update MMIO details in kvm_run struct */  
run->mmio.is_write = false;  
run->mmio.phys_addr = fault_addr;  
run->mmio.len = len;
```

```
/* Try to handle MMIO access in the kernel */  
if (!kvm_io_bus_read(vcpu, KVM_MMIO_BUS, fault_addr, len, data_buf)) {  
    /* Successfully handled MMIO access in the kernel so resume */  
    memcpy(run->mmio.data, data_buf, len);  
    vcpu->stat.mmio_exit_kernel++;  
    kvm_riscv_vcpu_mmio_return(vcpu, run);  
    return 1;  
}
```

```
/* Exit to userspace for MMIO emulation */  
vcpu->stat.mmio_exit_user++;  
run->exit_reason = KVM_EXIT_MMIO;
```



# 总结

