Code Commentary On The Linux Virtual Memory Manager

Mel Gorman

14th July 2003

Contents

List of Figures

List of Tables

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`>;

```
typedef struct bootmem_data {
        unsigned long node_boot_start;
        unsigned long node_low_pfn;
        void *node_bootmem_map;
        unsigned long last_offset;
        unsigned long last_pos;
} 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*)

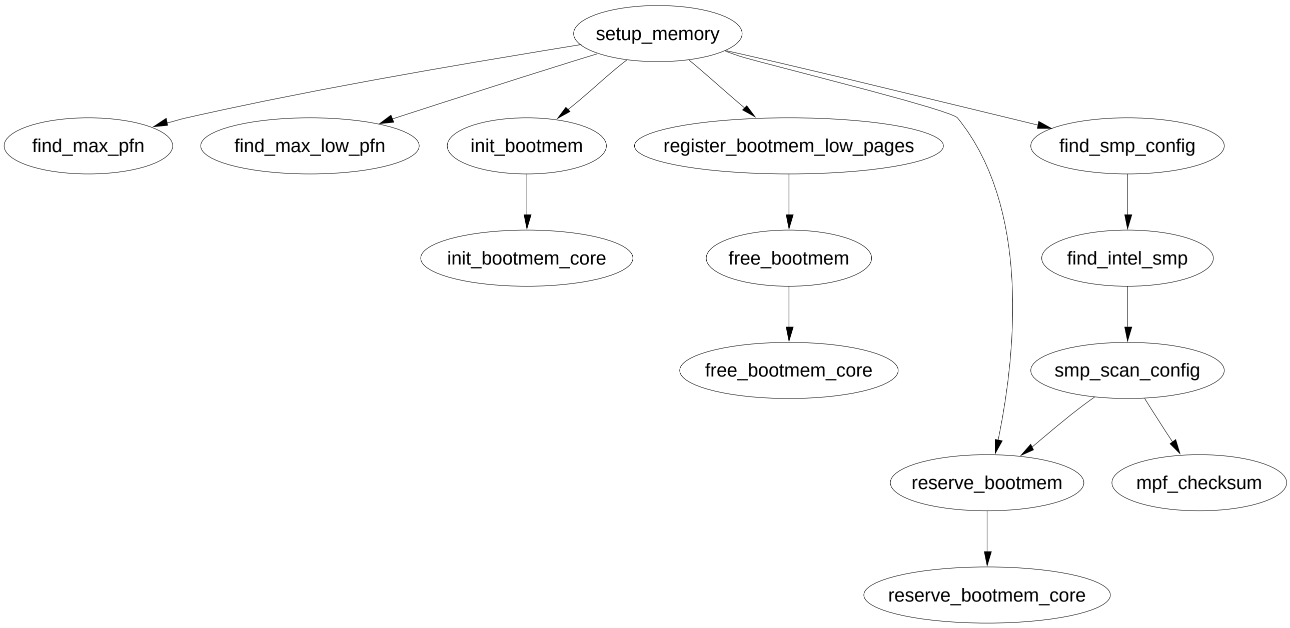

Figure 1.1: Call Graph: setup_memory

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 (`min_low_pfn`, `max_low_pfn`), the start and end PFN for high memory (`highstart_pfn`, `highend_pfn`) and the PFN for the last page in the system (`max_pfn`).
- Initialise the `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

```
static unsigned long __init setup_memory(void)
{
        unsigned long bootmap_size, start_pfn, max_low_pfn;

        /*
         * partially used pages are not usable - thus
         * we are rounding upwards:
         */
        start_pfn = PFN_UP(__pa(&_end));

        find_max_pfn();

```

```
961         max_low_pfn = find_max_low_pfn();
962
963 #ifdef CONFIG_HIGHMEM
964         highstart_pfn = highend_pfn = max_pfn;
965         if (max_pfn > max_low_pfn) {
966                 highstart_pfn = max_low_pfn;
967         }
968         printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
969                 pages_to_mb(highend_pfn - highstart_pfn));
970 #endif
971         printk(KERN_NOTICE "%ldMB LOWMEM available.\n",
972                         pages_to_mb(max_low_pfn));
```

957 `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

959 `find_max_pfn()` loops through the e820 map searching for the highest available pfn

961 `find_max_low_pfn()` finds the highest page frame addressable in `ZONE_NORMAL`

964-969 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

971-972 Print out an informational message on the amount of low memory

```
976         bootmap_size = init_bootmem(start_pfn, max_low_pfn);
977
978         register_bootmem_low_pages(max_low_pfn);
979
986         reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +
987                         bootmap_size + PAGE_SIZE-1) - (HIGH_MEMORY));
988
993         reserve_bootmem(0, PAGE_SIZE);
994
995 #ifdef CONFIG_SMP
1001         reserve_bootmem(PAGE_SIZE, PAGE_SIZE);
1002 #endif
```

976 `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 representing the pages and sets all pages as reserved

978 `registed_bootmem_low_pages()` reads the e820 map and calls `free_bootmem()` for all usable pages in the running system

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 <=
                    (max_low_pfn << PAGE_SHIFT)) {
1013                   reserve_bootmem(INITRD_START, INITRD_SIZE);
1014                   initrd_start =
1015                    INITRD_START ? INITRD_START + PAGE_OFFSET : 0;
1016                   initrd_end = initrd_start+INITRD_SIZE;
1017             }
1018             else {
1019                   printk(KERN_ERR
                           "initrd extends beyond end of memory "
1020                       "(0x%08lx > 0x%08lx)\ndisabling initrd\n",
1021                       INITRD_START + INITRD_SIZE,
1022                       max_low_pfn << PAGE_SHIFT);
1023                   initrd_start = 0;
1024             }
1025       }
1026 #endif
1027
1028       return max_low_pfn;
1029 }
```

1008 This function reserves memory that stores config information about the SMP setup

1010-1026 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.

```
304 unsigned long __init init_bootmem (unsigned long start,
                                 unsigned long pages)
305 {
306         max_low_pfn = pages;
307         min_low_pfn = start;
308         return(init_bootmem_core(&contig_page_data, start, 0, pages));
309 }
```

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

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

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

308 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

```
284 unsigned long __init init_bootmem_node (pg_data_t *pgdat,
                                      unsigned long freepfn,
                                      unsigned long startpfn,
                                      unsigned long endpfn)
285 {
286         return(init_bootmem_core(pgdat, freepfn, startpfn, endpfn));
287 }
```

286 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`.

```
 46 static unsigned long __init init_bootmem_core (pg_data_t *pgdat,
 47         unsigned long mapstart, unsigned long start, unsigned long end)
 48 {
 49         bootmem_data_t *bdata = pgdat->bdata;
 50         unsigned long mapsize = ((end - start)+7)/8;
 51 
 52         pgdat->node_next = pgdat_list;
 53         pgdat_list = pgdat;
 54 
 55         mapsize = (mapsize + (sizeof(long) - 1UL)) &
                    ~(sizeof(long) - 1UL);
 56         bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT);
```

```
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  }
```

46 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 (*mm/bootmem.c*)

```
311 void __init reserve_bootmem (unsigned long addr, unsigned long size)
312 {
313         reserve_bootmem_core(contig_page_data.bdata, addr, size);
314 }
```

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

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

```
289 void __init reserve_bootmem_node (pg_data_t *pgdat,
                                     unsigned long physaddr,
                                     unsigned long size)
290 {
291         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         /*
 78          * round up, partially reserved pages are considered
 79          * fully reserved.
 80          */
 81         unsigned long sidx = (addr - bdata->node_boot_start)/PAGE_SIZE;
 82         unsigned long eidx = (addr + size - bdata->node_boot_start +
 83                                 PAGE_SIZE-1)/PAGE_SIZE;
 84         unsigned long end = (addr + size + PAGE_SIZE-1)/PAGE_SIZE;
 85 
 86         if (!size) BUG();
 87 
 88         if (sidx < 0)
 89                 BUG();
 90         if (eidx < 0)
 91                 BUG();
 92         if (sidx >= eidx)
 93                 BUG();
 94         if ((addr >> PAGE_SHIFT) >= bdata->node_low_pfn)
 95                 BUG();
 96         if (end > bdata->node_low_pfn)
 97                 BUG();
 98         for (i = sidx; i < eidx; i++)
 99                 if (test_and_set_bit(i, bdata->node_bootmem_map))
100                         printk("hm, page %08lx reserved twice.\n",
                                i*PAGE_SIZE);
101 }
```

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

82 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

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

86 Check that a non-zero value has been given

88-89 Check the starting index is not before the start of the node

90-91 Check the end index is not before the start of the node

92-93 Check the starting index is not after the end index

94-95 Check the starting address is not beyond the memory this bootmem node represents

96-97 Check the ending address is not beyond the memory this bootmem node represents

88-100 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*)

```
38 #define alloc_bootmem(x) \
39         __alloc_bootmem((x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))
40 #define alloc_bootmem_low(x) \
41         __alloc_bootmem((x), SMP_CACHE_BYTES, 0)
42 #define alloc_bootmem_pages(x) \
43         __alloc_bootmem((x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS))
44 #define alloc_bootmem_low_pages(x) \
45         __alloc_bootmem((x), PAGE_SIZE, 0)
```

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

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

42 `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

44 `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

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 {
346         void *ptr;
347 
348         ptr = __alloc_bootmem_core(pgdat->bdata, size, align, goal);
349         if (ptr)
350               return (ptr);
351 
352         /*
353          * Whoops, we cannot satisfy the allocation request.
354          */
355         printk(KERN_ALERT "bootmem alloc of %lu bytes failed!\n", size);
356         panic("Out of memory");
357         return NULL;
358 }
```

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.

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

```
144 static void * __init __alloc_bootmem_core (bootmem_data_t *bdata,
145        unsigned long size, unsigned long align, unsigned long goal)
146 {
147        unsigned long i, start = 0;
148        void *ret;
149        unsigned long offset, remaining_size;
150        unsigned long areasize, preferred, incr;
151        unsigned long eidx = bdata->node_low_pfn -
152                          (bdata->node_boot_start >> PAGE_SHIFT);
153
154        if (!size) BUG();
155
156        if (align & (align-1))
157              BUG();
158
159        offset = 0;
160        if (align &&
161            (bdata->node_boot_start & (align - 1UL)) != 0)
162              offset = (align - (bdata->node_boot_start &
                              (align - 1UL)));
163        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

162 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`

```
169         if (goal && (goal >= bdata->node_boot_start) &&
170                     ((goal >> PAGE_SHIFT) < bdata->node_low_pfn)) {
171               preferred = goal - bdata->node_boot_start;
172         } else
173               preferred = 0;
174
175         preferred = ((preferred + align - 1) & ~(align - 1))
                        >> PAGE_SHIFT;
176         preferred += offset;
177         areasize = (size+PAGE_SIZE-1)/PAGE_SIZE;
178         incr = align >> PAGE_SHIFT ? : 1;
```

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

169 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 adressable by this node then ....

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

173 Else the preferred offset is 0

175-176 Adjust the preferred address to take the offset into account so that the address will be correctly aligned

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

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

```
179
180 restart_scan:
181         for (i = preferred; i < eidx; i += incr) {
182               unsigned long j;
183               if (test_bit(i, bdata->node_bootmem_map))
184                     continue;
185               for (j = i + 1; j < i + areasize; ++j) {
186                     if (j >= eidx)
187                           goto fail_block;
188                     if (test_bit (j, bdata->node_bootmem_map))
189                           goto fail_block;
190               }
191               start = i;
192               goto found;
```

```
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 satisifed starting from `goal`, this label is jumped back to for rescanning

181-194 Starting from `preferred`, scan lineraly 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                    areasize = 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                    areasize = (remaining_size+PAGE_SIZE-1)/PAGE_SIZE;
```

```
224                 ret = phys_to_virt(bdata->last_pos*PAGE_SIZE +
225                                  offset +
                                     bdata->node_boot_start);
226                 bdata->last_pos = start+areasize-1;
227                 bdata->last_offset = remaining_size;
228             }
229             bdata->last_offset &= ~PAGE_MASK;
230         } else {
231             bdata->last_pos = start + areasize - 1;
232             bdata->last_offset = size & ~PAGE_MASK;
233             ret = phys_to_virt(start * PAGE_SIZE +
                             bdata->node_boot_start);
234         }
```

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 redundent

`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 adjactent 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

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;
111       unsigned long sidx;
112       unsigned long eidx = (addr + size -
                          bdata->node_boot_start)/PAGE_SIZE;
113       unsigned long end = (addr + size)/PAGE_SIZE;
114 
115       if (!size) BUG();
116       if (end > bdata->node_low_pfn)
117             BUG();
118 
119       /*
120        * Round up the beginning of the address.
121        */
122       start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
123       sidx = start - (bdata->node_boot_start/PAGE_SIZE);
124 
125       for (i = sidx; i < eidx; i++) {
126             if (!test_and_clear_bit(i, bdata->node_bootmem_map))
127                   BUG();
128       }
129 }
```

112 Calculate the end index affected as `eidx`

113 The end address is the end of the affected area rounded down to the nearest page if it is not already page aligned

115 If a size of 0 is freed, call `BUG`

116-117 If the end PFN is after the memory addressable by this node, call `BUG`

122 Round the starting address up to the nearest page if it is not already page aligned

123 Calculate the starting index to free

125-127 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

```
507 void __init mem_init(void)
508 {
509         int codesize, reservedpages, datasize, initsize;
510 
511         if (!mem_map)
512               BUG();
513 
514         set_max_mapnr_init();
515 
516         high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);
517 
518         /* clear the zero-page */
519         memset(empty_zero_page, 0, PAGE_SIZE);
```

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

```
520 
521         reservedpages = free_pages_init();
522 
```

512 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

```
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",
528                 (unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
529                 max_mapnr << (PAGE_SHIFT-10),
530                 codesize >> 10,
531                 reservedpages << (PAGE_SHIFT-10),
532                 datasize >> 10,
533                 initsize >> 10,
534                 (unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
535                );
```

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

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.

```
481 static int __init free_pages_init(void)
482 {
483         extern int ppro_with_ram_bug(void);
484         int bad_ppro, reservedpages, pfn;
485 
486         bad_ppro = ppro_with_ram_bug();
487 
488         /* this will put all low memory onto the freelists */
489         totalram_pages += free_all_bootmem();
490 
491         reservedpages = 0;
492         for (pfn = 0; pfn < max_low_pfn; pfn++) {
493                 /*
494                  * Only count reserved RAM pages
495                  */
496                 if (page_is_ram(pfn) && PageReserved(mem_map+pfn))
497                         reservedpages++;
498         }
499 #ifdef CONFIG_HIGHMEM
500         for (pfn = highend_pfn-1; pfn >= highstart_pfn; pfn--)
501                 one_highpage_init((struct page *) (mem_map + pfn), pfn,
bad_ppro);
502         totalram_pages += totalhigh_pages;
503 #endif
504         return reservedpages;
505 }
```

- 486 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
- 489 Call `free_all_bootmem()` to retire the boot memory allocator
- 491-498 Cycle through all of memory and count the number of reserved pages that were left over by the boot memory allocator
- 500-501 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*)

```
unsigned long __init free_all_bootmem_node (pg_data_t *pgdat)
```

```
300 {
301         return(free_all_bootmem_core(pgdat));
302 }

321 unsigned long __init free_all_bootmem (void)
322 {
323         return(free_all_bootmem_core(&contig_page_data));
324 }
```

299-302 For NUMA, simply call the core function with the specified `pgdat`

321-324 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;

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

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

252 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 `BUG()`

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

255 `idx` is the last index that is addressable by this node

256-263 Cycle through all pages addressable by this node

257 If the page is marked free then...

258 Increase the running count of pages given to the buddy allocator

259 Clear the `PG_reserved` flag

260 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

261 Call the buddy allocator free function

264 `total` will come the total number of pages given over by this function

```
270         page = virt_to_page(bdata->node_bootmem_map);
271         count = 0;
272         for (i = 0;
            i < ((bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT)
                                         )/8 + PAGE_SIZE-1)/PAGE_SIZE;
            i++,page++) {
273               count++;
274               ClearPageReserved(page);
275               set_page_count(page, 1);
276               __free_page(page);
277         }
278         total += count;
279         bdata->node_bootmem_map = NULL;
280
281         return total;
282 }
```

Free the allocator bitmap and return

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

271 Count of pages freed by the bitmap

272-277 For all pages used by the bitmap, free them to the buddy allocator the same way the previous block of code did

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

281 Return the total number of pages freed by this function

Chapter 2

Physical Page Management

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

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

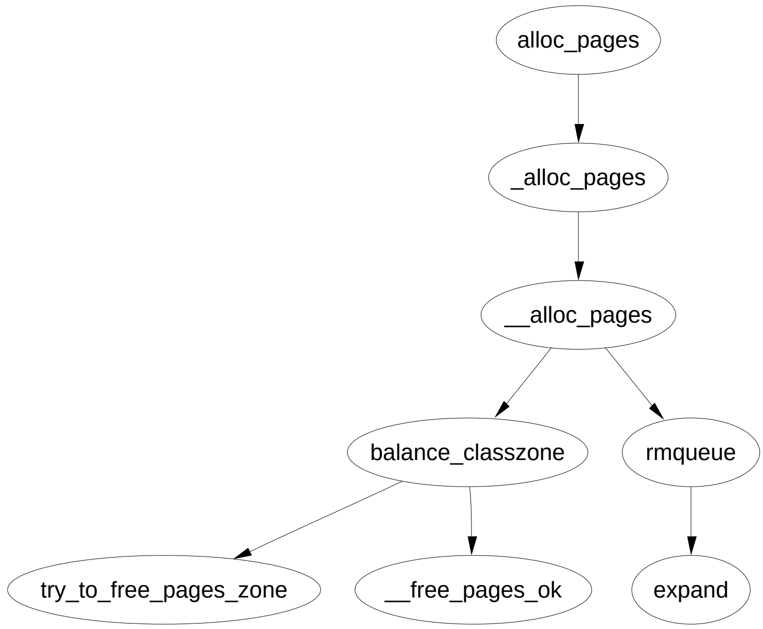

Figure 2.1: Call Graph: `alloc_pages()`

```
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`

```
244 #ifndef CONFIG_DISCONTIGMEM
245 struct page *_alloc_pages(unsigned int gfp_mask, unsigned int order)
246 {
247         return __alloc_pages(gfp_mask, order,
248                 contig_page_data.node_zonelists+(gfp_mask & GFP_ZONEMASK));
249 }
250 #endif
```

244 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 allocations

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

247 `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;
        mb();
        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;
        }
```

```
392
393         /* Atomic allocations - we can't balance anything */
394         if (!(gfp_mask & __GFP_WAIT))
395               return NULL;
396
397         page = balance_classzone(classzone, gfp_mask, order, &freed);
398         if (page)
399               return page;
400
401         zone = zonelist->zones;
402         min = 1UL << order;
403         for (;;) {
404               zone_t *z = *(zone++);
405               if (!z)
406                     break;
407
408               min += z->pages_min;
409               if (z->free_pages > min) {
410                     page = rmqueue(z, order);
411                     if (page)
412                           return page;
413               }
414         }
415
416         /* Don't let big-order allocations loop */
417         if (order > 3)
418               return NULL;
419
420         /* Yield for kswapd, and try again */
421         yield();
422         goto rebalance;
423 }
```

334 Set zone to be the preferred zone to allocate from

335 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

336-337 An unnecessary sanity check. `build_zonelists()` would need to be seriously broken for this to happen

338-350 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.

340 z is the zone currently been examined. zone is moved to the next fallback zone

`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 synchronusly 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

417-418 Satisifing 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

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.

```
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);

                        set_page_count(page, 1);
                        if (BAD_RANGE(zone,page))
                                BUG();
                        if (PageLRU(page))
```

```
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

227 Show that the page is in use

228-233 Sanity checks

234 Page block has been successfully allocated so return it

236-237 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

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

241 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.

```
177 static inline struct page * expand (zone_t *zone,
                                struct page *page,
                                unsigned long index,
                                int low,
                                int high,
                                free_area_t * area)
179 {
180         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 }
```

177 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`

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

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

182-192 Keep splitting until a block of the needed page order is found

183-184 Sanity check that checks to make sure the page this page belongs to this zone and is within the `zone_mem_map`

185 area is now the next `free_area_t` representing the lower order of page blocks

186 high is the next order of page blocks to be split

187 The size of the block been split is now half as big

188 Of the pair of buddies, the one lower in the `mem_map` is added to the free list for the lower order

189 Toggle the bit representing the pair of buddies

190 `index` now the index of the second buddy of the newly created pair

191 `page` now points to the second buddy of the newly created paid

193-194 Sanity check

195 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

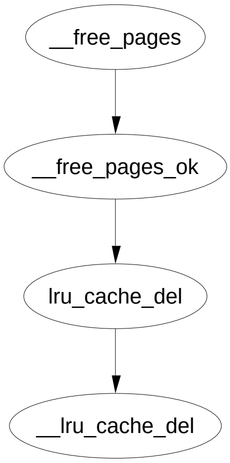

Figure 2.2: Call Graph: `__free_pages()`

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

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.

```
451 void __free_pages(struct page *page, unsigned int order)
452 {
453         if (!PageReserved(page) && put_page_testzero(page))
454                 __free_pages_ok(page, order);
455 }
```

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

453 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

454 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.

```
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;
back_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.
```

```
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
152             list_del(&buddy1->list);
153             mask <<= 1;
154             area++;
155             index >>= 1;
156             page_idx &= mask;
157         }
158         list_add(&(base + page_idx)->list, &area->free_list);
159
160         spin_unlock_irqrestore(&zone->lock, flags);
161         return;
162
163  local_freelist:
164         if (current->nr_local_pages)
165             goto back_local_freelist;
166         if (in_interrupt())
167             goto back_local_freelist;
168
169         list_add(&page->list, &current->local_pages);
170         page->index = order;
171         current->nr_local_pages++;
172 }
```

82 The parameters are the beginning of the page block to free and what order number of pages are to be freed.

32 A dirty page on the LRU will still have the LRU bit set when pinned for IO. It is just freed directly when the IO is complete so it just has to be removed from the LRU list

99-108 Sanity checks

109 The flags showing a page has being referenced and is dirty have to be cleared because the page is now free and not in use

`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 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}

154 area is a pointer within an array so area++ moves to the next index

155 The index in the bitmap of the higher order

156 The page index within the `zone_mem_map` for the buddy to merge

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

160-161 Changes to the zone is complete so free the lock and return

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

164-165 If the process already has reserved pages, it is not allowed to reserve any more so return back

166-167 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

169 Add the page block to the list for the processes `local_pages`

170 Record what order allocation it was for freeing later

171 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

```
#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

```
#define __get_free_page(gfp_mask) \
             __get_free_pages((gfp_mask),0)
```

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

```
428 unsigned long __get_free_pages(unsigned int gfp_mask,
                                unsigned int order)
428 {
430         struct page * page;
431
432         page = alloc_pages(gfp_mask, order);
433         if (!page)
434               return 0;
435         return (unsigned long) page_address(page);
436 }
```

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

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

433-434 Make sure the page is valid

435 `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

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

447 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

```
438 unsigned long get_zeroed_page(unsigned int gfp_mask)
439 {
440         struct page * page;
441
442         page = alloc_pages(gfp_mask, 0);
443         if (page) {
444               void *address = page_address(page);
445               clear_page(address);
446               return (unsigned long) address;
447         }
448         return 0;
449 }
```

438 `gfp_mask` are the flags which affect allocator behaviour.

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

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

445 `clear_page()` will fill the contents of a page with zero

446 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

```
457 void free_pages(unsigned long addr, unsigned int order)
458 {
459         if (addr != 0)
460                 __free_pages(virt_to_page(addr), order);
461 }
```

460 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

```
460 #define __free_page(page) __free_pages((page), 0)
```

Chapter 3

Non-Contiguous Memory Allocation

3.1 Allocating A Non-Contiguous Area

`vmalloc(unsigned long size)`	Allocate a number of pages in vmalloc space that satisfy the requested size
`vmalloc_dma(unsigned long size)`	Allocate a number of pages from `ZONE_DMA`
`vmalloc_32(unsigned long size)`	Allocate memory that is suitable for 32 bit addressing. This ensures it is in `ZONE_NORMAL` at least which some PCI devices require

Table 3.1: Non-Contiguous Memory Allocation API

Function: vmalloc (*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 `__vmalloc()`

```
static inline void * vmalloc (unsigned long size)
{
        return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL);
}


static inline void * vmalloc_dma (unsigned long size)
{
        return __vmalloc(size, GFP_KERNEL|GFP_DMA, PAGE_KERNEL);
}

```

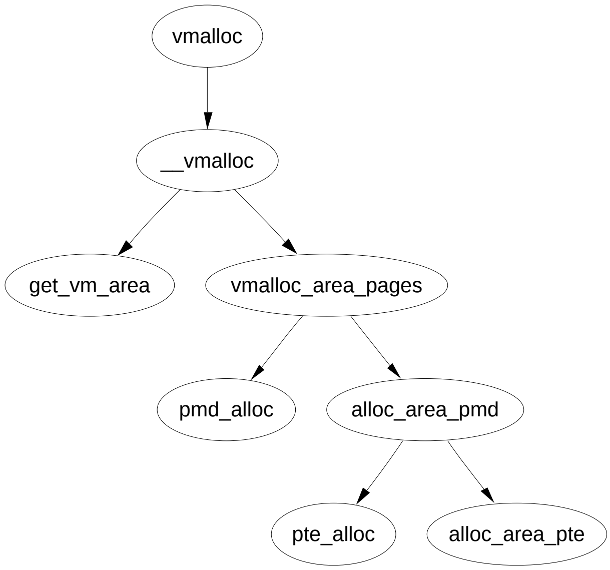

Figure 3.1: Call Graph: `vmalloc()`

```
50
51 static inline void * vmalloc_32(unsigned long size)
52 {
53         return __vmalloc(size, GFP_KERNEL, PAGE_KERNEL);
54 }
```

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

42 The flag indicates to only allocate from `ZONE_DMA`

51 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.

