

Memory Management

Intel x86 hardware

Supports segmentation over paging

(segid, offset)
| <- segmentation hardware
linear address
| <- paging hardware
physical address

Segment id is implicitly or explicitly associated with a Segment Selector register

CS - code segment (default for fetch accesses)
DS - data segment (default for non-stack data accesses)
SS - stack segment (default for stack ops (push/pop/call/ret))
ES - extra segment used by default in memory to memory copies
FS - extra segments never used by default
GS

Segment selector registers are 16 bits.

High order 13 bits provide segment descriptor index
TI bit indicates GDT or LDT
Low order two bits contain RPL / CPL (request / current) privilege level

Actual attributes of a segment are contained in the *segment descriptor*

Segment descriptors are 64 bits (8 bytes in size)
Reside in Global or Local descriptor tables of up to 8192 entries
Entry 0 always represents an invalid segment

Segment descriptors contain

32 bit base *virtual or linear* address of segment
20 bit size of the segment in bytes or pages
G (granularity) flag specifies bytes or pages
S (system) flag
4 bit type flag
 data
 code
 TSS
 LDT

Selector registers have an "invisible" extension that holds descriptor data

Validity checking is done when the selector register is loaded

Main Memory need not be accessed again to refer to segment attributes

Segmentation in Linux

Linux makes minimal use of segmentation

No LDT is used

GDT is as shown.

```
6 /*
7 * The layout of the GDT under Linux:
8 *
9 * 0 - null
10 * 1 - not used
11 * 2 - kernel code segment
12 * 3 - kernel data segment
13 * 4 - user code segment          <-- new cacheline
14 * 5 - user data segment
15 * 6 - not used
16 * 7 - not used
17 * 8 - APM BIOS support          <-- new cacheline
18 * 9 - APM BIOS support
19 * 10 - APM BIOS support
20 * 11 - APM BIOS support
21 *
22 * The TSS+LDT descriptors are spread so that every CPU
23 * has an exclusive cacheline for the per-CPU TSS and LDT:
24 *
25 * 12 - CPU#0 TSS                 <-- new cacheline
26 * 13 - CPU#0 LDT
27 * 14 - not used
28 * 15 - not used
29 * 16 - CPU#1 TSS                 <-- new cacheline
30 * 17 - CPU#1 LDT
31 * 18 - not used
32 * 19 - not used
33 * ... NR_CPUS per-CPU TSS+LDT's if on SMP
```

Descriptor attributes

Kernel and user code and data use

Base = 0x0000000
Limit = 0xffff
G = 1 (page size units)
D/B = 1 (32 bit offsets)

Kernel and user code use

type = 0x0a (readable code)

Kernel and user data use

type = 0x02

Kernel uses

S = 1
DPL = 0

User uses

S = 0
DPL = 3

Book refers to TSS per process.. this was discontinued in kernel 2.4

Intel x86 Paging

Pages are 4K bytes in size

Linear (virtual addresses) are 32 bits long

10 bits - Page directory index

10 bits - Page table index

12 bits - Offset into page

Page table and directory entries

20 bits - page frame address (20 bits used since frames are aligned)

P flag - is page present

A flag - has paged been accessed

D flag - has page been written

R flag - page is readonly

U flag - user mode page (U = 0 -> access by PL 3 is verboten)

S flag - pages may be 4K or 4 MB in size

PCD - disable hardware caching

PWT - use write through instead of copy back.

(both default to 0.. but can be reset as required)

Linux Paging

Intel hardware supports two level page mapping scheme

Linux software supports three level scheme (for 64 bit architectures)

Page global directory (pgd)

Page middle directory (pmd)

Page table (pt)

In 32 bit systems the pmd layer is effectively null.

Page table and page directory entries are 32 bits each

There are 1024 entries in each table

Thus page directories and page tables are 4K in size and 4K aligned

Permanently reserved page frames

Page 0 - used by BIOS to store config data
Page 0xa0 - 0xff more bios stuff

Kernel code and data starts at page 0x100 (1 MB) and is delimited by:

<code>_text</code>	start of code
<code>_etext</code>	end of code / start of initialized data
<code>_edata</code>	end of initialized data / start of un-init data.
<code>_end</code>	end of the kernel

From System.map

<code>c0100000</code>	A	<code>_text</code>
<code>c026ad91</code>	A	<code>_etext</code>
<code>c02e77a0</code>	A	<code>_edata</code>
<code>c0368e58</code>	A	<code>_end</code>

Hence this kernel is 0x268e58 in size --- slightly less than 2.5 MB.

Additional memory is also permanently allocated during system initialization.

Kernel memory mapping

While the kernel is loaded *physically* at the 1MB line, it is mapped *virtually* at location 0xc01000000 (the 3 GB + 1 MB line).

The value of this offset is stored in PAGE_OFFSET which is 0xc000000 for Intel

Address spaces

Each address space (heavyweight process) has a page directory

The kernel area is mapped by a single set of page tables

Pointers to these tables are install beginning at offset 768 inside the page directory

Real storage management

Each page frame is represented by the following structure
These are all allocated in a global array called `mem_map[]`
The array can be indexed by page number.

```
151 typedef struct page {
152     struct list_head list;          /* ->mapping has some page lists. */
153     struct address_space *mapping; /* The inode (or ...) we belong to. */
154     unsigned long index;           /* Our offset within mapping. */
155     struct page *next_hash;        /* Next page sharing our hash bucket in
156                                     the pagecache hash table. */
157     atomic_t count;                /* Usage count, see below. */
158     unsigned long flags;           /* atomic flags, some possibly updated
159                                     asynchronously */
160     struct list_head lru;          /* Pageout list, eg. active_list;
161                                     protected by pagemap_lru_lock !! */
162     wait_queue_head_t wait;        /* Page locked? Stand in line... */
163     struct page **pprev_hash;      /* Complement to *next_hash. */
164     struct buffer_head * buffers; /* Buffer maps us to a disk block. */
165     void *virtual;                 /* Kernel virtual address (NULL if
166                                     not kmapped, ie. highmem) */
167     struct zone_struct *zone;      /* Memory zone we are in. */
168 } mem_map_t;
```

A further complication is that real memory is partitioned into three *zones*.

