Chapter 7 Non-Contiguous Memory Allocation

It is preferable when dealing with large amounts of memory to use physically contiguous pages in memory both for cache related and memory access latency reasons. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible. Linux provides a mechanism via `vmalloc()` where non-contiguous physically memory can be used that is contiguous in virtual memory.

An area is reserved in the virtual address space between `VMALLOC_START` and `VMALLOC_END`. The location of `VMALLOC_START` depends on the amount of available physical memory but the region will always be at least `VMALLOC_RESERVE`in size, which on the x86 is 128MiB. The exact size of the region is discussed in Section 4.1.

The page tables in this region are adjusted as necessary to point to physical pages which are allocated with the normal physical page allocator. This means that allocation must be a multiple of the hardware page size. As allocations require altering the kernel page tables, there is a limitation on how much memory can be mapped with `vmalloc()` as only the virtual addresses space between `VMALLOC_START` and `VMALLOC_END`is available. As a result, it is used sparingly in the core kernel. In `2.4.22`, it is only used for storing the swap map information (see Chapter 11) and for loading kernel modules into memory.

This small chapter begins with a description of how the kernel tracks which areas in the vmalloc address space are used and how regions are allocated and freed.

7.1 Describing Virtual Memory Areas

The vmalloc address space is managed with a resource map allocator [*Vah96*]. The `struct vm_struct`is responsible for storing the base,size pairs. It is defined in<`linux/vmalloc.h`> as:

```
14 struct vm_struct {
15         unsigned long flags;
16         void * addr;
17         unsigned long size;
18         struct vm_struct * next;
19 };
```

A fully-fledged VMA could have been used but it contains extra information that does not apply to vmalloc areas and would be wasteful. Here is a brief description of the fields in this small struct.

flags These set either to `VM_ALLOC`, in the case of use with `vmalloc()` or `VM_IOREMAP` when ioremap is used to map high memory into the kernel virtual address space;

addr This is the starting address of the memory block;

size This is, predictably enough, the size in bytes;

next is a pointer to the next `vm_struct`. They are ordered by address and the list is protected by the `vmlist_lock` lock.

As is clear, the areas are linked together via the `next` field and are ordered by address for simple searches. Each area is separated by at least one page to protect against overruns. This is illustrated by the gaps in Figure 7.1.

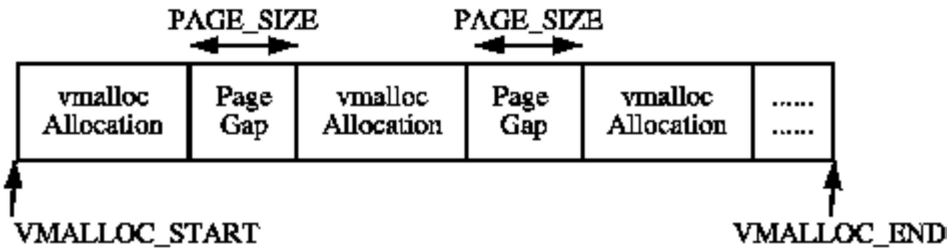

Figure 7.1: vmalloc Address Space

When the kernel wishes to allocate a new area, the `vm_struct` list is searched linearly by the function `get_vm_area()`. Space for the struct is allocated with `kmalloc()`. When the virtual area is used for remapping an area for IO (commonly referred to as *ioremapping*), this function will be called directly to map the requested area.

