Managing Physical Memory

Advanced Operating Systems

Overview

- Physical Memory (PM)
- PM Management in Linux
  - Zones, Nodes, Pages
  - The Memblock Allocator
  - The Buddy Allocator
- PM Management in OpenLSD
  - The Boot Allocator
  - Frame Allocation
- Advanced Linux Features
  - Overlay allocators (slab, vmalloc)
  - Sanity checks

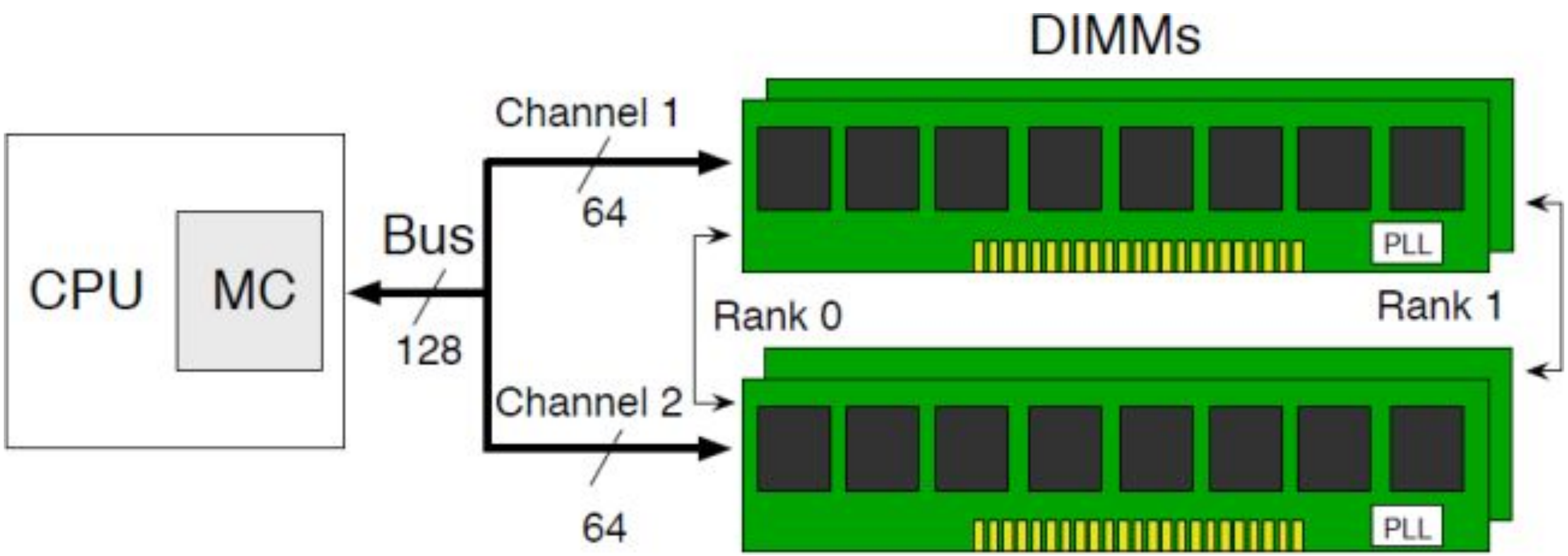
What is “Physical Memory” Anyway?

- **DRAM** (**D**ynamic **R**andom **A**ccess **M**emory)
  - Organized in memory cells with 1-bit data
  - Each cell has 1 capacitor + 1 transistor
  - Charge/discharge capacitor = 1/0 bit value
  - Capacitors leak charge → Need for periodic refresh
  - Key difference w.r.t. SRAM (used in CPU caches)
  - Typical refresh rates: 8ms - 64ms

What is “Physical Memory” Anyway?

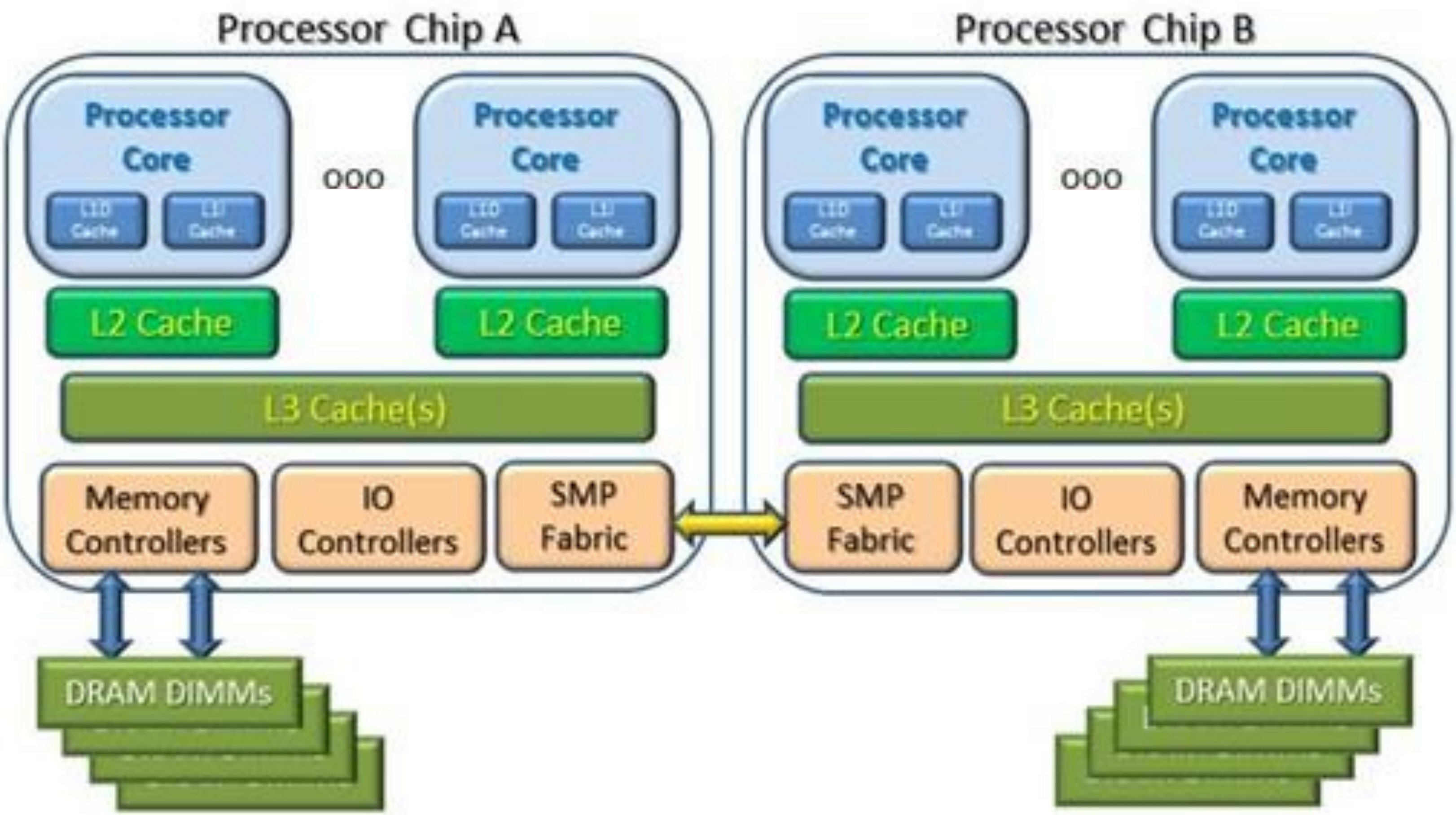
- **DRAM organization**
  - Channel
  - DIMM
  - Rank
  - Chip
  - Bank
  - Row/Column

What is “Physical Memory” Anyway?



- 1 / 2 channels, N DIMMs per channel
- 1 / 2 ranks per DIMM

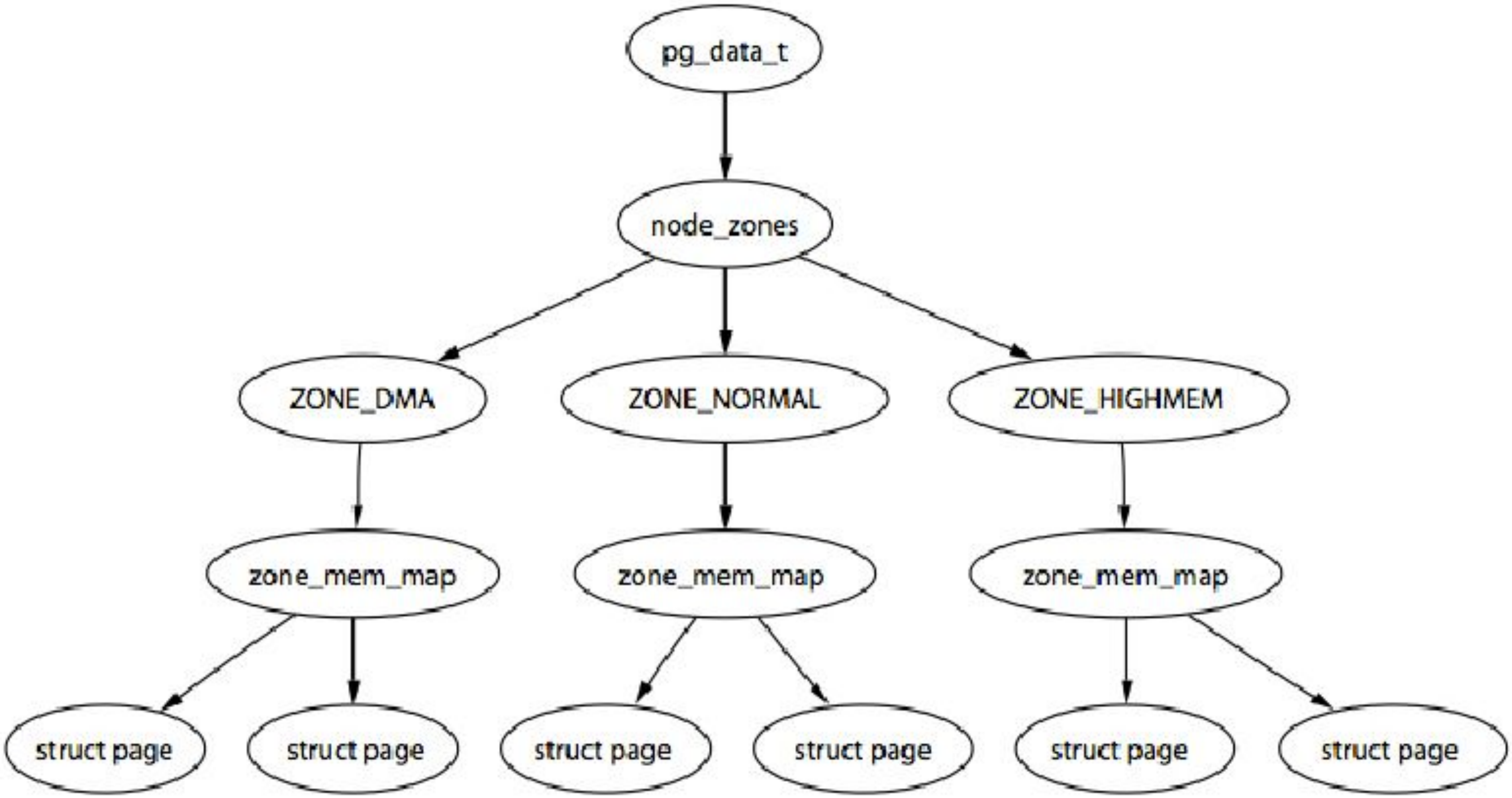
The Big Picture: UMA vs. NUMA



Managing Physical Memory: Linux

- Logically divided in a number of consecutive physical memory pages (page frames) identified by page frame numbers (PFNs)
- How are frames organized on Linux?
  - Nodes
    - The NUMA abstraction of N “banks”
  - Zones
    - Tagged regions for each node
  - Pages
    - Physical page frames for each zone

