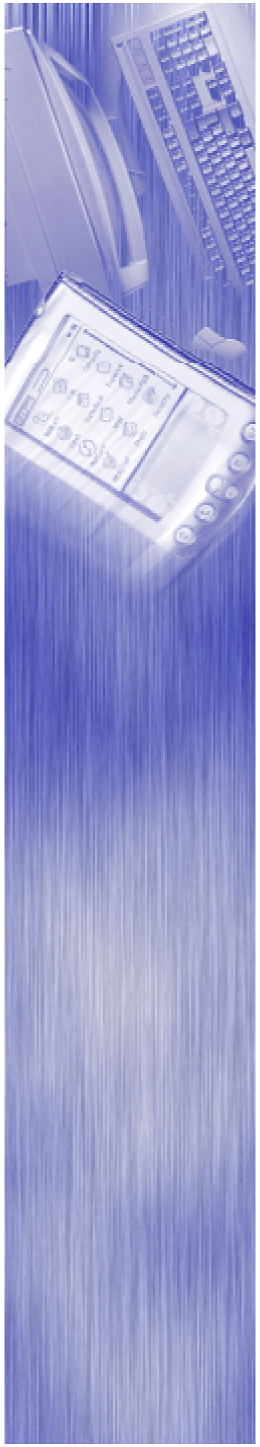
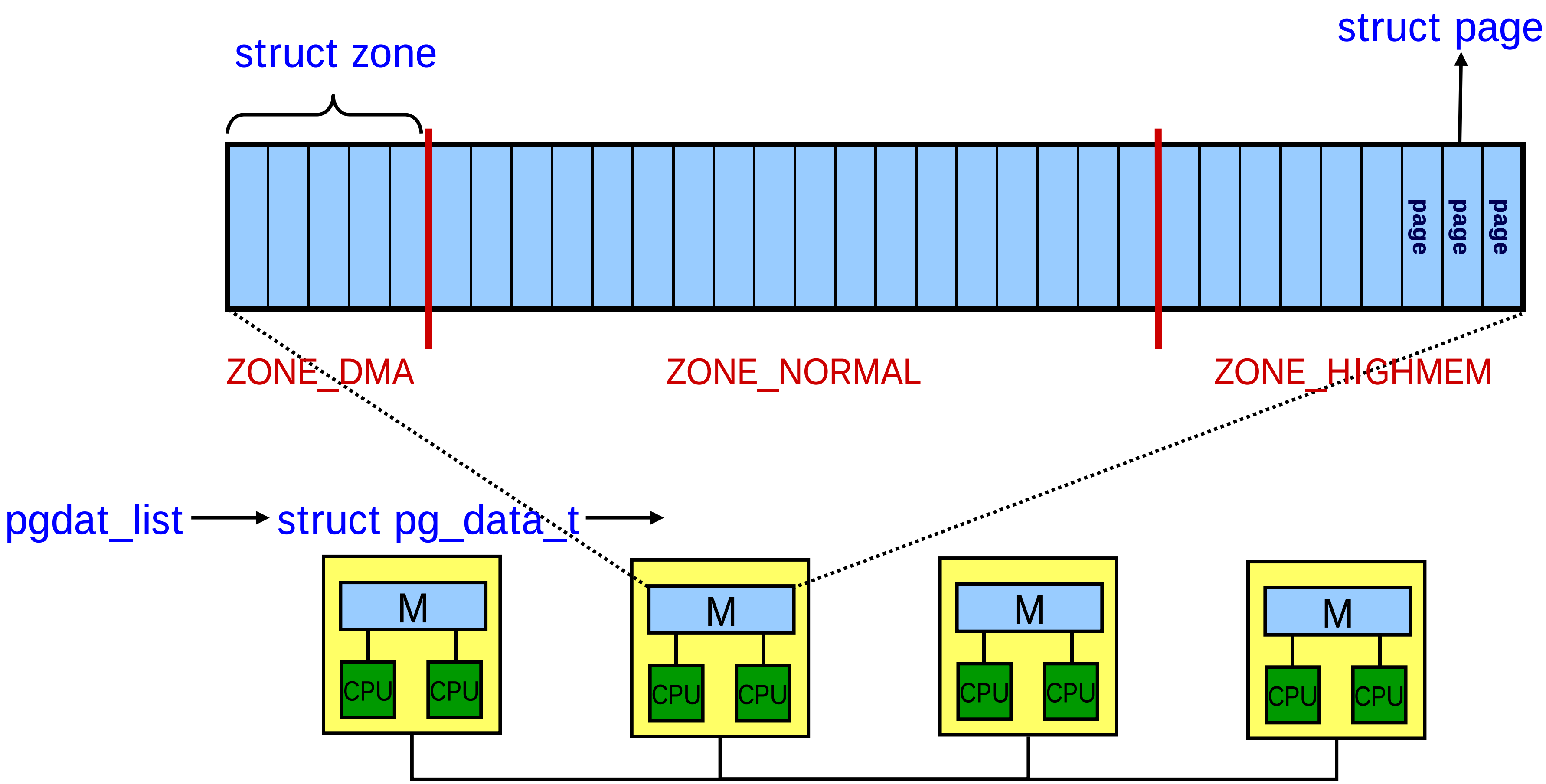
CS632/SEP564: Embedded Operating Systems (Fall 2008)

