Memory Management

Have a look

- Before getting into **Memory Management**, please take a look at
  - LK_Bird's Eye View sessions,
    S4 MMU,Early Kernel,Add Types,Low High Memory.
    S5 Buddy System,Slab Cache.
    S6 Cache,TLB,Swapping.

Pages

- Although the processor’s smallest addressable unit is a **byte** or a **word**, the memory management unit (***MMU***, the hardware that manages memory and performs virtual to physical address translations) typically deals in pages.

- Each architecture defines its own page size.
  Most 32-bit architectures have **4KB** pages, whereas most 64-bit architectures have **8KB** pages.

- The kernel represents every physical page on the system with a struct ***page*** structure.

- ***page*** major elements
  - ***flags*** : generally includes whether the page is, **dirty** “ the data in RAM and the data on a secondary storage have not been synchronized” or whether it is **locked** in memory “other parts of the kernel are not allowed to access the page, or Has blocks allocated on-disk, or To be reclaimed asap or Page is under writeback, ...

- ***flags****, Cont’d*
  flags field is very rich more than that, please check ***/include/linux/page-flags.h***

/include/linux/mm_types.h

```
struct page {
    unsigned long flags;        /* Atomic flags, some possibly
                                 * updated asynchronously */
    ...
    union {
        struct {    /* Page cache and anonymous pages */
            /**
             * @lru: Pageout list, eg. active_list protected by
             * lruvec->lru_lock.  Sometimes used as a generic list
             * by the page owner.
             */
            struct list_head lru;
            /* See page-flags.h for PAGE_MAPPING_FLAGS */
            struct address_space *mapping;
            pgoff_t index;      /* Our offset within mapping. */
            /**
             * @private: Mapping-private opaque data.
             * Usually used for buffer_heads if PagePrivate.
             * Used for swp_entry_t if PageSwapCache.
             * Indicates order in the buddy system if PageBuddy.
             */
            unsigned long private;
        };
        ...
        /** @rcu_head: You can use this to free a page by RCU. */
        struct rcu_head rcu_head;
    };

    union {     /* This union is 4 bytes in size. */
        /*
         * If the page can be mapped to userspace, encodes the number
         * of times this page is referenced by a page table.
         */
        atomic_t _mapcount;

        unsigned int page_type;

        unsigned int active;        /* SLAB */
        int units;          /* SLOB */
    };
    ...
} _struct_page_alignment;
```

Pages, *Cont’d*

- ***page*** major elements, *Cont’d*
  - ***_refcount*** : number of references to this page in the kernel, if **0**, the ***page*** structure is not currently in use and can therefore be removed.

  - ***_mapcount*** : indicates how many entries in the **page table** point to the page.

  - ***lru*** : allow grouping the page (Pages got accessed recently or not).

  - A compound page structure,
    What’s a compound page!
    It’s a grouping of two or more physically contiguous pages into a unit that can be treated as a single, **larger page** "huge pages".

    Allocating a compound page,
    alloc_pages() with the ***__GFP_COMP*** flag.

/include/linux/mm_types.h

```c
/* Usage count. *DO NOT USE DIRECTLY*. See page_ref.h */
    atomic_t _refcount;
```

```c
    /*
     * If the page can be mapped to userspace, encodes the number
     * of times this page is referenced by a page table.
     */
    atomic_t _mapcount;
```

```c
    struct {    /* Page cache and anonymous pages */
        /**
         * @lru: Pageout list, eg. active_list protected by
         * lruvec->lru_lock.  Sometimes used as a generic list
         * by the page owner.
         */
        struct list_head lru;
```

```c
    struct {    /* Tail pages of compound page */
        unsigned long compound_head;    /* Bit zero is set */

        /* First tail page only */
        unsigned char compound_dtor;
        unsigned char compound_order;
        atomic_t compound_mapcount;
        unsigned int compound_nr; /* 1 << compound_order */
    };
```

Pages, *Cont'd*

- ***page*** major elements, *Cont'd*
  - ***virtual*** : is used for pages in the **highmem** area.
    highmem : pages that cannot be directly mapped into kernel space.
    For details, please refer to
    LK_Bird's Eye View -
    S4 MMU,Early Kernel,Address Types,Low High Memory.
  - ***mapping*** : specifies the address space in which a page frame is located, ***index*** is the offset within the mapping.

/include/linux/mm_types.h

```
struct address_space *mapping;
pgoff_t index;       /* Our offset within mapping. */
```

/include/linux/mm_types.h

```
struct address_space {
    struct inode        *host;
    struct xarray       i_pages;
    gfp_t           gfp_mask;
    atomic_t        i_mmap_writable;
#ifdef CONFIG_READ_ONLY_THP_FOR_FS
    /* number of thp, only for non-shmem files */
    atomic_t        nr_thps;
#endif
    struct rb_root_cached   i_mmap;
    struct rw_semaphore i_mmap_rwsem;
    unsigned long       nrpages;
    pgoff_t         writeback_index;
    const struct address_space_operations *a_ops;
    unsigned long       flags;
    errseq_t        wb_err;
    spinlock_t      private_lock;
    struct list_head    private_list;
    void            *private_data;
} __attribute__((aligned(sizeof(long)))) __randomize_layout;
```

```
#if defined(WANT_PAGE_VIRTUAL)
    void *virtual;          /* Kernel virtual address (NULL if
                       not kmapped, ie. highmem) */
#endif /* WANT_PAGE_VIRTUAL */
```

Pages, Cont'd

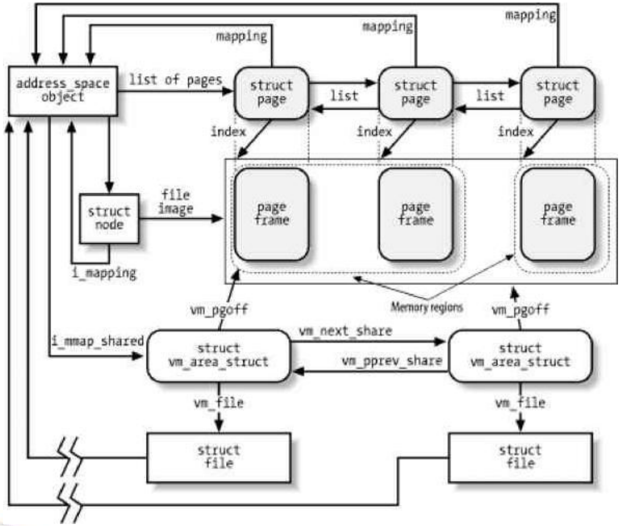
- Let's introduce the concept of **page cache** maintained by ***address_space*** structure.
  - What <u>**page cache**</u>: it is a cache of pages in **RAM**.The pages originate from reads and writes of regular filesystem files, block device files, and memory-mapped files. (Not Anonymous page).
  - Page cache contains chunks of recently accessed files.
  - During any page I/O operation, such as read(),.. kernel checks whether resides in the page cache, if not we need to the secondary storage.
  - So, ***address_space*** here describes the physical pages of a file, ***address_space*** instance serves as an abstraction for a set of pages owned by either a file **inode or block device file inode *(*host)***

    nrpages: Number of pages, protected by the ***i_pages*** lock.

    i_mmap : list of all mappings, allows the kernel to find the mappings associated with this cached file.

/include/linux/mm_types.h

```
struct address_space {
    struct inode          *host;
    struct xarray         i_pages;
    gfp_t                 gfp_mask;
    atomic_t              i_mmap_writable;
#ifdef CONFIG_READ_ONLY_THP_FOR_FS
    /* number of thp, only for non-shmem files */
    atomic_t              nr_thps;
#endif
    struct rb_root_cached   i_mmap;
    struct rw_semaphore i_mmap_rwsem;
    unsigned long         nrpages;
    pgoff_t               writeback_index;
    const struct address_space_operations *a_ops;
    unsigned long         flags;
    errseq_t              wb_err;
    spinlock_t            private_lock;
    struct list_head      private_list;
    void                  *private_data;
} __attribute__((aligned(sizeof(long)))) __randomize_layout;
```



Zones

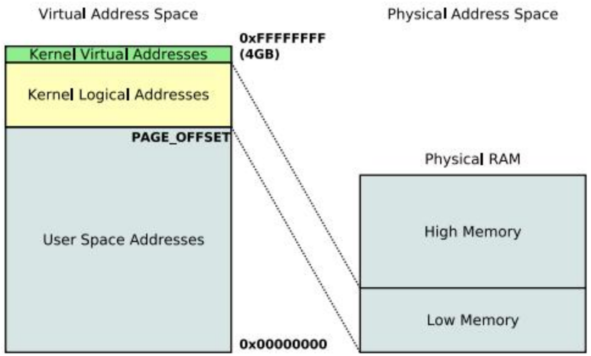
- Because of hardware limitations, the kernel divides pages into different ***zones.***

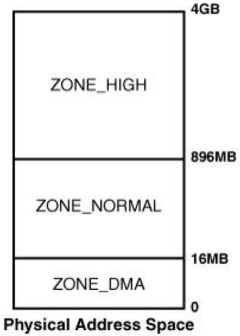
- The major limitations can be as follows,
  - Some hardware devices can perform ***DMA*** to only certain memory addresses.

