Memory Management in Linux

Desktop Companion to the Linux Source Code

by Abhishek Nayani
Mel Gorman & Rodrigo S. de Castro

Linux-2.4.19,
Version 0.4, 25 May '02

Contents

Preface

This document is a part of the Linux Kernel Documentation Project (http://freesoftware.fsf.org/lkdp) and attempts to describe how memory management is implemented in the Linux kernel. It is based on the Linux-2.4.19 kernel running on the intel 80x86 architecture. The reader is assumed to have some knowledge of memory management concepts and the intel 80x86 architecture. This document is best read with the kernel source by your side.

Acknowledgements

While preparing this document, I asked for reviewers on `#kernelnewbies` on irc.openprojects.net. I got a lot of response. The following individuals helped me with corrections, suggestions and material to improve this paper. They put in a big effort to help me get this document into its present shape. I would like to sincerely thank all of them. Naturally, all the mistakes you'll find in this book are mine.

Martin Devera, Joseph A Knapka, William Lee Irwin III,
Rik van Riel, David Parsons, Rene Herman, Srinidhi K.R.

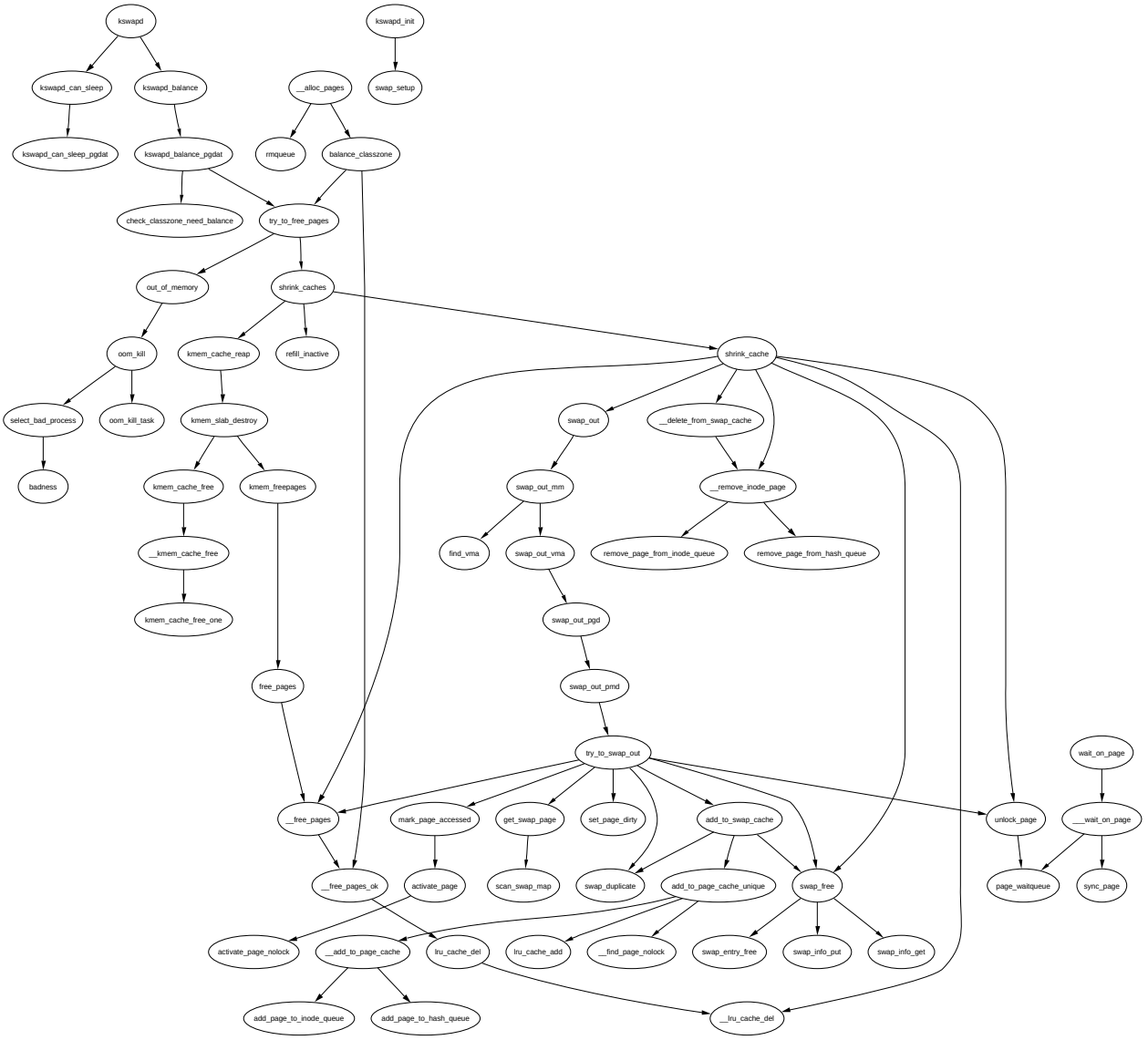

Figure 1: VM Callgraph [5] (magnify to get clear view)

Chapter 1

Initialization

1.1 Memory Detection

The first thing the kernel does (which is related to memory management) is find the amount of memory present in the system. This is done in the file arch/i386/boot/setup.S between the lines 281–382. Here it uses three routines, e820h to get the memory map, e801h to get the size and finally 88h which returns 0–64MB, all involving int 0x15. They are executed one after the other, regardless of the success or failure of any one of them. This redundancy is allowed as this is a very inexpensive one-time only process.

1.1.1 Method E820H

This method returns the memory classified into different types and also allows memory holes. It uses interrupt 0x15, function E820h (=AX) after which the method has been named. Its description and function is listed below:

```
AX = E820h
EAX = 0000E820h
EDX = 534D4150h ('SMAP')
EBX = continuation value or 00000000h
      to start at beginning of map
ECX = size of buffer for result,
      in bytes (should be >= 20 bytes)
ES:DI -> buffer for result

Return:
CF clear if successful
EAX = 534D4150h ('SMAP')
```

```
ES:DI buffer filled
EBX = next offset from which to copy
      or 00000000h if all done
ECX = actual length returned in bytes CF set on error
AH = error code (86h)
```

The format of the return buffer is:

```
Offset Size  Description
00h    QWORD base address
08h    QWORD length in bytes
10h    DWORD type of address range
```

The different memory types are:

```
01h     memory, available to OS
02h     reserved, not available
        (e.g. system ROM, memory-mapped device)
03h     ACPI Reclaim Memory
        (usable by OS after reading ACPI tables)
04h     ACPI NVS Memory (OS is required to save
        this memory between NVS sessions)

other not defined yet -- treat as Reserved
```

This method, uses the above routine to fill the memory pointed to by E820MAP[1] (address = 0x2d0), with the list of usable address/size duples (max 32). Eg. this routine returns the following information on my system (I modified the source to print the unmodified map).

```
    Address              Size              Type
0000000000000000     000000000009fc00        1
000000000009fc00     0000000000000400        1
00000000000f0000     0000000000010000        2
00000000fffc0000     0000000000010000        2
0000000000100000     00000000bf00000         1
```

This information in slightly more readable form:

[1] Declared in include/asm/e820.h

```
 Starting address       Size      Type
       0K               639K    Usable RAM
     639K                 1K    Usable RAM
     960K                64K    System ROM
    4G-64k               64K    System ROM
       1M               191M    Usable RAM
```

This is later converted into a more usable format in *sanitize_e820_map()*.

1.1.2 Method E801H

This routine will return the memory size in 1K chunks for the memory range 1MB to 16MB and in 64K chunks above 16MB. The description of the interrupt used is:

```
AX = E801h

Return:
CF clear if successful
AX   =   extended memory between 1M and 16M,
         in K (max 3C00h = 15MB)
BX   =   extended memory above 16M, in 64K blocks
CX   =   configured memory 1M to 16M, in K
DX   =   configured memory above 16M, in 64K blocks
CF set on error
```

The size calculated is stored in the address location 0x1e0h.

1.1.3 Method 88H

This routine is also used to find the amount of memory present in the system. This is expected to be successful in case the above routine fails as this function is supported by most BIOSes. It returns up to a maximum of 64MB or 16MB depending on the BIOS. The description of the interrupt used is:

```
AH = 88h

Return:
CF clear if successful
```

```
AX    =   number of contiguous KB starting
          at absolute address 100000h
        CF set on error
AH    =   status
          80h invalid command (PC,PCjr)
          86h unsupported function (XT,PS30)
```

The size calculated is stored in the address location 0x2h.

1.2 Provisional GDT

Before entering protected mode, the global descriptor table has to be setup. A provisional or temporary gdt is created with two entries, code and data segment, each covering the whole 4GB address space. The code[2] that loads the gdt is:

```
/** /arch/i386/boot/setup.S  **/

xorl    %eax, %eax      # Compute gdt_base
movw    %ds, %ax        # (Convert %ds:gdt to a linear ptr)
shll    $4, %eax
addl    $gdt, %eax
movl    %eax, (gdt_48+2)
lgdt    gdt_48          # load gdt with whatever is
                        # appropriate
```

where the variable gdt contains the table, gdt_48 contains the limit and the address of gdt. The code above gets the address of gdt and fills it in the address part of the gdt_48 variable.

1.3 Activating Paging

1.3.1 Significance of PAGE_OFFSET

The value of PAGE_OFFSET is 0xc0000000 which is 3GB. The linear address space of a process is divided into two parts:

[2] In file arch/i386/kernel/head.S

- Linear addresses from 0x00000000 to PAGE_OFFSET-1 can be addressed when the process is either in user or kernel mode.
- Linear addresses from PAGE_OFFSET to 0xffffffff can be addressed only when the process is in kernel mode. This address space is common to all the processes.

The address space after PAGE_OFFSET is reserved for the kernel and this is where the complete physical memory is mapped (eg. if a system has 64mb of RAM, it is mapped from PAGE_OFFSET to PAGE_OFFSET + 64mb). This address space is also used to map non-continuous physical memory into continuous virtual memory.

1.3.2 Provisional Kernel Page Tables

The purpose of this page directory is to map *virtual* address spaces 0–8mb and PAGE_OFFSET–(PAGE_OFFSET + 8mb) to the *physical* address space of 0–8mb. This mapping is done so that the address space out of which the code is executing, remains valid. Joseph A Knapka has explained this much better, from which I quote:

- *All pointers in the compiled kernel refer to addresses > PAGE_-OFFSET. That is, the kernel is linked under the assumption that its base address will be start_text (I think; I don't have the code on hand at the moment), which is defined to be PAGE_OFFSET+(some small constant, call it C).*
- *All the kernel bootstrap code is linked assuming that its base address is 0+C.*

head.S is part of the bootstrap code. It's running in protected mode with paging turned off, so all addresses are physical. In particular, the instruction pointer is fetching instructions based on physical address. The instruction that turns on paging (movl %eax, %cr0) is located, say, at some physical address A.

As soon as we set the paging bit in cr0, paging is enabled, and starting at the very next instruction, all addressing, including instruction fetches, pass through the address translation mechanism (page tables). IOW, all address are henceforth virtual. That means that

1. *We must have valid page tables, and*
2. *Those tables must properly map the instruction pointer to the next instruction to be executed.*

*That next instruction is physically located at address A+4 (the address immediately after the "movl %eax, %cr0" instruction), but from the point of view of all the kernel code – which has been linked at PAGE_OFFSET – that instruction is located at virtual address PAGE_OFFSET+(A+4). Turning on paging, however, does not magically change the value of EIP. The CPU fetches the next instruction from ***virtual*** address A+4; that instruction is the beginning of a short sequence that effectively relocates the instruction pointer to point to the code at PAGE_OFFSET+A+(something).*

*But since the CPU is, for those few instructions, fetching instructions based on physical addresses ***but having those instructions pass through address translation***, we must ensure that both the physical addresses and the virtual addresses are :*

1. *Valid virtual addresses,and*

2. *Point to the same code.*

That means that at the very least, the initial page tables must map virtual address PAGE_OFFSET+(A+4) to physical address (A+4), and must map virtual address A+4 to physical address A+4. This dual mapping for the first 8MB of physical RAM is exactly what the initial page tables accomplish. The 8MB initially mapped is more or less arbitrary. It's certain that no bootable kernel will be greater than 8MB in size. The identity mapping is discarded when the MM system gets initialized.

The variable swapper_pg_dir contains the page directory for the kernel, which is statically initialized at compile time. Using ".org" directives of the assembler, swapper_pg_dir is placed at address 0x00101000[3], similarly the first page table entry pg0 is placed at 0x00102000 and the second page table entry pg1 at 0x00103000. The page table entry pg1 is followed by empty_zero_page[4] at 0x00103000, whose only purpose is to act as a marker to denote the end, in a loop used to initialize the page tables. The swapper_pg_dir is as follows:

```
/** /arch/i386/kernel/head.S  **/

.org 0x1000
ENTRY(swapper_pg_dir)
  .long 0x00102007
  .long 0x00103007
```

[3]The kernel starts at 0x00100000 == 1MB, so .org 0x1000 is taken w.r.t the start of the kernel

[4]It is also used to store the boot parameters and the command line of the kernel.

```
.fill BOOT_USER_PGD_PTRS-2,4,0
/* default: 766 entries */
.long 0x00102007
.long 0x00103007
/* default: 254 entries */
.fill BOOT_KERNEL_PGD_PTRS-2,4,0
```

In the above structure:

- First and second entries point to pg0 and pg1 respectively.
- BOOT_USER_PGD_PTRS[5] gives the number of page directory entries mapping the user space (0–3GB) which is 0x300 (768 in decimal). This is used to initialize the rest of the entries mapping upto 3GB to zero.
- Page tables mapping PAGE_OFFSET to (PAGE_OFFSET + 8mb) are also initialized with pg0 and pg1 (lines 386–387).
- BOOT_KERNEL_PGD_PTRS gives the number of page directory entries mapping the kernel space (3GB–4GB). This is used to initialize the rest of remaining page tables to zero.

The page tables pg0 and pg1 are initialized in this loop:

```
/** /arch/i386/kernel/head.S  **/

   /* Initialize page tables */
   movl $pg0-__PAGE_OFFSET,%edi /* initialize page tables */
   movl $007,%eax           /* "007" doesn't mean with right
            to kill, but PRESENT+RW+USER */
2: stosl
   add $0x1000,%eax
   cmp $empty_zero_page-__PAGE_OFFSET,%edi
   jne 2b
```

In the above code:

1. Register EDI is loaded with the address of pg0.
2. EAX is loaded with the address + attributes of the page table entry. The combination maps the first 4k, starting from 0x00000000 with the attributes PRESENT+RW+USER.

[5] A macro defined in /include/asm-386/pgtable.h

3. The instruction "stosl" stores the contents of EAX at the address pointed by EDI, and increments EDI.

4. The base address of the page table entry is incremented by 0x1000 (4k). The attributes remain the same.

5. Check is made to see if we have reached the end of the loop by comparing the address pointed to be EDI with the address of empty_zero_page. If not, it jumps back to label[6] 2 and loops.

By the end of the loop, the complete 8mb will be mapped.

> **Note:** In the above code, while accessing pg0, swapper_pg_dir and other variables, they are addressed as pg0 - __PAGE_OFFSET, swapper_pg_dir - __PAGE_OFFSET and so on (ie. PAGE_OFFSET is being deducted). This is because the code (vmlinux) is actually linked to start at address starting from PAGE_OFFSET + 1mb (0xc0100000). So all symbols have addresses above PAGE_OFFSET, eg. swapper_pg_dir gets the address 0xc0101000. Therefore to get the physical addresses, PAGE_OFFSET must be deducted from the symbol address. This linking information is specified in the file arch/i386/vmlinux.lds. Also to get a better idea, "objdump -D vmlinux" will show you all the symbols and their addresses.

1.3.3 Paging

Paging is enabled by setting the most significant bit (PG) of the CR0 register. This is done in the following code:

```
/** /arch/i386/kernel/head.S  **/
    /*
    * Enable paging
    */
 3:
    movl $swapper_pg_dir-__PAGE_OFFSET,%eax
    movl %eax,%cr3    /* set the page table pointer.. */
    movl %cr0,%eax
    orl $0x80000000,%eax
    movl %eax,%cr0    /* ..and set paging (PG) bit */
    jmp 1f            /* flush the prefetch-queue */
```

[6] The char after 2 is a specifier which tells the assembler to jump forward or backward

```
1:
   movl $1f,%eax
   jmp *%eax         /* make sure eip is relocated */
1:
```

After enabling paged memory management, the first jump flushes the instruction queue. This is done because the instructions which have been already decoded (in the queue) will be using the old addresses. The second jump effectively relocates the instruction pointer to PAGE_OFFSET + something.

1.4 Final GDT

After the paging has been enabled, the final gdt is loaded. The gdt now contains code and data segments for both user and kernel. Along with these, segments are defined for APM and space is left for TSSs and LDTs of processes. Linux uses segments in a very limited way, ie. it uses the flat model, in which segments are created for code and data addressing the full 4GB memory space. The gdt is as follows:

```
/** /arch/i386/kernel/head.S  **/

ENTRY(gdt_table)
 .quad 0x0000000000000000 /*NULL descriptor */
 .quad 0x0000000000000000 /*not used */
 .quad 0x00cf9a000000ffff /*0x10 kernel 4GB code */
 .quad 0x00cf92000000ffff /*0x18 kernel 4GB data */
 .quad 0x00cffa000000ffff /*0x23 user   4GB code */
 .quad 0x00cff2000000ffff /*0x2b user   4GB data */
 .quad 0x0000000000000000 /*not used */
 .quad 0x0000000000000000 /*not used */
 /*
 * The APM segments have byte granularity and their bases
 * and limits are set at run time.
 */
 .quad 0x0040920000000000 /*0x40 APM set up for bad BIOS's
 .quad 0x00409a0000000000 /*0x48 APM CS    code*/
 .quad 0x00009a0000000000 /*0x50 APM CS 16 code (16 bit)*/
 .quad 0x0040920000000000 /*0x58 APM DS    data*/
 .fill NR_CPUS*4,8,0      /*space for TSS's and LDT's*/
```

1.5 Memory Detection Revisited

As we have previously seen, three assembly routines were used to detect the memory regions/size and the information was stored in some place in memory. The routine setup_arch()[7], which is called by start_kernel() to do architecture dependent initializations, is responsible for processing this information and setup up high level data structures necessary to do memory management. The following are the functions and their descriptions in the order they are called:

1.5.1 Function setup_arch()

File: arch/i386/kernel/setup.c
This description only covers code related to memory management.

```
setup_memory_region();
```

This call processes the memory map and stores the memory layout information in the global variable e820. Refer to section 1.5.2 for more details.

```
parse_mem_cmdline(cmdline_p);
```

This call will override the memory detection code with the user supplied values.

```
#define PFN_UP(x)       (((x) + PAGE_SIZE-1) >> PAGE_SHIFT)
#define PFN_DOWN(x)     ((x) >> PAGE_SHIFT)
#define PFN_PHYS(x)     ((x) << PAGE_SHIFT)
```

Description of the macros:

PFN_UP
: Returns the page frame number, after rounding the address to the next page frame boundary.

PFN_DOWN
: Returns the page frame number, after rounding the address to the previous page frame boundary.

[7] This routine is in the file arch/i386/kernel/setup.c

PFN_PHYS
Returns the physical address for the given page number.

```
/*
* 128MB for vmalloc and initrd
*/
#define VMALLOC_RESERVE (unsigned long)(128 << 20)
#define MAXMEM (unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE)
#define MAXMEM_PFN PFN_DOWN(MAXMEM)
#define MAX_NONPAE_PFN (1 << 20)
```

Description of the macros:

VMALLOC_RESERVE
Address space of this size (in the kernel address space) is reserved for vmalloc, evaluates to 128MB.

MAXMEM
Gives the maximum amount of RAM that can be directly mapped by the kernel. It evaluates to 896MB . In the above macro, -PAGE_OFFSET evaluates to 1GB (overflow of unsigned long).

MAXMEM_PFN
Returns the page frame number of the maximum memory which can be directly mapped by the kernel.

MAX_NONPAE_PFN
Gives the page frame number of the first page after 4GB. Memory above this can be accessed only when PAE has been enabled.

`Update:` The definitions of both *VMALLOC_RESERVE* and *MAXMEM* have been shifted to `include/asm-i386/page.h`.

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

The macro __pa is declared in the file `include/asm-i386/page.h`, it returns the physical address when given a virtual address. It just subtracts PAGE_OFFSET from the given value to do this. The identifier _end is used to represent the end of the kernel in memory. So the value that is stored in start_pfn is the page frame number immediately following the kernel.

```
/*
* Find the highest page frame number we have available
*/
max_pfn = 0;
for (i = 0; i < e820.nr_map; i++) {
   unsigned long start, end;
   /* RAM? */
   if (e820.map[i].type != E820_RAM)
      continue;
   start = PFN_UP(e820.map[i].addr);
   end = PFN_DOWN(e820.map[i].addr + e820.map[i].size);
   if (start >= end)
      continue;
   if (end > max_pfn)
      max_pfn = end;
}
```

The above code loops through the memory regions of type E820_RAM (usable RAM) and stores the page frame number of the last page frame in max_pfn.

```
/*
* Determine low and high memory ranges:
*/
max_low_pfn = max_pfn;
if (max_low_pfn > MAXMEM_PFN) {
```

If the system has memory greater than 896MB, the following code is used to find out the amount of HIGHMEM.

```
    if (highmem_pages == -1)
        highmem_pages = max_pfn - MAXMEM_PFN;
```

The variable *highmem_pages* is used to store the no. of page frames above 896mb. It is initialized to -1 at the time of definition, so we know that the user has not specified any value for the highmem on the kernel command line using the *highmem=size* option if it remains equal to -1. The *highmem=size* option allows the user to specify the exact amount of high memory to use. Check the function `parse_mem_cmdline` to see how it is set. So the above code checks if the user has specified any value for the highmem, if not it calculates the higmem by subtracting the last page frame of *normal* memory from the total number of page frames.

```
    if (highmem_pages + MAXMEM_PFN < max_pfn)
        max_pfn = MAXMEM_PFN + highmem_pages;
```

This condition is used to adjust the value of *max_pfn* when the sum of highmem pages and normal pages is less than the total no. of pages. This happens when the user has specified lesser no. of highmem pages on the kernel command line than there are in the system.

```
    if (highmem_pages + MAXMEM_PFN > max_pfn) {
        printk("only %luMB highmem pages available,
                ignoring highmem size of %uMB.\n",
        pages_to_mb(max_pfn - MAXMEM_PFN),
        pages_to_mb(highmem_pages));
        highmem_pages = 0;
    }
```

This code is executed if the user specifies more no. of highmem pages than there are in the system on the kernel command line. The above code will print an error message and ignores the highmem pages.

```
    max_low_pfn = MAXMEM_PFN;

#ifndef CONFIG_HIGHMEM
/* Maximum memory usable is what is directly addressable */
    printk(KERN_WARNING "Warning only %ldMB will be used.\n",
                                                MAXMEM>>20);

    if (max_pfn > MAX_NONPAE_PFN)
        printk(KERN_WARNING "Use a PAE enabled kernel.\n");
    else
        printk(KERN_WARNING "Use HIGHMEM enabled kernel");

#else /* !CONFIG_HIGHMEM */
```

If `CONFIG_HIGHMEM` is not defined, the above code prints the amount of RAM that will be used (which is the amount of RAM which is directly addressable ie. max of 896mb). If the available RAM is greater than 4GB, then it prints a message to use a PAE enabled kernel (which allows the use of 64GB of memory in processors starting from pentium pro) else suggests to enable HIGHMEM.

```
#ifndef CONFIG_X86_PAE
    if (max_pfn > MAX_NONPAE_PFN) {
        max_pfn = MAX_NONPAE_PFN;
        printk(KERN_WARNING "Warning only 4GB will be used");
```

```
        printk(KERN_WARNING "Use a PAE enabled kernel.\n");
    }
#endif /* !CONFIG_X86_PAE */
#endif /* !CONFIG_HIGHMEM */
```

If CONFIG_HIGHMEM was enabled but the system has RAM more than 4GB and CONFIG_X86_PAE was not enabled, it warns the user to enable it to use memory more than 4GB.

```
} else {
    if (highmem_pages == -1)
        highmem_pages = 0;
```

It comes here if the amount of RAM in the system is less than 896mb. Even here, the user has got the option to use some *normal* memory as highmem (mainly for debugging purposes). So the above code checks to see if the user wants to have any higmem.

```
#if CONFIG_HIGHMEM
    if (highmem_pages >= max_pfn) {
        printk(KERN_ERR "highmem size specified (%uMB)
               is bigger than pages available (%luMB)!.\n",
               pages_to_mb(highmem_pages),
               pages_to_mb(max_pfn));
        highmem_pages = 0;
    }
```

If CONFIG_HIGHMEM is enabled, the above code checks if the user specified highmem size is greater than the amount of RAM present in the system. This request gets completely ignored.

```
    if (highmem_pages) {
      if(max_low_pfn-highmem_pages < 64*1024*1024/PAGE_SIZE){
        printk(KERN_ERR "highmem size %uMB results in smaller
                 than 64MB lowmem, ignoring it.\n",
                 pages_to_mb(highmem_pages));
          highmem_pages = 0;
        }
        max_low_pfn -= highmem_pages;
    }
```

You can only use some amount of normal memory as high memory if you have atleast 64mb of RAM after deducting memory for highmem. So, if your system has 192mb of RAM, you can use upto 128mb as highmem. If this condition is not satisfied, no highmem is created. If the request can be satisfied, the highmem is deducted from *max_low_pfn* which gives the new amount of normal memory present in the system.

```
#else
    if (highmem_pages)
        printk(KERN_ERR
       "ignoring highmem size on non-highmem kernel!\n");
#endif
}
```

The normal memory can be used as highmem only if `CONFIG_HIGHMEM` was enabled.

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

The above code just prints the available (usable) memory above 896MB if CONFIG_HIGHMEM has been enabled.

```
/*
* Initialize the boot-time allocator (with low memory only):
*/
bootmap_size = init_bootmem(start_pfn, max_low_pfn);
```

This call initializes the bootmem allocator. Refer to section 1.7.2 for more details. It also reserves all the pages.

```
/*
* Register fully available low RAM pages with the
* bootmem allocator.
*/
```

```
for (i = 0; i < e820.nr_map; i++) {
   unsigned long curr_pfn, last_pfn, size;
/*
* Reserve usable low memory
*/
   if (e820.map[i].type != E820_RAM)
      continue;
/*
* We are rounding up the start address of usable memory:
*/
   curr_pfn = PFN_UP(e820.map[i].addr);
   if (curr_pfn >= max_low_pfn)
      continue;
/*
* ... and at the end of the usable range downwards:
*/
   last_pfn = PFN_DOWN(e820.map[i].addr +
                       e820.map[i].size);
   if (last_pfn > max_low_pfn)
      last_pfn = max_low_pfn;
/*
* .. finally, did all the rounding and playing
* around just make the area go away?
*/
   if (last_pfn <= curr_pfn)
      continue;
   size = last_pfn - curr_pfn;
   free_bootmem(PFN_PHYS(curr_pfn), PFN_PHYS(size));
}
```

This loop goes through all usable RAM and marks it as available using the free_bootmem() routine. So after this, only memory of type 1 (usable RAM) is marked as available. Refer to section 1.7.3 for more details.

```
/*
* Reserve the bootmem bitmap itself as well. We do this in two
* steps (first step was init_bootmem()) because this catches
* the (very unlikely) case of us accidentally initializing the
* bootmem allocator with an invalid RAM area.
*/
reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +
```

```
                bootmap_size + PAGE_SIZE-1) - (HIGH_MEMORY));
```

This call marks the memory occupied by the kernel and the bootmem bitmap as reserved. Here HIGH_MEMORY is equal to 1MB, the start of the kernel. Refer to section 1.7.4 for more details.

```
paging_init();
```

This call initializes the data structures necessary for paged memory management. Refer to section 1.8.1 for more details.

1.5.2 Function setup_memory_region()

File: arch/i386/kernel/setup.c
This function is used to process and copy the memory map (section 1.1.1) into the global variable e820. If it fails to do that, it creates a fake memory map. It basically does this:

- Call sanitize_e820_map() with the location of the e820 retrieved data which does the actual processing of the raw data.
- Call copy_e820_map() to do the actual copying.
- If unsuccessful, create a fake memory map, one 0–636k and the other 1mb to the maximum of either of what routines e801h or 88h returns.
- Print the final memory map.

1.5.3 Function sanitize_e820_map()

File: arch/i386/kernel/setup.c
This function is used to remove any overlaps in the memory maps reported by the BIOS. More detail later.

1.5.4 Function copy_e820_map()

File: arch/i386/kernel/setup.c
This function copies the memory maps after doing some checks. It also does some sanity checking.

```
if (nr_map < 2)
    return -1;
```

```
do {
    unsigned long long start = biosmap->addr;
    unsigned long long size = biosmap->size;
    unsigned long long end = start + size;
    unsigned long type = biosmap->type;
```

Read one entry.

```
/* Overflow in 64 bits? Ignore the memory map. */
    if (start > end)
        return -1;
/*
* Some BIOSes claim RAM in the 640k - 1M region.
* Not right. Fix it up.
*/
    if (type == E820_RAM) {
        if (start < 0x100000ULL && end > 0xA0000ULL) {
```

If start is below 1MB and end is greater than 640K:

```
            if (start < 0xA0000ULL)
              add_memory_region(start, 0xA0000ULL-start, type);
```

If *start* is less than 640K, add the memory region from *start* to 640k.

```
             if (end <= 0x100000ULL)
               continue;
             start = 0x100000ULL;
             size = end - start;
```

If *end* is greater than 1MB, then start from 1MB and add the memory region avoiding the 640k to 1MB hole.

```
        }
    }

    add_memory_region(start, size, type);
} while (biosmap++,--nr_map);
return 0;
```

1.5.5 Function add_memory_region()

File: arch/i386/kernel/setup.c
Adds the actual entry to e820.

```
int x = e820.nr_map;
```

Get the number of entries already added, used to add the new entry at the end.

```
if (x == E820MAX) {
    printk(KERN_ERR "Oops! Too many entries in
                            the memory map!\n");
    return;
}
```

If the number of entries has already reached 32, display a warning and return.

```
e820.map[x].addr = start;
e820.map[x].size = size;
e820.map[x].type = type;
e820.nr_map++;
```

Add the entry and increment *nr_map*.

1.5.6 Function print_memory_map()

File: arch/i386/kernel/setup.c
Prints the memory map to the console. eg:

```
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 00000000000a0000 (usable)
BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
BIOS-e820: 0000000000100000 - 000000000c000000 (usable)
BIOS-e820: 00000000fffff0000 - 0000000100000000 (reserved)
```

The above is the sanitised version of the data we got from the routine E820h.

1.6 NUMA

Before going any further, a brief overview of NUMA. From Documentation/vm/numa (by Kanoj Sarcar):

It is an architecture where the memory access times for different regions of memory from a given processor varies according to the "distance" of the memory region from the processor. Each region of memory to which access times are the same from any cpu, is called a node. On such architectures, it is beneficial if the kernel tries to minimize inter node communications. Schemes for this range from kernel text and read-only data replication across nodes, and trying to house all the data structures that key components of the kernel need on memory on that node.

Currently, all the numa support is to provide efficient handling of widely discontiguous physical memory, so architectures which are not NUMA but can have huge holes in the physical address space can use the same code. All this code is bracketed by CONFIG_DISCONTIGMEM.

The initial port includes NUMAizing the bootmem allocator code by encapsulating all the pieces of information into a bootmem_data_t structure. Node specific calls have been added to the allocator. In theory, any platform which uses the bootmem allocator should be able to to put the bootmem and mem_map data structures anywhere it deems best.

Each node's page allocation data structures have also been encapsulated into a pg_data_t. The bootmem_data_t is just one part of this. To make the code look uniform between NUMA and regular UMA platforms, UMA platforms have a statically allocated pg_data_t too (contig_page_data). For the sake of uniformity, the variable "numnodes" is also defined for all platforms. As we run benchmarks, we might decide to NUMAize more variables like low_on_memory, nr_free_pages etc into the pg_data_t.

1.6.1 struct pglist_data

File: include/linux/mmzone.h
Information of each node is stored in a structure of type pg_data_t. The structure is as follows:

```
typedef struct pglist_data {
     zone_t node_zones[MAX_NR_ZONES];
     zonelist_t node_zonelists[GFP_ZONEMASK+1];
     int nr_zones;
     struct page *node_mem_map;
     unsigned long *valid_addr_bitmap;
```

```
    struct bootmem_data *bdata;
    unsigned long node_start_paddr;
    unsigned long node_start_mapnr;
    unsigned long node_size;
    int node_id;
    struct pglist_data *node_next;
} pg_data_t;
```

The description of the elements of the above structure follows:

node_zones
Array of zones present in the node (MAX_NR_ZONES is 3). For more information about a zone refer section 1.9 .

node_zonelists
Its an array of zonelist_t structures. A zonelist_t is a structure containing a null terminated array of 3 zone pointers (total 4, 1 for NULL). Total of GFP_ZONEMASK+1 (16) zonelist_t structures are created. For each type of requirement, there is a mask specifying the order of zones, in which they must be queried for allocation of memory (priority). Each of these structures represent one order (sequence of priority), and are passed on to memory allocation routines.

nr_zones
No. of zones present in this node.

node_mem_map
Array of structures representing the physical pages of the node.

valid_addr_bitmap
Contains a bitmap of usable and unusable pages.

bdata
The bootmem structure, contains information of the bootmem of the node. More information in section 1.7.

node_start_paddr
The start of the physical address of the node.

node_start_mapnr
The page frame number of the first page of the node.

node_size
The total number of pages present on this node.

node_id
The index of the current node.

node_next
A circular linked list of nodes is maintained. This points to the next node (in i386, made to point to itself).

For i386, there is only one node which is represented by *contig_page_data*[8] of type pg_data_t. The *bdata* member of contig_page_data is initialized to zeroes by assigning it to a statically allocated bootmem structure (variables declared static are automatically initialized to 0, the variable contig_bootmem_data is used only for this purpose).

1.7 Bootmem Allocator

The bootmem allocator is used only at boot, to reserve and allocate pages for kernel use. It uses a bitmap to keep track of reserved and free pages. This bitmap is created exactly after the end of the kernel (after _end) and is used to manage only low memory, ie. less than 896MB. This structure used to store the bitmap is of type bootmem_data.

1.7.1 struct bootmem_data

File: include/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;
```

The Descriptions of the member elements:

node_boot_start
The start of the bootmem memory (the first page, normally 0).

[8] declared in mm/numa.c

node_low_pfn
Contains the end of low memory of the node.

node_bootmem_map
Start of the bootmem bitmap.

last_offset
This is used to store the offset of the last byte allocated in the previous allocation from *last_pos* to avoid internal memory fragmentation (see below).

last_pos
This is used to store the page frame number of the last page of the previous allocation. It is used in the function `__alloc_bootmem_core()` to reduce internal fragmentation by merging contiguous memory requests.

1.7.2 Function init_bootmem()

File: mm/bootmem.c
Prototypes:

```
unsigned long init_bootmem(unsigned long start,
                           unsigned long pages);
unsigned long init_bootmem_core (pg_data_t *pgdat,
                                 unsigned long mapstart,
                                 unsigned long start,
                                 unsigned long end);
```

The function `init_bootmem()` is used only at initialization to setup the bootmem allocator. It is actually a wrapper over the function `init_bootmem_core()` which is NUMA aware. The function `init_bootmem()` is passed the page frame number of the end of the kernel and max_low_pfn, the page frame number of the end of low memory. It passes this information along with the node *contig_page_data* to `init_bootmem_core()`.

```
bootmem_data_t *bdata = pgdat->bdata;
```

Initialize bdata, this is done just for the convenience.

```
unsigned long mapsize = ((end - start)+7)/8;
```

The size of the bootmem bitmap is calculated and stored in mapsize. In the above line, (end - start) gives the number of page frames present. We are adding 7 to round it upwards before dividing to get the number of bytes required (each byte maps 8 page frames).

```
pgdat->node_next = pgdat_list;
pgdat_list = pgdat;
```

The variable *pgdat_list* is used to point to the head of the circular linked list of nodes. Since we have only one node, make it point to itself.

```
mapsize = (mapsize + (sizeof(long) - 1UL)) &
                                    ~(sizeof(long) - 1UL);
```

The above line rounds mapsize upwards to the next multiple of 4 (the cpu word size).

1. (mapsize + (sizeof(long) - 1UL) is used to round it upwards, here (sizeof(long) - 1UL) = (4 - 1) = 3.
2. $\sim (sizeof(long) - 1UL)$ is used to mask the result and make it a multiple of 4.

Eg. assume there are 40 pages of physical memory. So we get the mapsize as 5 bytes. So the above operation becomes $(5 + (4 - 1))\& \sim (4 - 1)$ which becomes (8& $\sim$ 3) which is (00001000&11111100). The last two bits get masked off, effectively making it a multiple of 4.

```
bdata->node_bootmem_map = phys_to_virt(mapstart
                                        << PAGE_SHIFT);
```

Point node_bootmem_map to mapstart which is the end of the kernel. The macro phys_to_virt() returns the virtual address of the given physical address (it just adds PAGE_OFFSET to the given value).

```
bdata->node_boot_start = (start << PAGE_SHIFT);
```

Initialize node_boot_start with the starting physical address of the node (here its 0x00000000).

```
bdata->node_low_pfn = end;
```

Initialize node_low_pfn with the page frame number of the last page of low memory.

```
/*
* Initially all pages are reserved - setup_arch() has to
```

```
* register free RAM areas explicitly.
*/
memset(bdata->node_bootmem_map, 0xff, mapsize);

return mapsize;
```

Mark all page frames as reserved by setting all bits to 1 and return the mapsize.

1.7.3 Function free_bootmem()

File: mm/bootmem.c
Prototypes:

```
void free_bootmem (unsigned long addr,
                   unsigned long size);
void free_bootmem_core (bootmem_data_t *bdata,
                        unsigned long addr,
                        unsigned long size);
```

This function is used to mark the given range of pages as free (available) in the bootmem bitmap. As above the real work is done by the NUMA aware `free_bootmem_core()`.

```
/*
* round down end of usable mem, partially free pages are
* considered reserved.
*/
unsigned long sidx;
unsigned long eidx = (addr + size -
                      bdata->node_boot_start)/PAGE_SIZE;
```

The variable eidx is initialized to the total no. of page frames.

```
unsigned long end = (addr + size)/PAGE_SIZE;
```

The variable end is initialized to the page frame no. of the last page.

```
if (!size) BUG();
if (end > bdata->node_low_pfn)
    BUG();
```

The above two are assert statements checking impossible conditions.

```
/*
* Round up the beginning of the address.
*/
start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
sidx = start - (bdata->node_boot_start/PAGE_SIZE);
```

start is initialized to the page frame no. of the first page (rounded upwards) and sidx (start index) to the page frame no. relative to node_boot_start.

```
for (i = sidx; i < eidx; i++) {
    if (!test_and_clear_bit(i, bdata->node_bootmem_map))
        BUG();
}
```

Clear all the bits from sidx to eidx marking all the pages as available.

1.7.4 Function reserve_bootmem()

File: mm/bootmem.c
Prototypes:

```
void reserve_bootmem (unsigned long addr, unsigned long size);
void reserve_bootmem_core(bootmem_data_t *bdata,
                          unsigned long addr,
                          unsigned long size);
```

This function is used for reserving pages. To reserve a page, it just sets the appropriate bit to 1 in the bootmem bitmap.

```
unsigned long sidx = (addr - bdata->node_boot_start)
                                          / PAGE_SIZE;
```

The identifier sidx (start index) in initialized to the page frame no. relative to node_boot_start.

```
unsigned long eidx = (addr + size - bdata->node_boot_start +
                                       PAGE_SIZE-1)/PAGE_SIZE;
```

The variable eidx is initialized to the total no. of page frames (rounded upwards).

```
unsigned long end = (addr + size + PAGE_SIZE-1)/PAGE_SIZE;
```

The variable end is initialized to the page frame no. of the last page (rounded upwards).

```
if (!size) BUG();
if (sidx < 0)
    BUG();
if (eidx < 0)
    BUG();
if (sidx >= eidx)
    BUG();
if ((addr >> PAGE_SHIFT) >= bdata->node_low_pfn)
    BUG();
if (end > bdata->node_low_pfn)
    BUG();
```

Various assert conditions.

```
for (i = sidx; i < eidx; i++)
    if (test_and_set_bit(i, bdata->node_bootmem_map))
       printk("hm, page %08lx reserved twice.\n",
                                    i*PAGE_SIZE);
```

Set the bits from sidx to eidx to 1.

1.7.5 Function __alloc_bootmem()

File: mm/bootmem.c
Prototypes:

```
void * __alloc_bootmem (unsigned long size,
                        unsigned long align,
                        unsigned long goal);
void * __alloc_bootmem_core (bootmem_data_t *bdata,
                             unsigned long size,
                             unsigned long align,
                             unsigned long goal);
```

The function `__alloc_bootmem()` tries to allocate pages from different nodes in a round robin manner. Since in i386 there is only one node, it is the one that is used every time. The description of `__alloc_bootmem_core()` follows:

```
unsigned long i, start = 0;
void *ret;
unsigned long offset, remaining_size;
unsigned long areasize, preferred, incr;
unsigned long eidx = bdata->node_low_pfn -
                    (bdata->node_boot_start >> PAGE_SHIFT);
```

Initialize eidx with the total number of page frames present in the node.

```
if (!size) BUG();
if (align & (align-1))
    BUG();
```

Assert conditions. We check to see if size is not zero and align is a power of 2.

```
/*
* We try to allocate bootmem pages above 'goal'
* first, then we try to allocate lower pages.
*/
if (goal && (goal >= bdata->node_boot_start) &&
            ((goal >> PAGE_SHIFT) < bdata->node_low_pfn)) {
    preferred = goal - bdata->node_boot_start;
} else
    preferred = 0;
preferred = ((preferred + align - 1) & ~(align - 1))
                                              >> PAGE_SHIFT;
```

The preferred page frame for the begining of the allocation is calculated in two steps:

1. If goal is non-zero and is valid, *preferred* is initialized with it (after correcting it w.r.t node_boot_start) else it is zero.

2. The preferred physical address is aligned according to the parameter align and the respective page frame number is derived.

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

Get the number of pages required (rounded upwards).

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

The above line of code calculates the *incr* value (a.k.a. step). This value is added to the *preferred* address in the loop below to find free memory of the given alignment. The above line is using a gcc extension which evaluates to:

```
incr = (align >> PAGE_SHIFT) ? (align >> PAGE_SHIFT) : 1;
```

If the alignment required is greater than the size of a page, then *incr* is align/4k pages else it is 1 page.

```
restart_scan:
for (i = preferred; i < eidx; i += incr) {
    unsigned long j;
    if (test_bit(i, bdata->node_bootmem_map))
        continue;
```

This loop is used to find the first free page frame starting from the preferred page frame number. The macro `test_bit()` returns 1 if the given bit is set.

```
    for (j = i + 1; j < i + areasize; ++j) {
        if (j >= eidx)
            goto fail_block;
        if (test_bit (j, bdata->node_bootmem_map))
           goto fail_block;
    }
```

This loop is used to see if there are enough free page frames after the first to satisfy the memory request. If any of the pages is not free, jump to *fail_block*.

```
    start = i;
    goto found;
```

If it came till here, then enough free page frames have been found starting from *i*. So jump over the fail_block and continue.

```
    fail_block:;
}
if (preferred) {
    preferred = 0;
    goto restart_scan;
```

If it came here, then successive page frames tp satisfy the request were not found from the preferred page frame. So we ignore the preferred value (hint) and start scanning from 0.

```
}
return NULL;
```

Enough memory was not found to satisfy the request. Exit returning NULL.

`found:`

Enough memory was found. Continue processing the request.

```
if (start >= eidx)
    BUG();
```

Check for the impossible conditions (assert).

```
/*
* Is the next page of the previous allocation-end the start
* of this allocation's buffer? If yes then we can 'merge'
* the previous partial page with this allocation.
*/
if (align <= PAGE_SIZE && bdata->last_offset
                       && bdata->last_pos+1 == start) {
    offset = (bdata->last_offset+align-1) & ~(align-1);
    if (offset > PAGE_SIZE)
        BUG();
    remaining_size = PAGE_SIZE-offset;
```

The *if* statement checks for these conditions:

1. The alignment requested is less than page size (4k). This is done because if an alignment of size `PAGE_SIZE` was requested, then there in no chance of merging, as we need to start on a page boundary (completely new page).

2. The variable last_offset is non-zero. If it is zero, the previous allocation completed on a perfect page frame boundary, so no internal fragmentation.

3. Check whether the present memory request is adjacent to the previous memory requet, if it is, then the two allocations can be merged.

If all conditions are satisfied, *remaining_size* is initialized with the space remaining in the last page of previous allocation.

```
    if (size < remaining_size) {
        areasize = 0;
        // last_pos unchanged
        bdata->last_offset = offset+size;
        ret = phys_to_virt(bdata->last_pos*PAGE_SIZE
                + offset + bdata->node_boot_start);
```

If size of the memory request is smaller than the space available in the last page of the previous allocation, then there is no need to reserve any new pages. The variable last_offset is incremented to new offset, last_pos is unchanged because it is still not full. The physical address of the start of this new allocation is stored in the variable *ret*. The macro `phys_to_virt()` returns the virtual address of given physical address.

```
    } else {
        remaining_size = size - remaining_size;
        areasize = (remaining_size+PAGE_SIZE-1)/PAGE_SIZE;
        ret = phys_to_virt(bdata->last_pos*PAGE_SIZE
                + offset + bdata->node_boot_start);
        bdata->last_pos = start+areasize-1;
        bdata->last_offset = remaining_size;
```

The requested size is greated than the remaining size. So now we need to find the number of pages required after subtracting the space left in the last page of the previous allocation and update the variables last_pos and last_offset.

Eg. in a previous allocation, if 9k was allocated, page_pos will be 3 (as three page frames are required), the internal fragmentation will be 12k - 9k = 3k. So page_offset would be 1k and remaining size being 3k. If the new request is for 1k, then it would fit in the 3rd page frame itself, but if it was 10k, ((10 - 3) + PAGE_SIZE-1)/PAGE_SIZE would give the number of new pages that need to be reserved. Which is 2 (for 7k), so page_pos will now become 3+2 = 5 and the new page_offset is 3k.

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

This code is executed if we cannot merge as some condition has failed, we just set the last_pos and last_offset to their new values directly without considering their old values. The value of last_pos is incremented by the number of page frames requested and the new page_offset is calculated by masking out all bits except those used to get the page offset. This operation is performed by "size & $\sim$ PAGE_MASK". PAGE_MASK is 0x00000FFF, the least significant 12 bits are used as page offset, so PAGE_MASK is a value which can be

used to mask it. Using its inversion ∼ PAGE_MASK, will just get page offset which is equivalent to dividing the size by 4k and taking the remainder.

```
/*
* Reserve the area now:
*/

for (i = start; i < start+areasize; i++)
    if (test_and_set_bit(i, bdata->node_bootmem_map))
        BUG();
memset(ret, 0, size);
return ret;
```

Now that we have the memory, we need to reserve it. The macro `test_and_set_bit()` is used to test and set a bit to 1. It returns 0 if the previous value of the bit was 0 and 1, if it was 1. We put an assert condition to check for the highly impossible condition for it returning 1 (maybe bad RAM). We then initialize the memory to 0's and return it to the calling function.

1.7.6 Function free_all_bootmem()

File: mm/bootmem.c
Prototypes:

```
void free_all_bootmem (void);
void free_all_bootmem_core(pg_data_t *pgdat);
```

This function is used for freeing pages at boot and cleanup the bootmem allocator.

```
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);
```

Initialize idx to the number of low memory page frames in the node after the end of the kernel.

```
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);
    }
}
```

Go through the bootmem bitmap, find free pages and mark the corresponding entries in the mem_map as free. The function `set_page_count()` sets the count field of the page structure while `__free_page()` actually frees the page and modifies the buddy bitmap.

```
total += count;

/*
* Now free the allocator bitmap itself, it's not
* needed anymore:
*/
page = virt_to_page(bdata->node_bootmem_map);
count = 0;
for (i = 0; i < ((bdata->node_low_pfn-(bdata->node_boot_start
                >> PAGE_SHIFT))/8 + PAGE_SIZE-1)/PAGE_SIZE;
                                                i++,page++) {
    count++;
    ClearPageReserved(page);
    set_page_count(page, 1);
    __free_page(page);
}
```

Get the starting address of the bootmem, and free the pages containing it.

```
total += count;
bdata->node_bootmem_map = NULL;
return total;
```

Set the bootmem_map member of the node to NULL and return the total number of free pages.

1.8 Page Table Setup

1.8.1 Function paging_init()

File: arch/i386/mm/init.c
This function is called only once by `setup_arch()` to setup the page tables of the kernel. The description follows:

```
pagetable_init();
```

The above routine actually builds the kernel page tables. For more information refer section 1.8.2.

```
__asm__( "movl %%ecx,%%cr3\n" ::"c"(__pa(swapper_pg_dir)));
```

Since the page tables are now ready, load the address of *swapper_pg_dir* (contains the page directory of the kernel) into the *CR3* register.

```
#if CONFIG_X86_PAE
/*
* We will bail out later - printk doesnt work right now so
* the user would just see a hanging kernel.
*/
if (cpu_has_pae)
    set_in_cr4(X86_CR4_PAE);
#endif

__flush_tlb_all();
```

The above is a macro which invalidates the *Translation Lookaside Buffers*. TLB maintain a few of the recent virtual to physical address translations. Every time the page directory is changed, it needs to be flushed.

```
#ifdef CONFIG_HIGHMEM
kmap_init();
#endif
```

If CONFIG_HIGHMEM has been enabled, then structures used by kmap need to be initialized. Refer to section 1.8.4 for more information.

```
{
unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
unsigned int max_dma, high, low;

max_dma = virt_to_phys((char *)MAX_DMA_ADDRESS)
                                    >> PAGE_SHIFT;
```

Only memory below 16MB can be used for ISA DMA (*Direct Memory Access*) as the x86 ISA bus has only 24 address lines. In the above line, *max_dma* is used to store the page frame number of 16MB.

```
low = max_low_pfn;
high = highend_pfn;

if (low < max_dma)
    zones_size[ZONE_DMA] = low;
else {
    zones_size[ZONE_DMA] = max_dma;
    zones_size[ZONE_NORMAL] = low - max_dma;
#ifdef CONFIG_HIGHMEM
    zones_size[ZONE_HIGHMEM] = high - low;
#endif
}
```

The sizes for the three zones are calculated and stored in the array *zones_size*. The three zones are:

ZONE_DMA
: Memory from 0–16MB is allotted to this zone.

ZONE_NORMAL
: Memory above 16MB and less than 896MB is alloted to this zone.

ZONE_HIGHMEM
: Memory above 896MB is allotted to this zone.

More about zones in section 1.9 .

```
free_area_init(zones_size);
}

return;
```

The function free_area_init() is used to initialize the zone allocator. More information in section 1.9.2.

1.8.2 Function pagetable_init()

File: arch/i386/mm/init.c
This function actually builds the page tables in *swapper_pg_dir*, the kernel page directory. Description:

```
unsigned long vaddr, end;
pgd_t *pgd, *pgd_base;
int i, j, k;
pmd_t *pmd;
pte_t *pte, *pte_base;

/*
* This can be zero as well - no problem, in that case we exit
* the loops anyway due to the PTRS_PER_* conditions.
*/
end = (unsigned long)__va(max_low_pfn*PAGE_SIZE);
```

Calculate the virtual address of max_low_pfn and store it in *end*.

```
pgd_base = swapper_pg_dir;
```

Point pgd_base (page global directory base) to swapper_pg_dir.

```
#if CONFIG_X86_PAE
for (i = 0; i < PTRS_PER_PGD; i++)
    set_pgd(pgd_base + i, __pgd(1 + __pa(empty_zero_page)));
#endif
```

If PAE has been enabled, PTRS_PER_PGD[9] is 4.The variable swapper_pg_dir is used as a page-directory-pointer table and the *empty_zero_page* is used for this. The macro `set_pgd()` is defined in include/asm-i386/pgtable-3level.h.

[9]File: include/asm-i386/pgtable-3level.h

```
i = __pgd_offset(PAGE_OFFSET);
pgd = pgd_base + i;
```

The macro `__pgd_offset()` retrieves the corresponding index in a page directory of the given address. So __pgd_offset(PAGE_OFFSET) returns 0x300 (or 768 decimal), the index from where the kernel address space starts. Therefore *pgd* now points to the 768th entry.

