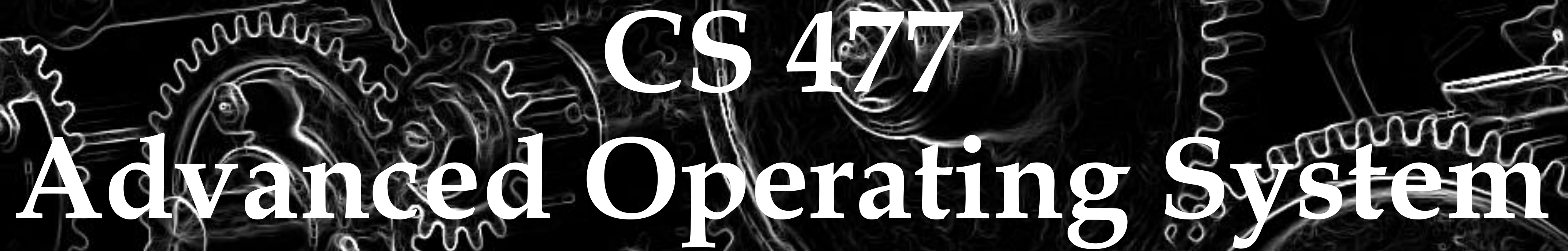



## Lecture 09: Physical Memory Management

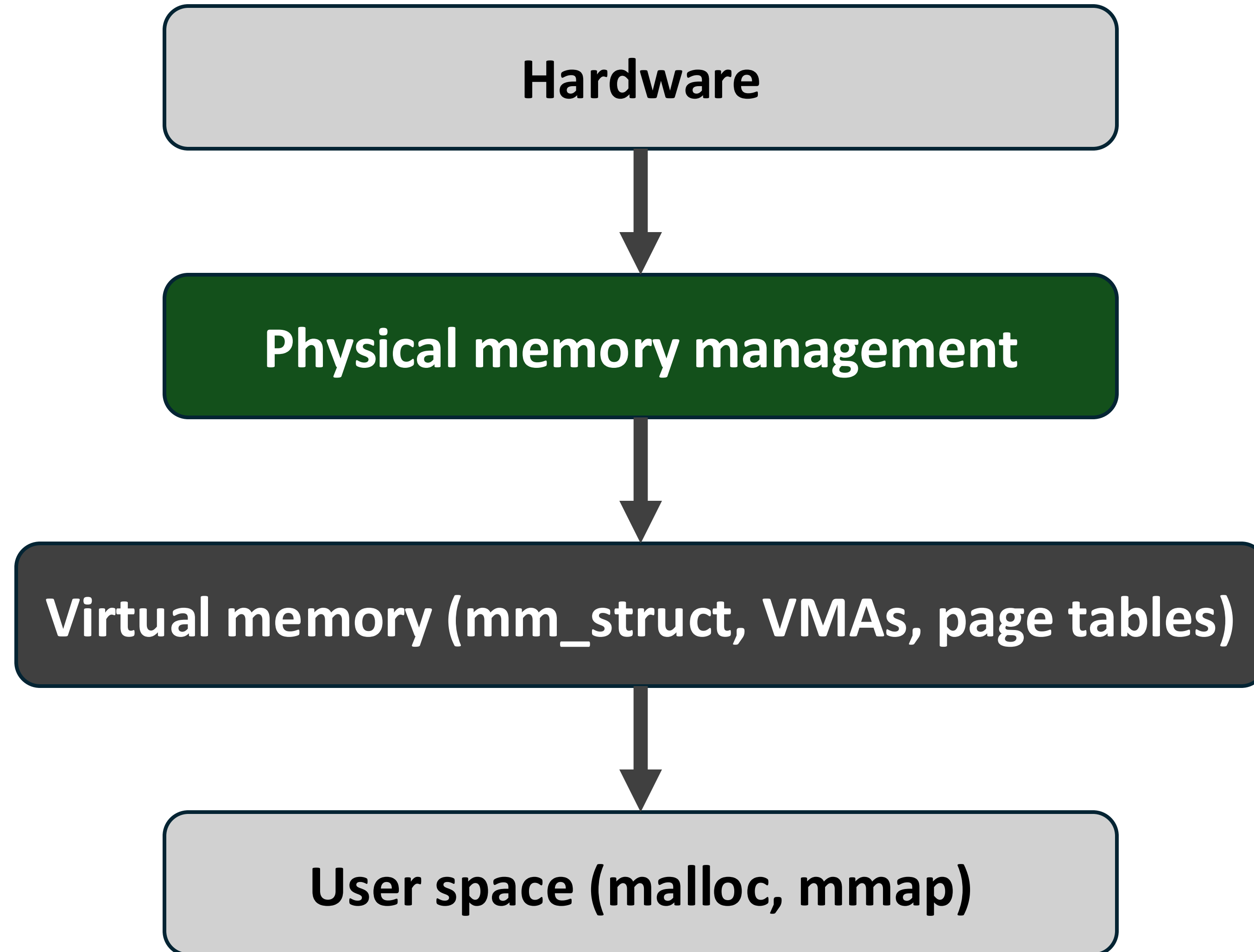
EPFL

# Logistics

- Lab 2 deadline: Nov 7th
- Late days policy:
  - Still intact
  - For group lab (lab 4)
    - Can still use it for the lab 4
    - Students will be penalized per person if they exceed by 7 days

EPFL

# Memory management



# Today's agenda

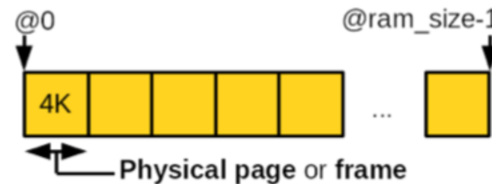
- **Organization** → track memory (zones, NUMA, struct page)
- **Allocation** → hand out physical memory region (buddy, slab)
- **Reclamation** → reclaim memory region (LRU, swap)
- **Performance** → making it fast (TLB, hugepages)

→ **What happens *physically* when you call malloc()**

**Q. How does the kernel keep track of 8GB of RAM?**

# Pages

- Memory is divided into **physical pages** or **frames**




- The page is the basic management unit in the kernel
- Page size is *machine dependent*
  - Determined by MMU (memory management unit)
  - **4KB** in general, some are **2MB and 1 GB**
    - getconf PAGESIZE

EPFL

# Pages representation

- **`struct page`** represents each **physical page**

```
struct page {
    unsigned long flags;     /* page status (permission, dirty, etc.) */
    unsigned counters;       /* usage count */
    struct address_space *mapping;
                             /* address space mapping */
    pgoff_t index;           /* offset within the mapping */
    struct list_head lru;    /* LRU list buffer cache */
    void *virtual;           /* kernel virtual address when kmapped */
}
```

- One struct page per **physical page frame**
    - Like a page descriptor
- 64 bytes in size
    - # pages for 8 GB DRAM: 8 GB / 4KB = 2M pages * 64 bytes = 128 MB overhead (~1.5%)

EPFL

# What does a page track?

- Keeps track of the page owner
  - User space process, kernel statically/dynamically allocated data, page cache etc.
- Reference count: can we *free* this page?
- Keeps track of the page state
  - `PG_locked`, `PG_dirty`, `PG_referenced`, `PG_uptodate`, `PG_slab`, etc.
- Ties the **virtual memory system**, **page cache**, **allocator**, and **reclamation** subsystems

**Q. Does the hardware treat each physical page the same?**

# Zones: Segregate different physical memory regions

**Problem:** Not all physical memory is equal

**Historical context:**

- Legacy ISA DMA devices: can only address the first 16 MB (ZONE_DMA)
- x86_32 with 4GB+ RAM: kernel can't directly map it all (ZONE_HIGHMEM)

**Solution:**

- Physical memory is partitioned into **zones** having the same constraints
  - Zone layout is architecture- and machine-dependent
- Page allocators consider the constraints when allocating pages

# Zones: Types with modern take

| Name | Description |
| --- | --- |
| ZONE_DMA | First 16 MB (legacy DMA) |
| ZONE_DMA32 | First 4GB (32-bit DMA devices) |
| ZONE_NORMAL | Diretly mapped, most important<br>Pages always mapped to this region |
| ZONE_HIGHMEM | x86_32 only (>896 MB)<br>Pages should be mapped prior to access |