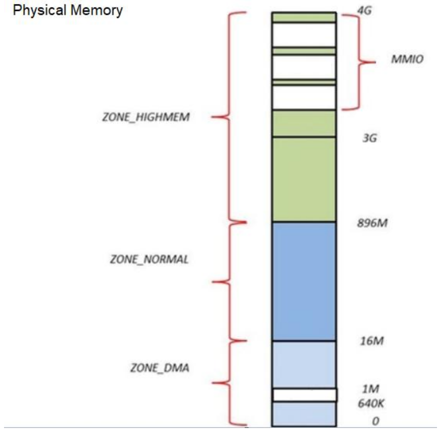
  - Some architectures can physically addressing larger amounts of memory than they can virtually address. Consequently, some memory is not permanently mapped into the kernel address space.

    Remember when,
    High memory → kernel virtual address.

- i.e. x86-32 has the following memory zones.







Zones, *Cont'd*

- Linux generally declares the following zones,
  - ***ZONE_DMA*** : This zone contains pages that can be used by **DMA**.
  - ***ZONE_DMA32*** : Like ***ZOME_DMA***, but these pages are accessible only by **32-bit** devices.
  - ***ZONE_NORMAL*** : This zone contains normal, regularly mapped, pages.
  - ***ZONE_HIGHMEM*** : This zone contains "**high memory**," which are pages not permanently mapped into the kernel address space.
  - ***ZONE_MOVABLE*** : Since physically scattered memory can always be mapped to virtually contiguous address space through page tables.

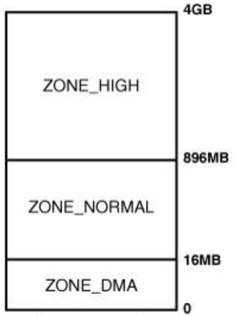
    Introduction of ***ZONE_MOVABLE*** is one of the attempts to overcome the fragmentation issues that happen in most of the systems through the runtime.

The idea is to track ***movable*** pages in each zone and represent them under this **pseudo** zone, which helps prevent fragmentation

As a general rule, the memory manager is configured to consider migration of pages from the highest populated zone ***(ZONE_HIGHMEM, ZONE_DMA32, …)*** to ***ZONE_MOVABLE.***

/include/linux/mmzone.

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
    ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
    ZONE_DMA32,
#endif

    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM

    ZONE_HIGHMEM,
#endif

    ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
    ZONE_DEVICE,
#endif
    __MAX_NR_ZONES
};
```



Physical Address Space

Zones, *Cont'd*

- Linux kernel zones, ***Cont'd***
  - ***ZONE_DEVICE*** : This zone is designed to support **hotplug memories**,like large capacity **persistent memory arrays**

    Persistent memories are very similar to DRAM but its size (usually measured in terabytes).

    For the kernel to support such memories with **4KB** page size, it would need to enumerate billions of ***page*** structures.

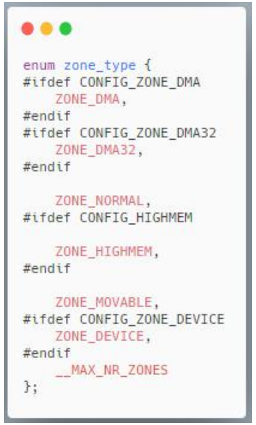
    So, the kernel consider persistent memory as **device** and manages it through the **device drivers**.

    The ***devm_memremap_pages()*** routine of the persistent memory driver maps a region of persistent memory into kernel address space with relevant **page structures**.

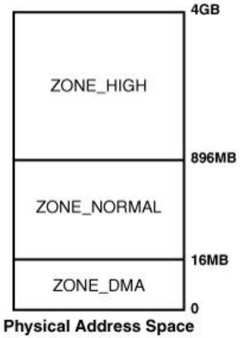
    All pages under these mappings are grouped under ***ZONE_DEVICE.***

Check the link for more details about "Persistent Memory"

/include/linux/mmzone.h

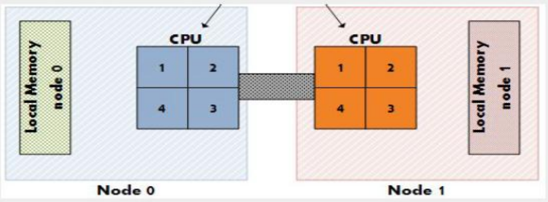


```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
    ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
    ZONE_DMA32,
#endif

    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM

    ZONE_HIGHMEM,
#endif

    ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
    ZONE_DEVICE,
#endif
    __MAX_NR_ZONES
};
```



Zones, Cont'd

- Each zone is represented by ***struct zone,***
  - ***lock*** : a spin lock that protects the structure from concurrent access.
  - Zone pages,
    UMA machines : ***pglist_data*zone_pgdat***, points the only single node exist in the system.
    NUMA machines : ***pglist_data* zone_pgdat***, points to the node which zone belongs to.



node_zones[] : Contains the different types of zones for the node.

page *node_mem_map : In case of ***FLATMEM***, points to an array of page instances used to describe all physical pages of the node. It includes the pages of all zones in the node.

/include/linux/mmzone.h

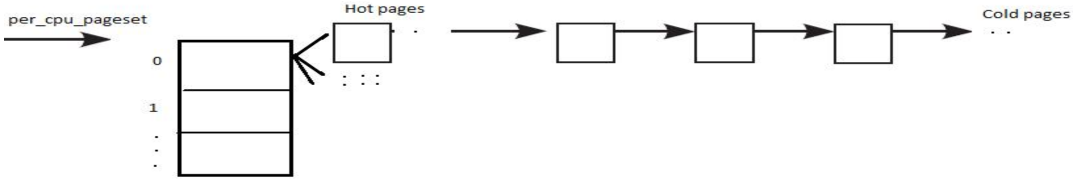
```
struct zone {

    unsigned long _watermark[NR_WMARK];
    ...
    long lowmem_reserve[MAX_NR_ZONES];

    struct pglist_data  *zone_pgdat;
    struct per_cpu_pages    __percpu *per_cpu_pageset;


    /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
    unsigned long       zone_start_pfn;
    ...
    /* free areas of different sizes */
    struct free_area    free_area[MAX_ORDER];

    /* Primarily protects free_area */
    spinlock_t      lock;

    /* Zone statistics */
    atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
} ____cacheline_internodealigned_in_smp;
```

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];

    int nr_zones; /* number of populated zones in this node */
#ifdef CONFIG_FLATMEM   /* means !SPARSEMEM */
    struct page *node_mem_map;
    ...
#endif
} pg_data_t;
```

Zones, Cont’d

- ***struct zone***, *Cont’d,*
  - ***zone_start_pfn*** : Index of first page frame in zone.
    Kernel uses ***pfn_to_page()*** to get the the first ***struct page * mem_map*** page pointer for zone.
  - ***spanned_pages*** : the total pages spanned by the zone.
  - ***per_cpu_pageset***: Points to **hot-n-cold** pages.
    what is **hot-n-cold**?
    hot : refers to a page that is in a CPU cache and its data can be accessed quicker than if it were in RAM.
    cold : refers to page is not held in cache.

    Each ***zone***, for each CPU, there are **3 hot and cold** page linked list, corresponding to ***MIGRATE_UNMOVABLE***, ***MIGRATE_MOVABLE***, ***MIGRATE_RECLAIMABLE.***



One list for **hot-n-cold** pages such that,
If a ***hot*** page is required, remove a page from the header (this page is the most "**hot**"), and if a ***cold*** page is required, remove the page from the end of the list (this page is "**cold**").

/include/linux/mmzone.h

```c
struct zone {

    unsigned long _watermark[NR_WMARK];
    ...
    long lowmem_reserve[MAX_NR_ZONES];

    struct pglist_data  *zone_pgdat;
    struct per_cpu_pages    __percpu *per_cpu_pageset;


    /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
    unsigned long       zone_start_pfn;
    ...
    /* free areas of different sizes */
    struct free_area    free_area[MAX_ORDER];

    /* Primarily protects free_area */
    spinlock_t      lock;

    /* Zone statistics */
    atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
} ____cacheline_internodealigned_in_smp;
```

Zones, Cont'd

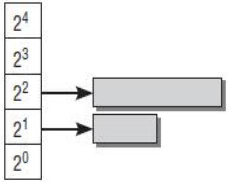
- ***struct zone***, *Cont'd,*
  - ***free_area*** : is an array of data structures of the same name used to implement the buddy system.
    Each array element stands for contiguous memory areas of a fixed size.

  - ***_watermark*** : The kernel can write pages to hard disk if insufficient RAM memory is available, this array holds the **min**, **low**, and **high** watermarks for this zone.

    The kernel uses watermarks such that,
    if the **free pages** inside zone **> high** , the state of the zone is ideal.

    free pages < low, the kernel begins to swap pages out onto the hard disk.
    free pages < **min**, the pressure to reclaim pages is increased because free pages are urgently needed in the zone.

- ***lowmem_reserve*** : array specifies several pages for each memory zone that are reserved for critical allocations that must not fail under any circumstances.



/include/linux/mmzone.h

```c
struct zone {

    unsigned long _watermark[NR_WMARK];
    ...
    long lowmem_reserve[MAX_NR_ZONES];

    struct pglist_data  *zone_pgdat;
    struct per_cpu_pages    __percpu *per_cpu_pageset;


    /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
    unsigned long       zone_start_pfn;
    ...
    /* free areas of different sizes */
    struct free_area    free_area[MAX_ORDER];

    /* Primarily protects free_area */
    spinlock_t      lock;

    /* Zone statistics */
    atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
} ____cacheline_internodealigned_in_smp;
```

Zones, Cont'd

- ***struct zone***, *Cont'd*
  - ***vm_stat*** : keeps statistical information about the zone.

