APPENDICES :: LINUX MEMORY MANAGEMENT

Table of Contents

Appendix A : Memory Management Hardware Layer on IA-32 (x86)....3
Appendix B : Linux OS Kernel Segment on the Intel IA-32....11
[L]PAE....15
Appendix C : ARM (32) Processor Exceptions and Modes....17
ARM Processor Exceptions and Modes....17
Exceptions, Interrupts, and the Vector Table....18
Appendix D : Translation Lookaside Buffer (TLB) and Caches....20
Caches....26
Appendix E : Magic SysRq 'm' - the show_mem Functionality....29
Appendix F : A Brief Note on Kernel Threads....39
Appendix G :: 'Buffers' and 'Cached' Memory....41
Appendix H :: Interpretation of the */proc/meminfo* (meminfo) output....44
Appendix I :: Virtual to Physical Address Translation....49
Appendix J :: (Some) VM Complexities....53
3.5.2 Avoiding Fragmentation....53
Grouping Pages by Mobility....53
CMA : The Contiguous Memory Allocator....63
Per-CPU Page (pcp) Lists....68
Appendix K :: COW (Copy-On-Write) Internal Handling....71
More on COW....71
Bibliography....77

Appendix A : Memory Management Hardware Layer on IA-32 (x86)

<< Note: IA-32 architecture largely apply to the *Intel Atom* processor family as well (with additional power-saving features of course). >>

Tip
An interesting animation on how Segmentation in hardware (basically) works can be viewed here!

Hardware Segmentation

When operating in protected mode, all accesses go through the Global Descriptor Table (**GDT**) or Local Descriptor Table (LDT) [1]. These tables contain entries called **segment descriptors**. A segment descriptor provides the base address of a segment and access rights, type and usage information. Each segment descriptor has a **segment selector** associated with it. The segment selector provides an index into the GDT or LDT (to its associated segment descriptor), a global/local flag (that determines whether the segment selector points to the GDT/LDT), and access rights information.

To access a byte in a segment, a segment selector and an offset are supplied. The segment selector provides access to the segment descriptor for the segment (in the GDT or LDT). From the segment descriptor, the process obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data or stack segment in the GDT or LDT, provided the segment is accessible in the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

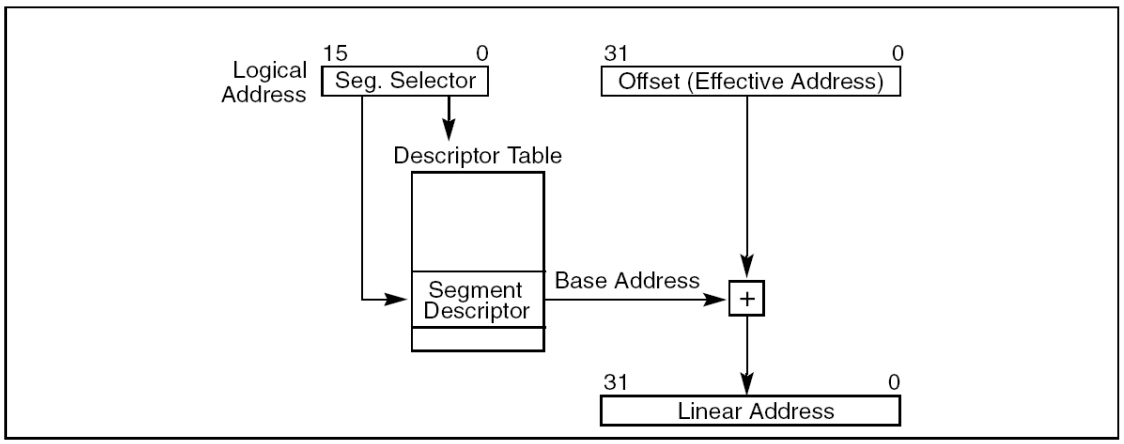


Figure 3-5. Logical Address to Linear Address Translation

[Figure above from Intel's "IA-32 Intel Architecture Software Developers Manual" Vol 3, page 73]

Segments are just used to carve up the linear address space into arbitrary chunks. The linear space is what's managed by the VM subsystem [2]. The x86 architecture supports segmentation in hardware: you can specify addresses as offsets into a particular segment, where a segment is defined as a range of linear (virtual) addresses with particular characteristics such as protection. In fact, you *must* use the segmentation mechanism on x86 machines; so we set up four segments, each spanning the entire virtual address space*:

- A kernel text segment from 0 to 4GB.
- A kernel data segment from 0 to 4GB.
- A user text segment from 0 to 4GB.
- A user data segment from 0 to 4GB.

* We know that every memory access always goes through a segment. On the 386 processor and later, because of 32-bit segment offsets and limits, it is possible to make segments cover the entire addressable memory, which makes segment-relative addressing transparent to the user.

Additional Resource

Memory Management: **Paging**, by Paul Krzyzanowski
Look up the *Case Study / Intel IA-32 and x86-64* section.

Source

Modern x86 kernels use a “flat memory model” without any segmentation. Flat model means that the “logical” to “linear” (virtual) addresses coincide, so we can basically ignore segmentation. But all of the linear addresses are still fed to the paging unit (assuming paging is turned on), so there’s more magic that happens there to transform a linear address into a physical address.

On a modern x86 kernel, address translation is done in two steps:

1. Logical (seg:off) address → Linear (virtual) address
2. Linear (virtual) address → Physical address

Flat model eliminates the first step (logical to linear), but the second step remains.

…

Having the same segment only affects the translation of logical addresses into linear addresses - and flat mode makes them the same. Most CPUs don’t even have a distinction between “logical” and “linear” - they only deal with linear and physical addresses. It’s an accident of history that x86 ended up with this logical/linear distinction, and x86-64 <<*almost*>> basically gets rid of it.

But each process still has a different set of _page tables_ mapping its 32-bit linear address space

into physical addresses. Processes have their own _page tables_, not segments.

<< *Source* >>

Thus, we effectively allow access to the entire virtual address space using any of the available segment selectors.

Questions:

- Where are the segments set up?

  Answer: the Global Descriptor Table is defined in head.S at line 450. The GDT register is loaded to point at the GDT on line 250. << links are for an old kernel version >>

- Why separate kernel and user segments, if they both permit access to the entire 4GB range?

  Answer: the properties of the kernel and user segments differ:

```
.quad 0x00cf9a000000ffff      /* 0x10 kernel 4GB code at 0x00000000 */
.quad 0x00cf92000000ffff      /* 0x18 kernel 4GB data at 0x00000000 */
.quad 0x00cffa000000ffff      /* 0x23 user   4GB code at 0x00000000 */
.quad 0x00cff2000000ffff      /* 0x2B user   4GB data at 0x00000000 */
```

  The segment registers (CS, DS, etc.) contain a 13-bit index into the descriptor table; the descriptor at that index tells the CPU the properties of the selected segment. The 3 low-order bits of a segment selector are not used to index the descriptor table; rather, they contain the descriptor-type (global or local) and the requested privilege level. Thus the kernel segment selectors 0x10 and 0x18 use RPL 0, while the user selectors 0x23 and 0x2B use RPL 3, the least-privileged level.

  Also notice that the high nibble of the third high-order byte differs in the kernel and user cases: in the kernel case, the Descriptor Privilege Level is 0 (most privileged), while the user segment descriptors' DPL is 3 (least privileged). If you read the Intel documentation, you will be able to figure out exactly what all this means, but since x86 segment protection does not figure much in the Linux kernel, I won't discuss it any further here.

All segment descriptors in Linux are stored in the **Global Descriptor Table** (the GDT). Essentially, the GDT holds pointers to the actual segments- the kernel code, kernel data, user code & user data segments. In addition, it also points to APM code & data, per-CPU information, etc. The base address of the GDT is stored in the GDTR register of each processor.

LDTs are not used by Linux but a modify_ldt() system call is available for processes to create their own LDTs (useful for segment-oriented Windows-like applications on Linux- eg. The Wine project).

Segments and the GDT

1. Kernel Text (code) segment:
Some of the fields of it's Segment Descriptor (64 bits long) are:

Base:	0x00000000	=> segment start address
Limit:	0x000FFFFF	=> segment end address
G:	Granularity flag = 1	=> segment size in pages
S:	System flag = 1	=> normal code/data segment
Type:	0xA	=> code segment has read & execute
DPL:	Descriptor Privilege Level = 0	=> Kernel Mode
D/B:	32-bit address flag = 1	=> 32-bit offset addresses

- Linear address space is whole of virtual memory from 0 to 2^{32} -1 (4 GB).
- The Segment Selector (for this kernel code entry) is accessed in kernel code via the __KERNEL__CS macro. To address the segment, kernel code just loads the value yielded by this macro into the CS register of the CPU.

<< The Linux x86 implementation of the GDT / LDT can be seen here:
include/asm-x86/desc_defs.h:struct desc_struct
>>

2. Kernel Data segment:
Some of the fields of it's Segment Descriptor (64 bits long) are:

Base:	0x00000000	=> segment start address
Limit:	0x000FFFFF	=> segment end address
G:	Granularity flag = 1	=> segment size in pages
S:	System flag = 1	=> normal code/data segment
Type:	2	=> data segment has read & write
DPL:	Descriptor Privilege Level = 0	=> Kernel Mode
D/B:	32-bit address flag = 1	=> 32-bit offset addresses

- This segment is actually identical to the above kernel code segment, except for the Type field.
- Linear address space is whole of virtual memory from 0 to 2^{32} -1 (4 GB).
- The Segment Selector (for this kernel data entry) is accessed in kernel code via the __KERNEL__DS macro. To address the segment, kernel code just loads the value yielded by this macro into the DS register of the CPU.

3. User Text (Code) segment:
Some of the fields of it's Segment Descriptor (64 bits long) are:

Base:	0x00000000	=> segment start address
Limit:	0x000FFFFF	=> segment end address
G:	Granularity flag = 1	=> segment size in pages
S:	System flag = 1	=> normal code/data segment
Type:	0xA	=> code segment has read & execute

DPL:	Descriptor Privilege Level = 3	=> User Mode
D/B:	32-bit address flag = 1	=> 32-bit offset addresses

- This segment is shared by all processes executing in user mode.
- Linear address space is whole of virtual memory from 0 to 2^{32} -1 (4 GB).
- The Segment Selector (for this user code entry) is accessed (both in kernel & user mode) via the __USER__CS macro.

4. User Data segment:

Some of the fields of it's Segment Descriptor (64 bits long) are:

Base:	0x00000000	=> segment start address
Limit:	0x000FFFFF	=> segment end address
G:	Granularity flag = 1	=> segment size in pages
S:	System flag = 1	=> normal code/data segment
Type:	2	=> data segment has read & write
DPL:	Descriptor Privilege Level = 3	=> User Mode
D/B:	32-bit address flag = 1	=> 32-bit offset addresses

- This segment is shared by all processes executing in user mode.
- Linear address space is whole of virtual memory from 0 to 2^{32} -1 (4 GB).
- The Segment Selector (for this user data entry) is accessed (both in kernel & user mode) via the __USER__DS macro.

5. Task State Segment (TSS) segment:

- There is a TSS for each processor
- The TSS is small (just 236 bytes long) and is thus stored within the virtual address space corresponding to the kernel data segment.
- The TSS holds
  - *hardware context* information for each CPU (typically used to effect context switching correctly)
  - I/O port permissions
  - Privilge level stack pointers
  - previous TSS link

- Actually speaking, Linux 2.6 does *not* use hardware context switching - it uses software to perform a process (context) switch; the reasons for this is:
  1. Better checking (on segmentation register values) is implemented.
  2. The time difference between hardware & software context switching is not significant.

- If this is the case, why use a TSS to hold hardware context at all?
  The reasons are:
  1. During the U->K mode switch, the x86 CPU gets the address of the Kernel Mode stack from the TSS.
  2. The ability of a User Mode process to access an I/O port (in/out instructions) is restricted; a

permission check may be required, which uses a permission bitmap that is stored in the TSS.

<< Ref: TSS on Wikipedia.org >>

6. Local Descriptor Table (LDT) segment:

- Most Linux userspace applications do not make use of an LDT, so Linux defines just one default LDT to be shared by most processes
- The default LDT has five entries but only two are effectively made use of: a call gate* for iBCS executables, and a call gate for Solaris x86 binaries.

* ***Call Gates*** *are a mechanism provided by 80x86 processors to change the privilege level of the CPU while invoking a predefined function.*
<< Ref: http://en.wikipedia.org/wiki/Call_Gate >>

The Current Privilege Level (**CPL**) of the CPU indicates whether the process is in User or Kernel Mode (specified by the RPL field of the segment selector). Whenever the CPL changes, some segmentation registers need to be updated. For instance, when the CPL is 3 (User Mode), the DS register must contain the Segment Selector of the User Data segment; when the CPL is 0 (Kernel Mode), the DS register must contain the Segment Selector of the Kernel Data segment (similarly for the Stack segment).

Segment Selector

The Segment Selector (16 bits) is not just a simple index into the GDT/LDT. Rather, it has an index (as the MSB 13 bits), a Table Indicator field (1 bit) which determines whether the index is to the GDT (value 0) or LDT (value 1) and the 2 LSB bits determine the CPL (Current Privilege Level): Intel provides four “ring levels”; most modern OS's on x86 make use of only two (including Windows, Solaris x86 and Linux x86); 0 implies only a task running in kernel-mode. RPL 0 can legally make the access. 3 implies a task running in User-mode.

[Figure from Intel's “IA-32 Intel Architecture Software Developers Manual” Vol 3 page 74]

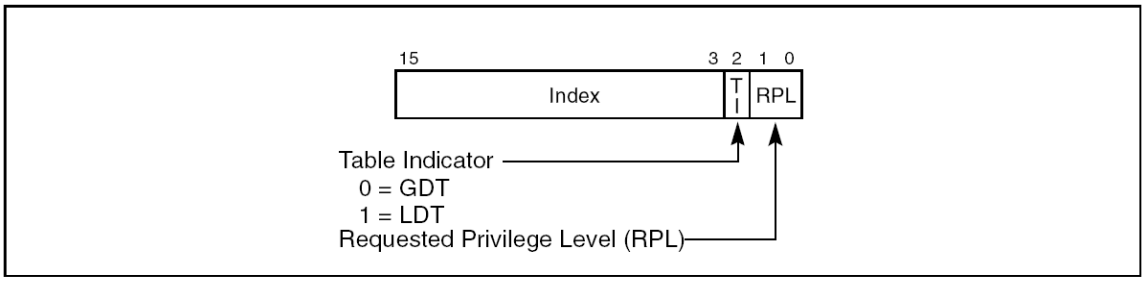


Figure 3-6. Segment Selector

The CS register's LSB 2 bits always represent the CPL (Current Privilege Level) the CPU is currently executing code in; only two values are valid on Linux – 0 (0000b) and 3 (0011b) representing Kernel and User mode CPL respectively.

SIDEBAR :: user_mode

The kernel uses the inline “user_mode” function to determine whether a register set came from user mode or not:

…

arch/x86/include/asm/segment.h

```
...
/* Bottom two bits of selector give the ring privilege level */
#define SEGMENT_RPL_MASK    0x3
...
```

arch/x86/include/asm/ptrace.h

```
...
/* User mode is privilege level 3 */
#define USER_RPL                0x3
...
static inline int user_mode(struct pt_regs *regs)
{
#ifdef CONFIG_X86_32
        return (regs->cs & SEGMENT_RPL_MASK) == USER_RPL;
#else
        return !!(regs->cs & 3);
#endif
}
```

…

SIDEBAR

Actual usage example of changing context to userspace from kernel-mode :: Signal Handling within the kernel

Context: A process is running in kernel mode; a user-space signal (or from kernel) is delivered. The process must *handle the signal. The kernel takes over: it will invoke the registered signal handler routine, which always resides in usewr-space. How exactly is this done?*

On the 2.6.30 kernel:
in *arch/x86/kernel/signal.c* we find- (call trace):

```
do_signal -> handle_signal -> setup_rt_frame -> ia32_setup_frame ->
__setup_frame
```

__setup_frame sets up the calling context to be userspace and sets cpu registers such that the userspace signal handler code is executed. The final part of the func does:

```
...
/* Set up registers for signal handler */
regs->sp = (unsigned long)frame;
regs->ip = (unsigned long)ka->sa.sa_handler;
```

```
regs->ax = (unsigned long)sig;
regs->dx = 0;
regs->cx = 0;

regs->ds = __USER_DS;
regs->es = __USER_DS;
regs->ss = __USER_DS;
regs->cs = __USER_CS;

return 0;
}
```

--setting up:
-the usermode stack ptr (sp) to the appropriate stack frame (constructed above),
-the accumulator (EAX) to hold the signal #
-the IP to hold the code addr of the sig handler fn
-the segment regs to point to userspace segments (via macro __USER_[C|D]S).

A Quick Note:

Check out the sophistication of signal handling within the kernel (to a target process) here:

kernel/signal.c:force_sig_info

This example originates from the kernel page fault handling code; upon discovering a user bug, it 'forces' the SIGSEGV signal onto the process context!

Appendix B : Linux OS Kernel Segment on the Intel IA-32

Source: "Professional Linux Kernel Architecture", W Mauerer, Wrox Press

[Note: IA-32 x86 specific]

<<
Note:
The complexity and considerations below on *highmem* and mapping those pages into the kernel segment, persistent and fixed mappings, essentially disappear on most 64-bit architectures (and several 32-bit ones as well!); the IA-32 is really the one where these mechanisms are required.
>>

Division of address space in a ratio of 3 : 1 is only an approximate reflection of the situation in the kernel as the kernel address space itself is split into various sections. Figure 3-15 graphically illustrates the situation.