Managing Physical Memory: Linux



Managing Physical Memory: Nodes

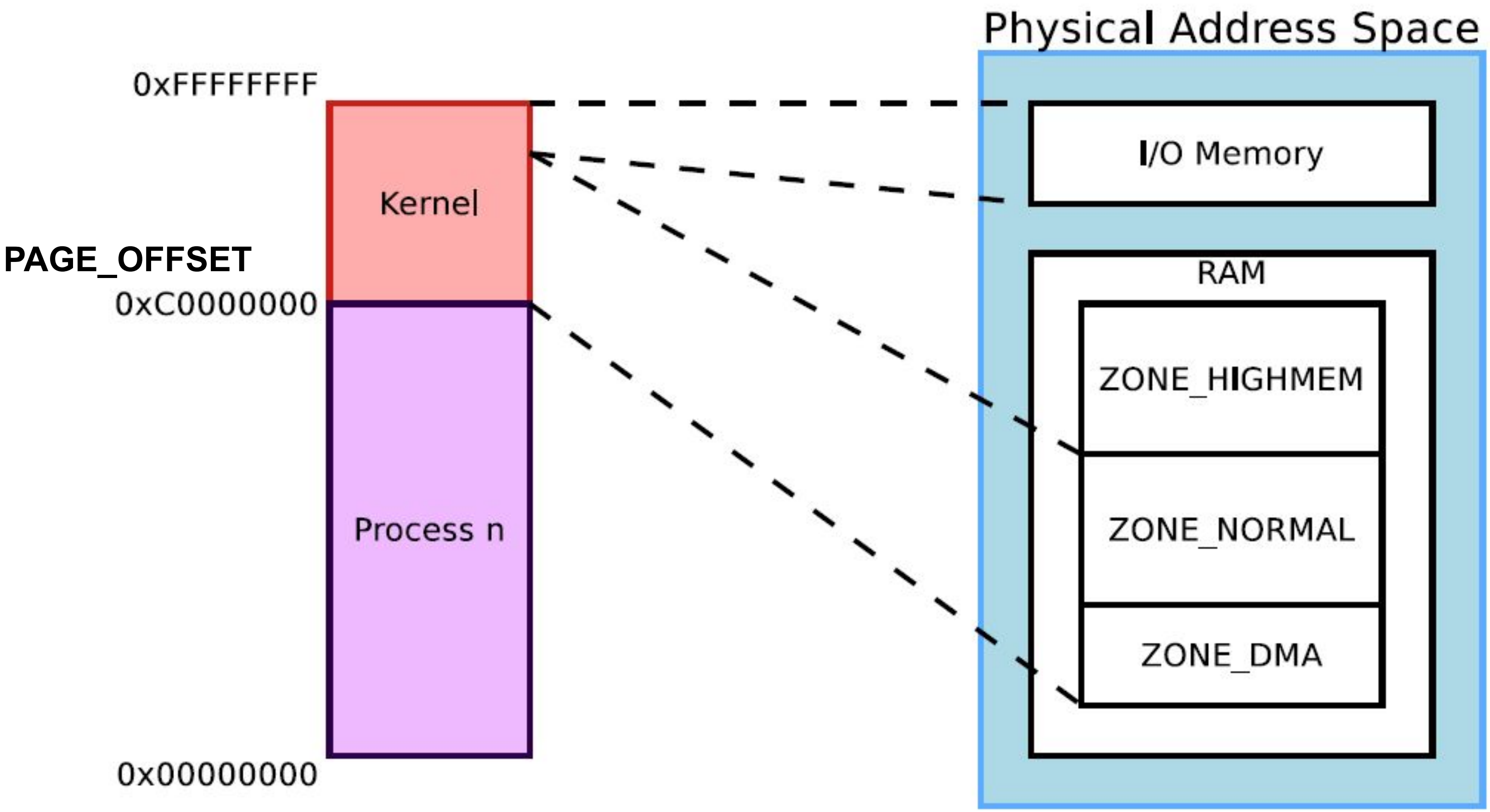
linux/mmzone.h:

```
662 typedef struct pglist_data {
            struct zone node_zones[MAX_NR_ZONES];
            struct zonelist node_zonelists[MAX_ZONELISTS];
            int nr_zones;
            struct page *node_mem_map;
            unsigned long node_start_pfn;
            unsigned long node_present_pages; /* no. of phys pages */
            unsigned long node_spanned_pages; /* size of physical page
                                                 range, with holes */
            int node_id;
    #ifdef CONFIG_COMPACTION
            /* ... */
    #endif
    } pg_data_t;
```

Managing Physical Memory: Zones

- Zone ranges:
  - ZONE_DMA: 0 MB - 16 MB (DMA)
  - ZONE_DMA32: 16 MB - 4 GB (DMA32)
  - ZONE_NORMAL: 16 MB - 896 MB (kernel)
  - ZONE_HIGHMEM: 896 MB - End (kmap, x86)
- Managed independently
- When under memory pressure:
  - Zone boundaries become “blurry”
  - Watermarking strategy to free pages (**kswapd**):
    - WMARK_MIN: Trigger direct reclaim
    - WMARK_LOW: Trigger async reclaim
    - WMARK_HIGH: Stop reclaiming

Managing Physical Memory: Zones (x86)



Managing Physical Memory: Zones

linux/mmzone.h:

```
385 struct zone {
          unsigned long _watermark[NR_WMARK];
          struct pglist_data      *zone_pgdat;
          struct per_cpu_pageset __percpu *pageset;
          unsigned long           zone_start_pfn;
          unsigned long           spanned_pages;
          unsigned long           present_pages;
          const char              *name;
          struct free_area        free_area[MAX_ORDER];
          spinlock_t              lock;
          ZONE_PADDING(_pad2_)
          atomic_long_t           vm_stat[NR_VM_ZONE_STAT_ITEMS];
    } ____cacheline_internodealigned_in_smp;
```

Managing Physical Memory: Pages

- 4KB units of physical memory
- `struct page` objects in a `mem_map` array

linux/mmzone.h:

```
642     /*     The     array     of     struct     pages     */
    extern struct page *mem_map;
```

asm/page.h

```
59 #define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
69 #define virt_to_page(kaddr) pfn_to_page(__pa(kaddr) >> PAGE_SHIFT)
71 extern bool __virt_addr_valid(unsigned long kaddr);
```

asm-generic/memory_model.h:

```
33 #define pfn_to_page(pfn)     (mem_map + ((pfn) - ARCH_PFN_OFFSET))
```

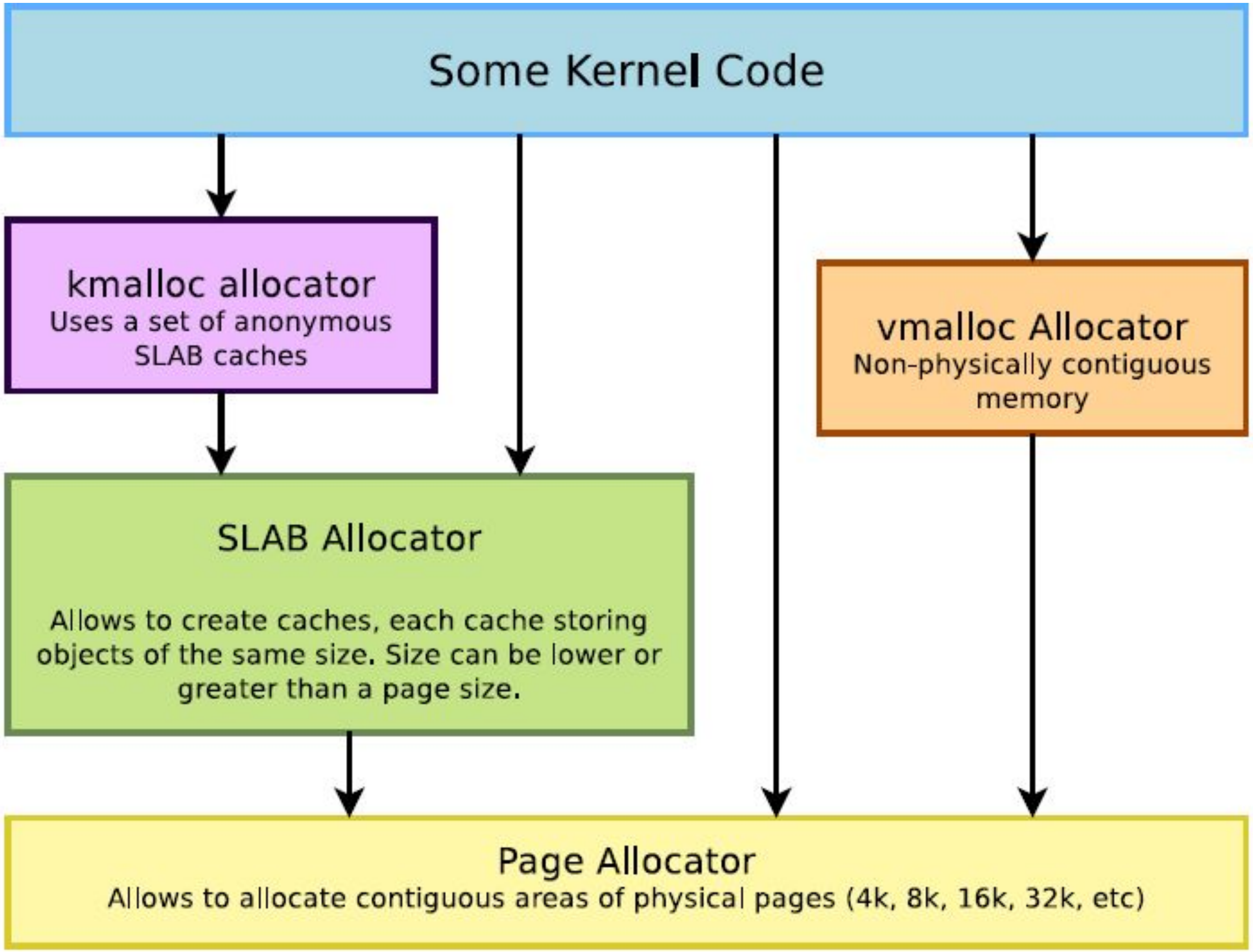
Managing Physical Memory: Pages

linux/mm_types.h:

```
68 struct page {
    unsigned long flags;
    struct list_head lru;
    struct address_space *mapping;
    pgoff_t index;
    atomic_t _refcount;
    struct mem_cgroup *mem_cgroup;
    void *virtual;
} _struct_page_alignment;
```

```
$ cat /proc/*info* | more
```