# Zones representation

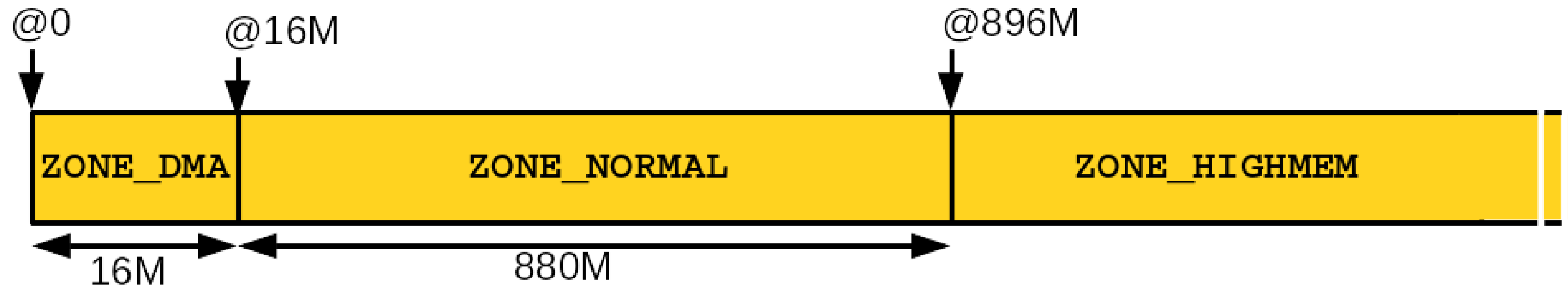
## **`struct zone`** data structure manages each zone

```
struct zone {
    const char *name;                /* Name of this zone */
    unsigned long zone_start_pfn;    /* starting page frame number of the zone */
    unsigned long watermark[NR_WMARK];
                        /* minimum, low, and high watermarks
                         * for per-zone memory allocation */
    spinlock_t lock;    /* protects against concurrent accesses */
    struct free_area free_area[MAX_ORDER];
                        /* list of free pages of different sizes */
};
```

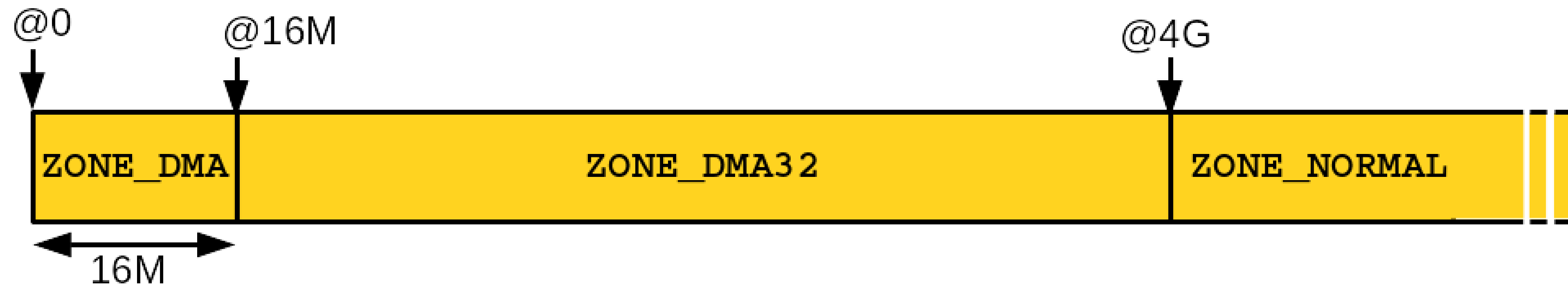
- Manages one **contiguous region of physical memory** for page allocator
  - Tracks how much free memory exists in that region
  - Decides when to reclaim pages (based on watermarks)
  - Ensures safe concurrent access within the zone with a spinlock

# Zones are mostly architecture dependent

## x86_32 zones layout




## x86_64 zones layout




x86_64's large virtual address space eliminates HIGHMEM problem

EPFL

# Zones used for other physical memory management

| Name | Description |
|---|---|
| ZONE_DMA | First 16 MB (legacy DMA) |
| ZONE_DMA32 | First 4GB (32-bit DMA devices) |
| ZONE_NORMAL | Diretly mapped, most important<br>Pages always mapped to this region |
| ZONE_HIGHMEM | x86_32 only (>896 MB)<br>Pages should be mapped prior to access |
| **ZONE_MOVABLE** | Logical memory zone; tracks which pages can be migrated to maintain physical contiguity |
| **ZONE_DEVICE** | Device-managed physical memory (owned by device like GPU) & CPU has no control over it |

EPFL

# Zones demo

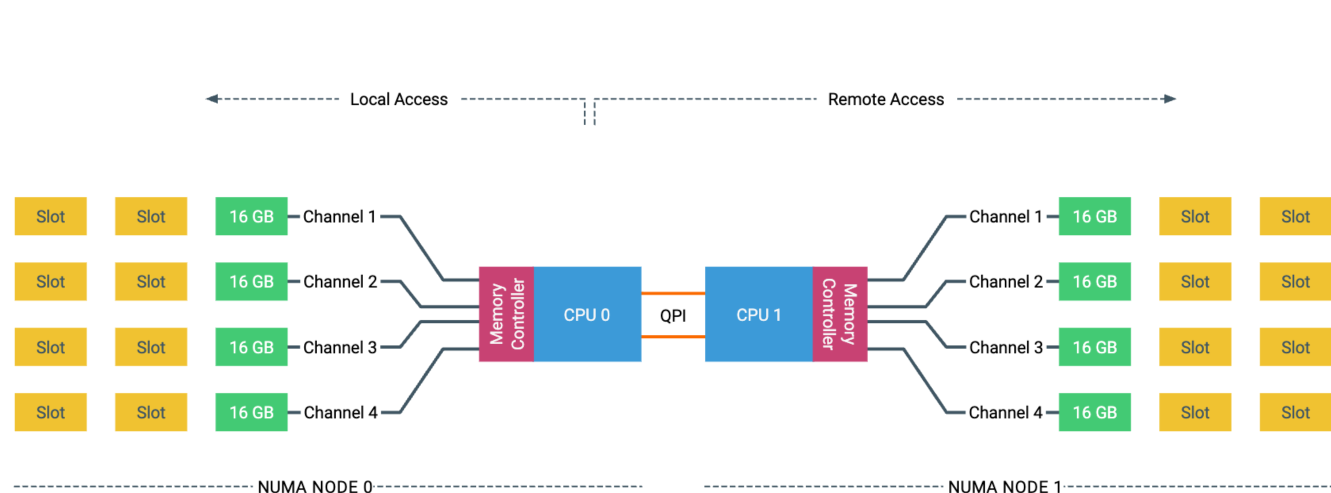
**`cat /proc/zoneinfo`**

- Node 0 corresponds to NUMA node 0
- Keeps track of:
  - Free pages currently
  - Watermark threshold controlling when the kernel triggers background reclaim (kswapd)
  - Locked/mlocked pages that cannot be swapped out

**Q. What is NUMA, and why do we care about it?**

```
Node 0, zone      DMA
  per-node stats
      nr_inactive_anon 3733
      nr_active_anon 538221
      nr_inactive_file 4907601
      nr_active_file 424993

Node 0, zone    DMA32
  pages free     588050
        boost    0
        min      312
        low      899
Node 0, zone   Normal
  pages free     24200150
        boost    77014
        min      93595
        low      124800
        high     156005
Node 0, zone  Movable
  pages free     0
        boost    0
        min      32
        low      32
        high     32
        promo    32
```