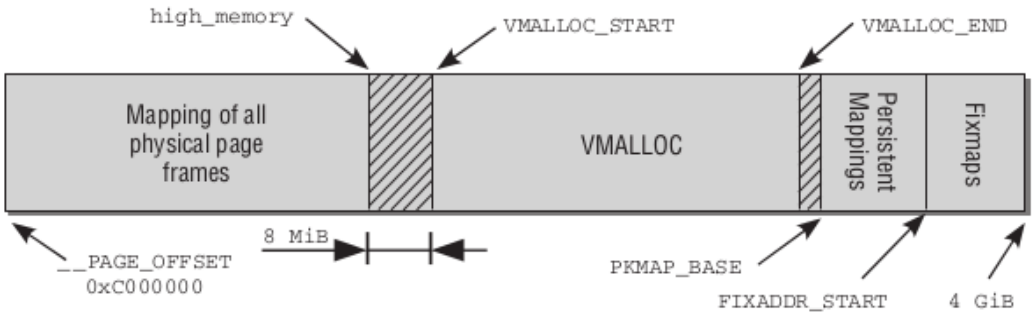


Figure 3-15: Division of the kernel address space on IA-32 systems.

The first section of the address space is used to map all physical pages of the system into the virtual address space of the kernel *<< the so-called "direct-mapped" or "identity-mapped" region >>*.

Because this address space begins at an offset of 0xC0000000 — the frequently mentioned 3 GiB — each virtual address x corresponds to the physical address x − 0xC0000000, and is therefore a simple linear shift.

As the figure shows, the direct mapping area extends from 0xC0000000 to the *high_memory address* whose exact value I discuss shortly
<< essentially, *high_memory address = __PAGE_OFFSET + amount of physical RAM* >>.

As intimated in Chapter 1, there is a problem with this scheme. Because the virtual address space of the kernel comprises only 1 GiB, a maximum of 1 GiB of RAM memory can be mapped.

The fact that the maximum memory configuration on IA-32 systems (without PAE) can be up to 4 GiB raises the question of what to do with the remaining memory.

Here's the bad news. The kernel cannot map the whole of physical memory at once if it is larger than 896 MiB. [13]

<<
Good LWN article on low and high memory on 32-bit Linux:
Virtual Memory I: The Problem
>>

[13] It would also be possible to get rid of the split completely by introducing two 4 GiB address spaces, one for the kernel and one for each userspace program. However, context switches between kernel and user mode are more costly in this case << the hardware TLB would require to be flushed on every user<->kernel transition; this is considered *very* expensive >>.

This value is even less than the previously stated maximum limit of 1 GiB because the kernel must reserve the last 128 MiB of its address space for other purposes << *the so-called persistent kernel ("pkmap") and fixed ("fixmap") mappings* >> which I explain shortly. Adding these 128 MiB to the 896 MiB of direct RAM mapping results in a total virtual kernel address space of 1,024 MiB = 1 GiB.

The kernel uses the two frequently employed abbreviations "normal" and "highmem" to distinguish between pages that can be mapped directly and those than cannot.

The kernel port must provide two macros for each architecture to translate between physical and virtual addresses in the identity-mapped part of virtual kernel memory (ultimately this is a platform-dependent task). [14]

- ❑ __pa(vaddr) returns the physical address associated with the virtual address vaddr .
- ❑ __va(paddr) yields the virtual address corresponding to the physical address paddr .

Both functions operate with void pointers and with unsigned long s because both data types are equally valid for the representation of memory addresses.

[14] The kernel places only two conditions on the functions that must remain as invariants;

```
x1 < x2 ⇒ __va (x1) < __va (x2)
```

must be valid (for any physical addresses x i), and

```
__va ( __pa (x)) = x
```

must be valid for any addresses x within the direct mapping.

Caution: The functions are not valid to deal with arbitrary addresses from the virtual address space, but only work for the identity-mapped part! This is why they can usually be implemented with simple linear transformations and do not require a detour over the page tables.

IA-32 maps the page frames into the virtual address space starting from PAGE_OFFSET , and correspondingly the following simple transformation is sufficient:

include/asm-x86/page_32.h

```
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

For what purpose does the kernel use the last 128 MiB of its address space
<< FIXMAP_BASE to 4 GB >> ? As Figure 3-15 shows, it is put to three uses:

1. Virtually contiguous memory areas that are not contiguous in physical memory can be reserved in the vmalloc area. While this mechanism is commonly used with user processes, the kernel itself tries to avoid non-contiguous physical addresses as best it can. It usually succeeds because most of the large memory blocks are allocated for the kernel at boot time when RAM is not yet fragmented. However, on systems that have been running for longer periods, situations can arise in which the kernel requires physical memory but the space available is not contiguous. A prime example of such a situation is when modules are loaded dynamically.

2. Persistent mappings are used to map non-persistent pages from the highmem area into the kernel.

<< FYI, *more details:*

The kmap function must be used if highmem pages are to be mapped into kernel address space for a longer period (as a persistent mapping).

arch/arm/mm/highmem.c

```
void *kmap(struct page *page)
{
    might_sleep();
    if (!PageHighMem(page))
        return page_address(page);
    return kmap_high(page);
}
EXPORT_SYMBOL(kmap);
```

The page to be mapped is specified by means of a pointer to page as the function parameter. The function creates a mapping when this is necessary (i.e., if the page really is a highmem page) and returns the address of the data.

This task is simple if highmem support is not enabled << *typically the case on x86_64* >>. In this case, all pages can be accessed directly so it is only necessary to return the page address; there is no need to create a mapping explicitly.

The situation is more complicated if highmem pages are actually present << *typically the case on the IA-32 with physical RAM > 896 MB* >>. As with vmalloc , the kernel must first establish an association between the highmem pages and the addresses at which they are mapped. An area in virtual address space must also be reserved to map the pages, and finally, the kernel must keep track of which parts of the virtual area are already in use and which are still free.
...
>>

3. Fixmaps are virtual address space entries associated with a fixed but freely selectable page in physical address space. In contrast to directly mapped pages that are associated with RAM memory by means of a fixed formula, the association between a virtual fixmap address and the position in

RAM memory can be freely defined and is then always observed by the kernel.

<< *Fixmap mappings:*

Source "The 'fixmap' region refers to an area of virtual memory set aside for simple and fixed virtual memory mappings whose virtual addresses must be known at compile time and whose corresponding physical addresses can be changed later on. It lives at FIXADDR_START through to FIXADDR_TOP. Though it seems rarely used under ARM."

One downside to *kmap()* is that it might sleep; obviously, this is not allowable in interrupt contexts. Thus, there is a *kmap_atomic()* function (arch-specific implemetation) for these situations – it sets up a temporary kernel mapping between a (kernel) virtual page and a page frame in high memory.

The fixmap mechanism discussed earlier makes the memory needed to create atomic mappings available in the kernel address space. An area that can be used to map highmem pages is set up between FIX_KMAP_BEGIN and FIX_KMAP_END in the *fixed_addresses* array. The exact position is calculated on the basis of the CPU currently active and the desired mapping type.

arch/x86/include/asm/fixmap.h

```
…
 * for x86_32: We allocate these special addresses
 * from the end of virtual memory (0xfffff000) backwards.
 * Also this lets us do fail-safe vmalloc(), we
 * can guarantee that these special addresses and
 * vmalloc()-ed addresses never overlap.
 *
 * These 'compile-time allocated' memory buffers are
 * fixed-size 4k pages (or larger if used with an increment
 * higher than 1). Use set_fixmap(idx,phys) to associate
 * physical memory with fixmap indices.
 *
 * TLB entries of such buffers will not be flushed across
 * task switches.
 */
enum fixed_addresses {
#ifdef CONFIG_X86_32
    FIX_HOLE,
…
#ifdef CONFIG_X86_32
    FIX_KMAP_BEGIN, /* reserved pte's for temporary kernel
mappings */
    FIX_KMAP_END = FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1,
…
>>
```

Two pre-processor symbols are important in this context: __VMALLOC_RESERVE sets the size of the vmalloc area, and MAXMEM denotes the maximum possible amount of physical RAM that can be directly addressed by the kernel.

The splitting of memory into the individual areas is controlled by means of the constants shown in Figure 3-15. The constants may have different values depending on the kernel and system configuration. The bound of the direct mappings is specified by *high_memory* .

…

Mapping Functions on Machines without Highmem

Many architectures do not support high memory because they don't need it — 64-bit architectures head this list. << So do most ARM ports >>.

However, to permit use of the above functions without having to constantly distinguish between highmem and non-highmem architectures, the kernel defines several macros that implement compatible functions in normal memory (these are also used when highmem support is disabled on highmem-capable machines).

```
<highmem.h>
#ifdef CONFIG_HIGHMEM
...
#else
static inline void *kmap(struct page *page)
{
might_sleep();
return page_address(page);
}

#define kunmap(page)                do { (void) (page); } while (0)
#define kmap_atomic(page, idx)      page_address(page)
#define kunmap_atomic(addr, idx)    do { } while (0)
#endif
```

[OPTIONAL / FYI]

[L]PAE

- A noteworthy exception case of using 3-level paging on 32-bit platforms, is the case of Linux using the **PAE** (Physical Address Extension Mechanism) of Pentium Pro (& later) microprocessors to access "high" memory: here, 3-level paging comes into play (Linux's PGD corresponds to the CPU's Page Directory Pointer Table, Linux's PMD corresponds to the CPU Page Directory and Linux's Page Table to the CPU's Page Table), providing in effect, a 36-bit

address space (2^36 = 64GB).

<< The ARM Cortex (ARMv7 ISA) A7/12/15 family also supports PAE for more then 4GB ram support; here it's known as LPAE (Large PAE). Provides 40-bit PAE extending the address range to 1 TB ! >>

- However, it is important to realize that with a default 3:1 GB split of the virtual address space (user:kernel) per process, the 1GB of kernel VM becomes insufficient to hold paging tables for a very large number of processes: this effectively reduces the actual usable physical memory on 32-bit systems to 16GB RAM. To get around this limitation, both Windows 2003 Server and Linux provide a (kernel) configuration option to make the virtual address space **split as 2:2 GB**. Increasing the kernel VM space to 2GB makes it viable to support an actual 64GB of RAM; this, of course, is at the cost of reducing the maximum size of a user process to 2GB. Well, another case of the TANSTAAFL principle - “There Ain't No Such Thing As A Free Lunch”!

Note- The VM x:y GB split feature is available from the 2.6.16 Linux kernel (default is CONFIG_VMSPLIT_3G : under “Processor type and features / Memory Split” .

Additional Resources

Memory Management: **Paging**, by Paul Krzyzanowski

FYI (LWN):
“... It should be pointed out that high memory was a spectacularly successful failure, extending the useful life of 32-bit systems for some years. It still shows up in surprising places - you editor's phone is running a high-memory-enabled kernel. So calling high memory a failure is something like calling the floppy driver a failure; it may see little use now, but there was a time when we were glad we had it.
...”

Appendix C : ARM (32) Processor Exceptions and Modes

Source :: "ARM System Developer's Guide – Designing and Optimizing System Software" by Sloss, Symes and Wright, published by Elsevier.

An exception is any condition that needs to halt the normal sequential execution of instructions. Examples are when the ARM core is reset, when an instruction fetch or memory access fails, when an undefined instruction is encountered, when a software interrupt instruction is executed, or when an external interrupt has been raised. Exception handling is the method of processing these exceptions.

Most exceptions have an associated software exception handler—a software routine that executes when an exception occurs. For instance, a Data Abort exception will have a Data Abort handler. The handler first determines the cause of the exception and then services the exception. Servicing takes place either within the handler or by branching to a specific service routine. The Reset exception is a special case since it is used to initialize an embedded system.

…

ARM Processor Exceptions and Modes

Table 9.1 lists the ARM processor exceptions. Each exception causes the core to enter a specific mode. In addition, any of the ARM processor modes can be entered manually by changing the cpsr. User and system mode are the only two modes that are not entered by a corresponding exception, in other words, to enter these modes you must modify the cpsr.

When an exception causes a mode change, the core automatically :

Table 9.1 ARM processor exceptions and associated modes.

Exception	Mode	Main purpose
Fast Interrupt Request	*FIQ*	fast interrupt request handling
Interrupt Request	*IRQ*	interrupt request handling
SWI and Reset	*SVC*	protected mode for operating systems
Prefetch Abort and Data Abort	*abort*	virtual memory and/or memory protection handling
Undefined Instruction	*undefined*	software emulation of hardware coprocessors

- saves the cpsr to the spsr of the exception mode
- saves the pc to the lr of the exception mode

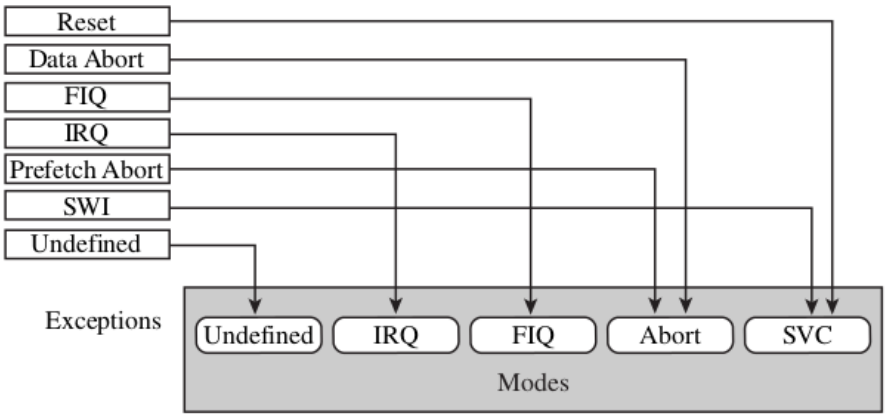


Figure 9.1 Exceptions and associated modes.

…

Exceptions, Interrupts, and the Vector Table

When an exception or interrupt occurs, the (ARM) processor sets the pc to a specific memory address. The address is within a special address range called the vector table. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt. << equivalent to the IDT on x86 >>.

The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher address in memory (starting at the offset `0xffff0000` – virtual address). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

■ Reset vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
■ Undefined instruction vector is used when the processor cannot decode an instruction.
■ Software interrupt vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine << *on Linux – system calls* >>
■ Prefetch abort vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
■ Data abort vector is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions << *this is what ultimately (from the MMU to OS fault handler to...) leads to your "Segmentation fault, [core dumped]" signals, etc* >>

Table 2.6 The vector table.

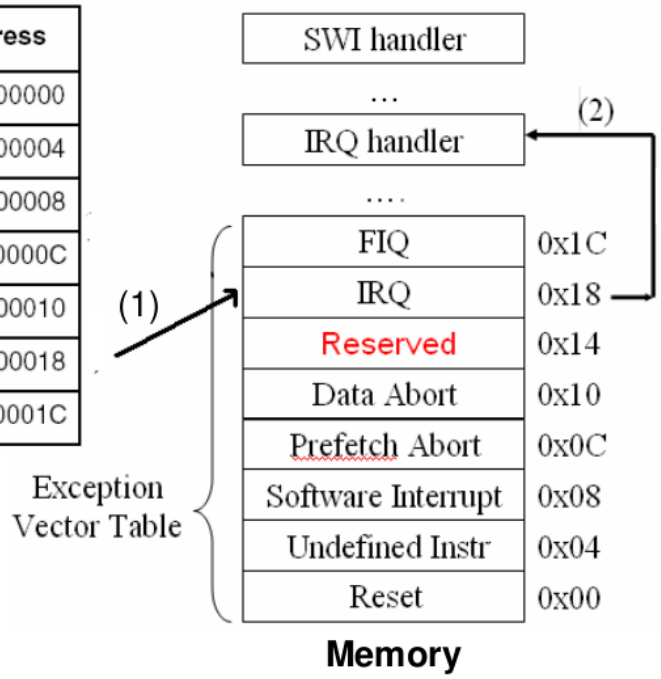
Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

- Reset vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- Interrupt request vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the cpsr.
- Fast interrupt request vector is similar to the interrupt request but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the cpsr.

...

• ARM Exception Vectors

Exception Type	Mode	Address
Reset	Supervisor	0x00000000
Undefined Instructions	Undefined	0x00000004
Software Interrupts (SWI)	Supervisor	0x00000008
Prefetch Abort	Abort	0x0000000C
Data Abort	Abort	0x00000010
IRQ (Normal Interrupt)	IRQ	0x00000018
FIQ (Fast interrupt)	FIQ	0x0000001C



Appendix D : Translation Lookaside Buffer (TLB) and Caches

Source: "Understanding the Linux Virtual Memory Manager" << ULVMM >> by Mel Gorman.

3.8 Translation Lookaside Buffer (TLB)

Initially, when the processor needs to map a virtual address to a physical address, it must traverse the full page directory searching for the PTE of interest. This would normally imply that each assembly instruction that references memory actually requires several separate memory references for the page table traversal [Tan01].

To avoid this considerable overhead, architectures take advantage of the fact that most processes exhibit a locality of reference, or, in other words, large numbers of memory references tend to be for a small number of pages. They take advantage of this reference locality by providing a Translation Lookaside Buffer (TLB), which is a small associative memory that caches virtual to physical page table resolutions.

Linux assumes that most architectures support some type of TLB, although the architecture-independent code does not care how it works. Instead, architecture-dependent hooks are dispersed throughout the VM code at points where it is known that some hardware with a TLB would need to perform a TLB-related operation. For example, when the page tables have been updated, such as after a page fault has completed, the processor may need to update the TLB for that virtual address mapping.

Not all architectures require these type of operations, but, because some do, the hooks have to exist. If the architecture does not require the operation to be performed, the function for that TLB operation will be a null operation that is optimized out at compile time.

A quite large list of TLB API hooks, most of which are declared in *<asm/pgtable.h>*, are listed in Tables 3.2 and 3.3, and the APIs are quite well documented in the kernel source by *Documentation/cachetlb.txt* [Mil00].

It is possible to have just one TLB flush function, but, because both TLB flushes and TLB refills are very expensive operations, unnecessary TLB flushes should be avoided if at all possible. For example, when context switching, Linux will avoid loading new page tables using Lazy TLB Flushing, discussed further in Section 4.3.

Table 3.2. Translation Lookaside Buffer Flush API

`void flush_tlb_all(void)`
This flushes the entire TLB on all processors running in the system, which makes it the most expensive TLB flush operation. After it completes, all modifications to the page tables will be visible globally. This is required after the kernel page tables, which are global in nature, have been modified, such as after vfree() (see Chapter 7) completes or after the PKMap is flushed (see Chapter 9).

`void flush_tlb_mm(struct mm_struct *mm)`
This flushes all TLB entries related to the userspace portion (i.e., below PAGE_OFFSET) for the

requested mm context. In some architectures, such as MIPS, this will need to be performed for all processors, but usually it is confined to the local processor. This is only called when an operation has been performed that affects the entire address space, such as after all the address mapping has been duplicated with dup_mmap() for fork or after all memory mappings have been deleted with exit_mmap().