Allocating Memory: Linux Overview

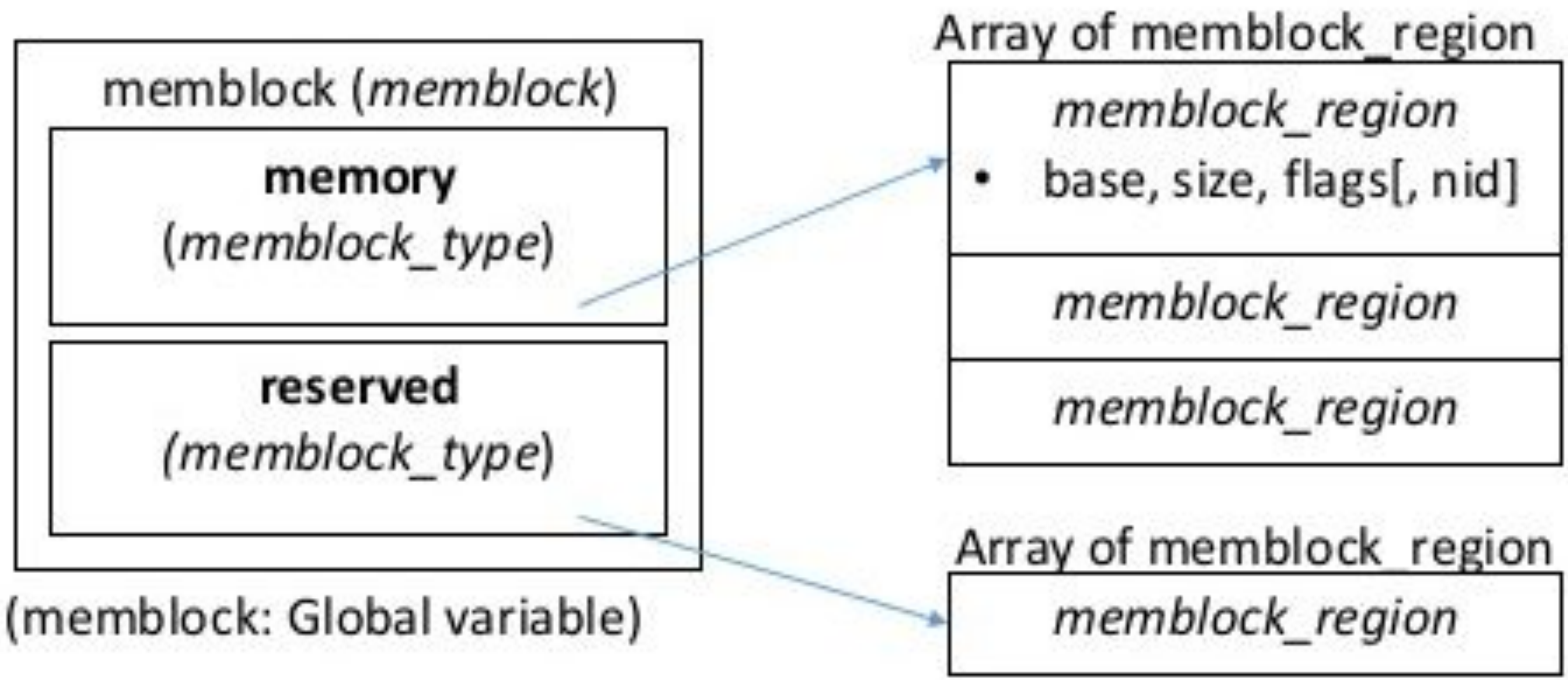


Memblock Allocator

The Linux Memblock Allocator

- Early boot-time (low memory) allocator
- Replaces the old bootmem allocator
- Mostly used to initialize buddy allocator
- Discarded after initialization
- Consists of two main arrays:
    - **`memory`**: all present memory in the system
    - **`reserved`**: allocated memory ranges
- Allocates by finding regions in:
    - **`memory && !reserved`**

The Linux Memblock Allocator

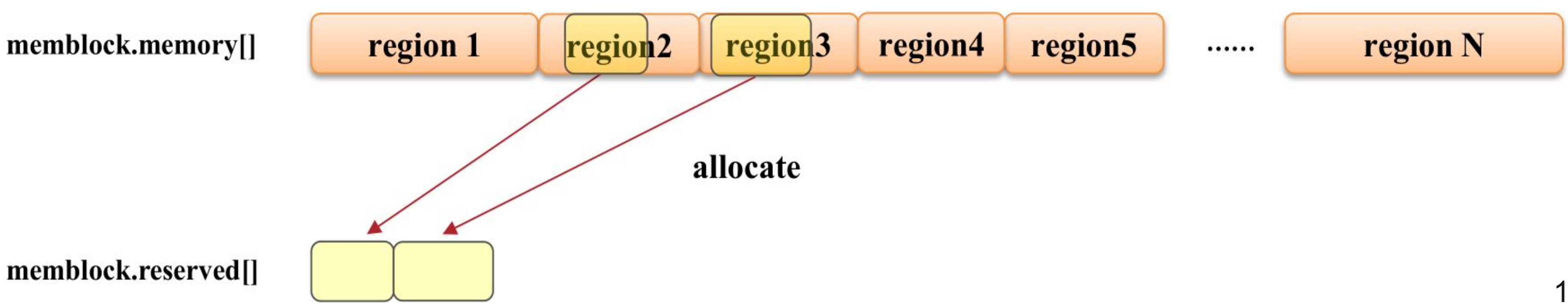


The Linux Memblock Allocator

linux/memblock.h:

```
struct memblock_region {
    phys_addr_t base;
    phys_addr_t size;
    enum memblock_flags flags;
    /* … */
};
```

```
struct memblock_type {
    unsigned long cnt; /* #regions */
    unsigned long max; /* alloc size */
    phys_addr_t total_size; /* tot size */
    struct memblock_region *regions;
    char *name;
};
```

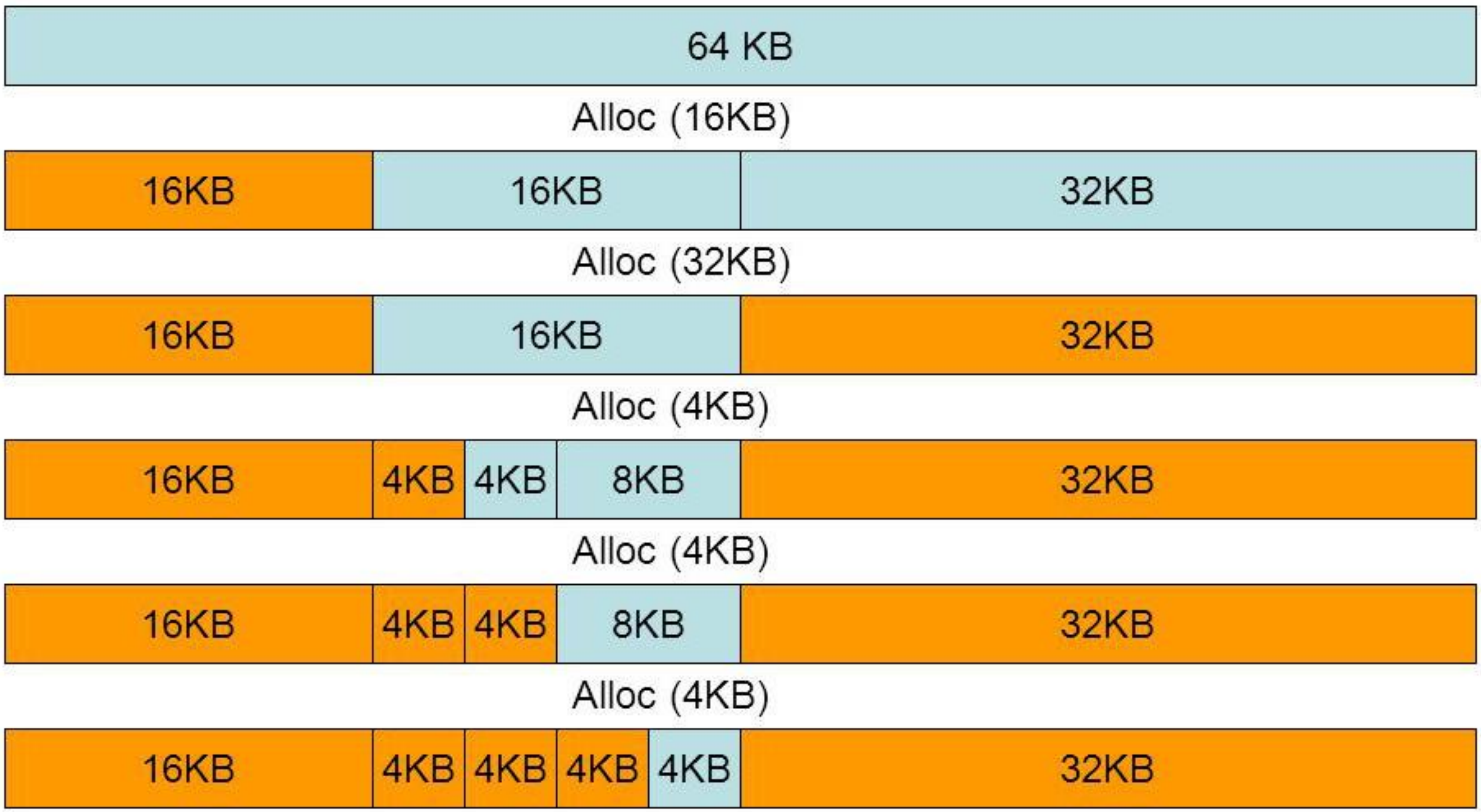


The Linux Memblock Allocator

- Setup:
  - Add all available phys memory regions to `memory`
  - Add “reserved” ones to `reserved`
  - All regions sorted by `base` address
- Allocation:
  - First-fit `memory && !reserved`
  - Simply `memblock_add_range` to `reserved`
  - Merge neighboring regions as necessary
- Deallocation:
  - Linear scan in `reserved` for containing region
  - Simply `memblock_remove_range` from `reserved`
  - First split region if necessary