# Modern machines have non-uniform-memory access



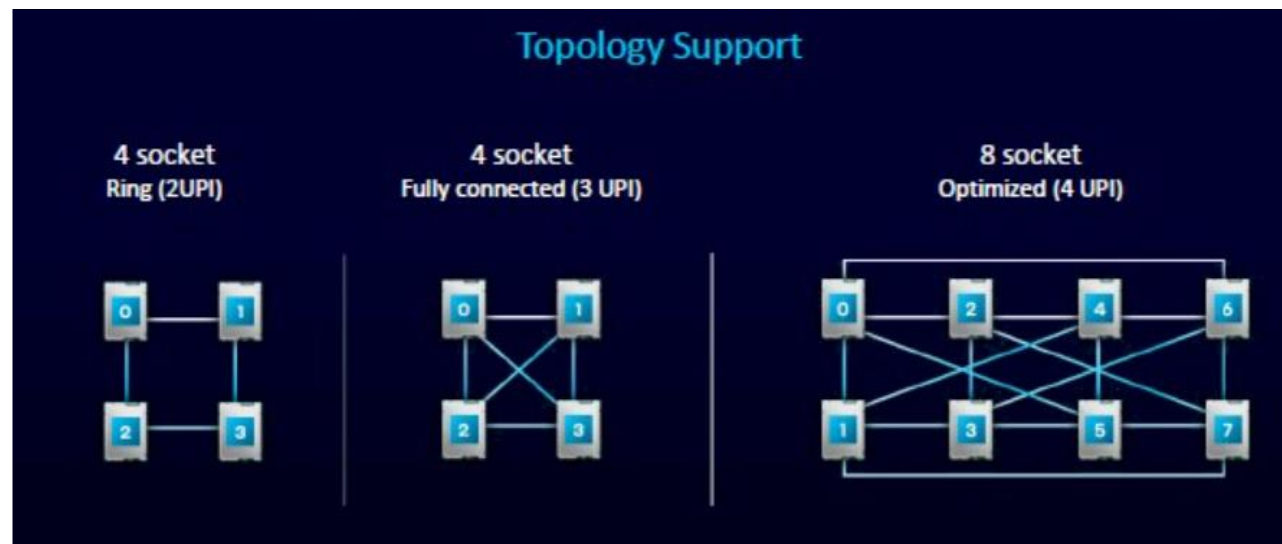

Min=15.3ns Median=82.5ns Max=87.0ns

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CPU 1 | | 18 | 18 | 16 | 17 | 18 | 85 | 85 | 86 | 84 | 85 | 85 |
| CPU 2 | 18 | | 17 | 16 | 16 | 16 | 83 | 84 | 84 | 83 | 84 | 83 |
| CPU 3 | 18 | 17 | | 16 | 18 | 17 | 85 | 86 | 87 | 85 | 86 | 86 |
| CPU 4 | 16 | 16 | 16 | | 15 | 15 | 83 | 84 | 84 | 82 | 83 | 83 |
| CPU 5 | 17 | 16 | 18 | 15 | | 16 | 83 | 84 | 84 | 83 | 84 | 84 |
| CPU 6 | 18 | 16 | 17 | 15 | 16 | | 84 | 85 | 85 | 83 | 84 | 84 |
| CPU 7 | 85 | 83 | 85 | 83 | 83 | 84 | | 18 | 20 | 19 | 19 | 19 |
| CPU 8 | 85 | 84 | 86 | 84 | 84 | 85 | 18 | | 21 | 18 | 20 | 18 |
| CPU 9 | 86 | 84 | 87 | 84 | 84 | 85 | 20 | 21 | | 20 | 21 | 19 |
| CPU 10 | 84 | 83 | 85 | 82 | 83 | 83 | 19 | 18 | 20 | | 18 | 17 |
| CPU 11 | 85 | 84 | 86 | 83 | 84 | 84 | 19 | 20 | 21 | 18 | | 18 |
| CPU 12 | 85 | 83 | 86 | 83 | 84 | 84 | 19 | 18 | 19 | 17 | 18 | |

- Shared bus bottlenecked in a **uniform-memory access (UMA)** architecture
- Each **CPU socket (node)** has its **own local memory controller and DRAM banks**
  - CPUs are connected via high-speed interconnects (e.g., UPI, QPI, AMD Infinity Fabric).
  - CPU can still access remote memory, but with higher latency
  - Local access: ~100 ns; remote access: ~200-300ns (2-3x slower)
- Check: https://github.com/nviennot/core-to-core-latency

https://frankdenneman.nl/2016/07/08/numa-deep-dive-part-2-system-architecture/

# Modern machines have varying NUMA topology




- Machines have support up to 8 sockets (nodes)
  - Mostly Intel
- Sparc provided up to 32 sockets (now discontinued)
- Also, known as **scale-up architecture**

EPFL

# NUMA allocation policies

- **Default:** Allocate on local node (first-touch policy)
- **Interleave:** Spread allocations across nodes (bandwidth)
- **Bind:** Force to specific nodes
- **AutoNUMA:** Kernel automatically migrates pages to local nodes

```
$ numactl --hardware
ailable: 8 nodes (0-7)
de 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
de 0 size: 95297 MB
de 0 free: 91857 MB
de 1 cpus: 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
de 1 size: 96715 MB
de 1 free: 92039 MB
de 2 cpus: 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
de 2 size: 96758 MB
de 2 free: 95108 MB
de 3 cpus: 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 1
de 3 size: 96758 MB
de 3 free: 93514 MB
de 4 cpus: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 1
de 4 size: 96758 MB
de 4 free: 93666 MB
de 5 cpus: 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 1
de 5 size: 96758 MB
de 5 free: 73381 MB
de 6 cpus: 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 1
de 6 size: 96758 MB
de 6 free: 94430 MB
de 7 cpus: 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 2
de 7 size: 96745 MB
de 7 free: 94603 MB
de distances:
de   0   1   2   3   4   5   6   7
0:  10  21  21  31  21  31  31  31
1:  21  10  31  21  31  21  31  31
2:  21  31  10  21  31  31  21  31
3:  31  21  21  10  31  31  31  21
4:  21  31  31  31  10  21  31  21
5:  31  21  31  31  21  10  21  31
6:  31  31  21  31  31  21  10  21
7:  31  31  31  21  21  31  21  10
```

EPFL

# NUMA nodes in Linux

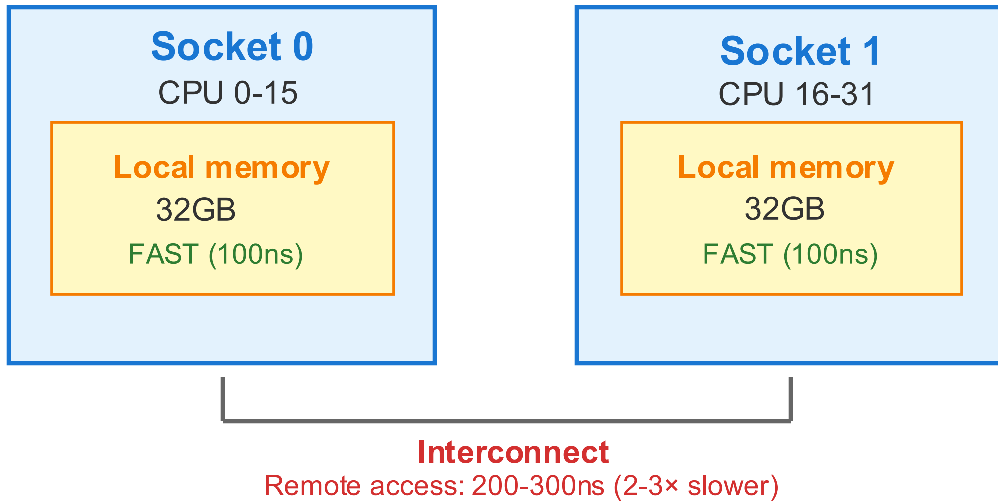
- Each node represented by **`pg_data_t`** structure
  - Contains its own zones

```
typedef struct pglist_data {
        /*
        |* node_zones contains just the zones for THIS node. Not all of the
        |* zones may be populated, but it is the full list. It is referenced by
        |* this node's node_zonelists as well as other node's node_zonelists.
        |*/
        struct zone node_zones[MAX_NR_ZONES];

        /*
        |* node_zonelists contains references to all zones in all nodes.
        |* Generally the first zones will be references to this node's
        |* node_zones.
        |*/
        struct zonelist node_zonelists[MAX_ZONELISTS];

        int nr_zones; /* number of populated zones in this node */
        unsigned long node_start_pfn;
        unsigned long node_present_pages; /* total number of physical pages */
        unsigned long node_spanned_pages; /* total size of physical page
                                          |    range, including holes */
        int node_id;
```

```
Node 0, zone      DMA
Node 0, zone    DMA32
Node 0, zone   Normal
Node 0, zone  Movable
Node 0, zone   Device
Node 1, zone      DMA
Node 1, zone    DMA32
Node 1, zone   Normal
Node 1, zone  Movable
Node 1, zone   Device
Node 2, zone      DMA
Node 2, zone    DMA32
Node 2, zone   Normal
Node 2, zone  Movable
Node 2, zone   Device
Node 3, zone      DMA
Node 3, zone    DMA32
Node 3, zone   Normal
Node 3, zone  Movable
Node 3, zone   Device
Node 4, zone      DMA
Node 4, zone    DMA32
Node 4, zone   Normal
Node 4, zone  Movable
Node 4, zone   Device
Node 5, zone      DMA
Node 5, zone    DMA32
Node 5, zone   Normal
Node 5, zone  Movable
Node 5, zone   Device
Node 6, zone      DMA
Node 6, zone    DMA32
Node 6, zone   Normal
Node 6, zone  Movable
Node 6, zone   Device
Node 7, zone      DMA
Node 7, zone    DMA32
Node 7, zone   Normal
Node 7, zone  Movable
Node 7, zone   Device
```

