MMDOC. Linux Memory Management Documentation.

S. Mohan Kumar
mohan_knb@yahoo.com, http://mmdoc.sourceforge.net

1 Abstract

This document aims at documenting the Linux V2.4.18 Memory Management subsystem for the i386 architecture. This i plan to do by giving a brief description of the algorithms used for memory management, and explanation to bits and pieces of code from the linux kernel where ever needed. It does not talk about highmem yet. This document does not aim to be a code comentry, but tries to explain the logic and algorithm that goes behind the code. As such this document tries to minimize the learning curve (that i had) while trying to understand the linux kernel

2 Introduction

Memory management system is, the part of the operating system kernel that manages the Physical memory. Apart from that it also does many things like providing the,

- Physical pages to various layers of the memory management system.
- Contiguous kernel memory using a slab allocator.
- Non-contiguous memory using the vmalloc algorithm
- user-space processes with Virtual memory.

Some knowledge about the ix86 paging mechanism will be helpfull (i think mandatory) to proceed with this document. I have seen a lot of good documents from Intel, that are really usefull. I will try to link them in the bibliography.
It is not necessary to enable paging and segmentation in ix86 architecture. But enabling them provides lot of functionality like segment wise protection and virtual memory. So paging is enabled linux and is done in the early boot process itself.

2.1 A overview of the MM system

It is helpfull to have a overall theoretical view of the system before we do any real analysis of code or algorithm. First we will see about the various types of memory addresses used by the kernel.

- **Physical address** - The memory as known by the low-level hardware.
- **Direct-mapped kernel virtual address** - The physical address applied an offset.(This also is different in some architectures).
- **Pure kernel virtual address** - The address that can be dereferenced by the kernel, but which does not map into physical address directly. i.e., cannot dereferenced just by offsetting as in the case of Direct-mapped KVA
- **User virtual address** - The address as seen by user space programs

All of the physical memory is mapped to the upper one GB of kernel virtual memory. This is the reason kernel is able to use only 1GB of physical memory. But this is addressed and solved by the himem logic.

Next we will see in breif about the most basic algorithms used in the MM sub-system.

- The kernel memory map uses 4m pages in Pentium architecture. The normal virtual memory in linux is maintained as 4k pages.
- Memory is logically split into various zones. The split is different in each machine and depends, among other things,on the amount of memory we have.
- The physical memory is served by the Buddy allocator(name because it uses the buddy algorithm). The size of page served by the buddy allocator is 4k. The buddy system keeps in mind about the various zones of memory. Therefore it effectively becomes a zoned buddy allocator.
- The slab allocator is used by the kernel to allocate in-kernel memory. It is dynamically allocated memory and uses the "slab allcation algorithm". Slabs are built on top of physical pages allocated by the buddy allocator. The memory allocated by the slab allocator is contiguous
- The slab allocator provides dynamic memory upto 128 KB. When memory is needed grater than this size it is reserved at boot time itself. In other words you have to plan well ahead and allcoation is done at when the page tables are setup at boot time.
- When memory grater than 128k chunks are needed and you are not bothered weather it is physically contiguous or not, we can use the vmalloc allocator. This allocates virtually contiguous memory that may or may-not be physically contigoues.
- Memory is allocated to the user space programs by data segment adjustment (by sys_brk call) and then by servicing the page fault that arises when the user-space app tries to access that page.

2.2 Various terms used in Linux

Also it will be helpful if we understand some of the common terms used in linux. This is not very improtant, but may confuse a newbie at this point of time. This piece of information at HTTP://kos.enix.org/pub/linux-vmm.html might be usefull.

3 MM Sub-System initialization.

This section discusses in detail how the MM subsystem is initialized. SMP initialization is not discussed here.The mm sub system initialization takes place at the very beginning of the boot process itself. This is understandable as, usage of memory starts right from the boot process.