The Linux Buddy Allocator

- Power-of-two allocator with free coalescing
- Blocks are arranged in 2^N (N=order) pages
- Allocations are satisfied by exact N
  - If not possible, split larger 2^(N+1) block to 2 x 2^N
  - The two smaller blocks are called *buddies*
  - If not possible, split larger 2^(N+2) block twice, etc...
- Deallocations return block to allocator
  - If buddy is free, coalesce into a larger block
  - If larger block’s buddy is free, coalesce again, etc...
- Behavior in short: recursive 2^N split/merge operations on allocation/deallocation



The Linux Buddy Allocator: Split (and Merge)

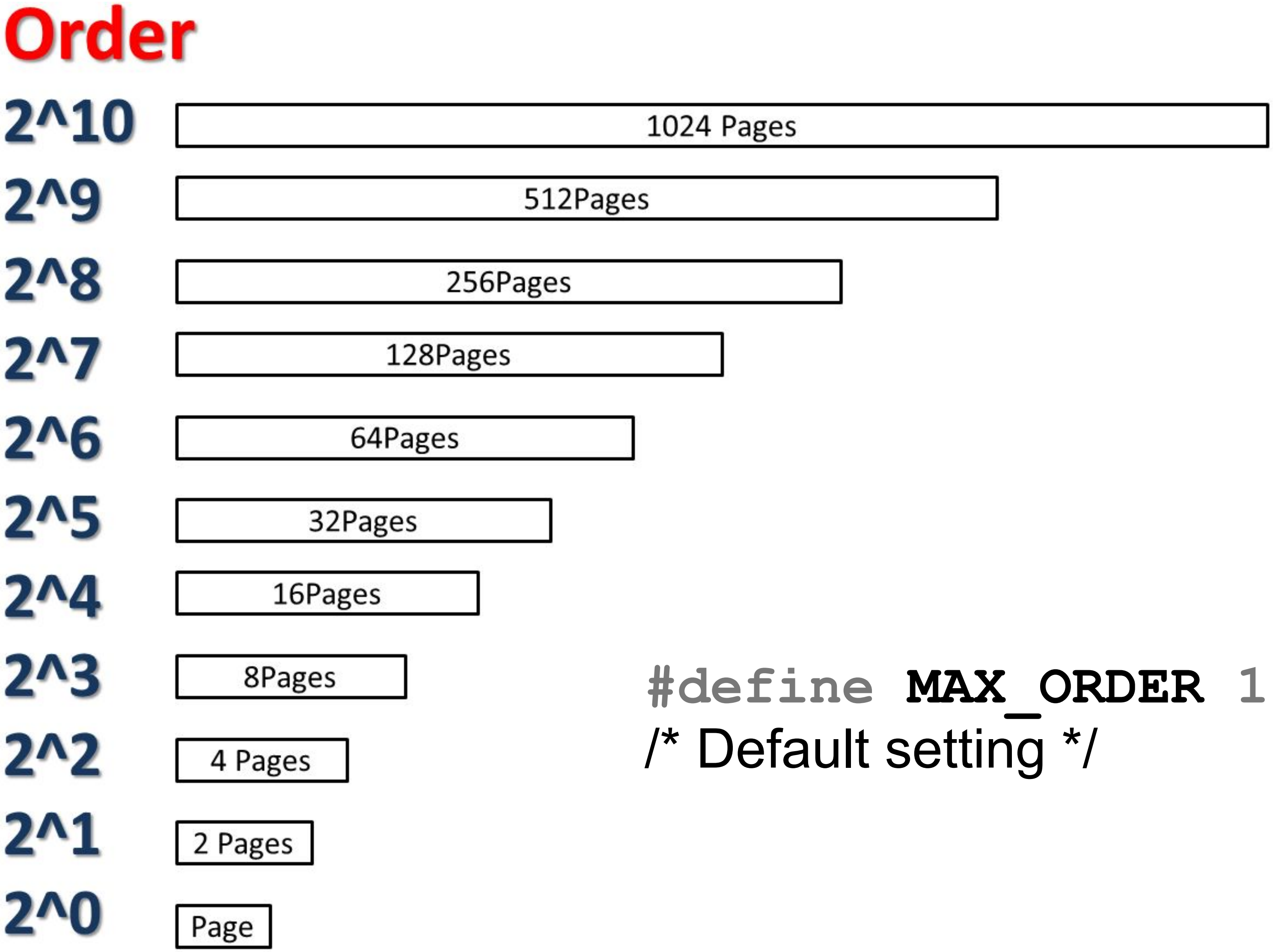
The Linux Buddy Allocator: Internals

- Per node,zone array of `MAX_ORDER` freelists
  - Freelist N maintains free blocks of size 2^N
  - Split/merge: moves block(s) to previous/next freelist

linux/mmzone.h:

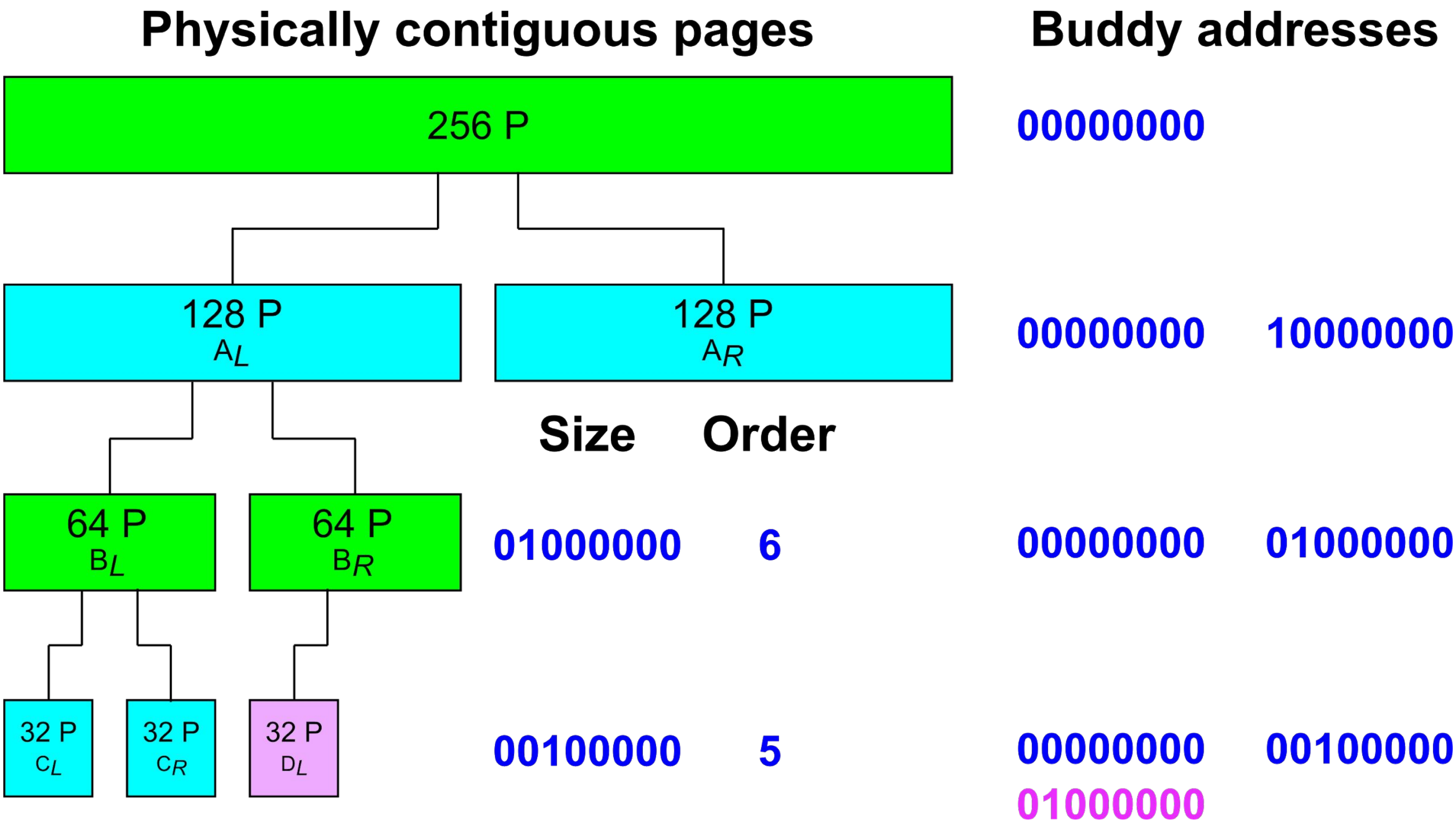
```
98 struct free_area {
        struct list_head free_list[MIGRATE_TYPES];
        unsigned long    nr_free;
   };
```

- `PG_buddy`, order in page attributes (set if free)
- Buddy operations
  - O(1) *reserve* (update page flags)
  - O(1) *lookup* (flip “order” bit in block address)

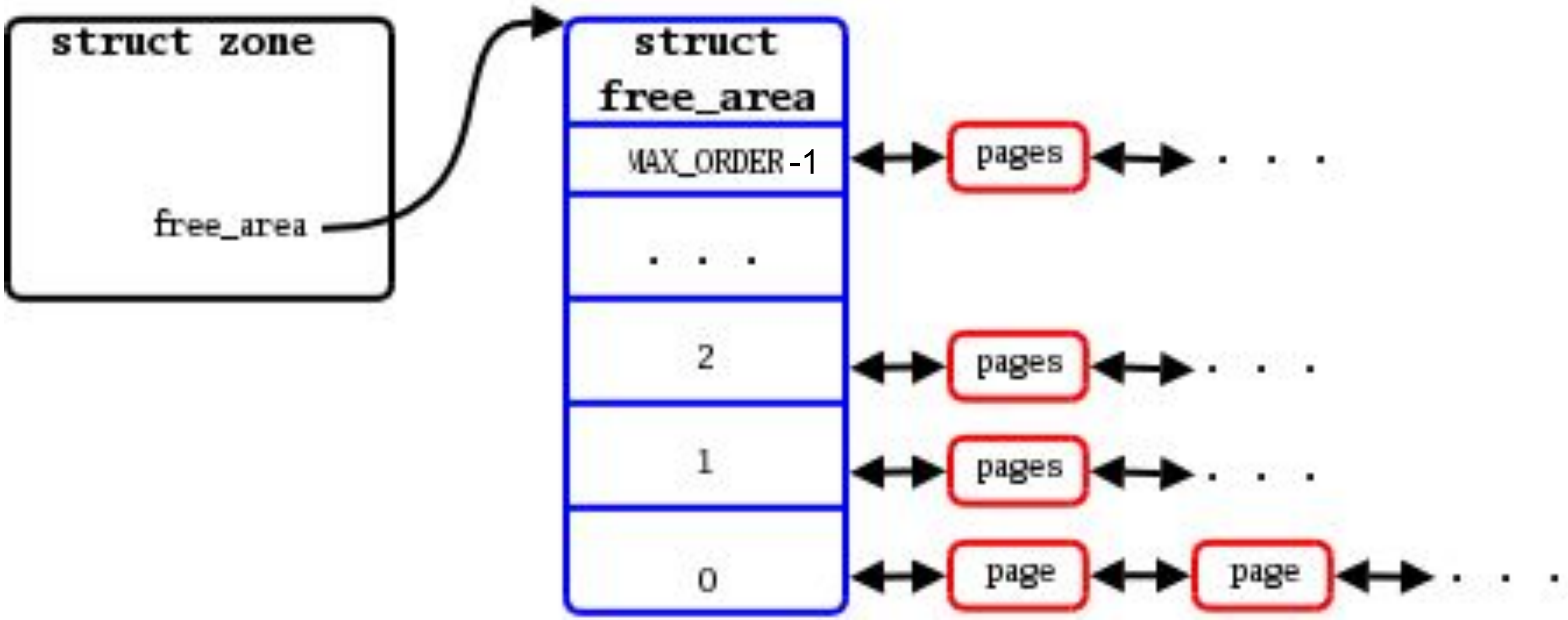



```
#define MAX_ORDER 11
```
/* Default setting */

