Unit 2: Memory Management

This slice set is based on Prof. Shih-Kun Huang's material and the book "Understanding the Linux Kernel"

Memory Allocation

- Allocate memory in user space
  - malloc()

- Allocate memory in kernel space
  - alloc_pages()
  - kmalloc()
  - vmalloc()
  - Memory allocation in the kernel is not harder than in the user space. It’s just different

Kernel Memory Management

- Contiguous Memory Area
  - Page Frame Management
    - Fixed size page (say 4k)
    - Zone (for NUMA or UMA)
      - DMA, Multi-cores
    - buddy system
    - __get_free_pages(), alloc_pages()
  - Memory Area Management
    - Small area
    - Slab allocator
    - kmem_cache_alloc(), kmalloc()
- Noncontiguous Memory Area
  - vmalloc() or vmalloc32()

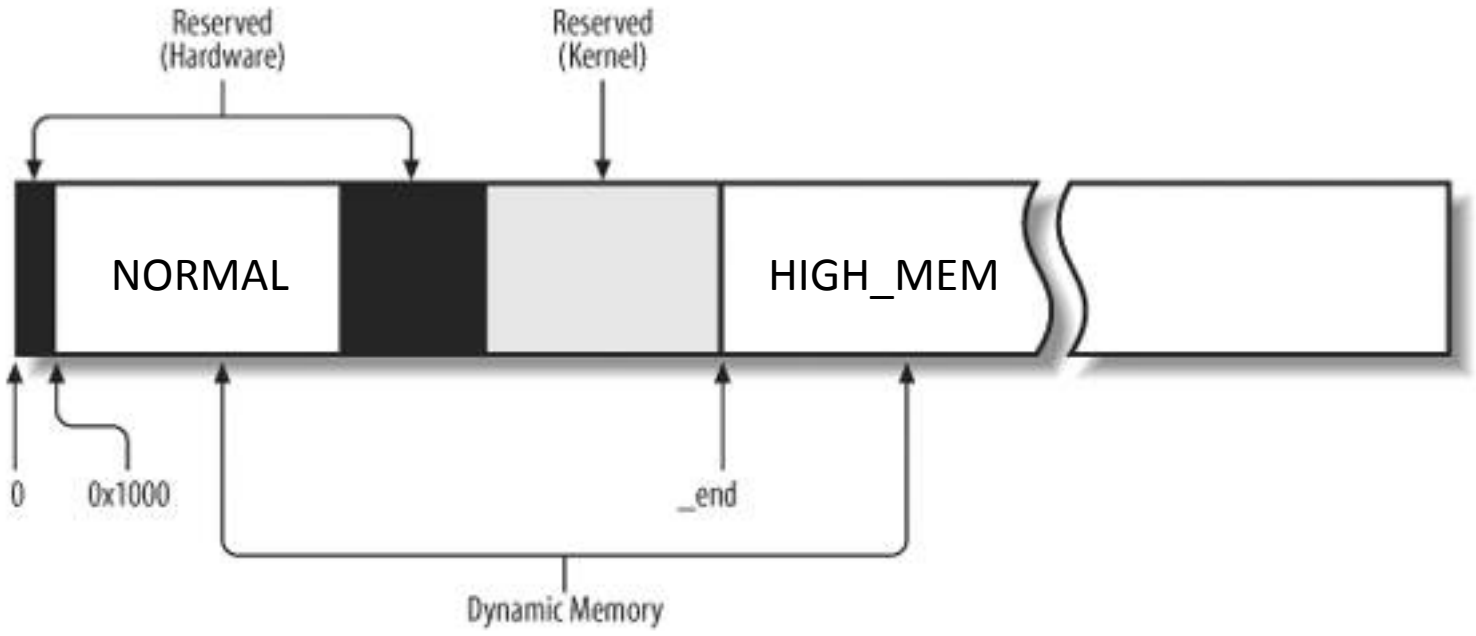
Page Frame Management

Page Frame Management

- Divide the Dynamic Memory into page frames
  - 4 KB or 4 MB for processes
- Page descriptors
  - struct *page*, 32 byts
  - The status/meta data of each page frame
    - ~1% space overhead for 4KB pages
  - All page descriptors are in a static array mem_map
    - One for each node in NUMA

```
3911        if (pgdat == NODE_DATA(0)) {
3912                mem_map = NODE_DATA(0)->node_mem_map;
```

Page Frame Management



Page Descriptors

- struct page
  - State information of a page frame
- struct page *mem_map
  - All page descriptors
- virt_to_page(addr) macro
  - the address of the page descriptor associated with the linear address addr
- pfn_to_page(pfn) macro
  - the address of the page descriptor associated with the page frame having number pfn

```
82 #define virt_to_page(addr)     (mem_map + (((unsigned long)(addr)-PAGE_OFFSET) >> PAGE_SHIFT))
83 #define page_to_virt(page)     ((((page) - mem_map) << PAGE_SHIFT) + PAGE_OFFSET)
```

Page Descriptors

- kernel keep track of the current status of each page frame.
  - Distinguish the page frames that are used to contain pages that belong to processes from those that contain kernel code or kernel data structures
  - Determine whether a page frame in dynamic memory is free

Page Descriptors: struct page

Table 8-1. The fields of the page descriptor

Type	Name	Description
unsigned long	flags	Array of flags (see Table 8-2). Also encodes the zone number to which the page frame belongs.
atomic_t	_count	Page frame's reference counter.
atomic_t	_mapcount	Number of Page Table entries that refer to the page frame (-1 if none).
unsigned long	private	Available to the kernel component that is using the page (for instance, it is a buffer head pointer in case of buffer page; see "Block Buffers and Buffer Heads" in Chapter 15). If the page is free, this field is used by the buddy system (see later in this chapter).
struct address_space *	mapping	Used when the page is inserted into the page cache (see the section "The Page Cache" in Chapter 15), or when it belongs to an anonymous region (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
unsigned long	index	Used by several kernel components with different meanings. For instance, it identifies the position of the data stored in the page frame within the page's disk image or within an anonymous region (Chapter 15), or it stores a swapped-out page identifier (Chapter 17).
struct list_head	lru	Contains pointers to the least recently used doubly linked list of pages.