/include/linux/mmzone.h

```
enum zone_stat_item {
    /* First 128 byte cacheline (assuming 64 bit words) */
    NR_FREE_PAGES,
    NR_ZONE_LRU_BASE, /* Used only for compaction and reclaim retry */
    NR_ZONE_INACTIVE_ANON = NR_ZONE_LRU_BASE,
    NR_ZONE_ACTIVE_ANON,
    NR_ZONE_INACTIVE_FILE,
    NR_ZONE_ACTIVE_FILE,
    NR_ZONE_UNEVICTABLE,
    NR_ZONE_WRITE_PENDING,  /* Count of dirty, writeback and unstable pages */
    NR_MLOCK,       /* mlock()ed pages found and moved off LRU */
    /* Second 128 byte cacheline */
    NR_BOUNCE,
#if IS_ENABLED(CONFIG_ZSMALLOC)
    NR_ZSPAGES,     /* allocated in zsmalloc */
#endif
    NR_FREE_CMA_PAGES,
    NR_VM_ZONE_STAT_ITEMS };
```

NR_FREE_PAGES : Free pages in the zone.
NR_ZONE_INACTIVE_ANON : Anonymous memory pages that has not been used recently and can be swapped out.

```
/* Zone statistics */
atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
```

NR_ZONE_ACTIVE_ANON : Anonymous memory pages that has been used more recently and usually not swapped out.

NR_ZONE_INACTIVE_FILE : Pagecache memory that can be reclaimed.

NR_ZONE_ACTIVE_FILE : Pagecache memory that has been used more recently and usually not reclaimed until needed.

NR_ZONE_UNEVICTABLE : Unevictable pages can't be swapped out for a variety of reasons.

NR_MLOCK : Pages locked to memory using the ***mlock()*** system call,
mlock(): lock part or all of the calling process's virtual address space into RAM, preventing that memory from being paged to the swap area.

Zones, Cont'd

- ***struct zone**, Cont'd,*
  - *ZONE_PADDING() ,*
    ____cacheline_internodealigned_in_smp

    zone structures are very frequently accessed.
    On multiprocessor systems, it commonly occurs that different CPUs try to access structure elements at the same time.Those structure elements are held in caches and caches are divided into lines.

    so, for fast access the elements specifically the ***lock***, the kernel invokes the ***ZONE_PADDING*** macro to generate ''**padding**'' that is added to the structure to ensure that the **lock** is in its own cache line.

    For the rest sections, keep the relevant elements in each section in a cache line for quick access.

```
struct zone_padding {
    char x[0];
} ____cacheline_internodealigned_in_smp;
#define ZONE_PADDING(name)  struct zone_padding name;
```

The compiler keyword ***__cacheline_maxaligned_in_smp*** is also used to achieve optimal cache alignment for the structure.

/include/linux/mmzone.h

```
struct zone {
    .
    .
    unsigned long       spanned_pages;
    unsigned long       present_pages;
#ifdef CONFIG_MEMORY_HOTPLUG
    /* see spanned/present_pages for more
description */
    seqlock_t       span_seqlock;
#endif
    /* Write-intensive fields used from the
page allocator */
    ZONE_PADDING(_pad1_)
    .
    .
    /* free areas of different sizes */
    struct free_area    free_area[MAX_ORDER];
    /* Primarily protects free_area */
    spinlock_t      lock;

    /* Write-intensive fields used by
compaction and vmstats. */
    ZONE_PADDING(_pad2_)
    .
    .
    ZONE_PADDING(_pad3_)
    .
    .
} ____cacheline_internodealigned_in_smp;
```

Non-Uniform Memory Access (NUMA)

- NUMA systems generally is the model of nonuniformity organization of the system memories in which the **access times** for different memory locations from a given CPU may vary.

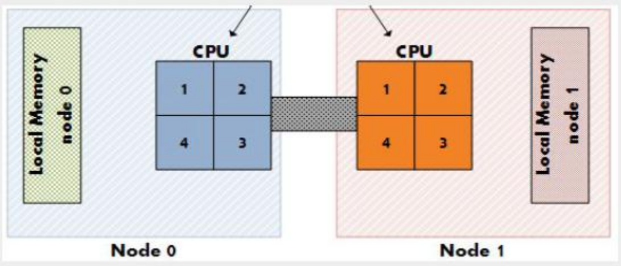
- The physical memory of the system is partitioned in several **nodes**.

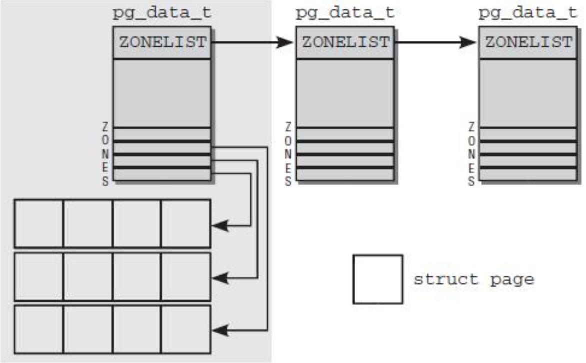
- The time needed by a given CPU to access pages within a single node is the same. However, this time might not be the same for two different CPUs.

- kernel makes use of NUMA even for some peculiar uniprocessor systems that have huge "**holes**" in the physical address space. The kernel handles these architectures by assigning the contiguous subranges of valid physical addresses to different memory **nodes**.

- The kernel represents the memory **node** associated with each processor by ***pg_data_t*** structure.

- Each **node** is split into ***zones,*** each ***zone*** has its own memory pages as shown.





Non-Uniform Memory Access (NUMA), *Cont'd*

- Node structure ***pg_data_t***,
  - ***node_zones*** is an array that holds the data structures of the zones in the node.

  - ***node_zonelists*** specifies alternative nodes and their zones in the order in which they are used for memory allocation if no more space is available in the current zone.
  - ***node_start_pfn*** is the logical number of the first page frame of the NUMA node. The page frames of all nodes in the system are numbered consecutively, and each frame is given a number that is globally unique (not just unique to the node).

    node_start_pfn is always 0 in a UMA system because there only one node whose first page frame is therefore 0.

  - ***node_present_pages*** specifies the number of page frames in the zone and ***node_spanned_page*** : total size of physical page range, including holes.

/include/linux/mmzone.h

- ***kswapd_wait*** is the wait queue for the swap daemon needed when swapping frames out of the zone.

- ***kswapd*** points to the task structure of the swap daemon responsible for the zone

```
typedef struct pglist_data {

    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];

    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 */
    ...
    wait_queue_head_t kswapd_wait;

    struct task_struct *kswapd; /* Protected by
                                   mem_hotplug_begin/end() */
    ...
    /* Write-intensive fields used by page reclaim */
    ZONE_PADDING(_pad1_)
    ...
    ZONE_PADDING(_pad2_)
    ...
} pg_data_t;
```

```
struct zoneref {
    struct zone *zone;    /* Pointer to actual zone */
    int zone_idx;         /* zone_idx(zoneref->zone) */
};
struct zonelist {
    struct zoneref _zonerefs[MAX_ZONES_PER_ZONELIST + 1];
};
```

Node and Zone Initialization

- ***start_kernel()*** includes the system initialization functions associated with memory management.

- ***build_all_zonelists()*** builds the data structures required to manage nodes and their zones.

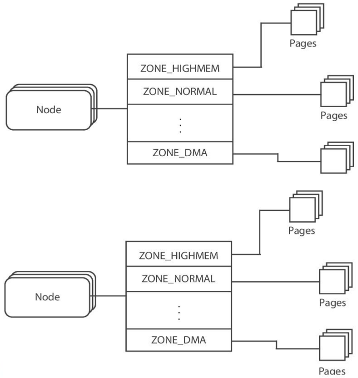
- ***build_all_zonelists()*** doing the main functionality through ***build_all_zonelists_init().***

- ***build_all_zonelists_init()*** calls ***__build_all_zonelists()*** that generally iterates over all active nodes in the system, as for **UMA** systems have only one node, ***build_zonelists()*** is invoked just once to create the zone lists for the whole of memory.
  For **NUMA** systems must invoke the function as many times as there are **nodes**.