```
for (; i < PTRS_PER_PGD; pgd++, i++) {
    vaddr = i*PGDIR_SIZE;
    if (end && (vaddr >= end))
        break;
```

PTRS_PER_PGD is 4 if CONFIG_X86_PAE is enabled, otherwise it is 1024, the number of entries in the table (page directory or page-directory-pointer table). We find the virtual address and use it to find whether we have reached the end. PGDIR_SIZE gives us the amount of RAM that can be mapped by a single page directory entry. It is 4MB or 1GB when CONFIG_X86_PAE is set.

```
#if CONFIG_X86_PAE
    pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
    set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
#else
    pmd = (pmd_t *)pgd;
#endif
```

If CONFIG_X86_PAE has been set, allocate a page (4k) of memory using the bootmem allocator to hold the page middle directory and set its address in the page global directory (AKA page-directory-pointer table), else there is no page middle directory, it directly maps onto the page directory (it is folded).

```
    if (pmd != pmd_offset(pgd, 0))
        BUG();
    for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
        vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
        if (end && (vaddr >= end))
            break;
        if (cpu_has_pse) {
            unsigned long __pe;
            set_in_cr4(X86_CR4_PSE);
```

```
            boot_cpu_data.wp_works_ok = 1;
            __pe = _KERNPG_TABLE + _PAGE_PSE + __pa(vaddr);
            /* Make it "global" too if supported */
            if (cpu_has_pge) {
                set_in_cr4(X86_CR4_PGE);
                __pe += _PAGE_GLOBAL;
            }
            set_pmd(pmd, __pmd(__pe));
            continue;
        }
```

Now starting to fill the page middle director (is page directory, without PAE). The virtual address is calculated. PMD_SIZE evaluates to 0 if PAE is not enabled. So vaddr = i * 4MB. Eg. The virtual address mapped by entry 0x300 is 0x300 * 4MB = 3GB. Next we check to see if PSE (*Page Size Extension*, is available on Pentium and above) is available. If it is, then we avoid using the page table and directly create 4MB pages. The macro `cpu_has_pse`[10] is used to find out if the processor has that feature and `set_in_cr4()` is used to enable it.

Processors starting from Pentium Pro, can have an additional attribute, the PGE (*Page Global Enable*). When a page is marked global and PGE is set, the page table or page directory entry for that page is not invalidated when a task switch occurs or when the cr3 is loaded. This will improve the performance and it is also one of the reasons for giving the kernel, all the address space above 3GB. After selecting all the attributes, the entry is set in the page middle directory.

```
        pte_base = pte = (pte_t *)
                  alloc_bootmem_low_pages(PAGE_SIZE);
```

This code is executed if PSE is not available. It allocates space for a page table (4k).

```
        for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
            vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
            if (end && (vaddr >= end))
                break;
```

There are 1024 entries in a page table (= 512, if PAE), each entry maps 4k (1 page).

[10] Defined in include/asm-i386/processor.h

```
            *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
        }
```

The macro `mk_pte_phys()` is used to create a page table entry from a physical address. The attribute PAGE_KERNEL is set to make it accessible in kernel mode only.

```
        set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte_base)));
        if (pte_base != pte_offset(pmd, 0))
            BUG();
    }
}
```

The page table is added to the page middle directory with the call to `set_pmd()`.This is continued in a loop till all the physical memory has been mapped starting from PAGE_OFFSET.

```
/*
* Fixed mappings, only the page table structure has to be
* created - mappings will be set by set_fixmap():
*/
vaddr = __fix_to_virt(__end_of_fixed_addresses - 1)
                                            & PMD_MASK;
fixrange_init(vaddr, 0, pgd_base);
```

There are some virtual addresses, in the very top most region of memory (4GB - 128MB), which are used directly in some parts of the kernel source. These mappings are specified in the file include/asm/fixmap.h. The enum *__end_of_fixed_addresses* is used as an index. The macro `__fix_to_virt()` returns a virtual address given the index (enum). More information in section 1.8.3.1. The function `fixrange_init()` creates the appropriate page table entries for those virtual addresses. Note: Only entries in the page table are created, no mappings are done. These addresses can later be mapped using the function `set_fixmap()`.

```
#if CONFIG_HIGHMEM
/*
* Permanent kmaps:
*/
vaddr = PKMAP_BASE;
fixrange_init(vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base);
```

```
pgd = swapper_pg_dir + __pgd_offset(vaddr);
pmd = pmd_offset(pgd, vaddr);
pte = pte_offset(pmd, vaddr);
pkmap_page_table = pte;
#endif
```

If CONFIG_HIGHMEM has been enabled, then we can access memory above 896MB by temporarily mapping it at the virtual addresses reserved for this purpose. The value of PKMAP_BASE is 0xFE000000 which is 4064MB (ie. 32MB below limit, 4GB) and that of LAST_PKMAP is 1024 (is 512 if PAE). So entries covering 4MB starting from 4064MB are created in the page table by fixrange_init(). Next, *pkmap_page_table* is assigned the page table entry covering the 4mb memory.

```
#if CONFIG_X86_PAE
/*
* Add low memory identity-mappings - SMP needs it when
* starting up on an AP from real-mode. In the non-PAE
* case we already have these mappings through head.S.
* All user-space mappings are explicitly cleared after
* SMP startup.
*/
pgd_base[0] = pgd_base[USER_PTRS_PER_PGD];
#endif
```

1.8.3 Fixmaps

File: include/asm-i386/fixmap.h

Fixmaps are compile time fixed virtual addresses which are used for some special purposes. These virtual addresses are mapped to physical pages at boot time using the macro `set_fixmap()`. These virtual addresses are allocated from the very top of address space (0xFFFFE000, 4GB - 8k) downwards. The fixed addresses can be calculated using the enum *fixed_addresses*.

```
enum fixed_addresses {
#ifdef CONFIG_X86_LOCAL_APIC
/* local (CPU) APIC) -- required for SMP or not */
    FIX_APIC_BASE,
#endif
#ifdef CONFIG_X86_IO_APIC
    FIX_IO_APIC_BASE_0,
```

```
    FIX_IO_APIC_BASE_END = FIX_IO_APIC_BASE_0 +
                                MAX_IO_APICS-1,
#endif
#ifdef CONFIG_X86_VISWS_APIC
    FIX_CO_CPU,     /* Cobalt timer */
    FIX_CO_APIC,    /* Cobalt APIC Redirection Table */
    FIX_LI_PCIA,    /* Lithium PCI Bridge A */
    FIX_LI_PCIB,    /* Lithium PCI Bridge B */
#endif
#ifdef CONFIG_HIGHMEM
/* reserved pte's for temporary kernel mappings*/
    FIX_KMAP_BEGIN,
    FIX_KMAP_END = FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1,
#endif
    __end_of_fixed_addresses
};
```

The above enums are used as an index to get the virtual address using the macro *__fix_to_virt()*. The other important defines are:

```
#define FIXADDR_TOP (0xffffe000UL)
#define FIXADDR_SIZE (__end_of_fixed_addresses << PAGE_SHIFT)
#define FIXADDR_START (FIXADDR_TOP - FIXADDR_SIZE)
```

FIXADDR_TOP
: The top of the fixed address mappings. It starts just below the end of memory (leaving 2 pages worth of address space) and grows down.

FIXADDR_SIZE
: It is used to calculate the number of pages required by fixmap. It depends on the value of *__end_of_fixed_addresses* which again depends on the various ifdef/endif combinations. Eg. if *__end_of_fixed_addresses* evaluated to 4, then FIXADDR_SIZE would return 4 * 4k = 16k. PAGE_SHIFT is 12, so left shifting is same as multiplying with 2^{12}.

FIXADDR_START
: It gives the starting address of the fixmapped addresses.

1.8.3.1 Macro __fix_to_virt()

File: include/asm-i386/fixmap.h
It is defined as:

```
#define __fix_to_virt(x)  (FIXADDR_TOP - ((x) << PAGE_SHIFT))
```

It takes one of the enums in *fixed_addresses* and calculates the corresponding virtual address. Eg. if FIX_KMAP_BEGIN was 3, then the address is calculated by multiplying it by 2^{12} and subtracting it from FIXADDR_TOP.

1.8.3.2 Function __set_fixmap()

File: include/asm-i386/fixmap.h
Prototype:

```
void __set_fixmap (enum fixed_addresses idx,
                   unsigned long phys,
                   pgprot_t flags);
```

This function is used to map physical addresses to the fixmapped virtual addresses. Its parameters are:

idx
: An index into the enum fixed_addresses, used to calculate the virtual address.

phys
: The physical address which has to be mapped to the fixmapped virtual address.

flags
: The various protection flags of the pages (attributes).

```
unsigned long address = __fix_to_virt(idx);
```

Get the virtual address we are trying to map.

```
if (idx >= __end_of_fixed_addresses) {
    printk("Invalid __set_fixmap\n");
    return;
}
```

Check if an invalid index was passed.

```
set_pte_phys(address, phys, flags);
```

Do the actual mapping.

1.8.3.3 Function fixrange_init()

File: arch/i386/mm/init.c
Prototype:

```
void fixrange_init (unsigned long start,
                    unsigned long end,
                    pgd_t *pgd_base);
```

This function is the one which actually creates the page table entries for the fixmapped addresses. The code is as follows:

```
pgd_t *pgd;
pmd_t *pmd;
pte_t *pte;
int i, j;
unsigned long vaddr;

vaddr = start;
i = __pgd_offset(vaddr);
j = __pmd_offset(vaddr);
pgd = pgd_base + i;
```

Initialize pgd to point to the page directory entry which covers vaddr.

```
for ( ; (i < PTRS_PER_PGD) && (vaddr != end); pgd++, i++) {
#if CONFIG_X86_PAE
    if (pgd_none(*pgd)) {
        pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
        if (pmd != pmd_offset(pgd, 0))
            printk("PAE BUG #02!\n");
    }
    pmd = pmd_offset(pgd, vaddr);
#else
    pmd = (pmd_t *)pgd;
#endif
```

If PAE has been enabled, we need to create an additional page middle directory, otherwise we just fold it into page directory itself.

```
    for (; (j < PTRS_PER_PMD) && (vaddr != end); pmd++, j++) {
        if (pmd_none(*pmd)) {
```

```
            pte = (pte_t *)alloc_bootmem_low_pages(PAGE_SIZE);
            set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte)));
            if (pte != pte_offset(pmd, 0))
                BUG();
        }
        vaddr += PMD_SIZE;
    }
    j = 0;
}
```

Next we create the page tables and create entries for them in the page middle directory.

1.8.4 Function kmap_init()

File: arch/i386/mm/init.c
This function is just used to store the page table entry and the protection flags in *kmap_pte* and *kmap_prot* respectively (This is what the comment means by "cache it").

```
unsigned long kmap_vstart;

/* cache the first kmap pte */
kmap_vstart = __fix_to_virt(FIX_KMAP_BEGIN);
kmap_pte = kmap_get_fixmap_pte(kmap_vstart);

kmap_prot = PAGE_KERNEL;
```

The macro `kmap_get_fixmap_pte()` is used to get the page table entry for the given entry.

1.9 Memory Zones

Physical[11] memory has been divided into different zones to differentiate between intended uses, and are generally used to model different characteristics of the memory. Eg. on the x86, there is only 16MB of ISA DMA-able memory, so zone allocator will try to save DMA pages for processes specifically requesting ZONE_DMA. The available zones are:

ZONE_DMA
ISA DMA capable memory.(<16MB, directly mapped by the kernel)

[11] This explanation is from the FAQ on #kernelnewbies, thanks to the contributor.

ZONE_NORMAL
Memory which is directy mapped by the kernel (> 16MB and < 896MB).

ZONE_HIGHMEM
Memory which is not directly mapped by the kernel (> 896MB).

1.9.1 Structures

1.9.1.1 struct zone_struct

File: include/linux/mmzone.h
Each zone is represented by a struct *zone_struct*.

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

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

    wait_queue_head_t     * wait_table;
    unsigned long           wait_table_size;
    unsigned long           wait_table_shift;


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

/*
```

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

The description of the members of struct zone_struct:

lock
It is used for serialization of access to the other members of this structure.

free_pages
The number of free pages present in the zone.

pages_min
When the number of free pages in the zone reaches this number, only the kernel can allocate more memory.

pages_low
If the number of free pages gets below this point,the kernel starts swapping aggressively.

pages_high
The kernel tries to keep up to this amount of memory free; if memory comes below this point, the kernel gently starts swapping in the hopes that it never has to do real aggressive swapping.

need_balance
A flag kswapd uses to determine if it needs to balance.

free_area
Array of bitmaps and lists of pages used in buddy allocator.

wait_table
The array holding the hash table. The purpose of this table is to keep track of the processes waiting for a page to become available and make them runnable again when possible.

wait_table_size
The size of the hash table array.

wait_table_shift
Used to hold the no. of left shifts (1 <<) to get the table size.

zone_pgdat
The node in which the zone is.

zone_mem_map
The memory map of this zone.

zone_start_paddr
The starting physical address of the zone.

zone_start_mapnr
The index into mem_map.

name
The name of the zone.

size
The total size of physical memory in the zone.

1.9.1.2 struct page

File: include/linux/mm.h
Also each physical page of memory (or page frame) has an associated *struct page* which contains all the information needed to manage them.

```
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    struct page **pprev_hash;
    struct buffer_head * buffers;
    void *virtual;
} mem_map_t;
```

list
This is used to point to the next page in any list.

mapping
Used to specify the inode we are mapping.

index
Our offset within mapping.

next_hash
Points to the next page sharing the hash bucket in the pagecache hash table.

count
Number of references to this page (usage count).

flags
Different attributes of the page.

lru
Used to point to the head of the lru list the page is in (active_list, inactive_list).

pprev_hash
Complement to next_hash.

buffers
If this page is being used to hold buffers (buffered disk blocks), points to the first buffer head.

virtual
When highmem memory is mapped into the kernel's virtual address space, this variable is used to store the virtual address of this page.

1.9.2 Function free_area_init()

File: mm/page_alloc.c
Prototypes:

```
void free_area_init(unsigned long *zones_size);
void free_area_init_core(int nid, pg_data_t *pgdat,
                         struct page **gmap,
                         unsigned long *zones_size,
                         unsigned long zone_start_paddr,
                         unsigned long *zholes_size,
                         struct page *lmem_map);
```

This function is used to initialize the memory zones and create the memory map.

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

Alignment stuff, not yet clear to me though

```
if (zone_start_paddr & ~PAGE_MASK)
    BUG();
```

Check if the zone is starting on a page boundary.

```
totalpages = 0;
for (i = 0; i < MAX_NR_ZONES; i++) {
    unsigned long size = zones_size[i];
    totalpages += size;
}
```

Calculate the total number of pages in the node.

```
realtotalpages = totalpages;
if (zholes_size)
    for (i = 0; i < MAX_NR_ZONES; i++)
        realtotalpages -= zholes_size[i];
printk("On node %d totalpages: %lu\n", nid, realtotalpages);
```

Print the number of pages found.

```
INIT_LIST_HEAD(&active_list);
INIT_LIST_HEAD(&inactive_list);
```

Initialize the LRU lists (circular linked lists).

```
/*
* Some architectures (with lots of mem and discontinous memory
* maps) have to search for a good mem_map area:
* For discontigmem, the conceptual mem map array starts from
```

```
* PAGE_OFFSET, we need to align the actual array onto a
* mem map boundary, so that MAP_NR works.
*/
map_size = (totalpages + 1)*sizeof(struct page);
if (lmem_map == (struct page *)0) {
    lmem_map = (struct page *)
                  alloc_bootmem_node(pgdat, map_size);
    lmem_map = (struct page *)(PAGE_OFFSET +
                MAP_ALIGN((unsigned long)lmem_map
                                  - PAGE_OFFSET));
}
```

Allocate space for the local memory map (array of struct page, each struct represents one physical page, more below) and allign it.

```
*gmap = pgdat->node_mem_map = lmem_map;
pgdat->node_size = totalpages;
pgdat->node_start_paddr = zone_start_paddr;
pgdat->node_start_mapnr = (lmem_map - mem_map);
pgdat->nr_zones = 0;
```

Initialize the members of the node.

```
offset = lmem_map - mem_map;
```

The variable *mem_map* is a global sparse array of *struct page*s, each structure representing one physical page. The starting index of mem_map depends on the first zone of the first node, if it is zero, the index starts from zero else the corresponding page frame number. Each zone has its own map stored in *zone_mem_map* which is mapped into the containing node's *node_mem_map* which is in turn is part of the global mem_map.

In the above line of code, offset represents the node's memory map entry point (index) into the global mem_map. Here, it is zero as the page frame number starts from zero on the i386.

```
for (j = 0; j < MAX_NR_ZONES; j++) {
```

This loop is used to initialize the members of the zones.

```
    zone_t *zone = pgdat->node_zones + j;
    unsigned long mask;
    unsigned long size, realsize;
```

```
zone_table[nid * MAX_NR_ZONES + j] = zone;

realsize = size = zones_size[j];
```

The actual zone data is stored in the node, so take a pointer to the correct zone and get its size. Also initialize the *zone_table* entries at the same time.

```
if (zholes_size)
    realsize -= zholes_size[j];

printk("zone(%lu): %lu pages.\n", j, size);
```

Correct for any holes and print out the zone sizes. Sample output:

```
zone(0): 4096 pages.
zone(1): 45056 pages.
zone(2): 0 pages.
```

Here zone 2 is 0 as I have only 192mb of RAM in my system.

```
zone->size = size;
zone->name = zone_names[j];
zone->lock = SPIN_LOCK_UNLOCKED;
zone->zone_pgdat = pgdat;
zone->free_pages = 0;
zone->need_balance = 0;
```

Initialize the member elements.

```
if (!size)
    continue;
```

If the size of a zone is zero like my zone 2 (HIGH_MEM), no need for further initializations.

```
zone->wait_table_size = wait_table_size(size);
zone->wait_table_shift =
  BITS_PER_LONG - wait_table_bits(zone->wait_table_size);
zone->wait_table =
  (wait_queue_head_t *) alloc_bootmem_node
  (pgdat, zone->wait_table_size * sizeof(wait_queue_head_t));
for(i = 0; i < zone->wait_table_size; ++i)
    init_waitqueue_head(zone->wait_table + i);
```

Initialize the wait queues.

```
    pgdat->nr_zones = j+1;
    mask = (realsize / zone_balance_ratio[j]);
    if (mask < zone_balance_min[j])
        mask = zone_balance_min[j];
    else if (mask > zone_balance_max[j])
        mask = zone_balance_max[j];
```

Calculate the appropriate balance ratio.

```
    zone->pages_min = mask;
    zone->pages_low = mask*2;
    zone->pages_high = mask*3;
    zone->zone_mem_map = mem_map + offset;
    zone->zone_start_mapnr = offset;
    zone->zone_start_paddr = zone_start_paddr;
```

Set the watermarks and initialize zone_mem_map with the correct pointer into the global mem_map. The variable zone_start_mapnr is initialized with the index into the global mem_map.

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

/*
* Initially all pages are reserved - free ones are freed
* up by free_all_bootmem() once the early boot process is
* done. Non-atomic initialization, single-pass.
*/
    for (i = 0; i < size; i++) {
        struct page *page = mem_map + offset + i;
        set_page_zone(page, nid * MAX_NR_ZONES + j);
        set_page_count(page, 0);
        SetPageReserved(page);
        memlist_init(&page->list);
        if (j != ZONE_HIGHMEM)
            set_page_address(page, __va(zone_start_paddr));
        zone_start_paddr += PAGE_SIZE;
    }
```

Set the zone in which the page lies as one of the page's attributes in the flag. Also make the count of the page as zero and mark it as reserved (it will be un-reserved again in mem_init()). Initialize the list member of the page and also set the virtual address of the page in the *virtual* member of `struct page`.

```
    offset += size;
```

Increment the offset by size to point to the starting index of the next zone in mem_map.

```
    for (i = 0; ; i++) {
        unsigned long bitmap_size;
        memlist_init(&zone->free_area[i].free_list);
        if (i == MAX_ORDER-1) {
            zone->free_area[i].map = NULL;
            break;
        }
```

Initialize the linked list free_area[].free_list (more information in section 2.2) and the bitmap of the last order to NULL.

```
/*
* Page buddy system uses "index >> (i+1)",
* where "index" is at most "size-1".
*
* The extra "+3" is to round down to byte
* size (8 bits per byte assumption). Thus
* we get "(size-1) >> (i+4)" as the last byte
* we can access.
*
* The "+1" is because we want to round the
* byte allocation up rather than down. So
* we should have had a "+7" before we shifted
* down by three. Also, we have to add one as
* we actually _use_ the last bit (it's [0,n]
* inclusive, not [0,n[).
*
* So we actually had +7+1 before we shift
* down by 3. But (n+8) >> 3 == (n >> 3) + 1
* (modulo overflows, which we do not have).
```

```
*
* Finally, we LONG_ALIGN because all bitmap
* operations are on longs.
*/
        bitmap_size = (size-1) >> (i+4);
        bitmap_size = LONG_ALIGN(bitmap_size+1);
        zone->free_area[i].map = (unsigned long *)
                    alloc_bootmem_node(pgdat, bitmap_size);
    }
```

The size of the bitmap is calculated. It is then allocated using the bootmem allocator.

```
}

build_zonelists(pgdat);
```

Create the different zonelists in the node. These zonelists are used in allocation purposes to specify the order (priority, preference) of the zones in which to query for a free page.

1.9.3 Function build_zonelists()

File: mm/page_alloc.c

```
int i, j, k;

for (i = 0; i <= GFP_ZONEMASK; i++) {
    zonelist_t *zonelist;
    zone_t *zone;
    zonelist = pgdat->node_zonelists + i;
    memset(zonelist, 0, sizeof(*zonelist));
```

Get the pointer to the zonelist member of the node and initialize it with null pointers.

```
    j = 0;
    k = ZONE_NORMAL;
    if (i & __GFP_HIGHMEM)
        k = ZONE_HIGHMEM;
    if (i & __GFP_DMA)
        k = ZONE_DMA;
```

Compare the current mask with the three available and use it for the switch statement below.

```
    switch (k) {
        default:
            BUG();
    /*
    * fallthrough:
    */
        case ZONE_HIGHMEM:
            zone = pgdat->node_zones + ZONE_HIGHMEM;
            if (zone->size) {
            #ifndef CONFIG_HIGHMEM
                BUG();
            #endif
                zonelist->zones[j++] = zone;
            }
        case ZONE_NORMAL:
            zone = pgdat->node_zones + ZONE_NORMAL;
            if (zone->size)
                zonelist->zones[j++] = zone;
        case ZONE_DMA:
            zone = pgdat->node_zones + ZONE_DMA;
            if (zone->size)
                zonelist->zones[j++] = zone;
    }
```

The given mask specifies the order of preferrence,so we use it to find the entry point into the switch statement and just fall through it. So, if the mask was __GFP_DMA, the zonelist will contain only the DMA zone, if it was __GFP_HIGHMEM, it would have ZONE_HIGHMEM, ZONE_NORMAL and ZONE_DMA in that order.

```
    zonelist->zones[j++] = NULL;
}
```

Null terminate the list.

1.9.4 Function mem_init()

File: arch/i386/mm/init.c
This function is called by start_kernel to further initialize the zone allocator.

```
int codesize, reservedpages, datasize, initsize;
int tmp;
int bad_ppro;

if (!mem_map)
    BUG();

#ifdef CONFIG_HIGHMEM
highmem_start_page = mem_map + highstart_pfn;
max_mapnr = num_physpages = highend_pfn;
num_mappedpages = max_low_pfn;
```

If CONFIG_HIGHMEM is set then get the starting address of HIGHMEM and the total number of pages.

```
#else
max_mapnr = num_mappedpages = num_physpages = max_low_pfn;
#endif
```

Else the number of pages is just the number of normal memory pages.

```
high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);
```

Get the virtual address of the the last page of low memory.

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

/* this will put all low memory onto the freelists */
totalram_pages += free_all_bootmem();
reservedpages = 0;
```

The function free_all_bootmem() essentially frees all low memory and after this point bootmem allocator is no longer usable. Refer to section 1.7.6 for more information on this function.

```
/*
* Only count reserved RAM pages
*/
for (tmp = 0; tmp < max_low_pfn; tmp++)
    if (page_is_ram(tmp) && PageReserved(mem_map+tmp))
        reservedpages++;
```

Go through the mem_map and count reserved pages.

```
#ifdef CONFIG_HIGHMEM
for (tmp = highstart_pfn; tmp < highend_pfn; tmp++) {
    struct page *page = mem_map + tmp;
    if (!page_is_ram(tmp)) {
        SetPageReserved(page);
        continue;
    }
    if (bad_ppro && page_kills_ppro(tmp)) {
        SetPageReserved(page);
        continue;
    }
    ClearPageReserved(page);
    set_bit(PG_highmem, &page->flags);
    atomic_set(&page->count, 1);
    __free_page(page);
    totalhigh_pages++;
}

totalram_pages += totalhigh_pages;
#endif
```

Go through high memory and reserve pages which are not usable else mark the as PG_highmem and call __free_page() which frees it and modifies the buddy bitmap (refer section 2.2.2).

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

```
        (unsigned long)(totalhigh_pages << (PAGE_SHIFT-10)));
```

Calculate the sizes of various sections of the kernel and print out the statistics.

1.10 Initialization of Slab Allocator

1.10.1 Function kmem_cache_init()

File: mm/slab.c
This function is used to initialize the slab allocator.

```
size_t left_over;

init_MUTEX(&cache_chain_sem);
INIT_LIST_HEAD(&cache_chain);
```

Initialize the semaphore serializing access to the cache chain and also initialize the cache chain (circular linked list) itself.

```
kmem_cache_estimate(0, cache_cache.objsize, 0,
                    &left_over, &cache_cache.num);
```

The above function initializes the *cache_cache*. It calculates the number of objects that can be held on a single slab and the space that will be left (wasted, used for coloring). The variable *cache_cache* is used to cache other cache entries.

```
if (!cache_cache.num)
    BUG();
cache_cache.colour = left_over/cache_cache.colour_off;
```

The members of the above structure type will be covered in more detail in a later chapter but the brief explanation for *colour* is that it is used to store the coloring range. The var *cache_cache.colour_off* has been statically initialised to 32 bytes, the size of the cache line of L1 cache available on i386. So the above statement basically calculates the colour range available for this cache. Eg. if only 20 bytes were left, then only 0–19 can be used for colouring. The concept of colouring will be explained along with the slab allocator.

```
cache_cache.colour_next = 0;
```

Set the colour of the cache. Since this is the first cache, it has been set to 0.

1.10.2 Function kmem_cache_sizes_init()

File: mm/slab.c
This function is also called from `start_kernel()` to setup the general caches. Caches of sizes 32 bytes to 128k are created of both DMA and non-DMA memory.

```
cache_sizes_t *sizes = cache_sizes;
char name[20];
```

The variable *cache_sizes* is a statically allocated structure containing all the sizes filled in and the pointers to the actual caches initialized to NULL which are initialized by this function.

```
/*
* Fragmentation resistance on low memory - only use bigger
* page orders on machines with more than 32MB of memory.
*/
if (num_physpages > (32 << 20) >> PAGE_SHIFT)
    slab_break_gfp_order = BREAK_GFP_ORDER_HI;
```

If more than 32mb is available, then higher order pages (2^2) can be used for the slabs else it is only 2^1. This variable is used in `kmem_cache_create()`.

```
do {

/* For performance, all the general caches are L1 aligned.
* This should be particularly beneficial on SMP boxes, as it
* eliminates "false sharing".
* Note for systems short on memory removing the alignment will
* allow tighter packing of the smaller caches. */
    sprintf(name,"size-%Zd",sizes->cs_size);
    if (!(sizes->cs_cachep = kmem_cache_create(name,
                                    sizes->cs_size,0,
                                    SLAB_HWCACHE_ALIGN,
                                    NULL, NULL))) {
        BUG();
    }
```

Create the cache with hardware alignment and 0 offset. The *name* member is used to display information in *slabinfo* (cat /proc/slabinfo).

```
/* Inc off-slab bufctl limit until the ceiling is hit. */
    if (!(OFF_SLAB(sizes->cs_cachep))) {
        offslab_limit = sizes->cs_size-sizeof(slab_t);
        offslab_limit /= 2;
    }
```

Try to make it an off-slab, more details later when i get it in my head (:-).

```
    sprintf(name, "size-%Zd(DMA)",sizes->cs_size);
    sizes->cs_dmacachep = kmem_cache_create(name,
                                sizes->cs_size, 0,
                                SLAB_CACHE_DMA|
                                SLAB_HWCACHE_ALIGN,
                                NULL, NULL);
    if (!sizes->cs_dmacachep)
        BUG();
    sizes++;
```

Create the DMA cache with hardware alignment and 0 offset. Then increment the size for the next round.

```
} while (sizes->cs_size);
```

Chapter 2

Physical Memory Allocation

2.1 Zone Allocator

As previously mentioned, memory has been divided into different zones. From these zones, memory is allocated and de-allocated by the zone allocator using the buddy system algorithm.

2.2 Buddy System

The buddy system is a conceptually simple memory allocation algorithm. Its main use is to reduce external fragmentation as much as possible and at the same time allow fast allocation and de-allocation of pages. To reduce external fragmentation, free contiguous memory pages are grouped into lists of different sizes (or orders). This allows all 2 page sized blocks to be on one list, 4 page blocks on another and so on. If a requirement comes for 4 contiguous pages, the request can be quickly satisfied by checking to see if there are any free 4 page blocks. If available, it is used to satisfy the request else the next order (size) is tried. So if an 8 page block is available, it is split into 2 4-page blocks and one is returned to the requester while the other is added to the 4 block list. This avoids splitting large contiguous free page blocks when a request can be satisfied by a smaller block thus reducing external fragmentation. Also the physical address of the first page frame needs to be a multiple of the block size, ie a block of size 2^n has to be aligned with 4k * 2^n.

Conversely, when a page block of a certain order is being freed, attempt is made to merge it with its adjacent block (buddy) of the same order if it is already free, to get a free block of an higher order. This is done recursively until a bigger merge is not possible. This free block is then added to the

appropriate high order free list. This is also known as *Coalescence*.

2.2.0.1 struct free_area_struct

File: include/linux/mmzone.h
Linux uses lists of 1,2,4,8,16,32,64,128,256 and 512 page blocks. To manage these lists and implement the buddy system it uses the structure `free_area_struct` (a.k.a free_area_t).

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

The fields of the above structure are used as follows:

free_list
: Its a doubly linked list of free page blocks of a certain size. It points to the first and last page blocks, while the *list* member of `struct page` is used to link up the pages in between.

map
: Also known as the buddy bitmap, it is contains information about the availability of a buddy. Its size is calculated using the formula:

$$((\text{number of pages}) - 1 >> (\text{order} + 4)) + 1 \text{ bytes}$$

Each bit represents two adjacent blocks of the same size. Its value is 0 if both the blocks are either partially or fully used (busy) or completely free. It is 1 if exactly one of the blocks is completely free and the other is (partially or fully) used.

Each zone has an array of these structure, one for each size.

2.2.1 Example

Let us assume that we have a system with only 16 pages of RAM as shown in figure 2.1. Since there are only 16 pages of RAM, we will only have buddy bitmaps for four orders. They will be as follows:

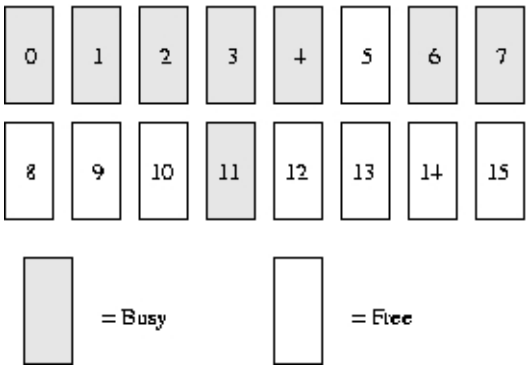

Figure 2.1: Example

```
pages:    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
order(0):  0   0   1   0   0    1     0     0
order(1):    0       0       1          0
order(2):        0                  1
order(3):                   0
```

In order(0), the first bit represents the first 2 pages, second bit the other 2 and so on. The 3rd bit is 1 as page 4 is busy while page 5 is free. Also, in order(1), bit 3 is 1 because one buddy is completely free (pages 8 and 9) and the other buddy (pages 10 and 11) is not, so there is a possibility of a merge.

2.2.1.1 Allocation

Following are the steps performed, if we want a free page block of order(1).

1. Initially the free lists will be:

   ```
   order(0):  5, 10
   order(1):  8 [8,9]
   order(2):  12 [12,13,14,15]
   order(3):
   ```

2. Since the order(1) list contains one free page block, it is returned to the user and removed from the list.

3. If we need another order(1) block, we again scan the free lists starting from the order(1) free list.

4. Since there in no free block available, we go to the next higher order, order(2).

5. Here there is one free page block, starting from page 12. This block is now made into two smaller order(1) blocks, [12,13] and [14,15]. The block starting [14,15] is added to the order(1) free list and the first block [12,13] is returned to the user.

6. Finally the free lists will be:

```
order(0):  5, 10
order(1):  14 [14,15]
order(2):
order(3):
```

2.2.1.2 De-Allocation

Taking the same example, following are the steps performed, if we are freeing page 11 (order 0).

1. Find the bit representing page 11 in the buddy bitmap of order(0) using the formula:

```
index = page_idx >> (order + 1)
      = 11 >> (0 + 1)
      = 5
```

2. Then we check the value of that bit. If it is 1, there is a free buddy adjacent to us. Bit 5 is 1, as its buddy page 10 is free.

3. So we now reset the bit to 0, as both the buddies are now completely free.

4. We remove page 10 from the free list of order(0).

5. We start all over again, with 2 free pages (10 and 11, order(1)).

6. The start of this new free page block is 10, so find its index in the buddy bitmap of order(1). Using the above formula, we get it as bit 2 (3rd bit).

7. Bit 2 (in order(1) bitmap) is again 1 as the buddy of the page block being freed consisting of pages 8 and 9 is free.

8. Reset bit 2 and remove page block of size 2 starting with page 8 from the free list of order(1).

9. We go up another order. We now have 4 contiguous free pages starting from page 8. We find its bit index in the order(2) buddy bitmap. It is bit 1 whose value is 1, signifying another merge.

10. Page block starting from page 12 of size 4 is removed from the free list of order(2) and merged with our page block. So now we have 8 free contiguous page starting from page 8.

11. We go another order up, to order(3). Its bit index is 0, whose value is also 0. Which means that the other buddy is not completely free. Since no merge is possible, we just set the bit as 1 and add the free page blocks to order(3)'s free list.

12. So finally we have 8 contiguous free blocks and the buddy bitmap looks like this:

```
pages:    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
order(0):  0   0   1   0   0    0     0     0
order(1):    0       0       0          0
order(2):        0                  0
order(3):                   1
```

2.2.2 Function __free_pages_ok()

File: mm/page_alloc.c
Prototype:

```
void __free_pages_ok (struct page *page,
                      unsigned int order);
```

This is the function which is used to free the pages when they are no longer required. Pages can be freed in blocks of specific orders (2^0,...,2^9) only when they are block aligned, ie. if you are trying to free a 16 page block, it needs to be on a 16 page boundary.

```
unsigned long index, page_idx, mask, flags;
free_area_t *area;
struct page *base;
zone_t *zone;

if (PageLRU(page))
    lru_cache_del(page);
```

Check to see if the page to be freed is in any of the lru lists, if it is, remove it from there.

```
if (page->buffers)
    BUG();
if (page->mapping)
    BUG();
if (!VALID_PAGE(page))
    BUG();
if (PageSwapCache(page))
    BUG();
if (PageLocked(page))
    BUG();
if (PageLRU(page))
    BUG();
if (PageActive(page))
    BUG();
```

The above assert conditions check for the following:

1. The page is not being used for storing any buffers.
2. Its not part of any fs mapping.
3. The page is valid.
4. The page is not in the swap cache.
5. The page has not been locked by any process.
6. The page is not on the LRU list (dead code, as already been done above).

7. The page is not on the active list.

```
page->flags &= ~((1<<PG_referenced) | (1<<PG_dirty));
```

Reset the referenced and dirty bits to 0.

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

When a process frees up the pages it needs, instead of freeing them back proper, it frees after setting the (task_struct) flag to `PF_FREE_PAGES`, so that the pages are added to the process's *local_freelist*. The above code checks for this condition and makes a jump to the correct code.

```
back_local_freelist:

zone = page_zone(page);
```

Get the zone the page is in.

```
mask = (~0UL) << order;
```

Create a mask for the order of pages being freed.

```
base = zone->zone_mem_map;
```

base is the first page in the current zone.

```
page_idx = page - base;
```

The index of the first page frame (if more than one page is being freed) inside the zone_mem_map.

```
if (page_idx & ~mask)
    BUG();
```

If the page is not aligned to the order size, it's a bug.

```
index = page_idx >> (1 + order);
```

index is the bit position of the page being freed in the buddy bitmap.

```
area = zone->free_area + order;
```

The free_area_t struct of the correct order is stored in area. This structure contains the free list of pages of the given order and also a corresponding buddy bitmap.

```
spin_lock_irqsave(&zone->lock, flags);
zone->free_pages -= mask;
```

Lock the zone and increment the no. of free pages (-mask == no. of pages being freed).

```
while (mask + (1 << (MAX_ORDER-1))) {
```

The value of MAX_ORDER is 10, where as the value of mask varies from:

```
                    mask (decimal)
order(0):  11111111111111111111111111111111
order(9):  11111111111111111111000000000
```

```
(1 << (MAX_ORDER-1)) = 00000000000000000000001000000000
```

When mask reaches its upper limit, the sum becomes 0 thus quitting the while loop.

```
    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.
*/
        break;
```

If the buddy of the page block being freed is already free (ie. a merge can be made), the corresponding bit in the free_area bitmap is 1, else if it is busy (not free), it is 0. So the above code checks if the bit is 0 or 1 and then toggles it. If it was 0, meaning the buddy is not free, `__test_and_change_bit` returns 0 after setting it to 1. Since we are using ! here, it evaluates to 1 and the code breaks the while loop and we cannot merger any further. We set it to 1 so that the next time we are here (adjacent block being freed), we can merge both of the blocks.

```
/*
* Move the buddy up one level.
*/
    buddy1 = base + (page_idx ^ -mask);
```

This statement is used to get an handle (pointer) to the structure (struct page) representing the first page of buddy of the block of pages being freed. Now, the block of pages being freed can be either in front of its buddy or follow its buddy. In other words, to get the pointer to the buddy, we may have to add the number of pages or subtract them. Lets take an example, if we are freeing page 5 (order 0), then its buddy is page 4 and vice versa. Thats why we use the *exclusive OR* operator here. Here, -mask is equivalent to the number of pages being freed.

To see how this works, we will take the same example mentioned above. we are freeing page 4, so the equation will look like:

```
buddy1 = 0 + (4 ^ 1);
4 ^ 1 == 00000100 ^ 00000001 = 00000101 = 5
```

Similarly if we were freeing page 5:

```
buddy1 = 0 + (5 ^ 1);
5 ^ 1 == 00000101 ^ 00000001 = 00000100 = 4

    buddy2 = base + page_idx;
```

This is pretty straight forward. Get the pointer to the structure of the first page of the block being freed.

```
    if (BAD_RANGE(zone,buddy1))
        BUG();
    if (BAD_RANGE(zone,buddy2))
        BUG();
    memlist_del(&buddy1->list);
```

Since buddy1 can be merged with buddy2, remove buddy1 from the free list its currently in. It can then be paired with the block being freed and added to the free list of an higher order.

```
    mask <<= 1;
    area++;
    index >>= 1;
    page_idx &= mask;
}
```

Update mask so that we can try to merge blocks of higher order. The operation "mask <<= 1" increases the order and with that the number of pages it is trying to merge (remember -mask == no. of pages it is trying to free). Also make `area` point to the `free_area_t` structure of the next order. Divide index by 2 to get the new bit position of the buddy blocks in the higher order bitmap. Also modify page_idx to make sure it points to the first buddy.

```
memlist_add_head(&(base + page_idx)->list, &area->free_list);
spin_unlock_irqrestore(&zone->lock, flags);
return;
```

We cannot merge anymore buddies so we just add it to the free list of the current order.

```
local_freelist:

if (current->nr_local_pages)
    goto back_local_freelist;
```

If the process has already freed pages for itself, don't give it more.

```
if (in_interrupt())
    goto back_local_freelist;
```

An interrupt doesn't have a current process to store pages on.

```
list_add(&page->list, &current->local_pages);
page->index = order;
current->nr_local_pages++;
```

Add the page onto the local list, update the page information and return.

2.2.3 Function __alloc_pages()

File: mm/page_alloc.c
Prototype:

```
struct page * __alloc_pages(unsigned int gfp_mask,
                            unsigned int order,
                            zonelist_t *zonelist)
```

This function is used to allocate free pages and is the heart of the zoned buddy allocator.

```
unsigned long min;
zone_t **zone, * classzone;
struct page * page;
int freed;

zone = zonelist->zones;
classzone = *zone;
```

The zonelist is an array of zones which is used to specify the preferred order for getting memory. The first zone is the most preferred zone, so save a reference to it in *classzone*.

```
min = 1UL << order;
```

Get the number of pages being requested.

```
for (;;) {
    zone_t *z = *(zone++);
```

Loop through each zone to find free pages.

```
    if (!z)
        break;

    min += z->pages_low;
```

If we come to the end of the zonelist, break. Each zone needs to have atleast `pages_low` number of pages free at any time. So to satisfy our request, it needs *pages_low* number of pages + the number of pages being requested.

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

If the number of free pages in the zone is more than our requirement, the function `rmqueue()` is used to allocate the pages and return. Refer section 2.2.4 for more details on `rmqueue()`.

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

The *pages_low* marker has been reached, so mark the zone as needing balancing and wake up kswapd which will start freeing pages in this zone.

```
zone = zonelist->zones;
min = 1UL << order;

for (;;) {
```

```
    unsigned long local_min;
    zone_t *z = *(zone++);
    if (!z)
        break;
    local_min = z->pages_min;
```

Start moving through the zones again. This time we ignore the `pages_low` water-mark hoping that *kswapd* will do its job. We still have to consider the second low water-mark, ie. `pages_min`. If we go below it, then we need to start recovering the pages ourself (instead of kswapd).

```
    if (!(gfp_mask & __GFP_WAIT))
        local_min >>= 2;
    min += local_min;
```

If the process cannot wait, we get ourself into a more tight position by decreasing the second water-mark (= dividing by 4). We then add the number of pages required to it.

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

If the required pages are available, we allocate them.

```
/* here we're in the low on memory slow path */
rebalance:
if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {
```

PF_MEMALLOC is set if the calling process wants to be treated as a memory allocator, kswapd for example. This process is high priority and should be served if at all possible. PF_MEMDIE is set by the OOM killer. The calling process is going to die no matter what but needs a bit of memory to die cleanly, hence give what it needs because we'll get it back soon.

```
    zone = zonelist->zones;
    for (;;) {
        zone_t *z = *(zone++);
        if (!z)
            break;

page = rmqueue(z, order);
        if (page)
            return page;
    }
    return NULL;
}
```

Here we don't check any water-marks or limits, we just try to give the memory if its possible.

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

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

We don't have any pages, so if the process cannot wait, just return NULL. If it can wait, then we try to balance the zone (ie. try to free pages). More about *balance_classzone()* in section 2.2.6.

```
if (page)
    return page;
```

If `balance_classzone()` was successful in freeing pages, return them.

```
zone = zonelist->zones;
min = 1UL << order;

for (;;) {
    zone_t *z = *(zone++);
    if (!z)
        break;
```

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

We go through the zones one last time looking for free pages.

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

If it was a big request, dump it.

```
/* Yield for kswapd, and try again */
current->policy |= SCHED_YIELD;
__set_current_state(TASK_RUNNING);
schedule();
goto rebalance;
```

Since the process can wait, set `SCHED_YIELD` and yield the CPU for one reschedule. Then try to rebalance.

2.2.4 Function rmqueue()

File: mm/page_alloc.c
Prototype:

```
struct page * rmqueue(zone_t *zone, unsigned int order)
```

This function is responsible for finding out what order of pages we have to go to, to satisfy the request. For example if there is no page block free to satisfy the order=0 (1 page) request, then see if there is a free block of order=1 that can be split into two order=0 pages.

```
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 = memlist_next(head);
```

Lock the zone. Make `head` point to the head of the free list. Then `memlist_next (head)` will point `curr` to the *list* member of the first page in the list else if the free list is empty, it will point to `head` itself.

```
    if (curr != head) {
        unsigned int index;
        page = memlist_entry(curr, struct page, list);
```

Check if the list is empty, if not get the reference to the first page. The macro `memlist_entry` is just an alias for `list_entry`.

```
if (BAD_RANGE(zone,page))
            BUG();

memlist_del(curr);
        index = page - zone->zone_mem_map;
```

Since we found a free page block, remove it from the current free list and get the page index.

```
if (curr_order != MAX_ORDER-1)
            MARK_USED(index, curr_order, area);
```

If the current order is of the maximum order (i.e. 9), then there is no buddy bitmap for it, else toggle the appropriate bit in the buddy bitmap.

```
zone->free_pages -= 1UL << order;
        page = expand(zone, page, index, order,
                                    curr_order, area);
```

Subtract the number of pages being allocated from the free_pages count and call `expand()` to distribute the excess pages into different free lists. More on this function below in section 2.2.5.

```
        spin_unlock_irqrestore(&zone->lock, flags);
        set_page_count(page, 1);
```

Unlock the zone and set the page count of the page to 1 thereby increasing the reference count.

```
        if (BAD_RANGE(zone,page))
            BUG();
        if (PageLRU(page))
            BUG();
        if (PageActive(page))
            BUG();

        return page;
    }
```

Check for some impossible conditions and then return the pages.

```
    curr_order++;
    area++;
```

If we came here, then there were no free pages available in that order, so now we have to go through the next higher order.

```
} while (curr_order < MAX_ORDER);

spin_unlock_irqrestore(&zone->lock, flags);
return NULL;
```

We go through all the orders till we find a free page and return it. If we could not find any free pages we just return NULL.

2.2.5 Function expand()

File: mm/page_alloc.c
Prototype:

```
struct page * expand (zone_t *zone, struct page *page,
                      unsigned long index, int low,
                      int high, free_area_t * area)
```

This function is used to break up high order free page blocks to return the page block of the requested order and then add the remaining pages into the appropriate free lists updating the buddy bitmaps on the way. For example, when an order(1) page is requested and only order(3) pages are available, the order(3) page block has to be divided into 2 order(2) blocks and then one order(2) block is again divided into 2 order(1) blocks, from which one is returned.

```
unsigned long size = 1 << high;
```

low is the original order requested and *high* is where we had to start to get a free block. If it turned out there was a free block of the right order to begin with, no splitting will take place.

```
while (high > low) {
    if (BAD_RANGE(zone,page))
        BUG();

    area--;
    high--;
    size >>= 1;
```

Mark that we are moving to the next area after we are finished shuffling the free order lists. Size is now half as big because the order dropped by 1.

```
    memlist_add_head(&(page)->list, &(area)->free_list);
    MARK_USED(index, high, area);
```

Add the page to the free list for the "lower" area note that the lower buddy is put on the free list and the higher buddy is considered for allocation, or splitting more if necessary.

```
    index += size;
    page += size;
}
```

index is the page number inside this zone and page is the actual address.

```
if (BAD_RANGE(zone,page))
    BUG();

return page;
```

2.2.6 Function balance_classzone()

File: mm/page_alloc.c
Prototype:

```
struct page * balance_classzone(zone_t * classzone,
                                unsigned int gfp_mask,
                                unsigned int order,
                                int * freed)
```

This function is called when there is very little memory available and we can't wait for kswapd to get us some pages.

```
struct page * page = NULL;
int __freed = 0;

if (!(gfp_mask & __GFP_WAIT))
    goto out;
```

If the request cannot wait, quit, as this is a slow path.

```
if (in_interrupt())
    BUG();

current->allocation_order = order;
current->flags |= PF_MEMALLOC | PF_FREE_PAGES;
```

We set the PF_FREE_PAGES flag to indicate to __free_pages_ok() to add the pages being freed to the local free list of the current process instead of freeing them proper.

```
__freed = try_to_free_pages(classzone, gfp_mask, order);
current->flags &= ~(PF_MEMALLOC | PF_FREE_PAGES);
```

The function `try_to_free_pages()` is used to free some pages by shrinking caches and swapping old pages to disk. More on this function in section 8.2.5. Then we reset the flags.

```
if (current->nr_local_pages) {
```

At the moment *nr_local_pages* is being used as a flag to indicate if there is a free page block on the local free list of the current process. The following code is incomplete and might be clearer when newer patches of Andrea are merged in. There is a mismatch between what __free_pages_ok is actually doing, and what balance_classzone is expecting it to do. The following code believes that there are many free blocks of different orders on the local free list instead of one and tries to find the block of correct order and return that to the process while freeing the rest of the page blocks in reverse order. So we will skip over this piece of code until someone manages to complete it.

```
    struct list_head * entry, * local_pages;
    struct page * tmp;
    int nr_pages;

    local_pages = &current->local_pages;
    if (likely(__freed)) {
    /* pick from the last inserted so we're lifo */
        entry = local_pages->next;



        do {
            tmp = list_entry(entry, struct page, list);
            if (tmp->index == order &&
                memclass(page_zone(tmp), classzone)) {
                list_del(entry);
                current->nr_local_pages--;
```

```
                set_page_count(tmp, 1);
                page = tmp;

                if (page->buffers)
                    BUG();
                if (page->mapping)
                    BUG();
                if (!VALID_PAGE(page))
                    BUG();
                if (PageSwapCache(page))
                    BUG();
                if (PageLocked(page))
                    BUG();
                if (PageLRU(page))
                    BUG();
                if (PageActive(page))
                    BUG();
                if (PageDirty(page))
                    BUG();
                break;
            }
        } while ((entry = entry->next) != local_pages);
    }

    nr_pages = current->nr_local_pages;

/* free in reverse order so that the global order will
 * be lifo
 */
    while ((entry = local_pages->prev) != local_pages) {
        list_del(entry);
        tmp = list_entry(entry, struct page, list);
        __free_pages_ok(tmp, tmp->index);

        if (!nr_pages--)
            BUG();
    }
    current->nr_local_pages = 0;
}

out:
```

```
*freed = __freed;
return page;
```

Chapter 3

Slab Allocator

The majority of memory allocation requests in the kernel are for small, frequently used data structures. The physical page allocator only deals with allocations in sizes of pages and makes no attempt to use the hardware as cleanly as posssible. The slab allocator exists to serve three purposes. It provide a pool of small memory buffers packed into pages to reduce internal fragmentation. These are called the `sizes caches`. It provide pools of commonly used objects like mm_struct's to avoid the overhead of creating and destroying complex objects. Last, but not least, it tries to use the hardware cache as efficiently as possible.

The slab allocator used by linux is the same as the one outlined in Bonwick's [7] paper. Some terminology:

cache
: It is a store of recently used objects of the same type. In the slab allocator, it is the highest logical unit of storage. It has a human parse-able name like dentry_cache etc.

slab
: A slab is a container for objects and is made up of one or more page frames. A cache consists of a number of slabs.

object
: This is the smallest unit. It resides on the slab and would be something like a single dentry.

The objective is that a single page can now be used to contain a number of objects thus saving memory and avoiding internal fragmentation. The slabs are organized into three types, full slabs, partial slabs and empty ones.