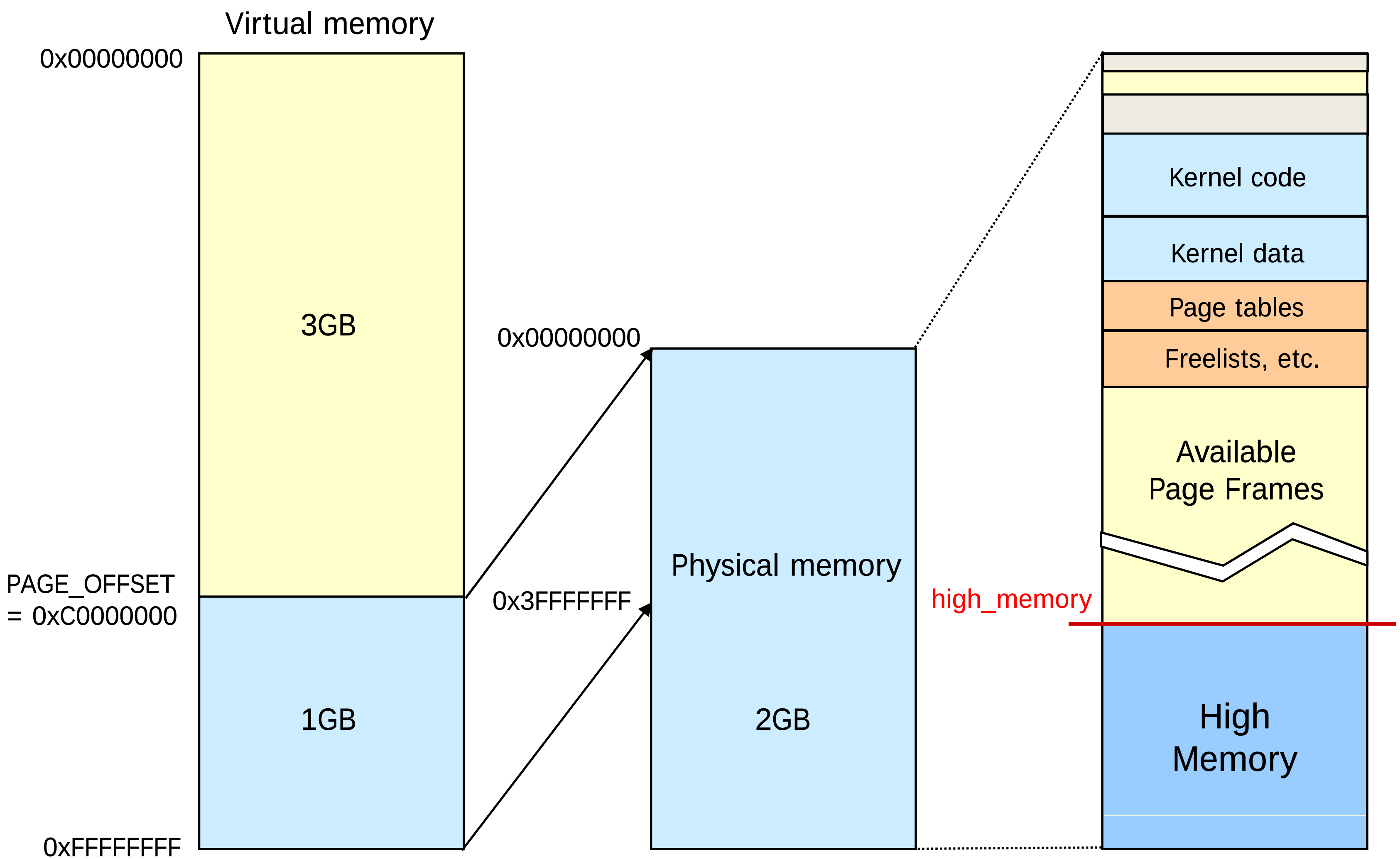


Memory Management

The Big Picture



Using Physical Memory



Pages

- **struct page** <linux/mm.h>
  - All page descriptors are stored in the mem_map array.
  - One entry for every physical page
    - – About 1MB for 128MB physical memory with 4KB page size

page_flags_t	flags;	32 flags defined in <linux/page-flags.h>
atomic_t	_count;	The number of references to this page Accessed via page_count().
atomic_t	_mapcount;	Count of PTEs mapped.
unsigned long	private;	Private data
struct address_space *	mapping;	How the page is used?
pgoff_t	index;	Offset within the mapping
struct list_head	lru;	LRU list

Zones (1)

- **Why memory zones?**
  - Some hardware devices are capable of performing DMA to only certain memory addresses.
  - Some architectures are capable of physically addressing larger amounts of memory than they can virtually address.
- **Memory zones in Linux**

ZONE_DMA	DMA-able pages	< 16MB
ZONE_NORMAL	Normally addressable pages	16MB – 896MB
ZONE_HIGHMEM	Dynamically addressable pages	> 896MB