Page Descriptors

- _count
  - -1: the page frame is free
  - >=0: assigned to one or more processes or store some kernel data structures
  - page_count( ) returns _count+1
    - # of users (e.g., processes) of this page
- _mapcount
  - # of page table entries mapped to the page frame
  - != _count if some processes have paged out the page frame

Page Descriptors

- mapping
  - Corresponds to the address space of
    - The swap device if the page is anonymous
    - A disk file if the page is mapped
- Index
  - Correspond to the page offset relative to
    - The swap device if the page is anonymous
    - A disk file if the page is mapped

Page Descriptors

- Flags
  - 32 flags
  - page frame status
  - Almost all bits are used now
- PageXyz macro
  - Retrieve flag value
- SetPageXyz and ClearPageXyz macro
  - Set and clear the corresponding bit

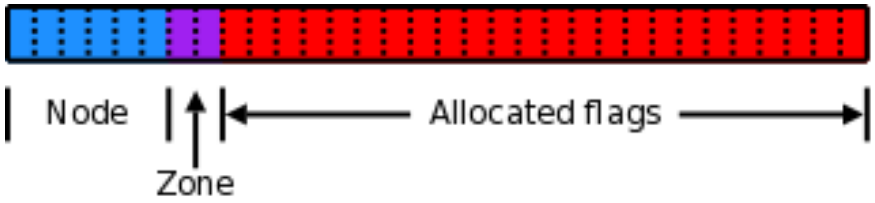


Table 8-2. Flags describing the status of a page frame

Flag name	Meaning
PG_locked	The page is locked; for instance, it is involved in a disk I/O operation.
PG_error	An I/O error occurred while transferring the page.
PG_referenced	The page has been recently accessed.
PG_uptodate	This flag is set after completing a read operation, unless a disk I/O error happened.
PG_dirty	The page has been modified (see the section "Implementing the PFRA" in Chapter 17).
PG_lru	The page is in the active or inactive page list (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).
PG_active	The page is in the active page list (see the section "The Least Recently Used (LRU) Lists" in Chapter 17).
PG_slab	The page frame is included in a slab (see the section "Memory Area Management" later in this chapter).
PG_highmem	The page frame belongs to the ZONE_HIGHMEM zone (see the following section "Non-Uniform Memory Access (NUMA)").
PG_checked	Used by some filesystems such as Ext2 and Ext3 (see Chapter 18).
PG_arch_1	Not used on the 80×86 architecture.
PG_reserved	The page frame is reserved for kernel code or is unusable.
PG_private	The private field of the page descriptor stores meaningful data.
PG_writeback	The page is being written to disk by means of the writepage method (see Chapter 16) .
PG_nosave	Used for system suspend/resume.
PG_compound	The page frame is handled through the extended paging mechanism (see the section "Extended Paging" in Chapter 2).
PG_swapcache	The page belongs to the swap cache (see the section "The Swap Cache" in Chapter 17).
PG_mappedtodisk	All data in the page frame corresponds to blocks allocated on disk.
PG_reclaim	The page has been marked to be written to disk in order to reclaim memory.
PG_nosave_free	Used for system suspend/resume.

Memory Zones

- It is ideal that all page frames can be equally used. However, real computers have constraints on the way page frames are used
- Linux kernel must deal with two hardware constraints of the 80x86 architecture:
  - Direct Memory Access (DMA) of old ISA buses can only address the first 16 MB of RAM
  - 32-bit CPUs cannot directly access all physical memory because the linear address space is too small

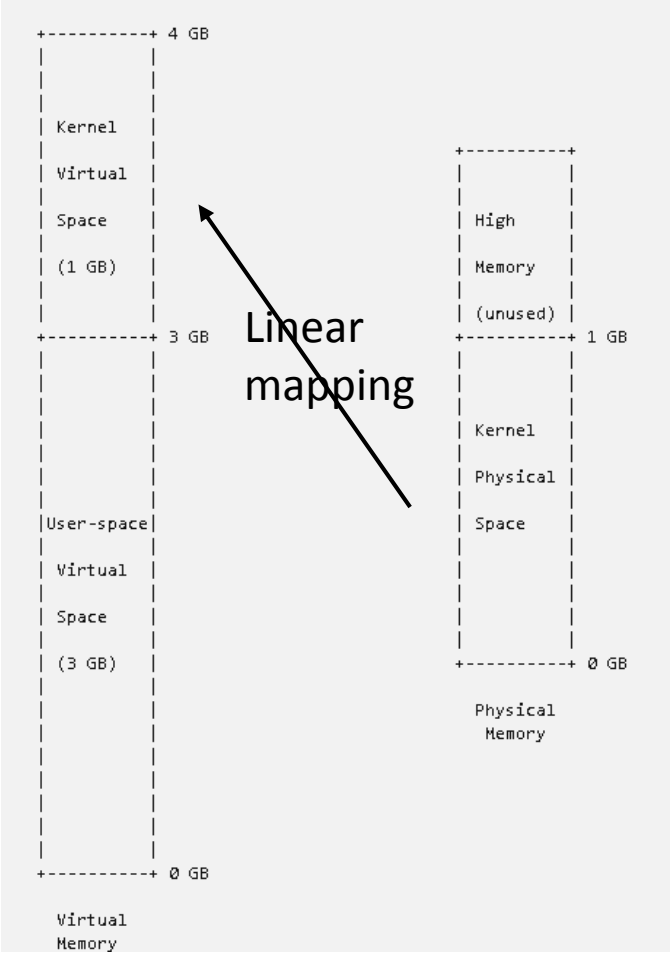
Memory Zones

- Linux 2.6 partitions the physical memory of every memory node into three zones
  - ZONE_DMA
    page frames of memory below 16 MB
  - ZONE_NORMAL
    page frames of memory at and above 16 MB and below 896 MB
  - ZONE_HIGHMEM
    page frames of memory at and above 896 MB

Memory Zones

- The ZONE_DMA zone includes page frames that can be used by old ISA-based devices by means of the DMA.
- The ZONE_DMA and ZONE_NORMAL zones include the "normal" page frames that can be directly accessed by the kernel through the linear mapping in the fourth gigabyte of the linear address space
- ZONE_HIGHMEM zone includes page frames that cannot be directly accessed by the kernel through the linear mapping in the fourth gigabyte of linear address space

Kernel Address Mapping



- The kernel uses linear mapping from the 1st GB of the physical memory to the 4th GB of the virtual memory
- Physical memory beyond the 1st GB (high memory) can be accessed by kernel only via kernel page mapping (i.e., kernel page table)