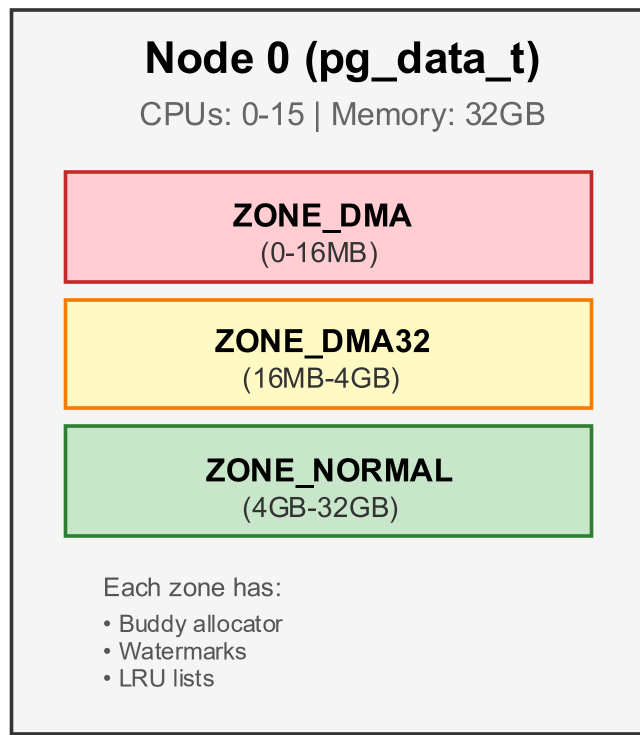
EPFL

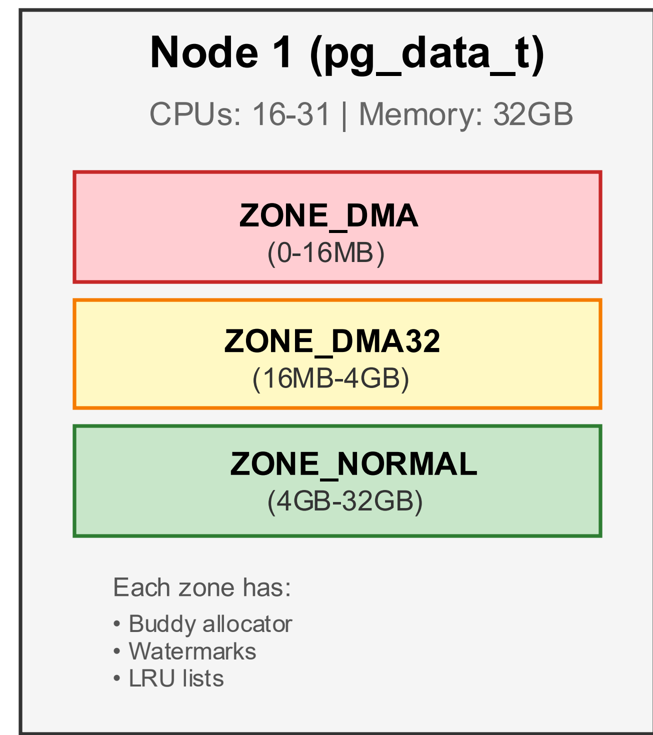
# Physical memory → Nodes



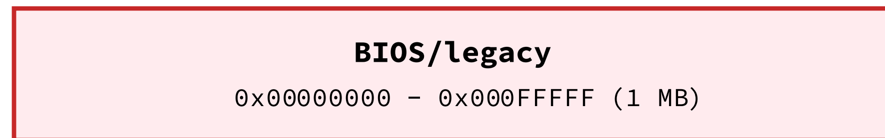

## Numa architecture

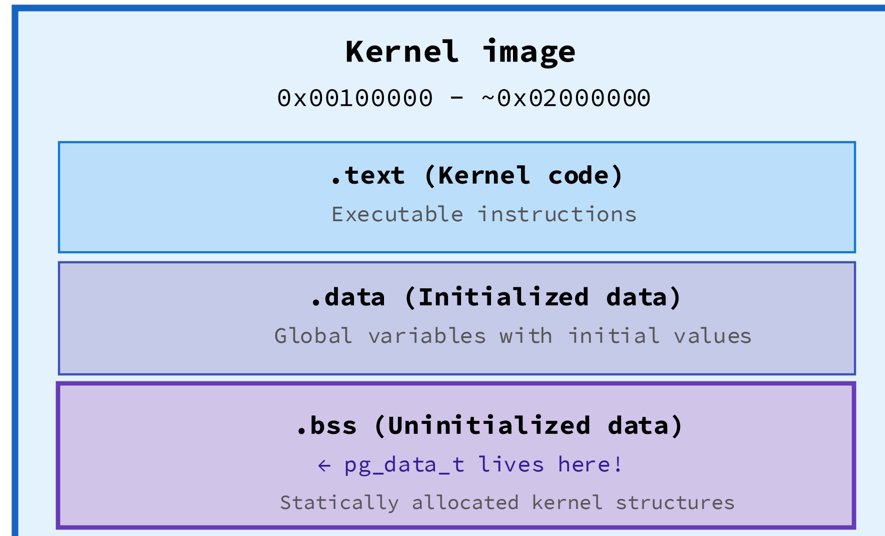
## Linux NUMA with zones

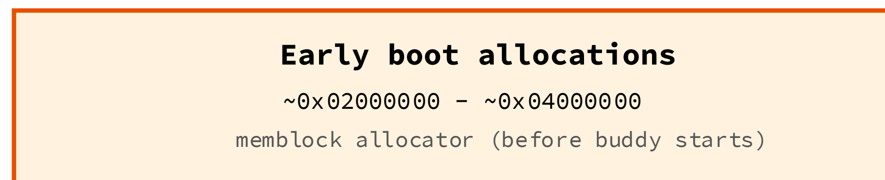


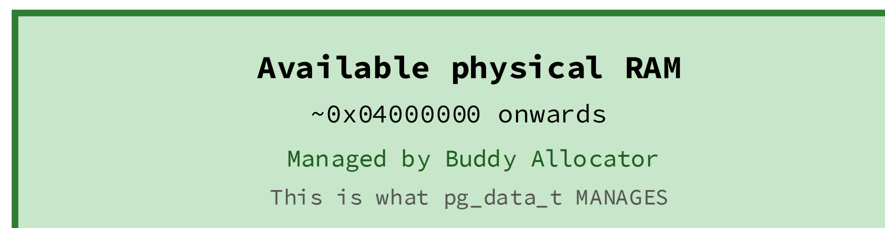



# Physical memory layout at boot time




node_data[0]
(pg_data_t for
NUMA node 0)

Page tables,
Initial stacks,
Early data structures

struct zone
free_area[11]
buddy lists
track FREE pages here

**`pg_data_t` is in the kernel, manages free memory**

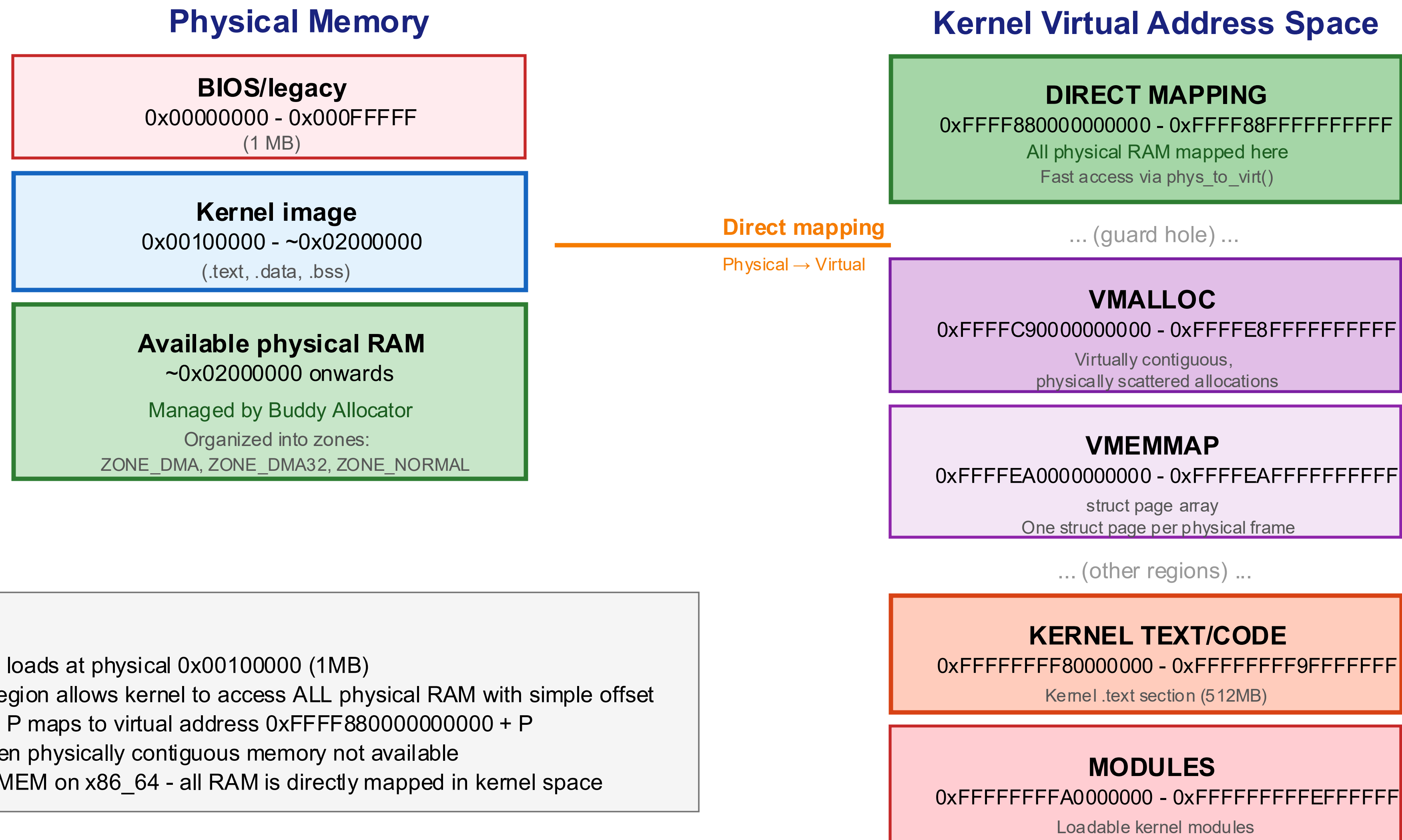
**1. Kernel image (`.bss` section):**
- Contains statically allocated pg_data_t structures
- Lives at ~1-2MB in physical memory
- node_data[0] = pg_data_t for NUMA node 0

**2. Early boot (memblock allocator):**
- Used before buddy allocator is ready
- Allocates page tables, stacks, early structures

**3. Buddy Allocator (free RAM):**
- Everything from ~4MB onwards becomes "free"
- pg_data_t->node_zones[]->free_area[] points to these free pages