Zones (2)

- **struct zone** <linux/mmzone.h>
  - Page frame reclamation is performed on a per zone basis.

spinlock_t	lock;	Spin lock protecting the descriptor
unsigned long	free_pages;	Number of free pages in the zone
unsigned long	pages_min, pages_low, pages_high;	Parameters for page frame reclaiming
struct list_head	active_list;	List of active pages in the zone
struct list_head	inactive_list;	List of inactive pages in the zone
struct pglist_data *	zone_pgdat;	Pointer to the node descriptor
struct page *	zone_mem_map;	Pointer to the first page descriptor
char *	name;	Name of the zone
struct free_area	free_area[];	Buddy map
...		

Zones (3)

▪ Reserved page frames

- Memory allocation requests can be satisfied immediately without memory reclaiming.
- Reduces the chance of failure in case of atomic memory allocation requests (GFP_ATOMIC).
- For ZONE_DMA and ZONE_NORMAL:
  - min_free_kbytes = sqrt(directly_mapped_memory * 16) (KB)
    - » 128KB ~ 64MB (512KB for 16MB, 8MB for 4GB)
  - zone→pages_min = the zone's contribution to min_free_kbytes / pagesize;
  - zone→pages_low = zone→pages_min * 5 / 4;
  - zone→pages_high = zone→pages_min * 3 / 2;

Nodes

- **pg_data_t** <linux/mmzone.h>
  - The physical memory is partitioned in several nodes.
  - All node descriptors are stored in a singly linked list, whose first element is pointed to by pgdat_list.

zone_t	node_zones[];	Zones for this node.
zonelist_t	node_zonelists[];	The order of zones that allocations are preferred from.
int	nr_zones;	The number of zones. Not all nodes will have the same number of zones.
struct page *	node_mem_map;	The first page of the struct page array
unsigned long	node_start_pfn;	Index of the first page frame in the node
pg_data_t *	pgdat_next;	Next item in the memory node list
int	node_id;	Node ID
...		

Zone Allocator (1)

- **Requesting page frames**
  - struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
  - struct page *alloc_page(gfp_t gfp_mask);
  - unsigned long __get_free_pages(gfp_t gfp_mask,
    unsigned int order);
  - unsigned long __get_free_page(gfp_t gfp_mask);
  - unsigned long get_zeroed_page(gfp_t gfp_mask);
  - unsigned long __get_dma_pages(gfp_t gfp_mask, int order);
    - = __get_free_pages((gfp_mask) | GFP_DMA, (order));

Zone Allocator (2)

- **Releasing page frames**
  - void free_pages(unsigned long addr, unsigned int order);
  - void __free_pages(struct page *page, unsigned int order);
  - void free_page(unsigned long addr);
  - void __free_page(struct page *page);

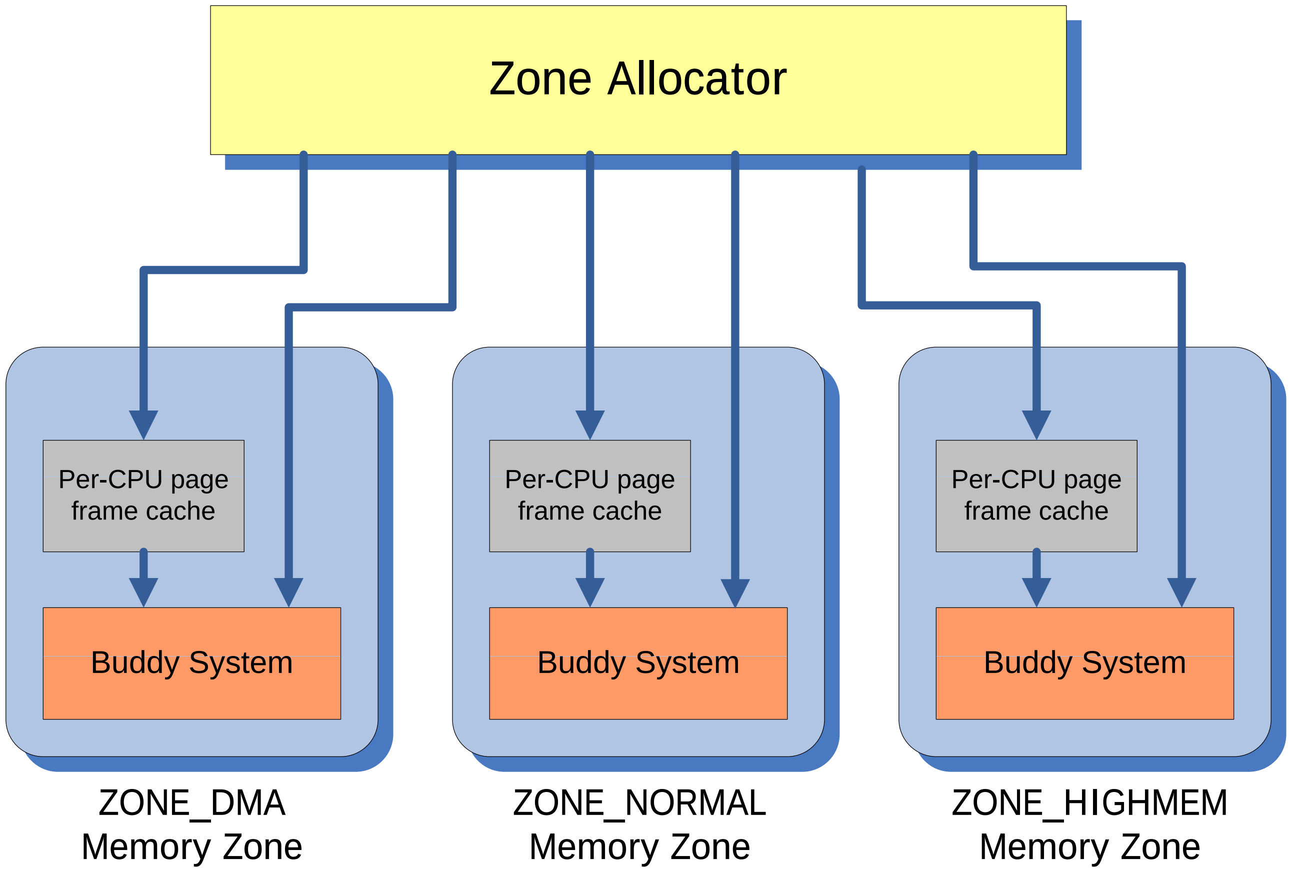
Zone Allocator (3)

