Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[3] Memory Management

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v4

Outline

1. **Memory Representation**
2. **The Buddy System**
3. **High Memory**
4. **Memory Finalization**
5. **Steady-state memory allocation**
    1. Fast Allocations & Quicklists
    2. SLAB Allocator
    3. CPU Caches
    4. Large Allocations & `vmalloc`
6. **User & Kernel Space**

2. Memory Management

Memory Representation

Memory Management

During the boot, the Kernel relies on a temporary memory manager:

- it's compact and not very efficient
- The rationale is that there are not many memory requests during the boot

At steady state the boot allocator can no more be used, because:

- allocations/deallocations are frequent
- memory must be used wisely, accounting for hardware performance

We must also discover how much physical memory is available, and how it is organized.

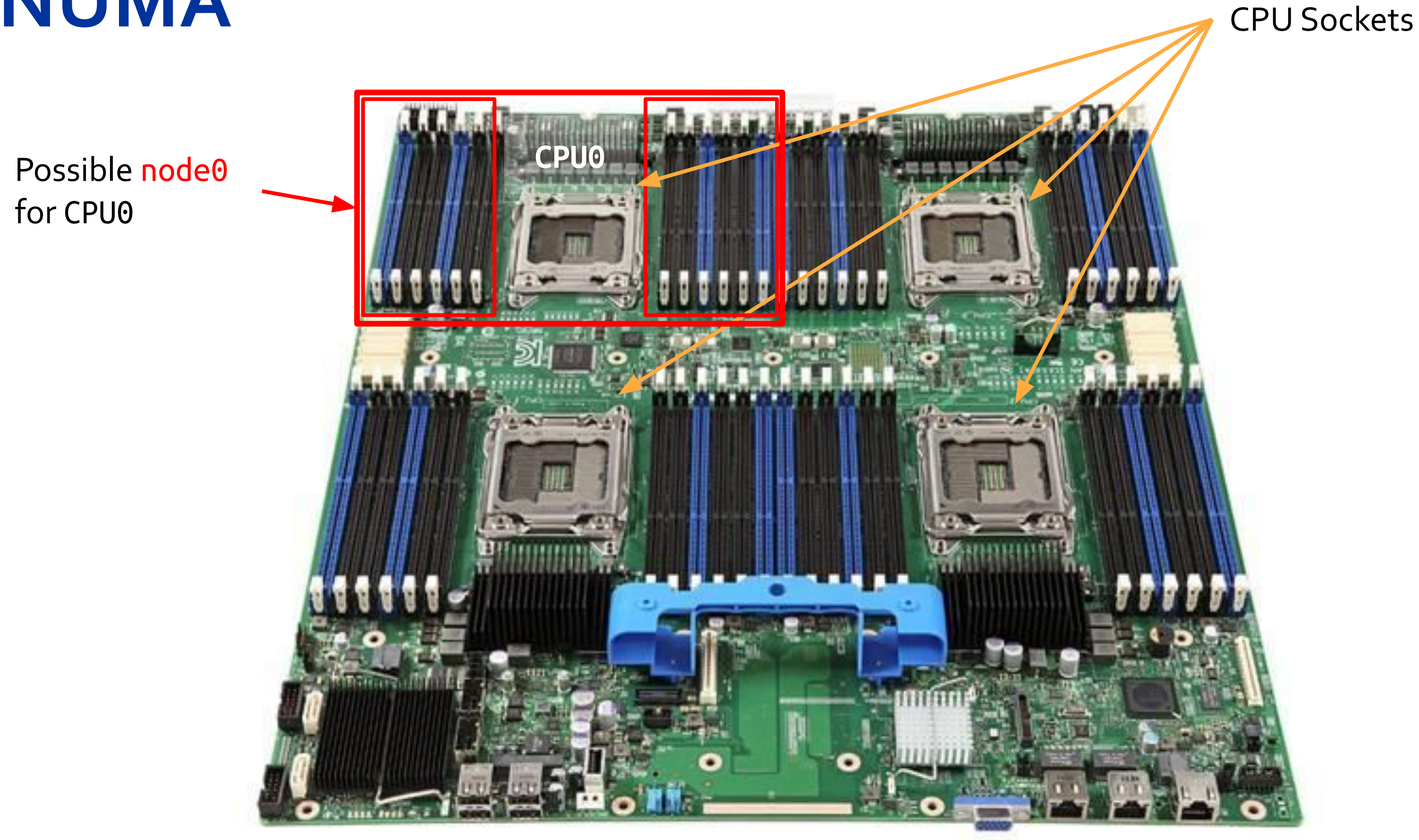
NUMA

V2.4

As anticipated, in modern computer architectures allows to see memory organized in **nodes**. This because memory access latency heavily depends on the distance between the CPU and the memory banks. This kind of memory addressing is called *Non-Uniform Memory Access* (NUMA).

In the Linux Kernel each node is represented by the `struct pg_data_t` and all nodes are kept in a NULL terminated linked list called `pgdat_list`. Each node is linked to the other with the field `pg_data_t->node_next`. In UMA architectures we only one `pg_data_t` referenced by `contig_page_data`.

NUMA



pg_data_t

V2.4

Preferred zone allocation order → `zonelist_t node_zonelists[GFP_ZONEMASK+1];`

Pointer to the first page of the array of frames of the node → `struct page *node_mem_map;`

Total number of pages in the node → `unsigned long node_size;`

Starting physical address of the node (PFN) → `unsigned long node_start_paddr;`

```
typedef struct pglist_data {
    zone_t node_zones[MAX_NR_ZONES];
    zonelist_t node_zonelists[GFP_ZONEMASK+1];
    int nr_zones;
    struct page *node_mem_map;
    unsigned long *valid_addr_bitmap;
    struct bootmem_data *bdata;
    unsigned long node_start_paddr;
    unsigned long node_start_mapnr;
    unsigned long node_size;
    int node_id;
    struct pglist_data *node_next;
} pg_data_t;
```

https://elixir.bootlin.com/linux/2.4.31/source/include/linux/mmzone.h#L166

https://elixir.bootlin.com/linux/v5.11/source/include/linux/mmzone.h#L705

Zones

Each node is divided in a number of blocks called zones, which represents ranges within the memory. A zone is described by the struct zone_struct typedef as `zone_t`. On x86 there are three kinds of zone:

- `ZONE_DMA` is directly mapped by the kernel in the lower part of memory and it is destined to ISA (Industry Standard Architecture) devices, in x86 first 16 MB
- `ZONE_NORMAL` is directly mapped by the kernel into the upper region of the linear address space, in x86 from 16MB to 896MB
- `ZONE_HIGHMEM` is the remaining available memory and it is not directly mapped by the kernel, in x86 from 896MB to end of memory.

The Page table is usually located at the top beginning of `ZONE_NORMAL`. To access memory between 1GB and 4GB the kernel temporarily maps pages from high memory to `ZONE_NORMAL`.

`ZONE_NORMAL` is fixed in size, addressing 16GiB can require 176MB of data structures!

Zones

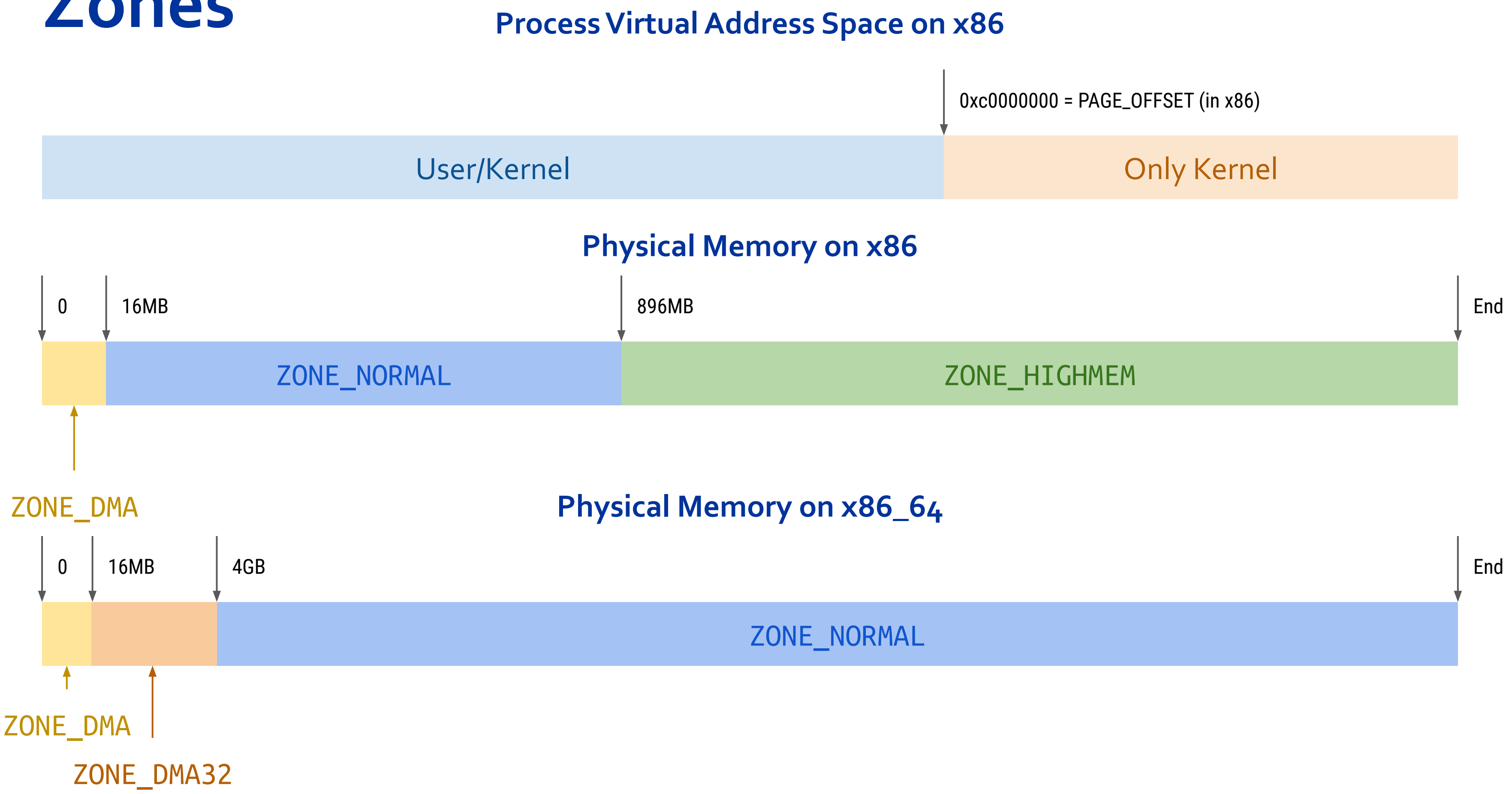
V5.11

Comparison

	x86	x86_64
`ZONE_DMA`	First 16MB	First 16MB
`ZONE_DMA32`	-	First 4GB
`ZONE_NORMAL`	From 16MB to 896MB	From 4GB to end
`ZONE_HIGHMEM`	From 896MB to end	-
`ZONE_MOVABLE`*	User Defined	User Defined