ISA DMA area	0x0 - 0xffff
Normal	0x100000 -0x37ffff (896 MB)
High memory	> 896MB

Linux uses the *Buddy System* in real storage management

Buddy system was designed as a good compromise between

Efficient operation of allocation/free
Avoidance of fragmentation of physical memory

Why worry about fragmented physical memory in a virtual environment
-> some graphics cards need large DMA areas

That problem *could* be addressed in other ways (the *big_phys_area*) hack

Free storage within each zone is mapped by one of *MAX_ORDER* (10) *free area* structures
The particular structure used depends upon whether a free page belongs to a block of:

<i>size</i>	<i>alignment</i>
1 page	4K aligned
2 pages	8K aligned
4 pages	16K aligned
:	
512 pages	8 MB aligned

```
21 typedef struct free_area_struct {
22     struct list_head    free_list;
23     unsigned long       *map;
24 } free_area_t;
25
```

The *free_list* field points to a list of *struct page* where each is the first free page of a free block

The *map* field is bitmap identifying states of buddies within the the entire zone

0 => both buddies free or both buddies allocated
1 => exactly one buddy free and one buddy allocated

The number of bits in the bitmap is equal to (size-of-zone) / (size-of-page * 2 ^ (order + 1))

Suppose there was exactly 1 MB of memory

There are $2^{20} / 2^{12} = 2^8$ pages

Order	block size	sets of buddies (bits)
0	4K	2^7
1	8K	2^6
2	16K	2^5
3	32K	2^4
4	64K	2^3
5	128K	2^2
6	256K	2^1
7	512K	2^0

When a page is freed it is a simple matter to convert its frame number to a bit offset and thus determine if the buddy is free is also free.

Zone mangagement

free_pages counts the number of available pages within the entire zone.
pages_min, *pages_low*, and *pages_high* drive the page stealing algorithm
need_balance is a flag indicating that the zone needs pages

```
36 typedef struct zone_struct {
37 /*
38 * Commonly accessed fields:
39 */
40 spinlock_t          lock;
41 unsigned long       free_pages;
42 unsigned long       pages_min, pages_low, pages_high;
43 int                 need_balance;
44
45 /*
46 * free areas of different sizes
47 */
48 free_area_t free_area[MAX_ORDER];
49
50 /*
51 * Discontig memory support fields.
52 */
53 struct pglst_data   *zone_pgdat;
54 struct page         *zone_mem_map;
55 unsigned long       zone_start_paddr;
56 unsigned long       zone_start_mapnr;
57
58
59 * rarely used fields:
60 */
61 char                *name;
62 unsigned long       size;
63 } zone_t;
```

This structure represents a layer above the zone structure that was introduced to support NUMA
A non-NUMA system consists of a single node with three zones

```
99 typedef struct pglst_data {
100 zone_t node_zones[MAX_NR_ZONES];
101 zonelist_t node_zonelists[GFP_ZONEMASK+1];
102 int nr_zones;
103 struct page *node_mem_map;
104 unsigned long *valid_addr_bitmap;
105 struct bootmem_data *bdata;
106 unsigned long node_start_paddr;
107 unsigned long node_start_mapnr;
108 unsigned long node_size;
109 int node_id;
110 struct pglst_data *node_next;
111 } pg_data_t;
112
```

Pageable memory is setup via the *free_area_init_core()* function

By now a considerable amount of initialization has already completed and the number of pages in each zone is should be known.

```
631 /*
632 * Set up the zone data structures:
633 *   - mark all pages reserved
634 *   - mark all memory queues empty
635 *   - clear the memory bitmaps
636 */
637 void __init free_area_init_core(
        int nid, /* 0 */
        pg_data_t *pgdat,
        struct page **gmap,
        unsigned long *zones_size,
        unsigned long zone_start_paddr, /* 0 */
        unsigned long *zholes_size, /* 0 */
        struct page *lmem_map) /* 0 */
640 {
641     struct page *p;
642     unsigned long i, j;
643     unsigned long map_size;
644     unsigned long totalpages, offset, realtotalpages;
645     const unsigned long zone_required_alignment = 1UL <<
        (MAX_ORDER-1);
646
647     if (zone_start_paddr & ~PAGE_MASK)
648         BUG();
649
650     totalpages = 0;
651     for (i = 0; i < MAX_NR_ZONES; i++) {
652         unsigned long size = zones_size[i];
653         totalpages += size;
654     }
655     realtotalpages = totalpages;
656     if (zholes_size)
657         for (i = 0; i < MAX_NR_ZONES; i++)
658             realtotalpages -= zholes_size[i];
659
660     printk("On node %d totalpages: %lu\n", nid, realtotalpages);
661
```