```
void flush_tlb_range(struct mm_struct *mm, unsigned long start, unsigned long end)
```

As the name indicates, this flushes all entries within the requested user space range for the mm context. This is used after a new region has been moved or changed as during mremap(), which moves regions, or mprotect(), which changes the permissions. The function is also indirectly used during unmapping a region with munmap(), which calls tlb_finish_mmu(), which tries to use flush_tlb_range() intelligently. This API is provided for architectures that can remove ranges of TLB entries quickly rather than iterating with flush_tlb_page().

Table 3.3. Translation Lookaside Buffer Flush API (cont.)

```
void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)
```

Predictably, this API is responsible for flushing a single page from the TLB. The two most common uses of it are for flushing the TLB after a page has been faulted in or has been paged out.

```
void flush_tlb_pgtables(struct mm_struct *mm, unsigned long start, unsigned long end)
```

This API is called when the page tables are being torn down and freed. Some platforms cache the lowest level of the page table, i.e., the actual page frame storing entries, which needs to be flushed when the pages are being deleted. This is called when a region is being unmapped and the page directory entries are being reclaimed.

```
void update_mmu_cache(struct vm_area_struct *vma, unsigned long addr, pte_t pte)
```

This API is only called after a page fault completes. It tells the architecture-dependent code that a new translation now exists at pte for the virtual address addr. Each architecture decides how this information should be used. For example, Sparc64 uses the information to decide if the local CPU needs to flush its data cache or does it need to send an Inter Processor Interrupt (IPI) to a remote processor.

<<

From arch/x86/include/asm/tlbflush.h :

…

```
/*
 * TLB flushing:
 *
 *  - flush_tlb() flushes the current mm struct TLBs
 *  - flush_tlb_all() flushes all processes TLBs
 *  - flush_tlb_mm(mm) flushes the specified mm context TLB's
 *  - flush_tlb_page(vma, vmaddr) flushes one page
```

```
 *  - flush_tlb_range(vma, start, end) flushes a range of pages
 *  - flush_tlb_kernel_range(start, end) flushes a range of kernel pages
 *  - flush_tlb_others(cpumask, mm, va) flushes TLBs on other cpus
 *
…

>>
```

Additional Resource

Memory Management: **Paging**, by Paul Krzyzanowski
Look up the *Case Study / ARMv7-A MMU Architecture / TLB* section.

3.9
Level 1 CPU Cache Management

Because Linux manages the CPU cache in a very similar fashion to the TLB, this section covers how Linux uses and manages the CPU cache. CPU caches, like TLB caches, take advantage of the fact that programs tend to exhibit a locality of reference [Sea00] [CS98].

To avoid having to fetch data from main memory for each reference, the CPU will instead cache very small amounts of data in the CPU cache. Frequently, there are two levels called the Level 1 and Level 2 CPU caches.

The Level 2 CPU caches are larger, but slower than the L1 cache, but Linux only concerns itself with the Level 1 or L1 cache.

CPU caches are organized into lines. Each line is typically quite small, usually 32 bytes, and each line is aligned to its boundary size. In other words, a cache line of 32 bytes will be aligned on a 32-byte address. With Linux, the size of the line is L1_CACHE_BYTES, which is defined by each architecture.

<<
On **Intel** processors (checked on both 32 and 64 bit), a cache line is typically 64 bytes.

From arch/x86/include/asm/cache.h :

```
/* L1 cache line size */
#define L1_CACHE_SHIFT  (CONFIG_X86_L1_CACHE_SHIFT)
#define L1_CACHE_BYTES  (1 << L1_CACHE_SHIFT)
```

From kernel configuration file:
CONFIG_X86_L1_CACHE_SHIFT=6

and 1 << 6 = 64.

On the ARM:

From arch/arm/asm/cache.h :

```
#define L1_CACHE_SHIFT      CONFIG_ARM_L1_CACHE_SHIFT
#define L1_CACHE_BYTES      (1 << L1_CACHE_SHIFT)
```

From kernel configuration file:
```
#define CONFIG_ARM_L1_CACHE_SHIFT 5
```

1 << 5 = 32

thus, cache line size by default on ARM is 32 bytes.

>>

<<
Alternate way of getting Cache statistics:

a) using getconf :
(output below on an Intel core i7 processor running a 32-bit Linux OS)

```
# getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                    8
LEVEL1_ICACHE_LINESIZE                64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                    8
LEVEL1_DCACHE_LINESIZE                64
LEVEL2_CACHE_SIZE                 262144
LEVEL2_CACHE_ASSOC                     8
LEVEL2_CACHE_LINESIZE                 64
LEVEL3_CACHE_SIZE                4194304
LEVEL3_CACHE_ASSOC                    16
LEVEL3_CACHE_LINESIZE                 64
LEVEL4_CACHE_SIZE                      0
LEVEL4_CACHE_ASSOC                     0
LEVEL4_CACHE_LINESIZE                  0
#
```

b) Via the 'lscpu' utility:

```
# lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian

[...]

L1d cache:             32K
L1i cache:             32K
```

```
L2 cache:               256K
L3 cache:               4096K
#
```

c) Via a system call (eg):
sysconf(_SC_LEVEL1_DCACHE_LINESIZE)

d) Via sysfs
/sys/devices/system/cpu/cpu?/cache/* << *at least on x86* >>
>>

How addresses are mapped to cache lines vary between architectures, but the mappings come under three headings, direct mapping, associative mapping and set associative mapping.

Direct mapping is the simplest approach where each block of memory maps to only one possible cache line. With associative mapping, any block of memory can map to any cache line. Set associative mapping is a hybrid approach where any block of memory can map to any line, but only within a subset of the available lines.

Regardless of the mapping scheme, they each have one thing in common. Addresses that are close together and aligned to the cache size are likely to use different lines. Hence Linux employs simple tricks to try and maximize cache use:

- Frequently accessed structure fields are at the start of the structure to increase the chance that only one line is needed to address the common fields.
- Unrelated items in a structure should try to be at least cache-size bytes in part to avoid false sharing between CPUs.
- Objects in the general caches, such as the mm_struct cache, are aligned to the L1 CPU cache to avoid false sharing.

If the CPU references an address that is not in the cache, a *cache miss* occurs, and the data is fetched from main memory. The cost of cache misses is quite high because a reference to a cache can typically be performed in less than 10ns where a reference to main memory typically will cost between 100ns and 200ns. The basic objective is then to have as many cache hits and as few cache misses as possible.

Just as some architectures do not automatically manage their TLBs, some do not automatically manage their CPU caches.

<<
Intel processors do auto-manage their CPU caches.

From arch/x86/include/asm/cacheflush.h :

```
#ifndef _ASM_X86_CACHEFLUSH_H
#define _ASM_X86_CACHEFLUSH_H

/* Keep includes the same across arches.  */
#include <linux/mm.h>
```

```
/* Caches aren't brain-dead on the intel. */
static inline void flush_cache_all(void) { }
static inline void flush_cache_mm(struct mm_struct *mm) { }
static inline void flush_cache_dup_mm(struct mm_struct *mm) { }
static inline void flush_cache_range(struct vm_area_struct *vma,
                     unsigned long start, unsigned long end) { }
static inline void flush_cache_page(struct vm_area_struct *vma,
                    unsigned long vmaddr, unsigned long pfn) { }
#define ARCH_IMPLEMENTS_FLUSH_DCACHE_PAGE 0
static inline void flush_dcache_page(struct page *page) { }
static inline void flush_dcache_mmap_lock(struct address_space *mapping)
{ }
static inline void flush_dcache_mmap_unlock(struct address_space
*mapping) { }
static inline void flush_icache_range(unsigned long start,
                       unsigned long end) { }
static inline void flush_icache_page(struct vm_area_struct *vma,
                     struct page *page) { }
static inline void flush_icache_user_range(struct vm_area_struct *vma,
                        struct page *page,
                        unsigned long addr,
                        unsigned long len) { }
static inline void flush_cache_vmap(unsigned long start, unsigned long
end) { }
static inline void flush_cache_vunmap(unsigned long start,
                       unsigned long end) { }

static inline void copy_to_user_page(struct vm_area_struct *vma,
                     struct page *page, unsigned long vaddr,
                     void *dst, const void *src,
                     unsigned long len)
{
    memcpy(dst, src, len);
}

static inline void copy_from_user_page(
                        struct vm_area_struct *vma,
                        struct page *page, unsigned long vaddr,
                        void *dst, const void *src,
                        unsigned long len)
{
    memcpy(dst, src, len);
}

…
```

>>.
<<
ARM: see *arch/arm/include/asm/cacheflush.h*
>>

The hooks are placed in locations where the virtual to physical mapping changes, such as during a

page table update.

The CPU cache flushes should always take place first because some CPUs require a virtual to physical mapping to exist when the virtual address is being flushed from the cache. The three operations that require proper ordering are important and are listed in Table 3.4.

Flushing Full MM
flush_cache_mm()
Change all page tables
flush_tlb_mm()

Flushing Range
flush_cache_range()
Change page table range
flush_tlb_range()

Flushing Page
flush_cache_page()
Change single PTE
flush_tlb_page()

Table 3.4. Cache and TLB Flush Ordering

The API used for flushing the caches is declared in <asm/pgtable.h> and is listed in Table 3.5. In many respects, it is very similar to the TLB flushing API.

…

Caches

12.8 Summary

<< Source: "ARM System Developer's Guide – Designing and Optimizing System Software" by Sloss, Symes and Wright, published by Elsevier. >>

A cache is a small, fast array of memory placed between the processor and main memory. It is a holding buffer that stores portions of recently referenced system memory. The processor uses cache memory in preference to system memory whenever possible to increase average system performance.

A write buffer is a very small FIFO memory placed between the processor core and main memory, which helps free the processor core and cache memory from the slow write time associated with writing to main memory.

The principle of locality of reference states that computer software programs frequently

run small loops of code that repeatedly operate on local sections of data memory and explains why the average system performance increases significantly when using a cached processor core.

There are many terms used by the ARM community to describe features of cache architecture. As a convenience we have created Table 12.14, which lists the features of all current ARM cached cores.

The cache line is a fundamental component in a cache and contains three parts: a directory store, a data section, and status information. The cache-tag is a directory entry indicating where a cache line was loaded from main memory. There are two common status bits within the cache: the valid bit and the dirty bit. The valid bit is set when the associated cache line contains active memory. The dirty bit is active when the cache is using a writeback policy and new data has been written to cache memory.

The placement of a cache before or after the MMU is either physical or logical. A logical cache is placed between the processor core and the MMU, and references code and data in a virtual address space. A physical cache is placed between the MMU and main memory, and references code and data memory using physical addresses.

Table 12.14 ARM cached core features.

Core	Cache type	Cache size (kilobytes)	Cache line size (words)	Associativity	Location	Cache lockdown support	Write buffer size (words)
ARM720T	unified	8	4	4-way	logical	no	8
ARM740T	unified	4 or 8	4	4-way		yes 1/4	8
ARM920T	split	16/16 D + I	8	64-way	logical	yes 1/64	16
ARM922T	split	8/8 D + I	8	64-way	logical	yes 1/64	16
ARM940T	split	4/4 D + I	4	64-way		yes 1/64	8
ARM926EJ-S	split	4–128/4–128 D + I	8	4-way	logical	yes 1/4	16
ARM946E-S	split	4–128/4–128 D + I	4	4-way		yes 1/4	4
ARM1022E	split	16/16 D + I	8	64-way	logical	yes 1/64	16
ARM1026EJ-S	split	4–128/4–128 D + I	8	4-way	logical	yes 1/4	8
Intel StrongARM	split	16/16 D + I	4	32-way	logical	no	32
Intel XScale	split	32/32 D + I	8	32-way	logical	yes 1/32	32
		2 D	8	2-way	logical	no	

A direct-mapped cache is a very simple cache architecture where there is a single location in cache for a given main memory location. A direct-mapped cache is subject to thrashing.

To reduce thrashing, a cache is divided into smaller equal units called ways. The use of ways provides multiple storage locations in cache for a single main memory address. These caches are known as set associative caches.

The core bus architecture helps determine the design of a cached system. A Von

Neumann architecture uses a unified cache to store code and data. A Harvard architecture uses a split cache: it has one cache for instructions and a separate cache for data.

The cache replacement policy determines which cache line is selected for replacement on a cache miss. The configured policy defines the algorithm a cache controller uses to select a cache line from the available set in cache memory. The cache line selected for replacement is a victim. The two replacement policies available in ARM cached cores are pseudorandom and round-robin.

There are two policies available when writing data to cache memory. If the controller only updates cache memory, it is a writeback policy. If the cache controller writes to both the cache and main memory, it is a writethrough policy.

There are two policies a cache controller uses to allocate a cache line on a cache miss. A read-allocate policy allocates a cache line when data is read from main memory. A write-allocate policy allocates a cache line on a write to main memory.

ARM uses the term clean to mean forcing a copyback of data in the D-cache to main memory. ARM uses the term flush to mean invalidating the contents of a cache.

Cache lockdown is a feature provided by some ARM cores. The lockdown feature allows code and data to be loaded into cache and marked as exempt from eviction.

We also provided example code showing how to clean and flush ARM cached cores, and to lock code and data in cache.

Appendix E : Magic SysRq 'm' - the show_mem Functionality

The output of the above SysRq 'm' is very verbose and interesting to interpret!

Eg.: On a x86_64 8-core system running Ubuntu 14.10 LTS:

```
# cat /etc/issue
Ubuntu 14.04.1 LTS \n \l

# cat /proc/version
Linux version 3.13.0-32-generic (buildd@kissel) (gcc version 4.8.2
(Ubuntu 4.8.2-19ubuntu1) ) #57-Ubuntu SMP Tue Jul 15 03:51:08 UTC 2014
# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 30
Stepping:              5
CPU MHz:               1599.000
BogoMIPS:              3457.87
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
# echo m > /proc/sysrq-trigger
# dmesg
...
[56876.060190] SysRq : Show Memory
[56876.060200] Mem-Info:
[56876.060205] Node 0 DMA per-cpu:
[56876.060211] CPU    0: hi:    0, btch:   1 usd:   0
[56876.060215] CPU    1: hi:    0, btch:   1 usd:   0
[56876.060218] CPU    2: hi:    0, btch:   1 usd:   0
[56876.060222] CPU    3: hi:    0, btch:   1 usd:   0
[56876.060225] CPU    4: hi:    0, btch:   1 usd:   0
[56876.060229] CPU    5: hi:    0, btch:   1 usd:   0
[56876.060232] CPU    6: hi:    0, btch:   1 usd:   0
[56876.060235] CPU    7: hi:    0, btch:   1 usd:   0
[56876.060238] Node 0 DMA32 per-cpu:
[56876.060243] CPU    0: hi:  186, btch:  31 usd: 158
[56876.060247] CPU    1: hi:  186, btch:  31 usd:  66
[56876.060250] CPU    2: hi:  186, btch:  31 usd:  44
```