```
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);
```

```
242         if (!area)
243                 return NULL;
245         addr = area->addr;
246         if (vmalloc_area_pages(VMALLOC_VMADDR(addr),
                                    size, gfp_mask, prot)) {
247                 vfree(addr);
248                 return NULL;
249         }
250         return addr;
251 }
```

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

236 Align the size to a page size

237 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

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

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

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

247-248 If the allocation fails, free any PTEs, pages and descriptions of the area

250 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 lineraly 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);
```

```
184         for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
185                 if ((size + addr) < addr)
186                         goto out;
187                 if (size + addr <= (unsigned long) tmp->addr)
188                         break;
189                 addr = tmp->size + (unsigned long) tmp->addr;
190                 if (addr > VMALLOC_END-size)
191                         goto out;
192         }
193         area->flags = flags;
194         area->addr = (void *)addr;
195         area->size = size;
196         area->next = *p;
197         *p = area;
198         write_unlock(&vmlist_lock);
199         return area;
200
201 out:
202         write_unlock(&vmlist_lock);
203         kfree(area);
204         return NULL;
205 }
```

`171` 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`

`176-178` Allocate space for the `vm_struct` description struct

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

`180-181` This is to ensure the size is not 0 after the padding

`182` Start the search at the beginning of the vmalloc address space

`183` Lock the list

`184-192` Walk through the list searching for an area large enough for the request

`185-186` Check to make sure the end of the addressable range has not been reached

`187-188` If the requested area would fit between the current address and the next area, the search is complete

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

`193-195` Copy in the area information

`196-197` Link the new area into the list

`198-199` Unlock the list and return

201 This label is reached if the request could not be satisfied

202 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.

```
140 inline int vmalloc_area_pages (unsigned long address, unsigned long size,
141                                int gfp_mask, pgprot_t prot)
142 {
143         pgd_t * dir;
144         unsigned long end = address + size;
145         int ret;
146 
147         dir = pgd_offset_k(address);
148         spin_lock(&init_mm.page_table_lock);
149         do {
150                 pmd_t *pmd;
151 
152                 pmd = pmd_alloc(&init_mm, dir, address);
153                 ret = -ENOMEM;
154                 if (!pmd)
155                         break;
156 
157                 ret = -ENOMEM;
158                 if (alloc_area_pmd(pmd, address, end - address,
                                    gfp_mask, prot))
159                         break;
160 
161                 address = (address + PGDIR_SIZE) & PGDIR_MASK;
162                 dir++;
163 
164                 ret = 0;
165         } while (address && (address < end));
166         spin_unlock(&init_mm.page_table_lock);
167         flush_cache_all();
168         return ret;
169 }
```

140 `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

144 The end address is the starting address plus the size

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

```
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;
}
```

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.

```
static inline int alloc_area_pte (pte_t * pte, unsigned long address,
                        unsigned long size, int gfp_mask, pgprot_t prot)
{
        unsigned long end;

        address &= ~PMD_MASK;
        end = address + size;
        if (end > PMD_SIZE)
                end = PMD_SIZE;
        do {
                struct page * page;
                spin_unlock(&init_mm.page_table_lock);
                page = alloc_page(gfp_mask);
                spin_lock(&init_mm.page_table_lock);
                if (!pte_none(*pte))
                        printk(KERN_ERR
                               "alloc_area_pte: page already exists\n");
                if (!page)
                        return -ENOMEM;
                set_pte(pte, mk_pte(page, prot));
                address += PAGE_SIZE;
                pte++;
        } while (address < end);
```

```
117         return 0;
118 }
```

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()`

```
void vfree(void * addr)
{
        struct vm_struct **p, *tmp;

        if (!addr)
                return;
```

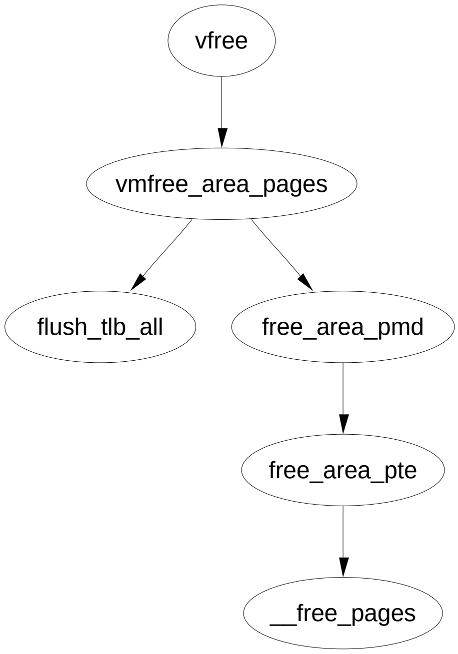

Figure 3.2: Call Graph: `vfree()`

```
213         if ((PAGE_SIZE-1) & (unsigned long) addr) {
214                 printk(KERN_ERR
                        "Trying to vfree() bad address (%p)\n", addr);
215                 return;
216         }
217         write_lock(&vmlist_lock);
218         for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
219                 if (tmp->addr == addr) {
220                         *p = tmp->next;
221                         vmfree_area_pages(VMALLOC_VMADDR(tmp->addr),
                                          tmp->size);
222                         write_unlock(&vmlist_lock);
223                         kfree(tmp);
224                         return;
225                 }
226         }
227         write_unlock(&vmlist_lock);
228         printk(KERN_ERR
                "Trying to vfree() nonexistent vm area (%p)\n", addr);
229 }
```

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

211-213 Ignore NULL addresses

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

87 Call `free_area_pmd()` to perform the second stage of the page table walk

89 `address` becomes the starting address of the next PGD

90 Move to the next PGD

92 Flush the TLB as the page tables have now changed

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.

```
56 static inline void free_area_pmd(pgd_t * dir, unsigned long address,
                                         unsigned long size)
57 {
58         pmd_t * pmd;
59         unsigned long end;
60
61         if (pgd_none(*dir))
62                 return;
63         if (pgd_bad(*dir)) {
64                 pgd_ERROR(*dir);
65                 pgd_clear(dir);
66                 return;
67         }
68         pmd = pmd_offset(dir, address);
69         address &= ~PGDIR_MASK;
70         end = address + size;
71         if (end > PGDIR_SIZE)
72                 end = PGDIR_SIZE;
73         do {
74                 free_area_pte(pmd, address, end - address);
75                 address = (address + PMD_SIZE) & PMD_MASK;
76                 pmd++;
77         } while (address < end);
78 }
```

56 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

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

```
static inline void free_area_pte(pmd_t * pmd, unsigned long address,
                                    unsigned long size)
{
        pte_t * pte;
        unsigned long end;

        if (pmd_none(*pmd))
                return;
        if (pmd_bad(*pmd)) {
                pmd_ERROR(*pmd);
                pmd_clear(pmd);
                return;
        }
        pte = pte_offset(pmd, address);
        address &= ~PMD_MASK;
        end = address + size;
        if (end > PMD_SIZE)
                end = PMD_SIZE;
        do {
                pte_t page;
                page = ptep_get_and_clear(pte);
                address += PAGE_SIZE;
                pte++;
                if (pte_none(page))
                        continue;
                if (pte_present(page)) {
                        struct page *ptpage = pte_page(page);
                        if (VALID_PAGE(ptpage) &&
                           (!PageReserved(ptpage)))
                                __free_page(ptpage);
                        continue;
                }
                printk(KERN_CRIT
                       "Whee.. Swapped out page in kernel page table\n");
        } while (address < end);
}
```

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

27-28 The PMD could be absent if this region is from a failed `vmalloc()`

29-33 A PMD can be bad if it's not in main memory, it's read only or it's marked dirty or accessed

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

35 Align the address to the PMD

36-38 The end is either the end of the requested region or the end of the PMD, whichever occurs first

38-53 Step through all PTEs, perform checks and free the PTE with its associated page

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

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

43 Move to the next PTE

44 If there was no PTE, simply continue

46-51 If the page is present, perform basic checks and then free it

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

48-49 Make sure the page is a valid page and it is not reserved before calling `__free_page()` to free the physical page

50 Continue to the next PTE

52 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

```
kmem_cache_t *
kmem_cache_create (const char *name, size_t size,
        size_t offset, unsigned long flags,
        void (*ctor)(void*, kmem_cache_t *, unsigned long),
        void (*dtor)(void*, kmem_cache_t *, unsigned long))
{
        const char *func_nm = KERN_ERR "kmem_create: ";
        size_t left_over, align, slab_size;
```

`kmem_cache_create(const char *name, size_t size,`
`     size_t offset, unsigned long flags,`
`     void (*ctor)(void*, kmem_cache_t *, unsigned long),`
`     void (*dtor)(void*, kmem_cache_t *, unsigned long))`
Creates a new cache and adds it to the cache chain

`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

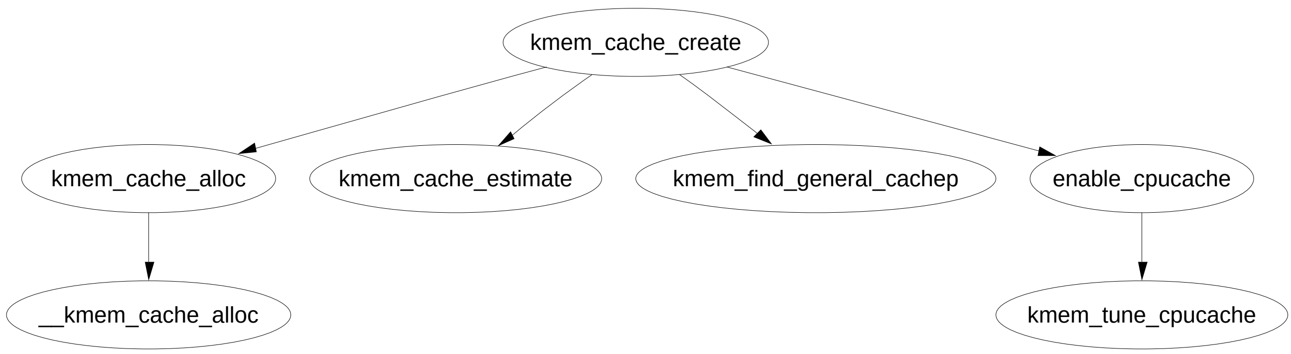

Figure 4.1: Call Graph: `kmem_cache_create()`

```
628         kmem_cache_t *cachep = NULL;
629
633         if ((!name) ||
634                 ((strlen(name) >= CACHE_NAMELEN - 1)) ||
635                 in_interrupt() ||
636                 (size < BYTES_PER_WORD) ||
637                 (size > (1<<MAX_OBJ_ORDER)*PAGE_SIZE) ||
638                 (dtor && !ctor) ||
639                 (offset < 0 || offset > size))
640                         BUG();
641
```

Perform basic sanity checks for bad usage

622 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

633-640 These are all serious usage bugs that prevent the cache even attempting to create

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

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

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

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

638 A destructor cannot be used if no constructor is available

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

640 Call `BUG()` to exit

```
642 #if DEBUG
643         if ((flags & SLAB_DEBUG_INITIAL) && !ctor) {
645                 printk("%sNo con, but init state check
                         requested - %s\n", func_nm, name);
646                 flags &= ~SLAB_DEBUG_INITIAL;
647         }
648
649         if ((flags & SLAB_POISON) && ctor) {
651                 printk("%sPoisoning requested, but con given - %s\n",
func_nm, name);
652                 flags &= ~SLAB_POISON;
653         }
654 #if FORCED_DEBUG
655         if ((size < (PAGE_SIZE>>3)) && !(flags & SLAB_MUST_HWCACHE_ALIGN))
660                 flags |= SLAB_RED_ZONE;
661         if (!ctor)
662                 flags |= SLAB_POISON;
663 #endif
664 #endif
670         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.

673 Allocate a cache descriptor object from the `cache_cache`(see Section 4.2.2)

674-675 If out of memory goto opps which handles the oom situation

676 Zero fill the object to prevent surprises with uninitialised data

```
682         if (size & (BYTES_PER_WORD-1)) {
683                 size += (BYTES_PER_WORD-1);
684                 size &= ~(BYTES_PER_WORD-1);
685                 printk("%sForcing size word alignment
                        - %s\n", func_nm, name);
686         }
687 
688 #if DEBUG
689         if (flags & SLAB_RED_ZONE) {
694                 flags &= ~SLAB_HWCACHE_ALIGN;
695                 size += 2*BYTES_PER_WORD;
696         }
697 #endif
698         align = BYTES_PER_WORD;
699         if (flags & SLAB_HWCACHE_ALIGN)
700                 align = L1_CACHE_BYTES;
701 
703         if (size >= (PAGE_SIZE>>3))
708                 flags |= CFLGS_OFF_SLAB;
709 
710         if (flags & SLAB_HWCACHE_ALIGN) {
714                 while (size < align/2)
715                         align /= 2;
716                 size = (size+align-1)&(~(align-1));
717         }
```

Align the object size to the word size

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

683 Increase the object by the size of a word

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

685 Print out an informational message for debugging purposes

688-697 If debugging is enabled then the alignments have to change slightly

694 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

695 The size of the object increases by two `BYTES_PER_WORD` to store the red zone mark at either end of the object

698 Align the object on a word size

699-700 If requested, align the objects to the L1 CPU cache

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

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

714-715 This is important to arches (e.g. Alpha or Pentium 4) with large L1 cache bytes. `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

716 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++;
```

```
753         } while (1);
754 
755         if (!cachep->num) {
756                 printk("kmem_cache_create: couldn't
                        create cache %s.\n", 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

```
761         slab_size = L1_CACHE_ALIGN(
                          cachep->num*sizeof(kmem_bufctl_t) +
                          sizeof(slab_t));
762
767         if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
768                 flags &= ~CFLGS_OFF_SLAB;
769                 left_over -= slab_size;
770         }
```

Align the slab size to the hardware cache

761 `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 left over 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

```
773         offset += (align-1);
774         offset &= ~(align-1);
775         if (!offset)
776                 offset = L1_CACHE_BYTES;
777         cachep->colour_off = offset;
778         cachep->colour = left_over/offset;
```

Calculate colour offsets.

773-774 `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

```
781         if (!cachep->gfporder && !(flags & CFLGS_OFF_SLAB))
782                 flags |= CFLGS_OPTIMIZE;
783
784         cachep->flags = flags;
785         cachep->gfpflags = 0;
786         if (flags & SLAB_CACHE_DMA)
787                 cachep->gfpflags |= GFP_DMA;
788         spin_lock_init(&cachep->spinlock);
789         cachep->objsize = size;
790         INIT_LIST_HEAD(&cachep->slabs_full);
```

```
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);
800
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_OPTIMIZE` 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
810                 list_for_each(p, &cache_chain) {
811                         kmem_cache_t *pc = list_entry(p,
```

```
                                       kmem_cache_t, next);
812
814                         if (!strcmp(pc->name, name))
815                                 BUG();
816                 }
817         }
818
822         list_add(&cachep->next, &cache_chain);
823         up(&cache_chain_sem);
824 opps:
825         return cachep;
826 }
```

Add the new cache to teh cache chain

806 Acquire the semaphore used to synchronize access to the cache chain

810-816 Check every cache on the cache chain and make sure there isn't a cache there with the same name. If there is, it means two caches of the same type are been created which is a serious bug

811 Get the cache from the list

814-815 Compare the names and if they match bug. It is worth noting that the new cache is not deleted, but this error is the result of sloppy programming during development and not a normal scenario

822 Link the cache into the chain.

823 Release the cache chain semaphore.

825 Return the new cache pointer

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.

```
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;

```

```
396         if (!(flags & CFLGS_OFF_SLAB)) {
397                 base = sizeof(slab_t);
398                 extra = sizeof(kmem_bufctl_t);
399         }
400         i = 0;
401         while (i*size + L1_CACHE_ALIGN(base+i*extra) <= wastage)
402                 i++;
403         if (i > 0)
404                 i--;
405 
406         if (i > SLAB_LIMIT)
407                 i = SLAB_LIMIT;
408 
409         *num = i;
410         wastage -= i*size;
411         wastage -= L1_CACHE_ALIGN(base+i*extra);
412         *left_over = wastage;
413 }
```

388 The parameters of the function are as follows

`gfporder` The $2^{gfporder}$ 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

392 `wastage` is decremented through the function. It starts with the maximum possible amount of wast-age.

393 `extra` is the number of bytes needed to store `kmem_bufctl_t`

394 `base` is where usable memory in the slab starts

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

400 `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*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 `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*extra` will calculate the amount of space needed to hold a kmem_bufctl_t for this object. As wast-age starts out as the size of the slab, its use is overloaded here.

403-404 Because the previous loop counts until the slab overflows, the number of objects that can be stored is `i-1`.

406-407 SLAB_LIMIT is the absolute largest number of objects a slab can store. Is is defined as 0xffffFFFE as this the largest number `kmem_bufctl_t()`, which is an unsigned int, can hold

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

410 Take away the space taken up by all the objects from wast-age

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

412 Wast-age has now been calculated as the left over space in the slab

4.0.3 Cache Shrinking

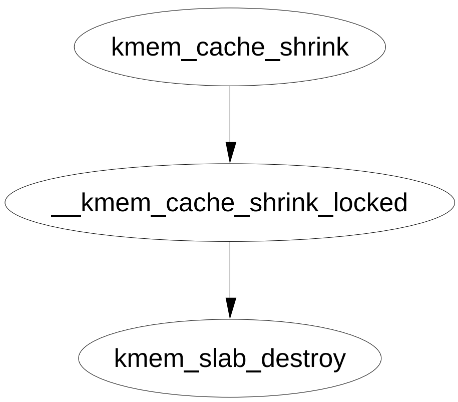

Figure 4.2: Call Graph: `kmem_cache_shrink()`

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.

```
int kmem_cache_shrink(kmem_cache_t *cachep)
{
        int ret;

```

```
970         if (!cachep || in_interrupt() || !is_chained_kmem_cache(cachep))
971                 BUG();
972 
973         spin_lock_irq(&cachep->spinlock);
974         ret = __kmem_cache_shrink_locked(cachep);
975         spin_unlock_irq(&cachep->spinlock);
976 
977         return ret << cachep->gfporder;
978 }
```

966 The parameter is the cache been shrunk

970 Check that

- 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

973 Acquire the cache descriptor lock and disable interrupts

974 Shrink the cache

975 Release the cache lock and enable interrupts

976 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.

```
945 static int __kmem_cache_shrink(kmem_cache_t *cachep)
946 {
947         int ret;
948 
949         drain_cpu_caches(cachep);
950 
951         spin_lock_irq(&cachep->spinlock);
952         __kmem_cache_shrink_locked(cachep);
953         ret = !list_empty(&cachep->slabs_full) ||
954                 !list_empty(&cachep->slabs_partial);
955         spin_unlock_irq(&cachep->spinlock);
956         return ret;
957 }
```

949 Remove all objects from the per-CPU objects cache

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

937 Re-enable interrupts. This function is called with interrupts disabled and this is to free the interrupt as quickly as possible.

938 Delete the slab (see Section 4.1.4)

939 Record the number of slabs freed

940 Acquire the cache descriptor lock and disable interrupts

4.0.4 Cache Destroying

When a module is unloaded, it is responsible for destroying any cache is 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.

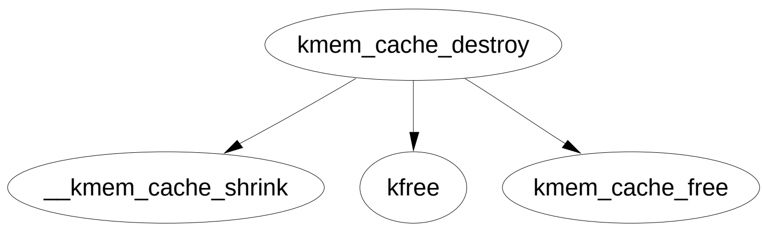

Figure 4.3: Call Graph: `kmem_cache_destroy()`

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

```
int kmem_cache_destroy (kmem_cache_t * cachep)
{
        if (!cachep || in_interrupt() || cachep->growing)
                BUG();

        /* Find the cache in the chain of caches. */
        down(&cache_chain_sem);
        /* the chain is never empty, cache_cache is never destroyed */
        if (clock_searchp == cachep)
                clock_searchp = list_entry(cachep->next.next,
                                                kmem_cache_t, next);
```

```
1006         list_del(&cachep->next);
1007         up(&cache_chain_sem);
1008
1009         if (__kmem_cache_shrink(cachep)) {
1010                 printk(KERN_ERR
                        "kmem_cache_destroy: Can't free all objects %p\n",
1011                        cachep);
1012                 down(&cache_chain_sem);
1013                 list_add(&cachep->next,&cache_chain);
1014                 up(&cache_chain_sem);
1015                 return 1;
1016         }
1017 #ifdef CONFIG_SMP
1018         {
1019                 int i;
1020                 for (i = 0; i < NR_CPUS; i++)
1021                         kfree(cachep->cpudata[i]);
1022         }
1023 #endif
1024         kmem_cache_free(&cache_cache, cachep);
1025
1026         return 0;
1027 }
```

`997-998` 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.

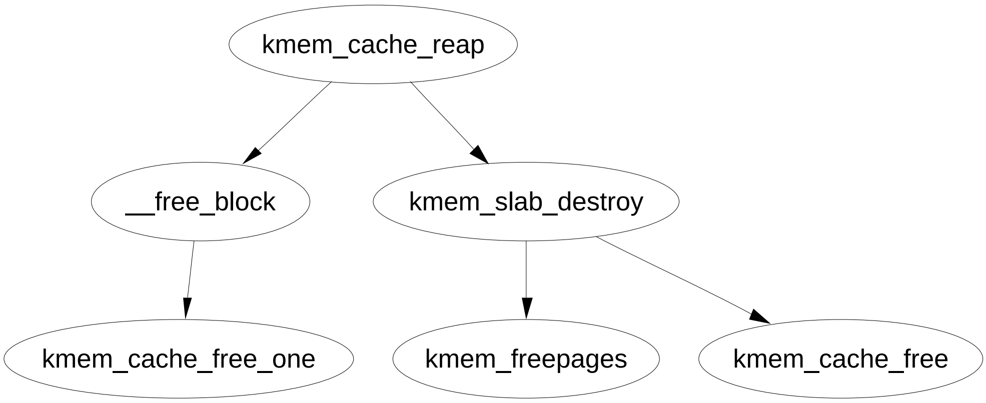

Figure 4.4: Call Graph: `kmem_cache_reap()`

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

```
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

```
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;
```

```
                if (searchp->dflags & DFLGS_GROWN) {
                        searchp->dflags &= ~DFLGS_GROWN;
                        goto next_unlock;
                }
#ifdef CONFIG_SMP
                {
                        cpucache_t *cc = cc_data(searchp);
                        if (cc && cc->avail) {
                                __free_block(searchp, cc_entry(cc),
                                             cc->avail);
                                cc->avail = 0;
                        }
                }
#endif

                full_free = 0;
                p = searchp->slabs_free.next;
                while (p != &searchp->slabs_free) {
                        slabp = list_entry(p, slab_t, list);
#if DEBUG
                        if (slabp->inuse)
                                BUG();
#endif
                        full_free++;
                        p = p->next;
                }

                pages = full_free * (1<<searchp->gfporder);
                if (searchp->ctor)
                        pages = (pages*4+1)/5;
                if (searchp->gfporder)
                        pages = (pages*4+1)/5;
                if (pages > best_pages) {
                        best_cachep = searchp;
                        best_len = full_free;
                        best_pages = pages;
                        if (pages >= REAP_PERFECT) {
                                clock_searchp =
                                     list_entry(searchp->next.next,
                                     kmem_cache_t,next);
                                goto perfect;
                        }
                }
next_unlock:
                spin_unlock_irq(&searchp->spinlock);
```

```
1816 next:
1817                 searchp =
                       list_entry(searchp->next.next,kmem_cache_t,next);
1818         } while (--scan && searchp != clock_searchp);
```

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

1765 Acquire an interrupt safe lock to the cache descriptor

1766-1767 If the cache is growing, skip it

1768-1771 If the cache has grown recently, skip it and clear the flag

1773-1779 Free any per CPU objects to the global pool

1784-1792 Count the number of slabs in the slabs_free list

1799 Calculate the number of pages all the slabs hold

1800-1801 If the objects have constructors, reduce the page count by one fifth to make it less likely to be selected for reaping

1802-1803 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

1804 If this is the best candidate found for reaping so far, check if it is perfect for reaping

1805-1807 Record the new maximums

1806 best_len is recorded so that it is easy to know how many slabs is half of the slabs in the free list

1808 If this cache is perfect for reaping then ....

1809 Update clock_searchp

1810 Goto perfect where half the slabs will be freed

1814 This label is reached if it was found the cache was growing after acquiring the lock

1815 Release the cache descriptor lock

1816 Move to the next entry in the cache chain

1818 Scan while REAP_SCANLEN has not been reached and we have not cycled around the whole cache chain

```
1820         clock_searchp = searchp;
1821
1822         if (!best_cachep)
1824                 goto out;
1825
```

```
1826         spin_lock_irq(&best_cachep->spinlock);
1827 perfect:
1828         /* free only 50% of the free slabs */
1829         best_len = (best_len + 1)/2;
1830         for (scan = 0; scan < best_len; scan++) {
1831                 struct list_head *p;
1832
1833                 if (best_cachep->growing)
1834                         break;
1835                 p = best_cachep->slabs_free.prev;
1836                 if (p == &best_cachep->slabs_free)
1837                         break;
1838                 slabp = list_entry(p,slab_t,list);
1839 #if DEBUG
1840                 if (slabp->inuse)
1841                         BUG();
1842 #endif
1843                 list_del(&slabp->list);
1844                 STATS_INC_REAPED(best_cachep);
1845
1846                 /* Safe to drop the lock. The slab is no longer
                      * linked to the
1847                  * cache.
1848                  */
1849                 spin_unlock_irq(&best_cachep->spinlock);
1850                 kmem_slab_destroy(best_cachep, slabp);
1851                 spin_lock_irq(&best_cachep->spinlock);
1852         }
1853         spin_unlock_irq(&best_cachep->spinlock);
1854         ret = scan * (1 << best_cachep->gfporder);
1855 out:
1856         up(&cache_chain_sem);
1857         return ret;
1858 }
```

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

1830-1852 Free `best_len` number of slabs

1833-1845 If the cache is growing, exit

1835 Get a slab from the list

1836-1837 If there is no slabs left in the list, exit

1838 Get the slab pointer

1840-1841 If debugging is enabled, make sure there isn't active objects in the slab

1843 Remove the slab from the `slabs_free` list

1844 Update statistics if enabled

1849 Free the cache descriptor and enable interrupts

1850 Destroy the slab. See Section 4.1.4

1851 Re-acquire the cache descriptor spinlock and disable interrupts

1853 Free the cache descriptor and enable interrupts

1854 `ret` is the number of pages that was freed

1856-1857 Free the cache semaphore and return the number of pages freed

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.

```
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 `bufctls` 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 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.

```
1030 static inline slab_t * kmem_cache_slabmgmt (
                                kmem_cache_t *cachep,
1031                            void *objp,
                                int colour_off,
                                int local_flags)
1032 {
1033         slab_t *slabp;
1034
1035         if (OFF_SLAB(cachep)) {
1037                 slabp = kmem_cache_alloc(cachep->slabp_cache,
                                                  local_flags);
1038                 if (!slabp)
1039                         return NULL;
1040         } else {
1045                 slabp = objp+colour_off;
1046                 colour_off += L1_CACHE_ALIGN(cachep->num *
1047                                 sizeof(kmem_bufctl_t) +
                                    sizeof(slab_t));
1048         }
1049         slabp->inuse = 0;
1050         slabp->colouroff = colour_off;
1051         slabp->s_mem = objp+colour_off;
1052
1053         return slabp;
1054 }
```

1030 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

1040-1048 Reserve space at the beginning of the slab

1045 The address of the slab will be the beginning of the slab (`objp`) plus the colour offset

1046 `colour_off` is calculated to be the offset where the first object will be placed. The address is L1 cache aligned. `cachep->num * sizeof(kmem_bufctl_t)` is the amount of space needed to hold the bufctls for each object in the slab and `sizeof(slab_t)` is the size of the slab descriptor. This effectively has reserved the space at the beginning of the slab

1049 The number of objects in use on the slab is 0

1050 The colouroff is updated for placement of the new object

1051 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 (*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 `slabp_cache`.

```
1618 kmem_cache_t * kmem_find_general_cachep (size_t size,
                                                 int gfpflags)
1619 {
1620         cache_sizes_t *csizep = cache_sizes;
1621 
1626         for ( ; csizep->cs_size; csizep++) {
1627                 if (size > csizep->cs_size)
1628                         continue;
1629                 break;
1630         }
1631         return (gfpflags & GFP_DMA) ? csizep->cs_dmacachep :
                                           csizep->cs_cachep;
1632 }
```

1618 `size` is the size of the slab descriptor. `gfpflags` is always 0 as DMA memory is not needed for a slab descriptor

1626-1630 Starting with the smallest size, keep increasing the size until a cache is found with buffers large enough to store the slab descriptor

1631 Return either a normal or DMA sized cache depending on the `gfpflags` passed in. In reality, only the `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 `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*)

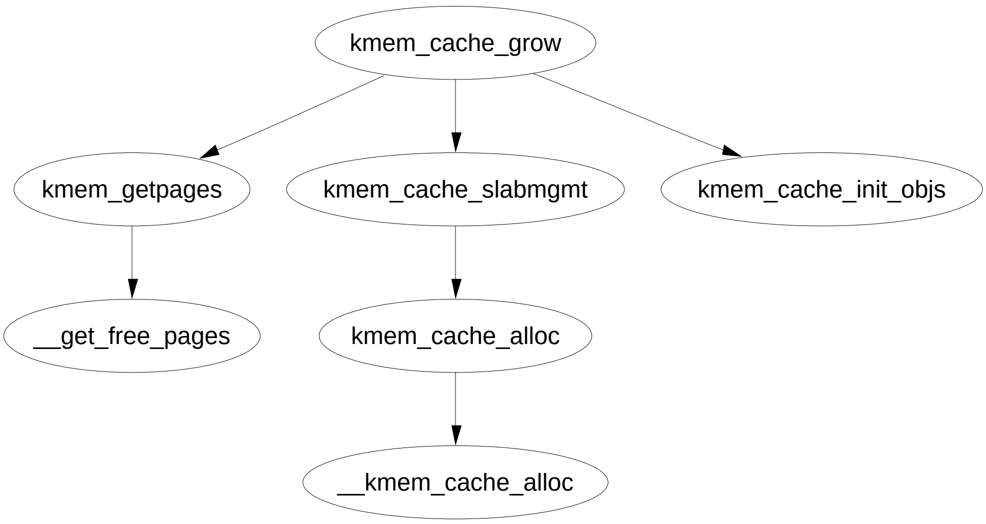

Figure 4.5: Call Graph: `kmem_cache_grow()`

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

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

```
1109        unsigned int     i, local_flags;
1110        unsigned long    ctor_flags;
1111        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

```
1116        if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW))
1117                BUG();
1118        if (flags & SLAB_NO_GROW)
1119                return 0;
1120
1127        if (in_interrupt() && (flags & SLAB_LEVEL_MASK) != SLAB_ATOMIC)
1128                BUG();
1129
1130        ctor_flags = SLAB_CTOR_CONSTRUCTOR;
1131        local_flags = (flags & SLAB_LEVEL_MASK);
1132        if (local_flags == SLAB_ATOMIC)
1137                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

```
1140        spin_lock_irqsave(&cachep->spinlock, save_flags);
1141
1143        offset = cachep->colour_next;
1144        cachep->colour_next++;
1145        if (cachep->colour_next >= cachep->colour)
1146                cachep->colour_next = 0;
1147        offset *= cachep->colour_off;
1148        cachep->dflags |= DFLGS_GROWN;
1149
1150        cachep->growing++;
1151        spin_unlock_irqrestore(&cachep->spinlock, save_flags);
```

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

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 }
1300 
```

Error handling

1192-1193 opps1 is reached if the pages for the slab were allocated. They must be freed

1195 Acquire the spinlock for accessing the cache descriptor

1196 Reduce the growing count

1197 Release the spinlock

1298 Return failure

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.