7.2 Allocating A Non-Contiguous Area

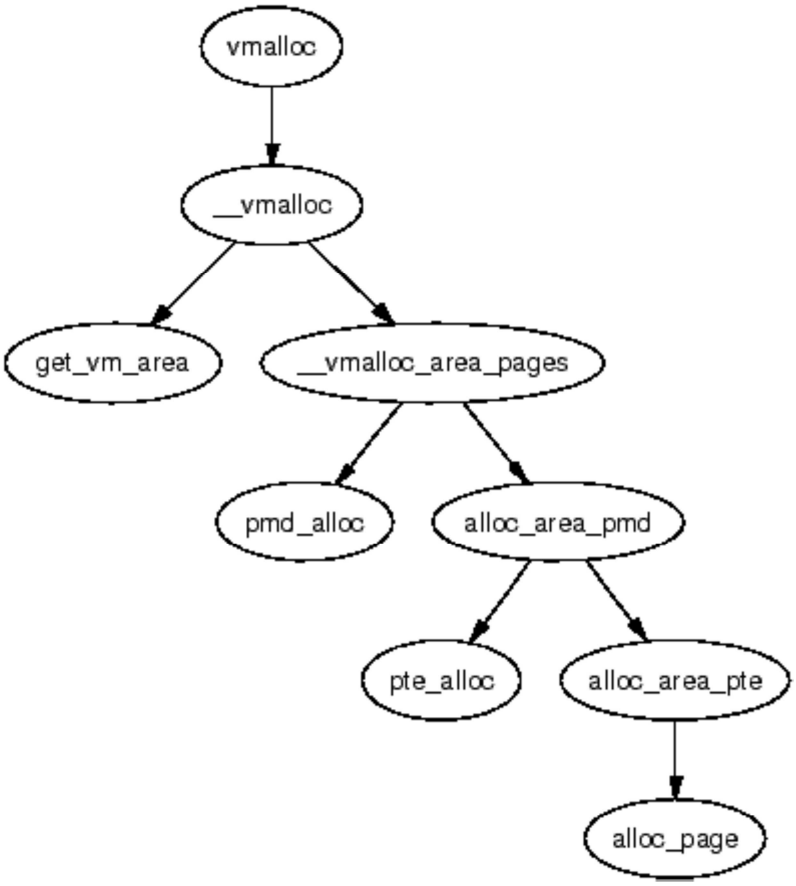

Figure 7.2: Call Graph: `vmalloc()`

The functions `vmalloc()`, `vmalloc_dma()` and `vmalloc_32()` are provided to allocate a memory area that is contiguous in virtual address space. They all take a single parameter `size` which is rounded up to the next page alignment. They all return a linear address for the new allocated area.

`void * vmalloc(unsigned long size)`
Allocate a number of pages in vmalloc space that satisfy the requested size
`void * vmalloc_dma(unsigned long size)`
Allocate a number of pages from `ZONE_DMA`
`void * vmalloc_32(unsigned long size)`
Allocate memory that is suitable for 32 bit addressing. This ensures that the physical page frames are in `ZONE_NORMAL` which 32 bit devices will require

Table 7.1: Non-Contiguous Memory Allocation API

As is clear from the call graph shown in Figure 7.2, there are two steps to allocating the area. The first step taken by `get_vm_area()` is to find a region large enough to store the request. It searches through a linear linked list of `vm_structs` and returns a new struct describing the allocated region.

The second step is to allocate the necessary PGD entries with `vmalloc_area_pages()`, PMD entries with `alloc_area_pmd()` and PTE entries with `alloc_area_pte()` before finally allocating the page with `alloc_page()`.

The page table updated by `vmalloc()` is not the current process but the reference page table stored at `init_mm→pgd`. This means that a process accessing the vmalloc area will cause a page fault exception as its page tables are not pointing to the correct area. There is a special case in the page fault handling code which knows that the fault occured in the vmalloc area and updates the current process page tables using information from the master page table. How the use of `vmalloc()` relates to the buddy allocator and page faulting is illustrated in Figure 7.3.