This information is printed during the boot sequence... hence total pages includes reserved and unreserved categories.

```
1253 Mar 21 17:16:34 waco kernel: On node 0 totalpages: 32704
1254 Mar 21 17:16:34 waco kernel: zone(0): 4096 pages.
1255 Mar 21 17:16:34 waco kernel: zone(1): 28608 pages.
1256 Mar 21 17:16:34 waco kernel: zone(2): 0 pages.
```

The *mem_map* table of *struct page* is allocated here via a special purpose low level allocator *alloc_bootmem_node()*

```
662 INIT_LIST_HEAD(&active_list);
663 INIT_LIST_HEAD(&inactive_list);
664
665 /*
666 * Some architectures (with lots of mem and discontinous memory
667 * maps) have to search for a good mem_map area:
668 * For discontigmem, the conceptual mem map array starts from
669 * PAGE_OFFSET, we need to align the actual array onto a mem map
670 * boundary, so that MAP_NR works.
671 */
672 map_size = (totalpages + 1)*sizeof(struct page);
673 if (lmem_map == (struct page *)0) {
674     lmem_map = (struct page *) alloc_bootmem_node(pgdat,
675         map_size);
676     lmem_map = (struct page *) (PAGE_OFFSET +
677         MAP_ALIGN((unsigned long)lmem_map - PAGE_OFFSET));
677 }
```

**gmap* is actually an alias here for the global variable *mem_map*.. ain't C wonderful!

```
678 *gmap = pgdat->node_mem_map = lmem_map;
679 pgdat->node_size = totalpages;
680 pgdat->node_start_paddr = zone_start_paddr;
681 pgdat->node_start_mapnr = (lmem_map - mem_map);
682 pgdat->nr_zones = 0;
683
```

Flag all pages initially reserved. They get unreserved at end of boot.

```
684 /*
685 * Initially all pages are reserved - free ones are freed
686 * up by free_all_bootmem() once the early boot process is
687 * done.
688 */
689 for (p = lmem_map; p < lmem_map + totalpages; p++) {
690     set_page_count(p, 0);
691     SetPageReserved(p);
692     init_waitqueue_head(&p->wait);
693     memlist_init(&p->list);
694 }
```

Initialize zone data structures for all zones.

```
696 offset = lmem_map - mem_map;
697 for (j = 0; j < MAX_NR_ZONES; j++) {
698     zone_t *zone = pgdat->node_zones + j;
699     unsigned long mask;
700     unsigned long size, realsize;
701
702     realsize = size = zones_size[j];
703     if (zholes_size)
704         realsize -= zholes_size[j];
705
706     printk("zone(%lu): %lu pages.\n", j, size);
707     zone->size = size;
708     zone->name = zone_names[j];
709     zone->lock = SPIN_LOCK_UNLOCKED;
710     zone->zone_pgdat = pgdat;
711     zone->free_pages = 0;
712     zone->need_balance = 0;
713     if (!size)
714         continue;
715
716     pgdat->nr_zones = j+1;
717
```

Initialize the "water marks" used to drive page stealing.

Balance ratios are set to {128, 128, 128}
realsize is the size of the zone in pages - any holes.
Balance mins are set to {20, 20, 20}
Balance maxes are set to {255, 255, 255}

Suppose a region has 64 MB.

Then $realsize = 2^{26} / 2^{12} = 2^{14}$

$mask = 2^{14} / 2^7 = 2^7$

$pages_min = 2^7$

$pages_low = 2^8$

$pages_high = 384$.

```
718     mask = (realsize / zone_balance_ratio[j]);
719     if (mask < zone_balance_min[j])
720         mask = zone_balance_min[j];
721     else if (mask > zone_balance_max[j])
722         mask = zone_balance_max[j];
723     zone->pages_min = mask;
724     zone->pages_low = mask*2;
725     zone->pages_high = mask*3;
726
727     zone->zone_mem_map = mem_map + offset;
728     zone->zone_start_mapnr = offset;
729     zone->zone_start_paddr = zone_start_paddr;
730
731     if ((zone_start_paddr >> PAGE_SHIFT) &
732         (zone_required_alignment-1))
733         printk("BUG: wrong zone alignment, it will crash\n");
734
735     for (i = 0; i < size; i++) {
736         struct page *page = mem_map + offset + i;
737         page->zone = zone;
738         if (j != ZONE_HIGHMEM)
739             page->virtual = __va(zone_start_paddr);
740         zone_start_paddr += PAGE_SIZE;
741     }
742
130 #define __pa(x)    ((unsigned long)(x)-PAGE_OFFSET)
131 #define __va(x)    ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

Initialize the buddy system structures here. *size* is expressed in pages.

```
742     offset += size;
743     for (i = 0; i i++) {
744         unsigned long bitmap_size;
745
746         memlist_init(&zone->free_area[i].free_list);
747         if (i == MAX_ORDER-1) {
748             zone->free_area[i].map = NULL;
749             break;
750         }
751
752         /*
753          * Page buddy system uses "index >> (i+1)",
754          * where "index" is at most "size-1".
755          *
756          * The extra "+3" is to round down to byte
757          * size (8 bits per byte assumption). Thus
758          * we get "(size-1) >> (i+4)" as the last byte
759          * we can access.
760          *
761          * The "+1" is because we want to round the
762          * byte allocation up rather than down. So
763          * we should have had a "+7" before we shifted
764          * down by three. Also, we have to add one as
765          * we actually use the last bit (it's [0,n]
766          * inclusive, not [0,n[).
767          *
768          * So we actually had +7+1 before we shift
769          * down by 3. But (n+8) >> 3 == (n >> 3) + 1
770          * (modulo overflows, which we do not have).
771          *
772          * Finally, we LONG_ALIGN because all bitmap
773          * operations are on longs.
774          */
775         bitmap_size = (size-1) >> (i+4);
776         bitmap_size = LONG_ALIGN(bitmap_size+1);
777         zone->free_area[i].map =
778             (unsigned long *) alloc_bootmem_node(pgdat,
779                 bitmap_size);
780     }
781     build_zonelists(pgdat);
```

build_zonelists() doesn't build the *free_lists*.

It builds zone eligibility / preference lists associated with allocation flags such as *GFP_DMA*

Real memory allocation.

`__get_free_pages()` is the internal entry point to the buddy system allocator
The order is the $\log_2(\#pages)$ required to satisfy the request.

GFP_MASK can be set to