- ***Build_zonelists()*** :
  - Establish a ranking order between the zones of the node currently being processed and the other nodes in the system; memory is then allocated according to this order. This is important if no memory is free in the desired node zone.

```
asmlinkage __visible void __init __no_sanitize_address start_kernel(void)
{
    ...
    build_all_zonelists(NULL);
    ...
}
```

```
static void __build_all_zonelists(void *data)
{
    ...
    /*
     * This node is hotadded and no memory is yet present.   So just
     * building zonelists is fine - no need to touch other nodes.
     */
    if (self && !node_online(self->node_id)) {
        build_zonelists(self);
    } else {
        for_each_online_node(nid) {
            pg_data_t *pgdat = NODE_DATA(nid);

            build_zonelists(pgdat);
        }
    ...
}
```



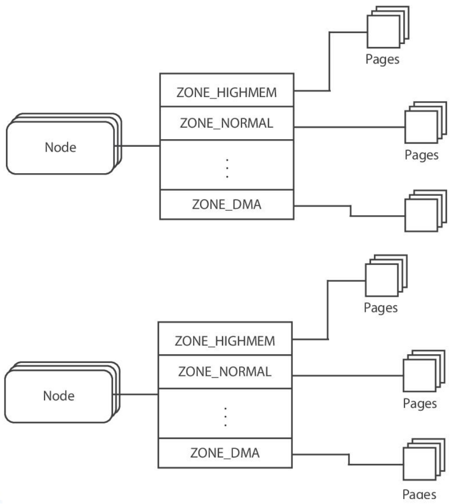
Node and Zone Initialization, *Cont'd*

- ***Build_zonelists()*** *Cont'd* : i.e.
  - Suppose the kernel wants to allocate **high memory** It first attempts to find a free segment of suitable size in the highmem area of the **current node**, if it fails, it looks at the **normal memory area of the node**.

  - If this also fails, it tries to perform allocation in the **DMA zone of the node.**
    If it cannot find a free area in any of the three local zones, it looks at **other nodes.**

  - In this case, the **alternative node** should be as **close** as possible to the primary node to minimize performance loss caused as a result of **accessing non-local memory**.

```c
static void build_zonelists(pg_data_t *pgdat)
{
    ...
    while ((node = find_next_best_node(local_node, &used_mask)) >= 0) {
        /*
         * We don't want to pressure a particular node.
         * So adding penalty to the first node in same
         * distance group to make it round-robin.
         */
        if (node_distance(local_node, node) !=
            node_distance(local_node, prev_node))
            node_load[node] = load;

        node_order[nr_nodes++] = node;
        prev_node = node;
        load--;
    }

    build_zonelists_in_node_order(pgdat, node_order, nr_nodes);
    build_thisnode_zonelists(pgdat);
}
```



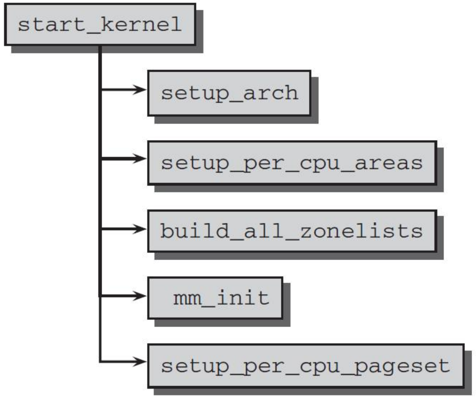
Initialization of Memory Management

- System Start
  - ***setup_arch()*** : is an architecture-specific set-up function responsible for, among other things, initialization of the system boot up and memory allocator related to it **bootmem**.

  - ***setup_per_cpu_areas()*** : On SMP systems, initializes per-CPU variables defined statically in the source code (using the per_cpu macro) and of which there is a separate copy for each CPU in the system. Variables of this kind are stored in a separate section of the kernel binaries. The purpose of ***setup_per_cpu_areas*** is to create a copy of these data for each system CPU.

  - ***build_all_zonelists()*** : sets up the node and zone data structures (as discussed).

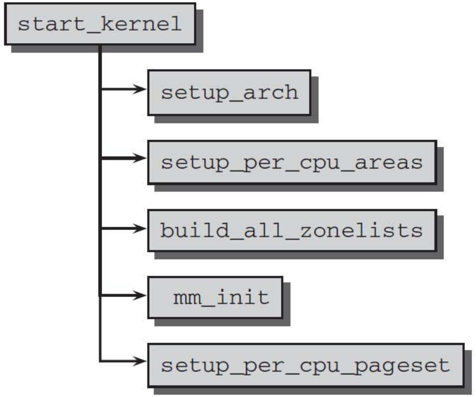
  - ***mm_init()*** is another architecture-specific function to disable the **bootmem** allocator and perform the transition to the actual memory management functions.



Initialization of Memory Management, *Cont'd*

- System Start, *Cont'd*
  - ***setup_per_cpu_pageset()*** : Allocate per cpu pagesets and initialize them (**hot-n-cold** per-cpu related to each zone as discussed earlier).
- Architecture-Specific Setup
  - Let's pick the **IA-32** architecture as an example.
  - **Arrangement of the Kernel in Memory**
    first, we need to examine the situation in RAM after the boot loader has copied the kernel into memory at case in which the kernel is loaded to a fixed position in physical RAM that is determined at compile time.

    The configuration option ***PHYSICAL_START*** determines the position in RAM in this case.




```
struct zone {
    ...
    struct per_cpu_pages    __percpu
*per_cpu_pageset;
    ...
}
```

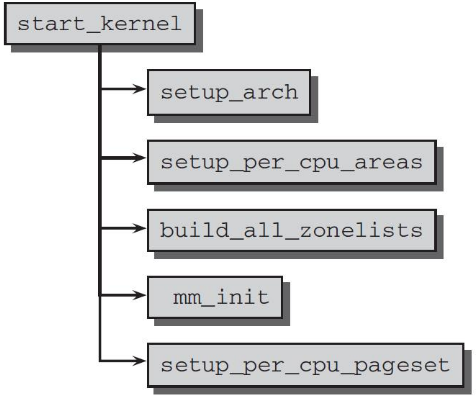
Initialization of Memory Management, *Cont'd*

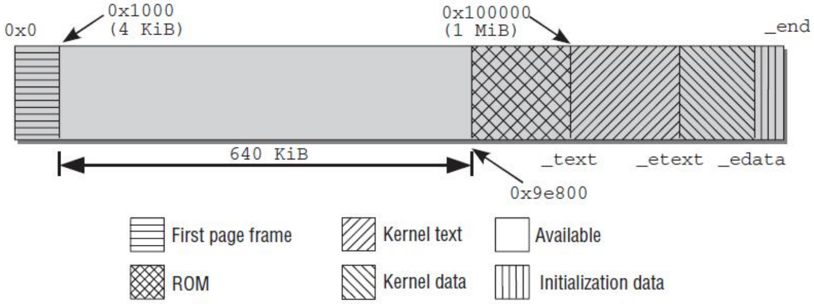
- Architecture-Specific Setup, *Cont'd*
  - **Arrangement of the Kernel in Memory**, *cont'd*
    The kernel also can be built as a **relocatable** binary, and the physical start address given at compile time is ignored in this case, the boot loader can decide where to put the kernel.

    Let see an example for **IA-32** shows the physical memory in which parts of the kernel image reside.
    Generally the required memory depends on how big the kernel binary is.

    - **First 4KB**: often reserved for the **BIOS**.
    - **640KB** : can be theoretically used but not used for kernel loading, because it's followed by an area reserved for the system (typically the system BIOS and the graphic card **ROM**).

    - So, IA-32 kernels use **0x100000** as the start address because the kernel should always be loaded into a contiguous memory range.





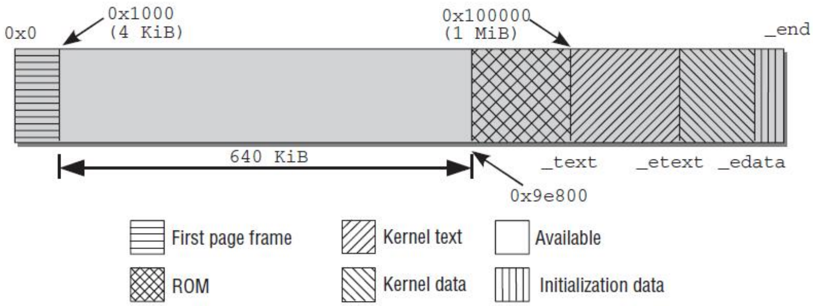
Initialization of Memory Management, *Cont'd*

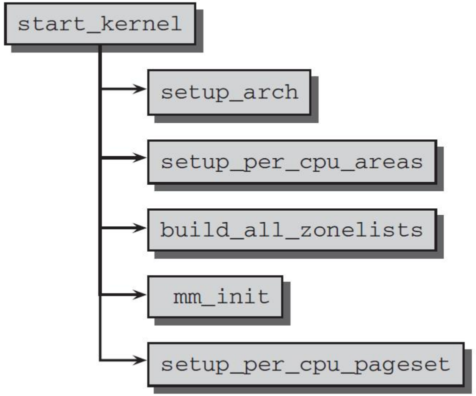
- Architecture-Specific Setup, *Cont'd*
  - **Arrangement of the Kernel in Memory**, *cont'd*
    - **_text** and **_etext** : the start and end address of the text section that contains the compiled kernel code.

    - **_etext** to **_edata :** data section in which most kernel variables are kept is located.

    - **_edata** to **_end** : Initialized data for the kernel.
  - ***setup_arch(char **cmdline_p)*** : huge code for init the HW, but generally it includes,

    - Setup memory regions.
    - Set values for the start of kernel code, **_edata** to **_end** of kernel code as depicted.





