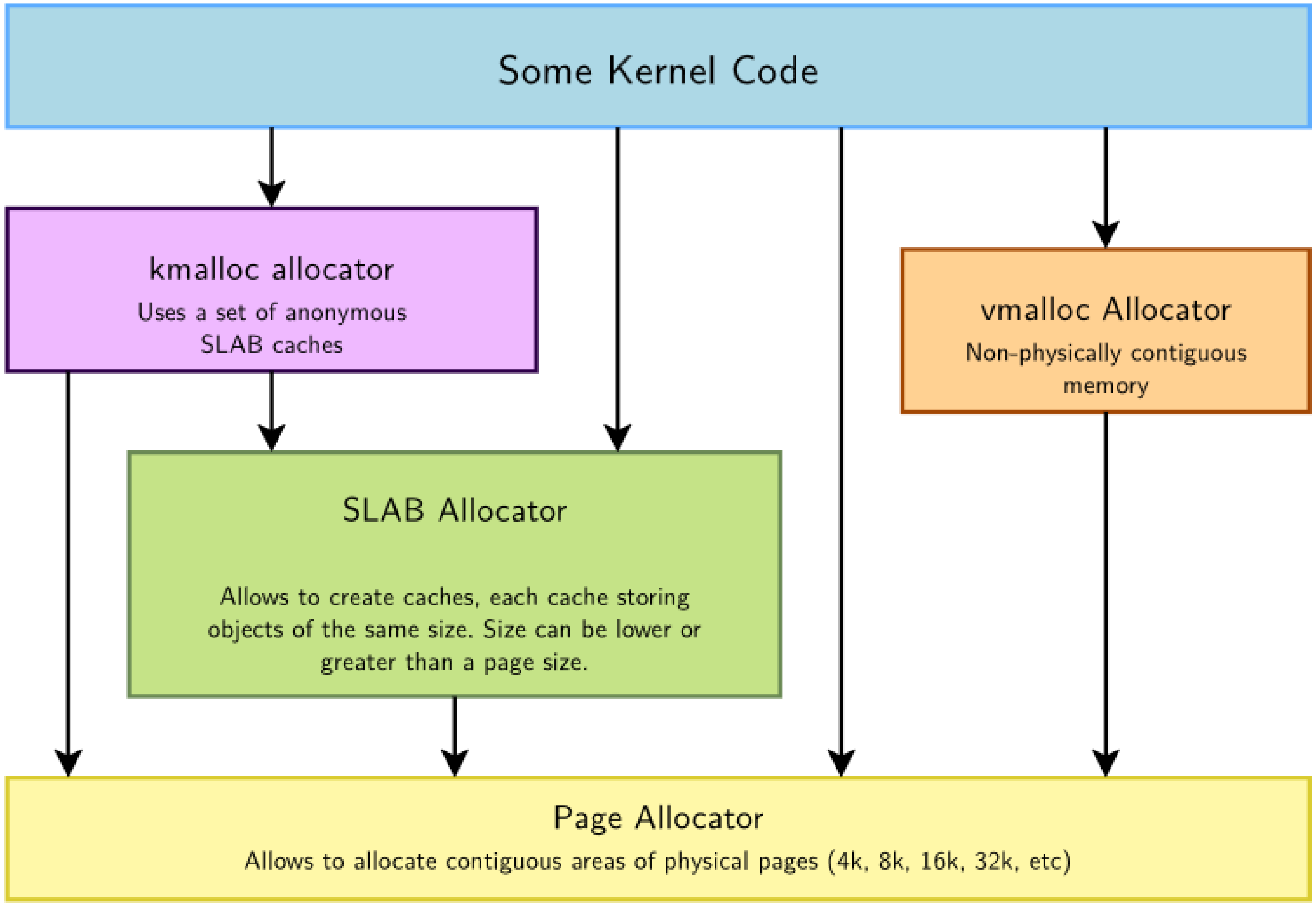
Memory management
Buddy allocator
Slab allocator

Table of contents

Memory management


Some Kernel Code
kmalloc allocator
Uses a set of anonymous SLAB caches
vmalloc Allocator
Non-physically contiguous memory
SLAB Allocator
Allows to create caches, each cache storing objects of the same size. Size can be lower or greater than a page size.
Page Allocator
Allows to allocate contiguous areas of physical pages (4k, 8k, 16k, 32k, etc)

Introduction

Based on kvmalloc() (Jonathan Corbet, January 2017)

The kernel offers two mechanisms for allocating memory, both of which are built on top of the kernel's **page allocator** (**zoned buddy allocator**):

- **slab allocator** obtains physically contiguous memory in the kernel's own address space; this allocator is typically accessed via **kmalloc()**,
- **vmalloc()** returns memory in a separate address space; that memory will be virtually contiguous but may be physically scattered.

As a general rule, **slab allocations are preferred for all but the largest of allocations**.

- In the absence of memory pressure, the slab allocator will be **faster**, since it does not need to make **address-space changes** or **TLB invalidation**.
- The slab allocator works best with allocations that are **less than one physical page** in size (**vmalloc()** can only allocate **entire page**).
- When memory gets fragmented, groups of **physically contiguous pages** can get hard to find, and system performance can suffer as the allocator struggles to create such groups.

Managing free page frames – Buddy allocator

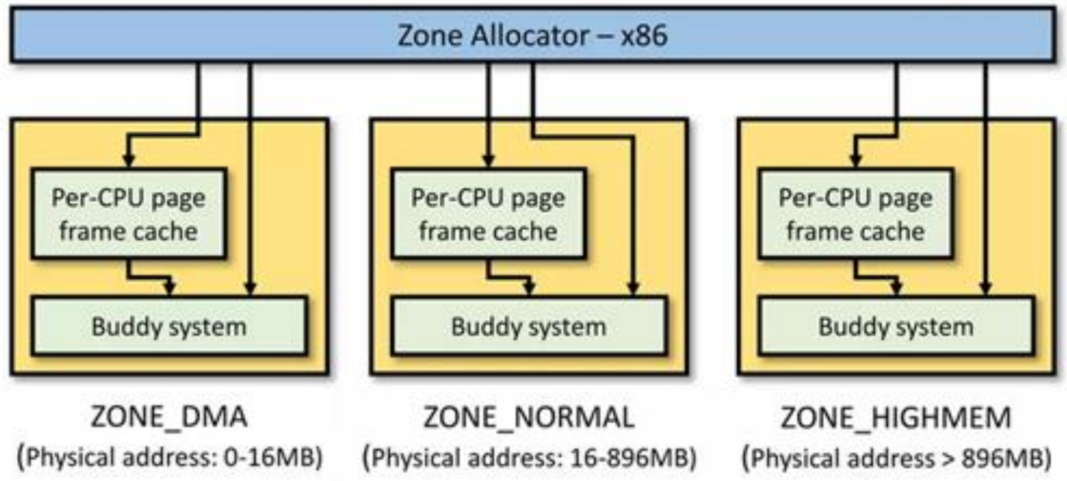
The kernel often needs **contiguous memory areas** that span multiple page frames, so when serving memory allocation requests, it must **minimize external fragmentation**.

This is the task of the **buddy allocator**.

The concept of the buddy allocator is to maintain **directly-mapped table** for memory **blocks** of various **orders**. The bottom level table contains the map for the smallest allocable units of memory (here, **pages**), and each level above it describes pairs of units from the levels below – **buddies**.

Linux uses a separate buddy allocator **in each zone**.

– Each of the buddy allocators uses the corresponding **subset of page descriptors**.
– Information about page frames containing **free** physical memory areas is stored in the **free_area** structure.
– Information about the **occupied** page frames is stored in the **process page tables** and the **kernel page table**.



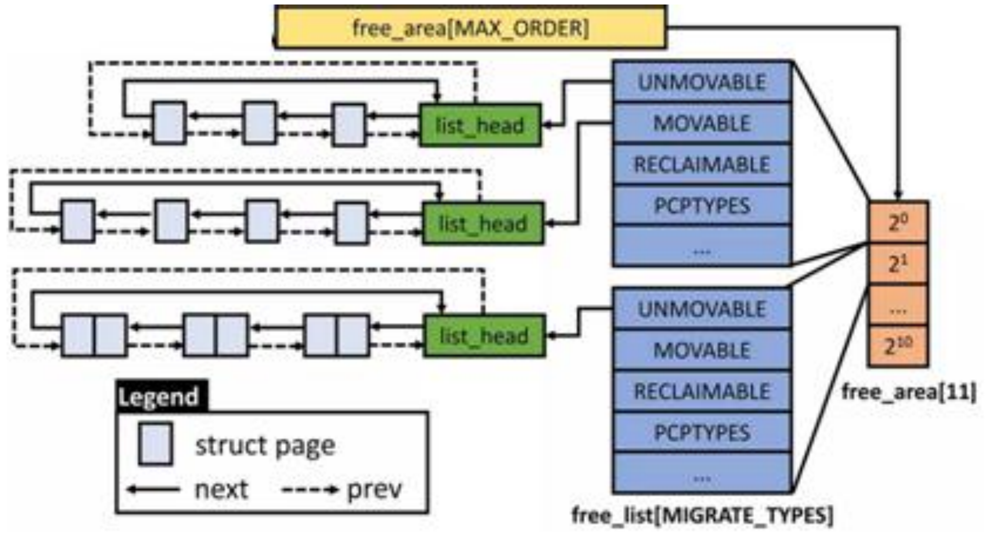
(source: Adrian Huang, Physical Memory Management, 2022)

Buddy allocator – data structures

The **free_area** is a table of structures of type **struct free_area**.

```
struct free_area free_area[MAX_ORDER];

struct free_area {
    struct list_head free_list[MIGRATE_TYPES];
    unsigned long nr_free;
};
```



(source: Adrian Huang, Physical Memory Management, 2022)

- The size of the **free_area** table is specified by the constant MAX_ORDER = 11.
- The i-th element of the table points to a **cyclic list of free** and **contiguous** memory areas of size: (2^i) * PAGE_SIZE, where i = 0..MAX_ORDER -1. The first list contains memory areas of the size of one page, and the last memory areas of the size of 1024 pages.
- Any such memory area is a **contiguous** block of physical memory.
- The **physical address of the first page frame** of the block is a **multiple of the block size**. For example, the start address of a block with a size of 16 page frames is a multiple of 16 * PAGE_SIZE.
- The list contains the page descriptors of the **first page frame** of each block, pointers to subsequent items in the list are stored in the **lru** field of the **page descriptor**.
- The field **nr_free** contains the number of free areas of a given size.