- **Some flags used to request page frames**

Flag	Description
__GFP_DMA	The page frame must belong to the ZONE_DMA zone. Equivalent to GFP_DMA.
__GFP_HIGHMEM	The page frame may belong to the ZONE_HIGHMEM zone.
__GFP_WAIT	Allowed to block the current process waiting for free page frames.
__GFP_HIGH	Allowed to access the pool of reserved page frames.
__GFP_IO	Allowed to perform I/O transfers on low memory pages in order to free page frames.
__GFP_FS	Allowed to perform filesystem-dependent operations
__GFP_ZERO	The page frame returned, if any, must be filled with zeros.
GFP_ATOMIC	__GFP_HIGH
GFP_KERNEL GPF_USER	__GFP_WAIT | __GFP_IO | __GFP_FS
GFP_HIGHMEM	__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM

Zone Allocator (4)

- **Components of the zone allocator**



Buddy System (1)

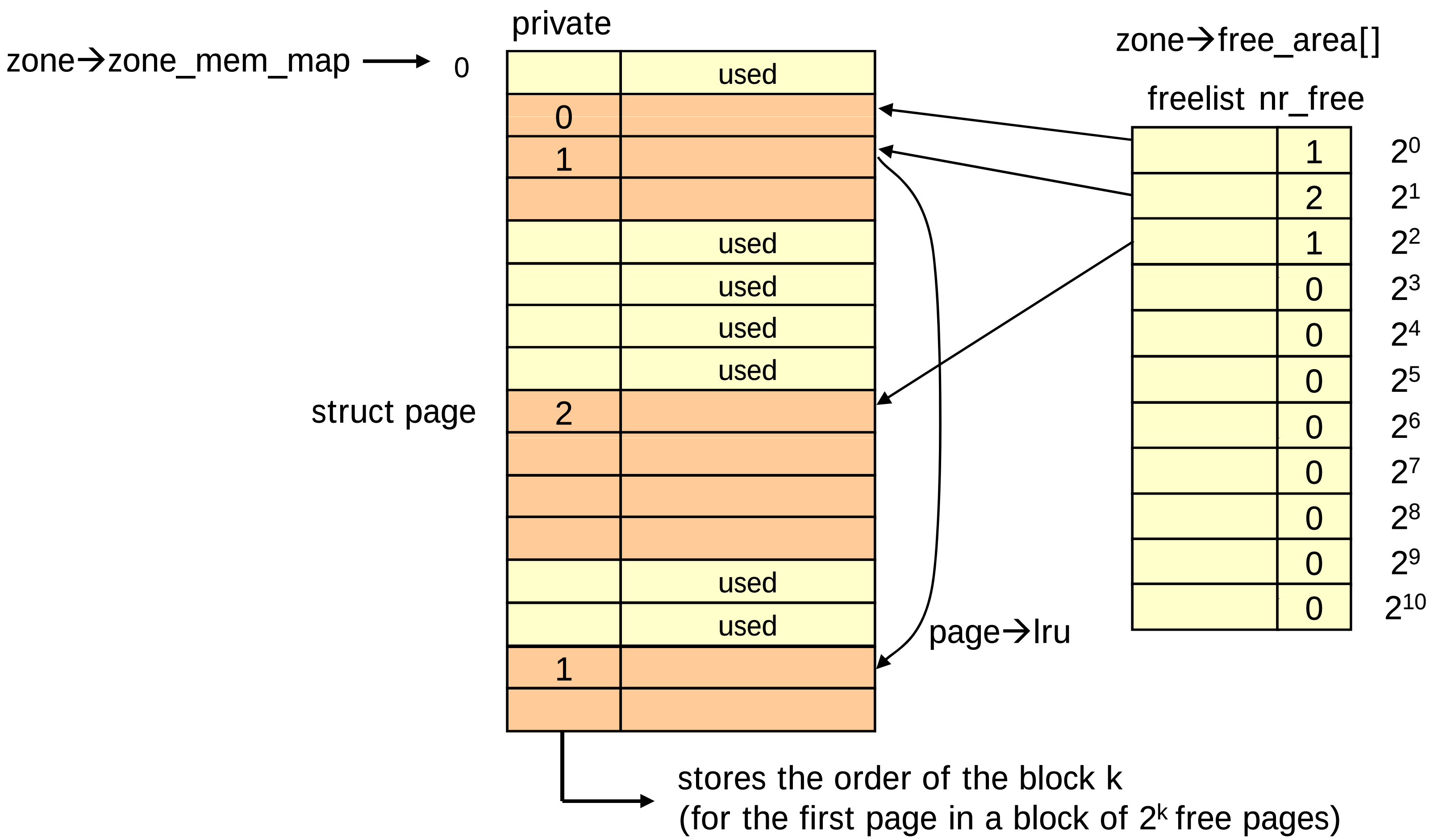
- **Buddy algorithm**
  - Treats physical memory as a collection of 2^n-page-sized blocks aligned on 2^n-page boundaries.
  - To allocate a block of a given order,
    - If a block is found at the specified order, it is allocated immediately.
    - If a block of higher order must be used, divide the larger block into two $2^{order-1}$ blocks, add the lower half to the appropriate freelist, and allocate the memory from the upper half, executing this step recursively.
  - When freeing a block,
    - If the block has a free buddy block, combine the two blocks into a single free block; and this process is performed recursively if necessary.