Partial slabs are used if available to avoid fragmentation. To see all information on caches and slabs available in a system, type **cat /proc/slabinfo** to see a list. The fields correspond to:

cache-name	A human readable name such as vm_area_struct
num-active-objs	Number of objects that are in use
total-objs	How many are available in total including unused
obj-size	The size of each object, typically small
num-active-slabs	Number of slabs containing objects that are active
total-slabs	How many in total
num-pages-per-slab	The pages required to take one slab, typically 1

If SMP is enabled, two more fields will be displayed after a colon. These are

limit	How many objects of this type can be assigned
batchcount	How many can be assigned to each processor

This refer to the per-CPU object caches. To improve hardware utilization and to reduce the number of locks needed for an allocation, a small pool of objects is stored for each CPU. This is described further in Section 3.5

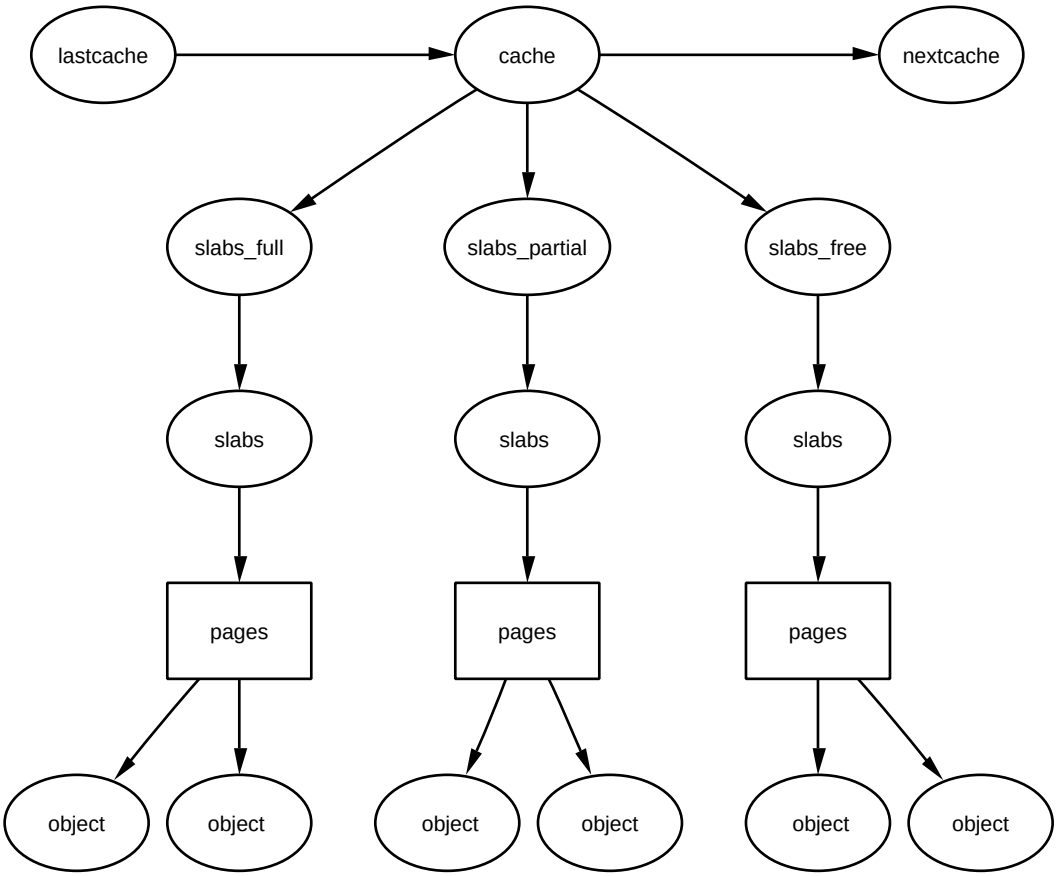

Figure 3.1: Cache Structure for the Slab Allocator

3.1 Caches

The structure of a cache is contained within a **struct kmem_cache_s** typedeffed to **kmem_cache_t**. Most of the struct is self-explanatory, but these are the principle elements to be concerned with.

List related elements	
struct list_head slabs_full	List of full slabs
struct list_head slabs_partial	List of partial slabs
struct list_head slabs_free	List of free slabs
struct list_head next	Next cache in the chain
Object properties	
char name[CACHE_NAMELEN]	Human readable name for the cache
unsigned int objsize	Size of object
unsigned int flags	Flags described later
unsigned int num	Number of objects per slab
Object creation	
void (*ctor)()	Constructor function for an object
void (*dtor)()	Destructor for object
SMP specific	
cpucache_t *cpudata[NR_CPUS]	Per-CPU cache of objects
unsigned int batchcount	Number of objects that can exist in per-cpu cache

The flags that can be assigned to a cache are as follows. This is taken directly from include/linux/slab.h.

Principle Flags	
SLAB_HWCACHE_ALIGN	align objs on a h/w cache lines
SLAB_NO_REAP	never reap from the cache
SLAB_CACHE_DMA	use GFP_DMA memory
With CONFIG_SLAB_DEBUG	
SLAB_DEBUG_FREE	Perform (expensive) checks on free
SLAB_DEBUG_INITIAL	Call constructor even if slab is a bogus creation
SLAB_RED_ZONE	Red zone objs in a cache to check for overflows
SLAB_POISON	Poison objects with known pattern for trapping uninitialized data access

To ensure that callers of *kmem_cache_create* don't use the wrong flags,

the bitmask is compared against a CREATE_MASK defined in *slab.c*. CREATE_MASK consists of all the legal flags that can be used when creating a cache. If an illegal flag is used, BUG() is invoked.

3.1.1 Cache Static Flags

The cache `flags` field is intended to give extra information about the slab. The following two flags are intended for use within the slab allocator but are not used much.

CFGS_OFF_SLAB
: Indicates that the slabs for this cache are kept off-slab. This is discussed further in Section 3.2.1

CFLGS_OPTIMIZE
: This flag is only ever set and never used

Other flags are exposed in *include/linux/slab.h* . These affect how the allocator treats the slabs.

SLAB_HWCACHE_ALIGN	Align the objects to the L1 CPU cache
SLAB_NO_REAP	Never reap slabs in this cache
SLAB_CACHE_DMA	Use memory from ZONE_DMA

If CONFIG_SLAB_DEBUG is set at compile time, the following flags are available

SLAB_DEBUG_FREE	Perform expensive checks on free
SLAB_DEBUG_INITIAL	After an object is freed, the constructor is called with a flag set that tells it to check to make sure it is initialised correctly
SLAB_RED_ZONE	This places a marker at either end of objects to trap overflows
SLAB_POISON	Poison objects with known a pattern for trapping changes made to objects not allocated or initialsed

To prevent callers using the wrong flags a **CREATE_MASK** is defined consisting of all the allowable flags.

3.1.2 Cache Dynamic Flags

The `dflags` field appears to have only one flag **DFLGS_GROWN** but it is important. The flag is set during `kmem_cache_grow` so that `kmem_cache_reap` will be unlikely to choose the cache for reaping. When the function does find a cache with this flag set, it skips the cache and removes the flag.

3.1.3 Cache Colouring

To utilize hardware cache better, the slab allocator will offset objects in different slabs by different amounts depending on the amount of space left over in the slab. The offset is in units of `BYTES_PER_WORD` unless `SLAB_HWCACHE_ALIGN` is set in which case it is aligned to blocks of L1_CACHE_BYTES for alignment to the L1 hardware cache.

During cache creation, it is calculated how many objects can fit on a slab (See Section 3.1.5) and what the bytes wasted is. Based on that, two figures are calculated for the cache desriptor

colour	The number of different offset that can be used
colour_off	The amount to offset the objects at

With the objects offset, they will use different lines on the associative hardware cache. Therefore, objects from slabs are less likely to overwrite each other in memory.

The result of this is easiest explained with example. Let us say that s_mem (the address of the first object) on the slab is 0 for convinience, that 100 bytes are wasted on the slab and alignment is to be at 32 bytes to the L1 Hardware Cache on a Pentium 2.

In this scenario, the first slab created will have it's objects start at 0. The second will start at 32, the third at 64, the fourth at 96 and the fifth will start back at 0. With this, objects from each of the slabs will not hit the same hardware cache line on the CPU.

3.1.4 Creating a Cache

The following tasks are performed by the function `kmem_cache_create` in order to create a cache.

- 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

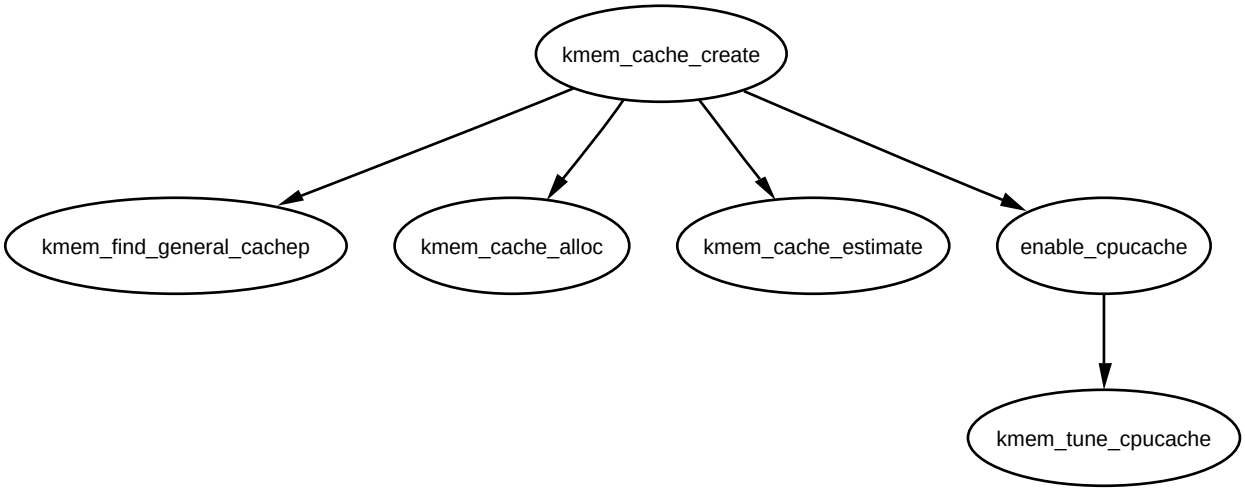

Figure 3.2: kmem_cache_create

- Initialise remaining fields in cache descriptor
- Add the new cache to the cache chain

3.1.4.1 Function kmem_cache_create()

File: mm/slab.c
Prototype:

```
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))
```

This function is responsible for creating new caches and adding them to the cache chain. For clarity, debugging information and sanity checks will be ignored as they are only important during development and secondary to the slab allocator itself. The only check that is important is the check of flags against the CREATE_MASK as the caller may request flags that are simply not available.

The arguments to kmem_cache_create are as follows

const char *name	Human readable name of the cache
size_t size	Size of the slab to create
size_t offset	Offset between each object (color)
unsigned long flags	Flags to assign to the cache as described above
void (*ctor)()	Pointer to constructor function
void (*dtor)()	Pointer to destructor

The whole beginning of the function is all debugging checks so we'll start with the last sanity check

```
/*
* Always checks flags, a caller might be
* expecting debug support which isn't available.
*/
BUG_ON(flags & ~CREATE_MASK);
```

CREATE_MASK is the full set of flags that are allowable. If debugging flags are used when they are not available, BUG will be called.

```
cachep = (kmem_cache_t *) kmem_cache_alloc
                (&cache_cache, SLAB_KERNEL);
if (!cachep)
    goto opps;
memset(cachep, 0, sizeof(kmem_cache_t));
```

Request a kmem_cache_t from the cache_cache. The cache_cache is statically initialised to avoid a chicken and egg problem, see section 3.6

```
/* Check that size is in terms of words.
* This is needed to avoid unaligned accesses
* for some archs when redzoning is used, and makes
* sure any on-slab bufctl's are also correctly aligned.
*/
if (size & (BYTES_PER_WORD-1)) {
    size += (BYTES_PER_WORD-1);
    size &= ~(BYTES_PER_WORD-1);
    printk("%sForcing size word alignment - %s\n",
                                    func_nm, name);
}
```

Comment says it all really. The next block is debugging code so is skipped here.

```
align = BYTES_PER_WORD;
if (flags & SLAB_HWCACHE_ALIGN)
    align = L1_CACHE_BYTES;
```

This will align the object size to the system word size for quicker retrieval. If the wasted space is less important than good L1 cache performance, the alignment will be made L1_CACHE_BYTES.

```
if (size >= (PAGE_SIZE>>3))
/*
* Size is large, assume best to place
* the slab management obj off-slab
* (should allow better packing of objs).
*/
flags |= CFLGS_OFF_SLAB;
```

Comment says it all really

```
if (flags & SLAB_HWCACHE_ALIGN) {
    while (size < align/2)
       align /= 2;
    size = (size+align-1)&(~(align-1));
}
```

If the cache is SLAB_HWCACHE_ALIGN, it's aligning on the size of L1_CACHE_BYES which is quiet large, 32 bytes on an Intel. So, align is adjusted to that two objects could fit in a cache line. If 2 would fit, then try 4, until as many objects are packed in. Then size is adjusted to the new alignment

```
/* Cal size (in pages) of slabs, and the num
* of objs per slab. This could be made much more
* intelligent.  For now, try to avoid using high
* page-orders for slabs.  When the gfp() funcs
* are more friendly towards high-order requests,
* this should be changed.
*/
do {
    unsigned int break_flag = 0;
```

```
cal_wastage:
    kmem_cache_estimate(cachep->gfporder, size, flags,
                            &left_over, &cachep->num);
```

Comment says it all

```
    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) {
/* Oops, this num of objs will cause problems. */
        cachep->gfporder--;
        break_flag++;
        goto cal_wastage;
    }
```

The break_flag is set so that the gfporder is reduced only once when off-slab slab_t's are in use. The second check is so the order doesn't get higher than whats possible. If num is zero, it means the gfporder is too low and needs to be increased. The last check is if the slab_t is offslab. There is a limit to how many objects can be managed off-slab. If it's hit, the order is reduced and kmem_cache_estimate is called again.

```
/*
 * The Buddy Allocator will suffer if it has to deal with
 * too many allocators of a large order. So while large
 * numbers of objects is good, large orders are not so
 * slab_break_gfp_order forces a balance
 */
    if (cachep->gfporder >= slab_break_gfp_order)
        break;
```

Comment says it all

```
    if ((left_over*8) <= (PAGE_SIZE<<cachep->gfporder))
        break;  /* Acceptable internal fragmentation. */
```

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

```
next:
    cachep->gfporder++;
} while (1);
```

This will increase the order to see if it's worth using another page to balance how many objects can be in a slab against the slab_break_gfp_order and internal fragmentation.

```
if (!cachep->num) {
     printk("kmem_cache_create: couldn't create cache %s.\n",
                                                        name);
    kmem_cache_free(&cache_cache, cachep);
    cachep = NULL;
    goto opps;
}
```

The objects must be too large to fit into the slab so clean up and goto opps that just returns.

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

The size of a slab_t is the number of objects by the size of the kmem_bufctl_ for each of them plus the size of the slab_t struct itself presuming it's kept on-slab.

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

The calculation for slab_size included slab_t even if the slab_t would be off-slab. These checks see if it would fit on-slab and if it would, place it.

```
/* Offset must be a multiple of the alignment. */
offset += (align-1);
offset &= ~(align-1);
if (!offset)
    offset = L1_CACHE_BYTES;
cachep->colour_off = offset;
cachep->colour = left_over/offset;
```

offset is the offset between each object so that the slab is coloured so that each object would get different cache lines.

```
/* init remaining fields */
if (!cachep->gfporder && !(flags & CFLGS_OFF_SLAB))
    flags |= CFLGS_OPTIMIZE;

cachep->flags = flags;
cachep->gfpflags = 0;
if (flags & SLAB_CACHE_DMA)
    cachep->gfpflags |= GFP_DMA;

spin_lock_init(&cachep->spinlock);
cachep->objsize = size;
INIT_LIST_HEAD(&cachep->slabs_full);
INIT_LIST_HEAD(&cachep->slabs_partial);
INIT_LIST_HEAD(&cachep->slabs_free);

if (flags & CFLGS_OFF_SLAB)
    cachep->slabp_cache =
            kmem_find_general_cachep(slab_size,0);
cachep->ctor = ctor;
cachep->dtor = dtor;
/* Copy name over so we don't have
 * problems with unloaded modules */
strcpy(cachep->name, name);
```

This just copies the information into the kmem_cache_t and initializes it's fields. `kmem_find_general_cachep` finds the appropriate sized sizes cache to allocate a slab descriptor from when the slab manager is kept off-slab.

```
#ifdef CONFIG_SMP
if (g_cpucache_up)
    enable_cpucache(cachep);
#endif
```

If SMP is available, enable_cpucache will create a per CPU cache of objects for this cache and set proper values for avail and limit based on how large each object is. See Section 3.5 for more details.

```
/*
* Need the semaphore to access the chain.
* Cycle through the chain to make sure there
* isn't a cache of the same name available.
*/
down(&cache_chain_sem);
{
    struct list_head *p;

    list_for_each(p, &cache_chain) {
        kmem_cache_t *pc = list_entry(p, kmem_cache_t, next);

        /* The name field is constant - no lock needed. */
        if (!strcmp(pc->name, name))
            BUG();
    }
}
```

Comment covers it

```
        /* There is no reason to lock our new cache before we
         * link it in - no one knows about it yet...
         */
        list_add(&cachep->next, &cache_chain);
        up(&cache_chain_sem);
opps:
        return cachep;
}
```

3.1.5 Calculating the Number of Objects on a Slab

During cache creation, it is determined how many objects can be stored in a slab and how much wasteage 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.

3.1.5.1 Function kmem_cache_estimate()

File: mm/slab.c
Prototype:

```
static void kmem_cache_estimate (unsigned long gfporder, size_t size,
                 int flags, size_t *left_over, unsigned int *num)
{
```

gfporder
The $2^{gfporder}$ number of pages to allocate for each slab

size
The size of each object

flags
The cache flags. See Section 3.1.1

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

```
        int i;
        size_t wastage = PAGE_SIZE<<gfporder;

        size_t extra = 0;
        size_t base = 0;
```

`wastage` is decremented through the function. It starts with the maximum possible amount of wastage.

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

`base` is where usable memory in the slab starts. If the slab descriptor is kept on cache, the base begins at the end of the slab_t struct and the number of bytes needed to store the bufctl is the size of kmem_bufctl_t. `extra` is the number of bytes needed to store kmem_bufctl_t

```
        i = 0;
        while (i*size + L1_CACHE_ALIGN(base+i*extra) <= wastage)
                i++;
```

`i` becomes the number of objects the slab can hold

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 wastage starts out as the size of the slab, it's use is overloaded here.

```
        if (i > 0)
                i--;

        if (i > SLAB_LIMIT)
                i = SLAB_LIMIT;
```

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

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

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

- `num` is now the number of objects a slab can hold
- Take away the space taken up by all the objects from wastage
- Take away the space taken up by the kmem_bufctl_t
- Wastage has now been calculated as the left over space in the slab
- Add the cache to the chain and return.

3.1.6 Growing a Cache

At this point, we have seen how the cache is created, but on creation, it is an empty cache with empty lists for it's `slab_full`, `slab_partial` and `slabs_free`.

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 takes are

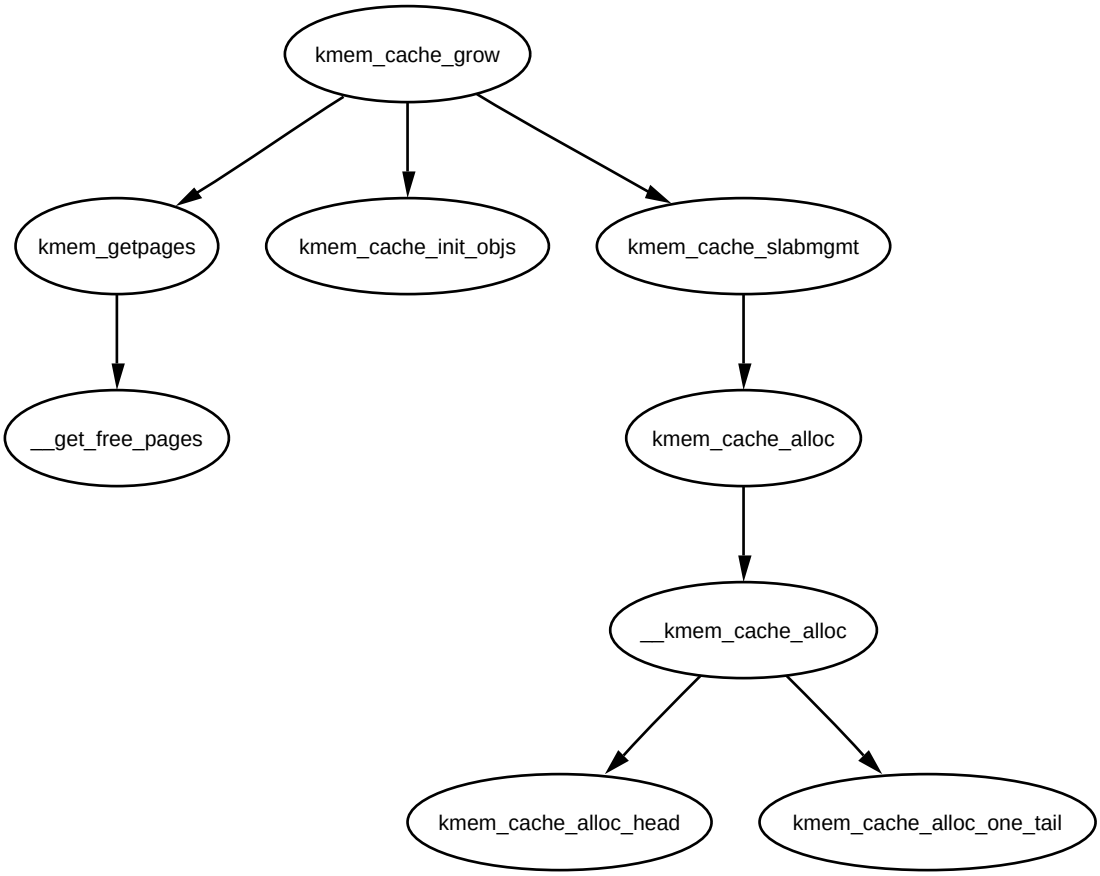

Figure 3.3: kmem_cache_grow

- 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 3.2
- Initalise objects in the slab
- Add the slab to the cache

3.1.6.1 Function kmem_cache_grow()

File: mm/slab.c
Prototype:

```
int kmem_cache_grow (kmem_cache_t * cachep,
                     int flags)
```

When there is no partial of free slabs left, the cache has to grow by allocating a new slab and placing it on the free list. It is quiet long but not too complex.

```
slab_t  *slabp;
struct page     *page;
void            *objp;
size_t           offset;
unsigned int     i, local_flags;
unsigned long    ctor_flags;
unsigned long    save_flags;

/* Be lazy and only check for valid flags here,
* keeping it out of the critical path in kmem_cache_alloc().
*/
if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW))
    BUG();
if (flags & SLAB_NO_GROW)
    return 0;
```

Straight forward. Make sure we are not trying to grow a slab that shouldn't be grown.

```
if (in_interrupt() && (flags & SLAB_LEVEL_MASK)
                                != SLAB_ATOMIC)
    BUG();
```

Make sure that if we are in an interrupt that the appropriate ATOMIC flags are set so we don't accidently sleep.

```
ctor_flags = SLAB_CTOR_CONSTRUCTOR;
local_flags = (flags & SLAB_LEVEL_MASK);
if (local_flags == SLAB_ATOMIC)
/*
* Not allowed to sleep.  Need to tell a
* constructor about this - it might need
* to know...
*/
ctor_flags |= SLAB_CTOR_ATOMIC;
```

Set the appropriate flags for growing a cache and set ATOMIC if necessary. SLAB_LEVEL_MASK is the collection of GFP masks that determines how the buddy allocator will behave.

```
/* About to mess with non-constant members - lock. */
spin_lock_irqsave(&cachep->spinlock, save_flags);
```

An interrupt safe lock has to be acquired because it's possible for an interrupt hander to affect the cache descriptor.

```
/* Get colour for the slab, and cal the next value. */
offset = cachep->colour_next;
cachep->colour_next++;
if (cachep->colour_next >= cachep->colour)
    cachep->colour_next = 0;
offset *= cachep->colour_off;
```

The colour will affect what cache line each object is assigned to on the CPU cache (See Section 3.1.3). This block of code says what offset to use for this block of objects and calculates what the next offset will me. `colour` is the number of different offsets that can be used hence `colour_next` wraps when it reaches `colour`

```
cachep->dflags |= DFLGS_GROWN;

cachep->growing++;
```

This two lines will ensure that this cache won't be reaped for some time (See Section 3.1.9). As the cache is grown, it doesn't make sense that the slab just allocated here would be deleted by kswapd in a short space of time.

```
spin_unlock_irqrestore(&cachep->spinlock, save_flags);
```

Restore the lock

```
/* Get mem for the objs. */
if (!(objp = kmem_getpages(cachep, flags)))
    goto failed;
```

Just a wrapper around __alloc_pages(). See Section 3.7

```
/* Get slab management. */
if (!(slabp = kmem_cache_slabmgmt(cachep,
                                  objp, offset,
                                  local_flags)))
    goto opps1;
```

This will allocate a slab_t struct to manage this slab. How this function decides whether to place a slab_t on or off the slab will be discussed later.

```
i = 1 << cachep->gfporder;
page = virt_to_page(objp);
do {
    SET_PAGE_CACHE(page, cachep);
    SET_PAGE_SLAB(page, slabp);
    PageSetSlab(page);
    page++;
} while (--i);
```

The struct page is used to keep track of the cachep and slabs (See Section **??**). From the head, search forward for the cachep and search back for the slabp. SET_PAGE_CACHE inserts the cachep onto the front of the list. SET_PAGE_SLAB will place the slab on end of the list. PageSetSlab is a macro which sets the PG_slab bit on the page flags. The while loop will do this for each page that was allocated for this slab.

```
kmem_cache_init_objs(cachep, slabp, ctor_flags);
```

This function, described in Section 3.3.1

```
spin_lock_irqsave(&cachep->spinlock, save_flags);
cachep->growing--;
```

Lock the cache so the slab can be inserted on the list and say that we are not growing any more so that the cache will be considered for reaping again later.

```
/* Make slab active. */
list_add_tail(&slabp->list, &cachep->slabs_free);
STATS_INC_GROWN(cachep);
cachep->failures = 0;
```

Add the slab to the list and set some statistics.

```
spin_unlock_irqrestore(&cachep->spinlock, save_flags);
return 1;
```

Unlock and return success.

```
opps1:
kmem_freepages(cachep, objp);
failed:
spin_lock_irqsave(&cachep->spinlock, save_flags);
cachep->growing--;
spin_unlock_irqrestore(&cachep->spinlock, save_flags);
return 0;
}
```

opps1 is reached if a slab manager could not be allocated. failed is reached if pages could not be allocated for the slab at all.

3.1.7 Shrinking Caches

Periodically it is necessary to shrink a cache, for instance when kswapd is woken as zones need to be balanced. Before a cache is shrinked, it is checked to make sure it isn't called from inside an interrupt. The code behind *kmem_shrink_cache()* looks a bit convulated at first glance. It's tasks are

- Delete all objects in the per CPU caches
- Delete all slabs from slabs_free unless the growing flag gets set

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.

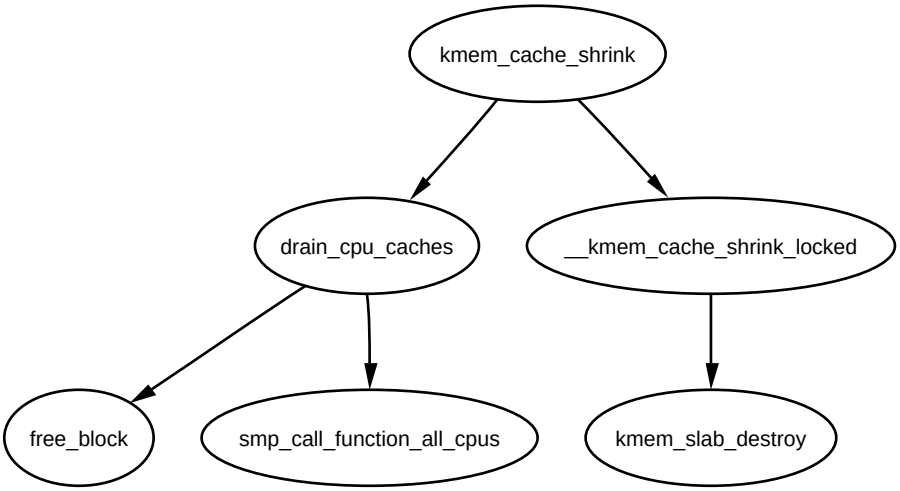

Figure 3.4: kmem_cache_shrink

3.1.7.1 Function kmem_cache_shrink()

File: mm/slab.c
Prototype:

```
int kmem_cache_shrink(kmem_cache_t *cachep)
```

```
int ret;

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

drain_cpu_caches(cachep);
```

drain_cpu_caches (Section 3.5.5.1) will try and remove the objects kept available for a particular CPU that would have been allocated earlier with kmem_cache_alloc_batch.

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

Lock and shrink

```
return ret << cachep->gfporder;
```

As the number of slabs freed is returned, bit shifting it by gfporder will give the number of pages freed. There is a similar function called __kmem_cache_shrink. The only difference with it is that it returns a boolean on whether the whole cache is free or not.

3.1.7.2 Function kmem_cache_shrink_locked()

File: mm/slab.c
Prototype:

```
int __kmem_cache_shrink_locked(kmem_cache_t *cachep)
```

This function cycles through all the slabs_free in the cache and calls kmem_slab_destory (described below) on each of them. The code is very straight forward.

```
slab_t *slabp;
int ret = 0;

/* If the cache is growing, stop shrinking. */
while (!cachep->growing) {
    struct list_head *p;

    p = cachep->slabs_free.prev;
    if (p == &cachep->slabs_free)
        break;
```

If the list `slabs_free` is empty, then both *slabs_free.prev* and *slabs_free.next* point to itself. The above code checks for this condition and quits as there are no empty slabs to free.

```
    slabp = list_entry(cachep->slabs_free.prev, slab_t, list);
```

There is an empty slab available, so get a pointer to it.

```
#if DEBUG
    if (slabp->inuse)
        BUG();
#endif
```

A bug condition where a partially used slab is in the free slab list.

```
    list_del(&slabp->list);
```

Since we are going to free this slab, remove it from the *slabs_free* list.

```
    spin_unlock_irq(&cachep->spinlock);
    kmem_slab_destroy(cachep, slabp);
    ret++;
    spin_lock_irq(&cachep->spinlock);
}
return ret;
```

Call `kmem_slab_destroy()` (which is discussed below) to actually do the formalities of freeing the slab. Increment the value of *ret*, which is used to count the number of slabs being freed.

3.1.7.3 Function __kmem_slab_destroy()

File: mm/slab.c
Prototype:

```
void kmem_slab_destroy (kmem_cache_t *cachep,
                        slab_t *slabp)
```

This function cycles through all objects in a slab and does the required cleanup. Before calling, the slab must have been unlinked from the cache.

```
if (cachep->dtor
#if DEBUG
    || cachep->flags & (SLAB_POISON | SLAB_RED_ZONE)
#endif
) {
```

If a destructor exists for this slab, or if DEBUG is enabled and the necessary flags are present, continue.

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

Cycle through all objects in the slab.

```
#if DEBUG
        if (cachep->flags & SLAB_RED_ZONE) {
            if (*((unsigned long*)(objp)) != RED_MAGIC1)
                BUG();
            if (*((unsigned long*)(objp + cachep->objsize
                        - BYTES_PER_WORD)) != RED_MAGIC1)
                BUG();
            objp += BYTES_PER_WORD;
        }
#endif

        if (cachep->dtor)
            (cachep->dtor)(objp, cachep, 0);
```

If a destructor exists for this slab, then invoke it for the object.

```
#if DEBUG
        if (cachep->flags & SLAB_RED_ZONE) {
            objp -= BYTES_PER_WORD;
        }
        if ((cachep->flags & SLAB_POISON)  &&
               kmem_check_poison_obj(cachep, objp))
            BUG();
#endif
    }
}

kmem_freepages(cachep, slabp->s_mem-slabp->colouroff);
```

`kmem_freepages()` will call the buddy allocator to free the pages for the slab.

```
if (OFF_SLAB(cachep))
    kmem_cache_free(cachep->slabp_cache, slabp);
```

If the slab_t is kept off-slab, it's cache entry must be removed.

3.1.8 Destroying Caches

Destroying a cache is yet another glorified list manager. It is called when a module is unloading itself or is being destroyed. This is to prevent caches with duplicate caches been created if the module is unloaded and loaded several times.

The steps taken to destroy a cache are

- Delete the cache from the cache chain
- Shrink the cache to delete all slabs (See Section 3.1.7)
- Free any per CPU caches (`kfree`)
- Delete the cache descriptor from the `cache_cache` (See Section: 3.3.4)

Figure 3.5 Shows the call graph for this task.

3.1.8.1 Function kmem_cache_destroy()

File: mm/slab.c
Prototype:

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

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's in use

```
        down(&cache_chain_sem);
```

Acquire the semaphore for accessing the cache chain

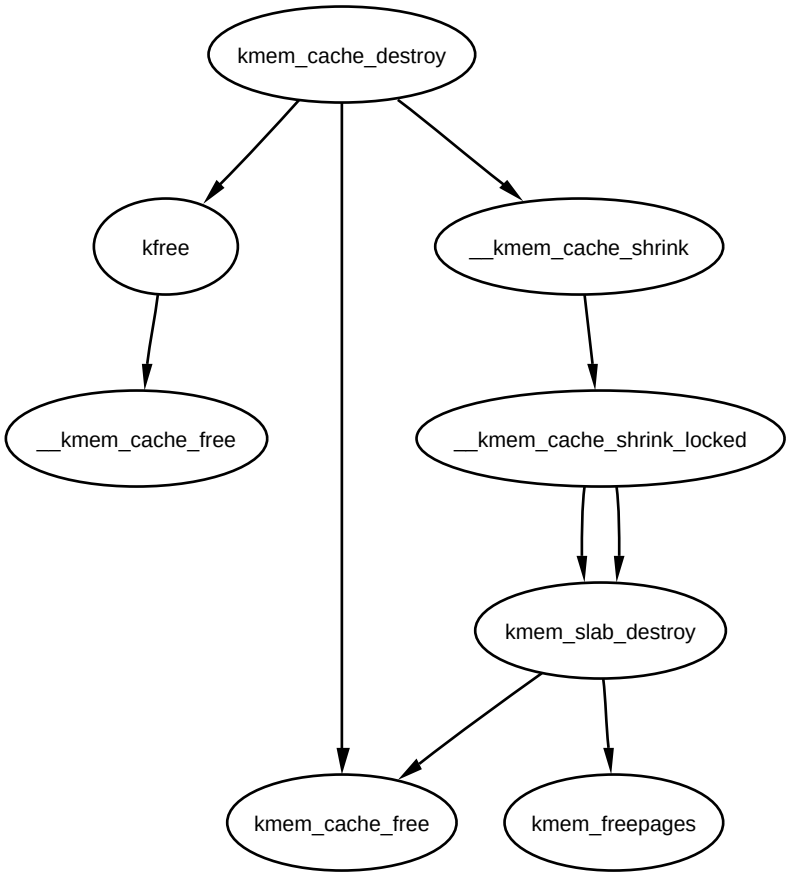

Figure 3.5: kmem_cache_destroy

```
        if (clock_searchp == cachep)
                clock_searchp = list_entry(cachep->next.next,
                                                kmem_cache_t, next);
        list_del(&cachep->next);
        up(&cache_chain_sem);
```

- Acquire the semaphore for accessing the cache chain
- Acquire the list entry from the cache chain
- Delete this cache from the cache chain
- Release the cache chain semaphore

```
        if (__kmem_cache_shrink(cachep)) {
                printk(KERN_ERR "kmem_cache_destroy: Can't free all objects %p\n",
                       cachep);
                down(&cache_chain_sem);
                list_add(&cachep->next,&cache_chain);
                up(&cache_chain_sem);
                return 1;
        }
```

Shrink the cache to free all slabs (See Section 3.1.7) 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

```
#ifdef CONFIG_SMP
        {
                int i;
                for (i = 0; i < NR_CPUS; i++)
                        kfree(cachep->cpudata[i]);
        }
#endif
```

If SMP is enabled, each per CPU data is freed using `kfree`

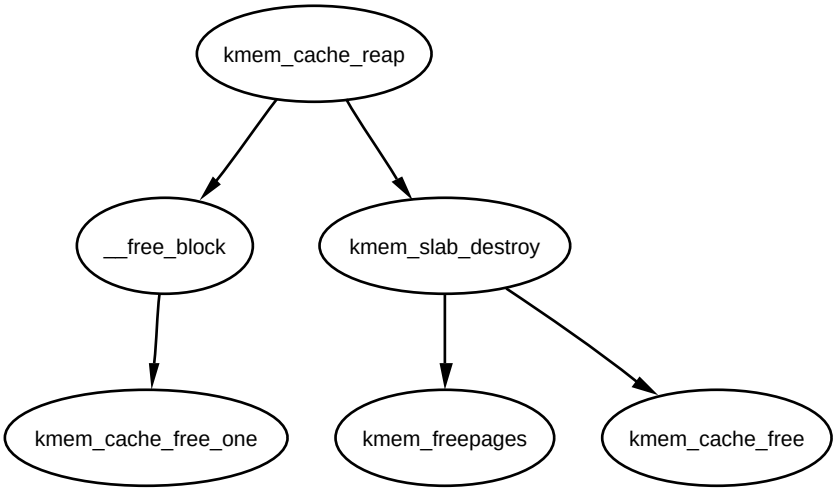

Figure 3.6: kmem_cache_reap

```
        kmem_cache_free(&cache_cache, cachep);

        return 0;
}
```

Delete the cache descriptor from the cache_cache

3.1.9 Cache Reaping

When the page allocator notices that memory is getting tight, it wakes `kswapd` to begin freeing up pages. 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.

The call graph in Figure 3.6 is deceptively simple. The task of selecting the proper cache to reap is quiet 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 it's slabs in slabs_free

3.1.9.1 Function kmem_cache_reap()

File: mm/slab.c
Prototype:

There is three distinct sections to this function. 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

```
int kmem_cache_reap (int gfp_mask)
{
        slab_t *slabp;
        kmem_cache_t *searchp;
        kmem_cache_t *best_cachep;
        unsigned int best_pages;
        unsigned int best_len;
        unsigned int scan;
        int ret = 0;
```

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

```
        if (gfp_mask & __GFP_WAIT)
                down(&cache_chain_sem);
        else
                if (down_trylock(&cache_chain_sem))
                        return 0;
```

If the caller can sleep, then acquire the semaphore else, try and acquire the semaphore and if not available, return

```
scan = REAP_SCANLEN;
best_len = 0;
best_pages = 0;
best_cachep = NULL;
searchp = clock_searchp;
```

REAP_SCANLEN is the number of caches to examine. searchp to be the last cache that was examined at the last reap

The next do..while loop scans REAP_SCANLEN caches and selects a cache to reap slabs from.

```
do {
        unsigned int pages;
        struct list_head* p;
        unsigned int full_free;

        if (searchp->flags & SLAB_NO_REAP)
                goto next;
```

If SLAB_NO_REAP is set, slip immediately

```
        spin_lock_irq(&searchp->spinlock);
```

Acquire an interrupt safe lock

```
        if (searchp->growing)
                goto next_unlock;

        if (searchp->dflags & DFLGS_GROWN) {
                searchp->dflags &= ~DFLGS_GROWN;
                goto next_unlock;
        }
```

If the cache is growing or has grown recently, skip it

```
#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
```

Free any per CPU objects to the global pool

```
                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);
```

Count the number of slabs in the slabs_free list and calculate the number of pages all the slabs hold

```
                if (searchp->ctor)
                        pages = (pages*4+1)/5;
```

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

```
                if (searchp->gfporder)
                        pages = (pages*4+1)/5;
```

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

```
                if (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;
                        }
                }
```

If this is the best canditate found for reaping so far, check if it is perfect for reaping. If this cache is perfect for reaping then update `clock_searchp` and goto perfect where half the slabs will be freed. Otherwise record the new maximums. best_len is recorded so that it is easy to know how many slabs is half of the slabs in the free list

```
next_unlock:
                spin_unlock_irq(&searchp->spinlock);
next:
                searchp =
                        list_entry(searchp->next.next,kmem_cache_t,next);
        } while (--scan && searchp != clock_searchp);
```

This next_unlock label is reached if it was found the cache was growing after acquiring the lock so the cache descriptor lock is released. Move to the next entry in the cache chain and keep scanning until REAP_SCANLEN is reached or until the whole chain has been examined.

At this point a cache has been selected to reap from. The next block will free half of the free slabs from the selected cache.

```
        clock_searchp = searchp;

        if (!best_cachep)
                goto out;
```

Update clock_searchp for the next cache reap. If a cache was not selected, goto out to free the cache chain and exit

```
        spin_lock_irq(&best_cachep->spinlock);
```

Acquire the cache chain spinlock and disable interrupts

```
perfect:
        best_len = (best_len + 1)/2;

        for (scan = 0; scan < best_len; scan++) {
```

Adjust best_len to be the number of slabs to free and free best_len number of slabs.

```
                struct list_head *p;

                if (best_cachep->growing)
                        break;
```

If the cache is growing, exit

```
                p = best_cachep->slabs_free.prev;
                if (p == &best_cachep->slabs_free)
                        break;
                slabp = list_entry(p,slab_t,list);
```

Get a slab from the list and check to make sure there is slabs left to free on it before acquiring the slab pointer.

```
#if DEBUG
                if (slabp->inuse)
                        BUG();
#endif
           list_del(&slabp->list);
                STATS_INC_REAPED(best_cachep);
```

A debugging check if enabled. Remove the slab from the list as it's about to be destroyed. Update statistics if enabled.

```
                spin_unlock_irq(&best_cachep->spinlock);
                kmem_slab_destroy(best_cachep, slabp);
                spin_lock_irq(&best_cachep->spinlock);
        }
```

Release the cache descriptor while deleting the slab because the cache descriptor is safe and move to the next slab to free in the cache

```
        spin_unlock_irq(&best_cachep->spinlock);
        ret = scan * (1 << best_cachep->gfporder);
out:
        up(&cache_chain_sem);
        return ret;
}
```

The requesite number of slabs has been freed to record the number of pages that were freed, release the cache descriptor locks and return the result.

3.2 Slabs

As mentioned, a slab consists of one or more pages assigned to contain objects. The job of this struct is to manage the objects in the slab. The struct to describe a slab is simple:

```
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 head of the list this slab belongs to.

colouroff
The colour to help utilise the hardware cache better.

s_mem
Starting address for objects.

inuse
Number of active objects in the slab.

free
Used for linking free objects together.

The array kmem_bufctl_t array is stored immediately after this structure. See Section 3.4 for more details on the kmem_bufctl_t array.

3.2.1 Storing the Slab Descriptor

The slab_t struct has to be stored somewhere. It can be either stored off slab in which case the memory will be allocated from one of the sizes caches. Else it will be stored within the slab itself. The sizes caches are described in a later section dealing with kmalloc. They are caches which store blocks of memory of sizes that are powers of two.

The reader will note that given the slab manager or an object within the slab, there does not appear to be a way to determine what slab or cache they belong to. This is addressed by using the page→list that makes up the cache. **SET_PAGE_CACHE** and **SET_PAGE_SLAB** use next and prev on the page list to track what cache and slab an object belongs to. To get the descriptors from the page, the macros **GET_PAGE_CACHE** and **GET_PAGE_SLAB** are available. This is illustrated as best as possible in Figure 3.7

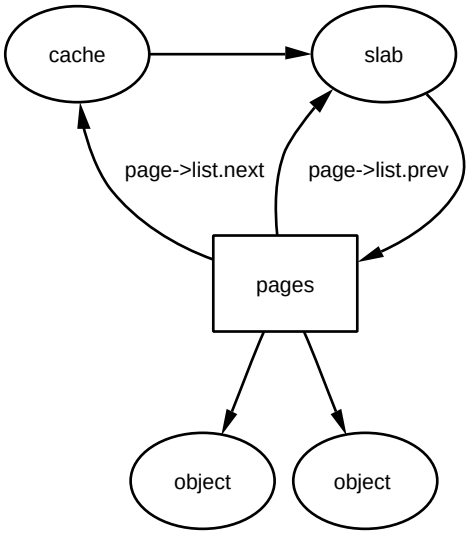

Figure 3.7: Page to Cache and Slab Relationship

Caches are linked together with the *next* field. Each cache consists of one or more slabs which are blocks of memory of one or more pages. Each slab contains multiple numbers of objects, possibly with gaps between them so that they hit different cache lines. If, during cache creation, the flag SLAB_HWCACHE_ALIGN is specified, the objsize is adjusted up to L1_CACHE_BYTES so that the objects will be cache aligned. This will create the gaps between objects. The slab_t or slab management structure may

be kept on the slab or off it. If on the slab, it is at the beginning. If off-cache, it is stored in an appropriately sized memory cache.

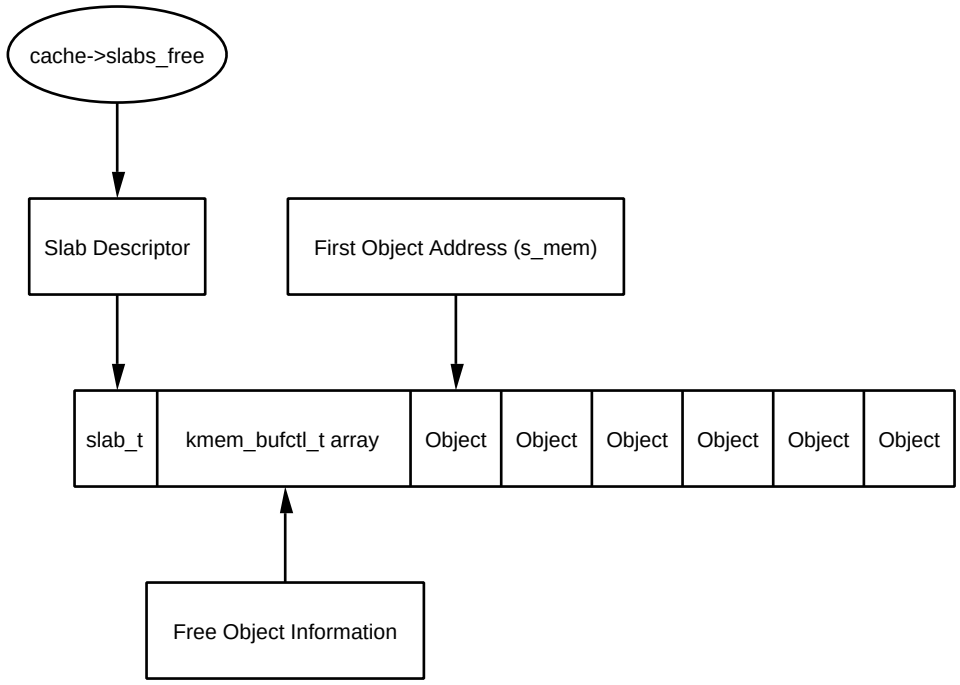

Figure 3.8: Slab With Descriptor On-Slab

Figure 3.9 illustrates how a cache uses a sizes cache to store the slab descriptor.

The `struct page`'s *list* element is used to track where cache_t and slab_t are stored (see kmem_cache_grow). The $list \rightarrow next$ pointer points to kmem_cache_t (the cache it belongs to) and $list \rightarrow prev$ points to slab_t (the slab it is part of). So given an object, we can easily find the associated cache and slab through these pointers.

3.2.1.1 Function kmem_cache_slabmgmt()

File: `mm/slab.c`
Prototype:

```
slab_t * kmem_cache_slabmgmt (kmem_cache_t *cachep,
                              void *objp,
                              int colour_off,
                              int local_flags)
```

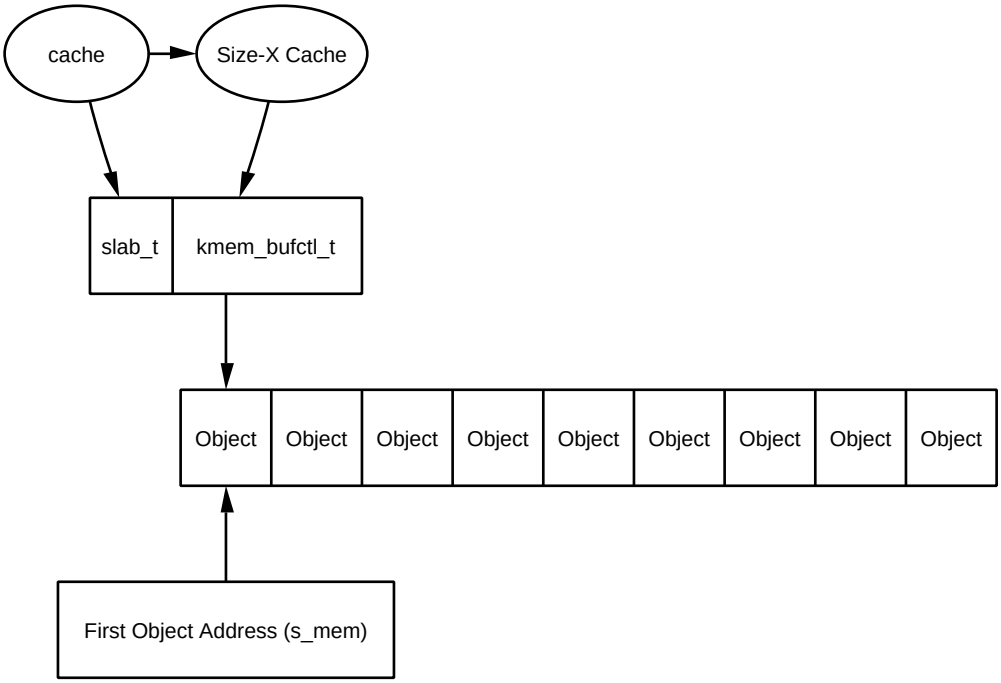

Figure 3.9: Slab With Descriptor Off-Slab

This function allocates a new slab_t and places it in the correct place.

```
slab_t *slabp;

if (OFF_SLAB(cachep)) {
    /* Slab management obj is off-slab. */
    slabp = kmem_cache_alloc(cachep->slabp_cache,
                             local_flags);
    if (!slabp)
        return NULL;
```

The first check is to see if the slab_t is kept off the slab. If it is, *cachep* → *slabp_cache* will be pointing to the cache of memory allocations large enough to contain the slab_t. The different size caches are the same ones used by kmalloc.

```
} else {
    slabp = objp+colour_off;
```

```
    colour_off += L1_CACHE_ALIGN(cachep->num *
                            sizeof(kmem_bufctl_t)
                            + sizeof(slab_t));
}
```

Otherwise the slab_t struct is contained on the slab itself at the beginning of the slab.

```
slabp->inuse = 0;
slabp->colouroff = colour_off;
slabp->s_mem = objp+colour_off;

return slabp;
```

The most important one to note here is the value of s_mem. It'll be set to be at the beginning of the slab if the slab manager is off slab but at the end of the slab_t if it's on slab.

3.2.1.2 Function kmem_find_general_cachep()

File: mm/slab.c
Prototype:

If the slab descriptor is to be kept off-slab, this function, called during cache creation will find the appropriate sizes cache to use and will be stored within the cache descriptor in the field `slabp_cache`.

```
kmem_cache_t * kmem_find_general_cachep (size_t size,
                                                int gfpflags)
{
```

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

```
        cache_sizes_t *csizep = cache_sizes;

        for ( ; csizep->cs_size; csizep++) {
                if (size > csizep->cs_size)
                        continue;
                break;
        }
```

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

```
        return (gfpflags & GFP_DMA) ? csizep->cs_dmacachep :
                                              csizep->cs_cachep;
}
```

Return either a normal or DMA sized cache depending on the gfpflags passed in. In reality, only the `cs_cachep` is ever passed back

3.3 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.

3.3.1 Initializing Objects

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 it's 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.

3.3.1.1 Function kmem_cache_init_objs()

File: mm/slab.c
Prototype:

```
void kmem_cache_init_objs (kmem_cache_t * cachep,
                           slab_t * slabp,
                           unsigned long ctor_flags)
```

This function is called to initialize all the objects on a slab once by `kmem_cache_grow` when creating a new slab.

```
int i;

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

This steps through the number of objects that can be contained onslab. $(cachep \rightarrow objsize * i)$ will give an offset from s_mem where ith object is. [note: s_mem is used to point to the first object].