```
522 /* Zone modifiers in GFP_ZONEMASK (see linux/mmzone.h - low four bits) */
523 #define __GFP_DMA 0x01
524 #define __GFP_HIGHMEM 0x02
525
526 /* Action modifiers - doesn't change the zoning */
527 #define __GFP_WAIT 0x10 /* Can wait and reschedule? */
528 #define __GFP_HIGH 0x20 /* Should access emergency pools? */
529 #define __GFP_IO 0x40 /* Can start low memory physical IO? */
530 #define __GFP_HIGHIO 0x80 /* Can start high mem physical IO? */
531 #define __GFP_FS 0x100 /* Can call down to low-level FS? */
532
533 #define GFP_NOHIGHIO (__GFP_HIGH | __GFP_WAIT | __GFP_IO)
534 #define GFP_NOIO (__GFP_HIGH | __GFP_WAIT)
535 #define GFP_NOFS (__GFP_HIGH | __GFP_WAIT | __GFP_IO | __GFP_HIGHIO)
536 #define GFP_ATOMIC (__GFP_HIGH)
537 #define GFP_USER (__GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS)
538 #define GFP_HIGHUSER (__GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS | __GFP_HIGHMEM)
539 #define GFP_KERNEL (__GFP_HIGH | __GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS)
540 #define GFP_NFS (__GFP_HIGH | __GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS)
541 #define GFP_KSWAPD (__GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS)
542
```

```
406 unsigned long __get_free_pages(unsigned int gfp_mask,
                                unsigned int order)
407 {
408     struct page * page;
409
410     page = alloc_pages(gfp_mask, order);
411     if (!page)
412         return 0;
413     return (unsigned long) page_address(page);
414 }
```

`alloc_pages` is a front for the usual insulating layers

```
361 static inline struct page * alloc_pages(unsigned int gfp_mask,
                                           unsigned int order)
362 {
363     /*
364     * Gets optimized away by the compiler.
365     */
366     if (order >= MAX_ORDER)
367         return NULL;
368     return _alloc_pages(gfp_mask, order);
369 }
```

Each `pg_data` structure contains a table of 15 zone lists

A zone list is a table of pointers to zone structures

Zone lists are predefined to include zone pointers to legal/preferred zones for each request type.

```
65 #define ZONE_DMA      0
66 #define ZONE_NORMAL  1
67 #define ZONE_HIGHMEM 2
68 #define MAX_NR_ZONES 3
69
70 /*
71 * One allocation request operates on a zonelist. A zonelist
72 * is a list of zones, the first one is the 'goal' of the
73 * allocation, the other zones are fallback zones, in decreasing
74 * priority.
75 *
76 * Right now a zonelist takes up less than a cacheline. We never
77 * modify it apart from boot-up, and only a few indices are used,
78 * so despite the zonelist table being relatively big, the cache
79 * footprint of this construct is very small.
80 */
81 typedef struct zonelist_struct {
82     zone_t * zones [MAX_NR_ZONES+1]; // NULL delimited
83 } zonelist_t;
84
85 #define GFP_ZONEMASK 0x0f
86
```

This wrapper uses the `gfp_mask` to identify the correct zone list containing legal/preferred zones for each request type.

```
221 struct page *_alloc_pages(unsigned int gfp_mask, unsigned int
                        order)
222 {
223     return __alloc_pages(gfp_mask, order,
224         contig_page_data.node_zonelist+(gfp_mask & GFP_ZONEMASK));
225 }
```

The actual allocation is done here.

```
305 struct page * __alloc_pages(unsigned int gfp_mask,
                               unsigned int order, zonelist_t *zonelist)
306 {
307     unsigned long min;
308     zone_t **zone, * classzone;
309     struct page * page;
310     int freed;
311
```

For each zone in the suitable zone list,

See if the number of *free_pages* minus the number requested is still greater than *pages_low* watermark of the zone.

If so, the actual allocation is done by *rmqueue*.

The most preferred zone is remembered as the *classzone*.

```
312 zone = zonelist->zones;
313 classzone = *zone;
314 min = 1UL << order;
315 for (;;) {
316     zone_t *z = *(zone++);
317     if (!z)
318         break;
319
320     min += z->pages_low;
321     if (z->free_pages > min) {
322         page = rmqueue(z, order);
323         if (page)
324             return page;
325     }
326 }
327
```

Arrival here means that there were not enough available pages. In this case:

Mark the zone as needing to be replenished with pages and
Wakeup the the page stealer (kswapd) to steal some.