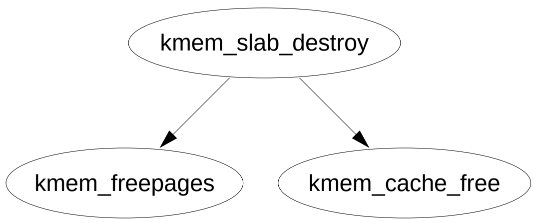

Figure 4.6: Call Graph: `kmem_slab_destroy()`

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.

```
555 static void kmem_slab_destroy (kmem_cache_t *cachep, slab_t *slabp)
556 {
557         if (cachep->dtor
561         ) {
562                 int i;
563                 for (i = 0; i < cachep->num; i++) {
564                         void* objp = slabp->s_mem+cachep->objsize*i;
```

565-574 DEBUG: Check red zone markers

```
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 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.

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

```
1059         int i;
1060
1061         for (i = 0; i < cachep->num; i++) {
1062                 void* objp = slabp->s_mem+cachep->objsize*i;

1063-1070             /* Debugging Part 1 */

1077                 if (cachep->ctor)
1078                         cachep->ctor(objp, cachep, ctor_flags);

1079-1092             /* Debugging Part 2 */

1093                 slab_bufctl(slabp)[i] = i+1;
1094         }
1095         slab_bufctl(slabp)[i-1] = BUFCTL_END;
1096         slabp->free = 0;
1097 }
```

1056 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

1061 Initialise `cache→num` number of objects

1062 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)`

1077-1078 If a constructor is available, call it

1093 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.

1096 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

```
1063 #if DEBUG
1064                 if (cachep->flags & SLAB_RED_ZONE) {
1065                         *((unsigned long*)(objp)) = RED_MAGIC1;
1066                         *((unsigned long*)(objp + cachep->objsize -
1067                                         BYTES_PER_WORD)) = RED_MAGIC1;
1068                         objp += BYTES_PER_WORD;
1069                 }
1070 #endif
```

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)
1084                         kmem_poison_obj(cachep, objp);
1085                 if (cachep->flags & SLAB_RED_ZONE) {
1086                         if (*((unsigned long*)(objp)) != RED_MAGIC1)
1087                                 BUG();
1088                         if (*((unsigned long*)(objp + cachep->objsize -
1089                                         BYTES_PER_WORD)) != RED_MAGIC1)
1090                                 BUG();
1091                 }
1092 #endif
```

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);
1531 }
```

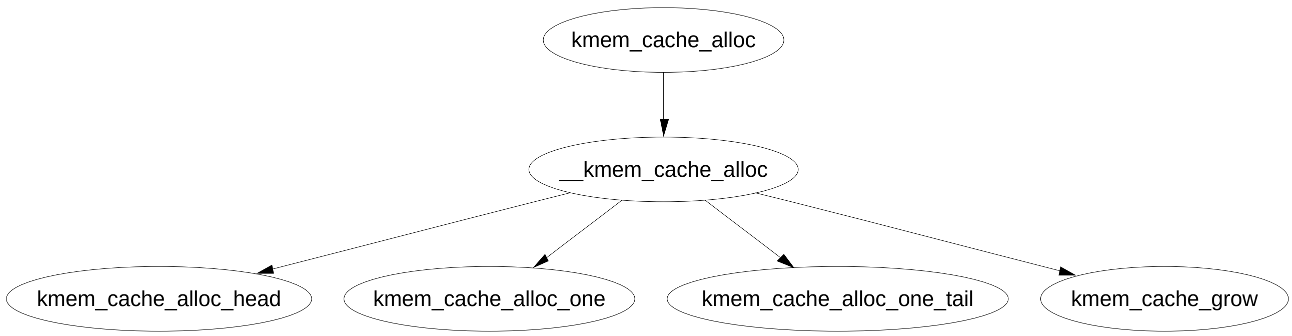

Figure 4.7: Call Graph: `kmem_cache_alloc()`

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.

```
1336 static inline void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)
1337 {
1338         unsigned long save_flags;
1339         void* objp;
1340
1341         kmem_cache_alloc_head(cachep, flags);
1342 try_again:
1343         local_irq_save(save_flags);

1365         objp = kmem_cache_alloc_one(cachep);

1367         local_irq_restore(save_flags);
1368         return objp;
1369 alloc_new_slab:

1374         local_irq_restore(save_flags);
1375         if (kmem_cache_grow(cachep, flags))
1379                 goto try_again;
1380         return NULL;
1381 }
```

1336 The parameters are the cache to allocate from and allocation specific flags.

1341 This function makes sure the appropriate combination of DMA flags are in use

1343 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

1365 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

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
1341         kmem_cache_alloc_head(cachep, flags);
1342 try_again:
1343         local_irq_save(save_flags);
1345         {
1346                 cpucache_t *cc = cc_data(cachep);
1347
1348                 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 =
                          kmem_cache_alloc_batch(cachep,cc,flags);
1355                                 if (!objp)
1356                                   goto alloc_new_slab_nolock;
1357                         }
1358                 } else {
1359                         spin_lock(&cachep->spinlock);
1360                         objp = kmem_cache_alloc_one(cachep);
1361                         spin_unlock(&cachep->spinlock);
1362                 }
1363         }
1364         local_irq_restore(save_flags);
1368         return objp;
1369 alloc_new_slab:
1371         spin_unlock(&cachep->spinlock);
1372 alloc_new_slab_nolock:
```

```
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)
1230 {
1231         if (flags & SLAB_DMA) {
1232                 if (!(cachep->gfpflags & GFP_DMA))
1233                         BUG();
```

```
1234         } else {
1235                 if (cachep->gfpflags & GFP_DMA)
1236                         BUG();
1237         }
1238 }
```

1229 The parameters are the cache we are allocating from and the flags requested for the allocation

1231 If the caller has requested memory for DMA use and ....

1232 The cache is not using DMA memory then `BUG()`

1235 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)

```
1281 #define kmem_cache_alloc_one(cachep)                    \
1282 ({                                                      \
1283         struct list_head * slabs_partial, * entry;      \
1284         slab_t *slabp;                                  \
1285                                                         \
1286         slabs_partial = &(cachep)->slabs_partial;       \
1287         entry = slabs_partial->next;                    \
1288         if (unlikely(entry == slabs_partial)) {         \
1289                 struct list_head * slabs_free;          \
1290                 slabs_free = &(cachep)->slabs_free;     \
1291                 entry = slabs_free->next;               \
1292                 if (unlikely(entry == slabs_free))      \
1293                         goto alloc_new_slab;            \
1294                 list_del(entry);                        \
1295                 list_add(entry, slabs_partial);         \
1296         }                                               \
1297                                                         \
1298         slabp = list_entry(entry, slab_t, list);        \
1299         kmem_cache_alloc_one_tail(cachep, slabp);       \
1300 })
```

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

1292-1293 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

1294-1295 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

1298 Obtain the slab from the list

1299 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.

```
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;
}
```

1230 The parameters are the cache and slab been allocated from

1245-1247 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

1250 `inuse` is the number of objects active on this slab

1251 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

1252 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.

1254-1257 If the slab is full, remove it from the `slabs_partial` list and place it on the `slabs_full`.

1258-1272 Debugging code

1273 Without debugging, the object is returned to the caller

1259-1261 If the object was poisoned with a known pattern, check it to guard against uninitialised access

1264-1265 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

1267-1269 Check the marker at the end of the object and change it to check for writes after the object later

1270 Update the object pointer to point to after the red marker

1273 Return the object

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. */
```

```
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

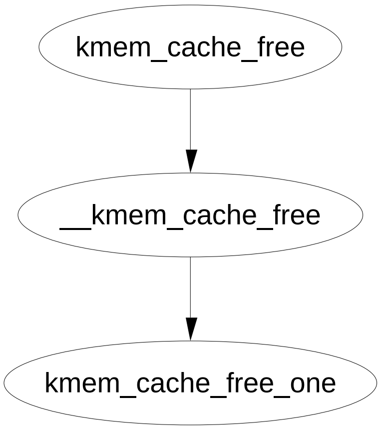

Figure 4.8: Call Graph: `kmem_cache_free()`

4.2.3 Object Freeing

Function: kmem_cache_free (*mm/slab.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 
1583         local_irq_save(flags);
1584         __kmem_cache_free(cachep, objp);
1585         local_irq_restore(flags);
1586 }
```

1574 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 `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 `__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: __kmem_cache_free (*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 __kmem_cache_free (kmem_cache_t *cachep, void* objp)
1492 {
1515         kmem_cache_free_one(cachep, objp);
1517 }
```

Function: __kmem_cache_free (*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 __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

1505 The pool has overflowed so batchcount number of objects is going to be freed to the global pool

1506 Update the number of available (`avail`) objects

1507-1508 Free a block of objects to the global cache

1509 Free the requested object and place it on the per CPU pool

1511 If the per CPU cache is not available, then free this object to the global pool

Function: kmem_cache_free_one (*mm/slab.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();
        }
        if (cachep->flags & SLAB_POISON)
                kmem_poison_obj(cachep, objp);
        if (kmem_extra_free_checks(cachep, slabp, objp))
                return;
#endif
        {
                unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;

                slab_bufctl(slabp)[objnr] = slabp->free;
                slabp->free = objnr;
        }
        STATS_DEC_ACTIVE(cachep);
```

```
1455
1457         {
1458                 int inuse = slabp->inuse;
1459                 if (unlikely(!--slabp->inuse)) {
1460                         /* Was partial or full, now empty. */
1461                         list_del(&slabp->list);
1462                         list_add(&slabp->list, &cachep->slabs_free);
1463                 } else if (unlikely(inuse == cachep->num)) {
1464                         /* Was full. */
1465                         list_del(&slabp->list);
1466                         list_add(&slabp->list, &cachep->slabs_partial);
1467                 }
1468         }
1469 }
```

`1416` Make sure the page is a slab page

`1423` Get the slab descriptor for the page

`1425-1447` Debugging material. Discussed at end of section

`1449` Calculate the index for the object been freed

`1452` As this object is now free, update the bufctl to reflect that. See the companion document for seeing how to track free objects.

`1454` If statistics are enabled, disable the number of active objects in the slab

`1459-1462` If `inuse` reaches 0, the slab is free and is moved to the `slabs_free` list

`1463-1466` If the number in use equals the number of objects in a slab, it is full so move it to the `slabs_full` list

`1469` Return

`1426-1431` If `SLAB_DEBUG_INITIAL` is set, the constructor is called to verify the object is in an initialised state

`1433-1442` 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

`1443-1444` Poison the freed object with a known pattern

`1445-1446` 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

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

```
1479 static void free_block (kmem_cache_t* cachep, void** objpp, int len)
1480 {
1481         spin_lock(&cachep->spinlock);
1482         __free_block(cachep, objpp, len);
1483         spin_unlock(&cachep->spinlock);
1484 }
```

1479 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

1483 Acquire a lock to the cache descriptor

1484 Discussed in next section

1485 Release the lock

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

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

```
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.

```
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;
```

```
446         do {
452                 snprintf(name, sizeof(name), "size-%Zd",
                              sizes->cs_size);
453                 if (!(sizes->cs_cachep =
454                         kmem_cache_create(name,
                                              sizes->cs_size,
455                                           0, SLAB_HWCACHE_ALIGN,
                                              NULL, NULL))) {
456                         BUG();
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-%Zd(DMA)",
                                                 sizes->cs_size);
465                 sizes->cs_dmacachep = kmem_cache_create(name,
                                sizes->cs_size, 0,
466                             SLAB_CACHE_DMA|SLAB_HWCACHE_ALIGN,
                                NULL, NULL);
467                 if (!sizes->cs_dmacachep)
468                         BUG();
469                 sizes++;
470         } while (sizes->cs_size);
471 }
```

438 Get a pointer to the cache_sizes array. See Section 4.3

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

444-445 `slab_break_gfp_order` determines how many pages a slab may use unless 0 objects fit into the slab. It is statically initialised to `BREAK_GFP_ORDER_LO` (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.

446-470 Create two caches for each size of memory allocation needed

452 Store the human readable cache name in `name`

453-454 Create the cache, aligned to the L1 cache. See Section 4.0.1

460-463 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.

464 The human readable name for the cache for DMA use

465-466 Create the cache, aligned to the L1 cache and suitable for DMA user. See Section 4.0.1

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

469 Move to the next element in the `cache_sizes` array

470 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)

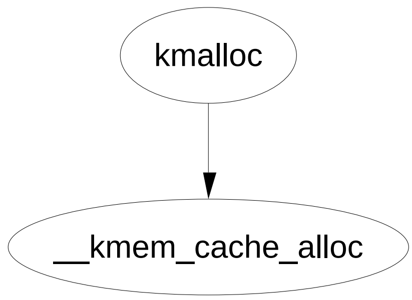

Figure 4.9: kmalloc

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

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

1555 `cache_sizes` is the array of caches for each size (See Section 4.3)

`1557-1562` Starting with the smallest cache, examine the size of each cache until one large enough to satisfy the request is found

`1560` If the allocation is for use with DMA, allocate an object from `cs_dmacachep` else use the `cs_cachep`

`1563` 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.

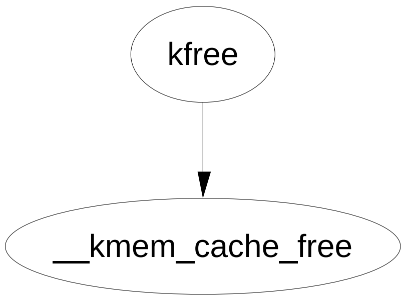

Figure 4.10: kfree

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

It is worth noting that the work this function does is almost identical to the function `kmem_cache_free()` with debugging enabled (See Section 4.2.3).

```
1595 void kfree (const void *objp)
1596 {
1597         kmem_cache_t *c;
1598         unsigned long flags;
1599
1600         if (!objp)
1601                 return;
1602         local_irq_save(flags);
1603         CHECK_PAGE(virt_to_page(objp));
1604         c = GET_PAGE_CACHE(virt_to_page(objp));
1605         __kmem_cache_free(c, (void*)objp);
1606         local_irq_restore(flags);
1607 }
```

`1600` 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

1602 Disable interrupts

1603 Make sure the page this object is in is a slab page

1604 Get the cache this pointer belongs to (See Section 4.1)

1605 Free the memory object

1606 Re-enable interrupts

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

```
        cpucache_t              *cpudata[NR_CPUS];
```

This structure is very simple

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

`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

```
#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 (`obj`) to the CPU cache (`cc`), the following block of code is used

```
cc_entry(cc)[cc->avail++] = obj;
```

To remove an object

```
obj = cc_entry(cc)[--cc->avail];
```

`cc_entry()` is a helper macro which gives a pointer to the first object in the cpucache. It is defined as

```
#define cc_entry(cpucache) \
        ((void **)(((cpucache_t*)(cpucache))+1))
```

This takes a pointer to a `cpucache`, increments the value by the size of the `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 kmalloc. The function `enable_cpucache()` is responsible for deciding what size to make the cache and calling `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 `g_cpucache_up()` is used to prevent cpucache's been enabled before it is possible. The function `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: 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.

```
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;
```

```
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
                    "enable_cpucache failed for %s, error %d.\n",
1709                                          cachep->name, -err);
1710 }
```

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

1699-1700 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

1701-1702 For smaller objects, just make sure the cache doesn't go above 3MiB in size

1706 Allocate the memory for the cpucache

1708-1709 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_ccupdate_local()` which swaps the new information with the old information in the cache descriptor.

```
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;
```

```
1666                 }
1667         }
1668         new.cachep = cachep;
1669         spin_lock_irq(&cachep->spinlock);
1670         cachep->batchcount = batchcount;
1671         spin_unlock_irq(&cachep->spinlock);
1672 
1673         smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
1674 
1675         for (i = 0; i < smp_num_cpus; i++) {
1676                 cpucache_t* ccold = new.new[cpu_logical_map(i)];
1677                 if (!ccold)
1678                         continue;
1679                 local_irq_disable();
1680                 free_block(cachep, cc_entry(ccold), ccold->avail);
1681                 local_irq_enable();
1682                 kfree(ccold);
1683         }
1684         return 0;
1685 oom:
1686         for (i--; i >= 0; i--)
1687                 kfree(new.new[cpu_logical_map(i)]);
1688         return -ENOMEM;
1689 }
```

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

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

```
typedef struct ccupdate_struct_s
{
        kmem_cache_t *cachep;
        cpucache_t *new[NR_CPUS];
} 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`.

```
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 
865         if (smp_call_function(func, arg, 1, 1))
866                 BUG();
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 
879         cc_data(new->cachep) = new->new[smp_processor_id()];
880         new->new[smp_processor_id()] = old;
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)

903 Re-enable interrupts

904 Show that no objects are available

906 The information for each CPU has been updated so call `do_ccupdate_local()` for each CPU to put the information back into the cache descriptor

907 Release the semaphore for the cache chain

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 `kmem_cache_t` from the `cache_cache` or `kmem_cache` cache. This is an obvious chicken and egg problem so the `cache_cache` has to be statically initialised as

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

358-360 Initialise the three lists as empty lists

361 The size of each object is the size of a cache descriptor

362 The creation and deleting of caches is extremely rare so do not consider it for reaping ever

363 Initialise the spinlock unlocked

364 Align the objects to the L1 cache

365 The human readable name

That statically defines all the fields that can be calculated at compile time. To initialise the rest of the struct, `kmem_cache_init()` is called from `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 `cache_cache` colour

```
416 void __init kmem_cache_init(void)
417 {
418         size_t left_over;
419
420         init_MUTEX(&cache_chain_sem);
421         INIT_LIST_HEAD(&cache_chain);
422
423         kmem_cache_estimate(0, cache_cache.objsize, 0,
424                         &left_over, &cache_cache.num);
425         if (!cache_cache.num)
426                 BUG();
427
428         cache_cache.colour = left_over/cache_cache.colour_off;
429         cache_cache.colour_next = 0;
430 }
```

420 Initialise the semaphore for access the cache chain

421 Initialise the cache chain linked list

423 This estimates the number of objects and amount of bytes wasted. See Section 4.0.2

425 If even one `kmem_cache_t` cannot be stored in a page, there is something seriously wrong

428 `colour` is the number of different cache lines that can be used while still keeping L1 cache alignment

429 `colour_next` indicates which line to use next. Start at 0

4.6 Interfacing with the Buddy Allocator

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

This allocates pages for the slab allocator

```
486 static inline void * kmem_getpages (kmem_cache_t *cachep, unsigned long
flags)
487 {
488         void    *addr;
495         flags |= cachep->gfpflags;
496         addr = (void*) __get_free_pages(flags, cachep->gfporder);
503         return addr;
504 }
```

495 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

496 Call the buddy allocator (See Section 2.3)

503 Return the pages or NULL if it failed

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

```
507 static inline void kmem_freepages (kmem_cache_t *cachep, void *addr)
508 {
509         unsigned long i = (1<<cachep->gfporder);
510         struct page *page = virt_to_page(addr);
511
517         while (i--) {
518                 PageClearSlab(page);
519                 page++;
520         }
521         free_pages((unsigned long)addr, cachep->gfporder);
522 }
```

509 Retrieve the order used for the original allocation

510 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`>

```
struct mm_struct {
        struct vm_area_struct * mmap;
        rb_root_t mm_rb;
        struct vm_area_struct * mmap_cache;
        pgd_t * pgd;
        atomic_t mm_users;
        atomic_t mm_count;
        int map_count;
        struct rw_semaphore mmap_sem;
        spinlock_t page_table_lock;

        struct list_head mmlist;

        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        unsigned long cpu_vm_mask;
        unsigned long swap_address;

        unsigned dumpable:1;

        /* Architecture-specific MM context */
        mm_context_t context;
};

```

`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 takes this semaphore with `down_read()`. If they need to write, it must be taken with `down_write()` and the `page_table_lock` must be taken as well

`page_table_lock` This protects a number of things. It protects the page tables, the rss count and the vma from modification

`mmlist` All mms are linked together via this field

`start_code, end_code` The start and end address of the code section

`start_data, end_data` The start and end address of the data section

`start_brk, brk` The start and end address of the heap

`start_stack` Predictably enough, the start of the stack region

`arg_start, arg_end` The start and end address of command line arguments

`env_start, env_end` The start and end address of environment variables

`rss` Resident Set Size, the number of resident pages for this process

`total_vm` The total memory space occupied by all vma regions in the process

`locked_vm` The amount of memory locked with mlock by the process

`def_flags` It has only one possible value, `VM_LOCKED`. It is used to determine if all future mappings are locked by default or not

`cpu_vm_mask` A bitmask representing all possible CPUs in an SMP system. The mask is used with IPI to determine if a processor should execute a particular function or not. This is important during TLB flush for each CPU for example

`swap_address` Used by the vmscan code to record the last address that was swapped from

`dumpable` Set by `prctl()`, this flag is important only to ptrace

`context` Architecture specific MMU context

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*)

```
226 #define allocate_mm()   (kmem_cache_alloc(mm_cachep, SLAB_KERNEL))
```

226 Allocate a `mm_struct` from the slab allocator

Function: mm_alloc (*kernel/fork.c*)

```
247 struct mm_struct * mm_alloc(void)
248 {
249         struct mm_struct * mm;
250 
251         mm = allocate_mm();
252         if (mm) {
253                 memset(mm, 0, sizeof(*mm));
254                 return mm_init(mm);
255         }
256         return NULL;
257 }
```

251 Allocate a `mm_struct` from the slab allocator

253 Zero out all contents of the struct

254 Perform basic initialisation

5.2.2 Initalising 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()`.

```
#define INIT_MM(name) \
{                                                    \
        mm_rb:           RB_ROOT,                    \
        pgd:             swapper_pg_dir,             \
        mm_users:        ATOMIC_INIT(2),             \
        mm_count:        ATOMIC_INIT(1),             \
        mmap_sem:        __RWSEM_INITIALIZER(name.mmap_sem),\
        page_table_lock: SPIN_LOCK_UNLOCKED,         \
        mmlist:          LIST_HEAD_INIT(name.mmlist),       \
}
```

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`.

```
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
        struct mm_struct * mm, *oldmm;
        int retval;

```

```
        tsk->min_flt = tsk->maj_flt = 0;
        tsk->cmin_flt = tsk->cmaj_flt = 0;
        tsk->nswap = tsk->cnswap = 0;

        tsk->mm = NULL;
        tsk->active_mm = NULL;

        /*
         * Are we cloning a kernel thread?
         *
         * We need to steal a active VM for that..
         */
        oldmm = current->mm;
        if (!oldmm)
                return 0;

        if (clone_flags & CLONE_VM) {
                atomic_inc(&oldmm->mm_users);
                mm = oldmm;
                goto good_mm;
        }

        retval = -ENOMEM;
        mm = allocate_mm();
        if (!mm)
                goto fail_nomem;

        /* Copy the current MM stuff.. */
        memcpy(mm, oldmm, sizeof(*mm));
        if (!mm_init(mm))
                goto fail_nomem;

        if (init_new_context(tsk,mm))
                goto free_pt;

        down_write(&oldmm->mmap_sem);
        retval = dup_mmap(mm);
        up_write(&oldmm->mmap_sem);

        if (retval)
                goto free_pt;

        /*
         * child gets a private LDT (if there was an LDT in the parent)
         */
```

```
364         copy_segments(tsk, mm);
365
366 good_mm:
367         tsk->mm = mm;
368         tsk->active_mm = mm;
369         return 0;
370
371 free_pt:
372         mmput(mm);
373 fail_nomem:
374         return retval;
375 }
```

314 The parameters are the flags passed for clone and the task that is creating a copy of the `mm_struct`

319-324 Initialise the `task_struct` fields related to memory management

331 Borrow the mm of the current running process to copy from

332 A kernel thread has no mm so it can return immediately

335-340 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

342 Allocate a new mm

347-349 Copy the parent mm and initialise the process specific mm fields with `init_mm()`

351-352 Initialise the MMU context for architectures that do not automatically manage their MMU

354-356 Call `dup_mmap()` which is responsible for copying all the VMAs regions in use by the parent process

358 `dup_mmap()` returns 0 on success. If it failed, the label `free_pt` will call `mmput()` which decrements the use count of the mm

364 This copies the LDT for the new process based on the parent process

367-369 Set the new mm, `active_mm` and return success

Function: mm_init (*kernel/fork.c*)

This function initialises process specific mm fields.