```
[56876.060254] CPU    3: hi:  186, btch:  31 usd: 133
[56876.060257] CPU    4: hi:  186, btch:  31 usd:  77
[56876.060260] CPU    5: hi:  186, btch:  31 usd:  83
[56876.060264] CPU    6: hi:  186, btch:  31 usd: 168
[56876.060267] CPU    7: hi:  186, btch:  31 usd:  98
[56876.060270] Node 0 Normal per-cpu:
[56876.060274] CPU    0: hi:  186, btch:  31 usd: 139
[56876.060278] CPU    1: hi:  186, btch:  31 usd:  38
[56876.060281] CPU    2: hi:  186, btch:  31 usd:  85
[56876.060284] CPU    3: hi:  186, btch:  31 usd:  85
[56876.060288] CPU    4: hi:  186, btch:  31 usd:  76
[56876.060291] CPU    5: hi:  186, btch:  31 usd:  56
[56876.060295] CPU    6: hi:  186, btch:  31 usd: 162
[56876.060298] CPU    7: hi:  186, btch:  31 usd:  80
[56876.060307] active_anon:526921 inactive_anon:213206 isolated_anon:0
[56876.060307]  active_file:44319 inactive_file:50606 isolated_file:0
[56876.060307]  unevictable:20 dirty:15 writeback:0 unstable:0
[56876.060307]  free:44561 slab_reclaimable:39232
slab_unreclaimable:13848
[56876.060307]  mapped:60302 shmem:18216 pagetables:16587 bounce:0
[56876.060307]  free_cma:0
[56876.060315] Node 0 DMA free:15752kB min:268kB low:332kB high:400kB
active_anon:24kB inactive_anon:24kB active_file:28kB inactive_file:12kB
unevictable:0kB isolated(anon):0kB isolated(file):0kB present:15976kB
managed:15892kB mlocked:0kB dirty:0kB writeback:0kB mapped:36kB shmem:0kB
slab_reclaimable:0kB slab_unreclaimable:52kB kernel_stack:0kB
pagetables:0kB unstable:0kB bounce:0kB free_cma:0kB writeback_tmp:0kB
pages_scanned:1 all_unreclaimable? no
[56876.060327] lowmem_reserve[]: 0 2901 3859 3859
[56876.060334] Node 0 DMA32 free:113512kB min:50596kB low:63244kB
high:75892kB active_anon:1804208kB inactive_anon:489856kB
active_file:117544kB inactive_file:146112kB unevictable:80kB
isolated(anon):0kB isolated(file):0kB present:3052468kB managed:2973504kB
mlocked:80kB dirty:48kB writeback:0kB mapped:182244kB shmem:53904kB
slab_reclaimable:110828kB slab_unreclaimable:34892kB kernel_stack:4432kB
pagetables:44752kB unstable:0kB bounce:0kB free_cma:0kB writeback_tmp:0kB
pages_scanned:0 all_unreclaimable? no
[56876.060346] lowmem_reserve[]: 0 0 958 958
[56876.060353] Node 0 Normal free:48980kB min:16712kB low:20888kB
high:25068kB active_anon:303452kB inactive_anon:362944kB
active_file:59704kB inactive_file:56300kB unevictable:0kB
isolated(anon):0kB isolated(file):0kB present:1048576kB managed:981344kB
mlocked:0kB dirty:12kB writeback:0kB mapped:58928kB shmem:18960kB
slab_reclaimable:46100kB slab_unreclaimable:20448kB kernel_stack:2888kB
pagetables:21596kB unstable:0kB bounce:0kB free_cma:0kB writeback_tmp:0kB
pages_scanned:0 all_unreclaimable? no
[56876.060364] lowmem_reserve[]: 0 0 0 0
[56876.060370] Node 0 DMA: 6*4kB (M) 2*8kB (M) 2*16kB (U) 2*32kB (U)
0*64kB 2*128kB (UM) 2*256kB (UM) 1*512kB (M) 2*1024kB (UM) 2*2048kB (MR)
2*4096kB (EM) = 15752kB
[56876.060399] Node 0 DMA32: 456*4kB (UEM) 3709*8kB (UEM) 666*16kB (UEM)
643*32kB (UEM) 336*64kB (UEM) 137*128kB (EM) 26*256kB (EM) 10*512kB (EM)
0*1024kB 0*2048kB 0*4096kB = 113544kB
[56876.060425] Node 0 Normal: 49*4kB (UER) 2402*8kB (UEM) 436*16kB (UEM)
```

```
282*32kB (UEM) 90*64kB (UEM) 27*128kB (UEM) 7*256kB (M) 3*512kB (M)
1*1024kB (M) 0*2048kB 0*4096kB = 48980kB
[56876.060454] Node 0 hugepages_total=0 hugepages_free=0 hugepages_surp=0
hugepages_size=2048kB
[56876.060457] 149132 total pagecache pages
[56876.060460] 35991 pages in swap cache
[56876.060463] Swap cache stats: add 630224, delete 594233, find
139788/161683
[56876.060466] Free swap  = 2737600kB
[56876.060469] Total swap = 4109308kB
[56876.060471] 1029255 pages RAM
[56876.060473] 0 pages HighMem/MovableOnly
[56876.060476] 16808 pages reserved
#
```

The code that generates this output is here – the "Magic" SysRq area:
[All code below is from kernel ver 3.10.24].

show_mem()

File : [kernel ver 3.10.24] drivers/tty/sysrq.c

```
...
 311 static void sysrq_handle_showmem(int key)
 312 {
 313     show_mem(0);
 314 }
 315 static struct sysrq_key_op sysrq_showmem_op = {
 316     .handler    = sysrq_handle_showmem,
 317     .help_msg   = "show-memory-usage(m)",
 318     .action_msg = "Show Memory",
 319     .enable_mask    = SYSRQ_ENABLE_DUMP,
 320 };
...
```

File : [kernel ver 3.10.24] : lib/show_mem.c

```
...
 12 void show_mem(unsigned int filter)
 13 {
 14     pg_data_t *pgdat;
 15     unsigned long total = 0, reserved = 0, shared = 0,
 16         nonshared = 0, highmem = 0;
 17
 18     printk("Mem-Info:\n");
<<
The output corresponding to the above code is shown in blue color below:
[56876.060200] Mem-Info:
>>
 19     show_free_areas(filter);    << see below >>
...
 24     for_each_online_pgdat(pgdat) {
 25         unsigned long i, flags;
...
<< lib/show_mem.c output continues after the show_free_areas function
```

below. >>

show_free_areas()

File : mm/page_alloc.c

```
...
2974 /*
2975  * Show free area list (used inside shift_scroll-lock stuff)
2976  * We also calculate the percentage fragmentation. We do this by
counting the
2977  * memory on each free list with the exception of the first item on
the list.
2978  * Suppresses nodes that are not allowed by current's cpuset if
2979  * SHOW_MEM_FILTER_NODES is passed.
2980  */
2981 void show_free_areas(unsigned int filter)
2982 {
2983     int cpu;
2984     struct zone *zone;
2985
2986     for_each_populated_zone(zone) {
2987         if (skip_free_areas_node(filter, zone_to_nid(zone)))
2988             continue;
2989         show_node(zone);
<<
[56876.060205] Node 0
>>
2990         printk("%s per-cpu:\n", zone->name);
<<
DMA per-cpu:
>>
2991
2992         for_each_online_cpu(cpu) {
2993             struct per_cpu_pageset *pageset;
2994
2995             pageset = per_cpu_ptr(zone->pageset, cpu);
2996
2997             printk("CPU %4d: hi:%5d, btch:%4d usd:%4d\n",
2998                    cpu, pageset->pcp.high,
2999                    pageset->pcp.batch, pageset->pcp.count);
<<
...
[56876.060267] CPU    7: hi:  186, btch:  31 usd:  98
[56876.060270] Node 0 Normal per-cpu:
[56876.060274] CPU    0: hi:  186, btch:  31 usd: 139
[56876.060278] CPU    1: hi:  186, btch:  31 usd:  38
[56876.060281] CPU    2: hi:  186, btch:  31 usd:  85
[56876.060284] CPU    3: hi:  186, btch:  31 usd:  85
[56876.060288] CPU    4: hi:  186, btch:  31 usd:  76
[56876.060291] CPU    5: hi:  186, btch:  31 usd: 150
[56876.060295] CPU    6: hi:  186, btch:  31 usd: 162
```

high watermark, emptying chunk size if needed for buddy add/remove

```
[56876.060298] CPU    7: hi:  186, btch:  31 usd:  80
...
>>
3000          }
3001      }
3002
3003      printk("active_anon:%lu inactive_anon:%lu isolated_anon:%lu\n"
3004          " active_file:%lu inactive_file:%lu isolated_file:%lu\n"
3005          " unevictable:%lu"
3006          " dirty:%lu writeback:%lu unstable:%lu\n"
3007          " free:%lu slab_reclaimable:%lu slab_unreclaimable:%lu\n"
3008          " mapped:%lu shmem:%lu pagetables:%lu bounce:%lu\n"
3009          " free_cma:%lu\n",
3010          global_page_state(NR_ACTIVE_ANON),
3011          global_page_state(NR_INACTIVE_ANON),
3012          global_page_state(NR_ISOLATED_ANON),
3013          global_page_state(NR_ACTIVE_FILE),
3014          global_page_state(NR_INACTIVE_FILE),
3015          global_page_state(NR_ISOLATED_FILE),
3016          global_page_state(NR_UNEVICTABLE),
3017          global_page_state(NR_FILE_DIRTY),
3018          global_page_state(NR_WRITEBACK),
3019          global_page_state(NR_UNSTABLE_NFS),
3020          global_page_state(NR_FREE_PAGES),
3021          global_page_state(NR_SLAB_RECLAIMABLE),
3022          global_page_state(NR_SLAB_UNRECLAIMABLE),
3023          global_page_state(NR_FILE_MAPPED),
3024          global_page_state(NR_SHMEM),
3025          global_page_state(NR_PAGETABLE),
3026          global_page_state(NR_BOUNCE),
3027          global_page_state(NR_FREE_CMA_PAGES));
<<
[56876.060307] active_anon:526921 inactive_anon:213206 isolated_anon:0
[56876.060307]  active_file:44319 inactive_file:50606 isolated_file:0
[56876.060307]  unevictable:20 dirty:15 writeback:0 unstable:0
[56876.060307]  free:44561 slab_reclaimable:39232
slab_unreclaimable:13848
[56876.060307]  mapped:60302 shmem:18216 pagetables:16587 bounce:0
[56876.060307]  free_cma:0
>>
<<
The above NR_foo_bar items are captured in the global (atomic) vm_stat
array

include/linux/vmstat.h :
93 extern atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];

and are enumerated here:
     include/linux/mmzone.h : enum zone_stat_item .
(Some are commented as well).
>>
3028
3029      for_each_populated_zone(zone) {    << for each zone in this
[N]UMA node ... >>
```

number of pages in the list

```
3030         int i;
3031
3032         if (skip_free_areas_node(filter, zone_to_nid(zone)))
3033             continue;
3034         show_node(zone);
<<
2887 static inline void show_node(struct zone *zone)
2888 {
2889     if (IS_ENABLED(CONFIG_NUMA))
2890         printk("Node %d ", zone_to_nid(zone));
2891 }
>>
3035         printk("%s"
3036             " free:%lukB"
3037             " min:%lukB"
3038             " low:%lukB"
3039             " high:%lukB"
3040             " active_anon:%lukB"
3041             " inactive_anon:%lukB"
3042             " active_file:%lukB"
3043             " inactive_file:%lukB"
3044             " unevictable:%lukB"
3045             " isolated(anon):%lukB"
3046             " isolated(file):%lukB"
3047             " present:%lukB"
3048             " managed:%lukB"
3049             " mlocked:%lukB"
3050             " dirty:%lukB"
3051             " writeback:%lukB"
3052             " mapped:%lukB"
3053             " shmem:%lukB"
3054             " slab_reclaimable:%lukB"
3055             " slab_unreclaimable:%lukB"
3056             " kernel_stack:%lukB"
3057             " pagetables:%lukB"
3058             " unstable:%lukB"
3059             " bounce:%lukB"
3060             " free_cma:%lukB"
3061             " writeback_tmp:%lukB"
3062             " pages_scanned:%lu"
3063             " all_unreclaimable? %s"
3064             "\n",
3065             zone->name,
3066             K(zone_page_state(zone, NR_FREE_PAGES)),
3067             K(min_wmark_pages(zone)),
3068             K(low_wmark_pages(zone)),
3069             K(high_wmark_pages(zone)),
3070             K(zone_page_state(zone, NR_ACTIVE_ANON)),
3071             K(zone_page_state(zone, NR_INACTIVE_ANON)),
3072             K(zone_page_state(zone, NR_ACTIVE_FILE)),
3073             K(zone_page_state(zone, NR_INACTIVE_FILE)),
3074             K(zone_page_state(zone, NR_UNEVICTABLE)),
3075             K(zone_page_state(zone, NR_ISOLATED_ANON)),
3076             K(zone_page_state(zone, NR_ISOLATED_FILE)),
```

```
3077             K(zone->present_pages),
3078             K(zone->managed_pages),
3079             K(zone_page_state(zone, NR_MLOCK)),
3080             K(zone_page_state(zone, NR_FILE_DIRTY)),
3081             K(zone_page_state(zone, NR_WRITEBACK)),
3082             K(zone_page_state(zone, NR_FILE_MAPPED)),
3083             K(zone_page_state(zone, NR_SHMEM)),
3084             K(zone_page_state(zone, NR_SLAB_RECLAIMABLE)),
3085             K(zone_page_state(zone, NR_SLAB_UNRECLAIMABLE)),
3086             zone_page_state(zone, NR_KERNEL_STACK) *
3087                 THREAD_SIZE / 1024,
3088             K(zone_page_state(zone, NR_PAGETABLE)),
3089             K(zone_page_state(zone, NR_UNSTABLE_NFS)),
3090             K(zone_page_state(zone, NR_BOUNCE)),
3091             K(zone_page_state(zone, NR_FREE_CMA_PAGES)),
3092             K(zone_page_state(zone, NR_WRITEBACK_TEMP)),
3093             zone->pages_scanned,
3094             (zone->all_unreclaimable ? "yes" : "no")
3095             );
<<
...
...     [ we'll show the output for ZONE_NORMAL here as an example... ]
...
Normal free:48980kB min:16712kB low:20888kB high:25068kB
active_anon:303452kB inactive_anon:362944kB active_file:59704kB
inactive_file:56300kB unevictable:0kB isolated(anon):0kB
isolated(file):0kB present:1048576kB managed:981344kB mlocked:0kB
dirty:12kB writeback:0kB mapped:58928kB shmem:18960kB
slab_reclaimable:46100kB slab_unreclaimable:20448kB kernel_stack:2888kB
pagetables:21596kB unstable:0kB bounce:0kB free_cma:0kB writeback_tmp:0kB
pages_scanned:0 all_unreclaimable? no
>>
3096         printk("lowmem_reserve[]:");
3097         for (i = 0; i < MAX_NR_ZONES; i++)
3098             printk(" %lu", zone->lowmem_reserve[i]);
3099         printk("\n");
3100     }
<<
lowmem_reserve[]: 0 0 0 0
>>
3101
3102     for_each_populated_zone(zone) {   << for each zone in this
[N]UMA node ... >>
3103         unsigned long nr[MAX_ORDER], flags, order, total = 0;
3104         unsigned char types[MAX_ORDER];
3105
3106         if (skip_free_areas_node(filter, zone_to_nid(zone)))
3107             continue;
3108         show_node(zone);
3109         printk("%s: ", zone->name);
<<
...
...     [ we'll show the output for ZONE_NORMAL here as an example... ]
...
```

```
Normal
>>
3110
3111          spin_lock_irqsave(&zone->lock, flags);
3112          for (order = 0; order < MAX_ORDER; order++) {   << for each
order of each
                                                              zone's buddy system
freelist... >>
3113              struct free_area *area = &zone->free_area[order];
3114              int type;
3115
3116              nr[order] = area->nr_free;
3117              total += nr[order] << order;
3118
3119              types[order] = 0;
3120              for (type = 0; type < MIGRATE_TYPES; type++) {
3121                  if (!list_empty(&area->free_list[type]))
3122                      types[order] |= 1 << type;
3123              }
3124          }
3125          spin_unlock_irqrestore(&zone->lock, flags);
3126          for (order = 0; order < MAX_ORDER; order++) {
3127              printk("%lu*%lukB ", nr[order], K(1UL) << order);
3128              if (nr[order])
3129                  show_migration_types(types[order]);
3130          }
<<
 49*4kB (UER) 2402*8kB (UEM) 436*16kB (UEM) 282*32kB (UEM) 90*64kB (UEM)
27*128kB (UEM) 7*256kB (M) 3*512kB (M) 1*1024kB (M) 0*2048kB 0*4096kB
>>
<<
Migration Types:
2947 static void show_migration_types(unsigned char type)
2948 {
2949     static const char types[MIGRATE_TYPES] = {
2950          [MIGRATE_UNMOVABLE] = 'U',
2951          [MIGRATE_RECLAIMABLE]   = 'E',
2952          [MIGRATE_MOVABLE]   = 'M',
2953          [MIGRATE_RESERVE]   = 'R',
2954 #ifdef CONFIG_CMA
2955          [MIGRATE_CMA]       = 'C',
2956 #endif
2957 #ifdef CONFIG_MEMORY_ISOLATION
2958          [MIGRATE_ISOLATE]   = 'I',
2959 #endif
2960     };
...
[
 File : [3.10.24 ] : include/linux/mmzone.h
  38 enum {
  39     MIGRATE_UNMOVABLE,
  40     MIGRATE_RECLAIMABLE,
  41     MIGRATE_MOVABLE,
  42     MIGRATE_PCPTYPES,   /* the number of types on the pcp lists */
```

```
  43      MIGRATE_RESERVE = MIGRATE_PCPTYPES,
  44  #ifdef CONFIG_CMA
  45      /*
  46       * MIGRATE_CMA migration type is designed to mimic the way
  47       * ZONE_MOVABLE works.
...
]
>>
3131          printk("= %lukB\n", K(total));
<<
= 48980kB
>>
3132      }
3133
3134      hugetlb_show_meminfo();
<<
...
Node 0 hugepages_total=0 hugepages_free=0 hugepages_surp=0
hugepages_size=2048kB
>>
3135
3136      printk("%ld total pagecache pages\n",
global_page_state(NR_FILE_PAGES));
  149132 total pagecache pages
>>
3137
3138      show_swap_cache_info();
<<
35991 pages in swap cache
Swap cache stats: add 630224, delete 594233, find 139788/161683
Free swap  = 2737600kB
Total swap = 4109308kB
>>
3139 }
<< We return back to the caller function now, lib/show_mem() ;
See below...
>>