Buddy allocator – data structures

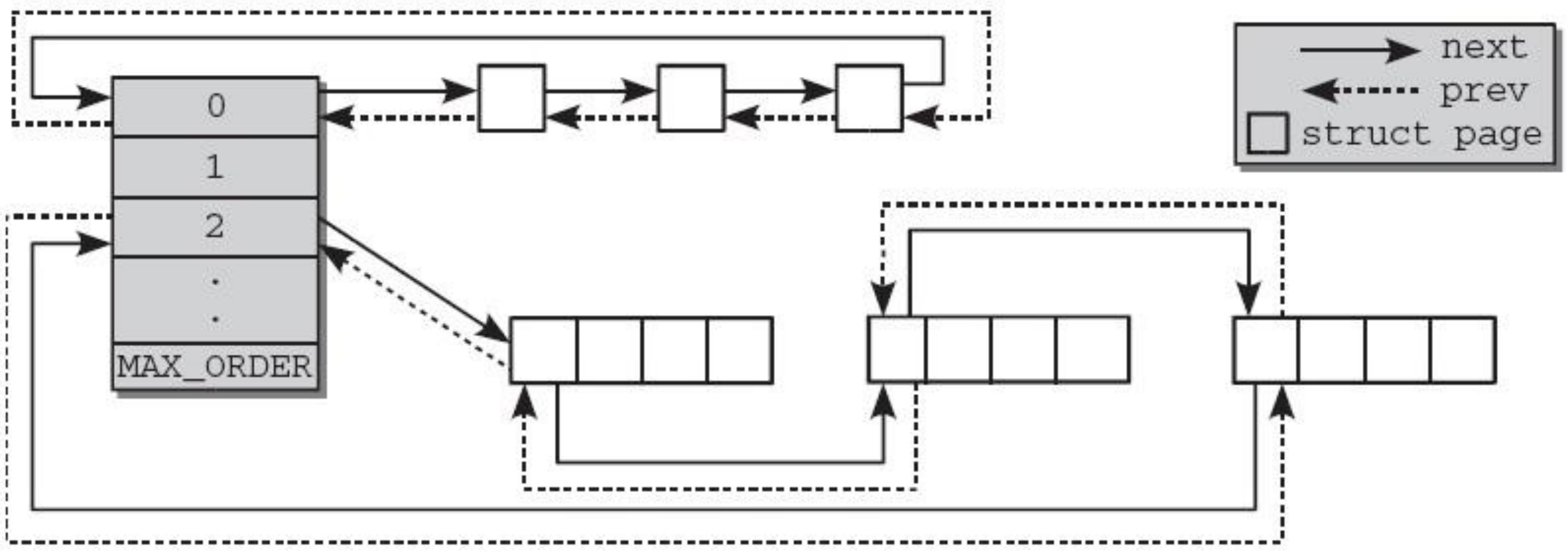


Figure 3-22: Linking blocks in the buddy system.

Linking blocks in the buddy allocator
(source: W. Mauerer, Professional Linux Kernel Architecture)

[RFC][PATCH] no bitmap buddy allocator: remove free_area->map (September 2004)

Buddy allocator – data structures

The **/proc/buddyinfo** file shows the status of memory under the supervision of the buddy allocator.

Each column shows the number of pages available in blocks of a given size (order).

In the case shown (my old workstation) there are 12 blocks of size 2^2 * PAGE_SIZE in the DMA zone and 47 blocks of size 2^3 * PAGE_SIZE in the NORMAL zone.

```
Node 0, zone   DMA        111    59   12    1   1   1   1   1   1   1   0
Node 0, zone   Normal    2770   753  169   47   0   0   0   0   0   0   1
Node 0, zone   HighMem     23     4    2    1   1   0   0   0   0   0   0
```

On ‘students’ (2024-04-07).

```
Node 0, zone   DMA          0       0      0       0      0      0     0     0     1    1     2
Node 0, zone   DMA32       34      43     36     142    111     77    63    24    15   39   347
Node 0, zone   Normal  413787  278148  92164  197638  54000  14153  3639   481   146   82     0
```

Buddy allocator – data structures

Information about being a **free area of a certain size** is included in the structure **page**:

• the **private** field of the first page frame in a block 2^k of free page frames holds the number k,

• the **_mapcount** field of the descriptor holds special value PAGE_BUDDY_MAPCOUNT_VALUE (-128).

The **PageBuddy()** returns TRUE, when the page frame is managed by the buddy allocator.

Useful macros:

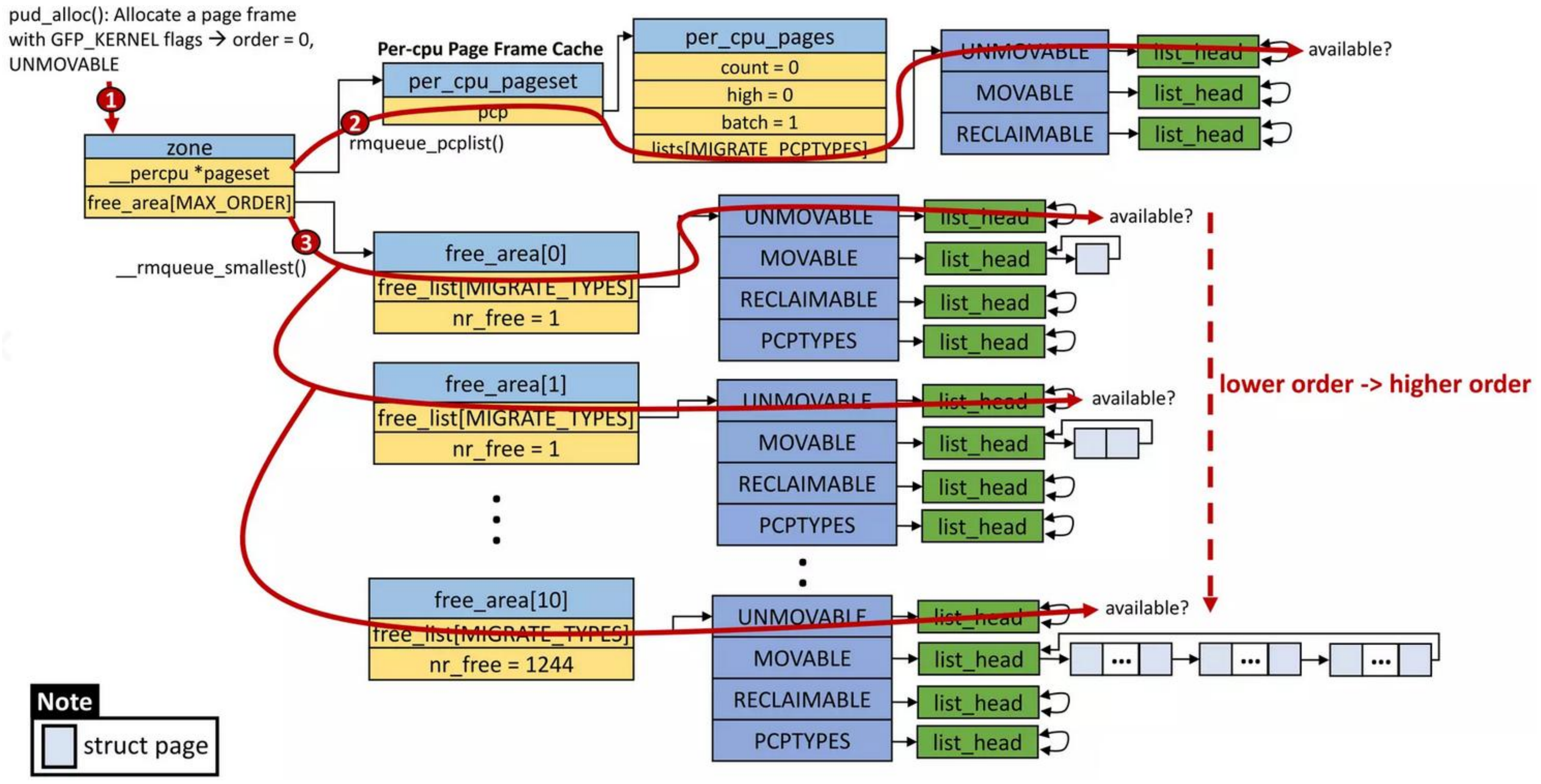
__SetPageBuddy()
__ClearPageBuddy()

The code can be found in **/include/linux/page-flags.h**.

linux/mm/page_alloc.c

```
/ *
 * This function checks whether a page is free && is the buddy
 * we can do coalesce a page and its buddy if
 * (a) the buddy is not in a hole (check before calling!) &&
 * (b) the buddy is in the buddy system &&
 * (c) a page and its buddy have the same order &&
 * (d) a page and its buddy are in the same zone.
 *
 * For recording page's order, we use page_private(page).
 */
static inline int page_is_buddy(struct page *page, struct page *buddy, unsigned
int order)
{
	...
	if (PageBuddy(buddy) && page_order(buddy) == order) {
			if (page_zone_id(page) != page_zone_id(buddy))
				return 0;
			VM_BUG_ON_PAGE(page_count(buddy) != 0, buddy);
				return 1;
	}
	return 0;
}
```

Allocating page frame – more details



(source: Adrian Huang, Physical Memory Management, 2022)

Buddy allocator – freeing page frames

This function frees the area indicated (indirectly) by page of size: (2^{order}) * PAGE_SIZE and inserts it into one of the queues in the **free_area** table.

The freed area had to be once assigned. If it was assigned, it had to be separated from a block two times bigger. Perhaps its **buddy** (that is, the other half of this larger block) is also **free** and can they be **combined back** into one larger contiguous area.

If it succeeds, then you can look for a buddy for this larger piece, etc.

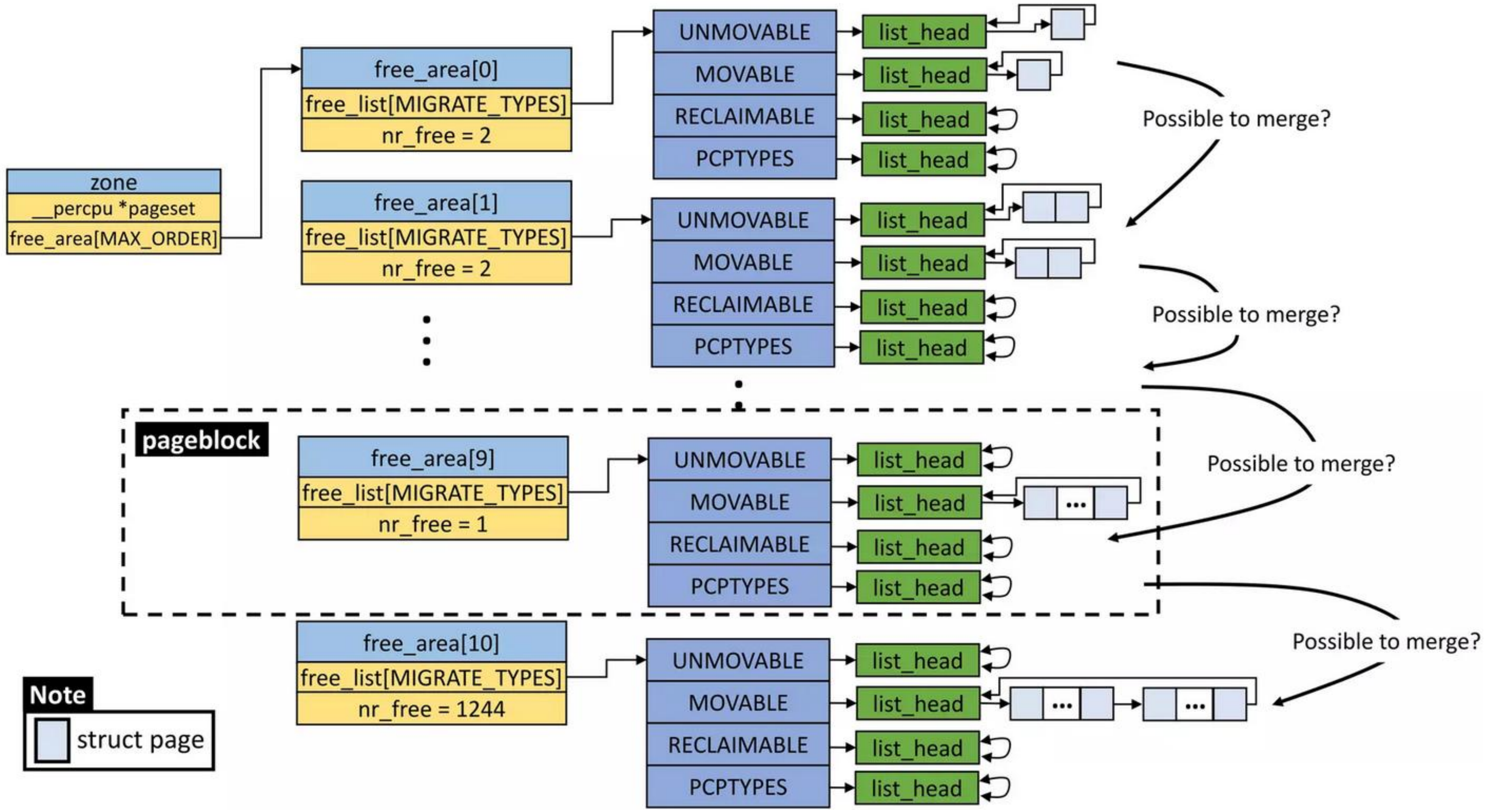
The block is then added to the appropriate queue, and the information about those areas as belonging to the buddy allocator is updated.

