Memory Management

Intel x86 hardware

Supports segmentation over paging

\[(\text{segid, offset})\]
\[\text{linear address}\]
\[\text{physical address}\]

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.

```c
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 = 0x00000000
Limit = 0xfffff
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:

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

From System.map

c0100000 A _text
 c026ad91 A _etext
 c02e77a0 A _edata
 c0368e58 A _end

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.

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

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

- **ISA DMA area** 0x0 - 0xfffff
- **Normal** 0x100000 -0x37fffff (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:

<table>
<thead>
<tr>
<th>size</th>
<th>alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 page</td>
<td>4K aligned</td>
</tr>
<tr>
<td>2 pages</td>
<td>8K aligned</td>
</tr>
<tr>
<td>4 pages</td>
<td>16K aligned</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>512 pages</td>
<td>8 MB aligned</td>
</tr>
</tbody>
</table>

```c
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 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 \((\text{size-of-zone}) / (\text{size-of-page} \times 2^{(\text{order} + 1)})\)

Suppose there was exactly 1 MB of memory

There are \(2^{20} / 2^{12} = 2^8\) pages

<table>
<thead>
<tr>
<th>Order</th>
<th>block size</th>
<th>sets of buddies (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4K</td>
<td>(2^7)</td>
</tr>
<tr>
<td>1</td>
<td>8K</td>
<td>(2^6)</td>
</tr>
<tr>
<td>2</td>
<td>16K</td>
<td>(2^5)</td>
</tr>
<tr>
<td>3</td>
<td>32K</td>
<td>(2^4)</td>
</tr>
<tr>
<td>4</td>
<td>64K</td>
<td>(2^3)</td>
</tr>
<tr>
<td>5</td>
<td>128K</td>
<td>(2^2)</td>
</tr>
<tr>
<td>6</td>
<td>256K</td>
<td>(2^1)</td>
</tr>
<tr>
<td>7</td>
<td>512K</td>
<td>(2^0)</td>
</tr>
</tbody>
</table>

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 management

`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

```c
typedef struct zone_struct {
  /* Commonly accessed fields: */
  spinlock_t          lock;
  unsigned long      free_pages;
  unsigned long      pages_min, pages_low, pages_high;
  int                need_balance;

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

  /* Discontig memory support fields. */
  struct pglist_data  *zone_pgdat;
  struct page          *zone_mem_map;
  unsigned long        zone_start_paddr;
  unsigned long        zone_start_mapnr;

  /* rarely used fields: */
  char                *name;
  unsigned long       size;
} 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

```c
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;
```
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.

```c
/*
 * Set up the zone data structures:
 *   - mark all pages reserved
 *   - mark all memory queues empty
 *   - clear the memory bitmaps
 */

void __init free_area_init_core(
    int nid,    /* 0 */
    pg_data_t *pgdat,
    struct page **gmap,
    unsigned long *zones_size,    /* 0 */
    unsigned long zone_start_paddr,    /* 0 */
    unsigned long *zholes_size,    /* 0 */
    struct page *lmem_map)    /* 0 */
{

    struct page *p;
    unsigned long i, j;
    unsigned long map_size;
    unsigned long totalpages, offset, realtime.totalpages;
    const unsigned long zone_required_alignment = 1UL <<
        (MAX_ORDER-1);

    if (zone_start_paddr & ~PAGE_MASK)
        BUG();

    totalpages = 0;
    for (i = 0; i < MAX_NR_ZONES; i++) {
        unsigned long size = zones_size[i];
        totalpages += size;
    }
    realtime.totalpages = totalpages;
    if (zholes_size)
        for (i = 0; i < MAX_NR_ZONES; i++)
            realtime.totalpages -= zholes_size[i];
    printk("On node %d totalpages: %lu\n", nid, realtime.totalpages);

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

Mar 21 17:16:34 waco kernel: On node 0 totalpages: 32704
Mar 21 17:16:34 waco kernel: zone(0): 4096 pages.
The `mem_map` table of `struct page` is allocated here via a special purpose low level allocator `alloc_bootmem_node()`

```c
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));
678 }
679 /*
680 *gmap is actually an alias here for the global variable `mem_map`.. ain't C wonderful!
681 */
682 *gmap = pgdat->node_mem_map = lmem_map;
683 pgdat->node_size = totalpages;
684 pgdat->node_start_paddr = zone_start_paddr;
685 pgdat->node_start_mapnr = (lmem_map - mem_map);
686 pgdat->nr_zones = 0;
687 
688 Flag all pages initially reserved. They get unreserved at end of boot.
689 /*
690 * Initially all pages are reserved - free ones are freed
691 * up by free_all_bootmem() once the early boot process is
692 * done.
693 */
694 for (p = lmem_map; p < lmem_map + totalpages; p++) {
695     set_page_count(p, 0);
696     SetPageReserved(p);
697     init_waitqueue_head(&p->wait);
698     memlist_init(&p->list);
699 }
```

11
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;
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 $realsize = 2^{26} / 2^{12} = 2^{14}$

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

$pages\_min = 2^7$

$pages\_low = 2^8$

$pages\_high = 384.$

```c
mask = (realsize / zone_balance_ratio[j]);
if (mask < zone_balance_min[j])
    mask = zone_balance_min[j];
else if (mask > zone_balance_max[j])
    mask = zone_balance_max[j];
zone->pages_min = mask;
zone->pages_low = mask*2;
zone->pages_high = mask*3;
```

```c
zone->zone_mem_map = mem_map + offset;
zone->zone_start_mapnr = offset;
zone->zone_start_paddr = zone_start_paddr;
```

```c
if (((zone_start_paddr >> PAGE_SHIFT) &
    (zone_required_alignment-1)))
    printk("BUG: wrong zone alignment, it will crash\n");
```

```c
for (i = 0; i < size; i++) {
    struct page *page = mem_map + offset + i;
    page->zone = zone;
    if (j != ZONE_HIGHMEM)
        page->virtual = __va(zone_start_paddr);
    zone_start_paddr += PAGE_SIZE;
}
```

```c
#define __pa(x)  ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x)  ((void *)((unsigned long)(x)+PAGE_OFFSET))
```
Initialize the buddy system structures here. `size` is expressed in pages.

```c
    offset += size;
    for (i = 0; ; i++) {
        unsigned long bitmap_size;
        memlist_init(&zone->free_area[i].free_list);
        if (i == MAX_ORDER-1) {
            zone->free_area[i].map = NULL;
            break;
        }
    }
    /*
     * Page buddy system uses "index >> (i+1)",
     * where "index" is at most "size-1".
     *
     * The extra "+3" is to round down to byte
     * size (8 bits per byte assumption). Thus
     * we get "(size-1) >> (i+4)" as the last byte
     * we can access.
     *
     * The "+1" is because we want to round the
     * byte allocation up rather than down. So
     * we should have had a "+7" before we shifted
     * down by three. Also, we have to add one as
     * we actually _use_ the last bit (it's [0,n]
     * inclusive, not [0,n[).
     *
     * So we actually had +7+1 before we shift
     * down by 3. But (n+8) >> 3 == (n >> 3) + 1
     * (modulo overflows, which we do not have).
     *
     * Finally, we LONG_ALIGN because all bitmap
     * operations are on longs.
     */
    bitmap_size = (size-1) >> (i+4);
    bitmap_size = LONG_ALIGN(bitmap_size+1);
    zone->free_area[i].map =
        (unsigned long *) alloc_bootmem_node(pgdat,
            bitmap_size);
    }
    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 log2(#pages) required to satisfy the request.

GFP_MASK can be set to

```c
#define __GFP_DMA 0x01
#define __GFP_HIGHMEM 0x02

/* Action modifiers - doesn't change the zoning */
define __GFP_WAIT 0x10 /* Can wait and reschedule? */
define __GFP_HIGH 0x20 /* Should access emergency pools? */
define __GFP_IO 0x40 /* Can start low memory physical IO? */
define __GFP_HIGHLIO 0x80 /* Can start high mem physical IO? */
define __GFP_FS 0x100 /* Can call down to low-level FS? */
define __GFP_NOHIGHIO (__GFP_HIGH | __GFP_WAIT | __GFP_IO)
define __GFP_NOFS (__GFP_HIGH | __GFP_WAIT | __GFP_IO | __GFP_HIGHLIO)
define __GFP_ATOMIC (__GFP_HIGH)
define __GFP_HUSER (__GFP_WAIT | __GFP_IO | __GFP_HIGHLIO | __GFP_FS)
define __GFP_KERNEL (__GFP_HIGH | __GFP_WAIT | __GFP_IO | __GFP_HIGHLIO | __GFP_FS | __GFP_HIGHLIO | __GFP_FS)
define __GFP_KSWAPD (__GFP_WAIT | __GFP_IO | __GFP_HIGHLIO | __GFP_FS)
```

```c
unsigned long __get_free_pages(unsigned int gfp_mask,
                              unsigned int order)
{
    struct page * page;
    page = alloc_pages(gfp_mask, order);
    if (!page)
        return 0;
    return (unsigned long) page_address(page);
}
```

alloc_pages is a front for the usual insulating layers

```c
static inline struct page * alloc_pages(unsigned int gfp_mask,
                                        unsigned int order)
{
    /* Gets optimized away by the compiler. */
    if (order >= MAX_ORDER)
        return NULL;
    return _alloc_pages(gfp_mask, order);
}
```
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_zonelists+(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
312    zone = zonelist->zones;
313    classzone = *zone;
314    min = 1UL << order;
315    for (;;) {
316        zone_t *z = *(zone++);
317        if (!z)
318            break;
319        min += z->pages_low;
320        if (z->free_pages > min) {
321            page = rmqueue(z, order);
322            if (page)
323                return page;
324        }
325    }
326 }
```
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            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
Here we test agains 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.
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         page = rmqueue(z, order);
362         if (page)
363             return page;
364     }
365     return NULL;
366 }
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
19
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.

```c
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`

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


`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.

```c
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.

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

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

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

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.

```c
191                if (curr != head) {
192                    unsigned int index;
193```
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 $2^{\text{order}}$ pages.
The ones we don’t need must go back on the free lists of lower orders.

```c
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.

```c
196    if (curr_order != MAX_ORDER-1)
197        MARK_USED(index, curr_order, area);
198    zone->free_pages -= 1UL << order;
199    page = expand(zone, page, index, order, curr_order, area);
200    spin_unlock_irqrestore(&zone->lock, flags);
201    set_page_count(page, 1);
202    if (BAD_RANGE(zone, page))
203        BUG();
204    if (PageLRU(page))
205        BUG();
206    if (PageActive(page))
207        BUG();
208    return page;
209  }
210  curr_order++;
211  area++;
212 } while (curr_order < MAX_ORDER);
213 spin_unlock_irqrestore(&zone->lock, flags);
214 return NULL;
215 ```
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    while (high > low) {  
164        if (BAD_RANGE(zone,page))  
165            BUG();  
166        area--;  
167        high--;  
168        size >>= 1;  
169        memlist_add_head(&(page)->list, & (area)->free_list);  
170        MARK_USED(index, high, area);  
171        index += size;  
172        page += size;  
173    }  
174    if (BAD_RANGE(zone,page))  
175        BUG();  
176    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 in 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 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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Active obj</th>
<th>#obj</th>
<th>Sz Obj</th>
<th>ActSlb</th>
<th>#Slb</th>
<th>#Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>size-131072 (DMA)</td>
<td>0</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>size-131072</td>
<td>0</td>
<td>0</td>
<td>131072</td>
<td>0</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>size-65536 (DMA)</td>
<td>0</td>
<td>0</td>
<td>65536</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>size-65536</td>
<td>0</td>
<td>0</td>
<td>65536</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>size-32768 (DMA)</td>
<td>0</td>
<td>0</td>
<td>32768</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>size-32768</td>
<td>0</td>
<td>0</td>
<td>32768</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>size-16384 (DMA)</td>
<td>0</td>
<td>0</td>
<td>16384</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>size-16384</td>
<td>0</td>
<td>0</td>
<td>16384</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>size-8192 (DMA)</td>
<td>2</td>
<td>2</td>
<td>8192</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>size-8192</td>
<td>1</td>
<td>1</td>
<td>8192</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>size-4096 (DMA)</td>
<td>0</td>
<td>0</td>
<td>4096</td>
<td>0</td>
<td>1</td>
<td>60</td>
</tr>
<tr>
<td>size-4096</td>
<td>0</td>
<td>0</td>
<td>4096</td>
<td>29</td>
<td>29</td>
<td>60</td>
</tr>
<tr>
<td>size-2048 (DMA)</td>
<td>0</td>
<td>0</td>
<td>2048</td>
<td>0</td>
<td>1</td>
<td>60</td>
</tr>
<tr>
<td>size-2048</td>
<td>0</td>
<td>0</td>
<td>2048</td>
<td>57</td>
<td>57</td>
<td>60</td>
</tr>
<tr>
<td>size-1024 (DMA)</td>
<td>2</td>
<td>4</td>
<td>1024</td>
<td>1</td>
<td>1</td>
<td>124</td>
</tr>
<tr>
<td>size-1024</td>
<td>2</td>
<td>4</td>
<td>1024</td>
<td>57</td>
<td>57</td>
<td>124</td>
</tr>
<tr>
<td>size-512 (DMA)</td>
<td>0</td>
<td>0</td>
<td>512</td>
<td>0</td>
<td>1</td>
<td>124</td>
</tr>
<tr>
<td>size-512</td>
<td>0</td>
<td>0</td>
<td>512</td>
<td>36</td>
<td>36</td>
<td>124</td>
</tr>
<tr>
<td>size-256 (DMA)</td>
<td>0</td>
<td>0</td>
<td>256</td>
<td>0</td>
<td>1</td>
<td>252</td>
</tr>
<tr>
<td>size-256</td>
<td>0</td>
<td>0</td>
<td>256</td>
<td>20</td>
<td>20</td>
<td>252</td>
</tr>
<tr>
<td>size-128 (DMA)</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>0</td>
<td>1</td>
<td>252</td>
</tr>
<tr>
<td>size-128</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>37</td>
<td>37</td>
<td>252</td>
</tr>
<tr>
<td>size-64 (DMA)</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>1</td>
<td>252</td>
</tr>
<tr>
<td>size-64</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>17</td>
<td>17</td>
<td>252</td>
</tr>
<tr>
<td>size-32 (DMA)</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>1</td>
<td>252</td>
</tr>
<tr>
<td>size-32</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>43</td>
<td>43</td>
<td>252</td>
</tr>
</tbody>
</table>
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.

```c
struct kmem_cache_s {
    /* 1) each alloc & free */
    /* full, partial first, then free */
    struct list_head slabs_full;
    struct list_head slabs_partial;
    struct list_head slabs_free;
    unsigned int objsize;
    unsigned int flags; /* constant flags */
    unsigned int num; /* # of objs per slab */
    spinlock_t spinlock;
}
```

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.

```c
typedef struct slab_s {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem; /* including colour offset */
    unsigned int inuse; /* num of objs active in slab */
    kmem_bufctl_t free; /* This is an int "index" */
} slab_t;
```
The `kmem_cache_s` structure used to describe a cache is itself allocated from the `cache_cache` which is statically initialized as follows.

```c
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
1327   :
1328 #else
1329   objp = kmem_cache_alloc_one(cachep);
1330 #define kmem_cache_alloc_one(cachep)
1331 ({
1332  struct list_head * slabs_partial, * entry;
1333  slab_t *slabp;
1334
See if there are any partially allocated slabs in this cache

1335  slabs_partial = &(cachep)->slabs_partial;
1336  entry = slabs_partial->next;
1337
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.

1338  if (unlikely(entry == slabs_partial)) {
1339  struct list_head * slabs_free;
1340  slabs_free = &cachep)->slabs_free;
1341  entry = slabs_free->next;
1342 If no completely free slabs either, then it will be necessary to allocate a new slab

1343  if (unlikely(entry == slabs_free))
1344     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)

```c
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.

```c
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.

```c
1222 static inline void * kmem_cache_alloc_one_tail(
    kmem_cache_t *cachep,
    slab_t *slabp)
{ 
    void *objp;
1225     STATS_INC_ALLOCED(cachep);
1226     STATS_INC_ACTIVE(cachep);
1227     STATS_SET_HIGH(cachep);
1228     /* get obj pointer */
1229     slabp->inuse++;
1230     objp = slabp->s_mem + slabp->free*cachep->objsize;
1231     slabp->free=slab_bufctl(slabp)[slabp->free];
```
If we just consumed the last object in the list move the slab.:

```c
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:

```c
1394 static inline void kmem_cache_free_one(kmem_cache_t *cachep,
                                          void *objp)
1395 {
1396     slab_t* slabp;
1397     CHECK_PAGE(virt_to_page(objp));
1405     slabp = GET_PAGE_SLAB(virt_to_page(objp));
1430      {
1431         unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
1433         slab_bufctl(slabp)[objnr] = slabp->free;
1434         slabp->free = objnr;
1435     }
1436     STATS_DEC_ACTIVE(cachep);
```
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 management.

```c
/* Get the memory for a slab management obj. */
static inline slab_t * kmem_cache_slabmgmt (
    kmem_cache_t *cachep,
    void *objp,
    int colour_off,
    int local_flags)
{
    slab_t *slabp;

    /* FIXME: change to */
    else {
        /* FIXME: change to */
        slabp = objp
        * if you enable OPTIMIZE
        */
        colour_off += L1_CACHE_ALIGN(cachep->num *
            sizeof(kmem_bufctl_t) + sizeof(slab_t));
        } else {
            slabp->inuse = 0;
            slabp->colouroff = colour_off;
            slabp->s_mem = objp+colour_off;
        return slabp;
    }

    slabp = kmem_cache_alloc(cachep->slabp_cache,
        local_flags);
    if (!slabp)
        return NULL;

    if (OFF_SLAB(cachep)) {
        /* Slab management obj is off-slab. */
        slabp = kmem_cache_alloc(cachep->slabp_cache,
            local_flags);
        if (!slabp)
            return NULL;
    } else {
        /* FIXME: change to */
        slabp = objp+colour_off;
        colour_off += L1_CACHE_ALIGN(cachep->num *
            sizeof(kmem_bufctl_t) + sizeof(slab_t));
    }
    return slabp;
}
```

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

For off-slab management, the management area is allocated from the `kmem_cache` cache.

For on-slab management, 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.
This routine is called by `kmem_cache_grow()` just after it calls `kmem_cache_mgmt`
The free object chains are setup here.

```c
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     for (i = 0; i < cachep->num; i++) {
1043         void* objp = slabp->s_mem+cachep->objsize*i;
1044         if (cachep->ctor)
1045             cachep->ctor(objp, cachep, ctor_flags);
1046         slab_bufctl(slabp)[i] = i+1;
1047     }
1048     slab_bufctl(slabp)[i-1] = BUFCTL_END;
1049     slabp->free = 0;
1050 }
```

35