```
229 static struct mm_struct * mm_init(struct mm_struct * mm)
230 {
231         atomic_set(&mm->mm_users, 1);
232         atomic_set(&mm->mm_count, 1);
233         init_rwsem(&mm->mmap_sem);
234         mm->page_table_lock = SPIN_LOCK_UNLOCKED;
235         mm->pgd = pgd_alloc(mm);
236         mm->def_flags = 0;
237         if (mm->pgd)
238                 return mm;
239         free_mm(mm);
240         return NULL;
241 }
```

231 Set the number of users to 1

232 Set the reference count of the mm to 1

233 Initialise the semaphore protecting the VMA list

234 Initialise the spinlock protecting write access to it

235 Allocate a new PGD for the struct

236 By default, pages used by the process are not locked in memory

237 If a PGD exists, return the initialised struct

239 Initialisation failed, delete the `mm_struct` and return

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*)

```
void mmput(struct mm_struct *mm)
{
        if (atomic_dec_and_lock(&mm->mm_users, &mmlist_lock)) {
                extern struct mm_struct *swap_mm;
                if (swap_mm == mm)
```

```
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*)

```
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*)

```
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 }
```

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

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next;

        pgprot_t vm_page_prot;
        unsigned long vm_flags;

        rb_node_t vm_rb;

        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        /* Function pointers to deal with this struct. */
        struct vm_operations_struct * vm_ops;

        /* Information about our backing store: */
        unsigned long vm_pgoff;
        struct file * vm_file;
        unsigned long vm_raend;
        void * vm_private_data;
};
```

`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 `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(logN) 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>`

```
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 by sync-ed

`locked_pages` The number of pages locked in memory `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 manger 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>

```
struct address_space_operations {
        int (*writepage)(struct page *);
        int (*readpage)(struct file *, struct page *);
        int (*sync_page)(struct page *);
        /*
         * ext3 requires that a successful prepare_write()
         * call be followed
         * by a commit_write() call - they must be balanced
         */
        int (*prepare_write)(struct file *, struct page *,
                        unsigned, unsigned);
        int (*commit_write)(struct file *, struct page *,
                        unsigned, unsigned);
        /* Unfortunately this kludge is needed for FIBMAP.
         * Don't use it */
        int (*bmap)(struct address_space *, long);
        int (*flushpage) (struct page *, unsigned long);
        int (*releasepage) (struct page *, int);
#define KERNEL_HAS_O_DIRECT
        int (*direct_IO)(int, struct inode *, struct kiobuf *,
                       unsigned long, int);
};
```

`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()`

`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_pgoff()` 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_pgoff()`.

`do_mmap_pgoff()` 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 `pgoff` and finally that the process does not have too many mapped regions already.

Function: do_mmap_pgoff (*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

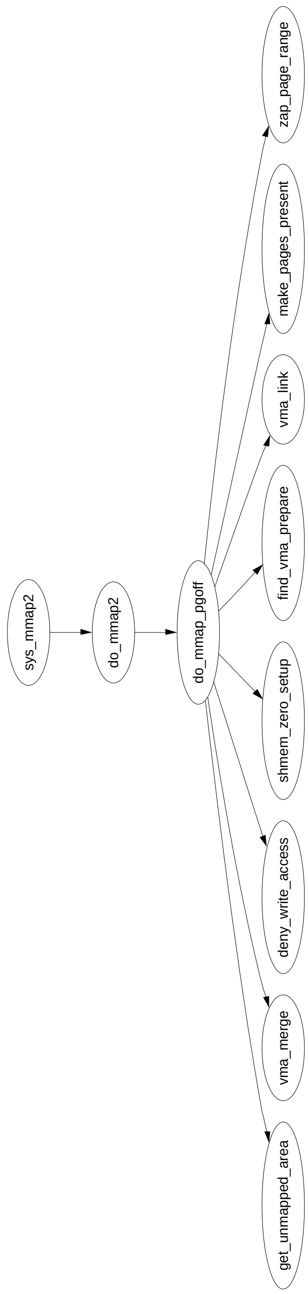

Figure 5.1: Call Graph: `sys_mmap2()`

- Allocate a `vm_area_struct` from the slab allocator and fill in its entries
- Link in the new VMA
- Call the filesystem or device specific `mmap()` function
- Update statistics and exit

```
393 unsigned long do_mmap_pgoff(struct file * file, unsigned long addr,
                         unsigned long len, unsigned long prot,
394                        unsigned long flags, unsigned long pgoff)
395 {
396       struct mm_struct * mm = current->mm;
397       struct vm_area_struct * vma, * prev;
398       unsigned int vm_flags;
399       int correct_wcount = 0;
400       int error;
401       rb_node_t ** rb_link, * rb_parent;
402
403       if (file && (!file->f_op || !file->f_op->mmap))
404             return -ENODEV;
405
406       if ((len = PAGE_ALIGN(len)) == 0)
407             return addr;
408
409       if (len > TASK_SIZE)
410             return -EINVAL;
411
412       /* offset overflow? */
413       if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
414             return -EINVAL;
415
416       /* Too many mappings? */
417       if (mm->map_count > max_map_count)
418             return -ENOMEM;
419
```

393 The parameters which correspond directly to the parameters to the mmap system call are

`file` the struct file to mmap if this is a file backed mapping

`addr` the requested address to map

`len` the length in bytes to mmap

`prot` is the permissions on the area

`flags` are the flags for the mapping

`pgoff` is the offset within the file to begin the mmap at

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
```

423 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

```
427         /* Do simple checking here so the lower-level routines won't have
428          * to. we assume access permissions have been handled by the open
429          * of the memory object, so we don't do any here.
430          */
431         vm_flags = calc_vm_flags(prot,flags) | mm->def_flags
                          | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
432
433         /* mlock MCL_FUTURE? */
434         if (vm_flags & VM_LOCKED) {
435               unsigned long locked = mm->locked_vm << PAGE_SHIFT;
436               locked += len;
437               if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
438                     return -EAGAIN;
439         }
440
```

431 `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

```
441         if (file) {
442                 switch (flags & MAP_TYPE) {
443                 case MAP_SHARED:
444                         if ((prot & PROT_WRITE) &&
                              !(file->f_mode & FMODE_WRITE))
445                                 return -EACCES;
446 
447                         /* Make sure we don't allow writing to
                               an append-only file.. */
448                         if (IS_APPEND(file->f_dentry->d_inode) &&
                              (file->f_mode & FMODE_WRITE))
449                                 return -EACCES;
450 
451                         /* make sure there are no mandatory
                               locks on the file. */
452                         if (locks_verify_locked(file->f_dentry->d_inode))
453                                 return -EAGAIN;
454 
455                         vm_flags |= VM_SHARED | VM_MAYSHARE;
456                         if (!(file->f_mode & FMODE_WRITE))
457                                 vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
458 
459                         /* fall through */
460                 case MAP_PRIVATE:
461                         if (!(file->f_mode & FMODE_READ))
462                                 return -EACCES;
463                         break;
464 
465                 default:
466                         return -EINVAL;
467                 }
468         } else {
469                 vm_flags |= VM_SHARED | VM_MAYSHARE;
470                 switch (flags & MAP_TYPE) {
471                 default:
472                         return -EINVAL;
473                 case MAP_PRIVATE:
474                         vm_flags &= ~(VM_SHARED | VM_MAYSHARE);
475                         /* fall through */
476                 case MAP_SHARED:
477                         break;
478                 }
479         }
```

441-468 If a file is been memory mapped, check the files access permissions

`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             goto munmap_back;
488       }
489
490       /* Check against address space limit. */
491       if ((mm->total_vm << PAGE_SHIFT) + len
492           > 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) && rb_parent)
503             if (vma_merge(mm, prev, rb_parent, addr, addr + len,
vm_flags))
504                   goto out;
505
```

`483` This function steps through the RB tree for he vma corresponding to a given address

`484-486` If a VMA was found and it is part of the new mmaping, remove the old mapping as the new one will cover both

`491-493` Make sure the new mapping will not 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

```
506         /* Determine the object being mapped and call the appropriate
507          * specific mapper. the address has already been validated, but
508          * not unmapped, but the maps are removed from the list.
509          */
510         vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
511         if (!vma)
512                 return -ENOMEM;
513 
514         vma->vm_mm = mm;
515         vma->vm_start = addr;
516         vma->vm_end = addr + len;
517         vma->vm_flags = vm_flags;
518         vma->vm_page_prot = protection_map[vm_flags & 0x0f];
519         vma->vm_ops = NULL;
520         vma->vm_pgoff = pgoff;
521         vma->vm_file = NULL;
522         vma->vm_private_data = NULL;
523         vma->vm_raend = 0;
524 
525         if (file) {
526                 error = -EINVAL;
527                 if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
528                         goto free_vma;
529                 if (vm_flags & VM_DENYWRITE) {
530                         error = deny_write_access(file);
531                         if (error)
532                                 goto free_vma;
533                         correct_wcount = 1;
534                 }
535                 vma->vm_file = file;
536                 get_file(file);
537                 error = file->f_op->mmap(file, vma);
538                 if (error)
539                         goto unmap_and_free_vma;
540         } else if (flags & MAP_SHARED) {
541                 error = shmem_zero_setup(vma);
542                 if (error)
543                         goto free_vma;
544         }
545 
```

510 Allocate a `vm_area_struct` from the slab allocator

514-523 Fill in the basic `vm_area_struct` fields

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

```
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
                       breaking
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
                      find_vma_prepare
561                  * to update the tree pointers.
562                  */
563                 addr = vma->vm_start;
564                 stale_vma = find_vma_prepare(mm, addr, &prev,
565                                         &rb_link, &rb_parent);
566                 /*
567                  * Make sure the lowlevel driver did its job right.
568                  */
569                 if (unlikely(stale_vma && stale_vma->vm_start <
                             vma->vm_end)) {
570                         printk(KERN_ERR "buggy mmap operation: [<%p>]\n",
571                                 file ? file->f_op->mmap : NULL);
572                         BUG();
573                 }
574         }
575 
576         vma_link(mm, vma, prev, rb_link, rb_parent);
577         if (correct_wcount)
578                 atomic_inc(&file->f_dentry->d_inode->i_writecount);
579 
```

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

```
580 out:
581         mm->total_vm += len >> PAGE_SHIFT;
582         if (vm_flags & VM_LOCKED) {
583                 mm->locked_vm += len >> PAGE_SHIFT;
584                 make_pages_present(addr, addr + len);
585         }
586         return addr;
587 
588 unmap_and_free_vma:
589         if (correct_wcount)
590                 atomic_inc(&file->f_dentry->d_inode->i_writecount);
591         vma->vm_file = NULL;
592         fput(file);
593 
594         /* Undo any partial mapping done by a device driver. */
595         zap_page_range(mm, vma->vm_start, vma->vm_end - vma->vm_start);
596 free_vma:
597         kmem_cache_free(vm_area_cachep, vma);
598         return error;
599 }
```

581-586 Update statistics for the process `mm_struct` and return the new address

588-595 This is reached if the file has been partially mapped before failing. The write statistics are updated and then all user pages are removed with `zap_page_range()`

596-598 This goto is used if the mapping failed immediately after the `vm_area_struct` is created. It is freed back to the slab allocator before the error is returned

5.3.2 Finding a Mapped Memory Region

Function: find_vma (*mm/mmap.c*)

```
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
{
        struct vm_area_struct *vma = NULL;

        if (mm) {
                /* Check the cache first. */
                /* (Cache hit rate is typically around 35%.) */
                vma = mm->mmap_cache;
                if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
                        rb_node_t * rb_node;
```

```
669
670                   rb_node = mm->mm_rb.rb_node;
671                   vma = NULL;
672
673                   while (rb_node) {
674                         struct vm_area_struct * vma_tmp;
675
676                         vma_tmp = rb_entry(rb_node,
                                  struct vm_area_struct, vm_rb);
677
678                         if (vma_tmp->vm_end > addr) {
679                               vma = vma_tmp;
680                               if (vma_tmp->vm_start <= addr)
681                                     break;
682                               rb_node = rb_node->rb_left;
683                         } else
684                               rb_node = rb_node->rb_right;
685                   }
686                   if (vma)
687                         mm->mmap_cache = vma;
688             }
689       }
690       return vma;
691 }
```

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

687 If the VMA is valid, set the `mmap_cache` for the next call to `find_vma()`

690 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*)

```
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
```

```
                                     vm_area_struct, vm_rb);
731                     if ((rb_prev ? (*pprev)->vm_next : mm->mmap) !=
vma)
732                             BUG();
733                     return vma;
734             }
735     }
736     *pprev = NULL;
737     return NULL;
738 }
```

694-721 This is essentially the same as the `find_vma()` function already described. The only difference is that the last right node accesses is remembered as this will represent the vma previous to the requested vma.

723-727 If the returned VMA has a left node, it means that it has to be traversed. It first takes the left leaf and then follows each right leaf until the bottom of the tree is found.

729-730 Extract the VMA from the red-black tree node

731-732 A debugging check, if this is the previous node, then its next field should point to the VMA being returned. If it is not, it is a bug

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

```
662 static inline struct vm_area_struct * find_vma_intersection(struct
mm_struct * mm, unsigned long start_addr, unsigned long end_addr)
663 {
664     struct vm_area_struct * vma = find_vma(mm,start_addr);
665
666     if (vma && end_addr <= vma->vm_start)
667             vma = NULL;
668     return vma;
669 }
```

664 Return the VMA closest to the starting address

666 If a VMA is returned and the end address is still less than the beginning of the returned VMA, the VMA does not intersect

668 Return the VMA if it does intersect

5.3.3 Finding a Free Memory Region

Function: get_unmapped_area (*mm/mmap.c*)

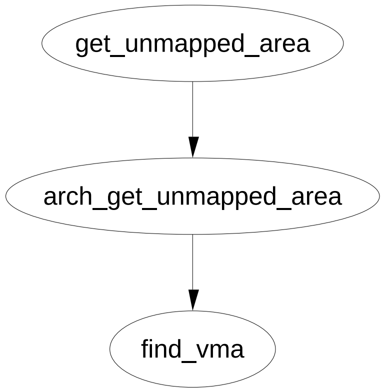

Figure 5.2: Call Graph: `get_unmapped_area()`

```
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,
                                                 len, pgoff, flags);
654
655       return arch_get_unmapped_area(file, addr, len, pgoff, flags);
656 }
```

642 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

644-650 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*)

```
612 #ifndef HAVE_ARCH_UNMAPPED_AREA
613 static inline unsigned long arch_get_unmapped_area(struct file *filp,
                  unsigned long addr, unsigned long len,
                  unsigned long pgoff, unsigned long flags)
614 {
615       struct vm_area_struct *vma;
616
617       if (len > TASK_SIZE)
618             return -ENOMEM;
619
620       if (addr) {
621             addr = PAGE_ALIGN(addr);
622             vma = find_vma(current->mm, addr);
623             if (TASK_SIZE - len >= addr &&
624                 (!vma || addr + len <= vma->vm_start))
625                   return addr;
626       }
627       addr = PAGE_ALIGN(TASK_UNMAPPED_BASE);
628
629       for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
630             /* At this point:  (!vma || addr < vma->vm_end). */
631             if (TASK_SIZE - len < addr)
632                   return -ENOMEM;
633             if (!vma || addr + len <= vma->vm_start)
634                   return addr;
635             addr = vma->vm_end;
636       }
637 }
638 #else
639 extern unsigned long arch_get_unmapped_area(struct file *, unsigned long,
unsigned long, unsigned long, unsigned long);
640 #endif
```

612 If this is not defined, it means that the architecture does not provide its own `arch_get_unmapped_area()` so this one is used instead

613 The parameters are the same as those for `get_unmapped_area()`

617-618 Sanity check, make sure the required map length is not too long

620-626 If an address is provided, use it for the mapping

621 Make sure the address is page aligned

622 `find_vma()` will return the region closest to the requested address

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

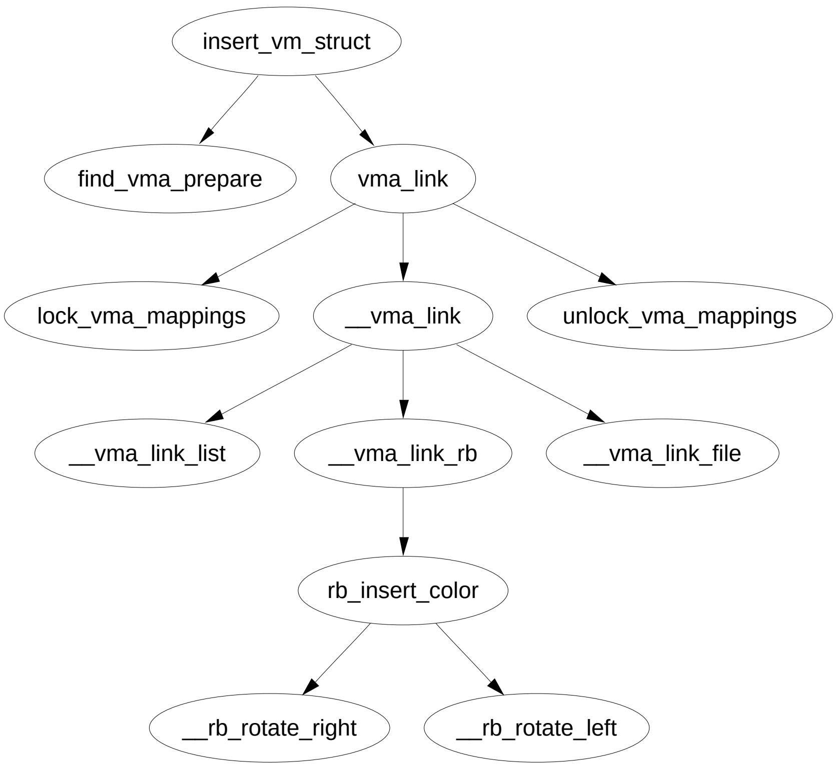

Figure 5.3: Call Graph: `insert_vm_struct()`

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,
                              struct vm_area_struct * vma)
1169 {
1170       struct vm_area_struct * __vma, * prev;
```

```
1171        rb_node_t ** rb_link, * rb_parent;
1172
1173        __vma = find_vma_prepare(mm, vma->vm_start, &prev,
                                     &rb_link, &rb_parent);
1174        if (__vma && __vma->vm_start < vma->vm_end)
1175              BUG();
1176        __vma_link(mm, vma, prev, rb_link, rb_parent);
1177        mm->map_count++;
1178        validate_mm(mm);
1179 }
```

1168 The arguments are the `mm_struct` mm that represents the linear address space and the `vm_area_struct` that is to be inserted

1173 `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

1174-1175 This is a check to make sure the returned vma is invalid. It is unclear how such a broken vma could exist

1176 This function does the actual work of linking the vma struct into the linear linked list and the red-black tree

1177 Increase the `map_count` to show a new mapping has been added

1178 `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 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 (*mm/mmap.c*)

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.

```
static struct vm_area_struct * find_vma_prepare(struct mm_struct * mm,
                                  unsigned long addr,
                                  struct vm_area_struct ** pprev,
                                  rb_node_t *** rb_link,
                                  rb_node_t ** rb_parent)
{
        struct vm_area_struct * vma;
        rb_node_t ** __rb_link, * __rb_parent, * rb_prev;

        __rb_link = &mm->mm_rb.rb_node;
```

```
254         rb_prev = __rb_parent = NULL;
255         vma = NULL;
256 
257         while (*__rb_link) {
258                 struct vm_area_struct *vma_tmp;
259 
260                 __rb_parent = *__rb_link;
261                 vma_tmp = rb_entry(__rb_parent,
                                struct vm_area_struct, vm_rb);
262 
263                 if (vma_tmp->vm_end > addr) {
264                         vma = vma_tmp;
265                         if (vma_tmp->vm_start <= addr)
266                                 return vma;
267                         __rb_link = &__rb_parent->rb_left;
268                 } else {
269                         rb_prev = __rb_parent;
270                         __rb_link = &__rb_parent->rb_right;
271                 }
272         }
273 
274         *pprev = NULL;
275         if (rb_prev)
276                 *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
277         *rb_link = __rb_link;
278         *rb_parent = __rb_parent;
279         return vma;
280 }
```

246 The function arguments are described above

253-255 Initialise the search

267-272 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

275-276 Get the back linking vma via the red-black tree

279 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

```
static inline void vma_link(struct mm_struct * mm,
                            struct vm_area_struct * vma,
                            struct vm_area_struct * prev,
```

```
338                        rb_node_t ** rb_link, rb_node_t * rb_parent)
339 {
340         lock_vma_mappings(vma);
341         spin_lock(&mm->page_table_lock);
342         __vma_link(mm, vma, prev, rb_link, rb_parent);
343         spin_unlock(&mm->page_table_lock);
344         unlock_vma_mappings(vma);
345
346         mm->map_count++;
347         validate_mm(mm);
348 }
```

337 `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

340 This function acquires the spinlock protecting the `address_space` representing the file that is been memory mapped.

341 Acquire the page table lock which protects the whole `mm_struct`

342 Insert the VMA

343 Free the lock protecting the `mm_struct`

345 Unlock the `address_space` for the file

346 Increase the number of mappings in this mm

347 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.

```
329 static void __vma_link(struct mm_struct * mm,
                     struct vm_area_struct * vma,
                     struct vm_area_struct * prev,
330                  rb_node_t ** rb_link, rb_node_t * rb_parent)
331 {
332         __vma_link_list(mm, vma, prev, rb_parent);
333         __vma_link_rb(mm, vma, rb_link, rb_parent);
334         __vma_link_file(vma);
335 }
```

332 This links the VMA into the linear linked lists of VMAs in this mm via the `vm_next field`

333 This links the VMA into the red-black tree of VMAs in this mm whose root is stored in the vm_rb field

334 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*)

```
282 static inline void __vma_link_list(struct mm_struct * mm,
                                 struct vm_area_struct * vma,
                                 struct vm_area_struct * prev,
283                              rb_node_t * rb_parent)
284 {
285        if (prev) {
286              vma->vm_next = prev->vm_next;
287              prev->vm_next = vma;
288        } else {
289              mm->mmap = vma;
290              if (rb_parent)
291                    vma->vm_next = rb_entry(rb_parent, struct
vm_area_struct, vm_rb);
292              else
293                    vma->vm_next = NULL;
294        }
295 }
```

285 If prev is not null, the vma is simply inserted into the list

289 Else this is the first mapping and the first element of the list has to be stored in the `mm_struct`

290 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.

```
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)
{
      rb_link_node(&vma->vm_rb, rb_parent, rb_link);
      rb_insert_color(&vma->vm_rb, &mm->mm_rb);
}
```

Function: __vma_link_file (*mm/mmap.c*)

This function links the VMA into a linked list of shared file mappings.

```
304 static inline void __vma_link_file(struct vm_area_struct * vma)
305 {
306         struct file * file;
307 
308         file = vma->vm_file;
309         if (file) {
310                 struct inode * inode = file->f_dentry->d_inode;
311                 struct address_space *mapping = inode->i_mapping;
312                 struct vm_area_struct **head;
313 
314                 if (vma->vm_flags & VM_DENYWRITE)
315                         atomic_dec(&inode->i_writecount);
316 
317                 head = &mapping->i_mmap;
318                 if (vma->vm_flags & VM_SHARED)
319                         head = &mapping->i_mmap_shared;
320 
321                 /* insert vma into inode's share list */
322                 if((vma->vm_next_share = *head) != NULL)
323                         (*head)->vm_pprev_share = &vma->vm_next_share;
324                 *head = vma;
325                 vma->vm_pprev_share = head;
326         }
327 }
```

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 `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: vma_merge (*mm/mmap.c*)

This function checks to see if a region pointed to be `prev` may be expanded forwards to cover the area from `addr` to `end` instead of allocating a new VMA. If it cannot, the VMA ahead is checked to see can it be expanded backwards instead.

```
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;
}
```

350 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

353 This is the lock to the mm struct

354-357 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`

358-375 Check to see can the region pointed to by `prev` may be expanded to cover the current region

358 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

361 Lock the mm struct

362 Expand the end of the VMA region (`vm_end`) to the end of the new mapping (`end`)

363 `next` is now the VMA in front of the newly expanded VMA

364 Check if the expanded region can be merged with the VMA in front of it

365 If it can, continue to expand the region to cover the next VMA

366 As a VMA has been merged, one region is now defunct and may be unlinked

367 No further adjustments are made to the mm struct so the lock is released

369 There is one less mapped region to reduce the `map_count`

370 Delete the struct describing the merged VMA

371 Return success

377 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

382-388 Same idea as the above block except instead of adjusted `vm_end` to cover `end`, `vm_start` is expanded to cover `addr`

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`

```
571 static inline int can_vma_merge(struct vm_area_struct * vma, unsigned long vm_flags)
572 {
573         if (!vma->vm_file && vma->vm_flags == vm_flags)
574                 return 1;
575         else
576                 return 0;
577 }
```

573 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*)

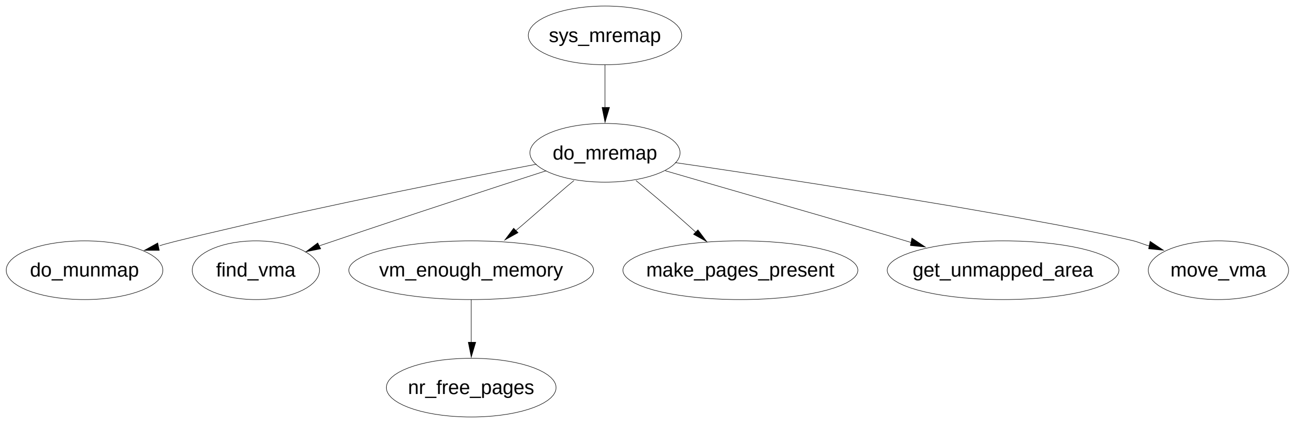

Figure 5.4: Call Graph: sys_mremap

This is the system service call to remap a memory region

```
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;
}
```

342-344 The parameters are the same as those described in the mremap man page

348 Acquire the mm semaphore

349 `do_mremap()` is the top level function for remapping a region

350 Release the mm semaphore

351 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

```
214 unsigned long do_mremap(unsigned long addr,
215       unsigned long old_len, unsigned long new_len,
216       unsigned long flags, unsigned long new_addr)
217 {
218       struct vm_area_struct *vma;
219       unsigned long ret = -EINVAL;
220 
221       if (flags & ~(MREMAP_FIXED | MREMAP_MAYMOVE))
222             goto out;
223 
224       if (addr & ~PAGE_MASK)
225             goto out;
226 
227       old_len = PAGE_ALIGN(old_len);
228       new_len = PAGE_ALIGN(new_len);
229 
```

214 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

219 At this point, the default return is EINVAL for invalid arguments

221-222 Make sure flags other than the two allowed flags are not used

224-225 The address passed in must be page aligned

227-228 Page align the passed region lengths

```
231         if (flags & MREMAP_FIXED) {
232                 if (new_addr & ~PAGE_MASK)
233                         goto out;
234                 if (!(flags & MREMAP_MAYMOVE))
235                         goto out;
236 
237                 if (new_len > TASK_SIZE || new_addr > TASK_SIZE - new_len)
238                         goto out;
239 
240                 /* Check if the location we're moving into overlaps the
241                  * old location at all, and fail if it does.
242                  */
243                 if ((new_addr <= addr) && (new_addr+new_len) > addr)
244                         goto out;
245 
246                 if ((addr <= new_addr) && (addr+old_len) > new_addr)
247                         goto out;
248 
249                 do_munmap(current->mm, new_addr, new_len);
250         }
```

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.

231 `MREMAP_FIXED` is the flag which indicates the location is fixed

232-233 The `new_addr` requested has to be page aligned

234-235 If `MREMAP_FIXED` is specified, then the `MAYMOVE` flag must be used as well

237-238 Make sure the resized region does not exceed `TASK_SIZE`

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

```
256         ret = addr;
257         if (old_len >= new_len) {
258                 do_munmap(current->mm, addr+new_len, old_len - new_len);
259                 if (!(flags & MREMAP_FIXED) || (new_addr == addr))
260                         goto out;
261         }
```

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

```
266         ret = -EFAULT;
267         vma = find_vma(current->mm, addr);
268         if (!vma || vma->vm_start > addr)
269                 goto out;
270         /* We can't remap across vm area boundaries */
271         if (old_len > vma->vm_end - addr)
272                 goto out;
273         if (vma->vm_flags & VM_DONTEXPAND) {
274                 if (new_len > old_len)
275                         goto out;
276         }
277         if (vma->vm_flags & VM_LOCKED) {
278                 unsigned long locked = current->mm->locked_vm << PAGE_SHIFT;
279                 locked += new_len - old_len;
280                 ret = -EAGAIN;
281                 if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
282                         goto out;
283         }
284         ret = -ENOMEM;
285         if ((current->mm->total_vm << PAGE_SHIFT) + (new_len - old_len)
286             > current->rlim[RLIMIT_AS].rlim_cur)
287                 goto out;
288         /* Private writable mapping? Check memory availability.. */
289         if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
290             !(flags & MAP_NORESERVE)                              &&
291             !vm_enough_memory((new_len - old_len) >> PAGE_SHIFT))
292                 goto out;
```

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

```
        if (old_len == vma->vm_end - addr &&
            !((flags & MREMAP_FIXED) && (addr != new_addr)) &&
            (old_len != new_len || !(flags & MREMAP_MAYMOVE))) {
              unsigned long max_addr = TASK_SIZE;
              if (vma->vm_next)
                    max_addr = vma->vm_next->vm_start;
              /* can we just expand the current mapping? */
              if (max_addr - addr >= new_len) {
                    int pages = (new_len - old_len) >> PAGE_SHIFT;
                    spin_lock(&vma->vm_mm->page_table_lock);
                    vma->vm_end = addr + new_len;
                    spin_unlock(&vma->vm_mm->page_table_lock);
                    current->mm->total_vm += pages;
                    if (vma->vm_flags & VM_LOCKED) {
                          current->mm->locked_vm += pages;
                          make_pages_present(addr + old_len,
                                             addr + new_len);
                    }
                    ret = addr;
                    goto out;
              }
        }
```

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

```
        ret = -ENOMEM;
        if (flags & MREMAP_MAYMOVE) {
                if (!(flags & MREMAP_FIXED)) {
                        unsigned long map_flags = 0;
                        if (vma->vm_flags & VM_SHARED)
                                map_flags |= MAP_SHARED;

                        new_addr = get_unmapped_area(vma->vm_file, 0,
                                    new_len, vma->vm_pgoff, map_flags);
                        ret = new_addr;
                        if (new_addr & ~PAGE_MASK)
                                goto out;
                }
                ret = move_vma(vma, addr, old_len, new_len, new_addr);
        }
out:
        return ret;
}
```

To expand the region, a new one has to be allocated and the old one moved to it

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 implementing **`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*)

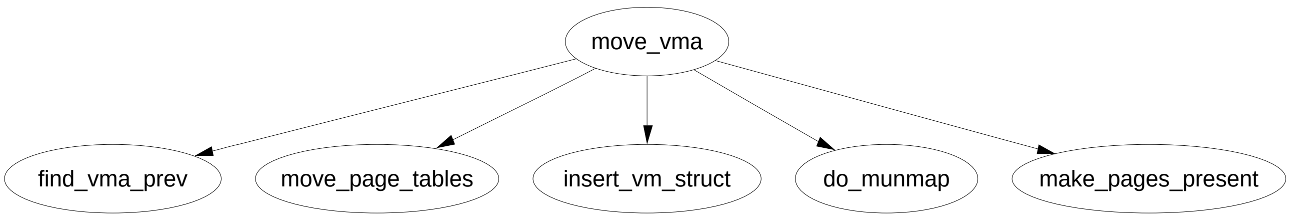

Figure 5.5: Call Graph: move_vma

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

- Call `move_page_tables()`, fill in the new VMA details if a new one was allocated and update statistics before returning

```
125 static inline unsigned long move_vma(struct vm_area_struct * vma,
126       unsigned long addr, unsigned long old_len, unsigned long new_len,
127       unsigned long new_addr)
128 {
129       struct mm_struct * mm = vma->vm_mm;
130       struct vm_area_struct * new_vma, * next, * prev;
131       int allocated_vma;
132
133       new_vma = NULL;
134       next = find_vma_prev(mm, new_addr, &prev);
```