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

If debugging is enabled, RED_MAGIC1 will be written at the beginning and end of the object. Later when the object is used, this will be checked again. If the values are not still RED_MAGIC1, it's known that the object was activated twice or else was overrun.

```
    if (cachep->ctor)
        cachep->ctor(objp, cachep, ctor_flags);
```

A constructor is called for the object if available. Users are warned that a cache with a constructor can not allocate memory from itself because it would end up recursively calling this.

```
#if DEBUG
    if (cachep->flags & SLAB_RED_ZONE)
        objp -= BYTES_PER_WORD;
```

This block of debugging code will adjust the address of objp to take into account the size of RED_MAGIC1 that was added before calling the constructor. The constructor receives a pointer to the actual data block and not the debugging marker.

```
    if (cachep->flags & SLAB_POISON)
        /* need to poison the objs */
        kmem_poison_obj(cachep, objp);
```

This function won't be discussed in detail. It simply fills an object with POISON_BYTES and marks the end with POISON_END.

```
    if (cachep->flags & SLAB_RED_ZONE) {
        if (*((unsigned long*)(objp)) != RED_MAGIC1)
            BUG();
        if (*((unsigned long*)(objp + cachep->objsize
                    - BYTES_PER_WORD)) != RED_MAGIC1)
            BUG();
    }
#endif
```

This checks to make sure RED_MAGIC1 is preserved by the poisoning.

```
    slab_bufctl(slabp)[i] = i+1;
}
```

This initialises the kmem_bufctl_t array. See Section 3.4

```
slab_bufctl(slabp)[i-1] = BUFCTL_END;
slabp->free = 0;
```

Mark the end of the kmem_bufctl_t array with BUFCTL_END. free is set to 0 so that the first object allocated will be the 0th object on the slab.

3.3.2 Allocating Objects

This section covers what is needed to allocate an object. The allocator behaves slightly different in the UP and SMP cases and will be treated seperatly in this section. Figure 3.10 shows the basic call graph that is used to allocate an object in the UP case.

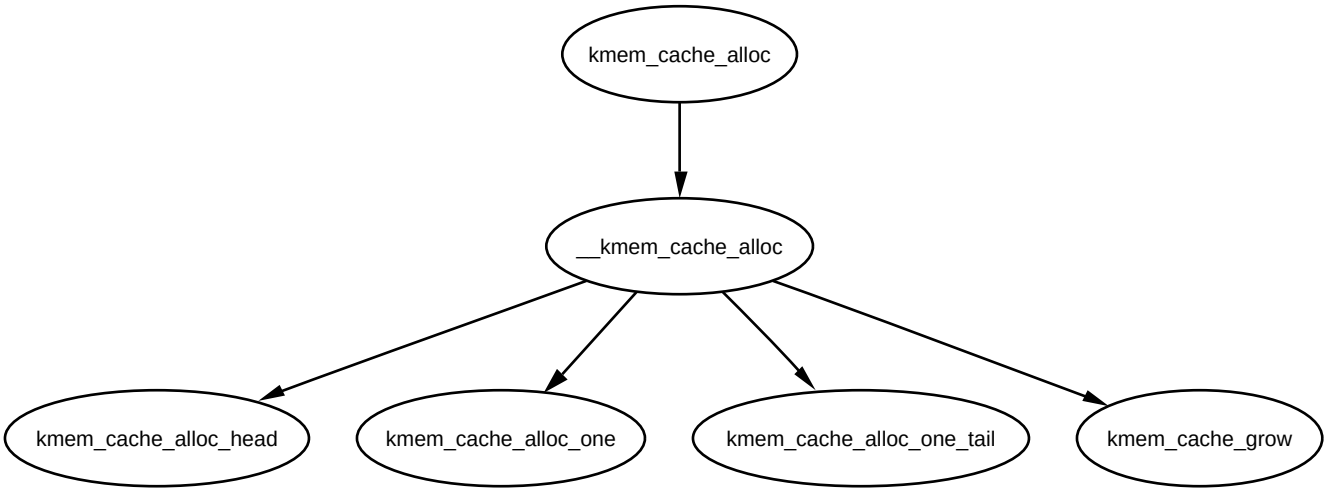

Figure 3.10: kmem_cache_alloc UP

As is clear, there is four basic steps. The first step (head) covers basic checking to make sure the allocation is allowable. The second step is to select which slabs list to allocate from. This is one of slabs_partial or slabs_free. If there is no slabs in slabs_free, the cache is grown (See Section 3.1.6) to

create a new slab in slabs_free. The final step is to allocate the object from the selected slab.

The SMP case takes one futher step. Before allocating one object, it will check to see if there is one available from the per-CPU cache and use it if there is. If there is not, it will allocate `batchcount` number of objects in bulk and place them in it's per-cpu cache. See Section 3.5 for details.

3.3.2.1 Function __kmem_cache_alloc()

File: mm/slab.c
Prototype:

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

The function takes two parameters:

kmem_cache_t *cachep	The cache to allocate from
int flags	Flags for the allocation

The flags are defined in include/linux/slab.h and correspond to GFP page flag options, mainly of important to the allocator. Callers sometimes call with either SLAB_ or GFP_ flags. This section will only deal with the SLAB_ flags and what they mean. They can be one of:

SLAB_NOFS	This flag tells the page free logic to not make any calls to the file-system layer. This is important for the allocation of buffer heads for instance where it is important the file-system does not end up recursively calling itself
SLAB_NOIO	Do not start any IO. For example, in `try_to_free_buffers()`, no attempt to write out busy buffer pages will be made if this slab flag is used
SLAB_NOHIGHIO	Treated the same as SLAB_NOIO according to buffer.c
SLAB_ATOMIC	Allocations made with this flag may take whatever measures necessary to get a page without sleeping. This is used for the buffer head emergency pool for instance. The page allocator will not sleep when this flag is set.
SLAB_USER	This translates to say that the allocator may sleep, make FS calls and engage in IO. In reality, the flag does not appear to be used anywhere in the code and is probably included to have a nice one to one mapping to the GFP_ flags.

SLAB_KERNEL	Used when the caller just wants the object to be allocated and are not particular about what needs to be done to get it. The caller will perform IO, sleep and can make calls to the file-system.
SLAB_NFS	Supplied to provide a mapping to GFP_NFS. In reality, it is never used. The only caller that needs it uses GFP_NFS directly.
SLAB_DMA	Used to flag a cache that is the should allocate memory suitable for use with DMA. This will make the allocation from the sizes cache dealing with DMA and if the page allocator is used, it'll only allocate from ZONE_DMA.

For completeness, there are two other SLAB flags which exist. They are:

SLAB_LEVEL_MASK	This rarely used mask removes any bits from the flags which the slab allocator is not aware of.
SLAB_NO_GROW	This flags a cache that the number of slabs within it should not grow. It only appears to be used by `kmem_cache_grow` but does not appear to be set anywhere in the code.

They largely affect how the buddy allocator will behave later. *kmem_cache_alloc* calls *__kmem_cache_alloc* directly. It comes in two flavors, UP and SMP.

3.3.2.2 Allocation on UP

With the #defines removed, this is what the function looks like.

```
void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
    unsigned long save_flags;
    void* objp;

    kmem_cache_alloc_head(cachep, flags);
```

kmem_cache_alloc_head() is a simple sanity check. It asserts that the wrong combination of SLAB_DMA and GFP_DMA are not used with the flags.

```
try_again:
```

```
    local_irq_save(save_flags);
    objp = kmem_cache_alloc_one(cachep);
    local_irq_restore(save_flags);

    return objp;
```

The macro `kmem_cache_alloc_one` which will be described in section 3.3.3 allocates an object if there is a partially allocated or completely free slab available. `local_irq_save` disables interrupts and saves the flags. This will guarantee synchronization which is needed for `kmem_cache_alloc_one`. A spinlock can not be used because an interrupt handler can not take out a spinlock and an interrupt handler can call this function.

```
alloc_new_slab:
    local_irq_restore(save_flags);
    if (kmem_cache_grow(cachep, flags))
        /* Someone may have stolen our objs.
         * Doesn't matter, we'll
         * just come back here again.
         */
        goto try_again;
    return NULL;
}
```

Note the label alloc_new_slab which has no goto apparently, is used in `kmem_cache_alloc_one`. We come here if there are no free or partially free slabs available. So we grow the cache by one more slab and try again.

3.3.2.3 Allocation on SMP

There are two principle differences between allocations on UP and on SMP. The first one is the use of spinlocks, they become necessary for SMP. The second is that slabs and objects are bound to processors for better use of hardware cache. We'll see how this is achieved. First, this is what __kmem_cache_alloc looks like for the SMP case.

Most of this is the same as for the UP case so we'll only deal with the SMP related code.

```
void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
    unsigned long save_flags;
```

```
    void* objp;

    kmem_cache_alloc_head(cachep, flags);
try_again:

    local_irq_save(save_flags);

    {
        cpucache_t *cc = cc_data(cachep);
```

cc_data is a macro which returns the cpucache_s struct for this CPU. The struct has two members `avail` and `limit`. `avail` is how many objects are available and `limit` is the maximum number that this processor may have.

```
        if (cc) {
            if (cc->avail) {
                STATS_INC_ALLOCHIT(cachep);
                objp = cc_entry(cc)[--cc->avail];
```

If the cpucache_t data is available, check to see if there is an object available. If there is, allocate it. From the cc_entry macro, it would appear that the objects are stored in memory after the cpucache_t .

```
            } else {
               STATS_INC_ALLOCMISS(cachep);
               objp = kmem_cache_alloc_batch(cachep,cc,flags);
               if (!objp)
                   goto alloc_new_slab_nolock;
            }
```

Else, there isn't an object available from the cache so more have to be allocated. The function `kmem_cache_alloc_batch()` will be discussed in detail in section **??**.

```
        } else {
            spin_lock(&cachep->spinlock);
            objp = kmem_cache_alloc_one(cachep);
            spin_unlock(&cachep->spinlock);
        }
    }
```

If a cpucache is not available, just allocate one object in the same way a UP does it except that a spinlock is held.

```
    local_irq_restore(save_flags);
    return objp;

/* kmem_cache_alloc_one contains a goto to this label */
alloc_new_slab:
    spin_unlock(&cachep->spinlock);
alloc_new_slab_nolock:
    local_irq_restore(save_flags);
    if (kmem_cache_grow(cachep, flags))
        /* Someone may have stolen our objs.
         * Doesn't matter, we'll
         * just come back here again.
         */
        goto try_again;
    return NULL;
}
```

3.3.3 Macro kmem_cache_alloc_one()

File: mm/slab.c
Prototype:

```
kmem_cache_alloc_one(cachep)
```

```
#define kmem_cache_alloc_one(cachep)                \
({                                                  \
        struct list_head * slabs_partial, * entry;  \
        slab_t *slabp;                              \
                                                    \
        slabs_partial = &(cachep)->slabs_partial;   \
        entry = slabs_partial->next;                \
        if (unlikely(entry == slabs_partial)) {     \
                struct list_head * slabs_free;      \
                slabs_free = &(cachep)->slabs_free; \
                entry = slabs_free->next;           \
                if (unlikely(entry == slabs_free))  \
                        goto alloc_new_slab;        \
```

```
                list_del(entry);                    \
                list_add(entry, slabs_partial);     \
        }                                           \
                                                    \
        slabp = list_entry(entry, slab_t, list);    \
        kmem_cache_alloc_one_tail(cachep, slabp);   \
})
```

This is nice and straight forward. It's checks are simply

- If there is a partially filled slab, use it
- If there is a free slab, use it
- Otherwise goto alloc_new_slab to allocate a new slab. Another goto will bring us back later.

3.3.3.1 Function kmem_cache_alloc_one_tail()

File: mm/slab.c
Prototype:

```
void * kmem_cache_alloc_one_tail (kmem_cache_t *cachep,
                                  slab_t *slabp)
```

Once a slab is found that can be used, *kmem_cache_alloc_one_tail()* is called. The main complexity in this function is in the debugging so lets examine it in pieces:

```
void *objp;

STATS_INC_ALLOCED(cachep);
STATS_INC_ACTIVE(cachep);
STATS_SET_HIGH(cachep);
```

This just sets some stats about the usage of the cache.

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

s_mem is the pointer to the beginning of the objects within the slab and *free* is the index of the first object on the slab's free-list. Multiplying it by the size of each object will make objp the address of a free object. slab_bufctl is a macro which casts kmem_bufctl_t to slab_t and adds 1 to it effectively giving the address of the next free object.

Without debugging, the objp would be returned as is, but with debugging enabled, more work is done.

```
#if DEBUG
if (cachep->flags & SLAB_POISON)
    if (kmem_check_poison_obj(cachep, objp))
        BUG();
```

If an object is poisoned, it'll be marked with POISON_BYTES with a POISON_END at the end of it. If objects were accidently overlapped, kmem_cache_poison_obj will find POISON_END at the wrong place and BUG it.

```
if (cachep->flags & SLAB_RED_ZONE) {
/* Set alloc red-zone, and check old one. */
    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
```

This checks for overflow of the area. When an object is inactive, it will be marked at either end with RED_MAGIC1. The object is now becoming active to either end is now marked with RED_MAGIC2. If another object had overflowed, the magic number would have been overwritten so BUG is called to signal that

```
return objp;
}
```

Return the object which has been allocated.

3.3.3.2 Function kmem_cache_alloc_batch()

File: mm/slab.c
Prototype:

```
void* kmem_cache_alloc_batch(kmem_cache_t* cachep,
                             cpucache_t* cc,
                             int flags)
```

kmem_cache_alloc_batch() is very simple. It allocates batchcount number of new objects and places each of them on the cpucache to be used for later allocations. This leads to better cache utilization.

```
int batchcount = cachep->batchcount;

spin_lock(&cachep->spinlock);
while (batchcount--) {
```

Loop batchcount times

```
 struct list_head * slabs_partial, * entry;
 slab_t *slabp;
 /* Get slab alloc is to come from. */
 slabs_partial = &(cachep)->slabs_partial;
 entry = slabs_partial->next;
```

Find a slab that is partially full

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

If there isn't a partial slab, find an empty one

```
     if (unlikely(entry == slabs_free))
         break;
```

If there isn't a free one, break which will either return an object that has been allocated or else return NULL which will grow the cache.

```
     list_del(entry);
     list_add(entry, slabs_partial);
 }
```

Otherwise remove the slab from the list it's on and place it on the slabs_partial list

```
    slabp = list_entry(entry, slab_t, list);
    cc_entry(cc)[cc->avail++] =
        kmem_cache_alloc_one_tail(cachep, slabp);
}
```

Get a slabp from the slabs_partial list and allocate one object in the same way a UP does it.

```
spin_unlock(&cachep->spinlock);

if (cc->avail)
    return cc_entry(cc)[--cc->avail];
return NULL;
```

Free the spinlock and return an object if possible. Otherwise return NULL to the cache can be grown.

3.3.4 Object Freeing

This section covers what is needed to free an object. In many ways, it is similiar to how objects are allocated and just like the allocation, there is a UP and SMP flavour. The principle difference is that the SMP version frees the object to the per CPU cache. Figure 3.11 shows the very simply call graph used

3.3.4.1 Function kmem_cache_free()

File: mm/slab.c
Prototype:

```
void kmem_cache_free (kmem_cache_t *cachep, void *objp)
{
        unsigned long flags;
#if DEBUG
        CHECK_PAGE(virt_to_page(objp));
        if (cachep != GET_PAGE_CACHE(virt_to_page(objp)))
                BUG();
#endif
```

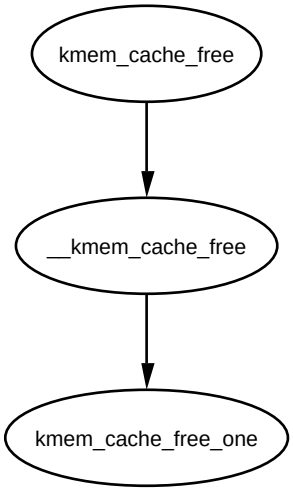

Figure 3.11: kmem_cache_free

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 **??**)

```
        local_irq_save(flags);
        __kmem_cache_free(cachep, objp);
        local_irq_restore(flags);
}
```

Interrupts are disabled to protect the path. __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. Reenable interrupts.

3.3.4.2 Function __kmem_cache_free()

File: mm/slab.c
Prototype:

This covers what the function does in the UP case. It is obvious the object is just freed to the global pool. The SMP case will be dealt with in the next section

```
static inline void __kmem_cache_free (kmem_cache_t *cachep, void* objp)
{
        kmem_cache_free_one(cachep, objp);
}
```

3.3.4.3 Function __kmem_cache_free()

File: mm/slab.c
Prototype:
This case is slightly more interesting.

```
static inline void __kmem_cache_free (kmem_cache_t *cachep, void* objp)
{
        cpucache_t *cc = cc_data(cachep);
```

Get the data for this per CPU cache (See Section 3.5

```
        CHECK_PAGE(virt_to_page(objp));

        if (cc)
```

Make sure the page is a slab page. 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

```
                int batchcount;
                if (cc->avail < cc->limit) {
                        STATS_INC_FREEHIT(cachep);
                        cc_entry(cc)[cc->avail++] = objp;
                        return;
                }
```

If the number of available in the per CPU cache is below limit, then add the object to the free list and return. Update statistics if enabled.

```
                STATS_INC_FREEMISS(cachep);
                batchcount = cachep->batchcount;
                cc->avail -= batchcount;
                free_block(cachep,
                        &cc_entry(cc)[cc->avail],batchcount);
                cc_entry(cc)[cc->avail++] = objp;
                return;
```

The pool has overflowed so batchcount number of objects is going to be freed to the global pool. Update the number of available (avail) objects. Free a block of objects to the global cache. Free the requested object and place it on the per CPU pool.

```
        } else {
                free_block(cachep, &objp, 1);
        }
}
```

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

3.3.4.4 Function kmem_cache_free_one()

File: mm/slab.c
Prototype:

```
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));
```

Make sure the page is a slab page. Get a slab descriptor for the page.

```
#if DEBUG
        if (cachep->flags & SLAB_DEBUG_INITIAL)
                cachep->ctor(objp, cachep,
                        SLAB_CTOR_CONSTRUCTOR|SLAB_CTOR_VERIFY);
```

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

```
        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();
        }
```

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

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

Poison the freed object with a known pattern. This function will confirm the object is a part of this slab and cache. It will then check the free list (bufctl) to make sure this is not a double free. See Section **??**

```
        {
                unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;

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

Calculate the index for the object been freed. As this object is now free, update the bufctl to reflect that. See Section 3.4

```
        STATS_DEC_ACTIVE(cachep);

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

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

```
                } else if (unlikely(inuse == cachep->num)) {
                        /* Was full. */
                        list_del(&slabp->list);
                        list_add(&slabp->list, &cachep->slabs_partial);
                }
        }
}
```

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

3.3.4.5 Function free_block()

File: mm/slab.c
Prototype:

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

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

The parameters are

cachep
The cache that objects are been freed from

objpp
Pointer to the first object to free

len
The number of objects to free

The code ....

- Acquire a lock to the cache descriptor
- Discussed in next section
- Release the lock

3.3.4.6 Function __free_block()

File: mm/slab.c
Prototype:

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);
}
```

3.4 Tracking Free Objects

The slab allocator has to have a quick and simple way of tracking where free objects are on the partially filled slabs. It achieves this via a mechanism called **kmem_bufctl_t** that is associated with each slab manager as obviously it is up to the slab manager to know where it's free objects are.

Historically, and according to the paper describing the slab allocator [7], **kmem_bufctl_t** was a linked list of objects. In Linux 2.2.x, this struct was a union of three items, a pointer to the next free object, a pointer to the slab manager and a pointer to the object. Which it was depended on the state of the object.

Today, the slab and cache a page belongs to is determined by the list field in `struct page` illustrated in Figure 3.7 in Section 3.2

3.4.1 kmem_bufctl_t

The kmem_bufctl_t is simply an unsigned integer and is treated as an array stored after the slab manager (See Section 3.2). The number of elements in the array is the same as the number of objects on the slab.

```
typedef unsigned int kmem_bufctl_t;
```

As the array is kept after the slab descriptor and there is no pointer to the first element directly, a helper macro **slab_bufctl** is provided.

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

This seemingly cryptic macro is quiet simple when broken down. The parameter `slabp` is to the slab manager. The block `((slab_t*)slabp)+1` casts slabp to a slab_t struct and adds 1 to it. This will give a `slab_t *` pointer to the beginning of the kmem_bufctl_t array. `(kmem_bufctl_t *)` recasts that pointer back to the required type. The results in blocks of code that contain `slab_bufctl(slabp)[i]`. Translated that says, take a pointer to a slab descriptor, offset it with slab_bufctl to the beginning of the kmem_bufctl_t array and give the ith element of the array.

The index to the next free object in the slab is stored in `slab_t→free` eliminating the need for a linked list to track free objects. When objects are allocated or freed, this pointer is updated based on information in the kmem_bufctl_t array.

3.4.2 Initialising the kmem_bufctl_t Array

When a cache is grown, alll the objects and the kmem_bufctl_t array on the slab are initialised. The array is filled with the index of each object beginning with 1 and ending with the marker `BUFCTL_END`.

The value 0 is stored in `slab_t→free` as the $0th$ object is the first free object to be used. See section 3.3.1 to see the function which initialised the array.

The idea is that for a given object n, the index of the next free object will be stored in kmem_bufctl_t[n]. Looking at the array above, the next object free after 0 is 1. After 1, there is two and so on.

3.4.3 Finding the Next Free Object

`kmem_cache_alloc` is the function which allocates an object. It uses the function `kmem_cache_alloc_one_tail` (See Section 3.3.3.1) to allocate the object and update the kmem_bufctl_t array.

`slab_t→free` has the index of the first free object. The index of the next free object is at kmem_bufctl_t[slab_t→free]. In code terms, this looks like

```
objp = slabp->s_mem + slabp->free*cachep->objsize;
slabp->free=slab_bufctl(slabp)[slabp->free];
```

`slabp→s_mem` is the index of the first object on the slab. `slabp→free` is the index of the object to allocate and it has to be multipled by the size of an object.

The index of the next free object to allocate is stored at kmem_bufctl_t[slabp→free]. There is no pointer directly to the array hence

the helper macro slab_bufctl is used. Note that the kmem_bufctl_t array is not changed during allocations but that the elements that are unallocated are unreachable. For example, after two allocations, index 0 and 1 of the kmem_bufctl_t array are not pointed to by any other element.

3.4.4 Updating kmem_bufctl_t

The kmem_bufctl_t list is only updated when an object is freed in the function `kmem_cache_free_one`. The array is updated with this block of code

```
unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;

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

`objp` is the object about to be freed and objnr is it's index. `kmem_bufctl_t[objnr]` is updated to pointer to the current value of `slabp→free` efficively placing the object pointed to by free on the pseudo linked list. slabp→free is updated to the object been freed so that it will be the next one allocated.

3.5 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.

3.5.1 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 immediatly after the cpucache_t struct. This is very similiar to how objects are stored after a slab descriptor illustrated in Section **??**.

3.5.2 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 major 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.

3.5.3 Enabling Per-CPU Caches

When a cache is created, it's 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 guarenteed safe access to it.

3.5.3.1 Function enable_all_cpucaches()

File: mm/slab.c
Prototype:

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;
```

Obtain the semaphore to the cache chain and get the first cache on the chain

```
        do {
                kmem_cache_t* cachep = list_entry(p, kmem_cache_t, next);

                enable_cpucache(cachep);
                p = cachep->next.next;
        } while (p != &cache_cache.next);
```

Cycle through the whole chain. For each cache on it, enable it's cpucache. Note that this will skip the first cache on the chain but cache_cache doesn't need a cpucache as it's so rarely used.

```
        up(&cache_chain_sem);
}
```

Release the semaphore

3.5.3.2 Function enable_cpucache()

File: mm/slab.c
Prototype:

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.

```
static void enable_cpucache (kmem_cache_t *cachep)
{
        int err;
        int limit;

        if (cachep->objsize > PAGE_SIZE)
                return;
        if (cachep->objsize > 1024)
                limit = 60;
        else if (cachep->objsize > 256)
                limit = 124;
        else
                limit = 252;
```

If an object is larger than a page, don't create a per CPU cache as they are too expensive. If an object is larger than 1KB, keep the cpu cache below 3MB in size. The limit is set to 124 objects to take the size of the cpucache descriptors into account. For smaller objects, just make sure the cache doesn't go above 3MB in size

```
        err = kmem_tune_cpucache(cachep, limit, limit/2);
```

Allocate the memory for the cpucache.

```
        if (err)
                printk(KERN_ERR
                        "enable_cpucache failed for %s, error %d.\n",
                                        cachep->name, -err);
}
```

Print out an error message if the allocation failed

3.5.3.3 Function kmem_tune_cpucache()

File: mm/slab.c
Prototype:

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)
{
```

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

```
        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;
```

Sanity checks. They have to be made because this function can be called as a result of writing to /proc/slabinfo .

```
        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;
                }
        }
```

Clear the ccupdate_struct_t struct. For every CPU on the system, allocate memory for the cpucache. The size of it is the size of the descriptor plus limit number of pointers to objects. The new cpucaches are stored in the new array where they will be swapped into the cache descriptor later by do_ccupdate_local().

```
        new.cachep = cachep;
        spin_lock_irq(&cachep->spinlock);
        cachep->batchcount = batchcount;
        spin_unlock_irq(&cachep->spinlock);

        smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
```

Fill in the rest of the struct and call smp_call_function_all_cpus which will make sure each CPU gets it's new cpucache.

```
        for (i = 0; i < smp_num_cpus; i++) {
                cpucache_t* ccold = new.new[cpu_logical_map(i)];
                if (!ccold)
                        continue;
                local_irq_disable();
                free_block(cachep, cc_entry(ccold), ccold->avail);
                local_irq_enable();
                kfree(ccold);
        }
```

The function do_ccupdate_local() swaps what is in the cache descriptor with the new cpucaches. This block cycles through all the old cpucaches and frees the memory.

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

3.5.4 Updating Per-CPU Information

When the per-cpu caches have been created or changed, each CPU has to be told about it. It's 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.

3.5.4.1 Function smp_function_all_cpus()

File: mm/slab.c
Prototype:

This calls the function `func` for all CPU's. In the context of the slab allocator, the function is do_ccupdate_local and the arguement is ccupdate_struct_t.

```
static void smp_call_function_all_cpus(void (*func) (void *arg),
                                       void *arg)
{
        local_irq_disable();
        func(arg);
        local_irq_enable();

        if (smp_call_function(func, arg, 1, 1))
                BUG();
}
```

This function is quiet simply. First it disable interrupts locally and call the function for this CPU. It then calls smp_call_function which makes sure that every other CPU executes the function `func`. In the context of the slab allocator, this will always be do_ccupdate_local.

3.5.4.2 Function do_ccupdate_local()

File: mm/slab.c
Prototype:

This function swaps the cpucache information in the cache descriptor with the information in `info` for this CPU.

```
static void do_ccupdate_local(void *info)
{
        ccupdate_struct_t *new = (ccupdate_struct_t *)info;
        cpucache_t *old = cc_data(new->cachep);
```

The parameter passed in is a pointer to the `ccupdate_struct_t` passed to `smp_call_function_all_cpus`. 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

```
        cc_data(new->cachep) = new->new[smp_processor_id()];
        new->new[smp_processor_id()] = old;
}
```

Place the new cpucache in cache descriptor. cc_data returns the pointer to the cpucache for this CPU. 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`

3.5.5 Draining a Per-CPU Cache

When a cache is been shrunk, it's 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 it's trouble.

3.5.5.1 Function drain_cpu_caches()

File: mm/slab.c
Prototype:

```
static void drain_cpu_caches(kmem_cache_t *cachep)
{
        ccupdate_struct_t new;
        int i;

        memset(&new.new,0,sizeof(new.new));

        new.cachep = cachep;

        down(&cache_chain_sem);
        smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
```

This block blanks out the new ccupdate_struct_t, acquires the cache chain semaphore and calls smp_call_function_cpus to get all the cpucache information for each cpu

```
        for (i = 0; i < smp_num_cpus; i++) {
                cpucache_t* ccold = new.new[cpu_logical_map(i)];
                if (!ccold || (ccold->avail == 0))
                        continue;
                local_irq_disable();
                free_block(cachep, cc_entry(ccold), ccold->avail);
                local_irq_enable();
                ccold->avail = 0;
        }
```

All the objects in each CPU are freed and the cpucache struct updated to show that there is no available objects in it

```
        smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
        up(&cache_chain_sem);
}
```

All the cpucaches have been updated so call smp_call_function_all_cpus to place them all back in the cache descriptor again and release the cache chain semaphore.

3.6 Slab Allocator Initialization

The first function called from *start_kernel* is **kmem_cache_init()**. This takes the following very simple steps

- Initialize a mutex for access to the cache chain
- Initialize the linked list for the cache chain
- Initialize the cache_cache
- Sets the cache_cache colour

The term *cache chain* is simply a fancy name for a circular linked list of caches the slab allocator knows about. It then goes on to initialize a cache of caches called **kmem_cache**. This is a cache of objects of type **kmem_cache_t** which describes information about the cache itself.

3.6.1 Initializing cache_cache

This cache is initialized as follows

```
static kmem_cache_t cache_cache = {
slabs_full:    LIST_HEAD_INIT(cache_cache.slabs_full),
slabs_partial: LIST_HEAD_INIT(cache_cache.slabs_partial),
slabs_free:    LIST_HEAD_INIT(cache_cache.slabs_free),
objsize:       sizeof(kmem_cache_t),
flags:         SLAB_NO_REAP,
spinlock:      SPIN_LOCK_UNLOCKED,
colour_off:    L1_CACHE_BYTES,
name:          "kmem_cache",
};
```

slabs_full	Standard list init
slabs_partial	Standard list init
slabs_free	Standard list init
objsize	Size of the struct. See the kmem_cache_s struct
flags	Make sure this cache can't be reaped
spinlock	Initialize unlocked
colour_off	Align the objects to the L1 Cache
name	Name of the cache

3.6.1.1 Function kmem_cache_init()

File: mm/slab.c
Prototype:

```
void __init kmem_cache_init(void)
{
        size_t left_over;

        init_MUTEX(&cache_chain_sem);
        INIT_LIST_HEAD(&cache_chain);

        kmem_cache_estimate(0, cache_cache.objsize, 0,
                        &left_over, &cache_cache.num);
        if (!cache_cache.num)
                BUG();

        cache_cache.colour = left_over/cache_cache.colour_off;
```

```
        cache_cache.colour_next = 0;
}
```

- Initialise the cache chain linked list
- Initialise the semaphore for access the cache chain
- This estimates the number of objects and amount of bytes wasted. See Section 3.1.5.1
- Calculate the cache_cache colour

3.7 Interfacing with the Buddy Allocator

The slab allocator doesn't come with pages attached, it must ask the physical page allocator for it's pages. For this two interfaces are provided, kmem_getpages and kmem_freepages. They are basically wrappers around the buddy allocators API so that slab flags will be taken into account for allocations

3.7.0.1 Function kmem_getpages()

File: mm/slab.c
Prototype:
This allocates pages for the slab allocator

```
static inline void * kmem_getpages (kmem_cache_t *cachep, unsigned long
flags)
{
        void    *addr;
        flags |= cachep->gfpflags;
```

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

```
        addr = (void*) __get_free_pages(flags, cachep->gfporder);
        return addr;
}
```

Call the buddy allocator and return the pages or NULL if it failed

3.7.0.2 Function kmem_freepages()

File: mm/slab.c
Prototype:

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

```
static inline void kmem_freepages (kmem_cache_t *cachep, void *addr)
{
        unsigned long i = (1<<cachep->gfporder);
        struct page *page = virt_to_page(addr);
```

The original order for the allocation is stored in the cache descriptor. The physical page allocator expects a struct page which virt_to_page provides.

```
        while (i--) {
                PageClearSlab(page);
                page++;
        }
```

Clear the PG_slab bit for each page

```
        free_pages((unsigned long)addr, cachep->gfporder);
}
```

Call the buddy allocator

3.8 Sizes Cache

Linux keeps two sets of caches for small memory allocations. One suitable for use with DMA and the other suitable for normal use. The human readable names for these caches **size-X cache** and **size-X(DMA) cache** viewable from `/proc/cpuinfo`. Information for each sized cache is stored in a **cache_sizes_t** struct defined in *mm/slab.c*

```
typedef struct cache_sizes {
        size_t           cs_size;
        kmem_cache_t    *cs_cachep;
        kmem_cache_t    *cs_dmacachep;
} cache_sizes_t;
```

cs_size The size of the memory block

cs_cachep The cache of blocks for normal memory use

cs_dmacachep The cache of blocks for use with DMA

kmem_cache_sizes_init() is called to create a set of caches of different sizes. On a system with a page size of 4096, the smallest chunk is 32 bytes, otherwise it is 64 bytes. Two caches will be created for every size, both of them cacheline-aligned, and one suitable for ISA DMA. So the smallest caches of memory are called *s*ize-32 and *s*ize-32(DMA). Caches for each subsequent power of two will be created until two caches of size of 131072 bytes are created. These will be used by *kmalloc* later.

```
static cache_sizes_t cache_sizes[] = {
#if PAGE_SIZE == 4096
        {    32,        NULL, NULL},
#endif
        {    64,        NULL, NULL},
        {   128,        NULL, NULL},
        {   256,        NULL, NULL},
        {   512,        NULL, NULL},
        {  1024,        NULL, NULL},
        {  2048,        NULL, NULL},
        {  4096,        NULL, NULL},
        {  8192,        NULL, NULL},
        { 16384,        NULL, NULL},
        { 32768,        NULL, NULL},
        { 65536,        NULL, NULL},
        {131072,        NULL, NULL},
        {     0,        NULL, NULL}
```

As is obvious, this is a statis array that is zero terminated consisting of buffers of succeeding powers of 2 from 2^5 to 2^{17} . An array now exists that describes each sized cache which must be initialised with caches at system startup.

3.8.1 kmalloc

With the existance 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. All the hard work is in cache allocation (See Section ??

```
void * kmalloc (size_t size, int flags)
{
cache_sizes_t *csizep = cache_sizes;

for (; csizep->cs_size; csizep++) {
    if (size > csizep->cs_size)
        continue;
    return __kmem_cache_alloc(flags & GFP_DMA ?
        csizep->cs_dmacachep : csizep->cs_cachep,
                                    flags);
}
return NULL;
}
```

Go through all the available sizes until a cache is found that holds sizes large enough for this allocation, then call __kmem_cache_alloc() to allocate from the cache as normal.

3.8.2 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 3.3.4)

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

if (!objp)
    return;
local_irq_save(flags);

/* CHECK_PAGE makes sure this is a slab cache. */
CHECK_PAGE(virt_to_page(objp));

/* The struct page list stores the
 * pointer to the kmem_cache_t */
```

```
c = GET_PAGE_CACHE(virt_to_page(objp));

__kmem_cache_free(c, (void*)objp);
local_irq_restore(flags);
}
```

Chapter 4

Non-Contiguous Memory Allocation

The vmalloc interface provides us with functions to map non-contiguous page frames into contiguous virtual memory pages. The free virtual memory addresses in the kernel space are used for this purpose. As mentioned previously in page 4 with regard to the significance of PAGE_OFFSET, the top 1GB address space is used by the kernel to map all the available physical memory. After the mapping, there usually is a lot of space left. Eg. taking my system having 192MB RAM as an example, all the RAM is directly mapped from PAGE_OFFSET to PAGE_OFFSET + 192MB. So out of the total of 1GB address space, we are only using 192MB. The remaining 832MB (1024 - 192) of virtual address space can now be used by the vmalloc interface. To account for cases where there is more physical memory than 1GB, some memory is reserved. At the moment 128MB is being reserved (see page 11) due to which the size of the normal zone is 896MB.

These allocations start from `VMALLOC_START` which is the end of directly mapped physical memory + a gap of 8MB (`VMALLOC_OFFSET`) which is just a safety net. To describe these memory areas, the following structure is used:

4.1 Structures

4.1.1 struct vm_struct

```
struct vm_struct {
     unsigned long flags;
     void * addr;
     unsigned long size;
```

```
    struct vm_struct * next;
};
```

flags Used to specify how this area was allocated, through vmalloc() itself or ioremap().

addr The starting virtual address of this allocation.

size The size of the allocation + 4k (padding between two areas).

next Used to link up all the structures.

These non-contiguous memory area descriptors are chained together on a list whose head is pointed to by `vmlist`. The vmalloc interface is contained in the file *mm/vmalloc.c* and provides functions for allocation, de-allocation, reading, writing etc.

4.2 Allocation

4.2.1 Function vmalloc()

Prototypes:

```
void * vmalloc (unsigned long size)
```

vmalloc itself just takes size as a parameter and is a front end for the lower layers.

```
return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM,
                                        PAGE_KERNEL);
```

It makes sure that pages are allocated for the kernel and protects the pages from been swapped out by accident by setting the PAGE_KERNEL flag.

4.2.2 Function __vmalloc()

Prototypes:

```
void * __vmalloc (unsigned long size,
                  int gfp_mask,
                  pgprot_t prot)
```

This does the real work of the allocation. Pages allocated will not be contiguous in physical memory, only in the linear address space. Do not call this function directly. Use vmalloc which will call with the correct flags and protection.

```
void * addr;
struct vm_struct *area;

size = PAGE_ALIGN(size);
```

size is rounded to a multiple of page size (if size = 3440 Bytes, make it 4k).

```
if (!size || (size >> PAGE_SHIFT) > num_physpages) {
    BUG();
    return NULL;
}
```

If the size is 0 or the request is larger than the number of physical frames, fail the allocation.

```
area = get_vm_area(size, VM_ALLOC);
if (!area)
    return NULL;
```

The function `get_vm_area()` allocates a block of linear addresses that can fit the allocation and returns a *struct vm_struct*. Refer section 4.2.3

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

The function `vmalloc_area_pages()` begins the work of allocating the PMD, PTE's and finally the physical pages for the allocation (described in section 4.2.4).

```
return addr;
```

Return the virtual address.

4.2.3 Function get_vm_area()

Prototypes:

```
struct vm_struct * get_vm_area(unsigned long size,
                               unsigned long flags)
```

This is a helper function for vmalloc to find a block of linear addresses large enough to accommodate the size being allocated.

```
unsigned long addr;
struct vm_struct **p, *tmp, *area;

area = (struct vm_struct *)kmalloc(sizeof(*area), GFP_KERNEL);
if (!area)
    return NULL;
```

First the slab allocator is called to allocate a piece of memory to store information about the non-contiguous area.

```
size += PAGE_SIZE;
addr = VMALLOC_START;
```

The size is incremented by PAGE_SIZE to give a gap mentioned at the beginning of the section between each allocated area. addr is initially set to VMALLOC_START in case this is the first area to be allocated.

```
write_lock(&vmlist_lock);
for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
    if ((size + addr) < addr)
        goto out;
    if (size + addr <= (unsigned long) tmp->addr)
        break;
    addr = tmp->size + (unsigned long) tmp->addr;
    if (addr > VMALLOC_END-size)
        goto out;
}
```

First the list is locked to protect the list. Then the vmlist is stepped through and the checks are made as followed.

- Has we wrapped around the address space and overflowed ?

- If our allocation fits here, stop we found a place.
- Move addr to the end of the current vm_struct and make sure we are not past VMALLOC_END.

If either check one or three fail, the label `out` is reached.

```
area->flags = flags;
area->addr = (void *)addr;
area->size = size;
area->next = *p;
*p = area;
write_unlock(&vmlist_lock);
return area;
```

A satisfactory area was found. We can insert the area into the list, and return the address.

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

If we came here, we were unable to find a suitable area. So free the lock, free the area we had assigned and return failure.

4.2.4 Function vmalloc_area_pages()

Prototypes:

```
int vmalloc_area_pages (unsigned long address,
                        unsigned long size,
                        int gfp_mask, pgprot_t prot)
```

This function begins doing the grunt work of assigning the linear space needed for the allocation. It will allocate a PMD for each PGD entry that is needed to cover the full linear space for this allocation.

```
pgd_t * dir;
unsigned long end = address + size;
int ret;

dir = pgd_offset_k(address);
spin_lock(&init_mm.page_table_lock);
```

dir is set to be the first PGD entry for the kernel page tables and then the mm for the kernel is locked.

```
do {
    pmd_t *pmd;

    pmd = pmd_alloc(&init_mm, dir, address);
    ret = -ENOMEM;
    if (!pmd)
        break;
```

This simply tries to allocate a PMD block for the address as it currently is. If more than one PMD is required for the allocation, it will be allocated during the next iteration of the while loop.

```
    ret = -ENOMEM;

    if (alloc_area_pmd(pmd, address, end - address,
                                    gfp_mask, prot))
        break;
```

This ret to -ENOMEM is dead code. alloc_area_pmd is discussed in the section 4.2.5.

```
    address = (address + PGDIR_SIZE) & PGDIR_MASK;
    dir++;
    ret = 0;
} while (address && (address < end));
```

This prepares to move to the next PGD if the amount of memory to be allocated is larger than what one PGD can address and then cycles through allocating PMD and PTE's again.

```
spin_unlock(&init_mm.page_table_lock);
flush_cache_all();
return ret;
```

Free the lock and return back success or failure to __vmalloc.

4.2.5 Function alloc_area_pmd()

Prototypes:

```
int alloc_area_pmd(pmd_t * pmd, unsigned long address,
                   unsigned long size, int gfp_mask,
                   pgprot_t prot)
```

This function is responsible for stepping through all the PMD's required for this allocation and calling alloc_area_pte to assign enough PTE's for each PMD.

```
unsigned long end;

address &= ~PGDIR_MASK;
end = address + size;
if (end > PGDIR_SIZE)
    end = PGDIR_SIZE;
```

This is basic sanity checking and making sure the address has the lower bits cleared so that the address is aligned to a PGD.

```
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;
```

This allocates a PTE for each PMD entry required for this allocation. First it allocates the actual PTE entry and alloc_area_pte is responsible for finding page frames for each of the entries. Once they are allocated, the address is incremented, making sure it is aligned to a PMD entry.

4.2.6 Function alloc_area_pte()

Prototypes:

```
int alloc_area_pte (pte_t * pte, unsigned long address,
                    unsigned long size, int gfp_mask,
                    pgprot_t prot)
```

This function is used to create the actual PTE entries.

```
unsigned long end;

address &= ~PMD_MASK;
end = address + size;

if (end > PMD_SIZE)
    end = PMD_SIZE;
```

This starts with the same sanity checks as alloc_area_pmd.

```
do {
    struct page * page;

    spin_unlock(&init_mm.page_table_lock);
    page = alloc_page(gfp_mask);
    spin_lock(&init_mm.page_table_lock);
```

This allocates a page frame for the PTE we are currently looking at. The page table lock is released because it's not required while a page is allocated via the buddy algorithm.

```
    if (!pte_none(*pte))
        printk(KERN_ERR
                 "alloc_area_pte: page already exists\n");
    if (!page)
        return -ENOMEM;
```

The first check is a sanity check. If the buddy algorithm returns a page that is swapped out or otherwise not present, there is something serious wrong.

```
    set_pte(pte, mk_pte(page, prot));
    address += PAGE_SIZE;
    pte++;
} while (address < end);

return 0;
```

This protects the page to make sure it is not swapped out or otherwise interfered with. Then the next PTE is moved to so it will be allocated before returning success.

4.3 De-Allocation

4.3.1 Function vfree()

Prototypes:

```
void vfree(void * addr)
```

This function takes the base address. It must be page aligned and the one returned by vmalloc earlier. It cycles through the vm_structs and ultimately deallocate all the PMD's, PTE's and page frames previously allocated.

```
struct vm_struct **p, *tmp;

if (!addr)
    return;

if ((PAGE_SIZE-1) & (unsigned long) addr) {
    printk(KERN_ERR
             "Trying to vfree() bad address (%p)\n", addr);
    return;
}

write_lock(&vmlist_lock);
```

This is basic sanity checking. The first is to make sure a NULL address wasn't passed in and the second one is to make sure the address is page aligned as all allocations should have been made on a page boundary. The vmlist is then locked to protect it.

```
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        *p = tmp->next;
        vmfree_area_pages(VMALLOC_VMADDR(tmp->addr)
                                         ,tmp->size);
        write_unlock(&vmlist_lock);
        kfree(tmp);
        return;
```

```
    }
}
```

This block searches through the vmlist until the correct vm_struct is found for this area. Once it's found, vmfree_area_pages is called which steps through the page tables in the same fashion vmalloc_area_pages did.

```
write_unlock(&vmlist_lock);
printk(KERN_ERR "Trying to vfree() nonexistent vm area (%p)",
                                                        addr);
```

If the area is not found, the vmlist is unlocked and an error message is printed before returning.

4.3.2 Function vmfree_area_pages()

Prototypes:

```
void vmfree_area_pages(unsigned long address,
                       unsigned long size)

pgd_t * dir;
unsigned long end = address + size;

dir = pgd_offset_k(address);
```

This just sets dir to be the first PGD entry for the address.

```
flush_cache_all();
```

This has no effect on the x86, but in some architectures, the CPU cache has to be explicitly told to flush itself.

```
do {
    free_area_pmd(dir, address, end - address);
    address = (address + PGDIR_SIZE) & PGDIR_MASK;
    dir++;
} while (address && (address < end));
```

For each PGD that is used by this allocation, call free_area_pmd() on it so that that all the PTE's and page frames allocated can be freed. Afterwards move the address on making sure it is aligned to a PGD.

```
flush_tlb_all();
```

At this point, the page tables look very different to the TLB is invalid and needs to be flushed before returning back.

4.3.3 Function free_area_pmd()

Prototypes:

```
void free_area_pmd(pgd_t * dir,
                   unsigned long address,
                   unsigned long size)

pmd_t * pmd;
unsigned long end;

if (pgd_none(*dir))
    return;
if (pgd_bad(*dir)) {
    pgd_ERROR(*dir);
    pgd_clear(dir);
    return;
}
```

Some sanity checking. If the function is called with a missing PGD, it already has been freed. This could happen if an earlier vmalloc failed half way through and vfree had to be called on the whole linear area. pgd_bad makes sure the PGD about to be freed isn't either

- Not in main memory, which should never happen for vmalloc-ed memory.
- It's read only.
- It's marked as accessed or dirty.

```
pmd = pmd_offset(dir, address);
address &= ~PGDIR_MASK;
end = address + size;
if (end > PGDIR_SIZE)
    end = PGDIR_SIZE;
```

Set pmd to be the first PMD to be freed. Make sure address is PGD aligned and record what the end of this PGDIR is.

```
do {
    free_area_pte(pmd, address, end - address);
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
} while (address < end);
```

This goes through every PTE referenced by this PMD and calls free_area_pte on it so that the page frame can be freed.

4.3.4 Function free_area_pte()

Prototypes:

```
void free_area_pte(pmd_t * pmd,
                   unsigned long address,
                   unsigned long size)
```

free_area_pte is mainly sanity checking code to make sure a wrong page is not freed by accident.

```
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;
```

Similar sanity checks and principles to free_area_pmd.

```
do {
    pte_t page;
    page = ptep_get_and_clear(pte);
    address += PAGE_SIZE;
    pte++;
```

This is the beginning of the while loop which steps through every PTE we can reach from this PMD. ptep_get_and_clear retrieves the pte_t entry and then removes it from the page tables.

```
    if (pte_none(page))
        continue;
```

If it was not allocated because of a failed vmalloc or similar reason, continue on as normal.

```
    if (pte_present(page)) {
        struct page *ptpage = pte_page(page);
        if (VALID_PAGE(ptpage) && (!PageReserved(ptpage)))
            __free_page(ptpage);
        continue;
    }
```

If the page is present, get the struct page for this PTE and hand it back to the buddy allocator.

```
    printk(KERN_CRIT
           "Whee.. Swapped out page in kernel page table\n");
} while (address < end);
```

If the page was not present, it means it was swapped out which is a major screwup so start shouting blue murder. In the normal scheme of things, all the PTE's will be freed for this PMD and the function returns quietly.

4.4 Read/Write

The read and write functions appear to be provided for character devices so that they can read through memory that is vmalloc-ed in the same fashion as a normal read on a character device would take place.

4.4.1 Function vread()

Prototypes:

```
long vread(char *buf, char *addr,
           unsigned long count)
```

This reads an area of vmalloc-ed memory like a character device would. It does not have to read from a "valid" area. If the reader enters an area that is not in use, it will put 0's in the buf.

```
struct vm_struct *tmp;
char *vaddr, *buf_start = buf;
unsigned long n;

/* Don't allow overflow */
if ((unsigned long) addr + count < count)
    count = -(unsigned long) addr;
```

This overflow check is to make sure the caller doesn't try to read off the end of memory. If it would overflow, count is changed to just read to the end of memory.

```
read_lock(&vmlist_lock);
for (tmp = vmlist; tmp; tmp = tmp->next) {
    vaddr = (char *) tmp->addr;
    if (addr >= vaddr + tmp->size - PAGE_SIZE)
        continue;
```

This cycles through all the vmlists trying to find which vm_struct this address belongs to.

```
    while (addr < vaddr) {
        if (count == 0)
            goto finished;
        *buf = '\0';
        buf++;
        addr++;
        count--;
    }
```

Once we reach here, we have found the vm_struct we need but there is nothing to say that we are in a valid area to read from. If addr is not in a valid area, the buffer is zero filled until either count bytes has been read or that the vm_struct area is reached. This could happen for instance if someone tried to vread a large block of memory that crossed two vm_struct's.

```
    n = vaddr + tmp->size - PAGE_SIZE - addr;
    do {
        if (count == 0)
            goto finished;

        *buf = *addr;
        buf++;
        addr++;
        count--;
    } while (--n > 0);
}
```

Here we have reached a valid vm_struct so n is set to the number of bytes that can be read before the end of the area can be read. This is to prevent overflow. This block does a byte by byte read into buf until either count is reached or the next vm_struct needs to be read.

```
finished:
read_unlock(&vmlist_lock);
return buf - buf_start;
```

By here, all the bytes have been read or else there is no more vm_structs to read from. The lock is released and the number of bytes read is returned.

4.4.2 Function vwrite()

Prototypes:

```
long vwrite(char *buf, char *addr,
            unsigned long count)
```

This is virtually identical to vread except for two important differences.

- Bytes written that are not to valid areas are simply discarded silently.
- In valid areas, the vm_struct area is been written to rather than read from.

```
struct vm_struct *tmp;
char *vaddr, *buf_start = buf;
unsigned long n;

/* Don't allow overflow */
if ((unsigned long) addr + count < count)
    count = -(unsigned long) addr;

read_lock(&vmlist_lock);
for (tmp = vmlist; tmp; tmp = tmp->next) {
    vaddr = (char *) tmp->addr;
    if (addr >= vaddr + tmp->size - PAGE_SIZE)
        continue;
    while (addr < vaddr) {
        if (count == 0)
            goto finished;

        buf++;
        addr++;
        count--;
    }

    n = vaddr + tmp->size - PAGE_SIZE - addr;
    do {
        if (count == 0)
            goto finished;
        *addr = *buf;
        buf++;
        addr++;
                count--;
    } while (--n > 0);
}

finished:
read_unlock(&vmlist_lock);
return buf - buf_start;
```

Chapter 5

Process Virtual Memory Management

5.1 Structures

5.1.1 struct mm_struct

File: include/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;
```

```
    mm_context_t context;
};
```

mmap
A linked list of VMAs belonging to this address space sorted by address.

mm_rb
When the number of VMAs increase beyond a certain number, a red black tree is also used to access them. mm_rb points to the root node.

mmap_cache
Points to the last VMA accessed.

pgd
Is the Page Global Directory of the process.

mm_users
Number of process sharing this structure.

mm_count
Number of non-user references to it + 1 (for all the users).

map_count
Number of VMAs.

mmap_sem
Semaphore used to serialize access to this structure.

page_table_lock
Protects page tables and the rss field from concurrent access.

mmlist
List of all active mm's.These are globally strung together off init_mm.mmlist and are protected by mmlist_lock.

start_code
Points to the starting address of the code section.

end_code
Points to the end address of the code section.

start_data
Points to the starting address of the data section.

end_data
Points to the end address of the data section.

start_brk
Points to the start address of the heap area.

brk
Points to the end address of the heap area.

start_stack
Points to the start address of the stack.

arg_start
Points to the start address of the arguments.

arg_end
Points to the end address of the arguments.

env_start
Points to the start address of the environmet.

env_end
Points to the end address of the environment.

rss
Number of pages currently in memory.

total_vm
Total number of pages used by this process.

locked_vm
Number of pages locked by this process (ie. unswappable pages).

def_flags
The default flags for this address space.

cpu_vm_mask
A mask used to keep track of all the CPUs accessing this mm (and have TLB entries). Used for TLB shootdown.

swap_address
Used to store the last address swapped to disk. Set in `swap_out_pmd` and used by `swap_out_mm` to find the VMA being swapped out.

dumpable
This bit is used as a flag which controls the creation of a core dump.

context
Used to store segment information.