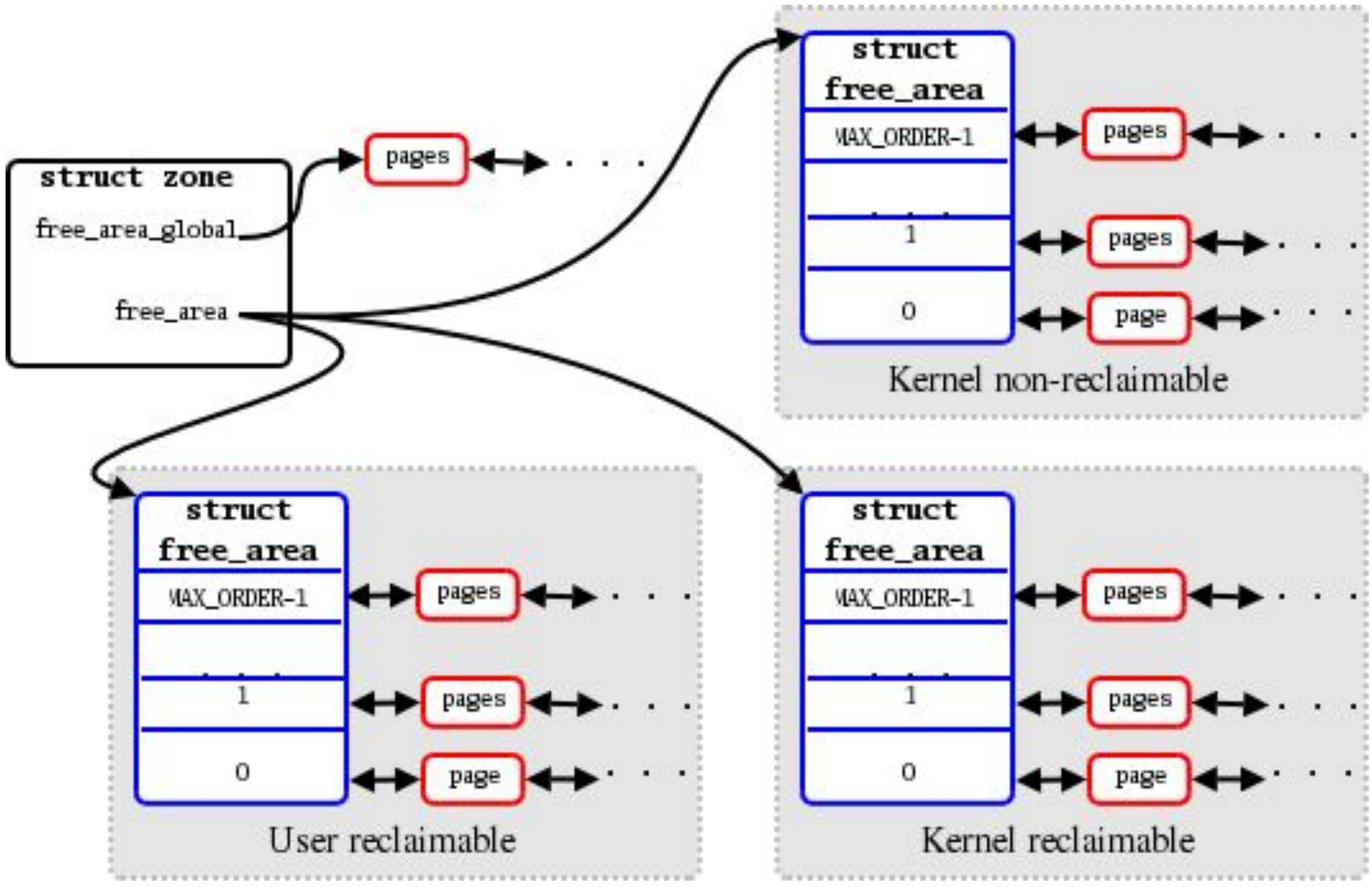
The Linux Buddy Allocator: Per-order Freelists



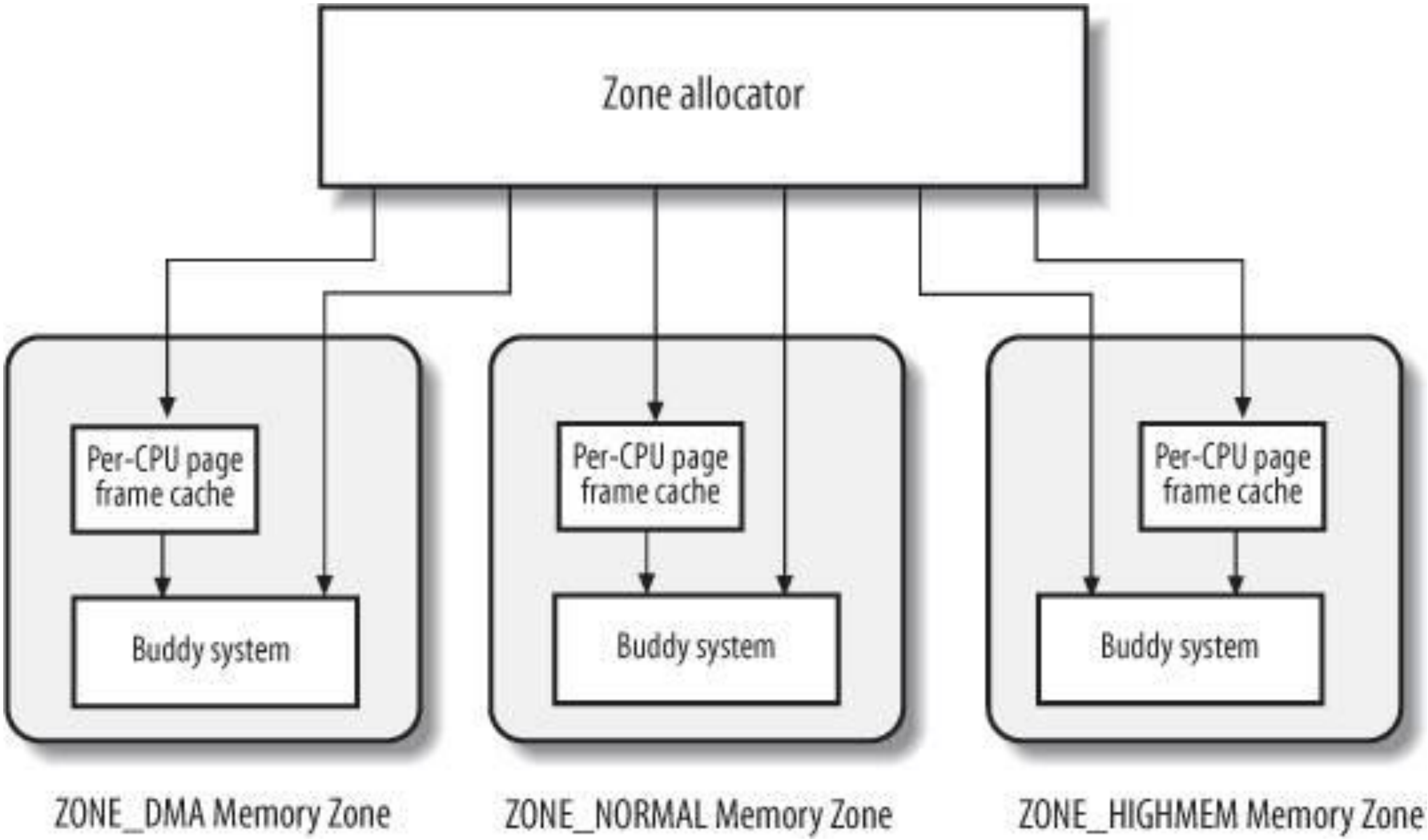
The Linux Buddy Allocator: Buddy Operations



The Linux Buddy Allocator: Per-zone Data Structures



The Linux Buddy Allocator: Per-zone Data Structures



The Linux Buddy Allocator: Per-CPU Frame Cache

The Linux Buddy Allocator: Interface

- Core API (with many wrappers):
  - `struct page * alloc_pages(gfp_t gfp_mask, unsigned int order);`
  - `void __free_pages(struct page *page, unsigned int order);`

- GFP (Get Free Page) flags (**`linux/gfp.h`**):

```
#define ___GFP_DMA          0x01u
#define ___GFP_HIGHMEM      0x02u
#define ___GFP_DMA32        0x04u
#define ___GFP_MOVABLE      0x08u
#define ___GFP_HIGH         0x20u
#define ___GFP_NOFAIL       0x800u
#define ___GFP_ZERO         0x8000u
#define ___GFP_ATOMIC       0x80000u
```

From Linux to Lab1 in OpenLSD

- UMA (1 node)
- Single memory type (1 zone)
- Simple alloc-only boot-time allocator
- Simple buddy page frame allocator

The OpenLSD Boot Memory Allocator

- A simple page-granular buffer allocator
  - `boot_alloc(uint32_t n);`
- Maintains an index to the next free block
- Can’t free or reuse memory
- What happens when out of memory?
- Can’t hand over memory to other allocators
- Only used for very few long-lived data structures allocated early on
  - e.g., initial page directory

The OpenLSD Page Frame Allocator

- A simple buddy page frame allocator
  - `struct page_info *page_alloc(int flags);`
  - `void page_free(struct page_info *pp);`
- No built-in sanity checks
- Bonus: Linux-like sanity checks (details later)
- Can allocate one page at the time
- How many freelists of page descriptors?
  - `page_free_list[]` (kernel/mem/buddy.c)

The Linux Buddy Allocator: Fragmentation

- External fragmentation:
  - Inability to service allocations due to excessive number of small free blocks
  - How addressed:
    - free coalescing
    - `vmalloc` allocator for large allocations
- Internal fragmentation:
  - Space wastage due to large block assigned to smaller allocation(s)
  - How addressed:
    - Slab allocators