125-127 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

134 Find the VMA preceding the address been moved to indicated by `prev` and return the region after the new mapping as `next`

```
135       if (next) {
136             if (prev && prev->vm_end == new_addr &&
137                 can_vma_merge(prev, vma->vm_flags) &&
                    !vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
138                   spin_lock(&mm->page_table_lock);
139                   prev->vm_end = new_addr + new_len;
140                   spin_unlock(&mm->page_table_lock);
141                   new_vma = prev;
142                   if (next != prev->vm_next)
143                         BUG();
144                   if (prev->vm_end == next->vm_start &&
                        can_vma_merge(next, prev->vm_flags)) {
145                         spin_lock(&mm->page_table_lock);
146                         prev->vm_end = next->vm_end;
147                         __vma_unlink(mm, next, prev);
148                         spin_unlock(&mm->page_table_lock);
149
150                         mm->map_count--;
151                         kmem_cache_free(vm_area_cachep, next);
152                   }
153             } else if (next->vm_start == new_addr + new_len &&
```

```
154                         can_vma_merge(next, vma->vm_flags) &&
                            !vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
155                      spin_lock(&mm->page_table_lock);
156                      next->vm_start = new_addr;
157                      spin_unlock(&mm->page_table_lock);
158                      new_vma = next;
159              }
160      } else {
```

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

150 There is one less mapping now so update the map_count

151 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`

```
161             prev = find_vma(mm, new_addr-1);
162             if (prev && prev->vm_end == new_addr &&
163                 can_vma_merge(prev, vma->vm_flags) && !vma->vm_file &&
                            !(vma->vm_flags & VM_SHARED)) {
164                   spin_lock(&mm->page_table_lock);
165                   prev->vm_end = new_addr + new_len;
166                   spin_unlock(&mm->page_table_lock);
167                   new_vma = prev;
168             }
169       }
```

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`

```
170
171       allocated_vma = 0;
172       if (!new_vma) {
173             new_vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
174             if (!new_vma)
175                   goto out;
176             allocated_vma = 1;
177       }
178
```

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

```
179       if (!move_page_tables(current->mm, new_addr, addr, old_len)) {
180             if (allocated_vma) {
181                   *new_vma = *vma;
```

```
182                     new_vma->vm_start = new_addr;
183                     new_vma->vm_end = new_addr+new_len;
184                     new_vma->vm_pgoff +=
                              (addr - vma->vm_start) >> PAGE_SHIFT;
185                     new_vma->vm_raend = 0;
186                     if (new_vma->vm_file)
187                           get_file(new_vma->vm_file);
188                     if (new_vma->vm_ops && new_vma->vm_ops->open)
189                           new_vma->vm_ops->open(new_vma);
190                     insert_vm_struct(current->mm, new_vma);
191               }
192               do_munmap(current->mm, addr, old_len);
193               current->mm->total_vm += new_len >> PAGE_SHIFT;
194               if (new_vma->vm_flags & VM_LOCKED) {
195                     current->mm->locked_vm += new_len >> PAGE_SHIFT;
196                     make_pages_present(new_vma->vm_start,
197                                        new_vma->vm_end);
198               }
199               return new_addr;
200         }
201         if (allocated_vma)
202               kmem_cache_free(vm_area_cachep, new_vma);
203  out:
204         return -ENOMEM;
205 }
```

179 `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

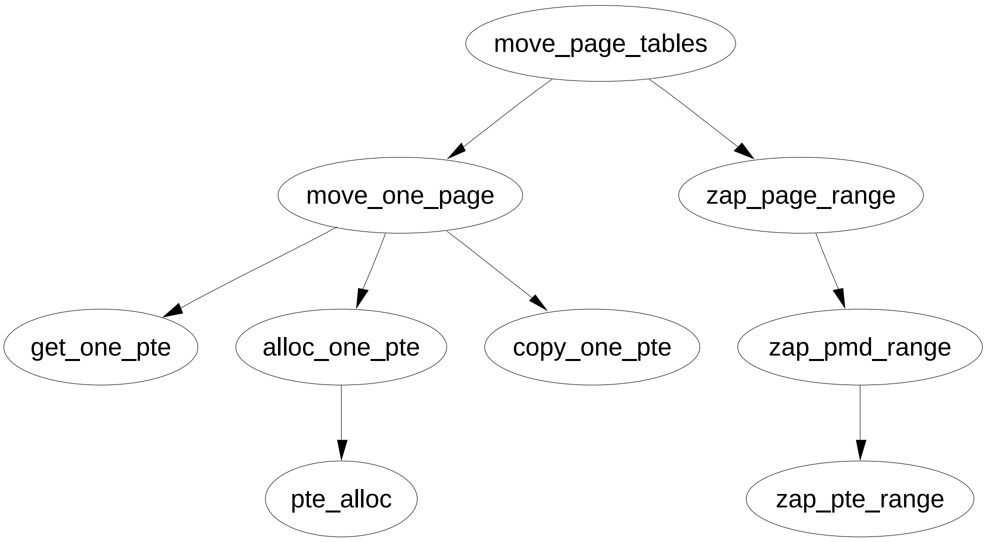

Figure 5.6: Call Graph: `move_page_tables()`

Function: move_page_tables (*mm/mremap.c*)

This function is responsible copying all the page table entries from the region pointed to be `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.

```
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;
```

```
123 }
```

90 The parameters are the mm for the process, the new location, the old location and the length of the region to move entries for

95 `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

102-106 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

107 Flush the TLB for the old region

108 Return success

118-120 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

121 Zap any pages that were allocated for the move

122 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()`

```
77 static int move_one_page(struct mm_struct *mm,
                      unsigned long old_addr, unsigned long new_addr)
78 {
79         int error = 0;
80         pte_t * src;
81 
82         spin_lock(&mm->page_table_lock);
83         src = get_one_pte(mm, old_addr);
84         if (src)
85                 error = copy_one_pte(mm, src, alloc_one_pte(mm, new_addr));
86         spin_unlock(&mm->page_table_lock);
87         return error;
88 }
```

82 Acquire the mm lock

83 Call `get_one_pte()` which walks the page tables to get the correct PTE

84-85 If the PTE exists, allocate a PTE for the destination and call `copy_one_pte()` to copy the PTEs

86 Release the lock

87 Return whatever `copy_one_pte()` returned

Function: get_one_pte (*mm/mremap.c*)

This is a very simple page table walk.

```
18 static inline pte_t *get_one_pte(struct mm_struct *mm, unsigned long addr)
19 {
20         pgd_t * pgd;
21         pmd_t * pmd;
22         pte_t * pte = NULL;
23 
24         pgd = pgd_offset(mm, addr);
25         if (pgd_none(*pgd))
26                 goto end;
27         if (pgd_bad(*pgd)) {
28                 pgd_ERROR(*pgd);
29                 pgd_clear(pgd);
30                 goto end;
31         }
32 
33         pmd = pmd_offset(pgd, addr);
34         if (pmd_none(*pmd))
35                 goto end;
36         if (pmd_bad(*pmd)) {
37                 pmd_ERROR(*pmd);
38                 pmd_clear(pmd);
39                 goto end;
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 {
51         pmd_t * pmd;
52         pte_t * pte = NULL;
53 
54         pmd = pmd_alloc(mm, pgd_offset(mm, addr), addr);
55         if (pmd)
56                 pte = pte_alloc(mm, pmd, addr);
57         return pte;
58 }
```

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 {
62         int error = 0;
63         pte_t pte;
64 
65         if (!pte_none(*src)) {
66                 pte = ptep_get_and_clear(src);
67                 if (!dst) {
68                         /* No dest?  We must put it back. */
69                         dst = src;
70                         error++;
71                 }
72                 set_pte(dst, pte);
73         }
74         return error;
75 }
```

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 exceeeded and that the region to lock is page aligned.

```
195 asmlinkage long sys_mlock(unsigned long start, size_t len)
196 {
197         unsigned long locked;
198         unsigned long lock_limit;
199         int error = -ENOMEM;
200 
201         down_write(&current->mm->mmap_sem);
202         len = PAGE_ALIGN(len + (start & ~PAGE_MASK));
203         start &= PAGE_MASK;
204 
205         locked = len >> PAGE_SHIFT;
206         locked += current->mm->locked_vm;
207 
208         lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
209         lock_limit >>= PAGE_SHIFT;
210 
211         /* check against resource limits */
212         if (locked > lock_limit)
213                 goto out;
214 
215         /* we may lock at most half of physical memory... */
216         /* (this check is pretty bogus, but doesn't hurt) */
217         if (locked > num_physpages/2)
218                 goto out;
219 
220         error = do_mlock(start, len, 1);
221 out:
222         up_write(&current->mm->mmap_sem);
223         return error;
224 }
```

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

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 clostest 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.

```
238 static int do_mlockall(int flags)
239 {
240         int error;
241         unsigned int def_flags;
242         struct vm_area_struct * vma;
243 
244         if (!capable(CAP_IPC_LOCK))
245               return -EPERM;
246 
247         def_flags = 0;
248         if (flags & MCL_FUTURE)
249               def_flags = VM_LOCKED;
250         current->mm->def_flags = def_flags;
251 
252         error = 0;
253         for (vma = current->mm->mmap; vma ; vma = vma->vm_next) {
254               unsigned int newflags;
255 
256               newflags = vma->vm_flags | VM_LOCKED;
257               if (!(flags & MCL_CURRENT))
258                     newflags &= ~VM_LOCKED;
259               error = mlock_fixup(vma, vma->vm_start, vma->vm_end,
newflags);
260               if (error)
261                     break;
262         }
263         return error;
264 }
```

244-245 The calling process must be either root or have `CAP_IPC_LOCK` capabilities

248-250 The `MCL_FUTURE` flag says that all future pages should be locked so if set, the `def_flags` for VMAs should be `VM_LOCKED`

253-262 Cycle through all VMAs

256 Set the `VM_LOCKED` flag in the current VMA flags

257-258 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

259 Call `mlock_fixup()` which will adjust the regions as necessary

260-261 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

263 Return the success or error value

Function: do_mlock (*mm/mlock.c*)

This function is 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.

```
148 static int do_mlock(unsigned long start, size_t len, int on)
149 {
150         unsigned long nstart, end, tmp;
151         struct vm_area_struct * vma, * next;
152         int error;
153 
154         if (on && !capable(CAP_IPC_LOCK))
155                 return -EPERM;
156         len = PAGE_ALIGN(len);
157         end = start + len;
158         if (end < start)
159                 return -EINVAL;
160         if (end == start)
161                 return 0;
162         vma = find_vma(current->mm, start);
163         if (!vma || vma->vm_start > start)
164                 return -ENOMEM;
```

Page align the request and find the VMA

154 Only root processes can lock pages

156 Page align the length despite it being already done in the calling function. This is probably an oversight

157-159 Calculate the end of the locking and make sure it is a valid region. Return -EINVAL if it is not

160-161 if locking a region of size 0, just return

162 Find the VMA that will be affected by this locking

163-164 If the VMA for this address range does not exist, return -ENOMEM

```
165
166         for (nstart = start ; ; ) {
167                 unsigned int newflags;
168
170
171                 newflags = vma->vm_flags | VM_LOCKED;
172                 if (!on)
173                         newflags &= ~VM_LOCKED;
174
175                 if (vma->vm_end >= end) {
176                         error = mlock_fixup(vma, nstart, end, newflags);
177                         break;
178                 }
179
180                 tmp = vma->vm_end;
181                 next = vma->vm_next;
182                 error = mlock_fixup(vma, nstart, tmp, newflags);
183                 if (error)
184                         break;
185                 nstart = tmp;
186                 vma = next;
187                 if (!vma || vma->vm_start != nstart) {
188                         error = -ENOMEM;
189                         break;
190                 }
191         }
192         return error;
193 }
```

Walk through the VMAs affected by this locking and call `mlock_fixup()` for each of them.

166-192 Cycle through as many VMAs as necessary to lock the pages

171 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 paramter 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

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()`.

```
static int mlock_fixup(struct vm_area_struct * vma,
        unsigned long start, unsigned long end, unsigned int newflags)
{
        int pages, retval;

        if (newflags == vma->vm_flags)
                return 0;

        if (start == vma->vm_start) {
                if (end == vma->vm_end)
```

```
127                 retval = mlock_fixup_all(vma, newflags);
128           else
129                 retval = mlock_fixup_start(vma, end, newflags);
130     } else {
131           if (end == vma->vm_end)
132                 retval = mlock_fixup_end(vma, start, newflags);
133           else
134                 retval = mlock_fixup_middle(vma, start,
                                      end, newflags);
135     }
136     if (!retval) {
137           /* keep track of amount of locked VM */
138           pages = (end - start) >> PAGE_SHIFT;
139           if (newflags & VM_LOCKED) {
140                 pages = -pages;
141                 make_pages_present(start, end);
142           }
143           vma->vm_mm->locked_vm -= pages;
144     }
145     return retval;
146 }
```

122-123 If no change is to be made, just return

125 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

126-127 The full VMA is been locked, call `mlock_fixup_all()`

128-129 Only a portion is to be locked, call `mlock_fixup_start()`

130 Else either the a region at the end is to be locked or a region in the middle

131-132 The end of the locking match the end of the VMA, call `mlock_fixup_end()`

133-134 A region in the middle is to be locked, call `mlock_fixup_middle()`

136-144 The fixup functions return 0 on success. If the fixup of the regions succeed and the regions are now marked as locked, call `make_pages_present()` which makes some basic checks before calling `get_user_pages()` which faults in all the pages in the same way the page fault handler does

Function: mlock_fixup_all (*mm/mlock.c*)

```
15 static inline int mlock_fixup_all(struct vm_area_struct * vma,
                            int newflags)
16 {
17     spin_lock(&vma->vm_mm->page_table_lock);
```

```
18         vma->vm_flags = newflags;
19         spin_unlock(&vma->vm_mm->page_table_lock);
20         return 0;
21 }
```

17-19 Trivial, lock the VMA with the spinlock, set the new flags, release the lock and return success

Function: mlock_fixup_start (*mm/mlock.c*)

Slightly more compilcated. A new VMA is required to represent the affected region. The start of the old VMA is moved forward

```
23 static inline int mlock_fixup_start(struct vm_area_struct * vma,
24         unsigned long end, int newflags)
25 {
26         struct vm_area_struct * n;
27 
28         n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
29         if (!n)
30                 return -EAGAIN;
31         *n = *vma;
32         n->vm_end = end;
33         n->vm_flags = newflags;
34         n->vm_raend = 0;
35         if (n->vm_file)
36                 get_file(n->vm_file);
37         if (n->vm_ops && n->vm_ops->open)
38                 n->vm_ops->open(n);
39         vma->vm_pgoff += (end - vma->vm_start) >> PAGE_SHIFT;
40         lock_vma_mappings(vma);
41         spin_lock(&vma->vm_mm->page_table_lock);
42         vma->vm_start = end;
43         __insert_vm_struct(current->mm, n);
44         spin_unlock(&vma->vm_mm->page_table_lock);
45         unlock_vma_mappings(vma);
46         return 0;
47 }
```

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

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 
54       n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
55       if (!n)
56             return -EAGAIN;
57       *n = *vma;
58       n->vm_start = start;
59       n->vm_pgoff += (n->vm_start - vma->vm_start) >> PAGE_SHIFT;
60       n->vm_flags = newflags;
61       n->vm_raend = 0;
62       if (n->vm_file)
63             get_file(n->vm_file);
64       if (n->vm_ops && n->vm_ops->open)
65             n->vm_ops->open(n);
66       lock_vma_mappings(vma);
67       spin_lock(&vma->vm_mm->page_table_lock);
68       vma->vm_end = start;
69       __insert_vm_struct(current->mm, n);
70       spin_unlock(&vma->vm_mm->page_table_lock);
71       unlock_vma_mappings(vma);
72       return 0;
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

62-63 If the VMA has a file or device mapping, `get_file()` will increment the reference count

64-65 If an `open()` function is provided, call it

66 `lock_vma_mappings()` will lock any files if this VMA is a shared region

67-70 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

71 Unlock the file mappings with `unlock_vma_mappings()`

72 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.

```
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);
      }
```

```
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 accidently 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

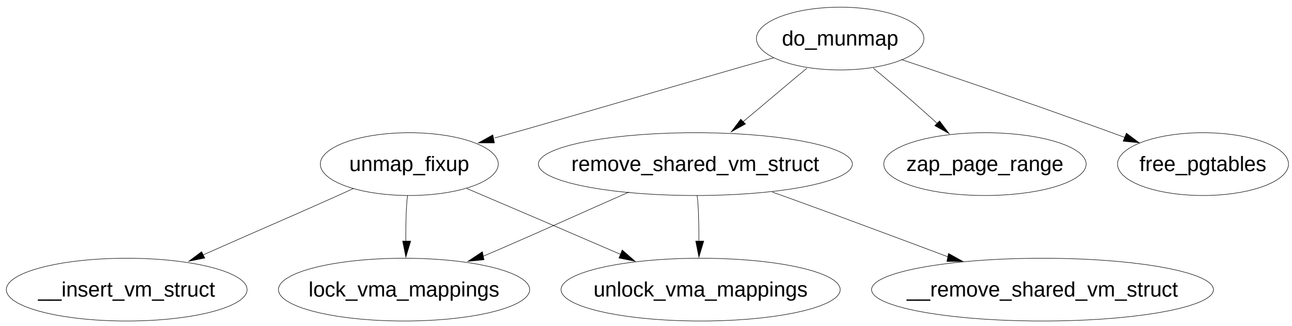

Figure 5.7: do_munmap

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

```
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  */
```

```
938
939         if (mpnt->vm_start >= addr+len)
940               return 0;
941
943         if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
944             && mm->map_count >= max_map_count)
945               return -ENOMEM;
946
951         extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
952         if (!extra)
953               return -ENOMEM;
```

919 The parameters are as follows;

mmThe mm for the processes performing the unmap operation

addrThe starting address of the region to unmap

lenThe length of the region

923-924 Ensure the address is page aligned and that the area to be unmapped is not in the kernel virtual address space

926-927 Make sure the region size to unmap is page aligned

934 Find the VMA that contains the starting address and the preceding VMA so it can be easily unlinked later

935-936 If no mpnt was returned, it means the address must be past the last used VMA so the address space is unused, just return

939-940 If the returned VMA starts past the region we are trying to unmap, then the region in unused, just return

943-945 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

951-953 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

```
955         npp = (prev ? &prev->vm_next : &mm->mmap);
956         free = NULL;
957         spin_lock(&mm->page_table_lock);
958         for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
959               *npp = mpnt->vm_next;
960               mpnt->vm_next = free;
961               free = mpnt;
962               rb_erase(&mpnt->vm_rb, &mm->mm_rb);
963         }
```

```
964         mm->mmap_cache = NULL;  /* Kill the cache. */
965         spin_unlock(&mm->page_table_lock);
```

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.

- 955 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
- 956 free is the head of a linked list of VMAs that are affected by the unmapping
- 957 Lock the mm
- 958 Cycle through the list until the start of the current VMA is past the end of the region to be unmapped
- 959 npp becomes the next VMA in the list
- 960-961 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
- 962 Delete mpnt from the red-black tree
- 964 Remove the cached result in case the last looked up result is one of the regions to be unmapped
- 965 Free the mm

```
966
967         /* Ok - we have the memory areas we should free on the 'free'
list,
968          * so release them, and unmap the page range..
969          * If the one of the segments is only being partially unmapped,
970          * it will put new vm_area_struct(s) into the address space.
971          * In that case we have to be careful with VM_DENYWRITE.
972          */
973         while ((mpnt = free) != NULL) {
974               unsigned long st, end, size;
975               struct file *file = NULL;
976
977               free = free->vm_next;
978
979               st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
980               end = addr+len;
981               end = end > mpnt->vm_end ? mpnt->vm_end : end;
982               size = end - st;
983
```

```
984                 if (mpnt->vm_flags & VM_DENYWRITE &&
985                     (st != mpnt->vm_start || end != mpnt->vm_end) &&
986                     (file = mpnt->vm_file) != NULL) {
987                      atomic_dec(&file->f_dentry->d_inode->i_writecount);
988                 }
989                 remove_shared_vm_struct(mpnt);
990                 mm->map_count--;
991
992                 zap_page_range(mm, st, size);
993
994                 /*
995                  * Fix the mapping, and free the old area if it wasn't reused.
996                  */
997                 extra = unmap_fixup(mm, mpnt, st, size, extra);
998                 if (file)
999                    atomic_inc(&file->f_dentry->d_inode->i_writecount);
1000        }
```

973 Keep stepping through the list until no VMAs are left

977 Move free to the next element in the list leaving mpnt as the head about to be removed

979 `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

980-981 Calculate the end of the region to map in a similar fashion

982 Calculate the size of the region to be unmapped in this pass

984-988 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

989 Remove the file mapping. If the file is still partially mapped, it will be acquired again during `unmap_fixup()`

990 Reduce the map count

992 Remove all pages within this region

997 Call the fixup routing

998-999 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

```
1001         validate_mm(mm);
1002
1003         /* Release the extra vma struct if it wasn't used */
1004         if (extra)
1005               kmem_cache_free(vm_area_cachep, extra);
1006
1007         free_pgtables(mm, prev, addr, addr+len);
1008
1009         return 0;
1010 }
```

1001 A debugging function only. If enabled, it will ensure the VMA tree for this mm is still valid

1004-1005 If extra VMA was not required, delete it

1007 Free all the page tables that were used for the unmapped region

1009 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.

```
785 static struct vm_area_struct * unmap_fixup(struct mm_struct *mm,
786        struct vm_area_struct *area, unsigned long addr, size_t len,
787        struct vm_area_struct *extra)
788 {
789        struct vm_area_struct *mpnt;
790        unsigned long end = addr + len;
791
792        area->vm_mm->total_vm -= len >> PAGE_SHIFT;
793        if (area->vm_flags & VM_LOCKED)
794              area->vm_mm->locked_vm -= len >> PAGE_SHIFT;
795
```

Function preamble.

785 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 `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         /* Unmapping the whole area. */
797         if (addr == area->vm_start && end == area->vm_end) {
798                 if (area->vm_ops && area->vm_ops->close)
799                         area->vm_ops->close(area);
800                 if (area->vm_file)
801                         fput(area->vm_file);
802                 kmem_cache_free(vm_area_cachep, area);
803                 return extra;
804         }
```

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) {
808                 /*
809                  * here area isn't visible to the semaphore-less readers
810                  * so we don't need to update it under the spinlock.
811                  */
812                 area->vm_end = addr;
813                 lock_vma_mappings(area);
814                 spin_lock(&mm->page_table_lock);
815         }
```

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

813 If a file/device is being mapped, the lock protecting shared access to it is taken in the function `lock_vm_mappings()`

814 Lock the mm. Later in the function, the remaining VMA will be reinserted into the mm

```
815               else if (addr == area->vm_start) {
816             area->vm_pgoff += (end - area->vm_start) >> PAGE_SHIFT;
817             /* same locking considerations of the above case */
818             area->vm_start = end;
819             lock_vma_mappings(area);
820             spin_lock(&mm->page_table_lock);
821       }
```

Handle the case where the VMA is been unmapped from the start to some part in the middle

816 Increase the offset within the file/device mapped by the number of pages this unmapping represents

818 Move the start of the VMA to the end of the region being unmapped

819-820 Lock the file/device and mm as above

```
            else {
822       /* Unmapping a hole: area->vm_start < addr <= end < area->vm_end */
823             /* Add end mapping -- leave beginning for below */
824             mpnt = extra;
825             extra = NULL;
826
827             mpnt->vm_mm = area->vm_mm;
828             mpnt->vm_start = end;
829             mpnt->vm_end = area->vm_end;
830             mpnt->vm_page_prot = area->vm_page_prot;
831             mpnt->vm_flags = area->vm_flags;
832             mpnt->vm_raend = 0;
833             mpnt->vm_ops = area->vm_ops;
834             mpnt->vm_pgoff = area->vm_pgoff +
                             ((end - area->vm_start) >> PAGE_SHIFT);
835             mpnt->vm_file = area->vm_file;
836             mpnt->vm_private_data = area->vm_private_data;
837             if (mpnt->vm_file)
838                   get_file(mpnt->vm_file);
839             if (mpnt->vm_ops && mpnt->vm_ops->open)
840                   mpnt->vm_ops->open(mpnt);
841             area->vm_end = addr;    /* Truncate area */
842
```

```
843             /* Because mpnt->vm_file == area->vm_file this locks
844              * things correctly.
845              */
846             lock_vma_mappings(area);
847             spin_lock(&mm->page_table_lock);
848             __insert_vm_struct(mm, mpnt);
849     }
```

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

824-825 Take the extra VMA and make VMA NULL so that the calling function will know it is in use and cannot be freed

826-836 Copy in all the VMA information

837 If a file/device is mapped, get a reference to it with `get_file()`

839-840 If an open function is provided, call it

841 Truncate the VMA so that it ends at the start of the region to be unmapped

846-847 Lock the files and mm as with the two previous cases

848 Insert the extra VMA into the mm

```
850
851     __insert_vm_struct(mm, area);
852     spin_unlock(&mm->page_table_lock);
853     unlock_vma_mappings(area);
854     return extra;
855 }
```

851 Reinsert the VMA into the mm

852 Unlock the page tables

853 Unlock the spinlock to the shared mapping

854 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.

```
1122 void exit_mmap(struct mm_struct * mm)
1123 {
1124         struct vm_area_struct * mpnt;
1125 
1126         release_segments(mm);
1127         spin_lock(&mm->page_table_lock);
1128         mpnt = mm->mmap;
1129         mm->mmap = mm->mmap_cache = NULL;
1130         mm->mm_rb = RB_ROOT;
1131         mm->rss = 0;
1132         spin_unlock(&mm->page_table_lock);
1133         mm->total_vm = 0;
1134         mm->locked_vm = 0;
1135 
1136         flush_cache_mm(mm);
1137         while (mpnt) {
1138                 struct vm_area_struct * next = mpnt->vm_next;
1139                 unsigned long start = mpnt->vm_start;
1140                 unsigned long end = mpnt->vm_end;
1141                 unsigned long size = end - start;
1142 
1143                 if (mpnt->vm_ops) {
1144                         if (mpnt->vm_ops->close)
1145                                 mpnt->vm_ops->close(mpnt);
1146                 }
1147                 mm->map_count--;
1148                 remove_shared_vm_struct(mpnt);
1149                 zap_page_range(mm, start, size);
1150                 if (mpnt->vm_file)
1151                         fput(mpnt->vm_file);
1152                 kmem_cache_free(vm_area_cachep, mpnt);
1153                 mpnt = next;
1154         }
1155         flush_tlb_mm(mm);
1156 
1157         /* This is just debugging */
1158         if (mm->map_count)
1159                 BUG();
1160 
1161         clear_page_tables(mm, FIRST_USER_PGD_NR, USER_PTRS_PER_PGD);
1162 }
```

1126 `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

1127 Lock the mm

1128 mpnt becomes the first VMA on the list

1129 Clear VMA related information from the mm so it may be unlocked

1132 Unlock the mm

1133-1134 Clear the mm statistics

1136 Flush the CPU for the address range

1137-1154 Step through every VMA that was associated with the mm

1138 Record what the next VMA to clear will be so this one may be deleted

1139-1141 Record the start, end and size of the region to be deleted

1143-1146 If there is a close operation associated with this VMA, call it

1147 Reduce the map count

1148 Remove the file/device mapping from the shared mappings list

1149 Free all pages associated with this region

1150-1151 If a file/device was mapped in this region, free it

1152 Free the VMA struct

1153 Move to the next VMA

1155 Flush the TLB for this whole mm as it is about to be unmapped

1158-1159 If the `map_count` is positive, it means the map count was not accounted for properly so call `BUG()` to mark it