Please remind that for example in x86_64 if we have only 2GB of RAM, all the RAM will be ZONE_DMA32. If instead we have 16GB the kernel will allocate memory by following possible flags you pass and the available memory type. See also `/proc/pagetypeinfo`

Zones


Process Virtual Address Space on x86
0xc0000000 = PAGE_OFFSET (in x86)
User/Kernel
Only Kernel
Physical Memory on x86
0
16MB
896MB
End
ZONE_NORMAL
ZONE_HIGHMEM
ZONE_DMA
Physical Memory on x86_64
0
16MB
4GB
End
ZONE_NORMAL
ZONE_DMA
ZONE_DMA32

Zones Initialization

Zones are initialized after the kernel page tables have been fully set up by paging_init(). The goal is to determine what parameters to send to:

- `free_area_init()` for UMA machines
- `free_area_init_node()` for NUMA machines

The initialization grounds on PFNs max PFN is read from BIOS e820 table.

```
[    +0.000000] e820: BIOS-provided physical RAM map:
[    +0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009f7ff] usable
[    +0.000000] BIOS-e820: [mem 0x000000000009f800-0x000000000009ffff] reserved
[    +0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[    +0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000bfd9ffff] usable
[    +0.000000] BIOS-e820: [mem 0x00000000bfda0000-0x00000000bfdd0fff] ACPI NVS
[    +0.000000] BIOS-e820: [mem 0x00000000bfdd1000-0x00000000bfdfffff] ACPI data
[    +0.000000] BIOS-e820: [mem 0x00000000bfe00000-0x00000000bfefffff] reserved
[    +0.000000] BIOS-e820: [mem 0x00000000e0000000-0x00000000efffffff] reserved
[    +0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000ffffffff] reserved
[    +0.000000] BIOS-e820: [mem 0x0000000100000000-0x000000043effffff] usable
```

zone_t

V2.4

```
typedef struct zone_struct {
	spinlock_t lock;
	unsigned long free_pages;
	zone_watermarks_t watermarks[MAX_NR_ZONES];
	unsigned long need_balance;
	unsigned long nr_active_pages,nr_inactive_pages;
	unsigned long nr_cache_pages;
	free_area_t free_area[MAX_ORDER];
	wait_queue_head_t *wait_table;
	unsigned long wait_table_size;
	unsigned long wait_table_shift;
	struct pglist_data *zone_pgdat;
	struct page *zone_mem_map;
	unsigned long zone_start_paddr;
	unsigned long zone_start_mapnr;
	char *name;
	unsigned long size;
	unsigned long realsize;
} zone_t;
```

https://elixir.bootlin.com/linux/2.4.31/source/include/linux/mmzone.h#L39

Zone Watermarks

When available memory in the system is low, the pageout daemon **kswapd** is woken up to start freeing pages. Each zone has **three watermarks** called *pages low*, *pages min* and *pages high*, which help track how much pressure a zone is under.

- **pages low** When the pages low number of free pages is reached, kswapd is woken up by the buddy allocator
- **pages min** When pages min is reached, the allocator will do the kswapd work in a synchronous fashion
- **pages high** After kswapd has been woken to start freeing pages, it will not consider the zone to be "balanced" when pages high pages are free

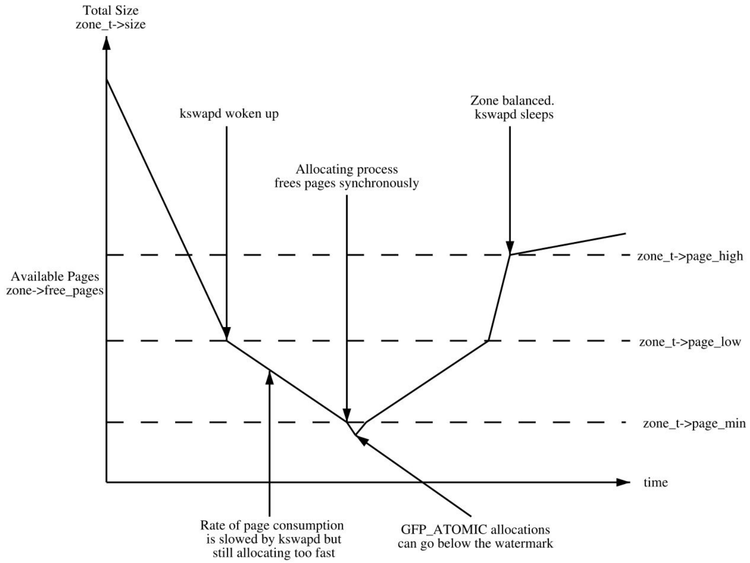


Figure 2.2. Zone Watermarks

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

Nodes, Zones and Pages

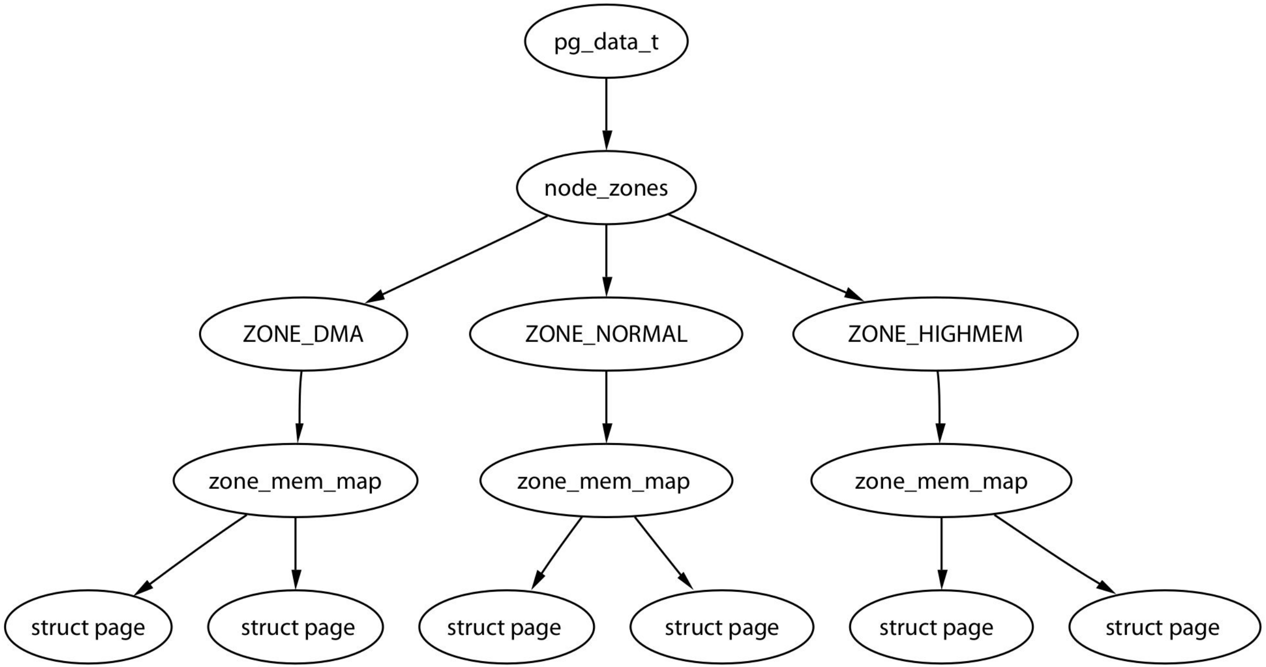


Figure 2.1. Relationship Between Nodes, Zones and Pages

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

Core Map

V2.4

The Core Map is an array of mem_map_t structures defined in include/linux/mm.h and kept in ZONE_NORMAL. The struct page is associated to every physical frame available in the system.

```
typedef struct page {
	struct list_head list;
	struct address_space *mapping;
	unsigned long index;
	struct page *next_hash;
	atomic_t count;
	unsigned long flags;
	struct list_head lru;
	struct page **pprev_hash;
	struct buffer_head * buffers;
#if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
	void *virtual;
#endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
} mem_map_t;
```

List head to which the page belongs. A page may belong to different lists

Address space (e.g. inode) and index to which the page belongs

Usage counter, if zero the page may be free'd

```
Page flags:
#define PG_locked 0
#define PG_referenced 2
#define PG_uptodate 3
#define PG_dirty 4
#define PG_lru 6
#define PG_reserved 14
```

Core Map

V2.4

How to manage flags

Bit Name	Set	Test	Clear
PG_active	SetPageActive()	PageActive()	ClearPageActive()
PG_arch_1	None	None	None
PG_checked	SetPageChecked()	PageChecked()	None
PG_dirty	SetPageDirty()	PageDirty()	ClearPageDirty()
PG_error	SetPageError()	PageError()	ClearPageError()
PG_highmem	None	PageHighMem()	None
PG_launder	SetPageLaunder()	PageLaunder()	ClearPageLaunder()
PG_locked	LockPage()	PageLocked()	UnlockPage()
PG_lru	TestSetPageLRU()	PageLRU()	TestClearPageLRU()
PG_referenced	SetPageReferenced()	PageReferenced()	ClearPageReferenced()
PG_reserved	SetPageReserved()	PageReserved()	ClearPageReserved()
PG_skip	None	None	None
PG_slab	PageSetSlab()	PageSlab()	PageClearSlab()
PG_unused	None	None	None
PG_uptodate	SetPageUptodate()	PageUptodate()	ClearPageUptodate()

Table 2.2. Macros for Testing, Setting and Clearing `page→flags` Status Bits

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

Core Map

On UMA

Initially we have the core map pointer, mem_map defined in `mm/memory.c`. The pointer initialization is done within the function `free_area_init()`. After the initialization each entry will keep the value 0 within the count field and the value 1 into flags for the `PG_RESERVED` flag. Therefore we do not have any virtual reference to the frame and the frame is reserved. The un-reserving is done by the `mem_init()` function.