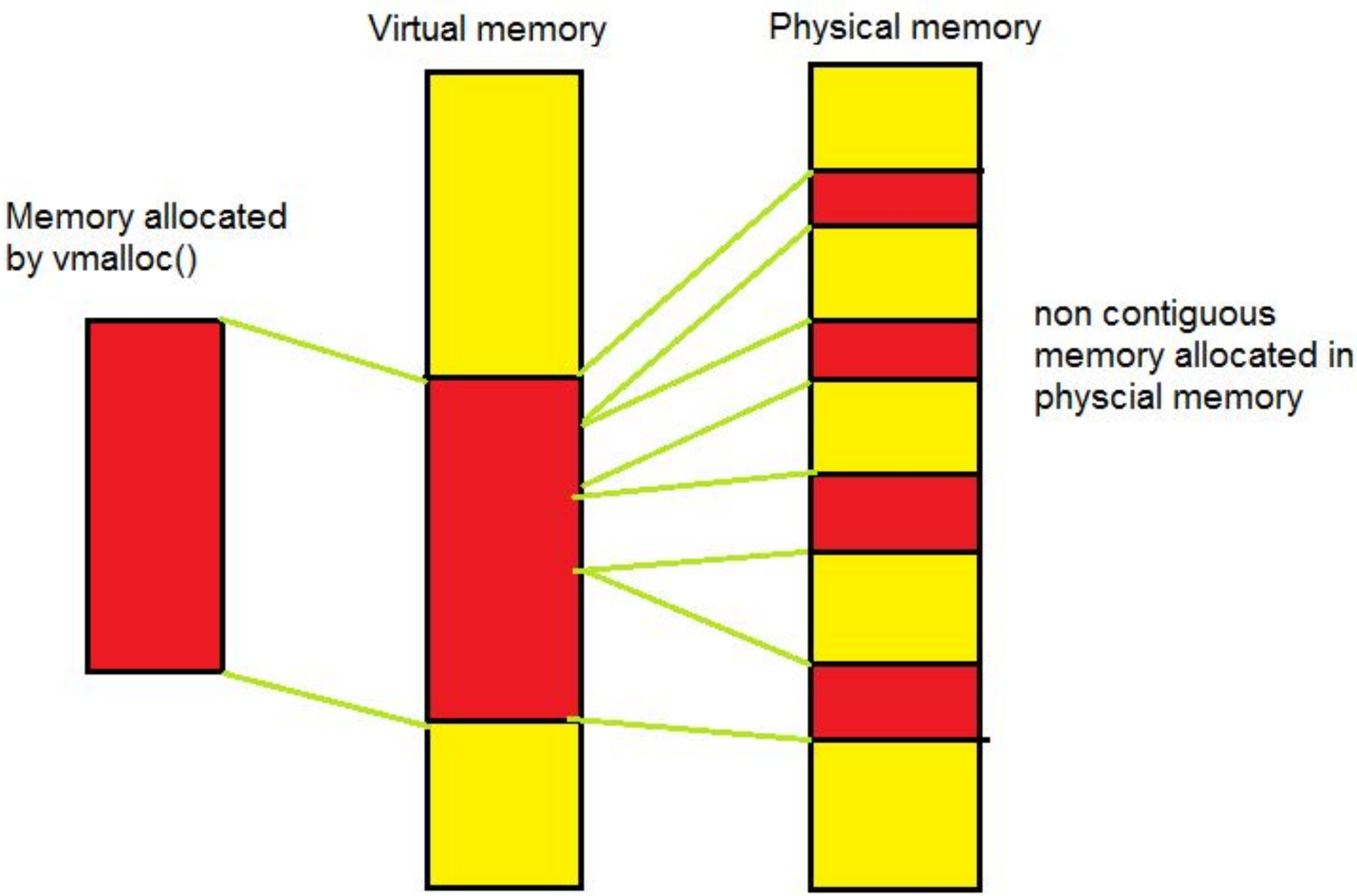
The Linux `vmalloc` allocator

- Allocator for large allocations to deal with external fragmentation:
  - `void *vmalloc(unsigned long size);`
  - `void vfree(const void *addr);`

- Basic behavior:
  - First-fit, similar in spirit to memblock

- To satisfy allocation of size N:
  - Allocate N page frames
  - Map page frames in virtually continuous buffer

The Linux `vmalloc` allocator



The Linux `vmalloc` allocator

mm/vmalloc.c:

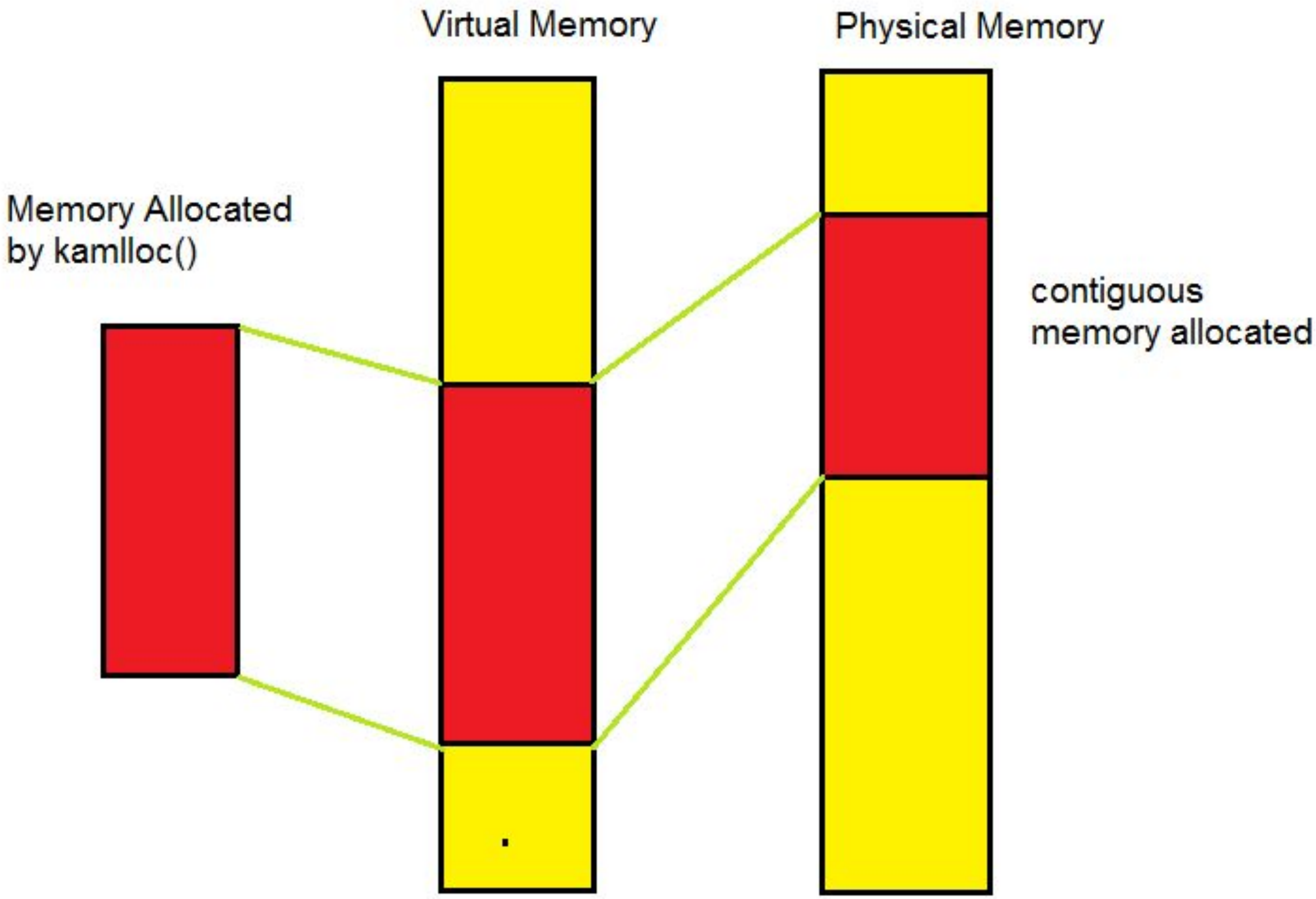
```
static void *__vmalloc_area_node(struct vm_struct *area, gfp_t gfp,
                pgprot_t prot, int node)
{
    for (i = 0; i < area->nr_pages; i++) {
        struct page *page;
        page = alloc_pages_node(node, alloc_mask|highmem_mask, 0);
        area->pages[i] = page;
    }

    if (map_kernel_range(area->addr, get_vm_area_size(area),
                    prot, pages)<0)
        goto fail;
    return area->addr;
}
```

The Linux slab allocator

- Allocator for small allocations to deal with internal fragmentation (high-level interface):
  - `void *kmalloc(size_t size, gfp_t flags);`
  - `void kfree(const void *addr);`

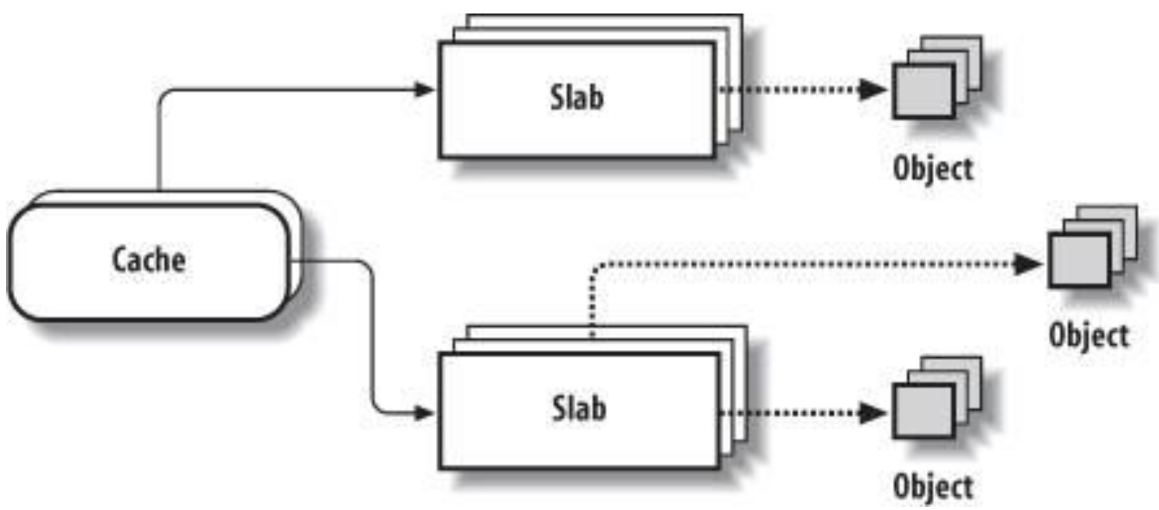
- Unlike `vmalloc`:
  - Many implementations:
    - SLAB, SLOB, SLUB, ...
  - Memory is allocated from per-object-size *caches*:
    - Typical range [8B; 8KB]
  - Allocated memory is physically contiguous