Table 8-4. The fields of the zone descriptor

Type	Name	Description
unsigned long	free_pages	Number of free pages in the zone.
unsigned long	pages_min	Number of reserved pages of the zone (see the section "The Pool of Reserved Page Frames" later in this chapter).
unsigned long	pages_low	Low watermark for page frame reclaiming; also used by the zone allocator as a threshold value (see the section "The Zone Allocator" later in this chapter).
unsigned long	pages_high	High watermark for page frame reclaiming; also used by the zone allocator as a threshold value.
unsigned long []	lowmem_reserve	Specifies how many page frames in each zone must be reserved for handling low-on-memory critical situations.
struct per_cpu_pageset[]	pageset	Data structure used to implement special caches of single page frames (see the section "The Per-CPU Page Frame Cache" later in this chapter).
spinlock_t	lock	Spin lock protecting the descriptor.
struct free_area []	free_area	Identifies the blocks of free page frames in the zone (see the section "The Buddy System Algorithm" later in this chapter).
spinlock_t	lru_lock	Spin lock for the active and inactive lists.
struct list head	active_list	List of active pages in the zone (see Chapter 17).
struct list head	inactive_list	List of inactive pages in the zone (see Chapter 17).
unsigned long	nr_scan_active	Number of active pages to be scanned when reclaiming memory (see the section "Low On Memory Reclaiming" in Chapter 17).
unsigned long	nr_scan_inactive	Number of inactive pages to be scanned when reclaiming memory.
unsigned long	nr_active	Number of pages in the zone's active list.
unsigned long	nr_inactive	Number of pages in the zone's inactive list.
unsigned long	pages_scanned	Counter used when doing page frame reclaiming in the zone.
int	all_unreclaimable	Flag set when the zone is full of unreclaimable pages.
int	temp_priority	Temporary zone's priority (used when doing page frame reclaiming).
int	prev_priority	Zone's priority ranging between 12 and 0 (used by the page frame reclaiming algorithm, see the section "Low On Memory Reclaiming" in Chapter 17).

The Pool of Reserved Page Frames

- Memory allocation requests can be satisfied in two different ways. If enough free memory is available, the request can be satisfied immediately.
- Otherwise, some memory reclaiming must take place, and the kernel control path that made the request is blocked until additional memory has been freed

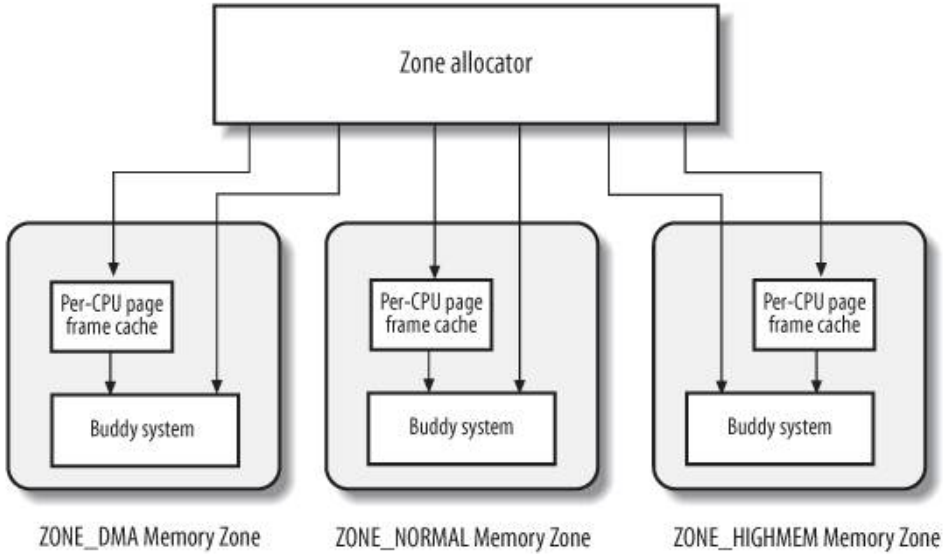
The Pool of Reserved Page Frames

- Some kernel control paths cannot be blocked while requesting memory this happens, for instance, when handling an interrupt or when executing code inside a critical region.
  - kernel using the GFP_ATOMIC flag
- The kernel reserves a pool of page frames for atomic memory allocation requests to be used only on low-on-memory conditions.

The Zoned Page Frame Allocator

- The kernel subsystem that handles the memory allocation requests for groups of (physically) contiguous page frames is called the zoned page frame allocator
- Zone allocator receives the requests for allocation and de-allocation of dynamic memory

Figure 8-2. Components of the zoned page frame allocator



Requesting and releasing page frames

- alloc_pages(gfp_mask, order)
  Macro used to request 2order contiguous page frames.
- alloc_page(gfp_mask)
  Macro used to get a single page frame;
- _ _get_free_pages(gfp_mask, order)
  Function that is similar to alloc_pages( ), but it returns the linear address of the first allocated page.

Requesting and releasing page frames

- _ _get_free_page(gfp_mask)
  Macro used to get a single page frame.
- get_zeroed_page(gfp_mask)
  Function used to obtain a page frame filled with zeros;
- _ _get_dma_pages(gfp_mask, order)
  Macro used to get page frames suitable for DMA.

Requesting and releasing page frames

Table 8-5. Flag used to request page frames

Flag	Description
__GFP_DMA	The page frame must belong to the ZONE_DMA memory zone. Equivalent to GFP_DMA.
__GFP_HIGHMEM	The page frame may belong to the ZONE_HIGHMEM memory zone.
__GFP_WAIT	The kernel is allowed to block the current process waiting for free page frames.
__GFP_HIGH	The kernel is allowed to access the pool of reserved page frames.
__GFP_IO	The kernel is allowed to perform I/O transfers on low memory pages in order to free page frames.
__GFP_FS	If clear, the kernel is not allowed to perform filesystem-dependent operations.
__GFP_COLD	The requested page frames may be "cold" (see the later section "The Per-CPU Page Frame Cache").
__GFP_NOWARN	A memory allocation failure will not produce a warning message.
__GFP_REPEAT	The kernel keeps retrying the memory allocation until it succeeds.
__GFP_NOFAIL	Same as __GFP_REPEAT.
__GFP_NORETRY	Do not retry a failed memory allocation.
__GFP_NO_GROW	The slab allocator does not allow a slab cache to be enlarged (see the later section "The Slab Allocator").
__GFP_COMP	The page frame belongs to an extended page (see the section "Extended Paging" in Chapter 2).
__GFP_ZERO	The page frame returned, if any, must be filled with zeros.