3.1 Memory categorization

File : ***arch/i386/boot/setup.S***

First of all, we have to know about the memory that is present in the system. This data will be usefull as long as the kernel is alive and kicking. The amount of memory present can be found out by querying the BIOS. There are 3 ways to get a map of physical memory present in the system.All the 3 ways use bios int 15 and different contents in the registers. (TODO add detail of os_faq)

- 0xE820 in %eax
- 0x801 in %ax
- 0x88 in ah

Memory maps are got using all these three methods and stored in separate places. Later in the setup sequence (in setup.c) they will be used one after other, on failure of previous, to generate a singe valid memory map The memory map structure (snipped from include/asm-i386/e820.h). This is an array of maximum 32 entries of the structure e820entry.

```
struct e820map {
int nr_map;
struct e820entry
{
unsigned long long addr; /* start of memory segment */
unsigned long long size; /* size of memory segment */
unsigned long type; /* type of memory segment */
} map[E820MAX];
};
```

The whole e820map is setup using bios interrupt 15 (and 0xe820 in %eax, hence the name), in the predefined address "0x2d0".

If the bios does not support e820 query, then the size is setup as the number of 1k chunks of memory available got using 0xe801 method in the predefined memory "0x1e0".

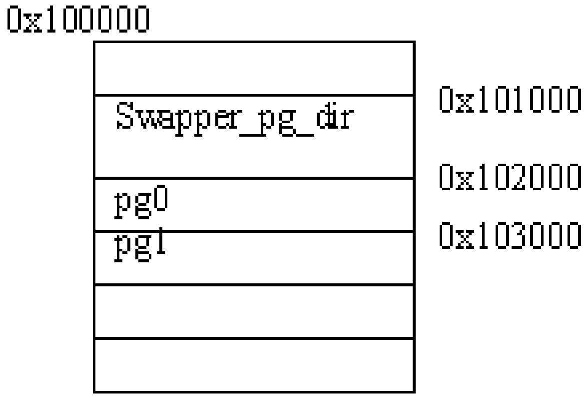
If the bios does not support 0xe801 query then we take the 0x88 query result , which will work on most (fairly all of the) bios. Here memory is given as the total MB available, but only till 64 MB.

Then a series of other initializations are done and startup_32(in head.S) is called.

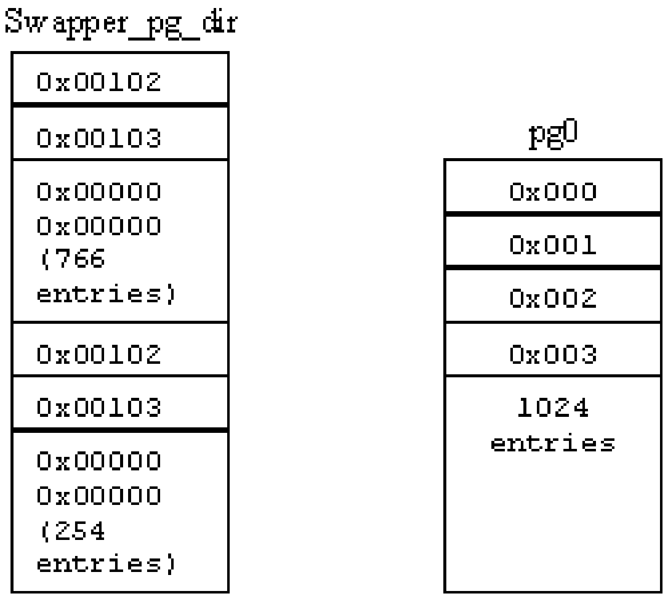
3.2 Paging start

File : ***arch/i386/kernel/head.S***

"head.S" holds the code where paging is started. The entry point name is startup_32.
All page table initializations are done here. Initial page tables (which are compiled are into the kernel) are set up and paging is started by setting the PG bit in cr0 register and the page directory pointer in the cr3 register. The compiled in page directory is stored in swapper_pg_dir symbol and the page tables in pg0 and pg1 symbols. (remember: page middle tables are not used in i386 arch. They fold into the first level page table. This also is not entirely true. When PAE is enabled the scheme changes, which we will see separately later)
The kernel code is loaded in physical address 0x100000. This is how it would look. (only symbols of interest to MM are shown here)



The contents of swapper_pg_dir and pg0 are:



Considering the above:

A virtual address of 0x100000 and 0xC0000000 will point to the same page tables and thus will transform to the same physical address. This identity mapping is needed for the kernel at boot time. Later, Proper page tables will be initialized in paging_init function.

