Code Commentary On The Linux Virtual Memory Manager

Mel Gorman

14th July 2003
## Contents

1 **Boot Memory Allocator** .............................................. 8

1.1 Representing the Boot Map ....................................... 8

1.2 Initialising the Boot Memory Allocator ......................... 8

  - Function: setup_memory ........................................ 8
  - Function: init_bootmem ....................................... 11
  - Function: init_bootmem_node ................................ 12
  - Function: init_bootmem_core ................................ 12

1.3 Allocating Memory ................................................. 13

  - Function: reserve_bootmem .................................... 13
  - Function: reserve_bootmem_node .............................. 14
  - Function: reserve_bootmem_core ............................. 14
  - Function: alloc_bootmem ...................................... 15
  - Function: _alloc_bootmem .................................... 16
  - Function: alloc_bootmem_node .............................. 16
  - Function: _alloc_bootmem_node ............................. 17
  - Function: _alloc_bootmem_core ............................ 17

1.4 Freeing Memory ................................................... 22

  - Function: free_bootmem ...................................... 22
  - Function: free_bootmem_core ................................ 23

1.5 Retiring the Boot Memory Allocator ............................. 24

  - Function: mem_init ........................................... 24
  - Function: free_pages_init ................................... 26
  - Function: free_all_bootmem .................................. 26
  - Function: free_all_bootmem_core ............................ 27

2 **Physical Page Management** ...................................... 29

2.1 Allocating Pages .................................................. 29

  - Function: alloc_pages ....................................... 29
  - Function: _alloc_pages ....................................... 30
  - Function: __alloc_pages ...................................... 31
  - Function: rmqueue ............................................. 35
  - Function: expand .............................................. 37

2.2 Free Pages .......................................................... 38

  - Function: __free_pages ....................................... 39
  - Function: __free_pages_ok .................................... 39

2.3 Page Allocate Helper Functions ................................ 43
4.2.1 Initialising Objects in a Slab ........................................ 88
  Function: kmem_cache_init_objs ........................................ 88
4.2.2 Object Allocation .................................................. 90
  Function: kmem_cache_alloc .............................................. 90
  Function: __kmem_cache_alloc (UP Case) .............................. 91
  Function: __kmem_cache_alloc (SMP Case) ............................ 92
  Function: kmem_cache_alloc_head ..................................... 93
  Function: kmem_cache_alloc_one ..................................... 94
  Function: kmem_cache_alloc_one_tail ................................. 95
  Function: kmem_cache_alloc_batch .................................. 96
4.2.3 Object Freeing ....................................................... 98
  Function: kmem_cache_free ............................................. 98
  Function: __kmem_cache_free ......................................... 99
  Function: __kmem_cache_free ......................................... 99
  Function: kmem_cache_free_one ..................................... 100
  Function: free_block ................................................ 102
  Function: __free_block ............................................. 102
4.3 Sizes Cache .......................................................... 102
  Function: kmem_cache_sizes_init .................................. 102
4.3.1 kmalloc ............................................................ 104
  Function: kmalloc ................................................... 104
4.3.2 kfree ............................................................... 105
  Function: kfree ....................................................... 105
4.4 Per-CPU Object Cache ................................................ 106
4.4.1 Describing the Per-CPU Object Cache ............................ 106
4.4.2 Adding/Removing Objects from the Per-CPU Cache .............. 107
4.4.3 Enabling Per-CPU Caches ....................................... 107
  Function: enable_all_cpucaches ................................... 107
  Function: enable_cpucache ........................................... 108
  Function: kmem_tune_cpucache ..................................... 109
4.4.4 Updating Per-CPU Information .................................. 111
  Function: smp_function_all_cpus ................. 111
  Function: do_ccupdate_local ...................................... 112
4.4.5 Draining a Per-CPU Cache ....................................... 112
  Function: drain_cpu_caches ...................................... 113
4.5 Slab Allocator Initialisation ....................................... 114
  Function: kmem_cache_init ....................................... 114
4.6 Interfacing with the Buddy Allocator ............................. 115
  Function: kmem_getpages .......................................... 115
  Function: kmem_freepages ....................................... 116

5 Process Address Space .................................................. 117
5.1 Managing the Address Space ....................................... 117
5.2 Process Memory Descriptors ....................................... 117
  5.2.1 Allocating a Descriptor ..................................... 119
    Function: allocate_mm ........................................... 119
5.3.9 Fixing up regions after locking/unlocking ........................................ 173
  Function: mlock_fixup .................................................. 173
  Function: mlock_fixup_all ............................................ 174
  Function: mlock_fixup_start .......................................... 175
  Function: mlock_fixup_end ........................................... 176
  Function: mlock_fixup_middle ....................................... 177

5.3.10 Deleting a memory region ................................................. 179
  Function: do_munmap .................................................. 179
  Function: unmap_fixup ............................................... 183

5.3.11 Deleting all memory regions .......................................... 186
  Function: exit_mmap .................................................. 186

5.4 Page Fault Handler ............................................................ 188
  5.4.1 Handling the Page Fault .......................................... 198
    Function: handle_mm_fault ......................................... 198
    Function: handle_pte_fault ....................................... 199
  5.4.2 Demand Allocation .................................................. 200
    Function: do_no_page ................................................ 200
    Function: do_anonymous_page ..................................... 204
  5.4.3 Demand Paging ........................................................ 206
    Function: do_swap_page ............................................ 206
  5.4.4 Copy On Write (COW) Pages ...................................... 209

6 High Memory Management .................................................................. 212
  6.1 Mapping High Memory Pages ............................................. 212
    Function: kmap .......................................................... 212
    Function: kmap_high .................................................. 213
    Function: map_new_virtual .......................................... 213
    Function: flush_all_zero_pkmaps .................................. 216
  6.1.1 Unmapping Pages ........................................................ 217
    Function: kunmap ...................................................... 217
    Function: kunmap_high ............................................... 217
  6.2 Mapping High Memory Pages Atomically .................................... 218
    Function: kmap_atomic ................................................. 218
    Function: kunmap_atomic ............................................ 219
  6.3 Bounce Buffers .................................................................. 220
    Function: create_buffers ............................................. 220
    Function: alloc_bounce_bh .......................................... 222
    Function: alloc_bounce_page ........................................ 223
  6.3.1 Copying via Bounce Buffers ......................................... 225
    Function: bounce_end_io_write ..................................... 225
    Function: bounce_end_io_read ...................................... 225
    Function: copy_from_high_bh ....................................... 225
    Function: copy_to_high_bh_irq ..................................... 226
    Function: bounce_end_io ............................................. 226
  6.4 Emergency Pools ................................................................ 228
    Function: init_emergency_pool ...................................... 228
7 Page Frame Reclamation

7.1 Page Swap Daemon

Function: kswapd_init
Function: kswapd
Function: kswapd_can_sleep
Function: kswapd_can_sleep_pgdat
Function: kswapd_balance
Function: kswapd_balance_pgdat

7.2 Page Cache

Function: lru_cache_add
Function: add_page_to_active_list
Function: add_page_to_inactive_list
Function: lru_cache_del
Function: __lru_cache_del
Function: del_page_from_active_list
Function: del_page_from_inactive_list
Function: mark_page_accessed
Function: activate_lock
Function: activate_page_nolock
Function: page_cache_get
Function: page_cache_release
Function: add_to_page_cache
Function: __add_to_page_cache

7.3 Shrinking all caches

Function: shrink_caches
Function: try_to_free_pages
Function: try_to_free_pages_zone

7.4 Refilling inactive_list

Function: refill_inactive

7.5 Reclaiming pages from the page cache

Function: shrink_cache

7.6 Swapping Out Process Pages

Function: swap_out
Function: swap_out_mm
Function: swap_out_vma
Function: swap_out_pgd
Function: swap_out_pmd
Function: try_to_swap_out

8 Swap Management

8.1 Describing the Swap Area

8.2 Scanning for free entries

Function: get_swap_page
Function: scan_swap_map

8.3 Swap Cache

Function: add_to_swap_cache
8.4 Activating a Swap Area ........................................ 274
Function: sys_swapon ........................................ 274

8.5 Deactivating a Swap Area .................................. 285
Function: sys_swapoff ....................................... 285
Function: try_to_unuse ...................................... 289
List of Figures

1.1 Call Graph: setup_memory ................................... 9

2.1 Call Graph: alloc_pages() .................................. 30
2.2 Call Graph: __free_pages() .................................. 39

3.1 Call Graph: vmalloc() ....................................... 47
3.2 Call Graph: vfree() ......................................... 54

4.1 Call Graph: kmem_cache_create() ............................. 60
4.2 Call Graph: kmem_cache_shrink() ............................. 70
4.3 Call Graph: kmem_cache_destroy() ............................ 73
4.4 Call Graph: kmem_cache_reap() ............................... 75
4.5 Call Graph: kmem_cache_grow() ............................... 83
4.6 Call Graph: kmem_slab_destroy() ............................. 87
4.7 Call Graph: kmem_cache_alloc() ............................... 91
4.8 Call Graph: kmem_cache_free() ............................... 98
4.9 kmalloc .................................................... 104
4.10 kfree ....................................................... 105

5.1 Call Graph: sys_mmap2() .................................... 129
5.2 Call Graph: get_unmapped_area() ............................. 143
5.3 Call Graph: insert_vm_struct() ............................... 145
5.4 Call Graph: sys_mremap ....................................... 153
5.5 Call Graph: move_vma ......................................... 159
5.6 Call Graph: move_page_tables() .............................. 164
5.7 do_munmap ................................................... 179
5.8 Call Graph: do_page_fault() .................................. 189
5.9 Call Graph: do_no_page() .................................... 201
5.10 do_swap_page ................................................ 210
5.11 do_wp_page .................................................. 211

6.1 Call Graph: kmap() .......................................... 212
6.2 Call Graph: create_bounce .................................... 220

7.1 shrink_cache .................................................. 240
7.2 Call Graph: swap_out ......................................... 251

8.1 Call Graph: get_swap_page() ................................. 262

9
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Physical Pages Allocation API</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>Physical Pages Free API</td>
<td>38</td>
</tr>
<tr>
<td>3.1</td>
<td>Non-Contiguous Memory Allocation API</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>Non-Contiguous Memory Free API</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Slab Allocator API for caches</td>
<td>60</td>
</tr>
</tbody>
</table>
Chapter 1

Boot Memory Allocator

1.1 Representing the Boot Map

A `bootmem_data` struct exists for each node of memory in the system. It contains the information needed for the boot memory allocator to allocate memory for a node such as the bitmap representing allocated pages and where the memory is located. It is declared as follows in `<linux/bootmem.h>`;

```c
25 typedef struct bootmem_data {
26     unsigned long node_boot_start;
27     unsigned long node_low_pfn;
28     void *node_bootmem_map;
29     unsigned long last_offset;
30     unsigned long last_pos;
31 } bootmem_data_t;
```

- `node_boot_start` is the starting physical address of the represented block
- `node_low_pfn` is the end physical address, in other words, the end of the `ZONE_NORMAL` this node represents
- `node_bootmem_map` is the location of the bitmap representing allocated or free pages with each bit
- `last_offset` is the offset within the the page of the end of the last allocation. If 0, the page used is full
- `last_pos` is the the PFN of the page used with the last allocation. Using this with the `last_offset` field, a test can be made to see if allocations can be merged with the page used for the last allocation rather than using up a full new page

1.2 Initialising the Boot Memory Allocator

Function: `setup_memory` (`arch/i386/kernel/setup.c`)
1.2. Initialising the Boot Memory Allocator

This function gets the necessary information to give to the boot memory allocator to initialise itself. It is broken up into a number of different tasks.

- Find the start and ending Page Frame Number (PFN) for low memory (\texttt{min\_low\_PFN}, \texttt{max\_low\_PFN}), the start and end PFN for high memory (\texttt{highstart\_PFN}, \texttt{highend\_PFN}) and the PFN for the last page in the system (\texttt{max\_PFN}).

- Initialise the \texttt{bootmem\_data} structure and declare which pages may be used by the boot memory allocator

- Mark all pages usable by the system as “free” and then reserve the pages used by the bitmap representing the pages

- Reserve pages used by the SMP config or the initrd image if one exists

```c
949 static unsigned long __init setup_memory(void) {
950    unsigned long bootstrap_size, start_pfn, max_low_pfn;
951
952    /*
953     * partially used pages are not usable - thus
954     * we are rounding upwards:
955     */
956
957    start_pfn = PFN_UP(__pa(&end));
958
959    find_max_pfn();
960```

Figure 1.1: Call Graph: \texttt{setup\_memory}
1.2. Initialising the Boot Memory Allocator

max_low_pfn = find_max_low_pfn();

#ifdef CONFIG_HIGHMEM
  highstart_pfn = highend_pfn = max_pfn;
  if (max_pfn > max_low_pfn) {
    highstart_pfn = max_low_pfn;
  }
  printk(KERN_NOTICE "%ldMB HIGHMEM available.
    pages_to_mb(highend_pfn - highstart_pfn));
#endif

printk(KERN_NOTICE "%ldMB LOWMEM available.
    pages_to_mb(max_low_pfn));

PFN_UP() takes a physical address, rounds it up to the next page and returns the page
frame number. _end is the address of the end of the loaded kernel image so start_pfn
is now the offset of the first physical page frame that may be used.

find_max_pfn() loops through the e820 map searching for the highest available pfn.
find_max_low_pfn() finds the highest page frame addressable in ZONE_NORMAL.

If high memory is enabled, start with a high memory region of 0. If it turns out
there is memory after max_low_pfn, put the start of high memory (highstart_pfn)
there and the end of high memory at max_pfn. Print out an informational message on
the availability of high memory.

print out an informational message on the amount of low memory

bootmap_size = init_bootmem(start_pfn, max_low_pfn);
register_bootmem_low_pages(max_low_pfn);
reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +
    bootmap_size + PAGE_SIZE-1) - (HIGH_MEMORY));
reserve_bootmem(0, PAGE_SIZE);

#ifdef CONFIG_SMP
reserve_bootmem(PAGE_SIZE, PAGE_SIZE);
#endif

init_bootmem() initialises the bootmem_data struct for the config_page_data node.
It sets where physical memory begins and ends for the node, allocates a bitmap rep-
resenting the pages and sets all pages as reserved.

registered_bootmem_low_pages() reads the e820 map and calls free_bootmem() for
all usable pages in the running system.
1.2. Initialising the Boot Memory Allocator

986-987 Reserve the pages that are being used by the bitmap representing the pages

993 Reserve page 0 as it is often a special page used by the bios

1001 Reserve an extra page which is required by the trampoline code. The trampoline code deals with how userspace enters kernel space

1003
1004 #ifdef CONFIG_X86_LOCAL_APIC
1005 /*
1006 * Find and reserve possible boot-time SMP configuration:
1007 */
1008 find_smp_config();
1009 #endif
1010 #ifdef CONFIG_BLK_DEV_INITRD
1011 if (LOADER_TYPE && INITRD_START) {
1012 if (INITRD_START + INITRD_SIZE <=
1013 (max_low_pfn << PAGE_SHIFT)) {
1014 reserve_bootmem(INITRD_START, INITRD_SIZE);
1015 initrd_start =
1016 INITRD_START ? INITRD_START + PAGE_OFFSET : 0;
1017 initrd_end = initrd_start+INITRD_SIZE;
1018 }
1019 else {
1020 printk(KERN_ERR
1021 "initrd extends beyond end of memory "
1022 "(0x%08lx > 0x%08lx)\n\ndisabling initrd\n",
1023 INITRD_START + INITRD_SIZE,
1024 max_low_pfn << PAGE_SHIFT);
1025 initrd_start = 0;
1026 }
1027 }
1028 #endif
1029 }
1030 return max_low_pfn;
1031 }
1032
1033 This function reserves memory that stores config information about the SMP setup
1040-1026 If initrd is enabled, the memory containing its image will be reserved. initrd provides a tiny filesystem image which is used to boot the system
1028 Return the upper limit of addressable memory in ZONE_NORMAL

Function: init_bootmem (mm/bootmem.c)
Called by UMA architectures to initialise their bootmem data.
Initialising the Boot Memory Allocator

```c
unsigned long __init init_bootmem (unsigned long start, unsigned long pages)
{
    max_low_pfn = pages;
    min_low_pfn = start;
    return(init_bootmem_core(&contig_page_data, start, 0, pages));
}
```

Confusingly, the `pages` parameter is actually the end PFN of the memory addressable by this node, not the number of pages as the name implies.

Set the max PFN addressable by this node in case the architecture dependent code did not.

Set the min PFN addressable by this node in case the architecture dependent code did not.

Call `init_bootmem_core()` which does the real work of initialising the `bootmem_data`.

**Function: init_bootmem_node** *(mm/bootmem.c)*
Used by NUMA architectures to initialise bootmem data for a given node.

```c
unsigned long __init init_bootmem_node (pg_data_t *pgdat, unsigned long freepfn, unsigned long startpfn, unsigned long endpfn)
{
    return(init_bootmem_core(pgdat, freepfn, startpfn, endpfn));
}
```

Just call `init_bootmem_core()` directly.

**Function: init_bootmem_core** *(mm/bootmem.c)*
Initialises the appropriate `struct bootmem_data_t` and inserts the node into the linked list of nodes `pgdat_list`.

```c
static unsigned long __init init_bootmem_core (pg_data_t *pgdat, unsigned long mapstart, unsigned long start, unsigned long end)
{
    bootmem_data_t *bdata = pgdat->bdata;
    unsigned long mapsize = ((end - start)+7)/8;
    pgdat->node_next = pgdat_list;
    pgdat_list = pgdat;
    mapsize = (mapsize + (sizeof(long) - 1UL)) & ~(sizeof(long) - 1UL);
    bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT);
```
1.3 Allocating Memory

Function: reserve_bootmem (mm/bootmem.c)

```c
void __init reserve_bootmem (unsigned long addr, unsigned long size) {
    reserve_bootmem_core(contig_page_data.bdata, addr, size);
}
```

Just call `reserve_bootmem_core()` passing the bootmem data from `contig_page_data` as the node to reserve memory from.

---

57    bdata->node_boot_start = (start << PAGE_SHIFT);
58    bdata->node_low_pfn = end;
59
60    /*
61    * Initially all pages are reserved - setup_arch() has to
62    * register free RAM areas explicitly.
63    */
64    memset(bdata->node_bootmem_map, 0xff, mapsize);
65
66    return mapsize;
67 }

The parameters are;

- `pgdat` is the node descriptor been initialised
- `mapstart` is the beginning of the memory that will be usable
- `start` is the beginning PFN of the node
- `end` is the end PFN of the node

50 Each page requires one bit to represent it so the size of the map required is the number
of pages in this node rounded up to the nearest multiple of 8 and then divided by 8 to
give the number of bytes required

52-53 As the node will be shortly considered initialised, insert it into the global `pgdat_list`

55 Round the mapsize up to the closest word boundary

56 Convert the mapstart to a virtual address and store it in `bdata->node_bootmem_map`

57 Convert the starting PFN to a physical address and store it on `node_boot_start`

58 Store the end PFN of `ZONE_NORMAL` in `node_low_pfn`

64 Fill the full map with 1’s marking all pages as allocated. It is up to the architecture
dependent code to mark the usable pages

---
### 1.3. Allocating Memory

#### Function: `reserve_bootmem_node` (*mm/bootmem.c*)

289 void __init reserve_bootmem_node (pg_data_t *pgdat,
   unsigned long physaddr,
   unsigned long size)

290 {
   reserve_bootmem_core(pgdat->bdata, physaddr, size);
292 }

291 Just call `reserve_bootmem_core()` passing it the bootmem data of the requested node

#### Function: `reserve_bootmem_core` (*mm/bootmem.c*)

74 static void __init reserve_bootmem_core(bootmem_data_t *bdata,
   unsigned long addr,
   unsigned long size)

75 {
76   unsigned long i;
77   /*
    * round up, partially reserved pages are considered
    * fully reserved.
    */
78   unsigned long sidx = (addr - bdata->node_boot_start)/PAGE_SIZE;
79   unsigned long eidx = (addr + size - bdata->node_boot_start +
                      PAGE_SIZE-1)/PAGE_SIZE;
80   unsigned long end = (addr + size + PAGE_SIZE-1)/PAGE_SIZE;
81   if (!size) BUG();
82   if (sidx < 0) BUG();
83   if (eidx < 0) BUG();
84   if (sidx >= eidx) BUG();
85   if ((addr >> PAGE_SHIFT) >= bdata->node_low_pfn)
     BUG();
86   if (end > bdata->node_low_pfn)
     BUG();
87   for (i = sidx; i < eidx; i++)
53     if (test_and_set_bit(i, bdata->node_bootmem_map))
54       printk("hm, page %08lx reserved twice.\n",
                i*PAGE_SIZE);
89 }

81 The `sidx` is the starting index to serve pages from. The value is obtained by subtracting the starting address from the requested address and dividing by the size of a page.
1.3. Allocating Memory

A similar calculation is made for the ending index `eidx` except that the allocation is rounded up to the nearest page. This means that requests to partially reserve a page will result in the full page being reserved.

`end` is the last PFN that is affected by this reservation.

Check that a non-zero value has been given.

Check the starting index is not before the start of the node.

Check the end index is not before the start of the node.

Check the starting index is not after the end index.

Check the starting address is not beyond the memory this bootmem node represents.

Check the ending address is not beyond the memory this bootmem node represents.

Starting with `sidx` and finishing with `eidx`, test and set the bit in the bootmem map that represents the page marking it as allocated. If the bit was already set to 1, print out a message saying it was reserved twice.

**Function: alloc_bootmem** *(mm/bootmem.c)*

```c
#define alloc_bootmem(x) __alloc_bootmem((x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_low(x) __alloc_bootmem((x), SMP_CACHE_BYTES, 0)
#define alloc_bootmem_pages(x) __alloc_bootmem((x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_low_pages(x) __alloc_bootmem((x), PAGE_SIZE, 0)
```

`alloc_bootmem()` will align to the L1 hardware cache and start searching for a page after the maximum address usable for DMA.

`alloc_bootmem_low()` will align to the L1 hardware cache and start searching from page 0.

`alloc_bootmem_pages()` will align the allocation to a page size so that full pages will be allocated starting from the maximum address usable for DMA.

`alloc_bootmem_pages()` will align the allocation to a page size so that full pages will be allocated starting from physical address 0.
Function: `__alloc_bootmem` (*mm/bootmem.c*)

326 void * __init __alloc_bootmem (unsigned long size, 
    unsigned long align, unsigned long goal)

327 {
328     pg_data_t *pgdat;
329     void *ptr;
330
331     for_each_pgdat(pgdat)
332         if ((ptr = __alloc_bootmem_core(pgdat->bdata, size, 
333             align, goal)))
334             return(ptr);
335
336     /*
337      * Whoops, we cannot satisfy the allocation request.
338     */
339     printk(KERN_ALERT "bootmem alloc of %lu bytes failed!\n", size);
340     panic("Out of memory");
341     return NULL;
342 }

326 The parameters are;

* `size` is the size of the requested allocation
* `align` is the desired alignment and must be a power of 2. Currently either 
  `SMP_CACHE_BYTES` or `PAGE_SIZE`
* `goal` is the starting address to begin searching from

331-334 Cycle through all available nodes and try allocating from each in turn. In the 
UMA case, this will just allocate from the `contig_page_data` node

349-340 If the allocation fails, the system is not going to be able to boot so the kernel 
panics

Function: `alloc_bootmem_node` (*mm/bootmem.c*)

53 #define alloc_bootmem_node(pgdat, x) \ 54     __alloc_bootmem_node((pgdat), (x), SMP_CACHE_BYTES, 
    __pa(MAX_DMA_ADDRESS))
55 #define alloc_bootmem_pages_node(pgdat, x) \ 56     __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, 
    __pa(MAX_DMA_ADDRESS))
57 #define alloc_bootmem_low_pages_node(pgdat, x) \ 58     __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, 0)

53-54 `alloc_bootmem_node()` will allocate from the requested node and align to the L1 
hardware cache and start searching for a page after the maximum address usable for 
DMA
1.3. Allocating Memory

55-56 alloc_bootmem_pages() will allocate from the requested node and align the allocation to a page size so that full pages will be allocated starting from the maximum address usable for DMA.

57-58 alloc_bootmem_pages() will allocate from the requested node and align the allocation to a page size so that full pages will be allocated starting from physical address 0.

Function: __alloc_bootmem_node (mm/bootmem.c)

344 void * __init __alloc_bootmem_node (pg_data_t *pgdat,
                        unsigned long size,
                        unsigned long align,
                        unsigned long goal)

345 {
    void *ptr;
    ptr = __alloc_bootmem_core(pgdat->bdata, size, align, goal);
    if (ptr)
        return (ptr);
    /*
     * Whoops, we cannot satisfy the allocation request.
     */
    printk(KERN_ALERT "bootmem alloc of \%lu bytes failed!\n", size);
    panic("Out of memory");
    return NULL;
}

344 The parameters are the same as for __alloc_bootmem_node() except the node to allocate from is specified.

348 Call the core function __alloc_bootmem_core() to perform the allocation.

349-350 Return a pointer if it was successful.

355-356 Otherwise print out a message and panic the kernel as the system will not boot if memory can not be allocated even now.

Function: __alloc_bootmem_core (mm/bootmem.c)

This is the core function for allocating memory from a specified node with the boot memory allocator. It is quite large and broken up into the following tasks:

- Function preamble. Make sure the parameters are sane.
- Calculate the starting address to scan from based on the goal parameter.
- Check to see if this allocation may be merged with the page used for the previous allocation to save memory.
1.3. Allocating Memory

- Mark the pages allocated as 1 in the bitmap and zero out the contents of the pages

```c
static void *__init __alloc_bootmem_core (bootmem_data_t *bdata,
unsigned long size, unsigned long align, unsigned long goal)
{
    unsigned long i, start = 0;
    void *ret;
    unsigned long offset, remaining_size;
    unsigned long areysize, preferred, incr;
    unsigned long eidx = bdata->node_low_pfn -
        (bdata->node_boot_start >> PAGE_SHIFT);
    if (!size) BUG();
    if (align & (align-1))
        BUG();
    offset = 0;
    if (align &&
        (bdata->node_boot_start & (align - 1UL)) != 0)
        offset = (align - (bdata->node_boot_start &
            (align - 1UL)));
    offset >>= PAGE_SHIFT;
```

Function preamble, make sure the parameters are sane

144 The parameters are;

    `bdata` is the bootmem for the struct being allocated from
    `size` is the size of the requested allocation
    `align` is the desired alignment for the allocation. Must be a power of 2
    `goal` is the preferred address to allocate above if possible

151 Calculate the ending bit index `eidx` which returns the highest page index that may be used for the allocation

154 Call `BUG()` if a request size of 0 is specified

156-156 If the alignment is not a power of 2, call `BUG()`

159 The default offset for alignments is 0

160 If an alignment has been specified and...

161 And the requested alignment is the same alignment as the start of the node then calculate the offset to use
The offset to use is the requested alignment masked against the lower bits of the starting address. In reality, this offset will likely be identical to align for the prevalent values of align.

```c
if (goal && (goal >= bdata->node_boot_start) &&
    ((goal >> PAGE_SHIFT) < bdata->node_low_pfn)) {
    preferred = goal - bdata->node_boot_start;
} else
    preferred = 0;
```

```c
preferred = ((preferred + align - 1) & ~(align - 1))
            >> PAGE_SHIFT;
```

```c
preferred += offset;
```

```c
areasize = (size+PAGE_SIZE-1)/PAGE_SIZE;
```

```c
incr = align >> PAGE_SHIFT ? : 1;
```

Calculate the starting PFN to start scanning from based on the goal parameter.

```c
if a goal has been specified and the goal is after the starting address for this node and
the PFN of the goal is less than the last PFN addressable by this node then ....
```

```c
The preferred offset to start from is the goal minus the beginning of the memory
addressable by this node
```

```c
Else the preferred offset is 0
```

```c
Adjust the preferred address to take the offset into account so that the address
will be correctly aligned
```

```c
The number of pages that will be affected by this allocation is stored in areasize
```

```c
incr is the number of pages that have to be skipped to satisfy alignment requirements
if they are over one page
```

```c
restart_scan:
for (i = preferred; i < eidx; i += incr) {
    unsigned long j;
    if (test_bit(i, bdata->node_bootmem_map))
        continue;
    for (j = i + 1; j < i + areasize; ++j) {
        if (j >= eidx)
            goto fail_block;
        if (test_bit (j, bdata->node_bootmem_map))
            goto fail_block;
    }
    start = i;
    goto found;
```
1.3. Allocating Memory

193 fail_block:;
194 }
195 if (preferred) {
196    preferred = offset;
197    goto restart_scan;
198 }
199 return NULL;

Scan through memory looking for a block large enough to satisfy this request

180 If the allocation could not be satisfied starting from goal, this label is jumped back to for rescanning

181-194 Starting from preferred, scan linearly searching for a free block large enough to satisfy the request. Walk the address space in incr steps to satisfy alignments greater than one page. If the alignment is less than a page, incr will just be 1

183-184 Test the bit, if it is already 1, it is not free so move to the next page

185-190 Scan the next areasize number of pages and see if they are also free. It fails if the end of the addressable space is reached (eidx) or one of the pages is already in use

191-192 A free block is found so record the start and jump to the found block

195-198 The allocation failed so start again from the beginning

199 If that also failed, return NULL which will result in a kernel panic

200 found:
201   if (start >= eidx)
202      BUG();
203 209   if (align <= PAGE_SIZE
210      && bdata->last_offset && bdata->last_pos+1 == start) {
211      offset = (bdata->last_offset+align-1) & ~(align-1);
212      if (offset > PAGE_SIZE)
213         BUG();
214      remaining_size = PAGE_SIZE-offset;
215      if (size < remaining_size) {
216         areasisze = 0;
217         // last_pos unchanged
218         bdata->last_offset = offset+size;
219         ret = phys_to_virt(bdata->last_pos*PAGE_SIZE + offset +
220                          bdata->node_boot_start);
221     } else {
222         remaining_size = size - remaining_size;
223         areasisze = (remaining_size+PAGE_SIZE-1)/PAGE_SIZE;
1.3. Allocating Memory

```
ret = phys_to_virt(bdata->last_pos*PAGE_SIZE +
    offset +
    bdata->node_boot_start);

bdata->last_pos = start+areasize-1;
bdata->last_offset = remaining_size;

} else {
    bdata->last_offset &= ~PAGE_MASK;

    bdata->last_pos = start + areasize - 1;
bdata->last_offset = size & ~PAGE_MASK;
    ret = phys_to_virt(start * PAGE_SIZE +
        bdata->node_boot_start);
}
```

Test to see if this allocation may be merged with the previous allocation.

201-202 Check that the start of the allocation is not after the addressable memory. This check was just made so it is redundant.

209-230 Try and merge with the previous allocation if the alignment is less than a PAGE_SIZE, the previously page has space in it (last_offset != 0) and that the previously used page is adjacent to the page found for this allocation.

231-234 Else record the pages and offset used for this allocation to be used for merging with the next allocation.

211 Update the offset to use to be aligned correctly for the requested align.

212-213 If the offset now goes over the edge of a page, BUG() is called. This condition would require a very poor choice of alignment to be used. As the only alignment commonly used is a factor of PAGE_SIZE, it is impossible for normal usage.

214 remaining_size is the remaining free space in the previously used page.

215-221 If there is enough space left in the old page then use the old page totally and update the bootmem_data struct to reflect it.

221-228 Else calculate how many pages in addition to this one will be required and update the bootmem_data.

216 The number of pages used by this allocation is now 0.

218 Update the last_offset to be the end of this allocation.

219 Calculate the virtual address to return for the successful allocation.

222 remaining_size is how space will be used in the last page used to satisfy the allocation.

223 Calculate how many more pages are needed to satisfy the allocation.
1.4. Freeing Memory

224 Record the address the allocation starts from

226 The last page used is the start page plus the number of additional pages required to satisfy this allocation areasize

227 The end of the allocation has already been calculated

229 If the offset is at the end of the page, make it 0

231 No merging took place so record the last page used to satisfy this allocation

232 Record how much of the last page was used

233 Record the starting virtual address of the allocation

238 for (i = start; i < start+areasize; i++)
239     if (test_and_set_bit(i, bdata->node_bootmem_map))
240         BUG();
241     memset(ret, 0, size);
242     return ret;
243 }

Mark the pages allocated as 1 in the bitmap and zero out the contents of the pages

238-240 Cycle through all pages used for this allocation and set the bit to 1 in the bitmap. If any of them are already 1, then a double allocation took place so call BUG()

241 Zero fill the pages

242 Return the address of the allocation

1.4 Freeing Memory

Function: free_bootmem (mm/bootmem.c)

294 void __init free_bootmem_node (pg_data_t *pgdat,
               unsigned long physaddr, unsigned long size)
295 {
296     return(free_bootmem_core(pgdat->bdata, physaddr, size));
297 }

316 void __init free_bootmem (unsigned long addr, unsigned long size)
317 {
318     return(free_bootmem_core(contig_page_data.bdata, addr, size));
319 }

296 Call the core function with the corresponding bootmem data for the requested node

318 Call the core function with the bootmem data for contig_page_data
Function: free_bootmem_core ($mm/bootmem.c$)

103 static void __init free_bootmem_core(bootmem_data_t *bdata,  
   unsigned long addr,  
   unsigned long size)

104 {
105   unsigned long i;
106   unsigned long start;
107   unsigned long sidx;
108   unsigned long eidx = (addr + size -  
       bdata->node_boot_start)/PAGE_SIZE;
109   unsigned long end = (addr + size)/PAGE_SIZE;
110
111   if (!size) BUG();
112   if (end > bdata->node_low_pfn)
113     BUG();
114
115   /*
116    * Round up the beginning of the address.
117    */
118   start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
119   sidx = start - (bdata->node_boot_start/PAGE_SIZE);
120
121   for (i = sidx; i < eidx; i++) {
122     if (!test_and_clear_bit(i, bdata->node_bootmem_map))
123       BUG();
124   }
125 }
126
127 Calculate the end index affected as eidx
128
129 The end address is the end of the affected area rounded down to the nearest page if it is not already page aligned
130
131 If a size of 0 is freed, call BUG
132
133-134 If the end PFN is after the memory addressable by this node, call BUG
135
136 Round the starting address up to the nearest page if it is not already page aligned
137
138 Calculate the starting index to free
139
140-141 For all full pages that are freed by this action, clear the bit in the boot bitmap.
   If it is already 0, it is a double free or is memory that was never used so call BUG
1.5  Retiring the Boot Memory Allocator

**Function: mem_init** *(arch/i386/mm/init.d)*

The important part of this function for the boot memory allocator is that it calls `free_pages_init()`. The function is broken up into the following tasks:

- Function preamble, set the PFN within the global `mem_map` for the location of high memory and zero out the system wide zero page
- Call `free_pages_init()`
- Print out an informational message on the availability of memory in the system
- Check the CPU supports PAE if the config option is enabled and test the WP bit on the CPU. This is important as without the WP bit, the function `verify_write()` has to be called for every write to userspace from the kernel. This only applies to old processors like the 386
- Fill in entries for the userspace portion of the PGD for `swapper_pg_dir`, the kernel page tables. The zero page is mapped for all entries

```c
void __init mem_init(void)
{
    int codesize, reservedpages, datasize, initsize;

    if (!mem_map)
        BUG();
    set_max_mapnr_init();
    high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);

    /* clear the zero-page */
    memset(empty_zero_page, 0, PAGE_SIZE);

    reservedpages = free_pages_init();
}
```

514 This function records the PFN high memory starts in `mem_map` (highmem_start_page), the maximum number of pages in the system (max_mapnr and num_physpages) and finally the maximum number of pages that may be mapped by the kernel (num_mappedpages)

516 `high_memory` is the virtual address where high memory begins

519 Zero out the system wide zero page
Call `free_pages_init()` which tells the boot memory allocator to retire itself as well as initialising all pages in high memory for use with the buddy allocator.

```c
523 codesize = (unsigned long) &_etext - (unsigned long) &_text;
524 datasize = (unsigned long) &_edata - (unsigned long) &_etext;
525 initsize = (unsigned long) &__init_end - (unsigned long)
             &__init_begin;
526
527 printk(KERN_INFO "Memory: %luk/%luk available (%dk kernel code,
        %dk reserved, %dk data, %dk init, %ldk highmem)\n",
        (unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
        max_mapnr << (PAGE_SHIFT-10),
        codesize >> 10,
        reservedpages << (PAGE_SHIFT-10),
        datasize >> 10,
        initsize >> 10,
        (unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
        );
```

Print out an informational message.

523 Calculate the size of the code segment, data segment and memory used by initialisation code and data (all functions marked `__init` will be in this section).

527-535 Print out a nice message on how the availability of memory and the amount of memory consumed by the kernel.

536
537 #if CONFIG_X86_PAE
538   if (!cpu_has_pae)
539     panic("cannot execute a PAE-enabled kernel on a PAE-less
      CPU!");
540 #endif
541 if (boot_cpu_data.wp_works_ok < 0)
542   test_wp_bit();
543
538-539 If PAE is enabled but the processor does not support it, panic.

541-542 Test for the availability of the WP bit.

550 #ifndef CONFIG_SMP
551   zap_low_mappings();
552 #endif
553
554 }

551 Cycle through each PGD used by the userspace portion of `swapper_pg_dir` and map the zero page to it.
1.5. Retiring the Boot Memory Allocator

**Function: free_pages_init** *(arch/i386/mm/init.c)*

This function has two important functions, to call `free_all_bootmem()` to retire the boot memory allocator and to free all high memory pages to the buddy allocator.

```c
static int __init free_pages_init(void)
{
    extern int ppro_with_ram_bug(void);
    int bad_ppro, reservedpages, pfn;
    bad_ppro = ppro_with_ram_bug();
    /* this will put all low memory onto the freelists */
    totalram_pages += free_all_bootmem();
    reservedpages = 0;
    for (pfn = 0; pfn < max_low_pfn; pfn++) {
        /* Only count reserved RAM pages */
        if (page_is_ram(pfn) && PageReserved(mem_map+pfn))
            reservedpages++;
    }
    #ifdef CONFIG_HIGHMEM
    for (pfn = highend_pfn-1; pfn >= highstart_pfn; pfn--)
    one_highpage_init((struct page *) (mem_map + pfn), pfn, bad_ppro);
    totalram_pages += totalhigh_pages;
    #endif
    return reservedpages;
}
```

There is a bug in the Pentium Pros that prevent certain pages in high memory being used. The function `ppro_with_ram_bug()` checks for its existance.

**Call free_all_bootmem()** to retire the boot memory allocator

**Cycle through all of memory and count the number of reserved pages that were left over by the boot memory allocator**

**For each page in high memory, call one_highpage_init().** This function clears the PG_reserved bit, sets the PG_high bit, sets the count to 1, calls `__free_pages()` to give the page to the buddy allocator and increments the totalhigh_pages count. Pages which kill buggy Pentium Pro's are skipped

**Function: free_all_bootmem** *(mm/bootmem.c)*

```c
unsigned long __init free_all_bootmem_node (pg_data_t *pgdat)
```
1.5. Retiring the Boot Memory Allocator

For NUMA, simply call the core function with the specified pgdat
For UMA, call the core function with the only node contig_page_data

Function: free_all_bootmem_core (mm/bootmem.c)
This is the core function which “retires” the boot memory allocator. It is divided into two major tasks

- For all unallocated pages known to the allocator for this node;
  - Clear the PG_reserved flag in its struct page
  - Set the count to 1
  - Call __free_pages() so that the buddy allocator (discussed next chapter) can build its free lists
- Free all pages used for the bitmap and free to them to the buddy allocator

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;
  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)) {
      ClearPageReserved(page);
      set_page_count(page, 1);
      __free_page(page);
    }
  }
  total += count;
}
If no map is available, it means that this node has already been freed and something woeful is wrong with the architecture dependent code so call \texttt{BUG()}

A running count of the number of pages given to the buddy allocator

\texttt{idx} is the last index that is addressable by this node

Cycle through all pages addressable by this node

If the page is marked free then...

Increase the running count of pages given to the buddy allocator

Clear the \texttt{PG\_reserved} flag

Set the count to 1 so that the buddy allocator will think this is the last user of the page and place it in its free lists

Call the buddy allocator free function

\texttt{total} will come the total number of pages given over by this function

Get the struct page that is at the beginning of the bootmem map

Count of pages freed by the bitmap

For all pages used by the bitmap, free them to the buddy allocator the same way the previous block of code did

Set the bootmem map to NULL to prevent it been freed a second time by accident

Return the total number of pages freed by this function
Chapter 2

Physical Page Management

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc_pages(unsigned int gfp_mask, unsigned int order)</td>
<td>Allocate $2^order$ number of pages and returns a struct page</td>
</tr>
<tr>
<td>__get_dma_pages(unsigned int gfp_mask, unsigned int order)</td>
<td>Allocate $2^order$ number of pages from the DMA zone and return a struct page</td>
</tr>
<tr>
<td>__get_free_pages(unsigned int gfp_mask, unsigned int order)</td>
<td>Allocate $2^order$ number of pages and return a virtual address</td>
</tr>
<tr>
<td>alloc_page(unsigned int gfp_mask)</td>
<td>Allocate a single page and return a struct address</td>
</tr>
<tr>
<td>__get_free_page(unsigned int gfp_mask)</td>
<td>Allocate a single page and return a virtual address</td>
</tr>
<tr>
<td>get_free_page(unsigned int gfp_mask)</td>
<td>Allocate a single page, zero it and return a virtual address</td>
</tr>
</tbody>
</table>

Table 2.1: Physical Pages Allocation API

2.1 Allocating Pages

**Function:** alloc_pages (include/linux/mm.h)

The toplevel alloc_pages() function is declared as
2.1. Allocating Pages

Figure 2.1: Call Graph: alloc_pages()

428 static inline struct page * alloc_pages(unsigned int gfp_mask,
                                          unsigned int order)
429 {
433     if (order >= MAX_ORDER)
434         return NULL;
435     return _alloc_pages(gfp_mask, order);
436 }

428 The gfp_mask (Get Free Pages) flags tells the allocator how it may behave. For example GFP_WAIT is not set, the allocator will not block and instead return NULL if memory is tight. The order is the power of two number of pages to allocate

433-434 A simple debugging check optimized away at compile time

435 This function is described next

Function: _alloc_pages (mm/page_alloc.c)

The function _alloc_pages() comes in two varieties. The first is in mm/page_alloc.c is designed to only work with UMA architectures such as the x86. It only refers to the static node contig_page_data. The second is in mm/numa.c and is a simple extension. It uses a node-local allocation policy which means that memory will be allocated from the bank closest to the processor. For the purposes of this document, only the mm/page_alloc.c version will be examined but for completeness the reader should glance at the functions _alloc_pages() and _alloc_pages_pgdat() in mm/numa.c
The ifndef is for UMA architectures like the x86. NUMA architectures used the
_alloc_pages() function in mm/numa.c which employs a node local policy for al-
llocations

The gfp_mask flags tell the allocator how it may behave. The order is the power of
two number of pages to allocate

node_zonelists is an array of preferred fallback zones to allocate from. It is initialised
in build_zonelists() The lower 16 bits of gfp_mask indicate what zone is preferable
to allocate from. gfp_mask & GFP_ZONEMASK will give the index in node_zonelists
we prefer to allocate from.

Function: __alloc_pages (mm/page_alloc.c)
At this stage, we’ve reached what is described as the "heart of the zoned buddy allocator", the __alloc_pages() function. It is responsible for cycling through the fallback zones and
selecting one suitable for the allocation. If memory is tight, it will take some steps to address
the problem. It will wake kswapd and if necessary it will do the work of kswapd manually.

struct page * __alloc_pages(unsigned int gfp_mask, unsigned int order,
    zonelist_t *zonelist)
{
    unsigned long min;
    zone_t **zone, * classzone;
    struct page * page;
    int freed;

    zone = zonelist->zones;
    classzone = *zone;
    if (classzone == NULL)
        return NULL;
    min = 1UL << order;
    for (; ; ) {
        zone_t *z = *(zone++);
        if (!z)
            break;

        min += z->pages_low;
        if (z->free_pages > min) {
            page = rmqueue(z, order);
        }
    }
}
if (page)
    return page;
}
}

classzone->need_balance = 1;
b();
if (waitqueue_active(&kswapd_wait))
    wake_up_interruptible(&kswapd_wait);

zone = zonelist->zones;
min = 1UL << order;
for (;; ) {
    unsigned long local_min;
    zone_t *z = *(zone++);
    if (!z)
        break;
    local_min = z->pages_min;
    if (!((gfp_mask & __GFP_WAIT))
        local_min >>= 2;
    min += local_min;
    if (z->free_pages > min) {
        page = rmqueue(z, order);
        if (page)
            return page;
    }
}

/* here we’re in the low on memory slow path */

rebalance:
if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {
    zone = zonelist->zones;
    for (; ; ) {
        zone_t *z = *(zone++);
        if (!z)
            break;
        page = rmqueue(z, order);
        if (page)
            return page;
    }
    return NULL;
}
2.1. Allocating Pages

    /* Atomic allocations - we can’t balance anything */
    if (!(gfp_mask & __GFP_WAIT))
        return NULL;

    page = balance_classzone(classzone, gfp_mask, order, &freed);
    if (page)
        return page;

    zone = zonelist->zones;
    min = 1UL << order;
    for (; ;) {
        zone_t *z = *(zone++);
        if (!z)
            break;

        min += z->pages_min;
        if (z->free_pages > min) {
            page = rmqueue(z, order);
            if (page)
                return page;
        }
    }

    /* Don’t let big-order allocations loop */
    if (order > 3)
        return NULL;

    /* Yield for kswapd, and try again */
    yield();
    goto rebalance;

Set zone to be the preferred zone to allocate from

The preferred zone is recorded as the classzone. If one of the pages low watermarks is
reached later, the classzone is marked as needing balance

An unnecessary sanity check. build_zonelists() would need to be seriously
broken for this to happen

This style of block appears a number of times in this function. It reads as
"cycle through all zones in this fallback list and see can the allocation be satisfied
without violating watermarks. Note that the pages_low for each fallback zone is
added together. This is deliberate to reduce the probability a fallback zone will be
used.

z is the zone currently been examined. zone is moved to the next fallback zone
2.1. Allocating Pages

341-342 If this is the last zone in the fallback list, break

344 Increment the number of pages to be allocated by the watermark for easy comparisons. This happens for each zone in the fallback zones. While it would appear to be a bug, it is assumed that this behavior is intended to reduce the probability a fallback zone is used.

345-349 Allocate the page block if it can be assigned without reaching the pages_min watermark. rmqueue() is responsible from removing the block of pages from the zone

347-348 If the pages could be allocated, return a pointer to them

352 Mark the preferred zone as needing balance. This flag will be read later by kswapd

353 This is a memory barrier. It ensures that all CPU’s will see any changes made to variables before this line of code. This is important because kswapd could be running on a different processor to the memory allocator.

354-355 Wake up kswapd if it is asleep

357-358 Begin again with the first preferred zone and min value

360-374 Cycle through all the zones. This time, allocate the pages if they can be allocated without hitting the pages_min watermark

365 local_min how low a number of free pages this zone can have

366-367 If the process can not wait or reschedule (__GFP_WAIT is clear), then allow the zone to be put in further memory pressure than the watermark normally allows

378 This label is returned to after an attempt is made to synchronously free pages. From this line on, the low on memory path has been reached. It is likely the process will sleep

379-391 These two flags are only set by the OOM killer. As the process is trying to kill itself cleanly, allocate the pages if at all possible as it is known they will be freed very soon

394-395 If the calling process can not sleep, return NULL as the only way to allocate the pages from here involves sleeping

397 This function does the work of kswapd in a synchronous fashion. The principle difference is that instead of freeing the memory into a global pool, it is kept for the process using the current->local_pages field

398-399 If a page block of the right order has been freed, return it. Just because this is NULL does not mean an allocation will fail as it could be a higher order of pages that was released

403-414 This is identical to the block above. Allocate the page blocks if it can be done without hitting the pages_min watermark
2.1. Allocating Pages

Satisfying a large allocation like $2^4$ number of pages is difficult. If it has not been satisfied by now, it is better to simply return NULL.

Yield the processor to give kswapd a chance to work.

Attempt to balance the zones again and allocate.

**Function: rmqueue** *(mm/page_alloc.c)*

This function is called from `__alloc_pages()`. It is responsible for finding a block of memory large enough to be used for the allocation. If a block of memory of the requested size is not available, it will look for a larger order that may be split into two buddies. The actual splitting is performed by the `expand()` function.

```c
static FASTCALL(struct page *rmqueue(zone_t *zone, unsigned int order));
static struct page * 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);
            spin_unlock_irqrestore(&zone->lock, flags);
        }
        if (PageLRU(page))
            set_page_count(page, 1);
    }
    if (BAD_RANGE(zone,page))
        BUG();
    if (PageLRU(page))
        set_page_count(page, 1);
    return page;
}
```

417-418 Satisfying a large allocation like $2^4$ number of pages is difficult. If it has not been satisfied by now, it is better to simply return NULL.

421 Yield the processor to give kswapd a chance to work.

422 Attempt to balance the zones again and allocate.
2.1. Allocating Pages

231     BUG();
232     if (PageActive(page))
233         BUG();
234     return page;
235 }
236     curr_order++;
237     area++;
238 } while (curr_order < MAX_ORDER);
239 spin_unlock_irqrestore(&zone->lock, flags);
240
241 return NULL;
242 }

199 The parameters are the zone to allocate from and what order of pages are required

201 Because the free_area is an array of linked lists, the order may be used an an index
within the array

207 Acquire the zone lock

208-238 This while block is responsible for finding what order of pages we will need to
allocate from. If there isn’t a free block at the order we are interested in, check the
higher blocks until a suitable one is found

209 head is the list of free page blocks for this order

210 curr is the first block of pages

212-235 If there is a free page block at this order, then allocate it

215 page is set to be a pointer to the first page in the free block

216-217 Sanity check that checks to make sure the page this page belongs to this zone
and is within the zone_mem_map. It is unclear how this could possibly happen without
severe bugs in the allocator itself that would place blocks in the wrong zones

218 As the block is going to be allocated, remove it from the free list

219 index treats the zone_mem_map as an array of pages so that index will be the offset
within the array

220-221 Toggle the bit that represents this pair of buddies. MARK_USED() is a macro which
calculates which bit to toggle

222 Update the statistics for this zone. 1UL << order is the number of pages been
allocated

224 expand() is the function responsible for splitting page blocks of higher orders

225 No other updates to the zone need to take place so release the lock
Show that the page is in use

Sanity checks

Page block has been successfully allocated so return it

If a page block was not free of the correct order, move to a higher order of page blocks and see what can be found there

No other updates to the zone need to take place so release the lock

No page blocks of the requested or higher order are available so return failure

**Function: expand** (*mm/page_alloc.c*)

This function splits page blocks of higher orders until a page block of the needed order is available.

```c
177 static inline struct page * expand (zone_t *zone,
    struct page *page,
    unsigned long index,
    int low,
    int high,
    free_area_t * area)
179 {
    unsigned long size = 1 << high;
181
182     while (high > low) {
183         if (BAD_RANGE(zone,page))
184             BUG();
185         area--;
186         high--;
187         size >>= 1;
188         list_add(&(page)->list, &(area)->free_list);
189         MARK_USED(index, high, area);
190         index += size;
191         page += size;
192     }
193     if (BAD_RANGE(zone,page))
194         BUG();
195     return page;
196 }
```

The parameters are

- `zone` is where the allocation is coming from
- `page` is the first page of the block been split
- `index` is the index of page within `mem_map`
2.2. Free Pages

low is the order of pages needed for the allocation
high is the order of pages that is been split for the allocation
area is the free_area_t representing the high order block of pages

size is the number of pages in the block that is to be split

Keep splitting until a block of the needed page order is found
Sanity check that checks to make sure the page this page belongs to this zone and is within the zone_mem_map
area is now the next free_area_t representing the lower order of page blocks
high is the next order of page blocks to be split
The size of the block been split is now half as big
Of the pair of buddies, the one lower in the mem_map is added to the free list for the lower order
Toggle the bit representing the pair of buddies
index now the index of the second buddy of the newly created pair
page now points to the second buddy of the newly created paid
Sanity check
The blocks have been successfully split so return the page

2.2 Free Pages

__free_pages(struct page *page, unsigned int order)
   Free an order number of pages from the given page

__free_page(struct page *page)
   Free a single page

free_page(void *addr)
   Free a page from the given virtual address

Table 2.2: Physical Pages Free API
2.2. Free Pages

Confusingly, the opposite to alloc_pages() is not free_pages(), it is __free_pages().
free_pages() is a helper function which takes an address as a parameter, it will be discussed in a later section.

```c
void __free_pages(struct page *page, unsigned int order) {
    if (!PageReserved(page) && put_page_testzero(page))
        __free_pages_ok(page, order);
}
```

The parameters are the page we wish to free and what order block it is

Sanity checked. PageReserved() indicates that the page is reserved by the boot memory allocator. put_page_testzero() decrements the usage count and makes sure it is zero

Call the function that does all the hard work

**Function: **__free_pages_ok (mm/page_alloc.c)

This function will do the actual freeing of the page and coalesce the buddies if possible.

```c
static void FASTCALL(__free_pages_ok (struct page *page, unsigned int order));
static void __free_pages_ok (struct page *page, unsigned int order) {
    unsigned long index, page_idx, mask, flags;
    free_area_t *area;
    struct page *base;
    zone_t *zone;
    ```
if (PageLRU(page)) {
    if (unlikely(in_interrupt()))
        BUG();
    lru_cache_del(page);
}

if (page->buffers)
    BUG();
if (page->mapping)
    BUG();
if (!VALID_PAGE(page))
    BUG();
if (PageLocked(page))
    BUG();
if (PageActive(page))
    BUG();
page->flags &= ~(1<<PG_referenced) | (1<<PG_dirty));

if (current->flags & PF_FREE_PAGES)
    goto local_freelist;

zone = page_zone(page);
mask = (~0UL) << order;
base = zone->zone_mem_map;
page_idx = page - base;
if (page_idx & ~mask)
    BUG();
index = page_idx >> (1 + order);

area = zone->free_area + order;
spin_lock_irqsave(&zone->lock, flags);
zone->free_pages -= mask;

while (mask + (1 << (MAX_ORDER-1))) {
    struct page *buddy1, *buddy2;
    if (area >= zone->free_area + MAX_ORDER)
        BUG();
    if (!__test_and_change_bit(index, area->map))
        /*
           * the buddy page is still allocated.
        */
2.2. Free Pages

```
138     */
139     break;
140     /*
141     * Move the buddy up one level.
142     * This code is taking advantage of the identity:
143     *    -mask = 1+~mask
144     */
145     buddy1 = base + (page_idx ^ -mask);
146     buddy2 = base + page_idx;
147     if (BAD_RANGE(zone,buddy1))
148         BUG();
149     if (BAD_RANGE(zone,buddy2))
150         BUG();
151     list_del(&buddy1->list);
152     mask <<= 1;
153     area++;
154     index >>= 1;
155     page_idx &= mask;
156     }
157     list_add(&base + page_idx)->list, &area->free_list);
158     spin_unlock_irqrestore(&zone->lock, flags);
159     return;
160 }
161
162 local_freelist:
163     if (current->nr_local_pages)
164         goto back_local_freelist;
165     if (in_interrupt())
166         goto back_local_freelist;
167     list_add(&page->list, &current->local_pages);
168     page->index = order;
169     current->nr_local_pages++;
170 }
171
82 The parameters are the beginning of the page block to free and what order number of
83 pages are to be freed.
84
32 A dirty page on the LRU will still have the LRU bit set when pinned for IO. It is just
85 freed directly when the IO is complete so it just has to be removed from the LRU list
86
99-108 Sanity checks
109 The flags showing a page has being referenced and is dirty have to be cleared because
110 the page is now free and not in use
```
2.2. Free Pages

111-112 If this flag is set, the pages freed are to be kept for the process doing the freeing. This is set during page allocation if the caller is freeing the pages itself rather than waiting for kswapd to do the work.

115 The zone the page belongs to is encoded within the page flags. The page_zone() macro returns the zone.

117 The calculation of mask is discussed in companion document. It is basically related to the address calculation of the buddy.

118 base is the beginning of this zone_mem_map. For the buddy calculation to work, it was to be relative to an address 0 so that the addresses will be a power of two.

119 page_idx treats the zone_mem_map as an array of pages. This is the index page within the map.

120-121 If the index is not the proper power of two, things are severely broken and calculation of the buddy will not work.

122 This index is the bit index within free_area→map.

124 area is the area storing the free lists and map for the order block the pages are been freed from.

126 The zone is about to be altered so take out the lock.

128 Another side effect of the calculation of mask is that -mask is the number of pages that are to be freed.

130-157 The allocator will keep trying to coalesce blocks together until it either cannot merge or reaches the highest order that can be merged. mask will be adjusted for each order block that is merged. When the highest order that can be merged is reached, this while loop will evaluate to 0 and exit.

133-134 If by some miracle, mask is corrupt, this check will make sure the free_area array will not be read beyond the end.

135 Toggle the bit representing this pair of buddies. If the bit was previously zero, both buddies were in use. As this buddy is been freed, one is still in use and cannot be merged.

145-146 The calculation of the two addresses is discussed in the companion document.

147-150 Sanity check to make sure the pages are within the correct zone_mem_map and actually belong to this zone.

152 The buddy has been freed so remove it from any list it was part of.

153-156 Prepare to examine the higher order buddy for merging.

153 Move the mask one bit to the left for order \(2^{k+1}\).
area is a pointer within an array so area++ moves to the next index

The index in the bitmap of the higher order

The page index within the zone_mem_map for the buddy to merge

As much merging as possible as completed and a new page block is free so add it to the free_list for this order

Changes to the zone is complete so free the lock and return

This is the code path taken when the pages are not freed to the main pool but instead are reserved for the process doing the freeing.

If the process already has reserved pages, it is not allowed to reserve any more so return back

An interrupt does not have process context so it has to free in the normal fashion. It is unclear how an interrupt could end up here at all. This check is likely to be bogus and impossible to be true

Add the page block to the list for the processes local_pages

Record what order allocation it was for freeing later

Increase the use count for nr_local_pages

### 2.3 Page Allocate Helper Functions

This section will cover miscellaneous helper functions and macros the Buddy Allocator uses to allocate pages. Very few of them do "real" work and are available just for the convenience of the programmer.

**Function: alloc_page** *(include/linux/mm.h)*

This trivial macro just calls alloc_pages() with an order of 0 to return 1 page. It is declared as follows

```c
#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
```

**Function: __get_free_page** *(include/linux/mm.h)*

This trivial function calls __get_free_pages() with an order of 0 to return 1 page. It is declared as follows

```c
#define __get_free_page(gfp_mask) \ 
__get_free_pages((gfp_mask),0)
```
2.3. Page Allocate Helper Functions

**Function: **`__get_free_pages` *(mm/page_alloc.c)*

This function is for callers who do not want to worry about pages and only get back an address it can use. It is declared as follows

```c
unsigned long __get_free_pages(unsigned int gfp_mask,
                             unsigned int order)
```

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

gfp_mask are the flags which affect allocator behaviour. Order is the power of 2 number of pages required.

`alloc_pages()` does the work of allocating the page block. See Section 2.1

`page_address()` returns the physical address of the page

**Function: **`__get_dma_pages` *(include/linux/mm.h)*

This is of principle interest to device drivers. It will return memory from ZONE_DMA suitable for use with DMA devices. It is declared as follows

```c
#define __get_dma_pages(gfp_mask, order) \
    __get_free_pages((gfp_mask) | GFP_DMA,(order))
```

The gfp_mask is or-ed with GFP_DMA to tell the allocator to allocate from ZONE_DMA

**Function: **`get_zeroed_page` *(mm/page_alloc.c)*

This function will allocate one page and then zero out the contents of it. It is declared as follows

```c
unsigned long get_zeroed_page(unsigned int gfp_mask)
```

```c
{
    struct page * page;
    page = alloc_pages(gfp_mask, 0);
    if (page) {
        void *address = page_address(page);
        clear_page(address);
        return (unsigned long) address;
    }
    return 0;
}
```
The flags which affect allocator behaviour.
alloc_pages() does the work of allocating the page block. See Section 2.1
page_address() returns the physical address of the page
clear_page() will fill the contents of a page with zero
Return the address of the zeroed page

2.4 Page Free Helper Functions

This section will cover miscellaneous helper functions and macros the Buddy Allocator uses to free pages. Very few of them do "real" work and are available just for the convenience of the programmer. There is only one core function for the freeing of pages and it is discussed in Section 2.2.

The only functions then for freeing are ones that supply an address and for freeing a single page.

Function: free_pages (mm/page_alloc.c)
This function takes an address instead of a page as a parameter to free. It is declared as follows

```
void free_pages(unsigned long addr, unsigned int order)
{
    if (addr != 0)
        __free_pages(virt_to_page(addr), order);
}
```

The function is discussed in Section 2.2. The macro virt_to_page() returns the struct page for the addr

Function: __free_page (include/linux/mm.h)
This trivial macro just calls the function __free_pages() (See Section 2.2 with an order 0 for 1 page. It is declared as follows

```
#define __free_page(page) __free_pages((page), 0)
```
Chapter 3

Non-Contiguous Memory Allocation

3.1 Allocating A Non-Contiguous Area

Table 3.1: Non-Contiguous Memory Allocation API

Function: \texttt{vmalloc} \texttt{(include/linux/vmalloc.h)}

They only difference between these macros is the GFP flags (See the companion document for an explanation of GFP flags). The size parameter is page aligned by \texttt{__vmalloc()}

```c
33 static inline void * vmalloc (unsigned long size)
34 {  
35     return __vmalloc(size, GFP_KERNEL | GFP_HIGHMEM, PAGE_KERNEL);
36 }
37
41
42 static inline void * vmalloc_dma (unsigned long size)
43 {  
44     return __vmalloc(size, GFP_KERNEL|GFP_DMA, PAGE_KERNEL);
45 }
46
46
```
3.1. Allocating A Non-Contiguous Area

The flags indicate that to use either ZONE_NORMAL or ZONE_HIGHMEM as necessary

The flag indicates to only allocate from ZONE_DMA

Only physical pages from ZONE_NORMAL will be allocated

**Function: **__vmalloc (mm/vmalloc.c)

This function has three tasks. It page aligns the size request, asks get_vm_area() to find an area for the request and uses vmalloc_area_pages() to allocate the PTEs for the pages.

```c
void * __vmalloc (unsigned long size, int gfp_mask, pgprot_t prot)
{
    void * addr;
    struct vm_struct *area;

    size = PAGE_ALIGN(size);
    if (!size || (size >> PAGE_SHIFT) > num_physpages) {
        BUG();
        return NULL;
    }
    area = get_vm_area(size, VM_ALLOC);
    return __vmalloc(size, GFP_KERNEL, PAGE_KERNEL);
}
```

Figure 3.1: Call Graph: vmalloc()
3.1. Allocating A Non-Contiguous Area

if (!area)
    return NULL;
addr = area->addr;
if (vmalloc_area_pages(VMALLOC_VMADDR(addr),
    size, gfp_mask, prot)) {
    vfree(addr);
    return NULL;
}
return addr;

The parameters are the size to allocate, the GFP flags to use for allocation and what protection to give the PTE.

Align the size to a page size.

Sanity check. Make sure the size is not 0 and that the size requested is not larger than the number of physical pages has been requested.

Find an area of virtual address space to store the allocation (See Section 3.1).

The addr field has been filled by `get_vm_area()`.

Allocate the PTE entries needed for the allocation with `vmalloc_area_pages()`. If it fails, a non-zero value -ENOMEM is returned.

If the allocation fails, free any PTEs, pages and descriptions of the area.

Return the address of the allocated area.

**Function: get_vm_area (mm/vmalloc.c)**

To allocate an area for the `vm_struct`, the slab allocator is asked to provide the necessary memory via `kmalloc()`. It then searches the `vm_struct` list linearly looking for a region large enough to satisfy a request, including a page pad at the end of the area.

```
struct vm_struct * get_vm_area(unsigned long size, unsigned long flags)
{
    unsigned long addr;
    struct vm_struct **p, *tmp, *area;
    area = (struct vm_struct *) kmalloc(sizeof(*area), GFP_KERNEL);
    if (!area)
        return NULL;
    size += PAGE_SIZE;
    if(!size)
        return NULL;
    addr = VMALLOC_START;
    write_lock(&vmlist_lock);
```
for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
    if ((size + addr) < addr)
        goto out;
    if (size + addr <= (unsigned long) tmp->addr)
        break;
    addr = tmp->size + (unsigned long) tmp->addr;
    if (addr > VMALLOC_END-size)
        goto out;
    area->flags = flags;
    area->addr = (void *)addr;
    area->size = size;
    area->next = *p;
    *p = area;
    write_unlock(&vmlist_lock);
    return area;
}

out:
    write_unlock(&vmlist_lock);
    kfree(area);
    return NULL;

The parameters is the size of the requested region which should be a multiple of the page size and the area flags, either VM_ALLOC or VM_IOREMAP

Allocate space for the vm_struct description struct

Pad the request so there is a page gap between areas. This is to help against overwrites

This is to ensure the size is not 0 after the padding

Start the search at the beginning of the vmalloc address space

Lock the list

Walk through the list searching for an area large enough for the request

Check to make sure the end of the addressable range has not been reached

If the requested area would fit between the current address and the next area, the search is complete

Make sure the address would not go over the end of the vmalloc address space

Copy in the area information

Link the new area into the list

Unlock the list and return
This label is reached if the request could not be satisfied
Unlock the list
203–204 Free the memory used for the area descriptor and return

**Function: vmalloc_area_pages** *(mm/vmalloc.c)*

This is the beginning of a standard page table walk function. This top level function will step through all PGDs within an address range. For each PGD, it will call pmd_alloc() to allocate a PMD directory and call alloc_area_pmd() for the directory.

```c
inline int vmalloc_area_pages (unsigned long address, unsigned long size, int gfp_mask, pgprot_t prot)
{
    pgd_t *dir;
    unsigned long end = address + size;
    int ret;
    dir = pgd_offset_k(address);
    spin_lock(&init_mm.page_table_lock);
    do {
        pmd_t *pmd;
        pmd = pmd_alloc(&init_mm, dir, address);
        ret = -ENOMEM;
        if (!pmd)
            break;
        ret = -ENOMEM;
        if (alloc_area_pmd(pmd, address, end - address, gfp_mask, prot))
            break;
        address = (address + PGDIR_SIZE) & PGDIR_MASK;
        dir++;
    } while (address && (address < end));
    spin_unlock(&init_mm.page_table_lock);
    flush_cache_all();
    return ret;
}
```

*address* is the starting address to allocate PMDs for. *size* is the size of the region, *gfp_mask* is the GFP flags for alloc_pages() and *prot* is the protection to give the PTE entry

The end address is the starting address plus the size
3.1. Allocating A Non-Contiguous Area

147 Get the PGD entry for the starting address
148 Lock the kernel page table
149-165 For every PGD within this address range, allocate a PMD directory and call alloc_area_pmd()
152 Allocate a PMD directory
158 Call alloc_area_pmd() which will allocate a PTE for each PTE slot in the PMD
161 address becomes the base address of the next PGD entry
162 Move dir to the next PGD entry
166 Release the lock to the kernel page table
167 flush_cache_all() will flush all CPU caches. This is necessary because the kernel page tables have changed
168 Return success

**Function: alloc_area_pmd** (mm/vmalloc.c)

This is the second stage of the standard page table walk to allocate PTE entries for an address range. For every PMD within a given address range on a PGD, pte_alloc() will creates a PTE directory and then alloc_area_pte() will be called to allocate the physical pages

```c
static inline int alloc_area_pmd(pmd_t * pmd, unsigned long address, unsigned long size, int gfp_mask, pgprot_t prot)
{
    unsigned long end;

    address &= ~PGDIR_MASK;
    end = address + size;
    if (end > PGDIR_SIZE)
        end = PGDIR_SIZE;
    do {
        pte_t * pte = pte_alloc(&init_mm, pmd, address);
        if (!pte)
            return -ENOMEM;
        if (alloc_area_pte(pte, address, end - address, gfp_mask, prot))
            return -ENOMEM;
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
    } while (address < end);
    return 0;
}
```
3.1. Allocating A Non-Contiguous Area

120 **address** is the starting address to allocate PMDs for. **size** is the size of the region, **gfp_mask** is the GFP flags for **alloc_pages()** and **prot** is the protection to give the PTE entry.

124 Align the starting address to the PGD

125-127 Calculate end to be the end of the allocation or the end of the PGD, whichever occurs first

128-136 For every PMD within the given address range, allocate a PTE directory and call **alloc_area_pte()**

129 Allocate the PTE directory

132 Call **alloc_area_pte()** which will allocate the physical pages

134 address becomes the base address of the next PMD entry

135 Move pmd to the next PMD entry

137 Return success

**Function: alloc_area_pte (mm/vmalloc.c)**

This is the last stage of the page table walk. For every PTE in the given PTE directory and address range, a page will be allocated and associated with the PTE.

95 static inline int alloc_area_pte (pte_t * pte, unsigned long address,
96 unsigned long size, int gfp_mask, pgprot_t prot)
97 {
98 unsigned long end;
99
100 address &= ~PMD_MASK;
101 end = address + size;
102 if (end > PMD_SIZE)
103 end = PMD_SIZE;
104 do {
105 struct page * page;
106 spin_unlock(&init_mm.page_table_lock);
107 page = alloc_page(gfp_mask);
108 spin_lock(&init_mm.page_table_lock);
109 if (!pte_none(*pte))
110 printk(KERN_ERR
111 "alloc_area_pte: page already exists\n");
112 if (!page)
113 return -ENOMEM;
114 set_pte(pte, mk_pte(page, prot));
115 address += PAGE_SIZE;
116 pte++;
117 } while (address < end);
3.2. Freeing A Non-Contiguous Area

100 Align the address to a PMD directory
101-103 The end address is the end of the request or the end of the directory, whichever occurs first
104-116 For every PTE in the range, allocate a physical page and set it to the PTE
106 Unlock the kernel page table before calling alloc_page(). alloc_page() may sleep and a spinlock must not be held
108 Re-acquire the page table lock
109-110 If the page already exists it means that areas must be overlapping somehow
112-113 Return failure if physical pages are not available
113 Assign the struct page to the PTE
114 address becomes the address of the next PTE
115 Move to the next PTE
117 Return success

3.2 Freeing A Non-Contiguous Area

vfree(void *addr)

Free a region of memory allocated with vmalloc, vmalloc_dma or vmalloc_32

Table 3.2: Non-Contiguous Memory Free API

**Function: vfree (mm/vmalloc.c)**

This is the top level function responsible for freeing a non-contiguous area of memory. It performs basic sanity checks before finding the vm_struct for the requested addr. Once found, it calls vmfree_area_pages()

207 void vfree(void * addr)
208 {
209     struct vm_struct **p, *tmp;
210     if (!addr)
211         return;
3.2. Freeing A Non-Contiguous Area

if ((PAGE_SIZE-1) & (unsigned long) addr) {
    printk(KERN_ERR
           "Trying to vfree() bad address (%p)\n", addr);
    return;
}
write_lock(&vmlist_lock);
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        *p = tmp->next;
        vmfree_area_pages(VMALLOC_VMADDR(tmp->addr),
                         tmp->size);
        write_unlock(&vmlist_lock);
        kfree(tmp);
        return;
    }
}
write_unlock(&vmlist_lock);
printk(KERN_ERR
        "Trying to vfree() nonexistent vm area (%p)\n", addr);
}

The parameter is the address returned by get_vm_area() returns for ioremaps and
vmalloc returns for allocations

211-213 Ignore NULL addresses
3.2. Freeing A Non-Contiguous Area

213-216 This checks the address is page aligned and is a reasonable quick guess to see if the area is valid or not

217 Acquire a write lock to the vmlist

218 Cycle through the vmlist looking for the correct vm_struct for addr

219 If this it the correct address then ...

220 Remove this area from the vmlist linked list

221 Free all pages associated with the address range

222 Release the vmlist lock

223 Free the memory used for the vm_struct and return

227-228 The vm_struct() was not found. Release the lock and print a message about the failed free

Function: vmfree_area_pages (mm/vmalloc.c)

This is the first stage of the page table walk to free all pages and PTEs associated with an address range. It is responsible for stepping through the relevant PGDs and for flushing the TLB.

80 void vmfree_area_pages(unsigned long address, unsigned long size)
81 {
82     pgd_t * dir;
83     unsigned long end = address + size;
84
85     dir = pgd_offset_k(address);
86     flush_cache_all();
87     do {
88         free_area_pmd(dir, address, end - address);
89         address = (address + PGDIR_SIZE) & PGDIR_MASK;
90         dir++;
91     } while (address && (address < end));
92     flush_tlb_all();
93 }

80 The parameters are the starting address and the size of the region
82 The address space end is the starting address plus its size
85 Get the first PGD for the address range
86 Flush the cache CPU so cache hits will not occur on pages that are to be deleted. This is a null operation on many architectures including the x86
3.2. Freeing A Non-Contiguous Area

Call \texttt{free_area_pmd()} to perform the second stage of the page table walk

\texttt{address} becomes the starting address of the next PGD

Move to the next PGD

Flush the TLB as the page tables have now changed

\textbf{Function: free_area_pmd (mm/vmalloc.c)}

This is the second stage of the page table walk. For every PMD in this directory, call free_area_pte to free up the pages and PTEs.

\begin{verbatim}
static inline void free_area_pmd(pgd_t * dir, unsigned long address,
                                unsigned long size)
{
    pmd_t * pmd;
    unsigned long end;
    if (pgd_none(*dir))
        return;
    if (pgd_bad(*dir)) {
        pgd_ERROR(*dir);
        pgd_clear(dir);
        return;
    }
    pmd = pmd_offset(dir, address);
    address &= ~PGDIR_MASK;
    end = address + size;
    if (end > PGDIR_SIZE)
        end = PGDIR_SIZE;
    do {
        free_area_pte(pmd, address, end - address);
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
    } while (address < end);
}
\end{verbatim}

The parameters are the PGD been stepped through, the starting address and the length of the region

61-62 If there is no PGD, return. This can occur after vfree is called during a failed allocation

63-67 A PGD can be bad if the entry is not present, it is marked read-only or it is marked accessed or dirty

68 Get the first PMD for the address range

69 Make the address PGD aligned
3.2. Freeing A Non-Contiguous Area

70-72 end is either the end of the space to free or the end of this PGD, whichever is first

73-77 For every PMD, call free_area_pte() to free the PTE entries

75 address is the base address of the next PMD

76 Move to the next PMD

Function: free_area_pte (mm/vmalloc.c)

This is the final stage of the page table walk. For every PTE in the given PMD within the address range, it will free the PTE and the associated page

22 static inline void free_area_pte(pmd_t * pmd, unsigned long address, unsigned long size)

23 {
24     pte_t * pte;
25     unsigned long end;
26
27     if (pmd_none(*pmd))
28         return;
29     if (pmd_bad(*pmd)) {
30         pmd_ERROR(*pmd);
31         pmd_clear(pmd);
32         return;
33     }
34     pte = pte_offset(pmd, address);
35     address &= ~PMD_MASK;
36     end = address + size;
37     if (end > PMD_SIZE)
38         end = PMD_SIZE;
39     do {
40         pte_t page;
41         page = ptep_get_and_clear(pte);
42         address += PAGE_SIZE;
43         pte++;
44         if (pte_none(page))
45             continue;
46         if (pte_present(page)) {
47             struct page *ptpage = pte_page(page);
48             if (VALID_PAGE(ptpage) &&
49                 (!PageReserved(ptpage)))
50                 __free_page(ptpage);
51             continue;
52         }
53         printk(KERN_CRIT
54                     "Whee.. Swapped out page in kernel page table\n");
55     } while (address < end);
56 }
3.2. Freeing A Non-Contiguous Area

The parameters are the PMD that PTEs are been freed from, the starting address and the size of the region to free.

The PMD could be absent if this region is from a failed `vmalloc()`.

A PMD can be bad if it’s not in main memory, it’s read only or it’s marked dirty or accessed.

`pte` is the first PTE in the address range.

Align the address to the PMD.

The end is either the end of the requested region or the end of the PMD, whichever occurs first.

Step through all PTEs, perform checks and free the PTE with its associated page.

`pte_get_and_clear()` will remove a PTE from a page table and return it to the caller.

`address` will be the base address of the next PTE.

Move to the next PTE.

If there was no PTE, simply continue.

If the page is present, perform basic checks and then free it.

`pte_page()` uses the global `mem_map` to find the `struct page` for the PTE.

Make sure the page is a valid page and it is not reserved before calling `__free_page()` to free the physical page.

Continue to the next PTE.

If this line is reached, a PTE within the kernel address space was somehow swapped out. Kernel memory is not swappable and so is a critical error.
Chapter 4

Slab Allocator

4.0.1 Cache Creation

This section covers the creation of a cache. The tasks that are taken to create a cache are

- Perform basic sanity checks for bad usage
- Perform debugging checks if CONFIG_SLAB_DEBUG is set
- Allocate a kmem_cache_t from the cache_cache slab cache
- Align the object size to the word size
- Calculate how many objects will fit on a slab
- Align the slab size to the hardware cache
- Calculate colour offsets
- Initialise remaining fields in cache descriptor
- Add the new cache to the cache chain

See Figure 4.1 to see the call graph relevant to the creation of a cache. The depth of it is shallow as the depths will be discussed in other sections.

Function: kmem_cache_create (mm/slab.c)

Because of the size of this function, it will be dealt with in chunks. Each chunk is one of the items described in the previous section.

621 kmem_cache_t *
622 kmem_cache_create (const char *name, size_t size,
623 size_t offset, unsigned long flags,
624 void (*ctor)(void*, kmem_cache_t *, unsigned long),
625 void (*dtor)(void*, kmem_cache_t *, unsigned long))
626 {
627     const char *func_nm = KERN_ERR "kmem_create: ";
628     size_t left_over, align, slab_size;
4.0.1. Cache Creation

Caches are created and managed using the `kmem_cache_create` function:

```c
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))
```

This function creates a new cache and adds it to the cache chain.

- `kmem_cache_reap(int gfp_mask)`
  - Scans at most `REAP_SCANLEN` caches and selects one for reaping all per-cpu objects and free slabs from. Called when memory is tight.

- `kmem_cache_shrink(kmem_cache_t *cachep)`
  - This function will delete all per-cpu objects associated with a cache and delete all slabs in the `slabs_free` list. It returns the number of pages freed.

- `kmem_cache_alloc(kmem_cache_t *cachep, int flags)`
  - Allocate a single object from the cache and return it to the caller.

- `kmem_cache_free(kmem_cache_t *cachep, void *objp)`
  - Free an object and return it to the cache.

- `kmalloc(size_t size, int flags)`
  - Allocate a block of memory from one of the sizes cache.

- `kfree(const void *objp)`
  - Free a block of memory allocated with kmalloc.

- `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 4.1: Slab Allocator API for caches

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmem_cache_create</td>
<td>Creates a new cache and adds it to the cache chain</td>
</tr>
<tr>
<td>kmem_cache_reap</td>
<td>Scans at most <code>REAP_SCANLEN</code> caches and selects one for reaping</td>
</tr>
<tr>
<td>kmem_cache_shrink</td>
<td>This function will delete all per-cpu objects associated with a cache</td>
</tr>
<tr>
<td>kmem_cache_alloc</td>
<td>Allocate a single object from the cache and return it to the caller</td>
</tr>
<tr>
<td>kmem_cache_free</td>
<td>Free an object and return it to the cache</td>
</tr>
<tr>
<td>kmalloc</td>
<td>Allocate a block of memory from one of the sizes cache</td>
</tr>
<tr>
<td>kfree</td>
<td>Free a block of memory allocated with kmalloc</td>
</tr>
<tr>
<td>kmem_cache_destroy</td>
<td>Destroys all objects in all slabs and frees up all associated memory</td>
</tr>
</tbody>
</table>

Figure 4.1: Call Graph: `kmem_cache_create()`
4.0.1. Cache Creation

Perform basic sanity checks for bad usage

The parameters of the function are

- **name** The human readable name of the cache
- **size** The size of an object
- **offset** This is used to specify a specific alignment for objects in the cache but it usually left as 0
- **flags** Static cache flags
- **ctor** A constructor function to call for each object during slab creation
- **dtor** The corresponding destructor function. It is expected the destructor function leaves an object in an initialised state

These are all serious usage bugs that prevent the cache even attempting to create

If the human readable name is greater than the maximum size for a cache name (CACHE_NAMELEN)

An interrupt handler cannot create a cache as access to spinlocks and semaphores is needed

The object size must be at least a word in size. Slab is not suitable for objects that are measured in bits

The largest possible slab that can be created is $2^{MAX\_OBJ\_ORDER}$ number of pages which provides 32 pages.

A destructor cannot be used if no constructor is available

The offset cannot be before the slab or beyond the boundary of the first page

Call **BUG()** to exit
4.0.1. Cache Creation

642 #if DEBUG
643 if ((flags & SLAB_DEBUG_INITIAL) && !ctor) {
644 printk("%sNo con, but init state check
 requested - %s\n", func_nm, name);
645 flags &= ~SLAB_DEBUG_INITIAL;
646 }
647
648 if ((flags & SLAB_POISON) && ctor) {
649 printk("%sPoisoning requested, but con given - %s\n",
 func_nm, name);
650 flags &= ~SLAB_POISON;
651 }
652 #if FORCED_DEBUG
653 if ((size < (PAGE_SIZE>>3)) && !(flags & SLAB_MUST_HWCACHE_ALIGN))
654 flags |= SLAB_RED_ZONE;
655 if (!ctor)
656 flags |= SLAB_POISON;
657 #endif
658 #endif
659 BUG_ON(flags & ~CREATE_MASK);

This block performs debugging checks if CONFIG_SLAB_DEBUG is set

643-646 The flag SLAB_DEBUG_INITIAL requests that the constructor check the objects to
make sure they are in an initialised state. For this, a constructor must obviously exist.
If it doesn’t, the flag is cleared

649-653 A slab can be poisoned with a known pattern to make sure an object wasn’t used
before it was allocated but a constructor would ruin this pattern falsely reporting a
bug. If a constructor exists, remove the SLAB_POISON flag if set

655-660 Only small objects will be red zoned for debugging. Red zoning large objects
would cause severe fragmentation

661-662 If there is no constructor, set the poison bit

670 The CREATE_MASK is set with all the allowable flags kmem_cache_create() can be
called with. This prevents callers using debugging flags when they are not available
and BUG()s it instead

673 cachep =
   (kmem_cache_t *) kmem_cache_alloc(&cache_cache,
      SLAB_KERNEL);
674 if (!cachep)
675 goto opps;
676 memset(cachep, 0, sizeof(kmem_cache_t));

Allocate a kmem_cache_t from the cache_cache slab cache.
Allocate a cache descriptor object from the cache_cache(see Section 4.2.2)

If out of memory goto opps which handles the oom situation

Zero fill the object to prevent surprises with uninitialised data

If the size is not aligned to the size of a word then...

Increase the object by the size of a word

Mask out the lower bits, this will effectively round the object size up to the next word boundary

Print out an informational message for debugging purposes

If debugging is enabled then the alignments have to change slightly

Don’t bother trying to align things to the hardware cache. The red zoning of the object is going to offset it by moving the object one word away from the cache boundary
The size of the object increases by two \texttt{BYTES\_PER\_WORD} to store the red zone mark at either end of the object.

Align the object on a word size.

If requested, align the objects to the L1 CPU cache.

If the objects are large, store the slab descriptors off-slab. This will allow better packing of objects into the slab.

If hardware cache alignment is requested, the size of the objects must be adjusted to align themselves to the hardware cache.

This is important to arches (e.g. Alpha or Pentium 4) with large L1 cache bytes. \texttt{align} will be adjusted to be the smallest that will give hardware cache alignment. For machines with large L1 cache lines, two or more small objects may fit into each line. For example, two objects from the size-32 cache will fit on one cache line from a Pentium 4.

Round the cache size up to the hardware cache alignment.


do {
    unsigned int break_flag = 0;
    cal_wastage:
    kmem_cache_estimate(cachep->gfporder,
        size, flags,
        &left_over,
        &cachep->num);
    if (break_flag)
        break;
    if (cachep->gfporder >= MAX_GFP_ORDER)
        break;
    if (!cachep->num)
        goto next;
    if (flags & CFLGS\_OFF\_SLAB &\&
        cachep->num > offslab_limit) {
        cachep->gfporder--;
        break_flag++;
        goto cal_wastage;
    }
    if (cachep->gfporder >= slab_break_gfp_order)
        break;
    if ((left_over*8) <= (PAGE\_SIZE<<cachep->gfporder))
        break;
    next:
    cachep->gfporder++;
4.0.1.Cache Creation

753 } while (1);
754
755 if (!cachep->num) {
756 printk("kmem_cache_create: couldn’t
    create cache %s.
", name);
757 kmem_cache_free(&cache_cache, cachep);
758 cachep = NULL;
759 goto opps;
760 }

Calculate how many objects will fit on a slab and adjust the slab size as necessary

727-728 kmem_cache_estimate() (see Section 4.0.2) calculates the number of objects that can fit on a slab at the current gfp order and what the amount of leftover bytes will be

729-730 The break_flag is set if the number of objects fitting on the slab exceeds the number that can be kept when offslab slab descriptors are used

731-732 The order number of pages used must not exceed MAX_GFP_ORDER (5)

733-734 If even one object didn’t fill, goto next: which will increase the gfporder used for the cache

735 If the slab descriptor is kept off-cache but the number of objects exceeds the number that can be tracked with bufctl’s off-slab then ....

737 Reduce the order number of pages used

738 Set the break_flag so the loop will exit

739 Calculate the new wastage figures

746-747 The slab_break_gfp_order is the order to not exceed unless 0 objects fit on the slab. This check ensures the order is not exceeded

749-759 This is a rough check for internal fragmentation. If the wastage as a fraction of the total size of the cache is less than one eight, it is acceptable

752 If the fragmentation is too high, increase the gfp order and recalculate the number of objects that can be stored and the wastage

755 If after adjustments, objects still do not fit in the cache, it cannot be created

757-758 Free the cache descriptor and set the pointer to NULL

758 Goto opps which simply returns the NULL pointer
4.0.1. Cache Creation

```c
slab_size = L1_CACHE_ALIGN(
    cachep->num*sizeof(kmem_bufctl_t) +
    sizeof(slab_t));
```

Align the slab size to the hardware cache

`slab_size` is the total size of the slab descriptor *not* the size of the slab itself. It is the size `slab_t` struct and the number of objects * size of the bufctl

767-769 If there is enough leftover space for the slab descriptor and it was specified to place the descriptor off-slab, remove the flag and update the amount of `left_over` bytes there is. This will impact the cache colouring but with the large objects associated with off-slab descriptors, this is not a problem

```c
if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
    flags &= ~CFLGS_OFF_SLAB;
    left_over -= slab_size;
}
```

Calculate colour offsets.

`offset` is the offset within the page the caller requested. This will make sure the offset requested is at the correct alignment for cache usage

775-776 If somehow the offset is 0, then set it to be aligned for the CPU cache

777 This is the offset to use to keep objects on different cache lines. Each slab created will be given a different colour offset

778 This is the number of different offsets that can be used

```c
offset += (align-1);
offset &= ~(align-1);
if (!offset)
    offset = L1_CACHE_BYTES;
cachep->colour_off = offset;
cachep->colour = left_over/offset;
```

```c
if (!cachep->gfporder && !(flags & CFLGS_OFF_SLAB))
    flags |= CFLGS_OPTIMIZE;
cachep->flags = flags;
cachep->gfpflags = 0;
if (flags & SLAB_CACHE_DMA)
    cachep->gfpflags |= GFP_DMA;
spin_lock_init(&cachep->spinlock);
cachep->objsize = size;
INIT_LIST_HEAD(&cachep->slabs_full);
```
4.0.1. Cache Creation

791 INIT_LIST_HEAD(&cachep->slabs_partial);
792 INIT_LIST_HEAD(&cachep->slabs_free);
793
794 if (flags & CFLGS_OFF_SLAB)
795     cachep->slabp_cache = kmem_find_general_cachep(slab_size,0);
796 cachep->ctor = ctor;
797 cachep->dtor = dtor;
799 strcpy(cachep->name, name);

801 #ifdef CONFIG_SMP
802     if (g_cpucache_up)
803         enable_cpucache(cachep);
804 #endif

Initialise remaining fields in cache descriptor

781-782 For caches with slabs of only 1 page, the CFLGS_OPTIMISE flag is set. In reality it makes no difference as the flag is unused

784 Set the cache static flags

785 Zero out the gfpflags. Defunct operation as memset after the cache descriptor was allocated would do this

786-787 If the slab is for DMA use, set the GFP_DMA flag so the buddy allocator will use ZONE_DMA

788 Initialise the spinlock for access the cache

789 Copy in the object size, which now takes hardware cache alignment if necessary

790-792 Initialise the slab lists

794-795 If the descriptor is kept off-slab, allocate a slab manager and place it for use in slabp_cache. See Section 4.1.1

796-797 Set the pointers to the constructor and destructor functions

799 Copy in the human readable name

802-803 If per-cpu caches are enabled, create a set for this cache. See Section 4.4

806     down(&cache_chain_sem);
807     {
808         struct list_head *p;
809         list_for_each(p, &cache_chain) {
811         kmem_cache_t *pc = list_entry(p,
4.0.2 Calculating the Number of Objects on a Slab

Function: kmem_cache_estimate (mm/slab.c)

During cache creation, it is determined how many objects can be stored in a slab and
how much waste-age there will be. The following function calculates how many objects may
be stored, taking into account if the slab and bufctl's must be stored on-slab.

```c
static void kmem_cache_estimate (unsigned long gfporder, size_t size,
   int flags, size_t *left_over, unsigned int *num)
{
   int i;
   size_t wastage = PAGE_SIZE<<gfporder;
   size_t extra = 0;
   size_t base = 0;
```
4.0.2. Calculating the Number of Objects on a Slab

```c
if (!(flags & CFLGS_OFF_SLAB)) {
    base = sizeof(slab_t);
    extra = sizeof(kmem_bufctl_t);
}

i = 0;
while (i*size + L1_CACHE_ALIGN(base+i*extra) <= wastage)
    i++;
if (i > 0)
    i--;

if (i > SLAB_LIMIT)
    i = SLAB_LIMIT;

*num = i;
wastage -= i*size;
wastage -= L1_CACHE_ALIGN(base+i*extra);
*left_over = wastage;
```

The parameters of the function are as follows:

- **gfporder** The \(2^{\text{gfporder}}\) number of pages to allocate for each slab
- **size** The size of each object
- **flags** The cache flags
- **left_over** The number of bytes left over in the slab. Returned to caller
- **num** The number of objects that will fit in a slab. Returned to caller

\(\text{wastage}\) is decremented through the function. It starts with the maximum possible amount of wast-age.

\(\text{extra}\) is the number of bytes needed to store \text{kmem_bufctl_t}

\(\text{base}\) is where usable memory in the slab starts

If the slab descriptor is kept on cache, the base begins at the end of the \text{slab_t} struct and the number of bytes needed to store the bufctl is the size of \text{kmem_bufctl_t}

\(i\) becomes the number of objects the slab can hold

401-402 This counts up the number of objects that the cache can store. \(i*\text{size}\) is the amount of memory needed to store the object itself. \(L1\_CACHE\_ALIGN(base+i*extra)\) is slightly trickier. This is calculating the amount of memory needed to store the \text{kmem_bufctl_t} of which one exists for every object in the slab. As it is at the beginning of the slab, it is L1 cache aligned so that the first object in the slab will be aligned to hardware cache. \(i*\text{extra}\) will calculate the amount of space needed to hold a \text{kmem_bufctl_t} for this object. As wast-age starts out as the size of the slab, its use is overloaded here.
4.0.3 Cache Shrinking

Because the previous loop counts until the slab overflows, the number of objects that can be stored is \( i-1 \).

SLAB_LIMIT is the absolute largest number of objects a slab can store. It is defined as 0xffffffff as this the largest number `kmem_bufctl_t()`, which is an unsigned int, can hold.

`num` is now the number of objects a slab can hold.

Take away the space taken up by all the objects from wastage.

Take away the space taken up by the `kmem_bufctl_t`.

Wastage has now been calculated as the left over space in the slab.

4.0.3 Cache Shrinking

Two varieties of shrink functions are provided. `kmem_cache_shrink()` removes all slabs from `slabs_free` and returns the number of pages freed as a result. `__kmem_cache_shrink()` frees all slabs from `slabs_free` and then verifies that `slabs_partial` and `slabs_full` are empty. This is important during cache destruction when it doesn’t matter how many pages are freed, just that the cache is empty.

**Function: kmem_cache_shrink (mm/slab.c)**

This function performs basic debugging checks and then acquires the cache descriptor lock before freeing slabs. At one time, it also used to call `drain_cpu_caches()` to free up objects on the per-cpu cache. It is curious that this was removed as it is possible slabs could not be freed due to an object been allocation on a per-cpu cache but not in use.

```c
int kmem_cache_shrink(kmem_cache_t *cachep)
{
    int ret;
```
4.0.3. Cache Shrinking

```c
if (!cachep || in_interrupt() || !is_chained_kmem_cache(cachep))
    BUG();

spin_lock_irq(&cachep->spinlock);
ret = __kmem_cache_shrink_locked(cachep);
spin_unlock_irq(&cachep->spinlock);
return ret << cachep->gfporder;
```

The parameter is the cache been shrunk

- The cache pointer is not null
- That an interrupt isn’t trying to do this
- That the cache is on the cache chain and not a bad pointer

Acquire the cache descriptor lock and disable interrupts

Shrink the cache

Release the cache lock and enable interrupts

This returns the number of pages freed but does not take into account the objects freed by draining the CPU.

**Function: __kmem_cache_shrink** *(mm/slab.c)*

This function is identical to kmem_cache_shrink() except it returns if the cache is empty or not. This is important during cache destruction when it is not important how much memory was freed, just that it is safe to delete the cache and not leak memory.

```c
static int __kmem_cache_shrink(kmem_cache_t *cachep)
{
    int ret;
    drain_cpu_caches(cachep);
    spin_lock_irq(&cachep->spinlock);
    __kmem_cache_shrink_locked(cachep);
    ret = !list_empty(&cachep->slabs_full) ||
        !list_empty(&cachep->slabs_partial);
    spin_unlock_irq(&cachep->spinlock);
    return ret;
}
```

Remove all objects from the per-CPU objects cache
4.0.3. Cache Shrinking

951 Acquire the cache descriptor lock and disable interrupts
952 Free all slabs in the slabs_free list
954-954 Check the slabs Partial and slabs Full lists are empty
955 Release the cache descriptor lock and re-enable interrupts
956 Return if the cache has all its slabs free or not

Function: __kmem_cache_shrink_locked (mm/slab.c)
This does the dirty work of freeing slabs. It will keep destroying them until the growing flag gets set, indicating the cache is in use or until there is no more slabs in slabs free.

917 static int __kmem_cache_shrink_locked(kmem_cache_t *cachep)
918 {
919     slab_t *slabp;
920     int ret = 0;
921
923     while (!cachep->growing) {
924         struct list_head *p;
925
926         p = cachep->slabs_free.prev;
927         if (p == &cachep->slabs_free)
928             break;
929
930         slabp = list_entry(cachep->slabs_free.prev, slab_t, list);
931         #if DEBUG
932             if (slabp->inuse)
933                 BUG();
934         #endif
935         list_del(&slabp->list);
936
937         spin_unlock_irq(&cachep->spinlock);
938         kmem_slab_destroy(cachep, slabp);
939         ret++;
940         spin_lock_irq(&cachep->spinlock);
941     }
942     return ret;
943 }

923 While the cache is not growing, free slabs
926-930 Get the last slab on the slabs_free list
932-933 If debugging is available, make sure it is not in use. If it is not in use, it should not be on the slabs_free list in the first place
935 Remove the slab from the list
When a module is unloaded, it is responsible for destroying any cache it has created as during module loading, it is ensured there is not two caches of the same name. Core kernel code often does not destroy its caches as their existence persists for the life of the system. The steps taken to destroy a cache are

- Delete the cache from the cache chain
- Shrink the cache to delete all slabs (see Section 4.0.3)
- Free any per CPU caches (kfree())
- Delete the cache descriptor from the cache_cache (see Section: 4.2.3)

Figure 4.3 Shows the call graph for this task.

![Call Graph: kmem_cache_destroy()](image)

Function: kmem_cache_destroy (mm/slab.c)

```c
995 int kmem_cache_destroy (kmem_cache_t * cachep)
996 {
997     if (!cachep || in_interrupt() || cachep->growing)
998         BUG();
999
1000     /* Find the cache in the chain of caches. */
1001     down(&cache_chain_sem);
1002     /* the chain is never empty, cache_cache is never destroyed */
1003     if (clock_searchp == cachep)
1004         clock_searchp = list_entry(cachep->next.next,
1005                                        kmem_cache_t, next);
```
Sanity check. Make sure the cachep is not null, that an interrupt isn't trying to
do this and that the cache hasn't been marked growing, indicating it is in use

1001 Acquire the semaphore for accessing the cache chain

1003-1005 Acquire the list entry from the cache chain

1006 Delete this cache from the cache chain

1007 Release the cache chain semaphore

1009 Shrink the cache to free all slabs (see Section 4.0.3)

1010-1015 The shrink function returns true if there is still slabs in the cache. If there is,
the cache cannot be destroyed so it is added back into the cache chain and the error
reported

1020-1021 If SMP is enabled, the per-cpu data structures are deleted with kfree kfree()

1024 Delete the cache descriptor from the cache_cache
4.0.5 Cache Reaping

When the page allocator notices that memory is getting tight, it wakes kswapd to begin freeing up pages (see Section 2.1). One of the first ways it accomplishes this task is telling the slab allocator to reap caches. It has to be the slab allocator that selects the caches as other subsystems should not know anything about the cache internals.

![Call Graph: kmem_cache_reap()](image)

The call graph in Figure 4.4 is deceptively simple. The task of selecting the proper cache to reap is quite long. In case there is many caches in the system, only REAP_SCANLEN caches are examined in each call. The last cache to be scanned is stored in the variable clock_searchp so as not to examine the same caches over and over again. For each scanned cache, the reaper does the following

- Check flags for SLAB_NO_REAP and skip if set
- If the cache is growing, skip it
- if the cache has grown recently (DFLGS_GROWN is set in dflags), skip it but clear the flag so it will be reaped the next time
- Count the number of free slabs in slabs_free and calculate how many pages that would free in the variable pages
- If the cache has constructors or large slabs, adjust pages to make it less likely for the cache to be selected.
- If the number of pages that would be freed exceeds REAP_PERFECT, free half of the slabs in slabs_free
- Otherwise scan the rest of the caches and select the one that would free the most pages for freeing half of its slabs in slabs_free
Function: kmem_cache_reap (mm/slab.c)

Because of the size of this function, it will be broken up into three separate sections. The first is simple function preamble. The second is the selection of a cache to reap and the third is the freeing of the slabs.

```c
1736 int kmem_cache_reap (int gfp_mask)
1737 {
1738     slab_t *slabp;
1739     kmem_cache_t *searchp;
1740     kmem_cache_t *best_cachep;
1741     unsigned int best_pages;
1742     unsigned int best_len;
1743     unsigned int scan;
1744     int ret = 0;
1745     
1746     if (gfp_mask & __GFP_WAIT)
1747         down(&cache_chain_sem);
1748     else
1749         if (down_trylock(&cache_chain_sem))
1750             return 0;
1751     
1752     scan = REAP_SCANLEN;
1753     best_len = 0;
1754     best_pages = 0;
1755     best_cachep = NULL;
1756     searchp = clock_searchp;
```

1736 The only parameter is the GFP flag. The only check made is against the __GFP_WAIT flag. As the only caller, kswapd, can sleep, this parameter is virtually worthless.

1746-1747 Can the caller sleep? If yes, then acquire the semaphore.

1749-1750 Else, try and acquire the semaphore and if not available, return.

1752 REAP_SCANLEN (10) is the number of caches to examine.

1756 Set searchp to be the last cache that was examined at the last reap.

```c
1757     do {
1758         unsigned int pages;
1759         struct list_head* p;
1760         unsigned int full_free;
1761         
1763         if (searchp->flags & SLAB_NO_REAP)
1764             goto next;
1765         spin_lock_irq(&searchp->spinlock);
1766         if (searchp->growing)
1767             goto next_unlock;
```
4.0.5. Cache Reaping

1768     if (searchp->dflags & DFLGS_GROWN) {
1769         searchp->dflags &= ~DFLGS_GROWN;
1770         goto next_unlock;
1771     }
1772 #ifdef CONFIG_SMP
1773     {
1774         cpucache_t *cc = cc_data(searchp);
1775         if (cc && cc->avail) {
1776             __free_block(searchp, cc_entry(cc),
                            cc->avail);
1777             cc->avail = 0;
1778         }
1779     }
1780 #endif
1781     full_free = 0;
1782     p = searchp->slabs_free.next;
1783     while (p != &searchp->slabs_free) {
1784         slabp = list_entry(p, slab_t, list);
1785         #if DEBUG
1786             if (slabp->inuse)
1787                 BUG();
1788         #endif
1789         full_free++;
1790         p = p->next;
1791     }
1792     }
1793     pages = full_free * (1<<searchp->gfporder);
1794     if (searchp->ctor)
1795         pages = (pages*4+1)/5;
1796     if (searchp->gfporder)
1797         pages = (pages*4+1)/5;
1798     if (pages > best_pages) {
1799         best_cachep = searchp;
1800         best_len = full_free;
1801         best_pages = pages;
1802         if (pages >= REAP_PERFECT) {
1803             clock_searchp =
1804                 list_entry(searchp->next.next,
1805                     kmem_cache_t,next);
1806             goto perfect;
1807         }
1808     }
1809     next_unlock:
1810     spinUnlockIrq(&searchp->spinlock);
next:
    searchp =
    list_entry(searchp->next.next,kmem_cache_t,next);
} while (--scan && searchp != clock_searchp);

This block examines REAP_SCANLEN number of caches to select one to free

Acquire an interrupt safe lock to the cache descriptor

If the cache is growing, skip it

If the cache has grown recently, skip it and clear the flag

Free any per CPU objects to the global pool

Count the number of slabs in the slabs_free list

Calculate the number of pages all the slabs hold

If the objects have constructors, reduce the page count by one fifth to make it less likely to be selected for reaping

If the slabs consist of more than one page, reduce the page count by one fifth. This is because high order pages are hard to acquire

If this is the best candidate found for reaping so far, check if it is perfect for reaping

Record the new maximums

best_len is recorded so that it is easy to know how many slabs is half of the slabs in the free list

If this cache is perfect for reaping then ....

Update clock_searchp

Goto perfect where half the slabs will be freed

This label is reached if it was found the cache was growing after acquiring the lock

Release the cache descriptor lock

Move to the next entry in the cache chain

Scan while REAP_SCANLEN has not been reached and we have not cycled around the whole cache chain

clock_searchp = searchp;
if (!best_cachep)
    goto out;
4.0.5. Cache Reaping

    spin_lock_irq(&best_cachep->spinlock);
    perfect:
    /* free only 50% of the free slabs */
    best_len = (best_len + 1)/2;
    for (scan = 0; scan < best_len; scan++) {
        struct list_head *p;
        if (best_cachep->growing)
            break;
        p = best_cachep->slabs_free.prev;
        if (p == &best_cachep->slabs_free)
            break;
        slabp = list_entry(p,slab_t,list);
        #if DEBUG
        if (slabp->inuse)
            BUG();
        #endif
        list_del(&slabp->list);
        STATS_INC_REAPED(best_cachep);
        /* Safe to drop the lock. The slab is no longer
         * linked to the
         * cache.
         */
        spin_unlock_irq(&best_cachep->spinlock);
        kmem_slab_destroy(best_cachep, slabp);
        spin_lock_irq(&best_cachep->spinlock);
    }
    spin_unlock_irq(&best_cachep->spinlock);
    ret = scan * (1 << best_cachep->gfporder);
    out:
    up(&cache_chain_sem);
    return ret;
}

This block will free half of the slabs from the selected cache

1820 Update clock_searchp for the next cache reap

1822-1824 If a cache was not found, goto out to free the cache chain and exit

1826 Acquire the cache chain spinlock and disable interrupts. The cachep descriptor has to be held by an interrupt safe lock as some caches may be used from interrupt context. The slab allocator has no way to differentiate between interrupt safe and unsafe caches

1829 Adjust best_len to be the number of slabs to free
4.1 Slabs

This section will describe how a slab is structured and managed. The struct which describes it is much simpler than the cache descriptor, but how the slab is arranged is slightly more complex. We begin with the descriptor.

```c
typedef struct slab_s {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem;
    unsigned int inuse;
    kmem_bufctl_t free;
} slab_t;
```

- `list` The list the slab belongs to. One of `slab_full`, `slab_partial` and `slab_free`
- `colouroff` The colour offset is the offset of the first object within the slab. The address of the first object is `s_mem + colouroff`. See Section 4.1.1
- `s_mem` The starting address of the first object within the slab
- `inuse` Number of active objects in the slab
- `free` This is an array of `bufctl`s used for storing locations of free objects. See the companion document for seeing how to track free objects.
4.1.1 Storing the Slab Descriptor

**Function:** `kmem_cache_slabmgmt` *(mm/slab.c)*

This function will either allocate space to keep the slab descriptor off cache or reserve enough space at the beginning of the slab for the descriptor and the bufctl’s.

```c
1030 static inline slab_t * kmem_cache_slabmgmt (kmem_cache_t *cachep,
1031             void *objp,
1032             int colour_off,
1033             int local_flags)
1034 {
1035     slab_t *slabp;
1036
1037     if (OFF_SLAB(cachep)) {
1038         slabp = kmem_cache_alloc(cachep->slabp_cache,
1039                                  local_flags);
1040     }
1041     else {
1042         slabp = objp+colour_off;
1043         colour_off += L1_CACHE_ALIGN(cachep->num *
1044                                    sizeof(kmem_bufctl_t) +
1045                                    sizeof(slab_t));
1046     }
1047     slabp->inuse = 0;
1048     slabp->colouroff = colour_off;
1049     slabp->s_mem = objp+colour_off;
1050     return slabp;
1051 }
```

The parameters of the function are:

- `cachep` The cache the slab is to be allocated to
- `objp` When the function is called, this points to the beginning of the slab
- `colour_off` The colour offset for this slab
- `local_flags` These are the flags for the cache. They are described in the companion document

1035–1040 If the slab descriptor is kept off cache....

1037 Allocate memory from the sizes cache. During cache creation, `slabp_cache` is set to the appropriate size cache to allocate from. See Section 4.0.1

1038 If the allocation failed, return
4.1.1. Storing the Slab Descriptor

Reserve space at the beginning of the slab.

The address of the slab will be the beginning of the slab \((\text{objp})\) plus the colour offset \(\text{colour\_off}\) is calculated to be the offset where the first object will be placed. The address is L1 cache aligned. \(\text{cachep->num} \times \text{sizeof(kmem\_bufctl\_t)}\) is the amount of space needed to hold the bufctls for each object in the slab and \(\text{sizeof(slab\_t)}\) is the size of the slab descriptor. This effectively has reserved the space at the beginning of the slab.

The number of objects in use on the slab is 0.

The colouroff is updated for placement of the new object.

The address of the first object is calculated as the address of the beginning of the slab plus the offset.

**Function: kmem\_find\_general\_cachep** \((\text{mm/slab.c})\)

If the slab descriptor is to be kept off-slab, this function, called during cache creation (see Section 4.0.1) will find the appropriate sizes cache to use and will be stored within the cache descriptor in the field \(\text{slabp\_cache}\).

```c
1618 kmem\_cache\_t * kmem\_find\_general\_cachep (size\_t size, int gfpflags)
1619 {
1620     cache\_sizes\_t *csizep = cache\_sizes;
1621     for ( ; csizep->cs\_size; csizep++) {
1622         if (size > csizep->cs\_size)
1623             continue;
1624         break;
1625     }
1626     return (gfpflags & GFP\_DMA) ? csizep->cs\_dmacachep : csizep->cs\_ cachep;
1627 }
```

\(\text{size}\) is the size of the slab descriptor. \(\text{gfpflags}\) is always 0 as DMA memory is not needed for a slab descriptor.

Starting with the smallest size, keep increasing the size until a cache is found with buffers large enough to store the slab descriptor.

Return either a normal or DMA sized cache depending on the \(\text{gfpflags}\) passed in. In reality, only the \(\text{cs\_cachep}\) is ever passed back.
4.1.2 Slab Structure

4.1.3 Slab Creation

This section will show how a cache is grown when no objects are left in the slabs_partial list and there is no slabs in slabs_free. The principle function for this is \texttt{kmem_cache\_grow()}. The tasks it fulfills are

- Perform basic sanity checks to guard against bad usage
- Calculate colour offset for objects in this slab
- Allocate memory for slab and acquire a slab descriptor
- Link the pages used for the slab to the slab and cache descriptors (see Section 4.1)
- Initialise objects in the slab
- Add the slab to the cache

**Function: kmem_cache\_grow** (*mm/slab.c*)

```c
static int kmem_cache\_grow (kmem_cache_t * cachep, int flags)
{
    slab_t *slabp;
    struct page *page;
    void *objp;
    size_t offset;
```

Figure 4.5: Call Graph: \texttt{kmem_cache\_grow()}

Figure 4.5 shows the call graph to grow a cache. This function will be dealt with in blocks. Each block corresponds to one of the tasks described in the previous section.
4.1.3. Slab Creation

```c
unsigned int i, local_flags;
unsigned long ctor_flags;
unsigned long save_flags;
```

Basic declarations. The parameters of the function are

cachep The cache to allocate a new slab to
flags The flags for a slab creation

```c
if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW))
  BUG();
if (flags & SLAB_NO_GROW)
  return 0;
```

```c
if (in_interrupt() && (flags & SLAB_LEVEL_MASK) != SLAB_ATOMIC)
  BUG();
ctor_flags = SLAB_CTOR_CONSTRUCTOR;
local_flags = (flags & SLAB_LEVEL_MASK);
if (local_flags == SLAB_ATOMIC)
  ctor_flags |= SLAB_CTOR_ATOMIC;
```

Perform basic sanity checks to guard against bad usage. The checks are made here rather than `kmem_cache_alloc()` to protect the critical path. There is no point checking the flags every time an object needs to be allocated.

1116-1117 Make sure only allowable flags are used for allocation
1118-1119 Do not grow the cache if this is set. In reality, it is never set
1127-1128 If this called within interrupt context, make sure the ATOMIC flag is set
1130 This flag tells the constructor it is to init the object
1131 The local_flags are just those relevant to the page allocator
1132-1137 If the SLAB_ATOMIC flag is set, the constructor needs to know about it in case it wants to make new allocations

```c
spin_lock_irqsave(&cachep->spinlock, save_flags);
offset = cachep->colour_next;
cachep->colour_next++;
if (cachep->colour_next >= cachep->colour)
  cachep->colour_next = 0;
offset *= cachep->colour_off;
cachep->dflags |= DFLGS_GROWN;
cachep->growing++;
spin_unlock_irqrestore(&cachep->spinlock, save_flags);
```
4.1.3. Slab Creation

Calculate colour offset for objects in this slab

1140 Acquire an interrupt safe lock for accessing the cache descriptor
1143 Get the offset for objects in this slab
1144 Move to the next colour offset
1145-1146 If colour has been reached, there is no more offsets available, so reset colour_next to 0
1147 colour_off is the size of each offset, so offset * colour_off will give how many bytes to offset the objects to
1148 Mark the cache that it is growing so that kmem_cache_reap() will ignore this cache
1150 Increase the count for callers growing this cache
1151 Free the spinlock and re-enable interrupts

1163 if (!(objp = kmem_getpages(cachep, flags)))
1164 goto failed;
1165
1167 if (!(slabp = kmem_cache_slabmgmt(cachep, objp, offset, local_flags)))
1158 goto opps1;

Allocate memory for slab and acquire a slab descriptor

1163-1164 Allocate pages from the page allocator for the slab. See Section 4.6
1167 Acquire a slab descriptor. See Section 4.1.1

1171 i = 1 << cachep->gfporder;
1172 page = virt_to_page(objp);
1173 do {
1174 SET_PAGE_CACHE(page, cachep);
1175 SET_PAGE_SLAB(page, slabp);
1176 PageSetSlab(page);
1177 page++;
1178 } while (--i);

Link the pages for the slab used to the slab and cache descriptors

1171 i is the number of pages used for the slab. Each page has to be linked to the slab and cache descriptors.
1172 objp is a pointer to the beginning of the slab. The macro virt_to_page() will give the struct page for that address
4.1.3. Slab Creation

1173-1178 Link each pages list field to the slab and cache descriptors

1174 SET_PAGE_CACHE() links the page to the cache descriptor. See the companion document for details

1176 SET_PAGE_SLAB() links the page to the slab descriptor. See the companion document for details

1176 Set the PG_slab page flag. See the companion document for a full list of page flags

1177 Move to the next page for this slab to be linked

1180 kmem_cache_init_objs(cachep, slabp, ctor_flags);

1180 Initialise all objects. See Section 4.2.1

1182 spin_lock_irqsave(&cachep->spinlock, save_flags);
1183 cachep->growing--;
1184
1186 list_add_tail(&slabp->list, &cachep->slabs_free);
1187 STATS_INC_GROWN(cachep);
1188 cachep->failures = 0;
1189
1190 spin_unlock_irqrestore(&cachep->spinlock, save_flags);
1191 return 1;

Add the slab to the cache

1182 Acquire the cache descriptor spinlock in an interrupt safe fashion

1183 Decrease the growing count

1186 Add the slab to the end of the slabs_free list

1187 If STATS is set, increase the cachep→grown field

1188 Set failures to 0. This field is never used elsewhere

1190 Unlock the spinlock in an interrupt safe fashion

1191 Return success

1192 opps1:
1193 kmem_freepages(cachep, objp);
1194 failed:
1195 spin_lock_irqsave(&cachep->spinlock, save_flags);
1196 cachep->growing--;
1197 spin_unlock_irqrestore(&cachep->spinlock, save_flags);
1298 return 0;
1299 }
4.1.4 Slab Destroying

When a cache is been shrunk or destroyed, the slabs will be deleted. As the objects may have destructors, they must be called so the tasks of this function are

- If available, call the destructor for every object in the slab
- If debugging is enabled, check the red marking and poison pattern
- Free the pages the slab uses

The call graph at Figure 4.6 is very simple.

![Call Graph](call_graph.png)

**Function: kmem_slab_destroy (mm/slab.c)**

The debugging section has been omitted from this function but are almost identical to the debugging section during object allocation. See Section 4.2.1 for how the markers and poison pattern are checked.

```c
static void kmem_slab_destroy (kmem_cache_t *cachep, slab_t *slabp)
{
    if (cachep->dtor
        ) {
        int i;
        for (i = 0; i < cachep->num; i++) {
            void* objp = slabp->s_mem+cachep->objsize*i;

            DEBUG: Check red zone markers
```
4.2. Objects

This section will cover how objects are managed. At this point, most of the real hard work has been completed by either the cache or slab managers.

4.2.1 Initialising Objects in a Slab

When a slab is created, all the objects in it put in an initialised state. If a constructor is available, it is called for each object and it is expected when an object is freed, it is left in its initialised state. Conceptually this is very simple, cycle through all objects and call the constructor and initialise the `kmem_bufctl` for it. The function `kmem_cache_init_objs()` is responsible for initialising the objects.

**Function: kmem_cache_init_objs (mm/slab.c)**

The vast part of this function is involved with debugging so we will start with the function without the debugging and explain that in detail before handling the debugging part. The two sections that are debugging are marked in the code excerpt below as Part 1 and Part 2.

```c
1056 static inline void kmem_cache_init_objs (kmem_cache_t * cachep,
1057                                             slab_t * slabp, unsigned long ctor_flags)
1058 {
```

575 if (cachep->dtor)
576 (cachep->dtor)(objp, cachep, 0);

577-584 DEBUG: Check poison pattern

585 }
586 }
587
588 kmem_freepages(cachep, slabp->s_mem-slabp->colouroff);
589 if (OFF_SLAB(cachep))
590     kmem_cache_free(cachep->slabp_cache, slabp);
591 }

557-586 If a destructor is available, call it for each object in the slab
563-585 Cycle through each object in the slab
564 Calculate the address of the object to destroy
575-576 Call the destructor
588 Free the pages been used for the slab
589 If the slab descriptor is been kept off-slab, then free the memory been used for it
4.2.1. Initialising Objects in a Slab

```c
int i;
for (i = 0; i < cachep->num; i++) {
    void* objp = slabp->s_mem+cachep->objsize*i;
    /* Debugging Part 1 */
    if (cachep->ctor)
        cachep->ctor(objp, cachep, ctor_flags);
    /* Debugging Part 2 */
    slab_bufctl(slabp)[i] = i+1;
    slab_bufctl(slabp)[i-1] = BUFCTL_END;
    slabp->free = 0;
}
```

The parameters of the function are:

- `cachep` The cache the objects are been initialised for
- `slabp` The slab the objects are in
- `ctor_flags` Flags the constructor needs whether this is an atomic allocation or not

Initialise `cache->num` number of objects.

The base address for objects in the slab is `s_mem`. The address of the object to allocate is then `i * (size of a single object)`.

If a constructor is available, call it.

The macro `slab_bufctl()` casts `slabp` to a `slab_t` slab descriptor and adds one to it. This brings the pointer to the end of the slab descriptor and then casts it back to a `kmem_bufctl_t` effectively giving the beginning of the bufctl array.

The index of the first free object is 0 in the bufctl array.

That covers the core of initialising objects. Next the first debugging part will be covered.

```c
#if DEBUG
    if (cachep->flags & SLAB_RED_ZONE) {
        *((unsigned long*)(objp)) = RED_MAGIC1;
        *((unsigned long*)(objp + cachep->objsize -
            BYTES_PER_WORD)) = RED_MAGIC1;
        objp += BYTES_PER_WORD;
    }
#endif
```
4.2.2. Object Allocation

1064 If the cache is to be red zones then place a marker at either end of the object

1065 Place the marker at the beginning of the object

1066 Place the marker at the end of the object. Remember that the size of the object takes into account the size of the red markers when red zoning is enabled

1068 Increase the objp pointer by the size of the marker for the benefit of the constructor which is called after this debugging block

1079 #if DEBUG
1080     if (cachep->flags & SLAB_RED_ZONE)
1081         objp -= BYTES_PER_WORD;
1082     if (cachep->flags & SLAB_POISON)
1083         kmem_poison_obj(cachep, objp);
1084     if (cachep->flags & SLAB_RED_ZONE) {
1085         if (*((unsigned long*)(objp)) != RED_MAGIC1)
1086             BUG();
1087         if (*((unsigned long*)(objp + cachep->objsize - BYTES_PER_WORD)) != RED_MAGIC1)
1088             BUG();
1089     }
1091 #endif
1092

This is the debugging block that takes place after the constructor, if it exists, has been called.

1080-1081 The objp was increased by the size of the red marker in the previous debugging block so move it back again

1082-1084 If there was no constructor, poison the object with a known pattern that can be examined later to trap uninitialised writes

1086 Check to make sure the red marker at the beginning of the object was preserved to trap writes before the object

1088-1089 Check to make sure writes didn’t take place past the end of the object

4.2.2 Object Allocation

Function: kmem_cache_alloc (mm/slab.c)

This trivial function simply calls __kmem_cache_alloc().

1527 void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)
1529 {
1530     return __kmem_cache_alloc(cachep, flags);
4.2.2. Object Allocation

**Function: **\_\_kmem\_cache\_alloc (UP Case) (mm/slab.c)

This will take the parts of the function specific to the UP case. The SMP case will be dealt with in the next section.

```c
1336 static inline void * \_\_kmem\_cache\_alloc (kmem_cache_t *cachep, int flags)
1337 {
1338     unsigned long save_flags;
1339     void* objp;
1340     kmem_cache_alloc_head(cachep, flags);
1342     try_again:
1343         local_irq_save(save_flags);
1345         objp = kmem_cache_alloc_one(cachep);
1367     local_irq_restore(save_flags);
1368     return objp;
1369 
1374     local_irq_restore(save_flags);
1375     if (kmem_cache_grow(cachep, flags))
1379         goto try_again;
1380     return NULL;
1381 }
```

The parameters are the cache to allocate from and allocation specific flags.

This function makes sure the appropriate combination of DMA flags are in use.

Disable interrupts and save the flags. This function is used by interrupts so this is the only way to provide synchronisation in the UP case.

This macro (see Section 4.2.2) allocates an object from one of the lists and returns it. If no objects are free, it calls goto alloc\_new\_slab at the end of this function.
4.2.2. Object Allocation

1367-1368 Restore interrupts and return

1374 At this label, no objects were free in slabs_partial and slabs_free is empty so a new slab is needed

1375 Allocate a new slab (see Section 4.1.3)

1379 A new slab is available so try again

1380 No slabs could be allocated so return failure

Function: __kmem_cache_alloc (SMP Case) (mm/slab.c)
This is what the function looks like in the SMP case

1336 static inline void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)
1337 {
1338     unsigned long save_flags;
1339     void* objp;
1340     kmem_cache_alloc_head(cachep, flags);
1342     try_again:
1343         local_irq_save(save_flags);
1344         {
1345             cpucache_t *cc = cc_data(cachep);
1347             if (cc) {
1349                 if (cc->avail) {
1350                     STATS_INC_ALLOCHIT(cachep);
1351                     objp = cc_entry(cc)[--cc->avail];
1352                 } else {
1353                     STATS_INC_ALLOCMISS(cachep);
1354                     objp =
1355                         kmem_cache_alloc_batch(cachep,cc,flags);
1356                     if (!objp)
1357                         goto alloc_new_slab_nolock;
1359                     }
1360                 }
1362             } else {
1363                 spin_lock(&cachep->spinlock);
1365                 objp =
1366                     kmem_cache_alloc_one(cachep);
1368                 spin_unlock(&cachep->spinlock);
1370             }
1372             local_irq_restore(save_flags);
1374             return objp;
1376         alloc_new_slab:
1378         spin_unlock(&cachep->spinlock);
1380 alloc_new_slab_nolock:
4.2.2. Object Allocation

1373  local_irq_restore(save_flags);
1375  if (kmem_cache_grow(cachep, flags))
1379      goto try_again;
1380  return NULL;
1381 }

1336-1345 Same as UP case

1347 Obtain the per CPU data for this cpu

1348-1358 If a per CPU cache is available then ....

1349 If there is an object available then ....

1350 Update statistics for this cache if enabled

1351 Get an object and update the avail figure

1352 Else an object is not available so ....

1353 Update statistics for this cache if enabled

1354 Allocate batchcount number of objects, place all but one of them in the per CPU cache and return the last one to objp

1355-1356 The allocation failed, so goto alloc_new_slab_nolock to grow the cache and allocate a new slab

1358-1362 If a per CPU cache is not available, take out the cache spinlock and allocate one object in the same way the UP case does. This is the case during the initialisation for the cache_cache for example

1361 Object was successfully assigned, release cache spinlock

1364-1368 Re-enable interrupts and return the allocated object

1369-1370 If kmem_cache_alloc_one() failed to allocate an object, it will goto here with the spinlock still held so it must be released

1373-1381 Same as the UP case

Function: kmem_cache_alloc_head (mm/slab.c)

This simple function ensures the right combination of slab and GFP flags are used for allocation from a slab. If a cache is for DMA use, this function will make sure the caller does not accidently request normal memory and vice versa

1229 static inline void kmem_cache_alloc_head(kmem_cache_t *cachep, int flags) {
1231     if (flags & SLAB_DMA) {
1232         if (!((cachep->gfpflags & GFP_DMA))
1233             BUG();

4.2.2. Object Allocation

The parameters are the cache we are allocating from and the flags requested for the allocation.

If the caller has requested memory for DMA use and ....

Else if the caller has not requested DMA memory and this cache is for DMA use, BUG()

Function: kmem_cache_alloc_one (mm/slab.c)

This is a preprocessor macro. It may seem strange to not make this an inline function but it is a preprocessor macro for for a goto optimisation in __kmem_cache_alloc() (see Section 4.2.2)

```c
#define kmem_cache_alloc_one(cachep) 
({
    struct list_head * slabs_partial, * entry;
    slab_t *slabp;
    slabs_partial = &(cachep)->slabs_partial;
    entry = slabs_partial->next;
    if (unlikely(entry == slabs_partial)) {
        struct list_head * slabs_free;
        slabs_free = &(cachep)->slabs_free;
        entry = slabs_free->next;
        if (unlikely(entry == slabs_free))
            goto alloc_new_slab;
        list_del(entry);
        list_add(entry, slabs_partial);
    }
    slabp = list_entry(entry, slab_t, list);
    kmem_cache_alloc_one_tail(cachep, slabp);
})
```

1286-1287 Get the first slab from the slabs_partial list

1288-1296 If a slab is not available from this list, execute this block

1289-1291 Get the first slab from the slabs_free list
4.2.2. Object Allocation

If there is no slabs on slabs_free, then goto alloc_new_slab(). This goto label is in __kmem_cache_alloc() and it is will grow the cache by one slab.

Else remove the slab from the free list and place it on the slabs_partial list because an object is about to be removed from it.

Obtain the slab from the list

Allocate one object from the slab

Function: kmem_cache_alloc_one_tail (mm/slab.c)
This function is responsible for the allocation of one object from a slab. Much of it is debugging code.

```c
static inline void * kmem_cache_alloc_one_tail (kmem_cache_t *cachep, slab_t *slabp) {
    void *objp;
    STATS_INC_ALLOCED(cachep);
    STATS_INC_ACTIVE(cachep);
    STATS_SET_HIGH(cachep);
    slabp->inuse++;
    objp = slabp->s_mem + slabp->free*cachep->objsize;
    slabp->free = slab_bufctl(slabp)[slabp->free];
    if (unlikely(slabp->free == BUFCTL_END)) {
        list_del(&slabp->list);
        list_add(&slabp->list, &cachep->slabs_full);
    }
    #if DEBUG
    if (cachep->flags & SLAB_POISON)
        if (kmem_check_poison_obj(cachep, objp))
            BUG();
    if (cachep->flags & SLAB_RED_ZONE) {
        if (xchg((unsigned long *)objp, RED_MAGIC2) != RED_MAGIC1)
            BUG();
        if (xchg((unsigned long *)(objp+cachep->objsize - BYTES_PER_WORD), RED_MAGIC2) != RED_MAGIC1)
            BUG();
        objp += BYTES_PER_WORD;
    }
    #endif
    return objp;
}
```
4.2.2. Object Allocation

The parameters are the cache and slab been allocated from

If stats are enabled, this will set three statistics. **ALLOCED** is the total number of objects that have been allocated. **ACTIVE** is the number of active objects in the cache. **HIGH** is the maximum number of objects that were active as a single time

**inuse** is the number of objects active on this slab

Get a pointer to a free object. **s_mem** is a pointer to the first object on the slab. **free** is an index of a free object in the slab. **index * object size** gives an offset within the slab

This updates the free pointer to be an index of the next free object. See the companion document for seeing how to track free objects.

If the slab is full, remove it from the **slabs_partial** list and place it on the **slabs_full**.

Debugging code

Without debugging, the object is returned to the caller

If the object was poisoned with a known pattern, check it to guard against uninitialised access

If red zoning was enabled, check the marker at the beginning of the object and confirm it is safe. Change the red marker to check for writes before the object later

Check the marker at the end of the object and change it to check for writes after the object later

Update the object pointer to point to after the red marker

**Function:** kmem_cache_alloc_batch (**mm/slab.c**)

This function allocate a batch of objects to a CPU cache of objects. It is only used in the SMP case. In many ways it is very similar kmem_cache_alloc_one() (see Section 4.2.2).

```
void* kmem_cache_alloc_batch(kmem_cache_t* cachep,
    cpucache_t* cc, int flags)
{
    int batchcount = cachep->batchcount;

    spin_lock(&cachep->spinlock);

    while (batchcount--) {
        struct list_head * slabs_partial, * entry;
        slab_t *slabp;

        /* Get slab alloc is to come from. */
```
4.2.3. Object Freeing

1312 slabs_partial = &(cachep)->slabs_partial;
1313 entry = slabs_partial->next;
1314 if (unlikely(entry == slabs_partial)) {
1315   struct list_head * slabs_free;
1316   slabs_free = &(cachep)->slabs_free;
1317   entry = slabs_free->next;
1318   if (unlikely(entry == slabs_free))
1319     break;
1320   list_del(entry);
1321   list_add(entry, slabs_partial);
1322 }
1323
1324 slabp = list_entry(entry, slab_t, list);
1325 cc_entry(cc)[cc->avail++] =
1326   kmem_cache_alloc_one_tail(cachep, slabp);
1327 }
1328 spin_unlock(&cachep->spinlock);
1329
1330 if (cc->avail)
1331   return cc_entry(cc)[--cc->avail];
1332 return NULL;
1333 }

1303 The parameters are the cache to allocate from, the per CPU cache to fill and allocation flags.
1305 batchcount is the number of objects to allocate.
1307 Obtain the spinlock for access to the cache descriptor.
1308-1327 Loop batchcount times.
1309-1322 This is example the same as kmem_cache_alloc_one() (See Section 4.2.2). It selects a slab from either slabs_partial or slabs_free to allocate from. If none are available, break out of the loop.
1324-1325 Call kmem_cache_alloc_one_tail() (See Section 4.2.2) and place it in the per CPU cache.
1328 Release the cache descriptor lock.
1330-1331 Take one of the objects allocated in this batch and return it.
1332 If no object was allocated, return. __kmem_cache_alloc() will grow the cache by one slab and try again.
4.2.3 Object Freeing

Function: kmem_cache_free \( (mm/slab.c) \)

```c
1574 void kmem_cache_free (kmem_cache_t *cachep, void *objp)
1575 {
1576     unsigned long flags;
1577     #if DEBUG
1578         CHECK_PAGE(virt_to_page(objp));
1579         if (cachep != GET_PAGE_CACHE(virt_to_page(objp)))
1580             BUG();
1581     #endif
1582     local_irq_save(flags);
1583     __kmem_cache_free(cachep, objp);
1584     local_irq_restore(flags);
1585 }
```

The parameter is the cache the object is been freed from and the object itself.

1577-1581 If debugging is enabled, the page will first be checked with \texttt{CHECK\_PAGE()} to make sure it is a slab page. Secondly the page list will be examined to make sure it belongs to this cache (See Section 4.1.2)

1583 Interrupts are disabled to protect the path.

1584 \texttt{__kmem_cache_free()} will free the object to the per CPU cache for the SMP case and to the global pool in the normal case.

1585 Re-enable interrupts.
Function: \texttt{__kmem_cache_free} (\texttt{mm/slab.c})

This covers what the function looks like in the UP case. Clearly, it simply releases the object to the slab.

1491 static inline void \texttt{__kmem_cache_free} (kmem_cache_t *cachep, void* objp) 
1492 { 
1515 kmem_cache_free_one(cachep, objp); 
1517 }

Function: \texttt{__kmem_cache_free} (\texttt{mm/slab.c})

This case is slightly more interesting. In this case, the object is released to the per-cpu cache if it is available.

1491 static inline void \texttt{__kmem_cache_free} (kmem_cache_t *cachep, void* objp) 
1492 { 
1494 cpucache_t *cc = cc_data(cachep); 
1495 
1496 CHECK_PAGE(virt_to_page(objp)); 
1497 if (cc) { 
1498 int batchcount; 
1499 if (cc->avail < cc->limit) { 
1500 STATS_INC_FREEHIT(cachep); 
1501 cc_entry(cc)[cc->avail++]= objp; 
1502 return; 
1503 } 
1504 STATS_INC_FREEMISS(cachep); 
1505 batchcount = cachep->batchcount; 
1506 cc->avail -= batchcount; 
1507 free_block(cachep, 
1508 &cc_entry(cc)[cc->avail],batchcount); 
1509 cc_entry(cc)[cc->avail++]= objp; 
1510 return; 
1511 } else { 
1512 free_block(cachep, &objp, 1); 
1513 } 
1517 }

1494 Get the data for this per CPU cache (See Section 4.4)

1496 Make sure the page is a slab page

1497-1511 If a per CPU cache is available, try to use it. This is not always available. During cache destruction for instance, the per CPU caches are already gone

1499-1503 If the number of available in the per CPU cache is below limit, then add the object to the free list and return

1504 Update Statistics if enabled
The pool has overflowed so batchcount number of objects is going to be freed to the global pool.

Update the number of available (avail) objects.

Free a block of objects to the global cache.

Free the requested object and place it on the per CPU pool.

If the per CPU cache is not available, then free this object to the global pool.

**Function: kmem_cache_free_one (mm/slab.c)**

```c
static inline void kmem_cache_free_one(kmem_cache_t *cachep, void *objp) {
    slab_t* slabp;
    CHECK_PAGE(virt_to_page(objp));
    slabp = GET_PAGE_SLAB(virt_to_page(objp));

    #if DEBUG
    if (cachep->flags & SLAB_DEBUG_INITIAL)
        cachep->ctor(objp, cachep,
                      SLAB_CTOR_CONSTRUCTOR|SLAB_CTOR_VERIFY);
    
    if (cachep->flags & SLAB_RED_ZONE) {
        objp -= BYTES_PER_WORD;
        if (xchg((unsigned long *)objp, RED_MAGIC1) != RED_MAGIC2)
            BUG();
        if (xchg((unsigned long *)(objp+cachep->objsize - BYTES_PER_WORD), RED_MAGIC1) != RED_MAGIC2)
            BUG();
    }
    #endif

    if (cachep->flags & SLAB_POISON)
        kmem_poison_obj(cachep, objp);
    if (kmem_extra_free_checks(cachep, slabp, objp))
        return;

    unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
    slab_bufctl(slabp)[objnr] = slabp->free;
    slabp->free = objnr;
    STATS_DEC_ACTIVE(cachep);
```
4.2.3. Object Freeing

```c
int inuse = slabp->inuse;
if (unlikely(!--slabp->inuse)) {
    /* Was partial or full, now empty. */
    list_del(&slabp->list);
    list_add(&slabp->list, &cachep->slabs_free);
} else if (unlikely(inuse == cachep->num)) {
    /* Was full. */
    list_del(&slabp->list);
    list_add(&slabp->list, &cachep->slabs_partial);
}
```

Make sure the page is a slab page

Get the slab descriptor for the page

Debugging material. Discussed at end of section

Calculate the index for the object been freed

As this object is now free, update the bufctl to reflect that. See the companion document for seeing how to track free objects.

If statistics are enabled, disable the number of active objects in the slab

If inuse reaches 0, the slab is free and is moved to the slabs_free list

If the number in use equals the number of objects in a slab, it is full so move it to the slabs_full list

Return

If SLAB_DEBUG_INITIAL is set, the constructor is called to verify the object is in an initialised state

Verify the red marks at either end of the object are still there. This will check for writes beyond the boundaries of the object and for double frees

Poison the freed object with a known pattern

This function will confirm the object is a part of this slab and cache. It will then check the free list (bufctl) to make sure this is not a double free
4.3. Sizes Cache

Function: **free_block** *(mm/slab.c)*

This function is only used in the SMP case when the per CPU cache gets too full. It is used to free a batch of objects in bulk.

```c
static void free_block (kmem_cache_t* cachep, void** objpp, int len)
{
    spin_lock(&cachep->spinlock);
    __free_block(cachep, objpp, len);
    spin_unlock(&cachep->spinlock);
}
```

The parameters are
- **cachep**  The cache that objects are been freed from
- **objpp**  Pointer to the first object to free
- **len**  The number of objects to free

Function: **__free_block** *(mm/slab.c)*

This function is trivial. Starting with objpp, it will free len number of objects.

```c
static inline void __free_block (kmem_cache_t* cachep, void** objpp, int len)
{
    for ( ; len > 0; len--, objpp++)
        kmem_cache_free_one(cachep, *objpp);
}
```

4.3 Sizes Cache

Function: **kmem_cache_sizes_init** *(mm/slab.c)*

This function is responsible for creating pairs of caches for small memory buffers suitable for either normal or DMA memory.

```c
void __init kmem_cache_sizes_init(void)
{
    cache_sizes_t *sizes = cache_sizes;
    char name[20];
    if (num_physpages > (32 << 20) >> PAGE_SHIFT)
        slab_break_gfp_order = BREAK_GFP_ORDER_HI;
```
4.3. Sizes Cache

446 do {
452 snprintf(name, sizeof(name), "size-%ld",
        sizes->cs_size);
453 if (!(sizes->cs_cachep =
454     kmem_cache_create(name,
        sizes->cs_size,
        0, SLAB_HWCACHE_ALIGN,
        NULL, NULL))) {
        BUG();
456 }
457 }
458
460 if (!(OFF_SLAB(sizes->cs_cachep))) {
461 offslab_limit = sizes->cs_size-sizeof(slab_t);
462 offslab_limit /= 2;
463 }
464 snprintf(name, sizeof(name), "size-%ld(DMA)",
        sizes->cs_size);
465 sizes->cs_dmacachep = kmem_cache_create(name,
        sizes->cs_size, 0,
        SLAB_CACHE_DMA|SLAB_HWCACHE_ALIGN,
        NULL, NULL);
466 if (!sizes->cs_dmacachep)
467 BUG();
468 sizes++;
470 } while (sizes->cs_size);
471 }

Get a pointer to the cache_sizes array. See Section 4.3

The human readable name of the cache. Should be sized CACHE_NAMELEN which is
defined to be 20 long

slab_break_gfp_order determines how many pages a slab may use unless 0 ob-
jects fit into the slab. It is statically initialised to BREAK_GFP_ORDER_L0 (1). This check
sees if more than 32MiB of memory is available and if it is, allow BREAK_GFP_ORDER_HI
number of pages to be used because internal fragmentation is more acceptable when
more memory is available.

Create two caches for each size of memory allocation needed

Store the human readable cache name in name

Create the cache, aligned to the L1 cache. See Section 4.0.1

Calculate the off-slab bufctl limit which determines the number of objects that
can be stored in a cache when the slab descriptor is kept off-cache.

The human readable name for the cache for DMA use
4.3.1. kmalloc

Create the cache, aligned to the L1 cache and suitable for DMA user. See Section 4.0.1

If the cache failed to allocate, it is a bug. If memory is unavailable this early, the machine will not boot.

Move to the next element in the cache_sizes array

The array is terminated with a 0 as the last element

4.3.1 kmalloc

With the existence of the sizes cache, the slab allocator is able to offer a new allocator function, kmalloc() for use when small memory buffers are required. When a request is received, the appropriate sizes cache is selected and an object assigned from it. The call graph on Figure 4.9 is therefore very simple as all the hard work is in cache allocation (See Section 4.2.2)

![Figure 4.9: kmalloc](image)

**Function: kmalloc (mm/slab.c)**

```c
1553 void * kmalloc (size_t size, int flags)
1554 {
1555     cache_sizes_t *csizep = cache_sizes;
1556     for (; csizep->cs_size; csizep++) {
1557         if (size > csizep->cs_size)
1558             continue;
1559         return __kmem_cache_alloc(flags & GFP_DMA ?
1560             csizep->cs_dmacachep :
1561             csizep->cs_cachep, flags);
1562     }
1563     return NULL;
1564 }
```

1555 cache_sizes is the array of caches for each size (See Section 4.3)
4.3.2. kfree

Starting with the smallest cache, examine the size of each cache until one large enough to satisfy the request is found.

If the allocation is for use with DMA, allocate an object from `cs_dmacachep` else use the `cs_cachep`.

If a sizes cache of sufficient size was not available or an object could not be allocated, return failure.

4.3.2 kfree

Just as there is a `kmalloc()` function to allocate small memory objects for use, there is a `kfree()` for freeing it. As with `kmalloc`, the real work takes place during object freeing (See Section 4.2.3) so the call graph in Figure 4.9 is very simple.

```
void kfree (const void *objp)
{
    kmem_cache_t *c;
    unsigned long flags;

    if (!objp)
        return;

    local_irq_save(flags);
    CHECK_PAGE(virt_to_page(objp));
    c = GET_PAGE_CACHE(virt_to_page(objp));
    __kmem_cache_free(c, (void*)objp);
    local_irq_restore(flags);
}
```

Return if the pointer is NULL. This is possible if a caller used `kmalloc()` and had a catch-all failure routine which called `kfree()` immediately.
4.4. Per-CPU Object Cache

One of the tasks the slab allocator is dedicated to is improved hardware cache utilization. An aim of high performance computing in general is to use data on the same CPU for as long as possible. Linux achieves this by trying to keep objects in the same CPU cache with a Per-CPU object cache, called a cpucache for each CPU in the system.

When allocating or freeing objects, they are placed in the cpucache. When there is no objects free, a batch of objects is placed into the pool. When the pool gets too large, half of them are removed and placed in the global cache. This way the hardware cache will be used for as long as possible on the same CPU.

4.4.1 Describing the Per-CPU Object Cache

Each cache descriptor has a pointer to an array of cpucaches, described in the cache descriptor as

```c
typedef struct cpucache_s {
    unsigned int avail;
    unsigned int limit;
} cpucache_t;
```

This structure is very simple

- **avail** is the number of free objects available on this cpucache
- **limit** is the total number of free objects that can exist

A helper macro `cc_data()` is provided to give the cpucache for a given cache and processor. It is defined as

```c
#define cc_data(cachep) \
    ((cachep)->cpudata[smp_processor_id()])
```

This will take a given cache descriptor (cachep) and return a pointer from the cpucache array (cpudata). The index needed is the ID of the current processor, `smp_processor_id()`.

Pointers to objects on the cpucache are placed immediately after the cpucache_t struct. This is very similar to how objects are stored after a slab descriptor illustrated in Section 4.1.2.
4.4.2 Adding/Removing Objects from the Per-CPU Cache

To prevent fragmentation, objects are always added or removed from the end of the array. To add an object (\texttt{obj}) to the CPU cache (\texttt{cc}), the following block of code is used:

\begin{verbatim}
cc_entry(cc)[cc->avail++] = obj;
\end{verbatim}

To remove an object:

\begin{verbatim}
obj = cc_entry(cc)[--cc->avail];
\end{verbatim}

\texttt{cc_entry()} is a helper macro which gives a pointer to the first object in the cpucache. It is defined as:

\begin{verbatim}
#define cc_entry(cpucache) \n ((void **)(((cpucache_t*)(cpucache))+1))
\end{verbatim}

This takes a pointer to a \texttt{cpucache}, increments the value by the size of the \texttt{cpucache_t} descriptor giving the first object in the cache.

4.4.3 Enabling Per-CPU Caches

When a cache is created, its CPU cache has to be enabled and memory allocated for it using \texttt{kmalloc}. The function \texttt{enable_cpucache()} is responsible for deciding what size to make the cache and calling \texttt{kmem_tune_cpucache()} to allocate memory for it.

Obviously a CPU cache cannot exist until after the various sizes caches have been enabled so a global variable \texttt{g_cpucache_up()} is used to prevent cpucache's been enabled before it is possible. The function \texttt{enable_all_cpucaches()} cycles through all caches in the cache chain and enables their cpucache.

Once the CPU cache has been setup, it can be accessed without locking as a CPU will never access the wrong cpucache so it is guaranteed safe access to it.

**Function:** \texttt{enable_all_cpucaches (mm/slab.c)}

This function locks the cache chain and enables the cpucache for every cache. This is important after the cache \_cache and sizes cache have been enabled.

\begin{verbatim}
static void enable_all_cpucaches (void)
{
    struct list_head* p;
    down(&cache_chain_sem);
    p = &cache_cache.next;
    do {
        kmem_cache_t* cachep = list_entry(p, kmem_cache_t, next);
        enable_cpucache(cachep);
        p = cachep->next.next;
    }
\end{verbatim}
4.4.3. Enabling Per-CPU Caches

1724         } while (p != &cache_cache.next);
1725
1726         up(&cache_chain_sem);
1727 }

1716 Obtain the semaphore to the cache chain
1717 Get the first cache on the chain
1719-1724 Cycle through the whole chain
1720 Get a cache from the chain. This code will skip the first cache on the chain but
    cache_cache doesn’t need a cpucache as it is so rarely used
1722 Enable the cpucache
1723 Move to the next cache on the chain
1724 Release the cache chain semaphore

Function: enable_cpucache (mm/slab.c)
    This function calculates what the size of a cpucache should be based on the size of
    the objects the cache contains before calling kmem_tune_cpucache() which does the actual
    allocation.

1691 static void enable_cpucache (kmem_cache_t *cachep)
1692 {
1693     int err;
1694     int limit;
1695
1697     if (cachep->objsize > PAGE_SIZE)
1698         return;
1699     if (cachep->objsize > 1024)
1700         limit = 60;
1701     else if (cachep->objsize > 256)
1702         limit = 124;
1703     else
1704         limit = 252;
1705
1706     err = kmem_tune_cpucache(cachep, limit, limit/2);
1707     if (err)
1708         printk(KERN_ERR
1709             "enable_cpucache failed for %s, error %d.\n",
1710                 cachep->name, -err);
1711 }

1697-1698 If an object is larger than a page, don’t have a Per CPU cache. They are too expensive
4.4.3. Enabling Per-CPU Caches

If an object is larger than 1KiB, keep the cpu cache below 3MiB in size. The limit is set to 124 objects to take the size of the cpucache descriptors into account.

For smaller objects, just make sure the cache doesn’t go above 3MiB in size.

Allocate the memory for the cpucache.

Print out an error message if the allocation failed.

Function: `kmem_tune_cpucache` (*mm/slab.c*)

This function is responsible for allocating memory for the cpucaches. For each CPU on the system, kmalloc gives a block of memory large enough for one cpu cache and fills a `cpupdate_struct_t` struct. The function `smp_call_function_all_cpus()` then calls `do_cpupdate_local()` which swaps the new information with the old information in the cache descriptor.

```c
static int kmem_tune_cpucache (kmem_cache_t* cachep, int limit, int batchcount)
{
    ccupdate_struct_t new;
    int i;
    /*
     * These are admin-provided, so we are more graceful.
     */
    if (limit < 0)
        return -EINVAL;
    if (batchcount < 0)
        return -EINVAL;
    if (batchcount > limit)
        return -EINVAL;
    if (limit != 0 && !batchcount)
        return -EINVAL;
    memset(&new.new,0,sizeof(new.new));
    if (limit) {
        for (i = 0; i< smp_num_cpus; i++) {
            cpucache_t* ccnew;
            ccnew = kmalloc(sizeof(void*)*limit+sizeof(cpucache_t), GFP_KERNEL);
            if (!ccnew)
                goto oom;
            ccnew->limit = limit;
            ccnew->avail = 0;
            new.new[cpu_logical_map(i)] = ccnew;
        }
    }
    return 0;
}
```

4.4.3. Enabling Per-CPU Caches

        }
    }
    new.cachep = cachep;
    spin_lock_irq(&cachep->spinlock);
    cachep->batchcount = batchcount;
    spin_unlock_irq(&cachep->spinlock);
    smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
    for (i = 0; i < smp_num_cpus; i++) {
        cpucache_t* ccold = new.new[cpu_logical_map(i)];
        if (!ccold)
            continue;
        local_irq_disable();
        free_block(cachep, cc_entry(ccold), ccold->avail);
        local_irq_enable();
        kfree(ccold);
    }
    return 0;
}

oom:
    for (i--; i >= 0; i--)
        kfree(new.new[cpu_logical_map(i)]);
    return -ENOMEM;
}

1637 The parameters of the function are

    cachep  The cache this cpucache is been allocated for
    limit   The total number of objects that can exist in the cpucache
    batchcount  The number of objects to allocate in one batch when the cpucache is empty

1645 The number of objects in the cache cannot be negative
1647 A negative number of objects cannot be allocated in batch
1649 A batch of objects greater than the limit cannot be allocated
1651 A batchcount must be provided if the limit is positive
1654 Zero fill the update struct
1655 If a limit is provided, allocate memory for the cpucache
1656-1666 For every CPU, allocate a cpucache
1659 The amount of memory needed is limit number of pointers and the size of the cpucache descriptor
4.4.4. Updating Per-CPU Information

1661 If out of memory, clean up and exit

1663-1664 Fill in the fields for the cpucache descriptor

1665 Fill in the information for ccupdate_update_t struct

1668 Tell the ccupdate_update_t struct what cache is been updated

1669-1671 Acquire an interrupt safe lock to the cache descriptor and set its batchcount

1673 Get each CPU to update its cpucache information for itself. This swaps the old cpucaches in the cache descriptor with the new ones in new

1675-1683 After smp_call_function_all_cpus(), the old cpucaches are in new. This block of code cycles through them all, frees any objects in them and deletes the old cpucache

1684 Return success

1686 In the event there is no memory, delete all cpucaches that have been allocated up until this point and return failure

4.4.4 Updating Per-CPU Information

When the per-cpu caches have been created or changed, each CPU has to be told about it. It is not sufficient to change all the values in the cache descriptor as that would lead to cache coherency issues and spinlocks would have to used to protect the cpucache’s. Instead a ccupdate_t struct is populated with all the information each CPU needs and each CPU swaps the new data with the old information in the cache descriptor. The struct for storing the new cpucache information is defined as follows

868 typedef struct ccupdate_struct_s
869 {
870   kmem_cache_t *cachep;
871   cpucache_t *new[NR_CPUS];
872 } ccupdate_struct_t;

The cachep is the cache been updated and the array new is of the cpucache descriptors for each CPU on the system. The function smp_function_all_cpus() is used to get each CPU to call the do_ccupdate_local() function which swaps the information from ccupdate_struct_t with the information in the cache descriptor. Once the information has been swapped, the old data can be deleted.

Function: smp_function_all_cpus (mm/slab.c)

This calls the function func() for all CPU’s. In the context of the slab allocator, the function is do_ccupdate_local() and the argument is ccupdate_struct_t.
4.4.5 Draining a Per-CPU Cache

859 static void smp_call_function_all_cpus(void (*func) (void *arg),
    void *arg)
860 {
861     local_irq_disable();
862     func(arg);
863     local_irq_enable();
864     if (smp_call_function(func, arg, 1, 1))
865         BUG();
866 }
867 }

861-863 Disable interrupts locally and call the function for this CPU

865 For all other CPU’s, call the function. smp_call_function() is an architecture specific
function and will not be discussed further here

Function: do_ccupdate_local (mm/slab.c)

This function swaps the cpucache information in the cache descriptor with the information in info for this CPU.

874 static void do_ccupdate_local(void *info)
875 {
876     ccupdate_struct_t *new = (ccupdate_struct_t *)info;
877     cpucache_t *old = cc_data(new->cachep);
878     cc_data(new->cachep) = new->new[smp_processor_id()];
879     new->new[smp_processor_id()] = old;
880 }
881 }

876 info is a pointer to the ccupdate_struct_t to pass to smp_call_function_all_cpus()

877 Part of the ccupdate_struct_t is a pointer to the cache this cpucache belongs to.
cc_data() returns the cpucache_t for this processor

879 Place the new cpucache in cache descriptor. cc_data() returns the pointer to the
cpucache for this CPU.

880 Replace the pointer in new with the old cpucache so it can be deleted later by the
caller of smp_call_function_call_cpus(), kmem_tune_cpucache() for example

4.4.5 Draining a Per-CPU Cache

When a cache is been shrunk, its first step is to drain the cpucaches of any objects they
might have. This is so the slab allocator will have a clearer view of what slabs can be freed
or not. This is important because if just one object in a slab is placed in a Per-CPU cache,
that whole slab cannot be freed. If the system is tight on memory, saving a few milliseconds
on allocations is the least of its trouble.
Function: drain_cpu_caches (mm/slab.c)

885 static void drain_cpu_caches(kmem_cache_t *cachep)
886 {
887     ccupdate_struct_t new;
888     int i;
889
890     memset(&new.new,0,sizeof(new.new));
891
892     new.cachep = cachep;
893
894     down(&cache_chain_sem);
895     smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
896
897     for (i = 0; i < smp_num_cpus; i++) {
898         cpucache_t* ccold = new.new[cpu_logical_map(i)];
899         if (!ccold || (ccold->avail == 0))
900             continue;
901         local_irq_disable();
902         free_block(cachep, cc_entry(ccold), ccold->avail);
903         local_irq_enable();
904         ccold->avail = 0;
905     }
906     smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
907     up(&cache_chain_sem);
908 }

890 Blank the update structure as it is going to be clearing all data

892 Set new.cachep to cachep so that smp_call_function_all_cpus() knows what cache
it is affecting

894 Acquire the cache descriptor semaphore

895 do_ccupdate_local() swaps the cpucache_t information in the cache descriptor with
the ones in new so they can be altered here

897-905 For each CPU in the system ....

898 Get the cpucache descriptor for this CPU

899 If the structure does not exist for some reason or there is no objects available in it,
move to the next CPU

901 Disable interrupts on this processor. It is possible an allocation from an interrupt
handler elsewhere would try to access the per CPU cache

902 Free the block of objects (See Section 4.2.3)
4.5 Slab Allocator Initialisation

Here we will describe the slab allocator initialises itself. When the slab allocator creates a new cache, it allocates the \texttt{kmem_cache_t} from the \texttt{cache_cache} or \texttt{kmem_cache} cache. This is an obvious chicken and egg problem so the \texttt{cache_cache} has to be statically initialised as:

```c
static kmem_cache_t cache_cache = {
    slabs_full: LIST_HEAD_INIT(cache_cache.slabs_full),
    slabs_partial: LIST_HEAD_INIT(cache_cache.slabs_partial),
    slabs_free: LIST_HEAD_INIT(cache_cache.slabs_free),
    objsize: sizeof(kmem_cache_t),
    flags: SLAB_NO_REAP,
    spinlock: SPIN_LOCK_UNLOCKED,
    colour_off: L1_CACHE_BYTES,
    name: "kmem_cache",
};
```

- **Initialise the three lists as empty lists**
- **The size of each object is the size of a cache descriptor**
- **The creation and deleting of caches is extremely rare so do not consider it for reaping ever**
- **Initialise the spinlock unlocked**
- **Align the objects to the L1 cache**
- **The human readable name**

That statically defines all the fields that can be calculated at compile time. To initialise the rest of the struct, \texttt{kmem_cache_init()} is called from \texttt{start_kernel()}.

**Function: kmem_cache_init (mm/slab.c)**

This function will:

- Initialise the cache chain linked list
- Initialise a mutex for accessing the cache chain
- Calculate the \texttt{cache_cache} colour
Function: kmem_getpages (mm/slab.c)
This allocates pages for the slab allocator

static inline void * kmem_getpages (kmem_cache_t *cachep, unsigned long flags)
{
    void *addr;
    flags |= cachep->gfpflags;
    addr = (void*) __get_free_pages(flags, cachep->gfporder);
    return addr;
}

Whatever flags were requested for the allocation, append the cache flags to it. The only flag it may append is GFP_DMA if the cache requires DMA memory

Call the buddy allocator (See Section 2.3)

Return the pages or NULL if it failed
4.6. Interfacing with the Buddy Allocator

Function: **kmem_freepages** (*mm/slab.c*)

This frees pages for the slab allocator. Before it calls the buddy allocator API, it will remove the PG_slab bit from the page flags.

```c
static inline void kmem_freepages (kmem_cache_t *cachep, void *addr)
{
    unsigned long i = (1<<cachep->gfporder);
    struct page *page = virt_to_page(addr);

    while (i--) {
        PageClearSlab(page);
        page++;
    }

    free_pages((unsigned long)addr, cachep->gfporder);
}
```

507 Retrieve the order used for the original allocation
509 Get the struct page for the address
517-520 Clear the PG_slab bit on each page
521 Call the buddy allocator (See Section 2.4)
Chapter 5

Process Address Space

5.1 Managing the Address Space

5.2 Process Memory Descriptors

The process address space is described by the `mm_struct` defined in `<linux/sched.h>`
5.2. Process Memory Descriptors

```
210 struct mm_struct {
211     struct vm_area_struct * mmap;
212     rb_root_t mm_rb;
213     struct vm_area_struct * mmap_cache;
214     pgd_t * pgd;
215     atomic_t mm_users;
216     atomic_t mm_count;
217     int map_count;
218     struct rw_semaphore mmap_sem;
219     spinlock_t page_table_lock;
220     struct list_head mmlist;
221     unsigned long start_code, end_code, start_data, end_data;
222     unsigned long start_brk, brk, start_stack;
223     unsigned long arg_start, arg_end, env_start, env_end;
224     unsigned long rss, total_vm, locked_vm;
225     unsigned long def_flags;
226     unsigned long cpu_vm_mask;
227     unsigned long swap_address;
228     unsigned dumpable:1;
229     /* Architecture-specific MM context */
230     mm_context_t context;
231 };`
```

**mmap** The head of a linked list of all VMA regions in the address space

**mm_rb** The VMAs are arranged in a linked list and in a red-black tree. This is the root of the tree

**pgd** The Page Global Directory for this process

**mm_users** Count of the number of threads accessing an mm. A cloned thread will up this count to make sure an **mm_struct** is not destroyed early. The **swap_out()** code will increment this count when swapping out portions of the mm

**mm_count** A reference count to the mm. This is important for lazy TLB switches where a task may be using one **mm_struct** temporarily

**map_count** Number of VMAs in use

**mmap_sem** This is a long lived lock which protects the vma list for readers and writers. As the taker could run for so long, a spinlock is inappropriate. A reader of the list
5.2.1 Allocating a Descriptor

Two functions are provided to allocate. To be slightly confusing, they are essentially the name. allocate_mm() will allocate a mm_struct from the slab allocator. mm_alloc() will allocate and call the function mm_init() to initialise it.

Function: allocate_mm (kernel/fork.c)

```c
#define allocate_mm() (kmem_cache_alloc(mm_cachep, SLAB_KERNEL))
```

Allocate a mm_struct from the slab allocator
5.2.2 Initialising a Descriptor

The initial mm_struct in the system is called init_mm and is statically initialised at compile time using the macro INIT_MM().

Once it is established, new mm_struct’s are copies of their parent mm_struct copied using copy_mm() with the process specific fields initialised with init_mm().

Function: copy_mm (kernel/fork.c)

This function makes a copy of the mm_struct for the given task. This is only called from do_fork() after a new process has been created and needs its own mm_struct.
5.2.2. Initialising a Descriptor

```c
5.2.2. Initialising a Descriptor

121 tsk->min_flt = tsk->maj_flt = 0;
122 tsk->cmin_flt = tsk->cmaj_flt = 0;
123 tsk->nswap = tsk->cnswap = 0;
124
125 tsk->mm = NULL;
126 tsk->active_mm = NULL;
127
128 /*
129  * Are we cloning a kernel thread?
130  *
131  * We need to steal a active VM for that..
132  */
133 oldmm = current->mm;
134 if (!oldmm)
135    return 0;
136
137 if (clone_flags & CLONE_VM) {
138    atomic_inc(&oldmm->mm_users);
139    mm = oldmm;
140    goto good_mm;
141 }
142
143 retval = -ENOMEM;
144 mm = allocate_mm();
145 if (!mm)
146   goto fail_nomem;
147
148 /* Copy the current MM stuff.. */
149 memcpy(mm, oldmm, sizeof(*mm));
150 if (!mm_init(mm))
151   goto fail_nomem;
152
153 if (init_new_context(tsk,mm))
154   goto free_pt;
155
156 down_write(&oldmm->mmap_sem);
157 retval = dup_mmap(mm);
158 up_write(&oldmm->mmap_sem);
159
160 if (retval)
161   goto free_pt;
162
163 /*
164  * child gets a private LDT (if there was an LDT in the parent)
165  */
```
5.2.2. Initialising a Descriptor

```c
    copy_segments(tsk, mm);
    
    good_mm:
    tsk->mm = mm;
    tsk->active_mm = mm;
    return 0;

    free_pt:
    mmput(mm);

    fail_nomem:
    return retval;
  }
```

The parameters are the flags passed for clone and the task that is creating a copy of the mm_struct.

Initialise the task_struct fields related to memory management.

Borrow the mm of the current running process to copy from.

A kernel thread has no mm so it can return immediately.

If the CLONE_VM flag is set, the child process is to share the mm with the parent process. This is required by users like pthreads. The mm_users field is incremented so the mm is not destroyed prematurely later. The good_mm label sets the mm and active_mm and returns success.

Allocate a new mm.

Copy the parent mm and initialise the process specific mm fields with init_mm().

 Initialise the MMU context for architectures that do not automatically manage their MMU.

Call dup_mmap() which is responsible for copying all the VMAs regions in use by the parent process.

dup_mmap() returns 0 on success. If it failed, the label free_pt will call mmput() which decrements the use count of the mm.

This copies the LDT for the new process based on the parent process.

Set the new mm, active_mm and return success.

**Function: mm_init (kernel/fork.c)**

This function initialises process specific mm fields.
5.2.3 Destroying a Descriptor

A new user to an mm increments the usage count with a simple call,

atomic_inc(&mm->mm_users);  

It is decremented with a call to mmput(). If the mm_users count reaches zero, all the mapped regions are deleted with exit_mmap() and the page tables destroyed as there is no longer any users of the userspace portions. The mm_count count is decremented with mmdrop() as all the users of the page tables and VMAs are counted as one mm_struct user. When mm_count reaches zero, the mm_struct will be destroyed.

**Function: mmput (kernel/fork.c)**

275 void mmput(struct mm_struct *mm)  
276 {  
277     if (atomic_dec_and_lock(&mm->mm_users, &mmlist_lock)) {  
278         extern struct mm_struct *swap_mm;  
279         if (swap_mm == mm)  

5.2.3. Destroying a Descriptor

```c
280     swap_mm = list_entry(mm->mmlist.next,
                     struct mm_struct, mmlist);
281     list_del(&mm->mmlist);
282     mmlist_nr--;
283     spin_unlock(&mmlist_lock);
284     exit_mmap(mm);
285     mmdrop(mm);
286     }
287 }
```

277 Atomically decrement the mm_users field while holding the mmlist_lock lock. Return with the lock held if the count reaches zero

278-285 If the usage count reaches zero, the mm and associated structures need to be removed

278-280 The swap_mm is the last mm that was swapped out by the vmscan code. If the current process was the last mm swapped, move to the next entry in the list

281 Remove this mm from the list

282-283 Reduce the count of mms in the list and release the mmlist lock

284 Remove all associated mappings

285 Delete the mm

**Function:** mmdrop (include/linux/sched.h)

```c
767 static inline void mmdrop(struct mm_struct * mm)
768 {
769     if (atomic_dec_and_test(&mm->mm_count))
770         __mmdrop(mm);
771 }
```

769 Atomically decrement the reference count. The reference count could be higher if the mm was been used by lazy tlb switching tasks

770 If the reference count reaches zero, call __mmdrop()

**Function:** __mmdrop (kernel/fork.c)

```c
264 inline void __mmdrop(struct mm_struct *mm)
265 {
266     BUG_ON(mm == &init_mm);
267     pgd_free(mm->pgd);
268     destroy_context(mm);
269     free_mm(mm);
270 }
```
5.3 Memory Regions

266 Make sure the init_mm is not destroyed

267 Delete the PGD entry

268 Delete the LDT

269 Call kmem_cache_free() for the mm freeing it with the slab allocator

5.3 Memory Regions

44 struct vm_area_struct {
45     struct mm_struct * vm_mm;
46     unsigned long vm_start;
47     unsigned long vm_end;
48
49     /* linked list of VM areas per task, sorted by address */
50     struct vm_area_struct *vm_next;
51
52     pgprot_t vm_page_prot;
53     unsigned long vm_flags;
54
55     rb_node_t vm_rb;
56
57     struct vm_area_struct *vm_next_share;
58     struct vm_area_struct **vm_pprev_share;
59
60     /* Function pointers to deal with this struct. */
61     struct vm_operations_struct * vm_ops;
62
63     /* Information about our backing store: */
64     unsigned long vm_pgoff;
65     struct file * vm_file;
66     unsigned long vm_raend;
67     void * vm_private_data;
68
69 };}

vm_mm The mm_struct this VMA belongs to

vm_start The starting address

vm_end The end address

vm_next All the VMAs in an address space are linked together in an address ordered linked list with this field

vm_page_prot The protection flags for all pages in this VMA. See the companion document for a full list of flags
5.3. Memory Regions

vm_rb As well as been in a linked list, all the VMAs are stored on a red-black tree for fast lookups

vm_next_share Shared VMA regions such as shared library mappings are linked together with this field

vm_pprev_share The complement to vm_next_share

vm_ops The vm_ops field contains functions pointers for open(), close() and nopage(). These are needed for syncing with information from the disk

vm_pgoff This is the page aligned offset within a file that is mmap’ed

vm_file The struct file pointer to the file been mapped

vm_raend This is the end address of a readahead window. When a fault occurs, a readahead window will page in a number of pages after the fault address. This field records how far to read ahead

vm_private_data Used by some device drivers to store private information. Not of concern to the memory manager

As mentioned, all the regions are linked together on a linked list ordered by address. When searching for a free area, it is a simple matter of traversing the list. A frequent operation is to search for the VMA for a particular address, during page faulting for example. In this case, the Red-Black tree is traversed as it has $O(\log N)$ search time on average.

In the event the region is backed by a file, the vm_file leads to an associated address_space. The struct contains information of relevance to the filesystem such as the number of dirty pages which must be flushed to disk. It is defined as follows in `<linux/fs.h>`

```c
struct address_space {
    struct list_head clean_pages;
    struct list_head dirty_pages;
    struct list_head locked_pages;
    unsigned long nrpages;
    struct address_space_operations *a_ops;
    struct inode *host;
    struct vm_area_struct *i_mmap;
    struct vm_area_struct *i_mmap_shared;
    spinlock_t i_shared_lock;
    int gfp_mask;
};
```

clean_pages A list of clean pages which do not have to be synchronized with the disk

dirty_pages Pages that the process has touched and need to be sync-ed

locked_pages The number of pages locked in memory
5.3. Memory Regions

nrpages Number of resident pages in use by the address space

a_ops A struct of function pointers within the filesystem

host The host inode the file belongs to

i_mmap A pointer to the vma the address space is part of

i_mmap_shared A pointer to the next VMA which shares this address space

i_shared_lock A spinlock to protect this structure

gfp_mask The mask to use when calling __alloc_pages() for new pages

Periodically the memory manager will need to flush information to disk. The memory manager doesn’t know and doesn’t care how information is written to disk, so the a_ops struct is used to call the relevant functions. It is defined as follows in <linux/fs.h>

```
382 struct address_space_operations {
383     int (*writepage)(struct page *);
384     int (*readpage)(struct file *, struct page *);
385     int (*sync_page)(struct page *);
386     /*
387     * ext3 requires that a successful prepare_write()
388     * call be followed
389     * by a commit_write() call - they must be balanced
390     */
391     int (*prepare_write)(struct file *, struct page *,
392                          unsigned, unsigned);
393     int (*commit_write)(struct file *, struct page *,
394                          unsigned, unsigned);
395     /* Unfortunately this kludge is needed for FIBMAP.
396     * Don’t use it */
397     int (*bmap)(struct address_space *, long);
398     int (*flushpage)(struct page *, unsigned long);
399     int (*releasepage)(struct page *, int);
400 #define KERNEL_HAS_O_DIRECT
401     int (*direct_IO)(int, struct inode *, struct kiobuf *,
402                        unsigned long, int);
403 }
```

writepage Write a page to disk. The offset within the file to write to is stored within the page struct. It is up to the filesystem specific code to find the block. See buffer.c:block_write_full_page()

readpage Read a page from disk. See buffer.c:block_read_full_page()

sync_page Sync a dirty page with disk. See buffer.c:block_sync_page()
5.3.1 Creating A Memory Region

**prepare_write** This is called before data is copied from userspace into a page that will be written to disk. With a journaled filesystem, this ensures the filesystem log is up to date. With normal filesystems, it makes sure the needed buffer pages are allocated. See `buffer.c:block_prepare_write()`

**commit_write** After the data has been copied from userspace, this function is called to commit the information to disk. See `buffer.c:block_commit_write()`

**bmap** Maps a block so raw IO can be performed. Only of concern to the filesystem specific code.

**flushpage** This makes sure there is no IO pending on a page before releasing it. See `buffer.c:discard_bh_page()`

**releasepage** This tries to flush all the buffers associated with a page before freeing the page itself. See `try_to_free_buffers()`

### 5.3.1 Creating A Memory Region

The system call `mmap()` is provided for creating new memory regions within a process. For the x86, the function calls `sys_mmap2()` which calls `do_mmap2()` directly with the same parameters. `do_mmap2()` is responsible for acquiring the parameters for `do_mmap_pgooff()` to use which is the principle function for creating new areas for all architectures.

`do_mmap2()` first clears the `MAP_DENYWRITE` and `MAP_EXECUTABLE` bits from the `flags` parameter as they are ignored by Linux, which is confirmed by the `mmap()` manual page. If a file is being mapped, `do_mmap2()` will look up the `struct file` based on the file descriptor passed as a parameter and acquire the `mm_struct→mmap_sem` semaphore before calling `do_mmap_pgooff()`.

`do_mmap_pgooff()` begins by performing some basic sanity checks. It first checks the appropriate filesystem or device functions are available if a file or device is being mapped. It then ensures the size of the mapping is page aligned and that it does not attempt to create a mapping in the kernel portion of the address space. It then makes sure the size of the mapping does not overflow the range of `pgooff` and finally that the process does not have too many mapped regions already.

**Function: do_mmap_pgooff** (`mm/mmap.c`)

This function is very large and so is broken up into a number of sections. Broadly speaking the sections are

- Sanity check the parameters
- Find a free linear address space large enough for the memory mapping. If a filesystem or device specific `get_unmapped_area()` function is provided, it will be used otherwise `arch_get_unmapped_area()` is called
- Calculate the VM flags and check them against the file access permissions
- If an old area exists where the mapping is to take place, fix it up so it is suitable for the new mapping
5.3.1. Creating A Memory Region

Figure 5.1: Call Graph: `sys_mmap2()`
5.3.1. Creating A Memory Region

- Allocate a \texttt{vm\_area\_struct} from the slab allocator and fill in its entries
- Link in the new VMA
- Call the filesystem or device specific \texttt{mmap()} function
- Update statistics and exit

393 unsigned long do_mmap_pgoff(struct file * file, unsigned long addr,
                          unsigned long len, unsigned long prot,
                          unsigned long flags, unsigned long pgoff)
394 {
395   struct mm_struct * mm = current->mm;
396   struct vm_area_struct * vma, * prev;
397   unsigned int vm_flags;
398   int correct_wcount = 0;
399   int error;
400   rb_node_t ** rb_link, * rb_parent;
401   if (file && (!file->f_op || !file->f_op->mmap))
402       return -ENODEV;
403   if ((len = PAGE_ALIGN(len)) == 0)
404       return addr;
405   if (len > TASK_SIZE)
406       return -EINVAL;
407   /* offset overflow? */
408   if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
409       return -EINVAL;
410   /* Too many mappings? */
411   if (mm->map_count > max_map_count)
412       return -ENOMEM;
413
414   if (file && (!file->f_op || !file->f_op->mmap))
415       return -ENODEV;
416   if ((len = PAGE_ALIGN(len)) == 0)
417       return addr;
418   if (len > TASK_SIZE)
419       return -EINVAL;
420   /* offset overflow? */
421   if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
422       return -EINVAL;
423   /* Too many mappings? */
424   if (mm->map_count > max_map_count)
425       return -ENOMEM;
426
393 The parameters which correspond directly to the parameters to the mmap system call
are

\begin{itemize}
  \item \texttt{file} the struct file to mmap if this is a file backed mapping
  \item \texttt{addr} the requested address to map
  \item \texttt{len} the length in bytes to mmap
  \item \texttt{prot} is the permissions on the area
  \item \texttt{flags} are the flags for the mapping
  \item \texttt{pgoff} is the offset within the file to begin the mmap at
5.3.1. Creating A Memory Region

403-404 If a file or device is been mapped, make sure a filesystem or device specific mmap function is provided. For most filesystems, this is `generic_file_mmap()`.

406-407 Make sure a zero length mmap is not requested.

409 Ensure that it is possible to map the requested area. The limit on the x86 is `PAGE_OFFSET` or 3GiB.

413-414 Ensure the mapping will not overflow the end of the largest possible file size.

417-488 Only `max_map_count` number of mappings are allowed. By default this value is `DEFAULT_MAX_MAP_COUNT` or 65536 mappings.

420 /* Obtain the address to map to. We verify (or select) it and
421 * ensure that it represents a valid section of the address space.
422 */
423 addr = get_unmapped_area(file, addr, len, pgoff, flags);
424 if (addr & ~PAGE_MASK)
425 return addr;
426
427 After basic sanity checks, this function will call the device or file specific `get_unmapped_area()` function. If a device specific one is unavailable, `arch_get_unmapped_area()` is called. This function is discussed in Section 5.3.3.

428 /* Do simple checking here so the lower-level routines won’t have
429 * to. We assume access permissions have been handled by the open
430 * of the memory object, so we don’t do any here.
431 */
432 vm_flags = calc_vm_flags(prot, flags) | mm->def_flags
433 | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
434
435 /* mlock MCL_FUTURE? */
436 if (vm_flags & VM_LOCKED) {
437 unsigned long locked = mm->locked_vm << PAGE_SHIFT;
438 locked += len;
439 if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
440 return -EAGAIN;
441 }
442
443 calc_vm_flags() translates the `prot` and `flags` from userspace and translates them to their `VM_` equivalents.
434-438 Check if it has been requested that all future mappings be locked in memory. If yes, make sure the process isn’t locking more memory than it is allowed to. If it is, return -EAGAIN.
If a file is been memory mapped, check the files access permissions

441-468 If a file is been memory mapped, check the files access permissions
5.3.1. Creating A Memory Region

444-445 If write access is requested, make sure the file is opened for write

448-449 Similarly, if the file is opened for append, make sure it cannot be written to. It is unclear why it is not the prot field that is checked here

451 If the file is mandatory locked, return EAGAIN so the caller will try a second type

455-457 Fix up the flags to be consistent with the file flags

461-462 Make sure the file can be read before mmapping it

469-479 If the file is been mapped for anonymous use, fix up the flags if the requested mapping is MAP_PRIVATE to make sure the flags are consistent

480 481 /* Clear old maps */
482 munmap_back:
483     vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
484     if (vma && vma->vm_start < addr + len) {
485         if (do_munmap(mm, addr, len))
486             return -ENOMEM;
487     } else {
488         goto munmap_back;
489     }
490 491 /* Check against address space limit. */
492     if ((mm->total_vm << PAGE_SHIFT) + len > current->rlim[RLIMIT_AS].rlim_cur)
493         return -ENOMEM;
494 495 /* Private writable mapping? Check memory availability.. */
496     if ((vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
497         !(flags & MAP_NORESERVE) &&
498         !vm_enough_memory(len >> PAGE_SHIFT))
499         return -ENOMEM;
500 501 /* Can we just expand an old anonymous mapping? */
502     if (!file && !(vm_flags & VM_SHARED) && ri_parent)
503         if (vma_merge(mm, prev, rb_parent, addr, addr + len,
504                         vm_flags))
505                 goto out;

483 This function steps through the RB tree for the vma corresponding to a given address

484-486 If a VMA was found and it is part of the new mmapping, remove the old mapping as the new one will cover both
5.3.1. Creating A Memory Region

491-493 Make sure the new mapping will not exceed the total VM a process is allowed to have. It is unclear why this check is not made earlier.

496-499 If the caller does not specifically request that free space is not checked with MAP_NORESERVE and it is a private mapping, make sure enough memory is available to satisfy the mapping under current conditions.

502-504 If two adjacent anonymous memory mappings can be treated as one, expand an old mapping rather than creating a new one.
/* Determine the object being mapped and call the appropriate */
* specific mapper. the address has already been validated, but
* not unmapped, but the maps are removed from the list.
*/

vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma)
    return -ENOMEM;

vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = protection_map[vm_flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = pgoff;
vma->vm_file = NULL;
vma->vm_private_data = NULL;
vma->vm_raend = 0;

if (file) {
    error = -EINVAL;
    if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
        goto free_vma;
    if (vm_flags & VM_DENYWRITE) {
        error = deny_write_access(file);
        if (error)
            goto free_vma;
        correct_wcount = 1;
    }
    vma->vm_file = file;
    get_file(file);
    error = file->f_op->mmap(file, vma);
    if (error)
        goto unmap_and_free_vma;
} else if (flags & MAP_SHARED) {
    error = shmem_zero_setup(vma);
    if (error)
        goto free_vma;
}

Allocate a vm_area_struct from the slab allocator

Fill in the basic vm_area_struct fields
5.3.1. Creating A Memory Region

525-540 Fill in the file related fields if this is a file been mapped

527-528 These are both invalid flags for a file mapping so free the `vm_area_struct` and return

529-534 This flag is cleared by the system call `mmap` so it is unclear why the check is still made. Historically, an ETXTBUSY signal was sent to the calling process if the underlying file was been written to

535 Fill in the `vm_file` field

536 This increments the file use count

537 Call the filesystem or device specific `mmap` function

538-539 If an error called, goto `unmap_and_free_vma` to clean up and return the error

541 If an anonymous shared mapping is required, call `shmem_zero_setup()` to do the hard work
5.3.1. Creating A Memory Region

546       /* Can addr have changed??
547       *
548       * Answer: Yes, several device drivers can do it in their
549       * f_op->mmap method. -DaveM
550       */
551       if (addr != vma->vm_start) {
552           /*
553           * It is a bit too late to pretend changing the virtual
554           * area of the mapping, we just corrupted userspace
555           * in the do_munmap, so FIXME (not in 2.4 to avoid
556           * the driver API).
557           */
558           struct vm_area_struct * stale_vma;
559           /* Since addr changed, we rely on the mmap op to prevent
560           * collisions with existing vmas and just use
561           * find_vma_prepare
562           * to update the tree pointers.
563           */
564           addr = vma->vm_start;
565           stale_vma = find_vma_prepare(mm, addr, &prev,
566               &rb_link, &rb_parent);
567           /*
568           * Make sure the lowlevel driver did its job right.
569           */
570           if (unlikely(stale_vma && stale_vma->vm_start <
571               vma->vm_end)) {
572               printk(KERN_ERR "buggy mmap operation: [<%p>]
",
573                   file ? file->f_op->mmap : NULL);
574               BUG();
575           }
576       }
577       vma_link(mm, vma, prev, rb_link, rb_parent);
578       if (correct_wcount)
579           atomic_inc(&file->f_dentry->d_inode->i_writecount);

551-574 If the address has changed, it means the device specific mmap operation mapped
the vma somewhere else. find_vma_prepare() is used to find the new vma that was
set up

576 Link in the new vm_area_struct

577-578 Update the file write count
5.3.2. Finding a Mapped Memory Region

Function: find_vma (mm/mmap.c)

659 struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
660 {  
661    struct vm_area_struct *vma = NULL;
662    if (mm) {
663      /* Check the cache first. */
664      /* (Cache hit rate is typically around 35%). */
665      vma = mm->mmap_cache;
666      if (!(vma && vma->vm_end > addr && vma->vm_start <= addr))
667      {
668          rb_node_t * rb_node;
5.3.2. Finding a Mapped Memory Region

669 rb_node = mm->mm_rb.rb_node;
670 vma = NULL;
671
672 while (rb_node) {
673     struct vm_area_struct * vma_tmp;
674     vma_tmp = rb_entry(rb_node,
675                         struct vm_area_struct, vm_rb);
676     if (vma_tmp->vm_end > addr) {
677         vma = vma_tmp;
678         if (vma_tmp->vm_start <= addr)
679             break;
680         else
681             rb_node = rb_node->rb_left;
682     } else
683         rb_node = rb_node->rb_right;
684 }
685     if (vma)
686         mm->mmap_cache = vma;
687 }
688 return vma;
689 }

659 The two parameters are the top level mm_struct that is to be searched and the address the caller is interested in
661 Default to returning NULL for address not found
663 Make sure the caller does not try and search a bogus mm
666 mmap_cache has the result of the last call to find_vma(). This has a chance of not having to search at all through the red-black tree
667 If it is a valid VMA that is being examined, check to see if the address being searched is contained within it. If it is, the VMA was the mmap_cache one so it can be returned, otherwise the tree is searched
668-672 Start at the root of the tree
673-685 This block is the tree walk
676 The macro, as the name suggests, returns the VMA this tree node points to
678 Check if the next node traversed by the left or right leaf
680 If the current VMA is what is required, exit the while loop
If the VMA is valid, set the `mmap_cache` for the next call to `find_vma()`.

Return the VMA that contains the address or as a side effect of the tree walk, return the VMA that is closest to the requested address.

**Function: find_vma_prev** *(mm/mmap.c)*

```c
struct vm_area_struct * find_vma_prev(struct mm_struct * mm, unsigned long addr, struct vm_area_struct **pprev)
{
    if (mm) {
        /* Go through the RB tree quickly. */
        struct vm_area_struct * vma;
        rb_node_t * rb_node, * rb_last_right, * rb_prev;
        rb_node = mm->mm_rb.rb_node;
        rb_last_right = rb_prev = NULL;
        vma = NULL;
        while (rb_node) {
            struct vm_area_struct * vma_tmp;
            vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
            if (vma_tmp->vm_end > addr) {
                vma = vma_tmp;
                rb_prev = rb_last_right;
                if (vma_tmp->vm_start <= addr)
                    break;
                rb_node = rb_node->rb_left;
            } else {
                rb_last_right = rb_node;
                rb_node = rb_node->rb_right;
            }
        }
        if (vma) {
            if (vma->vm_rb.rb_left) {
                rb_prev = vma->vm_rb.rb_left;
                while (rb_prev->rb_right)
                    rb_prev = rb_prev->rb_right;
            }
            *pprev = NULL;
            if (rb_prev)
                *pprev = rb_entry(rb_prev, struct
```
5.3.3. Finding a Free Memory Region

Function: find_vma_intersection (include/linux/mm.h)

static inline struct vm_area_struct * find_vma_intersection(struct mm_struct * mm, unsigned long start_addr, unsigned long end_addr)
{
    struct vm_area_struct * vma = find_vma(mm, start_addr);
    if (vma && end_addr <= vma->vm_start)
        vma = NULL;
    return vma;
}

Return the VMA closest to the starting address

If a VMA is returned and the end address is still less than the beginning of the returned VMA, the VMA does not intersect

Return the VMA if it does intersect

5.3.3 Finding a Free Memory Region

Function: get_unmapped_area (mm/mmap.c)
5.3.3. Finding a Free Memory Region

Figure 5.2: Call Graph: `get_unmapped_area()`

```
642 unsigned long get_unmapped_area(struct file *file, unsigned long addr,
    unsigned long len, unsigned long pgoff,
    unsigned long flags)
643 {
644     if (flags & MAP_FIXED) {
645         if (addr > TASK_SIZE - len)
646             return -ENOMEM;
647         if (addr & ~PAGE_MASK)
648             return -EINVAL;
649         return addr;
650     }
651
652     if (file && file->f_op && file->f_op->get_unmapped_area)
653         return file->f_op->get_unmapped_area(file, addr,
654                                             len, pgoff, flags);
654
655     return arch_get_unmapped_area(file, addr, len, pgoff, flags);
656 }
```

The parameters passed are

- `file` The file or device being mapped
- `addr` The requested address to map to
- `len` The length of the mapping
- `pgoff` The offset within the file being mapped
- `flags` Protection flags

642-650 Sanity checked. If it is required that the mapping be placed at the specified address, make sure it will not overflow the address space and that it is page aligned

652 If the struct file provides a `get_unmapped_area()` function, use it

655 Else use the architecture specific function
Function: arch_get_unmapped_area (mm/mmap.c)

```c
612 #ifndef HAVE_ARCH_UNMAPPED_AREA
613 static inline unsigned long arch_get_unmapped_area(struct file *filp,
614       unsigned long addr, unsigned long len,
615       unsigned long pgoff, unsigned long flags)
616 {
617     struct vm_area_struct *vma;
618
619     if (len > TASK_SIZE)
620         return -ENOMEM;
621
622     if (addr) {
623         addr = PAGE_ALIGN(addr);
624         vma = find_vma(current->mm, addr);
625         if (TASK_SIZE - len >= addr &&
626             (!vma || addr + len <= vma->vm_start))
627             return addr;
628     }
629     addr = PAGE_ALIGN(TASK_UNMAPPED_BASE);
630
631     for (vma = find_vma(current->mm, addr); vma; vma = vma->vm_next) {
632         /* At this point: (!vma || addr < vma->vm_end). */
633         if (TASK_SIZE - len < addr)
634             return -ENOMEM;
635         if (!vma || addr + len <= vma->vm_start)
636             return addr;
637         addr = vma->vm_end;
638     }
639     return addr;
640 #endif

641 If this is not defined, it means that the architecture does not provide its own
642 arch_get_unmapped_area() so this one is used instead

643 The parameters are the same as those for get_unmapped_area()

644-645 Sanity check, make sure the required map length is not too long

646-652 If an address is provided, use it for the mapping

647 Make sure the address is page aligned

648 find_vma() will return the region closest to the requested address
```
5.3.4. Inserting a memory region

623-625 Make sure the mapping will not overlap with another region. If it does not, return it as it is safe to use. Otherwise it gets ignored.

627 TASK_UNMAPPED_BASE is the starting point for searching for a free region to use.

629-636 Starting from TASK_UNMAPPED_BASE, linearly search the VMAs until a large enough region between them is found to store the new mapping. This is essentially a first fit search.

639 If an external function is provided, it still needs to be declared here.

5.3.4 Inserting a memory region

Figure 5.3: Call Graph: insert_vm_struct()

Function: __insert_vm_struct (mm/mmap.c)

This is the top level function for inserting a new vma into an address space. There is a second function like it called simply insert_vm_struct() that is not described in detail here as the only difference is the one line of code increasing the map_count.

1168 void __insert_vm_struct(struct mm_struct * mm,
1169     struct vm_area_struct * vma)
1170 {
1171     struct vm_area_struct * __vma, * prev;
The arguments are the mm_struct mm that represents the linear address space and the vm_area_struct that is to be inserted.

find_vma_prepare() locates where the new vma can be inserted. It will be inserted between prev and __vma and the required nodes for the red-black tree are also returned.

This is a check to make sure the returned vma is invalid. It is unclear how such a broken vma could exist.

This function does the actual work of linking the vma struct into the linear linked list and the red-black tree.

Increase the map_count to show a new mapping has been added.

validate_mm() is a debugging macro for red-black trees. If DEBUG_MM_RB is set, the linear list of VMAs and the tree will be traversed to make sure it is valid. The tree traversal is a recursive function so it is very important that it is used only if really necessary as a large number of mappings could cause a stack overflow. If it is not set, validate_mm() does nothing at all.

Function: find_vma_prepare

This is responsible for finding the correct places to insert a VMA at the supplied address. It returns a number of pieces of information via the actual return and the function arguments. The forward VMA to link to is returned with return. pprev is the previous node which is required because the list is a singly linked list. rb_link and rb_parent are the parent and leaf node the new VMA will be inserted between.
5.3.4. Inserting a memory region

```c
while (*__rb_link) {
    struct vm_area_struct *vma_tmp;
    __rb_parent = *__rb_link;
    vma_tmp = rb_entry(__rb_parent, struct vm_area_struct, vm_rb);
    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            return vma;
        __rb_link = &__rb_parent->rb_left;
    } else {
        __rb_link = &__rb_parent->rb_right;
    }
}
*pprev = NULL;
if (rb_prev)
    *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
*rb_link = __rb_link;
*rb_parent = __rb_parent;
return vma;
```

The function arguments are described above

Initialise the search

This is a similar tree walk to what was described for `find_vma()`. The only real difference is the nodes last traversed are remembered with the `__rb_link` and `__rb_parent` variables

Get the back linking vma via the red-black tree

Return the forward linking VMA

**Function: vma_link** (*mm/mmap.c*)

This is the top-level function for linking a VMA into the proper lists. It is responsible for acquiring the necessary locks to make a safe insertion
5.3.4. Inserting a memory region

```c
rb_node_t ** rb_link, rb_node_t * rb_parent)
{
    lock_vma_mappings(vma);
    spin_lock(&mm->page_table_lock);
    __vma_link(mm, vma, prev, rb_link, rb_parent);
    spin_unlock(&mm->page_table_lock);
    unlock_vma_mappings(vma);
    mm->map_count++;
    validate_mm(mm);
}
```

`mm` is the address space the vma is to be inserted into. `prev` is the backwards linked vma for the linear linked list of VMAs. `rb_link` and `rb_parent` are the nodes required to make the rb insertion.

This function acquires the spinlock protecting the `address_space` representing the file that is been memory mapped.

Acquire the page table lock which protects the whole `mm_struct`

Insert the VMA

Free the lock protecting the `mm_struct`

Unlock the `address_space` for the file

Increase the number of mappings in this mm

If `DEBUG_MM_RB` is set, the RB trees and linked lists will be checked to make sure they are still valid

**Function: __vma_link (mm/mmap.c)**

This simply calls three helper functions which are responsible for linking the VMA into the three linked lists that link VMAs together.

```c
static void __vma_link(struct mm_struct * mm,
                        struct vm_area_struct * vma,
                        struct vm_area_struct * prev,
                        rb_node_t ** rb_link, rb_node_t * rb_parent)
{
    __vma_link_list(mm, vma, prev, rb_parent);
    __vma_link_rb(mm, vma, rb_link, rb_parent);
    __vma_link_file(vma);
}
```

This links the VMA into the linear linked lists of VMAs in this mm via the `vm_next` field
5.3.4. Inserting a memory region

This links the VMA into the red-black tree of VMAs in this mm whose root is stored in the vm_rb field

This links the VMA into the shared mapping VMA links. Memory mapped files are linked together over potentially many mms by this function via the vm_next_share and vm_pprev_share fields

Function: __vma_link_list (mm/mmap.c)

```c
282 static inline void __vma_link_list(struct mm_struct * mm,
         struct vm_area_struct * vma,
         struct vm_area_struct * prev,
         rb_node_t * rb_parent)
283 {
284     if (prev) {
285         vma->vm_next = prev->vm_next;
286         prev->vm_next = vma;
288     } else {
289         mm->mmap = vma;
290         if (rb_parent)
291             vma->vm_next = rb_entry(rb_parent, struct
292                 vm_area_struct, vm_rb);
292         else
293             vma->vm_next = NULL;
294     }
295 }
```

If prev is not null, the vma is simply inserted into the list

Else this is the first mapping and the first element of the list has to be stored in the mm_struct

The vma is stored as the parent node

Function: __vma_link_rb (mm/mmap.c)

The principle workings of this function are stored within <linux/rbtree.h> and will not be discussed in detail with this document.

```c
297 static inline void __vma_link_rb(struct mm_struct * mm,
         struct vm_area_struct * vma,
         rb_node_t ** rb_link,
         rb_node_t * rb_parent)
299 {
300     rb_link_node(&vma->vm_rb, rb_parent, rb_link);
301     rb_insert_color(&vma->vm_rb, &mm->mm_rb);
302 }
```
5.3.5. Merging contiguous region

Function: \texttt{\_\_vma\_link\_file} (\textit{mm/mmap.c})

This function links the VMA into a linked list of shared file mappings.

304 static inline void \texttt{\_\_vma\_link\_file} (struct \textit{vm\_area\_struct} * \textit{vma})
305 {
306     struct file * \textit{file};
307
308     \textit{file} = \textit{vma}->vm\_file;
309     if (\textit{file}) {
310         struct inode * \textit{inode} = \textit{file}->f\_dentry->d\_inode;
311         struct address\_space *\textit{mapping} = \textit{inode}->i\_mapping;
312         struct \textit{vm\_area\_struct} **\textit{head};
313
314         if (\textit{vma}->vm\_flags & VM\_DENYWRITE)
315             atomic\_dec(&\textit{inode}->i\_writecount);
316
317         \textit{head} = &\textit{mapping}->i\_mmap;
318         if (\textit{vma}->vm\_flags & VM\_SHARED)
319             \textit{head} = &\textit{mapping}->i\_mmap\_shared;
320
321     /* insert vma into inode's share list */
322     if((\textit{vma}->vm\_next\_share = *\textit{head}) != NULL)
323         (*\textit{head})->vm\_pprev\_share = &\textit{vma}->vm\_next\_share;
324     *\textit{head} = \textit{vma};
325     \textit{vma}->vm\_pprev\_share = \textit{head};
326 }  

309 Check to see if this VMA has a shared file mapping. If it does not, this function has nothing more to do

310-312 Extract the relevant information about the mapping from the VMA

314-315 If this mapping is not allowed to write even if the permissions are ok for writing, decrement the \texttt{i\_writecount} field. A negative value to this field indicates that the file is memory mapped and may not be written to. Efforts to open the file for writing will now fail

317-319 Check to make sure this is a shared mapping

322-325 Insert the VMA into the shared mapping linked list

5.3.5 Merging contiguous region

Function: \texttt{vma\_merge} (\textit{mm/mmap.c})

This function checks to see if a region pointed to be \texttt{prev} may be expanded forwards to cover the area from \texttt{addr} to \texttt{end} instead of allocating a new VMA. If it cannot, the VMA ahead is checked to see can it be expanded backwards instead.
5.3.5. Merging contiguous region

```c
static int vma_merge(struct mm_struct * mm, struct vm_area_struct * prev,
                      rb_node_t * rb_parent, unsigned long addr, unsigned long end,
                      unsigned long vm_flags)
{
    spinlock_t * lock = &mm->page_table_lock;
    if (!prev) {
        prev = rb_entry(rb_parent, struct vm_area_struct, vm_rb);
        goto merge_next;
    }
    if (prev->vm_end == addr && can_vma_merge(prev, vm_flags)) {
        struct vm_area_struct * next;
        spin_lock(lock);
        prev->vm_end = end;
        next = prev->vm_next;
        if (next && prev->vm_end == next->vm_start &&
            can_vma_merge(next, vm_flags)) {
            prev->vm_end = next->vm_end;
            __vma_unlink(mm, next, prev);
            spin_unlock(lock);
            mm->map_count--;
            kmem_cache_free(vm_area_cachep, next);
            return 1;
        }
        spin_unlock(lock);
        return 1;
    }
    prev = prev->vm_next;
    if (prev) {
        merge_next:
        if (!can_vma_merge(prev, vm_flags))
            return 0;
        if (end == prev->vm_start) {
            spin_lock(lock);
            prev->vm_start = addr;
            spin_unlock(lock);
            return 1;
        }
    }
    return 0;
}
```
5.3.5. Merging contiguous region

The parameters are as follows;

- **mm**: The mm the VMAs belong to
- **prev**: The VMA before the address we are interested in
- **rb_parent**: The parent RB node as returned by `find_vma_prepare()`
- **addr**: The starting address of the region to be merged
- **end**: The end of the region to be merged
- **vm_flags**: The permission flags of the region to be merged

This is the lock to the mm struct.

If prev is not passed it, it is taken to mean that the VMA being tested for merging is in front of the region from addr to end. The entry for that VMA is extracted from the **rb_parent**.

Check to see can the region pointed to by **prev** may be expanded to cover the current region.

The function **can_vma_merge()** checks the permissions of **prev** with those in **vm_flags** and that the VMA has no file mappings. If it is true, the area at **prev** may be expanded.

Lock the mm struct.

Expand the end of the VMA region (**vm_end**) to the end of the new mapping (**end**).

**next** is now the VMA in front of the newly expanded VMA.

Check if the expanded region can be merged with the VMA in front of it.

If it can, continue to expand the region to cover the next VMA.

As a VMA has been merged, one region is now defunct and may be unlinked.

No further adjustments are made to the mm struct so the lock is released.

There is one less mapped region to reduce the **map_count**.

Delete the struct describing the merged VMA.

Return success.

If this line is reached it means the region pointed to by prev could not be expanded forward so a check is made to see if the region ahead can be merged backwards instead.

Same idea as the above block except instead of adjusted **vm_end** to cover **end**, **vm_start** is expanded to cover **addr**.
5.3.6. Remapping and moving a memory region

**Function: can_vma_merge** (*include/linux/mm.h*)

This trivial function checks to see if the permissions of the supplied VMA match the permissions in `vm_flags`.

```c
static inline int can_vma_merge(struct vm_area_struct *vma, unsigned long vm_flags)
{
    if (!vma->vm_file && vma->vm_flags == vm_flags)
        return 1;
    else
        return 0;
}
```

Self explanatory, true if there is no file/device mapping and the flags equal each other

---

5.3.6 Remapping and moving a memory region

**Function: sys_mremap** (*mm/mremap.c*)

This is the system service call to remap a memory region.

```c
asmlinkage unsigned long sys_mremap(unsigned long addr,
    unsigned long old_len, unsigned long new_len,
    unsigned long flags, unsigned long new_addr)
{
    unsigned long ret;
    down_write(&current->mm->mmap_sem);
    ret = do_mremap(addr, old_len, new_len, flags, new_addr);
    up_write(&current->mm->mmap_sem);
    return ret;
}
```

---

*Figure 5.4: Call Graph: sys_mremap*
Remapping and moving a memory region

The parameters are the same as those described in the mremap man page

Acquire the mm semaphore

do_mremap() is the top level function for remapping a region

Release the mm semaphore

Return the status of the remapping

Function: do_mremap (mm/mremap.c)

This function does most of the actual “work” required to remap, resize and move a memory region. It is quite long but can be broken up into distinct parts which will be dealt with separately here. The tasks are broadly speaking

- Check usage flags and page align lengths
- Handle the condition where MAP_FIXED is set and the region is been moved to a new location.
- If a region is shrinking, allow it to happen unconditionally
- If the region is growing or moving, perform a number of checks in advance to make sure the move is allowed and safe
- Handle the case where the region is been expanded and cannot be moved
- Finally handle the case where the region has to be resized and moved

The parameters of the function are

addr is the old starting address
old_len is the old region length
new_len is the new region length
flags is the option flags passed. If MREMAP_MAYMOVE is specified, it means that the region is allowed to move if there is not enough linear address space at the current space. If MREMAP_FIXED is specified, it means that the whole region is to move to the specified new_addr with the new length. The area from new_addr to new_addr+new_len will be unmapped with do_munmap().

new_addr is the address of the new region if it is moved

At this point, the default return is EINVAL for invalid arguments

Make sure flags other than the two allowed flags are not used

The address passed in must be page aligned

Page align the passed region lengths

if (flags & MREMAP_FIXED) {
    if (new_addr & ~PAGE_MASK)
        goto out;
    if (!(flags & MREMAP_MAYMOVE))
        goto out;

    if (new_len > TASK_SIZE || new_addr > TASK_SIZE - new_len)
        goto out;

    /* Check if the location we’re moving into overlaps the old location at all, and fail if it does. */
    if ((new_addr <= addr) && (new_addr+new_len) > addr)
        goto out;

    if ((addr <= new_addr) && (addr+old_len) > new_addr)
        goto out;

    do_munmap(current->mm, new_addr, new_len);
}

This block handles the condition where the region location is fixed and must be fully moved. It ensures the area been moved to is safe and definitely unmapped.

MREMAP_FIXED is the flag which indicates the location is fixed

The new_addr requested has to be page aligned

If MREMAP_FIXED is specified, then the MAYMOVE flag must be used as well

Make sure the resized region does not exceed TASK_SIZE
5.3.6. Remapping and moving a memory region

243-244 Just as the comments indicate, the two regions been used for the move may not overlap

249 Unmap the region that is about to be used. It is presumed the caller ensures that the region is not in use for anything important

```c
ret = addr;
if (old_len >= new_len) {
    do_munmap(current->mm, addr+new_len, old_len - new_len);
    if (!(flags & MREMAP_FIXED) || (new_addr == addr))
        goto out;
}
```

256 At this point, the address of the resized region is the return value

257 If the old length is larger than the new length, then the region is shrinking

258 Unmap the unused region

259-230 If the region is not to be moved, either because MREMAP_FIXED is not used or the new address matches the old address, goto out which will return the address

```c
ret = -EFAULT;
vm = find_vma(current->mm, addr);
if (!vm || vm->vm_start > addr)
    goto out;
/* We can’t remap across vm area boundaries */
if (old_len > vm->vm_end - addr)
    goto out;
if (vm->vm_flags & VM_DONTEXPAND) {
    if (new_len > old_len)
        goto out;
}
if (vm->vm_flags & VM_LOCKED) {
    unsigned long locked = current->mm->locked_vm << PAGE_SHIFT;
    locked += new_len - old_len;
    ret = -EAGAIN;
    if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
        goto out;
}
if ((current->mm->total_vm << PAGE_SHIFT) + (new_len - old_len)
    > current->rlim[RLIMIT_AS].rlim_cur)
    goto out;
/* Private writable mapping? Check memory availability.. */
if ((vm->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
    !(flags & MAP_NORESERVE) &&
    !vm_enough_memory((new_len - old_len) >> PAGE_SHIFT))
    goto out;
```
5.3.6. Remapping and moving a memory region

Do a number of checks to make sure it is safe to grow or move the region

266 At this point, the default action is to return EFAULT causing a segmentation fault as the ranges of memory been used are invalid

267 Find the VMA responsible for the requested address

268 If the returned VMA is not responsible for this address, then an invalid address was used so return a fault

271-272 If the old_len passed in exceeds the length of the VMA, it means the user is trying to remap multiple regions which is not allowed

273-276 If the VMA has been explicitly marked as non-resizable, raise a fault

277-278 If the pages for this VMA must be locked in memory, recalculate the number of locked pages that will be kept in memory. If the number of pages exceed the ulimit set for this resource, return EAGAIN indicating to the caller that the region is locked and cannot be resized

284 The default return at this point is to indicate there is not enough memory

285-287 Ensure that the user will not exist their allowed allocation of memory

289-292 Ensure that there is enough memory to satisfy the request after the resizing

297 if (old_len == vma->vm_end - addr &&
298 !((flags & MREMAP_FIXED) && (addr != new_addr)) &&
299 (old_len != new_len || !(flags & MREMAP_MAYMOVE))) {
300 unsigned long max_addr = TASK_SIZE;
301 if (vma->vm_next)
302 max_addr = vma->vm_next->vm_start;
303 /* can we just expand the current mapping? */
304 if (max_addr - addr >= new_len) {
305 int pages = (new_len - old_len) >> PAGE_SHIFT;
306 spin_lock(&vma->vm_mm->page_table_lock);
307 vma->vm_end = addr + new_len;
308 spin_unlock(&vma->vm_mm->page_table_lock);
309 current->mm->total_vm += pages;
310 if (vma->vm_flags & VM_LOCKED) {
311 current->mm->locked_vm += pages;
312 make_pages_present(addr + old_len,
313 addr + new_len);
314 }
315 ret = addr;
316 goto out;
317 }
318 }
Handle the case where the region is been expanded and cannot be moved

297 If it is the full region that is been remapped and ...

298 The region is definitely not been moved and ...

299 The region is been expanded and cannot be moved then ...

300 Set the maximum address that can be used to TASK_SIZE, 3GiB on an x86

301-302 If there is another region, set the max address to be the start of the next region

304-317 Only allow the expansion if the newly sized region does not overlap with the next VMA

305 Calculate the number of extra pages that will be required

306 Lock the mm spinlock

307 Expand the VMA

308 Free the mm spinlock

309 Update the statistics for the mm

310-314 If the pages for this region are locked in memory, make them present now

315-316 Return the address of the resized region

can t

324 ret = -ENOMEM;
325 if (flags & MREMAP_MAYMOVE) {
326 if (!(flags & MREMAP_FIXED)) {
327 unsigned long map_flags = 0;
328 if (vma->vm_flags & VM_SHARED)
329 map_flags |= MAP_SHARED;
330
331 new_addr = get_unmapped_area(vma->vm_file, 0,
332 new_len, vma->vm_pgoff, map_flags);
333 ret = new_addr;
334 if (new_addr & ~PAGE_MASK)
335 goto out;
336 }
337 return ret;
338 }
5.3.6. Remapping and moving a memory region

324 The default action is to return saying no memory is available
325 Check to make sure the region is allowed to move
326 If MREMAP_FIXED is not specified, it means the new location was not supplied so one
must be found
328-329 Preserve the MAP_SHARED option
331 Find an unmapped region of memory large enough for the expansion
332 The return value is the address of the new region
333-334 For the returned address to be not page aligned, get_unmapped_area() would
need to be broken. This could possibly be the case with a buggy device driver imple-
menting get_unmapped_area() incorrectly
336 Call move_vma to move the region
338-339 Return the address if successful and the error code otherwise

Function: move_vma (mm/mremap.c)

This function is responsible for moving all the page table entries from one VMA to
another region. If necessary a new VMA will be allocated for the region being moved to.
Just like the function above, it is very long but may be broken up into the following distinct
parts.

- Function preamble, find the VMA preceding the area about to be moved to and the
VMA in front of the region to be mapped
- Handle the case where the new location is between two existing VMAs. See if the
preceding region can be expanded forward or the next region expanded backwards to
cover the new mapped region
- Handle the case where the new location is going to be the last VMA on the list. See
if the preceding region can be expanded forward
- If a region could not be expanded, allocate a new VMA from the slab allocator
5.3.6. Remapping and moving a memory region

- Call `move_page_tables()`, fill in the new VMA details if a new one was allocated and update statistics before returning.

```c
static inline unsigned long move_vma(struct vm_area_struct * vma,
    unsigned long addr, unsigned long old_len, unsigned long new_len,
    unsigned long new_addr)
{
    struct mm_struct * mm = vma->vm_mm;
    struct vm_area_struct * new_vma, * next, * prev;
    int allocated_vma;

    new_vma = NULL;
    next = find_vma_prev(mm, new_addr, &prev);

    if (next) {
        if (prev && prev->vm_end == new_addr &&
            can_vma_merge(prev, vma->vm_flags) &&
            !vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
            spin_lock(&mm->page_table_lock);
            prev->vm_end = new_addr + new_len;
            spin_unlock(&mm->page_table_lock);
            new_vma = prev;

            if (next != prev->vm_next)
                BUG();

            if (prev->vm_end == next->vm_start &&
                can_vma_merge(next, prev->vm_flags)) {
                spin_lock(&mm->page_table_lock);
                prev->vm_end = next->vm_end;
                spin_unlock(&mm->page_table_lock);

                __vma_unlink(mm, next, prev);
            }
        } else if (next->vm_start == new_addr + new_len &&
```

The parameters are

- `vma` The VMA that the address been moved belongs to
- `addr` The starting address of the moving region
- `old_len` The old length of the region to move
- `new_len` The new length of the region moved
- `new_addr` The new address to relocate to

Find the VMA preceding the address been moved to indicated by `prev` and return the region after the new mapping as `next`.

```c
    mm->map_count--;
    kmem_cache_free(vm_area_cachep, next);
}
```
5.3.6. Remapping and moving a memory region

In this block, the new location is between two existing VMAs. Checks are made to see can be preceding region be expanded to cover the new mapping and then if it can be expanded to cover the next VMA as well. If it cannot be expanded, the next region is checked to see if it can be expanded backwards.

136-137 If the preceding region touches the address to be mapped to and may be merged then enter this block which will attempt to expand regions

138 Lock the mm

139 Expand the preceding region to cover the new location

140 Unlock the mm

141 The new vma is now the preceding VMA which was just expanded

142-143 Unnecessary check to make sure the VMA linked list is intact. It is unclear how this situation could possibly occur

144 Check if the region can be expanded forward to encompass the next region

145 If it can, then lock the mm

146 Expand the VMA further to cover the next VMA

147 There is now an extra VMA so unlink it

148 Unlock the mm

149 There is one less mapping now so update the map_count

150 Free the memory used by the memory mapping

153 Else the prev region could not be expanded forward so check if the region pointed to be next may be expanded backwards to cover the new mapping instead

155 If it can, lock the mm

156 Expand the mapping backwards

157 Unlock the mm

158 The VMA representing the new mapping is now next
prev = find_vma(mm, new_addr-1);
if (prev && prev->vm_end == new_addr &&
can_vma_merge(prev, vma->vm_flags) && !vma->vm_file &&
!(vma->vm_flags & VM_SHARED)) {
    spin_lock(&mm->page_table_lock);
    prev->vm_end = new_addr + new_len;
    spin_unlock(&mm->page_table_lock);
    new_vma = prev;
}

This block is for the case where the newly mapped region is the last VMA (next is NULL) so a check is made to see can the preceding region be expanded.

161 Get the previously mapped region
162-163 Check if the regions may be mapped
164 Lock the mm
165 Expand the preceding region to cover the new mapping
166 Lock the mm
167 The VMA representing the new mapping is now prev

allocated_vma = 0;
if (!new_vma) {
    new_vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!new_vma)
        goto out;
    allocated_vma = 1;
}

171 Set a flag indicating if a new VMA was not allocated
172 If a VMA has not been expanded to cover the new mapping then...
173 Allocate a new VMA from the slab allocator
174-175 If it could not be allocated, goto out to return failure
176 Set the flag indicated a new VMA was allocated

if (!move_page_tables(current->mm, new_addr, addr, old_len)) {
    if (allocated_vma) {
        *new_vma = *vma;
5.3.6. Remapping and moving a memory region

new_vma->vm_start = new_addr;
new_vma->vm_end = new_addr+new_len;
new_vma->vm_pgoff +=
    (addr - vma->vm_start) >> PAGE_SHIFT;
new_vma->vm_raend = 0;
if (new_vma->vm_file)
    get_file(new_vma->vm_file);
if (new_vma->vm_ops && new_vma->vm_ops->open)
    new_vma->vm_ops->open(new_vma);
insert_vm_struct(current->mm, new_vma);
}
do_munmap(current->mm, addr, old_len);
current->mm->total_vm += new_len >> PAGE_SHIFT;
if (new_vma->vm_flags & VM_LOCKED) {
    current->mm->locked_vm += new_len >> PAGE_SHIFT;
    make_pages_present(new_vma->vm_start,
                        new_vma->vm_end);
}
return new_addr;
if (allocated_vma)
    kmem_cache_free(vm_area_cachep, new_vma);
out:
    return -ENOMEM;

move_page_tables() is responsible for copying all the page table entries. It returns 0 on success.

180-191 If a new VMA was allocated, fill in all the relevant details, including the file/device entries and insert it into the various VMA linked lists with insert_vm_struct().

192 Unmap the old region as it is no longer required.

193 Update the total_vm size for this process. The size of the old region is not important as it is handled within do_munmap().

194-198 If the VMA has the VM_LOCKED flag, all the pages within the region are made present with mark_pages_present().

199 Return the address of the new region.

201-202 This is the error path. If a VMA was allocated, delete it.

204 Return an out of memory error.
Function: move_page_tables (mm/mremap.c)

This function is responsible for copying all the page table entries from the region pointed to by old_addr to new_addr. It works by literally copying page table entries one at a time. When it is finished, it deletes all the entries from the old area. This is not the most efficient way to perform the operation, but it is very easy to error recover.

```c
static int move_page_tables(struct mm_struct * mm,
unsigned long new_addr, unsigned long old_addr, unsigned long len)
{
    unsigned long offset = len;
    flush_cache_range(mm, old_addr, old_addr + len);
    while (offset) {
        offset -= PAGE_SIZE;
        if (move_one_page(mm, old_addr + offset, new_addr + offset))
            goto oops_we_failed;
    }
    flush_tlb_range(mm, old_addr, old_addr + len);
    return 0;
}

oops_we_failed:
    flush_cache_range(mm, new_addr, new_addr + len);
    while ((offset += PAGE_SIZE) < len)
        move_one_page(mm, new_addr + offset, old_addr + offset);
    zap_page_range(mm, new_addr, len);
    return -1;
```
5.3.6. Remapping and moving a memory region

The parameters are the mm for the process, the new location, the old location and the length of the region to move entries for

flush_cache_range() will flush all CPU caches for this range. It must be called first as some architectures, notably Sparc's require that a virtual to physical mapping exist before flushing the TLB

This loops through each page in the region and calls move_one_page() to move the PTE. This translates to a lot of page table walking and could be performed much better but it is a rare operation

Flush the TLB for the old region

Return success

This block moves all the PTEs back. A flush_tlb_range() is not necessary as there is no way the region could have been used yet so no TLB entries should exist

Zap any pages that were allocated for the move

Return failure

Function: move_one_page (mm/mremap.c)

This function is responsible for acquiring the spinlock before finding the correct PTE with get_one_pTE() and copying it with copy_one_pTE()

Acquire the mm lock

Call get_one_pTE() which walks the page tables to get the correct PTE

If the PTE exists, allocate a PTE for the destination and call copy_one_pTE() to copy the PTEs

Release the lock

Return whatever copy_one_pTE() returned
5.3.6. Remapping and moving a memory region

**Function: get_one_pte (mm/mremap.c)**

This is a very simple page table walk.

```c
18 static inline pte_t *get_one_pte(struct mm_struct *mm, unsigned long addr) {
19     pgd_t * pgd;
20     pmd_t * pmd;
21     pte_t * pte = NULL;
22
23     pgd = pgd_offset(mm, addr);
24     if (pgd_none(*pgd))
25         goto end;
26     if (pgd_bad(*pgd)) {
27         pgd_ERROR(*pgd);
28         pgd_clear(pgd);
29         goto end;
30     }
31
32     pmd = pmd_offset(pgd, addr);
33     if (pmd_none(*pmd))
34         goto end;
35     if (pmd_bad(*pmd)) {
36         pmd_ERROR(*pmd);
37         pmd_clear(pmd);
38         goto end;
39     }
40 }
41
42     pte = pte_offset(pmd, addr);
43     if (pte_none(*pte))
44         pte = NULL;
45 end:
46     return pte;
47 }
```

24 Get the PGD for this address

25-26 If no PGD exists, return NULL as no PTE will exist either

27-31 If the PGD is bad, mark that an error occurred in the region, clear its contents and return NULL

33-40 Acquire the correct PMD in the same fashion as for the PGD

42 Acquire the PTE so it may be returned if it exists
Function: alloc_one_pte (mm/mremap.c)

Trivial function to allocate what is necessary for one PTE in a region.

49 static inline pte_t *alloc_one_pte(struct mm_struct *mm,
   unsigned long addr)

50 {
   pmd_t * pmd;
   pte_t * pte = NULL;

   pmd = pmd_alloc(mm, pgd_offset(mm, addr), addr);
   if (pmd)
      pte = pte_alloc(mm, pmd, addr);

   return pte;
}

54 If a PMD entry does not exist, allocate it

55-56 If the PMD exists, allocate a PTE entry. The check to make sure it succeeded is
performed later in the function copy_one_pte()

Function: copy_one_pte (mm/mremap.c)

Copies the contents of one PTE to another.

60 static inline int copy_one_pte(struct mm_struct *mm,
   pte_t * src, pte_t * dst)

61 {
   int error = 0;
   pte_t pte;

   if (!pte_none(*src)) {
      pte = ptep_get_and_clear(src);
      if (!dst) {
         /* No dest? We must put it back. */
         dst = src;
         error++;
      }
      set_pte(dst, pte);
   }

   return error;
}

65 If the source PTE does not exist, just return 0 to say the copy was successful

66 Get the PTE and remove it from its old location

67-71 If the dst does not exist, it means the call to alloc_one_pte() failed and the copy
operation has failed and must be aborted

72 Move the PTE to its new location

74 Return an error if one occurred
5.3.7 Locking a Memory Region

**Function:** `sys_mlock (mm/mlock.c)`

This is the system call `mlock()` for locking a region of memory into physical memory. This function simply checks to make sure that process and user limits are not exceeded and that the region to lock is page aligned.

```c
asmlinkage long sys_mlock(unsigned long start, size_t len)
{
    unsigned long locked;
    unsigned long lock_limit;
    int error = -ENOMEM;
    down_write(&current->mm->mmap_sem);
    len = PAGE_ALIGN(len + (start & ~PAGE_MASK));
    start &= PAGE_MASK;
    locked = len >> PAGE_SHIFT;
    locked += current->mm->locked_vm;
    lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
    lock_limit >>= PAGE_SHIFT;
    /* check against resource limits */
    if (locked > lock_limit)
        goto out;
    /* we may lock at most half of physical memory... */
    /* (this check is pretty bogus, but doesn’t hurt) */
    if (locked > num_physpages/2)
        goto out;
    error = do_mlock(start, len, 1);
    out:
    up_write(&current->mm->mmap_sem);
    return error;
}
```

201 Take the semaphore, we are likely to sleep during this so a spinlock can not be used

202 Round the length up to the page boundary

203 Round the start address down to the page boundary

205 Calculate how many pages will be locked

206 Calculate how many pages will be locked in total by this process
5.3.7. Locking a Memory Region

208-209 Calculate what the limit is to the number of locked pages

212-213 Do not allow the process to lock more than it should

217-218 Do not allow the process to map more than half of physical memory

220 Call `do_mlock()` which starts the “real” work by find the VMA closetest to the area to lock before calling `mlock_fixup()`

222 Free the semaphore

223 Return the error or success code from `do_mmap()`

**Function: sys_mlockall (mm/mlock.c)**

This is the system call `mlockall()` which attempts to lock all pages in the calling process in memory. If `MCL_CURRENT` is specified, all current pages will be locked. If `MCL_FUTURE` is specified, all future mappings will be locked. The flags may be or-ed together.

```
static int do_mlockall(int flags)
{
    int error;
    unsigned int def_flags;
    struct vm_area_struct * vma;
    if (!capable(CAP_IPC_LOCK))
        return -EPERM;
    def_flags = 0;
    if (flags & MCL_FUTURE)
        def_flags = VM_LOCKED;
    current->mm->def_flags = def_flags;
    error = 0;
    for (vma = current->mm->mmap; vma ; vma = vma->vm_next) {
        unsigned int newflags;
        newflags = vma->vm_flags | VM_LOCKED;
        if (!(flags & MCL_CURRENT))
            newflags &= ~VM_LOCKED;
        error = mlock_fixup(vma, vma->vm_start, vma->vm_end, newflags);
        if (error)
            break;
    }
    return error;
}
```

244-245 The calling process must be either root or have CAP_IPC_LOCK capabilities
The MCL_FUTURE flag says that all future pages should be locked so if set, the def_flags for VMAs should be VM_LOCKED.

Cycle through all VMAs.

Set the VM_LOCKED flag in the current VMA flags.

If the MCL_CURRENT flag has not been set requesting that all current pages be locked, then clear the VM_LOCKED flag. The logic is arranged like this so that the unlock code can use this same function just with no flags.

Call mlock_fixup() which will adjust the regions as necessary.

If a non-zero value is returned at any point, stop locking. It is interesting to note that VMAs already locked will not be unlocked.

Return the success or error value.

**Function: do_mlock (mm/mlock.c)**

This function is responsible for starting the work needed to either lock or unlock a region depending on the value of the on parameter. It is broken up into two sections. The first makes sure the region is page aligned (despite the fact the only two callers of this function do the same thing) before finding the VMA that is to be adjusted. The second part then sets the appropriate flags before calling mlock_fixup() for each VMA that is affected by this locking.

```c
static int do_mlock(unsigned long start, size_t len, int on) {
    unsigned long nstart, end, tmp;
    struct vm_area_struct *vma, *next;
    int error;
    
    if (on && !capable(CAP_IPC_LOCK))
        return -EPERM;
    len = PAGE_ALIGN(len);
    end = start + len;
    if (end < start)
        return -EINVAL;
    if (end == start)
        return 0;
    vma = find_vma(current->mm, start);
    if (!vma || vma->vm_start > start)
        return -ENOMEM;
    
    Page align the request and find the VMA
    
    Only root processes can lock pages
```
5.3.7. Locking a Memory Region

Page align the length despite it being already done in the calling function. This is probably an oversight.

Calculate the end of the locking and make sure it is a valid region. Return EINVAL if it is not.

if locking a region of size 0, just return.

Find the VMA that will be affected by this locking.

If the VMA for this address range does not exist, return -ENOMEM.

for (nstart = start ; ; ) {
    unsigned int newflags;

    newflags = vma->vm_flags | VM_LOCKED;
    if (!on)
        newflags &= ~VM_LOCKED;

    if (vma->vm_end >= end) {
        error = mlock_fixup(vma, nstart, end, newflags);
        break;
    }

    tmp = vma->vm_end;
    next = vma->vm_next;
    error = mlock_fixup(vma, nstart, tmp, newflags);
    if (error)
        break;

    nstart = tmp;
    vma = next;
    if (!vma || vma->vm_start != nstart) {
        error = -ENOMEM;
        break;
    }
}

return error;

Walk through the VMAs affected by this locking and call mlock_fixup() for each of them.

Cycle through as many VMAs as necessary to lock the pages.

Set the VM_LOCKED flag on the VMA.
172-173 Unless this is an unlock in which case, remove the flag

175-177 If this VMA is the last VMA to be affected by the unlocking, call mlock_fixup() with the end address for the locking and exit

180-190 Else this is whole VMA needs to be locked so call mlock_fixup() with the end of this VMA as a parameter rather than the end of the actual locking

180 tmp is the end of the mapping on this VMA

181 next is the next VMA that will be affected by the locking

182 Call mlock_fixup() for this VMA

183-184 If an error occurs, back out. Note that the VMAs already locked are not fixed up right

185 The next start address is the start of the next VMA

186 Move to the next VMA

187-190 If there is no VMA, return -ENOMEM. The next condition though would require the regions to be extremly broken as a result of mlock_fixup() or have overlapping VMAs

192 Return the error or success value

5.3.8 Unlocking the region

Function: sys_munlock (mm/mlock.c)

Page align the request before calling do_mlock() which begins the real work of fixing up the regions.

226 asmlinkage long sys_munlock(unsigned long start, size_t len)
227 {
228   int ret;
229
230   down_write(&current->mm->mmap_sem);
231   len = PAGE_ALIGN(len + (start & ~PAGE_MASK));
232   start &= PAGE_MASK;
233   ret = do_mlock(start, len, 0);
234   up_write(&current->mm->mmap_sem);
235   return ret;
236 }

230 Acquire the semaphore protecting the mm_struct

231 Round the length of the region up to the nearest page boundary

232 Round the start of the region down to the nearest page boundary
5.3.9 Fixing up regions after locking/unlocking

233 Call do_mlock() to fix up the regions

234 Release the semaphore

235 Return the success or failure code

Function: sys_munlockall (mm/mlock.c)

Trivial function. If the flags to mlockall are 0 it gets translated as none of the current
pages must be present and no future mappings should be locked either which means the
VM_LOCKED flag will be removed on all VMAs.

293 asmlinkage long sys_munlockall(void)
294 {
295    int ret;
296
297    down_write(&current->mm->mmap_sem);
298    ret = do_mlockall(0);
299    up_write(&current->mm->mmap_sem);
300    return ret;
301 }

297 Acquire the semaphore protecting the mm_struct

298 Call do_mlockall() with 0 as flags which will remove the VM_LOCKED from all VMAs

299 Release the semaphore

300 Return the error or success code

5.3.9 Fixing up regions after locking/unlocking

Function: mlock_fixup (mm/mlock.c)

This function identifies four separate types of locking that must be addressed. There
first is where the full VMA is to be locked where it calls mlock_fixup_all(). The second is
where only the beginning portion of the VMA is affected, handled by mlock_fixup_start().
The third is the locking of a region at the end handled by mlock_fixup_end() and the last
is locking a region in the middle of the VMA with mlock_fixup_middle().

117 static int mlock_fixup(struct vm_area_struct * vma,
118    unsigned long start, unsigned long end, unsigned int newflags)
119 {
120    int pages, retval;
121
122    if (newflags == vma->vm_flags)
123        return 0;
124
125    if (start == vma->vm_start) {
126        if (end == vma->vm_end)
5.3.9. Fixing up regions after locking/unlocking

If no change is to be made, just return

If the start of the locking is at the start of the VMA, it means that either the full
region is to the locked or only a portion at the beginning

The full VMA is been locked, call \texttt{mlock\_fixup\_all()}

Only a portion is to be locked, call \texttt{mlock\_fixup\_start()}

Else either the a region at the end is to be locked or a region in the middle

The end of the locking match the end of the VMA, call \texttt{mlock\_fixup\_end()}

A region in the middle is to be locked, call \texttt{mlock\_fixup\_middle()}

The fixup functions return 0 on success. If the fixup of the regions succeed and
the regions are now marked as locked, call \texttt{make\_pages\_present()} which makes some
basic checks before calling \texttt{get\_user\_pages()} which faults in all the pages in the same
way the page fault handler does

\textbf{Function: mlock\_fixup\_all (mm/mlock.c)}

\begin{verbatim}
static inline int mlock_fixup_all(struct vm_area_struct * vma, int newflags)
    { spin_lock(&vma->vm_mm->page_table_lock);

    retval = mlock_fixup_all(vma, newflags);

    if (end == vma->vm_end)
        retval = mlock_fixup_end(vma, start, newflags);
    else
        retval = mlock_fixup_middle(vma, start, end, newflags);

    if (!retval) {
        /* keep track of amount of locked VM */
        pages = (end - start) >> PAGE_SHIFT;
        if (newflags & VM_LOCKED) {
            pages = -pages;
            make_pages_present(start, end);
        }
        vma->vm_mm->locked_vm -= pages;
    }

    return retval;

}
\end{verbatim}
### Function: mlock_fixup_start (mm/mlock.c)

Slightly more complicated. A new VMA is required to represent the affected region.

The start of the old VMA is moved forward

```c
static inline int mlock_fixup_start(struct vm_area_struct * vma, unsigned long end, int newflags) {
    struct vm_area_struct * n;
    n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!n) return -EAGAIN;
    *n = *vma;
    n->vm_end = end;
    n->vm_flags = newflags;
    n->vm_raend = 0;
    if (n->vm_file)
        get_file(n->vm_file);
    if (n->vm_ops && n->vm_ops->open)
        n->vm_ops->open(n);
    vma->vm_pgoff += (end - vma->vm_start) >> PAGE_SHIFT;
    lock_vma_mappings(vma);
    spin_lock(&vma->vm_mm->page_table_lock);
    vma->vm_start = end;
    __insert_vm_struct(current->mm, n);
    spin_unlock(&vma->vm_mm->page_table_lock);
    unlock_vma_mappings(vma);
    return 0;
}
```

28 Alloc a VMA from the slab allocator for the affected region

31-34 Copy in the necessary information

35-36 If the VMA has a file or device mapping, get_file() will increment the reference count

37-38 If an open() function is provided, call it
5.3.9. Fixing up regions after locking/unlocking

39 Update the offset within the file or device mapping for the old VMA to be the end of the locked region

40 lock_vma_mappings() will lock any files if this VMA is a shared region

41-44 Lock the parent mm_struct, update its start to be the end of the affected region, insert the new VMA into the processes linked lists (See Section 5.3.4) and release the lock

45 Unlock the file mappings with unlock_vma_mappings()

46 Return success

**Function: mlock_fixup_end (mm/mlock.c)**

Essentially the same as mlock_fixup_start() except the affected region is at the end of the VMA.

49 static inline int mlock_fixup_end(struct vm_area_struct * vma,
50 unsigned long start, int newflags)
51 {
52 struct vm_area_struct * n;
53 n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
54 if (!n)
55     return -EAGAIN;
56 *n = *vma;
57 n->vm_start = start;
58 n->vm_pgoff += (n->vm_start - vma->vm_start) >> PAGE_SHIFT;
59 n->vm_flags = newflags;
60 n->vm_raend = 0;
61 if (n->vm_file)
62     get_file(n->vm_file);
63 if (n->vm_ops && n->vm_ops->open)
64     n->vm_ops->open(n);
65 lock_vma_mappings(vma);
66 spin_lock(&vma->vm_mm->page_table_lock);
67 vma->vm_end = start;
68 __insert_vm_struct(current->mm, n);
69 spin_unlock(&vma->vm_mm->page_table_lock);
70 unlock_vma_mappings(vma);
71 }
72 }
73
54 Alloc a VMA from the slab allocator for the affected region

57-61 Copy in the necessary information and update the offset within the file or device mapping
If the VMA has a file or device mapping, `get_file()` will increment the reference count.

If an `open()` function is provided, call it.

`lock_vma_mappings()` will lock any files if this VMA is a shared region.

Lock the parent `mm_struct`, update its start to be the end of the affected region, insert the new VMA into the processes linked lists (See Section 5.3.4) and release the lock.

Unlock the file mappings with `unlock_vma_mappings()`.

Return success.

**Function: mlock_fixup_middle** (`mm/mlock.c`)

Similar to the previous two fixup functions except that 2 new regions are required to fix up the mapping.

```c
static inline int mlock_fixup_middle(struct vm_area_struct *vma, unsigned long start, unsigned long end, int newflags) {
    struct vm_area_struct *left, *right;
    left = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!left)
        return -EAGAIN;
    right = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!right) {
        kmem_cache_free(vm_area_cachep, left);
        return -EAGAIN;
    }
    *left = *vma;
    *right = *vma;
    left->vm_end = start;
    right->vm_start = end;
    right->vm_pgoff += (right->vm_start - left->vm_start) >> PAGE_SHIFT;
    vma->vm_flags = newflags;
    left->vm_raend = 0;
    right->vm_raend = 0;
    if (vma->vm_file)
        atomic_add(2, &vma->vm_file->f_count);
    if (vma->vm_ops && vma->vm_ops->open) {
        vma->vm_ops->open(left);
        vma->vm_ops->open(right);
    }
}
```
5.3.10. Deleting a memory region

103   vma->vm_raend = 0;
104   vma->vm_pgoff += (start - vma->vm_start) >> PAGE_SHIFT;
105   lock_vma_mappings(vma);
106   spin_lock(&vma->vm_mm->page_table_lock);
107   vma->vm_start = start;
108   vma->vm_end = end;
109   vma->vm_flags = newflags;
110   __insert_vm_struct(current->mm, left);
111   __insert_vm_struct(current->mm, right);
112   spin_unlock(&vma->vm_mm->page_table_lock);
113   unlock_vma_mappings(vma);
114   return 0;
115 }

80-87 Allocate the two new VMAs from the slab allocator
88-89 Copy in the information from the old VMA into them
90 The end of the left region is the start of the region to be affected
91 The start of the right region is the end of the affected region
92 Update the file offset
93 The old VMA is now the affected region so update its flags
94-95 Make the readahead window 0 to ensure pages not belonging to their regions are not accidentally read ahead
96-97 Increment the reference count to the file/device mapping if there is one
99-102 Call the open() function for the two new mappings
103-104 Cancel the readahead window and update the offset within the file to be the beginning of the locked region
105 Lock the shared file/device mappings
106-112 Lock the parent mm_struct, update the VMA and insert the two new regions into the process before releasing the lock again
113 Unlock the shared mappings
114 Return success
5.3.10 Deleting a memory region

Function: do_munmap \((mm/mmap.c)\)

This function is responsible for unmapping a region. If necessary, the unmapping can span multiple VMAs and it can partially unmap one if necessary. Hence the full unmapping operation is divided into two major operations. This function is responsible for finding what VMAs are affected and `unmap_fixup()` is responsible for fixing up the remaining VMAs.

This function is divided up in a number of small sections will be dealt with in turn. The are broadly speaking:

- Function preamble and find the VMA to start working from
- Take all VMAs affected by the unmapping out of the mm and place them on a linked list headed by the variable `free`
- Cycle through the list headed by `free`, unmap all the pages in the region to be unmapped and call `unmap_fixup()` to fix up the mappings
- Validate the mm and free memory associated with the unmapping

```c
int do_munmap(struct mm_struct *mm, unsigned long addr, size_t len) {
    struct vm_area_struct *mpnt, *prev, **npp, *free, *extra;
    if ((addr & ~PAGE_MASK) || addr > TASK_SIZE ||
        len > TASK_SIZE-addr)
        return -EINVAL;
    if ((len = PAGE_ALIGN(len)) == 0)
        return -EINVAL;
    mpnt = find_vma_prev(mm, addr, &prev);
    if (!mpnt)
        return 0;
    /* we have addr < mpnt->vm_end */
```
5.3.10. Deleting a memory region

```c
if (mpnt->vm_start >= addr+len)
    return 0;
if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
    && mm->map_count >= max_map_count)
    return -ENOMEM;
extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!extra)
    return -ENOMEM;
```

The parameters are as follows:

- `mm`: The mm for the processes performing the unmap operation
- `addr`: The starting address of the region to unmap
- `len`: The length of the region

Ensure the address is page aligned and that the area to be unmapped is not in the kernel virtual address space

Make sure the region size to unmap is page aligned

Find the VMA that contains the starting address and the preceding VMA so it can be easily unlinked later

If no mpnt was returned, it means the address must be past the last used VMA so the address space is unused, just return

If the returned VMA starts past the region we are trying to unmap, then the region in unused, just return

The first part of the check sees if the VMA is just been partially unmapped, if it is, another VMA will be created later to deal with a region being broken into so to the `map_count` has to be checked to make sure it is not too large

In case a new mapping is required, it is allocated now as later it will be much more difficult to back out in event of an error

```c
npp = (prev ? &prev->vm_next : &mm->mmap);
free = NULL;
spin_lock(&mm->page_table_lock);
for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
    *npp = mpnt->vm_next;
    mpnt->vm_next = free;
    free = mpnt;
    rb_erase(&mpnt->vm_rb, &mm->mm_rb);
}
```
5.3.10. Deleting a memory region

This section takes all the VMAs affected by the unmapping and places them on a separate linked list headed by a variable called free. This makes the fixup of the regions much easier.

npp becomes the next VMA in the list during the for loop following below. To initialise it, it is either the current VMA (mpnt) or else it becomes the first VMA in the list.

free is the head of a linked list of VMAs that are affected by the unmapping.

Lock the mm

Cycle through the list until the start of the current VMA is past the end of the region to be unmapped.

npp becomes the next VMA in the list.

Remove the current VMA from the linear linked list within the mm and place it on a linked list headed by free. The current mpnt becomes the head of the free linked list.

Delete mpnt from the red-black tree.

Remove the cached result in case the last looked up result is one of the regions to be unmapped.

Free the mm.

```c
mm->mmap_cache = NULL; /* Kill the cache. */
spin_unlock(&mm->page_table_lock);

/* Ok - we have the memory areas we should free on the 'free' list, so release them, and unmap the page range.. */
if (mpnt = free) != NULL {
    unsigned long st, end, size;
    struct file *file = NULL;
    free = free->vm_next;
    st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
    end = addr+len;
    end = end > mpnt->vm_end ? mpnt->vm_end : end;
    size = end - st;
```
5.3.10. Deleting a memory region

if (mpnt->vm_flags & VM_DENYWRITE &&
   (st != mpnt->vm_start || end != mpnt->vm_end) &&
   (file = mpnt->vm_file) != NULL) {
    atomic_dec(&file->f_dentry->d_inode->i_writecount);
}
remove_shared_vm_struct(mpnt);
mm->map_count--;
zap_page_range(mm, st, size);
/*
 * Fix the mapping, and free the old area if it wasn’t reused.
 */
extra = unmap_fixup(mm, mpnt, st, size, extra);
if (file)
    atomic_inc(&file->f_dentry->d_inode->i_writecount);
}

Keep stepping through the list until no VMAs are left

Move free to the next element in the list leaving mpnt as the head about to be removed

st is the start of the region to be unmapped. If the addr is before the start of the VMA, the starting point is mpnt→vm_start, otherwise it is the supplied address

Calculate the end of the region to map in a similar fashion

Calculate the size of the region to be unmapped in this pass

If the VM_DENYWRITE flag is specified, a hole will be created by this unmapping and a file is mapped then the writecount is decremented. When this field is negative, it counts how many users there is protecting this file from being opened for writing

Remove the file mapping. If the file is still partially mapped, it will be acquired again during unmap_fixup()

Reduce the map count

Remove all pages within this region

Call the fixup routing

Increment the writecount to the file as the region has been unmapped. If it was just partially unmapped, this call will simply balance out the decrement at line 987
5.3.10. Deleting a memory region

validate_mm(mm);

/* Release the extra vma struct if it wasn’t used */
if (extra)
    kmem_cache_free(vm_area_cachep, extra);
free_pgtables(mm, prev, addr, addr+len);
return 0;
}

A debugging function only. If enabled, it will ensure the VMA tree for this mm is still valid

If extra VMA was not required, delete it
Free all the page tables that were used for the unmapped region
Return success

Function: unmap_fixup (mm/mmap.c)

This function fixes up the regions after a block has been unmapped. It is passed a list of VMAs that are affected by the unmapping, the region and length to be unmapped and a spare VMA that may be required to fix up the region if a whole is created. There is four principle cases it handles; The unmapping of a region, partial unmapping from the start to somewhere in the middle, partial unmapping from somewhere in the middle to the end and the creation of a hole in the middle of the region. Each case will be taken in turn.

static struct vm_area_struct * unmap_fixup(struct mm_struct *mm,
    struct vm_area_struct *area, unsigned long addr, size_t len,
    struct vm_area_struct *extra)
{
    struct vm_area_struct *mpnt;
    unsigned long end = addr + len;
    area->vm_mm->total_vm -= len >> PAGE_SHIFT;
    if (area->vm_flags & VM_LOCKED)
        area->vm_mm->locked_vm -= len >> PAGE_SHIFT;
    area->vm_mm->total_vm -= len >> PAGE_SHIFT;
    if (area->vm_flags & VM_LOCKED)
        area->vm_mm->locked_vm -= len >> PAGE_SHIFT;

    Function preamble.

The parameters to the function are;

mm is the mm the unmapped region belongs to
area is the head of the linked list of VMAs affected by the unmapping
addr is the starting address of the unmapping
5.3.10. Deleting a memory region

`len` is the length of the region to be unmapped

`extra` is a spare VMA passed in for when a hole in the middle is created

790 Calculate the end address of the region being unmapped

792 Reduce the count of the number of pages used by the process

793-794 If the pages were locked in memory, reduce the locked page count

796-799 /* Unmapping the whole area. */
    if (addr == area->vm_start && end == area->vm_end) {
        if (area->vm_ops && area->vm_ops->close)
            area->vm_ops->close(area);
        if (area->vm_file)
            fput(area->vm_file);
        kmem_cache_free(vm_area_cachep, area);
        return extra;
    }

The first, and easiest, case is where the full region is being unmapped

797 The full region is unmapped if the addr is the start of the VMA and the end is the end of the VMA. This is interesting because if the unmapping is spanning regions, it is possible the end is beyond the end of the VMA but the full of this VMA is still being unmapped

798-799 If a close operation is supplied by the VMA, call it

800-801 If a file or device is mapped, call `fput()` which decrements the usage count and releases it if the count falls to 0

802 Free the memory for the VMA back to the slab allocator

803 Return the extra VMA as it was unused

807 if (end == area->vm_end) {
    /*
     * here area isn’t visible to the semaphore-less readers
     * so we don’t need to update it under the spinlock.
     */
    area->vm_end = addr;
    lock_vma_mappings(area);
    spin_lock(&mm->page_table_lock);
}

Handle the case where the middle of the region to the end is been unmapped

812 Truncate the VMA back to `addr`. At this point, the pages for the region have already freed and the page table entries will be freed later so no further work is required
5.3.10. Deleting a memory region

If a file/device is being mapped, the lock protecting shared access to it is taken in the function `lock_vm_mappings()`.

Lock the mm. Later in the function, the remaining VMA will be reinserted into the mm.

```c
else if (addr == area->vm_start) {
    area->vm_pgoff += (end - area->vm_start) >> PAGE_SHIFT;
    /* same locking considerations of the above case */
    area->vm_start = end;
    lock_vma_mappings(area);
    spin_lock(&mm->page_table_lock);
}
```

Handle the case where the VMA is been unmapped from the start to some part in the middle.

Increase the offset within the file/device mapped by the number of pages this unmapping represents.

Move the start of the VMA to the end of the region being unmapped.

Lock the file/device and mm as above.

```c
else {
    /* Unmapping a hole: area->vm_start < addr <= end < area->vm_end */
    /* Add end mapping -- leave beginning for below */
    mpnt = extra;
    extra = NULL;

    mpnt->vm_mm = area->vm_mm;
    mpnt->vm_start = end;
    mpnt->vm_end = area->vm_end;
    mpnt->vm_page_prot = area->vm_page_prot;
    mpnt->vm_flags = area->vm_flags;
    mpnt->vm_raend = 0;
    mpnt->vm_ops = area->vm_ops;
    mpnt->vm_pgoff = area->vm_pgoff +
        ((end - area->vm_start) >> PAGE_SHIFT);
    mpnt->vm_file = area->vm_file;
    mpnt->vm_private_data = area->vm_private_data;
    if (mpnt->vm_file)
        get_file(mpnt->vm_file);
    if (mpnt->vm_ops && mpnt->vm_ops->open)
        mpnt->vm_ops->open(mpnt);
    area->vm_end = addr; /* Truncate area */
```
Handle the case where a hole is being created by a partial unmapping. In this case, the extra VMA is required to create a new mapping from the end of the unmapped region to the end of the old VMA.

Take the extra VMA and make VMA NULL so that the calling function will know it is in use and cannot be freed.

Copy in all the VMA information.

If a file/device is mapped, get a reference to it with `get_file()`.

If an open function is provided, call it.

Truncate the VMA so that it ends at the start of the region to be unmapped.

Lock the files and mm as with the two previous cases.

Insert the extra VMA into the mm.

Reinsert the VMA into the mm.

Unlock the page tables.

Unlock the spinlock to the shared mapping.

Return the extra VMA if it was not used and NULL if it was.

### 5.3.11 Deleting all memory regions

**Function:** exit_mmap *(mm/mmap.c)*

This function simply steps through all VMAs associated with the supplied mm and unmaps them.
5.3.11. Deleting all memory regions

void exit_mmap(struct mm_struct * mm) {
    struct vm_area_struct * mpnt;

    release_segments(mm);
    spin_lock(&mm->page_table_lock);
    mpnt = mm->mmap;
    mm->mmap = mm->mmap_cache = NULL;
    mm->mm_rb = RB_ROOT;
    mm->rss = 0;
    spin_unlock(&mm->page_table_lock);
    mm->total_vm = 0;
    mm->locked_vm = 0;

    flush_cache_mm(mm);

    while (mpnt) {
        struct vm_area_struct * next = mpnt->vm_next;
        unsigned long start = mpnt->vm_start;
        unsigned long end = mpnt->vm_end;
        unsigned long size = end - start;

        if (mpnt->vm_ops) {
            if (mpnt->vm_ops->close)
                mpnt->vm_ops->close(mpnt);
            mm->map_count--;

            remove_shared_vm_struct(mpnt);
            zap_page_range(mm, start, size);
            if (mpnt->vm_file)
                fput(mpnt->vm_file);
            kmem_cache_free(vm_area_cachep, mpnt);
            mpnt = next;
        }

        flush_tlb_mm(mm);
    }

    /* This is just debugging */
    if (mm->map_count)
        BUG();

    clear_page_tables(mm, FIRST_USER_PGD_NR, USER_PTRS_PER_PGD);
}

release_segments() will release memory segments associated with the process on its
Local Descriptor Table (LDT) if the architecture supports segments and the process
was using them. Some applications, notably WINE use this feature.

Lock the mm
mpnt becomes the first VMA on the list
Clear VMA related information from the mm so it may be unlocked
Unlock the mm
Clear the mm statistics
Flush the CPU for the address range
Step through every VMA that was associated with the mm
Record what the next VMA to clear will be so this one may be deleted
Record the start, end and size of the region to be deleted
If there is a close operation associated with this VMA, call it
Reduce the map count
Remove the file/device mapping from the shared mappings list
Free all pages associated with this region
If a file/device was mapped in this region, free it
Free the VMA struct
Move to the next VMA
Flush the TLB for this whole mm as it is about to be unmapped
If the map_count is positive, it means the map count was not accounted for properly so call BUG() to mark it
Clear the page tables associated with this region

5.4 Page Fault Handler
This function is the x86 architecture dependent function for the handling of page fault exception handlers. Each architecture registers their own but all of them have similar responsibilities.

asmlinkage void do_page_fault(struct pt_regs *regs,
                                       unsigned long error_code)
{
    struct task_struct *tsk;
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    unsigned long address;
    unsigned long page;
Function preamble. Get the fault address and enable interrupts

The parameters are

- **regs** is a struct containing what all the registers at fault time
- **error_code** indicates what sort of fault occurred

As the comment indicates, the cr2 register is the fault address

If the fault is from within an interrupt, enable them

Set the current task
if (address >= TASK_SIZE && !(error_code & 5))
goto vmalloc_fault;

mm = tsk->mm;
info.si_code = SEGV_MAPERR;

if (in_interrupt() || !mm)
goto no_context;

Check for exceptional faults, kernel faults, fault in interrupt and fault with no memory context

If the fault address is over TASK_SIZE, it is within the kernel address space. If the error code is 5, then it means it happened while in kernel mode and is not a protection error so handle a vmalloc fault

Record the working mm

If this is an interrupt, or there is no memory context (such as with a kernel thread), there is no way to safely handle the fault so goto no_context

down_read(&mm->mmap_sem);

vma = find_vma(mm, address);
if (!vma)
goto bad_area;
if (vma->vm_start <= address)
goto good_area;
if (!((vma->vm_flags & VM_GROWSDOWN))
goto bad_area;
if (error_code & 4) {
/*
 * accessing the stack below %esp is always a bug.
 * The "+ 32" is there due to some instructions (like
 * pusha) doing post-decrement on the stack and that
 * doesn’t show up until later..
 */
if (address + 32 < regs->esp)
goto bad_area;
}
if (expand_stack(vma, address))
goto bad_area;

If a fault in userspace, find the VMA for the faulting address and determine if it is a good area, a bad area or if the fault occurred near a region that can be expanded such as the stack
Take the long lived mm semaphore

Find the VMA that is responsible or is closest to the faulting address

If a VMA does not exist at all, goto bad_area

If the start of the region is before the address, it means this VMA is the correct VMA for the fault so goto good_area which will check the permissions

For the region that is closest, check if it can grow down (VM_GROWSDOWN). If it does, it means the stack can probably be expanded. If not, goto bad_area

Check to make sure it isn’t an access below the stack. If the error_code is 4, it means it is running in userspace

expand the stack, if it fails, goto bad_area

good_area:

switch (error_code & 3) {
  default: /* 3: write, present */
  #ifdef TEST_VERIFY_AREA
  if (regs->cs == KERNEL_CS)
    printk("WP fault at %08lx
\n", regs->eip);
  #endif
  /* fall through */
  case 2: /* write, not present */
    if (!((vma->vm_flags & VM_WRITE))
      goto bad_area;
    write++;
  break;
  case 1: /* read, present */
    goto bad_area;
  case 0: /* read, not present */
    if (!((vma->vm_flags & (VM_READ | VM_EXEC)))
      goto bad_area;
}

There is the first part of a good area is handled. The permissions need to be checked in case this is a protection fault.

By default return an error

Check the error code against bits 0 and 1 of the error code. Bit 0 at 0 means page was not present. At 1, it means a protection fault like a write to a read-only area. Bit 1 is 0 if it was a read fault and 1 if a write

If it is 3, both bits are 1 so it is a write protection fault
5.4. Page Fault Handler

221 Bit 1 is a 1 so it is a write fault

222-223 If the region can not be written to, it is a bad write to goto bad_area. If the region can be written to, this is a page that is marked Copy On Write (COW)

224 Flag that a write has occurred

226-227 This is a read and the page is present. There is no reason for the fault so must be some other type of exception like a divide by zero, goto bad_area where it is handled

228-230 A read occurred on a missing page. Make sure it is ok to read or exec this page. If not, goto bad_area. The check for exec is made because the x86 can not exec protect a page and instead uses the read protect flag. This is why both have to be checked

233 survive:
239 switch (handle_mm_fault(mm, vma, address, write)) {
240 case 1:
241 tsk->min_flt++;
242 break;
243 case 2:
244 tsk->maj_flt++;
245 break;
246 case 0:
247 goto do_sigbus;
248 default:
249 goto out_of_memory;
250 }
251 goto out_of_memory;
252 */
253 * Did it hit the DOS screen memory VA from vm86 mode?
254 */
255 if (regs->eflags & VM_MASK) {
256 unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
257 if (bit < 32)
258 tsk->thread.screen_bitmap |= 1 << bit;
259 }
260 up_read(&mm->mmap_sem);
261 return;

At this point, an attempt is going to be made to handle the fault gracefully with handle_mm_fault().

239 Call handle_mm_fault() with the relevant information about the fault. This is the architecture independent part of the handler

240-242 A return of 1 means it was a minor fault. Update statistics

243-245 A return of 2 means it was a major fault. Update statistics
A return of 0 means some IO error happened during the fault so go to the do_sigbus handler.

Any other return means memory could not be allocated for the fault so we are out of memory. In reality this does not happen as another function out_of_memory() is invoked in mm/oom_kill.c before this could happen which is a lot more graceful about who it kills.

Not sure

Release the lock to the mm

Return as the fault has been successfully handled

This is the bad area handler such as using memory with no vm_area_struct managing it. If the fault is not by a user process or the f00f bug, the no_context label is fallen through to.
5.4. Page Fault Handler

271 An error code of 4 implies userspace so it is a simple case of sending a SIGSEGV to kill the process

272-274 Set thread information about what happened which can be read by a debugger later

275 Record that a SIGSEGV signal was sent

276 clear errno

278 Record the address

279 Send the SIGSEGV signal. The process will exit and dump all the relevant information

280 Return as the fault has been successfully handled

286-295 An bug in the first Pentiums was called the f00f bug which caused the processor to constantly page fault. It was used as a local DoS attack on a running Linux system. This bug was trapped within a few hours and a patch released. Now it results in a harmless termination of the process rather than a locked system

296
297 no_context:
298    /* Are we prepared to handle this kernel fault? */
299    if ((fixup = search_exception_table(regs->eip)) != 0) {
300      regs->eip = fixup;
301      return;
302    }

299-302 Check can this exception be handled and if so, call the proper exception handler after returning. This is really important during copy_from_user() and copy_to_user() when an exception handler is especially installed to trap reads and writes to invalid regions in userspace without having to make expensive checks. It means that a small fixup block of code can be called rather than falling through to the next block which causes an oops

303
304 /*
305 * Oops. The kernel tried to access some bad page. We'll have to
306 * terminate things with extreme prejudice.
307 */
308
309    bust_spinlocks(1);
310
311    if (address < PAGE_SIZE)
312      printk(KERN_ALERT "Unable to handle kernel NULL pointer dereference");
313    else
314         printk(KERN_ALERT "Unable to handle kernel paging request");
315         printk(" at virtual address  \%08lx\n",address);
316         printk(" printing eip:  \n");
317         printk("\%08lx\n", regs->eip);
318         asm("movl  \%cr3,\%0":"=r" (page));
319         page = ((unsigned long *) __va(page))[address >> 22];
320         printk(KERN_ALERT "*pde = \%08lx\n", page);
321         if (page & 1) {
322             page &= PAGE_MASK;
323             address &= 0x003ff000;
324             page = ((unsigned long *)
325             __va(page))[address >> PAGE_SHIFT];
326             printk(KERN_ALERT "*pte = \%08lx\n", page);
327         }
328         die("Oops", regs, error_code);
329         bust_spinlocks(0);
330         do_exit(SIGKILL);

This is the no_context handler. Some bad exception occurred which is going to end up in the process been terminated in all likeliness. Otherwise the kernel faulted when it definitely should have and an OOPS report is generated.

309-329 Otherwise the kernel faulted when it really shouldn’t have and it is a kernel bug. This block generates an oops report

309 Forcibly free spinlocks which might prevent a message getting to console

311-312 If the address is < PAGE_SIZE, it means that a null pointer was used. Linux deliberately has page 0 unassigned to trap this type of fault which is a common programming error

313-314 Otherwise it is just some bad kernel error such as a driver trying to access userspace incorrectly

315-320 Print out information about the fault

321-326 Print out information about the page been faulted

327 Die and generate an oops report which can be used later to get a stack trace so a developer can see more accurately where and how the fault occurred

329 Forcibly kill the faulting process

335 out_of_memory:
336         if (tsk->pid == 1) {
337             yield();
338             goto survive;
The out of memory handler. Usually ends with the faulting process getting killed unless it is init

336-339 If the process is init, just yield and goto survive which will try to handle the fault gracefully. init should never be killed

340 Free the mm semaphore

341 Print out a helpful “You are Dead” message

342 If from userspace, just kill the process

344 If in kernel space, go to the no_context handler which in this case will probably result in a kernel oops

345
346 do_sigbus:
347 up_read(&mm->mmap_sem);
348
tsk->thread.cr2 = address;
tsk->thread.error_code = error_code;
tsk->thread.trap_no = 14;
info.si_signo = SIGBUS;
info.si_errno = 0;
info.si_code = BUS_ADRERR;
info.si_addr = (void *)address;
force_sig_info(SIGBUS, &info, tsk);

362 /* Kernel mode? Handle exceptions or die */
363 if (!(error_code & 4))
364 goto no_context;
365 return;

347 Free the mm lock

353-359 Fill in information to show a SIGBUS occurred at the faulting address so that a debugger can trap it later

360 Send the signal

363-364 If in kernel mode, try and handle the exception during no_context
If in userspace, just return and the process will die in due course.

```c
int offset = _-_pgd_offset(address);
pgd_t *pgd, *pgd_k;
pmd_t *pmd, *pmd_k;
pte_t *pte_k;

asm("movl %%cr3,%0":"=r" (pgd));
pgd = offset + (pgd_t *)__va(pgd);
pgd_k = init_mm.pgd + offset;

if (!pgd_present(*pgd_k))
goto no_context;
set_pgd(pgd, *pgd_k);

pmd = pmd_offset(pgd, address);
pmd_k = pmd_offset(pgd_k, address);
if (!pmd_present(*pmd_k))
goto no_context;
set_pmd(pmd, *pmd_k);

pte_k = pte_offset(pmd_k, address);
if (!pte_present(*pte_k))
goto no_context;
return;
```

This is the vmalloc fault handler. In this case the process page table needs to be synchronized with the reference page table. This could occur if a global TLB flush flushed some kernel page tables as well and the page table information just needs to be copied back in.

Get the offset within a PGD

Copy the address of the PGD for the process from the cr3 register to pgd

Calculate the pgd pointer from the process PGD

Calculate for the kernel reference PGD

If the pgd entry is invalid for the kernel page table, goto no_context

Set the page table entry in the process page table with a copy from the kernel reference page table
5.4.1 Handling the Page Fault

This is the top level pair of functions for the architecture independent page fault handler.

Function: handle_mm_fault (mm/memory.c)

This function allocates the PMD and PTE necessary for this new PTE hat is about to be allocated. It takes the necessary locks to protect the page tables before calling handle_ pte_fault () to fault in the page itself.

```
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma, 
                   unsigned long address, int write_access) 
{
    pgd_t *pgd;
    pmd_t *pmd;

    current->state = TASK_RUNNING;
    pgd = pgd_offset(mm, address);

    /*
     * We need the page table lock to synchronize with kswapd
     * and the SMP-safe atomic PTE updates.
     */
    spin_lock(&mm->page_table_lock);
    pmd = pmd_alloc(mm, pgd, address);

    if (pmd) {
        pte_t * pte = pte_alloc(mm, pmd, address);
        if (pte)
            return handle_pfeault(mm, vma, address, 
                                   write_access, pte);
    }
    spin_unlock(&mm->page_table_lock);
    return -1;
}
```

The parameters of the function are;

389-393 Same idea for the PMD. Copy the page table entry from the kernel reference page table to the process page tables

395 Check the PTE

396-397 If it is not present, it means the page was not valid even in the kernel reference page table so goto no_context to handle what is probably a kernel bug, probably a reference to a random part of unused kernel space

398 Otherwise return knowing the process page tables have been updated and are in sync with the kernel page tables
5.4.1. Handling the Page Fault

mm is the mm_struct for the faulting process
vma is the vm_area_struct managing the region the fault occurred in
address is the faulting address
write_access is 1 if the fault is a write fault

1370 Set the current state of the process
1371 Get the pgd entry from the top level page table
1377 Lock the mm_struct as the page tables will change
1378 pmd_alloc will allocate a pmd_t if one does not already exist
1380 If the pmd has been successfully allocated then...
1381 Allocate a PTE for this address if one does not already exist
1382-1383 Handle the page fault with handle_pte_fault() and return the status code
1385 Failure path, unlock the mm_struct
1386 Return -1 which will be interpreted as an out of memory condition which is correct
as this line is only reached if a PMD or PTE could not be allocated

Function: handle_pte_fault (mm/memory.c)
This function decides what type of fault this is and which function should handle it.
do_no_page() is called if this is the first time a page is to be allocated. do_swap_page() handles the case where the page was swapped out to disk. do_wp_page() breaks COW pages. If none of them are appropriate, the PTE entry is simply updated. If it was written to, it is marked dirty and it is marked accessed to show it is a young page.

1331 static inline int handle_pte_fault(struct mm_struct *mm,
1332 struct vm_area_struct * vma, unsigned long address,
1333 int write_access, pte_t * pte)
1334 {
1335 pte_t entry;
1336
1337 entry = *pte;
1338 if (!pte_present(entry)) {
1339 /*
1340 * If it truly wasn’t present, we know that kswapd
1341 * and the PTE updates will not touch it later. So
1342 * drop the lock.
1343 */
1344 if (pte_none(entry))
1345 return do_no_page(mm, vma, address,
1346 write_access, pte);
1346 return do_swap_page(mm, vma, address, pte, entry,
### 5.4.2 Demand Allocation

The parameters of the function are the same as those for `handle_mm_fault()` except the PTE for the fault is included.

• **Record the PTE**
• **Handle the case where the PTE is not present**
• **If the PTE has never been filled, handle the allocation of the PTE with `do_no_page()`**
• **If the page has been swapped out to backing storage, handle it with `do_swap_page()`**
• **Handle the case where the page is been written to**
• **If the PTE is marked write-only, it is a COW page so handle it with `do_wp_page()`**
• **Otherwise just simply mark the page as dirty**
• **Mark the page as accessed**

`establish_pte()` copies the PTE and then updates the TLB and MMU cache. This does not copy in a new PTE but some architectures require the TLB and MMU update.

**Unlock the mm_struct and return that a minor fault occurred**

#### 5.4.2 Demand Allocation

**Function: do_no_page** *(mm/memory.c)*

This function is called the first time a page is referenced so that it may be allocated and filled with data if necessary. If it is an anonymous page, determined by the lack of a `vm_ops` available to the VMA or the lack of a `nopage()` function, then `do_anonymous_page()` is called. Otherwise the supplied `nopage()` function is called to allocate a page and it is inserted into the page tables here. The function has the following tasks:
5.4.2. Demand Allocation

- Check if `do_anonymous_page()` should be used and if so, call it and return the page it allocates. If not, call the supplied `nopage()` function and ensure it allocates a page successfully.
- Break COW early if appropriate
- Add the page to the page table entries and call the appropriate architecture dependent hooks

```c
1245 static int do_no_page(struct mm_struct * mm, struct vm_area_struct * vma,
1246 unsigned long address, int write_access, pte_t *page_table)
1247 {
1248 struct page * new_page;
1249 pte_t entry;
1250 if (!vma->vm_ops || !vma->vm_ops->nopage)
1251 return do_anonymous_page(mm, vma, page_table,
1252 write_access, address);
1253 spin_unlock(&mm->page_table_lock);
1254 new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
1256 if (new_page == NULL) /* no page was available -- SIGBUS */
1257 return 0;
1259 if (new_page == NOPAGE_OOM)
1260 return -1;
```

The parameters supplied are the same as those for `handle_pte_fault()`

1251-1252 If no `vm_ops` is supplied or no `nopage()` function is supplied, then call `do_anonymous_page()` to allocate a page and return it
Otherwise free the page table lock as the \texttt{nopage()} function can not be called with spinlocks held.

Call the supplied nopage function, in the case of filesystems, this is frequently \texttt{filemap\_nopage()} but will be different for each device driver.

If NULL is returned, it means some error occurred in the nopage function such as an IO error while reading from disk. In this case, 0 is returned which results in a SIGBUS being sent to the faulting process.

If \texttt{NOPAGE\_OOM} is returned, the physical page allocator failed to allocate a page and -1 is returned which will forcibly kill the process.

Break COW early in this block if appropriate. COW is broken if the fault is a write fault and the region is not shared with VM\_SHARED. If COW was not broken in this case, a second fault would occur immediately upon return.

Check if COW should be broken early.

If so, allocate a new page for the process.

If the page could not be allocated, reduce the reference count to the page returned by the \texttt{nopage()} function and return -1 for out of memory.

Otherwise copy the contents.

Reduce the reference count to the returned page which may still be in use by another process.

Add the new page to the LRU lists so it may be reclaimed by kswapd later.
5.4.2. Demand Allocation

1292    flush_icache_page(vma, new_page);
1293    entry = mk_pte(new_page, vma->vm_page_prot);
1294    if (write_access)
1295        entry = pte_mkwrite(pte_mkdirty(entry));
1296    set_pte(page_table, entry);
1297 } else {
1298    /* One of our sibling threads was faster, back out. */
1299    page_cache_release(new_page);
1300    spin_unlock(&mm->page_table_lock);
1301    return 1;
1302 }
1303
1304    /* no need to invalidate: a not-present page shouldn’t be cached */
1305    update_mmu_cache(vma, address, entry);
1306    spin_unlock(&mm->page_table_lock);
1307    return 2;    /* Major fault */
1308 }

1277 Lock the page tables again as the allocations have finished and the page tables are about to be updated

1289 Check if there is still no PTE in the entry we are about to use. If two faults hit here at the same time, it is possible another processor has already completed the page fault and this one should be backed out

1290-1297 If there is no PTE entered, complete the fault

1290 Increase the RSS count as the process is now using another page

1291 As the page is about to be mapped to the process space, it is possible for some architectures that writes to the page in kernel space will not be visible to the process. flush_page_to_ram() ensures the cache will be coherent

1292 flush_icache_page() is similar in principle except it ensures the icache and dcache’s are coherent

1293 Create a pte_t with the appropriate permissions

1294-1295 If this is a write, then make sure the PTE has write permissions

1296 Place the new PTE in the process page tables

1297-1302 If the PTE is already filled, the page acquired from the nopage() function must be released

1299 Decrement the reference count to the page. If it drops to 0, it will be freed
1300-1301 Release the mm_struct lock and return 1 to signal this is a minor page fault as no major work had to be done for this fault as it was all done by the winner of the race

1305 Update the MMU cache for architectures that require it

1306-1307 Release the mm_struct lock and return 2 to signal this is a major page fault

Function: do_anonymous_page (mm/memory.c)
This function allocates a new page for a process accessing a page for the first time. If it is a read access, a system wide page containing only zeros is mapped into the process. If it is write, a zero filled page is allocated and placed within the page tables

1190 static int do_anonymous_page(struct mm_struct * mm,
struct vm_area_struct * vma,
pte_t *page_table, int write_access,
unsigned long addr)
1191 {
    pte_t entry;

    /* Read-only mapping of ZERO_PAGE. */
    entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));

    /* ..except if it’s a write access */
    if (write_access) {
        struct page *page;
        /* Allocate our own private page. */
        spin_unlock(&mm->page_table_lock);

        page = alloc_page(GFP_HIGHUSER);
        if (!page)
            goto no_mem;
        clear_user_highpage(page, addr);
        spin_lock(&mm->page_table_lock);
        if (!pte_none(*page_table)) {
            page_cache_release(page);
            spin_unlock(&mm->page_table_lock);
            return 1;
        }

        mm->rss++;
        flush_page_to_ram(page);
        entry = pte_mkdirty(mk_pte(page, vma->vm_page_prot));
        lru_cache_add(page);
        mark_page_accessed(page);
    }
5.4.3. Demand Paging

```c
set_pte(page_table, entry);

/* No need to invalidate - it was non-present before */
update_mmu_cache(vma, addr, entry);
spin_unlock(&mm->page_table_lock);
return 1; /* Minor fault */

no_mem:
return -1;

The parameters are the same as those passed to handle_pte_fault()

For read accesses, simply map the system wide empty_zero_page which the
ZERO_PAGE macro returns with the given permissions. The page is write protected
so that a write to the page will result in a page fault

If this is a write fault, then allocate a new page and zero fill it

Unlock the mm_struct as the allocation of a new page could sleep

Allocate a new page

If a page could not be allocated, return -1 to handle the OOM situation

Zero fill the page

Reacquire the lock as the page tables are to be updated

Ensure the cache is coherent

Mark the PTE writable and dirty as it has been written to

Add the page to the LRU list so it may be reclaimed by the swapper later

Mark the page accessed which ensures the page is marked hot and on the top of the
active list

Fix the PTE in the page tables for this process

Update the MMU cache if the architecture needs it

Free the page table lock

Return as a minor fault as even though it is possible the page allocator spent time
writing out pages, data did not have to be read from disk to fill this page
5.4.3 Demand Paging

Function: do_swap_page (mm/memory.c)

This function handles the case where a page has been swapped out. A swapped out page may exist in the swap cache if it is shared between a number of processes or recently swapped in during readahead. This function is broken up into three parts:

- Search for the page in swap cache
- If it does not exist, call swapin_readahead() to read in the page
- Insert the page into the process page tables

```
1117 static int do_swap_page(struct mm_struct * mm,
1118 struct vm_area_struct * vma, unsigned long address,
1119 pte_t * page_table, pte_t orig_pte, int write_access)
1120 {
1121    struct page *page;
1122    swp_entry_t entry = pte_to_swp_entry(orig_pte);
1123    pte_t pte;
1124    int ret = 1;
1125
1126    spin_unlock(&mm->page_table_lock);
1127    page = lookup_swap_cache(entry);

    Function preamble, check for the page in the swap cache

1117-1119 The parameters are the same as those supplied to handle_pte_fault()

1122 Get the swap entry information from the PTE

1126 Free the mm_struct spinlock

1127 Lookup the page in the swap cache

1128    if (!page) {
1129      swapin_readahead(entry);
1130      page = read_swap_cache_async(entry);
1131      if (!page) {
1132        int retval;
1133        spin_lock(&mm->page_table_lock);
1134        retval = pte_same(*page_table, orig_pte) ? -1 : 1;
1135        spin_unlock(&mm->page_table_lock);
1136        return retval;
1137      }
1138    }
1139
1140    /* Had to read the page from swap area: Major fault */
1141    ret = 2;
1142
1143 }
5.4.3. Demand Paging

If the page did not exist in the swap cache, then read it from backing storage with
\texttt{swapin\_readhead()} which reads in the requested pages and a number of pages after it.
Once it completes, \texttt{read\_swap\_cache\_async()} should be able to return the page.

1128-1145 This block is executed if the page was not in the swap cache

1129 \texttt{swapin\_readahead()} reads in the requested page and a number of pages after it. The
number of pages read in is determined by the \texttt{page\_cluster} variable in \texttt{mm/swap.c}
which is initialised to 2 on machines with less than 16MiB of memory and 3 otherwise.
\texttt{2*page\_cluster} pages are read in after the requested page unless a bad or empty page
entry is encountered

1230 Look up the requested page

1131-1141 If the page does not exist, there was another fault which swapped in this page
and removed it from the cache while spinlocks were dropped

1137 Lock the \texttt{mm\_struct}

1138 Compare the two PTEs. If they do not match, -1 is returned to signal an IO error,
else 1 is returned to mark a minor page fault as a disk access was not required for this
particular page.

1139-1140 Free the \texttt{mm\_struct} and return the status

1144 The disk had to be accessed to mark that this is a major page fault

\begin{verbatim}
1147   mark\_page\_accessed(page);
1148
1149   lock\_page(page);
1150
1151   /*
1152    * Back out if somebody else faulted in this pte while we
1153    * released the page table lock.
1154   */
1155   spin\_lock(&mm->page\_table\_lock);
1156   if (!pte\_same(*page\_table, orig\_pte)) {
1157     spin\_unlock(&mm->page\_table\_lock);
1158     unlock\_page(page);
1159     page\_cache\_release(page);
1160     return 1;
1161   }
1162
1163   /* The page isn’t present yet, go ahead with the fault. */
1164
1165   swap\_free(entry);
1166   if (vm\_swap\_full())
1167       remove\_exclusive\_swap\_page(page);
\end{verbatim}
5.4.3. Demand Paging

mm->rss++;

pte = mk_pte(page, vma->vm_page_prot);
if (write_access && can_share_swap_page(page))
    pte = pte_mkdirty(pte_mkwrite(pte));
unlock_page(page);
flush_page_to_ram(page);
flush_icache_page(vma, page);
set_pte(page_table, pte);

/* No need to invalidate - it was non-present before */
update_mmu_cache(vma, address, pte);
spin_unlock(&mm->page_table_lock);
return ret;

Place the page in the process page tables

Mark the page as active so it will be moved to the top of the active LRU list

Lock the page which has the side effect of waiting for the IO swapping in the page to complete

If someone else faulted in the page before we could, the reference to the page is dropped, the lock freed and return that this was a minor fault

The function swap_free() reduces the reference to a swap entry. If it drops to 0, it is actually freed

Page slots in swap space are reserved for pages once they have been swapped out once if possible. If the swap space is full though, the reservation is broken and the slot freed up for another page

The page is now going to be used so increment the mm_struct’s RSS count

Make a PTE for this page

If the page is been written to and it is shared between more than one process, mark it dirty so that it will be kept in sync with the backing storage and swap cache for other processes

Unlock the page

As the page is about to be mapped to the process space, it is possible for some architectures that writes to the page in kernel space will not be visible to the process. flush_page_to_ram() ensures the cache will be coherent

flush_icache_page() is similar in principle except it ensures the icache and dcache’s are coherent
5.4.4. Copy On Write (COW) Pages

1177 Set the PTE in the process page tables

1180 Update the MMU cache if the architecture requires it

1181-1182 Unlock the mm_struct and return whether it was a minor or major page fault

5.4.4 Copy On Write (COW) Pages
Figure 5.10: do_swap_page
5.4.4. Copy On Write (COW) Pages

Figure 5.11: do_wp_page
Chapter 6
High Memory Management

6.1 Mapping High Memory Pages

Figure 6.1: Call Graph: kmap()

Function: kmap (include/asm-i386/highmem.h)

62 static inline void *kmap(struct page *page)
63 {
64     if (in_interrupt())
65         out_of_line_bug();
66     if (page < highmem_start_page)
67         return page_address(page);
68     return kmap_high(page);
69 }

64-65 This function may not be used from interrupt as it may sleep. out_of_line_bug() calls do_exit() and returns an error code. BUG() is not used because BUG() kills the
process with extreme prejudice which would result in the fabled “Aiee, killing interrupt handler!” kernel panic

66-67 If the page is already in low memory, return a direct mapping

68 Call kmap_high() for the beginning of the architecture independent work

**Function: kmap_high** *(mm/highmem.c)*

```c
129 void *kmap_high(struct page *page)
130 {
131     unsigned long vaddr;
132
139     spin_lock(&kmap_lock);
140     vaddr = (unsigned long) page->virtual;
141     if (!vaddr)
142         vaddr = map_new_virtual(page);
143     pkmap_count[PKMAP_NR(vaddr)]++;
144     if (pkmap_count[PKMAP_NR(vaddr)] < 2)
145         BUG();
146     spin_unlock(&kmap_lock);
147     return (void*) vaddr;
148 }
```

139 The kmap_lock protects the virtual field of a page and the pkmap_count array

140 Get the virtual address of the page

141-142 If it is not already mapped, call map_new_virtual() which will map the page and return the virtual address

143 Increase the reference count for this page mapping

144-145 If the count is currently less than 2, it is a serious bug. In reality, severe breakage would have to be introduced to cause this to happen

146 Free the kmap_lock

**Function: map_new_virtual** *(mm/highmem.c)*

This function is divided into three principle parts. The scanning for a free slot, waiting on a queue if none is available and mapping the page.

```c
80 static inline unsigned long map_new_virtual(struct page *page)
81 {
82     unsigned long vaddr;
83     int count;
84
85     start:
86     count = LAST_PKMAP;
```
6.1. Mapping High Memory Pages

87 /* Find an empty entry */
88 for (;;) {
89     last_pkmap_nr = (last_pkmap_nr + 1) & LAST_PKMAP_MASK;
90     if (!last_pkmap_nr) {
91         flush_all_zero_pkmaps();
92         count = LAST_PKMAP;
93     }
94     if (!pkmap_count[last_pkmap_nr])
95         break; /* Found a usable entry */
96     if (--count)
97         continue;
98
99 Start scanning at the last possible slot
100-119 This look keeps scanning and waiting until a slot becomes free. This allows the possibility of an infinite loop for some processes if they were unlucky
101 last_pkmap_nr is the last pkmap that was scanned. To prevent searching over the same pages, this value is recorded so the list is searched circularly. When it reaches LAST_PKMAP, it wraps around to 0
102-93 When last_pkmap_nr wraps around, call flush_all_zero_pkmaps() which will set all entries from 1 to 0 in the pkmap_count array before flushing the TLB. Count is set back to LAST_PKMAP to restart scanning
104-95 If this element is 0, a usable slot has been found for the page
105 Move to the next index to scan
106 {
107     DECLARE_WAITQUEUE(wait, current);
108     current->state = TASK_UNINTERRUPTIBLE;
109     add_wait_queue(&pkmap_map_wait, &wait);
110     spin_unlock(&kmap_lock);
111     schedule();
112     remove_wait_queue(&pkmap_map_wait, &wait);
113     spin_lock(&kmap_lock);
114     /* Somebody else might have mapped it while we slept */
115     if (page->virtual)
116         return (unsigned long) page->virtual;
117     /* Re-start */
118     goto start;
119 }
6.1. Mapping High Memory Pages

If there is no available slot after scanning all the pages once, we sleep on the `pkmap_map_wait` queue until we are woken up after an unmap.

103 Declare the wait queue

105 Set the task as interruptible because we are sleeping in kernel space

106 Add ourselves to the `pkmap_map_wait` queue

107 Free the `kmap_lock` spinlock

108 Call `schedule()` which will put us to sleep. We are woken up after a slot becomes free after an unmap

109 Remove ourselves from the ait queue

110 Re-acquire `kmap_lock`

113-114 If someone else mapped the page while we slept, just return the address and the reference count will be incremented by `kmap_high()`

117 Restart the scanning

120 `vaddr = PKMAP_ADDR(last_pkmap_nr);`
121 `set_pte(&(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap_prot));`

122
123 `pkmap_count[last_pkmap_nr] = 1;`
124 `page->virtual = (void *) vaddr;`
125
126 `return vaddr;`

A slot has been found, map the page

120 Get the virtual address for the slot found

121 Make the PTE entry with the page and required protection and place it in the page tables at the found slot

123 Initialise the value in the `pkmap_count` array to 1. The count is incremented in the parent function and we are sure this is the first mapping if we are in this function in the first place

124 Set the `virtual` field for the page

126 Return the virtual address
Function: `flush_all_zero_pkmaps` *(mm/highmem.c)*

This function cycles through the `pkmap_count` array and sets all entries from 1 to 0 before flushing the TLB.

```c
static void flush_all_zero_pkmaps(void)
{
    int i;

    flush_cache_all();

    for (i = 0; i < LAST_PKMAP; i++) {
        struct page *page;

        if (pkmap_count[i] != 1)
            continue;

        pkmap_count[i] = 0;

        /* sanity check */
        if (pte_none(pkmap_page_table[i]))
            BUG();

        page = pte_page(pkmap_page_table[i]);
        pte_clear(&pkmap_page_table[i]);
        page->virtual = NULL;
    }

    flush_tlb_all();
}
```

As the global page tables are about to change, the CPU caches of all processors have to be flushed.

48-76 Cycle through the entire `pkmap_count` array

57-58 If the element is not 1, move to the next element

59 Set from 1 to 0

62-63 Make sure the PTE is not somehow mapped

72-73 Unmap the page from the PTE and clear the PTE

75 Update the `virtual` field as the page is unmapped

77 Flush the TLB
6.1.1 Unmapping Pages

Function: **kunmap** *(include/asm-i386/highmem.h)*

```c
71 static inline void kunmap(struct page *page) {
    73    if (in_interrupt())
    74        out_of_line_bug();
    75    if (page < highmem_start_page)
    76        return;
    77    kunmap_high(page);
78 }
```

73-74 `kunmap()` cannot be called from interrupt so exit gracefully

75-76 If the page already is in low memory, there is no need to unmap

77 Call the architecture independent function `kunmap_high()`

Function: **kunmap_high** *(mm/highmem.c)*

```c
150 void kunmap_high(struct page *page) {
152    unsigned long vaddr;
153    unsigned long nr;
154    int need_wakeup;
155    spin_lock(&kmap_lock);
156    vaddr = (unsigned long) page->virtual;
158    if (!vaddr)
159        BUG();
160    nr = PKMAP_NR(vaddr);
166    need_wakeup = 0;
167    switch (--pkmap_count[nr]) {
168        case 0:
169            BUG();
170        case 1:
171            need_wakeup = waitqueue_active(&pkmap_map_wait);
172    }
181    spin_unlock(&kmap_lock);
185    /* do wake-up, if needed, race-free outside of the spin lock */
186    if (need_wakeup)
187        wake_up(&pkmap_map_wait);
```

156 Acquire `kmap_lock` protecting the `virtual()` field and the `pkmap_count` array
6.2. Mapping High Memory Pages Atomically

The following is an example km_type enumeration for the x86. It lists the different uses interrupts have for atomically calling kmap. Note how KM_TYPE_NR is the last element so it doubles up as a count of the number of elements.

```c
enum km_type {
    KM_BOUNCE_READ,
    KM_SKB_SUNRPC_DATA,
    KM_SKB_DATA_SOFTIRQ,
    KM_USER0,
    KM_USER1,
    KM_BH_IRQ,
    KM_TYPE_NR
};
```

Function: kmap_atomic (include/asm-i386/highmem.h)

This is the atomic version of kmap(). Note that at no point is a spinlock held or does it sleep. A spinlock is not required as every processor has its own reserved space.

```c
static inline void *kmap_atomic(struct page *page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;

    if (page < highmem_start_page)
        return page_address(page);
    return page_address(page);
}
```
6.2. Mapping High Memory Pages Atomically

idx = type + KM_TYPE_NR*smp_processor_id();
vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
#if HIGHMEM_DEBUG
if (!pte_none(*(kmap_pte-idx)))
  out_of_line_bug();
#endif
set_pte(kmap_pte-idx, mk_pte(page, kmap_prot));
__flush_tlb_one(vaddr);
return (void*) vaddr;

The parameters are the page to map and the type of usage required. One slot per usage per processor is maintained.

If the page is in low memory, return a direct mapping.

type gives which slot to use. KM_TYPE_NR * smp_processor_id() gives the set of slots reserved for this processor.

Get the virtual address.

Debugging code. In reality a PTE will always exist.

Set the PTE into the reserved slot.

Flush the TLB for this slot.

Return the virtual address.

Function: kunmap_atomic (include/asm-i386/highmem.h)

This entire function is debug code. The reason is that as pages are only mapped here atomically, they will only be used in a tiny place for a short time before being unmapped. It is safe to leave the page there as it will not be referenced after unmapping and another mapping to the same slot will simply replace it.

static inline void kunmap_atomic(void *kvaddr, enum km_type type) {
#if HIGHMEM_DEBUG
  unsigned long vaddr = (unsigned long) kvaddr;
  enum fixed_addresses idx = type + KM_TYPE_NR*smp_processor_id();
  if (vaddr < FIXADDR_START) // FIXME
    return;
  if (vaddr != __fix_to_virt(FIX_KMAP_BEGIN+idx))
    out_of_line_bug();
}
6.3 Bounce Buffers

Function: create_buffers (mm/highmem.c)

Figure 6.2: Call Graph: create_bounce

High level function for the creation of bounce buffers. It is broken into two major parts, the allocation of the necessary resources, and the copying of data from the template.

398 struct buffer_head * create_bounce(int rw, struct buffer_head * bh_orig) 399 { 400 struct page *page;
401 struct buffer_head *bh;
402 403 if (!PageHighMem(bh_orig->b_page)) 404 return bh_orig;
405 406 }
6.3. Bounce Buffers

bh = alloc_bounce_bh();
page = alloc_bounce_page();
set_bh_page(bh, page, 0);

The parameters of the function are

- **rw** is set to 1 if this is a write buffer
- **bh_orig** is the template buffer head to copy from

If the template buffer head is already in low memory, simply return it.

Allocate a buffer head from the slab allocator or from the emergency pool if it fails.

Allocate a page from the buddy allocator or the emergency pool if it fails.

Associate the allocated page with the allocated **buffer_head**

bh->b_next = NULL;
bh->b_blocknr = bh_orig->b_blocknr;
bh->b_size = bh_orig->b_size;
bh->b_list = -1;
bh->b_dev = bh_orig->b_dev;
bh->b_count = bh_orig->b_count;
bh->b_rdev = bh_orig->b_rdev;
bh->b_state = bh_orig->b_state;

#ifdef HIGHMEM_DEBUG
bh->b_flushtime = jiffies;
bh->b_next_free = NULL;
bh->b_prev_free = NULL;
/* bh->b_this_page */
bh->b_reqnext = NULL;
bh->b_pprev = NULL;
#endif

/* bh->b_page */
if (rw == WRITE) {
bh->b_end_io = bounce_end_io_write;
copy_from_high_bh(bh, bh_orig);
} else
bh->b_end_io = bounce_end_io_read;
bh->b_private = (void *)bh_orig;
bh->b_rsector = bh_orig->b_rsector;
#ifdef HIGHMEM_DEBUG
memset(&bh->b_wait, -1, sizeof(bh->b_wait));
#endif
6.3. Bounce Buffers

Populate the newly created buffer_head

Copy in information essentially verbatim except for the b_list field as this buffer is not directly connected to the others on the list

Debugging only information

If this is a buffer that is to be written to then the callback function to end the IO is bounce_end_io_write() which is called when the device has received all the information. As the data exists in high memory, it is copied “down” with copy_from_high_bh()

If we are waiting for a device to write data into the buffer, then the callback function bounce_end_io_read() is used

Copy the remaining information from the template buffer_head

Return the new bounce buffer

Function: alloc_bounce_bh (mm/highmem.c)

This function first tries to allocate a buffer_head from the slab allocator and if that fails, an emergency pool will be used.

struct buffer_head *alloc_bounce_bh (void)
{
    struct list_head *tmp;
    struct buffer_head *bh;

    bh = kmem_cache_alloc(bh_cachep, SLAB_NOHIGHIO);
    if (bh)
        return bh;

    tmp = &emergency_bhs;
    spin_lock_irq(&emergency_lock);

    Try to allocate a new buffer_head from the slab allocator. Note how the request is made to not use IO operations that involve high IO to avoid recursion

    If the allocation was successful, return

    If it was not, wake up bdflush to launder pages

repeat_alloc:
    tmp = &emergency_bhs;
    spin_lock_irq(&emergency_lock);
The allocation from the slab failed so allocate from the emergency pool.

380 Get the end of the emergency buffer head list

381 Acquire the lock protecting the pools

382-386 If the pool is not empty, take a buffer_head from the list and decrement the nr_emergency_bhs counter

387 Release the lock

388-389 If the allocation was successful, return it

392 If not, we are seriously short of memory and the only way the pool will replenish is if high memory IO completes. Therefore, requests on tq_disk are started so the data will be written to disk, probably freeing up pages in the process

394 Yield the processor

395 Attempt to allocate from the emergency pools again

Function: alloc_bounce_page (mm/highmem.c)

This function is essentially identical to alloc_bounce_bh() It first tries to allocate a page from the buddy allocator and if that fails, an emergency pool will be used.

326 struct page *alloc_bounce_page (void)
327 {
328     struct list_head *tmp;
329     struct page *page;
330     page = alloc_page(GFP_NOHIGHIO);
332     if (page)
return page;

wakeup_bdflush();

Allocate from the buddy allocator and return the page if successful
Wake bdflush to launder pages

repeat_alloc:

tmp = &emergency_pages;
spin_lock_irq(&emergency_lock);
if (!list_empty(tmp)) {
    page = list_entry(tmp->next, struct page, list);
    list_del(tmp->next);
    nr_emergency_pages--;
}
spin_unlock_irq(&emergency_lock);
if (page)
    return page;

/* we need to wait I/O completion */
run_task_queue(&tq_disk);
yield();
goto repeat_alloc;

Get the end of the emergency buffer head list
Acquire the lock protecting the pools
If the pool is not empty, take a page from the list and decrement the number of available nr_emergency_pages
Release the lock
If the allocation was successful, return it
Run the IO task queue to try and replenish the emergency pool
Yield the processor
Attempt to allocate from the emergency pools again
6.3.1 Copying via Bounce Buffers

Function: bounce_end_io_write (mm/highmem.c)
This function is called when a bounce buffer used for writing to a device completes IO. As the buffer is copied from high memory and to the device, there is nothing left to do except reclaim the resources.

```c
312 static void bounce_end_io_write (struct buffer_head *bh, int uptodate) {
313     bounce_end_io(bh, uptodate);
314 }
```

Function: bounce_end_io_read (mm/highmem.c)
This is called when data has been read from the device and needs to be copied to high memory. It is called from interrupt so has to be more careful.

```c
317 static void bounce_end_io_read (struct buffer_head *bh, int uptodate) {
318     struct buffer_head *bh_orig =
319         (struct buffer_head *)(bh->b_private);
320
321     if (uptodate)
322         copy_to_high_bh_irq(bh_orig, bh);
323     bounce_end_io(bh, uptodate);
324 }
```

321-322 The data is just copied to the bounce buffer to needs to be moved to high memory with `copy_to_high_bh_irq()`.

323 Reclaim the resources

Function: copy_from_high_bh (mm/highmem.c)
This function copies data from a high memory buffer_head to a bounce buffer.

```c
208 static inline void copy_from_high_bh (struct buffer_head *to, struct buffer_head *from) {
209     struct page *p_from;
210     char *vfrom;
211
212     p_from = from->b_page;
213
214     vfrom = kmap_atomic(p_from, KM_USER0);
215
216     memcpy(to->b_data, vfrom + bh_offset(from), to->b_size);
217     kunmap_atomic(vfrom, KM_USER0);
218 }
Map the high memory page into low memory. This path is protected by the IRQ safe
lock `io_request_lock` so it is safe to call `kmap_atomic()`

Copy the data

Unmap the page

**Function:** `copy_to_high_bh_irq` *(mm/highmem.c)*

Called from interrupt after the device has finished writing data to the bounce buffer. This function copies data to high memory

```c
static inline void copy_to_high_bh_irq (struct buffer_head *to,
                                  struct buffer_head *from)
{
    struct page *p_to;
    char *vto;
    unsigned long flags;

    p_to = to->b_page;
    __save_flags(flags);
    __cli();
    vto = kmap_atomic(p_to, KM_BOUNCE_READ);
    memcpy(vto + bh_offset(to), from->b_data, to->b_size);
    kunmap_atomic(vto, KM_BOUNCE_READ);
    __restore_flags(flags);
}
```

Save the flags and disable interrupts

Map the high memory page into low memory

Copy the data

Unmap the page

Restore the interrupt flags

**Function:** `bounce_end_io` *(mm/highmem.c)*

Reclaims the resources used by the bounce buffers. If emergency pools are depleted, the resources are added to it.

```c
static inline void bounce_end_io (struct buffer_head *bh, int uptodate)
{
    struct page *page;
    struct buffer_head *bh_orig =
        (struct buffer_head *)(bh->b_private);
    unsigned long flags;
```
6.3.1. Copying via Bounce Buffers

243 bh_orig->b_end_io(bh_orig, uptodate);
244
245 page = bh->b_page;
246
247 spin_lock_irqsave(&emergency_lock, flags);
248 if (nr_emergency_pages >= POOL_SIZE)
249   __free_page(page);
250 else {
251     /*
252     * We are abusing page->list to manage
253     * the highmem emergency pool:
254     */
255     list_add(&page->list, &emergency_pages);
256     nr_emergency_pages++;
257 }
258
259 if (nr_emergency_bhs >= POOL_SIZE) {
260 #ifdef HIGHMEM_DEBUG
261     /* Don’t clobber the constructed slab cache */
262     init_waitqueue_head(&bh->b_wait);
263 #endif
264     kmem_cache_free(bh_cachep, bh);
265 } else {
266     /*
267     * Ditto in the bh case, here we abuse b_inode_buffers:
268     */
269     list_add(&bh->b_inode_buffers, &emergency_bhs);
270     nr_emergency_bhs++;
271 }
272 spin_unlock_irqrestore(&emergency_lock, flags);
273 }

243 Call the IO completion callback for the original buffer_head
245 Get the pointer to the buffer page to free
247 Acquire the lock to the emergency pool
248-249 If the page pool is full, just return the page to the buddy allocator
250-257 Otherwise add this page to the emergency pool
259-265 If the buffer_head pool is full, just return it to the slab allocator
265-271 Otherwise add this buffer_head to the pool
272 Release the lock
6.4 Emergency Pools

There is only one function of relevance to the emergency pools and that is the init function. It is called during system startup and then the code is deleted as it is never needed again.

**Function**: `init_emergency_pool (mm/highmem.c)`

Create a pool for emergency pages and for emergency `buffer_heads`

```c
static __init int init_emergency_pool(void)
{
    struct sysinfo i;
    si_meminfo(&i);
    si_swapinfo(&i);
    if (!i.totalhigh)
        return 0;

    spin_lock_irq(&emergency_lock);
    while (nr_emergency_pages < POOL_SIZE) {
        struct page * page = alloc_page(GFP_ATOMIC);
        if (!page) {
            printk("couldn’t refill highmem emergency pages");
            break;
        }
        list_add(&page->list, &emergency_pages);
        nr_emergency_pages++;
    }
    spin_unlock_irq(&emergency_lock);
    printk("allocated %d pages and %d bhs reserved for the highmem bounces\n", nr_emergency_pages, nr_emergency_bhs);

    spin_lock_irq(&emergency_lock);
    while (nr_emergency_bhs < POOL_SIZE) {
        struct buffer_head * bh =
            kmem_cache_alloc(bh_cachep, SLAB_ATOMIC);
        if (!bh) {
            printk("couldn’t refill highmem emergency bhs");
            break;
        }
        list_add(&bh->b_inode_buffers, &emergency_bhs);
        nr_emergency_bhs++;
    }
    spin_unlock_irq(&emergency_lock);
```
6.4. Emergency Pools

294-302 Allocate POOL_SIZE buffer_heads from the slab allocator and add them to a linked list linked by b_inode_buffers. Keep track of how many heads are in the pool with nr_emergency_bhs.

303 Release the lock protecting the pools

307 Return success
Chapter 7

Page Frame Reclamation

7.1 Page Swap Daemon

**Function: kswapd_init (mm/vmscan.c)**
Start the kswapd kernel thread

767 static int __init kswapd_init(void)
768 {
769     printk("Starting kswapd\n");
770     swap_setup();
771     kernel_thread(kswapd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
772     return 0;
773 }

770 swap_setup() setups up how many pages will be prefetched when reading from backing storage based on the amount of physical memory

771 Start the **kswapd** kernel thread

**Function: kswapd (mm/vmscan.c)**
The main function of the **kswapd** kernel thread.

720 int kswapd(void *unused)
721 {
722     struct task_struct *tsk = current;
723     DECLARE_WAITQUEUE(wait, tsk);
724     daemonize();
725     strcpy(tsk->comm, "kswapd");
726     sigfillset(&tsk->blocked);
728     tsk->flags |= PF_MEMALLOC;
741     for (;;) {
747         __set_current_state(TASK_INTERRUPTIBLE);
7.1. Page Swap Daemon

748 add_wait_queue(&kswapd_wait, &wait);
749
750 mb();
751 if (kswapd_can_sleep())
752 schedule();
753
754 __set_current_state(TASK_RUNNING);
755 remove_wait_queue(&kswapd_wait, &wait);
756
762 kswapd_balance();
763 run_task_queue(&tq_disk);
764 }
765 }

725 Call daemonize() which will make this a kernel thread, remove the mm context, close all files and re-parent the process

726 Set the name of the process

727 Ignore all signals

741 By setting this flag, the physical page allocator will always try to satisfy requests for pages. As this process will always be trying to free pages, it is worth satisfying requests

746-764 Endlessly loop

747-748 This adds kswapd to the wait queue in preparation to sleep

750 The Memory Block (mb) function ensures that all reads and writes that occurred before this line will be visible to all CPU’s

751 kswapd_can_sleep() cycles through all nodes and zones checking the need_balance field. If any of them are set to 1, kswapd can not sleep

752 By calling schedule, kswapd will sleep until woken again by the physical page allocator

754-755 Once woken up, kswapd is removed from the wait queue as it is now running

762 kswapd_balance() cycles through all zones and calls try_to_free_pages_zone() for each zone that requires balance

763 Run the task queue for processes waiting to write to disk

Function: kswapd_can_sleep (mm/vmscan.c)

Simple function to cycle through all pgdats to call kswapd_can_sleep_pgdat() on each.
7.1. Page Swap Daemon

```c
695 static int kswapd_can_sleep(void)
696 {
697     pg_data_t * pgdat;
698     for_each_pgdat(pgdat) {
699         if (!kswapd_can_sleep_pgdat(pgdat))
700             return 0;
701     }
702     return 1;
703 }
```

699-702 `for_each_pgdat()` does exactly as the name implies. It cycles through all available pgdat's. On the x86, there will only be one.

**Function: kswapd_can_sleep_pgdat** *(mm/vmscan.c)*
Cycles through all zones to make sure none of them need balance.

```c
680 static int kswapd_can_sleep_pgdat(pg_data_t * pgdat)
681 {
682     zone_t * zone;
683     int i;
684     for (i = pgdat->nr_zones-1; i >= 0; i--) {
685         zone = pgdat->node_zones + i;
686         if (!zone->need_balance)
687             continue;
688         return 0;
689     }
690     return 1;
691 }
```

685-689 Simple for loop to cycle through all zones

686 The `node_zones` field is an array of all available zones so adding `i` gives the index

687-688 If the zone does not need balance, continue

689 0 is returned if any needs balance indicating `kswapd` cannot sleep

692 Return indicating `kswapd` can sleep if the for loop completes

**Function: kswapd_balance** *(mm/vmscan.c)*
Continuously cycle through each pgdat until none require balancing
static void kswapd_balance(void) {
    int need_more_balance;
    pg_data_t * pgdat;

do {
    need_more_balance = 0;

for_each_pgdat(pgdat)
    need_more_balance |= kswapd_balance_pgdat(pgdat);
} while (need_more_balance);
}

Continuously cycle through each pgdat

For each pgdat, call kswapd_balance_pgdat(). If any of them had required balancing, need_more_balance will be equal to 1

Function: kswapd_balance_pgdat (mm/vmscan.c)

static int kswapd_balance_pgdat(pg_data_t * pgdat) {
    int need_more_balance = 0, i;
    zone_t * zone;

for (i = pgdat->nr_zones-1; i >= 0; i--) {
    zone = pgdat->node_zones + i;
    if (unlikely(current->need_resched))
        schedule();
    if (!zone->need_balance)
        continue;
    if (!try_to_free_pages_zone(zone, GFP_KSWAPD)) {
        zone->need_balance = 0;
        __set_current_state(TASK_INTERRUPTIBLE);
        schedule_timeout(HZ);
        continue;
    }
    if (check_classzone_need_balance(zone))
        need_more_balance = 1;
    else
        zone->need_balance = 0;
}
return need_more_balance;

Cycle through each zone and call try_to_free_pages_zone() if it needs re-balancing
node_zones is an array and i is an index within it

Call schedule() if the quanta is expired to prevent kswapd hogging the CPU

If the zone does not require balance, move to the next one

If the function returns 0, it means the out_of_memory() function was called because a sufficient number of pages could not be freed. kswapd sleeps for 1 second to give the system a chance to reclaim the killed processes pages

If is was successful, check_classzone_need_balance() is called to see if the zone requires further balancing or not

Return 1 if one zone requires further balancing

7.2 Page Cache

Function: lru_cache_add (mm/swap.c)

Adds a page to the LRU inactive_list.

58 void lru_cache_add(struct page * page)
59 {
60     if (!PageLRU(page)) {
61         spin_lock(&pagemap_lru_lock);
62         if (!TestSetPageLRU(page))
63             add_page_to_inactive_list(page);
64         spin_unlock(&pagemap_lru_lock);
65     }
66 }

If the page is not already part of the LRU lists, add it

Acquire the LRU lock

Test and set the LRU bit. If it was clear then call add_page_to_inactive_list()

Release the LRU lock

Function: add_page_to_active_list (include/linux/swap.h)

Adds the page to the active_list

#define add_page_to_active_list(page) \  
179 do { \ 180     DEBUG_LRU_PAGE(page); \ 181     SetPageActive(page); \ 182     list_add(&(page)->lru, &active_list); \ 183     nr_active_pages++; \ 184 } while (0)
The `DEBUG_LRU_PAGE()` macro will call `BUG()` if the page is already on the LRU list or is marked been active.

Update the flags of the page to show it is active.

Add the page to the `active_list`.

Update the count of the number of pages in the `active_list`.

**Function: add_page_to_inactive_list** *(include/linux/swap.h)*

Add the page to the `inactive_list`.

```c
#define add_page_to_inactive_list(page) \
  do { \
    DEBUG_LRU_PAGE(page); \
    list_add(&(page)->lru, &inactive_list); \ 
    nr_inactive_pages++; \ 
  } while (0)
```

The `DEBUG_LRU_PAGE()` macro will call `BUG()` if the page is already on the LRU list or is marked been active.

Add the page to the `inactive_list`.

Update the count of the number of inactive pages on the list.

**Function: lru_cache_del** *(mm/swap.c)*

Acquire the lock protecting the LRU lists before calling `__lru_cache_del()`.

```c
void lru_cache_del(struct page * page) 
{
  spin_lock(&pagemap_lru_lock);
  __lru_cache_del(page);
  spin_unlock(&pagemap_lru_lock);
}
```

Acquire the LRU lock.

`__lru_cache_del()` does the “real” work of removing the page from the LRU lists.

Release the LRU lock.
Function: `__lru_cache_del` (*mm/swap.c*)
Select which function is needed to remove the page from the LRU list.

75 void __lru_cache_del(struct page * page)  
76 {  
77 if (TestClearPageLRU(page)) {  
78 if (PageActive(page)) {  
79 del_page_from_active_list(page);  
80 } else {  
81 del_page_from_inactive_list(page);  
82 }  
83 }  
84 }

77 Test and clear the flag indicating the page is in the LRU
78-82 If the page is on the LRU, select the appropriate removal function
78-79 If the page is active, then call `del_page_from_active_list()` else delete from the
inactive list with `del_page_from_inactive_list()`

Function: `del_page_from_active_list` (*include/linux/swap.h*)
Remove the page from the `active_list`

194 #define del_page_from_active_list(page) \  
195 do {  
196 list_del(& (page)->lru);  
197 ClearPageActive(page);  
198 nr_active_pages--;  
199 } while (0)

196 Delete the page from the list
197 Clear the flag indicating it is part of `active_list`. The flag indicating it is part of the
LRU list has already been cleared by `__lru_cache_del()`
198 Update the count of the number of pages in the `active_list`

Function: `del_page_from_inactive_list` (*include/linux/swap.h*)

201 #define del_page_from_inactive_list(page) \  
202 do {  
203 list_del(& (page)->lru);  
204 nr_inactive_pages--;  
205 } while (0)

203 Remove the page from the LRU list
204 Update the count of the number of pages in the `inactive_list`
Function: **mark_page_accessed** (*mm/filemap.c*)

This marks that a page has been referenced. If the page is already on the `active_list` or the referenced flag is clear, the referenced flag will be simply set. If it is in the `inactive_list` and the referenced flag has been set, `activate_page()` will be called to move the page to the top of the `active_list`.

```c
1316 void mark_page_accessed(struct page *page) {
1317     if (!PageActive(page) && PageReferenced(page)) {
1318         activate_page(page);
1319         ClearPageReferenced(page);
1320     } else
1321         SetPageReferenced(page);
1322 }
```

1318-1321 If the page is on the `inactive_list` (!PageActive) and has been referenced recently (PageReferenced), `activate_page()` is called to move it to the `active_list`

Otherwise, mark the page as been referenced

Function: **activate_lock** (*mm/swap.c*)

Acquire the LRU lock before calling `activate_page_nolock()` which moves the page from the `inactive_list` to the `active_list`.

```c
47 void activate_page(struct page * page) {
48     spin_lock(&pagemap_lru_lock);
49     activate_page_nolock(page);
50     spin_unlock(&pagemap_lru_lock);
51 }
```

49 Acquire the LRU lock
50 Call the main work function
51 Release the LRU lock

Function: **activate_page_nolock** (*mm/swap.c*)

Move the page from the `inactive_list` to the `active_list`

```c
39 static inline void activate_page_nolock(struct page * page) {
40     if (PageLRU(page) && !PageActive(page)) {
41         del_page_from_inactive_list(page);
42         add_page_to_active_list(page);
43     }
44 }
```

41 Make sure the page is on the LRU and not already on the `active_list`
42-43 Delete the page from the `inactive_list` and add to the `active_list`
Function: page_cache_get (include/linux/pagemap.h)

31 #define page_cache_get(x) get_page(x)

31 Simple call get_page() which simply uses atomic_inc() to increment the page reference count

Function: page_cache_release (include/linux/pagemap.h)

32 #define page_cache_release(x) __free_page(x)

32 Call __free_page() which decrements the page count. If the count reaches 0, the page will be freed

Function: add_to_page_cache (mm/filemap.c)

Acquire the lock protecting the page cache before calling __add_to_page_cache() which will add the page to the page hash table and inode queue which allows the pages belonging to files to be found quickly.

665 void add_to_page_cache(struct page * page, 
  struct address_space * mapping, 
  unsigned long offset)
666 {
667   spin_lock(&pagecache_lock);
668   __add_to_page_cache(page, mapping, 
      offset, page_hash(mapping, offset));
669   spin_unlock(&pagecache_lock);
670   lru_cache_add(page);
671 }

667 Acquire the lock protecting the page hash and inode queues
668 Call the function which performs the “real” work
669 Release the lock protecting the hash and inode queue
670 Add the page to the page cache

Function: __add_to_page_cache (mm/filemap.c)

Clear all page flags, lock it, take a reference and add it to the inode and hash queues.

651 static inline void __add_to_page_cache(struct page * page, 
652   struct address_space * mapping, unsigned long offset, 
653   struct page **hash)
654 {
655   unsigned long flags;
656
657   flags = page->flags & ~(1 << PG_uptodate |
7.3 Shrinking all caches

Function: shrink_caches (mm/vmscan.c)

```c
560 static int shrink_caches(zone_t *classzone, int priority,
                             unsigned int gfp_mask, int nr_pages)
561 {
562   int chunk_size = nr_pages;
563   unsigned long ratio;
564   nr_pages -= kmem_cache_reap(gfp_mask);
565   if (nr_pages <= 0)
566     return 0;
567   nr_pages = chunk_size;
568   /* try to keep the active list 2/3 of the size of the cache */
569   ratio = (unsigned long) nr_pages *
570         nr_active_pages / ((nr_inactive_pages + 1) * 2);
571   refill_inactive(ratio);
572   nr_pages = shrink_cache(nr_pages, classzone, gfp_mask, priority);
```

Clear all page flags

Lock the page

Take a reference to the page in case it gets freed prematurely

Update the index so it is known what file offset this page represents

Add the page to the inode queue. This links the page via the page→list to the
  clean_pages list in the address_space and points the page→mapping to the same
  address_space

Add it to the page hash. Pages are hashed based on the address_space and the inode.
  It allows pages belonging to an address_space to be found without having to linearly
  search the inode queue

7.3 Shrinking all caches

Function: shrink_caches (mm/vmscan.c)
7.3. Shrinking all caches
7.3. Shrinking all caches

```c
if (nr_pages <= 0)
    return 0;

shrink_dcache_memory(priority, gfp_mask);
shrink_icache_memory(priority, gfp_mask);
#ifdef CONFIG_QUOTA
    shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
#endif
return nr_pages;
```

The parameters are as follows;

- `classzone` is the zone that pages should be freed from
- `priority` determines how much work will be done to free pages
- `gfp_mask` determines what sort of actions may be taken
- `nr_pages` is the number of pages remaining to be freed

Ask the slab allocator to free up some pages. If enough are freed, the function returns otherwise `nr_pages` will be freed from other caches.

Move pages from the `active_list` to the `inactive_list` with `refill_inactive()`. The number of pages moved depends on how many pages need to be freed and to have `active_list` about two thirds the size of the page cache.

Shrink the page cache, if enough pages are freed, return.

Shrink the dcache, icache and dqcache. These are small objects in themselves but the cascading effect frees up a lot of disk buffers.

Return the number of pages remaining to be freed.

Function: `try_to_free_pages` *(mm/vmscan.c)*

This function cycles through all pgdats and tries to balance the preferred allocation zone (usually `ZONE_NORMAL`) for each of them. This function is only called from one place, `buffer.c:free_more_memory()` when the buffer manager fails to create new buffers or grow existing ones. It calls `try_to_free_pages()` with `GFP_NOIO` as the `gfp_mask`.

This results in the first zone in `pg_data_t:node_zonelists` having pages freed so that buffers can grow. This array is the preferred order of zones to allocate from and usually will begin with `ZONE_NORMAL` which is required by the buffer manager. On NUMA architectures, some nodes may have `ZONE_DMA` as the preferred zone if the memory bank is dedicated to IO devices and UML also uses only this zone. As the buffer manager is restricted in the zones it uses, there is no point balancing other zones.

```c
int try_to_free_pages(unsigned int gfp_mask)
{
```
7.3. Shrinking all caches

609    pg_data_t *pgdat;
610    zonelist_t *zonelist;
611    unsigned long pf_free_pages;
612    int error = 0;
613
614    pf_free_pages = current->flags & PF_FREE_PAGES;
615    current->flags &= ~PF_FREE_PAGES;
616
617    for_each_pgdat(pgdat) {
618        zonelist = pgdat->node_zonelists +
                  (gfp_mask & GFP_ZONEMASK);
619        error |= try_to_free_pages_zone(
                  zonelist->zones[0], gfp_mask);
620    }
621
622    current->flags |= pf_free_pages;
623    return error;
624 }

614-615 This clears the PF_FREE_PAGES flag if it is set so that pages freed by the process
will be returned to the global pool rather than reserved for the process itself

617-620 Cycle through all nodes and call try_to_free_pages() for the preferred zone in
each node

618 This function is only called with GFP_NOIO as a parameter. When ANDed with
GFP_ZONEMASK, it will always result in 0

622-623 Restore the process flags and return the result

Function: try_to_free_pages_zone (mm/vmscan.c)
Try to free SWAP_CLUSTER_MAX pages from the supplied zone.

587    int try_to_free_pages_zone(zone_t *classzone, unsigned int gfp_mask)
588    {
589        int priority = DEF_PRIORITY;
590        int nr_pages = SWAP_CLUSTER_MAX;
591        gfp_mask = pf_gfp_mask(gfp_mask);
592        do {
593            nr_pages = shrink_caches(classzone, priority,
594                gfp_mask, nr_pages);
595            if (nr_pages <= 0)
596                return 1;
597        } while (--priority);
598    /*
7.4 Refilling inactive_list

Function: refill_inactive (mm/vmscan.c)
Move nr_pages from the active_list to the inactive_list

```c
533 static void refill_inactive(int nr_pages)
534 {
535     struct list_head * entry;
536
537     spin_lock(&pagemap_lru_lock);
538     entry = active_list.prev;
539     while (nr_pages && entry != &active_list) {
540         struct page * page;
541
542         page = list_entry(entry, struct page, lru);
543         entry = entry->prev;
544         if (PageTestandClearReferenced(page)) {
545             list_del(&page->lru);
546             list_add(&page->lru, &active_list);
547             continue;
548         }
549     }
```
7.5 Reclaiming pages from the page cache

Function: shrink_cache (mm/vmscan.c)

338 static int shrink_cache(int nr_pages, zone_t * classzone, unsigned int gfp_mask, int priority) {  
339    struct list_head * entry;  
340    int max_scan = nr_inactive_pages / priority;  
341    int max_mapped = min((nr_pages << (10 - priority)), max_scan / 10);  
342    spin_lock(&pagemap_lru_lock);  
343    while (--max_scan >= 0 &&  
344        (entry = inactive_list.prev) != &inactive_list) {

338 The parameters are as follows;  

nr_pages The number of pages to swap out  

classzone The zone we are interested in swapping pages out for. Pages not belonging  
to this zone are skipped
The gfp mask determining what actions may be taken
The priority of the function, starts at DEF_PRIORITY (6) and decreases to the highest priority of 1

The maximum number of pages to scan is the number of pages in the active_list divided by the priority. At lowest priority, 1/6th of the list may scanned. At highest priority, the full list may be scanned

The maximum amount of process mapped pages allowed is either one tenth of the max_scan value or \( nr_{pages} \times 2^{10-priority} \). If this number of pages are found, whole processes will be swapped out

Lock the LRU list

Keep scanning until max_scan pages have been scanned or the inactive_list is empty

Reschedule if the quanta has been used up
Free the LRU lock as we are about to sleep
Show we are still running
Call schedule() so another process can be context switched in
Re-acquire the LRU lock
Move to the next page, this has the curious side effect of skipping over one page. It is unclear why this happens and is possibly a bug
7.5. Reclaiming pages from the page cache

365 * Zero page counts can happen because we unlink the pages
366 * _after_ decrementing the usage count..
367 */
368 if (unlikely(!page_count(page))
369     continue;
370
371 if (!memclass(page_zone(page), classzone))
372     continue;
373
374 /* Racy check to avoid trylocking when not worthwhile */
375 if (!page->buffers &&
376     (page_count(page) != 1 || !page->mapping))
377     goto page_mapped;
378
379 Get the struct page for this entry in the LRU
380
381-382 It is a bug if the page either belongs to the active_list or is currently marked
382 as active
383
384-385 Move the page to the top of the inactive_list so that if the page is skipped, it
385 will not be simply examined a second time
386
387-388 If the page count has already reached 0, skip over it. This is possible if another
388 process has just unlinked the page and is waiting for something like IO to complete
389 before removing it from the LRU
390
391-392 Skip over this page if it belongs to a zone we are not currently interested in
392
393-394 If the page is mapped by a process, then goto page_mapped where the max_mapped
394 is decremented and next page examined. If max_mapped reaches 0, process pages will
395 be swapped out
396
397 if (unlikely(TryLockPage(page))) {
398     if (PageLaunder(page) && (gfp_mask & __GFP_FS)) {
399         page_cache_get(page);
400         spin_unlock(&pagemap_lru_lock);
401         wait_on_page(page);
402         page_cache_release(page);
403         spin_lock(&pagemap_lru_lock);
404     }
405     continue;
406 }
407
408 Page is locked and the launder bit is set. In this case, wait until the IO is complete
409 and then try to free the page
If we could not lock the page, the PG_launder bit is set and the GFP flags allow
the caller to perform FS operations, then...

Take a reference to the page so it does not disappear while we sleep

Free the LRU lock

Wait until the IO is complete

Release the reference to the page. If it reaches 0, the page will be freed

Re-acquire the LRU lock

Move to the next page

if (PageDirty(page) &&
    is_page_cache_freeable(page) && page->mapping) {
    int (*writepage)(struct page *);
    writepage = page->mapping->a_ops->writepage;
    if ((gfp_mask & __GFP_FS) && writepage) {
        ClearPageDirty(page);
        SetPageLaunder(page);
        page_cache_get(page);
        spin_unlock(&pagemap_lru_lock);
        writepage(page);
        page_cache_release(page);
        spin_lock(&pagemap_lru_lock);
        continue;
    }
}

This handles the case where a page is dirty, is not mapped by any process has no
buffers and is backed by a file or device mapping. The page is cleaned and will be removed
by the previous block of code during the next pass through the list.

PageDirty checks the PG_dirty bit, is_page_cache_freeable() will return true if it
is not mapped by any process and has no buffers

Get a pointer to the necessary writepage() function for this mapping or device

This block of code can only be executed if a writepage() function is available
and the GFP flags allow file operations

Clear the dirty bit and mark that the page is being laundered

Take a reference to the page so it will not be freed unexpectedly
Unlock the LRU list

Call the writepage function

Release the reference to the page

Re-acquire the LRU list lock and move to the next page

if (page->buffers) {
    spin_unlock(&pagemap_lru_lock);
    /* avoid to free a locked page */
    page_cache_get(page);
    if (try_to_release_page(page, gfp_mask)) {
        if (!page->mapping) {
            spin_lock(&pagemap_lru_lock);
            UnlockPage(page);
            __lru_cache_del(page);
            page_cache_release(page);
            if (--nr_pages)
                continue;
            break;
        } else {
            page_cache_release(page);
        }
    } else {
        UnlockPage(page);
        page_cache_release(page);
        spin_lock(&pagemap_lru_lock);
        continue;
    }
}

Page has buffers associated with it that must be freed.

Release the LRU lock as we may sleep

Take a reference to the page

Call try_to_release_page() which will attempt to release the buffers associated with the page. Returns 1 if it succeeds
Handle where the release of buffers succeeded

If the mapping is not filled, it is an anonymous page which must be removed from the page cache

Take the LRU list lock, unlock the page, delete it from the page cache and free it

Update \texttt{nr\_pages} to show a page has been freed and move to the next page

If \texttt{nr\_pages} drops to 0, then exit the loop as the work is completed

If the page does have an associated mapping then simply drop the reference to the page and re-acquire the LRU lock

If the buffers could not be freed, then unlock the page, drop the reference to it, re-acquire the LRU lock and move to the next page

\begin{verbatim}
    spin_lock(&pagecache_lock);
    if (!page->mapping || !is_page_cache_freeable(page)) {
        spin_unlock(&pagecache_lock);
        UnlockPage(page);
    page_mapped:
        if (--max_mapped >= 0)
            continue;
    spin_unlock(&pagemap_lru_lock);
    swap_out(priority, gfp_mask, classzone);
    return nr_pages;
    }
\end{verbatim}

From this point on, pages in the swap cache are likely to be examined which is protected by the \texttt{pagecache\_lock} which must be now held

An anonymous page with no buffers is mapped by a process

Release the page cache lock and the page

Decrement \texttt{max\_mapped}. If it has not reached 0, move to the next page

Too many mapped pages have been found in the page cache. The LRU lock is released and \texttt{swap\_out}() is called to begin swapping out whole processes

\begin{verbatim}
    if (PageDirty(page)) {
        spin_unlock(&pagecache_lock);
        UnlockPage(page);
        continue;
    }
\end{verbatim}
The page has no references but could have been dirtied by the last process to free it if the dirty bit was set in the PTE. It is left in the page cache and will get laundered later. Once it has been cleaned, it can be safely deleted.

```c
/* point of no return */
if (likely(!PageSwapCache(page))) {
    __remove_inode_page(page);
    spin_unlock(&pagecache_lock);
} else {
    swp_entry_t swap;
    swap.val = page->index;
    __delete_from_swap_cache(page);
    spin_unlock(&pagecache_lock);
    swap_free(swap);
}

__lru_cache_del(page);
UnlockPage(page);

/* effectively free the page here */
page_cache_release(page);

if (--nr_pages)
    continue;
break;
```

500-503 If the page does not belong to the swap cache, it is part of the inode queue so it is removed.

504-508 Remove it from the swap cache as there is no more references to it.

511 Delete it from the page cache.

512 Unlock the page.

515 Free the page.

517-518 Decrement `nr_pages` and move to the next page if it is not 0.

519 If it reaches 0, the work of the function is complete.

```c
spin_unlock(&pagemap_lru_lock);
return nr_pages;
```

521-524 Function exit. Free the LRU lock and return the number of pages left to free.
7.6 Swapping Out Process Pages

Function: swap_out (mm/vmscan.c)

This function linearly searches through every process's page tables trying to swap out SWAP_CLUSTER_MAX number of pages. The process it starts with is the swap_mm and the starting address is mm->swap_address.

```
296 static int swap_out(unsigned int priority, unsigned int gfp_mask,
                zone_t * classzone)
297 {
298     int counter, nr_pages = SWAP_CLUSTER_MAX;
299     struct mm_struct *mm;
300     counter = mmlist_nr;
301     do {
302         if (unlikely(current->need_resched)) {
303             __set_current_state(TASK_RUNNING);
304             schedule();
305         }
306     }
307     spin_lock(&mmlist_lock);
308     mm = swap_mm;
309     while (mm->swap_address == TASK_SIZE || mm == &init_mm) {
310         mm->swap_address = 0;
311         mm = list_entry(mm->mmlist.next,
```
7.6. Swapping Out Process Pages

```c
struct mm_struct, mmlist);
    if (mm == swap_mm)
        goto empty;
    swap_mm = mm;
}

/* Make sure the mm doesn’t disappear
   when we drop the lock.. */
    atomic_inc(&mm->mm_users);
    spin_unlock(&mmlist_lock);
    nr_pages = swap_out_mm(mm, nr_pages, &counter, classzone);
    mmput(mm);
    if (!nr_pages)
        return 1;
    } while (--counter >= 0);
    return 0;
}

empty:
    spin_unlock(&mmlist_lock);
    return 0;
}
```

301 Set the counter so the process list is only scanned once

303-306 Reschedule if the quanta has been used up to prevent CPU hogging

308 Acquire the lock protecting the mm list

309 Start with the swap_mm. It is interesting this is never checked to make sure it is valid.
   It is possible, albeit unlikely that the mm has been freed since the last scan and the
   slab holding the mm_struct released making the pointer totally invalid. The lack of
   bug reports might be because the slab never managed to get freed up and would be
difficult to trigger

310-316 Move to the next process if the swap_address has reached the TASK_SIZE or if
   the mm is the init_mm

311 Start at the beginning of the process space

312 Get the mm for this process

313-314 If it is the same, there is no running processes that can be examined

315 Record the swap_mm for the next pass
Increase the reference count so that the mm does not get freed while we are scanning.

Release the mm lock.

Begin scanning the mm with `swap_out_mm()`.

Drop the reference to the mm.

If the required number of pages has been freed, return success.

If we failed on this pass, increase the priority so more processes will be scanned.

Return failure.

**Function:** `swap_out_mm` *(mm/vmscan.c)*

Walk through each VMA and call `swap_out_mm()` for each one.

```c
256 static inline int swap_out_mm(struct mm_struct * mm, int count, 
        int * mmcounter, zone_t * classzone)
257 {
    unsigned long address;
    struct vm_area_struct* vma;
260
    spin_lock(&mm->page_table_lock);
266    address = mm->swap_address;
267    if (address == TASK_SIZE || swap_mm != mm) {
        /* We raced: don’t count this mm but try again */
268        ***mmcounter;
270        goto out_unlock;
271    }
272    vma = find_vma(mm, address);
273    if (vma) {
274        if (address < vma->vm_start)
275            address = vma->vm_start;
276
277        for (;;) {
278            count = swap_out_vma(mm, vma, address,
            count, classzone);
279            vma = vma->vm_next;
280            if (!vma)
281                break;
282            if (!count)
283                goto out_unlock;
284            address = vma->vm_start;
285        }
286    }
287    /* Indicate that we reached the end of address space */
288    mm->swap_address = TASK_SIZE;
```
7.6. Swapping Out Process Pages

289
290 out_unlock:
291 spin_unlock(&mm->page_table_lock);
292 return count;
293 }

265 Acquire the page table lock for this mm
266 Start with the address contained in swap_address
267-271 If the address is TASK_SIZE, it means that a thread raced and scanned this process already. Increase mmcounter so that swap_out_mm() knows to go to another process.
272 Find the VMA for this address
273 Presuming a VMA was found then ....
274-275 Start at the beginning of the VMA
277-285 Scan through this and each subsequent VMA calling swap_out_vma() for each one. If the requisite number of pages (count) is freed, then finish scanning and return.
288 Once the last VMA has been scanned, set swap_address to TASK_SIZE so that this process will be skipped over by swap_out_mm() next time.

Function: swap_out_vma (mm/vmscan.c)
Walk through this VMA and for each PGD in it, call swap_out_pgd().

227 static inline int swap_out_vma(struct mm_struct * mm,
228          struct vm_area_struct * vma,
229          unsigned long address, int count,
230          zone_t * classzone)
231 {
232    pgd_t *pgdir;
233    unsigned long end;
234    /* Don’t swap out areas which are reserved */
235    if (vma->vm_flags & VM_RESERVED)
236        return count;
237    pgdir = pgd_offset(mm, address);
238    end = vma->vm_end;
239    BUG_ON(address >= end);
240    do {
241        count = swap_out_pgd(mm, vma, pgdir,
242                             address, end, count, classzone);
243        if (!count)
7.6. Swapping Out Process Pages

\[ \text{break;} \]
\[ \text{address = (address + PGDIR_SIZE) & PGDIR_MASK;} \]
\[ \text{pgdir++;} \]
\[ } \text{while (address && (address < end));} \]
\[ \text{return count;} \]
\[ } \]

233–234 Skip over this VMA if the VM_RESERVED flag is set. This is used by some device drivers such as the SCSI generic driver

236 Get the starting PGD for the address

238 Mark where the end is and BUG() it if the starting address is somehow past the end

240 Cycle through PGDs until the end address is reached

241 Call swap_out_pgd() keeping count of how many more pages need to be freed

242–243 If enough pages have been freed, break and return

244–245 Move to the next PGD and move the address to the next PGD aligned address

247 Return the remaining number of pages to be freed

Function: swap_out_pgd (mm/vmscan.c)
Step through all PMD’s in the supplied PGD and call swap_out_pmd()

197 static inline int swap_out_pgd(struct mm_struct * mm,
struct vm_area_struct * vma, pgd_t *dir,
unsigned long address, unsigned long end,
int count, zone_t * classzone)

198 {
199 pmd_t * pmd;
200 unsigned long pgd_end;
201
202 if (pgd_none(*dir))
203 return count;
204 if (pgd_bad(*dir)) {
205 pgd_ERROR(*dir);
206 pgd_clear(dir);
207 return count;
208 }
209
210 pmd = pmd_offset(dir, address);
211
212 pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
213 if (pgd_end && (end > pgd_end))
214 end = pgd_end;
7.6. Swapping Out Process Pages

215 do {
216     count = swap_out_pmd(mm, vma, pmd, address, end, count, classzone);
217     if (!count) break;
218     address = (address + PMD_SIZE) & PMD_MASK;
219     pmd++;
220 } while (address && (address < end));
221 return count;
222 }

202-203 If there is no PGD, return
204-208 If the PGD is bad, flag it as such and return
210 Get the starting PMD
212-214 Calculate the end to be the end of this PGD or the end of the VMA been scanned, whichever is closer
216-222 For each PMD in this PGD, call swap_out_pmd(). If enough pages get freed, break and return
223 Return the number of pages remaining to be freed

Function: swap_out_pmd (mm/vmscan.c)
For each PTE in this PMD, call try_to_swap_out(). On completion, mm->swap_address
is updated to show where we finished to prevent the same page been examined soon after
this scan.

158 static inline int swap_out_pmd(struct mm_struct * mm,
           struct vm_area_struct * vma, pmd_t *dir,
           unsigned long address, unsigned long end,
           int count, zone_t * classzone)
159 {
160     pte_t * pte;
161     unsigned long pmd_end;
162     if (pmd_none(*dir))
163         return count;
164     if (pmd_bad(*dir)) {
165         pmd_ERROR(*dir);
166         pmd_clear(dir);
167         return count;
168     }
169     pte = pte_offset(dir, address);

7.6. Swapping Out Process Pages

172  pmd_end = (address + PMD_SIZE) & PMD_MASK;
173  if (end > pmd_end)
174      end = pmd_end;
175
176  do {
177      if (pte_present(*pte)) {
178          struct page *page = pte_page(*pte);
179
180          if (VALID_PAGE(page) && !PageReserved(page)) {
181              count -= try_to_swap_out(mm, vma,
182                  address, pte,
183                  page, classzone);
184
185              if (!count) {
186                  address += PAGE_SIZE;
187                  break;
188              }
189          }
190      }
191      address += PAGE_SIZE;
192      pte++;
193  } while (address && (address < end));
194
195  mm->swap_address = address;
196  return count;
197

163-164 Return if there is no PMD
165-169 If the PMD is bad, flag it as such and return
171 Get the starting PTE
173-175 Calculate the end to be the end of the PMD or the end of the VMA, whichever is closer
177-191 Cycle through each PTE
178 Make sure the PTE is marked present
179 Get the struct page for this PTE
181 If it is a valid page and it is not reserved then ...
182 Call try_to_swap_out()
183-186 If enough pages have been swapped out, move the address to the next page and break to return
189-190 Move to the next page and PTE
7.6. Swapping Out Process Pages

192 Update the swap_address to show where we last finished off

193 Return the number of pages remaining to be freed

**Function: try_to_swap_out (mm/vmscan.c)**

This function tries to swap out a page from a process. It is quite a large function so will be dealt with in parts. Broadly speaking they are

- Function preamble, ensure this is a page that should be swapped out
- Remove the page and PTE from the page tables
- Handle the case where the page is already in the swap cache
- Handle the case where the page is dirty or has associated buffers
- Handle the case where the page has been added to the swap cache

```c
47 static inline int try_to_swap_out(struct mm_struct * mm,
    struct vm_area_struct* vma,
    unsigned long address,
    pte_t * page_table,
    struct page *page,
    zone_t * classzone)
{
    pte_t pte;
    swp_entry_t entry;

    /* Don’t look at this pte if it’s been accessed recently. */
    if ((vma->vm_flags & VM_LOCKED) ||
        ptep_test_and_clear_young(page_table)) {
        mark_page_accessed(page);
        return 0;
    }

    /* Don’t bother unmapping pages that are active */
    if (PageActive(page))
        return 0;

    /* Don’t bother replenishing zones not under pressure.. */
    if (!memclass(page_zone(page), classzone))
        return 0;

    if (TryLockPage(page))
        return 0;

    53-56 If the page is locked (for tasks like IO) or the PTE shows the page has been accessed recently then clear the referenced bit and call mark_page_accessed() to make the struct page reflect the age. Return 0 to show it was not swapped out
7.6. Swapping Out Process Pages

59-60 If the page is on the active_list, do not swap it out

63-64 If the page belongs to a zone we are not interested in, do not swap it out

66-67 If the page could not be locked, do not swap it out

74 Call the architecture hook to flush this page from all CPU’s

75 Get the PTE from the page tables and clear it

76 Call the architecture hook to flush the TLB

78-79 If the PTE was marked dirty, mark the struct page dirty so it will be laundered correctly

86 if (PageSwapCache(page)) {
    entry.val = page->index;
    swap_duplicate(entry);
    set_swap_pte:
    set_pte(page_table, swp_entry_to_pte(entry));
    drop_pte:
    mm->rss--;
    UnlockPage(page);
    {
        int freeable =
            page_count(page) - !!page->buffers <= 2;
        page_cache_release(page);
        return freeable;
    }
}

Handle the case where the page is already in the swap cache

87-88 Fill in the index value for the swap entry. swap_duplicate() verifies the swap identifier is valid and increases the counter in the swap_map if it is

90 Fill the PTE with information needed to get the page from swap

92 Update RSS to show there is one less page

93 Unlock the page
The page is free-able if the count is currently 2 or less and has no buffers

Decrement the reference count and free the page if it reaches 0

Return if the page was freed or not

115 if (page->mapping)
116     goto drop_pte;
117 if (!PageDirty(page))
118     goto drop_pte;
124 if (page->buffers)
125     goto preserve;

115–116 If the page has an associated mapping, simply drop it and it will be caught during another scan of the page cache later

117–118 If the page is clean, it is safe to simply drop it

124–125 If it has associated buffers due to a truncate followed by a page fault, then re-attach the page and PTE to the page tables as it can’t be handled yet

126
127 /*
128 * This is a dirty, swappable page. First of all,
129 * get a suitable swap entry for it, and make sure
130 * we have the swap cache set up to associate the
131 * page with that swap entry.
132 */
133 for (;;) {
134     entry = get_swap_page();
135     if (!entry.val)
136         break;
137     /* Add it to the swap cache and mark it dirty
138      * (adding to the page cache will clear the dirty
139      * and uptodate bits, so we need to do it again)
140      */
141     if (add_to_swap_cache(page, entry) == 0) {
142         SetPageUptodate(page);
143         set_page_dirty(page);
144         goto set_swap_pte;
145     }
146     /* Raced with "speculative" read_swap_cache_async */
147     swap_free(entry);
148 }
150 /* No swap space left */
151 preserve:
152     set_pte(page_table, pte);
UnlockPage(page);
return 0;
}

Allocate a swap entry for this page

If one could not be allocated, break out where the PTE and page will be re-attached to the process page tables

Add the page to the swap cache

Mark the page as up to date in memory

Mark the page dirty so that it will be written out to swap soon

Goto set_swap_pte which will update the PTE with information needed to get the page from swap later

If the add to swap cache failed, it means that the page was placed in the swap cache already by a readahead so drop the work done here

Reattach the PTE to the page tables

Unlock the page

Return that no page was freed
Chapter 8
Swap Management

8.1 Describing the Swap Area

8.2 Scanning for free entries

Figure 8.1: Call Graph: get_swap_page()

Function: get_swap_page (mm/swapfile.c)

This is the high level API function for getting a slot in swap space.

```c
99 swp_entry_t get_swap_page(void)
100 {
101     struct swap_info_struct * p;
102     unsigned long offset;
103     swp_entry_t entry;
104     int type, wrapped = 0;
105
106     entry.val = 0; /* Out of memory */
107     swap_list_lock();
108     type = swap_list.next;
109     if (type < 0)
110         goto out;
111     if (nr_swap_pages <= 0)
```
8.2. Scanning for free entries

112     goto out;
113
114     while (1) {
115         p = &swap_info[type];
116         if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
117             swap_device_lock(p);
118             offset = scan_swap_map(p);
119             swap_device_unlock(p);
120             if (offset) {
121                 entry = SWP_ENTRY(type,offset);
122                 type = swap_info[type].next;
123                 if (type < 0 ||
124                     p->prio != swap_info[type].prio) {
125                     swap_list.next = swap_list.head;
126                 } else {
127                     swap_list.next = type;
128                 }
129                 goto out;
130             }
131         }
132         type = p->next;
133         if (!wrapped) {
134             if (type < 0 || p->prio != swap_info[type].prio) {
135                 type = swap_list.head;
136                 wrapped = 1;
137             } else
138             if (type < 0)
139                 goto out;  /* out of swap space */
140         }
141     }
142 out:
143     swap_list_unlock();
144     return entry;
145 }

107 Lock the list of swap pages

108 Get the next swap area that is to be used for allocating from

109-110 If there is no swap areas, return NULL

111-112 If the accounting says there is no swap pages, return NULL

114-141 Cycle through all swap areas

115 Get the swap info struct

116 If this swap area is available for writing to and is active...
117 Lock the swap area
118 Call `scan_swap_map()` which searches for a free slot
119 Unlock the swap device
120-130 If a slot was free...
121 Encode an identifier for the entry with `SWP_ENTRY()`
122 Record the next swap area to use
123-126 If the next area is the end of the list or the priority of the next swap area does not match the current one, move back to the head
126-128 Otherwise move to the next area
129 Goto out
132 Move to the next swap area
133-138 Check for wraparound. Set `wrapped` to 1 if we get to the end of the list of swap areas
139-140 If there was no available swap areas, goto out
142 The exit to this function
143 Unlock the swap area list
144 Return the entry if one was found and NULL otherwise

**Function: scan_swap_map (mm/swapfile.c)**

This function tries to allocate `SWAPFILE.Cluster` number of pages sequentially in swap. When it has allocated that many, it searches for another block of free slots of size `SWAPFILE.Cluster`. If it fails to find one, it resorts to allocating the first free slot.

```c
36 static inline int scan_swap_map(struct swap_info_struct *si)
37 {
38     unsigned long offset;
39     if (si->cluster_nr) {
40         while (si->cluster_next <= si->highest_bit) {
41             offset = si->cluster_next++;
42             if (si->swap_map[offset])
43                 continue;
44             si->cluster_nr--;
45             goto got_page;
46         }
47     }
48 }
```
Allocate \texttt{SWAPFILE\_CLUSTER} pages sequentially. \texttt{cluster\_nr} is initialised to \texttt{SWAPFILE\_CLUSTER} and decrements with each allocation

47 If \texttt{cluster\_nr} is still positive, allocate the next available sequential slot

48 While the current offset to use (\texttt{cluster\_next}) is less than the highest known free slot (\texttt{highest\_bit}) then ...

49 Record the offset and update \texttt{cluster\_next} to the next free slot

50-51 If the slot is not actually free, move to the next one

52 Slot has been found, decrement the \texttt{cluster\_nr} field

53 Goto the out path

56 \texttt{si->cluster\_nr = SWAPFILE\_CLUSTER;}

57

58 /* try to find an empty (even not aligned) cluster. */

59 offset = si->lowest\_bit;

60 \texttt{check\_next\_cluster:}

61 \texttt{if (offset+SWAPFILE\_CLUSTER-1 <= si->highest\_bit)}

62 \{

63 \texttt{int nr;}

64 \texttt{for (nr = offset; nr < offset+SWAPFILE\_CLUSTER; nr++)}

65 \texttt{if (si->swap\_map[nr])}

66 \texttt{ }

67 \texttt{offset = nr+1;}

68 \texttt{goto check\_next\_cluster;}

69 \}

70 /* We found a completely empty cluster, so start

71 * using it.

72 */

73 \texttt{goto got\_page;}

74 \}

At this stage, \texttt{SWAPFILE\_CLUSTER} pages have been allocated sequentially so find the next free block of \texttt{SWAPFILE\_CLUSTER} pages.

56 Re-initialise the count of sequential pages to allocate to \texttt{SWAPFILE\_CLUSTER}

59 Starting searching at the lowest known free slot

61 If the offset plus the cluster size is less than the known last free slot, then examine all the pages to see if this is a large free block

64 Scan from \texttt{offset} to \texttt{offset + SWAPFILE\_CLUSTER}

65-69 If this slot is used, then start searching again for a free slot beginning after this known allocated one
A large cluster was found so use it

/* No luck, so now go finegrined as usual. -Andrea */
for (offset = si->lowest_bit; offset <= si->highest_bit ;
     offset++) {
  if (si->swap_map[offset])
    continue;
  si->lowest_bit = offset+1;
}

This unusual for loop extract starts scanning for a free page starting from lowest_bit

If the slot is in use, move to the next one

Update the lowest_bit known probable free slot to the succeeding one

got_page:
  if (offset == si->lowest_bit)
    si->lowest_bit++;
  if (offset == si->highest_bit)
    si->highest_bit--;
  if (si->lowest_bit > si->highest_bit) {
    si->lowest_bit = si->max;
    si->highest_bit = 0;
  }
  si->swap_map[offset] = 1;
  nr_swap_pages--;
  si->cluster_next = offset+1;
  return offset;
}

A slot has been found, do some housekeeping and return it

If this offset is the known lowest free slot(lowest_bit), increment it

If this offset is the highest known likely free slot, decrement it

If the low and high mark meet, the swap area is not worth searching any more so
set the low slot to be the highest possible slot and the high mark to 0 to cut down on
search time later. This will be fixed up by the next free

Set the reference count for the slot

Update the accounting for the number of available swap pages (nr_swap_pages)

Set cluster_next to the adjacent slot so the next search will start here

Return the free slot

No free slot available, mark the area unsearchable and return 0
8.3 Swap Cache

**Function: add_to_swap_cache** *(mm/swap_state.c)*

This function wraps around the normal page cache handler. It first checks if the page is already in the swap cache with `swap_duplicate()` and if it does not, it calls `add_to_page_cache_unique()` instead.

```c
70 int add_to_swap_cache(struct page *page, swp_entry_t entry)
71 {
72     if (page->mapping)
73         BUG();
74     if (!swap_duplicate(entry)) {
75         INC_CACHE_INFO(noent_race);
76         return -ENOENT;
77     }
78     if (add_to_page_cache_unique(page, &swapper_space, entry.val,
79         page_hash(&swapper_space, entry.val)) != 0) {
80         swap_free(entry);
81         INC_CACHE_INFO(exist_race);
82         return -EEXIST;
83     }
84     if (!PageLocked(page))
85         BUG();
86     if (!PageSwapCache(page))
87         BUG();
88     INC_CACHE_INFO(add_total);
89     return 0;
90 }
```

72-73 A check is made with `PageSwapCache()` before this function is called which ensures the page has no existing mapping. If code is calling this function directly, it should have ensured no existing mapping existed.

74-77 Try an increment the count for this entry with `swap_duplicate()`. If a slot already exists in the `swap_map`, increment the statistic recording the number of races involving adding pages to the swap cache and return `ENOENT`.

78 Try and add the page to the page cache with `add_to_page_cache_unique()`. This function is similar to `add_to_page_cache()` except it searches the page cache for a duplicate entry with `__find_page_nolock()`. The managing address space is `swapper_space`. The "offset within the file" in this case is the offset within `swap_map`, hence `entry.val` and finally the page is hashed based on `address_space` and offset within `swap_map`.

80-83 If it already existed in the page cache, we raced so increment the statistic recording the number of races to insert an existing page into the swap cache and return `EEXIST`.

84-85 If the page is locked for IO, it is a bug.
If it is not now in the swap cache, something went seriously wrong

Increment the statistic recording the total number of pages in the swap cache

Return success

**Function: swap_duplicate** *(mm/swapfile.c)*

This function verifies a swap entry is valid and if so, increments its swap map count.

```c
int swap_duplicate(swp_entry_t entry)
{
    struct swap_info_struct * p;
    unsigned long offset, type;
    int result = 0;

    type = SWP_TYPE(entry);
    if (type >= nr_swapfiles)
        goto bad_file;
    p = type + swap_info;
    offset = SWP_OFFSET(entry);

    swap_device_lock(p);
    if (offset < p->max && p->swap_map[offset]) {
        if (p->swap_map[offset] < SWAP_MAP_MAX - 1) {
            p->swap_map[offset]++;
            result = 1;
        } else if (p->swap_map[offset] <= SWAP_MAP_MAX) {
            if (swap_overflow++ < 5)
                printk(KERN_WARNING "swap_dup: swap entry overflow\n");
            p->swap_map[offset] = SWAP_MAP_MAX;
            result = 1;
        }
    }
    swap_device_unlock(p);
    return result;
}
bad_file:
    printk(KERN_ERR "swap_dup: %s%08lx\n", Bad_file, entry.val);
    goto out;
}
```

The parameter is the swap entry to increase the swap_map count for

Get the offset within the swap_info for the swap_info_struct containing this entry. If it is greater than the number of swap areas, goto bad_file
Get the relevant `swap_info_struct` and get the offset within its `swap_map`

Lock the swap device

Make a quick sanity check to ensure the offset is within the `swap_map` and that the slot indicated has a positive count. A 0 count would mean the slot is not free and this is a bogus `swp_entry_t`

If the count is not `SWAP_MAP_MAX`, simply increment it and return 1 for success

Else the count would overflow so set it to `SWAP_MAP_MAX` and reserve the slot permanently. In reality this condition is virtually impossible

Unlock the swap device and return

If a bad device was used, print out the error message and return failure

**Function: swap_free** *(mm/swapfile.c)*
Decrement the corresponding `swap_map` entry for the `swp_entry_t`

```c
void swap_free(swp_entry_t entry)
{
  struct swap_info_struct * p;
  p = swap_info_get(entry);
  if (p) {
    swap_entry_free(p, SWP_OFFSET(entry));
    swap_info_put(p);
  }
}
```

`swap_info_get()` fetches the correct `swap_info_struct` and performs a number of debugging checks to ensure it is a valid area and a valid `swap_map` entry. If all is sane, it will lock the swap device

If it is valid, the corresponding `swap_map` entry is decremented with `swap_entry_free()` and `swap_info_put` called to free the device

**Function: swap_entry_free** *(mm/swapfile.c)*

```c
static int swap_entry_free(struct swap_info_struct *p,
                           unsigned long offset)
{
  int count = p->swap_map[offset];
  if (count < SWAP_MAP_MAX) {
    count--;
    p->swap_map[offset] = count;
    if (!count) {
```
8.3. Swap Cache

200 if (offset < p->lowest_bit) 
201 p->lowest_bit = offset;
202 if (offset > p->highest_bit) 
203 p->highest_bit = offset;
204 nr_swap_pages++;
205 }
206 }
207 return count;
208 }

194 Get the current count
196 If the count indicates the slot is not permanently reserved then...
197-198 Decrement the count and store it in the swap_map
199 If the count reaches 0, the slot is free so update some information
200-201 If this freed slot is below lowest_bit, update lowest_bit which indicates the lowest known free slot
202-203 Similarly, update the highest_bit if this newly freed slot is above it
204 Increment the count indicating the number of free swap slots
207 Return the current count

Function: swap_info_get (mm/swapfile.c)
This function finds the swap_info_struct for the given entry, performs some basic checking and then locks the device.

147 static struct swap_info_struct * swap_info_get(swp_entry_t entry) 
148 { 
149 struct swap_info_struct * p;
150 unsigned long offset, type;
151 
152 if (!entry.val) 
153 goto out;
154 type = SWP_TYPE(entry);
155 if (type >= nr_swapfiles) 
156 goto bad_nofile;
157 p = & swap_info[type];
158 if (!(p->flags & SWP_USED)) 
159 goto bad_device;
160 offset = SWP_OFFSET(entry);
161 if (offset >= p->max) 
162 goto bad_offset;
163 if (!p->swap_map[offset])

8.3. Swap Cache

164     goto bad_free;
165     swap_list_lock();
166     if (p->prio > swap_info[swap_list.next].prio)
167         swap_list.next = type;
168     swap_device_lock(p);
169     return p;
170
171 bad_free:
172     printk(KERN_ERR "swap_free: %s%08lx\n", Unused_offset, entry.val);
173     goto out;
174 bad_offset:
175     printk(KERN_ERR "swap_free: %s%08lx\n", Bad_offset, entry.val);
176     goto out;
177 bad_device:
178     printk(KERN_ERR "swap_free: %s%08lx\n", Unused_file, entry.val);
179     goto out;
180 bad_nofile:
181     printk(KERN_ERR "swap_free: %s%08lx\n", Bad_file, entry.val);
182 out:
183     return NULL;
184 }

152-153 If the supplied entry is NULL, return
154 Get the offset within the swap_info array
155-156 Ensure it is a valid area
157 Get the address of the area
158-159 If the area is not active yet, print a bad device error and return
160 Get the offset within the swap_map
161-162 Make sure the offset is not after the end of the map
163-164 Make sure the slot is currently in use
165 Lock the swap area list
166-167 If this area is of higher priority than the area that would be next, ensure the current area is used
168-169 Lock the swap device and return the swap area descriptor
8.3. Swap Cache

Function: swap_info_put (mm/swapfile.c)
This function simply unlocks the area and list

```c
static void swap_info_put(struct swap_info_struct * p)
{
    swap_device_unlock(p);
    swap_list_unlock();
}
```

Unlock the device
Unlock the swap area list

Function: lookup_swap_cache (mm/swap_state.c)
Top level function for finding a page in the swap cache

```c
struct page * lookup_swap_cache(swp_entry_t entry)
{
    struct page *found;
    found = find_get_page(&swapper_space, entry.val);
    /*
     * Unsafe to assert PageSwapCache and mapping on page found:
     * if SMP nothing prevents swapoff from deleting this page from
     * the swap cache at this moment. find_lock_page would prevent
     * that, but no need to change: we _have_ got the right page.
     */
    INC_CACHE_INFO(find_total);
    if (found)
        INC_CACHE_INFO(find_success);
    return found;
}
```

find_get_page() is the principle function for returning the struct page. It uses
the normal page hashing and cache functions for quickly finding it

Increase the statistic recording the number of times a page was searched for in the
cache

If one was found, increment the successful find count

Return the struct page or NULL if it did not exist

Function: find_get_page (include/linux/pagemap.h)
Top level macro for finding a page in the page cache. It simply looks up the page hash

```c
#define find_get_page(mapping, index)  
    __find_get_page(mapping, index, page_hash(mapping, index))
```

page_hash() locates an entry in the page_hash_table based on the address_space
and offset
Function: \texttt{\_\_\_find\_get\_page} (\texttt{mm/filemap.c})
This function is responsible for finding a \texttt{struct page} given an entry in \texttt{page\_hash\_table} as a starting point.

\texttt{struct page * \_\_\_find\_get\_page(struct address\_space *mapping, unsigned long offset, struct page **hash)}

\begin{verbatim}
915 struct page * __find_get_page(struct address_space *mapping,
916                                      unsigned long offset, struct page **hash)
917 {
918     struct page *page;
919
920     /*
921      * We scan the hash list read-only. Addition to and removal from
922      * the hash-list needs a held write-lock.
923      */
924     spin_lock(&pagecache_lock);
925     page = __find_page_nolock(mapping, offset, *hash);
926     if (page)
927         page_cache_get(page);
928     spin_unlock(&pagecache_lock);
929     return page;
930 }
\end{verbatim}

924 Acquire the read-only page cache lock
925 Call the page cache traversal function which presumes a lock is held
926-927 If the page was found, obtain a reference to it with \texttt{page\_cache\_get()} so it is not freed prematurely
928 Release the page cache lock
929 Return the page or NULL if not found

Function: \texttt{\_\_\_find\_page\_nolock} (\texttt{mm/filemap.c})
This function traverses the hash collision list looking for the page specified by the \texttt{address\_space} and \texttt{offset}.

\begin{verbatim}
441 static inline struct page * __find_page_nolock(
     struct address_space *mapping,
     unsigned long offset,
     struct page *page)
442 {
443     goto inside;
444
445     for (;;)
446     {
447         page = page->next_hash;
448     inside:
449         if (!page)
450             goto not_found;
\end{verbatim}
8.4 Activating a Swap Area

Function: sys_swapon (mm/swapfile.c)

This, quite large, function is responsible for the activating of swap space. Broadly speaking the tasks it takes are as follows;

- Find a free swap_info_struct in the swap_info array and initialise it with default values
- Call user_path_walk() which traverses the directory tree for the supplied specialfile and populates a namidata structure with the available data on the file, such as the dentry and the filesystem information for where it is stored (vfsmount)
- Populate swap_info_struct fields pertaining to the dimensions of the swap area and how to find it. If the swap area is a partition, the block size will be configured to the PAGE_SIZE before calculating the size. If it is a file, the information is obtained directly from the inode
- Ensure the area is not already activated. If not, allocate a page from memory and read the first page sized slot from the swap area. This page contains information such as the number of good slots and how to populate the swap_info_struct→swap_map with the bad entries
- Allocate memory with vmalloc() for swap_info_struct→swap_map and initialise each entry with 0 for good slots and SWAP_MAP_BAD otherwise. Ideally the header
information will be a version 2 file format as version 1 was limited to swap areas of just under 128MiB for architectures with 4KiB page sizes like the x86\(^1\)

- After ensuring the information indicated in the header matches the actual swap area, fill in the remaining information in the `swap_info_struct` such as the maximum number of pages and the available good pages. Update the global statistics for `nr_swap_pages` and `total_swap_pages`

- The swap area is now fully active and initialised and so it is inserted into the swap list in the correct position based on priority of the newly activated area

855  asmlinkage long sys_swapon(const char * specialfile, int swap_flags)
856  {
857     struct swap_info_struct * p;
858     struct nameidata nd;
859     struct inode * swap_inode;
860     unsigned int type;
861     int i, j, prev;
862     int error;
863     static int least_priority = 0;
864     union swap_header *swap_header = 0;
865     int swap_header_version;
866     int nr_good_pages = 0;
867     unsigned long maxpages = 1;
868     int swapfilesize;
869     struct block_device *bdev = NULL;
870     unsigned short *swap_map;
871
872     if (!capable(CAP_SYS_ADMIN))
873         return -EPERM;
874     lock_kernel();
875     swap_list_lock();
876     p = swap_info;

875  The two parameters are the path to the swap area and the flags for activation
872-873  The activating process must have the `CAP_SYS_ADMIN` capability or be the super-user to activate a swap area
874  Acquire the Big Kernel Lock
875  Lock the list of swap areas
876  Get the first swap area in the `swap_info` array

\(^1\)See the Code Commentary for the comprehensive reason for this
for (type = 0 ; type < nr_swapfiles ; type++,p++)
  if (!(p->flags & SWP_USED))
    break;
error = -EPERM;
if (type >= MAX_SWAPFILES) {
  swap_list_unlock();
  goto out;
}
if (type >= nr_swapfiles)
  nr_swapfiles = type+1;
p->flags = SWP_USED;
p->swap_file = NULL;
p->swap_vfsmnt = NULL;
p->swap_device = 0;
p->swap_map = NULL;
p->lowest_bit = 0;
p->highest_bit = 0;
p->cluster_nr = 0;
p->sdev_lock = SPIN_LOCK_UNLOCKED;
p->next = -1;
if (swap_flags & SWAP_FLAG_PREFER) {
  p->prio =
    (swap_flags & SWAP_FLAG_PRIO_MASK)>>SWAP_FLAG_PRIO_SHIFT;
} else {
  p->prio = --least_priority;
}
swap_list_unlock();

Find a free swap_info_struct and initialise it with default values

877-879 Cycle through the swap_info until a struct is found that is not in use

880 By default the error returned is Permission Denied which indicates the caller did not have the proper permissions or too many swap areas are already in use

881 If no struct was free, MAX_SWAPFILE areas have already been activated so unlock the swap list and return

885-886 If the selected swap area is after the last known active area (nr_swapfiles), then update nr_swapfiles

887 Set the flag indicating the area is in use

888-896 Initialise fields to default values

897-902 If the caller has specified a priority, use it else set it to least_priority and decrement it. This way, the swap areas will be prioritised in order of activation
8.4. Activating a Swap Area

903 Release the swap list lock

904     error = user_path_walk(specialfile, &nd);
905     if (error)
906         goto bad_swap_2;
907
908     p->swap_file = nd.dentry;
909     p->swap_vfsmnt = nd.mnt;
910     swap_inode = nd.dentry->d_inode;
911     error = -EINVAL;

Traverse the VFS and get some information about the special file

904 user_path_walk() traverses the directory structure to obtain a nameidata structure describing the specialfile

905-906 If it failed, return failure

908 Fill in the swap_file field with the returned dentry

909 Similarly, fill in the swap_vfsmnt

910 Record the inode of the special file

911 Now the default error is EINVAL indicating that the special file was found but it was not a block device or a regular file

913     if (S_ISBLK(swap_inode->i_mode)) {
914         kdev_t dev = swap_inode->i_rdev;
915         struct block_device_operations *bdops;
916         devfs_handle_t de;
917
918         p->swap_device = dev;
919         set_blocksize(dev, PAGE_SIZE);
920
921         bd_acquire(swap_inode);
922         bdev = swap_inode->i_bdev;
923         de = devfs_get_handle_from_inode(swap_inode);
924         bdops = devfs_get_ops(de);
925         if (bdops) bdev->bd_op = bdops;
926
927         error = blkdev_get(bdev, FMODE_READ|FMODE_WRITE, 0,
928                             BDEV_SWAP);
929         devfs_put_ops(de);/*Decrement module use count now we’re safe*/
930     if (error)
930         goto bad_swap_2;
8.4. Activating a Swap Area

```c
set_blocksize(dev, PAGE_SIZE);
error = -ENOMEM;
if (!dev || (blk_size[MAJOR(dev)] &&
    !blk_size[MAJOR(dev)][MINOR(dev)]))
goto bad_swap;
swapfilesize = 0;
if (blk_size[MAJOR(dev)])
    swapfilesize = blk_size[MAJOR(dev)][MINOR(dev)]
    >> (PAGE_SHIFT - 10);
} else if (S_ISREG(swap_inode->i_mode))
    swapfilesize = swap_inode->i_size >> PAGE_SHIFT;
else
    goto bad_swap;
```

If a partition, configure the block device before calculating the size of the area, else obtain it from the inode for the file.

913 Check if the special file is a block device
914-939 This code segment handles the case where the swap area is a partition
914 Record a pointer to the device structure for the block device
918 Store a pointer to the device structure describing the special file which will be needed for block IO operations
919 Set the block size on the device to be PAGE_SIZE as it will be page sized chunks swap is interested in
921 The bd_acquire() function increments the usage count for this block device
922 Get a pointer to the block_device structure which is a descriptor for the device file which is needed to open it
923 Get a devfs handle if devfs is enabled. devfs is beyond the scope of this document
924-925 Increment the usage count of this device entry
927 Open the block device in read/write mode and set the BDEV_SWAP flag which is an enumerated type but is ignored when do_open() is called
928 Decrement the use count of the devfs entry
929-930 If an error occured on open, return failure
931 Set the block size again
932 After this point, the default error is to indicate no device could be found
933-935 Ensure the returned device is ok
8.4. Activating a Swap Area

937-939 Calculate the size of the swap file as the number of page sized chunks that exist in the block device as indicated by blk_size. The size of the swap area is calculated to make sure the information in the swap area is sane.

941 If the swap area is a regular file, obtain the size directly from the inode and calculate how many page sized chunks exist.

943 If the file is not a block device or regular file, return error.

945 error = -EBUSY;
946 for (i = 0 ; i < nr_swapfiles ; i++) {
947 struct swap_info_struct *q = &swap_info[i];
948 if (i == type || !q->swap_file)
949 continue;
950 if (swap_inode->i_mapping ==
951 q->swap_file->d_inode->i_mapping)
952 goto bad_swap;
953 }
954 swap_header = (void *)__get_free_page(GFP_USER);
955 if (!swap_header) {
956 printk("Unable to start swapping: out of memory :-)
957 error = -ENOMEM;
958 goto bad_swap;
959 }
960 lock_page(virt_to_page(swap_header));
961 rw_swap_page_nolock(READ, SWP_ENTRY(type,0),
962 (char *) swap_header);
963 if (!memcmp("SWAP-SPACE",swap_header->magic.magic,10))
964 swap_header_version = 1;
965 else if (!memcmp("SWAPSPACE2",swap_header->magic.magic,10))
966 swap_header_version = 2;
967 else {
968 printk("Unable to find swap-space signature\n感");
969 error = -EINVAL;
970 goto bad_swap;
971 }
972
945 The next check makes sure the area is not already active. If it is, the error EBUSY will be returned.

946-962 Read through the while swap_info struct and ensure the area to be activated is not already active.

954-959 Allocate a page for reading the swap area information from disk.
The function `lock_page()` locks a page and makes sure it is synced with disk if it is file backed. In this case, it’ll just mark the page as locked which is required for the `rw_swap_page_nolock()` function.

Read the first page slot in the swap area into `swap_header`.

Decide which version the swap area information is and set the `swap_header_version` variable with it. If the swap area could not be identified, return `EINVAL`.

```c
switch (swap_header_version) {
    case 1:
        memset(((char *) swap_header)+PAGE_SIZE-10,0,10);
        j = 0;
        p->lowest_bit = 0;
        p->highest_bit = 0;
        for (i = 1 ; i < 8*PAGE_SIZE ; i++) {
            if (test_bit(i,(char *) swap_header)) {
                if (!p->lowest_bit)
                    p->lowest_bit = i;
                p->highest_bit = i;
                maxpages = i+1;
                j++;
            }
        }
        nr_good_pages = j;
        p->swap_map = vmalloc(maxpages * sizeof(short));
        if (!p->swap_map) {
            error = -ENOMEM;
            goto bad_swap;
        }
        for (i = 1 ; i < maxpages ; i++) {
            if (test_bit(i,(char *) swap_header))
                p->swap_map[i] = 0;
            else
                p->swap_map[i] = SWAP_MAP_BAD;
        }
        break;
    }
    break;
}
```

Read in the information needed to populate the `swap_map` when the swap area is version 1.

Zero out the magic string identifying the version of the swap area.

Initialise fields in `swap_info_struct` to 0.

A bitmap with 8*PAGE_SIZE entries is stored in the swap area. The full page, minus 10 bits for the magic string, is used to describe the swap map limiting swap...
8.4. Activating a Swap Area

areas to just under 128MiB in size. If the bit is set to 1, there is a slot on disk available. This pass will calculate how many slots are available so a swap_map may be allocated.

981 Test if the bit for this slot is set

982-983 If the lowest_bit field is not yet set, set it to this slot. In most cases, lowest_bit will be initialised to 1

984 As long as new slots are found, keep updating the highest_bit

985 Count the number of pages

986 j is the count of good pages in the area

990 Allocate memory for the swap_map with vmalloc()

991-994 If memory could not be allocated, return ENOMEM

995-1000 For each slot, check if the slot is “good”. If yes, initialise the slot count to 0, else set it to SWAP_MAP_BAD so it will not be used

1001 Exit the switch statement

1003 case 2:
1006 if (swap_header->info.version != 1) {
1007 printk(KERN_WARNING
1008 "Unable to handle swap header version %d\n",
1009 swap_header->info.version);
1010 error = -EINVAL;
1011 goto bad_swap;
1012 }
1013
1014 p->lowest_bit = 1;
1015 maxpages = SWP_OFFSET(SWP_ENTRY(0,~0UL)) - 1;
1016 if (maxpages > swap_header->info.last_page)
1017 maxpages = swap_header->info.last_page;
1018 p->highest_bit = maxpages - 1;
1019
1020 error = -EINVAL;
1021 if (swap_header->info.nr_badpages > MAX_SWAP_BADPAGES)
1022 goto bad_swap;
1023
1025 if (!(p->swap_map = vmalloc(maxpages * sizeof(short)))) {
1026 error = -ENOMEM;
1027 goto bad_swap;
1028 }
1029
1030 error = 0;
8.4. Activating a Swap Area

1031     memset(p->swap_map, 0, maxpages * sizeof(short));
1032     for (i=0; i<swap_header->info.nr_badpages; i++) {
1033         int page = swap_header->info.badpages[i];
1034         if (page <= 0 ||
1035             page >= swap_header->info.last_page)
1036             error = -EINVAL;
1037         else
1038             p->swap_map[page] = SWAP_MAP_BAD;
1039     }
1039     nr_good_pages = swap_header->info.last_page -
1040     swap_header->info.nr_badpages -
1041     1 /* header page */;
1042     if (error)
1043         goto bad_swap;
1044 }

Read the header information when the file format is version 2

1006-1012 Make absolutely sure we can handle this swap file format and return EINVAL if we cannot. Remember that with this version, the swap_header struct is placed nicely on disk

1014 Initialise lowest_bit to the known lowest available slot

1015-1017 Calculate the maxpages initially as the maximum possible size of a swap_map and then set it to the size indicated by the information on disk. This ensures the swap_map array is not accidently overloaded

1018 Initialise highest_bit

1020-1022 Make sure the number of bad pages that exist does not exceed MAX_SWAP_BADPAGES

1025-1028 Allocate memory for the swap_map with vmalloc()

1031 Initialise the full swap_map to 0 indicating all slots are available

1032-1038 Using the information loaded from disk, set each slot that is unusuable to SWAP_MAP_BAD

1039-1041 Calculate the number of available good pages

1042-1043 Return if an error occurred

1045
1046     if (swapfilesize && maxpages > swapfilesize) {
1047         printk(KERN_WARNING
1048             "Swap area shorter than signature indicates\n");
1049         error = -EINVAL;
1050         goto bad_swap;
8.4. Activating a Swap Area

1051 } }  
1052 if (!nr_good_pages) {  
1053 printk(KERN_WARNING "Empty swap-file\n");  
1054 error = -EINVAL;  
1055 goto bad_swap;  
1056 }  
1057 p->swap_map[0] = SWAP_MAP_BAD;  
1058 swap_list_lock();  
1059 swap_device_lock(p);  
1060 p->max = maxpages;  
1061 p->flags = SWP_WRITEOK;  
1062 p->pages = nr_good_pages;  
1063 nr_swap_pages += nr_good_pages;  
1064 total_swap_pages += nr_good_pages;  
1065 printk(KERN_INFO "Adding Swap: %dk swap-space (priority %d)\n",  
1066 nr_good_pages<<(PAGE_SHIFT-10), p->prio);  

1046-1051 Ensure the information loaded from disk matches the actual dimensions of the swap area. If they do not match, print a warning and return an error  

1052-1056 If no good pages were available, return an error  

1057 Make sure the first page in the map containing the swap header information is not used. If it was, the header information would be overwritten the first time this area was used  

1058-1059 Lock the swap list and the swap device  

1060-1062 Fill in the remaining fields in the swap_info_struct  

1063-1064 Update global statistics for the number of available swap pages (nr_swap_pages) and the total number of swap pages (total_swap_pages)  

1065-1066 Print an informational message about the swap activation  

1068 /* insert swap space into swap_list: */  
1069 prev = -1;  
1070 for (i = swap_list.head; i >= 0; i = swap_info[i].next) {  
1071 if (p->prio >= swap_info[i].prio) {  
1072 break;  
1073 }  
1074 prev = i;  
1075 }  
1076 p->next = i;  
1077 if (prev < 0) {  
1078 swap_list.head = swap_list.next = p - swap_info;  
1079 } else {  
1080 swap_info[prev].next = p - swap_info;
Insert the new swap area into the correct slot in the swap list based on priority

Unlock the swap device

Unlock the swap list

Return success

Drop the reference to the block device

This is the error path where the swap list need to be unlocked, the slot in swap_info reset to being unused and the memory allocated for swap_map freed if it was assigned

Drop the reference to the special file
1106-1107 Release the page containing the swap header information as it is no longer needed

1108 Drop the Big Kernel Lock

1109 Return the error or success value

8.5 Deactivating a Swap Area

Function: sys_swapoff (mm/swapfile.c)

This function is principally concerned with updating the swap_info_struct and the swap lists. The main task of paging in all pages in the area is the responsibility of try_to_unuse(). The function tasks are broadly

- Call user_path_walk() to acquire the information about the special file to be deactivated and then take the BKL

- Remove the swap_info_struct from the swap list and update the global statistics on the number of swap pages available (nr_swap_pages) and the total number of swap entries (total_swap_pages). Once this is acquired, the BKL can be released again

- Call try_to_unuse() which will page in all pages from the swap area to be deactivated.

- If there was not enough available memory to page in all the entries, the swap area is reinserted back into the running system as it cannot be simply dropped. If it succeeded, the swap_info_struct is placed into an uninitialised state and the swap_map memory freed with vfree()

707 asmlinkage long sys_swapoff(const char * specialfile)
708 {
709    struct swap_info_struct * p = NULL;
710    unsigned short *swap_map;
711    struct nameidata nd;
712    int i, type, prev;
713    int err;
714
715    if (!capable(CAP_SYS_ADMIN))
716        return -EPERM;
717
718    err = user_path_walk(specialfile, &nd);
719    if (err)
720        goto out;
721
715-716 Only the superuser or a process with CAP_SYS_ADMIN capabilities may deactivate an area
Deactivating a Swap Area

718-719 Acquire information about the special file representing the swap area with user_path_walk(). Return on error

722-724 Lock the swap list

725-732 Traverse the swap list and find the swap_info_struct for the requested area. Use the dentry to identify the area

734-737 If the struct could not be found, return

739-747 Remove from the swap list making sure that this is not the head

Acquire the BKL, find the swap_info_struct for the area to be deactivated and remove it from the swap list.
8.5. Deactivating a Swap Area

748 Update the total number of free swap slots

749 Update the total number of existing swap slots

750 Mark the area as active but may not be written to

751 swap_list_unlock();
752 unlock_kernel();
753 err = try_to_unuse(type);

751 Unlock the swap list

752 Release the BKL

753 Page in all pages from this swap area

754 lock_kernel();
755 if (err) {
    /* re-insert swap space back into swap_list */
756     swap_list_lock();
757     for (prev = -1, i = swap_list.head; i >= 0; prev = i, i =
        swap_info[i].next)
758         if (p->prio >= swap_info[i].prio)
759             break;
760     p->next = i;
    if (prev < 0)
762     swap_list.head = swap_list.next = p - swap_info;
764 else
765     swap_info[prev].next = p - swap_info;
766     nr_swap_pages += p->pages;
767     total_swap_pages += p->pages;
768     p->flags = SWP_WRITEOK;
769     swap_list_unlock();
770     goto out_dput;
771 }

    Acquire the BKL. If we failed to page in all pages, then reinsert the area into the swap
    list

754 Acquire the BKL

757 Lock the swap list

758-765 Reinsert the area into the swap list. The position it is inserted at depends on the
    swap area priority

766-767 Update the global statistics

768 Mark the area as safe to write to again
Unlock the swap list and return

```c
if (p->swap_device)
    blkdev_put(p->swap_file->d_inode->i_bdev, BDEV_SWAP);
path_release(&nd);
swap_list_lock();
swap_device_lock(p);
nd.mnt = p->swap_vfsmnt;
nd.dentry = p->swap_file;
p->swap_vfsmnt = NULL;
p->swap_file = NULL;
p->swap_device = 0;
p->max = 0;
swap_map = p->swap_map;
p->swap_map = NULL;
p->flags = 0;
swap_device_unlock(p);
swap_list_unlock();
vfree(swap_map);
err = 0;
out_dput:
    unlock_kernel();
    path_release(&nd);
out:
    return err;
}
```

Else the swap area was successfully deactivated to close the block device and mark the swap_info_struct free

Close the block device
Release the path information
Acquire the swap list and swap device lock
Reset the fields in swap_info_struct to default values
Release the swap list and swap device
Free the memory used for the swap_map
Release the BKL
Release the path information in the event we reached here via the error path
Return success or failure
8.5. Deactivating a Swap Area

Function: try_to_unuse (mm/swapfile.c)

This function is heavily commented in the source code albeit it consists of speculation or is slightly inaccurate. The comments are omitted here.

```c
513 static int try_to_unuse(unsigned int type)
514 {
515     struct swap_info_struct * si = &swap_info[type];
516     struct mm_struct *start_mm;
517     unsigned short *swap_map;
518     unsigned short swcount;
519     struct page *page;
520     swp_entry_t entry;
521     int i = 0;
522     int retval = 0;
523     int reset_overflow = 0;
524     start_mm = &init_mm;
525     atomic_inc(&init_mm.mm_users);

539-540 The starting mm_struct to page in pages for is init_mm. The count is incremented even though this particular struct will not disappear to prevent having to write special cases in the remainder of the function

555     while ((i = find_next_to_unuse(si, i))) {
556         /*
557         * Get a page for the entry, using the existing swap
558         * cache page if there is one. Otherwise, get a clean
559         * page and read the swap into it.
560         */
561         swap_map = &si->swap_map[i];
562         entry = SWP_ENTRY(type, i);
563         page = read_swap_cache_async(entry);
564         if (!page) {
565             if (!*swap_map)
566                 continue;
567             retval = -ENOMEM;
568             break;
569         }
570         /*
571         * Don’t hold on to start_mm if it looks like exiting.
572         */
573         if (atomic_read(&start_mm->mm_users) == 1) {
574             mmput(start_mm);
575             start_mm = &init_mm;
```
This is the beginning of the major loop in this function. Starting from the beginning of the swap_map, it searches for the next entry to be freed with find_next_to_unuse() until all swap map entries have been paged in.

Get the swp_entry_t and call read_swap_cache_async() to find the page in the swap cache or have a new page allocated for reading in from the disk.

If we failed to get the page, it means the slot has already been freed independently by another process or thread (process could be exiting elsewhere) or we are out of memory. If independently freed, we continue to the next map, else we return ENOMEM.

Check to make sure this mm is not exiting. If it is, decrement its count and go back to init_mm.

/*
 * Wait for and lock page. When do_swap_page races with
 * try_to_unuse, do_swap_page can handle the fault much
 * faster than try_to_unuse can locate the entry. This
 * apparently redundant "wait_on_page" lets try_to_unuse
 * defer to do_swap_page in such a case - in some tests,
 * do_swap_page and try_to_unuse repeatedly compete.
 */
wait_on_page(page);
lock_page(page);

/*
 * Remove all references to entry, without blocking.
 * Whenever we reach init_mm, there’s no address space
 * to search, but use it as a reminder to search shmem.
 */
swcount = *swap_map;
if (swcount > 1) {
    flush_page_to_ram(page);
    if (start_mm == &init_mm)
        shmem_unuse(entry, page);
    else
        unuse_process(start_mm, entry, page);
}

Wait on the page to complete IO. Once it returns, we know for a fact the page exists in memory with the same information as that on disk.

Lock the page.

Get the swap map reference count.
If the count is positive then...

As the page is about to be inserted into process page tables, it must be freed from the D-Cache or the process may not “see” changes made to the page by the kernel.

If we are using the init_mm, call shmem_unuse() which will free the page from any shared memory regions that are in use.

Else update the PTE in the current mm which references this page.

```c
if (*swap_map > 1) {
    int set_start_mm = (*swap_map >= swcount);
    struct list_head *p = &start_mm->mmlist;
    struct mm_struct *new_start_mm = start_mm;
    struct mm_struct *mm;

    spin_lock(&mmlist_lock);
    while (*swap_map > 1 &&
        (p = p->next) != &start_mm->mmlist) {
        mm = list_entry(p, struct mm_struct,
                        mmlist);
        swcount = *swap_map;
        if (mm == &init_mm) {
            set_start_mm = 1;
            shmem_unuse(entry, page);
        } else {
            unuse_process(mm, entry, page);
            if (set_start_mm && *swap_map < swcount) {
                new_start_mm = mm;
                set_start_mm = 0;
            }
        }
        atomic_inc(&new_start_mm->mm_users);
    }
    spin_unlock(&mmlist_lock);
    mmput(start_mm);
    start_mm = new_start_mm;
}
```

If an entry still exists, begin traversing through all mm_structs finding references to this page and update the respective PTE.

Lock the mm list.

Keep searching until all mm_structs have been found. Do not traverse the full list more than once.

Get the mm_struct for this list entry.
8.5. Deactivating a Swap Area

621-625 Call shmem_unuse() if the mm is init_mm, else call unuse_process() to traverse the process page tables and update the PTE.

626-627 Record if we need to start searching mm_structs starting from init_mm again.

650-657 If the swap map entry is permanently mapped, we have to hope that all processes have their PTEs updated to point to the page and in reality the swap map entry is free. In reality, it is highly unlikely a slot would be permanently reserved in the first place.

641-657 Lock the list and swap device, set the swap map entry to 1, unlock them again and record that a reset overflow occurred.

674-679 In the very rare event a reference still exists to the page, write the page back to disk so at least if another process really has a reference to it, it’ll copy the page back in from disk correctly.

678-679 Delete the page from the swap cache so the page swap daemon will not use the page under any circumstances.

686-688 Mark the page dirty so that the swap out code will preserve the page and if it needs to remove it again, it’ll write it correctly to a new swap area.

687 Unlock the page.

688 Release our reference to it in the page cache.
if (current->need_resched)
    schedule();
}

mmput(start_mm);
if (reset_overflow) {
    printk(KERN_WARNING "swapoff: cleared swap entry
        overflow\n");
    swap_overflow = 0;
}
return retval;

Call schedule() if necessary so the deactivation of swap does not hog the entire CPU

Drop our reference to the mm

If a permanently mapped page had to be removed, then print out a warning so that in the very unlikely event an error occurs later, there will be a hint to what might have happened

Return success or failure
Index

activate_lock, 237
activate_page_nolock, 237
add_page_to_active_list, 234
add_page_to_inactive_list, 235
__add_to_page_cache, 238
add_to_page_cache, 238
add_to_swap_cache, 267
address_space, 126
alloc_area_pmd, 51
alloc_area_pte, 52
alloc_bootmem, 15
__alloc_bootmem, 16
__alloc_bootmem_node, 16
alloc_bootmem_node, 16
__alloc_bootmem_node, 17
alloc_bounce_bh, 222
alloc_bounce_page, 223
alloc_one_pte, 167
alloc_page, 43
alloc_pages, 29
__alloc_pages, 30
__alloc_pages, 31
allocate_mm, 119
arch_get_unmapped_area, 131, 144

bootmem_data, 8
bounce_end_io, 226
bounce_end_io_read, 225
bounce_end_io_write, 225
BREAK_GFP_ORDER_HI, 103
BREAK_GFP_ORDER_LO, 103

cache_cache, 114
CACHE_NAMELEN, 103
calc_vm_flags, 131
can_vma_merge, 153
cc_data, 106
cc_entry, 107
ccupdate_t, 111

CHECK_PAGE, 98
clock_searchp, 75
contig_page_data, 30
copy_from_high_bh, 225
copy_mm, 120
copy_one_pte, 167
copy_to_high_bh_irq, 226
cpu_vm_mask, 119
cpucache, 106
create_buffers, 220
def_flags, 119
DEFAULT_MAX_MAP_COUNT, 131
del_page_from_active_list, 236
del_page_from_inactive_list, 236
do_anonymous_page, 204
do_ccupdate_local, 111, 112
do_mlock, 170
do_mmap2, 128
do_mmap_pgoff, 128
do_mremap, 154
do_munmap, 179
do_no_page, 200
do_swap_page, 206
drain_cpu_caches, 112

enable_all_cpucaches, 107
enable_cpucache, 107, 108
exit_mmap, 123, 186
expand, 35, 37
find_get_page, 272
__find_get_page, 273
__find_page_nolock, 273
find_vma, 139
find_vma_intersection, 142
find_vma_prepare, 146
find_vma_prev, 141
flush_all_zero_pkmaps, 216

294
INDEX

free_all_bootmem, 26
free_all_bootmem_core, 27
free_area_pmd, 56
free_block, 101
__free_block, 102
free_bootmem, 22
free_bootmem_core, 23
__free_page, 45
__free_pages, 39
free_pages, 39, 45
free_pages_init, 26
__free_pages_ok, 39
g_cpu_cache_up, 107
__get_dma_pages, 44
__get_free_page, 43
__get_free_pages, 44
get_one_pte, 166
get_swap_page, 262
get_unmapped_area, 142
get_vm_area, 48
get_zeroed_page, 44
gfp_mask, 30
handle_mm_fault, 198
handle_pte_fault, 199
__init, 25
init_bootmem, 11
init_bootmem_core, 12
init_bootmem_node, 12
init_emergency_pool, 228
INIT_MM, 120
init_mm, 120
__insert_vm_struct, 145
kfree, 105
km_type, 218
kmalloc, 104
kmap, 212
kmap_atomic, 218
kmap_high, 213
kmem_cache, 114
__kmem_cache_alloc (SMP Case), 92
__kmem_cache_alloc (UP Case), 91
kmem_cache_alloc, 90
kmem_cache_alloc_batch, 96
kmem_cache_alloc_head, 93
kmem_cache_alloc_one, 94
kmem_cache_alloc_one_tail, 95
kmem_cache_create, 59
kmem_cache_destroy, 73
kmem_cache_estimate, 68
kmem_cache_free, 97
__kmem_cache_free, 98, 99
kmem_cache_free_one, 100
kmem_cache_grow, 83
kmem_cache_init, 114
kmem_cache_init_objs, 88
kmem_cache_reap, 76
kmem_cache_shrink, 70
__kmem_cache_shrink, 71
__kmem_cache_shrink_locked, 72
kmem_cache_sizes_init, 102
kmem_cache_slabmgmt, 81
kmem_find_general_cachep, 82
kmem_freepages, 116
kmem_getpages, 115
kmem_slab_destroy, 87
kmem_tune_cpu_cache, 107, 109
kswapd, 230
kswapd_balance, 232
kswapd_balance_pgd, 233
kswapd_can_sleep, 231
kswapd_can_sleep_pgd, 232
kswapd_init, 230
kunmap, 217
kunmap_atomic, 219
kunmap_high, 217
locked_vm, 119
lookup_swap_cache, 272
lru_cache_add, 234
lru_cache_del, 235
__lru_cache_del, 236
map_new_virtual, 213
mark_page_accessed, 237
max_map_count, 131
mem_init, 24
mlock_fixup, 173
mlock_fixup_all, 174
INDEX

mlock_fixup_end, 176
mlock_fixup_middle, 177
mlock_fixup_start, 175
mm_alloc, 119, 120
mm_init, 119, 122
mmmap_sem, 118
mmdrop, 123, 124
_mmdrop, 124
mm_list, 119
mmput, 123
move_one_page, 165
move_page_tables, 164
move_vma, 159

one_highpage_init, 26
Page Frame Number (PFN), 9
page_cache_get, 238
page_cache_release, 238
ptep_get_and_clear, 58
REAP_SCANLEN, 75
refill_inactive, 243
reserve_bootmem, 13
reserve_bootmem_core, 14
reserve_bootmem_node, 14
rmqueue, 35
rss, 119

scan_swap_map, 264
SET_PAGE_CACHE, 86
SET_PAGE_SLAB, 86
setup_memory, 8
shrink_cache, 244
shrink_caches, 239
slab_break_gfp_order, 103
smp_function_all_cpus, 111
STATS_INC_GROWN, 86
swap_duplicate, 268
swap_entry_free, 269
swap_free, 269
swap_info_get, 270
swap_info_put, 272
swap_out, 251
swap_out_mm, 253
swap_out_pgd, 255
swap_out_pmd, 256
swap_out_vma, 254
SWP_ENTRY, 264
sys_mlock, 168
sys_mlockall, 169
sys_mmap2, 128
sys_mremap, 153
sys_munlock, 172
sys_munlockall, 173
sys_swapoff, 285
sys_swapon, 274
total_vm, 119
try_to_free_pages, 241
try_to_free_pages_zone, 242
try_to_swap_out, 258
try_to_unuse, 289
unmap_fixup, 183
vfree, 53
vma_link, 147
__vma_link, 148
__vma_link_file, 150
__vma_link_list, 149
__vma_link_rb, 149
vma_merge, 150
vmalloc, 46
__vmalloc, 47
vmalloc_area_pages, 50
vmfree_area_pages, 55