1161 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;
```

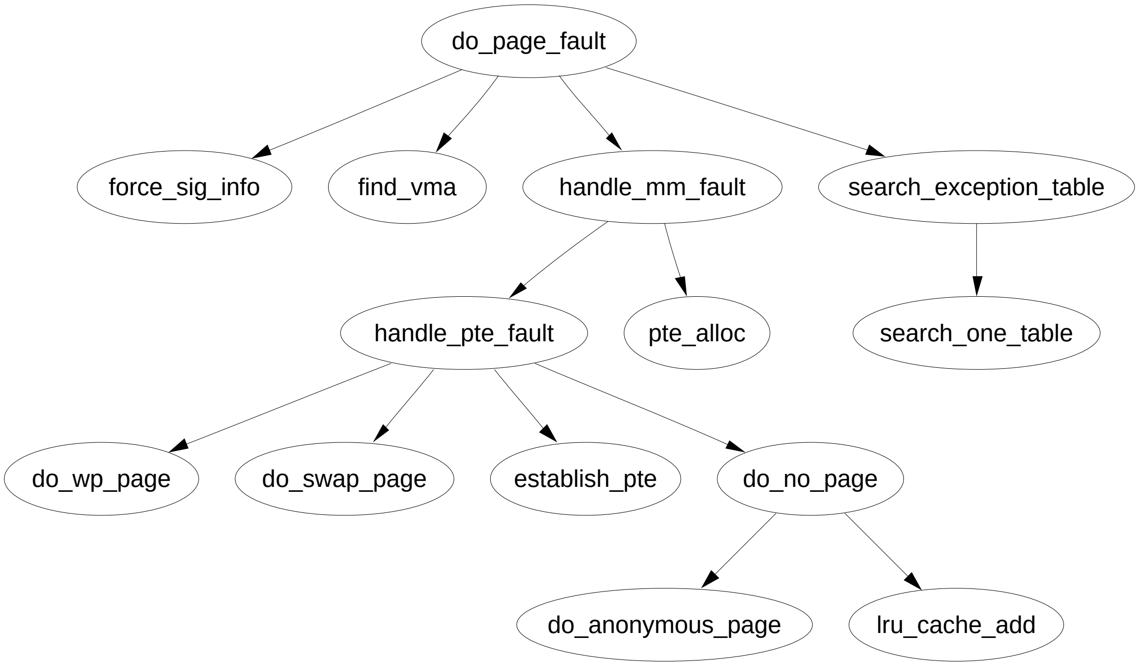

Figure 5.8: Call Graph: `do_page_fault()`

```
147         unsigned long fixup;
148         int write;
149         siginfo_t info;
150
151         /* get the address */
152         __asm__("movl %%cr2,%0":"=r" (address));
153
154         /* It's safe to allow irq's after cr2 has been saved */
155         if (regs->eflags & X86_EFLAGS_IF)
156               local_irq_enable();
157
158         tsk = current;
159
```

Function preamble. Get the fault address and enable interrupts

140 The parameters are

- `regs` is a struct containing what all the registers at fault time
- `error_code` indicates what sort of fault occurred

152 As the comment indicates, the cr2 register is the fault addres

155-156 If the fault is from within an interrupt, enable them

158 Set the current task

```
173         if (address >= TASK_SIZE && !(error_code & 5))
174                 goto vmalloc_fault;
175 
176         mm = tsk->mm;
177         info.si_code = SEGV_MAPERR;
178 
183         if (in_interrupt() || !mm)
184                 goto no_context;
185 
```

Check for exceptional faults, kernel faults, fault in interrupt and fault with no memory context

173 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

176 Record the working mm

183 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

```
186         down_read(&mm->mmap_sem);
187 
188         vma = find_vma(mm, address);
189         if (!vma)
190                 goto bad_area;
191         if (vma->vm_start <= address)
192                 goto good_area;
193         if (!(vma->vm_flags & VM_GROWSDOWN))
194                 goto bad_area;
195         if (error_code & 4) {
196                 /*
197                  * accessing the stack below %esp is always a bug.
198                  * The "+ 32" is there due to some instructions (like
199                  * pusha) doing post-decrement on the stack and that
200                  * doesn't show up until later..
201                  */
202                 if (address + 32 < regs->esp)
203                         goto bad_area;
204         }
205         if (expand_stack(vma, address))
206                 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

186 Take the long lived mm semaphore

188 Find the VMA that is responsible or is closest to the faulting address

189-190 If a VMA does not exist at all, goto `bad_area`

191-192 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

193-194 For the region that is closest, check if it can gown down (`VM_GROWSDOWN`). If it does, it means the stack can probably be expanded. If not, goto `bad_area`

195-204 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

205-206 expand the stack, if it fails, goto `bad_area`

```
211 good_area:
212         info.si_code = SEGV_ACCERR;
213         write = 0;
214         switch (error_code & 3) {
215                 default:        /* 3: write, present */
216 #ifdef TEST_VERIFY_AREA
217                         if (regs->cs == KERNEL_CS)
218                                 printk("WP fault at %08lx\n", regs->eip);
219 #endif
220                         /* fall through */
221                 case 2:         /* write, not present */
222                         if (!(vma->vm_flags & VM_WRITE))
223                                 goto bad_area;
224                         write++;
225                         break;
226                 case 1:         /* read, present */
227                         goto bad_area;
228                 case 0:         /* read, not present */
229                         if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
230                                 goto bad_area;
231         }
```

There is the first part of a good area is handled. The permissions need to be checked in case this is a protection fault.

212 By default return an error

214 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

215 If it is 3, both bits are 1 so it is a write protection fault

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
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

246-247 A return of 0 means some IO error happened during the fault so go to the do_sigbus handler

248-249 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

255-259 Not sure

260 Release the lock to the mm

261 Return as the fault has been successfully handled

```
bad_area:
        up_read(&mm->mmap_sem);

        /* User mode accesses just cause a SIGSEGV */
        if (error_code & 4) {
                tsk->thread.cr2 = address;
                tsk->thread.error_code = error_code;
                tsk->thread.trap_no = 14;
                info.si_signo = SIGSEGV;
                info.si_errno = 0;
                /* info.si_code has been set above */
                info.si_addr = (void *)address;
                force_sig_info(SIGSEGV, &info, tsk);
                return;
        }

        /*
         * Pentium F0 0F C7 C8 bug workaround.
         */
        if (boot_cpu_data.f00f_bug) {
                unsigned long nr;

                nr = (address - idt) >> 3;

                if (nr == 6) {
                        do_invalid_op(regs, 0);
                        return;
                }
        }
```

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.

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 *)
                        __va(page))[address >> PAGE_SHIFT];
325             printk(KERN_ALERT "*pte = %08lx\n", page);
326       }
327       die("Oops", regs, error_code);
328       bust_spinlocks(0);
329       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;
```

```
339         }
340         up_read(&mm->mmap_sem);
341         printk("VM: killing process %s\n", tsk->comm);
342         if (error_code & 4)
343               do_exit(SIGKILL);
344         goto no_context;
```

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
353         tsk->thread.cr2 = address;
354         tsk->thread.error_code = error_code;
355         tsk->thread.trap_no = 14;
356         info.si_signo = SIGBUS;
357         info.si_errno = 0;
358         info.si_code = BUS_ADRERR;
359         info.si_addr = (void *)address;
360         force_sig_info(SIGBUS, &info, tsk);
361
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

365 If in userspace, just return and the process will die in due course

```
367 vmalloc_fault:
368      {
376            int offset = __pgd_offset(address);
377            pgd_t *pgd, *pgd_k;
378            pmd_t *pmd, *pmd_k;
379            pte_t *pte_k;
380
381            asm("movl %%cr3,%0":"=r" (pgd));
382            pgd = offset + (pgd_t *)__va(pgd);
383            pgd_k = init_mm.pgd + offset;
384
385            if (!pgd_present(*pgd_k))
386                  goto no_context;
387            set_pgd(pgd, *pgd_k);
388
389            pmd = pmd_offset(pgd, address);
390            pmd_k = pmd_offset(pgd_k, address);
391            if (!pmd_present(*pmd_k))
392                  goto no_context;
393            set_pmd(pmd, *pmd_k);
394
395            pte_k = pte_offset(pmd_k, address);
396            if (!pte_present(*pte_k))
397                  goto no_context;
398            return;
399      }
400 }
```

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.

376 Get the offset within a PGD

381 Copy the address of the PGD for the process from the cr3 register to pgd

382 Calculate the pgd pointer from the process PGD

383 Calculate for the kernel reference PGD

385-386 If the pgd entry is invalid for the kernel page table, goto no_context

386 Set the page table entry in the process page table with a copy from the kernel reference page table

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

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.

```
1364 int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct * vma,
1365       unsigned long address, int write_access)
1366 {
1367       pgd_t *pgd;
1368       pmd_t *pmd;
1369 
1370       current->state = TASK_RUNNING;
1371       pgd = pgd_offset(mm, address);
1372 
1373       /*
1374        * We need the page table lock to synchronize with kswapd
1375        * and the SMP-safe atomic PTE updates.
1376        */
1377       spin_lock(&mm->page_table_lock);
1378       pmd = pmd_alloc(mm, pgd, address);
1379 
1380       if (pmd) {
1381             pte_t * pte = pte_alloc(mm, pmd, address);
1382             if (pte)
1383                   return handle_pte_fault(mm, vma, address,
                                  write_access, pte);
1384       }
1385       spin_unlock(&mm->page_table_lock);
1386       return -1;
1387 }
```

1364 The parameters of the function are;

`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.

```
static inline int handle_pte_fault(struct mm_struct *mm,
        struct vm_area_struct * vma, unsigned long address,
        int write_access, pte_t * pte)
{
        pte_t entry;

        entry = *pte;
        if (!pte_present(entry)) {
                /*
                 * If it truly wasn't present, we know that kswapd
                 * and the PTE updates will not touch it later. So
                 * drop the lock.
                 */
                if (pte_none(entry))
                        return do_no_page(mm, vma, address,
                                          write_access, pte);
                return do_swap_page(mm, vma, address, pte, entry,
```

```
                                    write_access);
1347        }
1348
1349        if (write_access) {
1350              if (!pte_write(entry))
1351                    return do_wp_page(mm, vma, address, pte, entry);
1352
1353              entry = pte_mkdirty(entry);
1354        }
1355        entry = pte_mkyoung(entry);
1356        establish_pte(vma, address, pte, entry);
1357        spin_unlock(&mm->page_table_lock);
1358        return 1;
1359 }
```

1331 The parameters of the function are the same as those for `handle_mm_fault()` except the PTE for the fault is included

1337 Record the PTE

1338 Handle the case where the PTE is not present

1344 If the PTE has never been filled, handle the allocation of the PTE with `do_no_page()`

1346 If the page has been swapped out to backing storage, handle it with `do_swap_page()`

1349-1354 Handle the case where the page is been written to

1350-1351 If the PTE is marked write-only, it is a COW page so handle it with `do_wp_page()`

1353 Otherwise just simply mark the page as dirty

1355 Mark the page as accessed

1356 `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

1357 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;

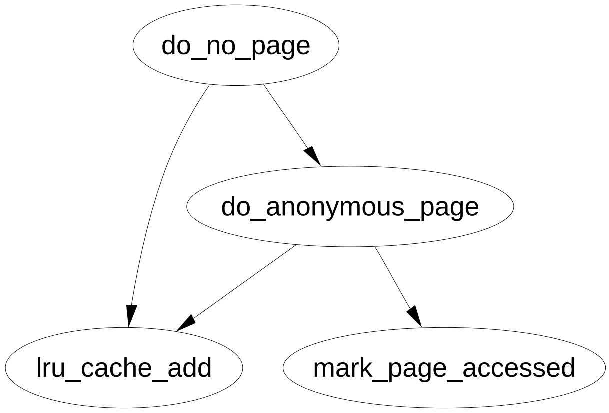

Figure 5.9: Call Graph: `do_no_page()`

- 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

```
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 
1251       if (!vma->vm_ops || !vma->vm_ops->nopage)
1252             return do_anonymous_page(mm, vma, page_table,
                                    write_access, address);
1253       spin_unlock(&mm->page_table_lock);
1254 
1255       new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
1256 
1257       if (new_page == NULL)   /* no page was available -- SIGBUS */
1258             return 0;
1259       if (new_page == NOPAGE_OOM)
1260             return -1;
```

1245 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

1253 Otherwise free the page table lock as the `nopage()` function can not be called with spinlocks held

1255 Call the supplied nopage function, in the case of filesystems, this is frequently `filemap_nopage()` but will be different for each device driver

1257-1258 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 been sent to the faulting process

1259-1260 If NOPAGE_OOM is returned, the physical page allocator failed to allocate a page and -1 is returned which will forcibly kill the process

```
1265        if (write_access && !(vma->vm_flags & VM_SHARED)) {
1266              struct page * page = alloc_page(GFP_HIGHUSER);
1267              if (!page) {
1268                    page_cache_release(new_page);
1269                    return -1;
1270              }
1271              copy_user_highpage(page, new_page, address);
1272              page_cache_release(new_page);
1273              lru_cache_add(page);
1274              new_page = page;
1275        }
```

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.

1265 Check if COW should be broken early

1266 If so, allocate a new page for the process

1267-1270 If the page could not be allocated, reduce the reference count to the page returned by the `nopage()` function and return -1 for out of memory

1271 Otherwise copy the contents

1272 Reduce the reference count to the returned page which may still be in use by another process

1273 Add the new page to the LRU lists so it may be reclaimed by kswapd later

```
1276
1277        spin_lock(&mm->page_table_lock);
1288        /* Only go through if we didn't race with anybody else... */
1289        if (pte_none(*page_table)) {
1290              ++mm->rss;
1291              flush_page_to_ram(new_page);
```

```
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 {
1192       pte_t entry;
1193
1194       /* Read-only mapping of ZERO_PAGE. */
1195       entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));
1196
1197       /* ..except if it's a write access */
1198       if (write_access) {
1199             struct page *page;
1200
1201             /* Allocate our own private page. */
1202             spin_unlock(&mm->page_table_lock);
1203
1204             page = alloc_page(GFP_HIGHUSER);
1205             if (!page)
1206                   goto no_mem;
1207             clear_user_highpage(page, addr);
1208
1209             spin_lock(&mm->page_table_lock);
1210             if (!pte_none(*page_table)) {
1211                   page_cache_release(page);
1212                   spin_unlock(&mm->page_table_lock);
1213                   return 1;
1214             }
1215             mm->rss++;
1216             flush_page_to_ram(page);
1217             entry = pte_mkwrite(
                         pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
1218             lru_cache_add(page);
1219             mark_page_accessed(page);
```

```
1220         }
1221 
1222         set_pte(page_table, entry);
1223 
1224         /* No need to invalidate - it was non-present before */
1225         update_mmu_cache(vma, addr, entry);
1226         spin_unlock(&mm->page_table_lock);
1227         return 1;       /* Minor fault */
1228 
1229 no_mem:
1230         return -1;
1231 }
```

1190 The parameters are the same as those passed to `handle_pte_fault()`

1195 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

1198-1220 If this is a write fault, then allocate a new page and zero fill it

1202 Unlock the `mm_struct` as the allocation of a new page could sleep

1204 Allocate a new page

1205 If a page could not be allocated, return -1 to handle the OOM situation

1207 Zero fill the page

1209 Reacquire the lock as the page tables are to be updated

1216 Ensure the cache is coherent

1217 Mark the PTE writable and dirty as it has been written to

1218 Add the page to the LRU list so it may be reclaimed by the swapper later

1219 Mark the page accessed which ensures the page is marked hot and on the top of the active list

1222 Fix the PTE in the page tables for this process

1225 Update the MMU cache if the architecture needs it

1226 Free the page table lock

1227 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) {
1136                   int retval;
1137                   spin_lock(&mm->page_table_lock);
1138                   retval = pte_same(*page_table, orig_pte) ? -1 :
1;
1139                   spin_unlock(&mm->page_table_lock);
1140                   return retval;
1141             }
1142 
1143             /* Had to read the page from swap area: Major fault */
1144             ret = 2;
1145       }
```

If the page did not exist in the swap cache, then read it from backing storage with `swapin_readhead()` which reads in the requested pages and a number of pages after it. Once it completes, `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 `swapin_readahead()` reads in the requested page and a number of pages after it. The number of pages read in is determined by the `page_cluster` variable in `mm/swap.c` which is initialised to 2 on machines with less than 16MiB of memory and 3 otherwise. $2^{page\_cluster}$ 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 `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 `mm_struct` and return the status

1144 The disk had to be accessed to mark that this is a major page fault

```
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);
```

```
1168
1169         mm->rss++;
1170         pte = mk_pte(page, vma->vm_page_prot);
1171         if (write_access && can_share_swap_page(page))
1172               pte = pte_mkdirty(pte_mkwrite(pte));
1173         unlock_page(page);
1174
1175         flush_page_to_ram(page);
1176         flush_icache_page(vma, page);
1177         set_pte(page_table, pte);
1178
1179         /* No need to invalidate - it was non-present before */
1180         update_mmu_cache(vma, address, pte);
1181         spin_unlock(&mm->page_table_lock);
1182         return ret;
1183 }
```

Place the page in the process page tables

`1147` Mark the page as active so it will be moved to the top of the active LRU list

`1149` Lock the page which has the side effect of waiting for the IO swapping in the page to complete

`1155-1161` 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

`1165` The function `swap_free()` reduces the reference to a swap entry. If it drops to 0, it is actually freed

`1166-1167` 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

`1169` The page is now going to be used so increment the `mm_struct`'s RSS count

`1170` Make a PTE for this page

`1171` 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

`1173` Unlock the page

`1175` 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

`1176` `flush_icache_page()` is similar in principle except it ensures the icache and dcache's are coherent

`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

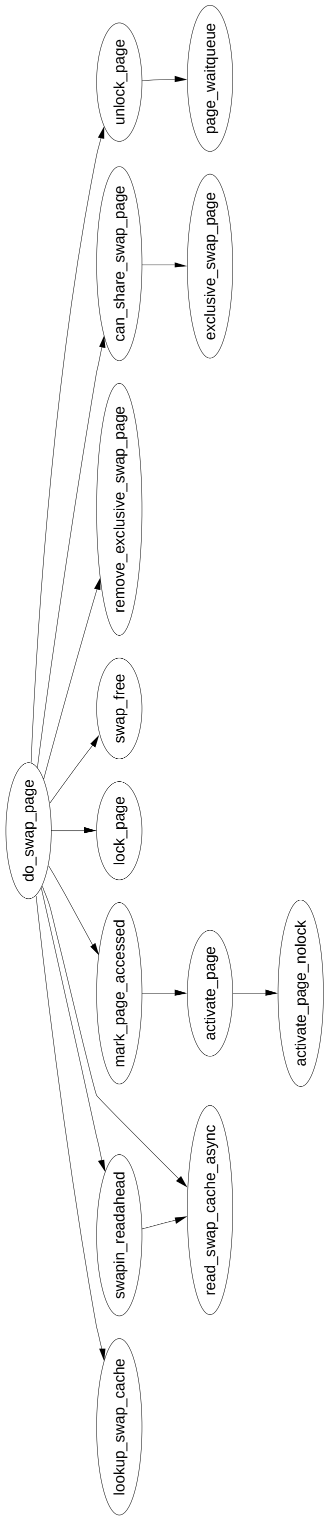

Figure 5.10: do_swap_page

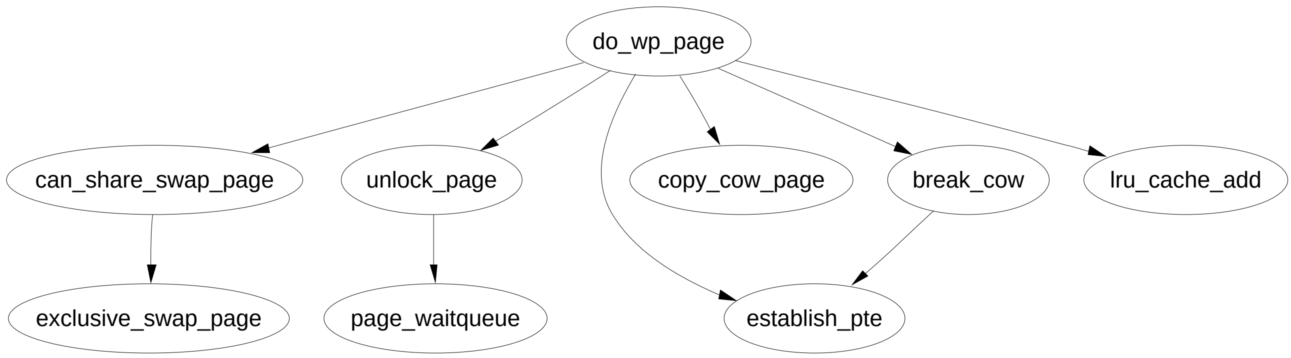

Figure 5.11: do_wp_page

Chapter 6

High Memory Management

6.1 Mapping High Memory Pages

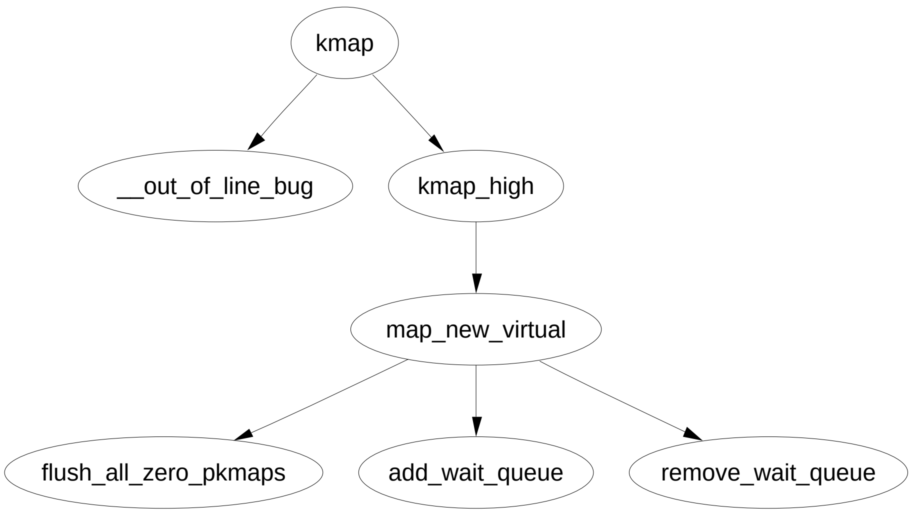

Figure 6.1: Call Graph: `kmap()`

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*)

```
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 avaialble and mapping the page.

```
static inline unsigned long map_new_virtual(struct page *page)
{
        unsigned long vaddr;
        int count;

start:
        count = LAST_PKMAP;
```

```
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
```

86 Start scanning at the last possible slot

88-119 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

89 `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

90-93 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

94-95 If this element is 0, a usable slot has been found for the page

96-96 Move to the next index to scan

```
102                 {
103                         DECLARE_WAITQUEUE(wait, current);
104
105                         current->state = TASK_UNINTERRUPTIBLE;
106                         add_wait_queue(&pkmap_map_wait, &wait);
107                         spin_unlock(&kmap_lock);
108                         schedule();
109                         remove_wait_queue(&pkmap_map_wait, &wait);
110                         spin_lock(&kmap_lock);
111
112                         /* Somebody else might have mapped it while we
                               slept */
113                         if (page->virtual)
114                                 return (unsigned long) page->virtual;
115
116                         /* Re-start */
117                         goto start;
118                 }
119         }
```

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;
127 }
```

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.

```
42 static void flush_all_zero_pkmaps(void)
43 {
44         int i;
45
46         flush_cache_all();
47
48         for (i = 0; i < LAST_PKMAP; i++) {
49                 struct page *page;
50
57                 if (pkmap_count[i] != 1)
58                         continue;
59                 pkmap_count[i] = 0;
60
61                 /* sanity check */
62                 if (pte_none(pkmap_page_table[i]))
63                         BUG();
64
72                 page = pte_page(pkmap_page_table[i]);
73                 pte_clear(&pkmap_page_table[i]);
74
75                 page->virtual = NULL;
76         }
77         flush_tlb_all();
78 }
```

46 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*)

```
 71 static inline void kunmap(struct page *page)
 72 {
 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*)

```
150 void kunmap_high(struct page *page)
151 {
152         unsigned long vaddr;
153         unsigned long nr;
154         int need_wakeup;
155 
156         spin_lock(&kmap_lock);
157         vaddr = (unsigned long) page->virtual;
158         if (!vaddr)
159                 BUG();
160         nr = PKMAP_NR(vaddr);
161 
166         need_wakeup = 0;
167         switch (--pkmap_count[nr]) {
168         case 0:
169                 BUG();
170         case 1:
181                 need_wakeup = waitqueue_active(&pkmap_map_wait);
182         }
183         spin_unlock(&kmap_lock);
184 
185         /* do wake-up, if needed, race-free outside of the spin lock */
186         if (need_wakeup)
187                 wake_up(&pkmap_map_wait);
188 }
```

156 Acquire `kmap_lock` protecting the `virtual()` field and the `pkmap_count` array

157 Get the virtual page

158-159 If the virtual field is not set, it is a double unmapping or unmapping of a non-mapped page so `BUG()`

160 Get the index within the `pkmap_count` array

166 By default, a wakeup call to processes calling `kmap()` is not needed

167 Check the value of the index after decrement

168-169 Falling to 0 is a bug as the TLB needs to be flushed to make 0 a valid entry

170-181 If it has dropped to 1 (free entry but needs TLB flush), check to see if there is anyone sleeping on the `pkmap_map_wait` queue. If necessary, the queue will be woken up after the spinlock is freed

183 Free `kmap_lock`

186-187 If there is waiters on the queue and a slot has been freed, wake them up

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.

```
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.

```
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);
```

```
 93
 94         idx = type + KM_TYPE_NR*smp_processor_id();
 95         vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
 96 #if HIGHMEM_DEBUG
 97         if (!pte_none(*(kmap_pte-idx)))
 98                 out_of_line_bug();
 99 #endif
100         set_pte(kmap_pte-idx, mk_pte(page, kmap_prot));
101         __flush_tlb_one(vaddr);
102
103         return (void*) vaddr;
104 }
```

86 The parameters are the page to map and the type of usage required. One slot per usage per processor is maintained

91-92 If the page is in low memory, return a direct mapping

94 `type` gives which slot to use. `KM_TYPE_NR * smp_processor_id()` gives the set of slots reserved for this processor

95 Get the virtual address

97-98 Debugging code. In reality a PTE will always exist

100 Set the PTE into the reserved slot

101 Flush the TLB for this slot

103 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 replce 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();
```

```
117
118         /*
119          * force other mappings to Oops if they'll try to access
120          * this pte without first remap it
121          */
122         pte_clear(kmap_pte-idx);
123         __flush_tlb_one(vaddr);
124 #endif
125 }
```

109 Get the virtual address

112-113 If the address supplied is not in the fixed area, return

115-116 If the address does not correspond to the reserved slot for this type of usage and processor, declare it

122-123 Unmap the page now so that if it is referenced again, it will cause an Oops

6.3 Bounce Buffers

Function: create_buffers (*mm/highmem.c*)

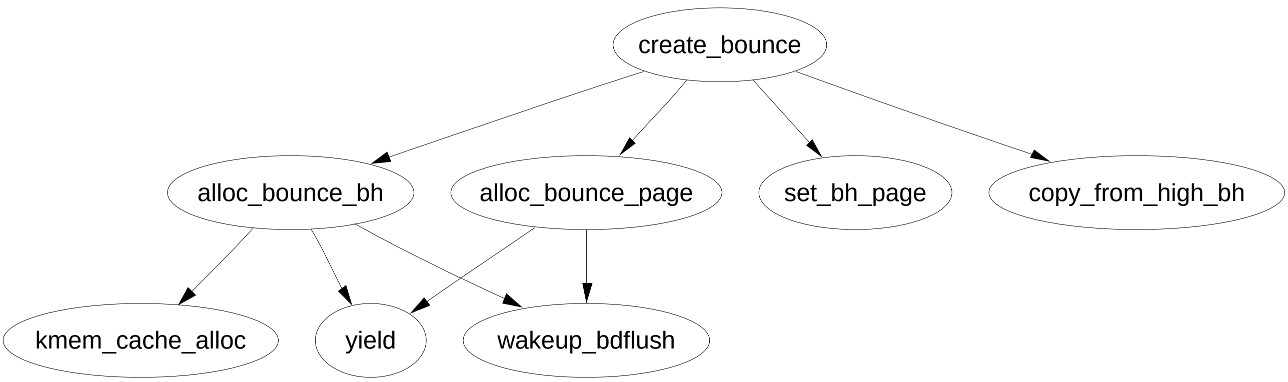

Figure 6.2: Call Graph: create_bounce

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.