Buddy System (2)

- **Implementation** <mm/page_alloc.c>
  - A different buddy system for each zone.
  - All free page frames are grouped into 11 lists of blocks
    - Groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 contiguous page frames
  - struct page *__rmqueue(struct zone *zone, unsigned int order);
    - Buddy allocation
  - void __free_one_page(struct page *page, struct zone *zone, unsigned int order);
    - Buddy deallocation
  - void page_is_buddy(struct page *page, int order);
    - Checks whether the block is free and it's a buddy.

Buddy System (3)

- **Implementation (cont'd)**



Per-CPU Page Frame Cache (1)

- **Motivation**
  - The kernel often requests and releases single page frames.
  - Prepare some pre-allocated page frames to be used for single memory requests issued by the local CPU.
  - Hot cache
    - Page frames whose contents are likely to be included in the CPU's hardware cache.
    - Taking a page frame from the hot cache is beneficial for system performance.
  - Cold cache
    - Useful if the page frame is going to be filled with a DMA operation

Per-CPU Page Frame Cache (2)

- **struct per_cpu_pages** <linux/mmzone.h>

int	count;	Number of pages frame in the cache
int	low;	Low watermark for cache replenishing
int	high;	High watermark for cache depletion
int	batch;	Number of page frames to be added or subtracted from the cache
struct list_head	list;	List of descriptors of the page frames included in the cache

- **struct per_cpu_pageset** <linux/mmzone.h>
  - struct per_cpu_pageset {
    struct per_cpu_pages pcp[2];
    };
  - pcp[0]: hot cache, pcp[1]: cold cache

Per-CPU Page Frame Cache (3)

- **Managing per-CPU cache**
  - If (**count** < **low**), replenish the cache by allocating **batch** single page frames from the buddy system.
  - If (**count** > **high**), release to the buddy system **batch** single frames in the cache.
  - The kernel always assumes the freed page frame is hot.
  - The cold cache is replenished when the low watermark has been reached.

  - struct page * buffered_rmqueue (struct zonelist *zonelist,
    struct zone *zone, int order, gfp_t gfp_flags);
  - void free_hot_cold_page (struct page *page, int cold);

Slab Allocator (1)

- **Slab allocator**
  - Developed by Sun Microsystems for Solaris 2.4 in 1994.
  - The kernel functions tend to request memory areas of the same type repeatedly.
  - The slab allocator does not discard the objects that have been allocated and then released but saves them in memory.
  - When a new object is then requested, it can be taken from memory without having to be reintialized.
  - The slab allocator works on top of the buddy system.