...
File : [kernel ver 3.10.24 ] : lib/show_mem.c
...
 12 void show_mem(unsigned int filter)
 13 {
 14      pg_data_t *pgdat;
 15      unsigned long total = 0, reserved = 0, shared = 0,
 16          nonshared = 0, highmem = 0;
 17
 18      printk("Mem-Info:\n");
 19      show_free_areas(filter); << we just saw this (above) >>
 20
 21      if (filter & SHOW_MEM_FILTER_PAGE_COUNT)
 22          return;
 23
 24      for_each_online_pgdat(pgdat) {    << for each online [N]UMA node
```

```
>>
 25         unsigned long i, flags;
 26
 27         pgdat_resize_lock(pgdat, &flags);
 28         for (i = 0; i < pgdat->node_spanned_pages; i++) { << for
each physical page in this node; node_spanned_pages = total size of
physical page range, including holes  >>
 29             struct page *page;
 30             unsigned long pfn = pgdat->node_start_pfn + i; << pfn:
phy frame num >>
 31
 32             if (unlikely(!(i % MAX_ORDER_NR_PAGES)))
 33                 touch_nmi_watchdog();
 34
 35             if (!pfn_valid(pfn))
 36                 continue;
 37
 38             page = pfn_to_page(pfn);
<<
Get logical (virtual) page from phy:
File: include/asm-generic/memory_model.h
...
 30 #define __pfn_to_page(pfn)  (mem_map + ((pfn) - ARCH_PFN_OFFSET))
...
 73 #define pfn_to_page __pfn_to_page
...
[ File: include/linux/mmzone.h
 684 /* The array of struct pages - for discontigmem use pgdat->lmem_map
*/
 685 extern struct page *mem_map;
]
>>
 39                                      << Cumulatively total up stats
>>
 40             if (PageHighMem(page))
 41                 highmem++;
 42
 43             if (PageReserved(page))
 44                 reserved++;
 45             else if (page_count(page) == 1)
 46                 nonshared++;
 47             else if (page_count(page) > 1)
 48                 shared += page_count(page) - 1;
 49
 50             total++;
 51         }
 52         pgdat_resize_unlock(pgdat, &flags);
 53     }
 54
 55     printk("%lu pages RAM\n", total);
 56 #ifdef CONFIG_HIGHMEM
 57     printk("%lu pages HighMem\n", highmem);
 58 #endif
 59     printk("%lu pages reserved\n", reserved);
```

```
 60     printk("%lu pages shared\n", shared);
 61     printk("%lu pages non-shared\n", nonshared);
 62 #ifdef CONFIG_QUICKLIST
 63     printk("%lu pages in pagetable cache\n",
 64         quicklist_total_size());
 65 #endif
<<
1029255 pages RAM                  [ = 1029255 * 4 / 1024 MB ~= 4020 MB
~= 4 GB ]
0 pages HighMem/MovableOnly
16808 pages reserved
>>
 66 }

<<
The output is slightly different from the code above (wrt the
"Highmem/MovableOnly" output)?? Yes, because on 3.13 (the kernel where
this output was captured, the tail end of the show_mem() function is
slightly different from 3.10.24 :
...
 42         printk("%lu pages RAM\n", total);
 43         printk("%lu pages HighMem/MovableOnly\n", highmem);
 44         printk("%lu pages reserved\n", reserved);
 45 #ifdef CONFIG_QUICKLIST
 46         printk("%lu pages in pagetable cache\n",
 47                 quicklist_total_size());
 48 #endif
 49 }
>>
```

Appendix F : A Brief Note on Kernel Threads

Another interesting point regards the memory descriptor of **kernel threads**. Since kernel threads run only in Kernel Mode, they never access virtual addresses below PAGE_OFFSET (0xc0000000).

Also, unlike user processes, kernel threads do **not** use memory regions (VMAs).

Since the Page Table Entries that refer to virtual addresses above PAGE_OFFSET are the *same* for all user processes, it does not really matter *which* user processes' Page Tables are referred to by the kernel when it is in the context of executing a kernel thread - any user processes' Page Tables will do; however, to avoid useless TLB and cache flushes, kernel threads use the Page Tables of the just-previously scheduled process. This is achieved by having 2 fields in the process descriptor refer to the memory descriptor - *mm* and *active_mm* ; mm refers to the memory descriptor owned by the user-process.

Kernel threads do not have a "user-space" mapping -thus their *mm* field is always NULL; however,

they require *some* mapping – a dummy one, called an “anonymous address space” by Linus, they make use of the *active_mm* field, which is set to the value of *active_mm* of the previously-scheduled process.

So, a sure way to distinguish between user and kernel threads is to look up their *mm* value : if NULL => it's a kernel thread!

Also, if you find several tasks having the same “mm” field, they are threads of the same application.

Linus's terminology: *Source: Documentation/vm/active_mm.txt*

...

“- we have "real address spaces" and "anonymous address spaces". The difference is that an anonymous address space doesn't care about the user-level page tables at all, so when we do a context switch into an anonymous address space we just leave the previous address space active.”

…

Appendix G :: 'Buffers' and 'Cached' Memory

Read this (an FAQ!):
"Linux Kernel: What is the major difference between the buffer cache and the page cache?"

"...
Starting with Linux kernel version 2.4, the contents of the two caches were unified. The VM subsystem now drives I/O and it does so out of the page cache. If cached data has both a file and a block representation—as most data does—the buffer cache will simply point into the page cache; thus only one instance of the data is cached in memory. The page cache is what you picture when you think of a disk cache: It caches file data from a disk to make subsequent I/O faster.

The buffer cache remains, however, as the kernel still needs to perform block I/O in terms of blocks, not pages. As most blocks represent file data, most of the buffer cache is represented by the page cache. But a small amount of block data isn't file backed—metadata and raw block I/O for example—and thus is solely represented by the buffer cache.
..."

Q. In the output of the 'free' utility, what do the 'buffers' and 'cached' columns really mean at the code level?

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:          3888       3742        145          0         49        494
-/+ buffers/cache:       3198        690
Swap:         8191        210       7981
$
```

A.
*1. Lets first look up what exactly '**buffers**' is:*

'free' works by using the procfs node /proc/meminfo.
/proc/meminfo is implemented in (kernel codebase):

File : *fs/proc/meminfo.c:meminfo_proc_show(...)*

```
...
static int meminfo_proc_show(struct seq_file *m, void *v)
{
   struct sysinfo i;
...
   si_meminfo(&i);         // see below
   si_swapinfo(&i);
...
   seq_printf(m,
```

```
        "MemTotal:       %8lu kB\n"
        "MemFree:        %8lu kB\n"
         "Buffers:        %8lu kB\n"
           "Cached:         %8lu kB\n"
         …
    ,
        K(i.totalram),
        K(i.freeram),
         K(i.bufferram),
            K(cached),
 ...
}
…

void si_meminfo(struct sysinfo *val)
{
   val->totalram = totalram_pages;
   val->sharedram = 0;
   val->freeram = global_page_state(NR_FREE_PAGES);
   val->bufferram = nr_blockdev_pages();
   val->totalhigh = totalhigh_pages;
   val->freehigh = nr_free_highpages();
   val->mem_unit = PAGE_SIZE;
}

...

long nr_blockdev_pages(void)
{
   struct block_device *bdev;
   long ret = 0;
   spin_lock(&bdev_lock);
   list_for_each_entry(bdev, &all_bdevs, bd_list) {
       ret += bdev->bd_inode->i_mapping->nrpages;
   }
   spin_unlock(&bdev_lock);
   return ret;
}
```

So: 'buffers' is the total global memory currently used by all block devices for managing filesystem-to-disk IO requests; these are 'mapped' to memory pages for performance. It's generally called the 'buffer cache'.

2. 'cached' is:

Similarly, in (kernel codebase): fs/proc/meminfo.c:meminfo_proc_show(...)

```
…
cached = global_page_state(NR_FILE_PAGES) -
           total_swapcache_pages - i.bufferram;
   if (cached < 0)
       cached = 0;
```

…

include/linux/vmstat.h :

```
...
static inline unsigned long global_page_state(enum zone_stat_item item)
{
   long x = atomic_long_read(&vm_stat[item]);
…
}
```

mm/vmstat.c :

```
…
/*
* Manage combined zone based / global counters
*
* vm_stat contains the global counters
*/
atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];
EXPORT_SYMBOL(vm_stat);
…
```

include/linux/mmzone.h :

```
...
enum zone_stat_item {
   /* First 128 byte cacheline (assuming 64 bit words) */
   NR_FREE_PAGES,
   ...
   NR_FILE_PAGES,
   NR_FILE_DIRTY,
  NR_WRITEBACK,
   …
};
…
```

So: 'cached' is the global memory used to cache files minus the memory pages used for the swap cache and buffer cache. It's generally called the 'page cache'.

Appendix H :: Interpretation of the */proc/meminfo* (meminfo) output

Eg. 1. On a (x86_64) Linux box having a total of 8 GB RAM (running Linux Mint 17 LTS):

```
$ uname -r
3.13.0-24-generic
$
$ free -m
             total       used       free     shared    buffers     cached
Mem:          7644       7009        635        550        375       2626
-/+ buffers/cache:       4007       3637
Swap:        16382        223      16159
$ cat /proc/meminfo
MemTotal:        7827768 kB
MemFree:          651744 kB
Buffers:          384360 kB
Cached:          2687772 kB
SwapCached:        10956 kB
Active:          4569904 kB
Inactive:        1985316 kB
Active(anon):    3131796 kB
Inactive(anon):   912260 kB
Active(file):    1438108 kB
Inactive(file):  1073056 kB
Unevictable:         232 kB
Mlocked:             232 kB
SwapTotal:      16776188 kB
SwapFree:       16547560 kB
Dirty:                12 kB
Writeback:             0 kB
AnonPages:       3474420 kB
Mapped:           359196 kB
Shmem:            560968 kB
Slab:             466560 kB
SReclaimable:     401780 kB
SUnreclaim:        64780 kB
KernelStack:        5352 kB
PageTables:        54808 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    20690072 kB
Committed_AS:    8872492 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      368956 kB
VmallocChunk:   34359360624 kB
HardwareCorrupted:     0 kB
AnonHugePages:    360448 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

```
DirectMap4k:      261172 kB
DirectMap2M:     7774208 kB
DirectMap1G:           0 kB
$
```

Eg. 2. On a (x86 IA-32) Linux box having a total of 3 GB RAM (running Ubuntu 14.10 LTS):

```
# uname -r
3.13.0-32-generic
# cat /proc/meminfo
MemTotal:        2976316 kB
MemFree:          247196 kB
Buffers:            3200 kB
Cached:           649944 kB
SwapCached:        33144 kB
Active:          1665956 kB
Inactive:         860416 kB
Active(anon):    1545724 kB
Inactive(anon):   753900 kB
Active(file):     120232 kB
Inactive(file):   106516 kB
Unevictable:          80 kB
Mlocked:              80 kB
HighTotal:       2117252 kB
HighFree:          92492 kB
LowTotal:         859064 kB
LowFree:          154704 kB
SwapTotal:       5998588 kB
SwapFree:        5245736 kB
Dirty:               604 kB
Writeback:             0 kB
AnonPages:       1860312 kB
Mapped:           240660 kB
Shmem:            426284 kB
Slab:             110956 kB
SReclaimable:      77468 kB
SUnreclaim:        33488 kB
KernelStack:        5808 kB
PageTables:        25488 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     7486744 kB
Committed_AS:    8047408 kB
VmallocTotal:     122880 kB
VmallocUsed:       21204 kB
VmallocChunk:      85980 kB
HardwareCorrupted:     0 kB
AnonHugePages:    509952 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
```

```
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:       38904 kB
DirectMap2M:      874496 kB
#
```

Source: https://www.kernel.org/doc/Documentation/filesystems/proc.txt

…
meminfo:

Provides information about distribution and utilization of memory. This
varies by architecture and compile options. The following is from a
16GB PIII, which has highmem enabled. You may not have all of these fields.
...
MemTotal: Total usable ram (i.e. physical ram minus a few reserved
bits and the kernel binary code)
MemFree: The sum of LowFree+HighFree
MemAvailable: An estimate of how much memory is available for starting new
applications, without swapping. Calculated from MemFree,
SReclaimable, the size of the file LRU lists, and the low
watermarks in each zone.
The estimate takes into account that the system needs some
page cache to function well, and that not all reclaimable
slab will be reclaimable, due to items being in use. The
impact of those factors will vary from system to system.
Buffers: Relatively temporary storage for raw disk blocks
shouldn't get tremendously large (20MB or so)
Cached: in-memory cache for files read from the disk (the
pagecache). Doesn't include SwapCached
SwapCached: Memory that once was swapped out, is swapped back in but
still also is in the swapfile (if memory is needed it
doesn't need to be swapped out AGAIN because it is already
in the swapfile. This saves I/O)
Active: Memory that has been used more recently and usually not
reclaimed unless absolutely necessary.
Inactive: Memory which has been less recently used. It is more
eligible to be reclaimed for other purposes
HighTotal:
HighFree: Highmem is all memory above ~860MB of physical memory
Highmem areas are for use by userspace programs, or
for the pagecache. The kernel must use tricks to access
this memory, making it slower to access than lowmem.
LowTotal:
LowFree: Lowmem is memory which can be used for everything that
highmem can be used for, but it is also available for the
kernel's use for its own data structures. Among many
other things, it is where everything from the Slab is
allocated. Bad things happen when you're out of lowmem.

SwapTotal: total amount of swap space available
SwapFree: Memory which has been evicted from RAM, and is temporarily
on the disk
Dirty: Memory which is waiting to get written back to the disk
Writeback: Memory which is actively being written back to the disk
AnonPages: Non-file backed pages mapped into userspace page tables
AnonHugePages: Non-file backed huge pages mapped into userspace page tables
Mapped: files which have been mmaped, such as libraries
Slab: in-kernel data structures cache
SReclaimable: Part of Slab, that might be reclaimed, such as caches
SUnreclaim: Part of Slab, that cannot be reclaimed on memory pressure
PageTables: amount of memory dedicated to the lowest level of page
tables.
NFS_Unstable: NFS pages sent to the server, but not yet committed to stable
storage
Bounce: Memory used for block device "bounce buffers"
WritebackTmp: Memory used by FUSE for temporary writeback buffers
CommitLimit: Based on the overcommit ratio ('vm.overcommit_ratio'),
this is the total amount of memory currently available to
be allocated on the system. This limit is only adhered to
if strict overcommit accounting is enabled (mode 2 in
'vm.overcommit_memory').
The CommitLimit is calculated with the following formula:
CommitLimit = ([total RAM pages] - [total huge TLB pages]) *
overcommit_ratio / 100 + [total swap pages]
For example, on a system with 1G of physical RAM and 7G
of swap with a `vm.overcommit_ratio` of 30 it would
yield a CommitLimit of 7.3G.
For more details, see the memory overcommit documentation
in vm/overcommit-accounting.
Committed_AS: The amount of memory presently allocated on the system.
The committed memory is a sum of all of the memory which
has been allocated by processes, even if it has not been
"used" by them as of yet. A process which malloc()'s 1G
of memory, but only touches 300M of it will show up as
using 1G. This 1G is memory which has been "committed" to
by the VM and can be used at any time by the allocating
application. With strict overcommit enabled on the system
(mode 2 in 'vm.overcommit_memory'),allocations which would
exceed the CommitLimit (detailed above) will not be permitted.
This is useful if one needs to guarantee that processes will
not fail due to lack of memory once that memory has been
successfully allocated.
VmallocTotal: total size of vmalloc memory area
VmallocUsed: amount of vmalloc area which is used
VmallocChunk: largest contiguous block of vmalloc area which is free.

.....

Appendix I :: Virtual to Physical Address Translation

Source : ULVMM-MG

...

3.7.1 Mapping Physical to Virtual Kernel Addresses

Linux sets up a direct mapping from the physical address 0 to the virtual address PAGE_OFFSET at 3GiB on the x86.

SIDEBAR | Using 'kdb' to see virtual and/or physical memory

After breaking into kdb:

```
[0]kdb> help
Command  Usage                 Description
----------------------------------------------------------
md       <vaddr>  Display Memory Contents, also mdWcN, e.g. m
d8c1
mdr      <vaddr>  <bytes>      Display Raw Memory
mdp      <paddr> <bytes>      Display Physical Memory
…
[0]kdb> md 0xc0000000       << view VM starting at PAGE_OFFSET >>
0xc0000000 f000ff53 f000ff53 f000e2c3 f000ff53   S...S.......S...
0xc0000010 f000ff53 f000ff53 f000ff53 f000ff53   S...S...S...S...
0xc0000020 f000fea5 f000e987 f000c659 f000c659   ........Y...Y...
0xc0000030 f000c659 f000c659 f000ef57 f000c659   Y...Y...W...Y...
0xc0000040 c00083f9 f000f84d f000f841 f000e3fe   ....M...A.......
0xc0000050 f000e739 f000f859 f000e82e f000efd2   9...Y...........
0xc0000060 f000c67e cb800018 f000fe6e f000ff53   ~.......n...S...
0xc0000070 f000ff53 f000ff53 f000ce80 c000133e   S...S.......>...
[0]kdb> mdp 0x0           << view physical memory starting at 0x0 >>
phys
0x00000000 f000ff53 f000ff53 f000e2c3 f000ff53   S...S.......S...
phys 0x00000010 f000ff53 f000ff53 f000ff53 f000ff53   S...S...S...S...
phys 0x00000020 f000fea5 f000e987 f000c659 f000c659   ........Y...Y...
phys 0x00000030 f000c659 f000c659 f000ef57 f000c659   Y...Y...W...Y...
phys 0x00000040 c00083f9 f000f84d f000f841 f000e3fe   ....M...A.......
phys 0x00000050 f000e739 f000f859 f000e82e f000efd2   9...Y...........
phys 0x00000060 f000c67e cb800018 f000fe6e f000ff53   ~.......n...S...
phys 0x00000070 f000ff53 f000ff53 f000ce80 c000133e   S...S.......>...