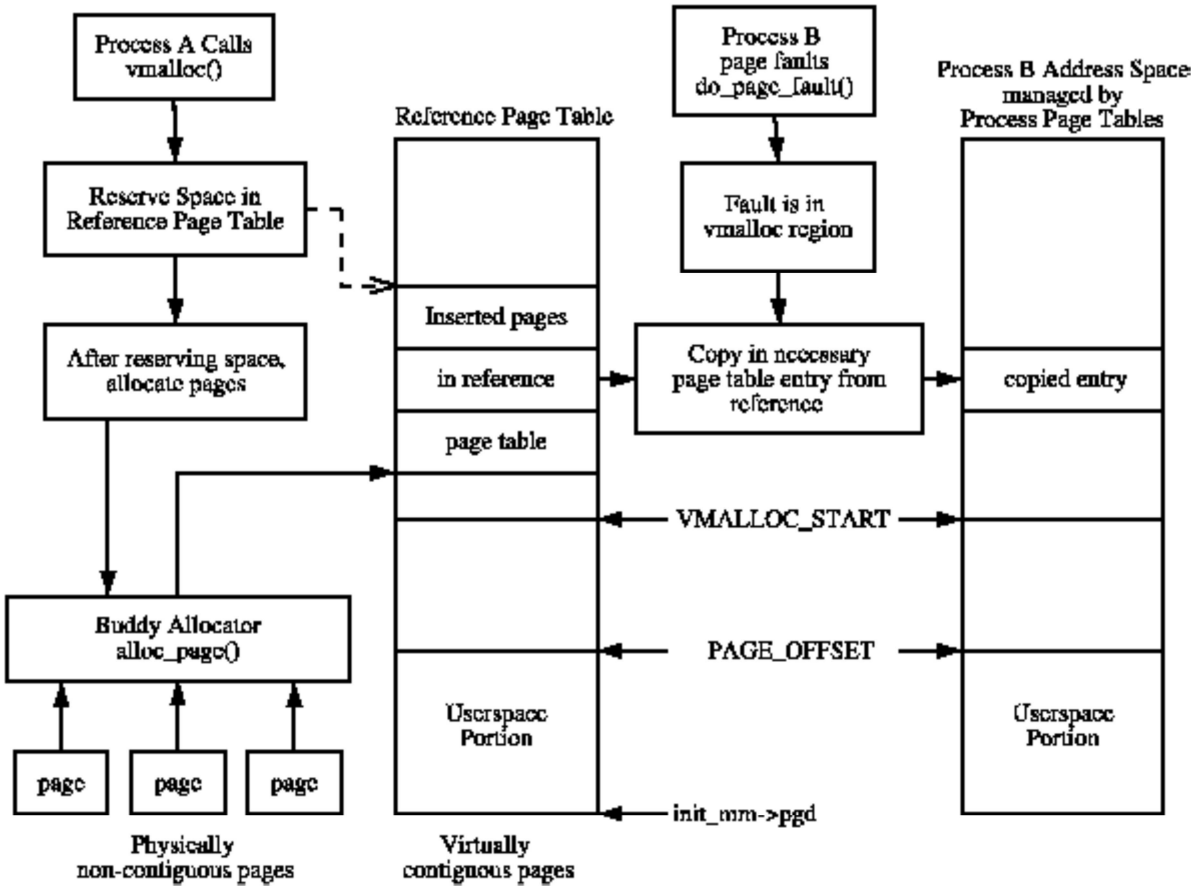

Figure 7.3: Relationship between `vmalloc()`, `alloc_page()` and Page Faulting

7.3 Freeing A Non-Contiguous Area

The function `vfree()` is responsible for freeing a virtual area. It linearly searches the list of `vm_structs` looking for the desired region and then calls `vmfree_area_pages()` on the region of memory to be freed.

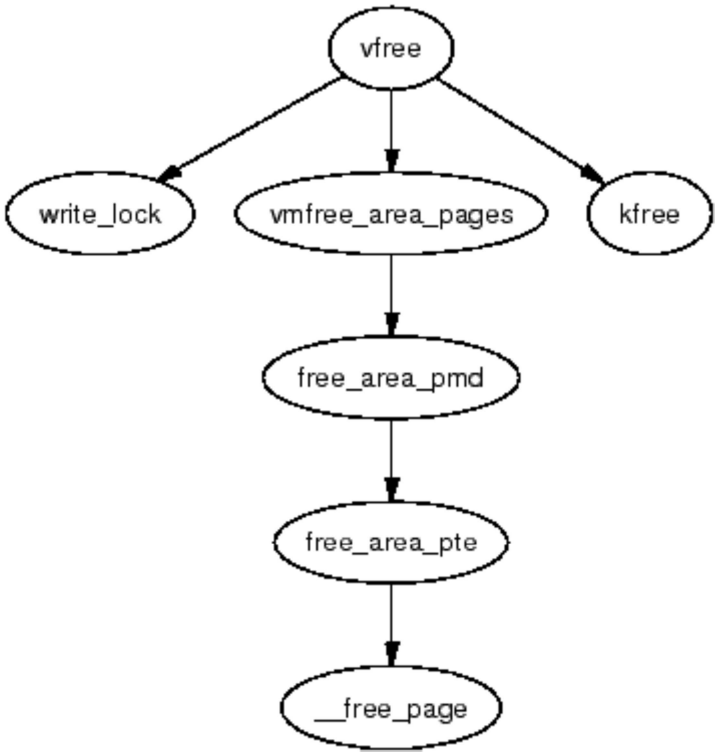

Figure 7.4: Call Graph: `vfree()`

` `vmfree_area_pages()` is the exact opposite of `vmalloc_area_pages()`. It walks the page tables freeing up the page table entries and associated pages for the region.

`void vfree(void *addr)`
Free a region of memory allocated with `vmalloc()`, `vmalloc_dma()` or `vmalloc_32()`

Table 7.2: Non-Contiguous Memory Free API

7.4 Whats New in 2.6

Non-contiguous memory allocation remains essentially the same in 2.6. The main difference is a slightly different internal API which affects when the pages are allocated. In 2.4, `vmalloc_area_pages()` is responsible for beginning a page table walk and then allocating pages when the PTE is reached in the function `alloc_area_pte()`. In 2.6, all the pages are allocated in advance by `__vmalloc()` and placed in an array which is passed to `map_vm_area()` for insertion into the kernel page tables.

The `get_vm_area()` API has changed very slightly. When called, it behaves the same as previously as it searches the entire vmalloc virtual address space for a free area. However, a caller can search just a subset of the vmalloc address space by calling `__get_vm_area()` directly and specifying the range. This is only used by the ARM architecture when loading modules.

The last significant change is the introduction of a new interface`vmap()` for the insertion of an array of pages in the vmalloc address space and is only used by the sound subsystem core. This interface was backported to 2.4.22 but it is totally unused. It is either the result of an accidental backport or was merged to ease the application of vendor-specific patches that require `vmap()`.