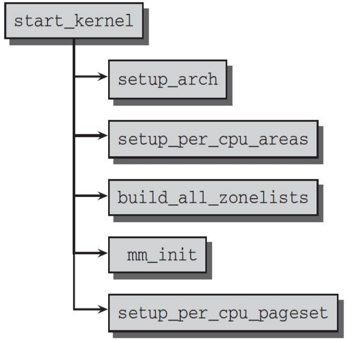
Initialization of Memory Management, *Cont'd*

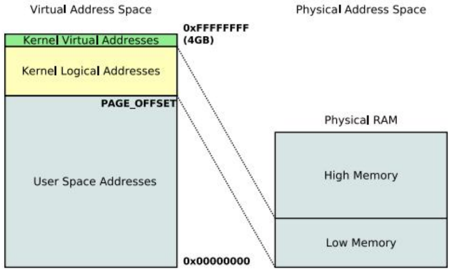
- Architecture-Specific Setup, *Cont'd*
  - ***setup_arch(char **cmdline_p)***, *Cont'd*
    - ***parse_early_param***, performs interpretation of the command-line parameters passed by the bootloader relating to memory management setup; i.e. the total size of available physical memory, or the position of specific ACPI and BIOS memory areas.

    - Setup **bootmem** allecator.

    - ***paging_init,*** sets up the **kernel's** reference page table to map physical memory and also the ***vmalloc*** areas.

  - ***mm_init()***
    Set up kernel memory allocators (Buddy system, Slab allocators,...).





```c
static void __init mm_init(void)
{
	/*
	 * page_ext requires contiguous pages,
	 * bigger than MAX_ORDER unless SPARSEMEM.
	 */
	page_ext_init_flatmem();
	init_mem_debugging_and_hardening();
	kfence_alloc_pool();
	report_meminit();
	stack_depot_init();
	mem_init();
	mem_init_print_info();
	/* page_owner must be initialized after buddy is ready */
	page_ext_init_flatmem_late();
	kmem_cache_init();
	kmemleak_init();
	pgtable_init();
	debug_objects_mem_init();
	vmalloc_init();
	/* Should be run before the first non-init thread is created */
	init_espfix_bsp();
	/* Should be run after espfix64 is set up. */
	pti_init();
}
```

Getting Pages

- Let's look at the interfaces the kernel implements to enable you to allocate and free memory **within the kernel** with a **page-sized** granularity.

- ***alloc_pages()*** : Allocates 2^order contiguous physical pages and returns a pointer to the first page's page structure.

- ***alloc_page()*** : Getting only one page directly.

- ***page_address()*** : Get the mapped virtual address of a page.
  if null, kernel uses ***map_new_virtual()*** to map the page to VAS.

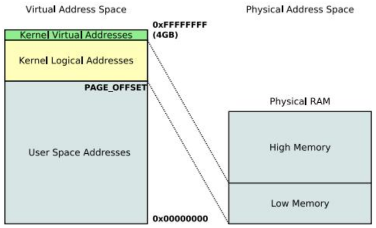
- Zeroed Pages : If you need the returned page filled with zeros, ***get_zeroed_page().***

- Freeing Pages: Family of functions enables you to free allocated pages,
  __free_pages(), ***free_pages()***, ***free_page().***

- i.e. want to allocate 8 pages →

```
static inline struct page *
alloc_pages(gfp_t gfp_mask, unsigned int order)
{
    return alloc_pages_current(gfp_mask, order);
}
```

```
void *page_address(const struct page *page)
{
    ...
}
```




```
unsigned long get_zeroed_page(gfp_t gfp_mask)
{
    return __get_free_pages(gfp_mask | __GFP_ZERO, 0);
}
```

```
page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
    /* insufficient memory: you must handle this error! */
    return -ENOMEM;
}
/* 'page' is now the address of the first of eight contiguous pages ...
*/
/* And here we free the eight pages, after we are done using them: */
free_pages(page, 3);
```

kmalloc

- Allocating physically **contiguous kernel memory** in byte-sized chunks.

- ***NULL*** returned in case of insufficient amount of memory is available.

- ***gfp_t*** Flags,
  - gfp stands for “**get free pages**”.

  - Flags are broken up into three categories: **action modifiers**, **zone modifiers**, and **types**.

  - **Action modifiers** specify how the kernel is supposed to allocate the requested memory, only certain methods can be employed to allocate memory,
    i.e. While writing interrupt handlers must instruct the kernel not to sleep (because interrupt handlers cannot reschedule) in the course of allocating memory.
    These allocations can be specified together.
    i.e. ***ptr = kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);***

```
void * kmalloc(size_t size, gfp_t flags)
{

}
```

```
/* Example */
p = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
    /* handle error ... */
```

Action Modifiers

Flag	Description
__GFP_WAIT	The allocator can sleep.
__GFP_HIGH	The allocator can access emergency pools.
__GFP_IO	The allocator can start disk I/O.
__GFP_FS	The allocator can start filesystem I/O.
__GFP_NOWARN	The allocator does not print failure warnings.

kmalloc, *Cont'd*

- ***gfp_t*** Flags, *Cont'd*
  - **Zone modifiers** specify from where to allocate memory,"which memory zone".
  - **Type** flags specify a combination of **action** and **zone** modifiers as needed by a certain **type** of memory allocation,
  - Instead of providing a **combination of action and zone modifiers**, you can specify just one **type** flag.

```
/* Example */
p = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
    /* handle error ... */
```

Zone Modifiers

Flag	Description
__GFP_DMA	Allocates only from ZONE_DMA
__GFP_DMA32	Allocates only from ZONE_DMA32
__GFP_HIGHMEM	Allocates from ZONE_HIGHMEM or ZONE_NORMAL

Modifiers Behind Each Type Flag

Flag	Modifier Flags
GFP_ATOMIC	__GFP_HIGH
GFP_NOWAIT	0
GFP_NOIO	__GFP_WAIT
GFP_NOFS	(__GFP_WAIT | __GFP_IO)
GFP_KERNEL	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_USER	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_HIGHUSER	(__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM)
GFP_DMA	__GFP_DMA

Type Flags

Flag	Description
GFP_ATOMIC	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spin-lock, and in other situations where you cannot sleep.
GFP_NOWAIT	Like GFP_ATOMIC, except that the call will not fallback on emergency memory pools. This increases the liklihood of the memory allocation failing.
GFP_NOIO	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when you cannot cause more disk I/O, which might lead to some unpleasant recursion.
GFP_NOFS	This allocation can block and can initiate disk I/O, if it must, but it will not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.
GFP_KERNEL	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to do to obtain the memory requested by the caller. This flag should be your default choice.
GFP_USER	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
GFP_HIGHUSER	This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.
GFP_DMA	This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the preceding flags.

kmalloc, *Cont'd*

- ***gfp_t*** Flags, *Cont'd*
  - **Type** flag, *Cont'd*

```
/* Example */
p = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
    /* handle error ... */
```

Which Flag to Use When

Situation	Solution
Process context, can sleep	Use GFP_KERNEL.
Process context, cannot sleep	Use GFP_ATOMIC, or perform your allocations with GFP_KERNEL at an earlier or later point when you can sleep.
Interrupt handler	Use GFP_ATOMIC.
Softirq	Use GFP_ATOMIC.
Tasklet	Use GFP_ATOMIC.
Need DMA-able memory, can sleep	Use (GFP_DMA | GFP_KERNEL).
Need DMA-able memory, cannot sleep	Use (GFP_DMA | GFP_ATOMIC), or perform your allocation at an earlier point when you can sleep.

Zone Modifiers

Flag	Description
__GFP_DMA	Allocates only from ZONE_DMA
__GFP_DMA32	Allocates only from ZONE_DMA32
__GFP_HIGHMEM	Allocates from ZONE_HIGHMEM or ZONE_NORMAL

Type Flags

Flag	Description
GFP_ATOMIC	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spinlock, and in other situations where you cannot sleep.
GFP_NOWAIT	Like GFP_ATOMIC, except that the call will not fallback on emergency memory pools. This increases the liklihood of the memory allocation failing.
GFP_NOIO	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when you cannot cause more disk I/O, which might lead to some unpleasant recursion.
GFP_NOFS	This allocation can block and can initiate disk I/O, if it must, but it will not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.
GFP_KERNEL	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to do to obtain the memory requested by the caller. This flag should be your default choice.
GFP_USER	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
GFP_HIGHUSER	This is an allocation from ZONE_HIGHMEM and might block. This flag is used to allocate memory for user-space processes.
GFP_DMA	This is an allocation from ZONE_DMA. Device drivers that need DMA-able memory use this flag, usually in combination with one of the preceding flags.

- ***kfree()*** : The counterpart to ***kmalloc()***

```
buf = kmalloc(BUF_SIZE, GFP_ATOMIC);
if (!buf)
/* error allocating memory ! */
...
kfree(buf);
```

vmalloc

- Similar to ***kmalloc()***, except it allocates memory that is only **virtually contiguous** and **not necessarily physically contiguous**.

- It does this by allocating non contiguous chunks of physical memory and **“fixing up” the page tables to map the memory into a contiguous chunk of the logical address space**.

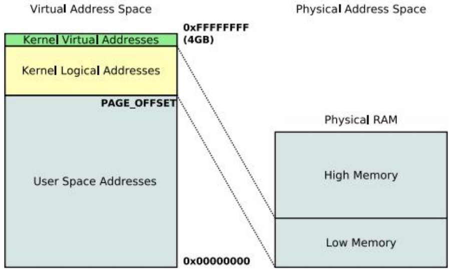
- For the most part, only **hardware devices** require physically contiguous memory allocations so that, ***vmalloc()*** not used in such cases.