[0]kdb>
                                          << They're the same! >>
```

Resources:
kdb website.
kdb tutorial.

This means that any virtual address can be translated to the physical address by simply subtracting

PAGE_OFFSET, which is essentially what the function *virt_to_phys()* with the macro *pa()* does:

From arch/x86/kernel/head_32.S :

```
/* Physical address */
#define pa(X) ((X) - __PAGE_OFFSET)
```

and from *arch/x86/include/asm/page.h :*

```
#define __pa(x)     __phys_addr((unsigned long)(x))
```

From arch/x86/include/asm/page_32.h :

```
#define __phys_addr_nodebug(x)  ((x) - PAGE_OFFSET)
#ifdef CONFIG_DEBUG_VIRTUAL
extern unsigned long __phys_addr(unsigned long);
#else
#define __phys_addr(x)      __phys_addr_nodebug(x)
#endif
```

VA to PA Translation:

From arch/x86/include/asm/io.h :

```
/**
 *  virt_to_phys    -   map virtual addresses to physical
 *  @address: address to remap
 *
 *  The returned physical address is the physical (CPU) mapping
    for the memory address given. It is only valid to use this
    function on addresses directly mapped or allocated via
    kmalloc.
 *
 *  This function does not give bus mappings for DMA transfers.
    In almost all conceivable cases a device driver should not be
    using this function
 */

static inline phys_addr_t virt_to_phys(volatile void *address)
{
    return __pa(address);
}
```

Obviously, the reverse operation involves simply adding PAGE_OFFSET, which is carried out by the function phys_to_virt() with the macro va(). Next we see how this helps the mapping of struct pages to physical addresses.

There is one exception where virt_to_phys() cannot be used to convert virtual addresses to physical ones.[1] Specifically, on the PPC and ARM architectures, virt_to_phys() cannot be used to convert

addresses that have been returned by the function consistent_alloc(). consistent_alloc() is used on PPC and ARM architectures to return memory from non-cached for use with DMA.

[1] This tricky issue was pointed out to me by Jeffrey Haran.

On the ARM

From *arch/arm/include/asm/memory.h :*

```
…
/*
 * These are *only* valid on the kernel direct mapped RAM memory.
 * Note: Drivers should NOT use these.  They are the wrong
 * translation for translating DMA addresses.  Use the driver
 * DMA support - see dma-mapping.h.
 */
static inline unsigned long virt_to_phys(void *x)
{
    return __virt_to_phys((unsigned long)(x));
}

static inline void *phys_to_virt(unsigned long x)
{
    return (void *)(__phys_to_virt((unsigned long)(x)));
}

/*
 * Drivers should NOT use these either.
 */
#define __pa(x)          __virt_to_phys((unsigned long)(x))
#define __va(x)          ((void *)__phys_to_virt((unsigned long)(x)))

…

/*
 * Physical vs virtual RAM address space conversion.  These are
 * private definitions which should NOT be used outside memory.h
 * files.  Use virt_to_phys/phys_to_virt/__pa/__va instead.
 */
#ifndef __virt_to_phys
#define __virt_to_phys(x)    ((x) - PAGE_OFFSET + PHYS_OFFSET)
#define __phys_to_virt(x)    ((x) - PHYS_OFFSET + PAGE_OFFSET)
#endif

…

#ifndef PHYS_OFFSET
#define PHYS_OFFSET          UL(CONFIG_DRAM_BASE)
#endif

…
```

Also note on recent kernels:

```
CONFIG_ARM_PATCH_PHYS_VIRT:
 Patch phys-to-virt and virt-to-phys translation functions at
boot and module load time according to the position of the
kernel in system memory.
```

Appendix J :: (Some) VM Complexities

The buddy system alllocator, indeed, the entire memory management code, has in reality much complexity to deal with. Many issues exist:

- Performance
  - fragmentation
  - as few “lock” paths as possible
  - minimize free page searching by keeping similar (migration) types of pages together
  - NUMA considerations
- Large physically contiguous memory regions allocation
- etc..

These have given rise to various solutions, many of which are now deeply merged into the fabric of Linux MM; these include the notions of (and corresponding technology) :
- pagesets
- page migration types
- CMA (Contoguous Memory Allocator).

Source (below): “Professional Linux Kernel Architecture” by Wolfgang Mauerer, Wrox Press

3.5.2 Avoiding Fragmentation

In the simplified explanation given (earlier), one doubly linked list was sufficient to satisfy all the needs of the buddy system. This has, indeed, been the situation until kernel 2.6.23.

During the development of the kernel 2.6.24, the buddy system has, however, seen the integration of patches disputed among the kernel developers for an unusually long amount of time. Since the buddy system is one of the most venerable components of the kernel, changes are not accepted lightly.

Grouping Pages by Mobility

The basic principle of the buddy system, has been discussed in the preceding section. The scheme has, indeed, worked very well during the last couple of years.

However, there is one issue that has been a long-standing problem with Linux memory management: After systems have been up and **running for longer times, physical memory tends to become fragmented**. The situation is depicted in Figure 3-24.

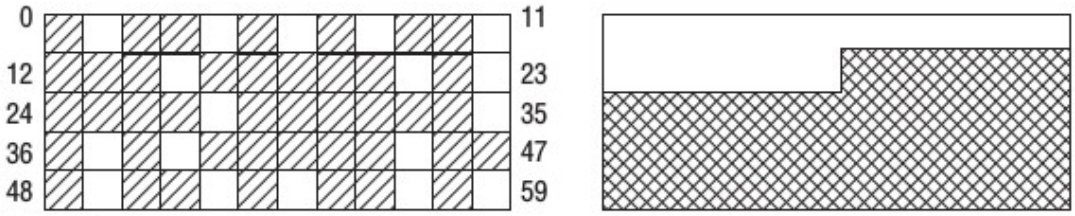


Figure 3-24: Fragmentation of physical memory.

…
(In the example above) Although roughly 25 percent of the physical memory is still unallocated, the largest continuous free area is only a single page.

This is no problem for userspace applications: Since their memory is mapped over page tables, it will always appear continuous to them irrespective of how the free pages are distributed in physical memory.

The right-hand side shows the situation with the same number of used and free pages, but with all free pages located in a continuous area.

Fragmentation is, however, a problem for the kernel: Since (most) RAM is identity-mapped into the kernel's portion of the address space, it cannot map an area larger than a single page in this scenario.

While many kernel allocations are small, there is sometimes the need to allocate more than a single page *<< Eg. whenever a thread is created, we typically need to allocate 2 pages for it's kernel mode stack >>*. Clearly, the situation on the right-hand side, where all reserved and free pages are in continuous regions, would be preferable.

Interestingly, problems with fragmentation can already occur when most of the memory is still unallocated. Consider the situation in Figure 3-25. Only 4 pages are reserved, but the largest contiguous area that can be allocated is 8 pages because the buddy system can only work with allocation ranges that are powers of 2.

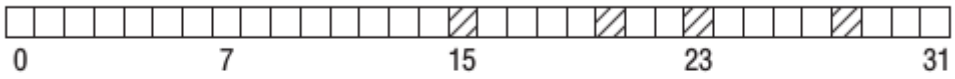


Figure 3-25: Memory fragmentation where few reserved pages prevent the allocation of larger contiguous blocks.

… Most modern CPUs provide the possibility to work with huge pages whose page size is much bigger than for regular pages. This has benefits for memory-intensive applications. When bigger pages are used, the translation lookaside buffer has to handle fewer entries, and the chance of a TLB cache miss is reduced. Allocating huge pages, however, requires free contiguous areas of physical RAM!

Fragmentation of physical memory has, indeed, belonged to the weaker points of Linux for an unusually long time span. Although many approaches have been suggested, none could satisfy the

demanding needs of the numerous workloads that Linux has to face without having too great an impact on others.

During the development of kernel 2.6.24, means to prevent fragmentation finally found their way into the kernel. Before I discuss their strategy, one point calls for clarification: Fragmentation is also known from filesystems, and in this area the problem is typically solved by defragmentation tools: They analyze the filesystem and rearrange the allocated blocks such that larger continuous areas arise. This approach would also be possible for RAM, in principle, but is complicated by the fact that many physical pages cannot be moved to an arbitrary location. Therefore, the kernel's approach is ***anti-fragmentation***: Try to prevent fragmentation as well as possible from the very beginning.

How does anti-fragmentation work? To understand the approach, we must be aware that the kernel distinguishes three different types of reserved pages:

❑ Non-movable pages have a fixed position in memory and cannot be moved anywhere else. Most allocations of the core kernel fall into this category. << *kmalloc & friends* >>

❑ Reclaimable pages cannot be moved directly, but they can be deleted and their contents regenerated from some source. Data mapped from files fall into this category, for instance. Reclaimable pages are periodically freed by the kswapd daemon depending on how often they are accessed. ...

It is also possible to initiate page reclaim when there is an acute shortage of memory, that is, when an allocation has failed. You will see further below when the kernel deems it necessary to do so.

❑ Movable pages can be moved around as desired. Pages that belong to userspace applications fall into this category. They are mapped via page tables. If they are copied into a new location, the page table entries can be updated accordingly, and the application won't notice anything.

A page has a certain mobility depending into which of the three categories it falls. The anti-fragmentation technique used by the kernel is based on the idea of grouping pages with identical mobility together.

Why does this approach help to reduce fragmentation? Recall from Figure 3-25 that a (single) page that cannot be moved somewhere else can prevent continuous allocations in an otherwise nearly completely empty RAM area. By distributing pages onto different lists depending on their mobility, this situation is prevented. For instance, a non-movable page cannot be located in the middle of a block of movable pages and effectively prevent any larger part of the block from being used.

Imagine that most of the free pages in Figure 3-25 belong to the reclaimable category, while the reserved pages are non-movable. If the pages had been collected on two different lists, the situation

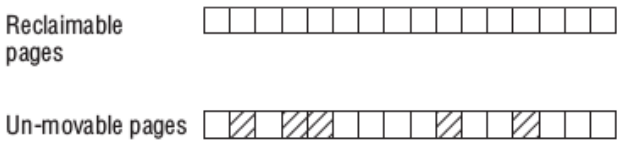


Figure 3-26: Memory fragmentation is reduced by grouping pages together depending on their mobility.

might, however, look as shown in Figure 3-26. It is still hard to find a large continuous free space for non-movable pages, but much easier for reclaimable pages.
…

Data Structure

Although the anti-fragmentation technique used by the kernel is highly effective, it has astonishingly little impact on code and data structures of the buddy allocator. The kernel defines some macros <<*now an enum*>> to represent the different migrate types:

File : [3.10.24] : include/linux/mmzone.h

```
...
  38 enum {      << enumeration of page migration types >>
  39     MIGRATE_UNMOVABLE,
  40     MIGRATE_RECLAIMABLE,
  41     MIGRATE_MOVABLE,
  42     MIGRATE_PCPTYPES,    /* the number of types on the pcp lists
*/<< 3 >>
  43     MIGRATE_RESERVE = MIGRATE_PCPTYPES,
  44 #ifdef CONFIG_CMA           << CMA = Contiguous Memory Allocator >>
  45     /*
  46      * MIGRATE_CMA migration type is designed to mimic the way
  47      * ZONE_MOVABLE works.  Only movable pages can be allocated
  48      * from MIGRATE_CMA pageblocks and page allocator never
  49      * implicitly change migration type of MIGRATE_CMA pageblock.
  50      *
  51      * The way to use it is to change migratetype of a range of
  52      * pageblocks to MIGRATE_CMA which can be done by
  53      * __free_pageblock_cma() function.  What is important though
  54      * is that a range of pageblocks must be aligned to
  55      * MAX_ORDER_NR_PAGES should biggest page be bigger then
  56      * a single pageblock.
  57      */
  58     MIGRATE_CMA,
  59 #endif
  60 #ifdef CONFIG_MEMORY_ISOLATION        <<  emergency memory reserve
>>
  61     MIGRATE_ISOLATE,    /* can't allocate from here */
  62 #endif
  63     MIGRATE_TYPES    << curr value = 5 or 6, depending on CMA and
                             ISOLATE being enabled or not >>
  64 };
...
```

The types MIGRATE_UNMOVABLE , MIGRATE_RECLAIMABLE , and MIGRATE_MOVABLE have already been introduced.

MIGRATE_RESERVE provides an emergency memory reserve if an allocation request cannot be

fulfilled from the mobility-specific lists (it is filled during initialization of the memory subsystem with setup_zone_migrate_reserve , but I will not go into detail about this).

MIGRATE_ISOLATE is a special virtual zone that is required to move physical pages across NUMA nodes. On large systems, it can be beneficial to bring physical pages closer to the CPUs that use them most.

MIGRATE_TYPES , finally, is also not a zone, but just denotes the number of migrate types. << *It's value will be 5 or 6, depending on CONFIG_CMA and CONFIG_MEMORY_ISOLATION being enabled or not.*

These 4 types are always present:
UNMOVABLE, RECLAIMABLE, MOVABLE and RESERVED. >>

The core adjustment to the buddy system data structures is that the free list is broken into a MIGRATE_TYPES number of lists:

```
83 struct free_area {
84     struct list_head    free_list[MIGRATE_TYPES];
85     unsigned long       nr_free;   << # of free 2^n page blocks >>
86 };
```

nr_free counts the number of free pages on all lists, but **a specific free list is provided for each migrate type (for each order!!)**.

The macro *for_each_migratetype_order(order, type)* can be used to iterate over the migrate types of all allocation orders.

```
72 #define for_each_migratetype_order(order, type) \
73     for (order = 0; order < MAX_ORDER; order++) \
74         for (type = 0; type < MIGRATE_TYPES; type++)
```

[P. T. O.]

<<
So, prior to 2.6.24, the buddy-system can be visualized as:

```
                          :: order 0 list
                          :: order 1 list
Node R :: Zone            ::   --snip--
                          :: order n-2 list
                          :: order n-1 list ; where n = MAX_ORDER (11)
```

Now, 2.6.24 onwards, for a system with N (NUMA) nodes and page mobility grouping:

```
                                  :: order 0 list
                                  :: order 1 list
Node R :: Zone :: Migration Type  ::   --snip--
                                  :: order n-2 list
                                  :: order n-1 list ; where n = MAX_ORDER
                                                       (11 on
ARM,x86)
```

Node Zone Migration Type

```
Node 0 : Zone DMA      : MIGRATE_UNMOVABLE     :: order 0, 1, 2, … (n-1)
lists
Node 0 : Zone DMA      : MIGRATE_RECLAIMABLE   :: order 0, 1, 2, … (n-1)
lists
Node 0 : Zone DMA      : MIGRATE_MOVABLE
Node 0 : Zone DMA      : MIGRATE_RESERVED      ...
Node 0 : Zone DMA      : [MIGRATE_CMA]
Node 0 : Zone DMA      : [MIGRATE_ISOLATE]     :: order 0, 1, 2, … (n-1)
lists

Node 0 : Zone NORMAL   : MIGRATE_UNMOVABLE     :: order 0, 1, 2, … (n-1)
lists
Node 0 : Zone NORMAL   : MIGRATE_RECLAIMABLE   :: order 0, 1, 2, … (n-1)
lists
Node 0 : Zone NORMAL   : MIGRATE_MOVABLE
Node 0 : Zone NORMAL   : MIGRATE_RESERVED      ...
Node 0 : Zone NORMAL   : [MIGRATE_CMA]
Node 0 : Zone NORMAL   : [MIGRATE_ISOLATE]     :: order 0, 1, 2, … (n-1)
lists

Node 0 : Zone HIGHMEM  : MIGRATE_UNMOVABLE     :: order 0, 1, 2, … (n-1)
lists
Node 0 : Zone HIGHMEM  : MIGRATE_RECLAIMABLE   :: order 0, 1, 2, … (n-1)
lists
Node 0 : Zone HIGHMEM  : MIGRATE_MOVABLE
Node 0 : Zone HIGHMEM  : MIGRATE_RESERVED      ...
Node 0 : Zone HIGHMEM  : [MIGRATE_CMA]
Node 0 : Zone HIGHMEM  : [MIGRATE_ISOLATE]     :: order 0, 1, 2, … (n-1)
lists
```

```
...
...