A small piece of code helps start paging and relocate the eip from physical address to kernel virtual address (1MB + something to __PAGE_OFFSET + something) in head.S.

3.3 Start Paging

File : ***arch/i386/kernel/setup.c***

"setup.c" holds the code which completes the E820 memory map setup in setup.S and starts paging.

For easy understanding the logic used for page frame number (PFN) calculation is given here.

1. Setup the memory region by calling setup_memory_region()
   - If E820 query was a success (if number of entries in the E820 map is greater than 2) then copy the map into a local one. While doing so check weather the bios claims the 640K to 1MB ram. If it does so reclaim it by adding an entry for it into the local map.
   - If E820 query was a failure then get the values retrieved using the 801 and 88 queries and use the greatest among them. Add two entries to the local E820 map structure for "0 to 636 k" and "1MB to reported memory size".
2. Get the lowest and highest PFN.
3. If the highest PFN is greater than 229376 (the margin at 896 MB. It is not at 1024 MB because, 128 MB address space is allocated for vmalloc and inird), than HIGH_MEMORY is present.
4. If the highest PFN is grater than 1048576, then PAE memory is present. (PAE memory is a enhancement provided by some Intel processors to give 64GB address space, using 36-bit addresses)
5. Get the start_pfn as the address of the _end symbol of the kernel. The address is rounded off to the nearest next page boundary (4kb).
6. calculate max_pfn. This is done using the following logic.
   - If max_pfn is less than 229376 then proceed.
   - If max_pfn is greater than 229376 thenmax_low_pfn = 229376and highstart_pfn = 229377 and highend_pfn = max_pfn
7. Call init_bootmem and save the size of the page bitmap. (init_bootmem is called only with the low memory i.e.start_pfn and max_low_pfn)
8. Now init_bootmem would have reserved all the page frames. So go through all the page frames and free every region of memory termed as E820_RAM in the E820 memory map.
9. Reserve the physical page 0 with the boot memory as it is a bios page (not on all but some bios). Also Reserve the whole bootmem bitmap.
10. call paging_init (of arch/i386/mm/init.c).

3.4 Bootmem Handling

File : ***mm/bootmem.c***

Bootmem handling means how memory is allocated at boottime prior to paging start. This initial allocator allocates memory in page granular allocations. The bootmem allocator in Linux handles this by maintaining a bitmap large enough for accommodating all the low-memory pages (¿896 MB) in the system. Each bit in the bitmap represents one page and its index in the bitmap represents the page frame number. A value of 0 in any bit in the bitmap indicates the corresponding page is free, and 1 indicates that the page is in use. Later in the startup code, the bootmem allocator bitmap is used to determine which memory pages are in use. The corresponding page structures are marked as reserved. After paging is enabled, the bootmem allocator is not needed. So the allocator bitmap is freed.

The important functions here are

- **init_bootmem** Initializes the bootmem allocator. Initializes the bootmem bitmap.
- **reserve_bootmem** Reserves a region of the memory catered by the bootmem allocator. Memory marked as reserved in the E820 map will be successively reserved using this function.
- **free_bootmem** Frees a region of the memory catered by the bootmem allocator.
- **free_all_bootmem** Frees all the memory catered by the bootmem allocator.
- **alloc_bootmem** hands out contiguous physical pages using a firstfit algorithm. It checks the bitmap starting from the 'prefered' page [see note below], and stops if it finds the required number of contiguous free bits (or if the allocation request cannot be satisfied). It then marks this range of pages as used and returns the starting address of the range.

Note: The 'prefered' and 'goal' variables are used to specify the position from where __alloc_bootmem() should start searching. This is useful if the caller does not want to get DMA-able memory (first 16MB physical memory).