On NUMA

There's not a global mem_map array since every node keeps its own map in its own memory. The map is pointed by `pg_data_t -> node_mem_map` but the map organization is the same.

3.2

2. Memory Management

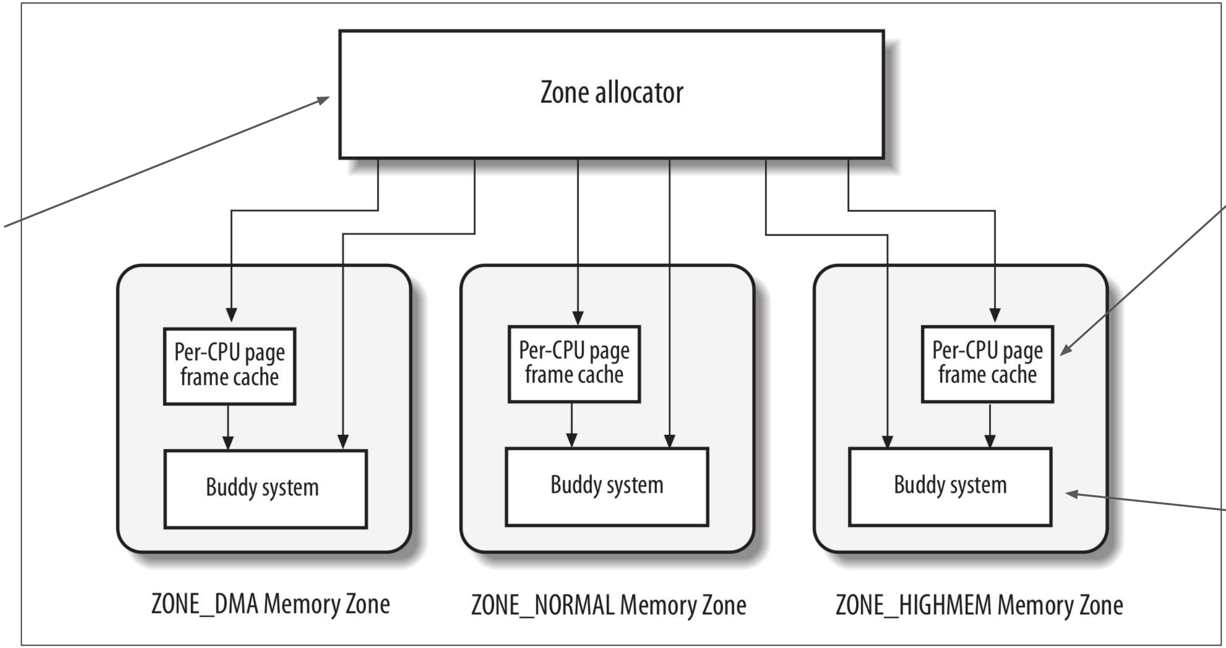
The Buddy System

Zoned Page Frame Allocator

V2.4

The kernel subsystem that handles the memory allocation for contiguous page frames is called *zoned page frame allocator.*

Receives the allocation requests for dynamic memory, then it **searches** a **zone** for performing the allocation



Small **cache** for speeding up allocation requests for single page frames

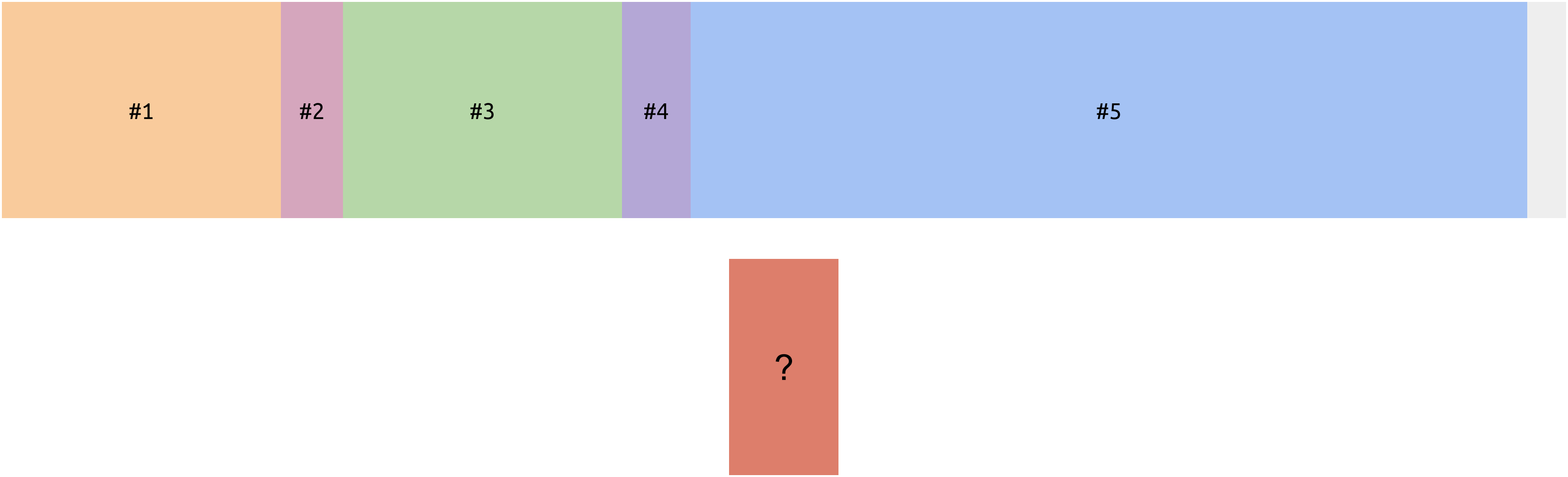
Manages the page frames allocation **inside** each **zone**

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

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Fragmentation

When allocating groups of contiguous page frames, the algorithm that we need to design, must deal with a well-know problem called **External Fragmentation**.We allocate #1, #2, #3, #4 and #5 consecutively, then we deallocate #2 and #4. Where can we put a new allocation request for a size of #2 + #4 for example? We have that memory available but it is not **contiguous**.



Fragmentation

There are two approaches, in general to solve the problem:

1. use the **paging circuitry** to map group of non-contiguous pages into intervals of contiguous linear addresses
2. develop a suitable technique to keep track of the existing blocks of free contiguous page frames, avoiding as much as possible the need to **split up large free block** to satisfy a request for a smaller one

The Linux kernel prefers the second, for 3 good reasons:

- in some cases we really need contiguous pages, not only contiguous linear addresses (e.g. DMA)
- frequent page table modifications lead to higher average memory access times, e.g. flushing the TLB
- large chunks of physical memory can be accessed with 4MB pages, reducing TLB miss and speeding up access times

Buddy System

V2.4

The technique followed by the Linux kernel for solving external fragmentation is based on the well-known buddy system algorithm. The Buddy System keeps all the free pages grouped into 11 lists of blocks that contain groups of 1,2,4,8,16,32,64,128,256,512 and 1024 contiguous frames. 1024 page frames correspond to 4MB of memory.

The data structures used by the algorithm are:

- the `mem_map` array, that is the **core map** that we already discussed. Actually, each zone is concerned with a subset of the mem_map elements
- an **array** of **eleven elements** of `free_area_t`, one for each group size. This array is stored in the `free_area` field of the zone descriptor and contains the linked list of free page blocks and a pointer to a bitmap (*map), in which each bit represents a **pair of buddies**. <u>The bit is set to `0` when both buddies are full or free, and `1` when only one buddy is used</u>.

Buddy System

V2.4

Data Structures

```
typedef struct zone_struct {

        /*
         * free areas of different sizes
         */
        free_area_t             free_area[MAX_ORDER];
```

https://elixir.bootlin.com/linux/2.4.31/source/include/linux/mmzone.h#L78

```
/*
 * Simple doubly linked list implementation.
 *
 * Some of the internal functions ("__xxx") are useful when
 * manipulating whole lists rather than single entries, as
 * sometimes we already know the next/prev entries and we can
 * generate better code by using them directly rather than
 * using the generic single-entry routines.
 */

struct list_head {
        struct list_head *next, *prev;
};
```

https://elixir.bootlin.com/linux/2.4.31/source/include/linux/list.h#L18

```
typedef struct free_area_struct {
        struct list_head        free_list;
        unsigned long           *map;
} free_area_t;
```