5.1.2 struct vm_area_struct

File: include/linux/mm.h

This struct defines a memory VMM memory area. There is one of these per VM-area/task. A VM area is any part of the process virtual memory space that has a special rule for the page-fault handlers (ie a shared library, the executable area etc).

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    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;
    struct vm_operations_struct * vm_ops;
    unsigned long vm_pgoff;
    struct file * vm_file;
    unsigned long vm_raend;
    void * vm_private_data;
};
```

vm_mm
The address space we belong to.

vm_start
Our start address within vm_mm.

vm_end
The first byte after our end address within vm_mm.

vm_next
Used to point to the next VMA in a list.

vm_page_prot
Access permissions of this VMA.

vm_flags
Various flags describing this memory area.

vm_rb
A rb tree used to contain all the VMAs for faster access when more in number.

vm_next_share
If this VMA is mapping a file, this field points to another VMA (different process), mapping (sharing) the same part of the file.

vm_pprev_share
Same function as above, but points to previous node in the list.

vm_ops
A set of functions to act on this memory region.

vm_pgoff
If we are mapping a file, this field gives us the offset within the file this region maps in terms of number of pages.

vm_file
If this memory region is mapping a file, this pointer is used to point to it (can be NULL).

vm_raend
Stores the file offset (from *vm_pgoff*) till which the data will be read, in the next read-ahead operation.

vm_private_data
Used by drivers to store their own data.

5.2 Creating a Process Address Space

5.2.1 Function copy_mm()

File: kernel/fork.c
Prototype:

```
int copy_mm(unsigned long clone_flags,
            struct task_struct * tsk)
```

This function is called from do_fork() to create a new process address space. The parameters of this function are:

clone_flags The flags with which *fork()* has been called with.

tsk The descriptor of the new task whose address space has to be created.

Depending on the various flags, the address space is either shared or duplicated.

```
    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;
```

The memory related counters in the task descriptor are initialised. Briefly, these counters are used as follows:

min_flt Counts the number of minor page faults (ie. a new page had to be allocated).

maj_flt Counts the number of major page faults (ie. when ever a page had to be loaded from the swap).

cmin_flt Counts the number of minor page faults of its children.

cmaj_flt Counts the number of major page faults of its children.

nswap Not used or updated anywhere, dead code.

cnswap Not used or updated anywhere, dead code.

```
    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;
```

The *current* task is the parent of the task being created. So get a pointer to its memory descriptor.

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

If the CLONE_VM flag is set, then the new process shares the same memory descriptor. So increment the counter *mm_users* of the mm_struct and goto good_mm where it is assigned to the new process.

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

If we came here, we need to create a new mm_struct, so call `allocate_mm()` which returns a new descriptor from the slab cache (mm_cachep).

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

Next we copy the mm_struct of parent to the newly created descriptor. Then we initialize some of its fields by calling `mm_init()` which is discussed further in section **??**.

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

The function `init_new_context()` is a no-op on i386.

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

    if (retval)
        goto free_pt;
```

Then we call `dup_mmap()` to initialize the rest of the fields and also copy the memory region descriptors (vm_area_struct). It is covered in section 5.2.2.

```
/*
* child gets a private LDT (if there was an LDT in the parent)
*/
    copy_segments(tsk, mm);
```

If the parent task has an LDT (Local Descriptor Table), it is copied to the new memory descriptor.

```
good_mm:
    tsk->mm = mm;
    tsk->active_mm = mm;
    return 0;
```

We come here when the CLONE_VM flag is set. We just point to (use) the same memory descriptor as the parent.

```
free_pt:
    mmput(mm);
```

We couldn't initialize the new mm_struct descriptor successfully, so deallocate it.

```
fail_nomem:
    return retval;
```

There is no memory available in the system, so return with an error.

5.2.2 Function dup_mmap()

File: kernel/fork.c
Prototype:

```
int dup_mmap(struct mm_struct * mm)
```

This function is called to initialize some fields and memory region descriptors of a mm_struct.

```
struct vm_area_struct * mpnt, *tmp, **pprev;
int retval;

flush_cache_mm(current->mm);
```

This function is used to flush all pages belonging to the given *mm* from the cache. This function is a no-op on the i386.

```
mm->locked_vm = 0;
mm->mmap = NULL;
mm->mmap_cache = NULL;
mm->map_count = 0;
mm->rss = 0;
mm->cpu_vm_mask = 0;
mm->swap_address = 0;
pprev = &mm->mmap;
```

Basic initialization.

```
/*
* Add it to the mmlist after the parent.
* Doing it this way means that we can order the list,
* and fork() won't mess up the ordering significantly.
* Add it first so that swapoff can see any swap entries.
*/
spin_lock(&mmlist_lock);
list_add(&mm->mmlist, &current->mm->mmlist);
mmlist_nr++;
spin_unlock(&mmlist_lock);
```

We add this new structure to the global list of address spaces immediately after its parents address space. Then we increment the *mmlist_nr* counter which keeps track of the number of address spaces in the list. Access to this list is protected by `mmlist_lock`.

```
for (mpnt = current->mm->mmap ; mpnt ; mpnt = mpnt->vm_next) {
    struct file *file;

    retval = -ENOMEM;
    if(mpnt->vm_flags & VM_DONTCOPY)
        continue;
```

Next we go through the list of VMAs of the parent process and duplicate them in the child's address space.

First we check whether the VMA has the VM_DONTCOPY flag set which protects it from being copied. If it has, then we skip this VMA and continue with the next.

```
    tmp = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!tmp)
        goto fail_nomem;
```

We get a new `vm_area_struct` from the slab cache.

```
*tmp = *mpnt;
tmp->vm_flags &= ~VM_LOCKED;
tmp->vm_mm = mm;
tmp->vm_next = NULL;
```

We copy the parents vma to the child's newly allocated vma. Then we reset the VM_LOCKED flag of the child. Initialize its fields *vm_mm* to point to the child's address space and *vm_next* with NULL (as it may be the last node in the list).

```
file = tmp->vm_file;
if (file) {
    struct inode *inode = file->f_dentry->d_inode;
    get_file(file);
    if (tmp->vm_flags & VM_DENYWRITE)
        atomic_dec(&inode->i_writecount);
```

If the vma we are copying was mapping a file, the file related fields must also be initialized. After we confirm that we are indeed mapping a file, we get a reference to its inode. We then call the function `get_file` on the *file* to increment its counter of number of mappings.

Simultaneous read-write and read-only support is not available at the moment. So if the flag VM_DENYWRITE is set, its a read-only mapping else its read-write. The number of readers or writers on the file mapping is kept track of by the inode's *i_writecount* field. If its a read-only mapping, its value is decremented else it is incremented. So by looking at *i_writecount* we can know whether the mapping is read-only (negative) or read-write (positive).

```
/* insert tmp into the share list, just after mpnt */
        spin_lock(&inode->i_mapping->i_shared_lock);
        if((tmp->vm_next_share = mpnt->vm_next_share) != NULL)
            mpnt->vm_next_share->vm_pprev_share =
                                        &tmp->vm_next_share;
        mpnt->vm_next_share = tmp;
        tmp->vm_pprev_share = &mpnt->vm_next_share;
```

```
        spin_unlock(&inode->i_mapping->i_shared_lock);
    }



/*
* Link in the new vma and copy the page table entries:
* link in first so that swapoff can see swap entries.
*/
    spin_lock(&mm->page_table_lock);
    *pprev = tmp;
    pprev = &tmp->vm_next;
    mm->map_count++;
```

We now add the VMA to the mmap list and also increment the counter.

```
    retval = copy_page_range(mm, current->mm, tmp);
    spin_unlock(&mm->page_table_lock);
```

Next we call `copy_page_range()` to copy the page table entries.

```
    if (tmp->vm_ops && tmp->vm_ops->open)
        tmp->vm_ops->open(tmp);

    if (retval)
        goto fail_nomem;
}
```

If there is an open() function defined for this memory region (to perform any initializations), we call it.

```
retval = 0;
build_mmap_rb(mm);
```

Next we call `build_mmap_rb()` which creates a red-black tree with the VMAs for faster searches.

```
fail_nomem:
flush_tlb_mm(current->mm);
return retval;
```

Then we flush the TLB.

5.3 Deleting a Process Address Space

5.3.1 Function exit_mm()

File: kernel/exit.c
Prototypes:

```
void exit_mm(struct task_struct * tsk)
void __exit_mm(struct task_struct * tsk)
```

This function is called from `do_exit()` whenever a process exits, to delete its address space.

```
struct mm_struct * mm = tsk->mm;

mm_release();
```

The function `mm_release()` is only called to notify the parent about the death of its child if the child was created via *vfork()*.

```
if (mm) {
    atomic_inc(&mm->mm_count);
    BUG_ON(mm != tsk->active_mm);
```

We check to see if mm is still valid (not yet dropped) and then increment its *mm_count* to stop it being dropped from under us. Also *mm* and *active_mm* needs to be the same.

```
/* more a memory barrier than a real lock */
    task_lock(tsk);
    tsk->mm = NULL;
    task_unlock(tsk);
    enter_lazy_tlb(mm, current, smp_processor_id());
```

Since we are about to modify the task structure, we take a lock on it. Then we remove the *mm*'s reference from the task structure. After unlocking the task struct, `enter_lazy_tlb()` is called which is a no-op on a uni-processor.

```
    mmput(mm);
}
```

Finally `mmput()` is called to actually destroy *mm_struct*.

5.3.2 Function mmput()

File: kernel/fork.c
Prototype:

```
void mmput(struct mm_struct *mm)
```

This function is used to de-allocate various resources held by the `mm_struct` and then drop it.

```
if (atomic_dec_and_lock(&mm->mm_users, &mmlist_lock)) {
```

We can drop a `mm_struct` only if the number of users sharing this is 1. So the above line decrements *mm_users* and if it becomes 0, locks the structure.

```
    extern struct mm_struct *swap_mm;
    if (swap_mm == mm)
        swap_mm = list_entry(mm->mmlist.next,
                             struct mm_struct, mmlist);
```

The global `swap_mm` is used to point to the *mm_struct* that is going to be swapped out next. Here in the above code we test to see if *swap_mm* is the same *mm* we are dropping. If it is, then we update *swap_mm* to point to the next *mm* on the *mm_list*.

```
    list_del(&mm->mmlist);
    mmlist_nr--;
    spin_unlock(&mmlist_lock);
```

Next we remove the mm_struct from the global *mm_list*, decrement the *mmlist_nr* counter and unlock the spinlock on mm_list which was locked previously in the call to `atomic_dec_and_lock()`.

```
    exit_mmap(mm);
```

We call `exit_mmap()` to do the actual release of all the memory.

```
    mmdrop(mm);
}
```

Lastly `mmdrop` is called to release the *mm_struct* to the slab allocator.

5.3.3 Function exit_mmap()

File: mm/mmap.c
Prototype:

```
void exit_mmap(struct mm_struct * mm)
```

This function does all the grunt work of releasing all the resources from the given *mm_struct*.

```
struct vm_area_struct * mpnt;

release_segments(mm);
```

If this address space has an associated LDT, it is freed.

```
spin_lock(&mm->page_table_lock);
mpnt = mm->mmap;
mm->mmap = mm->mmap_cache = NULL;
mm->mm_rb = RB_ROOT;
mm->rss = 0;
spin_unlock(&mm->page_table_lock);
mm->total_vm = 0;
mm->locked_vm = 0;
```

Next we reset most of the variables (probably because it will be re-used by the slab allocator).

```
flush_cache_mm(mm);
```

The above function is called to flush the caches (L1 and L2). This function on an i386 is a no-op.

```
while (mpnt) {
    struct vm_area_struct * next = mpnt->vm_next;
    unsigned long start = mpnt->vm_start;
    unsigned long end = mpnt->vm_end;
    unsigned long size = end - start;
```

Then we start going through each of the VMAs.

```
    if (mpnt->vm_ops) {
        if (mpnt->vm_ops->close)
            mpnt->vm_ops->close(mpnt);
    }
```

If there is a *vm_ops* defined, then call the close operation on the memory region.

```
    mm->map_count--;
    remove_shared_vm_struct(mpnt);
    zap_page_range(mm, start, size);
```

We decrement the number of VMAs counter, *map_count* and remove the VMA from the list of shared mappings if it is mapping a file. Then the call to `zap_page_range()` will remove all the page table entries covered by this VMA.

```
    if (mpnt->vm_file)
        fput(mpnt->vm_file);
    kmem_cache_free(vm_area_cachep, mpnt);
    mpnt = next;
}
```

If we were mapping a file, then `fput()` is called to decrement the number of users count of the file and if it becomes 0, then drop the file structure. Then we release the VMA to the slab allocator and continue with the rest of the VMAs.

```
flush_tlb_mm(mm);
```

Then we flush the TLB cache.

```
/* This is just debugging */
if (mm->map_count)
    BUG();

clear_page_tables(mm, FIRST_USER_PGD_NR, USER_PTRS_PER_PGD);
```

Lastly, all the page directory and page midle directory entries are cleared.

5.4 Allocating a Memory Region

5.4.1 Function do_mmap()

File: include/linux/mm.h
Prototype:

```
unsigned long do_mmap(struct file *file,
                      unsigned long addr,
                      unsigned long len,
                      unsigned long prot,
                      unsigned long flag,
                      unsigned long offset)
```

This function is used to create a new memory region for a process. Its parameters are:

file
File descriptor of the file being mapped.

addr
Preferred address where this mapping should start from.

len
Size of the mapping.

prot
Protection flags of the pages in this region (defined in include/asm-i386/mman.h).

PROT_READ
Pages can be read.

PROT_WRITE
Pages can be written.

PROT_EXEC
Pages can be executed.

PROT_NONE
Pages can not be accessed.

flag
Used to specify the type of the mapping. The various flags are (defined in include/asm-i386/mman.h):

MAP_FIXED
Do not select a different address than the one specified. If the specified address cannot be used, mmap will fail. If MAP_FIXED is specified, start must be a multiple of the pagesize.

MAP_SHARED
Share this mapping with all other processes that map this object. Storing to the region is equivalent to writing to the file.

MAP_PRIVATE
Create a private copy-on-write mapping. Stores to the region do not affect the original file.

MAP_DENYWRITE
This region maps a file read-only.

MAP_NORESERVE
If set, we don't check if there is enough memory for the allocation (overcommit).

MAP_ANONYMOUS
No file is associated with this memory region.

MAP_GROWSDOWN
This mapping can expand towards lower addresses (eg. stack).

MAP_EXECUTABLE
Mapping contains executable code.

MAP_LOCKED
Pages in this mapping are locked and cannot be swapped out.

offset
Offset within the file the mapping is going to start.

```
unsigned long ret = -EINVAL;

if ((offset + PAGE_ALIGN(len)) < offset)
    goto out;
```

We check for an overflow.

```
if (!(offset & ~PAGE_MASK))
    ret = do_mmap_pgoff(file, addr, len, prot,
```

```
                        flag, offset >> PAGE_SHIFT);
out:
return ret;
```

After checking that the offset is page aligned, we call `do_mmap_pgoff()` to do the real work of allocating the VMA.

5.4.2 Function do_mmap_pgoff()

File: mm/mmap.c
Prototype:

```
unsigned long do_mmap_pgoff(struct file * file,
                            unsigned long addr,
                            unsigned long len,
                            unsigned long prot,
                            unsigned long flags,
                            unsigned long pgoff)
```

This function does the actual work of creating a VMA.

```
struct mm_struct * mm = current->mm;
struct vm_area_struct * vma, * prev;
unsigned int vm_flags;
int correct_wcount = 0;
int error;
rb_node_t ** rb_link, * rb_parent;

if (file && (!file->f_op || !file->f_op->mmap))
    return -ENODEV;
```

If we are trying to map a file, *file*, $file \rightarrow f\_op$ and $file \rightarrow f\_op \rightarrow mmap$ should not be NULL.

```
if ((len = PAGE_ALIGN(len)) == 0)
    return addr;

if (len > TASK_SIZE)
```

```
    return -EINVAL;
```

If the size of the request is 0 or if it exceeds the maximum limit of the process which is 3GB, it just returns with the appropriate error value.

```
/* offset overflow? */
if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
    return -EINVAL;

/* Too many mappings? */
if (mm->map_count > max_map_count)
    return -ENOMEM;
```

We check for an overflow and also see if we have not reached the limit of number of VMAs a process can have. The limit is currently 65536.

```
/* Obtain the address to map to. we verify
 * (or select) it and ensure that it represents
 * a valid section of the address space.
 */
addr = get_unmapped_area(file, addr, len, pgoff, flags);
if (addr & ~PAGE_MASK)
    return addr;
```

The function `get_unmapped_area()` returns the starting address of an unused address space big enough to hold the new memory region.

```
/* Do simple checking here so the lower-level
 * routines won't have to. we assume access
 * permissions have been handled by the open
 * of the memory object, so we don't do any here.
 */
vm_flags = calc_vm_flags(prot,flags) | mm->def_flags |
                VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
```

Then we get the new flags for the memory region by combining the *prot* and *flags* fields.

```
/* mlock MCL_FUTURE? */
if (vm_flags & VM_LOCKED) {
    unsigned long locked = mm->locked_vm << PAGE_SHIFT;
    locked += len;
    if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
        return -EAGAIN;
}
```

If one of the flags happens to be VM_LOCKED, then we check whether we are within the limit on the number of locked pages. As previously mentioned, $mm \rightarrow locked\_vm$ gives the number of pages already locked. So the above expression first converts it into the number of bytes and then adds it to *len*, thereby giving the total number of pages that are going to be locked. By default, the limit is infinity.

```
if (file) {
    switch (flags & MAP_TYPE) {
    case MAP_SHARED:
        if ((prot & PROT_WRITE) && !(file->f_mode & FMODE_WRITE))
            return -EACCES;

/* Make sure we don't allow writing to an append-only file.. */
        if (IS_APPEND(file->f_dentry->d_inode)
              && (file->f_mode & FMODE_WRITE))
            return -EACCES;

/* make sure there are no mandatory locks on the file. */
        if (locks_verify_locked(file->f_dentry->d_inode))
            return -EAGAIN;

        vm_flags |= VM_SHARED | VM_MAYSHARE;
        if (!(file->f_mode & FMODE_WRITE))
```

```
            vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
```

If we are mapping a file and its a shared mapping, then we check for the following conditions:

1. If the pages in the mapping can be written to, but the file was opened in read-only mode, return with an access error.
2. The comment says it all. We cannot write to a file that is opened in O_APPEND mode.
3. If there is a lock on the file, then return and let the user try again later.
4. Set the flags VM_SHARED and VM_MAYSHARE. Then if the file is read-only, the flags VM_MAYWRITE and VM_SHARED are reset.

```
    /* fall through */
    case MAP_PRIVATE:
        if (!(file->f_mode & FMODE_READ))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }
```

We cannot map a file privately which is not opened in read-only mode. So we just return an "Access Denied". The default action is to return the error "Invalid Value".

```
} else {
    vm_flags |= VM_SHARED | VM_MAYSHARE;
    switch (flags & MAP_TYPE) {
    default:
        return -EINVAL;
    case MAP_PRIVATE:
        vm_flags &= ~(VM_SHARED | VM_MAYSHARE);
```

```
        /* fall through */
    case MAP_SHARED:
        break;
    }
}
```

If we are not mapping a file, set the flags VM_SHARED and VM_MAYSHARE. Then if its a private mapping, we reset the flags which were set above else if it is a shared mapping, we do nothing as the correct flags were already set.

```
/* Clear old maps */
munmap_back:
vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
if (vma && vma->vm_start < addr + len) {
    if (do_munmap(mm, addr, len))
        return -ENOMEM;
    goto munmap_back;
}
```

The function `find_vma_prepare()` will return a VMA the given address lies in. If the address does not lie in any VMA, then it sets the value of *vma* to NULL. It also modifies the values of *rb_link* and *rb_parent* to point to the parent and link of the new VMA. So the above code checks if the given address lies in any VMA, if it does, it de-allocates that VMA to re-create it later with a bigger size.

```
/* Check against address space limit. */
if ((mm->total_vm << PAGE_SHIFT) + len
                  > current->rlim[RLIMIT_AS].rlim_cur)
    return -ENOMEM;
```

As already mentioned, *total_vm* gives the number of pages already allocated to this process. So we convert it into number of bytes , add the size of the new request and compare it with the address space limit.

```
/* Private writable mapping? Check memory availability.. */
if ((vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
                            !(flags & MAP_NORESERVE) &&
                    !vm_enough_memory(len >> PAGE_SHIFT))
    return -ENOMEM;
```

If all the three conditions are true, quit.

```
/* Can we just expand an old anonymous mapping? */
if (!file && !(vm_flags & VM_SHARED) && rb_parent)
    if (vma_merge(mm, prev, rb_parent, addr, addr + len, vm_flags))
        goto out;
```

If we are not mapping a file and its not a shared mapping and the parent node is not NULL, then we call `vma_merge()` to try to increase the size of parent node (VMA) to include this range of addresses also. If its successfull, we avoid creating a new VMA and jump to the end.

```
/* Determine the object being mapped and call the appropriate
 * specific mapper. the address has already been validated, but
 * not unmapped, but the maps are removed from the list.
 */
vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma)
    return -ENOMEM;
```

Get a new *vm_area_struct* from the slab allocator.

```
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = protection_map[vm_flags & 0x0f];
```

```
vma->vm_ops = NULL;
vma->vm_pgoff = pgoff;
vma->vm_file = NULL;
vma->vm_private_data = NULL;
vma->vm_raend = 0;
```

Initialize its members.

```
if (file) {
    error = -EINVAL;
    if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
        goto free_vma;
```

If we are mapping a file, it cannot grow, so we release the new VMA and quit.

```
    if (vm_flags & VM_DENYWRITE) {
        error = deny_write_access(file);
        if (error)
            goto free_vma;
        correct_wcount = 1;
    }
```

If the VM_DENYWRITE flag is set (ie. specifying a read-only region), we call `deny_write_access()` to decrement the $file \rightarrow f_dentry \rightarrow d_inode \rightarrow i_writecount$ counter. It returns 0 if successful else -ETXTBSY if file was already mapped read-write.

```
    vma->vm_file = file;
    get_file(file);
    error = file->f_op->mmap(file, vma);
    if (error)
        goto unmap_and_free_vma;
```

Then we assign the file to the VMA and call `get_file()` to increment the file counter *f_count*, which is used to keep track of the number of users mapping

this file. Finally the `mmap()` function of the file is called to do the actuall mapping.

```
} else if (flags & MAP_SHARED) {
    error = shmem_zero_setup(vma);
    if (error)
        goto free_vma;
}
```

If we are not mapping a file and MAP_SHARED flag is set, then its a shared anonymous mapping. We call `shmem_zero_setup()` to create an anonymous file in memory (shmfs) and assign it to this VMA's *vm_file* field.

```
/* Can addr have changed??
 *
 * Answer: Yes, several device drivers can do it in their
 *         f_op->mmap method. -DaveM
 */
if (addr != vma->vm_start) {

/*
 * It is a bit too late to pretend changing the virtual
 * area of the mapping, we just corrupted userspace
 * in the do_munmap, so FIXME (not in 2.4 to avoid breaking
 * the driver API).
 */
    struct vm_area_struct * stale_vma;

/* Since addr changed, we rely on the mmap op to prevent
 * collisions with existing vmas and just use find_vma_prepare
 * to update the tree pointers.
 */
    addr = vma->vm_start;
    stale_vma = find_vma_prepare(mm, addr, &prev,
                                 &rb_link, &rb_parent);
/*
 * Make sure the lowlevel driver did its job right.
 */
```

```
    if (unlikely(stale_vma && stale_vma->vm_start
                                < vma->vm_end)) {
        printk(KERN_ERR "buggy mmap operation: [<%p>]\n",
                        file ? file->f_op->mmap : NULL);
        BUG();
    }
}

vma_link(mm, vma, prev, rb_link, rb_parent);
if (correct_wcount)
    atomic_inc(&file->f_dentry->d_inode->i_writecount);
```

Since the VMA is now ready, it is added to the rbtree.

```
out:
mm->total_vm += len >> PAGE_SHIFT;
if (vm_flags & VM_LOCKED) {
    mm->locked_vm += len >> PAGE_SHIFT;
    make_pages_present(addr, addr + len);
}
return addr;
```

```
unmap_and_free_vma:
if (correct_wcount)
    atomic_inc(&file->f_dentry->d_inode->i_writecount);
vma->vm_file = NULL;
fput(file);

/* Undo any partial mapping done by a device driver. */
zap_page_range(mm, vma->vm_start, vma->vm_end - vma->vm_start);
```

xxx

```
free_vma:
kmem_cache_free(vm_area_cachep, vma);
return error;
```

5.4.3 Function get_unmapped_area()

File: mm/mmap.c
Prototype:

```
unsigned long
get_unmapped_area(struct file *file,
                  unsigned long addr,
                  unsigned long len,
                  unsigned long pgoff,
                  unsigned long flags)
```

This function is used to find a suitable address space for the mapping.

```
if (flags & MAP_FIXED) {
    if (addr > TASK_SIZE - len)
        return -ENOMEM;
    if (addr & ~PAGE_MASK)
        return -EINVAL;
    return addr;
}
```

If the MAP_FIXED flag is set, ie. the mapping has to start at the given address only, the following checks are made:

1. The start address + length of the mapping is not overflowing the process address space limit of TASK_SIZE (3GB).
2. The address must start on a page boundary.

If both conditions are met, the address is returned as is with no further checks.

```
if (file && file->f_op && file->f_op->get_unmapped_area)
    return file->f_op->get_unmapped_area(file, addr,
                                         len, pgoff, flags);
```

If we were mapping a file and the corresponding file operation functions are defined, we call its get_unmapped_area() operation.

```
return arch_get_unmapped_area(file,addr,len,pgoff,flags);
```

5.4.4 Function arch_get_unmapped_area()

File: mm/mmap.c
Prototype:

```
unsigned long
arch_get_unmapped_area(struct file *filp,
                       unsigned long addr,
                       unsigned long len,
                       unsigned long pgoff,
                       unsigned long flags)
```

This function is used to find a free address space which can hold the anonymous mapping of the given size. This function is a generic function and can be replaced by architecture specific code by defining HAVE_ARCH_UNMAPPED_AREA and using the same prototype for its implementation. This is done in *alpha, ia64* and *sparc* architectures.

```
struct vm_area_struct *vma;

if (len > TASK_SIZE)
    return -ENOMEM;
```

Check to see if the size is not greater than the available address space.

```
if (addr) {
    addr = PAGE_ALIGN(addr);
    vma = find_vma(current->mm, addr);
    if (TASK_SIZE - len >= addr &&
       (!vma || addr + len <= vma->vm_start))
        return addr;
```

```
}
```

If *addr* is non-zero, we align it to a page boundary. We call the function `find_vma()` to see if the given address is contained in an existing VMA. If it is not contained in any VMA and the end of the mapping is within the process address space, we return the address.

```
addr = PAGE_ALIGN(TASK_UNMAPPED_BASE);
```

No preferred address has been specified, so we start the search from the default start address of TASK_UNMAPPED_BASE (1GB).

```
for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
    if (TASK_SIZE - len < addr)
        return -ENOMEM;
    if (!vma || addr + len <= vma->vm_start)
        return addr;
    addr = vma->vm_end;
}
```

We repeat the exercise of finding whether the address is contained in any existing VMA or not. If it is not contained in any of them and the mapping will not overflow the process address space, we return the address else we continue the search. The loop exits when we find a suitable address or we run out of address space.

5.4.5 Function find_vma_prepare()

File: mm/mmap.c
Prototype:

```
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;
rb_prev = __rb_parent = NULL;
vma = NULL;

while (*__rb_link) {
    struct vm_area_struct *vma_tmp;

    __rb_parent = *__rb_link;
    vma_tmp = rb_entry(__rb_parent, struct vm_area_struct, vm_rb);

    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            return vma;
        __rb_link = &__rb_parent->rb_left;
    } else {
        rb_prev = __rb_parent;
        __rb_link = &__rb_parent->rb_right;
    }
}

*pprev = NULL;
if (rb_prev)
    *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
*rb_link = __rb_link;
*rb_parent = __rb_parent;
return vma;
```

5.4.6 Function vm_enough_memory()

File: mm/mmap.c
Prototype:

```
int vm_enough_memory(long pages)
```

This function is used to check that a process has enough memory to allocate a new virtual mapping.

```
unsigned long free;

/* Sometimes we want to use more memory than we have. */
if (sysctl_overcommit_memory)
    return 1;
```

The variable `sysctl_overcommit_memory` can be set through the */proc/sys/vm/overcommit_memory* interface. This value contains a flag that enables memory overcommitment. When this flag is 0, the kernel checks before each malloc() to see if there's enough memory left. If the flag is nonzero, the system pretends there's always enough memory.

```
/* The page cache contains buffer pages these days.. */
free = atomic_read(&page_cache_size);
free += nr_free_pages();
free += nr_swap_pages;
```

We start calculating the amount of free allocatable memory present in the system. The variable `page_cache_size` is the number of pages in the page cache hash table. The function `nr_free_pages()` returns the total number of free pages in all the three zones. The variable `nr_swap_pages` gives the number of pages that can be accomodated in the swap.

```
/*
 * This double-counts: the nrpages are both in the page-cache
 * and in the swapper space. At the same time, this
 * compensates for the swap-space over-allocation (ie
 * "nr_swap_pages" being too small.
 */
free += swapper_space.nrpages;
```

Adding the number of pages being used by the swap cache.

```
/*
 * The code below doesn't account for free space in the inode
 * and dentry slab cache, slab cache fragmentation, inodes and
 * dentries which will become freeable under VM load, etc.
 * Lets just hope all these (complex) factors balance out...
 */
free += (dentry_stat.nr_unused * sizeof(struct dentry))
                                          >> PAGE_SHIFT;
free += (inodes_stat.nr_unused * sizeof(struct inode))
                                          >> PAGE_SHIFT;

return free > pages;
```

Add the number of pages taken up by the dentry and inode slab caches. Returns 1 if the number of pages available is greater than the number of pages requested, else it returns 0.

5.5 De-Allocating a Memory Region

5.5.1 Function sys_munmap()

File: mm/mmap.c
Prototype:

```
long sys_munmap(unsigned long addr,
                size_t len)
```

This function is used to remove an existing mapping from the process address space.

```
int ret;
struct mm_struct *mm = current->mm;

down_write(&mm->mmap_sem);
ret = do_munmap(mm, addr, len);
up_write(&mm->mmap_sem);
return ret;
```

We lock the *mm_struct* and call `do_munmap()` which does the actual work of releasing the pages etc.

5.5.2 Function do_munmap()

File: mm/mmap.c
Prototype:

```
int do_munmap(struct mm_struct *mm,
              unsigned long addr,
              size_t len)
```

This function is responsible for deleting a memory region.

```
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;

/* Check if this memory area is ok - put it on the temporary
 * list if so..  The checks here are pretty simple --
 * every area affected in some way (by any overlap) is put
 * on the list.  If nothing is put on, nothing is affected.
 */
mpnt = find_vma_prev(mm, addr, &prev);
if (!mpnt)
    return 0;



/* we have  addr < mpnt->vm_end  */

if (mpnt->vm_start >= addr+len)
    return 0;
```

```
/* If we'll make "hole", check the vm areas limit */
if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
    && mm->map_count >= max_map_count)
    return -ENOMEM;
```

```
/*
 * We may need one additional vma to fix up the mappings ...
 * and this is the last chance for an easy error exit.
 */
extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!extra)
    return -ENOMEM;
```

```
npp = (prev ? &prev->vm_next : &mm->mmap);
free = NULL;
spin_lock(&mm->page_table_lock);
for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
    *npp = mpnt->vm_next;
    mpnt->vm_next = free;
    free = mpnt;
    rb_erase(&mpnt->vm_rb, &mm->mm_rb);
}
```

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

/* Ok - we have the memory areas we should free on the 'free' list,
 * so release them, and unmap the page range..
 * If the one of the segments is only being partially unmapped,
 * it will put new vm_area_struct(s) into the address space.
 * In that case we have to be careful with VM_DENYWRITE.
 */
```

```
while ((mpnt = free) != NULL) {
    unsigned long st, end, size;
    struct file *file = NULL;

    free = free->vm_next;

    st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
    end = addr+len;
    end = end > mpnt->vm_end ? mpnt->vm_end : end;
    size = end - st;

    if (mpnt->vm_flags & VM_DENYWRITE &&
      (st != mpnt->vm_start || end != mpnt->vm_end) &&
      (file = mpnt->vm_file) != NULL) {
        atomic_dec(&file->f_dentry->d_inode->i_writecount);
    }
    remove_shared_vm_struct(mpnt);
    mm->map_count--;

    zap_page_range(mm, st, size);

/*
 * Fix the mapping, and free the old area if it wasn't reused.
 */
    extra = unmap_fixup(mm, mpnt, st, size, extra);
    if (file)
        atomic_inc(&file->f_dentry->d_inode->i_writecount);
}
validate_mm(mm);

/* Release the extra vma struct if it wasn't used */
if (extra)
    kmem_cache_free(vm_area_cachep, extra);

free_pgtables(mm, prev, addr, addr+len);

return 0;
```

5.6 Modifying Heap

5.6.1 Function sys_brk()

File: mm/mmap.c
Prototype:

```
unsigned long sys_brk(unsigned long brk)
```

This is a system call which is used to manipulate the size of the heap of a process. The parameter `brk` specifies the new value of the end address of the data section. $current \rightarrow mm \rightarrow brk$.

```
unsigned long rlim, retval;
unsigned long newbrk, oldbrk;
struct mm_struct *mm = current->mm;

down_write(&mm->mmap_sem);
```

Since we are about to access/modify the structure representing the current process address space, we need to lock it by using the semaphore `mmap_sem`.

```
if (brk < mm->end_code)
    goto out;
```

The data section comes after the code section. The above check is used to see if the new value is invalid.

```
newbrk = PAGE_ALIGN(brk);
oldbrk = PAGE_ALIGN(mm->brk);
if (oldbrk == newbrk)
    goto set_brk;
```

If the new value of *brk* is the same as the old value, we just jump over all the checks and .....

```
/* Always allow shrinking brk. */
if (brk <= mm->brk) {
    if (!do_munmap(mm, newbrk, oldbrk-newbrk))
        goto set_brk;
    goto out;
}
```

xxxx

```
/* Check against rlimit.. */
rlim = current->rlim[RLIMIT_DATA].rlim_cur;
if (rlim < RLIM_INFINITY && brk - mm->start_data > rlim)
    goto out;
```

xxxx

```
/* Check against existing mmap mappings. */
if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
    goto out;
```

xxxx

```
/* Check if we have enough memory.. */
if (!vm_enough_memory((newbrk-oldbrk) >> PAGE_SHIFT))
    goto out;
```

xxxx

```
/* Ok, looks good - let it rip. */
if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
    goto out;
```

xxxx

```
set_brk:
mm->brk = brk;
out:
retval = mm->brk;
up_write(&mm->mmap_sem);
return retval;
```

5.6.2 Function do_brk()

File: mm/mmap.c
Prototype:

```
unsigned long do_brk(unsigned long addr,
                     unsigned long len)
```

```
struct mm_struct * mm = current->mm;
struct vm_area_struct * vma, * prev;
unsigned long flags;
rb_node_t ** rb_link, * rb_parent;

len = PAGE_ALIGN(len);
if (!len)
    return addr;

/*
 * mlock MCL_FUTURE?
 */
if (mm->def_flags & VM_LOCKED) {
    unsigned long locked = mm->locked_vm << PAGE_SHIFT;
    locked += len;
```

```
    if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
        return -EAGAIN;
}

/*
 * Clear old maps.  this also does some error checking for us
 */
 munmap_back:
vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
if (vma && vma->vm_start < addr + len) {
    if (do_munmap(mm, addr, len))
        return -ENOMEM;
    goto munmap_back;
}

/* Check against address space limits *after* clearing old maps... */
if ((mm->total_vm << PAGE_SHIFT) + len
    > current->rlim[RLIMIT_AS].rlim_cur)
    return -ENOMEM;

if (mm->map_count > max_map_count)
    return -ENOMEM;

if (!vm_enough_memory(len >> PAGE_SHIFT))
    return -ENOMEM;

flags = VM_DATA_DEFAULT_FLAGS | mm->def_flags;

/* Can we just expand an old anonymous mapping? */
if (rb_parent && vma_merge(mm, prev,
              rb_parent, addr, addr + len, flags))
    goto out;

/*
 * create a vma struct for an anonymous mapping
 */
vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma)
    return -ENOMEM;

vma->vm_mm = mm;
```

```
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = flags;
vma->vm_page_prot = protection_map[flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = 0;
vma->vm_file = NULL;
vma->vm_private_data = NULL;

vma_link(mm, vma, prev, rb_link, rb_parent);

out:
mm->total_vm += len >> PAGE_SHIFT;
if (flags & VM_LOCKED) {
    mm->locked_vm += len >> PAGE_SHIFT;
    make_pages_present(addr, addr + len);
}
return addr;
```

5.7 Unclassified

5.7.1 Function __remove_shared_vm_struct()

File: mm/mmap.c
Prototype:

```
void __remove_shared_vm_struct(struct vm_area_struct *vma)


struct file * file = vma->vm_file;

if (file) {
    struct inode *inode = file->f_dentry->d_inode;
    if (vma->vm_flags & VM_DENYWRITE)
        atomic_inc(&inode->i_writecount);
    if(vma->vm_next_share)
        vma->vm_next_share->vm_pprev_share = vma->vm_pprev_share;
    *vma->vm_pprev_share = vma->vm_next_share;
}
```

5.7.2 Function remove_shared_vm_struct()

File: mm/mmap.c
Prototype:

```
void remove_shared_vm_struct(struct vm_area_struct *vma)
```

```
lock_vma_mappings(vma);
__remove_shared_vm_struct(vma);
unlock_vma_mappings(vma);
```

5.7.3 Function lock_vma_mappings()

File: mm/mmap.c
Prototype:

```
void lock_vma_mappings(struct vm_area_struct *vma)
```

```
struct address_space *mapping;

mapping = NULL;
if (vma->vm_file)
    mapping = vma->vm_file->f_dentry->d_inode->i_mapping;
if (mapping)
    spin_lock(&mapping->i_shared_lock);
```

5.7.4 Function unlock_vma_mappings()

File: mm/mmap.c
Prototype:

```
void unlock_vma_mappings(struct vm_area_struct *vma)
```

```
struct address_space *mapping;

mapping = NULL;
if (vma->vm_file)
    mapping = vma->vm_file->f_dentry->d_inode->i_mapping;
```

```
if (mapping)
    spin_unlock(&mapping->i_shared_lock);
```

5.7.5 Function calc_vm_flags()

File: mm/mmap.c
Prototype:

```
unsigned long calc_vm_flags(unsigned long prot,
                            unsigned long flags)


#define _trans(x,bit1,bit2) \
((bit1==bit2)?(x&bit1):(x&bit1)?bit2:0)

unsigned long prot_bits, flag_bits;
prot_bits =
    _trans(prot, PROT_READ, VM_READ) |
    _trans(prot, PROT_WRITE, VM_WRITE) |
    _trans(prot, PROT_EXEC, VM_EXEC);
flag_bits =
    _trans(flags, MAP_GROWSDOWN, VM_GROWSDOWN) |
    _trans(flags, MAP_DENYWRITE, VM_DENYWRITE) |
    _trans(flags, MAP_EXECUTABLE, VM_EXECUTABLE);
return prot_bits | flag_bits;
#undef _trans
```

5.7.6 Function __vma_link_list()

File: mm/mmap.c
Prototype:

```
void __vma_link_list(struct mm_struct * mm,
                     struct vm_area_struct * vma,
                     struct vm_area_struct * prev,
                             rb_node_t * rb_parent)


if (prev) {
    vma->vm_next = prev->vm_next;
```

```
    prev->vm_next = vma;
} else {
    mm->mmap = vma;
    if (rb_parent)
        vma->vm_next = rb_entry(rb_parent,
                  struct vm_area_struct, vm_rb);
    else
        vma->vm_next = NULL;
}
```

5.7.7 Function __vma_link_rb()

File: mm/mmap.c
Prototype:

```
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);
```

5.7.8 Function __vma_link_file()

File: mm/mmap.c
Prototype:

```
void __vma_link_file(struct vm_area_struct * vma)


struct file * file;

file = vma->vm_file;
if (file) {
    struct inode * inode = file->f_dentry->d_inode;
    struct address_space *mapping = inode->i_mapping;
    struct vm_area_struct **head;
```

```
    if (vma->vm_flags & VM_DENYWRITE)
        atomic_dec(&inode->i_writecount);

        head = &mapping->i_mmap;
        if (vma->vm_flags & VM_SHARED)
            head = &mapping->i_mmap_shared;

        /* insert vma into inode's share list */
        if((vma->vm_next_share = *head) != NULL)
(*head)->vm_pprev_share = &vma->vm_next_share;
        *head = vma;
        vma->vm_pprev_share = head;
}
```

5.7.9 Function __vma_link()

File: mm/mmap.c
Prototype:

```
void __vma_link(struct mm_struct * mm,
                struct vm_area_struct * vma,
                struct vm_area_struct * prev,
                rb_node_t ** rb_link,
                rb_node_t * rb_parent)


__vma_link_list(mm, vma, prev, rb_parent);
__vma_link_rb(mm, vma, rb_link, rb_parent);
__vma_link_file(vma);
```

5.7.10 Function vma_link()

File: mm/mmap.c
Prototype:

```
void vma_link(struct mm_struct * mm,
              struct vm_area_struct * vma,
              struct vm_area_struct * prev,
              rb_node_t ** rb_link,
              rb_node_t * rb_parent)
```

```
lock_vma_mappings(vma);
spin_lock(&mm->page_table_lock);
__vma_link(mm, vma, prev, rb_link, rb_parent);
spin_unlock(&mm->page_table_lock);
unlock_vma_mappings(vma);

mm->map_count++;
validate_mm(mm);
```

5.7.11 Function vma_merge()

File: mm/mmap.c
Prototype:

```
int vma_merge(struct mm_struct * mm,
              struct vm_area_struct * prev,
              rb_node_t * rb_parent,
              unsigned long addr,
              unsigned long end,
              unsigned long vm_flags)


spinlock_t * lock = &mm->page_table_lock;
if (!prev) {
    prev = rb_entry(rb_parent, struct vm_area_struct, vm_rb);
    goto merge_next;
}
if (prev->vm_end == addr && can_vma_merge(prev, vm_flags)) {
    struct vm_area_struct * next;

    spin_lock(lock);
    prev->vm_end = end;
    next = prev->vm_next;
    if (next && prev->vm_end == next->vm_start
             && can_vma_merge(next, vm_flags)) {
        prev->vm_end = next->vm_end;
        __vma_unlink(mm, next, prev);
        spin_unlock(lock);

        mm->map_count--;
```

```
            kmem_cache_free(vm_area_cachep, next);
            return 1;
        }
        spin_unlock(lock);
        return 1;
    }

    prev = prev->vm_next;
    if (prev) {
     merge_next:
        if (!can_vma_merge(prev, vm_flags))
            return 0;
        if (end == prev->vm_start) {
            spin_lock(lock);
            prev->vm_start = addr;
            spin_unlock(lock);
            return 1;
        }
    }

    return 0;
```

5.7.12 Function find_vma()

File: mm/mmap.c
Prototype:

```
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;
```

```
        rb_node = mm->mm_rb.rb_node;
        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;
                if (vma_tmp->vm_start <= addr)
                     break;
                rb_node = rb_node->rb_left;
            } else
                rb_node = rb_node->rb_right;
        }
        if (vma)
            mm->mmap_cache = vma;
    }
}
return vma;
```

5.7.13 Function find_vma_prev()

File: mm/mmap.c
Prototype:

```
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);
        if ((rb_prev ? (*pprev)->vm_next : mm->mmap) != vma)
            BUG();
        return vma;
    }
}
*pprev = NULL;
return NULL;
```

5.7.14 Function find_extend_vma()

File: mm/mmap.c
Prototype:

```
struct vm_area_struct *
find_extend_vma(struct mm_struct * mm,
                unsigned long addr)


struct vm_area_struct * vma;
unsigned long start;

addr &= PAGE_MASK;
vma = find_vma(mm,addr);
if (!vma)
    return NULL;
if (vma->vm_start <= addr)
    return vma;
if (!(vma->vm_flags & VM_GROWSDOWN))
    return NULL;
start = vma->vm_start;
if (expand_stack(vma, addr))
    return NULL;
if (vma->vm_flags & VM_LOCKED) {
    make_pages_present(addr, start);
}
return vma;
```

5.7.15 Function unmap_fixup()

File: mm/mmap.c
Prototype:

```
struct vm_area_struct *
unmap_fixup(struct mm_struct *mm,
            struct vm_area_struct *area,
            unsigned long addr,
            size_t len,
            struct vm_area_struct *extra)


struct vm_area_struct *mpnt;
unsigned long end = addr + len;

area->vm_mm->total_vm -= len >> PAGE_SHIFT;
```

```
if (area->vm_flags & VM_LOCKED)
    area->vm_mm->locked_vm -= len >> PAGE_SHIFT;

/* Unmapping the whole area. */
if (addr == area->vm_start && end == area->vm_end) {
    if (area->vm_ops && area->vm_ops->close)
        area->vm_ops->close(area);
    if (area->vm_file)
        fput(area->vm_file);
    kmem_cache_free(vm_area_cachep, area);
    return extra;
}

/* Work out to one of the ends. */
if (end == area->vm_end) {
/*
 * here area isn't visible to the semaphore-less readers
 * so we don't need to update it under the spinlock.
 */
    area->vm_end = addr;
    lock_vma_mappings(area);
    spin_lock(&mm->page_table_lock);
} else if (addr == area->vm_start) {
    area->vm_pgoff += (end - area->vm_start) >> PAGE_SHIFT;
    /* same locking considerations of the above case */
    area->vm_start = end;
    lock_vma_mappings(area);
    spin_lock(&mm->page_table_lock);
} else {
/* Unmapping a hole: area->vm_start < addr <= end < area->vm_end */
/* Add end mapping -- leave beginning for below */
    mpnt = extra;
    extra = NULL;

    mpnt->vm_mm = area->vm_mm;
    mpnt->vm_start = end;
    mpnt->vm_end = area->vm_end;
    mpnt->vm_page_prot = area->vm_page_prot;
    mpnt->vm_flags = area->vm_flags;
    mpnt->vm_raend = 0;
    mpnt->vm_ops = area->vm_ops;
```

```
    mpnt->vm_pgoff = area->vm_pgoff +
                     ((end - area->vm_start) >> PAGE_SHIFT);
    mpnt->vm_file = area->vm_file;
    mpnt->vm_private_data = area->vm_private_data;
    if (mpnt->vm_file)
        get_file(mpnt->vm_file);
    if (mpnt->vm_ops && mpnt->vm_ops->open)
        mpnt->vm_ops->open(mpnt);
    area->vm_end = addr; /* Truncate area */

/* Because mpnt->vm_file == area->vm_file this locks
 * things correctly.
 */
    lock_vma_mappings(area);
    spin_lock(&mm->page_table_lock);
    __insert_vm_struct(mm, mpnt);
}

__insert_vm_struct(mm, area);
spin_unlock(&mm->page_table_lock);
unlock_vma_mappings(area);
return extra;
```

5.7.16 Function free_pgtables()

File: mm/mmap.c
Prototype:

```
void free_pgtables(struct mm_struct * mm,
                   struct vm_area_struct *prev,
                   unsigned long start,
                   unsigned long end)


unsigned long first = start & PGDIR_MASK;
unsigned long last = end + PGDIR_SIZE - 1;
unsigned long start_index, end_index;

if (!prev) {
    prev = mm->mmap;
```

```
    if (!prev)
        goto no_mmaps;
    if (prev->vm_end > start) {
        if (last > prev->vm_start)
            last = prev->vm_start;
        goto no_mmaps;
    }
}
for (;;) {
    struct vm_area_struct *next = prev->vm_next;

    if (next) {
        if (next->vm_start < start) {
            prev = next;
            continue;
        }
        if (last > next->vm_start)
            last = next->vm_start;
}
if (prev->vm_end > first)
            first = prev->vm_end + PGDIR_SIZE - 1;
    break;
}
no_mmaps:
/*
 * If the PGD bits are not consecutive in the virtual address, the
 * old method of shifting the VA >> by PGDIR_SHIFT doesn't work.
 */
start_index = pgd_index(first);
end_index = pgd_index(last);
if (end_index > start_index) {
    clear_page_tables(mm, start_index, end_index - start_index);
    flush_tlb_pgtables(mm, first & PGDIR_MASK, last & PGDIR_MASK);
}
```

5.7.17 Function build_mmap_rb()

File: mm/mmap.c
Prototype:

```
void build_mmap_rb(struct mm_struct * mm)


struct vm_area_struct * vma;
rb_node_t ** rb_link, * rb_parent;

mm->mm_rb = RB_ROOT;
rb_link = &mm->mm_rb.rb_node;
rb_parent = NULL;
for (vma = mm->mmap; vma; vma = vma->vm_next) {
    __vma_link_rb(mm, vma, rb_link, rb_parent);
    rb_parent = &vma->vm_rb;
    rb_link = &rb_parent->rb_right;
}
```

5.7.18 Function __insert_vm_struct()

File: mm/mmap.c
Prototype:

```
void __insert_vm_struct(struct mm_struct * mm,
                        struct vm_area_struct * vma)


struct vm_area_struct * __vma, * prev;
rb_node_t ** rb_link, * rb_parent;

__vma = find_vma_prepare(mm, vma->vm_start, &prev, &rb_link, &rb_parent);
if (__vma && __vma->vm_start < vma->vm_end)
    BUG();
__vma_link(mm, vma, prev, rb_link, rb_parent);
mm->map_count++;
validate_mm(mm);
```

5.7.19 Function insert_vm_struct()

File: mm/mmap.c
Prototype:

```
void insert_vm_struct(struct mm_struct * mm,
                      struct vm_area_struct * vma)


struct vm_area_struct * __vma, * prev;
rb_node_t ** rb_link, * rb_parent;

__vma = find_vma_prepare(mm, vma->vm_start, &prev, &rb_link, &rb_parent);
if (__vma && __vma->vm_start < vma->vm_end)
    BUG();
vma_link(mm, vma, prev, rb_link, rb_parent);
validate_mm(mm);
```

Chapter 6

Demand Paging

6.0.1 Function copy_cow_page()

Prototype:

```
void copy_cow_page(struct page * from,
                   struct page * to,
                   unsigned long address)


if (from == ZERO_PAGE(address)) {
clear_user_highpage(to, address);
return;
}
copy_user_highpage(to, from, address);
```

6.0.2 Function __free_pte()

Prototype:

```
void __free_pte(pte_t pte)


struct page *page = pte_page(pte);
if ((!VALID_PAGE(page)) || PageReserved(page))
return;
if (pte_dirty(pte))
set_page_dirty(page);
free_page_and_swap_cache(page);
```

6.0.3 Function free_one_pmd()

Prototype:

```
void free_one_pmd(pmd_t * dir)


pte_t * pte;

if (pmd_none(*dir))
return;
if (pmd_bad(*dir)) {
pmd_ERROR(*dir);
pmd_clear(dir);
return;
}
pte = pte_offset(dir, 0);
pmd_clear(dir);
pte_free(pte);
```

6.0.4 Function free_one_pgd()

Prototype:

```
void free_one_pgd(pgd_t * dir)


int j;
pmd_t * pmd;

if (pgd_none(*dir))
return;
if (pgd_bad(*dir)) {
pgd_ERROR(*dir);
pgd_clear(dir);
return;
}
pmd = pmd_offset(dir, 0);
pgd_clear(dir);
for (j = 0; j < PTRS_PER_PMD ; j++) {
prefetchw(pmd+j+(PREFETCH_STRIDE/16));
free_one_pmd(pmd+j);
}
pmd_free(pmd);
```

6.0.5 Function check_pgt_cache()

Prototype:

```
int check_pgt_cache(void)
```

Returns the number of pages freed.

```
return do_check_pgt_cache(pgt_cache_water[0], pgt_cache_water[1]);
```

6.0.6 Function clear_page_tables()

Prototype:

```
void clear_page_tables(struct mm_struct *mm,
                       unsigned long first,
                       int nr)
```

This function clears all user-level page tables of a process - this is needed by execve(), so that old pages aren't in the way.