Node N-1 : Zone DMA     : MIGRATE_UNMOVABLE   :: order 0, 1, 2, … (n-1)
lists
Node N-1 : Zone DMA     : MIGRATE_RECLAIMABLE :: order 0, 1, 2, … (n-1)
lists
Node N-1 : Zone DMA     : MIGRATE_MOVABLE
Node N-1 : Zone DMA     : MIGRATE_RESERVED    ...
Node N-1 : Zone DMA     : [MIGRATE_CMA]
Node N-1 : Zone DMA     : [MIGRATE_ISOLATE]     :: order 0, 1, 2, … (n-1)
lists

Node N-1 : Zone NORMAL    : MIGRATE_UNMOVABLE   :: order 0, 1, 2, … (n-1)
lists
Node N-1 : Zone NORMAL    : MIGRATE_RECLAIMABLE :: order 0, 1, 2, … (n-1)
lists
Node N-1 : Zone NORMAL    : MIGRATE_MOVABLE
Node N-1 : Zone NORMAL    : MIGRATE_RESERVED    ...
Node N-1 : Zone NORMAL    : [MIGRATE_CMA]
Node N-1 : Zone NORMAL    : [MIGRATE_ISOLATE]   :: order 0, 1, 2, … (n-1)
lists

Node N-1 : Zone HIGHMEM   : MIGRATE_UNMOVABLE   :: order 0, 1, 2, … (n-1)
lists
Node N-1 : Zone HIGHMEM   : MIGRATE_RECLAIMABLE :: order 0, 1, 2, … (n-1)
lists
Node N-1 : Zone HIGHMEM   : MIGRATE_MOVABLE
Node N-1 : Zone HIGHMEM   : MIGRATE_RESERVED    ...
Node N-1 : Zone HIGHMEM   : [MIGRATE_CMA]
Node N-1 : Zone HIGHMEM   : [MIGRATE_ISOLATE]   :: order 0, 1, 2, … (n-1)
lists
```

What happens if the kernel cannot fulfill an allocation request for a given migrate type? … The kernel provides a fallback list - `fallbacks -` regulating which migrate types should be used next if a request cannot be fulfilled from the desired list:

File : [3.10.24] : mm/page_alloc.c

```
...
 913 /*
 914  * This array describes the order lists are fallen back to when
 915  * the free lists for the desirable migrate type are depleted
 916  */
 917 static int fallbacks[MIGRATE_TYPES][4] = {
 918     [MIGRATE_UNMOVABLE]   = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE,
MIGRATE_RESERVE },
 919     [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE,   MIGRATE_MOVABLE,
MIGRATE_RESERVE },
 920 #ifdef CONFIG_CMA
 921     [MIGRATE_MOVABLE]     = { MIGRATE_CMA,
MIGRATE_RECLAIMABLE, MIGRATE_UNMOVABLE, MIGRATE_RESERVE },
 922     [MIGRATE_CMA]         = { MIGRATE_RESERVE }, /* Never used */
```

```
 923 #else
 924     [MIGRATE_MOVABLE]     = { MIGRATE_RECLAIMABLE,
MIGRATE_UNMOVABLE,   MIGRATE_RESERVE },
 925 #endif
 926     [MIGRATE_RESERVE]     = { MIGRATE_RESERVE }, /* Never used */
 927 #ifdef CONFIG_MEMORY_ISOLATION
 928     [MIGRATE_ISOLATE]     = { MIGRATE_RESERVE }, /* Never used */
 929 #endif
 930 };
…
```

The data structure is mostly self-explanatory: When the kernel wants to allocate un-movable pages, but the corresponding list is empty, then it falls back to reclaimable pages, then to movable pages, and finally to the emergency reserve.

… Notice that the current state of page distribution across the migrate lists can be found in */proc/pagetypeinfo* :

[P.T.O.]

```
# cat /proc/pagetypeinfo
Page block order: 9
Pages per block:  512

Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone      DMA, type    Unmovable      9      5      9      3      4      4      3      0      0      0      0
Node    0, zone      DMA, type  Reclaimable     14      8     27     18     14      7      0      0      0      0      0
Node    0, zone      DMA, type      Movable     21     18     20     16      9      8      1      0      0      1      0
Node    0, zone      DMA, type      Reserve      0      0      0      0      0      0      0      0      0      1      0
Node    0, zone      DMA, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone      DMA, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type    Unmovable     89     77     32     15      5      4      2      1      0      0      0
Node    0, zone   Normal, type  Reclaimable    211    161    103    138    267     86     38     20      7      0      0
Node    0, zone   Normal, type      Movable   2614    958    304    218     37     86     53     31      9      0      0
Node    0, zone   Normal, type      Reserve      2      1      1      1      0      0      0      0      0      1      0
Node    0, zone   Normal, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type    Unmovable      1      1      1      1      0      2      2      1      0      0      0
Node    0, zone  HighMem, type  Reclaimable      3   1568     81     76     34     17      3      0      0      0      0
Node    0, zone  HighMem, type      Movable   7199   1068     62     90     22      7      5      0      0      0      0
Node    0, zone  HighMem, type      Reserve      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type      Isolate      0      0      0      0      0      0      0      0      0      0      0

Number of blocks type     Unmovable  Reclaimable      Movable      Reserve          CMA      Isolate
Node 0, zone      DMA             2            2            3            1            0            0
Node 0, zone   Normal            61          121          254            2            0            0
Node 0, zone  HighMem            24          218         1363            1            0            0
#
```

<< Above output on an IA-32 with 3 GB RAM, running Linux kernel ver 3.13.0, (distro: Ubuntu 14.04.1 LTS) >>

Global Variables and Auxiliary Functions

While page mobility grouping is always compiled into the kernel, it only makes sense if enough

memory that can be distributed across multiple migrate lists is present in a system.

Since on each migrate list a suitable amount of memory should be present, the kernel needs a notion of ‘‘suitable.’’

This is provided by the two global variables *pageblock_order* and *pageblock_nr_pages* . The first denotes an allocation order that is considered to be ‘‘large”, and *pageblock_nr_pages* denotes the corresponding number of pages for this allocation order. Usually the the page order is selected to be the order of huge pages if such are provided by the architecture:

```
<pageblock-flags.h>
#define pageblock_order HUGETLB_PAGE_ORDER
```

On the IA-32 architecture, huge pages are 4 MiB in size, so each huge page consists of 1,024 regular pages and HUGETLB_PAGE_ORDER is defined to be **10**.

The IA-64 architecture, in contrast, allows varying regular and huge page sizes, so the value of HUGETLB_PAGE_ORDER depends on the kernel configuration.

If an architecture does not support huge pages, then the second highest allocation order is taken as a large order:

```
<pageblock-flags.h>
#define pageblock_order (MAX_ORDER-1)
```

<<
Notice, in the output of */proc/pagetypeinfo* above, the first two lines are the page block order (i.e. the variable *pageblock_order*) and the pages per “largest” block (i.e. the variable *pageblock_nr_pages*):

```
# cat /proc/pagetypeinfo
Page block order: 9          << allocation order that is “large” => pageblock_order >>
Pages per block:  512        << number of pages per block for this “large” order >>
…
```
>>

Page migration will not provide any benefits if each migrate type cannot at least be equipped with one large page block, so the feature is turned off by the kernel if too little memory is available. This is checked in the function *build_all_zonelists* , which is used to initialize the zone lists. If not enough memory is available, the global variable *page_group_by_mobility* is set to 0, otherwise to 1.[18]

[18] Note that systems not only with little memory but also with extremely large page sizes can be affected by this since the check is performed on a pages-per-list basis.

<<

[OPTIONAL / FYI]

Kernel ver 2.6.35

Memory Compaction

…
Contemporary processors are not limited to 4K pages; they can work with much larger pages ("huge pages" on x86_64 of 2 MB) in portions of a process's address space. There can be real performance advantages to using huge pages, mostly as a result of reduced pressure on the processor's translation lookaside buffer.

But the use of huge pages requires that the system be able to find physically-contiguous areas of memory which are not only big enough, but which are properly aligned as well. Finding that kind of space can be quite challenging on systems which have been running for any period of time.

Over the years, the kernel developers have made various attempts to mitigate this problem; techniques like ZONE_MOVABLE and lumpy reclaim have been the result. There is still more that can be done, though, especially in the area of fixing fragmentation to recover larger chunks of memory. After taking a break from this area, Mel Gorman has recently returned with a new patch set implementing memory compaction. Here we'll take a quick look at how this patch works.

Imagine a very small memory zone which looks like this:

[| | | | || | | | | | | | | **]**

| = free memory page
| = in-use memory page
…

Eventually the two algorithms will meet somewhere toward the middle of the zone. At that point, it's mostly just a matter of invoking the page migration code (which is not just for NUMA systems anymore) to shift the used pages to the free space at the top of the zone, yielding a pretty picture like this:

[| | | | | | | | | | | | | | | | **]**

We now have a nice, eight-page, contiguous span of free space which can be used to satisfy higher-order allocations if need be.

Of course, the picture given here has been simplified considerably from what happens on a real system. To begin with, the memory zones will be much larger; that means there's more work to do, but the resulting free areas may be much larger as well.

But all this only works if the pages in question can actually be moved. Not all pages can be moved at will; only those which are addressed through a layer of indirection and which are not otherwise pinned down are movable. So most user-space pages - which are accessed through user virtual addresses - can be moved; all that is needed is to tweak the relevant page table entries accordingly.

Most memory used by the kernel directly cannot be moved - though some of it is reclaimable, meaning that it can be freed entirely on demand. It only takes one non-movable page to ruin a contiguous segment of memory. The good news here is that the kernel already takes care to separate movable and non-movable pages, so, in reality, non-movable pages should be a smaller problem than one might think.

The running of the compaction algorithm can be triggered in either of two ways. One is to write a node number to */proc/sys/vm/compact_node*, causing compaction to happen on the indicated NUMA node. The other is for the system to fail in an attempt to allocate a higher-order page; in this case, compaction will run as a preferable alternative to freeing pages through direct reclaim. In the absence of an explicit trigger, the compaction algorithm will stay idle; there is a cost to moving pages around which is best avoided if it is not needed.

Mel ran some simple tests showing that, with compaction enabled, he was able to allocate over 90% of the system's memory as huge pages while simultaneously decreasing the amount of reclaim activity needed. So it looks like a useful bit of work. It is memory management code, though, so the amount of time required to get into the mainline is never easy to predict in advance.

<< is in: *Kernel Features* :
CONFIG_COMPACTION :
  | Allows the compaction of memory for the allocation of huge pages.
>>

```
# ls -l /proc/sys/vm/compact_memory
--w------- 1 root root 0 Jul 31 18:55 /proc/sys/vm/compact_memory
#
```
>>

CMA : The Contiguous Memory Allocator

Source: A reworked contiguous memory allocator, Jon Corbet, LWN

The problem of allocating large chunks of physically-contiguous memory has often been discussed in these pages. Virtual memory, by its nature, tends to scatter pages throughout the system; the kernel does not have to be running for long before free pages which happen to be next to each other become scarce. For many years, the way kernel developers have dealt with this problem has been to avoid dependencies on large contiguous allocations whenever possible. Kernel code which tries to allocate more than two physically-contiguous pages is rare.

...

LWN looked at the contiguous memory allocator (CMA) patches which were meant to be an answer to this problem. This patch set followed the venerable tradition of reserving a chunk of memory at boot time for the sole purpose of satisfying large allocation requests. Over the years, this technique has been used by the "bigphysarea" patch, or simply by booting the kernel with a *mem=* parameter that left a range of physical memory unused. The out-of-tree Android pmem driver also allocates memory chunks from a reserved range. This approach certainly works; nearly 20 years of experience verifies that. The down side is that the reserved memory is not available for any other use; if the device is not using the memory, it simply sits idle. That kind of waste tends to be unpopular with kernel developers - and users.

For that reason and more, the CMA patches were never merged. The problem has not gone away, though, and neither have the developers who are working on it. The latest version of the CMA patch set looks quite a bit different; while some issues still need to be resolved, this patch set looks like it may have a much better chance of getting into the mainline.

The CMA allocator can still work with a reserved region of memory, but that is clearly not the intended mode of operation. Instead, the new CMA tries to maintain regions of memory where contiguous chunks can be created when the need arises. To that end, CMA relies on the "migration type" mechanism built deeply into the memory management code.

Within each zone, blocks of pages are marked as being for use by pages which are (or are not) movable or reclaimable *<< ('M' or 'R' flag resp, seen later) >>* . Movable pages *<< ('M' flag, seen later) >>* are, primarily, page cache or anonymous memory pages; they are accessed via page tables and the page cache radix tree. The contents of such pages can be moved somewhere else as long as the tables and tree are updated accordingly.

Reclaimable pages *<< ('R' flag, seen later) >>*, instead, might possibly be given back to the kernel on demand; they hold data structures like the inode cache. Unmovable pages *<< ('U' flag, seen later) >>*are usually those for which the kernel has direct pointers; memory obtained from *kmalloc()* cannot normally be moved without breaking things, for example.

The memory management subsystem tries to keep movable pages together. If the goal is to free a larger chunk by moving pages out of the way, it only takes a single nailed-down page to ruin the whole effort. By grouping movable pages, the kernel hopes to be able to free larger ranges on demand without running into such snags.

<<
This “grouping together” of pages of the same migration type is precisely what the newer (2.6.24 onward) buddy system freelists do!

Take a relook at the relevant data structures: the page migration types *enum*, the *zone* structure and the *free_area* structure from the previous section.
>>

The memory compaction code relies on these ranges of movable pages to be able to do its job.

CMA extends this mechanism by adding a new "CMA" migration type; it works much like the "movable" type, but with a couple of differences. The "CMA" type is sticky; pages which are marked as being for CMA should never have their migration type changed by the kernel. The memory allocator will never allocate unmovable pages from a CMA area, and, for any other use, it only allocates CMA pages when alternatives are not available. So, with luck, the areas of memory which are marked for use by CMA should contain only movable pages, and it should have a relatively high number of free pages.

In other words, memory which is marked for use by CMA remains available to the rest of the system with the one restriction that it can only contain movable pages. When a driver comes along with a need for a contiguous range of memory, the CMA allocator can go to one of its special ranges and try to shove enough pages out of the way to create a

contiguous buffer of the appropriate size. If the pages contained in that area are truly movable (the real world can get in the way sometimes), it should be possible to give that driver the buffer it needs. When that buffer is not needed, though, the memory can be used for other purposes.

...

The end result is that there's likely to be at least one more iteration of this patch set before it gets into the mainline. But CMA addresses a real need which has been met, thus far, with out-of-tree hacks of varying kludginess. This code has the potential to make physically-contiguous allocations far more reliable while minimizing the impact on the rest of the system. It seems worth having.

<<

File : [3.10.24] : include/linux/dma-contiguous.h

```
...
/*
 * Contiguous Memory Allocator
 *
 *   The Contiguous Memory Allocator (CMA) makes it possible to
 *   allocate big contiguous chunks of memory after the system has
 *   booted.
 *
 * Why is it needed?
 *
 *   Various devices on embedded systems have no scatter-getter and/or
 *   IO map support and require contiguous blocks of memory to
 *   operate.  They include devices such as cameras, hardware video
 *   coders, etc.
 *
 *   Such devices often require big memory buffers (a full HD frame
 *   is, for instance, more then 2 mega pixels large, i.e. more than 6
 *   MB of memory), which makes mechanisms such as kmalloc() or
 *   alloc_page() ineffective.
 *
 *   At the same time, a solution where a big memory region is
 *   reserved for a device is suboptimal since often more memory is
 *   reserved then strictly required and, moreover, the memory is
 *   inaccessible to page system even if device drivers don't use it.
 *
 *   CMA tries to solve this issue by operating on memory regions
 *   where only movable pages can be allocated from.  This way, kernel
 *   can use the memory for pagecache and when device driver requests
 *   it, allocated pages can be migrated.
 *
 * Driver usage
 *
 *   CMA should not be used by the device drivers directly. It is
 *   only a helper framework for dma-mapping subsystem.
 *
 *   For more information, see kernel-docs in drivers/base/dma-
contiguous.c
```

```
 */
...
```

<<
On an Android phone (an S7):

```
$ adb shell
herolte:/ $ uname -a
Linux localhost 3.18.14-11104523 #1 SMP PREEMPT Mon Jul 10 17:36:37 KST 2017 aarch64
herolte:/ $ cat /proc/pagetypeinfo
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4      5      6      7
8      9     10
Node    0, zone   Normal, type    Unmovable    147     45      0      0      0      0      0      0
0      0      0
Node    0, zone   Normal, type  Reclaimable     51     21      0      0      0      0      0      0
0      0      0
Node    0, zone   Normal, type      Movable   6432   2976     11      0      0      0      0      0
0      0      0
Node    0, zone   Normal, type      Reserve      0      0      8      0      0      0      0      0
0      0      0
Node    0, zone   Normal, type          CMA   2517   4380    902     54      0      0      0      0
0      0      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0
0      0      0

Number of blocks type     Unmovable  Reclaimable      Movable      Reserve          CMA      Isolate
Node 0, zone   Normal          155            9          785            2           66            0
>>
```

<<
From the kernel configuration menu:

```
$ make ARCH=arm menuconfig
...
 CONFIG_CMA:
   |
   | This enables the Contiguous Memory Allocator which allows drivers
   | to allocate big physically-contiguous blocks of memory for use with
   | hardware components that do not support I/O map nor scatter-gather.
   |
   | For more information see <include/linux/dma-contiguous.h>.
   | If unsure, say "n".