Requesting and releasing page frames

Table 8-6. Groups of flag values used to request page frames

Group name	Corresponding flags
GFP_ATOMIC	__GFP_HIGH
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

Zone Preference Order

- If __GFP_DMA is set, then free pages can only be allocated rom ZONE_DMA
- If __GFP_HIGHMEM is not set, then free pages are allocated from ZONE_NORMAL and ZONE_DMA, in the order of preference
  - Otherwise, allocated from ZONE_HIGHMEN, ZONE_NORMAL, and ZONE_DMA
- Pages from ZONE_HIGHMEM are not accessible unless kernel mapping is properly set
  - Those from ZONE_DMA and ZONE_NORAML are directly accessible via linear mapping

Kernel Mappings of High-Memory Page Frames

- Pages use linear mapping
  - Pages from normal and DMA zones
  - __va((unsigned long)(page - mem_map) << 12) for virtual address

- Pages require kernel mapping
  - Pages from high memory
  - Non-contiguous page frames (via vmalloc())

Kernel Mappings of High-Memory Page Frames

- Pages higher than 896MB are not linearly mapped to the 4th GB of virtual memory. To obtain the virtual address high-memory page frames:
  - __get_free_pages(GFP_HIGHMEM,0) returns a page descriptor
  - kmap(page) or kmap_atomic(page) obtains the virtual address of the page

Kernel Mappings of High-Memory Page Frames

- To establish kernel mapping of high-memory pages
  - The kernel uses page_address_htable hash table to map *page descriptors* of the mapped pages to their *virtual addresses*
  - The kernel uses Page Tables in the master kernel page tables to map *virtual addresses* of the mapped pages to their *physical addresses*

Kernel Mappings of High-Memory Page Frames

- Permanent kernel mapping
  - kmap()
  - Long lasting mapping
  - May block, cannot be used in ISR
  - Find a free slot in the kernel page table
    - If none can be found, sleep until other processes release some

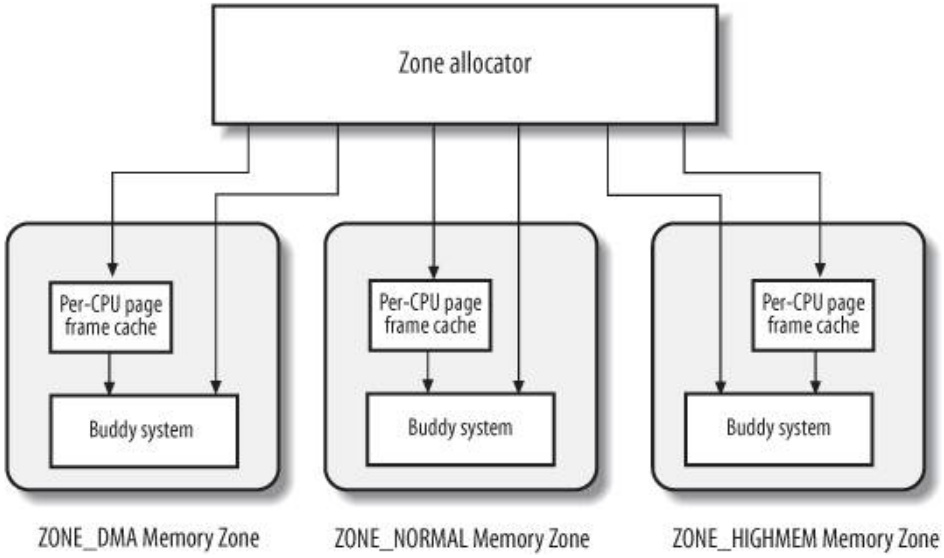
Kernel Mappings of High-Memory Page Frames

- Temporary kernel mapping
  - kmap_atomic()
  - Fast, do not sleep
  - Can be used in ISR
  - Find an free slot from a predefined set of page table slots
    - If none available, return fail

The Buddy System

- Implements the alloc_page() and related routines
- Allocates physically contiguous page frames

Figure 8-2. Components of the zoned page frame allocator



The Buddy System Algorithm

- There are essentially two ways to avoid external fragmentation:
  - Use the paging circuitry to map groups of noncontiguous free page frames into intervals of contiguous linear addresses.
  - Keep track of the existing blocks of free contiguous page frames, avoiding as much as possible the need to split up a large free block to satisfy a request for a smaller one.

The Buddy System Algorithm

- Three good reasons of the second approach :
  - (Physically) contiguous page frames are really necessary, because contiguous linear addresses are not sufficient to satisfy the request.
  - Contiguous page frame allocation offers the big advantage of leaving the kernel paging tables unchanged.
    - frequent Page Table modifications lead to higher average memory access times, because they make the CPU flush the contents of the translation lookaside buffers.
  - Large chunks of contiguous physical memory can be accessed by the kernel through 4 MB pages. This reduces the translation lookaside buffers misses, thus significantly speeding up the average memory access time.

The well-known buddy system algorithm

- Assume there is a request for a group of 256 contiguous page frames .
- If a free block in the 256-page-frame list exists.
- If there is no such block, looks for the next larger blocka free block.
- If such a block, allocates 256 of the 512 page frames to satisfy the request, inserts the remaining 256 page frames into the list of free 256-page-frame blocks.
- If there is no such block, then looks for the next larger block (i.e., a free 1024-page-frame block).
- If such a block exists, allocates 256 of the 1024 page frames to satisfy the request, inserts the first 512 of the remaining 768 page frames into the list of free 512-page-frame blocks, and inserts the last 256 page frames into the list of free 256-page-frame blocks.

Refer to your undergraduate OS text book!

Releasing blocks of page frames

- The kernel attempts to merge pairs of free buddy blocks of size *b* together int[illegible] single block of size 2*b*.
- Two blocks a[illegible]
  - Both [illegible]ay *b*.
  - They a[illegible]onysical addresses.
  - The p[illegible]irst page frame of the fi[illegible]ple of 2 x *b* x 2^{12}.
- The algorith[illegible]s iterative; if it succeeds in merging released blocks, it doubles *b* and tries again so as to create even bigger blocks.

Refer to your undergraduate OS text book!

Data structures

- Linux 2.6 uses a different buddy system for each zone.
- There are 3 buddy systems:
  - The first handles the page frames suitable for ISA DMA.
  - The second handles the "normal" page frames.
  - The third handles the high-memory page frames.
- Each buddy system relies on the following main data structures :
  - The mem_map array. the zone_mem_map and size fields of the zone descriptor.
  - An array consisting of eleven elements of type free_area, one element for each group size. The array is stored in the free_area field of the zone descriptor.
    - Groups of 2^0~2^{10} pages in the buddy system

The Per-CPU Page Frame Cache

- Kernel often requests and releases single page frames
- To boost system performance, each memory zone defines a *per-CPU page frame cache*. Each per-CPU cache includes some pre-allocated page frames to be used for single memory requests issued by the local CPU.
- Two caches for each memory zone and for each CPU:
  - a *hot cache* , which stores page frames whose contents are likely to be included in the CPU's hardware cache, and a *cold cache* .
  - A cold cache, whose pages are unlikely in the hardware cache. Good candidates for DMA operations

The Per-CPU Page Frame Cache

- Difference between the per-CPU page frame cache and the pool of reserved pages
  - The former is closely related to the CPU cache
  - The latter is mainly for non-blocking page allocation

The Zone Allocator

- The *zone allocator* is the frontend of the kernel page frame allocator. This component must locate a memory zone that includes a number of free page frames large enough to satisfy the memory request.