Slab Allocator (2)

- **Caches**
  - struct kmem_cache <mm/slab.c>
  - The slab allocator groups objects into caches.
  - Each cache is a “store” of objects of the same type.
  - General caches
    - kmem_cache: objects are the cache descriptors of the remaining caches used by the kernel.
    - General-purpose caches for 13 geometrically distributed sizes
      - » 32, 64, 128, 256, 512, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K
      - » Two caches for each size: one for normal, the other for DMA
    - Initialized in kmem_cache_init()
  - Specific caches
    - For specific objects used frequently by the kernel.
    - Created by kmem_cache_create()

Slab Allocator (3)

cafe.kaist.ac.kr - PuTTY

```
names_cache            7      7   4096    1    1 : tunables   24   12    8 : slabdata      7      7      0
idr_layer_cache      299    319    136   29    1 : tunables  120   60    8 : slabdata     11     11      0
buffer_head       104419 146736     52   72    1 : tunables  120   60    8 : slabdata   2038   2038      0
mm_struct            164    180    448    9    1 : tunables   54   27    8 : slabdata     20     20      0
vm_area_struct     12173  12364     88   44    1 : tunables  120   60    8 : slabdata    281    281      0
fs_cache             159    236     64   59    1 : tunables  120   60    8 : slabdata      4      4      0
files_cache          149    189    448    9    1 : tunables   54   27    8 : slabdata     21     21      0
signal_cache         193    220    384   10    1 : tunables   54   27    8 : slabdata     22     22      0
sighand_cache        193    198   1344    3    1 : tunables   24   12    8 : slabdata     66     66      0
task_struct          221    228   1360    3    1 : tunables   24   12    8 : slabdata     76     76      0
anon_vma            5342   5588     12  254    1 : tunables  120   60    8 : slabdata     22     22      0
pgd                  145    145   4096    1    1 : tunables   24   12    8 : slabdata    145    145      0
size-131072(DMA)       0      0 131072    1   32 : tunables    8    4    0 : slabdata      0      0      0
size-131072            0      0 131072    1   32 : tunables    8    4    0 : slabdata      0      0      0
size-65536(DMA)        0      0  65536    1   16 : tunables    8    4    0 : slabdata      0      0      0
size-65536             1      1  65536    1   16 : tunables    8    4    0 : slabdata      1      1      0
size-32768(DMA)        0      0  32768    1    8 : tunables    8    4    0 : slabdata      0      0      0
size-32768             0      0  32768    1    8 : tunables    8    4    0 : slabdata      0      0      0
size-16384(DMA)        0      0  16384    1    4 : tunables    8    4    0 : slabdata      0      0      0
size-16384             8     16  16384    1    4 : tunables    8    4    0 : slabdata      8     16      0
size-8192(DMA)         0      0   8192    1    2 : tunables    8    4    0 : slabdata      0      0      0
size-8192            230    230   8192    1    2 : tunables    8    4    0 : slabdata    230    230      0
size-4096(DMA)         0      0   4096    1    1 : tunables   24   12    8 : slabdata      0      0      0
size-4096            163    163   4096    1    1 : tunables   24   12    8 : slabdata    163    163      0
size-2048(DMA)         0      0   2048    2    1 : tunables   24   12    8 : slabdata      0      0      0
size-2048            634    642   2048    2    1 : tunables   24   12    8 : slabdata    321    321      0
size-1024(DMA)         0      0   1024    4    1 : tunables   54   27    8 : slabdata      0      0      0
size-1024            340    340   1024    4    1 : tunables   54   27    8 : slabdata     85     85      0
size-512(DMA)          0      0    512    8    1 : tunables   54   27    8 : slabdata      0      0      0
size-512             600    656    512    8    1 : tunables   54   27    8 : slabdata     82     82      0
size-256(DMA)          0      0    256   15    1 : tunables  120   60    8 : slabdata      0      0      0
size-256             844    930    256   15    1 : tunables  120   60    8 : slabdata     62     62      0
size-128(DMA)          0      0    128   30    1 : tunables  120   60    8 : slabdata      0      0      0
size-128            2018   2250    128   30    1 : tunables  120   60    8 : slabdata     75     75      0
size-64(DMA)           0      0     64   59    1 : tunables  120   60    8 : slabdata      0      0      0
size-32(DMA)           0      0     32  113    1 : tunables  120   60    8 : slabdata      0      0      0
size-64             5069   5251     64   59    1 : tunables  120   60    8 : slabdata     89     89      0
size-32             3273   3616     32  113    1 : tunables  120   60    8 : slabdata     32     32      0
kmem_cache           150    150    128   30    1 : tunables  120   60    8 : slabdata      5      5      0
[dev1:/home/jinsoo-22]
```

Slab Allocator (4)

▪ Slabs

- struct slab <mm/slab.c>
- Each slab consists of one or more contiguous page frames.
- Each slab contains both allocated and freed objects.
- External slab descriptor
  - Slab descriptor is stored outside the slab. (in general cache)
- Internal slab descriptor
  - Slab descriptor is stored inside the slab.
  - When the object size is smaller than 512B, or when internal fragmentation leaves enough space.