Two blocks are buddies if:

- have the **same size**, e.g. **b**,
- are **adjacent** to each other,
- the **physical address** of the first frame of the first block is a multiple of 2 x **b** x PAGE_SIZE.

While combining the areas, the kernel has to calculate two values: the **address** of the buddy and the **index** of the pair of buddies after they are combined back.

Freeing page frame – more details



(source: Adrian Huang, [Physical Memory Management](), 2022)

Buddy allocator – freeing page frames

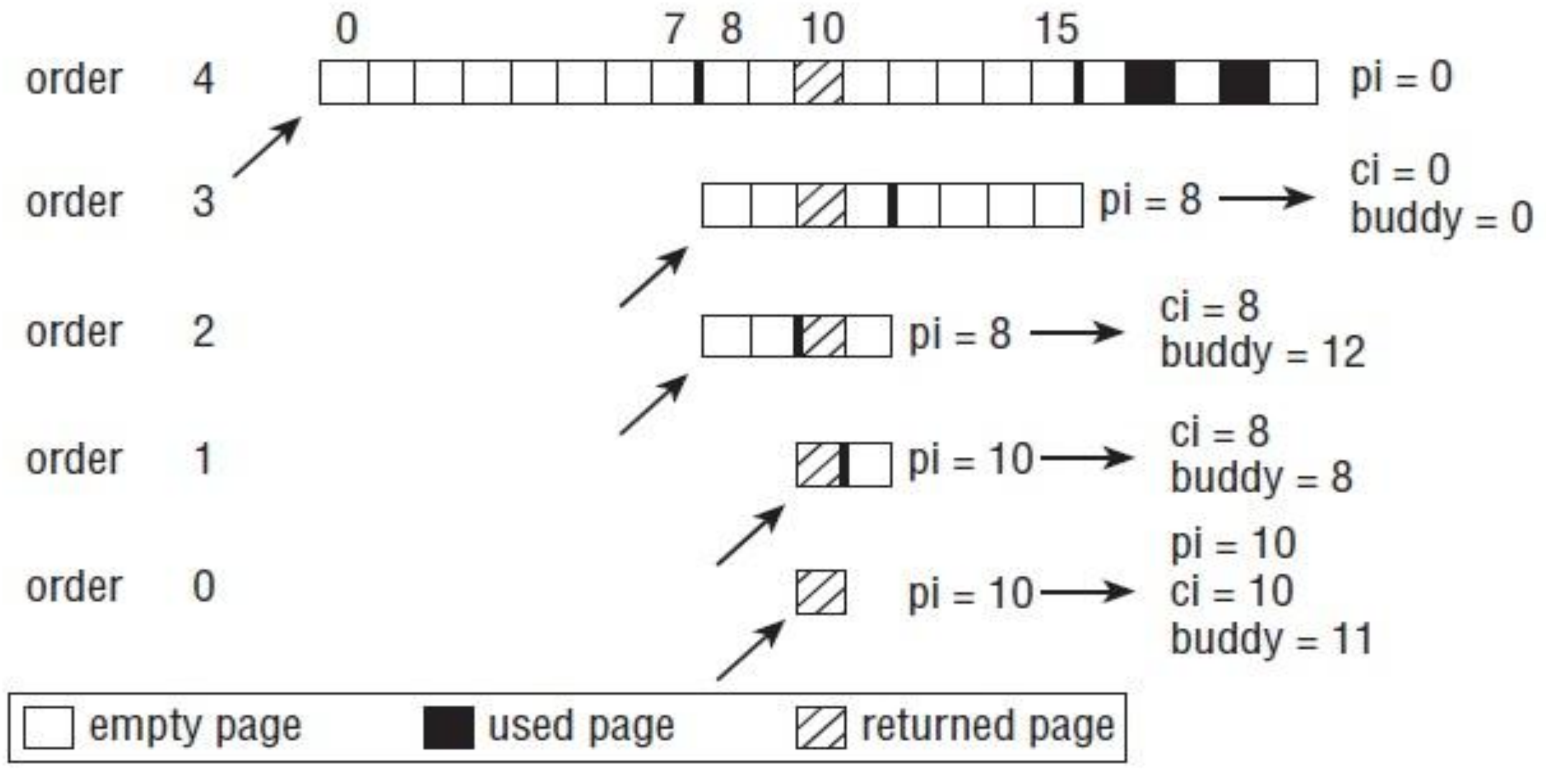


Figure 3-36: Returning a page into the buddy system can cause higher-order allocations to be coalesced. pi stands for `page_index`, while ci denotes `combined_index`.

Freeing page frames in buddy allocator – example
(source: W. Mauerer, Professional Linux Kernel Architecture)

Buddy allocator API

page *alloc_pages(gfp_mask, order) – allocates 2^{order} of physically contiguous page frames and returns the pointer to the descriptor of the first one;

unsigned long __get_free_pages(gfp_mask, order) – allocates 2^{order} of physically contiguous page frames and returns the linear address of the first one (**unsigned long**). *Never use with __GFP_HIGHMEM because the returned address cannot represent highmem pages. Use **alloc_pages** and then **kmap** if you need to access highmem.*

void *page_address() converts the page descriptor into its linear address. Function returns the **kernel virtual address** of the page frame or **NULL** when the frame is in highmem and has not been mapped.

#define **alloc_page**(gfp_mask) **alloc_pages**(gfp_mask, 0)

#define **__get_dma_pages**(gfp_mask, order) **__get_free_pages**((gfp_mask | GFP_DMA, (order))

unsigned long **get_zeroed_page**(gfp_t gfp_mask) {return **__get_free_page**(gfp_mask | __GFP_ZERO);
}

Buddy allocator – final notes

The kernel tries to counteract fragmentation by dividing pages by type, which depends on the **mobility** of the page:

- **non-movable pages** – most pages of the kernel itself,
- **reclaimable pages** – can be removed from memory and retrieved from a source, e.g. a file mapped into memory,
- **movable pages** – pages that can be moved in memory, e.g. pages of user processes.

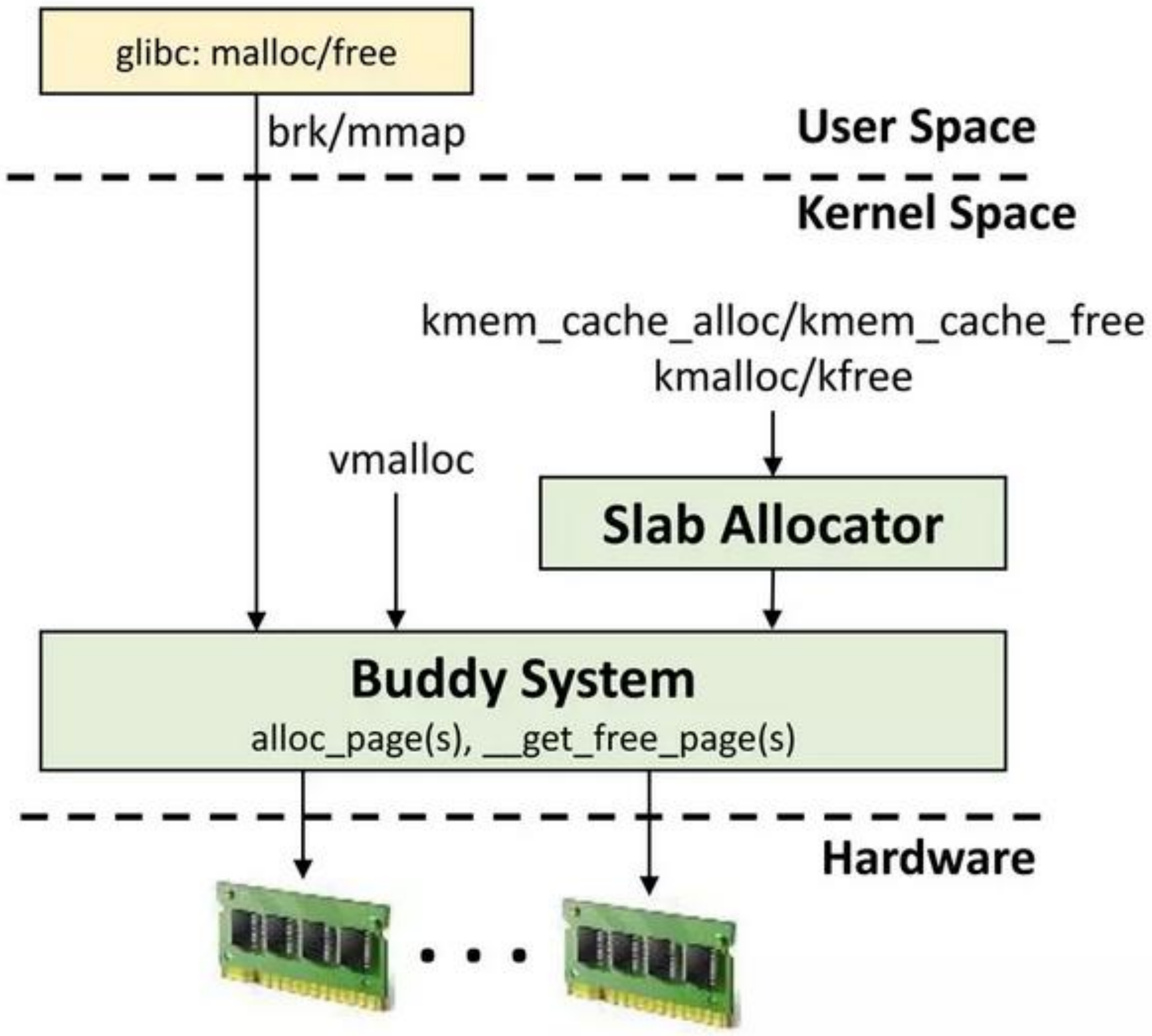
Pages of different types are placed on separate lists (therefore the kernel maintains an array of lists of size **MIGRATE_TYPES**).

What is the **free_area** structure for, if the **user process** which requests several page frames, gets them **one by one** and not at all in a contiguous block? This structure was created mainly for the **use of the kernel**, which **must** be able to allocate **contiguous** memory areas in different sizes.