https://elixir.bootlin.com/linux/2.4.31/source/include/linux/mmzone.h#L30

Buddy System

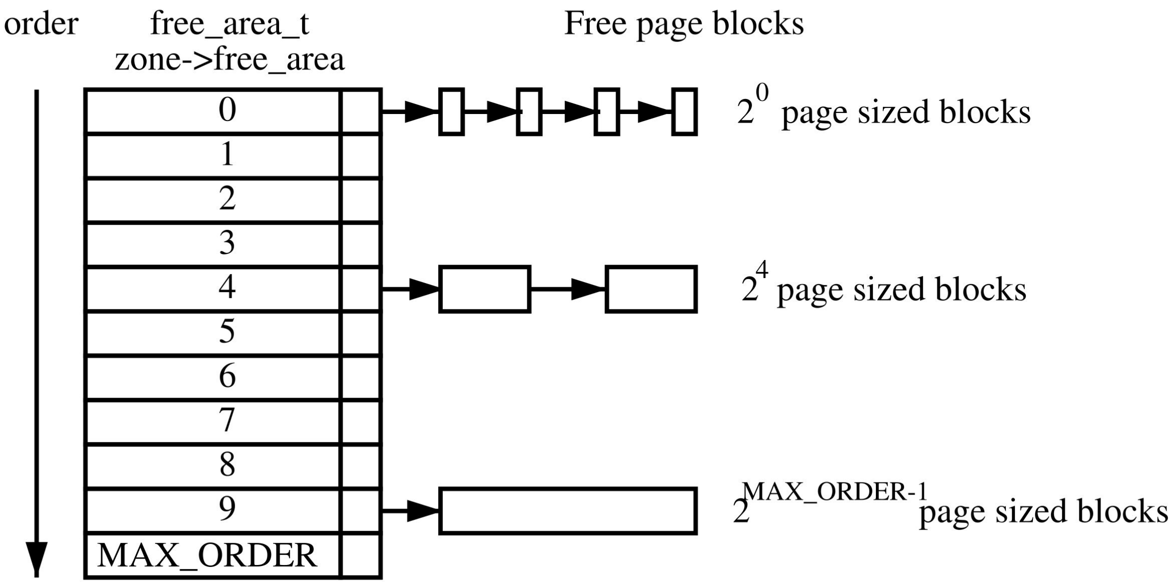


Figure 6.1. Free Page Block Management

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

Buddy System

V2.4

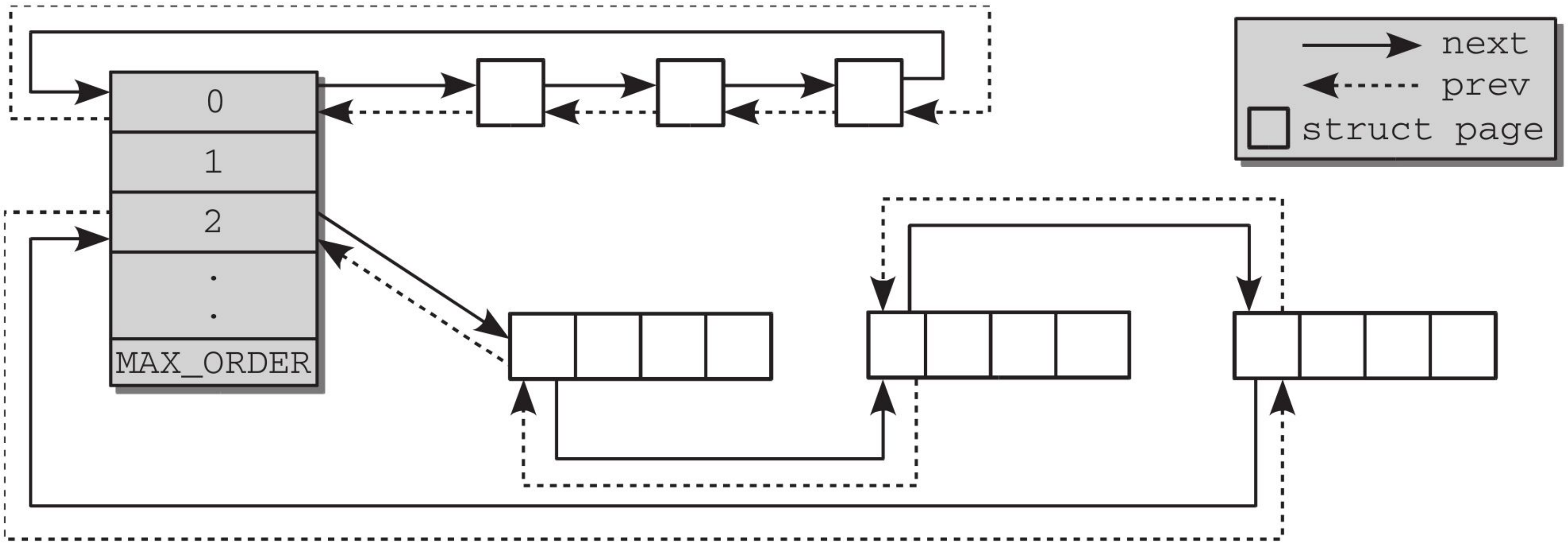


Figure 3-22: Linking blocks in the buddy system.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Buddy System

V2.4

Allocation

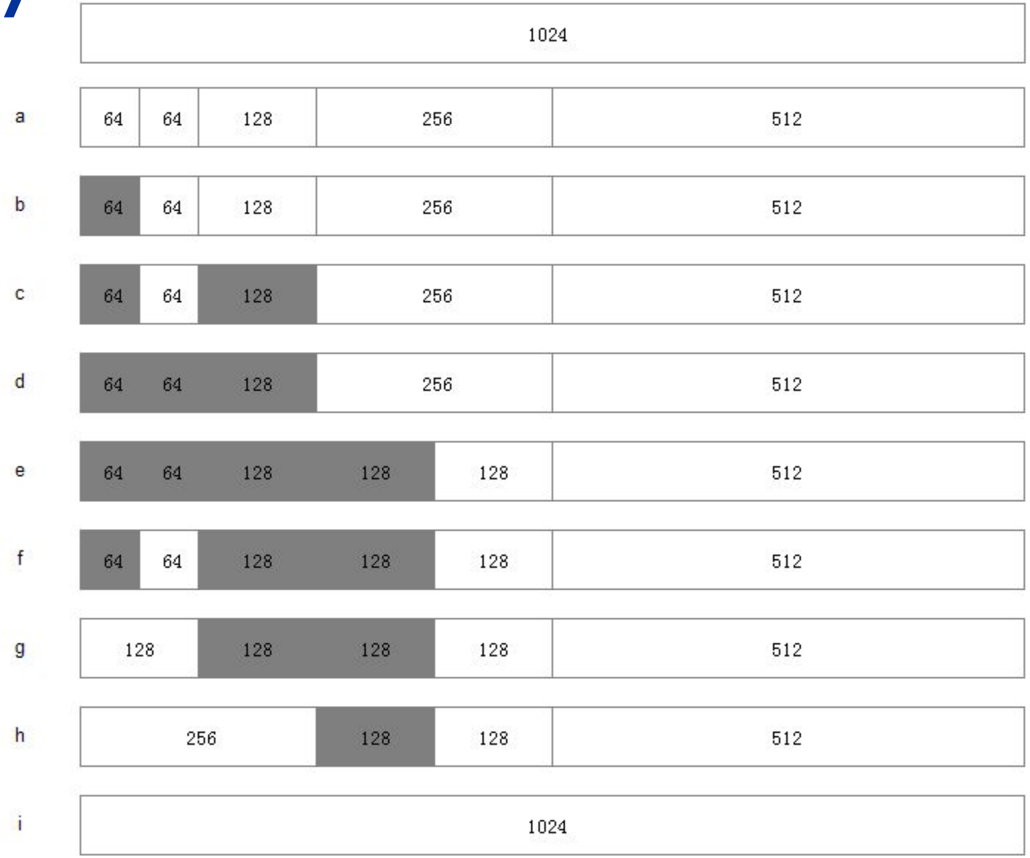
Suppose that you want to allocate 256 contiguous page frames, the algorithm check if there is a free 256 block, if not it checks in the list of 512. If it exists it allocates 256 pages for satisfying the request and the other 256 are added into the list of free 256-page-frame blocks. If there is no free 512-page block the kernel looks for next larger block, 1024. If it exists, it allocates 256 of the 1024 page frames to satisfy the request, then inserts the first 512 of the remaining 768 into the list of free 512-page-frame blocks and the last 256 pages frames into the list of free 256-page-frame blocks.

Deallocation

When freeing memory, the kernel attempts to merge a pair of buddy blocks of size b together into a single block of size 2b. Only if (i) they have the same size, (ii) they are contiguous, (iii) the physical address of the first block is multiple of $2 \times b \times 2^{12}$.

During the allocation and deallocation interrupts must be disabled and this is node by using a particular kind of spinlock (we will see later in the course).

Buddy System



https://www.programmersought.com/article/96354519239/

Retrieving a page from free_area list

V2.4

The function rmqueue() is used to find a free block in a zone.

```c
static struct page * fastcall rmqueue(zone_t *zone, unsigned int order)
{
	free_area_t * area = zone->free_area + order;
	unsigned int curr_order = order;
	struct list_head *head, *curr;
	unsigned long flags;
	struct page *page;

	spin_lock_irqsave(&zone->lock, flags);
	do {
		head = &area->free_list;
		curr = head->next;

		if (curr != head) {
			unsigned int index;

			page = list_entry(curr, struct page, list);
			if (BAD_RANGE(zone,page))
				BUG();
			list_del(curr);
			index = page - zone->zone_mem_map;
			if (curr_order != MAX_ORDER-1)
				MARK_USED(index, curr_order, area);
			zone->free_pages -= 1UL << order;

			page = expand(zone, page, index, order, curr_order, area);
	} while (curr_order < MAX_ORDER);
	spin_unlock_irqrestore(&zone->lock, flags);

	return NULL;
}
```

https://elixir.bootlin.com/linux/2.4.31/source/mm/page_alloc.c#L242

```c
/**
 * list_entry - get the struct for this entry
 * @ptr:	the &struct list_head pointer.
 * @type:	the type of the struct this is embedded in.
 * @member:	the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
	((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

https://elixir.bootlin.com/linux/2.4.31/source/include/linux/list.h#L187

The list_entry macro allows you to retrieve the entry in the linked list that has the ptr you specify.

In this case it is used for retrieving the struct page from the `free_area` list

Adding a page to free_area list

V2.4

The expand() function called by rmqueue() add the free block to the zone by using the function/macro (in other kernel versions) list_add().