Slab Allocator (5)

- **Objects**
  - Objects consist of a set of data structures and methods called the constructor and the destructor.
  - Each object has a short descriptor of type kmem_bufctl_t <mm/slab.c>
  - An object descriptor is meaningful only when the object is free.
    - It contains the index of the next free object in the slab.
    - The last element is marked by BUFCTL_END (0xffff).
  - External object descriptors vs.
    internal object descriptors

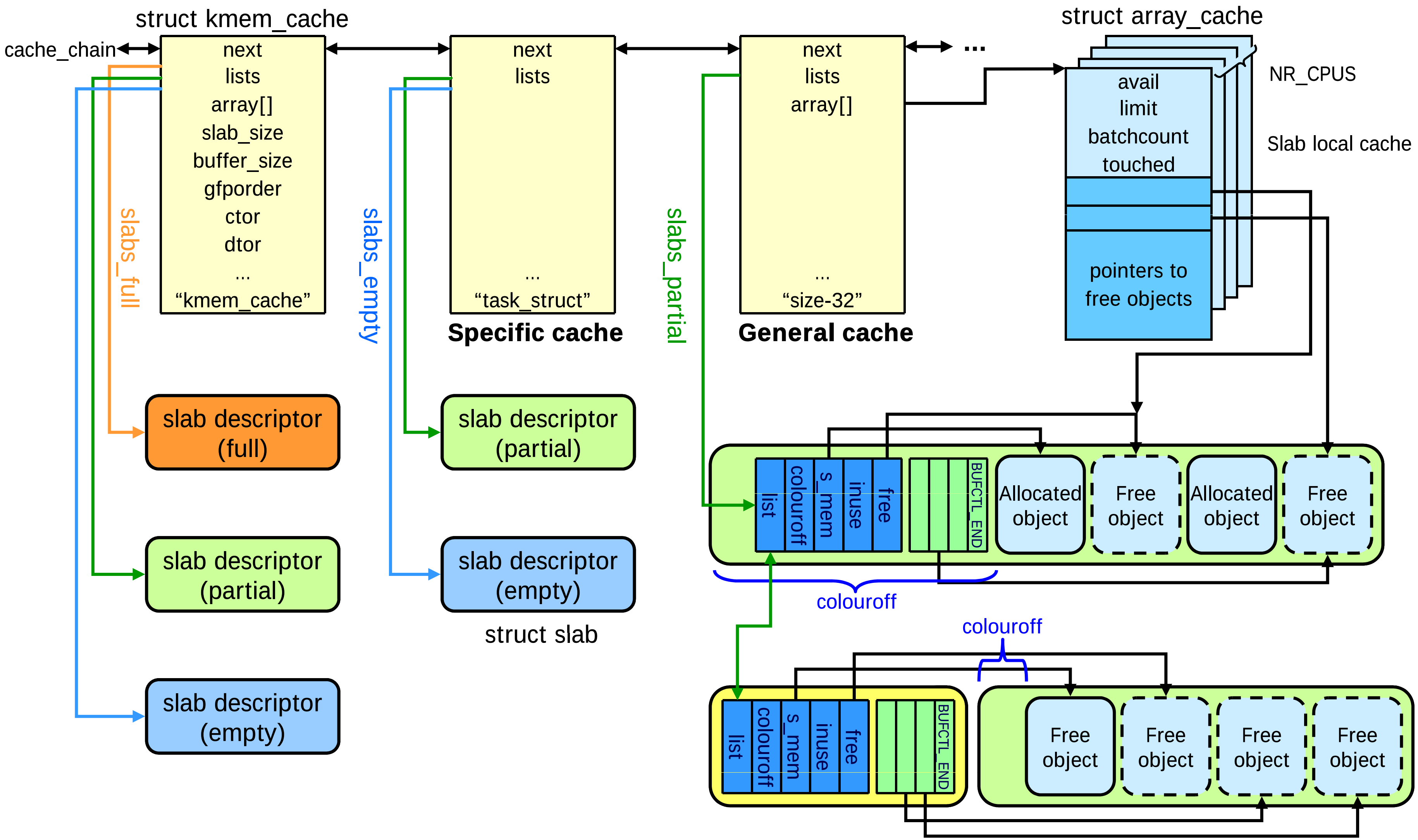
Slab Allocator (6)

- **Slab coloring**
  - Objects that have the same offset within different slabs will, with a relatively high probability, end up mapped in the same cache line.
  - The slab allocator assigns different arbitrary values called colors to the slabs.
  - Slabs having different colors store the first object in different memory locations, while satisfying the alignment constraint.
  - Coloring works only when there is enough space inside the slab.

Slab Allocator (7)

- **Slab local cache**
  - struct array_cache <mm/slab.c>
  - Per-CPU data structure consisting of a small array of pointers to freed objects.
  - Reduce spin lock contention among processors and make better use of the hardware caches.
  - The cache size depends on the size of the objects
    - 1 ~ 120 pointers
  - Local caches are refilled or emptied from the slab if needed.