The buddy allocator is **very fast**. This is due to the fact that the majority of arithmetic operations involve a **binary shift** or a **bit change**. That's why the **free_area** array is indexed with the power of two.

Each area added to the queue is inserted at the **beginning** of the queue, while the retrieval takes place from the **beginning** or from the **middle** (if we want to remove the buddy with the given address).

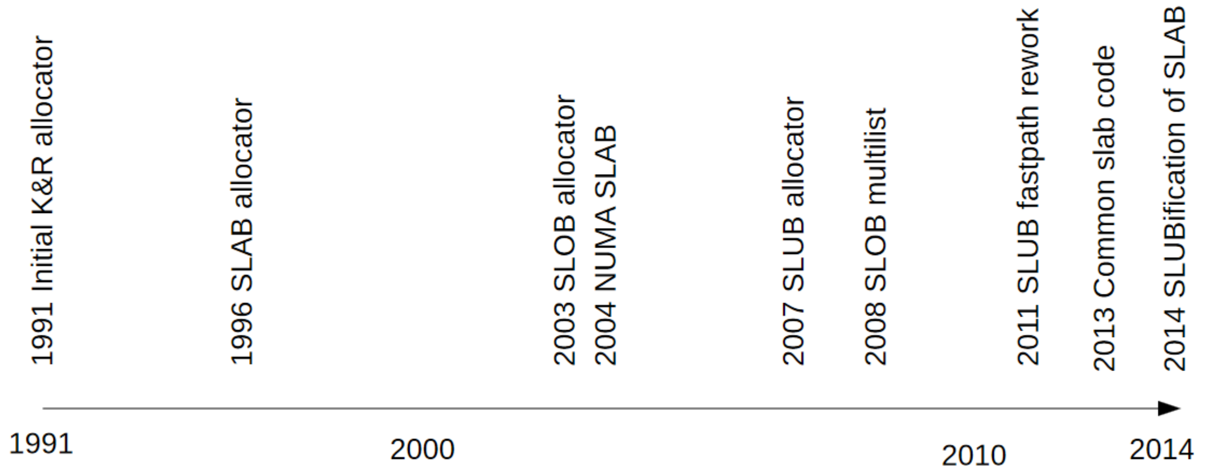
Why do we need yet another memory allocator?



(source: Adrian Huang, Slab Allocator in Linux Kernel, 2022)

Slab allocators – readings

- The SLUB allocator (Jonathan Corbet, February 2007)
- Cramming more into struct page (Jonathan Corbet, 2013).
- **Chris Lameter** (one of the maintainers) in 2014 delivered the presentation *Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB* (slides, talk)
- Toward a more efficient slab allocator (Jonathan Corbet, January 2015)
- Making slab-allocated objects moveable (Jonathan Corbet, April 2019)
- Pulling slabs out of struct page (Jonathan Corbet, October 2021)
- SLOB nears the end of the road (Jonathan Corbet, December 2022)



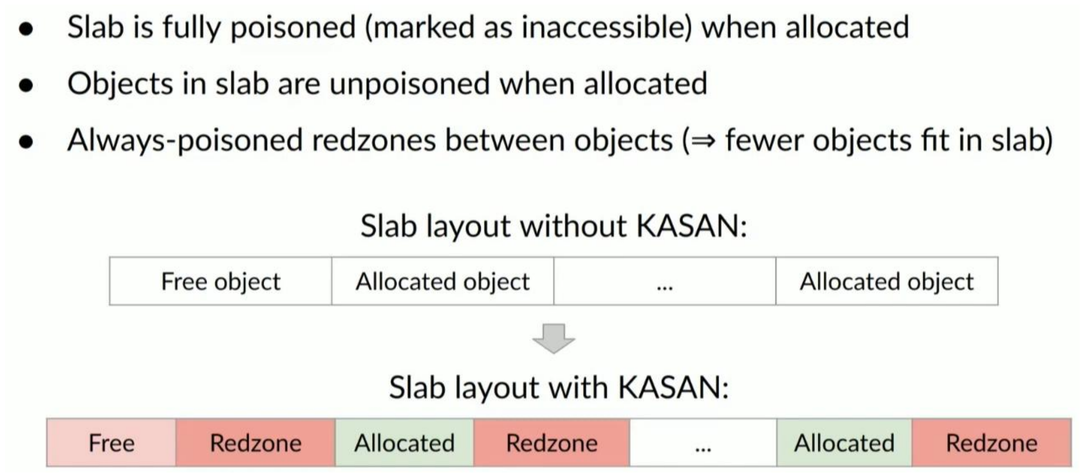
Slab allocators – readings

- The slab allocators of past, present, and future (**Vlastimil Babka**, Linux Plumbers Conference 2022).
- SL[OA]B removal and future of SLUB (Vlastimil Babka, The Linux Storage, Filesystem, Memory Management & BPF Summit @ OSS NA 2023).
- Reducing the Kernel's Slab Allocators: Progress Report (Vlastimil Babka, LinuxCon, OSS 2023).
- [PATCH v2 0/6] remove SLOB and allow kfree() with kmem_cache_alloc() @ 2023-03-17 10:43 Vlastimil Babka.
- The rise and fall of kernel slab allocators (Vlastimil Babka, SUSE Labs Conference 2023).
- SLAB – deprecated in 6.5, removed in 6.8. Kernel developers now have greater freedom to improve SLUB without worrying about breaking the others.

Extra reading

- Sanitizing the Linux Kernel – On KASAN and other Dynamic Bug-finding Tools – Andrey Konovalov

  (how KASAN works, shadow memory, **redzone in slabs** and kmalloc)



- Slab is fully poisoned (marked as inaccessible) when allocated
- Objects in slab are unpoisoned when allocated
- Always-poisoned redzones between objects (⇒ fewer objects fit in slab)

Slab layout without KASAN:

Free object	Allocated object	...	Allocated object

Slab layout with KASAN:

Free	Redzone	Allocated	Redzone	...	Allocated	Redzone

Slab allocators – introduction

The **slab allocator** supports memory allocation **for the kernel**.

The kernel needs many different **temporary objects**, such as the dentry, mm_struct, inode, files_struct structures.

Temporary kernel objects can be both **very small** and **very large**, moreover, they are often allocated and often freed, so you have to perform these operations efficiently.

The **buddy allocator**, which operates with areas composed of **entire frames** of memory, is **not suitable** for this.

If the allocator is aware of concepts such as object size, page size, and total cache size, it can make more **intelligent decisions**.

If part of the (object) cache is made **per-processor** (separate and unique to each processor on the system), allocations and frees can be performed without an **SMP lock**.

If the allocator is **NUMA-aware**, it can fulfill allocations from the same memory node as the requestor.

Alternative slab allocators

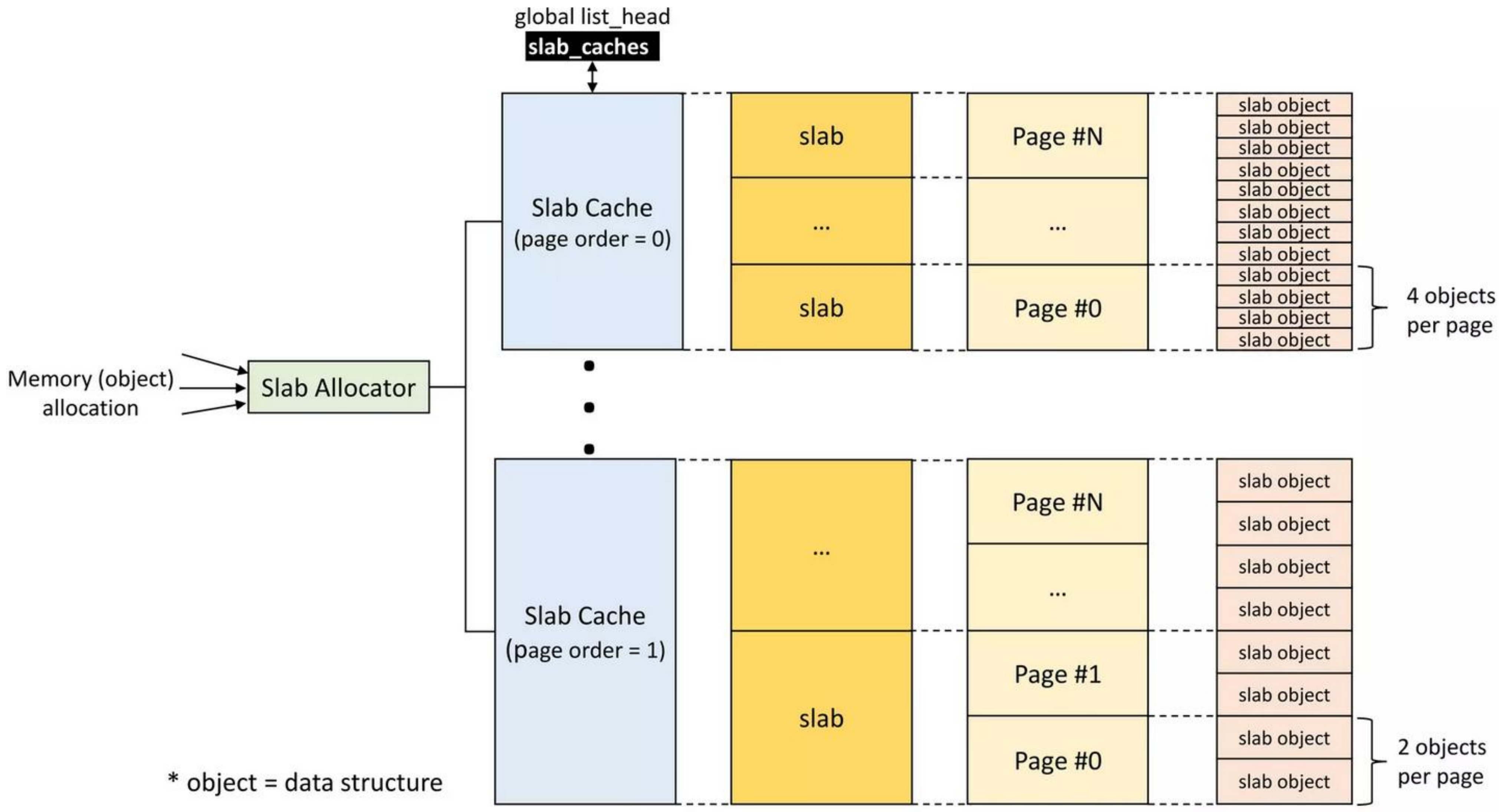
- **SLOB** allocator – designed for **small systems**. The slob name comes from **lists of blocks**, the allocator only takes 600 lines of code, a simple **first fit** algorithm is used to allocate memory, can suffer from fragmentation, smallest memory footprint; (removed in 6.4-rc1 by Vlastimil Babka).
- **SLAB** allocator – based on allocator from Solaris, relatively stable implementation and performance (better for some workloads?), smaller memory usage than SLUB (?); (deprecated in 6.5,removed in 6.8 by Vlastimil Babka – kernel developers now have greater freedom to improve SLUB without worrying about breaking the others).
- **SLUB** allocator – designed for **large systems**. Overall best performance, the best debugging features (always compiled-in, boot-time enabled), primary target for new features (PREEMPT_RT).