```c
static inline struct page * expand (zone_t *zone, struct page *page,
         unsigned long index, int low, int high, free_area_t * area)
{
        unsigned long size = 1 << high;

        while (high > low) {
                if (BAD_RANGE(zone,page))
                        BUG();
                area--;
                high--;
                size >>= 1;
                list_add(&(page)->list, &(area)->free_list);
                MARK_USED(index, high, area);
                index += size;
                page += size;
        }
        if (BAD_RANGE(zone,page))
                BUG();
        return page;
}
```

```c
/**
 * list_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static inline void list_add(struct list_head *new, struct list_head *head)
{
        __list_add(new, head, head->next);
}
```

```c
/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_add(struct list_head *new,
                              struct list_head *prev,
                              struct list_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}
```

3.3

2. Memory Management

High Memory

Concept

On x86 the kernel directly maps only `ZONE_DMA` and `ZONE_NORMAL` for a total of 896MB, but obviously machines started to have more than 4GB of RAM. Due to the fixed limit 3GB/1GB of the address space, the kernel cannot map directly more than 896MB, for this reason all the memory mapping that exceeds that size are temporarily and they refer to the **High Memory** concept.

Mapping area of HighMem in kernel the virtual address space

Figure 3-15: Division of the kernel address space on IA-32 systems.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

PKMap

The kernel virtual address spaces from address PKMAP_BASE to FIXADDR_START is reserved for a PKMap, namely a **Persistent Kernel Map** located near the end of the address space. There are about 32MB of page table space for mapping pages from high memory into the usable space.

For mapping pages, a simple PT of 1024 entries is stored at the beginning of the PKMap area to allow the **temporary** (very short time) mapping of up to 1024 pages from high mem with functions `kmap()` and `kunmap()`. That page is initialized at the end of pagetable_init() function.

The current state of page table entries is managed by a simple array called `pkmap_count` with LAST_KMAP (= PTRS_PER_PTE = 1024 or 512 when PAE is enabled) entries.

pkmap_count

V2.4

```c
/*
 * Virtual_count is not a pure "count".
 *  0 means that it is not mapped, and has not been mapped
 *    since a TLB flush - it is usable.
 *  1 means that there are no users, but it has been mapped
 *    since the last TLB flush - so we can't use it.
 *  n means that there are (n-1) current users of it.
 */
static int pkmap_count[LAST_PKMAP];
```

https://elixir.bootlin.com/linux/2.4.31/source/mm/highmem.c#L33

APIs

V2.4

- `kmap()` it permits a short-duration mapping of a single page, requires global synchronization
- `kmap_atomic()` permits a very short duration mapping of a single page but it is restricted to the CPU that issued it and the task must be on that CPU until the termination, usage is discouraged
- `kunmap()` decrements the associated page counter. When the counter is 1 the mapping is not needed anymore but the CPU has still cached that mapping, for this reason TLB must be flushed manually
- `kunmap_atomic()` unmaps a page that has been mapped atomically

2. Memory Management

Memory Finalization

Reclaiming Boot Memory

The finalization of memory management is done within the function `mem_init()` which is in charge of destroying the bootmem allocator, calculating the dimensions of low and high memory and printing out an informational message to the user.

On x86 the principle function called by mem_init is `free_pages_init()`.

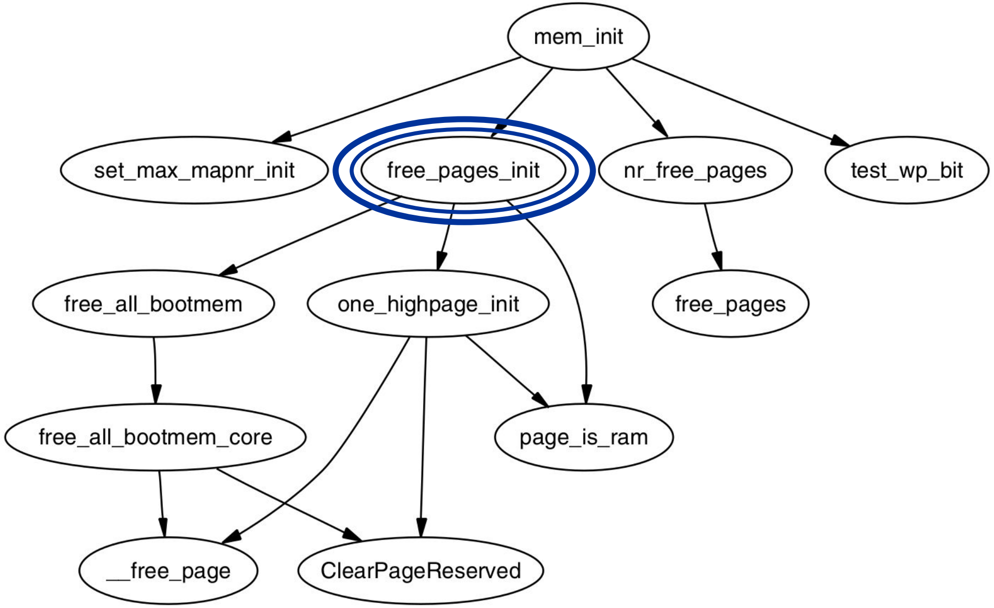


Figure 5.2. Call Graph: `mem_init()`

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

free_all_bootmem_core

The free_all_bootmem is called by each NUMA node and in the end it calls free_all_bootmem_core which does the following.

For each unallocated pages known to the allocator of that node

- clears the PG_RESERVED bit
- set usage count to 1
- call `__free_pages()` so that the buddy allocator can build its free lists

Free all pages used for the bitmap and give them to the buddy allocator.

free_all_bootmem_core

V2.4

```c
static unsigned long __init free_all_bootmem_core(pg_data_t *pgdat)
{
	struct page *page = pgdat->node_mem_map;
	bootmem_data_t *bdata = pgdat->bdata;
	unsigned long i, count, total = 0;
	unsigned long idx;

	if (!bdata->node_bootmem_map) BUG();

	count = 0;
	idx = bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT);
	for (i = 0; i < idx; i++, page++) {
		if (!test_bit(i, bdata->node_bootmem_map)) {
			count++;
			ClearPageReserved(page);
			set_page_count(page, 1);
			__free_page(page);
		}
	}
	total += count;
```

https://elixir.bootlin.com/linux/2.4.31/source/mm/bootmem.c#L245

Finalizing

When free_all_bootmem returns all the pages in `ZONE_NORMAL` have been given to the buddy allocator, the rest of `free_pages_init` initializes the high memory.

In particular, [`one_highpage_init()`]() is called for every page between `highstart_pfn` and `highend_pfn` and it simply:

- clears the `PG_RESERVED` flag
- set the `PG_HIGHMEM` flag
- set the count to 1
- calls `__free_pages()` to release it to the Buddy Allocator

At this point, the boot memory allocator is no longer required, and the buddy allocator is the main physical page allocator for the system. Note also that not only is the data for the boot allocator removed, but also all code that was used to bootstrap the system. `free_all_bootmem()` is marked by `__init()`.

2. Memory Management

Steady-state memory allocation

Allocation Contexts

In general, in a kernel, we can recognize two kinds of memory allocation contexts at steady-state.

- **Process Context**, that refers to an allocation that has been requested through a system call, typical of userspace processes.

  Within this context, if the request cannot be served, the process is put on wait by following also a priority-based approach

- **Interrupt Context** , that refers to an allocation due to a interrupt handler

  Within this context, if the request cannot be served there's no waiting time and the approach is not priority based

Physical Frame Allocation APIs

Within the kernel, the following functions for memory allocation can be used, they are declared at <linux/malloc.h>.

Memory allocation requests created with these functions are obviously managed by the Buddy Allocator.

`struct page * alloc_page(unsigned int gfp_mask)`	Allocates a single page and returns a struct address.
`struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)`	Allocates 2^{order} number of pages and returns a struct page.
`unsigned long get_free_page(unsigned int gfp_mask)`	Allocates a single page, zeros it, and returns a virtual address.
`unsigned long __get_free_page(unsigned int gfp_mask)`	Allocates a single page and returns a virtual address.
`unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)`	Allocates 2^{order} number of pages and returns a virtual address.
`struct page * __get_dma_pages(unsigned int gfp_mask, unsigned int order)`	Allocates 2^{order} number of pages from the DMA zone and returns a struct page.

Table 6.1. Physical Pages Allocation API

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

Physical Frame Deallocation API

`void __free_pages(struct page *page, unsigned int order)`	Frees an order number of pages from the given page.
`void __free_page(struct page *page)`	Frees a single page.
`void free_page(void *addr)`	Frees a page from the given virtual address.

Table 6.2. Physical Pages Free API

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

Remember that within the Buddy Allocator, the caller needs to remember the allocated size and the address. If you pass a wrong `void* addr` to `free_page()` you could corrupt the kernel.

Flags

Flag	Description
__GFP_WAIT	Indicates that the caller is not high priority and can sleep or reschedule.
__GFP_HIGH	Used by a high priority or kernel process. Kernel 2.2.x used it to determine if a process could access emergency pools of memory. In 2.4.x kernels, it does not appear to be used.
__GFP_IO	Indicates that the caller can perform low-level I/O. In 2.4.x, the main effect this has is determining if `try_to_free_buffers()` can flush buffers. It is used by at least one journaled filesystem.
__GFP_HIGHIO	Determines that I/O can be performed on pages mapped in high memory. It is only used in `try_to_free_buffers()`.
__GFP_FS	Indicates if the caller can make calls to the filesystem layer. This is used when the caller is filesystem related, the buffer cache, for instance, and wants to avoid recursively calling itself.

Table 6.4. Low-Level GFP Flags Affecting Allocator Behavior

Flag	Description
__GFP_DMA	Allocate from ZONE_DMA if possible.
__GFP_HIGHMEM	Allocate from ZONE_HIGHMEM if possible.
GFP_DMA	Act as alias for __GFP_DMA.

Table 6.3. Low-Level GFP Flags Affecting Zone Allocation

Gorman, Mel. Understanding the Linux virtual memory manager.
Upper Saddle River: Prentice Hall, 2004.

Flags

High priority

Flag	Low-Level Flag Combination
GFP_ATOMIC	HIGH
GFP_NOIO	HIGH — WAIT
GFP_NOHIGHIO	HIGH — WAIT — IO
GFP_NOFS	HIGH — WAIT — IO — HIGHIO
GFP_KERNEL	HIGH — WAIT — IO — HIGHIO — FS
GFP_NFS	HIGH — WAIT — IO — HIGHIO — FS
GFP_USER	WAIT — IO — HIGHIO — FS
GFP_HIGHUSER	WAIT — IO — HIGHIO — FS — HIGHMEM
GFP_KSWAPD	WAIT — IO — HIGHIO — FS

Table 6.5. Low-Level GFP Flag Combinations for High-Level Use

Can sleep

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

NUMA Policies

When we have a NUMA architecture, the function __get_free_pages() calls alloc_page_node() specifying a NUMA policy. A **NUMA policy** determines from which node the memory will be allocated. This support was added in kernel 2.6.

set_mempolicy()

The function set_mempolicy sets the NUMA memory policy of the calling process.