```
328 classzone->need_balance = 1;
329 mb();
330 if (waitqueue_active(&kswapd_wait))
331     wake_up_interruptible(&kswapd_wait);
332
333 zone = zonelist->zones;
334 min = 1UL << order;
335 for (;;)
336 {
337     unsigned long local_min;
338     zone_t *z = *(zone++);
339     if (!z)
340         break;
341 }
```

Here we test against *pages_min* instead of *pages_low*..

The value of *min* is the pages wanted.

The value of *local_min* reflects the number of free pages that must remain after the alloc.

It will either be *pages_min* or *pages_min* / 4 depending on `__GFP_WAIT`

Suppose *pages_min* = 20 and *min* = 4.

Then *local_min* is first set to 20

If this is a no waiting request it is further reduced to 5

So *min* + *local_min* is equal to either 9 or 25.

Thus the allocation will be made here iff *free_pages* > 9 or 25.

```
341     local_min = z->pages_min;
342     if (!(gfp_mask & __GFP_WAIT))
343         local_min >>= 2;
344     min += local_min;
345     if (z->free_pages > min) {
346         page = rmqueue(z, order);
347         if (page)
348             return page;
349     }
350 }
351 }
```

As it says if we get here we are *really* low on memory...

If process flags require it, (under what conditions are the flags set???) we go through the zone list one more time and this time take *anything* we can find.

```
352 /* here we're in the low on memory slow path */
353
354 rebalance:
355 if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {
356     zone = zonelist->zones;
357     for (;;) {
358         zone_t *z = *(zone++);
359         if (!z)
360             break;
361
362         page = rmqueue(z, order);
363         if (page)
364             return page;
365     }
366     return NULL;
367 }
368
```

Arrival here means the request just can't be satisfied now.

For atomic requests made by interrupt handlers that can't sleep, it's necessary to bail out now.

```
369 /* Atomic allocations - we can't balance anything */
370 if (!(gfp_mask & __GFP_WAIT))
371     return NULL;
372
```

The variable *classzone* still points to the original target.

We try to replenish it by stealing some stuff here... balancing may return us the memory we need.

```
373 page = balance_classzone(classzone, gfp_mask, order, &freed);
374 if (page)
375     return page;
376
```

Make yet another pass over the zonelist.

This pass requires *pages_min* remain after the allocation

It doesn't contain the *local_min* hack because we can't get here with `__GFP_WAIT`

```
377 zone = zonelist->zones;
378 min = 1UL << order;
379 for (;;) {
380     zone_t *z = *(zone++);
381     if (!z)
382         break;
383
384     min += z->pages_min;
385     if (z->free_pages > min) {
386         page = rmqueue(z, order);
387         if (page)
388             return page;
389     }
390 }
391
392 /* Don't let big-order allocations loop */
393 if (order > 3)
394     return NULL;
395
396 /* Yield for kswapd, and try again */
397 current->policy |= SCHED_YIELD;
398 __set_current_state(TASK_RUNNING);
399 schedule();
400 goto rebalance;
401 }
```

rmqueue performs the mechanics of removing the free block and partitioning it if necessary. The input parameter *order* is the number of pages needed to satisfy the request..

```
175 static struct page * rmqueue(
        zone_t *zone,
        unsigned int order)
176 {
177     free_area_t * area = zone->free_area + order;
178     unsigned int curr_order = order;
179     struct list_head *head, *curr;
180     unsigned long flags;
181     struct page *page;
182
183     spin_lock_irqsave(&zone->lock, flags);
184     do {
```

area points to the *free area* struct associated with the target allocation order in the target zone

```
185         head = &area->free_list;
186         curr = memlist_next(head);
187
```

If (*curr* == *head*) *this* free list is empty..

We may (or may not) know that adequate memory exists in this *zone*, but we never know if it resides in this order or a higher order list.

```
188         if (curr != head) {
189             unsigned int index;
190
```

Now we have found the memory area we will use..

The free list is made up of page descriptors for *the first page* in the block only

Thus by deleting the entry we effectively delete 2^{order} pages ..

The ones we don't need must go back on the free lists of lower orders.

```
191         page = memlist_entry(curr, struct page, list);
192         if (BAD_RANGE(zone,page))
193             BUG();
194         memlist_del(curr);
195         index = page - zone->zone_mem_map;
```

MARK_USED is a macro used to update the bitmap associated with the area.

There is no recombining that takes place in the top order, and thus the bitmap is not relevant

The expand function is used to reallocate the pages that were not used on lists of lower

order. If we use a block of size 8 free pages to satisfy a one page request, we create one new free block of order 1, 2, and 4.

```
196         if (curr_order != MAX_ORDER-1)
197             MARK_USED(index, curr_order, area);
198         zone->free_pages -= 1UL << order;
199
200         page = expand(zone, page, index, order, curr_order,
201                     area);
202
203         set_page_count(page, 1);
204         if (BAD_RANGE(zone,page))
205             BUG();
206         if (PageLRU(page))
207             BUG();
208         if (PageActive(page))
209             BUG();
210         return page;
211     }
212     curr_order++;
213     area++;
214 } while (curr_order < MAX_ORDER);
215 spin_unlock_irqrestore(&zone->lock, flags);
216
217 return NULL;
218 }
```

This routine fractures high order blocks to satisfy lower order allocations
Note that the page used to satisfy the allocation comes from the *end* of the fractured block.