The Zone Allocator

- Zone allocator satisfy several goals:
  - It should protect the *pool of reserved page*.
  - It should trigger the *page frame reclaiming algorithm* when memory is scarce and blocking the current process is allowed; once some page frames have been freed, the zone allocator will retry the allocation.
  - It should preserve the small, precious ZONE_DMA memory zone, if possible. For instance, the zone allocator should be somewhat reluctant to assign page frames in the ZONE_DMA memory zone if the request was for ZONE_NORMAL or ZONE_HIGHMEM page frames.

Memory Area Management

Memory Area Managements

- Byte-wise memory allocation
  - How are we going to deal with requests for small memory areas, say a few tens or hundreds of bytes?
- A new data structures that describe how small memory areas are allocated within the same page frame.
  - A new problem called *internal fragmentation*.

The Slab Allocator

- The type of data to be stored may affect how memory areas are allocated
  - E.g., allocate pages for DMA
  - Collect memory objects of the same type avoid frequent object initialization/cleanup
- The kernel functions tend to request memory areas of the same type (and size) repeatedly
  - E.g., file system inodes
  - A slab cache completely eliminate fragmentation by grouping objects of the same size for allocation/deallocation

The Slab Allocator

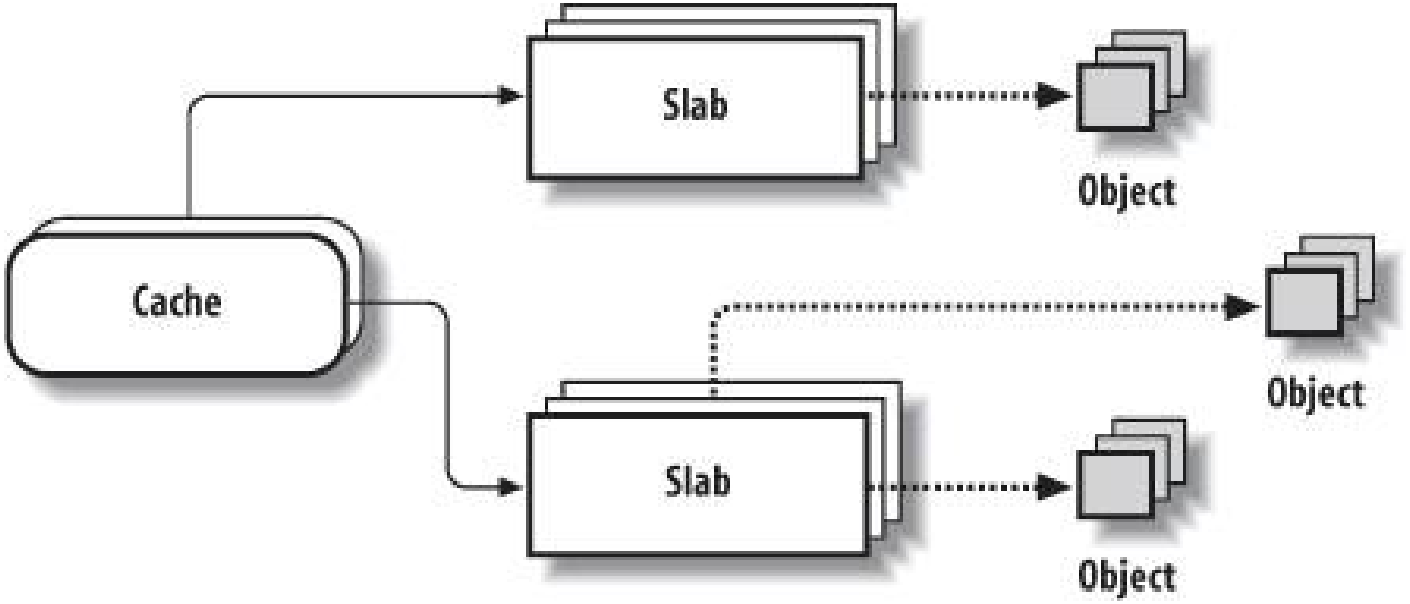
- Requests for memory areas can be classified according to their frequency
  - Usually memory object size is closely related to object allocation frequency
  - Harmonic allocation frequencies lowers the degree of memory fragmentation

The Slab Allocator

- The slab allocator groups objects into caches
  - Each cache is a "store" of objects of the same type
  - Objects of the same type are allocated from a number of physically contiguous page frames
- A slab cache
  - grows by allocating new page frames
  - shrinks by returning page frames containing no objects