```
#include <numaif.h>
int set_mempolicy(int mode, unsigned long *nodemask, unsigned long maxnode);
```

Where mode can be:

- MPOL_DEFAULT allocate on node of the CPU that issued the command
- MPOL_BIND strictly allocate to the specified nodemask
- MPOL_INTERLEAVE interleaves allocation to the specified nodemask nodes
- MPOL_PREFERRED sets the preferred node(s) for the allocation as nodemask

nodemask points to a bit mask of node IDs that contains up to maxnode bits

https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html
https://linux.die.net/man/2/set_mempolicy

NUMA Policies

mbind()

The function mbind() assigns a NUMA policy to the specified set of memory addresses.

```
#include <numaif.h>
long mbind(void *addr, unsigned long len, int mode,
           const unsigned long *nodemask, unsigned long maxnode, unsigned flags);
```

move_pages()

This function moves the specified pages of the process pid to the memory nodes specified by nodes. The result of the move is reflected in status. The flags parameter indicates constraints on the pages to be moved.

```
#include <numaif.h>
long move_pages(int pid, unsigned long count,
                void **pages, const int *nodes, int *status, int flags);
```

3.5.1

2. Memory Management

5. Steady-State Memory Allocation

Fast Allocations & Quicklists

Frequent Allocations and Deallocations

In general, within the kernel, fixed size data structures are very often allocated and released. The Buddy System that we presented earlier clearly does not scale:

- this is a classic case of frequent logical contention
- the buddy system on each NUMA node is protected by a (*spin*)lock
- internal fragmentation can rise too much

Example

Allocation and release of page tables requires a frequent allocation and deallocation of the same fixed size structures. The functions that allows us to create page tables like

- pgd_alloc(), pmd_alloc() and pte_alloc()
- pgd_free(), pmd_free() and pte_free()

They relies on Kernel-level **fast allocators**.

Fast Allocators

There are two fast allocators in the kernel:

- **quicklists**, used only for paging
- **SLAB Allocator**, used for other buffers. There are three implementations of the SLAB allocator:
    - the SLAB: implemented around 1994
    - the SLUB: the unqueued SLAB allocator, default since 2.6.23
    - the SLOB: Simple List Of Blocks, if the SLAB is not enabled this is the fallback

Quicklists

Quicklists are used for implementing the page table cache. For the three functions `pgd/pmd/pte_alloc()` we have three quicklists `pgd/pmd/pte_quicklist` **per CPU**. Each architecture implements its own version of quicklists but the principle is the same.

One method is the one of using the LIFO (Last-In First-Out) approach. During the **allocation**, one page is popped off the list, and during **free**, one is placed as the new head of the list. This is done while keeping a count of how many pages are used in the cache.

If a page is not available in the cache, then it will be allocated by using the Buddy System. Obviously, a large amount of free pages can exist in these caches, for this reason they are **pruned** by using a watermarking strategy.

quicklist_alloc

v2.6

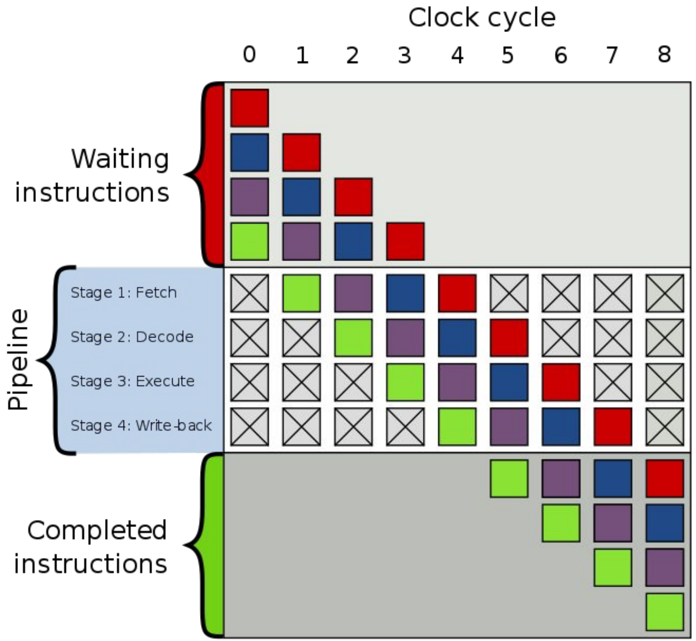
```
static inline void *quicklist_alloc(int nr, gfp_t flags, void (*ctor)(void *))
{
	struct quicklist *q;
	void **p = NULL;

	q =&get_cpu_var(quicklist)[nr];
	p = q->page;
	if (likely(p)) {
		q->page = p[0];
		p[0] = NULL;
		q->nr_pages--;
	}
	put_cpu_var(quicklist);
	if (likely(p))
		return p;

	p = (void *)__get_free_page(flags | __GFP_ZERO);
	if (ctor && p)
		ctor(p);
	return p;
}
```

https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/quicklist.h#L33

`likely()` and `unlikely()`

The `likely()` and `unlikely()` are used for the branch prediction mechanism of the CPU. Branch prediction allows to optimize the CPU pipeline and increasing the performance of the CPU. The likely instruction will tell the compiler that the if condition will likely hit and the CPU can prepare the pipeline for that jump.

The converse is for unlikely. When an likely branch will not be hit then the entire CPU pipeline will be flushed. This will have an impact on performances but it will rarely happen.



https://en.wikipedia.org/wiki/Branch_predictor

3.5.2

2. Memory Management

5. Steady-State Memory Allocation

SLAB Allocator

Overview

The general idea behind the SLAB allocator is to have caches of commonly used objects kept in a initialized state available for use by the kernel.

The SLAB allocator consists of a variable number of **caches**, linked together by a doubly linked list called *cache chain*. Every cache manages objects of particular kind (e.g. `mm_struct`). Each cache maintains a block of contiguous pages in memory called **slabs**.

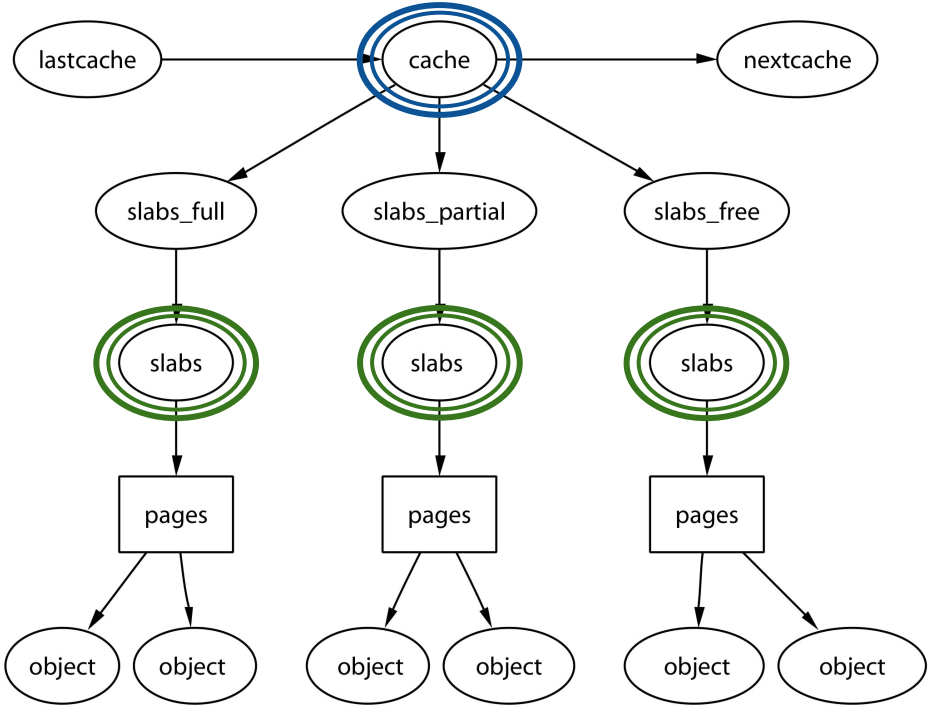


Figure 8.1. Layout of the Slab Allocator

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

Aims

The purpose of the SLAB allocator is threefold:

1. allocating small blocks of memory to help **eliminate internal fragmentation** caused by the Buddy System
2. **caching commonly** used **blocks** so that the system does not wait time allocating, initializing and destroying object
3. **better usage** of L1 and L2 **caches** by aligning objects

Aim #1

Two sets of caches are maintained for allocating objects from 2^5 (32KB) to 2^{17} (131'072KB) bytes. One for DMA and one for standard allocation. These caches are called **size-N** (or **size-N(DMA)**), where N is the size of the allocation and they are allocated with the function **`kmalloc()`**.

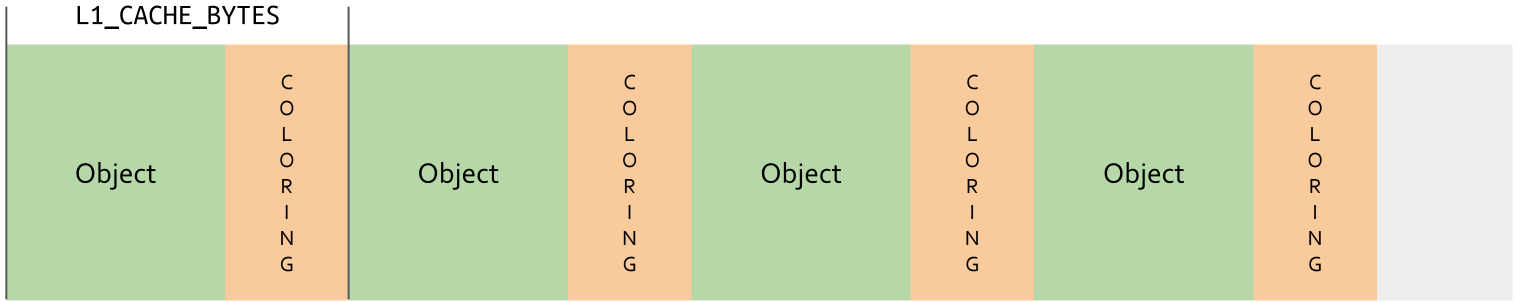
Aims

Aim #2

When a new slab is created a number of objects are packed into it and initialized using a constructor if available. When an object is free'd, it is left in a initialized state so the next allocation will be faster

Aim #3 - Coloring

If there is space left over after objects packed into a slab, the remaining space is used to color the slab. Coloring is used for having objects in different line of CPU caches which helps ensure that objects from the same slab cache will unlikely flush each other.



Caches

There is one cache for each object to be cached (see `/proc/slabinfo`).