EPFL

# Physical memory → virtual address space layout (boot)




**Key points:**

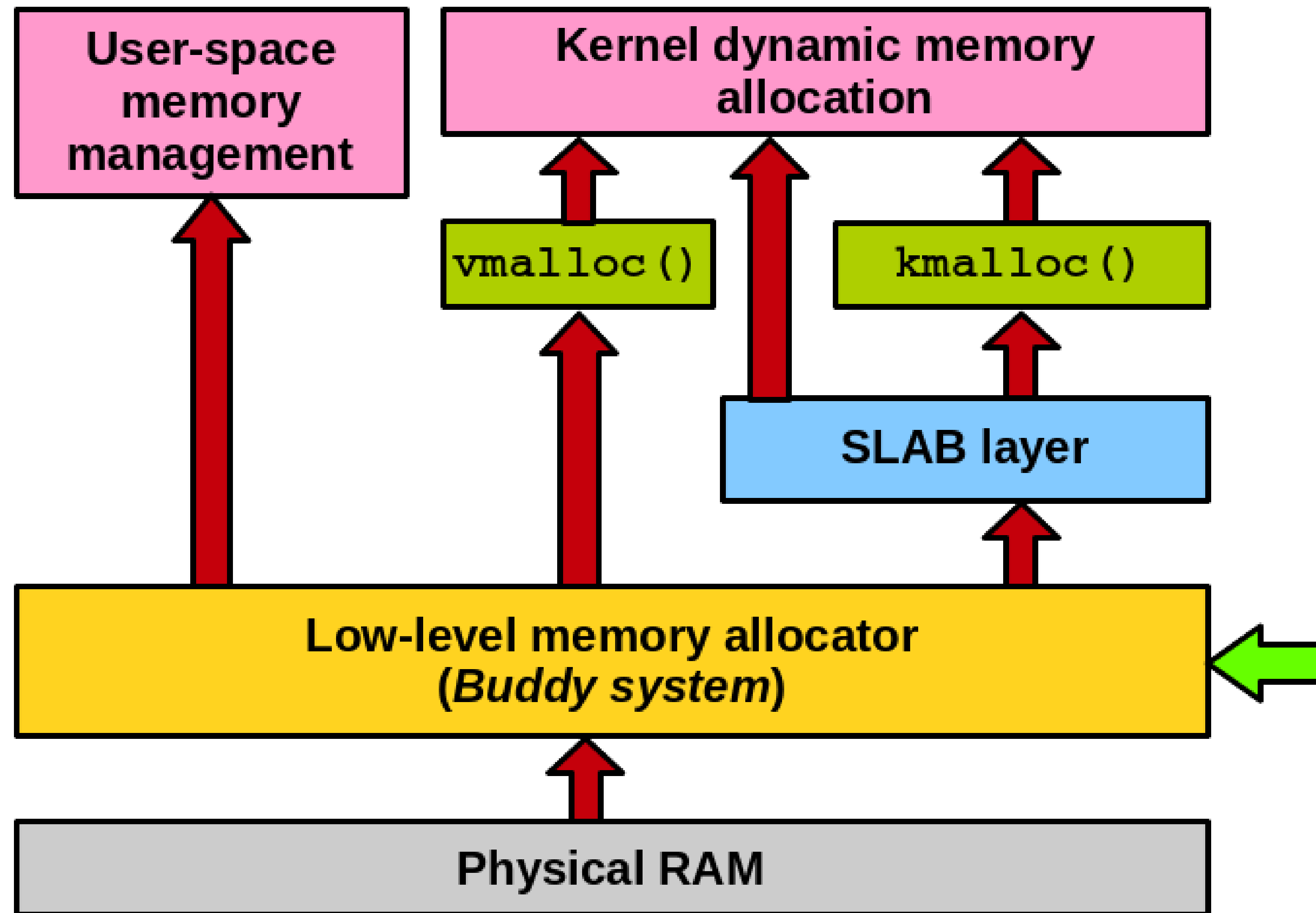
- Kernel boots and loads at physical 0x00100000 (1MB)
- Direct mapping region allows kernel to access ALL physical RAM with simple offset
- Physical address P maps to virtual address 0xFFFF880000000000 + P
- vmalloc used when physically contiguous memory not available
- No ZONE_HIGHMEM on x86_64 - all RAM is directly mapped in kernel space