...
Symbol: CMA [=y]
   | Type  : boolean
   | Prompt: Contiguous Memory Allocator
   |   Location:
   |     -> Device Drivers
   | (1)   -> Generic Driver Options
```

```
   |   Defined at drivers/base/Kconfig:205
   |   Depends on: HAVE_DMA_CONTIGUOUS [=y] && HAVE_MEMBLOCK [=y]
   |   Selects: MIGRATION [=y] && MEMORY_ISOLATION [=y]
   |   Selected by: DA8XX_REMOTEPROC [=n] && ARCH_DAVINCI_DA8XX [=n]
...
Symbol: CMA_SIZE_MBYTES [=16]
   | Type  : integer
   | Prompt: Size in Mega Bytes
   |   Location:
   |     -> Device Drivers
   |       -> Generic Driver Options
   | (8)     -> Contiguous Memory Allocator (CMA [=y])
...
>>
```

<<

A brief note on ***per-cpu*** variables:

A per-CPU variable in the Linux kernel is actually an array with one instance of the variable for each processor. Each processor works with its own copy of the variable; this can be done with no locking, and with no worries about cache line bouncing. [Kernel preemption (and implicitly cpu migration) must be disabled; thus, access to per-CPU variables is normally bracketed with calls to *get_cpu_var()* and *put_cpu_var()*. [More details covered in the "Kernel Synchronization" module.]
>>

[OPTIONAL / FYI]

<< Professional Linux Kernel Architecture by Wolfgang Mauerer, Wrox Press. >>

Pagesets and pcp Lists

…

pageset is an array to implement per-CPU hot-n-cold page lists. The kernel uses these lists to store fresh pages that can be used to satisfy implementations. However, they are distinguished by their (cpu) cache status: Pages that are most likely still cache-hot and can therefore be quickly accessed are separated from cache-cold pages.

(In addition, in conjunction with the buddy-system freelists, pagesets are used to group together pages having the same *mobility* (covered earlier).)

...

Source : ULVMM

Per-CPU Page (pcp) Lists

The most important addition to the page allocation is the addition of the per-cpu lists, first discussed in Section 2.6.

In 2.4, a page allocation requires an interrupt safe spinlock to be held while the allocation takes place. In 2.6, pages are allocated from a *struct per_cpu_pageset* by *buffered_rmqueue()*. If the low watermark (*per_cpu_pageset→low*) has not been reached, the pages will be allocated from the pageset with no requirement for a spinlock to be held. Once the low watermark is reached, a large number of pages will be allocated in bulk with the interrupt safe spinlock held, added to the per-cpu list and then one returned to the caller.

<<
File : [kernel ver 3.10.24] : include/linux/mmzone.h

```
...
  38 enum {      << enumeration of page migration types >>
  39     MIGRATE_UNMOVABLE,
  40     MIGRATE_RECLAIMABLE,
  41     MIGRATE_MOVABLE,
  42     MIGRATE_PCPTYPES,    /* the number of types on the pcp lists */
  43     MIGRATE_RESERVE = MIGRATE_PCPTYPES,
  44 #ifdef CONFIG_CMA     << Contiguous Memory Allocator >>
  45     /*
  46      * MIGRATE_CMA migration type is designed to mimic the way
  47      * ZONE_MOVABLE works.  Only movable pages can be allocated
...
  58     MIGRATE_CMA,
  59 #endif
  60 #ifdef CONFIG_MEMORY_ISOLATION
  61     MIGRATE_ISOLATE,    /* can't allocate from here */
  62 #endif
  63     MIGRATE_TYPES    << Value will be 5 or 6, depending on CMA and
                             ISOLATE being enabled or not >>
  64 };
...
...
 238 struct per_cpu_pages {
 239     int count;       /* number of pages in the list */  << this is
the
                                                    'usd' # in sysrq-m o/p
>>
 240     int high;        /* high watermark, emptying needed */
 241     int batch;       /* chunk size for buddy add/remove */
 242
 243     /* Lists of pages, one per migrate type stored on the pcp-lists
*/
 244     struct list_head lists[MIGRATE_PCPTYPES];        << = 3 >>
 245 };
 246
 247 struct per_cpu_pageset {                  << 1 of these per-cpu >>
 248     struct per_cpu_pages pcp;
 249 #ifdef CONFIG_NUMA
```

```
 250     s8 expire;
 251 #endif
 252 #ifdef CONFIG_SMP
 253     s8 stat_threshold;
 254     s8 vm_stat_diff[NR_VM_ZONE_STAT_ITEMS];
 255 #endif
 256 };
...
```

>>

<<

…

Whereas `count` keeps a record of the number of pages associated with the element, `high` is a watermark.

If the value of count exceeds `high` , this indicates that there are too many pages in the list. No explicit watermark for low fill states is used: When no elements are left, the list is refilled.

`lists` is a doubly linked list that holds the per-CPU pages and is handled using standard methods of the kernel.

If possible, the per-CPU caches are not filled with individual pages but with multipage chunks. `batch` is a guideline to the number of pages to be added in a single pass.

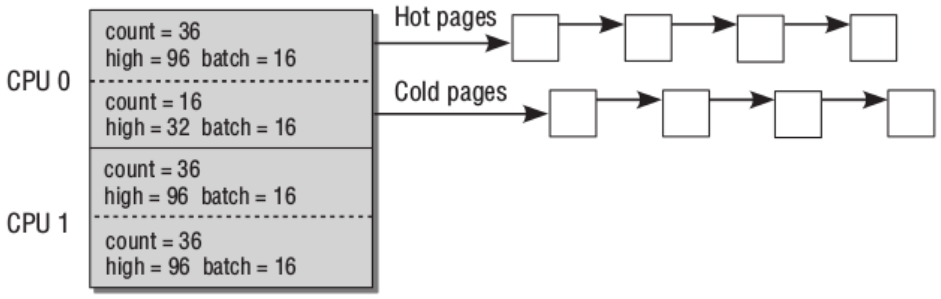


Figure 3-6: Per-CPU cache on a dual-processor system.

Figure 3-6 illustrates graphically how the data structures of the per-CPU cache are filled on a dual- processor system.

>>

…

Higher order allocations, which are relatively rare, still require the interrupt safe spinlock to be held and there will be no delay in the splits or coalescing. With 0 order allocations, splits will be delayed until the low watermark is reached in the per-cpu set and coalescing will be delayed until the high watermark is reached.

However, strictly speaking, this is not a lazy buddy algorithm [BL89]. While pagesets introduce a merging delay for order-0 allocations, it is a side-effect rather than an intended feature and there is no method available to drain the pagesets and merge the buddies. In other words, despite the per-cpu and new accounting code which bulks up the amount of code in *mm/page_alloc.c*, the core of the buddy algorithm remains the same as it was in 2.4.

The implication of this change is straight forward; the number of times the spinlock protecting the buddy lists must be acquired is reduced. Higher order allocations are relatively rare in Linux so the optimisation is for the common case. This change will be noticeable on large number of CPU machines but will make little difference to single CPUs. There are a few issues with pagesets but they are not recognised as a serious problem. The first issue is that high order allocations may fail if the pagesets hold order-0 pages that would normally be merged into higher order contiguous blocks. The second is that an order-0 allocation may fail if memory is low, the current CPU pageset is empty and other CPU's pagesets are full, as no mechanism exists for reclaiming pages from “remote” pagesets. The last potential problem is that buddies of newly freed pages could exist in other pagesets leading to possible fragmentation problems.

Freeing Pages

Two new API function have been introduced for the freeing of pages called free_hot_page() and free_cold_page()

<< Note: On recent 3.10.24 :
File : mm/page_alloc.c
```
1312 /*
1313  * Free a 0-order page
1314  * cold == 1 ? free a cold page : free a hot page
1315  */
1316 void free_hot_cold_page(struct page *page, int cold)
```
>>

. Predictably, they determine if the freed pages are placed on the hot or cold lists in the per-cpu pagesets. ~~However, while the free_cold_page() is exported and available for use, it is actually never called.~~

Order-0 page frees from __free_pages() and frees resuling from page cache releases by __page_cache_release() are placed on the hot list where as higher order allocations are freed immediately with __free_pages_ok().

Order-0 are usually related to userspace and are the most common type of allocation and free. By keeping them local to the CPU lock contention will be reduced as most allocations will also be of order-0.
…

Appendix K :: COW (Copy-On-Write) Internal Handling

COW Pages

Once upon a time, the full parent address space was duplicated for a child when a process forked. This was an extremely expensive operation because it is possible a significant percentage of the process would have to be swapped in from backing storage. To avoid this considerable overhead, a technique called COW (Copy On Write) is employed.

During a fork, the PTEs of the two processes are made read-only so that, when a write occurs, there will be a page fault. Linux recognizes a COW page because, even though the PTE is write protected, the controlling VMA shows the region is writable. It uses the function do_wp_page(), to handle it by making a copy of the page and assigning it to the writing process. If necessary, a new swap slot will be reserved for the page. With this method, only the page table entries have to be copied during a fork.

More on COW

Copy-On-Write across a fork() for malloc'ed memory pages

Pseudocode:

```
p = malloc(16*1024)   /* malloc 16Kb = 4 pages (assuming 4K pages)
*/
fork()
  In child:
     printf("Child  %d: p = %p\n", getpid(), p);
  In parent:
     printf("Parent %d: p = %p\n", getpid(), p);
```

```
$ ./fork_cow_chk &
[1] 9600
$ Parent 9600: p = 0x1478010     ← these are the virtual addresses
of course
  Child  9601: p = 0x1478010
```

We'll use the powerful '**crash**' utility to take a look under the hood. Use 'crash' for "live debug" by invoking it with the debug vmlinux image for your distro (or kernel) and */proc/kcore* (in place of a system dump):

```
$ uname -r
3.16.0-37-generic
```

$ sudo crash <...>/vmlinux-3.16.0-37-generic /proc/kcore /boot/System.map-3.16.0-37-generic

```
crash 7.0.8
Copyright (C) 2002-2014  Red Hat, Inc.
Copyright (C) 2004, 2005, 2006, 2010  IBM Corporation
Copyright (C) 1999-2006  Hewlett-Packard Co
Copyright (C) 2005, 2006, 2011, 2012  Fujitsu Limited
Copyright (C) 2006, 2007  VA Linux Systems Japan K.K.
Copyright (C) 2005, 2011  NEC Corporation
Copyright (C) 1999, 2002, 2007  Silicon Graphics, Inc.
Copyright (C) 1999, 2000, 2001, 2002  Mission Critical Linux, Inc.
This program is free software, covered by the GNU General Public
License,
and you are welcome to change it and/or distribute copies of it
under
certain conditions.  Enter "help copying" to see the conditions.
This program has absolutely no warranty.  Enter "help warranty"
for details.

GNU gdb (GDB) 7.6
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu"...

  SYSTEM MAP: /boot/System.map-3.16.0-37-generic
DEBUG KERNEL: /home/kaiwan/Downloads/vmlinux-3.16.0-37-generic
(3.16.0-37-generic)
    DUMPFILE: /proc/kcore
        CPUS: 4
...
     RELEASE: 3.16.0-37-generic
     VERSION: #51-Ubuntu SMP Tue May 5 13:45:59 UTC 2015
     MACHINE: x86_64  (2691 Mhz)
      MEMORY: 7.9 GB
         PID: 25671
     COMMAND: "crash"
        TASK: ffff880019cf0000  [THREAD_INFO: ffff88000b1fc000]
         CPU: 3
       STATE: TASK_RUNNING (ACTIVE)
```

crash>

...

```
crash> set 9600       ← set context to this process (the parent)
crash: current context no longer exists -- restoring "crash"
context:

    PID: 25671
COMMAND: "crash"
   TASK: ffff880019cf0000  [THREAD_INFO: ffff88000b1fc000]
    CPU: 3
  STATE: TASK_RUNNING (ACTIVE)

    PID: 9600
COMMAND: "fork_cow_chk"
   TASK: ffff88000b008000  [THREAD_INFO: ffff880063384000]
    CPU: 2
  STATE: TASK_INTERRUPTIBLE
crash>
crash> vtop 1478010      ← virtual to physical <va>
VIRTUAL     PHYSICAL
1478010     48ffc010

   PML: 63236000 => 95926067
   PUD: 95926000 => e440067
   PMD: e440050 => b5434067
   PTE: b54343c0 => 8000000048ffc865
  PAGE: 48ffc000

      PTE          PHYSICAL  FLAGS
8000000048ffc865  48ffc000  (PRESENT|USER|ACCESSED|DIRTY|NX)

      VMA           START       END     FLAGS FILE
ffff88000a16f180    1478000    149d000 8100073
```

<<

Interpretation of VMA Flags

See *include/linux/mm.h* for all VMA flag definitions.

Current VMA flags value is 0x8100073. Interpretation follows:

```
crash> eval 0x8100073
hexadecimal: 8100073
    decimal: 135266419
      octal: 1004000163
     binary:
00000000000000000000000010000001000000000001110011
crash>

1000 0001 0000 0000 0000 0111 0011   = 0x8100073
```

```
SD.A NHNA .DDR SIL. DP.G MMMM SXWR      ← VMA
DD.R LTRC .ECR ROK. WM.D SXWR HCRD        flag
2222 2222 1111 1111 1100 0000 0000      ← Bit #
7654 3210 9876 5432 1098 7654 3210
```

Bit#	Flag	Meaning

```
00      RD      VM_READ      0x00000001  /* currently active flags */
01      WR      VM_WRITE     0x00000002
02      XC      VM_EXEC       0x00000004
03      SH      VM_SHARED     0x00000008

/* mprotect() hardcodes VM_MAYREAD >> 4 == VM_READ, and so for r/w/x bits.
*/
04      MR      VM_MAYREAD    0x00000010  /* limits for mprotect() etc */
05      MW      VM_MAYWRITE   0x00000020
06      MX      VM_MAYEXEC    0x00000040
07      MS      VM_MAYSHARE   0x00000080

08      GD      VM_GROWSDOWN  0x00000100  /* general info on the segment */
09      <unused>
10      PM      VM_PFNMAP     0x00000400  /* Page-ranges managed without
"struct page",
                                           just pure PFN */
11      DW      VM_DENYWRITE  0x00000800  /* ETXTBSY on write attempts.. */

12      <unused>
13      LK      VM_LOCKED     0x00002000
14      IO      VM_IO         0x00004000  /* Memory mapped I/O or similar */
            /* Used by sys_madvise() */
15      SR      VM_SEQ_READ   0x00008000  /* App will access data
sequentially */

16      RR      VM_RAND_READ  0x00010000  /* App will not benefit from
clustered reads */
17      DC      VM_DONTCOPY   0x00020000      /* Do not copy this vma on
fork */
18      DE      VM_DONTEXPAND 0x00040000  /* Cannot expand with mremap() */
19      <unused>

20      AC      VM_ACCOUNT    0x00100000  /* Is a VM accounted object */
21      NR      VM_NORESERVE  0x00200000  /* should the VM suppress
accounting */
22      HT      VM_HUGETLB    0x00400000  /* Huge TLB Page VM */
23      NL      VM_NONLINEAR  0x00800000  /* Is non-linear
(remap_file_pages) */

24      AR      VM_ARCH_1     0x01000000  /* Architecture-specific flag */
25      <unused>
26      DD      VM_DONTDUMP   0x04000000  /* Do not include in the core dump
*/
    #ifdef CONFIG_MEM_SOFT_DIRTY
27      SD      VM_SOFTDIRTY  0x08000000  /* Not soft dirty clean area */
```

...

So the set VMA flags (for this malloc'ed (heap) region) are:

```
1000 0001 0000 0000 0000 0111 0011    = 0x8100073
SD.A NHNA .DDR SIL. DP.G MMMM SXWR      ← VMA
DD.R LTRC .ECR ROK. WM.D SXWR HCRD        flag
2222 2222 1111 1111 1100 0000 0000      ← Bit #
7654 3210 9876 5432 1098 7654 3210

VM_READ | VM_WRITE | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC |
VM_ACCOUNT |  VM_SOFTDIRTY
```

As VM_SHARED is cleared, it implies a private mapping.
The fact that VM_SHARED is cleared and that VM_MAYWRITE is set, implies a Copy-On-Write (COW) mapping.

The function *is_cow_mapping()* confirms this:

```
mm/memory.c
static inline bool is_cow_mapping(vm_flags_t flags)
{
    return (flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;
}
```

See this article - "Design of fork followed by exec in Linux" - especially the section on "Linux Implementation of Copy-on-write (COW) for shared Virtual Memory Areas".

>>

```
      PAGE         PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea000123ff00  48ffc000 ffff880132b6a551     1478  2
1ffff0000080068 uptodate,lru,active,swapbacked     << Page reference
count is 2 >>
crash>
crash> set 9601      ← set context to this process (the child)
    PID: 9601
COMMAND: "fork_cow_chk"
   TASK: ffff88000b00dbb0  [THREAD_INFO: ffff8800168ec000]
    CPU: 3
  STATE: TASK_INTERRUPTIBLE
crash>
crash> vtop 1478010
VIRTUAL     PHYSICAL
1478010     48ffc010        ← identical physical address!

   PML: 5a14a000 => e5c4067
```

```
    PUD: e5c4000 => 36444067
    PMD: 36444050 => 19d80067
    PTE: 19d803c0 => 8000000048ffc845
   PAGE: 48ffc000

       PTE          PHYSICAL  FLAGS
8000000048ffc845  48ffc000  (PRESENT|USER|DIRTY|NX)
Child's page reference count is 2
       VMA           START       END     FLAGS FILE
ffff88000a16f300    1478000    149d000 8100073

       PAGE         PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea000123ff00  48ffc000 ffff880132b6a551     1478  2
1ffff0000080068 uptodate,lru,active,swapbacked
crash>
crash>
```

Bibliography

1. *“Professional Linux Kernel Architecture” by Wolfgang Mauerer, Wrox Press*
2. *“Understanding the Linux Virtual Memory Manager” by Mel Gorman, Bruce Perns Open Source Series*
3. *Gustav Duarte's simply superb blog articles*
4. *The “IA-32 Intel Architecture Software Developers Manual” Vol 3, Intel*
5. *“Linux Kernel Development” by Robert Love, 3rd Ed, Pearson*
6. *Wikipedia.org*
7. *“Understanding the Linux Kernel” by Bovet & Cesati, 3rd Ed, O'Reilly*
8. *"Outline of the Linux Memory Management System" by Joe Knapa*
9. *“Operating System Principles” 7th Edition by Silberschatz, Galvin and Gagne, Wiley*
10. *“ARM System Developer's Guide – Designing and Optimizing System Software” by Sloss, Symes and Wright, published by Elsevier.*

Useful Resources

The Linux Memory Wiki

Linux MM Wiki Site

The Linux kernel mailing list @ linux.kernel (the LKML, can be accessed here: http://groups.google.com/group/linux.kernel)

Special Features of Linux Memory Management Mechanism

Computer Architecture: A Quantitative Approach, Hennessy & Patterson, 5th Ed.

... and so many more!