The Linux slab allocator

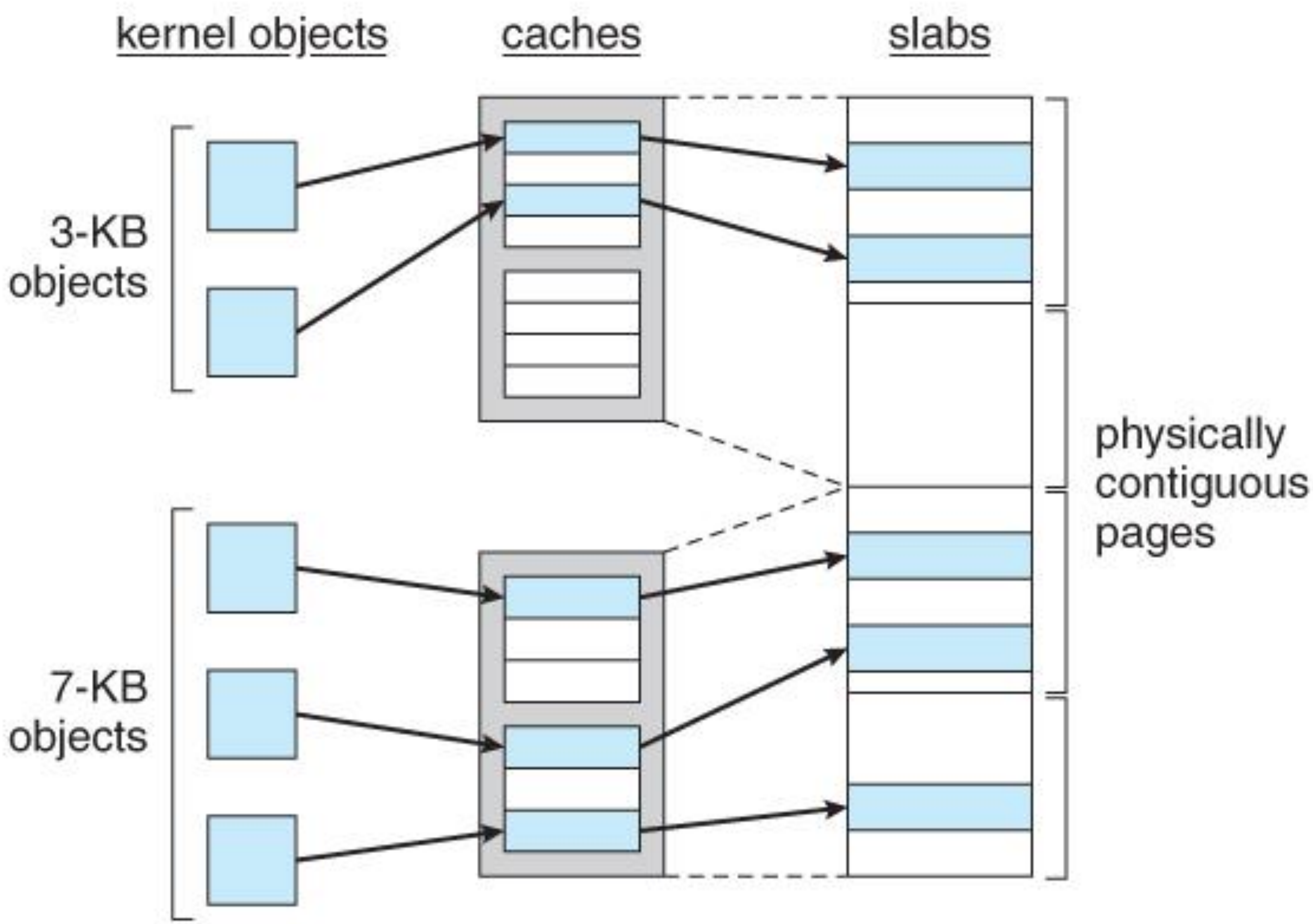


The Linux slab allocator

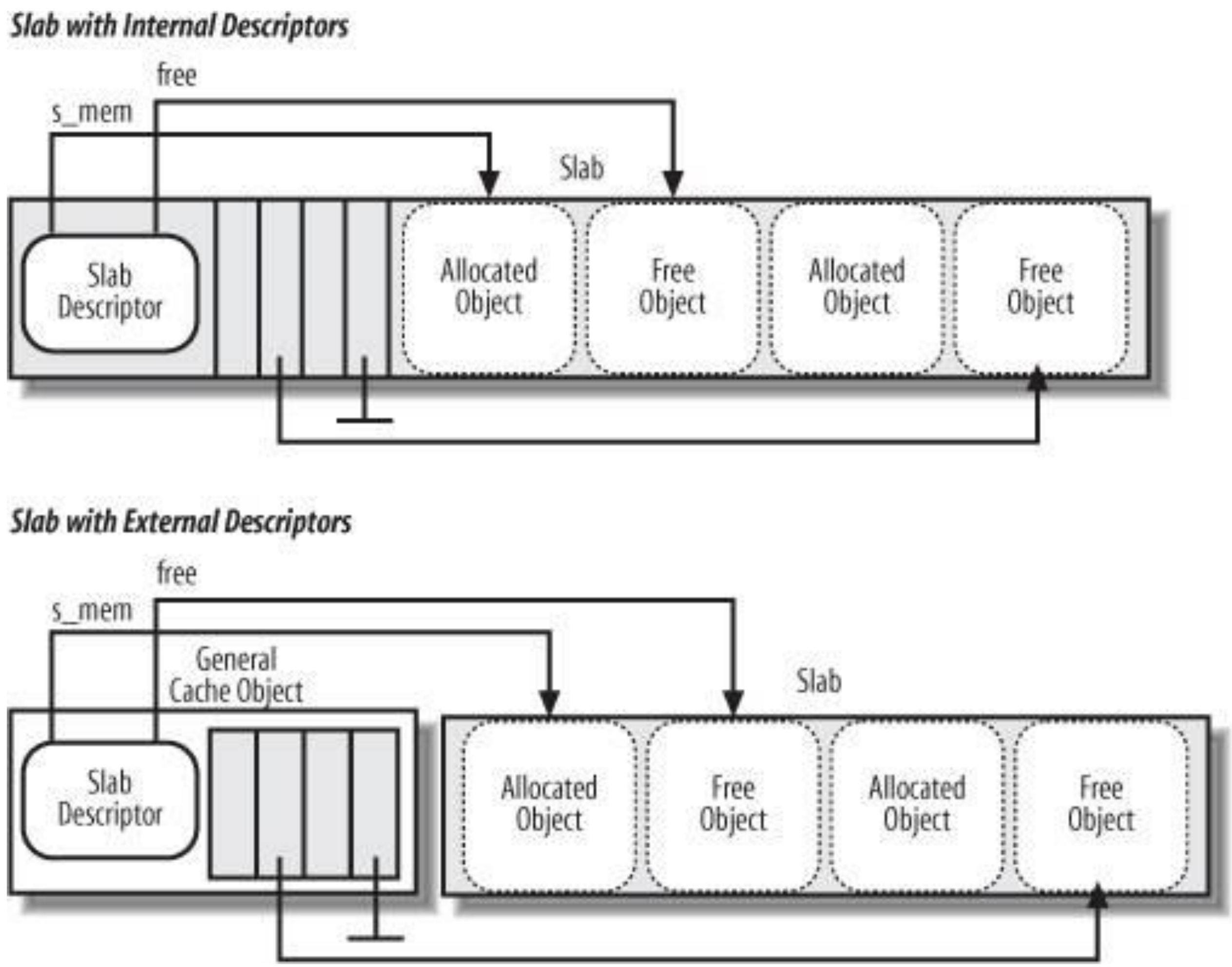
- Cache interface:
  - `struct kmem_cache *kmem_cache_create(…);`
  - `void kmem_cache_destroy(struct kmem_cache*);`
  - `void *kmem_cache_alloc(struct kmem_cache*,…);`
  - `void kmem_cache_free(struct kmem_cache*,…);`

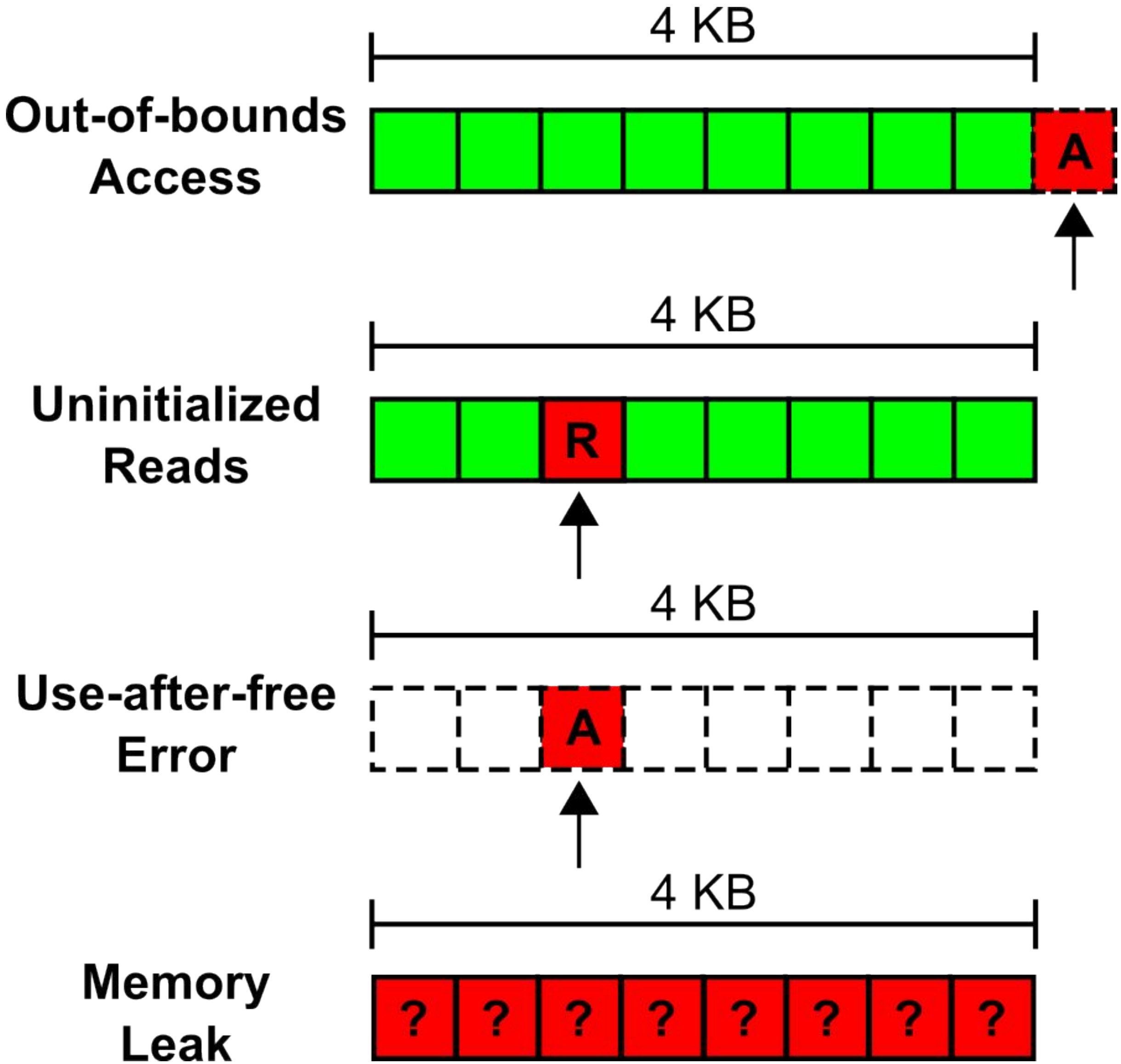


The Linux slab allocator



The Linux slab allocator





Frame Allocation: Memory Errors

Frame Allocation: Sanity Checks

- CONFIG_DEBUG_PAGEALLOC
- 1 - Out-of-bounds detection
  - **Page guarding** (`page_is_guard`)