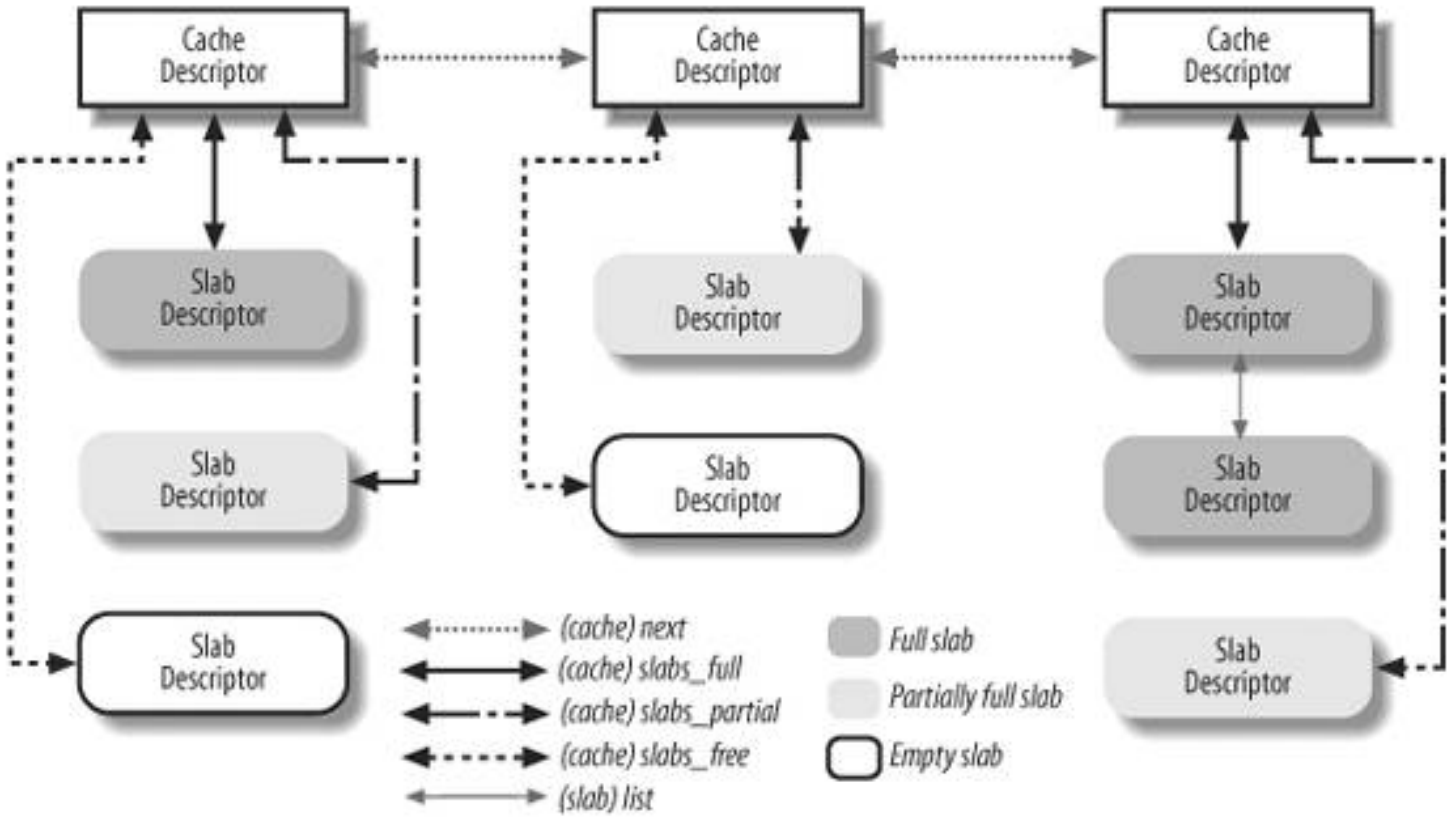
The Slab Allocator

Figure 8-3. The slab allocator components



Cache and Slab Descriptors

Figure 8-4. Relationship between cache and slab descriptors



Cache Descriptor

- Each cache is described by a structure of type kmem_cache_t.
  - Renamed to kmel_cache in 2.6.34

Slab Descriptor

- Each slab of a cache has its own descriptor of type slab.
- Slab descriptors can be stored in two possible places:
  - External slab descriptor
    Stored outside the slab, in one of the general caches not suitable for ISA DMA pointed to by cache_sizes.
  - Internal slab descriptor
    Stored inside the slab, at the beginning of the first page frame assigned to the slab.

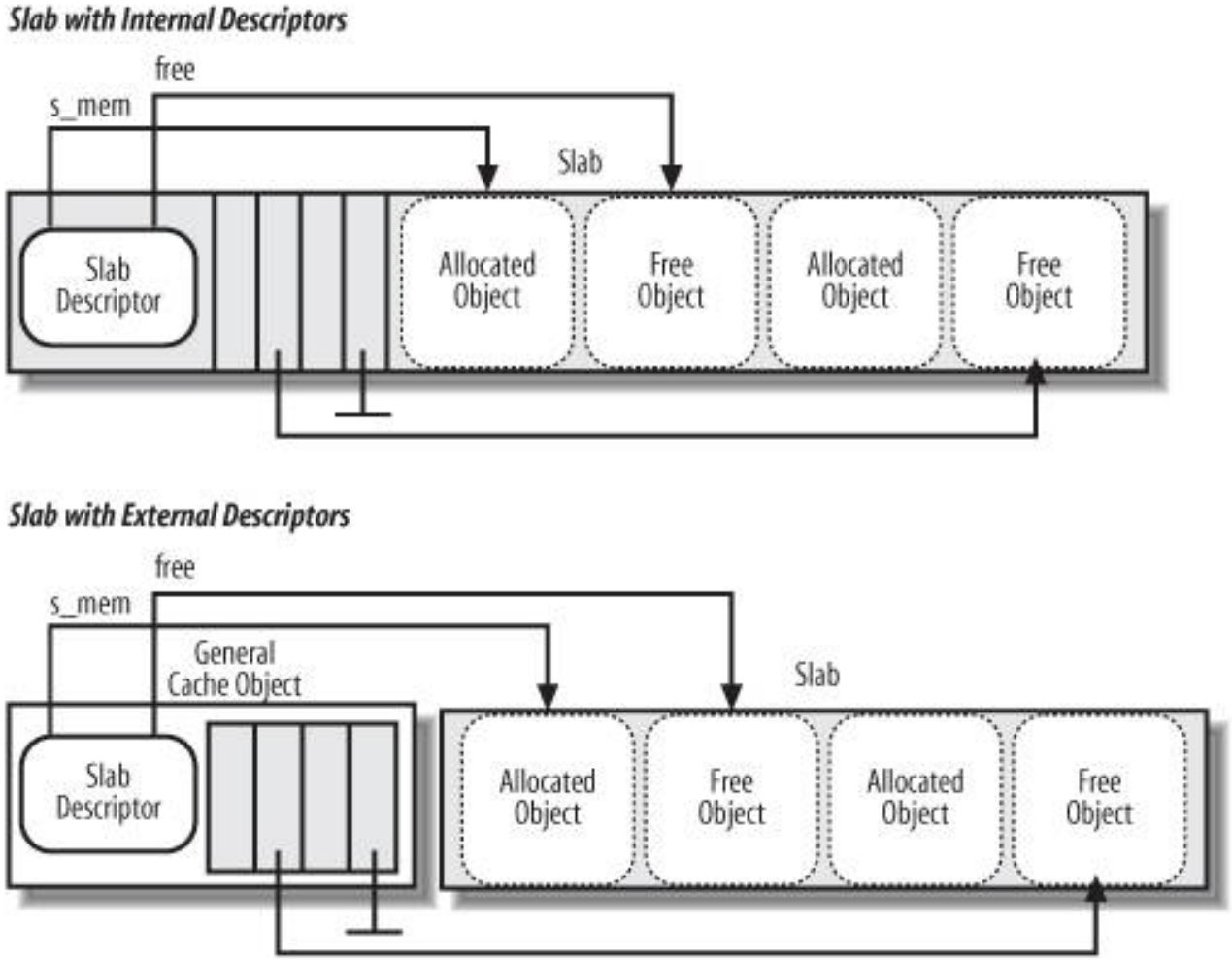
Object Descriptor

- Each object has a descriptor of type kmem_bufctl_t

- Object descriptors are stored in an array placed right after the corresponding slab descriptor
  - External object descriptors
    Stored outside of the slab, in the general cache pointed to by the slabp_cache field of the cache descriptor
  - Internal object descriptors
    Stored inside the slab, right before the objects they describe
- Generally, small objects/slabs use more descriptors
  - The kernel use internal descriptors

Object and Slab Descriptors

Figure 8-5. Relationships between slab and object descriptors



General and Specific Caches

- Caches are divided into two types:
  - *General caches*: the caches of geometrically distributed objects, ranging from 2^5 to 2^{17} bytes, for memory allocation via kmalloc()
  - *Specific caches* : the caches created for specific purposes, e.g., inode cache, dentry cache, etc
- All cache information can be obtained by reading /proc/slabinfo

Interfacing the Slab Allocator with the Zoned Page Frame Allocator

- When the slab allocator creates a new slab, it relies on the zoned page frame allocator to obtain a group of free contiguous page frames
  - It invokes the kmem_getpages( ) function, which, in turn, calls alloc_pages()

- We can say that slab caches are built on top of the zoned page frame allocator

Allocating a Slab to a Cache

- A newly created cache does not contain a slab and therefore does not contain any free objects. New slabs are assigned to a cache only when both of the following are true:
  - A request has been issued to allocate a new object.
  - The cache does not include a free object.

Releasing a Slab from a Cache

- There are too many free objects in the slab cache.
- A timer function invoked periodically determines that there are fully unused slabs that can be released

Aligning Objects in Memory

- The objects managed by the slab allocator are aligned in memory
  - Aligned objects can be accessed more efficiently, e.g., aligning DWORD to 4 bytes
- The objects stored in memory cells whose initial physical addresses are multiples of a given constant(alignment factor)
  - Power of 2, up to 4096
- Alignment is combined with slab coloring

Slab Coloring

- In a cache, objects that have the same offset within different slabs will, with a relatively high probability, end up mapped in the same cache line
  - E.g., the 10^{th} objects in the 1^{st} and the 2^{nd} slabs
  - Set-associative cache
- The cache hardware might therefore waste memory cycles transferring two objects from the same cache line back and forth to different RAM locations, while other cache lines go underutilized.

Slab Coloring

- The slab allocator tries to reduce this unpleasant cache behavior by a policy called slab coloring : different arbitrary values called colors are assigned to the slabs
- Idea: slabs of different colors have different "initial offset" for their first objects

Slab Coloring

- aln: alignment
- osize: object size
- dsize: descriptor size
- In the slab with color=0, if osize=32 and aln=4
  - Objects are at 0, 32, 64, ….
- In the slab with color=1,
  - Objects are at 0+4, 32+4, 64+4

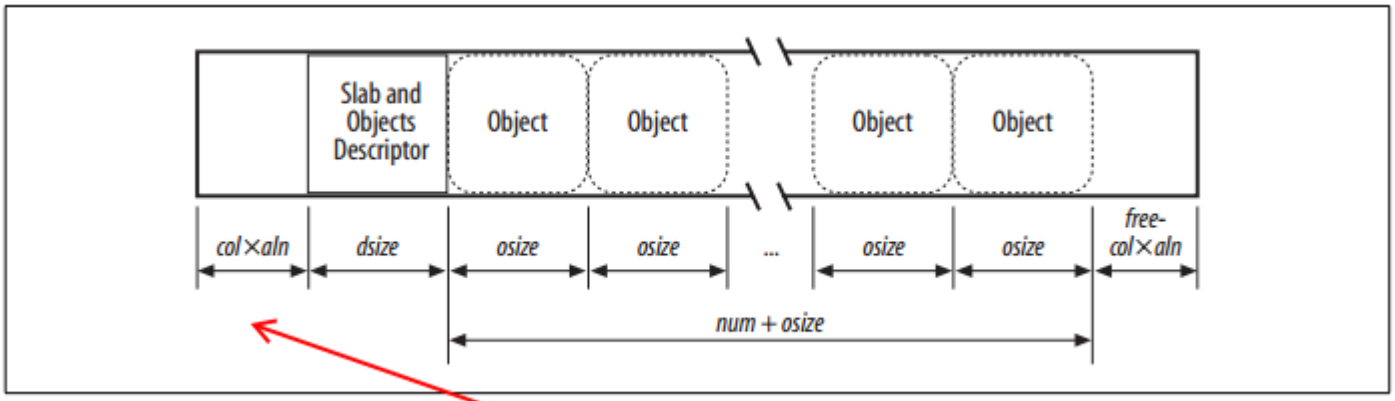


Figure 8-6. Slab with color col and alignment aln

Local Caches of Free Slab Objects

- To reduce spin lock contention among processors and to make better use of the hardware caches, each cache of the slab allocator includes a per-CPU data structure consisting of a small array of pointers to freed objects called the slab local cache .

Noncontiguous Memory Area Management

Noncontiguous Memory Area Management

- alloc_pages and kmalloc() obtains physically contiguous pages/memory area
  - Do not utilize the paging mechanism
  - Limited largest allocation size

- For memory objects without frequent allocation/deallocation, it make sense to allocate non-contiguous page frames and access them through linear virtual addresses

Linear Addresses of Noncontiguous Memory Areas

- vmalloc()

- Find available page frames (need not be contiguous)

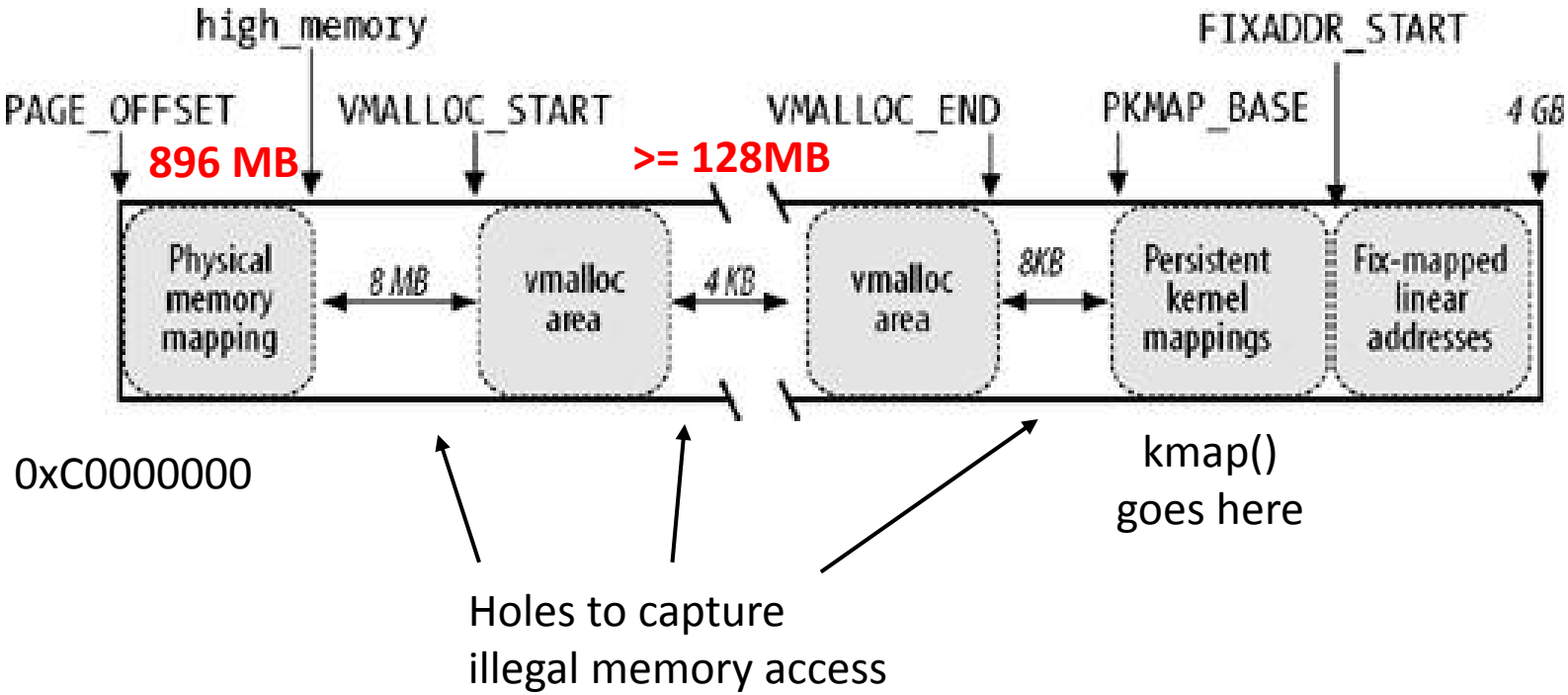
- Find a free range of linear addresses
  - we can look in the area starting from PAGE_OFFSET (usually 0xc0000000, the beginning of the fourth gigabyte).

Linear Addresses of Noncontiguous Memory Areas

- The beginning of the area includes the linear addresses that map the first 896 MB of RAM; the linear address that corresponds to the end of the directly mapped physical memory is stored in the high_memory variable.
- The end of the area contains the fix-mapped linear addresses .
- Starting from PKMAP_BASE we find the linear addresses used for the persistent kernel mapping of high-memory page.
- The remaining linear addresses can be used for noncontiguous memory areas. A safety interval of size 8 MB (macro VMALLOC_OFFSET) is inserted between the end of the physical memory mapping and the first memory area.

Linear Addresses of Noncontiguous Memory Areas

Figure 8-7. The linear address interval starting from PAGE_OFFSET



Descriptors of Noncontiguous Memory Areas

- Each noncontiguous memory area is associated with a descriptor of type vm_struct.

Table 8-13. The fields of the vm_struct descriptor

Type	Name	Description
`void *`	`addr`	Linear address of the first memory cell of the area
`unsigned long`	`size`	Size of the area plus 4,096 (inter-area safety interval)
`unsigned long`	`flags`	Type of memory mapped by the noncontiguous memory area
`struct page **`	`pages`	Pointer to array of `nr_pages` pointers to page descriptors
`unsigned int`	`nr_pages`	Number of pages filled by the area
`unsigned long`	`phys_addr`	Set to 0 unless the area has been created to map the I/O shared memory of a hardware device
`struct vm_struct *`	`next`	Pointer to next `vm_struct` structure

```
void * vmalloc(unsigned long size)
{
    struct vm_struct *area;
    struct page **pages;
    unsigned int array_size, i;
    size = (size + PAGE_SIZE - 1) & PAGE_MASK;
    area = get_vm_area(size, VM_ALLOC);
    if (!area)
        return NULL;
    area->nr_pages = size >> PAGE_SHIFT;
    array_size = (area->nr_pages * sizeof(struct page *));
    area->pages = pages = kmalloc(array_size, GFP_KERNEL);
    if (!area_pages) {
        remove_vm_area(area->addr);
        kfree(area);
        return NULL;
    }
    memset(area->pages, 0, array_size);
    for (i=0; i<area->nr_pages; i++) {
        area->pages[i] = alloc_page(GFP_KERNEL|__GFP_HIGHMEM);
        if (!area->pages[i]) {
            area->nr_pages = i;
    fail:   vfree(area->addr);
            return NULL;
       }
    }
    if (map_vm_area(area, __pgprot(0x63), &pages))
        goto fail;
    return area->addr;
}
```

Get a linear address range

Get an array of pointers to page descriptors

Allocate needed page frames, one by one

Map the page frames to the linear address range

Allocating a Memory Area for Noncontiguous Page Frames

- map_vm_area() only set up the master kernel page table but not the page tables of processes
  - When processes enter the kernel and access the linear address area, the memory access triggers page faults
    - The kernel then re-direct the corresponding page table pointers to the corresponding kernel page tables
  - In this way, vmalloc() needs not to update page tables of every process

Releasing a Noncontiguous Memory Area

- The vfree( ) function releases noncontiguous memory areas created by vmalloc( ) or vmalloc_32( ), while the vunmap( ) function releases memory areas created by vmap( )
- vunmap() clear the corresponding kernel page table entries. If any process access the memory released by vfree() then a page fault will be generated

Review

- alloc_pages() related functions
  - Return page descriptors, allocate in pages
  - Need to obtain virtual addresses via kernel mapping
  - May sleep. Can be atomic (no sleep).
- kmalloc()
  - Return virtual address
  - Max 2^{10} pages (HW dependent), physically contiguous
  - May sleep
- vmalloc()
  - Return virtual address
  - Can allocate very large memory, not physically contiguous
  - May sleep

End of Unit 2