- Most kernel code uses ***kmalloc()*** because ***vmalloc()*** to make non physically contiguous pages contiguous in the virtual address space, must set up slightly different page table where, pages obtained via ***vmalloc()*** must be mapped by their individual pages (because they are not physically contiguous), which results in greater **TLB**.

- Blocks of memory used only by software i.e. **process-related buffers** are fine using memory that is only **virtually contiguous** ***vmalloc()***.

- Because of these concerns, ***vmalloc()*** is used only when absolutely necessary, to obtain large regions of memory, i.e.
when **modules are dynamically inserted into the kernel *insmod***, they are loaded into memory created via ***vmalloc()***.

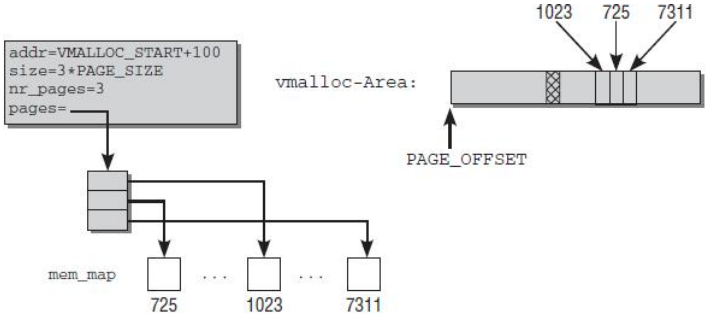
```
char *buf;
buf = vmalloc(16 * PAGE_SIZE); /* get 16 pages */
if (!buf)
/* error! failed to allocate memory */
...
vfree(buf);
```



vmalloc, *Cont'd*

- When it manages the **vmalloc** area in virtual memory, the kernel must keep track of which area allocated with **vmalloc.**
- ***static struct vm_struct *vmlist __initdata;*** List of vmalloc sections created in the system.
- ***vm_struct ,***
  - ***addr*** defines the start address of the allocated area in virtual address space; ***size*** indicates the size of the area.
  - ***pages*** is a pointer to an array of page pointers. Each element represents the page instance of a physical page mapped into virtual address space.
  - ***nr_pages*** specifies the number of entries in pages which is the number of memory pages involved.
  - ***phys_addr*** is required only if physical memory areas described by a physical address are mapped with ***ioremap*** (mapping for I/O devices area to kernel virtual address).
  - i.e. 3 physical pages positions in RAM are 1,023, 725 and 7,311 are mapped one after the other

in the virtual vmalloc area, the kernel sees them as a contiguous memory area starting at the VMALLOC_START + 100.




```
struct vm_struct {
    struct vm_struct    *next;
    void               *addr;
    unsigned long       size;
    unsigned long       flags;
    struct page        **pages;
#ifdef CONFIG_HAVE_ARCH_HUGE_VMALLOC
    unsigned int        page_order;
#endif
    unsigned int        nr_pages;
    phys_addr_t        phys_addr;
    const void         *caller;
};
```

Contiguous Memory Allocator (CMA)

- Despite of virtually mapped allocations solve the problem of large memory allocations to a greater extent. However, there are a few scenarios that mandate the allocation of physically contiguous buffers.
    - **DMA transfers**.
    - **Device drivers** : drivers dealing with specific classes of devices such as multimedia often find themselves searching for huge blocks of contiguous memory.

- To meet this, A contiguous Memory Allocator (**CMA**) is a kernel mechanism introduced to manage ***reserved*** memories.

- **CMA** mechanism **reserve** some memories under the allocator algorithm, and such memory is referred to as **CMA** area.

```
enum migratetype {
    MIGRATE_UNMOVABLE,
    MIGRATE_MOVABLE,
    MIGRATE_RECLAIMABLE,
    MIGRATE_PCPTYPES,
    MIGRATE_HIGHATOMIC = MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
    MIGRATE_CMA,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
    MIGRATE_ISOLATE,
/* can't allocate from here */
#endif
    MIGRATE_TYPES
};
```

- **CMA** allows allocations for both devices' and system's use, this is achieved by building a page descriptor list for pages in **reserve** memory, and enumerating it into the buddy system, which enables allocation of CMA pages through the page allocator for regular needs, kernel subsystems and through ***DMA*** allocation routines for device drivers.

- Pages enumerated by CMA into buddy system are assigned the ***MIGRATE_CMA*** property, which indicates that pages are ***MOVALBE***.

- When for example a DMA allocation request arrives, CMA pages held by kernel allocations are moved out of the **reserved** region (through a page-migration mechanism), resulting in the availability of memory for the device driver's use.

- In case of DMA, when pages are allocated, their ***migratetype*** is changed from ***MIGRATE_CMA*** to ***MIGRATE_ISOLATE***, making them invisible to the buddy system.

Slab Layer

- Also called the **slab allocator**, allocating and freeing data structures "**Objects**" for the common operations inside the kernel.

- To facilitate frequent allocations and deallocations of data, programmers often introduce like **free lists**, that contains a block of available, already allocated, data structures "Objets".

- In this sense, the free list acts as an object cache, caching a frequently used ***type*** of object.

- So, the slab layer acts as a generic data **structure-caching** layer.

- Slab Layer Design
  - The Linux slab allocator has evolved, there have been 3 different implementations.

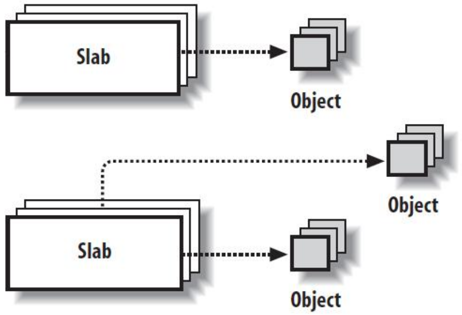
  - **SLOB Allocator**: Was the original slab allocator as implemented in Solaris OS. Now used for embedded systems where memory is scarce, performs well when allocating very small chunks of memory. Based on the first-fit allocation algorithm.

- **SLAB Allocator**: An improvement over the SLOB allocator, aims to be very "cache-friendly" , **Slab coloring.**

- **SLUB Allocator**: Has better execution time than the SLAB allocator by reducing the number of queues/chains used.

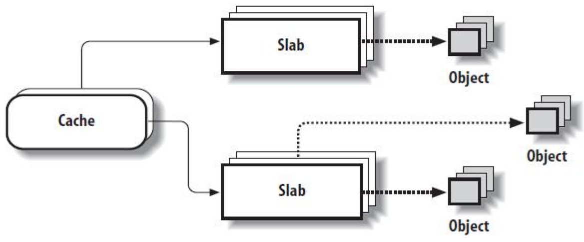
- Nowadays (on most distributions) the default Slab allocator is the **SLUB** allocator.

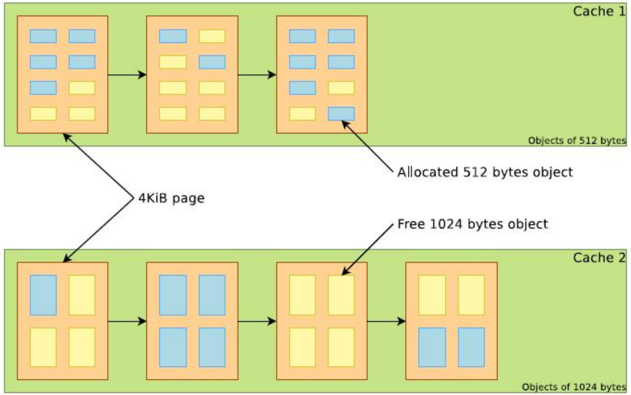
- Note: The generic "Slab Allocator" term can be used to refer to all three allocators.



Slab Layer, *Cont'd*

- Slab Layer Design, *Cont'd*
  - ***kmalloc()*** interface is built on top of the slab layer, using a family of general purpose caches.
  - A slab cache contains multiple slabs which in turn, contain multiple objects.
  - The slab allocator provides two main classes of caches:
    - **Dedicated** : These are caches that are created in the kernel for commonly used objects (e.g.,***task_struct, inode, mm_struct, vm_area_struct,...***) .
    - **Generic** (size-N and size-N(DMA)) : These are general purpose caches, which in most cases are of sizes corresponding to powers of two, i.e. ***kmalloc().***
- **SLAB** Cache Management
  - The 4 main **structures** that are used to manage caches in the SLAB allocator are: ***kmem_cache{}, kmem_cache_node{}, array_cache{} , slab{}.***






```
struct kmem_cache {
    struct array_cache __percpu *cpu_cache;
    ...
    struct kmem_cache_node *node[MAX_NUMNODES];
};
```

Slab Layer, *Cont'd*

- **SLAB** Cache Management, *Cont'd*
  - ***kmem_cache*** : ***kmem_cache_create()*** used to create a cache and allocates objects in the cache.
    gfporder : defines the order (2^n) of pages per slab. This is used by the slab allocator to request memory from the **buddy** allocator.

    int object_size : the size of the object without metadata.
    unsigned int num : number of objs per slab.