```
159 static inline struct page * expand (
        zone_t *zone,
        struct page *page,
160     unsigned long index,
        int low,
        int high,
        free_area_t * area)
161 {
162     unsigned long size = 1 << high;
163
164     while (high > low) {
165         if (BAD_RANGE(zone,page))
166             BUG();
167         area--;
168         high--;
169         size >>= 1;
170         memlist_add_head(&(page)->list, &(area)->free_list);
171         MARK_USED(index, high, area);
172         index += size;
173         page += size;
174     }
175     if (BAD_RANGE(zone,page))
176         BUG();
177     return page;
178 }

156 #define MARK_USED(index, order, area) \
157 __change_bit((index) >> (1+(order)), (area)->map)
90 static __inline__ void __change_bit(int nr, volatile void * addr)
91 {
92     __asm__ __volatile__(
93     "btcl %1,%0"
94     : "=m" (ADDR)
95     : "Ir" (nr));
96 }
```

Efficient management of kernel memory objects

The kernel often needs to keep large arrays of data structures having fixed size but...

The fixed size is not often page size.

Having to interact with the buddy system for each entity allocated

wastes space (esp. if the objects are much smaller than page size)

wastes time since the buddy system is not esp. efficient.

The slab allocator introduced in Sun's *Solaris* provides a solution

The *slab allocator* pre-allocates *caches*

Each *cache* contains one or more *slabs*

Each *slab* holds multiple objects

The buddy system gets involved only when slabs are allocated or freed

`/proc/slabinfo` provides the state of the slab allocator.

cache-name, num-active-objs, total-objs, object size

num-active-slabs, total-slabs, num-pages-per-slab

Name	Active obj	#obj	Sz Obj	ActSlb	#Slb	#Pg		
kmem_cache	68	68	232	4	4	1	:	252 126
clip_arp_cache	0	0	128	0	0	1	:	252 126
ip_contrack	172	253	352	22	23	1	:	124 62
tcp_tw_bucket	160	160	96	4	4	1	:	252 126
tcp_bind_bucket	339	339	32	3	3	1	:	252 126
tcp_open_request	118	118	64	2	2	1	:	252 126
inet_peer_cache	177	177	64	3	3	1	:	252 126
ip_fib_hash	15	226	32	2	2	1	:	252 126
ip_dst_cache	286	288	160	12	12	1	:	252 126
arp_cache	150	150	128	5	5	1	:	252 126
uhci_urb_priv	1	67	56	1	1	1	:	252 126
blkdev_requests	5376	5400	96	135	135	1	:	252 126
nfs_read_data	117	150	384	12	15	1	:	124 62
nfs_write_data	80	80	384	8	8	1	:	124 62
nfs_page	320	320	96	8	8	1	:	252 126
dnotify cache	0	0	20	0	0	1	:	252 126
file lock cache	126	126	92	3	3	1	:	252 126
fasync cache	1	202	16	1	1	1	:	252 126
uid_cache	133	226	32	2	2	1	:	252 126
skbuff_head_cache	426	552	160	23	23	1	:	252 126
sock	165	387	832	36	43	2	:	124 62
sigqueue	261	261	132	9	9	1	:	252 126
cdev_cache	3783	3835	64	65	65	1	:	252 126
bdev_cache	10974	10974	64	186	186	1	:	252 126
inode_cache	273084	273952	480	34240	34244	1	:	124 62
dentry_cache	275238	276750	128	9225	9225	1	:	252 126
filp	2613	2640	96	66	66	1	:	252 126
names_cache	12	12	4096	12	12	1	:	60 30
buffer_head	24748	25440	96	636	636	1	:	252 126
mm_struct	210	210	128	7	7	1	:	252 126
vm_area_struct	1471	1770	64	30	30	1	:	252 126
fs_cache	236	236	64	4	4	1	:	252 126
files_cache	117	117	416	13	13	1	:	124 62
signal_act	99	99	1312	33	33	1	:	60 30

The following general purpose slabs are used for *kmalloc()* requests

Name	Active	obj	#obj	Sz	Obj	ActSlb	#Slb	#Pg		
size-131072(DMA)	0	0	0	131072	0	0	32	:	0	0
size-131072	0	0	0	131072	0	0	32	:	0	0
size-65536(DMA)	0	0	0	65536	0	0	16	:	0	0
size-65536	1	1	1	65536	1	1	16	:	0	0
size-32768(DMA)	0	0	0	32768	0	0	8	:	0	0
size-32768	0	0	0	32768	0	0	8	:	0	0
size-16384(DMA)	0	0	0	16384	0	0	4	:	0	0
size-16384	0	0	0	16384	0	0	4	:	0	0
size-8192(DMA)	2	2	2	8192	2	2	2	:	0	0
size-8192	1	1	1	8192	1	1	2	:	0	0
size-4096(DMA)	0	0	0	4096	0	0	1	:	60	30
size-4096	29	29	29	4096	29	29	1	:	60	30
size-2048(DMA)	0	0	0	2048	0	0	1	:	60	30
size-2048	114	114	114	2048	57	57	1	:	60	30
size-1024(DMA)	2	4	4	1024	1	1	1	:	124	62
size-1024	212	216	216	1024	54	54	1	:	124	62
size-512(DMA)	0	0	0	512	0	0	1	:	124	62
size-512	288	288	288	512	36	36	1	:	124	62
size-256(DMA)	0	0	0	256	0	0	1	:	252	126
size-256	300	300	300	256	20	20	1	:	252	126
size-128(DMA)	0	0	0	128	0	0	1	:	252	126
size-128	1110	1110	1110	128	37	37	1	:	252	126
size-64(DMA)	0	0	0	64	0	0	1	:	252	126
size-64	1003	1003	1003	64	17	17	1	:	252	126
size-32(DMA)	0	0	0	32	0	0	1	:	252	126
size-32	4859	4859	4859	32	43	43	1	:	252	126

Each of these caches is described by the *kmem_cache_s* structure

The three lists describe slabs that are
full of active objects
have allocated objects and free slots
have no allocated objects at all.