```
pgd_t * page_dir = mm->pgd;

spin_lock(&mm->page_table_lock);
page_dir += first;
do {
free_one_pgd(page_dir);
page_dir++;
} while (--nr);
spin_unlock(&mm->page_table_lock);

/* keep the page table cache within bounds */
check_pgt_cache();
```

6.0.7 Function copy_page_range()

Prototype:

```
int copy_page_range(struct mm_struct *dst,
                    struct mm_struct *src,
                    struct vm_area_struct *vma)
```

```
pgd_t * src_pgd, * dst_pgd;
unsigned long address = vma->vm_start;
unsigned long end = vma->vm_end;
unsigned long cow = (vma->vm_flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;

src_pgd = pgd_offset(src, address)-1;
dst_pgd = pgd_offset(dst, address)-1;

for (;;) {
pmd_t * src_pmd, * dst_pmd;

src_pgd++; dst_pgd++;

/* copy_pmd_range */

if (pgd_none(*src_pgd))
goto skip_copy_pmd_range;
if (pgd_bad(*src_pgd)) {
pgd_ERROR(*src_pgd);
pgd_clear(src_pgd);
skip_copy_pmd_range: address = (address + PGDIR_SIZE) & PGDIR_MASK;
if (!address || (address >= end))
goto out;
continue;
}

src_pmd = pmd_offset(src_pgd, address);
dst_pmd = pmd_alloc(dst, dst_pgd, address);
if (!dst_pmd)
goto nomem;

do {
pte_t * src_pte, * dst_pte;

/* copy_pte_range */

if (pmd_none(*src_pmd))
goto skip_copy_pte_range;
if (pmd_bad(*src_pmd)) {
pmd_ERROR(*src_pmd);
pmd_clear(src_pmd);
```

```
skip_copy_pte_range: address = (address + PMD_SIZE) & PMD_MASK;
if (address >= end)
goto out;
goto cont_copy_pmd_range;
}

src_pte = pte_offset(src_pmd, address);
dst_pte = pte_alloc(dst, dst_pmd, address);
if (!dst_pte)
goto nomem;

spin_lock(&src->page_table_lock);
do {
pte_t pte = *src_pte;
struct page *ptepage;

/* copy_one_pte */

if (pte_none(pte))
goto cont_copy_pte_range_noset;
if (!pte_present(pte)) {
swap_duplicate(pte_to_swp_entry(pte));
goto cont_copy_pte_range;
}
ptepage = pte_page(pte);
if ((!VALID_PAGE(ptepage)) ||
    PageReserved(ptepage))
goto cont_copy_pte_range;

/* If it's a COW mapping, write protect it both in the parent and the child */
if (cow && pte_write(pte)) {
ptep_set_wrprotect(src_pte);
pte = *src_pte;
}

/* If it's a shared mapping, mark it clean in the child */
if (vma->vm_flags & VM_SHARED)
pte = pte_mkclean(pte);
pte = pte_mkold(pte);
get_page(ptepage);
dst->rss++;
```

```
cont_copy_pte_range: set_pte(dst_pte, pte);
cont_copy_pte_range_noset: address += PAGE_SIZE;
if (address >= end)
goto out_unlock;
src_pte++;
dst_pte++;
} while ((unsigned long)src_pte & PTE_TABLE_MASK);
spin_unlock(&src->page_table_lock);

cont_copy_pmd_range: src_pmd++;
dst_pmd++;
} while ((unsigned long)src_pmd & PMD_TABLE_MASK);
}
out_unlock:
spin_unlock(&src->page_table_lock);
out:
return 0;
nomem:
return -ENOMEM;
```

6.0.8 Function forget_pte()

Prototype:

```
void forget_pte(pte_t page)


if (!pte_none(page)) {
printk("forget_pte: old mapping existed!\n");
BUG();
}
```

6.0.9 Function zap_pte_range()

Prototype:

```
int zap_pte_range(mmu_gather_t *tlb,
                  pmd_t * pmd,
                  unsigned long address,
                  unsigned long size)
```

```
unsigned long offset;
pte_t * ptep;
int freed = 0;

if (pmd_none(*pmd))
return 0;
if (pmd_bad(*pmd)) {
pmd_ERROR(*pmd);
pmd_clear(pmd);
return 0;
}
ptep = pte_offset(pmd, address);
offset = address & ~PMD_MASK;
if (offset + size > PMD_SIZE)
size = PMD_SIZE - offset;
size &= PAGE_MASK;
for (offset=0; offset < size; ptep++, offset += PAGE_SIZE) {
pte_t pte = *ptep;
if (pte_none(pte))
continue;
if (pte_present(pte)) {
struct page *page = pte_page(pte);
if (VALID_PAGE(page) && !PageReserved(page))
freed ++;
/* This will eventually call __free_pte on the pte. */
tlb_remove_page(tlb, ptep, address + offset);
} else {
free_swap_and_cache(pte_to_swp_entry(pte));
pte_clear(ptep);
}
}

return freed;
```

6.0.10 Function zap_pmd_range()

Prototype:

```
int zap_pmd_range(mmu_gather_t *tlb,
                  pgd_t * dir,
                  unsigned long address,
```

```
                unsigned long size)


pmd_t * pmd;
unsigned long end;
int freed;

if (pgd_none(*dir))
return 0;
if (pgd_bad(*dir)) {
pgd_ERROR(*dir);
pgd_clear(dir);
return 0;
}
pmd = pmd_offset(dir, address);
end = address + size;
if (end > ((address + PGDIR_SIZE) & PGDIR_MASK))
end = ((address + PGDIR_SIZE) & PGDIR_MASK);
freed = 0;
do {
freed += zap_pte_range(tlb, pmd, address, end - address);
address = (address + PMD_SIZE) & PMD_MASK;
pmd++;
} while (address < end);
return freed;
```

6.0.11 Function zap_page_range()

Prototype:

```
void zap_page_range(struct mm_struct *mm,
                    unsigned long address,
                    unsigned long size)


mmu_gather_t *tlb;
pgd_t * dir;
unsigned long start = address, end = address + size;
int freed = 0;

dir = pgd_offset(mm, address);
```

```
/*
 * This is a long-lived spinlock. That's fine.
 * There's no contention, because the page table
 * lock only protects against kswapd anyway, and
 * even if kswapd happened to be looking at this
 * process we _want_ it to get stuck.
 */
if (address >= end)
BUG();
spin_lock(&mm->page_table_lock);
flush_cache_range(mm, address, end);
tlb = tlb_gather_mmu(mm);

do {
freed += zap_pmd_range(tlb, dir, address, end - address);
address = (address + PGDIR_SIZE) & PGDIR_MASK;
dir++;
} while (address && (address < end));

/* this will flush any remaining tlb entries */
tlb_finish_mmu(tlb, start, end);

/*
 * Update rss for the mm_struct (not necessarily current->mm)
 * Notice that rss is an unsigned long.
 */
if (mm->rss > freed)
mm->rss -= freed;
else
mm->rss = 0;
spin_unlock(&mm->page_table_lock);
```

6.0.12 Function follow_page()

Prototype:

```
struct page * follow_page(struct mm_struct *mm,
                          unsigned long address,
                          int write)


pgd_t *pgd;
```

```
pmd_t *pmd;
pte_t *ptep, pte;

pgd = pgd_offset(mm, address);
if (pgd_none(*pgd) || pgd_bad(*pgd))
goto out;

pmd = pmd_offset(pgd, address);
if (pmd_none(*pmd) || pmd_bad(*pmd))
goto out;

ptep = pte_offset(pmd, address);
if (!ptep)
goto out;

pte = *ptep;
if (pte_present(pte)) {
if (!write ||
    (pte_write(pte) && pte_dirty(pte)))
return pte_page(pte);
}

out:
return 0;
```

6.0.13 Function get_page_map()

Prototype:

```
struct page * get_page_map(struct page *page)


if (!VALID_PAGE(page))
return 0;
return page;
```

6.0.14 Function get_user_pages()

Prototype:

```
int get_user_pages(struct task_struct *tsk,
                   struct mm_struct *mm,
```

```
                    unsigned long start,
                    int len, int write,
                    int force, struct page **pages,
                    struct vm_area_struct **vmas)


int i;
unsigned int flags;

/*
 * Require read or write permissions.
 * If 'force' is set, we only require the "MAY" flags.
 */
flags = write ? (VM_WRITE | VM_MAYWRITE) : (VM_READ | VM_MAYREAD);
flags &= force ? (VM_MAYREAD | VM_MAYWRITE) : (VM_READ | VM_WRITE);
i = 0;

do {
struct vm_area_struct * vma;

vma = find_extend_vma(mm, start);

if ( !vma || (pages && vma->vm_flags & VM_IO) || !(flags & vma->vm_flags) )
return i ? : -EFAULT;

spin_lock(&mm->page_table_lock);
do {
struct page *map;
while (!(map = follow_page(mm, start, write))) {
spin_unlock(&mm->page_table_lock);
switch (handle_mm_fault(mm, vma, start, write)) {
case 1:
tsk->min_flt++;
break;
case 2:
tsk->maj_flt++;
break;
case 0:
if (i) return i;
return -EFAULT;
default:
```

```
if (i) return i;
return -ENOMEM;
}
spin_lock(&mm->page_table_lock);
}
if (pages) {
pages[i] = get_page_map(map);
/* FIXME: call the correct function,
 * depending on the type of the found page
 */
if (!pages[i])
goto bad_page;
page_cache_get(pages[i]);
}
if (vmas)
vmas[i] = vma;
i++;
start += PAGE_SIZE;
len--;
} while(len && start < vma->vm_end);
spin_unlock(&mm->page_table_lock);
} while(len);
out:
return i;

/*
 * We found an invalid page in the VMA.  Release all we have
 * so far and fail.
 */
bad_page:
spin_unlock(&mm->page_table_lock);
while (i--)
page_cache_release(pages[i]);
i = -EFAULT;
goto out;
```

6.0.15 Function map_user_kiobuf()

Prototype:

```
int map_user_kiobuf(int rw,
```

```
                    struct kiobuf *iobuf,
                    unsigned long va,
                    size_t len)


int pgcount, err;
struct mm_struct * mm;

/* Make sure the iobuf is not already mapped somewhere. */
if (iobuf->nr_pages)
return -EINVAL;

mm = current->mm;
dprintk ("map_user_kiobuf: begin\n");

pgcount = (va + len + PAGE_SIZE - 1)/PAGE_SIZE - va/PAGE_SIZE;
/* mapping 0 bytes is not permitted */
if (!pgcount) BUG();
err = expand_kiobuf(iobuf, pgcount);
if (err)
return err;

iobuf->locked = 0;
iobuf->offset = va & (PAGE_SIZE-1);
iobuf->length = len;

/* Try to fault in all of the necessary pages */
down_read(&mm->mmap_sem);
/* rw==READ means read from disk, write into memory area */
err = get_user_pages(current, mm, va, pgcount,
(rw==READ), 0, iobuf->maplist, NULL);
up_read(&mm->mmap_sem);
if (err < 0) {
unmap_kiobuf(iobuf);
dprintk ("map_user_kiobuf: end %d\n", err);
return err;
}
iobuf->nr_pages = err;
while (pgcount--) {
/* FIXME: flush superflous for rw==READ,
 * probably wrong function for rw==WRITE
```

```
 */
flush_dcache_page(iobuf->maplist[pgcount]);
}
dprintk ("map_user_kiobuf: end OK\n");
return 0;
```

6.0.16 Function mark_dirty_kiobuf()

Prototype:

```
void mark_dirty_kiobuf(struct kiobuf *iobuf,
                       int bytes)


int index, offset, remaining;
struct page *page;

index = iobuf->offset >> PAGE_SHIFT;
offset = iobuf->offset & ~PAGE_MASK;
remaining = bytes;
if (remaining > iobuf->length)
remaining = iobuf->length;

while (remaining > 0 && index < iobuf->nr_pages) {
page = iobuf->maplist[index];

if (!PageReserved(page))
SetPageDirty(page);

remaining -= (PAGE_SIZE - offset);
offset = 0;
index++;
}
```

6.0.17 Function unmap_kiobuf()

Prototype:

```
void unmap_kiobuf (struct kiobuf *iobuf)
```

```
int i;
struct page *map;

for (i = 0; i < iobuf->nr_pages; i++) {
map = iobuf->maplist[i];
if (map) {
if (iobuf->locked)
UnlockPage(map);
/* FIXME: cache flush missing for rw==READ
 * FIXME: call the correct reference counting function
 */
page_cache_release(map);
}
}

iobuf->nr_pages = 0;
iobuf->locked = 0;
```

6.0.18 Function lock_kiovec()

Prototype:

```
int lock_kiovec(int nr,
                struct kiobuf *iovec[],
                int wait)


struct kiobuf *iobuf;
int i, j;
struct page *page, **ppage;
int doublepage = 0;
int repeat = 0;

 repeat:

for (i = 0; i < nr; i++) {
iobuf = iovec[i];

if (iobuf->locked)
continue;
```

```
ppage = iobuf->maplist;
for (j = 0; j < iobuf->nr_pages; ppage++, j++) {
page = *ppage;
if (!page)
continue;

if (TryLockPage(page)) {
while (j--) {
struct page *tmp = *--ppage;
if (tmp)
UnlockPage(tmp);
}
goto retry;
}
}
iobuf->locked = 1;
}

return 0;

 retry:

/*
 * We couldn't lock one of the pages.  Undo the locking so far,
 * wait on the page we got to, and try again.
 */

unlock_kiovec(nr, iovec);
if (!wait)
return -EAGAIN;

/*
 * Did the release also unlock the page we got stuck on?
 */
if (!PageLocked(page)) {
/*
 * If so, we may well have the page mapped twice
 * in the IO address range.  Bad news.  Of
 * course, it _might_ just be a coincidence,
 * but if it happens more than once, chances
 * are we have a double-mapped page.
```

```
 */
if (++doublepage >= 3)
return -EINVAL;

/* Try again...  */
wait_on_page(page);
}

if (++repeat < 16)
goto repeat;
return -EAGAIN;
```

6.0.19 Function unlock_kiovec()

Prototype:

```
int unlock_kiovec(int nr,
                  struct kiobuf *iovec[])



struct kiobuf *iobuf;
int i, j;
struct page *page, **ppage;

for (i = 0; i < nr; i++) {
iobuf = iovec[i];

if (!iobuf->locked)
continue;
iobuf->locked = 0;

ppage = iobuf->maplist;
for (j = 0; j < iobuf->nr_pages; ppage++, j++) {
page = *ppage;
if (!page)
continue;
UnlockPage(page);
}
}
return 0;
```

6.0.20 Function zeromap_pte_range()

Prototype:

```
void zeromap_pte_range(pte_t * pte,
                       unsigned long address,
                       unsigned long size,
                       pgprot_t prot)


unsigned long end;

address &= ~PMD_MASK;
end = address + size;
if (end > PMD_SIZE)
end = PMD_SIZE;
do {
pte_t zero_pte = pte_wrprotect(mk_pte(ZERO_PAGE(address), prot));
pte_t oldpage = ptep_get_and_clear(pte);
set_pte(pte, zero_pte);
forget_pte(oldpage);
address += PAGE_SIZE;
pte++;
} while (address && (address < end));
```

6.0.21 Function zeromap_pmd_range()

Prototype:

```
int zeromap_pmd_range(struct mm_struct *mm,
                      pmd_t * pmd,
                      unsigned long address,
                      unsigned long size,
                      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(mm, pmd, address);
if (!pte)
return -ENOMEM;
zeromap_pte_range(pte, address, end - address, prot);
address = (address + PMD_SIZE) & PMD_MASK;
pmd++;
} while (address && (address < end));
return 0;
```

6.0.22 Function zeromap_page_range()

Prototype:

```
int zeromap_page_range(unsigned long address,
                       unsigned long size,
                       pgprot_t prot)


int error = 0;
pgd_t * dir;
unsigned long beg = address;
unsigned long end = address + size;
struct mm_struct *mm = current->mm;

dir = pgd_offset(mm, address);
flush_cache_range(mm, beg, end);
if (address >= end)
BUG();

spin_lock(&mm->page_table_lock);
do {
pmd_t *pmd = pmd_alloc(mm, dir, address);
error = -ENOMEM;
if (!pmd)
break;
error = zeromap_pmd_range(mm, pmd, address, end - address, prot);
if (error)
break;
address = (address + PGDIR_SIZE) & PGDIR_MASK;
dir++;
```

```
} while (address && (address < end));
spin_unlock(&mm->page_table_lock);
flush_tlb_range(mm, beg, end);
return error;
```

6.0.23 Function remap_pte_range()

Prototype:

```
void remap_pte_range(pte_t * pte,
                     unsigned long address,
                     unsigned long size,
                     unsigned long phys_addr,
                     pgprot_t prot)


unsigned long end;

address &= ~PMD_MASK;
end = address + size;
if (end > PMD_SIZE)
end = PMD_SIZE;
do {
struct page *page;
pte_t oldpage;
oldpage = ptep_get_and_clear(pte);

page = virt_to_page(__va(phys_addr));
if ((!VALID_PAGE(page)) || PageReserved(page))
  set_pte(pte, mk_pte_phys(phys_addr, prot));
forget_pte(oldpage);
address += PAGE_SIZE;
phys_addr += PAGE_SIZE;
pte++;
} while (address && (address < end));
```

6.0.24 Function remap_pmd_range()

Prototype:

```
int remap_pmd_range(struct mm_struct *mm,
                    pmd_t * pmd,
```

```
                    unsigned long address,
                    unsigned long size,
                    unsigned long phys_addr,
                    pgprot_t prot)


unsigned long end;

address &= ~PGDIR_MASK;
end = address + size;
if (end > PGDIR_SIZE)
end = PGDIR_SIZE;
phys_addr -= address;
do {
pte_t * pte = pte_alloc(mm, pmd, address);
if (!pte)
return -ENOMEM;
remap_pte_range(pte, address, end - address, address + phys_addr, prot);
address = (address + PMD_SIZE) & PMD_MASK;
pmd++;
} while (address && (address < end));
return 0;
```

6.0.25 Function remap_page_range()

Prototype:

```
int remap_page_range(unsigned long from,
                     unsigned long phys_addr,
                     unsigned long size,
                     pgprot_t prot)


int error = 0;
pgd_t * dir;
unsigned long beg = from;
unsigned long end = from + size;
struct mm_struct *mm = current->mm;

phys_addr -= from;
dir = pgd_offset(mm, from);
flush_cache_range(mm, beg, end);
```

```
if (from >= end)
BUG();

spin_lock(&mm->page_table_lock);
do {
pmd_t *pmd = pmd_alloc(mm, dir, from);
error = -ENOMEM;
if (!pmd)
break;
error = remap_pmd_range(mm, pmd, from, end - from, phys_addr + from, prot);
if (error)
break;
from = (from + PGDIR_SIZE) & PGDIR_MASK;
dir++;
} while (from && (from < end));
spin_unlock(&mm->page_table_lock);
flush_tlb_range(mm, beg, end);
return error;
```

6.0.26 Function establish_pte()

Prototype:

```
void establish_pte(struct vm_area_struct * vma,
                   unsigned long address,
                   pte_t *page_table,
                   pte_t entry)


set_pte(page_table, entry);
flush_tlb_page(vma, address);
update_mmu_cache(vma, address, entry);
```

6.0.27 Function break_cow()

Prototype:

```
void break_cow(struct vm_area_struct * vma,
               struct page * new_page,
               unsigned long address,
               pte_t *page_table)
```

```
flush_page_to_ram(new_page);
flush_cache_page(vma, address);
establish_pte(vma, address, page_table, pte_mkwrite(pte_mkdirty(mk_pte(new_page, vma->
```

6.0.28 Function do_wp_page()

Prototype:

```
int do_wp_page(struct mm_struct *mm,
               struct vm_area_struct * vma,
               unsigned long address,
               pte_t *page_table,
               pte_t pte)


struct page *old_page, *new_page;

old_page = pte_page(pte);
if (!VALID_PAGE(old_page))
goto bad_wp_page;

if (!TryLockPage(old_page)) {
int reuse = can_share_swap_page(old_page);
unlock_page(old_page);
if (reuse) {
flush_cache_page(vma, address);
establish_pte(vma, address, page_table, pte_mkyoung(pte_mkdirty(pte_mkwrite(pte))));
spin_unlock(&mm->page_table_lock);
return 1; /* Minor fault */
}
}

/*
 * Ok, we need to copy. Oh, well..
 */
page_cache_get(old_page);
spin_unlock(&mm->page_table_lock);

new_page = alloc_page(GFP_HIGHUSER);
if (!new_page)
goto no_mem;
```

```
copy_cow_page(old_page,new_page,address);

/*
 * Re-check the pte - we dropped the lock
 */
spin_lock(&mm->page_table_lock);
if (pte_same(*page_table, pte)) {
if (PageReserved(old_page))
++mm->rss;
break_cow(vma, new_page, address, page_table);
lru_cache_add(new_page);

/* Free the old page.. */
new_page = old_page;
}
spin_unlock(&mm->page_table_lock);
page_cache_release(new_page);
page_cache_release(old_page);
return 1; /* Minor fault */

bad_wp_page:
spin_unlock(&mm->page_table_lock);
printk("do_wp_page: bogus page at address %08lx (page 0x%lx)\n",address,(unsigne
return -1;
no_mem:
page_cache_release(old_page);
return -1;
```

6.0.29 Function vmtruncate_list()

Prototype:

```
void vmtruncate_list(struct vm_area_struct *mpnt,
                     unsigned long pgoff)


do {
struct mm_struct *mm = mpnt->vm_mm;
unsigned long start = mpnt->vm_start;
unsigned long end = mpnt->vm_end;
unsigned long len = end - start;
```

```
unsigned long diff;

/* mapping wholly truncated? */
if (mpnt->vm_pgoff >= pgoff) {
zap_page_range(mm, start, len);
continue;
}

/* mapping wholly unaffected? */
len = len >> PAGE_SHIFT;
diff = pgoff - mpnt->vm_pgoff;
if (diff >= len)
continue;

/* Ok, partially affected.. */
start += diff << PAGE_SHIFT;
len = (len - diff) << PAGE_SHIFT;
zap_page_range(mm, start, len);
} while ((mpnt = mpnt->vm_next_share) != NULL);
```

6.0.30 Function vmtruncate()

Prototype:

```
int vmtruncate(struct inode * inode,
               loff_t offset)


unsigned long pgoff;
struct address_space *mapping = inode->i_mapping;
unsigned long limit;

if (inode->i_size < offset)
goto do_expand;
inode->i_size = offset;
spin_lock(&mapping->i_shared_lock);
if (!mapping->i_mmap && !mapping->i_mmap_shared)
goto out_unlock;

pgoff = (offset + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
if (mapping->i_mmap != NULL)
```

```
vmtruncate_list(mapping->i_mmap, pgoff);
if (mapping->i_mmap_shared != NULL)
vmtruncate_list(mapping->i_mmap_shared, pgoff);

out_unlock:
spin_unlock(&mapping->i_shared_lock);
truncate_inode_pages(mapping, offset);
goto out_truncate;

do_expand:
limit = current->rlim[RLIMIT_FSIZE].rlim_cur;
if (limit != RLIM_INFINITY && offset > limit)
goto out_sig;
if (offset > inode->i_sb->s_maxbytes)
goto out;
inode->i_size = offset;

out_truncate:
if (inode->i_op && inode->i_op->truncate) {
lock_kernel();
inode->i_op->truncate(inode);
unlock_kernel();
}
return 0;
out_sig:
send_sig(SIGXFSZ, current, 0);
out:
return -EFBIG;
```

6.0.31 Function swapin_readahead()

Prototype:

```
void swapin_readahead(swp_entry_t entry)


int i, num;
struct page *new_page;
unsigned long offset;

/*
```

```
 * Get the number of handles we should do readahead io to.
 */
num = valid_swaphandles(entry, &offset);
for (i = 0; i < num; offset++, i++) {
/* Ok, do the async read-ahead now */
new_page = read_swap_cache_async(SWP_ENTRY(SWP_TYPE(entry), offset));
if (!new_page)
break;
page_cache_release(new_page);
}
return;
```

6.0.32 Function do_swap_page()

Prototype:

```
int do_swap_page(struct mm_struct * mm,
                 struct vm_area_struct * vma,
                 unsigned long address,
                 pte_t * page_table,
                 pte_t orig_pte,
                 int write_access)


struct page *page;
swp_entry_t entry = pte_to_swp_entry(orig_pte);
pte_t pte;
int ret = 1;

spin_unlock(&mm->page_table_lock);
page = lookup_swap_cache(entry);
if (!page) {
swapin_readahead(entry);
page = read_swap_cache_async(entry);
if (!page) {
/*
 * Back out if somebody else faulted in this pte while
 * we released the page table lock.
 */
int retval;
spin_lock(&mm->page_table_lock);
```

```
retval = pte_same(*page_table, orig_pte) ? -1 : 1;
spin_unlock(&mm->page_table_lock);
return retval;
}

/* Had to read the page from swap area: Major fault */
ret = 2;
}

mark_page_accessed(page);

lock_page(page);

/*
 * Back out if somebody else faulted in this pte while we
 * released the page table lock.
 */
spin_lock(&mm->page_table_lock);
if (!pte_same(*page_table, orig_pte)) {
spin_unlock(&mm->page_table_lock);
unlock_page(page);
page_cache_release(page);
return 1;
}

/* The page isn't present yet, go ahead with the fault. */

swap_free(entry);
if (vm_swap_full())
remove_exclusive_swap_page(page);

mm->rss++;
pte = mk_pte(page, vma->vm_page_prot);
if (write_access && can_share_swap_page(page))
pte = pte_mkdirty(pte_mkwrite(pte));
unlock_page(page);

flush_page_to_ram(page);
flush_icache_page(vma, page);
set_pte(page_table, pte);
```

```
/* No need to invalidate - it was non-present before */
update_mmu_cache(vma, address, pte);
spin_unlock(&mm->page_table_lock);
return ret;
```

6.0.33 Function do_anonymous_page()

Prototype:

```
int do_anonymous_page(struct mm_struct * mm,
                      struct vm_area_struct * vma,
                      pte_t *page_table,
                      int write_access,
                      unsigned long addr)


pte_t entry;

/* Read-only mapping of ZERO_PAGE. */
entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));

/* ..except if it's a write access */
if (write_access) {
struct page *page;

/* Allocate our own private page. */
spin_unlock(&mm->page_table_lock);

page = alloc_page(GFP_HIGHUSER);
if (!page)
goto no_mem;
clear_user_highpage(page, addr);

spin_lock(&mm->page_table_lock);
if (!pte_none(*page_table)) {
page_cache_release(page);
spin_unlock(&mm->page_table_lock);
return 1;
}
mm->rss++;
flush_page_to_ram(page);
```

```
entry = pte_mkwrite(pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
lru_cache_add(page);
mark_page_accessed(page);
}

set_pte(page_table, entry);

/* No need to invalidate - it was non-present before */
update_mmu_cache(vma, addr, entry);
spin_unlock(&mm->page_table_lock);
return 1; /* Minor fault */

no_mem:
return -1;
```

6.0.34 Function do_no_page()

Prototype:

```
int do_no_page(struct mm_struct * mm,
               struct vm_area_struct * vma,
               unsigned long address,
               int write_access,
               pte_t *page_table)


struct page * new_page;
pte_t entry;

if (!vma->vm_ops || !vma->vm_ops->nopage)
return do_anonymous_page(mm, vma, page_table, write_access, address);
spin_unlock(&mm->page_table_lock);

new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);

if (new_page == NULL) /* no page was available -- SIGBUS */
return 0;
if (new_page == NOPAGE_OOM)
return -1;

/*
```

```
 * Should we do an early C-O-W break?
 */
if (write_access && !(vma->vm_flags & VM_SHARED)) {
struct page * page = alloc_page(GFP_HIGHUSER);
if (!page) {
page_cache_release(new_page);
return -1;
}
copy_user_highpage(page, new_page, address);
page_cache_release(new_page);
lru_cache_add(page);
new_page = page;
}

spin_lock(&mm->page_table_lock);
/*
 * This silly early PAGE_DIRTY setting removes a race
 * due to the bad i386 page protection. But it's valid
 * for other architectures too.
 *
 * Note that if write_access is true, we either now have
 * an exclusive copy of the page, or this is a shared mapping,
 * so we can make it writable and dirty to avoid having to
 * handle that later.
 */
/* Only go through if we didn't race with anybody else... */
if (pte_none(*page_table)) {
++mm->rss;
flush_page_to_ram(new_page);
flush_icache_page(vma, new_page);
entry = mk_pte(new_page, vma->vm_page_prot);
if (write_access)
entry = pte_mkwrite(pte_mkdirty(entry));
set_pte(page_table, entry);
} else {
/* One of our sibling threads was faster, back out. */
page_cache_release(new_page);
spin_unlock(&mm->page_table_lock);
return 1;
}
```

```
/* no need to invalidate: a not-present page shouldn't be cached */
update_mmu_cache(vma, address, entry);
spin_unlock(&mm->page_table_lock);
return 2; /* Major fault */
```

6.0.35 Function handle_pte_fault()

Prototype:

```
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);
}

if (write_access) {
if (!pte_write(entry))
return do_wp_page(mm, vma, address, pte, entry);

entry = pte_mkdirty(entry);
}
entry = pte_mkyoung(entry);
establish_pte(vma, address, pte, entry);
spin_unlock(&mm->page_table_lock);
return 1;
```

6.0.36 Function handle_mm_fault()

Prototype:

```
int handle_mm_fault(struct mm_struct *mm,
                    struct vm_area_struct * vma,
                    unsigned long address,
                    int write_access)


pgd_t *pgd;
pmd_t *pmd;

current->state = TASK_RUNNING;
pgd = pgd_offset(mm, address);

/*
 * We need the page table lock to synchronize with kswapd
 * and the SMP-safe atomic PTE updates.
 */
spin_lock(&mm->page_table_lock);
pmd = pmd_alloc(mm, pgd, address);

if (pmd) {
pte_t * pte = pte_alloc(mm, pmd, address);
if (pte)
return handle_pte_fault(mm, vma, address, write_access, pte);
}
spin_unlock(&mm->page_table_lock);
return -1;
```

6.0.37 Function __pmd_alloc()

Prototype:

```
pmd_t *__pmd_alloc(struct mm_struct *mm,
                   pgd_t *pgd,
                   unsigned long address)


pmd_t *new;
```

```
/* "fast" allocation can happen without dropping the lock.. */
new = pmd_alloc_one_fast(mm, address);
if (!new) {
spin_unlock(&mm->page_table_lock);
new = pmd_alloc_one(mm, address);
spin_lock(&mm->page_table_lock);
if (!new)
return NULL;

/*
 * Because we dropped the lock, we should re-check the
 * entry, as somebody else could have populated it..
 */
if (!pgd_none(*pgd)) {
pmd_free(new);
goto out;
}
}
pgd_populate(mm, pgd, new);
out:
return pmd_offset(pgd, address);
```

6.0.38 Function pte_alloc()

Prototype:

```
pte_t *pte_alloc(struct mm_struct *mm,
                 pmd_t *pmd,
                 unsigned long address)


if (pmd_none(*pmd)) {
pte_t *new;

/* "fast" allocation can happen without dropping the lock.. */
new = pte_alloc_one_fast(mm, address);
if (!new) {
spin_unlock(&mm->page_table_lock);
new = pte_alloc_one(mm, address);
spin_lock(&mm->page_table_lock);
if (!new)
```

```
return NULL;

/*
 * Because we dropped the lock, we should re-check the
 * entry, as somebody else could have populated it..
 */
if (!pmd_none(*pmd)) {
pte_free(new);
goto out;
}
}
pmd_populate(mm, pmd, new);
}
out:
return pte_offset(pmd, address);
```

6.0.39 Function make_pages_present()

Prototype:

```
int make_pages_present(unsigned long addr,
                       unsigned long end)


int ret, len, write;
struct vm_area_struct * vma;

vma = find_vma(current->mm, addr);
write = (vma->vm_flags & VM_WRITE) != 0;
if (addr >= end)
BUG();
if (end > vma->vm_end)
BUG();
len = (end+PAGE_SIZE-1)/PAGE_SIZE-addr/PAGE_SIZE;
ret = get_user_pages(current, current->mm, addr,
len, write, 0, NULL, NULL);
return ret == len ? 0 : -1;
```

6.0.40 Function vmalloc_to_page()

Prototype:

```
struct page * vmalloc_to_page(void * vmalloc_addr)


unsigned long addr = (unsigned long) vmalloc_addr;
struct page *page = NULL;
pmd_t *pmd;
pte_t *pte;
pgd_t *pgd;

pgd = pgd_offset_k(addr);
if (!pgd_none(*pgd)) {
pmd = pmd_offset(pgd, addr);
if (!pmd_none(*pmd)) {
pte = pte_offset(pmd, addr);
if (pte_present(*pte)) {
page = pte_page(*pte);
}
}
}
return page;
}
```

Chapter 7

The Page Cache

7.1 The Buffer Cache

Chapter 8

Swapping

8.1 Structures

8.1.1 swp_entry_t

File: include/linux/shmem_fs.h

This type defines a swap entry address.

```
typedef struct {
        unsigned long val;
} swp_entry_t;
```

val
Stores the swap entry address. This address is dependant on the architecture and the arch-independent code uses some macros to handle these addresses:

SWP_ENTRY(type, offset)
Given a `type` and `offset`, returns a swap entry. On i386, the `type` is stored within the bits 1 and 7, the `offset` within 8 and 31. The bit 0 is used by the PRESENT bit, i.e, it is always zero.

SWP_TYPE(x)
From a swap entry, it returns its swap type.

SWP_OFFSET(x)
From a swap entry, it returns its swap offset.

8.1.2 struct swap_info_struct

File: include/linux/swap.h

This struct is defined for each swap area (partition or device). It holds all information about the swap area, like flags and the swap map used to assign and control swap entries.

```
struct swap_info_struct {
        unsigned int flags;
        kdev_t swap_device;
        spinlock_t sdev_lock;
        struct dentry * swap_file;
        struct vfsmount *swap_vfsmnt;
        unsigned short * swap_map;
        unsigned int lowest_bit;
        unsigned int highest_bit;
        unsigned int cluster_next;
        unsigned int cluster_nr;
        int prio;                       /* swap priority */
        int pages;
        unsigned long max;
        int next;                       /* next entry on swap list */
};
```

flags
: Used to mark this swap area as used (SWP_USED), writeable (SWP_WRITEOK) or unused (zero).

swap_device
: Pointer to the device if this area is a partition. It is NULL for swap files.

sdev_lock
: Spinlock that protects this struct and all its fields.

swap_file
: Pointer to the dentry of the partition or file.

swap_vfsmnt
: Pointer to the mount point.

swap_map
Array that holds information about all the swap entries. This information consists of a counter that, when zero, means that the entry is free.

lowest_bit
Stores the lowest offset within the `swap_map` which has a free entry (counter is zero).

highest_bit
Stores the highest offset within the `swap_map` which has a free entry (counter is zero).

cluster_next
Holds the next offset in the current swap cluster. This will be the starting point when checking the `swap_map` for a free entry.

cluster_nr
Accounts the number of missing entries to finish the current cluster.

prio
Priority of this swap area.

pages
Number of good pages on this swap area (i.e, total number minus the number of bad pages and the first block).

max
Maximum number of pages on this swap area.

next
Next swap area on the swap list.

8.2 Freeing Pages from Caches

8.2.1 LRU lists

The Linux VM architecture is composed of two LRU lists known as **active** and **inactive** lists. As soon as a page is added to the page cache (includes swap cache), it is added to the **inactive** list. The aging process tries to detect, through the page table entry bits and the page bits, which pages are the most accessed, moving them to the **active list**.

Under memory pressure scenarios, the VM first tries to free memory by reaping slab caches. When that procedure does not free enough memory, it focus on freeing memory from the page cache. Firstly, it checks the pages on the **active** list, moving the less accessed ones to the **inactive** list, refilling it. Secondly, the **inactive** list is scanned, synchronizing the pages with buffers and trying to free the freeable pages (i.e, pages without users). If dirty freeable pages are found, they are written out.

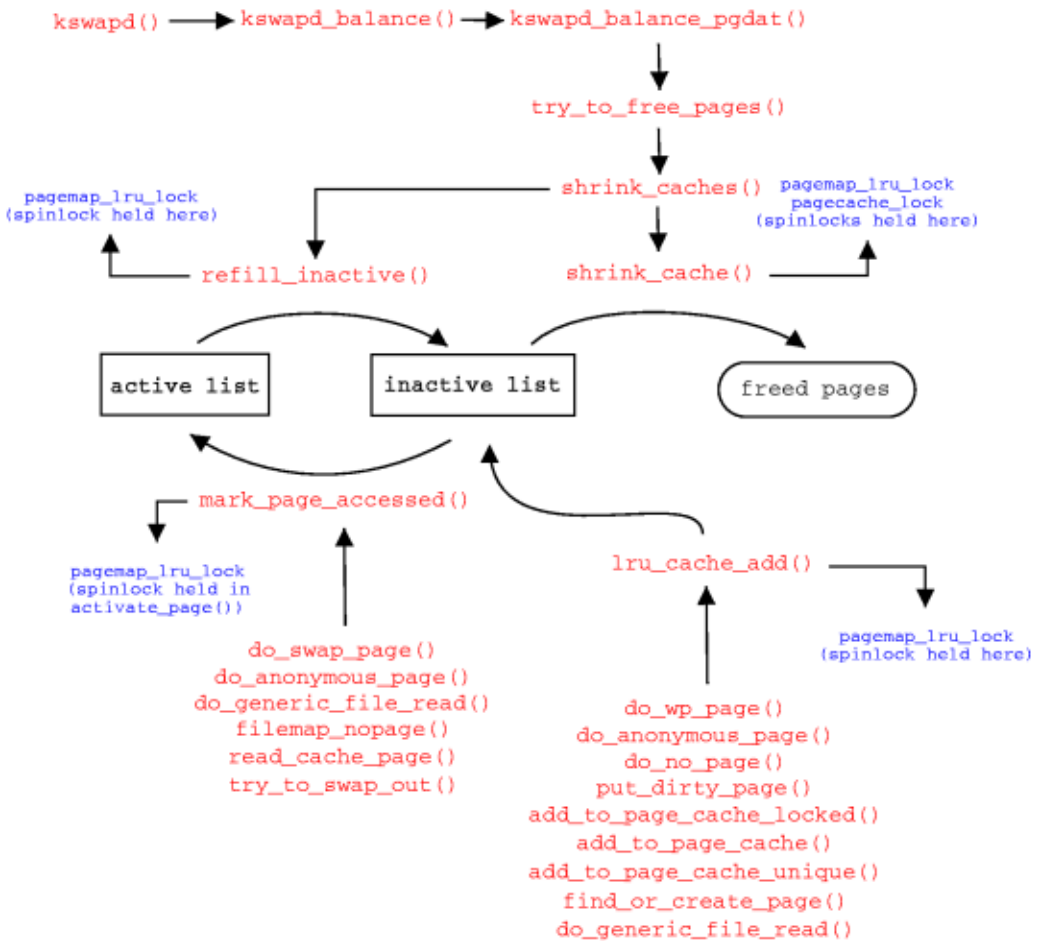

Figure 8.1: LRU lists

Nevertheless, pages on the **active** and **inactive** lists may have users, what usually means that they are mapped by processes. When many mapped pages are found on the **inactive** list, the unmapping process is invoked calling **swap_out()** function (check out the **Unmapping Pages from Processes** section).

As a last resource, if unable to free pages from the page cache, the VM

system shrinks the file system caches, like the inode cache, dentry cache and quota cache.

If still unable to free memory, the VM system runs into the out of memory scenario, where it picks an active process and tries to kill it to free memory.

8.2.2 Function shrink_cache()

File: mm/vmscan.c
Prototype:

```
int shrink_cache(int nr_pages,
                 zone_t * classzone,
                 unsigned int gfp_mask,
                 int priority)
```

This function shrinks the page and swap cache, checking the inactive list and trying to free pages from it. It may be needed to clean dirty pages by writing them, what will be done if possible (ie, `gfp_mask` allows).

The return value is an **int** value. If **zero**, it means that the function could free the number of pages requested previously (`nr_pages` parameter). If **not zero**, the value means how many pages were missed to free in order to achieve the requested number of pages. For example, a return value of 3 means that this function was able of free (`nr_pages` - 3) pages.

```
struct list_head * entry;
int max_scan = nr_inactive_pages / priority;
int max_mapped = min((nr_pages < < (10 - priority)),
                     max_scan / 10);
```

Here it is calculated how many pages at most will be scanned by this function if it cannot return first (`max_scan` variable). This value is based on the number of inactive pages (ie. pages on the inactive list) and on the priority. In this case, lower the priority, higher the number of pages that may be scanned.

The maximum number of mapped pages that can be found during the scan process is also computed here (`max_mapped` variable). It will be the maximum value between the `nr_pages` times a value dependant on the priority and a tenth of the `max_scan` value. Both values (`max_scan` and `max_mapped`) are known as *magic values*.

```
spin_lock(&pagemap_lru_lock);
while (--max_scan >= 0 &&
       (entry = inactive_list.prev) != &inactive_list) {
```

The while is very clear. It is scanned the minimum value between `max_scan` number of pages and the whole inactive list.

Two other return conditions will be found below. If the maximum number of mapped pages is reached or the requested number of pages has been freed, this function will return too.

```
struct page * page;

if (unlikely(current->need_resched)) {
        spin_unlock(&pagemap_lru_lock);
        __set_current_state(TASK_RUNNING);
        schedule();
        spin_lock(&pagemap_lru_lock);
        continue;
}
```

Improves fairness among process by rescheduling the process if it has been for a long time using CPU resources.

```
page = list_entry(entry, struct page, lru);

BUG_ON(!PageLRU(page));
BUG_ON(PageActive(page));

list_del(entry);
list_add(entry, &inactive_list);
```

Obtains the page from `struct list_head` pointer, and move it to the back of the inactive list.

```
/*
 * Zero page counts can happen because we unlink the pages
 * _after_ decrementing the usage count..
 */
if (unlikely(!page_count(page)))
        continue;
```

Since the page may be removed from lru lists after having its counter decremented, a race condition may happen: this page gets accessed here right after its counter is zeroed, but before being unlinked from the lru lists. The above `if` handles this case.

```
if (!memclass(page_zone(page), classzone))
        continue;
```

Only checks pages that are from the zone which is under pressure.

```
/* Racy check to avoid trylocking when not worthwhile */
if (!page->buffers && (page_count(page) != 1 ||
                                        !page->mapping))
        goto page_mapped;
```

Before trying to lock the page, first checks if it is mapped or anonymous and does not have buffers to be freed. If those conditions are true, account it as a mapped page (see below). In case it has buffers, even if mapped to processes, go on to try to free them (what may imply to synchronize).

```
/*
 * The page is locked. IO in progress?
 * Move it to the back of the list.
 */
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;
}
```

Tries to lock the page at once. If it is locked and PageLaunder bit is true, wait until it gets unlocked. PageLaunder bit will be only set for a page that has been submitted to IO in order to be cleaned in this function. Of course, wait on the page will only take place if `gfp_mask` allows it.

A reference on this page is got (`page_cache_get()`) before sleeping (to wait on the page) to ensure it will not be freed in the meanwhile.

There is an obsolete comment above (*Move it to the back of the list*). It does not make sense since the page has already been moved to the back of the list.

```
if (PageDirty(page) && is_page_cache_freeable(page) &&
                                        page->mapping) {
```

Dirty pages completely unmapped by processes that are in the page cache (swap cache is only a part of it) are eligible to be written to its backing storage. Even if a page table entry (pte) accesses this page, it will be only remapped after the IO is complete, ie, the faultin path is able lock this page.

```
                /*
                 * It is not critical here to write it only if
                 * the page is unmapped beause any direct writer
                 * like O_DIRECT would set the PG_dirty bitflag
                 * on the phisical page after having successfully
                 * pinned it and after the I/O to the page is
                 * finished, so the direct writes to the page
                 * cannot get lost.
                 */
                int (*writepage)(struct page *);

                writepage = page->mapping->a_ops->writepage;
                if ((gfp_mask & __GFP_FS) && writepage) {
                        ClearPageDirty(page);
                        SetPageLaunder(page);
                        page_cache_get(page);
                        spin_unlock(&pagemap_lru_lock);

                        writepage(page);
                        page_cache_release(page);

                        spin_lock(&pagemap_lru_lock);
                        continue;
                }
        }
```

Only pages from page cache which have the `writepage()` function defined can be cleaned. Actually, the `gfp_mask` is also checked to know if it allows this code path to perform FS operations. When both are true, the page PageLaunder bit is set, its Dirty bit is cleared, and the `writepage()` function is called. Here a reference to the page is got in order to avoid it to be ocasionally freed in the meanwhile.

```
        /*
         * If the page has buffers, try to free the buffer
         * mappings associated with this page. If we succeed
```

```
         * we try to free the page as well.
         */
        if (page->buffers) {
                spin_unlock(&pagemap_lru_lock);

                /* avoid to free a locked page */
                page_cache_get(page);

                if (try_to_release_page(page, gfp_mask)) {
```

Does the page have buffers? No matter if is completely unmapped from its processes, tries to free them by calling `try_to_release_page()` function which will eventually call `try_to_free_buffers()`. The latter function will free the buffers if they are clean, otherwise will synchronize them (`gfp_mask` must allow that).

```
                        if (!page->mapping) {
                                /*
                                 * We must not allow an anon page
                                 * with no buffers to be visible
                                 * on the LRU, so we unlock the
                                 * page after taking the lru lock
                                 */
                                spin_lock(&pagemap_lru_lock);
                                UnlockPage(page);
                                __lru_cache_del(page);

                                /* effectively free the page here */
                                page_cache_release(page);

                                if (--nr_pages)
                                        continue;
                                break;
```

The page has had its buffers freed. Is it an anonymous page? In order to be an anonymous page with buffers, it must have already been unmapped from all processes that have mapped it beforehand. It has also been removed from the page cache since its mapping had to invalidate/truncate its pages. In this case, simply remove the page from the inactive list and release it.

```
                        } else {
```

```
                                /*
                                 * The page is still in pagecache
                                 * so undo the stuff before the
                                 * try_to_release_page since we've
                                 * not finished and we can now
                                 * try the next step.
                                 */
                                page_cache_release(page);

                                spin_lock(&pagemap_lru_lock);
                        }
```

The pages's buffers are gone, so goes on to the next step since the page is still in the page cache. It needs to be removed from the page cache if it is completely unmapped from process. Otherwise, gives up on it since it is still to be unmapped and it is not freeable at this moment.

```
                } else {
                        /* failed to drop the buffers
                         * so stop here */
                        UnlockPage(page);
                        page_cache_release(page);

                        spin_lock(&pagemap_lru_lock);
                        continue;
                }
        }
```

The buffers could not be freed, so gives up on this page. It is time to try another page from the inactive list.

```
        spin_lock(&pagecache_lock);

        /*
         * this is the non-racy check for busy page.
         */
        if (!page->mapping || !is_page_cache_freeable(page)) {
                spin_unlock(&pagecache_lock);
                UnlockPage(page);
page_mapped:
                if (--max_mapped >= 0)
                        continue;
```

For anonymous pages without buffers, that is a race check since they probably have been removed from the page cache in the meantime. For pages from page cache that just had its buffers freed and are still mapped by processes, accounts them to the `max_mapped` variable.

```
                /*
                 * Alert! We've found too many mapped pages on the
                 * inactive list, so we start swapping out now!
                 */
                spin_unlock(&pagemap_lru_lock);
                swap_out(priority, gfp_mask, classzone);
                return nr_pages;
        }
```

When a `max_mapped` number of pages have been observed to be mapped to processes, starts unmapping pages that are still mapped to processes. That is why `swap_out()` function is called here. After it gets called, returns since reaching a `max_mapped` number of mapped pages is one of the conditions to stop the scan process.

```
        /*
         * It is critical to check PageDirty _after_ we made sure
         * the page is freeable* so not in use by anybody.
         */
        if (PageDirty(page)) {
                spin_unlock(&pagecache_lock);
                UnlockPage(page);
                continue;
        }
```

Checks once again for the dirtiness of the page since it might have been set dirty right after being unmapped by any process (for example, in memory.c:__free_pte()).

```
        /* point of no return */
        if (likely(!PageSwapCache(page))) {
                __remove_inode_page(page);
                spin_unlock(&pagecache_lock);
        } else {
                swp_entry_t swap;
                swap.val = page->index;
                __delete_from_swap_cache(page);
```

```
                spin_unlock(&pagecache_lock);
                swap_free(swap);
        }

        __lru_cache_del(page);
        UnlockPage(page);

        /* effectively free the page here */
        page_cache_release(page);

        if (--nr_pages)
                continue;
        break;
}
spin_unlock(&pagemap_lru_lock);

return nr_pages;
```

That is the part of the code where the page is not mapped by any process, it is not dirty and does not have buffers, so it can be removed from page cache (removing from swap cache will remove from the page cache anyway), deleted from LRU lists (inactive list) and freed.

8.2.3 Function refill_inactive()

File: mm/vmscan.c
Prototype:

```
void refill_inactive(int nr_pages)
```

This function tries to to move a requested number of pages (`nr_pages`) from the active list to the inactive list. It also updates the aging of every page checked. The aging is represented by the `Referenced` bit.

```
struct list_head * entry;

spin_lock(&pagemap_lru_lock);
entry = active_list.prev;
while (nr_pages && entry != &active_list) {
```

Stops when all the pages on the active list have been scanned or the requested number of pages has been moved to the inactive 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;
    }
```

Pages with `Referenced` bit on are likely to have been accessed recently, so clear this bit and move them to the back of active list since they are likely to be accessed soon again.

```
    nr_pages--;

    del_page_from_active_list(page);
    add_page_to_inactive_list(page);
    SetPageReferenced(page);
}
spin_unlock(&pagemap_lru_lock);
```

Pages that do not have `Referenced` bit on are taken as old pages, so can moved to inactive list. Mark this page as `Referenced`, so if they are accessed when on the inactive list, they will be moved back to active list at the first access.

8.2.4 Function shrink_caches()

File: mm/vmscan.c
Prototype:

```
int shrink_caches(zone_t * classzone,
                  int priority,
                  unsigned int gfp_mask,
                  int nr_pages)
```

Very important role in the page freeing process, this function defines the priority for each type of memory (slab caches, page and swap caches, dentry cache, inode cache and quota cache), trying to free the pages in the order previously set.

Given a zone (`classzone` parameter), this very function tries to free the requested number of pages (`nr_pages` parameter), following a GFP mask for permissions throughout the freeing process (`gfp_mask`) and a priority that is used to know how hard it must tried to free pages from a certain type of memory.

The return value is an integer value. A zero value means that the requested number of pages has been freed. A non-zero value is the number of pages missed to achieve the requested number of pages.

```
int chunk_size = nr_pages;
```

The requested number of pages to be freed is stored in `chunk_size` variable, since it may be changed and the original value will be needed below.

```
unsigned long ratio;

nr_pages -= kmem_cache_reap(gfp_mask);
```

The first try is to reap all the slab caches that can be reaped (that is defined when creating a slab cache). Thus, all those slab caches will free the memory pages that have only unused data structures.

```
if (nr_pages <= 0)
        return 0;
```

When only reaping the slab caches could free all the requested number of pages, return.

```
nr_pages = chunk_size;
```

For many times, reaping the slab caches will not make the requested number of pages, so try to free the original number of pages from other types of pages (page and swap cache). Restoring the original number of pages instead of using the missing number of pages is used since `shrink_cache()` (to be called) may write out memory pages and if that happens, it is nice to write a chunk of them.

```
/* try to keep the active list 2/3 of the size of the cache */
ratio = (unsigned long) nr_pages *
        nr_active_pages / ((nr_inactive_pages + 1) * 2);
refill_inactive(ratio);
```

The first step to free pages from page and swap caches is to refill the inactive list, since only pages from this list are freed. In order to keep the active list not empty, it is computed how many pages (at most) should be moved to inactive list (`ratio` variable).

Note: one is added to the number of inactive page (`nr_inactive_pages + 1`) to handle the case where the `nr_inactive_pages` is zero.

```
nr_pages = shrink_cache(nr_pages, classzone, gfp_mask, priority);
```

No matter the inactive list has been refilled or not, calls `shrink_cache()` function to shrink the page and swap caches.

```
if (nr_pages <= 0)
        return 0;
```

If all the requested number of pages has been freed from page and swap caches, return.

```
shrink_dcache_memory(priority, gfp_mask);
shrink_icache_memory(priority, gfp_mask);
#ifdef CONFIG_QUOTA
shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
#endif

return nr_pages;
```

As a last try, shrink the dentry cache, the inode cache and also the quota cache (if quota is enabled). Even if these caches have been shrunk, return as having failed (return the number of missed pages to achieve the original requested number). This last try is done to free some memory and avoid many failed allocations, but they will not avoid calling `out_of_memory()` if that's the case (check below).