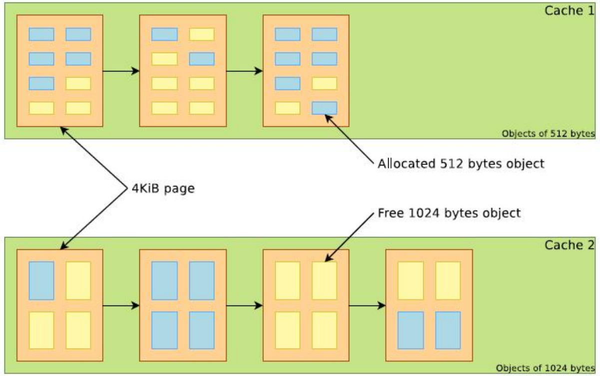
    Each slab hosts a page or a group of page frames.

    struct array_cache *cpu_cache : Used to reduce the number of linked list traversals/operations.
    It has **LIFO** ordering to take advantage of cache hotness and aims to hand out "cache warm" objects, ***entry[]*** of this structure holds an array of recently freed pointers (i.e., free'd objects which are cache hot).

    struct kmem_cache_node *node[] : ***kmem_cache_node*** keeps 3 **linked lists**, each list holds the slabs that are <u>partially filled</u>, the slabs which are <u>full</u> and the slabs that are <u>free</u> and can be allocated.

/include/linux/slab_def.h

```
struct kmem_cache {
    struct array_cache __percpu *cpu_cache;
    ...
    unsigned int size;

    unsigned int num;       /* # of objs per slab */

    /* order of pgs per slab (2^n) */
    unsigned int gfporder;
    ...
    struct kmem_cache_node *node[MAX_NUMNODES];
};
```




```
/*
 * struct array_cache
 *
 * Purpose:
 * - LIFO ordering, to hand out cache-warm objects from _alloc
 * - reduce the number of linked list operations
 * - reduce spinlock operations
 *
 * The limit is stored in the per-cpu structure to reduce the data cache
 * footprint.
 *
 */
struct array_cache {
    unsigned int avail;
    unsigned int limit;
    unsigned int batchcount;
    unsigned int touched;
    void *entry[];
};
```

```
struct kmem_cache_node {
    ...
#ifdef CONFIG_SLAB
    struct list_head slabs_partial;
    struct list_head slabs_full;
    struct list_head slabs_free;
    ...
#endif
    ...
};
```

Slab Layer, *Cont'd*

- **SLAB** Cache Management, *Cont'd*
  - Older kernel versions relied on a separate struct ***slab_s*** to define a slab.Then, the slab management is kept into the ***struct page*** as an anonymous ***struct.*** but in later patches it has been removed from ***page*** and became isolated ***struct slab{}*** because of page folios approach **v5.17.**

  - ***struct slab{}:***
    void * s_mem : Points to the first object related to the slab.

    struct kmem_cache *slab_cache : Points to which cache, the slab belongs.

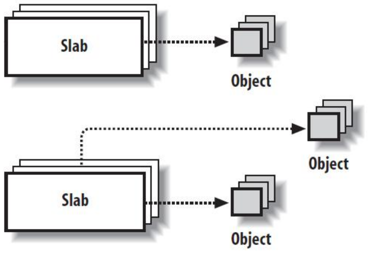
    struct list_head slab_list : Used to keep track of which slab list (partial/full/free) in the cache, the slab belongs.

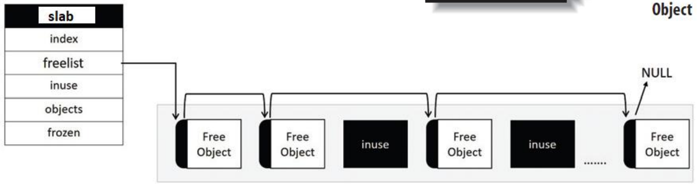
    void *freelist : All free objects in a slab form a linked list, the ***freelist*** pointer refers to the first free object in the list.

    Each free object is composed of a metadata area that contain a pointer to the next free object in the list.

```c
/* Reuses the bits in struct page */
struct slab {
    unsigned long __page_flags;

#if defined(CONFIG_SLAB)

    union {
        struct list_head slab_list;
        struct rcu_head rcu_head;
    };
    struct kmem_cache *slab_cache;
    void *freelist; /* array of free object indexes */
    void *s_mem;    /* first object */
    unsigned int active;

#elif defined(CONFIG_SLUB)

    union {
        struct list_head slab_list;
        struct rcu_head rcu_head;
#ifdef CONFIG_SLUB_CPU_PARTIAL
        struct {
            struct slab *next;
            int slabs;  /* Nr of slabs left */
        };
#endif
    };
    struct kmem_cache *slab_cache;
    /* Double-word boundary */
    void *freelist;     /* first free object */
    union {
        unsigned long counters;
        struct {
            unsigned inuse:16;
            unsigned objects:15;
            unsigned frozen:1;
        };
    };
    unsigned int __unused;

#elif defined(CONFIG_SLOB)

    struct list_head slab_list;
    void *__unused_1;
    void *freelist;     /* first free block */
    long units;
    unsigned int __unused_2;
    ...
```

```c
struct page {
    ...
    struct {    /* slab, slob and slub */
            union {
                struct list_head slab_list;
                struct {    /* Partial pages */
                    struct page *next;
#ifdef CONFIG_64BIT
                    int pages;  /* Nr of pages left */
                    int pobjects;   /* Approximate count */
#else
                    short int pages;
                    short int pobjects;
#endif
                };
            };
            struct kmem_cache *slab_cache; /* not slob */
            /* Double-word boundary */
            void *freelist;     /* first free object */
            union {
                void *s_mem;    /* slab: first object */
                unsigned long counters;     /* SLUB */
                struct {            /* SLUB */
                    unsigned inuse:16;
                    unsigned objects:15;
                    unsigned frozen:1;
                };
            };
        };
        ...
}
```





Slab Layer, *Cont'd*

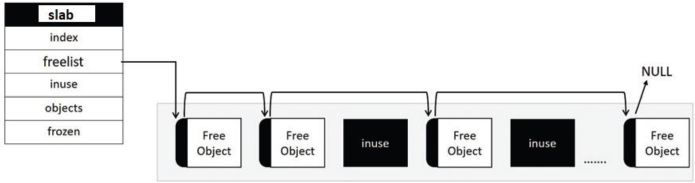
- **SLAB** Cache Management, *Cont'd*
  - ***struct slab{}**, Cont'd:*

    inuse : contains the total count of allocated objects.

    objects : contains the total number of objects.

    frozen : is a flag that is used as a page lock, if a page has been frozen by a CPU core, only that core can retrieve free objects from the page.

```c
/* Reuses the bits in struct page */
struct slab {
    unsigned long __page_flags;

#if defined(CONFIG_SLAB)

    union {
        struct list_head slab_list;
        struct rcu_head rcu_head;
    };
    struct kmem_cache *slab_cache;
    void *freelist; /* array of free object indexes */
    void *s_mem;    /* first object */
    unsigned int active;

#elif defined(CONFIG_SLUB)

    union {
        struct list_head slab_list;
        struct rcu_head rcu_head;
#ifdef CONFIG_SLUB_CPU_PARTIAL
        struct {
            struct slab *next;
            int slabs;  /* Nr of slabs left */
        };
#endif
    };
    struct kmem_cache *slab_cache;
    /* Double-word boundary */
    void *freelist;     /* first free object */
    union {
        unsigned long counters;
        struct {
            unsigned inuse:16;
            unsigned objects:15;
            unsigned frozen:1;
        };
    };
    unsigned int __unused;

#elif defined(CONFIG_SLOB)

    struct list_head slab_list;
    void *__unused_1;
    void *freelist;     /* first free block */
    long units;
    unsigned int __unused_2;
    ...
```



Slab Layer, *Cont'd*

- **SLUB** Cache Management
  - The SLUB allocator also uses 4 main structures to manage slab caches: ***kmem_cache{}, kmem_cache_node{}, kmem_cache_cpu{}, slab{}.***

  - ***kmem_cache_node*** :
    nr_partial, partial : Partial slabs need to be tracked, the SLUB allocator has no interest in tracking **full** slabs whose objects have all been allocated, or **empty** slabs whose objects are free.

    SLUB tracks partial slabs for each node through an array of pointers of type
    struct kmem_cache_node* node[MAX_NUMNODES]

/include/linux/slub_def.h

```
struct kmem_cache {
    struct kmem_cache_cpu __percpu *cpu_slab;
    ...
    struct list_head list;  /* List of slab caches */
    ...
    struct kmem_cache_node *node[MAX_NUMNODES];
};
```

```
struct kmem_cache_node {
    ...
#ifdef CONFIG_SLUB
    unsigned long nr_partial;
    struct list_head partial;
#ifdef CONFIG_SLUB_DEBUG
    atomic_long_t nr_slabs;
    atomic_long_t total_objects;
    struct list_head full;
#endif
#endif

};
```

```
struct kmem_cache_cpu {
    void **freelist;
/* Pointer to next available object */
    unsigned long tid;
/* Globally unique transaction id */
    struct page *page;
/* The slab from which we are allocating */
...
};
```

Slab Layer, *Cont'd*

- **SLUB** Cache Management, *Cont'd*
  - Memory Allocation

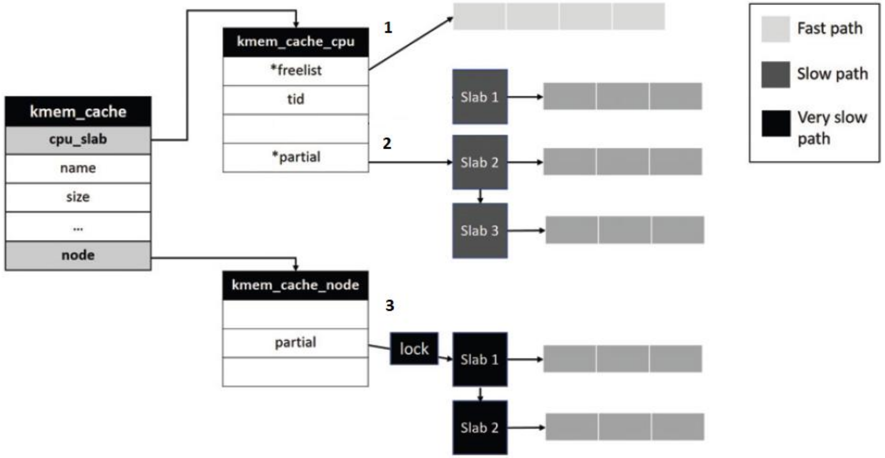
    When an allocation request arrives, the allocator takes the fast path and looks into the ***freelist*** of the per-CPU cache, and it then returns free objects.