Design philosophies:

- **SLOB** – as compact as possible.
- **SLAB** – as cache friendly as possible.
- **SLUB** – simple and instruction cost counts. Defragmentation. Execution time friendly. Default since 2.6.23.

The higher levels of the kernel do not have to be aware of which slab allocator is actually used. The API is the same.

Slab allocator components



(source: Adrian Huang, Slab Allocator in Linux Kernel, 2022)

Slab allocator in Linux

The main task of the slab allocator in Linux is to **reduce the number of references to the buddy allocator**.

The slab allocator maintains many different **caches**.

For frequently used objects (such as files_struct) it maintains a **dedicated cache**, and for other objects a number of **generic caches**, one for each area of size being the next power of two.

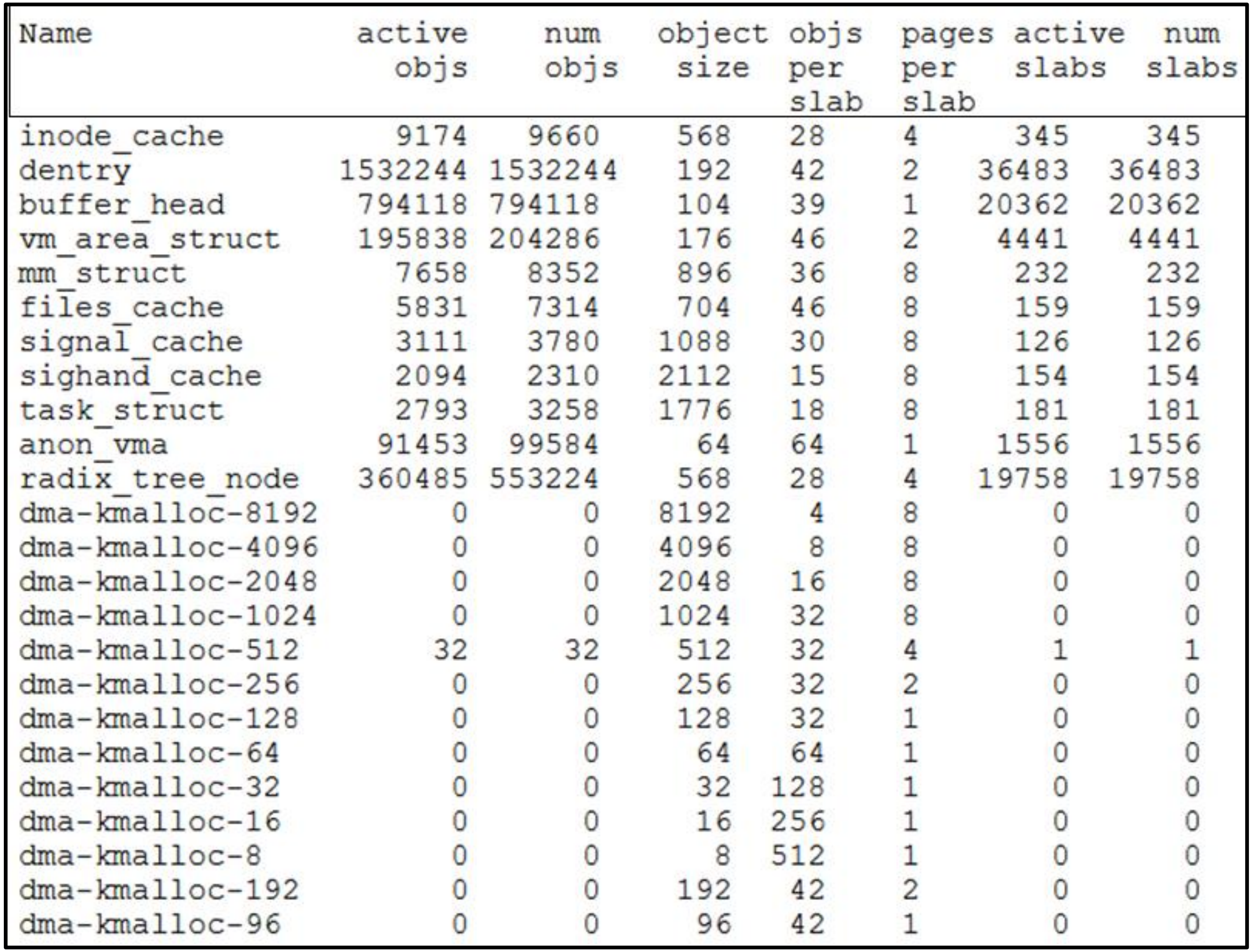
These are **dedicated caches**:

```
extern struct kmem_cache    *vm_area_cachep;
extern struct kmem_cache    *mm_cachep;
extern struct kmem_cache    *files_cachep;
extern struct kmem_cache    *fs_cachep;
extern struct kmem_cache    *sighand_cachep;
```

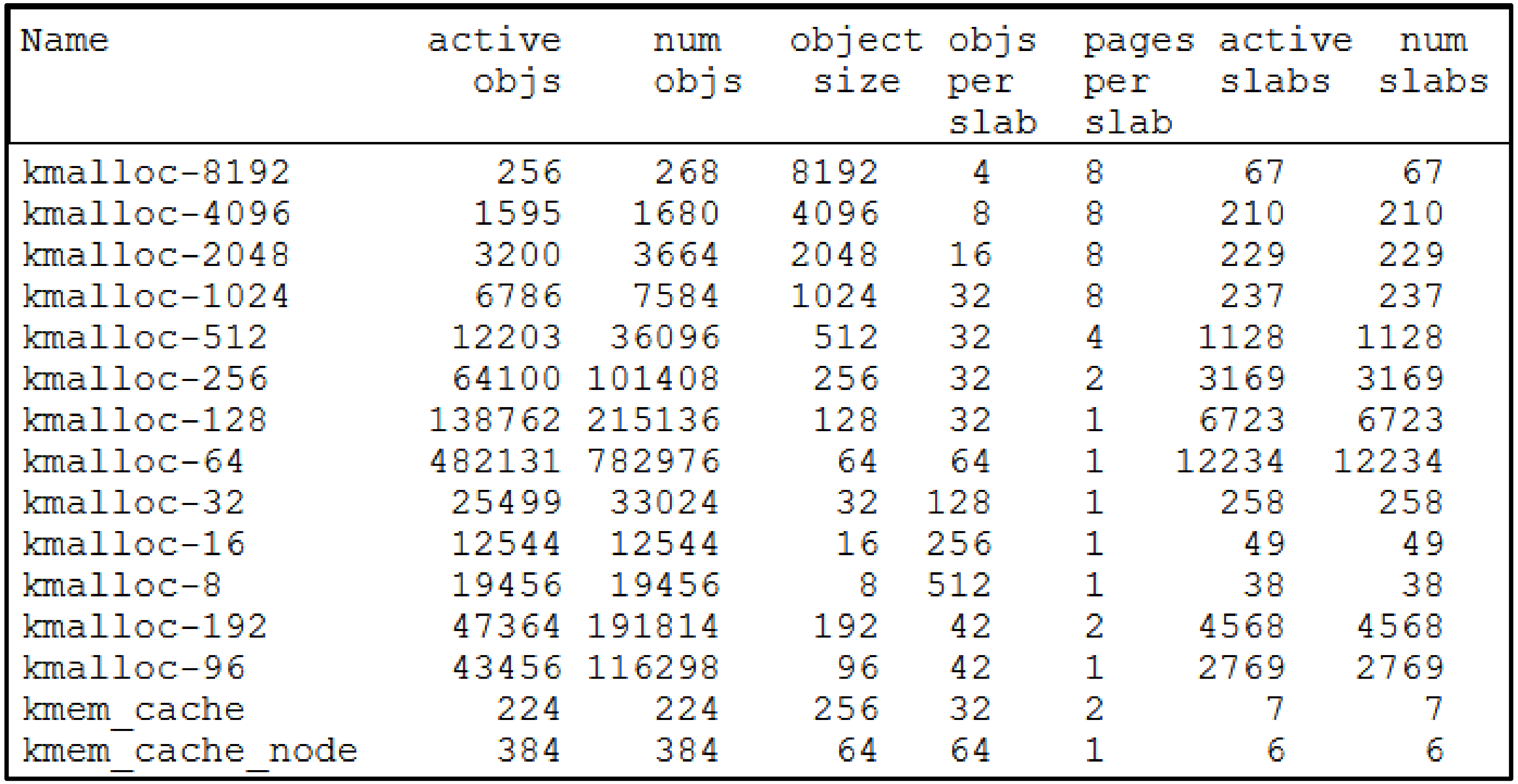
Slab allocator in Linux

This is the sample content of the **/proc/slabinfo**. The columns contain:

name	cache nam
active_objs	number of objects that are currently active (i.e. in use)
num_objs	total number of allocated objects (i.e. objects that are both in use and not in use)
object_size	size of objects in this slab, in bytes
objsperslab	number of objects stored in each slab
pagesperslab	number of pages allocated for each slab
active_slabs	number of active slabs
num_slabs	total number of slabs



Name	active objs	num objs	object size	objs per slab	pages per slab	active slabs	num slabs
inode_cache	9174	9660	568	28	4	345	345
dentry	1532244	1532244	192	42	2	36483	36483
buffer_head	794118	794118	104	39	1	20362	20362
vm_area_struct	195838	204286	176	46	2	4441	4441
mm_struct	7658	8352	896	36	8	232	232
files_cache	5831	7314	704	46	8	159	159
signal_cache	3111	3780	1088	30	8	126	126
sighand_cache	2094	2310	2112	15	8	154	154
task_struct	2793	3258	1776	18	8	181	181
anon_vma	91453	99584	64	64	1	1556	1556
radix_tree_node	360485	553224	568	28	4	19758	19758
dma-kmalloc-8192	0	0	8192	4	8	0	0
dma-kmalloc-4096	0	0	4096	8	8	0	0
dma-kmalloc-2048	0	0	2048	16	8	0	0
dma-kmalloc-1024	0	0	1024	32	8	0	0
dma-kmalloc-512	32	32	512	32	4	1	1
dma-kmalloc-256	0	0	256	32	2	0	0
dma-kmalloc-128	0	0	128	32	1	0	0
dma-kmalloc-64	0	0	64	64	1	0	0
dma-kmalloc-32	0	0	32	128	1	0	0
dma-kmalloc-16	0	0	16	256	1	0	0
dma-kmalloc-8	0	0	8	512	1	0	0
dma-kmalloc-192	0	0	192	42	2	0	0
dma-kmalloc-96	0	0	96	42	1	0	0



Name	active objs	num objs	object size	objs per slab	pages per slab	active slabs	num slabs
kmalloc-8192	256	268	8192	4	8	67	67
kmalloc-4096	1595	1680	4096	8	8	210	210
kmalloc-2048	3200	3664	2048	16	8	229	229
kmalloc-1024	6786	7584	1024	32	8	237	237
kmalloc-512	12203	36096	512	32	4	1128	1128
kmalloc-256	64100	101408	256	32	2	3169	3169
kmalloc-128	138762	215136	128	32	1	6723	6723
kmalloc-64	482131	782976	64	64	1	12234	12234
kmalloc-32	25499	33024	32	128	1	258	258
kmalloc-16	12544	12544	16	256	1	49	49
kmalloc-8	19456	19456	8	512	1	38	38
kmalloc-192	47364	191814	192	42	2	4568	4568
kmalloc-96	43456	116298	96	42	1	2769	2769
kmem_cache	224	224	256	32	2	7	7
kmem_cache_node	384	384	64	64	1	6	6

The memory is physically allocated and initialized with **whole slabs**.