8.2.5 Function try_to_free_pages()

Prototype:

```
int try_to_free_pages(zone_t *classzone,
                      unsigned int gfp_mask,
                      unsigned int order)
```

Simple function that tries to free pages from a certain zone (`classzone` parameter) by calling `shrink_caches()`, increasing priority if necessary. The `shrink_caches()` will follow the GFP mask (`gfp_mask` parameter).

In the case it has not been able to free the defined number of pages (`SWAP_CLUSTER_MAX`), calls `out_of_memory()` function which may kill some application.

It returns an **int** value. A value of **one** means that this function was sucessful freeing the defined number of pages and **zero** if it has failed.

The `order` parameter is unused.

```
int priority = DEF_PRIORITY;
int nr_pages = SWAP_CLUSTER_MAX;

gfp_mask = pf_gfp_mask(gfp_mask);
```

If the current task cannot block on IO operations, the `pf_gfp_mask()` macro makes sure the `gfp_mask` signs that.

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

Starting with the lowest priority, tries to free the defined number of pages. If it couldn't make it, increases priority (by decreasing the `priority` variable) and try again.

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

Couldn't free the enough number of pages, even with the highest priority? Checks if it is time of kill some application calling `out_of_memory()` function.

8.3 Unmapping Pages from Processes

Systems the have many mapped pages on the **inactive** list (see **Freeing Pages from Caches** section) must start to unmap pages from the processes. It means that process will start to have their page tables scanned and all the page table entries checked. Entries not accessed recently will be cleared (unmapped) and, when previously set to anonymous pages, set to a new address (remapped). Anonymous pages are pages without a backing store.

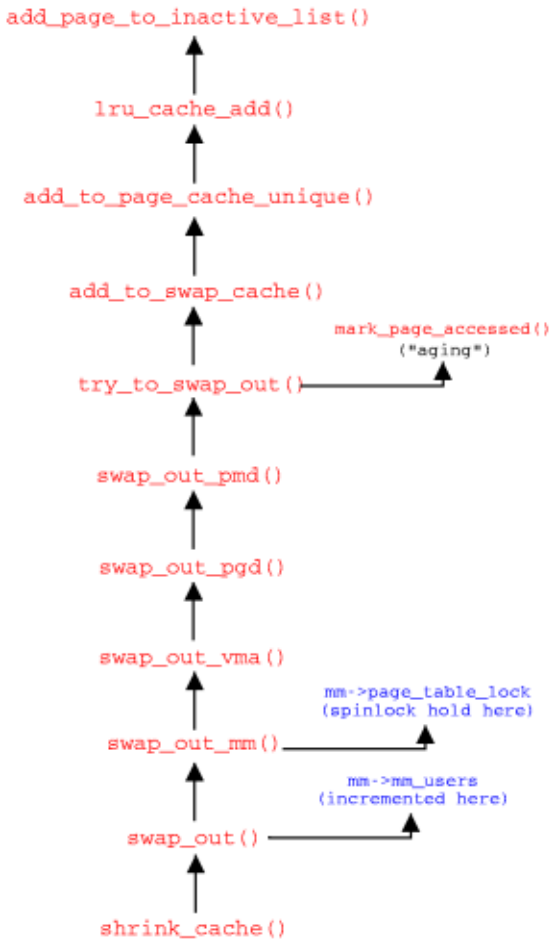

Figure 8.2: Unmapping Process

8.3.1 Function try_to_swap_out()

File: mm/vmscan.c
Prototype:

```
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)
```

The role of the `try_to_swap_out()` function is to try to unmap pages from processes mapping them. This is the first part of the whole swap out process, since pages can only be freed if all processes mapping them have already been safely unmapped. Unmapping means that, given a page table entry (`pte`), either it is just cleared (file mapped pages) or remapped to a swap address (anonymous pages). In both cases (cleared or remapped to swap address), the present bit of the new pte will be off. Therefore, the process to which this pte belongs will not be able to access it directly, causing a page fault for any future access.

This function returns an **int** value. That value will be **zero** if no freeable page (ie, a page not mapped by any process any longer) has been freed. That will happen even in the case a page got unmapped from a process, but is still mapped by other processes. That return value will be **one** if a page has been freed from its last process (no process is mapping it at the moment this function exits).

```
pte_t pte;
swp_entry_t entry;

/* Don't look at this pte if it's been accessed recently. */
if ((vma->vm_flags & VM_LOCKED)
    || ptep_test_and_clear_young(page_table)) {
        mark_page_accessed(page);
        return 0;
}
```

That is part of VM aging process. Here, based on the young bit from the pte, `try_to_swap_out()` sets this page as accessed (Accessed bit). If this page is already set as accessed (i.e, the second time it is set accessed) and it happens that it is still on inactive list, `mark_page_accessed()` will move this page to the active list. The page previously set as accessed will have its Accessed bit cleared though.

The page will also be marked as accessed if this vm area is locked by `mlock` system call.

```
/* Don't bother unmapping pages that are active */
if (PageActive(page))
        return 0;
```

Active pages are supposed to have been accessed often. Therefore, it is worthless to unmap them since it is likely they will be mapped back soon.

```
/* Don't bother replenishing zones not under pressure.. */
if (!memclass(page_zone(page), classzone))
        return 0;
```

It is unreasonable free pages from zones other that the ones that are under memory shortage.

```
if (TryLockPage(page))
        return 0;
```

The page is tried to lock at once (i.e, do not sleep to get lock on this page) given that the unmapping process is not dependant on an specific page and it is not worth to sleep to try to unmap any page.

```
/* From this point on, the odds are that we're going to
 * nuke this pte, so read and clear the pte.  This hook
 * is needed on CPUs which update the accessed and dirty
 * bits in hardware.
 */
flush_cache_page(vma, address);
pte = ptep_get_and_clear(page_table);
flush_tlb_page(vma, address);
```

Read the page table entry data into pte. Also clear it in the page table to avoid having this pte modified in the meanwhile (for example in cases where CPUs that update bits like accessed and dirty in hardware, like explained in the comment).

```
if (pte_dirty(pte))
        set_page_dirty(page);

/*
 * Is the page already in the swap cache? If so, then
 * we can just drop our reference to it without doing
 * any IO - it's already up-to-date on disk.
```

```
 */
if (PageSwapCache(page)) {
        entry.val = page->index;
        swap_duplicate(entry);
set_swap_pte:
        set_pte(page_table, swp_entry_to_pte(entry));
```

In the case this page has already been added to the swap cache, there is only the need to increase the swap entry counter (`swap_duplicate()`) and set this pte to the swap address this swap cache page is already set to. The swap address is stored in the `index` field of the `struct page`.

```
drop_pte:
        mm->rss--;
```

The process which has this page unmapped will have its RSS number decreased.

```
        UnlockPage(page);
        {
                int freeable = page_count(page) - !!page->buffers <= 2;
                page_cache_release(page);
                return freeable;
        }
}
```

If there are no more users of this page (including processes mapping it), the return value will be **one**, since this page is completely unmapped from the processes and can be freed. Otherwise, the return value will be **zero**.

```
/*
 * Is it a clean page? Then it must be recoverable
 * by just paging it in again, and we can just drop
 * it..  or if it's dirty but has backing store,
 * just mark the page dirty and drop it.
 *
 * However, this won't actually free any real
 * memory, as the page will just be in the page cache
 * somewhere, and as such we should just continue
 * our scan.
 *
 * Basically, this just makes it possible for us to do
```

```
 * some real work in the future in "refill_inactive()".
 */
if (page->mapping)
        goto drop_pte;
if (!PageDirty(page))
        goto drop_pte;

/*
 * Anonymous buffercache pages can be left behind by
 * concurrent truncate and pagefault.
 */
if (page->buffers)
        goto preserve;
```

Anonymous pages are pages without backing store, ie. not mapped to any address space. Anonymous buffer cache pages are anonymous pages with buffers. In particular, these pages have already been mapped to an address space, but aren't any longer because of a concurrent truncate operation and page fault.

```
/*
 * This is a dirty, swappable page.  First of all,
 * get a suitable swap entry for it, and make sure
 * we have the swap cache set up to associate the
 * page with that swap entry.
 */
for (;;) {
        entry = get_swap_page();
        if (!entry.val)
                break;
        /* Add it to the swap cache and mark it dirty
         * (adding to the page cache will clear the dirty
         * and uptodate bits, so we need to do it again)
         */
        if (add_to_swap_cache(page, entry) == 0) {
                SetPageUptodate(page);
                set_page_dirty(page);
                goto set_swap_pte;
        }
```

That is a dirty and anonymous page, so let's get a swap entry for it in order to remap its pte to this new address. Once a swap entry has been got,

this page will be added to the swap cache, which will, for its turn, add it to the page cache and also a LRU list (actually the inactive one).

Given that this page has no backing store (recall it is an anonymous page), this page needs to be set as dirty in order to not be released without being stored on the swap.

```
        /* Raced with "speculative" read_swap_cache_async */
        swap_free(entry);
}
```

When servicing a page fault for a swap address, some pages are read ahead if the page is not present in the swap cache. In this case, a page might have been added to the swap cache by the read ahead code (to be read from disk) with the very swap entry just got above, but before this code path could add it to the cache. Thus it is necessary to drop the counter of this swap entry and get a new one.

```
/* No swap space left */
preserve:
set_pte(page_table, pte);
UnlockPage(page);
return 0;
```

A free swap entry was not available, so no swap space is left. Hence `try_to_swap_out()` is unable to unmap this anonymous page. So, sets the page table entry back to the original value and returns zero since no freeable page has been unmapped after this try.

8.3.2 Function swap_out_pmd()

File: mm/vmscan.c
Prototype:

```
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)
```

This function scans all the page table entries of a page middle directory (`dir` parameter) until the end of the page middle directory or the end of the vm area. It returns an **int** value, which is the number of pages missed to reach the requested number of completely unmapped pages (`count` parameter).

```
pte_t * pte;
unsigned long pmd_end;

if (pmd_none(*dir))
        return count;
```

Returns the original count value when the page middle directory points to no page table.

```
if (pmd_bad(*dir)) {
        pmd_ERROR(*dir);
        pmd_clear(dir);
        return count;
}
```

Checks if the contents of this memory address points to a valid page table. In this case, prints an error message (`pmd_ERROR()`), clears this entry and returns.

```
pte = pte_offset(dir, address);
```

From the page middle directory pointer and the address, gets the pointer to a page table entry.

```
pmd_end = (address + PMD_SIZE) & PMD_MASK;
if (end > pmd_end)
        end = pmd_end;
```

Computes the end of the page table pointed by this page middle directory entry. If the end of the VM area is beyond the end of this page table, set this variable (`end`) to this value.

```
do {
        if (pte_present(*pte)) {
```

Only page table entries that are mapped to pages in memory can be unmapped. That way, page table entries set to no page or set to swap addresses are not scanned by `try_to_swap_out()`.

```
                struct page *page = pte_page(*pte);

                if (VALID_PAGE(page) && !PageReserved(page)) {
```

Given the page table entry (pte), gets the page to which this page table entry is mapped and checks if it is a valid address as well as if it not reserved. Reserved pages cannot be unmapped from their processes.

```
                    count -= try_to_swap_out(mm, vma,
                                address, pte, page, classzone);
```

Calls `try_to_swap_out()` function that will try to unmap this page (`page` variable) from the page table entry which is mapped to it. It will return 1 if it has unmapped and this page is freeable (i.e, does not have other users). The return value of 0 does not mean that the page has not been unmapped, but simply that it is not freeable (ie, it has other users, probably other processes still map it).

```
                if (!count) {
                        address += PAGE_SIZE;
                        break;
                }
```

If the initial requested number of completely unmapped pages have been reached, return. Since the `swap_address` (the last address scanned to be swapped out) of the `mm_struct` is updated here, adds the page size to the address variable, since this address has been scanned and the scan should be resumed from the next address.

```
                }
        }
        address += PAGE_SIZE;
        pte++;
} while (address && (address < end));
```

Updates the `address` variable to the next address and the `pte` pointer to the next offset to be scanned. Stops the loop when `address` is beyond the VM area.

```
mm->swap_address = address;
return count;
```

Updates the last address scanned in `mm_struct` and return the `count` variable, which informs the number of pages missed to reach the inital requested number of completely unmapped pages.

8.3.3 Function swap_out_pgd()

File: mm/vmscan.c
Prototype:

```
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)
```

This function scans all the page middle directories of a page global directory offset (`dir` parameter) until the end of the vm area (`vma` parameter). It returns an int value, which means how many pages are missing to the requested number of completely unmapped pages (`count` parameter).

```
pmd_t * pmd;
unsigned long pgd_end;

if (pgd_none(*dir))
        return count;
```

Returns the original `count` value when the page global directory points to no page middle directory.

```
if (pgd_bad(*dir)) {
        pgd_ERROR(*dir);
        pgd_clear(dir);
        return count;
}
```

Checks if the entry points to a bad page table. In this case, prints that an error message (`pgd_ERROR()`), clears this entry and returns the original `count` value.

```
pmd = pmd_offset(dir, address);
```

From the page global directory entry and the address, gets the pointer to the page table to be scanned.

```
pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
if (pgd_end && (end > pgd_end))
        end = pgd_end;
```

Obtains the end of the space addressable by this page global directory. If the end of the VM area is greater than the end of this page global directory, the new `end` will be the page global directory boundary.

```
do {
        count = swap_out_pmd(mm, vma, pmd, address, end, count,
                             classzone);
```

For every page table (until the end of the vm area), scans all its page table entries in `swap_out_pmd()` function.

```
        if (!count)
                break;
```

The return value of `swap_out_pmd()` function tells how many pages still need to be unmapped to reach the initial requested number. If all the needed pages have been unmapped, stops scanning and return.

```
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
} while (address && (address < end));
return count;
```

Goes to the next page middle directory, updating the start address of it (`address` variable) and the offset within the page global directory (`pmd` variable).

8.3.4 Function swap_out_vma()

File: mm/vmscan.c
Prototype:

```
int swap_out_vma(struct mm_struct * mm,
                 struct vm_area_struct * vma,
                 unsigned long address,
                 int count, zone_t * classzone)
```

This function scans a VM area (`vma` parameter), returning the number of missing pages to reach the requested number of completely unmapped pages (`count` parameter).

```
pgd_t *pgdir;
unsigned long end;

/* Don't swap out areas which are reserved */
if (vma->vm_flags & VM_RESERVED)
        return count;
```

Some special cases (usually drivers) define the VM area as reserved (`VM_RESERVED` flag) to avoid this VM area to have its entries unmapped.

```
pgdir = pgd_offset(mm, address);
```

Now, based on the `mm_struct` and the `address` to be scanned, gets the page middle directory (actually the offset within the page global directory) to scan.

```
end = vma->vm_end;
BUG_ON(address >= end);
do {
        count = swap_out_pgd(mm, vma, pgdir, address, end,
                             count, classzone);
```

Calls `swap_out_pgd()` to swap out the page middle directory. The `address` and `end` parameters tell the beginning and the end of the memory address to be scanned.

```
        if (!count)
                break;
```

Leave the loop and return if all the requested number of pages have been completely unmapped.

```
        address = (address + PGDIR_SIZE) & PGDIR_MASK;
        pgdir++;
} while (address && (address < end));
return count;
```

Updates the `address` variable to the next page middle directory and `pgdir` variable to the next offset within the page global directory. If it did not reach the end of this VM area address space, scans the next page middle directory.

8.3.5 Function swap_out_mm()

File: mm/vmscan.c
Prototype:

```
int swap_out_mm(struct mm_struct * mm,
                int count, int * mmcounter,
                zone_t * classzone)
```

This function scans all the VM areas from a process (`mm` parameter). It returns how many pages were missing to get to the initial request amount (`count` parameter). If return value is zero, it means that all the requested pages were completely unmapped.

```
unsigned long address;
struct vm_area_struct* vma;

/*
 * Find the proper vm-area after freezing the vma chain
 * and ptes.
 */
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;
}
```

Checks for a race condition. Before getting the `mm->page_table_lock`, another code path might have been faster and scanned all the address space of this task (`address == TASK_SIZE` condition).

It could also have been completely scanned and another `mm_struct` is the current one to be scanned (`swap_mm != mm` condition). In either case, increments the `mmcounter` variable in order to make the caller function to try this process again (if unable to unmap the necessary pages checking other processes first).

```
vma = find_vma(mm, address);
```

Finds the first VM area that ends after this `address`.

```
if (vma) {
        if (address < vma->vm_start)
                address = vma->vm_start;
```

Sets `address` variable to the beginning of the found VM area if it does not belong to a VM area.

```
        for (;;) {
                count = swap_out_vma(mm, vma, address, count,
                                     classzone);
```

Calls `swap_out_vma()` to scan all the VM area address space.

```
                vma = vma->vm_next;
                if (!vma)
                        break;
                if (!count)
                        goto out_unlock;
                address = vma->vm_start;
        }
}
```

Sets `vma` variable to the next VM area. Leave the `while` loop if there are no VM areas to be scanned. If the number of pages to be unmapped has been reached, leave the function since this VM area scan does not need to be continued. Otherwise, set `address` to the beginning of the next VM area and go on.

```
/* Indicate that we reached the end of address space */
mm->swap_address = TASK_SIZE;
```

All VM areas have been scanned, so sets the `swap_address` of this process's `mm_struct` to `TASK_SIZE` to mark it as having been completely scanned.

```
out_unlock:
spin_unlock(&mm->page_table_lock);
return count;
```

Returns the number of pages missing to the initial amount of pages to be completely unmapped.

8.3.6 Function swap_out()

File: mm/vmscan.c
Prototype:

```
int swap_out(unsigned int priority,
             unsigned int gfp_mask,
             zone_t * classzone)
```

This function picks a task which will have its page table scanned. More specifically, a mm_struct – which is one per task – is chosen.

The return value is an **int** which means that the number of requested pages (defined below) has been achieved (returns **one**) or not (returns **zero**).

```
int counter, nr_pages = SWAP_CLUSTER_MAX;
```

Tries to unmap nr_pages pages from the process to be selected.

```
struct mm_struct *mm;

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

Improves fairness by rescheduling the current task.

```
        spin_lock(&mmlist_lock);
        mm = swap_mm;
```

Sets mm variable to the latest mm_struct which has been scanned (or is being scanned).

```
        while (mm->swap_address == TASK_SIZE ||
               mm == &init_mm) {
```

No task has ever been scanned (mm == &init_mm condition) or all the address space of the selected task has been scanned (mm->swap_address == TASK_SIZE condition). In any of theses cases, tries to get a new mm_struct to scan.

```
                mm->swap_address = 0;
```

Makes this mm_struct available again to be scanned.

```
            mm = list_entry(mm->mmlist.next, struct mm_struct,
                                                        mmlist);
```

From the list of all active mm_struct, mmlist, picks the next mm_struct.

```
            if (mm == swap_mm)
                    goto empty;
```

In the case the list is empty, returns.

```
            swap_mm = mm;
    }
    /* Make sure the mm doesn't disappear
       when we drop the lock.. */
    atomic_inc(&mm->mm_users);
    spin_unlock(&mmlist_lock);

    nr_pages = swap_out_mm(mm, nr_pages, &counter,
                                            classzone);
```

Chosen an mm_struct(), call swap_out_mm() function, which will "swap out" the VM areas of this mm_struct.

```
    mmput(mm);
```

Once all the VM areas have been scanned, decrement the mm_struct counter, deleting if it is the last reference.

```
    if (!nr_pages)
            return 1;
```

No remaining pages to be unmapped? This function is sucessful, so it is time to return.

```
} while (--counter >= 0);

return 0;

empty:
spin_unlock(&mmlist_lock);
return 0;
```

Unsuccesful since either the mmlist does not have other mm_structs or even after scanning all mmlist, it was unable to unmap the requested number of pages.

8.4 Checking Memory Pressure

8.4.1 Function check_classzone_need_balance()

File: mm/vmscan.c
Prototype:

```
int check_classzone_need_balance(zone_t * classzone)
```

The role of this function is to check if a zone (`classzone` parameter) and all zones below it need to be balances, i.e, if the number of free pages is lower than the higher watermark (`classzone->pages_high`).

Its return value is an **int**, which indicates if it needs balance (**one**) or not (**zero**).

```
zone_t * first_classzone;

first_classzone = classzone->zone_pgdat->node_zones;
```

The `first_classzone` is set to the first zone on this NUMA (Non-Uniform Memory Architecture) node.

```
while (classzone >= first_classzone) {
        if (classzone->free_pages > classzone->pages_high)
                return 0;
        classzone--;
}
return 1;
```

If the zone (or any zone below it) does not need to be balanced, returns zero. Otherwise checks until the first zone, returning one.

8.4.2 Function kswapd_balance_pgdat()

File: mm/vmscan.c
Prototype:

```
int kswapd_balance_pgdat(pg_data_t * pgdat)
```

This function balances every zone from an NUMA node (`pgdat` parameter) that has the `need_balance` flag set.

Its return value is an **int**, which will be **one** if any zone still need to be balanced after it is tried to free pages and the pages could not be freed. A **zero** return value means that all zones are balanced (either were already balanced or have been balanced).

```
int need_more_balance = 0, i;
zone_t * zone;

for (i = pgdat->nr_zones-1; i >= 0; i--) {
        zone = pgdat->node_zones + i;
```

For every zone on this node.

```
        if (unlikely(current->need_resched))
                schedule();
```

Fairness is improved by relinquishing CPU if that is needed.

```
        if (!zone->need_balance)
                continue;
```

Zones that do not need to be balanced can be skipped. This flag (`need_balance`) is set in `page_alloc.c:__alloc_pages()` whenever a zone (on all nodes) has a number of free pages smaller than the low watermark (`zone->free_pages`).

```
        if (!try_to_free_pages(zone, GFP_KSWAPD, 0)) {
                zone->need_balance = 0;
                __set_current_state(TASK_INTERRUPTIBLE);
                schedule_timeout(HZ);
                continue;
        }
```

This zone needs to be balanced, so calls `try_to_free_pages()` function. If it could free its defined number of pages, zeroes `need_balance` variable and reschedule the current task.

```
        if (check_classzone_need_balance(zone))
                need_more_balance = 1;
        else
                zone->need_balance = 0;
}
```

It could not free the defined number of pages for this zone, so check if this zone (or any below it) still needs to be balanced.

Note that `zone->need_balance` has been set to 1 when the number of free pages was lower than the low watermark. In order to return zero, the `check_classzone_need_balance()` function checks if the zone has a number of free pages higher than the **high** watermark.

```
return need_more_balance;
```

If `try_to_free_pages()` failed to free pages and any zone still needs to be balanced, according to `check_classzone_need_balance()` function, return 1. Otherwise, return 0.

8.4.3 Function kswapd_balance()

File: mm/vmscan.c
Prototype:

```
void kswapd_balance(void)
```

Main function called from `kswapd()`, `kswapd_balance()` simply balances every node on the system, looping untill all node are balanced.

```
int need_more_balance;
pg_data_t * pgdat;

do {
        need_more_balance = 0;
        pgdat = pgdat_list;
        do
                need_more_balance |= kswapd_balance_pgdat(pgdat);
        while ((pgdat = pgdat->node_next));
} while (need_more_balance);
```

8.4.4 Function kswapd_can_sleep_pgdat()

File: mm/vmscan.c
Prototype:

```
int kswapd_can_sleep_pgdat(pg_data_t * pgdat)
```

Auxiliar function used by `kswapd_can_sleep()` to know if a certain node (`pgdat` parameter) needs to be balanced. If any zone needs to be balanced, the **int** return value will be **zero**, since kswapd cannot sleep. If no node needs to be balanced, the return value will be **one**.

```
zone_t * zone;
int i;

for (i = pgdat->nr_zones-1; i >= 0; i--) {
```

```
        zone = pgdat->node_zones + i;
        if (!zone->need_balance)
                continue;
        return 0;
}

return 1;
```

8.4.5 Function kswapd_can_sleep()

File: mm/vmscan.c
Prototype:

```
int kswapd_can_sleep(void)
```

This function is used by `kswapd()` to know if any node on the system has to be balanced. If that happens, `kswap()` cannot sleep, thus the return value (an **int**) will be **zero**. Otherwise, the return value will be **one**.

```
pg_data_t * pgdat;

pgdat = pgdat_list;
do {
        if (kswapd_can_sleep_pgdat(pgdat))
                continue;
        return 0;
} while ((pgdat = pgdat->node_next));

return 1;
```

8.4.6 Function kswapd()

File: mm/vmscan.c
Prototype:

```
int kswapd(void *unused)
```

The `kswap()` function is run as a kernel thread. Its main role in the virtual memory system is to perform the pageout process when there is need, what happens usually when a zone is under a certain limit of free available memory pages to be used for allocation. This process is done on a per-zone basis for each node (if it is a NUMA system).

```
struct task_struct *tsk = current;
DECLARE_WAITQUEUE(wait, tsk);
```

A wait queue is declared to be added to the `kswapd_wait` wait queue header below. It will be used by `__alloc_pages()` to know if it can actually wake up kswapd process.

```
daemonize();
strcpy(tsk->comm, "kswapd");
sigfillset(&tsk->blocked);

/*
 * Tell the memory management that we're a "memory allocator",
 * and that if we need more memory we should get access to it
 * regardless (see "__alloc_pages()"). "kswapd" should
 * never get caught in the normal page freeing logic.
 *
 * (Kswapd normally doesn't need memory anyway, but sometimes
 * you need a small amount of memory in order to be able to
 * page out something else, and this flag essentially protects
 * us from recursively trying to free more memory as we're
 * trying to free the first piece of memory in the first place).
 */
tsk->flags |= PF_MEMALLOC;

/*
 * Kswapd main loop.
 */
for (;;) {
    __set_current_state(TASK_INTERRUPTIBLE);
```

Sets the task flag to be interruptible if `kswap` sleeps below.

```
    add_wait_queue(&kswapd_wait, &wait);
```

Adding `wait` to the `kswapd_wait` wait queue header turns `kswapd_wait` into an **active** wait queue. In the case any allocation happens from now on and the number of free pages in any zone is under the minimum limit, `kswap` will be able to be awaken (if it is sleeping).

```
    mb();
```

This stands for *memory barrier* and ensures that memory ordering will happen, ie, that on a SMP system each CPU has the same view of the memory.

```
    if (kswapd_can_sleep())
        schedule();
```

Here `kswapd` checks if any zone in any node needs to be balanced. A zone will be marked "to be balanced" (ie. `zone->need_balance = 1`) if when page allocation happens, we have memory shortage, ie. the number of free pages is lower than the minimum watermark for the respective zone. In the case no zone has to be balanced, `kswapd` sleeps by calling `schedule()`.

```
    __set_current_state(TASK_RUNNING);
    remove_wait_queue(&kswapd_wait, &wait);
```

After `kswapd` is waken up, its current task state is set to `TASK_RUNNING` and `wait` is removed from `kswapd_wait` wait queue header. Thus, `__alloc_pages()` will not be able to wake `kswapd` any longer, since that `kswapd` will be already running trying to balance all zones under memory shortage.

```
    /*
     * If we actually get into a low-memory situation,
     * the processes needing more memory will wake us
     * up on a more timely basis.
     */
    kswapd_balance();
```

That's the part of `kswapd` where it actually does its work. The `kswapd_balance()` tries to free enough pages for each zone which needs to be balanced. Enough here stands for freeing a number of pages to make the zone to end up having more free pages than its minimum watermark.

```
    run_task_queue(&tq_disk);
}
```

Now it runs the task queue `tq_disk` to perform disk related bottom half activities. Since `kswapd` might have written some pages, it would be very nice that they get flushed in order to be freed soon.

8.4.7 Function kswapd_init()

File: mm/vmscan.c
Prototype:

```
static int __init kswapd_init(void)
```

This initialization function simply performs any necessary swap setup and starts `kswapd()` function as a kernel thread.

```
printk("Starting kswapd\n");
swap_setup();
kernel_thread(kswapd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
return 0;
```

8.5 Handling Swap Entries

8.5.1 Function scan_swap_map()

File: mm/swapfile.c
Prototype:

```
static inline int scan_swap_map(struct swap_info_struct *si)
```

This function scans the swap map of this swap device or partition (defined in the `si` swap info struct parameter) checking for a free entry in this map. If found, the offset within this swap map is returned. If not found, zero is returned.

```
unsigned long offset;
/*
 * We try to cluster swap pages by allocating them
 * sequentially in swap.  Once we've allocated
 * SWAPFILE_CLUSTER pages this way, however, we resort to
 * first-free allocation, starting a new cluster.  This
 * prevents us from scattering swap pages all over the entire
 * swap partition, so that we reduce overall disk seek times
 * between swap pages.  -- sct */
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;
        }
}
```

First checks if there are any active clustering try going on. Always try to allocate pages sequentially in the swap, like explained in the comment above.

Returns the available offset above `si->cluster_next` (and below `si->highest_bit`), if any.

```
si->cluster_nr = SWAPFILE_CLUSTER;
```

Either no swap cluster has ever been started or the current one has reached the defined number of pages, so starts (another) swap cluster by setting the number of pages in the cluster (`SWAPFILE_CLUSTER`).

```
/* try to find an empty (even not aligned) cluster. */
offset = si->lowest_bit;
check_next_cluster:
if (offset+SWAPFILE_CLUSTER-1 <= si->highest_bit)
{
        int nr;
        for (nr = offset; nr < offset+SWAPFILE_CLUSTER; nr++)
                if (si->swap_map[nr])
                {
                        offset = nr+1;
                        goto check_next_cluster;
                }
        /* We found a completly empty cluster, so start
         * using it.
         */
        goto got_page;
}
```

In the above "if" block, tries to find a completely empty cluster, even if it is not aligned with the previous one. If a completely free cluster is found, returns the first offset within it. Otherwise, tries to scan the map for an empty entry (see below).

```
/* No luck, so now go finegrined as usual. -Andrea */
for (offset = si->lowest_bit; offset <= si->highest_bit ;
```

```
     offset++) {
        if (si->swap_map[offset])
                continue;
```

There are neither no active cluster, nor an empty cluster, so performs the search checking every entry between the lowest (`si->lowest_bit`) and highest bit (`si->highest_bit`), returning the first that is actually unused.

```
        si->lowest_bit = offset+1;
got_page:
        if (offset == si->lowest_bit)
                si->lowest_bit++;
        if (offset == si->highest_bit)
                si->highest_bit--;
        if (si->lowest_bit > si->highest_bit) {
                si->lowest_bit = si->max;
                si->highest_bit = 0;
        }
```

Once an unused entry has been found, updates the lowest and highest bits, i.e, the lowest unused entry and the highest unused entry.

```
        si->swap_map[offset] = 1;
        nr_swap_pages--;
        si->cluster_next = offset+1;
        return offset;
}
```

Also sets the swap map counter to one, turning this entry into an used one and updates the number of reserved swap pages (`nr_swap_pages`). The `cluster_next` is set in order to try to cluster the next requested swap pages (see the first "if" block of this function).

```
si->lowest_bit = si->max;
si->highest_bit = 0;
return 0;
```

No free entry has been found, thus updates the lowest and highest bits to avoid unnecessary searches in the future, and return zero.

8.5.2 Function get_swap_page()

File: mm/swapfile.c
Prototype:

```
swp_entry_t get_swap_page(void)
```

This function checks swap types (ie, devices and partitions) for a free entry. It returns a `swp_entry_t` type which will have the zero value if no entry could be found, or a non-zero value which will be the swap address.

```
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;
```

Picks the next swap type to be checked. When its value is below zero, returns since there is no active swap.

```
if (nr_swap_pages <= 0)
        goto out;
```

If the counter of available swap pages shows that there are no available swap pages on any swap types (ie, devices and partitions).

```
while (1) {
        p = &swap_info[type];
        if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
```

Now checks if this swap type is writeable and only then keep searching. Otherwise, tries the next swap type. A swap type present on the swap list will not be writeable only if it is in the middle of a swapon or swapoff process.

```
                swap_device_lock(p);
                offset = scan_swap_map(p);
                swap_device_unlock(p);
```

Locks the device and scans its swap map for an unused offset. Then unlocks the device.

```
                if (offset) {
                        entry = SWP_ENTRY(type,offset);
                        type = swap_info[type].next;
                        if (type < 0 ||
                                p->prio != swap_info[type].prio) {
                                        swap_list.next = swap_list.head;
                        } else {
                                swap_list.next = type;
                        }
                        goto out;
                }
```

A free entry has been found, so set which swap type will be looked up in the next call and leave, returning the offset found. That is done since it is desirable to distribute all the swap addresses equally among all swap types.

```
        }
        type = p->next;
        if (!wrapped) {
                if (type < 0 || p->prio != swap_info[type].prio) {
                        type = swap_list.head;
                        wrapped = 1;
                }
        } else
                if (type < 0)
                        goto out;       /* out of swap space */
}
```

No offset has been found in the chosen swap type or it is not writeable, so try the next swap type. The "if" block above simply makes sure the `swap_list` isn't checked twice. When the whole list is checked once and nothing was found, it returns a zeroed swap entry.

```
out:
swap_list_unlock();
return entry;
```

8.5.3 Function swap_info_get()

File: mm/swapfile.c
Prototype:

```
static struct swap_info_struct * swap_info_get(swp_entry_t entry)
```

This function verifies if that is a valid entry, i.e, if it is set to a valid device, valid offset, and locks the swap list and the device where this entry is from. The return value is a pointer to the `swap_info_struct` type from the swap type (got from `entry` parameter. It will be NULL if some assertion failed or non-NULL otherwise

```
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;
```

Checks if it is a non-NULL entry and if a valid swap type number.

```
p = & swap_info[type];
if (!(p->flags & SWP_USED))
        goto bad_device;
```

Checks if that is an active swap type.

```
offset = SWP_OFFSET(entry);
if (offset >= p->max)
        goto bad_offset;
```

Ensures that the offset number is valid.

```
if (!p->swap_map[offset])
        goto bad_free;
```

Makes sure that it being used.

```
swap_list_lock();
if (p->prio > swap_info[swap_list.next].prio)
        swap_list.next = type;
swap_device_lock(p);
```

Once the swap type and swap offset are valid, locks the swap list and also the device. In the case the priority of this swap type is higher than the next swap type to be looked up, updates the swap list with this very type.

```
return p;
```

Returns the pointer to the `swap_info_struct` of this swap type.

```
bad_free:
printk(KERN_ERR "swap_free: %s%08lx\n", Unused_offset,
       entry.val);
goto out;

bad_offset:
printk(KERN_ERR "swap_free: %s%08lx\n", Bad_offset,
       entry.val);
goto out;

bad_device:
printk(KERN_ERR "swap_free: %s%08lx\n", Unused_file,
       entry.val);
goto out;

bad_nofile:
printk(KERN_ERR "swap_free: %s%08lx\n", Bad_file,
       entry.val);
out:
return NULL;
```

8.5.4 Function swap_info_put()

File: mm/swapfile.c
Prototype:

```
static void swap_info_put(struct swap_info_struct * p)
```

The role of this function is the opposite of `swap_info_get()`: it unlocks the swap device and the swap list.

```
swap_device_unlock(p);
swap_list_unlock();
```

8.5.5 Function swap_entry_free()

File: mm/swapfile.c
Prototype:

```
static int swap_entry_free(struct swap_info_struct *p,
                           unsigned long offset)
```

This function decreases the swap map counter, freeing the swap entry if the counter gets down to zero.

```
int count = p->swap_map[offset];

if (count < SWAP_MAP_MAX) {
```

As soon as the swap map count gets to the SWAP_MAP_MAX value, it will not get incremented nor decremented any longer. The reason behind this is that incrementing will overflow the space reserved for it. On the other hand, since it is unable to keep incrementing, decrementing cannot be done either because the counter is not going to be accurate any longer. Therefore, it will be only reclaimed in the swapoff process, so simply returns if that is the case.

```
        count--;
        p->swap_map[offset] = count;
```

In the case it has a counter that has not reached the SWAP_MAP_MAX value, decrements the value and set it to the swap map.

```
        if (!count) {
                if (offset < p->lowest_bit)
                        p->lowest_bit = offset;
                if (offset > p->highest_bit)
                        p->highest_bit = offset;
                nr_swap_pages++;
        }
}
return count;
```

If that was the last reference, updates the lowest and the highest free offsets (lowest_bit and highest_bit), if necessary, and increments the variable that accounts the number of available swap pages.

8.5.6 Function swap_free()

File: mm/swapfile.c
Prototype:

```
void swap_free(swp_entry_t entry)
```

This function is very simple: it locks the swap list and the swap device of the `entry` parameter, calls the `swap_entry_free()` function, and unlocks the list and the device.

```
struct swap_info_struct * p;

p = swap_info_get(entry);
if (p) {
        swap_entry_free(p, SWP_OFFSET(entry));
        swap_info_put(p);
}
```

8.5.7 Function swap_duplicate()

File: mm/swapfile.c
Prototype:

```
int swap_duplicate(swp_entry_t entry)
```

Given an entry, `swap_duplicate()` checks if it is a valid entry, increasing its reference counter in this case. It returns an **int** value which will be **one** (if sucess) or **zero** (if failure).

```
struct swap_info_struct * p;
unsigned long offset, type;
int result = 0;

type = SWP_TYPE(entry);
if (type >= nr_swapfiles)
        goto bad_file;
```

Is that entry set to a valid swap type? If it isn't, prints a warning message (see `bad_file` block below) and returns zero.

```
p = type + swap_info;
offset = SWP_OFFSET(entry);

swap_device_lock(p);
if (offset < p->max && p->swap_map[offset]) {
```

After locking the swap device, ensures that this entry is set a valid offset and that this entry is used. If any of these conditions are false, returns zero.

```
        if (p->swap_map[offset] < SWAP_MAP_MAX - 1) {
                p->swap_map[offset]++;
                result = 1;
```

If the reference counter for this entry will not reach the `SWAP_MAP_MAX` value when increased, simply increase it and set `result` to one.

```
        } else if (p->swap_map[offset] <= SWAP_MAP_MAX) {
                if (swap_overflow++ < 5)
                        printk(KERN_WARNING "swap_dup: swap
                                entry overflow\n");
                p->swap_map[offset] = SWAP_MAP_MAX;
                result = 1;
        }
}
```

For entries that will reach or have already reached the `SWAP_MAP_MAX` value, just set the counter to `SWAP_MAP_MAX` and assign one to `result`.

```
swap_device_unlock(p);
out:
return result;
```

Unlocks the device and returns the `result`.

```
bad_file:
printk(KERN_ERR "swap_dup: %s%08lx\n", Bad_file, entry.val);
goto out;
```

8.5.8 Function swap_count()

File: mm/swapfile.c
Prototype:

```
int swap_count(struct page *page)
```

Unused function, `swap_count()` returns the reference counter of the swap entry, if valid and used, to which a page (`page` parameter) is set.

```
struct swap_info_struct * p;
unsigned long offset, type;
swp_entry_t entry;
int retval = 0;

entry.val = page->index;
if (!entry.val)
        goto bad_entry;
```

Null entry, so prints a warning message and returns.

```
type = SWP_TYPE(entry);
if (type >= nr_swapfiles)
        goto bad_file;
```

Entry set to a invalid swap type, then prints a warning message and returns.

```
p = type + swap_info;
offset = SWP_OFFSET(entry);
if (offset >= p->max)
        goto bad_offset;
```

The offset of this entry is invalid, prints a warning message and returns.

```
if (!p->swap_map[offset])
        goto bad_unused;
```

Unused entry? Prints a warning message and returns.

```
retval = p->swap_map[offset];
out:
return retval;
```

Valid swap type and a valid and used offset? Returns its counter from `swap_map`.

```
bad_entry:
printk(KERN_ERR "swap_count: null entry!\n");
goto out;
bad_file:
printk(KERN_ERR "swap_count: %s%08lx\n", Bad_file, entry.val);
```

```
goto out;
bad_offset:
printk(KERN_ERR "swap_count: %s%08lx\n", Bad_offset, entry.val);
goto out;
bad_unused:
printk(KERN_ERR "swap_count: %s%08lx\n", Unused_offset, entry.val);
goto out;
```

8.6 Unusing Swap Entries

Analogous to the **Unmapping Pages from Processes** section, unusing checks the page tables from processes on the system. However, instead of unmapping the page table entries, it checks if it is set to a swap address located on the swap area (partition or swap file) being deactivated. In this case, it is remapped to a memory page which holds the same data.

8.6.1 Function unuse_pte()

File: mm/swapfile.c
Prototype:

```
static inline void unuse_pte(struct vm_area_struct * vma,
        unsigned long address, pte_t *dir,
        swp_entry_t entry, struct page* page)
```

This function checks if a page table entry (`dir` parameter) is set to the swap entry it is being unused (`entry` parameter), setting this pte to the memory page that holds the very same data as stored on the swap.

```
pte_t pte = *dir;

if (likely(pte_to_swp_entry(pte).val != entry.val))
        return;
```

Returns if the page table entry is set to any value different from the swap entry which is being unused.

```
if (unlikely(pte_none(pte) || pte_present(pte)))
        return;
```

Seems redundant, but checks if the page table entry is NULL (`pte_none()`) or have an address which has the present bit on (i.e, not swap entries). That is needed because `pte_to_swp_entry()` is architecture dependant and may change some lower bits, turning out that the first condition end up being true for a page table entry not actually set to a swap entry.

```
get_page(page);
set_pte(dir, pte_mkold(mk_pte(page, vma->vm_page_prot)));
swap_free(entry);
++vma->vm_mm->rss;
```

The page table entry is set to the swap entry which is being unused, so increments the reference count to this swap cache page (`get_page()`, since this process will point to this page; sets the pte to the page address, with the protections of the vm area it belongs to; and decrements the swap counter (`swap_free()`).

8.6.2 Function unuse_pmd()

File: mm/swapfile.c
Prototype:

```
static inline void unuse_pmd(struct vm_area_struct * vma,
        pmd_t *dir, unsigned long address,
        unsigned long size, unsigned long offset,
        swp_entry_t entry, struct page* page)
```

This function checks every page table entry from a page middle directory (`dir` parameter), trying to unuse them. Those page table entries set to the swap entry which is being unused (`entry` parameter) will be set to the memory page (`page` parameter) which holds the data stored on swap, thus unusing the swap address.

```
pte_t * pte;
unsigned long end;

if (pmd_none(*dir))
        return;
```

Does this page middle directory offset point to no page table? There is nothing to do, so just returns.

```
if (pmd_bad(*dir)) {
        pmd_ERROR(*dir);
        pmd_clear(dir);
        return;
}
```

Checks if the contents of this memory address points to a valid page table. If it does not, prints an error message (`pmd_ERROR`), clear this entry and returns.

```
pte = pte_offset(dir, address);
offset += address & PMD_MASK;
address &= ~PMD_MASK;
end = address + size;
if (end > PMD_SIZE)
        end = PMD_SIZE;
do {
        unuse_pte(vma, offset+address-vma->vm_start, pte, entry,
                  page);
        address += PAGE_SIZE;
        pte++;
} while (address && (address < end));
```

For every page table entry, until it reaches the end of this page middle directory or the end of the vm area (`end` stores the minimum between them), calls `unuse_pte()` which will check the page table entry data and will unuse it if it's the case.

8.6.3 Function unuse_pgd()

File: mm/swapfile.c
Prototype:

```
static inline void unuse_pgd(struct vm_area_struct * vma,
        pgd_t *dir, unsigned long address, unsigned long size,
        swp_entry_t entry, struct page* page)
```

This function checks every page middle directory from a page global directory (`dir` parameter) within the vm area (`vma`, `address` and `size`), trying to unuse all the page table entries from them. Those page table entries set to the swap entry which is being unused (`entry` parameter) will be set to the memory page (`page` parameter) which holds the data stored on swap, thus unusing the swap address.

```
pmd_t * pmd;
unsigned long offset, end;

if (pgd_none(*dir))
        return;
```

If this page global directory does not have a page middle directory, returns.

```
if (pgd_bad(*dir)) {
        pgd_ERROR(*dir);
        pgd_clear(dir);
        return;
}
```

Checks if the entry points to a bad page table. In this case, prints that there is an error (`pgd_ERROR()`) and clears this entry.

```
pmd = pmd_offset(dir, address);
offset = address & PGDIR_MASK;
address &= ~PGDIR_MASK;
end = address + size;
if (end > PGDIR_SIZE)
        end = PGDIR_SIZE;
if (address >= end)
        BUG();
do {
        unuse_pmd(vma, pmd, address, end - address, offset,
                  entry, page);
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
} while (address && (address < end));
```

For every page middle directory, until it reaches the end of this page global directory or the end of the vm area, calls `unuse_pmd()` which will check the page middle directory and unuse all the necessary page table entries.

8.6.4 Function unuse_vma()

File: mm/swapfile.c
Prototype:

```
static void unuse_vma(struct vm_area_struct * vma,
                      pgd_t *pgdir, swp_entry_t entry,
                      struct page* page)
```

This function checks every page global directory from a vm area (`vma` parameter), trying to unuse all the page table entries from them. Those page table entries set to the swap entry which is being unused (`entry` parameter) will be set to the memory page (`page` parameter) which holds the data stored on swap, thus unusing the swap address.

```
unsigned long start = vma->vm_start, end = vma->vm_end;

if (start >= end)
        BUG();
do {
        unuse_pgd(vma, pgdir, start, end - start, entry, page);
        start = (start + PGDIR_SIZE) & PGDIR_MASK;
        pgdir++;
} while (start && (start < end));
```

The first page global directory is already passed as parameter (`pgdir` parameter). So, for every page middle directory, until it reaches the end of this vm area, call `unuse_pgd()` which will check the page middle directory and unuse all of its page table entries it if it's the case.

8.6.5 Function unuse_process()

File: mm/swapfile.c
Prototype:

```
static void unuse_process(struct mm_struct * mm,
                          swp_entry_t entry, struct page* page)
```

This function checks every vm area from a process (actually, its mm struct, ie the `mm` parameter), trying to unuse all the page table entries from them. Those page table entries set to the swap entry which is being unused (`entry` parameter) will be set to the memory page (`page` parameter) which holds the data stored on swap, thus unusing the swap address.

```
struct vm_area_struct* vma;

/*
```

```
 * Go through process' page directory.
 */
spin_lock(&mm->page_table_lock);
for (vma = mm->mmap; vma; vma = vma->vm_next) {
        pgd_t * pgd = pgd_offset(mm, vma->vm_start);
        unuse_vma(vma, pgd, entry, page);
}
spin_unlock(&mm->page_table_lock);
return;
```

From the mm struct list of vm areas, tries to unuse every vm area of this process calling `unuse_vma()` which will check the vm area and unuses all of its page table entries it if it's the case.

8.6.6 Function find_next_to_unuse()

File: mm/swapfile.c
Prototype:

```
static int find_next_to_unuse(struct swap_info_struct *si,
                              int prev)
```

The `find_next_to_unuse()` function checks for a swap map entry which is used in order to be unused by the `try_to_unuse()` function.

```
int max = si->max;
int i = prev;
int count;

/*
 * No need for swap_device_lock(si) here: we're just looking
 * for whether an entry is in use, not modifying it; false
 * hits are okay, and sys_swapoff() has already prevented new
 * allocations from this area (while holding swap_list_lock()).
 */
for (;;) {
        if (++i >= max) {
                if (!prev) {
                        i = 0;
                        break;
                }
                /*
```

```
                 * No entries in use at top of swap_map,
                 * loop back to start and recheck there.
                 */
                max = prev + 1;
                prev = 0;
                i = 1;
        }
```

If the next value to be checked is greater than the maximum value of this swap type, restart the check, but set the variables (`prev`) to make sure it won't restart more than once (ie, if all the swap entries are unused).

```
        count = si->swap_map[i];
        if (count && count != SWAP_MAP_BAD)
                break;
}
```

Returns its offset if this swap map entry is used.

```
return i;
```

8.6.7 Function try_to_unuse()

File: mm/swapfile.c
Prototype:

```
static int try_to_unuse(unsigned int type)
```

Given a swap type (`type`), tries to unuse all the used swap entries by checking all the page table entries from all processes until all the swap entries have been sucessfully unused.

```
struct swap_info_struct * si = &swap_info[type];
struct mm_struct *start_mm;
unsigned short *swap_map;
unsigned short swcount;
struct page *page;
swp_entry_t entry;
int i = 0;
int retval = 0;
int reset_overflow = 0;
```

```
/*
 * When searching mms for an entry, a good strategy is to
 * start at the first mm we freed the previous entry from
 * (though actually we don't notice whether we or coincidence
 * freed the entry).  Initialize this start_mm with a hold.
 *
 * A simpler strategy would be to start at the last mm we
 * freed the previous entry from; but that would take less
 * advantage of mmlist ordering (now preserved by swap_out()),
 * which clusters forked address spaces together, most recent
 * child immediately after parent.  If we race with dup_mmap(),
 * we very much want to resolve parent before child, otherwise
 * we may miss some entries: using last mm would invert that.
 */
start_mm = &init_mm;
atomic_inc(&init_mm.mm_users);

/*
 * Keep on scanning until all entries have gone.  Usually,
 * one pass through swap_map is enough, but not necessarily:
 * mmput() removes mm from mmlist before exit_mmap() and its
 * zap_page_range().  That's not too bad, those entries are
 * on their way out, and handled faster there than here.
 * do_munmap() behaves similarly, taking the range out of mm's
 * vma list before zap_page_range().  But unfortunately, when
 * unmapping a part of a vma, it takes the whole out first,
 * then reinserts what's left after (might even reschedule if
 * open() method called) - so swap entries may be invisible
 * to swapoff for a while, then reappear - but that is rare.
 */
while ((i = find_next_to_unuse(si, i))) {
```

Tries to unuse every used swap entry of this swap type.

```
        /*
         * Get a page for the entry, using the existing swap
         * cache page if there is one.  Otherwise, get a clean
         * page and read the swap into it.
         */
        swap_map = &si->swap_map[i];
        entry = SWP_ENTRY(type, i);
        page = read_swap_cache_async(entry);
```

Comments are very clear. Note that all these actions are done in `read_swap_cache_async()` function. This page is read because in order to unuse the swap address, since all the page table entries set to this swap entry will be remapped to this page.

```
if (!page) {
        /*
         * Either swap_duplicate() failed because entry
         * has been freed independently, and will not be
         * reused since sys_swapoff() already disabled
         * allocation from here, or alloc_page() failed.
         */
        if (!*swap_map)
                continue;
```

Checks if the counter for this entry is zero. In this case, it has been freed concurrently with `read_swap_cache_async()` call.

```
        retval = -ENOMEM;
        break;
}
```

The swap entry is still used, so problems allocating a new page in `read_swap_cache_async()`. Gives up and returns `-ENOMEM`.

```
/*
 * Don't hold on to start_mm if it looks like exiting.
 */
if (atomic_read(&start_mm->mm_users) == 1) {
        mmput(start_mm);
        start_mm = &init_mm;
        atomic_inc(&init_mm.mm_users);
}
```

The process to which `start_mm` belongs is exiting (this function holds the last reference), therefore it is not worth looking up all its page tables here since they will be looked up in the exit code path.

```
/*
 * Wait for and lock page.  When do_swap_page races with
 * try_to_unuse, do_swap_page can handle the fault much
 * faster than try_to_unuse can locate the entry.  This
```

```
         * apparently redundant "wait_on_page" lets try_to_unuse
         * defer to do_swap_page in such a case - in some tests,
         * do_swap_page and try_to_unuse repeatedly compete.
         */
        wait_on_page(page);
        lock_page(page);
```

Locks the page, even if it has to sleep for that. Check comments above about the race between `do_swap_page()` and `try_to_unuse()`.

```
        /*
         * Remove all references to entry, without blocking.
         * Whenever we reach init_mm, there's no address space
         * to search, but use it as a reminder to search shmem.
         */
        swcount = *swap_map;
        if (swcount > 1) {
                flush_page_to_ram(page);
                if (start_mm == &init_mm)
                        shmem_unuse(entry, page);
                else
                        unuse_process(start_mm, entry, page);
        }
```

If the swap entry is still used, calls `unuse_process()` to search all its addressing space, trying to unuse its page table entries which are set to this swap entry. In the case `start_mm` is set to `init_mm` (see comment above), it checks the `shmem_unuse()` since `init_mm` does not have any address space to be searched.