# Now that we know how physical memory is organized, the next question is how we *allocate* from it.

**Q. What do we need to allocate pages that can service user space applications and kernel services?**

EPFL

# Hierarchy of memory allocators



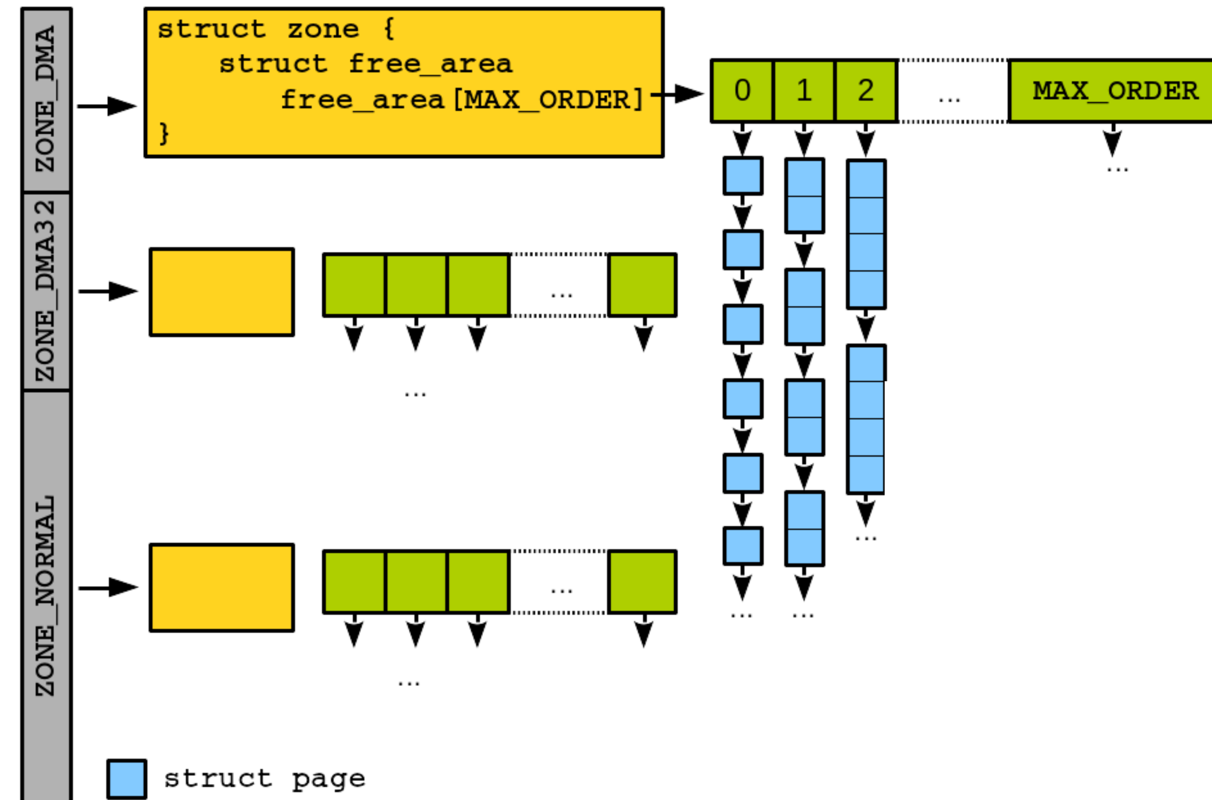

https://hammertux.github.io/slab-allocator

EPFL

# The low-level memory allocator

**Problem**: How to allocate contiguous physical pages efficiently while preventing fragmentation?

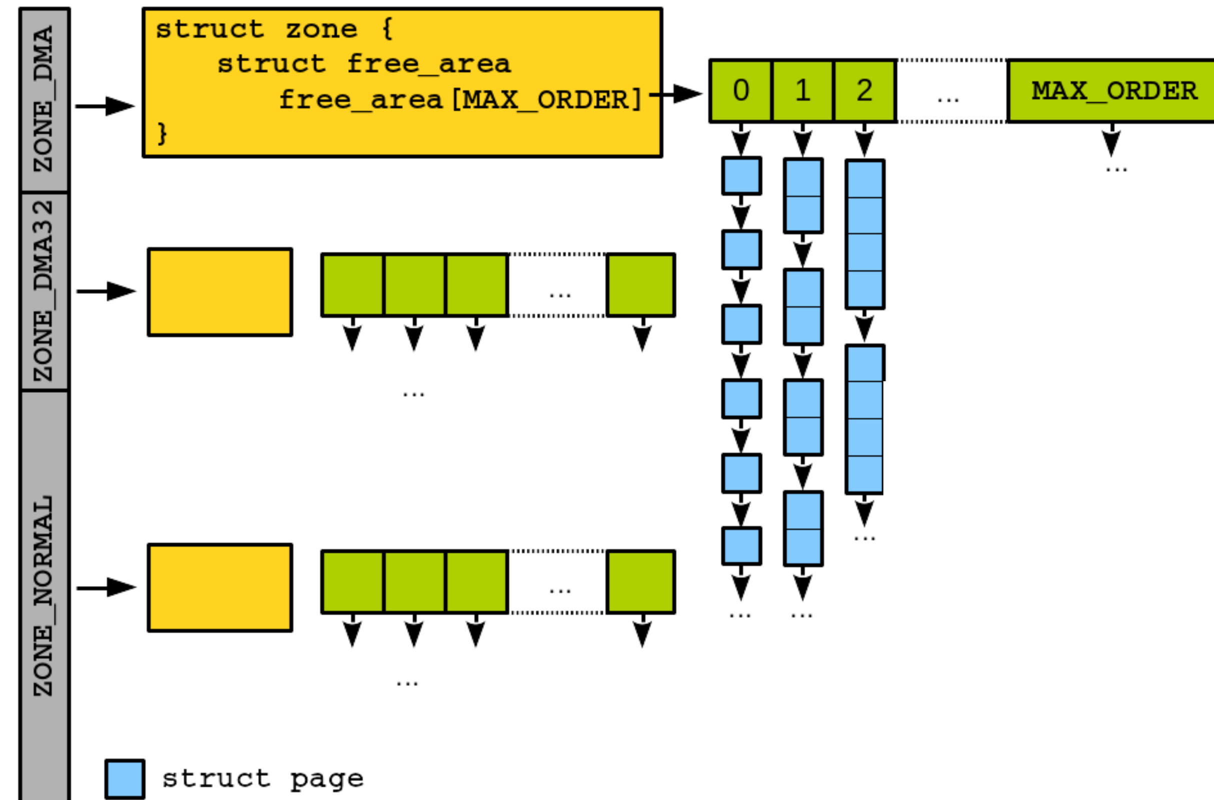
- **Buddy system**
  - Allocate memory at the page granularity
  - Maintains free lists for blocks of size $2^1, 2^2, 2^3 \ldots 2^{10}$
- Each zone has its own buddy allocator



EPFL

# The low-level memory allocator

- Allocation (order=1: want 2 pages):
  1. Check free_area[1] – any 2-page blocks? Yes → take it, done
  2. If no, check free_area[2] – any 4-page blocks?
     Yes → split into two 2-page blocks, take one, put other on free_area[1]
  3. If no, check free_area[3], and so on ...
- Deallocation
  1. Free 2-page blocks
  2. Check if its buddy is also free (buddy = adjacent 2-page block)
  3. If yes, merge into a 4-page block, and repeat



# Buddy allocator demo

```
└$ cat /proc/buddyinfo
Node 0, zone      DMA      0      0      0      0      0      0      0      0      1      1      2
Node 0, zone    DMA32      4      4      2      3      2      2      5      4      6      1    390
Node 0, zone   Normal   2565   1632   1375    116    658    524    162    116     73     49  17214
Node 1, zone   Normal  10967  13682   8490   8428   4244   2570   1383    916    511    261  21001
Node 2, zone   Normal  28025  23609  15617   9843   5423   2015    977    285     73     33  23247
Node 3, zone   Normal   2759   2690   1493    839    625    375    221    164     77     30  21890
Node 4, zone   Normal   1627  11514  11084   7021   4407   2473   1134    463    193    102  22926
Node 5, zone   Normal  21100  22192  19182  11854   7055   2566   1193    607    256    111  17620
Node 6, zone   Normal  15781  12215   7389   3142   1278    194    150    326    167     79  23071
Node 7, zone   Normal   1414   1805   2761   2701   1600    667    473    416    262    117  21737
```

EPFL

# Page allocation / deallocation

```c
/**
 * Allocate 2^{order} *physically* contiguous pages
 * Return the address of the first allocated `struct page`
 */
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
struct page *alloc_page(gfp_t gfp_mask);
```

```c
/**
 * Deallocate 2^{order} *physically* contiguous pages
 * Be careful to put the correct order otherwise corrupt the memory
 */
void __free_pages(struct page *page, unsigned int order);
void __free_page(struct page *page);
```

# Allocating zeroed pages

- By default, the page data is not removed
- May leak information through the page allocation
- For user space request, allocate a zeroed-out page:

```
unsigned long get_zeroed_page(gfp_t gfp_mask);
```

# The gfp_t flags: Decoding allocation behavior

**Problem:** Allocation is not one-size-fits-all

- Sleep waiting for memory?
- Trigger disk IO to free memory?
- Which zone to use?

**Solution:** gfp_t (get free pages) flags

Consists of three categories

# 1. gfp_t category: Zone modifiers (where to allocate)

- `__GFP_DMA // Must be in ZONE_DMA`
- `__GFP_DMA32 // ZONE_DMA32 or below`
- `__GFP_HIGHMEM // Can use ZONE_HIGHMEM (x86_32 only)`
- `// No flag = prefer ZONE_NORMAL`

EPFL

# 2. `gfp_t` category: Action allocators (how to allocate)

- `__GFP_WAIT // Can sleep waiting for memory`
- `__GFP_IO // Can start disk IO to reclaim`
- `__GFP_FS // Can make filesystem calls`
- `__GFP_HIGH // Can access emergency reserves`
- `__GFP_REPEAT // Retry if allocation fails`
- `__GFP_NOFAIL // MUST NOT FAIL (use carefully)`

# 3. gfp_t category: Type flags (common combinations)

- GFP_KERNEL = __GFP_WAIT | __GFP_IO | __GFP_FS
  // Process context, can sleep, can do IO
  // DEFAULT for most kernel allocations
- GFP_ATOMIC = __GFP_HIGH
  *// Cannot sleep! Use emergency pools*
  *// For interrupts, spinlocks, atomic contexts*
- GFP_NOIO = __GFP_WAIT
  *// Can sleep but no IO (avoid recursion in block layer)*
- GFP_NOFS = __GFP_WAIT | __GFP_IO
  *// Can sleep and IO, but no FS calls (avoid recursion in FS)*
- GFP_USER = __GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL
  *// User-space allocations*
- GFP_HIGHUSER = GFP_USER | __GFP_HIGHMEM
  *// User pages, can use HIHGMEM*

# Quick reference table

| Context | Flags to use | Why |
| --- | --- | --- |
| Process context, can sleep | **GFP_KERNEL** | Default, safest |
| Interrupt handler | **GFP_ATOMIC** | Cannot sleep |
| Holding spinlock | **GFP_ATOMIC** | Cannot sleep |
| Block device driver | **GFP_NOIO** | Prevent IO recursion |
| File system code | **GFP_NFS** | Prevent FS recursion |
| Need DMA-able memory | **GFP_DMA** | Allocate from ZONE_DMA; for old/legacy devices |
| User space allocations | **GFP_HIGHUSER** | Can use all zones |

**GFP_NOIO**: Don't perform any IO that may invoke a block IO operation

- Allocator cannot write dirty pages to the swap file to free up memory
- User: block device driver

**GFP_NOFS**: Can perform raw block IO, but don't call file system layer

- Don't reclaim file-related caches or free memory by writing data to a file
- User: file system code

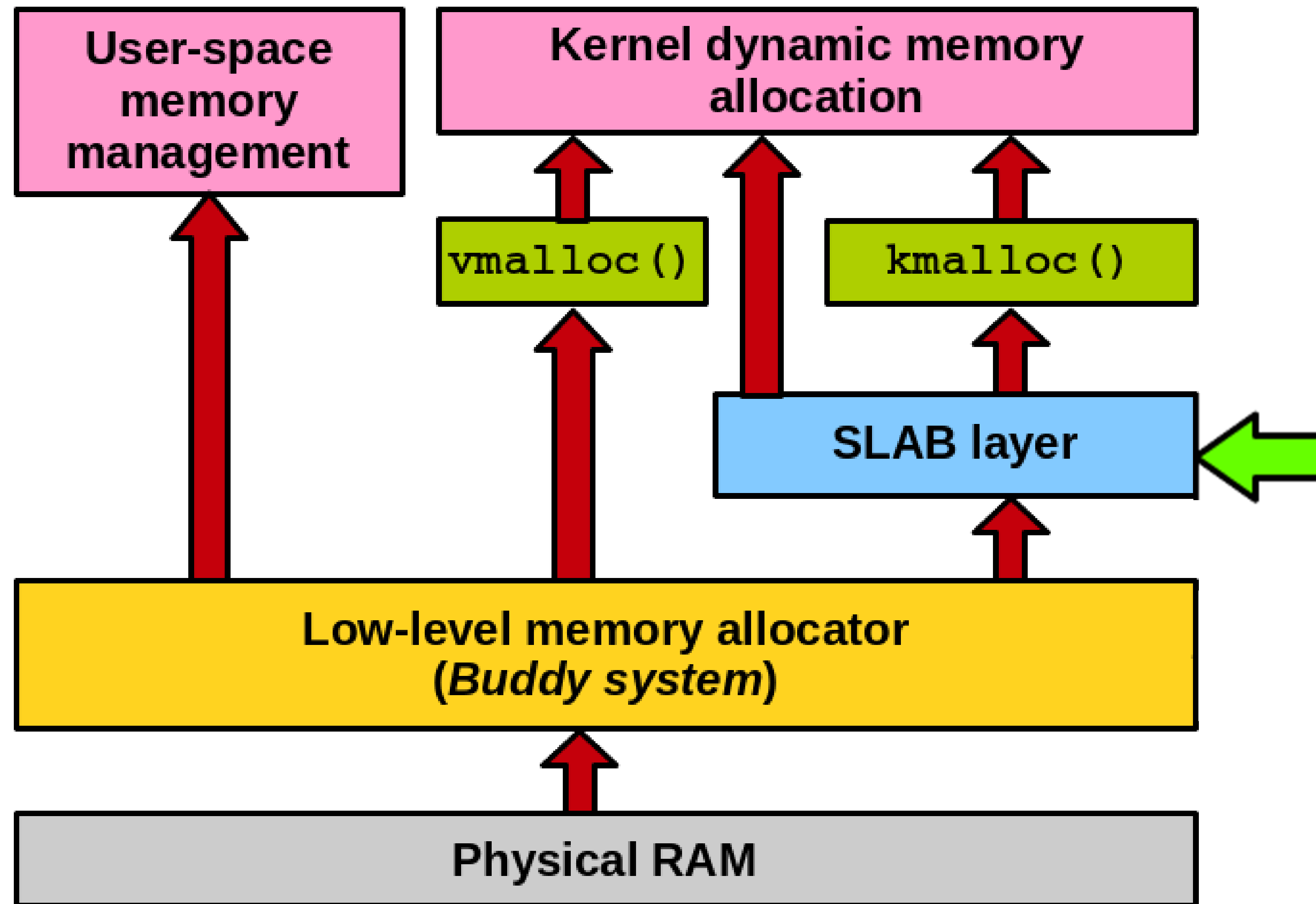
# Kernel object allocators (kmalloc/kfree)