Each slab consists of **one or more page frames** containing both **allocated** and **free** objects.

SLAB cache – data structures

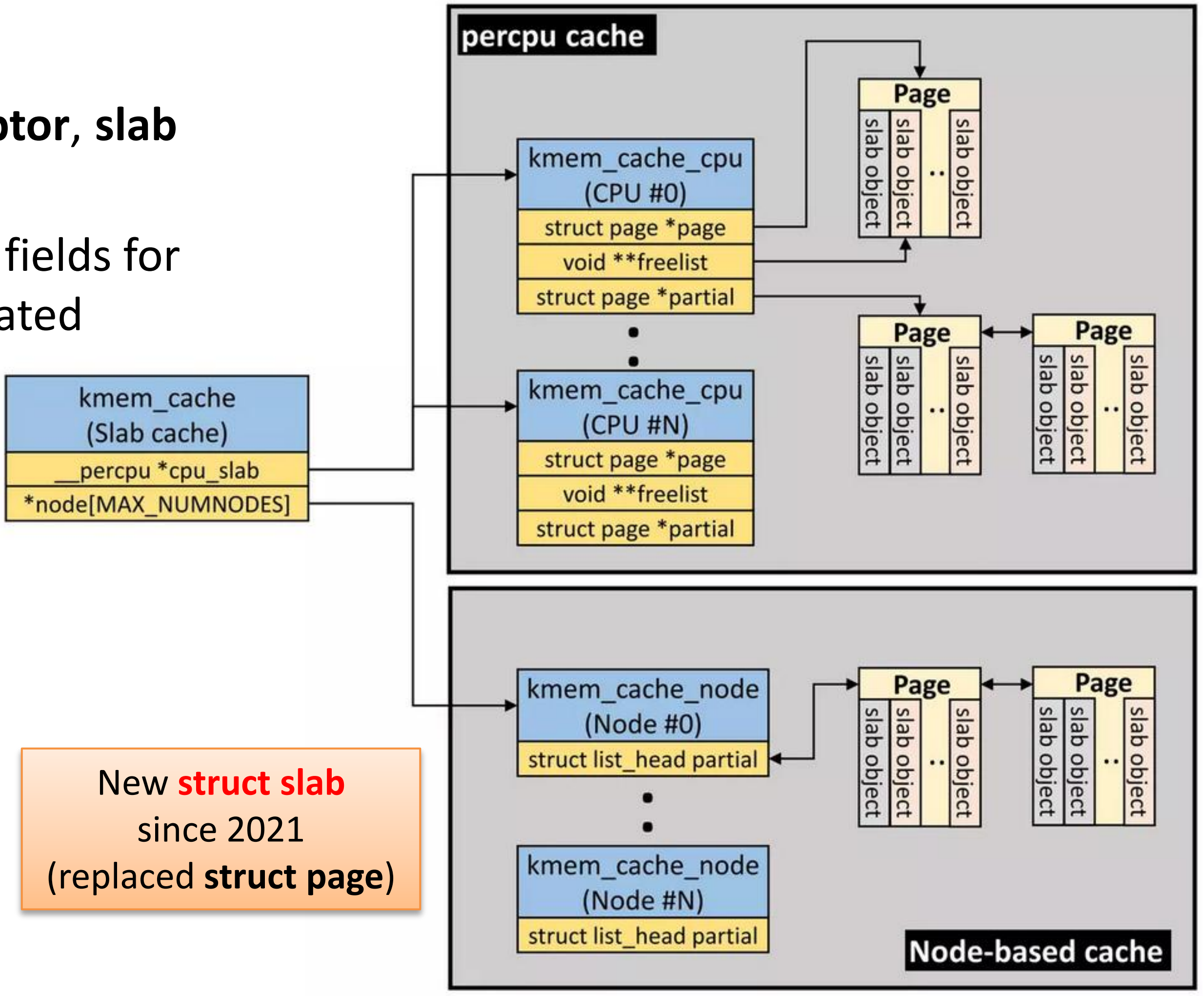
The **SLAB** allocator uses three data structures: **cache descriptor**, **slab descriptor**, **object descriptor**.

In the **cache descriptor** (**kmem_cache**), in addition to many fields for data management (such as the number of free and allocated objects and flags) there are two important elements:

- a pointer to the small **array** through which **recently freed objects for each CPU** can be reached (up to a **limit**). The array has one entry for each CPU.

  The entry points to the **array_cache**, which contains the data needed to manage the objects.

- **array of pointers** (one entry per each **NUMA node**) to the structure **kmem_cache_node**, which contains pointers to three lists: completely full slabs, partially full slabs, free slabs.



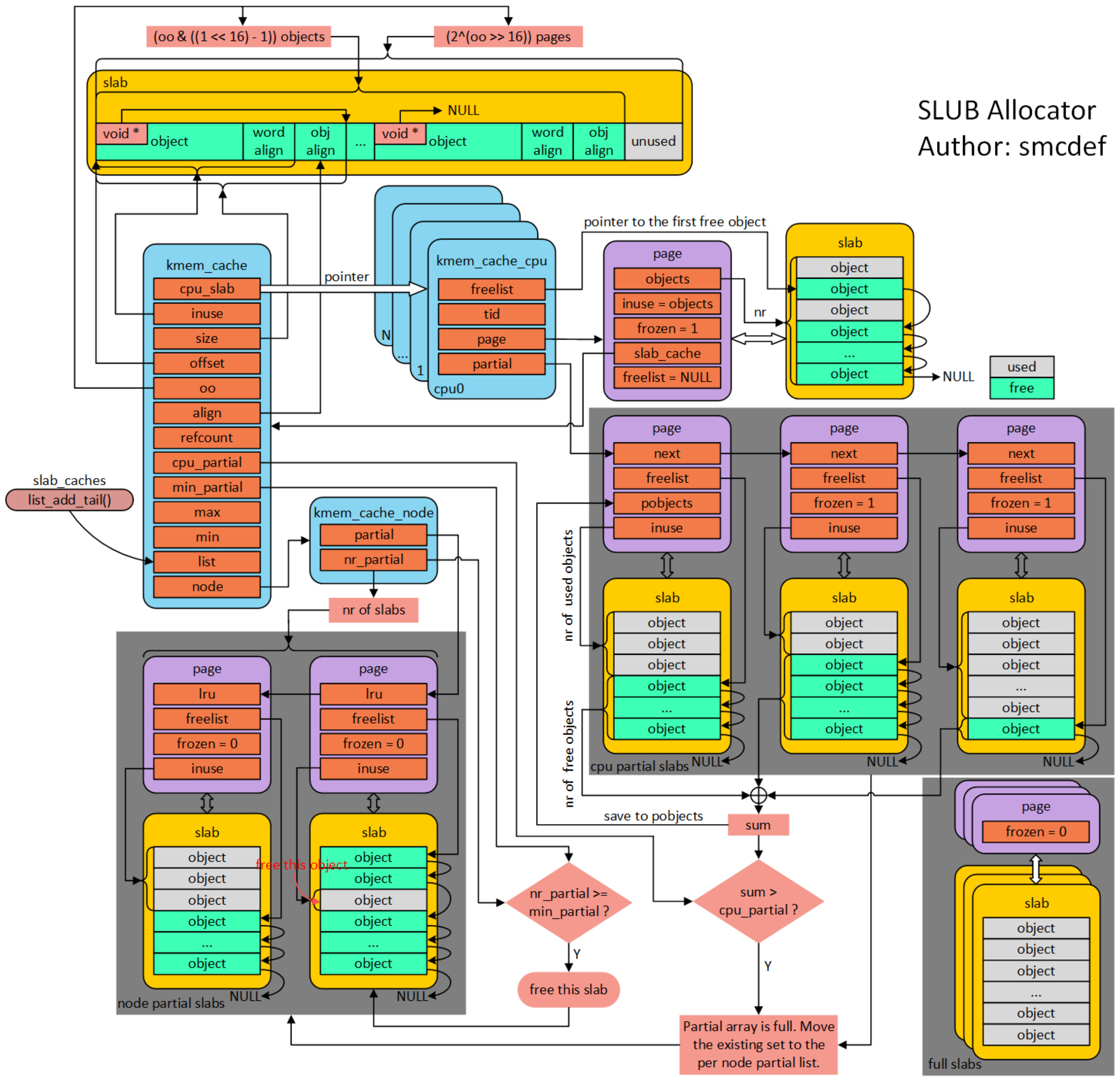
New **struct slab** since 2021 (replaced **struct page**)

(source: Adrian Huang, Slab Allocator in Linux Kernel, 2022, based on kernel 5.11)

SLAB allocator – data structures

https://nasm.re/posts/kmem_cache/
Valid for linux kernel 5.18.12.

New **struct slab** since 2021 (replaced **struct page**)



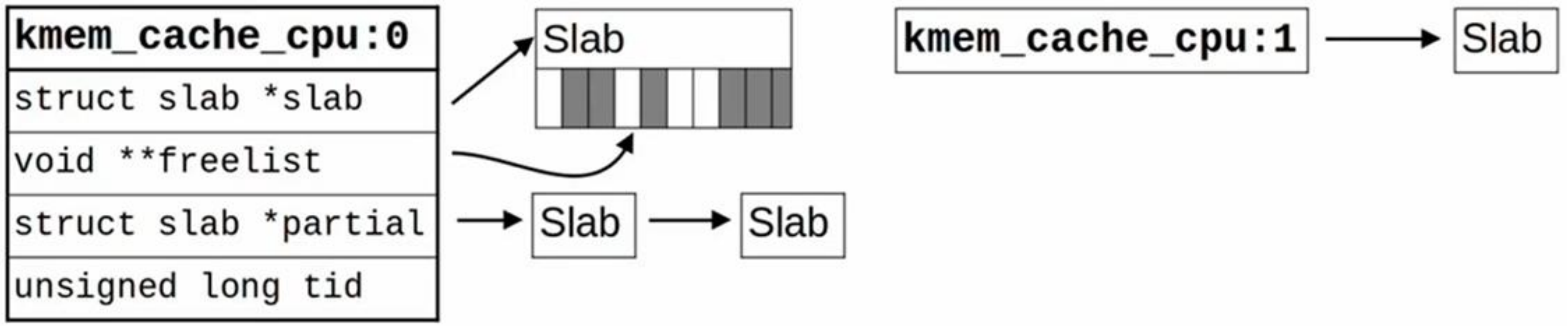
SLAB cache – data structures

SLUB dedicates whole **slab pages** for each **CPU** and also grabs their **freelist** (and can cache more slabs in „cpu partial” lists.

Allocations from **kmem_cache_cpu->freelist** are fast, just a cmpxchg_double().

Freeing if the object belongs to **kmem_cache_cpu->slab** – equally fast:

- But that's much less likely than object belonging to the same **NUMA** node,
- Slab belongs to another **CPU** – also cmxchg_double() but likely more expensive,
- Slab in on a list – might need to be taken off in some cases, which needs the **spinlock**.



The rise and fall of kernel slab allocators (Vlastimil Babka, SUSE Labs Conference 2023)

SLAB allocator – data structures

The **per-CPU pointers** are important to best exploit the **CPU caches**. The **LIFO** principle is applied when objects are **allocated** and **returned**.

Only when the per-CPU caches are empty are free objects from the slabs used to refill them.