```
        if (*swap_map > 1) {
                int set_start_mm = (*swap_map >= swcount);
                struct list_head *p = &start_mm->mmlist;
                struct mm_struct *new_start_mm = start_mm;
                struct mm_struct *mm;

                spin_lock(&mmlist_lock);
                while (*swap_map > 1 &&
                                (p = p->next) != &start_mm->mmlist) {
                        mm = list_entry(p, struct mm_struct, mmlist);
                        swcount = *swap_map;
```

```
                if (mm == &init_mm) {
                        set_start_mm = 1;
                        shmem_unuse(entry, page);
                } else
                        unuse_process(mm, entry, page);
                if (set_start_mm && *swap_map < swcount) {
                        new_start_mm = mm;
                        set_start_mm = 0;
                }
        }
        atomic_inc(&new_start_mm->mm_users);
        spin_unlock(&mmlist_lock);
        mmput(start_mm);
        start_mm = new_start_mm;
}
```

The swap entry is still used after checking the `start_mm`, so unuses each active process on the system while the swap entry is still active.

After unusing the `start_mm`, if the swap entry is used by the same or greater number of users, changes the `start_mm` for the first mm struct which effectively helped decreasing the number of users.

```
/*
 * How could swap count reach 0x7fff when the maximum
 * pid is 0x7fff, and there's no way to repeat a swap
 * page within an mm (except in shmem, where it's the
 * shared object which takes the reference count)?
 * We believe SWAP_MAP_MAX cannot occur in Linux 2.4.
 *
 * If that's wrong, then we should worry more about
 * exit_mmap() and do_munmap() cases described above:
 * we might be resetting SWAP_MAP_MAX too early here.
 * We know "Undead"s can happen, they're okay, so don't
 * report them; but do report if we reset SWAP_MAP_MAX.
 */
if (*swap_map == SWAP_MAP_MAX) {
        swap_list_lock();
        swap_device_lock(si);
        nr_swap_pages++;
        *swap_map = 1;
        swap_device_unlock(si);
```

```
                swap_list_unlock();
                reset_overflow = 1;
        }
```

Now handle the cases where the swap map counter has reached the SWAP_MAP_MAX counter. It seems it cannot occur in Linux 2.4, so it is more a sanity check since the author(s) is(are) not sure that cannot actually happen.

```
        /*
         * If a reference remains (rare), we would like to leave
         * the page in the swap cache; but try_to_swap_out could
         * then re-duplicate the entry once we drop page lock,
         * so we might loop indefinitely; also, that page could
         * not be swapped out to other storage meanwhile.  So:
         * delete from cache even if there's another reference,
         * after ensuring that the data has been saved to disk -
         * since if the reference remains (rarer), it will be
         * read from disk into another page.  Splitting into two
         * pages would be incorrect if swap supported "shared
         * private" pages, but they are handled by tmpfs files.
         * Note shmem_unuse already deleted its from swap cache.
         */
        if ((*swap_map > 1) && PageDirty(page) && PageSwapCache(page)) {
                rw_swap_page(WRITE, page);
                lock_page(page);
        }
        if (PageSwapCache(page))
                delete_from_swap_cache(page);

        /*
         * So we could skip searching mms once swap count went
         * to 1, we did not mark any present ptes as dirty: must
         * mark page dirty so try_to_swap_out will preserve it.
         */
        SetPageDirty(page);
        UnlockPage(page);
        page_cache_release(page);

        /*
         * Make sure that we aren't completely killing
         * interactive performance.  Interruptible check on
```

```
         * signal_pending() would be nice, but changes the spec?
         */
        if (current->need_resched)
                schedule();
}

mmput(start_mm);
if (reset_overflow) {
        printk(KERN_WARNING
               "swapoff: cleared swap entry overflow\n");
        swap_overflow = 0;
}
return retval;
```

8.7 Exclusive Swap Pages

8.7.1 Function exclusive_swap_page()

File: mm/swapfile.c
Prototype:

```
static int exclusive_swap_page(struct page *page)
```

This function checks if the swap address to which the page parameter is set has only one reference (ie, checks the swap map counter). It returns an **int** value, which will be **one** if the previous condition is true and **zero** otherwise.

```
int retval = 0;
struct swap_info_struct * p;
swp_entry_t entry;

entry.val = page->index;
p = swap_info_get(entry);
```

Locks the swap list and the swap device of the entry the page is set to.

```
if (p) {
        /* Is the only swap cache user the cache itself? */
        if (p->swap_map[SWP_OFFSET(entry)] == 1) {
```

As the comment says, checks if this page is the only user of the swap entry. It is said "swap cache" above because a reference to this swap entry was got when the page was added to the swap cache.

```
/* Recheck the page count with the
   pagecache lock held.. */
spin_lock(&pagecache_lock);
if (page_count(page) -
           !!page->buffers == 2)
        retval = 1;
spin_unlock(&pagecache_lock);
```

This function is called by `can_share_swap_page()`, which checks the page count without the pagecache lock, so it's better recheck it with this lock held. If the page count is still the one expected, return one.

```
        }
        swap_info_put(p);
```

Unlock the swap list and the swap device.

```
}
return retval;
```

8.7.2 Function can_share_swap_page()

File: mm/swapfile.c
Prototype:

```
int can_share_swap_page(struct page *page)
```

This function returns an **int** value that means if this page can be shared (return value: **one**) or not (return value: **zero**). Here "to be shared" means that this page nor its swap entry are mapped by other process.

```
int retval = 0;

if (!PageLocked(page))
        BUG();
switch (page_count(page)) {
```

Starts checking if it can be shared looking into its page counter.

```
case 3:
        if (!page->buffers)
                break;
        /* Fallthrough */
```

A page with its counter set to 3 either is mapped by a process or have buffers. If it doesn't have buffers, it is surely mapped by a process, so returns zero.

```
case 2:
        if (!PageSwapCache(page))
                break;
```

Checks if the page is in the swap cache. Mapped pages that are not in the swap cache must return zero.

```
        retval = exclusive_swap_page(page);
```

For cases where that is a swap cache page with counter 3 and buffers or counter 2, check if that is an exclusive swap page by calling `exclusive_swap_page()`. That function will check again the page counter and also the swap count for the swap entry this page is set to. In the case no process has mapped this page in the meanwhile and also the swap entry is exclusive. The return value of this function will the value returned by `exclusive_swap_page()` function.

```
        break;
case 1:
        if (PageReserved(page))
                break;
        retval = 1;
}
return retval;
```

Pages with counter 1 can be shared.

8.7.3 Function remove_exclusive_swap_page()

File: mm/swapfile.c
Prototype:

```
int remove_exclusive_swap_page(struct page *page)
```

This function performs the same task as `exclusive_swap_page()`, checking if the page nor its swap entry do not have users (includes being mapped by processes), but also removes the page from swap cache if it is exclusive. It returns an **int** value, which is **one** if the page was removed and **zero** otherwise.

```
int retval;
struct swap_info_struct * p;
swp_entry_t entry;

if (!PageLocked(page))
        BUG();
```

The page is supposed to be locked, so BUG() if it is unlocked.

```
if (!PageSwapCache(page))
        return 0;
```

Since the page might have been removed from swap cache before the caller got the lock on it, checks if the page is still in the swap cache. If it is not, returns zero (it was not removed).

```
/* 2: us + cache */
if (page_count(page) - !!page->buffers != 2)
        return 0;
```

Pages that have more than 2 users (plus eventual buffers) have users, so they are not exclusive and cannot be remove. In this case, returns zero.

The page count is checked without the `pagecache_lock` held, but will be rechecked below with this lock held.

```
entry.val = page->index;
p = swap_info_get(entry);
if (!p)
        return 0;
```

Locks the swap list and swap device of the swap entry this page is set to. If that is an invalid entry (`swap_info_get()` returns NULL in this case), returns zero.

```
/* Is the only swap cache user the cache itself? */
retval = 0;
if (p->swap_map[SWP_OFFSET(entry)] == 1) {
```

Verifies the number of users of this swap entry. If more than one, it is not exclusive, so unlocks the swap device, the lists and returns zero (see below).

```
        /* Recheck the page count with the pagecache
           lock held.. */
        spin_lock(&pagecache_lock);
        if (page_count(page) - !!page->buffers == 2) {
                __delete_from_swap_cache(page);
                SetPageDirty(page);
                retval = 1;
        }
        spin_unlock(&pagecache_lock);
}
swap_info_put(p);
```

For swap entries which the swap cache page is the only user, rechecks the counter with `pagecache_lock` held. If the page is still unmapped by processes, deletes it from swap cache. Sets it dirty in order to be kept by the swap out code.

```
if (retval) {
        block_flushpage(page, 0);
        swap_free(entry);
        page_cache_release(page);
}

return retval;
```

For pages removed from swap cache, flushes them to the disk, drops their reference on the swap entry and drops the reference got by page cache when it was added to the swap cache.

8.7.4 Function free_swap_and_cache()

File: mm/swapfile.c
Prototype:

```
void free_swap_and_cache(swp_entry_t entry)
```

Given a swap entry, this function decrements its reference counter (calling `swap_entry_free()`). In the case the counter, after decreased, gets to one, checks if this last reference belongs to a swap cache page, trying to free it (only if it could be locked at once).

```
struct swap_info_struct * p;
struct page *page = NULL;

p = swap_info_get(entry);
```

Locks the swap list and the swap device of this entry, returning the pointer to the swap device info structure.

```
if (p) {
        if (swap_entry_free(p, SWP_OFFSET(entry)) == 1)
```

If a valid entry (`p != NULL`), decremens its reference counter calling `swap_entry_free()`.

```
                page = find_trylock_page(&swapper_space,
                        entry.val);
```

If, after decrementing, there is only one reference on this swap entry, checks if this reference is owned by a swap cache page, trying to lock it at once (i.e, without sleeping to lock).

```
        swap_info_put(p);
}
```

Unlocks the swap list and the swap device after freeing the swap entry.

```
if (page) {
        page_cache_get(page);
        /* Only cache user (+us), or swap space full? Free it! */
        if (page_count(page) - !!page->buffers == 2 ||
            vm_swap_full()) {
                delete_from_swap_cache(page);
                SetPageDirty(page);
        }
        UnlockPage(page);
        page_cache_release(page);
}
```

The swap cache page has been found and could be locked at once, so checks if the page does not have other users and frees it, removing it from swap cache. Also it can be removed from swap cache if the swap is full.

8.8 Swap Areas

8.8.1 Function sys_swapoff()

File: mm/swapfile.c
Prototype:

```
asmlinkage long sys_swapoff(const char * specialfile)
```

This function tries to disable swap files or partitions, i.e, it performs swapoff system call role. The return value is a long, which will return the error code. If zero, no error has ocurred and the swap file or partition has been succesfully disabled.

```
struct swap_info_struct * p = NULL;
unsigned short *swap_map;
struct nameidata nd;
int i, type, prev;
int err;

if (!capable(CAP_SYS_ADMIN))
        return -EPERM;
```

Checks capabilities of this process, before going on. If they do not allow performing this task, return `-EPERM` (not permitted) error.

```
err = user_path_walk(specialfile, &nd);
if (err)
        goto out;
```

Given the swap file or partition name (`specialfile` parameter), tries to get its namei information (`nameidata`). If not found, return the error `user_path_walk()` found.

```
lock_kernel();
prev = -1;
swap_list_lock();
for (type = swap_list.head; type >= 0;
     type = swap_info[type].next) {
        p = swap_info + type;
        if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
                if (p->swap_file == nd.dentry)
                        break;
```

```
        }
        prev = type;
}
err = -EINVAL;
if (type < 0) {
        swap_list_unlock();
        goto out_dput;
}
```

Locks the kernel, the swap list and searches the swap type which is set to this dentry (if any). If not found (`type < 0`), returns.

```
if (prev < 0) {
        swap_list.head = p->next;
} else {
        swap_info[prev].next = p->next;
}
if (type == swap_list.next) {
        /* just pick something that's safe... */
        swap_list.next = swap_list.head;
}
```

Fixes the swap list (previous entry and swap list head and next fields).

```
nr_swap_pages -= p->pages;
total_swap_pages -= p->pages;
p->flags = SWP_USED;
swap_list_unlock();
unlock_kernel();
```

Updates the control variables: number of free swap pages (`nr_swap_pages`) and total number of swap pages (`total_swap_pages`). Also changes the flag from the `SWP_WRITEOK` to `SWP_USED`, so this swap type cannot be used to assign new swap entries.

After these changes, unlocks the swap list and the kernel global lock.

```
err = try_to_unuse(type);
```

Calls `try_to_unuse()` which will try to unuse all the used swap entries from this swap type.

```
lock_kernel();
if (err) {
        /* re-insert swap space back into swap_list */
        swap_list_lock();
        for (prev = -1, i = swap_list.head; i >= 0;
             prev = i, i = swap_info[i].next)
                if (p->prio >= swap_info[i].prio)
                        break;
        p->next = i;
        if (prev < 0)
                swap_list.head = swap_list.next = p - swap_info;
        else
                swap_info[prev].next = p - swap_info;
        nr_swap_pages += p->pages;
        total_swap_pages += p->pages;
        p->flags = SWP_WRITEOK;
        swap_list_unlock();
        goto out_dput;
}
```

If `try_to_unuse()` couldn't unuse all the swap entries, undo all the previous changes and return.

```
if (p->swap_device)
        blkdev_put(p->swap_file->d_inode->i_bdev, BDEV_SWAP);
```

This swap type is totally unused, so drop the reference on the block device if that is a swap partition (not a swap file).

```
path_release(&nd);
```

Drops the reference on this dentry and vfsmnt got when this swap was activated in `sys_swapon()`.

```
swap_list_lock();
swap_device_lock(p);
nd.mnt = p->swap_vfsmnt;
nd.dentry = p->swap_file;
p->swap_vfsmnt = NULL;
p->swap_file = NULL;
p->swap_device = 0;
p->max = 0;
```

```
swap_map = p->swap_map;
p->swap_map = NULL;
p->flags = 0;
swap_device_unlock(p);
swap_list_unlock();
vfree(swap_map);
err = 0;
```

With the swap list and device properly locked, zeroes the swap type structure and free the `swap_map` table. Also set the return value (`err`) to zero.

```
out_dput:
unlock_kernel();
path_release(&nd);
out:
return err;
```

Unlocks the global kernel lock, drop the reference on the dentry and vfsmnt got when the pathname was looked up and returns.

8.8.2 Function get_swaparea_info()

File: mm/swapfile.c
Prototype:

```
int get_swaparea_info(char *buf)
```

This function is used by the proc entry (`/proc/swap`) to display information about the swap types. It returns an **int** value telling the lenght of the output string.

```
char * page = (char *) __get_free_page(GFP_KERNEL);
struct swap_info_struct *ptr = swap_info;
int i, j, len = 0, usedswap;

if (!page)
        return -ENOMEM;
```

Allocates a new page which will be used by `d_path` below. If the page cannot be allocated, returns `-ENOMEM` (out of memory) error.

```
len += sprintf(buf, "Filename\t\t\tType\t\tSize\tUsed\tPriority\n");
```

Prints the header.

```
for (i = 0 ; i < nr_swapfiles ; i++, ptr++) {
```

For every swap type.
The `ptr` variable is initialized with the first swap type.

```
     if ((ptr->flags & SWP_USED) && ptr->swap_map) {
```

Only swap types which are used (even if they are being unused in the swapoff process) will be displayed. Also make sure that `swap_map` is non-null to avoid displaying swap types that are being "swapped on" in `sys_swapon`.

```
char * path = d_path(ptr->swap_file, ptr->swap_vfsmnt,
                            page, PAGE_SIZE);
```

The `d_path` function will write the path name (it can just a device name) into the page. It will return to the `path` variable the address of the path name start.

```
len += sprintf(buf + len, "%-31s ", path);

if (!ptr->swap_device)
        len += sprintf(buf + len, "file\t\t");
else
        len += sprintf(buf + len, "partition\t")
```

Prints the path name and whether it is a file or partition.

```
usedswap = 0;
for (j = 0; j < ptr->max; ++j)
        switch (ptr->swap_map[j]) {
                case SWAP_MAP_BAD:
                case 0:
                        continue;
                default:
                        usedswap++;
        }
```

Accounts the number of swap entries from this type that are used. Since the swap device isn't held, it is not accurate.

```
                len += sprintf(buf + len, "%d\t%d\t%d\n", ptr->pages << (PAGE_SH
                        usedswap << (PAGE_SHIFT - 10), ptr->prio);
        }
}
```

Prints information like the total number of pages available on this swap type, the number of used pages (computed above) and the swap priority.

```
free_page((unsigned long) page);
return len;
```

Frees the page used as buffer and returns the length of the string printed into the buffer (which will be displayed by the procfs).

8.8.3 Function is_swap_partition()

File: mm/swapfile.c
Prototype:

```
int is_swap_partition(kdev_t dev)
```

Given a device, checks if it is a swap partition. It returns an **int** value (**one** if partition, **zero** otherwise).

```
struct swap_info_struct *ptr = swap_info;
int i;

for (i = 0 ; i < nr_swapfiles ; i++, ptr++) {
        if (ptr->flags & SWP_USED)
                if (ptr->swap_device == dev)
                        return 1;
}
return 0;
```

Simply looks up every swap type, checking if it is used and if it is set to the device passed as parameter (only partition cases, since swap file cases will have a null `swap_device`).

8.8.4 Function sys_swapon()

File: mm/swapfile.c
Prototype:

```
asmlinkage long sys_swapon(const char * specialfile,
                           int swap_flags)
```

This function tries to activate swap files or partitions, ie it performs swapon system call role. The return value is a **long**, which will return the error code. If **zero**, no error has ocurred and the swap file or partition has been succesfully enabled.

```
struct swap_info_struct * p;
struct nameidata nd;
struct inode * swap_inode;
unsigned int type;
int i, j, prev;
int error;
static int least_priority = 0;
union swap_header *swap_header = 0;
int swap_header_version;
int nr_good_pages = 0;
unsigned long maxpages = 1;
int swapfilesize;
struct block_device *bdev = NULL;
unsigned short *swap_map;

if (!capable(CAP_SYS_ADMIN))
        return -EPERM;
```

Checks capabilities of this process, before going on. If they do not allow performing this task, returns `-EPERM` (not permitted) error.

```
lock_kernel();
swap_list_lock();
p = swap_info;
for (type = 0 ; type < nr_swapfiles ; type++,p++)
        if (!(p->flags & SWP_USED))
                break;
```

Looks for the first swap type in `swap_info` struct which is unused that can be used by this new swap type. This search is protected by the swap list lock.

```
error = -EPERM;
if (type >= MAX_SWAPFILES) {
        swap_list_unlock();
        goto out;
}
```

Returns `-EPERM` (operation not permitted) error if there is no swap type available,

```
if (type >= nr_swapfiles)
        nr_swapfiles = type+1;
```

Updates the variable that stores the last swap type used.

```
p->flags = SWP_USED;
p->swap_file = NULL;
p->swap_vfsmnt = NULL;
p->swap_device = 0;
p->swap_map = NULL;
p->lowest_bit = 0;
p->highest_bit = 0;
p->cluster_nr = 0;
p->sdev_lock = SPIN_LOCK_UNLOCKED;
p->next = -1;
```

Initializes the swap type.

```
if (swap_flags & SWAP_FLAG_PREFER) {
        p->prio =
          (swap_flags & SWAP_FLAG_PRIO_MASK)>>
          SWAP_FLAG_PRIO_SHIFT;
} else {
        p->prio = --least_priority;
}
```

The `swap_flags` parameter indicates if the priority parameter has been specified. If specified, sets the priority of this swap type to the value passed in the `swap_flags` parameter. If not specified, sets every new swapon with a lower priority.

```
swap_list_unlock();
error = user_path_walk(specialfile, &nd);
if (error)
        goto bad_swap_2;
```

Gets the inode information (actually, the nameidata) for this file. If it does not exist, back out all the previous changes and return the error `user_path_walk` found.

```
p->swap_file = nd.dentry;
p->swap_vfsmnt = nd.mnt;
swap_inode = nd.dentry->d_inode;
error = -EINVAL;
```

Sets the file and vfs mount point of this swap type.

```
if (S_ISBLK(swap_inode->i_mode)) {
```

This dentry inode is a block device.

```
        kdev_t dev = swap_inode->i_rdev;
        struct block_device_operations *bdops;
        devfs_handle_t de;

        p->swap_device = dev;
```

Stores the device into the `swap_device`.

```
        set_blocksize(dev, PAGE_SIZE);
```

Sets the device block size to `PAGE_SIZE`.

```
        bd_acquire(swap_inode);
```

Gets a reference on the block device, if it exists. Or acquire a new block device structure, setting the `swap_inode` to this block device.

```
        bdev = swap_inode->i_bdev;
        de = devfs_get_handle_from_inode(swap_inode);
        /*  Increments module use count  */
        bdops = devfs_get_ops(de);
        if (bdops) bdev->bd_op = bdops;
```

If using devfs, gets the handle, increments the modules usage counter and defines the block device operations.

```
        error = blkdev_get(bdev, FMODE_READ|FMODE_WRITE, 0,
                           BDEV_SWAP);
```

Opens the block device.

```
        /*Decrement module use count now we're safe*/
        devfs_put_ops(de);
```

For systems with devfs only, it decrements the usage counter of this module.

```
        if (error)
                goto bad_swap_2;
```

If the block device couldn't be opened, backs out the changes and returns `-EINVAL` (invalid argument) error.

```
        set_blocksize(dev, PAGE_SIZE);
        error = -ENODEV;
        if (!dev || (blk_size[MAJOR(dev)] &&
             !blk_size[MAJOR(dev)][MINOR(dev)]))
                goto bad_swap;
        swapfilesize = 0;
        if (blk_size[MAJOR(dev)])
                swapfilesize = blk_size[MAJOR(dev)][MINOR(dev)]
                        >> (PAGE_SHIFT - 10);
```

Checks if the device and block sizes are consistent, backing out the changes and returning `-EINVAL` (invalid argument) if they are not. Also computes the size of the swap into the `swapfilesize` variable if the size of the block device is defined.

```
} else if (S_ISREG(swap_inode->i_mode))
        swapfilesize = swap_inode->i_size >> PAGE_SHIFT;
```

The inode is a regular file, so simply sets the `swapfilesize` variable as the file size (`i_size`).

```
else
        goto bad_swap;
```

Nor a partition nor a regular file, so backs out the previous changes and returns `-EINVAL` (invalid argument) error.

```
error = -EBUSY;
for (i = 0 ; i < nr_swapfiles ; i++) {
        struct swap_info_struct *q = &swap_info[i];
        if (i == type || !q->swap_file)
                continue;
        if (swap_inode->i_mapping ==
            q->swap_file->d_inode->i_mapping)
                goto bad_swap;
}
```

Makes sure this device has not been activated by other swap type. If it has already been activated, backs out all the changes and return `-EBUSY` (device or resource busy) error.

```
swap_header = (void *) __get_free_page(GFP_USER);
if (!swap_header) {
        printk("Unable to start swapping: out of memory :-)\n");
        error = -ENOMEM;
        goto bad_swap;
}
```

Allocates a page that will hold the swap header (the first block of this swap type). If that page cannot be allocated, backs out the previous changes and returns `-ENOMEM` (out of memory) error.

```
lock_page(virt_to_page(swap_header));
rw_swap_page_nolock(READ, SWP_ENTRY(type,0), (char *) swap_header);
```

Reads the first block (block zero) of this swap type into the just allocated page.

```
if (!memcmp("SWAP-SPACE",swap_header->magic.magic,10))
        swap_header_version = 1;
else if (!memcmp("SWAPSPACE2",swap_header->magic.magic,10))
        swap_header_version = 2;
else {
        printk("Unable to find swap-space signature\n");
        error = -EINVAL;
        goto bad_swap;
}
```

Checks the swap version and sets the `swap_header_version` variable. If neither version 1 nor 2, backout the changes and return `-EINVAL` (invalid argument) error.

```
switch (swap_header_version) {
case 1:
        memset(((char *) swap_header)+PAGE_SIZE-10,0,10);
        j = 0;
        p->lowest_bit = 0;
        p->highest_bit = 0;
        for (i = 1 ; i < 8*PAGE_SIZE ; i++) {
                if (test_bit(i,(char *) swap_header)) {
                        if (!p->lowest_bit)
                                p->lowest_bit = i;
                        p->highest_bit = i;
                        maxpages = i+1;
                        j++;
                }
        }
```

In the version 1 of swap space, the bad blocks were set in the swap header. So, in order to initialize the `lowest` and `highest` bits, and the `swap_map` as well, those bits are tested. A bit one means that is a valid entry.

```
        nr_good_pages = j;
        p->swap_map = vmalloc(maxpages * sizeof(short));
        if (!p->swap_map) {
                error = -ENOMEM;
                goto bad_swap;
        }
```

Allocates the `swap_map`. The swap map is allocated using `vmalloc` because the map might not be able to be allocated using kmalloc since it may be huge. If the `swap_map` cannot be allocated, backs out the changes and returns `-ENOMEM` (out of memory) error.

```
        for (i = 1 ; i < maxpages ; i++) {
                if (test_bit(i,(char *) swap_header))
                        p->swap_map[i] = 0;
                else
                        p->swap_map[i] = SWAP_MAP_BAD;
        }
        break;
```

In the same way as the `highest` and `lowest` bits were set above, every entry in the swap map is initialized testing the bits of the swap header.

```
case 2:
        /* Check the swap header's sub-version and the size of
           the swap file and bad block lists */
        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;
        }
```

The kernel has support only for subversion 1 of swap header version 2. If any other subversion, backs out the changes and returns `-EINVAL` (invalid argument) error.

```
        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;
```

Sets the lowest (`lowest_bit` and highest `highest_bit` offsets for this swap type based on the `info.last_page`. Also sets an auxiliar variable that stores the maximum number of pages.

```
        error = -EINVAL;
        if (swap_header->info.nr_badpages > MAX_SWAP_BADPAGES)
                goto bad_swap;
```

The version 2 of swap space has a maximum number of bad pages. Then reads the number of bad pages from the header and checks if it is not greater than the maximum allowed number of bad pages (`MAX_SWAP_BADPAGES`).

```
        /* OK, set up the swap map and apply the bad block list */
        if (!(p->swap_map = vmalloc(maxpages * sizeof(short)))) {
                error = -ENOMEM;
                goto bad_swap;
        }
        error = 0;
        memset(p->swap_map, 0, maxpages * sizeof(short));
```

Allocates the `swap_map`. The swap map is allocated using `vmalloc` because the map might not be able to be allocated using kmalloc since it may be huge. Also zeroes all the `swap_map` using `memset`.

In the case the `swap_map` cannot be allocated, backs out the previous changes and returns `-ENOMEM` (out of memory) error.

```
        for (i=0; i<swap_header->info.nr_badpages; i++) {
                int page = swap_header->info.badpages[i];
                if (page <= 0 ||
                    page >= swap_header->info.last_page)
                        error = -EINVAL;
                else
                        p->swap_map[page] = SWAP_MAP_BAD;
        }
```

For every index in the `info.badpages` array from the swap header, sets that index as a bad block in the `swap_map`. If any of those index are invalid (not between 0 and `info.last_page`), backs out the changes (below) and return `-EINVAL` (invalid argument) error.

```
        nr_good_pages = swap_header->info.last_page -
                        swap_header->info.nr_badpages -
                        1 /* header page */;
        if (error)
                goto bad_swap;
}
```

And sets the number of good pages.

```
if (swapfilesize && maxpages > swapfilesize) {
        printk(KERN_WARNING
               "Swap area shorter than signature indicates\n");
        error = -EINVAL;
        goto bad_swap;
}
```

From now on it is independent on the swap version. That is a sanity check to know if the swap header is consistent with the swap file size.

```
if (!nr_good_pages) {
        printk(KERN_WARNING "Empty swap-file\n");
        error = -EINVAL;
        goto bad_swap;
}
```

Now checks if there are any good page. If there are no good pages, it is an empty swap type, thus backs out the previous changes and return `-EINVAL` (invalid argument) error.

```
p->swap_map[0] = SWAP_MAP_BAD;
```

The first block is the swap header, so mark it as a bad block.

```
swap_list_lock();
swap_device_lock(p);
p->max = maxpages;
p->flags = SWP_WRITEOK;
p->pages = nr_good_pages;
nr_swap_pages += nr_good_pages;
total_swap_pages += nr_good_pages;
```

With the swap list and swap device locked, finishes setting the swap info structure. Also sets some control variables, like the free number of swap pages (`nr_swap_pages` variable) and the total number of swap pages (`total_swap_pages`).

```
printk(KERN_INFO "Adding Swap: %dk swap-space (priority %d)\n",
       nr_good_pages\<\<(PAGE_SHIFT-10), p->prio);
```

Prints information about the new swap being added to the system.

```
/* insert swap space into swap_list: */
prev = -1;
for (i = swap_list.head; i >= 0; i = swap_info[i].next) {
        if (p->prio >= swap_info[i].prio) {
                break;
        }
        prev = i;
}
p->next = i;
if (prev < 0) {
        swap_list.head = swap_list.next = p - swap_info;
} else {
        swap_info[prev].next = p - swap_info;
}
swap_device_unlock(p);
swap_list_unlock();
```

Adds this swap type to the swap list, in priority ordering.

```
error = 0;
goto out;
bad_swap:
if (bdev)
        blkdev_put(bdev, BDEV_SWAP);
bad_swap_2:
swap_list_lock();
swap_map = p->swap_map;
nd.mnt = p->swap_vfsmnt;
nd.dentry = p->swap_file;
p->swap_device = 0;
p->swap_file = NULL;
p->swap_vfsmnt = NULL;
p->swap_map = NULL;
p->flags = 0;
if (!(swap_flags & SWAP_FLAG_PREFER))
        ++least_priority;
swap_list_unlock();
if (swap_map)
        vfree(swap_map);
path_release(&nd);
```

This is the block to back out if there is any error while trying to activate this swap space.

```
out:
if (swap_header)
        free_page((long) swap_header);
unlock_kernel();
return error;
```

Frees the page to be used as buffer for swap header and return.

8.8.5 Function si_swapinfo()

File: mm/swapfile.c
Prototype:

```
void si_swapinfo(struct sysinfo *val)
```

This function returns, in the `val` parameter, the number of free swap pages and also the number of total swap pages. It is used by some functions in the kernel, e.g. when displaying memory information in `/proc/meminfo`.

```
unsigned int i;
unsigned long nr_to_be_unused = 0;

swap_list_lock();
for (i = 0; i < nr_swapfiles; i++) {
        unsigned int j;
        if (swap_info[i].flags != SWP_USED)
                continue;
```

Only look up the `swap_map` of swap spaces that have `SWP_USED` as it only flags, since they are being deactivated in `sys_swapoff()` or activated in `sys_swapon()`. Active swap spaces have their pages already accounted in `nr_swap_pages` and `total_swap_pages`.

```
        for (j = 0; j < swap_info[i].max; ++j) {
                switch (swap_info[i].swap_map[j]) {
                        case 0:
                        case SWAP_MAP_BAD:
                                continue;
                        default:
                                nr_to_be_unused++;
                }
        }
}
```

For swap spaces that are being deactivated or activated, accounts the used entries into the `nr_to_be_unused` variable since they are not any longer accounted in `nr_swap_pages` and `total_swap_pages` variables.

```
val->freeswap = nr_swap_pages + nr_to_be_unused;
val->totalswap = total_swap_pages + nr_to_be_unused;
swap_list_unlock();
```

And adds the used entries in the swap devices that are being deactivated to the free number of swap pages and to the total number of swap pages. It assumes that those pages are going to be sucessfully unused.

8.8.6 Function get_swaphandle_info()

File: mm/swapfile.c
Prototype:

```
void get_swaphandle_info(swp_entry_t entry, unsigned long *offset,
                         kdev_t *dev, struct inode **swapf)
```

Used by `rw_swap_page_base()` IO function, `get_swaphandle_info()` returns info to perform IO on swap pages. It returns the offset (in the `offset` parameter) of a certain entry (`entry` parameter) and also checks if the entry is located on a swap device, returning the device (in `dev` parameter), or the swap file (in the `swapf` parameter) otherwise.

```
unsigned long type;
struct swap_info_struct *p;

type = SWP_TYPE(entry);
if (type >= nr_swapfiles) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n",
               Bad_file, entry.val);
        return;
}
```

Checks if it is a valid type, printing a warning message and returning if that is not the case.

```
p = &swap_info[type];
*offset = SWP_OFFSET(entry);
if (*offset >= p->max && *offset != 0) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n",
               Bad_offset, entry.val);
        return;
}
```

Now check if it is a valid offset. It does not allow the offset to be zero, since that is the offset of the swap header. Prints a warning message and returns if an invalid offset.

```
if (p->swap_map && !p->swap_map[*offset]) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n",
               Unused_offset, entry.val);
        return;
}
```

Sanity check to know if it is an used entry. Prints a warning message and returns when it is unused.

```
if (!(p->flags & SWP_USED)) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n",
               Unused_file, entry.val);
        return;
}
```

To conclude, checks if the swap type is used. Also prints a warning message and returns if that is the case.

```
if (p->swap_device) {
        *dev = p->swap_device;
```

Sets `dev` to the `swap_device` of this swap type, if any, and returns.

```
} else if (p->swap_file) {
        *swapf = p->swap_file->d_inode;
```

There is no `swap_device`, so checks for a swap file (`swap_file`), assigns the inode of this swap file to `swapf` and returns.

```
} else {
        printk(KERN_ERR "rw_swap_page: no swap file or device\n");
}
return;
```

An error ocurred: no swap file nor swap device. Prints a warning message then and return.

8.8.7 Function valid_swaphandles()

File: mm/swapfile.c
Prototype:

```
int valid_swaphandles(swp_entry_t entry, unsigned long *offset)
```

This function returns the initial offset of the swap cluster to the read ahead (in the `offset` parameter) and the number of swap entries that must be read from disk (return value, which is an **int**).

```
int ret = 0, i = 1 \<\< page_cluster;
unsigned long toff;
struct swap_info_struct *swapdev = SWP_TYPE(entry) + swap_info;

if (!page_cluster)      /* no readahead */
        return 0;
```

If `page_cluster` is zero, it means that swap pages shouldn't be grouped, so no readahead must be performed.

```
toff = (SWP_OFFSET(entry) >> page_cluster) \<\< page_cluster;
```

Finds the offset of the first entry in the cluster this entry is located in.

```
if (!toff)              /* first page is swap header */
        toff++, i--;
```

If the first entry in the cluster is the first offset of this swap type, skips it, and decrements the cluster size.

```
*offset = toff;
```

Set the `offset` to the first offset in the cluster (or second, if the first was the first block on the swap).

```
swap_device_lock(swapdev);
do {
        /* Don't read-ahead past the end of the swap area */
        if (toff >= swapdev->max)
                break;
        /* Don't read in free or bad pages */
        if (!swapdev->swap_map[toff])
                break;
        if (swapdev->swap_map[toff] == SWAP_MAP_BAD)
                break;
        toff++;
        ret++;
} while (--i);
swap_device_unlock(swapdev);
return ret;
```

Computes the number of pages that will be read ahead. Only contiguous, used and good pages up to the end of the swap area will be read. If any of these conditions happen to be false, returns the number computed so far.

8.9 Swap Cache

8.9.1 Function swap_writepage()

File: mm/swap_state.c
Prototype:

```
static int swap_writepage(struct page *page)
```

This function is used to write a swap page (`page` parameter). It is called from `shrink_cache()` and always return zero.

```
if (remove_exclusive_swap_page(page)) {
        UnlockPage(page);
        return 0;
}
```

If that an exclusive swap page (ie, the swap page is the only user of this swap entry), tries to remove it from swap cache, since it doesn't need to be actually written to the swap.

```
rw_swap_page(WRITE, page);
return 0;
```

That's not an exclusive swap page, so simple call `rw_swap_page()` to write it.

8.9.2 Function add_to_swap_cache()

File: mm/swap_state.c
Prototype:

```
int add_to_swap_cache(struct page *page, swp_entry_t entry)
```

This function adds a page (`page` parameter) to the swap cache, setting the page to the swap entry passed as parameter (`entry`). It returns an **int** value that corresponds to the error, so a **zero** value means that the page has been added succesfully.

```
if (page->mapping)
        BUG();
```

Only pages that are not in the page cache are eligible to be added to the swap cache (which is part of page cache).

```
if (!swap_duplicate(entry)) {
        INC_CACHE_INFO(noent_race);
        return -ENOENT;
}
```

Gets a reference on this swap entry. If `swap_duplicate()` fails to do that, probably a race condition happened. In this case, returns `-ENOENT` (invalid entry) error.

```
if (add_to_page_cache_unique(page, &swapper_space, entry.val,
                page_hash(&swapper_space, entry.val)) != 0) {
```

Calls `add_to_page_cache_unique()` to add the page to the page cache. The called function is supposed to be race condition proof. A non-zero return value means that it has raced, so a swap page set to this entry has been added before it could make it. In this case, returns (see below).

```
        swap_free(entry);
        INC_CACHE_INFO(exist_race);
        return -EEXIST;
}
```

The page couldn't be added to the page cache because another swap page was added to the page cache before that, so drops the reference on the swap entry (`swap_free()`) and returns `-EEXIST` error.

```
if (!PageLocked(page))
        BUG();
```

Makes sure the page was locked when added to page cache and it remains locked.

```
if (!PageSwapCache(page))
        BUG();
```

Verifies if the page mapping is set to `swapper_space`, as expected.

```
INC_CACHE_INFO(add_total);
return 0;
```

The page has been sucessfully added to the page cache, then return zero as error value.

8.9.3 Function __delete_from_swap_cache()

File: mm/swap_state.c
Prototype:

```
void __delete_from_swap_cache(struct page *page)
```

This function removes the page from swap cache, but does not drop the reference on the swap entry nor the reference page cache has got on this page. The caller must hold the `pagecache_lock` spinlock.

```
if (!PageLocked(page))
        BUG();
if (!PageSwapCache(page))
        BUG();
```

Checks if the page is locked and actually in swap cache.

```
ClearPageDirty(page);
__remove_inode_page(page);
INC_CACHE_INFO(del_total);
```

Clears the dirty bit and removes the page from page cache (`__remove_inode_page()`).

8.9.4 Function delete_from_swap_cache()

File: mm/swap_state.c
Prototype:

```
void delete_from_swap_cache(struct page *page)
```

This function flushes the swap cache page and deletes it from the swap cache, dropping the reference on the swap entry and also the reference page cache has got on this page.

```
swp_entry_t entry;

if (!PageLocked(page))
        BUG();
```

Makes sure this page is locked.

```
block_flushpage(page, 0);
```

Flushes the page to the disk.

```
entry.val = page->index;
```

Gets the entry value before it gets deleted from swap cache, since the reference on this swap entry will be only dropped later.

```
spin_lock(&pagecache_lock);
__delete_from_swap_cache(page);
spin_unlock(&pagecache_lock);
```

Deletes the page from swap cache.

```
swap_free(entry);
page_cache_release(page);
```

Drops the reference on the swap entry and the reference that page cache has got on the page.

8.9.5 Function free_page_and_swap_cache()

File: mm/swap_state.c
Prototype:

```
void free_page_and_swap_cache(struct page *page)
```

The main role of this function is to drop a reference on a page. It will also, if it is a swap cache page and it is able to lock the page at once, check if that's an exclusive swap page, remove it from swap cache and drop its reference on the swap entry.

```
/*
 * If we are the only user, then try to free up the swap cache.
 *
 * Its ok to check for PageSwapCache without the page lock
 * here because we are going to recheck again inside
 * exclusive_swap_page() _with_ the lock.
 *                                      - Marcelo
 */
if (PageSwapCache(page) && !TryLockPage(page)) {
        remove_exclusive_swap_page(page);
        UnlockPage(page);
}
page_cache_release(page);
```

8.9.6 Function lookup_swap_cache()

File: mm/swap_state.c
Prototype:

```
struct page * lookup_swap_cache(swp_entry_t entry)
```

This function looks up a certain swap entry (`entry` parameter) in the swap cache, getting a reference on the page if found. It returns a **pointer to the found page**, if any, or null, otherwise.

```
struct page *found;

found = find_get_page(&swapper_space, entry.val);
```

Searches for this entry in the page cache. Pages from swap cache are mapped to `swapper_space`, so it must simply look for pages mapped to this address space and to the wanted entry.

```
/*
 * Unsafe to assert PageSwapCache and mapping on page found:
 * if SMP nothing prevents swapoff from deleting this page from
 * the swap cache at this moment.  find_lock_page would prevent
 * that, but no need to change: we _have_ got the right page.
 */
INC_CACHE_INFO(find_total);
if (found)
        INC_CACHE_INFO(find_success);
return found;
```

8.9.7 Function read_swap_cache_async()

File: mm/swap_state.c
Prototype:

```
struct page * read_swap_cache_async(swp_entry_t entry)
```

This function tries to find an entry in the swap cache. If not found, it allocates a page, adds it to the swap cache and reads the data into it from disk. A pointer to this page (found or added to the swap cache) is returned to the system.

```
struct page *found_page, *new_page = NULL;
int err;

do {
        /*
         * First check the swap cache.  Since this is normally
         * called after lookup_swap_cache() failed, re-calling
         * that would confuse statistics: use find_get_page()
         * directly.
         */
        found_page = find_get_page(&swapper_space, entry.val);
        if (found_page)
                break;
```

Searches the page cache for this entry. If found, returns the found page (`found_page`), freeing the page that might have been allocated (see below).

```
        /*
         * Get a new page to read into from swap.
         */
        if (!new_page) {
                new_page = alloc_page(GFP_HIGHUSER);
                if (!new_page)
                        break;          /* Out of memory */
        }
```

The page has not been found in page cache, hence allocates a new page if it has not yet been allocated. If it couldn't be allocated, return the `found_page`, i.e. a NULL pointer.

```
        /*
         * Associate the page with swap entry in the swap cache.
         * May fail (-ENOENT) if swap entry has been freed since
         * our caller observed it.  May fail (-EEXIST) if there
         * is already a page associated with this entry in the
         * swap cache: added by a racing read_swap_cache_async,
         * or by try_to_swap_out (or shmem_writepage) re-using
         * the just freed swap entry for an existing page.
         */
        err = add_to_swap_cache(new_page, entry);
        if (!err) {
```

```
                /*
                 * Initiate read into locked page and return.
                 */
                rw_swap_page(READ, new_page);
                return new_page;
        }
```

Adds the new page to the swap cache for this entry. If it could be sucessfully added to the swap cache (`!err`), reads the data from disk and returns it. If it couldn't be added to the swap cache, this swap entry might have been freed in the meanwhile (`-ENOENT`) or it may have been found in the page cache (`-EEXIST`). In these cases, do not read the page from disk since it it not in the swap cache.

```
} while (err != -ENOENT);
```

If the swap entry has not been freed (`err == -EEXIST`), tries again this whole procedure. Otherwise (`err == -ENOENT`) gives up trying, frees the allocated page and returns the `found_page`, i.e. a NULL pointer.

```
if (new_page)
        page_cache_release(new_page);
return found_page;
```

Appendix A

Intel Architecture

Work under progress ..... Most of the information that will be here will be from Intel Arch Manuals, so I will write it last. Anyone willing to take this up ?

This chapter is a refresher on how memory is addressed in the intel x86 processor. The concepts dealt here are also valid for other architectures also. The x86 processor supports two modes of addressing:

- Segmentation
- Paging

A.1 Segmentation

This addressing mode is the default and cannot be disabled. In real mode the address is specified by loading the segment register by a 16 bit value, to specify the base, and a general purpose register is loaded with the 16 bit offset. In protected mode, the segment register is loaded by a segment selector. The format of the segment selector is described in appendix . The most signinficant 13 bits are used as an index into the global descriptor table whose base address is contained in the GDTR register.

A.2 Paging

This addressing mode is enabled by setting the most significant bit (PG) of the CR0 register.

Appendix B

Miscellaneous

B.1 Page Flags

This section will describe the bit values the page→flags can have. The are all declared in include/linux/mm.h. This is a description of each bit.

PG_locked
: This bit is set when the page must be locked in memory for disk I/O. When I/O starts, this bit is set and released when it completes

PG_error
: If an error occurs during disk I/O, this bit is set

PG_referenced
: If a page is mapped and it is referenced through the mapping,index hash table, this bit is set. It's used during page replacement for moving the page around the LRU lists

PG_uptodate
: When a page is read from disk without error, this bit will be set.

PG_dirty
: This indicates if a page needs to be flushed to disk. When a page is written to that is backed by disk, it is not flushed immediately, this bit is needed to ensure a dirty page is not freed before it's written out

PG_unused
: This bit is literally unused

PG_lru
: If a page is on either the active_list or the inactive_list, this bit will be set.

PG_active
This bit is set if a page is on the active_list LRU and cleared when it is removed. It marks a page as been hot.

PG_slab
This will flag a page as been used by the slab allocator

PG_skip
Used by some architectures so skip over parts of the address space.

PG_highmem
Pages in high memory cannot be mapped permanently by the kernel. Pages that are in high memory are flagged with this bit during `mem_init`

PG_checked
Only used by the EXT2 file-system

PG_arch_1
Quoting directly from the code. *PG_arch_1 is an architecture specific page state bit. The generic code guarantees that this bit is cleared for a page when it first is entered into the page cache*

PG_reserved
This is set for pages that can never be swapped out. It is set during init until the machine as booted up. Later it is used to flag empty pages or ones that do not even exist

PG_launder
This bit is important only to the page replacement policy. When the VM wants to swap out a page, it will set this bit and call the writepage function. When scanning, it encounters a page with this bit and PG_locked set, it will wait for the I/O to complete

There are helper macros provided to help set, test and clear the bits.

Bit name	Set	Test	Clear
PG_locked	LockPage	PageLocked	UnlockPage
PG_error	SetPageError	PageError	ClearPageError
PG_referenced	SetPageReferenced	PageReferenced	ClearPageReferenced
PG_uptodate	SetPageUptodate	PageUptodate	ClearPageUptodate

PG_dirty	SetPageDirty	PageDirty	ClearPageDirty
PG_unused	n/a	n/a	n/a
PG_lru	TestSetPageLRU	PageLRU	TestClearPageLRU
PG_active	SetPageActive	PageActive	ClearPageActive
PG_slab	PageSetSlab	PageSlab	PageClearSlab
PG_skip	n/a	n/a	n/a
PG_highmem	n/a	PageHighMem	n/a
PG_checked	SetPageChecked	PageChecked	n/a
PG_arch_1	n/a	n/a	n/a
PG_reserved	SetPageReserved	PageReserved	ClearPageReserved
PG_launder	SetPageLaunder	PageLaunder	ClearPageLaunder

B.2 GFP Flags

A persistent concept through out the whole VM are the **GFP (Get Free Page)** flags. They determine how the allocator and kswapd may behave for the allocation and freeing of pages. For example, an interrupt handler may not sleep so it will *not* have the __GFP_WAIT flag set, as this flag indicates the caller may sleep. There are three sets of GFP flags, all defined in include/linux/mm.h.

The first set are zone modifiers. These flags indicate that the caller must try to allocate from a particular zone. The reader will note that there is no zone modifier for ZONE_NORMAL. This is because the zone modifier flag is used as an offset within an array and 0 implicitly means allocate from ZONE_NORMAL.

__GFP_DMA	Allocate from ZONE_DMA if possible
__GFP_HIGHMEM	Allocate from ZONE_HIGHMEM if possible
GFP_DMA	Alias for __GFP_DMA

The next flags are action modifiers. They change the behavior of the VM and what the calling process may do.

__GFP_WAIT
Indicates that the caller is not high priority and can sleep or reschedule

__GFP_HIGH
Used by a high priority or kernel process. Kernel 2.2.x used it to determine if a process could access emergency pools of memory. In 2.4.x kernels, it does not appear to be used

__GFP_IO
Indicates that the caller can perform low level IO. In 2.4.x, the main affect this has is determining if **try_to_free_buffers()** can flush buffers or not. It is used by at least one journelled file-system

__GFP_HIGHIO
Determines that IO can be performed on pages mapped in high memory. Only used in **try_to_free_buffers()**

__GFP_FS
Indicates if the caller can make calls to the file-system layer. This is used when the caller is file-system related, the buffer cache for instance, and wants to avoid recursively calling itself

These flags on their own are too primitive to be easily used. Knowing what the correct combinations for each instance is unwieldy and leads to buggy programming so a few high level combinations are defined to make life simpler. For clarity the __GFP_ is removed from the below combinations. So, the __GFP_HIGH flag will read as HIGH below. The combinations and their flags are

GFP_ATOMIC	HIGH
GFP_NOIO	HIGH | WAIT
GFP_NOHIGHIO	HIGH | WAIT | IO
GFP_NOFS	HIGH | WAIT | IO | HIGHIO
GFP_KERNEL	HIGH | WAIT | IO | HIGHIO | FS
GFP_NFS	HIGH | WAIT | IO | HIGHIO | FS
GFP_USER	WAIT | IO | HIGHIO | FS
GFP_HIGHUSER	WAIT | IO | HIGHIO | FS | HIGHMEM
GFP_KSWAPD	WAIT | IO | HIGHIO | FS

To help understand this, take GFP_ATOMIC as an example. It has only the __GFP_HIGH flag set. This means it is high priority, use emergency pools (if they existed) but will not sleep, perform IO or access the file-system. This would be the case for an interrupt handler for example. The following is a description of where the combined flags are used.

GFP_ATOMIC
: This flag is used whenever the caller cannot sleep and must be serviced if at all possible. Any interrupt handler that requires memory must use this flag to avoid sleeping or IO. Many subsystems during init will use this system such as `buffer_init` and `inode_init`

GFP_NOIO
: This is used by callers who are already performing an IO related function. For example, when the loop back device is trying to get a page for a buffer head, it uses this flag to make sure it will not perform some action that would result in more IO. In fact, it appears this flag was introduced specifically to fix a loopback device deadlock

GFP_NOHIGHIO
: This is only used in one place, in `alloc_bounce_page()` during the creating of a bounce buffer for IO

GFP_NOFS
: This is only used by the buffer cache and file-systems to make sure they do not recursively call themselves by accident

GFP_KERNEL
The most liberal of the combined flags. It indicates that the caller is free to do whatever it pleases. Strictly speaking the difference between this flag and GFP_USER is that this could use emergency pools of pages but that is a no-op on 2.4.x kernels

GFP_NFS
This flag is defunct. In the 2.0.x series, this flag determined what the reserved page size was. Normally 20 free pages were reserved. If this flag was set, only 5 would be reserved. Now it is not treated differently anywhere anymore

GFP_USER
Another flag of historical significance. In the 2.2.x series, an allocation was given a LOW, MEDIUM or HIGH priority. If memory was tight, a request with GFP_USER (low) would fail where as the others would keep trying. Now it has no significance and is not treated any different to GFP_KERNEL

GFP_HIGHUSER
This flag indicates that the allocator should allocate from ZONE_HIGHMEM if possible. It is used when the page is allocated on behalf of a user process

GFP_KSWAPD
More historical significance. In reality this is not treated any different to GFP_KERNEL

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some

word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent

copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- State on the Title page the name of the publisher of the Modified Version, as the publisher.

- Preserve all the copyright notices of the Document.

- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- Include an unaltered copy of this License.

- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections

as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Bibliography

[1] Daniel P. Bovet & Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001, ISBN 81-7366-233-9.

[2] Joe Knapka. *Outline of the Linux Memory Management System*. http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html.

[3] Linux MM. *Website and mailing list for linux-mm*. http://www.linux-mm.org. Has a lot of links to memory management documentation.

[4] Intel Architecture. *Intel Pentium III Processor Manuals*. http://www.intel.com/design/PentiumIII/manuals/index.htm.

[5] Martin Devera. *Functional Callgraph of the Linux VM*. http://luxik.cdi.cz/~devik/mm.htm. Contains a patch for gcc which was used to create the call-graph poster provided with this doc.

[6] Mel Gorman. *Documentation Patches for the linux kernel*. http://www.csn.ul.ie/~mel/projects/vm/. Along with documenting the linux VM, commented a lot of code which are available as patches. One of the main sources for the material in this document.

[7] Jeff Bonwick. *The Slab Allocator: An Object Caching Kernel Memory Allocator*. http://www.usenix.org/publications/library/proceedings/bos94/bonwick.html. This paper presents a comprehensive design overview of the SunOS 5.4 kernel memory allocator.

[8] Ralf Brown. *Interrupt List*. http://www.ctyme.com/rbrown.htm. This list contains every documented and undocumented interrupt call known.

Index

G

H

K

L

M

T

V

W

Z