```
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount>
<sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
inode_cache        35086  35086    608   53    8 : tunables    0    0    0 : slabdata    662    662      0
dentry            228365 228438    192   42    2 : tunables    0    0    0 : slabdata   5439   5439      0
vm_area_struct     98901  99240    200   40    2 : tunables    0    0    0 : slabdata   2481   2481      0
mm_struct            780    780   1088   30    8 : tunables    0    0    0 : slabdata     26     26      0
files_cache         1104   1104    704   46    8 : tunables    0    0    0 : slabdata     24     24      0
pid                 3424   3424    128   32    1 : tunables    0    0    0 : slabdata    107    107      0
dma-kmalloc-8k         0      0   8192    4    8 : tunables    0    0    0 : slabdata      0      0      0
...
kmalloc-rcl-8k         0      0   8192    4    8 : tunables    0    0    0 : slabdata      0      0      0
...
kmalloc-8k           436    436   8192    4    8 : tunables    0    0    0 : slabdata    109    109      0
kmalloc-4k          1376   1376   4096    8    8 : tunables    0    0    0 : slabdata    172    172      0
kmalloc-2k         14654  14928   2048   16    8 : tunables    0    0    0 : slabdata    933    933      0
kmalloc-1k          6532   6816   1024   32    8 : tunables    0    0    0 : slabdata    213    213      0
kmalloc-512        37177  37888    512   32    4 : tunables    0    0    0 : slabdata   1184   1184      0
kmalloc-256        14656  14656    256   32    2 : tunables    0    0    0 : slabdata    458    458      0
kmalloc-192        12508  12852    192   42    2 : tunables    0    0    0 : slabdata    306    306      0
kmalloc-128         3998   4256    128   32    1 : tunables    0    0    0 : slabdata    133    133      0
kmalloc-96         16884  16884     96   42    1 : tunables    0    0    0 : slabdata    402    402      0
kmalloc-64         41614  43776     64   64    1 : tunables    0    0    0 : slabdata    684    684      0
kmalloc-32         62336  62336     32  128    1 : tunables    0    0    0 : slabdata    487    487      0
kmalloc-16         39424  39424     16  256    1 : tunables    0    0    0 : slabdata    154    154      0
kmalloc-8          25600  25600      8  512    1 : tunables    0    0    0 : slabdata     50     50      0
kmem_cache_node      832    832     64   64    1 : tunables    0    0    0 : slabdata     13     13      0
kmem_cache           448    448    256   32    2 : tunables    0    0    0 : slabdata     14     14      0
```

Caches

kmem_cache_node

```
/*
 * The slab lists for all objects.
 */
struct kmem_cache_node {
	spinlock_t list_lock;

#ifdef CONFIG_SLAB
	struct list_head slabs_partial;	/* partial list first, better asm code */
	struct list_head slabs_full;
	struct list_head slabs_free;
	unsigned long total_slabs;	/* length of all slab lists */
	unsigned long free_slabs;	/* length of free slab list only */
	unsigned long free_objects;
	unsigned int free_limit;
	unsigned int colour_next;	/* Per-node cache coloring */
	struct array_cache *shared;	/* shared per node */
	struct alien_cache **alien;	/* on other nodes */
	unsigned long next_reap;	/* updated without locking */
	int free_touched;		/* updated without locking */
#endif

#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

};
```

https://elixir.bootlin.com/linux/v5.11.6/source/mm/slab.h#L525

Caches

Structure

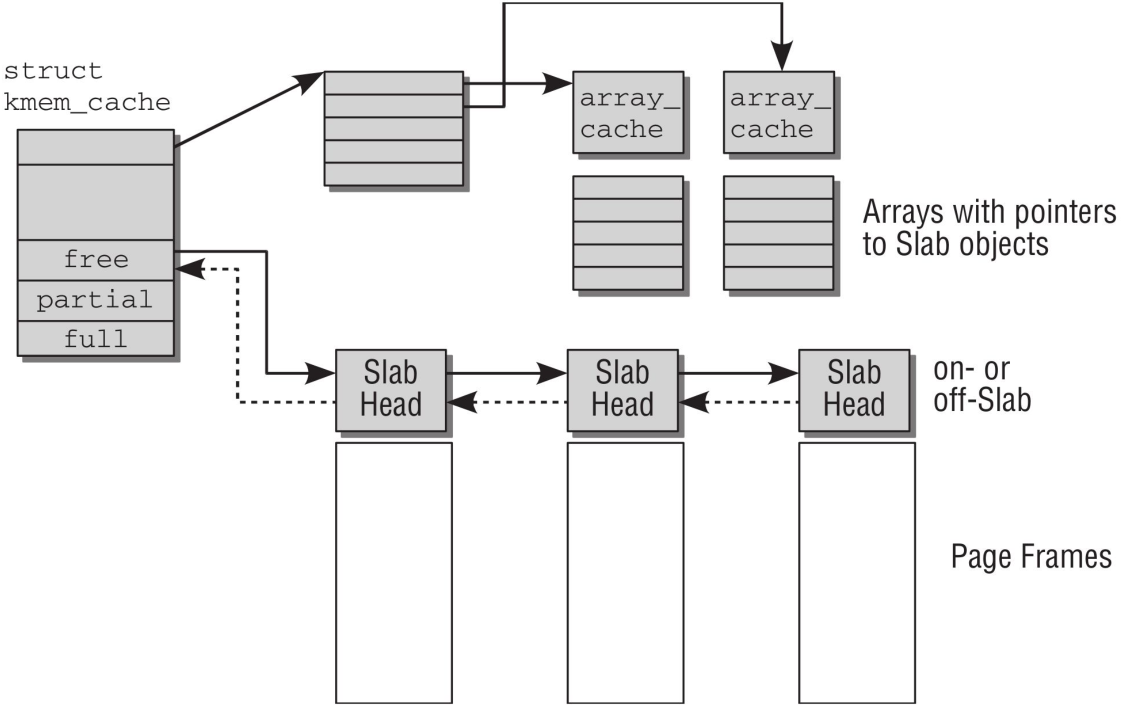


Figure 3-45: Fine structure of a slab cache.

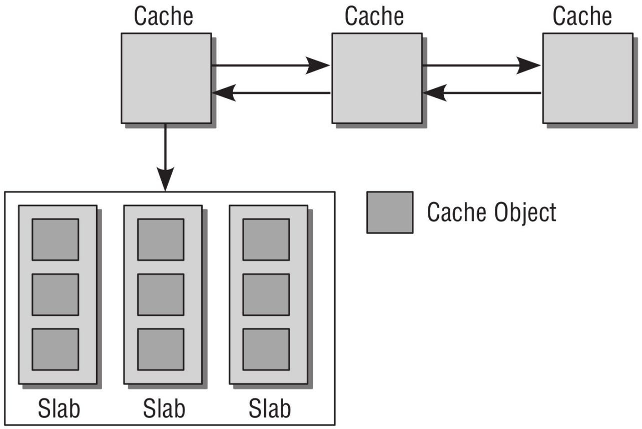


Figure 3-44: Components of the slab allocator.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

APIs

```
kmem_cache_t * kmem_cache_create(const char *name, size_t size,
size_t offset, unsigned long flags,
      void (*ctor)(void*, kmem_cache_t *, unsigned long),
     void (*dtor)(void*, kmem_cache_t *, unsigned long))
```

Creates a new cache and adds it to the cache chain.

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
```

Allocates a single object from the cache and returns it to the caller.

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

Frees an object and returns it to the cache.

```
void * kmalloc(size_t size, int flags)
```

Allocates a block of memory from one of the sizes cache.

```
void kfree(const void *objp)
```

Frees a block of memory allocated with kmalloc.

```
int kmem_cache_destroy(kmem_cache_t * cachep)
```

Destroys all objects in all slabs and frees up all associated memory before removing the cache from the chain.

Table 8.1. Slab Allocator API for Caches

Gorman, Mel. *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.

3.5.3

2. Memory Management

5. Steady-State Memory Allocation

CPU Caches

DIAG

CPU Caches

Caches lines are generally small (32/64 bits), the macro `L1_CACHE_BYTES` sets the number of bytes for the L1 cache.