```
struct buffer_head * create_bounce(int rw, struct buffer_head * bh_orig)
{
        struct page *page;
        struct buffer_head *bh;

        if (!PageHighMem(bh_orig->b_page))
                return bh_orig;
```

```
405
406         bh = alloc_bounce_bh();
413         page = alloc_bounce_page();
414
415         set_bh_page(bh, page, 0);
416
```

398 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

403-404 If the template buffer head is already in low memory, simply return it

406 Allocate a buffer head from the slab allocator or from the emergency pool if it fails

413 Allocate a page from the buddy allocator or the emergency pool if it fails

415 Associate the allocated page with the allocated `buffer_head`

```
417         bh->b_next = NULL;
418         bh->b_blocknr = bh_orig->b_blocknr;
419         bh->b_size = bh_orig->b_size;
420         bh->b_list = -1;
421         bh->b_dev = bh_orig->b_dev;
422         bh->b_count = bh_orig->b_count;
423         bh->b_rdev = bh_orig->b_rdev;
424         bh->b_state = bh_orig->b_state;
425 #ifdef HIGHMEM_DEBUG
426         bh->b_flushtime = jiffies;
427         bh->b_next_free = NULL;
428         bh->b_prev_free = NULL;
429         /* bh->b_this_page */
430         bh->b_reqnext = NULL;
431         bh->b_pprev = NULL;
432 #endif
433         /* bh->b_page */
434         if (rw == WRITE) {
435                 bh->b_end_io = bounce_end_io_write;
436                 copy_from_high_bh(bh, bh_orig);
437         } else
438                 bh->b_end_io = bounce_end_io_read;
439         bh->b_private = (void *)bh_orig;
440         bh->b_rsector = bh_orig->b_rsector;
441 #ifdef HIGHMEM_DEBUG
442         memset(&bh->b_wait, -1, sizeof(bh->b_wait));
443 #endif
```

```
444
445         return bh;
446 }
```

Populate the newly created `buffer_head`

- **424** Copy in information essentially verbatim except for the `b_list` field as this buffer is not directly connected to the others on the list
- **426-431** Debugging only information
- **434-437** 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()`
- **437-438** If we are waiting for a device to write data into the buffer, then the callback function `bounce_end_io_read()` is used
- **439-440** Copy the remaining information from the template `buffer_head`
- **445** 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.

```
362 struct buffer_head *alloc_bounce_bh (void)
363 {
364         struct list_head *tmp;
365         struct buffer_head *bh;
366
367         bh = kmem_cache_alloc(bh_cachep, SLAB_NOHIGHIO);
368         if (bh)
369                 return bh;
373
374         wakeup_bdflush();
```

- **367** 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
- **368-369** If the allocation was successful, return
- **374** If it was not, wake up **bdflush** to launder pages

```
375
376 repeat_alloc:
380         tmp = &emergency_bhs;
381         spin_lock_irq(&emergency_lock);
```

```
382         if (!list_empty(tmp)) {
383                 bh = list_entry(tmp->next, struct buffer_head,
                                    b_inode_buffers);
384                 list_del(tmp->next);
385                 nr_emergency_bhs--;
386         }
387         spin_unlock_irq(&emergency_lock);
388         if (bh)
389                 return bh;
390 
391         /* we need to wait I/O completion */
392         run_task_queue(&tq_disk);
393 
394         yield();
395         goto repeat_alloc;
396 }
```

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.

```
struct page *alloc_bounce_page (void)
{
        struct list_head *tmp;
        struct page *page;

        page = alloc_page(GFP_NOHIGHIO);
        if (page)
```

```
333                 return page;
337
338         wakeup_bdflush();
```

331-333 Allocate from the buddy allocator and return the page if successful

338 Wake bdflush to launder pages

```
339
340 repeat_alloc:
344         tmp = &emergency_pages;
345         spin_lock_irq(&emergency_lock);
346         if (!list_empty(tmp)) {
347                 page = list_entry(tmp->next, struct page, list);
348                 list_del(tmp->next);
349                 nr_emergency_pages--;
350         }
351         spin_unlock_irq(&emergency_lock);
352         if (page)
353                 return page;
354
355         /* we need to wait I/O completion */
356         run_task_queue(&tq_disk);
357
358         yield();
359         goto repeat_alloc;
360 }
```

344 Get the end of the emergency buffer head list

334 Acquire the lock protecting the pools

346-350 If the pool is not empty, take a page from the list and decrement the number of available `nr_emergency_pages`

351 Release the lock

352-353 If the allocation was successful, return it

356 Run the IO task queue to try and replenish the emergency pool

394 Yield the processor

395 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

```
static void bounce_end_io_write (struct buffer_head *bh, int uptodate)
{
        bounce_end_io(bh, uptodate);
}
```

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

```
317 static void bounce_end_io_read (struct buffer_head *bh, int uptodate)
318 {
319         struct buffer_head *bh_orig =
                            (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.

```
static inline void copy_from_high_bh (struct buffer_head *to,
                            struct buffer_head *from)
{
        struct page *p_from;
        char *vfrom;

        p_from = from->b_page;

        vfrom = kmap_atomic(p_from, KM_USER0);
        memcpy(to->b_data, vfrom + bh_offset(from), to->b_size);
        kunmap_atomic(vfrom, KM_USER0);
}
```

216 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()`

217 Copy the data

218 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

```
221 static inline void copy_to_high_bh_irq (struct buffer_head *to,
222                          struct buffer_head *from)
223 {
224         struct page *p_to;
225         char *vto;
226         unsigned long flags;
227 
228         p_to = to->b_page;
229         __save_flags(flags);
230         __cli();
231         vto = kmap_atomic(p_to, KM_BOUNCE_READ);
232         memcpy(vto + bh_offset(to), from->b_data, to->b_size);
233         kunmap_atomic(vto, KM_BOUNCE_READ);
234         __restore_flags(flags);
235 }
```

229-230 Save the flags and disable interrupts

231 Map the high memory page into low memory

232 Copy the data

233 Unmap the page

234 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.

```
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;

```

```
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`

```
275 static __init int init_emergency_pool(void)
276 {
277         struct sysinfo i;
278         si_meminfo(&i);
279         si_swapinfo(&i);
280 
281         if (!i.totalhigh)
282                 return 0;
283 
284         spin_lock_irq(&emergency_lock);
285         while (nr_emergency_pages < POOL_SIZE) {
286                 struct page * page = alloc_page(GFP_ATOMIC);
287                 if (!page) {
288                         printk("couldn't refill highmem emergency pages");
289                         break;
290                 }
291                 list_add(&page->list, &emergency_pages);
292                 nr_emergency_pages++;
293         }
```

281-282 If there is no high memory available, do not bother

284 Acquire the lock protecting the pools

285-293 Allocate `POOL_SIZE` pages from the buddy allocator and add them to a linked list. Keep a count of the number of pages in the pool with `nr_emergency_pages`

```
294         while (nr_emergency_bhs < POOL_SIZE) {
295                 struct buffer_head * bh =
                        kmem_cache_alloc(bh_cachep, SLAB_ATOMIC);
296                 if (!bh) {
297                         printk("couldn't refill highmem emergency bhs");
298                         break;
299                 }
300                 list_add(&bh->b_inode_buffers, &emergency_bhs);
301                 nr_emergency_bhs++;
302         }
303         spin_unlock_irq(&emergency_lock);
304         printk("allocated %d pages and %d bhs reserved for the
                highmem bounces\n",
```

```
305                 nr_emergency_pages, nr_emergency_bhs);
306
307         return 0;
308 }
```

294-302 Allocate POOL_SIZE `buffer_head`s 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
725         daemonize();
726         strcpy(tsk->comm, "kswapd");
727         sigfillset(&tsk->blocked);
728
741         tsk->flags |= PF_MEMALLOC;
742
746         for (;;) {
747                 __set_current_state(TASK_INTERRUPTIBLE);
```

```
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.

```
695 static int kswapd_can_sleep(void)
696 {
697         pg_data_t * pgdat;
698
699         for_each_pgdat(pgdat) {
700                 if (!kswapd_can_sleep_pgdat(pgdat))
701                         return 0;
702         }
703
704         return 1;
705 }
```

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.

```
680 static int kswapd_can_sleep_pgdat(pg_data_t * pgdat)
681 {
682         zone_t * zone;
683         int i;
684
685         for (i = pgdat->nr_zones-1; i >= 0; i--) {
686                 zone = pgdat->node_zones + i;
687                 if (!zone->need_balance)
688                         continue;
689                 return 0;
690         }
691
692         return 1;
693 }
```

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 can not 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

```
667 static void kswapd_balance(void)
668 {
669         int need_more_balance;
670         pg_data_t * pgdat;
671
672         do {
673                 need_more_balance = 0;
674
675                 for_each_pgdat(pgdat)
676                         need_more_balance |= kswapd_balance_pgdat(pgdat);
677         } while (need_more_balance);
678 }
```

672-677 Continuously cycle through each pgdat

675 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*)

```
641 static int kswapd_balance_pgdat(pg_data_t * pgdat)
642 {
643         int need_more_balance = 0, i;
644         zone_t * zone;
645
646         for (i = pgdat->nr_zones-1; i >= 0; i--) {
647                 zone = pgdat->node_zones + i;
648                 if (unlikely(current->need_resched))
649                         schedule();
650                 if (!zone->need_balance)
651                         continue;
652                 if (!try_to_free_pages_zone(zone, GFP_KSWAPD)) {
653                         zone->need_balance = 0;
654                         __set_current_state(TASK_INTERRUPTIBLE);
655                         schedule_timeout(HZ);
656                         continue;
657                 }
658                 if (check_classzone_need_balance(zone))
659                         need_more_balance = 1;
660                 else
661                         zone->need_balance = 0;
662         }
663
664         return need_more_balance;
665 }
```

646-662 Cycle through each zone and call `try_to_free_pages_zone()` if it needs re-balancing

647 node_zones is an array and i is an index within it

648-649 Call `schedule()` if the quanta is expired to prevent kswapd hogging the CPU

650-651 If the zone does not require balance, move to the next one

652-657 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

658-661 If is was successful, `check_classzone_need_balance()` is called to see if the zone requires further balancing or not

664 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 }
```

60 If the page is not already part of the LRU lists, add it

61 Acquire the LRU lock

62-63 Test and set the LRU bit. If it was clear then call `add_page_to_inactive_list()`

64 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)          \
do {                                           \
        DEBUG_LRU_PAGE(page);                  \
        SetPageActive(page);                   \
        list_add(&(page)->lru, &active_list);  \
        nr_active_pages++;                     \
} while (0)
```

181 The `DEBUG_LRU_PAGE()` macro will call `BUG()` if the page is already on the LRU list or is marked been active

182 Update the flags of the page to show it is active

183 Add the page to the `active_list`

184 Update the count of the number of pages in the `active_list`

Function: add_page_to_inactive_list (*include/linux/swap.h*)
Adds the page to the `inactive_list`

```
187 #define add_page_to_inactive_list(page)       \
188 do {                                          \
189       DEBUG_LRU_PAGE(page);                   \
190       list_add(&(page)->lru, &inactive_list); \
191       nr_inactive_pages++;                    \
192 } while (0)
```

189 The `DEBUG_LRU_PAGE()` macro will call `BUG()` if the page is already on the LRU list or is marked been active

190 Add the page to the `inactive_list`

191 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()`.

```
 90 void lru_cache_del(struct page * page)
 91 {
 92       spin_lock(&pagemap_lru_lock);
 93       __lru_cache_del(page);
 94       spin_unlock(&pagemap_lru_lock);
 95 }
```

92 Acquire the LRU lock

93 `__lru_cache_del()` does the “real” work of removing the page from the LRU lists

94 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`.

```
1316 void mark_page_accessed(struct page *page)
1317 {
1318         if (!PageActive(page) && PageReferenced(page)) {
1319                 activate_page(page);
1320                 ClearPageReferenced(page);
1321         } else
1322                 SetPageReferenced(page);
1323 }
```

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`

1322 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`.

```
47 void activate_page(struct page * page)
48 {
49         spin_lock(&pagemap_lru_lock);
50         activate_page_nolock(page);
51         spin_unlock(&pagemap_lru_lock);
52 }
```

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`

```
39 static inline void activate_page_nolock(struct page * page)
40 {
41         if (PageLRU(page) && !PageActive(page)) {
42                 del_page_from_inactive_list(page);
43                 add_page_to_active_list(page);
44         }
45 }
```

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 |
```

```
                             1 << PG_error | 1 << PG_dirty |
                             1 << PG_referenced | 1 << PG_arch_1 |
                             1 << PG_checked);
658         page->flags = flags | (1 << PG_locked);
659         page_cache_get(page);
660         page->index = offset;
661         add_page_to_inode_queue(mapping, page);
662         add_page_to_hash_queue(page, hash);
663 }
```

657 Clear all page flags

658 Lock the page

659 Take a reference to the page in case it gets freed prematurely

660 Update the index so it is known what file offset this page represents

661 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

662 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 lineraly search the inode queue

7.3 Shrinking all caches

Function: shrink_caches (*mm/vmscan.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
565         nr_pages -= kmem_cache_reap(gfp_mask);
566         if (nr_pages <= 0)
567               return 0;
568
569         nr_pages = chunk_size;
570         /* try to keep the active list 2/3 of the size of the cache */
571         ratio = (unsigned long) nr_pages *
                  nr_active_pages / ((nr_inactive_pages + 1) * 2);
572         refill_inactive(ratio);
573
574         nr_pages = shrink_cache(nr_pages, classzone, gfp_mask, priority);
```

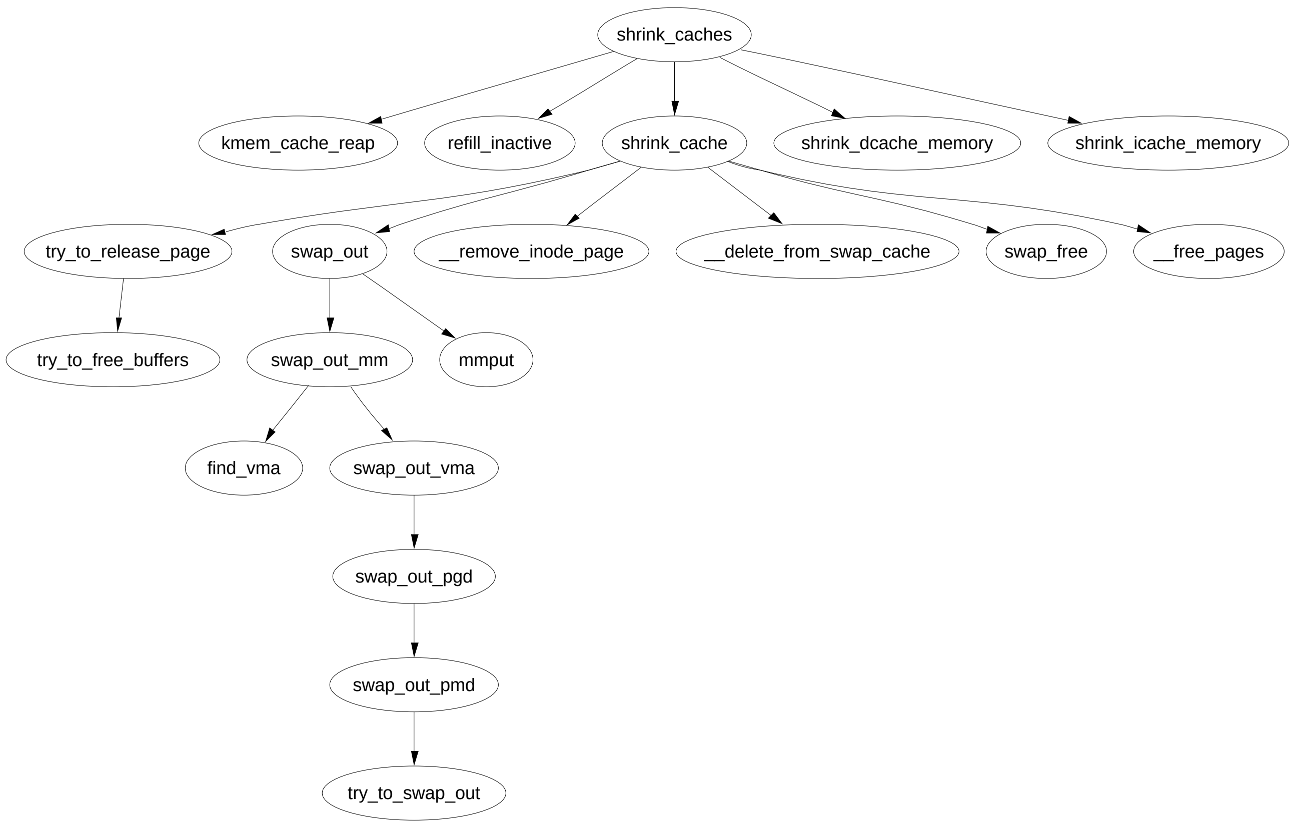

Figure 7.1: shrink_cache

```
575         if (nr_pages <= 0)
576                 return 0;
577
578         shrink_dcache_memory(priority, gfp_mask);
579         shrink_icache_memory(priority, gfp_mask);
580 #ifdef CONFIG_QUOTA
581         shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
582 #endif
583
584         return nr_pages;
585 }
```

560 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

565-567 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

571-572 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

574-575 Shrink the page cache, if enough pages are freed, return

578-582 Shrink the dcache, icache and dqcache. These are small objects in themselves but the cascading effect frees up a lot of disk buffers

584 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 is uses, there is no point balancing other zones.

```
int try_to_free_pages(unsigned int gfp_mask)
{
```

```
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.

```
int try_to_free_pages_zone(zone_t *classzone, unsigned int gfp_mask)
{
      int priority = DEF_PRIORITY;
      int nr_pages = SWAP_CLUSTER_MAX;

      gfp_mask = pf_gfp_mask(gfp_mask);
      do {
            nr_pages = shrink_caches(classzone, priority,
                                  gfp_mask, nr_pages);
            if (nr_pages <= 0)
                  return 1;
      } while (--priority);

      /*
```

```
600          * Hmm.. Cache shrink failed - time to kill something?
601          * Mhwahahhaha! This is the part I really like. Giggle.
602          */
603         out_of_memory();
604         return 0;
605 }
```

589 Start with the lowest priority. Statically defined to be 6

590 Try and free SWAP_CLUSTER_MAX pages. Statically defined to be 32

592 `pf_gfp_mask()` checks the PF_NOIO flag in the current process flags. If no IO can be performed, it ensures there is no incompatible flags in the GFP mask

593-597 Starting with the lowest priority and increasing with each pass, call `shrink_caches()` until nr_pages has been freed

595-596 If enough pages were freed, return indicating that the work is complete

603 If enough pages could not be freed even at highest priority (where at worst the full `inactive_list` is scanned) then check to see if we are out of memory. If we are, then a process will be selected to be killed

604 Return indicating that we failed to free enough pages

7.4 Refilling inactive_list

Function: refill_inactive (*mm/vmscan.c*)
Move nr_pages from the `active_list` to the `inactive_list`

```
static void refill_inactive(int nr_pages)
{
        struct list_head * entry;

        spin_lock(&pagemap_lru_lock);
        entry = active_list.prev;
        while (nr_pages && entry != &active_list) {
                struct page * page;

                page = list_entry(entry, struct page, lru);
                entry = entry->prev;
                if (PageTestandClearReferenced(page)) {
                        list_del(&page->lru);
                        list_add(&page->lru, &active_list);
                        continue;
                }

```

```
550             nr_pages--;
551 
552             del_page_from_active_list(page);
553             add_page_to_inactive_list(page);
554             SetPageReferenced(page);
555         }
556         spin_unlock(&pagemap_lru_lock);
557 }
```

537 Acquire the lock protecting the LRU list

538 Take the last entry in the `active_list`

539-555 Move nr_pages or until the `active_list` is empty

542 Get the struct page for this entry

544-548 Test and clear the referenced flag. If it has been referenced, then it is moved back to the top of the `active_list`

550-553 Move one page from the `active_list` to the `inactive_list`

554 Mark it referenced so that if it is referenced again soon, it will be promoted back to the `active_list` without requiring a second reference

556 Release the lock protecting the LRU list

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 {
340         struct list_head * entry;
341         int max_scan = nr_inactive_pages / priority;
342         int max_mapped = min((nr_pages << (10 - priority)),
                                max_scan / 10);
343 
344         spin_lock(&pagemap_lru_lock);
345         while (--max_scan >= 0 &&
                   (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 `gfp_mask` The gfp mask determining what actions may be taken

`priority` The priority of the function, starts at DEF_PRIORITY (6) and decreases to the highest priority of 1

341 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

342 The maximum amount of process mapped pages allowed is either one tenth of the max_scan value or $nr_pages * 2^{10-priority}$. If this number of pages are found, whole processes will be swapped out

344 Lock the LRU list

345 Keep scanning until `max_scan` pages have been scanned or the `inactive_list` is empty

```
346                 struct page * page;
347 
348                 if (unlikely(current->need_resched)) {
349                         spin_unlock(&pagemap_lru_lock);
350                         __set_current_state(TASK_RUNNING);
351                         schedule();
352                         spin_lock(&pagemap_lru_lock);
353                         continue;
354                 }
355 
```

348-354 Reschedule if the quanta has been used up

349 Free the LRU lock as we are about to sleep

350 Show we are still running

351 Call `schedule()` so another process can be context switched in

352 Re-acquire the LRU lock

353 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

```
356                 page = list_entry(entry, struct page, lru);
357 
358                 BUG_ON(!PageLRU(page));
359                 BUG_ON(PageActive(page));
360 
361                 list_del(entry);
362                 list_add(entry, &inactive_list);
363 
364                 /*
```

```
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 &&
                   (page_count(page) != 1 || !page->mapping))
376                 goto page_mapped;
377 
3
```

356 Get the struct page for this entry in the LRU

358-359 It is a bug if the page either belongs to the `active_list` or is currently marked as active

361-362 Move the page to the top of the `inactive_list` so that if the page is skipped, it will not be simply examined a second time

368-369 If the page count has already reached 0, skip over it. This is possible if another process has just unlinked the page and is waiting for something like IO to complete before removing it from the LRU

371-372 Skip over this page if it belongs to a zone we are not currently interested in

375-376 If the page is mapped by a process, then goto page_mapped where the `max_mapped` is decremented and next page examined. If `max_mapped` reaches 0, process pages will be swapped out

```
            if (unlikely(TryLockPage(page))) {
                if (PageLaunder(page) && (gfp_mask & __GFP_FS)) {
                    page_cache_get(page);
                    spin_unlock(&pagemap_lru_lock);
                    wait_on_page(page);
                    page_cache_release(page);
                    spin_lock(&pagemap_lru_lock);
                }
                continue;
            }
```

Page is locked and the launder bit is set. In this case, wait until the IO is complete and then try to free the page

382-383 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...

384 Take a reference to the page so it does not disappear while we sleep

385 Free the LRU lock

386 Wait until the IO is complete

387 Release the reference to the page. If it reaches 0, the page will be freed

388 Re-acquire the LRU lock

390 Move to the next page

```
392
393                 if (PageDirty(page) &&
                      is_page_cache_freeable(page) && page->mapping) {
402                         int (*writepage)(struct page *);
403
404                         writepage = page->mapping->a_ops->writepage;
405                         if ((gfp_mask & __GFP_FS) && writepage) {
406                                 ClearPageDirty(page);
407                                 SetPageLaunder(page);
408                                 page_cache_get(page);
409                                 spin_unlock(&pagemap_lru_lock);
410
411                                 writepage(page);
412                                 page_cache_release(page);
413
414                                 spin_lock(&pagemap_lru_lock);
415                                 continue;
416                         }
417                 }
```

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.

393 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

404 Get a pointer to the necessary `writepage()` function for this mapping or device

405-416 This block of code can only be executed if a `writepage()` function is available and the GFP flags allow file operations

406-407 Clear the dirty bit and mark that the page is being laundered

408 Take a reference to the page so it will not be freed unexpectedly

409 Unlock the LRU list

411 Call the writepage function

412 Release the reference to the page

414-415 Re-acquire the LRU list lock and move to the next page

```
424             if (page->buffers) {
425                     spin_unlock(&pagemap_lru_lock);
426 
427                     /* avoid to free a locked page */
428                     page_cache_get(page);
429 
430                     if (try_to_release_page(page, gfp_mask)) {
431                             if (!page->mapping) {
438                                     spin_lock(&pagemap_lru_lock);
439                                     UnlockPage(page);
440                                     __lru_cache_del(page);
441 
443                                     page_cache_release(page);
444 
445                                     if (--nr_pages)
446                                             continue;
447                                     break;
448                             } else {
454                                     page_cache_release(page);
455 
456                                     spin_lock(&pagemap_lru_lock);
457                             }
458                     } else {
460                             UnlockPage(page);
461                             page_cache_release(page);
462 
463                             spin_lock(&pagemap_lru_lock);
464                             continue;
465                     }
466             }
```

Page has buffers associated with it that must be freed.

425 Release the LRU lock as we may sleep

428 Take a reference to the page

430 Call `try_to_release_page()` which will attempt to release the buffers associated with the page. Returns 1 if it succeeds

431-447 Handle where the release of buffers succeeded

431-448 If the mapping is not filled, it is an anonymous page which must be removed from the page cache

438-440 Take the LRU list lock, unlock the page, delete it from the page cache and free it

445-446 Update `nr_pages` to show a page has been freed and move to the next page

447 If `nr_pages` drops to 0, then exit the loop as the work is completed

449-456 If the page does have an associated mapping then simply drop the reference to the page and re-acquire the LRU lock

459-464 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

```
467
468             spin_lock(&pagecache_lock);
469
473             if (!page->mapping || !is_page_cache_freeable(page)) {
474                     spin_unlock(&pagecache_lock);
475                     UnlockPage(page);
476 page_mapped:
477                     if (--max_mapped >= 0)
478                             continue;
479
484                     spin_unlock(&pagemap_lru_lock);
485                     swap_out(priority, gfp_mask, classzone);
486                     return nr_pages;
487             }
```

468 From this point on, pages in the swap cache are likely to be examined which is protected by the `pagecache_lock` which must be now held

473-487 An anonymous page with no buffers is mapped by a process

474-475 Release the page cache lock and the page

477-478 Decrement `max_mapped`. If it has not reached 0, move to the next page

484-485 Too many mapped pages have been found in the page cache. The LRU lock is released and `swap_out()` is called to begin swapping out whole processes

```
493             if (PageDirty(page)) {
494                     spin_unlock(&pagecache_lock);
495                     UnlockPage(page);
496                     continue;
497             }
```

493-497 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

```
498
499                 /* point of no return */
500                 if (likely(!PageSwapCache(page))) {
501                         __remove_inode_page(page);
502                         spin_unlock(&pagecache_lock);
503                 } else {
504                         swp_entry_t swap;
505                         swap.val = page->index;
506                         __delete_from_swap_cache(page);
507                         spin_unlock(&pagecache_lock);
508                         swap_free(swap);
509                 }
510
511                 __lru_cache_del(page);
512                 UnlockPage(page);
513
514                 /* effectively free the page here */
515                 page_cache_release(page);
516
517                 if (--nr_pages)
518                         continue;
519                 break;
520         }
```

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_page` and move to the next page if it is not 0

519 If it reaches 0, the work of the function is complete

```
521         spin_unlock(&pagemap_lru_lock);
522
523         return nr_pages;
524 }
```

521-524 Function exit. Free the LRU lock and return the number of pages left to free

7.6 Swapping Out Process Pages

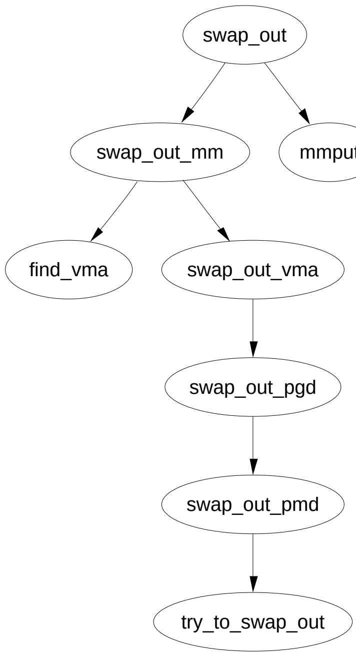

Figure 7.2: Call Graph: swap_out

Function: swap_out (*mm/vmscan.c*)
This function linearaly searches through every processes 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`

```
static int swap_out(unsigned int priority, unsigned int gfp_mask,
              zone_t * classzone)
{
      int counter, nr_pages = SWAP_CLUSTER_MAX;
      struct mm_struct *mm;

      counter = mmlist_nr;
      do {
            if (unlikely(current->need_resched)) {
                  __set_current_state(TASK_RUNNING);
                  schedule();
            }

            spin_lock(&mmlist_lock);
            mm = swap_mm;
            while (mm->swap_address == TASK_SIZE || mm == &init_mm) {
                  mm->swap_address = 0;
                  mm = list_entry(mm->mmlist.next,
```

```
                                  struct mm_struct, mmlist);
313                     if (mm == swap_mm)
314                           goto empty;
315                     swap_mm = mm;
316               }
317
318               /* Make sure the mm doesn't disappear
                     when we drop the lock.. */
319               atomic_inc(&mm->mm_users);
320               spin_unlock(&mmlist_lock);
321
322               nr_pages = swap_out_mm(mm, nr_pages, &counter, classzone);
323
324               mmput(mm);
325
326               if (!nr_pages)
327                     return 1;
328         } while (--counter >= 0);
329
330         return 0;
331
332 empty:
333         spin_unlock(&mmlist_lock);
334         return 0;
335 }
```

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

319 Increase the reference count so that the mm does not get freed while we are scanning

320 Release the mm lock

322 Begin scanning the mm with `swap_out_mm()`

324 Drop the reference to the mm

326-327 If the required number of pages has been freed, return success

328 If we failed on this pass, increase the priority so more processes will be scanned

330 Return failure

Function: swap_out_mm (*mm/vmscan.c*)

Walk through each VMA and call `swap_out_mm()` for each one.

```
static inline int swap_out_mm(struct mm_struct * mm, int count,
                      int * mmcounter, zone_t * classzone)
{
        unsigned long address;
        struct vm_area_struct* vma;

        spin_lock(&mm->page_table_lock);
        address = mm->swap_address;
        if (address == TASK_SIZE || swap_mm != mm) {
                /* We raced: don't count this mm but try again */
                ++*mmcounter;
                goto out_unlock;
        }
        vma = find_vma(mm, address);
        if (vma) {
                if (address < vma->vm_start)
                        address = vma->vm_start;

                for (;;) {
                        count = swap_out_vma(mm, vma, address,
                                          count, classzone);
                        vma = vma->vm_next;
                        if (!vma)
                                break;
                        if (!count)
                                goto out_unlock;
                        address = vma->vm_start;
                }
        }
        /* Indicate that we reached the end of address space */
        mm->swap_address = TASK_SIZE;
```

```
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()`.