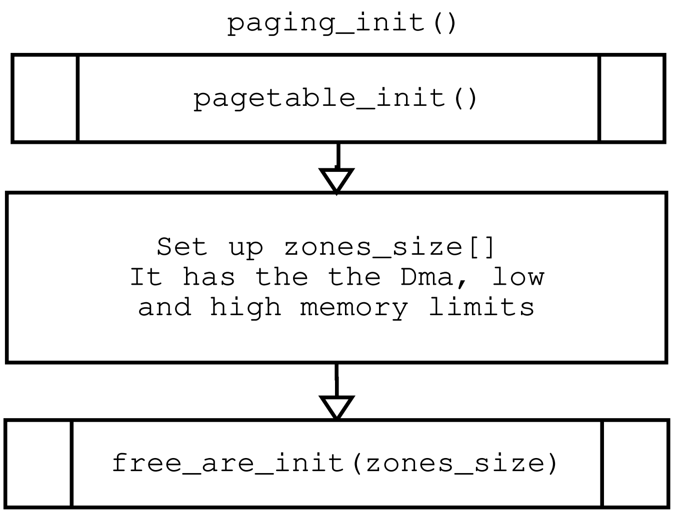
Actually, Bootmem allocation requests need not be in multiples of PAGE_SIZE, any 'size' argument is valid. But the minimum allocation unit is a page (since only 1 bit is maintained per page). So there is a good chance that memory gets fragmented. To avoid this, the bootmem allocator re-uses fragments of the last allocation whenever possible. It keeps track of the unused portion the last allocated page (using last_offset). For the next allocation, it uses this part first.

3.5 Page table setup

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

3.5.1 paging_init()

paging init is called by setup_arch() to set up the page tables and initialize the buddy allocator.
Page tables are set up by the pagetable_init() function. This function is called by the paging_init(). Here we will see in detail how page tables are set up. Also we will see some basics of zone management.



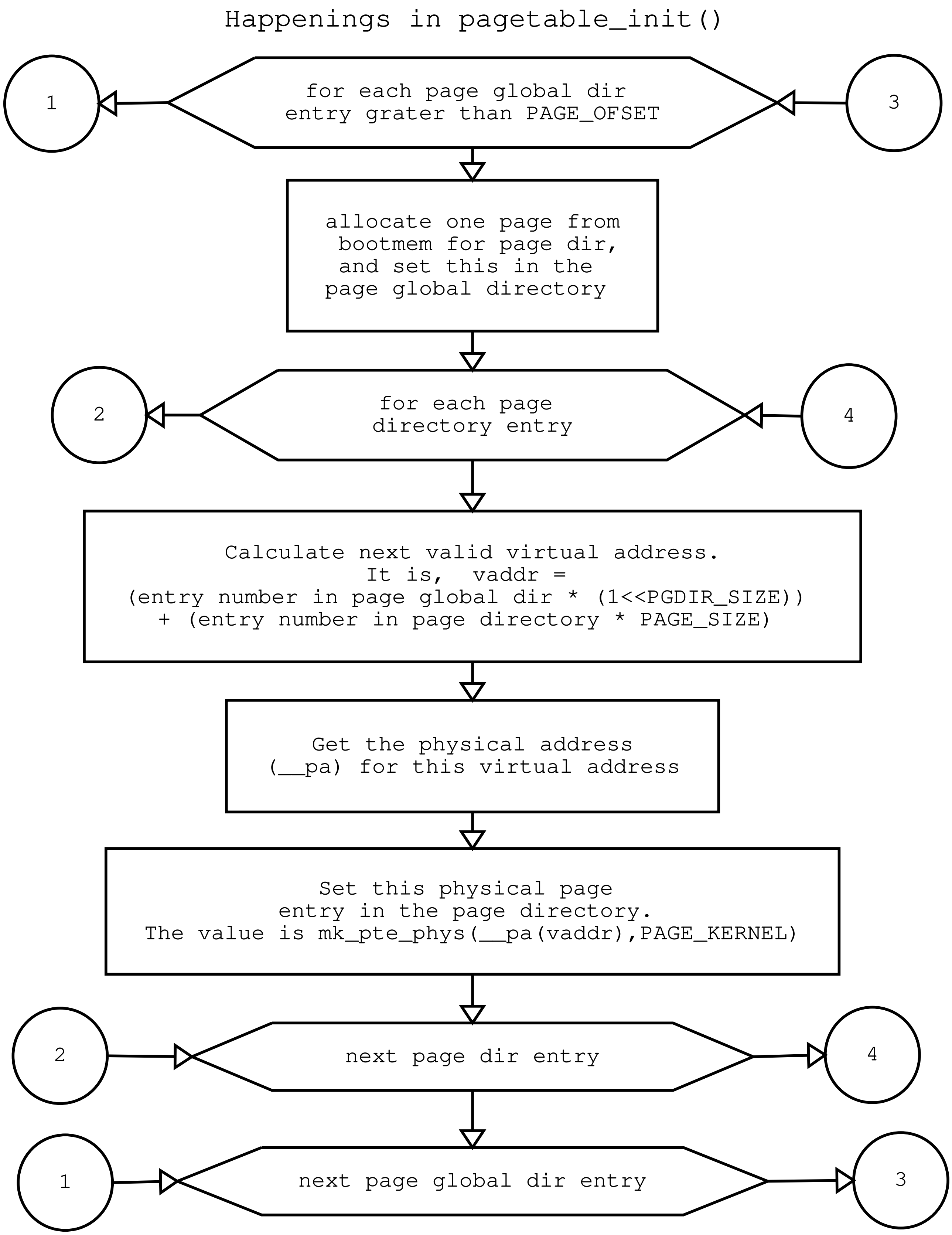
3.5.2 pagetable_init()