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

- Allocate byte-sized chunks (like user-space **`malloc`**)
- Memory is **physically contiguous** (DMA buffers)
  - Maximum size allowed is 4 MB
    - For larger allocation, rely on **`vmalloc`**
- Returns kernel virtual address
- Built on top of slab allocator
- Good for small allocations (<= 4 KB)
- Performance critical (uses slab cache)

EPFL

# Hierarchy of memory allocators




https://hammertux.github.io/slab-allocator

# Slab allocator

**Problem:** Allocating/freeing frequently used structures is expensive

- `task_struct`, `inode`, `dentry`, `mm_struct`, `vm_area_struct` – created/destroyed frequently

**Simple approach:** Use kmalloc for time!

**Expensive:** lock contention, cache misses, memory fragmentation

**Q. How to minimize such an overhead?**

# Slab allocator

**Problem:** Allocating/freeing frequently used structures is expensive

- `task_struct`, `inode`, `dentry`, `mm_struct`, `vm_area_struct` – created/destroyed frequently

**Simple approach:** Use kmalloc for time!

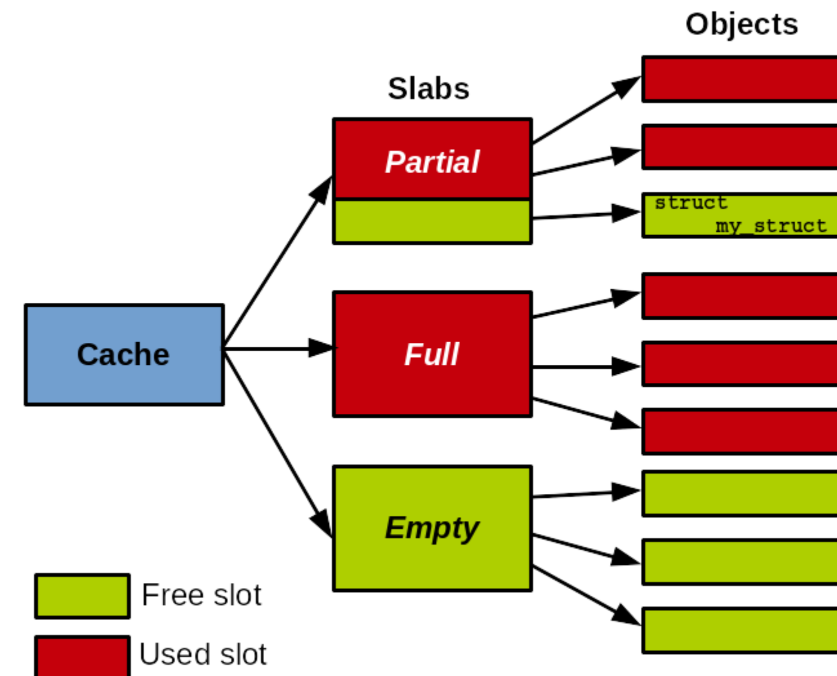
**Expensive:** lock contention, cache misses, memory fragmentation

- **Object caching:** Pre-allocate pools of common objects
  - **Allocate** → take from pool (fast!)
  - **Free** → return to pool (fast!)
  - Caching hot objects improves CPU cache performance

EPFL

# Slab allocator: Object caching

- Each slab
  - One of more physically contiguous pages
  - Divided into fixed-size objects
  - State: empty, partial, or full
- Allocation strategy:
  1. Try to allocate from the partial slab (good cache locality)
  2. If none, try an empty slab
  3. If none, allocate new pages and create a new slab



EPFL

# Slab allocator: APIs

```c
/**
 * Create a cache for a data structure type
 */
struct kmem_cache *kmem_cache_create(
    const char *name,     /* Name of the cache */
    size_t size,          /* Size of objects */
    size_t align,         /* Offset of the first element
                             within pages */
    unsigned long flags,  /* Options */
    void (*ctor)(void *)  /* Constructor */
);


/**
 * Destroy the cache
 * - Should be only called when all slabs in the cache are empty
 * - Should not access the cache during the destruction
 */
void kmem_cache_destroy(struct kmem_cache *cachep);


/**
 * Allocate an object from the cache
 */
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);

/**
 * Free an object allocated from a cache
 */
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

EPFL

# Slab allocator: advanced flags

| Flags | Purpose |
|---|---|
| **SLAB_HWCACHE_ALIGN** | Align objects to the CPU cache line<br>Prevents false sharing; Cost: more memory waste |
| **SLAB_POISON** | Fill freed objects with 0xa5a5a5a5<br>Catch use-after-free bugs |
| **SLAB_RED_ZONE** | Padding around objects<br>Catch buffer overflows |
| **SLAB_PANIC** | Panic kernel if allocation fails<br>For critical caches only |
| **SLAB_CACHE_DMA** | Allocate from ZONE_DMA<br>For DMA-able objects |

# Slab allocator: optimizing caching

- **Per-CPU caching**
  - Each CPU has a small cache of objects
  - Fast path: no locking needed
  - Slow path: refill from slab (requires lock)

- **NUMA awareness:**
  - Slabs can be node-local
  - **`kmem_cache_alloc_node()`** for explicit placement
  - Reduces *remote* memory acecss

EPFL

# Slab/slub allocator: demo

```
└─$ sudo cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
nf_conntrack_expect      0      0    208   39    2 : tunables    0    0    0 : slabdata      0      0      0
nf_conntrack            64     64    256   64    4 : tunables    0    0    0 : slabdata      1      1      0
QIPCRTR                 78     78    832   39    8 : tunables    0    0    0 : slabdata      2      2      0
rpc_inode_cache         46     46    704   46    8 : tunables    0    0    0 : slabdata      1      1      0
ext4_groupinfo_4k    29832  29832    184   44    2 : tunables    0    0    0 : slabdata    678    678      0
scsi_sense_cache        64     64    128   64    2 : tunables    0    0    0 : slabdata      1      1      0
fsverity_info            0      0    264   62    4 : tunables    0    0    0 : slabdata      0      0      0
fscrypt_inode_info       0      0    128   64    2 : tunables    0    0    0 : slabdata      0      0      0
MPTCPv6                  0      0   2176   15    8 : tunables    0    0    0 : slabdata      0      0      0
ip6-frags                0      0    184   44    2 : tunables    0    0    0 : slabdata      0      0      0
PINGv6                 104    104   1216   26    8 : tunables    0    0    0 : slabdata      4      4      0
RAWv6                  988    988   1216   26    8 : tunables    0    0    0 : slabdata     38     38      0
UDPv6                 3096   3096   1344   24    8 : tunables    0    0    0 : slabdata    129    129      0
tw_sock_TCPv6          124    124    264   62    4 : tunables    0    0    0 : slabdata      2      2      0
request_sock_TCPv6       0      0    312   52    4 : tunables    0    0    0 : slabdata      0      0      0
TCPv6                 1380   1380   2624   12    8 : tunables    0    0    0 : slabdata    115    115      0
kcopyd_job               0      0   3240   10    8 : tunables    0    0    0 : slabdata      0      0      0
dm_uevent                0      0   2888   11    8 : tunables    0    0    0 : slabdata      0      0      0
bio-136              10164  10164    192   42    2 : tunables    0    0    0 : slabdata    242    242      0
io_kiocb              5440   5440    256   64    4 : tunables    0    0    0 : slabdata     85     85      0
bio-264                 51     51    320   51    4 : tunables    0    0    0 : slabdata      1      1      0
mqueue_inode_cache     612    612    960   34    8 : tunables    0    0    0 : slabdata     18     18      0
fuse_request          6144   6144    168   48    2 : tunables    0    0    0 : slabdata    128    128      0
fuse_inode            3276   3276    896   36    8 : tunables    0    0    0 : slabdata     91     91      0
ecryptfs_inode_cache      0      0   1024   32    8 : tunables    0    0    0 : slabdata      0      0      0
ecryptfs_file_cache      0      0     16  256    1 : tunables    0    0    0 : slabdata      0      0      0
fat_inode_cache         82     82    784   41    8 : tunables    0    0    0 : slabdata      2      2      0
fat_cache                0      0     40  102    1 : tunables    0    0    0 : slabdata      0      0      0
squashfs_inode_cache  10304  10304    704   46    8 : tunables    0    0    0 : slabdata    224    224      0
jbd2_journal_head    18224  18564    120   68    2 : tunables    0    0    0 : slabdata    273    273      0
jbd2_revoke_table_s    512    512     16  256    1 : tunables    0    0    0 : slabdata      2      2      0
ext4_inode_cache   1032745 1033872   1168   28    8 : tunables    0    0    0 : slabdata  36924  36924      0
ext4_allocation_context   6784   6784    152   53    2 : tunables    0    0    0 : slabdata    128    128      0
ext4_prealloc_space   4608   4608    112   36    1 : tunables    0    0    0 : slabdata    128    128      0
ext4_io_end_vec      18432  18944     32  128    1 : tunables    0    0    0 : slabdata    148    148      0
```

# Other allocations: Per-CPU data structure

- Allow each core to maintain their own objects
  - No locking required
  - Reduces cache thrashing
- Implemented through arrays → each index corresponds to a CPU