```
189 struct kmem_cache_s {
190 /* 1) each alloc & free */
191 /* full, partial first, then free */
192 struct list_head      slabs_full;
193 struct list_head      slabs_partial;
194 struct list_head      slabs_free;
195 unsigned int          objsize;
196 unsigned int          flags; /* constant flags */
197 unsigned int          num; /* # of objs per slab */
198 spinlock_t            spinlock;

203 /* 2) slab additions /removals */
204 /* order of pgs per slab (2^n) */
205 unsigned int          gfporder;
206
207 /* force GFP flags, e.g. GFP_DMA */
208 unsigned int          gfpflags;
209
```

Each slab is described by the *slab_t* structure

s_mem addresses are *virtual*
free is an object index
These are *somehow* used to chain free objects together.

```
147 /*
148 * slab_t
149 *
150 * Manages the objs in a slab. Placed either at the beginning of
    mem allocated
151 * for a slab, or allocated from an general cache.
152 * Slabs are chained into three list: fully used, partial, fully
    free slabs.
153 */
154 typedef struct slab_s {
155 struct list_head list;
156 unsigned long    colouroff;
157 void             *s_mem; /* including colour offset */
158 unsigned int     inuse; /* num of objs active in slab */
159 kmem_bufctl_t    free; /* This is an int "index" */
160 } slab_t;
16
```

The *kmem_cache_s* structure used to describe a cache is itself allocated from the *cache_cache* which is statically initialized as follows.

```
355 /* internal cache of cache description objs */
356 static kmem_cache_t cache_cache = {
357     slabs_full:      LIST_HEAD_INIT(cache_cache.slabs_full),
358     slabs_partial:  LIST_HEAD_INIT(cache_cache.slabs_partial),
359     slabs_free:     LIST_HEAD_INIT(cache_cache.slabs_free),
360     objsize:        sizeof(kmem_cache_t),
361     flags:          SLAB_NO_REAP,
362     spinlock:       SPIN_LOCK_UNLOCKED,
363     colour_off:     L1_CACHE_BYTES,
364     name:           "kmem_cache",
365 };
366
```

The *slab_s* structures used to describe a slab contain not only the data items shown in the structure itself but also a table of *kmem_buf_ctl_t* items (which are actually ints) with one entry per object in the slab. Thus the slab control structure is variable length.

Allocate a new object from an existing cache

```
1318 static inline void * __kmem_cache_alloc(  
        kmem_cache_t *cachep, int flags)  
1319 {  
1320     unsigned long save_flags;  
1321     void* objp;  
1322
```

Ensures SLAB_DMA bit of the cachep->gfpflags is consistent with the GFP_DMA setting of flags.
The *try_again* tag is used if we are forced to allocate a new slab to the cache.

```
1323     kmem_cache_alloc_head(cachep, flags);  
1324 try_again:  
1325     local_irq_save(save_flags);  
1326 #ifdef CONFIG_SMP  
        :  
1346 #else  
1347     objp = kmem_cache_alloc_one(cachep);  
  
1263 #define kmem_cache_alloc_one(cachep)  
1264 ({  
1265     struct list_head * slabs_partial, * entry;  
1266     slab_t *slabp;  
1267
```

See if there are any partially allocated slabs in this cache

```
1268     slabs_partial = &(cachep)->slabs_partial;  
1269     entry = slabs_partial->next;
```

The standard list manager is used so that `entry == slabs_partial` is true only if the `slabs_partial` list is empty. If so see if there are any completely free slabs in this cache.

```
1270     if (unlikely(entry == slabs_partial)) {  
1271         struct list_head * slabs_free;  
1272         slabs_free = &(cachep)->slabs_free;  
1273         entry = slabs_free->next;
```

If no completely free slabs either, then it will be necessary to allocate a new slab

```
1274         if (unlikely(entry == slabs_free))  
1275             goto alloc_new_slab;
```

If there is a completely free slab move it from free list and add it back to the partially full list.

```
1276         list_del(entry);                                \
1277         list_add(entry, slabs_partial);                  \
1278     }                                                    \
1279
```

When we arrive here then *entry* is pointing to a non-empty slab.

Note the macro based assignment in which the value returned by *kmem_cache_alloc_one_tail()* is assigned to the *objp* lvalue in the macro call far above.

```
1280     slabp = list_entry(entry, slab_t, list);            \
1281     kmem_cache_alloc_one_tail(cachep, slabp);           \
1282 })
1283

1348
1349     local_irq_restore(save_flags);
1350     return objp;
```

Try to allocate a new slab for the cache and if that works, go back to *try_again* which was defined above.

```
1351 alloc_new_slab:
```

```
1356     local_irq_restore(save_flags);
1357     if (kmem_cache_grow(cachep, flags))
1358 /* Someone may have stolen our objs. Doesn't matter, we'll
1359 * just come back here again.
1360 */
1361         goto try_again;
1362     return NULL;
1363 }
```

The details of internal object management are fairly nasty...

Objects are managed either on slab (small objects) or off slab (large objects)

```
692 /* Determine if the slab management is 'on' or 'off' slab. */
693     if (size >= (PAGE_SIZE>>3))
694         /*
695          * Size is large, best to place the slab management obj
696          * off-slab (should allow better packing of objs).
697          */
698         flags |= CFLGS_OFF_SLAB;
```

This macro appears to increment a pointer to a *slab_t* structure by the length of a *slab_t* struct, It is used to access the object management list associated with a slab.

k_mem_bufctl_t is an *int* that represents an object index.