Allocation of objects takes place on three levels and the **cost** of allocation and the **negative impact** of these operations on a **processor cache** and **TLB** increases from level to level:

1. Per-CPU objects from the CPU cache.
2. Unused objects from an existing slab.
3. Unused objects from the new slab obtained on request from the buddy allocator.

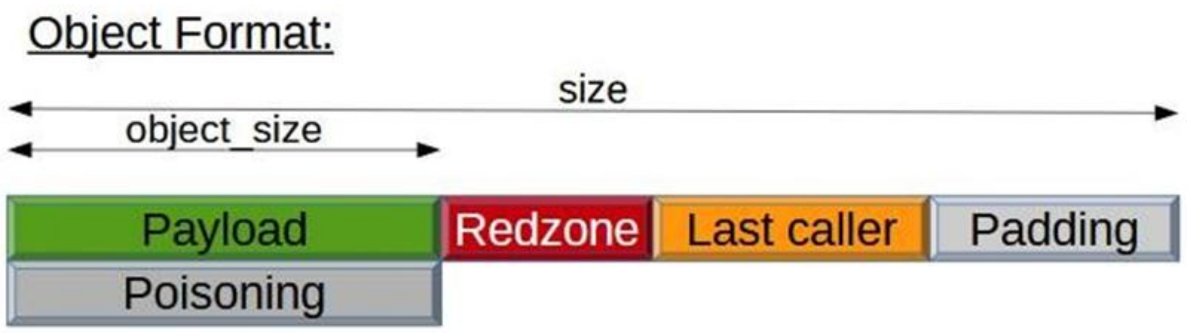
SLAB allocator – data structures

The **redzone** is used to detect **writes** after the object. All bytes should always have the **same value**. If there is any deviation then it is due to a **write after the object boundary**. (Redzone information is only available if SLAB_RED_ZONE is set.)

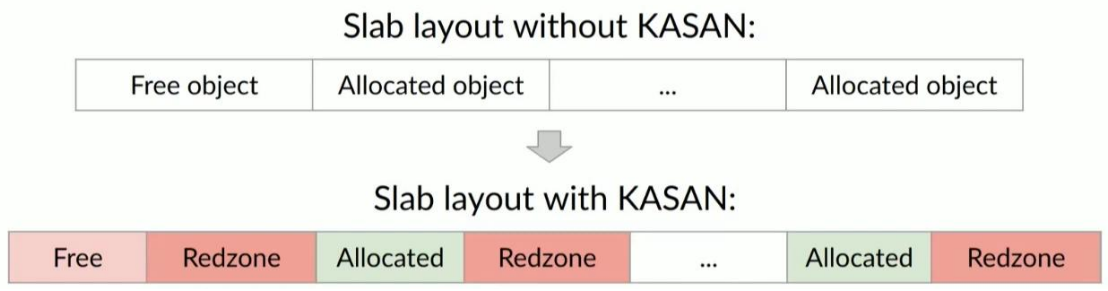
If the object is **inactive** then the bytes typically contain **poison values**. Any non-poison value shows a corruption by a **write after free**.

Padding is an **unused data** to fill up the space in order to get the next object properly aligned.

The kernel sets the page flag **PG_slab** for each physical page, that is allocated for the slab allocator.



- Slab is fully poisoned (marked as inaccessible) when allocated
- Objects in slab are unpoisoned when allocated
- Always-poisoned redzones between objects (⇒ fewer objects fit in slab)



SLAB allocator – slab coloring

Disclaimer – probably not present in current verion of the kernel

The final task of the SLAB allocator is **optimal hardware cache use**.

If there is space left over after objects are packed into a slab, the remaining space is used to color the slab. Slab coloring is a scheme that attempts to have **objects in different slabs use different lines in the cache**.

By placing objects at a **different starting offset** within the slab, objects will likely use **different lines** in the **CPU cache**, which helps ensure that objects from the same slab cache will be **unlikely** to **flush** each other.

Space that would otherwise be wasted fulfills a new function.

To use the hardware cache better, the slab allocator will **offset** objects in different slabs by different amounts depending on the amount of space left over in the slab.

During cache creation, it is calculated how many objects can fit on a slab and how many bytes would be wasted. Based on wastage, two figures are calculated for the cache descriptor:

- **colour**: the number of different offsets that can be used,
- **colour_off**: the multiple to offset each object in the slab.

SLAB allocator – slab coloring

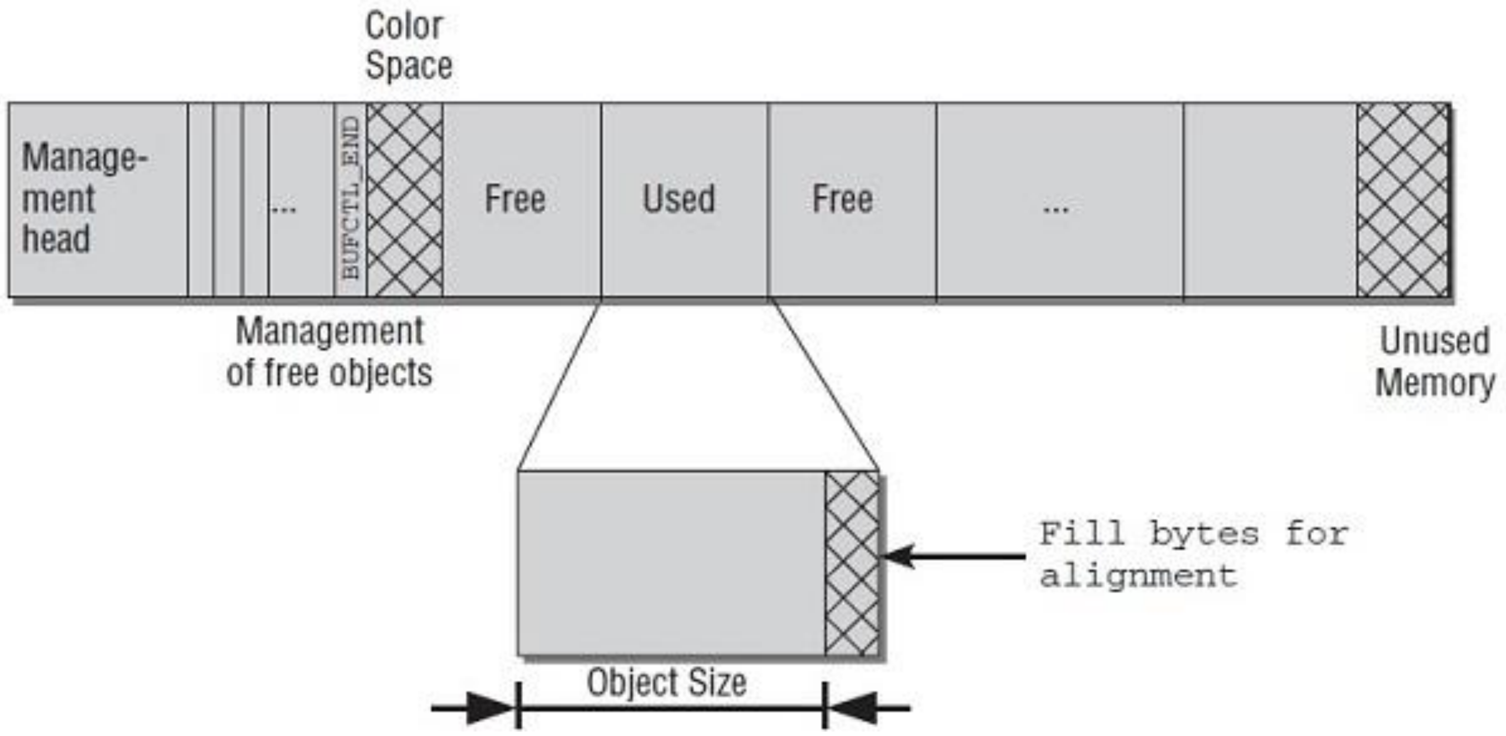
Disclaimer – probably not present in current verion of the kernel

Slabs having different colors store the first object of the slab in different memory locations, while satisfying the alignment constraint.

Coloring leads to **moving some of the free area of the slab from the end to the beginning**.

The various colors are distributed equally among slabs of a given object type.

Each slab is created with a different color from the previous one, up to the maximum available colors.



SLAB coloring (source: W. Mauerer, Professional Linux Kernel Architecture)

Slab allocator API

The function used to initialize **general caches**:

```
void __init kmem_cache_init(void)
```

The function used to create **dedicated caches**:

```
/**
 * kmem_cache_create - Create a cache.
 * @name: A string which is used in /proc/slabinfo to identify this cache.
 * @size: The size of objects to be created in this cache.
 * @align: The required alignment for the objects.
 * @flags: SLAB flags
 * @ctor: A constructor for the objects.
 */

struct kmem_cache *
kmem_cache_create (const char *name, size_t size,
      size_t align, slab_flags_t flags, void (*ctor)(void*))
```

Slab allocator API

The allocation of the object from **dedicated cache memory** is performed by **kmem_cache_alloc()** function. The function first attempts to allocate the object in a **partially** filled slab, then in a **free** slab. If it fails, it tries to **allocate new frames** from the **buddy allocator**.

The **kmem_cache_free()** function is used to free memory allocated for the kernel. Releasing **empty** slabs occurs while **deleting** the cache. The function of the buddy allocator will be finally called here.

```
void *kmem_cache_alloc(kmem_cache_t *cache, gfp_t flags);
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

Examples of calls to the memory allocation function in the kernel code.

```
tmp = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL)
struct fs_struct *fs = kmem_cache_alloc(fs_cachep, GFP_KERNEL);
newf = kmem_cache_alloc(files_cachep, GFP_KERNEL);
pgd_t *pgd_alloc(struct mm_struct *mm)
{
	return kmem_cache_alloc(pgd_cachep, PGALLOC_GFP);
}
```

Slab allocator API

The **kmalloc()** function, similar to the **malloc()** function used in **user** mode, is used to allocate memory for the **kernel** from **general purpose caches** (the parameter is the number of bytes, not the object type).

The function determines the size being the nearest multiple of 2 (rounded up), and then calls **kmem_cache_alloc()** indicating the appropriate cache.

The function **kfree()** is used to release such objects, which is equivalent to **free()** from user mode.

```
void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *objp);
```

Examples of calls to the memory allocation function in the kernel code.