Slab Allocator (8)



Slab Allocator (9)

Using specific caches

- struct kmem_cache *kmem_cache_create(const char *name,
  size_t size, size_t align, unsigned long flags,
  void (*ctor)(void *, struct kmem_cache *, unsigned long),
  void (*dtor)(void *, struct kmem_cache *, unsigned long));
- void *kmem_cache_alloc(struct kmem_cache *cachep, int flags);
- void kmem_cache_free(struct kmem_cache *cachep, void *objp);

Using general caches

- void *kmalloc(size_t size, int flags);
- void kfree(const void *objp);

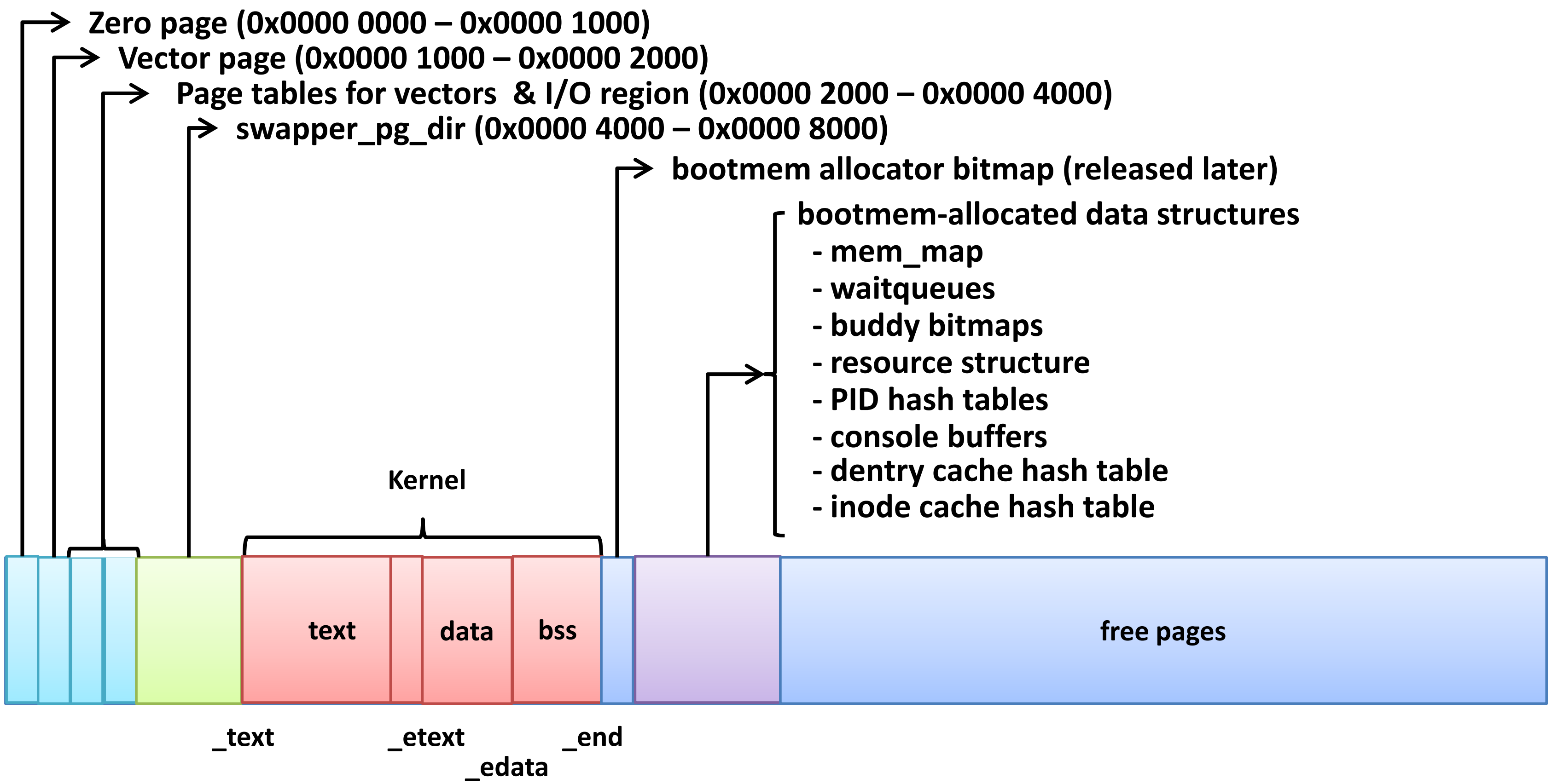
Noncontiguous Areas

- **Noncontiguous memory areas**
  - Virtually contiguous, but physically noncontiguous
  - Need to modify kernel page tables
  - Can make use of high memory page frames

- **Using a noncontiguous memory area**
  - void *vmalloc(unsigned long size);
  - void vfree(void *addr);

Where Are My Pages? (1)

- **After mem_init();**
  - Linux kernel 2.6.10 on ARM9



Where Are My Pages? (2)

- **Free pages**
  - Buddy
  - PCP (Per-CPU Page frame cache)
- **Slab**
  - For kernel data structures
  - Specific & general
- **Vmalloc**
- **Kernel stacks**
- **Page tables**
  - Kernel page tables
  - User page tables

Where Are My Pages? (3)

- **Anonymous pages**
  - Private vs. shared
- **File-mapped pages**
  - Page caches
  - Buffer caches
  - Mapped pages (private vs. shared)
- **Modules**
- **Device drivers**
  - DMA areas, buffers, etc.
- **Other page-level data structures**
  - PID bitmap, pipes, etc.

Summary

- **Memory allocation in the kernel**
  - Page-aligned
    - alloc_pages()
  - Specific caches from slab
    - kmem_cache_alloc()
  - General caches from slab
    - kmalloc()
  - From non-contiguous memory area
    - vmalloc()