```
162 #define slab_bufctl(slabp) \
163     ((kmem_bufctl_t *) ( ((slab_t*)slabp)+1) )
```

slabp->free contains the index of the first free object within the slab.

slabp->mem is a pointer to the start of the object store.

The free objects are managed as a stack.

```
1222 static inline void * kmem_cache_alloc_one_tail(
1223     kmem_cache_t *cachep,
1224     slab_t *slabp)
1225 {
1226     void *objp;
1227     STATS_INC_ALLOCED(cachep);
1228     STATS_INC_ACTIVE(cachep);
1229     STATS_SET_HIGH(cachep);
1230
```

Free object management appears to be a table of *ints* in which *value[index] = index* of the next free object and *slabp->free* points to the first free object.

```
1231     /* get obj pointer */
1232     slabp->inuse++;
1233     objp = slabp->s_mem + slabp->free*cachep->objsize;
1234     slabp->free=slab_bufctl(slabp)[slabp->free];
1235
```

If we just consumed the last object in the list move the slab..

```
1236     if (unlikely(slabp->free == BUFCTL_END)) {
1237         list_del(&slabp->list);
1238         list_add(&slabp->list, &cachep->slabs_full);
1239     }
1252     objp += BYTES_PER_WORD;
1255     return objp;
1256 }
```

Freeing of a single object

```
1394 static inline void kmem_cache_free_one(kmem_cache_t *cachep,
        void *objp)
1395 {
1396     slab_t* slabp;
1397
1398     CHECK_PAGE(virt_to_page(objp));
1405     slabp = GET_PAGE_SLAB(virt_to_page(objp));
        : (debugging stuff deleted)
```

Here we insert the old object into the head of the free list of the slab

The object number is computed by dividing the object offset by the object size.
The existing first free object number is copied into the array slot index of this object
Then the slab's free list start address is set to index of this object.

```
1430 {
1431     unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
1432
1433     slab_bufctl(slabp)[objnr] = slabp->free;
1434     slabp->free = objnr;
1435 }
1436     STATS_DEC_ACTIVE(cachep);
1437 }
```

Since something was just freed a full slab may now be partial (or empty) and a partial slab may be empty

```
1438     /* fixup slab chains */
1439 {
1440     int inuse = slabp->inuse;
1441     if (unlikely(!--slabp->inuse)) {
1442         /* Was partial or full, now empty. */
1443         list_del(&slabp->list);
1444         list_add(&slabp->list, &cachep->slabs_free);
1445     } else if (unlikely(inuse == cachep->num)) {
1446         /* Was full. */
1447         list_del(&slabp->list);
1448         list_add(&slabp->list, &cachep->slabs_partial);
1449     }
1450 }
1451 }
```

This routine is called by *kmem_cache_grow()* when a new slab is required
It can provide some insight into understanding on slab / off slab object managemtn.

```
1010
1011 /* Get the memory for a slab management obj. */
1012 static inline slab_t * kmem_cache_slabmgmt (
1013     kmem_cache_t *cachep,
1014     void *objp,
1015     int colour_off,
1016     int local_flags)
1017 {
1018     slab_t *slabp;
1019 }
```

The OFF_CACHE macro just test a flag bit in the *kmem_cache_s* structure.

For off slab managment the management area is allocated from the *kmem_cache* cache.

```
1017     if (OFF_SLAB(cachep)) {
1018         /* Slab management obj is off-slab. */
1019         slabp = kmem_cache_alloc(cachep->slabp_cache,
1020                                 local_flags);
1021         if (!slabp)
1022             return NULL;
1023 }
```

For on-slab managment the management area is suballocated from the area normally used to hold objects. The *colour_off* field is then incremented so as include the slab management area.

```
1024     } else {
1025         /* FIXME: change to
1026         slabp = objp
1027         * if you enable OPTIMIZE
1028         */
1029         slabp = objp+colour_off;
1030         colour_off += L1_CACHE_ALIGN(cachep->num *
1031                                     sizeof(kmem_bufctl_t) + sizeof(slab_t));
1032     }
1033     slabp->inuse = 0;
1034     slabp->colouroff = colour_off;
1035     slabp->s_mem = objp+colour_off;
1036     return slabp;
1037 }
```

This routine is called by *kmem_cache_grow()* just after it calls *kmem_cache_mgmt*
The free object chains are setup here.

```
1038 static inline void kmem_cache_init_objs (kmem_cache_t * cachep,  
1039 slab_t * slabp, unsigned long ctor_flags)  
1040 {  
1041     int i;  
1042  
1043     for (i = 0; i < cachep->num; i++) {  
1044         void* objp = slabp->s_mem+cachep->objsize*i;  
  
1053  
1054         /*  
1055         * Constructors are not allowed to allocate memory from  
1056         * the same cache which they are a constructor for.  
1057         * Otherwise, deadlock. They must also be threaded.  
1058         */  
1059         if (cachep->ctor)  
1060             cachep->ctor(objp, cachep, ctor_flags);  
  
1075             slab_bufctl(slabp)[i] = i+1;  
1076         }  
1077         slab_bufctl(slabp)[i-1] = BUFCTL_END;  
1078         slabp->free = 0;  
1079     }
```