Independently of the mapping scheme, close addresses fall in the same line but cache-aligned addresses fall in different lines. We need to cope with cache *performance issues at the level of kernel programming* (typically not of explicit concern for user level programming).

Performance issues

- **common members access**: most-used members in a data structure should be placed at its head to maximize cache hits. This should happen provided that the slab- allocation (`kmalloc()`) system gives cache-line aligned addresses for dynamically allocated memory chunks
- **loosely related fields** should be placed sufficiently distant in the data structure so as to avoid performance penalties due to *false cache sharing*.

The Kernel also need to face with *cache aliasing*.

(Cache False Sharing)

This example explains the Cache False Sharing problem.

Suppose that the `sum_a` and `sum_b` function run concurrently. `inc_b` modifies only the y value but doing this invalidates the cache, `sum_a` is therefore obliged to reload from memory the entire structure foo even if `f.x` will be always the same.

For this reason, *loosely related fields* should be located in the struct as much distant as possible, in order to fall in different cache lines and prevent the Cache False Sharing issue.

```c
struct foo {
    int x;
    int y;
};

static struct foo f;

/* The two following functions are running concurrently: */

int sum_a(void)
{
    int s = 0;
    for (int i = 0; i < 1000000; ++i)
        s += f.x;
    return s;
}

void inc_b(void)
{
    for (int i = 0; i < 1000000; ++i)
        ++f.y;
}
```

(Cache Aliasing)

Cache aliasing occurs when multiple mappings to a physical page of memory have conflicting caching states, such as cached and uncached. Due to these conflicting states, data in that physical page may become corrupted when the processor's cache is flushed. If that page is being used for DMA by a driver, this can lead to hardware stability problems and system lockups.

In general we have a Cache Aliasing issue when the same physical address is mapped with different virtual addresses. Therefore, if your cache is indexed by the virtual address you will load the same physical addresses multiple times. This problem is typical in ARM architectures (Source).

Cache Flush Operation

Cache flushes automation can be partial (similar to TLB), therefore there are function declared in the kernel which deal with cache flushing operations and they are implemented according to the specific architecture. In some cases, the flush operation uses the physical address of the cached data to support flushing ("strict caching systems", e.g. HyperSparc). Hence, TLB flushes should always be placed after the corresponding data cache flush calls.

Flushing Full MM	**Flushing Range**	**Flushing Page**
`flush_cache_mm()`	`flush_cache_range()`	`flush_cache_page()`
Change all page tables	Change page table range	Change single PTE
`flush_tlb_mm()`	`flush_tlb_range()`	`flush_tlb_page()`

Table 3.4. Cache and TLB Flush Ordering

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

Cache Flush APIs

`void flush_cache_all(void)`

This flushes the entire CPU cache system, which makes it the most severe flush operation to use. It is used when changes to the kernel page tables, which are global in nature, are to be performed.

`void flush_cache_mm(struct mm_struct mm)`

This flushes all entries related to the address space. On completion, no cache lines will be associated with `mm`.

`void flush_cache_range(struct mm_struct *mm, unsigned long start, unsigned long end)`

This flushes lines related to a range of addresses in the address space. Like its TLB equivalent, it is provided in case the architecture has an efficient way of flushing ranges instead of flushing each individual page.

`void flush_cache_page(struct vm_area_struct *vma, unsigned long vmaddr)`

This is for flushing a single-page-sized region. The VMA is supplied because the `mm_struct` is easily accessible through `vma→vm_mm`. Additionally, by testing for the `VM_EXEC` flag, the architecture will know if the region is executable for caches that separate the instructions and data caches. VMAs are described further in Chapter 4.

Table 3.5. CPU Cache Flush API

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

2. Memory Management

5. Steady-State Memory Allocation

Large Allocations & `vmalloc`

Large-size Allocations

It is preferable when dealing with large amounts of memory to use physically contiguous pages in memory both for cache-related and memory-access-latency reasons. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible. Linux provides a mechanism through `vmalloc()` where **non-contiguous physical memory can be used that is contiguous in virtual memory**. If you remember the Linux virtual memory layout, the area is limited (128MB).

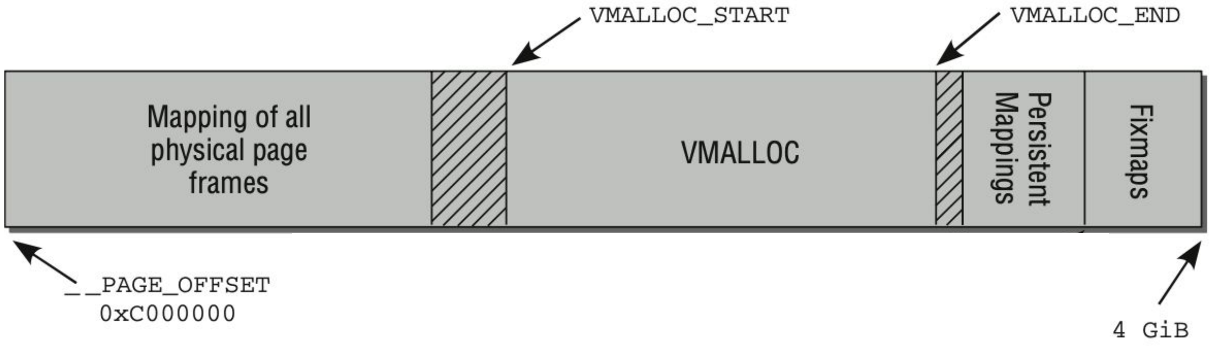


Figure 3-15: Division of the kernel address space on IA-32 systems.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Large-size Allocations

On x86, due to the limited size of the VMALLOC area, that kind of memory allocation is used sparingly, only for swap information and for mounting external kernel modules.

APIs

`void * vmalloc(unsigned long size)`
Allocates a number of pages in vmalloc space that satisfy the requested size.

`void * vmalloc_dma(unsigned long size)`
Allocates a number of pages from `ZONE_DMA`.

`void * vmalloc_32(unsigned long size)`
Allocates memory that is suitable for 32-bit addressing. This ensures that the physical page frames are in `ZONE_NORMAL`, which 32-bit devices will require

Table 7.1. Noncontiguous Memory Allocation API

`void vfree(void *addr)`
Frees a region of memory allocated with `vmalloc()`, `vmalloc_dma()` or `vmalloc_32()`

Table 7.2. Noncontiguous Memory Free API

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

`kmalloc()` vs `vmalloc()`

Allocation size:

- Bounded for kmalloc (cache aligned): the boundary depends on the architecture and the Linux version. Current implementations handle up to 8KB
- 64/128 MB for vmalloc

Physical contiguousness

- Yes for kmalloc
- No for vmalloc

Effects on TLB

- None for kmalloc
- Global for vmalloc (transparent to vmalloc users)

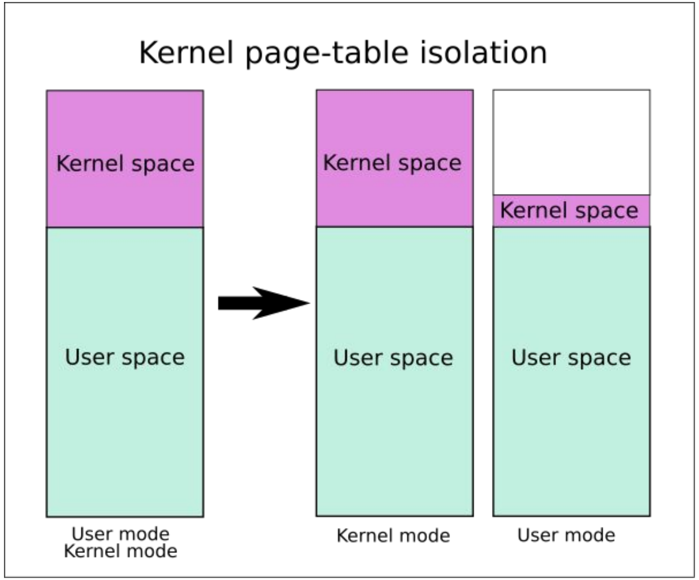
3.6

2. Memory Management

User & Kernel Space

Kernel Page Table Isolation (KPTI)

It is a protection mechanism introduced in Kernel 4.15 for facing the Meltdown vulnerability. The idea is that the Kernel address space when in user mode is reduced and contains only a small subset of pages, essential for calling the kernel facilities from user space (system calls).



https://en.wikipedia.org/wiki/Kernel_page-table_isolation
https://www.kernel.org/doc/html/latest/x86/pti.html

User/Kernel Level Data Movement

`unsigned long copy_from_user(void *to, const void *from, unsigned long n)`
Copies n bytes from the user address(`from`) to the kernel address space(`to`).

`unsigned long copy_to_user(void *to, const void *from, unsigned long n)`
Copies n bytes from the kernel address(`from`) to the user address space(`to`).

`void copy_user_page(void *to, void *from, unsigned long address)`
Copies data to an anonymous or COW page in userspace. Ports are responsible for avoiding D-cache aliases. It can do this by using a kernel virtual address that would use the same cache lines as the virtual address.

`void clear_user_page(void *page, unsigned long address)`
Similar to `copy_user_page()`, except it is for zeroing a page.

`void get_user(void *to, void *from)`
Copies an integer value from userspace (`from`) to kernel space (`to`).

`void put_user(void *from, void *to)`
Copies an integer value from kernel space (`from`) to userspace (`to`).

`long strncpy_from_user(char *dst, const char *src, long count)`
Copies a null terminated string of at most `count` bytes long from userspace (`src`) to kernel space (`dst`).

`long strlen_user(const char *s, long n)`
Returns the length, upper bound by `n`, of the userspace string including the terminating NULL.

`int access_ok(int type, unsigned long addr, unsigned long size)`
Returns nonzero if the userspace block of memory is valid and zero otherwise.

Table 4.6. Accessing Process Address Space API

Gorman, Mel. Understanding the Linux virtual memory manager. Upper Saddle River: Prentice Hall, 2004.

Advanced Operating Systems and Virtualization

[3] Memory Management

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG