Lecture 10
Linux Memory Management

Hui Wang, Yitao
Zheng, Jianuo Zhu

Fall 2025

Linux 地址空间基础结构

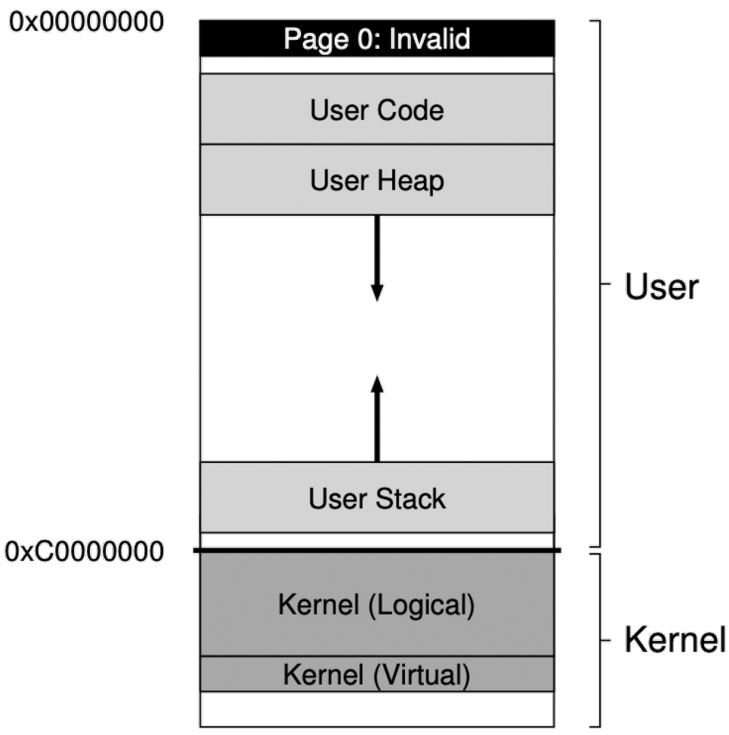
- 每个进程的虚拟地址空间，会被分成「用户空间」和「内核空间」两部分。用户空间的地址范围是 0x00000000 ~ 0xBFFFFFFF

- 注意：Page 0（地址0开头的页）是“无效”的 —— 这是为了检测空指针（NULL）：如果程序误访问了 NULL 指针（对应地址0），会直接触发错误，避免非法操作。

- 用户空间里包含：代码段（User Code）、堆（User Heap，向上生长）、栈（User Stack，向下生长）。

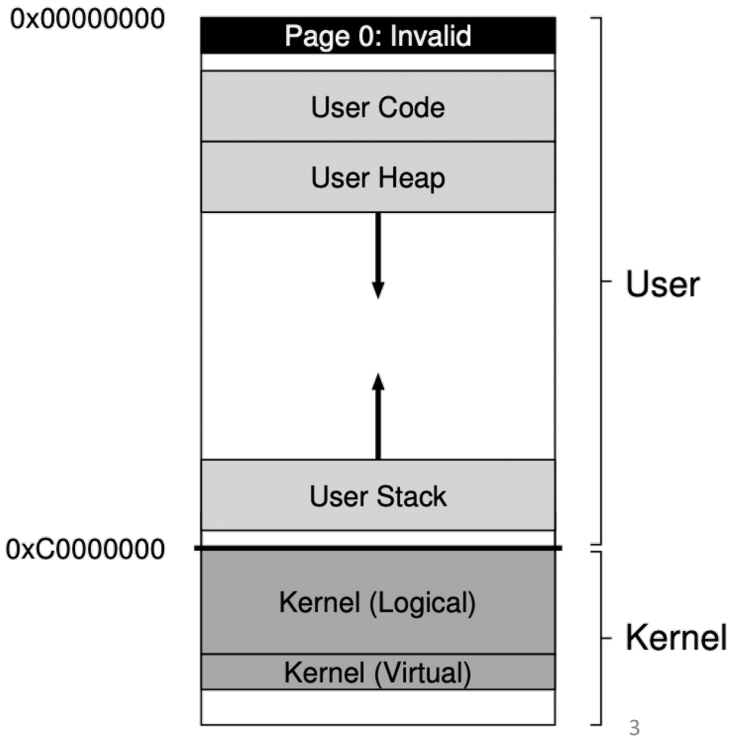
- 内核空间的地址范围是 0xC0000000 ~ 0xFFFFFFFF。这部分是内核专属的虚拟地址空间，包含内核的逻辑地址、虚拟地址等。

- 补充：64 位 Linux 的划分逻辑类似，但分割的地址范围会略有不同。

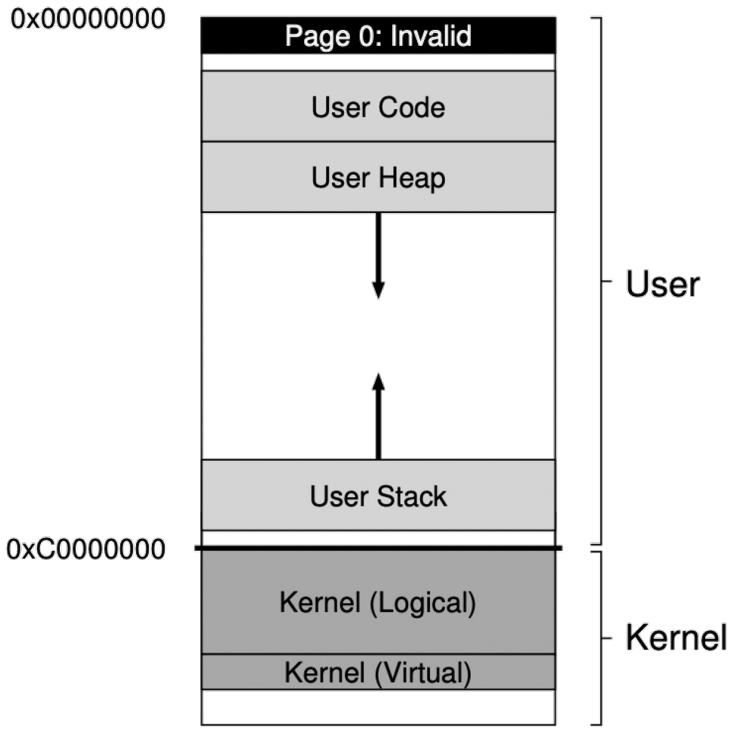


为什么每个进程的地址空间都要映射内核内存

- 好处 1：减少切换开销
- 当进程触发系统调用、中断、异常时，会从用户态陷入内核态。此时不需要切换页表（不用修改 CR3 寄存器），也不用刷新 TLB（快表）—— 因为内核空间已经提前映射到进程地址空间里了，能提升效率。
- 好处 2：方便内核访问用户内存
- 内核代码需要和用户空间交互时（比如系统调用传递数据），可以直接访问用户空间的内存，不用额外拷贝。
- 关键特性：所有进程的内核空间是“共享同一份”的
- 虽然每个进程都能看到内核空间，但内核空间的内容是全局统一的（不是每个进程有独立内核空间）。

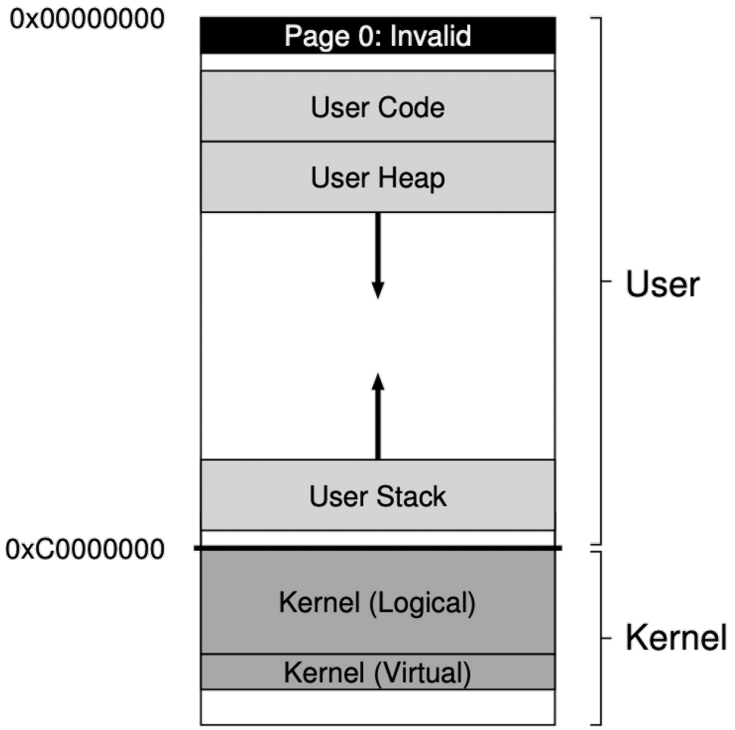


内核空间的两种地址类型

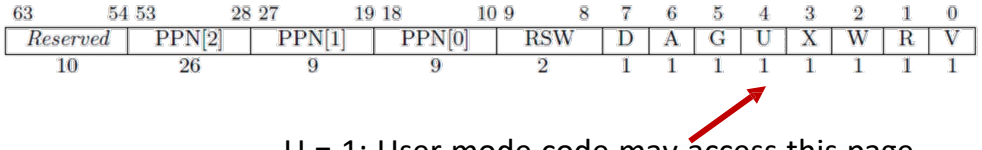


内核逻辑地址（Kernel Logical）
用途：存放内核核心数据结构（页表、进程的内核栈），通过 kmalloc() 分配，不会被换出内存。
地址特点：从 0xC0000000 开始，虚拟地址和物理地址是连续对应的（物理地址从 0x00000000 开始，虚拟地址 = 物理地址 +0xC0000000）。
优势：适合 DMA（直接内存访问）或需要连续物理内存的设备（因为地址是连续映射的）。

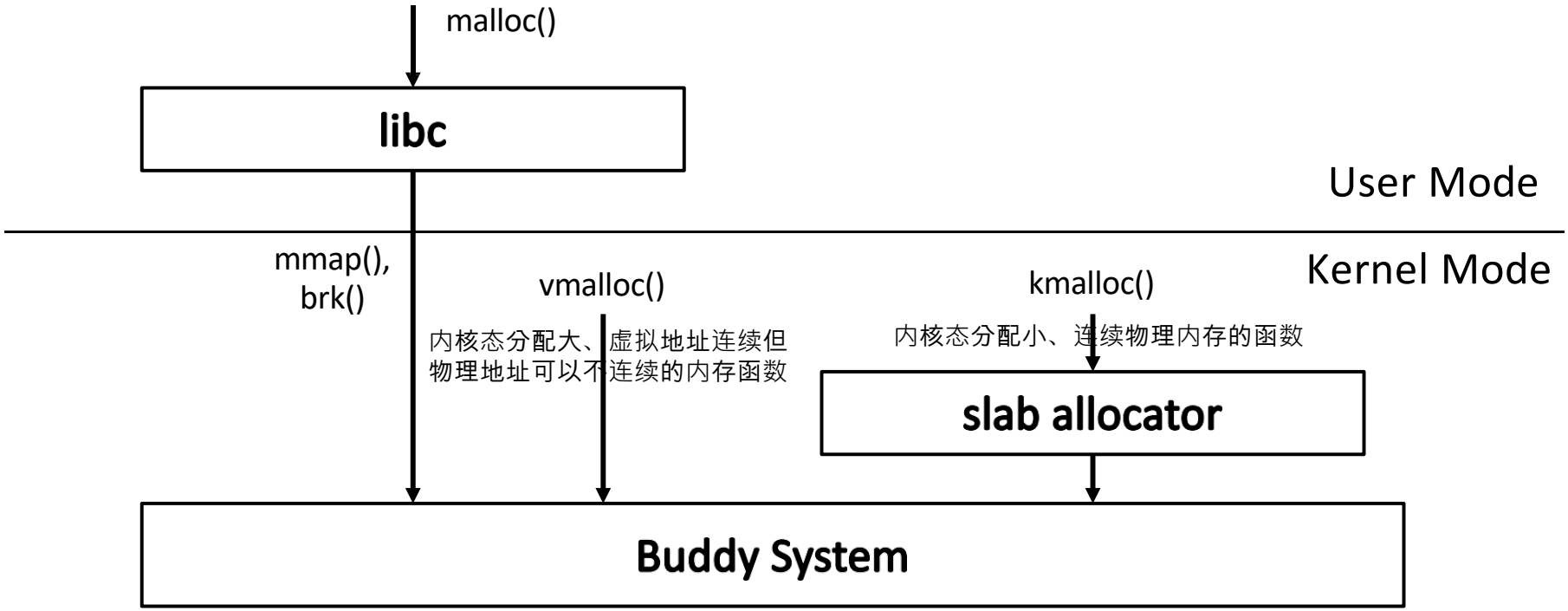
内核虚拟地址（Kernel Virtual）
用途：通过 vmalloc() 分配，是虚拟地址连续、但物理地址可以不连续的内存。
场景：用于分配大内存块（物理内存可能碎片化时，虚拟地址可以“拼接”不连续的物理页）。



- 进程间隔离：每个进程的用户空间是独立的（地址空间不同），所以进程之间不会互相干扰。
- 用户与内核的隔离：靠页表的权限位实现（PPT 里的页表项中 U 位）：
- U=1：用户态代码可以访问该页；
- U=0：只有内核态代码能访问该页。
- 内核空间对应的页表项会把 U 设为 0，所以用户进程无法直接访问内核空间，保证了内核的安全性。



Linux Physical Memory Management



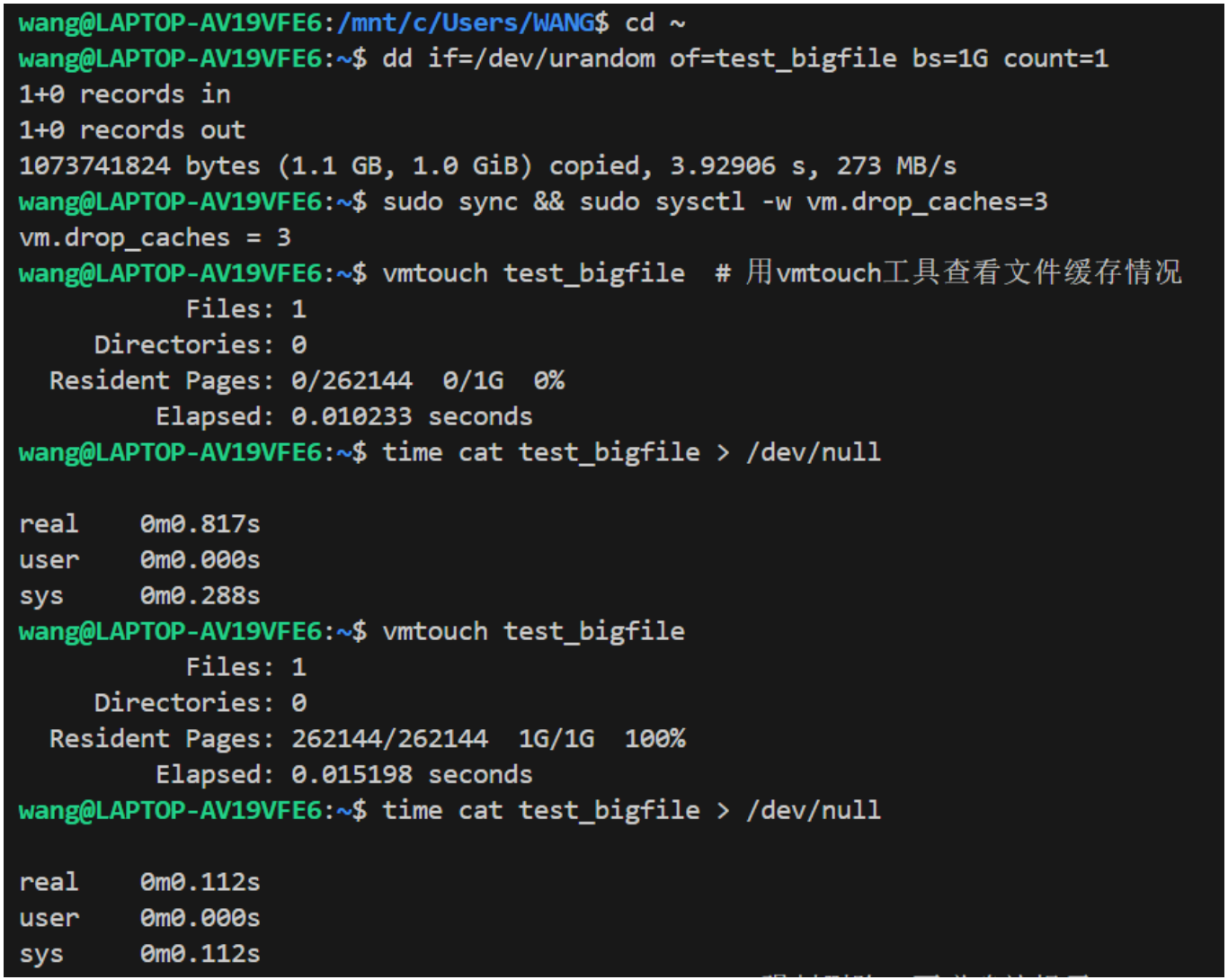
操作系统的内存管理，首先要解决‘怎么把物理内存分出去’的问题——buddy system是底层基础，负责把物理内存切成页框、按需求分配；而内核经常需要小内存，直接用伙伴系统会浪费，所以有了slab allocator，专门优化内核小内存的分配。

分出去的内存要‘用得高效’——页缓存就是一个典型场景：我们把磁盘数据缓存到从伙伴系统分配来的物理页框里，让文件读写速度接近内存速度，这是内存‘高效利用’的体现。

页缓存

定义：页缓存（Page Cache）是 Linux 内核中专门用来加速文件 I/O 的核心机制，它会把磁盘上的文件数据，按照系统 “内存页” 的大小（默认 4KB），缓存到物理内存里。

核心价值：解决内存和磁盘的 “速度鸿沟”—— 内存的访问速度是纳秒级（10^{-9}秒），而磁盘是毫秒级（10^{-3} 秒），差了整整 6 个数量级。页缓存通过 “数据复用”，避免了频繁的磁盘读写，让文件操作速度直接翻倍。



生成了 1 个 1GB 的大文件。
第一次读这个文件：数据从硬盘搬到内存，记了时间。
第二次读同一个文件：直接从内存里读，再记一次时间。

第二步是直接从内存读取缓存的文件，耗时仅 0.112 秒，是第一次读取耗时的1/7 左右。

读操作流程

缓存命中（Hit）：进程要读文件时，内核先查页缓存 —— 根据文件的 inode（文件唯一标识）和数据偏移量，找对应的缓存页。如果找到了，直接把内存里的数据拷贝给进程，全程不碰磁盘，速度极快。

缓存未命中（Miss）：如果缓存里没有目标数据，内核会先分配一个空闲的物理内存页，然后从磁盘上把对应的数据块读到这个内存页里，再把这个页加入页缓存，最后才把数据拷贝给进程。这样下次再读同一部分数据，就会命中缓存。

写操作流程

进程写文件时，不会直接把数据写到磁盘（太慢），而是先写到页缓存对应的页面里，然后给这个页面打个 “脏标记”—— 表示内存里的数据和磁盘上的不一致了。

内核会用后台守护进程（比如 pdflush、flushers），在合适的时机（比如脏页数量达到阈值、系统空闲时、内存不足时），把 “脏页” 批量写回磁盘。这种 “异步写回” 的方式，避免了每次写操作都等磁盘响应，大幅提升写效率。

关键优化：预读（Read-ahead）

内核会预判进程的访问习惯：比如进程读文件时，通常是顺序读（先读第 1 行，再读第 2 行）。所以当进程只读了 1KB 数据时，内核会主动把后面的 4KB、8KB 数据一起读到页缓存里。这样后续进程再读后面的内容时，直接命中缓存，不用再触发磁盘 I/O。

```c
struct folio *__filemap_get_folio(struct address_space *mapping, pgoff_t index,
                fgf_t fgp_flags, gfp_t gfp)
{
        struct folio *folio;

repeat:
        folio = filemap_get_entry(mapping, index);
        if (xa_is_value(folio))
                folio = NULL;
        if (!folio)
                goto no_page;

        if (fgp_flags & FGP_LOCK) {
                if (fgp_flags & FGP_NOWAIT) {
                        if (!folio_trylock(folio)) {
                                folio_put(folio);
                                return ERR_PTR(-EAGAIN);
                        }
                } else {
                        folio_lock(folio);
                }

                /* Has the page been truncated? */
                if (unlikely(folio->mapping != mapping)) {
                        folio_unlock(folio);
                        folio_put(folio);
                        goto repeat;
                }
                VM_BUG_ON_FOLIO(!folio_contains(folio, index), folio);
        }

        if (fgp_flags & FGP_ACCESSED)
                folio_mark_accessed(folio);
        else if (fgp_flags & FGP_WRITE) {
                /* Clear idle flag for buffer write */
                if (folio_test_idle(folio))
                        folio_clear_idle(folio);
        }

        if (fgp_flags & FGP_STABLE)
                folio_wait_stable(folio);
```

```
struct folio *__filemap_get_folio(...) {
    struct folio *folio;

repeat:
    // 1. 先查缓存：根据文件标识（mapping）和页索引（index）找folio
    folio = filemap_get_entry(mapping, index);
    if (!folio) goto no_page;  // 没找到，走“缓存未命中”流程

    // 2. 找到缓存：加锁、校验有效性（避免文件被删除/修改）
    if (要加锁) {  加锁保护folio，防止并发修改
        if (folio已失效) {  比如文件被截断，folio不属于当前文件了
            解锁+释放folio，回到repeat重新查
        }
    }

    // 3. 标记缓存使用状态（比如“已访问”，方便LRU淘汰）
    if (需要标记访问) folio_mark_accessed(folio);
    return folio;  // 缓存命中，返回找到的folio

no_page:  // 缓存未命中：分配新folio并加入缓存
    if (允许创建新folio) {
        计算folio大小（按文件配置的最小/最大页组）
        分配新的物理内存folio（按指定内存分配策略gfp）
        初始化folio（标记访问状态、是否缓存等）
        把新folio加入文件的缓存管理（mapping）中
        if (添加成功) return 新folio;  // 新folio已就绪，返回
        else 释放分配的folio，重试或返回错误
    }
    return 错误（没找到且不能创建）;
}
```

1. 第一步：先查缓存（对应“缓存命中”）
内核先通过 filemap_get_entry(mapping, index) 找缓存：
mapping = 文件唯一标识（对应 inode），告诉内核“找哪个文件的缓存”；
index = 数据页索引（对应文件偏移量），告诉内核“找这个文件的第几页”。
找到有效 folio（缓存页组）→ 加锁保护（防止别人同时改）、标记“已访问”（方便后续 LRU 算法判断是否保留缓存）→ 返回 folio，**缓存命中成功**。
2. 第二步：没找到缓存（对应“缓存未命中”）
如果没找到 folio，就走 no_page 分支：
按文件的配置，分配一块新的物理内存（folio）；
把新 folio 加入这个文件的缓存管理（mapping），相当于“把磁盘数据后续要读入的内存空间先准备好”；
分配成功后返回新 folio，后续内核会把磁盘数据读到这个 folio 里→ **完成缓存加载**。

```
ssize_t filemap_read(...) {
    struct file_ra_state *ra = &filp->f_ra;  // 预读状态管理（之前讲的预读优化）
    struct folio_batch fbatch;  // 批量管理缓存页（提高效率）
    folio_batch_init(&fbatch);  // 初始化缓存页批量容器

    do {
        // 1. 核心步骤1：获取缓存页（命中则直接拿，未命中则读磁盘加载）
        error = filemap_get_pages(..., &fbatch, ...);  // 调用之前聊的页缓存查询逻辑
        if (error < 0) break;  // 失败直接返回

        // 2. 校验文件大小：避免读超文件实际长度
        isize = 文件实际大小;
        if (当前读位置 >= 文件大小) break;

        // 3. 核心步骤2：把缓存页的数据拷贝到用户进程（比如cat命令的进程）
        for (每个缓存页folio) {
            标记folio“已访问”（方便LRU淘汰）;
            处理数据一致性（比如避免缓存别名）;
            copied = 把folio里的数据拷贝到用户空间;  // 实验中“读数据”的最终步骤
            累计已读字节数、更新当前读位置;
        }

        // 4. 释放缓存页引用，准备下一轮读取
        释放fbatch里的所有folio;
        folio_batch_init(&fbatch);
    } while (还有数据要读 && 没到文件尾 && 没错误);

    // 5. 记录读位置，为预读优化提供依据
    ra->prev_pos = 最后读位置;
    return 已读总字节数;
}
```

1. 第一件事：获取缓存页
（filemap_get_pages()）—— 决定 “走磁盘还是走缓存”

2. 第二件事：数据拷贝
不管是 “磁盘读进来的 folio” 还是 “缓存里的 folio”，最终都要把数据拷贝给用户进程（比如cat命令）：

3. 第三件事：预读优化
函数里的ra（read-ahead 预读状态）是个 “隐藏优化”：它会记录这次的读位置（ra->prev_pos = last_pos），判断你是不是 “顺序读”。下次再读这个文件时，filemap_get_pages() 会根据ra的状态，主动预读后面的缓存页 。

```
ssize_t generic_perform_write(...) {
    struct file *file = 要写的文件;
    loff_t pos = 当前写位置;
    struct address_space *mapping = 文件的缓存管理标识;
    size_t chunk = 每次最多写一个缓存页组的大小;
    ssize_t written = 已写字节数;

    do {
        struct folio *folio;  // 要写的缓存页
        size_t bytes = 本次要写的字节数;

        // 1. 准备缓存页：获取/分配要写的folio
        status = a_ops->write_begin(..., &folio, ...);  // 关键：拿到缓存页
        if (status < 0) break;

        // 2. 数据拷贝：把用户进程的写数据，拷贝到缓存页（内存里）
        copied = copy_folio_from_iter_atomic(folio, 偏移量, bytes, 数据来源);
        处理数据一致性（避免缓存错乱）;

        // 3. 收尾：标记脏页+完成写缓存
        status = a_ops->write_end(..., folio, ...);  // 关键：标记脏页
        if (status < 0) break;

        // 4. 更新进度：移动写位置、累计已写字节数
        pos += status;
        written += status;
    } while (还有数据要写);

    更新文件的最终写位置;
    return 已写字节数;
}
```

1. 第一步：拿到 “可写的缓存页（folio）

如果要写的文件位置已经在缓存里（比如之前读过这部分）：直接拿到对应的 folio；如果没在缓存里：分配新的物理内存页→初始化后作为 folio；

2. 第二步：数据从用户进程拷贝到缓存页

把用户进程要写的数据（比如echo “abc” >> file里的 “abc”），从用户空间拷贝到内核的缓存页（folio）里；

特点：“原子拷贝”（atomic）—— 避免拷贝过程中被其他进程干扰，保证数据完整；

3. 第三步：标记脏页 + 收尾

写完数据后做收尾，核心是标记脏页：write_end 内部会调用 set_page_dirty()，给这个 folio 打上 “脏标记”—— 告诉内核：“这个缓存页的数据和磁盘上的不一样了，之后要同步到磁盘”；同时解锁 folio、更新缓存状态，让其他进程能访问这个页；

Linux 现在用的是 **“LRU 链表 + 冷热页分离” 的增强版 LRU**，比基础 LRU 更聪明：

核心逻辑：

内核给每个缓存页维护 “访问时间戳”，并把页分成两个链表：

- **活跃链表：**存 “最近被访问过的热数据”（比如频繁读的数据库索引）；
- **非活跃链表：**存 “很久没被访问的冷数据”（比如昨天读的旧日志）。

淘汰规则：

内存不足时，优先淘汰**非活跃链表尾部**的页（最久没被访问的冷数据）；同时会定期把 “活跃链表中很久没被访问的页” 移到非活跃链表，保证活跃链表只存真·热数据。

为啥不用基础 LRU？

基础 LRU 容易被 “一次性读大文件”（比如读 10GB 日志）冲垮 —— 增强版通过 “冷热分离”，能避免这类临时数据挤走真正的热数据。

一、大页是什么？

默认内存页是 4KB 小页，大页是 更大的内存页（比如 2MB、1GB），大小由硬件（如 x86 架构）支持。

核心价值：减少 CPU 的地址转换开销（TLB Miss）——1 个 2MB 大页能覆盖 512 个 4KB 小页的地址范围，让地址转换更快。

大页能提速，但有内存内部碎片的权衡：比如应用仅需 1KB 内存，却分配了 2MB 大页，剩余的 “2MB-1KB” 内存会被浪费，无法给其他程序使用。

二、Linux 提供的 2 种大页

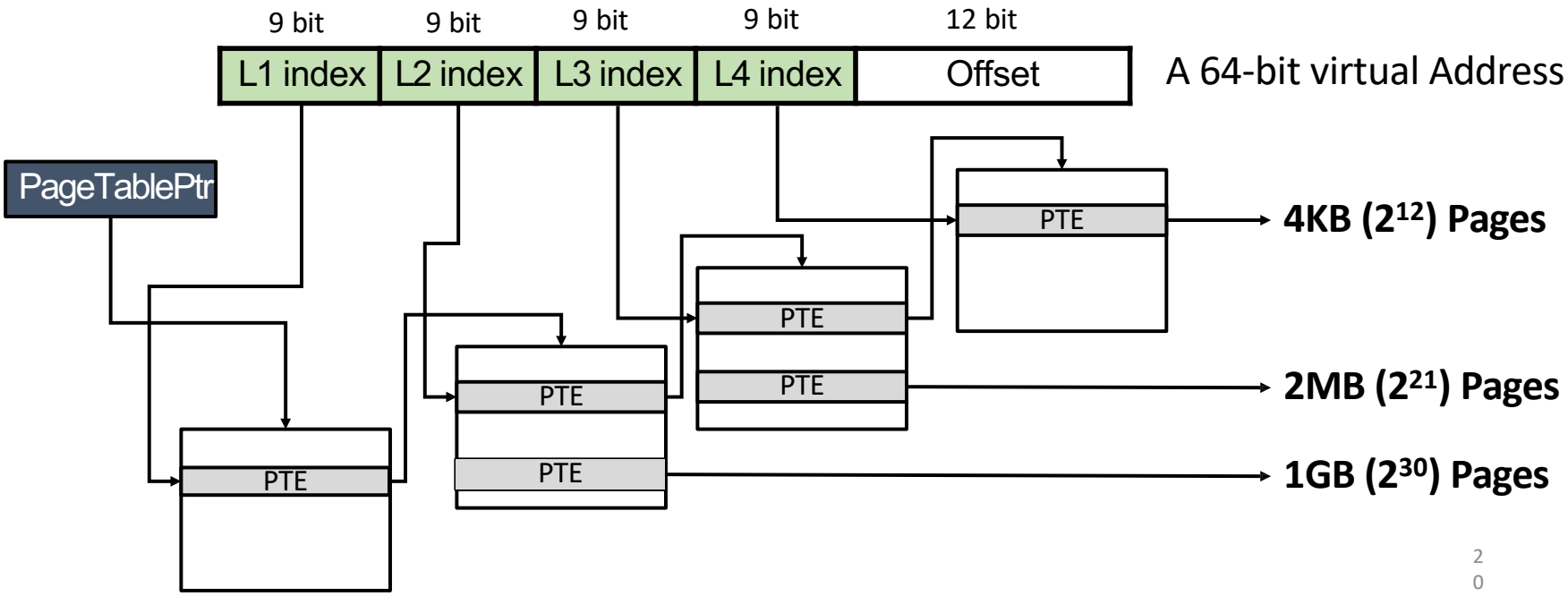
模式	特点	适用场景
透明大页（THP）	内核自动合并 / 拆分大页，应用无感知	普通应用（如 Java 服务），不用改代码享优化
HugeTLBFS（显式大页）	需手动分配大页池，应用显式调用	高性能应用（如 KVM、数据库），精准控制无波动

Large Page Support

- x86 support 4KB, 2MB, 1GB pages
  - Hardware enforces page alignments
  - 4KB pages are 4KB aligned (lower 12 bits are 0)
  - 2MB pages are 2MB aligned (lower 21 bits are 0)
  - 1GB pages are 1GB aligned (lower 30 bits are 0)
- Linux also adds supports to ***huge page*** (Linux term)
  - Fewer TLB misses
  - Applications may need physically continuous physical memory
  - Leads to internal fragmentation

Large Page Support

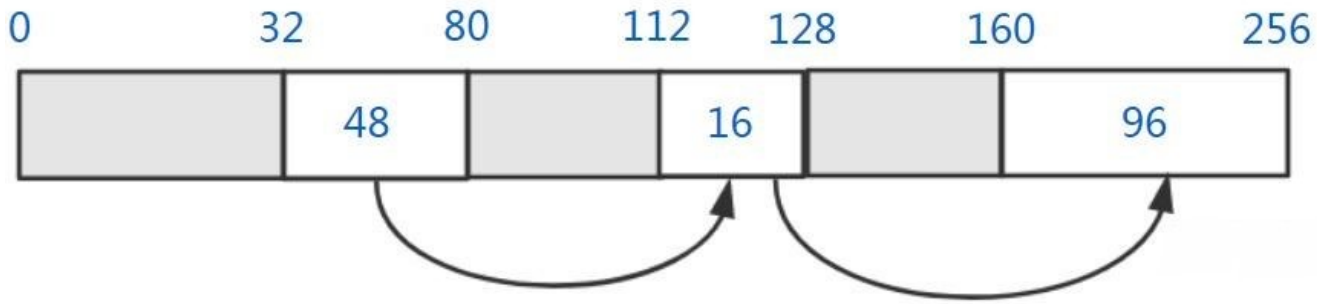
- Different page size uses different level of page tables



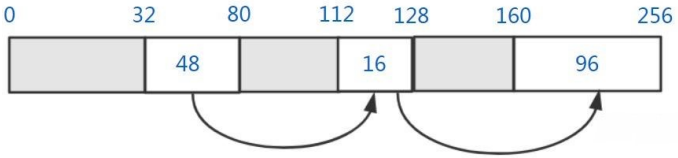
Memory Management Strategies

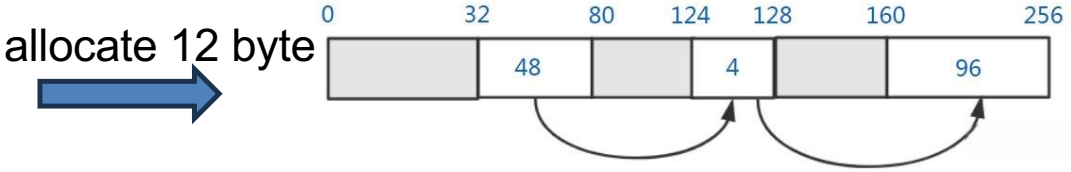
Free List

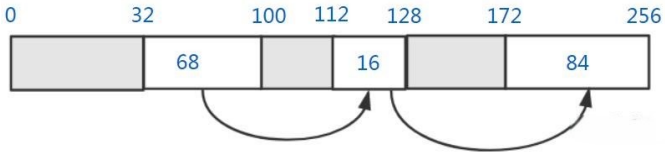
Free lists typically store **pointers** to memory regions that have been released and can be reused.

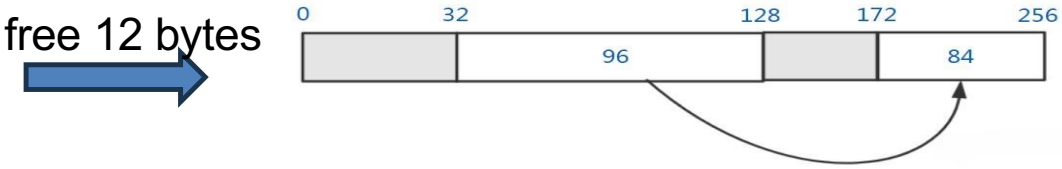


Free List


0
32
80
112
128
160
256
48
16
96

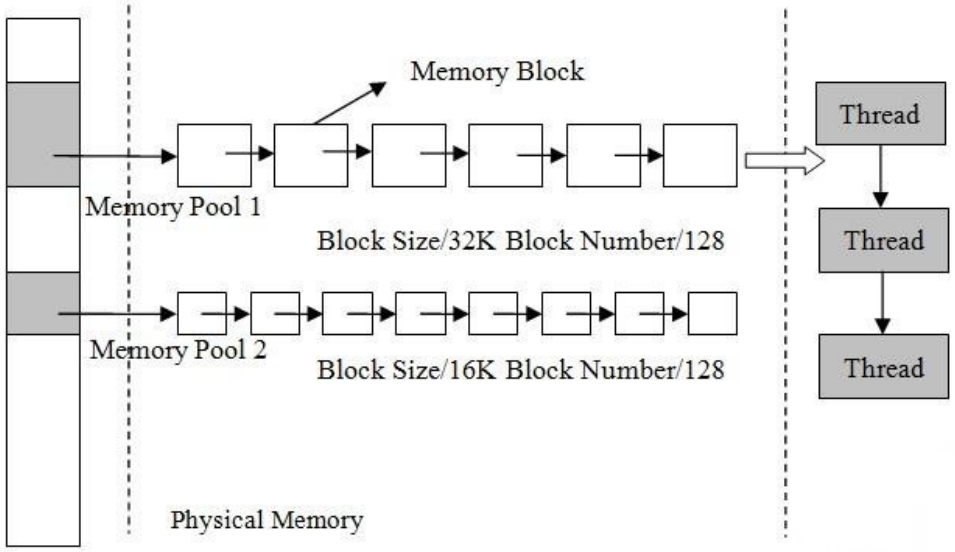

allocate 12 byte
0
32
80
124
128
160
256
48
4
96


0
32
100
112
128
172
256
68
16
84

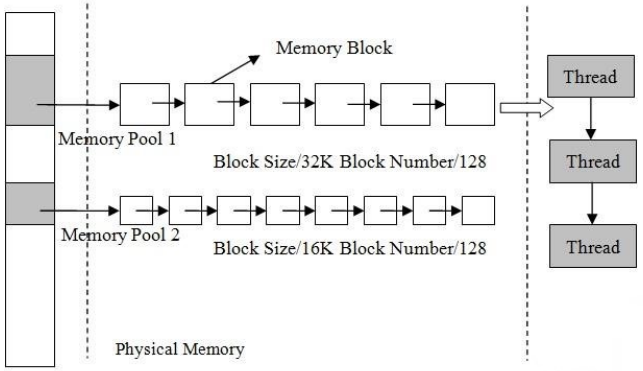

free 12 bytes
0
32
128
172
256
96
84

Memory Pool

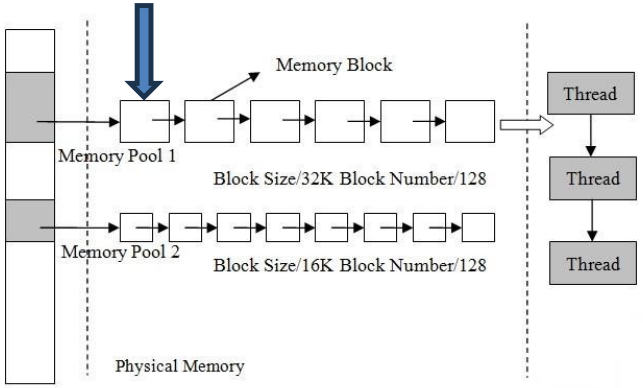
A memory pool is a **pre-allocated region** of memory that is divided into fixed-size or variable-size blocks and managed independently from the system allocator.

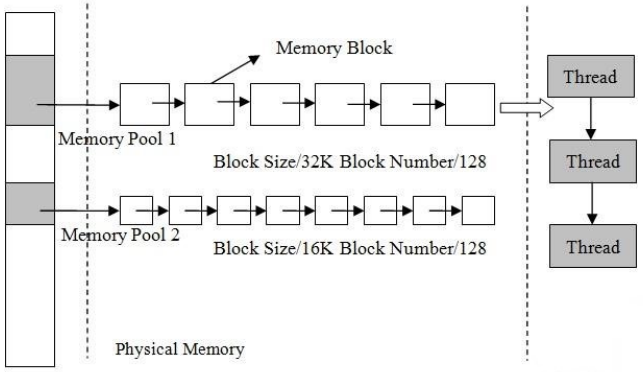


Memory Pool



allocate 30K





allocate 60K

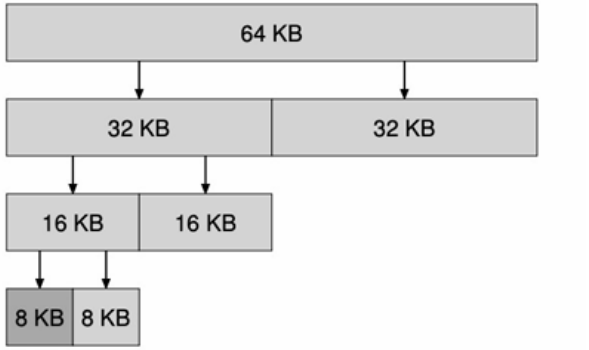
malloc(60K)

Buddy System

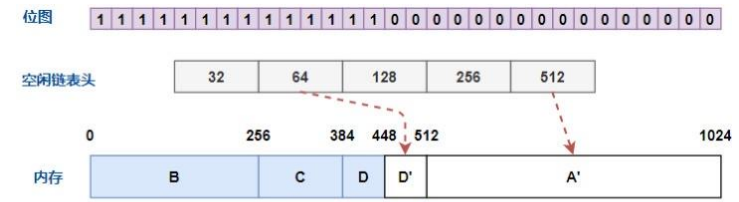
The buddy system is a memory allocation technique that **manages free memory in blocks whose sizes are powers of two**.

When a request arrives, the allocator repeatedly splits a larger block into two equal "buddy" blocks until it finds the smallest block large enough to satisfy the request.

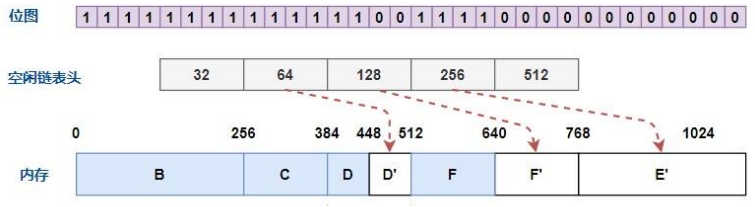
When a block is freed, the allocator attempts to merge it with its unused buddy to form a larger block, reducing external fragmentation.



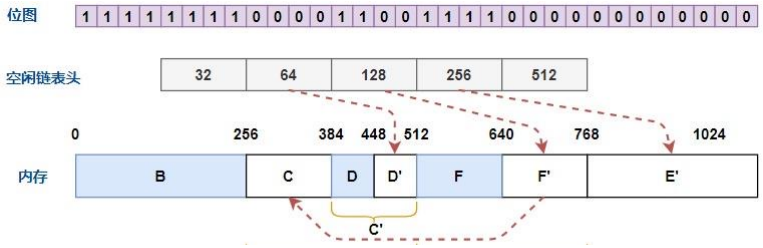
Buddy System



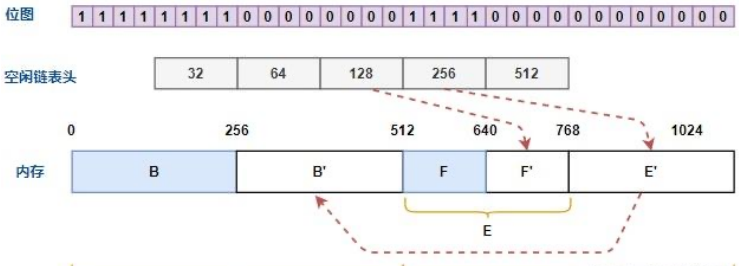
allocate 128 byte



free C



free D

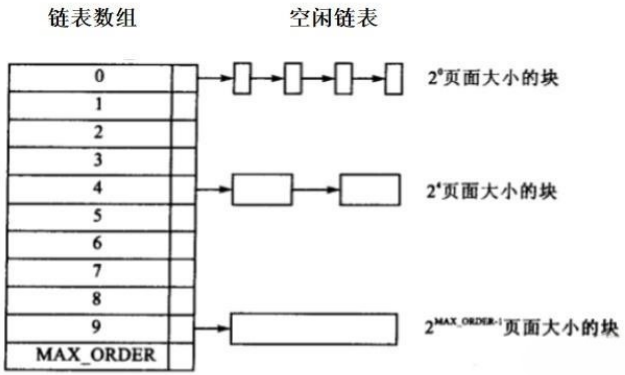


Implementation

Repository: https://gitee.com/cybsac/my_ucore/tree/lab2_buddy/

```
struct Page {
    int ref;                        // page frame's reference counter
    uint32_t flags;                 // array of flags that describe the status of the page frame
    unsigned int property;          // the num of free block, used in first fit pm manager
    list_entry_t page_link;         // free list link
};
```

```
typedef struct {
    unsigned int max_order;                         // 伙伴二叉树的层数
    list_entry_t free_array[MAX_BUDDY_ORDER + 1];   // 链表数组(现在默认有14层，即2^14 = 16384个可分配物理页)，
    unsigned int nr_free;                           // 伙伴系统中剩余的空闲块
} free_buddy_t;
```



Initialize

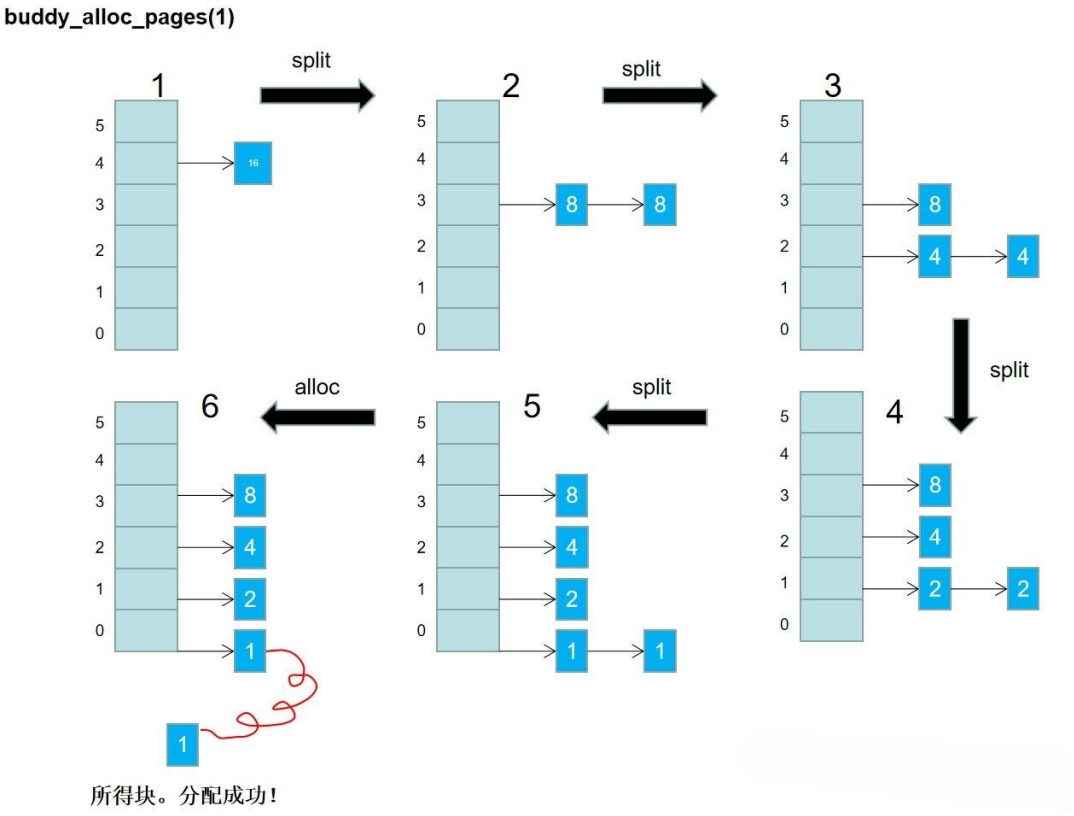
```
static void
buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    size_t pnum;
    unsigned int order;
    pnum = ROUNDDOWN2(n);       // 将页数向下取整为2的幂
    pnum = 8; // test!!!
    order = getOrderOf2(pnum);   // 求出页数对应的2的幂
    cprintf("[!]BS: AVA Page num after rounding down to powers of 2: %d = 2^%d\n", pnum, order);
    struct Page *p = base;
    // 初始化pages数组中范围内的每个Page
    for (; p != base + pnum; p ++) {
        assert(PageReserved(p));
        p->flags = 0;
        p->property = -1;   // 全部初始化为非头页
        set_page_ref(p, 0);
    }
    max_order = order;
    nr_free = pnum;
    list_add(&(buddy_array[max_order]), &(base->page_link)); // 将第一页base插入数组的最后一个链表，作为初始化的最大块—16384,的头页
    base->property = max_order;                              // 将第一页base的property设为最大块的2幂

    return;
}
```

Allocate

```
static struct Page *
buddy_alloc_pages(size_t n) {
    // require n > 0, or panic
    assert(n > 0);

    // if the number of required pages beyond what we have currently, return NULL
    if (n > nr_free) {
        return NULL;
    }

    struct Page *page = NULL;
    size_t pnum = ROUNDUP2(n);  // 处理所要分配的页数，向上取整至2的幂
    size_t order = 0;

    order = getOrderOf2(pnum);  // 求出所需页数对应的幂pow
    cprintf("[!]BS: Allocating %d-->%d = 2^%d pages ...\n", n, pnum, order);
    cprintf("[!]BS: Buddy array before ALLOC:\n");
    show_buddy_array();
```

```
find:
    // 若pow对应的链表中含有空闲块，则直接分配
    if (!list_empty(&(buddy_array[order]))) {
        page = le2page(list_next(&(buddy_array[order])), page_link);
        list_del(list_next(&(buddy_array[order])));
        SetPageProperty(page); // 将分配块的头页设置为已被占用
        cprintf("[!]BS: Buddy array after ALLOC NO.%d page:\n", page2ppn(page));
        show_buddy_array();
        goto done;
    }
    else {
        for (int i = order;i < max_order + 1;i ++) {
            // 找到pow后第一个非空链表，分裂空闲块
            if (!list_empty(&(buddy_array[i]))) {
                buddy_split(i);
                cprintf("[!]BS: Buddy array after SPLITT:\n");
                show_buddy_array();
                goto find;      // 重新检查现在是否可以分配
            }
        }
    }

done:
    nr_free -= pnum;
    cprintf("[!]BS: nr_free: %d\n", nr_free);
    return page;
}
```

Allocate


buddy_alloc_pages(1)
1
split
2
split
3
split
4
split
5
alloc
6
所得块。分配成功！

Split

```
// 默认分裂数组中第n条链表的第一块
static void buddy_split(size_t n) {
    assert(n > 0 && n <= max_order);
    assert(!list_empty(&(buddy_array[n])));
    cprintf("[!]BS: SPLITTING!\n");
    struct Page *page_a;
    struct Page *page_b;

    page_a = le2page(list_next(&(buddy_array[n])), page_link);
    page_b = page_a + (1 << (n - 1));
    page_a->property = n - 1;
    page_b->property = n - 1;

    list_del(list_next(&(buddy_array[n])));
    list_add(&(buddy_array[n-1]), &(page_a->page_link));
    list_add(&(page_a->page_link), &(page_b->page_link));

    return;
}
```

Free

```
static void
buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    unsigned int pnum = 1 << (base->property);
    assert(ROUNDUP2(n) == pnum);
    cprintf("[!]BS: Freeing NO.%d page leading %d pages block: \n", page2ppn(base), pnum);
    struct Page* left_block = base;
    struct Page *buddy = NULL;
    struct Page* tmp = NULL;

    buddy = buddy_get_buddy(left_block);
    list_add(&(buddy_array[left_block->property]), &(left_block->page_link));
    cprintf("[!]BS: add to list\n");
    show_buddy_array();
    // 当伙伴块空闲，且当前块不为最大块时
    while (!PageProperty(buddy) && left_block->property < max_order) {
        cprintf("[!]BS: Buddy free, MERGING!\n");
        if (left_block > buddy) { // 若当前左块为更大块的右块
            left_block->property = -1;
            ClearPageProperty(left_block);
            tmp = left_block;
            left_block = buddy;
            buddy = tmp;
        }
        list_del(&(left_block->page_link));
        list_del(&(buddy->page_link));
        left_block->property += 1;
        list_add(&(buddy_array[left_block->property]), &(left_block->page_link)); // 头插入相应链表
        show_buddy_array();
        buddy = buddy_get_buddy(left_block);
    }
    cprintf("[!]BS: Buddy array after FREE:\n");
    ClearPageProperty(left_block); // 将回收块的头页设置为空闲
    nr_free += pnum;
    show_buddy_array();

    cprintf("[!]BS: nr_free: %d\n", nr_free);
    return;
}
```

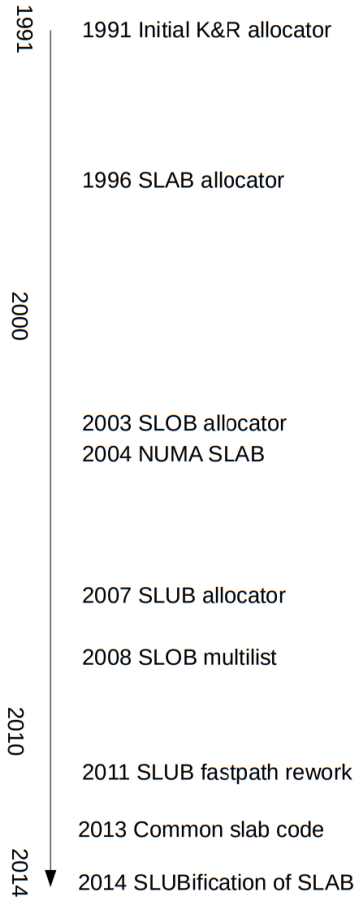
Slab, Slob, Slub 的演进之路

从“批发”到“零售”的艺术

为什么需要内存分配器？

伙伴系统 (Buddy System) 最小单位：1 Page (4KB)。
实际应用时需要很多小对象
直接分配会导致巨大的 内部碎片 (Internal Fragmentation)。
目标：高效管理小对象，缓存复用。

timeline


1991
2000
2010
2014
1991 Initial K&R allocator
1996 SLAB allocator
2003 SLOB allocator
2004 NUMA SLAB
2007 SLUB allocator
2008 SLOB multilist
2011 SLUB fastpath rework
2013 Common slab code
2014 SLUBification of SLAB

Slab - 开山鼻祖

A slab consists of one or more physically contiguous pages

A cache consists of one or more slabs

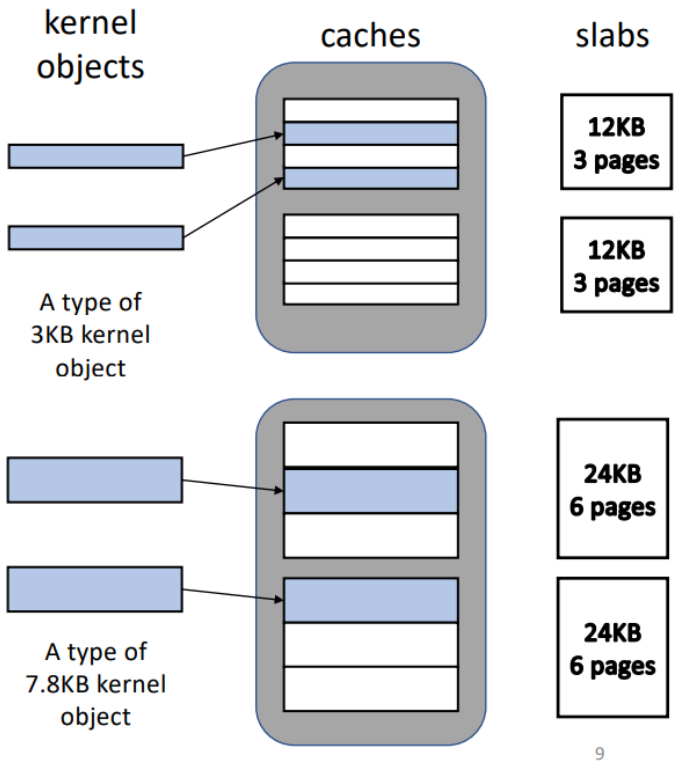
One cache for each type of kernel objects

Cache（缓存池）

->

Slab（大块内存，通常是几个连续物理页）

-> Object（具体对象）



Slab的管理

三链表管理：每个 Cache 维护三个链表：

slabs_full：完全分配完的 Slab。

slabs_partial：分配了一部分，还有空位的 Slab。

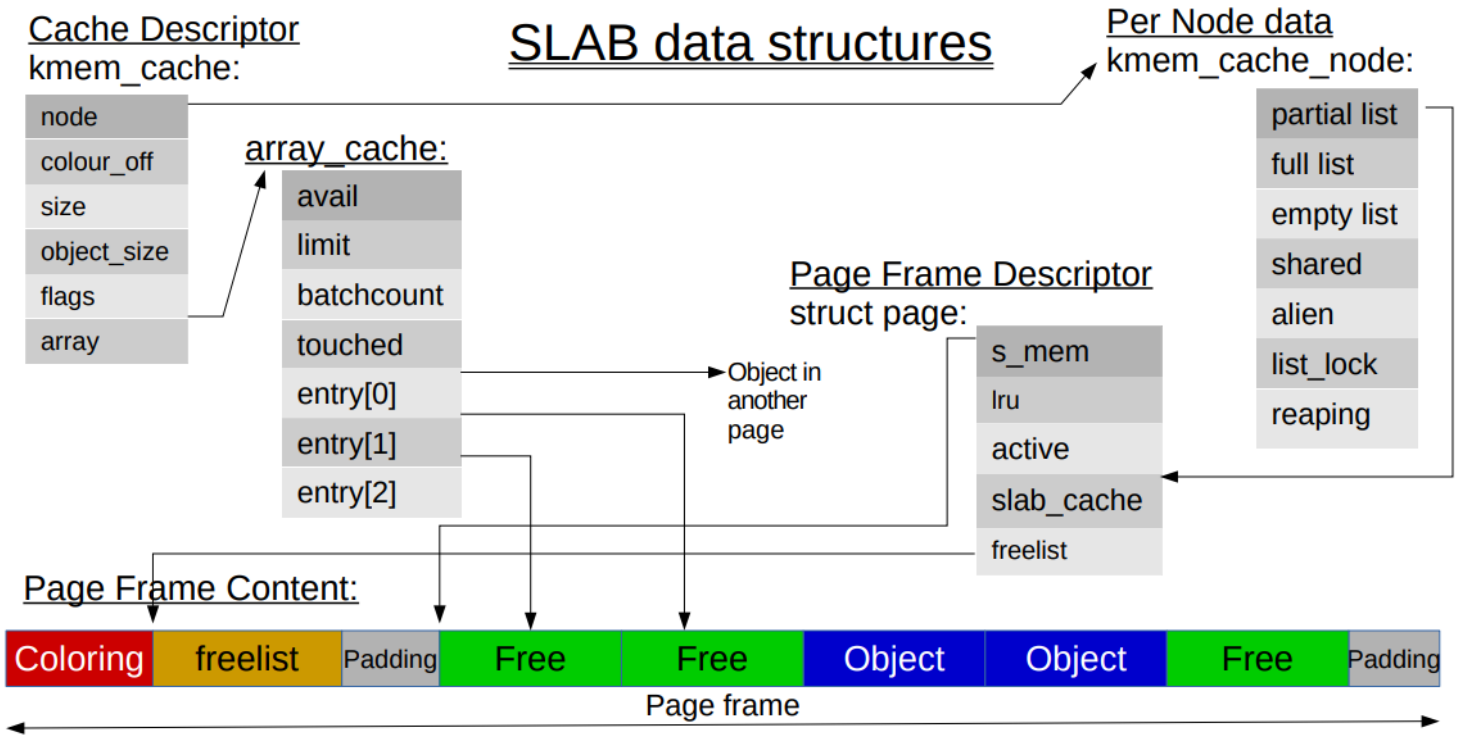
slabs_free：完全空闲的 Slab。

申请内存时：

slabs_partial-> slabs_free-> buddy system

```
struct page {
    ...
    union {
        ...
        struct {    /* slab, slob and slub */
            union {
                struct list_head slab_list;
                ...
            };
            struct kmem_cache *slab_cache; /* not slob */
            /* Double-word boundary */
            void *freelist;     /* first free object */
            union {
                void *s_mem;    /* slab: first object */
            };
        };
        ...
    };
} _struct_page_alignment;
```

Slab的结构


SLAB data structures
Cache Descriptor
kmem_cache:
node
colour_off
size
object_size
flags
array
array_cache:
avail
limit
batchcount
touched
entry[0]
entry[1]
entry[2]
Object in another page
Per Node data
kmem_cache_node:
partial list
full list
empty list
shared
alien
list_lock
reaping
Page Frame Descriptor
struct page:
s_mem
lru
active
slab_cache
freelist
Page Frame Content:
Coloring
freelist
Padding
Free
Free
Object
Object
Free
Padding
Page frame

Slab的优越性

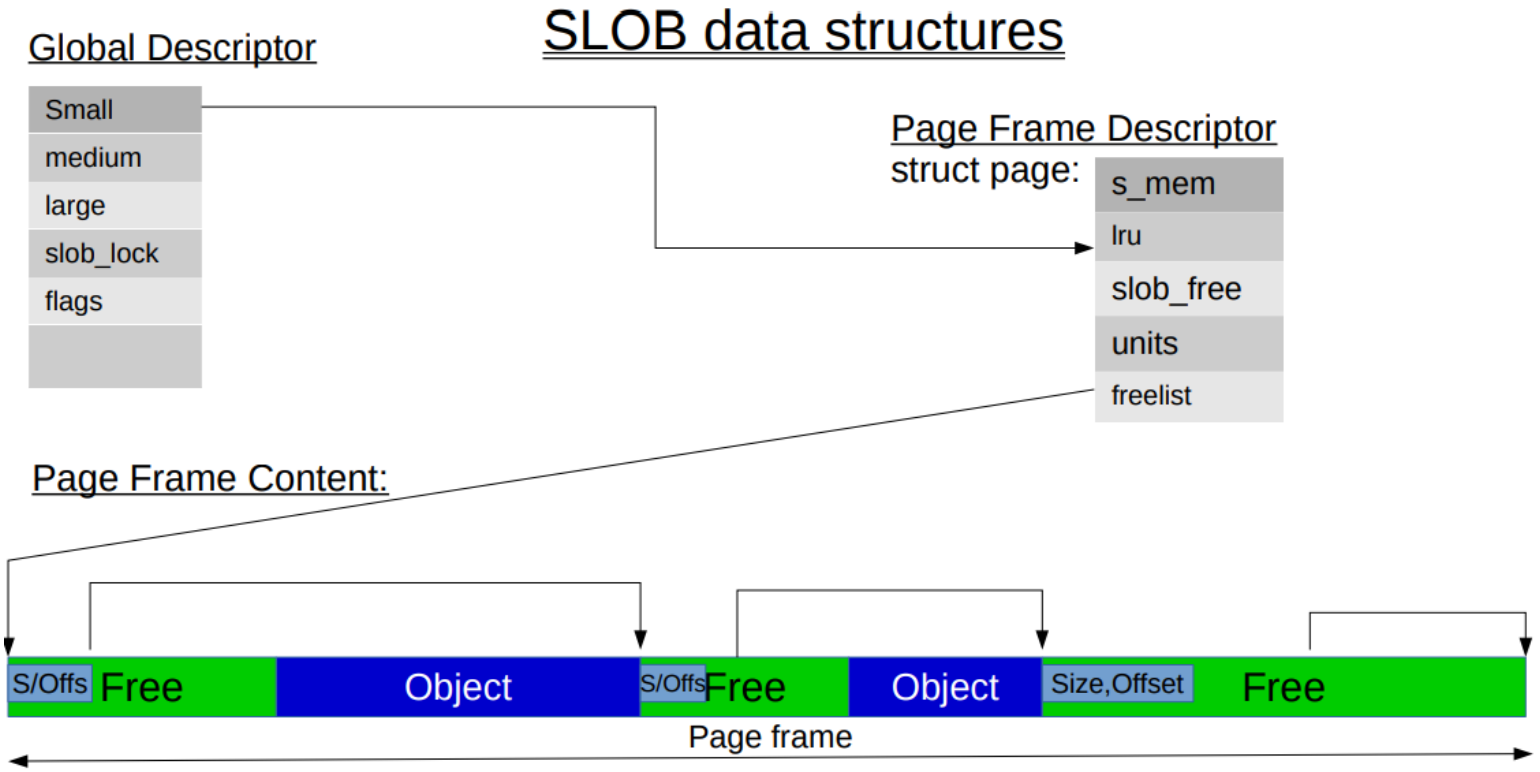
1. Spatial Locality

Slab 把相同类型的对象（比如一堆 inode）紧挨着放在一起。当你读取 inode A 时，CPU 会把紧挨着的 inode B 也顺便加载进 L1 Cache。如果你接下来正好要遍历这些对象，速度会很快

2. Slab Coloring

故意浪费一点内存偏移量，通过让不同页里的对象起始地址错开，防止它们在 CPU 的同一条Cache Line里‘撞车’。

Slob – Simple List Of Blocks



Slob

三个全局链表：

free_slob_small: 页面内剩余空间 < 256 字节。

free_slob_medium: 页面内剩余空间 < 1024 字节。

free_slob_large: 页面内剩余空间 >= 1024 字节。

分配方法：

选择链表：Slob 根据你要的大小，去 small, medium 或 large 链表中找对应的页。

First-Fit 扫描（最核心的算法）：

Slob 拿到一个页，开始遍历页内的空闲块链表。

它查看第一个空闲块：够不够 x 字节？

如果不够，看下一个。

找到第一个足够大的空闲块（假设该块大小为 Y，Y >= x）。

切割：

如果 Y 刚好等于 x，直接拿走，修改链表指针。

如果 Y 比 x 大很多，Slob 会把这个块切成两半。前 x 字节给你，剩下的 Y-x 字节变成一个新的更小的空闲块，插回链表。

无页可用：如果所有链表里的页都塞不下了，Slob 就向底层的伙伴系统申请一个新的 4KB 页，把它挂到 free_slob_large 链表上，然后开始切。

Slob的优缺点

极度节省内存元数据，不需要像Slab 那样浪费几百字节去存管理结构，对于小内存（嵌入式）设备比较友好

使得内存的碎片化变得很严重。同时分配时需要遍历链表，内存越满，碎片越多，遍历越慢。。

Removed in Linux 6.4

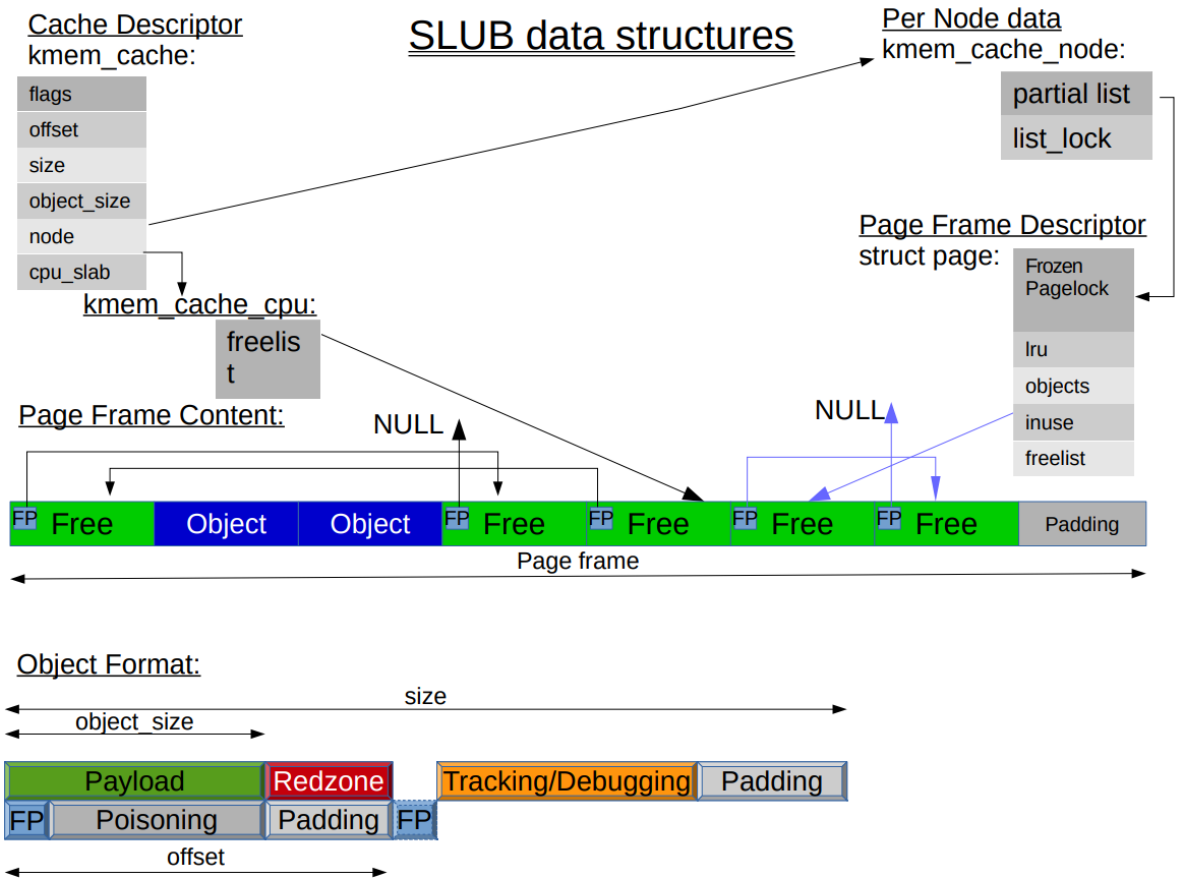
Slub：The Unqueued Slab

地位：目前linux内核的默认分配器。

核心改进：

- 复用 struct page：不再单独分配元数据空间，而是把元数据塞进 struct page 的联合体中。
- Per-CPU 缓存：无锁路径，优化性能。

Slub的结构


SLUB data structures
Cache Descriptor
kmem_cache:
flags
offset
size
object_size
node
cpu_slab
kmem_cache_cpu:
freelis
t
Per Node data
kmem_cache_node:
partial list
list_lock
Page Frame Descriptor
struct page:
Frozen
Pagelock
lru
objects
inuse
freelist
Page Frame Content:
NULL
NULL
FP
Free
Object
Object
FP
Free
FP
Free
FP
Free
FP
Free
Padding
Page frame
Object Format:
size
object_size
Payload
Redzone
Tracking/Debugging
Padding
FP
Poisoning
Padding
FP
offset

Slub的工作流程

分配流程：

- Fast Path（快路径）：
  - 关闭抢占。
  - 读取当前 CPU 的 kmem_cache_cpu。
  - 检查 freelist 是否不为空。
  - 成功：直接返回 freelist 指向的地址，并将 freelist 更新为下一个空闲对象。开启抢占，结束。（耗时极短，几条指令）
  - 失败：freelist 是空的，说明当前页用完了，进入慢路径。
- Slow Path（慢路径）：
  - 借用：去 kmem_cache_node 的 partial 链表里找一个还没满的页。
  - 冻结 (Frozen)：如果找到了，把这个页从 Node 的链表里摘下来，标记为 Frozen（表示这个页现在归我这个 CPU 独占了），然后把它挂到我的 kmem_cache_cpu 上，变成新的本地页。
  - 新建：如果 partial 链表也是空的，向伙伴系统申请全新的物理页，初始化后挂载。

总结

特性	Slab (旧)	Slob (已删)	Slub (现任霸主)
元数据开销	高 (额外结构体)	极低	低 (复用 Page 结构)
复杂度	复杂 (着色、队列)	简单 (链表)	中等 (无队列)
性能	较好	差 (碎片化后)	**最好**
适用场景	早期通用系统	极低内存嵌入式	**通用服务器、手机、PC**