```
static inline int swap_out_vma(struct mm_struct * mm,
                               struct vm_area_struct * vma,
                               unsigned long address, int count,
                               zone_t * classzone)
{
        pgd_t *pgdir;
        unsigned long end;

        /* Don't swap out areas which are reserved */
        if (vma->vm_flags & VM_RESERVED)
                return count;

        pgdir = pgd_offset(mm, address);

        end = vma->vm_end;
        BUG_ON(address >= end);
        do {
                count = swap_out_pgd(mm, vma, pgdir,
                                     address, end, count, classzone);
                if (!count)
```

```
243                 break;
244             address = (address + PGDIR_SIZE) & PGDIR_MASK;
245             pgdir++;
246         } while (address && (address < end));
247         return count;
248 }
```

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()`

```
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)
{
        pmd_t * pmd;
        unsigned long pgd_end;

        if (pgd_none(*dir))
                return count;
        if (pgd_bad(*dir)) {
                pgd_ERROR(*dir);
                pgd_clear(dir);
                return count;
        }

        pmd = pmd_offset(dir, address);

        pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
        if (pgd_end && (end > pgd_end))
                end = pgd_end;
```

```
215
216         do {
217                 count = swap_out_pmd(mm, vma, pmd, address, end, count,
classzone);
218                 if (!count)
219                         break;
220                 address = (address + PMD_SIZE) & PMD_MASK;
221                 pmd++;
222         } while (address && (address < end));
223         return count;
224 }
```

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.

```
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)
{
        pte_t * pte;
        unsigned long pmd_end;

        if (pmd_none(*dir))
                return count;
        if (pmd_bad(*dir)) {
                pmd_ERROR(*dir);
                pmd_clear(dir);
                return count;
        }

        pte = pte_offset(dir, address);
```

```
172
173         pmd_end = (address + PMD_SIZE) & PMD_MASK;
174         if (end > pmd_end)
175               end = pmd_end;
176
177         do {
178               if (pte_present(*pte)) {
179                     struct page *page = pte_page(*pte);
180
181                     if (VALID_PAGE(page) && !PageReserved(page)) {
182                           count -= try_to_swap_out(mm, vma,
                                                 address, pte,
                                                 page, classzone);
183                           if (!count) {
184                                 address += PAGE_SIZE;
185                                 break;
186                           }
187                     }
188               }
189               address += PAGE_SIZE;
190               pte++;
191         } while (address && (address < end));
192         mm->swap_address = address;
193         return count;
194 }
```

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

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 is been added to the swap cache

```
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)
48 {
49         pte_t pte;
50         swp_entry_t entry;
51 
52         /* Don't look at this pte if it's been accessed recently. */
53         if ((vma->vm_flags & VM_LOCKED) ||
              ptep_test_and_clear_young(page_table)) {
54                 mark_page_accessed(page);
55                 return 0;
56         }
57 
58         /* Don't bother unmapping pages that are active */
59         if (PageActive(page))
60                 return 0;
61 
62         /* Don't bother replenishing zones not under pressure.. */
63         if (!memclass(page_zone(page), classzone))
64                 return 0;
65 
66         if (TryLockPage(page))
67                 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

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          flush_cache_page(vma, address);
75          pte = ptep_get_and_clear(page_table);
76          flush_tlb_page(vma, address);
77
78          if (pte_dirty(pte))
79                  set_page_dirty(page);
80
```

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)) {
87                  entry.val = page->index;
88                  swap_duplicate(entry);
89  set_swap_pte:
90                  set_pte(page_table, swp_entry_to_pte(entry));
91  drop_pte:
92                  mm->rss--;
93                  UnlockPage(page);
94                  {
95                          int freeable =
                               page_count(page) - !!page->buffers <= 2;
96                          page_cache_release(page);
97                          return freeable;
98                  }
99          }
```

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

95 The page is free-able if the count is currently 2 or less and has no buffers

96 Decrement the reference count and free the page if it reaches 0

97 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         }
149
150         /* No swap space left */
151 preserve:
152         set_pte(page_table, pte);
```

```
153         UnlockPage(page);
154         return 0;
155 }
```

134 Allocate a swap entry for this page

135-136 If one could not be allocated, break out where the PTE and page will be re-attached to the process page tables

141 Add the page to the swap cache

142 Mark the page as up to date in memory

143 Mark the page dirty so that it will be written out to swap soon

144 Goto set_swap_pte which will update the PTE with information needed to get the page from swap later

147 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

152 Reattach the PTE to the page tables

153 Unlock the page

154 Return that no page was freed

Chapter 8

Swap Management

8.1 Describing the Swap Area

8.2 Scanning for free entries

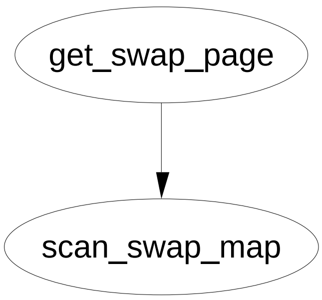

Figure 8.1: Call Graph: `get_swap_page()`

Function: get_swap_page (*mm/swapfile.c*)
This is the high level API function for getting a slot in swap space.

```
swp_entry_t get_swap_page(void)
{
      struct swap_info_struct * p;
      unsigned long offset;
      swp_entry_t entry;
      int type, wrapped = 0;

      entry.val = 0;  /* Out of memory */
      swap_list_lock();
      type = swap_list.next;
      if (type < 0)
            goto out;
      if (nr_swap_pages <= 0)
```

```
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                         }
138                 } else
139                         if (type < 0)
140                                 goto out;       /* out of swap space */
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 wrapaound. 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.

```
static inline int scan_swap_map(struct swap_info_struct *si)
{
        unsigned long offset;
        if (si->cluster_nr) {
                while (si->cluster_next <= si->highest_bit) {
                        offset = si->cluster_next++;
                        if (si->swap_map[offset])
                                continue;
                        si->cluster_nr--;
                        goto got_page;
                }
        }
```

Allocate SWAPFILE_CLUSTER pages sequentially. cluster_nr is initialised to SWAPFILE_CLUTER and decrements with each allocation

47 If cluster_nr is still postive, allocate the next available sequential slot

48 While the current offset to use (cluster_next) is less then the highest known free slot (highest_bit) then ...

49 Record the offset and update 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 cluster_nr field

53 Goto the out path

```
56         si->cluster_nr = SWAPFILE_CLUSTER;
57 
58         /* try to find an empty (even not aligned) cluster. */
59         offset = si->lowest_bit;
60  check_next_cluster:
61         if (offset+SWAPFILE_CLUSTER-1 <= si->highest_bit)
62         {
63                 int nr;
64                 for (nr = offset; nr < offset+SWAPFILE_CLUSTER; nr++)
65                         if (si->swap_map[nr])
66                         {
67                                 offset = nr+1;
68                                 goto check_next_cluster;
69                         }
70                 /* We found a completly empty cluster, so start
71                  * using it.
72                  */
73                 goto got_page;
74         }
```

At this stage, SWAPFILE_CLUSTER pages have been allocated sequentially so find the next free block of SWAPFILE_CLUSTER pages.

56 Re-initialise the count of sequential pages to allocate to 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 offset to offset + SWAPFILE_CLUSTER

65-69 If this slot is used, then start searching again for a free slot beginning after this known alloated one

73 A large cluster was found so use it

```
75         /* No luck, so now go finegrined as usual. -Andrea */
76         for (offset = si->lowest_bit; offset <= si->highest_bit ;
                                                   offset++) {
77                 if (si->swap_map[offset])
78                         continue;
79                 si->lowest_bit = offset+1;
```

This unusual for loop extract starts scanning for a free page starting from `lowest_bit`

77-78 If the slot is in use, move to the next one

79 Update the `lowest_bit` known probable free slot to the succeeding one

```
80         got_page:
81                 if (offset == si->lowest_bit)
82                         si->lowest_bit++;
83                 if (offset == si->highest_bit)
84                         si->highest_bit--;
85                 if (si->lowest_bit > si->highest_bit) {
86                         si->lowest_bit = si->max;
87                         si->highest_bit = 0;
88                 }
89                 si->swap_map[offset] = 1;
90                 nr_swap_pages--;
91                 si->cluster_next = offset+1;
92                 return offset;
93         }
94         si->lowest_bit = si->max;
95         si->highest_bit = 0;
96         return 0;
97 }
```

A slot has been found, do some housekeeping and return it

81-82 If this offset is the known lowest free slot(`lowest_bit`), increment it

83-84 If this offset is the highest known likely free slot, decrement it

85-88 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

89 Set the reference count for the slot

90 Update the accounting for the number of available swap pages (`nr_swap_pages`)

91 Set `cluster_next` to the adjacent slot so the next search will start here

92 Return the free slot

94-96 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.

```
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

86-87 If it is not now in the swap cache, something went seriously wrong

88 Increment the statistic recording the total number of pages in the swap cache

89 Return success

Function: swap_duplicate (*mm/swapfile.c*)
This function verifies a swap entry is valid and if so, increments its swap map count.

```
1143 int swap_duplicate(swp_entry_t entry)
1144 {
1145         struct swap_info_struct * p;
1146         unsigned long offset, type;
1147         int result = 0;
1148
1149         type = SWP_TYPE(entry);
1150         if (type >= nr_swapfiles)
1151                 goto bad_file;
1152         p = type + swap_info;
1153         offset = SWP_OFFSET(entry);
1154
1155         swap_device_lock(p);
1156         if (offset < p->max && p->swap_map[offset]) {
1157                 if (p->swap_map[offset] < SWAP_MAP_MAX - 1) {
1158                         p->swap_map[offset]++;
1159                         result = 1;
1160                 } else if (p->swap_map[offset] <= SWAP_MAP_MAX) {
1161                         if (swap_overflow++ < 5)
1162                                 printk(KERN_WARNING "swap_dup: swap entry
                                                         overflow\n");
1163                         p->swap_map[offset] = SWAP_MAP_MAX;
1164                         result = 1;
1165                 }
1166         }
1167         swap_device_unlock(p);
1168 out:
1169         return result;
1170
1171 bad_file:
1172         printk(KERN_ERR "swap_dup: %s%08lx\n", Bad_file, entry.val);
1173         goto out;
1174 }
```

1143 The parameter is the swap entry to increase the `swap_map` count for

1149-1151 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`

1152-1153 Get the relevant swap_info_struct and get the offset within its swap_map

1155 Lock the swap device

1156 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

1157-1159 If the count is not SWAP_MAP_MAX, simply increment it and return 1 for success

1160-1165 Else the count would overflow so set it to SWAP_MAP_MAX and reserve the slot permanently. In reality this condition is virtually impossible

1167-1169 Unlock the swap device and return

1172-1173 If a bad device was used, print out the error message and return failure

Function: swap_free (*mm/swapfile.c*)

Decrements the corresponding swap_map entry for the swp_entry_t

```
214 void swap_free(swp_entry_t entry)
215 {
216         struct swap_info_struct * p;
217 
218         p = swap_info_get(entry);
219         if (p) {
220                 swap_entry_free(p, SWP_OFFSET(entry));
221                 swap_info_put(p);
222         }
223 }
```

218 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

219-222 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*)

```
192 static int swap_entry_free(struct swap_info_struct *p,
                             unsigned long offset)
193 {
194         int count = p->swap_map[offset];
195 
196         if (count < SWAP_MAP_MAX) {
197                 count--;
198                 p->swap_map[offset] = count;
199                 if (!count) {
```

```
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.

```
static struct swap_info_struct * swap_info_get(swp_entry_t entry)
{
        struct swap_info_struct * p;
        unsigned long offset, type;

        if (!entry.val)
                goto out;
        type = SWP_TYPE(entry);
        if (type >= nr_swapfiles)
                goto bad_nofile;
        p = & swap_info[type];
        if (!(p->flags & SWP_USED))
                goto bad_device;
        offset = SWP_OFFSET(entry);
        if (offset >= p->max)
                goto bad_offset;
        if (!p->swap_map[offset])
```

```
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

Function: swap_info_put (*mm/swapfile.c*)
This function simply unlocks the area and list

```
186 static void swap_info_put(struct swap_info_struct * p)
187 {
188         swap_device_unlock(p);
189         swap_list_unlock();
190 }
```

188 Unlock the device

189 Unlock the swap area list

Function: lookup_swap_cache (*mm/swap_state.c*)
Top level function for finding a page in the swap cache

```
161 struct page * lookup_swap_cache(swp_entry_t entry)
162 {
163         struct page *found;
164 
165         found = find_get_page(&swapper_space, entry.val);
166         /*
167          * Unsafe to assert PageSwapCache and mapping on page found:
168          * if SMP nothing prevents swapoff from deleting this page from
169          * the swap cache at this moment.  find_lock_page would prevent
170          * that, but no need to change: we _have_ got the right page.
171          */
172         INC_CACHE_INFO(find_total);
173         if (found)
174                 INC_CACHE_INFO(find_success);
175         return found;
176 }
```

165 `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

172 Increase the statistic recording the number of times a page was searched for in the cache

173-174 If one was found, increment the successful find count

175 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

```
75 #define find_get_page(mapping, index) \
76         __find_get_page(mapping, index, page_hash(mapping, index))
```

76 `page_hash()` locates an entry in the `page_hash_table` based on the `address_space` and offset

Function: __find_get_page (*mm/filemap.c*)

This function is responsible for finding a struct page given an entry in `page_hash_table` as a starting point.

```
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 }
```

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 `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: __find_page_nolock (*mm/filemap.c*)

This function traverses the hash collision list looking for the page specified by the `address_space` and `offset`.

```
static inline struct page * __find_page_nolock(
                        struct address_space *mapping,
                        unsigned long offset,
                        struct page *page)
{
      goto inside;

      for (;;) {
            page = page->next_hash;
inside:
            if (!page)
                  goto not_found;
```

```
450                 if (page->mapping != mapping)
451                         continue;
452                 if (page->index == offset)
453                         break;
454         }
455
456 not_found:
457         return page;
458 }
```

443 Begin by examining the first page in the list

448-449 If the page is NULL, the right one could not be found so return NULL

450 If the `address_space` does not match, move to the next page on the collision list

452 If the `offset` matchs, return it, else move on

446 Move to the next page on the hash list

457 Return the found page or NULL if not

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 is takes are as follows;

- Find a free `swap_info_struct` in the `swap_info` array an 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;
```

855 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

```
877         for (type = 0 ; type < nr_swapfiles ; type++,p++)
878               if (!(p->flags & SWP_USED))
879                     break;
880         error = -EPERM;
881         if (type >= MAX_SWAPFILES) {
882               swap_list_unlock();
883               goto out;
884         }
885         if (type >= nr_swapfiles)
886               nr_swapfiles = type+1;
887         p->flags = SWP_USED;
888         p->swap_file = NULL;
889         p->swap_vfsmnt = NULL;
890         p->swap_device = 0;
891         p->swap_map = NULL;
892         p->lowest_bit = 0;
893         p->highest_bit = 0;
894         p->cluster_nr = 0;
895         p->sdev_lock = SPIN_LOCK_UNLOCKED;
896         p->next = -1;
897         if (swap_flags & SWAP_FLAG_PREFER) {
898               p->prio =
899                 (swap_flags &
SWAP_FLAG_PRIO_MASK)>>SWAP_FLAG_PRIO_SHIFT;
900         } else {
901               p->prio = --least_priority;
902         }
903         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

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;
912
```

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 Similarily, 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,
                                  BDEV_SWAP);
928               devfs_put_ops(de);/*Decrement module use count now we're
safe*/
929               if (error)
930                     goto bad_swap_2;
```

```
931                 set_blocksize(dev, PAGE_SIZE);
932                 error = -ENODEV;
933                 if (!dev || (blk_size[MAJOR(dev)] &&
934                      !blk_size[MAJOR(dev)][MINOR(dev)]))
935                       goto bad_swap;
936                 swapfilesize = 0;
937                 if (blk_size[MAJOR(dev)])
938                       swapfilesize = blk_size[MAJOR(dev)][MINOR(dev)]
939                             >> (PAGE_SHIFT - 10);
940           } else if (S_ISREG(swap_inode->i_mode))
941                 swapfilesize = swap_inode->i_size >> PAGE_SHIFT;
942           else
943                 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

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 ==
                          q->swap_file->d_inode->i_mapping)
951                    goto bad_swap;
952        }
953
954        swap_header = (void *) __get_free_page(GFP_USER);
955        if (!swap_header) {
956              printk("Unable to start swapping: out of memory :-)\n");
957              error = -ENOMEM;
958              goto bad_swap;
959        }
960
961        lock_page(virt_to_page(swap_header));
962        rw_swap_page_nolock(READ, SWP_ENTRY(type,0),
                         (char *) swap_header);
963
964        if (!memcmp("SWAP-SPACE",swap_header->magic.magic,10))
965              swap_header_version = 1;
966        else if (!memcmp("SWAPSPACE2",swap_header->magic.magic,10))
967              swap_header_version = 2;
968        else {
969              printk("Unable to find swap-space signature\n");
970              error = -EINVAL;
971              goto bad_swap;
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

961 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

962 Read the first page slot in the swap area into `swap_header`

964-672 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`

```
974         switch (swap_header_version) {
975         case 1:
976               memset(((char *) swap_header)+PAGE_SIZE-10,0,10);
977               j = 0;
978               p->lowest_bit = 0;
979               p->highest_bit = 0;
980               for (i = 1 ; i < 8*PAGE_SIZE ; i++) {
981                     if (test_bit(i,(char *) swap_header)) {
982                           if (!p->lowest_bit)
983                                 p->lowest_bit = i;
984                           p->highest_bit = i;
985                           maxpages = i+1;
986                           j++;
987                     }
988               }
989               nr_good_pages = j;
990               p->swap_map = vmalloc(maxpages * sizeof(short));
991               if (!p->swap_map) {
992                     error = -ENOMEM;
993                     goto bad_swap;
994               }
995               for (i = 1 ; i < maxpages ; i++) {
996                     if (test_bit(i,(char *) swap_header))
997                           p->swap_map[i] = 0;
998                     else
999                           p->swap_map[i] = SWAP_MAP_BAD;
1000              }
1001              break;
1002
```

Read in the information needed to populate the `swap_map` when the swap area is version 1.

976 Zero out the magic string identifing the version of the swap area

978-979 Initialise fields in `swap_info_struct` to 0

980-988 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 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

```
        case 2:
              if (swap_header->info.version != 1) {
                    printk(KERN_WARNING
                         "Unable to handle swap header version %d\n",
                         swap_header->info.version);
                    error = -EINVAL;
                    goto bad_swap;
              }

              p->lowest_bit  = 1;
              maxpages = SWP_OFFSET(SWP_ENTRY(0,~0UL)) - 1;
              if (maxpages > swap_header->info.last_page)
                    maxpages = swap_header->info.last_page;
              p->highest_bit = maxpages - 1;

              error = -EINVAL;
              if (swap_header->info.nr_badpages > MAX_SWAP_BADPAGES)
                    goto bad_swap;

              if (!(p->swap_map = vmalloc(maxpages * sizeof(short)))) {
                    error = -ENOMEM;
                    goto bad_swap;
              }

              error = 0;
```

```
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 ||
                        page >= swap_header->info.last_page)
1035                        error = -EINVAL;
1036                  else
1037                        p->swap_map[page] = SWAP_MAP_BAD;
1038            }
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 absolutly 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 occured

```
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;
```

```
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;
```

```
1081         }
1082         swap_device_unlock(p);
1083         swap_list_unlock();
1084         error = 0;
1085         goto out;
```

1070-1080 Insert the new swap area into the correct slot in the swap list based on priority

1082 Unlock the swap device

1083 Unlock the swap list

1084-1085 Return success

```
1086 bad_swap:
1087         if (bdev)
1088                 blkdev_put(bdev, BDEV_SWAP);
1089 bad_swap_2:
1090         swap_list_lock();
1091         swap_map = p->swap_map;
1092         nd.mnt = p->swap_vfsmnt;
1093         nd.dentry = p->swap_file;
1094         p->swap_device = 0;
1095         p->swap_file = NULL;
1096         p->swap_vfsmnt = NULL;
1097         p->swap_map = NULL;
1098         p->flags = 0;
1099         if (!(swap_flags & SWAP_FLAG_PREFER))
1100                 ++least_priority;
1101         swap_list_unlock();
1102         if (swap_map)
1103                 vfree(swap_map);
1104         path_release(&nd);
1105 out:
1106         if (swap_header)
1107                 free_page((long) swap_header);
1108         unlock_kernel();
1109         return error;
1110 }
```

1087-1088 Drop the reference to the block device

1090-1104 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

1104 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

718-719 Acquire information about the special file representing the swap area with `user_path_walk()`. Return on error

```
722        lock_kernel();
723        prev = -1;
724        swap_list_lock();
725        for (type = swap_list.head; type >= 0;
              type = swap_info[type].next) {
726                p = swap_info + type;
727                if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
728                        if (p->swap_file == nd.dentry)
729                          break;
730                }
731                prev = type;
732        }
733        err = -EINVAL;
734        if (type < 0) {
735                swap_list_unlock();
736                goto out_dput;
737        }
738
739        if (prev < 0) {
740                swap_list.head = p->next;
741        } else {
742                swap_info[prev].next = p->next;
743        }
744        if (type == swap_list.next) {
745                /* just pick something that's safe... */
746                swap_list.next = swap_list.head;
747        }
748        nr_swap_pages -= p->pages;
749        total_swap_pages -= p->pages;
750        p->flags = SWP_USED;
```

Acquire the BKL, find the `swap_info_struct` for the area to be deactivated and remove it from the swap list.

722 Acquire the BKL

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

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) {
756              /* re-insert swap space back into swap_list */
757              swap_list_lock();
758              for (prev = -1, i = swap_list.head; i >= 0; prev = i, i =
swap_info[i].next)
759                    if (p->prio >= swap_info[i].prio)
760                          break;
761              p->next = i;
762              if (prev < 0)
763                    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

769-770 Unlock the swap list and return

```
772         if (p->swap_device)
773               blkdev_put(p->swap_file->d_inode->i_bdev, BDEV_SWAP);
774         path_release(&nd);
775 
776         swap_list_lock();
777         swap_device_lock(p);
778         nd.mnt = p->swap_vfsmnt;
779         nd.dentry = p->swap_file;
780         p->swap_vfsmnt = NULL;
781         p->swap_file = NULL;
782         p->swap_device = 0;
783         p->max = 0;
784         swap_map = p->swap_map;
785         p->swap_map = NULL;
786         p->flags = 0;
787         swap_device_unlock(p);
788         swap_list_unlock();
789         vfree(swap_map);
790         err = 0;
791 
792 out_dput:
793         unlock_kernel();
794         path_release(&nd);
795 out:
796         return err;
797 }
```

Else the swap area was successfully deactivated to close the block device and mark the `swap_info_struct` free

772-773 Close the block device

774 Release the path information

776-777 Acquire the swap list and swap device lock

778-786 Reset the fields in `swap_info_struct` to default values

787-788 Release the swap list and swap device

788 Free the memory used for the `swap_map`

793 Release the BKL

794 Release the path information in the event we reached here via the error path

796 Return success or failure

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.

```
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 
539         start_mm = &init_mm;
540         atomic_inc(&init_mm.mm_users);
541 
```

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) {
571                         if (!*swap_map)
572                                 continue;
573                         retval = -ENOMEM;
574                         break;
575                 }
576 
577                 /*
578                  * Don't hold on to start_mm if it looks like exiting.
579                  */
580                 if (atomic_read(&start_mm->mm_users) == 1) {
581                         mmput(start_mm);
582                         start_mm = &init_mm;
```

```
583                         atomic_inc(&init_mm.mm_users);
584                 }
```

555 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

561-563 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

564-575 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`

580 Check to make sure this mm is not exiting. If it is, decrement its count and go back to `init_mm`

```
586                 /*
587                  * Wait for and lock page.  When do_swap_page races with
588                  * try_to_unuse, do_swap_page can handle the fault much
589                  * faster than try_to_unuse can locate the entry.  This
590                  * apparently redundant "wait_on_page" lets try_to_unuse
591                  * defer to do_swap_page in such a case - in some tests,
592                  * do_swap_page and try_to_unuse repeatedly compete.
593                  */
594                 wait_on_page(page);
595                 lock_page(page);
596 
597                 /*
598                  * Remove all references to entry, without blocking.
599                  * Whenever we reach init_mm, there's no address space
600                  * to search, but use it as a reminder to search shmem.
601                  */
602                 swcount = *swap_map;
603                 if (swcount > 1) {
604                         flush_page_to_ram(page);
605                         if (start_mm == &init_mm)
606                                 shmem_unuse(entry, page);
607                         else
608                                 unuse_process(start_mm, entry, page);
609                 }
```

594 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

595 Lock the page

602 Get the swap map reference count

603 If the count is positive then...

605 As the page is about to be inserted into proces page tables, it must be freed from the D-Cache or the process may not "see" changes made to the page by the kernel

605-606 If we are using the `init_mm`, call `shmem_unuse()` which will free the page from any shared memory regions that are in use

608 Else update the PTE in the current mm which references this page

```
610             if (*swap_map > 1) {
611                     int set_start_mm = (*swap_map >= swcount);
612                     struct list_head *p = &start_mm->mmlist;
613                     struct mm_struct *new_start_mm = start_mm;
614                     struct mm_struct *mm;
615 
616                     spin_lock(&mmlist_lock);
617                     while (*swap_map > 1 &&
618                             (p = p->next) != &start_mm->mmlist) {
619                             mm = list_entry(p, struct mm_struct,
                                        mmlist);
620                             swcount = *swap_map;
621                             if (mm == &init_mm) {
622                                     set_start_mm = 1;
623                                     shmem_unuse(entry, page);
624                             } else
625                                     unuse_process(mm, entry, page);
626                             if (set_start_mm && *swap_map < swcount) {
627                                     new_start_mm = mm;
628                                     set_start_mm = 0;
629                             }
630                     }
631                     atomic_inc(&new_start_mm->mm_users);
632                     spin_unlock(&mmlist_lock);
633                     mmput(start_mm);
634                     start_mm = new_start_mm;
635             }
```

610-635 If an entry still exists, begin traversing through all `mm_structs` finding references to this page and update the respective PTE

616 Lock the mm list

617-630 Keep searching until all `mm_structs` have been found. Do not traverse the full list more than once

619 Get the `mm_struct` for this list entry

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_struct`s starting from `init_mm` again

```
650             if (*swap_map == SWAP_MAP_MAX) {
651                     swap_list_lock();
652                     swap_device_lock(si);
653                     nr_swap_pages++;
654                     *swap_map = 1;
655                     swap_device_unlock(si);
656                     swap_list_unlock();
657                     reset_overflow = 1;
658             }
```

650 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 permanetly 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 occured

```
674             if ((*swap_map > 1) && PageDirty(page) &&
                  PageSwapCache(page)) {
675                     rw_swap_page(WRITE, page);
676                     lock_page(page);
677             }
678             if (PageSwapCache(page))
679                     delete_from_swap_cache(page);
```

674-677 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             SetPageDirty(page);
687             UnlockPage(page);
688             page_cache_release(page);
```

686 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

```
695             if (current->need_resched)
696                 schedule();
697         }
698 
699         mmput(start_mm);
700         if (reset_overflow) {
701             printk(KERN_WARNING "swapoff: cleared swap entry
                              overflow\n");
702             swap_overflow = 0;
703         }
704         return retval;
705 }
```

695-696 Call `schedule()` if necessary so the deactivation of swap does not hog the entire CPU

699 Drop our reference to the mm

700-703 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 happend

704 Return success or failure

Index