```
c->wbuf = kmalloc(c->wbuf_pagesize, GFP_KERNEL);
/* don't ask for more than the kmalloc() max size */
if (size > KMALLOC_MAX_SIZE)
    size = KMALLOC_MAX_SIZE;
buf = kmalloc(size, GFP_KERNEL);
if (!buf)
    return -ENOMEM;
```

*The **kmalloc() function can not allocate high mem**.*

*There is a **limit on the memory size allocated by kmalloc()**.*

It depends on the architecture and configuration of the kernel.

It can be assumed that this limit is 128 KB (32 frames size 4 KB).

Memory management – address types

SUMMARY!

Based on:

- LDD 3rd ed., Allocating Memory (chapter 8),
- Linux Kernel Documentation.

User virtual addresses – these are the regular addresses seen by **user-space programs**. Each process has its own **virtual address space**.

Physical addresses – the addresses used between the processor and the system's memory.

Kernel logical addresses – these make up the normal address space of the kernel. These addresses map some portion (perhaps all) of main memory and are often treated as if they were physical addresses.

On most architectures, **logical** addresses and their associated **physical** addresses differ only by a **constant offset**.

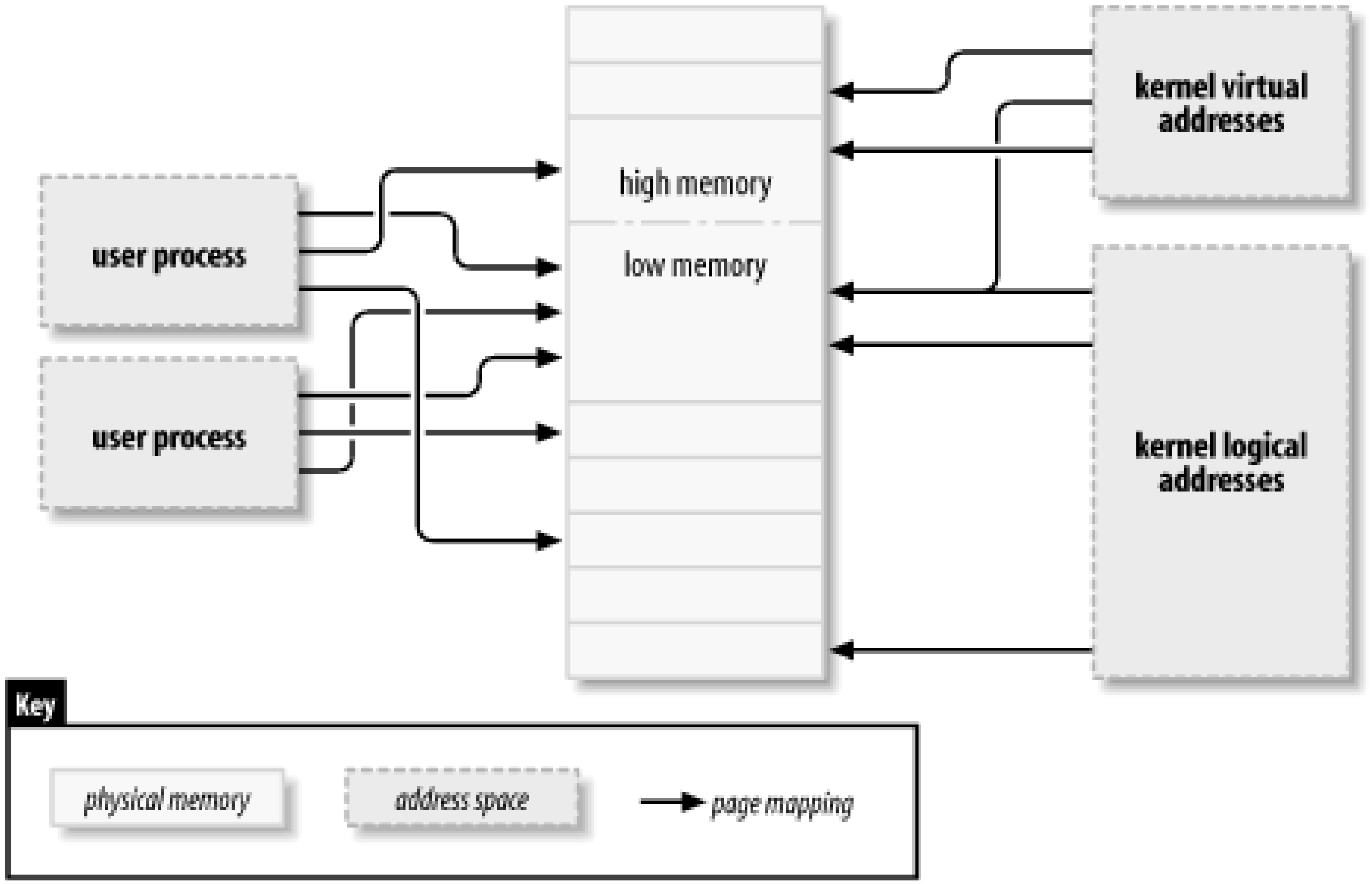
Logical addresses use the hardware's native **pointer size** and, therefore, may be unable to address all of physical memory on **32-bit systems**.

Logical addresses are usually stored in variables of type **unsigned long** or **void ***. Memory returned from **kmalloc()** has a **kernel logical address**.

Kernel virtual addresses – they are similar to logical addresses in that they are a mapping from a kernel-space address to a physical address.

Memory management – address types

SUMMARY!



Address types in Linux (source: Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini, Linux Device Drivers, 3rd Edition)

Memory management – address types

SUMMARY!

Kernel virtual addresses **do not** necessarily have the **linear**, one-to-one mapping to physical addresses that characterize the logical address space.

All **logical** addresses are kernel **virtual** addresses, but many kernel virtual addresses are not logical addresses.

Memory allocated by **vmalloc()** has a virtual address (but no direct physical mapping).

The **kmap()** function also returns **virtual addresses**. Virtual addresses are usually stored in **pointer** variables.

If you have a **logical** address, the macro **__pa()** returns its associated **physical** address.

Physical addresses can be mapped back to **logical** addresses with **__va()**, but only for **low-memory pages**.

The (**virtual**) address range used **by kmalloc()** and **__get_free_pages()** features a **one-to-one** mapping to **physical** memory, possibly shifted by a constant **PAGE_OFFSET** value, the functions don't need to modify the page tables for that address range.

The address range used by **vmalloc()**, on the other hand, is completely synthetic, and each allocation builds the (**virtual**) **memory area** by suitably setting up the **page tables**.

This difference can be perceived by **comparing** the **pointers** returned by the allocation functions.

On some platforms (e.g. x86), addresses returned by **vmalloc()** are just beyond the addresses that **kmalloc()** uses.

Memory management – high and low memory

SUMMARY!

Low memory – memory for which **logical** addresses **exist** in **kernel** space.

High memory – memory for which **logical** addresses **do not exist**, because it is beyond the address range set aside for kernel virtual addresses.

Kernel functions that deal with memory are increasingly using pointers to **struct page**.

This data structure contains the field **void *virtual**, which keeps the **kernel virtual address** of the page, if it is mapped, **NULL**, otherwise.

Low-memory pages are always **mapped**, **high-memory** pages usually are **not**.

If you want to look at this field, the proper method is to use the **page_address()** macro.

It returns the **kernel virtual address** of this page, if such an address exists.

For **high memory**, that address exists only if the page has been **mapped**.

In most situations, you want to use a version of **kmap()** rather than **page_address()**.

Function **kmap()** returns a **kernel virtual address** for any page in the system.

For **low-memory pages**, it just returns the **logical address** of the page; for **high-memory pages**, **kmap()** creates a special **mapping** in a dedicated part of the kernel address space. A limited number of such mappings is available, so it is better not to hold on to them for too long.

Many **kernel data structures** must be placed in **low memory**, **high memory** tends to be reserved for **user-space** process pages.

Memory management API

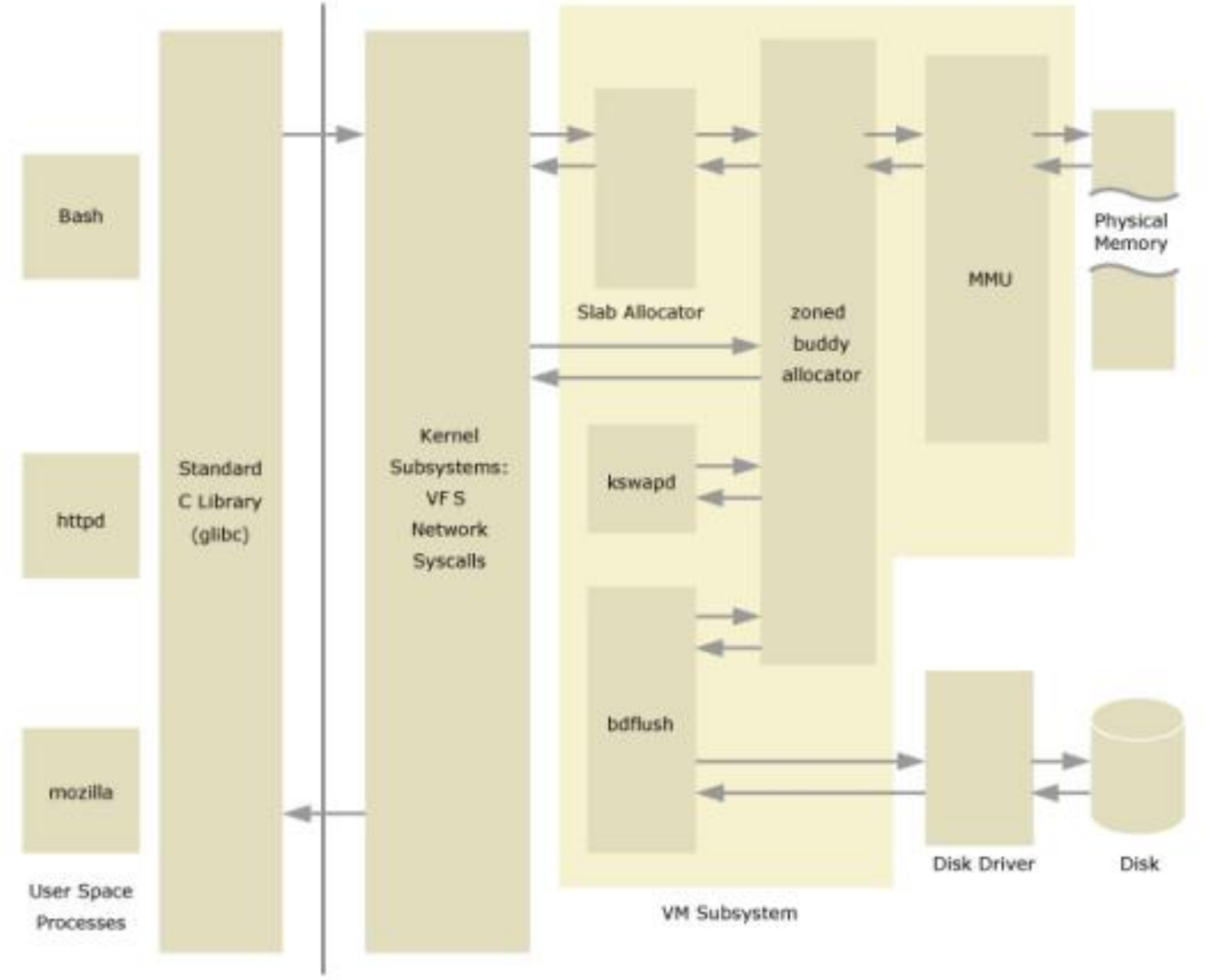
SUMMARY!

Function **kmap()** can sleep if no mappings are available. For **<u>non-blocking</u>** mapping use **kmap_atomic()**.

The kernel obtains **dynamic memory** by calling functions:

- **__get_free_pages()** or **alloc_pages()** to get **contiguous** page frame areas from the **buddy allocator**,
- **kmem_cache_alloc()** or **kmalloc()** to get objects from the **slab allocator** (**specialized** or **general purpose**),
- **vmalloc()** to get a **noncontiguous area of memory**.

If memory is available, the kernel requests are executed immediately and the function returns the address of the **page descriptor** or the **linear address** identifying the allocated dynamic memory area.



High overview of Virtual Memory subsystem
(source: N. Murray, N. Horman, Understanding Virtual Memory)