Setting up page table is just a task of allocating pages for the page directories and adding page table entries in these pages. The algorithm is presented here. The algorithm does not explain about page middle directories yet.

1. Memory for the swapper_pg_dir is already allocated. Allocate physical pages (using the bootmem allocator).
2. The number of pages required depends on the amount of physical pages present and the number of "page table entries" a page directory can accomodate (Specified by PTRS_PER_PTE define).
3. For each physical page add an entry in the page directory.
4. Now, put the page directory entries into the page global directory.

When such an arrangement is done, it is possible to transform the virtual address passsed to the processor, into physical one, by the paging circutary.

The actual happenings in pagetable_init() is presented as a flow chart here


Happenings in pagetable_init()
1
for each page global dir
entry grater than PAGE_OFSET
3
allocate one page from
bootmem for page dir,
and set this in the
page global directory
2
for each page
directory entry
4
Calculate next valid virtual address.
It is, vaddr =
(entry number in page global dir * (1<<PGDIR_SIZE))
+ (entry number in page directory * PAGE_SIZE)
Get the physical address
(__pa) for this virtual address
Set this physical page
entry in the page directory.
The value is mk_pte_phys(__pa(vaddr),PAGE_KERNEL)
2
next page dir entry
4
1
next page global dir entry
3

3.5.3 free_area_init()

This is where the global ‘mem_map‘ array is set up and zone information is added and the ‘pgdat‘ structure. It is important that we understand the zoned buddy allocator before we proceed. So please read the zoned buddy allocator (in the next section) introduction and come back here.

The mem_map array is set up and the zone list is built here. The algorithm followed is given here.

- Calculate the total number of pages. i.e, pages in all the zones.
- Calculate the holes present in memory.
- Calculate the actual number of pages using the above two values (i.e, calculate the difference).
- Check weather mem_map is already allocated. If not allocate necessary memory (number of pages times the size of ’struct page’) from the bootmem handler.
- Set up other necessary fields in pgdat.
- Reserve all the memory by using the set_page_count and SetPageReserved calls
- For each zone present,
  - Set up the necessary fields in the zone_t structure in pgdat.
  - Set up the virtual address for all the pages to the initial value. Once allocation starts this will start changing as new allocations are made.
  - Set up the buddy system bitmaps in the zone.
- Build the allocation fall back lists using the build_zonelist call.