```
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(name, type);

get_cpu_var(name);   /* Start accessing the per-CPU variable */
put_cpu_var(name);   /* Done accessing the per-CPU variable */

/* Access per-CPU data through pointers */
get_cpu_ptr(name);
put_cpu_ptr(name);

per_cpu(name, cpu); /* Access other CPU's data */
```

# Other allocations: Stack

- Each process has:
  - A user-space stack for execution
  - A kernel stack for in-kernel execution
- **User space** stack is large and grows dynamically
- **Kernel stack** has a fixed size → two pages (8 KB)
- **Interrupt stack** is for interrupt handlers → 1 page / CPU
- Goal: Keep the minimum usage of the kernel stack
  - Local variables and function parameters

Now we know how to allocate physical memory, but what happens when we run out?

# Memory pressure problem

**Scenario:**

- System has 8 GB DRAM
- Running applications want 12 GB
- What happens?

**Without swapping:** OOM (out-of-memory) killer

- Kernel kills processes to free memory
- Unpredictable, destructive

**With swapping:** Use disk as “slow RAM”

- Move inactive pages to disk
- Free physical pages for active data
- Allows overcommitment

**Virtual memory lets us use more memory than what physically exist**

# Tracking page activity

## Q. Which pages should we evict to disk?

**Least-recently used (LRU)**: approximates temporal locality

- Evicts pages that have not been accessed for a long time

Pseudocode:

```
on memory_access(page P):
    P.timestamp = current_time

on page_eviction_needed():
    evict_page = argmin(P.timestamp)
    free(evict_page)
```

EPFL

# Tracking page activity

Linux implements approximate LRU:

- a two-tier aging scheme that uses **active/inactive lists**

```
struct page {
    ...
    struct list_head lru;   // links into LRU list
    ...
};
```

Each page belongs to one LRU list at a time

```
struct zone {
    struct lruvec {
        struct list_head lists[NR_LRU_LISTS];
    } lruvec;
};
```

Zone maintains LRU lists

EPFL

# Several LRU lists

```
enum lru_list {
    LRU_INACTIVE_ANON, // Anonymous pages, not recently used
    LRU_ACTIVE_ANON, // Anonymous pages, recently used
    LRU_INACTIVE_FILE, // File-backed pages, not recently used
    LRU_ACTIVE_FILE, // File-backed pages, recently used
    LRU_UNEVICTABLE // Cannot evict
};
```

Separate lists

- Anonymous vs. file-backed have different costs
  - **Anonymous**: must write to swap before evict (expensive)
  - **File-backed**: can drop if clean, or write back if dirty (cheaper)
- Active vs inactive: two-stage aging
  - New pages go to inactive
  - Referenced again → promoted to active
  - Active pages age back to inactive if not used
    - Prevents one-time scans from evicting hot pages

# Page state and transitions for eviction

```
New page → Inactive list
     ↓ (referenced again)
Active list
     ↓ (idle for long)
Inactive list
     ↓ (still unused under pressure)
Evicted (swap/disk)
```

```
└─$ cat /proc/meminfo | grep -E "Active|Inactive"
Active:           3900640 kB
Inactive:        19868472 kB
Active(anon):     2194132 kB
Inactive(anon):     14932 kB
Active(file):     1706508 kB
Inactive(file):  19853540 kB
```

Ratio of active/inactive
→ memory pressure

**Q. Who is responsible for reclaiming?**

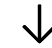
EPFL

# kswapd: The page reclaim demon

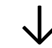
- One per NUMA node worker thread
- Maintains per zone watermark (MIN, LOW, HIGH)
- Reclaim process →
- If free pages drop below MIN, alloc_pages() must reclaim synchronously

```
└$ free -h
               total        used        free      shared  buff/cache   available
Mem:           121Gi       6.2Gi        94Gi        26Mi        22Gi       115Gi
Swap:          8.0Gi        83Mi       7.9Gi
```

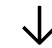
Free pages < LOW watermark



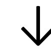
kswapd wakes up



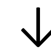
Scans LRU lists



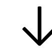
For each candidate page:



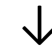
Anonymous + dirty → write to swap
File-backed + dirty → write back to file
File-backed + clean → drop from cache



Add page to free list



Free pages ≥ HIGH watermark



kswapd sleeps

EPFL

# Performance aspect of pages: hardware optimizations for faster access to pages

EPFL

# TLB problem

- Every memory access needs virtual → physical translation
- Page table walks: 4 memory accesses (PGD→PUD→PMD→PTE)
- **TLB (translation lookaside buffer)** caches translations
- But TLB is small (typically 64—1024 entries)
  - TLB with 512 entries will cover 512 * 4KB = **2MB of memory**
  - If working set size > 2 MB → TLB thrashing

**Use huge pages:** 2MB or 1GB pages instead of 4KB

- 4 KB: 64 + ~1.5K entries → 6 MB coverage
- 2 MB: ~32 entries → ~64 MB coverage
- 1 GB: 4—8 entries → 4—8 coverage

**Q. Do we have to allocate huge pages explicitly all the time?**

# Transparent huge page (THP) service

1. Process allocates memory normally
2. Kernel notices adjacent 4 KB pages are virtually contiguous (VMA)
3. Kernel allocates a 2 MB page
4. Copies 512 * 4KB pages into 2 MB
5. Updates page tables to use 2 MB entry
6. Frees old 4 KB pages

# THP benefits and costs

- Reduces TLB pressure
- Fewer TLB misses
- Better performance for large-memory workloads

**Costs:**

- Internal fragmentation (if you only need 8 KB, you get 2 MB)
- Difficult to allocate (need 512 contiguous pages)
- COW becomes expensive (copy 2 MB pages instead of 4 KB)
- Swap becomes expensive (2 MB chunks)

# Huge page pros and cons

- When to use huge pages:
  - Database servers (large buffer pools)
  - Scientific computing (large arrays)
  - Virtual machines (guest memory)
  - In-memory caches (Redis, memcached)
- When NOT to use:
  - Small memory footprint
  - Short-lived process
  - Heavily forking application (COW overhead)

# TLB management

- Filled automatically by hardware (x86)
- Must be flushed when page tables change (by the OS)
- When to flush?
  - Context switch (switch to a different process)
  - Changing page table entries (mmap, munmap, mprotect)
  - Page migration (NUMA, compaction)

```
// Flush entire TLB
 flush_tlb_all();

// Flush one page
flush_tlb_page(struct vm_area_struct *vma,
unsigned long addr);

// Flush range
flush_tlb_range(struct vm_area_struct *vma,
unsigned long start, unsigned long end);
```

EPFL

# Summary

- Physical page fundamentals
  - **struct page:** Kernel tracks every physical page
  - **Zone:** Hardware constraints
  - **NUMA:** Multi-node, non-uniform access
- Allocation mechanism:
  - **Buddy allocator:** Physical pages ($2^{Order}$ blocks)
  - **kmalloc/vmalloc:** Kernel memory allocation
  - **Slab allocator:** Object caching for performance

# Summary (contd …)

- Memory reclaim (under pressure)
  - **LRU lists:** Track page activity
  - **kswapd:** Background reclaimation
  - **Swapping:** Using disk as a “slow RAM”
- Performance:
  - **Huge pages:** Reduce TLB pressure
  - **TLB:** Hardware translation cache

**struct page → zones → buddy → kmalloc/slab → LRU → TLB**

EPFL

# Further reading

- Virtual memory: 3 What is Virtual memory?
- 20 years of Linux virtual memory