  - **Properties**:
    - Immediately detects each out-of-bounds access
    - Only accesses to [-4KB, +4KB]

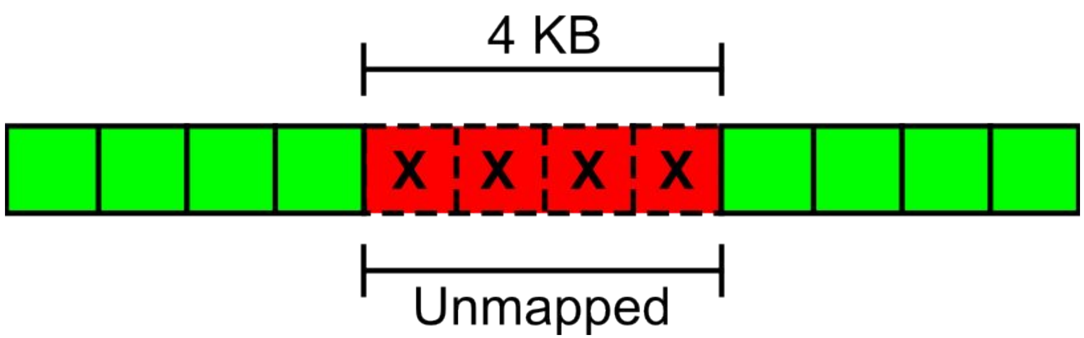
Frame Allocation: Sanity Checks

- CONFIG_DEBUG_PAGEALLOC
- 1 - Out-of-bounds detection
  - **Page canaries** (unimplemented alternative)

  - **Properties**:
    - Lazily detects out-of-bounds writes by mismatch
    - Only writes to [-sizeof(canary), +sizeof(canary)]

Frame Allocation: Sanity Checks

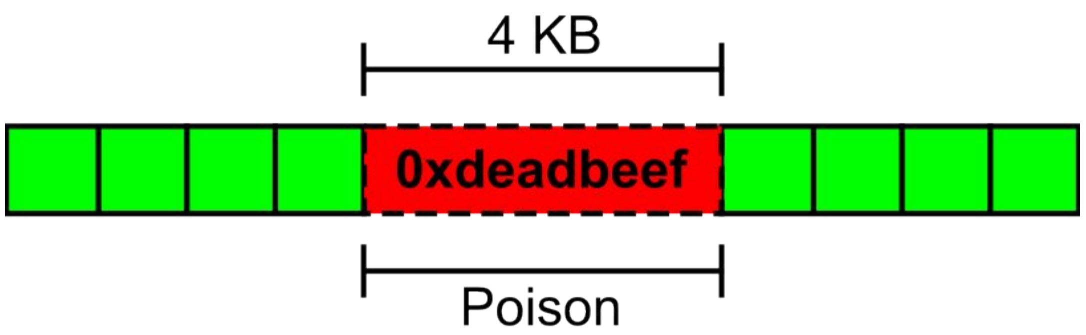
- CONFIG_DEBUG_PAGEALLOC
- 2 - Use-after-free detection
    - **Page remapping** (`kernel_map_pages`)



    - **Properties**:
        - Detects each use-after-free when unallocated
        - What happens after memory reuse?

Frame Allocation: Sanity Checks

- CONFIG_DEBUG_PAGEALLOC
- 2 - Use-after-free detection
  - **Page poisoning** (`kernel_poison_pages`)



  - **Properties**:
    - Lazily detects each write-after-free by mismatch
    - Can’t handle memory reuse

Frame Allocation: Sanity Checks

- A number of object-level *sanitizers*
- `kmemcheck` (uninitialized reads, now off-tree)
  - Memory protection-based
  - Traps at every read to check if data initialized
- `kmemleak` (memory leaks)
  - Periodic conservative garbage collection
  - Reports objects not pointed by any likely pointers
- `kasan` (out-of-bounds and use-after-free)
  - Compiler-based
  - Combines canary and poisoning ideas
  - Checks each access to check if target is live object
- Others: `UBSAN`, `KCSAN`

Frame Allocation: Sanity Checks

- Built-in integrity checks in frame allocator
- `check_new_page`
  - Semantic checks on page descriptor
  - Memory error checks?
- `free_pages` (`virt_addr_valid`)
  - Invalid free detection
  - Limitations?

References

[1] Gorman, Mel. *Understanding the Linux Virtual Memory Manager*, 2007.

[2] “Development tools for the kernel,” n.d. https://www.kernel.org/doc/html/latest/dev-tools.

[3] “The ‘too Small to Fail’ memory-Allocation Rule,” n.d. https://lwn.net/Articles/627419.

Managing Physical Memory

Backup Slides

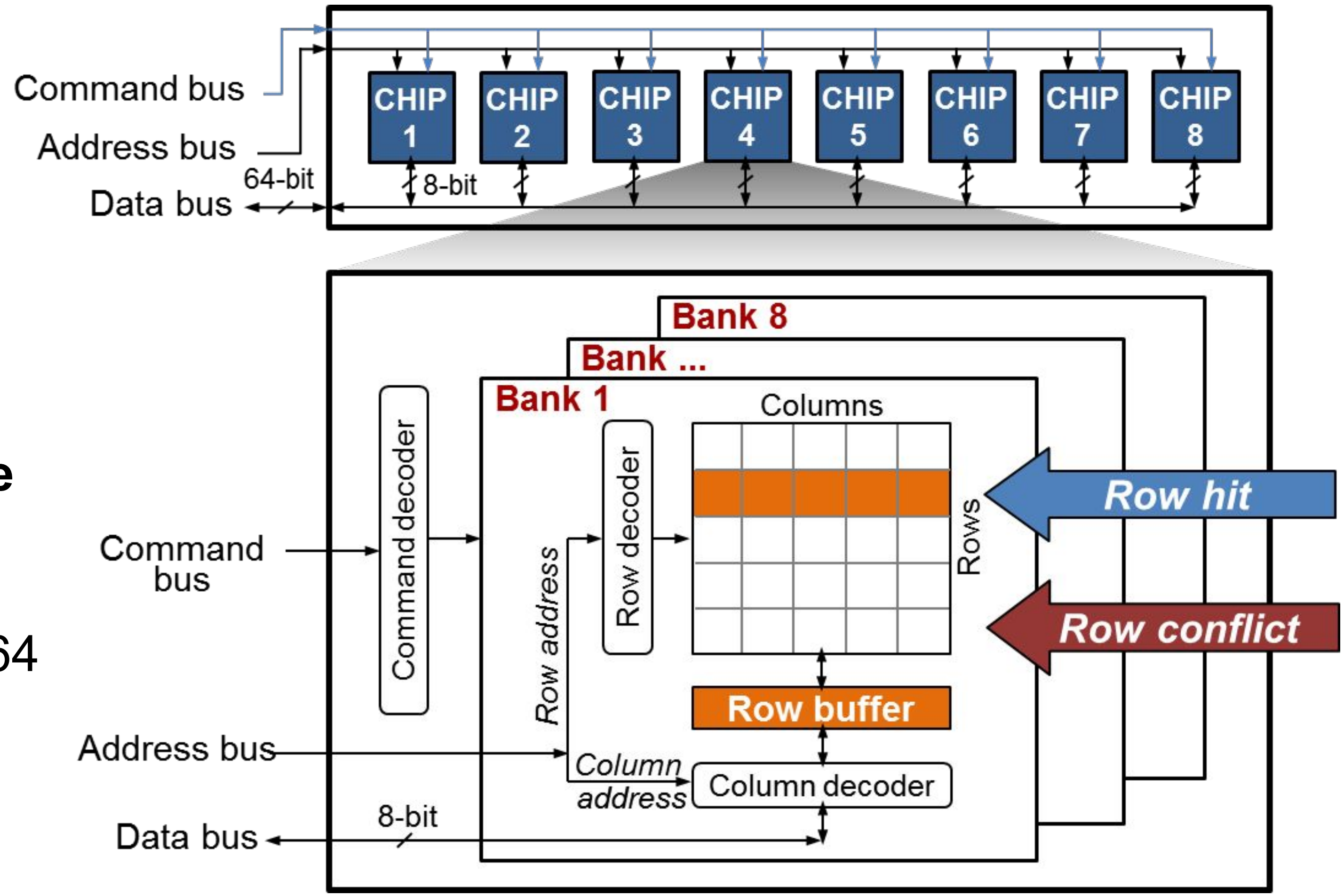
What is "Physical Memory" Anyway?

DRAM Rank

DRAM Chip

BANK example (R x c x bits):

4096 x 1024 x 64



What is “Physical Memory” Anyway?

```
mov [0x7f00], 1 # real mode
```

Q: What gets physically written to memory?

A: Hard to say in practice:

- Address scrambling
  - Complex *paddr* to DRAM <C, R, B> mapping
  - Distributes load to avoid “bank trashing”
- Data scrambling
  - Logical 1 != DRAM 1
  - Avoids data burst (resonance) on the data bus
  - Mitigates (some) cold-boot attacks

What is Wrong With DRAM?

Side channels:

- Bank collisions can be detected via timing
- Allows attackers to detect sensitive events

Hardware faults:

- Capacitors can drop charge too quickly
- Bit flips due to cross-cell/-row interference
  - **Rowhammer**: originally reliability problem
  - **Flip Feng Shui**: bit flips controllable and exploitable
  - ECC *might* make this better

What is Wrong With DRAM?

You will learn more about this at “Hardware Security”...