    When the fast path fails, the allocator takes the slow path and looks through ***partial*** lists of the cpu cache sequentially.

    If no free objects are found, the allocator moves into the partial lists of nodes; this operation requires the allocator to contend for appropriate exclusion lock. On failure, the allocator gets a new slab from the **buddy system**.

    Acquiring a new slab from **buddy system** is considered very slow paths.

```
struct kmem_cache_cpu {
    void **freelist;    /* Pointer to next available object */
    unsigned long tid;  /* Globally unique transaction id */
    struct page *page;  /* The slab from which we are allocating */
#ifdef CONFIG_SLUB_CPU_PARTIAL
    struct page *partial;   /* Partially allocated frozen slabs */
#endif
#ifdef CONFIG_SLUB_STATS
    unsigned stat[NR_SLUB_STAT_ITEMS];
#endif
};
```



Slab Layer, *Cont'd*

- Acquiring a new slab from **buddy system**
  - Creating Caches : ***kmem_cache_create()*** must be invoked to create a new slab cache.
  - ***size*** : The size of objects to be created in this cache.
  - ***align*** : offset of the first object within a slab, to ensure a particular alignment within the page, normally, zero is sufficient.
  - ***flags*** : specifies optional settings controlling the cache's behavior.
  - ***flags*** : i.e.
    SLAB_HWCACHE_ALIGN : to align each object within a slab to a cache line.

    SLAB_RED_ZONE : to insert "red zones" around the allocated memory to help detect buffer overruns.

    SLAB_PANIC : causes the slab layer to panic if the allocation fails.

```
struct kmem_cache *
kmem_cache_create(const char *name, unsigned int
size, unsigned int align,
        slab_flags_t flags, void (*ctor)(void *))
{
    return kmem_cache_create_usercopy(name, size,
          align, flags, 0, 0, ctor);
}
EXPORT_SYMBOL(kmem_cache_create);
```

Buddy Allocator

- As we know, physical memory is broken up into large chunks of memory where each chunk is a "page order" (i.e., ***2^n * PAGE_SIZE***).

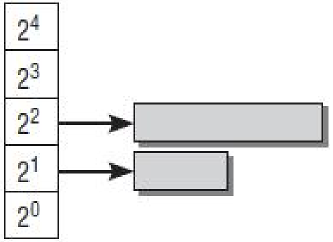
- Whenever a block of memory needs to be allocated and the size of it is not available, one big chunk is **halved** continuously, until a chunk of the correct size is found.

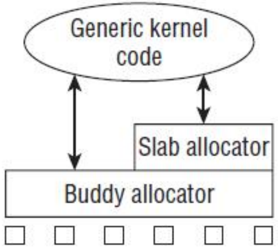
- These two halves are also known as **Buddies,** one will be used to satisfy the allocation request, and the other will remain free.

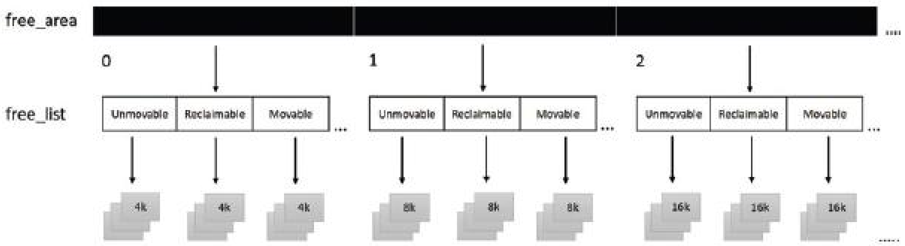
- At a later stage, if and when that memory is free'd, the two buddies (if both free) will **coalesce** forming a larger chunk of free memory.

- Linux uses a buddy allocator for each memory ***zone***, so the buddy allocator keeps track of free areas via an array of "queues" of type ***struct free_area*** which keeps track of the free chunks.

  It goes from 0 (min order) to 10 (MAX_ORDER - 1).
  2^0 , 2^1 , ...








```
struct zone {
    ...
    /* free areas of different sizes */
    struct free_area    free_area[MAX_ORDER];
    ...
}
```

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

Buddy Allocator, *Cont'd*

- ***MIGRATE_TYPES***
  Page migration is a process of moving data of a virtual page from one physical memory region to another.

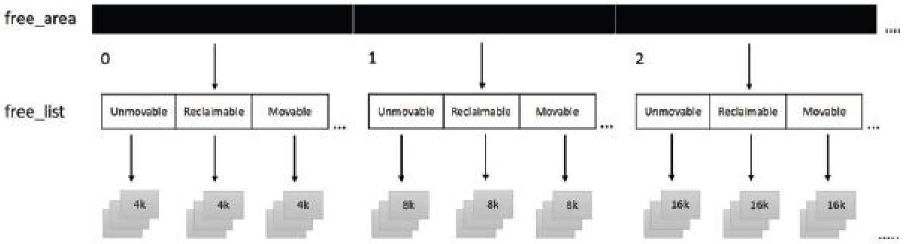
  - ***MIGRATE_UNMOVABLE*** : Physical pages which are pinned and reserved for a specific allocation are considered unmovable.
  - ***MIGRATE_MOVABLE*** : Physical pages that can be moved to different regions through page migration mechanism.
  - ***MIGRATE_RECLAIMABLE*** : The pages that can be freed at any time, so that they can be swapped out, to the disk.

- As we mentioned before, there will be a problem with the buddy algorithm "**Internal fragmentation**"

```
enum migratetype {
    MIGRATE_UNMOVABLE,
    MIGRATE_MOVABLE,
    MIGRATE_RECLAIMABLE,
    MIGRATE_PCPTYPES,   /* the number of
types on the pcp lists */
    MIGRATE_HIGHATOMIC =
MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
    MIGRATE_CMA,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
    MIGRATE_ISOLATE,    /* can't allocate
from here */
#endif
    MIGRATE_TYPES
};
```

```
struct zone {
    ...
    /* free areas of different sizes */
    struct free_area    free_area[MAX_ORDER];
    ...
}
```

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




```
static inline struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
void free_pages(unsigned long addr, unsigned int order);
```

Per-CPU Allocations

- What's **per-cpu**? SMP (Symmetric multiprocessors) use per-CPU data which is data that is unique to each processor.
- Why we need it?
    - Reduction locking requirements which might need when we have normal shared data, only we need this ***processor accesses this data.***
    - **per-CPU** data reduces cache invalidation, If one processor manipulates data held in another processor's cache, that processor must flush or otherwise update its cache.
- We can imagine per-cpu as follows

```
    unsigned long my_percpu[NR_CPUS];
    int cpu;
    cpu = get_cpu(); /* get current processor and
                        disable kernel preemption */
    my_percpu[cpu]++; /* ... or whatever */
    printk("my_percpu on cpu=%d is %lu\n", cpu,
my_percpu[cpu]);
    put_cpu(); /* enable kernel preemption */
```

- No lock is required because this data is unique to the current processor.
- Because the call ***get_cpu()***, on top of returning the current processor number, also disables kernel preemption, there is no fear of the valid of the data accessed.

```
#define get_cpu()       ({ preempt_disable(); __smp_processor_id(); })
#define put_cpu()       preempt_enable()
```

Per-CPU Allocations, *Cont'd*

- Per-CPU Data Allocation
  - Compile-Time : Creates an instance of a variable of type ***type***, named ***name***, for each processor on the system. We can manipulate it as follows,

```
#define DEFINE_PER_CPU(type, name)              \
    DEFINE_PER_CPU_SECTION(type, name, "")
```

```
get_cpu_var(name)++; /* increment
name on this processor */
put_cpu_var(name); /* done; enable
kernel preemption */
```

```
#define get_cpu_var(var)      \
(*({                          \
    preempt_disable();        \
    this_cpu_ptr(&var);       \
}))
```

  - Runtime : The kernel implements a dynamic allocator, similar to ***kmalloc()***, for creating ***per-CPU*** data.

```
#define alloc_percpu(type)          \
    (typeof(type) __percpu *)__alloc_percpu(sizeof(type),   \
                        __alignof__(type))
```

```
void free_percpu(void __percpu *ptr)
{

}
```